一、PSP表格

(2.1)在开始实现程序之前，在附录提供的PSP表格记录下你估计将在程序的各个模块的开发上耗费的时间。（3'）

(2.2)在你实现完程序之后，在附录提供的PSP表格记录下你在程序的各个模块上实际花费的时间。（3'）

PSP2.1	Personal Software Process Stages	预估耗时（分钟）	实际耗时（分钟）
Planning	计划	30	40
· Estimate	· 估计这个任务需要多少时间	30	40
Development	开发	600	800
· Analysis	· 需求分析 (包括学习新技术)	90	120
· Design Spec	· 生成设计文档	30	40
· Design Review	· 设计复审	15	20
· Coding Standard	· 代码规范 (为目前的开发制定合适的规范)	30	30
· Design	· 具体设计	60	80
· Coding	· 具体编码	240	300
· Code Review	· 代码复审	60	120
· Test	· 测试（自我测试，修改代码，提交修改）	75	90
Reporting	报告	80	100
· Test Repor	· 测试报告	20	40
· Size Measurement	· 计算工作量	30	30
· Postmortem & Process Improvement Plan	· 事后总结, 并提出过程改进计划	30	30
	· 合计	710	940

二、任务要求的实现

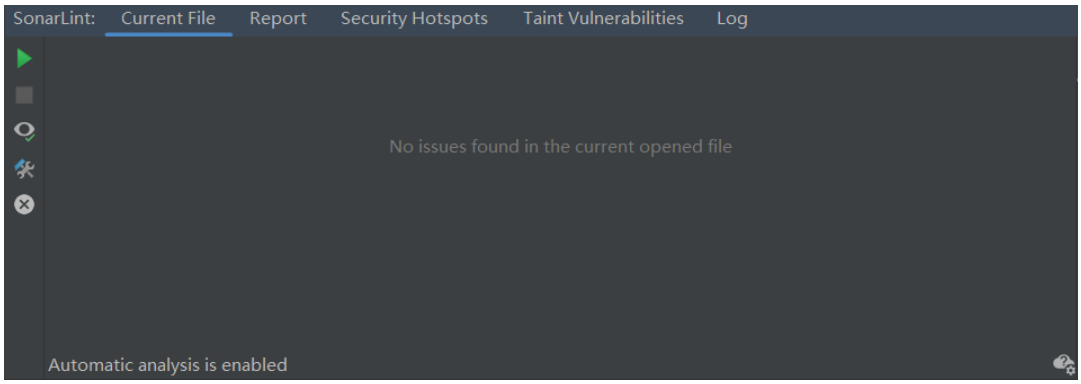
(3.1)项目设计与技术栈。从阅读完题目到完成作业，这一次的任务被你拆分成了几个环节？你分别通过什么渠道、使用什么方式方法完成了各个环节？列出你完成本次任务所使用的技术栈。（5'）

- **计划**

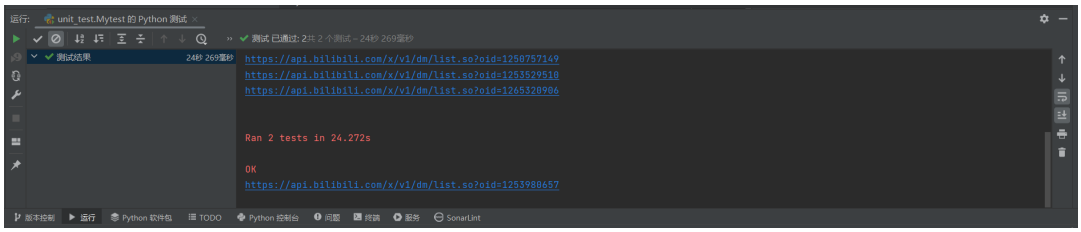
  我首先对任务进行了整体规划，按照 PSP（Personal Software Process）表格的思路，将任务分解为以下几个环节：

1. **需求分析**：理解任务需求，明确需要从指定日期范围内的 B 站视频中获取弹幕数据，并进行频次统计。
2. **技术调研**：查找相关的 B 站 API，以及学习如何使用 Python 的相关库来实现功能。
3. **编码实现**：编写代码实现各个功能模块，包括获取 bvid、cid，抓取弹幕，数据存储和统计。
4. **测试调试**：对代码进行测试，解决出现的异常和错误，确保程序稳定运行，同时对代码进行性能上的优化改进。
5. **结果输出**：将最终的弹幕数据和统计结果保存到 Excel 文件中，并绘制词云图等展示结果。

