aplex Documentation

Release 1.0.1

Lunluen

Oct 24, 2019

Contents:

CHAPTER 1

User’s Guide

1.1 Installation

1.1.1 Python Version

Aplex supports Python3.5+.

1.1.2 Dependencies

Required

- None

Optional

- uvloop is a fast, drop-in replacement of the built-in asyncio event loop.

1.1.3 Install Aplex

For General Users

Use the package manager pip or pipenv to install aplex.

With pip:

```
$ pip install aplex
```

Or with pipenv:

```
$ pipenv install aplex
```

Install Optional Dependencies

Simply add a suffix:

```
$ pip install aplex[uvloop]
```

For Contributors

Install with pipenv(recommand if you want to build docs):

```
git clone https://github.com/lunluen/aplex.git
cd aplex
pipenv install --dev
```

or with setuptools:

```
git clone https://github.com/lunluen/aplex.git
cd aplex
python setup.py develop
```

1.2 Aplex Quickstart

Translations: |

“Aplex”, short for “**a**synchronous **p**ool **ex**ecutor”, is a Python library for combining asyncio with multiprocessing and threading.

- Aplex helps you run coroutines and functions in other processes or threads with asyncio concurrently and in parallel (if with processes).
- Aplex provides a usage like that of standard library `concurrent.futures`, which is familiar to you and intuitive.
- Aplex lets you do load balancing in a simple way if you need.

1.2.1 Installation

For general users, use the package manager pip to install aplex.

```
pip install aplex
```

For contributors, install with pipenv:

```
git clone https://github.com/lunluen/aplex.git
cd aplex
pipenv install --dev
```

or with setuptools:

```
git clone https://github.com/lunluen/aplex.git
cd aplex
python setup.py develop
```

1.2.2 Usage

Definition to know:

> A `work` is a `callable` you want to run with asyncio and multiprocessing or threading. It can be a coroutine function or just a function.

In below case, the `work` is the coroutine function `demo`.

Submit

You can submit your work like:

```
import aiohttp
from aplex import ProcessAsyncPoolExecutor

async def demo(url):
    async with aiohttp.request('GET', url) as response:
        return response.status

if __name__ == '__main__':
    pool = ProcessAsyncPoolExecutor(pool_size=8)
    future = pool.submit(demo, 'http://httpbin.org')
    print('Status: %d.' % future.result())
```

Note: If you are running python on windows, `if __name__ == '__main__':` is necessary. That's the design of multiprocessing.

Result:

```
Status: 200
```

Map

For multiple works, try `map`:

```
iterable = ('http://httpbin.org' for __ in range(10))
for status in pool.map(demo, iterable, timeout=10):
    print('Status: %d.' % status)
```

Awaiting results

Aplex allows one to `await` results with the event loop that already exists. It's quite simple.

Just set keyword argument `awaitable` to `True`!

For example:

```
pool = ProcessAsyncPoolExecutor(awaitable=True)
```

Then

```
future = pool.submit(demo, 'http://httpbin.org')
status = await future
```

How about `map`?

```
async for status in pool.map(demo, iterable, timeout=10):
    print('Status: %d.' % status)
```

Load balancing

In aplex, each worker running your works is the process or thread on your computer. That is, they have the same capability computing. *But*, your works might have different workloads. Then you need a load balancer.

Aplex provides some useful load balancers. They are `RoundRobin`, `Random`, and `Average`. The default is `RoundRobin`.

Simply set what you want in the keyword argument of contruction:

```
from aplex import ProcessAsyncPoolExecutor
from aplex.load_balancers import Average

if __name__ == '__main__':
    pool = ProcessAsyncPoolExecutor(load_balancer=Average)
```

Done. So easy. :100:

You can also customize one:

```
from aplex import LoadBalancer

class MyAwesomeLoadBalancer(LoadBalancer):
    def __init__(*args, **kwargs):
        super().__init__(*args, **kwargs)  # Don't forget this.
        awesome_attribute = 'Hello Aplex!'

    def get_proper_worker(self):
        the_poor_guy = self.workers[0]
        return the_poor_guy
```

