import sys

import asyncio

import logging

logging.basicConfig(level=logging.INFO)

# 一次使用异步 处处使用异步

import aiomysql

def log(sql,args=()):

logging.info('SQL:%s' %sql)

@asyncio.coroutine

def create\_pool(loop, \*\*kw): #这里的\*\*kw是一个dict

logging.info(' start creating database connection pool')

global \_\_pool

# 理解这里的yield from 是很重要的

#dict有一个get方法，如果dict中有对应的value值，则返回对应于key的value值，否则返回默认值，例如下面的host，如果dict里面没有

#'host',则返回后面的默认值，也就是'localhost'

#这里有一个关于Pool的连接，讲了一些Pool的知识点，挺不错的，[点击打开链接](http://aiomysql.readthedocs.io/en/latest/pool.html" \t "https://blog.csdn.net/haskei/article/details/_blank)，下面这些参数都会讲到，以及destroy\_\_pool里面的

#wait\_closed()

\_\_pool=yield from aiomysql.create\_pool(

host=kw.get('host','localhost'),

port=kw.get('port',3306),

user=kw['user'],

password=kw['password'],

db=kw['db'],

charset=kw.get('charset','utf8'),

autocommit=kw.get('autocommit',True), #默认自动提交事务，不用手动去提交事务

maxsize=kw.get('maxsize',10),

minsize=kw.get('minsize',1),

loop=loop

)

@asyncio.coroutine

def destroy\_pool():

global \_\_pool

if \_\_pool is not None :

\_\_pool.close() #关闭进程池,The method is not a coroutine,就是说close()不是一个协程，所有不用yield from

yield from \_\_pool.wait\_closed() #但是wait\_close()是一个协程，所以要用yield from,到底哪些函数是协程，上面Pool的链接中都有

# 我很好奇为啥不用commit 事务不用提交么？我觉得是因为上面再创建进程池的时候，有一个参数autocommit=kw.get('autocommit',True)

# 意思是默认会自动提交事务

@asyncio.coroutine

def select(sql, args, size=None):

log(sql,args)

global \_\_pool

# 666 建立游标

# -\*- yield from 将会调用一个子协程，并直接返回调用的结果

# yield from从连接池中返回一个连接， 这个地方已经创建了进程池并和进程池连接了，进程池的创建被封装到了create\_pool(loop, \*\*kw)

with (yield from \_\_pool) as conn: #使用该语句的前提是已经创建了进程池，因为这句话是在函数定义里面，所以可以这样用

cur = yield from conn.cursor(aiomysql.DictCursor) #A cursor which returns results as a dictionary.

yield from cur.execute(sql.replace('?', '%s'), args)

if size:

rs = yield from cur.fetchmany(size) #一次性返回size条查询结果，结果是一个list，里面是tuple

else:

rs = yield from cur.fetchall() #一次性返回所有的查询结果

yield from cur.close() #关闭游标，不用手动关闭conn，因为是在with语句里面，会自动关闭，因为是select，所以不需要提交事务(commit)

logging.info('rows have returned %s' %len(rs))

return rs #返回查询结果，元素是tuple的list

# 封装INSERT, UPDATE, DELETE

# 语句操作参数一样，所以定义一个通用的执行函数，只是操作参数一样，但是语句的格式不一样

# 返回操作影响的行号

# 我想说的是 知道影响行号有个叼用

@asyncio.coroutine

def execute(sql,args, autocommit=True):

log(sql)

global \_\_pool

with (yield from \_\_pool) as conn:

try:

# 因为execute类型sql操作返回结果只有行号，不需要dict

cur = yield from conn.cursor()

# 顺便说一下 后面的args 别掉了 掉了是无论如何都插入不了数据的

yield from cur.execute(sql.replace('?', '%s'), args)

yield from conn.commit() #这里为什么还要手动提交数据

affected\_line=cur.rowcount

yield from cur.close()

print('execute : ', affected\_line)

except BaseException as e:

raise

return affected\_line

# 这个函数主要是把查询字段计数 替换成sql识别的?

# 比如说：insert into `User` (`password`, `email`, `name`, `id`) values (?,?,?,?) 看到了么 后面这四个问号

def create\_args\_string(num):

lol=[]

for n in range(num):

lol.append('?')

