atasker

Dec 15, 2019

Contents

Python library for modern thread / multiprocessing pooling and task processing via asyncio.

No matter how your code is written, atasker automatically detects blocking functions and coroutines and launches them in a proper way, in a thread, asynchronous loop or in multiprocessing pool.

Tasks are grouped into pools. If there's no space in pool, task is being placed into waiting queue according to their priority. Pool also has "reserve" for the tasks with priorities "normal" and higher. Tasks with "critical" priority are always executed instantly.

This library is useful if you have a project with many similar tasks which produce approximately equal CPU/memory load, e.g. API responses, scheduled resource state updates etc.

CHAPTER 1

Install

```
pip3 install atasker
```

Sources: https://github.com/alttch/atasker

Documentation: https://atasker.readthedocs.io/

CHAPTER 2

Why

- asynchronous programming is a perfect way to make your code fast and reliable
- multithreading programming is a perfect way to run blocking code in the background

atasker combines advantages of both ways: atasker tasks run in separate threads however task supervisor and workers are completely asynchronous. But all their public methods are thread-safe.

CHAPTER 3

Why not standard Python thread pool?

- threads in a standard pool don't have priorities
- workers

CHAPTER 4

Why not standard asyncio loops?

- compatibility with blocking functions
- async workers

CHAPTER 5

Why not concurrent.futures?

concurrent.futures is a great standard Python library which allows you to execute specified tasks in a pool of workers.

atasker method *background_task* solves the same problem but in slightly different way, adding priorities to the tasks, while *atasker* workers do absolutely different job:

- in *concurrent.futures* worker is a pool member which executes the single specified task.
- in *atasker* worker is an object, which continuously *generates* new tasks with the specified interval or on external event, and executes them in thread or multiprocessing pool.

CHAPTER 6

Code examples

6.1 Start/stop

```
from atasker import task_supervisor

# set pool size
task_supervisor.set_thread_pool(pool_size=20, reserve_normal=5, reserve_high=5)
task_supervisor.start()
# ...
# start workers, other threads etc.
# ...
# optionally block current thread
task_supervisor.block()

# stop from any thread
task_supervisor.stop()
```

6.2 Background task

```
from atasker import background_task, TASK_LOW, TASK_HIGH, wait_completed

# with annotation
@background_task
def mytask():
    print('I am working in the background!')
    return 777

task = mytask()

# optional
result = wait_completed(task)
```

(continues on next page)

(continued from previous page)

```
print(task.result) # 777
print(result) # 777

# with manual decoration
def mytask2():
    print('I am working in the background too!')

task = background_task(mytask2, priority=TASK_HIGH)()
```

6.3 Async tasks

```
# new asyncio loop is automatically created in own thread
a1 = task_supervisor.create_aloop('myaloop', default=True)

async def calc(a):
    print(a)
    await asyncio.sleep(1)
    print(a * 2)
    return a * 3

# call from sync code

# put coroutine
task = background_task(calc)(1)

wait_completed(task)

# run coroutine and wait for result
result = a1.run(calc(1))
```

6.4 Worker examples

```
from atasker import background_worker, TASK_HIGH

@background_worker
def worker1(**kwargs):
    print('I am a simple background worker')

@background_worker
async def worker_async(**kwargs):
    print('I am async background worker')

@background_worker(interval=1)
def worker2(**kwargs):
    print('I run every second!')

@background_worker(queue=True)
def worker3(task, **kwargs):
    print('I run when there is a task in my queue')
```

(continues on next page)

(continued from previous page)

```
@background_worker(event=True, priority=TASK_HIGH)
def worker4(**kwargs):
    print('I run when triggered with high priority')

worker1.start()
worker_async.start()
worker2.start()
worker3.start()
worker4.start()

worker3.put('todo1')
worker4.trigger()

from atasker import BackgroundIntervalWorker

class MyWorker(BackgroundIntervalWorker):

    def run(self, **kwargs):
        print('I am custom worker class')

worker5 = MyWorker(interval=0.1, name='worker5')
worker5.start()
```

6.4.1 Task supervisor

Task supervisor is a component which manages task thread pool and run task *schedulers (workers)*.

