

Django ORM 理论与练习

授课人 令狐东邪

版权声明

九章的所有课程均受法律保护，不允许录像与传播录像
一经发现，将被追究法律责任和赔偿经济损失

今天需要做什么



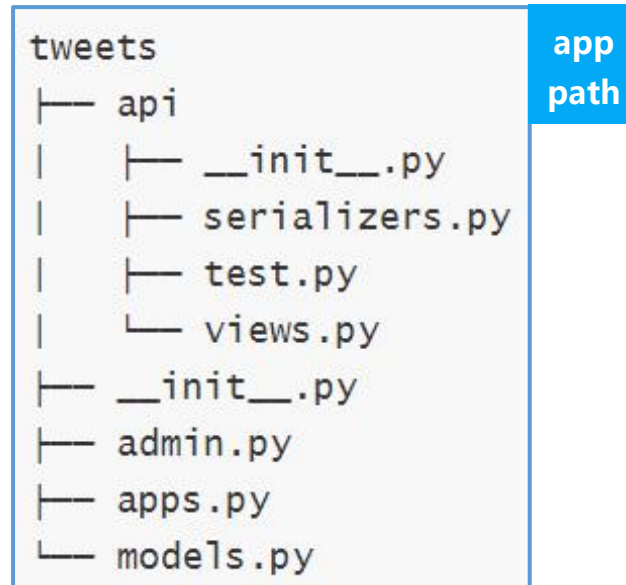
01 Model 简介

models.py 文件

右图是 tweet 应用的目录结构，里面有一个 models.py 文件，这个文件不需要用户去手动创建，当我们执行如下命令时，就会自动生成：

```
python manage.py startapp tweet
```

通常我们定义的 Model 都会单独放在这个文件中。



Model 介绍

模型的作用是帮你定义要如何存储数据，数据的类型是什么、数据之间的关系是什么等等。一般来说，每一个模型都映射一张数据库表。

- 每个模型都是一个 Python 的类，这些类继承 `django.db.models.Model`。
- 模型类的每个属性都相当于一个数据库的字段。

```
class Tweet(models.Model):  
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)  
    content = models.TextField()  
    created_at = models.DateTimeField(auto_now_add=True)  
    photos = models.ManyToManyField(Photo, blank=True)  
    ...  
    likes_count = models.BigIntegerField(default=0, null=True)  
    comments_count = models.BigIntegerField(default=0, null=True)
```

models.py

Model 介绍

数据表和Model的对应关系

tweets_tweet

🔑 id: int
content: longtext
◆ created_at: datetime(6)
◆ user_id: int
comments_count: bigint
likes_count: bigint

```
class Tweet(models.Model):  
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)  
    content = models.TextField()  
    created_at = models.DateTimeField(auto_now_add=True)  
    photos = models.ManyToManyField(Photo, blank=True)  
    ...  
    likes_count = models.BigIntegerField(default=0, null=True)  
    comments_count = models.BigIntegerField(default=0, null=True)
```

models.py

下面让我们来看看 Tweet 模型各个的组成部分。

```
class Tweet(models.Model):  
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)  
    content = models.TextField()  
    created_at = models.DateTimeField(auto_now_add=True)  
    photos = models.ManyToManyField(Photo, blank=True)  
    ...  
    likes_count = models.BigIntegerField(default=0, null=True)  
    comments_count = models.BigIntegerField(default=0, null=True)
```

models.py

表示**模型的名字**，对应实际数据库中名字是 tweets_tweet，前一个 tweets 表示应用的名字。

表示**字段的名字**，对应实际数据库中名字与代码中相同。

表示**字段类型**，具体内容我们会在后面进行介绍。

表示**位置参数**，即想要关联的模型类名，具体内容后面介绍。

表示**字段选项**，可以设置某个字段一些特定属性，用关键字参数的形式表示，具体内容后面介绍。

02 如何进行数据迁移?

