django 开发简介二

王敏虎

电子科协网宣部

2017 年 6 月 28 日

EESAST

EESAST

数据库

长期储存在计算机内的，有组织的，可共享的数据集合。
A Database is a collection of data, typically describing the activities of one or more related organizations.

为什么需要数据库

- 网页开发有大量的用户数据需要存储
- 网页开发需要快速的对用户的请求做出响应
- 用户数据的安全性和稳定性非常重要
- 用户数据需要保持逻辑独立和物理独立

数据模型

数据模型是对现实世界中数据特征的抽象，是描述、组织、操作数据的基础。

数据模型的组成

- 数据结构
- 数据操作
- 数据的约束条件

目前使用的最多的是关系模型。

关系模型

关系模型以关系代数为数学基础。关系模型中，数据的逻辑结构就是一张表。

主要术语

- 关系：即一张二维表
- 元组：表中的一行
- 属性：表中的一列
- 主码：唯一确定元祖的一个属性组
- 域：属性的取值范围

例子

学号	姓名	年龄	性别	院系
10000	xxx	xx	男	电子系
10001	xx	xx	女	电子系

解释

一位学生的信息是一个元组；
一组信息，如学号，是一个属性；
通常，我们把学号当做主码；
性别的域是 {男，女}

外码

外码

如果某关系的一个属性，不是自己的主码，而是另一个关系的主码，那么它被称为外码。外码用来表示关系之间的联系。

例子

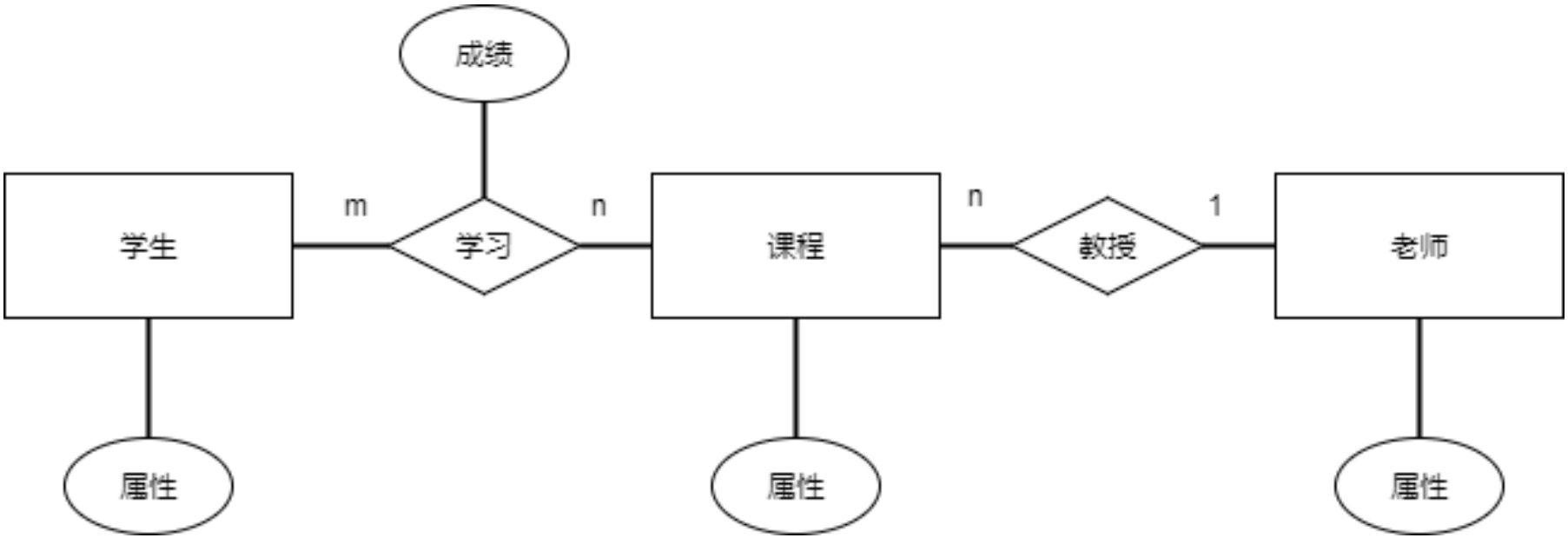
学生的属性“院系”中储存的是院系号，通过院系号在另一张院系表中可以查询到院系的详细资料。在这里院系号就是一个外码。

