ssdb-py Documentation

Release 0.01

wrongway

January 11, 2016

Contents

CHAPTER 1

What is SSDB?

SSDB is a fast NoSQL database for storing big list of billions of elements.

SSDB is stable, production-ready and is widely used by many Internet companies including QIHU 360. It's repository is https://github.com/ideawu/ssdb

CHAPTER 2

About ssdb-py

ssdb-py is a ssdb python client like redis. It provides two types of connection-pool and a group of functions of ssdb.

It's repository is https://github.com/wrongwaycn/ssdb-py

CHAPTER 3

Quickstart

```
>>> from ssdb import SSDB
>>> import time
>>> ssdb = SSDB(host='127.0.0.1', port=8888)
>>> ssdb.multi_set(set_a='a', set_b='b', set_c='c', set_d='d')
4
>>> ssdb.multi_set(set_x1='x1', set_x2='x2', set_x3='x3', set_x4='x4')
4
>>> ssdb.multi_set(set_abc='abc', set_count=10)
2
>>> ssdb.multi_hset('hash_1', a='A', b='B', c='C', d='D', e='E', f='F',
...                 g='G')
7
>>> ssdb.multi_hset('hash_2',
...                 key1=42,
...                 key2=3.1415926,
...                 key3=-1.41421,
...                 key4=256,
...                 key5='e',
...                 key6='log'
...                )
6
>>> ssdb.multi_zset('zset_1', a=30, b=20, c=100, d=1, e=64, f=-3,
...                 g=0)
7
>>> ssdb.multi_hset('zset_2',
...                 key1=42,
...                 key2=314,
...                 key3=1,
...                 key4=256,
...                 key5=0,
...                 key6=-5
...                )
6
>>> ssdb.get('set_a')
'a'
>>> ssdb.setx('set_ttl', 'ttl', 5)
True
>>> ssdb.get('set_ttl')
'ttl'
>>> time.sleep(5)
>>> ssdb.get('set_ttl')
>>>
>>> ssdb.exists('set_a')
```

```
True
>>> ssdb.incr('set_count', 3)
13
>>> ssdb.multi_get('a', 'b', 'c', 'd')
{'a': 'a', 'c': 'c', 'b': 'b', 'd': 'd'}
>>> ssdb.keys('set_x ', 'set_xx', 3)
['set_x1', 'set_x2', 'set_x3']
>>> ssdb.scan('set_x ', '', 10)
{'set_x1': 'x1', 'set_x2': 'x2', 'set_x3': 'x3', 'set_x4': 'x4'}
>>> ssdb.delete('set_abc')
True
>>> ssdb.hget("hash_1", 'a')
'A'
>>> ssdb.hexists('hash_2', 'key2')
True
>>> ssdb.hdecr('hash_2', 'key1', 7)
36
>>> ssdb.hsize('hash_1')
7
>>> ssdb.hlist('hash_ ', 'hash_z', 10)
['hash_1', 'hash_2']
>>> ssdb.hscan('hash_1', 'a', 'g', 10)
{'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E', 'f': 'F', 'g': 'G'}
>>> ssdb.zget("zset_1", 'b')
20
>>> ssdb.zset("zset_1", 'z', 1024)
True
>>> ssdb.zset_exists('zset_2')
True
>>> ssdb.multi_zget('zset_1', 'a', 'b', 'c', 'd')
{'a': 30, 'c': 100, 'b': 20, 'd': 1}
>>> ssdb.zkeys('zset_1', '', 0, 200, 3)
['g', 'd', 'b']
>>> ssdb.zscan('zset_1', '', 0, 200, 10)
{'g': 0, 'd': 1, 'b': 20, 'a': 30, 'e': 64, 'c': 100}
>>> ssdb.zrscan('zset_1', 'a', 30, -1000, 3)
{'b': 20, 'd': 1, 'g': 0}
>>> ssdb.zrank('zset_1','d')
2
>>> ssdb.zrrange('zset_1', 0, 4)
{'c': 100, 'e': 64, 'a': 30, 'b': 20}
```

CHAPTER 4

Content

4.1 Connection

class ssdb.connection.**Connection**(*host='127.0.0.1'*, *port=8888*, *socket_timeout=None*, *socket_connect_timeout=None*, *socket_keepalive=False*, *socket_keepalive_options=None*, *retry_on_timeout=False*, *encoding='utf-8'*, *encoding_errors='strict'*, *decode_responses=False*, *parser_class=<class 'ssdb.connection.PythonParser'>*, *socket_read_size=65536*)

Manages TCP communication to and from a SSDB server

```
>>> from ssdb.connection import Connection
>>> conn = Connection(host='localhost', port=8888)
```

class ssdb.connection.**ConnectionPool**(*connection_class=<class 'ssdb.connection.Connection'>*, *max_connections=None*, ***connection_kwargs*)

Generic connection pool.

```
>>> from ssdb.client import SSDB
>>> client = SSDB(connection_pool=ConnectionPool())
```

If max_connections is set, then this objectraises ssdb.ConnectionError when the pool's limit is reached. By default, TCP connections are created connection_class is specified. Any additionan keyword arguments are passed to the constructor of connection_class.

class ssdb.connection.**BlockingConnectionPool**(*max_connections=50*, *timeout=20*, *connection_class=<class 'ssdb.connection.Connection'>*, *queue_class=<class Queue.LifoQueue>*, ***connection_kwargs*)

Thread-safe blocking connection pool:

```
>>> from ssdb.client import SSDB
>>> client = SSDB(connection_pool=BlockingConnectionPool())
```

It performs the same function as default :py:class: ~ssdb.connection.ConnectionPool implementation, in that, it maintains a pool of reusable connections that can be shared by multiple ssdb 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: ~ssdb.exceptions.ConnectionError (as the default :py:class: ~ssdb.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)
```

4.2 Client

class ssdb.client.**SSDB**(*host='localhost'*, *port=8888*, *socket_timeout=None*, *connection_pool=None*, *charset='utf-8'*, *errors='strict'*, *decode_responses=False*)

Provides backwards compatibility with older versions of ssdb-py(1.6.6) that changed arguments to some commands to be more Pythonic, sane, or by accident.

class ssdb.client.**StrictSSDB**(*host='localhost'*, *port=8888*, *socket_timeout=None*, *connection_pool=None*, *charset='utf-8'*, *errors='strict'*, *decode_responses=False*)

Implementation of the SSDB protocol.

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

Connection derives from this, implementing how the commands are sent and received to the SSDB server.

4.2.1 Key/Value

A container of (key, value) pairs in ssdb. A key name maps a string value.

