2012
12
3
11:02 PM
16347次查看

SQLAlchemy 使用经验

分类：Python 标签：Python

上篇文章提到了，最近在用 Python 做一个网站。除了 Tornado ，主要还用到了 SQLAlchemy。这篇就是介绍我在使用 SQLAlchemy 的过程中，学到的一些知识。

首先说下，由于最新的 0.8 版还是开发版本，因此我使用的是 0.79 版，API 也许会有些不同。
因为我是搭配 MySQL InnoDB 使用，所以使用其他数据库的也不能完全照搬本文。

接着就从安装开始介绍吧，以 Debian/Ubuntu 为例（请确保有管理员权限）：

1. MySQL
   apt-get install mysql-server
   apt-get install mysql-client
   apt-get install libmysqlclient15-dev
2. python-mysqldb
   apt-get install python-mysqldb
3. easy_install
   wget http://peak.telecommunity.com/dist/ez_setup.py
   python ez_setup.py
4. MySQL-Python
   easy_install MySQL-Python
5. SQLAlchemy
   easy_install SQLAlchemy

如果是用其他操作系统，遇到问题就 Google 一下吧。我是在 Mac OS X 上开发的，途中也遇到些问题，不过当时没记下来……
值得一提的是我用了 MySQL-Python 来连 MySQL，因为不支持异步调用，所以和 Tornado 不是很搭。不过性能其实很好，因此以后再去研究下其他方案吧……

装好后就可以开始使用了：

```
from sqlalchemy import create_engine   ▲Top
from sqlalchemy.orm import sessionmaker


DB_CONNECT_STRING =
'mysql+mysqldb://root:123@localhost/ooxx?charset=utf8'
engine = create_engine(DB_CONNECT_STRING, echo=True)
DB_Session = sessionmaker(bind=engine)
session = DB_Session()
```

这里的 DB_CONNECT_STRING 就是连接数据库的路径。“mysql+mysqldb”指定了使用 MySQL-Python 来连接，“root”和“123”分别是用户名和密码，“localhost”是数据库的域名，“ooxx”是使用的数据库名（可省略），“charset”指定了连接时使用的字符集（可省略）。
create_engine() 会返回一个数据库引擎，echo 参数为 True 时，会显示每条执行的 SQL 语句，生产环境下可关闭。
sessionmaker() 会生成一个数据库会话类。这个类的实例可以当成一个数据库连接，它同时还记录了一些查询的数据，并决定什么时候执行 SQL 语句。由于 SQLAlchemy 自己维护了一个数据库连接池（默认 5 个连接），因此初始化一个会话的开销并不大。对 Tornado 而言，可以在 BaseHandler 的 initialize() 里初始化：

```
class BaseHandler(tornado.web.RequestHandler):
    def initialize(self):
        self.session = models.DB_Session()

    def on_finish(self):
        self.session.close()
```

对其他 Web 服务器来说，可以使用 sqlalchemy.orm.scoped_session，它能保证每个线程获得的 session 对象都是唯一的。不过 Tornado 本身就是单线程的，如果使用了异步方式，就可能会出现问题，因此我并没使用它。

拿到 session 后，就可以执行 SQL 了：

```
session.execute('create database abc')
print session.execute('show databases').fetchall()
session.execute('use abc')
# 建 user 表的过程略
print session.execute('select * from user where id =
1').first()
print session.execute('select * from user where id = :id',
{'id': 1}).first()
```

不过这和直接使用 MySQL-Python 没啥区别，所以就不介绍了；我还是喜欢 ORM 的方式，这也是我采用 SQLAlchemy 的唯一原因。

于是来定义一个表：

```
from sqlalchemy import Column
from sqlalchemy.types import CHAR, Integer, String
from sqlalchemy.ext.declarative import declarative_base


BaseModel = declarative_base()

def init_db():
    BaseModel.metadata.create_all(engine)

def drop_db():
    BaseModel.metadata.drop_all(engine)


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(CHAR(30)) # or Column(String(30))

init_db()
```

declarative_base() 创建了一个 BaseModel 类，这个类的子类可以自动与一个表关联。
以 User 类为例，它的 __tablename__ 属性就是数据库中该表的名称，它有 id 和 name 这两个字段，分别为整型和 30 个定长字符。Column 还有一些其他的参数，我就不解释了。
最后，BaseModel.metadata.create_all(engine) 会找到 BaseModel 的所有子类，并在数据库中建立这些表；drop_all() 则是删除这些表。

