aiomisc Documentation

Выпуск 17.3.0

Dmitry Orlov

мар. 08, 2024

Оглавление

Вам, как программисту, знакомы проблемы, связанные с дизайном и обслуживанием программ. Одним из мест, которое может быть особенно сложным, является создание архитектуры программы, использующего асинхронный ввод-вывод.

Вот тут на сцену выходит `aiomisc`. Это библиотека Python, которая предоставляет набор служебных функций и классов для работы с асинхронным IO более интуитивно понятным и эффективным способом. Она построена используя библиотеку asyncio и предназначена для облегчения написания асинхронного кода разработчиками, который является надежным и масштабируемым.

С aiomisc вы можете воспользоваться такими мощными функциями, как: *worker пул*, *connection пул*, *шаблон «предохранитель»*, и механизмы повторов такие как *asyncbackoff* и *asyncretry* чтобы сделать ваш асинхронный код более надежным и простым в обслуживании. В этой документации мы более подробно рассмотрим, что может предложить `aiomisc` и как он может помочь вам упростить разработку сервисов с asyncio.

Глава 1

Установка

Возможна установка стандартными способами, такими как PyPI или установка из репозитория git напрямую.

Установка с PyPI:

```
pip3 install aiomisc
```

Установка из репозитория на github.com:

```
# Using git tool
pip3 install git+https://github.com/aiokitchen/aiomisc.git

# Alternative way using http
pip3 install \
    https://github.com/aiokitchen/aiomisc/archive/refs/heads/master.zip
```

Пакет содержит несколько дополнений, и вы можете установить дополнительные зависимости, если вы укажете их таким образом.

Вместе с uvloop

```
pip3 install "aiomisc[uvloop]"
```

Вместе с aiohttp:

```
pip3 install "aiomisc[aiohttp]"
```

Полная таблица дополнений ниже:

пример	описание
`pip install aiomisc[aiohttp]`	Для запуска приложений написанных с aiohttp.
`pip install aiomisc[asgi]`	Для запуска ASGI приложений
`pip install aiomisc[carbon]`	Чтобы посылать метрики в carbon (часть graphite)
`pip install aiomisc[cron]`	планирование задач с библиотекой croniter
`pip install aiomisc[raven]`	Чтобы посылать исключения в sentry используя raven
`pip install aiomisc[rich]`	Можете использовать rich для логирования
`pip install aiomisc[uvicorn]`	For running ASGI application using uvicorn
`pip install aiomisc[uvloop]`	используйте uvloop как основной event-loop

Вы можете комбинировать эти значения разделяя их запятыми, пример:

```
pip3 install "aiomisc[aiohttp,cron,rich,uvloop]"
```

Глава 2

Быстрый старт

В этом разделе будет рассказано, как эта библиотека создает и использует цикл обработки событий и создает службы. Конечно, обо всем тут не напишешь, но о многом можно прочитать в разделе *Учебник*, и всегда можно обратиться к разделу *Модули* и разделу *Описание API* для справки.

2.1 Eventloop и entrypoint

Сначала рассмотрим этот простой пример:

```
import asyncio
import logging

import aiomisc

log = logging.getLogger(__name__)

async def main():
    log.info('Starting')
    await asyncio.sleep(3)
    log.info('Exiting')


if __name__ == '__main__':
    with aiomisc.entrypoint(log_level="info", log_format="color") as loop:
        loop.run_until_complete(main())
```

Этот код объявляет асинхронную функцию `main()`, которая завершается через 3 секунды. Казалось бы, ничего интересного, но все дело в `entrypoint`.

Что делает `entrypoint`, казалось бы не так уж и много, она создает event-loop и передает управление пользователю. Однако под капотом настраивается журналирование в отдельном потоке, создается пул потоков, запускаются сервисы, но об этом позже и сервисов в данном примере нет.

В принципе вы можете не использовать точку входа, а просто создать eventloop и установите его по умолчанию для текущего потока:

```
import asyncio
import aiomisc

# * Installs uvloop event loop is it's has been installed.
# * Creates and set `aiomisc.thread_pool.ThreadPoolExecutor`
#   as a default executor
# * Sets just created event-loop as a current event-loop for this thread.
aiomisc.new_event_loop()

async def main():
    await asyncio.sleep(1)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Приведенный выше пример полезен, если в вашем коде уже неявно используется созданный eventloop, тогда вам придется изменить меньше кода, просто добавьте `aiomisc.new_event_loop()` и все вызовы `asyncio.get_event_loop()` вернет созданный экземпляр.

Однако можно обойтись и одним вызовом. Следующий пример закрывает неявно созданный eventloop asyncio и устанавливает новый:

```
import asyncio
import aiomisc

async def main():
    await asyncio.sleep(3)

if __name__ == '__main__':
    loop = aiomisc.new_event_loop()
    loop.run_until_complete(main())
```

2.2 Сервисы

Главное, что делает точка входа, — это запускает и корректно останавливает «Сервисы».

Концепция «Сервис», в этой библиотеке, означает класс, наследованный от класса `aiosmic.Service` и реализующий метод `async def start(self) -> None` и, опционально, метод `async def stop(self, exc: Optional[ Exception]) -> None`.

Концепция остановки службы не обязательно заключается в нажатии пользователем клавиш `Ctrl+C`, на самом деле это просто выход из контекстного менеджера `entrypoint`.

Пример ниже иллюстрирует, как может выглядеть ваш сервис:

```
from aiomisc import entrypoint, Service

class MyService(Service):
    async def start(self):
        do_something_when_start()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    async def stop(self, exc):
        do_graceful_shutdown()


with entrypoint(MyService()) as loop:
    loop.run_forever()
```

Точка входа может запускать любое количество экземпляров службы, и все они будут запускаться конкурентно.

Также есть способ, если метод `start` является полезной нагрузкой для сервиса, и тогда нет необходимости реализовывать метод `stop`, так как задача с функцией `start` будет отменена на этапе выхода из entrypoint. Но в этом случае вам придется уведомить `entrypoint` о том, что инициализация экземпляра службы завершена и ее можно продолжить.

Примерно так:

```
import asyncio
from threading import Event
from aiomisc import entrypoint, Service

event = Event()

class MyService(Service):
    async def start(self):
        # Send signal to entrypoint for continue running
        self.start_event.set()
        await asyncio.sleep(3600)


with entrypoint(MyService()) as loop:
    assert event.is_set()
```

Примечание: `entrypoint` передает управление телу контекстного менеджера только после того, как все экземпляры службы запущены. Как упоминалось выше, стартом считается завершение метода `start` или установка стартового события с помощью `self.start_event.set()`.

Вся мощь этой библиотеки это набор уже реализованных или абстрактных сервисов таких как: *AIOHTTPService*, *ASGIService*, *TCPServer*, *UDPServer*, *TCPClient*, *PeriodicService*, *CronService* и так далее.

К сожалению в данном разделе нет возможности уделить этому больше внимания, обратите внимание на раздел *Учебник*, там больше примеров и пояснений, ну и конечно вы всегда можете узнать ответ на *Описание API* или в исходном коде. Авторы постарались сделать исходный код максимально понятным и простым, поэтому не стесняйтесь исследовать его.

Глава 3

Версионирование

Это программное обеспечение следует методологиии Семантического Версионирования

Кратко: учитывая номер версии МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ, следует увеличивать:

- МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
- МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
- ПАТЧ-версию, когда вы делаете обратно совместимые исправления.
- Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.

В этом проекте версия пакета назначается автоматически с помощью poem-plugins, он использует тег в репозитории как МАЖОР и МИНОР, а также счетчик, который берет количество коммитов между тегом и головой ветки.

Глава 4

Как разрабатывать этот проект?

Этот проект, как и многие другие open source проекты, разрабатывается энтузиастами, и вы можете присоединиться к разработке, создавайте issues в github или присылайте свои правки как merge request.

Чтобы начать разработку в этом репозитории, вам необходимо сделать следующее:

Должно быть установлено

- Python 3.7+ как `python3`
- Установлен Poetry как `poetry`

Для настройки окружения разработчика выполните:

```
# installing all dependencies
poetry install

# setting up pre-commit hooks
poetry run pre-commit install

# adding poem-plugins to the poetry
poetry self add poem-plugins
```

4.1 Учебник

`aiomisc` — это библиотека Python, которая предоставляет набор утилит для создание асинхронных сервисов. Это позволяет вам разделить вашу программу на небольшие независимые службы, которые могут работать одновременно, улучшая общую производительность и масштабируемость вашего приложения.

Основным подходом в этой библиотеке является разделение вашей программы на независимые сервисы, которые могут работать конкурентно в асинхронном режиме. Библиотека также предоставляет набор готовых к использованию сервисов с заранее написанной логикой запуска и остановки.

Подавляющее большинство функций и классов написаны таким образом, что их можно использовать в программе, которая изначально не была разработана в соответствии с принципами, изложенными в этом руководстве. Так что если вы не планируете слишком сильно переписывать свой код, а хотите использовать только несколько полезных для вас функций или классов, то все должно работать.

В целом, aiomisc — это мощный инструмент для разработчиков, стремящихся создавать эффективные и масштабируемые асинхронные сервисы на Python.

4.1.1 Сервисы

Если вы хотите запустить tcp или веб-сервер, вам придется написать что-то вроде этого:

```
import asyncio

async def example_async_func():
    # Делаем что-то асинхронное и очень важное
    await init_db()
    await init_cache()
    await start_http_server()
    await start_metrics_server()

loop = asyncio.get_event_loop()
loop.run_until_complete(example_async_func())
# Продолжаем выполнение асинхронных задач
loop.run_forever()
```

Для того, чтобы запустить или остановить асинхронные программы, обычно используется функция `asyncio.run(example_async_func())`, которая доступна начиная с Python 3.7. Функция принимает экземпляр сопрограммы и отменяет через *cancel()* всем задачи перед тем как вернуть результат. Чтобы продолжить выполнение кода вечно, вы можете применить следующий трюк:

```
import asyncio

async def example_async_func():
    # Делаем что-то асинхронное и очень важное
    await init_db()
    await init_cache()
    await start_http_server()
    await start_metrics_server()

    # Создаем future которая никогда не будет завершена
    # используется вместо loop.run_forever()
    await asyncio.Future()

asyncio.run(example_async_func())
```

Если пользователь нажимает *Ctrl+C*, программа просто завершается, но если вы хотите явно освободить некоторые ресурсы, например, закрыть соединения с базой данных или откатить незавершенные транзакции, то вам нужно сделать что-то вроде этого:

```
import asyncio

async def example_async_func():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    try:
        # Делаем что-то асинхронное и очень важное
        await init_db()
        await init_cache()
        await start_http_server()
        await start_metrics_server()

        # Создаем future которая никогда не будет завершена
        # используется вместо loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
        # Зайдем в этот блок будет когда получим SIGTERM
        pass
    finally:
        # Зайдем сюда перед выходом
        ...

asyncio.run(example_async_func())
```

Это хорошее решение, потому что оно реализовано без каких-либо сторонних библиотек. Но когда ваша программа начнет расти, вы, вероятно, захотите оптимизировать время запуска простым способом, а именно выполнить всю инициализацию конкурентно. На первый взгляд кажется, что этот код решит проблему:

```
import asyncio

async def example_async_func():
    try:
        # Делаем что-то асинхронное и очень важное
        await asyncio.gather(
            init_db(),
            init_cache(),
            start_http_server(),
            start_metrics_server(),
        )

        # Создаем future которая никогда не будет завершена
        # используется вместо loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
        # Зайдем в этот блок будет когда получим SIGTERM
        pass
    finally:
        # Зайдем сюда перед выходом
        ...

asyncio.run(example_async_func())
```

Но если вдруг какая-то часть инициализации пойдет не по плану, то вам каким-то образом придется выяснить, что именно пошло не так. Поэтому при конкурентном выполнении код уже не будет таким простым, как в этом примере.

И для того, чтобы как-то упорядочить код, вы должны сделать отдельную функцию, которая будет

содержать блок `try/except/finally` и содержать обработку ошибок.

```
import asyncio

async def init_db():
    try:
        # инициализируем соединение
    finally:
        # закрываем соединение
        ...

async def example_async_func():
    try:
        # Делаем что-то асинхронное и очень важное
        await asyncio.gather(
            init_db(),
            init_cache(),
            start_http_server(),
            start_metrics_server(),
        )

        # Создаем future которая никогда не будет завершена
        # используется вместо loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
        # Зайдем в этот блок будет когда получим SIGTERM
        # TODO: нужно как-то корректно все завершить
        pass
    finally:
        # Зайдем сюда перед выходом
        ...

asyncio.run(example_async_func())
```

И теперь, если пользователь нажимает Ctrl+C, вам нужно снова описать логику завершения работы, но уже в блоке `except`.

Для того, чтобы описать логику запуска и остановки в одном месте, а также тестирования одним-единственным способом, и существует абстракция `Service`.

Сервис представляет из себя абстрактный базовый класс, в котором нужно реализовать метод `start()` и не обязательно метод `stop()`

Сервис может работать в двух режимах. Первый это когда метод старт выполняется вечно, тогда не нужно реализовывать стоп, но нужно сообщить что инициализация успешно закончена с помощью `self.start_event.set()`.

