ORM

ORM，对象关系映射，对象和关系之间的映射，使用面向对象的方式来操作数据库。

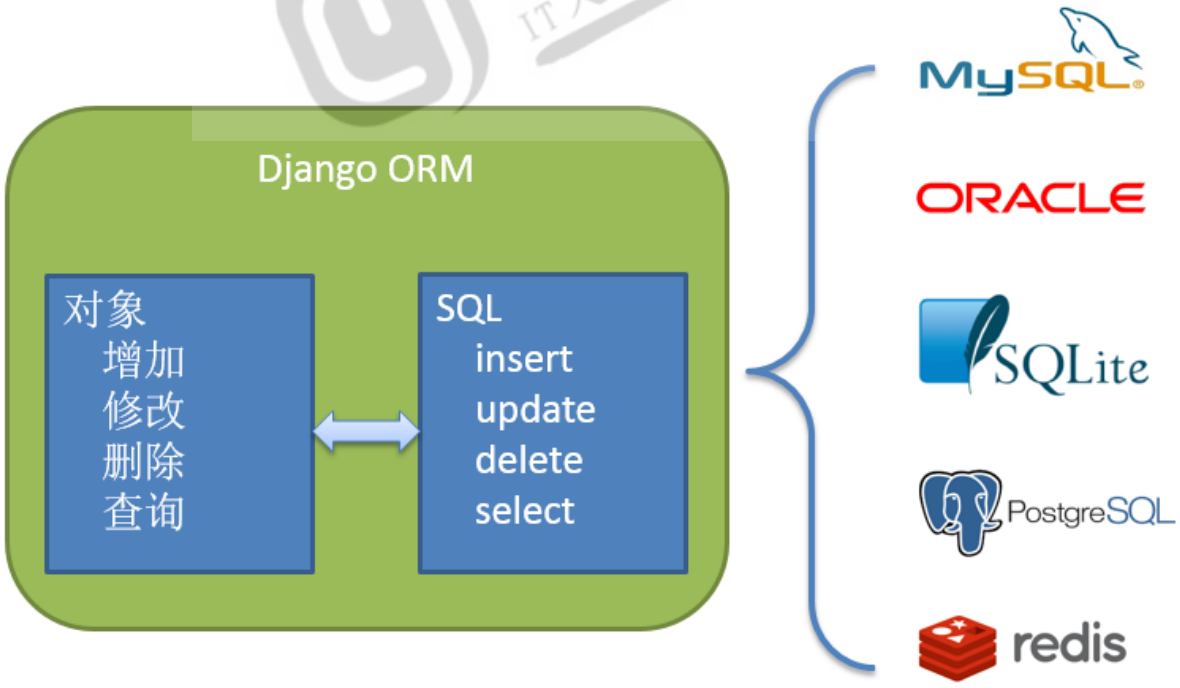
```
关系模型和Python对象之间的映射
table  => class     ，表映射为类
row    => object    ，行映射为实例
column => property  ，字段映射为属性
```

举例，有表student，字段为id int，name varchar，age int
映射到Python为

```
class Student:
    id = ?某类型字段
    name = ?某类型字段
    age = ?某类型字段

最终得到实例
class Student:
    def __init__(self):
        self.id = ?
        self.name = ?
        self.age = ?
```

Django ORM



对模型对象的CRUD，被Django ORM转换成相应的SQL语句以操作不同的数据源。

安装

2.2.15为LTS版本。

```
$ pip install django==2.2.15
```

项目准备

```
$ django-admin startproject salary .
```

创建应用

```
$ python manage.py startapp employee
```

配置

打开salary/settings.py主配置文件

- 修改数据库配置
- 修改时区
- 注册应用

```
# 数据库配置
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'test',
        'USER': 'wayne',
        'PASSWORD': 'wayne',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}

# 时区
TIME_ZONE = 'Asia/Shanghai'

# 注册应用
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'employee',
]
```

Django日志

Django的日志配置在settings.py中。

```
LOGGING = {
    'version': 1,
```

```
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
    },
}
```

配置后，就可以在控制台看到执行的SQL语句。

注意，settings.py中必须**DEBUG=True**，同时loggers的**level是DEBUG**，否则从控制台看不到SQL语句。

Django内建loggers可以参考https://docs.djangoproject.com/en/2.2/topics/logging/#django-db-backends

