aiomisc Documentation

Release 17.3.0

Dmitry Orlov

Apr 18, 2024

CONTENTS

As a programmer, you are no stranger to the challenges that come with building and maintaining software applications. One area that can be particularly difficult is making architecture of the asynchronous I/O software.

This is where `aiomisc` comes in. It is a Python library that provides a collection of utility functions and classes for working with asynchronous I/O in a more intuitive and efficient way. It is built on top of the `asyncio` library and is designed to make it easier for developers to write asynchronous code that is both reliable and scalable.

With `aiomisc`, you can take advantage of powerful features like *worker pools*, *connection pools*, *circuit breaker pattern*, and retry mechanisms such as *asyncbackoff* and *asyncretry* to make your asyncio code more robust and easier to maintain. In this documentation, we'll take a closer look at what `aiomisc` has to offer and how it can help you streamline your asyncio service development.

CHAPTER

ONE

INSTALLATION

Installation is possible in standard ways, such as PyPI or installation from a git repository directly.

Installing from PyPI:

```
pip3 install aiomisc
```

Installing from 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
```

The package contains several extras and you can install additional dependencies if you specify them in this way.

With uvloop:

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

With aiohttp:

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

Complete table of extras bellow:

example	description
`pip install aiomisc[aiohttp]`	For running aiohttp applications.
`pip install aiomisc[asgi]`	For running ASGI applications
`pip install aiomisc[carbon]`	Sending metrics to carbon (part of graphite)
`pip install aiomisc[cron]`	use croniter for scheduling tasks
`pip install aiomisc[raven]`	Sending exceptions to sentry using raven
`pip install aiomisc[rich]`	You might using rich for logging
`pip install aiomisc[uvicorn]`	For running ASGI application using uvicorn
`pip install aiomisc[uvloop]`	use uvloop as a default event loop

You can combine extras values by separating them with commas, for example:

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

CHAPTER

TWO

QUICK START

This section will cover how this library creates and uses the event loop and creates services. For more details see *Tutorial* section, and you can always refer to the *Modules* and *API Reference* sections for help.

2.1 Event-loop and entrypoint

Let's look at this simple example first:

```
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())
```

This code declares an asynchronous `main()` function that exits for 3 seconds. It would seem nothing interesting, but the whole point is in the `entrypoint`.

At the first glance the `entrypoint` did not do much, just creates an event-loop and transfers control to the user. However, under the hood, the logger is configured in a separate thread, a pool of threads is created, services are started, but more on that later as there are no services in this example.

Alternatively, you can choose not to use an `entrypoint`, just create an event-loop and set it as a default for current thread:

```
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
```

(continues on next page)

(continued from previous page)

```
# * 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())
```

The example above is useful if your code is already using an implicitly created event loop, you will have to modify less code, just add `aiomisc.new_event_loop()` and all calls to `asyncio.get_event_loop()` will return the created instance.

However, you can do with one call. Following example closes implicitly created asyncio event loop and install a new one:

```
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 Services

The main thing that an `entrypoint` does is start and gracefully stop services.

The service concept within this library means a class derived from the `aiosmic.Service` class and implementing the `async def start(self) -> None:` method and optionally the `async def stop(self, exc: Optional[Exception]) -> None` method.

The concept of stopping a service is not necessarily is pressing `Ctrl+C` keys by user, it's actually just exiting the `entrypoint` context manager.

The example below shows what your service might look like:

```
from aiomisc import entrypoint, Service

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

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


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

The entry point can start as many instances of the service as it likes, and all of them will start concurrently.

There is also a way if the `start` method is a payload for a service, and then there is no need to implement the stop method, since the running task with the `start` function will be canceled at the stop stage. But in this case, you will have to notify the `entrypoint` that the initialization of the service instance is complete and it can continue.

Like this:

```
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()
```

Note: The `entrypoint` passes control to the body of the context manager only after all service instances have started. As mentioned above, a start is considered to be the completion of the `start` method or the setting of an start event with `self.start_event.set()`.

The whole power of this library is in the set of already implemented or abstract services. Such as: *AIOHTTPService*, *ASGIService*, *TCPServer*, *UDPServer*, *TCPClient*, *PeriodicService*, *CronService* and so on.

Unfortunately in this section it is not possible to pay more attention to this, please pay attention to the *Tutorial* section section, there are more examples and explanations, and of cource you always can find out an answer on the *API Reference* or in the source code. The authors have tried to make the source code as clear and simple as possible, so feel free to explore it.

CHAPTER

THREE

VERSIONING

This software follows Semantic Versioning

Summary: it's given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backwards compatible manner
- PATCH version when you make backwards compatible bug fixes
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

In this case, the package version is assigned automatically with poem-plugins, it using on the tag in the repository as a major and minor and the counter, which takes the number of commits between tag to the head of branch.

CHAPTER

FOUR

HOW TO DEVELOP?

This project, like most open source projects, is developed by enthusiasts, you can join the development, submit issues, or send your merge requests.

In order to start developing in this repository, you need to do the following things.

Should be installed:

- Python 3.7+ as `python3`
- Installed Poetry as `poetry`

For setting up developer environment just execute:

```
# 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 Tutorial

`aiomisc` is a Python library that provides a set of utilities for building asynchronous services. It allows you to split your program into smaller, independent services that can run concurrently, improving the overall performance and scalability of your application.

The main approach in this library is to split your program into independent services that can work concurrently in asynchronous mode. The library also provides a set of ready-to-use services with pre-written start and stop logic.

The vast majority of functions and classes are written in such a way that they can be used in a program that was not originally designed according to the principles outlined in this manual. This means that if you don't plan to modify your code too much, but only use a few useful functions or classes, then everything should work.

Overall, aiomisc is a powerful tool for developers looking to build efficient and scalable asynchronous services in Python.