```
import asyncio
import aiomisc


class InfinityService(aiomisc.Service):
    async def start(self):
        # Сервис готов работать
        self.start_event.set()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        while True:
            # Делаем что-то полезное
            await asyncio.sleep(1)
```

В этом случае остановка сервиса будет заключаться в завершении сопрограммы которую породила `start()`

Второй способ это явное описание способа запуска и остановки реализовав методы `start()` и `stop()`

```
import asyncio
import aiomisc
from typing import Any


class OrdinaryService(aiomisc.Service):
    async def start(self):
        # Делаем что-то полезное
        ...

    async def stop(self, exception: Exception = None) -> Any:
        # Делаем что-то полезное
        ...
```

В этом случае запуск и остановка сервиса будут выполнены однократно.

4.1.2 Конфигурация сервисов

Так как `Service` это метакласс, он может обрабатывать специальные аттрибуты классов наследуемых от него на этапе объявления класса.

Вот простой императивный пример как инициализация сервиса может быть расширена через наследование.

```
from typing import Any
import aiomisc

class HelloService(aiomisc.Service):
    def __init__(self, name: str = "мир", **kwargs: Any):
        super().__init__(**kwargs)
        self.name = name

    async def start(self) -> Any:
        print(f"Привет {self.name}")

with aiomisc.entrypoint(
    HelloService(),
    HelloService(name="Гвидо")
) as loop:
    pass

# python hello.py
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# <<< Привет мир
# <<< Привет Гвидо
```

На самом деле, можно было ничего и не делать, так как метакласс установит все переданные именованные параметры в self по умолчанию.

```
import aiomisc

class HelloService(aiomisc.Service):
    name: str = "мир"

    async def start(self):
        print(f"Привет {self.name}")

with aiomisc.entrypoint(
    HelloService(),
    HelloService(name="Гвидо")
) as loop:
    pass

# python hello.py
# <<< Привет мир
# <<< Привет Гвидо
```

Если-же объявлен специальный аттрибут `__required__`, сервис будет требовать чтобы он был передан явно при инициализации как именованный параметр.

```
import aiomisc

class HelloService(aiomisc.Service):
    __required__ = ("name", "title")

    name: str
    title: str

    async def start(self):
        await asyncio.sleep(0.1)
        print(f"Привет {self.title} {self.name}")

with aiomisc.entrypoint(
    HelloService(name="Гвидо", title="мистер")
) as loop:
    pass
```

Также очень полезный специальным атрибут класса это `__async_required__`. В общем, это полезно для написания базовых классов. Он содержит кортеж имен методов, которые должны быть явно объявлены асинхронными (через `async def`).

```
import aiomisc

class HelloService(aiomisc.Service):
    __required__ = ("name", "title")
    __async_required__ = ("greeting",)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    name: str
    title: str

    async def greeting(self) -> str:
        await asyncio.sleep(0.1)
        return f"Привет {self.title} {self.name}"

    async def start(self):
        print(await self.greeting())

class HelloEmojiService(HelloService):
    async def greeting(self) -> str:
        await asyncio.sleep(0.1)
        return f" {self.title} {self.name}"

with aiomisc.entrypoint(
    HelloService(name="Гвидо", title="мистер"),
    HelloEmojiService(name="", title="")
) as loop:
    pass

# Привет мистер Гвидо
#
```

Если наследник объявит эти методы по-другому, будет ошибка на этапе объявления класса.

```
class BadHello(HelloService):
    def greeting(self) -> str:
        return f"{self.title} {self.name}"

#Traceback (most recent call last):
#...
#TypeError: ('Following methods must be coroutine functions', ('BadHello.greeting',))
```

4.1.3 dependency injection

В некоторых случаях перед запуском службы необходимо выполнить некоторый асинхронный код, например, чтобы передать соединение с базой данных экземпляру сервиса. Или если вы хотите использовать один экземпляр какой-то сущности для нескольких сервисов.

Для таких сложных конфигураций существует плагин aiomisc-dependency, который распространяется как независимый отдельный пакет.

Посмотрите на примеры в документации, aiomisc-dependency прозрачно интегрируется с `entrypoint`.

4.1.4 entrypoint

Итак сервисы описаны, что дальше? `asyncio.run` не умеет с ними работать, вызывать их вручную не стало проще, что тут можно предложить?

Наверное самый магический, сложный, и в то-же время достаточно хорошо протестированный код в библиотеке - это `entrypoint`. Изначально идеей `entrypoint` было избавление от рутины: настройка логов, настройка пула потоков, ну и запуск и корректная остановка сервисов.

Давайте посмотрим на пример:

```
import asyncio
import aiomisc

...

with aiomisc.entrypoint(
    OrdinaryService(),
    InfinityService()
) as loop:
    loop.run_forever()
```

В этом примере мы запускаем два сервиса, описанных выше, и продолжаем выполнение до тех пор пока пользователь его не прервёт. Далее, благодаря контекстному менеджеру, мы корректно завершаем все экземпляры сервисов.

Примечание: Entrypoint вызывает все методы `start()` во всех сервисах конкурентно, и если хотя-бы один из них упадет, все остальные сервисы будут остановлены.

Я упоминал, что я хотел убрать много рутины, давайте посмотрим на тот же пример, только передадим явно все параметры по умолчанию в `entrypoint`.

```
import asyncio
import aiomisc


...

with aiomisc.entrypoint(
    OrdinaryService(),
    InfinityService(),
    pool_size=4,
    log_level="info",
    log_format="color",
    log_buffering=True,
    log_buffer_size=1024,
    log_flush_interval=0.2,
    log_config=True,
    policy=asyncio.DefaultEventLoopPolicy(),
    debug=False
) as loop:
    loop.run_forever()
```

Давайте не будем останавливаться на том, что делает каждый параметр. Но в целом, `entrypoint`

создала цикл событий, пул из четырех потоков, настроила его для текущего цикла событий, настроила логгер с «цветным» буферизованным выводом и запустила два сервиса.

Также вы можете запустить `entrypoint` без сервисов, просто сконфигурировать логирование и прочее:

```
import asyncio
import logging
import aiomisc


async def sleep_and_exit():
    logging.info("Started")
    await asyncio.sleep(1)
    logging.info("Exiting")


with aiomisc.entrypoint(log_level="info") as loop:
    loop.run_until_complete(sleep_and_exit())
```

Еще стоит обратить внимание на `aiomisc.run` который похож по своему назначению на `asyncio.run` при этом поддерживает запуск и остановку сервисов и прочее.

```
import asyncio
import logging
import aiomisc


async def sleep_and_exit():
    logging.info("Started")
    await asyncio.sleep(1)
    logging.info("Exiting")


aiomisc.run(
    # первый аргумент
    # это основная корутина
    sleep_and_exit(),
    # Другие позиционные аргументы
    # это экземпляры сервисов
    OrdinaryService(),
    InfinityService(),
    # другие именованные аргументы
    # будут просто переданы в entrypoint
    log_level="info"
)
```

Примечание: Как я и упоминал ранее библиотека содержит большое количество готовых абстрактных сервисов, которые вы можете использовать в своем проекте, просто реализовав несколько методов.

Полный список сервисов и примеры их использования можно найти на странице *Сервисы*.

4.1.5 Выполнение кода в пуле потоков или процессов

Как объясняется в разделе working with threads в официальной документации по Python, eventloop в asyncio запускает пул потоков.

Этот пул нужен для того, чтобы запустить, например, разрешение имен и не блокировать eventloop, пока работает низкоуровневый вызов *gethostbyname*.

Размер этого пула потоков должен быть настроен при запуске приложения, иначе вы можете столкнуться со всевозможными проблемами, когда этот пул слишком велик или слишком мал.

По умолчанию `entrypoint` создает пул потоков с размером, равным количеству ядер процессора (минимум — 4, и максимум — 32). Конечно, вы можете указать столько, сколько вам нужно.

Декоратор `@aiomisc.threaded`

В разделе working with threads официальной документации по python даются следующие рекомендации по вызову блокирующих функций в потоках:

```
import asyncio

def blocking_io():
    # Файловые операции (такие как логирование) могут заблокировать event loop.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

async def main():
    loop = asyncio.get_running_loop()
    result = await loop.run_in_executor(None, blocking_io)

asyncio.run(main())
```

Эта библиотека содержит очень простой способ сделать тоже самое:

```
import aiomisc

@aiomisc.threaded
def blocking_io():
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

async def main():
    result = await blocking_io()

aiomisc.run(main())
```

Как видно из этого примера, достаточно обернуть функцию декоратором `aiomisc.threaded`, и она начнет возвращать awaitable объект, но код внутри функции будет отправлен в пул потоков по умолчанию.

Декоратор `@aiomisc.threaded_separate`

Если блокирующая функция работает долго, а то и бесконечно долго, иными словами, если стоимость создания потока незначительна по сравнению с сколько функция работает, то можно попробовать использовать декоратор `aiomisc.threaded_separate`.

Декоратор запускает новый поток, не связанный с каким-либо пулом потоков. Поток завершится после выхода из функции.

```
import hashlib
import aiomisc

@aiomisc.threaded_separate
def another_one_useless_coin_miner():
    with open('/dev/urandom', 'rb') as f:
        hasher = hashlib.sha256()
        while True:
            hasher.update(f.read(1024))
            if hasher.hexdigest().startswith("0000"):
                return hasher.hexdigest()

async def main():
    print(
        "Хэш получился вот такой:",
        await another_one_useless_coin_miner()
    )

aiomisc.run(main())
```

Примечание: Такой подход позволяет не занимать потоки в пуле надолго, но при этом никак не ограничивает количество создаваемых потоков.

Больше примеров можно найти на странице *Работа с потоками*

Декоратор `@aiomisc.threaded_iterable`

Если генератор нужно выполнить в потоке, возникают проблемы с синхронизацией потока и eventloop. Эта библиотека предоставляет пользовательский декоратор, предназначенный для превращения синхронного генератора в асинхронный.

Это очень полезно, если, например, драйвер очереди или базы данных синхронный, но вы хотите эффективно использовать его в асинхронном коде.

```
import aiomisc

@aiomisc.threaded_iterable(max_size=8)
def urandom_reader():
    with open('/dev/urandom', "rb") as fp:
        while True:
            yield fp.read(8)

async def main():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    counter = 0
    async for chunk in urandom_reader():
        print(chunk)
        counter += 1
        if counter > 16:
            break

aiomisc.run(main())
```

Под капотом этот декоратор возвращает специальный объект с очередью, а интерфейс асинхронного итератора обеспечивает доступ к этой очереди.

Всегда следует указывать параметр `max_size`, который ограничивает размер этой очереди и предотвращает отправку кода, который работает в потоке, слишком большого количества элементов в асинхронный код, в случае асинхронной итерации в случае, если асинхронный итератор забирает элементы из этой очереди реже чем они поступают.

Заключение

На этом нам нужно закончить этот учебник, надеюсь тут все было понятно, и вы почерпнули для себя много полезного. Полное описание остальных сервисов представлено в разделе *Модули*, или в исходном коде. Авторы постарались сделать исходный код максимально понятным и простым, поэтому не стесняйтесь исследовать его.

4.2 Модули

4.2.1 Точка входа (*entrypoint*)

В общем случае точка входа это сущность помогающая создать event loop и закрыть все еще запущенные корутины при выходе.

```
import asyncio
import aiomisc

async def main():
    await asyncio.sleep(1)

with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

Пример целиком:

```
import asyncio
import aiomisc
import logging

async def main():
    await asyncio.sleep(1)
    logging.info("Hello there")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
with aiomisc.entrypoint(
    pool_size=2,
    log_level='info',
    log_format='color',                             # по умолчанию если "rich" не␣
↪установлен
    log_buffer_size=1024,                           # по умолчанию
    log_flush_interval=0.2,                         # по умолчанию
    log_config=True,                                # по умолчанию
    policy=asyncio.DefaultEventLoopPolicy(),        # по умолчанию
    debug=False,                                    # по умолчанию
    catch_signals=(signal.SIGINT, signal.SIGTERM), # по умолчанию
    shutdown_timeout=60,                            # по умолчанию
) as loop:
    loop.run_until_complete(main())
```

Запуск точки входа (*entrypoint*) из асинхронного кода

```
import asyncio
import aiomisc
import logging
from aiomisc.service.periodic import PeriodicService

log = logging.getLogger(__name__)

class MyPeriodicService(PeriodicService):
    async def callback(self):
        log.info('Running periodic callback')
        # ...

async def main():
    service = MyPeriodicService(interval=1, delay=0)  # once per minute

    # вернет экземпляр entrypoint потому, что event-loop
    # уже запущен и может быть получен через asyncio.get_event_loop()
    async with aiomisc.entrypoint(service) as ep:
        try:
            await asyncio.wait_for(ep.closing(), timeout=1)
        except asyncio.TimeoutError:
            pass


asyncio.run(main())
```

Динамический запуск сервисов

Иногда бывает недостаточно добавить сервисы в точку входа на старте или нет возможности получить параметры сервиса до старта event-loop. В этом случае возможен запуск сервисов после запуска событийного цикла, эта функция доступна с версии 17.

```
import asyncio
import aiomisc
import logging

from aiomisc.service.periodic import PeriodicService

log = logging.getLogger(__name__)


class MyPeriodicService(PeriodicService):
    async def callback(self):
        log.info('Running periodic callback')


async def add_services():
    entrypoint = aiomisc.entrypoint.get_current()

    services = [
        MyPeriodicService(interval=2, delay=1),
        MyPeriodicService(interval=2, delay=0),
    ]

    await entrypoint.start_services(*services)
    await asyncio.sleep(10)
    await entrypoint.stop_services(*services)


with aiomisc.entrypoint() as loop:
    loop.create_task(add_services())
    loop.run_forever()
```

Конфигурация из переменных окружения

Модуль поддерживает конфигурацию из переменных окружения:

- `AIOMISC_LOG_LEVEL` - уровень логирования по умолчанию
- `AIOMISC_LOG_FORMAT` - формат логирования по умолчанию
- `AIOMISC_LOG_DATE_FORMAT` - формат дат в логах по умолчанию
- `AIOMISC_LOG_CONFIG` - следует ли настраивать логирование
- `AIOMISC_LOG_FLUSH` - интервал сброса буфера логов logs
- `AIOMISC_LOG_BUFFERING` - следует ли включать буфферизацию логирования
- `AIOMISC_LOG_BUFFER_SIZE` - максимальный размер буфера логов
- `AIOMISC_POOL_SIZE` - размер пула потоков

- `AIOMISC_USE_UVLOOP` - следует ли использовать uvloop, `0` чтобы отключить
- `AIOMISC_SHUTDOWN_TIMEOUT` - Если после получения сигнала программа не завершается в течение этого таймаута, происходит принудительный выход.

4.2.2 Функция `run()`

`aiomisc.run()` - это простой способ создать и разрушить `aiomisc.entrypoint`. Это очень похоже на `asyncio.run()` но управляет сервисами `aiomisc.Service` и принимает прочие аргументы entrypoint.

```
import asyncio
import aiomisc

async def main():
    loop = asyncio.get_event_loop()
    now = loop.time()
    await asyncio.sleep(0.1)
    assert now < loop.time()


aiomisc.run(main())
```

4.2.3 Конфигурация журналов

`entrypoint` принимает аргумент `log_format` с определенным набором форматов, в которых журналы будут записываться в stderr.

- `stream` - стандартный python логгер
- `color` - логирование через модуль `colorlog`
- `json` - json структура, одна на строчку
- `syslog` - `logging.handlers.SysLogHandler` из стандартной библиотеки
- `plain` - просто сообщения, без даты или информации об уровне логирования
- `journald` - доступно только если `logging-journald` модуль установлен.
- `rich`/`rich_tb` - доступно только если установлен модуль `rich`. `rich_tb` тоже самое что и `rich` только с подробными трейсбэками.

Также вы можете настроить уровень логирования параметром `log_level` и формат дат в логах параметром `log_date_format`

`entrypoint` вызовет `aiomisc.log.basic_config` неявно используя пеараметры `log_*=`. В качестве альтернативы, вы можете вызвать `aiomisc.log.basic_config` вручную передав ей экземпляр `eventloop`.

Однако вы можете настроить логирование раньше, используя `aiomisc_log.basic_config`, но вы потеряете буферизацию и запись в буфер отдельном потоке. Эта функция фактически вызывается во время настройки ведения журнала, `entrypoint` передает обертку для logging handler, чтобы он записывал в буфер в отдельном потоке.

```
import logging

from aiomisc_log import basic_config

```

(continues on next page)

(продолжение с предыдущей страницы)

```
basic_config(log_format="color")
logging.info("Hello")
```

Если вы хотите настроить ведение журнала перед запуском `entrypoint`, например, после разбора аргументов, это безопасно настроить его дважды (или больше).

```
import logging

import aiomisc
from aiomisc_log import basic_config


basic_config(log_format="color")
logging.info("Hello from usual python")


async def main():
    logging.info("Hello from async python")


with aiomisc.entrypoint(log_format="color") as loop:
    loop.run_until_complete(main())
```

Иногда вы хотите настроить ведение журнала самостоятельно, пример ниже демонстрирует, как это сделать:

```
import os
import logging
from logging.handlers import RotatingFileHandler
from gzip import GzipFile

import aiomisc


class GzipLogFile(GzipFile):
    def write(self, data) -> int:
        if isinstance(data, str):
            data = data.encode()
        return super().write(data)


class RotatingGzipFileHandler(RotatingFileHandler):
    """ Really added just for example you have to test it properly """

    def shouldRollover(self, record):
        if not os.path.isfile(self.baseFilename):
            return False
        if self.stream is None:
            self.stream = self._open()
        return 0 < self.maxBytes < os.stat(self.baseFilename).st_size
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def _open(self):
        return GzipLogFile(filename=self.baseFilename, mode=self.mode)


async def main():
    for _ in range(1_000):
        logging.info("Hello world")


with aiomisc.entrypoint(log_config=False) as loop:
    gzip_handler = RotatingGzipFileHandler(
        "app.log.gz",
        # Максимум 100 файлов по 10 мегабайт
        maxBytes=10 * 2 ** 20, backupCount=100
    )
    stream_handler = logging.StreamHandler()

    formatter = logging.Formatter(
        "[%(asctime)s] <%(levelname)s> "
        "%(filename)s:%(lineno)d (%(threadName)s): %(message)s"
    )

    gzip_handler.setFormatter(formatter)
    stream_handler.setFormatter(formatter)

    logging.basicConfig(
        level=logging.INFO,
        # Обертывание всех обработчиков в отдельные потоки не заблокирует
        # event-loop даже если gzip занимает много времени, чтобы открыть
        # файл.
        handlers=map(
            aiomisc.log.wrap_logging_handler,
            (gzip_handler, stream_handler)
        )
    )
    loop.run_until_complete(main())
```

4.2.4 Сервисы

`Сервисы` - это абстракция, помогающая организовать множество различных задач в одном процессе. Каждый сервис должен реализовывать обязательный метод `start()` и опционально метод stop ().

Экземпляр сервиса должен быть передан в «точку входа» (entrypoint) и будет запущен после создания event loop.

Примечание: Текущий event-loop будет установлен до вызова метода *start()*. Event loop будет установлен для этого потока.

Пожалуйста, избегайте использования `asyncio.get_event_loop()` явно внутри метода `start()`. Вместо этого используйте атрибут сервиса `self.loop`:

```
import asyncio
from threading import Event
from aiomisc import entrypoint, Service

event = Event()

class MyService(Service):
  async def start(self):
      # Отправляем сигнал в entrypoint что можно продолжать
      self.start_event.set()

      event.set()
      # Запуск чего-то полезного
      await asyncio.sleep(3600)


with entrypoint(MyService()) as loop:
    assert event.is_set()
```

Метод `start()` создается как отдельная задача, которая может выполняться бесконечно. Но в этом случае необходимо утановить событие вызовом `self.start_event.set()` для уведомления entrypoint об окончании запуска.

Во время завершения работы сначала будет вызван метод `stop()`, а после этого все запущенные задачи будут отменены (включая `start()`).

Этот пакет содержит несколько полезных базовых классов для написания простых сервисов.

Класс `TCPServer`

`TCPServer` - это базовый класс для реализации TCP сервера. Просто реализуйте `handle_client(reader, writer)`, чтобы принимать TCP соединения.

```
import asyncio
import logging
from aiomisc import entrypoint
from aiomisc.service import TCPServer


log = logging.getLogger(__name__)


class EchoServer(TCPServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while not reader.at_eof():
            writer.write(await reader.read(255))

        log.info("Client connection closed")


async def echo_client(host, port):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    reader, writer = await asyncio.open_connection(host=host, port=port)
    writer.write(b"hello\n")
    assert await reader.readline() == b"hello\n"

    writer.write(b"world\n")
    assert await reader.readline() == b"world\n"

    writer.close()
    await writer.wait_closed()


with entrypoint(
    EchoServer(address='localhost', port=8901),
) as loop:
    loop.run_until_complete(echo_client("localhost", 8901))
```

Класс `UDPServer`

`UDPServer` - это базовый класс для реализации UDP сервера. Просто реализуйте `handle_datagram(data, addr)`, чтобы принимать UDP соединения.

```
class UDPPrinter(UDPServer):
    async def handle_datagram(self, data: bytes, addr):
        print(addr, '->', data)


with entrypoint(UDPPrinter(address='localhost', port=3000)) as loop:
    loop.run_forever()
```

Класс `TLSServer`

TLSServer - это базовый класс для написания TCP-серверов с использованием TLS. Просто реализуйте `handle_client(reader, writer)`.

```
class SecureEchoServer(TLSServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while True:
            writer.write(await reader.readline())

service = SecureEchoServer(
    address='localhost',
    port=8900,
    ca='ca.pem',
    cert='cert.pem',
    key='key.pem',
    verify=False,
)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
with entrypoint(service) as loop:
    loop.run_forever()
```

TCPClient

`TCPServer` - это базовый класс для реализации TCP сервера. Просто реализуйте `handle_client(reader, writer)`, чтобы принимать TCP соединения.

```
import asyncio
import logging
from aiomisc import entrypoint
from aiomisc.service import TCPServer, TCPClient

log = logging.getLogger(__name__)


class EchoServer(TCPServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while not reader.at_eof():
            writer.write(await reader.read(255))

        log.info("Client connection closed")


class EchoClient(TCPClient):

    async def handle_connection(self, reader: asyncio.StreamReader,
                                writer: asyncio.StreamWriter) -> None:
        writer.write(b"hello\n")
        assert await reader.readline() == b"hello\n"

        writer.write(b"world\n")
        assert await reader.readline() == b"world\n"

        writer.write_eof()
        writer.close()
        await writer.wait_closed()


with entrypoint(
    EchoServer(address='localhost', port=8901),
    EchoClient(address='localhost', port=8901),
) as loop:
    loop.run_until_complete(asyncio.sleep(0.1))
```

TLSClient

TCPServer - это базовый класс для реализации TCP сервера. Просто реализуйте handle_client(reader, writer), чтобы принимать TCP соединения.

```
import asyncio
import logging
from aiomisc import entrypoint
from aiomisc.service import TCPServer, TCPClient

log = logging.getLogger(__name__)


class EchoServer(TLSServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while not reader.at_eof():
            writer.write(await reader.read(255))

        log.info("Client connection closed")


class EchoClient(TLSClient):

    async def handle_connection(self, reader: asyncio.StreamReader,
                                writer: asyncio.StreamWriter) -> None:
        writer.write(b"hello\n")
        assert await reader.readline() == b"hello\n"

        writer.write(b"world\n")
        assert await reader.readline() == b"world\n"

        writer.write_eof()
        writer.close()
        await writer.wait_closed()


with entrypoint(
    EchoServer(
        address='localhost', port=8901,
        ca='ca.pem',
        cert='server.pem',
        key='server.key',
    ),
    EchoClient(
        address='localhost', port=8901,
        ca='ca.pem',
        cert='client.pem',
        key='client.key',
    ),
) as loop:
    loop.run_until_complete(asyncio.sleep(0.1))
```

RobustTCPClient

TLSServer - это базовый класс для написания TCP-серверов с использованием TLS. Просто реализуйте `handle_client(reader, writer)`.

```
import asyncio
import logging
from aiomisc import entrypoint
from aiomisc.service import TCPServer, RobustTCPClient

log = logging.getLogger(__name__)


class EchoServer(TCPServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while not reader.at_eof():
            writer.write(await reader.read(255))

        log.info("Client connection closed")


class EchoClient(RobustTCPClient):

    async def handle_connection(self, reader: asyncio.StreamReader,
                                writer: asyncio.StreamWriter) -> None:
        writer.write(b"hello\n")
        assert await reader.readline() == b"hello\n"

        writer.write(b"world\n")
        assert await reader.readline() == b"world\n"

        writer.write_eof()
        writer.close()
        await writer.wait_closed()


with entrypoint(
    EchoServer(address='localhost', port=8901),
    EchoClient(address='localhost', port=8901),
) as loop:
    loop.run_until_complete(asyncio.sleep(0.1))
```

RobustTLSClient

TLSServer - это базовый класс для написания TCP-серверов с использованием TLS. Просто реализуйте `handle_client(reader, writer)`.

```
import asyncio
import logging
from aiomisc import entrypoint
from aiomisc.service import TCPServer, RobustTCPClient
```

(continues on next page)

(продолжение с предыдущей страницы)

```
log = logging.getLogger(__name__)


class EchoServer(TLSServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while not reader.at_eof():
            writer.write(await reader.read(255))

        log.info("Client connection closed")


class EchoClient(RobustTLSClient):

    async def handle_connection(self, reader: asyncio.StreamReader,
                                writer: asyncio.StreamWriter) -> None:
        writer.write(b"hello\n")
        assert await reader.readline() == b"hello\n"

        writer.write(b"world\n")
        assert await reader.readline() == b"world\n"

        writer.write_eof()
        writer.close()
        await writer.wait_closed()


with entrypoint(
    EchoServer(
        address='localhost', port=8901,
        ca='ca.pem',
        cert='server.pem',
        key='server.key',
    ),
    EchoClient(
        address='localhost', port=8901,
        ca='ca.pem',
        cert='client.pem',
        key='client.key',
    ),
) as loop:
    loop.run_until_complete(asyncio.sleep(0.1))
```

Класс PeriodicService

PeriodicService запускает PeriodicCallback как сервис и ожидает завершения обратного вызова при остановке. Вам необходимо использовать PeriodicService в качестве базового класса и переопределить асинхронный метод callback.

Сервисный класс принимает обязательный аргумент `interval` - интервал запуска в секундах и необязательный аргумент `delay` - задержку первого выполнения в секундах (по умолчанию 0).

```
import aiomisc
from aiomisc.service.periodic import PeriodicService


class MyPeriodicService(PeriodicService):
    async def callback(self):
        log.info('Running periodic callback')
        # ...

service = MyPeriodicService(interval=3600, delay=0)  # раз в час

with entrypoint(service) as loop:
    loop.run_forever()
```

Класс CronService

`CronService` запускает `CronCallback` в качестве сервиса и ожидает завершения выполнения обратных вызовов при остановке.

Основан на croniter. Вы можете зарегистрировать асинхронный метод с аргументом `spec` - формат, подобный cron:

Предупреждение: необходимо установить библиотеку croniter:

```
pip install croniter
```

или как дополнительную зависимость

```
pip install aiomisc[cron]
```

```
import aiomisc
from aiomisc.service.cron import CronService


async def callback():
    log.info('Running cron callback')
    # ...

service = CronService()
service.register(callback, spec="0 * * * *") # каждый час в 0 минут minutes

with entrypoint(service) as loop:
    loop.run_forever()
```

Вы также можете наследовать от `CronService`, но помните, что регистрация обратного вызова должна выполняться до запуска

```
import aiomisc
from aiomisc.service.cron import CronService


class MyCronService(CronService):
    async def callback(self):
        log.info('Running cron callback')
        # ...

    async def start(self):
        self.register(self.callback, spec="0 * * * *")
        await super().start()

service = MyCronService()

with entrypoint(service) as loop:
    loop.run_forever()
```

Несколько сервисов

Передайте несколько экземпляров сервиса в `entrypoint`, чтобы запустить их все сразу. После выхода экземпляры сервиса точки входа будут корректно закрыты вызовом метода `stop()` или через отмену метода `start()`.

```
import asyncio
from aiomisc import entrypoint
from aiomisc.service import Service, TCPServer, UDPServer


class LoggingService(PeriodicService):
    async def callabck(self):
        print('Hello from service', self.name)


class EchoServer(TCPServer):
    async def handle_client(self, reader: asyncio.StreamReader,
                            writer: asyncio.StreamWriter):
        while True:
            writer.write(await reader.readline())


class UDPPrinter(UDPServer):
    async def handle_datagram(self, data: bytes, addr):
        print(addr, '->', data)


services = (
    LoggingService(name='#1', interval=1),
    EchoServer(address='localhost', port=8901),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    UDPPrinter(address='localhost', port=3000),
)


with entrypoint(*services) as loop:
    loop.run_forever()
```

Конфигурация

Метакласс `Service` принимает все именованные аргументы в `__init__` и устанавливает из как атрибуты в `self`.

```
import asyncio
from aiomisc import entrypoint
from aiomisc.service import Service, TCPServer, UDPServer


class LoggingService(Service):
    # обязательные именованные аргументы
    __required__ = frozenset({'name'})

    # default value
    delay: int = 1

    async def start(self):
        self.start_event.set()
        while True:
            # аттрибут ``name`` из именованных аргументов
            # должен быть передан при создании экземпляра
            print('Hello from service', self.name)

            # аттрибут ``delay`` из именованных аргументов
            await asyncio.sleep(self.delay)

services = (
    LoggingService(name='#1'),
    LoggingService(name='#2', delay=3),
)


with entrypoint(*services) as loop:
    loop.run_forever()
```

aiohttp сервис

Предупреждение: требуется установленная библиотека aiohttp

```
pip install aiohttp
```

или как дополнительную зависимость

```
pip install aiomisc[aiohttp]
```

Приложение aiohttp может быть запущено как сервис:

```
import aiohttp.web
import argparse
from aiomisc import entrypoint
from aiomisc.service.aiohttp import AIOHTTPService

parser = argparse.ArgumentParser()
group = parser.add_argument_group('HTTP options')

group.add_argument("-l", "--address", default="::",
                   help="Listen HTTP address")
group.add_argument("-p", "--port", type=int, default=8080,
                   help="Listen HTTP port")


async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return aiohttp.web.Response(text=text)


class REST(AIOHTTPService):
    async def create_application(self):
        app = aiohttp.web.Application()

        app.add_routes([
            aiohttp.web.get('/', handle),
            aiohttp.web.get('/{name}', handle)
        ])

        return app

arguments = parser.parse_args()
service = REST(address=arguments.address, port=arguments.port)

with entrypoint(service) as loop:
    loop.run_forever()
```

Класс `AIOHTTPSSLService` похож на `AIOHTTPService`, но создает HTTPS сервер. Вы должны передать требуемые для SSL параметры (см. Класс TLSServer).

asgi сервис

Предупреждение: требуется установленная библиотека aiohttp-asgi:

```
pip install aiohttp-asgi
```

или как дополнительную зависимость

```
pip install aiomisc[asgi]
```

Любое ASGI совместимое приложение может быть запущено как сервис:

```
import argparse

from fastapi import FastAPI

from aiomisc import entrypoint
from aiomisc.service.asgi import ASGIHTTPService, ASGIApplicationType

parser = argparse.ArgumentParser()
group = parser.add_argument_group('HTTP options')

group.add_argument("-l", "--address", default="::",
                   help="Listen HTTP address")
group.add_argument("-p", "--port", type=int, default=8080,
                   help="Listen HTTP port")


app = FastAPI()


@app.get("/")
async def root():
    return {"message": "Hello World"}


class REST(ASGIHTTPService):
    async def create_asgi_app(self) -> ASGIApplicationType:
        return app


arguments = parser.parse_args()
service = REST(address=arguments.address, port=arguments.port)

with entrypoint(service) as loop:
    loop.run_forever()
```

Класс `ASGIHTTPSSLService` похож на `ASGIHTTPService`, но создает HTTPS сервер. Вы должны передать требуемые для SSL параметры (см. Класс `TLSServer`).

uvicorn service

Предупреждение: requires installed uvicorn:

```
pip install uvicorn
```

или как дополнительную зависимость

```
pip install aiomisc[uvicorn]
```

Any ASGI-like application can be started via uvicorn as a service:

```
import argparse

from fastapi import FastAPI

from aiomisc import entrypoint
from aiomisc.service.uvicorn import UvicornApplication, UvicornService

parser = argparse.ArgumentParser()
group = parser.add_argument_group('HTTP options')

group.add_argument("-l", "--host", default="::",
                   help="Listen HTTP host")
group.add_argument("-p", "--port", type=int, default=8080,
                   help="Listen HTTP port")


app = FastAPI()


@app.get("/")
async def root():
    return {"message": "Hello World"}


class REST(UvicornService):
    async def create_application(self) -> UvicornApplication:
        return app


arguments = parser.parse_args()
service = REST(host=arguments.host, port=arguments.port)

with entrypoint(service) as loop:
    loop.run_forever()
```

GRPC service

Это пример GRPC-сервиса, который определяется в файле и загружает файл *hello.proto* без кодогенерации, этот пример является одним из примеров из *grpcio*, остальные примеры будут работать как ожидается.

Определение proto файла

```
syntax = "proto3";

package helloworld;

// Определение сервиса приветствия.
service Greeter {
  // Сказать привет
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// Сообщение запроса, содержащее имя пользователя.
message HelloRequest {
  string name = 1;
}

// Ответное сообщение, содержащее приветствие
message HelloReply {
  string message = 1;
}
```

Пример инициализации сервиса:

```
import grpc

import aiomisc
from aiomisc.service.grpc_server import GRPCService


protos, services = grpc.protos_and_services("hello.proto")


class Greeter(services.GreeterServicer):
    async def SayHello(self, request, context):
        return protos.HelloReply(message='Hello, %s!' % request.name)


def main():
    grpc_service = GRPCService(compression=grpc.Compression.Gzip)
    services.add_GreeterServicer_to_server(
        Greeter(), grpc_service,
    )
    grpc_service.add_insecure_port('[::]:0')
    grpc_service.add_insecure_port('[::1]:0')
    grpc_service.add_insecure_port('127.0.0.1:0')
    grpc_service.add_insecure_port('localhost:0')
    grpc_service.add_secure_port('localhost:0', grpc.local_server_credentials())
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    grpc_service.add_secure_port('[::]:0', grpc.local_server_credentials())

    with aiomisc.entrypoint(grpc_service) as loop:
        loop.run_forever()


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

To enable reflection for the service you use reflection flag:

```
GRPCService(reflection=True)
```

Трассировщик памяти

Простой и полезный сервис для логирования больших объектов Python, размещенных в памяти.

```
import asyncio
import os
from aiomisc import entrypoint
from aiomisc.service import MemoryTracer


async def main():
    leaking = []

    while True:
        leaking.append(os.urandom(128))
        await asyncio.sleep(0)


with entrypoint(MemoryTracer(interval=1, top_results=5)) as loop:
    loop.run_until_complete(main())
```

Пример вывода:

```
[T:[1] Thread Pool] INFO:aiomisc.service.tracer: Top memory usage:
 Objects | Obj.Diff |   Memory | Mem.Diff | Traceback
      12 |       12 |   1.9KiB |   1.9KiB | aiomisc/periodic.py:40
      12 |       12 |   1.8KiB |   1.8KiB | aiomisc/entrypoint.py:93
       6 |        6 |   1.1KiB |   1.1KiB | aiomisc/thread_pool.py:71
       2 |        2 |   976.0B |   976.0B | aiomisc/thread_pool.py:44
       5 |        5 |   712.0B |   712.0B | aiomisc/thread_pool.py:52

[T:[6] Thread Pool] INFO:aiomisc.service.tracer: Top memory usage:
 Objects | Obj.Diff |   Memory | Mem.Diff | Traceback
   43999 |    43999 |   7.1MiB |   7.1MiB | scratches/scratch_8.py:11
      47 |       47 |   4.7KiB |   4.7KiB | env/bin/../lib/python3.7/abc.py:143
      33 |       33 |   2.8KiB |   2.8KiB | 3.7/lib/python3.7/tracemalloc.py:113
      44 |       44 |   2.4KiB |   2.4KiB | 3.7/lib/python3.7/tracemalloc.py:185
      14 |       14 |   2.4KiB |   2.4KiB | aiomisc/periodic.py:40
```

`Profiler` - профилировщик

Простой сервис для профилирования. Необязательный аргумент *path* может быть предоставлен для выгрузки полных данных профилирования, которые позже могут быть использованы, например, snakeviz. Также можно изменить порядок с аргументом `order` (по умолчанию «cumulative»).

```
import asyncio
import os
from aiomisc import entrypoint
from aiomisc.service import Profiler


async def main():
    for i in range(100):
        time.sleep(0.01)


with entrypoint(Profiler(interval=0.1, top_results=5)) as loop:
    loop.run_until_complete(main())
```

Пример вывода:

```
108 function calls in 1.117 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100    1.117    0.011    1.117    0.011 {built-in method time.sleep}
     1    0.000    0.000    0.000    0.000 <...>/lib/python3.7/pstats.py:89(__init__)
     1    0.000    0.000    0.000    0.000 <...>/lib/python3.7/pstats.py:99(init)
     1    0.000    0.000    0.000    0.000 <...>/lib/python3.7/pstats.py:118(load_stats)
     1    0.000    0.000    0.000    0.000 <...>/lib/python3.7/cProfile.py:50(create_
↪stats)
```

Raven сервис

Простой сервис для отправки необработанных исключений в сервис sentry.

Простой пример:

```
import asyncio
import logging
import sys

from aiomisc import entrypoint
from aiomisc.version import __version__
from aiomisc.service.raven import RavenSender


async def main():
    while True:
        await asyncio.sleep(1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        try:
            1 / 0
        except ZeroDivisionError:
            logging.exception("Exception")


raven_sender = RavenSender(
    sentry_dsn=(
        "https://583ca3b555054f80873e751e8139e22a@o429974.ingest.sentry.io/"
        "5530251"
    ),
    client_options=dict(
        # По умолчанию возьмет значение переменной окружения SENTRY_NAME
        name="example-from-aiomisc",
        # По умолчанию возьмет значение переменной окружения SENTRY_ENVIRONMENT
        environment="simple_example",
        # По умолчанию возьмет значение переменной окружения SENTRY_RELEASE
        release=__version__,
    )
)


with entrypoint(raven_sender) as loop:
    loop.run_until_complete(main())
```

Все опции для клиента:

```
import asyncio
import logging
import sys

from aiomisc import entrypoint
from aiomisc.version import __version__
from aiomisc.service.raven import RavenSender


async def main():
    while True:
        await asyncio.sleep(1)

        try:
            1 / 0
        except ZeroDivisionError:
            logging.exception("Exception")


raven_sender = RavenSender(
    sentry_dsn=(
        "https://583ca3b555054f80873e751e8139e22a@o429974.ingest.sentry.io/"
        "5530251"
    ),
    client_options=dict(
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        # По умолчанию возьмет значение переменной окружения SENTRY_NAME
        name="",
        # По умолчанию возьмет значение переменной окружения SENTRY_ENVIRONMENT
        environment="full_example",
        # По умолчанию возьмет значение переменной окружения SENTRY_RELEASE
        release=__version__,

        # Умолчания для остальных аргументов
        include_paths=set(),
        exclude_paths=set(),
        auto_log_stacks=True,
        capture_locals=True,
        string_max_length=400,
        list_max_length=50,
        site=None,
        include_versions=True,
        processors=(
            'raven.processors.SanitizePasswordsProcessor',
        ),
        sanitize_keys=None,
        context={'sys.argv': getattr(sys, 'argv', [])[:]},
        tags={},
        sample_rate=1,
        ignore_exceptions=(),
    )
)


with entrypoint(raven_sender) as loop:
    loop.run_until_complete(main())
```

Вы можете найти полное описание параметров в документации Raven.

SDWatchdogService

Готовый к использованию сервис, просто добавьте его в entrypoint и он будет отправлять уведомления сторожевому таймеру SystemD.

Вы можете безопасно добавлять это в любом случае, так как если сервис не найдет переменных окружения, которые устанавливает systemd, Сервис просто не запустится, однако выполнение приложения продолжится.

Пример python файла:

```
import logging
from time import sleep

from aiomisc import entrypoint
from aiomisc.service.sdwatchdog import SDWatchdogService


if __name__ == '__main__':
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with entrypoint(SDWatchdogService()) as loop:
        pass
```

Пример systemd сервис-файла:

```
[Service]
# Активизируем механизм уведомлений
Type=notify

# Команда которую следует запускать
ExecStart=/home/mosquito/.venv/aiomisc/bin/python /home/mosquito/scratch.py

# Время через которое программа должны посылать нотификации сторожевого таймера
WatchdogSec=5

# Процесс будет убит если он перестал отвечать сторожевому таймеру
WatchdogSignal=SIGKILL

# Сервис будет перезапущен в случае ошибки
Restart=on-failure

# Пробуем убить сам процесс вместо всей cgroup
KillMode=process

# Пробуем остановить сервис "помягче"
KillSignal=SIGINT

# Пробуем остановить сервис "помягче" при перезапуске
RestartKillSignal=SIGINT

# Слать SIGKILL если произошел таймаут остановки
FinalKillSignal=SIGKILL
SendSIGKILL=yes
```

Класс ProcessService

Базовый класс для запуска функции отдельным системным процессом и завершения при остановке родительского процесса.

```
from typing import Dict, Any

import aiomisc.service

# Реализация вымышленного майнера
from .my_miner import Miner


class MiningService(aiomisc.service.ProcessService):
    bitcoin: bool = False
    monero: bool = False
    dogiecoin: bool = False
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def in_process(self) -> Any:
        if self.bitcoin:
            miner = Miner(kind="bitcoin")
        elif self.monero:
            miner = Miner(kind="monero")
        elif self.dogiecoin:
            miner = Miner(kind="dogiecoin")
        else:
            # Ничего делать не нужно
            return

        miner.do_mining()


services = [
    MiningService(bitcoin=True),
    MiningService(monero=True),
    MiningService(dogiecoin=True),
]

if __name__ == '__main__':
    with aiomisc.entrypoint(*services) as loop:
        loop.run_forever()
```

Класс RespawningProcessService

Базовый класс для запуска функции отдельным системным процессом и завершения при остановке родительского процесса. Это очень похоже на *ProcessService* с одним отличием - если дочерний процесс неожиданно завершится, то он будет перезапущен.

```
import logging
from typing import Any

import aiomisc

from time import sleep


class SuicideService(aiomisc.service.RespawningProcessService):
    def in_process(self) -> Any:
        sleep(10)
        logging.warning("Goodbye mad world")
        exit(42)


if __name__ == '__main__':
    with aiomisc.entrypoint(SuicideService()) as loop:
        loop.run_forever()
```

4.2.5 Абстрактный пул соединений

`Aiomisc.PoolBase` - абстрактный класс для реализации определяемого пользователем пула соединений.

Пример для `aioredis`:

```
import asyncio
import aioredis
import aiomisc


class RedisPool(aiomisc.PoolBase):
    def __init__(self, uri, maxsize=10, recycle=60):
        super().__init__(maxsize=maxsize, recycle=recycle)
        self.uri = uri

    async def _create_instance(self):
        return await aioredis.create_redis(self.uri)

    async def _destroy_instance(self, instance: aioredis.Redis):
        instance.close()
        await instance.wait_closed()

    async def _check_instance(self, instance: aioredis.Redis):
        try:
            await asyncio.wait_for(instance.ping(1), timeout=0.5)
        except:
            return False

        return True


async def main():
    pool = RedisPool("redis://localhost")
    async with pool.acquire() as connection:
        await connection.set("foo", "bar")

    async with pool.acquire() as connection:
        print(await connection.get("foo"))


asyncio.run(main())
```

4.2.6 Контекст

Иногда сервисы должны запрашивать данные друг друга. В этом случае вам следует использовать «Контекст».

`Context` по своей сути это репозиторий ассоциированный с запущенным `entrypoint`.

Экземпляр класса `Context` создается в `entrypoint` запускается и ассоциируется с запущенным event loop.

Взаимозависимые сервисы могут ожидать или устанавливать данные друг друга через контекст.

Для сервисов контекст доступн как `self.context` с момента запуска Entrypoint. В других случаях функция `get_context()` возвращает текущий контекст.

```
import asyncio
from random import random, randint

from aiomisc import entrypoint, get_context, Service


class LoggingService(Service):
    async def start(self):
        context = get_context()

        wait_time = await context['wait_time']

        print('Wait time is', wait_time)
        self.start_event.set()

        while True:
            print('Hello from service', self.name)
            await asyncio.sleep(wait_time)


class RemoteConfiguration(Service):
    async def start(self):
        # Понарошку делаем запрос с удаленного сервера
        await asyncio.sleep(random())

        self.context['wait_time'] = randint(1, 5)


services = (
    LoggingService(name='#1'),
    LoggingService(name='#2'),
    LoggingService(name='#3'),
    RemoteConfiguration()
)

with entrypoint(*services) as loop:
    pass
```

Примечание: Это не панацея. В простом случае службы можно настроить, передав kwargs в `__init__`.

4.2.7 @aiomisc.timeout

Декоратор который ограничивает время выполнения оборачиваемой сопрограммы.

```
from aiomisc import timeout

@timeout(1)
async def bad_func():
    await asyncio.sleep(2)
```

4.2.8 @aiomisc.asyncbackoff

``asyncbackoff `` декоратор который обеспечивает политику повторов и политику максимального количества повторных попыток выполнения асинхронной функции.

Основной принцип может быть описан пятью правилами:

- функция будет отменена, если она будет выполнена дольше, чем `deadline` (если указано)
- функция будет отменена при выполнении дольше, чем ``attempt_timeout` (если указано) после этого будет выполнена повторная попытка.
- Попытки выполняются после `pause` секунд (если указано, по умолчанию `0`)
- Попытки будут выполняться не более `max_tries` раз.
- аргумент `giveup` - это функция, которая решает, следует ли «сдаться» и прекратить дальнейшие попыток или продолжать.

Все эти правила работают одновременно.

Описание аргументов:

- `attempt_timeout` максимально допустимое время выполнения одной попытки.
- `deadline` максимально допустимое время выполнения всех попыток.
- `pause` промежуток времени между попытками.
- `exceptions` делает последующие попытки только если эти будут брошены эти исключения.
- `giveup` (именовынный аргумент) это функция-предикат, которая может решить по данному исключению, следует ли нам продолжать повторять попытки.
- `max_tries` (именовынный аргумент) - максимальное количество попыток ($>= 1$).

Декоратор, обеспечивающий соблюдение временных ограничений «attempt_timeout» и «deadline» декорированной функцией.

В случае возникновения исключения функция будет вызвана снова с аналогичными аргументами через кол-во секунд переданное в аргументе «pause».

Объявление через позиционные аргументы:

```
import aiomisc

attempt_timeout = 0.1
deadline = 1
pause = 0.1

@aiomisc.asyncbackoff(attempt_timeout, deadline, pause)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def db_fetch():
    ...


@aiomisc.asyncbackoff(0.1, 1, 0.1)
async def db_save(data: dict):
    ...


# Передаем список исключений
@aiomisc.asyncbackoff(0.1, 1, 0.1, TypeError, RuntimeError, ValueError)
async def db_fetch(data: dict):
    ...
```

Объявление через именованные аргументы

```
import aiomisc

attempt_timeout = 0.1
deadline = 1
pause = 0.1

@aiomisc.asyncbackoff(attempt_timeout=attempt_timeout,
                      deadline=deadline, pause=pause)
async def db_fetch():
    ...


@aiomisc.asyncbackoff(attempt_timeout=0.1, deadline=1, pause=0.1)
async def db_save(data: dict):
    ...


# Передаем список исключений
@aiomisc.asyncbackoff(attempt_timeout=0.1, deadline=1, pause=0.1,
                      exceptions=[TypeError, RuntimeError, ValueError])
async def db_fetch(data: dict):
    ...


# Будет повторено не больше чем 2 раза (всего 3 попытки)
@aiomisc.asyncbackoff(attempt_timeout=0.5, deadline=1,
                      pause=0.1, max_tries=3,
                      exceptions=[TypeError, RuntimeError, ValueError])
async def db_fetch(data: dict):
    ...


# Повторы будут только если соединение было сброшено (только POSIX системы)
@asyncbackoff(attempt_timeout=0.5, deadline=1, pause=0.1,
              exceptions=[OSError],
              giveup=lambda e: e.errno != errno.ECONNABORTED)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def db_fetch(data: dict):
    ...
```

4.2.9 asyncretry

Аналог `asyncbackoff(None, None, 0, *args, **kwargs)`. Прсто повторяет выполнение функции `max_tries` раз.

Примечание: По умолчанию будет повторена попытка при любом исключении. Это очень просто и полезно в общих случаях, при этом следует указать список исключений тогда, когда обернутые функции вызывают сотни раз в секунду, потому что у вас есть риск быть причиной отказа в обслуживании в случае, если ваша функция вызывает какой-то сервис по сети.

```
from aiomisc import asyncretry

@asyncretry(5)
async def try_download_file(url):
    ...

@asyncretry(3, exceptions=(ConnectionError,))
async def get_cluster_lock():
    ...
```

4.2.10 Предохранитель

Предохранитель - это шаблон проектирования, используемый при разработке программного обеспечения. Он используется для обнаружения сбоев и инкапсулирует логику предотвращения постоянного повторения сбоя во время обслуживания, временного сбоя внешней системы или неожиданных системных проблем.

Следующий пример демонстрирует простое использование текущей реализации `aiomisc.CircuitBreaker`. Экземпляр `CircuitBreaker`, собирает статистику вызовов функций. Он содержит счетчики которые содержат успешные и неудачные вызовы функции. Вызовы функций должны быть обернуты методом *CircuitBreaker.call*, чтобы собирать статистику.

Пример использования:

```
from aiohttp import web, ClientSession
from aiomisc.service.aiohttp import AIOHTTPService
import aiohttp
import aiomisc


async def public_gists(request):
    async with aiohttp.ClientSession() as session:
        # Используем как контекстный менеджер
        with request.app["github_cb"].context():
            url = 'https://api.github.com/gists/public'
            async with session.get(url) as response:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                data = await response.text()

    return web.Response(
        text=data,
        headers={"Content-Type": "application/json"}
    )


class API(AIOHTTPService):
    async def create_application(self):
        app = web.Application()
        app.add_routes([web.get('/', public_gists)])

        # Если ошибок 30% за 20 секунд
        # Сломаемся на 5 секунд
        app["github_cb"] = aiomisc.CircuitBreaker(
            error_ratio=0.2,
            response_time=20,
            exceptions=[aiohttp.ClientError],
            broken_time=5
        )

        return app


async def main():
    async with ClientSession() as session:
        async with session.get("http://localhost:8080/") as response:
            assert response.headers


if __name__ == '__main__':
    with aiomisc.entrypoint(API(port=8080)) as loop:
        loop.run_until_complete(main())
```

Объект *CircuitBreaker* может находится в одном из трех состояний:

- **PASSING** - аналогия с «ток протекает» (прим. пер.)
- **BROKEN** - аналогия со «сгоревшим предохранителем» (прим. пер.)
- **RECOVERING** - восстанавливается

PASSING означает, что все вызовы будут переданы как есть, но будет собираться статистика. Следующее состояние будет определено после сбора статистики за `passing_time` секунд. Если эффективный коэффициент ошибок больше чем *error_ ratio*, тогда следующее состояние будет установлено как **BROKEN** (предохранитель перегорел, прим. пер.), в противном случае оно останется неизменным.

BROKEN означает, что обернутая функция не будет вызываться и вместо нее будет брошено исключение `CircuitBroken`. Состояние **BROKEN** будет сохраняться в течение `broken_time` секунд.

Примечание: Исключение `CircuitBroken` является следствием состояния **BROKEN** или **RECOVERY** и никогда не учитывается в статистике.

После этого предохранитель переходит в состояние **RECOVERING**. В этом состоянии фактически будет выполняться небольшая выборка обернутых вызовов функции, но будет собираться статистика. Если эффективный коэффициент ошибок после `recovery_time` ниже, чем `error_ratio`, то следующее состояние будет установлено в **PASSING**, в противном случае снова **BROKEN**.

Аргумент `exception_inspector` это функция, которая вызывается всякий раз, когда происходит исключение из списка отслеживаемых исключений. Если она возвращает `False` - исключение будет проигнорировано.

4.2.11 Декоратор `cutout` - «рубильник»

Декоратор оборачивающий функцию таким образом, что все вызовы проходят через предохраннитель, а именно через экземпляр `CircuitBreaker` для этой функции.

