链滴

Django 常用 ORM 方法和查询集 API

作者：zyk

原文链接：https://ld246.com/article/1565188629748

来源网站：链滴

前言

Django内置了数据库抽象API，通过调用指定函数，封装对象的方法来完成对数据库检索、增加、删、修改、聚合等操作，无需手写SQL，可以大大提升开发效率。

● 开始教程之前，我们已经在 **mysite/blog/models.py** 中创建了三个实体类——Blog，Author Entry。

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
```

```
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):
        return self.headline
```

一.创建和更新

在Django中一个实体类对应一张表，该类的实例表示数据库中的记录，对该类的实例化表示数据库级的操作。

1. 创建对象

假设实体类存在于 **mysite/blog/models.py** 中，现在要创建对象，先初始化

名称为b的Blog实体类，再传递参数对name和tagline实例化，通过save()操作完成创建对象。相当执行SQL中的INSERT操作。

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

2. 修改对象

修改对象与创建对象类似，但要先获取需要更新的对象，实例化指定参数，并调用save()函数。相当执行SQL中的UPDATE操作。

```
>>> b5 = Blog.objects.get(id=1)
>>> b5.name = 'New name'
>>> b5.save()
```

3. 保存ForeignKey和ManyToManyField字段

保存外键ForeignKey与保存普通对象类似，只需要将外键对象获取过来，并实例化相应字段即可。

例如更新Entry中的blog属性。（Blog是Entry的外键）

```
>>> from blog.models import Blog, Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

更新ManyToManyField（多对多关系）的方法略有不同，先获取名称为joe的Author对象，然后使用add()函数向entry中添加该记录。此示例将joe添加到entry对象中：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

要让ManyToManyField一次性添加多个记录，要在调用中包含多个add()函数，如下所示：

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

二. 查询

要从数据库中检索对象，需要在模型类上通过 Manager来构建一个 QuerySet对象。

QuerySet表示数据库中的对象集合。它可以有零个，一个或多个过滤器（filters）。过滤器根据给定参数缩小查询结果范围。在SQL术语中，QuerySet等同于SELECT语句，过滤器是例如WHERE或LIMI等条件语句。

可以通过 Manager（管理器）来获取 QuerySet。每个模型至少有一个 Manager，默认叫做objects。可以通过模型类直接访问它，如下所示：

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

管理器只能通过模型类来访问，而不能通过实例化的模型实例来访问，以在“表级”操作和“记录级操作之间实现分离。

1. 查询所有对象

可以通过all()函数来获取所有Entry对象。如下所示:

```
>>> all_entries = Entry.objects.all()
```

2. 条件查询

有时候我们需要通过一些条件对查询进行过滤，因此all()函数并不能满足我们的需求。

那么就要对 QuerySet添加条件过滤。两种最常见的条件查询方法是：

- filter(**kwargs)：**返回包含指定查询参数的QuerySet。**
- exclude(**kwargs)：**返回不包含指定查询参数的QuerySet（排除查询）。**

查找参数（**kwargs在上面的函数定义中）应采用下面的条件查询中描述的格式。

例如，要获取2006年的博客条目的QuerySet，可以使用如下的filter()函数：

```
Entry.objects.filter(pub_date__year=2006)
```

若使用默认的manager类，它和以下获取的结果相同：

```
Entry.objects.all().filter(pub_date__year=2006)
```

链式过滤器

连接多个过滤条件之后仍然是一个QuerySet对象，所以可以用filter()函数和exclude()函数将QuerySe对象拼接在一起。例如：

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime.date(2005, 1, 30)
... )
```

上述代码将获取以** ‘What’ **开头，从2005年1月30日至今天的所有Entry条目的QuerySet对象。

过滤的QuerySet都是唯一的

每当你执行一次查询，都会获得一个全新的QuerySet对象，和之前没有关系，可以独立和重复使用例如：

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

QuerySet是惰性的

QuerySet是惰性的，创建查询集并不会进行数据库层级的操作，Django会对你创建的QuerySet对象行评估，当你提交时，才会执行相应的数据库层级操作。例如：

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

只有当执行到最后一条print(q)语句时，Django才会真正地执行数据库级别的查询操作。在这之前，有的操作只是暂时被保存在缓存中。

3. 单一对象查询

使用filter()函数进行查询时，哪怕只有一个对象符合条件，它也会返回QuerySet对象，只是此时的QurySet对象中只包含一个元素。

如果你想搜索唯一确定的对象，可以调用管理器中的get()函数。如下：

