

# 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 |   |   |   |─ wrld.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

- bilibili\_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",
description="作品id/Video id")):
16      """
17      # [中文]
18      ### 用途:
19      - 获取单个视频详情信息
20      ### 参数:
21      - bv_id: 作品id
22      ### 返回:
23      - 视频详情信息
24
25      # [示例/Example]
26      bv_id = "BV1M1421t7hT"
27      """
28      try:
29          data = await BilibiliWebCrawler.fetch_one_video(bv_id)
30          return ResponseModel(code=200,
                                router=request.url.path,
31                                data=data)
32
33      except Exception as e:
34          status_code = 400
35          detail = ErrorResponseModel(code=status_code,
36                                      router=request.url.path,
37                                      params=dict(request.query_params),
38                                      )
```

```

39         raise HTTPException(status_code=status_code, detail=detail)
40
41
42 # 获取视频流地址
43 @router.get("/fetch_video_playurl", response_model=ResponseModel, summary="获取
视频流地址/Get video playurl")
44 async def fetch_one_video(request: Request,
45                             bv_id: str = Query(example="BV1y7411Q7Eq",
description="作品id/Video id"),
46                             cid: str = Query(example="171776208", description="作品
cid/Video cid")):
47     """
48     # [中文]
49     ### 用途:
50     - 获取视频流地址
51     ### 参数:
52     - bv_id: 作品id
53     - cid: 作品cid
54     ### 返回:
55     - 视频流地址
56
57     # [示例/Example]
58     bv_id = "BV1y7411Q7Eq"
59     cid = "171776208"
60     """
61     try:
62         data = await BilibiliWebCrawler.fetch_video_playurl(bv_id, cid)
63         return ResponseModel(code=200,
64                               router=request.url.path,
65                               data=data)
66     except Exception as e:
67         status_code = 400
68         detail = ErrorResponseModel(code=status_code,
69                                     router=request.url.path,
70                                     params=dict(request.query_params),
71                                     )
72         raise HTTPException(status_code=status_code, detail=detail)
73
74
75 # 获取用户发布视频作品数据
76 @router.get("/fetch_user_post_videos", response_model=ResponseModel,
77             summary="获取用户主页作品数据/Get user homepage video data")
78 async def fetch_user_post_videos(request: Request,
79                                   uid: str = Query(example="178360345",
description="用户UID"),
80                                   pn: int = Query(default=1, description="页
码/Page number"),):

```

```

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     try:
96         data = await BilibiliWebCrawler.fetch_user_post_videos(uid, pn)
97         return ResponseModel(code=200,
98                               router=request.url.path,
99                               data=data)
100     except Exception as e:
101         status_code = 400
102         detail = ErrorResponseModel(code=status_code,
103                                     router=request.url.path,
104                                     params=dict(request.query_params),
105                                     )
106         raise HTTPException(status_code=status_code, detail=detail)
107
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     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="收藏夹id/collection folder id"),
148                                           pn: int = Query(default=1,
149                                                         description="页码/Page number")
150                                           ):
151         """
152         # [中文]
153         ### 用途:
154         - 获取指定收藏夹内视频数据
155         ### 参数:
156         - folder_id: 用户UID
157         - pn: 页码
158         ### 返回:
159         - 指定收藏夹内视频数据
160
161         # [示例/Example]
162         folder_id = "1756059545"
163         pn = 1
164         """
165         try:
166             data = await BilibiliWebCrawler.fetch_folder_videos(folder_id, pn)
167             return ResponseModel(code=200,
168                                 router=request.url.path,
169                                 data=data)
170         except Exception as e:
171             status_code = 400
172             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",
description="用户UID")):
181     """
182     # [中文]
183     ### 用途:
184     - 获取指定用户的信息
185     ### 参数:
186     - uid: 用户UID
187     ### 返回:
188     - 指定用户的个人信息
189
190     # [示例/Example]
191     uid = "178360345"
192     """
193     try:
194         data = await BilibiliWebCrawler.fetch_user_profile(uid)
195         return ResponseModel(code=200,
196                             router=request.url.path,
197                             data=data)
198     except Exception as e:
199         status_code = 400
200         detail = ErrorResponseModel(code=status_code,
201                                    router=request.url.path,
202                                    params=dict(request.query_params),
203                                    )
204         raise HTTPException(status_code=status_code, detail=detail)
205
206
207 # 获取综合热门视频信息
208 @router.get("/fetch_com_popular", response_model=ResponseModel,
209             summary="获取综合热门视频信息/Get comprehensive popular video
information")
210 async def fetch_collect_folders(request: Request,
211                                pn: int = Query(default=1, description="页
码/Page number")):
212     """
213     # [中文]

```

