aredis Documentation

Release 1.0.7

NoneGG

Mar 01, 2020

Contents

An efficient and user-friendly async redis client ported from redis-py (which is a Python interface to the Redis key-value). And the cluster part is ported from redis-py-cluster aredis is the async version of these to redis clients, with effort to enable you using redis with asyncio more easily.

The source code is available on github.

CHAPTER 1

Installation

aredis requires a running Redis server.

To install aredis, simply:

```
$ pip3 install aredis
```

or alternatively (you really should be using pip though):

```
$ easy_install aredis
```

or from source:

```
$ python setup.py install
```

CHAPTER 2

Getting started

For more example

2.1 single node client

```
>>> import asyncio
>>> from aredis import StrictRedis
>>>
>>> async def example():
>>>     client = StrictRedis(host='127.0.0.1', port=6379, db=0)
>>>     await client.flushdb()
>>>     await client.set('foo', 1)
>>>     assert await client.exists('foo') is True
>>>     await client.incr('foo', 100)
>>>
>>>     assert int(await client.get('foo')) == 101
>>>     await client.expire('foo', 1)
>>>     await asyncio.sleep(0.1)
>>>     await client.ttl('foo')
>>>     await asyncio.sleep(1)
>>>     assert not await client.exists('foo')
>>>
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(example())
```

2.2 cluster client

```
>>> import asyncio
>>> from aredis import StrictRedisCluster
>>>
```

(continues on next page)

(continued from previous page)

```
>>> async def example():
>>>     client = StrictRedisCluster(host='172.17.0.2', port=7001)
>>>     await client.flushdb()
>>>     await client.set('foo', 1)
>>>     await client.lpush('a', 1)
>>>     print(await client.cluster_slots())
>>>
>>>     await client.rpoplpush('a', 'b')
>>>     assert await client.rpop('b') == b'1'
>>>
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(example())
{(10923, 16383): [{'host': b'172.17.0.2', 'node_id': b
↪'332f41962b33fa44bbc5e88f205e71276a9d64f4', 'server_type': 'master', 'port': 7002},
{'host': b'172.17.0.2', 'node_id': b'c02deb8726cdd412d956f0b9464a88812ef34f03',
↪'server_type': 'slave', 'port': 7005}],
(5461, 10922): [{'host': b'172.17.0.2', 'node_id': b
↪'3d1b020fc46bf7cb2ffc36e10e7d7befca7c5533', 'server_type': 'master', 'port': 7001},
{'host': b'172.17.0.2', 'node_id': b'aac4799b65ff35d8dd2ad152a5515d15c0dc8ab7',
↪'server_type': 'slave', 'port': 7004}],
(0, 5460): [{'host': b'172.17.0.2', 'node_id': b
↪'0932215036dc0d908cf662fdfca4d3614f221b01', 'server_type': 'master', 'port': 7000},
{'host': b'172.17.0.2', 'node_id': b'f6603ab4cb77e672de23a6361ec165f3a1a2bb42',
↪'server_type': 'slave', 'port': 7003}]}
```

CHAPTER 3

Dependencies & supported python versions

hiredis and uvloop can make aredis faster, but it is up to you whether to install them or not.

- Optional Python: hiredis >= *0.2.0*. Older versions might work but is not tested.
- Optional event loop policy: uvloop >= *0.8.0*. Older versions might work but is not tested.
- A working Redis cluster based on version >= *3.0.0* is required. Only *3.0.x* releases is supported.

CHAPTER 4

Supported python versions

- 3.5
- 3.6

Experimental:

- 3.7-dev

Note: Python < 3.5

I tried to change my code to make aredis compatible for Python under 3.5, but it failed because of some api of asyncio. Since asyncio is stabilize from Python 3.5, i think it may be better to use the new release of asyncio.

Note: pypy

For now, uvloop is not supported by pypy, and you can only use it with cpython & hiredis to accelerate your code. pypy 3.5-v5.8.0 is tesed and with it code can run twice faster than before.

CHAPTER 5

API reference

Most API are described in **‘redis command reference<https://redis.io/commands>‘_** what makes difference and those should be noticed are referred in doc specially. You can post a new issue / read redis command reference / read annotation of API (mainly about how to use them) if you have any problem about the API. Related issue are welcome.

CHAPTER 6

The Usage Guide

6.1 API Reference

The connection part is rewritten to make client async, and most API is ported from redis-py. So most API and usage are the same as redis-py. If you use redis-py in your code, just use *async/await* syntax with your code. for more examples

The official Redis command documentation does a great job of explaining each command in detail. aredis only shift StrictRedis class from redis-py that implement these commands. The StrictRedis class attempts to adhere to the official command syntax. There are a few exceptions:

- **SELECT**: Not implemented. See the explanation in the Thread Safety section below.
- **DEL**: 'del' is a reserved keyword in the Python syntax. Therefore aredis uses 'delete' instead.
- **CONFIG GET|SET**: These are implemented separately as config_get or config_set.
- **MULTI/EXEC**: These are implemented as part of the Pipeline class. The pipeline is wrapped with the MULTI and EXEC statements by default when it is executed, which can be disabled by specifying transaction=False. See more about Pipelines below.
- **SUBSCRIBE/LISTEN**: Similar to pipelines, PubSub is implemented as a separate class as it places the underlying connection in a state where it can't execute non-pubsub commands. Calling the pubsub method from the Redis client will return a PubSub instance where you can subscribe to channels and listen for messages. You can only call PUBLISH from the Redis client.
- **SCAN/SSCAN/HSCAN/ZSCAN**: The *SCAN commands are implemented as they exist in the Redis documentation. In addition, each command has an equivilant iterator method. These are purely for convenience so the user doesn't have to keep track of the cursor while iterating. (Use Python 3.6 and the scan_iter/sscan_iter/hscan_iter/zscan_iter methods for this behavior. **iter functions are not supported in Python 3.5**)

