aiogibson Documentation

Release 0.1.0

Nikolay Novik

September 16, 2014

Contents

aiogibson is a library for accessing a gibson cache database from the asyncio (PEP-3156/tulip) framework.

Gibson is a high efficiency, tree based memory cache server. It uses a special trie structure allowing the user to perform operations on multiple key sets using a prefix expression achieving the same performance grades in the worst case, even better on an average case then regular cache implementations based on hash tables.

Contents:

CHAPTER 1

Installation

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

```
pip install aiogibson
```

CHAPTER 2

Contribute

- Issue Tracker: https://github.com/jettify/aiogibson/issues
- Source Code: https://github.com/jettify/aiogibson

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

CHAPTER 3

Contents

3.1 Getting started

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

```
pip install aiogibson
```

Make sure that gibson server installed and started according official documentation. We assume that you have your *gibson* started using unix sockets (by default) with address `/tmp/aiogibson.sock`, your `python` version >= 3.3.

aiogibson has straightforward api, just like *memcached*:

3.1.1 Basic Example

```
import asyncio
from aiogibson import create_gibson

loop = asyncio.get_event_loop()


@asyncio.coroutine
def go():
    gibson = yield from create_gibson('/tmp/aio.sock', loop=loop)
    # set value
    yield from gibson.set(b'foo', b'bar', 7)
    yield from gibson.set(b'numfoo', 100, 7)

    # get value
    result = yield from gibson.get(b'foo')
    print(result)

    # set ttl to the value
    yield from gibson.ttl(b'foo', 10)

    # increment given key
    yield from gibson.inc(b'numfoo')

    # decrement given key
    yield from gibson.dec(b'numfoo')
```

```
    # lock key from modification
    yield from gibson.lock(b'numfoo')

    # unlock given key
    yield from gibson.unlock(b'numfoo')

    # delete value
    yield from gibson.delete(b'foo')

    # Get system stats about the Gibson instance
    info = yield from gibson.stats()


loop.run_until_complete(go())
```

Underlying data structure trie allows us to perform operations on multiple key sets using a prefix expression:

3.1.2 Multi Commands

```
import asyncio
from aiogibson import create_gibson

loop = asyncio.get_event_loop()


@asyncio.coroutine
def go():
    gibson = yield from create_gibson('/tmp/aio.sock', loop=loop)

    # set the value for keys verifying the given prefix
    yield from gibson.mset(b'fo', b'bar', 7)
    yield from gibson.mset(b'numfo', 100, 7)

    # get the values for keys with given prefix
    result = yield from gibson.mget(b'fo')

    # set the TTL for keys verifying the given prefix
    yield from gibson.mttl(b'fo', 10)

    # increment by one keys verifying the given prefix.
    yield from gibson.minc(b'numfo')

    # decrement by one keys verifying the given prefix
    yield from gibson.mdec(b'numfoo')

    # lock keys with prefix from modification
    yield from gibson.mlock(b'fo')

    # unlock keys with given prefix
    yield from gibson.munlock(b'fo')

    # delete keys verifying the given prefix.
    yield from gibson.mdelete(b'fo')

    # return list of keys with given prefix ''fo''
    yield from gibson.keys(b'fo')
```

```
    # count items for a given prefi
    info = yield from gibson.stats()


loop.run_until_complete(go())
```

aiogibson has connection pooling support using context-manager:

3.1.3 Connection Pool Example

```
import asyncio
from aiogibson import create_pool

loop = asyncio.get_event_loop()

@asyncio.coroutine
def go():
    pool = yield from create_pool('/tmp/aio.sock', minsize=5, maxsize=10,
                                  loop=loop)

    with (yield from pool) as gibson:
        yield from gibson.set('foo', 'bar')
        value = yield from gibson.get('foo')
        print(value)

    pool.clear()

loop.run_until_complete(go())
```

Also you can have simple low-level interface to *gibson* server:

3.1.4 Low Level Commands

```
import asyncio
from aiogibson import create_gibson

loop = asyncio.get_event_loop()


@asyncio.coroutine
def go():
    gibson = yield from create_connection('/tmp/aio.sock', loop=loop)
    # set value
    yield from gibson.execute(b'set', b'foo', b'bar', 7)
    # get value
    result = yield from gibson.execute(b'get', b'foo')
    print(result)
    # delete value
    yield from gibson.execute(b'del', b'foo')

loop.run_until_complete(go())
```

3.2 aiogibson — API Reference

3.3 Connection Object

Low level connection with raw interface.

aiogibson.connection.**create_connection**(*address*, *, *encoding=None*, *loop=None*)
Creates GibsonConnection connection. Opens connection to Gibson server specified by address argument.

Parameters

- **address** – str for unix socket path, or tuple for (host, port) tcp connection.
- **encoding** – this argument can be used to decode byte-replies to strings. By default no decoding is done.

class aiogibson.connection.**GibsonConnection**(*reader*, *writer*, *address*, *, *encoding=None*, *loop=None*)
Gibson connection.

close()
Close connection.

closed
True if connection is closed.

encoding
Current set codec or None.

execute(*command*, **args*, *encoding=<object object at 0x7f0da1e2a140>*)
Executes raw gibson command.