```

214     ### 用途:
215     - 获取综合热门视频信息
216     ### 参数:
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         ### 用途:
249         - 获取指定视频的评论
250         ### 参数:
251         - bv_id: 作品id
252         - pn: 页码
253         ### 返回:
254         - 指定视频的评论数据
255
256         # [示例/Example]
257         bv_id = "BV1M1421t7hT"
258         pn = 1
259         """
260     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     try:
299         data = await BilibiliWebCrawler.fetch_comment_reply(bv_id, pn, rpid)
300         return ResponseModel(code=200,
301                               router=request.url.path,
302                               data=data)
303     except Exception as e:
304         status_code = 400
305         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     try:
333         data = await BilibiliWebCrawler.fetch_user_dynamic(uid, offset)
334         return ResponseModel(code=200,
335                             router=request.url.path,
336                             data=data)
337     except Exception as e:
338         status_code = 400
339         detail = ErrorResponseModel(code=status_code,
340                                     router=request.url.path,
341                                     params=dict(request.query_params),
342                                     )
343         raise HTTPException(status_code=status_code, detail=detail)
344
345 # 获取视频实时弹幕
346 @router.get("/fetch_video_danmaku", response_model=ResponseModel, summary="获取
347 视频实时弹幕/Get Video Danmaku")

```