什么是 Migrations

迁移 (Migrations) 是 Django 将模型的更改 (添加字段, 删除模型等) 应用到数据库中的一种方式。

下面是我们最常用到的两条命令:

```
python manage.py makemigrations
```

基于模型的修改创建迁移。

```
python manage.py migrate
```

负责应用和撤销迁移。

所以, 当我们修改一个模型之后, 例如, 想给 Tweet 模型增加一个 `unlike_count` 字段的时候, 便可以先

```
python manage.py makemigrations 再 python manage.py migrate
```

这个时候, 数据表就会增加一个 `unlike_count` 字段。

常见字段及索引介绍

03

- ▶ 字段命名规范
- ▶ 常见的 ORM 字段
- ▶ 与 ORM 字段类型对应的数据库数据类型
- ▶ 索引

字段命名规范

类型	字段名	例子
计数类型	名词复数 + _ + count	点赞数 - likes_count 评论数 - comments_count
时间类型	动词过去式 + _ + at	创建时间 - created_at 更新时间 - updated_at
日期类型	动词过去式 + _ + at + date	创建时间 - created_at_date
布尔类型	is_* (是不是) has_* (有没有) can_be_* (能不能)	是不是空的 - is_empty 有没有删除 - has_deleted 能不能删除 - can_be_deleted
外键类型	关联模型名的小写	关联了 Tweet 模型 - tweet 关联了 Photo 模型 - photo

注意：定义字段名时应小心避免使用与模型 API 冲突的名称，如 clean, save, delete 等。

自增长字段

ORM	MySQL	Size
AutoField	INTEGER AUTO_INCREMENT	4 bytes
BigAutoField	BIGINT AUTO_INCREMENT	8 bytes
SmallAutoField	SMALLINT AUTO_INCREMENT	2 bytes

自增长字段通常作为主键使用，通常情况下，我们不会显式地指定一个主键，因为 Django 会帮我们做这件事。如果你想自己指定主键，可以在你想要设置为主键的字段上添加**字段选项** `primary_key=True`，例如 `id = models.AutoField(primary_key=True)`。如果 Django 看到你显式地设置了 `Field.primary_key`，将不会自动在表（模型）中添加 `id` 列。

布尔类型字段

ORM	MySQL	Size
BooleanField	TINYINT(1)	1 bytes
NullBooleanField	TINYINT(1)	1 bytes

Django提供了两种布尔类型的字段，第一种这种不能为空，第二种的字段的值可以为空。

MySQL没有内置的布尔类型，它使用TINYINT(1)。

```
class Photo(models.Model):
    status = models.IntegerField(
        default=PhotoStatus.PENDING,
        choices=PHOTO_STATUS_CHOICES,
    )
    file = models.FileField()
    ...
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    ...
    has_deleted = models.BooleanField(default=False)
    deleted_at = models.DateTimeField(null=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

软删除(soft delete)标记，当一个照片被删除的时候，首先会被标记为已经被删除，在一定时间之后才会被真正的删除。这样做的目的是，如果在 tweet 被删除的时候马上执行真删除的通常会花费一定的时间，影响效率。可以用异步任务在后台慢慢做真删除。

```
class Photo(models.Model):
    status = models.IntegerField(
        default=PhotoStatus.PENDING,
        choices=PHOTO_STATUS_CHOICES,
    )
    file = models.FileField()
    ...
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    ...
    has_deleted = models.BooleanField(default=False)
    deleted_at = models.DateTimeField(null=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

default 选项

- 用来设置字段的默认值，它可以是一个值或者一个可调用对象。并且它不能使用可变类型，如 list、dict 等。
- 这个默认值只是在ORM层面上起作用，也就是说当你使用 ORM 的方式创建一条数据时，该数据的这个字段会设置默认值。如果使用 SQL 语句创建一条数据时，则不会设置默认值。

整型字段

ORM	MySQL	Size
IntegerField	INTEGER	4 bytes
PositiveIntegerField	INTEGER UNSIGNED	4 bytes
BigIntegerField	BIGINT	8 bytes
PositiveBigIntegerField	BIGINT UNSIGNED	8 bytes
SmallIntegerField	SMALLINT	2 bytes
PositiveSmallIntegerField	SMALLINT UNSIGNED	2 bytes

Django ORM 提供了6种不同的整型字段，可以按照两个标准来进行分类，一个是按正负数来分（**Positive**），一个是按数值大小来分（**Small、Big、Integer**）具体分类如上。

IntegerField

```
class Photo(models.Model):
    status = models.IntegerField(
        default=PhotoStatus.PENDING,
        choices=PHOTO_STATUS_CHOICES,
    )
    file = models.FileField()
    ...
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    ...
    has_deleted = models.BooleanField(default=False)
    deleted_at = models.DateTimeField(null=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

