

环境安装

1. 创建虚拟环境

```
mkvirtualenv django_py3_1.11 -p python3
```

2. 安装 Django

使用 django 1.11.11 版本

```
pip install django==1.11.11
```

创建工程

1. 创建工程的命令为

```
django-admin startproject 工程名称
```

2. 工程目录说明

查看创建的工程目录，结构如下

```
(django1) python@ubuntu:~/Desktop/demo$ tree
.
├── demo
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py

1 directory, 5 files
```

与项目同名的目录，此处为 demo。

settings.py 是项目的整体配置文件。

urls.py 是项目的 URL 配置文件。

wsgi.py 是项目与 WSGI 兼容的 Web 服务器入口。

manage.py 是项目管理文件，通过它管理项目。

3. 运行开发服务器

在开发阶段，为了能够快速预览到开发的效果，django 提供了一个纯 python 编写的轻量级 web 服务器，仅在开发阶段使用。

运行服务器命令如下：

```
python manage.py runserver ip:端口
```

可以不写 IP 和端口，默认 IP 是 127.0.0.1，默认端口为 8000。

注：pycharm 中编译环境选择：home/python/.virtualenvs/虚拟环境

运行 manage.py 文件需要配置参数：runserver

创建子应用

在 Flask 框架中也有类似子功能应用模块的概念，即蓝图 Blueprint。

Django 的视图编写是放在子应用中的。

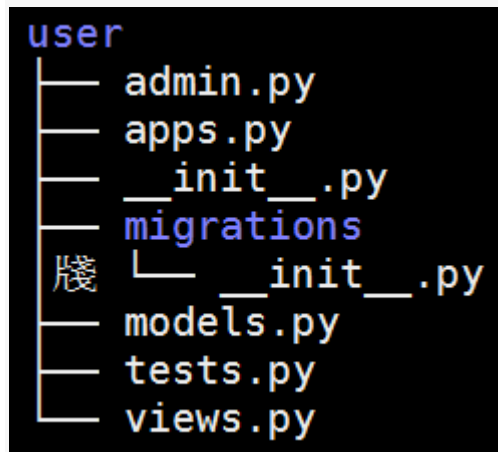
1. 创建

在 django 中，创建子应用模块目录仍然可以通过命令来操作，即：

```
python manage.py startapp 子应用名称
```

2. 子应用目录说明

查看此时的工程目录，结构如下：



admin.py 文件跟网站的后台管理站点配置相关。

apps.py 文件用于配置当前子应用的相关信息。

migrations 目录用于存放数据库迁移历史文件。

models.py 文件用户保存数据库模型类。

tests.py 文件用于开发测试用例，编写单元测试。

views.py 文件用于编写 Web 应用视图。

3. 注册安装子应用

创建出来的子应用目录文件虽然被放到了工程项目目录中，但是 django 工程并不能立即直接使用该子应用，需要注册安装后才能使用。

在工程配置文件 **settings.py** 中，**INSTALLED_APPS** 项保存了工程中已经注册安装的子应用。

注册安装一个子应用的方法，即是将子应用的配置信息文件 **apps.py** 中的 **Config** 类添加到 **INSTALLED_APPS** 列表中。

例如，将刚创建的子应用添加到工程中，可在 **INSTALLED_APPS** 列表中添加 '子应用名称.apps.UsersConfig'。

创建视图

1. 创建

打开刚创建的 **users** 模块，在 **views.py** 中编写视图代码。

```
from django.http import HttpResponse
```

```
def index(request):
```

```

"""
index 视图
:param request: 包含了请求信息的请求对象
:return: 响应对象
"""

return HttpResponse("hello the world!")

```

说明:

视图函数的第一个传入参数必须定义，用于接收 Django 构造的包含了请求数据的 **HttpRequest** 对象，通常名为 **request**。

视图函数的返回值必须为一个响应对象，不能像 Flask 一样直接返回一个字符串，可以将要返回的字符串数据放到一个 **HttpResponse** 对象中。

2. 定义路由 URL

- 1) 在子应用中新建一个 `urls.py` 文件用于保存该应用的路由。
- 2) 在 `users/urls.py` 文件中定义路由信息。

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

```
from . import views
```

`urlpatterns` 是被 django 自动识别的路由列表变量

```
urlpatterns = [
    # 每个路由信息都需要使用 url 函数来构造
    # url(路径, 视图)
    url(r'^index/$', views.index),
]
```

- 3) 在工程总路由 `demo/urls.py` 中添加子应用的路由数据。

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

```
urlpatterns = [
    url(r'^admin/', admin.site.urls), # django 默认包含的
    # 添加
    url(r'^users/', include('users.urls')),
]
```

使用 `include` 来将子应用 `users` 里的全部路由包含进工程路由中:

`r'^users/'` 决定了 `users` 子应用的所有路由都已 `/users/` 开头，如我们刚定义的视图 `index`，其最终的完整访问路径为 `/users/index/`。

- 4) 启动运行

重新启动 django 程序

配置文件

1. BASE_DIR

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
__file__: 当前文件
```

`os.path.abspath(__file__)`: 当前文件的绝对路径

`os.path.dirname(os.path.abspath(__file__))`: 当前文件的绝对路径的父文件名称

2. DEBUG

调试模式，创建工程后初始值为 **True**，即默认工作在调试模式下。

作用：

修改代码文件，程序自动重启

Django 程序出现异常时，向前端显示详细的错误追踪信息，

注意：部署线上运行的 Django 不要运行在调试模式下，记得修改 `DEBUG=False`。

3. 本地语言与时区

Django 支持本地化处理，即显示语言与时区支持本地化。

初始化的工程默认语言和时区为英语和 UTC 标准时区

```
LANGUAGE_CODE = 'en-us' # 语言
```

```
TIME_ZONE = 'UTC' # 时区
```

将语言和时区修改为中国大陆信息

```
LANGUAGE_CODE = 'zh-hans'
```

```
TIME_ZONE = 'Asia/Shanghai'
```

静态文件

静态文件可以放在项目根目录下，也可以放在应用的目录下，由于有些静态文件在项目中是通用的，所以推荐放在项目的根目录下，方便管理。

为了提供静态文件，需要配置两个参数：

STATICFILES_DIRS 存放查找静态文件的目录

STATIC_URL 访问静态文件的 URL 前缀

示例

1) 在项目根目录下创建 `static_files` 目录来保存静态文件。

2) 在 `demo/settings.py` 中修改静态文件的两个参数为

```
STATIC_URL = '/static/'
```

```
STATICFILES_DIRS = [  
    os.path.join(BASE_DIR, 'static_files'),  
]
```

3) 此时在 `static_files` 添加的任何静态文件都可以使用网址 `/static/文件在static_files中的路径` 来访问了。

注意

Django 仅在调试模式下 (`DEBUG=True`) 能对外提供静态文件。

当 `DEBUG=False` 工作在生产模式时，Django 不再对外提供静态文件，需要是用 `collectstatic` 命令来收集静态文件并交由其他静态文件服务器来提供。

路由说明

1. 路由定义位置

Django 的主要路由信息定义在工程同名目录下的 `urls.py` 文件中，该文件是 Django 解析路由的入口。

每个子应用为了保持相对独立，可以在各个子应用中定义属于自己的 `urls.py` 来保存该应用的路由。然后用主路由文件包含各应用的子路由数据。

2. 路由解析顺序

Django 在接收到一个请求时，从主路由文件中的 `urlpatterns` 列表中以由上至下的顺序查找对应路由规则，如果发现规则为 `include` 包含，则再进入被包含的 `urls` 中的 `urlpatterns` 列表由上至下进行查询。

值得关注的由上至下的顺序，有可能会使上面的路由屏蔽掉下面的路由，带来非预期结果。

提示：需要注意定义路由的顺序，避免出现屏蔽效应。

3. 路由命名

在定义路由的时候，可以为路由命名，方便查找特定视图的具体路径信息。

1) 在使用 `include` 函数定义路由时，可以使用 `namespace` 参数定义路由的命名空间，如

```
url(r'^users/', include('users.urls', namespace='users')),
```

命名空间的作用：避免不同应用中的路由使用了相同的名字发生冲突，使用命名空间区别开。

2) 在定义普通路由时，可以使用 `name` 参数指明路由的名字，如

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

4. reverse 反解析

使用 `reverse` 函数，可以根据路由名称，返回具体的路径，如：

