

Реализация очереди запуска процессов и запросов к базам данных в веб-ориентированной информационной системе с использованием Python

Фатеенков Данила Витальевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье рассмотрена реализация такой структуры данных как очередь на Python. В представленной реализации задействованы объекты модуля реализации асинхронных вычислений Asyncio. Разработанная структура данных предназначена для запуска недоверенного кода в виде процессов на стороне сервера с последующим сохранением результатов работы кода в отдельную таблицу в БД, спроектированной с помощью СУБД PostgreSQL.

Ключевые слова: структуры данных, очередь запросов, асинхронная очередь запросов, потоки, запуск процессов, асинхронное программирование, Python, Asyncio

Implementing process startup queue and database queries in a web-oriented information system using Python

Fateenkov Danila Vitalievich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article considers the implementation of such data structure as a queue in Python. The presented implementation uses objects of the Asyncio asynchronous computing module. The developed data structure is designed to run untrusted code in the form of processes on the server side with the subsequent saving of the code results to a separate table in the database designed with the help of PostgreSQL DBMS.

Keywords: data structures, query queue, asynchronous query queue, threads, process startup, asynchronous programming, Python, Asyncio

1. Введение

1.1 Актуальность

Объёмы обрабатываемой информации в сфере информационных технологий растут с каждым днём, что формирует задачу оптимизации кода за счёт внедрения более продвинутых алгоритмов и модулей. Такая зависимость между алгоритмами и объёмами информации прослеживается с начала появления программирования как сферы деятельности человека.

Одним из способов оптимизации программного кода можно считать внедрение многопоточных алгоритмов в информационную систему с целью

распараллеливания процессов, что позволяет значительно ускорить работу программы.

Многие классические структуры данных и алгоритмы в настоящее время имеют многопоточную реализацию, а сами модули для создания многопоточных алгоритмов задействуют вариации стандартных структур данных (например, очереди). Таким образом, многопоточные алгоритмы и структуры данных являются одними из наиболее актуальных в настоящее время.

1.2 Обзор исследований

А.В. Скрыпников, В.В. Денисенко и А.О. Амирханян в своей статье рассмотрели асинхронную обработку данных, задачи, в которых она нужна, а также функции очередей сообщений и как они работают [1].

Могилатов Р.К. описал в своей работе особенности и результаты применения шаблона проектирования внедрения зависимостей при разработке асинхронных мониторинговых служб на базе модуля Asyncio языка программирования Python [2].

К. С. Новиков, А. В. Григорьев, А. Г. Избасов, И.И. Кочегаров и Р.К. Валеев в своей работе рассмотрели способы реализации многопоточности на примере программных систем, написанных на языке программирования Python [3]. Также описываются функции стандартных библиотек Treading и Multiprocessing.

Д.А. Дворяк в своей научной работе рассмотрел основные типы динамических структур данных, такие как связанные списки, стеки, очереди, деревья и графы, а также алгоритмы, которые позволяют эффективно работать с ними [4].

Т.А. Ершов и А.А. Голубничий рассмотрели в своей статье объектно-реляционное отображение в разработке приложений при создании запросов к базе данных с помощью модуля ORM SQLAlchemy [5]. В статье рассматриваются варианты современных решений для разработки ORM систем, и описаны преимущества и недостатки этой технологии.

1.3 Цель исследования

Цель – разработать асинхронную очередь для запуска недоверенного кода с последующим сохранением результатов работы в БД на языке программирования Python.

2 Материалы и методы

Для разработки использован язык программирования Python, в качестве основной СУБД задействован PostgreSQL. Также для реализации и организации работы структуры данных задействованы модули SQLAlchemy и FastAPI.

3 Результаты и обсуждения

В первую очередь нужно определить информационную систему, в которую будет внедрена описанная ниже очередь.

