aiomyorm

Release 0.1.0

Jun 19, 2022

Contents

aiomyorm is a simple and easy-to-use ORM framework, which has a similar interface to Django and fully supports asyncio.

CHAPTER 1

Features

- Perfect support for asyncio and uvloop.
- Simple and easy to use API, similar to Django.
- Support mysql and SQLite.

CHAPTER 2

Installation

```
pip install aiomyorm
```

CHAPTER 3

Getting Started

```
from aiomyorm import Model, IntField, StringField, SmallIntField, auto_increment
from aiomyorm import set_config, close_db_connection
import asyncio

set_config(engine='sqlite',
           db='test.db')


class Test(Model):
    __table__ = 'test'
    pk = IntField(primary_key=True, default=auto_increment())
    id = StringField(50)
    age = IntField(comment='the age of student.')
    birth_place = StringField(50, default='china')
    grade = SmallIntField()


async def go():
    insert_rows = await Test.insert(Test(pk=5000, age=18, birth_place='place1'),
                          Test(pk=5001, age=21, birth_place='place2'),
                          Test(pk=5002, age=19, birth_place='place3'))
    all = await Test.find()
    print('insert rows: ', insert_rows)
        for r in all:
            print(r)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(Test.create_table())
    loop.run_until_complete(go())
    loop.run_until_complete(close_db_connection())
```

the results:

```
to create table test.
insert rows:  3
pk:5000, id:, age:18, birth_place:place1, grade:0
pk:5001, id:, age:21, birth_place:place2, grade:0
pk:5002, id:, age:19, birth_place:place3, grade:0
```

more use see *Quick Start*

CHAPTER 4

Source code

The project is in Github aiomyorm.

Feel free to file an issue.

CHAPTER 5

Dependencies

- Python >= 3.5.3
- aiomysql (for MySQL)
- aiosqlite (for sqlite)

CHAPTER 6

Tests

I have a simple test for you.

It's better for you to test in a `venv`.

first:

```
git clone git@github.com:yulansp/aiomyorm.git
```

then:

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

Recipe you must install MySQL and configure the user name and password in the `tests/test_mysql/config.py` file.

then:

```
make test
```

CHAPTER 7

License

MIT

CHAPTER 8

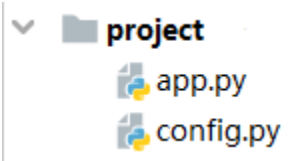
Contents

8.1 Quick Start

8.1.1 Configuration database

To use aiomyorm, first configure the database information.

The best way to configure it is to add a config.py file to your project directory, as shown in the following figure.



Then create a dict named aiomyorm in config.py, which contains your database related configuration, such as:

```
aiomyorm = {
             'maxsize': 3,
             'user': 'root',
             'password': '123456',
             'db': 'test',
             'engine': 'mysql'
         }
```

more configuration option see *set_config*

You can use the `set_config_model()` method to specify a specific Python file as the config file.

If you do not want to add an additional configuration file, you can also use the `set_config()` method to configure manually:

```
set_config(engine='sqlite',
           db='test.db')
```

8.1.2 Create Model

Like most ORM frameworks, you can create model classes to map your data tables.

```
class Test(Model):
__table__ = 'test'
pk = IntField(primary_key=True, default=auto_increment())
id = StringField(50)
age = IntField()
birth_place = StringField(50, default='china')
grade = SmallIntField()
```

more field see *field*

You can also specify the `__auto__` property to be `True`. This model will automatically query all fields from the corresponding tables in the database and map them to itself.

```
class All_Field(Model):
__table__ = 'all_field'
__auto__ = True
```

Warning: `__auto__` not support sqlite.

8.1.3 Create Table

aiomyorm allows you to create database tables directly using the framework without relying on other database management tools.

```
asyncio.get_event_loop().run_until_complete(Test.create_table())
```

8.1.4 Insert data

Inserting data with aiomyorm is simple. You only need to create a instance and call `save()` method.

```
async def go_save():
    await Test(pk=9999, age=20).save()

loop = asyncio.get_event_loop()
loop.run_until_complete(go_save())
loop.run_until_complete(close_db_connection())
```

All unassigned fields will be set to the `default` value of this `Field`.

