OMQS(Offline Message Queue Service) Tutorial (V1.0)

By Dennis Wu <wj_egg@163.com> Oct. 2015

Overview

Objects

OMQS is a Python service wrapper which passing the messages to the message queue server (particularlly, using RabbitMQ). The major purpose of this service is to simplify the tasks of processing the long-time produces in our Web server backend, like sending sign up SMS, sending emails, or generating the news in notification center.

Architecture

There are two major facts that taken into account when designing the OMQS:

1. The main process must emit the message as soon as possible;
2. The message could be processed offline easily and flexibly.

For the 1st one, OMQS using the Python native IPC way - Queue, to pass the message to another process, letting the main process continue its own routine right after delivering the message.

For the 2nd one, OMQS is designed in a "Subclass-able" structure, making each core module easily customized, as well as hiding those uncommon settings and miscellaneous from the common user scenarios.

The basic architecture of OMQS is as follows:

Main Process --(emit msg)--> **Worker Process** --(msg looping)--> **Publisher threads** --(send msg)--> **MQ Server** <--(receive msg)-- **Consumer Process**

There are two types of built-in workers: Event Worker and Log Worker. Corresponding to them, two types of built-in publisher and consumer are used, one is running in synchorous mode while the other one is running in asynchronous mode.

Just as mentioned above, the OMQS is kind of "subclass-able" structure, both workers are subclass of the base class - *MsgWorker*. And both consumers are subclass of *SyncConsumer*.

> **NOTE**:
>
> Althought currently the publisher mode is bound to the corresponding worker, the consumer has no restriction on choosing the mode. And that is what happened now in current implementation: Both event consumer and log consumer are using the SyncConsumer base class.

Usage

Configuration:

Before you starting to use the OMQS service, it is highly recommended to check the global configuration file. OMQS uses a global configuration file called **OMQS.cfg** in the main path of the OMQS module. It is a common *.ini* formated file. Here is the standard example of OMQS.cfg:

```
;------------------- Global Settings -------------------
[Global]
MQURL=amqp://artcm:111@192.168.31.225:5672/%%2F?heartbeat_interval=600

RECONNECT_INTERVAL=5
RECONNECT_TIMES=100

;------------------- Log Settings -------------------
[Log]
FOLDER=./log/
LEVEL=INFO
FORMAT=('%%(levelname) -5s %%(asctime)s %%(filename) -10s:%%(lineno) -
5d: %%(message)s')

...

; ====================================================================
=
; The following settings are the defaults for each type of MQ publishe
r
; Normally you should not change these settings
;

...

;------------------- Topic Publisher -------------------
[AsyncPublisher]
EXCHANGE=omqs.default.asyncPublisher.exchange
EXCHANGE_TYPE=topic
QUEUE=omqs.default.asyncPublisher.queue
ROUTING_KEY=omqs.default.asyncPublisher.key



[SyncPublisher]
EXCHANGE=omqs.default.syncPublisher.exchange
EXCHANGE_TYPE=topic
QUEUE=omqs.default.syncPublisher.queue
ROUTING_KEYS=omqs.default.syncPublisher.key

...
```

> **NOTED:**

> While most of the settings can be left un-modified, there is still most important one that you should check each time before you initialize your project with OMQS service: **MQURL** item, which identify the MQ server address and connection parameters.

When using this configuration file, values of the settings listed in the section *Publisher* are loaded as following priority (in descending order):

1. the parameters passed in during the calling
2. the global config file - OMQS.cfg
3. the default value hard coded in each class constants.

Basic Example:

1. Sending Message

First, you need a OMQSManager:

```
manager = OMQSManager()
```

Before sending out the messages, you need to start the manager:

```
manager.run()
```

Then, you can deliver the message using different API basing on the message type: log or event. For example, to send a info log, you can call send_info(). To send a debug log, just call send_debug(). Or you can use send_log() directly, passing in the **key** argument with the key string if you want use different strings as keys.(But be careful that you need check the same keys on the consumer side.)

```
manager.send_info(log)
manager.send_debug(log)
manager.send_event('simple event') # this event will be sent using def
ault key: 'omqs.key.event'

manager.send_log('This is a debug log', 'debug')
manager.send_event('{Email: wj_egg@163.com}', 'email_notification')
```

Fairly easy!! Isn't it?

Now you can stop the manager when complete the tasks:

```
manager.stop()
```

So, puting together:

```
if __name__ == '__main__':
    manager = OMQSManager()

    manager.run()

    t1 = time.time()
    for i in range(1, 100):
        msg = 'msg %d' % i
        manager.send_info(msg)
        if i % 10 == 0:
            manager.send_event(msg)

    t2 = time.time()
    dt = t2 - t1
    print 'total time: %f' % dt

    time.sleep(3)
    manager.stop()
```

> **NOTED**:
>
> The sleep() call before the stop() is to make sure the manager has finished sending all the messages out to the server. It is a good practice to do this whenever you try to shut down the manager.

