

Table of Contents

Introduction	1.1
第一章：学前准备	1.2
第一节：虚拟环境	1.2.1
第二节：准备工作	1.2.2
第三节：Django介绍	1.2.3
第四节：URL组成部分	1.2.4
第二章：URL与视图	1.3
第一节：第一个Django项目	1.3.1
第二节：视图与URL分发器	1.3.2
第三章：模板	1.4
第一节：模板介绍	1.4.1
第二节：模板变量	1.4.2
第三节：常用标签	1.4.3
第四节：常用过滤器	1.4.4
第五节：自定义过滤器	1.4.5
第七节：模版结构优化	1.4.6
第八节：加载静态文件	1.4.7
第四章：数据库	1.5
第一节：MySQL相关软件	1.5.1
第二节：数据库操作	1.5.2
第三节：ORM模型	1.5.3
第四节：模型常用字段	1.5.4
第五节：外键和表关系	1.5.5
第六节：增删改查操作	1.5.6
第七节：查询操作	1.5.7
第八节：QuerySet API	1.5.8
第九节：ORM模型迁移	1.5.9
第十节：ORM作业	1.5.10
第十一节：ORM作业参考答案	1.5.11
第十二节：Pycharm连接数据库	1.5.12
第五章：视图高级	1.6

第一节：限制请求method	1.6.1
第二节：页面重定向	1.6.2
第三节：HttpRequest对象	1.6.3
第四节：HttpResponse对象	1.6.4
第五节：生成CSV文件	1.6.5
第六节：类视图	1.6.6
第七节：错误处理	1.6.7
第八节：分页	1.6.8
第六章：表单	1.7
第一节：表单概述	1.7.1
第二节：用表单验证数据	1.7.2
第三节：ModelForm	1.7.3
第四节：文件上传	1.7.4
第七章：cookie和session	1.8
第八章：上下文处理器和中间件	1.9
第一节：上下文处理器	1.9.1
第二节：中间件	1.9.2
第九章：安全	1.10
第一节：CSRF攻击	1.10.1
第二节：XSS攻击	1.10.2
第三节：点击劫持攻击	1.10.3
第四节：SQL注入	1.10.4
第十章：信号	1.11
第一节：什么是信号	1.11.1
第十一章：验证和授权	1.12
第一节：概述	1.12.1
第二节：用户对象	1.12.2
第三节：权限和分组	1.12.3
第十二章：Admin系统	1.13
第一节：admin系统	1.13.1
第十三章：Django的缓存	1.14
1.部署	1.14.1
第十四章：memcached	1.15
第十五章：Redis	1.16

知了课堂Django课程教学课件

本课件是知了课堂《[超详细讲解Django打造大型企业官网](#)》VIP配套课件。请勿用于商业用途，如有转载，请注明出处！

虚拟环境

为什么需要虚拟环境：

到目前位置，我们所有的第三方包安装都是直接通过 `pip install xx` 的方式进行安装的，这样安装会将那个包安装到你的系统级的 `Python` 环境中。但是这样有一个问题，就是如果你现在用 `Django 1.10.x` 写了个网站，然后你的领导跟你说，之前有一个旧项目是用 `Django 0.9` 开发的，让你来维护，但是 `Django 1.10` 不再兼容 `Django 0.9` 的一些语法了。这时候就会碰到一个问题，我如何在我的电脑中同时拥有 `Django 1.10` 和 `Django 0.9` 两套环境呢？这时候我们就可以通过虚拟环境来解决这个问题。

虚拟环境原理介绍：

虚拟环境相当于一个抽屉，在这个抽屉中安装的任何软件包都不会影响到其他抽屉。并且在项目中，我可以指定这个项目的虚拟环境来配合我的项目。比如我们现在有一个项目是基于 `Django 1.10.x` 版本，又有一个项目是基于 `Django 0.9.x` 的版本，那么这时候就可以创建两个虚拟环境，在这两个虚拟环境中分别安装 `Django 1.10.x` 和 `Django 0.9.x` 来适配我们的项目。

安装 `virtualenv`：

`virtualenv` 是用来创建虚拟环境的软件工具，我们可以通过 `pip` 或者 `pip3` 来安装：

```
pip install virtualenv
pip3 install virtualenv
```

