Улучшенный планировщик заданий Python

Мод… …естен как Advanced Python Scheduler (улучшенный планировщик заданий). Как следует из наз… …родвинутых библиотек планировщика, доступных в Python, с множеством различных функций и пар…

Ус… …г в виртуальное окружение.

Так… …входит в стандартную библиотеку Python, его необходимо установить отдельно. сделать это мож… …етов pip.

```
# …ие, если нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# обновляем `pip`
(VirtualEnv):~$ python3 -m pip install -U pip
# ставим модуль `APScheduler`
(VirtualEnv):~$ python3 -m pip install apscheduler
```

> *Примечание: Стандартная библиотека Python также имеет планировщик событий общего назначения sched.*

Содержание:

Основные понятия модуля APScheduler.

- **Триггеры** содержат логику планирования. Каждое задание имеет свой собственный триггер, который определяет, когда задание должно быть запущено следующим. Помимо своей первоначальной конфигурации, триггеры не имеют состояния.
- **Хранилище заданий** содержат запланированные задания. Хранилище заданий *по умолчанию* просто хранит задания в памяти, но другие встроенные хранилища хранят их в различных базах данных. Данные задания сериализуются, когда они сохраняются в …янном хранилище заданий, и десериализуются, когда они загружаются обратно из него. Хранилища заданий (кроме

хранилища по умолчанию) выступают в качестве посредников для сохранения, загрузки, обновления и поиска заданий в бэкэнде . Хранилища заданий никогда не должны использоваться совместно планировщиками.

- **Исполнители** - это то, что обрабатывает выполнение заданий. Другими словами исполнитель, отправляют назначенный вызываемый объект (функцию) в задании в поток или пул процессов. Когда задание выполнено, он уведомляет планировщик, который затем генерирует соответствующее событие.

- **Планировщик** - это то, что связывает все вместе. Другими словами это надлежащий интерфейс для работы с хранилищами заданий, исполнителями или триггерами. Настройка хранилищ заданий и исполнителей выполняется через планировщик, как и добавление, изменение и удаление заданий.

Как правильно выбрать планировщик, хранилище и исполнителей.

- BlockingScheduler: используется, когда в процессе работает только планировщик, например, отдельный скрипт/модуль, заменяющий системный cron операционной системы Linux .
- BackgroundScheduler: используется, когда не используются фреймворки, указанные ниже и необходимо, чтобы планировщик работал в фоновом режиме внутри какого-то приложения.
- AsyncIOScheduler: используется, если приложение использует модуль asyncio.
- GeventScheduler: используется, если приложение использует gevent.
- TornadoScheduler: используется, если создается приложение Tornado.
- TwistedScheduler: используется, если создается приложение Twisted.
- QtScheduler: используйте, если создается приложение Qt.

Если приложение использует один из вышеперечисленных фреймворков, то выбор исполнителя очевиден. В противном случае, для большинства целей исполнителя по умолчанию ThreadPoolExecutor должно быть достаточно. Если рабочая нагрузка включает в себя операции с интенсивным использованием ЦП, то следует рассмотреть возможность использования ProcessPoolExecutor, чтобы использовать несколько ядер ЦП. Можно использовать оба сразу, добавив исполнитель пула процессов в качестве вторичного исполнителя.

Чтобы выбрать подходящее хранилище заданий, нужно понять, должны ли задания сохраняться после перезапуска планировщика или нет. Если задания всегда воссоздаются в начале приложения кодом, то, можно использовать значение по умолчанию (MemoryJobStore). Но если нужно, чтобы задания сохранялись после перезапуска планировщика или сбоя приложения, то выбор обычно сводится к тому, какие инструменты используются в кодовой базе приложения. Если есть свободный выбор, то рекомендуется использовать SQLAlchemyJobStore, а на сервере PostgreSQL из-за надежной защиты целостности данных.

