aiocache Documentation

Release 0.8.0

Manuel Miranda

Jan 23, 2018

Contents

CHAPTER 1

Installing

- `pip install aiocache`
- `pip install aiocache[redis]`
- `pip install aiocache[memcached]`
- `pip install aiocache[redis,memcached]`

CHAPTER 2

Usage

Using a cache is as simple as

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> from aiocache import SimpleMemoryCache
>>> cache = SimpleMemoryCache()
>>> loop.run_until_complete(cache.set('key', 'value'))
True
>>> loop.run_until_complete(cache.get('key'))
'value'
```

Here we are using the *SimpleMemoryCache* but you can use any other listed in *Caches*. All caches contain the same minimum interface which consists on the following functions:

- `add`: Only adds key/value if key does not exist. Otherwise raises ValueError.
- `get`: Retrieve value identified by key.
- `set`: Sets key/value.
- `multi_get`: Retrieves multiple key/values.
- `multi_set`: Sets multiple key/values.
- `exists`: Returns True if key exists False otherwise.
- `increment`: Increment the value stored in the given key.
- `delete`: Deletes key and returns number of deleted items.
- `clear`: Clears the items stored.
- `raw`: Executes the specified command using the underlying client.

You can also setup cache aliases like in Django settings:

```
import asyncio

from aiocache import caches, SimpleMemoryCache, RedisCache
```

```python
from aiocache.serializers import StringSerializer, PickleSerializer

caches.set_config({
    'default': {
        'cache': "aiocache.SimpleMemoryCache",
        'serializer': {
            'class': "aiocache.serializers.StringSerializer"
        }
    },
    'redis_alt': {
        'cache': "aiocache.RedisCache",
        'endpoint': "127.0.0.1",
        'port': 6379,
        'timeout': 1,
        'serializer': {
            'class': "aiocache.serializers.PickleSerializer"
        },
        'plugins': [
            {'class': "aiocache.plugins.HitMissRatioPlugin"},
            {'class': "aiocache.plugins.TimingPlugin"}
        ]
    }
})


async def default_cache():
    cache = caches.get('default')   # This always returns the same instance
    await cache.set("key", "value")

    assert await cache.get("key") == "value"
    assert isinstance(cache, SimpleMemoryCache)
    assert isinstance(cache.serializer, StringSerializer)


async def alt_cache():
    # This generates a new instance every time! You can also use `caches.create('alt
↪')`
    # or even `caches.create('alt', namespace="test", etc...)` to override extra args
    cache = caches.create(**caches.get_alias_config('redis_alt'))
    await cache.set("key", "value")

    assert await cache.get("key") == "value"
    assert isinstance(cache, RedisCache)
    assert isinstance(cache.serializer, PickleSerializer)
    assert len(cache.plugins) == 2
    assert cache.endpoint == "127.0.0.1"
    assert cache.timeout == 1
    assert cache.port == 6379
    await cache.close()


def test_alias():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(default_cache())
    loop.run_until_complete(alt_cache())

    cache = RedisCache()
    loop.run_until_complete(cache.delete("key"))
```

```
    loop.run_until_complete(cache.close())

    loop.run_until_complete(caches.get('default').close())