4.1.1 Services

If you want to run a tcp or web server, you will have to write something like this:

```
import asyncio

async def example_async_func():
    # do my async business logic
    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())
# Continue running all background tasks
loop.run_forever()
```

In order to start or stop an async programs, usually using function `asyncio.run(example_async_func())` which available since Python 3.7. This function takes an instance of a coroutine and cancels all still running tasks before returning a result. To continue executing the code indefinitely, you can perform the following trick:

```
import asyncio

async def example_async_func():
    # do my async business logic
    await init_db()
    await init_cache()
    await start_http_server()
    await start_metrics_server()

    # Make future which never be done
    # using instead loop.run_forever()
    await asyncio.Future()

asyncio.run(example_async_func())
```

When the user presses *Ctrl+C*, the program simply terminates, but if you want to explicitly free up some resources, for example, close database connections or rolling back incomplete transactions, then you have to do something like this:

```
import asyncio

async def example_async_func():
    try:
        # do my async business logic
        await init_db()
        await init_cache()
        await start_http_server()
        await start_metrics_server()

        # Make future which never be done
        # using instead loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
```

(continues on next page)

(continued from previous page)

```
        # Do this block when SIGTERM has been received
        pass
    finally:
        # Do this block on exit
        ...

asyncio.run(example_async_func())
```

It is a good solution because it is implemented without any 3rd-party libraries. When your program starts to grow, you will probably want to optimize the startup time in a simple way, namely to do all initialization competitively. At first glance it seems that this code will solve the problem:

```
import asyncio

async def example_async_func():
    try:
        # do my async business logic
        await asyncio.gather(
            init_db(),
            init_cache(),
            start_http_server(),
            start_metrics_server(),
        )

        # Make future which never be done
        # using instead loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
        # Do this block when SIGTERM has been received
        pass
    finally:
        # Do this block on exit
        ...

asyncio.run(example_async_func())
```

But if suddenly some part of the initialization does not go according to plan, then you somehow have to figure out what exactly went wrong, so with concurrent execution, the code will no longer be as simple as in this example.

And in order to somehow organize the code, you should make a separate function that will contain the `try/except/finally` block and contain error handling.

```
import asyncio

async def init_db():
    try:
        # initialize connection
    finally:
        # close connection
        ...

async def example_async_func():
    try:
```

(continues on next page)

(continued from previous page)

```
        # do my async business logic
        await asyncio.gather(
            init_db(),
            init_cache(),
            start_http_server(),
            start_metrics_server(),
        )

        # Make future which never be done
        # using instead loop.run_forever()
        await asyncio.Future()
    except asyncio.CancelledError:
        # Do this block when SIGTERM has been received
        # TODO: shutdown all things correctly
        pass
    finally:
        # Do this block on exit
        ...

asyncio.run(example_async_func())
```

And now if the user presses Ctrl+C, you need to describe the shutdown logic again, but now in the `except` block.

In order to describe the logic of starting and stopping in one place, as well as testing in one single way, there is a `Service` abstraction.

The service is an abstract base class with mandatory `start()` and optional `stop()` methods.

The service can operate in two modes. The first is when the `start()` method runs forever, then you do not need to implement a `stop()`, but you need to report that the initialization is successfully completed by setting `self.start_event.set()`.

```
import asyncio
import aiomisc


class InfinityService(aiomisc.Service):
    async def start(self):
        # Service is ready
        self.start_event.set()

        while True:
            # do some stuff
            await asyncio.sleep(1)
```

In this case, stopping the service will consist in the completion of the coroutine that was created by `start()`.

The second method is an explicit description of the way to `start()` and `stop()`.

```
import asyncio
import aiomisc
from typing import Any
```

(continues on next page)

(continued from previous page)

```
class OrdinaryService(aiomisc.Service):
    async def start(self):
        # do some stuff
        ...

    async def stop(self, exception: Exception = None) -> Any:
        # do some stuff
        ...
```

In this case, the service will be started and stopped once.

4.1.2 Service configuration

The `Service` is a metaclass, it handles the special attributes of classes inherited from it at on the their declaration stage.

Here is a simple imperative example of how service initialization can be extended through inheritance.

```
from typing import Any
import aiomisc

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

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

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

# python hello.py
# <<< Hello world
# <<< Hello Guido
```

In fact, you can do nothing of this, since the Service metaclass sets all the passed keyword parameters to self by default.

```
import aiomisc

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

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

with aiomisc.entrypoint(
    HelloService(),
    HelloService(name="Guido")
```

(continues on next page)

(continued from previous page)

```
) as loop:
    pass

# python hello.py
# <<< Hello world
# <<< Hello Guido
```

If a special class property `__required__` is declared, then the service will required for the user to declare these named parameters.

```
import aiomisc

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

    name: str
    title: str

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

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

Also a very useful special class attribute is `__async_required__`. It is useful for writing base classes, in general. This contains the tuple of method names that must be declared asynchronous explicitly (via `async def`).

```
import aiomisc

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

    name: str
    title: str

    async def greeting(self) -> str:
        await asyncio.sleep(0.1)
        return f"Hello {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(
```

(continues on next page)

(continued from previous page)

```
    HelloService(name="Guido", title="mr."),
    HelloEmojiService(name="", title="")
) as loop:
    pass

# Hello mr. Guido
#
```

If the inheritor declares these methods differently, there will be an error at the class declaration stage.

```
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

In some cases, you need to execute some asynchronous code before the service starts, for example, to pass a database connection to the service instance. Or if you want to use one instance of some entity for several services.

For such complex configurations, there is aiomisc-dependency plugin which is distributed as a independent separate package.

Look at the examples in the documentation, aiomisc-dependency are transparently integrates with the `entrypoint`.

4.1.4 `entrypoint`

So the service abstraction is declared, what's next? `asyncio.run` does not know how to work with them, calling them manually has not become easier, what can this library offer here?

Probably the most magical, complex, and at the same time quite well-tested code in the library is `entrypoint`. Initially, the idea of `entrypoint` was to get rid of the routine: setting up logs, setting up a thread pool, as well as starting and stopping services correctly.

Lets check an example:

```
import asyncio
import aiomisc

...

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

In this example, we will launch the two services described above and continue execution until the user interrupts them. Next, thanks to the context manager, we correctly terminate all instances of services.

Note: Entrypoint calls all the `start()` methods in all services concurrently, and if at least one of them fails, then all services will be stopped.

As mentioned above I just wanted to remove a lot of routine, let's look at the same example, just pass all the default parameters to the `entrypoint` explicitly.