```
from aiohttp import web, ClientSession
from aiomisc.service.aiohttp import AIOHTTPService
import aiohttp
import aiomisc


# Если 20% ошибок за 30 секунд
# Сломаемся на 30 секунд
@aiomisc.cutout(0.2, 30, aiohttp.ClientError)
async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()


async def public_gists(request):
    async with aiohttp.ClientSession() as session:
        data = await fetch(
            session,
            'https://api.github.com/gists/public'
        )

    return web.Response(
        text=data,
        headers={"Content-Type": "application/json"}
    )


class API(AIOHTTPService):
    async def create_application(self):
        app = web.Application()
        app.add_routes([web.get('/', public_gists)])
        return app


async def main():
    async with ClientSession() as session:
        async with session.get("http://localhost:8080/") as response:
            assert response.headers
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == '__main__':
    with aiomisc.entrypoint(API(port=8080)) as loop:
        loop.run_until_complete(main())
```

4.2.12 @aiomisc.aggregate

Параметрический декоратор, который агрегирует несколько (но не больше, чем `max_count`, по умолчанию `None`) вызовов с одним параметром (`res1 = await func(arg1)`, `res2 = await func(arg2)`, ...) асинхронной функции с переменными количеством позиционных параметров (`async def func(*args, pho=1, bo=2) -> Iterable`) в единственный вызов с несколькими параметрами (`res1, res2, ... = await func(arg1, arg2, ...)`), собранными в течение окна `leeway_ms`. Он позволяет пожертвовать задержкой ради увеличения пропускной способности.

Если `func` бросает исключение, тогда все агрегированные вызовы бросят то же самое исключение. Если один агрегированный вызов будет отменён во время выполнения `func`, тогда другой попробует выполнить `func` вместо него.

Этот декоратор может быть полезен, если `func` выполняет медленные IO-задачи, часто вызывается, а использование кеширования не предпочтительно. В качестве примера, пусть `func` запрашивает запись из БД по ID пользователя во время каждого запроса к нашему сервису. Если запрос к БД занимает 100 мс, а нагрузка на сервис составляет 1000 RPS, то с 10% увеличением задержки (до 110 ms) количество запросов к БД упадёт в 10 раз (до 100 QPS)

```
import asyncio
import math
from aiomisc import aggregate, entrypoint

@aggregate(leeway_ms=10, max_count=2)
async def pow(*nums: float, power: float = 2.0):
    return [math.pow(num, power) for num in nums]

async def main():
    await asyncio.gather(pow(1.0), pow(2.0))

with entrypoint() as loop:
    loop.run_until_complete(main())
```

Для более низкоуровнего подхода можно воспользоваться декоратором *aggregate_async*. В этом случае агрегирующая функция принимает в качестве параметров переменные *Arg*, содержащие значение параметра *value* и футуру *future*. Функция ответственна за выставление результатов работы для всех футур (вместо обычного возврата результатов).

```
import asyncio
import math
from aiomisc import aggregate_async, entrypoint
from aiomisc.aggregate import Arg


@aggregate_async(leeway_ms=10, max_count=2)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def pow(*args: Arg, power: float = 2.0):
    for arg in args:
        arg.future.set_result(math.pow(arg.value, power))

async def main():
    await asyncio.gather(pow(1), pow(2))

with entrypoint() as loop:
    loop.run_until_complete(main())
```

4.2.13 асинхронные операции с файлами

Асинхронные файловые операции, включая поточную компрессию данных. Работают в пуле потоков «под капотом».

```
import aiomisc
import tempfile
from pathlib import Path


async def file_write():
    with tempfile.TemporaryDirectory() as tmp:
        fname = Path(tmp) / 'test.txt'

        # Некоторые инструменты, такие как туру, не смогут вывести тип
        # из функции async_open основываясь на переданном символе `b` в режим.
        # Придется тут подсказать тип явно.
        afp: aiomisc.io.AsyncTextIO

        async with aiomisc.io.async_open(fname, 'w+') as afp:
            await afp.write("Hello")
            await afp.write(" ")
            await afp.write("world")

            await afp.seek(0)
            print(await afp.read())


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(file_write())
```

Этот способ работы с файлами основан на потоках. Это очень похоже на то как сделано в библиотеке aiofiles с теми-же ограничениями.

Разумеется вы можете использовать библиотеку aiofile для этого. Но это не панацея, так как имеет ограничения связанные с API операционной системы.

В основном, для небольших нагрузок приложений, я рекомендую придерживаться следующих правил:

- Если чтение и запись маленьких или больших кусочков в файлы со случайным доступом основная задача приложения - стоит использовать aiofile.
- Иначе можно взять этот модуль или aiofiles

- Если основная задача читать большие куски файлов для дальнейшей их обработки оба вышеописанных метода будут не оптимальны, так как переключения контекста каждую IO операцию - это скорее всего не будет оптимально для файлового кеша и можно потерять бóльшую часть времени исполнения на переключение контекста исполнения. В случае имплементации асинхронного IO на основе потоков цена переключения контектса между потоками может оказаться выше чем суммарно время исполнения всех IO операций.

  Просто попробуйте завернуть все блокирующие вызовы в отдельные функции и вызывайте их используя пул потоков (см. пример ниже):

```
import os
import aiomisc
import hashlib
import tempfile
from pathlib import Path


@aiomisc.threaded
def hash_file(filename, chunk_size=65535, hash_func=hashlib.blake2b):
    hasher = hash_func()

    with open(filename, "rb") as fp:
        for chunk in iter(lambda: fp.read(chunk_size), b""):
            hasher.update(chunk)

    return hasher.hexdigest()


@aiomisc.threaded
def fill_random_file(filename, size, chunk_size=65535):
    with open(filename, "wb") as fp:
        while fp.tell() < size:
            fp.write(os.urandom(chunk_size))

        return fp.tell()


async def main(path):
    filename = path / "one"
    await fill_random_file(filename, 1024 * 1024)
    first_hash = await hash_file(filename)

    filename = path / "two"
    await fill_random_file(filename, 1024 * 1024)
    second_hash = await hash_file(filename)

    assert first_hash != second_hash


with tempfile.TemporaryDirectory(prefix="random.") as path:
    aiomisc.run(
        main(Path(path))
    )
```

Поточное сжатие

Чтобы включить поточное сжатие, нужно передать аргумент *compression* в функцию *async_open*.

Поддерживаемые алгоритмы сжатия:

- `aiomisc.io.Compression.NONE`
- `aiomisc.io.Compression.GZIP`
- `aiomisc.io.Compression.BZ2`
- `aiomisc.io.Compression.LZMA`

Пример использования

```
import tempfile
from aiomisc import run
from aiomisc.io import async_open, Compression
from pathlib import Path


async def file_write():
    with tempfile.TemporaryDirectory() as tmp:
        fname = Path(tmp) / 'test.txt'

        async with async_open(
            fname, 'w+', compression=Compression.GZIP
        ) as afp:
            for _ in range(10000):
                await afp.write("Hello World\n")

        file_size = fname.stat().st_size
        assert file_size < 10000, f"File too large {file_size} bytes"

run(file_write())
```

4.2.14 Работа с потоками

Можно обернуть блокирующую функцию и запустить ее в отдельном потоке или пуле потоков.

Поддержка `contextvars`

Все нижеописанные декоратоы и функции поддерживают модуль `contextvars`, из PyPI для python моложе 3.7 так и встроенный в стандартную библиотеку модуль для python 3.7.

```
import asyncio
import aiomisc
import contextvars
import random
import struct


user_id = contextvars.ContextVar("user_id")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
record_struct = struct.Struct(">I")


@aiomisc.threaded
def write_user():
    with open("/tmp/audit.bin", 'ab') as fp:
        fp.write(record_struct.pack(user_id.get()))


@aiomisc.threaded
def read_log():
    with open("/tmp/audit.bin", "rb") as fp:
        for chunk in iter(lambda: fp.read(record_struct.size), b''):
            yield record_struct.unpack(chunk)[0]


async def main():
    futures = []
    for _ in range(5):
        user_id.set(random.randint(1, 65535))
        futures.append(write_user())

    await asyncio.gather(*futures)

    async for data in read_log():
        print(data)


if __name__ == '__main__':
    with aiomisc.entrypoint() as loop:
        loop.run_until_complete(main())
```

Пример вывода

```
6621
33012
1590
45008
56844
```

Примечание: `contextvars` нужны для другого случая нежели класс `Context`. `contextvars` хорошо применимы для передачи контекстных переменных сквозь стек вызовов, однако дочерние задачи не смогут модифицировать контекстные переменные родительских задач потому, что `contextvars` делает их «легкие копии» перед запуском новой задачи. Класс `Context` наоборот позволяет модификацию отовсюду так как не копирует ничего.

`@aiomisc.threaded`

Оборачивает блокирующую функцию и запускает ее в пуле потоков.

```
import asyncio
import time
from aiomisc import new_event_loop, threaded


@threaded
def blocking_function():
    time.sleep(1)


async def main():
    # Параллельный запуск
    await asyncio.gather(
        blocking_function(),
        blocking_function(),
    )


if __name__ == '__main__':
    loop = new_event_loop()
    loop.run_until_complete(main())
```

В случае если функция это генератор тогда декоратор `@threaded` вернет `IteratorWrapper` (см Threaded generator decorator).

`@aiomisc.threaded_separate`

Оборачивает блокирующую функцию и запускает ее в отдельном новом потоке.Крайне рекомендовано для длительных фоновых задач:

```
import asyncio
import time
import threading
import aiomisc


@aiomisc.threaded
def blocking_function():
    time.sleep(1)


@aiomisc.threaded_separate
def long_blocking_function(event: threading.Event):
    while not event.is_set():
        print("Running")
        time.sleep(1)
    print("Выходим")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
async def main():
    stop_event = threading.Event()

    loop = asyncio.get_event_loop()
    loop.call_later(10, stop_event.set)

    # Параллельный запуск
    await asyncio.gather(
        blocking_function(),
        # Будет запущен отдельный новый поток
        long_blocking_function(stop_event),
    )


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

Threaded iterator decorator

Оборачивает блокирующую функцию-генератор и запусакет ее в текущем пуле потоков или новом отдельном потоке.

Следующий пример читает свой собственный код, и обновляет хеш для каждой следующей строки от предидущих строк, и отправляет все это по TCP:

```
import asyncio
import hashlib

import aiomisc

# Мой первый блокчейн

@aiomisc.threaded_iterable
def blocking_reader(fname):
    with open(fname, "r+") as fp:
        md5_hash = hashlib.md5()
        for line in fp:
            bytes_line = line.encode()
            md5_hash.update(bytes_line)
            yield bytes_line, md5_hash.hexdigest().encode()


async def main():
    reader, writer = await asyncio.open_connection("127.0.0.1", 2233)
    async with blocking_reader(__file__) as gen:
        async for line, digest in gen:
            writer.write(digest)
            writer.write(b'\t')
            writer.write(line)
            await writer.drain()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
with aiomisc.entrypoint() as loop
    loop.run_until_complete(main())
```