На базе Приамурского государственного университета им. Шолом-Алейхема с 2023-го года работа информационная система обучения программированию. В системе есть возможность автоматической проверки решений к задачам на основе заранее заготовленных тестов. Модуль проверки реализован не оптимально и не учитывает многих особенностей (например, перенаправление потоков ввода и вывода, параллельный запуск нескольких тестов одновременно и др.), что формирует необходимость создания полноценного проверочного модуля.

Проверочный модуль на основе нескольких основных компонентов:

1. Модуль компиляторов и интерпретаторов, в котором собраны необходимые для работы ИС языки программирования.
2. Обработчик решений и запросов к БД, реализованный в виде очереди.
3. Конечные точки, предоставляющие возможность отправки и получения решений через API.
4. Модуль подключения и работы с БД, построенной в PostgreSQL.

В рамках рассматриваемой темы нужно обратиться ко второму пункту и разработать для него соответствующий класс очереди. Нужно учесть структуру базы данных, которая задействована в информационной системе (см. рис. 1). В коде программы представлены следующие модели для базы данных:

1. Таблица пользователей, которые совершают отправки недоверенного кода на сервер.
2. Таблица отправок, в которой хранятся записи о всех отправка пользователей.
3. Таблица результатов проверок.
4. Таблица API-токенов, которые выдаются пользователям во время работы приложения для обеспечения безопасности.
5. Таблица пакетных отправок (позволяет пользователю проверить решение на нескольких тестах сразу).
6. Таблица языков программирования с информацией о названиях, версиях и командах для запуска процесса.

