saiorm Documentation

Release stable

Mar 01, 2020

Contents

Saiorm : surely an incorrect orm.

Saiorm is a very lightweight translator for accessing kinds of database with the same syntax,including SQL and NoSQL.

It only translate python code and arguments to database statement,no longer need models.Directly operate the data in the database. No data type conversion, minimize the performance loss.

Support MySQL, PostgreSQL, SQL Server ,SQLite and MongoDB,require pymysql psycopg2 pymssql sqlite3 pymongo for each database type.

It will take you have an easy way to use kinds of database with the same syntax,including SQL and NoSQL.The syntax looks like a mixture of SQL and mongodb.

You can inherit from saiorm.base.ChainDB to support other types of database with the same API.

Common methods,both SQL and NoSQL:

- **insert, select, update, delete, increase, decrease** should be called **finally**,they will take effect immediately.
- **select** return all data with list.
- **get** return the latest line with dict.
- **update**, **delete**, **execute** return a dict,including lastrowid, rowcount, rownumber, query.

Different methods

- Special methods to SQL databases:

  You can call **execute** and **executemany** to execute SQL.

  get_fields_name get a list of all fields name, cache them by default.

  where can receive list type(recommend) or string type.

  Use various **join**,should use string for **join** and **where**.

- last query:

  MySQL and **PostgreSQL** returns by default.

  SQL Server , **MongoDB** , **SQLite** return empty string by default,pass **return_query=True** when calling **connect** to enable it.

  Because the package they required does not return it, it's generated by saiorm via formating query string with params, maybe not real.

- native function

  Add ‘ as a prefix.

- **MongoDB is not full support:**

  Only support select,get,update,insert,insert_many,delete,increase,decrease,where,limit,order_by

  where receive list type

ATTENTION

Saiorm does not convert value type in native functions and IN and other condition(eg.limit,order_by,group_by,various join). If you want to use the values passed from user,you must check them,because it's easily to triggering injection vulnerability.

CHAPTER 1

Initialization

saiorm.init() use MySQL by default,you could set database type by param **driver** explicitly.

MySQL:

```
import saiorm
DB = saiorm.init()  # without table name prefix,default driver is MySQL
# or
DB = saiorm.init(driver="MySQL",table_name_prefix="abc_") # mysql with table name␣
↪prefix
DB.connect({"host": "", "port": 3306, "database": "", "user": "", "password": ""})
table = DB.table("xxx")
```

PostgreSQL:

```
import saiorm
DB = saiorm.init(driver="PostgreSQL")   # without table name prefix
# or
DB = saiorm.init(driver="PostgreSQL", table_name_prefix="abc_") # with table name␣
↪prefix
DB.connect({"host": "", "port": "5432", "database": "", "user": "", "password": ""})
table = DB.table("xxx")
```

SQL Server:

You should pass **primary_key** to method table,because SQL Server does not support LIMIT,we will use primary_key to implement method limit.

```
import saiorm
DB = saiorm.init(driver="SQLServer")   # without table name prefix
# or
DB = saiorm.init(driver="SQLServer", table_name_prefix="abc_") # with table name␣
↪prefix
DB.connect({"host": "", "port": "1433", "database": "", "user": "", "password": ""})
```

(continues on next page)

(continued from previous page)

```
# DB.connect({"host": "", "port": "1433", "database": "", "user": "", "password": ""},
↪ return_query=True) # can get latest query you executed
table = DB.table("xxx", primary_key="id")  # For LIMIT implement with SQL Server
```

SQLite:

The only param **host** should be the path to db file.

```
import saiorm
DB = saiorm.init(driver="SQLite")   # without table name prefix
# or
DB = saiorm.init(driver="SQLite", table_name_prefix="abc_") # with table name prefix
DB.connect({"host": "test.db"})
# DB.connect({"host": "test.db"}, return_query=True) # can get latest query you␣
↪executed
table = DB.table("xxx")
```

MongoDB:

```
import saiorm
DB = saiorm.init(driver="MongoDB")
DB.connect({"host": "127.0.0.1", "port": "27017", "database": "x", "user": "",
↪"password": ""})
# or
DB.connect({"host": "127.0.0.1", "port": "27017", "database": "x", "user": "",
↪"password": ""}, return_query=True)# can get latest query you executed
table = DB.table("xxx")
```

The SQL in usages following is MySQL style,it's a little different from PostgreSQL and SQL Server, especially LIMIT.