模型Model

字段类型

字段类	说明
AutoField	自增的整数字段。 如果不指定，django会为模型类自动增加主键字段
BooleanField	布尔值字段，True和False 对应表单控件CheckboxInput
NullBooleanField	比BooleanField多一个null值
CharField	字符串，max_length设定字符长度 对应表单控件TextInput
TextField	大文本字段，一般超过4000个字符使用 对应表单控件Textarea
IntegerField	整数字段
BigIntegerField	更大整数字段，8字节
DecimalField	使用Python的Decimal实例表示十进制浮点数。max_digits总位数，decimal_places小数点后的位数
FloatField	Python的Float实例表示的浮点数
DateField	使用Python的datetime.date实例表示的日期 auto_now=False每次修改对象自动设置为当前时间。 auto_now_add=False对象第一次创建时自动设置为当前时间。 auto_now_add、auto_now、default互斥 对应控件为TextInput，关联了一个js编写的日历控件
TimeField	使用Python的datetime.time实例表示的时间，参数同上
DateTimeField	使用Python的datetime.datetime实例表示的时间，参数同上
FileField	一个上传文件的字段
ImageField	继承了FileField的所有属性和方法，但是对上传的文件进行校验，确保是一个有效的图片
EmailField	能做Email检验，基于CharField，默认max_length=254
GenericIPAddressField	支持IPv4、IPv6检验，缺省对应文本框输入
URLField	能做URL检验，基于基于CharField，默认max_length=200

缺省主键

缺省情况下，Django的每一个Model都有一个名为id的AutoField字段，如下

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

如果显式定义了主键，这种缺省主键就不会被创建了。Python之禅中说“显式优于隐式”，所以，如果有必要，还是尽量使用自己定义的主键，哪怕该字段名就是id，也是一种不错的选择。

字段选项

参考 https://docs.djangoproject.com/en/1.11/ref/models/fields/#field-options

值	说明
db_column	表中字段的名称。如果未指定，则使用属性名
primary_key	是否主键
unique	是否是唯一键
default	缺省值。这个缺省值不是数据库字段的缺省值，而是新对象产生的时候被填入的缺省值
null	表的字段是否可为null，默认为False
blank	Django表单验证中，是否可以不填写，默认为False
db_index	字段是否有索引

关系类型字段类

类	说明
ForeignKey	外键，表示一对多 ForeignKey('production.Manufacturer') 自关联ForeignKey('self')
ManyToManyField	表示多对多
OneToOneField	表示一对一

Model类

- 基类 django.db.models.Model
- 表名不指定默认使用 `<appname>_<model_name>`。使用Meta类db_table修改表名

```
from django.db import models

class Employee(models.Model):
    class Meta:
        db_table = 'employees'
    emp_no = models.IntegerField(primary_key=True)
    birth_date = models.DateField(null=False)
    first_name = models.CharField(null=False, max_length=14)
    last_name = models.CharField(null=False, max_length=16)
    gender = models.SmallIntegerField(null=False)
    hire_date = models.DateField(null=False)

    def __repr__(self):
        return "<Employee: {} {} {}>".format(self.emp_no, self.first_name,
self.last_name)

    __str__ = __repr__
```

在项目根目录编写一个test.py，内容如下

```
import os
import django

# 参考wsgi.py
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'salary.settings')
django.setup()

# 访问数据，一定要写在上面四行之后
from employee.models import Employee

emps = Employee.objects.all() # 结果集，本句不发起查询
print(type(emps))
print(*list(emps), sep='\n') # 所有员工
```

管理器对象

Django会为模型类提供一个**objects对象**，它是django.db.models.manager.Manager类型，用于与数据库交互。当定义模型类的时候没有指定管理器，则Django会为模型类提供一个objects的管理器。
如果在模型类中手动指定管理器后，Django不再提供默认的objects的管理器了。

管理器是Django的模型进行数据库**查询**操作的接口，Django应用的每个模型都至少拥有一个管理器。

用户也可以自定义管理器类，继承自django.db.models.manager.Manager，实现表级别控制。

查询

查询集

查询会返回结果的集，它是django.db.models.query.QuerySet类型。
它是惰性求值，和sqlalchemy一样。结果就是查询的集。
它是可迭代对象。