```
from django.core.urlresolvers import reverse # 注意导包路径  
  
def index(request):  
    url = reverse('users:index') # 返回 /users/index/  
    print(url)  
    return HttpResponse("hello the world!")
```

App 应用配置

`AppConfig.verbose_name` 属性用于设置该应用的直观可读的名字，此名字在 Django 提供的 Admin 管理站点中会显示，如

```
from django.apps import AppConfig
```

```
class UsersConfig(AppConfig):  
    name = 'users'  
    verbose_name = '用户管理'
```

请求

利用 HTTP 协议向服务器传参有几种途径？

提取 URL 的特定部分，如 `/weather/beijing/2018`，可以在服务器端的路由中用正则表达式截取：

查询字符串（`query string`），形如 `key1=value1&key2=value2`；

请求体（`body`）中发送的数据，比如表单数据、`json`、`xml`；

在 `http` 报文的头（`header`）中。

1. URL 路径参数

在定义路由 URL 时，可以使用正则表达式提取参数的方法从 URL 中获取请求参数，Django 会将提取的参数直接传递到视图的传入参数中。

- 未命名参数按定义顺序传递，如

```
url(r'^weather/([a-z]+)/(\d{4})/$', views.weather),
```

```
def weather(request, city, year):
```

```
    print('city=%s' % city)
```

```
    print('year=%s' % year)
```

```
    return HttpResponse('OK')
```

- 命名参数按名字传递，如

```
url(r'^weather/(?P<city>[a-z]+)/(?P<year>\d{4})/$', views.weather),
```

```
def weather(request, year, city):
```

```
    print('city=%s' % city)
```

```
    print('year=%s' % year)
```

```
    return HttpResponse('OK')
```

2. Django 中的 QueryDict 对象

与 python 字典不同，`QueryDict` 类型的对象用来处理同一个键带有多个值的情况

- 方法 `get()`：根据键获取值

如果一个键同时拥有多个值将获取最后一个值

如果键不存在则返回 `None` 值，可以设置默认值进行后续处理

```
dict.get('键', 默认值)
```

可简写为

```
dict['键']
```

- 方法 `getlist()`：根据键获取值，值以列表返回，可以获取指定键的所有值

如果键不存在则返回空列表 `[]`，可以设置默认值进行后续处理

```
dict.getlist('键', 默认值)
```

3. 查询字符串 Query String

获取请求路径中的查询字符串参数（形如 `?k1=v1&k2=v2`），可以通过 `request.GET` 属性获取，返回 `QueryDict` 对象。

```
# /qs/?a=1&b=2&a=3
```

```
def qs(request):
```

```
    a = request.GET.get('a')
```

```
    b = request.GET.get('b')
```

```
    alist = request.GET.getlist('a')
```

```
    print(a) # 3
```

```
print(b) # 2
print(alist) # ['1', '3']
return HttpResponse('OK')
```

重要：查询字符串不区分请求方式，即使客户端进行 **POST** 方式的请求，依然可以通过 **request.GET** 获取请求中的查询字符串数据。

4 请求体

请求体数据格式不固定，可以是表单类型字符串，可以是 JSON 字符串，可以是 XML 字符串，应区别对待。

可以发送请求体数据的请求方式有 **POST**、**PUT**、**PATCH**、**DELETE**。

Django 默认开启了 **CSRF 防护**，会对上述请求方式进行 CSRF 防护验证，在测试时可以关闭 CSRF 防护机制，方法为在 `settings.py` 文件中注释掉 CSRF 中间件。

4.1 表单类型 Form Data

前端发送的表单类型的请求体数据，可以通过 `request.POST` 属性获取，返回 `QueryDict` 对象。

```
def get_body(request):
    a = request.POST.get('a')
    b = request.POST.get('b')
    alist = request.POST.getlist('a')
    print(a)
    print(b)
    print(alist)
    return HttpResponse('OK')
```

重要：只要请求体的数据是表单类型，无论是哪种请求方式（**POST**、**PUT**、**PATCH**、**DELETE**），都是使用 **request.POST** 来获取请求体的表单数据。

4.2 非表单类型 Non-Form Data

非表单类型的请求体数据，**Django** 无法自动解析，可以通过 `request.body` 属性获取最原始的请求体数据，自己按照请求体格式（JSON、XML 等）进行解析。`request.body` 返回 **bytes** 类型。

例如要获取请求体中的如下 JSON 数据 `{"a": 1, "b": 2}`

可以进行如下方法操作：

```
import json

def get_body_json(request):
    json_str = request.body
    json_str = json_str.decode() # python3.6 无需执行此步
    req_data = json.loads(json_str)
    print(req_data['a'])
    print(req_data['b'])
    return HttpResponse('OK')
```

5 请求头

可以通过 `request.META` 属性获取请求头 `headers` 中的数据，`request.META` 为字典类型。

具体使用如：

```
def get_headers(request):  
    print(request.META['CONTENT_TYPE'])  
    return HttpResponse('OK')
```

响应

视图在接收请求并处理后，必须返回 `HttpResponse` 对象或子对象。`HttpRequest` 对象由 Django 创建，`HttpResponse` 对象由开发人员创建。

1 HttpResponse

可以使用 `django.http.HttpResponse` 来构造响应对象。

`HttpResponse` (content=响应体, content_type=响应体数据类型, status=状态码)

也可通过 `HttpResponse` 对象属性来设置响应体、响应体数据类型、状态码：

content: 表示返回的内容。

status_code: 返回的 HTTP 响应状态码。

content_type: 指定返回数据的 MIME 类型。

响应头可以直接将 `HttpResponse` 对象当做字典进行响应头键值对的设置：

```
response = HttpResponse()
```

```
response['Itcast'] = 'Python' # 自定义响应头 Itcast, 值为 Python
```

示例：

```
from django.http import HttpResponse
```

```
def demo_view(request):  
    return HttpResponse('itcast python', status=400)  
或者  
response = HttpResponse('itcast python')  
response.status_code = 400  
response['Itcast'] = 'Python'  
return response
```

2 HttpResponse 子类

Django 提供了一系列 `HttpResponse` 的子类，可以快速设置状态码

```
HttpResponseRedirect 301  
HttpResponsePermanentRedirect 302  
HttpResponseNotModified 304  
HttpResponseBadRequest 400  
HttpResponseNotFound 404  
HttpResponseForbidden 403  
HttpResponseNotAllowed 405  
HttpResponseGone 410  
HttpResponseServerError 500
```

3 JsonResponse

若要返回 json 数据，可以使用 `JsonResponse` 来构造响应对象，作用：

帮助我们将数据转换为 json 字符串

设置响应头 **Content-Type** 为 **application/json**

```
from django.http import JsonResponse
```

```
def demo_view(request):  
    return JsonResponse({'city': 'beijing', 'subject': 'python'})
```

4 redirect 重定向

```
from django.shortcuts import redirect
```

```
def demo_view(request):  
    return redirect('/index.html')
```

Cookie

Cookie 的特点

1. Cookie 以键值对的格式进行信息的存储。
2. Cookie 基于域名安全，不同域名的 Cookie 是不能互相访问的
3. 当浏览器请求某网站时，会将浏览器存储的跟网站相关的所有 Cookie 信息提交给网站服务器。

1 设置 Cookie

可以通过 **HttpResponse** 对象中的 **set_cookie** 方法来设置 cookie。

HttpResponse.set_cookie(cookie 名, value=cookie 值, max_age=cookie 有效期)

max_age 单位为秒，默认为 **None**。如果是临时 cookie，可将 **max_age** 设置为 **None**。

示例：

```
def demo_view(request):  
    response = HttpResponse('ok')  
    response.set_cookie('itcast1', 'python1') # 临时 cookie  
    response.set_cookie('itcast2', 'python2', max_age=3600) # 有效期一小时  
    return response
```

2 读取 Cookie

可以通过 **HttpRequest** 对象的 **COOKIES** 属性来读取本次请求携带的 cookie 值。

request.COOKIES 为字典类型。

```
def demo_view(request):  
    cookie1 = request.COOKIES.get('itcast1')  
    print(cookie1)  
    return HttpResponse('OK')
```

Session

1 启用 Session

Django 项目默认启用 **Session**。

如需禁用 session，将 session 中间件注释掉即可。

2 存储方式

在 settings.py 文件中，可以设置 session 数据的存储方式，可以保存在数据库、本地缓存等。


2.1 数据库

存储在数据库中，如下设置可以写，也可以不写，这是默认存储方式。

```
SESSION_ENGINE='django.contrib.sessions.backends.db'
```


如果存储在数据库中，需要在项 INSTALLED_APPS 中安装 Session 应用。

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



数据库中的表如图所示

