```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


# ========== 全局配置 ==========
DEBUG_MODE = False  # 设置为 False 以减少生产环境的日志输出

import cv2
import socket
import struct
import pickle
import sys
import time
import pyaudio
import threading
import tkinter as tk
from tkinter import ttk, simpledialog, messagebox, filedialog
from PIL import Image, ImageTk
import numpy as np
import os
from datetime import datetime
import queue
import soundfile as sf
import torch
import torchaudio
from concurrent.futures import ThreadPoolExecutor
from queue import Queue, Empty


# ========== 数据库管理类 ==========
class BiometricDatabaseManager:
    """生物识别数据库管理器（统一管理人脸和说话人数据）"""

    def __init__(self, host='localhost', port='5432', database='record',
                 user='xlevon', password='xlevon!2025'):
        """初始化数据库连接池"""
        try:
            from psycopg2 import pool
            import psycopg2.extras

            self.db_pool = pool.ThreadedConnectionPool(
                minconn=2,
                maxconn=8,
                host=host,
```

```
                port=port,
                database=database,
                user=user,
                password=password
            )
            print(f"[数据库] ✓ 连接池初始化成功 ({host}:{port}/{database})")

            # 创建表结构
            self._create_tables()

        except Exception as e:
            print(f"[数据库] ✗ 初始化失败: {e}")
            self.db_pool = None
            raise

    def _create_tables(self):
        """创建数据库表结构"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            # 人脸识别表 - 直接存储图像数据
            cur.execute("""
                CREATE TABLE IF NOT EXISTS face_embeddings (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    model_type VARCHAR(50) NOT NULL,
                    embedding BYTEA NOT NULL,
                    image_data BYTEA,
                    image_format VARCHAR(10),
                    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    UNIQUE(name, model_type)
                )
            """)

            # 说话人识别表 - 直接存储音频数据
            cur.execute("""
                CREATE TABLE IF NOT EXISTS speaker_embeddings (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    model_type VARCHAR(50) NOT NULL,
                    embedding BYTEA NOT NULL,
                    audio_data BYTEA,
```

```
                    audio_format VARCHAR(10),
                    sample_rate INTEGER,
                    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    UNIQUE(name, model_type)
                )
            """)

            # 识别日志表（复用原有设计）
            cur.execute("""
                CREATE TABLE IF NOT EXISTS face_recognition_log (
                    id SERIAL PRIMARY KEY,
                    person_name VARCHAR(255),
                    confidence FLOAT,
                    recognized_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
            """)

            conn.commit()
            cur.close()
            print("[数据库] ✓ 表结构检查完成")

        except Exception as e:
            print(f"[数据库] ⚠ 表创建失败: {e}")
            if conn:
                conn.rollback()
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def save_face_embedding(self, name, model_type, embedding, image_data=None, image_format='jpg'):
        """保存人脸特征到数据库（OpenGauss 兼容）- 直接存储图像数据"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            # 序列化 embedding
            embedding_bytes = pickle.dumps(embedding)

            # 将图像数据转换为字节（如果提供）
            image_bytes = None
            if image_data is not None:
                if isinstance(image_data, np.ndarray):
```

```
                    # 如果是 numpy 数组，编码为 JPEG
                    success, encoded = cv2.imencode(f'.{image_format}', image_data)
                    if success:
                        image_bytes = encoded.tobytes()
                elif isinstance(image_data, bytes):
                    image_bytes = image_data

            # 先尝试更新
            cur.execute("""
                UPDATE face_embeddings
                SET embedding = %s, image_data = %s, image_format = %s, registered_at =
CURRENT_TIMESTAMP
                WHERE name = %s AND model_type = %s
            """, (embedding_bytes, image_bytes, image_format, name, model_type))

            # 如果没有更新任何行，则插入
            if cur.rowcount == 0:
                cur.execute("""
                    INSERT INTO face_embeddings (name, model_type, embedding,
image_data, image_format)
                    VALUES (%s, %s, %s, %s, %s)
                """, (name, model_type, embedding_bytes, image_bytes, image_format))

            conn.commit()
            cur.close()
            return True

        except Exception as e:
            print(f"[数据库] 保存人脸特征失败: {e}")
            if conn:
                conn.rollback()
            return False
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def load_face_embeddings(self, model_type):
        """从数据库加载人脸特征"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            cur.execute("""
```

```
                SELECT name, embedding FROM face_embeddings
                WHERE model_type = %s
            """, (model_type,))

            results = cur.fetchall()
            cur.close()

            # 反序列化
            embeddings = {}
            for name, embedding_bytes in results:
                embeddings[name] = pickle.loads(embedding_bytes)

            return embeddings

        except Exception as e:
            print(f"[数据库] 加载人脸特征失败: {e}")
            return {}
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def save_speaker_embedding(self, name, model_type, embedding, audio_data=None, sample_rate=16000, audio_format='wav'):
        """保存说话人特征到数据库（OpenGauss 兼容）- 直接存储音频数据"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            # 序列化 embedding
            embedding_bytes = pickle.dumps(embedding)

            # 将音频数据转换为字节（如果提供）
            audio_bytes = None
            if audio_data is not None:
                if isinstance(audio_data, np.ndarray):
                    # 使用 soundfile 将 numpy 数组编码为 WAV 字节流
                    import io
                    audio_io = io.BytesIO()
                    sf.write(audio_io, audio_data, sample_rate, format=audio_format.upper())
                    audio_bytes = audio_io.getvalue()
                elif isinstance(audio_data, bytes):
                    audio_bytes = audio_data
```

```
                # 先尝试更新
                cur.execute("""
                    UPDATE speaker_embeddings
                    SET embedding = %s, audio_data = %s, audio_format = %s, sample_rate
= %s, registered_at = CURRENT_TIMESTAMP
                    WHERE name = %s AND model_type = %s
                """, (embedding_bytes, audio_bytes, audio_format, sample_rate, name,
model_type))

                # 如果没有更新任何行，则插入
                if cur.rowcount == 0:
                    cur.execute("""
                        INSERT INTO speaker_embeddings (name, model_type, embedding,
audio_data, audio_format, sample_rate)
                        VALUES (%s, %s, %s, %s, %s, %s)
                    """, (name, model_type, embedding_bytes, audio_bytes, audio_format,
sample_rate))

                conn.commit()
                cur.close()
                return True

            except Exception as e:
                print(f"[数据库] 保存说话人特征失败: {e}")
                if conn:
                    conn.rollback()
                return False
            finally:
                if conn:
                    self.db_pool.putconn(conn)

        def load_speaker_embeddings(self, model_type):
            """从数据库加载说话人特征"""
            conn = None
            try:
                conn = self.db_pool.getconn()
                cur = conn.cursor()

                cur.execute("""
                    SELECT name, embedding FROM speaker_embeddings
                    WHERE model_type = %s
                """, (model_type,))
```

```
            results = cur.fetchall()
            cur.close()

            # 反序列化
            embeddings = {}
            for name, embedding_bytes in results:
                embeddings[name] = pickle.loads(embedding_bytes)

            return embeddings

        except Exception as e:
            print(f"[数据库] 加载说话人特征失败: {e}")
            return {}
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def delete_face_embedding(self, name, model_type):
        """删除人脸特征"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            cur.execute("""
                DELETE FROM face_embeddings
                WHERE name = %s AND model_type = %s
            """, (name, model_type))

            deleted = cur.rowcount > 0
            conn.commit()
            cur.close()
            return deleted

        except Exception as e:
            print(f"[数据库] 删除人脸特征失败: {e}")
            if conn:
                conn.rollback()
            return False
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def delete_speaker_embedding(self, name, model_type):
```

```
        """删除说话人特征"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            cur.execute("""
                DELETE FROM speaker_embeddings
                WHERE name = %s AND model_type = %s
            """, (name, model_type))

            deleted = cur.rowcount > 0
            conn.commit()
            cur.close()
            return deleted

        except Exception as e:
            print(f"[数据库] 删除说话人特征失败: {e}")
            if conn:
                conn.rollback()
            return False
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def log_face_recognition(self, person_name, confidence):
        """记录人脸识别日志（复用原有接口）"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()
            cur.execute(
                "INSERT INTO face_recognition_log (person_name, confidence) VALUES
(%s, %s)",
                (person_name, float(confidence))
            )
            conn.commit()
            cur.close()
        except Exception as e:
            print(f"[数据库] 记录识别日志失败: {e}")
            if conn:
                conn.rollback()
        finally:
            if conn:
```

```
                self.db_pool.putconn(conn)

    def export_face_image(self, name, model_type, save_path):
        """从数据库导出人脸图像到文件"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()

            cur.execute("""
                SELECT image_data, image_format FROM face_embeddings
                WHERE name = %s AND model_type = %s
            """, (name, model_type))

            result = cur.fetchone()
            cur.close()

            if result and result[0]:
                image_bytes, image_format = result
                # 解码图像
                nparr = np.frombuffer(image_bytes, np.uint8)
                img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

                if img is not None:
                    cv2.imwrite(save_path, img)
                    print(f"[数据库] 图像已导出: {save_path}")
                    return True

            return False

        except Exception as e:
            print(f"[数据库] 导出图像失败: {e}")
            return False
        finally:
            if conn:
                self.db_pool.putconn(conn)

    def export_audio(self, name, model_type, save_path):
        """从数据库导出音频到文件"""
        conn = None
        try:
            conn = self.db_pool.getconn()
            cur = conn.cursor()
```

```
                cur.execute("""
                    SELECT audio_data, audio_format, sample_rate FROM speaker_embeddings
                    WHERE name = %s AND model_type = %s
                """, (name, model_type))

                result = cur.fetchone()
                cur.close()

                if result and result[0]:
                    audio_bytes, audio_format, sample_rate = result
                    # 将字节流写入文件
                    with open(save_path, 'wb') as f:
                        f.write(audio_bytes)
                    print(f"[数据库] 音频已导出: {save_path}")
                    return True

                return False

            except Exception as e:
                print(f"[数据库] 导出音频失败: {e}")
                return False
            finally:
                if conn:
                    self.db_pool.putconn(conn)

        def close(self):
            """关闭数据库连接池"""
            if self.db_pool:
                self.db_pool.closeall()
                print("[数据库] 连接池已关闭")


# ========== 日志辅助函数 ==========
def debug_print(message):
    """调试信息输出（仅在 DEBUG_MODE=True 时输出）"""
    if DEBUG_MODE:
        print(message)

def info_print(message):
    """重要信息输出（始终输出）"""
    print(message)

# 修复 torchaudio 兼容性
if not hasattr(torchaudio, 'list_audio_backends'):
```

```
    def _list_audio_backends():
        return ["soundfile"]
    torchaudio.list_audio_backends = _list_audio_backends

try:
    import onnxruntime as ort
    ONNX_AVAILABLE = True
except ImportError:
    ONNX_AVAILABLE = False
    print("错误: onnxruntime  未安装")
    sys.exit(1)

try:
    from speechbrain.inference.speaker import EncoderClassifier
except ImportError:
    try:
        from speechbrain.pretrained import EncoderClassifier
    except ImportError:
        EncoderClassifier = None
        print("警告: SpeechBrain  未安装，说话人识别功能不可用")


# ========== 无锁环形缓冲区 ==========
class LockFreeRingBuffer:
    """
    无锁环形缓冲区（Lock-Free Ring Buffer）
    用于音频数据的高效传输，避免锁竞争

    特性:
    - 单生产者单消费者模型（SPSC）
    - 使用原子操作避免锁
    - 固定大小，自动覆盖旧数据
    """

    def __init__(self, capacity):
        """
        初始化环形缓冲区

        Args:
            capacity: 缓冲区容量（样本数）
        """
        self.capacity = capacity
        self.buffer = np.zeros(capacity, dtype=np.int16)
        self.write_pos = 0  # 写指针
```

```
        self.read_pos = 0     # 读指针
        self.available = 0    # 可读数据量

        # 使用简单的计数而非锁（SPSC 模型下安全）
        print(f"[环形缓冲区] 初始化: {capacity} 样本 ({capacity/16000:.1f}秒)")

    def write(self, data):
        """
        写入数据（生产者调用）

        Args:
            data: numpy array (int16)
        """
        data_len = len(data)
        if data_len == 0:
            return

        # 写入数据（可能分两段）
        end_pos = self.write_pos + data_len

        if end_pos <= self.capacity:
            # 一次写入
            self.buffer[self.write_pos:end_pos] = data
        else:
            # 分两段写入（环形）
            first_part = self.capacity - self.write_pos
            self.buffer[self.write_pos:] = data[:first_part]
            self.buffer[:data_len - first_part] = data[first_part:]

        # 更新写指针
        self.write_pos = end_pos % self.capacity

        # 更新可读数据量
        self.available = min(self.available + data_len, self.capacity)

    def read(self, size):
        """
        读取数据（消费者调用）

        Args:
            size: 要读取的样本数

        Returns:
            numpy array (int16) 或 None（数据不足）
```

```
        """
        if self.available < size:
            return None

        # 读取数据（可能分两段）
        end_pos = self.read_pos + size

        if end_pos <= self.capacity:
            # 一次读取
            result = self.buffer[self.read_pos:end_pos].copy()
        else:
            # 分两段读取（环形）
            first_part = self.capacity - self.read_pos
            result = np.concatenate([
                self.buffer[self.read_pos:],
                self.buffer[:size - first_part]
            ])

        # 更新读指针
        self.read_pos = end_pos % self.capacity

        # 更新可读数据量
        self.available -= size

        return result

    def available_samples(self):
        """返回可读样本数"""
        return self.available

    def clear(self):
        """清空缓冲区"""
        self.write_pos = 0
        self.read_pos = 0
        self.available = 0


# 帧内存池管理
class FramePool:
    """
    帧内存池（Frame Memory Pool）
    预分配固定大小的帧缓冲，避免频繁分配/释放

    特性:
```

- 预分配 N 个帧缓冲
- 使用完毕后回收到池中
- 减少 GC 压力
"""

```
def __init__(self, frame_shape=(480, 640, 3), pool_size=10, dtype=np.uint8):
    """
    初始化帧池

    Args:
        frame_shape: 帧尺寸 (height, width, channels)
        pool_size: 池大小
        dtype: 数据类型
    """
    self.frame_shape = frame_shape
    self.dtype = dtype
    self.pool_size = pool_size

    # 预分配内存池
    self.pool = []
    self.available = []  # 可用帧列表

    for i in range(pool_size):
        frame = np.zeros(frame_shape, dtype=dtype)
        self.pool.append(frame)
        self.available.append(i)

    self.lock = threading.Lock()

    memory_mb = (pool_size * np.prod(frame_shape) * dtype(0).itemsize) / (1024 * 1024)
    print(f"[帧内存池] 初始化: {pool_size} 个帧, 总计 {memory_mb:.1f} MB")

def acquire(self):
    """
    获取一个空闲帧

    Returns:
        numpy array 或 None（池已空）
    """
    with self.lock:
        if len(self.available) == 0:
            return None

        idx = self.available.pop(0)
```

```
                return self.pool[idx], idx

        def release(self, idx):
            """
            释放帧回池中

            Args:
                idx: 帧索引
            """
            with self.lock:
                if idx not in self.available:
                    self.available.append(idx)

        def get_usage(self):
            """获取池使用率"""
            with self.lock:
                used = self.pool_size - len(self.available)
                return used, self.pool_size


class SpeakerRecognitionSystem:
    """说话人识别系统 - 支持 ECAPA-TDNN、ResNet34-SE 和融合模式"""

    # 模型类型常量
    MODEL_ECAPA = "ecapa"
    MODEL_RESNET = "resnet"
    MODEL_FUSION = "fusion"  # 新增：融合模式

    def __init__(self, model_type="ecapa", device="cpu", db_manager=None):
        """
        初始化说话人识别系统

        Args:
            model_type: 模型类型 ("ecapa"、"resnet" 或 "fusion")
            device: 运行设备
            db_manager: 数据库管理器实例
        """
        self.model_type = model_type
        self.device = device
        self.sample_rate = 16000
        self.min_duration = 1.0

        self.speaker_database = {}
        self.classifier = None
```

```
        self.db_manager = db_manager  # 数据库管理器

        # 初始化 Silero VAD 模型（用于人声检测）
        self.vad_model = None
        self._init_vad_model()

        # 融合模式需要加载两个模型
        if model_type == self.MODEL_FUSION:
            # 融合权重（仅融合模式需要）
            self.ecapa_weight = 0.6
            self.resnet_weight = 0.4

            self.ecapa_classifier = None
            self.resnet_classifier = None
            self.ecapa_database = {}
            self.resnet_database = {}

            # 加载两个模型
            self._init_fusion_models()
            self._load_fusion_databases()

            print(f"[说话人识别] ✓ 融合模式就绪")
            print(f"   - ECAPA-TDNN 权重: {self.ecapa_weight:.1%}")
            print(f"   - ResNet34-SE 权重: {self.resnet_weight:.1%}")

        else:
            # 单模型模式
            # 加载对应模型
            if model_type == self.MODEL_RESNET:
                self._init_resnet_model()
            else:
                self._init_ecapa_model()

            # 从数据库加载
            self.load_database()

        print(f"[说话人识别] ✓ 已加载 {len(self.speaker_database)} 个注册说话人")

    def _init_vad_model(self):
        """初始化 Silero VAD 模型（从本地加载）"""
        local_model_path = "pretrained_models/silero-vad/files/silero_vad.jit"

        if not os.path.exists(local_model_path):
            raise RuntimeError(f"VAD 模型文件不存在: {local_model_path}")
```

```
        print(f"[VAD] 正在加载模型: {local_model_path}")
        self.vad_model = torch.jit.load(local_model_path, map_location=self.device)
        self.vad_model.eval()
        print(f"[VAD] ✓ 模型加载成功 (device: {self.device})")


    def detect_speech(self, audio_chunk):
        """使用 Silero VAD 模型检测是否为人声"""
        # 转换为 float32 并归一化到 [-1, 1]
        audio_float = audio_chunk.astype(np.float32) / 32768.0
        audio_tensor = torch.from_numpy(audio_float).to(self.device)

        # 使用滑动窗口检测 (512 样本窗口, 256 样本步长)
        window_size, step_size = 512, 256
        speech_probs = []

        for i in range(0, len(audio_tensor) - window_size, step_size):
            window = audio_tensor[i:i+window_size]
            with torch.no_grad():
                prob = self.vad_model(window.unsqueeze(0), 16000).item()
                speech_probs.append(prob)

        if not speech_probs:
            return False, 0.0

        # 计算平均置信度，阈值 0.5
        avg_confidence = np.mean(speech_probs)
        return avg_confidence > 0.5, avg_confidence

    def _init_ecapa_model(self):
        """初始化 ECAPA-TDNN 模型（从本地加载）"""
        if EncoderClassifier is None:
            raise RuntimeError("SpeechBrain 未安装")

        print("[说话人识别] 正在加载 ECAPA-TDNN...")
