aiomysql Documentation

Release 0.0.20

Nikolay Novik

May 22, 2019

Contents

aiomysql is a library for accessing a *MySQL* database from the asyncio (PEP-3156/tulip) framework. It depends and reuses most parts of *PyMySQL* . **aiomysql** tries to be like awesome aiopg library and preserve same api, look and feel.

Internally **aiomysql** is copy of PyMySQL, underlying io calls switched to async, basically `yield from` and `asyncio.coroutine` added in proper places. *sqlalchemy* support ported from aiopg.

CHAPTER 1

Features

- Implements *asyncio* *DBAPI like* interface for *MySQL*. It includes *Connection*, *Cursor* and *Pool* objects.
- Implements *optional* support for charming *sqlalchemy* functional sql layer.

CHAPTER 2

Basics

aiomysql based on *PyMySQL* , and provides same api, you just need to use `yield from conn.f()` instead of just call `conn.f()` for every method.

Properties are unchanged, so `conn.prop` is correct as well as `conn.prop = val`.

See example:

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='', db='mysql',
                                       loop=loop)

    cur = yield from conn.cursor()
    yield from cur.execute("SELECT Host,User FROM user")
    print(cur.description)
    r = yield from cur.fetchall()
    print(r)
    yield from cur.close()
    conn.close()

loop.run_until_complete(test_example())
```

CHAPTER 3

Installation

```
pip3 install aiomysql
```

Note: *aiomysql* requires *PyMySQL* library.

Also you probably want to use *aiomysql.sa*.

aiomysql.sa module is **optional** and requires *sqlalchemy*. You can install *sqlalchemy* by running:

```
pip3 install sqlalchemy
```

CHAPTER 4

Source code

The project is hosted on GitHub

Please feel free to file an issue on bug tracker if you have found a bug or have some suggestion for library improvement.

The library uses Travis for Continious Integration and Coveralls for coverage reports.

CHAPTER 5

Dependencies

- Python 3.5.3+
- *PyMySQL*
- aiomysql.sa requires *sqlalchemy*.

CHAPTER 6

Authors and License

The `aiomysql` package is written by Nikolay Novik, *PyMySQL* and aio-libs contributors. It's MIT licensed (same as PyMySQL).

Feel free to improve this package and send a pull request to GitHub.

CHAPTER 7

Contents:

7.1 aiomysql — API Reference

7.2 Connection

The library provides a way to connect to MySQL database with simple factory function `aiomysql.connnect()`. Use this function if you want just one connection to the database, consider connection pool for multiple connections.

Example:

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='', db='mysql',
                                       loop=loop)

    cur = yield from conn.cursor()
    yield from cur.execute("SELECT Host,User FROM user")
    print(cur.description)
    r = yield from cur.fetchall()
    print(r)
    yield from cur.close()
    conn.close()

loop.run_until_complete(test_example())
```

connect(host="localhost", user=None, password="",
db=None, port=3306, unix_socket=None,
charset='', sql_mode=None,

read_default_file=None, conv=decoders, use_unicode=None,
client_flag=0, cursorclass=Cursor, init_command=None,
connect_timeout=None, read_default_group=None,
no_delay=False, autocommit=False, echo=False,
ssl=None, auth_plugin='', program_name='',
server_public_key=None, loop=None)

A coroutine that connects to MySQL.

The function accepts all parameters that `pymysql.connect()` does plus optional keyword-only *loop* and *timeout* parameters.

param str host host where the database server is located, default: *localhost*.

param str user username to log in as.

param str password password to use.

param str db database to use, None to not use a particular one.

param int port MySQL port to use, default is usually OK.

param str unix_socket optionally, you can use a unix socket rather than TCP/IP.

param str charset charset you want to use, for example 'utf8'.

param sql_mode default sql-mode to use, like 'NO_BACKSLASH_ESCAPES'

param read_default_file specifies my.cnf file to read these parameters from under the [client] section.

param conv decoders dictionary to use instead of the default one. This is used to provide custom marshalling of types. See *pymysql.converters*.

param use_unicode whether or not to default to unicode strings.

param client_flag custom flags to send to MySQL. Find potential values in *pymysql.constants.CLIENT*.

param cursorclass custom cursor class to use.

param str init_command initial SQL statement to run when connection is established.

param connect_timeout Timeout in seconds before throwing an exception when connecting.

param str read_default_group Group to read from in the configuration file.

param bool no_delay disable Nagle's algorithm on the socket

param autocommit Autocommit mode. None means use server default. (default: `False`)

param ssl Optional SSL Context to force SSL

param auth_plugin String to manually specify the authentication plugin to use, i.e you will want to use mysql_clear_password when using IAM authentication with Amazon RDS. (default: Server Default)

param program_name Program name string to provide when handshaking with MySQL. (default: sys.argv[0])

param server_public_key SHA256 authenticaiton plugin public key value.

param loop asyncio event loop instance or `None` for default one.

returns `Connection` instance.

Representation of a socket with a mysql server. The proper way to get an instance of this class is to call `aiomysql.connnect()`.

Its insterface is almost the same as *pymysql.connection* except all methods are coroutines.

The most important methods are:

`aiomysql.`**`cursor`**`(`*cursor=None*`)`
A coroutine that creates a new cursor object using the connection.

By default, *Cursor* is returned. It is possible to also give a custom cursor through the *cursor* parameter, but it needs to be a subclass of *Cursor*

Parameters **cursor** – subclass of *Cursor* or `None` for default cursor.

Returns *Cursor* instance.

`aiomysql.`**`close`**`()`
Immediately close the connection.

Close the connection now (rather than whenever *del* is executed). The connection will be unusable from this point forward.

`aiomysql.`**`ensure_closed`**`()`
A coroutine ends quit command and then closes socket connection.