```
class PhotoStatus:
    PENDING = 0
    APPROVED = 1
    REJECTED = 2

PHOTO_STATUS_CHOICES = (
    (PhotoStatus.PENDING, 'Pending'),
    (PhotoStatus.APPROVED, 'Approved'),
    (PhotoStatus.REJECTED, 'Rejected'),
)
```

choices 选项:

choices 的值通常是一个可迭代的序列，序列的元素是一个二元组。每个元组中的第一个元素是要在模型上设置的实际值，第二个元素是人可读的名称。

BigIntegerField

```
class Tweet(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    photos = models.ManyToManyField(Photo, blank=True)
    ...
    likes_count = models.BigIntegerField(default=0, null=True)
    comments_count = models.BigIntegerField(default=0, null=True)
```

null 选项

如果是 True, Django 将在数据库中存储空值为 NULL。默认为 False。

为什么使用 BigInteger(2^{64}) ?

了解一下鸟叔如何搞挂 youtube 的。

<https://www.youtube.com/watch?v=9bZkp7q19f0>

PositiveIntegerField

```
class Like(models.Model):
    # https://docs.djangoproject.com/en/3.1/ref/contrib/contenttypes/#generic-relations
    object_id = models.PositiveIntegerField()
    content_type = models.ForeignKey(
        ContentType,
        on_delete=models.SET_NULL,
        null=True,
    )
    # user liked content_object at created_at
    content_object = GenericForeignKey('content_type', 'object_id')
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

字符串类型

ORM	MySQL	Size
CharField	VARCHAR	0-65535 bytes
TextField	LONGTEXT	0-4 294 967 295 bytes

CharField 是一个字符串字段, **TextField** 是一个大的文本字段。

```
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.SET_NULL, null=True)
    avatar = models.FileField(null=True)
    nickname = models.CharField(null=True, max_length=200)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

max_length 选项

字段的最大长度 (以字符为单位), 所有的 CharField 字段都要设置这个字段选项。

```
class Comment(models.Model):  
    """..."""  
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)  
    tweet = models.ForeignKey(Tweet, null=True, on_delete=models.SET_NULL)  
    content = models.TextField(max_length=140)  
    created_at = models.DateTimeField(auto_now_add=True)  
    updated_at = models.DateTimeField(auto_now=True)
```

max_length 选项

字段的最大长度 (以字符为单位),
所有的 TextField 字段都要设置
这个字段选项。

时间类型

ORM	MySQL
DateField()	DATE
DateTimeField()	DATETIME(6)
DurationField()	BIGINT

DateField 类型可用于需要日期值而不需要时间的字段，MySQL 以 'YYYY-MM-DD' 格式检索与显示 DATE 值。支持的范围是 '1000-01-01' 到 '9999-12-31'。

DateTimeField 类型可用于同时包含日期和时间的字段。MySQL 以 'YYYY-MM-DD HH:mm:ss' 格式检索与显示 DATETIME 类型。支持的范围是 '1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'。

DurationField 一个用于存储时间段的字段——在 Python 中用 timedelta 建模。

```
class Photo(models.Model):
    status = models.IntegerField(
        default=PhotoStatus.PENDING,
        choices=PHOTO_STATUS_CHOICES,
    )
    file = models.FileField()
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    has_deleted = models.BooleanField(default=False)
    deleted_at = models.DateTimeField(null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    created_at_date = models.DateField(auto_now_add=True)
```

auto_now_add 选项

当将选项设置为 True 时，第一次创建对象会自动将该字段设置为现在。对**创建时间戳**很有用。


```
class Comment(models.Model):  
    """ ... """  
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)  
    tweet = models.ForeignKey(Tweet, null=True, on_delete=models.SET_NULL)  
    content = models.TextField(max_length=140)  
    created_at = models.DateTimeField(auto_now_add=True)  
    updated_at = models.DateTimeField(auto_now=True)
```

auto_now 选项

当将选项设置为 True 时，每次保存对象会自动将该字段设置为现在。对于“**最后修改**”的时间戳很有用。

文件类型

ORM	MySQL
FileField()	varchar

例如保存用户上传的头像，图片不会被保存进数据库，而是图片保存在文件系统中，在数据表的相应字段存放图片在文件系统中的路径。Django ORM提供了 **FileField** 保存文件的路径信息。