        local_model_dir = "pretrained_models/spkrec-ecapa"

        if not os.path.exists(os.path.join(local_model_dir, "hyperparams.yaml")):
            raise RuntimeError(f"ECAPA 模型文件不存在: {local_model_dir}")

        self.classifier = EncoderClassifier.from_hparams(
            source=local_model_dir,
            savedir=local_model_dir,
```

```
            run_opts={"device": self.device}
        )
        print("[说话人识别] ✓ ECAPA-TDNN 加载成功")

    def _init_resnet_model(self):
        """初始化 ResNet34-SE 模型（从本地加载）"""
        if EncoderClassifier is None:
            raise RuntimeError("SpeechBrain 未安装")

        print("[说话人识别] 正在加载 ResNet34-SE...")
        local_model_dir = "pretrained_models/spkrec-resnet"

        if not os.path.exists(os.path.join(local_model_dir, "hyperparams.yaml")):
            raise RuntimeError(f"ResNet 模型文件不存在: {local_model_dir}")

        self.classifier = EncoderClassifier.from_hparams(
            source=local_model_dir,
            savedir=local_model_dir,
            run_opts={"device": self.device}
        )
        print("[说话人识别] ✓ ResNet34-SE 加载成功")

    def _init_fusion_models(self):
        """初始化融合模式 - 同时加载两个模型（从本地）"""
        if EncoderClassifier is None:
            raise RuntimeError("SpeechBrain 未安装")

        print("[说话人识别] 正在加载融合模式...")

        # 加载 ECAPA-TDNN
        ecapa_dir = "pretrained_models/spkrec-ecapa"
        if not os.path.exists(os.path.join(ecapa_dir, "hyperparams.yaml")):
            raise RuntimeError(f"ECAPA 模型文件不存在: {ecapa_dir}")

        print("  [1/2] 加载 ECAPA-TDNN...")
        self.ecapa_classifier = EncoderClassifier.from_hparams(
            source=ecapa_dir,
            savedir=ecapa_dir,
            run_opts={"device": self.device}
        )
        print("  ✓ ECAPA-TDNN 加载成功")

        # 加载 ResNet34-SE
        resnet_dir = "pretrained_models/spkrec-resnet"
```

```
        if not os.path.exists(os.path.join(resnet_dir, "hyperparams.yaml")):
            raise RuntimeError(f"ResNet 模型文件不存在: {resnet_dir}")

        print("   [2/2] 加载 ResNet34-SE...")
        self.resnet_classifier = EncoderClassifier.from_hparams(
            source=resnet_dir,
            savedir=resnet_dir,
            run_opts={"device": self.device}
        )
        print("   ✓ ResNet34-SE 加载成功")

        print(f"[说话人识别] ✓ 融合模式加载完成 (ECAPA:{self.ecapa_weight:.0%} + ResNet:{self.resnet_weight:.0%})")


    def _load_fusion_databases(self):
        """加载融合模式的两个数据库"""
        if not self.db_manager:
            raise RuntimeError("数据库管理器未初始化，无法加载数据")

        # 从数据库加载
        self.ecapa_database = self.db_manager.load_speaker_embeddings('ecapa')
        self.resnet_database = self.db_manager.load_speaker_embeddings('resnet')

        # 合并数据库到 speaker_database（用于统一接口，如列表显示）
        # 使用两个数据库的并集
        all_names = set(self.ecapa_database.keys()) | set(self.resnet_database.keys())
        self.speaker_database = {}
        for name in all_names:
            # 只要在任一数据库中存在就添加
            self.speaker_database[name] = {
                'ecapa': self.ecapa_database.get(name),
                'resnet': self.resnet_database.get(name)
            }

        print(f"[说话人识别] 融合数据库加载完成")
        print(f"   - ECAPA 数据库: {len(self.ecapa_database)} 人")
        print(f"   - ResNet 数据库: {len(self.resnet_database)} 人")
        print(f"   - 总共: {len(self.speaker_database)} 人")

    def load_database(self):
        """加载说话人数据库"""
        if self.model_type == self.MODEL_FUSION:
            # 融合模式已经在 _load_fusion_databases() 中加载
```

```
                pass
            else:
                if not self.db_manager:
                    raise RuntimeError("数据库管理器未初始化，无法加载数据")

                # 从数据库加载
                self.speaker_database =
self.db_manager.load_speaker_embeddings(self.model_type)

    def save_database(self):
        """保存说话人数据库"""
        if self.model_type == self.MODEL_FUSION:
            # 融合模式：保存已在 register_speaker_from_audio 中处理
            pass
        else:
            # 单模型模式：保存已在 register_speaker_from_audio 中处理
            pass

    def extract_embedding(self, audio_tensor):
        """提取说话人特征（统一接口）"""
        if audio_tensor.dim() == 1:
            audio_tensor = audio_tensor.unsqueeze(0)

        audio_tensor = audio_tensor.to(self.device)

        with torch.no_grad():
            embeddings = self.classifier.encode_batch(audio_tensor)

        return embeddings.squeeze().cpu().numpy()

    def extract_embedding_ecapa(self, audio_tensor):
        """提取 ECAPA-TDNN 特征（融合模式专用）"""
        if audio_tensor.dim() == 1:
            audio_tensor = audio_tensor.unsqueeze(0)
        audio_tensor = audio_tensor.to(self.device)
        with torch.no_grad():
            embeddings = self.ecapa_classifier.encode_batch(audio_tensor)
        return embeddings.squeeze().cpu().numpy()

    def extract_embedding_resnet(self, audio_tensor):
        """提取 ResNet34-SE 特征（融合模式专用）"""
        if audio_tensor.dim() == 1:
            audio_tensor = audio_tensor.unsqueeze(0)
        audio_tensor = audio_tensor.to(self.device)
```

```
        with torch.no_grad():
            embeddings = self.resnet_classifier.encode_batch(audio_tensor)
        return embeddings.squeeze().cpu().numpy()

    def register_speaker_from_file(self, name, audio_path):
        """从文件注册说话人"""
        try:
            # 使用 soundfile 加载
            audio, sr = sf.read(audio_path, dtype='float32')
            audio = torch.from_numpy(audio)

            # 立体声转单声道
            if audio.dim() > 1 and audio.shape[1] > 1:
                audio = torch.mean(audio, dim=1)

            if audio.dim() == 1:
                audio = audio.unsqueeze(0)

            # 重采样
            if sr != self.sample_rate:
                resampler = torchaudio.transforms.Resample(sr, self.sample_rate)
                audio = resampler(audio)

            # 提取特征
            embedding = self.extract_embedding(audio)

            # 保存
            self.speaker_database[name] = embedding
            self.save_database()

            return True, f"成功注册: {name}"
        except Exception as e:
            return False, f"注册失败: {e}"

    def _normalize_audio(self, audio_data):
        """音频预处理和归一化（与识别时保持一致）"""
        # 转换为浮点数
        audio_data = audio_data.astype(np.float32) / 32768.0

        # 保守的静音修剪（与识别时一致）
        frame_length = 400  # 25ms
        hop_length = 160    # 10ms
        energy = np.array([
            np.sum(audio_data[i:i+frame_length]**2)
```

```
                for i in range(0, len(audio_data)-frame_length, hop_length)
            ])

            if len(energy) > 0:
                # 使用相同的阈值（2%，非常保守）
                energy_threshold = np.mean(energy) * 0.02
                voiced_frames = energy > energy_threshold

                # 使用相同的保留比例（10%）
                if np.sum(voiced_frames) > len(voiced_frames) * 0.1:
                    voiced_indices = np.where(voiced_frames)[0]
                    if len(voiced_indices) > 0:
                        start_idx = voiced_indices[0] * hop_length
                        end_idx = min(voiced_indices[-1] * hop_length + frame_length,
len(audio_data))
                        audio_data = audio_data[start_idx:end_idx]

            # 温和的音量归一化（与识别时一致）
            max_amp = np.max(np.abs(audio_data))
            if max_amp > 0.1:  # 只有信号足够强才归一化
                audio_data = audio_data / max_amp * 0.8

            # 确保最小长度
            min_length = 16000  # 1 秒
            if len(audio_data) < min_length:
                audio_data = np.pad(audio_data, (0, min_length - len(audio_data)))

            return audio_data

        def _save_to_dual_databases(self, name, audio_tensor, audio_data=None):
            """保存到双模型数据库（ECAPA + ResNet）- 直接存储音频数据"""
            try:
                if not self.db_manager:
                    print(f"[说话人注册] ⚠ 数据库管理器未初始化，跳过跨模型保存")
                    return False

                # 加载必要的模块
                from speechbrain.pretrained import EncoderClassifier

                # 保存到两个模型
                models = [
                    ('ecapa', 'pretrained_models/spkrec-ecapa',
'speechbrain/spkrec-ecapa-voxceleb'),
                    ('resnet', 'pretrained_models/spkrec-resnet',
```

```
'speechbrain/spkrec-resnet-voxceleb')
                ]

                for model_name, local_dir, remote_source in models:
                    # 提取特征
                    if os.path.exists(os.path.join(local_dir, "hyperparams.yaml")):
                        classifier = EncoderClassifier.from_hparams(
                            source=local_dir,
                            savedir=local_dir,
                            run_opts={"device": self.device}
                        )
                        with torch.no_grad():
                            embedding = classifier.encode_batch(
                                audio_tensor.to(self.device)
                            ).squeeze().cpu().numpy()

                        # L2 归一化
                        embedding = embedding / (np.linalg.norm(embedding) + 1e-8)

                        # 保存到数据库（直接存储音频数据）
                        self.db_manager.save_speaker_embedding(
                            name, model_name, embedding,
                            audio_data=audio_data,
                            sample_rate=self.sample_rate
                        )

                print(f"[说话人注册] ✓ 已保存到 ECAPA 和 ResNet 数据库")
                return True
            except Exception as e:
                print(f"[说话人注册] ⚠ 跨模型保存失败: {e}")
                return False

    def register_speaker_from_audio(self, name, audio_data, save_audio=True):
        """从音频数据注册说话人（优化版 - 增强质量检查）

        Args:
            name: 说话人姓名
            audio_data: 音频数据 (numpy array, int16)
            save_audio: 是否保存音频文件到本地
        """
        try:
            print(f"\n[说话人注册] 开始注册: {name}")
            print(f"  - 原始音频长度: {len(audio_data)} 样本 ({len(audio_data)/16000:.2f}
秒)")
```

```
            # === 音频质量检查 ===
            # 1. 长度检查
            if len(audio_data) < 16000:
                return False, "❌ 音频太短！至少需要 1 秒，建议 3-5 秒"

            if len(audio_data) > 16000 * 30:
                return False, "❌ 音频太长！最长 30 秒，建议 3-5 秒"

            # 2. 能量检查（确保有足够的语音信号）
            audio_float = audio_data.astype(np.float32)
            rms = np.sqrt(np.mean(audio_float ** 2))
            print(f"   - 音频 RMS 能量: {rms:.1f}")

            if rms < 100:
                return False, f"❌ 音频信号太弱（RMS={rms:.1f}）！\n 请靠近麦克风或提高音量"

            # 3. 语音帧比例检查
            frame_size = 400
            voice_frame_count = 0
            total_frames = 0
            energy_threshold = rms * 0.5  # 动态阈值

            for i in range(0, len(audio_float) - frame_size, frame_size):
                frame = audio_float[i:i+frame_size]
                frame_rms = np.sqrt(np.mean(frame ** 2))
                if frame_rms > energy_threshold:
                    voice_frame_count += 1
                total_frames += 1

            voice_ratio = voice_frame_count / max(total_frames, 1)
            print(f"   - 语音帧比例: {voice_ratio:.1%}")

            if voice_ratio < 0.3:
                return False, f"❌ 语音内容太少（{voice_ratio:.1%}）！\n 请说话连贯，减少停顿"

            print(f"   ✓ 音频质量检查通过")

            # === 预处理音频（与识别时保持一致）===
            audio_data = self._normalize_audio(audio_data)
            print(f"   - 预处理后长度: {len(audio_data)} 样本 ({len(audio_data)/16000:.2f} 秒)")
```

```
                audio_tensor = torch.from_numpy(audio_data).unsqueeze(0)

                # 不再保存音频文件到本地，直接上传到数据库
                print(f"   - 音频数据将直接上传到数据库（不保存本地文件）")

                # 根据模式保存
                if self.model_type == self.MODEL_FUSION:
                    # 融合模式：使用已加载的两个模型
                    print(f"   - 提取 ECAPA-TDNN 特征...")
                    ecapa_embedding = self.extract_embedding_ecapa(audio_tensor)

                    print(f"   - 提取 ResNet34 特征...")
                    resnet_embedding = self.extract_embedding_resnet(audio_tensor)

                    # L2 归一化
                    ecapa_embedding = ecapa_embedding / (np.linalg.norm(ecapa_embedding) +
1e-8)
                    resnet_embedding = resnet_embedding / (np.linalg.norm(resnet_embedding) +
1e-8)

                    # 保存到数据库（直接存储音频数据）
                    if not self.db_manager:
                        raise RuntimeError("数据库管理器未初始化，无法保存数据")

                    self.db_manager.save_speaker_embedding(
                        name, 'ecapa', ecapa_embedding,
                        audio_data=audio_data,
                        sample_rate=self.sample_rate
                    )
                    self.db_manager.save_speaker_embedding(
                        name, 'resnet', resnet_embedding,
                        audio_data=audio_data,
                        sample_rate=self.sample_rate
                    )

                    # 重新加载到内存缓存
                    self._load_fusion_databases()

                    print(f"   ✓ 已同时保存到 ECAPA 和 ResNet 数据库")
                    print(f"     • ECAPA: 维度={len(ecapa_embedding)}, L2范数
={np.linalg.norm(ecapa_embedding):.3f}")
                    print(f"     • ResNet: 维度={len(resnet_embedding)}, L2范数
={np.linalg.norm(resnet_embedding):.3f}")
```

```
                return True, f"✓ 注册成功！\n 时长: {len(audio_data)/16000:.1f}秒\nRMS: {rms:.0f}\n 语音帧: {voice_ratio:.0%}"

            else:
                # 单模型模式：保存到当前模型数据库
                print(f"  - 提取 {self.model_type.upper()} 特征...")