`aiomysql.`**`autocommit`**`(`*value*`)`
A coroutine to enable/disable autocommit mode for current MySQL session. :param bool value: toggle atutocommit mode.

`aiomysql.`**`get_autocommit`**`()`
Returns autocommit status for current MySQL sesstion. :returns bool: current autocommit status.

`aiomysql.`**`begin`**`()`
A coroutine to begin transaction.

`aiomysql.`**`commit`**`()`
Commit changes to stable storage coroutine.

`aiomysql.`**`rollback`**`()`
Roll back the current transaction coroutine.

`aiomysql.`**`select_db`**`(`*db*`)`
A coroutine to set current db.

Parameters **db** (*str*) – database name

`aiomysql.`**`closed`**
The readonly property that returns `True` if connections is closed.

`aiomysql.`**`host`**
MySQL server IP address or name.

`aiomysql.`**`port`**
MySQL server TCP/IP port.

`aiomysql.`**`unix_socket`**
ySQL Unix socket file location.

`aiomysql.`**`db`**
Current database name.

`aiomysql.`**`user`**
User used while connecting to MySQL

aiomysql.**echo**
Return echo mode status.

aiomysql.**encoding**
Encoding employed for this connection.

aiomysql.**charset**
Returns the character set for current connection.

7.3 Cursor

class Cursor

A cursor for connection.

Allows Python code to execute *MySQL* command in a database session. Cursors are created by the `Connection.cursor()` coroutine: they are bound to the connection for the entire lifetime and all the commands are executed in the context of the database session wrapped by the connection.

Cursors that are created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors. Cursors created from different connections can or can not be isolated, depending on the connections' isolation level.

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='',
                                       db='mysql', loop=loop)

    # create default cursor
    cursor = yield from conn.cursor()

    # execute sql query
    yield from cursor.execute("SELECT Host, User FROM user")

    # fetch all results
    r = yield from cursor.fetchall()

    # detach cursor from connection
    yield from cursor.close()

    # close connection
    conn.close()

loop.run_until_complete(test_example())
```

Use `Connection.cursor()` for getting cursor for connection.

connection
This read-only attribute return a reference to the `Connection` object on which the cursor was created

echo
Return echo mode status.

description
This read-only attribute is a sequence of 7-item sequences.

Each of these sequences is a collections.namedtuple containing information describing one result column:

0. name: the name of the column returned.
1. type_code: the type of the column.
2. display_size: the actual length of the column in bytes.
3. internal_size: the size in bytes of the column associated to this column on the server.
4. precision: total number of significant digits in columns of type `NUMERIC`. None for other types.
5. scale: count of decimal digits in the fractional part in columns of type `NUMERIC`. None for other types.
6. null_ok: always None.

This attribute will be None for operations that do not return rows or if the cursor has not had an operation invoked via the *Cursor.execute()* method yet.

rowcount
Returns the number of rows that has been produced of affected.

This read-only attribute specifies the number of rows that the last *Cursor.execute()* produced (for Data Query Language statements like SELECT) or affected (for Data Manipulation Language statements like `UPDATE` or `INSERT`).

The attribute is -1 in case no *Cursor.execute()* has been performed on the cursor or the row count of the last operation if it can't be determined by the interface.

rownumber
Row index. This read-only attribute provides the current 0-based index of the cursor in the result set or `None` if the index cannot be determined.

arraysize
How many rows will be returned by *Cursor.fetchmany()* call.

This read/write attribute specifies the number of rows to fetch at a time with *Cursor.fetchmany()*. It defaults to 1 meaning to fetch a single row at a time.

lastrowid
This read-only property returns the value generated for an *AUTO_INCREMENT* column by the previous *INSERT* or *UPDATE* statement or None when there is no such value available. For example, if you perform an *INSERT* into a table that contains an *AUTO_INCREMENT* column, *Cursor.lastrowid* returns the *AUTO_INCREMENT* value for the new row.

closed
The readonly property that returns `True` if connections was detached from current cursor

close()
Coroutine to close the cursor now (rather than whenever `del` is executed). The cursor will be unusable from this point forward; closing a cursor just exhausts all remaining data.

execute(*query*, *args=None*)
Coroutine, executes the given operation substituting any markers with the given parameters.

For example, getting all rows where id is 5:

```
yield from cursor.execute("SELECT * FROM t1 WHERE id=%s", (5,))
```

Parameters

- **query** (*str*) – sql statement
- **args** (*list*) – tuple or list of arguments for sql query

Returns int number of rows that has been produced of affected

executemany (*query*, *args*)

The *executemany()* coroutine will execute the operation iterating over the list of parameters in seq_params.

Example: Inserting 3 new employees and their phone number:

```
data = [
    ('Jane','555-001'),
    ('Joe', '555-001'),
    ('John', '555-003')
   ]
stmt = "INSERT INTO employees (name, phone)
    VALUES ('%s','%s')"
yield from cursor.executemany(stmt, data)
```

INSERT statements are optimized by batching the data, that is using the MySQL multiple rows syntax.

Parameters

- **query** (*str*) – sql statement
- **args** (*list*) – tuple or list of arguments for sql query

callproc (*procname*, *args*)

Execute stored procedure procname with args, this method is coroutine.

Compatibility warning: PEP-249 specifies that any modified parameters must be returned. This is currently impossible as they are only available by storing them in a server variable and then retrieved by a query. Since stored procedures return zero or more result sets, there is no reliable way to get at OUT or INOUT parameters via *callproc*. The server variables are named *@_procname_n*, where *procname* is the parameter above and n is the position of the parameter (from zero). Once all result sets generated by the procedure have been fetched, you can issue a *SELECT @_procname_0, ...* query using *Cursor.execute()* to get any OUT or INOUT values. Basic usage example:

```
conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                   user='root', password='',
                                   db='mysql', loop=self.loop)

cur = yield from conn.cursor()
yield from cur.execute("""CREATE PROCEDURE myinc(p1 INT)
                        BEGIN
                            SELECT p1 + 1;
                        END
                        """)

yield from cur.callproc('myinc', [1])
(ret, ) = yield from cur.fetchone()
assert 2, ret
```

(continues on next page)

(continued from previous page)

```
yield from cur.close()
conn.close()
```

Compatibility warning: The act of calling a stored procedure itself creates an empty result set. This appears after any result sets generated by the procedure. This is non-standard behavior with respect to the DB-API. Be sure to use `Cursor.nextset()` to advance through all result sets; otherwise you may get disconnected.

Parameters

- **procname** (*str*) – name of procedure to execute on server
- **args** – sequence of parameters to use with procedure

Returns the original args.

fetchone()

Fetch the next row coroutine.

fetchmany(*size=None*)

Coroutine the next set of rows of a query result, returning a list of tuples. When no more rows are available, it returns an empty list.

The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's *Cursor.arraysize* determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned

```
cursor = yield from connection.cursor()
yield from cursor.execute("SELECT * FROM test;")
r = cursor.fetchmany(2)
print(r)
# [(1, 100, "abc'def"), (2, None, 'dada')]
r = yield from cursor.fetchmany(2)
print(r)
# [(3, 42, 'bar')]
r = yield from cursor.fetchmany(2)
print(r)
# []
```

Parameters **size** (*int*) – number of rows to return

Returns list of fetched rows

fetchall()

Coroutine returns all rows of a query result set:

```
yield from cursor.execute("SELECT * FROM test;")
r = yield from cursor.fetchall()
print(r)
# [(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
```

Returns list list of fetched rows

scroll(*value*, *mode='relative'*)

Scroll the cursor in the result set to a new position according to mode. This method is coroutine.

If mode is `relative` (default), value is taken as offset to the current position in the result set, if set to `absolute`, value states an absolute target position. An IndexError should be raised in case a scroll operation would leave the result set. In this case, the cursor position is left undefined (ideal would be to not move the cursor at all).

Note: According to the *DBAPI*, the exception raised for a cursor out of bound should have been `IndexError`. The best option is probably to catch both exceptions in your code:

```
try:
    yield from cur.scroll(1000 * 1000)
except (ProgrammingError, IndexError), exc:
    deal_with_it(exc)
```

Parameters

- **value** (*int*) – move cursor to next position according to mode.
- **mode** (*str*) – scroll mode, possible modes: *relative* and *absolute*

class DictCursor

A cursor which returns results as a dictionary. All methods and arguments same as *Cursor*, see example:

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='',
                                       db='mysql', loop=loop)

    # create dict cursor
    cursor = yield from conn.cursor(aiomysql.DictCursor)

    # execute sql query
    yield from cursor.execute(
        "SELECT * from people where name='bob'")

    # fetch all results
    r = yield from cursor.fetchone()
    print(r)
    # {'age': 20, 'DOB': datetime.datetime(1990, 2, 6, 23, 4, 56),
    # 'name': 'bob'}

loop.run_until_complete(test_example())
```

You can customize your dictionary, see example:

```
import asyncio
import aiomysql

class AttrDict(dict):
    """Dict that can get attribute by dot, and doesn't raise KeyError"""
```

(continues on next page)

(continued from previous page)

```
    def __getattr__(self, name):
        try:
            return self[name]
        except KeyError:
            return None

class AttrDictCursor(aiomysql.DictCursor):
    dict_type = AttrDict

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='',
                                       db='mysql', loop=loop)

    # create your dict cursor
    cursor = yield from conn.cursor(AttrDictCursor)

    # execute sql query
    yield from cursor.execute(
        "SELECT * from people where name='bob'")

    # fetch all results
    r = yield from cursor.fetchone()
    print(r)
    # {'age': 20, 'DOB': datetime.datetime(1990, 2, 6, 23, 4, 56),
    # 'name': 'bob'}
    print(r.age)
    # 20
    print(r.foo)
    # None

loop.run_until_complete(test_example())
```

class SSCursor

Unbuffered Cursor, mainly useful for queries that return a lot of data, or for connections to remote servers over a slow network.

Instead of copying every row of data into a buffer, this will fetch rows as needed. The upside of this, is the client uses much less memory, and rows are returned much faster when traveling over a slow network, or if the result set is very big.

There are limitations, though. The MySQL protocol doesn't support returning the total number of rows, so the only way to tell how many rows there are is to iterate over every row returned. Also, it currently isn't possible to scroll backwards, as only the current row is held in memory. All methods are the same as in *Cursor* but with different behaviour.

fetchall()
Same as :meth:`Cursor.fetchall` :ref:`coroutine <coroutine>`,
useless for large queries, as all rows fetched one by one.

fetchmany(*size=None*, *mode='relative'*)
Same as :meth:`Cursor.fetchall`, but each row fetched one by one.

scroll(*size=None*)
Same as :meth:`Cursor.scroll`, but move cursor on server side one by

one. If you want to move 20 rows forward scroll will make 20 queries to move cursor. Currently only forward scrolling is supported.

class SSDictCursor
An unbuffered cursor, which returns results as a dictionary.

7.4 Pool

The library provides *connection pool* as well as plain `Connection` objects.

