starlog Documentation

Release 1.1.1a1

Michael Peick

Apr 21, 2019

Contents:

starlog is a python library to improve python's standard logger capabilities.

Highlights:

- *Status handler*: log status lines in regular intervals
- *Multiprocess handler* bubble up messages of sub-processes to the main process when you need to deal with multiple processes

Installation of the latest stable version:

```
pip install 'starlog[zmq]'
```

Installation of the latest pre-release / developer version:

```
pip install --pre 'starlog[zmq]'
```

Installation of the latest pre-release / developer version from git:

```
pip install 'git+https://gitlab.com/peick/starlog#egg=starlog[zmq]'
```

CHAPTER 1

Status handler

A log handler that examines every log and does some statistics on it. It generates a log message in regular intervals.

Metrics that are collected:

- the number of messages per log level, available in formatter as:
  - `%(DEBUG)d`
  - `%(INFO)d`
  - `%(WARN)d`
  - `%(ERROR)d`
  - `%(FATAL)d`
- the size of messages per log level, available in formatter as
  - `%(DEBUG-SIZE)d`
  - `%(INFO-SIZE)d`
  - `%(WARN-SIZE)d`
  - `%(ERROR-SIZE)d`
  - `%(FATAL-SIZE)d`
- custom values examples:

```
from starlog import inc
logger.info('one more log line', extra=inc('requests').inc('2xx'))
logger.info('bad request', extra=inc('4xx').update(error='Invalid input'))
```

  - `%(requests)d`
  - `%(2xx)d`
  - `%(4xx)d`

Use case: write out status logs every some seconds to standard out and sent full logs to syslog or a file.

class starlog.**StatusHandler**(*interval='5s'*, *logger='starlog.status'*)

StatusHandler is a log handler that aggregates log messages and generates a status log message in regular intervals.

Parameters **interval** (*str or int*) – the log status interval. Values must be of the format **Xs** (seconds), **Xm** (minutes) or **Xh** (hours), where **X** is a positive integer value. If you pass an integer instead of a string, then this value is taken as the number of seconds. Examples: **15s**, **5m**, **1h**

Example:

```
import logging
import starlog
import sys
import time


formatter = logging.Formatter('Seen info messages: %(INFO)d')
handler = starlog.StatusHandler('1s')
logging.root.addHandler(handler)
logging.root.setLevel(logging.DEBUG)

stdout = logging.StreamHandler(sys.stdout)
stdout.setFormatter(formatter)
logging.getLogger('starlog.status').addHandler(stdout)

logging.root.info('you will not see this message - just the status')

time.sleep(5)

# output is:
>>> Seen info messages: 1
>>> Seen info messages: 0
>>> Seen info messages: 0
>>> Seen info messages: 0
>>> Seen info messages: 0
```

CHAPTER 2

Multiprocess handler

Defines several log handlers to operate in a multi-process application.

All log handler defined here are propagating log messages generated in a sub-process to a central log process. From the central log process it's easier to handle all log messages to a target log handler, like a file or syslog.

It depends on the way you created the sub-process to choose the right log handler. Here's a short overview:

Sub-Process Created By	Compatible Log Handler
os.fork()	*starlog.ZmqHandler*
multiprocessing.Process()	*starlog.MultiprocessHandler*, flexlog.ZmqHandler
multiprocessing.Process() in "spawn" mode	*starlog.ZmqHandler*

class starlog.**MultiprocessHandler**(*queue=None*, *manager_queue=True*, *logger='starlog.logsink'*)

The MultiprocessHandler uses multiprocessing.Queue to send log records between processes.

All subprocesses sends messages to the queue in the *emit* method. A background thread of the main process retrieves these messages and delegates them to the logger (default: starlog.sink) in the main process.

The MultiprocessHandler is expected to be set up by the main process.

Parameters

- **queue** – A multiprocessing capable queue. If not set, then a new multiprocessing queue is created.
- **manager_queue** (*bool*) – If *queue* is *None* and this argument is set to *True*, then a new queue is created by calling multiprocessing.managers.SyncManager.Queue().
- **logger** (*str*) – name of the logger where log records are send to in the main process.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

class starlog.**ZmqHandler**(*address='tcp://127.0.0.1:5557'*, *logger='starlog.logsink'*)

The ZmqHandler creates a zmq connection between the main process and its children processes. The main process establishes a `zmq.PULL` socket. Child processes sets up a `zmq.PUSH` socket.

