``模型层

配置Mysql数据库

> 在确保mysql数据库可以连接使用的情况下；
>
> 首先在数据库中创建专为django使用的库`django_data`

```
create database django_data;
```

> 配置django的settings.py文件中的DATABASES属性如下

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # 数据库引擎
        'NAME': "django_data", # 使用的库名
        "USER": "root", # 用户名
        "PASSWORD": "woaini21g", # 数据库密码
        "HOST": "localhsot", # 数据库主机地址
        "PORT": "3306"
    }
}
```

> 由于使用`django`的`Python`版本为3+；
>
> 此时对于`mysql`的支持已经变为`pymysql`，而对于`django`加载数据库引擎时还需要使用2版本的`mysqldb`名称
>
> 现在先需要我们安装`pymysql`之后在项目中重申mysql引擎

1. 首先安装pymysql

```
 pip install pymysql -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2. 项目主目录下的`__init__`文件中添加如下内容

```
import pymysql
pymysql.install_as_MySQLdb()
```

3. 现在整个项目的数据库使用已经切换到了`mysql`

模型层字段

> 在模型层类中的字段即是数据库中表的字段，表的字段设计非常重要
>
> 每一个字段都是Field基类的一个实例（Field类用来建立字段与数据库之间的映射）
>
> 模型字段定义不能以下划线结尾

- django会根据在模型类中定义的字段属性来确定以下几点工作
  - 数据库中使用的数据类型
  - 模型类对应的表单类渲染时使用的表单类型及`HTML`部件
  - 必填字段等最低限度的验证要求检查，包括`admin`界面下自动生成的表单

BooleanField

> `BooleanField(**options)`：True/False字段，默认值为None

> 表单类型：**CheckboxInput**，`<input type='checkbox' ...>`

CharField

> `CharField(max_length=None)`：字符串字段

> 含有一个必须参数：`max_length`设置最大的**字符数**长度限制；

> 表单类型：**TextInput**，`<input type="text" ...>`

DateField

> `DateField(auto_now=False, auto_now_add=False,**options)`：以`datetime.date`实例表示的日期

> 含有两个可选参数：`auto_now`、`auto_now_add`
>
> `auto_now`：该值为True时，每次在保存数据对象时，自动设置该字段为当前时间，也可以理解为自动更新最后一次修改时间
>
> `auto_now_add`：该值为True时，该字段设置在第一次数据对象创建时，可以记录当前字段创建的时间值

> **注意**：避免矛盾，`auto_now`，`auto_now_add`，`default`不能同时出现，一个字段属性只能有其中一条设置，当设置了`auto_now`，或`auto_now_add`时，也会让该字段默认具有`blank=True`（字段可以为空）属性

> 表单类型：**TextInput**，`<input type="text" ...>`

DatetimeField

> `DatetimeField(auto_now=False, auto_now_add=False,**options)`：以`datetime.datetime`实例表示的日期和时间
>
> 和`DateField`具有相同的字段属性

DecimalField

> `DecimalField(max_digits=None,decimal_places=None, **options)`：以Decimal实例标示的十进制浮点数类型

> 含有两个可选参数：`max_digits`、`decimal_places`
>
> `max_digits`：位数总数，包括小数点后的位数，必须大于`decimal_places`参数
>
> `decimal_places`：小数点后的数字数量，精度

> 表单类型：**TextInput**，`<input type="text" ...>`

EmailField

> `EmailField(max_length=254, **option)`：`CharField`子类，表示`Email`字段，并会检查是否为合法邮箱地址

> 默认参数：`max_length`，表示邮箱地址长度，默认为254

> 表单类型：**TextInput**，`<input type="text" ...>`

FloatField

> `FloatField(**options)`：使用`float`实例来表示的浮点数

> 表单类型：**TextInput**，`<input type="text" ...>`

IntegerField

> `IntegerField(**options)`：一个整数，范围由`-2147483648`到`2147483647`

GenericIPAddressField

> `GenericIPAddressField(protocol=both, unpack_ipv4=False, **options)`：一个IPV4或IPV6地址的字符串

> 默认参数：`protocol`、`unpack_ipv4`
>
> `protocol`：IP协议，ipv4或ipv6，默认`both`为全选
>
> `unpack_ipv4`：解析IP地址，只有当协议为`both`时才可以使用

> 表单类型：**TextInput**，`<input type="text"...>`

SlugField

> `SlugField(max_length=50, **option)`：只包含字母、数字、下划线的字符串，常用来表示连接中的`path`部分或者一些其他短标题类型数据

