aioredis Documentation

Release 1.2.0

Alexey Popravka

Dec 21, 2018

Contents

1 Features **3**

2 Installation **5**

3 Requirements **7**

4 Benchmarks **9**

5 Contribute **11**

6 License **13**

7 Contents **15**

7.1 Getting started . . . 15
7.1.1 Commands Pipelining . . . 15
7.1.2 Multi/Exec transactions . . . 16
7.1.3 Pub/Sub mode . . . 16
7.1.4 Python 3.5 `async with` / `async for` support . . . 18
7.1.5 SSL/TLS support . . . 18
7.2 Migrating from v0.3 to v1.0 . . . 18
7.2.1 aioredis.create_pool . . . 19
7.2.2 aioredis.create_reconnecting_redis . . . 19
7.2.3 aioredis.Redis . . . 20
7.2.4 Blocking operations and connection sharing . . . 20
7.2.5 Sorted set commands return values . . . 21
7.2.6 Hash `hscan` command now returns list of tuples . . . 22
7.3 `aioredis` — API Reference . . . 23
7.3.1 Connection . . . 23
7.3.2 Connections Pool . . . 26
7.3.3 Pub/Sub Channel object . . . 29
7.3.4 Exceptions . . . 30
7.3.5 Commands Interface . . . 32
7.4 `aioredis.Redis` — Commands Mixins Reference . . . 34
7.4.1 Generic commands . . . 35
7.4.2 Geo commands . . . 37
7.4.3 Strings commands . . . 39
7.4.4 Hash commands . . . 41

7.4.5 List commands . . . 42
7.4.6 Set commands . . . 44
7.4.7 Sorted Set commands . . . 45
7.4.8 Server commands . . . 47
7.4.9 HyperLogLog commands . . . 49
7.4.10 Transaction commands . . . 50
7.4.11 Scripting commands . . . 52
7.4.12 Server commands . . . 52
7.4.13 Pub/Sub commands . . . 54
7.4.14 Cluster commands . . . 55
7.4.15 Streams commands . . . 55
7.5 aioredis.abc — Interfaces Reference . . . 56
7.6 aioredis.pubsub — Pub/Sub Tools Reference . . . 58
7.7 aioredis.sentinel — Sentinel Client Reference . . . 60
7.7.1 RedisSentinel . . . 60
7.7.2 SentinelPool . . . 62
7.8 Examples of aioredis usage . . . 63
7.8.1 Low-level connection usage example . . . 63
7.8.2 Connections pool example . . . 64
7.8.3 Commands example . . . 64
7.8.4 Transaction example . . . 65
7.8.5 Pub/Sub example . . . 66
7.8.6 Scan command example . . . 66
7.8.7 Sentinel client . . . 67
7.9 Contributing . . . 67
7.9.1 Code style . . . 68
7.9.2 Running tests . . . 68
7.9.3 Writing tests . . . 69
7.10 Releases . . . 71
7.10.1 1.2.0 (2018-10-24) . . . 71
7.10.2 1.1.0 (2018-02-16) . . . 71
7.10.3 1.0.0 (2017-11-17) . . . 71
7.10.4 0.3.5 (2017-11-08) . . . 73
7.10.5 0.3.4 (2017-10-25) . . . 73
7.10.6 0.3.3 (2017-06-30) . . . 73
7.10.7 0.3.2 (2017-06-21) . . . 73
7.10.8 0.3.1 (2017-05-09) . . . 73
7.10.9 0.3.0 (2017-01-11) . . . 73
7.10.10 0.2.9 (2016-10-24) . . . 74
7.10.11 0.2.8 (2016-07-22) . . . 74
7.10.12 0.2.7 (2016-05-27) . . . 74
7.10.13 0.2.6 (2016-03-30) . . . 75
7.10.14 0.2.5 (2016-03-02) . . . 75
7.10.15 0.2.4 (2015-10-13) . . . 75
7.10.16 0.2.3 (2015-08-14) . . . 76
7.10.17 0.2.2 (2015-07-07) . . . 76
7.10.18 0.2.1 (2015-07-06) . . . 76
7.10.19 0.2.0 (2015-06-04) . . . 76
7.10.20 0.1.5 (2014-12-09) . . . 77
7.10.21 0.1.4 (2014-09-22) . . . 77
7.10.22 0.1.3 (2014-08-08) . . . 77
7.10.23 0.1.2 (2014-07-31) . . . 77
7.10.24 0.1.1 (2014-07-07) . . . 78
7.10.25 0.1.0 (2014-06-24) . . . 78

7.11 Glossary 78

8 Indices and tables **79**

Python Module Index **81**

asyncio (**PEP 3156**) Redis client library.

The library is intended to provide simple and clear interface to Redis based on *asyncio*.

CHAPTER 1

Features

hiredis parser	Yes
Pure-python parser	Yes
Low-level & High-level APIs	Yes
Connections Pool	Yes
Pipelining support	Yes
Pub/Sub support	Yes
Sentinel support	Yes[1]
Redis Cluster support	WIP
Trollius (python 2.7)	No
Tested CPython versions	3.5, 3.6[2]
Tested PyPy3 versions	5.9.0
Tested for Redis server	2.6, 2.8, 3.0, 3.2, 4.0
Support for dev Redis server	through low-level API

[1] Sentinel support is available in master branch. This feature is not yet stable and may have some issues.
[2] For Python 3.3, 3.4 support use aioredis v0.3.

CHAPTER 2

Installation

The easiest way to install aioredis is by using the package on PyPi:

```
pip install aioredis
```

CHAPTER 3

Requirements

- Python 3.5.3+
- *hiredis*

CHAPTER 4

Benchmarks

Benchmarks can be found here: https://github.com/popravich/python-redis-benchmark

CHAPTER 5

Contribute

- Issue Tracker: https://github.com/aio-libs/aioredis/issues
- Source Code: https://github.com/aio-libs/aioredis
- Contributor's guide: *Contributing*

Feel free to file an issue or make pull request if you find any bugs or have some suggestions for library improvement.

CHAPTER 6

License

The aioredis is offered under MIT license.

CHAPTER 7

Contents

7.1 Getting started

7.1.1 Commands Pipelining

Commands pipelining is built-in.

Every command is sent to transport at-once (ofcourse if no `TypeError`/`ValueError` was raised)

When you making a call with `await` / `yield from` you will be waiting result, and then gather results.

Simple example show both cases (`get source code`):

```
# No pipelining;
async def wait_each_command():
    val = await redis.get('foo')     # wait until `val` is available
    cnt = await redis.incr('bar')    # wait until `cnt` is available
    return val, cnt

# Sending multiple commands and then gathering results
async def pipelined():
    fut1 = redis.get('foo')      # issue command and return future
    fut2 = redis.incr('bar')     # issue command and return future
    # block until results are available
    val, cnt = await asyncio.gather(fut1, fut2)
    return val, cnt
```

Note: For convenience *aioredis* provides *pipeline()* method allowing to execute bulk of commands as one (`get source code`):

```
# Explicit pipeline
async def explicit_pipeline():
    pipe = redis.pipeline()
```

(continues on next page)

(continued from previous page)

```
    fut1 = pipe.get('foo')
    fut2 = pipe.incr('bar')
    result = await pipe.execute()
    val, cnt = await asyncio.gather(fut1, fut2)
    assert result == [val, cnt]
    return val, cnt
```

7.1.2 Multi/Exec transactions

aioredis provides several ways for executing transactions:

- when using raw connection you can issue `Multi`/`Exec` commands manually;
- when using *aioredis.Redis* instance you can use *multi_exec()* transaction pipeline.

multi_exec() method creates and returns new *MultiExec* object which is used for buffering commands and then executing them inside MULTI/EXEC block.

Here is a simple example (`get source code`):

```
1 async def transaction():
2     tr = redis.multi_exec()
3     future1 = tr.set('foo', '123')
4     future2 = tr.set('bar', '321')
5     result = await tr.execute()
6     assert result == await asyncio.gather(future1, future2)
7     return result
```

As you can notice `await` is **only** used at line 5 with `tr.execute` and **not with** `tr.set(...)` calls.

Warning: It is very important not to `await` buffered command (ie `tr.set('foo', '123')`) as it will block forever.

The following code will block forever:

```
tr = redis.multi_exec()
await tr.incr('foo')    # that's all. we've stuck!
```

7.1.3 Pub/Sub mode

aioredis provides support for Redis Publish/Subscribe messaging.

To switch connection to subscribe mode you must execute `subscribe` command by yield'ing from *subscribe()* it returns a list of *Channel* objects representing subscribed channels.

As soon as connection is switched to subscribed mode the channel will receive and store messages (the `Channel` object is basically a wrapper around `asyncio.Queue`). To read messages from channel you need to use *get()* or *get_json()* coroutines.

Note: In Pub/Sub mode redis connection can only receive messages or issue (P)SUBSCRIBE / (P)UNSUBSCRIBE commands.

Pub/Sub example (get source code):

```
sub = await aioredis.create_redis(
     'redis://localhost')

ch1, ch2 = await sub.subscribe('channel:1', 'channel:2')
assert isinstance(ch1, aioredis.Channel)
assert isinstance(ch2, aioredis.Channel)

async def async_reader(channel):
    while await channel.wait_message():
        msg = await channel.get(encoding='utf-8')
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

tsk1 = asyncio.ensure_future(async_reader(ch1))

# Or alternatively:

async def async_reader2(channel):
    while True:
        msg = await channel.get(encoding='utf-8')
        if msg is None:
            break
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

tsk2 = asyncio.ensure_future(async_reader2(ch2))
```

Pub/Sub example (get source code):

```
async def reader(channel):
    while (await channel.wait_message()):
        msg = await channel.get(encoding='utf-8')
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

        if msg == STOPWORD:
            return

with await pool as conn:
    await conn.execute_pubsub('subscribe', 'channel:1')
    channel = conn.pubsub_channels['channel:1']
    await reader(channel)  # wait for reader to complete
    await conn.execute_pubsub('unsubscribe', 'channel:1')

# Explicit connection usage
conn = await pool.acquire()
try:
    await conn.execute_pubsub('subscribe', 'channel:1')
    channel = conn.pubsub_channels['channel:1']
    await reader(channel)  # wait for reader to complete
    await conn.execute_pubsub('unsubscribe', 'channel:1')
finally:
    pool.release(conn)
```

7.1.4 Python 3.5 `async with` / `async for` support

aioredis is compatible with **PEP 492**.

`Pool` can be used with async with (`get source code`):

```
pool = await aioredis.create_pool(
    'redis://localhost')
async with pool.get() as conn:
    value = await conn.execute('get', 'my-key')
    print('raw value:', value)
```

It also can be used with `await`:

```
pool = await aioredis.create_pool(
    'redis://localhost')
# This is exactly the same as:
#   with (yield from pool) as conn:
with (await pool) as conn:
    value = await conn.execute('get', 'my-key')
    print('raw value:', value)
```

New `scan`-family commands added with support of async for (`get source code`):

```
redis = await aioredis.create_redis(
    'redis://localhost')

async for key in redis.iscan(match='something*'):
    print('Matched:', key)

async for name, val in redis.ihscan(key, match='something*'):
    print('Matched:', name, '->', val)

async for val in redis.isscan(key, match='something*'):
    print('Matched:', val)

async for val, score in redis.izscan(key, match='something*'):
    print('Matched:', val, ':', score)
```

7.1.5 SSL/TLS support

Though Redis server does not support data encryption it is still possible to setup Redis server behind SSL proxy. For such cases *aioredis* library support secure connections through `asyncio` SSL support. See BaseEventLoop.create_connection for details.

7.2 Migrating from v0.3 to v1.0

API changes and backward incompatible changes:

- *aioredis.create_pool*
- *aioredis.create_reconnecting_redis*
- *aioredis.Redis*

- *Blocking operations and connection sharing*
- *Sorted set commands return values*
- *Hash `hscan` command now returns list of tuples*

7.2.1 aioredis.create_pool

`create_pool()` now returns *`ConnectionsPool`* instead of `RedisPool`.

This means that pool now operates with *`RedisConnection`* objects and not *`Redis`*.

v0.3	`pool = await aioredis.create_pool(('localhost', 6379))`
	
	`with await pool as redis:`
	`  # calling methods of Redis class`
	`  await redis.lpush('list-key', 'item1', 'item2')`

v1.0	`pool = await aioredis.create_pool(('localhost', 6379))`
	
	`with await pool as conn:`
	`  # calling conn.lpush will raise AttributeError exception`
	`  await conn.execute('lpush', 'list-key', 'item1', 'item2')`

7.2.2 aioredis.create_reconnecting_redis

`create_reconnecting_redis()` has been dropped.

`create_redis_pool()` can be used instead of former function.

v0.3	`redis = await aioredis.create_reconnecting_redis(`
	`  ('localhost', 6379))`
	
	`await redis.lpush('list-key', 'item1', 'item2')`

v1.0	`redis = await aioredis.create_redis_pool(`
	`  ('localhost', 6379))`
	
	`await redis.lpush('list-key', 'item1', 'item2')`

`create_redis_pool` returns *Redis* initialized with `ConnectionsPool` which is responsible for reconnecting to server.

Also `create_reconnecting_redis` was patching the `RedisConnection` and breaking `closed` property (it was always `True`).

7.2.3 aioredis.Redis

Redis class now operates with objects implementing *aioredis.abc.AbcConnection* interface. *RedisConnection* and *ConnectionsPool* are both implementing `AbcConnection` so it is become possible to use same API when working with either single connection or connections pool.

v0.3	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.lpush('list-key', 'item1', 'item2')`
	
	`pool = await aioredis.create_pool(('localhost', 6379))`
	`redis = await pool.acquire()  # get Redis object`
	`await redis.lpush('list-key', 'item1', 'item2')`

v1.0	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.lpush('list-key', 'item1', 'item2')`
	
	`redis = await aioredis.create_redis_pool(('localhost', 6379))`
	`await redis.lpush('list-key', 'item1', 'item2')`

7.2.4 Blocking operations and connection sharing

Current implementation of `ConnectionsPool` by default **execute every command on random connection**. The *Pros* of this is that it allowed implementing `AbcConnection` interface and hide pool inside `Redis` class, and also keep pipelining feature (like RedisConnection.execute). The *Cons* of this is that **different tasks may use same connection and block it** with some long-running command.

We can call it **Shared Mode** — commands are sent to random connections in pool without need to lock [connection]:

```
redis = await aioredis.create_redis_pool(
    ('localhost', 6379),
    minsize=1,
    maxsize=1)

async def task():
    # Shared mode
    await redis.set('key', 'val')
```

(continues on next page)

(continued from previous page)

```
asyncio.ensure_future(task())
asyncio.ensure_future(task())
# Both tasks will send commands through same connection
# without acquiring (locking) it first.
```

Blocking operations (like `blpop`, `brpop` or long-running LUA scripts) in **shared mode** mode will block connection and thus may lead to whole program malfunction.

This *blocking* issue can be easily solved by using exclusive connection for such operations:

```
redis = await aioredis.create_redis_pool(
    ('localhost', 6379),
    minsize=1,
    maxsize=1)

async def task():
   # Exclusive mode
   with await redis as r:
       await r.set('key', 'val')
asyncio.ensure_future(task())
asyncio.ensure_future(task())
# Both tasks will first acquire connection.
```

We can call this **Exclusive Mode** — context manager is used to acquire (lock) exclusive connection from pool and send all commands through it.