Parameters

- **command** – str or bytes gibson command.
- **args** – tuple of arguments required for gibson command.
- **encoding** – str default encoding for unpacked data.

Raises

- **TypeError** – if any of args can not be encoded as bytes.
- **ProtocolError** – when response can not be decoded meaning connection is broken.

3.4 Connection Pool

Pool of connection using context manager protocol:

```
import asyncio
from aiogibson import create_pool

loop = asyncio.get_event_loop()

@asyncio.coroutine
def go():
    pool = yield from create_pool('/tmp/aio.sock', minsize=5, maxsize=10,
                                  loop=loop)
```

```
    with (yield from pool) as gibson:
        yield from gibson.set('foo', 'bar')
        value = yield from gibson.get('foo')
        print(value)

    pool.clear()

loop.run_until_complete(go())
```

aiogibson.pool.**create_pool**(*address*, ***, *encoding=None*, *minsize=10*, *maxsize=10*, *commands_factory=<class 'aiogibson.commands.Gibson'>*, *loop=None*)

Creates Gibson Pool.

By default it creates pool of commands_factory instances, but it is also possible to create pool of plain connections by passing `lambda conn: conn` as commands_factory. All arguments are the same as for create_connection. Returns GibsonPool instance.

class aiogibson.pool.**GibsonPool**(*address*, *encoding=None*, ***, *minsize*, *maxsize*, *commands_factory*, *loop=None*)

Gibson connections pool.

acquire()

Acquires a connection from free pool.

Creates new connection if needed.

clear()

Clear pool connections.

Close and remove all free connections.

encoding

Current set codec or None.

freesize

Current number of free connections.

maxsize

Maximum pool size.

minsize

Minimum pool size.

release(*conn*)

Returns used connection back into pool.

size

Current pool size.

3.5 Protocol Parser

The `Reader` class has two methods that are used when parsing replies from a stream of data. Reader.feed takes a string argument that is appended to the internal buffer. `Reader`.gets reads this buffer and returns a reply when the buffer contains a full reply. If a single call to feed contains multiple replies, gets should be called multiple times to extract all replies.

```
>>> reader = aiogibson.Reader()
>>> reader.feed(b'\x06\x00\x05\x03\x00\x00\x00bar')
>>> reader.gets()
b'bar'
```

When the buffer does not contain a full reply, gets returns False. This means extra data is needed and feed should be called again before calling gets again:

```
>>> reader.feed(b'\')
>>> reader.gets()
False
>>> reader.feed(b'\x03\x00\x00\x00bar')
>>> reader.gets()
b'bar'
```

note api same as in *hiredis*.

This module has `encode_command`, packs *gibson* command to binary format suitable to send over socket to *gibson* server:

```
>>> encode_command(b'set', 3600, 'foo', 3.14)
b'\x0f\x00\x00\x00\x01\x003600 foo 3.14'
```

`aiogibson.parser.`**`encode_command`**(*command*, **args*)

Pack and encode *gibson* command according to gibson binary protocol

See http://gibson-db.in/protocol/

Parameters

- **command** – `bytes`, gibson command (get, set, etc.)
- **args** – required arguments for given command.

Returns `bytes` packed and encoded command.

class `aiogibson.parser.`**`Reader`**

This class is responsible for parsing replies from the stream of data that is read from a *Gibson* connection. It does not contain functionality to handle I/O

`feed`(*data*)

Put raw chunk of data obtained from connection to buffer.

Parameters **data** – `bytes`, raw input data.

`gets`()

When the buffer does not contain a full reply, gets returns False. This means extra data is needed and feed should be called again before calling gets again:

Returns `False` there is no full reply or parsed obj.

3.6 Exceptions

exception `aiogibson.errors.`**`GibsonError`**

Base exception class for aiogibson exceptions.

exception `aiogibson.errors.`**`ProtocolError`**

Raised when protocol error occurs.

exception `aiogibson.errors.`**`ReplyError`**

Generic error while executing the query

exception `aiogibson.errors.`**`ExpectedANumber`**

Expected a number (TTL or TIME) but the specified value was invalid.

exception aiogibson.errors.**MemoryLimitError**
The server reached configuration memory limit and will not accept any new value until its freeing routine will be executed.

exception aiogibson.errors.**KeyLockedError**
The specified key was locked by a *OP_LOCK* or a *OP_MLOCK* query.

3.7 High Level Commands

aiogibson.commands.**create_gibson**(*address*, ***, *encoding=None*, *commands_factory=<class 'aiogibson.commands.Gibson'>*, *loop=None*)
Create high-level Gibson interface.