- **开发**
  - **样本验证**
    通过一个视频的弹幕爬虫实践，验证计划步骤的可行性。
  - **获取 bvid 和 cid**
    通过对 B 站网页和 API 的分析，了解了如何使用搜索接口获取视频列表，并从中提取每个视频的 `bvid`，使用关键字和分页参数，构造 B 站的搜索 API 请求 URL。发送请求后，解析返回的 JSON 数据结构，从中提取每个视频的 `bvid`。使用获取到的 `bvid`，构造另一个 API 请求，获取视频的 `cid`，这是获取弹幕所需的参数。
    主要技术栈：浏览器开发者模式的使用，浏览器json文件分析，Python正则表达式，Python文档写入
  - **编程实现获取弹幕**
    在获取到视频的 `cid` 后，使用弹幕接口来获取弹幕数据，并对数据进行处理和存储：构造弹幕 API 的请求 URL，发送请求获取弹幕的 XML 数据，使用正则表达式从 XML 数据中提取弹幕文本内容，使用 `ThreadPoolExecutor` 创建线程池，提升数据抓取的效率，将获取的弹幕数据保存到文本文件和 Excel 文件中，方便后续的分析和处理，
    主要技术栈：`requests`、`re`、`openpyxl`、`pandas`、`collections`，`concurrent.futures.ThreadPoolExecutor`
  - **弹幕出现频次统计**
    读取保存的弹幕数据，使用 `collections.Counter` 对弹幕进行词频统计，并将结果保存到新的 Excel 文件中。通过Python字典和sorted的函数降序排列弹幕文本以及出现的频次，通过openpyxl库的函数写入Excel中，并输出出现频次前8的弹幕
    主要技术栈：Python读文档，Python字典，Python写入Excel，Python sorted( )函数
  - **词云图绘制**
    通过wordcloud，jieba，imageio等库的函数完成对所有弹幕内容的文本分割，图片识别，词云图设置及生成。
    主要技术栈：wordcloud，jieba，imageio库相关函数的使用，Adobe Photoshop的使用
- **性能分析与改进**
  - 采用异常抛出处理
  - 利用pycharm的插件SonarLint进行Code Quality Analysis，参照官方检测规则及修复示例消除了所有警告，如图：



- 利用python自带的unittest进行单元测试，顺利通过单元测试



- 使用cProfile对各函数执行次数和执行时间，在此基础上对脚本性能进行改进
- 利用**减少API调用频率**，**HTTP连接重用**，**异步IO**，**缓存处理**等优化代码

(3.2)爬虫与数据处理。说明业务逻辑，简述代码的设计过程（例如可介绍有几个类，几个函数，他们之间的关系），并对关键的函数或算法进行说明。（20'）

业务逻辑图

- 先从b站API中获取视频的bvid号。先加上headers头部信息，伪装来源，减少被拦截的可能性，然后通过HTTP的get方法获取json文件，再用python内置函数将json文件转换成字典格式。其数据结构可以从html文件中看出，由此可以得到视频的bvid号。
  其中用到request.get()函数和json.loads()函数，json.loads()函数可以把requests.get()函数得到的json文件转为字典格式

```
# 定义函数，获取搜索结果中的bvid（视频的唯一标识符）
def get_bvid(page_number, number):
    # 构造搜索API的URL，page_number是页码，number是该页中的视频编号
    url = f'https://api.bilibili.com/x/web-interface/search/type?page=
{page_number}&page_size=50&keyword=2024%E5%B7%B4%E9%BB%8E%E5%A5%A5%E8%BF%90%E
4%BC%9A&search_type=video'
    response = requests.get(url=url, headers=headers)  # 发送请求
    try:
        # 解析返回的JSON数据，提取视频的bvid
        json_data = json.loads(response.text)
        print(json_data)
        bvid = json_data['data']['result'][number]['bvid']
        print(f"获取到bvid: {bvid}")
        return bvid  # 返回bvid
    except (KeyError, IndexError, json.JSONDecodeError,
requests.RequestException) as e:
        print(f"获取bvid时出错: {e}")
        # 捕获错误并返回None，防止程序崩溃
        return None
```

- 得到视频的bvid后，在根据b站的另一个API获取指定视频的cid号，同样利用get方法得到json文件，后转成字典，再其根据数据结构得到cid号

其中同样用到request.get()函数和json.loads()函数，json.loads()函数可以把requests.get()函数得到的json文件转为字典格式

```
# 定义函数，根据bvid获取视频的cid（弹幕对应的唯一标识符）
def get_cid(bvid):
    try:
        # 通过bvid构造获取cid的API请求URL
        url = f'https://api.bilibili.com/x/player/pagelist?bvid=
{bvid}&jsonp=jsonp'
        response = requests.get(url, headers=headers)  # 发送请求
        if response.status_code != 200:
            # 如果请求状态码不是200，返回None
            return None
        # 解析返回的JSON数据，提取cid
        json_dict = json.loads(response.text)
        return json_dict['data'][0]['cid']  # 返回cid
    except (KeyError, IndexError, json.JSONDecodeError,
requests.RequestException):
        return None  # 捕获错误并返回None
```

- 有了视频的cid号之后就可以进行视频弹幕的爬取了，利用字符串拼接，得到300个视频的弹幕地址，使用ThreadPoolExecutor创建线程池，用于并发请求，并使用as_completed（）先加快处理速度
  其中用到了ThreadPoolExecutor（），as_completed（）等函数

```
# 定义函数，批量获取bvid和cid，并创建并发任务
def put_api():
    tasks = []
    # 使用ThreadPoolExecutor创建线程池，用于并发请求
    with ThreadPoolExecutor(max_workers=10) as executor:
        # 控制页码范围（1到5页），每页50个视频
        for i in range(1, 7):
            for j in range(50):
                bvid = get_bvid(i, j)  # 获取bvid
                if bvid:
                    cid = get_cid(bvid)  # 获取cid
                    if cid:
                        # 提交弹幕抓取任务到线程池

 tasks.append(executor.submit(fetch_and_save_bulletchat, cid))
    return tasks  # 返回任务列表

# 定义函数，处理并发任务，收集所有弹幕数据
def get_data(tasks):
    all_bulletchats = []
    # 遍历所有完成的任务，获取结果
    for task in as_completed(tasks):
        bulletchat_data = task.result()
        if bulletchat_data:
            all_bulletchats.extend(bulletchat_data)  # 将弹幕数据加入总列表
    return all_bulletchats  # 返回所有弹幕数据
```