Note: This technique is similar to v0.3 pool usage:

```
# in aioredis v0.3
pool = await aioredis.create_pool(('localhost', 6379))
with await pool as redis:
    # Redis is bound to exclusive connection
    redis.set('key', 'val')
```

7.2.5 Sorted set commands return values

Sorted set commands (like `zrange`, `zrevrange` and others) that accept `withscores` argument now **return list of tuples** instead of plain list.

v0.3	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.zadd('zset-key', 1, 'one', 2, 'two')`
	`res = await redis.zrage('zset-key', withscores=True)`
	`assert res == [b'one', 1, b'two', 2]`
	
	`# not an esiest way to make a dict`
	`it = iter(res)`
	`assert dict(zip(it, it)) == {b'one': 1, b'two': 2}`

v1.0	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.zadd('zset-key', 1, 'one', 2, 'two')`
	`res = await redis.zrage('zset-key', withscores=True)`
	`assert res == [(b'one', 1), (b'two', 2)]`
	
	`# now its easier to make a dict of it`
	`assert dict(res) == {b'one': 1, b'two': 2}`

7.2.6 Hash `hscan` command now returns list of tuples

`hscan` updated to return a list of tuples instead of plain mixed key/value list.

v0.3	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.hmset('hash', 'one', 1, 'two', 2)`
	`cur, data = await redis.hscan('hash')`
	`assert data == [b'one', b'1', b'two', b'2']`
	
	`# not an esiest way to make a dict`
	`it = iter(data)`
	`assert dict(zip(it, it)) == {b'one': b'1', b'two': b'2'}`

v1.0	`redis = await aioredis.create_redis(('localhost', 6379))`
	`await redis.hmset('hash', 'one', 1, 'two', 2)`
	`cur, data = await redis.hscan('hash')`
	`assert data == [(b'one', b'1'), (b'two', b'2')]`
	
	`# now its easier to make a dict of it`
	`assert dict(data) == {b'one': b'1': b'two': b'2'}`

7.3 `aioredis` — API Reference

7.3.1 Connection

Redis Connection is the core function of the library. Connection instances can be used as is or through *pool* or *high-level API*.

Connection usage is as simple as:

```
import asyncio
import aioredis

async def connect_uri():
    conn = await aioredis.create_connection(
        'redis://localhost/0')
    val = await conn.execute('GET', 'my-key')

async def connect_tcp():
    conn = await aioredis.create_connection(
        ('localhost', 6379))
    val = await conn.execute('GET', 'my-key')

async def connect_unixsocket():
    conn = await aioredis.create_connection(
        '/path/to/redis/socket')
    # or uri 'unix:///path/to/redis/socket?db=1'
    val = await conn.execute('GET', 'my-key')
```

(continues on next page)

(continued from previous page)

```
asyncio.get_event_loop().run_until_complete(connect_tcp())
asyncio.get_event_loop().run_until_complete(connect_unixsocket())
```

coroutine aioredis.**create_connection**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *parser=None*, *loop=None*, *timeout=None*)

Creates Redis connection.

Changed in version v0.3.1: `timeout` argument added.

Changed in version v1.0: `parser` argument added.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be one of the following:
  - a Redis URI — `"redis://host:6379/0?encoding=utf-8"`; `"redis://:password@host:6379/0?encoding=utf-8"`;
  - a (host, port) tuple — `('localhost', 6379)`;
  - or a unix domain socket path string — `"/path/to/redis.sock"`.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or None*) – Password to use if redis server instance requires authorization.
- **ssl** (`ssl.SSLContext` or True or None) – SSL context that is passed through to `asyncio.BaseEventLoop.create_connection()`.
- **encoding** (*str or None*) – Codec to use for response decoding.
- **parser** (*callable or None*) – Protocol parser class. Can be used to set custom protocol reader; expected same interface as `hiredis.Reader`.
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).
- **timeout** (*float greater than 0 or None*) – Max time to open a connection, otherwise raise `asyncio.TimeoutError` exception. `None` by default

Returns *RedisConnection* instance.

class aioredis.**RedisConnection**

Bases: *abc.AbcConnection*

Redis connection interface.

address

Redis server address; either IP-port tuple or unix socket str (*read-only*). IP is either IPv4 or IPv6 depending on resolved host part in initial address.

New in version v0.2.8.

db

Current database index (*read-only*).

encoding

Current codec for response decoding (*read-only*).

closed

Set to `True` if connection is closed (*read-only*).

in_transaction
Set to `True` when MULTI command was issued (*read-only*).

pubsub_channels
Read-only dict with subscribed channels. Keys are bytes, values are *Channel* instances.

pubsub_patterns
Read-only dict with subscribed patterns. Keys are bytes, values are *Channel* instances.

in_pubsub
Indicates that connection is in PUB/SUB mode. Provides the number of subscribed channels. *Read-only*.

execute(*command*, **args*, *encoding=_NOTSET*)
Execute Redis command.

The method is **not a coroutine** itself but instead it writes to underlying transport and returns a `asyncio.Future` waiting for result.

Parameters

- **command** (*str, bytes, bytearray*) – Command to execute
- **encoding** (*str or None*) – Keyword-only argument for overriding response decoding. By default will use connection-wide encoding. May be set to None to skip response decoding.

Raises

- **TypeError** – When any of arguments is None or can not be encoded as bytes.
- ***aioredis.ReplyError*** – For redis error replies.
- ***aioredis.ProtocolError*** – When response can not be decoded and/or connection is broken.

Returns Returns bytes or int reply (or str if encoding was set)

execute_pubsub(*command*, **channels_or_patterns*)
Method to execute Pub/Sub commands. The method is not a coroutine itself but returns a `asyncio.gather()` coroutine. Method also accept *aioredis.Channel* instances as command arguments:

```
>>> ch1 = Channel('A', is_pattern=False, loop=loop)
>>> await conn.execute_pubsub('subscribe', ch1)
[[b'subscribe', b'A', 1]]
```

Changed in version v0.3: The method accept *Channel* instances.

Parameters

- **command** (*str, bytes, bytearray*) – One of the following Pub/Sub commands: `subscribe`, `unsubscribe`, `psubscribe`, `punsubscribe`.
- ***channels_or_patterns** – Channels or patterns to subscribe connection to or unsubscribe from. At least one channel/pattern is required.

Returns

Returns a list of subscribe/unsubscribe messages, ex:

```
>>> await conn.execute_pubsub('subscribe', 'A', 'B')
[[b'subscribe', b'A', 1], [b'subscribe', b'B', 2]]
```

close()
Closes connection.

Mark connection as closed and schedule cleanup procedure.

All pending commands will be canceled with *ConnectionForcedCloseError*.

wait_closed()
Coroutine waiting for connection to get closed.

select(*db*)
Changes current db index to new one.

Parameters **db** (*int*) – New redis database index.

Raises

- **TypeError** – When db parameter is not int.
- **ValueError** – When db parameter is less than 0.

Return True Always returns True or raises exception.

auth(*password*)
Send AUTH command.

Parameters **password** (*str*) – Plain-text password

Return bool True if redis replied with 'OK'.

7.3.2 Connections Pool

The library provides connections pool. The basic usage is as follows:

```
import aioredis

async def sample_pool():
    pool = await aioredis.create_pool('redis://localhost')
    val = await pool.execute('get', 'my-key')
```

aioredis.**create_pool**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *minsize=1*, *maxsize=10*, *parser=None*, *loop=None*, *create_connection_timeout=None*, *pool_cls=None*, *connection_cls=None*)
A coroutine that instantiates a pool of *RedisConnection*.

Changed in version v0.2.7: `minsize` default value changed from 10 to 1.

Changed in version v0.2.8: Disallow arbitrary ConnectionsPool maxsize.

Deprecated since version v0.2.9: *commands_factory* argument is deprecated and will be removed in *v1.0*.

Changed in version v0.3.2: `create_connection_timeout` argument added.

New in version v1.0: `parser`, `pool_cls` and `connection_cls` arguments added.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be one of the following:
  - a Redis URI — `"redis://host:6379/0?encoding=utf-8"`;
  - a (host, port) tuple — `('localhost', 6379)`;
  - or a unix domain socket path string — `"/path/to/redis.sock"`.
- **db** (*int*) – Redis database index to switch to when connected.

- **password** (*str or None*) – Password to use if redis server instance requires authorization.
- **ssl** (ssl.SSLContext or True or None) – SSL context that is passed through to asyncio.BaseEventLoop.create_connection().
- **encoding** (*str or None*) – Codec to use for response decoding.
- **minsize** (*int*) – Minimum number of free connection to create in pool. 1 by default.
- **maxsize** (*int*) – Maximum number of connection to keep in pool. 10 by default. Must be greater than 0. None is disallowed.
- **parser** (*callable or None*) – Protocol parser class. Can be used to set custom protocol reader; expected same interface as hiredis.Reader.
- **loop** (EventLoop) – An optional *event loop* instance (uses asyncio.get_event_loop() if not specified).
- **create_connection_timeout** (*float greater than 0 or None*) – Max time to open a connection, otherwise raise an asyncio.TimeoutError. None by default.
- **pool_cls** (aioredis.abc.AbcPool) – Can be used to instantiate custom pool class. This argument **must be** a subclass of *AbcPool*.
- **connection_cls** (aioredis.abc.AbcConnection) – Can be used to make pool instantiate custom connection classes. This argument **must be** a subclass of *AbcConnection*.

Returns *ConnectionsPool* instance.

class aioredis.**ConnectionsPool**

Bases: *abc.AbcPool*

Redis connections pool.

minsize
A minimum size of the pool (*read-only*).

maxsize
A maximum size of the pool (*read-only*).

size
Current pool size — number of free and used connections (*read-only*).

freesize
Current number of free connections (*read-only*).

db
Currently selected db index (*read-only*).

encoding
Current codec for response decoding (*read-only*).

closed
True if pool is closed.

New in version v0.2.8.

execute(*command*, **args*, ***kwargs*)
Execute Redis command in a free connection and return asyncio.Future waiting for result.

This method tries to pick a free connection from pool and send command through it at once (keeping pipelining feature provided by *aioredis.RedisConnection.execute()*). If no connection is found — returns coroutine waiting for free connection to execute command.

New in version v1.0.

execute_pubsub(*command*, **channels*)
Execute Redis (p)subscribe/(p)unsubscribe command.

ConnectionsPool picks separate free connection for pub/sub and uses it until pool is closed or connection is disconnected (unsubscribing from all channels/pattern will leave connection locked for pub/sub use).

There is no auto-reconnect for Pub/Sub connection as this will hide from user messages loss.

Has similar to *execute()* behavior, ie: tries to pick free connection from pool and switch it to pub/sub mode; or fallback to coroutine waiting for free connection and repeating operation.

New in version v1.0.

get_connection(*command*, *args=()*)
Gets free connection from pool returning tuple of (connection, address).

If no free connection is found – None is returned in place of connection.

Return type tuple(*RedisConnection* or None, str)

New in version v1.0.

coroutine clear()
Closes and removes all free connections in the pool.

coroutine select(*db*)
Changes db index for all free connections in the pool.

Parameters **db** (*int*) – New database index.

coroutine acquire(*command=None*, *args=()*)
Acquires a connection from *free pool*. Creates new connection if needed.

Parameters

- **command** – reserved for future.
- **args** – reserved for future.

Raises ***aioredis.PoolClosedError*** – if pool is already closed

release(*conn*)
Returns used connection back into pool.

When returned connection has db index that differs from one in pool the connection will be dropped. When queue of free connections is full the connection will be dropped.

Note: This method is **not a coroutine**.

Parameters **conn** (aioredis.RedisConnection) – A RedisConnection instance.

close()
Close all free and in-progress connections and mark pool as closed.

New in version v0.2.8.

coroutine wait_closed()
Wait until pool gets closed (when all connections are closed).

New in version v0.2.8.

7.3.3 Pub/Sub Channel object

Channel object is a wrapper around queue for storing received pub/sub messages.

class aioredis.**Channel**(*name*, *is_pattern*, *loop=None*)
Bases: *abc.AbcChannel*

Object representing Pub/Sub messages queue. It's basically a wrapper around asyncio.Queue.

name
Holds encoded channel/pattern name.

is_pattern
Set to True for pattern channels.

is_active
Set to True if there are messages in queue and connection is still subscribed to this channel.

coroutine get(***, *encoding=None*, *decoder=None*)
Coroutine that waits for and returns a message.

Return value is message received or None signifying that channel has been unsubscribed and no more messages will be received.

Parameters

- **encoding** (*str*) – If not None used to decode resulting bytes message.
- **decoder** (*callable*) – If specified used to decode message, ex. json.loads()

Raises ***aioredis.ChannelClosedError*** – If channel is unsubscribed and has no more messages.

get_json(***, *encoding="utf-8"*)
Shortcut to get(encoding="utf-8", decoder=json.loads)

coroutine wait_message()
Waits for message to become available in channel or channel is closed (unsubscribed).

Main idea is to use it in loops:

```
>>> ch = redis.channels['channel:1']
>>> while await ch.wait_message():
...     msg = await ch.get()
```

Return type bool

coroutine async-for iter(***, *encoding=None*, *decoder=None*)
Same as *get()* method but it is a native coroutine.

Usage example:

```
>>> async for msg in ch.iter():
...     print(msg)
```

New in version 0.2.5: Available for Python 3.5 only

7.3.4 Exceptions

exception aioredis.**RedisError**

Bases Exception

Base exception class for aioredis exceptions.

exception aioredis.**ProtocolError**

Bases *RedisError*

Raised when protocol error occurs. When this type of exception is raised connection must be considered broken and must be closed.

exception aioredis.**ReplyError**

Bases *RedisError*

Raised for Redis *error replies*.

exception aioredis.**MaxClientsError**

Bases *ReplyError*

Raised when maximum number of clients has been reached (Redis server configured value).

exception aioredis.**AuthError**

Bases *ReplyError*

Raised when authentication errors occur.

exception aioredis.**ConnectionClosedError**

Bases *RedisError*

Raised if connection to server was lost/closed.

exception aioredis.**ConnectionForcedCloseError**

Bases *ConnectionClosedError*

Raised if connection was closed with *RedisConnection.close()* method.

exception aioredis.**PipelineError**

Bases *RedisError*

Raised from *pipeline()* if any pipelined command raised error.

exception aioredis.**MultiExecError**

Bases *PipelineError*

Same as *PipelineError* but raised when executing multi_exec block.

exception aioredis.**WatchVariableError**

Bases *MultiExecError*

Raised if watched variable changed (EXEC returns None). Subclass of *MultiExecError*.

exception aioredis.**ChannelClosedError**

Bases *RedisError*

Raised from *aioredis.Channel.get()* when Pub/Sub channel is unsubscribed and messages queue is empty.

exception aioredis.**PoolClosedError**

Bases *RedisError*

Raised from *aioredis.ConnectionsPool.acquire()* when pool is already closed.

exception aioredis.**ReadOnlyError**

Bases *RedisError*

Raised from slave when read-only mode is enabled.

exception aioredis.**MasterNotFoundError**

Bases *RedisError*

Raised by Sentinel client if it can not find requested master.

exception aioredis.**SlaveNotFoundError**

Bases *RedisError*

Raised by Sentinel client if it can not find requested slave.

exception aioredis.**MasterReplyError**

Bases *RedisError*

Raised if establishing connection to master failed with `RedisError`, for instance because of required or wrong authentication.

exception aioredis.**SlaveReplyError**

Bases *RedisError*

Raised if establishing connection to slave failed with `RedisError`, for instance because of required or wrong authentication.

Exceptions Hierarchy

```
Exception
   RedisError
      ProtocolError
      ReplyError
         MaxClientsError
         AuthError
      PipelineError
         MultiExecError
            WatchVariableError
      ChannelClosedError
      ConnectionClosedError
         ConnectionForcedCloseError
      PoolClosedError
      ReadOnlyError
      MasterNotFoundError
      SlaveNotFoundError
      MasterReplyError
      SlaveReplyError
```

7.3.5 Commands Interface

The library provides high-level API implementing simple interface to Redis commands.

