Django 1.8.2 文档

模型

模型是你的数据的唯一的、权威的信息源。它包含你所储存数据的必要字段和行为。通常，每个模型对应数据库中唯一的一张表。

基础：

- 每个模型都是`django.db.models.Model`的一个Python 子类。
- 模型的每个属性都表示数据库中的一个字段。
- Django 提供一套自动生成的用于数据库访问的API；详见执行查询。

简短的例子

这个例子定义一个`Person`模型，它有`first_name` 和`last_name` 两个属性：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name`和`last_name`是模型的两个字段。每个字段都被指定成一个类属性，每个属性映射到一个数据库的列。

上面的`Person` 模型会在数据库中创建这样一张表：

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

一些技术上的注意事项：

- 这个表的名称`myapp_person`，是根据 模型中的元数据自动生成的，也可以覆写为别的名称，详见Table names。
- `id` 字段是自动添加的，但这个行为可以被重写。详见自增主键字段。
- 这个例子中的`CREATE TABLE` SQL 语句使用PostgreSQL 语法格式，要注意的是Django 会根据设置文件 中指定的数据库类型来使用相应的SQL 语句。

使用模型

定义好模型之后，接下来你需要告诉Django 使用这些模型。你要做的就是修改配置文件中的`INSTALLED_APPS` 设置，在其中添加`models.py`所在应用的名称。

例如，如果你的应用的模型位于`myapp.models`模块（`manage.py startapp` 脚本为一个应用创建的包结构），`INSTALLED_APPS`部分看上去应该是：

```
INSTALLED_APPS = (
    #...
    'myapp',
    #...
)
```

当你在`INSTALLED_APPS` 中添加新的应用名时，请确保运行命令`manage.py migrate`，可以事先使用`manage.py makemigrations` 给应用生成迁移脚本。

字段

模型中不可或缺且最为重要的，就是字段集，它是一组数据库字段的列表。字段被指定为类属性。 要注意选择的字段名称不要和模型 API 冲突，比如`clean`、`save` 或者`delete`。

目录

浏览

- 上一页：模型和数据库

例如：

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

字段类型

模型中的每个字段都是 `Field` 子类的某个实例。Django 根据字段类的类型确定以下信息：

- 数据库当中的列类型 (比如: `INTEGER`, `VARCHAR`)。
- 渲染表单时使用的默认HTML 部件（例如，`<input type="text">`, `<select>`）。
- 最低限度的验证需求，它被用在 Django 管理站点和自动生成的表单中。

Django 自带数十种内置的字段类型；完整字类型列表可以在模型字段参考 中找到。如果内置类型仍不能满足你的要求，你可以自由地编写符合你要求的字段类型； 详见编写自定义的模型字段。

字段选项

每个字段有一些特有的参数，详见模型字段参考。例如，`CharField`（和它的派生类）需要`max_length` 参数来指定`VARCHAR` 数据库字段的大小。

还有一些适用于所有字段的通用参数。 这些参数在参考中有详细定义，这里我们只简单介绍一些最常用的：

`null`

如果为`True`，Django 将用`NULL` 来在数据库中存储空值。 默认值是 `False`.

`blank`

如果为`True`，该字段允许不填。默认为`False`。

要注意，这与 `null` 不同。`null`纯粹是数据库范畴的，而 `blank` 是数据验证范畴的。如果一个字段的 `blank=True`，表单的验证将允许该字段是空值。如果字段的`blank=False`，该字段就是必填的。

`choices`

由二元组组成的一个可迭代对象（例如，列表或元组），用来给字段提供选择项。 如果设置了choices ，默认的表单将是一个选择框而不是标准的文本框，而且这个选择框的选项就是choices 中的选项。

这是一个关于 choices 列表的例子：

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

每个元组中的第一个元素，是存储在数据库中的值；第二个元素是在管理界面或 ModelChoiceField 中用作显示的内容。 在一个给定的 model 类的实例中，想得到某个 choices 字段的显示值，就调用 `get_FOO_display` 方法 (这里的 FOO 就是 choices 字段的名称)。例如：

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```


- 下一页：执行查询

你在这里：

- Django 1.8.2 文档
  - 使用 Django
    - 模型和数据库
      - 模型

本页

- Show Source

Quick search

Go

Enter search terms or a module, class or function name.