```

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

```

```
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
420     # [示例/Example]
421     room_id = "1815229528"
422     """
423     try:
424         data = await BilibiliWebCrawler.fetch_live_videos(room_id)
425         return ResponseModel(code=200,
426                               router=request.url.path,
427                               data=data)
428     except Exception as e:
429         status_code = 400
430         detail = ErrorResponseModel(code=status_code,
431                                     router=request.url.path,
432                                     params=dict(request.query_params),
433                                     )
434         raise HTTPException(status_code=status_code, detail=detail)
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
live area")
439 async def fetch_collect_folders(request: Request,
440                               area_id: str = Query(example="9",
description="直播分区id/Live area ID"),
441                               pn: int = Query(default=1, description="页
码/Page number")):
442     """
443     # [中文]
444     ### 用途:
445     - 获取指定分区正在直播的主播
446     ### 参数:
447     - area_id: 直播分区id
448     - pn: 页码
449     ### 返回:
450     - 指定分区正在直播的主播
451
452     # [示例/Example]
453     area_id = "9"
454     pn = 1
455     """
456     try:
457         data = await BilibiliWebCrawler.fetch_live_streamers(area_id, pn)
458         return ResponseModel(code=200,
459                             router=request.url.path,
460                             data=data)
461     except Exception as e:
462         status_code = 400
463         detail = ErrorResponseModel(code=status_code,
464                                     router=request.url.path,
465                                     params=dict(request.query_params),
466                                     )
467         raise HTTPException(status_code=status_code, detail=detail)
468
469 # 获取所有直播分区列表
470 @router.get("/fetch_all_live_areas", response_model=ResponseModel,
471           summary="获取所有直播分区列表/Get a list of all live areas")
472 async def fetch_collect_folders(request: Request,):
473     """
474     # [中文]
475     ### 用途:
476     - 获取所有直播分区列表
477     ### 参数:
478     ### 返回:
479     - 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         try:
517             data = await BilibiliWebCrawler.bv_to_aid(bv_id)
518             return ResponseModel(code=200,
519                                   router=request.url.path,
520                                   data=data)
521         except Exception as e:
522             status_code = 400
523             detail = ErrorResponseModel(code=status_code,
524                                         router=request.url.path,
525                                         params=dict(request.query_params),
526                                         )
527             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",
description="作品id/Video id")):
532     """
533     # [中文]
534     ### 用途:
535     - 通过bv号获得视频分p信息
536     ### 参数:
537     - bv_id: 作品id
538     ### 返回:
539     - 视频分p信息
540
541     # [示例/Example]
542     bv_id = "BV1vf421i7hV"
543     """
544     try:
545         data = await BilibiliWebCrawler.fetch_video_parts(bv_id)
546         return ResponseModel(code=200,
547                               router=request.url.path,
548                               data=data)
549     except Exception as e:
550         status_code = 400
551         detail = ErrorResponseModel(code=status_code,
552                                     router=request.url.path,
553                                     params=dict(request.query_params),
554                                     )
555         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
    ErrorResponseModel # 导入响应模型
13 from data_collection_service.crawlers.hybrid.hybrid_crawler import
    HybridCrawler # 导入混合数据爬虫
14
15 router = APIRouter()
16 HybridCrawler = HybridCrawler()
17
18 # 读取上级再上级目录的配置文件
19 config_path =
    os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__
        file__))), 'config.yaml')
20 with open(config_path, 'r', encoding='utf-8') as file:
21     config = yaml.safe_load(file)
22
23
24 async def fetch_data(url: str, headers: dict = None):
25     headers = {
26         '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'
27     } if headers is None else headers.get('headers')
28     async with httpx.AsyncClient() as client:
29         response = await client.get(url, headers=headers)
30         response.raise_for_status() # 确保响应是成功的
31     return response
32
33
34 # 下载视频专用
35 async def fetch_data_stream(url: str, request: Request, headers: dict = None,
    file_path: str = None):
36     headers = {
37         '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'
38     } if headers is None else headers.get('headers')
39     async with httpx.AsyncClient() as client:
40         # 启用流式请求
41         async with client.stream("GET", url, headers=headers) as response:
42             response.raise_for_status()
43
44             # 流式保存文件
45             async with aiofiles.open(file_path, 'wb') as out_file:
46                 async for chunk in response.aiter_bytes():
47                     if await request.is_disconnected():
48                         print("客户端断开连接, 清理未完成的文件")
49                     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:
Request, output_path: str,
57                                     headers: dict) -> bool:
58     """
59     下载并合并 Bilibili 的视频流和音频流
60     """
61     try:
62         # 创建临时文件
63         with tempfile.NamedTemporaryFile(suffix='.m4v', delete=False) as
video_temp:
64             video_temp_path = video_temp.name
65         with tempfile.NamedTemporaryFile(suffix='.m4a', delete=False) as
audio_temp:
66             audio_temp_path = audio_temp.name
67
68         # 下载视频流
69         video_success = await fetch_data_stream(video_url, request,