```
mysql> show tables;  
+-----+  
| Tables_in_test2 |  
+-----+  
| auth_group       |  
| auth_group_permissions |  
| auth_permission  |  
| auth_user        |  
| auth_user_groups  |  
| auth_user_user_permissions |  
| bookinfo         |  
| booktest_areainfo |  
| booktest_heroinfo |  
| django_admin_log  |  
| django_content_type |  
| django_migrations |  
| django_session    |  
+-----+  
13 rows in set (0.00 sec)
```



表结构如下

```
mysql> desc django_session;  
+-----+-----+-----+-----+-----+-----+  
| Field      | Type      | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+-----+  
| session_key | varchar(40) | NO   | PRI | NULL    |       |  
| session_data | longtext   | NO   |     | NULL    |       |  
| expire_date | datetime(6) | NO   | MUL | NULL    |       |  
+-----+-----+-----+-----+-----+-----+  
3 rows in set (0.04 sec)
```

由表结构可知，操作 Session 包括三个数据：键，值，过期时间。

2.2 本地缓存

存储在本机内存中，如果丢失则不能找回，比数据库的方式读写更快。

```
SESSION_ENGINE='django.contrib.sessions.backends.cache'
```

2.3 混合存储

优先从本机内存中存取，如果没有则从数据库中存取。

```
SESSION_ENGINE='django.contrib.sessions.backends.cached_db'
```

2.4 Redis

在 redis 中保存 session，需要引入第三方扩展，我们可以使用 **django-redis** 来解决。

1) 安装扩展

```
pip install django-redis
```

2) 配置

在 settings.py 文件中做如下设置

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    }
}

SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"
```

注意

如果 redis 的 ip 地址不是本地回环 127.0.0.1，而是其他地址，访问 Django 时，可能出现 Redis 连接错误，如下：

```
ConnectionError at /response/
Error 61 connecting to 10.211.55.5:6379. Connection refused.

Request Method: GET
Request URL: http://127.0.0.1:8000/response/
Django Version: 1.11.11
Exception Type: ConnectionError
Exception Value: Error 61 connecting to 10.211.55.5:6379. Connection refused.
Exception Location: /Users/delron/.virtualenv/django_py3_1.11/lib/python3.6/site-packages/redis/connection.py in connect, line 489
Python Executable: /Users/delron/.virtualenv/django_py3_1.11/bin/python
Python Version: 3.6.2
Python Path: ['/Users/delron/Desktop/code/demo',
              '/Users/delron/Desktop/code/demo',
              '/Users/delron/.virtualenv/django_py3_1.11/lib/python3.6.zip',
              '/Users/delron/.virtualenv/django_py3_1.11/lib/python3.6',
              '/Users/delron/.virtualenv/django_py3_1.11/lib/python3.6/lib-dynload',
              '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6/lib/python3.6',
              '/Users/delron/.virtualenv/django_py3_1.11/lib/python3.6/site-packages',
              '/Applications/PyCharm.app/Contents/helpers/pycharm_matplotlib_backend']
Server time: 星期六, 5 五月 2018 07:08:32 +0800
```

解决方法：


修改 redis 的配置文件，添加特定 ip 地址。

打开 redis 的配置文件

```
sudo vim /etc/redis/redis.conf
```

在如下配置项进行修改（如要添加 10.211.55.5 地址）

```
61 # By default Redis listens for connections from all the network interfaces
62 # available on the server. It is possible to listen to just one or multiple
63 # interfaces using the "bind" configuration directive, followed by one or
64 # more IP addresses.
65 #
66 # Examples:
67 #
68 # bind 192.168.1.100 10.0.0.1
69 # bind 127.0.0.1
70 bind 127.0.0.1 10.211.55.5
71
```



重新启动 redis 服务

```
sudo service redis-server restart
```

3 Session 操作

通过 HttpRequest 对象的 session 属性进行会话的读写操作。

1) 以键值对的格式写 session。

```
request.session['键']=值
```

2) 根据键读取值。

```
request.session.get('键',默认值)
```

3) 清除所有 session，在存储中删除值部分。

```
request.session.clear()
```

4) 清除 session 数据，在存储中删除 session 的整条数据。

```
request.session.flush()
```

5) 删除 session 中的指定键及值，在存储中只删除某个键及对应的值。

```
del request.session['键']
```

6) 设置 session 的有效期

```
request.session.set_expiry(value)
```

如果 value 是一个整数，session 将在 value 秒没有活动后过期。

如果 value 为 0，那么用户 session 的 Cookie 将在用户的浏览器关闭时过期。

如果 value 为 None，那么 session 有效期将采用系统默认值，默认为两周，可以通过在 settings.py 中设置 **SESSION_COOKIE_AGE** 来设置全局默认值。

类视图

1 类视图引入

以函数的方式定义的视图称为**函数视图**。但是代码可读性与复用性都不佳。

在 Django 中也可以使用类来定义一个视图，称为**类视图**。

使用类视图可以将视图对应的不同请求方式以类中的不同方法来区别定义。如下所示

```
from django.views.generic import View
```

```
class RegisterView(View):
    """类视图：处理注册"""

    def get(self, request):
        """处理 GET 请求，返回注册页面"""
        return render(request, 'register.html')
```

```
def post(self, request):  
    """处理 POST 请求，实现注册逻辑"""  
    return HttpResponse('这里实现注册逻辑')
```

类视图的好处：

1. 代码可读性好
2. 类视图相对于函数视图有更高的复用性，如果其他地方需要用到某个类视图的某个特定逻辑，直接继承该类视图即可

2 类视图使用

定义类视图需要继承自 Django 提供的父类 View，可使用 `from django.views.generic import View` 或者 `from django.views.generic.base import View` 导入。

配置路由时，使用类视图的 `as_view()` 方法来添加。

```
urlpatterns = [  
    # 视图函数：注册  
    # url(r'^register/$', views.register, name='register'),  
    # 类视图：注册  
    url(r'^register/$', views.RegisterView.as_view(), name='register'),  
]
```

3 类视图原理

```

@classonlymethod
def as_view(cls, **initkwargs):
    """
    Main entry point for a request-response process.
    """
    ...省略代码...

    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        # 调用dispatch方法，按照不同请求方式调用不同请求方法
        return self.dispatch(request, *args, **kwargs)

    ...省略代码...

    # 返回真正的函数视图
    return view

def dispatch(self, request, *args, **kwargs):
    # Try to dispatch to the right method; if a method doesn't exist,
    # defer to the error handler. Also defer to the error handler if the
    # request method isn't on the approved list.
    if request.method.lower() in self.http_method_names:
        handler = getattr(self, request.method.lower(), self.http_method_not_allowed)
    else:
        handler = self.http_method_not_allowed
    return handler(request, *args, **kwargs)

```

4 类视图使用装饰器

为类视图添加装饰器，可以使用三种方法。

为了理解方便，我们先来定义一个**为函数视图准备的装饰器**（在设计装饰器时基本都以函数视图作为考虑的被装饰对象），及一个要被装饰的类视图。

```

def my_decorator(func):
    def wrapper(request, *args, **kwargs):

```

```

        print('自定义装饰器被调用了')
        print('请求路径%s' % request.path)
        return func(request, *args, **kwargs)
    return wrapper

class DemoView(View):
    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')

```

4.1 在 URL 配置中装饰

```

urlpatterns = [
    url(r'^demo/$', my_decorate(DemoView.as_view()))
]

```

此种方式最简单，但因装饰行为被放置到了 `url` 配置中，单看视图的时候无法知道此视图还被添加了装饰器，不利于代码的完整性，不建议使用。

此种方式会为类视图中的所有请求方法都加上装饰器行为（因为是在视图入口处，分发请求方式前）。

4.2 在类视图中装饰

在类视图中使用为函数视图准备的装饰器时，不能直接添加装饰器，需要使用 **method_decorator** 将其转换为适用于类视图方法的装饰器。

```

from django.utils.decorators import method_decorator

```

为全部请求方法添加装饰器

```

class DemoView(View):
    @method_decorator(my_decorator)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)

    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')

```

为特定请求方法添加装饰器

```

class DemoView(View):

```

```
@method_decorator(my_decorator)
def get(self, request):
    print('get 方法')
    return HttpResponse('ok')
```

```
def post(self, request):
    print('post 方法')
    return HttpResponse('ok')
```

method_decorator 装饰器还支持使用 **name** 参数指明被装饰的方法

为全部请求方法添加装饰器

```
@method_decorator(my_decorator, name='dispatch')
```

```
class DemoView(View):
    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')
```

为特定请求方法添加装饰器

```
@method_decorator(my_decorator, name='get')
```

```
class DemoView(View):
    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')
```

为什么需要使用 **method_decorator**???