1、惰性求值：
创建查询集不会带来任何数据库的访问，直到调用方法使用数据时，才会访问数据库。在迭代、序列化、if语句中都会立即求值。

2、缓存：
每一个查询集都包含一个缓存，来最小化对数据库的访问。
新建查询集，缓存为空。首次对查询集求值时，会发生数据库查询，Django会把查询的结果存在这个缓存中，并返回请求的结果，接下来对查询集求值将使用缓存的结果。

观察下面的2个例子是要看真正生成的语句了
1) 没有使用缓存，每次都要去查库，查了2次库

```
[user.name for user in User.objects.all()]
[user.name for user in User.objects.all()]
```

2) 下面的语句使用缓存，因为使用同一个结果集

```
qs = User.objects.all()
[user.name for user in qs]
[user.name for user in qs]
```

限制查询集（切片）

分页功能实现，使用限制查询集。

查询集对象可以直接使用索引下标的方式（不支持负索引），相当于SQL语句中的limit和offset子句。注意使用索引返回的新的结果集，依然是惰性求值，不会立即查询。

```
qs = Employee.objects.all()[10:15]
# LIMIT 5 OFFSET 10
qs = Employee.objects.all()[20:30]
# LIMIT 10 OFFSET 20
```

注：在使用print函数打印结果集的时候，看到SQL语句有自动添加的LIMIT 21，这是怕打印的太长了。使用for循环迭代就没了。

结果集方法

名称	说明
all()	
filter()	过滤，返回满足条件的数据
exclude()	排除，排除满足条件的数据
order_by()	排序，注意参数是字符串
values()	返回一个对象字典的列表，列表的元素是字典，字典内是字段和值的键值对

`filter(k1=v1).filter(k2=v2)` 等价于 `filter(k1=v1, k2=v2)`

`filter(pk=10)` 这里pk指的就是主键， 不用关心主键字段名，当然也可以使用使用主键名 `filter(emp_no=10)`

```
mgr = Employee.objects
print(mgr.all())
print(mgr.values())
print(mgr.filter(pk=10010).values())
print(mgr.exclude(emp_no=10001))
print(mgr.exclude(emp_no=10002).order_by('emp_no'))
print(mgr.exclude(emp_no=10002).order_by('-pk'))
print(mgr.exclude(emp_no=10002).order_by('-pk').values())
```

返回单个值的方法

名称	说明
get()	仅返回单个满足条件的对象 如果未能返回对象则抛出DoesNotExist异常；如果能返回多条，抛出MultipleObjectsReturned异常
count()	返回当前查询的总条数
first()	返回第一个对象
last()	返回最后一个对象
exist()	判断查询集中是否有数据，如果有则返回True

```
mgr = Employee.objects
print(mgr.filter(pk=10010).get())
print(mgr.get(pk=10001))
#print(mgr.exclude(pk=10010).get()) # get严格一个

print(mgr.first()) # limit 1
print(mgr.exclude(pk=10010).last()) # desc, limit 1
print(mgr.filter(pk=10010, gender=1).first()) # AND，找不到返回None
print(mgr.count())
print(mgr.exclude(pk=10010).count())
```

字段查询（Field Lookup）表达式

字段查询表达式可以作为filter()、exclude()、get()的参数，实现where子句。

语法：`属性名称__比较运算符=值`

注意：属性名和运算符之间使用双下划线

比较运算符如下

名称	举例	说明
exact	filter(isdeleted=False) filter(isdeleted__exact=False)	严格等于，可省略不写
contains	exclude(title__contains='天')	是否包含，大小写敏感，等价于`like '%天%'`
startswith endswith	filter(title__startswith='天')	以什么开头或结尾，大小写敏感
isnull isnotnull	filter(title__isnull=False)	是否为null
iexact icontains istartswith iendswith		i的意思是忽略大小写
in	filter(pk__in=[1,2,3,100])	是否在指定范围数据中

gt、gte
lt、lte
	`filter(id__gt=3)`
	`filter(pk__lte=6)`
	`filter(pub_date__gt=date(2000,1,1))`
		大于、小于等

year、month、day week_day hour、minute、second	filter(pub_date__year=2000)	对日期类型属性处理

```
mgr = Employee.objects
print(mgr.filter(emp_no__exact=10010)) # 就是等于，所以很少用exact
print(mgr.filter(pk__in=[10010, 10009]))
print(mgr.filter(last_name__startswith='P'))
print(mgr.exclude(pk__gt=10003))
```