headers=headers, file_path=video_temp_path)
70         # 下载音频流
71         audio_success = await fetch_data_stream(audio_url, request,
headers=headers, file_path=audio_temp_path)
72
73         if not video_success or not audio_success:
74             print("Failed to download video or audio stream")
75             return False
76
77         # 使用 FFmpeg 合并视频和音频
78         ffmpeg_cmd = [
79             'ffmpeg', '-y', # -y 覆盖输出文件
80             '-i', video_temp_path, # 视频输入
81             '-i', audio_temp_path, # 音频输入
82             '-c:v', 'copy', # 复制视频编码, 不重新编码
83             '-c:a', 'copy', # 复制音频编码, 不重新编码 (保持原始质量)
84             '-f', 'mp4', # 确保输出格式为MP4
85             output_path
86         ]
87
88         print(f"FFmpeg command: {' '.join(ffmpeg_cmd)}")
89         result = subprocess.run(ffmpeg_cmd, capture_output=True, text=True)
90         print(f"FFmpeg return code: {result.returncode}")
91         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
122 example="https://www.douyin.com/video/7372484719365098803",
123                                description="视频或图片的URL地址，支持抖
124                                音|TikTok|Bilibili的分享链接，例如：https://v.douyin.com/e4J8Q7A/ 或
125                                https://www.bilibili.com/video/BV1xxxxxxxxx"),
126                                prefix: bool = True,
127                                with_watermark: bool = False):
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/BV1xxxxxxxxx
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
139     ### 返回:
140     - 返回下载的视频或图片文件响应。
141
142     # [示例/Example]
143     url: https://www.bilibili.com/video/BV1U5efz2Egn
144     """
145     # 是否开启此端点/Whether to enable this endpoint
146     if not config["API"]["Download_Switch"]:
147         code = 400
148         message = "Download endpoint is disabled in the configuration file. |
149         配置文件中已禁用下载端点。"
150         return ErrorResponseModel(code=code, message=message,
151         router=request.url.path,
152         params=dict(request.query_params))
153
154     # 开始解析数据/Start parsing data
155     try:
156         data = await HybridCrawler.hybrid_parsing_single_video(url,
157         minimal=True)
158     except Exception as e:
159         code = 400
160         return ErrorResponseModel(code=code, message=str(e),
161         router=request.url.path, params=dict(request.query_params))
162
163     # 开始下载文件/Start downloading files
164     try:
165         data_type = data.get('type')
166         platform = data.get('platform')
167         video_id = data.get('video_id') # 改为使用video_id
168         file_prefix = config.get("API").get("Download_File_Prefix") if prefix
169     else ''
170     download_path = os.path.join(config.get("API").get("Download_Path"), f"
171     {platform}_{data_type}")
172
173     # 确保目录存在/Ensure the directory exists
174     os.makedirs(download_path, exist_ok=True)
175
176     # 下载视频文件/Download video file
177     if data_type == 'video':
178         file_name = f"{file_prefix}{platform}_{video_id}.mp4" if not
179         with_watermark else f"{file_prefix}{platform}_{video_id}_watermark.mp4"
180         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,
headers=__headers, file_path=file_path)
209         if not success:
210             raise HTTPException(
211                 status_code=500,
212                 detail="An error occurred while fetching data"
213             )
214
215         # # 保存文件
216         # async with aiofiles.open(file_path, 'wb') as out_file:
217         #     await out_file.write(response.content)
218
219         # 返回文件内容
220         return FileResponse(path=file_path, filename=file_name,
media_type="video/mp4")
221
222         # 下载图片文件/Download image file
223         elif data_type == 'image':
224             # 压缩文件属性/Compress file properties
225             zip_file_name = f"{file_prefix}{platform}_{video_id}_images.zip" if
not with_watermark else f"{file_prefix}
{platform}_{video_id}_images_watermark.zip"
226             zip_file_path = os.path.join(download_path, zip_file_name)
227
228             # 判断文件是否存在, 存在就直接返回、
229             if os.path.exists(zip_file_path):
230                 return FileResponse(path=zip_file_path,
filename=zip_file_name, media_type="application/zip")
231
232             # 获取图片文件/Get image file
233             urls = data.get('image_data').get('no_watermark_image_list') if not
with_watermark else data.get(
234                 'image_data').get('watermark_image_list')
235             image_file_list = []
236             for url in urls:
237                 # 请求图片文件/Request image file
238                 response = await fetch_data(url)
239                 index = int(urls.index(url))
240                 content_type = response.headers.get('content-type')
241                 file_format = content_type.split('/')[1]
242                 file_name = f"{file_prefix}{platform}_{video_id}_{index + 1}.
{file_format}" if not with_watermark else f"{file_prefix}
{platform}_{video_id}_{index + 1}_watermark.{file_format}"
243                 file_path = os.path.join(download_path, file_name)
244                 image_file_list.append(file_path)
245
246             # 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,
media_type="application/zip")
257
258         # 异常处理/Exception handling
259     except Exception as e:
260         print(e)
261         code = 400
262         return ErrorResponseModel(code=code, message=str(e),
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:
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;
SESSDATA=f4c1988b%2C1783510832%2Cc1ccd%2A11CjA7my_d4M4B0b2moHhApExeeDewhjrESg4
2o7MhCIs4HB5jyteQSGNhH7mFhf9v-
YSVjR1aGl1TXI0N2loU25UWUJNYklVb0NCdUNQN0p3MkJfTTZVUXNTT2xvZVBaCaEtaWHh6UjJ0dTZiW
mJfOHMZUtDQmZmTVBHRzRVZTZyNmpa1BWaUtRIIEC;
bili_jct=0f31589ace86c64a8261f09fbc272321; sid=q6g0j28t
9        'user-agent': Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
Edg/143.0.0.0
10

```