6.1.1 Loop

The event loop can be set with the loop keyworkd argugment. If no loop is given the default event loop will be

warning

asyncio.AbstractEventLoop is actually not thread safe and asyncio uses **BaseDefaultEventLoopPolicy** as default event policy(which create new event loop instead of sharing event loop between threads, being thread safe to some degree) So the StricRedis is still thread safe if your code works with default event loop. But if you customize event loop yourself, please make sure your event loop is thread safe(maybe you should customize on the base of **BaseDefaultEventLoopPolicy** instead of **AbstractEventLoop**)

Detailed discussion about the problem is in issue20

```
>>> import aredis
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> r = aredis.StrictRedis(host='localhost', port=6379, db=0, loop=loop)
```

6.1.2 Decoding

Param **encoding** and **decode_responses** are now used to support response encoding.

encoding is used for specifying with which encoding you want responses to be decoded. **decode_responses** is used for tell the client whether responses should be decoded.

If decode_responses is set to True and no encoding is specified, client will use 'utf-8' by default.

6.1.3 Connections

ConnectionPools manage a set of Connection instances. aredis ships with two types of Connections. The default, Connection, is a normal TCP socket based connection. The UnixDomainSocketConnection allows for clients running on the same device as the server to connect via a unix domain socket. To use a UnixDomainSocketConnection connection, simply pass the unix_socket_path argument, which is a string to the unix domain socket file. Additionally, make sure the unixsocket parameter is defined in your redis.conf file. It's commented out by default.

```
>>> r = redis.StrictRedis(unix_socket_path='/tmp/redis.sock')
```

You can create your own Connection subclasses as well. This may be useful if you want to control the socket behavior within an async framework. To instantiate a client class using your own connection, you need to create a connection pool, passing your class to the connection_class argument. Other keyword parameters you pass to the pool will be passed to the class specified during initialization.

```
>>> pool = redis.ConnectionPool(connection_class=YourConnectionClass,
                                your_arg='...', ...)
```

6.1.4 Parsers

Parser classes provide a way to control how responses from the Redis server are parsed. aredis ships with two parser classes, the PythonParser and the HiredisParser. By default, aredis will attempt to use the HiredisParser if you have the hiredis module installed and will fallback to the PythonParser otherwise.

Hiredis is a C library maintained by the core Redis team. Pieter Noordhuis was kind enough to create Python bindings. Using Hiredis can provide up to a 10x speed improvement in parsing responses from the Redis server. The performance increase is most noticeable when retrieving many pieces of data, such as from LRANGE or SMEMBERS operations.

Hiredis is available on PyPI, and can be installed via pip or easy_install just like aredis.

```
$ pip install hiredis
```

or

```
$ easy_install hiredis
```

6.1.5 Response Callbacks

The client class uses a set of callbacks to cast Redis responses to the appropriate Python type. There are a number of these callbacks defined on the Redis client class in a dictionary called RESPONSE_CALLBACKS.

Custom callbacks can be added on a per-instance basis using the set_response_callback method. This method accepts two arguments: a command name and the callback. Callbacks added in this manner are only valid on the instance the callback is added to. If you want to define or override a callback globally, you should make a subclass of the Redis client and add your callback to its REDIS_CALLBACKS class dictionary.

Response callbacks take at least one parameter: the response from the Redis server. Keyword arguments may also be accepted in order to further control how to interpret the response. These keyword arguments are specified during the command's call to execute_command. The ZRANGE implementation demonstrates the use of response callback keyword arguments with its "withscores" argument.

6.1.6 Thread Safety

Redis client instances can safely be shared between threads. Internally, connection instances are only retrieved from the connection pool during command execution, and returned to the pool directly after. Command execution never modifies state on the client instance.

However, there is one caveat: the Redis SELECT command. The SELECT command allows you to switch the database currently in use by the connection. That database remains selected until another is selected or until the connection is closed. This creates an issue in that connections could be returned to the pool that are connected to a different database.

As a result, aredis does not implement the SELECT command on client instances. If you use multiple Redis databases within the same application, you should create a separate client instance (and possibly a separate connection pool) for each database.

It is not safe to pass PubSub or Pipeline objects between threads.