创建虚拟环境：

创建虚拟环境非常简单，通过以下命令就可以创建了：

```
virtualenv [虚拟环境的名字]
```

如果你当前的 `Python3/Scripts` 的查找路径在 `Python2/Scripts` 的前面，那么将会使用 `python3` 作为这个虚拟环境的解释器。如果 `python2/Scripts` 在 `python3/Scripts` 前面，那么将会使用 `Python2` 来作为这个虚拟环境的解释器。

进入环境：

虚拟环境创建好了以后，那么可以进入到这个虚拟环境中，然后安装一些第三方包，进入虚拟环境在不同的操作系统中有不同的方式，一般分为两种，第一种是 `Windows`，第二种是 `*nix`：

1. `windows` 进入虚拟环境：进入到虚拟环境的 `Scripts` 文件夹中，然后执行 `activate`。

2. *nix 进入虚拟环境: `source /path/to/virtualenv/bin/activate`

一旦你进入到了这个虚拟环境中, 你安装包, 卸载包都是在这个虚拟环境中, 不会影响到外面的环境。

退出虚拟环境:

退出虚拟环境很简单, 通过一个命令就可以完成: `deactivate`。

创建虚拟环境的时候指定 Python 解释器:

在电脑的环境变量中, 一般是不会去更改一些环境变量的顺序的。也就是说比如你的 `Python2/Scripts` 在 `Python3/Scripts` 的前面, 那么你不会经常去更改他们的位置。但是这时候我确实是想在创建虚拟环境的时候用 `Python3` 这个版本, 这时候可以通过 `-p` 参数来指定具体的 `Python` 解释器:

```
virtualenv -p C:\Python36\python.exe [virtualenv name]
```

virtualenvwrapper:

`virtualenvwrapper` 这个软件包可以让我们管理虚拟环境变得更加简单。不用再跑到某个目录下通过 `virtualenv` 来创建虚拟环境, 并且激活的时候也要跑到具体的目录下去激活。

安装 `virtualenvwrapper` :

1. *nix: `pip install virtualenvwrapper`。
2. windows: `pip install virtualenvwrapper-win`。

`virtualenvwrapper` 基本使用:

1. 创建虚拟环境:

```
mkvirtualenv my_env
```

那么会在你当前用户下创建一个 `Env` 的文件夹, 然后将这个虚拟环境安装到这个目录下。如果你电脑中安装了 `python2` 和 `python3`, 并且两个版本中都安装了 `virtualenvwrapper`, 那么将会使用环境变量中第一个出现的 `Python` 版本来作为这个虚拟环境的 `Python` 解释器。

2. 切换到某个虚拟环境:

```
workon my_env
```

3. 退出当前虚拟环境:

```
deactivate
```

4. 删除某个虚拟环境:

```
rmvirtualenv my_env
```

5. 列出所有虚拟环境:

```
lsvirtualenv
```

6. 进入到虚拟环境所在的目录:

```
cdvirtualenv
```

修改 `mkvirtualenv` 的默认路径:

在 我的电脑->右键->属性->高级系统设置->环境变量->系统变量 中添加一个参数 `WORKON_HOME` , 将这个参数的值设置为你需要的路径。

创建虚拟环境的时候指定 `Python` 版本:

在使用 `mkvirtualenv` 的时候, 可以指定 `--python` 的参数来指定具体的 `python` 路径:

```
mkvirtualenv --python==C:\Python36\python.exe hy_env
```

学前准备

在学习 Django 之前，需要做好以下准备工作：

1. 确保已经安装 Python 3.6 以上的版本，教学以 Python 3.6 版本进行讲解。
2. 安装 virtualenvwrapper，这个是用来创建虚拟环境的包，使用虚拟环境可以让我们的包管理更加的方便，也为以后项目上线需要安装哪些包做好了准备工作。安装方式在不同的操作系统有区别。以下解释下：
 - windows: `pip instal virtualenvwrapper-win`。
 - linux/mac: `pip install virtualenvwrapper`。
3. 虚拟环境相关操作：
 - 创建虚拟环境: `mkvirtualenv --python='[python3.6文件所在路径]' [虚拟环境名字]`。比如 `mkvirtualenv --python='C:\Python36\python3.6' django-env`。
 - 进入到虚拟环境: `workon [虚拟环境名称]`。比如 `workon django-env`。
 - 退出虚拟环境: `deactivate`。
4. 首先进入到虚拟环境 `workon django-env`，然后通过 `pip install django==2.0` 安装 django，教学以 Django 2.0 版本为例进行讲解。
5. 安装 pycharm profession 2017版 或者 Sublime Text 3 等任意一款你喜欢的编辑器。（推荐使用 pycharm，如果由于电脑性能原因，可以退而求其次使用 Sublime Text）。如果使用 pycharm，切记一定要下载**profession**（专业版），**community**（社区版）不能用于网页开发。至于破解和正版，大家到网上搜下就知道啦。
6. 安装最新版 MySQL，windows 版的 MySQL 的下载地址是：<https://dev.mysql.com/downloads/windows/installer/5.7.html>。如果你用的是其他操作系统，那么可以来到这个界面选择具体的 MySQL 来进行下载：<https://dev.mysql.com/downloads/mysql/>。
7. 安装 pymysql，这个库是 Python 来操作数据库的。没有他，django 就不能操作数据库。安装方式也比较简单，`pip install pymysql` 就可以啦。

