ledis-py Documentation

Release 0.0.1

Andy McCurdy

October 29, 2014

Contents

CHAPTER 1

API Reference

class ledis.**Ledis**(*host='localhost'*, *port=6380*, *db=0*, *socket_timeout=None*, *connection_pool=None*, *charset='utf-8'*, *errors='strict'*, *decode_responses=False*, *unix_socket_path=None*)

Implementation of the Redis protocol.

This abstract class provides a Python interface to all LedisDB commands and an implementation of the Redis protocol.

Connection and Pipeline derive from this, implementing how the commands are sent and received to the Ledis server

bexpire(*name*, *time*)

Set timeout on key `name` with `time`

bexpireat(*name*, *when*)

Set an expire flag on key name for time seconds. time can be represented by an integer or a Python timedelta object.

bmsetbit(*name*, **args*)

Set any number of offset, value pairs to the key `name`. Pairs can be specified in the following way:

offset1, value1, offset2, value2, ...

bopt(*operation*, *dest*, **keys*)

Perform a bitwise operation using `operation` between `keys` and store the result in `dest`. `operation` is one of *and*, *or*, *xor*, *not*.

bpersist(*name*)

Removes an expiration on name

bttl(*name*)

Returns the number of seconds until the key name will expire

decr(*name*, *amount=1*)

Decrements the value of `key` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

decrby(*name*, *amount=1*)

Decrements the value of `key` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

delete(**names*)

Delete one or more keys specified by `names`

echo(*value*)

Echo the string back from the server

execute_command(**args*, ***options*)

Execute a command and return a parsed response

exists(*name*)
Returns a boolean indicating whether key `name` exists

expire(*name*, *time*)
Set an expire flag on key `name` for `time` seconds. `time` can be represented by an integer or a Python timedelta object.

expireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer indicating unix time or a Python datetime object.

classmethod from_url(*url*, *db=None*, ***kwargs*)
Return a Ledis client object configured from the given URL.

For example:

```
ledis://localhost:6380/0
unix:///path/to/socket.sock?db=0
```

There are several ways to specify a database number. The parse function will return the first specified option:

1. A `db` querystring option, e.g. ledis://localhost?db=0
2. If using the ledis:// scheme, the path argument of the url, e.g. ledis://localhost/0
3. The `db` argument to this function.

If none of these options are specified, db=0 is used.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class's initializer. In the case of conflicting arguments, querystring arguments always win.

get(*name*)
Return the value at key `name`, or None if the key doesn't exist

getset(*name*, *value*)
Set the value at key `name` to `value` if key doesn't exist Return the value at key `name` atomically

hclear(*name*)
Delete key `name` from hash

hdel(*name*, **keys*)
Delete `keys` from hash `name`

hexists(*name*, *key*)
Returns a boolean indicating if `key` exists within hash `name`

hexpire(*name*, *time*)
Set an expire flag on key name for time milliseconds. time can be represented by an integer or a Python timedelta object.

hexpireat(*name*, *when*)
Set an expire flag on key name. when can be represented as an integer representing unix time in milliseconds (unix time * 1000) or a Python datetime object.

hget(*name*, *key*)
Return the value of `key` within the hash `name`

hgetall(*name*)
Return a Python dict of the hash's name/value pairs

hincrby(*name*, *key*, *amount=1*)
Increment the value of `key` in hash `name` by `amount`

hkeys(*name*)
Return the list of keys within hash `name`

hlen(*name*)
Return the number of elements in hash `name`

hmclear(**names*)
Delete multiple keys `names` from hash

hmget(*name*, *keys*, **args*)
Returns a list of values ordered identically to `keys`

hmset(*name*, *mapping*)
Sets each key in the `mapping` dict to its corresponding value in the hash `name`

hpersist(*name*)
Removes an expiration on name

hset(*name*, *key*, *value*)
Set `key` to `value` within hash `name` Returns 1 if HSET created a new field, otherwise 0

httl(*name*)
Returns the number of seconds until the key name will expire

hvals(*name*)
Return the list of values within hash `name`

incr(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrby(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

lclear(*name*)
Delete the key of `name`

lexpire(*name*, *time*)
Set an expire flag on key `name` for `time` seconds. `time` can be represented by an integer or a Python timedelta object.

lexpireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer indicating unix time or a Python datetime object.

