# Django 基础教程

1. 相关文件：

(1) urls.py：网址入口，关联到对应的 views.py 中的一个函数（或者 generic 类），访问网址就对应一个函数。

(2) views.py：处理用户发出的请求，从 urls.py 中对应过来, 通过渲染 templates 中的网页可以将显示内容，比如登陆后的用户名，用户请求的数据，输出到网页。

(3) models.py：与数据库操作相关，存入或读取数据时用到这个，当然用不到数据库的时候 你可以不使用。

(4) forms.py：表单，用户在浏览器上输入数据提交，对数据的验证工作以及输入框的生成等工作，当然你也可以不使用。

(5) templates 文件夹：views.py 中的函数渲染 templates 中的 Html 模板，得到动态内容的网页，当然可以用缓存来提高速度。

(6) admin.py：后台，可以用很少量的代码就拥有一个强大的后台。

(7) settings.py：Django 的设置，配置文件，比如 DEBUG 的开关，静态文件的位置等。

2. 新建一个 django project 命令：

```
django-admin.py startproject project_name
```

project_name 是自己的项目名称，需要为合法的 Python 包名。

3. 新建 app：

```
django-admin.py startapp app_name
```

4. 创建数据库更改文件：

```
python manage.py makemigrations
```

一般而言，django 将数据库修改命令先统一生成更改文件，然后再应用到数据库中。

5. 将更改文件应用到数据库：

```
python manage.py migrate
```

6. 开发服务器，即开发时使用，一般修改代码后会自动重启，方便调试和开发，但是由于性能问题，建议只用来测试，不要用在生产环境。启动开发服务器：

```
python manage.py runserver 8001
```

当主机有多个 IP 时，也可以使用多个 IP：

```
python manage.py runserver 0.0.0.0:8000
```

7. 开发服务器会根据需要自动重新载入 Python 代码，而不需要重启服务器。但是，诸如添加文件之类的动作，不会重启服务器，需要手动重启服务器。

8. 清空数据库：

```
python manage.py flush
```

9. 创建超级管理员：

```
python manage.py createsuperuser
```

使用 changepassword 改变管理员密码：

```
python manage.py changepassword username
```

10. 导出数据：

```
python manage.py dumpdata appname > appname.json
```

导入数据：

```
python manage.py loaddata appname.json
```

11. 调用项目的 shell：

```
python manage.py shell
```

这个 shell 的作用是可以在这个 shell 里面调用当前项目的 models.py 中的 API。

12. 数据库命令行：

```
python manage.py dbshell
```

在这个终端可以执行数据库的 SQL 语句。如果您 SQL 比较熟悉，可能喜欢这种方式。

13. 修改密码有两种方式：

(1) 命令行修改：

```
$ python manage.py changepassword *username*
```

(2) 使用函数 set_password()修改：

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

14. 不同于创建超级用户，创建普通用户可以使用函数 create_user()来实现：

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com',
'johnpassword')
>>> user.last_name = 'Lennon'
>>> user.save()
```

15.终端上输入 python manage.py 可以看到详细的命令列表。

16.新建的 app 需要添加到 settings 的 INSTALL_APPS 中去。

17.新建一个 app：

(1) 执行新建命令：

```
django-admin.py startapp learn
```

该命令会在顶层目录新增一个 learn 目录。

(2) 把我们新定义的 app 加到 settings.py 中的 INSTALL_APPS 中：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'learn',
)
```

(3) 定义视图函数：

```
#coding:utf-8
from django.http import HttpResponse

def index(request):
    return HttpResponse(u"欢迎光临 自强学堂!")
```

视图函数的第一个参数必须是 request，与网页发来的请求有关，request 变量里面包含 get 或 post 的内容，用户浏览器，系统等信息在里面。

(4) 定义视图函数相关的 URL，修改 mysite/mysite/urls.py：

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns(
    url(r'^$', learn.views.index),  # new
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

这是在顶级 urls.py 中加入视图函数。也可以使用子级 urls.py 的方式来加入视图函数，将 url 修改为：

```
url(r'^$', include('learn.urls'))
```

然后在应用目录下新建一个 url.py：

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
   url(r'^$', views.index, name='index')
   url(r'^admin/', include(admin.site.urls)),
]
```

name 用于命名你的 URL。

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
   url(r'^polls/', include('polls.urls', namespace="polls")),
   url(r'^admin/', include(admin.site.urls)),
]
```

18. url()函数具有四个参数：两个必需的 regex 和 view，以及两个可选的 kwargs 和 name：

(1) regex：正则表达式。

(2) view：视图函数。

(3) kwargs：任何关键字参数都可以以字典形式传递给目标视图。

(4) name：命名你的 URL。这样就可以在 Django 的其它地方尤其是模板中，通过名称来明确地引用这个 URL。作用有点类似于函数名。

19. url()也可以使用 include 来包含另一个 urls.py：

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
   url(r'^$', views.index, name='index')
   url(r'^admin/', include(admin.site.urls)),
]
```

20. Django 默认启用后台管理。后台管理的登录界面为服务器根目录中的/admin/。

21. ModelAdmin 类管理着后台界面的表示形式。通常，需要在 admin.py 中实现该类

的子类。

