py-redismutex Documentation

Release 1.0.0

Abeer Upadhyay

Jul 07, 2022

Contents

Python implementation of mutex using redis.

Inspired from https://redis.io/topics/distlock.

CHAPTER 1

Quickstart

Install the package using *pip* command

```
pip install redismutex
```

To apply mutex to a block of code, first create a redis connection object using `redis.StrictRedis`. This connection object is necessary as all the mutex keys are stored in redis. Now use the `RedisMutex` to create a mutex object.

```
import redis
from redismutex import RedisMutex

conn = redis.StrictRedis(host='localhost', port=6379, db=1)
mutex = RedisMutex(conn)
mutex_key = 'YOUR-MUTEX-KEY'

with mutex.acquire_lock(mutex_key):
    # your blocking code
    # goes here...
    print(mutex.key, mutex.value)
```

Any other process (or thread) accessing the blocking code will wait for the mutex to unlock and only then it would be able to execute the blocking code.

NOTE: The `mutex.key` and `mutex.value` will be accessible only inside the mutex block. Right after unlocking the mutex, the `key` and `value` will be reset to `None`

CHAPTER 2

Configuring mutex object

You can manage the blocking time, poll interval and expiry of the mutex in the *RedisMutex* class. It allows the following paramters to be configured.

- **blocking** ***(bool)*** Represents if the mutex is a *blocking* mutex or a *non-blocking* one. (A non-blocking mutex does not wait for the lock to get free. If a lock exists, it simply returns). Default value is `True`.
- **block_time** ***(int)*** Maximum time (in seconds) for which the mutex will wait for the lock to get free. If the lock is still occupied after this time period, a `BlockTimeExceedError` is raised. Default value is `5`.
- **delay** ***(float)*** Represents the time interval (in seconds) after which the mutex will poll again to check if the lock has been freed. `delay` should always be less than the `block_time` else a `ValueError` is raised. Default value is `0.5`.
- **expiry** ***(int)*** Represents the "lock validity time" (in seconds). If the mutex fails to unlock and the process terminates, the expiry would ensure that the lock is removed after a certain time. This would allow other processes (or threads) to acquire the lock even if the earlier mutex fails to unlock it. `expiry` should always be greater than the `block_time` else a `ValueError` is raised. Default value is `7`.

```
import redis
from redismutex import RedisMutex

conn = redis.StrictRedis(host='localhost', port=6379, db=1)
mutex = RedisMutex(conn, blocking=True, block_time=10, delay=0.5, expiry=12)
```

For the blocking mutex mentioned above, if the lock is already acquired, it will poll every **0.5 seconds** (delay) to check if the lock is released. This process will continue for **10 seconds** (block_time), i.e., it can make upto 50 poll requests at maximum.

If the mutex is not able to acquire a lock within 10 seconds, it will raise a `BlockTimeExceedError`.

If the mutex acquires the lock and fails to release it, the lock will automatically be removed in **12 seconds** (expiry, timed from the creation of the lock).

CHAPTER 3

Using as decorator

You can use the `with_redismutex` decorator to wrap a function inside a mutex lock. It works in the same manner as the mutex object.

```python
import redis
from redismutex.decorators import with_redismutex

conn = redis.StrictRedis(host='localhost', port=6379, db=1)
mutex_key = 'YOUR-MUTEX-KEY'

@with_redismutex(conn, mutex_key)
def foobar():
    # some resource
    # critical task...
    return 'foobar'
```

NOTE: When using the `with_redismutex` decorator, it is NOT possible to access the `mutex.key` and `mutex.value` inside the function.

3.1 Generating dynamic mutex key in a decorator

As it is evident from the above example that the mutex key provided to the decorator will be static. Although, in most of the cases, you'd have to build the keys based on arguments passed to the function. For example, in a django view, you might want to generate a key based on `request.user.id`.

This can be achieved by overriding the `generate_key` method in the `with_redismutex` decorator. The parameters passed to the `generate_key` function will exactly be the same the those passed to the enclosing function.

```python
from redismutex.decorators import with_redismutex

class view_with_mutex(with_redismutex):
    """Decorator to add mutex to a django view
    """
```

(continues on next page)

(continued from previous page)

```
    def generate_key(self, *args, **kwargs):
        request = args[0]
        return str(request.user.id)
```

Now you only need to pass the redis connection object in the decorator, no need for the *key*. You can use the `view_with_mutex` something like..

```
@view_with_mutex(conn)
def user_settings(request, **kwargs):
    # This would set automatically set the mutex key
    # as request.user.id
    return HttpResponse()
```