Django

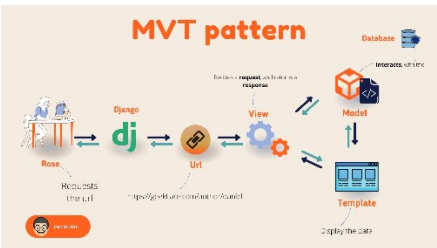
本身基于 MVC 模型（架构），即 Model（模型）+ View（视图）+ Controller（控制器）设计模式，MVC 模式使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。但在 Django 中更常被称为 MTV（Model-Template-View）。

- **模型（Model）：** 负责应用程序的数据和业务逻辑。通过将数据和逻辑从用户界面分离出来，使得模型可以独立于用户界面进行测试和修改。
- **视图（View）：** 负责显示用户界面，但通常没有直接访问应用程序的数据。这使得可以更容易地更改应用程序的外观而不影响数据处理。
- **控制器（Controller）：** 处理用户输入、更新模型和调整视图。通过将用户输入和应用程序逻辑分离，可以更容易地更改用户界面的交互方式而不影响数据和业务逻辑。

Django 提供了全栈开发所需的工具，包括数据库 ORM、模板引擎、路由系统、用户认证等，大幅减少重复代码。

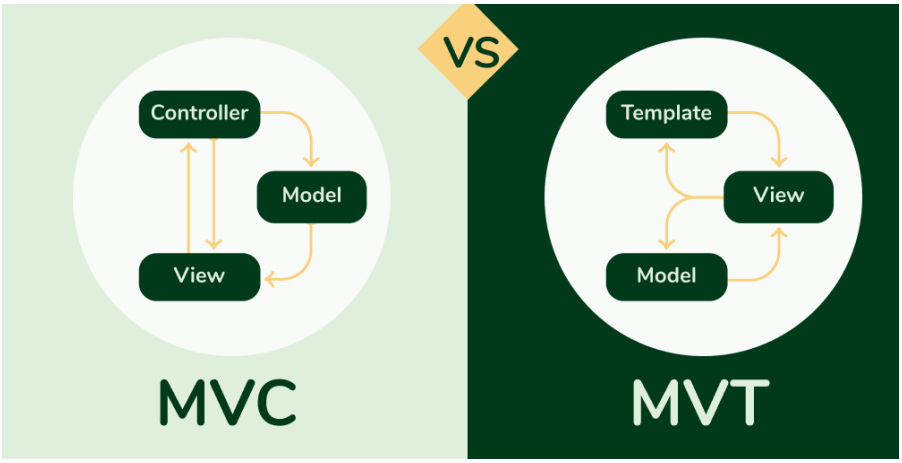
Django 的哲学:

- **DRY（Don't Repeat Yourself）:** 避免重复代码，提倡复用（如模板继承、模型继承）。
- **约定优于配置:** 默认提供合理配置（如自动生成 Admin 界面），减少决策成本。
- **快速开发：Django 提供了大量内置功能，如认证、管理后台、表单处理等，让开发者专注于业务逻辑，而非底层实现。**
- **自动化管理后台：只需简单的模型定义，即可生成强大的后台管理界面，支持增删改查。**
- **ORM 数据库映射：Django 内置 ORM (Object-Relational Mapping)，可以让开发者使用 Python 类与数据库交互，无需编写 SQL。**
- **强大的 URL 路由：使用正则表达式灵活定义 URL，轻松实现页面路由。**
- **模板引擎：内置强大的模板系统，支持逻辑判断、循环处理，方便渲染 HTML 页面。**
- **国际化支持：Django 支持多语言国际化，非常适合全球化应用。**
- **高安全性：内置多种安全保护措施，如防止 SQL 注入、XSS 攻击、CSRF 攻击等。**
- **丰富的社区与扩展：大量开源的第三方库，如 Django REST framework、Django CMS 等，快速扩展功能。**



内置功能

功能	说明
Admin 后台	自动生成管理界面，无需手动编写 CRUD 逻辑。
ORM	用 Python 类操作数据库，无需写 SQL。
表单处理	内置表单验证，防止 CSRF 攻击。
用户认证	提供登录、注册、权限管理（django.contrib.auth）。
路由系统	URL 映射灵活，支持正则表达式。
缓存机制	支持 Memcached、Redis 等后端。



MVC (Model-View-Controller)

- **Model (模型)**：处理与数据库的交互，定义数据的结构和业务逻辑。
- **View (视图)**：负责数据展示，生成用户看到的 HTML 页面。
- **Controller (控制器)**：接收用户请求，调用 Model 处理数据，并将结果传递给 View 渲染页面。

流程：

1. 用户发送请求到 Controller。

2. Controller 处理逻辑，调用 Model 获取数据。
3. Controller 将数据传递给 View。
4. View 渲染并返回 HTML 页面给用户。

MVT (Model-Template-View) —— Django 的实现方式

Django 中采用了 **MVT** 设计模式，类似于 MVC，但有一些区别：

- **Model (模型)**：与数据库交互，处理数据的创建、读取、更新、删除。
- **Template (模板)**：负责页面渲染，生成最终的 HTML 内容。
- **View (视图)**：Django 的 View 更偏向于控制器的角色，接收请求并决定使用哪个模板和数据。

流程：

用户访问 URL，请求被 Django 的 urls.py 映射到相应的 View。

url.py 下

```
urlpatterns = [
    url(r'^search-form/$', search.search_form),  # 映射到 search.py 中 search_form 视图
    url(r'^search/$', search.search),          # 映射到 search.py 中 search 视图
url(r'^search-post/$', search2.search_post),#映射到 search2.py 中 search_post 视图
]
```

- 当访问 **/search-form/** 时，Django 调用 **search.search_form** 视图函数
- 当提交表单到 **/search/** 时，调用 **search.search** 视图函数

View 处理业务逻辑，调用 Model 获取数据。

(1) 渲染搜索表单（但是这里无 Model 交互）

```
# search.py
def search_form(request):
    return render(request, 'search_form.html')  # 直接渲染模板
```

- **流程：**
  View 只负责返回模板，未涉及 Model 数据操作

---

(2) 处理搜索请求（带参数校验）

```
def search(request):
    request.encoding = 'utf-8'
    if 'q' in request.GET and request.GET['q']:  # 业务逻辑判断
        message = '你搜索的内容为: ' + request.GET['q']
    else:
        message = '你提交了空表单'
    return HttpResponse(message)  # 直接返回响应（未用模板）
```

（3）# 接收 POST 请求数据

```
def search_post(request):
    ctx ={}
    if request.POST:
        ctx['rlt'] = request.POST['q']
    return render(request, "post.html", ctx)
```

View 将数据传递给 Template。

```
return render(request, 'search_form.html')
```

在这个例子中，search_form 只是用来展示 HTML 表单页面，没有动态数据传入。

虽然这里没有传递变量，但你可以通过 render 的第三个参数传数据给模板：

```
return render(request, 'search_form.html', {'key': value})
```

第二个没有传输数据因为根本没有 template

第三个例子

```
return render(request, "post.html", ctx)
```

1. Template 渲染 HTML，最终返回给用户。

- **模板文件**: **search_form.html**

<form action="/search/" method="get">

<input type="text" name="q">

<input type="submit" value="搜索">

</form>

Tip：HTML <form> action 属性

<form action="/search/" method="get">

<input type="text" name="q">

<input type="submit" value="搜索">

</form>

(1) <form> 标签

- **action="/search/"**
    - 表单提交的目标 URL 地址
    - **/search/** 表示提交到当前网站的搜索端点（例如 **https://example.com/search/**）
    - 尾部斜杠 **/** 表示这是一个目录路径（符合 RESTful 风格）
- **method="get"**
    - 使用 HTTP GET 方法提交数据
    - 表单数据会附加在 URL 后（如 **/search/?q=关键词**）
    - 适合不修改服务器状态的查询操作（如搜索）

2. 实际工作流程

1. **用户交互**
    - 在文本框中输入内容（如"无人机"）
    - 点击"搜索"按钮
2. **浏览器行为**

- 自动组装 URL：**/search/?q=无人机**
- 发起 GET 请求（HTTP 头部示例）：

GET /search/?q=无人机 HTTP/1.1

Host: example.com

3. **服务器处理**
   - Django 路由将请求交给对应的 View 处理：

```
# urls.py
path('search/', views.search)
# views.py
def search(request):
    query = request.GET.get('q')  # 获取参数"q"的值
    return HttpResponse(f"搜索内容：{query}")
```

渲染过程:

1. **search_form** 视图调用 **render(request, 'search_form.html')**
2. Django 模板引擎找到 **templates/search_form.html**
3. 生成纯静态 HTML 返回给用户

第二个方式没有模板渲染，直接返回 httpresponse 响应。

第三个例子

```
<form action="/search-post/" method="post">
    {% csrf_token %}
    <input type="text" name="q">
    <input type="submit" value="搜索">
</form>
```

```
    <p>{{ rlt }}</p>
```

项目	GET 示例	POST 示例
URL 个数	2 个 (/search-form/ 和 /search/)	1 个 (/search-post/)
表单方法	method="get"	method="post"
数据传输方式	通过 URL 参数 (?q=...)	通过请求体
安全性	较低，数据暴露在 URL 中	更安全，数据在请求体中
CSRF Token	不需要	必须要
View 函数数量	2 个	1 个
显示与处理	分开处理	合并处理

(1) 您的两个视图函数

```
# 显示视图（表面看只负责显示）
def search_form(request):
    return render(request, 'search_form.html')  # 但本质上仍是 View 层

# 处理视图（表面看只负责处理）
def search(request):
    message = process_request(request.GET)  # 处理逻辑
    return HttpResponse(message)  # 响应生成
```

本质：
这两个函数都属于 **View 层**，只是内部逻辑侧重不同。Django 的 View 是统一的请求处理器，不强制分离显示与处理。

(2) 为什么不是两个 MVT?