```
>>> from ssdb.client import SSDB
>>> ssdb = SSDB()
>>> ssdb.multi_set(set_a='a', set_b='b', set_c='c', set_d='d')
>>> ssdb.multi_set(set_x1='x1', set_x2='x2', set_x3='x3', set_x4='x4')
>>> ssdb.multi_set(set_abc='abc', set_count=10)
```

get

StrictSSDB.**get**(*name*)

Return the value at key `name`, or `None` if the key doesn't exist

Like **Redis.GET**

Parameters **name** (*string*) – the key name

Returns the value at key `name`, or `None` if the key doesn't exist

Return type string

```
>>> ssdb.get("set_abc")
'abc'
>>> ssdb.get("set_a")
'a'
>>> ssdb.get("set_b")
'b'
>>> ssdb.get("not_exists_abc")
>>>
```

getset

StrictSSDB.**getset**(*name*, *value*)

Set the value at key `name` to `value` if key doesn't exist Return the value at key `name` atomically.

Like **Redis.GETSET**

Parameters

- **name** (*string*) – the key name
- **value** (*string*) – a string or an object can be converted to string

Returns `True` on success, `False` if not

Return type bool

```
>>> ssdb.getset("getset_a", 'abc')
None
>>> ssdb.getset("getset_a", 'def')
'abc'
>>> ssdb.getset("getset_a", 'ABC')
'def'
>>> ssdb.getset("getset_a", 123)
'ABC'
```

set

StrictSSDB.**set**(*name*, *value*)

Set the value at key `name` to `value` .

Like **Redis.SET**

Parameters

- **name** (*string*) – the key name
- **value** (*string*) – a string or an object can be converted to string

Returns `True` on success, `False` if not

Return type bool

```
>>> ssdb.set("set_cde", 'cde')
True
>>> ssdb.set("set_cde", 'test')
True
>>> ssdb.set("hundred", 100)
True
```

add

The same is *set*.

setnx

StrictSSDB.**setnx**(*name*, *value*)

Set the value at key `name` to `value` if and only if the key doesn't exist.

Like **Redis.SETNX**

Parameters

- **name** (*string*) – the key name
- **value** (*string*) – a string or an object can be converted to string

Returns `True` on success, `False` if not

Return type bool

```
>>> ssdb.setnx("setnx_test", 'abc')
True
>>> ssdb.setnx("setnx_test", 'cde')
False
```

expire

StrictSSDB.**expire**(*name*, *ttl*)

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

Like **Redis.EXPIRE**

Note: `Expire` **only expire** the *Key/Value* .

Parameters

- **name** (*string*) – the key name
- **ttl** (*int*) – number of seconds to live

Returns `True` on success, or `False` if the key doesn't exist or failure

Return type bool

```
>>> ssdb.expire('set_abc', 6)
True
>>> ssdb.expire('not_exist')
False
```

ttl

StrictSSDB.**ttl**(*name*)

Returns the number of seconds until the key `name` will expire.

Like **Redis.TTL**

Note: `ttl` **can only** be used to the *Key/Value* .

Parameters **name** (*string*) – the key name

Returns the number of seconds, or `-1` if the key doesn't exist or have no ttl

Return type int

```
>>> ssdb.ttl('set_abc')
6
>>> ssdb.ttl('not_exist')
-1
```

setx

`SSDB.`**`setx`**(*name*, *value*, *ttl*)

Set the value of key `name` to `value` that expires in `ttl` seconds. `ttl` can be represented by an integer or a Python timedelta object.

Like **Redis.SETEX**

Parameters

- **name** (*string*) – the key name
- **value** (*string*) – a string or an object can be converted to string
- **ttl** (*int*) – positive int seconds or timedelta object

Returns `True` on success, `False` if not

Return type bool

```
>>> import time
>>> ssdb.set("test_ttl", 'ttl', 4)
True
>>> ssdb.get("test_ttl")
'ttl'
>>> time.sleep(4)
>>> ssdb.get("test_ttl")
>>>
```

delete

`StrictSSDB.`**`delete`**(*name*)

Delete the key specified by `name` .

Like **Redis.DELETE**

Note: `Delete` **can't delete** the *Hash* or *Zsets*, use *hclear* for *Hash* and *zclear* for *Zsets*

Parameters **name** (*string*) – the key name

Returns `True` on deleting successfully, or `False` if the key doesn't exist or failure

Return type bool

```
>>> ssdb.delete('set_abc')
True
>>> ssdb.delete('set_a')
True
>>> ssdb.delete('set_abc')
False
>>> ssdb.delete('not_exist')
False
```

remove

The same is *delete*.

exists

`StrictSSDB.`**`exists`**`(`*name*`)`

Return a boolean indicating whether key `name` exists.

Like **Redis.EXISTS**

Note: `exists` **can't indicate** whether any *Hash*, *Zsets* or *Queue* exists, use *hash_exists* for *Hash* , *zset_exists* for *Zsets* and *queue_exists* for *Queue* .

Parameters **`name`** (*string*) – the key name

Returns `True` if the key exists, `False` if not

Return type bool

```
>>> ssdb.exists('set_abc')
True
>>> ssdb.exists('set_a')
True
>>> ssdb.exists('not_exist')
False
```

incr

`StrictSSDB.`**`incr`**`(`*name*, *amount=1*`)`

Increase the value at key `name` by `amount`. If no key exists, the value will be initialized as `amount` .

Like **Redis.INCR**

Parameters

- **`name`** (*string*) – the key name
- **`amount`** (*int*) – increments

Returns the integer value at key `name`

Return type int

```
>>> ssdb.incr('set_count', 3)
13
>>> ssdb.incr('set_count', 1)
14
>>> ssdb.incr('set_count', -2)
12
>>> ssdb.incr('temp_count', 42)
42
```

decr

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

Decrease the value at key `name` by `amount`. If no key exists, the value will be initialized as 0 - `amount` .

Like **Redis.DECR**

Parameters

- **name** (*string*) – the key name
- **amount** (*int*) – decrements

Returns the integer value at key `name`

Return type int

```
>>> ssdb.decr('set_count', 3)
7
>>> ssdb.decr('set_count', 1)
6
>>> ssdb.decr('temp_count', 42)
-42
```

getbit

StrictSSDB.**getbit**(*name*, *offset*)

Returns a boolean indicating the value of `offset` in `name`

Like **Redis.GETBIT**

Parameters

- **name** (*string*) – the key name
- **offset** (*int*) – the bit position
- **val** (*bool*) – the bit value

Returns the bit at the `offset` , `False` if key doesn't exist or offset exceeds the string length.

Return type bool

```
>>> ssdb.set('bit_test', 1)
True
>>> ssdb.getbit('bit_test', 0)
True
>>> ssdb.getbit('bit_test', 1)
False
```

setbit

StrictSSDB.**setbit**(*name*, *offset*, *val*)

Flag the `offset` in `name` as `value`. Returns a boolean indicating the previous value of `offset`.

Like **Redis.SETBIT**

Parameters

- **name** (*string*) – the key name
- **offset** (*int*) – the bit position
- **val** (*bool*) – the bit value

Returns the previous bit (False or True) at the `offset`

Return type bool

```
>>> ssdb.set('bit_test', 1)
True
>>> ssdb.setbit('bit_test', 1, 1)
False
>>> ssdb.get('bit_test')
3
>>> ssdb.setbit('bit_test', 2, 1)
False
>>> ssdb.get('bit_test')
7
```

countbit

StrictSSDB.**countbit**(*name*, *start=None*, *size=None*)

Returns the count of set bits in the value of `key`. Optional `start` and `size` paramaters indicate which bytes to consider.

Similiar with **Redis.BITCOUNT**

Parameters

- **name** (*string*) – the key name
- **start** (*int*) – Optional, if start is negative, count from start'th character from the end of string.
- **size** (*int*) – Optional, if size is negative, then that many characters will be omitted from the end of string.

Returns the count of the bit 1

Return type int