Contents

Usage

When **atasker** package is imported, default task supervisor is automatically created.

```
from atasker import task_supervisor

# thread pool
task_supervisor.set_thread_pool(
    pool_size=20, reserve_normal=5, reserve_high=5)
task_supervisor.start()
```

Warning: Task supervisor must be started before any scheduler/worker or task.

Task priorities

Task supervisor supports 4 task priorities:

- TASK_LOW
- TASK_NORMAL (default)
- TASK_HIGH
- TASK_CRITICAL

```
from atasker import TASK_HIGH

def test():
    pass

background_task(test, name='test', priority=TASK_HIGH)()
```

Pool size

Parameter **pool_size** for **task_supervisor.set_thread_pool** defines size of the task (thread) pool.

Pool size means the maximum number of the concurrent tasks which can run. If task supervisor receive more tasks than pool size has, they will wait until some running task is finished.

Actually, parameter **pool_size** defines pool size for the tasks, started with *TASK_LOW* priority. Tasks with higher priority have "reserves": *pool_size=20, reserve_normal=5* means create pool for 20 tasks but reserve 5 more places for the tasks with *TASK_NORMAL* priority. In this example, when task supervisor receives such task, pool is "extended", up to 5 places.

For *TASK_HIGH* pool size can be extended up to *pool_size* + *reserve_normal* + *reserve_high*, so in the example above: *20 + 5 + 5 = 30*.

Tasks with priority *TASK_CRITICAL* are always started instantly, no matter how busy task pool is, and thread pool is being extended for them with no limits. Multiprocessing critical tasks are started as soon as *multiprocessing.Pool* object has free space for the task.

To make pool size unlimited, set *pool_size=0*.

Parameters *min_size* and *max_size* set actual system thread pool size. If *max_size* is not specified, it's set to *pool_size + reserve_normal + reserve_high*. It's recommended to set *max_size* slightly larger manually to have a space for critical tasks.

By default, *max_size* is CPU count * 5. You may use argument *min_size='max'* to automatically set minimal pool size to max.

Note: pool size can be changed while task supervisor is running.

Poll delay

Poll delay is a delay (in seconds), which is used by task queue manager, in *workers* and some other methods like *start/stop*.

Lower poll delay = higher CPU usage, higher poll delay = lower reaction time.

Default poll delay is 0.1 second. Can be changed with:

```
task_supervisor.poll_delay = 0.01 # set poll delay to 10ms
```

Blocking

Task supervisor is started in its own thread. If you want to block current thread, you may use method

```
task_supervisor.block()
```

which will just sleep while task supervisor is active.

Timeouts

Task supervisor can log timeouts (when task isn't launched within a specified number of seconds) and run timeout handler functions:

```
def warning(t):
    # t = task thread object
    print('Task thread {} is not launched yet'.format(t))

def critical(t):
    print('All is worse than expected')

task_supervisor.timeout_warning = 5
task_supervisor.timeout_warning_func = warn
task_supervisor.timeout_critical = 10
task_supervisor.timeout_critical_func = critical
```

Stopping task supervisor

```
task_supervisor.stop(wait=True, stop_schedulers=True, cancel_tasks=False)
```

Params:

- **wait** wait until tasks and scheduler coroutines finish. If **wait=<number>**, task supervisor will wait until coroutines finish for the max. *wait* seconds. However if requested to stop schedulers (workers) or task threads are currently running, method *stop* wait until they finish for the unlimited time.
- **stop_schedulers** before stopping the main event loop, task scheduler will call *stop* method of all schedulers running.
- **cancel_tasks** if specified, task supervisor will try to forcibly cancel all scheduler coroutines.

aloops: async executors and tasks

Usually it's unsafe to run both *schedulers (workers)* executors and custom tasks in supervisor's event loop. Workers use event loop by default and if anything is blocked, the program may be freezed.