```
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
14     # 作品信息 (Post Detail)
15     POST_DETAIL = f"{BILIAPI_DOMAIN}/x/web-interface/view"
16
17     # 作品视频流
18     VIDEO_PLAYURL = f"{BILIAPI_DOMAIN}/x/player/wbi/playurl"
19
20     # 用户发布视频作品数据
21     USER_POST = f"{BILIAPI_DOMAIN}/x/space/wbi/arc/search"
22
23     # 收藏夹列表
24     COLLECT_FOLDERS = f"{BILIAPI_DOMAIN}/x/v3/fav/folder/created/list-all"
25
26     # 收藏夹视频
27     COLLECT_VIDEOS = f"{BILIAPI_DOMAIN}/x/v3/fav/resource/list"
28
29     # 用户个人信息
30     USER_DETAIL = f"{BILIAPI_DOMAIN}/x/space/wbi/acc/info"
31
32     # 综合热门
33     COM_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular"
34
35     # 每周必看
36     WEEKLY_POPULAR = f"{BILIAPI_DOMAIN}/x/web-interface/popular/series/one"
37
38     # 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"],
crawler_headers=kwargs["headers"])
48         async with base_crawler as crawler:
49             # 创建请求endpoint
50             endpoint = f"{BilibiliAPIEndpoints.POST_DETAIL}?bvid={bv_id}"
51             # 发送请求, 获取请求响应结果
52             response = await crawler.fetch_get_json(endpoint)
53             return response
54
55     # 获取视频流地址
56     async def fetch_video_playurl(self, bv_id: str, cid: str, qn: str = "64") -
> dict:
57         # 获取请求头信息
58         kwargs = await self.get_bilibili_headers()
59         # 创建基础爬虫对象
60         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
crawler_headers=kwargs["headers"])
61         async with base_crawler as crawler:
62             # 通过模型生成基本请求参数
63             params = PlayUrl(bvid=bv_id, cid=cid, qn=qn)
64             # 创建请求endpoint
65             generator = EndpointGenerator(params.dict())
66             endpoint = await generator.video_playurl_endpoint()
67             # 发送请求, 获取请求响应结果
68             response = await crawler.fetch_get_json(endpoint)
69             return response
70
71     # 获取用户发布视频作品数据
72     async def fetch_user_post_videos(self, uid: str, pn: int) -> dict:
73         """
74         :param uid: 用户uid
75         :param pn: 页码
76         :return:
77         """
78         # 获取请求头信息
79         kwargs = await self.get_bilibili_headers()
80         # 创建基础爬虫对象
81         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
crawler_headers=kwargs["headers"])
82         async with base_crawler as crawler:
83             # 通过模型生成基本请求参数
84             params = UserPostVideos(mid=uid, pn=pn)
85             # 创建请求endpoint
86             generator = EndpointGenerator(params.dict())
87             endpoint = await generator.user_post_videos_endpoint()
88             # 发送请求, 获取请求响应结果
89             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"],
98 crawler_headers=kwargs["headers"])
99         async with base_crawler as crawler:
100             # 创建请求endpoint
101             endpoint = f"{BilibiliAPIEndpoints.COLLECT_FOLDERS}?up_mid={uid}"
102             # 发送请求，获取请求响应结果
103             response = await crawler.fetch_get_json(endpoint)
104             # 分析响应结果
105             result_dict = await
106 ResponseAnalyzer.collect_folders_analyze(response=response)
107             return result_dict
108
109     # 获取指定收藏夹内视频数据
110     async def fetch_folder_videos(self, folder_id: str, pn: int) -> dict:
111         """
112         :param folder_id: 收藏夹id-- 可从<获取用户所有收藏夹信息>获得
113         :param pn: 页码
114         :return:
115         """
116         # 获取请求头信息
117         kwargs = await self.get_bilibili_headers()
118         # 创建基础爬虫对象
119         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
120 crawler_headers=kwargs["headers"])
121         # 发送请求，获取请求响应结果
122         async with base_crawler as crawler:
123             endpoint = f"{BilibiliAPIEndpoints.COLLECT_VIDEOS}?media_id=
124 {folder_id}&pn={pn}&ps=20&keyword=&order=mtime&type=0&tid=0&platform=web"
125             response = await crawler.fetch_get_json(endpoint)
126             return response
127
128     # 获取指定用户的信息
129     async def fetch_user_profile(self, uid: str) -> dict:
130         # 获取请求头信息
131         kwargs = await self.get_bilibili_headers()
132         # 创建基础爬虫对象
133         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
134 crawler_headers=kwargs["headers"])
135         async with base_crawler as crawler:
136             # 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"])
147         async with base_crawler as crawler:
148             # 通过模型生成基本请求参数
149             params = ComPopular(pn=pn)
150             # 创建请求endpoint
151             generator = EndpointGenerator(params.dict())
152             endpoint = await generator.com_popular_endpoint()
153             # 发送请求, 获取请求响应结果
154             response = await crawler.fetch_get_json(endpoint)
155             return response
156
157     # 获取指定视频的评论
158     async def fetch_video_comments(self, bv_id: str, pn: int) -> dict:
159         # 评论排序 -- 1:按点赞数排序. 0:按时间顺序排序
160         sort = 1
161         # 获取请求头信息
162         kwargs = await self.get_bilibili_headers()
163         # 创建基础爬虫对象
164         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
165 crawler_headers=kwargs["headers"])
166         async with base_crawler as crawler:
167             # 创建请求endpoint
168             endpoint = f"{BilibiliAPIEndpoints.VIDEO_COMMENTS}?type=1&oid=
169 {bv_id}&sort={sort}&nohot=0&ps=20&pn={pn}"
170             # 发送请求, 获取请求响应结果
171             response = await crawler.fetch_get_json(endpoint)
172             return response
173
174     # 获取视频下指定评论的回复
175     async def fetch_comment_reply(self, bv_id: str, pn: int, rpid: str) ->
dict:
176         """
177         :param bv_id: 目标视频bv号

```