```
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()
```

Let's not describe what each parameter does. But in general, `entrypoint` has create an event-loop, a four threads pool, set it for the current event-loop, has configure a colored logger with buffered output, and launched two services.

You can also run the `entrypoint` without services, just configure logging and so on.:

```
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())
```

It is also worth paying attention to the `aiomisc.run`, which is similar by its purpose to `asyncio.run` while supporting the start and stop of services and so on.

```
import asyncio
import logging
import aiomisc
```

(continues on next page)

(continued from previous page)

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


aiomisc.run(
    # the first argument
    # is a main coroutine
    sleep_and_exit(),
    # Other positional arguments
    # is service instances
    OrdinaryService(),
    InfinityService(),
    # keyword arguments will
    # be passed as well to the entrypoint
    log_level="info"
)
```

Note: As I mentioned above, the library contains lots of already realized abstract services that you can use in your project by simply implement several methods.

A full list of services and usage examples can be found on the on the *Services page*.

4.1.5 Executing code in thread or process-pools

As explained in working with threads section in official python documentation asyncio event loop starts thread pool.

This pool is needed in order to run, for example, name resolution and not blocks the event loop while low-level `gethostbyname` call works.

The size of this thread pool should be configured at application startup, otherwise you may run into all sorts of problems when this pool is too large or too small.

By default, the `entrypoint` creates a thread pool with size equal to the number of CPU cores, but not less than 4 and no more than 32 threads. Of course you can specify as you need.

`@aiomisc.threaded` decorator

The following recommendations for calling blocking functions in threads given in working with threads section in official Python documentation:

```
import asyncio

def blocking_io():
    # File operations (such as logging) can block the event loop.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

async def main():
```

(continues on next page)

(continued from previous page)

```
    loop = asyncio.get_running_loop()
    result = await loop.run_in_executor(None, blocking_io)

asyncio.run(main())
```

This library provides a very simple way to do the same:

```
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())
```

As you can see in this example, it is enough to wrap the function with a decorator `aiomisc.threaded`, after that it will return an awaitable object, but the code inside the function will be sent to the default thread pool.

`@aiomisc.threaded_separate` decorator

If the blocking function runs for a long time, or even indefinitely, in other words, if the cost of creating a thread is insignificant compared to the workload, then you can use the decorator `aiomisc.threaded_separate`.

The decorator starts a new thread not associated with any pool. he thread will be terminated after the function execution is done.

```
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(
        "the hash is",
        await another_one_useless_coin_miner()
    )

aiomisc.run(main())
```

Note: This approach allows you not to occupy threads in the pool for a long time, but at the same time does not limit

the number of created threads in any way.

More examples you can be found in *Working with threads*.

`@aiomisc.threaded_iterable` decorator

If a generator needs to be executed in a thread, there are problems with synchronization of the thread and the eventloop. This library provides a custom decorator designed to turn a synchronous generator into an asynchronous one.

This is very useful if, for example, a queue or database driver has written synchronous, but you want to use it efficiently in asynchronous code.

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

aiomisc.run(main())
```

Under the hood, this decorator returns a special object that has a queue, and asynchronous iterator interface provides access to that queue.

You should always specify the `max_size` parameter, which limits the size of this queue and prevents threaded code from sending too much items to asynchronous code, in case the asynchronous iteration in case the asynchronous iteration slacking.

Conclusion

On this we need to finish this tutorial, I hope everything was clear here, and you learned a lot of useful things for yourself. A full description of the remaining services is presented in the *Modules* section, or in the source code. The authors have tried to make the source code as clear and simple as possible, so feel free to explore it.

4.2 Modules

4.2.1 entrypoint

In the generic case, the entrypoint helper creates an event loop and cancels already running coroutines on exit.

```
import asyncio
import aiomisc

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

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

Complete example:

```
import asyncio
import aiomisc
import logging
import signal

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

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

Running entrypoint from async code

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

(continues on next page)

(continued from previous page)

```
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

    # returns an entrypoint instance because event-loop
    # already running and might be get via 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())
```

Dynamic running of services

Sometimes it is not enough to add services to the entrypoint at the start, or it is not possible to get the service parameters before the start of the event-loop. In this case it is possible to start services after the event-loop has started, this feature available from version 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)
```

(continues on next page)

(continued from previous page)

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


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

Configuration from environment

Module support configuration from environment variables:

- `AIOMISC_LOG_LEVEL` - default logging level
- `AIOMISC_LOG_FORMAT` - default log format
- `AIOMISC_LOG_DATE_FORMAT` - default logging date format
- `AIOMISC_LOG_CONFIG` - should logging be configured
- `AIOMISC_LOG_FLUSH` - interval between logs flushing from buffer
- `AIOMISC_LOG_BUFFERING` - should logging be buffered
- `AIOMISC_LOG_BUFFER_SIZE` - maximum log buffer size
- `AIOMISC_POOL_SIZE` - thread pool size
- `AIOMISC_USE_UVLOOP` - should use uvloop when it available, `0` to disable
- `AIOMISC_SHUTDOWN_TIMEOUT` - If, after receiving the signal, the program does not terminate within this timeout, a force-exit occurs.

4.2.2 `run()` shortcut

`aiomisc.run()` - it's the short way to create and destroy `aiomisc.entrypoint`. It's very similar to `asyncio.run()` but handle `Service`'s and other `entrypoint`'s kwargs.

```
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 Logging configuration

`entrypoint` accepts `log_format` argument with a specific set of formats, in which logs will be written to stderr:

- `stream` - Python's default logging handler
- `color` - logging with *colorlog* module
- `json` - json structure per each line
- `syslog` - logging using stdlib *logging.handlers.SysLogHandler*
- `plain` - just log messages, without date or level info
- `journald` - available only when *logging-journald* module has been installed.
- `rich`/`rich_tb` - available only when *rich* module has been installed. `rich_tb` it's the same as `rich` but with fully expanded tracebacks.

Additionally, you can specify log level using `log_level` argument and date format using `log_date_format` parameters.

An `entrypoint` will call `aiomisc.log.basic_config` function implicitly using passed `log_*` parameters. Alternatively you can call `aiomisc.log.basic_config` function manually passing it already created eventloop.

However, you can configure logging earlier using `aiomisc_log.basic_config`, but you will lose log buffering and flushing in a separate thread. This function is what is actually called during the logging configuration, the `entrypoint` passes a wrapper for the handler there to flush it into the separate thread.