The basic usage is:

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def go()
    pool = yield from aiomysql.create_pool(host='127.0.0.1', port=3306,
                                           user='root', password='',
                                           db='mysql', loop=loop, autocommit=False)

    with (yield from pool) as conn:
        cur = yield from conn.cursor()
        yield from cur.execute("SELECT 10")
        # print(cur.description)
        (r,) = yield from cur.fetchone()
       assert r == 10
    pool.close()
    yield from pool.wait_closed()

loop.run_until_complete(go())
```

create_pool(*minsize=1*, *maxsize=10*, *loop=None*, ***kwargs*)
A coroutine that creates a pool of connections to *MySQL* database.

Parameters

- **minsize** (*int*) – minimum sizes of the *pool*.
- **maxsize** (*int*) – maximum sizes of the *pool*.
- **loop** – is an optional *event loop* instance, `asyncio.get_event_loop()` is used if *loop* is not specified.
- **echo** (*bool*) – – executed log SQL queryes (`False` by default).
- **kwargs** – The function accepts all parameters that `aiomysql.connect()` does plus optional keyword-only parameters *loop*, *minsize*, *maxsize*.

Returns *Pool* instance.

class Pool

A connection pool.

After creation pool has *minsize* free connections and can grow up to *maxsize* ones.

If *minsize* is `0` the pool doesn't creates any connection on startup.

If *maxsize* is `0` than size of pool is unlimited (but it recycles used connections of course).

The most important way to use it is getting connection in *with statement*:

```
with (yield from pool) as conn:
    cur = yield from conn.cursor()
```

See also *Pool.acquire()* and *Pool.release()* for acquring `Connection` without *with statement*.

echo
Return *echo mode* status. Log all executed queries to logger named `aiomysql` if `True`

minsize
A minimal size of the pool (*read-only*), `1` by default.

maxsize
A maximal size of the pool (*read-only*), `10` by default.

size
A current size of the pool (*readonly*). Includes used and free connections.

freesize
A count of free connections in the pool (*readonly*).

clear()
A coroutine that closes all *free* connections in the pool. At next connection acquiring at least *minsize* of them will be recreated.

close()
Close pool.

Mark all pool connections to be closed on getting back to pool. Closed pool doesn't allow to acquire new connections.

If you want to wait for actual closing of acquired connection please call *wait_closed()* after *close()*.

> **Warning:** The method is not a coroutine.

terminate()
Terminate pool.

Close pool with instantly closing all acquired connections also.

wait_closed() should be called after *terminate()* for waiting for actual finishing.

> **Warning:** The method is not a coroutine.

wait_closed()
A coroutine that waits for releasing and closing all acquired connections.

Should be called after *close()* for waiting for actual pool closing.

acquire()
A coroutine that acquires a connection from *free pool*. Creates new connection if needed and *size* of pool is less than *maxsize*.

Returns a `Connection` instance.

release(*conn*)
Reverts connection *conn* to *free pool* for future recycling.

> **Warning:** The method is not a coroutine.

7.5 Tutorial

Python database access modules all have similar interfaces, described by the *DBAPI*. Most relational databases use the same synchronous interface, *aiomysql* tries to provide same api you just need to use `yield from conn.f()` instead of just call `conn.f()` for every method.

7.5.1 Installation

```
pip3 install aiomysql
```

Note: *aiomysql* requires *PyMySQL* library.

7.5.2 Getting Started

Lets start from basic example:

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='', db='mysql',
                                       loop=loop)

    cur = yield from conn.cursor()
    yield from cur.execute("SELECT Host,User FROM user")
    print(cur.description)
    r = yield from cur.fetchall()
    print(r)
    yield from cur.close()
    conn.close()

loop.run_until_complete(test_example())
```

Connection is established by invoking the `connect()` coroutine, arguments list are keyword arguments, almost same as in *PyMySQL* corresponding method. Example makes connection to *MySQL* server on local host to access *mysql* database with user name *root*' and empty password.

If `connect()` coroutine succeeds, it returns a `Connection` instance as the basis for further interaction with *MySQL*.

After the connection object has been obtained, code in example invokes `Connection.cursor()` coroutine method to create a cursor object for processing statements. Example uses cursor to issue a `SELECT Host,User FROM user;` statement, which returns a list of *host* and *user* from *MySQL* system table `user`:

```
cur = yield from conn.cursor()
yield from cur.execute("SELECT Host,User FROM user")
print(cur.description)
r = yield from cur.fetchall()
```

The cursor object's *Cursor.execute()* method sends the query the server and *Cursor.fetchall()* retrieves rows.

Finally, the script invokes *Cursor.close()* coroutine and connection object's `Connection.close()` method to disconnect from the server:

```
yield from cur.close()
conn.close()
```

After that, `conn` becomes invalid and should not be used to access the server.

7.5.3 Inserting Data

Let's take basic example of *Cursor.execute()* method:

```
import asyncio
import aiomysql


async def test_example_execute(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='',
                                  db='test_pymysql', loop=loop)

    cur = await conn.cursor()
    async with conn.cursor() as cur:
        await cur.execute("DROP TABLE IF EXISTS music_style;")
        await cur.execute("""CREATE TABLE music_style
                             (id INT,
                             name VARCHAR(255),
                             PRIMARY KEY (id));""")
        await conn.commit()

        # insert 3 rows one by one
        await cur.execute("INSERT INTO music_style VALUES(1,'heavy metal')")
        await cur.execute("INSERT INTO music_style VALUES(2,'death metal');")
        await cur.execute("INSERT INTO music_style VALUES(3,'power metal');")
        await conn.commit()

    conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example_execute(loop))
```

Please note that you need to manually call `commit()` bound to your Connection object, because by default it's set to `False` or in `aiomysql.connect()` you can transfer addition keyword argument `autocommit=True`.

