HotQueue Documentation

Release 1.3.88

Richard Henry

December 04, 2015

Contents

HotQueue is a Python library that allows you to use Redis as a FIFO message queue within your Python programs. Using HotQueue looks a little bit like this...

Establishing a queue:

```
>>> queue = HotQueue("myqueue")
```

Putting messages onto the queue:

```
>>> queue.put("my message")
>>> queue.put({'name': "Richard Henry", 'eyes': "blue"})
```

Getting messages off the queue:

```
>>> queue.get()
"my message"
>>> queue.get()
{'name': 'Richard Henry', 'eyes': 'blue'}
```

Iterating over a queue indefinitely, waiting if nothing is available:

```
>>> for item in queue.consume():
...     print item
```

More advanced features that make it easy to work with queues are available. To go deeper, you should read the HotQueue Tutorial.

The main advantage of the HotQueue model is that there is no queue server to run since the *redislite* module will handle starting/stopping a redis server automatically as needed. Plus, Redis is really fast!

CHAPTER 1

Installation

To install it, run:

```
pip install -U redislite-hotqueue
```

It also works with `easy_install`, if that's your jam. You can download versioned packages directly from PyPI.

The source code is available on GitHub and is a fork of the code available at‘GitHub <http://github.com/richardhenry/hotqueue>‘_.

To get help with HotQueue, use the HotQueue Users mailing list.

CHAPTER 2

Documentation

2.1 HotQueue Tutorial

A HotQueue is a simple FIFO queue that maps to a list key in Redis. The following is a brief introduction explaining how you can use HotQueue in practice with a simple example.

2.1.1 Connecting to Redis

Creating a queue is as simple as creating a *HotQueue* instance:

```
>>> from hotqueue import HotQueue
>>> queue = HotQueue("myqueue", dbfilename="/tmp/redis.rdb")
```

In this example, the queue will be stored as a Redis list named `hotqueue:myqueue`, on the *redislite* server. The `dbfilename` argument is optional; if none are given the *redislite* default setings will be used.

2.1.2 Putting Items Onto the Queue

Then you may have one (or many) Python programs pushing to the queue using *hotqueue.HotQueue.put()*:

```
>>> queue.put(4)
>>> queue.put(5)
```

You can push more than one item onto the queue at once:

```
>>> queue.put(6, "my message", 7)
```

You can safely push **any Python object** that can be pickled. Let's use Python's built-in `Decimal` as an example:

```
>>> from decimal import Decimal
>>> queue.put(Decimal('1.4'))
```

2.1.3 Getting Items Off the Queue

You can then pull items off the queue using *hotqueue.HotQueue.get()*. You would usually do this in another Python program, but you can do it wherever you like.

```
>>> queue.get()
4
>>> queue.get()
5
>>> queue.get()
6
```

```
>>> queue.get()
'my message'
>>> queue.get()
7
>>> dec = queue.get()
>>> dec
Decimal('1.4')
>>> dec + Decimal('0.3')
Decimal('1.7')
```

2.1.4 Consuming the Queue

A better way to pull items off the queue is to use *hotqueue.HotQueue.consume()*, which returns a generator that yields whenever an item is on the queue and blocks otherwise. Here's an example:

```
>>> for item in queue.consume():
...     print item
```

If you push to the queue using *hotqueue.HotQueue.put()* in another Python program, you will see this program print the message then wait indefinitely for another. Replace the `print` statement with something more interesting, like saving a record to a database, and you've created an asynchronous task.

2.1.5 Writing a Queue Worker

An *even better* way to pull items off the queue is to use the *hotqueue.HotQueue.worker()* decorator. Using this decorator is like wrapping the decorated function in a *hotqueue.HotQueue.consume()* loop. Here's an example:

```
from hotqueue import HotQueue

queue = HotQueue("myqueue", dbfilename="/tmp/redis.rdb")

@queue.worker
def square(num):
    print num * num
```

Then run the function:

```
>>> square()
```

It will wait indefinitely and print the square of any integers it pulls off the queue. Try pushing some integers to the queue in another Python program:

```
>>> queue.put(2, 3, 4)
```

To distribute the work, run a second instance of `square()`. You now have two queue workers. You can run as many workers as you like, and no two workers will ever receive the same message.

To run and manage your worker processes, you could use something like Supervisord.

2.1.6 Custom Serialization (JSON, etc)

If you don't want to use the pickle serializer, you can specify any other class or module that has the same API.

To serialize your data as JSON, you can use the json module. Here's an example:

```
>>> import json
>>> from hotqueue import HotQueue
>>> queue = HotQueue("myqueue", serializer=json, dbfilename="/tmp/redis.rdb")
>>> queue.put({'name': "Richard Henry", 'eyes': "blue"})
```

```
>>> queue.get()
{'name': 'Richard Henry', 'eyes': 'blue'}
```

JSON serialization is particularly useful if you will be accessing this Redis list from programming languages other than Python, or want to ensure that your queue can be read between Python versions.

If you can, you should use simplejson instead of json. It's updated more frequently, and can be significantly faster than the module that ships with the standard library. You should take a look at jsonpickle if you want to serialize more complex Python data structures to JSON.

Feel free to write your own serializer. Here's a dummy class to give you an idea of the API required:

```
class DummySerializer(object):
    """Serialization class that doesn't do anything. Fill in the dumps and
    loads methods with your own code.
    """
    @staticmethod
    def dumps(obj):
        """Serialize the given object."""
        return obj
    @staticmethod
    def loads(data):
        """De-serialize the given data back to an object."""
        return data
```

2.1.7 Disabling Serialization

If your messages can be converted to plain text without losing any information, then you can get some performance gains by not doing any serialization at all. This is ideal if you're queueing strings, CSV data, and so on.