建议：建议使用和课程中一样的环境来学习，避免环境问题造成一些莫名其妙的错误影响学习进度和效率。

Django介绍:

Django, 发音为[ˈdʒæŋɡəʊ], Django诞生于2003年秋天, 2005年发布正式版本, 由Simon和Andrian开发。当时两位作者的老板和记者要他们几天甚至几个小时之内增加新的功能。两人不得已开发了Django这套框架以实现快速开发目的, 因此Django生来就是为了节省开发者时间的。Django发展至今, 被许许多多国内外的开发者使用, 已经成为web开发者的首选框架。因此, 如果你是用python来做网站, 没有理由不学好Django。

选读:

1. [Python+Django如何支撑了7 亿月活用户的Instagram?](#)
2. [Django商业网站](#)

Django版本和Python版本:

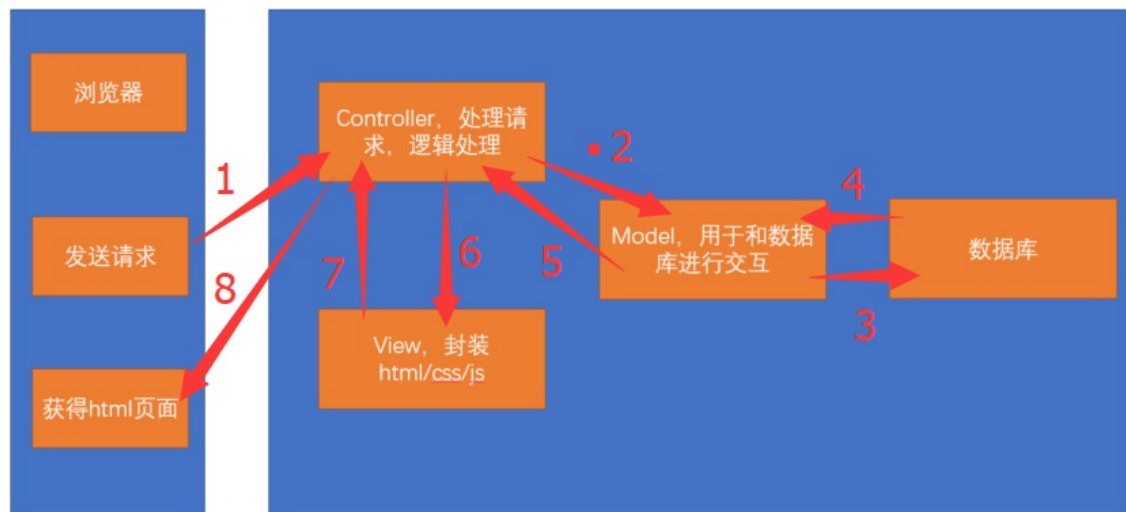
Django version	Python versions
1.8	2.7, 3.2 (until the end of 2016), 3.3, 3.4, 3.5
1.9, 1.10	2.7, 3.4, 3.5
1.11	2.7, 3.4, 3.5, 3.6
2.0	3.4, 3.5, 3.6
2.1	3.5, 3.6, 3.7

web服务器和应用服务器以及web应用框架:

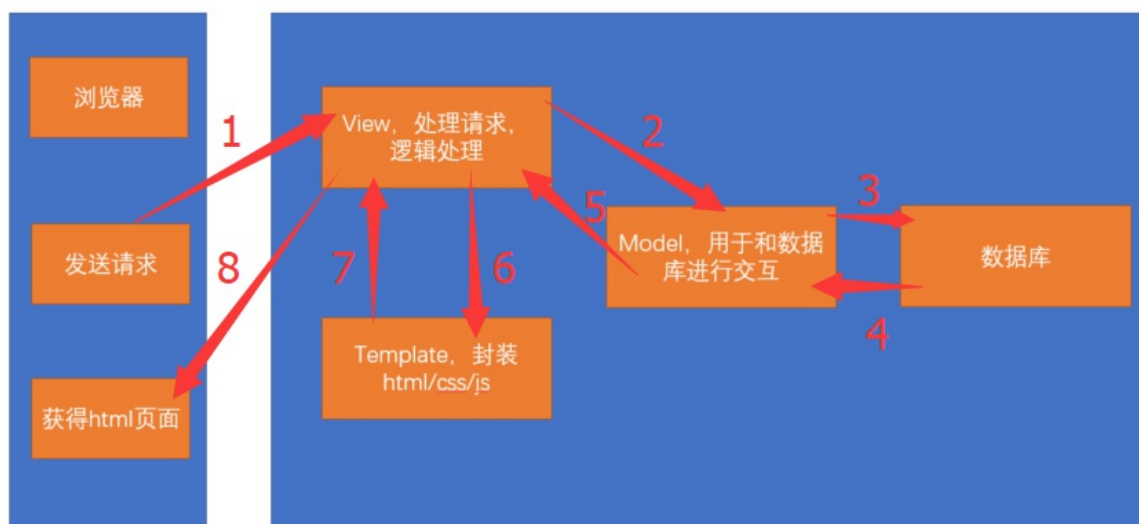
- **web服务器**: 负责处理http请求, 响应静态文件, 常见的有 Apache , Nginx 以及微软的 IIS .
- **应用服务器**: 负责处理逻辑的服务器。比如 php 、 python 的代码, 是不能直接通过 nginx 这种web服务器来处理的, 只能通过应用服务器来处理, 常见的应用服务器有 uwsgi 、 tomcat 等。
- **web应用框架**: 一般使用某种语言, 封装了常用的 web 功能的框架就是web应用框架, flask 、 Django 以及Java中的 SSH(Structs2+Spring3+Hibernate3) 框架都是web应用框架。

Django和MVC:

Django是一个遵循 MVC 设计模式的框架, MVC 是 Model 、 View 、 Controller 的三个单词的简写。分别代表 模型 、 视图 、 控制器 。以下图片说明这三者之间的关系:



而 Django 其实也是一个 MTV 的设计模式。MTV 是 Model、Template、View 三个单词的简写。分别代表 模型、模版、视图。以下图片说明这三者之间的关系：



更多：

1. Django 的官网: <https://www.djangoproject.com/>
2. Django Book 2.0版本的中文文档: <http://djangobook.py3k.cn/2.0/chapter01/>
3. Django 2.0版本的中文文档: <http://python.usyiyi.cn/translate/django2/index.html>

URL组成部分详解:

URL 是 Uniform Resource Locator 的简写，统一资源定位符。

一个 URL 由以下几部分组成:

```
scheme://host:port/path/?query-string=xxx#anchor
```

- **scheme:** 代表的是访问的协议，一般为 http 或者 https 以及 ftp 等。
- **host:** 主机名，域名，比如 www.baidu.com 。
- **port:** 端口号。当你访问一个网站的时候，浏览器默认使用80端口。
- **path:** 查找路径。比如: www.jianshu.com/trending/now ，后面的 trending/now 就是 path 。
- **query-string:** 查询字符串，比如: www.baidu.com/s?wd=python ，后面的 wd=python 就是查询字符串。
- **anchor:** 锚点，后台一般不用管，前端用来做页面定位的。