- 爬取弹幕完成后，就可以利用Excel中的数据进行词频统计了，这里关键在于利用collections中的Counter方法，统计出列表中各个弹幕的出现次数，将其存储在字典当中，再利用sorted方法以value进行排序，即得到出现次数前20的弹幕，将其输出到控制台和Excel中

```
# 定义函数，计算弹幕频次，并保存到Excel
def calculate_frequency():
    try:
        # 读取弹幕Excel文件
        fd = pd.read_excel(file_xlsx)
        lines = fd['弹幕']
        # 将所有弹幕拼接成一个字符串
        text = ' '.join(lines.astype(str))
        words = text.split()  # 将弹幕分割为单词
        word_counts = collections.Counter(words)  # 统计单词频次
        sorted_word_counts = sorted(word_counts.items(), key=lambda x: x[1],
reverse=True)  # 按频次排序

        # 创建新的Excel工作簿用于保存频次统计结果
        workbook = openpyxl.Workbook()
        sheet = workbook.active
        sheet.append(['弹幕', '频次'])  # 添加标题行

        # 将排序后的词频结果写入Excel
        for word, count in sorted_word_counts:
            sheet.append([word, count])

        workbook.save('我的统计弹幕出现次数.xlsx')  # 保存频次统计的Excel文件
    except Exception as e:
        print(f"计算频次时出错: {e}")
```

- 词云图和饼图绘制，饼图由上一步得到的Excel表格按公式生成

  词云图主要使用了jieba库分词，imageio库图像识别，stopwords集合的停止词设置，WordCloud()的词云图设置

```
import os
from os import path
from wordcloud import WordCloud
import jieba
from imageio.v3 import imread

def Bulletchat_Wordcloud():
    try:
        #获取当前文件路径
        d = path.dirname(__file__) if "__file__" in locals() else os.getcwd()
        text = open(path.join(d,r'提取.txt'), 'rb').read()
        # 设置模板图
        img_mask = imread(r'Paris.png')
        # 对弹幕进行精确模式分词
        text_list = jieba.lcut(text,cut_all=False)
        text_str = ' '.join(text_list)
        # print(text_str)
        # 设置中文字体
        font_path = 'msyh.ttc'
        # 停止词设置
```

```
        stopwords = set('')
        stopwords.update(['的','和','又','了','都','是','什么','所
以','这','呢','吧','吗','个','呀','嘛','哈'])
        wc = WordCloud(
            font_path=font_path,
            #max_words=500,  # 最多词个数
            #min_font_size=15,
            #max_font_size=100,
            mask=img_mask,
            background_color='white',
            stopwords=stopwords,
            colormap='copper'
        )
        wc.generate(text_str)
        wc.to_file(r'outParis.png')
    except Exception as e:
        print(f"词云图生成异常: {e}")

if __name__ == '__main__':
    Bulletchat_Wordcloud()
```

- 用cProfile库分析程序中步骤执行的时间和次数，并对性能加以改进

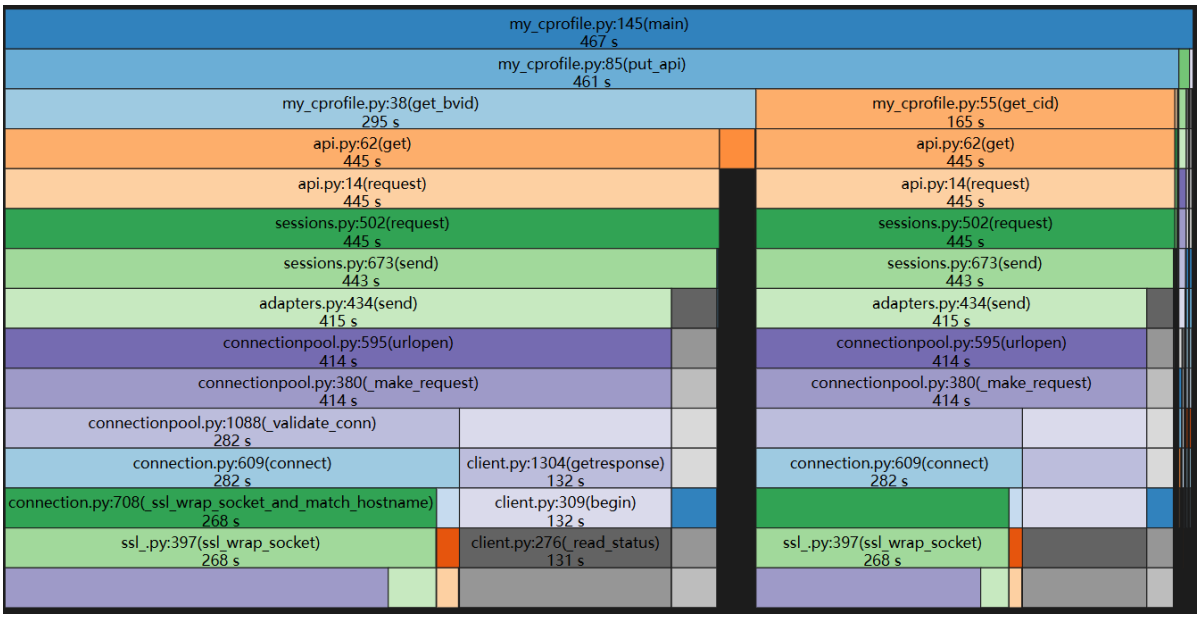
```
import cProfile  # 用于性能分析
# 创建性能分析器实例，并开始性能分析
profile = cProfile.Profile()
profile.enable()

# 停止性能分析
profile.disable()
# 将性能分析数据保存到文件中
profile.dump_stats('./youhua.prof')
```