If you want to insert more than one piece of data at a time, you need to call the `insert()` method.

```
async def go_insert():
    r = await Test.insert(Test(pk=5000, age=19, birth_place='place1'),
                          Test(pk=5001, age=21, birth_place='place2'),
                          Test(pk=5002, age=19, birth_place='place3'))
    assert r == 3

loop = asyncio.get_event_loop()
loop.run_until_complete(go_insert())
loop.run_until_complete(close_db_connection())
```

Note: `insert()` is a classmethod while `save()` is not.

8.1.5 Query from database

8.1.5.1 General query

aiomyorm provides three ways to execute queries: `find()`,‘‘pk_find()‘‘ and `find_first()`. The most common query method is `find()` and `pk_find()`.

Note: `find()`, `pk_find()` and `find_first()` are all classmethod.

query all

`find()` will execute the query based on your restrictions and return all query objects in a list, by default, he will query the entire table:

```
async def go_find():
    r = await Test.find()
    import pprint
    pprint.pprint(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_find())
loop.run_until_complete(close_db_connection())
```

results:

```
[<Test: {pk:5000, id:, age:19, birth_place:place1, grade:0}>,
 <Test: {pk:5001, id:, age:21, birth_place:place2, grade:0}>,
 <Test: {pk:5002, id:, age:19, birth_place:place3, grade:0}>,
 <Test: {pk:9999, id:, age:20, birth_place:china, grade:0}>]
```

filter results

You can use `query()` method to restrict the fields of the query and filter from the table using `filter()` method.

```
async def go_filter():
    r = await Test.query('pk', 'age').filter(age=19).find()
    import pprint
    pprint.pprint(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_filter())
loop.run_until_complete(close_db_connection())
```

results:

```
[<Test: {pk:5000, age:19}>, <Test: {pk:5002, age:19}>]
```

`filter()` will use equal as the filter condition. If you want to filter more flexibly, use `flex_filter()`.

```
async def go_flex_filter():
    r = await Test.query('pk', 'age').flex_filter(Test.age > 19).find()
    import pprint
    pprint.pprint(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_flex_filter())
loop.run_until_complete(close_db_connection())
```

results:

```
[<Test: {pk:5001, age:21}>, <Test: {pk:9999, age:20}>]
```

The `flex_filter()` can also filter data objects according to <, = or start_with, see *flex_filter()*

Warning: If you have used methods such as `filter()` to restrict query results, do not use `await` to surrender control until you execute the query method(`find()` `find_first()` and so on):

```
# This is an example of a mistake
    async def go_filter():
        Test.query('pk', 'age')
        await other_task()
        r = await Test.find()
```

change database

Sometimes your data is stored in different databases. You can use *change_db_one_time()* or *change_db()* to switch databases:

```
async def go_change_db():
    r = await Test.change_db_one_time('test2').find()
```

The difference between `change_db()` and `change_db_one_time()` is that `change_db()` will permanently change the database of this model, And `change_db_one_time()` will only change in this query.

`find_first()` and `pk_find()`

Both two methods will directly return a data object, in which `pk_find()` is based on the value of the primary key, and `find_first()` is based on the filter criteria, and the first matching object is returned.

pk_find:

```
async def go_pk_find():
    r = await Test.pk_find(5000)
    print(r)
    print(isinstance(r, Test))

loop = asyncio.get_event_loop()
loop.run_until_complete(go_pk_find())
loop.run_until_complete(close_db_connection())
```

results:

```
pk:5000, id:, age:19, birth_place:place1, grade:0
True
```

find_first:

```
async def go_find_first():
    r = await Test.flex_filter(Test.pk>5000).find_first()
    print(r)
    print(isinstance(r, Test))

loop = asyncio.get_event_loop()
loop.run_until_complete(go_find_first())
loop.run_until_complete(close_db_connection())
```

results:

```
pk:5001, id:, age:21, birth_place:place2, grade:0
True
```

Note: What is the different between `find_first()` and `limit()` ?

`Test.find_first()` will return the first object in table while `Test.limit(1).find()` will return a list, although there is only one object in the list.

more useful API

aiomyorm also provides many useful APIs, see *model*

8.1.5.2 Aggregate query

aiomyorm allows you to easily perform aggregate queries:

```
async def go_aggregate():
    r = await Test.aggregate(Count('age'), maxage=Max('age'))
    import pprint
    pprint.pprint(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_aggregate())
loop.run_until_complete(close_db_connection())
```

results:

```
{'COUNT__age': 4, 'maxage': 21}
```

or execute group aggregation query:

```
async def go_group():
    r = await Test.aggregate(Count('age'), maxage=Max('age'), group_by='age')
    import pprint
    pprint.pprint(r)

loop = asyncio.get_event_loop()
```