The usage is as simple as:

```
import aioredis

# Create Redis client bound to single non-reconnecting connection.
async def single_connection():
   redis = await aioredis.create_redis(
      'redis://localhost')
   val = await redis.get('my-key')

# Create Redis client bound to connections pool.
async def pool_of_connections():
   redis = await aioredis.create_redis_pool(
      'redis://localhost')
   val = await redis.get('my-key')

   # we can also use pub/sub as underlying pool
   #  has several free connections:
   ch1, ch2 = await redis.subscribe('chan:1', 'chan:2')
   # publish using free connection
   await redis.publish('chan:1', 'Hello')
   await ch1.get()
```

For commands reference — see *commands mixins reference*.

coroutine aioredis.**create_redis**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *commands_factory=Redis*, *parser=None*, *timeout=None*, *connection_cls=None*, *loop=None*)

This coroutine creates high-level Redis interface instance bound to single Redis connection (without auto-reconnect).

New in version v1.0: `parser`, `timeout` and `connection_cls` arguments added.

See also *RedisConnection* for parameters description.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be a (host, port) tuple, unix domain socket path string or a Redis URI string.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or bytes or None*) – Password to use if Redis server instance requires authorization.
- **ssl** (`ssl.SSLContext` or True or None) – SSL context that is passed through to `asyncio.BaseEventLoop.create_connection()`.
- **encoding** (*str or None*) – Codec to use for response decoding.
- **commands_factory** (*callable*) – A factory accepting single parameter – object implementing *AbcConnection* and returning an instance providing high-level interface to Redis. *Redis* by default.

- **parser** (*callable or None*) – Protocol parser class. Can be used to set custom protocol reader; expected same interface as `hiredis.Reader`.
- **timeout** (*float greater than 0 or None*) – Max time to open a connection, otherwise raise `asyncio.TimeoutError` exception. `None` by default
- **connection_cls** (`aioredis.abc.AbcConnection`) – Can be used to instantiate custom connection class. This argument **must be** a subclass of *AbcConnection*.
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).

Returns Redis client (result of `commands_factory` call), *Redis* by default.

coroutine aioredis.**create_redis_pool**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *commands_factory=Redis*, *minsize=1*, *maxsize=10*, *parser=None*, *timeout=None*, *pool_cls=None*, *connection_cls=None*, *loop=None*)

This coroutine create high-level Redis client instance bound to connections pool (this allows auto-reconnect and simple pub/sub use).

See also *ConnectionsPool* for parameters description.

Changed in version v1.0: `parser`, `timeout`, `pool_cls` and `connection_cls` arguments added.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be a (host, port) tuple, unix domain socket path string or a Redis URI string.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or bytes or None*) – Password to use if Redis server instance requires authorization.
- **ssl** (`ssl.SSLContext` or True or None) – SSL context that is passed through to `asyncio.BaseEventLoop.create_connection()`.
- **encoding** (*str or None*) – Codec to use for response decoding.
- **commands_factory** (*callable*) – A factory accepting single parameter – object implementing *AbcConnection* interface and returning an instance providing high-level interface to Redis. *Redis* by default.
- **minsize** (*int*) – Minimum number of connections to initialize and keep in pool. Default is 1.
- **maxsize** (*int*) – Maximum number of connections that can be created in pool. Default is 10.
- **parser** (*callable or None*) – Protocol parser class. Can be used to set custom protocol reader; expected same interface as `hiredis.Reader`.
- **timeout** (*float greater than 0 or None*) – Max time to open a connection, otherwise raise `asyncio.TimeoutError` exception. `None` by default
- **pool_cls** (`aioredis.abc.AbcPool`) – Can be used to instantiate custom pool class. This argument **must be** a subclass of *AbcPool*.
- **connection_cls** (`aioredis.abc.AbcConnection`) – Can be used to make pool instantiate custom connection classes. This argument **must be** a subclass of *AbcConnection*.
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).

Returns Redis client (result of `commands_factory` call), *Redis* by default.

7.4 aioredis.Redis — Commands Mixins Reference

This section contains reference for mixins implementing Redis commands.

Descriptions are taken from `docstrings` so may not contain proper markup.

class aioredis.**Redis**(*pool_or_conn*)

High-level Redis interface.

Gathers in one place Redis commands implemented in mixins.

For commands details see: http://redis.io/commands/#connection

Parameters **pool_or_conn** (*AbcConnection*) – Can be either *RedisConnection* or *ConnectionsPool*.

address

Redis connection address (if applicable).

auth(*password*)

Authenticate to server.

This method wraps call to *aioredis.RedisConnection.auth()*

close()

Close client connections.

closed

True if connection is closed.

connection

Either *aioredis.RedisConnection*, or *aioredis.ConnectionsPool* instance.

db

Currently selected db index.

echo(*message*, *, *encoding=<object object>*)

Echo the given string.

encoding

Current set codec or None.

in_transaction

Set to True when MULTI command was issued.

ping(*message=<object object>*, *, *encoding=<object object>*)

Ping the server.

Accept optional echo message.

quit()

Close the connection.

select(*db*)

Change the selected database for the current connection.

This method wraps call to *aioredis.RedisConnection.select()*

coroutine wait_closed()

Coroutine waiting until underlying connections are closed.

7.4.1 Generic commands

class aioredis.commands.**GenericCommandsMixin**
Generic commands mixin.

For commands details see: http://redis.io/commands/#generic

delete(*key*, **keys*)
Delete a key.

dump(*key*)
Dump a key.

exists(*key*, **keys*)
Check if key(s) exists.

Changed in version v0.2.9: Accept multiple keys; **return** type **changed** from bool to int.

expire(*key*, *timeout*)
Set a timeout on key.

if timeout is float it will be multiplied by 1000 coerced to int and passed to *pexpire* method.

Otherwise raises TypeError if timeout argument is not int.

expireat(*key*, *timestamp*)
Set expire timestamp on a key.

if timeout is float it will be multiplied by 1000 coerced to int and passed to *pexpireat* method.

Otherwise raises TypeError if timestamp argument is not int.

iscan(*, *match=None*, *count=None*)
Incrementally iterate the keys space using async for.

Usage example:

```
>>> async for key in redis.iscan(match='something*'):
...     print('Matched:', key)
```

keys(*pattern*, *, *encoding=<object object>*)
Returns all keys matching pattern.

migrate(*host*, *port*, *key*, *dest_db*, *timeout*, *, *copy=False*, *replace=False*)
Atomically transfer a key from a Redis instance to another one.

migrate_keys(*host*, *port*, *keys*, *dest_db*, *timeout*, *, *copy=False*, *replace=False*)
Atomically transfer keys from one Redis instance to another one.

Keys argument must be list/tuple of keys to migrate.

move(*key*, *db*)
Move key from currently selected database to specified destination.

Raises

- **TypeError** – if db is not int
- **ValueError** – if db is less than 0

object_encoding(*key*)
Returns the kind of internal representation used in order to store the value associated with a key (OBJECT ENCODING).

object_idletime(*key*)
Returns the number of seconds since the object is not requested by read or write operations (OBJECT IDLETIME).

object_refcount(*key*)
Returns the number of references of the value associated with the specified key (OBJECT REFCOUNT).

persist(*key*)
Remove the existing timeout on key.

pexpire(*key*, *timeout*)
Set a milliseconds timeout on key.

Raises **TypeError** – if timeout is not int

pexpireat(*key*, *timestamp*)
Set expire timestamp on key, timestamp in milliseconds.

Raises **TypeError** – if timeout is not int

pttl(*key*)
Returns time-to-live for a key, in milliseconds.

Special return values (starting with Redis 2.8):

- command returns -2 if the key does not exist.
- command returns -1 if the key exists but has no associated expire.

randomkey(**, encoding=<object object>*)
Return a random key from the currently selected database.

rename(*key*, *newkey*)
Renames key to newkey.

Raises **ValueError** – if key == newkey

renamenx(*key*, *newkey*)
Renames key to newkey only if newkey does not exist.

Raises **ValueError** – if key == newkey

restore(*key*, *ttl*, *value*)
Creates a key associated with a value that is obtained via DUMP.

scan(*cursor=0*, *match=None*, *count=None*)
Incrementally iterate the keys space.

Usage example:

```
>>> match = 'something*'
>>> cur = b'0'
>>> while cur:
...     cur, keys = await redis.scan(cur, match=match)
...     for key in keys:
...         print('Matched:', key)
```

sort(*key*, **get_patterns*, *by=None*, *offset=None*, *count=None*, *asc=None*, *alpha=False*, *store=None*)
Sort the elements in a list, set or sorted set.

touch(*key*, **keys*)
Alters the last access time of a key(s).

Returns the number of keys that were touched.

ttl (*key*)
Returns time-to-live for a key, in seconds.

Special return values (starting with Redis 2.8): * command returns -2 if the key does not exist. * command returns -1 if the key exists but has no associated expire.

type (*key*)
Returns the string representation of the value's type stored at key.

unlink (*key*, **keys*)
Delete a key asynchronously in another thread.

wait (*numslaves*, *timeout*)
Wait for the synchronous replication of all the write commands sent in the context of the current connection.

7.4.2 Geo commands

New in version v0.3.0.

class aioredis.commands.**GeoCommandsMixin**
Geo commands mixin.

For commands details see: http://redis.io/commands#geo

geoadd (*key*, *longitude*, *latitude*, *member*, **args*, ***kwargs*)
Add one or more geospatial items in the geospatial index represented using a sorted set.

Return type int

geodist (*key*, *member1*, *member2*, *unit='m'*)
Returns the distance between two members of a geospatial index.

Return type list[float or None]

geohash (*key*, *member*, **members*, ***kwargs*)
Returns members of a geospatial index as standard geohash strings.

Return type list[str or bytes or None]

geopos (*key*, *member*, **members*, ***kwargs*)
Returns longitude and latitude of members of a geospatial index.

Return type list[*GeoPoint* or None]

georadius (*key*, *longitude*, *latitude*, *radius*, *unit='m'*, ***, *with_dist=False*, *with_hash=False*, *with_coord=False*, *count=None*, *sort=None*, *encoding=<object object>*)
Query a sorted set representing a geospatial index to fetch members matching a given maximum distance from a point.

Return value follows Redis convention:

- if none of `WITH*` flags are set – list of strings returned:

```
>>> await redis.georadius('Sicily', 15, 37, 200, 'km')
[b"Palermo", b"Catania"]
```

- if any flag (or all) is set – list of named tuples returned:

```
>>> await redis.georadius('Sicily', 15, 37, 200, 'km',
...                       with_dist=True)
[GeoMember(name=b"Palermo", dist=190.4424, hash=None, coord=None),
 GeoMember(name=b"Catania", dist=56.4413, hash=None, coord=None)]
```

Raises

- **TypeError** – radius is not float or int
- **TypeError** – count is not int
- **ValueError** – if unit not equal `m`, `km`, `mi` or `ft`
- **ValueError** – if sort not equal `ASC` or `DESC`

Return type list[str] or list[*GeoMember*]

georadiusbymember(*key*, *member*, *radius*, *unit='m'*, ***, *with_dist=False*, *with_hash=False*, *with_coord=False*, *count=None*, *sort=None*, *encoding=<object object>*)

Query a sorted set representing a geospatial index to fetch members matching a given maximum distance from a member.

Return value follows Redis convention:

- if none of `WITH*` flags are set – list of strings returned:

```
>>> await redis.georadiusbymember('Sicily', 'Palermo', 200, 'km')
[b"Palermo", b"Catania"]
```

- if any flag (or all) is set – list of named tuples returned:

```
>>> await redis.georadiusbymember('Sicily', 'Palermo', 200, 'km',
...                               with_dist=True)
[GeoMember(name=b"Palermo", dist=190.4424, hash=None, coord=None),
 GeoMember(name=b"Catania", dist=56.4413, hash=None, coord=None)]
```

Raises

- **TypeError** – radius is not float or int
- **TypeError** – count is not int
- **ValueError** – if unit not equal `m`, `km`, `mi` or `ft`
- **ValueError** – if sort not equal `ASC` or `DESC`

Return type list[str] or list[*GeoMember*]

Geo commands result wrappers

class aioredis.commands.**GeoPoint**(*longitude*, *latitude*)

Bases: `tuple`

Named tuple representing result returned by `GEOPOS` and `GEORADIUS` commands.

Parameters

- **longitude** (*float*) – longitude value.
- **latitude** (*float*) – latitude value.

class aioredis.commands.**GeoMember**(*member*, *dist*, *hash*, *coord*)
Bases: tuple

Named tuple representing result returned by GEORADIUS and GEORADIUSBYMEMBER commands.

Parameters

- **member** (*str or bytes*) – Value of geo sorted set item;
- **dist** (*None or float*) – Distance in units passed to call. None if with_dist was not set in *georadius()* call.
- **hash** (*None or int*) – Geo-hash represented as number. None if with_hash was not in *georadius()* call.
- **coord** (*None or GeoPoint*) – Coordinate of geospatial index member. None if with_coord was not set in *georadius()* call.

7.4.3 Strings commands

class aioredis.commands.**StringCommandsMixin**
String commands mixin.

For commands details see: http://redis.io/commands/#string

append(*key*, *value*)
Append a value to key.

bitcount(*key*, *start=None*, *end=None*)
Count set bits in a string.

Raises **TypeError** – if only start or end specified.

bitop_and(*dest*, *key*, **keys*)
Perform bitwise AND operations between strings.

bitop_not(*dest*, *key*)
Perform bitwise NOT operations between strings.

bitop_or(*dest*, *key*, **keys*)
Perform bitwise OR operations between strings.

bitop_xor(*dest*, *key*, **keys*)
Perform bitwise XOR operations between strings.

bitpos(*key*, *bit*, *start=None*, *end=None*)
Find first bit set or clear in a string.

Raises **ValueError** – if bit is not 0 or 1

decr(*key*)
Decrement the integer value of a key by one.

decrby(*key*, *decrement*)
Decrement the integer value of a key by the given number.

Raises **TypeError** – if decrement is not int

get(*key*, ***, *encoding=<object object>*)
Get the value of a key.

getbit(*key*, *offset*)
Returns the bit value at offset in the string value stored at key.

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset is less than 0

getrange(*key*, *start*, *end*, ***, *encoding=<object object>*)
Get a substring of the string stored at a key.

Raises **TypeError** – if start or end is not int

getset(*key*, *value*, ***, *encoding=<object object>*)
Set the string value of a key and return its old value.

incr(*key*)
Increment the integer value of a key by one.

incrby(*key*, *increment*)
Increment the integer value of a key by the given amount.

Raises **TypeError** – if increment is not int

incrbyfloat(*key*, *increment*)
Increment the float value of a key by the given amount.

Raises **TypeError** – if increment is not int

mget(*key*, **keys*, *encoding=<object object>*)
Get the values of all the given keys.

mset(*key*, *value*, **pairs*)
Set multiple keys to multiple values.

Raises **TypeError** – if len of pairs is not event number

msetnx(*key*, *value*, **pairs*)
Set multiple keys to multiple values, only if none of the keys exist.

Raises **TypeError** – if len of pairs is not event number

psetex(*key*, *milliseconds*, *value*)
Set the value and expiration in milliseconds of a key.

Raises **TypeError** – if milliseconds is not int

set(*key*, *value*, ***, *expire=0*, *pexpire=0*, *exist=None*)
Set the string value of a key.

Raises **TypeError** – if expire or pexpire is not int

setbit(*key*, *offset*, *value*)
Sets or clears the bit at offset in the string value stored at key.

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset is less than 0 or value is not 0 or 1

setex(*key*, *seconds*, *value*)
Set the value and expiration of a key.

