### 1、数据库的连接方式以及设置：

在Django中默认使用的数据库类型是sqlite3，如果想要使用其他数据库就需要在settings中设置数据库的连接方式：

# Database

# https://docs.djangoproject.com/en/1.10/ref/settings/#databases

# sqlite3数据库连接方式

# DATABASES = {

# ‘default‘: {

# ‘ENGINE‘: ‘django.db.backends.sqlite3‘,

# ‘NAME‘: os.path.join(BASE\_DIR, ‘db.sqlite3‘),

# }

# }

# MySQL数据库连接方式

DATABASES = {

‘default‘: {

‘ENGINE‘: ‘django.db.backends.mysql‘,

‘NAME‘:‘dbname‘,

‘USER‘: ‘root‘,

‘PASSWORD‘: ‘xxx‘,

‘HOST‘: ‘‘,

‘PORT‘: ‘‘,

}

}

### 2、开始创建表

数据需要在models.py文件中创建

class UserInfo(models.Model):

# CharField类型不能为空,最少要指定一个长度

user = models.CharField(max\_length=32)

email = models.EmailField(max\_length=32)

pwd = models.CharField(max\_length=32)

user\_type = models.ForeignKey(‘UserType‘)

class UserType(models.Model):

nid = models.AutoField(primary\_key=True)

caption = models.CharField(max\_length=16)

注:在创建外键的时候直接写上UserType和‘UserType‘的区别就是python程序从上到下解释的顺序问题,如果把UserType这个类写到下面就会没事了

运行Djando项目程序，执行命令创建数据：

python manage.py makemigrations

python manage.py migrate

#### 创建表的的参数：

1、models.AutoField　　自增列 = int(11)

　　如果没有的话，默认会生成一个名称为 id 的列，如果要显示的自定义一个自增列，必须将给列设置为主键 primary\_key=True。

2、models.CharField　　字符串字段

　　必须 max\_length 参数

3、models.BooleanField　　布尔类型=tinyint(1)

　　不能为空，Blank=True

4、models.ComaSeparatedIntegerField　　用逗号分割的数字=varchar

　　继承CharField，所以必须 max\_lenght 参数

5、models.DateField　　日期类型 date

　　对于参数，auto\_now = True 则每次更新都会更新这个时间；auto\_now\_add 则只是第一次创建添加，之后的更新不再改变。

6、models.DateTimeField　　日期类型 datetime

　　同DateField的参数

7、models.Decimal　　十进制小数类型 = decimal

　　必须指定整数位max\_digits和小数位decimal\_places

8、models.EmailField　　字符串类型（正则表达式邮箱） =varchar

　　对字符串进行正则表达式

9、models.FloatField　　浮点类型 = double

10、models.IntegerField　　整形

11、models.BigIntegerField　　长整形

　　integer\_field\_ranges = {

　　　　‘SmallIntegerField‘: (-32768, 32767),

　　　　‘IntegerField‘: (-2147483648, 2147483647),

　　　　‘BigIntegerField‘: (-9223372036854775808, 9223372036854775807),

　　　　‘PositiveSmallIntegerField‘: (0, 32767),

　　　　‘PositiveIntegerField‘: (0, 2147483647),

　　}

12、models.IPAddressField　　字符串类型（ip4正则表达式）

13、models.GenericIPAddressField　　字符串类型（ip4和ip6是可选的）

　　参数protocol可以是：both、ipv4、ipv6

　　验证时，会根据设置报错

14、models.NullBooleanField　　允许为空的布尔类型

15、models.PositiveIntegerFiel　　正Integer

16、models.PositiveSmallIntegerField　　正smallInteger

17、models.SlugField　　减号、下划线、字母、数字

18、models.SmallIntegerField　　数字

　　数据库中的字段有：tinyint、smallint、int、bigint

19、models.TextField　　字符串=longtext

20、models.TimeField　　时间 HH:MM[:ss[.uuuuuu]]