To avoid this, it's strongly recommended to create independent async loops for your custom tasks. atasker supervisor has built-in engine for async loops, called "aloops", each aloop run in a separated thread and doesn't interfere with supervisor event loop and others.

Create

If you plan to use async worker executors, create aloop:

```
a = task_supervisor.create_aloop('myworkers', default=True, daemon=True)
# the loop is instantly started by default, to prevent add param start=False
# and then use
# task_supervisor.start_aloop('myworkers')
```

To determine in which thread executor is started, simply get its name. aloop threads are called "supervisor_aloop_<name>".

Using with workers

Workers automatically launch async executor function in default aloop, or aloop can be specified with *loop=* at init or *_loop=* at startup.

Executing own coroutines

aloops have 2 methods to execute own coroutines:

```
# put coroutine to loop
task = aloop.background_task(coro(args))

# blocking wait for result from coroutine
result = aloop.run(coro(args))
```

Other supervisor methods

Note: It's not recommended to create/start/stop aloops without supervisor

```
# set default aloop
task_supervisor.set_default_aloop(aloop):

# get aloop by name
task_supervisor.get_aloop(name)

# stop aloop (not required, supervisor stops all aloops at shutdown)
task_supervisor.stop_aloop(name)

# get aloop async event loop object for direct access
aloop.get_loop()
```

Multiprocessing

Multiprocessing pool may be used by workers and background tasks to execute a part of code.

To create multiprocessing pool, use method:

```
from atasker import task_supervisor

# task_supervisor.create_mp_pool(<args for multiprocessing.Pool>)
# e.g.
task_supervisor.create_mp_pool(processes=8)

# use custom mp Pool

from multiprocessing import Pool

pool = Pool(processes=4)
task_supervisor.mp_pool = pool

# set mp pool size. if pool wasn't created before, it will be initialized
# with processes=(pool_size+reserve_normal+reserve_high)
task_supervisor.set_mp_pool(
    pool_size=20, reserve_normal=5, reserve_high=5)
```

Custom task supervisor

```
from atasker import TaskSupervisor

my_supervisor = TaskSupervisor(
    pool_size=100, reserve_normal=10, reserve_high=10)

class MyTaskSupervisor(TaskSupervisor):
    # .......

my_supervisor2 = MyTaskSupervisor()
```

Putting own tasks

If you can not use *background tasks* for some reason, you may put own tasks manually and put it to task supervisor to launch:

```
task = task_supervisor.put_task(target=myfunc, args=(), kwargs={},
  priority=TASK_NORMAL, delay=None)
```

If *delay* is specified, the thread is started after the corresponding delay (seconds).

After the function thread is finished, it should notify task supervisor:

```
task_supervisor.mark_task_completed(task=task) # or task_id = task.id
```

If no *task_id* specified, current thread ID is being used:

```
# note: custom task targets always get _task_id in kwargs
 def mytask(**kwargs):
    # ... perform calculations
   task_supervisor.mark_task_completed(task_id=kwargs['_task_id'])

 task_supervisor.put_task(target=mytask)
```

Note: If you need to know task id, before task is put (e.g. for task callback), you may generate own and call *put_task* with *task_id=task_id* parameter.

Putting own tasks in multiprocessing pool

To put own task into multiprocessing pool, you must create tuple object which contains:

- unique task id
- task function (static method)
- function args
- function kwargs
- result callback function

```
import uuid

from atasker import TT_MP

task = task_supervisor.put_task(
   target=<somemodule.staticmethod>, callback=<somefunc>, tt=TT_MP)
```

After the function is finished, you should notify task supervisor:

```
task_supervisor.mark_task_completed(task_id=<task_id>, tt=TT_MP)
```

Creating own schedulers

Own task scheduler (worker) can be registered in task supervisor with:

```
task_supervisor.register_scheduler(scheduler)
```

Where *scheduler* = scheduler object, which should implement at least *stop* (regular) and *loop* (async) methods.