Q对象

虽然Django提供传入条件的方式，但是不方便，它还提供了Q对象来解决。

Q对象是django.db.models.Q，可以使用&、|操作符来组成逻辑表达式。`~`表示not。

```
from django.db.models import Q
mgr = Employee.objects

print(mgr.filter(Q(pk__lt=10006))) # 不如直接写filter(pk__lt=10006)

# 下面几句一样
print(mgr.filter(pk__gt=10003).filter(pk__lt=10006)) # 与
print(mgr.filter(pk__gt=10003, pk__lt=10006)) # 与
print(mgr.filter(pk__gt=10003) & mgr.filter(pk__lt=10006))
print(mgr.filter(Q(pk__gt=10003), Q(pk__lt=10006)))
print(mgr.filter(Q(pk__gt=10003) & Q(pk__lt=10006))) # 与
```

```
# 下面几句等价
print(mgr.filter(pk__in=[10003, 10006])) # in
print(mgr.filter(Q(pk=10003) | Q(pk=10006))) # 或
print(mgr.filter(pk=10003) | mgr.filter(pk=10006))

print(mgr.filter(~Q(pk__gt=10003))) # 非
```

可使用&|和Q对象来构造复杂的逻辑表达式，可以使用一个或多个Q对象。
如果混用关键字参数和Q对象，那么Q对象必须位于关键字参数的前面。

聚合、分组

aggregate() 返回字典，方便使用

```
from employee.models import Employee
from django.db.models import Q, Avg, Sum, Max, Min, Count

mgr = Employee.objects
print(mgr.filter(pk__gt=10010).count()) # 单值
print(mgr.filter(pk__gt=10010).aggregate(Count('pk'), Max('pk'))) # 字典
print(mgr.filter(pk__lte=10010).aggregate(Avg('pk')))
print(mgr.aggregate(Max('pk'), min=Min('pk'))) # 别名
```

annotate()方法用来分组聚合，返回查询集。

```
mgr = Employee.objects
print(mgr.filter(pk__gt=10010).aggregate(Count('pk'))) # 字典
s = mgr.filter(pk__gt=10010).annotate(Count('pk')) # 返回查询集，没指定使用主键分组
for x in s:
    print(x)
    print(x.__dict__) # 里面多了一个属性pk__count
```

values()方法，放在annotate前就是指定分组字段，之后就是取结果中的字段。

```
mgr = Employee.objects
s = mgr.filter(pk__gt=10010).values('gender').annotate(Count('pk')) # 查询集
print(s)
for x in s:
    print(x) # 字典

# 运行结果如下
<QuerySet [{'gender': 2, 'pk__count': 3}, {'gender': 1, 'pk__count': 7}]>
{'gender': 2, 'pk__count': 3}
{'gender': 1, 'pk__count': 7}
```

```
mgr = Employee.objects
s =
mgr.filter(pk__gt=10010).values('gender').annotate(c=Count('pk')).order_by('-c')
# 查询集
print(s)
for x in s:
    print(x) # 字典

# 运行结果如下
<QuerySet [{'gender': 1, 'c': 7}, {'gender': 2, 'c': 3}]>
{'gender': 1, 'c': 7}
{'gender': 2, 'c': 3}
```

```
mgr = Employee.objects
s = mgr.filter(pk__gt=10010).values('gender').annotate(
    Avg('pk'), c=Count('pk')
).order_by('-c').values('pk__avg', 'c') # 查询集,但后面的values过滤了每个对象字典的key
print(s)
for x in s:
    print(x) # 字典

# 运行结果如下
<QuerySet [{'pk__avg': 10015.5714, 'c': 7}, {'pk__avg': 10015.3333, 'c': 3}]>
{'pk__avg': 10015.5714, 'c': 7}
{'pk__avg': 10015.3333, 'c': 3}
```