(continues on next page)

(continued from previous page)

```
loop.run_until_complete(go_group())
loop.run_until_complete(close_db_connection())
```

results:

```
{19: {'COUNT__age': 2, 'maxage': 19},
 20: {'COUNT__age': 1, 'maxage': 20},
 21: {'COUNT__age': 1, 'maxage': 21}}
```

aiomyorm provides the following five aggregate functions:

```
Max(), Min(), Count(), Avg(), Sum()
```

more information see *aggregate()*

8.1.6 Delete from database

aiomyorm provides two methods to delete data from the database: `delete()` and `remove()`

8.1.6.1 `remove()`

`remove()` is used to delete the object from the table:

```
async def go_remove():
    t = await Test.pk_find(5000)
    await t.remove()

loop = asyncio.get_event_loop()
loop.run_until_complete(go_remove())
loop.run_until_complete(close_db_connection())
```

8.1.6.2 `delete()`

`delete()` is used for batch deletion. It can accept filter and other filter criteria and return the number of deleted rows.

```
async def go_delete():
    r = await Test.flex_filter(Test.age>=20).delete()
    print(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_delete())
loop.run_until_complete(close_db_connection())
```

results:

```
2
```

Note: `delete()` is a classmethod while `remove()` is not.

8.1.7 Update data

Similar to delete, aiomyorm provides two methods to perform update:`save()` and `update()`

8.1.7.1 `save()`

`save()` can be used not only to create data, but also to modify data:

```
async def go_save_update():
    await Test(pk=3333, id='old_data', age=20).save()
    r = await Test.pk_find(3333)
    print('old data: ', r)
    r.id = 'new data'
    r.age = 10
    await r.save()
    r_new = await Test.pk_find(3333)
    print('new data: ', r_new)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_save_update())
loop.run_until_complete(close_db_connection())
```

results:

```
old data:  pk:3333, id:old_data, age:20, birth_place:china, grade:0
new data:  pk:3333, id:new data, age:10, birth_place:china, grade:0
```

8.1.7.2 `update()`

`update()` is used to perform batch changes, which is similar to `delete()`:

```
async def go_update():
    r = await Test.find()
    import pprint
    print('old values:')
    pprint.pprint(r)
    rows = await Test.filter(grade=0).update(age=18)
    r = await Test.find()
    print('update affect %d rows, new value is:' % rows)
    pprint.pprint(r)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_update())
loop.run_until_complete(close_db_connection())
```

results:

```
old values:
[<Test: {pk:3333, id:new data, age:10, birth_place:china, grade:0}>,
 <Test: {pk:5002, id:, age:19, birth_place:place3, grade:0}>,
 <Test: {pk:10000, id: , age:21, birth_place:place4, grade:2}>,
 <Test: {pk:10001, id: , age:26, birth_place:place5, grade:1}>]
update affect 2 rows, new value is:
[<Test: {pk:3333, id:new data, age:18, birth_place:china, grade:0}>,
 <Test: {pk:5002, id:, age:18, birth_place:place3, grade:0}>,
```

(continues on next page)

(continued from previous page)

```
 <Test: {pk:10000, id: , age:21, birth_place:place4, grade:2}>,
 <Test: {pk:10001, id: , age:26, birth_place:place5, grade:1}>]
```

Note: `update()` is a classmethod while `save()` is not.

8.1.8 Complex SQL

Using ORM framework does not support complex queries very well, so sometimes you need to execute custom SQL statements.

The next example is shown in the following table:

	pk	id	age	birth_place	grade
1	5000		19	place1	3
2	5001		21	place2	1
3	5002		19	place3	1
4	9999		20	p2	4
5	10000		20	p1	1
6	10001		18	p2	3
7	10002		15	p3	2

8.1.8.1 use `model.select`

```
async def go_select_1():
    rs = await Test.select('SELECT * FROM test WHERE age>(SELECT age FROM test WHERE␣
↪pk=5002)')
    for r in rs:
        assert isinstance(r, Test)
    import pprint
    pprint.pprint(rs)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_select_1())
loop.run_until_complete(close_db_connection())
```

results:

```
[<Test: {pk:5001, id:, age:21, birth_place:place2, grade:1}>,
 <Test: {pk:9999, id:, age:20, birth_place:p2, grade:4}>,
 <Test: {pk:10000, id: , age:20, birth_place:p1, grade:1}>]
```