6.2 Benchmark

benchmark/comparation.py run on virtual machine(ubuntu, 4G memory and 2 cpu) with hiredis as parser

6.2.1 local redis server

num of query/time	aredis(asyncio)	aredis(uvloop)	aioredis(asyncio)	aioredis(uvloop)	asyncio_redis(asyncio)	asyncio_redis(uvloop)	redis-py
100	0.0190	0.01802	0.0400	0.01989	0.0391	0.0326	0.0111
1000	0.0917	0.05998	0.1237	0.05866	0.1838	0.1397	0.0396
10000	1.0614	0.66423	1.2277	0.62957	1.9061	1.5464	0.3944
100000	10.228	6.13821	10.400	6.06872	19.982	15.252	3.6307

6.2.2 redis server in local area network

Only run with uvloop, or it will be too slow. Although it seems like that running code in synchronous way perform more well than in asynchronous way, the point is that it won't block the other code to run.

num of query/time	aredis(uvloop)	aioredis(uvloop)	asyncio_redis(uvloop)	redis-py
100	0.06998	0.06019	0.1971	0.0556
1000	0.66197	0.61183	1.9330	0.7909
10000	5.81604	6.87364	19.186	7.1334
100000	58.4715	60.9220	189.06	58.979

test result may change according to your computer performance and network (you may run the sheet yourself to determine which one is the most suitable for you)

6.2.3 Advantage

1. aredis can be used howerver you install hiredis or not.
2. aredis' API are mostly ported from redis-py, which is easy to use indeed and make it easy to port your code with asyncio
3. according to my test, aredis is efficient enough (please run benchmarks/comparation.py to see which async redis client is suitable for you)
4. aredis can be run both with asyncio and uvloop, the latter can double the speed of your async code.

6.3 Publish / Subscribe

aredis includes a *PubSub* object that subscribes to channels and listens for new messages. Creating a *PubSub* object is easy.

```
>>> r = redis.StrictRedis(...)
>>> p = r.pubsub()
```

Once a *PubSub* instance is created, channels and patterns can be subscribed to.

```
>>> await p.subscribe('my-first-channel', 'my-second-channel', ...)
>>> await p.psubscribe('my-*', ...)
```

The *PubSub* instance is now subscribed to those channels/patterns. The subscription confirmations can be seen by reading messages from the *PubSub* instance.

```
>>> await p.get_message()
{'pattern': None, 'type': 'subscribe', 'channel': 'my-second-channel', 'data': 1L}
>>> await p.get_message()
{'pattern': None, 'type': 'subscribe', 'channel': 'my-first-channel', 'data': 2L}
>>> await p.get_message()
{'pattern': None, 'type': 'psubscribe', 'channel': 'my-*', 'data': 3L}
```

Every message read from a *PubSub* instance will be a dictionary with the following keys.

- **type**: One of the following: 'subscribe', 'unsubscribe', 'psubscribe', 'punsubscribe', 'message', 'pmessage'
- **channel**: The channel [un]subscribed to or the channel a message was published to

- **pattern**: The pattern that matched a published message's channel. Will be *None* in all cases except for 'pmessage' types.
- **data**: The message data. With [un]subscribe messages, this value will be the number of channels and patterns the connection is currently subscribed to. With [p]message messages, this value will be the actual published message.

Let's send a message now.

```
# the publish method returns the number matching channel and pattern
# subscriptions. 'my-first-channel' matches both the 'my-first-channel'
# subscription and the 'my-*' pattern subscription, so this message will
# be delivered to 2 channels/patterns
>>> await r.publish('my-first-channel', 'some data')
2
>>> await p.get_message()
{'channel': 'my-first-channel', 'data': 'some data', 'pattern': None, 'type': 'message
↪'}
>>> await p.get_message()
{'channel': 'my-first-channel', 'data': 'some data', 'pattern': 'my-*', 'type':
↪'pmessage'}
```

Unsubscribing works just like subscribing. If no arguments are passed to [p]unsubscribe, all channels or patterns will be unsubscribed from.

```
>>> await p.unsubscribe()
>>> await p.punsubscribe('my-*')
>>> await p.get_message()
{'channel': 'my-second-channel', 'data': 2L, 'pattern': None, 'type': 'unsubscribe'}
>>> await p.get_message()
{'channel': 'my-first-channel', 'data': 1L, 'pattern': None, 'type': 'unsubscribe'}
>>> await p.get_message()
{'channel': 'my-*', 'data': 0L, 'pattern': None, 'type': 'punsubscribe'}
```

aredis also allows you to register callback functions to handle published messages. Message handlers take a single argument, the message, which is a dictionary just like the examples above. To subscribe to a channel or pattern with a message handler, pass the channel or pattern name as a keyword argument with its value being the callback function.

When a message is read on a channel or pattern with a message handler, the message dictionary is created and passed to the message handler. In this case, a *None* value is returned from get_message() since the message was already handled.

```
>>> def my_handler(message):
...     print('MY HANDLER: ', message['data'])
>>> await p.subscribe(**{'my-channel': my_handler})
# read the subscribe confirmation message
>>> await p.get_message()
{'pattern': None, 'type': 'subscribe', 'channel': 'my-channel', 'data': 1L}
>>> await r.publish('my-channel', 'awesome data')
1
# for the message handler to work, we need tell the instance to read data.
# this can be done in several ways (read more below). we'll just use
# the familiar get_message() function for now
>>> await message = p.get_message()
MY HANDLER:  awesome data
# note here that the my_handler callback printed the string above.
# `message` is None because the message was handled by our handler.
>>> print(message)
None
```

If your application is not interested in the (sometimes noisy) subscribe/unsubscribe confirmation messages, you can ignore them by passing *ignore_subscribe_messages=True* to *r.pubsub()*. This will cause all subscribe/unsubscribe messages to be read, but they won't bubble up to your application.

```
>>> p = r.pubsub(ignore_subscribe_messages=True)
>>> await p.subscribe('my-channel')
>>> await p.get_message()  # hides the subscribe message and returns None
>>> await r.publish('my-channel')
1
>>> await p.get_message()
{'channel': 'my-channel', 'data': 'my data', 'pattern': None, 'type': 'message'}
```

There are three different strategies for reading messages.