最后更新：

May 13, 2015

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`default`

字段的默认值。可以是一个值或者可调用对象。如果可调用，每有新对象被创建它都会被调用。

`help_text`

表单部件额外显示的帮助内容。即使字段不在表单中使用，它对生成文档也很有用。

`primary_key`

如果为`True`，那么这个字段就是模型的主键。

如果你没有指定任何一个字段的`primary_key=True`，Django 就会自动添加一个`IntegerField`字段做为主键，所以除非你想覆盖默认的主键行为，否则没必要设置任何一个字段的`primary_key=True`。详见自增主键字段。

主键字段是只读的。如果你在一个已存在的对象上面更改主键的值并且保存，一个新的对象将会在原有对象之外创建出来。例如：

```
from django.db import models

class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name='Apple')
>>> fruit.name = 'Pear'
>>> fruit.save()
>>> Fruit.objects.values_list('name', flat=True)
['Apple', 'Pear']
```

`unique`

如果该值设置为 `True`, 这个数据字段的值在整张表中必须是唯一的

再说一次，这些仅仅是常用字段的简短介绍， 要了解详细内容，请查看 通用 model 字段选项参考(common model field option reference).

自增主键字段

默认情况下，Django 会给每个模型添加下面这个字段：

```
id = models.AutoField(primary_key=True)
```

这是一个自增主键字段。

如果你想指定一个自定义主键字段，只要在某个字段上指定 `primary_key=True` 即可。如果 Django 看到你显式地设置了 `Field.primary_key`，就不会自动添加 `id` 列。

每个模型只能有一个字段指定`primary_key=True`（无论是显式声明还是自动添加）。

字段的自述名

除`ForeignKey`、`ManyToManyField` 和 `OneToOneField` 之外，每个字段类型都接受一个可选的位置参数 —— 字段的自述名。如果没有给定自述名，Django 将根据字段的属性名称自动创建自述名 —— 将属性名称的下划线替换成空格。

在这个例子中，自述名是 `"person's first name"`：

```
first_name = models.CharField("person's first name", max_length=30)
```

在这个例子中，自述名是 `"first name"`：

```
first_name = models.CharField(max_length=30)
```

ForeignKey、ManyToManyField 和 OneToOneField 都要求第一个参数是一个模型类，所以要使用 verbose_name 关键字参数才能指定自述名：

```
poll = models.ForeignKey(Poll, verbose_name="the related poll")
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(Place, verbose_name="related place")
```

习惯上，verbose_name 的首字母不用大写。Django 在必要的时候会自动大写首字母。

关系

显然，关系数据库的威力体现在表之间的相互关联。 Django 提供了三种最常见的数据库关系：多对一(many-to-one)，多对多(many-to-many)，一对一(one-to-one)。

多对一关系

Django 使用 django.db.models.ForeignKey 定义多对一关系。和使用其它**字段**类型一样：在模型当中把它做为一个类属性包含进来。

ForeignKey 需要一个位置参数：与该模型关联的类。

比如，一辆Car有一个Manufacturer —— 但是一个Manufacturer 生产很多Car，每一辆Car 只能有一个 Manufacturer —— 使用下面的定义：