Триггеры планировщика заданий APScheduler

Когда планируется задание, то для него нужно выбрать триггер. Триггер определяет логику, по которой вычисляются даты/время при запуске задания. Модуль APScheduler поставляется с тремя встроенными типами триггеров:

- date: используется, когда нужно запустить задание только один раз в определенный момент времени.
- interval: используется, когда нужно запускать задание через фиксированные интервалы времени.
- cron: используется, когда нужно периодически запускать задание в определенное время дня.

Также возможно объединить несколько триггеров в один, который срабатывает либо в моменты времени, согласованные всеми участвующими триггерами, либо когда сработает любой из триггеров.

Настройка планировщика APScheduler.

APScheduler предоставляет множество различных способов настройки. Можно использовать словарь конфигурации или передать параметры в качестве ключевых аргументов . Также можно сначала создать экземпляр планировщика, а затем добавить задания и настроить планировщик.

Допустим, необходимо иметь два хранилища заданий с использованием двух исполнителей, а также настроить значения по умолчанию для новых заданий и установить другой часовой пояс.

- MongoDBJobStore с именем 'mongo',
- SQLAlchemyJobStore с именем 'default' (с использованием SQLite),
- ThreadPoolExecutor с именем 'default' и числом рабочих процессов, равным 20,
- ProcessPoolExecutor с именем 'processpool' с числом рабочих процессов, равным 5,
- UTC как часовой пояс планировщика,
- для новых заданий по умолчанию 'coalesce' отключено,
- для новых заданий максимальное количество экземпляров по умолчанию равно 3

Варианты настройки APScheduler:

Вверх

Следующие три варианта настройки полностью эквивалентны.

Вариант 1.

```
from pytz import utc
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ProcessPoolExecutor

jobstores = {
    'mongo': {'type': 'mongodb'},
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': {'type': 'threadpool', 'max_workers': 20},
    'processpool': ProcessPoolExecutor(max_workers=5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}

# СОЗДАЕМ ФОНОВЫЙ ПЛАНИРОВЩИК
# ЗАДАНИЙ ПО УМОЛЧАНИЮ
scheduler = BackgroundScheduler()

# настраиваем фоновый планировщик заданий
scheduler.configure(jobstores=jobstores, executors=executors,
           job_defaults=job_defaults, timezone=utc)

# добавляем задание
scheduler.add_job(job_func, trigger='interval', id='job_1', hour=1,
                  jobstore='default', executor='default')
```

Вариант 2.

```
from apscheduler.schedulers.background import BackgroundScheduler

# создаем и настраиваем планировщик заданий
# префикс `apscheduler` жестко запрограммирован
scheduler = BackgroundScheduler({
    'apscheduler.jobstores.mongo': {
         'type': 'mongodb'
    },
    'apscheduler.jobstores.default': {
        'type': 'sqlalchemy',
        'url': 'sqlite:///jobs.sqlite'
    },
    'apscheduler.executors.default': {
        'class': 'apscheduler.executors.pool:ThreadPoolExecutor',
        'max_workers': '20'
    },
    'apscheduler.executors.processpool': {
        'type': 'processpool',
        'max_workers': '5'
    },
    'apscheduler.job_defaults.coalesce': 'false',
    'apscheduler.job_defaults.max_instances': '3',
    'apscheduler.timezone': 'UTC',
})

# добавляем задание
scheduler.add_job(job_func, trigger='interval', id='job_1', hour=1,
                  jobstore='mongo', executor='default')
```

Вариант 3.

```
from pytz import utc
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.mongodb import MongoDBJobStore
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ThreadPoolExecutor, ProcessPoolExecutor
```

```
jobstores = {
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
    'mongo': MongoDBJobStore(),
}
                            ⋮
executors = {
    'default': ThreadPoolExecutor(20),
    'processpool': ProcessPoolExecutor(5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
# создаем и настраиваем планировщик заданий
scheduler = BackgroundScheduler(jobstores=jobstores, executors=executors,
                                job_defaults=job_defaults, timezone=utc)

# добавляем задание
scheduler.add_job(job_func, trigger='interval', id='job_1', hour=1,
                  jobstore='mongo', executor='processpool')
```

Пример планирования заданий с помощью APScheduler.

> *Необходимо понимать, что "ЗАДАНИЕ" - это задача, которая назначается планировщику, например, выполнение определенной функции.*

Создание планировщика APScheduler.

Прежде чем что-либо делать, необходимо настроить планировщик. В примерах не будем использовать никаких специальных фреймворков, а просто настроим и запустим простой планировщик. Этого будет более чем достаточно для большинства случаев использования.

```
from apscheduler.schedulers.background import BackgroundScheduler

# Создает фоновый планировщик по умолчанию
sched = BackgroundScheduler()
```

Планировщик по умолчанию использует хранилище заданий на основе оперативной памяти MemoryJobStore и имеет максимальное количество потоков, равное 10.

Добавление заданий с триггером interval.

Наиболее распространенный способ добавления заданий в планировщик - использование метода .add_job(). Этот метод также возвращает объект Job, который можно использовать в дальнейшей обработке позже. Например, изменение или удаление задания из планировщика.

Следующий код выполняет функцию prompt() каждые 5 секунд.

```
from apscheduler.schedulers.background import BackgroundScheduler

# Создает ФОНОВЫЙ планировщик
scheduler = BackgroundScheduler()

# функция - задание
def prompt():
    print("Executing Task...")

# планирование задания
scheduler.add_job(prompt, 'interval', seconds=5)

# Запуск запланированных заданий
scheduler.start()

# Запускает бесконечный цикл
while True:
    sleep(1)
```

Первым аргументом метод Scheduler.add_job() принимает имя вызываемой функции prompt(). Второй аргумент определяет триггер 'i Вверх . Аргументы, отвечающие за время выполнения задания будут индивидуальными, в зависимости от использования триггера. С триггером interval принимаются такие временные аргументы как: seconds, minutes, hours, day, week, month и

year.

Обратите внимание на бесконечный цикл в конце кода. *Фоновый планировщик* BackgroundScheduler, как следует из названия, запускает отдельный поток от основного потока. Как только основной поток завершает выполнение, поток, в котором работает фоновый планировщик, также закрывается. Это происходит, даже если есть запланированные задания для выполнения.

Если запустить приведенный выше код, в котором нет бесконечного цикла, то можно заметить, что планировщик не сработает. Это связано с тем, что основной поток завершил выполнение после вызова метода .start(). Использование в конце кода бесконечного цикла предотвращает закрытие основного потока.

Блокирующий планировщик BlockingScheduler в отличие от фонового работает в основном потоке. Следовательно, пока планировщик не будет остановлен в основном потоке с помощью метода .shutdown(), основной поток не закроется.

Код в приведенном ниже примере использует BlockingScheduler вместо BackgroundScheduler. Обратите внимание, что он будет продолжать работать, несмотря на отсутствие бесконечного цикла.

```
from time import sleep
from apscheduler.schedulers.background import BlockingScheduler

# Создает БЛОКИРУЮЩИЙ планировщик
scheduler = BlockingScheduler()

def prompt():
    print("Executing Task...")

scheduler.add_job(prompt,'interval', seconds=5)

scheduler.start()
```

Метод Scheduler.add_job() и декоратор @Scheduler.scheduled_job().

Синтаксис:

```
job = Scheduler.add_job(func, trigger=None, args=None, kwargs=None,
                        id=None, name=None, misfire_grace_time=undefined,
                        coalesce=undefined, max_instances=undefined,
                        next_run_time=undefined, jobstore='default', executor='default',
                        replace_existing=False, **trigger_args)

# версия декоратора `Scheduler.add_job()`,
# за исключением того, что `replace_existing`
# всегда имеет значение `True`.
@Scheduler.scheduled_job(trigger=None, args=None, kwargs=None, ...)
def job():
    pass
```

Метод Scheduler.add_job() и декоратор @Scheduler.scheduled_job() добавляет данное задание в список заданий и пробуждает планировщик, если он уже запущен.

Любой аргумент, который по умолчанию имеет значение undefined, будет заменена соответствующим значением по умолчанию при планировании задания (что происходит при запуске планировщика или сразу же, если планировщик уже запущен).

Аргумент func может быть задан либо как вызываемый объект, либо как текстовая ссылка в формате package.module:some.object, где первая половина (разделенная :) - это импортируемый модуль, а вторая половина - ссылка на вызываемый объект относительно модуля.

Аргумент trigger может быть: 'date', 'interval' или 'cron'.

Экземпляр триггера принимает аргументы:

- args (list|tuple): - список/кортеж позиционных аргументов для вызова func.
- kwargs (dict): - словарь ключевых аргументов для вызова func.
- id (str): - явный идентификатор задания (для его последующего изменения).
- name (str): - текстовое описание задания.
- misfire_grace_time (int): - секунд после назначенного времени выполнения, когда задание все еще может быть запущено (или None, чтобы разрешить выполнение задания независимо от того, насколько оно запоздало)
- coalesce (bool): - запускать один раз, а не много раз, если планировщик определяет, что задание должно выполняться более одного раза подряд.

Вверх

- max_instances ([int]): - максимальное количество одновременно запущенных экземпляров, разрешенных для этого задания.
- next_run_time ([datetime]): - когда запускать задание в первый раз, независимо от триггера (None, чтобы добавить задание как приостановленное).
- jobstore ([str]): - псевдоним хранилища заданий для хранения задания
- executor ([str]): - псевдоним исполнителя, с которым выполняется задание
- replace_existing ([bool]): - True для замены существующего задания с тем же идентификатором (но с сохранением количества запусков из существующего)

Планирование задания APScheduler с триггером date.

Триггер date используется для выполнения задания в определенный момент времени. Такой триггер срабатывает/выполняется только один раз.

Ниже демонстрируется пример с триггером date и возможность передать аргументы вызываемой функции (заданию) с помощью аргумента args. Это то, что является общим для всех трех триггеров. В примере также показывается три различных способа передачи значения аргумента run_date.

```
from time import sleep
from datetime import date, datetime
from apscheduler.schedulers.background import BlockingScheduler

scheduler = BlockingScheduler()

# Функция - задание
def job(text):
    print(text)

# выполнит функцию `job()` один раз 30.11.2022 г.
scheduler.add_job(job, 'date', run_date=date(2022, 11, 30),
                  args=['Job 1'], id='job_1')

# выполнит функцию `job()` один раз 30.11.2022 г. в 12:00:00
scheduler.add_job(job, 'date', run_date=datetime(2022, 4, 30, 12, 0, 0),
                  args=['Job 2'], id='job_2')

# выполнит функцию `job()` один раз 30.11.2022 г. в 8:00
scheduler.add_job(job, 'date', run_date='2022-4-30 08:00:00',
                  args=['Job 3'], id='job_3')

scheduler.start()
```

Обратите внимание на аргумент id - индификатор задания. Этот индификатор необходим для удаления заданий и должен быть уникальным.

Планирование задания APScheduler с триггером cron.

Триггер cron является очень популярным и мощным типом триггера. Триггер cron используется для периодического планирования задач, которые должны повторяться. Например, выполнение задачи в определенное время каждый день или выполнение задачи в определенные дни (например, с понедельника по пятницу).

Рассмотрим 5 разных примеров, каждый из которых демонстрирует разные способы планирования заданий cron с помощью APScheduler.

```
# Выполняется ежедневно, каждую минуту, в 17 часов
scheduler.add_job(job, 'cron', hour=17, minute='*',
                  args=['job 1'])
```

В коде примера используется специальный символ '*'. Это приведет к тому, что задание будет повторяться каждую минуту. Если использовать '*' с аргументом second, то задание будет повторяться каждую секунду.

```
# Выполняется ежедневно, каждые 5 минут, в 17 часов
scheduler.add_job(job, 'cron', hour=17, minute='*/5',
                  args=['job 2'])
```

Пример является вариантом первого, но использует формат '*/n'. Это приводит к тому, что задание повторяется каждые n минут.

Вверх

```
# Выполняется ежедневно, в 17:25 и 18:25
scheduler.add_job(job, 'cron', hour='17-18', minute='25',
                  args=['job 3'])
```

Пример показывает, как можно указать диапазон значений. Следовательно, это позволяет нам выполнять код в течение 2 часов, а не только один.

```
# Планирует запуск задания на третью пятницу
# в июне, июле, августе, ноябре и декабре
# в 00:00, 01:00, 02:00, 03:00 и 04:00.
scheduler.add_job(job, 'cron', month='6-8,11-12', day='3rd fri',
                  hour='0-4', args=['job 4'])
```

Пример демонстрирует две новые вещи. Во-первых, показывает, что можно указать несколько диапазонов, просто добавляя запятые между ними. Во-вторых, он показывает использование формата Xth Y, который выполняет задание в X по счету неделю, определенного дня недели каждого месяца, при этом значение Y может быть числом дня недели.

Есть еще один вариант, который можно использовать в аргументе day это значение 'last'. Использование day='last' приводит к тому, что задание запускается в последний день месяца. Использование значения, например day='last fri' или day='last 5' запланирует задание на последнюю пятницу месяца.

```
# Работает с понедельника по пятницу в 06:30 до 31.12.2022 00:00:00.
scheduler.add_job(job, 'cron', day_of_week='0-4', hour=6,
                  minute=30, end_date='2022-12-31',
                  args=['job 5'])
```

Приведенный выше код в основном демонстрирует тот факт, что можно использовать аргумент end_date, чтобы остановить выполнение кода после достижения определенной даты.

Смотрим еще пару примеров:

```
# запускает задание каждые 2 часа,
# в промежутке между 13 и 24 часами.
scheduler.add_job(job, 'cron', hour="12-23/2", args=['job 6'])
```

При планировании большого количества задач на определенное время может вызвать всплеск активности за долю секунды. Чтобы немного сбалансировать эти задания и добавить к ним случайный компонент, необходимо использовать аргумент jitter.

```
scheduler.add_job(job, 'cron', hour='*', jitter=60, args=['job 7'])
```

Пример выше будет выполнять функцию задания каждый час, но с дополнительной случайной задержкой от -60 до +60 секунд.

Параметры триггера cron.

- year: принимает значение года в 4-х значном виде.
- month: принимает значение от 1 до 12.
- day: принимает значение от 1 до 31.
- week: принимает значения от 1 to 53 (номер недели в году).
- day_of_week: принимает значения от 0 до 6, или mon, tue, wed, thu, fri, sat, sun (день недели).
- hour: принимает значения от 0 to 23
- minute: принимает значения от 0 to 59
- start_date: datetime или строка - начала выполнения задания.
- end_date: datetime или строка - окончания выполнения задания.
- timezone: datetime.tzinfo или строка - часовой пояс, который будет использоваться для вычисления даты и времени.
- jitter: задерживает выполнение задания на несколько секунд.

Типы выражений триггера cron.

- x: Срабатывает при возникновении x. hour=3, означает срабатывание на 4-м часу.
- *: Срабатывает по каждому значению. hour='*' означает каждый час.
- */a: Срабатывает при каждом значении a. hour="*/3" означает каждые 3 часа.
- a-b: Срабатывает при каждом значении между a и b включая эти значения.
- a-b/c: Срабатывает при каждом значении c в диапазоне a-b.
- Xth Y: Срабатывает при X наступлении дня недели Y в течение месяца (только для аргумента day).
- last X: Срабатывает на последнем X, где x - любой день недели. например: последняя пятница (только для аргумента

Вверх

- `last`: Срабатывает в последний день месяца (только для аргумента day).
- `x,y,z`: Используется для объединения различных выражений с помощью запятой.

Получение списка запланированных заданий APScheduler.