21、models.URLField　　字符串，地址正则表达式

22、models.BinaryField　　二进制

23、models.ImageField 图片

24、models.FilePathField 文件

#### 更多字段属性

1、null=True

　　数据库中字段是否可以为空

2、blank=True

　　django的 Admin 中添加数据时是否可允许空值

3、primary\_key = False

　　主键，对AutoField设置主键后，就会代替原来的自增 id 列

4、auto\_now 和 auto\_now\_add

　　auto\_now 自动创建---无论添加或修改，都是当前操作的时间

　　auto\_now\_add 自动创建---永远是创建时的时间

5、choices

GENDER\_CHOICE = (

(u‘M‘, u‘Male‘),

(u‘F‘, u‘Female‘),

)

gender = models.CharField(max\_length=2,choices = GENDER\_CHOICE)

6、max\_length

7、default　　默认值

8、verbose\_name　　Admin中字段的显示名称

9、name|db\_column　　数据库中的字段名称

10、unique=True　　不允许重复

11、db\_index = True　　数据库索引

12、editable=True　　在Admin里是否可编辑

13、error\_messages=None　　错误提示

14、auto\_created=False　　自动创建

15、help\_text　　在Admin中提示帮助信息

16、validators=[]

17、upload-to

### 3.开始创建数据

创建数据的时候有两种方式：

#### 第一种方式：

obj = models.表名(字段名=‘\*\*\*‘)

obj.save()

#### 第二种方式：

models.表名.objects.create(字段名=‘\*\*\*‘)

在views.py中写入数据：

from django.shortcuts import render,HttpResponse

from app01 import models

# Create your views here.

def index(request):

# 创建用户类型表

models.UserType.objects.create(caption=‘管理员‘)

models.UserType.objects.create(caption=‘普通用户‘)

models.UserType.objects.create(caption=‘超级管理员‘)

# 创建用户信息表

user\_info\_dict\_1 = {‘user‘: ‘ales‘,

‘email‘: ‘alex@qq.com‘,

‘pwd‘: 123,

‘user\_type‘: models.UserType.objects.get(nid=1),

}

user\_info\_dict\_2 = {‘user‘: ‘eric‘,

‘email‘: ‘eric@qq.com‘,

‘pwd‘: 123,

‘user\_type\_id‘: 2,

}

models.UserInfo.objects.create(\*\*user\_info\_dict\_1)

models.UserInfo.objects.create(\*\*user\_info\_dict\_2)

print(‘yes‘)

return HttpResponse(‘ok‘)