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    # ...
```

你还可以创建递归的关联关系（对象和自己进行多对一关联）和 与尚未定义的模型的关联关系；详见模型字段参考。

建议你用被关联的模型的小写名称做为ForeignKey 字段的名字（例如，上面manufacturer）。当然，你也可以起别的名字。例如：

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

另见

ForeignKey 字段还接受许多别的参数，在模型字段参考有详细介绍。这些选项帮助定义关联关系应该如何工作；它们都是可选的参数。

访问反向关联对象的细节，请见Following relationships backward example。

示例代码，请见多对一关系模型示例)。

多对多关系

ManyToManyField 用来定义多对多关系，用法和其他Field 字段类型一样：在模型中做为一个类属性包含进来。

ManyToManyField 需要一个位置参数：和该模型关联的类。

例如，一个**披萨**可以有多种**馅料** —— 一种**馅料** 可以位于多个披萨上，而且每个**披萨** 也可以有多种馅料 —— 如下展示：

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

和使用ForeignKey一样，你也可以创建递归的关联关系（对象与自己的多对多关联）和与尚未定义关系的模型的关联关系；详见模型字段参考。

建议你以被关联模型名称的复数形式做为ManyToManyField的名字（例如上例中的toppings）。

在哪个模型中设置ManyToManyField并不重要，在两个模型中任选一个即可——不要两个模型都设置。

通常，ManyToManyField实例应该位于可以编辑的表单中。在上面的例子中，toppings位于Pizza中（而不是在Topping里面设置pizzas的ManyToManyField字段），因为设想一个Pizza有多种Topping比一个Topping位于多个Pizza上要更加自然。按照上面的方式，在Pizza的表单中将允许用户选择不同的Toppings。

另见

完整的示例参见多对多关系模型示例。

ManyToManyField字段还接受别的参数，在模型字段参考中有详细介绍。这些选项帮助定义关系应该如何工作；它们都是可选的。

多对多关系中的其他字段

处理类似搭配pizza和topping这样简单的多对多关系时，使用标准的ManyToManyField就可以了。但是，有时你可能需要关联数据到两个模型之间的关系上。

例如，有这样一个应用，它记录音乐家所属的音乐小组。我们可以用一个ManyToManyField表示小组和成员之间的多对多关系。但是，有时你可能想知道更多成员关系的细节，比如成员是何时加入小组的。

对于这些情况，Django允许你指定一个模型来定义多对多关系。你可以将其他字段放在中介模型里面。源模型的ManyToManyField字段将使用through参数指向中介模型。对于上面的音乐小组的例子，代码如下：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

在设置中介模型时，要显式指定外键并关联到多对多关系涉及的模型。这个显式声明定义两个模型之间是如何关联的。

中介模型有一些限制：

- 中介模型必须有且只有一个外键到源模型（上面例子中的Group），或者你必须使用ManyToManyField.through_fields显式指定Django应该使用的外键。如果你的模型中存在不止一个外键，并且through_fields没有指定，将会触发一个无效的错误。对目标模型的外键有相同的限制（上面例子中的Person）。
- 对于通过中介模型与自己进行多对多关联的模型，允许存在到同一个模型的两个外键，但它们将被作为多对多关联关系的两个（不同的）方面。如果有超过 两个外键，同样你必须像上面一样指定through_fields，否则将引发一个验证错误。
- 使用中介模型定义与自身的多对多关系时，你必须设置symmetrical=False（详见模型字段参考）。

Changed in Django 1.7:

在Django 1.6及之前的版本中，中介模型禁止包含多于一个的外键。

既然你已经设置好ManyToManyField来使用中介模型（在这个例子中就是Membership），接下来你要开始创建多对多关系。你要做的就是创建中介模型的实例：

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

与普通的多对多字段不同，你不能使用add、create和赋值语句（比如，beatles.members = [...]）来创建关系：

```
# THIS WILL NOT WORK
>>> beatles.members.add(john)
# NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
# AND NEITHER WILL THIS
>>> beatles.members = [john, paul, ringo, george]
```

为什么不能这样做？ 这是因为你不能只创建 Person和 Group之间的关联关系，你还要指定 Membership模型中所需要的所有信息；而简单的add、create 和赋值语句是做不到这一点的。所以它们不能在使用中介模型的多对多关系中使用。此时，唯一的办法就是创建中介模型的实例。

remove()方法被禁用也是出于同样的原因。但是clear() 方法却是可用的。它可以清空某个实例所有的多对多关系：

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
[]
```