```

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=
187 {bv_id}&root={rpid}&&ps=20&pn={pn}"
188             # 发送请求, 获取请求响应结果
189             response = await crawler.fetch_get_json(endpoint)
190             return response
191
192     # 获取指定用户动态
193     async def fetch_user_dynamic(self, uid: str, offset: str) -> dict:
194         # 获取请求头信息
195         kwargs = await self.get_bilibili_headers()
196         # 创建基础爬虫对象
197         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
198 crawler_headers=kwargs["headers"])
199         async with base_crawler as crawler:
200             # 通过模型生成基本请求参数
201             params = UserDynamic(host_mid=uid, offset=offset)
202             # 创建请求endpoint
203             generator = EndpointGenerator(params.dict())
204             endpoint = await generator.user_dynamic_endpoint()
205             print(endpoint)
206             # 发送请求, 获取请求响应结果
207             response = await crawler.fetch_get_json(endpoint)
208             return response
209
210     # 获取视频实时弹幕
211     async def fetch_video_danmaku(self, cid: str):
212         # 获取请求头信息
213         kwargs = await self.get_bilibili_headers()
214         # 创建基础爬虫对象
215         base_crawler = BaseCrawler(proxies=kwargs["proxies"],
216 crawler_headers=kwargs["headers"])
217         async with base_crawler as crawler:
218             # 创建请求endpoint
219             endpoint = f"https://comment.bilibili.com/{cid}.xml"
220             # 发送请求, 获取请求响应结果
221             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"],
crawler_headers=kwargs["headers"])
272      async with base_crawler as crawler:
273          # 创建请求endpoint
274          endpoint = f"{BilibiliAPIEndpoints.VIDEO_PARTS}?bvid={bv_id}"
275          # 发送请求, 获取请求响应结果
276          response = await crawler.fetch_get_json(endpoint)
277          return response
278
279  # 获取所有直播分区列表
280  async def fetch_all_live_areas(self) -> dict:
281      # 获取请求头信息
282      kwargs = await self.get_bilibili_headers()
283      # 创建基础爬虫对象
284      base_crawler = BaseCrawler(proxies=kwargs["proxies"],
crawler_headers=kwargs["headers"])
285      async with base_crawler as crawler:
286          # 创建请求endpoint
287          endpoint = BilibiliAPIEndpoints.LIVE_AREAS
288          # 发送请求, 获取请求响应结果
289          response = await crawler.fetch_get_json(endpoint)
290          return response
291
292  if __name__ == '__main__':
293      # 初始化
294      BilibiliWebCrawler = BilibiliWebCrawler()
295
296      # 开始时间
297      start = time.time()
298
299      asyncio.run(BilibiliWebCrawler.main())
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] <<
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 =
False):
37         # 解析抖音视频/Parse Douyin video
38         if "douyin" in url:
39             platform = "douyin"
40             aweme_id = await self.DouyinWebCrawler.get_aweme_id(url)
41             data = await self.DouyinWebCrawler.fetch_one_video(aweme_id)
42             data = data.get("aweme_detail")
43             # $.aweme_detail.aweme_type
44             aweme_type = data.get("aweme_type")
45             # 解析TikTok视频/Parse TikTok video
46             elif "tiktok" in url:
47                 platform = "tiktok"
48                 aweme_id = await self.TikTokWebCrawler.get_aweme_id(url)
49
50                 # 2024-09-14: Switch to TikTokAPPCrawler instead of
TikTokWebCrawler
51                 # data = await self.TikTokWebCrawler.fetch_one_video(aweme_id)
52                 # data = data.get("itemInfo").get("itemStruct")
53
54                 data = await self.TikTokAPPCrawler.fetch_one_video(aweme_id)
55                 # $.imagePost exists if aweme_type is photo
56                 aweme_type = data.get("aweme_type")
57                 # 解析Bilibili视频/Parse Bilibili video
58                 elif "bilibili" in url or "b23.tv" in url:
59                     platform = "bilibili"
60                     aweme_id = await self.get_bilibili_bv_id(url) # BV号作为统一的
video_id
61                     response = await self.BilibiliWebCrawler.fetch_one_video(aweme_id)
62                     data = response.get('data', {}) # 提取data部分
63                     # Bilibili只有视频类型, aweme_type设为0(video)
64                     aweme_type = 0
65                 else:
66                     raise ValueError("hybrid_parsing_single_video: Cannot judge the
video source from the URL.")
67
68                 # 检查是否需要返回最小数据/Check if minimal data is required
69                 if not minimal:
70                     return data
71

```