Task supervisor can also register synchronous schedulers/workers, but it can only stop them when *stop* method is called:

```
task_supervisor.register_sync_scheduler(scheduler)
```

To unregister schedulers from task supervisor, use *unregister_scheduler* and *unregister_sync_scheduler* methods.

6.4.2 Tasks

Task is a Python function which will be launched in the separate thread.

Defining task with annotation

```
from atasker import background_task

@background_task
def mytask():
    print('I am working in the background!')

task = mytask()
```

It's not required to notify task supervisor about task completion, *background_task* will do this automatically as soon as task function is finished.

All start parameters (args, kwargs) are passed to task functions as-is.

Task function without annotation

To start task function without annotation, you must manually decorate it:

```
from atasker import background_task, TASK_LOW

def mytask():
    print('I am working in the background!')

task = background_task(mytask, name='mytask', priority=TASK_LOW)()
```

Multiprocessing task

Run as background task

To put task into *multiprocessing pool*, append parameter *tt=TT_MP*:

```
from atasker import TASK_HIGH, TT_MP

task = background_task(
    tests.mp.test, priority=TASK_HIGH, tt=TT_MP)(1, 2, 3, x=2)
```

Optional parameter *callback* can be used to specify function which handles task result.

Note: Multiprocessing target function always receives *_task_id* param.

Run in async way

You may put task from your coroutine, without using callback, example:

```
from atasker import co_mp_apply, TASK_HIGH

async def f1():
    result = await co_mp_apply(
        tests.mp.test, args=(1,2,3), kwargs={'x': 2},
        priority=TASK_HIGH)
```

Task object

If you saved only task.id but not the whole object, you may later obtain Task object again:

```
from atasker import task_supervisor

task = task_supervisor.get_task(task.id)
```

Task info object fields:

- **id** task id
- **task** task object
- **tt** task type (TT_THREAD, TT_MP)
- **priority** task priority
- **time_queued** time when task was queued
- **time_started** time when task was started
- **result** task result
- **status task status 0** queued **2** delayed **100** started **200** completed **-1** canceled

If task info is *None*, consider the task is completed and supervisor destroyed information about it.

Note: As soon as task is marked as completed, supervisor no longer stores information about it

Wait until completed

You may wait until pack of tasks is completed with the following method:

```
from atasker import wait_completed

wait_completed([task1, task2, task3 .... ], timeout=None)
```

The method return list of task results if all tasks are finished, or raises *TimeoutError* if timeout was specified but some tasks are not finished.

If you call method with a single task instead of list or tuple, single result is returned.

6.4.3 Async jobs

atasker has built-in integration with aiosched - simple and fast async job scheduler.

aiosched schedulers can be automatically started inside *aloop*:

```
async def test1():
   print('I am lightweight async job')

task_supervisor.create_aloop('jobs')
# if aloop id not specified, default aloop is used
task_supervisor.create_async_job_scheduler('default', aloop='jobs',
   default=True)
# create async job
job1 = task_supervisor.create_async_job(target=test1, interval=0.1)
# cancel async job
task_supervisor.cancel_async_job(job=job1)
```

Note: **aiosched** jobs are lightweight, don't report any statistic data and don't check is the job already running.

6.4.4 Workers

Worker is an object which runs specified function (executor) in a loop.

Contents

Common

Worker parameters

All workers support the following initial parameters:

- **name** worker name (default: name of executor function if specified, otherwise: auto-generated UUID)
- **func** executor function (default: *worker.run*)
- **priority** worker thread priority
- **o** special object, passed as-is to executor (e.g. object worker is running for)
- **on_error** a function which is called, if executor raises an exception
- **on_error_kwargs** kwargs for *on_error* function
- **supervisor** alternative *task supervisor*
- **poll_delay** worker poll delay (default: task supervisor poll delay)

Methods

Overriding parameters at startup

Initial parameters *name*, *priority* and *o* can be overriden during worker startup (first two - as *_name* and *_priority*)

```
myworker.start(_name='worker1', _priority=atasker.TASK_LOW)
```

Executor function

Worker executor function is either specified with annotation or named *run* (see examples below). The function should always have ***kwargs* param.