为函数视图准备的装饰器，其被调用时，第一个参数用于接收 **request** 对象

```
def my_decorate(func):
    def wrapper(request, *args, **kwargs): # 第一个参数 request 对象
        ...代码省略...
    return func(request, *args, **kwargs)
return wrapper
```

而类视图中请求方法被调用时，传入的第一个参数不是 **request** 对象，而是 **self** 视图对象本身，第二个位置参数才是 **request** 对象

```
class DemoView(View):
    def dispatch(self, request, *args, **kwargs):
        ...代码省略...

    def get(self, request):
```


...代码省略...

所以如果直接将用于函数视图的装饰器装饰类视图方法，会导致参数传递出现问题。

method_decorator 的作用是为函数视图装饰器补充第一个 **self** 参数，以适配类视图方法。

如果将装饰器本身改为可以适配类视图方法的，类似如下，则无需再使用 **method_decorator**。

```
def my_decorator(func):  
    def wrapper(self, request, *args, **kwargs): # 此处增加了 self  
        print('自定义装饰器被调用了')  
        print('请求路径%s' % request.path)  
        return func(self, request, *args, **kwargs) # 此处增加了 self  
    return wrapper
```

4.3 构造 Mixin 扩展类

使用面向对象多继承的特性。

```
class MyDecoratorMixin(object):  
    @classmethod  
    def as_view(cls, *args, **kwargs):  
        return my_decorator(super().as_view(*args, **kwargs))  
  
class DemoView(MyDecoratorMixin, View):  
    def get(self, request):  
        print('get 方法')  
        return HttpResponse('ok')  
  
    def post(self, request):  
        print('post 方法')  
        return HttpResponse('ok')
```

使用 **Mixin** 扩展类，也会为类视图的所有请求方法都添加装饰行为。

中间件

我们可以使用中间件，在 Django 处理视图的不同阶段对输入或输出进行干预。

1 中间件的定义方法

定义一个中间件工厂函数，然后返回一个可以被调用的中间件。

中间件工厂函数需要接收一个可以调用的 **get_response** 对象。

返回的中间件也是一个可以被调用的对象，并且像视图一样需要接收一个 **request** 对象参数，返回一个 **response** 对象。

```
def simple_middleware(get_response):  
    # 此处编写的代码仅在 Django 第一次配置和初始化的时候执行一次。  
  
    def middleware(request):  
        # 此处编写的代码会在每个请求处理视图前被调用。  
  
        response = get_response(request)
```

此处编写的代码会在每个请求处理视图之后被调用。

```
return response
```

```
return middleware
```

例如，在 users 应用中新建一个 middleware.py 文件，

```
def my_middleware(get_response):  
    print('init 被调用')  
    def middleware(request):  
        print('before request 被调用')  
        response = get_response(request)  
        print('after response 被调用')  
        return response  
    return middleware
```

定义好中间件后，需要在 **settings.py** 文件中添加注册中间件

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    # 'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    'users.middleware.my_middleware', # 添加中间件  
]
```

注意：Django 运行在调试模式下，中间件 **init** 部分有可能被调用两次。

2 多个中间件的执行顺序

在请求视图被处理前，中间件由上至下依次执行

在请求视图被处理后，中间件由下至上依次执行

示例：

定义两个中间件

```
def my_middleware(get_response):  
    print('init 被调用')  
    def middleware(request):  
        print('before request 被调用')  
        response = get_response(request)  
        print('after response 被调用')  
        return response  
    return middleware
```

```
def my_middleware2(get_response):  
    print('init2 被调用')  
    def middleware(request):
```

```

        print('before request 2 被调用')
        response = get_response(request)
        print('after response 2 被调用')
        return response
    return middleware

```

注册添加两个中间件

```

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'users.middleware.my_middleware', # 添加
    'users.middleware.my_middleware2', # 添加
]

```

执行结果

init2 被调用

init 被调用

before request 被调用

before request 2 被调用

view 视图被调用

after response 2 被调用

after response 被调用

数据库

ORM 框架

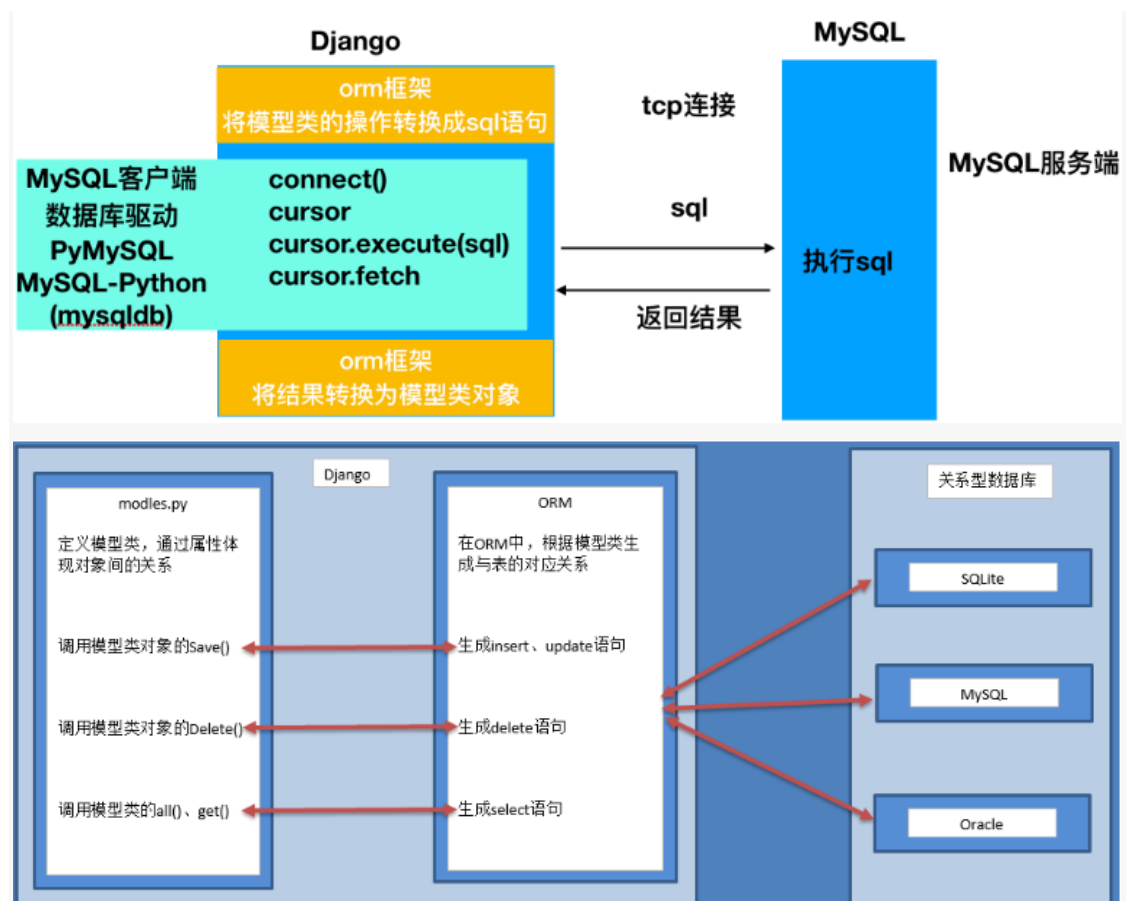
O 是 object，也就**类对象**的意思，R 是 relation，翻译成中文是关系，也就是关系数据库中**数据表**的意思，M 是 mapping，是**映射**的意思。在 ORM 框架中，它帮我们把类和数据库表进行了一个映射，可以让我们**通过类和类对象就能操作它所对应的表格中的数据**。ORM 框架还有一个功能，它可以**根据我们设计的类自动帮我们生成数据库中的表格**，省去了我们自己建表的过程。

django 中内嵌了 ORM 框架，不需要直接面向数据库编程，而是定义模型类，通过模型类和对象完成数据表的增删改查操作。

使用 django 进行数据库开发的步骤如下：

1. 配置数据库连接信息
2. 在 models.py 中定义模型类
3. 迁移
4. 通过类和对象完成数据增删改查操作

ORM 作用



配置

使用 **MySQL** 数据库首先需要安装驱动程序