8.1.8.2 use basic `select()`

```
from aiomyorm import select

async def go_select_2():
    rs = await select('SELECT * FROM test WHERE age>(SELECT age FROM test WHERE␣
↪pk=5002)')
    print(type(rs[0]))
    import pprint
    pprint.pprint(rs)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_select_2())
loop.run_until_complete(close_db_connection())
```

results:

```
<class 'dict'>
[{'age': 21, 'birth_place': 'place2', 'grade': 1, 'id': '', 'pk': 5001},
 {'age': 20, 'birth_place': 'p2', 'grade': 4, 'id': '', 'pk': 9999},
 {'age': 20, 'birth_place': 'p1', 'grade': 1, 'id': ' ', 'pk': 10000}]
```

Note: What's the difference between `Model.select()` and `select()` ?

They will all return a list containing query results. The difference is that each item in the list returned by `select()` is a dict, while `Model.select()` is a Model object.

8.1.8.3 execute insert, update and delete

To do a custom insert, update or delete, you can use either `execute()` or `Model.execute()`, which have the same performance and will return the number of rows affected.

```
async def go_execute():
    rs = await execute('UPDATE test set id="little boy" WHERE age>(SELECT age FROM␣
↪test WHERE pk=5002)')
    print(rs)

loop = asyncio.get_event_loop()
loop.run_until_complete(go_execute())
loop.run_until_complete(close_db_connection())
```

results:

```
3
```

8.1.9 Transaction

Every time aiomyorm accesses the database, transactions will be enabled by default, but sometimes you need to perform multiple queries in a transaction. aiomyorm provides this method for you:

```
async with Transaction() as conn:
    # some query ...
```

You only need to use method `use()` or directly specify `conn` in each query to use this `conn`, then all queries will be in the same transaction, and if there is an error, it will be automatically rolled back.

```
async def go_transaction():
    rs_old = await Test.find()
    rs_trans = None
    try:
        async with Transaction() as conn:
            r = await execute('insert into test (id,age,birth_place,grade) values (
↪"00020",18,"",1)', conn=conn)
            rs_trans = await Test.use(conn).find()
            100 / 0  # make error
    except ZeroDivisionError:
        pass
    assert r == 1
    rs_new = await Test.find()

    import pprint
    print('old:')
    pprint.pprint(rs_old)
    print('in transaction:')
    pprint.pprint(rs_trans)
    print('new:')
    pprint.pprint(rs_new)

    assert rs_old == rs_new
    assert rs_old != rs_trans
```

results:

```
old:
[<Test: {pk:5000, id:, age:19, birth_place:place1, grade:0}>,
 <Test: {pk:5001, id:, age:21, birth_place:place2, grade:0}>,
 <Test: {pk:5002, id:, age:19, birth_place:place3, grade:0}>]
in transaction:
[<Test: {pk:5000, id:, age:19, birth_place:place1, grade:0}>,
 <Test: {pk:5001, id:, age:21, birth_place:place2, grade:0}>,
 <Test: {pk:5002, id:, age:19, birth_place:place3, grade:0}>,
 <Test: {pk:5005, id:00020, age:18, birth_place:, grade:1}>]
new:
[<Test: {pk:5000, id:, age:19, birth_place:place1, grade:0}>,
 <Test: {pk:5001, id:, age:21, birth_place:place2, grade:0}>,
 <Test: {pk:5002, id:, age:19, birth_place:place3, grade:0}>]
```

Thanks for reading!

8.2 set_config

Config your database.

The best way is to include a config.py file in your project, and then a dict named aiomyorm, which contains your database configuration, as follow:

```
aiomyorm = {
        'maxsize': 3,
        'user': 'root',
```

(continues on next page)

(continued from previous page)

```
        'password': '123456',
        'db': 'test',
        'engine': 'mysql'
    }
```

By default, you need to include the config.py file in the same level directory of your main file. You can call the set_config_model() method to specify the location of the config file again, as follow:

```
set_config_model('myconfigdirectory.myconfig')
```

If you don't want to configure the config.py file, you can also call the set_config() method to configure your database connection, as follow:

```
set_config(engine='sqlite',
   db='test.db')
```

configuration options:

common:

- engine (str) : database engine (`'mysql', 'sqlite'`)

for sqlite:

- db (str) : the database file for sqlite.
- other options : same as sqlite3.

for mysql:

- db (str) : database to use, None to not use a particular one.
- minsize (int) : minimum sizes of the pool.
- maxsize (int) : maximum sizes of the pool.
- echo (bool) : executed log SQL queryes default : False.
- debug (bool) : echo the debug log default : False.
- host (str) : host where the database server is located, default : localhost.
- port (str) : MySQL port to use, default : 3306.
- user (str) : username to log in as.
- password (str) : password to use.
- unix_socket (str) :optionally, you can use a unix socket rather than TCP/IP.
- charset (str) : charset you want to use, for example 'utf8'.
- sql_mode: default sql-mode to use, like 'NO_BACKSLASH_ESCAPES'
- read_default_file : specifies my.cnf file to read these parameters from under the [client] section.See aiomysql.
- conv : decoders dictionary to use instead of the default one. This is used to provide custom marshalling of types. See pymysql.converters.
- client_flag : custom flags to send to MySQL. Find potential values in pymysql.constants.CLIENT.
- use_unicode : whether or not to default to unicode strings.
- connect_timeout : Timeout in seconds before throwing an exception when connecting.

- autocommit : Autocommit mode. None means use server default. (default: False)
- ssl : Optional SSL Context to force SSL
- server_public_key : SHA256 authenticaiton plugin public key value.
- read_default_group (str) : Group to read from in the configuration file.
- no_delay (bool) : disable Nagle's algorithm on the socket
- 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)
- program_name: Program name string to provide when handshaking with MySQL. (default: sys.argv[0])
- server_public_key: SHA256 authenticaiton plugin public key value.
- loop : is an optional event loop instance, asyncio.get_event_loop() is used if loop is not specified.
- init_command (str) : initial SQL statement to run when connection is established.

default:

```
minsize=1,
maxsize=10,
echo=False,
debug=False,
host="localhost",
port=3306,
user=None,
password="",
db=None,
unix_socket=None,
charset='',
sql_mode=None,use_unicode=None,
read_default_file=None,
connect_timeout=None,
autocommit=False,
ssl=None,
server_public_key=None,
loop=None,
auth_plugin='',
program_name='',
read_default_group=None,
no_delay=False,
init_command=None
```

aiomyorm.set_config.**set_config**(***kwargs*)
Set configs manually.

Parameters **kwargs** – see module's __doc__ configuration options.

aiomyorm.set_config.**set_config_model**(*config_model*)
Set a custom model as config model.

aiomyorm.set_config.**set_loop**(*loop*)
set a event loop.

aiomyorm.set_config.**unset_loop**(*loop*)
unset the event loop.

8.3 model

class aiomyorm.model.**Model**(*_new_created: bool = True*, *_pk_value=None*, ***kwargs*)

The basic model class.

__table__

The table name of this model.If you do not set this attribute, class name will be used by default.

__auto__

If true, fields will automatically retrieved from the table, default False. **Warnings: __auto__ dose not support sqlite.**

the field you define

All the fields you define. In class it will be field while in instance it is the value of this field.

classmethod aggregate(**args*, *group_by: Union[str, aiomyorm.field.Field, None] = None*, *conn=None*, ***kwargs*) → Union[List[Dict[KT, VT]], Dict[KT, VT], NoReturn]

Aggregate query.

Parameters

- **args** – aggregate function, it's alias will be taken as `(function)__(field name)`, e.g. `MAX__age`. you can use the follow aggregate function:

```
Max(), Min(), Count(), Avg(), Sum()
```

- **group_by** (*str*) – Same as sql, and only allow one field.
- **kwargs** – key is the alias of this aggregate field,and value is the aggregate function.
- **conn** – custom connection (this parameter is optional)

Returns If the group_by parameter is not specified, will return a dict which key is it's alias and value is it's result. If the group_by parameter is specified, will return a dict which key is the result of the group_by field in each group, and value is a dict as same as the first situation.

Sample: you can run this code:

```
from model import Max,Count,Min,Avg
async def run():
    rs = await Test.filter(grade=3).aggregate(Max('age'), minage=Min('age
↪'), avgage=Avg('age'),
                                              groupnum=Count(), group_by=
↪'birth_place')
    import pprint
    pprint.pprint(rs)

asyncio.get_event_loop().run_until_complete(run())
```

you will get the result:

```
{'someplace': {'MAX__age': 20, 'avgage': 20.0, 'groupnum': 1, 'minage':␣
↪20},
 'someplace1': {'MAX__age': 23, 'avgage': 20.5, 'groupnum': 2, 'minage':␣
↪18},
 'someplace3': {'MAX__age': 17, 'avgage': 17.0, 'groupnum': 1, 'minage':␣
↪17}}
```

classmethod change_db(*db: str*)
Change the database this model belongs to.