### 4、了不起的双下划线之外键正向查找和基本操作

#### 基本操作

**# 增**

#

# models.Tb1.objects.create(c1='xx', c2='oo') 增加一条数据，可以接受字典类型数据 \*\*kwargs

# obj = models.Tb1(c1='xx', c2='oo')

# obj.save()

**# 查**

# models.Tb1.objects.get(id=123) # 获取单条数据，不存在则报错（不建议）

# models.Tb1.objects.all() # 获取全部

# models.Tb1.objects.filter(name='seven') # 获取指定条件的数据

**# 删**

# models.Tb1.objects.filter(name='seven').delete() # 删除指定条件的数据

**# 改**

# models.Tb1.objects.filter(name='seven').update(gender='0') # 将指定条件的数据更新，均支持 \*\*kwargs

# obj = models.Tb1.objects.get(id=1)

# obj.c1 = '111'

# obj.save() # 修改单条数据

#### 进阶操作（双下划线）

**# 获取个数**

# models.Tb1.objects.filter(name='seven').count()

**# 大于，小于**

# models.Tb1.objects.filter(id\_\_gt=1) # 获取id大于1的值

# models.Tb1.objects.filter(id\_\_lt=10) # 获取id小于10的值

# models.Tb1.objects.filter(id\_\_lt=10, id\_\_gt=1) # 获取id大于1 且 小于10的值

**# in**

# models.Tb1.objects.filter(id\_\_in=[11, 22, 33]) # 获取id等于11、22、33的数据

# models.Tb1.objects.exclude(id\_\_in=[11, 22, 33]) # not in

**# contains**

# models.Tb1.objects.filter(name\_\_contains="ven")

# models.Tb1.objects.filter(name\_\_icontains="ven") # icontains大小写不敏感

# models.Tb1.objects.exclude(name\_\_icontains="ven")

**# range**

# models.Tb1.objects.filter(id\_\_range=[1, 2]) # 范围bettwen and

**# 其他类似**

# startswith，istartswith, endswith, iendswith,

**# order by**

# models.Tb1.objects.filter(name='seven').order\_by('id') # asc

# models.Tb1.objects.filter(name='seven').order\_by('-id') # desc

**# limit 、offset**

# models.Tb1.objects.all()[10:20]

**# group by**

from django.db.models import Count, Min, Max, Sum

# models.Tb1.objects.filter(c1=1).values('id').annotate(c=Count('num'))

# SELECT "app01\_tb1"."id", COUNT("app01\_tb1"."num") AS "c" FROM "app01\_tb1" WHERE "app01\_tb1"."c1" = 1 GROUP BY "app01\_tb1"."id"

#### 查

##### 单表查询：

# ret = models.UserType.objects.all()

# print(ret.query)

# ret.query后台返回的是查询的sql语句

结果：

SELECT `app01\_usertype`.`nid`, `app01\_usertype`.`caption` FROM `app01\_usertype`

获取查询结果的类型：

ret = models.UserType.objects.all()

print(type(ret), ret)

结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [<UserType: UserType object>, <UserType: UserType object>, <UserType: UserType object>]>

　可以看到类型是一个QuerySet类型，后面是所有的对象，每一个元素就是一个对象，可以循环拿出每一次的数据：

ret = models.UserType.objects.all()

print(type(ret), ret)

for item in ret:

print(item)

　其结果就是每一次循环出来的结果的对象：

UserType object

UserType object

UserType object

　　每一个对象都代表一个数据，要出去这些数据如下：

ret = models.UserType.objects.all()

print(type(ret), ret)

for item in ret:

print(item, item.nid, item.caption)

　　取出的结果：

UserType object 1 管路员

UserType object 2 普通用户

UserType object 3 超级管理员

从结果看出每次输出item的时候都是一个对象(一行数据中所有的对象，对象中封装了所有的数据)，在modes中有\_\_str\_\_方法（返回什么，就输出什么，就是查看方便）， 在python2.7中叫\_\_unicode\_\_如果在UserType这个类里面使用这个方法：

class UserType(models.Model):

nid = models.AutoField(primary\_key=True)

caption = models.CharField(max\_length=16)

def \_\_str\_\_(self):

return ‘%s-%s‘ % (self.nid, self.caption)

　　然后重新访问下：

1-管路员 1 管路员

2-普通用户 2 普通用户

3-超级管理员 3 超级管理员

　　就可以看到每一个对象都看到了返回的相对应的参数了。

**查询单个字段：**

ret = models.UserType.objects.all().values(‘nid‘)

print(type(ret), ret)

结果查询出nid字段对应的所有的数据 ：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [{‘nid‘: 1}, {‘nid‘: 2}, {‘nid‘: 3}]>

可以看查询的sql语句，用query方法：

ret = models.UserType.objects.all().values(‘nid‘)

print(type(ret), ret.query)

查询的结果：

<class ‘django.db.models.query.QuerySet‘> SELECT `app01\_usertype`.`nid` FROM `app01\_usertype`

当通过values循环取值的时候，如下：

ret = models.UserType.objects.all().values(‘nid‘)

print(type(ret), ret.query)

for item in ret:

print(item, type(item))

　　结果：

<class ‘django.db.models.query.QuerySet‘> SELECT `app01\_usertype`.`nid` FROM `app01\_usertype`

{‘nid‘: 1} <class ‘dict‘>

{‘nid‘: 2} <class ‘dict‘>

{‘nid‘: 3} <class ‘dict‘>

　　通过结果可以看出，最外部是QuerySet，内部元素封装了一个是封装了这一行所有数据的对象，另外只拿到了某几列的字典！

当通过values\_list循环取值的时候，如下：

ret = models.UserType.objects.all().values\_list(‘nid‘)

print(type(ret), ret)

　　查询结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [(1,), (2,), (3,)]>

　　依然是queryset，但是结果就是列表中包含的元组，values和values\_list的区别就是：values取的是字典类型，values\_list把内部元素变成元组了。通过for循环更直观，如下：

ret = models.UserType.objects.all().values\_list(‘nid‘)

print(type(ret), ret)

for item in ret:

print(type(item), item)