The examples above have been using *pubsub.get_message()*. If there's data available to be read, *get_message()* will read it, format the message and return it or pass it to a message handler. If there's no data to be read, *get_message()* will return None after the configured *timeout* (*timeout* should set to value larger than 0 or it will be ignore). This makes it trivial to integrate into an existing event loop inside your application.

```
>>> while True:
>>>     message = await p.get_message()
>>>     if message:
>>>         # do something with the message
>>>     await asyncio.sleep(0.001)  # be nice to the system :)
```

Older versions of aredis only read messages with *pubsub.listen()*. listen() is a generator that blocks until a message is available. If your application doesn't need to do anything else but receive and act on messages received from redis, listen() is an easy way to get up an running.

```
>>> for message in await p.listen():
...     # do something with the message
```

The third option runs an event loop in a separate thread. *pubsub.run_in_thread()* creates a new thread and use the event loop in main thread. The thread object is returned to the caller of *run_in_thread()*. The caller can use the *thread.stop()* method to shut down the event loop and thread. Behind the scenes, this is simply a wrapper around *get_message()* that runs in a separate thread, and use *asyncio.run_coroutine_threadsafe()* to run coroutines.

Note: Since we're running in a separate thread, there's no way to handle messages that aren't automatically handled with registered message handlers. Therefore, aredis prevents you from calling *run_in_thread()* if you're subscribed to patterns or channels that don't have message handlers attached.

```
>>> await p.subscribe(**{'my-channel': my_handler})
>>> thread = p.run_in_thread(sleep_time=0.001)
# the event loop is now running in the background processing messages
# when it's time to shut it down...
>>> thread.stop()
```

PubSub objects remember what channels and patterns they are subscribed to. In the event of a disconnection such as a network error or timeout, the PubSub object will re-subscribe to all prior channels and patterns when reconnecting. Messages that were published while the client was disconnected cannot be delivered. When you're finished with a PubSub object, call its *.close()* method to shutdown the connection.

```
>>> p = r.pubsub()
>>> ...
>>> p.close()
```

The PUBSUB set of subcommands CHANNELS, NUMSUB and NUMPAT are also supported:

```
>>> await r.pubsub_channels()
['foo', 'bar']
>>> await r.pubsub_numsub('foo', 'bar')
[('foo', 9001), ('bar', 42)]
>>> await r.pubsub_numsub('baz')
[('baz', 0)]
>>> await r.pubsub_numpat()
1204
```

6.4 Sentinel support

aredis can be used together with Redis Sentinel to discover Redis nodes. You need to have at least one Sentinel daemon running in order to use aredis's Sentinel support.

Connecting aredis to the Sentinel instance(s) is easy. You can use a Sentinel connection to discover the master and slaves network addresses:

```
>>> from redis.sentinel import Sentinel
>>> sentinel = Sentinel([('localhost', 26379)], stream_timeout=0.1)
>>> await sentinel.discover_master('mymaster')
('127.0.0.1', 6379)
>>> await sentinel.discover_slaves('mymaster')
[('127.0.0.1', 6380)]
```

You can also create Redis client connections from a Sentinel instance. You can connect to either the master (for write operations) or a slave (for read-only operations).

```
>>> master = sentinel.master_for('mymaster', stream_timeout=0.1)
>>> slave = sentinel.slave_for('mymaster', stream_timeout=0.1)
>>> master.set('foo', 'bar')
>>> slave.get('foo')
'bar'
```

The master and slave objects are normal StrictRedis instances with their connection pool bound to the Sentinel instance. When a Sentinel backed client attempts to establish a connection, it first queries the Sentinel servers to determine an appropriate host to connect to. If no server is found, a MasterNotFoundError or SlaveNotFoundError is raised. Both exceptions are subclasses of ConnectionError.

When trying to connect to a slave client, the Sentinel connection pool will iterate over the list of slaves until it finds one that can be connected to. If no slaves can be connected to, a connection will be established with the master.

See Guidelines for Redis clients with support for Redis Sentinel to learn more about Redis Sentinel.

6.5 LUA Scripting

aredis supports the EVAL, EVALSHA, and SCRIPT commands. However, there are a number of edge cases that make these commands tedious to use in real world scenarios. Therefore, aredis exposes a Script object that makes scripting much easier to use.

To create a Script instance, use the *register_script* function on a client instance passing the LUA code as the first argument. *register_script* returns a Script instance that you can use throughout your code.

The following trivial LUA script accepts two parameters: the name of a key and a multiplier value. The script fetches the value stored in the key, multiplies it with the multiplier value and returns the result.

```
>>> r = redis.StrictRedis()
>>> lua = """
... local value = redis.call('GET', KEYS[1])
... value = tonumber(value)
... return value * ARGV[1]"""
>>> multiply = r.register_script(lua)
```

multiply is now a Script instance that is invoked by calling it like a function. Script instances accept the following optional arguments:

- **keys**: A list of key names that the script will access. This becomes the KEYS list in LUA.
- **args**: A list of argument values. This becomes the ARGV list in LUA.
- **client**: A aredis Client or Pipeline instance that will invoke the script. If client isn't specified, the client that intiially created the Script instance (the one that *register_script* was invoked from) will be used.