```
>>> ssdb.set('bit_test', 1)
True
>>> ssdb.countbit('bit_test')
3
>>> ssdb.set('bit_test','1234567890')
True
>>> ssdb.countbit('bit_test', 0, 1)
3
>>> ssdb.countbit('bit_test', 3, -3)
16
```

substr

StrictSSDB.**substr**(*name*, *start=None*, *size=None*)

Return a substring of the string at key `name`. `start` and `size` are 0-based integers specifying the portion of the string to return.

Like **Redis.SUBSTR**

Parameters

- **name** (*string*) – the key name
- **start** (*int*) – Optional, the offset of first byte returned. If start is negative, the returned string will start at the start'th character from the end of string.
- **size** (*int*) – Optional, number of bytes returned. If size is negative, then that many characters will be omitted from the end of string.

Returns The extracted part of the string.

Return type string

```
>>> ssdb.set('str_test', 'abc12345678')
True
>>> ssdb.substr('str_test', 2, 4)
'c123'
>>> ssdb.substr('str_test', -2, 2)
'78'
>>> ssdb.substr('str_test', 1, -1)
'bc1234567'
```

strlen

StrictSSDB.**strlen**(*name*)

Return the number of bytes stored in the value of `name`

Like **Redis.STRLEN**

Parameters **name** (*string*) – the key name

Returns The number of bytes of the string, if key not exists, returns 0.

Return type int

```
>>> ssdb.set('str_test', 'abc12345678')
True
>>> ssdb.strlen('str_test')
11
```

multi_set

StrictSSDB.**multi_set**(***kvs*)

Set key/value based on a mapping dictionary as kwargs.

Like **Redis.MSET**

Parameters **kvs** (*dict*) – a key/value mapping dict

Returns the number of successful operation

Return type int

```
>>> ssdb.multi_set(set_a='a', set_b='b', set_c='c', set_d='d')
4
>>> ssdb.multi_set(set_abc='abc',set_count=10)
2
```

mset

The same is *multi_set*.

multi_get

StrictSSDB.**multi_get**(**names*)

Return a dictionary mapping key/value by names

Like **Redis.MGET**

Parameters **names** (*list*) – a list of keys

Returns a dict mapping key/value

Return type dict

```
>>> ssdb.multi_get('a', 'b', 'c', 'd')
{'a': 'a', 'c': 'c', 'b': 'b', 'd': 'd'}
>>> ssdb.multi_get('set_abc','set_count')
{'set_abc': 'set_abc', 'set_count': '10'}
```

mget

The same is *multi_get*.

multi_del

StrictSSDB.**multi_del**(**names*)

Delete one or more keys specified by names

Like **Redis.DELETE**

Parameters **names** (*list*) – a list of keys

Returns the number of successful deletion

Return type int

```
>>> ssdb.multi_del('a', 'b', 'c', 'd')
4
>>> ssdb.multi_del('set_abc','set_count')
2
```

mdel

The same is *multi_del*.

keys

StrictSSDB.**keys**(*name_start*, *name_end*, *limit=10*)

Return a list of the top `limit` keys between `name_start` and `name_end`

Similiar with **Redis.KEYS**

Note: The range is (`name_start`, `name_end`]. `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf
- **name_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of keys

Return type list

```
>>> ssdb.keys('set_x1', 'set_x3', 10)
['set_x2', 'set_x3']
>>> ssdb.keys('set_x ', 'set_xx', 3)
['set_x1', 'set_x2', 'set_x3']
>>> ssdb.keys('set_x ', '', 3)
['set_x1', 'set_x2', 'set_x3', 'set_x4']
>>> ssdb.keys('set_zzzzz ', '', )
[]
```

scan

StrictSSDB.**scan**(*name_start*, *name_end*, *limit=10*)

Scan and return a dict mapping key/value in the top `limit` keys between `name_start` and `name_end` in ascending order

Similiar with **Redis.SCAN**

Note: The range is (`name_start`, `name_end`]. `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf
- **name_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/value in ascending order

Return type OrderedDict

```
    >>> ssdb.scan('set_x1', 'set_x3', 10)
    {'set_x2': 'x2', 'set_x3': 'x3'}
    >>> ssdb.scan('set_x ', 'set_xx', 3)
    {'set_x1': 'x1', 'set_x2': 'x2', 'set_x3': 'x3'}
    >>> ssdb.scan('set_x ', '', 10)
    {'set_x1': 'x1', 'set_x2': 'x2', 'set_x3': 'x3', 'set_x4': 'x4'}
    >>> ssdb.scan('set_zzzzz ', '', 10)
    {}
```

rscan

StrictSSDB.**rscan**(*name_start*, *name_end*, *limit=10*)

Scan and return a dict mapping key/value in the top `limit` keys between `name_start` and `name_end` in descending order

Note: The range is (`name_start`, `name_end`]. `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The upper bound(not included) of keys to be returned, empty string `''` means +inf
- **name_end** (*string*) – The lower bound(included) of keys to be returned, empty string `''` means -inf
- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/value in descending order

Return type OrderedDict

```
    >>> ssdb.scan('set_x3', 'set_x1', 10)
    {'set_x2': 'x2', 'set_x1': 'x1'}
    >>> ssdb.scan('set_xx', 'set_x ', 3)
    {'set_x4': 'x4', 'set_x3': 'x3', 'set_x2': 'x2'}
    >>> ssdb.scan('', 'set_x ', 10)
    {'set_x4': 'x4', 'set_x3': 'x3', 'set_x2': 'x2', 'set_x1': 'x1'}
    >>> ssdb.scan('', 'set_zzzzz', 10)
    {}
```

4.2.2 Hash

A container of (key, dict) pairs in ssdb. A hash name maps a dict which contains key/value pairs

```
>>> from ssdb.client import SSDB
>>> ssdb = SSDB()
>>> ssdb.multi_hset('hash_1', a='A', b='B', c='C', d='D', e='E', f='F',
...                 g='G')
>>> ssdb.multi_hset('hash_2',
...                 key1=42,
...                 key2=3.1415926,
...                 key3=-1.41421,
...                 key4=256,
...                 key5='e',
...                 key6='log'
...                )
```

hget

StrictSSDB.**hget**(*name*, *key*)

Get the value of `key` within the hash `name`

Like **Redis.HGET**

Parameters

- **name** (*string*) – the hash name
- **key** (*string*) – the key name

Returns the value at `key` within hash `name` , or `None` if the `name` or `key` doesn't exist

Return type string