　　结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [(1,), (2,), (3,)]>

<class ‘tuple‘> (1,)

<class ‘tuple‘> (2,)

<class ‘tuple‘> (3,)

##### 连表查询：

连表查询的时候和sqlachemy有多不同，django没有join这个方法：

　　通过UserInfo做连表查询

ret = models.UserInfo.objects.all()

print(type(ret), ret)

for item in ret:

print(item, item.user\_type, item.id, item.email, item.user, item.pwd)

　　查询结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [<UserInfo: UserInfo object>, <UserInfo: UserInfo object>]>

UserInfo object UserType object 1 alex@qq.com ales 123

UserInfo object UserType object 2 eric@qq.com eric 123

　　首先输出的是查找出来的UserInfo用户信息表，封装了用户所有的信息，而UserType就是一个对象，这个对象里封装的就是一行信息对应的字段(nid和caption)，取出UserType对应的信息，例如：

ret = models.UserInfo.objects.all()

print(type(ret), ret)

for item in ret:

print(item, item.user\_type, item.id, item.email, item.user, item.pwd)

print(item.user\_type.nid, item.user\_type.caption)

　　取出的UserType对应的信息结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [<UserInfo: UserInfo object>, <UserInfo: UserInfo object>]>

UserInfo object UserType object 1 alex@qq.com ales 123

UserInfo object UserType object 2 eric@qq.com eric 123

1 管路员

2 普通用户

　　可以直接取出对应外键的那一列user\_type\_id：

ret = models.UserInfo.objects.all()

print(type(ret), ret)

for item in ret:

print(item, item.user\_type, item.id, item.email, item.user, item.pwd)

print(item.user\_type.nid, item.user\_type.caption, item.user\_type\_id)

　　结果：

<class ‘django.db.models.query.QuerySet‘> <QuerySet [<UserInfo: UserInfo object>, <UserInfo: UserInfo object>]>

UserInfo object UserType object 1 alex@qq.com ales 123

1 管路员 1

UserInfo object UserType object 2 eric@qq.com eric 123

2 普通用户 2

　　如果只想看到数据对应的名称，就需要加映射values：

ret = models.UserInfo.objects.all().values(‘user‘, ‘user\_type\_\_caption‘)

print(ret, ret.query)

　　结果：

<QuerySet [{‘user‘: ‘ales‘, ‘user\_type\_\_caption‘: ‘管路员‘}, {‘user‘: ‘eric‘, ‘user\_type\_\_caption‘: ‘普通用户‘}]> SELECT `app01\_userinfo`.`user`, `app01\_usertype`.`caption` FROM `app01\_userinfo` INNER JOIN `app01\_usertype` ON (`app01\_userinfo`.`user\_type\_id` = `app01\_usertype`.`nid`)

　　通过结果可以看到user\_type\_\_caption可以跨表直接获取相应的结果，再看sql语句中django本身加上了一个join。

　　如果在models中在加上一个表"P",然后在UserType中增加一条外键指向"平"P"，需求：要查找UserInfo表中的name和UserType表中的caption和P表中的neme的方法就是在找下一个对象的时候继续加双下划线：

##### 增加的表结构：



　　查询语法：

ret = models.UserInfo.objects.all().values(‘user‘, ‘user\_type\_\_caption‘, ‘user\_type\_\_p\_\_name‘)

print(ret, ret.query)

　　查询的结果：

<QuerySet [{‘user\_type\_\_p\_\_name‘: ‘allan‘, ‘user\_type\_\_caption‘: ‘管路员‘, ‘user‘: ‘ales‘}, {‘user\_type\_\_p\_\_name‘: ‘allan‘, ‘user\_type\_\_caption‘: ‘普通用户‘, ‘user‘: ‘eric‘}]> SELECT `app01\_userinfo`.`user`, `app01\_usertype`.`caption`, `app01\_p`.`name` FROM `app01\_userinfo` INNER JOIN `app01\_usertype` ON (`app01\_userinfo`.`user\_type\_id` = `app01\_usertype`.`nid`) INNER JOIN `app01\_p` ON (`app01\_usertype`.`p\_id` = `app01\_p`.`id`)

