jojohot304 / **tornado-redis**
forked from leporo/tornado-redis

Asynchronous Redis client that works within Tornado IO loop.

Edit

Manage topics

336 commits **1** branch **0** releases **30** contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

This branch is even with leporo:master. Pull request Compare

leporo Add the "project is not maintained" note to readme Latest commit `350fc8b` on 12 Jul 2016

demos	Python3 compatibility	4 years ago
tornadoredis	Again fix travis.yml and args positions	3 years ago
.gitignore	Added helper classes to handle multiple redis channel subscriptions u...	5 years ago
.travis.yml	Again fix travis.yml and args positions	3 years ago
README.md	Add the "project is not maintained" note to readme	2 years ago
requirements.txt	Added helper classes to handle multiple redis channel subscriptions u...	5 years ago
runtests.py	Added helper classes to handle multiple redis channel subscriptions u...	5 years ago
setup.py	2.5.0	3 years ago
tox.ini	issue#60 Bumped the version no	5 years ago

README.md

Tornado-Redis

build passing

THIS PROJECT IS NOT MAINTAINED ANYMORE.

HERE ARE SOME ALTERNATIVES:

- asyncio - based client (Python 3.3+)
- toredis

Asynchronous Redis client for the Tornado Web Server.

This is a fork of brükva redis client modified to be used via Tornado's native 'tornado.gen' interface instead of 'adisp' call dispatcher.

Tornado-Redis is licensed under the Apache Licence, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0.html).

Tornado-Redis vs Redis-py

I recommend using both tornado-redis and redis-py clients for your Tornado Web Application. Use tornado-redis to subscribe to Pub/Sub notifications and for blocking commands such as BLPOP, BRPOP, BRPOPLPUSH. You may safely use redis-py for most of other cases.

I suggest NOT to use connection pools with redis-py client. In most cases you may use a single 'global' instance of Redis object wherever you'll need it.

Note, that Tornado-redis is far less efficient than redis-py client in handling MGET/MSET requests and working with Pipelines. I suggest using the redis-py client and hiredis transport to get maximum performance on these commands.

Please check my answer on StackOverflow on querying Redis server from Tornado application for some additional details.

Installation

You may install the tornado-redis client library using pip or easy_install tools:

```
pip install tornado-redis
```

or

```
easy_install install tornado-redis
```

To build and install the tornado-redis client library from source, clone the git://github.com/leporo/tornado-redis.git repository or download the archive from the download page and extract it into the separate directory. Then execute the following commands in the source directory:

```
python setup.py build
python setup.py install
```

Usage

```
import tornadoredis
import tornado.web
import tornado.gen

...

c = tornadoredis.Client()
c.connect()

...

class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        foo = yield tornado.gen.Task(c.get, 'foo')
        bar = yield tornado.gen.Task(c.get, 'bar')
        zar = yield tornado.gen.Task(c.get, 'zar')
        self.set_header('Content-Type', 'text/html')
        self.render("template.html", title="Simple demo", foo=foo, bar=bar, zar=zar)
```

Pub/Sub

Tornado-redis comes with helper classes to simplify Pub/Sub implementation. These helper classes introduced in tornadoredis.pubsub module use a single redis server connection to handle multiple client and channel subscriptions.

Here is a sample SockJSConnection handler code:

```
# Use the synchronous redis client to publish messages to a channel
redis_client = redis.Redis()
# Create the tornadoredis.Client instance
# and use it for redis channel subscriptions
subscriber = tornadoredis.pubsub.SockJSSubscriber(tornadoredis.Client())

class SubscriptionHandler(sockjs.tornado.SockJSConnection):
    """
    SockJS connection handler.

    Note that there are no "on message" handlers - SockJSSubscriber class
    calls SockJSConnection.broadcast method to transfer messages
    to subscribed clients.
    """
    def __init__(self, *args, **kwargs):
        super(MessageHandler, self).__init__(*args, **kwargs)
        subscriber.subscribe('test_channel', self)

    def on_close(self):
        subscriber.unsubscribe('test_channel', self)
```

See the sockjs application in the demo folder and tornadoredis.pubsub module for more implementation details.

Using Pipelines

Pipelines correspond to the Redis transaction feature.

