Endterm Project

Тема:

Асинхронная система мониторинга и анализа логов (Async Log Analyzer)

Выполнил:

Сатыбалды Касымжомарт

Introduction

Цель проекта:
Разработать утилиту, которая в реальном времени обрабатывает и анализирует лог-файлы, используя возможности асинхронного и многопоточного программирования.

Используемые технологии:
- Python 3.8+
- Модуль asyncio для асинхронного чтения файлов
- Модуль aiofiles для асинхронного доступа к файлам
- Генераторы и итераторы для потоковой обработки строк
- Модуль concurrent.futures для многопоточного режима
- collections.Counter для подсчёта статистики
- Декораторы для логирования и измерения времени выполнения функций

Проблема, которую решает система:
- Обработка больших лог-файлов в реальном времени без блокировки программы
- Быстрая идентификация ошибок и предупреждений
- Сравнение производительности синхронного, многопоточного и асинхронного подходов

System Overview

Система состоит из:
- async_log_analyzer/ – основная логика проекта
- parser.py – синхронный и асинхронный парсер логов
- analyzer.py – подсчёт статистики (LogStats)
- decorators.py – декораторы для логирования и измерения времени
- exceptions.py – собственные исключения (InvalidLogFormatError, FileReadError)
- main.py – основной скрипт для запуска анализатора
- __init__.py – делает папку пакетом Python
- scripts/log_generator.py – генератор тестовых логов
- sample.log – лог-файл, в который записываются данные
- requirements.txt – зависимости проекта
- readme.txt – информация по запуску

Архитектура:

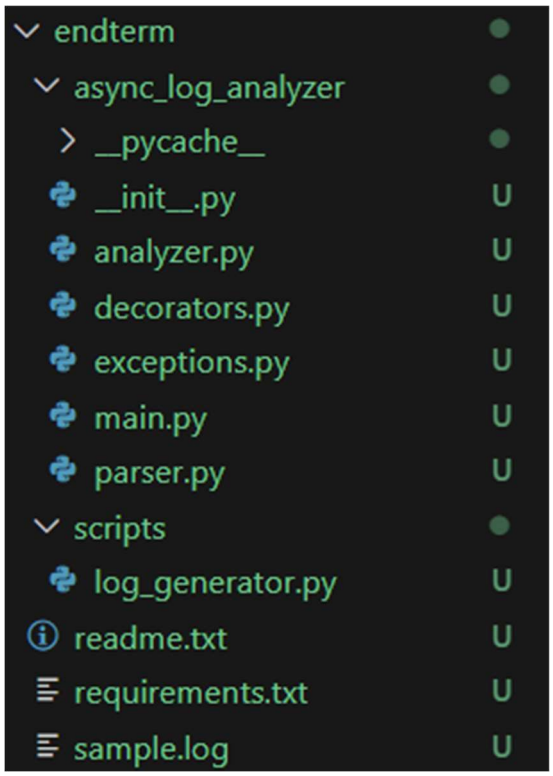

Figure 1 - Architecture of Async Log Analyzer

Implementation Details

Генераторы и итераторы

- Синхронный и асинхронный генераторы обрабатывают файл построчно.
- Позволяют не загружать весь файл в память и обрабатывать строки по мере появления.

Декораторы

- log_call – логирует вызовы функций и их аргументы
- timeit и async_timeit – измеряют время выполнения функций

```python
def log_call(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        logger.debug("Calling %s with args=%s kwargs=%s", func.__name__, args, kwargs)
        result = func(*args, **kwargs)
        logger.debug("%s returned %r", func.__name__, result)
        return result
    return wrapper


def timeit(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        try:
            return func(*args, **kwargs)
        finally:
            end = time.perf_counter()
            logger.info("%s took %.6f seconds", func.__name__, end - start)
    return wrapper
```

Figure 2 - Example of Decorator for Logging

Асинхронное чтение логов

- Используется aiofiles и asyncio для неблокирующего чтения.
- Асинхронный генератор tail_file_async читает новые строки и передаёт их в LogStats.

```python
async def async_reader(path, stats: LogStats, interval=0.1):
    async for entry in tail_file_async(path):
        if entry is None:
            await asyncio.sleep(interval)
            continue
        stats.feed(entry)
```

Figure 3 - Data Flow in Asynchronous Reading Loop

Обработка ошибок

- InvalidLogFormatError – строка не соответствует ожидаемому формату
- FileReadError – ошибка при открытии/чтении файла

Performance Comparison

Время работы разных режимов (пример с sample.log за 20 секунд генерации)

Figure 4 – Execution Time Comparison: Threading vs Asyncio

Режим	Время выполнения (сек)	Пример статистики (INFO/WARNING/ERROR)
Асинхронный (async)	20.02	826 / 884 / 826
Синхронный (sync)	20.24	965 / 1018 / 957
Многопоточный	20.26	4036 / 4216 / 3976

Conclusion

Достигнутые результаты:

- Реализован асинхронный, синхронный и многопоточный анализ логов
- Подсчитывается частота сообщений по уровням
- Асинхронный режим позволяет выводить отчёты каждые 5 секунд без блокировки
- Используются генераторы, декораторы и обработка ошибок

Потенциальные улучшения:

- Добавить поддержку фильтрации по ключевым словам
- Вывод отчётов с визуализацией (графики)
- Многопоточный режим с промежуточными отчетами в реальном времени
- Поддержка нескольких лог-файлов одновременно