Example with `autocommit=True`:

```
import asyncio
import aiomysql


async def test_example_execute(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='',
                                  db='test_pymysql', loop=loop,
                                  autocommit=True)

    cur = await conn.cursor()
    async with conn.cursor() as cur:
        await cur.execute("DROP TABLE IF EXISTS music_style;")
        await cur.execute("""CREATE TABLE music_style
                             (id INT,
                             name VARCHAR(255),
                             PRIMARY KEY (id));""")

        # insert 3 rows one by one
        await cur.execute("INSERT INTO music_style VALUES(1,'heavy metal')")
        await cur.execute("INSERT INTO music_style VALUES(2,'death metal');")
        await cur.execute("INSERT INTO music_style VALUES(3,'power metal');")

    conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example_execute(loop))
```

7.6 aiomysql.sa — support for SQLAlchemy functional SQL layer

7.6.1 Intro

Note: *sqlalchemy* support ported from aiopg, so api should be very familiar for aiopg user.

While core API provides a core support for access to *MySQL* database, manipulations with raw SQL strings too annoying.

Fortunately we can use excellent SQLAlchemy Core as **SQL query builder**.

Example:

```
import asyncio
import sqlalchemy as sa

from aiomysql.sa import create_engine


metadata = sa.MetaData()

tbl = sa.Table('tbl', metadata,
               sa.Column('id', sa.Integer, primary_key=True),
               sa.Column('val', sa.String(255)))
```

(continues on next page)

(continued from previous page)

```
@asyncio.coroutine
def go():
    engine = yield from create_engine(user='root',
                                      db='test_pymysql',
                                      host='127.0.0.1',
                                      password='')

    with (yield from engine) as conn:
        yield from conn.execute(tbl.insert().values(val='abc'))

        res = yield from conn.execute(tbl.select())
        for row in res:
            print(row.id, row.val)

        await conn.commit()

asyncio.get_event_loop().run_until_complete(go())
```

So you can execute SQL query built by `tbl.insert().values(val='abc')` or `tbl.select()` expressions.

sqlalchemy has rich and very powerful set of SQL construction functions, please read tutorial for full list of available operations.

Also we provide SQL transactions support. Please take a look on *`SAConnection.begin()`* method and family.

7.6.2 Engine

aiomysql.sa.**create_engine**(*, *minsize=1*, *maxsize=10*, *loop=None*, *dialect=dialect*, ***kwargs*)

A coroutine for *`Engine`* creation.

Returns *`Engine`* instance with embedded connection pool.

The pool has *minsize* opened connections to *MySQL* server.

At *kwargs* function accepts all parameters that `aiomysql.connect()` does.

aiomysql.sa.**dialect**

An instance of *SQLAlchemy* dialect set up for *pymysql* usage.

An `sqlalchemy.engine.interfaces.Dialect` instance.

See also:

`sqlalchemy.dialects.mysql.pymysql` PyMySQL dialect.

class aiomysql.sa.**Engine**

Connects a `aiomysql.Pool` and `sqlalchemy.engine.interfaces.Dialect` together to provide a source of database connectivity and behavior.

An *`Engine`* object is instantiated publicly using the *`create_engine()`* coroutine.

dialect

A `sqlalchemy.engine.interfaces.Dialect` for the engine, readonly property.

name

A name of the dialect, readonly property.

driver
A driver of the dialect, readonly property.

minsize
A minimal size of the pool (*read-only*), `1` by default.

maxsize
A maximal size of the pool (*read-only*), `10` by default.

size
A current size of the pool (*readonly*). Includes used and free connections.

freesize
A count of free connections in the pool (*readonly*).

close()

Close engine.

Mark all engine connections to be closed on getting back to engine. Closed engine doesn't allow to acquire new connections.

If you want to wait for actual closing of acquired connection please call *wait_closed()* after *close()*.

> **Warning:** The method is not a coroutine.

terminate()

Terminate engine.

Close engine's pool with instantly closing all acquired connections also.

wait_closed() should be called after *terminate()* for waiting for actual finishing.

> **Warning:** The method is not a coroutine.

wait_closed()
A coroutine that waits for releasing and closing all acquired connections.

Should be called after *close()* for waiting for actual engine closing.

acquire()
Get a connection from pool.

This method is a coroutine.

Returns a *SAConnection* instance.

release()

Revert back connection *conn* to pool.

> **Warning:** The method is not a coroutine.

7.6.3 Connection

class aiomysql.sa.**SAConnection**

A wrapper for `aiomysql.Connection` instance.

The class provides methods for executing *SQL queries* and working with *SQL transactions*.

execute(*query*, **multiparams*, ***params*)

Executes a *SQL query* with optional parameters.

This method is a coroutine.

Parameters

- **query** – a SQL query string or any *sqlalchemy* expression (see SQLAlchemy Core)
- ***multiparams/**params** – represent bound parameter values to be used in the execution. Typically, the format is either a dictionary passed to *multiparams:

```
yield from conn.execute(
    table.insert(),
    {"id":1, "value":"v1"}
)
```

... or individual key/values interpreted by **params:

```
yield from conn.execute(
    table.insert(), id=1, value="v1"
)
```

In the case that a plain SQL string is passed, a tuple or individual values in *multiparams may be passed:

```
yield from conn.execute(
    "INSERT INTO table (id, value) VALUES (%d, %s)",
    (1, "v1")
)

yield from conn.execute(
    "INSERT INTO table (id, value) VALUES (%s, %s)",
    1, "v1"
)
```

Returns *ResultProxy* instance with results of SQL query execution.

scalar(*query*, **multiparams*, ***params*)

Executes a *SQL query* and returns a scalar value.

This method is a coroutine.

See also:

SAConnection.execute() and *ResultProxy.scalar()*.

closed

The readonly property that returns `True` if connections is closed.

begin()

Begin a transaction and return a transaction handle.