return (','.join(lol))

# 定义Field类，负责保存(数据库)表的字段名和字段类型

class Field(object):

# 表的字段包含名字、类型、是否为表的主键和默认值

def \_\_init\_\_(self, name, column\_type, primary\_\_key, default):

self.name = name

self.column\_type=column\_type

self.primary\_key=primary\_\_key

self.default=default

def \_\_str\_\_(self):

# 返回 表名字 字段名 和字段类型

return "<%s , %s , %s>" %(self.\_\_class\_\_.\_\_name\_\_, self.name, self.column\_type)

# 定义数据库中五个存储类型

class StringField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False, default=None, ddl='varchar(100)'):

super().\_\_init\_\_(name,ddl,primary\_key,default)

# 布尔类型不可以作为主键

class BooleanField(Field):

def \_\_init\_\_(self, name=None, default=False):

super().\_\_init\_\_(name,'Boolean',False, default)

# 不知道这个column type是否可以自己定义 先自己定义看一下

class IntegerField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False, default=0):

super().\_\_init\_\_(name, 'int', primary\_key, default)

class FloatField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False,default=0.0):

super().\_\_init\_\_(name, 'float', primary\_key, default)

class TextField(Field):

def \_\_init\_\_(self, name=None, default=None):

super().\_\_init\_\_(name,'text',False, default)

# class Model(dict,metaclass=ModelMetaclass):

# -\*-定义Model的元类

# 所有的元类都继承自type

# ModelMetaclass元类定义了所有Model基类(继承ModelMetaclass)的子类实现的操作

# -\*-ModelMetaclass的工作主要是为一个数据库表映射成一个封装的类做准备：

# \*\*\*读取具体子类(user)的映射信息

# 创造类的时候，排除对Model类的修改

# 在当前类中查找所有的类属性(attrs)，如果找到Field属性，就将其保存到\_\_mappings\_\_的dict中，同时从类属性中删除Field(防止实例属性遮住类的同名属性)

# 将数据库表名保存到\_\_table\_\_中

# 完成这些工作就可以在Model中定义各种数据库的操作方法

# metaclass是类的模板，所以必须从`type`类型派生：

class ModelMetaclass(type):

# \_\_new\_\_控制\_\_init\_\_的执行，所以在其执行之前

# cls:代表要\_\_init\_\_的类，此参数在实例化时由Python解释器自动提供(例如下文的User和Model)

# bases：代表继承父类的集合

# attrs：类的方法集合

def \_\_new\_\_(cls, name, bases, attrs):

# 排除model 是因为要排除对model类的修改

if name=='Model':

return type.\_\_new\_\_(cls, name, bases, attrs)

# 获取table名称 为啥获取table名称 至于在哪里我也是不明白握草

table\_name=attrs.get('\_\_table\_\_', None) or name #r如果存在表名，则返回表名，否则返回 name

logging.info('found table: %s (table: %s) ' %(name,table\_name ))

# 获取Field所有主键名和Field

mappings=dict()

fields=[] #field保存的是除主键外的属性名

primaryKey=None

# 这个k是表示字段名

for k, v in attrs.items():

if isinstance(v, Field):

logging.info('Found mapping %s===>%s' %(k, v))

# 注意mapping的用法

mappings[k] = v

if v.primary\_key:

logging.info('fond primary key %s'%k)

# 这里很有意思 当第一次主键存在primaryKey被赋值 后来如果再出现主键的话就会引发错误

if primaryKey:

raise RuntimeError('Duplicated key for field') #一个表只能有一个主键，当再出现一个主键的时候就报错

primaryKey=k # 也就是说主键只能被设置一次

else:

fields.append(k)

if not primaryKey: #如果主键不存在也将会报错，在这个表中没有找到主键，一个表只能有一个主键，而且必须有一个主键

raise RuntimeError('Primary key not found!')