```
from django.contrib import admin
from .models import Article

class ArticleAdmin(admin.ModelAdmin):
   list_display = ('title','pub_date','update_time',)

admin.site.register(Article,ArticleAdmin)
```

list_display 表示在后台显示元组中的三项内容。

22. 使用 request.GET 来获取网页请求地址中的数据：

(1) 格式为“/add/?a=4&b=5”：

使用下列语句获取 a 和 b 的值：

```
from django.shortcuts import render
from django.http import HttpResponse

def add(request):
   a = request.GET['a']
   b = request.GET['b']
```

request.GET 类似于一个字典，更好的办法是用 request.GET.get('a',0)，当没有传递 a 的时候默认 a 为 0。(request.GET.get)

(2) 网址格式如“/add/3/4/”：

这时需要修改 urls.py，使用正则表达式匹配：

```
url(r'^add/(\d+)/(\d+)/$', calc_views.add2, name='add2'),
```

每一个正则表达式匹配一个变量。add2()定义如下：

```
def add2(request, a, b):
   a = request.GET['a']
   b = request.GET['b']
   c = int(a) + int(b)
   return HttpResponse(str(c))
```

注意 add2 的参数。

23. 在一个应用目录中使用如下的视图函数：

```
from django.shortcuts import render

def home(request):
   return render(request, 'home.html',content)
```

Home.html 是当前目录 template 目录下的 home.html 文件，content 默认为空，是一个添加到模板 home.html 的值的字典。rander 返回一个 HttpResponse 对象。

24. Django 模型是与数据库相关的，与数据库相关的代码一般写在 models.py 中。模

型需要继承自 models.Model，里面定义了一些数据库操作函数。使用和 scrapy 的模型有点类似：

```
from django.db import models

class Person(models.Model):
  name = models.CharField(max_length=30)
  age = models.IntegerField()
```

等号左边是要定义数据库中的列名，右边是列的数据类型。其余由系统完成。

25. 一个模型就是一个数据库表。

26. 模型中的每个字段都是某个 Field 子类的实例。Django 自带数十种内置的字段类型；Django 也支持字定义模型

27. 每个字段有一些特有的参数，还有一些适用于所有字段的通用参数：

(1) null：如果为 True，Django 将会把数据库中空值保存为 NULL。默认值是 False。

(2) blank：如果为 True，该字段允许为空值，默认为 False。要注意，这与 null 不同。null 纯粹是数据库范畴,指数据库中字段内容是否允许为空，而 blank 是表单数据输入验证范畴的。如果一个字段的 blank=True，表单的验证将允许输入一个空值。如果字段的 blank=False，该字段就是必填的。

(3) default：字段的默认值。可以是一个值或者可调用对象。如果可调用 ，每个新对象被创建它都会被调用。

(4) help_text：表单部件额外显示的帮助内容。即使字段不在表单中使用，它对生成文档也很有用。

(5) primary_key：如果为 True，那么这个字段就是模型的主键。如果你没有指定任何一个字段的 primary_key=True，Django 就会自动添加一个 IntegerField 字段做为主键，一般没有必要设置这个。

(6) unique：如果该值设置为 True, 这个数据字段在整张表中必须是唯一的。

```
from django.db import models

class Person(models.Model):
  name = models.CharField(max_length=30,null=true)
  age = models.IntegerField()
```

28. 使用 create 创建对象：

```
$ python manage.py shell

>>> from people.models import Person
>>> Person.objects.create(name="WeizhongTu", age=24)
<Person: Person object>
>>>
```

这时新建了一个用户 WeizhongTu。然后使用 get 从数据库是查询该对象：

```
>>> Person.objects.get(name="WeizhongTu")
```

注意，这里不是调用模型类的对象，管理器 object 函数的参数是模型的成员。

29.新建一个对象的方法有以下几种：

(1) Person.objects.create(name=name,age=age)

(2) p = Person(name="WZ", age=23)

p.save()

(3) p = Person(name="TWZ")

p.age = 23

p.save()

(4) Person.objects.get_or_create(name="WZT", age=23)

第 4 种方法是防止重复很好的方法，但是速度要相对慢些，返回一个元组，第一个为 Person 对象，第二个为 True 或 False, 新建时返回的是 True, 已经存在时返回 False.

30.获取对象有以下方法：

(1) Person.objects.all()　　获取所有数据

(2) Person.objects.all()[:10] 切片操作，获取 10 个人，不支持负索引，切片可以节约内存

(3) Person.objects.get(name=name)　　获取指定数据

31.从数据库中查询出来的结果一般是一个集合，这个集合叫做 QuerySet。

32.models.Model 相关数据库操作：

- filter()：用于查找符合条件的数据。

```
Person.objects.filter(name__contains="abc") # 名称中包含 "abc"的人
Person.objects.filter(name__icontains="abc") #名称中包含 "abc"，且 abc 不区分大小写
```

- delete()：删除指定数据