- **Model**：两个视图可能共享同一个 Model（如都查询 **Article** 表）
- **Template**：**search_form** 使用模板，**search** 直接返回响应（动态选择）
- **View**：都是接收 **request** 并返回响应的函数

创建第一个项目

使用 django-admin 来创建 HelloWorld 项目：

```
django-admin startproject HelloWorld
```

$ cd HelloWorld/

$ tree

```
HelloWorld/             # 项目根目录
├── manage.py           # 项目管理脚本
├── db.sqlite3          # SQLite 数据库文件
├── __pycache__/        # Python 字节码缓存
└── HelloWorld/         # 项目配置目录（与项目同名）
    ├── __init__.py     # 包标识文件
    ├── settings.py     # 项目设置
    ├── urls.py         # 主路由配置
    ├── asgi.py         # ASGI 配置
    └── wsgi.py         # WSGI 配置
```

目录说明：

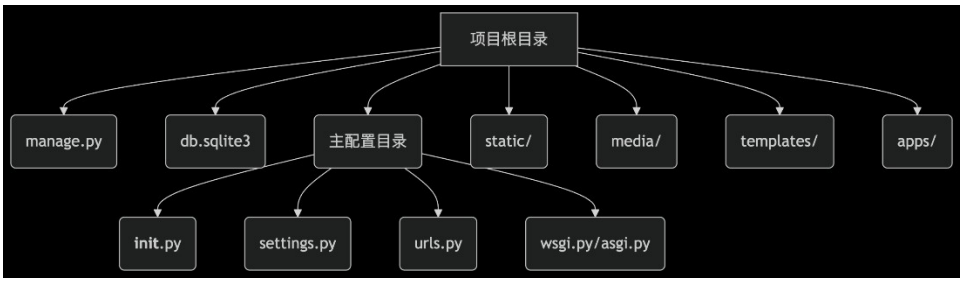
项目根目录 (HelloWorld/)：

文件/目录	作用
manage.py	Django 命令行工具入口，可让你以各种方式与该 Django 项目进行交互。用于运行开发服务器、数据库迁移等操作。
db.sqlite3	SQLite 数据库文件（默认数据库，开发环境使用）。
__pycache__/	Python 字节码缓存目录（自动生成，无需手动修改）。

- **HelloWorld:** 项目的容器。
- **HelloWorld/__init__.py:** 一个空文件，告诉 Python 该目录是一个 Python 包。
- **HelloWorld/asgi.py:** 一个 ASGI 兼容的 Web 服务器的入口，以便运行你的项目。
- **HelloWorld/settings.py:** 该 Django 项目的设置/配置。

- **核心配置文件**，包含：
  - 数据库设置 (DATABASES)

- 静态文件路径 (STATIC_URL)
- 应用注册 (INSTALLED_APPS)
- 调试模式 (DEBUG=True/False)

- **HelloWorld/urls.py:** 该 Django 项目的 URL 声明; 一份由 Django 驱动的网站"目录"。
- **主路由配置文件**，定义 URL 路径与视图的映射关系。
- 

- **HelloWorld/wsgi.py:** 一个 WSGI 兼容的 Web 服务器的入口，以便运行你的项目。

关键文件详解

1. settings.py（核心配置）

实例

```
# HelloWorld/settings.py 示例片段

# 安全警告：生产环境必须关闭 DEBUG！
DEBUG = True

# 允许访问的域名（DEBUG=False 时需配置）
ALLOWED_HOSTS = []

# 注册的 Django 应用
INSTALLED_APPS = [
    'django.contrib.admin', # 管理员后台
    'django.contrib.auth',  # 认证系统
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',  # 静态文件处理
]

# 数据库配置（默认 SQLite）
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',  # 数据库文件路径
    }
}

# 静态文件 URL（CSS/JS/图片）
STATIC_URL = 'static/'
```

2. urls.py（路由配置）

实例

```
# HelloWorld/urls.py 示例

from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),  # 后台管理路由
    # 可在此添加自定义路由，如：
    # path('blog/', include('blog.urls')),
]
```

3. manage.py（项目管理脚本）

实例

```
#!/usr/bin/env python
import os
import sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "HelloWorld.settings")
    from django.core.management import execute_from_command_line
    execute_from_command_line(sys.argv)
```

启动服务器

接下来我们进入 HelloWorld 目录输入以下命令，启动服务器：

python3 manage.py runserver 0.0.0.0:8000

0.0.0.0 让其它电脑可连接到开发服务器，8000 为端口号。如果不说明，那么端口号默认为 8000。

在浏览器输入你服务器的 ip（这里我们输入本机 IP 地址： **127.0.0.1:8000**） 及端口号。

常用 django-admin 命令详解

django-admin 是 Django 框架提供的一个命令行工具，它是管理 Django 项目的核心工具。

无论是创建新项目、运行开发服务器，还是执行数据库迁移，django-admin 都是不可或缺的工具。

要查看 django-admin 提供的所有命令，可以运行：

```
django-admin help
```

1. 创建新项目

django-admin startproject 项目名称

这个命令会在当前目录下创建一个新的 Django 项目，包含基本的项目结构：

- manage.py：项目管理脚本
- 项目名称/：项目主目录
    - __init__.py
    - settings.py：项目设置文件
    - urls.py：URL 路由配置
    - wsgi.py：WSGI 应用入口

2. 创建新应用

虽然通常使用 manage.py 来创建应用，但也可以通过 django-admin：

django-admin startapp 应用名称

这会创建一个新的 Django 应用，包含：

- migrations/：数据库迁移文件目录
- __init__.py
- admin.py：管理后台配置

- apps.py：应用配置
- models.py：数据模型定义
- tests.py：测试代码
- views.py：视图函数

models.py

定义数据模型，与数据库表对应：

实例

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField()

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

views.py

处理业务逻辑，返回响应：

实例

```
from django.shortcuts import render
from .models import Product

def product_list(request):
    products = Product.objects.all()
    return render(request, 'myapp/product_list.html', {'products': products})
```

admin.py

配置 Django 管理后台：

实例

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

@admin.register(Product)
```

```
class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price')
```

3. 检查项目配置

django-admin check

这个命令会检查你的 Django 项目是否有配置错误，包括：

- 模型定义是否正确
- URL 配置是否有效
- 模板设置是否正确
- 静态文件配置等

4. 数据库迁移

Django 使用迁移系统来管理数据库模式变更：

django-admin makemigrations # 创建迁移文件

django-admin migrate   # 应用迁移到数据库

5. 创建超级用户

django-admin createsuperuser

这个命令会引导你创建一个可以访问 Django 管理后台的超级用户。

6.启动开发服务器

django-admin runserver

基本语法:

python manage.py runserver [IP:端口]

扩展目录（非自动生成，但常用）

HelloWorld/

├── apps/   # 推荐：存放所有自定义应用

│   └── blog/   # 示例应用

├── static/   # 静态文件（CSS/JS/图片）

├── media/   # 用户上传文件

├── templates/   # 全局模板目录

└── requirements.txt # 项目依赖列表

1. apps/ 目录（推荐结构）

将应用集中管理，避免散落在项目根目录。

需在 settings.py 中配置 Python 路径：

import sys

sys.path.insert(0, os.path.join(BASE_DIR, 'apps'))

2. 静态文件与媒体文件

- **static/**：存放 CSS、JavaScript、图片等，通过 STATIC_URL 访问。
- **media/**：用户上传的文件（如头像），通过 MEDIA_URL 访问。需配置服务器在开发时提供访问：
- # urls.py（仅开发环境）
- from django.conf import settings
- from django.conf.urls.static import static
-

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

一个项目可以由多个应用

```
myproject/
|
├── manage.py
├── requirements.txt
├── static/
|   ├── css/
|   ├── js/
|   └── images/
├── media/
├── templates/
|   └── base.html
└── myproject/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── wsgi.py
    └── asgi.py
└── myapp1/
    ├── migrations/
    ├── templates/
    |   └── myapp1/
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── tests.py
```

├── urls.py

└── views.py

└── myapp2/

└── ... (类似结构)

实践建议

1. **项目与应用分离**：保持每个应用的独立性，便于复用
2. **环境配置**：使用不同的 settings 文件区分开发和生产环境
3. **静态文件管理**：开发时使用 STATICFILES_DIRS，生产时使用 collectstatic
4. **URL 设计**：在应用级别定义 URL，然后在项目级别包含
5. **模板组织**：为每个应用创建子目录存放模板

生产环境 vs 开发环境差异

文件/配置	开发环境	生产环境
DEBUG	True（显示错误详情）	False（隐藏错误，记录到日志）
数据库	SQLite（默认）	PostgreSQL/MySQL（性能优化）
静态文件	runserver 自动服务	使用 collectstatic 收集到 CDN
ALLOWED_HOSTS	空列表或 ['localhost']	必须配置域名（如 ['example.com']）

path() 函数

Django path() 可以接收四个参数，分别是两个必选参数：route、view 和两个可选参数：kwargs、name。

语法格式：

```
path(route, view, kwargs=None, name=None)
```

route： 字符串，定义 URL 的路径部分。可以包含变量，例如 <int:my_variable>，以从 URL 中捕获参数并将其传递给视图函数。

view： 视图函数，处理与给定路由匹配的请求。可以是一个函数或一个基于类的视图。

kwargs（可选）： 一个字典，包含传递给视图函数的额外关键字参数。

name（可选）： 为 URL 路由指定一个唯一的名称，以便在代码的其他地方引用它。这对于在模板中生成 URL 或在代码中进行重定向等操作非常有用。

这样我们就完成了使用模板来输出数据，从而实现数据与视图分离。

接下来我们将具体介绍模板中常用的语法规则。

Django 模板

Django 的模板系统（Template System）是用于将业务逻辑（Python）与展示层（HTML）分离的核心组件，它允许开发者通过简单的标签和变量动态生成 HTML 页面。

在上一章节中我们使用 **django.http.HttpResponse()** 来输出 **"Hello World！"**，该方式将数据与视图混合在一起，不符合 Django 的 MVT 思想。

本章节我们将为大家详细介绍 Django 模板的应用，模板是一个文本，用于分离文档的表现形式和内容。

功能	语法/示例	适用场景
变量渲染	{{ variable }}	动态显示数据
逻辑控制	{% if %}, {% for %}	条件/循环渲染
模板继承	{% extends %}, {% block %}	避免重复 HTML 结构
静态文件	{% static 'path' %}	加载 CSS/JS/图片
自定义过滤器	@register.filter	扩展模板功能

Django 模板标签

变量

模板语法：

```
view:  {"HTML 变量名" : "views 变量名"}

HTML:  { {变量名} }
```

列表

templates 中的 runoob.html 中，可以用 . 索引下标取出对应的元素。

HelloWorld/HelloWorld/views.py 文件代码：

```
from django.shortcuts import render
def runoob(request):
    views_list = ["菜鸟教程 1","菜鸟教程 2","菜鸟教程 3"]
    return render(request, "runoob.html", {"views_list": views_list})
```

templates 中的 runoob.html 中，可以用 `.键` 取出对应的值。

过滤器

模板语法：

```
{{ 变量名 | 过滤器：可选参数 }}
```

模板过滤器可以在变量被显示前修改它，过滤器使用管道字符，如下所示：

safe

将字符串标记为安全，不需要转义。

要保证 views.py 传过来的数据绝对安全，才能用 safe。

和后端 views.py 的 mark_safe 效果相同。

Django 会自动对 views.py 传到 HTML 文件中的标签语法进行转义，令其语义失效。加 safe 过滤器是告诉 Django 该数据是安全的，不必对其进行转义，可以让该数据语义生效。

if/else 标签

基本语法格式如下：

```
{% if condition %}

     ... display{% endif %}
```

或者：

```
{% if condition1 %}

   ... display 1{% elif condition2 %}

   ... display 2{% else %}

   ... display 3{% endif %}
```

根据条件判断是否输出。if/else 支持嵌套。

{% if %} 标签接受 and ， or 或者 not 关键字来对多个变量做判断 ，或者对变量取反 （ not)，例如：