2. Receiving Message

To receive messages, you can not use a unique manager as what you do in sending message. Instead, you need to set up the receiver, a.k.a, the consumer, by youself.

Corresponding to two built-in types of worker, there are also two types of built-in consumers: **LogConusmer**, and **EventConsumer**. The reason of this design is, generally speaking, message processing tasks would vary from each other and be much more complicated than message sending.

- ### LogConsumer

The most easy way to set up a consumer is like this:

```
consumer = LogMsgConsumer(keys=['debug', 'info', 'error'])
```

The only thing that you need to care about is the keys(conceptually, the 'topic' in MQ terminology) you want to check.

Then you can set the callback function by:

```
consumer.callback = my_callback
```

Or you can combine both steps into one initialization:

```
consumer = LogMsgConsumer(keys=['debug'], log_callback=my_callback)
```

The previous way would be helpful when you want to set up the callback later, not just at the first initialization stage.

Now you can start the consumer:

```
consumer.run()
```

Or stop it:

```
consumer.stop()
```

Before forwarding to next step, you need to be aware that each consumer would set up its own message loop in its process. That means once the consumer is running, it would block your process into a infinite looping. So if you have any other tasks besides of consuming the message, it's better to fork the consumer into a sub-process or a seperate thread.

You can use signals (like ^C or ^D) to interrupt the loop. So the recommended usage of consumer is as follows:

```
try:
    info_consumer = LogMsgConsumer(keys=['debug'], log_callback=my_cal
lback)
    info_consumer.run()

except KeyboardInterrupt:
    info_consumer.stop()

except Exception, e:
    print 'get the exception: %r' % e
    if info_consumer.ready:
        info_consumer.stop()
```

Besides of ***keys*** and ***log_callback*** , there are some other parameters you need know when you use log consumer:

Params	Description	Default Values
queue	the name of the queue which bound to the keys	None (using the default value in configure file)
queue_durable	the durable attribute of the queue	True (recommended if the message is important)

It is highly recommended to apply a explicite queue name each time when setting the ***keys*** . Because this would help the MQ server to seperate the messages in different queues from each other, helping decrease the load of each queue.

For the full description of the consumer API, please refer to the `Class Reference` Section.

- **EventConsumer**

Event Consumer is almost the same of Log Consumer, except OMQS makes some more restrictions on it. For example, there is no durable parameter in the class initialization, meaning you can not change the durable attribute of the queue. Another change you may not notice is that the ***keys*** parameter is not required but optional.

So you can set up a event consumer like this(leaving everything as defualt value):

```
consumer = EventMsgConsumer()
consumer.callback = my_callback
consumer.run()
```

At this time, the event message is send and received by using defualt key: `'omqs.key.event'`

> **NOTED:**
>
> If you leave everything as default in the consumer side, you must make sure the default value should be used in the other side(OMQSManager). For example, the event should be sent via calling `send_event()` without explicitly setting the key. So it is highly recommended to assign a key name for each message, and send/receive it in a consistent way.

A 'Hello world' Example

OK, now let's put all things together in one simple example. We call it 'Hello world'. In this sample, we need to send out a event message 'Hello world' from a *producer* process, and echo this message in another *consumer* process. And we will use each API with default settings, except we explicitly assign a customized key for the message - 'HelloKey'.

First, *producer.py*

```
# producer.py

from OMQSManager import OMQSManager
from OMQSLogManager import OMQSLogManager
import time

logManager = OMQSLogManager(name='producer.py', file_name='producer.lo
g')
logger = logManager.logger
logger.lever = 'INFO'

logger.info('Starting the manager...')
try:
    manager = OMQSManager()
    manager.run()

    logger.info('sending out the message ...')
    manager.send_event('Hello world', 'HelloKey')
except Exception, e:
    logger.error('Error: %r', e)

logger.info('Doing some other tasks ...')
# simulate other tasks
time.sleep(3)

logger.info('Shutting down the manager ....')
manager.stop()
```

Second, *consumer.py*

```
from EventMsgConsumer import EventMsgConsumer

try:
    consumer = EventMsgConsumer(keys=['HelloKey'])
    consumer.callback = hello_callback
    consumer.run()

except KeyboardInterrupt:
    consumer.stop()

except Exception, e:
    print 'get the exception: %r' % e
    if consumer.ready:
        consumer.stop()

def hello_callback(channel, method, properties, body):
    print " [x] hello_callback: %r:%r" % (method.routing_key, body,)
```

Advanced Customization:

OMQS is designed to be a sub-class-able structure. So besides of the built-in types, you can build you own class by customizing the base class.