# w下面位字段从类属性中删除Field 属性

for k in mappings.keys():

attrs.pop(k)

# 保存除主键外的属性为''列表形式

# 将除主键外的其他属性变成`id`, `name`这种形式，关于反引号``的用法，可以参考点击打开链接

escaped\_fields=list(map(lambda f:'`%s`' %f, fields))

# 保存属性和列的映射关系

attrs['\_\_mappings\_\_']=mappings

# 保存表名

attrs['\_\_table\_\_']=table\_name #这里的tablename并没有转换成反引号的形式

# 保存主键名称

attrs['\_\_primary\_key\_\_']=primaryKey

# 保存主键外的属性名

attrs['\_\_fields\_\_']=fields

# 构造默认的增删改查 语句

attrs['\_\_select\_\_']='select `%s`, %s from `%s` '%(primaryKey,', '.join(escaped\_fields), table\_name)

attrs['\_\_insert\_\_'] = 'insert into `%s` (%s, `%s`) values (%s) ' %(table\_name, ', '.join(escaped\_fields), primaryKey, create\_args\_string(len(escaped\_fields)+1))

attrs['\_\_update\_\_']='update `%s` set %s where `%s` = ?' % (table\_name, ', '.join(map(lambda f:'`%s`=?' % (mappings.get(f).name or f), fields)), primaryKey)

attrs['\_\_delete\_\_']='delete from `%s` where `%s`=?' %(table\_name, primaryKey)

return type.\_\_new\_\_(cls, name, bases, attrs)

# 定义ORM所有映射的基类：Model

# Model类的任意子类可以映射一个数据库表

# Model类可以看作是对所有数据库表操作的基本定义的映射

# 基于字典查询形式

# Model从dict继承，拥有字典的所有功能，同时实现特殊方法\_\_getattr\_\_和\_\_setattr\_\_，能够实现属性操作

# 实现数据库操作的所有方法，定义为class方法，所有继承自Model都具有数据库操作方法

class Model(dict,metaclass=ModelMetaclass):

def \_\_init\_\_(self, \*\*kw): #感觉这里去掉\_\_init\_\_的声明也是代码结果是没影响的

super(Model,self).\_\_init\_\_(\*\*kw)

def \_\_getattr\_\_(self, key):

try:

return self[key]

except KeyError:

raise AttributeError("'Model' object have no attribution: %s"% key)

def \_\_setattr\_\_(self, key, value):

self[key] =value

def getValue(self, key):

# 这个是默认内置函数实现的

return getattr(self, key, None)

def getValueOrDefault(self, key):

value=getattr(self, key , None)

if value is None:

field = self.\_\_mappings\_\_[key]

if field.default is not None:

value = field.default() if callable(field.default) else field.default

logging.info('using default value for %s : %s ' % (key, str(value)))

setattr(self, key, value)

return value

@classmethod

# 类方法有类变量cls传入，从而可以用cls做一些相关的处理。并且有子类继承时，调用该类方法时，传入的类变量cls是子类，而非父类。

@asyncio.coroutine

def find\_all(cls, where=None, args=None, \*\*kw):

sql = [cls.\_\_select\_\_]

if where:

sql.append('where')

sql.append(where)

if args is None:

args = []

orderBy = kw.get('orderBy', None)

if orderBy:

sql.append('order by')

sql.append(orderBy)

# dict 提供get方法 指定放不存在时候返回后学的东西 比如a.get('Fuck',None)

limit = kw.get('limit', None)

if limit is not None:

sql.append('limit')

if isinstance(limit, int):

sql.append('?')

args.append(limit)

elif isinstance(limit, tuple) and len(limit) ==2:

sql.append('?,?')