All subprocesses sends messages to the zmq connection in the *emit* method. A background thread of the main process retrieves these messages and delegates them to the `logger` (default: `starlog.sink`) in the main process.

The ZmqHandler is expected to be set up by the main process.

Parameters

- **address** (*str*) – the address of the connection which is passed to `zmq.Socket.connect()` / `zmq.Socket.bind()`. Examples: `tcp://127.0.0.1:12345`, `tcp://127.0.0.1`, `ipc:///tmp/log.sock`. Omitting the port in a `tcp://` address will bind the socket to a random port.
- **logger** (*str*) – name of the logger where log records are send to in the main process.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

CHAPTER 3

Lookback Handler

class starlog.**LookbackHandler**(*capacity*, *max_age=3600*, *flush_level=40*, ***kwargs*)

LookbackHandler is a handler which buffers logging records.

Flushing occurs whenever an event of certain severity or greater is seen.

When the buffer is full, logging records of lower severity levels are dropped.

Parameters

- **capacity** – the number of logging records to buffer per thread/process
- **max_age** (*int*) – the number of seconds after which logging records are dropped
- **flush_level** – flush buffered logging records if a logging record seen with this log level or higher

emit(*record*)

Emit a record.

Append the record. If shouldFlush() tells us to, call flush_sub_buffer() to process the buffer.

flush()

Flush all sub buffers.

flush_sub_buffer(*records*)

Sends buffered records of the thread/process record buffer to the target handler.

key(*record*)

Key of the record to identify the sub buffer. It returns a tuple of thread-id and process id of the record.

shouldFlush(*record*)

Check for record at the flushLevel or higher.

trim()

Deletes buffered logging records reaching the buffer capacity. Also deletes logging records older than max_age.

CHAPTER 4

Examples

4.1 Status Handler

Execute with `python doc/code/demo_status_handler.py --zmq`

4.1.1 code/demo_status_handler.py

```
"""Outputs status logs::

    2019-03-05 23:53:31 log messages: 1 ERROR, 4 WARNING 5 INFO
    2019-03-05 23:53:36 log messages: 4 ERROR, 4 WARNING 6 INFO
    2019-03-05 23:53:41 log messages: 6 ERROR, 1 WARNING 6 INFO
    2019-03-05 23:53:46 log messages: 2 ERROR, 3 WARNING 6 INFO
    2019-03-05 23:53:51 log messages: 3 ERROR, 2 WARNING 6 INFO
    2019-03-05 23:53:56 log messages: 0 ERROR, 3 WARNING 5 INFO
"""
import logging
import logging.config
import os
import random
import time


_log = logging.getLogger(__name__)


def _random_log_entry():
    level = random.choice(['info', 'warning', 'error'])
    if level == 'info':
        _log.info('testing info logging')
    if level == 'warning':
        _log.warning('testing warning logging')
```

(continues on next page)

(continued from previous page)

```
    if level == 'error':
        _log.error('testing error logging')


def generate_logs():
    # logs for 30 seconds
    start = time.time()

    duration = 0
    while duration < 30:
        _random_log_entry()
        time.sleep(random.random())
        duration = time.time() - start


def main():
    here = os.path.dirname(os.path.abspath(__file__))
    log_config_path = os.path.join(here, 'logging.conf')
    logging.config.fileConfig(log_config_path, disable_existing_loggers=False)

    generate_logs()


if __name__ == '__main__':
    main()
```

4.1.2 code/logging.conf

```
# minimalistic sample configuration demonstraging starlog.StatusHandler
#
# The 'status' handler aggregates all log messages occurring at the 'root'
# logger. Every 5 seconds a status log line is sent by 'starlog.status' to
# its handler 'status_stdout'
[loggers]
keys = root, starlog.status

[handlers]
keys = status, status_stdout

[formatters]
keys = status

[logger_root]
level = NOTSET
handlers = status

[logger_starlog.status]
handlers = status_stdout
qualname = starlog.status

[handler_status]
class = starlog.StatusHandler
args = ()

[handler_status_stdout]
```

(continues on next page)