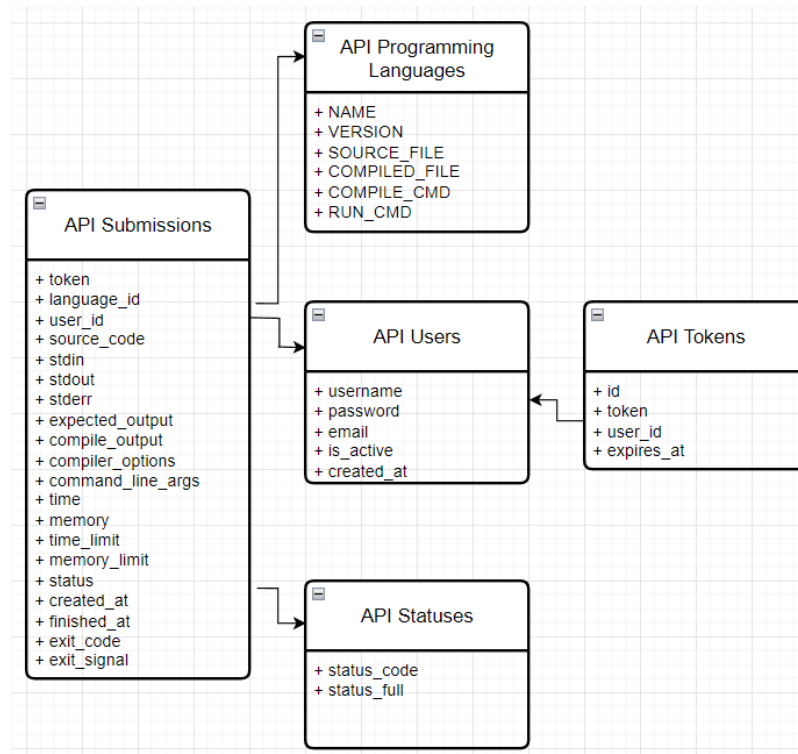


Рисунок 1. Логическая модель базы данных информационной системы

Наибольший интерес в контексте разрабатываемого модуля представляет таблица отправок. Для неё с помощью ORM SQLAlchemy определена следующая структура в коде программы:

```

class SubmissionModel(Base):
    __tablename__ = "submissions"

    id = Column(Integer, primary_key=True, index=True)
    token = Column(String(36), unique=True, nullable=False,
index=True)
    language_id = Column(Integer, ForeignKey("languages.id"),
nullable=False)
    source_code = Column(Text, nullable=False)
    stdin = Column(Text, nullable=True)
    stdout = Column(Text, nullable=True)
    stderr = Column(Text, nullable=True)
    expected_output = Column(Text, nullable=True)
    compile_output = Column(Text, nullable=True)
    compiler_options = Column(String(255), nullable=True)
    command_line_args = Column(String(255), nullable=True)
    time = Column(Integer, default=0, nullable=False)
    memory = Column(Integer, default=0, nullable=False)
    time_limit = Column(Integer, default=2000, nullable=False)
    memory_limit = Column(Integer, default=128000,
nullable=False)
    status_id = Column(Integer, ForeignKey("statuses.id"),
nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow,
nullable=False)
  
```

```

finished_at = Column(DateTime, nullable=True)
exit_code = Column(Integer, nullable=True)
exit_signal = Column(Integer, nullable=True)
batch_id = Column(Integer, ForeignKey("batches.id"),
nullable=True)
user_id = Column(Integer, ForeignKey("users.id"),
nullable=False)

language = relationship("LanguageModel",
back_populates="submissions")
status = relationship("StatusModel",
back_populates="submissions")
batch = relationship("BatchModel",
back_populates="submissions")
user = relationship("UserModel",
back_populates="submissions")

```

Таблица 1. Структура таблицы «API Submissions»

Название атрибута	Тип данных	Краткое описание
token	VARCHAR	Токен проверки решения
language_id	INT	Внешний ключ на таблицу языков программирования
user_id	INT	Внешний ключ на таблицу пользователей
source_code	VARCHAR	Исходный код программы
stdin	VARCHAR	Входные данные
stdout	VARCHAR	Выходные данные
stderr	VARCHAR	Поток ошибок
expected_output	VARCHAR	Ожидаемый вывод
compile_output	VARCHAR	Информация о компиляции программы
compiler_options	VARCHAR	Настройки компилятора
command_line_args	VARCHAR	Аргументы командной строки
time	INT	Время, затраченное на выполнение процесса
memory	INT	Память, затраченная на выполнение процесса
time_limit	INT	Ограничение по времени
memory_limit	INT	Ограничение по памяти
status	INT	Внешний ключ на таблицу статусов проверки
created_at	DATETIME	Время создания записи
finished_at	DATETIME	Время завершения проверки решения
exit_code	INT	Код завершения работы программы
exit_signal	INT	Код возврата программы

Принцип работы API можно описать кратко следующими шагами:

1. Пользователь получает API-токен по логину и паролю, после чего ему открывается доступ к отправкам.
2. Пользователь формирует запрос к API в JSON-формате и отправляет его в виде POST-запроса на сервер.
3. На стороне сервера запускается процесс и код, отправленный пользователем, проверяется на тестовых входных и выходных данных.
4. Результат сохраняется в БД и пользователю возвращается токен отправки, по которому он может просмотреть результат работы процесса.

Первый и последний этап в рамках данной статьи не будут, а вот второй и третий непосредственно связаны с разрабатываемой очередью.

В проекте ИС был определён класс `SubmissionQueueManager`, отвечающий за обработку решений. Класс работает по принципу стандартной очереди. Очередь – это тип структуры данных, в которой элементы хранятся в определённой последовательности. В отличие от стеков, операции с очередью выполняются по принципу FIFO (первым пришёл – первым обслужен). Элемент, который вставляется первым, первым и удаляется (см. рис. 2). В рамках разрабатываемой ИС такая структура данных предоставит необходимый функционал для обработки пользовательских отправок.