接着就开始使用这个表吧：

```
from sqlalchemy import func, or_, not_


user = User(name='a')
session.add(user)
user = User(name='b')
session.add(user)
user = User(name='a')
session.add(user)
user = User()
```


Discuz!
Google
- Gmail
- Google Adsense
- Google Analytics
- Google App Engine
- Google Apps
- Google Calendar
- Google Chrome
- Google Cloud SQL
- Google Reader
- Google Storage
- Google Talk

WordPress
日记
游戏王
知乎日爆
编程
- ABAP
- C++
- CSS
- Flash
- HTML
- JavaScript
- Mac开发
- Objective-C
- PHP
- Python
- Ruby
- Shell
- Web
- iOS开发
- 吉里吉里
- 数据库
- 设计模式

网站建设
- SEO

腾讯
- QQ
- QQ中转站
- QQ空间

资源
随笔

标签

ABAP ACG AIR AJAX Apple C C++ CLANNAD CSS Diablo2 Diablo3 Discuz! Fate/Hollow Ataraxia Fate/Stay Night Firefox Flash GFW GalGame Gmail Google

```
session.add(user)
session.commit()

query = session.query(User)
print query # 显示SQL 语句
print query.statement # 同上
for user in query: # 遍历时查询
    print user.name
print query.all() # 返回的是一个类似列表的对象
print query.first().name # 记录不存在时，first() 会返回 None
# print query.one().name # 不存在，或有多行记录时会抛出异常
print query.filter(User.id == 2).first().name
print query.get(2).name # 以主键获取，等效于上句
print query.filter('id = 2').first().name # 支持字符串

query2 = session.query(User.name)
print query2.all() # 每行是个元组
print query2.limit(1).all() # 最多返回 1 条记录
print query2.offset(1).all() # 从第 2 条记录开始返回
print query2.order_by(User.name).all()
print query2.order_by('name').all()
print query2.order_by(User.name.desc()).all()
print query2.order_by('name desc').all()
print session.query(User.id).order_by(User.name.desc(),
User.id).all()

print query2.filter(User.id == 1).scalar() # 如果有记录，返回
第一条记录的第一个元素
print session.query('id').select_from(User).filter('id =
1').scalar()
print query2.filter(User.id > 1, User.name !=
'a').scalar() # and
query3 = query2.filter(User.id > 1) # 多次拼接的 filter 也是
and
query3 = query3.filter(User.name != 'a')
print query3.scalar()
print query2.filter(or_(User.id == 1, User.id == 2)).all()
# or
print query2.filter(User.id.in_((1, 2))).all() # in

query4 = session.query(User.id)
print query4.filter(User.name == None).scalar()
print query4.filter('name is null').scalar()
print query4.filter(not_(User.name == None)).all() # not
print query4.filter(User.name != None).all()

print query4.count()
print
session.query(func.count('*')).select_from(User).scalar()
print
session.query(func.count('1')).select_from(User).scalar()
print session.query(func.count(User.id)).scalar()
print session.query(func.count('*')).filter(User.id >
0).scalar() # filter() 中包含 User，因此不需要指定表
print session.query(func.count('*')).filter(User.name ==
'a').limit(1).scalar() == 1 # 可以用 limit() 限制 count() 的
返回数
print session.query(func.sum(User.id)).scalar()
print session.query(func.now()).scalar() # func 后可以跟任意
函数名，只要该数据库支持
print session.query(func.current_timestamp()).scalar()
print session.query(func.md5(User.name)).filter(User.id ==
1).scalar()

query.filter(User.id == 1).update({User.name: 'c'})
user = query.get(1)
print user.name

user.name = 'd'
session.flush() # 写数据库，但并不提交
print query.get(1).name
```

```
session.delete(user)
session.flush()
print query.get(1)

session.rollback()
print query.get(1).name
query.filter(User.id == 1).delete()
session.commit()
print query.get(1)
```