```
pip install PyMySQL
```

在 Django 的工程同名子目录的 `__init__.py` 文件中添加如下语句

```
from pymysql import install_as_MySQLdb
install_as_MySQLdb()
```

作用是让 Django 的 ORM 能以 `mysqlldb` 的方式来调用 PyMySQL。

修改 **DATABASES** 配置信息

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': '127.0.0.1', # 数据库主机
        'PORT': 3306, # 数据库端口
        'USER': 'root', # 数据库用户名
        'PASSWORD': 'mysql', # 数据库用户密码
        'NAME': 'django_demo' # 数据库名字
    }
}
```

在 MySQL 中创建数据库

```
create database django_demo default charset=utf8;
```

定义模型类

模型类被定义在"应用/models.py"文件中。
模型类必须继承自 Model 类，位于包 django.db.models 中。
接下来首先以"图书-英雄"管理为例进行演示。

1 定义

创建应用 booktest，在 models.py 文件中定义模型类。

```
from django.db import models

#定义图书模型类 BookInfo
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期')
    bread = models.IntegerField(default=0, verbose_name='阅读量')
    bcomment = models.IntegerField(default=0, verbose_name='评论量')
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_books' # 指明数据库表名
        verbose_name = '图书' # 在 admin 站点中显示的名称
        verbose_name_plural = verbose_name # 显示的复数名称

    def __str__(self):
        """定义每个数据对象的显示信息"""
        return self.btitle

#定义英雄模型类 HeroInfo
class HeroInfo(models.Model):
    GENDER_CHOICES = (
        (0, 'male'),
        (1, 'female')
    )
    hname = models.CharField(max_length=20, verbose_name='名称')
    hgender = models.SmallIntegerField(choices=GENDER_CHOICES, default=0,
    verbose_name='性别')
    hcomment = models.CharField(max_length=200, null=True, verbose_name='
描述信息')
    hbook = models.ForeignKey(BookInfo, on_delete=models.CASCADE,
    verbose_name='图书') # 外键
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_heros'
        verbose_name = '英雄'
        verbose_name_plural = verbose_name
```

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

1) 数据库表名

模型类如果未指明表名,Django 默认以 **小写 app 应用名_小写模型类名** 为数据库表名。
可通过 **db_table** 指明数据库表名。

2) 关于主键

django 会为表创建自动增长的主键列,每个模型只能有一个主键列,如果使用选项设置某属性为主键列后 django 不会再创建自动增长的主键列。

默认创建的主键列属性为 **id**,可以使用 **pk** 代替,**pk** 全拼为 **primary key**。

3) 属性命名限制

不能是 python 的保留关键字。

不允许使用连续的下划线,这是由 django 的查询方式决定的。

定义属性时需要指定字段类型,通过字段类型的参数指定选项,语法如下:

属性=models. 字段类型(选项)

4) 外键

在设置外键时,需要通过 **on_delete** 选项指明主表删除数据时,对于外键引用表数据如何处理,在 **django.db.models** 中包含了可选常量:

CASCADE 级联,删除主表数据时连通一起删除外键表中数据

PROTECT 保护,通过抛出 **ProtectedError** 异常,来阻止删除主表中被外键应用的数据

2 迁移

将模型类同步到数据库中。

1) 生成迁移文件

```
python manage.py makemigrations
```

2) 同步到数据库中

```
python manage.py migrate
```

3 添加测试数据

```
insert into tb_books(btitle,bpub_date,bread,bcomment,is_delete) values  
( '射雕英雄传', '1980-5-1',12,34,0),  
( '天龙八部', '1986-7-24',36,40,0),  
( '笑傲江湖', '1995-12-24',20,80,0),  
( '雪山飞狐', '1987-11-11',58,24,0);  
insert into tb_heros(hname,hgender,hbook_id,hcomment,is_delete) values  
( '郭靖',1,1,'降龙十八掌',0),  
( '黄蓉',0,1,'打狗棍法',0),  
( '黄药师',1,1,'弹指神通',0),  
( '欧阳锋',1,1,'蛤蟆功',0),  
( '梅超风',0,1,'九阴白骨爪',0),  
( '乔峰',1,2,'降龙十八掌',0),  
( '段誉',1,2,'六脉神剑',0),  
( '虚竹',1,2,'天山六阳掌',0),  
( '王语嫣',0,2,'神仙姐姐',0),  
( '令狐冲',1,3,'独孤九剑',0),  
( '任盈盈',0,3,'弹琴',0),
```

```
('岳不群',1,3,'华山剑法',0),
('东方不败',0,3,'葵花宝典',0),
('胡斐',1,4,'胡家刀法',0),
('苗若兰',0,4,'黄衣',0),
('程灵素',0,4,'医术',0),
('袁紫衣',0,4,'六合拳',0);
```

演示工具使用

1 shell 工具

Django 的 manage 工具提供了 **shell** 命令，帮助我们配置好当前工程的运行环境（如连接好数据库等），以便可以直接在终端中执行测试 python 语句。

通过如下命令进入 shell

```
python manage.py shell
```

导入两个模型类，以便后续使用

```
from booktest.models import BookInfo, HeroInfo
```

2 查看 MySQL 数据库日志

查看 mysql 数据库日志可以查看对数据库的操作记录。mysql 日志文件默认没有产生，需要做如下配置：

```
sudo vi /etc/mysql/mysql.conf.d/mysqld.cnf
```

把 68, 69 行前面的#去除，然后保存并使用如下命令重启 mysql 服务。

```
sudo service mysql restart
```

使用如下命令打开 mysql 日志文件。

```
tail -f /var/log/mysql/mysql.log # 可以实时查看数据库的日志内容
```

数据库操作一增、删、改、查

1 增加

增加数据有两种方法。

1) save

通过创建模型类对象，执行对象的 save() 方法保存到数据库中。

```
>>> from datetime import date
>>> book = BookInfo(
    btitle='西游记',
    bput_date=date(1988,1,1),
    bread=10,
    bcomment=10
)
>>> book.save()
>>> hero = HeroInfo(
    hname='孙悟空',
    hgender=0,
    hbook=book
)
```

```
>>> hero.save()
>>> hero2 = HeroInfo(
    hname='猪八戒',
    hgender=0,
    hbook_id=book.id
)
>>> hero2.save()
```

2) create

通过模型类.objects.create()保存。

```
>>> HeroInfo.objects.create(
    hname='沙悟净',
    hgender=0,
    hbook=book
)
```

<HeroInfo: 沙悟净>

2 查询

2.1 基本查询

get 查询单一结果，如果不存在会抛出模型类.DoesNotExist 异常。

all 查询多个结果。

count 查询结果数量。

```
>>> BookInfo.objects.all()
<QuerySet [<BookInfo: 射雕英雄传>, <BookInfo: 天龙八部>, <BookInfo: 笑傲江湖>,
<BookInfo: 雪山飞狐>, <BookInfo: 西游记>]>
>>> book = BookInfo.objects.get(btitle='西游记')
>>> book.id
5
```

```
>>> BookInfo.objects.get(id=3)
<BookInfo: 笑傲江湖>
>>> BookInfo.objects.get(pk=3)
<BookInfo: 雪山飞狐>
>>> BookInfo.objects.get(id=100)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File
"/Users/delron/.virtualenv/dj/lib/python3.6/site-packages/django/db/models/manager.py", line 85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File
"/Users/delron/.virtualenv/dj/lib/python3.6/site-packages/django/db/models/query.py", line 380, in get
    self.model._meta.object_name
db.models.DoesNotExist: BookInfo matching query does not exist.
```



```
>>> BookInfo.objects.count()
```

6

2.2 过滤查询

实现 SQL 中的 where 功能，包括

1. **filter** 过滤出多个结果
2. **exclude** 排除掉符合条件剩下的结果
3. **get** 过滤单一结果

对于过滤条件的使用，上述三个方法相同，故仅以 **filter** 进行讲解。

过滤条件的表达语法如下：

属性名称__比较运算符=值

属性名称和比较运算符间使用两个下划线，所以属性名不能包括多个下划线

1) 相等

exact: 表示判等。

例：查询编号为 1 的图书。

```
BookInfo.objects.filter(id__exact=1)
```

可简写为：

```
BookInfo.objects.filter(id=1)
```

2) 模糊查询

contains: 是否包含。

说明：如果要包含%无需转义，直接写即可。

例：查询书名包含'传'的图书。

```
BookInfo.objects.filter(btitle__contains='传')
```

startswith、endswith: 以指定值开头或结尾。

例：查询书名以'部'结尾的图书

```
BookInfo.objects.filter(btitle__endswith='部')
```

以上运算符都区分大小写，在这些运算符前加上 **i** 表示不区分大小写，如 **iexact**、**icontains**、**istartswith**、**iendswith**。