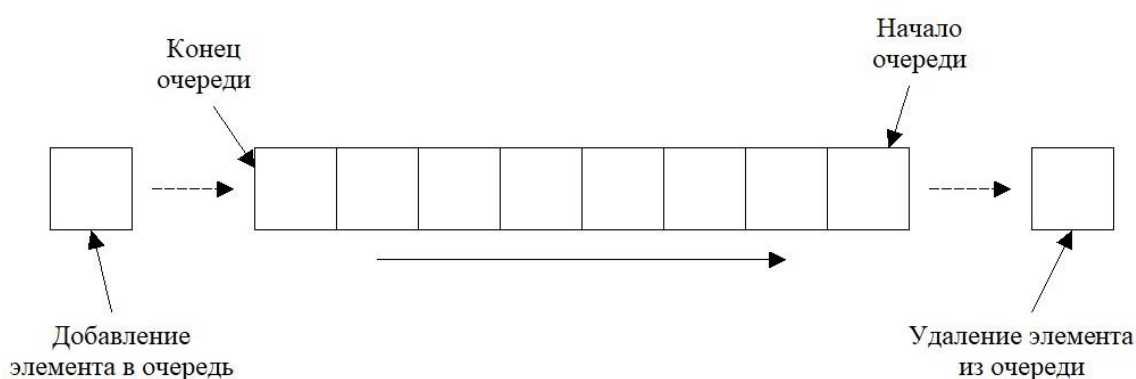


Рисунок 2. Схематичное представление работы очереди

Были определены следующие методы для `SubmissionQueueManager`:

1. Метод инициализации класса (`__init__`).
2. Метод начала работы очереди (`start`).
3. Метод завершения работы очереди (`stop`).
4. Метод добавления новой отправки в очередь (`enqueue_submission`).
5. Метод-обработчик массива отправок (`worker`).
6. Метод-обработчик каждой отправки по отдельности (`handle_submission`).

В методе инициализации класса задаются атрибуты класса и дополнительные параметры для дальнейшей работы. Были выделены следующие атрибуты: очередь из модуля `Asyncio`, семафор из того же модуля, переменная `worker_task` (будет хранить задачу, которая должна быть выполнена очередью, на этапе инициализации значение отсутствует) и логическая переменная `shutdown` (нужна как сигнал о том, что очередь в текущий момент времени не работает, по умолчанию `False`). Метод инициализации запускается автоматически при создании объекта соответствующего класса. Входными данными в методе являются параметр `self` и максимальное количество потоков `max_concurrent`.

Стоит подробнее остановиться на первых двух атрибутах: очереди и семафоре.

Основное отличие классической реализации очереди от реализации `Asyncio` заключается в возможности работы с корутинами – функциями,

которые могут быть приостановлены и возобновлены асинхронно. Такая реализация очереди больше подходит для дальнейшей работы, чем классическая, поэтому в методе инициализации вызывается объект «`asyncio.Queue`».

Семафор – примитив синхронизации работы процессов и потоков. В его основе лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу.

Используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

В Python параметр `self` в методах классов используется для ссылки на текущий экземпляр класса. Это необходимо для того, чтобы метод мог работать с атрибутами и другими методами этого конкретного экземпляра.

Итоговый вариант метода инициализации следующий:

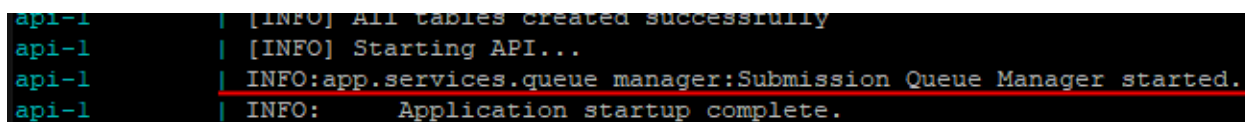
```
def __init__(self, max_concurrent: int):
    self.queue = asyncio.Queue()
    self.semaphore = asyncio.Semaphore(max_concurrent)
    self.worker_task = None
    self.shutdown = False
```

Метод начала работы объекта класса не содержит дополнительных входных параметров, только `self`. В методе производится проверка – запущен ли объект класса в текущий момент времени (если запущен, то атрибут `worker_task` не будет пустым, а занят обработкой отправок через метод `worker`). Если объект запущен, то метод завершит свою работу. Иначе метод создаст корутину с помощью метода `create_task` очереди `queue`.