```
>>> one_entry = Entry.objects.get(pk=1)
```

在get()函数中，你可以使用像filter()函数中一样的查询表达式。

但get()函数和filte()存在差异，如果没有查询到记录，get()函数会引发DoesNotExist 异常，这个异是正在执行查询的实体类的属性，例如，在上面的代码中，如果没有查询到主键为1的Entry对象，那将触发Entry.DoesNotExist异常。

如果调用了多个get()函数进行查询，结果超过了一个，Django会抛出 MultipleObjectsReturned异

，这个异常也是模型类本身的属性。

所以get()函数要慎用，使用时应注意捕捉DoesNotExist 异常。或者用fiter()查询函数来替代。

大多数情况下，用all()，get()，fiter()，exclude()函数来实现查询功能是足够了的。

4. 限制QuerySet

使用python的数组切片语法，可以将QuerySet限制为一定数量。相当于SQL中的LIMIT和OFFSET语句。

例如，这将返回前5个对象（）：LIMIT 5

```
>>> Entry.objects.all()[:5]
```

这将返回第七个到第十一个对象（）：OFFSET 6 LIMIT 5

```
>>> Entry.objects.all()[6:11]
```

注意：切片操作不支持负索引，例如Entry.objects.all()[-1]是不允许的

通常，切片QuerySet返回一个新的 QuerySet，它不会被立刻执行。如果指定切片操作中的step（步）参数，会立刻被执行。例如，以下代码会执行查询操作，并返回前十个对象中每第二个对象的列表。

```
>>> Entry.objects.all()[:10:2]
```

不要对切片操作的查询集进行进一步的过滤和排序，有可能会产生模糊性。

如果要检索单个对象，一般使用简单索引而不是切片操作，例如，对所有Entry对象按照标题排序之，返回数据库中的第一个Entry对象：等价于SQL——SELECT foo FROM bar LIMIT 1

```
>>> Entry.objects.order_by('headline')[0]
```

也相当于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

但如果没有查询到符合条件的对象，第一种写法将引发IndexError异常，第二个将引发DoesNotExist异常。

5. 字段查询

字段查询等同于SQL中的WHERE语句，在Django中是通过调用get()，filter()， exclude()函数进行查询基本格式为：field__lookuptype=value（**注意是双重下划线**）。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

等同于SQL语句:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

查询中的指定字段必须是模型中已经定义的字段之一，但有例外，对于外键ForeignKey可以指定后的名称字段_id，此时value参数应包含外部模型主键的原始值。例如：

```
>>> Entry.objects.filter(blog_id=4) # 查询Entry集合中，外键blog_id为4的所有记录
```

如果传递无效的关键字参数，将会引发TypeError异常。

以下介绍一些Django中常用的查询参数。

- exact：“精确”匹配（区分大小写）。例如：

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
# 等价于
# SELECT ... WHERE headline = 'Cat bites dog';
```

exact是默认的类型，当查询关键字参数不包含双下划线时，则查询类型默认为exact

- iexact：是不区分大小写的匹配项。例如查询：

```
>>> Blog.objects.get(name__iexact="beatles blog")
# 它将匹配 "Beatles Blog" "beatles blog" "BeAtlES blOG" 等等
```

- contains：区分大小写的模糊查询，例如：

```
Entry.objects.get(headline__contains='Lennon')
# 等价于 SQL
# SELECT ... WHERE headline LIKE '%Lennon%';
```

- icontains：不区分大小写的模糊查询，与contains相对应。
- startswith：以什么开头的模糊查询（**区分大小写**）
- istartswith：以什么开头的模糊查询（**不区分大小写**）
- endswith：以什么结尾的模糊查询（**区分大小写**）
- iendswith：以什么结尾的模糊查询（**不区分大小写**）

6. 跨关系查询

Django提供了一种比较方便的跨关系查询方式，它会在幕后帮你处理SQL的JOIN关系，只要在关联段名之后加入双下划线分割，就能查询到相应的记录。

例如：

```
# 返回所有满足条件的Entry对象--外键Blog的name属性值为'Beatles Blog'
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

反向操作也是可行的，指定反向关系只需要使用模型的小写名。例如：

```
# 返回Blog对象--它所关联的Entry对象中的headline字段包含'Lennon'
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

如果你跨越了多个关系进行查询，而中间某个模型的字段没有满足查询的条件，Django会把它当作个空对象（NULL）来处理，但仍是有效的对象。例如：

```
Blog.objects.filter(entry__authors__name='Lennon')
```

如果上述代码中的entry没有关联任何的author，它将被视作没有name，而不会因为确实author而出异常。大多数情况下，都是符合正常逻辑的。唯一可能让你产生困惑的是在使用isnull时。

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

这将会返回Blog对象，它关联的entry对象中的author的name属性为空，以及entry中的author对象空

。若你不想要后者，可以这样写。

```
Blog.objects.filter(entry__authors__isnull=False, entry__authors__name__isnull=True)
```

跨多值关联查询

有时候我们需要进行多值关联查询，例如，Entry对象 和 tags 对象是 ManyToManyField（多对多系），要从和 Entry 关联的 tags 条目中找到名为 "music" 和 "bands" 的条目，或要找到某个标签名为 music" 且状态为 "public" 的条目。

