cotyledon Documentation

Mehdi Abaakouk

Dec 21, 2018

Contents

CHAPTER 1

Contents:

1.1 Installation

At the command line:

```
$ pip install cotyledon
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv cotyledon
$ pip install cotyledon
```

1.2 API

class cotyledon.**Service**(*worker_id*)

Base class for a service

This class will be executed in a new child process/worker `ServiceWorker` of a *ServiceManager*. It registers signals to manager the reloading and the ending of the process.

Methods *run()*, *terminate()* and *reload()* are optional.

__init__(*worker_id*)

Create a new Service

Parameters **worker_id** (*int*) – the identifier of this service instance

The identifier of the worker can be used for workload repartition because it's consistent and always the same.

For example, if the number of workers for this service is 3, one will got 0, the second got 1 and the last got 2. if worker_id 1 died, the new spawned process will got 1 again.

graceful_shutdown_timeout = None
Timeout after which a gracefully shutdown service will exit. zero means endless wait. None means same as ServiceManager that launch the service

name = None
Service name used in the process title and the log messages in additionnal of the worker_id.

reload()
Reloading of the service

This method will be executed when the Service receives a SIGHUP.

If not implemented the process will just end with status 0 and `ServiceRunner` will start a new fresh process for this service with the same worker_id.

Any exceptions raised by this method will be logged and the worker will exit with status 1.

run()
Method representing the service activity

If not implemented the process will just wait to receive an ending signal.

This method is ran into the thread and can block or return as needed

Any exceptions raised by this method will be logged and the worker will exit with status 1.

terminate()
Gracefully shutdown the service

This method will be executed when the Service has to shutdown cleanly.

If not implemented the process will just end with status 0.

To customize the exit code, the `SystemExit` exception can be used.

Any exceptions raised by this method will be logged and the worker will exit with status 1.

class cotyledon.**ServiceManager**(*wait_interval=0.01*, *graceful_shutdown_timeout=60*)
Manage lifetimes of services

ServiceManager acts as a master process that controls the lifetime of children processes and restart them if they die unexpectedly. It also propagate some signals (SIGTERM, SIGALRM, SIGINT and SIGHUP) to them.

Each child process (`ServiceWorker`) runs an instance of a *Service*.

An application must create only one *ServiceManager* class and use *ServiceManager.run()* as main loop of the application.

Usage:

```
class MyService(Service):
    def __init__(self, worker_id, myconf):
        super(MyService, self).__init__(worker_id)
        preparing_my_job(myconf)
        self.running = True

    def run(self):
        while self.running:
            do_my_job()

    def terminate(self):
        self.running = False
        gracefully_stop_my_jobs()
```

(continues on next page)

(continued from previous page)

```
    def reload(self):
        restart_my_job()


class MyManager(ServiceManager):
    def __init__(self):
        super(MyManager, self).__init__()
        self.register_hooks(on_reload=self.reload)

        conf = {'foobar': 2}
        self.service_id = self.add(MyService, 5, conf)

    def reload(self):
        self.reconfigure(self.service_id, 10)

MyManager().run()
```

This will create 5 children processes running the service MyService.

__init__(*wait_interval=0.01*, *graceful_shutdown_timeout=60*)
Creates the ServiceManager object

Parameters **wait_interval** (*float*) – time between each new process spawn

add(*service*, *workers=1*, *args=None*, *kwargs=None*)
Add a new service to the ServiceManager

Parameters

- **service** (*callable*) – callable that return an instance of *Service*
- **workers** (*int*) – number of processes/workers for this service
- **args** (*tuple*) – additional positional arguments for this service
- **kwargs** (*dict*) – additional keyword arguments for this service

Returns a service id

Return type uuid.uuid4

reconfigure(*service_id*, *workers*)
Reconfigure a service registered in ServiceManager

Parameters

- **service_id** (*uuid.uuid4*) – the service id
- **workers** (*int*) – number of processes/workers for this service

Raises ValueError

register_hooks(*on_terminate=None*, *on_reload=None*, *on_new_worker=None*, *on_dead_worker=None*)
Register hook methods

This can be callable multiple times to add more hooks, hooks are executed in added order. If a hook raised an exception, next hooks will be not executed.

Parameters

- **on_terminate** (*callable()*) – method called on SIGTERM
- **on_reload** (*callable()*) – method called on SIGHUP

- **on_new_worker** (*callable(service_id, worker_id, exit_code)*) – method called in the child process when this one is ready
- **on_new_worker** – method called when a child died

If window support is planned, hooks callable must support to be pickle.pickle(). See CPython multiprocessing module documentation for more detail.

run()
Start and supervise services workers

This method will start and supervise all children processes until the master process asked to shutdown by a SIGTERM.

All spawned processes are part of the same unix process group.