See details of how to implement a load balancer at: LoadBalancer | API Reference

Worker loop factory

By the way, if you think the build-in asyncio loop is too slow:

```
import uvloop
from aplex import ProcessAsyncPoolExecutor

if __name__ == '__main__':
    pool = ProcessAsyncPoolExecutor(worker_loop_factory=uvloop.Loop)
```

1.2.3 Graceful Exit

Taking Python3.6 for example, a graceful exit without aplex would be something like this:

```
try:
    loop.run_forever()
finally:
    try:
        tasks = asyncio.Task.all_tasks()
        if tasks:
            for task in tasks:
                task.cancel()
            gather = asyncio.gather(*tasks)
            loop.run_until_complete(gather)
        loop.run_until_complete(loop.shutdown_asyncgens())
    finally:
        loop.close()
```

. . . It's definitely a joke.

Here, just treat pool as a context manager:

```
with ProcessAsyncPoolExecutor() as pool:
    do_something()
```

or remember to call `pool.shutdown()`. These help you deal with that joke.

. . .

What? You forget to call `pool.shutdown()`?!

Ok, fine. It will shut down automatically when the program exits or it gets garbage-collected.

1.2.4 Like this?

Scroll up and click `Watch - Releases only` and `Star` as a thumbs up! :+1:

1.2.5 Any feedback?

Feel free to open a issue (just don't abuse it).

Or contact me: `mas581301@gmail.com` :mailbox:

Anything about aplex is welcome, such like bugs, system design, variable naming, even English grammer of docstrings!

1.2.6 How to contribute

Contribution are welcome.

Asking and advising are also kinds of contribution.

Please see CONTRIBUTING.md

1.2.7 License

MIT

CHAPTER 2

API Reference

2.1 API

2.1.1 Executor Objects

class aplex.**ProcessAsyncPoolExecutor**(*, *pool_size: Optional[int] = 4*, *max_works_per_worker: Optional[int] = 300*, *load_balancer: Optional[aplex.load_balancers.LoadBalancer] = <class 'aplex.load_balancers.RoundRobin'>*, *awaitable: Optional[bool] = False*, *future_loop: asyncio.events.AbstractEventLoop = None*, *worker_loop_factory: Optional[asyncio.events.AbstractEventLoop] = None*)

Setups executor and adds self to executor track set.

Parameters

- **pool_size** – Number of workers, i.e., number of threads or processes.
- **max_works_per_worker** – The max number of works a worker can run at the same time. This does not **limit** the number of asyncio tasks of a worker.
- **load_balancer** – A subclass of aplex.LoadBalancer for submitted item load balancing that has implemented abstract method `get_proper_worker`.
- **awaitable** – If it's set to True, futures returned from `submit` method will be awaitable, and `map` will return async generator(async iterator if python3.5).
- **future_loop** – Loop instance set in awaitable futures returned from `submit` method.

  If specified, `awaitable` must be set to true.

  This loop can also be set in `set_future_loop` method.
- **worker_loop_factory** – A factory to generate loop instance for workers to run their job.

Raises `ValueError` – `future_loop` is specified while `awaitable` is False.

map(*work: Callable*, **iterables*, *timeout: Optional[float] = None*, *chunksize: int = 1*, *load_balancing_meta: Optional[Any] = None*) → Union[AsyncGenerator[T_co, T_contra], Generator[T_co, T_contra, V_co]]

map your work like the way in concurrent.futures.

The work submitted will be sent to the specific worker that the load balancer choose.

Note: The work you submit should be a callable, And a `coroutine` is **not** a callable. You should submit a `coroutine function` and specify its args and kwargs here instead.

Parameters

- **work** – The callable that will be run in a worker.
- ***iterables** – Position arguments for work. All of them are iterable and have same length.
- **timeout** – The time limit for waiting results.
- **chunksize** – Works are gathered, partitioned as chunks in this size, and then sent to workers.
- **load_balancing_meta** – This will be passed to load balancer for the choice of proper worker.

Returns

A async generator yielding the map results if `awaitable` is set to True, otherwise a generator. In python3.5, async iterator is used to replace async generator.

If a exception is raised in a work, it will be re-raised in the generator, and the remaining works will be cancelled.