CHAPTER 2

Usage for calling native function

```
DB.select("`NOW()")
DB.select("`SUM(1+2)")
```

will be transformed to SQL:

```
SELECT NOW();
SELECT SUM(1+2);
```

CHAPTER 3

Usage for select and get

- select and get receive a fields param,but invalid to MongoDB.
- select will return all data.
- get will overwrite method limit automatically,then return the latest line only.

```
# select all fields
table.select()

# get the latest line
table.order_by("id DESC").get()

# kinds of params in where,all by AND
table.where([
    ("a", 1),
    ("b", "BETWEEN", "1", "2"),
    ("c", "`ABS(?)", "2"),
    ("d", "!=", 0),
    ("e", "IN", ["1", "2", "3"]),
    ("f", "`ABS(-2)"),
]).select("e,f")

# kinds of params in where,mixing AND and OR
table.where([
    ("a", "OR", 1),
    ("b", "OR", "BETWEEN", "1", "2"),
    ("c", "OR", "`ABS(?)", "2"),
    ("d", "OR", "IS NOT", "NULL"),
    ("e", "NOT IN", ["1", "2", "3"]),
    ("f", "`ABS(-2)"),
]).select("e,f")
```

will be transformed to SQL:

```
SELECT * FROM xxx ;
SELECT * FROM xxx  ORDER BY id DESC LIMIT 1;
SELECT `e`,`f` FROM xxx WHERE a=1 AND b BETWEEN 1 AND 2 AND c=ABS(2) AND d!=0 AND e␣
↪IN (1,2,3) AND f=ABS(-2) ;
SELECT `e`,`f` FROM xxx WHERE a=1 OR b BETWEEN 1 AND 2 OR c=ABS(2) OR d IS NOT NULL␣
↪OR e NOT IN (1,2,3) AND f=ABS(-2)
```

CHAPTER 4

Usage for update

If you want use native function,you can pass a tuple.

```
table.where([
    ("a", "IN", ["1", "2", "3"]),
    ("b","`ABS(?)", "2"),
]).update({
    "c": "`ABS(2)",
    "d": ("`ABS(?)", 3),
    "e": "2",
})
```

will be transformed to SQL:

```
UPDATE xxx SET c=ABS(2),d=ABS(3),e='2' WHERE a IN (1,2,3) AND b=ABS(2) ;
```

CHAPTER 5

Usage for insert

insert function support two kinds of data

```
# use natural dict
table.insert({
    "a": "1",
    "b": "2",
})

# use split dict
table.insert({
    "fields": ["a", "b"],
    "values": ["1", "2"],
})

# use natural dict in list, SQL will in one line
table.insert_many([{
    "a": "1",
    "b": "2",
}, {
    "a": "3",
    "b": "4",
}, {
    "a": "5",
    "b": "6",
}])

# use split dict in list, SQL will in one line
table.insert_many({
    "fields": ["a", "b"],
    "values": [
        ["1", "2"],
        ["3", "4"],
        ["5", "6"]
    ]
})
```

will be transformed to SQL:

```
INSERT INTO xxx (a,b) VALUES ('1','2');
INSERT INTO xxx (a,b) VALUES ('1','2');
INSERT INTO xxx (a,b) VALUES ('1','2'),('3','4'),('5','6');
INSERT INTO xxx (a,b) VALUES ('1','2'),('3','4'),('5','6');
```

If pass split dict to insert or insert_many,fields is not necessary, if the dict has values only,it will insert by the order of table struct.

CHAPTER 6

Usage for delete

By default, **delete** must have **where** condition,or you can pass strict=False when initialization.

```
table.where({
    "a": "1",
    "b": "2",
    "c": ("`ABS(?)", "2"),
}).delete()

table.delete()  # will not be executed, or set strict=False when initialization
```

will be transformed to SQL:

```
DELETE FROM xxx WHERE a='1' AND b='2' AND c=ABS(2) ;
DELETE FROM xxx ;
```

CHAPTER 7

Usage for increase

Numerical field increase

```
table.increase("a", 1)
```

will be transformed to SQL:

```
UPDATE xxx SET a=a+1
```

CHAPTER 8

Usage for decrease

Numerical field decrease

```
table.decrease("a", 1)
```

will be transformed to SQL:

```
UPDATE xxx SET a=a-1
```

CHAPTER 9

Usage for left join

```
DB.table("tableA AS a")
  .left_join("tableB AS b")
  .on("b.bb = a.aa")
  .where([("expire_time", ">", now)])
  .select("a.*,b.disabled_function")
```

CHAPTER 10

Method limit and offset

Param should be str type.

basic usage:

```
table.limit(number)
```

with offset:

```
table.limit(number).offset(number)
```

CHAPTER 11

Method where

```
table.where([
    ("a", 1),
    ("b", "OR", "BETWEEN", "1", "2"),
    ("c", "!=", "`ABS(?)", "2"),
    ("d", "OR", "IS NOT", "NULL"),
    ("e", "NOT IN", ["1", "2", "3"]),
    ("f", "`ABS(-2)"),
]).select("e,f")
```

- The default parallel relationship with the next condition is AND,use tuple or list with the first item “or” to toggle to “or”.
- Condition will be equals value,or pass a second item(like !=) to change it.
- When calling native function the param placeholder should be ?.
- Pass string type is allowed with SQL databases.