Запустим TCP сервер с помощью `netcat` в терминале, и после запустим этот пример.

```
$ netcat -v -l -p 2233
Connection from 127.0.0.1:54734
dc80feba2326979f8976e387fbbc8121   import asyncio
78ec3bcb1c441614ede4af5e5b28f638   import hashlib
b7df4a0a4eac401b2f835447e5fc4139
f0a94eb3d7ad23d96846c8cb5e327454   import aiomisc
0c05dde8ac593bad97235e6ae410cb58
e4d639552b78adea6b7c928c5ebe2b67   # Мой первый блокчейн
5f04aef64f4cacce39170142fe45e53e
c0019130ba5210b15db378caf7e9f1c9   @aiomisc.threaded_iterable
a720db7e706d10f55431a921cdc1cd4c   def blocking_reader(fname):
0895d7ca2984ea23228b7d653d0b38f2       with open(fname, "r+") as fp:
0feca8542916af0b130b2d68ade679cf           md5_hash = hashlib.md5()
4a9ddfea3a0344cadd7a80a8b99ff85c           for line in fp:
f66fa1df3d60b7ac8991244455dff4ee               bytes_line = line.encode()
aaac23a5aa34e0f5c448a8d7e973f036               md5_hash.update(bytes_line)
2040bcaab6137b60e51ae6bd1e279546               yield bytes_line, md5_hash.hexdigest().
↪encode()
7346740fdcde6f07d42ecd2d6841d483
14dfb2bae89fa0d7f9b6cba2b39122c4
d69cc5fe0779f0fa800c6ec0e2a7cbbd   async def main():
ead8ef1571e6b4727dcd9096a3ade4da       reader, writer = await asyncio.open_connection(
↪"127.0.0.1", 2233)
275eb71a6b6fb219feaa5dc2391f47b7       async with blocking_reader(__file__) as gen:
110375ba7e8ab3716fd38a6ae8ec8b83           async for line, digest in gen:
c26894b38440dbdc31f77765f014f445               writer.write(digest)
27659596bd880c55e2bc72b331dea948               writer.write(b'\t')
8bb9e27b43a9983c9621c6c5139a822e               writer.write(line)
2659fbe434899fc66153decf126fdb1c               await writer.drain()
6815f69821da8e1fad1d60ac44ef501e
5acc73f7a490dcc3b805e75fb2534254
0f29ad9505d1f5e205b0cbfef572ab0e   if __name__ == '__main__':
8b04db9d80d8cda79c3b9c4640c08928       loop = aiomisc.new_event_loop()
9cc5f29f81e15cb262a46cf96b8788ba       loop.run_until_complete(main())
```

Придется использовать асинхронный контекст-менеджер в случае если генератор бесконечный или придется явно вызвать и дождаться метода `.close()` если вы избегаете использование асинхронных контекст-менеджеров.

```
import asyncio
import aiomisc


# Настраиваем буфер на 2 элемента вперед
@aiomisc.threaded_iterable(max_size=2)
def urandom_reader():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with open('/dev/urandom', "rb") as fp:
        while True:
            yield fp.read(8)


# Бесконечный буфер в отдельном потоке
@aiomisc.threaded_iterable_separate
def blocking_reader(fname):
    with open(fname, "r") as fp:
        yield from fp


async def main():
    reader, writer = await asyncio.open_connection("127.0.0.1", 2233)
    async for line in blocking_reader(__file__):
        writer.write(line.encode())

    await writer.drain()

    # Будем "кушать" белый шум
    gen = urandom_reader()
    counter = 0
    async for line in gen:
        writer.write(line)
        counter += 1

        if counter == 10:
            break

    await writer.drain()

    # Останавливаем запущенный генератор
    await gen.close()

    # Тоже самое, только через контекст менеджер
    async with urandom_reader() as gen:
        counter = 0
        async for line in gen:
            writer.write(line)
            counter += 1

            if counter == 10:
                break

    await writer.drain()


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

aiomisc.IteratorWrapper

Запускает блокирующий итератор в пуле потоков:

```
import concurrent.futures
import hashlib
import aiomisc


def urandom_reader():
    with open('/dev/urandom', "rb") as fp:
        while True:
            yield fp.read(1024)


async def main():
    # Создаем новый пул потоков
    pool = concurrent.futures.ThreadPoolExecutor(1)
    wrapper = aiomisc.IteratorWrapper(
        urandom_reader,
        executor=pool,
        max_size=2
    )

    async with wrapper as gen:
        md5_hash = hashlib.md5(b'')
        counter = 0
        async for item in gen:
            md5_hash.update(item)
            counter += 1

            if counter >= 100:
                break

    pool.shutdown()
    print(md5_hash.hexdigest())


if __name__ == '__main__':
    with aiomisc.entrypoint() as loop:
        loop.run_until_complete(main())
```

aiomisc.IteratorWrapperSeparate

Запускает блокирующий итераторы в отдельном новом потоке:

```
import concurrent.futures
import hashlib
import aiomisc


def urandom_reader():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with open('/dev/urandom', "rb") as fp:
        while True:
            yield fp.read(1024)


async def main():
    # Создаем новый пул потоков
    wrapper = aiomisc.IteratorWrapperSeparate(
        urandom_reader, max_size=2
    )

    async with wrapper as gen:
        md5_hash = hashlib.md5(b'')
        counter = 0
        async for item in gen:
            md5_hash.update(item)
            counter += 1

            if counter >= 100:
                break

    print(md5_hash.hexdigest())


if __name__ == '__main__':
    with aiomisc.entrypoint() as loop:
        loop.run_until_complete(main())
```

`aiomisc.ThreadPoolExecutor`

Реализация быстрого и простого пула потоков.

Устанавливаем как пул потоков по умолчанию:

```
import asyncio
from aiomisc import ThreadPoolExecutor

loop = asyncio.get_event_loop()
thread_pool = ThreadPoolExecutor(4)
loop.set_default_executor(thread_pool)
```

Примечание: `entrypoint` установит это по умолчанию.

Аргумент `pool_size` в `entrypoint` ограничивает кол-во потоков п пуле.

`aiomisc.sync_wait_coroutine`

Функции запущенные в потоке не могут вызывать и дожидаться сопрограмм по умолчанию. Эта функция это хелпер позволяет отправить сопрограмму в event loop и дождаться ее результата из текущего потока.

```
import asyncio
import aiomisc


async def coro():
    print("Coroutine started")
    await asyncio.sleep(1)
    print("Coroutine done")


@aiomisc.threaded
def in_thread(loop):
    print("Thread started")
    aiomisc.sync_wait_coroutine(loop, coro)
    print("Thread finished")


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(in_thread(loop))
```

4.2.15 ProcessPoolExecutor

Это простая реализация пула процессов.

Пример:

```
import asyncio
import time
import os
from aiomisc import ProcessPoolExecutor

def process_inner():
    for _ in range(10):
        print(os.getpid())
        time.sleep(1)

    return os.getpid()


loop = asyncio.get_event_loop()
process_pool = ProcessPoolExecutor(4)


async def main():
    print(
        await asyncio.gather(
            loop.run_in_executor(process_pool, process_inner),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            loop.run_in_executor(process_pool, process_inner),
            loop.run_in_executor(process_pool, process_inner),
            loop.run_in_executor(process_pool, process_inner),
        )
    )

loop.run_until_complete(main())
```

4.2.16 Утилиты

`select`

Иногда требуется дождаться выполнения хотя-бы одной задачи из многих. `select` ожидает выполнения одной из переданных сопрограмм (или объектов с реализованным методом *__await__*) И возвращает список результатов.

```
import asyncio
import aiomisc


async def main():
    loop = asyncio.get_event_loop()
    event = asyncio.Event()
    future = asyncio.Future()

    loop.call_soon(event.set)

    await aiomisc.select(event.wait(), future)
    print(event.is_set())       # True

    event = asyncio.Event()
    future = asyncio.Future()

    loop.call_soon(future.set_result, True)

    results = await aiomisc.select(future, event.wait())
    future_result, event_result = results

    print(results.result())             # True
    print(results.result_idx)           # 0
    print(event_result, future_result)  # None, True


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

Предупреждение: В случае если вы не желаете отменять запущенные задачии передайте аргумент `cancel=False`. Но в этом случае вам придется разобраться с завершением или отменой самостоятельон иначе будет предупреждение от интерпретатора.

cancel_tasks

Все переданные задачи будут отменены, при это функция возвращает *asyncio.Task*:

```
import asyncio
from aiomisc import cancel_tasks


async def main():
    done, pending = await asyncio.wait([
        asyncio.sleep(i) for i in range(10)
    ], timeout=5)

    print("Done", len(done), "tasks")
    print("Pending", len(pending), "tasks")
    await cancel_tasks(pending)


asyncio.run(main())
```

awaitable

Оборачивает функции таким образом что они всегда возвращают сопрограмму. Если функция возвращает объект *asyncio.Future*, будет возвращен оригинальный объект. Если функция итак возвращает сопрограмму, или объект с реализованным методом *__await__* будет возвращен оригинальный объект.В противном случае возвращаемый объект будет тобернут в сопрограмму, которая вернет этот объект.Это полезно если не хочется проверять возврат из функции перед тем как использовать ее в *await* выражении.

```
import asyncio
import aiomisc


async def do_callback(func, *args):
    awaitable_func = aiomisc.awaitable(func)

    return await awaitable_func(*args)


print(asyncio.run(do_callback(asyncio.sleep, 2)))
print(asyncio.run(do_callback(lambda: 45)))
```

bind_socket

Создает сокет и устанавливает для него необходимые для работы с asyncio флаги (вроде `setblocking(False)`). Так-же определяет семейство адресов (IPv6/IPv4) из формата аргумента `address` автоматически.

```
from aiomisc import bind_socket

# IPv4 socket
```

(continues on next page)

(продолжение с предыдущей страницы)

```
sock = bind_socket(address="127.0.0.1", port=1234)

# IPv6 socket (on Linux IPv4 socket will be bind too)
sock = bind_socket(address="::1", port=1234)
```

RecurringCallback

Запускает сопрограммы переодически с определяемой пользователем стратегией.

```
from typing import Union
from aiomisc import new_event_loop, RecurringCallback


async def callback():
    print("Hello")


FIRST_CALL = False


async def strategy(_: RecurringCallback) -> Union[int, float]:
    global FIRST_CALL
    if not FIRST_CALL:
        FIRST_CALL = True
        # Ждем 5 секунд если только-что запустились
        return 5

    # Ждем 10 секунд если это не первый запуск
    return 10


if __name__ == '__main__':
    loop = new_event_loop()

    periodic = RecurringCallback(callback)

    task = periodic.start(strategy)
    loop.run_forever()
```

Основная цель этого класса - предоставить возможность указывать стратегией асинхронную функцию, которая может быть так как вам нужно.

Кроме того, с помощью специальных исключений вы можете управлять поведением запущенного RecurringCallback.

```
from aiomisc import (
    new_event_loop, RecurringCallback, StrategySkip, StrategyStop
)


async def strategy(_: RecurringCallback) -> Union[int, float]:
    ...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    # Пропускаем эту попытку и ждем 10 секунд
    raise StrategySkip(10)

    ...

    # Stop execution
    raise StrategyStop()
```

если функция-стратегия возвращает неверное значение (не число) или не вызывает специальных исключений, повторяющееся выполнение завершается.

PeriodicCallback

Запускает сопрограммы переодически с заданным периодом времени, и необязательной задержкой при первом запуске. Использует `RecurringCallback` под капотом.

```
import asyncio
import time
from aiomisc import new_event_loop, PeriodicCallback


async def periodic_function():
    print("Hello")


if __name__ == '__main__':
    loop = new_event_loop()

    periodic = PeriodicCallback(periodic_function)

    # Ждем 10 секунд и вызываем это каждую секунду после этого
    periodic.start(1, delay=10)

    loop.run_forever()
```

CronCallback

Предупреждение: Придется установить пакет `croniter` чтобы пользоваться этим:

```
pip install croniter
```

Или указать это как extras при установке aiomisc:

```
pip install aiomisc[cron]
```

Запускает сопрограммы переодически, как-будто с помощью cron.Использует `RecurringCallback` под капотом.

```
import asyncio
import time
from aiomisc import new_event_loop, CronCallback


async def cron_function():
    print("Hello")


if __name__ == '__main__':
    loop = new_event_loop()

    periodic = CronCallback(cron_function)

    # Будем запускать это каждую секунду
    periodic.start(spec="* * * * * *")

    loop.run_forever()
```

4.2.17 WorkerPool

В Python есть модуль `multiprocessing` в котором реализован класс `Pool`, это аналог этого модуля, за единственным исключением - IPC в этом случае полностью синхронный. Этот модуль реализует Worker Pool на основе процессов, но IPC, при этом, полностью асинхронный на вызывающей стороне, при этом рабочие процессы не асинхронны.

Пример

Это полезно, когда вы хотите обрабатывать данные в отдельном процессе, при этом входные и выходные данные не велики. В противном случае это, конечно, будет работать нормально, но вам придется тратить время на передачу данных по IPC.

Хорошим примером является параллельная обработка изображений. Конечно, вы можете передавать байты изображений через IPC рабочего пула, но в общем случае передача имени файла будет лучше. Исключением будут случаи, когда изображение очень маленькое меньше, к примеру 1 КБ.

Давайте напишем программу, которая принимает изображения в формате JPEG и создает миниатюры. В этом случае у вас есть файл с исходным изображением, и вы должны сгенерировать выходной путь для функции «thumbnail».

Примечание: Придется установить Pillow - библиотеку для работы с изображениями, чтобы запустить этот код.