```
Person.objects.filter(name__contains="abc").delete()
```

删除名称中包含 "abc"的人。

- 批量更新，适用于.all()、.filter()、.exclude()等后面：

```
Person.objects.filter(name__contains="abc").update(name='xxx')
Person.objects.all().delete()
```

33.更新某项数据：

```
twz = Author.objects.get(name="WeizhongTu")
twz.name="WeizhongTu"
twz.email="tuweizhong@163.com"
twz.save() # 最后不要忘了保存！！！
```

34. QuerySet 是可迭代的：

```
es = Entry.objects.all()
for e in es:
  print(e.headline)
```

35. Django 官方提供了自定义 Field 的方法，继承自某个特定的 Field 类：

```
from django.db import models

class CompressedTextField(models.TextField):
……
```

36. 应用数据在后台默认是不显示的，需要使用 admin.site.register()函数向后台注册应用：

```
from django.contrib import admin
from .models import Article

admin.site.register(Article)
```

37. 应用模型需要在后面加上__unicode__()函数，否则无法在后台正确显示，例如：

```
# coding:utf-8
from django.db import models

class Article(models.Model):
  title = models.CharField(u'标题', max_length=256)
  content = models.TextField(u'内容')

  pub_date = models.DateTimeField(u'发表时间', auto_now_add=True, editable = True)
  update_time = models.DateTimeField(u'更新时间',auto_now=True, null=True)
```


左边是无法正确显示的，右边是正确显示的。

38. Django 默认使用 SQLite 数据库。如果想要更换数据库，则需要修改 mysite/settings.py 文件。

39. 使用 SQLite 之外的数据库引擎时，就必须添加 USER、 PASSWORD、HOST 等额外的设置。

40. 使用 SQLite，不需要事先创建任何东西，数据库文件将会在需要的时候自动创建。而使用 PostgreSQL 或者 MySQL 时，需要确保已经建立好一个数据库。

41. 编辑 mysite/settings.py 时，LANGUAGE_CODE 用于设置语言支持，TIME_ZONE 用于设置所在的时区：

```
LANGUAGE_CODE = 'zh-hans'

TIME_ZONE = 'Asia/Shanghai'
```

42. 自定义项目的模板：

(1) 在顶层目录创建一个 templates 目录。

(2) 打开配置文件（记住是 mysite/settings.py），在 TEMPLATES 设置中添加一个 DIRS 选项：

```
TEMPLATES = [
  {
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    'APP_DIRS': True,
    'OPTIONS': {
      'context_processors': [
        'django.template.context_processors.debug',
        'django.template.context_processors.request',
        'django.contrib.auth.context_processors.auth',
        'django.contrib.messages.context_processors.messages',
      ],
    },
  },
```

```
]
```

(3) 在 templates 下创建一个名为 admin 的文件夹，从 Django 安装的原目录下将模板页面的源文件 admin/base_site.html 复制到这个文件夹里。

(4) 编辑该文件并替换{{ site_header|default:_('Django administration') }} （包括花括号）为自己站点的名称。

43. render()用于返回一个 HttpResponse 对象，这个对象是一个渲染后的模板。第一个参数为请求对象，第二个参数模板名字，第三个参数为用于填充模板的一个字典

```
polls/views.py
from django.shortcuts import render
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

44. 在主 URLconf 下添加命名空间，可以实现分隔各应用名称：

```
mysite/urls.py
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^polls/', include('polls.urls', namespace="polls")),
    url(r'^admin/', include(admin.site.urls)),
]
```

定义之后，可以在模板里使用这个命名空间而不会出现混乱：

```
polls/templates/polls/index.html
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text
}}</a></li>
```

45. 图片、js、css 等文件称为静态文件。

46. Django 提供了三种最常见的数据库关系：多对一(many-to-one)，多对多(many-to-many)，一对一(one-to-one)。

(1) 多对一关系：Django 使用 django.db.models.ForeignKey 定义多对一关系。表 A 的任意一行对应表 B 的多行数据，反过来就行不通。

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass
```

```
class Car(models.Model):
  manufacturer = models.ForeignKey(Manufacturer)
  # ...
```

注意 ForeignKey()括号里的参数。（注意参数）

(2) 多对多关系：使用 ManyToManyField 来定义多对多关系。ManyToManyField 需要一个位置参数：和该模型关联的类。多对多的关系简单来说就是两张表里的数据，任何一行都可以对应另外一张表里的多行数据。表 A 的任意一行对应表 B 的多行数据，反之也行

```
from django.db import models

class Topping(models.Model):
  # ...
  pass

class Pizza(models.Model):
  # ...
  toppings = models.ManyToManyField(Topping)
```

注意 ManyToManyField()括号里的参数。（注意参数）

(3) 多对多关系中的其他字段：有时需要知道两个对应关系之间的详细信息，例如，音乐家表的每一个成员都对应多个音乐小组，音乐小组又多个音乐家。这里我们需要知道成员何时加入小组之类的详细信息，这就需要中介模型来定义多对多关系。其他字段放定义在中介模型里面。源模型的 ManyToManyField 字段将使用 through 参数指向中介模型。对于上面的音乐小组的例子，代码如下：

