py-elevator Documentation

Release 0.4

oleiade

October 23, 2012

CONTENTS

py-elevator is a python client for Elevator, in Python and based on levelDB allowing high performance on-disk bulk read/write. Which provides async, multithreaded and/or remote access to a multi-leveldb backend. It Relies on the zeromq network library and msgpack serialization format as a messaging protocol. It was made with portability, stability, and focus on performance in mind.

CHAPTER

ONE

REQUIREMENTS

```
pyzmq (built against zmq-3.X)
msgpack-python
```

CHAPTER

TWO

INSTALLATION

To install the last stable version (master)

```
$ git clone git@github.com/oleiade/py-elevator
$ cd py-elevator
$ python setup.py install
```

To install the last tag with pip

```
pip install -e git+git@github.com/oleiade/py-elevator@{{tag-name}}.git#egg=py-elevator
```

CHAPTER

THREE

USAGE

Nota : See Elevator documentation for details about server usage and implementation

3.1 Databases workaround

```
>>> from pyelevator import Elevator

# Elevator server holds a default db
# which the client will automatically
# connect to
>>> E = Elevator()
>>> E.db_name
'default'

# You can list remote databases
>>> E.listdb()
['default', ]

# Create a db
>>> E.createdb('testdb')
>>> E.listdb()
['default', 'testdb', ]

# And bind your client to that new Db.
>>> E.connect('testdb')

# Note that you canno't connect to a db that doesn't exist yet
>>> E.connect('dbthatdoesntexist')
DatabaseError : "Database does not exist"

# Sometimes, leveldb just messes up with the backend
# When you're done with a db, you can drop it. Note that all it's files
# will be droped too.
>>> E.repairdb()
>>> E.dropdb('testdb')


# You can even register a pre-existing leveldb db
# as an Elevator db. By creating it using it's path.
>>> E.createdb('/path/to/my/existing/leveldb')
>>> E.listdb()
['default', '/path/to/my/existing/leveldb', ]
```

3.2 Interact with a database

```
>>> from pyelevator import Elevator
>>> E = Elevator()                    # N.B : connected to 'default'

>>> E.Put('abc', '123')
>>> E.Put('easy as', 'do re mi')
>>> E.Get('abc')
'123'
>>> E.MGet(['abc', 'easy as', 'you and me'])
['123', 'do re mi', None]
>>> E.Delete('abc')
>>> for i in xrange(10):
...     E.Put(str(i), str(i))

# Range supports key_from, key_to params
>>> E.Range('1', '9')
[['1','1'],
 ['2','2'],
 ['3', '3'],
 ['4', '4'],
 ['5', '5'],
 ['6', '6'],
 ['7', '7'],
 ['8', '8'],
 ['9', '9'],
]


# Or key_from, limit params
>>> E.Slice('1', 2)
[['1', '1'],
 ['2', '2'],
]

# When RangeIter only knows about key_from/key_to for py-leveldb api
# compatibility reasons
>>> it = E.RangeIter('1', '2')
>>> list(it)
[['1', '1'],
 ['2', '2'],
]

# Elevator objects supports with_statement too
>>> with Elevator('testdb') as e:
>>> ....e.Get('1')
>>>
'1'
```

CHAPTER

FOUR

BATCHES

They're very handy and very fast when it comes to write a lot of datas to the database. See LevelDB documentation for more informations. Use it through the WriteBatch client module class. It has three base methods modeled on LevelDB's Put, Delete, Write.

```
>>> from pyelevator import WriteBatch, Elevator

# Just like Elevator object, WriteBatch connects to 'default' as a default
# But as it supports the exact same options that Elevator, you can
# Init it with a pre-existing db
>>> batch = WriteBatch()
>>> batch = WriteBatch('testdb')

>>> batch.Put('a', 'a')
>>> batch.Put('b', 'b')
>>> batch.Put('c', 'c')
>>> batch.Delete('c')
>>> batch.Write()

>>> E = Elevator()
>>> E.Get('a')
'a'
>>> E.Get('b')
'b'
>>> E.Get('c')
KeyError: "Key not found"

# Batches objects supports with_statement too
# Write will be automatically called on __exit__
>>> with WriteBatch('testdb') as batch:
>>> ....batch.Put('abc', '123')
>>> ....batch.Put('or simple as...', 'do re mi')
```

CHAPTER

FIVE

API

5.1 Elevator object

5.1.1 Database store management

- `connect` : *db_name*
- `listdb`
- `createdb` : *db_name*
- `dropdb` : *db_name*
- `repairdb`
- `Read/Write`

Nota : Every functions are handling a kwarg timeout param which defines in seconds how long the client should wait for a server response. You might wanna set this to a high value when processing large datas sets (Range/Rangeiter/MGet).

- `Get` : *key*, *value*
- `Put` : *key*, *value*
- `Delete` : *key*
- **`Range`** [*start*, *limit*] limit can whether be a string, and will be considered a stop key then, or an int, and will be considered as an offset.
- `RangeIter` : *key_from*, *key_to*
- **`MGet`** [*keys*] Keys should whether be a list or a tuple of strings Accepts a specific fill_cache kwarg, which is by default set to False. Defines if the leveldb backend cache should be updated with fetched values or not. When proceeding to "small" and/or repetitive random read, you might want to set this option to True; but for bulk reads on medium and large sets, keep it set to False.

5.2 WriteBatch object

Nota : idem than Read/Write

- `Put` : *key*, *value*
- `Delete` : *key*
- `Write`