1.3 Examples

```
# Licensed under the Apache License, Version 2.0 (the "License"); you may
# not use this file except in compliance with the License. You may obtain
# a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
# WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
# License for the specific language governing permissions and limitations
# under the License.

import logging
import os
import signal
import socket
import sys
import threading
import time

from oslo_config import cfg

import cotyledon
from cotyledon import _utils
from cotyledon import oslo_config_glue

if len(sys.argv) >= 3:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("127.0.0.1", int(sys.argv[2])))
    if os.name == "posix":
        stream = os.fdopen(s.fileno(), 'w')
    else:
        stream = s.makefile()
    logging.basicConfig(level=logging.DEBUG, stream=stream)
else:
    logging.basicConfig(level=logging.DEBUG)

LOG = logging.getLogger("cotyledon.tests.examples")
```

(continues on next page)

(continued from previous page)

```
# We don't want functional tests to wait for this:
cotyledon.ServiceManager._slowdown_respawn_if_needed = lambda *args: True


class FullService(cotyledon.Service):
    name = "heavy"

    def __init__(self, worker_id):
        super(FullService, self).__init__(worker_id)
        self._shutdown = threading.Event()
        LOG.error("%s init" % self.name)

    def run(self):
        LOG.error("%s run" % self.name)
        self._shutdown.wait()

    def terminate(self):
        LOG.error("%s terminate" % self.name)
        self._shutdown.set()
        sys.exit(42)

    def reload(self):
        LOG.error("%s reload" % self.name)


class LigthService(cotyledon.Service):
    name = "light"


class BuggyService(cotyledon.Service):
    name = "buggy"
    graceful_shutdown_timeout = 1

    def terminate(self):
        time.sleep(60)
        LOG.error("time.sleep done")


class BadlyCodedService(cotyledon.Service):
    def run(self):
        raise Exception("so badly coded service")


class OsloService(cotyledon.Service):
    name = "oslo"


class WindowService(cotyledon.Service):
    name = "window"


def on_terminate():
    LOG.error("master terminate hook")


def on_terminate2():
    LOG.error("master terminate2 hook")
```

(continues on next page)

(continued from previous page)

```
def on_reload():
    LOG.error("master reload hook")


def example_app():
    p = cotyledon.ServiceManager()
    p.add(FullService, 2)
    service_id = p.add(LigthService, 5)
    p.reconfigure(service_id, 1)
    p.register_hooks(on_terminate, on_reload)
    p.register_hooks(on_terminate2)
    p.run()


def buggy_app():
    p = cotyledon.ServiceManager()
    p.add(BuggyService)
    p.run()


def oslo_app():
    conf = cfg.ConfigOpts()
    conf([], project='openstack-app', validate_default_values=True,
         version="0.1")

    p = cotyledon.ServiceManager()
    oslo_config_glue.setup(p, conf)
    p.add(OsloService)
    p.run()


def window_sanity_check():
    p = cotyledon.ServiceManager()
    p.add(LigthService)
    t = _utils.spawn(p.run)
    time.sleep(10)
    os.kill(os.getpid(), signal.SIGTERM)
    t.join()


def badly_coded_app():
    p = cotyledon.ServiceManager()
    p.add(BadlyCodedService)
    p.run()


def exit_on_special_child_app():
    p = cotyledon.ServiceManager()
    sid = p.add(LigthService, 1)
    p.add(FullService, 2)

    def on_dead_worker(service_id, worker_id, exit_code):
        # Shutdown everybody if LigthService died
        if service_id == sid:
            p.shutdown()
```

(continues on next page)

(continued from previous page)

```
    p.register_hooks(on_dead_worker=on_dead_worker)
    p.run()


def sigterm_during_init():

    def kill():
        os.kill(os.getpid(), signal.SIGTERM)

    # Kill in 0.01 sec
    threading.Timer(0.01, kill).start()
    p = cotyledon.ServiceManager()
    p.add(LigthService, 10)
    p.run()


if __name__ == '__main__':
    globals()[sys.argv[1]]()
```

1.4 Note about non posix support

On non-posix platform the lib have some limitation.

When the master process receives a signal, the propagation to children processes is done manually on known pids instead of the process group.

SIGHUP is of course not supported.

Processes termination are not done gracefully. Even we use Popen.terminate(), children don't received SIGTERM/SIGBREAK as expected. The module multiprocessing doesn't allow to set CREATE_NEW_PROCESS_GROUP on new processes and catch SIGBREAK.

Also signal handlers are only run every second instead of just after the signal reception because non-posix platform does not support signal.set_wakeup_fd correctly

And to finish, the processes names are not set on non-posix platform.

1.5 Oslo.service migration examples

This example shows the same application with oslo.service and cotyledon. It uses a wide range of API of oslo.service, but most applications don't really uses all of this. In most case cotyledon.ServiceManager don't need to inherited.