```
from django.db import models

class Person(models.Model):
  name = models.CharField(max_length=128)

  def __str__(self):          # __unicode__ on Python 2
    return self.name

class Group(models.Model):
  name = models.CharField(max_length=128)
  members = models.ManyToManyField(Person, through='Membership')

  def __str__(self):          # __unicode__ on Python 2
    return self.name

class Membership(models.Model):
  person = models.ForeignKey(Person)
  group = models.ForeignKey(Group)
```

```
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

注意 ManyToManyField()括号里的参数。（注意参数）

47. 一对一关系：OneToOneField 用来定义一对一关系。 用法和其他字段类型一样：在模型里面做为类属性包含进来。

48. 创建中介模型的实例：

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
```

49. 管理器是对 Django 模型进行数据库查询的接口。Django 应用的每个模型都拥有至少一个管理器。

50. 默认情况下，Django 会为每个模型类添加一个名为 objects 的管理器，也可以使用 models.Manager()来重命令管理器：

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

51. 显式定义默认管理器：

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

52. 通过创建中介模型的实例来建立对多对多关系后，你就可以执行查询了：

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

53. 如果你使用了中介模型，你也可以利用中介模型的属性进行查询：

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr]
```

54. 模型元数据指任何不是字段的数据，比如排序选项（ordering），数据库表名

（db_table）等。用 class Meta 来定义。class Meta 是完全可选的，所有选项都不是必须的。

55. __str__()和__unicode__() 返回人类可读的字符串时。这个方法最好重新实现。

```
# coding:utf-8
from django.db import models

class Article(models.Model):
   title = models.CharField(u'标题', max_length=256)
   content = models.TextField(u'内容')

   pub_date = models.DateTimeField(u'发表时间', auto_now_add=True, editable = True)
   update_time = models.DateTimeField(u'更新时间',auto_now=True, null=True)

   def __unicode__(self)
      return self.title
```

## 数据库操作

1. 模型的 save()：保存到数据库中。假设模型存放于文件 mysite/blog/models.py 中，下面是一个例子：

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

save() 方法没有返回值。

2. 在 ManyToManyField 关系中，需要使用字段的 add()方法来增加关联关系的记录，下面这个例子向 entry 对象添加 Author 类的实例 joe：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

在这里，entry 和 authors 分别是两个多对多表 Entry 和 Authors 的实例。

为了在一条语句中，向 ManyToManyField 添加多条记录，可以在调用 add()方法时传入多个参数，像这样：

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

3. 管理器只可以通过模型的类访问，而不可以通过模型的实例访问。

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects          #这里应该用 Blog.objects 来访问管理器
Traceback:
  ...
AttributeError: "Manager isn't accessible via Blog instances."
```

4. 获取一个表中所有对象的最简单的方式是全部获取。可以使用管理器的 all() 方法：

```
>>> all_entries = Entry.objects.all()
```

5. 在原始的的查询集上增加一些过滤条件，使用过滤器获取特定对象：

- filter(**kwargs)：返回一个符合条件的新的查询集。
- exclude(**kwargs)：查询不符合条件的结果集，并返回。

举个例子，要获取年份为 2006 的所有文章的查询集，可以使用 filter()方法：

```
Entry.objects.filter(pub_date__year=2006)
```

6. 查询集的筛选结果本身还是查询集，所以可以将筛选语句链接在一起。像这样：

```
>>> Entry.objects.filter(
...    headline__startswith='What'
... ).exclude(
...    pub_date__gte=datetime.date.today()
... ).filter(
...    pub_date__gte=datetime(2005, 1, 30)
... )
```

7. 查询集是惰性执行的，直到查询集需要求值时，Django 才会真正运行这个查询。看下这个例子：

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

上面三条语句只有在最后一行（print(q)）时才访问一次数据库。

8. 管理器的 get() 方法可以获取特定对象：

```
>>> one_entry = Entry.objects.get(pk=1)
```

9. 可以使用 Python 的切片语法来限制查询集记录的数目 。例如，下面的语句返回前面 5 个对象(LIMIT 5)：

```
>>> Entry.objects.all()[:5]
```

注意：不支持负索引。

10. 字段查询是通过后接指定的查询类型来完成。一般在查询集方法 filter()、exclude() 和 get() 里使用。查询的基本形式是 field__lookuptype=value。（中间是两个下划

线）。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

后面的“__lte”指小于等于，是查询类型的一种。

11. 查询条件中指定的字段必须是模型字段的名称。但有一个例外，对于 ForeignKey 你可以使用字段名加上_id 后缀。在这种情况下，该参数的值应该是外键的原始值。例如：

```
>>> Entry.objects.filter(blog_id=4)
```

12. 数据库 API 支持大约二十多种查询的类型，下面是一些常见查询：

(1) exact：“精确”匹配。

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

(2) iexact：大小写不敏感的匹配。

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

(3) contains：大小写敏感的包含关系测试。