This method is a coroutine.

The returned object is an instance of *Transaction*. This object represents the "scope" of the transaction, which completes when either the *Transaction.rollback()* or *Transaction.commit()* method is called.

Nested calls to *begin()* on the same *SAConnection* will return new *Transaction* objects that represent an emulated transaction within the scope of the enclosing transaction, that is:

```
trans = yield from conn.begin()    # outermost transaction
trans2 = yield from conn.begin()   # "inner"
yield from trans2.commit()           # does nothing
yield from trans.commit()            # actually commits
```

Calls to *Transaction.commit()* only have an effect when invoked via the outermost *Transaction* object, though the *Transaction.rollback()* method of any of the *Transaction* objects will roll back the transaction.

See also:

SAConnection.begin_nested() - use a SAVEPOINT

SAConnection.begin_twophase() **- use a two phase (XA)** transaction

begin_nested()
Begin a nested transaction and return a transaction handle.

This method is a coroutine.

The returned object is an instance of *NestedTransaction*.

Any transaction in the hierarchy may `commit` and `rollback`, however the outermost transaction still controls the overall `commit` or `rollback` of the transaction of a whole. It utilizes SAVEPOINT facility of *MySQL* server.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*.

begin_twophase(*xid=None*)
Begin a two-phase or XA transaction and return a transaction handle.

This method is a coroutine.

The returned object is an instance of *TwoPhaseTransaction*, which in addition to the methods provided by *Transaction*, also provides a *prepare()* method.

Parameters **xid** – the two phase transaction id. If not supplied, a random id will be generated.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*.

recover_twophase()
Return a list of prepared twophase transaction ids.

This method is a coroutine.

rollback_prepared(*xid*)
Rollback prepared twophase transaction *xid*.

This method is a coroutine.

commit_prepared(*xid*)
Commit prepared twophase transaction *xid*.

This method is a coroutine.

in_transaction
The readonly property that returns `True` if a transaction is in progress.

close()
Close this *SAConnection*.

This method is a coroutine.

This results in a release of the underlying database resources, that is, the `aiomysql.Connection` referenced internally. The `aiomysql.Connection` is typically restored back to the connection-holding `aiomysql.Pool` referenced by the *Engine* that produced this *SAConnection*. Any transactional state present on the `aiomysql.Connection` is also unconditionally released via calling *Transaction.rollback()* method.

After *close()* is called, the *SAConnection* is permanently in a closed state, and will allow no further operations.

7.6.4 ResultProxy

class aiomysql.sa.**ResultProxy**
Wraps a *DB-API like* *Cursor* object to provide easier access to row columns.

Individual columns may be accessed by their integer position, case-sensitive column name, or by `sqlalchemy.schema.Column`` object. e.g.:

```
for row in (yield from conn.execute(...)):
    col1 = row[0]    # access via integer position
    col2 = row['col2']   # access via name
    col3 = row[mytable.c.mycol] # access via Column object.
```

ResultProxy also handles post-processing of result column data using `sqlalchemy.types.TypeEngine` objects, which are referenced from the originating SQL statement that produced this result set.

dialect
The readonly property that returns `sqlalchemy.engine.interfaces.Dialect` dialect for the *ResultProxy* instance.

See also:

dialect global data.

keys()
Return the current set of string keys for rows.

rowcount
The readonly property that returns the 'rowcount' for this result.

The 'rowcount' reports the number of rows *matched* by the WHERE criterion of an UPDATE or DELETE statement.

Note: Notes regarding *ResultProxy.rowcount*:

- This attribute returns the number of rows *matched*, which is not necessarily the same as the number of rows that were actually *modified* - an UPDATE statement, for example, may have no net change on a given row if the SET values given are the same as those present in the row already. Such a row would be matched but not modified.

- *ResultProxy.rowcount* is *only* useful in conjunction with an UPDATE or DELETE statement. Contrary to what the Python DBAPI says, it does *not* return the number of rows available from the results of a SELECT statement as DBAPIs cannot support this functionality when rows are unbuffered.
- Statements that use RETURNING does not return a correct rowcount.

---

lastrowid
Returns the 'lastrowid' accessor on the DBAPI cursor.

value generated for an *AUTO_INCREMENT* column by the previous INSERT or UPDATE statement or None when there is no such value available. For example, if you perform an *INSERT* into a table that contains an AUTO_INCREMENT column, *lastrowid* returns the *AUTO_INCREMENT* value for the new row.

returns_rows
A readonly property that returns `True` if this *ResultProxy* returns rows.

I.e. if it is legal to call the methods *ResultProxy.fetchone()*, *ResultProxy.fetchmany()*, *ResultProxy.fetchall()*.

closed
Return `True` if this *ResultProxy* is closed (no pending rows in underlying cursor).

close()
Close this *ResultProxy*.

Closes the underlying `aiomysql.Cursor` corresponding to the execution.

Note that any data cached within this *ResultProxy* is still available. For some types of results, this may include buffered rows.

This method is called automatically when:

- all result rows are exhausted using the fetchXXX() methods.
- cursor.description is None.

fetchall()
Fetch all rows, just like `aiomysql.Cursor.fetchall()`.

This method is a coroutine.

The connection is closed after the call.

Returns a list of *RowProxy*.

fetchone()
Fetch one row, just like `aiomysql.Cursor.fetchone()`.

This method is a coroutine.

If a row is present, the cursor remains open after this is called.

Else the cursor is automatically closed and `None` is returned.

Returns an *RowProxy* instance or `None`.

fetchmany(*size=None*)
Fetch many rows, just like `aiomysql.Cursor.fetchmany()`.

This method is a coroutine.

If rows are present, the cursor remains open after this is called.

Else the cursor is automatically closed and an empty list is returned.