Warnings: not support sqlite

classmethod change_db_one_time(*db: str*)
Change the database one time.

You can use this method temporarily change the database this model belongs in the next query or modification.

e.g.:

```
r = await Model.change_db_one_time(newdb).find()
```

This query will be performed in newdb.

Warnings: not support sqlite

classmethod create_table()
Create this table in current database.

classmethod delete(*conn=None*) → int
Delete objects from database.

You can use `filter()` and other method to filter the objects that want to delete, just as you did in `find()`.

Parameters **conn** – custom connection (this parameter is optional)

Returns (int) Total number of objects deleted.

classmethod distinct()
Add DISTINCT condition for your query.

classmethod exclude(***kwargs*) → None
Exclude filter your query.

Parameters **kwargs** – the field you wang to exclude and it's value.

Raises `ValueError` – An error occurred when argument is not a Field.

classmethod execute(*sql: str*, *args: Union[list*, *tuple*, *None] = ()*, *conn=None*) → int
Execute a insert,update or delete query, and return the number of affected rows.You can use this method when you encounter a query that ORM cannot complete.

Parameters

- **sql** (*str*) – a sql statement, use ? as placeholder.
- **args** (*list or tuple*) – argument in placeholder.
- **conn** – use this parameter to specify a custom connection.

Returns (int) affected rows.

classmethod filter(***kwargs*) → None
Filter your query,correspond to where key=value in sql.

Parameters **Kwargs** – The field you wang to filter and it's value.

Raises `ValueError` – An error occurred when argument is not a Field.

classmethod find(*conn=None*)
Do select.This method will return a list of YourModel objects.

Parameters **conn** – custom connection (this parameter is optional)

Returns (list) A list of YourModel objects.If no record can be found, will return an empty list.

classmethod find_first(*conn=None*)
Do select.This method will directly return one YourModel object instead of a list.

Parameters **conn** – custom connection (this parameter is optional)

Returns (Model) One YourModel object.If no record can be found, will return None.

classmethod flex_exclude(**conditions*)
Exclude your query flexibly, such as '>' '<' and so on.

Parameters **field you wang to exclude and it's value.** (*The*) –

You can use the following methods: same as `flex_filtter()`

Raises `ValueError` – An error occurred when you use it in the wrong way.

classmethod flex_filter(**conditions*)
Filter your query flexibly, such as '>' '<' and so on.

Parameters **conditions** – The field you wang to filter and it's value.

You can use the following methods:

```
flex_filter(Table.Field>100)                --in sql-->  where Table.Field>100
flex_filter(Table.Field>=100)               --in sql-->  where Table.Field>=100
flex_filter(Table.Field==100)               --in sql-->  where Table.Field=100
flex_filter(Table.Field<=100)               --in sql-->  where Table.Field<=100
flex_filter(Table.Field<100)                --in sql-->  where Table.Field<100
flex_filter(Table.Field.like('%abc%'))      --in sql-->  where Table.Field␣
↪LIKE '%abc%'
flex_filter(Table.Field.start_with(abc))    --in sql-->  where Table.Field␣
↪LIKE 'abc%'
flex_filter(Table.Field.end_with(abc))      --in sql-->  where Table.Field␣
↪LIKE '%abc'
flex_filter(Table.Field.has('abc'))         --in sql-->  where Table.Field␣
↪LIKE '%abc%'
```

Raises `ValueError` – An error occurred when you use it in the wrong way.

classmethod get_db()
Get the database this model belongs to.

classmethod get_fields()
Get the names of all fields

classmethod get_mapping()
Get the fields mapping.

classmethod get_primary_key()
Get the name of the primary key

classmethod insert(**insert_objects*, *conn=None*) → int
Insert objects to database.

This method can insert multiple objects and access the database only once.

Parameters

- **insert_objects** (`Model`) – One or more object of this Model.They must have the same format.

- **conn** – custom connection (this parameter is optional)

Raise: ValueError: An error occurred when argument is not the object of this model. RuntimeError: An error occurred when arguments do not have the same format.

Returns (int) Affected rows.

classmethod or_connect()
Connect your filter br 'OR' rather than 'AND'.

classmethod order_by(**args*)
Sort query results.