                embedding = self.extract_embedding(audio_tensor)

                # L2 归一化
                embedding = embedding / (np.linalg.norm(embedding) + 1e-8)

                # 保存到数据库（直接存储音频数据）
                if not self.db_manager:
                    raise RuntimeError("数据库管理器未初始化，无法保存数据")

                self.db_manager.save_speaker_embedding(
                    name, self.model_type, embedding,
                    audio_data=audio_data,
                    sample_rate=self.sample_rate
                )

                # 重新加载到内存缓存
                self.load_database()

                print(f"  ✓ 已保存到 {self.model_type.upper()} 数据库")
                print(f"    • 维度={len(embedding)}, L2范数={np.linalg.norm(embedding):.3f}")

                # 尝试保存到双模型数据库（如果模型可用）
                self._save_to_dual_databases(name, audio_tensor, audio_data)

                return True, f"✓ 注册成功！\n 时长: {len(audio_data)/16000:.1f}秒\nRMS: {rms:.0f}\n 语音帧: {voice_ratio:.0%}"

        except Exception as e:
            print(f"  ✗ 注册失败: {e}")
            import traceback
            traceback.print_exc()
            return False, f"注册失败: {e}"

    def recognize_speaker(self, audio_data, threshold=None):
        """识别说话人（支持融合模式）- 优化版"""
        # 检查数据库
```

```
if self.model_type == self.MODEL_FUSION:
    if len(self.ecapa_database) == 0 and len(self.resnet_database) == 0:
        return "未知", 0.0, False
else:
    if len(self.speaker_database) == 0:
        return "未知", 0.0, False

# 根据模型类型设置默认阈值
# 注意：单模型使用距离阈值，实际比较时转换为相似度
if threshold is None:
    if self.model_type == self.MODEL_FUSION:
        threshold = 0.35  # 相似度阈值（融合模式）
    elif self.model_type == self.MODEL_RESNET:
        threshold = 0.60  # 距离阈值 → 需要相似度 ≥ 40%（ResNet 更宽松）
    else:  # ECAPA
        threshold = 0.50  # 距离阈值 → 需要相似度 ≥ 50%（ECAPA 较严格）

try:
    # 检查音频长度（至少 1 秒）
    if len(audio_data) < 16000:
        return "未知", 0.0, False

    # 归一化
    audio_data = audio_data.astype(np.float32) / 32768.0

    # 优化音频预处理：更保守的处理，保留更多语音信息
    # 1. 轻量级静音修剪（只去除首尾的明显静音）
    frame_length = 400  # 25ms
    hop_length = 160    # 10ms
    energy = np.array([
        np.sum(audio_data[i:i+frame_length]**2)
        for i in range(0, len(audio_data)-frame_length, hop_length)
    ])

    # 非常宽松的静音修剪（只去除首尾空白）
    if len(energy) > 0:
        # 使用更低的阈值，只去除极端安静的部分
        energy_threshold = np.mean(energy) * 0.02  # 降低到 2%（更保守）
        voiced_frames = energy > energy_threshold

        # 只要有 10%以上的帧有声音就保留（之前是 20%）
        if np.sum(voiced_frames) > len(voiced_frames) * 0.1:
            # 找到第一个和最后一个有声帧，保留中间所有内容
            voiced_indices = np.where(voiced_frames)[0]
```

```
                    if len(voiced_indices) > 0:
                        start_idx = voiced_indices[0] * hop_length
                        end_idx = min(voiced_indices[-1] * hop_length + frame_length,
len(audio_data))
                        audio_data = audio_data[start_idx:end_idx]

            # 2. 温和的音量归一化（避免过度放大噪音）
            max_amp = np.max(np.abs(audio_data))
            if max_amp > 0.1:  # 只有信号足够强才归一化
                audio_data = audio_data / max_amp * 0.8  # 降低到 0.8（之前 0.95 太高）

            # 3. 确保最小长度（模型需要）
            min_length = 16000  # 1 秒
            if len(audio_data) < min_length:
                # 填充零到最小长度
                audio_data = np.pad(audio_data, (0, min_length - len(audio_data)))

            # 转换为 torch tensor
            audio_tensor = torch.from_numpy(audio_data).unsqueeze(0)

            if self.model_type == self.MODEL_FUSION:
                # 融合模式：使用两个模型进行识别
                return self._recognize_fusion(audio_tensor, threshold)
            else:
                # 单模型模式
                # 提取特征
                query_embedding = self.extract_embedding(audio_tensor)

                # L2 归一化（提高鲁棒性）
                query_embedding = query_embedding / (np.linalg.norm(query_embedding) +
1e-8)

                # 比较
                best_match = None
                best_similarity = 0.0  # 改用相似度而不是距离
                all_similarities = {}  # 用于调试

                for name, db_embedding in self.speaker_database.items():
                    # L2 归一化数据库特征
                    db_embedding_norm = db_embedding / (np.linalg.norm(db_embedding)
+ 1e-8)

                    # 计算余弦相似度（已归一化，可直接点积）
                    similarity = np.dot(query_embedding, db_embedding_norm)
```

```
                    all_similarities[name] = similarity

                    if similarity > best_similarity:
                        best_similarity = similarity
                        best_match = name

                # 打印所有相似度（调试用）
                if len(all_similarities) > 0:
                    sorted_sims = sorted(all_similarities.items(), key=lambda x: x[1], reverse=True)
                    debug_print(f"    [{self.model_type.upper()}] 相似度分数: {', '.join([f'{n}={s:.3f}' for n, s in sorted_sims])}")
                    debug_print(f"    [{self.model_type.upper()}] 特征维度: {len(query_embedding)}")
                else:
                    debug_print(f"  [{self.model_type.upper()}] 数据库为空")

                # 相似度需要高于 (1 - threshold) 才算匹配
                # threshold 是距离阈值（如 0.60），则相似度需要 ≥ 0.40
                similarity_threshold = 1 - threshold
                debug_print(f"    [{self.model_type.upper()}] 阈值要求: 相似度 ≥ {similarity_threshold:.3f} (距离阈值={threshold:.3f})")
                debug_print(f"    [{self.model_type.upper()}] 最佳匹配: {best_match} = {best_similarity:.3f}")

                if best_similarity >= similarity_threshold:
                    debug_print(f"  [{self.model_type.upper()}] ✓ 识别成功")
                    return best_match, best_similarity, True
                else:
                    debug_print(f"    [{self.model_type.upper()}] ✗ 相似度不足 ({best_similarity:.3f} < {similarity_threshold:.3f})")
                    return "未知", best_similarity, False

        except Exception as e:
            print(f"[说话人识别] 错误: {e}")
            return "错误", 0.0, False

    def _recognize_fusion(self, audio_tensor, threshold):
        """融合识别：结合 ECAPA-TDNN 和 ResNet34-SE 的结果"""
        try:
            # 提取两个模型的特征
            ecapa_embedding = self.extract_embedding_ecapa(audio_tensor)
            resnet_embedding = self.extract_embedding_resnet(audio_tensor)
```

```
            # L2 归一化（归一化后可直接使用点积计算余弦相似度）
            ecapa_embedding = ecapa_embedding / (np.linalg.norm(ecapa_embedding) +
1e-8)
            resnet_embedding = resnet_embedding / (np.linalg.norm(resnet_embedding) +
1e-8)

            # 分别计算相似度
            ecapa_scores = {}
            for name, db_embedding in self.ecapa_database.items():
                # 归一化数据库特征
                db_embedding_norm = db_embedding / (np.linalg.norm(db_embedding) +
1e-8)
                # 计算余弦相似度（已归一化，直接点积）
                similarity = np.dot(ecapa_embedding, db_embedding_norm)
                ecapa_scores[name] = similarity

            resnet_scores = {}
            for name, db_embedding in self.resnet_database.items():
                # 归一化数据库特征
                db_embedding_norm = db_embedding / (np.linalg.norm(db_embedding) +
1e-8)
                # 计算余弦相似度
                similarity = np.dot(resnet_embedding, db_embedding_norm)
                resnet_scores[name] = similarity

            # 融合分数（加权平均）
            fusion_scores = {}
            all_names = set(ecapa_scores.keys()) | set(resnet_scores.keys())

            for name in all_names:
                ecapa_sim = ecapa_scores.get(name, 0)
                resnet_sim = resnet_scores.get(name, 0)
                # 加权融合
                fusion_scores[name] = self.ecapa_weight * ecapa_sim + self.resnet_weight *
resnet_sim

            if not fusion_scores:
                return "未知", 0.0, False

            # 打印详细分数（调试用）
            debug_print(f"    [FUSION] ECAPA 分数: {', '.join([f'{n}={s:.3f}' for n, s in
sorted(ecapa_scores.items(), key=lambda x: x[1], reverse=True)])}")
            debug_print(f"    [FUSION] ResNet 分数: {', '.join([f'{n}={s:.3f}' for n, s in
sorted(resnet_scores.items(), key=lambda x: x[1], reverse=True)])}")
```

```
                debug_print(f"    [FUSION] 融合分数: {', '.join([f'{n}={s:.3f}' for n, s in sorted(fusion_scores.items(), key=lambda x: x[1], reverse=True)])}")

                # 找到最佳匹配
                best_match = max(fusion_scores, key=fusion_scores.get)
                best_score = fusion_scores[best_match]

                debug_print(f"    阈值: {threshold:.3f}，最佳: {best_match}={best_score:.3f}")

                # threshold 是相似度阈值（如 0.35），直接比较
                if best_score >= threshold:
                    return best_match, best_score, True
                else:
                    return "未知", best_score, False

            except Exception as e:
                print(f"[说话人融合识别] 错误: {e}")
                return "错误", 0.0, False

        def _cosine_distance(self, vec1, vec2):
            """计算余弦距离"""
            similarity = np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2) + 1e-8)
            return 1 - similarity

        def get_registered_speakers(self):
            """获取已注册说话人列表"""
            if self.model_type == self.MODEL_FUSION:
                # 融合模式：合并两个数据库的人名
                return list(set(self.ecapa_database.keys()) | set(self.resnet_database.keys()))
            else:
                return list(self.speaker_database.keys())

        def delete_speaker(self, name):
            """删除说话人"""
            if not self.db_manager:
                raise RuntimeError("数据库管理器未初始化，无法删除数据")

            if self.model_type == self.MODEL_FUSION:
                deleted = False
                # 从数据库删除
                if self.db_manager.delete_speaker_embedding(name, 'ecapa'):
                    deleted = True
                if self.db_manager.delete_speaker_embedding(name, 'resnet'):
                    deleted = True
```

```
            # 同步内存数据库
            if deleted:
                self._load_fusion_databases()

            return deleted
        else:
            # 从数据库删除
            deleted = self.db_manager.delete_speaker_embedding(name, self.model_type)

            # 重新加载以同步内存
            if deleted:
                self.load_database()

            return deleted


class FaceRecognitionSystem:
    """人脸识别系统（简化版 - 移除硬件加速代码）"""

    MODEL_MOBILEFACENET = "mobilefacenet"
    MODEL_ARCFACE = "arcface"
    MODEL_FUSION = "fusion"

    def __init__(self, model_type=MODEL_MOBILEFACENET, db_manager=None):
        self.model_type = model_type
        print(f"[人脸识别] 初始化 - 模型: {model_type}")

        # 数据库管理器
        self.db_manager = db_manager

        self._init_detector()

        if model_type == self.MODEL_FUSION:
            self._init_fusion_models()
        else:
            self.model_path = "w600k_mbf.onnx" if model_type == self.MODEL_MOBILEFACENET else "ms1mv2_r50.onnx"
            self.face_database = {}
            self._init_recognition_model()
            self.load_database()
            print(f"[人脸识别] ✓ 已加载 {len(self.face_database)} 个注册人脸")
```

```
    def _init_detector(self):
        """初始化 YuNet 检测器"""
        yunet_path = "yunet_face_detection_640x480.onnx"
        if not os.path.exists(yunet_path):
            raise FileNotFoundError(f"YuNet 模型不存在: {yunet_path}")

        self.detector = cv2.FaceDetectorYN.create(
            yunet_path, "", (320, 320), 0.6, 0.3, 5000
        )
        print("[人脸识别] ✓ YuNet 检测器加载成功")

    def _init_fusion_models(self):
        """初始化融合模式"""
        self.mbf_model_path = "w600k_mbf.onnx"
        self.mbf_database = {}

        self.arc_model_path = "ms1mv2_r50.onnx"
        self.arc_database = {}

        # 使用 CPU 执行提供程序（简化版）
        providers = ['CPUExecutionProvider']

        sess_options = ort.SessionOptions()
        sess_options.inter_op_num_threads = 2
        sess_options.intra_op_num_threads = 2
        sess_options.graph_optimization_level =
ort.GraphOptimizationLevel.ORT_ENABLE_ALL

        # 加载 MobileFaceNet
        print(f"[人脸识别] 正在加载 MobileFaceNet 模型...")
        self.mbf_session = ort.InferenceSession(
            self.mbf_model_path,
            sess_options=sess_options,
            providers=providers
        )
        print(f"   使用提供程序: CPU")
        self.mbf_input_name = self.mbf_session.get_inputs()[0].name
        self.mbf_output_name = self.mbf_session.get_outputs()[0].name
        self.mbf_input_size = (112, 112)

        # 加载 ArcFace
        print(f"[人脸识别] 正在加载 ArcFace 模型...")