(continued from previous page)

```
class = StreamHandler
args = (sys.stdout, )
formatter = status

[formatter_status]
format = %(asctime)s log messages: %(ERROR)d ERROR, %(WARNING)d WARNING %(INFO)d INFO
datefmt = %Y-%m-%d %H:%M:%S
```

4.2 ZMQ Handler

Execute with `python doc/code/demo_multi_processing.py --zmq`

4.2.1 code/demo_multi_processing.py

```
from __future__ import print_function
import logging
import logging.config
import os
import random
import multiprocessing
import time
from argparse import ArgumentParser

from starlog import inc


N = 5
T = 5


def _random_log_entry():
    loggername = random.choice(['example', 'example.app', None])
    logger = logging.getLogger(loggername)

    level = random.choice(['info', 'warning', 'error'])
    if level == 'info':
        logger.info('testing info logging',
                    extra=inc('requests').inc('foo').update({'OTHER': True}))
    if level == 'warning':
        logger.warning('testing warning logging')
    if level == 'error':
        logger.error('testing error logging')


def log_process():
    try:
        # logs for some seconds
        start = time.time()

        duration = 0
        while duration < T:
            _random_log_entry()
```

(continues on next page)

(continued from previous page)

```
            time.sleep(random.random())
            duration = time.time() - start
        print("%s done" % (os.getpid(), ))
    except KeyboardInterrupt:
        pass


def main_loop_with_os_fork():
    print("starting main loop")

    pids = []

    for ti in range(N):
        pid = os.fork()
        if pid == 0:
            log_process()
            return

        pids.append(pid)

    _waitpids(pids)


def _waitpids(pids):
    for pid in pids:
        os.waitpid(pid, 0)


def main_loop_with_multiprocessing_process():
    print("starting main loop")

    px = []

    for ti in range(N):
        p = multiprocessing.Process(target=log_process)
        p.start()
        px.append(p)

    _join_processes(px)


def _join_processes(px):
    # if MultiprocessHandler parameter manager_queue is set to False,
    # then this is the pattern to join exited processes. The reason is,
    # multiprocessing.Queue sometimes blocks the process at p.join forever
    # although it left the run method.
    px[0].join(T + 1)

    for p in px:
        p.join(timeout=1)

    for p in px:
        if p.is_alive():
            print("terminating %s %s" % (p, p.pid))
            p.terminate()
            p.join()
    print("all processes exited")
```

(continues on next page)

(continued from previous page)

```
def get_cli_args():
    parser = ArgumentParser()

    parser.add_argument(
        '--status',
        action='store_true',
        default=False,
        help='use starlog.StatusLogger')
    parser.add_argument(
        '--zmq',
        action='store_true',
        default=False,
        help='use starlog.ZmqHandler. If not set, then ' +
             'starlog.MultiprocessHandler is used. ')
    parser.add_argument(
        '--duration',
        type=int,
        default=5,
        help='How long the test should run in seconds')
    parser.add_argument(
        '--processes',
        type=int,
        default=5,
        help='The number of processes to start')
    parser.add_argument(
        '--fork',
        action='store_true',
        default=False,
        help='Use os.fork() to create a sub process. Default: use ' +
             'multiprocessing.Process')

    return parser.parse_args()


def main():
    global N, T

    args = get_cli_args()

    N = args.processes
    T = args.duration

    print("configuring logging")
    here = os.path.dirname(os.path.abspath(__file__))

    configs = {
        # status, zmq => log config file
        (False, False): 'logging_multiprocess.conf',
        (True,  False): 'logging_multiprocess_status.conf',
        (False,  True): 'logging_zmq.conf',
        (True,   True): 'logging_zmq_status.conf'}

    log_config_path = os.path.join(here, 'logging_zmq.conf')
    basename = configs[(args.status, args.zmq)]
    log_config_path = os.path.join(here, basename)
```

(continues on next page)

(continued from previous page)

```
    logging.config.fileConfig(log_config_path, disable_existing_loggers=False)

    print("using %s" % basename)

    if args.fork:
        if not args.zmq:
            print('\nWARNING: the combination of os.fork and ' +
                  'starlog.MultiprocessHandler is not reliable and thus not ' +
                  'recommend\n')
        main_loop_with_os_fork()
    else:
        main_loop_with_multiprocessing_process()


if __name__ == '__main__':
    main()
```