TextField

> `TextField(**options)`：大文本字段

> 表单类型：**Textarea**，`<textarea>...</textarea>`

URLField

> `URLField(max_length=200, **options)`：`CharField`的子类，存储URL的字段

> 表单类型：**TextInput**，`<input type="text"...>`

字段属性

> 以上所介绍的字段，均支持以下属性

null

> 如果该值为True，Django将在数据库中将控制存储为NULL
>
> 字符串字段CharField与TextField要避免使用null，因为空值字符串将存储空字符串（""）,而不是null值。

> 对于字符串类型的数据字段，大多数情况下，django使用空字符串代表空值

blank

> 如果该值为True，则在验证时该字段值可以为空；
>
> null为数据库存储层面可以为空，而blank为表单验证层面可以填写空值

choices

> 一个二元组的列表或元组；
>
> 元组中第一个值为真正在数据库中存储的值，第二个值为该选项的描述
>
> 该值一旦被设定，表单样式会显示选择框，而不是标准的文本框，选择框内的选项为choices中的元组

```
class TestTable(models.Model):
    CHAR_CHOICE = [
        ('H',"非常苦难"),
        ('M',"中等难度"),
        ('S',"非常简单"),
    ]
    choicechar = modesl.CharField(max_length=1,choices=CHAR_CHOICE)
```

- `choices`字段也支持分类的写法

```
CHAR_CHOICE = [
    ('A',
        (
            ('H',"Hard"),
        )
    ),
    ('B',
        (
            ('M',"Medium"),
        )
    ),
    …
]
```

> 分类的名称作为元组中的第一个值，
>
> 元组的第二个值为该分类下的一个新的二元组序列数据

db_column

> 数据库中用来表示该字段的名称，如果未指定，那么Django将会使用`Field`名作为字段名

db_index

> 当该值为True时，为该字段创建索引

default

> 该字段默认值，可以是一个值或是一个回调函数
>
> 当是一个函数对象时，在创建新对象时，函数调用

editable

> 如果设置该值为False，那么这个字段将不允许被编辑
>
> 不会出现在admin后台界面下，以及其他ModelForm表单中，同时也会跳过模型验证

primary_key

> 设置该值为`True`时，该字段成为模型的主键字段，**一个模型类同时只能有一个主键**
>
> 如果一个表中不存在任意一个设置好的主键字段，**django会自动设置一个自增的**`AutoField`字段来充当主键，该值可以用`pk`，`id`方式获取。主键的设置还意味着，`null=False`，`unique=True`

unique

> 如果该值为True，代表这个数据在当前的表中有唯一值
>
> 这个字段还会在模型层验证存储的数据是否唯一
>
> unique的设置也意味着当前字段具备索引的创建
>
> `ManyToManyField`、`OneToOneField`与`FileField`字段不可以使用该属性

verbose_name

> 对于字段的一个可读性更高的名称
>
> 如果没有设置该值，django将字段名中的下换线转换成空格，作为当前字段的数据库中名称

模型元属性

> 在模型类的`Meta`类中，可以提供一系列的元选项，可以方便对该模型类进行属性设置或约束等

```
class TestTable(models.Model):
    …
    class Meta:
        ordering = [Fields]
        …
```

abstract

> 代表当前模型类为抽象基类，不会创建真正的数据表，只是为了其他模型类继承使用

```
abstract = True
```

app_label

> 当模型类被定义在了其他app下，这个属性用来描述当前表属于哪个app应用

```
app_label = "MyApp"
```

db_table

> 当前模型类所对应的表名，未设置时，django默认将表名与app名由下划线组成，作为表名
>
> 需要注意这个表名为真实在数据库中所使用的，所以该元选项的使用应在数据表创建之前
>
> 如果在表已经存在的情况下去修改，会导致数据库内表与模型类表名不一致而查找不到报错

ordering

> 当前表中的数据存储时的排序规则，这是一个字段名的字符串，可以是一个列表或元组；
>
> 每一个字符串前可以使用"-"来倒序排序，使用"?"随机排序
>
> ordering排序规则的添加，也会增加数据库的开销

```
ordering = ['-birthday', 'age']
#先按照birthday倒序排序，再按照age字段进行排序。
```

unique_together

> 用来设置表中的不重复字段组合
>
> 格式为一个元组，元组中的每个数据都是一个元组，用来描述不重复的组合字段
>
> 如果只处理单一字段组合，可以是一个一维的元组
>
> 联合约束

```
unique_together = (('name','phone'),)
```

verbose_name

> 一般设置该表展示时所用的名称，名称被自动处理为复数，字符串后加一个"s"

verbose_name_plural

> 与`verbose_name`功能相同，但是不会自动在字符串后加"s"以表复数
>
> 设置表的复数名称

模型操作

> 在进行模型操作的学习之前，可以先创建一个测试的数据库模型类，如下所示

```
class Person(models.Model):
    name = models.CharField(max_length=10,verbose_name="姓名")
    age = models.IntegerField(verbose_name="年龄")
```