By default, it will be sorted from small to large. You can sort it from large to small by adding '-'.

Parameters **args** – (str) The sorting basis you specified.

Example:

```
User.query('id').order_by('id').find()
# will sort by id from small to large
User.query('id').order_by('-id').find()
# will sort by id from large to small
```

classmethod pk_find(*pk_value*, *conn=None*)
Get one object by primary key.

You should specify the primary key in this method. All the restrictions you have made before, except "`query()`", will not take effect. This method will directly return one YourModel object.

Parameters

- **pk_value** – value of primary key.
- **conn** – custom connection (this parameter is optional)

Returns (Model) One YourModel object.If no record can be found, will return None.

classmethod query(**args*) → None
Choice the field you want to query.

All the query method such as `find()` will query all the fields by default, so if you want to query all fields, just do not use this method.

Parameters **args** – The field you wang to query.You can use Model.Field or the name of field.

e.g.:

```
r = await User.query(User.id).find()
r = await User.query('id').find()
```

Raises `ValueError` – An error occurred when argument is not a Field.

remove(*conn=None*) → NoReturn
Remove this object from database.

Parameters **conn** – custom connection (this parameter is optional)

Raise: RuntimeError: An error occurred when can not find a primary key of this object. since every object queried from the database will query primary key explicitly or implicitly, this error will only appear

when the newly created object calls the `remove()` method, which itself is the wrong use. To delete an object without querying in advance, use `delete()` method.

Returns None

save(*conn=None*) → NoReturn
Save this object to database.

Parameters **conn** – custom connection (this parameter is optional)

Raise: RuntimeError: An error occurred when failed to save this object. AttributeError: An error occurred when primary key has been changed, which is not allowed in this method, use `update()`.

Returns None

classmethod select(*sql: str*, *args: Union[list*, *tuple*, *None] = ()*, *conn=None*) → list
Execute a select query, and turn the result into a model object. You can use this method when you encounter a query that ORM cannot complete

Parameters

- **sql** (*str*) – a sql statement, use ? as placeholder.
- **args** (*list or tuple*) – argument in placeholder.
- **conn** – use this parameter to specify a custom connection.

Returns (list) a list of model objects.

classmethod update(*conn=None*, ***kwargs*) → int
Update objects to database.

You can use `filter()` and other method to filter the objects that want to update, just as you did in `find()`.

Parameters **conn** – custom connection (this parameter is optional)

Returns (int) Total number of objects updated.

classmethod use(*conn=None*)
Specify a connection.

8.4 connection

aiomyorm.connection.**Connection**()
A async context manager to run a custom sql statement.

Creates new connection.Returns a Connection instance. You can also use this connection in ORM by specifying the conn parameter. If you have not set autocommit=True, you should commit manual by use `conn.commit()`.

aiomyorm.connection.**Transaction**()
Get a connection to do atomic transaction.

This is a subclass of Connection and they have the same usage, and on exit, this connection will automatically commit or roll back on error. You can also use this connection in ORM by specifying the conn parameter. Example:

```
async whit connection.Transaction() as conn:
    await Table(tl1='abc',tl2=123).save(conn=conn)
```

aiomyorm.connection.**close_db_connection**()
Close connection with database.You may sometime need it.

aiomyorm.connection.**execute**(*sql: str*, *args: Union[list, tuple, None] = ()*, *conn: Optional[aiomyorm.connection.Connection] = None*) → int
Execute a insert,update or delete query, and return the number of affected rows.You can use this method when you encounter a query that ORM cannot complete.

Parameters

- **sql** (*str*) – a sql statement, use ? as placeholder.
- **args** (*list or tuple*) – argument in placeholder.
- **conn** – use this parameter to specify a custom connection.

Returns (int) affected rows.

aiomyorm.connection.**select**(*sql: str*, *args: Union[list, tuple, None] = ()*, *conn: Optional[aiomyorm.connection.Connection] = None*) → list
Execute a select query, and return a list of result.You can use this method when you encounter a query that ORM cannot complete

Parameters

- **sql** (*str*) – a sql statement, use ? as placeholder.
- **args** (*list or tuple*) – argument in placeholder.
- **conn** – use this parameter to specify a custom connection.

Returns (list) a list of result.