Raises

- `ValueError` – If chunksize is less than 1.
- `TypeError` – If work is not a callable.

set_future_loop(*loop: asyncio.events.AbstractEventLoop*)

Sets loop for awaitable futures to await results.

This loop can also be set in initialization.

Parameters **loop** – The Loop needed for awaitable futures.

Raises

- `RuntimeError` – If executor has been shut down, or executor is set to be unawaitable.
- `AplexWorkerError` – If some workers are broken or raise BaseException.

shutdown(*wait: bool = True*)

Shuts down the executor and frees the resource.

Parameters **wait** – Whether to block until shutdown is finished.

submit(*work: Callable*, **args*, *load_balancing_meta: Optional[Any] = None*, ***kwargs*) → Union[aplex.futures.AsyncioFuture, aplex.futures.ConcurrentFuture]

submits your work like the way in concurrent.futures.

The work submitted will be sent to the specific worker that the load balancer choose.

Note: The work you submit should be a callable, And a `coroutine` is **not** a callable. You should submit a `coroutine function` and specify its args and kwargs here instead.

Parameters

- **`work`** – The callable that will be run in a worker.
- **`*args`** – Position arguments for work.
- **`load_balancing_meta`** – This will be passed to load balancer for the choice of proper worker.
- **`**kwargs`** – Keyword arguments for work.

Returns

A future.

The future will be awaitable like that in asyncio if `awaitable` is set to True in executor construction, otherwise, unawaitable like that in concurrent.futures.

Raises

- `RuntimeError` – If executor has been shut down.
- `AplexWorkerError` – If some workers are broken or raise BaseException.
- `TypeError` – If work is not a callable.

class aplex.**ThreadAsyncPoolExecutor**(**, pool_size: Optional[int] = 4, max_works_per_worker: Optional[int] = 300, load_balancer: Optional[aplex.load_balancers.LoadBalancer] = <class 'aplex.load_balancers.RoundRobin'>, awaitable: Optional[bool] = False, future_loop: asyncio.events.AbstractEventLoop = None, worker_loop_factory: Optional[asyncio.events.AbstractEventLoop] = None*)

Setups executor and adds self to executor track set.

Parameters

- **`pool_size`** – Number of workers, i.e., number of threads or processes.
- **`max_works_per_worker`** – The max number of works a worker can run at the same time. This does not **limit** the number of asyncio tasks of a worker.
- **`load_balancer`** – A subclass of aplex.LoadBalancer for submitted item load balancing that has implemented abstract method `get_proper_worker`.
- **`awaitable`** – If it's set to True, futures returned from `submit` method will be awaitable, and `map` will return async generator(async iterator if python3.5).
- **`future_loop`** – Loop instance set in awaitable futures returned from `submit` method.

  If specified, `awaitable` must be set to true.

  This loop can also be set in `set_future_loop` method.
- **`worker_loop_factory`** – A factory to generate loop instance for workers to run their job.

Raises `ValueError` – `future_loop` is specified while `awaitable` is False.

map(*work: Callable*, **iterables*, *timeout: Optional[float] = None*, *chunksize: int = 1*, *load_balancing_meta: Optional[Any] = None*) → Union[AsyncGenerator[T_co, T_contra], Generator[T_co, T_contra, V_co]]

map your work like the way in concurrent.futures.

The work submitted will be sent to the specific worker that the load balancer choose.

Note: The work you submit should be a callable, And a `coroutine` is **not** a callable. You should submit a `coroutine function` and specify its args and kwargs here instead.

Parameters

- **work** – The callable that will be run in a worker.
- ***iterables** – Position arguments for work. All of them are iterable and have same length.
- **timeout** – The time limit for waiting results.
- **chunksize** – Works are gathered, partitioned as chunks in this size, and then sent to workers.
- **load_balancing_meta** – This will be passed to load balancer for the choice of proper worker.

Returns

A async generator yielding the map results if `awaitable` is set to True, otherwise a generator. In python3.5, async iterator is used to replace async generator.

If a exception is raised in a work, it will be re-raised in the generator, and the remaining works will be cancelled.