现实世界到关系的转换

通常，我们需要将现实中的用自然语言描述的关系转换成严谨的使用数学描述的关系。举个例子：

现实世界中的关系

每个学生有姓名、年龄、学号、年级、院系、平均成绩等属性，每个课程有课程号、课程名、选课人数等属性，每位老师有教师号、姓名等属性。每位学生可以选修多门课程，每门课程有多名学生选修，每门课程由一位老师教授，一位老师可以同时教授多门课程。



这种图被称为 E-R 图，E-R 图中矩形表示一个关系，椭圆形表示关系的属性，菱形表示关系之间的联系。

从中我们可以得出关系模型

关系模型

学生：学号 (主码), 姓名、年龄、院系、平均成绩
课程：课程号 (主码)、课程名、选课人数、教师 (外码)
选课：学号 (主码)、课程号 (主码)、成绩

通常，我们使用 SQL 语言对数据库进行增、删、查、改等操作，但是 Django 定义了一套非常方便的数据库接口，因此我们不需要特别学习 SQL，也可以充分利用数据库的优势。

```
from django.db import models

class Department(models.Model):
    department_ID = models.CharField(max_length = 6,primary_key =
        True)
    name = models.CharField(max_length = 10)

def __str__(self):
    return self.name
```

```
class Student(models.Model):
    student_ID = models.CharField(max_length = 11,primary_key =
        True)
    name = models.CharField(max_length = 10)
    age = models.IntegerField(null = True)
    grade = models.FloatField(null = True)
    department = models.ForeignKey(Department, on_delete = models.
        CASCADE)

    def __str__(self):
        return self.name
```

几点说明

- Django 使用一个类来描述一个关系，类的一个元素作为关系的一个属性
- 需要指明属性的类型，具体类型可以通过查询 django 文档获得，django 提供了数十种类型以适应不同的需求
- `primary_key` 用来指明主码
- 此处的外码使用`ForeignKey` 来表示
- `__str__` 函数的意义接下来会详细解释

SQLite 和 DB Browser for SQLite

- 在部署中，我们使用的是功能强大的开源数据库 MySQL，但在开发中，使用更轻量级的 SQLite 会方便许多。
- SQLite 自动集成在 python 中，不需要特别安装，事实上，如果不在 Django 的设置文件中指明使用 MySQL，Django 会自动使用 SQLite。
- 使用 DB browser for SQLite(下载地址) 可以不通过 SQL 语言观察数据库和数据库中的表中的结构。

- 用 DB Browser 打开项目目录下的`db.sqlite` 文件。发现其中并没有我们刚刚写好的表。这是因为我们还没有把数据结构的改变传到数据库。
- 使用`python manage.py makemigrations` 和`python manage.py migrate` 两条指令完成数据库的同步。
- 这两条指令实际上使 Django 自动生成了对应的 SQL 文件，并在数据库中执行了相应文件，我们可以在`MyAPP/migrations` 文件夹中找到对应的 migration 文件,migration 文件再进一步转换为 SQL 文件。