if __name__ == "__main__":
    test_alias()
```

In examples folder you can check different use cases:

- Sanic, Aiohttp and Tornado
- Python object in Redis
- Custom serializer for compressing data
- TimingPlugin and HitMissRatioPlugin demos
- Using marshmallow as a serializer
- Using cached decorator.
- Using multi_cached decorator.

CHAPTER 3

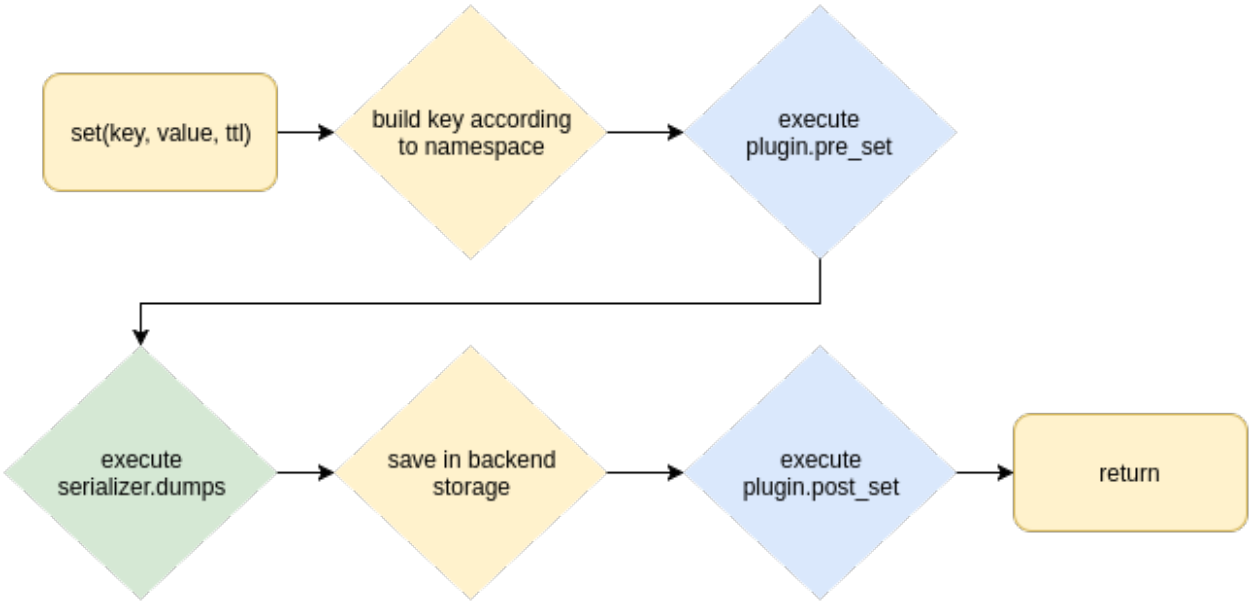
Contents

3.1 Caches

You can use different caches according to your needs. All the caches implement the same interface.

Caches are always working together with a serializer which transforms data when storing and retrieving from the backend. It may also contain plugins that are able to enrich the behavior of your cache (like adding metrics, logs, etc).

This is the flow of the `set` command:



Let's go with a more specific case. Let's pick Redis as the cache with namespace "test" and PickleSerializer as the serializer:

1. We receive `set("key", "value")`.

2. Hook `pre_set` of all attached plugins (none by default) is called.
3. "key" will become "test:key" when calling `build_key`.
4. "value" will become an array of bytes when calling `serializer.dumps` because of `PickleSerializer`.
5. the byte array is stored together with the key using `set` cmd in Redis.
6. Hook `post_set` of all attached plugins is called.

By default, all commands are covered by a timeout that will trigger an `asyncio.TimeoutError` in case of timeout. Timeout can be set at instance level or when calling the command.

The supported commands are:

- add
- get
- set
- multi_get
- multi_set
- delete
- exists
- increment
- expire
- clear
- raw

If you feel a command is missing here do not hesitate to open an issue

3.1.1 BaseCache

3.1.2 RedisCache

3.1.3 SimpleMemoryCache

3.1.4 MemcachedCache

3.2 Serializers

Serializers can be attached to backends in order to serialize/deserialize data sent and retrieved from the backend. This allows to apply transformations to data in case you want it to be saved in a specific format in your cache backend. For example, imagine you have your `Model` and want to serialize it to something that Redis can understand (Redis can't store python objects). This is the task of a serializer.

To use a specific serializer:

```
>>> from aiocache import SimpleMemoryCache
>>> from aiocache.serializers import PickleSerializer
cache = SimpleMemoryCache(serializer=PickleSerializer())
```

Currently the following are built in:

3.2.1 NullSerializer

3.2.2 StringSerializer

3.2.3 PickleSerializer

3.2.4 JsonSerializer

3.2.5 MessagePackSerializer

In case the current serializers are not covering your needs, you can always define your custom serializer as shown in `examples/serializer_class.py`:

```
import asyncio
import zlib

from aiocache import RedisCache
from aiocache.serializers import BaseSerializer


class CompressionSerializer(BaseSerializer):

    # This is needed because zlib works with bytes.
    # this way the underlying backend knows how to
    # store/retrieve values
    DEFAULT_ENCODING = None

    def dumps(self, value):
        print("I've received:\n{}".format(value))
        compressed = zlib.compress(value.encode())
        print("But I'm storing:\n{}".format(compressed))
        return compressed

    def loads(self, value):
        print("I've retrieved:\n{}".format(value))
        decompressed = zlib.decompress(value).decode()
        print("But I'm returning:\n{}".format(decompressed))
        return decompressed


cache = RedisCache(serializer=CompressionSerializer(), namespace="main")


async def serializer():
    text = (
        "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt"
        "ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation"
        "ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in"
        "reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur"
        "sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit"
        "anim id est laborum.")
    await cache.set("key", text)
```

```
    print("-----------------------------------")
    real_value = await cache.get("key")
    compressed_value = await cache.raw("get", "main:key")
    assert len(compressed_value) < len(real_value.encode())


def test_serializer():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(serializer())
    loop.run_until_complete(cache.delete("key"))
    loop.run_until_complete(cache.close())


if __name__ == "__main__":
    test_serializer()
```

You can also use marshmallow as your serializer (`examples/marshmallow_serializer_class.py`):

```
import random
import string
import asyncio

from marshmallow import fields, Schema, post_load

from aiocache import SimpleMemoryCache
from aiocache.serializers import BaseSerializer


class RandomModel:
    MY_CONSTANT = "CONSTANT"

    def __init__(self, int_type=None, str_type=None, dict_type=None, list_type=None):
        self.int_type = int_type or random.randint(1, 10)
        self.str_type = str_type or random.choice(string.ascii_lowercase)
        self.dict_type = dict_type or {}
        self.list_type = list_type or []

    def __eq__(self, obj):
        return self.__dict__ == obj.__dict__


class MarshmallowSerializer(Schema, BaseSerializer):
    int_type = fields.Integer()
    str_type = fields.String()
    dict_type = fields.Dict()
    list_type = fields.List(fields.Integer())

    # marshmallow Schema class doesn't play nicely with multiple inheritance and won
↪'t call
    # BaseSerializer.__init__
    encoding = 'utf-8'

    def dumps(self, *args, **kwargs):
        # dumps returns (data, errors), we just want to save data
        return super().dumps(*args, **kwargs).data

    def loads(self, *args, **kwargs):
        # dumps returns (data, errors), we just want to return data
```

```
        return super().loads(*args, **kwargs).data

    @post_load
    def build_my_type(self, data):
        return RandomModel(**data)

    class Meta:
        strict = True


cache = SimpleMemoryCache(serializer=MarshmallowSerializer(), namespace="main")


async def serializer():
    model = RandomModel()
    await cache.set("key", model)

    result = await cache.get("key")

    assert result.int_type == model.int_type
    assert result.str_type == model.str_type
    assert result.dict_type == model.dict_type
    assert result.list_type == model.list_type


def test_serializer():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(serializer())
    loop.run_until_complete(cache.delete("key"))


if __name__ == "__main__":
    test_serializer()
```

By default cache backends assume they are working with `str` types. If your custom implementation transform data to bytes, you will need to set the class attribute `encoding` to `None`.

3.3 Plugins

Plugins can be used to enrich the behavior of the cache. By default all caches are configured without any plugin but can add new ones in the constructor or after initializing the cache class:

```
>>> from aiocache import SimpleMemoryCache
>>> from aiocache.plugins import TimingPlugin
cache = SimpleMemoryCache(plugins=[HitMissRatioPlugin()])
cache.plugins += [TimingPlugin()]
```

You can define your custom plugin by inheriting from *BasePlugin* and overriding the needed methods (the overrides NEED to be async). All commands have `pre_<command_name>` and `post_<command_name>` hooks.

Warning: Both pre and post hooks are executed awaiting the coroutine. If you perform expensive operations with the hooks, you will add more latency to the command being executed and thus, there are more probabilities of raising a timeout error.

A complete example of using plugins:

```
import asyncio
import random
import logging

from aiocache import SimpleMemoryCache
from aiocache.plugins import HitMissRatioPlugin, TimingPlugin, BasePlugin


logger = logging.getLogger(__name__)


class MyCustomPlugin(BasePlugin):

    async def pre_set(self, *args, **kwargs):
        logger.info("I'm the pre_set hook being called with %s %s" % (args, kwargs))

    async def post_set(self, *args, **kwargs):
        logger.info("I'm the post_set hook being called with %s %s" % (args, kwargs))


cache = SimpleMemoryCache(
    plugins=[HitMissRatioPlugin(), TimingPlugin(), MyCustomPlugin()],
    namespace="main")


async def run():
    await cache.set("a", "1")
    await cache.set("b", "2")
    await cache.set("c", "3")
    await cache.set("d", "4")

    possible_keys = ["a", "b", "c", "d", "e", "f"]

    for t in range(1000):
        await cache.get(random.choice(possible_keys))

    assert cache.hit_miss_ratio["hit_ratio"] > 0.5
    assert cache.hit_miss_ratio["total"] == 1000

    assert cache.profiling["get_min"] > 0
    assert cache.profiling["set_min"] > 0
    assert cache.profiling["get_max"] > 0
    assert cache.profiling["set_max"] > 0

    print(cache.hit_miss_ratio)
    print(cache.profiling)


def test_run():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
    loop.run_until_complete(cache.delete("a"))
    loop.run_until_complete(cache.delete("b"))
    loop.run_until_complete(cache.delete("c"))
    loop.run_until_complete(cache.delete("d"))


```

```
if __name__ == "__main__":
    test_run()
```

3.3.1 BasePlugin

3.3.2 TimingPlugin

3.3.3 HitMissRatioPlugin

3.4 Configuration

3.4.1 Cache aliases

The caches module allows to setup cache configurations and then use them either using an alias or retrieving the config explicitly. To set the config, call `caches.set_config`:

To retrieve a copy of the current config, you can use `caches.get_config` or `caches.get_alias_config` for an alias config.

Next snippet shows an example usage:

```
import asyncio

from aiocache import caches, SimpleMemoryCache, RedisCache
from aiocache.serializers import StringSerializer, PickleSerializer

caches.set_config({
    'default': {
        'cache': "aiocache.SimpleMemoryCache",
        'serializer': {
            'class': "aiocache.serializers.StringSerializer"
        }
    },
    'redis_alt': {
        'cache': "aiocache.RedisCache",
        'endpoint': "127.0.0.1",
        'port': 6379,
        'timeout': 1,
        'serializer': {
            'class': "aiocache.serializers.PickleSerializer"
        },
        'plugins': [
            {'class': "aiocache.plugins.HitMissRatioPlugin"},
            {'class': "aiocache.plugins.TimingPlugin"}
        ]
    }
})


async def default_cache():
    cache = caches.get('default')    # This always returns the same instance
    await cache.set("key", "value")

    assert await cache.get("key") == "value"
```

```
    assert isinstance(cache, SimpleMemoryCache)
    assert isinstance(cache.serializer, StringSerializer)


async def alt_cache():
    # This generates a new instance every time! You can also use `caches.create('alt
↪')`
    # or even `caches.create('alt', namespace="test", etc...)` to override extra args
    cache = caches.create(**caches.get_alias_config('redis_alt'))
    await cache.set("key", "value")

    assert await cache.get("key") == "value"
    assert isinstance(cache, RedisCache)
    assert isinstance(cache.serializer, PickleSerializer)
    assert len(cache.plugins) == 2
    assert cache.endpoint == "127.0.0.1"
    assert cache.timeout == 1
    assert cache.port == 6379
    await cache.close()


def test_alias():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(default_cache())
    loop.run_until_complete(alt_cache())

