项目代码开发记录

一、当前项目结构

代码块

```
kol_ads_marketing
├─ data_collection_service: 数据采集服务
│   ├─app
│   │   ├─api
│   │   │   ├─endpoints
│   │   │   │   ├─bilibili_web.py
│   │   │   │   ├─download.py
│   │   │   ├─models
│   │   │   │   ├─APIResponseModel.py
│   ├─crawlers
│   │   ├─bilibili
│   │   │   ├─config.yaml
│   │   │   ├─endpoints.py
│   │   │   ├─models.py
│   │   │   ├─utils.py
│   │   │   ├─web_crawler.py
│   │   │   ├─wrid.py
│   │   ├─hybrid
│   │   │   ├─hybrid_crawler.py
│   │   ├─utils
│   │   │   ├─api_exceptions.py
│   │   │   ├─deprecated.py
│   │   │   ├─logger.py
│   │   │   ├─utils.py
│   │   ├─base_crawler.py
├─ deploy: 基础镜像配置
│   ├─docker-compose.yml
```

二、数据采集微服务模块

app

api

endpoints

- bilibili_web.py

代码块

```python
from fastapi import APIRouter, Body, Query, Request, HTTPException  # 导入FastAPI组件


from data_collection_service.app.api.models.APIResponseModel import ResponseModel, ErrorResponseModel  # 导入响应模型
from data_collection_service.crawlers.bilibili.web_crawler import BilibiliWebCrawler  # 导入哔哩哔哩web爬虫


router = APIRouter()
BilibiliWebCrawler = BilibiliWebCrawler()


# 获取单个视频详情信息
@router.get("/fetch_one_video", response_model=ResponseModel, summary="获取单个视频详情信息/Get single video data")
async def fetch_one_video(request: Request,
                          bv_id: str = Query(example="BV1M1421t7hT", description="作品id/Video id")):
    """
    # [中文]
    ### 用途:
    - 获取单个视频详情信息
    ### 参数:
    - bv_id: 作品id
    ### 返回:
    - 视频详情信息

    # [示例/Example]
    bv_id = "BV1M1421t7hT"
    """
    try:
        data = await BilibiliWebCrawler.fetch_one_video(bv_id)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取视频流地址
@router.get("/fetch_video_playurl", response_model=ResponseModel, summary="获取视频流地址/Get video playurl")
async def fetch_one_video(request: Request,
                          bv_id: str = Query(example="BV1y7411Q7Eq", description="作品id/Video id"),
                          cid:str = Query(example="171776208", description="作品cid/Video cid")):
    """
    # [中文]
    ### 用途:
    - 获取视频流地址
    ### 参数:
    - bv_id: 作品id
    - cid: 作品cid
    ### 返回:
    - 视频流地址

    # [示例/Example]
    bv_id = "BV1y7411Q7Eq"
    cid = "171776208"
    """
    try:
        data = await BilibiliWebCrawler.fetch_video_playurl(bv_id, cid)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取用户发布视频作品数据
@router.get("/fetch_user_post_videos", response_model=ResponseModel,
            summary="获取用户主页作品数据/Get user homepage video data")
async def fetch_user_post_videos(request: Request,
                                 uid: str = Query(example="178360345", description="用户UID"),
                                 pn: int = Query(default=1, description="页码/Page number"),):
```

```python
    """
    # [中文]
    ### 用途:
    - 获取用户发布的视频数据
    ### 参数:
    - uid: 用户UID
    - pn: 页码
    ### 返回:
    - 用户发布的视频数据

    # [示例/Example]
    uid = "178360345"
    pn = 1
    """
    try:
        data = await BilibiliWebCrawler.fetch_user_post_videos(uid, pn)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取用户所有收藏夹信息
@router.get("/fetch_collect_folders", response_model=ResponseModel,
            summary="获取用户所有收藏夹信息/Get user collection folders")
async def fetch_collect_folders(request: Request,
                                uid: str = Query(example="178360345",
description="用户UID")):
    """
    # [中文]
    ### 用途:
    - 获取用户收藏作品数据
    ### 参数:
    - uid: 用户UID
    ### 返回:
    - 用户收藏夹信息

    # [示例/Example]
    uid = "178360345"
    """
    try:
        data = await BilibiliWebCrawler.fetch_collect_folders(uid)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定收藏夹内视频数据
@router.get("/fetch_user_collection_videos", response_model=ResponseModel,
            summary="获取指定收藏夹内视频数据/Gets video data from a collection folder")
async def fetch_user_collection_videos(request: Request,
                                       folder_id: str = Query(example="1756059545",
                                                              description="收藏夹id/collection folder id"),
                                       pn: int = Query(default=1, description="页码/Page number")
                                       ):
    """
    # [中文]
    ### 用途:
    - 获取指定收藏夹内视频数据
    ### 参数:
    - folder_id: 用户UID
    - pn: 页码
    ### 返回:
    - 指定收藏夹内视频数据

    # [示例/Example]
    folder_id = "1756059545"
    pn = 1
    """
    try:
        data = await BilibiliWebCrawler.fetch_folder_videos(folder_id, pn)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定用户的信息
@router.get("/fetch_user_profile", response_model=ResponseModel,
            summary="获取指定用户的信息/Get information of specified user")
async def fetch_collect_folders(request: Request,
                                uid: str = Query(example="178360345", description="用户UID")):
    """
    # [中文]
    ### 用途:
    - 获取指定用户的信息
    ### 参数:
    - uid: 用户UID
    ### 返回:
    - 指定用户的个人信息

    # [示例/Example]
    uid = "178360345"
    """
    try:
        data = await BilibiliWebCrawler.fetch_user_profile(uid)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取综合热门视频信息
@router.get("/fetch_com_popular", response_model=ResponseModel,
            summary="获取综合热门视频信息/Get comprehensive popular video information")
async def fetch_collect_folders(request: Request,
                                pn: int = Query(default=1, description="页码/Page number")):
    """
    # [中文]
```

```python
    ### 用途:
    - 获取综合热门视频信息
    ### 参数:
    - pn: 页码
    ### 返回:
    - 综合热门视频信息

    # [示例/Example]
    pn = 1
    """
    try:
        data = await BilibiliWebCrawler.fetch_com_popular(pn)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定视频的评论
@router.get("/fetch_video_comments", response_model=ResponseModel,
            summary="获取指定视频的评论/Get comments on the specified video")
async def fetch_collect_folders(request: Request,
                                bv_id: str = Query(example="BV1M1421t7hT",
description="作品id/Video id"),
                                pn: int = Query(default=1, description="页
码/Page number")):
    """
    # [中文]
    ### 用途:
    - 获取指定视频的评论
    ### 参数:
    - bv_id: 作品id
    - pn: 页码
    ### 返回:
    - 指定视频的评论数据

    # [示例/Example]
    bv_id = "BV1M1421t7hT"
    pn = 1
    """
    try:
        data = await BilibiliWebCrawler.fetch_video_comments(bv_id, pn)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取视频下指定评论的回复
@router.get("/fetch_comment_reply", response_model=ResponseModel,
            summary="获取视频下指定评论的回复/Get reply to the specified comment")
async def fetch_collect_folders(request: Request,
                                bv_id: str = Query(example="BV1M1421t7hT", description="作品id/Video id"),
                                pn: int = Query(default=1, description="页码/Page number"),
                                rpid: str = Query(example="237109455120", description="回复id/Reply id")):
    """
    # [中文]
    ### 用途:
    - 获取视频下指定评论的回复
    ### 参数:
    - bv_id: 作品id
    - pn: 页码
    - rpid: 回复id
    ### 返回:
    - 指定评论的回复数据

    # [示例/Example]
    bv_id = "BV1M1421t7hT"
    pn = 1
    rpid = "237109455120"
    """
    try:
        data = await BilibiliWebCrawler.fetch_comment_reply(bv_id, pn, rpid)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定用户动态
@router.get("/fetch_user_dynamic", response_model=ResponseModel,
            summary="获取指定用户动态/Get dynamic information of specified user")
async def fetch_collect_folders(request: Request,
                                uid: str = Query(example="16015678", description="用户UID"),
                                offset: str = Query(default="", example="953154282154098691",
                                                    description="开始索引/offset")):
    """
    # [中文]
    ### 用途:
    - 获取指定用户动态
    ### 参数:
    - uid: 用户UID
    - offset: 开始索引
    ### 返回:
    - 指定用户动态数据

    # [示例/Example]
    uid = "178360345"
    offset = "953154282154098691"
    """
    try:
        data = await BilibiliWebCrawler.fetch_user_dynamic(uid, offset)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取视频实时弹幕
@router.get("/fetch_video_danmaku", response_model=ResponseModel, summary="获取视频实时弹幕/Get Video Danmaku")
```

```python
async def fetch_one_video(request: Request,
                          cid: str = Query(example="1639235405", description="作品cid/Video cid")):
    """
    # [中文]
    ### 用途:
    - 获取视频实时弹幕
    ### 参数:
    - cid: 作品cid
    ### 返回:
    - 视频实时弹幕

    # [示例/Example]
    cid = "1639235405"
    """
    try:
        data = await BilibiliWebCrawler.fetch_video_danmaku(cid)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定直播间信息
@router.get("/fetch_live_room_detail", response_model=ResponseModel,
            summary="获取指定直播间信息/Get information of specified live room")
async def fetch_collect_folders(request: Request,
                                room_id: str = Query(example="22816111", description="直播间ID/Live room ID")):
    """
    # [中文]
    ### 用途:
    - 获取指定直播间信息
    ### 参数:
    - room_id: 直播间ID
    ### 返回:
    - 指定直播间信息

    # [示例/Example]
    room_id = "22816111"
    """
    try:
        data = await BilibiliWebCrawler.fetch_live_room_detail(room_id)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定直播间视频流
@router.get("/fetch_live_videos", response_model=ResponseModel,
            summary="获取直播间视频流/Get live video data of specified room")
async def fetch_collect_folders(request: Request,
                                room_id: str = Query(example="1815229528", description="直播间ID/Live room ID")):
    """
    # [中文]
    ### 用途:
    - 获取指定直播间视频流
    ### 参数:
    - room_id: 直播间ID
    ### 返回:
    - 指定直播间视频流

    # [示例/Example]
    room_id = "1815229528"
    """
    try:
        data = await BilibiliWebCrawler.fetch_live_videos(room_id)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取指定分区正在直播的主播
```

```python
@router.get("/fetch_live_streamers", response_model=ResponseModel,
            summary="获取指定分区正在直播的主播/Get live streamers of specified live area")
async def fetch_collect_folders(request: Request,
                                area_id: str = Query(example="9", description="直播分区id/Live area ID"),
                                pn: int = Query(default=1, description="页码/Page number")):
    """
    # [中文]
    ### 用途:
    - 获取指定分区正在直播的主播
    ### 参数:
    - area_id: 直播分区id
    - pn: 页码
    ### 返回:
    - 指定分区正在直播的主播

    # [示例/Example]
    area_id = "9"
    pn = 1
    """
    try:
        data = await BilibiliWebCrawler.fetch_live_streamers(area_id, pn)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 获取所有直播分区列表
@router.get("/fetch_all_live_areas", response_model=ResponseModel,
            summary="获取所有直播分区列表/Get a list of all live areas")
async def fetch_collect_folders(request: Request,):
    """
    # [中文]
    ### 用途:
    - 获取所有直播分区列表
    ### 参数:
    ### 返回:
    - 所有直播分区列表

    # [示例/Example]
    """
    try:
        data = await BilibiliWebCrawler.fetch_all_live_areas()
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)


# 通过bv号获得视频aid号
@router.get("/bv_to_aid", response_model=ResponseModel, summary="通过bv号获得视频aid号/Generate aid by bvid")
async def fetch_one_video(request: Request,
                          bv_id: str = Query(example="BV1M1421t7hT", description="作品id/Video id")):
    """
    # [中文]
    ### 用途:
    - 通过bv号获得视频aid号
    ### 参数:
    - bv_id: 作品id
    ### 返回:
    - 视频aid号

    # [示例/Example]
    bv_id = "BV1M1421t7hT"
    """
    try:
        data = await BilibiliWebCrawler.bv_to_aid(bv_id)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)
```

```


# 通过bv号获得视频分p信息
@router.get("/fetch_video_parts", response_model=ResponseModel, summary="通过bv号获得视频分p信息/Get Video Parts By bvid")
async def fetch_one_video(request: Request,
                          bv_id: str = Query(example="BV1vf421i7hV", description="作品id/Video id")):
    """
    # [中文]
    ### 用途:
    - 通过bv号获得视频分p信息
    ### 参数:
    - bv_id: 作品id
    ### 返回:
    - 视频分p信息

    # [示例/Example]
    bv_id = "BV1vf421i7hV"
    """
    try:
        data = await BilibiliWebCrawler.fetch_video_parts(bv_id)
        return ResponseModel(code=200,
                             router=request.url.path,
                             data=data)
    except Exception as e:
        status_code = 400
        detail = ErrorResponseModel(code=status_code,
                                    router=request.url.path,
                                    params=dict(request.query_params),
                                    )
        raise HTTPException(status_code=status_code, detail=detail)
```