lindex(*name*, *index*)
Return the item from list `name` at position `index`

Negative indexes are supported and will return an item at the end of the list

llen(*name*)
Return the length of the list `name`

lmclear(**names*)
Delete multiple keys of `name`

lpersist(*name*)
Removes an expiration on `name`

lpop(*name*)
Remove and return the first item of the list `name`

lpush(*name*, **values*)
Push `values` onto the head of the list `name`

lrange(*name*, *start*, *end*)
Return a slice of the list `name` between position `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

lttl(*name*)
Returns the number of seconds until the key `name` will expire

mget(*keys*, **args*)
Returns a list of values ordered identically to `keys`

mset(**args*, ***kwargs*)
Sets key/values based on a mapping. Mapping can be supplied as a single dictionary argument or as kwargs.

parse_response(*connection*, *command_name*, ***options*)
Parses a response from the Ledis server

persist(*name*)
Removes an expiration on name

ping()
Ping the Ledis server

rpop(*name*)
Remove and return the last item of the list `name`

rpush(*name*, **values*)
Push `values` onto the tail of the list `name`

sadd(*name*, **values*)
Add `value(s)` to set `name`

scard(*name*)
Return the number of elements in set `name`

sclear(*name*)
Delete key `name` from set

sdiff(*keys*, **args*)
Return the difference of sets specified by `keys`

sdiffstore(*dest*, *keys*, **args*)
Store the difference of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

select(*db*)
Select a Ledis db, `db` is integer type

set(*name*, *value*)
Set the value of key `name` to `value`.

set_response_callback(*command*, *callback*)
Set a custom Response Callback

setnx(*name*, *value*)
Set the value of key `name` to `value` if key doesn't exist

sexpire(*name*, *time*)
Set an expire flag on key name for time milliseconds. time can be represented by an integer or a Python timedelta object.

sexpireat(*name*, *when*)
Set an expire flag on key name. when can be represented as an integer representing unix time in milliseconds (unix time * 1000) or a Python datetime object.

sinter(*keys*, **args*)
Return the intersection of sets specified by `keys`

sinterstore(*dest*, *keys*, **args*)
Store the intersection of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sismember(*name*, *value*)
Return a boolean indicating if `value` is a member of set `name`

smclear(**names*)
Delete multiple keys `names` from set

smembers(*name*)
Return all members of the set `name`

spersist(*name*)
Removes an expiration on name

srem(*name*, **values*)
Remove `values` from set `name`

sttl(*name*)
Returns the number of seconds until the key name will expire

sunion(*keys*, **args*)
Return the union of sets specified by `keys`

sunionstore(*dest*, *keys*, **args*)
Store the union of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

ttl(*name*)
Returns the number of seconds until the key `name` will expire

zadd(*name*, **args*, ***kwargs*)
Set any number of score, element-name pairs to the key `name`. Pairs can be specified in two ways:

As *args, in the form of: score1, name1, score2, name2, ... or as **kwargs, in the form of: name1=score1, name2=score2, ...

The following example would add four values to the 'my-key' key: ledis.zadd('my-key', 1.1, 'name1', 2.2, 'name2', name3=3.3, name4=4.4)

zcard(*name*)
Return the number of elements in the sorted set `name`

zclear(*name*)
Delete key of `name` from sorted set

zcount(*name*, *min*, *max*)
Return the number of elements in the sorted set at key `name` with a score between `min` and `max`. The min and max arguments have the same semantic as described for ZRANGEBYSCORE.

zexpire(*name*, *time*)
Set timeout on key `name` with `time`

zexpireat(*name*, *when*)

Set an expire flag on key name for time seconds. time can be represented by an integer or a Python timedelta object.

zincrby(*name*, *value*, *amount=1*)
Increment the score of `value` in sorted set `name` by `amount`

zmclear(**names*)
Delete multiple keys of `names` from sorted set

zpersist(*name*)
Removes an expiration on name

zrange(*name*, *start*, *end*, *desc=False*, *withscores=False*)
Return a range of values from sorted set `name` between `start` and `end` sorted in ascending order.

`start` and `end` can be negative, indicating the end of the range.

`desc` a boolean indicating whether to sort the results descendingly

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

zrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*)
Return a range of values from the sorted set `name` with scores between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

zrank(*name*, *value*)
Returns a 0-based value indicating the rank of `value` in sorted set `name`

zrem(*name*, **values*)
Remove member `values` from sorted set `name`

zremrangebyrank(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with ranks between `min` and `max`. Values are 0-based, ordered from smallest score to largest. Values can be negative indicating the highest scores. Returns the number of elements removed

zremrangebyscore(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with scores between `min` and `max`. Returns the number of elements removed.

zrevrange(*name*, *start*, *num*, *withscores=False*)
Return a range of values from sorted set `name` between `start` and `num` sorted in descending order.