```
>>> ssdb.hget("hash_1", 'a')
'A'
>>> ssdb.hget("hash_1", 'b')
'B'
>>> ssdb.hget("hash_1", 'z')
>>>
>>> ssdb.hget("hash_2", 'key1')
'42'
```

hset

StrictSSDB.**hset**(*name*, *key*, *value*)

Set the value of `key` within the hash `name` to `value`

Like **Redis.HSET**

Parameters

- **name** (*string*) – the hash name
- **key** (*string*) – the key name
- **value** (*string*) – a string or an object can be converted to string

Returns `True` if `hset` created a new field, otherwise `False`

Return type bool

```
>>> ssdb.hset("hash_3", 'yellow', '#FFFF00')
True
>>> ssdb.hset("hash_3", 'red', '#FF0000')
True
>>> ssdb.hset("hash_3", 'blue', '#0000FF')
True
>>> ssdb.hset("hash_3", 'yellow', '#FFFF00')
False
```

hadd

The same is *hadd*.

hclear

StrictSSDB.**hclear**(*name*)

Clear&Delete the hash specified by `name`

Parameters **name** (*string*) – the hash name

Returns the length of removed elements

Return type int

```
>>> ssdb.hclear('hash_1')
7
>>> ssdb.hclear('hash_1')
0
>>> ssdb.hclear('hash_2')
6
>>> ssdb.hclear('not_exist')
0
```

hdel

StrictSSDB.**hdel**(*name*, *key*)

Remove the `key` from hash `name`

Like **Redis.HDEL**

Parameters

- **name** (*string*) – the hash name
- **key** (*string*) – the key name

Returns `True` if deleted successfully, otherwise `False`

Return type bool

```
>>> ssdb.hdel("hash_2", 'key1')
True
>>> ssdb.hdel("hash_2", 'key2')
True
>>> ssdb.hdel("hash_2", 'key3')
True
>>> ssdb.hdel("hash_2", 'key_not_exist')
False
>>> ssdb.hdel("hash_not_exist", 'key1')
False
```

hremove

The same is *hdel*.

hash_exists

StrictSSDB.**hash_exists**(*name*)

Return a boolean indicating whether hash `name` exists

Parameters **name** (*string*) – the hash name

Returns `True` if the hash exists, `False` if not

Return type string

```
>>> ssdb.hash_exists('hash_1')
True
>>> ssdb.hash_exists('hash_2')
True
>>> ssdb.hash_exists('hash_not_exist')
False
```

hexists

`StrictSSDB.`**`hexists`**(*name*, *key*)

Return a boolean indicating whether `key` exists within hash `name`

Like **Redis.HEXISTS**

Parameters

- **`name`** (*string*) – the hash name
- **`key`** (*string*) – the key name

Returns `True` if the key exists, `False` if not

Return type bool

```
>>> ssdb.hexists('hash_1', 'a')
True
>>> ssdb.hexists('hash_2', 'key2')
True
>>> ssdb.hexists('hash_not_exist', 'a')
False
>>> ssdb.hexists('hash_1', 'z_not_exists')
False
>>> ssdb.hexists('hash_not_exist', 'key_not_exists')
False
```

hincr

`StrictSSDB.`**`hincr`**(*name*, *key*, *amount=1*)

Increase the value of `key` in hash `name` by `amount`. If no key exists, the value will be initialized as `amount`

Like **Redis.HINCR**

Parameters

- **`name`** (*string*) – the hash name
- **`key`** (*string*) – the key name
- **`amount`** (*int*) – increments

Returns the integer value of `key` in hash `name`

Return type int

```
>>> ssdb.hincr('hash_2', 'key1', 7)
49
>>> ssdb.hincr('hash_2', 'key2', 3)
6
```

```
>>> ssdb.hincr('hash_2', 'key_not_exists', 101)
101
>>> ssdb.hincr('hash_not_exists', 'key_not_exists', 8848)
8848
```

hdecr

StrictSSDB.**hdecr**(*name*, *key*, *amount=1*)

Decrease the value of `key` in hash `name` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

Parameters

- **name** (*string*) – the hash name
- **key** (*string*) – the key name
- **amount** (*int*) – increments

Returns the integer value of `key` in hash `name`

Return type int

```
>>> ssdb.hdecr('hash_2', 'key1', 7)
35
>>> ssdb.hdecr('hash_2', 'key2', 3)
0
>>> ssdb.hdecr('hash_2', 'key_not_exists', 101)
-101
>>> ssdb.hdecr('hash_not_exists', 'key_not_exists', 8848)
-8848
```

hsize

StrictSSDB.**hsize**(*name*)

Return the number of elements in hash `name`

Like **Redis.HLEN**

Parameters **name** (*string*) – the hash name

Returns the size of hash *name'*

Return type int

```
>>> ssdb.hsize('hash_1')
7
>>> ssdb.hsize('hash_2')
6
>>> ssdb.hsize('hash_not_exists')
0
```

hlen

The same is *hsize*.

multi_hget

StrictSSDB.**multi_hget**(*name*, **keys*)

Return a dictionary mapping key/value by `keys` from hash `names`

Like **Redis.HMGET**

Parameters

- **name** (*string*) – the hash name
- **keys** (*list*) – a list of keys

Returns a dict mapping key/value

Return type dict

```
>>> ssdb.multi_hget('hash_1', 'a', 'b', 'c', 'd')
{'a': 'A', 'c': 'C', 'b': 'B', 'd': 'D'}
>>> ssdb.multi_hget('hash_2', 'key2', 'key5')
{'key2': '3.1415926', 'key5': 'e'}
```

hmget

The same is *multi_hget*.

multi_hset

StrictSSDB.**multi_hset**(*name*, ***kvs*)

Set key to value within hash `name` for each corresponding key and value from the `kvs` dict.

Like **Redis.HMSET**

Parameters

- **name** (*string*) – the hash name
- **keys** (*list*) – a list of keys

Returns the number of successful creation

Return type int

```
>>> ssdb.multi_hset('hash_4', a='AA', b='BB', c='CC', d='DD')
4
>>> ssdb.multi_hset('hash_4', a='AA', b='BB', c='CC', d='DD')
0
>>> ssdb.multi_hset('hash_4', a='AA', b='BB', c='CC', d='DD', e='EE')
1
```

hmset

The same is *multi_hset*.

multi_hdel

StrictSSDB.**multi_hdel**(*name*, **keys*)

Remove `keys` from hash `name`

Like **Redis.HMDEL**

Parameters

- **name** (*string*) – the hash name
- **keys** (*list*) – a list of keys

Returns the number of successful deletion

Return type int

```
>>> ssdb.multi_hdel('hash_1', 'a', 'b', 'c', 'd')
4
>>> ssdb.multi_hdel('hash_1', 'a', 'b', 'c', 'd')
0
>>> ssdb.multi_hdel('hash_2', 'key2_not_exist', 'key5_not_exist')
0
```

hmdel

The same is *multi_hdel*.

hlist

StrictSSDB.**hlist**(*name_start*, *name_end*, *limit=10*)

Return a list of the top `limit` hash's name between `name_start` and `name_end` in ascending order

Note: The range is (`name_start`, `name_end`]. The `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of hash names to be returned, empty string `''` means -inf
- **name_end** (*string*) – The upper bound(included) of hash names to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of hash's name

Return type list