(3.3)数据统计接口部分的性能改进。记录在数据统计接口的性能上所花费的时间，描述你改进的思路，并展示一张性能分析图（例如可通过VS /JProfiler的性能分析工具自动生成），并展示你程序中消耗最大的函数。（6'）

优化前的性能数据具体分析：

- `get_bvid` **函数**：
  - 耗时 **295秒**。
  - 其中，API请求相关函数如 `api.py:62(get)` 和 `api.py:14(request)` 各自花费 **445秒**。
  - 网络请求的实际发送（`sessions.py:673(send)`）花费 **443秒**，这说明几乎所有的时间都耗费在网络请求上。
- `get_cid` **函数**：
  - 耗时 **165秒**，相比 `get_bvid` 要少一些，原因可能是 `cid` 数据更简单或者部分请求被缓存了。
  - 其他网络IO的函数，如 `connectionpool.py:609(connect)` 和 `ssl_wrap_socket` 等花费的时间分别为 **282秒** 和 **268秒**，进一步证明网络请求是整个程序的性能瓶颈。
- **总体时间**：

- 主程序（`my_cprofile.py:145(main)`）耗时 **467秒**，其中 `put_api` 操作耗时 **461秒**，几乎占据了程序的整个执行时间。
- 因为网络IO开销极大，导致程序在大量时间里处于等待网络响应的状态。



由此可见瓶颈主要集中在以下几个函数调用中：

1. **requests.get** 调用花费了相当长的时间，特别是在发送HTTP请求的部分（`connectionpool.py`，`connection.py`，`ssl.py`）。
2. 由于涉及大量网络请求，API请求的频率和总数是性能的主要瓶颈，其中耗时最多的函数为：`get_bvid()`，其次为`get_cid()`，

我考虑下述4种优化：

1. 减少API调用频率

- **并发请求的优化**：虽然我已经使用了 `ThreadPoolExecutor` 进行并发处理，但在每个 `get_bvid()` 和 `get_cid()` 请求之间，可能有一些等待时间。可以尝试通过减少重复请求以及批量化 API 请求来优化。

```
# 异步函数：处理并发任务，收集所有弹幕数据
async def fetch_all_bulletchats(session):
    all_bulletchats = []  # 用于存储所有的弹幕数据
    tasks = []  # 用于存储所有的异步任务

    total_requests = 6 * 50  # 总共请求 6 页，每页 50 个视频，共 300 个视频
    for i in range(total_requests):
        page_number = i // 50 + 1  # 计算当前请求的页码
        index = i % 50  # 计算当前页内的索引
        # 创建异步任务，获取每个视频的弹幕数据
        tasks.append(asyncio.ensure_future(fetch_bulletchat_data(session,
page_number, index)))

    # 使用 asyncio.as_completed 来迭代已完成的任务
    for task in asyncio.as_completed(tasks):
        bulletchat_data = await task
        if bulletchat_data:
            # 将获取的弹幕数据添加到总列表中
            all_bulletchats.extend(bulletchat_data)
```

```
    return all_bulletchats  # 返回所有的弹幕数据
```

2. HTTP连接重用

- **优化 HTTP 连接**：我通过 `requests.Session()` 来重用 HTTP 连接。现在每次调用 `requests.get()` 都会建立新的连接，使用 `Session()` 可以减少连接的建立和关闭时间。

```
# 异步函数：获取并保存某个视频的弹幕
async def fetch_and_save_bulletchat(session, cid):
    # 使用 cid 构造弹幕 API 的 URL
    url = tempApi.replace("{number}", str(cid))
    try:
        # 发送异步 GET 请求
        async with session.get(url) as response:
            # 获取响应的文本内容（XML 格式）
            response_text = await response.text()
            # 使用正则表达式提取所有弹幕内容
            data = re.findall('<d p=".*?">(.*?)</d>', response_text)
            # 如果有弹幕数据，返回列表，否则返回空列表
            return data if data else []
    except:
        # 如果出现异常，返回空列表
        return []
```

3. 异步IO

- **异步请求**：使用 `asyncio` 和 `aiohttp` 替换 `requests` 模块。`aiohttp` 可以实现真正的异步 I/O，在大量网络请求中将会显著提升性能。

```
# 异步函数：获取 bvid，带缓存功能
async def get_bvid(session, page, index):
    # 如果已经在缓存中，则直接返回缓存的 bvid
    if (page, index) in bvid_cache:
        return bvid_cache[(page, index)]

    # 构造 API 请求的 URL，查询指定页码和关键字的视频
    url = f'https://api.bilibili.com/x/web-interface/search/type?page=
{page}&page_size=50&keyword=2024%E5%B7%B4%E9%BB%8E%E5%A5%A5%E8%BF%90%E4%BC%9A
&search_type=video'

    # 发送异步 GET 请求
    async with session.get(url) as response:
        try:
            # 尝试将响应内容解析为 JSON 格式
            json_data = await response.json()
            # 提取第 index 个视频的 bvid
            bvid = json_data['data']['result'][index]['bvid']
            # 将 bvid 存入缓存
            bvid_cache[(page, index)] = bvid
            return bvid
        except (KeyError, IndexError, json.JSONDecodeError) as e:
            # 如果出现异常，打印错误信息和响应内容，返回 None
            print(f"获取 bvid 时出错: {e}")
```

```
            print(f"响应状态码: {response.status}")
            text = await response.text()
            print(f"响应内容: {text}")
            return None

# 异步函数：获取 cid，带缓存功能
async def get_cid(session, bvid):
    # 如果 bvid 已经在缓存中，则直接返回缓存的 cid
    if bvid in cid_cache:
        return cid_cache[bvid]

    # 构造 API 请求的 URL，查询指定 bvid 的视频信息
    url = f'https://api.bilibili.com/x/player/pagelist?bvid=
{bvid}&jsonp=jsonp'

    # 发送异步 GET 请求
    async with session.get(url) as response:
        try:
            # 尝试将响应内容解析为 JSON 格式
            json_dict = await response.json()
            # 提取第一个视频的 cid
            cid = json_dict['data'][0]['cid']
            # 将 cid 存入缓存
            cid_cache[bvid] = cid
            return cid
        except (KeyError, IndexError, json.JSONDecodeError):
            # 如果出现异常，返回 None
            return None
```

