# 1. 粉丝缓存类实现

# 2. Flask 日志

## 2.1 logging 基本使用

```python
import logging


def basic_log():
    format = "%(name)s %(levelname)s %(pathname)s %(lineno)d %(message)s"
    # logging.basicConfig(level=logging.DEBUG, format=format)
    # 日志写入文件中
    logging.basicConfig(level=logging.DEBUG, format=format, filename='测试日志.log')

    # 打印日志
    logging.debug('这是 debug 消息')
    logging.info('这是 info 消息')
    logging.warning('这是 warning 消息')
    logging.error('这是 error 消息')


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

如果传递了 filename ，那么日志会写入文件，此时，终端不在打印日志

## 2.2 自定义日志和日志处理器

```python
# 创建/获取自定义日志
my_logger = logging.getLogger('custom_log')
# 设置日志级别
my_logger.setLevel(logging.DEBUG)

# - 创建输出处理器
# - 日志器添加输出处理器
# 输出到终端
#  实例化终端日志处理器
stream_handler = logging.StreamHandler()
#  实例日志格式
formatter = logging.Formatter(fmt='%(name)s %(levelname)s %(pathname)s %(lineno)d %(message)s')
#  绑定终端日志处理器和日志格式
stream_handler.setFormatter(formatter)
#  终端日志处理器 添加到自定义日志中
my_logger.addHandler(stream_handler)

# 输出到文件
file_handler = logging.FileHandler(filename='my_logger.log')
file_handler.setFormatter(formatter)
my_logger.addHandler(file_handler)

my_logger.debug('自定义 debug 消息')
my_logger.info('自定义 info 消息')
my_logger.warning('自定义 warning 消息')
my_logger.error('自定义 error 消息')
```

## 2.3 SMTP 日志处理器

```python
MAIL_HOST = 'smtp.qq.com'
MAIL_PORT = 587
MAIL_SENDER = '772775481@qq.com'
MAIL_USER = '772775481@qq.com'
MAIL_CODE = 'hqexhbhjqylebfaa'

smtp_logger = logging.getLogger('smtp_logger')

mail_handler = SMTPHandler(
    mailhost=(MAIL_HOST, MAIL_PORT),  # SMTP 服务器地址和端口号
    fromaddr=MAIL_SENDER,  # 发件人地址
    toaddrs=['772775481@qq.com', ],  # 收件人地址
    subject='Log Error',
    credentials=(MAIL_USER, MAIL_CODE)  # 发件人的邮件地址 和 发件人的授权密码
)
formatter = logging.Formatter(fmt='%(name)s %(levelname)s %(pathname)s %(lineno)d %(message)s')

mail_handler.setFormatter(formatter)
#  mail_handler 只处理 ERROR 级别以上的日志
mail_handler.setLevel(logging.ERROR)

# 日志打印到终端
stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)
stream_handler.setLevel(logging.INFO)
```

## 2.4 Flask 日志

```python
smtp_logger = logging.getLogger('smtp_logger')

mail_handler = SMTPHandler(
    mailhost=(MAIL_HOST, MAIL_PORT),  # SMTP 服务器地址和端口号
    fromaddr=MAIL_SENDER,  # 发件人地址
    toaddrs=['772775481@qq.com', ],  # 收件人地址
    subject='Log Error',
    credentials=(MAIL_USER, MAIL_CODE)  # 发件人的邮件地址 和 发件人的授权密码
)
formatter = logging.Formatter(fmt='%(name)s %(levelname)s %(pathname)s %(lineno)d %(message)s')

mail_handler.setFormatter(formatter)
#  mail_handler 只处理 ERROR 级别以上的日志
mail_handler.setLevel(logging.ERROR)

# 绑定邮件日志处理器到 flask 中

app.logger.addHandler(mail_handler)


@app.route('/')
def index():
    app.logger.info('flask info 日志')
    app.logger.error('flask error 日志')
    return 'log'
```

## 2.5 日志中添加当前的请求信息

```python
class MyFormatter(logging.Formatter):
    def format(self, record):
        # 往 record 中 写入请求的 url 和 remote_adr
        # 1. 读取 request.url remote_adr
        # 2. _request_ctx_stack.top 判断是否在请求构成中
        #     在请求过程中就可以读取请求数据
        #     不在请求过程中就设置默认值
        top = _request_ctx_stack.top
        if top:
            record.url = request.url
            record.remote_addr = request.remote_addr
        else:
            record.url = 'null'
            record.remote_addr = 'null'

        return super(MyFormatter, self).format(record)
```

```python
# 绑定自定义日志处理器
# 日志处理器使用自定义的日志格式类
stream_handler = logging.StreamHandler()

formatter = MyFormatter(fmt='%(name)s %(levelname)s %(pathname)s %(lineno)d %(url)s %(remote_addr)s %(message

stream_handler.setFormatter(formatter)

app.logger.addHandler(stream_handler)