Parameters

- **address** – str for unix socket path, or tuple for (host, port) tcp connection.
- **encoding** – this argument can be used to decode byte-replies to strings. By default no decoding is done.
- **commands_factory** –
- **loop** – event loop to use

Returns high-level Gibson connection Gibson

class aiogibson.commands.**Gibson**(*connection*)
High-level Gibson interface

See http://gibson-db.in/commands/

closed
True if connection is closed.

count(*prefix*)
Count items for a given prefix.

Parameters **prefix** – bytes The key prefix to use as expression

Returns int number of elements

dec(*key*)
Decrement by one the given key.

Parameters **key** – bytes, key to decrement.

Returns int decremented value in case of success

delete(*key*)
Delete the given key.

Parameters **key** – bytes key to delete.

Returns bool true in case of success.

end()
Disconnects from the client from gibson instance.

get(*key*)
Get the value for a given key.

Parameters **key** – bytes key to get.

Returns bytes if value exists else None

inc(*key*)
Increment by one the given key.

Parameters **key** – `bytes`, key to increment.

Returns `int` incremented value

keys(*prefix*)
Return a list of keys matching the given prefix.

Parameters **prefix** – key prefix to use as expression.

Returns `list` of available keys

lock(*key*, *expire=0*)
Prevent the given key from being modified for a given amount of seconds.

Parameters

- **key** – `bytes`, key to decrement.
- **expire** – `int`, time in seconds to lock the item.

Returns `bool`

Raises **TypeError** if expire argument is not `int`

mdec(*prefix*)
Decrement by one keys verifying the given prefix.

Parameters **prefix** – prefix for keys.

Returns `int`, number of modified items, otherwise an error.

mdelete(*prefix*)
Delete keys verifying the given prefix.

Parameters **prefix** – prefix for keys.

Returns `int`, number of modified items, otherwise an error.

meta_access(*key*)
Timestamp of the last time the item was accessed.

Parameters **key** – `bytes`, key of interest.

Returns `int`, timestamp

meta_created(*key*)
Timestamp of item creation.

Parameters **key** – `bytes`, key of interest.

Returns `int`, timestamp

meta_encoding(*key*)
Gibson encoding for given value.

Parameters **key** – `bytes`, key of interest.

Returns `int`, gibson encoding, 0 - `bytes`, 2 - `int`.

meta_left(*key*)
Number of seconds left for the item to live if a ttl was specified, otherwise -1.

Parameters **key** – `bytes`, key of interest.

Returns `int`, Number of seconds left.

meta_lock(*key*)
Number of seconds the item is locked, -1 if there's no lock.

Parameters **key** – `bytes`, key of interest.

Returns `int`, number of seconds

meta_size(*key*)
The size in bytes of the item value.

Parameters **key** – `bytes`, key of interest.

Returns `int`, value size in bytes

meta_ttl(*key*)
Item specified time to live, -1 for infinite TTL.

Parameters **key** – `bytes`, key of interest.

Returns `int`, seconds of TTL.

mget(*prefix*)
Get the values for keys with given prefix.

Parameters **prefix** – prefix for keys.

Returns `list` of key/value pairs

minc(*prefix*)
Increment by one keys verifying the given prefix.

Parameters **prefix** – prefix for keys.

Returns `int`, number of modified items, otherwise an error.

mlock(*prefix*, *expire=0*)
Prevent keys verifying the given prefix from being modified for a given amount of seconds.

Parameters **prefix** – `bytes`, prefix for keys.

:param expire:`int`, lock period in seconds. :return: `int`, number of modified items, otherwise an error. :raises TypeError: if expire argument is not `int`

mset(*prefix*, *value*)
Set the value for keys verifying the given prefix.

Parameters **prefix** – prefix for keys.

Returns `int`, number of modified items, otherwise an error.

mttl(*prefix*, *expire=0*)
Set the TTL for keys verifying the given prefix.

Parameters

- **prefix** – prefix for keys.
- **expire** – `int`, new expiration time.

Returns `int`, number of modified items, otherwise an error.

Raises **TypeError** if expire argument is not `int`

munlock(*prefix*)
Remove the lock on keys verifying the given prefix.

Parameters **prefix** – prefix for keys.

Returns `int`, number of affected items, otherwise an error.

ping()
Ping the server instance to refresh client last seen timestamp.

Returns `True` or error.

set(*key*, *value*, *expire=0*)
Set the value for the given key, with an optional TTL.

Parameters

- **key** – `bytes` key to set.
- **value** – `bytes` value to set.
- **expire** – `int` optional ttl in seconds

Raises **TypeError** if expire argument is not `int`

stats()
Get system stats about the Gibson instance.

Returns `list` of pairs (stat, value).

ttl(*key*, *expire*)
Set the TTL of a key.

Parameters

- **key** – `bytes`, key to set ttl.
- **expire** – `int`, TTL in seconds.

Returns `bool`, True in case of success.

Raises **TypeError** if expire argument is not `int`

unlock(*key*)
Remove the lock from the given key.

Parameters **key** – `bytes` key ot unlock.

Returns `bool`, True in case of success.

CHAPTER 4

Indices and tables

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

a