```
Entry.objects.get(headline__contains='Lennon')
```

大体可以翻译成下面的 SQL：

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

13. 同时满足两个条件的查询：

```
Blog.objects.filter(entry__headline__contains='Lennon',
    entry__pub_date__year=2008)
```

至少满足一个条件的查询：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
    entry__pub_date__year=2008)
```

14. 为了方便，Django 提供一个查询快捷方式 pk ，它表示“primary key” 主键的意思。

15. Django 提供一种强大而又直观的方式来“处理”查询中的关联关系，它在后台自动处理 JOIN。 若要跨越关联关系，只需使用关联的模型字段的名称，并使用双下划线分隔，直至你想要的字段，假设有下面两个表：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
```

```
    name = models.CharField(max_length=50)
    email = models.EmailField()

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):              # __unicode__ on Python 2
```

下面这个例子根据 Blog 的 name 属性来获取所有的 Entry 对象：

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

跟 entry.authors.add(joe)差不多，只是这里是获取对象，entry.authors.add(joe)是添加作者信息。

16. 为了比较两个模型实例，只需要使用标准的 Python 比较操作符，即双等于符号：==。在后台，它会比较两个模型主键的值。

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

17. 删除方法，为了方便，就取名为 delete()。这个方法将立即删除对象且没有返回值。

```
e.delete()
```

每个查询集都有一个 delete() 方法，它将删除该查询集中的所有成员。

18. 拷贝模型实例：Django 没有内建的方法用于拷贝模型实例，要拷贝模型实例，需要将 pk 设置为 None。

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # blog.pk == 1

blog.pk = None
blog.save() # blog.pk == 2
```

如果用继承，那么会复杂一些。考虑下面 Blog 的子类：

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)

django_blog = ThemeBlog(name='Django', tagline='Django is easy',
theme='python')
django_blog.save() # django_blog.pk == 3
```

由于继承的工作方式，必须将 pk 和 id 都设置为 None：

```
django_blog.pk = None
django_blog.id = None
django_blog.save() # django_blog.pk == 4
```

19. 更新查询集：update()方法用于更新某个字段值。

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

20. 更新查询集唯一的限制是它只能访问一个数据库表，也就是模型的主表。

```
>>> b = Blog.objects.get(pk=1)

# Update all the headlines belonging to this Blog.
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

21. 在多对一关系中，如果 ForeignKey 在模型 A，模型 B 实例可以通过管理器 a_set 返回模型 A 的所有实例。默认情况下，这管理器的名字为 a_set，其中 a 是源模型的小写名称：

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.
```

## 原始 SQL 查询

1．raw()管理器方法用于执行原始的 SQL 查询，并返回模型的实例：

```
Manager.raw(raw_query, params=None, translations=None)
```

示例：

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
```

2．raw()方法支持索引访问：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

3．.可以向 raw()方法传递 params 参数：

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

为了避免受到 SQL 注入攻击，不能像下面这样定义：

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
>>> Person.objects.raw(query)
```

4. django.db.connection 对象提供了常规数据库连接的方式。为了使用数据库连接，先要调用 connection.cursor()方法来获取一个游标对象之后，调用 cursor.execute()来执行 sql 语句，调用 cursor.fetchone()或者 cursor.fetchall()来返回结果行。

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()
    return row
```

## 处理 HTTP 请求

1. URLconf 用于给一个应用设计 URL。

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

(1) 若要从 URL 中捕获一个值，只需要在它周围放置一对圆括号。

(2) 不需要添加一个前导的反斜杠。例如，应该是^articles 而不是 ^/articles。

(3) 正则表达式前面的'r'表示不要转义这个字符，是可选的。

/articles/2005/03/ 请求将匹配列表中的第三个模式。Django 将调用函数 views.month_archive(request, '2005', '03')。

2. 在 Python 正则表达式中，也可以给正则表达式命名一个组。命名正则表达式组的语法是(?P<name>pattern)，其中视图参数 name 是组的名称，pattern 是要匹配的模式。下面是第 1 点使用 URLconf 命名组的重写：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
```

```
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$',
views.article_detail),
]
```

说明：

- /articles/2005/03/ 请求将调用 views.month_archive(request, year='2005', month='03')函数，
- 如果是非命名正则表达式，则是调用 views.month_archive(request, '2005', '03')。注意参数是没有等号的。

3. 指定视图参数的默认值：

```
# URLconf
from django.conf.urls import url

from . import views

# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    ...
```

4. django.conf.urls.url()第三个参数是个字典，可选的，表示想要传递给视图函数的参数。

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

在这个例子中，对于/blog/2005/请求，Django 将调用 views.year_archive(request, year='2005', foo='bar')。

5. 在 Django 中返回 HTTP 错误代码是相当容易的。HttpResponse 的许多子类对应着除了 200（代表“OK”）以外的一些常用的 HTTP 状态码。

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

6. Django 提供了 Http404 异常。如果你在视图函数中的任何地方抛出 Http404 异常，Django 都会捕获它，并且带上 HTTP404 错误码返回你应用的标准错误页面。