```
>>> ssdb.hlist('hash_ ', 'hash_z', 10)
['hash_1', 'hash_2']
>>> ssdb.hlist('hash_ ', '', 3)
['hash_1', 'hash_2']
>>> ssdb.hlist('', 'aaa_not_exist', 10)
[]
```

hrlist

StrictSSDB.**hrlist**(*name_start*, *name_end*, *limit=10*)
Return a list of the top `limit` hash's name between `name_start` and `name_end` in descending order

Note: The range is (`name_start`, `name_end`]. The `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of hash names to be returned, empty string `''` means +inf
- **name_end** (*string*) – The upper bound(included) of hash names to be returned, empty string `''` means -inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of hash's name

Return type list

```
>>> ssdb.hrlist('hash_ ', 'hash_z', 10)
['hash_2', 'hash_1']
>>> ssdb.hrlist('hash_ ', '', 3)
['hash_2', 'hash_1']
>>> ssdb.hrlist('', 'aaa_not_exist', 10)
[]
```

hkeys

StrictSSDB.**hkeys**(*name*, *key_start*, *key_end*, *limit=10*)
Return a list of the top `limit` keys between `key_start` and `key_end` in hash `name`

Similiar with **Redis.HKEYS**

Note: The range is (`key_start`, `key_end`]. The `key_start` isn't in the range, but `key_end` is.

Parameters

- **name** (*string*) – the hash name
- **key_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf
- **key_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of keys

Return type list

```
>>> ssdb.hkeys('hash_1', 'a', 'g', 10)
['b', 'c', 'd', 'e', 'f', 'g']
>>> ssdb.hkeys('hash_2', 'key ', 'key4', 3)
['key1', 'key2', 'key3']
>>> ssdb.hkeys('hash_1', 'f', '', 10)
```

```
    ['g']
    >>> ssdb.hkeys('hash_2', 'keys', '', 10)
    []
```

hscan

StrictSSDB.**hscan**(*name*, *key_start*, *key_end*, *limit=10*)

Return a dict mapping key/value in the top `limit` keys between `key_start` and `key_end` within hash `name` in ascending order

Similiar with **Redis.HSCAN**

Note: The range is (`key_start`, `key_end`]. The `key_start` isn't in the range, but `key_end` is.

Parameters

- **name** (*string*) – the hash name
- **key_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf
- **key_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/value in ascending order

Return type OrderedDict

```
    >>> ssdb.hscan('hash_1', 'a', 'g', 10)
    {'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E', 'f': 'F', 'g': 'G'}
    >>> ssdb.hscan('hash_2', 'key ', 'key4', 3)
    {'key1': '42', 'key2': '3.1415926', 'key3': '-1.41421'}
    >>> ssdb.hscan('hash_1', 'f', '', 10)
    {'g': 'G'}
    >>> ssdb.hscan('hash_2', 'keys', '', 10)
    {}
```

hrscan

StrictSSDB.**hrscan**(*name*, *key_start*, *key_end*, *limit=10*)

Return a dict mapping key/value in the top `limit` keys between `key_start` and `key_end` within hash `name` in descending order

Note: The range is (`key_start`, `key_end`]. The `key_start` isn't in the range, but `key_end` is.

Parameters

- **name** (*string*) – the hash name
- **key_start** (*string*) – The upper bound(not included) of keys to be returned, empty string `''` means +inf
- **key_end** (*string*) – The lower bound(included) of keys to be returned, empty string `''` means -inf

- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/value in descending order

Return type OrderedDict

```
>>> ssdb.hrscan('hash_1', 'g', 'a', 10)
{'f': 'F', 'e': 'E', 'd': 'D', 'c': 'C', 'b': 'B', 'a': 'A'}
>>> ssdb.hrscan('hash_2', 'key7', 'key1', 3)
{'key6': 'log', 'key5': 'e', 'key4': '256'}
>>> ssdb.hrscan('hash_1', 'c', '', 10)
{'b': 'B', 'a': 'A'}
>>> ssdb.hscan('hash_2', 'keys', '', 10)
{}
```

4.2.3 Zsets

A sorted set in ssdb. It's contain keys with scores in `zset`

```
>>> from ssdb.client import SSDB
>>> ssdb = SSDB()
>>> ssdb.multi_zset('zset_1', a=30, b=20, c=100, d=1, e=64, f=-3,
...                 g=0)
>>> ssdb.multi_zset('zset_2',
...                 key1=42,
...                 key2=314,
...                 key3=1,
...                 key4=256,
...                 key5=0,
...                 key6=-5
...                )
```

zget

`StrictSSDB.`**`zget`**(*name*, *key*)

Return the score of element `key` in sorted set `name`

Like **Redis.ZSCORE**

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name

Returns the score, `None` if the zset `name` or the `key` doesn't exist

Return type int

```
>>> ssdb.zget("zset_1", 'a')
30
>>> ssdb.zget("zset_1", 'b')
20
>>> ssdb.zget("zset_1", 'z')
>>>
>>> ssdb.zget("zset_2", 'key1')
42
```

zset

StrictSSDB.**zset**(*name*, *key*, *score=1*)

Set the score of `key` from the zset `name` to `score`

Like **Redis.ZADD**

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name
- **score** (*int*) – the score for ranking

Returns `True` if `zset` created a new score, otherwise `False`

Return type bool

```
>>> ssdb.zset("zset_1", 'z', 1024)
True
>>> ssdb.zset("zset_1", 'a', 1024)
False
>>> ssdb.zset("zset_2", 'key_10', -4)
>>>
>>> ssdb.zget("zset_2", 'key1')
42
```

zadd

The same is *zset*.

zclear

StrictSSDB.**zclear**(*name*)

Clear&Delete the zset specified by `name`

Parameters **name** (*string*) – the zset name

Returns the length of removed elements

Return type int

```
>>> ssdb.zclear('zset_1')
7
>>> ssdb.zclear('zset_1')
0
>>> ssdb.zclear('zset_2')
6
>>> ssdb.zclear('not_exist')
0
```

zdel

StrictSSDB.**zdel**(*name*, *key*)

Remove the specified `key` from zset `name`

Like **Redis.ZREM**

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name

Returns `True` if deleted success, otherwise `False`

Return type bool

```
>>> ssdb.zdel("zset_2", 'key1')
True
>>> ssdb.zdel("zset_2", 'key2')
True
>>> ssdb.zdel("zset_2", 'key3')
True
>>> ssdb.zdel("zset_2", 'key_not_exist')
False
>>> ssdb.zdel("zset_not_exist", 'key1')
False
```

zremove

The same is *zdel*.

zset_exists

`StrictSSDB.`**`zset_exists`**(*name*)

Return a boolean indicating whether zset `name` exists

Parameters **name** (*string*) – the zset name

Returns `True` if the zset exists, `False` if not

Return type string

```
>>> ssdb.zset_exists('zset_1')
True
>>> ssdb.zset_exists('zset_2')
True
>>> ssdb.zset_exists('zset_not_exist')
False
```

zexists

`StrictSSDB.`**`zexists`**(*name*, *key*)

Return a boolean indicating whether `key` exists within zset `name`

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name

Returns `True` if the key exists, `False` if not

Return type bool