`asyncio.create_task()` – это функция из библиотеки `asyncio` в Python, которая создаёт задачу из корутины. Она планирует корутину для асинхронного выполнения и возвращает объект задачи, который можно использовать для отслеживания её статуса, отмены или ожидания завершения.

Также метод в консоль выводит сообщение о начале работы очереди (см. рис. 3) и сам метод представляет собой асинхронную функцию. Итоговый вариант метода следующий:

```
async def start(self):
    if self.worker_task is None:
        self.worker_task = asyncio.create_task(self.worker())
        logger.info("Submission Queue Manager started.")
```



```
api-1 | [INFO] All tables created successfully
api-1 | [INFO] Starting API...
api-1 | INFO:app.services.queue manager:Submission Queue Manager started.
api-1 | INFO: Application startup complete.
```

Рисунок 3. Запуск работы очереди после инициализации всех компонентов на стороне сервера

Метод завершения работы в очередь добавляет пустое значение (None), который будет останавливать выполнение метода worker (далее будет описан) и ждать завершения корутины в атрибуте worker_task. После завершения работы корутины выводится сообщение о том, что очередь завершила свою работу.

Итоговый вариант метода следующий:

```
async def stop(self):
    self.shutdown = True
    if self.worker_task:
        await self.queue.put(None)
        await self.worker_task
    logger.info("Submission Queue Manager stopped.")
```

Метод enqueue_submission добавляет в очередь кортеж из двух параметров: объект отправки (представлен как SubmissionModel и содержит данные об отправке пользовательского кода) и ссылка на объект сессии БД (представлен как Session и предоставляет доступ к выборке и добавлению данных из таблицы отправок). После постановки отправки в очередь выводится соответствующее сообщение в лог сервера (см. рис. 4).

```
async def enqueue_submission(self, submission: SubmissionModel,
db: Session):
    await self.queue.put((submission, db))
    logger.info(f"Submission {submission.token} enqueued.")
```

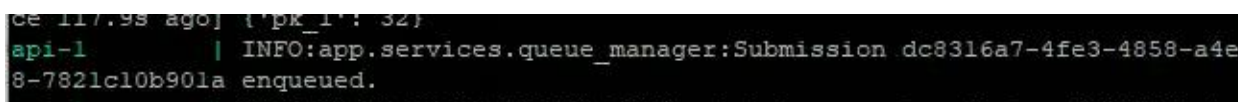
The image is a screenshot of a terminal window with a dark background. It shows a log entry from a service named 'api-1'. The log message is 'INFO:app.services.queue_manager:Submission dc8316a7-4fe3-4858-a4e8-7821c10b901a enqueued.' The text is displayed in a light green or cyan color. Above the main log line, there is a timestamp '2024-12-26 03:23:07.673' and another log level 'INFO'.

Рисунок 4. Отправка пользовательского кода в очередь

Метод worker представляет собой цикл, который будет выполняться до тех пор, пока очередь не будет выключена (то есть пока атрибут shutdown равен False). В цикле производится получение первой добавленной отправки, а затем проверяется – пустой ли данный объект (то есть равна ли она None) и если условие выполняется, очередь завершает свою работу (пустой объект добавляется из метода stop).

Если объект не равен None, то извлекается отправка и ссылка на сессию подключения к БД из объекта и после метод получает доступ к семафору через метод «semaphore.acquire()». Данный метод проверяет значение внутреннего счётчика: если оно больше нуля, то счётчик уменьшается на единицу и метод возвращает True. Если значение счётчика равно нулю, то вызвавший метод поток блокируется до тех пор, пока не будет вызван метод release().