Notice that the *Srcipt.__call__* is no longer useful(*async/await* can't be used in magic method), please use *Script.register* instead

Continuing the example from above:

```
>>> await r.set('foo', 2)
>>> await multiply.execute(keys=['foo'], args=[5])
10
```

The value of key 'foo' is set to 2. When multiply is invoked, the 'foo' key is passed to the script along with the multiplier value of 5. LUA executes the script and returns the result, 10.

Script instances can be executed using a different client instance, even one that points to a completely different Redis server.

```
>>> r2 = redis.StrictRedis('redis2.example.com')
>>> await r2.set('foo', 3)
>>> multiply.execute(keys=['foo'], args=[5], client=r2)
15
```

The Script object ensures that the LUA script is loaded into Redis's script cache. In the event of a NOSCRIPT error, it will load the script and retry executing it.

Script objects can also be used in pipelines. The pipeline instance should be passed as the client argument when calling the script. Care is taken to ensure that the script is registered in Redis's script cache just prior to pipeline execution.

```
>>> pipe = await r.pipeline()
>>> await pipe.set('foo', 5)
>>> await multiply(keys=['foo'], args=[5], client=pipe)
>>> await pipe.execute()
[True, 25]
```

6.6 Pipelines

Pipelines are a subclass of the base Redis class that provide support for buffering multiple commands to the server in a single request. They can be used to dramatically increase the performance of groups of commands by reducing the number of back-and-forth TCP packets between the client and server.

Pipelines are quite simple to use:

```
>>> async def example(client):
>>>     async with await client.pipeline(transaction=True) as pipe:
>>>     # will return self to send another command
>>>     pipe = await (await pipe.flushdb()).set('foo', 'bar')
>>>     # can also directly send command
>>>     await pipe.set('bar', 'foo')
>>>     # commands will be buffered
>>>     await pipe.keys('*')
>>>     res = await pipe.execute()
>>>     # results should be in order corresponding to your command
>>>     assert res == [True, True, True, [b'bar', b'foo']]
```

For ease of use, all commands being buffered into the pipeline return the pipeline object itself. Which enable you to use it like the example provided.

In addition, pipelines can also ensure the buffered commands are executed atomically as a group. This happens by default. If you want to disable the atomic nature of a pipeline but still want to buffer commands, you can turn off transactions.

```
>>> pipe = r.pipeline(transaction=False)
```

A common issue occurs when requiring atomic transactions but needing to retrieve values in Redis prior for use within the transaction. For instance, let's assume that the INCR command didn't exist and we need to build an atomic version of INCR in Python.

The completely naive implementation could GET the value, increment it in Python, and SET the new value back. However, this is not atomic because multiple clients could be doing this at the same time, each getting the same value from GET.

Enter the WATCH command. WATCH provides the ability to monitor one or more keys prior to starting a transaction. If any of those keys change prior the execution of that transaction, the entire transaction will be canceled and a WatchError will be raised. To implement our own client-side INCR command, we could do something like this:

```
>>> async def example():
>>>     async with await r.pipeline() as pipe:
...         while 1:
...             try:
...                 # put a WATCH on the key that holds our sequence value
...                 await pipe.watch('OUR-SEQUENCE-KEY')
...                 # after WATCHing, the pipeline is put into immediate execution
...                 # mode until we tell it to start buffering commands again.
...                 # this allows us to get the current value of our sequence
...                 current_value = await pipe.get('OUR-SEQUENCE-KEY')
...                 next_value = int(current_value) + 1
...                 # now we can put the pipeline back into buffered mode with MULTI
...                 pipe.multi()
...                 pipe.set('OUR-SEQUENCE-KEY', next_value)
...                 # and finally, execute the pipeline (the set command)
...                 await pipe.execute()
...                 # if a WatchError wasn't raised during execution, everything
...                 # we just did happened atomically.
...                 break
...             except WatchError:
...                 # another client must have changed 'OUR-SEQUENCE-KEY' between
...                 # the time we started WATCHing it and the pipeline's execution.
...                 # our best bet is to just retry.
...                 continue
```

Note that, because the Pipeline must bind to a single connection for the duration of a WATCH, care must be taken to ensure that the connection is returned to the connection pool by calling the reset() method. If the Pipeline is used as a context manager (as in the example above) reset() will be called automatically. Of course you can do this the manual way by explicitly calling reset():

```
>>> async def example():
>>>     async with await r.pipeline() as pipe:
>>>         while 1:
...             try:
...                 await pipe.watch('OUR-SEQUENCE-KEY')
...                 ...
...                 await pipe.execute()
...                 break
...             except WatchError:
...                 continue
...             finally:
...                 await pipe.reset()
```

A convenience method named "transaction" exists for handling all the boilerplate of handling and retrying watch errors. It takes a callable that should expect a single parameter, a pipeline object, and any number of keys to be WATCHed. Our client-side INCR command above can be written like this, which is much easier to read:

```
>>> async def client_side_incr(pipe):
...     current_value = await pipe.get('OUR-SEQUENCE-KEY')
...     next_value = int(current_value) + 1
...     pipe.multi()
...     await pipe.set('OUR-SEQUENCE-KEY', next_value)
>>>
>>> await r.transaction(client_side_incr, 'OUR-SEQUENCE-KEY')
[True]
```

6.7 Streams

Stream is a new feature provided by redis.

