Day 2 - 编写数据库模块

2294次阅读

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python-app中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

此外，在一个Web App中，有多个用户会同时访问，系统以多进程或多线程模式来处理每个用户的请求。假设以多线程为例，每个线程在访问数据库时，都必须创建仅属于自身的连接，对别的线程不可见，否则，就会造成数据库操作混乱。

所以，我们还要创建一个简单可靠的数据库访问模型，在一个线程中，能既安全又简单地操作数据库。

为什么不选择SQLAlchemy？SQLAlchemy太庞大，过度地面向对象设计导致API太复杂。

所以我们决定自己设计一个封装基本的SELECT、INSERT、UPDATE和DELETE操作的db模块：transwarp.db。

设计db接口

设计底层模块的原则是，根据上层调用者设计简单易用的API接口，然后，实现模块内部代码。

假设transwarp.db模块已经编写完毕，我们希望以这样的方式来调用它：

首先，初始化数据库连接信息，通过create_engine()函数：

```
from transwarp import db
db.create_engine(user='root', password='password', database='test', host='127.0.0.1', port=3306)
```

然后，就可以直接操作SQL了。

如果需要做一个查询，可以直接调用select()方法，返回的是list，每一个元素是用dict表示的对应的行：

```
users = db.select('select * from user')
# users =>
# [
#     { "id": 1, "name": "Michael"},
#     { "id": 2, "name": "Bob"},
#     { "id": 3, "name": "Adam"}
# ]
```

如果要执行INSERT、UPDATE或DELETE操作，执行update()方法，返回受影响的行数：

```
n = db.update('insert into user(id, name) values(?, ?)', 4, 'Jack')
```

update()函数签名为：

```
update(sql, *args)
```

统一用?作为占位符，并传入可变参数来绑定，从根本上避免SQL注入攻击。

每个select()或update()调用，都隐含地自动打开并关闭了数据库连接，这样，上层调用者就完全不必关心数据库底层连接。

但是，如果要在一个数据库连接里执行多个SQL语句怎么办？我们用一个with语句实现：

```
with db.connection():
    db.select('...')
    db.update('...')
    db.update('...')
```

如果要在一个数据库事务中执行多个SQL语句怎么办？我们还是用一个with语句实现：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

实现db模块

由于模块是全局对象，模块变量是全局唯一变量，所以，有两个重要的模块变量：

```
# db.py

# 数据库引擎对象:
class _Engine(object):
    def __init__(self, connect):
        self._connect = connect
    def connect(self):
        return self._connect()

engine = None

# 持有数据库连接的上下文对象:
class _DbCtx(threading.local):
    def __init__(self):
        self.connection = None
        self.transactions = 0

    def is_init(self):
        return not self.connection is None

    def init(self):
        self.connection = _LasyConnection()
        self.transactions = 0

    def cleanup(self):
        self.connection.cleanup()
        self.connection = None

    def cursor(self):
        return self.connection.cursor()

_db_ctx = _DbCtx()
```

由于_db_ctx是threadlocal对象，所以，它持有的数据库连接对于每个线程看到的都是不一样的。任何一个线程都无法访问到其他线程持有的数据库连接。

有了这两个全局变量，我们继续实现数据库连接的上下文，目的是自动获取和释放连接：

```
class _ConnectionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_cleanup = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_cleanup = True
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        if self.should_cleanup:
            _db_ctx.cleanup()

def connection():
    return _ConnectionCtx()
```

定义了__enter__()和__exit__()的对象可以用于with语句，确保任何情况下__exit__()方法可以被调用。

把_ConnectionCtx的作用域作用到一个函数调用上，可以这么写：

```
with connection():
    do_some_db_operation()
```

但是更简单的写法是写个@decorator：

```
@with_connection
def do_some_db_operation():
    pass
```

这样，我们实现select()、update()方法就更简单了：

```
@with_connection
def select(sql, *args):
    pass

@with_connection
def update(sql, *args):
    pass
```

注意到Connection对象是存储在_DbCtx这个threadlocal对象里的，因此，嵌套使用with connection()也没有问题。_DbCtx永远检测当前是否已存在Connection，如果存在，直接使用，如果不存在，则打开一个新的Connection。

对于transaction也是类似的，with transaction()定义了一个数据库事务：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

函数作用域的事务也有一个简化的@decorator：

```
@with_transaction
def do_in_transaction():
    pass
```

事务也可以嵌套，内层事务会自动合并到外层事务中，这种事务模型足够满足99%的需求。

事务嵌套比Connection嵌套复杂一点，因为事务嵌套需要计数，每遇到一层嵌套就+1，离开一

层嵌套就-1，最后到0时提交事务：

```
class _TransactionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_close_conn = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_close_conn = True
        _db_ctx.transactions = _db_ctx.transactions + 1
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        _db_ctx.transactions = _db_ctx.transactions - 1
        try:
            if _db_ctx.transactions==0:
                if exctype is None:
                    self.commit()
                else:
                    self.rollback()
        finally:
            if self.should_close_conn:
                _db_ctx.cleanup()

    def commit(self):
        global _db_ctx
        try:
            _db_ctx.connection.commit()
        except:
            _db_ctx.connection.rollback()
            raise

    def rollback(self):
        global _db_ctx
        _db_ctx.connection.rollback()
```

最后，把select()和update()方法实现了，db模块就完成了。