3) 空查询

isnull: 是否为 null。

例：查询书名不为空的图书。

```
BookInfo.objects.filter(btitle__isnull=False)
```

4) 范围查询

in: 是否包含在范围内。

例：查询编号为 1 或 3 或 5 的图书

```
BookInfo.objects.filter(id__in=[1, 3, 5])
```

5) 比较查询

1. **gt** 大于 (greater then)
2. **gte** 大于等于 (greater then equal)
3. **lt** 小于 (less then)
4. **lte** 小于等于 (less then equal)

例：查询编号大于 3 的图书

```
BookInfo.objects.filter(id__gt=3)
```

不等于的运算符，使用 **exclude()** 过滤器。

例：查询编号不等于 3 的图书

```
BookInfo.objects.exclude(id=3)
```

6) 日期查询

year、month、day、week_day、hour、minute、second：对日期时间类型的属性进行运算。

例：查询 1980 年发表的图书。

```
BookInfo.objects.filter(bpub_date__year=1980)
```

例：查询 1980 年 1 月 1 日后发表的图书。

```
BookInfo.objects.filter(bpub_date__gt=date(1980, 1, 1))
```

F 对象（两个属性比较）

之前的查询都是对象的属性与常量值比较，两个属性怎么比较呢？

答：使用 F 对象，被定义在 `django.db.models` 中。

语法如下：

F(属性名)

例：查询阅读量大于等于评论量的图书。

```
from django.db.models import F
```

```
BookInfo.objects.filter(bread__gte=F('bcomment'))
```

可以在 F 对象上使用算数运算。

例：查询阅读量大于 2 倍评论量的图书。

```
BookInfo.objects.filter(bread__gt=F('bcomment') * 2)
```

Q 对象（表示逻辑关系）

多个过滤器逐个调用表示逻辑与关系，同 sql 语句中 **where** 部分的 **and** 关键字。

例：查询阅读量大于 20，并且编号小于 3 的图书。

```
BookInfo.objects.filter(bread__gt=20,id__lt=3)
```

或

```
BookInfo.objects.filter(bread__gt=20).filter(id__lt=3)
```

如果需要实现逻辑或 **or** 的查询，需要使用 **Q()** 对象结合 **|** 运算符，Q 对象被定义在 `django.db.models` 中。

语法如下：

Q(属性名__运算符=值)

例：查询阅读量大于 20 的图书，改写为 Q 对象如下。

```
from django.db.models import Q
```

```
BookInfo.objects.filter(Q(bread__gt=20))
```

Q 对象可以使用 **&**、**|** 连接，**&** 表示逻辑与，**|** 表示逻辑或。

例：查询阅读量大于 20，或编号小于 3 的图书，只能使用 Q 对象实现

```
BookInfo.objects.filter(Q(bread__gt=20) | Q(pk__lt=3))
```

Q 对象前可以使用 **~** 操作符，表示非 **not**。

例：查询编号不等于 3 的图书。

```
BookInfo.objects.filter(~Q(pk=3))
```

聚合函数

使用 `aggregate()` 过滤器调用聚合函数。聚合函数包括: **Avg** 平均, **Count** 数量, **Max** 最大, **Min** 最小, **Sum** 求和, 被定义在 `django.db.models` 中。

例: 查询图书的总阅读量。

```
from django.db.models import Sum
```

```
BookInfo.objects.aggregate(Sum('bread'))
```

注意 `aggregate` 的返回值是一个字典类型, 格式如下:

```
{ '属性名__聚合类小写': 值 }
```

```
如: { 'bread__sum': 3 }
```

使用 `count` 时一般不使用 `aggregate()` 过滤器。

例: 查询图书总数。

```
BookInfo.objects.count()
```

注意 `count` 函数的返回值是一个数字。

2.3 排序

使用 `order_by` 对结果进行排序

```
BookInfo.objects.all().order_by('bread') # 升序
```

```
BookInfo.objects.all().order_by('-bread') # 降序
```

2.4 关联查询

由一到多的访问语法:

一对应的模型类对象. 多对应的模型类名小写 `_set` 例:

```
b = BookInfo.objects.get(id=1)
```

```
b.heroinfo_set.all()
```

由多到一的访问语法:

多对应的模型类对象. 多对应的模型类中的关系类属性名 例:

```
h = HeroInfo.objects.get(id=1)
```

```
h.hbook
```

访问一对应的模型类关联对象的 `id` 语法:

多对应的模型类对象. 关联类属性 `_id`

例:

```
h = HeroInfo.objects.get(id=1)
```

```
h.book_id
```

关联过滤查询

由多模型类条件查询一模型类数据:

语法如下:

关联模型类名小写 `__` 属性名 `__` 条件运算符 `=` 值

注意: 如果没有 `"__运算符"` 部分, 表示等于。

例:

查询图书, 要求图书英雄为 "孙悟空"

```
BookInfo.objects.filter(heroinfo__hname='孙悟空')
```

查询图书, 要求图书中英雄的描述包含 "八"

```
BookInfo.objects.filter(heroinfo__hcomment__contains='八')
```

由一模型类条件查询多模型类数据：

语法如下：

一模型类关联属性名__一模型类属性名__条件运算符=值

注意：如果没有"__运算符"部分，表示等于。

例：

查询书名为“天龙八部”的所有英雄。

```
HeroInfo.objects.filter(hbook__btitle='天龙八部')
```

查询图书阅读量大于 30 的所有英雄

```
HeroInfo.objects.filter(hbook__bread__gt=30)
```

3 修改

修改更新有两种方法

1) save

修改模型类对象的属性，然后执行 **save()** 方法

```
hero = HeroInfo.objects.get(hname='猪八戒')
```

```
hero.hname = '猪悟能'
```

```
hero.save()
```

2) update

使用模型类 **objects.filter().update()**，会返回受影响的行数

```
HeroInfo.objects.filter(hname='沙悟净').update(hname='沙僧')
```

4 删除

删除有两种方法

1) 模型类对象 delete

```
hero = HeroInfo.objects.get(id=13)
```

```
hero.delete()
```

2) 模型类.objects.filter().delete()

```
HeroInfo.objects.filter(id=14).delete()
```

查询集 QuerySet

1 概念

Django 的 ORM 中存在查询集的概念。

查询集，也称查询结果集、QuerySet，表示从数据库中获取的对象集合。

当调用如下过滤器方法时，Django 会返回查询集（而不是简单的列表）：

1. **all()**：返回所有数据。
2. **filter()**：返回满足条件的数据。
3. **exclude()**：返回满足条件之外的数据。
4. **order_by()**：对结果进行排序。

对查询集可以再次调用过滤器进行过滤，如

```
BookInfo.objects.filter(bread__gt=30).order_by('bpub_date')
```

判断某一个查询集中是否有数据：

exists()：判断查询集中是否有数据，如果有则返回 True，没有则返回 False。

2 两大特性

1) 惰性执行

创建查询集不会访问数据库，直到调用数据时，才会访问数据库，调用数据的情况包括迭代、序列化、与 `if` 合用

例如，当执行如下语句时，并未进行数据库查询，只是创建了一个查询集 `qs`

```
qs = BookInfo.objects.all()
```

继续执行遍历迭代操作后，才真正的进行了数据库的查询

```
for book in qs:
    print(book.btitle)
```

2) 缓存

使用同一个查询集，第一次使用时会发生数据库的查询，然后 `Django` 会把结果缓存下来，再次使用这个查询集时会使用缓存的数据，减少了数据库的查询次数。

情况一：如下是两个查询集，无法重用缓存，每次查询都会与数据库进行一次交互，增加了数据库的负载。

```
from booktest.models import BookInfo
[book.id for book in BookInfo.objects.all()]
[book.id for book in BookInfo.objects.all()]
```

情况二：经过存储后，可以重用查询集，第二次使用缓存中的数据。

```
qs=BookInfo.objects.all()
[book.id for book in qs]
[book.id for book in qs]
```