- download.py

代码块

```
import os
import zipfile
import subprocess
import tempfile

import aiofiles
import httpx
import yaml
from fastapi import APIRouter, Request, Query, HTTPException  # 导入FastAPI组件
```

```python
from starlette.responses import FileResponse

from data_collection_service.app.api.models.APIResponseModel import ErrorResponseModel  # 导入响应模型
from data_collection_service.crawlers.hybrid.hybrid_crawler import HybridCrawler  # 导入混合数据爬虫

router = APIRouter()
HybridCrawler = HybridCrawler()

# 读取上级再上级目录的配置文件
config_path = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.dirname(__file__)))), 'config.yaml')
with open(config_path, 'r', encoding='utf-8') as file:
    config = yaml.safe_load(file)


async def fetch_data(url: str, headers: dict = None):
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    } if headers is None else headers.get('headers')
    async with httpx.AsyncClient() as client:
        response = await client.get(url, headers=headers)
        response.raise_for_status()  # 确保响应是成功的
        return response


# 下载视频专用
async def fetch_data_stream(url: str, request: Request, headers: dict = None, file_path: str = None):
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    } if headers is None else headers.get('headers')
    async with httpx.AsyncClient() as client:
        # 启用流式请求
        async with client.stream("GET", url, headers=headers) as response:
            response.raise_for_status()

            # 流式保存文件
            async with aiofiles.open(file_path, 'wb') as out_file:
                async for chunk in response.aiter_bytes():
                    if await request.is_disconnected():
                        print("客户端断开连接，清理未完成的文件")
                        await out_file.close()
                        os.remove(file_path)
                        return False
                    await out_file.write(chunk)
            return True


async def merge_bilibili_video_audio(video_url: str, audio_url: str, request: Request, output_path: str,
                                     headers: dict) -> bool:
    """
    下载并合并 Bilibili 的视频流和音频流
    """
    try:
        # 创建临时文件
        with tempfile.NamedTemporaryFile(suffix='.m4v', delete=False) as video_temp:
            video_temp_path = video_temp.name
        with tempfile.NamedTemporaryFile(suffix='.m4a', delete=False) as audio_temp:
            audio_temp_path = audio_temp.name

        # 下载视频流
        video_success = await fetch_data_stream(video_url, request, headers=headers, file_path=video_temp_path)
        # 下载音频流
        audio_success = await fetch_data_stream(audio_url, request, headers=headers, file_path=audio_temp_path)

        if not video_success or not audio_success:
            print("Failed to download video or audio stream")
            return False

        # 使用 FFmpeg 合并视频和音频
        ffmpeg_cmd = [
            'ffmpeg', '-y',  # -y 覆盖输出文件
            '-i', video_temp_path,  # 视频输入
            '-i', audio_temp_path,  # 音频输入
            '-c:v', 'copy',  # 复制视频编码，不重新编码
            '-c:a', 'copy',  # 复制音频编码，不重新编码（保持原始质量）
            '-f', 'mp4',  # 确保输出格式为MP4
            output_path
        ]

        print(f"FFmpeg command: {' '.join(ffmpeg_cmd)}")
        result = subprocess.run(ffmpeg_cmd, capture_output=True, text=True)
        print(f"FFmpeg return code: {result.returncode}")
        if result.stderr:
            print(f"FFmpeg stderr: {result.stderr}")
        if result.stdout:
            print(f"FFmpeg stdout: {result.stdout}")

        # 清理临时文件
        try:
            os.unlink(video_temp_path)
            os.unlink(audio_temp_path)
        except:
            pass

        return result.returncode == 0

    except Exception as e:
        # 清理临时文件
        try:
            os.unlink(video_temp_path)
            os.unlink(audio_temp_path)
        except:
            pass
        print(f"Error merging video and audio: {e}")
        return False


@router.get("/download",
            summary="在线下载抖音|TikTok|Bilibili视频/图片/Online download Douyin|TikTok|Bilibili video/image")
async def download_file_hybrid(request: Request,
                               url: str = Query(
                                   example="https://www.douyin.com/video/7372484719365098803",
                                   description="视频或图片的URL地址，支持抖音|TikTok|Bilibili的分享链接，例如：https://v.douyin.com/e4J8Q7A/ 或 https://www.bilibili.com/video/BV1xxxxxxxxx"),
                               prefix: bool = True,
                               with_watermark: bool = False):
    """
    # [中文]
    ### 用途:
    - 在线下载抖音|TikTok|Bilibili 无水印或有水印的视频/图片
    - 通过传入的视频URL参数，获取对应的视频或图片数据，然后下载到本地。
    - 如果你在尝试直接访问TikTok单一视频接口的JSON数据中的视频播放地址时遇到HTTP403错误，那么你可以使用此接口来下载视频。
    - Bilibili视频会自动合并视频流和音频流，确保下载的视频有声音。
    - 这个接口会占用一定的服务器资源，所以在Demo站点是默认关闭的，你可以在本地部署后调用此接口。
    ### 参数:
```

```python
    - url: 视频或图片的URL地址，支持抖音|TikTok|Bilibili的分享链接，例如：https://v.douyin.com/e4J8Q7A/ 或 https://www.bilibili.com/video/BV1xxxxxxxxx
    - prefix: 下载文件的前缀，默认为True，可以在配置文件中修改。
    - with_watermark: 是否下载带水印的视频或图片，默认为False。(注意：Bilibili没有水印概念)
    ### 返回:
    - 返回下载的视频或图片文件响应。

    # [示例/Example]
    url: https://www.bilibili.com/video/BV1U5efz2Egn
    """
    # 是否开启此端点/Whether to enable this endpoint
    if not config["API"]["Download_Switch"]:
        code = 400
        message = "Download endpoint is disabled in the configuration file. | 配置文件中已禁用下载端点。"
        return ErrorResponseModel(code=code, message=message, router=request.url.path,
                                  params=dict(request.query_params))

    # 开始解析数据/Start parsing data
    try:
        data = await HybridCrawler.hybrid_parsing_single_video(url, minimal=True)
    except Exception as e:
        code = 400
        return ErrorResponseModel(code=code, message=str(e), router=request.url.path, params=dict(request.query_params))

    # 开始下载文件/Start downloading files
    try:
        data_type = data.get('type')
        platform = data.get('platform')
        video_id = data.get('video_id')  # 改为使用video_id
        file_prefix = config.get("API").get("Download_File_Prefix") if prefix else ''
        download_path = os.path.join(config.get("API").get("Download_Path"), f"{platform}_{data_type}")

        # 确保目录存在/Ensure the directory exists
        os.makedirs(download_path, exist_ok=True)

        # 下载视频文件/Download video file
        if data_type == 'video':
            file_name = f"{file_prefix}{platform}_{video_id}.mp4" if not with_watermark else f"{file_prefix}{platform}_{video_id}_watermark.mp4"
            file_path = os.path.join(download_path, file_name)

            # 判断文件是否存在，存在就直接返回
            if os.path.exists(file_path):
                return FileResponse(path=file_path, media_type='video/mp4', filename=file_name)

            # 获取对应平台的headers
            if platform == 'tiktok':
                __headers = await HybridCrawler.TikTokWebCrawler.get_tiktok_headers()
            elif platform == 'bilibili':
                __headers = await HybridCrawler.BilibiliWebCrawler.get_bilibili_headers()
            else:  # douyin
                __headers = await HybridCrawler.DouyinWebCrawler.get_douyin_headers()

            # Bilibili 特殊处理：音视频分离
            if platform == 'bilibili':
                video_data = data.get('video_data', {})
                video_url = video_data.get('nwm_video_url_HQ') if not with_watermark else video_data.get(
                    'wm_video_url_HQ')
                audio_url = video_data.get('audio_url')
                if not video_url or not audio_url:
                    raise HTTPException(
                        status_code=500,
                        detail="Failed to get video or audio URL from Bilibili"
                    )

                # 使用专门的函数合并音视频
                success = await merge_bilibili_video_audio(video_url, audio_url, request, file_path,
                                                           __headers.get('headers'))
                if not success:
                    raise HTTPException(
                        status_code=500,
                        detail="Failed to merge Bilibili video and audio streams"
                    )
            else:
                # 其他平台的常规处理
                url = data.get('video_data').get('nwm_video_url_HQ') if not with_watermark else data.get(
                    'video_data').get('wm_video_url_HQ')
                success = await fetch_data_stream(url, request, headers=__headers, file_path=file_path)
                if not success:
                    raise HTTPException(
                        status_code=500,
                        detail="An error occurred while fetching data"
                    )

            # # 保存文件
            # async with aiofiles.open(file_path, 'wb') as out_file:
            #     await out_file.write(response.content)

            # 返回文件内容
            return FileResponse(path=file_path, filename=file_name, media_type="video/mp4")

        # 下载图片文件/Download image file
        elif data_type == 'image':
            # 压缩文件属性/Compress file properties
            zip_file_name = f"{file_prefix}{platform}_{video_id}_images.zip" if not with_watermark else f"{file_prefix}{platform}_{video_id}_images_watermark.zip"
            zip_file_path = os.path.join(download_path, zip_file_name)

            # 判断文件是否存在，存在就直接返回、
            if os.path.exists(zip_file_path):
                return FileResponse(path=zip_file_path, filename=zip_file_name, media_type="application/zip")

            # 获取图片文件/Get image file
            urls = data.get('image_data').get('no_watermark_image_list') if not with_watermark else data.get(
                'image_data').get('watermark_image_list')
            image_file_list = []
            for url in urls:
                # 请求图片文件/Request image file
                response = await fetch_data(url)
                index = int(urls.index(url))
                content_type = response.headers.get('content-type')
                file_format = content_type.split('/')[1]
                file_name = f"{file_prefix}{platform}_{video_id}_{index + 1}.{file_format}" if not with_watermark else f"{file_prefix}{platform}_{video_id}_{index + 1}_watermark.{file_format}"
                file_path = os.path.join(download_path, file_name)
                image_file_list.append(file_path)

                # 保存文件/Save file
                async with aiofiles.open(file_path, 'wb') as out_file:
                    await out_file.write(response.content)

            # 压缩文件/Compress file
            with zipfile.ZipFile(zip_file_path, 'w') as zip_file:
                for image_file in image_file_list:
                    zip_file.write(image_file, os.path.basename(image_file))

            # 返回压缩文件/Return compressed file
            return FileResponse(path=zip_file_path, filename=zip_file_name, media_type="application/zip")

    # 异常处理/Exception handling
    except Exception as e:
        print(e)
        code = 400
        return ErrorResponseModel(code=code, message=str(e), router=request.url.path, params=dict(request.query_params))
```