To disable serialization, pass `None` to the serializer argument:

```
>>> queue = HotQueue("myqueue", serializer=None)
>>> queue.put("my,csv,data")
>>> queue.get()
"my,csv,data"
```

2.1.8 Monitoring

The following software is available for monitoring your HotQueue queue instances:

- HotWatch: Command line utility for monitoring the status of HotQueue queue instances.

2.2 API Reference

class `hotqueue.`**`HotQueue`**(*name*, *serializer=pickle*, ***kwargs*)

Simple FIFO message queue stored in a Redis list.

Parameters

- **`name`** (*str*) – name of the queue
- **`max_queue_length`** (*int*) – Maximum length the queue can grow to (default is None allows the queue to grow without any limits.
- **`serializer`** (*class, module, optional*) – the class or module to serialize msgs with, must have methods or functions named `dumps` and `loads`, pickle is the default, use `None` to store messages in plain text (suitable for strings, integers, etc)
- **`redis`** (*redis.Redis, redislite.Redis, optional*) – redis connection object, defaults to redislite.Redis with fallback to redis.Redis.

- ****kwargs** – Additional kwargs to pass to `redislite.Redis`, most commonly `dbfilename`.

Examples

```
>>> from hotqueue import HotQueue
>>> queue = HotQueue("myqueue", dbfilename="queue.rdb")
```

key
Key in Redis to store the queue

Returns The name of the key containing the queue in redis.

Return type str

clear()
Clear the queue of all messages, by deleting the Redis key.

consume(***kwargs*)
A blocking generator that yields whenever a message is waiting in the queue.

Parameters ****kwargs** – any arguments that *get()* can accept (`block` will default to `True` if not given)

Yields *object* – The deserialized object from the queue.

Examples

```
>>> queue = HotQueue("example")
>>> for msg in queue.consume(timeout=1):
...     print(msg)
my message
another message
```

get(*block=False*, *timeout=None*)
Get a message from the queue.

Parameters

- **block** (*bool*) – whether or not to wait until a msg is available in the queue before returning; `False` by default
- **timeout** (*int*) – When using `block`, if no msg is available for `timeout` in seconds, give up and return

Returns The deserialized object from the queue.

Return type object

Examples

```
>>> queue.get()
'my message'
>>> queue.get()
'another message'
```

put(**msgs*)
Put one or more messages onto the queue. Example:

```
>>> queue.put("my message")
>>> queue.put("another message")
```

To put messages onto the queue in bulk, which can be significantly faster if you have a large number of messages:

```
>>> queue.put("my message", "another message", "third message")
```

worker(**args*, ***kwargs*)

Decorator for using a function as a queue worker. Example:

```
>>> @queue.worker(timeout=1)
... def printer(msg):
...     print(msg)
>>> printer()
my message
another message
```

You can also use it without passing any keyword arguments:

```
>>> @queue.worker
... def printer(msg):
...     print(msg)
>>> printer()
my message
another message
```

Parameters **kwargs** – any arguments that *get()* can accept (`block` will default to `True` if not given)

2.3 Contributing

The source is available on GitHub. To contribute to the project, fork it on GitHub, run the tests, code analysis and style checks using the instructions below. Then send a pull request, all contributions and suggestions are welcome.

This project is a fork of the source available on GitHub.

2.3.1 Testing

The python tox tool will run the tests in multiple python virtual environments using all supported python interpreters installed on the computer system running the tests. To install it, run:

```
pip install tox
```

The tests using the redis module require that a redis-server be running with the default configuration. If the redis-server is not running the py27-redis and py34-redis test environments will fail. To run the tests in using all python interpreters installed on the system using both the redis and redisliste modules run:

```
tox
```

2.3.2 Code Analysis/Linting

The tox tool also is configured with a test envioronment to run the pylint tool. To check the code for common problems using pylint, run:

```
tox -e pylint
```

2.3.3 Style Check/PEP8

The code can be checked for compliance with the Python style guide using the pep8 tool. To do this, run:

```
tox -e pep8
```

2.3.4 Documentation

To build the package documentation using the tox tool, run:

```
tox -e build_docs
```

The resulting documentation will be in the build/sphinx/html directory.

2.4 Changelog

2.4.1 Changes in v1.3.0

- Add support for *redislite* to allow operation without installing a redis server.
- Implement CI/CD pipeline to automate the build and testing against multiple python versions.

2.4.2 Changes in v0.2.7

- `serializer` argument of `HotQueue.put` method now supports `None` for plain text serialization
- If multiple messages are passed to `HotQueue.put` they will be sent in the same command for a significant performance benefit

2.4.3 Changes in v0.2.6

- Removed the `HotQueue.__repr__` method as it is no longer supported

2.4.4 Changes in v0.2.5

- Fixed a bug in v0.2.4 that prevented install in some environments

2.4.5 Changes in v0.2.4

- `HotQueue.worker` decorator method can now be used to decorate a class method

2.4.6 Changes in v0.2.3

- Added support for custom serialization (JSON, etc)

2.4.7 Changes in v0.2.2

- Added `key_for_name` function

2.4.8 Changes in v0.2.1

- `HotQueue.worker` decorator method can now be used without any keyword arguments

2.4.9 Changes in v0.2.0

- Renamed `HotQueue.dequeue` method to `get`
- Renamed `HotQueue.enqueue` method to `put`
- Added `HotQueue.worker` decorator method
- `HotQueue.get` method now supports block and timeout arguments
- Added test suite

2.4.10 Changes in v0.1.0

- Initial release

CHAPTER 3

Requirements

- Python 2.7+ (tested on versions 2.7.10, 3.4.3, and pyp 2.6.1)
- redislite 1.0.254+

C

G

H

K

P

W