创建对象

> django自带了一个数据库测试的shell工具
>
> 这是一个非常方便可以让我们对django代码进行测试的环境

> 可以直接通过`python manage.py shell`命令行管理工具来打开

实例save创建数据

> **通过模型类的关键词参数实例化一个对象来进行数据的创建**

```
>>> from app.models import Person
>>> p1 = Person(name='张三',age=15)
>>> p1.save()
```

> 以上的代码，在为字段赋予值之后，通过实例的save函数进行该数据的保存
>
> 在数据库底层执行了`SQL`语句中的`insert`操作，并且，在我们显示调用`save`之前，`django`不会访问数据库，实例数据只存在于内存中

> **注意**：`save`函数没有返回值

create方法创建数据

```
>>> P1 = Person.objects.create(name='李四',age=20)
```

> 这条语句创建一条数据，并且返回一个数据在内存中的实例P1
>
> 之后可以通过这个实例字段P1对数据库中该条数据进行修改或删除操作
>
> `create`方法一步到位，`save`方式可以慢悠悠的赋予字段值，最后赋予结束再save

查找对象

> 接下来，我们将通过模型类中的管理器进行数据的查询；
>
> 管理器`(Manager)`是每一个模型类所具有的，默认名为`objects`
>
> 模型类通过模型类调用`orm`数据接口，其实就是在对数据表进行操作。
>
> **注意**，具体的某一条数据无法访问这个管理器

`all()`

> 获取一个表中的所有数据，返回`QuerySet`数据对象

- ```
  all_person = Person.objects.all()
  ```

`filter(**kwargs)`

> 返回一个包含数据对象的集合，满足参数中所给的条件

- ```
  res = Person.objects.all().filter(age__lt=16)
  res = Person.objects.filter(age__lt=16)
  ```

> 我们在查询过程中，除了直接使用字段属性进行验证
>
> 还可以在字段名之后使用双下化线来标明更加详细的字段筛选条件（在下一节会有详细的字段筛选条件介绍），也叫做链式过滤

> 这也是为什么表单类字段不可以以下换线结尾的原因

`exclude(**kwargs)`

> 返回一个包含数据对象的集合，数据为不满足参数中所给的条件

> filter()查询会始终返回一个结果集，哪怕只有一个数据。
>
> 但是有些时候，我们对于一些在数据表中的唯一数据进行查询时，可以使用更加合适的`get`方法

> **注意**：创建结果集的过程不涉及任何数据库的操作，查询工作是惰性的，在上面的查询方式中，查询代码不会实际访问数据库，只有查询集在真正使用时，django才会访问数据库

`get(**kwargs)`

> 获取唯一单条数据

> get获取数据只会返回一条匹配的结果，获取的数据只能在数据库中有一条
>
> 如果返回多个结果，会引发`MultipleObjectsReturned`异常
>
> 如果没有任何匹配到的结果也会引发`DoesNotExist`异常

- ```
  Person.objects.get(pk=1)
  ```

`order_by(*field)`

> 默认情况下，数据表使用模型类中的Meta中指定的ordering选项进行排序
>
> 现在也可以通过使用order_by函数进行查询结果的排序

- ```
  Person.objects.order_by('age')
  ```
- ```
  Person.objects.all().order_by('-age')
  ```

`count()`

> 返回数据库中对应字段的个数，并且该函数永远不会引发异常

```
models.Person.objects.filter(age=20).count()
Person.objects.count()
```

> 使用`count`函数时，还需要对数据表进行迭代访问
>
> 所以有时使用已生产好的结果集，通过len函数获取长度，这种方式效率会更高
>
> `count`方法的调用会导致额外的数据库查询

`values(*fields)`

> 返回一个查询集结果，但是迭代访问时返回的是字典，而不是数据实例对象
```

```
models.Person.objects.all().values()
models.Person.objects.values()
```

链式过滤条件

- `exact`

  > 如果在查询过程中，没有提供查询类型（没有`双下划线`），那么查询类型就会被默认指定为`exact`，这是一种严格查找的方式，用来在数据库中查找和查询时的关键词参数完全一致的内容