models

- APIResponseModel.py

代码块

```
from fastapi import Body, FastAPI, Query, Request, HTTPException
from pydantic import BaseModel
from typing import Any, Callable, Type, Optional, Dict
from functools import wraps
import datetime

app = FastAPI()


# 定义响应模型
class ResponseModel(BaseModel):
    code: int = 200
    router: str = "Endpoint path"
    data: Optional[Any] = {}


# 定义错误响应模型
class ErrorResponseModel(BaseModel):
    code: int = 400
    message: str = "An error occurred."
    support: str = "Please contact us on Github: https://github.com/Evil0ctal/Douyin_TikTok_Download_API"
    time: str = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    router: str
    params: dict = {}


# 混合解析响应模型
class HybridResponseModel(BaseModel):
    code: int = 200
    router: str = "Hybrid parsing single video endpoint"
    data: Optional[Any] = {}


# iOS_Shortcut响应模型
class iOS_Shortcut(BaseModel):
    version: str
    update: str
    link: str
    link_en: str
    note: str
    note_en: str
```

crawler

bilibili

- config.yaml

代码块

```
TokenManager:
  bilibili:
    headers:
      'accept-language': zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
      'origin': https://www.bilibili.com
      'referer': https://space.bilibili.com/
      'origin_2': https://space.bilibili.com
      'cookie': DedeUserID=292039314; DedeUserID__ckMd5=89698fb49e523c9f;
SESSDATA=f4c1988b%2C1783510832%2Cc1ccd%2A11CjA7my_d4M4BOb2moHhApExeeDewhjrsESg4
2o7MhCls4HB5jyteQSGNhH7mFhf9v-
YSVjR1aGllTXI0N2loU25UWUJNYklVb0NCdUNQN0p3MkJfTTZVUXNTT2xvZVBCaEtaWHh6UjJOdTZIW
mJfOHMzUTdDQmZmTVBHRzRVZTZyNmpra1BWaUtRIIEC;
bili_jct=0f31589ace86c64a8261f09fbc272321; sid=q6g0j28t
      'user-agent': Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
Edg/143.0.0.0

```

```

    proxies:
      http:
      https:
```

- endpoints.py

代码块

```python
class BilibiliAPIEndpoints:

    "-------------------------------------------------------域名-domain-----------------------------------------------------------------------"
    # 哔哩哔哩接口域名
    BILIAPI_DOMAIN = "https://api.bilibili.com"

    # 哔哩哔哩直播域名
    LIVE_DOMAIN = "https://api.live.bilibili.com"

    "-------------------------------------------------------接口-api------------------------------------------------------------------------"
    # 作品信息 (Post Detail)
    POST_DETAIL = f"{BILIAPI_DOMAIN}/x/web-interface/view"

    # 作品视频流
    VIDEO_PLAYURL = f"{BILIAPI_DOMAIN}/x/player/wbi/playurl"

    # 用户发布视频作品数据
    USER_POST = f"{BILIAPI_DOMAIN}/x/space/wbi/arc/search"

    # 收藏夹列表
    COLLECT_FOLDERS = f"{BILIAPI_DOMAIN}/x/v3/fav/folder/created/list-all"

    # 收藏夹视频
    COLLECT_VIDEOS = f"{BILIAPI_DOMAIN}/x/v3/fav/resource/list"

    # 用户个人信息
    USER_DETAIL = f"{BILIAPI_DOMAIN}/x/space/wbi/acc/info"

    # 综合热门
    COM_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular"

    # 每周必看
    WEEKLY_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular/series/one"

    # 入站必刷
    PRECIOUS_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular/precious"

    # 视频评论
    VIDEO_COMMENTS = f"{BILIAPI_DOMAIN}/x/v2/reply"

    # 用户动态
    USER_DYNAMIC = f"{BILIAPI_DOMAIN}/x/polymer/web-dynamic/v1/feed/space"

    # 评论的回复
    COMMENT_REPLY = f"{BILIAPI_DOMAIN}/x/v2/reply/reply"

    # 视频分p信息
    VIDEO_PARTS = f"{BILIAPI_DOMAIN}/x/player/pagelist"

    # 直播间信息
    LIVEROOM_DETAIL = f"{LIVE_DOMAIN}/room/v1/Room/get_info"

    # 直播分区列表
    LIVE_AREAS = f"{LIVE_DOMAIN}/room/v1/Area/getList"

    # 直播间视频流
    LIVE_VIDEOS = f"{LIVE_DOMAIN}/room/v1/Room/playUrl"

    # 正在直播的主播
    LIVE_STREAMER = f"{LIVE_DOMAIN}/xlive/web-interface/v1/second/getList"
```

- models.py

代码块

```python
import time
from pydantic import BaseModel


class BaseRequestsModel(BaseModel):
    wts: str = str(round(time.time()))


class UserPostVideos(BaseRequestsModel):
    dm_img_inter: str = '{"ds":[],"wh":[3557,5674,5],"of":[154,308,154]}'
    dm_img_list: list = []
    mid: str
    pn: int
    ps: str = "20"


```

```python
class UserProfile(BaseRequestsModel):
    mid: str


class UserDynamic(BaseRequestsModel):
    host_mid: str
    offset: str
    wts: str = str(round(time.time()))


class ComPopular(BaseRequestsModel):
    pn: int
    ps: str = "20"
    web_location: str = "333.934"


class PlayUrl(BaseRequestsModel):
    qn: str
    fnval: str = '4048'
    bvid: str
    cid: str
```

- utils.py

代码块

```python
from urllib.parse import urlencode
from data_collection_service.crawlers.bilibili import wrid
from data_collection_service.crawlers.utils.logger import logger
from data_collection_service.crawlers.bilibili.endpoints import BilibiliAPIEndpoints

class EndpointGenerator:
    def __init__(self, params: dict):
        self.params = params

    # 获取用户发布视频作品数据 生成enpoint
    async def user_post_videos_endpoint(self) -> str:
        # 添加w_rid
        endpoint = await WridManager.wrid_model_endpoint(params=self.params)
        # 拼接成最终结果并返回
        final_endpoint = BilibiliAPIEndpoints.USER_POST + '?' + endpoint
        return final_endpoint

    # 获取视频流地址 生成enpoint
    async def video_playurl_endpoint(self) -> str:
        # 添加w_rid
        endpoint = await WridManager.wrid_model_endpoint(params=self.params)
        # 拼接成最终结果并返回
        final_endpoint = BilibiliAPIEndpoints.VIDEO_PLAYURL + '?' + endpoint
        return final_endpoint

    # 获取指定用户的信息 生成enpoint
    async def user_profile_endpoint(self) -> str:
        # 添加w_rid
        endpoint = await WridManager.wrid_model_endpoint(params=self.params)
        # 拼接成最终结果并返回
        final_endpoint = BilibiliAPIEndpoints.USER_DETAIL + '?' + endpoint
        return final_endpoint

    # 获取综合热门视频信息 生成enpoint
    async def com_popular_endpoint(self) -> str:
        # 添加w_rid
        endpoint = await WridManager.wrid_model_endpoint(params=self.params)
        # 拼接成最终结果并返回
        final_endpoint = BilibiliAPIEndpoints.COM_POPULAR + '?' + endpoint
        return final_endpoint

    # 获取指定用户动态
    async def user_dynamic_endpoint(self):
        # 添加w_rid
        endpoint = await WridManager.wrid_model_endpoint(params=self.params)
        # 拼接成最终结果并返回
        final_endpoint = BilibiliAPIEndpoints.USER_DYNAMIC + '?' + endpoint
        return final_endpoint


class WridManager:
    @classmethod
    async def get_encode_query(cls, params: dict) -> str:
        params['wts'] = params['wts'] + "ea1db124af3c7062474693fa704f4ff8"
        params = dict(sorted(params.items()))  # 按照 key 重排参数
        # 过滤 value 中的 "!'()*" 字符
        params = {
            k: ''.join(filter(lambda chr: chr not in "!'()*", str(v)))
            for k, v
            in params.items()
        }
        query = urlencode(params)  # 序列化参数
        return query

    @classmethod
    async def wrid_model_endpoint(cls, params: dict) -> str:
        wts = params["wts"]
        encode_query = await cls.get_encode_query(params)
        # 获取w_rid参数
        w_rid = wrid.get_wrid(e=encode_query)
        params["wts"] = wts
        params["w_rid"] = w_rid
        return "&".join(f"{k}={v}" for k, v in params.items())

# BV号转为对应av号
async def bv2av(bv_id: str) -> int:
    table = "fZodR9XQDSUm21yCkr6zBqiveYah8bt4xsWpHnJE7jL5VG3guMTKNPAwcF"
    s = [11, 10, 3, 8, 4, 6, 2, 9, 5, 7]
    xor = 177451812
    add_105 = 8728348608
    add_all = 8728348608 - (2 ** 31 - 1) - 1
    tr = [0] * 128
    for i in range(58):
        tr[ord(table[i])] = i
    r = 0
    for i in range(6):
        r += tr[ord(bv_id[s[i]])] * (58 ** i)
    add = add_105
    if r < add:
        add = add_all
    aid = (r - add) ^ xor
    return aid

# 响应分析
class ResponseAnalyzer:
    # 用户收藏夹信息
    @classmethod
    async def collect_folders_analyze(cls, response: dict) -> dict:
        if response['data']:
            return response
        else:
            logger.warning("该用户收藏夹为空/用户设置为不可见")
            return {"code": 1, "message": "该用户收藏夹为空/用户设置为不可见"}
```