Since not all commands related are released officially(some commands are only referred in stream introduction), **you should make sure you know about it before using the api, and the API may be changed in the future.**

For now, according to command manual , only *XADD*, *XRANGE*, *XREVRANGE*, *XLEN*, *XREAD*, *XREADGROUP*, *XPENDING* commands are released. But commands you can find in stream introduction are all supported in aredis, you can try the new feature with it.

You can append entries to stream like code below:

```
>>> entry = dict(event=1, user='usr1')
>>> async def append_msg_to_stream(client, entry):
>>>     stream_id = await client.xadd('example_stream', entry, max_len=10)
>>>     return stream_id
```

notice - max length of the stream length will not be limited max_len is set to None - max_len should be int greater than 0, if set to 0 or negative, the stream length will not be limited - The *XADD* command will auto-generate a unique id for you if the id argument specified is the '*' character.

You can use use read entries from a stream using *XRANGE* & *XREVRANGE*

```
>>> async def fetch_entries(client, stream, count=10, reverse=False):
>>>     # if you do know the range of stream_id, you can specify it when using xrange
>>>     if reverse:
>>>         entries = await client.xrevrange(stream, start='10-0', end='1-0',␣
↪count=count)
>>>     else:
>>>         entries = await client.xrange(stream, start='1-0', end='10-0',␣
↪count=count)
>>>     return entries
```

Actually, stream feature is inspired by kafka, a stream can be consumed by *consumer* from a *group*, like code below:

```
>>> async def consuming_process(client):
>>>     # create a stream firstly
>>>     for idx in range(20):
>>>         # give progressive stream id when create entry
>>>         await client.xadd('test_stream', {'k1': 'v1', 'k2': 1}, stream_id=idx)
>>>     # now create a consumer group
>>>     # stream_id can be specified when creating a group,
>>>     # if given '0', group will consume the stream from the beginning
>>>     # if give '$', group will only consume newly appended entries
>>>     await r.xgroup_create('test_stream', 'test_group', '0')
>>>     # now consume the entries by 'consumer1' from group 'test_group'
>>>     entries = await r.xreadgroup('test_group', 'consumer1', count=5, test_stream=
↪'1')
```

6.8 Extra

6.8.1 Lock

For now Lock only support for single redis node, please don't use it in cluster env.

There are two kinds of *Lock class* available for now, you can also make your own for special requirements.

lock(*self*, *name*, *timeout=None*, *sleep=0.1*, *blocking_timeout=None*, *lock_class=None*, *thread_local=True*)

Return a new Lock object using key `name` that mimics the behavior of threading.Lock.

If specified, `timeout` indicates a maximum life for the lock. By default, it will remain locked until release() is called.

`sleep` indicates the amount of time to sleep per loop iteration when the lock is in blocking mode and another client is currently holding the lock.

`blocking_timeout` indicates the maximum amount of time in seconds to spend trying to acquire the lock. A value of `None` indicates continue trying forever. `blocking_timeout` can be specified as a float or integer, both representing the number of seconds to wait.

`lock_class` forces the specified lock implementation.

`thread_local` indicates whether the lock token is placed in thread-local storage. By default, the token is placed in thread local storage so that a thread only sees its token, not a token set by another thread. Consider the following timeline:

time: 0, thread-1 acquires *my-lock*, with a timeout of 5 seconds. thread-1 sets the token to "abc"

time: 1, thread-2 blocks trying to acquire *my-lock* using the Lock instance.

time: 5, thread-1 has not yet completed. redis expires the lock key.

time: 5, thread-2 acquired *my-lock* now that it's available. thread-2 sets the token to "xyz"

time: 6, thread-1 finishes its work and calls release(). if the token is *not* stored in thread local storage, then thread-1 would see the token value as "xyz" and would be able to successfully release the thread-2's lock.

In some use cases it's necessary to disable thread local storage. For example, if you have code where one thread acquires a lock and passes that lock instance to a worker thread to release later. If thread local storage isn't disabled in this case, the worker thread won't see the token set by the thread that acquired the lock. Our assumption is that these cases aren't common and as such default to using thread local storage.

```
>>> async def example():
>>>     client = aredis.StrictRedis()
>>>     await client.flushall()
>>>     lock = client.lock('lalala')
>>>     print(await lock.acquire())
>>>     print(await lock.acquire(blocking=False))
>>>     print(await lock.release())
>>>     print(await lock.acquire())
True
False
None
True
```

6.8.2 Cluster Lock

Cluster lock is supposed to solve distributed lock problem in redis cluster. Since high availability is provided by redis cluster using master-slave model, the kind of lock aims to solve the fail-over problem referred in distributed lock post given by redis official.

Why not use Redlock algorithm provided by official directly?

It is impossible to make a key hashed to different nodes in a redis cluster and hard to generate keys in a specific rule and make sure they do not migrated in cluster. In the worst situation, all key slots may exists in one node. Then the availability will be the same as one key in one node.