args.extend(limit)

else:

raise ValueError('Invalid limit value : %s ' % str(limit))

rs = yield from select(' '.join(sql),args) #返回的rs是一个元素是tuple的list

return [cls(\*\*r) for r in rs] # \*\*r 是关键字参数，构成了一个cls类的列表，其实就是每一条记录对应的类实例

@classmethod

@asyncio.coroutine

def findNumber(cls, selectField, where=None, args=None):

'''find number by select and where.'''

sql = ['select %s \_\_num\_\_ from `%s`' %(selectField, cls.\_\_table\_\_)]

if where:

sql.append('where')

sql.append(where)

rs = yield from select(' '.join(sql), args, 1)

if len(rs) == 0:

return None

return rs[0]['\_\_num\_\_']

@classmethod

@asyncio.coroutine

def find(cls, primarykey):

'''find object by primary key'''

#rs是一个list，里面是一个dict

rs = yield from select('%s where `%s`=?' %(cls.\_\_select\_\_, cls.\_\_primary\_key\_\_), [primarykey], 1)

if len(rs) == 0:

return None

return cls(\*\*rs[0]) #返回一条记录，以dict的形式返回，因为cls的夫类继承了dict类

@classmethod

@asyncio.coroutine

def findAll(cls, \*\*kw):

rs = []

if len(kw) == 0:

rs = yield from select(cls.\_\_select\_\_, None)

else:

args=[]

values=[]

for k, v in kw.items():

args.append('%s=?' % k )

values.append(v)

print('%s where %s ' % (cls.\_\_select\_\_, ' and '.join(args)), values)

rs = yield from select('%s where %s ' % (cls.\_\_select\_\_, ' and '.join(args)), values)

return rs

@asyncio.coroutine

def save(self):

args = list(map(self.getValueOrDefault, self.\_\_fields\_\_))

print('save:%s' % args)

args.append(self.getValueOrDefault(self.\_\_primary\_key\_\_))

rows = yield from execute(self.\_\_insert\_\_, args)

if rows != 1:

print(self.\_\_insert\_\_)

logging.warning('failed to insert record: affected rows: %s' %rows)

@asyncio.coroutine

# 显示方言错误是什么鬼。。。

def update(self): #修改数据库中已经存入的数据

args = list(map(self.getValue, self.\_\_fields\_\_)) #获得的value是User2实例的属性值，也就是传入的name，email，password值

args.append(self.getValue(self.\_\_primary\_key\_\_))

rows = yield from execute(self.\_\_update\_\_, args)

if rows != 1:

logging.warning('failed to update record: affected rows: %s'%rows)

@asyncio.coroutine

def delete(self):

args = [self.getValue(self.\_\_primary\_key\_\_)]

rows = yield from execute(self.\_\_delete\_\_, args)

if rows != 1:

logging.warning('failed to delete by primary key: affected rows: %s' %rows)

if \_\_name\_\_=="\_\_main\_\_":#一个类自带前后都有双下划线的方法，在子类继承该类的时候，这些方法会自动调用，比如\_\_init\_\_

class User2(Model): #虽然User类乍看没有参数传入，但实际上，User类继承Model类，Model类又继承dict类，所以User类的实例可以传入关键字参数

id = IntegerField('id',primary\_key=True) #主键为id， tablename为User，即类名

name = StringField('name')

email = StringField('email')

password = StringField('password')

#创建异步事件的句柄

loop = asyncio.get\_event\_loop()

#创建实例

@asyncio.coroutine

def test():

yield from create\_pool(loop=loop, host='localhost', port=3306, user='root', password='Limin123?', db='test')

#user = User2(id=2, name='Tom', email='slysly759@gmail.com', password='12345')

r = yield from User2.findAll()

print(r)

#yield from user.save()

#ield from user.update()

#yield from user.delete()

# r = yield from User2.find(8)

# print(r)

# r = yield from User2.findAll()

# print(1, r)

# r = yield from User2.findAll(name='sly')

# print(2, r)

yield from destroy\_pool() #关闭pool

loop.run\_until\_complete(test())

loop.close()

if loop.is\_closed():

sys.exit(0)