If seconds is float it will be multiplied by 1000 coerced to int and passed to *psetex* method.

Raises **TypeError** – if seconds is neither int nor float

setnx(*key*, *value*)
Set the value of a key, only if the key does not exist.

setrange(*key*, *offset*, *value*)
Overwrite part of a string at key starting at the specified offset.

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset less than 0

strlen(*key*)
Get the length of the value stored in a key.

7.4.4 Hash commands

class aioredis.commands.**HashCommandsMixin**
Hash commands mixin.

For commands details see: http://redis.io/commands#hash

hdel(*key*, *field*, **fields*)
Delete one or more hash fields.

hexists(*key*, *field*)
Determine if hash field exists.

hget(*key*, *field*, ***, *encoding=<object object>*)
Get the value of a hash field.

hgetall(*key*, ***, *encoding=<object object>*)
Get all the fields and values in a hash.

hincrby(*key*, *field*, *increment=1*)
Increment the integer value of a hash field by the given number.

hincrbyfloat(*key*, *field*, *increment=1.0*)
Increment the float value of a hash field by the given number.

hkeys(*key*, ***, *encoding=<object object>*)
Get all the fields in a hash.

hlen(*key*)
Get the number of fields in a hash.

hmget(*key*, *field*, **fields*, *encoding=<object object>*)
Get the values of all the given fields.

hmset(*key*, *field*, *value*, **pairs*)
Set multiple hash fields to multiple values.

hmset_dict(*key*, **args*, ***kwargs*)
Set multiple hash fields to multiple values.

dict can be passed as first positional argument:

```
>>> await redis.hmset_dict(
...     'key', {'field1': 'value1', 'field2': 'value2'})
```

or keyword arguments can be used:

```
>>> await redis.hmset_dict(
...     'key', field1='value1', field2='value2')
```

or dict argument can be mixed with kwargs:

```
>>> await redis.hmset_dict(
...     'key', {'field1': 'value1'}, field2='value2')
```

Note: `dict` and `kwargs` not get mixed into single dictionary, if both specified and both have same key(s) – `kwargs` will win:

```
>>> await redis.hmset_dict('key', {'foo': 'bar'}, foo='baz')
>>> await redis.hget('key', 'foo', encoding='utf-8')
'baz'
```

---

hscan (*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate hash fields and associated values.

hset (*key*, *field*, *value*)
Set the string value of a hash field.

hsetnx (*key*, *field*, *value*)
Set the value of a hash field, only if the field does not exist.

hstrlen (*key*, *field*)
Get the length of the value of a hash field.

hvals (*key*, *, *encoding=<object object>*)
Get all the values in a hash.

ihscan (*key*, *, *match=None*, *count=None*)
Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for name, val in redis.ihscan(key, match='something*'):
...     print('Matched:', name, '->', val)
```

7.4.5 List commands

class aioredis.commands.**ListCommandsMixin**
List commands mixin.

For commands details see: http://redis.io/commands#list

blpop (*key*, **keys*, *timeout=0*, *encoding=<object object>*)
Remove and get the first element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less than 0

brpop (*key*, **keys*, *timeout=0*, *encoding=<object object>*)
Remove and get the last element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less than 0

brpoplpush(*sourcekey*, *destkey*, *timeout=0*, *encoding=<object object>*)
Remove and get the last element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less than 0

lindex(*key*, *index*, ***, *encoding=<object object>*)
Get an element from a list by its index.

Raises **TypeError** – if index is not int

linsert(*key*, *pivot*, *value*, *before=False*)
Inserts value in the list stored at key either before or after the reference value pivot.

llen(*key*)
Returns the length of the list stored at key.

lpop(*key*, ***, *encoding=<object object>*)
Removes and returns the first element of the list stored at key.

lpush(*key*, *value*, **values*)
Insert all the specified values at the head of the list stored at key.

lpushx(*key*, *value*)
Inserts value at the head of the list stored at key, only if key already exists and holds a list.

lrange(*key*, *start*, *stop*, ***, *encoding=<object object>*)
Returns the specified elements of the list stored at key.

Raises **TypeError** – if start or stop is not int

lrem(*key*, *count*, *value*)
Removes the first count occurrences of elements equal to value from the list stored at key.

Raises **TypeError** – if count is not int

lset(*key*, *index*, *value*)
Sets the list element at index to value.

Raises **TypeError** – if index is not int

ltrim(*key*, *start*, *stop*)
Trim an existing list so that it will contain only the specified range of elements specified.

Raises **TypeError** – if start or stop is not int

rpop(*key*, ***, *encoding=<object object>*)
Removes and returns the last element of the list stored at key.

rpoplpush(*sourcekey*, *destkey*, ***, *encoding=<object object>*)
Atomically returns and removes the last element (tail) of the list stored at source, and pushes the element at the first element (head) of the list stored at destination.

rpush(*key*, *value*, **values*)
Insert all the specified values at the tail of the list stored at key.

rpushx(*key*, *value*)
Inserts value at the tail of the list stored at key, only if key already exists and holds a list.

7.4.6 Set commands

class aioredis.commands.**SetCommandsMixin**
Set commands mixin.

For commands details see: http://redis.io/commands#set

isscan(*key*, *, *match=None*, *count=None*)
Incrementally iterate set elements using async for.

Usage example:

```
>>> async for val in redis.isscan(key, match='something*'):
...     print('Matched:', val)
```

sadd(*key*, *member*, **members*)
Add one or more members to a set.

scard(*key*)
Get the number of members in a set.

sdiff(*key*, **keys*)
Subtract multiple sets.

sdiffstore(*destkey*, *key*, **keys*)
Subtract multiple sets and store the resulting set in a key.

sinter(*key*, **keys*)
Intersect multiple sets.

sinterstore(*destkey*, *key*, **keys*)
Intersect multiple sets and store the resulting set in a key.

sismember(*key*, *member*)
Determine if a given value is a member of a set.

smembers(*key*, *, *encoding=<object object>*)
Get all the members in a set.

smove(*sourcekey*, *destkey*, *member*)
Move a member from one set to another.

spop(*key*, *count=None*, *, *encoding=<object object>*)
Remove and return one or multiple random members from a set.

srandmember(*key*, *count=None*, *, *encoding=<object object>*)
Get one or multiple random members from a set.

srem(*key*, *member*, **members*)
Remove one or more members from a set.

sscan(*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate Set elements.

sunion(*key*, **keys*)
Add multiple sets.

sunionstore(*destkey*, *key*, **keys*)
Add multiple sets and store the resulting set in a key.

7.4.7 Sorted Set commands

class aioredis.commands.**SortedSetCommandsMixin**

Sorted Sets commands mixin.

For commands details see: http://redis.io/commands/#sorted_set

izscan(*key*, ***, *match=None*, *count=None*)

Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for val, score in redis.izscan(key, match='something*'):
...     print('Matched:', val, ':', score)
```

zadd(*key*, *score*, *member*, **pairs*, *exist=None*)

Add one or more members to a sorted set or update its score.

Raises

- **TypeError** – score not int or float
- **TypeError** – length of pairs is not even number

zcard(*key*)

Get the number of members in a sorted set.

zcount(*key*, *min=-inf*, *max=inf*, ***, *exclude=None*)

Count the members in a sorted set with scores within the given values.

Raises

- **TypeError** – min or max is not float or int
- **ValueError** – if min greater than max

zincrby(*key*, *increment*, *member*)

Increment the score of a member in a sorted set.

Raises **TypeError** – increment is not float or int

zinterstore(*destkey*, *key*, **keys*, *with_weights=False*, *aggregate=None*)

Intersect multiple sorted sets and store result in a new key.

Parameters **with_weights** (*bool*) – when set to true each key must be a tuple in form of (key, weight)

zlexcount(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*)

Count the number of members in a sorted set between a given lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes

zrange(*key*, *start=0*, *stop=-1*, *withscores=False*, *encoding=<object object>*)

Return a range of members in a sorted set, by index.

Raises

- **TypeError** – if start is not int
- **TypeError** – if stop is not int

zrangebylex(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*, *offset=None*, *count=None*, *encoding=<object object>*)
Return a range of members in a sorted set, by lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not bytes
- **TypeError** – if count is not bytes

zrangebyscore(*key*, *min=-inf*, *max=inf*, *withscores=False*, *offset=None*, *count=None*, ***, *exclude=None*, *encoding=<object object>*)
Return a range of members in a sorted set, by score.

Raises

- **TypeError** – if min or max is not float or int
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not int
- **TypeError** – if count is not int

zrank(*key*, *member*)
Determine the index of a member in a sorted set.

zrem(*key*, *member*, **members*)
Remove one or more members from a sorted set.

zremrangebylex(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*)
Remove all members in a sorted set between the given lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes

zremrangebyrank(*key*, *start*, *stop*)
Remove all members in a sorted set within the given indexes.

Raises

- **TypeError** – if start is not int
- **TypeError** – if stop is not int

zremrangebyscore(*key*, *min=-inf*, *max=inf*, ***, *exclude=None*)
Remove all members in a sorted set within the given scores.

Raises **TypeError** – if min or max is not int or float

zrevrange(*key*, *start*, *stop*, *withscores=False*, *encoding=<object object>*)
Return a range of members in a sorted set, by index, with scores ordered from high to low.

Raises **TypeError** – if start or stop is not int

zrevrangebylex(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*, *offset=None*, *count=None*, *encoding=<object object>*)
Return a range of members in a sorted set, by lexicographical range from high to low.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not bytes
- **TypeError** – if count is not bytes

zrevrangebyscore(*key*, *max=inf*, *min=-inf*, ***, *exclude=None*, *withscores=False*, *offset=None*, *count=None*, *encoding=<object object>*)
Return a range of members in a sorted set, by score, with scores ordered from high to low.

Raises

- **TypeError** – if min or max is not float or int
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not int
- **TypeError** – if count is not int

zrevrank(*key*, *member*)
Determine the index of a member in a sorted set, with scores ordered from high to low.

zscan(*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate sorted sets elements and associated scores.

zscore(*key*, *member*)
Get the score associated with the given member in a sorted set.

zunionstore(*destkey*, *key*, **keys*, *with_weights=False*, *aggregate=None*)
Add multiple sorted sets and store result in a new key.

7.4.8 Server commands

class aioredis.commands.**ServerCommandsMixin**
Server commands mixin.

For commands details see: http://redis.io/commands/#server

bgrewriteaof()
Asynchronously rewrite the append-only file.

bgsave()
Asynchronously save the dataset to disk.

client_getname(*encoding=<object object>*)
Get the current connection name.

client_kill()
Kill the connection of a client.

> **Warning:** Not Implemented

client_list()
Get the list of client connections.

Returns list of ClientInfo named tuples.

client_pause(*timeout*)
Stop processing commands from clients for *timeout* milliseconds.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less than 0

client_setname(*name*)
Set the current connection name.

command()
Get array of Redis commands.

command_count()
Get total number of Redis commands.

command_getkeys(*command*, **args*, *encoding='utf-8'*)
Extract keys given a full Redis command.

command_info(*command*, **commands*)
Get array of specific Redis command details.

config_get(*parameter='*'*)
Get the value of a configuration parameter(s).

If called without argument will return all parameters.

Raises **TypeError** – if parameter is not string

config_resetstat()
Reset the stats returned by INFO.

config_rewrite()
Rewrite the configuration file with the in memory configuration.

config_set(*parameter*, *value*)
Set a configuration parameter to the given value.

dbsize()
Return the number of keys in the selected database.

debug_object(*key*)
Get debugging information about a key.

debug_segfault(*key*)
Make the server crash.

debug_sleep(*timeout*)
Suspend connection for timeout seconds.

flushall(*async_op=False*)
Remove all keys from all databases.

Parameters **async_op** – lets the entire dataset to be freed asynchronously. Defaults to False

flushdb(*async_op=False*)
Remove all keys from the current database.

Parameters **async_op** – lets a single database to be freed asynchronously. Defaults to False

info(*section='default'*)
Get information and statistics about the server.

If called without argument will return default set of sections. For available sections, see http://redis.io/commands/INFO

Raises **ValueError** – if section is invalid

lastsave()
Get the UNIX time stamp of the last successful save to disk.

monitor()
Listen for all requests received by the server in real time.

> **Warning:** Will not be implemented for now.

role()
Return the role of the server instance.

Returns named tuples describing role of the instance. For fields information see http://redis.io/commands/role#output-format

save()
Synchronously save the dataset to disk.

shutdown(*save=None*)
Synchronously save the dataset to disk and then shut down the server.

slaveof(*host=<object object>*, *port=None*)
Make the server a slave of another instance, or promote it as master.

Calling `slaveof(None)` will send `SLAVEOF NO ONE`.

Changed in version v0.2.6: `slaveof()` form deprecated in favour of explicit `slaveof(None)`.

slowlog_get(*length=None*)
Returns the Redis slow queries log.

slowlog_len()
Returns length of Redis slow queries log.

slowlog_reset()
Resets Redis slow queries log.

sync()
Redis-server internal command used for replication.

time()
Return current server time.

7.4.9 HyperLogLog commands

class aioredis.commands.**HyperLogLogCommandsMixin**
HyperLogLog commands mixin.

For commands details see: http://redis.io/commands#hyperloglog

pfadd(*key*, *value*, **values*)
Adds the specified elements to the specified HyperLogLog.

pfcount (*key*, **keys*)
Return the approximated cardinality of the set(s) observed by the HyperLogLog at key(s).

pfmerge (*destkey*, *sourcekey*, **sourcekeys*)
Merge N different HyperLogLogs into a single one.

7.4.10 Transaction commands

class aioredis.commands.**TransactionsCommandsMixin**
Transaction commands mixin.

For commands details see: http://redis.io/commands/#transactions

Transactions HOWTO:

```
>>> tr = redis.multi_exec()
>>> result_future1 = tr.incr('foo')
>>> result_future2 = tr.incr('bar')
>>> try:
...     result = await tr.execute()
... except MultiExecError:
...     pass    # check what happened
>>> result1 = await result_future1
>>> result2 = await result_future2
>>> assert result == [result1, result2]
```

multi_exec()
Returns MULTI/EXEC pipeline wrapper.

Usage:

```
>>> tr = redis.multi_exec()
>>> fut1 = tr.incr('foo')   # NO `await` as it will block forever!
>>> fut2 = tr.incr('bar')
>>> result = await tr.execute()
>>> result
[1, 1]
>>> await asyncio.gather(fut1, fut2)
[1, 1]
```

pipeline()
Returns *Pipeline* object to execute bulk of commands.

It is provided for convenience. Commands can be pipelined without it.

Example:

```
>>> pipe = redis.pipeline()
>>> fut1 = pipe.incr('foo') # NO `await` as it will block forever!
>>> fut2 = pipe.incr('bar')
>>> result = await pipe.execute()
>>> result
[1, 1]
>>> await asyncio.gather(fut1, fut2)
[1, 1]
>>> #
>>> # The same can be done without pipeline:
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> fut1 = redis.incr('foo')     # the 'INCRY foo' command already sent
>>> fut2 = redis.incr('bar')
>>> await asyncio.gather(fut1, fut2)
[2, 2]
```

unwatch()
Forget about all watched keys.

watch(*key*, **keys*)
Watch the given keys to determine execution of the MULTI/EXEC block.

class aioredis.commands.**Pipeline**(*connection*, *commands_factory=lambda conn: conn*, ***, *loop=None*)
Commands pipeline.

Buffers commands for execution in bulk.

This class implements *__getattr__* method allowing to call methods on instance created with `commands_factory`.

Parameters

- **connection** (*aioredis.RedisConnection*) – Redis connection
- **commands_factory** (*callable*) – Commands factory to get methods from.
- **loop** (EventLoop) – An optional *event loop* instance (uses *asyncio.get_event_loop()* if not specified).

coroutine execute(***, *return_exceptions=False*)
Executes all buffered commands and returns result.