Установка через pip:

```
pip install Pillow
```

```
import asyncio
import sys
from multiprocessing import cpu_count
```

(continues on next page)

(продолжение с предыдущей страницы)

```
from typing import Tuple
from pathlib import Path
from PIL import Image
from aiomisc import entrypoint, WorkerPool


def thumbnail(src_path: str, dest_path: str, box: Tuple[int, int]):
    img = Image.open(src_path)
    img.thumbnail(box)
    img.save(
        dest_path, "JPEG", quality=65,
        optimize=True,
        icc_profile=img.info.get('icc_profile'),
        exif=img.info.get('exif'),
    )
    return img.size


sizes = [
    (1024, 1024),
    (512, 512),
    (256, 256),
    (128, 128),
    (64, 64),
    (32, 32),
]


async def amain(path: Path):
    # Создаем директории
    for size in sizes:
        size_dir = "x".join(map(str, size))
        size_path = (path / 'thumbnails' / size_dir)
        size_path.mkdir(parents=True, exist_ok=True)

    # Создаем и запускаем WorkerPool
    async with WorkerPool(cpu_count()) as pool:
        tasks = []
        for image in path.iterdir():
            if not image.name.endswith(".jpg"):
                continue

            if image.is_relative_to(path / 'thumbnails'):
                continue

            for size in sizes:
                rel_path = image.relative_to(path).parent
                size_dir = "x".join(map(str, size))
                dest_path = (
                    path / rel_path /
                    'thumbnails' / size_dir /
                    image.name
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                )

                tasks.append(
                    pool.create_task(
                        thumbnail,
                        str(image),
                        str(dest_path),
                        size
                    )
                )

        await asyncio.gather(*tasks)


if __name__ == '__main__':
    with entrypoint() as loop:
        image_dir = Path(sys.argv[1])
        loop.run_until_complete(amain(image_dir))
```

В этом примере каталог изображений используется в качестве первого аргумента командной строки и создает каталоги для эскизов. После этого запускается `WorkerPool` с таким количеством процессов, сколько ядер у процессора.

Главный процесс создает задачи для рабочих, каждая задача это конвертация одного изображения в один размер, и все эти задачи передаются в `WorkerPool`

`WorkerPool` обрабатывает задачи конкурентно, но не более одной задачи на одного рабочего в один момент времени

4.2.18 Конфигурация логирования

Конфигурация логирования по умолчанию может быть осуществлена через переменные окружения:

- *AIOMISC_LOG_LEVEL* - уровень логирования по умолчанию
- *AIOMISC_LOG_FORMAT* - формат логирования по умолчанию
- *AIOMISC_LOG_CONFIG* - следует ли настраивать логирование
- *AIOMISC_LOG_FLUSH* - интервал сброса буфера логов logs
- *AIOMISC_LOG_BUFFER* - максимальный размер буфера логов

```
$ export AIOMISC_LOG_LEVEL=debug
$ export AIOMISC_LOG_FORMAT=rich
```

Color

Настройка цветных логов:

```
import logging
from aiomisc.log import basic_config


# Configure logging
basic_config(level=logging.INFO, buffered=False, log_format='color')
```

JSON

Настройка json логов:

```
import logging
from aiomisc.log import basic_config


# Configure logging
basic_config(level=logging.INFO, buffered=False, log_format='json')
```

JournalD

JournalD демон для сбора логов он является частью systemd. *aiomisc.basic_config* может писать логи в JournalD.

Примечание: Этот обработчик выбирается по умолчанию если программа запускается как systemd сервис.

aiomisc.log.LogFormat.default() вернет *journald* в этом случае.

```
import logging
from aiomisc.log import basic_config

# Configure rich log handler
basic_config(level=logging.INFO, buffered=False, log_format='journald')

logging.info("JournalD log record")
```

Rich

Rich это Python библиотека которая делает форматирование в терминал прекрасным.

aiomisc.basic_config может использовать Rich для логирования. Но это не обязательно, поэтому вам придется установить Rich самостоятельно.

```
pip install rich
```

Примечание: Этот обработчик будет обработчиком по умолчанию если *Rich* установлен.

```
import logging
from aiomisc.log import basic_config

# Конфигурируем rich обработчик для журналов
basic_config(level=logging.INFO, buffered=False, log_format='rich')

logging.info("Rich logger")

# Конфигурируем rich обработчик для журналов но с показом трейсбеков
basic_config(level=logging.INFO, buffered=False, log_format='rich_tb')

try:
    1 / 0
except:
    logging.exception("Rich traceback logger")
```

Бufферизирующий лог-хендлер

Параметр *buffered=True* включает буферизацию логов в памяти, отдельный поток переодически сбрасывает логи в поток.

```
import asyncio
import logging
from aiomisc.log import basic_config
from aiomisc.periodic import PeriodicCallback
from aiomisc.utils import new_event_loop


# Глобально конфигурируем журналы
basic_config(level=logging.INFO, buffered=False, log_format='json')

async def write_log(loop):
    logging.info("Hello %f", loop.time())

if __name__ == '__main__':
    loop = new_event_loop()

    # Конфигурируем
    basic_config(
        level=logging.INFO,
        buffered=True,
        log_format='color',
        flush_interval=0.5
    )

    periodic = PeriodicCallback(write_log, loop)
    periodic.start(0.3)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    # Ждем пока журналы не попадут в журнал
    loop.run_until_complete(asyncio.sleep(1))
```

Примечание: `entrypoint` принимает аргумент `log_format` через который можно это настроить.

Список всех поддерживаемых форматов журналов доступен через `aiomisc.log.LogFormat.choices()`

4.2.19 Плагин для Pytest

Начиная с версии 17, интеграция с pytest распространяется в виде отдельного пакета aiomisc-pytest. Дополнительные инструкции см. в документации модуля aiomisc-pytest.

4.2.20 Signal

Позволяет зарегистрировать асинхронный callback для определенных событий entrypoint.

pre_start

`pre_start` сигнал происходящий когда entrypoint запускается но до запуска всех сервисов.

```
from aiomisc import entrypoint, receiver

@receiver(entrypoint.PRE_START)
async def prepare_database(entrypoint, services):
  ...

with entrypoint() as loop:
    loop.run_forever()
```

post_start

`post_start` сигнал происходящий после того как entrypoint запустит все сервисы.

```
from aiomisc import entrypoint, receiver

@receiver(entrypoint.POST_START)
async def startup_notifier(entrypoint, services):
  ...

with entrypoint() as loop:
    loop.run_forever()
```

pre_stop

pre_stop сигнал происходящий когда entrypoint завершается до остановки всех сервисов.

```
from aiomisc import entrypoint, receiver

@receiver(entrypoint.PRE_STOP)
async def shutdown_notifier(entrypoint):
  ...

with entrypoint() as loop:
    loop.run_forever()
```

post_stop

post_stop сигнал происходящий когда entrypoint завершается запускается после остановки всех сервисов.

```
from aiomisc import entrypoint, receiver

@receiver(entrypoint.POST_STOP)
async def cleanup(entrypoint):
  ...

with entrypoint() as loop:
    loop.run_forever()
```

4.2.21 Plugins

aiomisc можно расширить с помощью плагинов в виде отдельных python пакетов. Плагины взаимодействуют с aiomisc с помощью *signals*.

Чтобы сделать ваш плагин доступным для обнаружения с помощью aiomisc, вы должны добавить запись `aiomisc.plugins` к записи аргумента `entry_points` вызова `setup` в `setup.py` вашего python пакета.

```
# setup.py

setup(
    # ...
    entry_points={
        "aiomisc": ["myplugin = aiomisc_myplugin.plugin"]
    },
    # ...
)
```

В случае *pyproject.toml* можно это описать вот-так:

```
[tool.poetry.plugins.aiomisc]
myplugin = "aiomisc_myplugin.plugin"
```

Модули, представленные в *entry_points*`, должны иметь функцию `setup`. Эти функции будут вызываться aiomisc и должны поддерживать сигналы.

Если сервисы запускаются динамически, присоединенные функции будут выполняться каждый раз при запуске и остановке служб, однако только те службы, которые в данный момент запускаются или останавливаются, будут в параметре services.

```
# Content of: ``aiomisc_myplugin/plugin.py``
from typing import Tuple
from threading import Event

import aiomisc


event = Event()
# Will be shoum in ``python -m aiomisc.plugins``
__doc__ = "Example plugin"


async def hello(
    *,
    entrypoint: aiomisc.Entrypoint,
    services: Tuple[aiomisc.Service, ...]
) -> None:
    print('Hello from aiomisc plugin')
    event.set()


def setup() -> None:
    """
    This code will be called by loading plugins declared in
    ``pyproject.toml`` or ``setup.py``.
    """
    aiomisc.Entrypoint.PRE_START.connect(hello)

# Content of: ``my_plugin_example.py``
# =====================================================================
# The code below is not related to the plugin, but serves to demonstrate
# how it works.
# =====================================================================
def main():
    """ some function in user code """

    # This function will be called by aiomisc.plugin module
    # in this example it's just for demonstration.
    setup()

    assert not event.is_set()

    with aiomisc.entrypoint() as loop:
        pass

    assert event.is_set()

main()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# remove the plugin on when unneeded
aiomisc.entrypoint.PRE_START.disconnect(hello)
```

Список доступных сигналов такой:

- `Entrypoint.PRE_START` - Запускается перед *стартом* сервисов.
- `Entrypoint.PRE_STOP` - Запускается перед *остановкой* сервисов.
- `Entrypoint.POST_START` - Запускается после *старта* сервисов.
- `Entrypoint.POST_STOP` - Запускается после *остановки* сервисов

Список доступный плагинов

Чтобы просмотреть список всех доступных плагинов, вы можете вызвать из командной строки `python -m aiomisc.plugins`:

```
$ python -m aiomisc.plugins
[11:14:42] INFO     Available 1 plugins.
           INFO     'systemd_watchdog' - Adds SystemD watchdog support to the entrypoint.
systemd_watchdog
```

Вы также можете изменить поведение и вывод списка модулей. Для этого существуют следующие флаги:

```
$ python3 -m aiomisc.plugins -h
usage: python3 -m aiomisc.plugins [-h] [-q] [-n]
                                  [-l {critical,error,warning,info,debug,notset}]
                                  [-F {stream,color,json,syslog,plain,journald,rich,rich_
↪tb}]

optional arguments:
  -h, --help            show this help message and exit
  -q, -s, --quiet, --silent
                        Disable logs and just output plugin-list, alias for
                        --log-level=critical
  -n, --no-output       Disable output plugin-list to the stdout
  -l {critical,error,warning,info,debug,notset}, --log-level {critical,error,warning,
↪info,debug,notset}
                        Logging level
  -F {stream,color,json,syslog,plain,journald,rich,rich_tb}, --log-format {stream,color,
↪json,syslog,plain,journald,rich,rich_tb}
                        Logging format
```

Вот несколько примеров запуска.

```
$ python3 -m aiomisc.plugins -n
[12:25:57] INFO     Available 1 plugins.
           INFO     'systemd_watchdog' - Adds SystemD watchdog support to the entrypoint.
```

Этот пример, печатает удобочитаемый список плагинов и их описания.

```
$ python3 -m aiomisc.plugins -s
systemd_watchdog
```

Это полезно для `grep` или других утилит.

По умолчанию печатается человекочитаемый лог в stderr, и список плагинов в stdout, поэтому можно использовать это без параметров в конвейере, и прочитать список в stderr.

4.2.22 Статистические счетчики

`aiomisc` содержит внутренние счетчики статистики. Вы можете прочитать их с помощью функции *aiomisc.get_statistics()*.

Экземпляры классов статистики создаются динамически. Вы можете использовать определяемые пользователем имена для них добавляя аргумент `statistic_name: Optional[str] = None` в сущности поддерживающие это.

```
import aiomisc

async def main():
    for metric in aiomisc.get_statistics():
        print(
            str(metric.kind.__name__),
            metric.name,
            metric.metric,
            metric.value
        )


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

Этот код выведет что-то вроде:

```
ContextStatistic None get 0
ContextStatistic None set 0
ThreadPoolStatistic logger submitted 1
ThreadPoolStatistic logger sum_time 0
ThreadPoolStatistic logger threads 1
ThreadPoolStatistic logger done 0
ThreadPoolStatistic logger success 0
ThreadPoolStatistic logger error 0
ThreadPoolStatistic default submitted 0
ThreadPoolStatistic default sum_time 0
ThreadPoolStatistic default threads 12
ThreadPoolStatistic default done 0
ThreadPoolStatistic default success 0
ThreadPoolStatistic default error 0
```

4.3 Описание API

4.3.1 Модуль aiomisc

Модуль aiomisc.aggregate

Модуль aiomisc.backoff

Модуль aiomisc.circuit_breaker

Модуль aiomisc.compat

Модуль aiomisc.context

Модуль aiomisc.counters

Модуль aiomisc.cron

Модуль aiomisc.entrypoint

Модуль aiomisc.io

Модуль aiomisc.iterator_wrapper

Модуль aiomisc.log

Модуль aiomisc.periodic

Модуль aiomisc.plugins

Модуль aiomisc.pool

Модуль aiomisc.process_pool

Модуль aiomisc.recurring

Модуль aiomisc.signal

Модуль aiomisc.thread_pool

Модуль aiomisc.timeout

Модуль aiomisc.utils

Модуль aiomisc.worker_pool

4.3.2 Модуль aiomisc_log

Модуль aiomisc_log.formatter.color

Модуль aiomisc_log.formatter.journald

Модуль aiomisc_log.formatter.json

Модуль aiomisc_log.formatter.rich.py

Модуль aiomisc_log.enum

4.3.3 Модуль aiomisc_worker

Модуль aiomisc_worker.forking

Модуль aiomisc_worker.process

Модуль aiomisc_worker.process_inner

Модуль aiomisc_worker.protocol

Модуль aiomisc_worker.worker