```

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

```

```

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

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

            api_data = {
                'video_data': {
                    'wm_video_url': video_url,

```

```

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/@flukekgk83/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

```
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,
log_path=None):
53         self.logger.handlers.clear()
54         self.logger.setLevel(level)
55
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="
[%X]"))
65             self.logger.addHandler(ch)
66
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-
%S.log")
71             log_file = self.log_dir.joinpath(log_file_name)
72             fh = TimedRotatingFileHandler(
73                 log_file, when="midnight", interval=1, backupCount=99,
encoding="utf-8"
74             )
75             fh.setFormatter(
76                 logging.Formatter(
77                     "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
78                 )
79             )
80             self.logger.addHandler(fh)
81
82     @staticmethod
83     def ensure_log_dir_exists(log_path: Path):
84         log_path.mkdir(parents=True, exist_ok=True)
85

```

```

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,
log_path=temp_log_dir
126         )
127
128         # 只保留1000个日志文件
129         log_manager.clean_logs(1000)
130
131         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
38      Args:
```



```

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
93 %Y")
94
95         return datetime.datetime.fromtimestamp(float(timestamp)).strftime(format)
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 = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
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-
Cookie string, avoiding incorrect splitting on the value of the 'expires'
field)
129     # 拆分每个Cookie字符串,只获取第一个分段 (即key=value部分) / Split each Cookie
string, only getting the first segment (i.e., key=value part)
130     # 拼接所有的Cookie (Concatenate all cookies)
131     return ";".join(
132         cookie.split(";")[0] for cookie in re.split(", (?=[a-zA-Z])",
cookie_str)
133     )
134
135
136 def split_dict_cookie(cookie_dict: dict) -> str:
137     return "; ".join(f"{key}={value}" for key, value in cookie_dict.items())
138
139
140 def extract_valid_urls(inputs: Union[str, List[str]]) -> Union[str, List[str],
None]:
141     """从输入中提取有效的URL (Extract valid URLs from input)
142
143     Args:
144         inputs (Union[str, list[str]]): 输入的字符串或字符串列表 (Input string or
list of strings)
145
146     Returns:
147         Union[str, list[str]]: 提取出的有效URL或URL列表 (Extracted valid URL or
list of URLs)
148     """
149     url_pattern = re.compile(r"https?:\/\/\S+")
150
151     # 如果输入是单个字符串
152     if isinstance(inputs, str):
153         match = url_pattern.search(inputs)
154         return match.group(0) if match else None
155
156     # 如果输入是字符串列表
157     elif isinstance(inputs, list):
158         valid_urls = []
159
160         for input_str in inputs:
161             matches = url_pattern.findall(input_str)
162             if matches:
163                 valid_urls.extend(matches)
164
165         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     return []
182
183 def get_resource_path(filepath: str):
184     """获取资源文件的路径 (Get the path of the resource file)
185
186     Args:
187         filepath: str: 文件路径 (file path)
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 '.....' 代替。
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 <= "\u9fff") *
3
232     # 计算英文字符长度
233     english_length = sum(1 for char in text if char.isalpha())
234     # 计算下划线数量
235     num_underscores = text.count("_")
236
237     # 计算总长度
238     total_length = chinese_length + english_length + num_underscores
239
240     # 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
241     if total_length > filename_length_limit:
242         split_index = min(total_length, filename_length_limit) // 2 - 6
243         split_text = text[:split_index] + "....." + text[-split_index:]
244         return split_text
245     else:
246         return text
247
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 {0} 接口 JSON 失败:
128 {1}".format(response.url, e))
129             raise APIResponseError("解析JSON数据失败")
130
131     else:
132         if isinstance(response, Response):
133             logger.error(
134                 "获取数据失败。状态码: {0}".format(response.status_code)
135             )
136         else:
137             logger.error("无效响应类型。响应类型: {0}".format(type(response)))
138             raise APIResponseError("获取数据失败")
139
140     async def get_fetch_data(self, url: str):
141         """
142         获取GET端点数据 (Get GET endpoint data)
143
144         Args:
145             url (str): 端点URL (Endpoint URL)
146
147         Returns:
148             response: 响应内容 (Response content)
149         """
150         for attempt in range(self._max_retries):
151             try:
152                 response = await self.aclient.get(url, follow_redirects=True)
153                 if not response.text.strip() or not response.content:
154                     error_message = "第 {0} 次响应内容为空, 状态码: {1}, URL:
155 {2}".format(attempt + 1,
156
157                 response.status_code,
158
159                 response.url)
160
161                 logger.warning(error_message)
162
163                 if attempt == self._max_retries - 1:
164                     raise APIRetryExhaustedError(
165                         "获取端点数据失败, 次数达到上限"
166                     )