@app.route('/')
def index():
    app.logger.error('测试请求日志内容')
    return 'logger'


app.logger.debug('请求之外打印日志')
app.logger.info('请求之外打印日志')
```

# 3. 限流

## 3.1 基本使用

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)

limiter = Limiter(app=app,
                  #  默认速率
                  default_limits=['1000/day', '100/hour', '10/minute', '3/second'],
                  # 唯一识别一个请求
                  key_func=get_remote_address)


@app.route('/index_1')
@limiter.limit('5/minute')  # 使用自定义速率，覆盖调用默认速率
def index_1():
    return 'index 1'


@app.route('/index_2')
def index_2():  # 使用的是默认速率
    return 'index 2'


@app.route('/index_3')
@limiter.exempt  # 禁用限流
def index_3():
    return 'index 3'
```

## 3.2 自定义错误消息

```python
# 定义限流的错误消息，返回 json 数据
# 借助于 app 异常处理 捕获 429 http 错误状态
@app.errorhandler(429)
def limit_error(error):
    print(error)
    return {'code': 429, 'message': "当前请求过快，请稍后再试"}
```

## 3.3 一个接口设置多个速率

```python
# 设置多个速率 @limiter.limit('速率1;速率2;...')
@app.route('/index_4')
@limiter.limit('2/second;3/minute;10/hour')
def index_4():
    return 'index 4'


# @limiter.limit('速率1')
# @limiter.limit('速率2')
# ....
@app.route('/index_5')
@limiter.limit('2/second')
@limiter.limit('5/minute')
@limiter.limit('20/hour')
def index_5():
    return 'index 5'
```

## 3.4 白名单

```python

# 白名单过滤
# @limiter.request_filter
# def 过滤函数
# 如果有任何一个过滤返回 True,就表明当前请求属于白名单

@limiter.request_filter
def localhost():
    client_ip = request.remote_addr
    if client_ip == '127.0.0.1':
        return True
    return False

如果全部返回 False 那么才会限流

@limiter.request_filter
def super_user():
    # g.user.is_super
    user = getattr(g, 'user', None)
    if user and hasattr(user, 'is_super'):
        return True
    return False

```

## 3.5 自定义客户端唯一标识

```python
# 定义函数，返回客户端唯一标识
def username():
    return session.get('username', 'null')


# 实例化 limiter
limiter = Limiter(app=app,
                  #  默认速率
                  default_limits=['1000/day', '100/hour', '10/minute', '3/second'],
                  # 唯一识别一个请求
                  key_func=get_remote_address)


# 定义路由，使用客户端唯一标识进行限流
@app.route('/index')
@login_required
@limiter.limit('5/minute', key_func=username)  # 传递自定义函数
def index():
    return 'index '


app.run(host='0.0.0.0', port=8000, debug=True)
```

# 4. 聊天室

## 4.1 websocket 介绍

## 4.2 fastapi 介绍和安装

## 4.3 fastapi 应用搭建和运行

### 2.1 开发环境的搭建

fastapi 本身是用 python 所写，所以需要安装其本身 `pip install fastapi`

fastapi 还需要一个 web 服务器来运行用它所写的 web 应用，这里我们可以使用 `pip install uvicorn`

### 2.2 快速开发一个 web 应用

`main.py`

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/index")
async def index():
    return "Hello FastApi"
```

运行服务

```shell
uvicorn main:app --port 8080 --host 0.0.0.0 --reload
```

## 4.4 wesocket 服务端

## 4.5 聊天室搭建

```python
#         await client.send_text(f'当前时间{datetime.now()}')

# 定义一个数据容器保存客户端连接
clients = []


# 定义聊天室后台
@app.websocket('/ws')
async def room(client: WebSocket):
    # 建立客户端连接
    await client.accept()
    # 主动向客户端发送消息
    await client.send_text('你好，欢迎进入聊天室')
    # 保存客户端
    clients.append(client)

    while True:
        # 等待客户端发送消息
        message = await client.receive_text()
        # 广播消息
        for cl in clients:
            await cl.send_text(f'{client.client.host}: {message}')

```