It doesn't show how to replace the periodic task API, if you use it you should take a look to futurist documentation

oslo.service typical application:

```
import multiprocessing
from oslo.service import service
from oslo.config import cfg

class MyService(service.Service):
    def __init__(self, conf):
        # called before os.fork()
```

(continues on next page)

(continued from previous page)

```
        self.conf = conf
        self.master_pid = os.getpid()

        self.queue = multiprocessing.Queue()

    def start(self):
        # called when application start (parent process start)
        # and
        # called just after os.fork()

        if self.master_pid == os.getpid():
            do_master_process_start()
        else:
            task = self.queue.get()
            do_child_process_start(task)


    def stop(self):
        # called when children process stop
        # and
        # called when application stop (parent process stop)
        if self.master_pid == os.getpid():
            do_master_process_stop()
        else:
            do_child_process_stop()

    def restart(self):
        # called on SIGHUP
        if self.master_pid == os.getpid():
            do_master_process_reload()
        else:
            # Can't be reach oslo.service currently prefers to
            # kill the child process for safety purpose
            do_child_process_reload()

class MyOtherService(service.Service):
    pass


class MyThirdService(service.Service):
    pass


def main():
    conf = cfg.ConfigOpts()
    service = MyService(conf)
    launcher = service.launch(conf, service, workers=2, restart_method='reload')
    launcher.launch_service(MyOtherService(), worker=conf.other_workers)

    # Obviously not recommanded, because two objects will handle the
    # lifetime of the masterp process but some application does this, so...
    launcher2 = service.launch(conf, MyThirdService(), workers=2, restart_method=
↪'restart')

    launcher.wait()
    launcher2.wait()
```

(continues on next page)

(continued from previous page)

```
    # Here, we have no way to change the number of worker dynamically.
```

Cotyledon version of the typical application:

```
import cotyledon
from cotyledon import oslo_config_glue

class MyService(cotyledon.Service):
    name = "MyService fancy name that will showup in 'ps xaf'"

    # Everything in this object will be called after os.fork()
    def __init__(self, worker_id, conf, queue):
        self.conf = conf
        self.queue = queue

    def run(self):
        # Optional method to run the child mainloop or whatever
        task = self.queue.get()
        do_child_process_start(task)

    def terminate(self):
        do_child_process_stop()

    def reload(self):
        # Done on SIGHUP after the configuration file reloading
        do_child_reload()


class MyOtherService(cotyledon.Service):
    name = "Second Service"


class MyThirdService(cotyledon.Service):
    pass


class MyServiceManager(cotyledon.ServiceManager):
    def __init__(self, conf)
        super(MetricdServiceManager, self).__init__()
        self.conf = conf
        oslo_config_glue.setup(self, self.conf, restart_method='reload')
        self.queue = multiprocessing.Queue()

        # the queue is explicitly passed to this child (it will live
        # on all of them due to the usage of os.fork() to create children)
        sm.add(MyService, workers=2, args=(self.conf, queue))
        self.other_id = sm.add(MyOtherService, workers=conf.other_workers)
        sm.add(MyThirdService, workers=2)

    def run(self):
        do_master_process_start()
        super(MyServiceManager, self).run()
        do_master_process_stop()

    def reload(self):
        # The cotyledon ServiceManager have already reloaded the oslo.config files
```

(continues on next page)

(continued from previous page)

```
        do_master_process_reload()

        # Allow to change the number of worker for MyOtherService
        self.reconfigure(self.other_id, workers=self.conf.other_workers)

def main():
    conf = cfg.ConfigOpts()
    MyServiceManager(conf).run()
```

Other examples can be found here:

- *Examples*
- https://github.com/openstack/gnocchi/blob/master/gnocchi/cli.py#L287
- https://github.com/openstack/ceilometer/blob/master/ceilometer/cmd/collector.py

1.6 Contributing

Bugs should be filed on Github: https://github.com/sileht/cotyledon/issues

Contribution can be via Github pull requests: https://github.com/sileht/cotyledon/pulls

1.7 Cotyledon

Cotyledon provides a framework for defining long-running services.

It provides handling of Unix signals, spawning of workers, supervision of children processes, daemon reloading, sd-notify, rate limiting for worker spawning, and more.

- Free software: Apache license
- Documentation: http://cotyledon.readthedocs.org/
- Source: https://github.com/sileht/cotyledon
- Bugs: https://github.com/sileht/cotyledon/issues

1.7.1 Why Cotyledon

This library is mainly used in OpenStack Telemetry projects, in replacement of *oslo.service*. However, as *oslo.service* depends on *eventlet*, a different library was needed for project that do not need it. When an application do not monkeypatch the Python standard library anymore, greenlets do not in timely fashion. That made other libraries such as Tooz or oslo.messaging to fail with e.g. their heartbeat systems. Also, processes would not exist as expected due to greenpipes never being processed.

oslo.service is actually written on top of eventlet to provide two main features:

- periodic tasks
- workers processes management

The first feature was replaced by another library called futurist and the second feature is superseded by *Cotyledon*.

Unlike *oslo.service*, **Cotyledon** have:

- The same code path when workers=1 and workers>=2
- Reload API (on SIGHUP) hooks work in case of you don't want to restarting children
- A separated API for children process termination and for master process termination
- Seatbelt to ensure only one service workers manager run at a time.
- Is signal concurrency safe.
- Support non posix platform, because it's built on top of multiprocessing module instead of os.fork
- Provide functional testing

And doesn't:

- facilitate the creation of wsgi application (sockets sharing between parent and children process). Because too many wsgi webserver already exists.

oslo.service being impossible to fix and bringing an heavy dependency on eventlet, **Cotyledon** appeared.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

G

N

R

S

T