4.2.2 code/logging_zmq.conf

```
# minimalistic sample configuration demonstrating starlog.ZmqHandler
#
# All log records from any process is forwarded to the 'starlog.logsink'
# logger, where it's delegated to the 'stdout' handler.
[loggers]
keys = root, starlog.logsink, starlog.logsink.example

[handlers]
keys = multiprocess, stdout

[formatters]
keys = generic

[logger_root]
level = NOTSET
handlers = multiprocess

[logger_starlog.logsink]
level = NOTSET
handlers = stdout
propagate = 0
qualname = starlog.logsink

# do filtering / log handling in a central place for 'example' loggers
[logger_starlog.logsink.example]
level = ERROR
handlers =
propagate = 1
qualname = starlog.logsink.example

[handler_multiprocess]
class = starlog.ZmqHandler
# args = ('tcp://127.0.0.1:5557', )
args = ('tcp://127.0.0.1', )
# args = ()
formatter = generic
```

(continues on next page)

(continued from previous page)

```
[handler_stdout]
class = StreamHandler
args = (sys.stdout, )
formatter = generic

[formatter_generic]
format = %(asctime)s [%(name)s-%(process)d] %(levelname)s: %(message)s
datefmt = %Y-%m-%d %H:%M:%S
```

4.3 Lookback Handler

Execute with `python doc/code/demo_lookback.py`

4.3.1 code/demo_lookback_handler.py

```
"""Outputs logs::

    2019-03-24 17:50:17 [   INFO] message 3
    2019-03-24 17:50:18 [   INFO] message 4
    2019-03-24 17:50:18 [   INFO] message 5
    2019-03-24 17:50:18 [  ERROR] message 6
    2019-03-24 17:50:24 [   INFO] message 16
    2019-03-24 17:50:24 [   INFO] message 17
    2019-03-24 17:50:25 [   INFO] message 18
    2019-03-24 17:50:25 [  ERROR] message 19
    2019-03-24 17:50:26 [   INFO] message 20
    2019-03-24 17:50:26 [WARNING] message 21
    2019-03-24 17:50:27 [WARNING] message 22
    2019-03-24 17:50:27 [  ERROR] message 23
"""
import logging
import logging.config
import os
import random
import time


_log = logging.getLogger(__name__)


def _random_log_entry(number):
    rand = random.random()
    if rand < 0.05:
        level = logging.ERROR
    elif rand < 0.2:
        level = logging.WARNING
    else:
        level = logging.INFO

    if level == logging.INFO:
        _log.info('message %d', number)
```

(continues on next page)

(continued from previous page)

```
    if level == logging.WARNING:
        _log.warning('message %d', number)
    if level == logging.ERROR:
        _log.error('message %d', number)


def generate_logs():
    # logs for 30 seconds
    start = time.time()

    duration = 0
    number = 0
    while duration < 30:
        _random_log_entry(number)
        time.sleep(random.random())
        duration = time.time() - start
        number += 1


def main():
    here = os.path.dirname(os.path.abspath(__file__))
    log_config_path = os.path.join(here, 'logging_lookback.conf')
    logging.config.fileConfig(log_config_path, disable_existing_loggers=False)

    generate_logs()


if __name__ == '__main__':
    main()
```

4.3.2 code/logging_lookback.conf

```
# minimalistic sample configuration demonstrating starlog.LookbackHandler
[loggers]
keys = root

[handlers]
keys = lookback_stdout, stdout

[formatters]
keys = generic

[logger_root]
level = NOTSET
handlers = lookback_stdout

[handler_lookback_stdout]
class = starlog.LookbackHandler
# capacity=100, max_age=5
args = (100, 5)
target = stdout
formatter = generic

[handler_stdout]
class = StreamHandler
```

(continues on next page)

(continued from previous page)

```
args = (sys.stdout, )
formatter = generic

[formatter_generic]
format = %(asctime)s [%(levelname)7s] %(message)s
datefmt = %Y-%m-%d %H:%M:%S
```

4.4 gunicorn demo

4.4.1 code/gunicorn/app.py

```
import logging

from flask import Flask

app = Flask(__name__)

_log = logging.getLogger('app')


@app.route("/")
def hello():
    _log.info('incoming request')
    return "Hello World!"


if __name__ != '__main__':
    _log.info('my app is starting')
```