增删改查都涉及到了，自己看看输出的 SQL 语句就知道了，于是基础知识就介绍到此了。

下面开始介绍一些进阶的知识。

如何批量插入大批数据？
可以使用非 ORM 的方式：

```
session.execute(
    User.__table__.insert(),
    [{'name': `randint(1, 100)`,'age': randint(1, 100)}
for i in xrange(10000)]
)
session.commit()
```

上面我批量插入了 10000 条记录，半秒内就执行完了；而 ORM 方式会花掉很长时间。

如何让执行的 SQL 语句增加前缀？
使用 query 对象的 prefix_with() 方法：

```
session.query(User.name).prefix_with('HIGH_PRIORITY').all(
)
session.execute(User.__table__.insert().prefix_with('IGNOR
E'), {'id': 1, 'name': '1'})
```

如何替换一个已有主键的记录？
使用 session.merge() 方法替代 session.add()，其实就是 SELECT + UPDATE：

```
user = User(id=1, name='ooxx')
session.merge(user)
session.commit()
```

或者使用 MySQL 的 INSERT ... ON DUPLICATE KEY UPDATE，需要用到 @compiles 装饰器，有点难懂，自己看吧：《SQLAlchemy ON DUPLICATE KEY UPDATE》 和 sqlalchemy_mysql_ext。

如何使用无符号整数？
可以使用 MySQL 的方言：

```
from sqlalchemy.dialects.mysql import INTEGER

id = Column(INTEGER(unsigned=True), primary_key=True)
```

模型的属性名需要和表的字段名不一样怎么办？
开发时遇到过一个奇怪的需求，有个其他系统的表里包含了一个“from”字段，这在 Python 里是关键字，于是只能这样处理了：

```
from_ = Column('from', CHAR(10))
```

如何获取字段的长度？
Column 会生成一个很复杂的对象，想获取长度比较麻烦，这里以 User.name 为例：

```
User.name.property.columns[0].type.length
```

如何指定使用 InnoDB，以及使用 UTF-8 编码？
最简单的方式就是修改数据库的默认配置。如果非要在代码里指定的话，可以这样：

```
class User(BaseModel):
    __table_args__ = {
        'mysql_engine': 'InnoDB',
        'mysql_charset': 'utf8'
    }
```

MySQL 5.5 开始支持存储 4 字节的 UTF-8 编码的字符了，iOS 里自带的 emoji（如 🐶 字符）就属于这种。
如果是对表来设置的话，可以把上面代码中的 utf8 改成 utf8mb4，DB_CONNECT_STRING 里的 charset 也这样更改。
如果对库或字段来设置，则还是自己写 SQL 语句比较方便，具体细节可参考《How to support full Unicode in MySQL databases》。
不建议全用 utf8mb4 代替 utf8，因为前者更慢，索引会占用更多空间。

如何设置外键约束？

```
from random import randint
from sqlalchemy import ForeignKey


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    age = Column(Integer)


class Friendship(BaseModel):
    __tablename__ = 'friendship'

    id = Column(Integer, primary_key=True)
    user_id1 = Column(Integer, ForeignKey('user.id'))
    user_id2 = Column(Integer, ForeignKey('user.id'))


for i in xrange(100):
    session.add(User(age=randint(1, 100)))
session.flush() # 或 session.commit()，执行完后，user 对象的 id 属性才可以访问（因为 id 是自增的）

for i in xrange(100):
    session.add(Friendship(user_id1=randint(1, 100), user_id2=randint(1, 100)))
session.commit()

session.query(User).filter(User.age < 50).delete()
```

执行这段代码时，你应该会遇到一个错误：

```
sqlalchemy.exc.IntegrityError: (IntegrityError) (1451, 'Cannot delete or update a parent row: a foreign key constraint fails (`ooxx`.`friendship`, CONSTRAINT `friendship_ibfk_1` FOREIGN KEY (`user_id1`) REFERENCES `user` (`id`))') 'DELETE FROM user WHERE user.age < %s' (50,)
```