```
import logging

from aiomisc_log import basic_config


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

If you want to configure logging before the `entrypoint` is started, for example after the arguments parsing, it is safe to configure it twice (or more).

```
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())
```

Sometimes you want to configure logging manually, the following example demonstrates how to do this:

```
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

    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",
        # Maximum 100 files by 10 megabytes
        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,
        # Wrapping all handlers in separate streams will not block the
```

(continues on next page)

(continued from previous page)

```
        # event-loop even if gzip takes a long time to open the
        # file.
        handlers=map(
            aiomisc.log.wrap_logging_handler,
            (gzip_handler, stream_handler)
        )
    )
    loop.run_until_complete(main())
```

4.2.4 Services

`Services` is an abstraction to help organize lots of different tasks in one process. Each service must implement `start()` method and can implement `stop()` method.

Service instance should be passed to the `entrypoint`, and will be started after the event loop has been created.

Note: Current event-loop will be set before `start()` method called. The event loop will be set as current for this thread.

Please avoid using `asyncio.get_event_loop()` explicitly inside `start()` method. Use `self.loop` instead:

```
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()

      event.set()
      # Start service task
      await asyncio.sleep(3600)


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

Method `start()` creates as a separate task that can run forever. But in this case `self.start_event.set()` should be called for notifying `entrypoint`.

During graceful shutdown method `stop()` will be called first, and after that, all running tasks will be canceled (including `start()`).

This package contains some useful base classes for simple services writing.

TCPServer

TCPServer - it's a base class for writing TCP servers. Just implement `handle_client(reader, writer)` to use it.

```
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):
    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))
```

UDPServer

UDPServer - it's a base class for writing UDP servers. Just implement `handle_datagram(data, addr)` to use it.

```
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()
```

TLSServer

TLSServer - it's a base class for writing TCP servers with TLS. Just implement `handle_client(reader, writer)` to use it.

```
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,
)

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

TCPClient

TCPClient - it's a base class for writing TCP clients. Just implement `handle_connection(reader, writer)` to use it.

```
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"
```

(continues on next page)

(continued from previous page)

```
        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

TLSClient - it's a base class for writing TLS clients. Just implement `handle_connection(reader, writer)` to use it.

```
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()
```

(continues on next page)

(continued from previous page)

```
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

RobustTCPClient - it's a base class for writing TCP clients with auto-reconnection when connection lost. Just implement `handle_connection(reader, writer)` to use it.

```
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()
```

(continues on next page)

(continued from previous page)

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

RobustTLSClient

RobustTLSClient - it's a base class for writing TLS clients with auto-reconnection when connection lost. Just implement `handle_connection(reader, writer)` to use it.

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

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',
```

(continues on next page)

(continued from previous page)

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

PeriodicService

`PeriodicService` runs `PeriodicCallback` as a service and waits for the running callback to complete on the stop method. You need to use `PeriodicService` as a base class and override `callback` async coroutine method.

Service class accepts required `interval` argument - periodic interval in seconds and optional `delay` argument - periodic execution delay in seconds (0 by default).

```
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)  # once per hour

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

CronService

`CronService` runs `CronCallback's` as a service and waits for running callbacks to complete on the stop method.

It's based on croniter. You can register async coroutine method with `spec` argument - cron like format:

Warning: requires installed croniter:

```
pip install croniter
```

or using extras:

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

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


```

(continues on next page)

(continued from previous page)

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

service = CronService()
service.register(callback, spec="0 * * * *") # every hour at zero minutes

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

You can also inherit from `CronService`, but remember that callback registration should be proceeded before start

```
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()
```

Multiple services

Pass several service instances to the `entrypoint` to run all of them. After exiting the entrypoint service instances will be gracefully shut down.

```
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())
```

(continues on next page)

(continued from previous page)

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


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

Configuration

`Service` metaclass accepts all kwargs and will set it to `self` as attributes.

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


class LoggingService(Service):
    # required kwargs
    __required__ = frozenset({'name'})

    # default value
    delay: int = 1

    async def start(self):
        self.start_event.set()
        while True:
            # attribute ``name`` from kwargs
            # must be defined when instance initializes
            print('Hello from service', self.name)

            # attribute ``delay`` from kwargs
            await asyncio.sleep(self.delay)

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


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

aiohttp service

Warning: requires installed aiohttp:

```
pip install aiohttp
```

or using extras:

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

aiohttp application can be started as a service:

```
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()
```

Class `AIOHTTPSSLService` is similar to `AIOHTTPService` but creates an HTTPS server. You must pass SSL-required options (see `TLSServer` class).

asgi service

Warning: requires installed aiohttp-asgi:

```
pip install aiohttp-asgi
```

or using extras:

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

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

```
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()
```

Class `ASGIHTTPSSLService` is similar to `ASGIHTTPService` but creates HTTPS server. You must pass SSL-required options (see `TLSServer` class).

uvicorn service

Warning: requires installed uvicorn:

```
pip install uvicorn
```

or using extras:

```
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

This is an example of a GRPC service which is defined in a file and loads a *hello.proto* file without code generation, this example is one of the examples from *grpcio*, the other examples will work as expected.

Proto definition:

```
syntax = "proto3";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Service initialization example:

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

(continues on next page)

(continued from previous page)

```
    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)
```

Memory Tracer

Simple and useful service for logging large python objects allocated in memory.