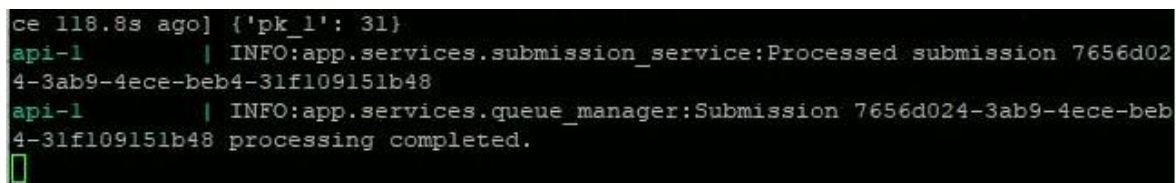
После проверки семафора создаётся новая задача с помощью asyncio.create_task, которой передаётся метод handle_submission с ссылкой на отставку и сессию подключения к БД, которые были извлечены из очереди ранее:

```
async def worker(self):
    while not self.shutdown:
        item = await self.queue.get()
        if item is None:
            break
        submission, db = item
        await self.semaphore.acquire()
        asyncio.create_task (self.handle_submission(submission,
db) )
```

Последний метод `handle_submission` передаёт отправку на обработку в отдельную функцию (не представлена в классе `SubmissionQueueManager`) и сохраняет результат выполнения кода в соответствующую таблицу. Обработка данных обёрнута в обработчик исключений для получения информации о возникающих ошибках с последующим продолжением работы программы.

Независимо от результата работы функции обработки кода значение счётчика семафора будет увеличено с помощью метода «`release()`», а задача в очереди будет завершена через метод «`task_done()`», после чего в лог работы сервера будет выведено сообщение о завершении обработки отправки (см. рис. 5):

```
async def handle_submission(self, submission: SubmissionModel,
db: Session):
    try:
        logger.info(f"Processing submission
{submission.token}.")
        submission.status_id = 2
        db.commit()
        await process_submission(submission, db)
    except Exception as e:
        logger.exception(f"Error processing submission
{submission.token}: {e}")
    finally:
        self.semaphore.release()
        self.queue.task_done()
        logger.info(f"Submission {submission.token} processing
completed.")
```



```
ce 118.8s ago] {'pk_1': 31}
api-1 | INFO:app.services.submission_service:Processed submission 7656d02
4-3ab9-4ece-beb4-31f109151b48
api-1 | INFO:app.services.queue_manager:Submission 7656d024-3ab9-4ece-beb
4-31f109151b48 processing completed.
```

Рисунок 5. Сообщение о завершении обработки отправки функцией «`process_submission`»

Для тестирования был сформирован скрипт на Python, отправляющий POST с кодом программы и тестовыми данными, а также ограничениями по времени и памяти (тестировалась как одиночная, так и пакетная отправка). В

результате был сформирован ответ со стороны сервера и получен токен для просмотра результата обработки отправки (см. рис. 6). Скрипт представлен ниже:

```
import requests
import json

url = "http://localhost/v1/submissions/"
token = input("Enter token: ")

payload = json.dumps({
    "language_id": 13,
    "source_code":
"YSxiID0gbWFwKGludCxpbnBldCgpLnNwbGl0KCkpCnByaW50KGErYik=",
    "stdin": "MiAz",
    "expected_output": "NQ==",
    "compiler_options": None,
    "command_line_args": None,
    "time_limit": 1000,
    "memory_limit": 32000
})
headers = {
    'Authorization': f'Bearer {token}',
    'Content-Type': 'application/json'
}

response = requests.request("POST", url, headers=headers,
data=payload)

print(response.text)
```

Также была для тестирования сформирована пакетная отправка, состоявшая из 3-х тестов одной и той же программы. В результате сервер вернул как токен пакетной отправки, так и токен каждого теста по отдельности (см. рис. 6). При выводе результата по токену пакетной отправки можно просмотреть результат обработки каждого теста.

Международной научно-практической конференции, Петрозаводск, 05 марта 2024 года. – Петрозаводск: Международный центр научного партнерства «Новая Наука» (ИП Ивановская И.И.), 2024. С. 108-114.

5. Ершов Т.А., Голубничий А.А. Введение в Object-Relational Mapping, работа с данными Postgresql с использованием языка Python и SQLAlchemy // Научно-технический вестник Поволжья. 2024. № 1. С. 151-154.