4. 缓存处理

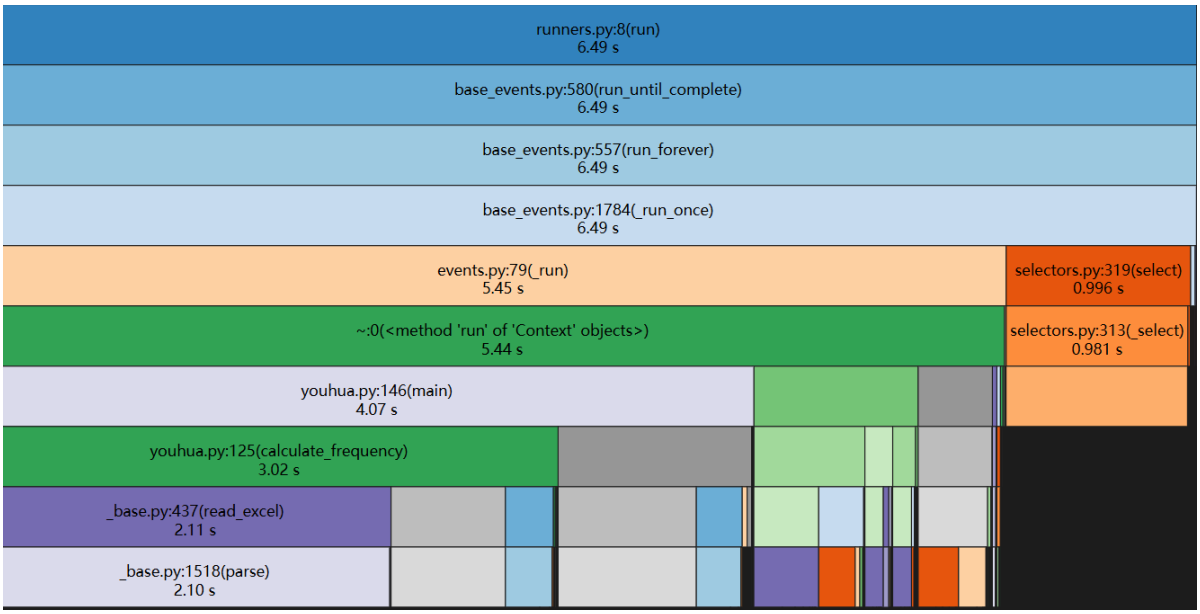
- **本地缓存数据**：我在获取 bvid 和 cid 的数据时引入缓存机制，避免重复请求，将数据写入本地，在下一次请求时复用缓存数据。

```
# 全局缓存，用于存储 bvid 和 cid，避免重复请求
bvid_cache = {}
cid_cache = {}
```

优化后的性能数据具体分析：

- `youhua.py:146(main)` **函数**：
  - 优化后的主程序耗时 **4.07秒**，相比优化前的 **467秒**，这是一个巨大的性能提升，缩短了将近 **463秒**，表明通过缓存和异步IO机制，API调用被显著减少。
- `calculate_frequency` **函数**：
  - 耗时 **3.02秒**，该部分为数据处理的核心部分，相比优化前，计算的时间比API调用要占比更大，表明大部分时间花费在了有意义的计算任务上。
- **文件读取**：
  - `read_excel` 和 `parse` 各自花费 **2.11秒** 和 **2.10秒**，这些是文件读取和解析的时间。相比于优化前将近 **300秒** 的 `get_bvid` 时间，这些文件IO操作的耗时较短且合理。
- **网络等待时间减少**：

- `selectors.py:319(select)` 和 `selectors.py:313(_select)` 分别只花费 **0.996秒** 和 **0.981秒**。相比优化前，网络IO的时间开销显著减少，这说明网络请求已经大幅减少。



由此可见**优化对性能产生显著影响**：

(3.4)数据结论的可靠性。介绍结论的内容，以及通过什么数据以及何种判断方式得出此结论（6'）

结论：

1. **生成/合成技术主导讨论**：AI生成和合成技术（如AI生成图像或内容）是观众关注的焦点，占据了绝大多数讨论。这可能是因为此类技术能够直接影响用户在观看奥运会内容时的视觉体验，如AI生成的体育场景、运动员的虚拟形象等。
2. **其他AI应用较少关注**：音效、配音、视频修复等方面的AI技术相对没有引起广泛讨论，可能是因为它们在奥运会视频内容的应用场景不如生成/合成技术显著。