```
from django.http import Http404
from django.shortcuts import render_to_response
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render_to_response('polls/detail.html', {'poll': p})
```

7. 当 Django 在处理文件上传的时候，文件数据被保存在 request.FILES。

   模型：

```
# In forms.py...
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

   视图：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
```

   这是一个普遍的方法，可能你会采用它来处理上传文件：

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

- request.method：是 HttpRequest 对象里的一个属性，存放的是 HTTP 的请求类型。
- request.FILES：表单上传的文件对象存储在类字典对象 request.FILES 中，表单格式需为 multipart/form-data
- chunks()：遍历文件，之所以不使用 read()，是因为确保大文件并不会占用系统过多的内存。

8. Django 函数：

(1) render()：结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的 HttpResponse 对象。

(2) render_to_response()：根据一个给定的上下文字典渲染一个给定的目标，并返回渲染后的 HttpResponse。

(3) redirect()：重定向，为传递进来的参数返回 HttpResponseRedirect 给正确的 URL。

(4) get_object_or_404()：在一个给定的模型管理器上调用 get()，但是引发 Http404，而不是模型的 DoesNotExist 异常。

(5) get_list_or_404()：返回一个列表，失败则返回 404。

## 认证系统

1. Django 的入口是 urls.py 文件。
2. auth 模块是 Django 提供的标准权限管理系统,可以提供用户身份认证, 用户组和权限管理。
3. @login_required 装饰器用于告诉程序，这个方法需要用户登陆。如果用户没有登陆，则重定向到 settings.LOGIN_URL，并将当前访问的绝对路径传递到查询字符串 next 中。

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

LOGIN_URL 在 settings 文件里设置。

4. @login_required 认证成功后，默认会跳转到 next 的参数的目录，如果想修改 next 的名字，如将 next 改为 redirect_field_name，可以给装饰器传递一个

redirect_field_name 参数：

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

同时还需要修改登陆模板中的 next 的名字。

5. 使用 authenticate()返回一个 User 对象。使用默认配置时参数是 username 和 password。如果匹配成功，返回一个 User 对象，失败则返回 None。

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    if user.is_active:
        print("User is valid, active and authenticated")
    else:
        print("The password is valid, but the account has been disabled!")
else:
    print("The username and password were incorrect.")
```

返回后可以使用 login()来登陆。

6. 每个用户是一个 User 对象。User 对象具有两个多对多的字段：groups 和 user_permissions：

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

User 对象可以使用 user_permissions 来对单个用户进行添加、删除、清空相应的权限。

附注：

- user.get_all_permissions()方法列出用户的所有权限，返回值是 permission name 的 list
- user.get_group_permissions()方法列出用户所属 group 的权限，返回值是 permission name 的 list

7. 使用 login()登陆，并添加到当前的会话中：

```
from django.contrib.auth import authenticate, login
```

```
def my_view(request):
  username = request.POST['username']
  password = request.POST['password']
  user = authenticate(username=username, password=password)
  if user is not None:
    if user.is_active:
      login(request, user)
      # Redirect to a success page.
    else:
      # Return a 'disabled account' error message
      ...
  else:
    # Return an 'invalid login' error message.
    ...
```

8. logout()用于退出登陆。

```
from django.contrib.auth import logout

def logout_view(request):
  logout(request)
  # Redirect to a success page.
```

logout_then_login 用于退出用户，并重定向到登录页面。

9. permission_required 装饰器用于检查用户是否具有特定的权限，格式如下：

```
permission_required([login_url=None, raise_exception=False]):
```

例如：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
  ...
```

权限格式形如“<app label>.<permission codename>”，如上面所示，表示 polls 应用下的 can_vote 权限。

10.除了上面提到的 login 和 logout，还有以下内建的视图函数

(1) logout_then_login()：注销一个用户然后重定向到一个登陆页面

(2) password_change()：允许用户修改他们的密码

(3) password_change_done()：用户修改密码后的页面

(4) password_reset()：通过生成的一个一次性的用来重置密码的发往他们注册邮箱的

链接来允许用户重置他们的密码

(5) password_reset_done()：重置密码后的页面

(6) redirect_to_login()：重定向到一个登陆页面然后在成功登陆后转向另一个 url

11. 当 django.contrib.auth 被加入 INSTALLED_APPS 的时候，添加、修改和删除三项权限已经为每个 django 模型创建好了。假设你有个应用的 app_label 是 foo，一个模型名为 Bar，那么你可以使用 has_perm()来测试用户是否拥有这三个权限：

```
user.has_perm('foo.add_bar')
user.has_perm('foo.change_bar')
user.has_perm('foo.delete_bar')
```

12. 要添加自定义权限，可以在 class Meta 中实现：