通过创建中介模型的实例来建立对多对多关系后，你就可以执行查询了。 和普通的多对多字段一样，你可以直接使用被关联模型的属性进行查询：

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

如果你使用了中介模型，你也可以利用中介模型的属性进行查询：

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr]
```

如果你需要访问一个成员的信息，你可以直接获取Membership模型：

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

另一种获取相同信息的方法是，在Person对象上查询多对多反转关系：

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

一对一关系

OneToOneField用来定义一对一关系。 用法和其他字段类型一样：在模型里面做为类属性包含进来。

当某个对象想扩展自另一个对象时，最常用的方式就是在这个对象的主键上添加一对一关系。

OneToOneField要一个位置参数：与模型关联的类。

例如，你想建一个“places”数据库，里面有一些常用的字段，比如address、 phone number 等等。 接下来，如果你想在Place 数据库的基础上建立一个Restaurant 数据库，而不想将已有的字段复制到Restaurant模型，那你可以在 Restaurant 添加一个OneToOneField 字段，这个字段指向Place（因为Restaurant 本身就是一个 Place；事实上，在处理这个问题的时候，你应该使用一个典型的 继承，它隐含一个一对一关系)。

和使用 ForeignKey一样，你可以定义 递归的关联关系和引用尚未定义关系的模型。详见模型字段参考。

另见

在一对一关系的模型例子 中有一套完整的例子。

OneToOneField字段也接受一个特定的可选的parent_link参数，在模型字段参考 中有详细介绍。

在以前的版本中，OneToOneField 字段会自动变成模型 的主键。不过现在已经不这么做了(不过要是你愿意的话，你仍可以传递 primary_key参数来创建主键字段)。所以一个 模型 中可以有多个OneToOneField 字段。

跨文件的模型

访问其他应用的模型是非常容易的。 在文件顶部你定义模型的地方，导入相关的模型来实现它。然后，无论在哪里需要的话，都可以引用它。例如：

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(ZipCode)
```

字段命名的限制

Django 对字段的命名只有两个限制：

1. 字段的名称不能是Python 保留的关键字，因为这将导致一个Python 语法错误。例如：

```
class Example(models.Model):
    pass = models.IntegerField() # 'pass' is a reserved word!
```

2. 由于Django 查询语法的工作方式，字段名称中连续的下划线不能超过一个。例如：

```
class Example(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
```

这些限制有变通的方法，因为没有要求字段名称必须与数据库的列名匹配。参 db_column 选项。

SQL 的保留字例如join、where 和select，可以用作模型的字段名，因为Django 会对底层的SQL 查询语句中的数据库表名和列名进行转义。 它根据你的数据库引擎使用不同的引用语法。

自定义字段类型

如果已有的模型字段都不合适，或者你想用到一些很少见的数据库列类型的优点，你可以创建你自己的字段类型。创建你自己的字段在编写自定义的模型字段中有完整讲述。

元选项

使用内部的`class Meta`定义模型的元数据，例如：

```
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

模型元数据是“任何不是字段的数据”，比如排序选项（`ordering`），数据表名（`db_table`）或者人类可读的单复数名称（`verbose_name`和`verbose_name_plural`）。在模型中添加`class Meta`是完全可选的，所有选项都不是必须的。

所有元选项的完整列表可以在模型选项参考找到。

模型的属性

`objects`

模型最重要的属性是`Manager`。它是Django 模型进行数据库查询操作的接口，并用于从数据库获取实例。如果没有自定义`Manager`，则默认的名称为`objects`。Managers 只能通过模型类访问，而不能通过模型实例访问。

模型的方法

可以在模型上定义自定义的方法来给你的对象添加自定义的“底层”功能。`Manager`方法用于“表范围”的事务，模型的方法应该着眼于特定的模型实例。

这是一个非常有价值的技术，让业务逻辑位于同一个地方 —— 模型中。

例如，下面的模型具有一些自定义的方法：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

这个例子中的最后一个方法是一个`property`。

模型实例参考具有一个完整的为模型自动生成的方法列表。你可以覆盖它们 —— 参见下文覆盖模型预定义的方法 —— 但是有些方法你会始终想要重新定义：

`__str__()` (Python 3)

Python 3 equivalent of `__unicode__()`.

`__unicode__()` (Python 2)

一个Python的特殊方法（magic method），返回对象的字符串表达式(unicode格式）当模型实例需要强制转换并显示为普通的字符串时，Python 和Django 将使用这个方法。最明显是在交互式控制台或者管理站点显示一个对象的时候。

你将永远想要定义这个方法；默认的方法几乎没有意义。

`get_absolute_url()`

它告诉Django 如何计算一个对象的URL。Django 在它的管理站点中使用到这个方法，在其它任何需要计算一个对象的URL 时也将用到。

任何具有唯一标识自己的URL 的对象都应该定义这个方法。

覆盖预定义的模型方法

还有另外一部分封装数据库行为的模型方法，你可能想要自定义它们。特别是，你将要经常改变`save()` 和`delete()` 的工作方式。

你可以自由覆盖这些方法（和其它任何模型方法）来改变它们的行为。

覆盖内建模型方法的一个典型的使用场景是，你想在保存一个对象时做一些其它事情。例如（参见`save()` 中关于它接受的参数的文档）：