数据：

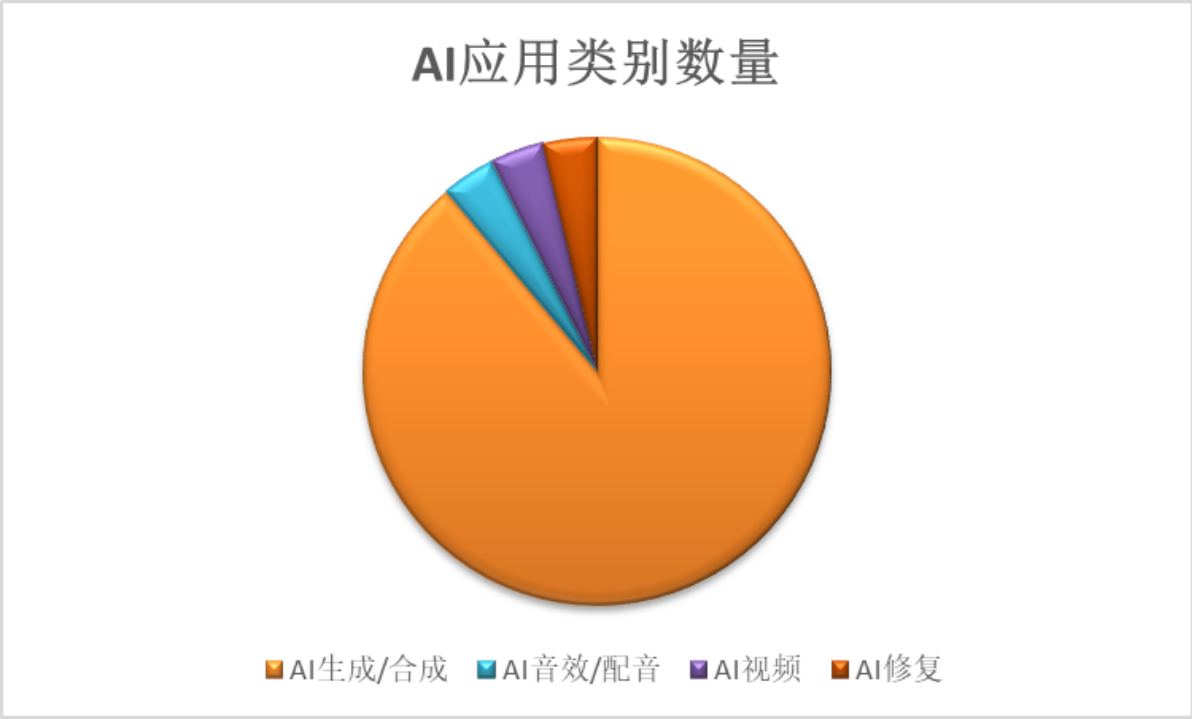
AI应用类别	数量
AI生成/合成	72
AI音效/配音	3
AI视频	3
AI修复	3

判断依据：

- **弹幕数量**：通过弹幕出现次数来判断哪些AI技术最受关注。生成/合成类弹幕数量显著高于其他类别（72次 vs. 3次），这一数值差异清晰地展示了不同类别的关注度。
- **弹幕内容**：一些弹幕提到了AI生成内容的逼真性，以及人们对AI生成画面是否影响真实观看体验的讨论。这种定性分析表明，观众对生成/合成技术的情感反应更为强烈。

(3.5)数据可视化界面的展示。在博客中介绍数据可视化界面的组件和设计的思路。（15'）

饼图：直接通过Excel绘制



下图是通过WordCloud库生成的词云图（GUASS原图是什么~）

组件：程序主要由以下组件构成：使用了 `os` 和 `os.path` 模块来处理文件路径，确保程序可以正确定位输入和输出文件；`wordcloud` 库用于生成词云图；`jieba` 库用于中文分词，能够精确地将中文文本切分成词语；`imageio.v3` 的 `imread` 函数用于读取作为词云形状的模板图片；程序还指定了字体路径（如 `msyh.ttc`）来支持中文字符显示，并设置了停止词以过滤常见的、不具备分析价值的词汇。输入的文本数据和模板图像分别从指定的文件路径读取，最终生成的词云图保存到指定的输出路径。

设计思路：程序的设计目的是从包含弹幕或其他文本的文件中生成一个定制化的词云图。首先，读取文本文件和模板图片，然后使用 `jieba` 库对中文文本进行精确模式分词，得到词语列表。将词语列表拼接成适合词云生成的字符串后，利用 `WordCloud` 对象设置词云的各项参数，如字体、最大词数、模板形状、背景颜色、停止词和配色方案等。通过调用 `generate` 方法生成词云，并使用 `to_file` 方法将结果保存为图片文件。程序还包含异常处理机制，捕获并输出在生成词云过程中可能出现的错误，确保程序的稳健性和用户的知情权。

接下来展示我做的代码雨，爬虫要有始有终，既然爬了弹幕，就要以弹幕的形式呈现结果。于是我就选择了这个代码雨程序，迫于时间有限没能做的很美观。

组件：程序主要使用了 Python 的标准库和第三方库，包括 `random` 模块用于生成随机数，`Pillow` 库中的 `Image`、`ImageDraw`、`ImageFont` 模块用于创建和绘制图像，`imageio` 库用于将生成的图像帧合成为 GIF 动画。此外，程序需要一个支持中文的字体文件（如 `msyh.ttc`）和包含中英文字幕的文本文件 `subtitles.txt` 作为资源。