```
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())
```

Output example:

```
[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

Simple service for profiling. Optional *path* argument can be provided to dump complete profiling data, which can be later used by, for example, snakeviz. Also can change ordering with the *order* argument ("cumulative" by default).

```
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())
```

Output example:

```
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 service

Simple service for sending unhandled exceptions to the sentry service instance.

Simple example:

```
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:
```

(continues on next page)

(continued from previous page)

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


raven_sender = RavenSender(
    sentry_dsn=(
        "https://583ca3b555054f80873e751e8139e22a@o429974.ingest.sentry.io/"
        "5530251"
    ),
    client_options=dict(
        # Got environment variable SENTRY_NAME by default
        name="example-from-aiomisc",
        # Got environment variable SENTRY_ENVIRONMENT by default
        environment="simple_example",
        # Got environment variable SENTRY_RELEASE by default
        release=__version__,
    )
)


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

Full configuration:

```
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(
        # Got environment variable SENTRY_NAME by default
```

(continues on next page)

(continued from previous page)

```
        name="",
        # Got environment variable SENTRY_ENVIRONMENT by default
        environment="full_example",
        # Got environment variable SENTRY_RELEASE by default
        release=__version__,

        # Default options values
        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())
```

You will find the full specification of options in the Raven documentation.

SDWatchdogService

Ready to use service just adding to your entrypoint and notifying SystemD service watchdog timer.

This can be safely added at any time, since if the service does not detect systemd-related environment variables, then its initialization is skipped.

Example of python file:

```
import logging
from time import sleep

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


if __name__ == '__main__':
    with entrypoint(SDWatchdogService()) as loop:
        pass
```

Example of systemd service file:

```
[Service]
# Activating the notification mechanism
Type=notify

# Command which should be started
ExecStart=/home/mosquito/.venv/aiomisc/bin/python /home/mosquito/scratch.py

# The time for which the program must send a watchdog notification
WatchdogSec=5

# Kill the process if it has stopped responding to the watchdog timer
WatchdogSignal=SIGKILL

# The service should be restarted on failure
Restart=on-failure

# Try to kill the process instead of cgroup
KillMode=process

# Trying to stop service properly
KillSignal=SIGINT

# Trying to restart service properly
RestartKillSignal=SIGINT

# Send SIGKILL when timeouts are exceeded
FinalKillSignal=SIGKILL
SendSIGKILL=yes
```

ProcessService

A base class for launching a function by a separate system process, and by termination when the parent process is stopped.

```
from typing import Dict, Any

import aiomisc.service

# Fictional miner implementation
from .my_miner import Miner


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

    def in_process(self) -> Any:
        if self.bitcoin:
            miner = Miner(kind="bitcoin")
        elif self.monero:
            miner = Miner(kind="monero")
```

(continues on next page)

(continued from previous page)

```
        elif self.dogiecoin:
            miner = Miner(kind="dogiecoin")
        else:
            # Nothing to do
            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()
```

RespawningProcessService

A base class for launching a function by a separate system process, and by termination when the parent process is stopped, It's pretty like *ProcessService* but have one difference when the process unexpectedly exited this will be respawned.

```
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 Abstract connection pool

`aiomisc.PoolBase` is an abstract class for implementing user-defined connection pool.

Example for `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 Context

Services can require each other's data. In this case, you should use `Context`.

`Context` is a repository associated with the running `entrypoint`.

`Context`-object will be created when `entrypoint` starts and linked to the running event loop.

Cross-dependent services might await or set each other's data via the context.

For service instances `self.context` is available since `entrypoint` started. In other cases `get_context()` function returns current context.

```python
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):
        # querying from remote server
        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
```

Note: It's not a silver bullet. In base case services can be configured by passing kwargs to the service `__init__` method.

4.2.7 @aiomisc.timeout

Decorator that ensures the execution time limit for the decorated function is met.

```
from aiomisc import timeout

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

4.2.8 @aiomisc.asyncbackoff

`asyncbackoff` it's a decorator that helps you guarantee maximal async function execution and retrying policy.

The main principle might be described in five rules:

- function will be canceled when executed longer than `deadline` (if specified)
- function will be canceled when executed longer than `attempt_timeout` (if specified) and will be retried
- Reattempts performs after `pause` seconds (if specified, default is `0`)
- Reattempts will be performed not more than `max_tries` times.
- `giveup` argument is a function that decides should give up the reattempts or continue retrying.

All these rules work at the same time.

Arguments description:

- `attempt_timeout` is maximum execution time for one execution attempt.
- `deadline` is maximum execution time for all execution attempts.
- `pause` is the time gap between execution attempts.
- `exceptions` retrying when these exceptions were raised.
- `giveup` (keyword only) is a predicate function that can decide by a given exception if we should continue to do retries.
- `max_tries` (keyword only) is maximum count of execution attempts (>= 1).

Decorator that ensures that `attempt_timeout` and `deadline` time limits are met by decorated function.

In case of exception, the function will be called again with similar arguments after `pause` seconds.

Position arguments notation:

```
import aiomisc

attempt_timeout = 0.1
deadline = 1
pause = 0.1

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


@aiomisc.asyncbackoff(0.1, 1, 0.1)
```

(continues on next page)

(continued from previous page)

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


# Passing exceptions for handling
@aiomisc.asyncbackoff(0.1, 1, 0.1, TypeError, RuntimeError, ValueError)
async def db_fetch(data: dict):
    ...
```

Keyword arguments notation:

```
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):
    ...


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


# Will be retried no more than 2 times (3 tries total)
@aiomisc.asyncbackoff(attempt_timeout=0.5, deadline=1,
                      pause=0.1, max_tries=3,
                      exceptions=[TypeError, RuntimeError, ValueError])
async def db_fetch(data: dict):
    ...


# Will be retried only on connection abort (on POSIX systems)
@asyncbackoff(attempt_timeout=0.5, deadline=1, pause=0.1,
              exceptions=[OSError],
              giveup=lambda e: e.errno != errno.ECONNABORTED)
async def db_fetch(data: dict):
    ...
```

4.2.9 asyncretry

Shortcut of `asyncbackoff(None, None, 0, *args, **kwargs)`. Just retries `max_tries` times.