```
from django.db import models

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

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
        do_something_else()
```

你还可以阻止保存：

```
from django.db import models

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

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "real" save()
method.
```

必须要记住调用超类的方法——`super(Blog, self).save(*args, **kwargs)` —— 来确保对象被保存到数据库中。如果你忘记调用超类的这个方法，默认的行为将不会发生且数据库不会有任何改变。

还要记住传递参数给这个模型方法 —— 即`*args, **kwargs`。 Django 未来将一直会扩展内建模型方法的功能并添加新的参数。如果在你的方法定义中使用`*args, **kwargs`，将保证你的代码自动支持这些新的参数。

批量操作中被覆盖的模型方法不会被调用

注意，当使用查询集批量删除对象时，将不会为每个对象调用`delete()` 方法。为确保自定义的删除逻辑得到执行，你可以使用`pre_delete` 和/或`post_delete` 信号。

不幸的是，当批量`creating` 或`updating` 对象时没有变通方法，因为不会调用`save()`、`pre_save`和 `post_save`。

执行自定义的SQL

另外一个常见的需求是在模型方法和模块级别的方法中编写自定义的SQL 语句。关于使用原始SQL 语句的更多细节，参见使用原始 SQL 的文档。

模型继承

Django 中的模型继承与 Python 中普通类继承方式几乎完全相同，但是本页头部列出的模型基本的要求还是要遵守。这表示自定义的模型类应该继承`django.db.models.Model`。

你唯一需要作出的决定就是你是想让父模型具有它们自己的数据库表，还是让父模型只持有一些共同的信息而这些信息只有在子模型中才能看到。

在Django 中有3种风格的继承。

1. 通常，你只想使用父类来持有一些信息，你不想在每个子模型中都敲一遍。这个类永远不会单独使用，所以你使

用抽象基类。

2. 如果你继承一个已经存在的模型且想让每个模型具有它自己的数据库表，那么应该使用多表继承。

3. 最后，如果你只是想改变模块Python 级别的行为，而不用修改模型的字段，你可以使用代理模型。

抽象基类

当你想将一些常见信息存储到很多model的时候，抽象化类是十分有用的。你编写完基类之后，在 Meta类中设置 abstract=True ，该类就不能创建任何数据表。取而代之的是，当它被用来作为一个其他model的基础类时，它将被加入那一子类中。如果抽象化基础类和它的子类有相同的项，那么将会出现error（并且Django将返回一个exception）。

一个例子

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

Student 模型将有三个项：name, age 和 home_group。CommonInfo 模型无法像一般的Django模型一样使用，因为它是一个抽象化基础类。它无法生成数据表单或者管理器，并且不能实例化或者储存。

对很多用户来说, 这种类型的模型继承就是你想要的。它提供一种在 Python 语言层级上提取公共信息的方式，但在数据库层级上，各个子类仍然只创建一个数据库表。

元 继承

当一个抽象类被创建的时候, Django会自动把你在基类中定义的 Meta 作为子类的一个属性。如果子类没有声明自己的Meta 类, 他将会继承父类的Meta. 如果子类想要扩展父类 的，可以继承父类的 Meta 即可，例如

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

继承时，Django 会对基类的 Meta类做一个调整：在安装 Meta属性之前，Django 会设置 abstract=False。这意味着抽象基类的子类不会自动变成抽象类。 当然，你可以让一个抽象类继承另一个抽象基类，不过每次都要显式地设置 abstract=True。

对于抽象基类而言，有些属性放在 Meta 内嵌类里面是没有意义的。例如，包含 db_table将意味着所有的子类(是指那些没有指定自己的 Meta 类的子类)都使用同一张数据表，一般来说，这并不是我们想要的。

小心使用 related_name

如果你在 ForeignKey或 ManyToManyField字段上使用 related_name属性，你必须总是为该字段指定一个唯一的反向名称。但在抽象基类上这样做就会引发一个很严重的问题。因为 Django 会将基类字段添加到每个子类当中，而每个子类的字段属性值都完全相同 (这里面就包括related_name)。

当你在(且仅在)抽象基类中使用 related_name 时，如果想绕过这个问题，名称中就要包含'%(app_label)s' 和 '%(class)s'。

- '%(class)s' 会替换为子类的小写加下划线格式的名称，字段在子类中使用。
- '%(app_label)s' 会替换为应用的小写加下划线格式的名称，应用包含子类。每个安装的应用名称都应该是唯一的，而且应用里每个模型类的名称也应该是唯一的，所以产生的名称应该彼此不同。

例如，假设有一个app叫做common/models.py：

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="%(app_label)s_%
(class)s_related")

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

以及另一个应用 rare/models.py：