```
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.SET_NULL, null=True)
    avatar = models.FileField(null=True)
    nickname = models.CharField(null=True, max_length=200)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{} {}'.format(self.user, self.nickname)
```

Django 还有一个 **ImageField**，但是尽量不要用，会有很多的其他问题，用 **FileField** 可以起到同样的效果。因为最后我们都是以文件形式存储起来，使用的是文件的 url 进行访问。

其他类型

ORM	MySQL	Features
FloatField()	DOUBLE PRECISION	浮点数
DecimalField()	numeric(%(max_digits)s, %(decimal_places)s)	一个固定精度的十进制数
EmailField()	varchar	电子邮箱地址
UUIDField()	char(32)	UUID
IPAddressField()	char(15)	IPv4 的地址

定义索引



当我们打开 Twitter 浏览 NewsFeed 时，在页面上会刷出来若干条 tweets，这实际上是我们服务器去数据库中查询 NewsFeed 表后返回的数据。

此时没有为 NewsFeed 表添加索引的话，就需要走全表扫描，如果表中的记录很多，速度就会很慢。

此时我们可以为 NewsFeed 添加一个**联合索引**，先依据 user_id 排序，再依据 created_at 排序。

定义索引

联合索引

```
class NewsFeed(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    tweet = models.ForeignKey(Tweet, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        index_together = (('user', 'created_at'),)
        unique_together = (('user', 'tweet'),)
        ordering = ('user', '-created_at',)
```

`index_together`表示联合索引，索引信息定义在 `class Meta` 嵌套类里面，可以使用一个二元元组表，每个子元组表示一个联合索引。如果只需要定义的联合索引只有一个，也可以写成下面的的一元元组的形式

```
index_together = ('user', 'created_at')
```

定义索引

```
class NewsFeed(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    tweet = models.ForeignKey(Tweet, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        index_together = (('user', 'created_at'),)
        unique_together = (('user', 'tweet'),)
        ordering = ('user', '-created_at',)
```

```
index_together = (('user', 'created_at'),)
```

此时的index_together将被翻译为如下的SQL语句

```
KEY `newsfeeds_newsfeed_user_id_created_at_1ae1d021_idx` (`user_id`,`created_at`),
```

定义唯一约束

```
class NewsFeed(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    tweet = models.ForeignKey(Tweet, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        index_together = (('user', 'created_at'),)
        unique_together = (('user', 'tweet'),)
        ordering = ('user', '-created_at',)
```

unique_together = (('user', 'tweet'),)

再NewsFeed表中还有一个唯一性约束

此时的unique_together将被翻译为如下的SQL语句

```
UNIQUE KEY `newsfeeds_newsfeed_user_id_tweet_id_5629d611_uniq` (`user_id`,`tweet_id`),
```


定义唯一约束

```
class NewsFeed(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    tweet = models.ForeignKey(Tweet, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        index_together = (('user', 'created_at'),)
        unique_together = (('user', 'tweet'),)
        ordering = ('user', '-created_at',)
```

`ordering = ('user', '-created_at',)`

ordering属性指定默认的排序规则

前面有一个可选的“-”字头，表示降序。没有前缀“-”的字段将按升序排列。

上面的语句作用查询操作，先根据user 升序排序，根据created_at降序排序

在数据库中，我们知道为了存储**关系型数据**，我们会把多张表单通过外键关联起来，接下来会介绍用 Django 的 ORM 怎么定义表和表之间的关系

建立表与表之间的关系

04

- ▶ 一对一关系
- ▶ 多对一关系
- ▶ 多对多关系

一对一关系

一个多对一的关系就如同一个人只能有一个配偶。