```
{% if athlete_list and coach_list %}

     athletes 和 coaches 变量都是可用的。{% endif %}
```

遍历字典: 可以直接用字典 **.items** 方法，用变量的解包分别获取键和值。

for 标签

{% for %} 允许我们在一个序列上迭代。

与 Python 的 for 语句的情形类似，循环语法是 for X in Y ，Y 是要迭代的序列而 X 是在每一个特定的循环中使用的变量名称。

每一次循环中，模板系统会渲染在 **{% for %}** 和 **{% endfor %}** 之间的所有内容。

例如，给定一个运动员列表 athlete_list 变量，我们可以使用下面的代码来显示这个列表：

{% empty %}

可选的 {% empty %} 从句：在循环为空的时候执行（即 in 后面的参数布尔值为 False ）。

注释标签

Django 注释使用 {# #}。

```
{# 这是一个注释 #}
```

include 标签

{% include %} 标签允许在模板中包含其它的模板的内容。

下面这个例子都包含了 nav.html 模板：

```
{% include "nav.html" %}
```

csrf_token

csrf_token 用于 form 表单中，作用是跨站请求伪造保护。

如果不用 **{% csrf_token %}** 标签，在用 form 表单时，要再次跳转页面会报 403 权限错误。

用了**{% csrf_token %}** 标签，在 form 表单提交数据时，才会成功。

自定义标签和过滤器

1、在应用目录下创建 **templatetags** 目录(与 templates 目录同级，目录名只能是 templatetags)

4、利用装饰器 @register.filter 自定义过滤器。

注意：装饰器的参数最多只能有 2 个。

4、利用装饰器 @register.filter 自定义过滤器。

注意：装饰器的参数最多只能有 2 个。

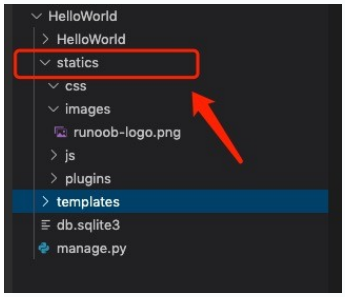
```
@register.filterdef my_filter(v1, v2):

    return v1 * v2
```

5、利用装饰器 @register.simple_tag 自定义标签。

配置静态文件

1、在项目根目录下创建 statics 目录。



2、在 settings 文件的最下方配置添加以下配置：

```
STATIC_URL = '/static/' # 别名

STATICFILES_DIRS = [

    os.path.join(BASE_DIR, "statics"), ]
```

3、在 statics 目录下创建 css 目录，js 目录，images 目录，plugins 目录， 分别放 css 文件，js 文件，图片，插件。

4、把 bootstrap 框架放入插件目录 plugins。

5、在 HTML 文件的 head 标签中引入 bootstrap。

注意：此时引用路径中的要用配置文件中的别名 static，而不是目录 statics。

模板继承

模板可以用继承的方式来实现复用，减少冗余内容。

网页的头部和尾部内容一般都是一致的，我们就可以通过模板继承来实现复用。

父模板用于放置可重复利用的内容，子模板继承父模板的内容，并放置自己的内容。。

父模板

标签 block...endblock: 父模板中的预留区域，该区域留给子模板填充差异性的内容，不同预留区域名字不能相同。

```
{% block 名称 %} 预留给子模板的区域，可以设置设置默认内容{% endblock 名称 %}
```

子模板

子模板使用标签 extends 继承父模板：

```
{% extends "父模板路径"%}
```

子模板如果没有设置父模板预留区域的内容，则使用在父模板设置的默认内容，当然也可以都不设置，就为空。

子模板设置父模板预留区域的内容：

```
{ % block 名称 % }内容 {% endblock 名称 %}
```

Django 模型使用自带的 ORM。

对象关系映射（Object Relational Mapping，简称 ORM ）用于实现面向对象编程语言里不同类型系统的数据之间的转换。

ORM 在业务逻辑层和数据库层之间充当了桥梁的作用。

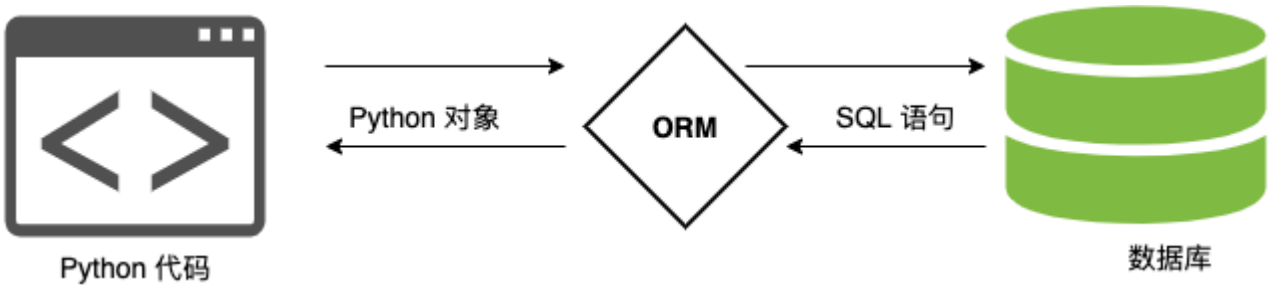
ORM 是通过使用描述对象和数据库之间的映射的元数据，将程序中的对象自动持久化到数据库中。

ORM 解析过程:

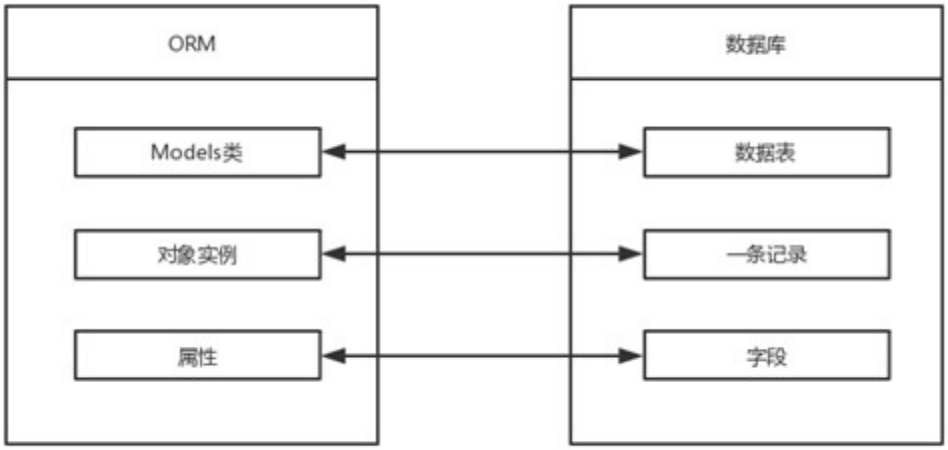
1、ORM 会将 Python 代码转成为 SQL 语句。

2、SQL 语句通过 pymysql 传送到数据库服务端。

3、在数据库中执行 SQL 语句并将结果返回。



ORM 对应关系表：



数据库配置

ORM 无法操作到数据库级别，只能操作到数据表

我们在项目的 settings.py 文件中找到 DATABASES 配置项，将其信息修改为：

HelloWorld/HelloWorld/settings.py: 文件代码：

DATABASES = { 'default': { 'ENGINE': 'django.db.backends.mysql', # 数据库引擎 'NAME': 'runoob', # 数据库名称 'HOST': '127.0.0.1', # 数据库地址，本机 ip 地址 127.0.0.1 'PORT': 3306, # 端口 'USER': 'root', # 数据库用户名 'PASSWORD': '123456', # 数据库密码 } }

上面包含数据库名称和用户的信息，它们与 MySQL 中对应数据库和用户的设置相同。Django 根据这一设置，与 MySQL 中相应的数据库和用户连接起来。

接下来，告诉 Django 使用 pymysql 模块连接 mysql 数据库：

实例

```
# 在与 settings.py 同级目录下的 __init__.py 中引入模块和进行配置
import pymysql
pymysql.install_as_MySQLdb()
```

定义模型

创建 APP

Django 规定，如果要使用模型，必须要创建一个 app。我们使用以下命令创建一个 TestModel 的 app:

```
django-admin startapp TestModel
```

```
# models.py from django.db import models class Test(models.Model): name =
models.CharField(max_length=20)
```

以上的类名代表了数据库表名，且继承了 models.Model，类里面的字段代表数据表中的字段(name)，数据类型则由 CharField（相当于 varchar）、DateField（相当于 datetime）， max_length 参数限定长度。

在命令行中运行：

```
$ python3 manage.py migrate   # 创建表结构

$ python3 manage.py makemigrations TestModel  # 让 Django 知道我们在我们的模型有一些变更

$ python3 manage.py migrate TestModel   # 创建表结构
```

看到几行 "Creating table..." 的字样，你的数据表就创建好了。

```
Creating tables ...
.....
Creating table TestModel_test  #我们自定义的表
.....
```

表名组成结构为：应用名_类名（如：TestModel_test）。

注意：尽管我们没有在 models 给表设置主键，但是 Django 会自动添加一个 id 作为主键。

添加数据

添加数据需要先创建对象，然后再执行 save 函数，相当于 SQL 中的 INSERT：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    test1 = Test(name='runoob')
    test1.save()
```

```
    return HttpResponse("<p>数据添加成功！</p>")
```

更新数据

修改数据可以使用 save() 或 update():

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 修改其中一个 id=1 的 name 字段，再 save，相当于 SQL 中的 UPDATE
    test1 = Test.objects.get(id=1)
    test1.name = 'Google'
    test1.save()

    # 另外一种方式
    #Test.objects.filter(id=1).update(name='Google')

    # 修改所有的列
    # Test.objects.all().update(name='Google')

    return HttpResponse("<p>修改成功</p>")
```

删除数据

删除数据库中的对象只需调用该对象的 delete()方法即可：

Django 表单

HTML 表单是网站交互性的经典方式。 本章将介绍如何用 Django 对用户提交的表单数据进行处理。

HTTP 请求

HTTP 协议以"请求－回复"的方式工作。客户发送请求时，可以在请求中附加数据。服务器通过解析请求，就可以获得客户传来的数据，并根据 URL 来提供特定的服务。

GET 方法

用户提交了表单，服务器解析表单并回复

我们在之前的项目中创建一个 search.py 文件，用于接收用户的请求：

```
from django.http import HttpResponse

from django.shortcuts import render

# 表单

def search_form(request):
```

```
    return render(request, 'search_form.html')


# 接收请求数据
def search(request):
    request.encoding='utf-8'
    if 'q' in request.GET and request.GET['q']:
        message = '你搜索的内容为: ' + request.GET['q']
    else:
        message = '你提交了空表单'
    return HttpResponse(message)
```

在模板目录 templates 中添加 search_form.html 表单：

/HelloWorld/templates/search_form.html 文件代码：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="搜索">
    </form>
</body>
</html>
```

urls.py 规则修改为如下形式：

/HelloWorld/HelloWorld/urls.py 文件代码：

```
from django.conf.urls import url
from . import views,testdb,search


urlpatterns = [
    url(r'^hello/$', views.runoob),
    url(r'^testdb/$', testdb.testdb),
    url(r'^search-form/$', search.search_form),
    url(r'^search/$', search.search),
]
```