　　如果要拿用户类型是管理员的所有用户：

ret = models.UserInfo.objects.filter(user\_type\_\_caption="管路员").values(‘user‘, ‘user\_type\_\_caption‘)

print(ret)

　　结果：

<QuerySet [{‘user‘: ‘ales‘, ‘user\_type\_\_caption‘: ‘管路员‘}]>

### 5、了不起的上下划线值外键反向查找

　　找到管理员相关联的信息：

obj = models.UserType.objects.filter(caption=‘管理员‘).first()

print(obj.nid, obj.caption)

　　输出结果：

1 管理员

　　这里查找的是usertype中的数据，如果通过usertype表查找到userinfo表中的信息就得用\*\*\*\_set:

obj = models.UserType.objects.filter(caption=‘管理员‘).first()

print(obj.nid, obj.caption)

print(obj.userinfo\_set.all())

ret = models.UserType.objects.all().values(‘nid‘, ‘caption‘, ‘userinfo\_\_user‘)

print(ret)

　　查询结果：

1 管理员

<QuerySet [<UserInfo: UserInfo object>]>

<QuerySet [{‘caption‘: ‘管理员‘, ‘userinfo\_\_user‘: ‘alex‘, ‘nid‘: 1}, {‘caption‘: ‘普通用户‘, ‘userinfo\_\_user‘: ‘eric‘, ‘nid‘: 2}, {‘caption‘: ‘超级管理员‘, ‘userinfo\_\_user‘: None, ‘nid‘: 3}]>

[27/Aug/2016 15:35:31] "GET /index/ HTTP/1.1" 200 2

　　如果需要固定字段查找：

obj = models.UserType.objects.all().values(‘nid‘, ‘userinfo\_\_user‘)

print(obj)

　　结果：

<QuerySet [{‘userinfo\_\_user‘: ‘alex‘, ‘nid‘: 1}, {‘userinfo\_\_user‘: ‘eric‘, ‘nid‘: 2}, {‘userinfo\_\_user‘: None, ‘nid‘: 3}]>

### 6、多对多之表创建

多对多创建表的时候直接使用ManyToManyField让django自动创建第三张表或者自己手动创建第三张表，第三张表如果指定多个字段的hu

models表结构(自定义的第三张表)：

class Host(models.Model):

hid = models.AutoField(primary\_key=True)

hostname = models.CharField(max\_length=32)

ip = models.CharField(max\_length=32)

class Group(models.Model):

gid = models.AutoField(primary\_key=True)

name = models.CharField(max\_length=16)

h2g = models.ManyToManyField(‘Host‘, through=‘HostToGroup‘)

class HostToGroup(models.Model):

hgid = models.AutoField(primary\_key=True)

host\_id = models.ForeignKey(‘Host‘)

group\_id = models.ForeignKey(‘Group‘)

status = models.IntegerField()

#### 插入数据：

models.Host.objects.create(hostname=‘c1‘, ip=‘1.1.1.1‘)

models.Host.objects.create(hostname=‘c2‘, ip=‘1.1.1.2‘)

models.Host.objects.create(hostname=‘c3‘, ip=‘1.1.1.3‘)

models.Host.objects.create(hostname=‘c4‘, ip=‘1.1.1.4‘)

models.Host.objects.create(hostname=‘c5‘, ip=‘1.1.1.5‘)

models.Group.objects.create(name=‘技术部‘)

models.Group.objects.create(name=‘财务部‘)

models.Group.objects.create(name=‘人事部‘)

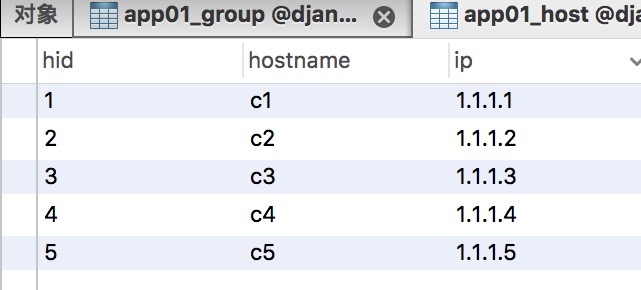
models.Group.objects.create(name=‘公关部‘)

models.Group.objects.create(name=‘运营部‘)

models.Group.objects.create(name=‘销售部‘)

models.Group.objects.create(name=‘客服部‘)