Here is a simple example of pipeline feature usage:

```
client = Client()
# Create a 'Pipeline' to pack a bunldle of Redis commands
# and send them to a Redis server in a single request
pipe = client.pipeline()
# Add commands to a bundle
pipe.hset('foo', 'bar', 1)
pipe.expire('foo', 60)
# Send them to the Redis server and retrieve execution results
res_hset, res_expire = yield gen.Task(pipe.execute)
```

Note that nothing is being sent to the Redis server until the `pipe.execute` method call so there is no need to wrap a `pipe.hset` and `pipe.expire` calls with the `yield gen.Task(...)` statement.

Using Locks

A Lock is a synchronization mechanism for the enforcing of limits on access to a resource in an environment where there are many threads of execution. In this case, the Lock uses the Redis server as state store, thus it can be used by multiple processes/computers to allow distributed coordination.

Here is a simple example of how to use a lock:

```
client = Client()

# Create a 'Lock' object, with a maximum lock time of 10 seconds, and a polling interval
# of 100ms
my_lock = client.lock("testLock", lock_ttl=10, polling_interval=0.1)

# Try and acquire the lock, blocking until it is acquired or an error has occured.
# This does not block the IOLoop in any way.
result = yield gen.Task(my_lock.acquire, blocking=True)

# Create another, identical lock and try and acquire it
his_lock = client.lock("testLock", lock_ttl=10, polling_interval=0.1)
# We will fail, as the lock is already acquired by my_lock. Result will be False
result = yield gen.Task(his_lock.acquire, blocking=False)

# Schedule my_lock to be released in 5 seconds
client._io_loop.add_timeout(client._io_loop.time() + 5, my_lock.release)

# Meanwhile, attempt to get the lock again. This time block until we succeed
# In 5 seconds, my_lock will be released and his_lock will be acquired instead
result = yield gen.Task(his_lock.acquire, blocking=True)

# Release the lock. Even if we don't, it will be timed out in 10 seconds
# (because of the lock_ttl setting)
yield gen.Task(his_lock.release)
```

In this case we chose to run both locks in the same process - generally, we prefer to use Redis locks only if we need to synchronize several processes on different machines.

Connection Pool Support

To limit a number of redis server connections opened by an application and reuse them the tornado-redis library has the connection pooling support. To activate it, create the ConnectionPool object instance and pass it as connection_pool argument to the Client object:

```
CONNECTION_POOL = tornadoredis.ConnectionPool(max_connections=500,
                                              wait_for_available=True)
# ...
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        c = tornadoredis.Client(connection_pool=CONNECTION_POOL)
        info = yield tornado.gen.Task(c.info)
        ....
        # Release the connection to be reused by connection pool.
        yield tornado.gen.Task(c.disconnect)
        self.render(....)
```

Note that you have to add a `disconnect` method call at the end of the code block using the Client instance to release the pooled connection (it's to be fixed it future library releases).

See the connection pool demo for an example of the 'connection pool' feature usage.

Note that connection pooling feature has multiple drawbacks and may affect your application performance. See the "Tornado-Redis vs Redis-py" note for more details.

Please consider using Pub/Sub helper classes implemented in tornadoredis.pubsub module or using a similar approach instead of connection polling for Pub/Sub operations.

Demos

Check the Demos folder for tornado-redis usage examples.

Here is the list of demo applications available from this repository:

simple - a very basic example of tornado-redis client usage

connection_pool - a 'connection pool' feature demo

websockets - a demo web chat application using WebSockets and Redis' PubSub feature.

sockjs - the same-looking demo web chat application but using the SockJS transport to support and using the SockJSSubscriber helper class to share a single redis server connection between subscribed clients.

Running Tests

The redis server must be started on the default (:6379) port.

Use the following command to run the test suite:

```
python -m tornado.testing tornadoredis.tests
```

Or use the tox to test how it works in different environments:

```
tox
```

or

```
tox -e pypy
```

Credits and Contributors

The brükva project has been started by Konstantin Merenkov but seem to be not maintained any more.

tornado-redis was inspired and based on the work of Andy McCurdy and redis-py contributors.

evilkost

Matt Dawson

maeldur

Olivier Yiptong

Juarez Bochi

Jakub Roztocil

nizox

Lessandro Mariano

Kwok-kuen Cheung

Ofir Herzas

Alon Diamant

Jonas Hagstedt

The Tornado-Redis project's source code and 'tornado-redis' PyPI package are maintained by Vlad Glushchuk.

Tornado is an open source version of the scalable, non-blocking web server and tools that power FriendFeed. Documentation and downloads are available at http://www.tornadoweb.org/