```
from common.models import Base

class ChildB(Base):
    pass
```

ChildA.m2m 字段的反向名称是 childa_related，而 ChildB.m2m 字段的反向名称是 childb_related。这取决于你如何使用 '%(class)s' 和'%(app_label)s来构造你的反向名称。如果你没有这样做，Django 就会在验证 model (或运行 migrate) 时抛出错误。

如果你没有在抽象基类中为某个关联字段定义 related_name 属性，那么默认的反向名称就是子类名称加上 '_set'，它能否正常工作取决于你是否在子类中定义了同名字段。例如，在上面的代码中，如果去掉 related_name属性，在 ChildA中，m2m 字段的反向名称就是 childa_set；而 ChildB的 m2m 字段的反向名称就是 childb_set。

多表继承

这是 Django 支持的第二种继承方式。使用这种继承方式时，同一层级下的每个子 model 都是一个真正意义上完整的 model 。 每个子 model 都有专属的数据表，都可以查询和创建数据表。 继承关系在子 model 和它的每个父类之间都添加一个链接 (通过一个自动创建的 OneToOneField来实现)。 例如：

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

Place里面的所有字段在 Restaurant中也是有效的，只不过数据保存在另外一张数据表当中。所以下面两个语句都是可以运行的：

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

如果你有一个 Place ，它同时也是一个 Restaurant， 那么你可以使用子 model 的小写形式从 Place 对象中获得与其对应的 Restaurant对象：

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

但是，如果上例中的 p 并不是 Restaurant (比如它仅仅只是 Place对象，或者它是其他类的父类)，那么在引用 p.restaurant就会抛出Restaurant.DoesNotExist 异常。

多表继承中的Meta

在多表继承中，子类继承父类的 Meta类是没什么意义的。所有的 Meta 选项已经对父类起了作用，再次使用只会起反作用。(这与使用抽象基类的情况正好相反，因为抽象基类并没有属于它自己的内容)

所以子 model 并不能访问它父类的 Meta 类。但是在某些受限的情况下，子类可以从父类继承某些 Meta ：如果子类没有指定 ordering属性或 get_latest_by 属性，它就会从父类中继承这些属性。

如果父类有了排序设置，而你并不想让子类有任何排序设置，你就可以显式地禁用排序：

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

继承与反向关联

因为多表继承使用了一个隐含的 OneToOneField来链接子类与父类，所以象上例那样，你可以用父类来指代子类。但是这个 OnetoOneField 字段默认的 related_name 值与 ForeignKey 和 ManyToManyField 默认的反向名称相同。如果你与其他 model 的子类做多对一或是多对多关系，你就必须在每个多对一和多对多字段上强制指定 related_name。如果你没这么做，Django 就会在你运行 验证(validation) 时抛出异常。

例如，仍以上面 Place类为例，我们创建一个带有 ManyToManyField字段的子类：

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

这会产生一个错误：

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

像下面那样，向customers字段中添加related_name可以解决这个错误：models.ManyToManyField(Place, related_name='provider')。

指定链接父类的字段

之前我们提到，Django 会自动创建一个 OneToOneField字段将子类链接至非抽象的父 model 。如果你想指定链接父类的属性名称，你可以创建你自己的 OneToOneField字段并设置 parent_link=True，从而使用该字段链接父类。

代理模型

使用 多表继承时，model 的每个子类都会创建一张新数据表，通常情况下，这正是我们想要的操作。这是因为子类需要一个空间来存储不包含在基类中的字段数据。 但有时，你可能只想更改 model 在 Python 层的行为实现。比如：更改默认的 manager ，或是添加一个新方法。

而这，正是代理 model 继承方式要做的：为原始 model 创建一个代理 。你可以创建，删除，更新代理 model 的实例，而且所有的数据都可以像使用原始 model 一样被保存。 不同之处在于：你可以在代理 model 中改变默认的排序设置和默认的 manager ， 更不会对原始 model 产生影响。