Executor function gets in args/kwargs:

- all parameters *worker.start* has been started with.
- **_worker** current worker object
- **_name** current worker name
- **_task_id** if executor function is started in multiprocessing pool - ID of current task (for thread pool, task id = thread name).

Note: If executor function return *False*, worker stops itself.

Asynchronous executor function

Executor function can be asynchronous, in this case it's executed inside *task supervisor* loop, no new thread is started and *priority* is ignored.

When *background_worker* decorator detects asynchronous function, class *BackgroundAsyncWorker* is automatically used instead of *BackgroundWorker* (*BackgroundQueueWorker*, *BackgroundEventWorker* and *BackgroundIntervalWorker* support synchronous functions out-of-the-box).

Additional worker parameter *loop* (*_loop* at startup) may be specified to put executor function inside external async loop.

Note: To prevent interference between supervisor event loop and executors, it's strongly recommended to specify own async event loop or create *aloop*.

Multiprocessing executor function

To use multiprocessing, *task supervisor* mp pool must be created.

If executor method *run* is defined as static, workers automatically detect this and use multiprocessing pool of task supervisor to launch executor.

Note: As executor is started in separate process, it doesn't have an access to *self* object.

Additionally, method *process_result* must be defined in worker class to process executor result. The method can stop worker by returning *False* value.

Example, let's define *BackgroundQueueWorker*. Python multiprocessing module can not pick execution function defined via annotation, so worker class is required. Create it in separate module as Python multiprocessing can not pick methods from the module where the worker is started:

Warning: Multiprocessing executor function should always finish correctly, without any exceptions otherwise callback function is never called and task become "freezed" in pool.

myworker.py

```
class MyWorker(BackgroundQueueWorker):

    # executed in another process via task_supervisor
    @staticmethod
    def run(task, *args, **kwargs):
        # .. process task
        return '<task result>'

    def process_result(self, result):
        # process result
```

main.py

```
from myworker import MyWorker

worker = MyWorker()
worker.start()
# .....
worker.put_threadsafe('task')
# .....
worker.stop()
```

Workers

BackgroundWorker

Background worker is a worker which continuously run executor function in a loop without any condition. Loop of this worker is synchronous and is started in separate thread instantly.

```
# with annotation - function becomes worker executor
from atasker import background_worker

@background_worker
def myfunc(*args, **kwargs):
    print('I am background worker')

# with class
from atasker import BackgroundWorker

class MyWorker(BackgroundWorker):

    def run(self, *args, **kwargs):
        print('I am a worker too')

myfunc.start()

myworker2 = MyWorker()
myworker2.start()

# ............

# stop first worker
myfunc.stop()
# stop 2nd worker, don't wait until it is really stopped
myworker2.stop(wait=False)
```

BackgroundAsyncWorker

Similar to *BackgroundWorker* but used for async executor functions. Has additional parameter *loop*= (*_loop* in start function) to specify either async event loop or *aloop* object. By default either task supervisor event loop or task supervisor default aloop is used.

```
# with annotation - function becomes worker executor
from atasker import background_worker

@background_worker
async def async_worker(**kwargs):
    print('I am async worker')

async_worker.start()

# with class
from atasker import BackgroundAsyncWorker

class MyWorker(BackgroundAsyncWorker):

    async def run(self, *args, **kwargs):
        print('I am async worker too')
```

(continues on next page)

(continued from previous page)

```
worker = MyWorker()
worker.start()
```

BackgroundQueueWorker

Background worker which gets data from asynchronous queue and passes it to synchronous or Asynchronous executor.