Returns a list of *RowProxy*.

first()
Fetch the first row and then close the result set unconditionally.

This method is a coroutine.

Returns `None` if no row is present or an *RowProxy* instance.

scalar()
Fetch the first column of the first row, and close the result set.

Returns `None` if no row is present or an *RowProxy* instance.

class aiomysql.sa.**RowProxy**
A `collections.abc.Mapping` for representing a row in query result.

Keys are column names, values are result values.

Individual columns may be accessed by their integer position, case-sensitive column name, or by `sqlalchemy.schema.Column`` object.

Has overloaded operators __eq__ and __ne__ for comparing two rows.

The *RowProxy* is *not hashable*.

..method:: as_tuple()

Return a tuple with values from `RowProxy.values()`.

7.6.5 Transaction objects

class aiomysql.sa.**Transaction**
Represent a database transaction in progress.

The *Transaction* object is procured by calling the *SAConnection.begin()* method of *SAConnection*:

```
with (yield from engine) as conn:
    trans = yield from conn.begin()
    try:
        yield from conn.execute("insert into x (a, b) values (1, 2)")
    except Exception:
        yield from trans.rollback()
    else:
        yield from trans.commit()
```

The object provides *rollback()* and *commit()* methods in order to control transaction boundaries.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*, *SAConnection.begin_nested()*.

is_active
A readonly property that returns `True` if a transaction is active.

connection
A readonly property that returns *SAConnection* for transaction.

close()
Close this *Transaction*.

This method is a coroutine.

If this transaction is the base transaction in a begin/commit nesting, the transaction will *Transaction.rollback()*. Otherwise, the method returns.

This is used to cancel a *Transaction* without affecting the scope of an enclosing transaction.

rollback()
Roll back this *Transaction*.

This method is a coroutine.

commit()
Commit this *Transaction*.

This method is a coroutine.

class aiomysql.sa.**NestedTransaction**
Represent a 'nested', or SAVEPOINT transaction.

A new *NestedTransaction* object may be procured using the *SAConnection.begin_nested()* method.

The interface is the same as that of *Transaction*.

See also:

SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT on *MySQL*:

class aiomysql.sa.**TwoPhaseTransaction**
Represent a two-phase transaction.

A new *TwoPhaseTransaction* object may be procured using the *SAConnection.begin_twophase()* method.

The interface is the same as that of *Transaction* with the addition of the *TwoPhaseTransaction.prepare()* method.

xid
A readonly property that returns twophase transaction id.

prepare()
Prepare this *TwoPhaseTransaction*.

This method is a coroutine.

After a *PREPARE*, the transaction can be committed.

See also:

MySQL commands for two phase transactions:

http://dev.mysql.com/doc/refman/5.7/en/xa-statements.html

7.7 Examples of aiomysql usage

Below is a list of examples from aiomysql/examples

Every example is a correct tiny python program that demonstrates specific feature of library.

7.7.1 Low-level API

Basic example, fetch host and user information from internal table: user.

```
import asyncio
import aiomysql


async def test_example(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='', db='mysql',
                                  loop=loop)

    async with conn.cursor() as cur:
        await cur.execute("SELECT Host,User FROM user")
        print(cur.description)
        r = await cur.fetchall()
        print(r)
    conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example(loop))
```

Example of stored procedure, which just increments input value.

```
import asyncio
import aiomysql


async def test_example(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='',
                                  db='test_pymysql', loop=loop)

    async with conn.cursor() as cur:
        await cur.execute('DROP PROCEDURE IF EXISTS myinc;')
        await cur.execute("""CREATE PROCEDURE myinc(p1 INT)
                             BEGIN
                                 SELECT p1 + 1;
                             END""")

        await cur.callproc('myinc', [1])
        (ret, ) = await cur.fetchone()
        assert 2, ret
        print(ret)

    conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example(loop))
```

Example of using *executemany* method:

```
import asyncio
import aiomysql
```

(continues on next page)

(continued from previous page)

```
async def test_example_executemany(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='',
                                  db='test_pymysql', loop=loop)

    cur = await conn.cursor()
    async with conn.cursor() as cur:
        await cur.execute("DROP TABLE IF EXISTS music_style;")
        await cur.execute("""CREATE TABLE music_style
                             (id INT,
                             name VARCHAR(255),
                             PRIMARY KEY (id));""")
        await conn.commit()

        # insert 3 rows one by one
        await cur.execute("INSERT INTO music_style VALUES(1,'heavy metal')")
        await cur.execute("INSERT INTO music_style VALUES(2,'death metal');")
        await cur.execute("INSERT INTO music_style VALUES(3,'power metal');")
        await conn.commit()

        # insert 3 row by one long query using *executemane* method
        data = [(4, 'gothic metal'), (5, 'doom metal'), (6, 'post metal')]
        await cur.executemany(
            "INSERT INTO music_style (id, name)"
            "values (%s,%s)", data)
        await conn.commit()

        # fetch all insert row from table music_style
        await cur.execute("SELECT * FROM music_style;")
        result = await cur.fetchall()
        print(result)

    conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example_executemany(loop))
```

Example of using transactions *rollback* and *commit* methods:

```
import asyncio
import aiomysql


async def test_example_transaction(loop):
    conn = await aiomysql.connect(host='127.0.0.1', port=3306,
                                  user='root', password='',
                                  db='test_pymysql', autocommit=False,
                                  loop=loop)

    async with conn.cursor() as cursor:
        stmt_drop = "DROP TABLE IF EXISTS names"
        await cursor.execute(stmt_drop)
        await cursor.execute("""
            CREATE TABLE names (
            id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
```

(continues on next page)