8.5 field

8.5.1 Basic Field

This is the most basic `Field` class, and all the common properties of all `Field` are given here, but I do not recommend you to use this class. I provide a more customized class later.

class aiomyorm.field.**Field**(*column_type*, *default*, *primary_key=False*, *null=False*, *comment=None*, *unsigned=None*)
A field is a mapping of a column in mysql table.

column_type
Type of this column.

Type str

primary_key
Whether is a primary key.

Type bool

default
The default value of this column, it can be a real value or a function.

Type any

belong_model
The model(table) this field(column) belongs to.

Type str

comment
Comment of this field.

Type str

null
Allow null value or not.

Type bool

unsigned
Whether unsigned. Only useful in Integer.

Type bool

8.5.2 Integer Field

class aiomyorm.field.**BoolField**(*primary_key=False*, *default=0*, *null=False*, *comment=''*)
Bases: *aiomyorm.field.Field*

A bool field.

class aiomyorm.field.**SmallIntField**(*primary_key=False*, *default=0*, *null=False*, *comment=''*, *unsigned=False*)
Bases: *aiomyorm.field.Field*

A smallint field.

class aiomyorm.field.**MediumIntField**(*primary_key=False*, *default=0*, *null=False*, *comment=''*, *unsigned=False*)
Bases: *aiomyorm.field.Field*

A mediumint field.

class aiomyorm.field.**IntField**(*primary_key=False*, *default=0*, *null=False*, *comment=''*, *unsigned=False*)
Bases: *aiomyorm.field.Field*

A int field.

class aiomyorm.field.**BigIntField**(*primary_key=False*, *default=0*, *null=False*, *comment=''*, *unsigned=False*)
Bases: *aiomyorm.field.Field*

A bigint field.

8.5.3 String Field

class aiomyorm.field.**StringField**(*length: int = 255*, *primary_key=False*, *default=''*, *null=False*, *comment=''*)
Bases: *aiomyorm.field.Field*

A string field.

Parameters **length** (*int*) – Maximum length of string in this field, default by 255.

class aiomyorm.field.**FixedStringField**(*length: int = 255*, *primary_key=False*, *default=''*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A fixed length string field.

8.5.4 Text Field

class aiomyorm.field.**TextField**(*primary_key=False*, *default=''*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A text field.

class aiomyorm.field.**MediumTextField**(*primary_key=False*, *default=''*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A medium text field.

class aiomyorm.field.**LongTextField**(*primary_key=False*, *default=''*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A long text field.

8.5.5 Decimal Field

class aiomyorm.field.**FloatField**(*total_digits: int = 255*, *decimal_digits: int = 30*, *primary_key=False*, *default=0.0*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A float field.

Parameters

- **total_digits** (*int*) – total digit for this float, default by 255.
- **decimal_digits** (*int*) – total decimal digit, default by 30.

class aiomyorm.field.**DoubleField**(*total_digits: int = 255*, *decimal_digits: int = 30*, *primary_key=False*, *default=0.0*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A double field.

Parameters

- **total_digits** (*int*) – total digit for this float, default by 255.
- **decimal_digits** (*int*) – total decimal digit, default by 30.

class aiomyorm.field.**DecimalField**(*total_digits: int = 65*, *decimal_digits: int = 30*, *primary_key=False*, *default=0.0*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A decimal field which is more precise than float or double.

Parameters

- **total_digits** (*int*) – total digit for this float, default by 65.
- **decimal_digits** (*int*) – total decimal digit, default by 30.

8.5.6 DateTime Field

class aiomyorm.field.**DatetimeField**(*primary_key=False*, *default=datetime.datetime(2022, 6, 19, 14, 58, 50)*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A datetime field, default value is now.

class aiomyorm.field.**DateField**(*primary_key=False*, *default=datetime.date(2022, 6, 19)*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A datetime field, default value is today.

class aiomyorm.field.**TimeField**(*primary_key=False*, *default=datetime.timedelta(seconds=53930)*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A time field, default value is now time.

class aiomyorm.field.**TimestampField**(*primary_key=False*, *default=datetime.datetime(2022, 6, 19, 14, 58, 50)*, *null=False*, *comment=''*)

Bases: *aiomyorm.field.Field*

A timestamp field, default value is now.

8.5.7 More Field

You can customize your field class based on the `Field`, such as:

```
class JsonField(Field):
    """A json field."""
    def __init__(self, primary_key=False, default="{}", null=False, comment=''):
        super().__init__(column_type='json', primary_key=primary_key, default=default,
↪ null=null,
                         comment=comment)
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U