原因是删除 user 表的数据，可能会导致 friendship 的外键不指向一个真实存在的记录。在默认情况下，MySQL 会拒绝这种操作，也就是 RESTRICT。InnoDB 还允许指定 ON DELETE 为 CASCADE 和 SET NULL，前者会删除 friendship 中无效的记录，后者会将这些记录的外键设为 NULL。
除了删除，还有可能更改主键，这也会导致 friendship 的外键失效。于是相应的就有 ON UPDATE 了。其中 CASCADE 变成了更新相应的外键，而不是删除。
而在 SQLAlchemy 中是这样处理的：

```
class Friendship(BaseModel):
    __tablename__ = 'friendship'
```

```
    id = Column(Integer, primary_key=True)
    user_id1 = Column(Integer, ForeignKey('user.id',
ondelete='CASCADE', onupdate='CASCADE'))
    user_id2 = Column(Integer, ForeignKey('user.id',
ondelete='CASCADE', onupdate='CASCADE'))
```

如何连接表？

```
from sqlalchemy import distinct
from sqlalchemy.orm import aliased


Friend = aliased(User, name='Friend')

print session.query(User.id).join(Friendship, User.id ==
Friendship.user_id1).all() # 所有有朋友的用户
print session.query(distinct(User.id)).join(Friendship,
User.id == Friendship.user_id1).all() # 所有有朋友的用户（去掉
重复的）
print session.query(User.id).join(Friendship, User.id ==
Friendship.user_id1).distinct().all() # 同上
print session.query(Friendship.user_id2).join(User,
User.id ==
Friendship.user_id1).order_by(Friendship.user_id2).distinc
t().all() # 所有被别人当成朋友的用户
print
session.query(Friendship.user_id2).select_from(User).join(
Friendship, User.id ==
Friendship.user_id1).order_by(Friendship.user_id2).distinc
t().all() # 同上，join 的方向相反，但因为不是 STRAIGHT_JOIN，所
以 MySQL 可以自己选择顺序
print session.query(User.id,
Friendship.user_id2).join(Friendship, User.id ==
Friendship.user_id1).all() # 用户及其朋友
print session.query(User.id,
Friendship.user_id2).join(Friendship, User.id ==
Friendship.user_id1).filter(User.id < 10).all() # id 小于
10 的用户及其朋友
print session.query(User.id, Friend.id).join(Friendship,
User.id == Friendship.user_id1).join(Friend, Friend.id ==
Friendship.user_id2).all() # 两次 join，由于使用到相同的表，因
此需要别名
print session.query(User.id,
Friendship.user_id2).outerjoin(Friendship, User.id ==
Friendship.user_id1).all() # 用户及其朋友（无朋友则为 None，使
用左连接）
```

这里我没提到 relationship，虽然它看上去很方便，但需要学习的内容实在太多，还要考虑很多性能上的问题，所以干脆自己 join 吧。

为什么无法删除 in 操作查询出来的记录？

```
session.query(User).filter(User.id.in_((1, 2,
3))).delete()
```

抛出这样的异常：

```
sqlalchemy.exc.InvalidRequestError: Could not evaluate current criteria in
Python.  Specify 'fetch' or False for the synchronize_session parameter.
```

但这样是没问题的：

```
session.query(User).filter(or_(User.id == 1, User.id == 2,
User.id == 3)).delete()
```

搜了下找到《Sqlalchemy delete subquery》这个问题，提到了 delete 的一个注意点：删除记录时，默认会尝试删除 session 中符合条件的对象，而 in 操作估计还不支持，于是就出错了。解决办法就是删除时不进行同步，然后再让 session 里的所有实体都过期：

```
session.query(User).filter(User.id.in_((1, 2,
```

```
3))).delete(synchronize_session=False)
session.commit() # or session.expire_all()
```

此外，update 操作也有同样的参数，如果后面立刻提交了，那么加上 synchronize_session=False 参数会更快。

如何扩充模型的基类？

declarative_base() 会生成一个 class 对象，这个对象的子类一般都和一张表对应。如果想增加这个基类的方法或属性，让子类都能使用，可以有三种方法：