Raises

- `ValueError` – If chunksize is less than 1.
- `TypeError` – If work is not a callable.

set_future_loop(*loop: asyncio.events.AbstractEventLoop*)

Sets loop for awaitable futures to await results.

This loop can also be set in initialization.

Parameters **loop** – The Loop needed for awaitable futures.

Raises

- `RuntimeError` – If executor has been shut down, or executor is set to be unawaitable.
- `AplexWorkerError` – If some workers are broken or raise BaseException.

shutdown(*wait: bool = True*)

Shuts down the executor and frees the resource.

Parameters **wait** – Whether to block until shutdown is finished.

submit(*work: Callable*, **args*, *load_balancing_meta: Optional[Any] = None*, ***kwargs*) → Union[aplex.futures.AsyncioFuture, aplex.futures.ConcurrentFuture]

submits your work like the way in concurrent.futures.

The work submitted will be sent to the specific worker that the load balancer choose.

> **Note:** The work you submit should be a callable, And a `coroutine` is **not** a callable. You should submit a `coroutine function` and specify its args and kwargs here instead.

Parameters

- **`work`** – The callable that will be run in a worker.
- **`*args`** – Position arguments for work.
- **`load_balancing_meta`** – This will be passed to load balancer for the choice of proper worker.
- **`**kwargs`** – Keyword arguments for work.

Returns

A future.

The future will be awaitable like that in asyncio if `awaitable` is set to True in executor construction, otherwise, unawaitable like that in concurrent.futures.

Raises

- `RuntimeError` – If executor has been shut down.
- `AplexWorkerError` – If some workers are broken or raise BaseException.
- `TypeError` – If work is not a callable.

2.1.2 Future Objects

class `aplex.futures.`**`ConcurrentFuture`**(*cancel_interface*)

A concurrent.futures.Future subclass that cancels like asyncio.Task.

`cancel`()

Tries to cancel the work submitted to worker.

Unlike `concurrent.futures`, the *running* work is *cancellable* as long as it's a `coroutine function`.

Returns True if cancellable, False otherwise.

class `aplex.futures.`**`AsyncioFuture`**(*concurrent_future*, *loop=None*)

Asyncio.Future subclass that cancels like asyncio.Task.

`cancel`()

Tries to cancel the work submitted to worker.

Unlike `concurrent.futures`, the *running* work is *cancellable* as long as it's a `coroutine function`.

Returns True if cancellable, False otherwise.

2.1.3 Load Balancer Objects

class `aplex.load_balancers.`**`LoadBalancer`**(*workers: List[Worker], workloads: Dict[Worker, int], max_works_per_worker: int*)

The base class of all load balancers.

Users can inherit this to write their own load balancers.

Initialization.

Note: Must call `super().__init__(*args, **kwargs)` in the beginning of the `__init__` block if you are trying to overwrite this.

Parameters

- **`workers`** – A argument for `workers` property.
- **`workloads`** – A argument for `workloads` property.
- **`max_works_per_worker`** – A argument for `max_works_per_worker` property.

`get_available_workers``()` → Iterator[Worker]

Returns the workers that does not reach the `max_works_per_worker` limit.

Returns A iterator of the available workers.

`get_proper_worker``(`*load_balancing_meta: Optional[Any]*`)` → Worker

The method to be implemented by users. Returns an available worker.

Note: There is always at least an available worker when this method is called.

Parameters **`load_balancing_meta`** – An optional argument specified in `submit` and `map` methods that users may need for choosing a proper worker.

Returns A worker that is available for work assignment.

`is_available``(`*worker: Worker*`)` → bool

Returns if the given worker reaches the `max_works_per_worker` limit.

Parameters **`worker`** – A worker object.

Returns True if available, else False.

`max_works_per_worker`

Returns tha max number of works a worker can run at the same time.

`workers`

Returns worker list.

`workloads`

Returns worker workload mapping.

`class` `aplex.load_balancers.`**`RoundRobin`**`(`**args*, ***kwargs*`)`

A load balancer based on round-robin algorithm.

`get_available_workers``()` → Iterator[Worker]

Returns the workers that does not reach the `max_works_per_worker` limit.

Returns A iterator of the available workers.

`get_proper_worker``(`*load_balancing_meta: Optional[Any]*`)` → Worker

Returns the next available worker.