在运行同步命令后，我们可以在数据库中找到刚刚写好的两张表。
要往表中插入元素。我们将在 django shell 中示范。
运行python manage.py shell

```
>>> from MyApp.models import Student
>>> from MyApp.models import Department
>>> depart = Department(department_ID = '100000',name = '电子系')
>>> depart
<Department: 电子系>
>>> depart.save()
>>> Department.objects.all()
<QuerySet [<Department: 电子系>]>
>>> Department.objects.create(department_ID = '100001', name = "
    微纳电子系")
<Department: 微纳电子系>
>>> Department.objects.all()
<QuerySet [<Department: 电子系>, <Department: 微纳电子系>]>
```

```
>>> Department.objects.filter(department_ID__exact = '100001')
<QuerySet [<Department: 微纳电子系>]>
>>> Department.objects.filter(department_ID__in = ['100000','
    100003'])
<QuerySet [<Department: 电子系>]>
>>> Department.objects.filter(department_ID__startswith = '10000'
    )
<QuerySet [<Department: 电子系>, <Department: 微纳电子系>]
>>> Department.objects.filter(department_ID__in = ['100000','
    100003'])[0].department_ID
'100000'
>>> Department.objects.get(department_ID__in = ['100000','100003'
    ]).department_ID
'100000'
```

```
>>> Department.objects.get(department_ID__in = ['100000','100001'
    ]).department_ID
Traceback (most recent call last):
File "<console>", line 1, in <module>
File "C:\Program␣Files\Python36\lib\site-packages\django\db\
    models\manager.py", line 85, in manager_method
return getattr(self.get_queryset(), name)(*args, **kwargs)
File "C:\Program␣Files\Python36\lib\site-packages\django\db\
    models\query.py", line 383, in get
(self.model._meta.object_name, num)
MyApp.models.MultipleObjectsReturned: get() returned more than
    one Department -- it returned 2!
```

```
>>> Department.delete(Department.objects.get(department_ID = '
    100001'))
(1, {'MyApp.Student': 0, 'MyApp.Department': 1})
>>> Department.objects.all()
<QuerySet [<Department: 电子系>]>
>>> depart = Department.objects.all()[0]
>>> stu = Student(student_ID = '10000')
>>> stu.name = "张三"
>>> stu.age = 10
>>> stu.grade = 0
>>> stu.department = depart
>>> stu
<Student: 张三>
>>> stu.save()
>>> Student.objects.all()
<QuerySet [<Student: 张三>]>
```

```
>>> stu = Student.objects.get(name = "张三")
>>> stu.department.department_ID
'100000'
>>> Department.objects.get(department_ID = stu.department.
    department_ID).student_set.all()
<QuerySet [<Student: 张三>]>
#在model中，可以修改field的related_name属性覆盖student_set
```

```
class Course(models.Model):
    course_ID = models.CharField(max_length = 11,primary_key =
        True)
    name = models.CharField(max_length = 20)
    student = models.ManyToManyField(Student,through = 'Select',
        through_fields = ('course','student'))

    def __str__(self):
        return self.name

class Select(models.Model):
    course = models.ForeignKey(Course, on_delete = models.CASCADE)
    student = models.ForeignKey(Student,on_delete = models.CASCADE
        )
    grade = models.IntegerField(null = True)
```

```
>>> a = Course(course_ID = '100',name = '电子电路与系统基础')
>>> a.save()
>>> a.student
<django.db.models.fields.related_descriptors.
    create_forward_many_to_many_manager.<locals>.
    ManyRelatedManager object at 0x000001C66A151390>
>>> a.student.all()
<QuerySet []>
>>> from MyApp.models import Select
>>> Select.objects.all()
<QuerySet []>
>>> sel = Select(student = Student.objects.all()[0], course =
    Course.objects.all()[0],grade = 90)
>>> sel.save()
>>> a.student
<django.db.models.fields.related_descriptors.
    create_forward_many_to_many_manager.<locals>.
    ManyRelatedManager object at 0x000001C66A1721D0>
>>> a.student.all()
<QuerySet [<Student: Student object>]>
```

```
>>> a.student.all()[0]
<Student: Student object>
>>> a.student.all()[0].name
'张三'
>>> b = Student.objects.all()[0]
>>> b
<Student: Student object>
>>> b.course_set.all()
<QuerySet [<Course: 电子电路与系统基础>]
>>> b.select_set.get(course = Course.objects.get(name = "电子电路
    与系统基础")).grade
90
```

- Django Admin 系统是 Django 最优秀的特性之一，也是使用 Django Model 的重要理由
- Django Admin 系统提供了简单的后台管理解决方案
- Admin 系统会根据设置文件中的语言选项自动进行翻译
- urls.py 中的第一行即是 admin 系统的域名，`http://domine/admin`

superuser

要使用 Django 的后台功能，需要创建一个超级用户账号。使用 `python manage.py createsuperuser` 并按照提示进行操作即可。

Django 用户系统

- Django 提供了一套完整的用户解决方案，包括注册、登陆、加密、账户和群组以及对应的权限设置。在数据库中可以看到对应的表。
- Django 的基础组件是彼此解耦的，但是很多方便的库 (django.contrib) 是彼此相关的。

登陆后台

在浏览器中输入后台的域名，输入创建的超级用户账户/密码，即可进入后台界面。

EESAST

我们需要在admin.py 中注册我们写好的类。

```
from django.contrib import admin
from .models import Student,Course,Department

class StudentAdmin(admin.ModelAdmin):
    pass

class CourseAdmin(admin.ModelAdmin):
    pass

class DepartmentAdmin(admin.ModelAdmin):
    pass

admin.site.register(Student,StudentAdmin)
admin.site.register(Course,CourseAdmin)
admin.site.register(Department,DepartmentAdmin)
```

关于 django admin 的说明

- django 设计者提出的 django 使用方法是，首先完成 model 和 admin 部分，而后让非开发者开始着手添加内容，开发者继续完成后续代码的编写。
- 代码一旦部署后就不能轻易改动，直接在数据库中编写 SQL 语言对于通常不具备开发能力的网站管理者也是不容易的，这正是后台的重要性所在。
- django 后台的样式和操作都是可以定制的，也可以使用 django 的权限系统为不同的账户赋予操作不同 model 的权限，这部分内容留待开发中自学。

EESAST