        self.arc_session = ort.InferenceSession(
            self.arc_model_path,
```

```
            sess_options=sess_options,
            providers=providers
        )
        print(f"   使用提供程序: CPU")
        self.arc_input_name = self.arc_session.get_inputs()[0].name
        self.arc_output_name = self.arc_session.get_outputs()[0].name
        self.arc_input_size = (112, 112)

        self._load_fusion_databases()

        self.mbf_weight = 0.4
        self.arc_weight = 0.6

        print(f"[人脸识别] ✓ 融合模式就绪")

    def _load_fusion_databases(self):
        """加载融合数据库"""
        if not self.db_manager:
            raise RuntimeError("数据库管理器未初始化，无法加载数据")

        # 从数据库加载
        self.mbf_database = self.db_manager.load_face_embeddings('mobilefacenet')
        self.arc_database = self.db_manager.load_face_embeddings('arcface')

    def _init_recognition_model(self):
        """初始化识别模型（简化版 - CPU 执行）"""
        # 使用 CPU 执行提供程序
        providers = ['CPUExecutionProvider']

        sess_options = ort.SessionOptions()
        sess_options.inter_op_num_threads = 2
        sess_options.intra_op_num_threads = 2
        sess_options.graph_optimization_level =
ort.GraphOptimizationLevel.ORT_ENABLE_ALL

        print(f"[人脸识别] 正在加载模型: {self.model_path}")
        self.session = ort.InferenceSession(
            self.model_path,
            sess_options=sess_options,
            providers=providers
        )

        print(f"   使用提供程序: CPU")
```

```
        self.input_name = self.session.get_inputs()[0].name
        self.output_name = self.session.get_outputs()[0].name
        self.input_size = (112, 112)

        print(f"[人脸识别] ✓ 模型加载成功")

    def detect_faces(self, frame):
        """检测人脸"""
        height, width = frame.shape[:2]
        self.detector.setInputSize((width, height))
        _, faces = self.detector.detect(frame)

        if faces is None:
            return []

        results = []
        for face in faces:
            x, y, w, h = face[:4].astype(int)
            results.append((x, y, w, h))
        return results

    def preprocess_face(self, face_image, target_size):
        """预处理人脸"""
        face_resized = cv2.resize(face_image, target_size)
        face_rgb = cv2.cvtColor(face_resized, cv2.COLOR_BGR2RGB)
        face_normalized = (face_rgb.astype(np.float32) - 127.5) / 128.0
        face_tensor = np.transpose(face_normalized, (2, 0, 1))
        face_tensor = np.expand_dims(face_tensor, axis=0).astype(np.float32)
        return face_tensor

    def _log_inference_time(self, time_ms, attr_name, label):
        """记录推理时间（通用函数）"""
        if not hasattr(self, attr_name):
            setattr(self, attr_name, [])

        times = getattr(self, attr_name)
        times.append(time_ms)

        if len(times) >= 100:
            avg_time = np.mean(times)
            print(f"[性能] {label} - 平均: {avg_time:.2f}ms")
            setattr(self, attr_name, [])

    def extract_embedding(self, face_image):
```

```
        """提取特征（单模型）"""
        start_time = time.perf_counter()
        face_tensor = self.preprocess_face(face_image, self.input_size)
        embedding = self.session.run([self.output_name], {self.input_name: face_tensor})[0]

        self._log_inference_time((time.perf_counter() - start_time) * 1000,
                                 'inference_times', '人脸识别')
        return embedding[0]

    def extract_embedding_mbf(self, face_image):
        """提取 MobileFaceNet 特征"""
        start_time = time.perf_counter()
        face_tensor = self.preprocess_face(face_image, self.mbf_input_size)
        embedding = self.mbf_session.run([self.mbf_output_name], {self.mbf_input_name:
face_tensor})[0]

        self._log_inference_time((time.perf_counter() - start_time) * 1000,
                                 'mbf_inference_times', 'MobileFaceNet')
        return embedding[0]

    def extract_embedding_arc(self, face_image):
        """提取 ArcFace 特征"""
        start_time = time.perf_counter()
        face_tensor = self.preprocess_face(face_image, self.arc_input_size)
        embedding = self.arc_session.run([self.arc_output_name], {self.arc_input_name:
face_tensor})[0]

        self._log_inference_time((time.perf_counter() - start_time) * 1000,
                                 'arc_inference_times', 'ArcFace')
        return embedding[0]

    def log_face_recognition(self, person_name, confidence):
        """记录人脸识别结果到数据库（复用统一接口）"""
        if self.db_manager:
            self.db_manager.log_face_recognition(person_name, confidence)

    def recognize_face(self, face_image):
        """识别人脸"""
        if self.model_type == self.MODEL_FUSION:
            return self._recognize_fusion(face_image)

        embedding = self.extract_embedding(face_image)

        if len(self.face_database) == 0:
```

```
            return "未知", 0.0

        best_match = None
        best_similarity = 0

        for name, db_embedding in self.face_database.items():
            similarity = np.dot(embedding, db_embedding) / (
                np.linalg.norm(embedding) * np.linalg.norm(db_embedding)
            )

            if similarity > best_similarity:
                best_similarity = similarity
                best_match = name

        if best_similarity > 0.4:
            #  记录到数据库
            self.log_face_recognition(best_match, best_similarity)
            return best_match, best_similarity
        return "未知", best_similarity

    def _recognize_fusion(self, face_image):
        """融合模式识别"""
        mbf_embedding = self.extract_embedding_mbf(face_image)
        arc_embedding = self.extract_embedding_arc(face_image)

        if len(self.mbf_database) == 0 and len(self.arc_database) == 0:
            return "未知", 0.0

        mbf_scores = {}
        arc_scores = {}

        # MobileFaceNet  比较
        for name, db_embedding in self.mbf_database.items():
            similarity = np.dot(mbf_embedding, db_embedding) / (
                np.linalg.norm(mbf_embedding) * np.linalg.norm(db_embedding)
            )
            mbf_scores[name] = similarity

        # ArcFace  比较
        for name, db_embedding in self.arc_database.items():
            similarity = np.dot(arc_embedding, db_embedding) / (
                np.linalg.norm(arc_embedding) * np.linalg.norm(db_embedding)
            )
            arc_scores[name] = similarity
```

```
        # 融合
        fusion_scores = {}
        all_names = set(mbf_scores.keys()) | set(arc_scores.keys())

        for name in all_names:
            mbf_sim = mbf_scores.get(name, 0)
            arc_sim = arc_scores.get(name, 0)
            fusion_scores[name] = self.mbf_weight * mbf_sim + self.arc_weight * arc_sim

        if not fusion_scores:
            return "未知", 0.0

        best_match = max(fusion_scores, key=fusion_scores.get)
        best_score = fusion_scores[best_match]

        if best_score > 0.4:
            # 记录到数据库
            self.log_face_recognition(best_match, best_score)
            return best_match, best_score
        return "未知", best_score

    def register_face(self, name, face_image, save_image=True):
        """注册人脸

        Args:
            name: 人员姓名
            face_image: 人脸图像
            save_image: 是否将图像数据上传到数据库（保留参数名以兼容）
        """
        # 不再保存到本地，直接上传到数据库
        print(f"[人脸注册] 图像数据将直接上传到数据库（不保存本地文件）")

        if self.model_type == self.MODEL_FUSION:
            mbf_emb = self.extract_embedding_mbf(face_image)
            arc_emb = self.extract_embedding_arc(face_image)

            if not self.db_manager:
                raise RuntimeError("数据库管理器未初始化，无法保存数据")

            # 保存到数据库（直接存储图像数据）
            self.db_manager.save_face_embedding(
                name, 'mobilefacenet', mbf_emb,
                image_data=face_image if save_image else None
```

```
                )
                self.db_manager.save_face_embedding(
                    name, 'arcface', arc_emb,
                    image_data=face_image if save_image else None
                )

                # 重新加载到内存缓存
                self._load_fusion_databases()
            else:
                embedding = self.extract_embedding(face_image)

                if not self.db_manager:
                    raise RuntimeError("数据库管理器未初始化，无法保存数据")

                # 保存到数据库（直接存储图像数据）
                self.db_manager.save_face_embedding(
                    name, self.model_type, embedding,
                    image_data=face_image if save_image else None
                )

                # 重新加载到内存缓存
                self.load_database()

            return True

        def load_database(self):
            """加载数据库"""
            if not self.db_manager:
                raise RuntimeError("数据库管理器未初始化，无法加载数据")

            # 从数据库加载
            self.face_database = self.db_manager.load_face_embeddings(self.model_type)

        def get_database_count(self):
            """获取数据库人数"""
            if self.model_type == self.MODEL_FUSION:
                return max(len(self.mbf_database), len(self.arc_database))
            return len(self.face_database)


    class MultimodalBiometricSystem:
        """多模态生物识别系统 UI"""

        def __init__(self, host='0.0.0.0', video_port=9999, audio_port=9998,
```

```
                db_host='192.168.240.6', db_port='7654', db_name='record',
                db_user='xlevon', db_password='xlevon!2025'):
self.root = tk.Tk()
self.root.title("多模态生物识别系统 - 人脸 + 说话人")
self.root.geometry("1600x900")

# 初始化数据库管理器（必须成功）
self.db_manager = BiometricDatabaseManager(
    host=db_host,
    port=db_port,
    database=db_name,
    user=db_user,
    password=db_password
)
print("[系统] ✓ 数据库管理器初始化成功")

# 网络传输参数
self.host = host
self.video_port = video_port
self.audio_port = audio_port
self.video_server = None
self.audio_server = None
self.video_client = None
self.audio_client = None
self.video_send_thread = None
self.audio_send_thread = None

# 系统状态
self.camera_running = False
self.server_running = False
self.recognition_enabled = False
self.speaker_recognition_enabled = False

# 相机和音频
self.cap = None
self.audio = None
self.current_frame = None

# 识别系统
self.face_recognition = None
self.speaker_recognition = None

# UI 变量（在 setup_ui 之前初始化）
self.model_var = None
```

```
self.recognition_var = None
self.speaker_recognition_var = None
self.speaker_model_var = None  # 新增：说话人识别模型选择
self.multimodal_fusion_var = None  # 新增：多模态融合开关
self.multimodal_fusion_enabled = False  # 多模态融合状态

# 动态线程池配置
cpu_count = os.cpu_count() or 4
print(f"[系统信息] 检测到 {cpu_count} 个 CPU 核心")

# 智能分配线程池大小
if cpu_count <= 4:
    worker_count = max(2, cpu_count - 1)
else:
    worker_count = cpu_count - 2

print(f"[线程池] 配置 {worker_count} 个工作线程")
self.thread_pool = ThreadPoolExecutor(
    max_workers=worker_count,
    thread_name_prefix="BiometricWorker"
)

# 动态队列大小配置
# 根据内存大小动态调整队列
# 假设每帧 640x480x3 ≈ 1MB
if cpu_count <= 4:  # 嵌入式设备
    frame_queue_size = 3  # 从 2 增加到 3,减少丢帧
    audio_queue_size = 20  # 从 10 增加到 20,音频优先级高
else:  # 高性能设备
    frame_queue_size = 5
    audio_queue_size = 30

print(f"[队列配置] 视频队列: {frame_queue_size}, 音频队列: {audio_queue_size}")
self.frame_queue = Queue(maxsize=frame_queue_size)
self.audio_queue = Queue(maxsize=audio_queue_size)

# 线程优先级标记（用于动态调度）
self.thread_priorities = {
    'camera': 'high',          # 相机采集 - 最高优先级
    'audio': 'high',           # 音频采集 - 最高优先级
    'recognition': 'medium',   # 识别计算 - 中等优先级
    'network': 'medium',     # 网络传输 - 中等优先级
    'ui': 'low'                # UI 更新 - 最低优先级
}
```

```
        # 线程
        self.camera_thread = None
        self.recognition_thread = None
        self.speaker_thread = None
        self.audio_capture_thread = None
        self.video_encoder_thread = None  # 新增：视频编码线程

        # 线程停止事件（优化资源释放）
        self.stop_event = threading.Event()

        # 视频传输队列（解耦设计）
        # 编码队列: 原始帧 → 编码线程
        self.encode_queue = Queue(maxsize=frame_queue_size)
        # 发送队列: 编码后数据 → 网络线程
        self.send_queue = Queue(maxsize=frame_queue_size)

        print(f"[视频管道] 编码队列: {frame_queue_size}, 发送队列: {frame_queue_size}")

        # 音频采集（环形缓冲区设计）
        # 使用环形缓冲区替代列表+锁
        audio_buffer_seconds = 10  # 10 秒缓冲
        audio_buffer_capacity = 16000 * audio_buffer_seconds
        self.speaker_audio_ring_buffer = LockFreeRingBuffer(audio_buffer_capacity)

        print(f"[音频管道] 环形缓冲区: {audio_buffer_capacity} 样本 ({audio_buffer_seconds}秒)")

        self.speaker_recognition_interval = 3.0  # 每 3 秒识别一次
        self.last_speaker_result = ("未知", 0.0)

        # 人脸识别优化
        self.frame_counter = 0
        self.recognition_frame_skip = 3  # 每 3 帧识别一次
        self.last_face_results = {}
        self.face_result_timeout = 1.0

        # 异步识别队列
        self.recognition_queue = []
        self.recognition_results = {}
        self.recognition_lock = threading.Lock()

        # 录音相关
        self.is_recording = False
```

```
        self.recorded_audio = []

        # 检测系统信息（优化：仅检测一次）
        self._init_platform_info()

    def _init_platform_info(self):
        """初始化平台信息（仅执行一次）"""
        import platform
        self.system_name = platform.system()

        # Windows 线程优先级相关
        if self.system_name == 'Windows':
            try:
                import ctypes
                self.kernel32 = ctypes.windll.kernel32
                self.has_windows_priority = True
            except:
                self.has_windows_priority = False
        else:
            self.has_windows_priority = False

    def set_thread_priority(self, priority_level):
        """设置线程优先级（优化版：避免重复导入）"""
        if not hasattr(self, 'system_name'):
            return  # 平台信息未初始化

        try:
            if self.system_name == 'Linux':
                nice_values = {'high': -10, 'medium': 0, 'low': 10}
                nice_value = nice_values.get(priority_level, 0)
                try:
                    os.nice(nice_value)
                except (PermissionError, AttributeError):
                    pass

            elif self.system_name == 'Windows' and self.has_windows_priority:
                priorities = {'high': 2, 'medium': 0, 'low': -1}
                priority = priorities.get(priority_level, 0)
                try:
                    handle = self.kernel32.GetCurrentThread()
                    self.kernel32.SetThreadPriority(handle, priority)
                except:
                    pass
        except:
```

```
            pass

    def setup_camera(self):
        """初始化摄像头"""
        try:
            self.cap = cv2.VideoCapture(0)
            if not self.cap.isOpened():
                messagebox.showerror("错误", "无法打开摄像头")
                return False

            self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
            self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
            self.cap.set(cv2.CAP_PROP_FPS, 30)

            print("✓ 摄像头初始化成功")
            return True
        except Exception as e:
            messagebox.showerror("错误", f"摄像头初始化失败: {e}")
            return False

    def setup_audio(self):
        """初始化音频（优化用于香橙派/openEuler 等嵌入式设备）"""
        try:
            self.audio = pyaudio.PyAudio()

            # 获取默认输入设备信息
            default_input = self.audio.get_default_input_device_info()
            device_index = default_input['index']
            device_name = default_input['name']

            print("=" * 60)
            print("    音频设备信息")
            print("=" * 60)
            print(f"设备索引: {device_index}")
            print(f"设备名称: {device_name}")
            print(f"输入声道: {default_input['maxInputChannels']}")
            print(f"默认采样率: {int(default_input['defaultSampleRate'])} Hz")
            print("=" * 60)

            # 列出所有可用的音频输入设备
            print("\n 可用的音频输入设备:")
            print("-" * 60)
            device_count = self.audio.get_device_count()
            for i in range(device_count):
```

```
                    info = self.audio.get_device_info_by_index(i)
                    if info['maxInputChannels'] > 0:  # 只显示输入设备
                        is_default = " [默认]" if i == device_index else ""
                        print(f"  [{i}] {info['name']}{is_default}")
                        print(f"      声道: {info['maxInputChannels']}, "
                              f"采样率: {int(info['defaultSampleRate'])} Hz")
                print("-" * 60)
                print()

                # 针对嵌入式设备（香橙派/openEuler）的优化配置
                # 增加缓冲区大小，减少 ALSA 断言失败的可能性
                frames_per_buffer = 2048  # 从 1024 增加到 2048

                self.audio_stream = self.audio.open(
                    format=pyaudio.paInt16,
                    channels=1,
                    rate=16000,
                    input=True,
                    input_device_index=device_index,
                    frames_per_buffer=frames_per_buffer,
                    # 关键：添加流配置以防止 ALSA 断言失败
                    stream_callback=None,  # 不使用回调模式，使用阻塞模式更稳定
                )

                # 给音频流一些预热时间（嵌入式设备需要）
                time.sleep(0.2)

                print(f"✓ 音频初始化成功 (使用设备: {device_name})")
                print(f"  - 缓冲区大小: {frames_per_buffer} 帧")
                print(f"  - 缓冲时长: {frames_per_buffer/16000*1000:.1f} ms")
                # 保存设备名称，稍后在 UI 中显示
                self.audio_device_name = device_name
                return True
            except Exception as e:
                messagebox.showerror("错误", f"音频初始化失败: {e}")
                return False

    def init_face_recognition(self):
        """初始化人脸识别"""
        try:
            model_type = self.model_var.get() if self.model_var else "fusion"
            self.face_recognition = FaceRecognitionSystem(model_type,
db_manager=self.db_manager)
            print("✓ 人脸识别系统初始化成功")
```

```
                return True
            except Exception as e:
                messagebox.showerror("错误", f"人脸识别初始化失败: {e}")
                return False

    def init_speaker_recognition(self, model_type=None):
        """初始化说话人识别"""
        try:
            if EncoderClassifier is None:
                raise RuntimeError("说话人识别模型不可用")

            # 获取选择的模型类型
            if model_type is None:
                model_type = self.speaker_model_var.get() if self.speaker_model_var else "ecapa"

            self.speaker_recognition = SpeakerRecognitionSystem(
                model_type=model_type,
                db_manager=self.db_manager
            )
            print(f"✓ 说话人识别系统初始化成功 - 模型: {model_type.upper()}")
            return True
        except Exception as e:
            messagebox.showwarning("警告", f"说话人识别初始化失败: {e}\n 说话人识别功能不可用")
            return False

    def switch_speaker_model(self):
        """切换说话人识别模型"""
        new_model = self.speaker_model_var.get()

        if self.speaker_recognition and self.speaker_recognition.model_type == new_model:
            self.log_info(f"已经在使用 {new_model.upper()} 模型")
            return

        self.log_info(f"正在切换到 {new_model.upper()} 模型...")