Any exception that is raised by any command is caught and raised later when processing results.

If `return_exceptions` is set to `True` then all collected errors are returned in resulting list otherwise single *aioredis.PipelineError* exception is raised (containing all collected errors).

Parameters **return_exceptions** (*bool*) – Raise or return exceptions.

Raises ***aioredis.PipelineError*** – Raised when any command caused error.

class aioredis.commands.**MultiExec**(*connection*, *commands_factory=lambda conn: conn*, ***, *loop=None*)
Bases: *Pipeline*.

Multi/Exec pipeline wrapper.

See *Pipeline* for parameters description.

coroutine execute(***, *return_exceptions=False*)
Executes all buffered commands and returns result.

see *Pipeline.execute()* for details.

Parameters **return_exceptions** (*bool*) – Raise or return exceptions.

Raises

- ***aioredis.MultiExecError*** – Raised instead of *aioredis.PipelineError*
- ***aioredis.WatchVariableError*** – If watched variable is changed

7.4.11 Scripting commands

class aioredis.commands.**ScriptingCommandsMixin**
Set commands mixin.

For commands details see: http://redis.io/commands#scripting

eval(*script*, *keys=[]*, *args=[]*)
Execute a Lua script server side.

evalsha(*digest*, *keys=[]*, *args=[]*)
Execute a Lua script server side by its SHA1 digest.

script_exists(*digest*, **digests*)
Check existence of scripts in the script cache.

script_flush()
Remove all the scripts from the script cache.

script_kill()
Kill the script currently in execution.

script_load(*script*)
Load the specified Lua script into the script cache.

7.4.12 Server commands

class aioredis.commands.**ServerCommandsMixin**
Server commands mixin.

For commands details see: http://redis.io/commands/#server

bgrewriteaof()
Asynchronously rewrite the append-only file.

bgsave()
Asynchronously save the dataset to disk.

client_getname(*encoding=<object object>*)
Get the current connection name.

client_kill()
Kill the connection of a client.

> **Warning:** Not Implemented

client_list()
Get the list of client connections.

Returns list of ClientInfo named tuples.

client_pause(*timeout*)
Stop processing commands from clients for *timeout* milliseconds.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less than 0

client_setname(*name*)
Set the current connection name.

command()
Get array of Redis commands.

command_count()
Get total number of Redis commands.

command_getkeys(*command*, **args*, *encoding='utf-8'*)
Extract keys given a full Redis command.

command_info(*command*, **commands*)
Get array of specific Redis command details.

config_get(*parameter='*'*)
Get the value of a configuration parameter(s).

If called without argument will return all parameters.

Raises **TypeError** – if parameter is not string

config_resetstat()
Reset the stats returned by INFO.

config_rewrite()
Rewrite the configuration file with the in memory configuration.

config_set(*parameter*, *value*)
Set a configuration parameter to the given value.

dbsize()
Return the number of keys in the selected database.

debug_object(*key*)
Get debugging information about a key.

debug_segfault(*key*)
Make the server crash.

debug_sleep(*timeout*)
Suspend connection for timeout seconds.

flushall(*async_op=False*)
Remove all keys from all databases.

Parameters **async_op** – lets the entire dataset to be freed asynchronously. Defaults to False

flushdb(*async_op=False*)
Remove all keys from the current database.

Parameters **async_op** – lets a single database to be freed asynchronously. Defaults to False

info(*section='default'*)
Get information and statistics about the server.

If called without argument will return default set of sections. For available sections, see http://redis.io/commands/INFO

Raises **ValueError** – if section is invalid

lastsave()
Get the UNIX time stamp of the last successful save to disk.

monitor()
Listen for all requests received by the server in real time.

> **Warning:** Will not be implemented for now.

role()
Return the role of the server instance.

Returns named tuples describing role of the instance. For fields information see http://redis.io/commands/role#output-format

save()
Synchronously save the dataset to disk.

shutdown(*save=None*)
Synchronously save the dataset to disk and then shut down the server.

slaveof(*host=<object object>*, *port=None*)
Make the server a slave of another instance, or promote it as master.

Calling `slaveof(None)` will send `SLAVEOF NO ONE`.

Changed in version v0.2.6: `slaveof()` form deprecated in favour of explicit `slaveof(None)`.

slowlog_get(*length=None*)
Returns the Redis slow queries log.

slowlog_len()
Returns length of Redis slow queries log.

slowlog_reset()
Resets Redis slow queries log.

sync()
Redis-server internal command used for replication.

time()
Return current server time.

7.4.13 Pub/Sub commands

Also see *aioredis.Channel*.

class aioredis.commands.**PubSubCommandsMixin**
Pub/Sub commands mixin.

For commands details see: http://redis.io/commands/#pubsub

channels
Returns read-only channels dict.

See *pubsub_channels*

in_pubsub
Indicates that connection is in PUB/SUB mode.

Provides the number of subscribed channels.

patterns
Returns read-only patterns dict.

See *pubsub_patterns*

psubscribe(*pattern*, **patterns*)
Switch connection to Pub/Sub mode and subscribe to specified patterns.

Arguments can be instances of *Channel*.

Returns asyncio.gather() coroutine which when done will return a list of subscribed *Channel* objects with is_pattern property set to True.

publish(*channel*, *message*)
Post a message to channel.

publish_json(*channel*, *obj*)
Post a JSON-encoded message to channel.

pubsub_channels(*pattern=None*)
Lists the currently active channels.

pubsub_numpat()
Returns the number of subscriptions to patterns.

pubsub_numsub(**channels*)
Returns the number of subscribers for the specified channels.

punsubscribe(*pattern*, **patterns*)
Unsubscribe from specific patterns.

Arguments can be instances of *Channel*.

subscribe(*channel*, **channels*)
Switch connection to Pub/Sub mode and subscribe to specified channels.

Arguments can be instances of *Channel*.

Returns asyncio.gather() coroutine which when done will return a list of *Channel* objects.

unsubscribe(*channel*, **channels*)
Unsubscribe from specific channels.

Arguments can be instances of *Channel*.

7.4.14 Cluster commands

Warning: Current release (1.2.0) of the library **does not support** Redis Cluster in a full manner. It provides only several API methods which may be changed in future.

7.4.15 Streams commands

class aioredis.commands.**StreamCommandsMixin**
Stream commands mixin

Streams are under development in Redis and not currently released.

xack(*stream*, *group_name*, *id*, **ids*)
Acknowledge a message for a given consumer group

xadd(*stream*, *fields*, *message_id=b'*'*, *max_len=None*, *exact_len=False*)
Add a message to a stream.

xclaim(*stream*, *group_name*, *consumer_name*, *min_idle_time*, *id*, **ids*)
Claim a message for a given consumer

xgroup_create(*stream*, *group_name*, *latest_id='$'*)
Create a consumer group

xgroup_delconsumer(*stream*, *group_name*, *consumer_name*)
Delete a specific consumer from a group

xgroup_destroy(*stream*, *group_name*)
Delete a consumer group

xgroup_setid(*stream*, *group_name*, *latest_id='$'*)
Set the latest ID for a consumer group

xinfo(*stream*)
Retrieve information about the given stream.

An alias for `xinfo_stream()`

xinfo_consumers(*stream*, *group_name*)
Retrieve consumers of a consumer group

xinfo_groups(*stream*)
Retrieve the consumer groups for a stream

xinfo_help()
Retrieve help regarding the `XINFO` sub-commands

xinfo_stream(*stream*)
Retrieve information about the given stream.

xpending(*stream*, *group_name*, *start=None*, *stop=None*, *count=None*, *consumer=None*)
Get information on pending messages for a stream

Returned data will vary depending on the presence (or not) of the start/stop/count parameters. For more details see: https://redis.io/commands/xpending

Raises **ValueError** – if the start/stop/count parameters are only partially specified

xrange(*stream*, *start='-'*, *stop='+'*, *count=None*)
Retrieve messages from a stream.

xread(*streams*, *timeout=0*, *count=None*, *latest_ids=None*)
Perform a blocking read on the given stream

Raises **ValueError** – if the length of streams and latest_ids do not match

xread_group(*group_name*, *consumer_name*, *streams*, *timeout=0*, *count=None*, *latest_ids=None*)
Perform a blocking read on the given stream as part of a consumer group

Raises **ValueError** – if the length of streams and latest_ids do not match

xrevrange(*stream*, *start='+'*, *stop='-'*, *count=None*)
Retrieve messages from a stream in reverse order.

7.5 aioredis.abc — Interfaces Reference

This module defines several abstract classes that must be used when implementing custom connection managers or other features.

class aioredis.abc.**AbcConnection**
Bases: abc.ABC

Abstract connection interface.

address
Connection address.

close()
Perform connection(s) close and resources cleanup.

closed
Flag indicating if connection is closing or already closed.

db
Current selected DB index.

encoding
Current set connection codec.

execute(*command*, **args*, ***kwargs*)
Execute redis command.

execute_pubsub(*command*, **args*, ***kwargs*)
Execute Redis (p)subscribe/(p)unsubscribe commands.

in_pubsub
Returns number of subscribed channels.

Can be tested as bool indicating Pub/Sub mode state.

pubsub_channels
Read-only channels dict.

pubsub_patterns
Read-only patterns dict.

coroutine wait_closed()
Coroutine waiting until all resources are closed/released/cleaned up.

class aioredis.abc.**AbcPool**
Bases: *aioredis.abc.AbcConnection*

Abstract connections pool interface.

Inherited from AbcConnection so both have common interface for executing Redis commands.

coroutine acquire()
Acquires connection from pool.

address
Connection address or None.

get_connection()
Gets free connection from pool in a sync way.

If no connection available — returns None.

release(*conn*)
Releases connection to pool.

Parameters **conn** (AbcConnection) – Owned connection to be released.

class aioredis.abc.**AbcChannel**
Bases: abc.ABC

Abstract Pub/Sub Channel interface.

close(*exc=None*)
Marks Channel as closed, no more messages will be sent to it.

Called by RedisConnection when channel is unsubscribed or connection is closed.

coroutine get()
Wait and return new message.

Will raise `ChannelClosedError` if channel is not active.

is_active
Flag indicating that channel has unreceived messages and not marked as closed.

is_pattern
Boolean flag indicating if channel is pattern channel.

name
Encoded channel name or pattern.

put_nowait(*data*)
Send data to channel.

Called by RedisConnection when new message received. For pattern subscriptions data will be a tuple of channel name and message itself.

7.6 aioredis.pubsub — Pub/Sub Tools Reference

Module provides a Pub/Sub listener interface implementing multi-producers, single-consumer queue pattern.

class aioredis.pubsub.**Receiver**(*loop=None*, *on_close=None*)
Multi-producers, single-consumer Pub/Sub queue.

Can be used in cases where a single consumer task must read messages from several different channels (where pattern subscriptions may not work well or channels can be added/removed dynamically).

Example use case:

```
>>> from aioredis.pubsub import Receiver
>>> from aioredis.abc import AbcChannel
>>> mpsc = Receiver(loop=loop)
>>> async def reader(mpsc):
...     async for channel, msg in mpsc.iter():
...         assert isinstance(channel, AbcChannel)
...         print("Got {!r} in channel {!r}".format(msg, channel))
>>> asyncio.ensure_future(reader(mpsc))
>>> await redis.subscribe(mpsc.channel('channel:1'),
...                       mpsc.channel('channel:3'))
...                       mpsc.channel('channel:5'))
>>> await redis.psubscribe(mpsc.pattern('hello'))
>>> # publishing 'Hello world' into 'hello-channel'
>>> # will print this message:
Got b'Hello world' in channel b'hello-channel'
>>> # when all is done:
>>> await redis.unsubscribe('channel:1', 'channel:3', 'channel:5')
>>> await redis.punsubscribe('hello')
>>> mpsc.stop()
>>> # any message received after stop() will be ignored.
```

Warning: Currently subscriptions implementation has few issues that will be solved eventually, but until then developer must be aware of the following:

- Single `Receiver` instance can not be shared between two (or more) connections (or client instances) because any of them can close `_Sender`.
- Several `Receiver` instances can not subscribe to the same channel or pattern. This is a flaw in subscription mode implementation: subsequent subscriptions to some channel always return first-created Channel object.

channel(*name*)
Create a channel.

Returns `_Sender` object implementing *AbcChannel*.

channels
Read-only channels dict.

check_stop(*channel*, *exc=None*)
TBD

coroutine get(*, *encoding=None*, *decoder=None*)
Wait for and return pub/sub message from one of channels.

Return value is either:

- tuple of two elements: channel & message;
- tuple of three elements: pattern channel, (target channel & message);
- or None in case Receiver is not active or has just been stopped.

Raises ***aioredis.ChannelClosedError*** – If listener is stopped and all messages have been received.

is_active
Returns True if listener has any active subscription.

iter(*, *encoding=None*, *decoder=None*)
Returns async iterator.

Usage example:

```
>>> async for ch, msg in mpsc.iter():
...     print(ch, msg)
```

pattern(*pattern*)
Create a pattern channel.

Returns `_Sender` object implementing *AbcChannel*.

patterns
Read-only patterns dict.

stop()
Stop receiving messages.

All new messages after this call will be ignored, so you must call unsubscribe before stopping this listener.

coroutine wait_message()
Blocks until new message appear.

class aioredis.pubsub.**_Sender**(*receiver*, *name*, *is_pattern*)
Write-Only Channel.

Does not allow direct `.get()` calls.

Bases: *aioredis.abc.AbcChannel*

Not to be used directly, returned by *Receiver.channel()* or *Receiver.pattern()* calls.

7.7 aioredis.sentinel — Sentinel Client Reference

This section contains reference for Redis Sentinel client.

Sample usage:

```
import aioredis

sentinel = await aioredis.create_sentinel(
   [('sentinel.host1', 26379), ('sentinel.host2', 26379)])

redis = sentinel.master_for('mymaster')
assert await redis.set('key', 'value')
assert await redis.get('key', encoding='utf-8') == 'value'

# redis client will reconnect/reconfigure automatically
#  by sentinel client instance
```

7.7.1 RedisSentinel

coroutine aioredis.sentinel.**create_sentinel**(*sentinels*, ***, *db=None*, *password=None*, *encoding=None*, *minsize=1*, *maxsize=10*, *ssl=None*, *parser=None*, *loop=None*)
Creates Redis Sentinel client.

Parameters

- **sentinels** (*list[tuple]*) – A list of Sentinel node addresses.
- **db** (*int*) – Redis database index to select for every master/slave connections.
- **password** (*str or bytes or None*) – Password to use if Redis server instance requires authorization.
- **encoding** (*str or None*) – Codec to use for response decoding.
- **minsize** (*int*) – Minimum number of connections (to master or slave) to initialize and keep in pool. Default is 1.
- **maxsize** (*int*) – Maximum number of connections (to master or slave) that can be created in pool. Default is 10.
- **ssl** (`ssl.SSLContext` or True or None) – SSL context that is passed through to `asyncio.BaseEventLoop.create_connection()`.
- **parser** (*callable or None*) – Protocol parser class. Can be used to set custom protocol reader; expected same interface as `hiredis.Reader`.
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).

Return type *RedisSentinel*

class aioredis.sentinel.**RedisSentinel**

Redis Sentinel client.

The class provides interface to Redis Sentinel commands as well as few methods to acquire managed Redis clients, see below.

closed

`True` if client is closed.

master_for(*name*)

Get *Redis* client to named master. The client is instantiated with special connections pool which is controlled by *SentinelPool*. **This method is not a coroutine.**

Parameters **name** (*str*) – Service name.

