Django ORM

一对多

员工和工资表是一对多关系

```
CREATE TABLE `employees` (
  `emp_no` int(11) NOT NULL,
  `birth_date` date NOT NULL,
  `first_name` varchar(14) NOT NULL,
  `last_name` varchar(16) NOT NULL,
  `gender` smallint(6) NOT NULL DEFAULT '1' COMMENT 'M=1, F=2',
  `hire_date` date NOT NULL,
  PRIMARY KEY (`emp_no`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;


CREATE TABLE `salaries` (
  `emp_no` int(11) NOT NULL,
  `salary` int(11) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date NOT NULL,
  PRIMARY KEY (`emp_no`,`from_date`),
  CONSTRAINT `salaries_ibfk_1` FOREIGN KEY (`emp_no`) REFERENCES `employees`
(`emp_no`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

联合主键问题

SQLAlchemy提供了联合主键支持，但是Django至今都没有支持。

Django只支持单一主键，这也是我提倡的。但对于本次基于Django测试的表就只能增加一个单一主键了。

原因，请参看 https://code.djangoproject.com/wiki/MultipleColumnPrimaryKeys 。Django 到目前为止也没有提供这种Composite primary key

Django不能直接添加自己的2个字段的联合主键，我们手动为表创建一个自增id主键。操作顺序如下：

1. 取消表所有联合主键，并删除所有外键约束后保存，成功再继续
2. 为表增加一个id字段，自增、主键。保存，如果成功，它会自动填充数据
3. 重建原来的外键约束即可

模型构建

```
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)

    @property
    def name(self):
        return "*{} {}*".format(self.first_name, self.last_name)

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

    __str__ = __repr__

class Salary(models.Model):
    class Meta:
        db_table = 'salaries'
    id = models.AutoField(primary_key=True) # 额外增加的，Django不支持联合主键
    # 候选键(emp_no, from_date)
    emp_no = models.ForeignKey('Employee', on_delete=models.CASCADE, null=False)
    from_date = models.DateField(null=False)
    salary = models.IntegerField(null=False)
    to_date = models.DateField(null=False)

    def __repr__(self):
        return "<Salary: {} {} {}>".format(
            self.emp_no, self.from_date, self.salary)

    __str__ = __repr__


# 测试一下，没有问题再开始
from employee.models import Employee, Salary

mgr = Employee.objects
print(mgr.filter(pk=10004))
print(Salary.objects.all())
```

```
# 测试的时候，使用 print(Salary.objects.all())
# 报错"Unknown column 'salaries.emp_no_id' in 'field list'"
# Django习惯给外键默认起名为xxx_id
# 修改Salary的emp_no，增加db_column来指定字段名称，如下
emp_no = models.ForeignKey('Employee', on_delete=models.CASCADE, null=False,
db_column='emp_no')
```

特殊属性

如果增加了外键后，Django会对一端和多端增加一些新的类属性

```
print(*Employee.__dict__.items(), sep='\n')
# 一端，Employee类中多了一个类属性
# ('salary_set',
<django.db.models.fields.related_descriptors.ReverseManyToOneDescriptor object at
0x000001303FB09B38>)
```

```
print(*Salary.__dict__.items(), sep='\n')
# 多端，Salary类中也多了一个类属性
# ('emp_no_id', <django.db.models.query_utils.DeferredAttribute object at
0x000001303FB09828>)
# ('emp_no',
<django.db.models.fields.related_descriptors.ForwardManyToOneDescriptor object at
0x000001303FB09860>) 指向Employee类的一个实例
```

从一端往多端查`<Employee_instance>.salary_set`

从多端往一端查`<Salary_instance>.emp_no`

查询

```
empmgr = Employee.objects

# 查询10004员工所有工资
# 方案一、从员工往工资查
# print(empmgr.filter(pk=10004).salary_set) # 错误，filter返回查询集，应该是员工对象上
调用xxx_set

print(empmgr.get(pk=10004).salary_set.all())
# SELECT `salaries`.`id`, `salaries`.`emp_no`, `salaries`.`from_date`,
`salaries`.`salary`, `salaries`.`to_date` FROM `salaries` WHERE
`salaries`.`emp_no` = 10004  LIMIT 21; args=(10004,)

## 特别注意查询语句
```

如果觉得salary_set不好用，可以使用**related_name**

```
class Salary(models.Model):
    emp_no = models.ForeignKey('Employee', on_delete=models.CASCADE, null=False,
                               db_column='emp_no', related_name='salaries')

print(empmgr.get(pk=10004).salaries.all())
```

```
empmgr = Employee.objects

# 查询10004员工所有工资
# 方案一、从员工表查
emp = empmgr.get(pk=10004) # 单一员工对象
print(emp.salaries.all())
print(emp.salaries.values('emp_no', 'from_date', 'salary')) # 投影
# 工资大于55000
print(emp.salaries.filter(salary__gt=55000).all())
```

```
# 查询10004员工所有工资及姓名
# 方案二、从工资往员工查
slist = list(salmgr.filter(emp_no=10004))
for s in slist:
    print(s.emp_no.name, s.emp_no_id, s.salary) # s.emp_no会引发填充对象

############## 特别注意 ######################
# 这种查询会导致列表中的n个Salary实例填充其中emp_no属性，会查n此数据库
# 所以，从salaries表往employees表查不合适，虽然可以改进，但是还是别扭，用的少
```

distinct

```
# 所有发了工资的员工
print(salarymgr.values('emp_no').distinct())