        # 检查音频流状态
        audio_stream_was_active = False
        if self.audio_stream:
            try:
                audio_stream_was_active = self.audio_stream.is_active()
            except:
                audio_stream_was_active = False
```

```
            # 临时禁用识别
            was_enabled = self.speaker_recognition_enabled
            self.speaker_recognition_enabled = False

            # 短暂等待，确保识别线程不再使用旧模型
            time.sleep(0.2)

            # 重新初始化
            if self.init_speaker_recognition(model_type=new_model):
                self.log_info(f"✓ 已切换到 {new_model.upper()} 模型")
                self.update_speaker_list()

                # 恢复识别状态
                if was_enabled:
                    self.speaker_recognition_enabled = True
            else:
                self.log_info(f"✗ 切换失败，保持原模型")
                # 恢复识别状态
                self.speaker_recognition_enabled = was_enabled

            # 确保音频流仍然打开（如果之前是打开的）
            if audio_stream_was_active and (not self.audio_stream or not
self.audio_stream.is_active()):
                self.log_info("⚠ 音频流意外关闭，正在重新打开...")
                self.setup_audio()

        def camera_loop(self):
            """
            相机采集线程（任务 2 优化版）
            职责：只负责从摄像头读取帧，不做任何处理
            优先级：HIGH（数据源优先）
            """
            # 设置线程优先级
            self.set_thread_priority('high')
            print("[相机采集] 线程已启动 (优先级: HIGH)")
            print("           职责: 仅采集，不编码/传输")

            frame_count = 0
            last_fps_time = time.time()
            dropped_frames = 0

            while self.camera_running and not self.stop_event.is_set():
                try:
```

```
ret, frame = self.cap.read()
if not ret:
    print("[相机采集] ⚠ 读取帧失败")
    time.sleep(0.01)
    continue

frame_count += 1

# 分发到不同队列

# 1. UI 显示用：直接引用（不 copy）
self.current_frame = frame

# 2. 网络传输用：放入编码队列
if self.server_running:
    try:
        # 如果编码队列满，移除旧帧（保持实时性）
        if self.encode_queue.full():
            try:
                self.encode_queue.get_nowait()
                dropped_frames += 1
            except:
                pass
        # 放入新帧（编码线程会 copy）
        self.encode_queue.put_nowait(frame)
    except:
        pass

# 3. 识别用：保留原有逻辑（向后兼容）
try:
    if self.frame_queue.full():
        try:
            self.frame_queue.get_nowait()
        except:
            pass
    self.frame_queue.put_nowait(frame)
except:
    pass

# FPS 统计
if frame_count % 30 == 0:
    elapsed = time.time() - last_fps_time
    fps = 30 / elapsed if elapsed > 0 else 0
```

```
                    if frame_count % 300 == 0:  # 每 10 秒输出一次
                        print(f"[相机采集] FPS: {fps:.1f}, "
                              f"编码队列: {self.encode_queue.qsize()}/{self.encode_queue.maxsize}, "
                              f"丢帧: {dropped_frames}")
                        dropped_frames = 0

                    last_fps_time = time.time()

                # 适度 sleep（1ms）
                time.sleep(0.001)

            except Exception as e:
                print(f"[相机采集] 错误: {e}")
                time.sleep(0.1)

        print(f"[相机采集] 已停止（总计: {frame_count} 帧）")

    def video_encoder_loop(self):
        """
        视频编码线程（任务 2 新增）
        职责：从编码队列取帧 → JPEG 编码 → 放入发送队列
        优先级：MEDIUM
        """
        # 设置线程优先级
        self.set_thread_priority('medium')
        print("[视频编码] 线程已启动 (优先级: MEDIUM)")
        print("  职责: JPEG 编码")

        # 编码参数（嵌入式设备优化）
        cpu_count = os.cpu_count() or 4
        jpeg_quality = 65 if cpu_count <= 4 else 75

        encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), jpeg_quality]
        print(f"  JPEG 质量: {jpeg_quality}")

        encode_count = 0
        total_encode_time = 0

        while self.camera_running and not self.stop_event.is_set():
            try:
                # 从编码队列获取原始帧
                try:
                    frame = self.encode_queue.get(timeout=0.1)
```

```
                except Empty:
                    continue

                # JPEG 编码
                start_time = time.time()
                result, encoded_frame = cv2.imencode('.jpg', frame, encode_param)
                encode_time = (time.time() - start_time) * 1000

                if not result:
                    print("[视频编码] ⚠ 编码失败")
                    continue

                # 序列化数据
                data = pickle.dumps(encoded_frame)

                # 放入发送队列
                try:
                    # 如果发送队列满，移除旧数据
                    if self.send_queue.full():
                        try:
                            self.send_queue.get_nowait()
                        except:
                            pass
                    self.send_queue.put_nowait(data)
                except:
                    pass

                # 统计
                encode_count += 1
                total_encode_time += encode_time

                # 定期输出性能
                if encode_count % 300 == 0:
                    avg_time = total_encode_time / encode_count
                    print(f"[视频编码] 平均: {avg_time:.1f}ms, 队列: "
                          f"输入 {self.encode_queue.qsize()} → 输出
{self.send_queue.qsize()}")

            except Exception as e:
                print(f"[视频编码] 错误: {e}")
                time.sleep(0.1)

        if encode_count > 0:
            avg_time = total_encode_time / encode_count
```

```
            print(f"[视频编码] 已停止（总计: {encode_count} 帧, 平均: {avg_time:.1f}ms）
")
        else:
            print("[视频编码] 已停止")

    def _async_recognition_worker(self):
        """
        异步识别工作线程（优化版）
        职责：从队列取出人脸图像并识别
        优先级：MEDIUM
        """
        # 设置线程优先级
        self.set_thread_priority('medium')
        print("[人脸识别] 异步识别线程已启动 (优先级: MEDIUM)")

        recognition_count = 0
        total_inference_time = 0

        while self.camera_running and not self.stop_event.is_set():
            try:
                # 获取待识别的人脸
                with self.recognition_lock:
                    if len(self.recognition_queue) == 0:
                        queue_empty = True
                    else:
                        queue_empty = False
                        face_data = self.recognition_queue.pop(0)

                # 队列为空，等待
                if queue_empty:
                    time.sleep(0.02)
                    continue

                # 执行识别（在后台线程，不阻塞 UI）
                face_img, bbox, frame_id = face_data

                start_time = time.time()
                name, similarity = self.face_recognition.recognize_face(face_img)
                inference_time = (time.time() - start_time) * 1000  # 转换为毫秒

                # 统计识别性能
                recognition_count += 1
                total_inference_time += inference_time
```

```
                    # 存储结果
                    with self.recognition_lock:
                        self.recognition_results[bbox] = (name, similarity, time.time(), inference_time)

                    # 定期输出识别信息和平均性能
                    if recognition_count % 30 == 0:
                        avg_time = total_inference_time / recognition_count
                        print(f"[识别] 平均: {avg_time:.1f}ms | 当前: {inference_time:.1f}ms | {name} ({similarity:.1%})")

                except Exception as e:
                    print(f"[异步识别] 错误: {e}")
                    import traceback
                    traceback.print_exc()
                    time.sleep(0.1)

            # 输出最终统计
            if recognition_count > 0:
                avg_time = total_inference_time / recognition_count
                print(f"[人脸识别] 总计识别 {recognition_count} 次，平均延迟: {avg_time:.1f}ms")
            print("[人脸识别] 异步识别线程已停止")

        def setup_network_server(self):
            """设置网络服务器"""
            try:
                # 视频服务器
                self.video_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.video_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.video_server.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, 65536)
                self.video_server.bind((self.host, self.video_port))
                self.video_server.listen(5)
                self.log_info(f"✓ 视频服务器: {self.host}:{self.video_port}")

                # 音频服务器
                self.audio_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.audio_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.audio_server.bind((self.host, self.audio_port))
                self.audio_server.listen(5)
                self.log_info(f"✓ 音频服务器: {self.host}:{self.audio_port}")

                return True
            except Exception as e:
```

```
            self.log_info(f"✗ 网络服务器启动失败: {e}")
            return False

    def accept_clients(self):
        """等待客户端连接（在独立线程中运行）"""
        try:
            self.log_info("等待客户端连接...")

            # 接受视频连接
            self.video_client, video_addr = self.video_server.accept()
            self.log_info(f"✓ 视频客户端已连接: {video_addr}")

            # 接受音频连接
            self.audio_client, audio_addr = self.audio_server.accept()
            self.log_info(f"✓ 音频客户端已连接: {audio_addr}")

            self.server_running = True

            # 启动编码线程
            # 启动视频编码线程（新增）
            self.video_encoder_thread = threading.Thread(target=self.video_encoder_loop,
daemon=True)
            self.video_encoder_thread.start()
            self.log_info("✓ 视频编码线程已启动")

            # 启动视频发送线程
            self.video_send_thread = threading.Thread(target=self.send_video_loop,
daemon=True)
            self.video_send_thread.start()

            # 音频由 send_audio_loop 统一处理，不需要 audio_capture_thread
            self.audio_send_thread = threading.Thread(target=self.send_audio_loop,
daemon=True)
            self.audio_send_thread.start()

        except Exception as e:
            self.log_info(f"✗ 客户端连接失败: {e}")

    def send_video_loop(self):
        """
        视频网络发送线程（任务 2 优化版）
        职责：从发送队列获取编码后的数据 → 网络传输
        优先级：MEDIUM
        """
```

```
        # 设置线程优先级
        self.set_thread_priority('medium')
        self.log_info("[视频发送] 线程已启动 (优先级: MEDIUM)")
        self.log_info("                职责: 仅网络传输")

        send_count = 0
        total_send_time = 0
        bytes_sent = 0

        while self.camera_running and self.server_running and not self.stop_event.is_set():
            try:
                # 从发送队列获取编码后的数据
                try:
                    data = self.send_queue.get(timeout=0.5)
                except Empty:
                    continue

                # 网络发送
                start_time = time.time()
                message_size = struct.pack("<L", len(data))
                self.video_client.sendall(message_size + data)
                send_time = (time.time() - start_time) * 1000

                # 统计
                send_count += 1
                total_send_time += send_time
                bytes_sent += len(data)

                # 定期输出性能
                if send_count % 300 == 0:
                    avg_time = total_send_time / send_count
                    bandwidth = (bytes_sent / 1024 / 1024) / 10  # MB/s (每 10 秒)
                    self.log_info(f"[视频发送] 平均: {avg_time:.1f}ms, "
                                  f"带宽: {bandwidth:.2f} MB/s, 队列: {self.send_queue.qsize()}")
                    bytes_sent = 0

            except Exception as e:
                if self.camera_running:
                    self.log_info(f"[视频发送] 错误: {e}")
                break

        if send_count > 0:
            avg_time = total_send_time / send_count
```

```
            self.log_info(f"[视频发送] 已停止（总计：{send_count} 帧，平均：{avg_time:.1f}ms）")
        else:
            self.log_info("[视频发送] 已停止")

    def send_audio_loop(self):
        """
        音频发送循环（任务 3 优化版）
        职责：采集音频 → 环形缓冲区 → 网络发送
        优先级：HIGH
        """
        # 设置线程优先级
        self.set_thread_priority('high')
        self.log_info("[音频发送] 线程已启动 (优先级: HIGH)")
        self.log_info("                使用无锁环形缓冲区")

        # 使用已有的 audio_stream 而不是创建新的
        chunk_size = 2048
        sample_count = 0

        try:
            while self.camera_running and self.server_running and not self.stop_event.is_set():
                try:
                    # 读取音频数据
                    audio_data = self.audio_stream.read(chunk_size, exception_on_overflow=False)
                    audio_array = np.frombuffer(audio_data, dtype=np.int16)

                    # 使用环形缓冲区（无锁）
                    # 写入环形缓冲区（无锁操作）
                    self.speaker_audio_ring_buffer.write(audio_array)

                    # 发送到网络
                    size = struct.pack("<L", len(audio_data))
                    self.audio_client.sendall(size + audio_data)

                    # 统计信息
                    sample_count += len(audio_array)
                    if sample_count >= 32000:  # 每 2 秒输出一次
                        available = self.speaker_audio_ring_buffer.available_samples()
                        print(f"[音频发送] 环形缓冲区：{available} 样本 ({available/16000:.1f}秒)")
                        sample_count = 0
```

```
                except IOError as e:
                    if e.errno in [-9981, -9980]:
                        time.sleep(0.05)
                        continue
                    break
                except Exception as e:
                    if self.camera_running:
                        self.log_info(f"[音频发送] 错误: {e}")
                    break

        except Exception as e:
            self.log_info(f"[音频发送] 流错误: {e}")

        self.log_info("[音频发送] 线程已停止")

    def toggle_network_server(self):
        """切换网络服务器开关"""
        if not self.server_running:
            # 启动服务器
            if self.setup_network_server():
                # 在独立线程中等待连接
                threading.Thread(target=self.accept_clients, daemon=True).start()
                self.network_button.config(text="停止服务器", bg='#F44336')
            else:
                messagebox.showerror("错误", "无法启动网络服务器")
        else:
            # 停止服务器
            self.server_running = False

            # 清空识别队列（防止积压）
            with self.recognition_lock:
                self.recognition_queue.clear()

            try:
                if self.video_client:
                    self.video_client.close()
                if self.audio_client:
                    self.audio_client.close()
                if self.video_server:
                    self.video_server.close()
                if self.audio_server:
                    self.audio_server.close()
            except:
                pass
```

```
            self.log_info("✓ 网络服务器已停止")
            self.network_button.config(text="启动网络服务器", bg='#4CAF50')

    def _check_audio_stream_status(self):
        """检查音频流状态（提取为独立函数）"""
        if not self.audio_stream:
            return False, "音频流对象不存在"

        try:
            if not self.audio_stream.is_active():
                return False, "音频流未激活"
        except:
            return False, "无法检查音频流状态"

        return True, "正常"

    def _restart_audio_stream(self):
        """重启音频流（提取为独立函数）"""
        try:
            self.audio_stream.stop_stream()
            time.sleep(0.1)
            self.audio_stream.start_stream()
            time.sleep(0.1)
            return True
        except:
            return False

    def audio_capture_loop(self):
        """
        音频采集循环（优化版：简化错误处理）
        职责：持续收集音频到缓冲区
        优先级：HIGH
        """
        self.set_thread_priority('high')
        print("[音频采集] 线程已启动 (优先级: HIGH)")

        chunk_size = 2048
        sample_count = 0
        error_count = 0
        max_errors = 10

        while self.camera_running and not self.stop_event.is_set():
            # 如果网络服务器正在运行，暂停此循环（避免冲突）
```

```
                if self.server_running:
                    print("[音频采集] 网络服务器运行中，本线程暂停...")
                    while self.server_running and self.camera_running:
                        time.sleep(0.5)
                    print("[音频采集] 恢复采集...")
                    continue

                try:
                    # 检查音频流状态
                    status_ok, status_msg = self._check_audio_stream_status()
                    if not status_ok:
                        error_count += 1
                        if error_count > max_errors:
                            print(f"[音频采集] {status_msg}，退出线程")
                            break
                        time.sleep(0.5)
                        continue

                    error_count = 0  # 重置错误计数

                    # 读取音频数据
                    audio_data = self.audio_stream.read(chunk_size,
exception_on_overflow=False)
                    audio_array = np.frombuffer(audio_data, dtype=np.int16)

                    # 添加到环形缓冲区
                    self.speaker_audio_ring_buffer.write(audio_array)

                    # 定期输出统计信息
                    sample_count += len(audio_array)
                    if sample_count >= 16000:  # 每秒输出一次
                        available = self.speaker_audio_ring_buffer.available_samples()
                        debug_print(f"[音频采集] 缓冲区：{available} 样本
({available/16000:.1f}秒)")
                        sample_count = 0

                except IOError as e:
                    # ALSA 缓冲区错误
                    error_count += 1
                    if error_count > max_errors:
                        print(f"[音频采集] IOError 过多，尝试重启音频流...")