Return type *aioredis.Redis*

slave_for(*name*)

Get *Redis* client to named slave. The client is instantiated with special connections pool which is controlled by *SentinelPool*. **This method is not a coroutine.**

Parameters **name** (*str*) – Service name.

Return type *aioredis.Redis*

execute(*command*, **args*, ***kwargs*)

Execute Sentinel command. Every command is prefixed with `SENTINEL` automatically.

Return type asyncio.Future

coroutine ping()

Send PING to Sentinel instance. Currently the ping command will be sent to first sentinel in pool, this may change in future.

master(*name*)

Returns a dictionary containing the specified master's state. Please refer to Redis documentation for more info on returned data.

Return type asyncio.Future

master_address(*name*)

Returns a `(host, port)` pair for the given service name.

Return type asyncio.Future

masters()

Returns a list of dictionaries containing all masters' states.

Return type asyncio.Future

slaves(*name*)

Returns a list of slaves for the given service name.

Return type asyncio.Future

sentinels(*name*)

Returns a list of Sentinels for the given service name.

Return type asyncio.Future

monitor(*name*, *ip*, *port*, *quorum*)

Add a new master to be monitored by this Sentinel.

Parameters

- **name** (*str*) – Service name.
- **ip** (*str*) – New node's IP address.
- **port** (*int*) – Node's TCP port.
- **quorum** (*int*) – Sentinel quorum.

remove(*name*)
Remove a master from Sentinel's monitoring.

Parameters **name** (*str*) – Service name

set(*name*, *option*, *value*)
Set Sentinel monitoring parameter for a given master. Please refer to Redis documentation for more info on options.

Parameters

- **name** (*str*) – Master's name.
- **option** (*str*) – Monitoring option name.
- **value** (*str*) – Monitoring option value.

failover(*name*)
Force a failover of a named master.

Parameters **name** (*str*) – Master's name.

check_quorum(*name*)
Check if the current Sentinel configuration is able to reach the quorum needed to failover a master, and the majority needed to authorize the failover.

Parameters **name** (*str*) – Master's name.

close()
Close all opened connections.

coroutine wait_closed()
Wait until all connections are closed.

7.7.2 SentinelPool

Warning: This API has not yet stabilized and may change in future releases.

coroutine aioredis.sentinel.**create_sentinel_pool**(*sentinels*, ***, *db=None*, *password=None*, *encoding=None*, *minsize=1*, *maxsize=10*, *ssl=None*, *parser=None*, *loop=None*)
Creates Sentinel connections pool.

class aioredis.sentinel.**SentinelPool**
Sentinel connections pool.

This pool manages both sentinel connections and Redis master/slave connections.

closed
`True` if pool and all connections are closed.

master_for(*name*)
Returns a managed connections pool for requested service name.

Parameters **name** (*str*) – Service name.

Return type ManagedPool

slave_for(*name*)
Returns a managed connections pool for requested service name.

Parameters **name** (*str*) – Service name.

Return type ManagedPool

execute(*command*, **args*, ***kwargs*)
Execute Sentinel command.

coroutine discover(*timeout=0.2*)
Discover Sentinels and all monitored services within given timeout.

This will reset internal state of this pool.

coroutine discover_master(*service*, *timeout*)
Perform named master discovery.

Parameters

- **service** (*str*) – Service name.
- **timeout** (*float*) – Operation timeout

Return type *aioredis.RedisConnection*

coroutine discover_slave(*service*, *timeout*)
Perform slave discovery.

Parameters

- **service** (*str*) – Service name.
- **timeout** (*float*) – Operation timeout

Return type *aioredis.RedisConnection*

close()
Close all controlled connections (both to sentinel and redis).

coroutine wait_closed()
Wait until pool gets closed.

7.8 Examples of aioredis usage

Below is a list of examples from aioredis/examples (see for more).

Every example is a correct python program that can be executed.

7.8.1 Low-level connection usage example

```
get source code
```

```
import asyncio
import aioredis


async def main():
    conn = await aioredis.create_connection(
        'redis://localhost', encoding='utf-8')

    ok = await conn.execute('set', 'my-key', 'some value')
    assert ok == 'OK', ok

    str_value = await conn.execute('get', 'my-key')
    raw_value = await conn.execute('get', 'my-key', encoding=None)
    assert str_value == 'some value'
    assert raw_value == b'some value'

    print('str value:', str_value)
    print('raw value:', raw_value)

    # optionally close connection
    conn.close()
    await conn.wait_closed()


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

7.8.2 Connections pool example

get source code

```
import asyncio
import aioredis


async def main():
    pool = await aioredis.create_pool(
        'redis://localhost',
        minsize=5, maxsize=10)
    with await pool as conn:    # low-level redis connection
        await conn.execute('set', 'my-key', 'value')
        val = await conn.execute('get', 'my-key')
    print('raw value:', val)
    pool.close()
    await pool.wait_closed()    # closing all open connections


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

7.8.3 Commands example

get source code

```
import asyncio
import aioredis


async def main():
    # Redis client bound to single connection (no auto reconnection).
    redis = await aioredis.create_redis(
        'redis://localhost')
    await redis.set('my-key', 'value')
    val = await redis.get('my-key')
    print(val)

    # gracefully closing underlying connection
    redis.close()
    await redis.wait_closed()


async def redis_pool():
    # Redis client bound to pool of connections (auto-reconnecting).
    redis = await aioredis.create_redis_pool(
        'redis://localhost')
    await redis.set('my-key', 'value')
    val = await redis.get('my-key')
    print(val)

    # gracefully closing underlying connection
    redis.close()
    await redis.wait_closed()


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
    asyncio.get_event_loop().run_until_complete(redis_pool())
```

7.8.4 Transaction example

get source code

```
import asyncio
import aioredis


async def main():
    redis = await aioredis.create_redis(
        'redis://localhost')
    await redis.delete('foo', 'bar')
    tr = redis.multi_exec()
    fut1 = tr.incr('foo')
    fut2 = tr.incr('bar')
    res = await tr.execute()
    res2 = await asyncio.gather(fut1, fut2)
    print(res)
    assert res == res2

    redis.close()
    await redis.wait_closed()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

7.8.5 Pub/Sub example

get source code

```
import asyncio
import aioredis


async def reader(ch):
    while (await ch.wait_message()):
        msg = await ch.get_json()
        print("Got Message:", msg)


async def main():
    pub = await aioredis.create_redis(
        'redis://localhost')
    sub = await aioredis.create_redis(
        'redis://localhost')
    res = await sub.subscribe('chan:1')
    ch1 = res[0]

    tsk = asyncio.ensure_future(reader(ch1))

    res = await pub.publish_json('chan:1', ["Hello", "world"])
    assert res == 1

    await sub.unsubscribe('chan:1')
    await tsk
    sub.close()
    pub.close()


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

7.8.6 Scan command example

get source code

```
import asyncio
import aioredis


async def main():
    """Scan command example."""
    redis = await aioredis.create_redis(
        'redis://localhost')
```

(continues on next page)

(continued from previous page)

```
    await redis.mset('key:1', 'value1', 'key:2', 'value2')
    cur = b'0'  # set initial cursor to 0
    while cur:
        cur, keys = await redis.scan(cur, match='key:*')
        print("Iteration results:", keys)

    redis.close()
    await redis.wait_closed()


if __name__ == '__main__':
    import os
    if 'redis_version:2.6' not in os.environ.get('REDIS_VERSION', ''):
        asyncio.get_event_loop().run_until_complete(main())
```

7.8.7 Sentinel client

get source code

```
import asyncio
import aioredis


async def main():
    sentinel_client = await aioredis.create_sentinel(
        [('localhost', 26379)])

    master_redis = sentinel_client.master_for('mymaster')
    info = await master_redis.role()
    print("Master role:", info)
    assert info.role == 'master'

    sentinel_client.close()
    await sentinel_client.wait_closed()


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

7.9 Contributing

To start contributing you must read all the following.

First you must fork/clone repo from github:

```
$ git clone git@github.com:aio-libs/aioredis.git
```

Next, you should install all python dependencies, it is as easy as running single command:

```
$ make devel
```

this command will install:

- `sphinx` for building documentation;
- `pytest` for running tests;
- `flake8` for code linting;
- and few other packages.

7.9.1 Code style

Code **must** be pep8 compliant.

You can check it with following command:

```
$ make flake
```

7.9.2 Running tests

You can run tests in any of the following ways:

```
# will run tests in a verbose mode
$ make test
# or
$ py.test

# will run tests with coverage report
$ make cov
# or
$ py.test --cov
```

SSL tests

Running SSL tests requires following additional programs to be installed:

- `openssl` – to generate test key and certificate;
- `socat` – to make SSL proxy;

To install these on Ubuntu and generate test key & certificate run:

```
$ sudo apt-get install socat openssl
$ make certificate
```

Different Redis server versions

To run tests against different redises use `--redis-server` command line option:

```
$ py.test --redis-server=/path/to/custom/redis-server
```

UVLoop

To run tests with *uvloop*:

```
$ pip install uvloop
$ py.test --uvloop
```

Note: Until Python 3.5.2 EventLoop has no `create_future` method so aioredis won't benefit from uvloop's futures.

7.9.3 Writing tests

aioredis uses *pytest* tool.

Tests are located under `/tests` directory.

Pure Python 3.5 tests (ie the ones using `async`/`await` syntax) must be prefixed with `py35_`, for instance see:

```
tests/py35_generic_commands_tests.py
tests/py35_pool_test.py
```

Fixtures

There is a number of fixtures that can be used to write tests:

loop
Current event loop used for test. This is a function-scope fixture. Using this fixture will always create new event loop and set global one to None.

```
def test_with_loop(loop):
    @asyncio.coroutine
    def do_something():
        pass
    loop.run_until_complete(do_something())
```

unused_port()
Finds and returns free TCP port.

```
def test_bind(unused_port):
    port = unused_port()
    assert 1024 < port <= 65535
```

coroutine create_connection(**args*, ***kw*)
Wrapper around *aioredis.create_connection()*. Only difference is that it registers connection to be closed after test case, so you should not be worried about unclosed connections.

coroutine create_redis(**args*, ***kw*)
Wrapper around *aioredis.create_redis()*.

coroutine create_pool(**args*, ***kw*)
Wrapper around *aioredis.create_pool()*.

redis
Redis client instance.

pool
RedisPool instance.

server
Redis server instance info. Namedtuple with following properties:

name server instance name.

port Bind port.

unixsocket Bind unixsocket path.

version Redis server version tuple.

serverB
Second predefined Redis server instance info.

start_server(*name*)
Start Redis server instance. Redis instances are cached by name.

Returns server info tuple, see *server*.

Return type tuple

ssl_proxy(*unsecure_port*)
Start SSL proxy.

Parameters **unsecure_port** (*int*) – Redis server instance port

Returns secure_port and ssl_context pair

Return type tuple

Helpers

aioredis also updates *pytest*'s namespace with several helpers.

pytest.**redis_version**(**version*, *reason*)
Marks test with minimum redis version to run.

Example:

```
@pytest.redis_version(3, 2, 0, reason="HSTRLEN new in redis 3.2.0")
def test_hstrlen(redis):
    pass
```

pytest.**logs**(*logger*, *level=None*)
Adopted version of `unittest.TestCase.assertEqual()`, see it for details.

Example:

```
def test_logs(create_connection, server):
    with pytest.logs('aioredis', 'DEBUG') as cm:
        conn yield from create_connection(server.tcp_address)
    assert cm.output[0].startswith(
      'DEBUG:aioredis:Creating tcp connection')
```

pytest.**assert_almost_equal**(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Adopted version of `unittest.TestCase.assertAlmostEqual()`.

pytest.**raises_regex**(*exc_type*, *message*)
Adopted version of `unittest.TestCase.assertRaisesRegex()`.

7.10 Releases

7.10.1 1.2.0 (2018-10-24)

NEW:

- Implemented new Stream command support (see #299);
- Reduce `encode_command()` cost about 60% (see #397);

FIX:

- Fix pipeline commands buffering was causing multiple `sendto` syscalls (see #464 and #473);
- Python 3.7 compatibility fixes (see #426);
- Fix typos in documentation (see #400);
- Fix `INFO` command result parsing (see #405);
- Fix bug in `ConnectionsPool._drop_closed` method (see #461);

MISC:

- Update dependencies versions;
- Multiple tests improvements;

7.10.2 1.1.0 (2018-02-16)

NEW:

- Implement new commands: `wait`, `touch`, `swapdb`, `unlink` (see #376);
- Add `async_op` argument to `flushall` and `flushdb` commands (see #364, and #370);

FIX:

- **Important!** Fix Sentinel sentinel client with pool `minsize` greater than 1 (see #380);
- Fix `SentinelPool.discover_timeout` usage (see #379);
- Fix `Receiver` hang on disconnect (see #354, and #366);
- Fix an issue with `subscribe`/`psubscribe` with empty pool (see #351, and #355);
- Fix an issue when `StreamReader`'s feed_data is called before set_parser (see #347);

MISC:

- Update dependencies versions;
- Multiple test fixes;

7.10.3 1.0.0 (2017-11-17)

NEW:

- **Important!** Drop Python 3.3, 3.4 support; (see #321, #323 and #326);
- **Important!** Connections pool has been refactored; now `create_redis` function will yield `Redis` instance instead of `RedisPool` (see #129);

- **Important!** Change sorted set commands reply format: return list of tuples instead of plain list for commands accepting `withscores` argument (see #334);
- **Important!** Change `hscan` command reply format: return list of tuples instead of mixed key-value list (see #335);
- Implement Redis URI support as supported `address` argument value (see #322);
- Dropped `create_reconnecting_redis`, `create_redis_pool` should be used instead;
- Implement custom `StreamReader` (see #273);
- Implement Sentinel support (see #181);
- Implement pure-python parser (see #212);
- Add `migrate_keys` command (see #187);
- Add `zrevrangebylex` command (see #201);
- Add `command`, `command_count`, `command_getkeys` and `command_info` commands (see #229);
- Add `ping` support in pubsub connection (see #264);
- Add `exist` parameter to `zadd` command (see #288);
- Add `MaxClientsError` and implement `ReplyError` specialization (see #325);
- Add `encoding` parameter to sorted set commands (see #289);

FIX:

- Fix `CancelledError` in `conn._reader_task` (see #301);
- Fix pending commands cancellation with `CancelledError`, use explicit exception instead of calling `cancel()` method (see #316);
- Correct error message on Sentinel discovery of master/slave with password (see #327);
- Fix `bytearray` support as command argument (see #329);
- Fix critical bug in patched asyncio.Lock (see #256);
- Fix Multi/Exec transaction canceled error (see #225);
- Add missing arguments to `create_redis` and `create_redis_pool`;
- Fix deprecation warning (see #191);
- Make correct `__aiter__()` (see #192);
- Backward compatibility fix for `with (yield from pool) as conn:` (see #205);
- Fixed pubsub receiver stop() (see #211);

MISC:

- Multiple test fixes;
- Add PyPy3 to build matrix;
- Update dependencies versions;
- Add missing Python 3.6 classifier;

7.10.4 0.3.5 (2017-11-08)

FIX:

- Fix for indistinguishable futures cancellation with `asyncio.CancelledError` (see #316), cherry-picked from master;

7.10.5 0.3.4 (2017-10-25)

FIX:

- Fix time command result decoding when using connection-wide encoding setting (see #266);

7.10.6 0.3.3 (2017-06-30)

FIX:

- Critical bug fixed in patched asyncio.Lock (see #256);

7.10.7 0.3.2 (2017-06-21)

NEW:

- Added `zrevrangebylex` command (see #201), cherry-picked from master;
- Add connection timeout (see #221), cherry-picked from master;

FIX:

- Fixed pool close warning (see #239 and #236), cherry-picked from master;
- Fixed asyncio Lock deadlock issue (see #231 and #241);

7.10.8 0.3.1 (2017-05-09)

FIX:

- Fix pubsub Receiver missing iter() method (see #203);

7.10.9 0.3.0 (2017-01-11)

NEW:

- Pub/Sub connection commands accept `Channel` instances (see #168);
- Implement new Pub/Sub MPSC (multi-producers, single-consumer) Queue – `aioredis.pubsub.Receiver` (see #176);
- Add `aioredis.abc` module providing abstract base classes defining interface for basic lib components; (see #176);
- Implement Geo commands support (see #177 and #179);

FIX:

- Minor tests fixes;

MISC:

- Update examples and docs to use `async`/`await` syntax also keeping `yield from` examples for history (see #173);
- Reflow Travis CI configuration; add Python 3.6 section (see #170);
- Add AppVeyor integration to run tests on Windows (see #180);
- Update multiple development requirements;

7.10.10 0.2.9 (2016-10-24)

NEW:

- Allow multiple keys in `EXISTS` command (see #156 and #157);

FIX:

- Close RedisPool when connection to Redis failed (see #136);
- Add simple `INFO` command argument validation (see #140);
- Remove invalid uses of `next()`

MISC:

- Update devel.rst docs; update Pub/Sub Channel docs (cross-refs);
- Update MANIFEST.in to include docs, examples and tests in source bundle;

7.10.11 0.2.8 (2016-07-22)

NEW:

- Add `hmset_dict` command (see #130);
- Add `RedisConnection.address` property;
- RedisPool `minsize`/`maxsize` must not be `None`;
- Implement `close()`/`wait_closed()`/`closed` interface for pool (see #128);

FIX:

- Add test for `hstrlen`;
- Test fixes

MISC:

- Enable Redis 3.2.0 on Travis;
- Add spell checking when building docs (see #132);
- Documentation updated;

7.10.12 0.2.7 (2016-05-27)

- `create_pool()` minsize default value changed to 1;
- Fixed cancellation of wait_closed (see #118);
- Fixed `time()` convertion to float (see #126);
- Fixed `hmset()` method to return bool instead of `b'OK'` (see #126);

- Fixed multi/exec + watch issue (changed watch variable was causing `tr.execute()` to fail) (see #121);
- Replace `asyncio.Future` uses with utility method (get ready to Python 3.5.2 `loop.create_future()`);
- Tests switched from unittest to pytest (see #126);
- Documentation updates;

7.10.13 0.2.6 (2016-03-30)

- Fixed Multi/Exec transactions cancellation issue (see #110 and #114);
- Fixed Pub/Sub subscribe concurrency issue (see #113 and #115);
- Add SSL/TLS support (see #116);
- `aioredis.ConnectionClosedError` raised in `execute_pubsub` as well (see #108);
- `Redis.slaveof()` method signature changed: now to disable replication one should call `redis.slaveof(None)` instead of `redis.slaveof()`;
- More tests added;

7.10.14 0.2.5 (2016-03-02)

- Close all Pub/Sub channels on connection close (see #88);
- Add `iter()` method to `aioredis.Channel` allowing to use it with `async for` (see #89);
- Inline code samples in docs made runnable and downloadable (see #92);
- Python 3.5 examples converted to use `async`/`await` syntax (see #93);
- Fix Multi/Exec to honor encoding parameter (see #94 and #97);
- Add debug message in `create_connection` (see #90);
- Replace `asyncio.async` calls with wrapper that respects asyncio version (see #101);
- Use NODELAY option for TCP sockets (see #105);
- New `aioredis.ConnectionClosedError` exception added. Raised if connection to Redis server is lost (see #108 and #109);
- Fix RedisPool to close and drop connection in subscribe mode on release;
- Fix `aioredis.util.decode` to recursively decode list responses;
- More examples added and docs updated;
- Add google groups link to README;
- Bump year in LICENSE and docs;

7.10.15 0.2.4 (2015-10-13)

- Python 3.5 `async` support:
  - New scan commands API (`iscan`, `izscan`, `ihscan`);
  - Pool made awaitable (allowing `with await pool: ...` and `async with pool.get() as conn:` constructs);

- Fixed dropping closed connections from free pool (see #83);
- Docs updated;

7.10.16 0.2.3 (2015-08-14)

- Redis cluster support work in progress;
- Fixed pool issue causing pool growth over max size & `acquire` call hangs (see #71);
- `info` server command result parsing implemented;
- Fixed behavior of util functions (see #70);
- `hstrlen` command added;
- Few fixes in examples;
- Few fixes in documentation;

7.10.17 0.2.2 (2015-07-07)

- Decoding data with `encoding` parameter now takes into account list (array) replies (see #68);
- `encoding` parameter added to following commands:
  - generic commands: keys, randomkey;
  - hash commands: hgetall, hkeys, hmget, hvals;
  - list commands: blpop, brpop, brpoplpush, lindex, lpop, lrange, rpop, rpoplpush;
  - set commands: smembers, spop, srandmember;
  - string commands: getrange, getset, mget;
- Backward incompatibility:

  `ltrim` command now returns bool value instead of 'OK';
- Tests updated;

7.10.18 0.2.1 (2015-07-06)

- Logging added (aioredis.log module);
- Fixed issue with `wait_message` in pub/sub (see #66);

7.10.19 0.2.0 (2015-06-04)

- Pub/Sub support added;
- Fix in `zrevrangebyscore` command (see #62);
- Fixes/tests/docs;

7.10.20 0.1.5 (2014-12-09)

- AutoConnector added;
- wait_closed method added for clean connections shutdown;
- `zscore` command fixed;
- Test fixes;

7.10.21 0.1.4 (2014-09-22)

- Dropped following Redis methods – `Redis.multi()`, `Redis.exec()`, `Redis.discard()`;
- `Redis.multi_exec` hack'ish property removed;
- `Redis.multi_exec()` method added;
- High-level commands implemented:
  - generic commands (tests);
  - transactions commands (api stabilization).
- Backward incompatibilities:
  - Following sorted set commands' API changed:

    `zcount`, `zrangebyscore`, `zremrangebyscore`, `zrevrangebyscore`;
  - set string command' API changed;

7.10.22 0.1.3 (2014-08-08)

- RedisConnection.execute refactored to support commands pipelining (see #33);
- Several fixes;
- WIP on transactions and commands interface;
- High-level commands implemented and tested:
  - hash commands;
  - hyperloglog commands;
  - set commands;
  - scripting commands;
  - string commands;
  - list commands;

7.10.23 0.1.2 (2014-07-31)

- `create_connection`, `create_pool`, `create_redis` functions updated: db and password arguments made keyword-only (see #26);
- Fixed transaction handling (see #32);
- Response decoding (see #16);

7.10.24 0.1.1 (2014-07-07)

- Transactions support (in connection, high-level commands have some issues);
- Docs & tests updated.

7.10.25 0.1.0 (2014-06-24)

- Initial release;
- RedisConnection implemented;
- RedisPool implemented;
- Docs for RedisConnection & RedisPool;
- WIP on high-level API.

7.11 Glossary

asyncio Reference implementation of **PEP 3156**

See https://pypi.python.org/pypi/asyncio

error replies Redis server replies that start with - (minus) char. Usually starts with `-ERR`.

hiredis Python extension that wraps protocol parsing code in hiredis.

See https://pypi.python.org/pypi/hiredis

pytest A mature full-featured Python testing tool. See http://pytest.org/latest/

uvloop Is an ultra fast implementation of asyncio event loop on top of libuv. See https://github.com/MagicStack/uvloop

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

aioredis, 23
aioredis.abc, 56
aioredis.commands, 15
aioredis.pubsub, 58
aioredis.sentinel, 60

Index

Symbols

_Sender (*class in aioredis.pubsub*), 59

A

AbcChannel (*class in aioredis.abc*), 57
AbcConnection (*class in aioredis.abc*), 56
AbcPool (*class in aioredis.abc*), 57
acquire() (*aioredis.abc.AbcPool method*), 57
acquire() (*aioredis.ConnectionsPool method*), 28
address (*aioredis.abc.AbcConnection attribute*), 57
address (*aioredis.abc.AbcPool attribute*), 57
address (*aioredis.Redis attribute*), 34
address (*aioredis.RedisConnection attribute*), 24
aioredis (*module*), 23
aioredis.abc (*module*), 56
aioredis.commands (*module*), 15, 34
aioredis.pubsub (*module*), 58
aioredis.sentinel (*module*), 60
append() (*aioredis.commands.StringCommandsMixin method*), 39
asyncio, **78**
auth() (*aioredis.Redis method*), 34
auth() (*aioredis.RedisConnection method*), 26
AuthError, 30

B

bgrewriteaof() (*aioredis.commands.ServerCommandsMixin method*), 47, 52
bgsave() (*aioredis.commands.ServerCommandsMixin method*), 47, 52
bitcount() (*aioredis.commands.StringCommandsMixin method*), 39
bitop_and() (*aioredis.commands.StringCommandsMixin method*), 39
bitop_not() (*aioredis.commands.StringCommandsMixin method*), 39
bitop_or() (*aioredis.commands.StringCommandsMixin method*), 39
bitop_xor() (*aioredis.commands.StringCommandsMixin method*), 39
bitpos() (*aioredis.commands.StringCommandsMixin method*), 39
blpop() (*aioredis.commands.ListCommandsMixin method*), 42
brpop() (*aioredis.commands.ListCommandsMixin method*), 42
brpoplpush() (*aioredis.commands.ListCommandsMixin method*), 43

C

Channel (*class in aioredis*), 29
channel() (*aioredis.pubsub.Receiver method*), 59
ChannelClosedError, 30
channels (*aioredis.commands.PubSubCommandsMixin attribute*), 54
channels (*aioredis.pubsub.Receiver attribute*), 59
check_quorum() (*aioredis.sentinel.RedisSentinel method*), 62
check_stop() (*aioredis.pubsub.Receiver method*), 59
clear() (*aioredis.ConnectionsPool method*), 28
client_getname() (*aioredis.commands.ServerCommandsMixin method*), 47, 52
client_kill() (*aioredis.commands.ServerCommandsMixin method*), 47, 52
client_list() (*aioredis.commands.ServerCommandsMixin method*), 47, 52
client_pause() (*aioredis.commands.ServerCommandsMixin method*), 48, 52
client_setname() (*aioredis.commands.ServerCommandsMixin*

method), 48, 52
close() (*aioredis.abc.AbcChannel method*), 58
close() (*aioredis.abc.AbcConnection method*), 57
close() (*aioredis.ConnectionsPool method*), 28
close() (*aioredis.Redis method*), 34
close() (*aioredis.RedisConnection method*), 25
close() (*aioredis.sentinel.RedisSentinel method*), 62
close() (*aioredis.sentinel.SentinelPool method*), 63
closed (*aioredis.abc.AbcConnection attribute*), 57
closed (*aioredis.ConnectionsPool attribute*), 27
closed (*aioredis.Redis attribute*), 34
closed (*aioredis.RedisConnection attribute*), 24
closed (*aioredis.sentinel.RedisSentinel attribute*), 61
closed (*aioredis.sentinel.SentinelPool attribute*), 62
command() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
command_count() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
command_getkeys() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
command_info() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
config_get() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
config_resetstat() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
config_rewrite() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
config_set() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
connection (*aioredis.Redis attribute*), 34
ConnectionClosedError, 30
ConnectionForcedCloseError, 30
ConnectionsPool (*class in aioredis*), 27
create_connection() (*built-in function*), 69
create_connection() (*in module aioredis*), 24
create_pool() (*built-in function*), 69
create_pool() (*in module aioredis*), 26
create_redis() (*built-in function*), 69
create_redis() (*in module aioredis*), 32
create_redis_pool() (*in module aioredis*), 33
create_sentinel() (*in module aioredis.sentinel*), 60
create_sentinel_pool() (*in module aioredis.sentinel*), 62

D

db (*aioredis.abc.AbcConnection attribute*), 57
db (*aioredis.ConnectionsPool attribute*), 27
db (*aioredis.Redis attribute*), 34
db (*aioredis.RedisConnection attribute*), 24
dbsize() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
debug_object() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
debug_segfault() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
debug_sleep() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
decr() (*aioredis.commands.StringCommandsMixin method*), 39
decrby() (*aioredis.commands.StringCommandsMixin method*), 39
delete() (*aioredis.commands.GenericCommandsMixin method*), 35
discover() (*aioredis.sentinel.SentinelPool method*), 63
discover_master() (*aioredis.sentinel.SentinelPool method*), 63
discover_slave() (*aioredis.sentinel.SentinelPool method*), 63
dump() (*aioredis.commands.GenericCommandsMixin method*), 35

E

echo() (*aioredis.Redis method*), 34
encoding (*aioredis.abc.AbcConnection attribute*), 57
encoding (*aioredis.ConnectionsPool attribute*), 27