# 工资大于55000的所有员工的姓名
emps = salarymgr.filter(salary__gt=55000).values('emp_no').distinct()
print(type(emps))
print(emps)
print(empmgr.filter(emp_no__in=[d.get('emp_no') for d in emps])) # in列表
print(empmgr.filter(emp_no__in=emps)) # in子查询
```

raw的使用

如果查询非常复杂，使用Django不方便，可以直接使用SQL语句

```
# 工资大于55000的所有员工的姓名
empmgr = Employee.objects

sql = """\
SELECT DISTINCT e.emp_no, e.first_name, e.last_name
FROM employees e JOIN salaries s
ON e.emp_no=s.emp_no
WHERE s.salary > 55000
"""

# DISTINCT 需要，结果会去重
emps = empmgr.raw(sql)
print(type(emps)) # RawQuerySet
print(list(emps))
# [<Employee: 10001 Georgi Facello>, <Employee: 10002 Bezalel Simmel>, <Employee:
10004 Chirstian Koblick>]
```

```
# 员工工资记录里超过70000的人的工资和姓名
sql = """\
select e.emp_no, e.first_name, e.last_name, s.salary from employees e join
salaries s
on e.emp_no = s.emp_no
where s.salary > 70000
"""
for x in empmgr.raw(sql):
    print(x.__dict__) # 将salary属性注入到当前Employee实例中
    print(x.name, x.salary)
    #print(x.gender) # 因为sql中没有投影gender，这条语句会触发查询
```

多对多

```
CREATE TABLE `employees` (
  `emp_no` int(11) NOT NULL,
  `birth_date` date NOT NULL,
  `first_name` varchar(14) NOT NULL,
  `last_name` varchar(16) NOT NULL,
  `gender` smallint(6) NOT NULL DEFAULT '1' COMMENT 'M=1, F=2',
  `hire_date` date NOT NULL,
  PRIMARY KEY (`emp_no`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `departments` (
  `dept_no` char(4) NOT NULL,
  `dept_name` varchar(40) NOT NULL,
  PRIMARY KEY (`dept_no`),
  UNIQUE KEY `dept_name` (`dept_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `dept_emp` (
  `emp_no` int(11) NOT NULL,
  `dept_no` char(4) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date NOT NULL,
  PRIMARY KEY (`emp_no`,`dept_no`),
  KEY `dept_no` (`dept_no`),
  CONSTRAINT `dept_emp_ibfk_1` FOREIGN KEY (`emp_no`) REFERENCES `employees`
(`emp_no`) ON DELETE CASCADE,
  CONSTRAINT `dept_emp_ibfk_2` FOREIGN KEY (`dept_no`) REFERENCES `departments`
(`dept_no`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

联合主键问题依然存在，所以做法同上。修改dept_emp表，增加id自增主键。

构建模型

```
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)

    @property
    def name(self):
        return "*{} {}*".format(self.first_name, self.last_name)

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

    __str__ = __repr__

class Department(models.Model):
    class Meta:
        db_table = 'departments'

    dept_no = models.CharField(primary_key=True, max_length=4)
    dept_name = models.CharField(max_length=40, null=False, unique=True)

    def __repr__(self):
        return "<Department: {} {}>".format(
            self.dept_no, self.dept_name)

    __str__ = __repr__

class Dept_emp(models.Model):
    id = models.AutoField(primary_key=True) # 新增自增主键，解决不支持联合主键问题
    emp_no = models.ForeignKey(to='Employee', on_delete=models.CASCADE,
                               db_column='emp_no') # 写模块.类名，当前模块写类名
    dept_no = models.ForeignKey(to='Department', on_delete=models.CASCADE,
max_length=4,
                                db_column='dept_no')
    # django会给外键字段自动加后缀_id，如果不需要加这个后缀，用db_column指定
    from_date = models.DateField(null=False)
    to_date = models.DateField(null=False)

    class Meta:
        db_table = 'dept_emp'

    def __repr__(self):
        return "<DeptEmp: {} {}>".format(self.emp_no, self.dept_no)

    __str__ = __repr__
```

```
import os
import django
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'salary.settings')
django.setup()

from employee.models import Employee, Department
```

```
empmgr = Employee.objects
deptmgr = Department.objects

# 查询10010员工的所在的部门编号及员工信息
emp = empmgr.filter(pk=10010).get() # 只查employees
print('-' * 30)
depts = emp.dept_emp_set.all() # 懒查
for x in depts: # 查dept_emp中的2个部门，然后再查departments 2次
    print(type(x), x) #
    e = x.emp_no #
    print(type(e), e)
    d = x.dept_no #
    print(type(d), d)

    print(e.emp_no, e.name, d.dept_no, d.dept_name)
    print()
```

迁移

如果建立好模型类，想从这些类来生成数据库的表，使用下面语句。

```
$ python manage.py makemigrations
$ python manage.py migrate employee
```

如果使用`manage.py migrate`是Django中所有为迁移的模型类都生成表。

练习

1、员工employees和头衔titles表什么关系

2、查询10009员工所有的头衔

总结

在开发中，一般都会采用ORM框架，这样就可以使用对象操作表了。

Django中，定义表映射的类，继承自Model类。Model类使用了元编程，改变了元类。

使用Field实例作为类属性来描述字段。

使用ForeignKey来定义外键约束。

是否使用外键约束?

1. 力挺派
   能使数据保证完整性一致性
2. 弃用派
   开发难度增加，大量数据的时候影响插入、修改、删除的效率。
   在业务层保证数据的一致性。

马哥教育
IT人的高薪职业学院