3 限制查询集

可以对查询集进行取下标或切片操作，等同于 `sql` 中的 `limit` 和 `offset` 子句。

注意：不支持负数索引。

对查询集进行切片后返回一个新的查询集，不会立即执行查询。

示例：获取第 1、2 项，运行查看。

```
qs = BookInfo.objects.all()[0:2]
```

管理器 Manager

管理器是 `Django` 的模型进行数据库操作的接口，`Django` 应用的每个模型类都拥有至少一个管理器。

当没有为模型类定义管理器时，`Django` 会为每一个模型类生成一个名为 `objects` 的管理器，它是 `models.Manager` 类的对象。

自定义管理器

我们可以自定义管理器，并应用到我们的模型类上。

注意：一旦为模型类指明自定义的过滤器后，`Django` 不再生成默认管理对象 `objects`。

自定义管理器类主要用于两种情况：

1. 修改原始查询集，重写 `all()` 方法。

a) 打开 `booktest/models.py` 文件，定义类 `BookInfoManager`

#图书管理器

```
class BookInfoManager(models.Manager):
    def all(self):
```

```
#默认查询未删除的图书信息
#调用父类的成员语法为: super().方法名
return super().filter(is_delete=False)
```

b) 在模型类 `BookInfo` 中定义管理器

```
class BookInfo(models.Model):
```

```
...
    books = BookInfoManager()
```

c) 使用方法

```
BookInfo.books.all()
```

2. 在管理器类中补充定义新的方法

a) 打开 `booktest/models.py` 文件, 定义方法 `create`。

```
class BookInfoManager(models.Manager):
    #创建模型类, 接收参数为属性赋值
    def create_book(self, title, pub_date):
        #创建模型类对象 self.model 可以获得模型类
        book = self.model()
        book.btitle = title
        book.bpub_date = pub_date
        book.bread=0
        book.bcommet=0
        book.is_delete = False
        # 将数据插入进数据表
        book.save()
        return book
```

b) 为模型类 `BookInfo` 定义管理器 `books` 语法如下

```
class BookInfo(models.Model):
```

```
...
    books = BookInfoManager()
```

c) 调用语法如下:

```
book=BookInfo.books.create_book("abc",date(1980,1,1))
```

模板使用

1 配置

在工程中创建模板目录 `templates`。

在 `settings.py` 配置文件中修改 **TEMPLATES** 配置项的 **DIRS** 值:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')], # 此处修改
        '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',
    ],
},
},
]

```

2 定义模板

在 `templates` 目录中新建一个模板文件，如 `index.html`

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>{{ city }}</h1>
</body>
</html>

```

3 模板渲染

调用模板分为三步骤：

1. 找到模板
2. 定义上下文
3. 渲染模板

例如，定义一个视图

```

from django.http import HttpResponse
from django.template import loader, RequestContext

def index(request):
    # 1. 获取模板
    template=loader.get_template('booktest/index.html')
    # 2. 定义上下文
    context=RequestContext(request,{'city': '北京'})
    # 3. 渲染模板
    return HttpResponse(template.render(context))

```

Django 提供了一个函数 **render** 可以简写上述代码。

`render(request 对象, 模板文件路径, 模板数据字典)`

```

from django.shortcuts import render

```

```

def index(request):
    context={'city': '北京'}
    return render(request,'index.html',context)

```

4 模板语法

4.1 模板变量

变量名必须由字母、数字、下划线（不能以下划线开头）和点组成。

语法如下：

`{{ 变量 }}`

模板变量可以使 python 的内建类型，也可以是对象。

```
def index(request):
    context = {
        'city': '北京',
        'adict': {
            'name': '西游记',
            'author': '吴承恩'
        },
        'alist': [1, 2, 3, 4, 5]
    }
    return render(request, 'index.html', context)

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>{{ city }}</h1>
    <h1>{{ adict }}</h1>
    <h1>{{ adict.name }}</h1> 注意字典的取值方法
    <h1>{{ alist }}</h1>
    <h1>{{ alist.0 }}</h1> 注意列表的取值方法
</body>
</html>
```

4.2 模板语句

1) for 循环：

`{% for item in 列表 %}`

循环逻辑

`{{forloop.counter}}`表示当前是第几次循环，从 1 开始

`{%empty%}` 列表为空或不存在时执行此逻辑

`{% endfor %}`

2) if 条件：

`{% if ... %}`

逻辑 1

`{% elif ... %}`

逻辑 2


```
{% else %}
```

逻辑 3

```
{% endif %}
```

比较运算符如下：

==

!=

<

>

<=

>=

布尔运算符如下：

and

or

not

注意：运算符左右两侧不能紧挨变量或常量，必须有空格。

```
{% if a == 1 %} # 正确
```

```
{% if a==1 %} # 错误
```

4.3 过滤器

语法如下：

- 使用管道符号|来应用过滤器，用于进行计算、转换操作，可以使用在变量、标签中。
- 如果过滤器需要参数，则使用冒号:传递参数。

变量|过滤器:参数

列举几个如下：

safe，禁用转义，告诉模板这个变量是安全的，可以解释执行

length，长度，返回字符串包含字符的个数，或列表、元组、字典的元素个数。

default，默认值，如果变量不存在时则返回默认值。

data|default:'默认值'

date，日期，用于对日期类型的值进行字符串格式化，常用的格式化字符如下：

Y 表示年，格式为 4 位，y 表示两位的年。

m 表示月，格式为 01,02,12 等。

d 表示日，格式为 01,02 等。

j 表示日，格式为 1,2 等。

H 表示时，24 进制，h 表示 12 进制的时。

i 表示分，为 0-59。

s 表示秒，为 0-59。

value|date:"Y 年 m 月 j 日 H 时 i 分 s 秒"

4.4 注释

1) 单行注释语法如下：

```
{#...#}
```

2) 多行注释使用 comment 标签，语法如下：

```
{% comment %}
```

```
...
```

```
{% endcomment %}
```

4.5 模板继承

模板继承和类的继承含义是一样的，主要是为了提高代码重用，减轻开发人员的工作量。

父模板

如果发现在多个模板中某些内容相同，那就应该把这段内容定义到父模板中。

标签 **block**: 用于在父模板中预留区域，留给子模板填充差异性的内容，名字不能相同。为了更好的可读性，建议给 **endblock** 标签写上名字，这个名字与对应的 **block** 名字相同。父模板中也可以使用上下文中传递过来的数据。

```
{% block 名称 %}
```

预留区域，可以编写默认内容，也可以没有默认内容

```
{% endblock 名称 %}
```

子模板

标签 **extends**: 继承，写在子模板文件的第一行。

```
{% extends "父模板路径"%}
```

子模版不用填充父模版中的所有预留区域，如果子模版没有填充，则使用父模版定义的默认值。

填充父模板中指定名称的预留区域。

```
{% block 名称 %}
```

实际填充内容

```
{{ block.super }}用于获取父模板中 block 的内容
```

```
{% endblock 名称 %}
```

表单使用

Django 提供对表单处理的支持，可以简化并自动化大部分的表单处理工作。

1 定义表单类

假如我们想在网页中创建一个表单，用来获取用户想保存的图书信息，可能类似的 **html** 表单如下：

```
<form action="" method="post">
    <input type="text" name="title">
    <input type="date" name="pub_date">
    <input type="submit">
</form>
```

我们可以据此来创建一个 **Form** 类来描述这个表单。

新建一个 **forms.py** 文件，编写 **Form** 类。

```
from django import forms
```

```
class BookForm(forms.Form):
    title = forms.CharField(label="书名", required=True, max_length=50)
    pub_date = forms.DateField(label='出版日期', required=True)
```

2 视图中使用表单类

```
from django.shortcuts import render
from django.views.generic import View
from django.http import HttpResponse
```

```

from .forms import BookForm

class BookView(View):
    def get(self, request):
        form = BookForm()
        return render(request, 'book.html', {'form': form})

    def post(self, request):
        form = BookForm(request.POST)
        if form.is_valid(): # 验证表单数据
            print(form.cleaned_data) # 获取验证后的表单数据
            return HttpResponse("OK")
        else:
            return render(request, 'book.html', {'form': form})

```

- `form.is_valid()` 验证表单数据的合法性
- `form.cleaned_data` 验证通过的表单数据

3 模板中使用表单类

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>书籍</title>
</head>
<body>
    <form action="" method="post">
        {% csrf_token %}
        {{ form }}
        <input type="submit">
    </form>
</body>
</html>

```

- `csrf_token` 用于添加 CSRF 防护的字段
- `form` 快速渲染表单字段的方法

4 模型类表单

如果表单中的数据与模型类对应，可以通过继承 **`forms.ModelForm`** 更快速的创建表单。