Note: By default will be retry when any Exception. It's very simple and useful in generic cases, but you should specify an exception list when you're wrapped functions calling hundreds of times per second, cause you have a risk be the reason for denial of service in case your function calls remote service.

```
from aiomisc import asyncretry

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

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

4.2.10 Circuit Breaker

Circuit breaker is a design pattern used in software development. It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure, or unexpected system difficulties.

The following example demonstrates the simple usage of the current implementation of `aiomisc.CircuitBreaker`. An instance of `CircuitBreaker` collecting function call statistics. That contains counters mapping with successful and failed function calls. Function calls must be wrapped by the *CircuitBreaker.call* the method to gather it.

Usage example:

```
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:
        # Using as context manager
        with request.app["github_cb"].context():
            url = 'https://api.github.com/gists/public'
            async with session.get(url) as response:
                data = await response.text()

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


class API(AIOHTTPService):
    async def create_application(self):
```

(continues on next page)

(continued from previous page)

```
        app = web.Application()
        app.add_routes([web.get('/', public_gists)])

        # When 30% errors in 20 seconds
        # Will be broken for 5 seconds
        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())
```

The *CircuitBreaker* object might be one of three states:

- **PASSING**
- **BROKEN**
- **RECOVERING**

PASSING means all calls will be passed as is and statistics will be gathered. The next state will be determined after collecting statistics for `passing_time` seconds. If an effective error ratio is greater than *error_ratio* then the next state will be set to **BROKEN**, otherwise, it will remain unchanged.

BROKEN means the wrapped function won't be called and `CircuitBroken` an exception will be raised instead. **BROKEN** state will be kept for `broken_time` seconds.

Note: `CircuitBroken` exception is a consequence of **BROKEN** or **RECOVERY** state and never be accounted for in the statistic.

After that, it changes to **RECOVERING** state. While in that state, a small sample of the wrapped function calls will be executed and statistics will be gathered. If the effective error ratio after `recovery_time` is lower than `error_ratio` then the next state will be set to **PASSING**, and otherwise - to **BROKEN**.

Argument `exception_inspector` is a function that is called whenever an exception from the list of monitored exceptions occurs. When `False` will be returned, this exception will be ignored.

4.2.11 cutout

Decorator for `CircuitBreaker` which wrapping functions.

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


# When 20% errors in 30 seconds
# Will be broken on 30 seconds
@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


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

4.2.12 @aiomisc.aggregate

Parametric decorator that aggregates multiple (but no more than `max_count` defaulting to `None`) single-argument executions (`res1 = await func(arg1)`, `res2 = await func(arg2)`, ...) of an asynchronous function with variadic positional arguments (`async def func(*args, pho=1, bo=2) -> Iterable`) into its single execution with multiple positional arguments (`res1, res2, ... = await func(arg1, arg2, ...)`) collected within a time window `leeway_ms`. It offers a trade-off between latency and throughput.

If `func` raises an exception, then, all of the aggregated calls will propagate the same exception. If one of the aggregated calls gets canceled during the `func` execution, then, another will try to execute the `func`.

This decorator may be useful if the `func` executes slow IO-tasks, is frequently called, and using cache is not a good option. As a toy example, assume that `func` fetches a record from the database by user ID and it is called during each request to our service. If it takes 100 ms to fetch a record and the load is 1000 RPS, then, with a 10% increase of the delay (to 110 ms), it may decrease the number of requests to the database by 10 times (to 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())
```

To employ a more low-level approach one can use *aggregate_async* instead. In this case, the aggregating function accepts *Arg* parameters, each containing *value* and *future* attributes. It is responsible for setting the results of execution for all the futures (instead of returning values).

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


@aggregate_async(leeway_ms=10, max_count=2)
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 asynchronous file operations

Asynchronous files operations including support for data compression on the fly. Based on the thread pool under the hood.

```
import aiomisc
import tempfile
from pathlib import Path


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

        # Some tools, such as mypy, will not be able to infer the type
        # from the `async_open` function based on the `b` character passed
        # to the mode.
        # We'll have to tell the type here explicitly.
        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())
```

This is the way to working with files based on threads. It's very similar to aiofiles project and same limitations.

Of course, you can use aiofile project for this. But it's not a silver bullet and has OS API-related limitations.

In general, for light loads, I would advise you to adhere to the following rules:

- If reading and writing small or big chunks from files with random access is the main task in your project, use aiofile.
- Otherwise use this module or aiofiles
- If the main task is to read large chunks of files for processing, both of the above methods are not optimal cause you will switch context each IO operation, it's often suboptimal for file cache and you will be lost execution time for context switches. In case for thread-based IO executor implementation thread context switches cost might be more expensive than IO operation time in summary.

  Just try pack all blocking staff in separate functions and call it in a thread pool, see the example bellow:

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

(continues on next page)

(continued from previous page)

```
@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))
   )
```

In the fly compression

To enable compression, you need to pass the *compression* argument to the *async_open* function.

Supported compressors:

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

An example of usage:

```
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 Working with threads

Wraps blocking function and run it in the different thread or thread pool.

contextvars support

All following decorators and functions support `contextvars` module, from PyPI for python earlier 3.7 and builtin a standard library for python 3.7.

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


user_id = contextvars.ContextVar("user_id")

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:
```

(continues on next page)

(continued from previous page)

```
        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())
```

Example output:

```
6621
33012
1590
45008
56844
```

Note: contextvars has different use cases then Context class. ``contextvars`` are applicable for passing context variables through the execution stack but created task can not change parent context variables because ``contextvars`` create lightweight copy. ``Context`` class allows it because does not copy context variables.

@aiomisc.threaded

Wraps blocking function and run it in the current thread pool.

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


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


async def main():
    # Running in parallel
    await asyncio.gather(
```

(continues on next page)

(continued from previous page)

```
        blocking_function(),
        blocking_function(),
    )


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

In case the function is a generator function `@threaded` decorator will return `IteratorWrapper` (see Threaded generator decorator).