Queue worker is created as soon as annotator detects *q=True* or *queue=True* param. Default queue is *asyncio.queues.Queue*. If you want to use e.g. priority queue, specify its class instead of just *True*.

```
# with annotation - function becomes worker executor
from atasker import background_worker

@background_worker(q=True)
def f(task, **kwargs):
    print('Got task from queue: {}'.format(task))

@background_worker(q=asyncio.queues.PriorityQueue)
def f2(task, **kwargs):
    print('Got task from queue too: {}'.format(task))

# with class
from atasker import BackgroundQueueWorker

class MyWorker(BackgroundQueueWorker):

    def run(self, task, *args, **kwargs):
        print('my task is {}'.format(task))


f.start()
f2.start()
worker3 = MyWorker()
worker3.start()
f.put_threadsafe('task 1')
f2.put_threadsafe('task 2')
worker3.put_threadsafe('task 3')
```

put method is used to put task into worker's queue. The method is thread-safe.

BackgroundEventWorker

Background worker which runs asynchronous loop waiting for the event and launches synchronous or asynchronous executor when it's happened.

Event worker is created as soon as annotator detects *e=True* or *event=True* param.

```
# with annotation - function becomes worker executor
from atasker import background_worker

@background_worker(e=True)
def f(task, **kwargs):
    print('happened')
```

(continues on next page)

(continued from previous page)

```
# with class
from atasker import BackgroundEventWorker

class MyWorker(BackgroundEventWorker):

    def run(self, *args, **kwargs):
        print('happened')


f.start()
worker3 = MyWorker()
worker3.start()
f.trigger_threadsafe()
worker3.trigger_threadsafe()
```

trigger_threadsafe method is used to put task into worker's queue. The method is thread-safe. If worker is triggered from the same asyncio loop, **trigger** method can be used instead.

BackgroundIntervalWorker

Background worker which runs synchronous or asynchronous executor function with the specified interval or delay.

Worker initial parameters:

- **interval** run executor with a specified interval (in seconds)
- **delay** delay *between* executor launches
- **delay_before** delay *before* executor launch

Parameters *interval* and *delay* can not be used together. All parameters can be overriden during startup by adding _ prefix (e.g. *worker.start(_interval=1)*)

Background interval worker is created automatically, as soon as annotator detects one of the parameters above:

```
@background_worker(interval=1)
def myfunc(**kwargs):
    print('I run every second!')

@background_worker(interval=1)
async def myfunc2(**kwargs):
    print('I run every second and I am async!')

myfunc.start()
myfunc2.start()
```

As well as event worker, **BackgroundIntervalWorker** supports manual executor triggering with *worker.trigger()* and *worker.trigger_threadsafe()*

6.4.5 Task collections

Task collections are useful when you need to run a pack of tasks e.g. on program startup or shutdown. Currently collections support running task functions only either in a foreground (one-by-one) or as the threads.

Function priority can be specified either as *TASK_** (e.g. *TASK_NORMAL*) or as a number (lower = higher priority).

FunctionCollection

Simple collection of functions.

```
from atasker import FunctionCollection, TASK_LOW, TASK_HIGH

def error(**kwargs):
   import traceback
   traceback.print_exc()

startup = FunctionCollection(on_error=error)

@startup
def f1():
    return 1

@startup(priority=TASK_HIGH)
def f2():
    return 2

@startup(priority=TASK_LOW)
def f3():
    return 3

result, all_ok = startup.execute()
```

TaskCollection

Same as function collection, but stored functions are started as tasks in threads.

Methods *execute()* and *run()* return result when all tasks in collection are finished.

6.4.6 Thread local proxy

```
from atasker import g

if not g.has('db'):
    g.set('db', <new_db_connection>)
```

Supports methods:

6.4.7 Locker helper/decorator

```
from atasker import Locker

def critical_exception():
    # do something, e.g. restart/kill myself
    import os, signal
    os.kill(os.getpid(), signal.SIGKILL)

lock1 = Locker(mod='main', timeout=5)
lock1.critical = critical_exception
```

(continues on next page)

(continued from previous page)

```
# use as decorator
@lock1
def test():
    # thread-safe access to resources locked with lock1

# with
with lock1:
    # thread-safe access to resources locked with lock1
```

Supports methods:

6.4.8 Debugging

The library uses logger "atasker" to log all events.

Additionally, for debug messages, method *atasker.set_debug()* should be called.