    cache = RedisCache()
    loop.run_until_complete(cache.delete("key"))
    loop.run_until_complete(cache.close())

    loop.run_until_complete(caches.get('default').close())


if __name__ == "__main__":
    test_alias()
```

When you do `caches.get('alias_name')`, the cache instance is built lazily the first time. Next accesses will return the **same** instance. If instead of reusing the same instance, you need a new one every time, use `caches.create('alias_name')`. One of the advantages of `caches.create` is that it accepts extra args that then are passed to the cache constructor. This way you can override args like namespace, endpoint, etc.

3.5 Decorators

aiocache comes with a couple of decorators for caching results from asynchronous functions. Do not use the decorator in synchronous functions, it may lead to unexpected behavior.

3.5.1 cached

```
import asyncio

from collections import namedtuple

from aiocache import cached, RedisCache
from aiocache.serializers import PickleSerializer
```

```

Result = namedtuple('Result', "content, status")


@cached(
    ttl=10, cache=RedisCache, key="key", serializer=PickleSerializer(), port=6379,␣
↪namespace="main")
async def cached_call():
    return Result("content", 200)


def test_cached():
    cache = RedisCache(endpoint="127.0.0.1", port=6379, namespace="main")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(cached_call())
    assert loop.run_until_complete(cache.exists("key")) is True
    loop.run_until_complete(cache.delete("key"))
    loop.run_until_complete(cache.close())


if __name__ == "__main__":
    test_cached()
```

3.5.2 multi_cached

```
import asyncio

from aiocache import multi_cached, RedisCache

DICT = {
    'a': "Z",
    'b': "Y",
    'c': "X",
    'd': "W"
}


@multi_cached("ids", cache=RedisCache, namespace="main")
async def multi_cached_ids(ids=None):
    return {id_: DICT[id_] for id_ in ids}


@multi_cached("keys", cache=RedisCache, namespace="main")
async def multi_cached_keys(keys=None):
    return {id_: DICT[id_] for id_ in keys}


cache = RedisCache(endpoint="127.0.0.1", port=6379, namespace="main")


def test_multi_cached():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(multi_cached_ids(ids=['a', 'b']))
    loop.run_until_complete(multi_cached_ids(ids=['a', 'c']))
    loop.run_until_complete(multi_cached_keys(keys=['d']))

```

```
    assert loop.run_until_complete(cache.exists('a'))
    assert loop.run_until_complete(cache.exists('b'))
    assert loop.run_until_complete(cache.exists('c'))
    assert loop.run_until_complete(cache.exists('d'))

    loop.run_until_complete(cache.delete("a"))
    loop.run_until_complete(cache.delete("b"))
    loop.run_until_complete(cache.delete("c"))
    loop.run_until_complete(cache.delete("d"))
    loop.run_until_complete(cache.close())


if __name__ == "__main__":
    test_multi_cached()
```

Warning: This was added in version 0.7.0 and the API is new. This means its open to breaking changes in future versions until the API is considered stable.

3.6 Locking

Warning: The implementations provided are **NOT** intented for consistency/synchronization purposes. If you need a locking mechanism focused on consistency, consider implementing your mechanism based on more serious tools like https://zookeeper.apache.org/.

There are a couple of locking implementations than can help you to protect against different scenarios:

3.6.1 RedLock

3.6.2 OptimisticLock

3.7 Testing

It's really easy to cut the dependency with aiocache functionality:

```
import asyncio

from asynctest import MagicMock

from aiocache.base import BaseCache


async def async_main():
    mocked_cache = MagicMock(spec=BaseCache)
    mocked_cache.get.return_value = "world"
    print(await mocked_cache.get("hello"))


if __name__ == "__main__":
```

```
    loop = asyncio.get_event_loop()
    loop.run_until_complete(async_main())
```

Note that we are passing the *BaseCache* as the spec for the Mock (you need to install `asynctest`).

Also, for debuging purposes you can use *AIOCACHE_DISABLE = 1 python myscript.py* to disable caching.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search