@aiomisc.threaded_separate

Wraps blocking function and run it in a new separate thread. Highly recommended for long background tasks:

```
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("Exitting")


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

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

    # Running in parallel
    await asyncio.gather(
        blocking_function(),
        # New thread will be spawned
        long_blocking_function(stop_event),
    )


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

Threaded iterator decorator

Wraps blocking generator function and run it in the current thread pool or on a new separate thread.

Following example reads itself file, chains hashes of every line with the hash of the previous line and sends hash and content via TCP:

```
import asyncio
import hashlib

import aiomisc

# My first blockchain

@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()


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

Run `netcat` listener in the terminal and run this example

```
$ netcat -v -l -p 2233
Connection from 127.0.0.1:54734
dc80feba2326979f8976e387fbbc8121   import asyncio
78ec3bcb1c441614ede4af5e5b28f638   import hashlib
b7df4a0a4eac401b2f835447e5fc4139
f0a94eb3d7ad23d96846c8cb5e327454   import aiomisc
0c05dde8ac593bad97235e6ae410cb58
e4d639552b78adea6b7c928c5ebe2b67   # My first blockchain
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()
```

(continues on next page)

(continued from previous page)

```
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())
```

You should use async context managers in the case when your generator works infinity, or you have to await the `.close()` method when you avoid context managers.

```python
import asyncio
import aiomisc


# Set 2 chunk buffer
@aiomisc.threaded_iterable(max_size=2)
def urandom_reader():
    with open('/dev/urandom', "rb") as fp:
        while True:
            yield fp.read(8)


# Infinity buffer on a separate thread
@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()

    # Feed white noise
    gen = urandom_reader()
    counter = 0
```

(continues on next page)

(continued from previous page)

```
    async for line in gen:
        writer.write(line)
        counter += 1

        if counter == 10:
            break

    await writer.drain()

    # Stop running generator
    await gen.close()

    # Using context manager
    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

Run iterables on dedicated thread pool:

```
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():
    # create a new thread pool
    pool = concurrent.futures.ThreadPoolExecutor(1)
    wrapper = aiomisc.IteratorWrapper(
        urandom_reader,
        executor=pool,
        max_size=2
    )
```

(continues on next page)

(continued from previous page)

```
    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

Run iterables on a separate thread:

```
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():
    # create a new thread pool
    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())
```

(continues on next page)

(continued from previous page)

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

aiomisc.ThreadPoolExecutor

This is a fast thread pool implementation.

Setting as a default thread pool:

```
import asyncio
from aiomisc import ThreadPoolExecutor

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

Note: `entrypoint` context manager will set it by default.

`entrypoint`'s argument `pool_size` limits thread pool size.

---

aiomisc.sync_wait_coroutine

Functions running in thread can't call and wait for a result from coroutines by default. This function is the helper for send coroutine to the event loop and waits for it in the current thread.

```
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

This is a simple process pool executor implementation.

Example:

```
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),
            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 Utilities

select

In some cases, you should wait for only one of multiple tasks. `select` waits first passed awaitable object and returns the list of results.

```
import asyncio
import aiomisc


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

    loop.call_soon(event.set)
```

(continues on next page)

(continued from previous page)

```
    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())
```

Warning: When you don't want to cancel pending tasks pass `cancel=False` argument. In this case, you have to handle task completion manually or get warnings.

cancel_tasks

All passed tasks will be canceled and the task will be returned:

```
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

Decorator wraps function and returns a function that returns awaitable object. In case a function returns a future, the original future will be returned. In case then the function returns a coroutine, the original coroutine will be returned. In case than function returns a non-awaitable object, it's will be wrapped to a new coroutine that just returns this object. It's useful when you don't want to check function results before use it in `await` expression.

```
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

Bind socket and set `setblocking(False)` for just created socket. This detects `address` format and selects the socket family automatically.

```
from aiomisc import bind_socket

# IPv4 socket
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

Runs coroutine function periodically with user-defined strategy.

```
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:
```

(continues on next page)

(continued from previous page)

```
        FIRST_CALL = True
        # Delay 5 second if just started
        return 5

    # Delay 10 seconds if it is not a first call
    return 10


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

    periodic = RecurringCallback(callback)

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

The main purpose is this class is to provide ability to specify the asynchronous strategy function, which can be written very flexible.

Also, with the special exceptions, you can control the behavior of the started `RecurringCallback`.

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


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

    # Skip this attempt and wait 10 seconds
    raise StrategySkip(10)

    ...

    # Stop execution
    raise StrategyStop()
```

if the strategy function returns an incorrect value (not a number), or does not raise special exceptions, the recurring execution is terminated.

PeriodicCallback

Runs coroutine function periodically with an optional delay of the first execution. Uses `RecurringCallback` under the hood.

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


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

(continues on next page)

(continued from previous page)

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

    periodic = PeriodicCallback(periodic_function)

    # Wait 10 seconds and call it each second after that
    periodic.start(1, delay=10)

    loop.run_forever()
```

CronCallback

Warning: You have to install `croniter` package for use this feature:

```
pip install croniter
```

Or add extras when installing aiomisc:

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

Runs coroutine function with cron scheduling execution. Uses `RecurringCallback` under the hood.

```
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)

    # call it each second after that
    periodic.start(spec="* * * * * *")

    loop.run_forever()
```

4.2.17 WorkerPool

Python has the `multiprocessing` module with `Pool` class which implements a similar worker pool. The IPC in this case uses a completely synchronous communication method. This module reimplements the process-based worker pool but IPC is completely asynchronous on the caller side, meanwhile, workers in separate processes aren't asynchronous.

Example

This would be useful when you want to process the data in a separate process, and the input and output data are not very large. Otherwise, it will work fine, of course, but you would have to spend time transmitting the data over IPC.

A good example is parallel image processing. Of course, you can transfer bytes of images through the IPC of the working pool, but in general, case passing the file name to the worker will be better. The exception will be cases when the image payload is significantly smaller the 1KB for example.

Let's write a program that accepts images in JPEG format and creates thumbnails for this. In this case, you have a file with the original image, and you should generate the output path for the 'thumbnail' function.

Note: You have to install the Pillow image processing library to run this example.

Installing pillow with pip:

```
pip install Pillow
```

```
import asyncio
import sys
from multiprocessing import cpu_count
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),
```

(continues on next page)

(continued from previous page)

```
]


async def amain(path: Path):
    # Create directories
    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)