1. Worker Customization

The base class of Worker is `MsgWorker`. To Customize MsgWorker baseclass, a few abstract methods need to be overrided:

```
class YourMsgWorker(MsgWorker):
    def __init__(self, q, name='YourMsgWorker'):
        super(YourMsgWorker, self).__init__(q, name)
        # TODO HERE ...
        # Your own class initialization

    def worker_will_run(self):
        # TODO HERE ...
        # Setting up your worker before it starts.
        # In OMQS, the most common initial works
        # are setting up the MQ publisher.
        # Two built-in publishers could be chose,
        # while you are free to build your own publisher.

    def msg_did_receive(self, msg):
        # TODO HERE ...
        # This is the main procedure of the worker, which
        # called each time a new message comes in.
        # In OMQS, you can publish the message to MQ here.

    def worker_will_stop(self):
        # TODO HERE ...
        # Finailzie your worker here.
        # In OMQS, you probably need to close the publisher
        # here.
```

For more details, please refer to the source code of *EventMsgWorker* or *LogMsgWorker*.

2. Consumer Customization

The base class of Consumer is `SyncConsumer` . Customizing SyncConsumer baseclass is much more easier than Worker. You just need to pay more attentions to the particular parameters of the baseclass:

```
class YourMsgConsumer(OMQSSyncConsumer):
    def __init__(self,
                 name='YourConsumer',
                 amqp_url=...,
                 exchange_name=...,
                 exchange_type=...,
                 exchange_durable=...,
                 queue_name=...,
                 queue_durable=...,
                 routing_keys=...,
                 no_ack=...,
                 callback=...):
```

For more details, please refer to the source code of *EventMsgConsumer* or *LogMsgConsumer*.

> **WARRNING**:
>
> As there are lots of operations on Networking and Configuration, when you customizing the worker/consumer, you should be very careful about the **exception handling**. Otherwise you main process may probably crash frequently without expectation!

OMQS Logger

To generate uniform and pretty log output, OMQS wraps a new logger basing on the Python standard *logging* module, called OMQSLogManager.

OMQSLogManager makes the log output easily configurable and customizable, exporting various settings in configuration file.

To use the OMQS logger, first, initialize a logger manager:

```
manager = OMQSLogManager()
```

This method creats a default manager using default settings, while it's highly recommended to set some important parameters when you're using the manager in practice:

```
manager = OMQSLogManager(name='YourModule', file_name='YourModule.log'
)
```

This would help you make the log output of the module seperated from each other.

Then, get the logger instance:

```
log = manager.logger
```

Then, use the instance just as freely as using the standard logging class:

```
log.info('info test')
log.debug('debug test')
log.error('error test')
...
```

And you can set the log level whenever you want ignore some logs:

```
# 'test 2' will be output
manager.level = 'DEBUG'
log.debug('debug test 2')

# 'test 3' will be ignored
manager.level = 'INFO'
log.debug('debug test 3')

# 'test 4' will be output
log.setLevel('DEBUG')
log.debug('debug test 4')
```

Log settings could be found in the **Log** section of *OMQS.cfg* file:

```
[Log]
FOLDER=./log/
LEVEL=INFO
FORMAT=[OMQS] %%(levelname) -5s %%(asctime)s %%(filename) -10s:%%(line
no) -5d: %%(message)s

; 'file', 'stream'
TYPE=stream

; True, False
ROTATION=True

; 'time', 'size'
ROTATION_TYPE=size

; max file size (MB) when ROTATION_TYPE = 'size'
SIZE=5

; max rotation time when ROTATION_TYPE = 'time'
; example:
;   1S - 1 Seconds
;   2M - 2 Minutes
;   3H - 3 Hours
;   4D - 4 Days
;   W0 - Weekday 0: Monday
;   MD - midnight
ROTATING_TIME=1S


; max rotation file number when ROTATION = True
BACKUP_COUNT=10
```

Detail description of each key:

Key	Value Type	Description
TYPE	'file' / 'stream'	Current output type: on standard console or write into file
ROTATION	True / False	If using rotation backup or not
ROTATION_TYPE	'size' / 'time'	Rotaiton mode, basing on size limitation or time trigger
SIZE	number (MB)	available if ROTAITON=True and ROTATION_TYPE=size
ROTATING_TIME	string	See the comments in the file sample above
BACKUP_COUNT	number	max rotation file number when ROTATION=True

ConfigReader

ConfigReader is a 3rd party 'ini' parsing module, which make it increadblly easiy to read and write the settings into ini configure file. For more details, please refer to

Here is a simple example. If you have a ini file *test.ini* like this:

```
[Section1]
key1=a

[Section2]
key2=b
```

Then you can retreive the values by following way:

```
C = ConfigReader()
C.read("test.ini")

print(C.Section1.key1)
print (C.Section2.key2)
```

> **NOTED**:
>
> You need to be careful the exceptions through out by the ConfigReader. It is a good

> practice to always put the config reading sections into a 'try...except...' block.

Class Reference:

(TBC)

a) Core Class

- OMQSManager
- MsgWorker
- EventMsgWorker
- LogMsgWorker
- AsyncPublisher
- SyncPublisher
- SyncConsumer
- AsyncConsumer
- LogMsgConsumer
- EventMsgConsumer

b) Utility Class:

- ConfigReader
- OMQSLogManager
- OMQSExceptions
- OMQSUtils