SiderPy

Release 0.7.0

Roma Koshel

Mar 26, 2023

CONTENTS

SiderPy is a Python 3.6+ minimalistic asyncio client to Redis.

CHAPTER

ONE

GUIDES

1.1 Getting Started

1.1.1 Installation

Make sure you have both pip and at least version 3.6 of Python.

To install SiderPy with hiredis support run

```
pip install git+https://github.com/levsh/siderpy.git#egg=siderpy[hiredis]
```

or with pure python parser

```
pip install git+https://github.com/levsh/siderpy.git#egg=siderpy
```

1.1.2 Basic usage

It's very simple.

SiderPy uses uri format to connect to the Redis server.

```
import siderpy

redis = siderpy.Redis('redis://username:password@localhost:6379?db=0')
```

or in case of unix socket

```
redis = siderpy.Redis('redis+unix://username:password@/var/run/redis.sock?db=0')
```

It's possibly to specify connection open timeout, read/write timeout, ssl, and server data decoding. For *__init__* method details see `API Reference`

Instead of opening connection to the server at the time `Redis` object was created, the connection is establish lazy at the first call. It's allows to create `Redis` instance inside *__init__* method.

`Redis` class doesn't explicitly define Redis commands as methods of itself, except *execute* (exec) and *delete* (del), but calling command is same as calling instance method.

```
await redis.ping()
await redis.ping('Hello!')
await redis.set('key', 'value')
response = await redis.get('key')
```

After the Redis object no longer needed call `close()` method to close underlying connection to the server and free resources.

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

1.1.3 Transactions with multi/exec

To use transaction just wraps your command into multi/exec block.

```
await redis.multi()
await redis.set('key', 'value')
...
await redis.execute()  # Redis 'exec' command
```

1.1.4 Pipeline

To enable pipeline call `pipeline_on()`. After that all subsequent commands are saved in the internall buffer until `pipeline_off()` method is called. To execute stored buffer run `pipeline_execute()`.

```
redis.pipeline_on()
await redis.set('key1', 'value1')
await redis.set('key2', 'value2')
...
await redis.set('keyN', 'valueN')
response = await redis.pipeline_execute()
redis.pipeline_off()

# or
with redis.pipeline():
    ...
await redis.pipeline_execute()
```

1.1.5 Publish/Subscribe

Publish to a channel:

```
await redis.publish('channel', 'Hello World!')
```

Subscribe to a channel(s):

```
await redis.subscribe('channel1', 'channel2', ..., 'channelN')
```

To receive messages from subscribed channels just iterate over `Redis` object.

```
async for message in redis:
    print(message)
```

or use `pubsub_queue` directly

```
# 1
message = await redis.pubsub_queue.get()
```

(continues on next page)

(continued from previous page)

```
# 2
async for message in redis.pubsub_queue:
    ...
```

If a error occurs during consuming then it will be raised.

```
async for mesasge in redis:
    print(message)
# connection error occurs

Traceback (most recent call last):
  File "test.py", line 24, in <module>
    asyncio.run(main())
    ...
    raise ConnectionError
ConnectionError
```

```
await redis.pubsub_queue.get()
# connection error occurs

Traceback (most recent call last):
  File "test.py", line 24, in <module>
    asyncio.run(main())
    ...
    raise ConnectionError
ConnectionError
```

In this case it's necessary to resubscribe again to continue recieving messages.

1.1.6 Pool

`Redis` class represents a signle network connection. If yout need a pool of connections use `RedisPool` or implement your own. `RedisPool` supports direct commands call except connection dependent commands such as subscribe, psubscribe, unsubscribe, punsubscribe, multi, exec, discard, etc.

```
pool = siderpy.RedisPool('redis://localhost:6379?db=0')
await pool.ping()
await pool.get('key')
```

But it's recommended to get the `Redis` object and use it

```
async with pool.get_redis() as redis:
    await redis.get('key')
    ...
```

1.2 API Reference

exception siderpy.**SiderPyError**

Bases: `Exception`

Base error

exception siderpy.**RedisError**

Bases: *siderpy.SiderPyError*

Redis error

exception siderpy.**QueueClosedError**

Bases: *siderpy.SiderPyError*

Closed PubSub queue error

class siderpy.**Redis**(*url: str = 'redis://localhost:6379/0', connect_timeout: Union[float, int] = None, timeout: Union[float, tuple, list] = None, ssl_ctx: ssl.SSLContext = None, encoding=None, errors=None, pubsub_queue_maxsize=None*)

Bases: `object`

Class representing a single connection to a Redis server. Connection to the server is established automatically during first request.