#### 操作表：

获取财务部的对象：

obj = models.Group.objects.get(gid=2)

print(obj.gid, obj.name, obj.h2g.all())

结果：

财务部 <QuerySet []>

给财务部添加主机：

# 添加一台主机

obj = models.Group.objects.get(gid=2)

# print(obj.gid, obj.name, obj.h2g.all())

h1 = models.Host.objects.get(hid=1)

obj.h2g.add(h1)

http://images2015.cnblogs.com/blog/866886/201608/866886-20160828214913059-1731083158.png

# 把剩下的全部添加

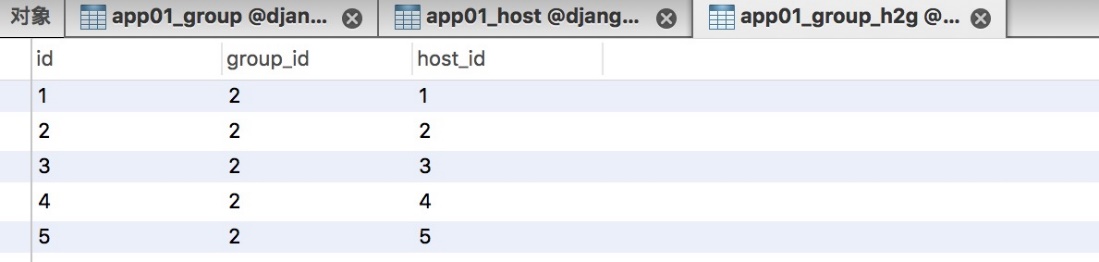
obj = models.Group.objects.get(gid=2)

# print(obj.gid, obj.name, obj.h2g.all())

# h1 = models.Host.objects.get(hid=1)

q = models.Host.objects.filter(hid\_\_gt=1)

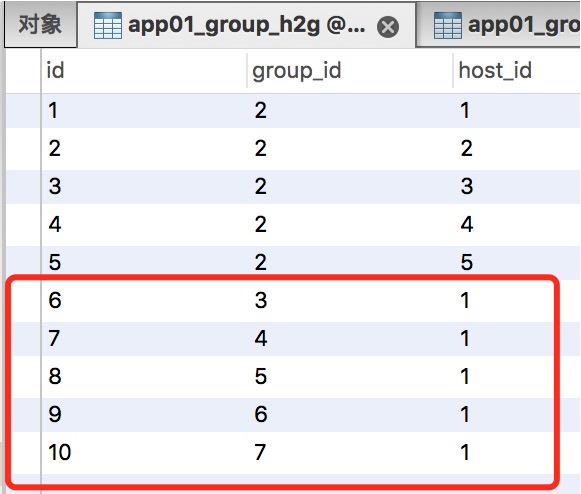
obj.h2g.add(\*q)



将一台机器分配给多个组(就得用反向操作了)：

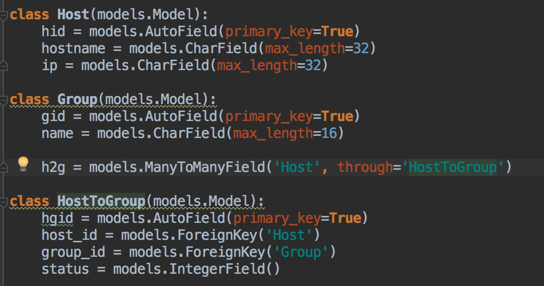
h = models.Host.objects.get(hid=1)

h.group\_set.add(\*models.Group.objects.filter(gid\_\_gt=2))



操作自己手动创建的第三张关系表:

　　表结构：



表的数据还是原来的数据。

　　创建表之间的关系，需要自己手动创建关系：

models.HostToGroup.objects.create(status=1, group\_id\_id=2, host\_id\_id=3)