设计思路：程序首先从字幕文件中读取字幕内容，提取所有出现的字符来构建代码雨的字符集，确保代码雨中包含字幕中的中英文字符。然后，初始化图像尺寸、字体大小、代码雨列数等参数，随机设置每一列代码雨的起始位置。在每一帧中，绘制代码雨效果，字符从上至下随机下落，颜色由亮变暗模拟雨滴的视觉效果，同时在图像底部显示当前的字幕。通过循环更新代码雨的位置，生成连续的帧，最后将所有帧合成为一个动画 GIF，呈现出动态的代码雨与字幕同步显示的效果。

(3.6)附加题展示。

爬取世界主流媒体的观点，预测事件走向

原先是想爬取YouTube相关视频弹幕以此获取世界主流媒体看法，后来遇到了种种困难，时间也不太充裕了，所有就在国内的新闻媒体上爬取了一些国外媒体的相关文字报道，经过许多无效文字的筛选，结果如下所示：

英国广播公司（BBC）:顶尖运动员面临着骇人听闻的网络辱骂。今年，巴黎奥运会正试图保护他们免受这种辱骂,人工智能算法正在从社交媒体用户发布的有关奥运会的海量内容中筛选出唯一的使命：消除网络辱骂。

路透社（Reuters）:基于人工智能的攻击也有可能扰乱奥运会，“无论是售票、欺诈，还是操纵直播”。国际拳击协会（AIBA）使用人工智能技术审查贝尔格莱德世界拳赛的裁判和裁判员,麦克拉伦将其描述为“消除比赛操纵的一大历史性进步”，并表示该技术可以作为其他奥林匹克运动项目使用裁判和裁判员的蓝图。

华盛顿邮报（The Washington Post）:你相信吗？人工智能会模仿声音录制奥运会片段

法国世界报（Le Monde）：[DJ 芭芭拉·布奇 (Barbara Butch) 在参加奥运会开幕式后提出网络骚扰投诉，需要通过技术手段（如AI）来监控和防范。

中国日报：随着中国国家跳水队在巴黎奥运会上继续淘金，其人工智能训练系统正在体育创新领域引起轰动。传统的视频录制无法捕捉快速序列，并且后续的数据分析既耗时又不及时。基于百度类似 ChatGPT 的产品和大型语言模型 Ernie Bot，该人工智能系统通过提供清晰、准确、全面的见解，推进数据量化和分析潜水来解决这些挑战。

观点分析：

各大媒体对2024年巴黎奥运会上人工智能的应用提出了不同的看法。BBC强调了顶尖运动员受到网络辱骂的威胁，奥运会正利用人工智能算法筛选并消除这些不良内容，保护运动员的心理健康。路透社警示了人工智能可能被用于攻击奥运会的风险，如售票欺诈和直播操纵。AIBA通过AI技术审查裁判，提升比赛的公平性，被视为消除比赛操纵的历史性进步。华盛顿邮报提到人工智能可以模仿声音录制奥运片段，暗示了AI在媒体内容生成方面的应用和潜在问题。法国世界报关注到参与者因网络骚扰提出投诉，进一步凸显了网络安全的重要性。中国日报则报道了中国跳水队利用AI训练系统取得优异成绩，展示了AI在体育训练和数据分析中的创新应用。