For more discussion please see: https://github.com/NoneGG/aredis/issues/55

To gather more ideas i also raise a problem in stackoverflow: Not_a_Golfer's solution is awesome, but considering the migration problem, i think this solution may be better. https://stackoverflow.com/questions/46438857/how-to-create-a-distributed-lock-using-redis-cluster

My solution is described below:

1. random token + SETNX + expire time to acquire a lock in cluster master node

2. if lock is acquired successfully then check the lock in slave nodes(may there be N slave nodes) using READONLY mode, if N/2+1 is synced successfully then break the check and return True, time used to check is also accounted into expire time

3.Use lua script described in redlock algorithm to release lock with the client which has the randomly generated token, if the client crashes, then wait until the lock key expired.

Actually you can regard the algorithm as a master-slave version of redlock, which is designed for multi master nodes.

Please read these article below before using this cluster lock in your app. https://redis.io/topics/distlock http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html http://antirez.com/news/101

```
>>> async def example():
>>>     client = aredis.StrictRedis()
>>>     await client.flushall()
>>>     lock = client.lock('lalala', lock_class=ClusterLock, timeout=1)
>>>     print(await lock.acquire())
>>>     print(await lock.acquire(blocking=False))
>>>     print(await lock.release())
>>>     print(await lock.acquire())
True
False
None
True
```

6.8.3 Cache

Cache has support for both single redis node and redis cluster.

There are two kinds of cache class(Cache & HerdCache) provided. Cache classes consists of IdentityGenerator(used to generate unique identity in redis), Serializer(used to serialize content before compress and finally put in redis), Compressor(used to compress cache to reduce memory usage of redis. IdentityGenerator, Serializer, Compressor can be overwritten to meet your special needs, and if you don't need it, just set them to None when intialize a cache:

cache(*self*, *name*, *cache_class=Cache*, *identity_generator_class=IdentityGenerator*, *compressor_class=Compressor*, *serializer_class=Serializer*, **args*, ***kwargs*)

Return a cache object using default identity generator, serializer and compressor.

`name` is used to identify the series of your cache

`cache_class` Cache is for normal use and HerdCache is used in case of Thundering Herd Problem

`identity_generator_class` is the class used to generate the real unique key in cache, can be overwritten to meet your special needs. It should provide *generate* API

`compressor_class` is the class used to compress cache in redis, can be overwritten with API *compress* and *decompress* retained.

`serializer_class` is the class used to serialize content before compress, can be overwritten with API *serialize* and *deserialize* retained.

```
>>> class CustomIdentityGenerator(IdentityGenerator):
>>>     def generate(self, key, content):
>>>         return key
>>>
>>> def expensive_work(data):
>>> """some work that waits for io or occupy cpu"""
>>>     return data
>>>
>>> async def example():
>>>     client = aredis.StrictRedis()
>>>     await client.flushall()
>>>     cache = client.cache('example_cache',
>>>             identity_generator_class=CustomIdentityGenerator)
>>>     data = {1: 1}
>>>     await cache.set('example_key', expensive_work(data), data)
>>>     res = await cache.get('example_key', data)
>>>     assert res == expensive_work(data)
```

For ease of use and expandability, only *set*, *set_many*, *exists*, *delete*, *delete_many*, *ttl*, *get* APIs are realized.

HerdCache is a solution for thundering herd problem It is suitable for scene with low consistency and in which refresh cache costs a lot. It will save redundant update work when there are multi process read cache from redis and the cache is expired. If the cache is expired is judged by the expire time saved with each key, and the real expire time of the key *real_expire_time = the time key is set + expire_time + herd_timeout* once a process find out that the cache is expired, it will reset the expire time saved in redis with *new_expire_time = the time key is found expired + extend_expire_time*, and return None(act like cache expired), so that other processes will not noticed the cache expired.

CHAPTER 7

The Community Guide

7.1 Testing

7.1.1 StrictRedis

All tests are built on the base of simplest redis server with default config.

Redis server setup

To test against the latest stable redis server from source, use:

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install tcl8.5
$ wget http://download.redis.io/releases/redis-stable.tar.gz
$ tar xzf redis-stable.tar.gz
$ cd redis-stable
$ make test
$ make install
$ sudo utils/install_server.sh
$ sudo service redis_6379 start
```

You can also use any version of Redis installed from your OS package manager (example for OSX: `brew install redis`), in which case starting the server is as simple as running:

```
$ redis-server
```

7.1.2 StrictRedisCluster

All tests are currently built around a 6 redis server cluster setup (3 masters + 3 slaves). One server must be using port 7000 for redis cluster discovery. The easiest way to setup a cluster is to use Docker.

Redis cluster setup

A fully functional docker image can be found at https://github.com/Grokzen/docker-redis-cluster

To turn on a cluster which should pass all tests, run:

```
$ docker run --rm -it -p7000:7000 -p7001:7001 -p7002:7002 -p7003:7003 -p7004:7004 -
↪p7005:7005 -e IP='0.0.0.0' grokzen/redis-cluster:latest
```

7.1.3 Run test

To run test you should install dependency firstly.