(continued from previous page)

```
            name VARCHAR(30) DEFAULT '' NOT NULL,
            cnt TINYINT UNSIGNED DEFAULT 0,
            PRIMARY KEY (id))""")
        await conn.commit()

        # Insert 3 records
        names = (('Geert',), ('Jan',), ('Michel',))
        stmt_insert = "INSERT INTO names (name) VALUES (%s)"
        await cursor.executemany(stmt_insert, names)

        # Roll back!!!!
        await conn.rollback()

        # There should be no data!
        stmt_select = "SELECT id, name FROM names ORDER BY id"
        await cursor.execute(stmt_select)
        resp = await cursor.fetchall()
        # Check there is no data
        assert not resp

        # Do the insert again.
        await cursor.executemany(stmt_insert, names)

        # Data should be already there
        await cursor.execute(stmt_select)
        resp = await cursor.fetchall()
        print(resp)
        # Do a commit
        await conn.commit()

        await cursor.execute(stmt_select)
        print(resp)

        # Cleaning up, dropping the table again
        await cursor.execute(stmt_drop)
        await cursor.close()
        conn.close()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example_transaction(loop))
```

Example of using connection pool:

```
import asyncio
import aiomysql


async def test_example(loop):
    pool = await aiomysql.create_pool(host='127.0.0.1', port=3306,
                                      user='root', password='',
                                      db='mysql', loop=loop)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 42;")
            print(cur.description)
            (r,) = await cur.fetchone()
```

(continues on next page)

(continued from previous page)

```
            assert r == 42
    pool.close()
    await pool.wait_closed()


loop = asyncio.get_event_loop()
loop.run_until_complete(test_example(loop))
```

7.7.2 sqlalchemy usage

7.8 Glossary

DBAPI PEP 249 – Python Database API Specification v2.0

ipdb ipdb exports functions to access the IPython debugger, which features tab completion, syntax highlighting, better tracebacks, better introspection with the same interface as the pdb module.

MySQL A popular database server.

http://www.mysql.com/

pep8 Python style guide checker

pep8 is a tool to check your Python code against some of the style conventions in PEP 8 – Style Guide for Python Code.

pyflakes passive checker of Python programs

A simple program which checks Python source files for errors.

Pyflakes analyzes programs and detects various errors. It works by parsing the source file, not importing it, so it is safe to use on modules with side effects. It's also much faster.

https://pypi.python.org/pypi/pyflakes

PyMySQL Pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb and work on CPython, PyPy, IronPython and Jython.

https://github.com/PyMySQL/PyMySQL

sqlalchemy The Python SQL Toolkit and Object Relational Mapper.

http://www.sqlalchemy.org/

7.9 Contributing

Thanks for your interest in contributing to `aiomysql`, there are multiple ways and places you can contribute.

7.9.1 Reporting an Issue

If you have found issue with *aiomysql* please do not hesitate to file an issue on the GitHub project. When filing your issue please make sure you can express the issue with a reproducible test case.

When reporting an issue we also need as much information about your environment that you can include. We never know what information will be pertinent when trying narrow down the issue. Please include at least the following information:

- Version of *aiomysql* and *python*.
- Version of MySQL/MariaDB.
- Platform you're running on (OS X, Linux, Windows).

7.9.2 Instructions for contributors

In order to make a clone of the GitHub repo: open the link and press the "Fork" button on the upper-right menu of the web page.

I hope everybody knows how to work with git and github nowadays :)

Workflow is pretty straightforward:

1. Clone the GitHub repo
2. Make a change
3. Make sure all tests passed
4. Commit changes to own aiomysql clone
5. Make pull request from github page for your clone

7.9.3 Preconditions for running aiomysql test suite

We expect you to use a python virtual environment to run our tests.

There are several ways to make a virtual environment.

If you like to use *virtualenv* please run:

```
$ cd aiomysql
$ virtualenv --python=`which python3` venv
```

For standard python *venv*:

```
$ cd aiomysql
$ python3 -m venv venv
```

For *virtualenvwrapper*:

```
$ cd aiomysql
$ mkvirtualenv --python=`which python3` aiomysql
```

There are other tools like *pyvenv* but you know the rule of thumb now: create a python3 virtual environment and activate it.

After that please install libraries required for development:

```
$ pip install -r requirements-dev.txt
```

Congratulations, you are ready to run the test suite

7.9.4 Install database

Fresh local installation of *mysql* has user *root* with empty password, tests use this values by default. But you always can override host/port, user and password in *aiomysql/tests/base.py* file or install corresponding environment variables. Tests require two databases to be created before running suit:

```
$ mysql -u root
mysql> CREATE DATABASE test_pymysql  DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_
↪general_ci;
mysql> CREATE DATABASE test_pymysql2 DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_
↪general_ci;
```

7.9.5 Run aiomysql test suite

After all the preconditions are met you can run tests typing the next command:

```
$ make test
```

The command at first will run the *flake8* tool (sorry, we don't accept pull requests with pep8 or pyflakes errors).

On *flake8* success the tests will be run.

Please take a look on the produced output.

Any extra texts (print statements and so on) should be removed.

7.9.6 Tests coverage

We are trying hard to have good test coverage; please don't make it worse.

Use:

```
$ make cov
```

to run test suite and collect coverage information. Once the command has finished check your coverage at the file that appears in the last line of the output: `open file:///.../aiomysql/coverage/index.html`

Please go to the link and make sure that your code change is covered.

7.9.7 Documentation

We encourage documentation improvements.

Please before making a Pull Request about documentation changes run:

```
$ make doc
```

Once it finishes it will output the index html page `open file:///.../aiomysql/docs/_build/html/index.html`.

Go to the link and make sure your doc changes looks good.

7.9.8 The End

After finishing all steps make a GitHub Pull Request, thanks.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index