POST 方法

上面我们使用了 GET 方法，视图显示和请求处理分成两个函数处理。

提交数据时更常用 POST 方法。我们下面使用该方法，并用一个 URL 和处理函数，同时显示视图和处理请求。

1. GET 方法实现（search.py）

python

```
# 两个独立的函数：一个显示表单，一个处理请求
def search_form(request):  # 显示表单
    return render(request, 'search_form.html')


def search(request):  # 处理请求
    if 'q' in request.GET and request.GET['q']:
        message = '你搜索的内容为: ' + request.GET['q']
    else:
        message = '你提交了空表单'
    return HttpResponse(message)
```

2. POST 方法实现（search2.py）

python

```
# 单个函数同时处理表单显示和请求处理
def search_post(request):
    ctx = {}
    if request.POST:  # 如果是 POST 请求（表单提交）
        ctx['rlt'] = request.POST['q']
    return render(request, "post.html", ctx)  # 总是返回模板
```

何时用 GET？何时用 POST？

适合 GET 的情况：

- **搜索功能**（用户可能想收藏或分享搜索结果 URL）
- **筛选和排序**（参数在 URL 中便于分享）
- **幂等操作**（不改变服务器状态的操作）

适合 POST 的情况：

- **登录/注册**（包含敏感信息）
- **表单提交**（创建新数据）
- **任何会改变服务器状态的操作**
- **大量数据提交**

最佳实践总结

1. **GET 完全可以用单函数处理**显示和处理逻辑
2. **原代码设计是过时的模式**，现代 Django 开发不推荐这样分离
3. **选择 GET 还是 POST 取决于具体场景**，而不是技术限制
4. **单函数模式**更简洁、更易维护

Request 对象

每个视图函数的第一个参数是一个 HttpRequest 对象，就像下面这个 runoob() 函数:

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

QueryDict 对象

在 HttpRequest 对象中, GET 和 POST 属性是 django.http.QueryDict 类的实例。

QueryDict 类似字典的自定义类，用来处理单键对应多值的情况。

QueryDict 实现所有标准的词典方法。还包括一些特有的方法：

```
def runoob(request):
    return HttpResponse("Hello world")
```

Django 视图

视图层

一个视图函数，简称视图，是一个简单的 Python 函数，它接受 Web 请求并且返回 Web 响应。

响应可以是一个 HTML 页面、一个 404 错误页面、重定向页面、XML 文档、或者一张图片...

无论视图本身包含什么逻辑，都要返回响应。代码写在哪里都可以，只要在 Python 目录下面，一般放在项目的 views.py 文件中。

每个视图函数都负责返回一个 HttpResponse 对象，对象中包含生成的响应。

视图层中有两个重要的对象：请求对象(request)与响应对象(HttpResponse)。

请求对象: HttpRequest 对象（简称 request 对象）

以下介绍几个常用的 request 属性。

1、GET

数据类型是 QueryDict，一个类似于字典的对象，包含 HTTP GET 的所有参数。

有相同的键，就把所有的值放到对应的列表里。

取值格式：**对象.方法**。

get()：返回字符串，如果该键对应有多个值，取出该键的最后一个值。

实例

```
def runoob(request):
    name = request.GET.get("name")
    return HttpResponse('姓名：{}'.format(name))
```

2、POST

数据类型是 QueryDict，一个类似于字典的对象，包含 HTTP POST 的所有参数。

常用于 form 表单，form 表单里的标签 name 属性对应参数的键，value 属性对应参数的值。

取值格式： **对象.方法**。

get()：返回字符串，如果该键对应有多个值，取出该键的最后一个值。

实例

```
def runoob(request):
    name = request.POST.get("name")
    return HttpResponse('姓名：{}'.format(name))
```

3、body

数据类型是二进制字节流，是原生请求体里的参数内容，在 HTTP 中用于 POST，因为 GET 没有请求体。

在 HTTP 中不常用，而在处理非 HTTP 形式的报文时非常有用，例如：二进制图片、XML、Json 等。

实例

```
def runoob(request):
    name = request.body
    print(name)
    return HttpResponse("菜鸟教程")
```

获取 URL 中的路径部分，数据类型是字符串。

实例

```
def runoob(request):
    name = request.path
    print(name)
    return HttpResponse("菜鸟教程")
```

4、path

获取 URL 中的路径部分，数据类型是字符串。

实例

```
def runoob(request):
    name = request.path
    print(name)
    return HttpResponse("菜鸟教程")
```

5、method

获取当前请求的方式，数据类型是字符串，且结果为大写。

实例

```
def runoob(request):
    name = request.method
    print(name)
    return HttpResponse("菜鸟教程")
```

响应对象：HttpResponse 对象

响应对象主要有三种形式：HttpResponse()、render()、redirect()。

HttpResponse(): 返回文本，参数为字符串，字符串中写文本内容。如果参数为字符串里含有 html 标签，也可以渲染。

ender(): 返回文本，第一个参数为 request，第二个参数为字符串（页面名称），第三个参数为字典（可选参数，向页面传递的参数：键为页面参数名，值为 views 参数名）。

实例

```
def runoob(request):
    name ="菜鸟教程"
    return render(request,"runoob.html",{"name":name})
```

redirect()：重定向，跳转新页面。参数为字符串，字符串中填写页面路径。一般用于 form 表单提交后，跳转到新页面。

实例

```
def runoob(request):
```

```
    return redirect("/index/")
```

render 和 redirect 是在 HttpResponse 的基础上进行了封装：

- render：底层返回的也是 HttpResponse 对象
- redirect：底层继承的是 HttpResponse 对象

Django 路由

路由简单的来说就是根据用户请求的 URL 链接来判断对应的处理程序，并返回处理结果，也就是 URL 与 Django 的视图建立映射关系。

Django 路由在 urls.py 配置，urls.py 中的每一条配置对应相应的处理方法。

Path 设置

Django 2.2.x 之后的版本

- path：用于普通路径，不需要自己手动添加正则首位限制符号，底层已经添加。
- re_path：用于正则路径，需要自己手动添加正则首位限制符号。

实例

```
from django.urls import re_path # 用 re_path 需要引入
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index), # 普通路径
    re_path(r'^articles/([0-9]{4})/$', views.articles), # 正则路径
]
```

正则路径中的分组

正则路径中的无名分组

无名分组按位置传参，一一对应。

views 中除了 request，其他形参的数量要与 urls 中的分组数量一致。

urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    re_path("^index/([0-9]{4})/$", views.index),
]
```

views.py

```
from django.shortcuts import HttpResponse


def index(request, year):
    print(year) # 一个形参代表路径中一个分组的内容，按顺序匹配
    return HttpResponse('菜鸟教程')
```

正则路径中的有名分组

语法：

(?P<组名>正则表达式)

有名分组按关键字传参，与位置顺序无关。

views 中除了 request，其他形参的数量要与 urls 中的分组数量一致， 并且 views 中的形参名称要与 urls 中的组名对应。

urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    re_path("^index/(?P[0-9]{4})/(?P[0-9]{2})/$", views.index),
]
```

views.py

```
from django.shortcuts import HttpResponse
def index(request, year, month):
    print(year,month) # 一个形参代表路径中一个分组的内容，按关键字对应匹配
    return HttpResponse('菜鸟教程')
```

路由分发(include)

存在问题：Django 项目里多个 app 目录共用一个 urls 容易造成混淆，后期维护也不方便。

解决：使用路由分发（include），让每个 app 目录都单独拥有自己的 urls。

步骤：

- 1、在每个 app 目录里都创建一个 urls.py 文件。
- 2、在项目名称目录下的 urls 文件里，统一将路径分发给各个 app 目录。

实例

```
from django.contrib import admin
from django.urls import path,include # 从 django.urls 引入 include
urlpatterns = [
    path('admin/', admin.site.urls),
    path("app01/", include("app01.urls")),
    path("app02/", include("app02.urls")),
```

反向解析

随着功能的增加，路由层的 url 发生变化，就需要去更改对应的视图层和模板层的 url，非常麻烦，不便维护。

这时我们可以利用反向解析，当路由层 url 发生改变，在视图层和模板层动态反向解析出更改后的 url，免去修改的操作。

反向解析一般用在模板中的超链接及视图中的重定向。

普通路径

在 urls.py 中给路由起别名，**name="路由别名"**。

```
path("login1/", views.login, name="login")
```

在 views.py 中，从 django.urls 中引入 reverse，利用 **reverse("路由别名")** 反向解析:

```
return redirect(reverse("login"))
```

在模板 templates 中的 HTML 文件中，利用 **{% url "路由别名" %}** 反向解析。

```
<form action="{% url 'login' %}" method="post">
```

正则路径（无名分组）

在 urls.py 中给路由起别名，**name="路由别名"**。

re_path(r"^login/([0-9]{2})/$", views.login, name="login")

在 views.py 中，从 django.urls 中引入 reverse，利用 **reverse("路由别名"，args=(符合正则匹配的参数,))** 反向解析。

return redirect(reverse("login",args=(10,)))

在模板 templates 中的 HTML 文件中利用 **{% url "路由别名" 符合正则匹配的参数 %}** 反向解析。

<form action="{% url 'login' 10 %}" method="post">

正则路径（有名分组）

在 urls.py 中给路由起别名，**name="路由别名"**。

re_path(r"^login/(?P<year>[0-9]{4})/$", views.login, name="login")

在 views.py 中，从 django.urls 中引入 reverse，利用 **reverse("路由别名"，kwargs={"分组名":符合正则匹配的参数})** 反向解析。

return redirect(reverse("login",kwargs={"year":3333}))

在模板 templates 中的 HTML 文件中，利用 **{% url "路由别名" 分组名=符合正则匹配的参数 %}** 反向解析。

<form action="{% url 'login' year=3333 %}" method="post">

命名空间

命名空间（英语：Namespace）是表示标识符的可见范围。

一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。

一个新的命名空间中可定义任何标识符，它们不会与任何重复的标识符发生冲突，因为重复的定义都处于其它命名空间中。

存在问题：路由别名 name 没有作用域，Django 在反向解析 URL 时，会在项目全局顺序搜索，当查找到第一个路由别名 name 指定 URL 时，立即返回。当在不同的 app 目录下的 urls 中定义相同的路由别名 name 时，可能会导致 URL 反向解析错误。

解决：使用命名空间。

普通路径

定义命名空间（include 里面是一个元组）格式如下：

include(("app 名称：urls"，"app 名称"))

实例：

path("app01/", include(("app01.urls","app01")))

path("app02/", include(("app02.urls","app02")))

在 app01/urls.py 中起相同的路由别名。

path("login/", views.login, name="login")

在 views.py 中使用名称空间，语法格式如下：

reverse("app 名称：路由别名")

实例：

return redirect(reverse("app01:login")

Django Admin 管理工具

Django 提供了基于 web 的管理工具。

Django 自动管理工具是 django.contrib 的一部分。你可以在项目的 settings.py 中的 INSTALLED_APPS 看到它：

/HelloWorld/HelloWorld/settings.py 文件代码：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```
django.contrib 是一套庞大的功能集，它是 Django 基本代码的组成部分。

激活管理工具

通常我们在生成项目时会在 urls.py 中自动设置好，我们只需去掉注释即可。

配置项如下所示：

/HelloWorld/HelloWorld/urls.py 文件代码：

```
# urls.py
from django.conf.urls import url
from django.contrib import admin
urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```
当这一切都配置好后，Django 管理工具就可以运行了。

你可以通过命令 **python manage.py createsuperuser** 来创建超级用户，如下所示：