声明代理 model 和声明普通 model 没有什么不同。 设置Meta类中 proxy 的值为 True，就完成了对代理 model 的声明。

举个例子，假设你想给 Person 模型添加一个方法。你可以这样做：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

MyPerson类和它的父类 Person 操作同一个数据表。特别的是，Person 的任何实例也可以通过 MyPerson访问，反之亦然：

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

你也可以使用代理 model 给 model 定义不同的默认排序设置。 你可能并不想每次都给Person模型排序，但是使用代理的时候总是按照last_name属性排序。这非常容易：

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

现在，普通的Person查询时无序的，而 OrderedPerson查询会按照last_name排序。

查询集始终返回请求的模型

也就是说，没有办法让Django在查询Person对象时返回MyPerson对象。Person 对象的查询集会返回相同类型的对象。代理对象的要点是：它会使用依赖于原生Person的代码，而你可以使用你添加进来的扩展对象（它不会依赖其它任何代码）。而并不是将Person模型（或者其它）在所有地方替换为其它你自己创建的模型。

基类的限制

代理 模型必须继承自一个非抽象基类。 你不能继承自多个非抽象基类，这是因为一个代理 model 不能连接不同的数据表。 代理 model 也可以继承任意多个抽象基类，但前提是它们没有 定义任何 model 字段。

代理模型的管理器

如果你没有在代理 模型中定义任何 管理器，代理模型就会从父类中继承 管理器 。 如果你在代理 模型中定义了一个 管理器，它就会变成默认的管理器，不过定义在父类中的管理器仍然有效。

继续上面的例子，当你查询Person模型的时候，你可以改变默认 管理器，例如：

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass

class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

如果你想要向代理中添加新的管理器，而不是替换现有的默认管理器，你可以使用自定义管理器管理器文档中描述的技巧：创建一个含有新的管理器的基类，并继承时把他放在主基类的后面：

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

你可能不需要经常这样做，但这样做是可行的。

代理 模型与非托管 模型之间的差异

代理 model 继承看上去和使用Meta类中的 managed 属性的非托管 model 非常相似。但两者并不相同，使用时选用哪种方案是一个值得考虑的问题

一个不同之处是你可以在Meta.managed=False的 model 中定义字段(事实上，是必须指定，除非你真的想得到一个空 model)。在创建非托管 model 时要谨慎设置Meta.db_table，这是因为创建的非托管 model 映射某个已存在的 model，并且有自己的方法。因此，如果你要保证这两个 model 同步并对程序进行改动，那么就会变得繁冗而脆弱。

另一个不同之处是两者对 管理器的处理方式不同。 代理 model 要与它所代理的 model 行为相似，所以代理 model 要继承父 model 的 managers，包括它的默认 manager 。 但在普通的多表继承中，子类不能继承父类的 manager，这是因为在处理非基类字段时，父类的 manager 未必适用。 后一种情况在 管理器文档有详细介绍。

我们实现了这两种特性之后，曾尝试把两者结合到一起。 结果证明，宏观的继承关系和微观的 管理器揉在一起，不仅导致 API 复杂难用，而且还难以理解。 由于任何场合下都可能需要这两个选项，所以目前二者仍是各自独立使用的。

所以，一般规则是：

1. 如果你要借鉴一个已有的 模型或数据表，且不想涉及所有的原始数据表的列，那就令 `Meta.managed=False`。通常情况下，对模型数据库创建视图和表格不需要由 Django 控制时，就使用这个选项。
2. 如果你想对 model 做 Python 层级的改动，又想保留字段不变，那就令 `Meta.proxy=True`。因此在数据保存时，代理 model 相当于完全复制了原始 模型的存储结构。

多重继承

就像Python的子类那样，Django的模型可以继承自多个父类模型。切记一般的Python名称解析规则也会适用。出现特定名称的第一个基类(比如Meta)是所使用的那个。例如，这意味着如果多个父类含有 Meta类，只有第一个会被使用，剩下的会忽略掉。

一般来说，你并不需要继承多个父类。多重继承主要对“mix-in”类有用：向每个继承mix-in的类添加一个特定的、额外的字段或者方法。你应该尝试将你的继承关系保持得尽可能简洁和直接，这样你就不必费很大力气来弄清楚某段特定的信息来自哪里。

> Changed in Django 1.7.

Django 1.7之前，继承多个含有id主键字段的模型不会抛出异常，但是会导致数据丢失。例如，考虑这些模型（由于id字段的冲突，它们不再有效）：