1. 定义一个新类，将它的方法设置为基类的方法：

```
class ModelMixin(object):
    @classmethod
    def get_by_id(cls, session, id, columns=None,
lock_mode=None):
        if hasattr(cls, 'id'):
            scalar = False
            if columns:
                if isinstance(columns, (tuple, list)):
                    query = session.query(*columns)
                else:
                    scalar = True
                    query = session.query(columns)
            else:
                query = session.query(cls)
            if lock_mode:
                query = query.with_lockmode(lock_mode)
            query = query.filter(cls.id == id)
            if scalar:
                return query.scalar()
            return query.first()
        return None
    BaseModel.get_by_id = get_by_id

    @classmethod
    def get_all(cls, session, columns=None,
offset=None, limit=None, order_by=None,
lock_mode=None):
        if columns:
            if isinstance(columns, (tuple, list)):
                query = session.query(*columns)
            else:
                query = session.query(columns)
                if isinstance(columns, str):
                    query = query.select_from(cls)
        else:
            query = session.query(cls)
        if order_by is not None:
            if isinstance(order_by, (tuple, list)):
                query = query.order_by(*order_by)
            else:
                query = query.order_by(order_by)
        if offset:
            query = query.offset(offset)
        if limit:
            query = query.limit(limit)
        if lock_mode:
            query = query.with_lockmode(lock_mode)
        return query.all()
    BaseModel.get_all = get_all

    @classmethod
    def count_all(cls, session, lock_mode=None):
        query =
session.query(func.count('*')).select_from(cls)
        if lock_mode:
            query = query.with_lockmode(lock_mode)
        return query.scalar()
    BaseModel.count_all = count_all
```

```
    @classmethod
    def exist(cls, session, id, lock_mode=None):
        if hasattr(cls, 'id'):
            query =
session.query(func.count('*')).select_from(cls).filter(
cls.id == id)
            if lock_mode:
                query = query.with_lockmode(lock_mode)
            return query.scalar() > 0
        return False
    BaseModel.exist = exist

    @classmethod
    def set_attr(cls, session, id, attr, value):
        if hasattr(cls, 'id'):
            session.query(cls).filter(cls.id ==
id).update({
                attr: value
            })
            session.commit()
    BaseModel.set_attr = set_attr

    @classmethod
    def set_attrs(cls, session, id, attrs):
        if hasattr(cls, 'id'):
            session.query(cls).filter(cls.id ==
id).update(attrs)
            session.commit()
    BaseModel.set_attrs = set_attrs
```

虽然很拙劣，但确实能用。顺便还附送了一些有用的玩意，你懂的。

2. 设置 declarative_base() 的 cls 参数：

```
BaseModel = declarative_base(cls=ModelMixin)
```

这种方法不需要执行"BaseModel.get_by_id = get_by_id"之类的代码。不足之处就是 PyCharm 仍然无法找到这些方法的位置。

3. 设置 __abstract__ 属性：

```
class BaseModel(BaseModel):
    __abstract__ = True
    __table_args__ = { # 可以省掉子类的 __table_args__ 了
        'mysql_engine': 'InnoDB',
        'mysql_charset': 'utf8'
    }
    # ...
```

这种方法最简单，也可以继承出多个类。

如何正确使用事务？

假设有一个简单的银行系统，一共两名用户：

```
class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    money = Column(DECIMAL(10, 2))

class TanseferLog(BaseModel):
    __tablename__ = 'tansefer_log'

    id = Column(Integer, primary_key=True)
    from_user = Column(Integer, ForeignKey('user.id',
ondelete='CASCADE', onupdate='CASCADE'))
    to_user = Column(Integer, ForeignKey('user.id',
ondelete='CASCADE', onupdate='CASCADE'))
    amount = Column(DECIMAL(10, 2))

user = User(money=100)
session.add(user)
```

```
user = User(money=0)
session.add(user)
session.commit()
```

然后开两个 session，同时进行两次转账操作：

```
session1 = DB_Session()
session2 = DB_Session()

user1 = session1.query(User).get(1)
user2 = session1.query(User).get(2)
if user1.money >= 100:
    user1.money -= 100
    user2.money += 100
    session1.add(TanseferLog(from_user=1, to_user=2,
amount=100))

user1 = session2.query(User).get(1)
user2 = session2.query(User).get(2)
if user1.money >= 100:
    user1.money -= 100
    user2.money += 100
    session2.add(TanseferLog(from_user=1, to_user=2,
amount=100))

session1.commit()
session2.commit()
```

现在看看结果：

```
>>> user1.money
Decimal('0.00')
>>> user2.money
Decimal('100.00')
>>> session.query(TanseferLog).count()
2L
```