4.4.2 code/gunicorn/log.conf

```
# Log record flow:
#   All log records are delegated by the "zmq" handler to the "starlog.logsink"
#   logger. The "starlog.sink" logger delegates all log records to the handlers
#   "status" and "log_file". The "status" handler aggregates log records. A status
#   line is generated in regular intervals by the logger "starlog.status" where
#   it's written to stdout and to a separate status file.
#
# logger "root"
#   -> handler "zmq"
#     -> logger "starlog.sink"
#       -> handler "log_file"
#       -> handler "status"
#         -> logger "starlog.status" (every 30 seconds)
#           -> handler "status_stdout"
#           -> handler "status_log_file"

[loggers]
keys=root, gunicorn.error, gunicorn.access, starlog.status, starlog.logsink

[handlers]
```

(continues on next page)

(continued from previous page)

```
keys=log_file, status, status_log_file, status_stdout, zmq

[formatters]
keys=generic, status

[logger_root]
level=INFO
handlers=zmq

[logger_gunicorn.error]
level=INFO
handlers=
propagate=1
qualname=gunicorn.error

[logger_gunicorn.access]
level=INFO
handlers=
propagate=1
qualname=gunicorn.access

[logger_starlog.status]
level=NOTSET
handlers=status_log_file, status_stdout
propagate=0
qualname=starlog.status

[logger_starlog.logsink]
level=NOTSET
handlers=log_file, status
propagate=0
qualname=starlog.logsink


[handler_zmq]
class = starlog.ZmqHandler
args=('ipc://log.sock',)
formatter=generic

[handler_log_file]
level=DEBUG
class=logging.FileHandler
formatter=generic
args=('app.log', )

[handler_status_log_file]
level=NOTSET
class=logging.FileHandler
formatter=status
args=('status.log', )

[handler_status_stdout]
level=NOTSET
class=logging.StreamHandler
formatter=status
args=(sys.stdout,)
```

(continues on next page)

(continued from previous page)

```
[handler_status]
class = starlog.StatusHandler
args = ('30s',)
formatter = status


[formatter_generic]
format=%(asctime)s [%(process)d:%(name)s:%(lineno)s] [%(levelname)s] %(message)s
datefmt=%Y-%m-%d %H:%M:%S
class=logging.Formatter

[formatter_status]
datefmt=%Y-%m-%d %H:%M:%S
format = %(asctime)s log messages: %(CRITICAL)d CRITICAL, %(ERROR)d ERROR,
↪%(WARNING)d WARNING %(INFO)d INFO, %(DEBUG)d DEBUG.
```

4.4.3 Output

```
$ gunicorn app:app --log-config log.conf
2019-03-17 15:47:09,927 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 5 INFO, 0 DEBUG.
2019-03-17 15:47:39,963 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 0 INFO, 0 DEBUG.
2019-03-17 15:47:55,983 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 3 INFO, 0 DEBUG.
```

```
$ tail -F app.log status.log

==> app.log <==
2019-03-17 15:46:39 [22283:gunicorn.error:271] [INFO] Starting gunicorn 19.9.0
2019-03-17 15:46:39 [22283:gunicorn.error:271] [INFO] Listening at: http://127.0.0.
↪1:8000 (22283)
2019-03-17 15:46:39 [22283:gunicorn.error:271] [INFO] Using worker: sync
2019-03-17 15:46:39 [22291:gunicorn.error:271] [INFO] Booting worker with pid: 22291
2019-03-17 15:46:39 [22291:app:17] [INFO] my app is starting

==> status.log <==
2019-03-17 15:47:09 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 5 INFO, 0 DEBUG.
2019-03-17 15:47:39 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 0 INFO, 0 DEBUG.

==> app.log <==
2019-03-17 15:47:54 [22283:gunicorn.error:271] [INFO] Handling signal: int
2019-03-17 15:47:54 [22291:gunicorn.error:271] [INFO] Worker exiting (pid: 22291)
2019-03-17 15:47:54 [22283:gunicorn.error:271] [INFO] Shutting down: Master

==> status.log <==
2019-03-17 15:47:55 log messages: 0 CRITICAL, 0 ERROR, 0 WARNING 3 INFO, 0 DEBUG.
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index