```
from django.db import models
from django.contrib.auth.models import User

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.SET_NULL, null=True)
    avatar = models.FileField(null=True)
    nickname = models.CharField(null=True, max_length=200)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{} {}'.format(self.user, self.nickname)
```

一个多对一的关系。需要两个位置参数：模型相关的类和 on_delete 选项。

on_delete 选项用于**级联删除**。Django 模拟了 SQL 约束 ON DELETE CASCADE 的行为

1、on_delete = **None**:

删除关联表的数据时，当前表与关联表的filed的行为。

2、on_delete = models.**CASCADE**:

表示级联删除，当关联表（子表）中的数据删除时，与其相对应的外键（父表）中的数据也删除。

3、on_delete = models.**DO_NOTHING**:

你删你的，父亲（外键）不想管你

4、on_delete = models.**PROTECT**:

保护模式，如采用这个方法，在删除关联数据时会抛出ProtectError错误

5、on_delete = models.**SET_DEFAULT**:





设置默认值，删除子表字段时，外键字段设置为默认值，所以定义外键的时候注意加上一个默认值。

6、on_delete = models.**SET** (值) :

删除关联数据时，自定义一个值，该值只能是对应指定的实体

用途：使用一对一关系，我们可以扩展原本的用户系统

Django自带的用户系统中并没有诸如头像、昵称、手机号等字段，我们可以创建一个新的模型一对一对应原有的模型

auth_user	accounts_userprofile
 id: int	 id: int
password: varchar(128)	avatar: varchar(100)
last_login: datetime(6)	nickname: varchar(200)
is_superuser: tinyint(1)	created_at: datetime(6)
 username: varchar(150)	updated_at: datetime(6)
first_name: varchar(150)	 user_id: int
last_name: varchar(150)	
email: varchar(254)	
is_staff: tinyint(1)	
is_active: tinyint(1)	
date_joined: datetime(6)	

Django ORM 表单关系

多对一关系——ForeignKey

什么是多对一关系？

在Twitter系统中，一个用户可以发多个帖子，多个帖子对应一个人，这便是多对一关系。

每个帖子下面可以有多个评论，这些评论和这个帖子也形成了多对一关系



Django ORM 表单关系

多对一关系——ForeignKey

多对一的关系。需要两个位置参数：模型相关的类和 on_delete 选项。

```
class Comment(models.Model):
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    tweet = models.ForeignKey(Tweet, null=True, on_delete=models.SET_NULL)
    content = models.TextField(max_length=140)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

上面的Comment类定义了两个外键，即有两个多对一关系，user对应评论者，tweet对应被评论的推文

Django ORM 表单关系

多对多关系——ManyToManyField

多对多关系，就像一个对象可以关联多个对象，被关联的对象也可以被其他多个对象关联

1



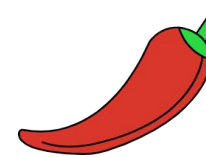
2



3



菜肴id	原料id
1	2
1	3
2	3
2	4
2	1
2	6
3	1
3	3
3	5
3	6



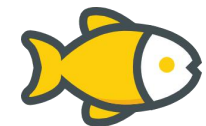
1



2



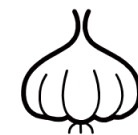
3



4



5



6

Django ORM 表单关系

```
class Tweet(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    photos = models.ManyToManyField(Photo, blank=True)
    likes_count = models.BigIntegerField(default=0, null=True)
    comments_count = models.BigIntegerField(default=0, null=True)
```

```
class Photo(models.Model):
    status = models.IntegerField(
        default=PhotoStatus.PENDING,
        choices=PHOTO_STATUS_CHOICES,
    )
    file = models.FileField()
    user = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    has_deleted = models.BooleanField(default=False)
    deleted_at = models.DateTimeField(null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    created_at_date = models.DateField(auto_now_add=True)
```




Tweet的photos字段就对应的**Photo**模型，即一张图片可以被多个推文使用，同时一个推文可以同时上传多张图片

Django ORM 表单关系







多对多关系——ManyToManyField

```
photos = models.ManyToManyField(Photo, blank=True)
```

多对多的关系。需要一个位置参数：模型相关的类，它的工作原理与 ForeignKey 完全相同，包括 递归 和 惰性 关系。在幕后，Django 创建了一个中间连接表来表示多对多的关系。

tweets_tweet	
	id: int
	content: longtext
	created_at: datetime(6)
	user_id: int
	comments_count: bigint
	likes_count: bigint

tweets_tweet_photos	
	id: int
	tweet_id: int
	photo_id: int

photos_photo	
	id: int
	status: int
	file: varchar(100)
	has_deleted: tinyint(1)
	deleted_at: datetime(6)
	created_at: datetime(6)
	created_at_date: date
	user_id: int

QuerySet执行查询

05

- ▶ 创建对象
- ▶ 将修改保存至对象
- ▶ 检索对象
- ▶ 删除对象

Django ORM QuerySet

前面介绍了如果定义模型，以及ORM字段和SQL类型的对应关系。接下来介绍如何通过ORM从数据库中**增删改查**数据

一旦创建 数据模型 后，Django 自动给予你一套数据库抽象 API，允许你创建，检索，更新和删除对象。本页介绍如何使用这些 API。

一个 QuerySet 代表来自数据库中对象的一个集合。它可以有 0 个，1 个或者多个 filters. Filters, 可以根据给定参数缩小查询结果量。在 SQL 的层面上，QuerySet 对应 SELECT 语句，而 *filters* 对应类似 WHERE 或 LIMIT 的限制子句。