    # Create and run 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
                )

                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))
```

This example takes the image directory as the first command-line argument and creates directories for the thumbnails. After that, a `WorkerPool` is started with as many processes as the processor has cores.

The main process creates tasks for the workers, each task is a conversion of one file to one size, after which all tasks fall into the `WorkerPool` instance.

The `WorkerPool` processes the tasks concurrently, but only one job for one a worker at the same time.

4.2.18 Logging configuration

Default logging configuration might be configured by setting environment variables:

- *AIOMISC_LOG_LEVEL* - default logging level
- *AIOMISC_LOG_FORMAT* - default log format
- *AIOMISC_LOG_CONFIG* - should logging be configured
- *AIOMISC_LOG_FLUSH* - interval between logs flushing from buffer
- *AIOMISC_LOG_BUFFER* - maximum log buffer size

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

Color

Setting up colorized logs:

```
import logging
from aiomisc.log import basic_config


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

JSON

Setting up json logs:

```
import logging
from aiomisc.log import basic_config


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

JournalD

JournalD daemon for collecting logs. It's a part of the systemd. *aiomisc.basic_config* has support for using JournalD for store logs.

Note: This handler is the default when the program starting as a systemd service.

aiomisc.log.LogFormat.default() will returns *journald* in this case.

```
import logging
from aiomisc.log import basic_config

# Configure rich log handler
```

(continues on next page)

(continued from previous page)

```
basic_config(level=logging.INFO, buffered=False, log_format='journald')

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

Rich

Rich is a Python library for rich text and beautiful formatting in the terminal.

aiomisc.basic_config has support for using Rich as a logging handler. But it isn't dependency and you have to install Rich manually.

```
pip install rich
```

Note: This handler is the default when the *Rich* has been installed.

```
import logging
from aiomisc.log import basic_config

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

logging.info("Rich logger")

# Configure rich log handler with rich tracebacks display
basic_config(level=logging.INFO, buffered=False, log_format='rich_tb')

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

Disabled

Disable to configure logging handler. Useful when you want to configure your own logging handlers using *handlers=* argument.

```
import logging
from aiomisc.log import basic_config

# Configure rich log handler
basic_config(
    level=logging.INFO,
    log_format='disabled',
    handlers=[logging.StreamHandler()],
    buffered=False,
)

logging.info("Use default python logger for example")
```

Buffered log handler

Parameter *buffered=True* enables a memory buffer that flushes logs in a thread. In case the *handlers=* each will be buffered.

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


# Configure logging globally
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()

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

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

    # Wait for flush just for example
    loop.run_until_complete(asyncio.sleep(1))
```

Note: `entrypoint` accepts `log_format` parameter for configure it.

List of all supported log formats is available from `aiomisc.log.LogFormat.choices()`

4.2.19 Pytest plugin

Starting with version 17, pytest integration is distributed as a separate package aiomisc-pytest. See the documentation for the aiomisc-pytest module for further instructions.

4.2.20 Signal

You can register async callback functions for specific events of an entrypoint.

pre_start

`pre_start` signal occurs on entrypoint starting up before any service has started.

```
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` signal occurs next entrypoint starting up after all services have been started.

```
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` signal occurs on entrypoint shutdown before any service have been stopped.

```
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 signal occurs on entrypoint shutdown after all services have been stopped.

```
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 can be extended with plugins as separate packages. Plugins can enhance aiomisc by mean of *signals*.

To make your plugin discoverable by aiomisc you should add `aiomisc.plugins` entry to entry to `entry_points` argument of `setup` call in `setup.py` of a plugin.

```
# setup.py

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

If you use *pyproject.toml* you might define it like this:

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

Modules provided in `entry_points` should have `setup` function. These functions would be called by aiomisc and must contain signals connecting.

If the services are started dynamically, the attached functions will run every time the services are started and stopped, however, only services that are currently starting or stopping will be in the `services` parameter.

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

import aiomisc


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


async def hello(
```

(continues on next page)

(continued from previous page)

```
    *,
    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()

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

The following signals are available in total:

- `Entrypoint.PRE_START` - Will be called before *starting* services.
- `Entrypoint.PRE_STOP` - Will be called before *stopping* services.
- `Entrypoint.POST_START` - Will be called after services has been *started*.
- `Entrypoint.POST_STOP` - Will be called after services has been *stopped*.

List available plugins

To see a list of all available plugins, you can call from the command line `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
```

You can also change the behavior and output of the list of modules. To do this, there are the following flags:

```
$ 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
```

Here are some run examples.

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

This prints human-readable list of plugins and its descriptions.

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

This useful for `grep` or other pipelining tools.

The default prints both, the human-readable log to stderr and the list of plugins to stdout, so you can use this without options in a pipeline, and read the list to stderr.

4.2.22 Statistic counters

`aiomisc` contains internal statistic counters. You may read this by *aiomisc.get_statistics()* function.

Statistic instances create dynamically. You might set custom names for this using `statistic_name: Optional[str] = None` argument for compatible entities.

```
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())
```

This code will print something like this:

```
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 API Reference

4.3.1 aiomisc module

aiomisc.aggregate module

aiomisc.backoff module

aiomisc.circuit_breaker module

aiomisc.compat module

aiomisc.context module

aiomisc.counters module

aiomisc.cron module

aiomisc.entrypoint module

aiomisc.io module

aiomisc.iterator_wrapper module

aiomisc.log module

aiomisc.periodic module

aiomisc.plugins module

aiomisc.pool module

aiomisc.process_pool module

aiomisc.recurring module

aiomisc.signal module

aiomisc.thread_pool module

aiomisc.timeout module

aiomisc.utils module

aiomisc.worker_pool module

4.3.2 aiomisc_log module

aiomisc_log.formatter.color module

aiomisc_log.formatter.journald module

aiomisc_log.formatter.json module

aiomisc_log.formatter.rich.py module

aiomisc_log.enum module

4.3.3 aiomisc_worker module

aiomisc_worker.forking module

aiomisc_worker.process module

aiomisc_worker.process_inner module

aiomisc_worker.protocol module

aiomisc_worker.worker module