```

```

165         await asyncio.sleep(self._timeout)
166         continue
167
168         # logger.info("响应状态码: {0}".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: {0} 代
174         理: {1} 类名: {2}"
175                                     .format(url, self.proxies,
176                                     self.__class__.__name__))
177
178     except httpx.HTTPStatusError as http_error:
179         self.handle_http_status_error(http_error, url, attempt + 1)
180
181     except APIError as e:
182         e.display_error()
183
184     async def post_fetch_data(self, url: str, params: dict = {}, data=None):
185         """
186         获取POST端点数据 (Get POST endpoint data)
187
188         Args:
189             url (str): 端点URL (Endpoint URL)
190             params (dict): POST请求参数 (POST request parameters)
191
192         Returns:
193             response: 响应内容 (Response content)
194         """
195         for attempt in range(self._max_retries):
196             try:
197                 response = await self.aclient.post(
198                     url,
199                     json=None if not params else dict(params),
200                     data=None if not data else data,
201                     follow_redirects=True
202                 )
203                 if not response.text.strip() or not response.content:
204                     error_message = "第 {0} 次响应内容为空, 状态码: {1}, URL:
205                     {2}".format(attempt + 1,
206

```

```

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: {0}".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: {0} 代理: {1} 类名:
224 {2}".format(url, self.proxies,
225 self.__class__.__name__)
226         )
227
228     except httpx.HTTPStatusError as http_error:
229         self.handle_http_status_error(http_error, url, attempt + 1)
230
231     except APIError as e:
232         e.display_error()
233
234     async def head_fetch_data(self, url: str):
235         """
236         获取HEAD端点数据 (Get HEAD endpoint data)
237
238         Args:
239             url (str): 端点URL (Endpoint URL)
240
241         Returns:
242             response: 响应内容 (Response content)
243         """
244         try:
245             response = await self.aclient.head(url)
246             # logger.info("响应状态码: {0}".format(response.status_code))
247             response.raise_for_status()
248             return response
249
250         except httpx.RequestError:
251             raise APIConnectionError("连接端点失败, 检查网络环境或代理: {0} 代理:
252 {1} 类名: {2}".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
270     Raises:
271         APIConnectionError: 连接端点失败 (Failed to connect to endpoint)
272         APIResponseError: 响应错误 (Response error)
273         APIUnavailableError: 服务不可用 (Service unavailable)
274         APINotFoundError: 端点不存在 (Endpoint does not exist)
275         APITimeoutError: 连接超时 (Connection timeout)
276         APIUnauthorizedError: 未授权 (Unauthorized)
277         APIRateLimitError: 请求频率过高 (Request frequency is too high)
278         APIRetryExhaustedError: 重试次数达到上限 (The number of retries has
279 reached the upper limit)
280     """
281     response = getattr(http_error, "response", None)
282     status_code = getattr(response, "status_code", None)
283
284     if response is None or status_code is None:
285         logger.error("HTTP状态错误: {0}, URL: {1}, 尝试次数: {2}".format(
286             http_error, url, attempt
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
端口
84     - KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1 # 单机环境副本数设为1
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:
```