Чтобы получить обрабатываемый машиной список запланированных заданий, можно использовать метод Scheduler.schedget_jobs(). Он вернет список экземпляров Job. Если интересуют только задания, содержащиеся в определенном хранилище заданий, то укажите псевдоним хранилища заданий в качестве второго аргумента.

Для удобства можно использовать метод Scheduler.print_jobs(), который распечатает отформатированный список заданий, их триггеры и время следующего выполнения.

Изменение заданий APScheduler.

Можно изменить любые атрибуты задания, вызвав либо Job.modify() (для экземпляра Job, который был получен при добавлении задания Scheduler.add_job()), либо Scheduler.modify_job(id) для изменения определенного id задания. При этом можно изменить любые атрибуты задания, кроме идентификатора id задания.

Например:

```
job.modify(max_instances=6, name='Alternate name')
# или
scheduler.modify_job(id='job_1', max_instances=6, name='Alternate name')
```

Если необходимо перепланировать задание, то есть изменить его триггер, то можно использовать либо Job.reschedule(), либо Scheduler.reschedule_job(). Эти методы создают новый триггер для задания и пересчитывают время его следующего выполнения на основе нового триггера.

```
job.reschedule(trigger='cron', minute='*/5')
# или
scheduler.reschedule_job(id='job_1', trigger='cron', minute='*/5')
```

Удаление заданий APScheduler.

Если удаляется задание из планировщика, то оно удаляется из связанного с ним хранилища заданий и больше не будет выполняться. Есть два способа сделать это:

- вызвав Scheduler.remove_job() с идентификатором задания и псевдонимом хранилища заданий.
- вызвав Job.remove() для экземпляра Job, который был получен при добавлении задания Scheduler.add_job().

Последний способ, вероятно, более удобен, но требует сохраненного экземпляра Job.

Если расписание задания завершается (т. е. его триггер больше не создает время выполнения), оно автоматически удаляется.

```
# используется экземпляр `Job`
job = scheduler.add_job(myfunc, 'interval', minutes=2)
job.remove()
```

То же самое, используя явный идентификатор задания id:

```
scheduler.add_job(myfunc, 'interval', minutes=2, id='my_job_id')
scheduler.remove_job('my_job_id')
```

Приостановка/возобновление отдельных заданий APScheduler.

Можно легко приостанавливать и возобновлять задания с помощью экземпляра задания или самого планировщика. Когда задание приостановлено, то время его следующего выполнения сбрасывается, и для него больше не будет рассчитываться время выполнения, пока задание не будет возобновлено. Чтобы приостановить задание, используйте любой метод:

- Job.pause() - приостанавливает задание для экземпляра Job, который был получен при добавлении задания Scheduler.add_job().
- Scheduler.pause_job(id) - приостанавливает конкретное задание по его id.

Возобновить:

- Job.resume() - возобновляет задание для экземпляра Job, который был получен при добавлении задания Scheduler.add_job().
- Scheduler.resume_job(id) - возобновляет конкретное задание по его id.

Приостановка/возобновление работы APScheduler.

Можно приостановить обработку запланированных заданий:

```
scheduler.pause()
```

Это приведет к тому, что планировщик не будет пробуждаться до тех пор, пока обработка не будет возобновлена:

```
scheduler.resume()
```

Также возможен запуск планировщика в состоянии паузы, то есть без первого пробуждающего вызова:

```
scheduler.start(paused=True)
```

Это полезно, когда нужно удалить нежелательные задания, прежде чем они смогут быть запущены.

Завершение работы планировщика APScheduler.

Чтобы закрыть планировщик:

```
scheduler.shutdown()
```

По умолчанию планировщик закрывает свои хранилища заданий и исполнителей и ждет, пока не будут завершены все текущие выполняемые задания. Если не надо ждать, то можно сделать следующее:

```
scheduler.shutdown(wait=False)
```

Этот код по-прежнему закроет хранилища заданий и исполнителей, но не будет ждать завершения каких-либо запущенных задач.

Вверх