                        if self._restart_audio_stream():
                            error_count = 0
                            print("[音频采集] ✓ 音频流已重启")
```

```
                    else:
                        print("[音频采集] ✗ 重启失败，退出线程")
                        break
                time.sleep(0.05)

            except OSError as e:
                # 音频流关闭错误
                if "Stream closed" in str(e) or (hasattr(e, 'errno') and e.errno == -9988):
                    print("[音频采集] 音频流已关闭，线程退出")
                    break
                error_count += 1
                if error_count > max_errors:
                    print("[音频采集] OSError 过多，线程退出")
                    break
                time.sleep(0.1)

            except Exception as e:
                print(f"[音频采集] 未知错误: {e}")
                error_count += 1
                if error_count > max_errors:
                    print("[音频采集] 错误过多，线程退出")
                    break
                time.sleep(0.1)

        print("[音频采集] 线程已退出")

    def speaker_recognition_loop(self):
        """
        说话人识别循环（优化版 - 累积更长音频）
        职责：定期从音频缓冲区识别说话人
        优先级：MEDIUM
        """
        # 设置线程优先级
        self.set_thread_priority('medium')

        # 关键修复：增加缓冲区到 3-5 秒，确保有足够的语音特征
        buffer_size = int(16000 * 4.0)  # 4 秒（增加缓冲）
        min_buffer_size = int(16000 * 2.5)  # 最小 2.5 秒

        print(f"[说话人识别] 识别线程已启动 (优先级: MEDIUM)")
        print(f"              缓冲区: {buffer_size} 样本 (4.0 秒), 最小: {min_buffer_size}
样本 (2.5 秒)")

        last_recognition_time = 0
```

```
recognition_cooldown = 1.0  # 冷却时间：1.0 秒
recognition_count = 0
total_inference_time = 0

# 噪音基线自适应
noise_baseline = None
noise_update_counter = 0

while self.camera_running and not self.stop_event.is_set():
    if not self.speaker_recognition_enabled or self.speaker_recognition is None:
        time.sleep(0.1)
        continue

    try:
        current_time = time.time()

        # 检查冷却时间
        if current_time - last_recognition_time < recognition_cooldown:
            time.sleep(0.1)
            continue

        # 从环形缓冲区读取（无锁）
        # 检查可用样本数
        available = self.speaker_audio_ring_buffer.available_samples()

        # 优先等待完整的 4 秒音频（修复显示 2.6 秒问题）
        if available >= buffer_size:
            # 读取标准缓冲区大小（4 秒）
            audio_chunk = self.speaker_audio_ring_buffer.read(buffer_size)
        elif available >= min_buffer_size:
            # 音频不够 4 秒但超过 2.5 秒
            # 检查是否接近 4 秒（3.5 秒以上就用）
            if available >= int(16000 * 3.5):
                audio_chunk = self.speaker_audio_ring_buffer.read(available)
            else:
                # 还不够，继续等待
                time.sleep(0.2)
                continue
        else:
            # 等待更多音频数据
            time.sleep(0.1)
            continue

        if audio_chunk is None:
```

```
                    time.sleep(0.1)
                    continue

                # === 使用 Silero VAD 模型进行人声检测 ===
                is_speech, vad_confidence = self.speaker_recognition.detect_speech(audio_chunk)

                if not is_speech:
                    print(f"[说话人识别] ✗ VAD 模型判断为非人声 (置信度={vad_confidence:.2%}), 跳过")
                    time.sleep(0.1)
                    continue

                # 额外的基础检查（能量过低直接跳过）
                rms = np.sqrt(np.mean(audio_chunk.astype(np.float32) ** 2))
                if rms < 30.0:
                    print(f"[说话人识别] ✗ 音量过低 (RMS={rms:.1f}), 跳过")
                    time.sleep(0.1)
                    continue

                # 执行识别
                print(f"[说话人识别] ✓ VAD 通过 ({len(audio_chunk)/16000:.1f}秒音频, 模型置信度={vad_confidence:.2%}, RMS={rms:.1f})")
                start_time = time.time()

                name, similarity, success = self.speaker_recognition.recognize_speaker(audio_chunk)

                inference_time = (time.time() - start_time) * 1000
                recognition_count += 1
                total_inference_time += inference_time

                print(f"[说话人识别] 结果: {name}, 相似度: {similarity:.2%}, 成功: {success}, 耗时: {inference_time:.0f}ms")

                # 定期输出平均性能
                if recognition_count % 10 == 0:
                    avg_time = total_inference_time / recognition_count
                    print(f"[说话人识别] 平均推理时间: {avg_time:.1f}ms")

                # 更新结果
                self.last_speaker_result = (name, similarity)
                self.root.after(0, lambda n=name, s=similarity, su=success: self.update_speaker_result(n, s, su))
```

```
                    # 更新上次识别时间
                    last_recognition_time = current_time

                    # 短暂休眠，避免 CPU 占用过高
                    time.sleep(0.1)

                except Exception as e:
                    print(f"[说话人识别] 错误: {e}")
                    import traceback
                    traceback.print_exc()
                    time.sleep(1.0)

            # 输出最终统计
            if recognition_count > 0:
                avg_time = total_inference_time / recognition_count
                print(f"[说话人识别] 总计识别 {recognition_count} 次，平均延迟: {avg_time:.1f}ms")
            print("[说话人识别] 识别线程已停止")


        def update_speaker_result(self, name, similarity, success):
            """更新说话人识别结果"""
            if success:
                # ECAPA-TDNN 置信度评级
                if similarity >= 0.70:
                    confidence_level = "极高"
                    color = 'dark green'
                elif similarity >= 0.55:
                    confidence_level = "高"
                    color = 'green'
                elif similarity >= 0.45:
                    confidence_level = "中等"
                    color = 'orange'
                else:
                    confidence_level = "较低"
                    color = 'dark orange'

                # 根据模型类型显示不同标签（修复 Fusion 显示问题）
                if self.speaker_recognition.model_type == "resnet":
                    model_label = "RESNET"
                elif self.speaker_recognition.model_type == "fusion":
                    model_label = "FUSION"
                else:
```

```
                model_label = "ECAPA"

            result_text = f"✓ [{model_label}] {name} (相似度:{similarity:.1%}, {confidence_level}置信)"
            self.speaker_result_label.config(text=result_text, fg=color)
        else:
            result_text = f"✗ 未识别 (最高相似度:{similarity:.1%})"
            self.speaker_result_label.config(text=result_text, fg='red')

    def start_recording(self):
        """开始录音（使用独立音频流，避免 ALSA 断言错误）"""
        if self.is_recording:
            messagebox.showwarning("警告", "正在录音中")
            return

        self.is_recording = True
        self.recorded_audio = []
        self.record_button.config(state=tk.DISABLED)
        self.stop_record_button.config(state=tk.NORMAL)
        self.log_info("  开始录音...")

        def record_thread():
            """录音线程 - 使用独立音频流避免与实时识别冲突"""
            record_stream = None

            try:
                # === 创建独立的录音流（关键修复）===
                print("[录音] 正在创建独立音频流...")
                record_stream = self.audio.open(
                    format=pyaudio.paInt16,
                    channels=1,
                    rate=16000,
                    input=True,
                    frames_per_buffer=2048,
                    stream_callback=None  # 使用阻塞模式，更稳定
                )
                print("[录音] ✓ 独立音频流创建成功")

                start_time = time.time()
                error_count = 0
                max_errors = 5
                consecutive_success = 0
                chunk_size = 2048
```

```
                while self.is_recording:
                    try:
                        # 检查录音时长
                        elapsed = time.time() - start_time
                        if elapsed > 30:  # 最长 30 秒
                            print("[录音] 达到最大录音时长(30 秒)，自动停止")
                            self.root.after(0, self.stop_recording)
                            break

                        # 从独立流读取音频（不会与实时识别冲突）
                        audio_data = record_stream.read(
                            chunk_size,
                            exception_on_overflow=False
                        )
                        audio_array = np.frombuffer(audio_data, dtype=np.int16)
                        self.recorded_audio.extend(audio_array)

                        # 重置错误计数
                        error_count = 0
                        consecutive_success += 1

                        # 更新录音时长
                        self.root.after(0, lambda e=elapsed: self.recording_time_label.config(
                            text=f"录音时长: {e:.1f}秒"
                        ))

                        # 适当休眠，降低 CPU 占用
                        time.sleep(0.005)

                    except (IOError, OSError) as e:
                        # ALSA 缓冲区错误处理
                        error_count += 1
                        error_msg = str(e)
                        print(f"[录音] 音频流错误 #{error_count}: {error_msg}")

                        if error_count >= max_errors:
                            print("[录音] ✗ 连续错误过多，停止录音")
                            self.is_recording = False
                            self.root.after(0, lambda: self.record_button.config(state=tk.NORMAL))
                            self.root.after(0, lambda: self.stop_record_button.config(state=tk.DISABLED))
                            self.root.after(0, lambda: messagebox.showerror(
```

```
                                "音频设备错误",
                                "录音过程中发生音频流错误，已停止。\n\n"
                                "可能原因：\n"
                                "• ALSA 驱动缓冲区同步问题\n"
                                "• 多线程访问冲突（已修复）\n"
                                "• 系统资源不足\n\n"
                                "建议：已使用独立音频流，重试即可"
                            ))
                            break

                        # 尝试快速恢复
                        time.sleep(0.05)

                    except Exception as e:
                        error_count += 1
                        print(f"[录音] 未知错误 #{error_count}: {type(e).__name__}:
{e}")

                        if error_count >= max_errors:
                            print("[录音] ✗ 错误过多，停止录音")
                            self.is_recording = False
                            self.root.after(0, self.stop_recording)
                            break

                        time.sleep(0.1)

                print(f"[录音] 线程退出 (成功读取: {consecutive_success} 次)")

            except Exception as e:
                print(f"[录音] ✗ 创建音频流失败: {e}")
                self.is_recording = False
                self.root.after(0, lambda: self.record_button.config(state=tk.NORMAL))
                self.root.after(0,                                                    lambda:
self.stop_record_button.config(state=tk.DISABLED))
                self.root.after(0, lambda: messagebox.showerror(
                    "录音错误",
                    f"无法创建录音流: {e}\n\n 请检查麦克风设备是否正常"
                ))

            finally:
                # 确保关闭独立的录音流
                if record_stream is not None:
                    try:
                        record_stream.stop_stream()
```

```
                    record_stream.close()
                    print("[录音] ✓ 独立音频流已关闭")
                except:
                    pass

        threading.Thread(target=record_thread, daemon=True).start()

    def stop_recording(self):
        """停止录音"""
        if not self.is_recording:
            return

        self.is_recording = False
        self.record_button.config(state=tk.NORMAL)
        self.stop_record_button.config(state=tk.DISABLED)

        if len(self.recorded_audio) < 16000:  # 少于 1 秒
            messagebox.showwarning("警告", "录音时长太短，至少需要 1 秒")
            self.recorded_audio = []
            self.recording_time_label.config(text="录音时长: 0.0 秒")
            return

        duration = len(self.recorded_audio) / 16000
        self.log_info(f"✓ 录音完成: {duration:.1f}秒")

        # 询问姓名并注册
        name = simpledialog.askstring("注册说话人", "请输入说话人姓名:")
        if name and name.strip():
            audio_array = np.array(self.recorded_audio)
            success, message = self.speaker_recognition.register_speaker_from_audio(
                name.strip(), audio_array
            )

            if success:
                self.log_info(f"✓ {message}")
                self.update_speaker_list()
                messagebox.showinfo("成功", message)
            else:
                self.log_info(f"✗ {message}")
                messagebox.showerror("失败", message)

        self.recorded_audio = []
        self.recording_time_label.config(text="录音时长: 0.0 秒")
```

```
def register_speaker_from_file(self):
    """从文件注册说话人"""
    file_path = filedialog.askopenfilename(
        title="选择音频文件",
        filetypes=[("音频文件", "*.wav *.mp3 *.flac *.m4a"), ("所有文件", "*.*")]
    )

    if not file_path:
        return

    name = simpledialog.askstring("注册说话人", "请输入说话人姓名:")
    if not name or not name.strip():
        return

    success, message = self.speaker_recognition.register_speaker_from_file(
        name.strip(), file_path
    )

    if success:
        self.log_info(f"✓ {message}")
        self.update_speaker_list()
        messagebox.showinfo("成功", message)
    else:
        self.log_info(f"✗ {message}")
        messagebox.showerror("失败", message)

def update_speaker_list(self):
    """更新说话人列表（已废弃，保留用于兼容）"""
    pass

def show_registered_people(self):
    """显示已注册人员信息窗口"""
    # 创建新窗口
    registered_window = tk.Toplevel(self.root)
    registered_window.title("已注册人员信息")
    registered_window.geometry("900x650")
    registered_window.resizable(True, True)

    # 创建笔记本（标签页）
    notebook = ttk.Notebook(registered_window)
    notebook.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

    # 人脸识别标签页
    face_tab = ttk.Frame(notebook)
```

```
        notebook.add(face_tab, text="    已注册人脸")

        # 说话人识别标签页
        speaker_tab = ttk.Frame(notebook)
        notebook.add(speaker_tab, text="    已注册说话人")

        # ========== 人脸识别标签页内容 ==========
        face_info_frame = tk.Frame(face_tab)
        face_info_frame.pack(fill=tk.X, padx=10, pady=5)

        # 获取人脸数据
        face_count = 0
        face_list = []
        if self.face_recognition:
            if self.face_recognition.model_type == "fusion":
                face_count = len(self.face_recognition.mbf_database)
                face_list = list(self.face_recognition.mbf_database.keys())
            else:
                face_count = len(self.face_recognition.face_database)
                face_list = list(self.face_recognition.face_database.keys())

        tk.Label(face_info_frame, text=f"已注册人脸: {face_count} 人",
                 font=('Arial', 14, 'bold'), fg='#2196F3').pack(pady=5)

        # 人脸列表框架
        face_list_frame = tk.Frame(face_tab)
        face_list_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)

        # 左侧列表
        left_face_frame = tk.Frame(face_list_frame)
        left_face_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        tk.Label(left_face_frame, text="人员列表:", font=('Arial', 11, 'bold')).pack(anchor='w')

        face_listbox_frame = tk.Frame(left_face_frame)
        face_listbox_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        face_scrollbar = tk.Scrollbar(face_listbox_frame)
        face_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        face_listbox = tk.Listbox(face_listbox_frame, font=('Arial', 11),
                                  yscrollcommand=face_scrollbar.set,
selectmode=tk.SINGLE)
        face_listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
```

```
        face_scrollbar.config(command=face_listbox.yview)

        for name in sorted(face_list):
            face_listbox.insert(tk.END, name)

        # 右侧信息面板
        right_face_frame = tk.Frame(face_list_frame, width=300)
        right_face_frame.pack(side=tk.RIGHT, fill=tk.BOTH, padx=(10, 0))
        right_face_frame.pack_propagate(False)

        tk.Label(right_face_frame, text="详细信息:", font=('Arial', 11, 'bold')).pack(anchor='w')

        face_detail_text = tk.Text(right_face_frame, font=('Courier', 10), wrap=tk.WORD)
        face_detail_text.pack(fill=tk.BOTH, expand=True, pady=5)

        def on_face_select(event):
            """选择人脸时显示详细信息"""
            selection = face_listbox.curselection()
            if not selection:
                return

            name = face_listbox.get(selection[0])
            face_detail_text.delete(1.0, tk.END)
            face_detail_text.insert(tk.END, f"姓名: {name}\n")
            face_detail_text.insert(tk.END, "=" * 40 + "\n\n")

            # 查询数据库获取详细信息
            if self.db_manager:
                conn = None
                try:
                    conn = self.db_manager.db_pool.getconn()
                    cur = conn.cursor()

                    if self.face_recognition.model_type == "fusion":
                        # 融合模式：查询两个模型
                        cur.execute("""
                            SELECT model_type, registered_at,
                                   CASE WHEN image_data IS NOT NULL THEN '已上传至数据库' ELSE '无' END as image_status
                            FROM face_embeddings
                            WHERE name = %s AND model_type IN ('mobilefacenet', 'arcface')
                            ORDER BY model_type
                        """, (name,))
```

```
                        else:
                            cur.execute("""
                                SELECT model_type, registered_at,
                                       CASE WHEN image_data IS NOT NULL THEN '已上传至数据库' ELSE '无' END as image_status
                                FROM face_embeddings
                                WHERE name = %s AND model_type = %s
                            """, (name, self.face_recognition.model_type))

                        results = cur.fetchall()
                        cur.close()

                        if results:
                            for model_type, registered_at, image_status in results:
                                face_detail_text.insert(tk.END, f"模型: {model_type}\n")
                                face_detail_text.insert(tk.END, f"注册时间: {registered_at}\n")
                                face_detail_text.insert(tk.END, f"图像数据: {image_status}\n")
                                face_detail_text.insert(tk.END, "\n")
                        else:
                            face_detail_text.insert(tk.END, "未找到数据库记录\n")

                    except Exception as e:
                        face_detail_text.insert(tk.END, f"查询失败: {e}\n")
                    finally:
                        if conn:
                            self.db_manager.db_pool.putconn(conn)

        face_listbox.bind('<<ListboxSelect>>', on_face_select)

        # 操作按钮
        face_btn_frame = tk.Frame(face_tab)
        face_btn_frame.pack(fill=tk.X, padx=10, pady=10)

        def delete_face():
            """删除选中的人脸（从数据库删除记录和图像数据）"""
            selection = face_listbox.curselection()
            if not selection:
                messagebox.showwarning("警告", "请先选择要删除的人员", parent=registered_window)
                return

            name = face_listbox.get(selection[0])
```

```
            if messagebox.askyesno("确认", f"确定要删除 {name} 的人脸信息吗？\n\n 将删除数据库中的记录和图像数据", parent=registered_window):
                deleted_files = []
                failed_files = []

                try:
                    # 直接从数据库删除记录（无需导出或删除本地文件）
                    if self.face_recognition.model_type == "fusion":
                        self.db_manager.delete_face_embedding(name, 'mobilefacenet')
                        self.db_manager.delete_face_embedding(name, 'arcface')
                        self.face_recognition._load_fusion_databases()
                    else:
                        self.db_manager.delete_face_embedding(name, self.face_recognition.model_type)
                        self.face_recognition.load_database()

                    # 显示结果
                    result_msg = f"✓ 已删除 {name}\n\n"
                    result_msg += f"数据库记录: 已删除（图像数据已从数据库中移除）"

                    messagebox.showinfo("删除完成", result_msg, parent=registered_window)
                    registered_window.destroy()
                    self.show_registered_people()  # 重新打开窗口

                except Exception as e:
                    messagebox.showerror("错误", f"删除失败: {e}", parent=registered_window)

        tk.Button(face_btn_frame, text="  删除选中人脸", command=delete_face,
                  font=('Arial', 11), bg='#F44336', fg='white').pack(side=tk.LEFT, padx=5)

        # ========== 说话人识别标签页内容 ==========
        speaker_info_frame = tk.Frame(speaker_tab)
        speaker_info_frame.pack(fill=tk.X, padx=10, pady=5)

        # 获取说话人数据
        speaker_count = 0
        speaker_list = []
        if self.speaker_recognition:
            speaker_list = self.speaker_recognition.get_registered_speakers()
            speaker_count = len(speaker_list)

        tk.Label(speaker_info_frame, text=f"已注册说话人: {speaker_count} 人",
```

```
                font=('Arial', 14, 'bold'), fg='#FF5722').pack(pady=5)

        # 说话人列表框架
        speaker_list_frame = tk.Frame(speaker_tab)
        speaker_list_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)

        # 左侧列表
        left_speaker_frame = tk.Frame(speaker_list_frame)
        left_speaker_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        tk.Label(left_speaker_frame, text="说话人列表:", font=('Arial', 11,
'bold')).pack(anchor='w')

        speaker_listbox_frame = tk.Frame(left_speaker_frame)
        speaker_listbox_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        speaker_scrollbar = tk.Scrollbar(speaker_listbox_frame)
        speaker_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        speaker_listbox = tk.Listbox(speaker_listbox_frame, font=('Arial', 11),
                                     yscrollcommand=speaker_scrollbar.set,
selectmode=tk.SINGLE)
        speaker_listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        speaker_scrollbar.config(command=speaker_listbox.yview)

        for name in sorted(speaker_list):
            speaker_listbox.insert(tk.END, name)

        # 右侧信息面板
        right_speaker_frame = tk.Frame(speaker_list_frame, width=300)
        right_speaker_frame.pack(side=tk.RIGHT, fill=tk.BOTH, padx=(10, 0))
        right_speaker_frame.pack_propagate(False)

        tk.Label(right_speaker_frame, text="详细信息:", font=('Arial', 11,
'bold')).pack(anchor='w')

        speaker_detail_text = tk.Text(right_speaker_frame, font=('Courier', 10),
wrap=tk.WORD)
        speaker_detail_text.pack(fill=tk.BOTH, expand=True, pady=5)

        def on_speaker_select(event):
            """选择说话人时显示详细信息"""
            selection = speaker_listbox.curselection()
            if not selection:
```

```
                return

            name = speaker_listbox.get(selection[0])
            speaker_detail_text.delete(1.0, tk.END)
            speaker_detail_text.insert(tk.END, f"姓名: {name}\n")
            speaker_detail_text.insert(tk.END, "=" * 40 + "\n\n")

            # 查询数据库获取详细信息
            if self.db_manager:
                conn = None
                try:
                    conn = self.db_manager.db_pool.getconn()
                    cur = conn.cursor()

                    if self.speaker_recognition.model_type == "fusion":
                        # 融合模式：查询两个模型
                        cur.execute("""
                            SELECT model_type, registered_at,
                                   CASE WHEN audio_data IS NOT NULL THEN '已上传至数据库' ELSE '无' END as audio_status
                            FROM speaker_embeddings
                            WHERE name = %s AND model_type IN ('ecapa', 'resnet')
                            ORDER BY model_type
                        """, (name,))
                    else:
                        cur.execute("""
                            SELECT model_type, registered_at,
                                   CASE WHEN audio_data IS NOT NULL THEN '已上传至数据库' ELSE '无' END as audio_status
                            FROM speaker_embeddings
                            WHERE name = %s AND model_type = %s
                        """, (name, self.speaker_recognition.model_type))

                    results = cur.fetchall()
                    cur.close()

                    if results:
                        for model_type, registered_at, audio_status in results:
                            speaker_detail_text.insert(tk.END, f"模型: {model_type}\n")
                            speaker_detail_text.insert(tk.END, f"注册时间: {registered_at}\n")
                            speaker_detail_text.insert(tk.END, f"音频数据: {audio_status}\n")
                            speaker_detail_text.insert(tk.END, "\n")
```

```
                    else:
                        speaker_detail_text.insert(tk.END, "未找到数据库记录\n")

                except Exception as e:
                    speaker_detail_text.insert(tk.END, f"查询失败: {e}\n")
                finally:
                    if conn:
                        self.db_manager.db_pool.putconn(conn)

        speaker_listbox.bind('<<ListboxSelect>>', on_speaker_select)

        # 操作按钮
        speaker_btn_frame = tk.Frame(speaker_tab)
        speaker_btn_frame.pack(fill=tk.X, padx=10, pady=10)

        def delete_speaker():
            """删除选中的说话人（包括数据库记录和本地文件）"""
            selection = speaker_listbox.curselection()
            if not selection:
                messagebox.showwarning("警告", "请先选择要删除的说话人", parent=registered_window)
                return

            name = speaker_listbox.get(selection[0])
            if messagebox.askyesno("确认", f"确定要删除 {name} 的说话人信息吗？\n\n将删除数据库中的记录和音频数据", parent=registered_window):
                deleted_files = []
                failed_files = []

                try:
                    # 直接从数据库删除记录（无需导出或删除本地文件）
                    if self.speaker_recognition.delete_speaker(name):
                        # 显示结果
                        result_msg = f"✓ 已删除 {name}\n\n"
                        result_msg += f"数据库记录: 已删除（音频数据已从数据库中移除）"

                        messagebox.showinfo("删除完成", result_msg, parent=registered_window)
                        registered_window.destroy()
                        self.show_registered_people()  # 重新打开窗口
                    else:
                        messagebox.showerror("错误", "数据库删除失败", parent=registered_window)
```

```
                except Exception as e:
                    messagebox.showerror("错误", f"删除失败：{e}",
parent=registered_window)

        tk.Button(speaker_btn_frame, text="  删除选中说话人", command=delete_speaker,
                  font=('Arial', 11), bg='#F44336', fg='white').pack(side=tk.LEFT, padx=5)

        # 关闭按钮
        close_btn_frame = tk.Frame(registered_window)
        close_btn_frame.pack(fill=tk.X, padx=10, pady=10)

        tk.Button(close_btn_frame, text="✕ 关闭", command=registered_window.destroy,
                  font=('Arial', 11), bg='#9E9E9E', fg='white').pack(side=tk.RIGHT)

    def delete_selected_speaker(self):
        """删除选中的说话人（已废弃，保留用于兼容）"""
        pass

    def toggle_speaker_recognition(self):
        """切换说话人识别"""
        self.speaker_recognition_enabled = self.speaker_recognition_var.get()
        status = "已启用" if self.speaker_recognition_enabled else "已禁用"
        self.log_info(f"说话人识别: {status}")

    def toggle_multimodal_fusion(self):
        """切换完整多模态融合"""
        self.multimodal_fusion_enabled = self.multimodal_fusion_var.get()

        if self.multimodal_fusion_enabled:
            # 检查必要条件
            if self.face_recognition is None:
                messagebox.showwarning("警告", "人脸识别未初始化")
                self.multimodal_fusion_var.set(False)
                self.multimodal_fusion_enabled = False
                return

            if self.speaker_recognition is None:
                messagebox.showwarning("警告", "说话人识别未初始化")
                self.multimodal_fusion_var.set(False)
                self.multimodal_fusion_enabled = False
                return

            # 自动启用人脸识别和说话人识别
```

```
            self.recognition_var.set(True)
            self.recognition_enabled = True
            self.speaker_recognition_var.set(True)
            self.speaker_recognition_enabled = True

            self.log_info("=" * 40)
            self.log_info("✓ 四模态融合认证已启用")
            self.log_info("   - 人脸: MobileFaceNet + ArcFace")
            self.log_info("   - 声音: ECAPA-TDNN + ResNet34-SE")
            self.log_info("=" * 40)

            self.multimodal_result_label.config(text="融合认证中...", fg='blue')
        else:
            self.log_info("✓ 四模态融合认证已禁用")
            self.multimodal_result_label.config(text="未启用", fg='gray')

    def perform_multimodal_fusion(self, face_result, speaker_result):
        """执行多模态融合认证

        Args:
            face_result: (name, similarity) 人脸识别结果
            speaker_result: (name, similarity, success) 说话人识别结果

        Returns:
            (name, score, success) 融合后的结果
        """
        face_name, face_sim = face_result
        speaker_name, speaker_sim, speaker_success = speaker_result

        # 获取融合权重
        face_weight = self.face_weight_scale.get()
        speaker_weight = self.speaker_weight_scale.get()

        # 归一化权重
        total_weight = face_weight + speaker_weight
        if total_weight > 0:
            face_weight = face_weight / total_weight
            speaker_weight = speaker_weight / total_weight
        else:
            face_weight = speaker_weight = 0.5

        # 融合逻辑
        if face_name == "未知" and speaker_name == "未知":
            return "未知", 0.0, False
```

```
        if face_name == "未知":
            # 只有说话人识别成功
            return speaker_name, speaker_sim * speaker_weight, speaker_success

        if speaker_name == "未知":
            # 只有人脸识别成功
            threshold = 0.4
            return face_name, face_sim * face_weight, face_sim > threshold

        # 两者都识别出来
        if face_name == speaker_name:
            # 身份一致 - 高置信度
            fusion_score = face_weight * face_sim + speaker_weight * speaker_sim
            return face_name, fusion_score, True
        else:
            # 身份不一致 - 需要更高的阈值
            fusion_score_face = face_weight * face_sim
            fusion_score_speaker = speaker_weight * speaker_sim

            # 选择分数更高的那个，但要求很高的置信度
            if fusion_score_face > fusion_score_speaker:
                return face_name, fusion_score_face, fusion_score_face > 0.6
            else:
                return speaker_name, fusion_score_speaker, fusion_score_speaker > 0.6

    def update_ui_frame(self):
        """更新 UI 显示（优化：异步识别，防止卡顿）"""
        if self.current_frame is not None:
            frame = self.current_frame.copy()

            face_result = ("未知", 0.0)
            speaker_result = self.last_speaker_result + (False,)

            # 人脸识别（优化：异步处理，不阻塞 UI 线程）
            if self.recognition_enabled and self.face_recognition:
                self.frame_counter += 1
                should_detect = (self.frame_counter % self.recognition_frame_skip == 0)

                if should_detect:
                    # 检测人脸
                    faces = self.face_recognition.detect_faces(frame)
                    current_time = time.time()
```

```
                    # 将检测到的人脸加入识别队列
                    for (x, y, w, h) in faces:
                        face_roi = frame[y:y+h, x:x+w]
                        if face_roi.size > 0:
                            bbox = (x, y, w, h)

                            # 检查是否已有最近的结果
                            with self.recognition_lock:
                                has_recent_result = False
                                for cached_bbox, result in list(self.recognition_results.items()):
                                    name, similarity, timestamp, inf_time = result
                                    # 如果位置相近且结果未过期
                                    if (abs(cached_bbox[0] - x) < 50 and
                                        abs(cached_bbox[1] - y) < 50 and
                                        current_time - timestamp < 1.0):
                                        has_recent_result = True
                                        break

                                # 只有没有最近结果且队列不满时才加入队列
                                if not has_recent_result and len(self.recognition_queue) < 10:
                                    self.recognition_queue.append((face_roi.copy(), bbox, self.frame_counter))

                    # 绘制识别结果（使用缓存的结果）
                    faces = self.face_recognition.detect_faces(frame)
                    current_time = time.time()

                    with self.recognition_lock:
                        for (x, y, w, h) in faces:
                            # 查找最匹配的缓存结果
                            best_match = None
                            best_distance = float('inf')

                            for cached_bbox, result in self.recognition_results.items():
                                name, similarity, timestamp, inference_time = result
                                # 计算位置距离
                                distance = abs(cached_bbox[0] - x) + abs(cached_bbox[1] - y)

                                # 如果距离近且结果未过期
                                if distance < 100 and current_time - timestamp < 2.0:
                                    if distance < best_distance:
                                        best_distance = distance
```

```
                                        best_match = (name, similarity, inference_time)

                                # 绘制
                                if best_match:
                                    name, similarity, inference_time = best_match
                                    face_result = (name, similarity)

                                    color = (0, 255, 0) if name != "未知" else (0, 0, 255)
                                    cv2.rectangle(frame, (x, y), (x+w, y+h), color, 2)

                                    label = f"{name} ({similarity:.2%})"
                                    sublabel = f"{inference_time:.0f}ms"

                                    # 主标签
                                    label_size = cv2.getTextSize(label,
cv2.FONT_HERSHEY_SIMPLEX, 0.6, 2)[0]
                                    cv2.rectangle(frame, (x, y-label_size[1]-10), (x+label_size[0],
y), color, -1)
                                    cv2.putText(frame, label, (x, y-5),
                                                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255,
255), 2)

                                    # 时间标签
                                    cv2.putText(frame, sublabel, (x, y+h+15),
                                                cv2.FONT_HERSHEY_SIMPLEX, 0.4, color, 1)
                                else:
                                    # 未找到结果，显示"检测中"
                                    color = (255, 255, 0)
                                    cv2.rectangle(frame, (x, y), (x+w, y+h), color, 2)
                                    cv2.putText(frame, "Detecting...", (x, y-5),
                                                cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2)

                        # 清理过期的识别结果
                        with self.recognition_lock:
                            expired_keys = [
                                bbox for bbox, result in self.recognition_results.items()
                                if current_time - result[2] > 5.0
                            ]
                            for key in expired_keys:
                                del self.recognition_results[key]

                    # 显示说话人识别结果
                    if self.speaker_recognition_enabled:
                        speaker_name, speaker_sim = self.last_speaker_result
```

```
                    speaker_success = speaker_sim > 0.48
                    speaker_result = (speaker_name, speaker_sim, speaker_success)

                    # 根据相似度选择颜色
                    if speaker_sim >= 0.55:
                        color = (0, 255, 0)  # 绿色 - 高置信
                    elif speaker_sim >= 0.45:
                        color = (0, 165, 255)  # 橙色 - 中等置信
                    else:
                        color = (0, 0, 255)  # 红色 - 低置信

                    text = f"Speaker: {speaker_name} ({speaker_sim:.1%})"
                    cv2.putText(frame, text, (10, 30),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)

                # 多模态融合结果显示
                if self.multimodal_fusion_enabled:
                    fusion_name, fusion_score, fusion_success =
self.perform_multimodal_fusion(
                        face_result, speaker_result
                    )

                    # 在界面上显示融合结果
                    if fusion_success:
                        color = (0, 255, 0) if fusion_score > 0.6 else (0, 165, 255)
                        result_text = f"✓ 融合认证: {fusion_name} ({fusion_score:.1%})"
                        fg_color = 'dark green' if fusion_score > 0.6 else 'orange'