Examples

```
>>> import siderpy
>>> redis = siderpy.Redis('redis://username:password@localhost:6379/0')
>>> await redis.ping()
>>> ...
>>> await redis.close()
```

__init__(*url: str = 'redis://localhost:6379/0', connect_timeout: Union[float, int] = None, timeout: Union[float, tuple, list] = None, ssl_ctx: ssl.SSLContext = None, encoding=None, errors=None, pubsub_queue_maxsize=None*)

Parameters

- **url** (`str`, optional) – The Redis server url and settings to connect as uri:
  - *redis://[USERNAME][:PASSWORD@]HOST[:PORT]/[DATABASE]*
  - *redis+unix://[USERNAME][:PASSWORD@]SOCKET_PATH[?db=DATABASE]*
  - *redis-socket://[USERNAME][:PASSWORD@]SOCKET_PATH[?db=DATABASE]*

  default: *redis://localhost:6379/0*
- **connect_timeout** (`float`, optional) – Timeout used to get initialized *Redis* instance and as `ssl_handshake_timeout` argument for `asyncio.open_connection` call.
- **timeout** (`float`, optional) – Timeout used for read and write operations. It is possibly to specify separately values for read and write.

Example

```
>>> Redis(timeout=(read_timeout, write_timeout))
```

If common or read timeout is specified it will affect all Redis blocking read commands such as blpop, etc. For example, this code will raise `asyncio.TimeoutError` after one second though a timeout of zero for blpop command can be used to block indefinitely.

```
>>> redis = siderpy.Redis(timeout=1)
>>> await redis.blpop('empty_list', 0)  # asyncio.TimeoutError␣
↪exception
>>>                                     # will occur here after 1␣
↪second
```

To avoid this situation set read timeout to `None`.

```
>>> redis = siderpy.Redis(timeout=(None, 15))
>>> await redis.blpop('empty_list', 0)  # will block indefinitely
```

- **encoding** (`str`, optional) – Encoding with which to decode raw data(bytes) from Redis.
- **errors** (`str`, optional) – Error handling scheme to use for handling of decoding errors.
- **ssl_ctx** (`ssl.SSLContext`, optional) – SSL context object to enable SSL(TLS).

async close()
Close established connection

async delete(**args*)
Redis *del* command

async execute()
Redis *exec* command

async execute_cmd(*cmd_name: str*, **args*)
Execute Redis command

Parameters **cmd_name** (`str`, optional) – Redis command name.

Example

```
>>> result = await redis.execute_cmd('get', 'key')
```

classmethod parse_url(*url: str*) → dict

pipeline()
Pipeline mode contextmanager

Example

```
>>> with redis.pipeline():
>>>     await redis.set('key1', 'value2')
>>>     await redis.set('key2', 'value2')
>>>     await redis.mget('key1', 'key2')
>>> result = await redis.pipeline_execute()
```

Also it's possible to resume or execute pipeline later, for example:

```
>>> with redis.pipeline():
>>>     await redis.set('key1', 'value2')
>>> # pause pipeline, do other stuff
>>> ...
>>> # continue with pipeline
>>> with redis.pipeline():
>>>     await redis.set('key2', 'value2')
>>> result = await redis.pipeline_execute()
```

pipeline_clear()
Clear internal pipeline buffer

async pipeline_execute()
Execute pipeline buffer

pipeline_off()
Disable pipeline mode

pipeline_on()
Enable pipeline mode. In this mode, all commands are saved to the internal pipeline buffer until `pipeline_off()` method is invoked directly. To execute stored buffer call `pipeline_execute()`

property pubsub_queue
Instance of `PubSubQueue` class. Holds incomming messages.

class siderpy.**RedisPool**(*url: str = 'redis://localhost:6379/0', connect_timeout: float = None, timeout: Union[float, tuple, list] = None, size: int = 4, pool_cls=<class 'siderpy.Pool'>, ssl_ctx: ssl.SSLContext = None*)

Bases: `object`

Class representing a pool of connections to a Redis server

```
>>> import siderpy
>>> pool = siderpy.RedisPool('redis://localhost:6379/0', size=10)
>>> await pool.ping()
>>> await pool.get('key')
>>> ...
>>> await pool.close()
```

Pool doesn't implement multi/exec and pub/sub commands. For performance reasons it's better to use Redis instance as command executor instead of pool itself. For example:

```
>>> with pool.get_redis() as redis:
>>>     await redis.set(...)
>>>     await redis.get(...)
>>>     ...
```

__init__(*url: str = 'redis://localhost:6379/0', connect_timeout: float = None, timeout: Union[float, tuple, list] = None, size: int = 4, pool_cls=<class 'siderpy.Pool'>, ssl_ctx: ssl.SSLContext = None*)

Parameters

- **url** (str, optional) – same as url argument for *Redis*.
- **connect_timeout** (float, optional) – same as connect_timeout argument for *Redis*.
- **timeout** (float, optional) – same as timeout argument for *Redis*.
- **size** (int, optional) – Pool size.
- **ssl_ctx** (ssl.SSLContext, optional) – same as ssl_ctx argument for *Redis*.

async close()
Close all established connections

async delete(**args*)
Redis *del* command

get_redis(*timeout: float = None*)
Context manager for getting Redis instance

Parameters **timeout** (*float*) – Timeout to get *Redis* instance

```
>>> async with pool.get_redis() as redis:
>>>     await redis.ping()
```

1.3 Benchmark

Benchmark tests

PYTHON MODULE INDEX

S

INDEX