peewee 中文翻译文档

翻译：黑洞

Release 2.4.0

始译于：2017．06．16

# 目 录

[peewee 中文翻译文档 1](#_Toc485371415)

[目 录 3](#_Toc485371416)

[1. 产品概述 4](#_Toc485371417)

[1.1 产品名称 4](#_Toc485371418)

[1.1 产品用途 4](#_Toc485371419)

[1.2 产品构成 4](#_Toc485371420)

# 安装和测试

大多数用户更愿意采用简单的方式来安装最新版本的peewee程序， 程序托管在PyPI上：

pip install peewee

## 1.1 使用 git进行安装

本项目托管在地址 <https://github.com/coleifer/peewee>上，用户可以使用git进行安装。

git clone https://github.com/coleifer/peewee.git

cd peewee

python setup.py install

**注意**： 在某些系统上你可能需要使用：s**udo python setup.py install** 去安装peewee全系统环境

## 1.2 进行测试

你可以使用如下命令对你的安装结果进行测试：

python setup.py test

# 或者使用测试runner程序

python runtests.py

你可以使用runtests.py 脚本测试特定的功能或者特定的数据库驱动. 默认情况下测试套件的运行使用的是SQLite数据库，playhouse扩展测试不会运行。要查看可用的测试运行器选项，请使用：

python runtests.py --help

# 快速开始

## 2.1 进行测试

本文档提供了一个简短而高层次概述的Peewee主要特点描述。本指南将涵盖：

•模型定义

•存储数据

•检索数据

**注意**：如果你想要更多的干货，这里有一个更加深入的教程：创建一个“推特”风格的Web应用程序通过使用Peewee和Flask框架。

我强烈建议打开交互式shell会话运行代码，这样你才能感受到输入查询的过程。

## 2.2 模型定义

每一个模型类将直接映射到某一个数据库表，每个字段将映射到该表上的某一列，每一个模型实例对应于表中的某一行。

**from peewee import \***

db = SqliteDatabase(’people.db’)

**class** **Person**(Model):

name = CharField()

birthday = DateField()

is\_relative = BooleanField()

**class Meta:**

database = db # This model uses the "people.db" database.

Peewee中有许多字段类型适合存储各种类型的数据。Peewee将会处理python和数据库之间的值转换的问题，这样您就可以在代码中使用Python类型而不必担心了。

当我们使用外键建立模型之间的关系时，Peewee很容易做到

**class Pet(Model):**

owner = ForeignKeyField(Person, related\_name=’pets’)

name = CharField()

animal\_type = CharField()

**class** **Meta**:

database = db # this model uses the people database

现在我们有了模型，让我们创建数据库中的表来存储数据。一下命令将创建带有适当的列、索引、序列和外键约束的表：

>>> db.create\_tables([Person, Pet])

## 2.2 存储数据

让我们开始人工填充一些数据到数据库。我们将使用save()和create()方法添加更新**Person**的记录。

>>> **from** **datetime** **import** date

>>> uncle\_bob = Person(name=’Bob’, birthday=date(1960, 1, 15), is\_relative=True)

>>> uncle\_bob.save() # bob is now stored in the database

**注意**：当你调用save()方法的时候，将会返回改变后的行数。

你也可以通过调用create()方法添加一个人，它会返回一个模型实例：

>>> grandma = Person.create(name=’Grandma’, birthday=date(1935, 3, 1), is\_relative=True)

>>> herb = Person.create(name=’Herb’, birthday=date(1950, 5, 5), is\_relative=False)

更新、修改模型的实例并调用save()方法使更改生效。在这里，我们将改变Grandma的名字然后将更改后的数据保存到数据库中。

>>> grandma.name = ’Grandma L.’

>>> grandma.save() # Update grandma’s name in the database.