  ```
  >>> Person.objects.filter(account='root')
  >>> Person.objects.filter(account__exact='root')
  ```

- `iexact`

  > 忽略大小写的匹配

  ```
  >>> Person.objects.filter(account__iexact='root')
  #匹配到的结果可能是Root, ROot, ROOt, ROOT
  ```

- `startswith、endswith`

  > 分别匹配开头和结尾，区分大小写

  ```
  >>> Person.objects.filter(passwd__startswith='admin')
  #匹配以admin开头的数据
  ```

- `istartswith、iendswith`

  > 分别匹配开头和结尾，忽略大小写

  ```
  >>> Person.objects.filter(passwd__istartswith='admin')
  匹配以不区分大小写的字符串admin为开头的数据
  ```

- `gte`

  > 大于或等于

  ```
  >>> Person.objects.filter(reg_data__gte=datetime.date.today)
  ```

- `lte`

  > 小于或等于

  ```
  >>> Person.objects.filter(reg_data__lte=datetime.date.today)
  ```

修改对象

> 获取到对应的数据实例之后，通过`.`的方式访问数据实例中的属性，进行数据的字段修改

```
p = models.Person.objects.get(pk=1)
p.age = 21
p.save()
```

> 对过滤出的结果链式调用`update()`函数，这样的修改，类似批量修改，`update`函数会返回成功修改的个数

```
models.Person.objects.filter(age__gt=100).update(age=25)
# 将所有年纪小于100的人的年纪改为20
```

删除对象

> 对于普通的单表数据删除，获取到数据实例对象后调用内置的`delete()`函数即可

```
models.Person.objects.get(pk=1).delete()
```

> 需要注意的是，删除一条数据之后，默认占有的主键ID值并不会被下一个新插入的值所占用
>
> 比如 1，2，3，4；删除掉3之后，剩下：1，2，4；下一个值存储时，id是5，3不会被复用

字段关系

- 字段关系是`django`维护表关系的方式；其中主要有一对一，多对一以及多对多，
- 现在的一对一及多对一关系中需要设置`on_delete`属性用来描述当关联数据被删除时的操作，有如下一些

> **models.CASCADE**：删除关联数据,与之关联也删除
>
> models.PROTECT：删除关联数据,引发错误ProtectedError
>
> **models.SET_NULL**：与之关联的值设置为null（前提FK字段需要设置为可空）
>
> models.SET_DEFAULT： 删除关联数据,与之关联的值设置为默认值（前提FK字段需要设置默认值）
>
> models.DO_NOTHING：删除关联数据,什么也不做

一对一关系

> 模型类使用`OneToOneField`用来定义一对一关系；
>
> 比如当你拥有一个老师表时，紧接着你还需要一个教授表，那么教授表可能拥有老师表的一系列属性，那么你还不想把老师表中的字段直接复制到教授表那么可以通过`OnToOneField`来实现教授表继承老师表。
>
> 其实，在使用模型类继承时，也隐含有一个一对一关系

- `OneToOneField(to, on_delete, parent_link=False, options)`

```
class Teacher(models.Model):
    name = models.CharField(max_length=50)
    age = models.CharField(max_length=50)
    def __str__(self):
        return self.name
class Professor(models.Model):
    teacher = models.OneToOneField(Teacher,primary_key=True,on_delete=models.CASCADE)
    big_project = models.CharField(max_length=50)
    def __str__(self):
        return self.teacher.name
```

> 在`manage.py shell`下进行数据库操作

```
>>> t1 = Teacher.objects.create(name='Jack',age='22')
>>> t2 = Teacher.objects.create(name='Bob',age='17')
>>> p1 = Professor.objects.create(teacher=t1,big_project='雾霾净化术')
>>> p1.teacher
<Teacher: Jack>
>>> p1.teacher = t2
>>> p1.save()
>>> p1.teacher
<Teacher: Bob>
```

> 在上面的测试中，看似已经将p1对应的教授变成了Bob；
>
> 但是在数据库中之前t1老师所对应的教授信息还存在，此时的赋值操作并不会覆盖掉教授他之前的教授数据，只是重新创建了一条。
>
> 正确的做法应该是将某一条数据的一对一关系通过`delete`关系先删除之后再重新赋予

多对一关系

> `Django`使用`django.db.models.ForeignKey`定义多对一关系。
>
> `ForeignKey`需要一个位置参数：与该模型关联的类

> 生活中的多对一关系：班主任，班级关系。一个班主任可以带很多班级，但是每个班级只能有一个班主任