```
class Article(models.Model):
    headline = models.CharField(max_length=50)
    body = models.TextField()

class Book(models.Model):
    title = models.CharField(max_length=50)

class BookReview(Book, Article):
    pass
```

这段代码展示了如何创建子类的对象，并覆写之前创建的父类对象中的值。

```
>>> article = Article.objects.create(headline='Some piece of news.')
>>> review = BookReview.objects.create(
...     headline='Review of Little Red Riding Hood.',
...     title='Little Red Riding Hood')
>>>
>>> assert Article.objects.get(pk=article.pk).headline == article.headline
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError
>>> # the "Some piece of news." headline has been overwritten.
>>> Article.objects.get(pk=article.pk).headline
'Review of Little Red Riding Hood.'
```

你可以在模型基类中使用显式的AutoField来合理使用多重继承：

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

或者是使用一个父类来持有AutoField：

```
class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
    ...

class BookReview(Book, Article):
    pass
```

Field name "hiding" is not permitted

普通的 Python 类继承允许子类覆盖父类的任何属性。 但在 Django 中，重写 Field实例是不允许的(至少现在还不行)。如果基类中有一个 author字段，你就不能在子类中创建任何名为 author的字段。

重写父类的字段会导致很多麻烦，比如：初始化实例(指定在 Model.__init__ 中被实例化的字段) 和序列化。而普通的 Python 类继承机制并不能处理好这些特性。所以 Django 的继承机制被设计成与 Python 有所不同，这样做并不是随意而为的。

这些限制仅仅针对做为属性使用的 Field实例，并不是针对 Python 属性，Python 属性仍是可以被重写的。 在 Python 看来，上面的限制仅仅针对字段实例的名称：如果你手动指定了数据库的列名称，那么在多重继承中，你就可以在子类和某个祖先类当中使用同一个列名称。(因为它们使用的是两个不同数据表的字段)。

如果你在任何一个祖先类中重写了某个 model 字段，Django 都会抛出 FieldError异常。

另见

模型参考

涵盖模型相关的API，包括模型字段、关联对象和**查询集**。

« previous | up | ne