aiothrottles

Release 0.2.0

Aug 21, 2022

Contents:

aiothrottles synchronization primitives are designed to be extension to asyncio synchronization primitives.

For more details, see aiothrottles Documentation.

CHAPTER 1

Usage

Throttle implements a rate limiting for asyncio task. A throttle can be used to guarantee limited access to a shared resources.

The preferred way to use a Throttle is an async with statement:

```
throttle = Throttle('3/s')

# ... later
async with throttle:
    # access shared state
```

which is equivalent to:

```
throttle  = Throttle('3/s')

# ... later
await throttle.acquire()
try:
    # access shared state
finally:
    throttle.release()
```

A call rate is determined by the `rate` argument. Pass the rate in the following formats:

- `"{integer limit}/{unit time}"`
- `"{limit's numerator}/{limit's denominator}{unit time}"`

`rate` examples:

- `4/s`, `5/m`, `6/h`, `7/d`
- `1/second`, `2/minute`, `3/hour`, `4/day`
- `1/3s`, `12/37m`, `1/5h`, `8/3d`

CHAPTER 2

Installation

```
pip install aiothrottles
```

or

```
python setup.py install
```

CHAPTER 3

Supported Python Versions

Python 3.6, 3.7, 3.8 and 3.9 are supported.

3.1 Getting Started

3.1.1 Installation

If you use pip, just type

```
pip install aiothrottles
```

You can install from the source code like

```
git clone https://github.com/KonstantinTogoi/aiothrottles.git
cd aiothrottles
python setup.py install
```

3.2 Examples

3.2.1 awaitable

Use of `aiothrottles.Throttle` as awaitable object:

```
>>> import time
>>> from aiothrottles import Throttle
>>>
>>> throttle = Throttle(rate='1/s')
>>>
>>> async def foo(n):
```

(continues on next page)

(continued from previous page)

```
...     print(n, time.time())
...
>>> for i in range(5):
...     await throttle
...     await foo(i)
...     throttle.release()
...
0 1563275828.253736
1 1563275829.2547996
2 1563275830.2562528
3 1563275831.257302
4 1563275832.2587304
```

3.2.2 context manager

Use of `aiothrottles.Throttle` as context:

```
>>> import time
>>> from aiothrottles import Throttle
>>>
>>> throttle = Throttle(rate='1/s')
>>>
>>> async def foo(n):
...     print(n, time.time())
...
>>> for i in range(5):
...     async with throttle:
...         await foo(i)
...
0 1563275898.6722345
1 1563275899.673589
2 1563275900.6750457
3 1563275901.6763387
4 1563275902.6777005
```

3.2.3 decorator

Use of `aiothrottles.Throttle` as decorator for coroutines:

```
>>> import time
>>> from aiothrottles import throttle  # Throttle alias
>>>
>>> @throttle(rate='1/s')
... async def foo(n):
...     print(n, time.time())
...
>>> for i in range(5):
...     await foo(i)
...
0 1563272100.4413373
1 1563272101.4427333
2 1563272102.4441307
3 1563272103.445542
4 1563272104.4468124
```

3.3 throttles

Rate limiting primitives.

3.3.1 AwaitableMixin

class aiothrottles.throttles.**AwaitableMixin**

Awaitable object.

This enables the idiom:

```
await throttle
```

as an alternative to:

```
await throttle.acquire()
```

3.3.2 ContextManagerMixin

class aiothrottles.throttles.**ContextManagerMixin**

Context manager.

This enables the following idiom for acquiring and releasing a throttle around a block:

```
async with throttle:
    <block>
```

3.3.3 DecoratorMixin

class aiothrottles.throttles.**DecoratorMixin**

Coroutine decorator.

This enables decorating of a coroutine that always need acquiring and releasing a throttle:

```
@throttle('3/s')
async def coroutine():
    <block>
```

3.3.4 RateMixin

class aiothrottles.throttles.**RateMixin**(*rate: str*)

Encapsulation of a rate limiting.

This enables setting the limiting rate in the following formats:

- `"{integer limit}/{unit time}"`
- `"{limit's numerator}/{limit's denominator}{unit time}"`

Examples of usage:

- `"1/s"`, `"2/m"`, `"3/h"`, `"4/d"`
- `"5/second"`, `"6/minute"`, `"7/hour"`, `"8/day"`

- `"1/3s"`, `"12/37m"`, `"1/5h"`, `"8/3d"`

3.3.5 Throttle

class `aiothrottles.throttles.`**`Throttle`**(*rate*, *, *loop=None*)

Primitive throttle objects.

A primitive throttle is a synchronization primitive that manages an internal counter and has a trace. A primitive throttle is in one of two states, ‘locked’ or ‘unlocked’. It is not owned by a particular coroutine when locked.

Each acquire() call:

i) appends the coroutine to a FIFO queue
ii) blocks until the throttle is ‘locked’
iii) decrements the counter

Each release() call:

i) appends current timestamp at the and of the trace
ii) increments the counter

Each locked() call:

i) removes expired timestamps from the trace
ii) returns True if the length of the trace exceeds the limit or the counter is equal to zero

Usage:

```
throttle = Throttle()
...
await throttle
try:
    ...
finally:
    throttle.release()
```

Context manager usage:

```
throttle = Throttle()
...
async with throttle:
    ...
```

Throttle objects can be tested for locking state:

```
if not throttle.locked():
    await throttle
else:
    # throttle is acquired
    ...
```

`Throttle.`**`locked`**`()` → bool

Return True if throttle can not be acquired immediately.

Returns: bool

`Throttle.`**`acquire`**`()` → None

Acquire a throttle.

`Throttle.release()` → None

Release a throttle.

Raises: ValueError: when Throttle aleready released

CHAPTER 4

Indices and tables

- genindex
- modindex

Python Module Index

Index