```
class Headmaster(models.Model):
    name = models.CharField(max_length=50)
    def __str__(self):
        return self.name
class Class(models.Model):
    class_name = models.CharField(max_length=50)
    teacher = models.ForeignKey(Headmaster,null=True,on_delete=models.SET_NULL)
    def __str__(self):
        return self.class_name
```

```
>>> H1 = Headmaster(name='渔夫')
>>> H1.save()
>>> H1
<Headmaster: 渔夫>
>>> H2 = Headmaster(name='农夫')
>>> H2.save()
>>> Headmaster.objects.all()
[<Headmaster: 渔夫>, <Headmaster: 农夫>]
```

> 以上创建了两条老师数据
>
> 由于我们设置外键关联可以为空`null=True`,所以此时在班级表创建时，可以直接保存，不需要提供老师数据

```
>>> C1 = Class(class_name='一班')
>>> C2 = Class(class_name='二班')
#如果外键设置不为空时，保存会引发以下错误
# IntegrityError: NOT NULL constraint failed: bbs_class.teacher_id
>>> C1.teacher = H1
>>> C2.teacher = H2
>>> C1.save()
>>> C2.save()
```

> 将老师分配个班级之后，由于班级表关联了老师字段，我们可以通过班级找到对应老师
>
> 虽然老师表中没有关联班级字段，但是也可以通过老师找到他所带的班级，这种查询方式也叫作关联查询
>
> 通过模型类名称后追加一个'_set'，来实现反向查询

```
>>> H1.class_set.all()
<QuerySet [<Class: 一班>]>
```

> 由于我们这是一个多对一的关系，也就说明我们的老师可以对应多个班级
>
> 我们可以继续给H1老师分配新的班级

```
>>> C3 = Class(class_name='三班')
>>> C3.teacher = H1
>>> C3.save()
>>> H1.class_set.all()
[<Class: 一班>, <Class: 三班>]
```

> 一个班级只能对应一个老师，外键是唯一的，那么你在继续给C1班级分配一个新的老师时，会覆盖之前的老师信息，并不会保存一个新的老师

```
>>> H3 = Headmaster(name='伙夫')
>>> H3.save()
>>> C1.teacher
<Headmaster: 渔夫>
>>> C1.teacher=H3
>>> C1.save()
>>> C1.teacher
<Headmaster: 伙夫>
```

> 把这个班级的老师删除，由于设置了外键字段可以为`null`，此时班级的老师选项为`null`

```
>>> t1 = Headmaster.objects.all().first()
>>> t1
>>> c1 = Class.objects.all().first()
<Headmaster: 渔夫>
>>> c1
<Class: 一班>
>>> c1.teacher
<Headmaster: 渔夫>
>>> t1.delete()
(1, {'modelsapp.Headmaster': 1})
>>> c1 = Class.objects.all().first()
>>> c1
<Class: 一班>
>>> c1.teacher
>>> #这里什么都没有，因为此时C1的老师已经是个None了
```

> 要记得删除之后要重新获取一次数据，否则查看到的结果中还是之前获取到的有老师的班级数据

多对多关系

> 多对多关系在模型中使用ManyToManyField字段定义
>
> 多对多关系可以是具有关联，也可以是没有关联，所以不需要明确指定`on_delete`属性

> 生活中，多对多关系：一个音乐家可以隶属于多个乐队，一个乐队可以有多个音乐家

```
class Artist(models.Model):
    artist_name = models.CharField(max_length=50)
    def __str__(self):
        return self.artist_name
class Band(models.Model):
    band_name = models.CharField(max_length=50)
    artist = models.ManyToManyField(Artist)
    def __str__(self):
        return self.band_name
```

> 创建音乐家以及乐队

```
>>> from bbs.models import Artist,Band
>>> A1 = Artist.objects.create(artist_name='Jack')
>>> A2 = Artist.objects.create(artist_name='Bob')
>>> B1 = Band.objects.create(band_name='FiveMonthDay')
>>> B2 = Band.objects.create(band_name='SHE')
```

> 创建出两个乐队之后对其进行音乐家的添加
>
> 多对多字段添加时，可以使用`add`函数进行多值增加

```
>>> B1.artist.add(A1,A2)
>>> B2.artist.add(A2)
```

> `B1`乐队含有`A1`,`A2`两名成员
>
> `B2`乐队含有`A1`成员

```
>>> B1.artist.all()
[<Artist: Bob>, <Artist: Jack>]
>>> B2.artist.all()
[<Artist: Jack>]
```

> 可以在音乐家表中查找某个乐家属于哪些乐队

```
>>> Band.objects.filter(artist=A1) # 这里使用的是我们模型类来进行查找。
[<Band: SHE>, <Band: FiveMonthDay>] # A1乐家属于，SHE以及FiveMonthDay
>>> Band.objects.filter(artist=A2)
[<Band: SHE>]
```

> 也可以查找这音乐家在哪个乐队

```
>>> A1.band_set.all() # 直接通过具体数据对象进行查找
[<Band: SHE>, <Band: FiveMonthDay>]
>>> A2.band_set.all()
[<Band: SHE>]
```

> 多对多关联字段的删除，要使用`remove`来进行关系的断开
>
> 而不是直接使用`delete`，`remove`只会断开数据之间的联系，但是不会将数据删除
>
> 现在在B1乐队中删除A1乐家

```
>>> B1.artist.remove(A1)
>>> B1.artist.all()
<QuerySet [<Artist: Bob>]>
```

关联表的查询

> 如果想要查询的字段在关联表，则使用`表名小写__字段`来进行跨表查询操作

> 创建一个多对一关系的父子表，一个父亲可能有多个儿子

```
class Father(models.Model):
    name = models.CharField(max_length=30)
    age = models.CharField(max_length=30)
    def __str__(self):
        return self.name
class Son(models.Model):
    father = models.ForeignKey(Father,on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    def __str__(self):
        return self.name
```

> 创建数据

```
>>> f1 = Father.objects.create(name='Jack',age='30')
>>> s1 = Son.objects.create(name='Json',father=f1)
>>> s2 = Son.objects.create(name='Json2',father=f1)

>>> f2 = Father.objects.create(name='Bob',age='40')
>>> s3 = Son.objects.create(name='Json3',father=f2)
```

> 查询所有父亲名字是`jack`的孩子

```
>>> Son.objects.filter(father__name__exact='Jack')
[<Son: Json>, <Son: Json2>]
```

> 查询所有儿子名开头为`J`的父亲

```
>>> Father.objects.filter(son__name__startswith='J')
[<Father: Jack>, <Father: Jack>, <Father: Bob>]
```

> 获取到某一个父亲的所有孩子，通过某一条数据的`小写表名_set`反向查询

```
>>> f1.son_set.all()
>>> [<Son: Json>, <Son: Json2>]
```

数据的反向查询

> 默认的，当有某一条数据获取到之后，我们可以通过模型类名称加上一个`_set`，来实现反向查询
>
> 现在设计两个表为军队和士兵表，并且士兵多对一关联军队

```
class Aramy(models.Model):
    name = models.CharField(max_length=30)
    def __str__(self):
        return self.name
class Soldier(models.Model):
    aramy = models.ForeignKey(Aramy,on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    def __str__(self):
        return self.name
```

> 创建一些数据

```
>>> a1 = Aramy(name='一军')
>>> a1.save()
>>> s1 = Soldier(name='张三',aramy=a1)
>>> s1.save()
>>> s2 = Soldier(name='李四',aramy=a1)
>>> s2.save()
```

> 通过`soldier_set`我们就可以关联到对应的士兵表
>
> 并且对应返回结果可以执行我们常用的`filter`，`exclude`等查询操作

```
>>> a1.soldier_set.all()
[<Soldier: 张三>, <Soldier: 李四>]
>>> a1.soldier_set.filter(name='张三')
[<Soldier: 张三>]
```

> 也可以通过定义关联字段中的`related_name`值，来实现自定义的反向查询名字
>
> 且`related_name`的值必须唯一

```
class Aramy(models.Model):
    name = models.CharField(max_length=30)
    def __str__(self):
        return self.name
class Soldier(models.Model):
    aramy = models.ForeignKey(Aramy,on_delete=models.CASCADE,related_name='soldier')
    name = models.CharField(max_length=30)
    def __str__(self):
        return self.name
```

> 接下来通过某条数据反向查询

```
>>> a1 = Aramy.objects.all()[0]
>>> s1 = Soldier.objects.get(name='张三')
>>> a1.soldier.all()
[<Soldier: 张三>, <Soldier: 李四>]
```

> **注意**：related_name一定是一个唯一的值，否则反向查找时会出现二异性错误

> 也可以将`related_name`初始化为`+`，来取消反向查询

课后作业

> 了解django中如何使用原生SQL的几种方式