```
class Task(models.Model):
  ...
  class Meta:
    permissions = (
      ("view_task", "Can see available tasks"),
      ("change_task_status", "Can change the status of tasks"),
      ("close_task", "Can remove a task by setting its status as closed"),
    )
```

此示例任务模型创建三个自定义权限。创建完要运行 migrate 命令在数据库中创建这些权限。

## Django 模板语言

1. Django 模板语言和 Jinja2 差不多，Jinja2 是依照 Django 模板语言做出来的。
2. 模板渲染指通过使用指定值来替换模板中的变量，并执行所有的区块标签。
3. 一般情况下 context 来解析模板，context 有个特殊的子类，RequestContext，在相应的 views.py 文件中导入 RequestContext 后：

```
from django.template import RquestContext
```

可以在该文件对象所有模板中使用一些变量：

(1) 在模板中判断用户是否登陆，可以使用 is_authenticated：（is_authenticated、是否登陆

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
```

```
  <p>Welcome, new user. Please log in.</p>
{% endif %}
```

(2) 当前已经登陆的用户的权限被存在模板变量{{perms}}里面：

```
{% if perms.foo %}
  <p>You have permission to do something in the foo app.</p>
  {% if perms.foo.can_vote %}
    <p>You can vote!</p>
  {% endif %}
  {% if perms.foo.can_drive %}
    <p>You can drive!</p>
  {% endif %}
{% else %}
  <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

3. Django 模板使用双大括号{{}}来定义一个变量：

```
{{ variable }}
```

4. 点号（.）用来访问变量的属性。

5. 模版系统遇到点("."), 它将以这样的顺序查询：

(1) 字典查询

(2) 属性或方法查询

(3) 数字索引查询

要注意，如果一个变量的字典有一个默认值，则先执行字典查询，不会再去查找变量属性，

6. 过滤器用于过滤掉某些内容，过滤器格式如下：

```
{{ name|lower }}
```

这将在变量 {{ name }} 被过滤器 lower 过滤后再显示它的值，该过滤器将文本转换成小写。使用管道符号 (|)来应用过滤器。

7. 一些过滤器允许带有参数：

```
{{ bio|truncatewords:30 }}
```

显示 bio 变量的前 30 个词。

8. 过滤器参数包含空格的话，必须被引号包起来。例如，使用逗号和空格去连接一个列表中的元素，你需要使用 {{ list|join:", " }}。

9. Django 提供了大约六十个内置的模版过滤器。为了体验一下它们的作用，这里有一

些常用的模版过滤器：

(1) default：如果一个变量是 false 或者为空，使用给定的默认值。否则，使用变量的值。例如：

```
{{ value|default:"nothing" }}
```

如果 value 没有被提供，或者为空， 上面的例子将显示“nothing”。

(2) length：返回值的长度。它对字符串和列表都起作用。例如：

```
{{ value|length }}
```

(3) filesizeformat：将该数值格式化为一个 “人类可读的” 文件容量大小 （例如 '13 KB', '4.1 MB', '102 bytes', 等等）。例如：

```
{{ value|filesizeformat }}
```

10. 标签看起来像是这样的： {% tag %}。Django 自带了大约 24 个内置的模版标签。为了体验一下它们的作用，这里有一些常用的标签：

(1) for：循环数组中的每个元素。例如，显示 athlete_list 中提供的运动员列表：

```
<ul>
{% for athlete in athlete_list %}
  <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

(2) if, elif, and else：计算一个变量，并且当变量是“true”时，显示块中的内容：

```
{% if athlete_list %}
  Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
  Athletes should be out of the locker room soon!
{% else %}
  No athletes.
{% endif %}
```

11. 单行注释为# #：

```
{# {% if foo %}bar{% else %} #}
```

12. 模板多行注释使用 comment 标签：

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
  <p>Commented out text with {{ create_date|date:"c" }}</p>
```

```
{% endcomment %}
```

在{% comment "Optional note" %}和{% endcomment %}之间的都是注释。

13. 模板继承允许你构建一个包含你站点共同元素的基本模板“骨架”，并定义子模板可以覆盖的块。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  {% block head %}
  <link rel="stylesheet" href="style.css" />
  <title>{% block title %}{% endblock %} - My Webpage</title>
  {% endblock %}
</head>
<body>
  <div id="content">{% block content %}{% endblock %}</div>
  <div id="footer">
    {% block footer %}
    &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
    {% endblock %}
  </div>
</body>
```

block 标签告诉模板引擎子模板可以覆盖模板中的这些部分，子模板可以继承非 block 部分的内容。

block 头标签和尾标签之间的内容可以被覆盖。

子模版可能看起来是这样的：

```
{% extends "base.html" %}/span>

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
  <h2>{{ entry.title }}</h2>
  <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

14. {% extend %} 标签告诉模板引擎这个模板“继承自”另一个模板。extends 标签应该是模板中的第一个标签。

15. 如果存在着一些 block，子模版没有定义，则使用父模板的值。

16. 在 base 模版中设置越多的{% block %} 标签越好。请记住，子模版不必定义全部父模版中的 blocks。

17. 如果需要获取父模板中的 block 的内容，可以使用{{ block.super }} 变量。

18. 为了更好的可读性，你也可以给你的 {% endblock %} 标签一个名字。例如：

```
{% block content %}
...
{% endblock content %}
```

19. url 标签：用于移除模板文件中的固定的 url 链接地址：

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

“detail”是 urls.py 文件中的 name 标签，question.id 是 urls.py 括号里捕获的参数，这个参数需要给出：

```
url(r'^specifics/(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
```

表示修改成类似“polls/specifics/12/”格式的 URL。

20. 部分模板标签：

(1) cycle：每当这个标签被访问，则传出一个它的可迭代参数的元素。第一次访问返回第一个元素，第二次访问返回第二个参数，以此类推。一旦所有的变量都被访问过了，就会回到最开始的地方，重复下去。

```
{% for o in some_list %}
  <tr class="{% cycle 'row1' 'row2' %}">
    ...
  </tr>
{% endfor %}
```

(2) filter：通过一个或多个过滤器对内容过滤。 作为灵活可变的语法，多个过滤器被管道符号相连接，且过滤器可以有参数。

```
{% filter force_escape|lower %}
  This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

(3) firstof：输出第一个不为 False 参数。如果传入的所有变量都为 False，就什么也不输出。

```
{% firstof var1 var2 var3 %}
```

(4) for ... empty：for 标签带有一个可选的{% empty %} 从句，以便在给出的组是空的或者没有被找到时，可以有所操作。

```
<ul>
```

```
{% for athlete in athlete_list %}
   <li>{{ athlete.name }}</li>
{% empty %}
   <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

(5) if：{% if %}会对一个变量求值，如果它的值是“True”（存在、不为空、且不是boolean 类型的 false 值），这个内容块会输出。

(6) ifchanged：检查一个值是否在上一次的迭代中改变。

```
{% for match in matches %}
   <div style="background-color:
     {% ifchanged match.ballot_id %}
        {% cycle "red" "blue" %}
     {% else %}
        gray
     {% endifchanged %}
   ">{{ match }}</div>
{% endfor %}
```

(7) ifequal：如果给定的两个参数是相等的，则显示被标签包含的内容。

```
{% ifequal user.pk comment.user_id %}
   ...
{% endifequal %}
```

(8) ifnotequal：就像 ifequal，不过它测试两个参数不相等。

(9) include：加载模板并以标签内的参数渲染。这是一种可以引入别的模板的方法。

```
{% include "foo/bar.html" %}
```

(10) load：加载自定义模板标签集。

(11) now：显示最近的日期或事件，可以通过给定的字符串格式显示。

```
It is {% now "jS F Y H:i" %}
```

(12) spaceless：删除 HTML 标签之间的空格。包括制表符和换行。

```
{% spaceless %}
   <p>
     <a href="foo/">Foo</a>
```

```
  </p>
{% endspaceless %}
```

(13) with：使用一个简单的名字缓存一个复杂的变量，当你需要使用一个“昂贵的”方法（比如访问数据库）很多次的时候是非常有用的：

```
{% with total=business.employees.count %}
  {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

你可以分配多个上下文变量：

```
{% with alpha=1 beta=2 %}
  ...
{% endwith %}
```

21.为了避免 XXS 攻击，Django 默认开启自动转义：

(1) < 会转换为<

(2) > 会转换为>

(3) '(单引号) 会转换为'

(4) " (双引号)会转换为 "

(5) & 会转换为 &

22.如果打算渲染成原始 HTML 的数据，不想转义这些内容

(1) 可以使用 safe 来关闭自动转义：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

输出如下：

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

(2) 使用 autoescape 来关闭一段代码的自动转义：

```
{% autoescape off %}
  Hello {{ name }}
{% endautoescape %}
```

23.自动转义标签在 base 模板中关闭，它也会在 child 模板中关闭。

24.大多数对象上的方法调用同样可用于模板中。这意味着模板能够访问到的不仅仅是类属性（比如字段名称）和视图中传入的变量。例如，Django ORM 提供了“entry_set” 语法用于查找关联到外键的对象集合。所以，如果模型

“comment” 有一个外键关联到模型“task”，你可以根据 task 遍历其所有的 comments：

```
{% for comment in task.comment_set.all %}
  {{ comment }}
{% endfor %}
```

25. QuerySets 提供了 count()方法来计算含有对象的总数。因此，你可以像这样获取所有关于当前任务的评论总数：

```
{{ task.comment_set.all.count }}
```

26. 可以访问已经显式定义在模型上的方法：

```
models.py
class Task(models.Model):
  def foo(self):
    return "bar"
```

```
template.html
{{ task.foo }}
```

27. 某些应用提供自定义的标签和过滤器库。要在模板中访问它们，确保应用已经在 INSTALLED_APPS 中。例如，我们添加了'django.contrib.humanize'过滤器，之后在模板中使用 load 标签：

```
{% load humanize %}

{{ 45000|intcomma }}
```

28. load 标签可以接受多个库名称，由空格分隔。例如：

```
{% load humanize i18n %}
```