```
$ pip install -r dev_requirements.txt
$ pytest tests/
```

7.2 Release Notes

7.2.1 master

- add TCP Keep-alive support by passing use the *socket_keepalive=True* option. Finer grain control can be achieved using the *socket_keepalive_options* option which expects a dictionary with any of the keys (*socket.TCP_KEEPIDLE*, *socket.TCP_KEEPCNT*, *socket.TCP_KEEPINTVL*) and integers for values. Thanks Stefan Tjarks.

7.2.2 1.0.1

- add scan_iter, sscan_iter, hscan_iter, zscan_iter and corresponding unit tests
- fix bug of *PubSub.run_in_thread*
- add more examples
- change *Script.register* to *Script.execute*

7.2.3 1.0.2

- add support for cache (Cache and HerdCache class)
- fix bug of *PubSub.run_in_thread*

7.2.4 1.0.4

- add support for command *pubsub channel*, *pubsub numpat* and *pubsub numsub*
- add support for command *client pause*
- reconsitution of commands to make develop easier(which is transparent to user)

7.2.5 1.0.5

- fix bug in setup.py when using pip to install aredis

7.2.6 1.0.6

- bitfield set/get/incrby/overflow supported
- new command *hstrlen* supported
- new command *unlink* supported
- new command *touch* supported

7.2.7 1.0.7

- introduce loop argument to aredis
- add support for command *cluster slots*
- add support for redis cluster

7.2.8 1.0.8

- fix initialization bug of redis cluster client
- add example to explain how to use *client reply on | off | skip*

7.2.9 1.0.9

- fix bug of pubsub, in some env AssertionError is raised because connection is used again after reader stream being fed eof
- add reponse decoding related options(*encoding* & *decode_responses*), make client easier to use
- add support for command *cluster forget*
- add support for command option *spop count*

7.2.10 1.1.0

- sync optimization of scripting from redis-py made by bgreenberg related pull request
- sync bug fixed of *geopos* from redis-py made by categulario related pull request
- fix bug which makes pipeline callback function not executed
- fix error caused by byte decode issues in sentinel
- add basic transaction support for single node in cluster
- fix bug of get_random_connection reported by myrfy001

7.2.11 1.1.1

- fix bug: connection with unread response being released to connection pool will lead to parse error, now this kind of connection will be destructed directly. related issue
- fix bug: remove Connection.can_read check which may lead to block in awaiting pubsub message. Connection.can_read api will be deprecated in next release. related issue
- add c extension to speedup crc16, which will speedup cluster slot hashing
- add error handling for asyncio.futures.Cancelled error, which may cause error in response parsing.
- sync optimization of client list made by swilly22 from redis-py
- add support for distributed lock using redis cluster

7.2.12 1.1.2

- fix bug: redis command encoding bug
- optimization: sync change on acquring lock from redis-py
- fix bug: decrement connection count on connection disconnected
- fix bug: optimize code proceed single node slots
- fix bug: initiation error of aws cluster client caused by not appropiate function list used
- fix bug: use *ssl_context* instead of ssl_keyfile,ssl_certfile,ssl_cert_reqs,ssl_ca_certs in intialization of connection_pool

7.2.13 1.1.3

- allow use of zadd options for zadd in sorted sets
- fix bug: use inspect.isawaitable instead of typing.Awaitable to judge if an object is awaitable
- fix bug: implicitly disconnection on cancelled error (#84)
- new: add support for streams‘(including commands not officially released, see ‘streams)

7.2.14 1.1.4

- fix bug: fix cluster port parsing for redis 4+(node info)
- fix bug: wrong parse method of scan_iter in cluster mode
- fix bug: When using “zrange” with “desc=True” parameter, it returns a coroutine without “await”
- fix bug: do not use stream_timeout in the PubSubWorkerThread
- opt: add socket_keepalive options
- new: add ssl param in get_redis_link to support ssl mode
- new: add ssl_context to StrictRedis constructor and make it higher priority than ssl parameter

7.2.15 1.1.5

- new: Dev conn pool max idle time (#111) release connection if max-idle-time exceeded
- update: discard travis-CI
- Fix bug: new stream id used for test_streams

7.2.16 1.1.6

- Fixbug: parsing stream messgae with empty payload will cause error(#116)
- Fixbug: Let ClusterConnectionPool handle skip_full_coverage_check (#118)
- New: threading local issue in coroutine, use contextvars instead of threading local in case of the safety of thread local mechanism being broken by coroutine (#120)
- New: support Python 3.8

7.2.17 1.1.7

- Fixbug: ModuleNotFoundError raised when install aredis 1.1.6 with Python3.6

7.2.18 1.1.8

- Fixbug: connection is disconnected before idel check, valueError will be raised if a connection(not exist) is removed from connection list
- Fixbug: abstract compat.py to handle import problem of asyncio.future
- Fixbug: When cancelling a task, CancelledError exception is not propagated to client
- Fixbug: XREAD command should accept 0 as a block argument
- Fixbug: In redis cluster mode, XREAD command does not function properly
- Fixbug: slave connection params when there are no slaves

7.3 Author

aredis is developed and maintained by Jason Chen (jason0916phoenix@gmail.com, please use 847671011@qq.com in case your email is not responsed)

Most of its code come from redis-py written by Andy McCurdy (sedrik@gmail.com).

The cluster part is ported from redis-py-cluster written by Grokzen

7.4 Project Contributors

Added in the order they contributed. Thank you for your help to make aredis better!

Authors who contributed code or testing:

- inytar - https://github.com/inytar

- hqy - https://github.com/hqy
- melhin - https://github.com/melhin
- stj - https://github.com/stj

7.5 Licensing

7.6 Todo list

1. more detailed doc for cluster
2. more tests on cluster part
3. more commands supported

Index