Django通过调用连续的fiter()方法或者在fiter()中传递多参数的方法来解决多值关联查询。但是有时用filter()会让人感觉困惑，以下通过举例来说明。

要查询所有满足关联条目中entry中的headline标题含有 "Lennon"，**且**发布于2008年的Blog对象（**个条件同时满足**），我们可以这样写。

```
Blog.objects.filter(entry__headline__contains='Lennon', entry__pub_date__year=2008)
```

要查询所有满足关联条目中entry中的headline标题含有 "Lennon"，**或**发布于2008年的Blog对象（**满足一个条件即可**），可以这样写。

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__pub_date__year=2008)
```

但是exclude()方法的使用与filter()不尽相同，例如：

```
Blog.objects.exclude(
    entry__headline__contains='Lennon',
    entry__pub_date__year=2008,
)
```

这并不会同时排除包含**'Lennon'**和发布日期为**2008**的记录(只排除其中的一个)，这次查询是OR关系，这和filter()恰好相反。

那么我们要查询标题不包含**'Lennon'**且发布日期不是**2008**的记录该怎么办，需要进行两次查询，下所示：

```
Blog.objects.exclude(
    entry__in=Entry.objects.filter(
        headline__contains='Lennon',
        pub_date__year=2008,
    ),
)
```

7. 使用F表达式为模型指定字段

之前我们都是将模型字段和常量作比较，但如果我们想将同一个模型中的字段和另一个字段作比较该么办。

Django提供了F表达式来实现这种比较。通过F()函数来引用模型中的字段，并在查询中使用该字段。

例如，要查询所有评论数大于 pingbacks 的 Entry 对象，构建了一个来指代 pingback 数量的 F() 对，然后在查询中调用该 F() 对象:

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django 支持对 F() 对象进行加、减、乘、除、求余和次方等数学操作，另一操作数可以是常量或者 F() 对象。例如，要查询comments两倍于 pingbacks 的 Entry对象，可以这样写:

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```

要查询所有rating低于 pingback 和 comments 总数之和的 Entry 对象，可以这样写:

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

也能在F()函数中加入双下划线进行关联属性查询，例如，要查询所有authors名称与'blog名称相同的ntry 对象，可以这么写：

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

对于 date 和 date/time 字段，你可以加上或减去一个 timedelta对象。例如，要查询发布三天后被改的Entry对象，可以这么写：

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

F() 对象可以调用 .bitand()， .bitor()， .bitrightshift() 和 .bitleftshift()方法来支持位操作。

例如：

```
>>> F('somefield').bitand(16)
```

8. 主键查询快捷方式——pk

Django 可以通过 pk 字段来进行主键查询，pk代表主键 primarykey。

对于主键是id的模型，下列三句查询是等效。

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

pk的使用不仅限于此，其他的查询选项也可以使用。例如：

```
 Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

pk也支持关联查询。以下三句是等效的：

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)       # __exact is implied
>>> Entry.objects.filter(blog__pk=3)       # __pk implies __id__exact
```

9. 在 LIKE 语句中转义百分号和下划线

Django中的iexact， contains， icontains， startswith， istartswith， endswith 和 iendswith

查询参数等效于SQL中的LIKE语句，使用这些参数时，它们会对%进行自动转义，同时Django还对下线进行了转义处理，你可以放心大胆地使用百分好和下划线进行查询了。例如：

```
>>> Entry.objects.filter(headline__contains='%')
# 等效于以下SQL语句：
# SELECT ... WHERE headline LIKE '%\%%';
```

10. 缓存和QuerySet

每个 `QuerySet都带有缓存，所以尽量减少对数据库的访问。理解缓存，有助于提高代码运行效率。

新创建的 QuerySet，其缓存是空的。一旦QuerySet被提交之后，就会执行数据库查询操作。随后，Django 就会将查询结果保存在 QuerySet 的缓存中，并返回这些显式请求的缓存。

我们需要合理利用缓存，提高程序运行效率。下列操作会执行两次数据库查询，加剧了数据库负载。两次对数据操作的间隙中，可能会有数据被添加或者删除，导致脏数据的产生。

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

为了避免此类问题，应当重复利用QuerySet。

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # Evaluate the query set.
>>> print([p.pub_date for p in queryset]) # Re-use the cache from the evaluation.
```

QuerySet何时不会被缓存

QuerySet并不会总是被缓存，当进行数组切片和索引操作时，QuerySet不会被缓存

例如，重复利用索引来调用查询集中的对象，会导致每次都查询数据库:

```
>>> queryset = Entry.objects.all()
>>> print(queryset[5]) # 查询数据库
>>> print(queryset[5]) # 再次查询数据库
```

不过，如果已对查询结果集进行检出操作（例如循环遍历操作），就会直接调用缓存中的数据:

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # 查询数据库，写入缓存
>>> print(queryset[5]) # 调用缓存
>>> print(queryset[5]) # 调用缓存
```

以下动作会触发全部查询结果集，并写入缓存。

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```