- web_crawler.py

代码块

```python
import asyncio  # 异步I/O
import os  # 系统操作
import time  # 时间操作
import yaml  # 配置文件
```

```python


# 基础爬虫客户端和哔哩哔哩API端点
from data_collection_service.crawlers.base_crawler import BaseCrawler
from data_collection_service.crawlers.bilibili.endpoints import
BilibiliAPIEndpoints
# 哔哩哔哩工具类
from data_collection_service.crawlers.bilibili.utils import EndpointGenerator,
bv2av, ResponseAnalyzer
# 数据请求模型
from data_collection_service.crawlers.bilibili.models import UserPostVideos,
UserProfile, ComPopular, UserDynamic, PlayUrl

# 配置文件路径
path = os.path.abspath(os.path.dirname(__file__))

# 读取配置文件
with open(f"{path}/config.yaml", "r", encoding="utf-8") as f:
    config = yaml.safe_load(f)


class BilibiliWebCrawler:

    # 从配置文件读取哔哩哔哩请求头
    async def get_bilibili_headers(self):
        bili_config = config['TokenManager']['bilibili']
        kwargs = {
            "headers": {
                "accept-language": bili_config["headers"]["accept-language"],
                "origin": bili_config["headers"]["origin"],
                "referer": bili_config["headers"]["referer"],
                "user-agent": bili_config["headers"]["user-agent"],
                "cookie": bili_config["headers"]["cookie"],
            },
            "proxies": {"http://": bili_config["proxies"]["http"], "https://":
bili_config["proxies"]["https"]},
        }
        return kwargs

    "-------------------------------------------------------handler接口列表-----
---------------------------------------------------"

    # 获取单个视频详情信息
    async def fetch_one_video(self, bv_id: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.POST_DETAIL}?bvid={bv_id}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取视频流地址
    async def fetch_video_playurl(self, bv_id: str, cid: str, qn: str = "64") -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 通过模型生成基本请求参数
            params = PlayUrl(bvid=bv_id, cid=cid, qn=qn)
            # 创建请求endpoint
            generator = EndpointGenerator(params.dict())
            endpoint = await generator.video_playurl_endpoint()
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取用户发布视频作品数据
    async def fetch_user_post_videos(self, uid: str, pn: int) -> dict:
        """
        :param uid: 用户uid
        :param pn: 页码
        :return:
        """
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 通过模型生成基本请求参数
            params = UserPostVideos(mid=uid, pn=pn)
            # 创建请求endpoint
            generator = EndpointGenerator(params.dict())
            endpoint = await generator.user_post_videos_endpoint()
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取用户所有收藏夹信息
    async def fetch_collect_folders(self, uid: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.COLLECT_FOLDERS}?up_mid={uid}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        # 分析响应结果
        result_dict = await ResponseAnalyzer.collect_folders_analyze(response=response)
        return result_dict

    # 获取指定收藏夹内视频数据
    async def fetch_folder_videos(self, folder_id: str, pn: int) -> dict:
        """
        :param folder_id: 收藏夹id-- 可从<获取用户所有收藏夹信息>获得
        :param pn: 页码
        :return:
        """
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        # 发送请求，获取请求响应结果
        async with base_crawler as crawler:
            endpoint = f"{BilibiliAPIEndpoints.COLLECT_VIDEOS}?media_id={folder_id}&pn={pn}&ps=20&keyword=&order=mtime&type=0&tid=0&platform=web"
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取指定用户的信息
    async def fetch_user_profile(self, uid: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 通过模型生成基本请求参数
            params = UserProfile(mid=uid)
            # 创建请求endpoint
            generator = EndpointGenerator(params.dict())
            endpoint = await generator.user_profile_endpoint()
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取综合热门视频信息
    async def fetch_com_popular(self, pn: int) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 通过模型生成基本请求参数
            params = ComPopular(pn=pn)
            # 创建请求endpoint
            generator = EndpointGenerator(params.dict())
            endpoint = await generator.com_popular_endpoint()
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取指定视频的评论
    async def fetch_video_comments(self, bv_id: str, pn: int) -> dict:
        # 评论排序 -- 1:按点赞数排序. 0:按时间顺序排序
        sort = 1
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.VIDEO_COMMENTS}?type=1&oid={bv_id}&sort={sort}&nohot=0&ps=20&pn={pn}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取视频下指定评论的回复
    async def fetch_comment_reply(self, bv_id: str, pn: int, rpid: str) -> dict:
        """
        :param bv_id: 目标视频bv号
        :param pn: 页码
        :param rpid: 目标评论id，可通过fetch_video_comments获得
        :return:
        """
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.COMMENT_REPLY}?type=1&oid={bv_id}&root={rpid}&&ps=20&pn={pn}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
            return response

    # 获取指定用户动态
    async def fetch_user_dynamic(self, uid: str, offset: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 通过模型生成基本请求参数
            params = UserDynamic(host_mid=uid, offset=offset)
            # 创建请求endpoint
            generator = EndpointGenerator(params.dict())
            endpoint = await generator.user_dynamic_endpoint()
            print(endpoint)
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取视频实时弹幕
    async def fetch_video_danmaku(self, cid: str):
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"https://comment.bilibili.com/{cid}.xml"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_response(endpoint)
        return response.text

    # 获取指定直播间信息
    async def fetch_live_room_detail(self, room_id: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.LIVEROOM_DETAIL}?room_id={room_id}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取指定直播间视频流
    async def fetch_live_videos(self, room_id: str) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.LIVE_VIDEOS}?cid={room_id}&quality=4"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取指定分区正在直播的主播
    async def fetch_live_streamers(self, area_id: str, pn: int):
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.LIVE_STREAMER}?platform=web&parent_area_id={area_id}&page={pn}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    "-------------------------------------------------------utils接口列表-------
-------------------------------------------------"

    # 通过bv号获得视频aid号
    async def bv_to_aid(self, bv_id: str) -> int:
        aid = await bv2av(bv_id=bv_id)
        return aid

    # 通过bv号获得视频分p信息
    async def fetch_video_parts(self, bv_id: str) -> str:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"],
crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = f"{BilibiliAPIEndpoints.VIDEO_PARTS}?bvid={bv_id}"
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

    # 获取所有直播分区列表
    async def fetch_all_live_areas(self) -> dict:
        # 获取请求头信息
        kwargs = await self.get_bilibili_headers()
        # 创建基础爬虫对象
        base_crawler = BaseCrawler(proxies=kwargs["proxies"],
crawler_headers=kwargs["headers"])
        async with base_crawler as crawler:
            # 创建请求endpoint
            endpoint = BilibiliAPIEndpoints.LIVE_AREAS
            # 发送请求，获取请求响应结果
            response = await crawler.fetch_get_json(endpoint)
        return response

if __name__ == '__main__':
    # 初始化
    BilibiliWebCrawler = BilibiliWebCrawler()

    # 开始时间
    start = time.time()

    asyncio.run(BilibiliWebCrawler.main())

    # 结束时间
    end = time.time()
    print(f"耗时: {end - start}")
```

- wrid.py

代码块

```
import urllib.parse

def srotl(t, e):
    return (t << e) | (t >> (32 - e))

def tendian(t):
    if isinstance(t, int):
        return (16711935 & srotl(t, 8)) | (4278255360 & srotl(t, 24))
    for e in range(len(t)):
        t[e] = tendian(t[e])
    return t

# 没问题
def tbytes_to_words(t):
    e = []
    r = 0
    for n in range(len(t)):
        if r >> 5 >= len(e):
            e.append(0)
        e[r >> 5] |= t[n] << (24 - r % 32)
        r += 8
    return e

def jbinstring_to_bytes(t):
    e = []
    for n in range(len(t)):
        e.append(ord(t[n]) & 255)
    return e

# 没问题
def estring_to_bytes(t):
    return jbinstring_to_bytes(urllib.parse.unquote(urllib.parse.quote(t)))

def _ff(t, e, n, r, o, i, a):
    # 计算中间值 c
    c = t + ((e & n) | (~e & r)) + (o & 0xFFFFFFFF) + a
    # 将 c 转换为 32 位无符号整数
    c = c & 0xFFFFFFFF
    # 左移和右移操作
    c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
```

```python
    # 返回结果
    return (c + e) & 0xFFFFFFFF

def _gg(t, e, n, r, o, i, a):
    # 计算中间值 c
    c = t + ((e & r) | (n & ~r)) + (o & 0xFFFFFFFF) + a
    # 将 c 转换为 32 位无符号整数
    c = c & 0xFFFFFFFF
    # 左移和右移操作
    c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
    # 返回结果
    return (c + e) & 0xFFFFFFFF

def _hh(t, e, n, r, o, i, a):
    # 计算中间值 c
    c = t + (e ^ n ^ r) + (o & 0xFFFFFFFF) + a
    # 将 c 转换为 32 位无符号整数
    c = c & 0xFFFFFFFF
    # 左移和右移操作
    c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
    # 返回结果
    return (c + e) & 0xFFFFFFFF

def _ii(t, e, n, r, o, i, a):
    # 计算中间值 c
    c = t + (n ^ (e | ~r)) + (o & 0xFFFFFFFF) + a
    # 将 c 转换为 32 位无符号整数
    c = c & 0xFFFFFFFF
    # 左移和右移操作
    c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
    # 返回结果
    return (c + e) & 0xFFFFFFFF

def o(i, a):
    if isinstance(i, str):
        i = estring_to_bytes(i)
    elif isinstance(i, (list, tuple)):
        i = list(i)
    elif not isinstance(i, (list, bytearray)):
        i = str(i)
    c = tbytes_to_words(i)
    u = 8 * len(i)
    s, l, f, p = 1732584193, -271733879, -1732584194, 271733878

    for d in range(len(c)):
        c[d] = (16711935 & (c[d] << 8 | c[d] >> 24)) | (4278255360 & (c[d] <<
24 | c[d] >> 8))
```

```python

    # 确保列表 c 的长度足够大
    while len(c) <= (14 + ((u + 64 >> 9) << 4)):
        c.append(0)

    c[u >> 5] |= 128 << (u % 32)
    c[14 + ((u + 64 >> 9) << 4)] = u

    h, v, y, m = _ff, _gg, _hh, _ii
    for d in range(0, len(c), 16):
        g, b, w, A = s, l, f, p
        # 确保在访问索引之前扩展列表的长度
        while len(c) <= d + 15:
            c.append(0)
        s = h(s, l, f, p, c[d + 0], 7, -680876936)
        p = h(p, s, l, f, c[d + 1], 12, -389564586)
        f = h(f, p, s, l, c[d + 2], 17, 606105819)
        l = h(l, f, p, s, c[d + 3], 22, -1044525330)
        s = h(s, l, f, p, c[d + 4], 7, -176418897)
        p = h(p, s, l, f, c[d + 5], 12, 1200080426)
        f = h(f, p, s, l, c[d + 6], 17, -1473231341)
        l = h(l, f, p, s, c[d + 7], 22, -45705983)
        s = h(s, l, f, p, c[d + 8], 7, 1770035416)
        p = h(p, s, l, f, c[d + 9], 12, -1958414417)
        f = h(f, p, s, l, c[d + 10], 17, -42063)
        l = h(l, f, p, s, c[d + 11], 22, -1990404162)
        s = h(s, l, f, p, c[d + 12], 7, 1804603682)
        p = h(p, s, l, f, c[d + 13], 12, -40341101)
        f = h(f, p, s, l, c[d + 14], 17, -1502002290)
        s = v(s, l := h(l, f, p, s, c[d + 15], 22, 1236535329), f, p, c[d + 1], 5, -165796510)
        p = v(p, s, l, f, c[d + 6], 9, -1069501632)
        f = v(f, p, s, l, c[d + 11], 14, 643717713)
        l = v(l, f, p, s, c[d + 0], 20, -373897302)
        s = v(s, l, f, p, c[d + 5], 5, -701558691)
        p = v(p, s, l, f, c[d + 10], 9, 38016083)
        f = v(f, p, s, l, c[d + 15], 14, -660478335)
        l = v(l, f, p, s, c[d + 4], 20, -405537848)
        s = v(s, l, f, p, c[d + 9], 5, 568446438)
        p = v(p, s, l, f, c[d + 14], 9, -1019803690)
        f = v(f, p, s, l, c[d + 3], 14, -187363961)
        l = v(l, f, p, s, c[d + 8], 20, 1163531501)
        s = v(s, l, f, p, c[d + 13], 5, -1444681467)
        p = v(p, s, l, f, c[d + 2], 9, -51403784)
        f = v(f, p, s, l, c[d + 7], 14, 1735328473)
        s = y(s, l := v(l, f, p, s, c[d + 12], 20, -1926607734), f, p, c[d + 5], 4, -378558)
        p = y(p, s, l, f, c[d + 8], 11, -2022574463)
        f = y(f, p, s, l, c[d + 11], 16, 1839030562)
        l = y(l, f, p, s, c[d + 14], 23, -35309556)
        s = y(s, l, f, p, c[d + 1], 4, -1530992060)
        p = y(p, s, l, f, c[d + 4], 11, 1272893353)
        f = y(f, p, s, l, c[d + 7], 16, -155497632)
        l = y(l, f, p, s, c[d + 10], 23, -1094730640)
        s = y(s, l, f, p, c[d + 13], 4, 681279174)
        p = y(p, s, l, f, c[d + 0], 11, -358537222)
        f = y(f, p, s, l, c[d + 3], 16, -722521979)
        l = y(l, f, p, s, c[d + 6], 23, 76029189)
        s = y(s, l, f, p, c[d + 9], 4, -640364487)
        p = y(p, s, l, f, c[d + 12], 11, -421815835)
        f = y(f, p, s, l, c[d + 15], 16, 530742520)
        s = m(s, l := y(l, f, p, s, c[d + 2], 23, -995338651), f, p, c[d + 0], 6, -198630844)
        p = m(p, s, l, f, c[d + 7], 10, 1126891415)
        f = m(f, p, s, l, c[d + 14], 15, -1416354905)
        l = m(l, f, p, s, c[d + 5], 21, -57434055)
        s = m(s, l, f, p, c[d + 12], 6, 1700485571)
        p = m(p, s, l, f, c[d + 3], 10, -1894986606)
        f = m(f, p, s, l, c[d + 10], 15, -1051523)
        l = m(l, f, p, s, c[d + 1], 21, -2054922799)
        s = m(s, l, f, p, c[d + 8], 6, 1873313359)
        p = m(p, s, l, f, c[d + 15], 10, -30611744)
        f = m(f, p, s, l, c[d + 6], 15, -1560198380)
        l = m(l, f, p, s, c[d + 13], 21, 1309151649)
        s = m(s, l, f, p, c[d + 4], 6, -145523070)
        p = m(p, s, l, f, c[d + 11], 10, -1120210379)
        f = m(f, p, s, l, c[d + 2], 15, 718787259)
        l = m(l, f, p, s, c[d + 9], 21, -343485551)

        s = (s + g) >> 0 & 0xFFFFFFFF
        l = (l + b) >> 0 & 0xFFFFFFFF
        f = (f + w) >> 0 & 0xFFFFFFFF
        p = (p + A) >> 0 & 0xFFFFFFFF

    return tendian([s, l, f, p])

def twords_to_bytes(t):
    e = []
    for n in range(0, 32 * len(t), 8):
        e.append((t[n >> 5] >> (24 - n % 32)) & 255)
    return e

def tbytes_to_hex(t):
    e = []
    for n in range(len(t)):
        e.append(hex(t[n] >> 4)[2:])
        e.append(hex(t[n] & 15)[2:])
    return ''.join(e)

def get_wrid(e):
    n = None
    i = twords_to_bytes(o(e, n))
    return tbytes_to_hex(i)
```

hybrid

- hybrid_crawler.py

代码块

```
import asyncio
import re
import httpx

# from crawlers.douyin.web.web_crawler import DouyinWebCrawler  # 导入抖音Web爬虫
# from crawlers.tiktok.web.web_crawler import TikTokWebCrawler  # 导入TikTok Web爬虫
# from crawlers.tiktok.app.app_crawler import TikTokAPPCrawler  # 导入TikTok App爬虫
from data_collection_service.crawlers.bilibili.web_crawler import BilibiliWebCrawler  # 导入Bilibili Web爬虫


class HybridCrawler:
    def __init__(self):
        # self.DouyinWebCrawler = DouyinWebCrawler()
        # self.TikTokWebCrawler = TikTokWebCrawler()
        # self.TikTokAPPCrawler = TikTokAPPCrawler()
        self.BilibiliWebCrawler = BilibiliWebCrawler()

    async def get_bilibili_bv_id(self, url: str) -> str:
        """
        从 Bilibili URL 中提取 BV 号，支持短链重定向
        """
        # 如果是 b23.tv 短链，需要重定向获取真实URL
        if "b23.tv" in url:
            async with httpx.AsyncClient() as client:
                response = await client.head(url, follow_redirects=True)
                url = str(response.url)

        # 从URL中提取BV号
```

```python
        bv_pattern = r'(?:video\/|\/)(BV[A-Za-z0-9]+)'
        match = re.search(bv_pattern, url)
        if match:
            return match.group(1)
        else:
            raise ValueError(f"Cannot extract BV ID from URL: {url}")

    async def hybrid_parsing_single_video(self, url: str, minimal: bool = False):
        # 解析抖音视频/Parse Douyin video
        if "douyin" in url:
            platform = "douyin"
            aweme_id = await self.DouyinWebCrawler.get_aweme_id(url)
            data = await self.DouyinWebCrawler.fetch_one_video(aweme_id)
            data = data.get("aweme_detail")
            # $.aweme_detail.aweme_type
            aweme_type = data.get("aweme_type")
        # 解析TikTok视频/Parse TikTok video
        elif "tiktok" in url:
            platform = "tiktok"
            aweme_id = await self.TikTokWebCrawler.get_aweme_id(url)

            # 2024-09-14: Switch to TikTokAPPCrawler instead of TikTokWebCrawler
            # data = await self.TikTokWebCrawler.fetch_one_video(aweme_id)
            # data = data.get("itemInfo").get("itemStruct")

            data = await self.TikTokAPPCrawler.fetch_one_video(aweme_id)
            # $.imagePost exists if aweme_type is photo
            aweme_type = data.get("aweme_type")
        # 解析Bilibili视频/Parse Bilibili video
        elif "bilibili" in url or "b23.tv" in url:
            platform = "bilibili"
            aweme_id = await self.get_bilibili_bv_id(url)  # BV号作为统一的video_id
            response = await self.BilibiliWebCrawler.fetch_one_video(aweme_id)
            data = response.get('data', {})  # 提取data部分
            # Bilibili只有视频类型，aweme_type设为0(video)
            aweme_type = 0
        else:
            raise ValueError("hybrid_parsing_single_video: Cannot judge the video source from the URL.")

        # 检查是否需要返回最小数据/Check if minimal data is required
        if not minimal:
            return data

        # 如果是最小数据，处理数据/If it is minimal data, process the data
        url_type_code_dict = {
            # common
            0: 'video',
            # Douyin
            2: 'image',
            4: 'video',
            68: 'image',
            # TikTok
            51: 'video',
            55: 'video',
            58: 'video',
            61: 'video',
            150: 'image'
        }
        # 判断链接类型/Judge link type
        url_type = url_type_code_dict.get(aweme_type, 'video')
        # print(f"url_type: {url_type}")

        """
        以下为(视频||图片)数据处理的四个方法,如果你需要自定义数据处理请在这里修改.
        The following are four methods of (video || image) data processing.
        If you need to customize data processing, please modify it here.
        """

        """
        创建已知数据字典(索引相同)，稍后使用.update()方法更新数据
        Create a known data dictionary (index the same),
        and then use the .update() method to update the data
        """

        # 根据平台适配字段映射
        if platform == 'bilibili':
            result_data = {
                'type': url_type,
                'platform': platform,
                'video_id': aweme_id,
                'desc': data.get("title"),  # Bilibili使用title
                'create_time': data.get("pubdate"),  # Bilibili使用pubdate
                'author': data.get("owner"),  # Bilibili使用owner
                'music': None,  # Bilibili没有音乐信息
                'statistics': data.get("stat"),  # Bilibili使用stat
                'cover_data': {},  # 将在各平台处理中填充
                'hashtags': None,  # Bilibili没有hashtags概念
            }
        else:
            result_data = {
                'type': url_type,
                'platform': platform,
                'video_id': aweme_id,  # 统一使用video_id字段，内容可能是aweme_id或bv_id
                'desc': data.get("desc"),
                'create_time': data.get("create_time"),
                'author': data.get("author"),
                'music': data.get("music"),
                'statistics': data.get("statistics"),
                'cover_data': {},  # 将在各平台处理中填充
                'hashtags': data.get('text_extra'),
            }
        # 创建一个空变量，稍后使用.update()方法更新数据/Create an empty variable and use the .update() method to update the data
        api_data = None
        # 判断链接类型并处理数据/Judge link type and process data
        # 抖音数据处理/Douyin data processing
        if platform == 'douyin':
            # 填充封面数据
            result_data['cover_data'] = {
                'cover': data.get("video", {}).get("cover"),
                'origin_cover': data.get("video", {}).get("origin_cover"),
                'dynamic_cover': data.get("video", {}).get("dynamic_cover")
            }
            # 抖音视频数据处理/Douyin video data processing
            if url_type == 'video':
                # 将信息储存在字典中/Store information in a dictionary
                uri = data['video']['play_addr']['uri']
                wm_video_url_HQ = data['video']['play_addr']['url_list'][0]
                wm_video_url = f"https://aweme.snssdk.com/aweme/v1/playwm/?video_id={uri}&radio=1080p&line=0"
                nwm_video_url_HQ = wm_video_url_HQ.replace('playwm', 'play')
                nwm_video_url = f"https://aweme.snssdk.com/aweme/v1/play/?video_id={uri}&ratio=1080p&line=0"
                api_data = {
                    'video_data':
                        {
                            'wm_video_url': wm_video_url,
                            'wm_video_url_HQ': wm_video_url_HQ,
                            'nwm_video_url': nwm_video_url,
                            'nwm_video_url_HQ': nwm_video_url_HQ
                        }
                }
            # 抖音图片数据处理/Douyin image data processing
            elif url_type == 'image':
                # 无水印图片列表/No watermark image list
                no_watermark_image_list = []
                # 有水印图片列表/With watermark image list
                watermark_image_list = []
                # 遍历图片列表/Traverse image list
                for i in data['images']:
                    no_watermark_image_list.append(i['url_list'][0])
                    watermark_image_list.append(i['download_url_list'][0])
                api_data = {
                    'image_data':
                        {
                            'no_watermark_image_list': no_watermark_image_list,
                            'watermark_image_list': watermark_image_list
                        }
                }
        # TikTok数据处理/TikTok data processing
        elif platform == 'tiktok':
            # 填充封面数据
            result_data['cover_data'] = {
                'cover': data.get("video", {}).get("cover"),
                'origin_cover': data.get("video", {}).get("origin_cover"),
                'dynamic_cover': data.get("video", {}).get("dynamic_cover")
            }
            # TikTok视频数据处理/TikTok video data processing
            if url_type == 'video':
                # 将信息储存在字典中/Store information in a dictionary
                # wm_video = data['video']['downloadAddr']
                # wm_video = data['video']['download_addr']['url_list'][0]
                wm_video = (
                    data.get('video', {})
                    .get('download_addr', {})
                    .get('url_list', [None])[0]
                )

                api_data = {
                    'video_data':
                        {
                            'wm_video_url': wm_video,
                            'wm_video_url_HQ': wm_video,
                            # 'nwm_video_url': data['video']['playAddr'],
                            'nwm_video_url': data['video']['play_addr']['url_list'][0],
                            # 'nwm_video_url_HQ': data['video']['bitrateInfo'][0]['PlayAddr']['UrlList'][0]
                            'nwm_video_url_HQ': data['video']['bit_rate'][0]['play_addr']['url_list'][0]
                        }
                }
            # TikTok图片数据处理/TikTok image data processing
```

```python
            elif url_type == 'image':
                # 无水印图片列表/No watermark image list
                no_watermark_image_list = []
                # 有水印图片列表/With watermark image list
                watermark_image_list = []
                for i in data['image_post_info']['images']:
                    no_watermark_image_list.append(i['display_image']['url_list'][0])
                    watermark_image_list.append(i['owner_watermark_image']['url_list'][0])
                api_data = {
                    'image_data':
                        {
                            'no_watermark_image_list': no_watermark_image_list,
                            'watermark_image_list': watermark_image_list
                        }
                }
        # Bilibili数据处理/Bilibili data processing
        elif platform == 'bilibili':
            # 填充封面数据
            result_data['cover_data'] = {
                'cover': data.get("pic"),  # Bilibili使用pic作为封面
                'origin_cover': data.get("pic"),
                'dynamic_cover': data.get("pic")
            }
            # Bilibili只有视频，直接处理视频数据
            if url_type == 'video':
                # 获取视频播放地址需要额外调用API
                cid = data.get('cid')  # 获取cid
                if cid:
                    # 获取播放链接，cid需要转换为字符串
                    playurl_data = await self.BilibiliWebCrawler.fetch_video_playurl(aweme_id, str(cid))
                    # 从播放数据中提取URL
                    dash = playurl_data.get('data', {}).get('dash', {})
                    video_list = dash.get('video', [])
                    audio_list = dash.get('audio', [])

                    # 选择最高质量的视频流
                    video_url = video_list[0].get('baseUrl') if video_list else None
                    audio_url = audio_list[0].get('baseUrl') if audio_list else None

                    api_data = {
                        'video_data': {
                            'wm_video_url': video_url,
                            'wm_video_url_HQ': video_url,
                            'nwm_video_url': video_url,  # Bilibili没有水印概念
                            'nwm_video_url_HQ': video_url,
                            'audio_url': audio_url,  # Bilibili音视频分离
                            'cid': cid,  # 保存cid供后续使用
                        }
                    }
                else:
                    api_data = {
                        'video_data': {
                            'wm_video_url': None,
                            'wm_video_url_HQ': None,
                            'nwm_video_url': None,
                            'nwm_video_url_HQ': None,
                            'error': 'Failed to get cid for video playback'
                        }
                    }
        # 更新数据/Update data
        result_data.update(api_data)
        return result_data

    async def main(self):
        # 测试混合解析单一视频接口/Test hybrid parsing single video endpoint
        # url = "https://v.douyin.com/L4FJNR3/"
        # url = "https://www.tiktok.com/@taylorswift/video/7359655005701311786"
        url = "https://www.tiktok.com/@flukegk83/video/7360734489271700753"
        # url = "https://www.tiktok.com/@minecraft/photo/7369296852669205791"
        minimal = True
        result = await self.hybrid_parsing_single_video(url, minimal=minimal)
        # print(result)

        # 占位
        pass


if __name__ == '__main__':
    # 实例化混合爬虫/Instantiate hybrid crawler
    hybird_crawler = HybridCrawler()
    # 运行测试代码/Run test code
    asyncio.run(hybird_crawler.main())
```

utils

- api_exceptions.py

代码块

```python
class APIError(Exception):
    """基本API异常类，其他API异常都会继承这个类"""

    def __init__(self, status_code=None):
        self.status_code = status_code
        print(
            "程序出现异常，请检查错误信息。"
        )

    def display_error(self):
        """显示错误信息和状态码（如果有的话）"""
        return f"Error: {self.args[0]}." + (
            f" Status Code: {self.status_code}." if self.status_code else ""
        )


class APIConnectionError(APIError):
    """当与API的连接出现问题时抛出"""

    def display_error(self):
        return f"API Connection Error: {self.args[0]}."


class APIUnavailableError(APIError):
    """当API服务不可用时抛出，例如维护或超时"""

    def display_error(self):
        return f"API Unavailable Error: {self.args[0]}."


class APINotFoundError(APIError):
    """当API端点不存在时抛出"""

    def display_error(self):
        return f"API Not Found Error: {self.args[0]}."


class APIResponseError(APIError):
    """当API返回的响应与预期不符时抛出"""

    def display_error(self):
        return f"API Response Error: {self.args[0]}."


class APIRateLimitError(APIError):
    """当达到API的请求速率限制时抛出"""

    def display_error(self):
        return f"API Rate Limit Error: {self.args[0]}."


class APITimeoutError(APIError):
    """当API请求超时时抛出"""

    def display_error(self):
        return f"API Timeout Error: {self.args[0]}."


class APIUnauthorizedError(APIError):
    """当API请求由于授权失败而被拒绝时抛出"""

    def display_error(self):
        return f"API Unauthorized Error: {self.args[0]}."


class APIRetryExhaustedError(APIError):
    """当API请求重试次数用尽时抛出"""

    def display_error(self):
        return f"API Retry Exhausted Error: {self.args[0]}."
```

- deprecated.py

代码块

```python
import warnings
import functools


def deprecated(message):
    def decorator(func):
        @functools.wraps(func)
        async def wrapper(*args, **kwargs):
            warnings.warn(
                f"{func.__name__} is deprecated: {message}",
                DeprecationWarning,
                stacklevel=2
            )
            return await func(*args, **kwargs)

        return wrapper

    return decorator
```

- logger.py

代码块

```
import threading
import time
import logging
import datetime

from pathlib import Path
from rich.logging import RichHandler
from logging.handlers import TimedRotatingFileHandler


class Singleton(type):
    _instances = {}  # 存储实例的字典
    _lock: threading.Lock = threading.Lock()  # 线程锁

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def __call__(cls, *args, **kwargs):
        """
        重写默认的类实例化方法。当尝试创建类的一个新实例时，此方法将被调用。
        如果已经有一个与参数匹配的实例存在，则返回该实例；否则创建一个新实例。
        """
        key = (cls, args, frozenset(kwargs.items()))
        with cls._lock:
            if key not in cls._instances:
                instance = super().__call__(*args, **kwargs)
                cls._instances[key] = instance
        return cls._instances[key]

    @classmethod
    def reset_instance(cls, *args, **kwargs):
        """
        重置指定参数的实例。这只是从 _instances 字典中删除实例的引用，
        并不真正删除该实例。如果其他地方仍引用该实例，它仍然存在且可用。
        """
        key = (cls, args, frozenset(kwargs.items()))
        with cls._lock:
            if key in cls._instances:
                del cls._instances[key]


class LogManager(metaclass=Singleton):
```

```python
    def __init__(self):
        if getattr(self, "_initialized", False):  # 防止重复初始化
            return

        self.logger = logging.getLogger("Douyin_TikTok_Download_API_Crawlers")
        self.logger.setLevel(logging.INFO)
        self.log_dir = None
        self._initialized = True

    def setup_logging(self, level=logging.INFO, log_to_console=False, log_path=None):
        self.logger.handlers.clear()
        self.logger.setLevel(level)

        if log_to_console:
            ch = RichHandler(
                show_time=False,
                show_path=False,
                markup=True,
                keywords=(RichHandler.KEYWORDS or []) + ["STREAM"],
                rich_tracebacks=True,
            )
            ch.setFormatter(logging.Formatter("{message}", style="{", datefmt="[%X]"))
            self.logger.addHandler(ch)

        if log_path:
            self.log_dir = Path(log_path)
            self.ensure_log_dir_exists(self.log_dir)
            log_file_name = datetime.datetime.now().strftime("%Y-%m-%d-%H-%M-%S.log")
            log_file = self.log_dir.joinpath(log_file_name)
            fh = TimedRotatingFileHandler(
                log_file, when="midnight", interval=1, backupCount=99, encoding="utf-8"
            )
            fh.setFormatter(
                logging.Formatter(
                    "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
                )
            )
            self.logger.addHandler(fh)

    @staticmethod
    def ensure_log_dir_exists(log_path: Path):
        log_path.mkdir(parents=True, exist_ok=True)

    def clean_logs(self, keep_last_n=10):
        """保留最近的n个日志文件并删除其他文件"""
        if not self.log_dir:
            return
        # self.shutdown()
        all_logs = sorted(self.log_dir.glob("*.log"))
        if keep_last_n == 0:
            files_to_delete = all_logs
        else:
            files_to_delete = all_logs[:-keep_last_n]
        for log_file in files_to_delete:
            try:
                log_file.unlink()
            except PermissionError:
                self.logger.warning(
                    f"无法删除日志文件 {log_file}, 它正被另一个进程使用"
                )

    def shutdown(self):
        for handler in self.logger.handlers:
            handler.close()
            self.logger.removeHandler(handler)
        self.logger.handlers.clear()
        time.sleep(1)  # 确保文件被释放


def log_setup(log_to_console=True):
    logger = logging.getLogger("Douyin_TikTok_Download_API_Crawlers")
    if logger.hasHandlers():
        # logger已经被设置，不做任何操作
        return logger

    # 创建临时的日志目录
    temp_log_dir = Path("./logs")
    temp_log_dir.mkdir(exist_ok=True)

    # 初始化日志管理器
    log_manager = LogManager()
    log_manager.setup_logging(
        level=logging.INFO, log_to_console=log_to_console, log_path=temp_log_dir
    )

    # 只保留1000个日志文件
    log_manager.clean_logs(1000)

    return logger
```

```


logger = log_setup()
```

- utils.py

代码块

```python
import re
import sys
import random
import secrets
import datetime
import browser_cookie3
import importlib_resources

from pydantic import BaseModel

from urllib.parse import quote, urlencode  # URL编码
from typing import Union, List, Any
from pathlib import Path

# 生成一个 16 字节的随机字节串 (Generate a random byte string of 16 bytes)
seed_bytes = secrets.token_bytes(16)

# 将字节字符串转换为整数 (Convert the byte string to an integer)
seed_int = int.from_bytes(seed_bytes, "big")

# 设置随机种子 (Seed the random module)
random.seed(seed_int)


# 将模型实例转换为字典
def model_to_query_string(model: BaseModel) -> str:
    model_dict = model.dict()
    # 使用urlencode进行URL编码
    query_string = urlencode(model_dict)
    return query_string


def gen_random_str(randomlength: int) -> str:
    """
    根据传入长度产生随机字符串 (Generate a random string based on the given length)

    Args:
        randomlength (int): 需要生成的随机字符串的长度 (The length of the random string to be generated)

    Returns:
        str: 生成的随机字符串 (The generated random string)
    """

    base_str = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-"
    return "".join(random.choice(base_str) for _ in range(randomlength))


def get_timestamp(unit: str = "milli"):
    """
    根据给定的单位获取当前时间 (Get the current time based on the given unit)

    Args:
        unit (str): 时间单位，可以是 "milli"、"sec"、"min" 等
            (The time unit, which can be "milli", "sec", "min", etc.)

    Returns:
        int: 根据给定单位的当前时间 (The current time based on the given unit)
    """

    now = datetime.datetime.utcnow() - datetime.datetime(1970, 1, 1)
    if unit == "milli":
        return int(now.total_seconds() * 1000)
    elif unit == "sec":
        return int(now.total_seconds())
    elif unit == "min":
        return int(now.total_seconds() / 60)
    else:
        raise ValueError("Unsupported time unit")


def timestamp_2_str(
        timestamp: Union[str, int, float], format: str = "%Y-%m-%d %H-%M-%S"
) -> str:
    """
    将 UNIX 时间戳转换为格式化字符串 (Convert a UNIX timestamp to a formatted string)

    Args:
        timestamp (int): 要转换的 UNIX 时间戳 (The UNIX timestamp to be converted)
        format (str, optional): 返回的日期时间字符串的格式。
                                默认为 '%Y-%m-%d %H-%M-%S'。
```

```python
                                  (The format for the returned date-time string
                                  Defaults to '%Y-%m-%d %H-%M-%S')

    Returns:
        str: 格式化的日期时间字符串 (The formatted date-time string)
    """
    if timestamp is None or timestamp == "None":
        return ""

    if isinstance(timestamp, str):
        if len(timestamp) == 30:
            return datetime.datetime.strptime(timestamp, "%a %b %d %H:%M:%S %z
%Y")

    return datetime.datetime.fromtimestamp(float(timestamp)).strftime(format)


def num_to_base36(num: int) -> str:
    """数字转换成base32 (Convert number to base 36)"""

    base_str = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

    if num == 0:
        return "0"

    base36 = []
    while num:
        num, i = divmod(num, 36)
        base36.append(base_str[i])

    return "".join(reversed(base36))


def split_set_cookie(cookie_str: str) -> str:
    """
    拆分Set-Cookie字符串并拼接 (Split the Set-Cookie string and concatenate)

    Args:
        cookie_str (str): 待拆分的Set-Cookie字符串 (The Set-Cookie string to be
split)

    Returns:
        str: 拼接后的Cookie字符串 (Concatenated cookie string)
    """

    # 判断是否为字符串 / Check if it's a string
    if not isinstance(cookie_str, str):
            raise TypeError("`set-cookie` must be str")

    # 拆分Set-Cookie字符串,避免错误地在expires字段的值中分割字符串 (Split the Set-Cookie string, avoiding incorrect splitting on the value of the 'expires' field)
    # 拆分每个Cookie字符串，只获取第一个分段（即key=value部分） / Split each Cookie string, only getting the first segment (i.e., key=value part)
    # 拼接所有的Cookie (Concatenate all cookies)
    return ";".join(
        cookie.split(";")[0] for cookie in re.split(", (?=[a-zA-Z])", cookie_str)
    )


def split_dict_cookie(cookie_dict: dict) -> str:
    return "; ".join(f"{key}={value}" for key, value in cookie_dict.items())


def extract_valid_urls(inputs: Union[str, List[str]]) -> Union[str, List[str], None]:
    """从输入中提取有效的URL (Extract valid URLs from input)

    Args:
        inputs (Union[str, list[str]]): 输入的字符串或字符串列表 (Input string or list of strings)

    Returns:
        Union[str, list[str]]: 提取出的有效URL或URL列表 (Extracted valid URL or list of URLs)
    """
    url_pattern = re.compile(r"https?://\S+")

    # 如果输入是单个字符串
    if isinstance(inputs, str):
        match = url_pattern.search(inputs)
        return match.group(0) if match else None

    # 如果输入是字符串列表
    elif isinstance(inputs, list):
        valid_urls = []

        for input_str in inputs:
            matches = url_pattern.findall(input_str)
            if matches:
                valid_urls.extend(matches)

        return valid_urls
```

```python


def _get_first_item_from_list(_list) -> list:
    # 检查是否是列表 (Check if it's a list)
    if _list and isinstance(_list, list):
        # 如果列表里第一个还是列表则提起每一个列表的第一个值
        # (If the first one in the list is still a list then bring up the first value of each list)
        if isinstance(_list[0], list):
            return [inner[0] for inner in _list if inner]
        # 如果只是普通列表，则返回这个列表包含的第一个项目作为新列表
        # (If it's just a regular list, return the first item wrapped in a list)
        else:
            return [_list[0]]
    return []


def get_resource_path(filepath: str):
    """获取资源文件的路径 (Get the path of the resource file)

    Args:
        filepath: str: 文件路径 (file path)
    """

    return importlib_resources.files("f2") / filepath


def replaceT(obj: Union[str, Any]) -> Union[str, Any]:
    """
    替换文案非法字符 (Replace illegal characters in the text)

    Args:
        obj (str): 传入对象 (Input object)

    Returns:
        new: 处理后的内容 (Processed content)
    """

    reSub = r"[^\u4e00-\u9fa5a-zA-Z0-9#]"

    if isinstance(obj, list):
        return [re.sub(reSub, "_", i) for i in obj]

    if isinstance(obj, str):
        return re.sub(reSub, "_", obj)

    return obj
    # raise TypeError("输入应为字符串或字符串列表")


def split_filename(text: str, os_limit: dict) -> str:
    """
    根据操作系统的字符限制分割文件名，并用 '......' 代替。

    Args:
        text (str): 要计算的文本
        os_limit (dict): 操作系统的字符限制字典

    Returns:
        str: 分割后的文本
    """
    # 获取操作系统名称和文件名长度限制
    os_name = sys.platform
    filename_length_limit = os_limit.get(os_name, 200)

    # 计算中文字符长度（中文字符长度*3）
    chinese_length = sum(1 for char in text if "\u4e00" <= char <= "\u9fff") *
3
    # 计算英文字符长度
    english_length = sum(1 for char in text if char.isalpha())
    # 计算下划线数量
    num_underscores = text.count("_")

    # 计算总长度
    total_length = chinese_length + english_length + num_underscores

    # 如果总长度超过操作系统限制或手动设置的限制，则根据限制进行分割
    if total_length > filename_length_limit:
        split_index = min(total_length, filename_length_limit) // 2 - 6
        split_text = text[:split_index] + "......" + text[-split_index:]
        return split_text
    else:
        return text


def ensure_path(path: Union[str, Path]) -> Path:
    """确保路径是一个Path对象 (Ensure the path is a Path object)"""
    return Path(path) if isinstance(path, str) else path


def get_cookie_from_browser(browser_choice: str, domain: str = "") -> dict:
    """
    根据用户选择的浏览器获取domain的cookie。

    Args:
        browser_choice (str): 用户选择的浏览器名称

    Returns:
        str: *.domain的cookie值
    """

    if not browser_choice or not domain:
        return ""

    BROWSER_FUNCTIONS = {
        "chrome": browser_cookie3.chrome,
        "firefox": browser_cookie3.firefox,
        "edge": browser_cookie3.edge,
        "opera": browser_cookie3.opera,
        "opera_gx": browser_cookie3.opera_gx,
        "safari": browser_cookie3.safari,
        "chromium": browser_cookie3.chromium,
        "brave": browser_cookie3.brave,
        "vivaldi": browser_cookie3.vivaldi,
        "librewolf": browser_cookie3.librewolf,
    }
    cj_function = BROWSER_FUNCTIONS.get(browser_choice)
    cj = cj_function(domain_name=domain)
    cookie_value = {c.name: c.value for c in cj if c.domain.endswith(domain)}
    return cookie_value


def check_invalid_naming(
        naming: str, allowed_patterns: list, allowed_separators: list
) -> list:
    """
    检查命名是否符合命名模板 (Check if the naming conforms to the naming template)

    Args:
        naming (str): 命名字符串 (Naming string)
        allowed_patterns (list): 允许的模式列表 (List of allowed patterns)
        allowed_separators (list): 允许的分隔符列表 (List of allowed separators)
    Returns:
        list: 无效的模式列表 (List of invalid patterns)
    """
    if not naming or not allowed_patterns or not allowed_separators:
        return []

    temp_naming = naming
    invalid_patterns = []

    # 检查提供的模式是否有效
    for pattern in allowed_patterns:
        if pattern in temp_naming:
            temp_naming = temp_naming.replace(pattern, "")

    # 此时，temp_naming应只包含分隔符
    for char in temp_naming:
        if char not in allowed_separators:
            invalid_patterns.append(char)

    # 检查连续的无效模式或分隔符
    for pattern in allowed_patterns:
        # 检查像"{xxx}{xxx}"这样的模式
        if pattern + pattern in naming:
            invalid_patterns.append(pattern + pattern)
        for sep in allowed_patterns:
            # 检查像"{xxx}-{xxx}"这样的模式
            if pattern + sep + pattern in naming:
                invalid_patterns.append(pattern + sep + pattern)

    return invalid_patterns


def merge_config(
        main_conf: dict = ...,
        custom_conf: dict = ...,
        **kwargs,
):
    """
    合并配置参数，使 CLI 参数优先级高于自定义配置，自定义配置优先级高于主配置，最终生成完整配置参数字典。

    Args:
        main_conf (dict): 主配置参数字典
        custom_conf (dict): 自定义配置参数字典
        **kwargs: CLI 参数和其他额外的配置参数

    Returns:
        dict: 合并后的配置参数字典
    """
    # 合并主配置和自定义配置
    merged_conf = {}
    for key, value in main_conf.items():
        merged_conf[key] = value  # 将主配置复制到合并后的配置中
    for key, value in custom_conf.items():
        if value is not None and value != "":  # 只有值不为 None 和 空值，才进行合并
            merged_conf[key] = value  # 自定义配置参数会覆盖主配置中的同名参数

    # 合并 CLI 参数与合并后的配置，确保 CLI 参数的优先级最高
    for key, value in kwargs.items():
        if key not in merged_conf:  # 如果合并后的配置中没有这个键，则直接添加
            merged_conf[key] = value
        elif value is not None and value != "":  # 如果值不为 None 和 空值，则进行合并
            merged_conf[key] = value  # CLI 参数会覆盖自定义配置和主配置中的同名参数

    return merged_conf
```

base_crawler.py

代码块

```python
import httpx
import json
import asyncio
import re

from httpx import Response

from data_collection_service.crawlers.utils.logger import logger
from data_collection_service.crawlers.utils.api_exceptions import (
    APIError,
    APIConnectionError,
    APIResponseError,
    APITimeoutError,
    APIUnavailableError,
    APIUnauthorizedError,
    APINotFoundError,
    APIRateLimitError,
    APIRetryExhaustedError,
)


class BaseCrawler:
    """
    基础爬虫客户端 (Base crawler client)
    """

    def __init__(
            self,
            proxies: dict = None,
            max_retries: int = 3,
            max_connections: int = 50,
            timeout: int = 10,
            max_tasks: int = 50,
            crawler_headers: dict = {},
    ):
        if isinstance(proxies, dict):
            self.proxies = proxies
            # [f"{k}://{v}" for k, v in proxies.items()]
        else:
            self.proxies = None

        # 爬虫请求头 / Crawler request header
        self.crawler_headers = crawler_headers or {}

        # 异步的任务数 / Number of asynchronous tasks
        self._max_tasks = max_tasks
        self.semaphore = asyncio.Semaphore(max_tasks)

        # 限制最大连接数 / Limit the maximum number of connections
        self._max_connections = max_connections
        self.limits = httpx.Limits(max_connections=max_connections)

        # 业务逻辑重试次数 / Business logic retry count
        self._max_retries = max_retries
        # 底层连接重试次数 / Underlying connection retry count
        self.atransport = httpx.AsyncHTTPTransport(retries=max_retries)

        # 超时等待时间 / Timeout waiting time
        self._timeout = timeout
        self.timeout = httpx.Timeout(timeout)
        # 异步客户端 / Asynchronous client
        self.aclient = httpx.AsyncClient(
            headers=self.crawler_headers,
            proxies=self.proxies,
            timeout=self.timeout,
            limits=self.limits,
            transport=self.atransport,
        )

    async def fetch_response(self, endpoint: str) -> Response:
        """获取数据 (Get data)

        Args:
            endpoint (str): 接口地址 (Endpoint URL)

        Returns:
            Response: 原始响应对象 (Raw response object)
        """
        return await self.get_fetch_data(endpoint)

    async def fetch_get_json(self, endpoint: str) -> dict:
        """获取 JSON 数据 (Get JSON data)

        Args:
            endpoint (str): 接口地址 (Endpoint URL)

        Returns:
            dict: 解析后的JSON数据 (Parsed JSON data)
        """
        response = await self.get_fetch_data(endpoint)
        return self.parse_json(response)

    async def fetch_post_json(self, endpoint: str, params: dict = {},
data=None) -> dict:
        """获取 JSON 数据 (Post JSON data)

        Args:
            endpoint (str): 接口地址 (Endpoint URL)

        Returns:
            dict: 解析后的JSON数据 (Parsed JSON data)
        """
        response = await self.post_fetch_data(endpoint, params, data)
        return self.parse_json(response)

    def parse_json(self, response: Response) -> dict:
        """解析JSON响应对象 (Parse JSON response object)

        Args:
            response (Response): 原始响应对象 (Raw response object)

        Returns:
            dict: 解析后的JSON数据 (Parsed JSON data)
        """
        if (
                response is not None
                and isinstance(response, Response)
                and response.status_code == 200
        ):
            try:
                return response.json()
            except json.JSONDecodeError as e:
                # 尝试使用正则表达式匹配response.text中的json数据
                match = re.search(r"\{.*\}", response.text)
                try:
                    return json.loads(match.group())
                except json.JSONDecodeError as e:
                    logger.error("解析 {0} 接口 JSON 失败: {1}".format(response.url, e))
                    raise APIResponseError("解析JSON数据失败")

        else:
            if isinstance(response, Response):
                logger.error(
                    "获取数据失败。状态码: {0}".format(response.status_code)
                )
            else:
                logger.error("无效响应类型。响应类型: {0}".format(type(response)))

            raise APIResponseError("获取数据失败")

    async def get_fetch_data(self, url: str):
        """
        获取GET端点数据 (Get GET endpoint data)

        Args:
            url (str): 端点URL (Endpoint URL)

        Returns:
            response: 响应内容 (Response content)
        """
        for attempt in range(self._max_retries):
            try:
                response = await self.aclient.get(url, follow_redirects=True)
                if not response.text.strip() or not response.content:
                    error_message = "第 {0} 次响应内容为空, 状态码: {1}, URL: {2}".format(attempt + 1,
                                                                                     response.status_code,
                                                                                     response.url)

                    logger.warning(error_message)

                    if attempt == self._max_retries - 1:
                        raise APIRetryExhaustedError(
                            "获取端点数据失败, 次数达到上限"
                        )

                    await asyncio.sleep(self._timeout)
                    continue

                # logger.info("响应状态码: {0}".format(response.status_code))
                response.raise_for_status()
                return response

            except httpx.RequestError:
                raise APIConnectionError("连接端点失败，检查网络环境或代理: {0} 代理: {1} 类名: {2}"
                                         .format(url, self.proxies, self.__class__.__name__)
                                         )

            except httpx.HTTPStatusError as http_error:
                self.handle_http_status_error(http_error, url, attempt + 1)

            except APIError as e:
                e.display_error()

    async def post_fetch_data(self, url: str, params: dict = {}, data=None):
        """
        获取POST端点数据 (Get POST endpoint data)

        Args:
            url (str): 端点URL (Endpoint URL)
            params (dict): POST请求参数 (POST request parameters)

        Returns:
            response: 响应内容 (Response content)
        """
        for attempt in range(self._max_retries):
            try:
                response = await self.aclient.post(
                    url,
                    json=None if not params else dict(params),
                    data=None if not data else data,
                    follow_redirects=True
                )
                if not response.text.strip() or not response.content:
                    error_message = "第 {0} 次响应内容为空, 状态码: {1}, URL: {2}".format(attempt + 1,
                                                                                        response.status_code,
                                                                                        response.url)

                    logger.warning(error_message)

                    if attempt == self._max_retries - 1:
                        raise APIRetryExhaustedError(
                            "获取端点数据失败，次数达到上限"
                        )

                    await asyncio.sleep(self._timeout)
                    continue

                # logger.info("响应状态码: {0}".format(response.status_code))
                response.raise_for_status()
                return response

            except httpx.RequestError:
                raise APIConnectionError(
                    "连接端点失败，检查网络环境或代理：{0} 代理：{1} 类名：{2}".format(url, self.proxies,
    self.__class__.__name__)
                )

            except httpx.HTTPStatusError as http_error:
                self.handle_http_status_error(http_error, url, attempt + 1)

            except APIError as e:
                e.display_error()

    async def head_fetch_data(self, url: str):
        """
        获取HEAD端点数据 (Get HEAD endpoint data)

        Args:
            url (str): 端点URL (Endpoint URL)

        Returns:
            response: 响应内容 (Response content)
        """
        try:
            response = await self.aclient.head(url)
            # logger.info("响应状态码: {0}".format(response.status_code))
            response.raise_for_status()
            return response

        except httpx.RequestError:
            raise APIConnectionError("连接端点失败，检查网络环境或代理：{0} 代理：{1} 类名：{2}".format(
                url, self.proxies, self.__class__.__name__
            )
            )

        except httpx.HTTPStatusError as http_error:
            self.handle_http_status_error(http_error, url, 1)

        except APIError as e:
            e.display_error()

    def handle_http_status_error(self, http_error, url: str, attempt):
        """
        处理HTTP状态错误 (Handle HTTP status error)

        Args:
            http_error: HTTP状态错误 (HTTP status error)
            url: 端点URL (Endpoint URL)
            attempt: 尝试次数 (Number of attempts)
        Raises:
            APIConnectionError: 连接端点失败 (Failed to connect to endpoint)
            APIResponseError: 响应错误 (Response error)
            APIUnavailableError: 服务不可用 (Service unavailable)
            APINotFoundError: 端点不存在 (Endpoint does not exist)
            APITimeoutError: 连接超时 (Connection timeout)
            APIUnauthorizedError: 未授权 (Unauthorized)
            APIRateLimitError: 请求频率过高 (Request frequency is too high)
            APIRetryExhaustedError: 重试次数达到上限 (The number of retries has reached the upper limit)
        """
        response = getattr(http_error, "response", None)
        status_code = getattr(response, "status_code", None)

        if response is None or status_code is None:
            logger.error("HTTP状态错误: {0}, URL: {1}, 尝试次数: {2}".format(
                http_error, url, attempt
            )
            )
            raise APIResponseError(f"处理HTTP错误时遇到意外情况: {http_error}")

        if status_code == 302:
            pass
        elif status_code == 404:
            raise APINotFoundError(f"HTTP Status Code {status_code}")
        elif status_code == 503:
            raise APIUnavailableError(f"HTTP Status Code {status_code}")
        elif status_code == 408:
            raise APITimeoutError(f"HTTP Status Code {status_code}")
        elif status_code == 401:
            raise APIUnauthorizedError(f"HTTP Status Code {status_code}")
        elif status_code == 429:
            raise APIRateLimitError(f"HTTP Status Code {status_code}")
        else:
            logger.error("HTTP状态错误: {0}, URL: {1}, 尝试次数: {2}".format(
                status_code, url, attempt
            )
            )
            raise APIResponseError(f"HTTP状态错误: {status_code}")

    async def close(self):
        await self.aclient.aclose()

    async def __aenter__(self):
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self.aclient.aclose()
```

三、基础镜像配置

docker-compose.yml

代码块

```yaml
services:
  # 关系型数据库: 存储核心结构化数据
  mysql:
    image: mysql:8.0
    container_name: mysql_8.0
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root_password
      MYSQL_DATABASE: KOL_platform
    ports:
      - "3306:3306"
    volumes:
      # 修改点: 使用具名卷mysql_data
      - mysql_data:/var/lib/mysql
#       - ./data/mysql:/var/lib/mysql

  # 缓存层: 支持高并发报价结果缓存
  redis:
    image: redis:7.0
    container_name: redis_7.0
    restart: always
    ports:
      - "6379:6379"

  # 非关系型数据库：存储各平台差异化的红人字段
  mongodb:
    image: mongo:6.0
    container_name: mongodb_6.0
    restart: always
    ports:
      - "27017:27017"
    volumes:
      # 修改点：使用具名卷 mongo_data
      - mongo_data:/data/db
#       - ./data/mongo:/data/db

  # 时序数据库：存储视频播放/互动监控数据
  clickhouse:
    image: clickhouse/clickhouse-server:23.8.16.16
    container_name: clickhouse_23.8
    restart: always
    environment:
      # 设置默认账户名为 admin
      - CLICKHOUSE_USER=${CLICKHOUSE_USER}
      # 设置您自定义的密码
      - CLICKHOUSE_PASSWORD=${CLICKHOUSE_PASSWORD}
      # 允许管理权限（可选，方便后续在DBeaver中管理用户）
      - CLICKHOUSE_DEFAULT_ACCESS_MANAGEMENT=1
    ports:
      - "8123:8123"
      - "9000:9000"
    volumes:
      # 修改点：使用具名卷 clickhouse_data
      - clickhouse_data:/var/lib/clickhouse
#       - ./data/clickhouse:/var/lib/clickhouse
    ulimits:
      nofile:
        soft: 262144
        hard: 262144

  # 消息队列：处理异步数据采集与通知
  zookeeper:
    image: zookeeper:3.9.1
    container_name: zookeeper_3.9.1
    restart: always
    environment:
      - ZOO_ENABLE_AUTH=no
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.5.0  # Confluent Kafka的稳定版本
    container_name: kafka_7.5.0
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper_3.9.1:2181 # 匹配Zookeeper的容器名（若有修改需对应）
#      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_BROKER_ID=1
      - KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092  # 开发环境暴露本地端口
      - KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1  # 单机环境副本数设为1

  # 服务注册与配置中心：Hertz 微服务体系的核心
  nacos:
    image: nacos/nacos-server:v2.2.3
    container_name: nacos_server
    environment:
      - MODE=standalone # 开发环境使用单机模式
      - NACOS_AUTH_ENABLE=false # 显示关闭鉴权，解决 NACOS_AUTH_TOKEN 报错问题
    ports:
      - "8848:8848"
      - "9848:9848" # Hertz/Go 客户端通信所需端口
volumes:
  mysql_data:
  mongo_data:
  clickhouse_data:
  # redis_data:
```