Parameters **`load_balancing_meta`** – An optional argument specified in `submit` and `map` methods that users may need for choosing a proper worker.

Returns A worker that is available for work assignment.

`is_available``(`*worker: Worker*`)` → bool

Returns if the given worker reaches the `max_works_per_worker` limit.

Parameters **`worker`** – A worker object.

Returns True if available, else False.

`max_works_per_worker`
Returns tha max number of works a worker can run at the same time.

`workers`
Returns worker list.

`workloads`
Returns worker workload mapping.

`class` `aplex.load_balancers.`**`Random`**`(`*workers: List[Worker], workloads: Dict[Worker, int], max_works_per_worker: int*`)`
A load balancer that chooses proper worker randomly.

Initialization.

Note: Must call `super().__init__(*args, **kwargs)` in the beginning of the `__init__` block if you are trying to overwrite this.

Parameters

- **`workers`** – A argument for `workers` property.
- **`workloads`** – A argument for `workloads` property.
- **`max_works_per_worker`** – A argument for `max_works_per_worker` property.

`get_available_workers``()` → Iterator[Worker]
Returns the workers that does not reach the `max_works_per_worker` limit.

Returns A iterator of the available workers.

`get_proper_worker``(`*load_balancing_meta: Optional[Any]*`)` → Worker
Randomly picks an avaiable worker.

Parameters **`load_balancing_meta`** – An optional argument specified in `submit` and `map` methods that users may need for choosing a proper worker.

Returns A worker that is available for work assignment.

`is_available``(`*worker: Worker*`)` → bool
Returns if the given worker reaches the `max_works_per_worker` limit.

Parameters **`worker`** – A worker object.

Returns True if available, else False.

`max_works_per_worker`
Returns tha max number of works a worker can run at the same time.

`workers`
Returns worker list.

`workloads`
Returns worker workload mapping.

`class` `aplex.load_balancers.`**`Average`**`(`*workers: List[Worker], workloads: Dict[Worker, int], max_works_per_worker: int*`)`
A load balancer that tries to equalize the workloads of all the workers.

To put it otherwise, it assign work to the worker having minimun workload.

Initialization.

Note: Must call `super().__init__(*args, **kwargs)` in the beginning of the `__init__` block if you are trying to overwrite this.

Parameters

- **`workers`** – A argument for `workers` property.
- **`workloads`** – A argument for `workloads` property.
- **`max_works_per_worker`** – A argument for `max_works_per_worker` property.

`get_available_workers``()` → Iterator[Worker]

Returns the workers that does not reach the `max_works_per_worker` limit.

Returns A iterator of the available workers.

`get_proper_worker``(`*load_balancing_meta: Optional[Any]*`)` → Worker

Returns the worker with minimum workload.

Parameters **`load_balancing_meta`** – An optional argument specified in `submit` and `map` methods that users may need for choosing a proper worker.

Returns A worker that is available for work assignment.

`is_available``(`*worker: Worker*`)` → bool

Returns if the given worker reaches the `max_works_per_worker` limit.

Parameters **`worker`** – A worker object.

Returns True if available, else False.

`max_works_per_worker`

Returns tha max number of works a worker can run at the same time.

`workers`

Returns worker list.

`workloads`

Returns worker workload mapping.

CHAPTER 3

Changelog

3.1 Aplex Changelog

3.1.1 Under Development

New Features

- TODO

Improvements

- TODO

Bugfixes

- TODO

Dependencies

- TODO

Deprecations

- TODO

Miscellaneous

- TODO

3.1.2 v1.0.1 (2019-02-10)

First release.

CHAPTER 4

The Contributor Guide

4.1 The Contributor Guide

4.1.1 Questions

It's better to ask on Stack Overflow, but not limited to. Remember to add a tag of aplex.

4.1.2 Bug Reports

It's better to tell me but not limited to:

- What you expected to happen
- What actually happens (include the complete traceback)
- How to reproduce the issue
- Your python and aplex versions

4.1.3 Pull requests

Keep the code style consistent. This package follows Google Style Guide.

CHAPTER 5

License

5.1 MIT License

Python Module Index

Index