encoding (*aioredis.Redis attribute*), 34
encoding (*aioredis.RedisConnection attribute*), 24
error replies, **78**
eval() (*aioredis.commands.ScriptingCommandsMixin method*), 52
evalsha() (*aioredis.commands.ScriptingCommandsMixin method*), 52
execute() (*aioredis.abc.AbcConnection method*), 57
execute() (*aioredis.commands.MultiExec method*), 51
execute() (*aioredis.commands.Pipeline method*), 51
execute() (*aioredis.ConnectionsPool method*), 27
execute() (*aioredis.RedisConnection method*), 25
execute() (*aioredis.sentinel.RedisSentinel method*), 61
execute() (*aioredis.sentinel.SentinelPool method*), 63
execute_pubsub() (*aioredis.abc.AbcConnection method*), 57

execute_pubsub() (*aioredis.ConnectionsPool method*), 28
execute_pubsub() (*aioredis.RedisConnection method*), 25
exists() (*aioredis.commands.GenericCommandsMixin method*), 35
expire() (*aioredis.commands.GenericCommandsMixin method*), 35
expireat() (*aioredis.commands.GenericCommandsMixin method*), 35

F

failover() (*aioredis.sentinel.RedisSentinel method*), 62
flushall() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
flushdb() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
freesize (*aioredis.ConnectionsPool attribute*), 27

G

GenericCommandsMixin (*class in aioredis.commands*), 35
geoadd() (*aioredis.commands.GeoCommandsMixin method*), 37
GeoCommandsMixin (*class in aioredis.commands*), 37
geodist() (*aioredis.commands.GeoCommandsMixin method*), 37
geohash() (*aioredis.commands.GeoCommandsMixin method*), 37
GeoMember (*class in aioredis.commands*), 38
GeoPoint (*class in aioredis.commands*), 38
geopos() (*aioredis.commands.GeoCommandsMixin method*), 37
georadius() (*aioredis.commands.GeoCommandsMixin method*), 37
georadiusbymember() (*aioredis.commands.GeoCommandsMixin method*), 38
get() (*aioredis.abc.AbcChannel method*), 58
get() (*aioredis.Channel method*), 29
get() (*aioredis.commands.StringCommandsMixin method*), 39
get() (*aioredis.pubsub.Receiver method*), 59
get_connection() (*aioredis.abc.AbcPool method*), 57
get_connection() (*aioredis.ConnectionsPool method*), 28
get_json() (*aioredis.Channel method*), 29
getbit() (*aioredis.commands.StringCommandsMixin method*), 39
getrange() (*aioredis.commands.StringCommandsMixin method*), 40
getset() (*aioredis.commands.StringCommandsMixin method*), 40

H

HashCommandsMixin (*class in aioredis.commands*), 41
hdel() (*aioredis.commands.HashCommandsMixin method*), 41
hexists() (*aioredis.commands.HashCommandsMixin method*), 41
hget() (*aioredis.commands.HashCommandsMixin method*), 41
hgetall() (*aioredis.commands.HashCommandsMixin method*), 41
hincrby() (*aioredis.commands.HashCommandsMixin method*), 41
hincrbyfloat() (*aioredis.commands.HashCommandsMixin method*), 41
hiredis, **78**
hkeys() (*aioredis.commands.HashCommandsMixin method*), 41
hlen() (*aioredis.commands.HashCommandsMixin method*), 41
hmget() (*aioredis.commands.HashCommandsMixin method*), 41
hmset() (*aioredis.commands.HashCommandsMixin method*), 41
hmset_dict() (*aioredis.commands.HashCommandsMixin method*), 41
hscan() (*aioredis.commands.HashCommandsMixin method*), 42
hset() (*aioredis.commands.HashCommandsMixin method*), 42
hsetnx() (*aioredis.commands.HashCommandsMixin method*), 42
hstrlen() (*aioredis.commands.HashCommandsMixin method*), 42
hvals() (*aioredis.commands.HashCommandsMixin method*), 42
HyperLogLogCommandsMixin (*class in aioredis.commands*), 49

I

ihscan() (*aioredis.commands.HashCommandsMixin method*), 42
in_pubsub (*aioredis.abc.AbcConnection attribute*), 57
in_pubsub (*aioredis.commands.PubSubCommandsMixin attribute*), 54
in_pubsub (*aioredis.RedisConnection attribute*), 25
in_transaction (*aioredis.Redis attribute*), 34
in_transaction (*aioredis.RedisConnection attribute*), 24

incr() (*aioredis.commands.StringCommandsMixin method*), 40
incrby() (*aioredis.commands.StringCommandsMixin method*), 40
incrbyfloat() (*aioredis.commands.StringCommandsMixin method*), 40
info() (*aioredis.commands.ServerCommandsMixin method*), 48, 53
is_active (*aioredis.abc.AbcChannel attribute*), 58
is_active (*aioredis.Channel attribute*), 29
is_active (*aioredis.pubsub.Receiver attribute*), 59
is_pattern (*aioredis.abc.AbcChannel attribute*), 58
is_pattern (*aioredis.Channel attribute*), 29
iscan() (*aioredis.commands.GenericCommandsMixin method*), 35
isscan() (*aioredis.commands.SetCommandsMixin method*), 44
iter() (*aioredis.Channel method*), 29
iter() (*aioredis.pubsub.Receiver method*), 59
izscan() (*aioredis.commands.SortedSetCommandsMixin method*), 45

K

keys() (*aioredis.commands.GenericCommandsMixin method*), 35

L

lastsave() (*aioredis.commands.ServerCommandsMixin method*), 49, 53
lindex() (*aioredis.commands.ListCommandsMixin method*), 43
linsert() (*aioredis.commands.ListCommandsMixin method*), 43
ListCommandsMixin (*class in aioredis.commands*), 42
llen() (*aioredis.commands.ListCommandsMixin method*), 43
loop, 69
lpop() (*aioredis.commands.ListCommandsMixin method*), 43
lpush() (*aioredis.commands.ListCommandsMixin method*), 43
lpushx() (*aioredis.commands.ListCommandsMixin method*), 43
lrange() (*aioredis.commands.ListCommandsMixin method*), 43
lrem() (*aioredis.commands.ListCommandsMixin method*), 43
lset() (*aioredis.commands.ListCommandsMixin method*), 43
ltrim() (*aioredis.commands.ListCommandsMixin method*), 43

M

master() (*aioredis.sentinel.RedisSentinel method*), 61
master_address() (*aioredis.sentinel.RedisSentinel method*), 61
master_for() (*aioredis.sentinel.RedisSentinel method*), 61
master_for() (*aioredis.sentinel.SentinelPool method*), 62
MasterNotFoundError, 31
MasterReplyError, 31
masters() (*aioredis.sentinel.RedisSentinel method*), 61
MaxClientsError, 30
maxsize (*aioredis.ConnectionsPool attribute*), 27
mget() (*aioredis.commands.StringCommandsMixin method*), 40
migrate() (*aioredis.commands.GenericCommandsMixin method*), 35
migrate_keys() (*aioredis.commands.GenericCommandsMixin method*), 35
minsize (*aioredis.ConnectionsPool attribute*), 27
monitor() (*aioredis.commands.ServerCommandsMixin method*), 49, 53
monitor() (*aioredis.sentinel.RedisSentinel method*), 61
move() (*aioredis.commands.GenericCommandsMixin method*), 35
mset() (*aioredis.commands.StringCommandsMixin method*), 40
msetnx() (*aioredis.commands.StringCommandsMixin method*), 40
multi_exec() (*aioredis.commands.TransactionsCommandsMixin method*), 50
MultiExec (*class in aioredis.commands*), 51
MultiExecError, 30

N

name (*aioredis.abc.AbcChannel attribute*), 58
name (*aioredis.Channel attribute*), 29

O

object_encoding() (*aioredis.commands.GenericCommandsMixin method*), 35
object_idletime() (*aioredis.commands.GenericCommandsMixin method*), 35
object_refcount() (*aioredis.commands.GenericCommandsMixin method*), 36

P

pattern() (*aioredis.pubsub.Receiver method*), 59
patterns (*aioredis.commands.PubSubCommandsMixin attribute*), 54
patterns (*aioredis.pubsub.Receiver attribute*), 59
persist() (*aioredis.commands.GenericCommandsMixin method*), 36
pexpire() (*aioredis.commands.GenericCommandsMixin method*), 36
pexpireat() (*aioredis.commands.GenericCommandsMixin method*), 36
pfadd() (*aioredis.commands.HyperLogLogCommandsMixin method*), 49
pfcount() (*aioredis.commands.HyperLogLogCommandsMixin method*), 49
pfmerge() (*aioredis.commands.HyperLogLogCommandsMixin method*), 50
ping() (*aioredis.Redis method*), 34
ping() (*aioredis.sentinel.RedisSentinel method*), 61
Pipeline (*class in aioredis.commands*), 51
pipeline() (*aioredis.commands.TransactionsCommandsMixin method*), 50
PipelineError, 30
pool, 69
PoolClosedError, 31
ProtocolError, 30
psetex() (*aioredis.commands.StringCommandsMixin method*), 40
psubscribe() (*aioredis.commands.PubSubCommandsMixin method*), 55
pttl() (*aioredis.commands.GenericCommandsMixin method*), 36
publish() (*aioredis.commands.PubSubCommandsMixin method*), 55
publish_json() (*aioredis.commands.PubSubCommandsMixin method*), 55
pubsub_channels (*aioredis.abc.AbcConnection attribute*), 57
pubsub_channels (*aioredis.RedisConnection attribute*), 25
pubsub_channels() (*aioredis.commands.PubSubCommandsMixin method*), 55
pubsub_numpat() (*aioredis.commands.PubSubCommandsMixin method*), 55
pubsub_numsub() (*aioredis.commands.PubSubCommandsMixin method*), 55
pubsub_patterns (*aioredis.abc.AbcConnection attribute*), 57
pubsub_patterns (*aioredis.RedisConnection attribute*), 25
PubSubCommandsMixin (*class in aioredis.commands*), 54
punsubscribe() (*aioredis.commands.PubSubCommandsMixin method*), 55
put_nowait() (*aioredis.abc.AbcChannel method*), 58
pytest, **78**
pytest.assert_almost_equal() (*built-in function*), 70
pytest.logs() (*built-in function*), 70
pytest.raises_regex() (*built-in function*), 70
pytest.redis_version() (*built-in function*), 70
Python Enhancement Proposals
    PEP 3156, 1, 78
    PEP 492, 18

Q

quit() (*aioredis.Redis method*), 34

R

randomkey() (*aioredis.commands.GenericCommandsMixin method*), 36
ReadOnlyError, 31
Receiver (*class in aioredis.pubsub*), 58
redis, 69
Redis (*class in aioredis*), 34
RedisConnection (*class in aioredis*), 24
RedisError, 30
RedisSentinel (*class in aioredis.sentinel*), 61
release() (*aioredis.abc.AbcPool method*), 57
release() (*aioredis.ConnectionsPool method*), 28
remove() (*aioredis.sentinel.RedisSentinel method*), 62
rename() (*aioredis.commands.GenericCommandsMixin method*), 36
renamenx() (*aioredis.commands.GenericCommandsMixin method*), 36
ReplyError, 30
restore() (*aioredis.commands.GenericCommandsMixin method*), 36
role() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
rpop() (*aioredis.commands.ListCommandsMixin method*), 43
rpoplpush() (*aioredis.commands.ListCommandsMixin method*), 43
rpush() (*aioredis.commands.ListCommandsMixin method*), 43
rpushx() (*aioredis.commands.ListCommandsMixin method*), 43

S

sadd() (*aioredis.commands.SetCommandsMixin method*), 44
save() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
scan() (*aioredis.commands.GenericCommandsMixin method*), 36
scard() (*aioredis.commands.SetCommandsMixin method*), 44
script_exists() (*aioredis.commands.ScriptingCommandsMixin method*), 52
script_flush() (*aioredis.commands.ScriptingCommandsMixin method*), 52
script_kill() (*aioredis.commands.ScriptingCommandsMixin method*), 52
script_load() (*aioredis.commands.ScriptingCommandsMixin method*), 52
ScriptingCommandsMixin (*class in aioredis.commands*), 52
sdiff() (*aioredis.commands.SetCommandsMixin method*), 44
sdiffstore() (*aioredis.commands.SetCommandsMixin method*), 44
select() (*aioredis.ConnectionsPool method*), 28
select() (*aioredis.Redis method*), 34
select() (*aioredis.RedisConnection method*), 26
SentinelPool (*class in aioredis.sentinel*), 62
sentinels() (*aioredis.sentinel.RedisSentinel method*), 61
server, 69
serverB, 70
ServerCommandsMixin (*class in aioredis.commands*), 47, 52
set() (*aioredis.commands.StringCommandsMixin method*), 40
set() (*aioredis.sentinel.RedisSentinel method*), 62
setbit() (*aioredis.commands.StringCommandsMixin method*), 40
SetCommandsMixin (*class in aioredis.commands*), 44
setex() (*aioredis.commands.StringCommandsMixin method*), 40
setnx() (*aioredis.commands.StringCommandsMixin method*), 40
setrange() (*aioredis.commands.StringCommandsMixin method*), 41
shutdown() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
sinter() (*aioredis.commands.SetCommandsMixin method*), 44
sinterstore() (*aioredis.commands.SetCommandsMixin method*), 44
sismember() (*aioredis.commands.SetCommandsMixin method*), 44
size (*aioredis.ConnectionsPool attribute*), 27
slave_for() (*aioredis.sentinel.RedisSentinel method*), 61
slave_for() (*aioredis.sentinel.SentinelPool method*), 63
SlaveNotFoundError, 31
slaveof() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
SlaveReplyError, 31
slaves() (*aioredis.sentinel.RedisSentinel method*), 61
slowlog_get() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
slowlog_len() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
slowlog_reset() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
smembers() (*aioredis.commands.SetCommandsMixin method*), 44
smove() (*aioredis.commands.SetCommandsMixin method*), 44
sort() (*aioredis.commands.GenericCommandsMixin method*), 36
SortedSetCommandsMixin (*class in aioredis.commands*), 45
spop() (*aioredis.commands.SetCommandsMixin method*), 44
srandmember() (*aioredis.commands.SetCommandsMixin method*), 44
srem() (*aioredis.commands.SetCommandsMixin method*), 44
sscan() (*aioredis.commands.SetCommandsMixin method*), 44
ssl_proxy() (*built-in function*), 70
start_server() (*built-in function*), 70
stop() (*aioredis.pubsub.Receiver method*), 59
StreamCommandsMixin (*class in aioredis.commands*), 55
StringCommandsMixin (*class in aioredis.commands*), 39
strlen() (*aioredis.commands.StringCommandsMixin method*), 41
subscribe() (*aioredis.commands.PubSubCommandsMixin method*), 55

sunion() (*aioredis.commands.SetCommandsMixin method*), 44
sunionstore() (*aioredis.commands.SetCommandsMixin method*), 44
sync() (*aioredis.commands.ServerCommandsMixin method*), 49, 54

T

time() (*aioredis.commands.ServerCommandsMixin method*), 49, 54
touch() (*aioredis.commands.GenericCommandsMixin method*), 36
TransactionsCommandsMixin (*class in aioredis.commands*), 50
ttl() (*aioredis.commands.GenericCommandsMixin method*), 36
type() (*aioredis.commands.GenericCommandsMixin method*), 37

U

unlink() (*aioredis.commands.GenericCommandsMixin method*), 37
unsubscribe() (*aioredis.commands.PubSubCommandsMixin method*), 55
unused_port() (*built-in function*), 69
unwatch() (*aioredis.commands.TransactionsCommandsMixin method*), 51
uvloop, **78**

W

wait() (*aioredis.commands.GenericCommandsMixin method*), 37
wait_closed() (*aioredis.abc.AbcConnection method*), 57
wait_closed() (*aioredis.ConnectionsPool method*), 28
wait_closed() (*aioredis.Redis method*), 34
wait_closed() (*aioredis.RedisConnection method*), 26
wait_closed() (*aioredis.sentinel.RedisSentinel method*), 62
wait_closed() (*aioredis.sentinel.SentinelPool method*), 63
wait_message() (*aioredis.Channel method*), 29
wait_message() (*aioredis.pubsub.Receiver method*), 59
watch() (*aioredis.commands.TransactionsCommandsMixin method*), 51
WatchVariableError, 30

X

xack() (*aioredis.commands.StreamCommandsMixin method*), 55
xadd() (*aioredis.commands.StreamCommandsMixin method*), 55
xclaim() (*aioredis.commands.StreamCommandsMixin method*), 55
xgroup_create() (*aioredis.commands.StreamCommandsMixin method*), 56
xgroup_delconsumer() (*aioredis.commands.StreamCommandsMixin method*), 56
xgroup_destroy() (*aioredis.commands.StreamCommandsMixin method*), 56
xgroup_setid() (*aioredis.commands.StreamCommandsMixin method*), 56
xinfo() (*aioredis.commands.StreamCommandsMixin method*), 56
xinfo_consumers() (*aioredis.commands.StreamCommandsMixin method*), 56
xinfo_groups() (*aioredis.commands.StreamCommandsMixin method*), 56
xinfo_help() (*aioredis.commands.StreamCommandsMixin method*), 56
xinfo_stream() (*aioredis.commands.StreamCommandsMixin method*), 56
xpending() (*aioredis.commands.StreamCommandsMixin method*), 56
xrange() (*aioredis.commands.StreamCommandsMixin method*), 56
xread() (*aioredis.commands.StreamCommandsMixin method*), 56
xread_group() (*aioredis.commands.StreamCommandsMixin method*), 56
xrevrange() (*aioredis.commands.StreamCommandsMixin method*), 56

Z

zadd() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zcard() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zcount() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zincrby() (*aioredis.commands.SortedSetCommandsMixin method*), 45

zinterstore() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zlexcount() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zrange() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zrangebylex() (*aioredis.commands.SortedSetCommandsMixin method*), 45
zrangebyscore() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zrank() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zrem() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zremrangebylex() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zremrangebyrank() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zremrangebyscore() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zrevrange() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zrevrangebylex() (*aioredis.commands.SortedSetCommandsMixin method*), 46
zrevrangebyscore() (*aioredis.commands.SortedSetCommandsMixin method*), 47
zrevrank() (*aioredis.commands.SortedSetCommandsMixin method*), 47
zscan() (*aioredis.commands.SortedSetCommandsMixin method*), 47
zscore() (*aioredis.commands.SortedSetCommandsMixin method*), 47
zunionstore() (*aioredis.commands.SortedSetCommandsMixin method*), 47