创建对象

为了用 Python 对象展示数据表对象，Django 使用了一套直观的系统：一个模型类代表一张数据表，一个模型类的实例代表数据库表中的一行记录。

要创建一个对象，用关键字参数初始化它，然后调用 `save()` 将其存入数据库。

```
from django.contrib.auth import authenticate

user = authenticate(username='admin', password='xxx')

tweet = Tweet(user=user, content="九章大法好")
tweet.save()
```

这在幕后执行了 INSERT SQL 语句。Django 在你显式调用 `save()` 才操作数据库。

修改对象

要将修改保存至数据库中已有的某个对象，使用 `save()`。

修改刚刚的推文内容，并调用`save`保存

```
tweet.content="九章大法超级好"  
tweet.save()
```

这在幕后执行了 `UPDATE SQL` 语句。Django 在你显示调用 `save()` 后才操作数据库。

检索对象

要从数据库检索对象，要通过模型类的 Manager 构建一个 **QuerySet**。

一个 **QuerySet** 代表来自数据库中对象的一个集合。在 SQL 的层面上，QuerySet 对应 **SELECT** 语句，而 **filters** 对应类似 **WHERE** 或 LIMIT 的限制子句。

你能通过模型的 Manager 获取 QuerySet。每个模型至少有一个 Manager，默认名称是 **objects**。像这样直接通过模型类使用它：

```
In [1]: from comments.models import Comment
```

```
In [2]: Comment.objects
```

```
Out[2]: <django.db.models.manager.Manager at 0x22d3d421d90>
```

检索对象

查找全部记录

Manager 是模型的 QuerySets 主要来源。例如 `Comment.objects.all()` 返回了一个 QuerySet，后者包含了数据库中所有的 Comment 记录。

```
queryset = Comment.objects.all()
```

从数据库中检索对象最简单的方式就是检索全部。为此，在 Manager 上调用 `all()` 方法：

方法 `all()` 返回了一个包含数据库中所有对象的 QuerySet 对象。

检索对象

通过过滤器检索指定对象

all() 返回的 **QuerySet** 包含了数据表中**所有**的对象。

大多数情况下，只需要完整对象集合的一个子集。这个时候可以使用filter或者exclude方法

Django ORM提供了两个方法帮助我们缩写查询范围

- ▶ **filter**方法对应SQL中的**where**子句, where condition
- ▶ **exclude**方法是filter方法的反选, where not(condition)

检索对象

filter(**kwargs)

返回一个新的 QuerySet，包含的对象满足给定查询参数

```
Comment.objects.filter( updated_at__gte = datetime.date.today() )
```

这条语句将查询更新日期大于今天的评论

TIPS: **__gte**表示**大于等于**

filter对应SQL中的where子句

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at`  
FROM `comments_comment` WHERE `comments_comment`.`updated_at` >= '2021-04-20 00:00:00';
```

检索对象

exclude(**kwargs)

返回一个新的 QuerySet，包含的对象 不 满足给定查询参数。

```
Comment.objects.exclude( updated_at__gte = datetime.date.today( ) )
```

exclude和filter是相反的

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at`  
FROM `comments_comment` WHERE NOT ( `comments_comment`.`updated_at` >= '2021-04-20 00:00:00' )
```

检索对象

链式过滤器

精炼 QuerySet 的结果本身还是一个 QuerySet，所以能串联精炼过程。

```
Comment.objects.exclude(  
    updated_at__gte = datetime.date.today()  
)<filter created_at__gte=datetime.date.today()  
>
```

通过如上的语句可以筛选出comment表下所有的今日创建并且是今天更新的评论

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at`  
FROM `comments_comment` WHERE (NOT (`comments_comment`.`updated_at` >= '2021-04-20 00:00:00'))  
AND `comments_comment`.`created_at` >= '2021-04-20 00:00:00')
```

检索对象

每个 QuerySet 都是唯一的

每次精炼一个 QuerySet，你就会获得一个全新的 QuerySet，后者与前者毫无关联。每次精炼都会创建一个单独的、不同的 QuerySet，能被存储，使用和复用。