                    else:
                        color = (0, 0, 255)
                        result_text = f"✗ 融合认证失败 ({fusion_score:.1%})"
                        fg_color = 'red'

                    # 在视频上显示
                    cv2.putText(frame, f"Fusion: {fusion_name} ({fusion_score:.1%})",
                                (10, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)

                    # 更新 UI 标签
                    self.root.after(0, lambda: self.multimodal_result_label.config(
                        text=result_text, fg=fg_color
                    ))
                else:
                    # 未启用融合时清空融合结果标签
                    self.root.after(0, lambda: self.multimodal_result_label.config(
                        text="未启用融合认证", fg='gray'
```

```
                ))

            # 转换为 PhotoImage
            frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            img = Image.fromarray(frame_rgb)
            img = img.resize((800, 600))
            photo = ImageTk.PhotoImage(image=img)

            self.video_label.configure(image=photo)
            self.video_label.image = photo

        if self.camera_running:
            self.root.after(33, self.update_ui_frame)

    def log_info(self, message):
        """记录信息"""
        timestamp = datetime.now().strftime("%H:%M:%S")
        self.info_text.insert(tk.END, f"[{timestamp}] {message}\n")
        self.info_text.see(tk.END)

    def register_current_face(self):
        """注册当前人脸"""
        if self.current_frame is None:
            messagebox.showwarning("警告", "没有视频画面")
            return

        frame = self.current_frame.copy()
        faces = self.face_recognition.detect_faces(frame)

        if len(faces) == 0:
            messagebox.showwarning("警告", "未检测到人脸")
            return

        if len(faces) > 1:
            messagebox.showwarning("警告", "检测到多个人脸，请确保只有一人")
            return

        name = simpledialog.askstring("注册人脸", "请输入姓名:")
        if name and name.strip():
            x, y, w, h = faces[0]
            face_roi = frame[y:y+h, x:x+w]

            if self.face_recognition.register_face(name.strip(), face_roi):
                self.log_info(f"✓ 人脸注册成功: {name}")
```

```
                messagebox.showinfo("成功", f"已注册: {name}")
            else:
                messagebox.showerror("错误", "注册失败")

    def toggle_recognition(self):
        """切换人脸识别"""
        self.recognition_enabled = self.recognition_var.get()
        status = "已启用" if self.recognition_enabled else "已禁用"
        self.log_info(f"人脸识别: {status}")

    def on_face_model_changed(self):
        """人脸识别模型切换时的回调"""
        # 如果选择了 Fusion 模式，禁用说话人识别控件（避免四模型融合冲突）
        if self.model_var.get() == "fusion":
            self.log_info("人脸识别: Fusion 模式，建议单独使用")

    def on_speaker_model_changed(self):
        """说话人识别模型切换时的回调"""
        # 先执行原有的模型切换逻辑
        self.switch_speaker_model()

        # 如果选择了 Fusion 模式，禁用人脸识别控件（避免四模型融合冲突）
        if self.speaker_model_var.get() == "fusion":
            self.log_info("说话人识别: Fusion 模式，建议单独使用")

    def update_skip_frames(self, value):
        """更新跳帧数"""
        self.recognition_frame_skip = int(value)
        fps_estimate = 30 / self.recognition_frame_skip
        self.log_info(f"识别频率: 每{value}帧识别一次 (约{fps_estimate:.1f}fps)")

    def setup_ui(self):
        """设置 UI"""
        # 主布局
        main_frame = tk.Frame(self.root)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # 左侧：视频显示
        left_frame = tk.Frame(main_frame)
        left_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=5)

        tk.Label(left_frame, text="实时视频", font=('Arial', 14, 'bold')).pack()
        self.video_label = tk.Label(left_frame, bg='black')
        self.video_label.pack(fill=tk.BOTH, expand=True, pady=5)
```

```
        # 右侧：控制面板
        right_frame = tk.Frame(main_frame, width=400)
        right_frame.pack(side=tk.RIGHT, fill=tk.BOTH, padx=5)
        right_frame.pack_propagate(False)

        # 网络传输控制（新增）
        network_frame = tk.LabelFrame(right_frame, text="  网络传输", font=('Arial', 12,
'bold'), fg='#2196F3')
        network_frame.pack(fill=tk.X, pady=5)

        tk.Label(network_frame, text=f"视频端口：{self.video_port}  音频端口：
{self.audio_port}",
                 font=('Arial', 9)).pack(pady=3)

        self.network_button = tk.Button(network_frame, text="启动网络服务器",
                                        command=self.toggle_network_server,
                                        font=('Arial', 11, 'bold'), bg='#4CAF50',
fg='white')
        self.network_button.pack(pady=5, fill=tk.X, padx=10)

        # 人脸识别控制
        face_frame = tk.LabelFrame(right_frame, text="人脸识别", font=('Arial', 12, 'bold'))
        face_frame.pack(fill=tk.X, pady=5)

        model_frame = tk.Frame(face_frame)
        model_frame.pack(fill=tk.X, padx=10, pady=5)

        tk.Label(model_frame, text="模型:").pack(side=tk.LEFT)
        # self.model_var 已在 run() 中初始化
        # 保存人脸识别模型选择按钮的引用
        self.face_model_radios = []
        self.face_model_radios.append(tk.Radiobutton(model_frame, text="MBF",
variable=self.model_var,
                                                     value="mobilefacenet"))
        self.face_model_radios[-1].pack(side=tk.LEFT, padx=5)
        self.face_model_radios.append(tk.Radiobutton(model_frame, text="ARC",
variable=self.model_var,
                                                     value="arcface"))
        self.face_model_radios[-1].pack(side=tk.LEFT, padx=5)
        self.face_model_radios.append(tk.Radiobutton(model_frame, text="Fusion",
variable=self.model_var,
                                                     value="fusion", command=self.on_face_model_changed))
        self.face_model_radios[-1].pack(side=tk.LEFT, padx=5)
```

```
        # self.recognition_var 已在 run() 中初始化
        self.face_recognition_checkbox = tk.Checkbutton(face_frame, text="启用人脸识别",
variable=self.recognition_var,
                                        command=self.toggle_recognition, font=('Arial', 11))
        self.face_recognition_checkbox.pack(pady=5)

        # 性能调节（跳帧设置）
        skip_frame = tk.Frame(face_frame)
        skip_frame.pack(fill=tk.X, padx=10, pady=3)
        tk.Label(skip_frame, text="识别频率:", font=('Arial', 9)).pack(side=tk.LEFT)
        self.skip_scale = tk.Scale(skip_frame, from_=1, to=10, orient=tk.HORIZONTAL,
                                   length=150, command=self.update_skip_frames)
        self.skip_scale.set(3)  # 默认每 3 帧识别一次
        self.skip_scale.pack(side=tk.LEFT, padx=5)
        tk.Label(skip_frame, text="帧", font=('Arial', 9)).pack(side=tk.LEFT)

        tk.Button(face_frame, text="注册当前人脸", command=self.register_current_face,
                  font=('Arial', 11), bg='#4CAF50', fg='white').pack(pady=5, fill=tk.X,
padx=10)

        # 说话人识别控制
        speaker_frame = tk.LabelFrame(right_frame, text="说话人识别", font=('Arial', 12,
'bold'))
        speaker_frame.pack(fill=tk.X, pady=5)

        if self.speaker_recognition is not None:
            # 模型选择
            speaker_model_frame = tk.Frame(speaker_frame)
            speaker_model_frame.pack(fill=tk.X, padx=10, pady=5)

            tk.Label(speaker_model_frame, text="模型:", font=('Arial', 10,
'bold')).pack(side=tk.LEFT)
            # self.speaker_model_var 已在 run() 中初始化
            # 保存说话人识别模型选择按钮的引用
            self.speaker_model_radios = []
            self.speaker_model_radios.append(tk.Radiobutton(speaker_model_frame,
text="ECAPA",
                                                            variable=self.speaker_model_var,
                                                            value="ecapa",
                                                            command=self.switch_speaker_model))
            self.speaker_model_radios[-1].pack(side=tk.LEFT, padx=3)
            self.speaker_model_radios.append(tk.Radiobutton(speaker_model_frame,
text="ResNet",
```

```
                                variable=self.speaker_model_var,
                                value="resnet",
                                command=self.switch_speaker_model))
            self.speaker_model_radios[-1].pack(side=tk.LEFT, padx=3)
            self.speaker_model_radios.append(tk.Radiobutton(speaker_model_frame,
text="Fusion",
                                variable=self.speaker_model_var,
                                value="fusion",
                                command=self.on_speaker_model_changed))
            self.speaker_model_radios[-1].pack(side=tk.LEFT, padx=3)

            # self.speaker_recognition_var 已在 run() 中初始化
            self.speaker_recognition_checkbox = tk.Checkbutton(speaker_frame, text="启用说
话人识别",
                                variable=self.speaker_recognition_var,
                                command=self.toggle_speaker_recognition,
                                font=('Arial', 11))
            self.speaker_recognition_checkbox.pack(pady=5)

            # 识别结果显示
            self.speaker_result_label = tk.Label(speaker_frame, text="未识别",
                                                 font=('Arial', 12, 'bold'), fg='gray')
            self.speaker_result_label.pack(pady=5)

            # 录音注册
            record_frame = tk.Frame(speaker_frame)
            record_frame.pack(fill=tk.X, padx=10, pady=5)

            self.record_button = tk.Button(record_frame, text="    开始录音",
                                           command=self.start_recording,
                                           font=('Arial', 10), bg='#FF5722', fg='white')
            self.record_button.pack(side=tk.LEFT, padx=2, expand=True, fill=tk.X)

            self.stop_record_button = tk.Button(record_frame, text="■ 停止录音",
                                                command=self.stop_recording,
                                                font=('Arial',      10),      bg='#9E9E9E',
fg='white',
                                                state=tk.DISABLED)
            self.stop_record_button.pack(side=tk.LEFT, padx=2, expand=True, fill=tk.X)

            self.recording_time_label = tk.Label(speaker_frame, text="录音时长: 0.0 秒",
                                                 font=('Arial', 9))
            self.recording_time_label.pack()
```

```
        else:
            tk.Label(speaker_frame, text="说话人识别不可用\n 请安装 SpeechBrain",
                     font=('Arial', 10), fg='red').pack(pady=10)

        # 已注册人查看按钮（独立区域）
        registered_frame = tk.LabelFrame(right_frame, text="    已注册信息", font=('Arial', 12,
'bold'), fg='#673AB7')
        registered_frame.pack(fill=tk.X, pady=5)

        tk.Button(registered_frame, text="    查看已注册人员",
                  command=self.show_registered_people,
                  font=('Arial', 12, 'bold'), bg='#673AB7', fg='white',
                  height=2).pack(pady=10, fill=tk.X, padx=10)

        # 完整多模态融合控制
        multimodal_frame = tk.LabelFrame(right_frame, text="    完整多模态融合",
                                         font=('Arial', 12, 'bold'), fg='#FF5722')
        multimodal_frame.pack(fill=tk.X, pady=5)

        self.multimodal_fusion_var = tk.BooleanVar(value=False)
        tk.Checkbutton(multimodal_frame, text="启用四模态融合认证",
                       variable=self.multimodal_fusion_var,
                       command=self.toggle_multimodal_fusion,
                       font=('Arial', 11, 'bold'), fg='#FF5722').pack(pady=5)

        tk.Label(multimodal_frame, text="融合：人脸(MBF+ARC) + 声音(ECAPA+ResNet)",
                 font=('Arial', 9), fg='gray').pack()

        # 融合结果显示
        self.multimodal_result_label = tk.Label(multimodal_frame, text="未启用",
                                                font=('Arial', 11, 'bold'), fg='gray')
        self.multimodal_result_label.pack(pady=5)

        # 融合权重设置
        weight_frame = tk.Frame(multimodal_frame)
        weight_frame.pack(fill=tk.X, padx=10, pady=3)

        tk.Label(weight_frame, text="人脸权重:", font=('Arial', 9)).pack(side=tk.LEFT)
        self.face_weight_scale = tk.Scale(weight_frame, from_=0, to=1, resolution=0.1,
                                          orient=tk.HORIZONTAL, length=100)
        self.face_weight_scale.set(0.5)
        self.face_weight_scale.pack(side=tk.LEFT, padx=5)
```

```
        tk.Label(weight_frame, text="声音权重:", font=('Arial', 9)).pack(side=tk.LEFT,
padx=(10,0))
        self.speaker_weight_scale = tk.Scale(weight_frame, from_=0, to=1, resolution=0.1,
                                             orient=tk.HORIZONTAL, length=100)
        self.speaker_weight_scale.set(0.5)
        self.speaker_weight_scale.pack(side=tk.LEFT, padx=5)

        # 信息显示
        info_frame = tk.LabelFrame(right_frame, text="系统日志", font=('Arial', 12, 'bold'))
        info_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        self.info_text = tk.Text(info_frame, height=15, font=('Courier', 9))
        self.info_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        info_scrollbar = tk.Scrollbar(info_frame, command=self.info_text.yview)
        info_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.info_text.config(yscrollcommand=info_scrollbar.set)

        self.log_info("=" * 40)
        self.log_info("多模态生物识别系统已启动")
        self.log_info("=" * 40)

        # 显示音频设备信息
        if hasattr(self, 'audio_device_name'):
            self.log_info(f"    音频设备: {self.audio_device_name}")

    def run(self):
        """运行系统"""
        # 先初始化 UI 变量
        self.model_var = tk.StringVar(value="fusion")
        self.recognition_var = tk.BooleanVar(value=False)
        self.speaker_recognition_var = tk.BooleanVar(value=False)
        self.speaker_model_var = tk.StringVar(value="ecapa")  # 默认使用 ECAPA-TDNN

        if not self.setup_camera():
            return
        if not self.setup_audio():
            return
        if not self.init_face_recognition():
            return

        # 尝试初始化说话人识别（可选）
        self.init_speaker_recognition()
```

```
        # 启动相机线程
        self.camera_running = True
        self.camera_thread = threading.Thread(target=self.camera_loop, daemon=True)
        self.camera_thread.start()

        # 启动异步识别线程（关键修复：避免阻塞 UI）
        self.recognition_thread = threading.Thread(target=self._async_recognition_worker,
daemon=True)
        self.recognition_thread.start()

        # 启动音频采集线程（持续收集音频）
        self.audio_capture_thread = threading.Thread(target=self.audio_capture_loop,
daemon=True)
        self.audio_capture_thread.start()

        # 启动说话人识别线程
        if self.speaker_recognition is not None:
            self.speaker_thread = threading.Thread(target=self.speaker_recognition_loop,
daemon=True)
            self.speaker_thread.start()

        # 设置 UI
        self.setup_ui()

        # 更新说话人列表
        if self.speaker_recognition is not None:
            self.update_speaker_list()

        # 启动 UI 更新
        self.root.after(100, self.update_ui_frame)

        # 运行主循环
        try:
            self.root.mainloop()
        except KeyboardInterrupt:
            pass
        finally:
            self.cleanup()

    def cleanup(self):
        """
        清理资源（优化版）
        优雅地关闭所有线程和资源
        """
```

```
print("\n" + "=" * 60)
print("正在关闭系统...")
print("=" * 60)

# 步骤 1: 设置停止标志
print("[1/6] 设置停止标志...")
self.camera_running = False
self.is_recording = False
self.server_running = False
self.stop_event.set()  # 通知所有线程停止

# 步骤 2: 关闭网络连接
print("[2/6] 关闭网络连接...")
network_resources = [
    (self.video_client, "视频客户端"),
    (self.audio_client, "音频客户端"),
    (self.video_server, "视频服务器"),
    (self.audio_server, "音频服务器")
]

for resource, name in network_resources:
    if resource:
        try:
            resource.close()
            print(f"  ✓ {name}已关闭")
        except Exception as e:
            print(f"  ✗ {name}关闭失败: {e}")

# 步骤 3: 等待所有工作线程结束（带超时）
print("[3/6] 等待工作线程结束...")
threads_to_wait = [
    (self.camera_thread, "相机采集线程", 2.0),
    (self.video_encoder_thread, "视频编码线程", 2.0),
    (self.recognition_thread, "识别线程", 3.0),
    (self.audio_capture_thread, "音频采集线程", 2.0),
    (self.speaker_thread, "说话人识别线程", 3.0),
    (self.video_send_thread, "视频发送线程", 2.0),
    (self.audio_send_thread, "音频发送线程", 2.0)
]

for thread, name, timeout in threads_to_wait:
    if thread and thread.is_alive():
        print(f"  等待 {name}...")
        thread.join(timeout=timeout)
```

```
            if thread.is_alive():
                print(f"   ⚠ {name} 超时未响应（已强制继续）")
            else:
                print(f"   ✓ {name} 已停止")

    # 步骤 4: 关闭线程池
    print("[4/6] 关闭线程池...")
    if hasattr(self, 'thread_pool') and self.thread_pool:
        try:
            self.thread_pool.shutdown(wait=True, timeout=3.0)
            print("   ✓ 线程池已关闭")
        except Exception as e:
            print(f"   ⚠ 线程池关闭失败: {e}")

    # 步骤 5: 关闭音频资源
    print("[5/6] 关闭音频资源...")
    if hasattr(self, 'audio_stream') and self.audio_stream:
        try:
            if self.audio_stream.is_active():
                self.audio_stream.stop_stream()
            self.audio_stream.close()
            print("   ✓ 音频流已关闭")
        except Exception as e:
            print(f"   ✗ 音频流关闭失败: {e}")

    if hasattr(self, 'audio') and self.audio:
        try:
            self.audio.terminate()
            print("   ✓ PyAudio 已终止")
        except Exception as e:
            print(f"   ✗ PyAudio 终止失败: {e}")

    # 步骤 6: 关闭相机和数据库
    print("[6/6] 关闭相机和数据库...")
    if hasattr(self, 'cap') and self.cap:
        try:
            self.cap.release()
            print("   ✓ 相机已释放")
        except Exception as e:
            print(f"   ✗ 相机释放失败: {e}")

    # 关闭数据库管理器
    if hasattr(self, 'db_manager') and self.db_manager:
        try:
```

```
                    self.db_manager.close()
                    print("   ✓ 数据库连接池已关闭")
                except Exception as e:
                    print(f"   ✗ 数据库连接池关闭失败: {e}")

            # 清理 OpenCV 窗口
            cv2.destroyAllWindows()

            print("=" * 60)
            print("✓ 系统已完全关闭")
            print("=" * 60)


def main():
    print("=" * 70)
    print("多模态生物识别系统 - 人脸 + 说话人识别")
    print("=" * 70)
    print("功能:")
    print("   ✓ 人脸识别: MobileFaceNet / ArcFace / Fusion")
    print("   ✓ 说话人识别: ECAPA-TDNN / ResNet34-SE")
    print("   ✓ 双模态融合认证")
    print("   ✓ 网络传输: 视频(9999) / 音频(9998)")
    print()
    print("优化特性:")
    print("   • 动态线程池 + 智能队列")
    print("   • 视频管道解耦 (采集→编码→发送)")
    print("   • 无锁环形缓冲区 (音频)")
    print("   • 帧内存池 (减少 GC)")
    print("   • CPU 自适应 (JPEG 质量)")
    print("=" * 70)

    system = MultimodalBiometricSystem()
    system.run()


if __name__ == "__main__":
    main()
```