```
# python manage.py createsuperuser
Username (leave blank to use 'root'): admin
Email address: admin@runoob.com
Password:
Password (again):
Superuser created successfully.
[root@solar HelloWorld]#
```

复杂模型

管理页面的功能强大，完全有能力处理更加复杂的数据模型。

django-admin.py startproject app01

接下来在 settings.py 中找到 INSTALLED_APPS 这一项，如下：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testmodel',              # 添加此项
)
```

建表

先在 TestModel/models.py 中增加一个更复杂的数据模型：

HelloWorld/TestModel/models.py: 文件代码：

```
from django.db import models

# Create your models here.
class Test(models.Model):
    name = models.CharField(max_length=20)

class Contact(models.Model):
    name   = models.CharField(max_length=200)
    age    = models.IntegerField(default=0)
    email  = models.EmailField()
    def __unicode__(self):
        return self.name

class Tag(models.Model):
    contact = models.ForeignKey(Contact, on_delete=models.CASCADE,)
    name    = models.CharField(max_length=50)
    def __unicode__(self):
        return self.name
```

这里有两个表。Tag 以 Contact 为外部键。一个 Contact 可以对应多个 Tag。

我们还可以看到许多在之前没有见过的属性类型，比如 IntegerField 用于存储整数。

用 django 创建表

```
$ python manage.py migrate   # 创建表结构
```

python manage.py makemigrations testmodel 让模型知道我们的表有变更

python manage.py migrate TestModel 创建表结构

接下来我们在 app01 项目里添加 views.py 和 models.py 文件，app01 项目目录结构：

```
app01
|-- app01
|   |-- __init__.py
|   |-- __pycache__
|   |-- asgi.py
|   |-- migrations
|   |-- models.py
|   |-- settings.py
```

```
|   |-- urls.py
|   |-- views.py
|   `-- wsgi.py
```

显示模型

在 TestModel/admin.py 注册多个模型并显示：

HelloWorld/TestModel/admin.py: 文件代码：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
admin.site.register([Test, Contact, Tag])
```

每个栏也可以定义自己的格式。

修改 TestModel/admin.py 为：

HelloWorld/TestModel/admin.py: 文件代码：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',), # CSS
            'fields': ('age',),
        }]
    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

上面的栏目分为了 Main 和 Advance 两部分。classes 说明它所在的部分的 CSS 格式。这里让 Advance 部分隐藏：

内联(Inline)显示

上面的 Contact 是 Tag 的外部键，所以有外部参考的关系。

而在默认的页面显示中，将两者分离开来，无法体现出两者的从属关系。我们可以使用内联显示，让 Tag 附加在 Contact 的编辑页面上显示。

修改 TestModel/admin.py：

HelloWorld/TestModel/admin.py: 文件代码：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class TagInline(admin.TabularInline):
    model = Tag

class ContactAdmin(admin.ModelAdmin):
    inlines = [TagInline]  # Inline
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',),
            'fields': ('age',),
        }]

    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

自定义该页面的显示

比如在列表中显示更多的栏目，只需要在 ContactAdmin 中增加 list_display 属性:

HelloWorld/TestModel/admin.py: 文件代码：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class TagInline(admin.TabularInline):
```

```
    model = Tag

class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email') # list
    inlines = [TagInline]  # Inline
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',),
            'fields': ('age',),
        }]

    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

搜索功能

search_fields 为该列表页增加搜索栏：

HelloWorld/TestModel/admin.py: 文件代码：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class TagInline(admin.TabularInline):
    model = Tag

class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email') # list
    search_fields = ('name',)
    inlines = [TagInline]  # Inline
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',),
            'fields': ('age',),
```

```
    }]

)

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

在 django2.0 后，定义外键和一对一关系的时候需要加 on_delete 选项，此参数为了避免两个表里的数据不一致问题，不然会报错：**TypeError: __init__() missing 1 required positional argument: 'on_delete'。**

owner=models.ForeignKey(UserProfile,on_delete=models.CASCADE) --在老版本这个参数（models.CASCADE）是默认值参数

说明：on_delete 有 CASCADE、PROTECT、SET_NULL、SET_DEFAULT、SET() 五个可选择的值。

- **CASCADE**：此值设置，是级联删除。
- **PROTECT**：此值设置，是会报完整性错误。
- **SET_NULL**：此值设置，会把外键设置为 null，前提是允许为 null。
- **SET_DEFAULT**：此值设置，会把设置为外键的默认值。
- **SET()**：此值设置，会调用外面的值，可以是一个函数。一般情况下使用 CASCADE 就可以了。

如果你之前还未创建表结构，可使用以下命令创建：

```
$ python manage.py makemigrations TestModel  # 让 Django 知道我们在我们的模型有一些变更
$ python manage.py migrate TestModel   # 创建表结构
```

数据库添加

规则配置：

app01/urls.py: 文件代码：

```
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('add_book/', views.add_book),
]
```

方式一：模型类实例化对象

需从 app 目录引入 models.py 文件：

from app 目录 import models

并且实例化对象后要执行 对象.save() 才能在数据库中新增成功。

app01/views.py: 文件代码：

```
from django.shortcuts import render,HttpResponse
from app01 import models
def add_book(request):
```

```
    book = models.Book(title="菜鸟教程",price=300,publish="菜鸟出版社",pub_date="2008-8-8")
    book.save()
    return HttpResponse("<p>数据添加成功！</p>")
```

方式二：通过 ORM 提供的 objects 提供的方法 create 来实现（推荐）

app01/views.py: 文件代码：

```
from django.shortcuts import render,HttpResponse
from app01 import models
def add_book(request):
    books = models.Book.objects.create(title="如来神掌",price=200,publish="功夫出版社",pub_date="2010-10-10")
    print(books, type(books)) # Book object (18)
    return HttpResponse("<p>数据添加成功！</p>")
```

查找

使用 **all()** 方法来查询所有内容。
返回的是 QuerySet 类型数据，类似于 list，里面放的是一个个模型类的对象，可用索引下标取出模型类的对象。
（for I in books）

```
    books = models.Book.objects.all()
    print(books,type(books)) # QuerySet 类型，类似于 list，访问 url 时数据显示在命令行窗口中。
```

查询符合条件的数据。

filter() 方法

返回的是 QuerySet 类型数据，类似于 list，里面放的是满足条件的模型类的对象，可用索引下标取出模型类的对象。

pk=3 的意思是主键 primary key=3，相当于 id=3。

因为 id 在 pycharm 里有特殊含义，是看内存地址的内置函数 id()，因此用 pk。

```
    books = models.Book.objects.filter(pk=5)
    print(books)
    print("////////////////////////////////////////")
    books = models.Book.objects.filter(publish='菜鸟出版社', price=300)
    print(books, type(books))  # QuerySet 类型，类似于 list。
```

查询不符合条件的数据。

exclude() 方法

返回的是 QuerySet 类型数据，类似于 list，里面放的是不满足条件的模型类的对象，可用索引下标取出模型类的对象。

```
books = models.Book.objects.exclude(pk=5)
print(books)
print("//////////////////////////////////////")
books = models.Book.objects.exclude(publish='菜鸟出版社', price=300)
print(books, type(books))  # QuerySet 类型，类似于 list。
```

get() 方法

用于查询符合条件的返回模型类的对象符合条件的对象只能为一个，如果符合筛选条件的对象超过了一个或者没有一个都会抛出错误。

```
books = models.Book.objects.get(pk=5)
books = models.Book.objects.get(pk=18)  # 报错，没有符合条件的对象
books = models.Book.objects.get(price=200)  # 报错，符合条件的对象超过一个
print(books, type(books))  # 模型类的对象
```

排序

order_by()

返回的是 QuerySet 类型数据，类似于 list，里面放的是排序后的模型类的对象，可用索引下标取出模型类的对象。

注意：

a、参数的字段名要加引号。

b、降序为在字段前面加个负号 -。

```
books = models.Book.objects.order_by("price") # 查询所有，按照价格升序排列
books = models.Book.objects.order_by("-price") # 查询所有，按照价格降序排列
```

对查询结果进行反转。

reverse() 方法

返回的是 QuerySe t 类型数据，类似于 list，里面放的是反转后的模型类的对象，可用索引下标取出模型类的对象。

```
# 按照价格升序排列：降序再反转
books = models.Book.objects.order_by("-price").reverse()
```

查询数据的数量

返回的数据是整数。

count() 方法

```
books = models.Book.objects.count() # 查询所有数据的数量
books = models.Book.objects.filter(price=200).count() # 查询符合条件数据的数量
```

first() 方法返回第一条数据返回的数据是模型类的对象也可以用索引下标 [0]。

```
books = models.Book.objects.first() # 返回所有数据的第一条数据
```

last() 方法返回最后一条数据返回的数据是模型类的对象不能用索引下标 [-1]，ORM 没有逆序索引。

```
books = models.Book.objects.last() # 返回所有数据的最后一条数据
```

是否存在数据

exists() 方法用于判断查询的结果 QuerySet 列表里是否有数据。

返回的数据类型是布尔，有为 true，没有为 false。

注意：判断的数据类型只能为 QuerySet 类型数据，不能为整型和模型类的对象。

```
books = models.Book.objects.exists()
# 报错，判断的数据类型只能为 QuerySet 类型数据，不能为整型
books = models.Book.objects.count().exists()
# 报错，判断的数据类型只能为 QuerySet 类型数据，不能为模型类对象
books = models.Book.objects.first().exists()
```

查询数据

values() 方法用于查询部分字段的数据。

返回的是 QuerySet 类型数据，类似于 list，里面不是模型类的对象，而是一个可迭代的字典序列，字典里的键是字段，值是数据。

注意：

参数的字段名要加引号

想要字段名和数据用 values

```
books = models.Book.objects.values("pk","price")
print(books[0]["price"],type(books)) # 得到的是第一条记录的 price 字段的数据
```

查询数据 2

values_list() 方法用于查询部分字段的数据。

返回的是 QuerySet 类型数据，类似于 list，里面不是模型类的对象，而是一个个元组，元组里放的是查询字段对应的数据。

注意：

- 参数的字段名要加引号
- 只想要数据用 values_list

```
books = models.Book.objects.values_list("price","publish")
print(books)
print(books[0][0],type(books)) # 得到的是第一条记录的 price 字段的数据
```

去重

distinct() 方法用于对数据进行去重。

返回的是 QuerySet 类型数据。

注意：

- 对模型类的对象去重没有意义，因为每个对象都是一个不一样的存在。
- distinct() 一般是联合 values 或者 values_list 使用。

```
def add_book(request):
    # 查询一共有多少个出版社
    books = models.Book.objects.values_list("publish").distinct() # 对模型类的对象去重没有意义，因为每个对象都是一个不一样的存在。
    books = models.Book.objects.distinct()
```

基于双下划线的模糊查询

filter() 方法（exclude 同理）。

注意：filter 中运算符号只能使用等于号 = ，不能使用大于号 > ，小于号 < ，等等其他符号。

__in 用于读取区间，= 号后面为列表 。

__**gt** 大于号 ，= 号后面为数字。__**gte** 大于等于，= 号后面为数字。__lt,__lte

__**range** 在 ... 之间，左闭右闭区间，= 号后面为两个元素的列表。

__**contains** 包含，= 号后面为字符串。

__**icontains** 不区分大小写的包含，= 号后面为字符串。

__**startswith** 以指定字符开头，= 号后面为字符串。

__**endswith** 以指定字符结尾，= 号后面为字符串。

__**year** 是 DateField 数据类型的年份，= 号后面为数字。__day/__month

删除

方式一：使用模型类的 **对象.delete()**。

返回值：元组，第一个元素为受影响的行数。

books=models.Book.objects.filter(pk=8).first().delete()

方式二：使用 QuerySet **类型数据.delete()**(推荐)

返回值：元组，第一个元素为受影响的行数。

books=models.Book.objects.filter(pk__in=[1,2]).delete()

- a. Django 删除数据时，会模仿 SQL 约束 ON DELETE CASCADE 的行为，也就是删除一个对象时也会删除与它相关联的外键对象。
- b. delete() 方法是 QuerySet 数据类型的方法，但并不适用于 Manager 本身。也就是想要删除所有数据，不能不写 all。

books=models.Book.objects.delete() # 报错

books=models.Book.objects.all().delete() # 删除成功

修改

方式一：

模型类的对象.属性 = 更改的属性值

模型类的对象.save()

返回值：编辑的模型类的对象。

books = models.Book.objects.filter(pk=7).first()

books.price = 400

books.save()

方式二：QuerySet 类型数据.update(字段名=更改的数据)（推荐）

返回值：整数，受影响的行数

实例

```
from django.shortcuts import render,HttpResponse
from app01 import models
def add_book(request):
    books = models.Book.objects.filter(pk__in=[7,8]).update(price=888)
```

ORM - 添加数据

一对多(外键 ForeignKey)

方式一: 传对象的形式，返回值的数据类型是对象，书籍对象。

步骤：

- a. 获取出版社对象
- b. 给书籍的出版社属性 pulish 传出版社对象

app01/views.py 文件代码：

```
def add_book(request):
    # 获取出版社对象
    pub_obj = models.Publish.objects.filter(pk=1).first()
    # 给书籍的出版社属性 publish 传出版社对象
    book = models.Book.objects.create(title="菜鸟教程", price=200, pub_date="2010-10-10", publish=pub_obj)
    print(book, type(book))
    return HttpResponse(book)
```

方式二: 传对象 id 的形式(由于传过来的数据一般是 id,所以传对象 id 是常用的)。

一对多中，设置外键属性的类(多的表)中，MySQL 中显示的字段名是:**外键属性名_id**。

返回值的数据类型是对象，书籍对象。

步骤：

- a. 获取出版社对象的 id
- b. 给书籍的关联出版社字段 pulish_id 传出版社对象的 id

多对多(ManyToManyField)：在第三张关系表中新增数据

方式一: 传对象形式，无返回值。

步骤：

- a. 获取作者对象
- b. 获取书籍对象
- c. 给书籍对象的 authors 属性用 add 方法传作者对象

方式二: 传对象 id 形式，无返回值。

步骤：

- a. 获取作者对象的 id
- b. 获取书籍对象
- c. 给书籍对象的 authors 属性用 add 方法传作者对象的 id

app01/views.py 文件代码：

```
def add_book(request):
    # 获取作者对象
    chong = models.Author.objects.filter(name="令狐冲").first()
    # 获取作者对象的 id
    pk = chong.pk
    # 获取书籍对象
    book = models.Book.objects.filter(title="冲灵剑法").first()
    # 给书籍对象的 authors 属性用 add 方法传作者对象的 id
    book.authors.add(pk)
```

关联管理器(对象调用)

前提：

- 多对多（双向均有关联管理器）
- 一对多（只有多的那个类的对象有关联管理器，即反向才有）

语法格式：

正向：属性名

反向：小写类名加 _set

注意：一对多只能反向

常用方法：

add()：用于多对多，把指定的模型对象添加到关联对象集（关系表）中。

注意：add() 在一对多(即外键)中，只能传对象（*QuerySet 数据类型），不能传 id（*[id 表]）。

***[ ]** 的使用:

```
# 方式一：传对象
book_obj = models.Book.objects.get(id=10)
author_list = models.Author.objects.filter(id__gt=2)
book_obj.authors.add(*author_list)  # 将 id 大于 2 的作者对象添加到这本书的作者集合中
# 方式二：传对象 id
book_obj.authors.add(*[1,3]) # 将 id=1 和 id=3 的作者对象添加到这本书的作者集合中
return HttpResponse("ok")
```

反向：**小写表名_set**

```
ying = models.Author.objects.filter(name="任盈盈").first()
book = models.Book.objects.filter(title="冲灵剑法").first()
ying.book_set.add(book)
```

1. 正向查询（Forward Query）

定义

- **从 “持有外键的模型” 访问 “被关联的模型”。**
- **使用 属性名 直接访问关联对象。**

2. 反向查询（Reverse Query）

定义

- 从 **“被关联的模型”** 访问 **“持有外键的模型”**。
- 默认使用 **小写类名_set** 作为关联管理器名称。

remove()：从关联对象集中移除执行的模型对象。

对于 ForeignKey 对象，这个方法仅在 null=True（可以为空）时存在，无返回值。

实例

```
author_obj =models.Author.objects.get(id=1)
book_obj = models.Book.objects.get(id=11)
author_obj.book_set.remove(book_obj)
```

clear()：从关联对象集中移除一切对象，删除关联，不会删除对象。

对于 ForeignKey 对象，这个方法仅在 null=True（可以为空）时存在。

无返回值。

```
#  清空独孤九剑关联的所有作者
book = models.Book.objects.filter(title="菜鸟教程").first()
book.authors.clear()
```

ORM 查询

基于对象的跨表查询。

正向：属性名称

反向：小写类名_set

一对多

查询主键为 1 的书籍的出版社所在的城市（正向）。

实例

```
book = models.Book.objects.filter(pk=10).first()
res = book.publish.city
```

查询明教出版社出版的书籍名（反向）。

反向：**对象.小写类名_set(pub.book_set)** 可以跳转到关联的表(书籍表)。

pub.book_set.all()：取出书籍表的所有书籍对象，在一个 QuerySet 里，遍历取出一个个书籍对象。

实例

```
pub = models.Publish.objects.filter(name="明教出版社").first()
res = pub.book_set.all()
for i in res:
    print(i.title)
```

一对一

查询令狐冲的电话（正向）

正向：对象.属性 (author.au_detail) 可以跳转到关联的表(作者详情表)

实例

```
author = models.Author.objects.filter(name="令狐冲").first()
res = author.au_detail.tel
print(res, type(res))
```

查询所有住址在黑木崖的作者的姓名（反向）。

一对一的反向，用 **对象.小写类名** 即可，不用加 _set。

反向：对象.小写类名(addr.author)可以跳转到关联的表(作者表)。

实例

```
addr = models.AuthorDetail.objects.filter(addr="黑木崖").first()
res = addr.author.name
print(res, type(res))
```

多对多

菜鸟教程所有作者的名字以及手机号（正向）。

正向：**对象.属性(book.authors)**可以跳转到关联的表(作者表)。

作者表里没有作者电话，因此再次通过**对象.属性(i.au_detail)**跳转到关联的表（作者详情表）。

实例

```
book = models.Book.objects.filter(title="菜鸟教程").first()
res = book.authors.all()
for i in res:
    print(i.name, i.au_detail.tel)
```

查询任我行出过的所有书籍的名字（反向）。

实例

```
author = models.Author.objects.filter(name="任我行").first()
res = author.book_set.all()
for i in res:
    print(i.title)
```

正向：属性名称__跨表的属性名称 反向：小写类名__跨表的属性名称

一对多

查询菜鸟出版社出版过的所有书籍的名字与价格。

实例

```
res = models.Book.objects.filter(publish__name="菜鸟出版社").values_list("title", "price")
```

正向：属性名称__跨表的属性名称 反向：小写类名__跨表的属性名称

一对多

查询菜鸟出版社出版过的所有书籍的名字与价格。

实例

res = models.Book.objects.filter(publish__name="菜鸟出版社").values_list("title", "price")

反向：通过 小**写类名__跨表的属性名称（book__title，book__price）** 跨表获取数据。

实例

res = models.Publish.objects.filter(name="菜鸟出版社").values_list("book__title","book__price")

return HttpResponse("ok")

多对多

查询任我行出过的所有书籍的名字。

正向：通过 属性名称__跨表的属性名称(authors__name) 跨表获取数据：

res = models.Book.objects.filter(authors__name="任我行").values_list("title")

反向：通过 小写类名__跨表的属性名称（book__title） 跨表获取数据：

res = models.Author.objects.filter(name="任我行").values_list("book__title")

一对一

查询任我行的手机号。

正向：通过 **属性名称__跨表的属性名称(au_detail__tel)** 跨表获取数据。

res = models.Author.objects.filter(name="任我行").values_list("au_detail__tel")

反向：通过 **小写类名__跨表的属性名称（author__name）** 跨表获取数据。

res = models.AuthorDetail.objects.filter(author__name="任我行").values_list("tel")

Django ORM – 多表实例（聚合与分组查询）

聚合查询（aggregate）

聚合查询函数是对一组值执行计算，并返回单个值。

Django 使用聚合查询前要先从 django.db.models 引入 Avg、Max、Min、Count、Sum（首字母大写）。

from django.db.models import Avg,Max,Min,Count,Sum # 引入函数

聚合查询返回值的数据类型是字典。

聚合函数 aggregate() 是 QuerySet 的一个终止子句， 生成的一个汇总值，相当于 count()。

使用 aggregate() 后，数据类型就变为字典，不能再使用 QuerySet 数据类型的一些 API 了。

日期数据类型(DateField)可以用 Max 和 Min。

返回的字典中：键的名称默认是（属性名称加上__聚合函数名），值是计算出来的聚合值。

如果要自定义返回字典的键的名称，可以起别名：

aggregate(别名 = 聚合函数名("属性名称"))

分组查询（annotate）

分组查询一般会用到聚合函数，所以使用前要先从 django.db.models 引入 Avg,Max,Min,Count,Sum（首字母大写）。

from django.db.models import Avg,Max,Min,Count,Sum # 引入函数

返回值：

- 分组后，用 values 取值，则返回值是 QuerySet 数据类型里面为一个个字典；
- 分组后，用 values_list 取值，则返回值是 QuerySet 数据类型里面为一个个元组。

MySQL 中的 limit 相当于 ORM 中的 QuerySet 数据类型的切片。

注意：

annotate 里面放聚合函数。

- **values 或者 values_list 放在 annotate 前面：**values 或者 values_list 是声明以什么字段分组，annotate 执行分组。前者相当于 GROUP BY,后者相当于聚合函数
- **values 或者 values_list 放在 annotate 后面：** annotate 表示直接以当前表的 pk 执行分组，values 或者 values_list 表示查询哪

些字段， 并且要将 annotate 里的聚合函数起别名，在 values 或者 values_list 里写其别名。

- **annotate 里相当于聚合函数，value 相当于 groupby 了**

```
res = models.Publish.objects.values("name").annotate(in_price = Min("book__price"))
print(res)
res = models.Book.objects.annotate(c = Count("authors__name")).values("title","c")
print(res)
```

F() 查询

F() 的实例可以在查询中引用字段，来比较同一个 model 实例中两个不同字段的值。

之前构造的过滤器都只是将字段值与某个常量做比较，如果想要对两个字段的值做比较，就需要用到 F()。

使用前要先从 django.db.models 引入 F:

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

用法：

```
F("字段名称")
```

F 动态获取对象字段的值，可以进行运算。

Django 支持 F() 对象之间以及 F() 对象和常数之间的加减乘除和取余的操作。

修改操作（update）也可以使用 F() 函数。

查询工资大于年龄的人：

实例

```
from django.db.models import F
...
book=models.Emp.objects.filter(salary__gt=F("age")).values("name","age")
```

Q() 查询

使用前要先从 django.db.models 引入 Q:

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

用法：

```
Q(条件判断)
```

例如：

```
Q(title__startswith="菜")
```

之前构造的过滤器里的多个条件的关系都是 and，如果需要执行更复杂的查询（例如 or 语句），就可以使用 Q 。

Q 对象可以使用 & | ~ （与 或 非）操作符进行组合。

优先级从高到低：~ & |。

可以混合使用 Q 对象和关键字参数，Q 对象和关键字参数是用"and"拼在一起的（即将逗号看成 and ），但是 Q 对象必须位于所有关键字参数的前面。

查询价格大于 350 或者名称以菜开头的书籍的名称和价格。

查询出版日期是 2004 或者 1999 年，并且书名中包含有"菜"的书籍。

Q 对象和关键字混合使用，Q 对象要在所有关键字的前面:

总结

books=models.Book.objects

print(books,type(books))

TestModel.Book.objects <class 'django.db.models.manager.Manager'>

books = models.Book.objects.create(title="如来神掌",price=200,publish="功夫出版社",pub_date="2010-10-10")

print(books, type(books)) # Book object (18)

Book object (7) <class 'TestModel.models.Book'>

books = models.Book.objects.all()

print(books,type(books))

<QuerySet [<Book: Book object (1)>, …,<Book: Book object (7)>]> <class 'django.db.models.query.QuerySet'>

books = models.Book.objects.first() # 返回所有数据的第一条数据

print(books, type(books))

Book object (1) <class 'TestModel.models.Book'>

等同于

books = models.Book.objects.filter() # 返回所有数据的第一条数据

print(books[0], type(books[0]))

Book object (1) <class 'TestModel.models.Book'>

books = models.Book.objects.values("pk","price")

print(books,type(books)) # 得到的是第一条记录的 price 字段的数据

<QuerySet [{'pk': 1, 'price': Decimal('300.00')},… , {'pk': 7, 'price': Decimal('200.00')}]> <class 'django.db.models.query.QuerySet'>

books = models.Book.objects.values_list("price","publish")

print(books,type(books))

<QuerySet [(Decimal('300.00'), '菜鸟出版社'), …. (Decimal('200.00'), '功夫出版社'), (Decimal('200.00'), '功夫出版社')]> <class 'django.db.models.query.QuerySet'>

1. Book 对象（<class 'TestModel.models.Book'>）

- **含义**：**Book** 是 Django 模型（**Model**）的一个实例，对应数据库中的一行记录。
- **作用**：表示单个数据对象，可以访问其字段（如 **book.title**、**book.price**）。
- **示例**：
- **特点**：
  - 是 **TestModel.models.Book** 类的实例。
  - 可以直接修改并保存到数据库（**book.save()**）。

---

2. Book.objects（<class 'django.db.models.manager.Manager'>）

- **含义**：**objects** 是 Django 模型的默认**管理器（Manager）**，用于执行数据库查询操作。
- **作用**：提供查询方法（如 **all()**, **filter()**, **get()**, **create()**），用于操作数据库。
- **示例**：
  - 是 **Manager** 类的实例。
  - 不直接返回数据，而是返回 **QuerySet**（查询集）或单个 Book。
  - 可以自定义管理器（如 **class BookManager(models.Manager)**）。

---

3. QuerySet（<class 'django.db.models.query.QuerySet'>）

- **含义**：**QuerySet** 是 Django 查询数据库返回的**结果集**，可以包含 0 个、1 个或多个 **Book** 对象。
- **作用**：用于链式查询（如过滤、排序、聚合等），但**不会立即执行查询**（惰性加载）。
- **示例**：
  - 是惰性的，只有在真正使用时（如遍历、**list(queryset)**、**len(queryset)**）才会查询数据库。
  - 可以进一步过滤（**filter()**）、排序（**order_by()**）、切片（**[:5]**）等。
  - 如果只返回一个对象（如 **get()**），则返回 **Book** 对象，而不是 **QuerySet**。

2. 为什么 QuerySet 类型不变，但数据形式不同?

- **QuerySet** 是 Django 的**惰性查询集**，它只是**生成 SQL 查询的容器**，具体返回的数据形式由调用的方法决定：
  - **all()** → 返回模型实例。
  - **values()** → 返回字典。
  - **values_list()** → 返回元组或单个值。
- **但 QuerySet 本身的类型始终是 django.db.models.query.QuerySet**，因为它代表的是“查询能力”，而不是具体的数据形式。

---

3. 如何选择合适的方法?

方法	返回的数据形式	适用场景
all()	**QuerySet[Model 实例]**	需要完整模型对象（如调用 **save()**、访问关联字段）
values("field1", "field2")	**QuerySet[dict]**	只需要部分字段，且希望用字段名访问
values_list("field1", "field2")	**QuerySet[tuple]**	只需要字段值，不关心字段名
values_list("field", flat=True)	**QuerySet[单值]**	只需要一个字段的列表（如 **[1, 2, 3]**）

是的！**authors = models.ManyToManyField("Author")** 这一行代码会自动创建一个 **中间表（关联表）** 来存储多对多关系，但不会直接修改 **Author** 表或当前模型表的结构。以下是详细解释：

---

1. 自动创建的中间表

Django 会为你生成一个名为 **<当前模型表名>_authors** 的中间表（例如，如果当前模型是 **Book**，则表名为 **book_authors**），其结构如下：

```
CREATE TABLE book_authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,  -- SQLite 示例
    book_id INTEGER NOT NULL,              -- 外键指向 Book 表
    author_id INTEGER NOT NULL,            -- 外键指向 Author 表
    FOREIGN KEY (book_id) REFERENCES book(id),
    FOREIGN KEY (author_id) REFERENCES author(id)
);
```

- 这个表只有 3 个字段：**id**（主键）、**book_id**（外键）、**author_id**（外键）。
- 你 **不需要手动定义这个表**，Django 在运行 **makemigrations** 和 **migrate** 时会自动创建。

5. 注意事项

1. **表名规则**：

- 自动生成的中间表名格式：**<当前模型表名>_<字段名>**（如 **book_authors**）。
- 如果当前模型表名是 **app_book**，则中间表名为 **app_book_authors**。

2. **迁移顺序**：
   - 确保 **Author** 模型已正确定义并先迁移，否则会报错。
3. **数据库一致性**：
   - 不要手动修改中间表，始终通过 Django ORM 操作（如 **add()**、**remove()**）。

总结

行为	结果
定义 **ManyToManyField**	Django **自动创建中间表**
默认中间表字段	**id**、**<当前模型>_id**、**<关联模型>_id**
自定义中间表	使用 **through=** 参数指定模型
操作关系	通过 **add()**、**remove()**、**clear()** 等方法

在 Django 中，当你定义一个 **ForeignKey** 字段时，例如：

python

复制

下载

```
publish = models.ForeignKey("Publish", on_delete=models.CASCADE)
```

Django 实际上会在数据库中创建一个名为 **publish_id** 的列，而不是 **publish**。这是由 Django 的设计机制决定的，原因如下：

---

1. 为什么字段名是 publish_id 而不是 publish?

(1) 数据库层面的外键存储

- 在关系型数据库（如 MySQL、PostgreSQL、SQLite）中，外键关系是通过 **存储目标表的主键值** 来实现的。
- 例如，如果 **Publish** 表的主键是 **id**，那么 **Book** 表会有一个 **publish_id** 列，存储的是 **Publish** 表中某条记录的 **id** 值。
- **1. 为什么列名不同也能关联?**
- 在关系型数据库（如 MySQL、PostgreSQL、SQLite）中，**外键关联的本质是值匹配，而不是列名匹配**。只要满足以下条件即可：
-

(2) Django 的 ORM 抽象

- 虽然数据库列名是 **publish_id**，但 Django 在 **Python 模型层面** 提供了一个名为 **publish** 的属性，用于方便地访问关联对象。
- 当你访问 **book.publish** 时，Django 会自动执行查询，获取对应的 **Publish** 对象。
- 而 **book.publish_id** 则直接返回数据库中存储的原始外键值（即 **Publish** 表的 **id**）。

2. Django 如何实现这种关联?

（1）数据库层面

- Django 生成的 SQL 会明确指定外键关联的目标列：

sql

复制

下载

```
FOREIGN KEY (publish_id) REFERENCES Publish(id)
```

这里显式声明了 **publish_id** 关联到 **Publish.id**，与列名无关。

（2）ORM 层面

- 当你访问 **book.publish** 时，Django 会自动执行类似以下的查询：

SELECT * FROM Publish WHERE id = book.publish_id;

通过 **publish_id** 的值找到对应的 **Publish** 记录。

---

3. 核心区别对比

特性	**.add()**	**.create()**
作用	关联已存在的对象	创建新对象并关联
是否创建新对象	□ 否	□ 是
适用关系类型	**ManyToManyField**, **ForeignKey**	**ManyToManyField**, **ForeignKey** 的反向查询
参数类型	对象或 ID	新对象的字段键值对
返回值	无	新创建的对象

Django Form 组件

Django Form 组件用于对页面进行初始化，生成 HTML 标签，此外还可以对用户提交的数据进行校验（显示错误信息）。

报错信息显示顺序：

先显示字段属性中的错误信息，然后再显示局部钩子的错误信息。

若显示了字段属性的错误信息，就不会显示局部钩子的错误信息。

若有全局钩子，则全局钩子是等所有的数据都校验完，才开始进行校验，并且全局钩子的错误信息一定会显示。

Django 用户认证（Auth）组件

Django 用户认证（Auth）组件一般用在用户的登录注册上，用于判断当前的用户是否合法，并跳转到登陆成功或失败页面。

Django 用户认证（Auth）组件需要导入 auth 模块:

```
# 认证模块
from django.contrib import auth

# 对应数据库
from django.contrib.auth.models import User
```

返回值是用户对象。

创建用户对象的三种方法：

- **create()**：创建一个普通用户，密码是明文的。

- **create_user()**：创建一个普通用户，密码是密文的。
- **create_superuser()**：创建一个超级用户，密码是密文的，要多传一个邮箱 email 参数。

验证用户的用户名和密码使用 authenticate() 方法，从需要 auth_user 表中过滤出用户对象。

使用前要导入：

from django.contrib import auth

参数：

username：用户名

password：密码

返回值：如果验证成功，就返回用户对象，反之，返回 None。

给验证成功的用户加 session，将 request.user 赋值为用户对象。

登陆使用 login() 方法。

使用前要导入：

from django.contrib import auth

参数：

- request：用户对象

注销用户使用 logout() 方法，需要清空 session 信息，将 request.user 赋值为匿名用户。

使用前要导入：

from django.contrib import auth

参数：

- request：用户对象

返回值：None

设置装饰器，给需要登录成功后才能访问的页面统一加装饰器。

使用前要导入：

from django.contrib.auth.decorators import login_required

实例

```
from django.contrib.auth.decorators import login_required @login_required
def index(request):
 return HttpResponse("index 页面。。。")
```

设置从哪个页面访问，登录成功后就返回哪个页面。

解析：

django 在用户访问页面时，如果用户是未登录的状态，就给用户返回登录页面。

此时，该登录页面的 URL 后面有参数：next=用户访问的页面的 URL。

因此，设置在用户登录成功后重定向的 URL 为 next 参数的值。

但是，若用户一开始就输入登录页面 logi，request.GET.get("next") 就取不到值，所以在后面加 or，可以设置自定义返回的页面。

Django cookie 与 session

Cookie 是存储在客户端计算机上的文本文件，并保留了各种跟踪信息。

识别返回用户包括三个步骤：

- 服务器脚本向浏览器发送一组 Cookie。例如：姓名、年龄或识别号码等。

- 浏览器将这些信息存储在本地计算机上，以备将来使用。
- 当下一次浏览器向 Web 服务器发送任何请求时，浏览器会把这些 Cookie 信息发送到服务器，服务器将使用这些信息来识别用户。

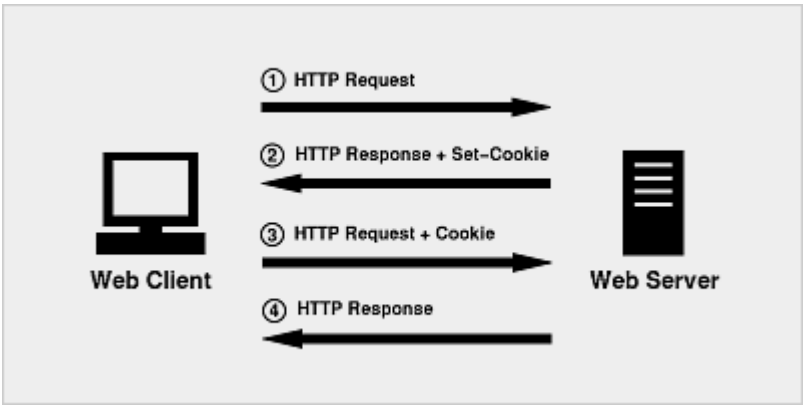
HTTP 是一种"无状态"协议，这意味着每次客户端检索网页时，客户端打开一个单独的连接到 Web 服务器，服务器会自动不保留之前客户端请求的任何记录。

但是仍然有以下三种方式来维持 Web 客户端和 Web 服务器之间的 session 会话：

Cookies

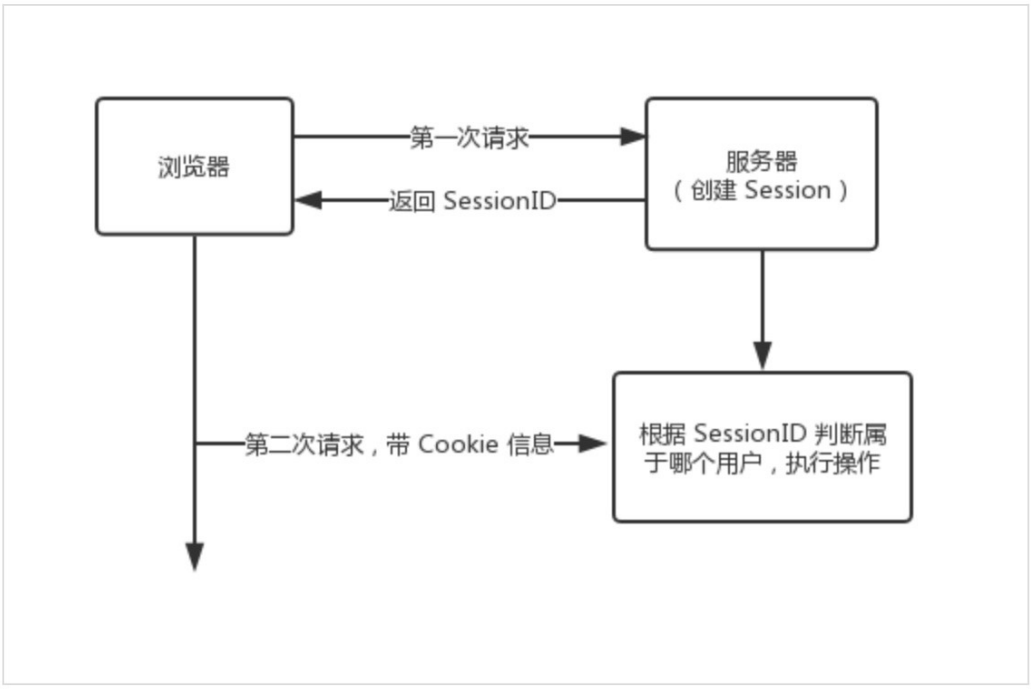
一个 Web 服务器可以分配一个唯一的 session 会话 ID 作为每个 Web 客户端的 cookie，对于客户端的后续请求可以使用接收到的 cookie 来识别。

在 Web 开发中，使用 session 来完成会话跟踪，session 底层依赖 Cookie 技术。



Session(保存在服务端的键值对)

服务器在运行时可以为每一个用户的浏览器创建一个其独享的 session 对象，由于 session 为用户浏览器独享，所以用户在访问服务器的 web 资源时，可以把各自的数据放在各自的 session 中，当用户再去访问该服务器中的其它 web 资源时，其它 web 资源再从用户各自的 session 中取出数据为用户服务。



工作原理

- a. 浏览器第一次请求获取登录页面 login。
- b. 浏览器输入账号密码第二次请求，若输入正确，服务器响应浏览器一个 index 页面和一个键为 sessionid，值为随机字符串的 cookie，即 set_cookie ("sessionid",随机字符串)。
- c. 服务器内部在 django.session 表中记录一条数据。

django.session 表中有三个字段。

  - session_key：存的是随机字符串，即响应给浏览器的 cookie 的 sessionid 键对应的值。
  - session_data：存的是用户的信息，即多个 request.session["key"]=value，且是密文。
  - expire_date：存的是该条记录的过期时间（默认 14 天）

- d. 浏览器第三次请求其他资源时，携带 cookie :{sessionid:随机字符串}，服务器从 django.session 表中根据该随机字符串取出该用户的数据，供其使用（即保存状态）。

注意: django.session 表中保存的是浏览器的信息，而不是每一个用户的信息。 因此， 同一浏览器多个用户请求只保存一条记录（后面覆盖前面）,多个浏览器请求才保存多条记录。

cookie 弥补了 http 无状态的不足，让服务器知道来的人是"谁"，但是 cookie 以文本的形式保存在浏览器端，安全性较差，且最大只支持 4096 字节，所以只通过 cookie 识别不同的用户，然后，在对应的 session 里保存私密的信息以及超过 4096 字节的文本。

session 设置：

request.session["key"] = value

执行步骤：

- a. 生成随机字符串
- b. 把随机字符串和设置的键值对保存到 django_session 表的 session_key 和 session_data 里
- c. 设置 **cookie：set_cookie("sessionid",随机字符串)** 响应给浏览器

session 获取：

request.session.get('key')

执行步骤：

- a. 从 cookie 中获取 sessionid 键的值，即随机字符串。
- b. 根据随机字符串从 django_session 表过滤出记录。
- c. 取出 session_data 字段的数据。

session 删除，删除整条记录（包括 session_key、session_data、expire_date 三个字段）：

request.session.flush()

删除 session_data 里的其中一组键值对：

del request.session["key"]

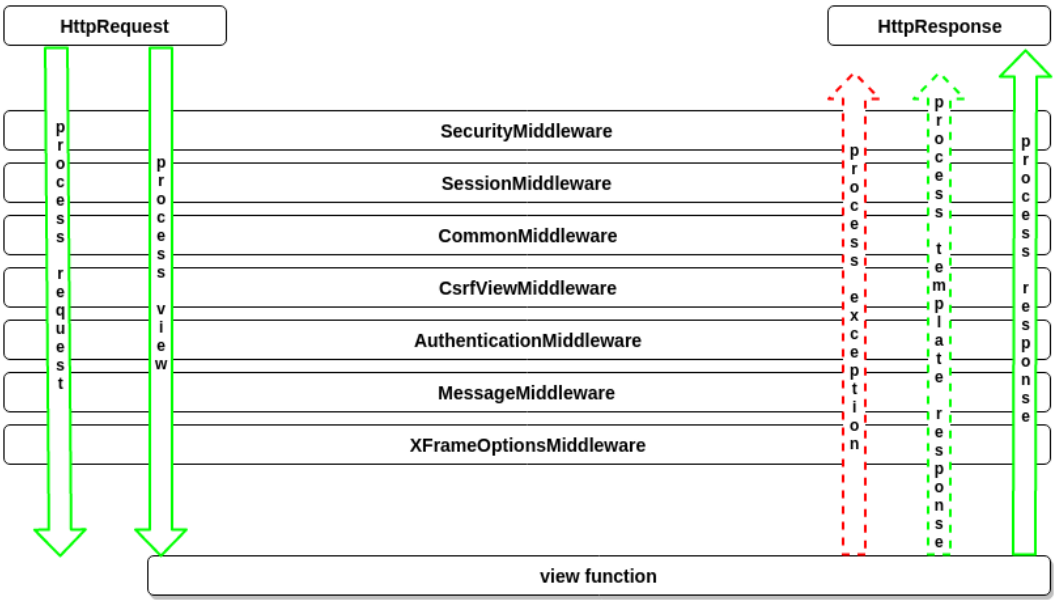
执行步骤：

- a. 从 cookie 中获取 sessionid 键的值，即随机字符串
- b. 根据随机字符串从 django_session 表过滤出记录
- c. 删除过滤出来的记录

Django 中间件

Django 中间件是修改 Django request 或者 response 对象的钩子，可以理解为是介于 HttpRequest 与 HttpResponse 处理之间的一道处理过程。

浏览器从请求到响应的过程中，Django 需要通过很多中间件来处理，可以看如下图所示：



Django 中间件作用：

- 修改请求，即传送到 view 中的 HttpRequest 对象。
- 修改响应，即 view 返回的 HttpResponse 对象。

中间件组件配置在 settings.py 文件的 MIDDLEWARE 选项列表中。
配置中的每个字符串选项都是一个类，也就是一个中间件。
自定义中间件
中间件可以定义四个方法，分别是：

process_request(self,request)
process_view(self, request, view_func, view_args, view_kwargs)
process_exception(self, request, exception)
process_response(self, request, response)