```
queryset = Comment.objects.all()
```

```
q1 = queryset.exclude(created_at__gte=datetime.date.today())
```

```
q2 = queryset.filter(updated_at__gte=datetime.date.today())
```

这三个 QuerySets 是独立的。第一个是基础 QuerySet，查询所有评论数据。第二个是第一个的子集，带有额外条件，排除了 created_at 是今天和今天之后的所有记录。第三个是第一个的子集，带有额外条件，只筛选 updated_at 是今天或未来的所有记录。最初的 QuerySet (q1) 不受筛选操作影响。

检索对象

QuerySet 是惰性的

QuerySet 是惰性的 —— 创建 QuerySet 并不会引发任何数据库活动。你可以将一整天的过滤器都堆积在一起，Django 只会在 QuerySet 被计算时执行查询操作。来瞄一眼这个例子：

```
q = Comment.objects.all()
q = q.exclude(created_at__gte=datetime.date.today())
q = queryset.filter(updated_at__gte=datetime.date.today())
print( q )
```

虽然这看起来像是三次数据库操作，实际上只在最后一行 (print(q)) 做了一次。一般来说，QuerySet 的结果直到你 “要使用” 时才会从数据库中拿出。当你要用时，才通过数据库 计算出 QuerySet。

检索对象

用 get() 检索单个对象

filter() 总是返回一个 QuerySet，即便只有一个对象满足查询条件——这种情况下，QuerySet 只包含了一个元素。若你知道只会有一个对象满足查询条件，你可以在 Manager 上使用 get() 方法，它会直接返回这个对象：

```
Comment.objects.get()
```

除了get之外，还有first和last

```
Comment.objects.first()
```

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at` FROM  
`comments_comment` ORDER BY `comments_comment`.`id` ASC LIMIT 1;
```


检索对象

first()和last()

除了get之外，还有first和last

```
Comment.objects.first()
```

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at` FROM  
`comments_comment` ORDER BY `comments_comment`.`id` ASC LIMIT 1;
```

```
Comment.objects.last()
```

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at` FROM  
`comments_comment` ORDER BY `comments_comment`.`id` DESC LIMIT 1;
```


检索对象

限制 QuerySet 条目数

利用 Python 的数组**切片**语法将 QuerySet 切成指定长度。这等价于 SQL 的 **LIMIT** 和 **OFFSET** 子句。

例如，这将返回前 5 个对象 (LIMIT 5):

```
Comment.objects.all()[:5]
```

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at` FROM  
`comments_comment` LIMIT 5;
```

TIPS: 不支持**负索引** (例如 `Entry.objects.all()[-1]`)

检索对象

限制 QuerySet 条目数

```
Comment.objects.all()[10:20]
```

上面的语句可以获取第11到第20条记录

这个功能非常适合用于[分页](#)的场景

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at` FROM  
`comments_comment` LIMIT 10 OFFSET 10;
```

检索对象

限制 QuerySet 条目数

切片一个未执行的 QuerySet 通常会返回另一个未执行的 QuerySet，但如果使用切片语法的 **step** 参数，Django 会执行数据库查询，并返回一个列表。

```
Comment.objects.all()[::2]
```

使用步长为2的切片，步长信息并没有出现在SQL语句中，返回的也是QuerySet，而是一个列表，因此，如果在切片中使用步长，会返回包含所有对象的列表，并对列表再做切片操作。

```
SELECT `comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at`  
FROM `comments_comment`;
```

检索对象

字段查询

SQL中有WHERE子句帮助我们精确查找

我也可以在filter函数中通过参数来使用where子句

```
Comment.objects.filter(  
    user=1,  
    tweet=32198,  
    content__startswith='like',  
    updated_at__gte=datetime.date.today() )
```

```
SELECT  
`comments_comment`.`id`, `comments_comment`.`user_id`, `comments_comment`.`tweet_id`,  
`comments_comment`.`content`, `comments_comment`.`created_at`, `comments_comment`.`updated_at`  
FROM `comments_comment`  
WHERE (`comments_comment`.`content` LIKE BINARY 'like%'  
    AND `comments_comment`.`tweet_id` = 32198  
    AND `comments_comment`.`updated_at` >= '2021-04-20 00:00:00'  
    AND `comments_comment`.`user_id` = 1)
```