```

class BookForm(forms.ModelForm):
    class Meta:
        model = BookInfo
        fields = ('btitle', 'bpub_date')

```

- `model` 指明从属于哪个模型类
- `fields` 指明向表单中添加模型类的哪个字段

使用 Admin 站点

使用 Django 的管理模块，需要按照如下步骤操作：

- 管理界面本地化
- 创建管理员
- 注册模型类
- 自定义管理页面

1 管理界面本地化

在 `settings.py` 中设置语言和时区

```
LANGUAGE_CODE = 'zh-hans' # 使用中国语言
```

```
TIME_ZONE = 'Asia/Shanghai' # 使用中国上海时间
```

2 创建超级管理员

创建管理员的命令如下，按提示输入用户名、邮箱、密码。

```
python manage.py createsuperuser
```

打开浏览器，在地址栏中输入如下地址后回车。

```
http://127.0.0.1:8000/admin/
```

输入前面创建的用户名、密码完成登录。

登录成功后界面如下，但是并没有我们自己应用模型的入口，接下来进行第三步操作。

3 注册模型类

登录后台管理后，默认没有我们创建的应用中定义的模型类，需要在自己应用中的 `admin.py` 文件中注册，才可以在后台管理中看到，并进行增删改查操作。

打开 `booktest/admin.py` 文件，编写如下代码：

```
from django.contrib import admin
from booktest.models import BookInfo, HeroInfo
```

```
admin.site.register(BookInfo)
```

```
admin.site.register(HeroInfo)
```

到浏览器中刷新页面，可以看到模型类 `BookInfo` 和 `HeroInfo` 的管理了。

点击类名称 "`BookInfo`"（图书）可以进入列表页，默认只有一列。

在列表页中点击 "增加" 可以进入增加页，Django 会根据模型类的不同，生成不同的表单控件，按提示填写表单内容后点击 "保存"，完成数据创建，创建成功后返回列表页。

在列表页中点击某行的第一列可以进入修改页。

按照提示进行内容的修改，修改成功后进入列表页。在修改页点击 "删除" 可以删除一项。

删除：在列表页勾选想要删除的复选框，可以删除多项。

点击执行后进入确认页面，删除后回来列表页面。

4 定义与使用 Admin 管理类

Django 提供的 Admin 站点的展示效果可以通过自定义 `ModelAdmin` 类来进行控制。

定义管理类需要继承自 `admin.ModelAdmin` 类，如下

```
from django.contrib import admin
```

```
class BookInfoAdmin(admin.ModelAdmin):
    pass
```

使用管理类有两种方式:

注册参数

```
admin.site.register(BookInfo, BookInfoAdmin)
```

装饰器

```
@admin.register(BookInfo)
```

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
pass
```

调整列表页展示

1 页大小

每页中显示多少条数据, 默认为每页显示 100 条数据, 属性如下:

```
list_per_page=100
```

打开 booktest/admin.py 文件, 修改 AreaAdmin 类如下:

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
    list_per_page = 2
```

2 "操作选项"的位置

顶部显示的属性, 设置为 True 在顶部显示, 设置为 False 不在顶部显示, 默认为 True。

```
actions_on_top=True
```

底部显示的属性, 设置为 True 在底部显示, 设置为 False 不在底部显示, 默认为 False。

```
actions_on_bottom=False
```

打开 booktest/admin.py 文件, 修改 BookInfoAdmin 类如下:

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
    ...
```

```
    actions_on_top = True
```

```
    actions_on_bottom = True
```

3 列表中的列

属性如下:

```
list_display=[模型字段 1,模型字段 2,...]
```

打开 booktest/admin.py 文件, 修改 BookInfoAdmin 类如下:

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
    ...
```

```
    list_display = ['id', 'btitle']
```

点击列头可以进行升序或降序排列。

4 将方法作为列

列可以是模型字段, 还可以是模型方法, 要求方法有返回值。

通过设置 **short_description** 属性, 可以设置在 admin 站点中显示的列名。

1) 打开 booktest/models.py 文件, 修改 BookInfo 类如下:

```
class BookInfo(models.Model):
```

```
    ...
```

```
    def pub_date(self):
```

```
        return self.bpub_date.strftime('%Y年%m月%d日')
```

```
pub_date.short_description = '发布日期' # 设置方法字段在 admin 中显示的标题
```

2) 打开 booktest/admin.py 文件, 修改 BookInfoAdmin 类如下:

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
...
```

```
list_display = ['id', 'atitle', 'pub_date']
```

方法列是不能排序的, 如果需要排序需要为方法指定排序依据。

admin_order_field=模型类字段

打开 booktest/models.py 文件, 修改 BookInfo 类如下:

```
class BookInfo(models.Model):
```

```
...
```

```
def pub_date(self):
```

```
    return self.bpub_date.strftime('%Y年%m月%d日')
```

```
pub_date.short_description = '发布日期'
```

```
pub_date.admin_order_field = 'bpub_date'
```

5 关联对象

无法直接访问关联对象的属性或方法, 可以在模型类中封装方法, 访问关联对象的成员。

1) 打开 booktest/models.py 文件, 修改 HeroInfo 类如下:

```
class HeroInfo(models.Model):
```

```
...
```

```
def read(self):
```

```
    return self.hbook.bread
```

```
read.short_description = '图书阅读量'
```

2) 打开 booktest/admin.py 文件, 修改 HeroInfoAdmin 类如下:

```
class HeroInfoAdmin(admin.ModelAdmin):
```

```
...
```

```
list_display = ['id', 'hname', 'hbook', 'read']
```

6 右侧栏过滤器

属性如下, 只能接收字段, 会将对应字段的值列出来, 用于快速过滤。一般用于有重复值的字段。

```
list_filter=[]
```

打开 booktest/admin.py 文件, 修改 HeroInfoAdmin 类如下:

```
class HeroInfoAdmin(admin.ModelAdmin):
```

```
...
```

```
list_filter = ['hbook', 'hgender']
```

7 搜索框

属性如下, 用于对指定字段的值进行搜索, 支持模糊查询。列表类型, 表示在这些字段上进行搜索。

```
search_fields=[]
```

打开 booktest/admin.py 文件, 修改 HeroInfoAdmin 类如下:

```
class HeroInfoAdmin(admin.ModelAdmin):
```

```
...
search_fields = ['hname']
```

调整编辑页展示

1. 显示字段

属性如下：

```
fields=[]
```

打开 booktest/admin.py 文件，修改 BookInfoAdmin 类如下：（增加 'btitle', 'bpub_date' 字段）

```
class BookInfoAdmin(admin.ModelAdmin):
    ...
    fields = ['btitle', 'bpub_date']
```

2. 分组显示

属性如下：

```
fieldset=(
    ('组 1 标题',{'fields':('字段 1','字段 2')}),
    ('组 2 标题',{'fields':('字段 3','字段 4')}),
)
```

1) 打开 booktest/admin.py 文件，修改 BookInfoAdmin 类如下：

```
class BookInfoAdmin(admin.ModelAdmin):
    ...
    # fields = ['btitle', 'bpub_date']
    fieldsets = (
        ('基本', {'fields': ['btitle', 'bpub_date']}),
        ('高级', {
            'fields': ['bread', 'bcomment'],
            'classes': ('collapse',) # 是否折叠显示
        })
    )
```

说明：fields 与 fieldsets 两者选一使用。

3. 关联对象

在一对多的关系中，可以在一端的编辑页面中编辑多端的对象，嵌入多端对象的方式包括表格、块两种。

- 类型 InlineModelAdmin：表示在模型的编辑页面嵌入关联模型的编辑。
- 子类 TabularInline：以表格的形式嵌入。
- 子类 StackedInline：以块的形式嵌入。

1) 打开 booktest/admin.py 文件，创建 HeroInfoStackInline 类。

```
class HeroInfoStackInline(admin.StackedInline):
    model = HeroInfo # 要编辑的对象
    extra = 1 # 附加编辑的数量
```

2) 打开 booktest/admin.py 文件，修改 BookInfoAdmin 类如下：

```
class BookInfoAdmin(admin.ModelAdmin):
```

```
...
    inlines = [HeroInfoStackInline]
```

3) 刷新浏览器效果如下图:

可以用表格的形式嵌入。

1) 打开 `booktest/admin.py` 文件, 创建 `HeroInfoTabularInline` 类。

```
class HeroInfoTabularInline(admin.TabularInline):
    model = HeroInfo
    extra = 1
```

2) 打开 `booktest/admin.py` 文件, 修改 `BookInfoAdmin` 类如下:

```
class BookInfoAdmin(admin.ModelAdmin):
    ...
    inlines = [HeroInfoTabularInline]
```

3) 刷新浏览器效果如下图:

