

项目代码开发记录

一、当前项目结构

代码块

```
1  kol_ads_marketing
2  |  └── data_collection_service: 数据采集服务
3  |  |  └── app
4  |  |  |  └── api
5  |  |  |  |  └── endpoints
6  |  |  |  |  |  └── bilibili_web.py
7  |  |  |  |  |  └── download.py
8  |  |  |  |  └── models
9  |  |  |  |  |  └── APIResponseModel.py
10 |  |  └── crawlers
11 |  |  |  └── bilibili
12 |  |  |  |  └── config.yaml
13 |  |  |  |  └── endpoints.py
14 |  |  |  |  └── models.py
15 |  |  |  |  └── utils.py
16 |  |  |  |  └── web_crawler.py
17 |  |  |  |  └── wrid.py
18 |  |  |  └── hybrid
19 |  |  |  |  └── hybrid_crawler.py
20 |  |  |  └── utils
21 |  |  |  |  └── api_exceptions.py
22 |  |  |  |  └── deprecated.py
23 |  |  |  |  └── logger.py
24 |  |  |  |  └── utils.py
25 |  |  |  └── base_crawler.py
26 |  └── deploy: 基础镜像配置
27 |  |  └── docker-compose.yml
```

二、数据采集微服务模块

app

api

endpoints

- bili_bili_web.py

代码块

```
1  from fastapi import APIRouter, Body, Query, Request, HTTPException # 导入
   FastAPI组件
2
3
4  from data_collection_service.app.api.models.APIResponseModel import
   ResponseModel, ErrorResponseModel # 导入响应模型
5  from data_collection_service.crawlers.bilibili.web_crawler import
   BilibiliWebCrawler # 导入哔哩哔哩web爬虫
6
7
8  router = APIRouter()
9  BilibiliWebCrawler = BilibiliWebCrawler()
10
11
12 # 获取单个视频详情信息
13 @router.get("/fetch_one_video", response_model=ResponseModel, summary="获取单个
   视频详情信息/Get single video data")
14 async def fetch_one_video(request: Request,
15                           bv_id: str = Query(example="BV1M1421t7hT",
16                                     description="作品id/Video id")):
17     """
18     # [中文]
19     ### 用途:
20     - 获取单个视频详情信息
21     ### 参数:
22     - bv_id: 作品id
23     ### 返回:
24     - 视频详情信息
25
26     # [示例/Example]
27     bv_id = "BV1M1421t7hT"
28     """
29
30     try:
31         data = await BilibiliWebCrawler.fetch_one_video(bv_id)
32         return ResponseModel(code=200,
33                               router=request.url.path,
34                               data=data)
35     except Exception as e:
36         status_code = 400
37         detail = ErrorResponseModel(code=status_code,
38                                     router=request.url.path,
39                                     params=dict(request.query_params),
40                                     )
```



```
81     """
82     # [中文]
83     ### 用途:
84     - 获取用户发布的视频数据
85     ### 参数:
86     - uid: 用户UID
87     - pn: 页码
88     ### 返回:
89     - 用户发布的视频数据
90
91     # [示例/Example]
92     uid = "178360345"
93     pn = 1
94     """
95
96     try:
97         data = await BilibiliWebCrawler.fetch_user_post_videos(uid, pn)
98         return ResponseModel(code=200,
99                             router=request.url.path,
100                            data=data)
101    except Exception as e:
102        status_code = 400
103        detail = ErrorResponseModel(code=status_code,
104                                    router=request.url.path,
105                                    params=dict(request.query_params),
106                                    )
107        raise HTTPException(status_code=status_code, detail=detail)
108
109    # 获取用户所有收藏夹信息
110 @router.get("/fetch_collect_folders", response_model=ResponseModel,
111             summary="获取用户所有收藏夹信息/Get user collection folders")
112 async def fetch_collect_folders(request: Request,
113                                 uid: str = Query(example="178360345",
114                               description="用户UID")):
115     """
116     # [中文]
117     ### 用途:
118     - 获取用户收藏作品数据
119     ### 参数:
120     - uid: 用户UID
121     ### 返回:
122     - 用户收藏夹信息
123
124     # [示例/Example]
125     uid = "178360345"
126     """
127
128     try:
```

```
127         data = await BilibiliWebCrawler.fetch_collect_folders(uid)
128     return ResponseModel(code=200,
129                         router=request.url.path,
130                         data=data)
131 except Exception as e:
132     status_code = 400
133     detail = ErrorResponseModel(code=status_code,
134                                 router=request.url.path,
135                                 params=dict(request.query_params),
136                                 )
137     raise HTTPException(status_code=status_code, detail=detail)
138
139
140 # 获取指定收藏夹内视频数据
141 @router.get("/fetch_user_collection_videos", response_model=ResponseModel,
142             summary="获取指定收藏夹内视频数据/Gets video data from a collection
143 folder")
144 async def fetch_user_collection_videos(request: Request,
145                                         folder_id: str =
146                                         Query(example="1756059545",
147                                               description="收藏
148                                               夹id/collection folder id"),
149                                         pn: int = Query(default=1,
150                                         description="页码/Page number"))
151                                         ):
152     """
153     # [中文]
154     ### 用途:
155     - 获取指定收藏夹内视频数据
156     ### 参数:
157     - folder_id: 用户UID
158     - pn: 页码
159     ### 返回:
160     - 指定收藏夹内视频数据
161
162     # [示例/Example]
163     folder_id = "1756059545"
164     pn = 1
165     """
166
167 try:
168     data = await BilibiliWebCrawler.fetch_folder_videos(folder_id, pn)
169     return ResponseModel(code=200,
170                         router=request.url.path,
171                         data=data)
172 except Exception as e:
173     status_code = 400
174     detail = ErrorResponseModel(code=status_code,
```

```
170                                     router=request.url.path,
171                                     params=dict(request.query_params),
172                                     )
173             raise HTTPException(status_code=status_code, detail=detail)
174
175
176     # 获取指定用户的信息
177     @router.get("/fetch_user_profile", response_model=ResponseModel,
178                 summary="获取指定用户的信息/Get information of specified user")
179     async def fetch_collect_folders(request: Request,
180                                     uid: str = Query(example="178360345",
181                                     description="用户UID")):
182         """
183         # [中文]
184         ### 用途:
185         - 获取指定用户的信息
186         ### 参数:
187         - uid: 用户UID
188         ### 返回:
189         - 指定用户的个人信息
190
191         # [示例/Example]
192         uid = "178360345"
193         """
194
195     try:
196         data = await BilibiliWebCrawler.fetch_user_profile(uid)
197         return ResponseModel(code=200,
198                             router=request.url.path,
199                             data=data)
200
201     except Exception as e:
202         status_code = 400
203         detail = ErrorResponseModel(code=status_code,
204                                     router=request.url.path,
205                                     params=dict(request.query_params),
206                                     )
207         raise HTTPException(status_code=status_code, detail=detail)
208
209
210     # 获取综合热门视频信息
211     @router.get("/fetch_com_popular", response_model=ResponseModel,
212                 summary="获取综合热门视频信息/Get comprehensive popular video
213                 information")
214     async def fetch_collect_folders(request: Request,
215                                     pn: int = Query(default=1, description="页
216                                     码/Page number")):
217         """
218         # [中文]
```

```
214     """ 用途:  
215     - 获取综合热门视频信息  
216     """  
217     - pn: 页码  
218     """ 返回:  
219     - 综合热门视频信息  
220  
221     # [示例/Example]  
222     pn = 1  
223     """  
224     try:  
225         data = await BilibiliWebCrawler.fetch_com_popular(pn)  
226         return ResponseModel(code=200,  
227                             router=request.url.path,  
228                             data=data)  
229     except Exception as e:  
230         status_code = 400  
231         detail = ErrorResponseModel(code=status_code,  
232                             router=request.url.path,  
233                             params=dict(request.query_params),  
234                             )  
235         raise HTTPException(status_code=status_code, detail=detail)  
236  
237  
238     # 获取指定视频的评论  
239     @router.get("/fetch_video_comments", response_model=ResponseModel,  
240                 summary="获取指定视频的评论/Get comments on the specified video")  
241     async def fetch_collect_folders(request: Request,  
242                                         bv_id: str = Query(example="BV1M1421t7hT",  
243                                         description="作品id/Video id"),  
244                                         pn: int = Query(default=1, description="页  
245                                         码/Page number")):  
246     """  
247     # [中文]  
248     """  
249     # 用途:  
250     - 获取指定视频的评论  
251     """  
252     - pn: 页码  
253     """  
254     # [示例/Example]  
255     bv_id = "BV1M1421t7hT"  
256     pn = 1  
257     """  
258     try:
```

```
259         data = await BilibiliWebCrawler.fetch_video_comments(bv_id, pn)
260     return ResponseModel(code=200,
261                         router=request.url.path,
262                         data=data)
263 except Exception as e:
264     status_code = 400
265     detail = ErrorResponseModel(code=status_code,
266                                router=request.url.path,
267                                params=dict(request.query_params),
268                                )
269     raise HTTPException(status_code=status_code, detail=detail)
270
271
272 # 获取视频下指定评论的回复
273 @router.get("/fetch_comment_reply", response_model=ResponseModel,
274             summary="获取视频下指定评论的回复/Get reply to the specified comment")
275 async def fetch_collect_folders(request: Request,
276                                 bv_id: str = Query(example="BV1M1421t7hT",
277                                         description="作品id/Video id"),
278                                 pn: int = Query(default=1, description="页
279                                 码/Page number"),
280                                 rpid: str = Query(example="237109455120",
281                                         description="回复id/Reply id")):
282     """
283     # [中文]
284     ### 用途:
285     - 获取视频下指定评论的回复
286     ### 参数:
287     - bv_id: 作品id
288     - pn: 页码
289     - rpid: 回复id
290     ### 返回:
291     - 指定评论的回复数据
292
293     # [示例/Example]
294     bv_id = "BV1M1421t7hT"
295     pn = 1
296     rpid = "237109455120"
297     """
298
299     try:
300         data = await BilibiliWebCrawler.fetch_comment_reply(bv_id, pn, rpid)
301     return ResponseModel(code=200,
302                         router=request.url.path,
303                         data=data)
304 except Exception as e:
305     status_code = 400
306     detail = ErrorResponseModel(code=status_code,
```

```
303                         router=request.url.path,
304                         params=dict(request.query_params),
305                         )
306             raise HTTPException(status_code=status_code, detail=detail)
307
308
309     # 获取指定用户动态
310     @router.get("/fetch_user_dynamic", response_model=ResponseModel,
311                 summary="获取指定用户动态/Get dynamic information of specified user")
312     async def fetch_collect_folders(request: Request,
313                                     uid: str = Query(example="16015678",
314                                     description="用户UID"),
315                                     offset: str = Query(default="",
316                                     example="953154282154098691",
317                                     description="开始索引/offset")):
318         """
319             # [中文]
320             ### 用途:
321             - 获取指定用户动态
322             ### 参数:
323             - uid: 用户UID
324             - offset: 开始索引
325             ### 返回:
326             - 指定用户动态数据
327
328             # [示例/Example]
329             uid = "178360345"
330             offset = "953154282154098691"
331             """
332
333     try:
334         data = await BilibiliWebCrawler.fetch_user_dynamic(uid, offset)
335         return ResponseModel(code=200,
336                             router=request.url.path,
337                             data=data)
338     except Exception as e:
339         status_code = 400
340         detail = ErrorResponseModel(code=status_code,
341                                     router=request.url.path,
342                                     params=dict(request.query_params),
343                                     )
344         raise HTTPException(status_code=status_code, detail=detail)
345
346
347     # 获取视频实时弹幕
348     @router.get("/fetch_video_danmaku", response_model=ResponseModel, summary="获取
349     视频实时弹幕/Get Video Danmaku")
```

```
346     async def fetch_one_video(request: Request,
347                               cid: str = Query(example="1639235405",
348                               description="作品cid/Video cid")):
349         """
350             # [中文]
351             ### 用途:
352                 - 获取视频实时弹幕
353             ### 参数:
354                 - cid: 作品cid
355             ### 返回:
356                 - 视频实时弹幕
357
358             # [示例/Example]
359             cid = "1639235405"
360             """
361
362     try:
363         data = await BilibiliWebCrawler.fetch_video_danmaku(cid)
364         return ResponseModel(code=200,
365                             router=request.url.path,
366                             data=data)
367     except Exception as e:
368         status_code = 400
369         detail = ErrorResponseModel(code=status_code,
370                                     router=request.url.path,
371                                     params=dict(request.query_params),
372                                     )
373         raise HTTPException(status_code=status_code, detail=detail)
374
375     # 获取指定直播间信息
376     @router.get("/fetch_live_room_detail", response_model=ResponseModel,
377                 summary="获取指定直播间信息/Get information of specified live room")
378     async def fetch_collect_folders(request: Request,
379                                     room_id: str = Query(example="22816111",
380                                     description="直播间ID/Live room ID")):
381         """
382             # [中文]
383             ### 用途:
384                 - 获取指定直播间信息
385             ### 参数:
386                 - room_id: 直播间ID
387             ### 返回:
388                 - 指定直播间信息
389
390             # [示例/Example]
391             room_id = "22816111"
392             """
```

```
391     try:
392         data = await BilibiliWebCrawler.fetch_live_room_detail(room_id)
393         return ResponseModel(code=200,
394                             router=request.url.path,
395                             data=data)
396     except Exception as e:
397         status_code = 400
398         detail = ErrorResponseModel(code=status_code,
399                                     router=request.url.path,
400                                     params=dict(request.query_params),
401                                     )
402         raise HTTPException(status_code=status_code, detail=detail)
403
404
405 # 获取指定直播间视频流
406 @router.get("/fetch_live_videos", response_model=ResponseModel,
407             summary="获取直播间视频流/Get live video data of specified room")
408 async def fetch_collect_folders(request: Request,
409                                 room_id: str = Query(example="1815229528",
410                               description="直播间ID/Live room ID")):
411     """
412     # [中文]
413     ### 用途:
414     - 获取指定直播间视频流
415     ### 参数:
416     - room_id: 直播间ID
417     ### 返回:
418     - 指定直播间视频流
419     # [示例/Example]
420     room_id = "1815229528"
421     """
422     try:
423         data = await BilibiliWebCrawler.fetch_live_videos(room_id)
424         return ResponseModel(code=200,
425                             router=request.url.path,
426                             data=data)
427     except Exception as e:
428         status_code = 400
429         detail = ErrorResponseModel(code=status_code,
430                                     router=request.url.path,
431                                     params=dict(request.query_params),
432                                     )
433         raise HTTPException(status_code=status_code, detail=detail)
434
435
436 # 获取指定分区正在直播的主播
```

```
437 @router.get("/fetch_live_streamers", response_model=ResponseModel,
438         summary="获取指定分区正在直播的主播/Get live streamers of specified
439         live area")
440     async def fetch_collect_folders(request: Request,
441                                     area_id: str = Query(example="9",
442                                     description="直播分区id/Live area ID"),
443                                     pn: int = Query(default=1, description="页
444                                     码/Page number")):
445         """
446         # [中文]
447         ### 用途:
448             - 获取指定分区正在直播的主播
449         ### 参数:
450             - area_id: 直播分区id
451             - pn: 页码
452         ### 返回:
453             - 指定分区正在直播的主播
454
455         # [示例/Example]
456         area_id = "9"
457         pn = 1
458         """
459
460     try:
461         data = await BilibiliWebCrawler.fetch_live_streamers(area_id, pn)
462         return ResponseModel(code=200,
463                             router=request.url.path,
464                             data=data)
465     except Exception as e:
466         status_code = 400
467         detail = ErrorResponseModel(code=status_code,
468                                     router=request.url.path,
469                                     params=dict(request.query_params),
470                                     )
471         raise HTTPException(status_code=status_code, detail=detail)
472
473     # 获取所有直播分区列表
474     @router.get("/fetch_all_live_areas", response_model=ResponseModel,
475         summary="获取所有直播分区列表/Get a list of all live areas")
476     async def fetch_collect_folders(request: Request,):
477         """
478         # [中文]
479         ### 用途:
480             - 获取所有直播分区列表
481         ### 参数:
482         ### 返回:
483             - 所有直播分区列表
```

```
481
482     # [示例/Example]
483     """
484     try:
485         data = await BilibiliWebCrawler.fetch_all_live_areas()
486         return ResponseModel(code=200,
487                             router=request.url.path,
488                             data=data)
489     except Exception as e:
490         status_code = 400
491         detail = ErrorResponseModel(code=status_code,
492                                     router=request.url.path,
493                                     params=dict(request.query_params),
494                                     )
495         raise HTTPException(status_code=status_code, detail=detail)
496
497
498     # 通过bv号获得视频aid号
499     @router.get("/bv_to_aid", response_model=ResponseModel, summary="通过bv号获得视频
500     aid号/Generate aid by bvid")
501     async def fetch_one_video(request: Request,
502                               bv_id: str = Query(example="BV1M1421t7hT",
503                               description="作品id/Video id")):
504         """
505         # [中文]
506         ### 用途:
507         - 通过bv号获得视频aid号
508         ### 参数:
509         - bv_id: 作品id
510         ### 返回:
511         - 视频aid号
512
513         # [示例/Example]
514         bv_id = "BV1M1421t7hT"
515         """
516
517         try:
518             data = await BilibiliWebCrawler.bv_to_aid(bv_id)
519             return ResponseModel(code=200,
520                                 router=request.url.path,
521                                 data=data)
522
523     except Exception as e:
524         status_code = 400
525         detail = ErrorResponseModel(code=status_code,
526                                     router=request.url.path,
527                                     params=dict(request.query_params),
528                                     )
529         raise HTTPException(status_code=status_code, detail=detail)
```

```

526
527
528     # 通过bv号获得视频分p信息
529     @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")
530     async def fetch_one_video(request: Request,
531                               bv_id: str = Query(example="BV1vf421i7hV",
532                                         description="作品id/Video id")):
533         """
534         # [中文]
535         ### 用途:
536             - 通过bv号获得视频分p信息
537         ### 参数:
538             - bv_id: 作品id
539         ### 返回:
540             - 视频分p信息
541
542         # [示例/Example]
543         bv_id = "BV1vf421i7hV"
544         """
545
546         try:
547             data = await BilibiliWebCrawler.fetch_video_parts(bv_id)
548             return ResponseModel(code=200,
549                                 router=request.url.path,
550                                 data=data)
551         except Exception as e:
552             status_code = 400
553             detail = ErrorResponseModel(code=status_code,
554                                         router=request.url.path,
555                                         params=dict(request.query_params),
556                                         )
557             raise HTTPException(status_code=status_code, detail=detail)

```

- download.py

代码块

```

1  import os
2  import zipfile
3  import subprocess
4  import tempfile
5
6  import aiofiles
7  import httpx
8  import yaml
9  from fastapi import APIRouter, Request, Query, HTTPException # 导入FastAPI组件

```

```
10 from starlette.responses import FileResponse
11
12 from data_collection_service.app.api.models.APIResponseModel import
13 ErrorResponseModel # 导入响应模型
14 from data_collection_service.crawlers.hybrid.hybrid_crawler import
15 HybridCrawler # 导入混合数据爬虫
16
17 router = APIRouter()
18 HybridCrawler = HybridCrawler()
19
20 # 读取上级再上级目录的配置文件
21 config_path =
22     os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.dirname(
23         file__)))), 'config.yaml')
24 with open(config_path, 'r', encoding='utf-8') as file:
25     config = yaml.safe_load(file)
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

```
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)

# 下载视频专用
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

# 流式保存文件
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()
```

```
50             os.remove(file_path)
51         return False
52     await out_file.write(chunk)
53 return True
54
55
56 async def merge_bilibili_video_audio(video_url: str, audio_url: str, request:
57 Request, output_path: str,
58                                         headers: dict) -> bool:
59     """
60     下载并合并 Bilibili 的视频流和音频流
61     """
62     try:
63         # 创建临时文件
64         with tempfile.NamedTemporaryFile(suffix='.m4v', delete=False) as
65         video_temp:
66             video_temp_path = video_temp.name
67         with tempfile.NamedTemporaryFile(suffix='.m4a', delete=False) as
68         audio_temp:
69             audio_temp_path = audio_temp.name
70
71             # 下载视频流
72             video_success = await fetch_data_stream(video_url, request,
73             headers=headers, file_path=video_temp_path)
74             # 下载音频流
75             audio_success = await fetch_data_stream(audio_url, request,
76             headers=headers, file_path=audio_temp_path)
77
78             if not video_success or not audio_success:
79                 print("Failed to download video or audio stream")
80                 return False
81
82             # 使用 FFmpeg 合并视频和音频
83             ffmpeg_cmd = [
84                 'ffmpeg', '-y', # -y 覆盖输出文件
85                 '-i', video_temp_path, # 视频输入
86                 '-i', audio_temp_path, # 音频输入
87                 '-c:v', 'copy', # 复制视频编码, 不重新编码
88                 '-c:a', 'copy', # 复制音频编码, 不重新编码 (保持原始质量)
89                 '-f', 'mp4', # 确保输出格式为MP4
90                 output_path
91             ]
92
93             print(f"FFmpeg command: {' '.join(ffmpeg_cmd)}")
94             result = subprocess.run(ffmpeg_cmd, capture_output=True, text=True)
95             print(f"FFmpeg return code: {result.returncode}")
96             if result.stderr:
```

```
92         print(f"FFmpeg stderr: {result.stderr}")
93     if result.stdout:
94         print(f"FFmpeg stdout: {result.stdout}")
95
96     # 清理临时文件
97     try:
98         os.unlink(video_temp_path)
99         os.unlink(audio_temp_path)
100    except:
101        pass
102
103    return result.returncode == 0
104
105 except Exception as e:
106     # 清理临时文件
107     try:
108         os.unlink(video_temp_path)
109         os.unlink(audio_temp_path)
110    except:
111        pass
112    print(f"Error merging video and audio: {e}")
113    return False
114
115
116 @router.get("/download",
117             summary="在线下载抖音|TikTok|Bilibili视频/图片/Online download
118             Douyin|TikTok|Bilibili video/image")
119             async def download_file_hybrid(request: Request,
120                                         url: str = Query(
121                                             example="https://www.douyin.com/video/7372484719365098803",
122                                             description="视频或图片的URL地址，支持抖
123             音|TikTok|Bilibili的分享链接，例如: https://v.douyin.com/e4J8Q7A/ 或
124             https://www.bilibili.com/video/BV1xxxxxxxxxx"),
125                                         prefix: bool = True,
126                                         with_watermark: bool = False):
127
128     """
129     # [中文]
130     ### 用途:
131     - 在线下载抖音/TikTok/Bilibili 无水印或有水印的视频/图片
132     - 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。
133     - 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
134     误，那么你可以使用此接口来下载视频。
135     - 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。
136     - 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
137     此接口。
138     ### 参数:
```

```
133     - url: 视频或图片的URL地址, 支持抖音/TikTok/Bilibili的分享链接, 例如:  
134         https://v.douyin.com/e4J8Q7A/ 或 https://www.bilibili.com/video/BV1xxxxxxxxxx  
135     - prefix: 下载文件的前缀, 默认为True, 可以在配置文件中修改。  
136     - 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  
137     印概念)  
138     #### 返回:  
139     - 返回下载的视频或图片文件响应。  
140  
141     # [示例/Example]  
142     url: https://www.bilibili.com/video/BV1U5efz2Egn  
143     """  
144     # 是否开启此端点/Whether to enable this endpoint  
145     if not config["API"]["Download_Switch"]:  
146         code = 400  
147         message = "Download endpoint is disabled in the configuration file. |  
148         配置文件中已禁用下载端点."  
149         return ErrorResponseModel(code=code, message=message,  
150             router=request.url.path,  
151             params=dict(request.query_params))  
152  
153     # 开始解析数据/Start parsing data  
154     try:  
155         data = await HybridCrawler.hybrid_parsing_single_video(url,  
156             minimal=True)  
157     except Exception as e:  
158         code = 400  
159         return ErrorResponseModel(code=code, message=str(e),  
160             router=request.url.path, params=dict(request.query_params))  
161  
162     # 开始下载文件/Start downloading files  
163     try:  
164         data_type = data.get('type')  
165         platform = data.get('platform')  
166         video_id = data.get('video_id') # 改为使用video_id  
167         file_prefix = config.get("API").get("Download_File_Prefix") if prefix  
168     else ''  
169         download_path = os.path.join(config.get("API").get("Download_Path"), f"  
170 {platform}_{data_type}")  
171  
172         # 确保目录存在/Ensure the directory exists  
173         os.makedirs(download_path, exist_ok=True)  
174  
175         # 下载视频文件/Download video file  
176         if data_type == 'video':  
177             file_name = f"{file_prefix}{platform}_{video_id}.mp4" if not  
178             with_watermark else f"{file_prefix}{platform}_{video_id}_watermark.mp4"  
179             file_path = os.path.join(download_path, file_name)
```

```
171
172     # 判断文件是否存在，存在就直接返回
173     if os.path.exists(file_path):
174         return FileResponse(path=file_path, media_type='video/mp4',
175     filename=file_name)
176
177     # 获取对应平台的headers
178     if platform == 'tiktok':
179         __headers = await
180             HybridCrawler.TikTokWebCrawler.get_tiktok_headers()
181     elif platform == 'bilibili':
182         __headers = await
183             HybridCrawler.BilibiliWebCrawler.get_bilibili_headers()
184     else: # douyin
185         __headers = await
186             HybridCrawler.DouyinWebCrawler.get_douyin_headers()
187
188     # Bilibili 特殊处理：音视频分离
189     if platform == 'bilibili':
190         video_data = data.get('video_data', {})
191         video_url = video_data.get('nwm_video_url_HQ') if not
192         with_watermark else video_data.get(
193             'wm_video_url_HQ')
194         audio_url = video_data.get('audio_url')
195         if not video_url or not audio_url:
196             raise HTTPException(
197                 status_code=500,
198                 detail="Failed to get video or audio URL from Bilibili"
199             )
200
201     # 使用专门的函数合并音视频
202     success = await merge_bilibili_video_audio(video_url,
203         audio_url, request, file_path,
204         __headers.get('headers'))
205     if not success:
206         raise HTTPException(
207             status_code=500,
208             detail="Failed to merge Bilibili video and audio
209             streams"
210         )
211     else:
212         # 其他平台的常规处理
213         url = data.get('video_data').get('nwm_video_url_HQ') if not
214         with_watermark else data.get(
215             'video_data').get('wm_video_url_HQ')
```

```
208             success = await fetch_data_stream(url, request,
209             headers=__headers, file_path=file_path)
210             if not success:
211                 raise HTTPException(
212                     status_code=500,
213                     detail="An error occurred while fetching data"
214                 )
215
216             # # 保存文件
217             # async with aiofiles.open(file_path, 'wb') as out_file:
218             #     await out_file.write(response.content)
219
220             # 返回文件内容
221             return FileResponse(path=file_path, filename=file_name,
222             media_type="video/mp4")
223
224             # 下载图片文件/Download image file
225             elif data_type == 'image':
226                 # 压缩文件属性/Compress file properties
227                 zip_file_name = f"{file_prefix}{platform}_{video_id}_images.zip" if
228                 not with_watermark else f"{file_prefix}"
229                 {platform}_{video_id}_images_watermark.zip"
230                 zip_file_path = os.path.join(download_path, zip_file_name)
231
232                 # 判断文件是否存在，存在就直接返回、
233                 if os.path.exists(zip_file_path):
234                     return FileResponse(path=zip_file_path,
235                     filename=zip_file_name, media_type="application/zip")
236
237                 # 获取图片文件/Get image file
238                 urls = data.get('image_data').get('no_watermark_image_list') if not
239                 with_watermark else data.get(
240                     'image_data').get('watermark_image_list')
241                 image_file_list = []
242                 for url in urls:
243                     # 请求图片文件/Request image file
244                     response = await fetch_data(url)
245                     index = int(urls.index(url))
246                     content_type = response.headers.get('content-type')
247                     file_format = content_type.split('/')[1]
248                     file_name = f"{file_prefix}{platform}_{video_id}_{index + 1}."
249                     {file_format}" if not with_watermark else f"{file_prefix}"
250                     {platform}_{video_id}_{index + 1}_watermark.{file_format}"
251                     file_path = os.path.join(download_path, file_name)
252                     image_file_list.append(file_path)
253
254                     # 保存文件/Save file
```

```

247             async with aiofiles.open(file_path, 'wb') as out_file:
248                 await out_file.write(response.content)
249
250             # 压缩文件/Compress file
251             with zipfile.ZipFile(zip_file_path, 'w') as zip_file:
252                 for image_file in image_file_list:
253                     zip_file.write(image_file, os.path.basename(image_file))
254
255             # 返回压缩文件/Return compressed file
256             return FileResponse(path=zip_file_path, filename=zip_file_name,
257             media_type="application/zip")
258
259         # 异常处理/Exception handling
260         except Exception as e:
261             print(e)
262             code = 400
263             return ErrorResponseModel(code=code, message=str(e),
264             router=request.url.path, params=dict(request.query_params))

```

models

- APIResponseModel.py

代码块

```

1  from fastapi import Body, FastAPI, Query, Request, HTTPException
2  from pydantic import BaseModel
3  from typing import Any, Callable, Type, Optional, Dict
4  from functools import wraps
5  import datetime
6
7  app = FastAPI()
8
9
10 # 定义响应模型
11 class ResponseModel(BaseModel):
12     code: int = 200
13     router: str = "Endpoint path"
14     data: Optional[Any] = {}
15
16
17 # 定义错误响应模型
18 class ErrorResponseModel(BaseModel):
19     code: int = 400
20     message: str = "An error occurred."
21     support: str = "Please contact us on Github:
22     https://github.com/Evil0ctal/Douyin_TikTok_Download_API"

```

```

22     time: str = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
23     router: str
24     params: dict = {}
25
26
27     # 混合解析响应模型
28     class HybridResponseModel(BaseModel):
29         code: int = 200
30         router: str = "Hybrid parsing single video endpoint"
31         data: Optional[Any] = {}
32
33
34     # iOS_Shortcut响应模型
35     class iOS_Shortcut(BaseModel):
36         version: str
37         update: str
38         link: str
39         link_en: str
40         note: str
41         note_en: str

```

crawler

bilibili

- config.yaml

代码块

```

1   TokenManager:
2     bilibili:
3       headers:
4         'accept-language': zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
5         'origin': https://www.bilibili.com
6         'referer': https://space.bilibili.com/
7         'origin_2': https://space.bilibili.com
8         'cookie': DedeUserID=292039314; DedeUserID__ckMd5=89698fb49e523c9f;
9           SESSDATA=f4c1988b%2C1783510832%2Cc1ccd%2A11CjA7my_d4M4B0b2moHhApExeeDewhjrsESg4
10          2o7MhCls4HB5jyteQSGNhH7mFhf9v-
11          YSVjR1aGllTXI0N2loU25UWUJNYk1Vb0NCdUNQN0p3MkJfTTZVUXNTT2xvZVBCaEtaWHh6UjJ0dTZIW
12          mJf0HMzUTdDQmZmTVBHRzRVZTZyNmpra1BWaUtRIIEC;
13          bili_jct=0f31589ace86c64a8261f09fbc272321; sid=q6g0j28t
14          'user-agent': Mozilla/5.0 (Windows NT 10.0; Win64; x64)
15            AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
16            Edg/143.0.0.0

```

```
11
12     proxies:
13         http:
14         https:
```

- endpoints.py

代码块

```
1  class BilibiliAPIEndpoints:
2
3      """-----域名-domain-----"""
4
5      # 哔哩哔哩接口域名
6      BILIAPI_DOMAIN = "https://api.bilibili.com"
7
8      # 哔哩哔哩直播域名
9      LIVE_DOMAIN = "https://api.live.bilibili.com"
10
11     """-----接口-api-----"""
12
13     # 作品信息 (Post Detail)
14     POST_DETAIL = f"{BILIAPI_DOMAIN}/x/web-interface/view"
15
16     # 作品视频流
17     VIDEO_PLAYURL = f"{BILIAPI_DOMAIN}/x/player/wbi/playurl"
18
19     # 用户发布视频作品数据
20     USER_POST = f"{BILIAPI_DOMAIN}/x/space/wbi/arc/search"
21
22     # 收藏夹列表
23     COLLECT_FOLDERS = f"{BILIAPI_DOMAIN}/x/v3/fav/folder/created/list-all"
24
25     # 收藏夹视频
26     COLLECT_VIDEOS = f"{BILIAPI_DOMAIN}/x/v3/fav/resource/list"
27
28     # 用户个人信息
29     USER_DETAIL = f"{BILIAPI_DOMAIN}/x/space/wbi/acc/info"
30
31     # 综合热门
32     COM_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular"
33
34     # 每周必看
35     WEEKLY_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular/series/one"
36
37     # 入站必刷
```

```

36     PRECIOUS_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular/precious"
37
38     # 视频评论
39     VIDEO_COMMENTS = f"{BILIAPI_DOMAIN}/x/v2/reply"
40
41     # 用户动态
42     USER_DYNAMIC = f"{BILIAPI_DOMAIN}/x/polymer/web-dynamic/v1/feed/space"
43
44     # 评论的回复
45     COMMENT_REPLY = f"{BILIAPI_DOMAIN}/x/v2/reply/reply"
46
47     # 视频分p信息
48     VIDEO_PARTS = f"{BILIAPI_DOMAIN}/x/player/pagelist"
49
50     # 直播间信息
51     LIVEROOM_DETAIL = f"{LIVE_DOMAIN}/room/v1/Room/get_info"
52
53     # 直播分区列表
54     LIVE.Areas = f"{LIVE_DOMAIN}/room/v1/Area/getList"
55
56     # 直播间视频流
57     LIVE_VIDEOS = f"{LIVE_DOMAIN}/room/v1/Room/playUrl"
58
59     # 正在直播的主播
60     LIVE_STREAMER = f"{LIVE_DOMAIN}/xlive/web-interface/v1/second/getList"

```

- models.py

代码块

```

1  import time
2  from pydantic import BaseModel
3
4
5  class BaseRequestsModel(BaseModel):
6      wts: str = str(round(time.time()))
7
8
9  class UserPostVideos(BaseRequestsModel):
10     dm_img_inter: str = '{"ds":[], "wh": [3557, 5674, 5], "of": [154, 308, 154]}'
11     dm_img_list: list = []
12     mid: str
13     pn: int
14     ps: str = "20"
15
16

```

```

17 class UserProfile(BaseRequestsModel):
18     mid: str
19
20
21 class UserDynamic(BaseRequestsModel):
22     host_mid: str
23     offset: str
24     wts: str = str(round(time.time()))
25
26
27 class ComPopular(BaseRequestsModel):
28     pn: int
29     ps: str = "20"
30     web_location: str = "333.934"
31
32
33 class PlayUrl(BaseRequestsModel):
34     qn: str
35     fnval: str = '4048'
36     bvid: str
37     cid: str

```

- utils.py

代码块

```

1  from urllib.parse import urlencode
2  from data_collection_service.crawlers.bilibili import wrid
3  from data_collection_service.crawlers.utils.logger import logger
4  from data_collection_service.crawlers.bilibili.endpoints import
    BilibiliAPIEndpoints
5
6  class EndpointGenerator:
7      def __init__(self, params: dict):
8          self.params = params
9
10     # 获取用户发布视频作品数据 生成endpoint
11     async def user_post_videos_endpoint(self) -> str:
12         # 添加w_rid
13         endpoint = await WridManager.wrid_model_endpoint(params=self.params)
14         # 拼接成最终结果并返回
15         final_endpoint = BilibiliAPIEndpoints.USER_POST + '?' + endpoint
16         return final_endpoint
17
18     # 获取视频流地址 生成endpoint
19     async def video_playurl_endpoint(self) -> str:

```

```
20     # 添加w_rid
21     endpoint = await WridManager.wrid_model_endpoint(params=self.params)
22     # 拼接成最终结果并返回
23     final_endpoint = BilibiliAPIEndpoints.VIDEO_PLAYURL + '?' + endpoint
24     return final_endpoint
25
26     # 获取指定用户的信息 生成endpoint
27     async def user_profile_endpoint(self) -> str:
28         # 添加w_rid
29         endpoint = await WridManager.wrid_model_endpoint(params=self.params)
30         # 拼接成最终结果并返回
31         final_endpoint = BilibiliAPIEndpoints.USER_DETAIL + '?' + endpoint
32         return final_endpoint
33
34     # 获取综合热门视频信息 生成endpoint
35     async def com_popular_endpoint(self) -> str:
36         # 添加w_rid
37         endpoint = await WridManager.wrid_model_endpoint(params=self.params)
38         # 拼接成最终结果并返回
39         final_endpoint = BilibiliAPIEndpoints.COM_POPULAR + '?' + endpoint
40         return final_endpoint
41
42     # 获取指定用户动态
43     async def user_dynamic_endpoint(self):
44         # 添加w_rid
45         endpoint = await WridManager.wrid_model_endpoint(params=self.params)
46         # 拼接成最终结果并返回
47         final_endpoint = BilibiliAPIEndpoints.USER_DYNAMIC + '?' + endpoint
48         return final_endpoint
49
50
51 class WridManager:
52     @classmethod
53         async def get_encode_query(cls, params: dict) -> str:
54             params['wts'] = params['wts'] + "ea1db124af3c7062474693fa704f4ff8"
55             params = dict(sorted(params.items())) # 按照 key 重排参数
56             # 过滤 value 中的 "!'()*" 字符
57             params = {
58                 k: ''.join(filter(lambda chr: chr not in "!'()*", str(v)))
59                 for k, v
60                 in params.items()
61             }
62             query = urlencode(params) # 序列化参数
63             return query
64
65     @classmethod
66         async def wrid_model_endpoint(cls, params: dict) -> str:
```

```

67     wts = params["wts"]
68     encode_query = await cls.get_encode_query(params)
69     # 获取w_rid参数
70     w_rid = wrid.get_wrid(e=encode_query)
71     params["wts"] = wts
72     params["w_rid"] = w_rid
73     return "&".join(f"{k}={v}" for k, v in params.items())
74
75     # BV号转为对应av号
76     async def bv2av(bv_id: str) -> int:
77         table = "fZodR9XQDSUm21yCkr6zBqiveYah8bt4xsWpHnJE7jL5VG3guMTKNPAwcF"
78         s = [11, 10, 3, 8, 4, 6, 2, 9, 5, 7]
79         xor = 177451812
80         add_105 = 8728348608
81         add_all = 8728348608 - (2 ** 31 - 1) - 1
82         tr = [0] * 128
83         for i in range(58):
84             tr[ord(table[i])] = i
85         r = 0
86         for i in range(6):
87             r += tr[ord(bv_id[s[i]])] * (58 ** i)
88         add = add_105
89         if r < add:
90             add = add_all
91         aid = (r - add) ^ xor
92         return aid
93
94     # 响应分析
95     class ResponseAnalyzer:
96         # 用户收藏夹信息
97         @classmethod
98         async def collect_folders_analyze(cls, response: dict) -> dict:
99             if response['data']:
100                 return response
101             else:
102                 logger.warning("该用户收藏夹为空/用户设置为不可见")
103                 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

```

1 import asyncio # 异步I/O
2 import os # 系统操作
3 import time # 时间操作
4 import yaml # 配置文件

```

```
5
6
7 # 基础爬虫客户端和哔哩哔哩API端点
8 from data_collection_service.crawlers.base_crawler import BaseCrawler
9 from data_collection_service.crawlers.bilibili.endpoints import
BilibiliAPIEndpoints
10 # 哔哩哔哩工具类
11 from data_collection_service.crawlers.bilibili.utils import EndpointGenerator,
bv2av, ResponseAnalyzer
12 # 数据请求模型
13 from data_collection_service.crawlers.bilibili.models import UserPostVideos,
UserProfile, ComPopular, UserDynamic, PlayUrl
14
15 # 配置文件路径
16 path = os.path.abspath(os.path.dirname(__file__))
17
18 # 读取配置文件
19 with open(f"{path}/config.yaml", "r", encoding="utf-8") as f:
20     config = yaml.safe_load(f)
21
22
23 class BilibiliWebCrawler:
24
25     # 从配置文件读取哔哩哔哩请求头
26     async def get_bilibili_headers(self):
27         bili_config = config['TokenManager']['bilibili']
28         kwargs = {
29             "headers": {
30                 "accept-language": bili_config["headers"]["accept-language"],
31                 "origin": bili_config["headers"]["origin"],
32                 "referer": bili_config["headers"]["referer"],
33                 "user-agent": bili_config["headers"]["user-agent"],
34                 "cookie": bili_config["headers"]["cookie"],
35             },
36             "proxies": {"http://": bili_config["proxies"]["http"], "https://":
bili_config["proxies"]["https"]},
37         }
38         return kwargs
39
40     """-----handler接口列表-----"
41
42     # 获取单个视频详情信息
43     async def fetch_one_video(self, bv_id: str) -> dict:
44         # 获取请求头信息
45         kwargs = await self.get_bilibili_headers()
46         # 创建基础爬虫对象
```

```
47         base_crawler = BaseCrawler(proxies=kwargs["proxies"],  
48         crawler_headers=kwargs["headers"])  
49             async with base_crawler as crawler:  
50                 # 创建请求endpoint  
51                 endpoint = f"{BilibiliAPIEndpoints.POST_DETAIL}?bvid={bv_id}"  
52                 # 发送请求, 获取请求响应结果  
53                 response = await crawler.fetch_get_json(endpoint)  
54             return response  
55  
56     # 获取视频流地址  
57     async def fetch_video_playurl(self, bv_id: str, cid: str, qn: str = "64") -> dict:  
58         # 获取请求头信息  
59         kwargs = await self.get_bilibili_headers()  
60         # 创建基础爬虫对象  
61         base_crawler = BaseCrawler(proxies=kwargs["proxies"],  
62         crawler_headers=kwargs["headers"])  
63             async with base_crawler as crawler:  
64                 # 通过模型生成基本请求参数  
65                 params = PlayUrl(bvid=bv_id, cid=cid, qn=qn)  
66                 # 创建请求endpoint  
67                 generator = EndpointGenerator(params.dict())  
68                 endpoint = await generator.video_playurl_endpoint()  
69                 # 发送请求, 获取请求响应结果  
70                 response = await crawler.fetch_get_json(endpoint)  
71             return response  
72  
73     # 获取用户发布视频作品数据  
74     async def fetch_user_post_videos(self, uid: str, pn: int) -> dict:  
75         """  
76             :param uid: 用户uid  
77             :param pn: 页码  
78             :return:  
79         """  
80             # 获取请求头信息  
81             kwargs = await self.get_bilibili_headers()  
82             # 创建基础爬虫对象  
83             base_crawler = BaseCrawler(proxies=kwargs["proxies"],  
84             crawler_headers=kwargs["headers"])  
85             async with base_crawler as crawler:  
86                 # 通过模型生成基本请求参数  
87                 params = UserPostVideos(mid=uid, pn=pn)  
88                 # 创建请求endpoint  
89                 generator = EndpointGenerator(params.dict())  
90                 endpoint = await generator.user_post_videos_endpoint()  
91                 # 发送请求, 获取请求响应结果  
92                 response = await crawler.fetch_get_json(endpoint)
```

```
90         return response
91
92     # 获取用户所有收藏夹信息
93     async def fetch_collect_folders(self, uid: str) -> dict:
94         # 获取请求头信息
95         kwargs = await self.get_bilibili_headers()
96         # 创建基础爬虫对象
97         base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
98         async with base_crawler as crawler:
99             # 创建请求endpoint
100            endpoint = f"{BilibiliAPIEndpoints.COLLECT_FOLDERS}?up_mid={uid}"
101            # 发送请求，获取请求响应结果
102            response = await crawler.fetch_get_json(endpoint)
103            # 分析响应结果
104            result_dict = await ResponseAnalyzer.collect_folders_analyze(response=response)
105        return result_dict
106
107    # 获取指定收藏夹内视频数据
108    async def fetch_folder_videos(self, folder_id: str, pn: int) -> dict:
109        """
110            :param folder_id: 收藏夹id-- 可从<获取用户所有收藏夹信息>获得
111            :param pn: 页码
112            :return:
113        """
114        # 获取请求头信息
115        kwargs = await self.get_bilibili_headers()
116        # 创建基础爬虫对象
117        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
118        # 发送请求，获取请求响应结果
119        async with base_crawler as crawler:
120            endpoint = f"{BilibiliAPIEndpoints.COLLECT_VIDEOS}?media_id={folder_id}&pnum={pn}&pss=20&keyword=&order=mtime&type=0&tid=0&platform=web"
121            response = await crawler.fetch_get_json(endpoint)
122        return response
123
124    # 获取指定用户的信息
125    async def fetch_user_profile(self, uid: str) -> dict:
126        # 获取请求头信息
127        kwargs = await self.get_bilibili_headers()
128        # 创建基础爬虫对象
129        base_crawler = BaseCrawler(proxies=kwargs["proxies"], crawler_headers=kwargs["headers"])
130        async with base_crawler as crawler:
131            # 通过模型生成基本请求参数
```

```
132     params = UserProfile(mid=uid)
133     # 创建请求endpoint
134     generator = EndpointGenerator(params.dict())
135     endpoint = await generator.user_profile_endpoint()
136     # 发送请求，获取请求响应结果
137     response = await crawler.fetch_get_json(endpoint)
138     return response
139
140     # 获取综合热门视频信息
141     async def fetch_com_popular(self, pn: int) -> dict:
142         # 获取请求头信息
143         kwargs = await self.get_bilibili_headers()
144         # 创建基础爬虫对象
145         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
146                                     crawler_headers=kwargs["headers"])
146         async with base_crawler as crawler:
147             # 通过模型生成基本请求参数
148             params = ComPopular(pn=pn)
149             # 创建请求endpoint
150             generator = EndpointGenerator(params.dict())
151             endpoint = await generator.com_popular_endpoint()
152             # 发送请求，获取请求响应结果
153             response = await crawler.fetch_get_json(endpoint)
154         return response
155
156     # 获取指定视频的评论
157     async def fetch_video_comments(self, bv_id: str, pn: int) -> dict:
158         # 评论排序 -- 1:按点赞数排序。0:按时间顺序排序
159         sort = 1
160         # 获取请求头信息
161         kwargs = await self.get_bilibili_headers()
162         # 创建基础爬虫对象
163         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
164                                     crawler_headers=kwargs["headers"])
164         async with base_crawler as crawler:
165             # 创建请求endpoint
166             endpoint = f"{BilibiliAPIEndpoints.VIDEO_COMMENTS}?type=1&oid={bv_id}&sort={sort}&nohot=0&ps=20&pn={pn}"
167             # 发送请求，获取请求响应结果
168             response = await crawler.fetch_get_json(endpoint)
169         return response
170
171     # 获取视频下指定评论的回复
172     async def fetch_comment_reply(self, bv_id: str, pn: int, rpid: str) ->
173         dict:
174             """
175             :param bv_id: 目标视频bv号
176         """
```

```
175     :param pn: 页码
176     :param rpid: 目标评论id, 可通过fetch_video_comments获得
177     :return:
178     """
179     # 获取请求头信息
180     kwargs = await self.get_bilibili_headers()
181     # 创建基础爬虫对象
182     base_crawler = BaseCrawler(proxies=kwargs["proxies"],
183                                 crawler_headers=kwargs["headers"])
184     async with base_crawler as crawler:
185         # 创建请求endpoint
186         endpoint = f"{BilibiliAPIEndpoints.COMMENT_REPLY}?type=1&oid={bv_id}&root={rpid}=&ps=20&pn={pn}"
187         # 发送请求, 获取请求响应结果
188         response = await crawler.fetch_get_json(endpoint)
189         return response
190
191     # 获取指定用户动态
192     async def fetch_user_dynamic(self, uid: str, offset: str) -> dict:
193         # 获取请求头信息
194         kwargs = await self.get_bilibili_headers()
195         # 创建基础爬虫对象
196         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
197                                     crawler_headers=kwargs["headers"])
198         async with base_crawler as crawler:
199             # 通过模型生成基本请求参数
200             params = UserDynamic(host_mid=uid, offset=offset)
201             # 创建请求endpoint
202             generator = EndpointGenerator(params.dict())
203             endpoint = await generator.user_dynamic_endpoint()
204             print(endpoint)
205             # 发送请求, 获取请求响应结果
206             response = await crawler.fetch_get_json(endpoint)
207             return response
208
209     # 获取视频实时弹幕
210     async def fetch_video_danmaku(self, cid: str):
211         # 获取请求头信息
212         kwargs = await self.get_bilibili_headers()
213         # 创建基础爬虫对象
214         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
215                                     crawler_headers=kwargs["headers"])
216         async with base_crawler as crawler:
217             # 创建请求endpoint
218             endpoint = f"https://comment.bilibili.com/{cid}.xml"
219             # 发送请求, 获取请求响应结果
220             response = await crawler.fetch_response(endpoint)
```

```
218     return response.text
219
220     # 获取指定直播间信息
221     async def fetch_live_room_detail(self, room_id: str) -> dict:
222         # 获取请求头信息
223         kwargs = await self.get_bilibili_headers()
224         # 创建基础爬虫对象
225         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
226                                     crawler_headers=kwargs["headers"])
227         async with base_crawler as crawler:
228             # 创建请求endpoint
229             endpoint = f"{BilibiliAPIEndpoints.LIVEROOM_DETAIL}?room_id={room_id}"
230             # 发送请求，获取请求响应结果
231             response = await crawler.fetch_get_json(endpoint)
232             return response
233
234     # 获取指定直播间视频流
235     async def fetch_live_videos(self, room_id: str) -> dict:
236         # 获取请求头信息
237         kwargs = await self.get_bilibili_headers()
238         # 创建基础爬虫对象
239         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
240                                     crawler_headers=kwargs["headers"])
241         async with base_crawler as crawler:
242             # 创建请求endpoint
243             endpoint = f"{BilibiliAPIEndpoints.LIVE_VIDEOS}?cid={room_id}&quality=4"
244             # 发送请求，获取请求响应结果
245             response = await crawler.fetch_get_json(endpoint)
246             return response
247
248     # 获取指定分区正在直播的主播
249     async def fetch_live_streamers(self, area_id: str, pn: int):
250         # 获取请求头信息
251         kwargs = await self.get_bilibili_headers()
252         # 创建基础爬虫对象
253         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
254                                     crawler_headers=kwargs["headers"])
255         async with base_crawler as crawler:
256             # 创建请求endpoint
257             endpoint = f"{BilibiliAPIEndpoints.LIVE_STREAMER}?platform=web&parent_area_id={area_id}&page={pn}"
258             # 发送请求，获取请求响应结果
259             response = await crawler.fetch_get_json(endpoint)
260             return response
```

```
259     "-----utils接口列表-----"
260
261     # 通过bv号获得视频aid号
262     async def bv_to_aid(self, bv_id: str) -> int:
263         aid = await bv2av(bv_id=bv_id)
264         return aid
265
266     # 通过bv号获得视频分p信息
267     async def fetch_video_parts(self, bv_id: str) -> str:
268         # 获取请求头信息
269         kwargs = await self.get_bilibili_headers()
270         # 创建基础爬虫对象
271         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
272                                     crawler_headers=kwargs["headers"])
273         async with base_crawler as crawler:
274             # 创建请求endpoint
275             endpoint = f"{BilibiliAPIEndpoints.VIDEO_PARTS}?bvid={bv_id}"
276             # 发送请求, 获取请求响应结果
277             response = await crawler.fetch_get_json(endpoint)
278             return response
279
280     # 获取所有直播分区列表
281     async def fetch_all_live_areas(self) -> dict:
282         # 获取请求头信息
283         kwargs = await self.get_bilibili_headers()
284         # 创建基础爬虫对象
285         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
286                                     crawler_headers=kwargs["headers"])
287         async with base_crawler as crawler:
288             # 创建请求endpoint
289             endpoint = BilibiliAPIEndpoints.LIVE AREAS
290             # 发送请求, 获取请求响应结果
291             response = await crawler.fetch_get_json(endpoint)
292             return response
293
294     if __name__ == '__main__':
295         # 初始化
296         BilibiliWebCrawler = BilibiliWebCrawler()
297
298         # 开始时间
299         start = time.time()
300
301         # 结束时间
302         end = time.time()
```

303 print(f"耗时: {end - start}")

- wrid.py

代码块

```
1  import urllib.parse
2
3  def srotl(t, e):
4      return (t << e) | (t >> (32 - e))
5
6  def tendian(t):
7      if isinstance(t, int):
8          return (16711935 & srotl(t, 8)) | (4278255360 & srotl(t, 24))
9      for e in range(len(t)):
10         t[e] = tendian(t[e])
11     return t
12
13 # 没问题
14 def tbytes_to_words(t):
15     e = []
16     r = 0
17     for n in range(len(t)):
18         if r >> 5 >= len(e):
19             e.append(0)
20         e[r >> 5] |= t[n] << (24 - r % 32)
21         r += 8
22     return e
23
24 def jbinstring_to_bytes(t):
25     e = []
26     for n in range(len(t)):
27         e.append(ord(t[n]) & 255)
28     return e
29
30 # 没问题
31 def estring_to_bytes(t):
32     return jbinstring_to_bytes(urllib.parse.unquote(urllib.parse.quote(t)))
33
34 def _ff(t, e, n, r, o, i, a):
35     # 计算中间值 c
36     c = t + ((e & n) | (~e & r)) + (o & 0xFFFFFFFF) + a
37     # 将 c 转换为 32 位无符号整数
38     c = c & 0xFFFFFFFF
39     # 左移和右移操作
40     c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
```

```
41     # 返回结果
42     return (c + e) & 0xFFFFFFFF
43
44 def _gg(t, e, n, r, o, i, a):
45     # 计算中间值 c
46     c = t + ((e & r) | (n & ~r)) + (o & 0xFFFFFFFF) + a
47     # 将 c 转换为 32 位无符号整数
48     c = c & 0xFFFFFFFF
49     # 左移和右移操作
50     c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
51     # 返回结果
52     return (c + e) & 0xFFFFFFFF
53
54 def _hh(t, e, n, r, o, i, a):
55     # 计算中间值 c
56     c = t + (e ^ n ^ r) + (o & 0xFFFFFFFF) + a
57     # 将 c 转换为 32 位无符号整数
58     c = c & 0xFFFFFFFF
59     # 左移和右移操作
60     c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
61     # 返回结果
62     return (c + e) & 0xFFFFFFFF
63
64 def _ii(t, e, n, r, o, i, a):
65     # 计算中间值 c
66     c = t + (n ^ (e | ~r)) + (o & 0xFFFFFFFF) + a
67     # 将 c 转换为 32 位无符号整数
68     c = c & 0xFFFFFFFF
69     # 左移和右移操作
70     c = (c << i | c >> (32 - i)) & 0xFFFFFFFF
71     # 返回结果
72     return (c + e) & 0xFFFFFFFF
73
74 def o(i, a):
75     if isinstance(i, str):
76         i = estring_to_bytes(i)
77     elif isinstance(i, (list, tuple)):
78         i = list(i)
79     elif not isinstance(i, (list, bytearray)):
80         i = str(i)
81     c = tbytes_to_words(i)
82     u = 8 * len(i)
83     s, l, f, p = 1732584193, -271733879, -1732584194, 271733878
84
85     for d in range(len(c)):
86         c[d] = (16711935 & (c[d] << 8 | c[d] >> 24)) | (4278255360 & (c[d] <<
87         24 | c[d] >> 8))
```

```
87
88     # 确保列表 c 的长度足够大
89     while len(c) <= (14 + ((u + 64 >> 9) << 4)):
90         c.append(0)
91
92     c[u >> 5] |= 128 << (u % 32)
93     c[14 + ((u + 64 >> 9) << 4)] = u
94
95     h, v, y, m = _ff, _gg, _hh, _ii
96     for d in range(0, len(c), 16):
97         g, b, w, A = s, l, f, p
98         # 确保在访问索引之前扩展列表的长度
99         while len(c) <= d + 15:
100             c.append(0)
101             s = h(s, l, f, p, c[d + 0], 7, -680876936)
102             p = h(p, s, l, f, c[d + 1], 12, -389564586)
103             f = h(f, p, s, l, c[d + 2], 17, 606105819)
104             l = h(l, f, p, s, c[d + 3], 22, -1044525330)
105             s = h(s, l, f, p, c[d + 4], 7, -176418897)
106             p = h(p, s, l, f, c[d + 5], 12, 1200080426)
107             f = h(f, p, s, l, c[d + 6], 17, -1473231341)
108             l = h(l, f, p, s, c[d + 7], 22, -45705983)
109             s = h(s, l, f, p, c[d + 8], 7, 1770035416)
110             p = h(p, s, l, f, c[d + 9], 12, -1958414417)
111             f = h(f, p, s, l, c[d + 10], 17, -42063)
112             l = h(l, f, p, s, c[d + 11], 22, -1990404162)
113             s = h(s, l, f, p, c[d + 12], 7, 1804603682)
114             p = h(p, s, l, f, c[d + 13], 12, -40341101)
115             f = h(f, p, s, l, c[d + 14], 17, -1502002290)
116             s = v(s, l := h(l, f, p, s, c[d + 15], 22, 1236535329), f, p, c[d +
117             1], 5, -165796510)
118             p = v(p, s, l, f, c[d + 6], 9, -1069501632)
119             f = v(f, p, s, l, c[d + 11], 14, 643717713)
120             l = v(l, f, p, s, c[d + 0], 20, -373897302)
121             s = v(s, l, f, p, c[d + 5], 5, -701558691)
122             p = v(p, s, l, f, c[d + 10], 9, 38016083)
123             f = v(f, p, s, l, c[d + 15], 14, -660478335)
124             l = v(l, f, p, s, c[d + 4], 20, -405537848)
125             s = v(s, l, f, p, c[d + 9], 5, 568446438)
126             p = v(p, s, l, f, c[d + 14], 9, -1019803690)
127             f = v(f, p, s, l, c[d + 3], 14, -187363961)
128             l = v(l, f, p, s, c[d + 8], 20, 1163531501)
129             s = v(s, l, f, p, c[d + 13], 5, -1444681467)
130             p = v(p, s, l, f, c[d + 2], 9, -51403784)
131             f = v(f, p, s, l, c[d + 7], 14, 1735328473)
132             s = y(s, l := v(l, f, p, s, c[d + 12], 20, -1926607734), f, p, c[d +
133             5], 4, -378558)
```

```

132     p = y(p, s, l, f, c[d + 8], 11, -2022574463)
133     f = y(f, p, s, l, c[d + 11], 16, 1839030562)
134     l = y(l, f, p, s, c[d + 14], 23, -35309556)
135     s = y(s, l, f, p, c[d + 1], 4, -1530992060)
136     p = y(p, s, l, f, c[d + 4], 11, 1272893353)
137     f = y(f, p, s, l, c[d + 7], 16, -155497632)
138     l = y(l, f, p, s, c[d + 10], 23, -1094730640)
139     s = y(s, l, f, p, c[d + 13], 4, 681279174)
140     p = y(p, s, l, f, c[d + 0], 11, -358537222)
141     f = y(f, p, s, l, c[d + 3], 16, -722521979)
142     l = y(l, f, p, s, c[d + 6], 23, 76029189)
143     s = y(s, l, f, p, c[d + 9], 4, -640364487)
144     p = y(p, s, l, f, c[d + 12], 11, -421815835)
145     f = y(f, p, s, l, c[d + 15], 16, 530742520)
146     s = m(s, l := y(l, f, p, s, c[d + 2], 23, -995338651), f, p, c[d + 0],
6, -198630844)
147     p = m(p, s, l, f, c[d + 7], 10, 1126891415)
148     f = m(f, p, s, l, c[d + 14], 15, -1416354905)
149     l = m(l, f, p, s, c[d + 5], 21, -57434055)
150     s = m(s, l, f, p, c[d + 12], 6, 1700485571)
151     p = m(p, s, l, f, c[d + 3], 10, -1894986606)
152     f = m(f, p, s, l, c[d + 10], 15, -1051523)
153     l = m(l, f, p, s, c[d + 1], 21, -2054922799)
154     s = m(s, l, f, p, c[d + 8], 6, 1873313359)
155     p = m(p, s, l, f, c[d + 15], 10, -30611744)
156     f = m(f, p, s, l, c[d + 6], 15, -1560198380)
157     l = m(l, f, p, s, c[d + 13], 21, 1309151649)
158     s = m(s, l, f, p, c[d + 4], 6, -145523070)
159     p = m(p, s, l, f, c[d + 11], 10, -1120210379)
160     f = m(f, p, s, l, c[d + 2], 15, 718787259)
161     l = m(l, f, p, s, c[d + 9], 21, -343485551)
162
163     s = (s + g) >> 0 & 0xFFFFFFFF
164     l = (l + b) >> 0 & 0xFFFFFFFF
165     f = (f + w) >> 0 & 0xFFFFFFFF
166     p = (p + A) >> 0 & 0xFFFFFFFF
167
168     return tendian([s, l, f, p])
169
170 def twords_to_bytes(t):
171     e = []
172     for n in range(0, 32 * len(t), 8):
173         e.append((t[n >> 5] >> (24 - n % 32)) & 255)
174     return e
175
176 def tbytes_to_hex(t):
177     e = []

```

```

178     for n in range(len(t)):
179         e.append(hex(t[n] >> 4)[2:])
180         e.append(hex(t[n] & 15)[2:])
181     return ''.join(e)
182
183 def get_wrid(e):
184     n = None
185     i = twords_to_bytes(o(e, n))
186     return tbytes_to_hex(i)

```

hybrid

- hybrid_crawler.py

代码块

```

1 import asyncio
2 import re
3 import httpx
4
5 # from crawlers.douyin.web.web_crawler import DouyinWebCrawler # 导入抖音Web爬虫
6 # from crawlers.tiktok.web.web_crawler import TikTokWebCrawler # 导入TikTok Web爬虫
7 # from crawlers.tiktok.app.app_crawler import TikTokAPPCrawler # 导入TikTok App爬虫
8 from data_collection_service.crawlers.bilibili.web_crawler import
BilibiliWebCrawler # 导入Bilibili Web爬虫
9
10
11 class HybridCrawler:
12     def __init__(self):
13         # self.DouyinWebCrawler = DouyinWebCrawler()
14         # self.TikTokWebCrawler = TikTokWebCrawler()
15         # self.TikTokAPPCrawler = TikTokAPPCrawler()
16         self.BilibiliWebCrawler = BilibiliWebCrawler()
17
18     async def get_bilibili_bv_id(self, url: str) -> str:
19         """
20             从 Bilibili URL 中提取 BV 号，支持短链重定向
21         """
22         # 如果是 b23.tv 短链，需要重定向获取真实URL
23         if "b23.tv" in url:
24             async with httpx.AsyncClient() as client:
25                 response = await client.head(url, follow_redirects=True)
26                 url = str(response.url)
27
28         # 从URL中提取BV号

```

```
29         bv_pattern = r'(?:video\/|\/)(BV[A-Za-z0-9]+)'  
30         match = re.search(bv_pattern, url)  
31         if match:  
32             return match.group(1)  
33         else:  
34             raise ValueError(f"Cannot extract BV ID from URL: {url}")  
35  
36     async def hybrid_parsing_single_video(self, url: str, minimal: bool =  
37         False):  
38         # 解析抖音视频/Parse Douyin video  
39         if "douyin" in url:  
40             platform = "douyin"  
41             aweme_id = await self.DouyinWebCrawler.get_aweme_id(url)  
42             data = await self.DouyinWebCrawler.fetch_one_video(aweme_id)  
43             data = data.get("aweme_detail")  
44             # $.aweme_detail.aweme_type  
45             aweme_type = data.get("aweme_type")  
46         # 解析TikTok视频/Parse TikTok video  
47         elif "tiktok" in url:  
48             platform = "tiktok"  
49             aweme_id = await self.TikTokWebCrawler.get_aweme_id(url)  
50  
51             # 2024-09-14: Switch to TikTokAPPCrawler instead of  
52             # TikTokWebCrawler  
53             # data = await self.TikTokWebCrawler.fetch_one_video(aweme_id)  
54             # data = data.get("itemInfo").get("itemStruct")  
55  
56             data = await self.TikTokAPPCrawler.fetch_one_video(aweme_id)  
57             # $.imagePost exists if aweme_type is photo  
58             aweme_type = data.get("aweme_type")  
59         # 解析Bilibili视频/Parse Bilibili video  
60         elif "bilibili" in url or "b23.tv" in url:  
61             platform = "bilibili"  
62             aweme_id = await self.get_bilibili_bv_id(url) # BV号作为统一的  
63             video_id  
64             response = await self.BilibiliWebCrawler.fetch_one_video(aweme_id)  
65             data = response.get('data', {}) # 提取data部分  
66             # Bilibili只有视频类型, aweme_type设为0(video)  
67             aweme_type = 0  
68         else:  
69             raise ValueError("hybrid_parsing_single_video: Cannot judge the  
70             video source from the URL.")  
71  
72         # 检查是否需要返回最小数据/Check if minimal data is required  
73         if not minimal:  
74             return data
```

```
72     # 如果是最小数据, 处理数据/If it is minimal data, process the data
73     url_type_code_dict = {
74         # common
75         0: 'video',
76         # Douyin
77         2: 'image',
78         4: 'video',
79         68: 'image',
80         # TikTok
81         51: 'video',
82         55: 'video',
83         58: 'video',
84         61: 'video',
85         150: 'image'
86     }
87     # 判断链接类型/Judge link type
88     url_type = url_type_code_dict.get(aweme_type, 'video')
89     # print(f"url_type: {url_type}")
90
91     """
92     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.
93     The following are four methods of (video || image) data processing.
94     If you need to customize data processing, please modify it here.
95     """
96
97     """
98     创建已知数据字典(索引相同), 稍后使用.update()方法更新数据
99     Create a known data dictionary (index the same),
100    and then use the .update() method to update the data
101    """
102
103    # 根据平台适配字段映射
104    if platform == 'bilibili':
105        result_data = {
106            'type': url_type,
107            'platform': platform,
108            'video_id': aweme_id,
109            'desc': data.get("title"), # Bilibili使用title
110            'create_time': data.get("pubdate"), # Bilibili使用pubdate
111            'author': data.get("owner"), # Bilibili使用owner
112            'music': None, # Bilibili没有音乐信息
113            'statistics': data.get("stat"), # Bilibili使用stat
114            'cover_data': {}, # 将在各平台处理中填充
115            'hashtags': None, # Bilibili没有hashtags概念
116        }
117    else:
118        result_data = {
```

```
119         'type': url_type,
120         'platform': platform,
121         'video_id': aweme_id, # 统一使用video_id字段, 内容可能是aweme_id
122         或bv_id
123         'desc': data.get("desc"),
124         'create_time': data.get("create_time"),
125         'author': data.get("author"),
126         'music': data.get("music"),
127         'statistics': data.get("statistics"),
128         'cover_data': {}, # 将在各平台处理中填充
129         'hashtags': data.get('text_extra'),
130     }
131     # 创建一个空变量, 稍后使用.update()方法更新数据/Create an empty variable
132     # and use the .update() method to update the data
133     api_data = None
134     # 判断链接类型并处理数据/Judge link type and process data
135     # 抖音数据处理/Douyin data processing
136     if platform == 'douyin':
137         # 填充封面数据
138         result_data['cover_data'] = {
139             'cover': data.get("video", {}).get("cover"),
140             'origin_cover': data.get("video", {}).get("origin_cover"),
141             'dynamic_cover': data.get("video", {}).get("dynamic_cover")
142         }
143         # 抖音视频数据处理/Douyin video data processing
144         if url_type == 'video':
145             # 将信息储存在字典中/Store information in a dictionary
146             uri = data['video']['play_addr']['uri']
147             wm_video_url_HQ = data['video']['play_addr']['url_list'][0]
148             wm_video_url = f"https://aweme.snssdk.com/aweme/v1/playwm??
149             video_id={uri}&radio=1080p&line=0"
150             nwm_video_url_HQ = wm_video_url_HQ.replace('playwm', 'play')
151             nwm_video_url = f"https://aweme.snssdk.com/aweme/v1/play??
152             video_id={uri}&ratio=1080p&line=0"
153             api_data = {
154                 'video_data':
155                 {
156                     'wm_video_url': wm_video_url,
157                     'wm_video_url_HQ': wm_video_url_HQ,
158                     'nwm_video_url': nwm_video_url,
159                     'nwm_video_url_HQ': nwm_video_url_HQ
160                 }
161             }
162             # 抖音图片数据处理/Douyin image data processing
163             elif url_type == 'image':
164                 # 无水印图片列表/No watermark image list
165                 no_watermark_image_list = []
```

```
162             # 有水印图片列表/With watermark image list
163             watermark_image_list = []
164             # 遍历图片列表/Traverse image list
165             for i in data['images']:
166                 no_watermark_image_list.append(i['url_list'][0])
167                 watermark_image_list.append(i['download_url_list'][0])
168             api_data = {
169                 'image_data':
170                 {
171                     'no_watermark_image_list': no_watermark_image_list,
172                     'watermark_image_list': watermark_image_list
173                 }
174             }
175             # TikTok数据处理/TikTok data processing
176             elif platform == 'tiktok':
177                 # 填充封面数据
178                 result_data['cover_data'] = {
179                     'cover': data.get("video", {}).get("cover"),
180                     'origin_cover': data.get("video", {}).get("origin_cover"),
181                     'dynamic_cover': data.get("video", {}).get("dynamic_cover")
182                 }
183                 # TikTok视频数据处理/TikTok video data processing
184                 if url_type == 'video':
185                     # 将信息储存在字典中/Store information in a dictionary
186                     # wm_video = data['video']['downloadAddr']
187                     # wm_video = data['video']['download_addr']['url_list'][0]
188                     wm_video = (
189                         data.get('video', {})
190                         .get('download_addr', {})
191                         .get('url_list', [None])[0]
192                     )
193
194                     api_data = {
195                         'video_data':
196                         {
197                             'wm_video_url': wm_video,
198                             'wm_video_url_HQ': wm_video,
199                             # 'nwm_video_url': data['video']['playAddr'],
200                             'nwm_video_url': data['video']['play_addr']
201                             ['url_list'][0],
202                                     # 'nwm_video_url_HQ': data['video']['bitrateInfo']
203                                     [0]['PlayAddr']['UrlList'][0]
204                                     'nwm_video_url_HQ': data['video']['bit_rate'][0]
205                                     ['play_addr']['url_list'][0]
206                         }
207                     }
208                     # TikTok图片数据处理/TikTok image data processing
```

```
206     elif url_type == 'image':
207         # 无水印图片列表/No watermark image list
208         no_watermark_image_list = []
209         # 有水印图片列表/With watermark image list
210         watermark_image_list = []
211         for i in data['image_post_info']['images']:
212             no_watermark_image_list.append(i['display_image'])
213             ['url_list'][0])
214             watermark_image_list.append(i['owner_watermark_image'])
215             ['url_list'][0])
216             api_data = {
217                 'image_data': {
218                     }
219                     'no_watermark_image_list': no_watermark_image_list,
220                     'watermark_image_list': watermark_image_list
221                     }
222                     }
223 # Bilibili数据处理/Bilibili data processing
224 elif platform == 'bilibili':
225     # 填充封面数据
226     result_data['cover_data'] = {
227         'cover': data.get("pic"), # Bilibili使用pic作为封面
228         'origin_cover': data.get("pic"),
229         'dynamic_cover': data.get("pic")
230     }
231     # Bilibili只有视频，直接处理视频数据
232     if url_type == 'video':
233         # 获取视频播放地址需要额外调用API
234         cid = data.get('cid') # 获取cid
235         if cid:
236             # 获取播放链接，cid需要转换为字符串
237             playurl_data = await
238             self.BilibiliWebCrawler.fetch_video_playurl(aweme_id, str(cid))
239             # 从播放数据中提取URL
240             dash = playurl_data.get('data', {}).get('dash', {})
241             video_list = dash.get('video', [])
242             audio_list = dash.get('audio', [])
243             # 选择最高质量的视频流
244             video_url = video_list[0].get('baseUrl') if video_list else
245             None
246             audio_url = audio_list[0].get('baseUrl') if audio_list else
247             None
248             api_data = {
249                 'video_data': {
250                     'wm_video_url': video_url,
```

```

248             'wm_video_url_HQ': video_url,
249             'nwm_video_url': video_url, # Bilibili没有水印概念
250             'nwm_video_url_HQ': video_url,
251             'audio_url': audio_url, # Bilibili音视频分离
252             'cid': cid, # 保存cid供后续使用
253         }
254     }
255 else:
256     api_data = {
257         'video_data': {
258             'wm_video_url': None,
259             'wm_video_url_HQ': None,
260             'nwm_video_url': None,
261             'nwm_video_url_HQ': None,
262             'error': 'Failed to get cid for video playback'
263         }
264     }
265 # 更新数据/Update data
266 result_data.update(api_data)
267 return result_data
268
269 async def main(self):
270     # 测试混合解析单一视频接口/Test hybrid parsing single video endpoint
271     # url = "https://v.douyin.com/L4FJNR3/"
272     # url = "https://www.tiktok.com/@taylorswift/video/7359655005701311786"
273     url = "https://www.tiktok.com/@flukegk83/video/7360734489271700753"
274     # url = "https://www.tiktok.com/@minecraft/photo/7369296852669205791"
275     minimal = True
276     result = await self.hybrid_parsing_single_video(url, minimal=minimal)
277     # print(result)
278
279     # 占位
280     pass
281
282
283 if __name__ == '__main__':
284     # 实例化混合爬虫/Instantiate hybrid crawler
285     hybird_crawler = HybridCrawler()
286     # 运行测试代码/Run test code
287     asyncio.run(hybird_crawler.main())

```

utils

- api_exceptions.py

代码块

```
1 class APIError(Exception):
2     """基本API异常类，其他API异常都会继承这个类"""
3
4     def __init__(self, status_code=None):
5         self.status_code = status_code
6         print(
7             "程序出现异常，请检查错误信息。"
8         )
9
10    def display_error(self):
11        """显示错误信息和状态码（如果有的话）"""
12        return f"Error: {self.args[0]}." + (
13            f" Status Code: {self.status_code}." if self.status_code else ""
14        )
15
16
17 class APIConnectionError(APIError):
18     """当与API的连接出现问题时抛出"""
19
20     def display_error(self):
21         return f"API Connection Error: {self.args[0]}."
22
23
24 class APIUnavailableError(APIError):
25     """当API服务不可用时抛出，例如维护或超时"""
26
27     def display_error(self):
28         return f"API Unavailable Error: {self.args[0]}."
29
30
31 class APINotFoundError(APIError):
32     """当API端点不存在时抛出"""
33
34     def display_error(self):
35         return f"API Not Found Error: {self.args[0]}."
36
37
38 class APIResponseError(APIError):
39     """当API返回的响应与预期不符时抛出"""
40
41     def display_error(self):
42         return f"API Response Error: {self.args[0]}."
43
44
45 class APIRateLimitError(APIError):
46     """当达到API的请求速率限制时抛出"""
47
```

```
48     def display_error(self):
49         return f"API Rate Limit Error: {self.args[0]}."
50
51
52 class APITimeoutError(APIError):
53     """当API请求超时时抛出"""
54
55     def display_error(self):
56         return f"API Timeout Error: {self.args[0]}."
57
58
59 class APIUnauthorizedError(APIError):
60     """当API请求由于授权失败而被拒绝时抛出"""
61
62     def display_error(self):
63         return f"API Unauthorized Error: {self.args[0]}."
64
65
66 class APIRetryExhaustedError(APIError):
67     """当API请求重试次数用尽时抛出"""
68
69     def display_error(self):
70         return f"API Retry Exhausted Error: {self.args[0]}."
```

- deprecated.py

代码块

```
1 import warnings
2 import functools
3
4
5 def deprecated(message):
6     def decorator(func):
7         @functools.wraps(func)
8         async def wrapper(*args, **kwargs):
9             warnings.warn(
10                 f"{func.__name__} is deprecated: {message}",
11                 DeprecationWarning,
12                 stacklevel=2
13             )
14             return await func(*args, **kwargs)
15
16     return wrapper
17
18 return decorator
```

- logger.py

代码块

```
1 import threading
2 import time
3 import logging
4 import datetime
5
6 from pathlib import Path
7 from rich.logging import RichHandler
8 from logging.handlers import TimedRotatingFileHandler
9
10
11 class Singleton(type):
12     _instances = {} # 存储实例的字典
13     _lock: threading.Lock = threading.Lock() # 线程锁
14
15     def __init__(self, *args, **kwargs):
16         super().__init__(*args, **kwargs)
17
18     def __call__(cls, *args, **kwargs):
19         """
20             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。
21             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。
22         """
23         key = (cls, args, frozenset(kwargs.items()))
24         with cls._lock:
25             if key not in cls._instances:
26                 instance = super().__call__(*args, **kwargs)
27                 cls._instances[key] = instance
28         return cls._instances[key]
29
30     @classmethod
31     def reset_instance(cls, *args, **kwargs):
32         """
33             重置指定参数的实例。这只是从 _instances 字典中删除实例的引用，
34             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。
35         """
36         key = (cls, args, frozenset(kwargs.items()))
37         with cls._lock:
38             if key in cls._instances:
39                 del cls._instances[key]
40
41
42 class LogManager(metaclass=Singleton):
```

```
43     def __init__(self):
44         if getattr(self, "_initialized", False): # 防止重复初始化
45             return
46
47         self.logger = logging.getLogger("Douyin_TikTok_Download_API_Crawlers")
48         self.logger.setLevel(logging.INFO)
49         self.log_dir = None
50         self._initialized = True
51
52     def setup_logging(self, level=logging.INFO, log_to_console=False,
53                      log_path=None):
54         self.logger.handlers.clear()
55         self.logger.setLevel(level)
56
56         if log_to_console:
57             ch = RichHandler(
58                 show_time=False,
59                 show_path=False,
60                 markup=True,
61                 keywords=(RichHandler.KEYWORDS or []) + ["STREAM"],
62                 rich_tracebacks=True,
63             )
64             ch.setFormatter(logging.Formatter("{message}", style="{", datefmt=
65             "[%X]"))
66             self.logger.addHandler(ch)
67
67         if log_path:
68             self.log_dir = Path(log_path)
69             self.ensure_log_dir_exists(self.log_dir)
70             log_file_name = datetime.datetime.now().strftime("%Y-%m-%d-%H-%M-
71             %S.log")
71             log_file = self.log_dir.joinpath(log_file_name)
72             fh = TimedRotatingFileHandler(
73                 log_file, when="midnight", interval=1, backupCount=99,
74                 encoding="utf-8"
75             )
76             fh.setFormatter(
77                 logging.Formatter(
78                     "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
79                 )
80             )
81             self.logger.addHandler(fh)
82
82     @staticmethod
83     def ensure_log_dir_exists(log_path: Path):
84         log_path.mkdir(parents=True, exist_ok=True)
```

```
86     def clean_logs(self, keep_last_n=10):
87         """保留最近的n个日志文件并删除其他文件"""
88         if not self.log_dir:
89             return
90         # self.shutdown()
91         all_logs = sorted(self.log_dir.glob("*.log"))
92         if keep_last_n == 0:
93             files_to_delete = all_logs
94         else:
95             files_to_delete = all_logs[:-keep_last_n]
96         for log_file in files_to_delete:
97             try:
98                 log_file.unlink()
99             except PermissionError:
100                 self.logger.warning(
101                     f"无法删除日志文件 {log_file}，它正被另一个进程使用"
102                 )
103
104     def shutdown(self):
105         for handler in self.logger.handlers:
106             handler.close()
107             self.logger.removeHandler(handler)
108         self.logger.handlers.clear()
109         time.sleep(1) # 确保文件被释放
110
111
112     def log_setup(log_to_console=True):
113         logger = logging.getLogger("Douyin_TikTok_Download_API_Crawlers")
114         if logger.hasHandlers():
115             # logger已经被设置，不做任何操作
116             return logger
117
118         # 创建临时的日志目录
119         temp_log_dir = Path("./logs")
120         temp_log_dir.mkdir(exist_ok=True)
121
122         # 初始化日志管理器
123         log_manager = LogManager()
124         log_manager.setup_logging(
125             level=logging.INFO, log_to_console=log_to_console,
126             log_path=temp_log_dir
127         )
128
129         # 只保留1000个日志文件
130         log_manager.clean_logs(1000)
131
132
133     return logger
```

```
132  
133  
134     logger = log_setup()
```

- utils.py

代码块

```
1  import re  
2  import sys  
3  import random  
4  import secrets  
5  import datetime  
6  import browser_cookie3  
7  import importlib_resources  
8  
9  from pydantic import BaseModel  
10  
11 from urllib.parse import quote, urlencode # URL编码  
12 from typing import Union, List, Any  
13 from pathlib import Path  
14  
15 # 生成一个 16 字节的随机字节串 (Generate a random byte string of 16 bytes)  
16 seed_bytes = secrets.token_bytes(16)  
17  
18 # 将字节字符串转换为整数 (Convert the byte string to an integer)  
19 seed_int = int.from_bytes(seed_bytes, "big")  
20  
21 # 设置随机种子 (Seed the random module)  
22 random.seed(seed_int)  
23  
24  
25 # 将模型实例转换为字典  
26 def model_to_query_string(model: BaseModel) -> str:  
27     model_dict = model.dict()  
28     # 使用urlencode进行URL编码  
29     query_string = urlencode(model_dict)  
30     return query_string  
31  
32  
33 def gen_random_str(randomlength: int) -> str:  
34     """  
35         根据传入长度产生随机字符串 (Generate a random string based on the given  
36         length)  
37     Args:38
```

```
38     randomlength (int): 需要生成的随机字符串的长度 (The length of the random
39     string to be generated)
40
41     Returns:
42         str: 生成的随机字符串 (The generated random string)
43         """
44
45     base_str =
46         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-"
47
48     return "".join(random.choice(base_str) for _ in range(randomlength))
49
50
51
52 def get_timestamp(unit: str = "milli"):
53     """
54     根据给定的单位获取当前时间 (Get the current time based on the given unit)
55
56     Args:
57         unit (str): 时间单位, 可以是 "milli"、"sec"、"min" 等
58             (The time unit, which can be "milli", "sec", "min", etc.)
59
60     Returns:
61         int: 根据给定单位的当前时间 (The current time based on the given unit)
62         """
63
64     now = datetime.datetime.utcnow() - datetime.datetime(1970, 1, 1)
65     if unit == "milli":
66         return int(now.total_seconds() * 1000)
67     elif unit == "sec":
68         return int(now.total_seconds())
69     elif unit == "min":
70         return int(now.total_seconds() / 60)
71     else:
72         raise ValueError("Unsupported time unit")
73
74
75 def timestamp_2_str(
76     timestamp: Union[str, int, float], format: str = "%Y-%m-%d %H-%M-%S"
77 ) -> str:
78     """
79     将 UNIX 时间戳转换为格式化字符串 (Convert a UNIX timestamp to a formatted
80     string)
81
82     Args:
83         timestamp (int): 要转换的 UNIX 时间戳 (The UNIX timestamp to be
84             converted)
85         format (str, optional): 返回的日期时间字符串的格式。
86             默认为 '%Y-%m-%d %H-%M-%S'。
```

```
81                                         (The format for the returned date-time string  
82                                         Defaults to '%Y-%m-%d %H:%M:%S')  
83  
84     Returns:  
85         str: 格式化的日期时间字符串 (The formatted date-time string)  
86         """  
87         if timestamp is None or timestamp == "None":  
88             return ""  
89  
90         if isinstance(timestamp, str):  
91             if len(timestamp) == 30:  
92                 return datetime.datetime.strptime(timestamp, "%a %b %d %H:%M:%S %z  
%Y")  
93  
94         return datetime.datetime.fromtimestamp(float(timestamp)).strftime(format)  
95  
96  
97     def num_to_base36(num: int) -> str:  
98         """数字转换成base32 (Convert number to base 36)"""  
99  
100    base_str = "0123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"  
101  
102    if num == 0:  
103        return "0"  
104  
105    base36 = []  
106    while num:  
107        num, i = divmod(num, 36)  
108        base36.append(base_str[i])  
109  
110    return "".join(reversed(base36))  
111  
112  
113    def split_set_cookie(cookie_str: str) -> str:  
114        """  
115            拆分Set-Cookie字符串并拼接 (Split the Set-Cookie string and concatenate)  
116  
117            Args:  
118                cookie_str (str): 待拆分的Set-Cookie字符串 (The Set-Cookie string to be  
119                split)  
120  
121            Returns:  
122                str: 拼接后的Cookie字符串 (Concatenated cookie string)  
123                """  
124  
125                # 判断是否为字符串 / Check if it's a string  
126                if not isinstance(cookie_str, str):
```

```
126         raise TypeError(`set-cookie` must be str)
127
128     # 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-
129     # Cookie string, avoiding incorrect splitting on the value of the 'expires'
130     # field)
131     # 拆分每个Cookie字符串, 只获取第一个分段 (即key=value部分) / Split each Cookie
132     # string, only getting the first segment (i.e., key=value part)
133     # 拼接所有的Cookie (Concatenate all cookies)
134     return ";" .join(
135         cookie.split(";" )[0] for cookie in re.split(", (?=[a-zA-Z])", cookie_str)
136     )
137
138
139
140     def split_dict_cookie(cookie_dict: dict) -> str:
141         return ";" .join(f"{key}={value}" for key, value in cookie_dict.items())
142
143
144     def extract_valid_urls(inputs: Union[str, List[str]]) -> Union[str, List[str], None]:
145         """从输入中提取有效的URL (Extract valid URLs from input)
146
147         Args:
148             inputs (Union[str, list[str]]): 输入的字符串或字符串列表 (Input string or
149                 list of strings)
150
151         Returns:
152             Union[str, list[str]]: 提取出的有效URL或URL列表 (Extracted valid URL or
153                 list of URLs)
154         """
155
156         url_pattern = re.compile(r"https?://\S+")
157
158         # 如果输入是单个字符串
159         if isinstance(inputs, str):
160             match = url_pattern.search(inputs)
161             return match.group(0) if match else None
162
163         # 如果输入是字符串列表
164         elif isinstance(inputs, list):
165             valid_urls = []
166
167             for input_str in inputs:
168                 matches = url_pattern.findall(input_str)
169                 if matches:
170                     valid_urls.extend(matches)
171
172             return valid_urls
```

```
166
167
168 def _get_first_item_from_list(_list) -> list:
169     # 检查是否是列表 (Check if it's a list)
170     if _list and isinstance(_list, list):
171         # 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
172         # (If the first one in the list is still a list then bring up the
173         # first value of each list)
174         if isinstance(_list[0], list):
175             return [inner[0] for inner in _list if inner]
176             # 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
177             # (If it's just a regular list, return the first item wrapped in a
178             list)
179         else:
180             return [_list[0]]
181
182 def get_resource_path(filepath: str):
183     """获取资源文件的路径 (Get the path of the resource file)
184
185     Args:
186         filepath: str: 文件路径 (file path)
187     """
188
189     return importlib_resources.files("f2") / filepath
190
191
192 def replaceT(obj: Union[str, Any]) -> Union[str, Any]:
193     """
194     替换文案非法字符 (Replace illegal characters in the text)
195
196     Args:
197         obj (str): 传入对象 (Input object)
198
199     Returns:
200         new: 处理后的内容 (Processed content)
201     """
202
203     reSub = r"[\u4e00-\u9fa5a-zA-Z0-9#]"
204
205     if isinstance(obj, list):
206         return [re.sub(reSub, "_", i) for i in obj]
207
208     if isinstance(obj, str):
209         return re.sub(reSub, "_", obj)
210
```

```
211     return obj
212     # raise TypeError("输入应为字符串或字符串列表")
213
214
215 def split_filename(text: str, os_limit: dict) -> str:
216     """
217     根据操作系统的字符限制分割文件名，并用 '.....' 替换。
218
219     Args:
220         text (str): 要计算的文本
221         os_limit (dict): 操作系统的字符限制字典
222
223     Returns:
224         str: 分割后的文本
225     """
226     # 获取操作系统名称和文件名长度限制
227     os_name = sys.platform
228     filename_length_limit = os_limit.get(os_name, 200)
229
230     # 计算中文字符长度 (中文字符长度*3)
231     chinese_length = sum(1 for char in text if "\u4e00" <= char <= "\uffff") *
232     3
233     # 计算英文字符长度
234     english_length = sum(1 for char in text if char.isalpha())
235     # 计算下划线数量
236     num_underscores = text.count("_")
237
238     # 计算总长度
239     total_length = chinese_length + english_length + num_underscores
240
241     # 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
242     if total_length > filename_length_limit:
243         split_index = min(total_length, filename_length_limit) // 2 - 6
244         split_text = text[:split_index] + "....." + text[-split_index:]
245         return split_text
246     else:
247         return text
248
249 def ensure_path(path: Union[str, Path]) -> Path:
250     """确保路径是一个Path对象 (Ensure the path is a Path object)"""
251     return Path(path) if isinstance(path, str) else path
252
253
254 def get_cookie_from_browser(browser_choice: str, domain: str = "") -> dict:
255     """
256     根据用户选择的浏览器获取domain的cookie。
```

```
257
258     Args:
259         browser_choice (str): 用户选择的浏览器名称
260
261     Returns:
262         str: *.domain的cookie值
263     """
264
265     if not browser_choice or not domain:
266         return ""
267
268     BROWSER_FUNCTIONS = {
269         "chrome": browser_cookie3.chrome,
270         "firefox": browser_cookie3.firefox,
271         "edge": browser_cookie3.edge,
272         "opera": browser_cookie3.opera,
273         "opera_gx": browser_cookie3.opera_gx,
274         "safari": browser_cookie3.safari,
275         "chromium": browser_cookie3.chromium,
276         "brave": browser_cookie3.brave,
277         "vivaldi": browser_cookie3.vivaldi,
278         "librewolf": browser_cookie3.librewolf,
279     }
280     cj_function = BROWSER_FUNCTIONS.get(browser_choice)
281     cj = cj_function(domain_name=domain)
282     cookie_value = {c.name: c.value for c in cj if c.domain.endswith(domain)}
283     return cookie_value
284
285
286     def check_invalid_naming(
287         naming: str, allowed_patterns: list, allowed_separators: list
288     ) -> list:
289     """
290         检查命名是否符合命名模板 (Check if the naming conforms to the naming template)
291
292     Args:
293         naming (str): 命名字字符串 (Naming string)
294         allowed_patterns (list): 允许的模式列表 (List of allowed patterns)
295         allowed_separators (list): 允许的分隔符列表 (List of allowed separators)
296     Returns:
297         list: 无效的模式列表 (List of invalid patterns)
298     """
299     if not naming or not allowed_patterns or not allowed_separators:
300         return []
301
302     temp_naming = naming
303     invalid_patterns = []
```

```
304
305     # 检查提供的模式是否有效
306     for pattern in allowed_patterns:
307         if pattern in temp_naming:
308             temp_naming = temp_naming.replace(pattern, "")
309
310     # 此时, temp_naming应只包含分隔符
311     for char in temp_naming:
312         if char not in allowed_separators:
313             invalid_patterns.append(char)
314
315     # 检查连续的无效模式或分隔符
316     for pattern in allowed_patterns:
317         # 检查像"{xxx}{xxx}"这样的模式
318         if pattern + pattern in naming:
319             invalid_patterns.append(pattern + pattern)
320         for sep in allowed_patterns:
321             # 检查像"{xxx}-{xxx}"这样的模式
322             if pattern + sep + pattern in naming:
323                 invalid_patterns.append(pattern + sep + pattern)
324
325     return invalid_patterns
326
327
328 def merge_config(
329     main_conf: dict = ...,
330     custom_conf: dict = ...,
331     **kwargs,
332 ):
333     """
334     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。
335
336     Args:
337         main_conf (dict): 主配置参数字典
338         custom_conf (dict): 自定义配置参数字典
339         **kwargs: CLI 参数和其他额外的配置参数
340
341     Returns:
342         dict: 合并后的配置参数字典
343     """
344     # 合并主配置和自定义配置
345     merged_conf = {}
346     for key, value in main_conf.items():
347         merged_conf[key] = value    # 将主配置复制到合并后的配置中
348     for key, value in custom_conf.items():
```

```
349         if value is not None and value != "": # 只有值不为 None 和 空值，才进行合  
并  
350             merged_conf[key] = value # 自定义配置参数会覆盖主配置中的同名参数  
351  
352     # 合并 CLI 参数与合并后的配置，确保 CLI 参数的优先级最高  
353     for key, value in kwargs.items():  
354         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  
355             merged_conf[key] = value  
356         elif value is not None and value != "": # 如果值不为 None 和 空值，则进行  
合并  
357             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  
358  
359     return merged_conf
```

base_crawler.py

代码块

```
1 import httpx  
2 import json  
3 import asyncio  
4 import re  
5  
6 from httpx import Response  
7  
8 from data_collection_service.crawlers.utils.logger import logger  
9 from data_collection_service.crawlers.utils.api_exceptions import (  
10     APIError,  
11     APIConnectionError,  
12     APIResponseError,  
13     APITimeoutError,  
14     APIUnavailableError,  
15     APIUnauthorizedError,  
16     APINotFoundError,  
17     APIRateLimitError,  
18     APIRetryExhaustedError,  
19 )  
20  
21  
22 class BaseCrawler:  
23     """  
24     基础爬虫客户端 (Base crawler client)  
25     """  
26  
27     def __init__(  
28         self,
```

```
29         proxies: dict = None,
30         max_retries: int = 3,
31         max_connections: int = 50,
32         timeout: int = 10,
33         max_tasks: int = 50,
34         crawler_headers: dict = {},
35     ):
36         if isinstance(proxies, dict):
37             self.proxies = proxies
38             # [f"{k}://{v}" for k, v in proxies.items()]
39         else:
40             self.proxies = None
41
42         # 爬虫请求头 / Crawler request header
43         self.crawler_headers = crawler_headers or {}
44
45         # 异步的任务数 / Number of asynchronous tasks
46         self._max_tasks = max_tasks
47         self.semaphore = asyncio.Semaphore(max_tasks)
48
49         # 限制最大连接数 / Limit the maximum number of connections
50         self._max_connections = max_connections
51         self.limits = httpx.Limits(max_connections=max_connections)
52
53         # 业务逻辑重试次数 / Business logic retry count
54         self._max_retries = max_retries
55         # 底层连接重试次数 / Underlying connection retry count
56         self.atransport = httpx.AsyncHTTPTransport(retries=max_retries)
57
58         # 超时等待时间 / Timeout waiting time
59         self._timeout = timeout
60         self.timeout = httpx.Timeout(timeout)
61         # 异步客户端 / Asynchronous client
62         self.aclient = httpx.AsyncClient(
63             headers=self.crawler_headers,
64             proxies=self.proxies,
65             timeout=self.timeout,
66             limits=self.limits,
67             transport=self.atransport,
68         )
69
70     async def fetch_response(self, endpoint: str) -> Response:
71         """获取数据 (Get data)
72
73         Args:
74             endpoint (str): 接口地址 (Endpoint URL)
75
```

```
76     Returns:  
77         Response: 原始响应对象 (Raw response object)  
78     """  
79     return await self.get_fetch_data(endpoint)  
80  
81     async def fetch_get_json(self, endpoint: str) -> dict:  
82         """获取 JSON 数据 (Get JSON data)  
83  
84         Args:  
85             endpoint (str): 接口地址 (Endpoint URL)  
86  
87         Returns:  
88             dict: 解析后的JSON数据 (Parsed JSON data)  
89         """  
90         response = await self.get_fetch_data(endpoint)  
91         return self.parse_json(response)  
92  
93     async def fetch_post_json(self, endpoint: str, params: dict = {},  
94     data=None) -> dict:  
95         """获取 JSON 数据 (Post JSON data)  
96  
97         Args:  
98             endpoint (str): 接口地址 (Endpoint URL)  
99  
100        Returns:  
101            dict: 解析后的JSON数据 (Parsed JSON data)  
102        """  
103        response = await self.post_fetch_data(endpoint, params, data)  
104        return self.parse_json(response)  
105  
106    def parse_json(self, response: Response) -> dict:  
107        """解析JSON响应对象 (Parse JSON response object)  
108  
109        Args:  
110            response (Response): 原始响应对象 (Raw response object)  
111  
112        Returns:  
113            dict: 解析后的JSON数据 (Parsed JSON data)  
114        """  
115        if (  
116            response is not None  
117            and isinstance(response, Response)  
118            and response.status_code == 200  
119        ):  
120            try:  
121                return response.json()  
122            except json.JSONDecodeError as e:
```

```
122             # 尝试使用正则表达式匹配response.text中的json数据
123             match = re.search(r"\{.*\}", response.text)
124             try:
125                 return json.loads(match.group())
126             except json.JSONDecodeError as e:
127                 logger.error("解析 {} 接口 JSON 失败:
128 {1}".format(response.url, e))
129
130             else:
131                 if isinstance(response, Response):
132                     logger.error(
133                         "获取数据失败。状态码: {}".
134                         format(response.status_code)
135                     )
136                 else:
137                     logger.error("无效响应类型。响应类型: {}".
138                         format(type(response)))
139
140         raise APIResponseError("获取数据失败")
141
142     """"
143     获取GET端点数据 (Get GET endpoint data)
144
145     Args:
146         url (str): 端点URL (Endpoint URL)
147
148     Returns:
149         response: 响应内容 (Response content)
150
151     """
152     for attempt in range(self._max_retries):
153         try:
154             response = await self.aclient.get(url, follow_redirects=True)
155             if not response.text.strip() or not response.content:
156                 error_message = "第 {} 次响应内容为空, 状态码: {}, URL:
157 {2}".
158                 format(attempt + 1,
159                         response.status_code,
160                         response.url)
161
162             logger.warning(error_message)
163
164             if attempt == self._max_retries - 1:
165                 raise APIRetryExhaustedError(
166                     "获取端点数据失败, 次数达到上限"
167                 )
```

```
165             await asyncio.sleep(self._timeout)
166         continue
167
168     # logger.info("响应状态码: {}".format(response.status_code))
169     response.raise_for_status()
170     return response
171
172 except httpx.RequestError:
173     raise APIConnectionError("连接端点失败, 检查网络环境或代理: {} 代
理: {} 类名: {}"
174             .format(url, self.proxies,
175                     self.__class__.__name__))
176
177 except httpx.HTTPStatusError as http_error:
178     self.handle_http_status_error(http_error, url, attempt + 1)
179
180 except APIError as e:
181     e.display_error()
182
183 async def post_fetch_data(self, url: str, params: dict = {}, data=None):
184     """
185     获取POST端点数据 (Get POST endpoint data)
186
187     Args:
188         url (str): 端点URL (Endpoint URL)
189         params (dict): POST请求参数 (POST request parameters)
190
191     Returns:
192         response: 响应内容 (Response content)
193     """
194     for attempt in range(self._max_retries):
195         try:
196             response = await self.aclient.post(
197                 url,
198                 json=None if not params else dict(params),
199                 data=None if not data else data,
200                 follow_redirects=True
201             )
202             if not response.text.strip() or not response.content:
203                 error_message = "第 {} 次响应内容为空, 状态码: {}, URL:
204 {2}".format(attempt + 1,
205                         response.status_code,
206                         response.url)
```

```
207             logger.warning(error_message)
208
209         if attempt == self._max_retries - 1:
210             raise APIRetryExhaustedError(
211                 "获取端点数据失败，次数达到上限"
212             )
213
214         await asyncio.sleep(self._timeout)
215         continue
216
217     # logger.info("响应状态码: {}".format(response.status_code))
218     response.raise_for_status()
219     return response
220
221 except httpx.RequestError:
222     raise APIConnectionError(
223         "连接端点失败，检查网络环境或代理: {} 代理: {} 类名: "
224         "{}".format(url, self.proxies,
225
226         self.__class__.__name__)
227     )
228
229 except httpx.HTTPStatusError as http_error:
230     self.handle_http_status_error(http_error, url, attempt + 1)
231
232 except APIError as e:
233     e.display_error()
234
235 @asyncio.coroutine
236 def head_fetch_data(self, url):
237     """
238     获取HEAD端点数据 (Get HEAD endpoint data)
239
240     Args:
241         url (str): 端点URL (Endpoint URL)
242
243     Returns:
244         response: 响应内容 (Response content)
245     """
246
247     try:
248         response = await self.aclient.head(url)
249         # logger.info("响应状态码: {}".format(response.status_code))
250         response.raise_for_status()
251         return response
252
253     except httpx.RequestError:
254         raise APIConnectionError("连接端点失败，检查网络环境或代理: {} 代理: "
255         "{} 类名: {}".format(
```

```
251             url, self.proxies, self.__class__.__name__
252         )
253     )
254
255     except httpx.HTTPStatusError as http_error:
256         self.handle_http_status_error(http_error, url, 1)
257
258     except APIError as e:
259         e.display_error()
260
261 def handle_http_status_error(self, http_error, url: str, attempt):
262     """
263     处理HTTP状态错误 (Handle HTTP status error)
264
265     Args:
266         http_error: HTTP状态错误 (HTTP status error)
267         url: 端点URL (Endpoint URL)
268         attempt: 尝试次数 (Number of attempts)
269     Raises:
270         APIConnectionError: 连接端点失败 (Failed to connect to endpoint)
271         APIResponseError: 响应错误 (Response error)
272         APIUnavailableError: 服务不可用 (Service unavailable)
273         APINotFoundError: 端点不存在 (Endpoint does not exist)
274         APITimeoutError: 连接超时 (Connection timeout)
275         APIUnauthorizedError: 未授权 (Unauthorized)
276         APIRateLimitError: 请求频率过高 (Request frequency is too high)
277         APIRetryExhaustedError: 重试次数达到上限 (The number of retries has
reached the upper limit)
278     """
279
280     response = getattr(http_error, "response", None)
281     status_code = getattr(response, "status_code", None)
282
283     if response is None or status_code is None:
284         logger.error("HTTP状态错误: {0}, URL: {1}, 尝试次数: {2}".format(
285             http_error, url, attempt
286         )
287         )
288         raise APIResponseError(f"处理HTTP错误时遇到意外情况: {http_error}")
289
290     if status_code == 302:
291         pass
292     elif status_code == 404:
293         raise APINotFoundError(f"HTTP Status Code {status_code}")
294     elif status_code == 503:
295         raise APIUnavailableError(f"HTTP Status Code {status_code}")
296     elif status_code == 408:
297         raise APITimeoutError(f"HTTP Status Code {status_code}")
```

```

297         elif status_code == 401:
298             raise APIUnauthorizedError(f"HTTP Status Code {status_code}")
299         elif status_code == 429:
300             raise APIRateLimitError(f"HTTP Status Code {status_code}")
301     else:
302         logger.error("HTTP状态错误: {0}, URL: {1}, 尝试次数: {2}".format(
303             status_code, url, attempt
304         )
305     )
306     raise APIResponseError(f"HTTP状态错误: {status_code}")
307
308     async def close(self):
309         await self.aclient.aclose()
310
311     async def __aenter__(self):
312         return self
313
314     async def __aexit__(self, exc_type, exc_val, exc_tb):
315         await self.aclient.aclose()

```

三、基础镜像配置

docker-compose.yml

代码块

```

1   services:
2     # 关系型数据库: 存储核心结构化数据
3     mysql:
4       image: mysql:8.0
5       container_name: mysql_8.0
6       restart: always
7       environment:
8         MYSQL_ROOT_PASSWORD: root_password
9         MYSQL_DATABASE: KOL_platform
10      ports:
11        - "3306:3306"
12      volumes:
13        # 修改点: 使用具名卷mysql_data
14        - mysql_data:/var/lib/mysql
15      # ./data/mysql:/var/lib/mysql
16
17      # 缓存层: 支持高并发报价结果缓存
18      redis:
19        image: redis:7.0

```

```
20      container_name: redis_7.0
21      restart: always
22      ports:
23          - "6379:6379"
24
25      # 非关系型数据库: 存储各平台差异化的红人字段
26      mongodb:
27          image: mongo:6.0
28          container_name: mongodb_6.0
29          restart: always
30          ports:
31              - "27017:27017"
32          volumes:
33              # 修改点: 使用具名卷 mongo_data
34              - mongo_data:/data/db
35      #           - ./data/mongo:/data/db
36
37      # 时序数据库: 存储视频播放/互动监控数据
38      clickhouse:
39          image: clickhouse/clickhouse-server:23.8.16.16
40          container_name: clickhouse_23.8
41          restart: always
42          environment:
43              # 设置默认账户名为 admin
44              - CLICKHOUSE_USER=${CLICKHOUSE_USER}
45              # 设置您自定义的密码
46              - CLICKHOUSE_PASSWORD=${CLICKHOUSE_PASSWORD}
47              # 允许管理权限 (可选, 方便后续在DBeaver中管理用户)
48              - CLICKHOUSE_DEFAULT_ACCESS_MANAGEMENT=1
49          ports:
50              - "8123:8123"
51              - "9000:9000"
52          volumes:
53              # 修改点: 使用具名卷 clickhouse_data
54              - clickhouse_data:/var/lib/clickhouse
55      #           - ./data/clickhouse:/var/lib/clickhouse
56          ulimits:
57             nofile:
58                  soft: 262144
59                  hard: 262144
60
61      # 消息队列: 处理异步数据采集与通知
62      zookeeper:
63          image: zookeeper:3.9.1
64          container_name: zookeeper_3.9.1
65          restart: always
66          environment:
```

```
67      - ZOO_ENABLE_AUTH=no
68  ports:
69      - "2181:2181"
70
71 kafka:
72     image: confluentinc/cp-kafka:7.5.0 # Confluent Kafka的稳定版本
73     container_name: kafka_7.5.0
74     depends_on:
75         - zookeeper
76     ports:
77         - "9092:9092"
78     environment:
79         - KAFKA_ZOOKEEPER_CONNECT=zookeeper_3.9.1:2181 # 匹配Zookeeper的容器名（若有修改需对应）
80     #           - ALLOW_PLAINTEXT_LISTENER=yes
81     #           - KAFKA_BROKER_ID=1
82     #           - KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT
83     #           - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092 # 开发环境暴露本地
84     端口
85
86     # 服务注册与配置中心: Hertz 微服务体系的核心
87 nacos:
88     image: nacos/nacos-server:v2.2.3
89     container_name: nacos_server
90     environment:
91         - MODE=standalone # 开发环境使用单机模式
92         - NACOS_AUTH_ENABLE=false # 显示关闭鉴权，解决 NACOS_AUTH_TOKEN 报错问题
93     ports:
94         - "8848:8848"
95         - "9848:9848" # Hertz/Go 客户端通信所需端口
96 volumes:
97     mysql_data:
98     mongo_data:
99     clickhouse_data:
100    # redis_data:
```