检索对象

介绍django model 的一些常用查询指令

Mysql中有大量的**运算符**，Django ORM提供了在参数后面加**两个下划线**的方式来模拟运算符操作

TIPS:

如果为比较提供的值是**None**，它将被解释为SQL **NULL**，下面两句话是等效的。

```
Entry.objects.get(id__exact=None)
```

```
SELECT ... WHERE id IS NULL;
```

__exact	精确等于 like 'aaa'
__iexact	精确等于 忽略大小写 ilike 'aaa'
__contains	包含 like '%aaa%'
__icontains	包含 忽略大小写 ilike '%aaa%'
__gt	大于
__gte	大于等于
__lt	小于
__lte	小于等于
__in	存在于一个list范围内
__startswith	以...开头
__istartswith	以...开头 忽略大小写
__endswith	以...结尾
__iendswith	以...结尾，忽略大小写
__range	在...范围内
__year	日期字段的年份
__month	日期字段的月份
__day	日期字段的日
__isnull	True/False

检索对象

前面的内容筛选出来的都是表的所有字段，但是有些是我们只需要表中的**部分字段**就可以了

比如只想知道某条推文的点赞数和评论数

可以借助**values** 方法实现，每个参数表示一个字段

```
Tweet.objects.values('likes_count', 'comments_count').filter(id = 1)
```

上面的语句将被翻译为下面的表达式

```
SELECT `tweets_tweet`.`likes_count`, `tweets_tweet`.`comments_count`  
FROM `tweets_tweet`  
WHERE `tweets_tweet`.`id` = 1 ;
```


删除对象

通常，删除方法被命名为 **delete()**。该方法立刻删除对象，并返回被删除的对象数量和一个包含了每个被删除对象类型的数量的**字典**。例子：

```
tweet = Tweet.objects.first()
tweet.delete()

# (1, {'tweets.Tweet': 1})
```

你也能批量删除对象。所有 QuerySet 都有个 delete() 方法，它会删除 QuerySet 中的所有成员。

下面的代码将删除**所有**点赞数小于100的推文

```
queryset = Tweet.objects.filter(likes_count__lt = 100)

queryset.delete()
```


删除对象

当 Django 删除某个对象时，默认会模仿 SQL 约束 ON DELETE CASCADE 的行为——换言之，某个对象被删除时，关联对象也会被删除。例如在删除该推文之前会把与其关联的图片先删除

```
DELETE FROM `tweets_tweet_photos` WHERE `tweets_tweet_photos`.`tweet_id` IN (4);  
  
DELETE FROM `tweets_tweet` WHERE `tweets_tweet`.`id` IN (4);
```

输出 SQL 语句

Django ORM训练

为了能让我们知道每一个ORM和数据库SQL之间的关系

可以在项目的settings.py文件中添加右侧的LOGGING属性，可以将我们的操作数据库的语句翻译为SQL语句。

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'DEBUG',
        },
    },
}
```

交互式解释器

通过 `python manage.py shell` 便可启动 Python 交互式解释器。

这样便可以方便的学习Django的ORM操作

```
In [6]: Tweet(user=user).save()
(0.000) SELECT VERSION(); args=None
(0.032) INSERT INTO `tweets_tweet` (`user_id`, `content`, `created_at`, `likes_count`, `comments_count`)
VALUES (1, '', '2021-04-22 06:49:38.779107', 0, 0); args=[1, '', '2021-04-22 06:49:38.779107', 0, 0]

In [7]: Tweet.objects.first()
(0.000) SELECT `tweets_tweet`.`id`, `tweets_tweet`.`user_id`, `tweets_tweet`.`content`,
`tweets_tweet`.`created_at`, `tweets_tweet`.`likes_count`, `tweets_tweet`.`comments_count` FROM
`tweets_tweet` ORDER BY `tweets_tweet`.`id` ASC LIMIT 1; args=()
Out[7]: (0.000) SELECT `auth_user`.`id`, `auth_user`.`password`, `auth_user`.`last_login`,
`auth_user`.`is_superuser`, `auth_user`.`username`, `auth_user`.`first_name`, `auth_user`.`last_name`,
`auth_user`.`email`, `auth_user`.`is_staff`, `auth_user`.`is_active`, `auth_user`.`date_joined` FROM
`auth_user` WHERE `auth_user`.`id` = 1 LIMIT 21; args=(1,)
<Tweet: 2021-04-22 06:47:24.399599+00:00 admin: >
```