```
>>> ssdb.zexists('zset_1', 'a')
True
>>> ssdb.zexists('zset_2', 'key2')
True
>>> ssdb.zexists('zset_not_exist', 'a')
False
>>> ssdb.zexists('zset_1', 'z_not_exists')
False
>>> ssdb.zexists('zset_not_exist', 'key_not_exists')
False
```

zincr

StrictSSDB.**zincr**(*name*, *key*, *amount=1*)

Increase the score of `key` in zset `name` by `amount`. If no key exists, the value will be initialized as `amount`

Like **Redis.ZINCR**

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name
- **amount** (*int*) – increments

Returns the integer value of `key` in zset `name`

Return type int

```
>>> ssdb.zincr('zset_2', 'key1', 7)
49
>>> ssdb.zincr('zset_2', 'key2', 3)
317
>>> ssdb.zincr('zset_2', 'key_not_exists', 101)
101
>>> ssdb.zincr('zset_not_exists', 'key_not_exists', 8848)
8848
```

zdecr

StrictSSDB.**zdecr**(*name*, *key*, *amount=1*)

Decrease the value of `key` in zset `name` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name
- **amount** (*int*) – increments

Returns the integer value of `key` in zset `name`

Return type int

```
>>> ssdb.zdecr('zset_2', 'key1', 7)
36
>>> ssdb.zdecr('zset_2', 'key2', 3)
```

```
    311
    >>> ssdb.zdecr('zset_2', 'key_not_exists', 101)
    -101
    >>> ssdb.zdecr('zset_not_exists', 'key_not_exists', 8848)
    -8848
```

zsize

StrictSSDB.**zsize**(*name*)

Return the number of elements in zset `name`

Like **Redis.ZCARD**

Parameters **name** (*string*) – the zset name

Returns the size of zset *name'*

Return type int

```
    >>> ssdb.zsize('zset_1')
    7
    >>> ssdb.zsize('zset_2')
    6
    >>> ssdb.zsize('zset_not_exists')
    0
```

zlen

The same is *zsize*.

zcard

The same is *zsize*.

multi_zget

StrictSSDB.**multi_zget**(*name*, **keys*)

Return a dictionary mapping key/value by `keys` from zset `names`

Parameters

- **name** (*string*) – the zset name
- **keys** (*list*) – a list of keys

Returns a dict mapping key/value

Return type dict

```
    >>> ssdb.multi_zget('zset_1', 'a', 'b', 'c', 'd')
    {'a': 30, 'c': 100, 'b': 20, 'd': 1}
    >>> ssdb.multi_zget('zset_2', 'key2', 'key5')
    {'key2': 314, 'key5': 0}
```

zmget

The same is *multi_zget*.

multi_zset

StrictSSDB.**multi_zset**(*name*, ***kvs*)

Return a dictionary mapping key/value by `keys` from zset `names`

Parameters

- **name** (*string*) – the zset name
- **keys** (*list*) – a list of keys

Returns the number of successful creation

Return type int

```
>>> ssdb.multi_zset('zset_4', a=100, b=80, c=90, d=70)
4
>>> ssdb.multi_zset('zset_4', a=100, b=80, c=90, d=70)
0
>>> ssdb.multi_zset('zset_4', a=100, b=80, c=90, d=70, e=60)
1
```

zmget

The same is *multi_zset*.

multi_zdel

StrictSSDB.**multi_zdel**(*name*, **keys*)

Remove `keys` from zset `name`

Parameters

- **name** (*string*) – the zset name
- **keys** (*list*) – a list of keys

Returns the number of successful deletion

Return type int

```
>>> ssdb.multi_zdel('zset_1', 'a', 'b', 'c', 'd')
4
>>> ssdb.multi_zdel('zset_1', 'a', 'b', 'c', 'd')
0
>>> ssdb.multi_zdel('zset_2', 'key2_not_exist', 'key5_not_exist')
0
```

zmdel

The same is *multi_zdel*.

zlist

StrictSSDB.**zlist**(*name_start*, *name_end*, *limit=10*)
Return a list of the top `limit` zset's name between `name_start` and `name_end` in ascending order

Note: The range is (`name_start`, `name_end`]. The `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of zset names to be returned, empty string `''` means -inf
- **name_end** (*string*) – The upper bound(included) of zset names to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of zset's name

Return type list

```
>>> ssdb.zlist('zset_ ', 'zset_z', 10)
['zset_1', 'zset_2']
>>> ssdb.zlist('zset_ ', '', 3)
['zset_1', 'zset_2']
>>> ssdb.zlist('', 'aaa_not_exist', 10)
[]
```

zrlist

StrictSSDB.**zrlist**(*name_start*, *name_end*, *limit=10*)
Return a list of the top `limit` zset's name between `name_start` and `name_end` in descending order

Note: The range is (`name_start`, `name_end`]. The `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of zset names to be returned, empty string `''` means +inf
- **name_end** (*string*) – The upper bound(included) of zset names to be returned, empty string `''` means -inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of zset's name

Return type list

```
>>> ssdb.zlist('zset_ ', 'zset_z', 10)
['zset_2', 'zset_1']
>>> ssdb.zlist('zset_ ', '', 3)
['zset_2', 'zset_1']
>>> ssdb.zlist('', 'aaa_not_exist', 10)
[]
```

zkeys

StrictSSDB.**zkeys**(*name*, *key_start*, *score_start*, *score_end*, *limit=10*)

Return a list of the top `limit` keys after `key_start` from zset `name` with scores between `score_start` and `score_end`

Note: The range is (`key_start``+``score_start`, `key_end`]. That means (key.score == score_start && key > key_start || key.score > score_start)

Parameters

- **name** (*string*) – the zset name
- **key_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf
- **key_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of keys

Return type list

```
>>> ssdb.zkeys('zset_1', '', 0, 200, 10)
['g', 'd', 'b', 'a', 'e', 'c']
>>> ssdb.zkeys('zset_1', '', 0, 200, 3)
['g', 'd', 'b']
>>> ssdb.zkeys('zset_1', 'b', 20, 200, 3)
['a', 'e', 'c']
>>> ssdb.zkeys('zset_1', 'c', 100, 200, 3)
[]
```

zscan

StrictSSDB.**zscan**(*name*, *key_start*, *score_start*, *score_end*, *limit=10*)

Return a dict mapping key/score of the top `limit` keys after `key_start` with scores between `score_start` and `score_end` in zset `name` in ascending order

Similiar with **Redis.ZSCAN**

Note: The range is (`key_start``+``score_start`, `key_end`]. That means (key.score == score_start && key > key_start || key.score > score_start)

Parameters

- **name** (*string*) – the zset name
- **key_start** (*string*) – The key related to score_start, could be empty string `''`
- **score_start** (*int*) – The minimum score related to keys(may not be included, depend on key_start), empty string `''` means -inf
- **score_end** (*int*) – The maximum score(included) related to keys, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/score in ascending order

Return type OrderedDict