两次转账都成功了，但是只转走了一笔钱，这明显不科学。

可见 MySQL InnoDB 虽然支持事务，但并不是那么简单的，还需要手动加锁。首先来试试读锁：

```
user1 = session1.query(User).with_lockmode('read').get(1)
user2 = session1.query(User).with_lockmode('read').get(2)
if user1.money >= 100:
    user1.money -= 100
    user2.money += 100
    session1.add(TanseferLog(from_user=1, to_user=2,
amount=100))

user1 = session2.query(User).with_lockmode('read').get(1)
user2 = session2.query(User).with_lockmode('read').get(2)
if user1.money >= 100:
    user1.money -= 100
    user2.money += 100
    session2.add(TanseferLog(from_user=1, to_user=2,
amount=100))
session1.commit()
session2.commit()
```

现在在执行 session1.commit() 的时候，因为 user1 和 user2 都被 session2 加了读锁，所以会等待锁被释放。超时以后，session1.commit() 会抛出个超时的异常，如果捕捉了的话，或者 session2 在另一个进程，那么 session2.commit() 还是能正常提交的。这种情况下，有一个事务是肯定会提交失败的，所以那些更改等于白做了。

接下来看看写锁，把上段代码中的 'read' 改成 'update' 即可。这次在执行 select 的时候就会被阻塞了：

```
user1 =
session2.query(User).with_lockmode('update').get(1)
```

这样只要在超时期间内，session1 完成了提交或回滚，那么 session2 就能正常判断 user1.money >= 100 是否成立了。
由此可见，如果需要更改数据，最好加写锁。

那么什么时候用读锁呢？如果要保证事务运行期间内，被读取的数据不被修改，自己也不去修改，加读锁即可。
举例来说，假设我查询一个用户的开支记录（同时包含余额和转账记录），可以直接把 user 和 tansefer_log 做个内连接。
但如果用户的转账记录特别多，我在查询前想先验证用户的密码（假设在 user 表中），确认相符后才查询转账记录。而这两次查询的期间内，用户可能收到了一笔转账，导致他的 money 字段被修改了，但我在展示给用户时，用户的余额仍然没变，这就不正常了。
而如果我在读取 user 时加了读锁，用户是无法收到转账的（因为无法被另一个事务加写锁来修改 money 字段），这就保证了不会查出额外的 tansefer_log 记录。等我查询完两张表，释放了读锁后，转账就可以继续进行了，不过我显示的数据在当时的确是正确和一致的。

另外要注意的是，如果被查询的字段没有加索引的话，就会变成锁整张表了：

```
session1.query(User).filter(User.id >
50).with_lockmode('update').all()
session2.query(User).filter(User.id <
40).with_lockmode('update').all() # 不会被锁，因为 id 是主键

session1.rollback()
session2.rollback()

session1.query(User).filter(User.money ==
50).with_lockmode('update').all()
session2.query(User).filter(User.money ==
40).with_lockmode('update').all() # 会等待解锁，因为 money 上
没有索引
```

要避免的话，可以这样：

```
money = Column(DECIMAL(10, 2), index=True)
```

另一个注意点是子事务。
InnoDB 支持子事务（savepoint 语句），可以简化一些逻辑。
例如有的方法是用于改写数据库的，它执行时可能提交了事务，但在后续的流程中却执行失败了，却没法回滚那个方法中已经提交的事务。这时就可以把那个方法当成子事务来运行了：

```
def step1():
    # ...
    if success:
        session.commit()
        return True
    session.rollback()
    return False

def step2():
    # ...
    if success:
        session.commit()
        return True
    session.rollback()
    return False

session.begin_nested()
if step1():
    session.begin_nested()
    if step2():
        session.commit()
    else:
        session.rollback()
else:
```

```
    session.rollback()
```

此外，rollback 一个子事务，可以释放这个子事务中获得的锁，提高并发性和降低死锁概率。

如何对一个字段进行自增操作？
最简单的办法就是获取时加上写锁：

```
user = session.query(User).with_lockmode('update').get(1)
user.age += 1
session.commit()
```

如果不想多一次读的话，这样写也是可以的：

```
session.query(User).filter(User.id == 1).update({
    User.age: User.age + 1
})
session.commit()
# 其实字段之间也可以做运算:
session.query(User).filter(User.id == 1).update({
    User.age: User.age + User.id
})
```

« Tornado 使用经验

遗书 »