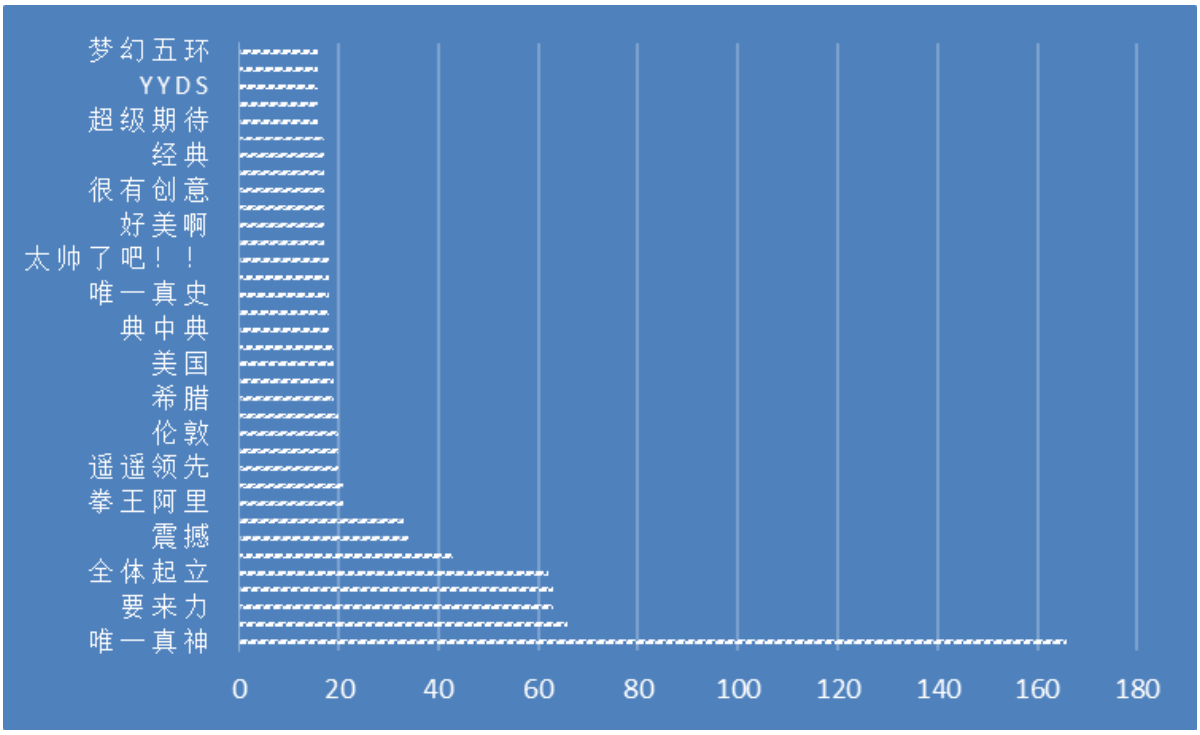
事件走向：

人工智能预计将在2024年巴黎奥运会上发挥关键作用，带来机遇与挑战并存的局面。一方面，AI技术将广泛应用于赛事管理、运动员保护和训练提升，推动奥运会的顺利和高水平进行。另一方面，需要警惕AI可能被滥用于网络攻击、欺诈和虚假信息传播。为了最大化地发挥AI的正面作用，奥运会组织者和各相关方可能会加强对AI技术的投入与监管，通过国际合作制定相关规范，确保赛事的公平、公正和安全。

自主发挥：爬取有趣的数据进行分析、制作数据可视化大屏等，有创意有乐趣即可。

我爬取了b站关键词为“2028年洛杉矶奥运会”的前300个视频中的弹幕，并做了词云分析和数据可视化，如下图：

我
就是
你
啊
就
这个
也
不是
奥运会
中国
美国
我们
真的
有
没有
年
说
在
好
来
印度
现在
看
不
人
自己
五环 展示
唯一 真神
洛圣
大
给
得
很多
要
地方
金牌
bgm
这里
最 喜欢
北京
有 创意
这样
韩国
还
一样
奥运
意大利
故乡
我 觉得
全体 起立
看到
还有
那个
雅典
还是
音乐
首都
好看
确实
前方 高能
日本
国家
被
震撼
时候
当时
到
跟
北京
开幕式
应该
哈哈哈 哈哈
城市
因为
伦敦 奥运会
知道
必须
不行
打
运动员
已经
挺 好
是不是
他们
对
其实
你们
火炬
经典
太美
洛杉矶
巴西
最好
上
法国
巴黎
奥运会 开幕式 梦幻 五环
比赛
希腊
想
开始
我 在 谁
项目
别人
怎么
很 好
伦敦
好像
为什么
取消
五环
英国
哈哈
笑 死
工业革命
可以
只有
不会
能
前面
这个 火炬
冬奥会
感觉
做
doge
历史
一个
这么
文化
很 有
现场
没
好家伙
举办
强
像
哈哈哈哈 哈哈哈
哈哈哈 哈哈哈 哈哈哈哈
但是



有趣的发现

- **对经典元素的致敬**：如 **“工业革命”**、**“拳王阿里”**、**“慕尼黑惨案”**，观众对节目中历史元素的呈现反应热烈。
- **游戏与现实的交融**：**“洛圣都”**、**“GTA”** 的出现，体现了年轻观众将游戏文化与现实场景联系的思维方式。
- **网络流行语的运用**：**“yyds”**、**“要来力”** 等，展现了弹幕文化的活力和创意。

高频词语解读

- 唯一真神（166 次）
  - 可能指代某个在开幕式或宣传片中表现突出的元素或人物，引发观众的热议和赞叹。
- 工业革命（66 次）
  - 可能与开幕式中涉及工业革命主题的表演相关，激起了观众的共鸣。
- 要来力（63 次）
  - 这是 **“Eureka”**（尤里卡）的音译或谐音，可能在节目中出现，引发观众模仿发送。
- 洛圣都（63 次）
  - **“洛圣都”** 是游戏《侠盗猎车手 V》（GTAV）中的虚构城市，原型是洛杉矶。观众可能因场景或氛围联想到游戏。
- 全体起立（62 次）
  - 表达对某个表演或人物的尊敬和赞美。

情感倾向分析

- **正面情感**：
  - **“太美了”**、**“震撼”**、**“好看”**、**“漂亮”**、**“帅”**、**“经典”** 等，体现了观众对节目内容的喜爱和赞赏。
- **致敬与怀念**：

        - **“拳王阿里”**、**“慕尼黑惨案”**、**“致敬”**，可能节目中有致敬历史人物或事件的环节。
    - **期待与兴奋**：
        - **“超级期待”**、**“历史最佳”**、**“梦幻五环”**，观众对奥运会的期待和对表演的高度评价。

三、心得体会

(4.1)在这儿写下你完成本次作业的心得体会，当然，如果你还有想表达的东西但在上面两个板块没有体现，也可以写在这儿~（10'）

- 在本次作业真正入手实践之间，在会运用爬虫技术爬取百度上的图片，并不会甚至没听说过可以爬取b站弹幕。在动手实践时也是遇到了许多困难，在爬虫代码的实现上所费时不多，更多的是在爬取过多次，被封后的等待时间以及在性能分析那块费时许久。git的使用也是摸索了好久，遭遇种种错误之后终于调通了。
- 总的来说，这次作业让我学习到了许多之前没了解过的领域，如Code Quality Analysis、多线程并发执行、异步爬虫等等，都是之前从未探索过的领域。同时更加熟悉了python的文件读取操作、matplotlib库进行绘图以及爬虫技术的拓展。最重要的是，锻炼了独立面对一个繁琐项目的能力，能够在有限的时间中完成各项任务，也加强了面对异常情况的查阅资料处理能力，对我来说这是一次启发性的项目体验，能够让我在今后的学习生活中更好的独立完成任务。
- 在最后的多线程优化环节折腾了一天多的时间，又是在deadline前爆赶，在以后面对比较繁琐的任务时，应该提前做好规划，分成小块，逐步完成。下次一定！