```
>>> ssdb.zscan('zset_1', '', 0, 200, 10)
{'g': 0, 'd': 1, 'b': 20, 'a': 30, 'e': 64, 'c': 100}
>>> ssdb.zscan('zset_1', '', 0, 200, 3)
{'g': 0, 'd': 1, 'b': 20}
>>> ssdb.zscan('zset_1', 'b', 20, 200, 3)
{'a': 30, 'e': 64, 'c': 100}
>>> ssdb.zscan('zset_1', 'c', 100, 200, 3)
{}
```

zrscan

StrictSSDB.**zrscan**(*name*, *key_start*, *score_start*, *score_end*, *limit=10*)

Return a dict mapping key/score of the top `limit` keys after `key_start` with scores between `score_start` and `score_end` in zset `name` in descending order

---

Note: The range is (`key_start``+``score_start`, `key_end`]. That means (key.score == score_start && key < key_start || key.score < score_start)

Parameters

- **name** (*string*) – the zset name
- **key_start** (*string*) – The key related to score_start, could be empty string `''`
- **score_start** (*int*) – The maximum score related to keys(may not be included, depend on key_start), empty string `''` means +inf
- **score_end** (*int*) – The minimum score(included) related to keys, empty string `''` means -inf
- **limit** (*int*) – number of elements will be returned.

Returns a dict mapping key/score in descending order

Return type OrderedDict

```
>>> ssdb.zrscan('zset_1', '', '', '', 10)
{'c': 100, 'e': 64, 'a': 30, 'b': 20, 'd': 1, 'g': 0, 'f': -3}
>>> ssdb.zrscan('zset_1', '', 1000, -1000, 3)
{'c': 100, 'e': 64, 'a': 30}
>>> ssdb.zrscan('zset_1', 'a', 30, -1000, 3)
{'b': 20, 'd': 1, 'g': 0}
>>> ssdb.zrscan('zset_1', 'g', 0, -1000, 3)
{'g': 0}
```

zrank

StrictSSDB.**zrank**(*name*, *key*)

Returns a 0-based value indicating the rank of `key` in zset `name`

Like **Redis.ZRANK**

> **Warning:** This method may be extremly SLOW! May not be used in an online service.

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name

Returns the rank of `key` in zset `name`, **-1** if the `key` or the `name` doesn't exists

Return type int

```
>>> ssdb.zrank('zset_1','d')
2
>>> ssdb.zrank('zset_1','f')
0
>>> ssdb.zrank('zset_1','x')
-1
```

zrrank

`StrictSSDB.`**`zrrank`**(*name*, *key*)

Returns a 0-based value indicating the descending rank of `key` in zset

Warning: This method may be extremly SLOW! May not be used in an online service.

Parameters

- **name** (*string*) – the zset name
- **key** (*string*) – the key name

Returns the reverse rank of `key` in zset `name`, **-1** if the `key` or the `name` doesn't exists

Return type int

```
>>> ssdb.zrrank('zset_1','d')
4
>>> ssdb.zrrank('zset_1','f')
6
>>> ssdb.zrrank('zset_1','x')
-1
```

zrange

`StrictSSDB.`**`zrange`**(*name*, *offset*, *limit*)

Return a dict mapping key/score in a range of score from zset `name` between `offset` and `offset+limit` sorted in ascending order.

Like **Redis.ZRANGE**

Warning: This method is SLOW for large offset!

Parameters

- **name** (*string*) – the zset name
- **offset** (*int*) – zero or positive,the returned pairs will start at this offset
- **limit** (*int*) – number of elements will be returned

Returns a dict mapping key/score in ascending order

Return type OrderedDict

```
>>> ssdb.zrange('zset_1', 2, 3)
{'d': 1, 'b': 20, 'a': 30}
>>> ssdb.zrange('zset_1', 0, 2)
{'f': -3, 'g': 0}
>>> ssdb.zrange('zset_1', 10, 10)
{}
```

zrrange

StrictSSDB.**zrrange**(*name*, *offset*, *limit*)

Return a dict mapping key/score in a range of score from zset `name` between `offset` and `offset+limit` sorted in descending order.

> **Warning:** This method is SLOW for large offset!

Parameters

- **name** (*string*) – the zset name
- **offset** (*int*) – zero or positive,the returned pairs will start at this offset
- **limit** (*int*) – number of elements will be returned

Returns a dict mapping key/score in ascending order

Return type OrderedDict

```
>>> ssdb.zrrange('zset_1', 0, 4)
{'c': 100, 'e': 64, 'a': 30, 'b': 20}
>>> ssdb.zrrange('zset_1', 4, 5)
{'d': 1, 'g': 0, 'f': -3}
>>> ssdb.zrrange('zset_1', 10, 10)
{}
```

zcount

StrictSSDB.**zcount**(*name*, *score_start*, *score_end*)

Returns the number of elements in the sorted set at key `name` with a score between `score_start` and `score_end`.

Like **Redis.ZCOUNT**

Note: The range is [`score_start`, `score_end`]

Parameters

- **name** (*string*) – the zset name
- **score_start** (*int*) – The minimum score related to keys(included), empty string `''` means -inf
- **score_end** (*int*) – The maximum score(included) related to keys, empty string `''` means +inf

Returns the number of keys in specified range

Return type int

```
>>> ssdb.zount('zset_1', 20, 70)
3
>>> ssdb.zcount('zset_1', 0, 100)
6
>>> ssdb.zcount('zset_1', 2, 3)
0
```

zsum

StrictSSDB.**zsum**(*name*, *score_start*, *score_end*)

Returns the sum of elements of the sorted set stored at the specified key which have scores in the range [score_start,score_end].

Note: The range is [score_start, score_end]

Parameters

- **name** (*string*) – the zset name
- **score_start** (*int*) – The minimum score related to keys(included), empty string '' means -inf
- **score_end** (*int*) – The maximum score(included) related to keys, empty string '' means +inf

Returns the sum of keys in specified range

Return type int

```
>>> ssdb.zsum('zset_1', 20, 70)
114
>>> ssdb.zsum('zset_1', 0, 100)
215
>>> ssdb.zsum('zset_1', 2, 3)
0
```

zavg

StrictSSDB.**zavg**(*name*, *score_start*, *score_end*)

Returns the average of elements of the sorted set stored at the specified key which have scores in the range [score_start,score_end].

Note: The range is [score_start, score_end]

Parameters

- **name** (*string*) – the zset name
- **score_start** (*int*) – The minimum score related to keys(included), empty string '' means -inf
- **score_end** (*int*) – The maximum score(included) related to keys, empty string '' means +inf

Returns the average of keys in specified range

Return type int

```
>>> ssdb.zavg('zset_1', 20, 70)
38
>>> ssdb.zavg('zset_1', 0, 100)
35
>>> ssdb.zavg('zset_1', 2, 3)
0
```

zremrangebyrank

StrictSSDB.**zremrangebyrank**(*name*, *rank_start*, *rank_end*)

Remove the elements of the zset which have rank in the range [rank_start,rank_end].

Note: The range is [rank_start, rank_end]

Parameters

- **name** (*string*) – the zset name
- **rank_start** (*int*) – zero or positive,the start position
- **rank_end** (*int*) – zero or positive,the end position

Returns the number of deleted elements

Return type int