1

现在我们已经在数据库中存储了3个人。让我们给他们一些宠物。Grandma不喜欢房子里的动物，所以她不会有，但Herb是一个动物爱好者：

>>> bob\_kitty = Pet.create(owner=uncle\_bob, name=’Kitty’, animal\_type=’cat’)

>>> herb\_fido = Pet.create(owner=herb, name=’Fido’, animal\_type=’dog’)

>>> herb\_mittens = Pet.create(owner=herb, name=’Mittens’, animal\_type=’cat’)

>>> herb\_mittens\_jr = Pet.create(owner=herb, name=’Mittens Jr’, animal\_type=’cat’)

经过很长一段时间的生活后，Mittens生病恶心而且死了。我们需要把它从数据库中删除：

>>> herb\_mittens.delete\_instance() # he had a great life

1

**注意**：返回数是删除的行数。

Bob的叔叔决定，太多的动物已经在Herb家死了，所以他接收了Fido：

>>> herb\_fido.owner = uncle\_bob

>>> herb\_fido.save()

>>> bob\_fido = herb\_fido # rename our variable for clarity

## 2.2 检索数据

我们的数据库的真正实力是允许我们通过查询检索数据。关系数据库对于即席查询是优秀的。

### 2.2.1 查询单个记录

让我们从数据库中检索Grandma的记录。若要从数据库获得单个记录，请使用：

SelectQuery.get()方法。

>>> grandma = Person.select().where(Person.name == ’Grandma L.’).get()

我们还可以使用等效Model.get()方法：

>>> grandma = Person.get(Person.name == ’Grandma L.’)

### 2.2.1 查询多个记录

让我们列出数据库中所有的人：

>>> **for** person **in** Person.select():

... **print** person.name, person.is\_relative

...

Bob True

Grandma L. True

Herb False

让我们列出数据库中所有的猫和其主人：

>>> query = Pet.select().where(Pet.animal\_type == ’cat’)

>>> **for** pet **in** query:

... **print** pet.name, pet.owner.name

...

Kitty Bob

Mittens Jr Herb

在之前的查询中存在一个大问题：因为我们是访问pet.owner.name而且在我们原来的查询中我们没有选择这个值，peewee将要执行一个查询来检索宠物的主人。这行为被称为N + 1，通常应该避免。

我们可以通过选择宠物和人并添加连接来避免额外的查询。

>>> query = (Pet

... .select(Pet, Person)

.. .join(Person)

... .where(Pet.animal\_type == ’cat’))

>>> **for** pet **in** query:

... **print** pet.name, pet.owner.name

...

Kitty Bob

Mittens Jr Herb

让我们获取所以Bob的宠物：

>>> **for** pet **in** Pet.select().join(Person).where(Person.name == ’Bob’):

... **print** pet.name

...

Kitty

Fido

我们可以在这里做另一件很酷的事来得到Bob的宠物。既然我们已经有一个对象来代表鲍伯，我们可以这么做：

>>> **for** pet **in** Pet.select().where(Pet.owner == uncle\_bob):

... **print** pet.name

通过添加一个order\_by()条款让我们确保这些都是按字母顺序排序：

>>> **for** pet **in** Pet.select().where(Pet.owner == uncle\_bob).order\_by(Pet.name):

... **print** pet.name

...

Fido

Kitty

让我们从小到老列出所有人：

>>> **for** person **in** Person.select().order\_by(Person.birthday.desc()):

... **print** person.name

...

Bob

Herb

Grandma L.

现在让我们列出所有的人和一些关于他们宠物的信息：

>>> **for** person **in** Person.select():

... **print** person.name, person.pets.count(), ’pets’

... **for** pet **in** person.pets:

... **print** ’ ’, pet.name, pet.animal\_type

...

Bob 2 pets

Kitty cat

Fido dog

Grandma L. 0 pets

Herb 1 pets

Mittens Jr cat

我们再次遇到了一个典型的n + 1查询行为示例。我们可以通过执行JOIN来避免这种情况。

汇总记录

>>> subquery = Pet.select(fn.COUNT(Pet.id)).where(Pet.owner == Person.id).

>>> query = (Person

... .select(Person, Pet, subquery.alias(’pet\_count’))

... .join(Pet, JOIN\_LEFT\_OUTER)

... .order\_by(Person.name))

>>> for person in query.aggregate\_rows(): # Note the ‘aggregate\_rows()‘ call.

... print person.name, person.pet\_count, ’pets’

... for pet in person.pets:

... print ’ ’, pet.name, pet.animal\_type

...

Bob 2 pets

Kitty cat

Fido dog

Grandma L. 0 pets

Herb 1 pets

Mittens Jr cat

即使我们创建的查询分开，**只有一个**查询的实际执行。

最后，让我们做一个复杂的例子。让我们把所有的生日都找回来：

•1940岁之前（祖母）

•1959后（鲍勃）