`start` and `num` can be negative, indicating the end of the range.

`withscores` indicates to return the scores along with the values The return type is a list of (value, score) pairs

zrevrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*)
Return a range of values from the sorted set `name` with scores between `min` and `max` in descending order.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

zrevrank(*name*, *value*)
Returns a 0-based value indicating the descending rank of `value` in sorted set `name`

zscore(*name*, *value*)
Return the score of element `value` in sorted set `name`

zttl(*name*)
Returns the number of seconds until the key name will expire

class `ledis.`**ConnectionPool**(*connection_class=<class 'ledis.connection.Connection'>*, *max_connections=None*, ***connection_kwargs*)
Generic connection pool

disconnect()
Disconnects all connections in the pool

classmethod from_url(*url*, *db=None*, ***kwargs*)
Return a connection pool configured from the given URL.

For example:

```
ledis://localhost:6380/0
unix:///path/to/socket.sock?db=0
```

Three URL schemes are supported: ledis:// creates a normal TCP socket connection unix:// creates a Unix Domain Socket connection

There are several ways to specify a database number. The parse function will return the first specified option:

1. A `db` querystring option, e.g. ledis://localhost?db=0
2. If using the ledis:// scheme, the path argument of the url, e.g. ledis://localhost/0
3. The `db` argument to this function.

If none of these options are specified, db=0 is used.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class's initializer. In the case of conflicting arguments, querystring arguments always win.

get_connection(*command_name*, **keys*, ***options*)
Get a connection from the pool

make_connection()
Create a new connection

release(*connection*)
Releases the connection back to the pool

class `ledis.`**BlockingConnectionPool**(*max_connections=50*, *timeout=20*, *connection_class=None*, *queue_class=None*, ***connection_kwargs*)
Thread-safe blocking connection pool:

```
>>> from ledis.client import Ledis
>>> client = Ledis(connection_pool=BlockingConnectionPool())
```

It performs the same function as the default `:py:class: ~ledis.connection.ConnectionPool` implementation, in that, it maintains a pool of reusable connections that can be shared by multiple ledis clients (safely across threads if required).

The difference is that, in the event that a client tries to get a connection from the pool when all of connections are in use, rather than raising a `:py:class: ~ledis.exceptions.ConnectionError` (as the default `:py:class: ~ledis.connection.ConnectionPool` implementation does), it makes the client wait ("blocks") for a specified number of seconds until a connection becomes available.

Use `max_connections` to increase / decrease the pool size:

```
>>> pool = BlockingConnectionPool(max_connections=10)
```

Use `timeout` to tell it either how many seconds to wait for a connection to become available, or to block forever:

Block forever. >>> pool = BlockingConnectionPool(timeout=None)

Raise a `ConnectionError` after five seconds if a connection is # not available. >>> pool = BlockingConnectionPool(timeout=5)

disconnect()
Disconnects all connections in the pool.

get_connection(*command_name*, **keys*, ***options*)
Get a connection, blocking for `self.timeout` until a connection is available from the pool.

If the connection returned is `None` then creates a new connection. Because we use a last-in first-out queue, the existing connections (having been returned to the pool after the initial `None` values were added) will be returned before `None` values. This means we only create new connections when we need to, i.e.: the actual number of connections will only increase in response to demand.

make_connection()
Make a fresh connection.

reinstantiate()
Reinstatiate this instance within a new process with a new connection pool set.

release(*connection*)
Releases the connection back to the pool.

class `ledis.`**Connection**(*host='localhost'*, *port=6380*, *db=0*, *socket_timeout=None*, *encoding='utf-8'*, *encoding_errors='strict'*, *decode_responses=False*, *parser_class=<class 'ledis.connection.PythonParser'>*)
Manages TCP communication to and from a Ledis server

connect()
Connects to the Ledis server if not already connected

disconnect()
Disconnects from the Ledis server

encode(*value*)
Return a bytestring representation of the value

on_connect()
Initialize the connection, authenticate and select a database

pack_command(**args*)
Pack a series of arguments into a value Ledis command

read_response()
Read the response from a previously sent command

send_command(**args*)
Pack and send a command to the Ledis server

send_packed_command(*command*)
Send an already packed command to the Ledis server

`ledis.`**from_url**(*url*, *db=None*, ***kwargs*)
Returns an active Ledis client generated from the given database URL.

Will attempt to extract the database id from the path url fragment, if none is provided.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

l

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

Z