```
>>> ssdb.zremrangebyrank('zset_1', 0, 2)
3
>>> ssdb.zremrangebyrank('zset_1', 1, 4)
5
>>> ssdb.zremrangebyrank('zset_1', 0, 0)
1
```

zremrangebyscore

StrictSSDB.**zremrangebyscore**(*name*, *score_start*, *score_end*)

Delete the elements of the zset which have rank in the range [score_start,score_end].

Note: The range is [score_start, score_end]

Parameters

- **name** (*string*) – the zset name
- **score_start** (*int*) – The minimum score related to keys(included), empty string '' means -inf
- **score_end** (*int*) – The maximum score(included) related to keys, empty string '' means +inf

Returns the number of deleted elements

Return type int

```
>>> ssdb.zremrangebyscore('zset_1', 20, 70)
3
>>> ssdb.zremrangebyscore('zset_1', 0, 100)
6
>>> ssdb.zremrangebyscore('zset_1', 2, 3)
0
```

4.2.4 Queue

A queue in ssdb.

```
>>> from ssdb.client import SSDB
>>> ssdb = SSDB()
>>> ssdb.qpush('queue_1', 'a', 'b', 'c', 'd', 'e', 'f', 'g')
>>> ssdb.qpush('queue_2',
...            'test1',
...            'test2',
...            'test3',
...            'test4',
...            'test5',
...            'test6',
...            )
```

qsize

StrictSSDB.**qsize**(*name*)

Return the length of the list `name` . If name does not exist, it is interpreted as an empty list and 0 is returned.

Like **Redis.LLEN**

Parameters **name** (*string*) – the queue name

Returns the queue length or 0 if the queue doesn't exist.

Return type int

```
>>> ssdb.qsize('queue_1')
7
>>> ssdb.qsize('queue_2')
6
>>> ssdb.qsize('queue_not_exists')
0
```

qlist

StrictSSDB.**qlist**(*name_start*, *name_end*, *limit*)

Return a list of the top `limit` keys between `name_start` and `name_end` in ascending order

Note: The range is (`name_start`, `name_end`]. `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means -inf

- **name_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means +inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of keys

Return type list

```
>>> ssdb.qlist('queue_1', 'queue_2', 10)
['queue_2']
>>> ssdb.qlist('queue_', 'queue_2', 10)
['queue_1', 'queue_2']
>>> ssdb.qlist('z', '', 10)
[]
```

qrlist

`StrictSSDB.`**`qrlist`**(*name_start*, *name_end*, *limit*)

Return a list of the top `limit` keys between `name_start` and `name_end` in descending order

Note: The range is (`name_start`, `name_end`]. `name_start` isn't in the range, but `name_end` is.

Parameters

- **name_start** (*string*) – The lower bound(not included) of keys to be returned, empty string `''` means +inf
- **name_end** (*string*) – The upper bound(included) of keys to be returned, empty string `''` means -inf
- **limit** (*int*) – number of elements will be returned.

Returns a list of keys

Return type list

```
>>> ssdb.qrlist('queue_2', 'queue_1', 10)
['queue_1']
>>> ssdb.qrlist('queue_z', 'queue_', 10)
['queue_2', 'queue_1']
>>> ssdb.qrlist('z', '', 10)
['queue_2', 'queue_1']
```

qclear

`StrictSSDB.`**`qclear`**(*name*)

Clear&Delete the queue specified by `name`

Parameters **name** (*string*) – the queue name

Returns the length of removed elements

Return type int

qfront

StrictSSDB.**qfront**(*name*)
Returns the first element of a queue.

Parameters **name** (*string*) – the queue name

Returns None if queue empty, otherwise the item returned

Return type string

qback

StrictSSDB.**qback**(*name*)
Returns the last element of a queue.

Parameters **name** (*string*) – the queue name

Returns None if queue empty, otherwise the item returned

Return type string

qget

StrictSSDB.**qget**(*name*, *index*)
Get the element of index within the queue name

Parameters

- **name** (*string*) – the queue name
- **index** (*int*) – the specified index, can < 0

Returns the value at index within queue name , or None if the element doesn't exist

Return type string

qrange

StrictSSDB.**qrange**(*name*, *offset*, *limit*)
Return a limit slice of the list name at position offset

offset can be negative numbers just like Python slicing notation

Similiar with **Redis.LRANGE**

Parameters

- **name** (*string*) – the queue name
- **offset** (*int*) – the returned list will start at this offset
- **limit** (*int*) – number of elements will be returned

Returns a list of elements

Return type list

qslice

StrictSSDB.**qslice**(*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

Like **Redis.LRANGE**

Parameters

- **name** (*string*) – the queue name
- **start** (*int*) – the returned list will start at this offset
- **end** (*int*) – the returned list will end at this offset

Returns a list of elements

Return type list

qpush_front

StrictSSDB.**qpush_front**(*name*, **items*)
Push `items` onto the head of the list `name`

Like **Redis.LPUSH**

Parameters

- **name** (*string*) – the queue name
- **index** (*int*) – the specified index
- **value** (*string*) – the element value

Returns length of queue

Return type int

qpush_back

StrictSSDB.**qpush_back**(*name*, **items*)
Push `items` onto the tail of the list `name`

Like **Redis.RPUSH**

Parameters

- **name** (*string*) – the queue name
- **items** (*list*) – the list of items

Returns length of queue

Return type int

qpush

StrictSSDB.**qpush**(*name*, **items*)
Push `items` onto the tail of the list `name`

Like **Redis.RPUSH**

Parameters

- **name** (*string*) – the queue name
- **items** (*list*) – the list of items

Returns length of queue

Return type int

qpop_front

`StrictSSDB.`**`qpop_front`**(*name*, *size=1*)

Remove and return the first `size` item of the list `name`

Like **Redis.LPOP**

Parameters

- **name** (*string*) – the queue name
- **size** (*int*) – the length of result

Returns the list of pop elements

Return type list

qpop

`StrictSSDB.`**`qpop`**(*name*, *size=1*)

Remove and return the first `size` item of the list `name`

Like **Redis.LPOP**

Parameters

- **name** (*string*) – the queue name
- **size** (*int*) – the length of result

Returns the list of pop elements

Return type list

qpop_back

`StrictSSDB.`**`qpop_back`**(*name*, *size=1*)

Remove and return the last `size` item of the list `name`

Like **Redis.RPOP**

Parameters

- **name** (*string*) – the queue name
- **size** (*int*) – the length of result

Returns the list of pop elements

Return type list

queue_exists

StrictSSDB.**queue_exists**(*name*)

Return a boolean indicating whether queue `name` exists

Parameters **name** (*string*) – the queue name

Returns `True` if the queue exists, `False` if not

Return type string

```
>>> ssdb.queue_exists('queue_1')
True
>>> ssdb.queue_exists('queue_2')
True
>>> ssdb.queue_exists('queue_not_exist')
False
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S

T

Z