注意: URL 中的所有字符都是 ASCII 字符集，如果出现非 ASCII 字符，比如中文，浏览器会进行编码再进行传输。

第一个Django项目

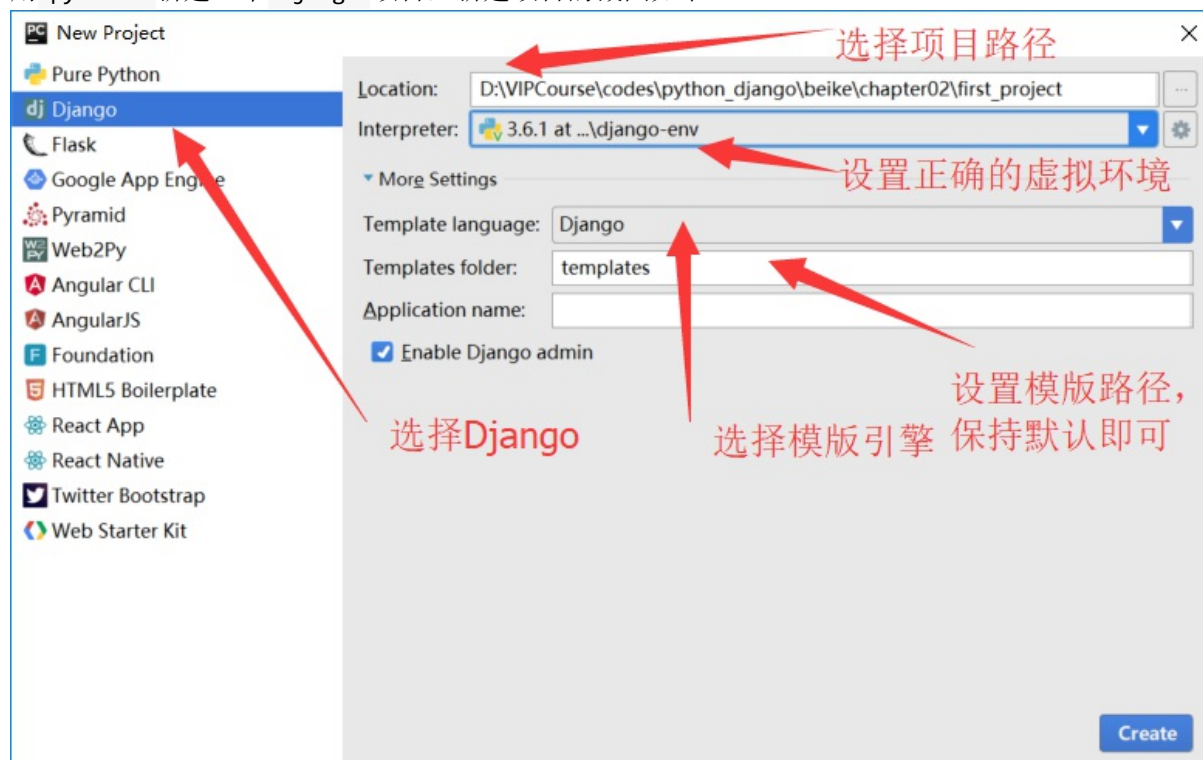
创建 Django 项目：

用命令行的方式：

1. 创建项目：打开终端，使用命令：`django-admin startproject [项目名称]` 即可创建。比如：`django-admin startproject first_project`。
2. 创建应用（app）：一个项目类似于是一个架子，但是真正起作用的还是 app。在终端进入到项目所在的路径，然后执行 `python manage.py startapp [app名称]` 创建一个app。

用 pycharm 的方式：

用 pycharm 新建一个 Django 项目，新建项目的截图如下：



使用 pycharm 创建完项目后，还是需要重新进入到命令行单独创建 app 的。

运行Django项目：

1. 通过命令行的方式：`python manage.py runserver`。这样可以在本地访问你的网站，默认端口号是 8000，这样就可以在浏览器中通过 `http://127.0.0.1:8000/` 来访问你的网站啦。如果想要修改端口号，那么在运行的时候可以指定端口号，`python manage.py runserver 9000` 这样就可以通过 9000 端口来访问啦。另外，这样运行的项目只能在本机上能访问，如果想要在其他电脑上也能访问本网站，那么需要指定 ip 地址为 0.0.0.0。示例为：`python`

`manage.py runserver 0.0.0.0:8000` 。

2. 通过 `pycharm` 运行。直接点击右上角的绿色箭头按钮即可运行。

项目结构介绍：

1. `manage.py` ：以后和项目交互基本上都是基于这个文件。一般都是在终端输入 `python manage.py [子命令]` 。可以输入 `python manage.py help` 看下能做什么事情。除非你知道你自己正在做什么，一般情况下不应该编辑这个文件。
2. `settings.py` ：本项目的设置项，以后所有和项目相关的配置都是放在这个里面。
3. `urls.py` ：这个文件是用来配置URL路由的。比如访问 `http://127.0.0.1/news/` 是访问新闻列表页，这些东西就需要在这个文件中完成。
4. `wsgi.py` ：项目与 `WSGI` 协议兼容的 `web` 服务器入口，部署的时候需要用到的，一般情况下也是不需要修改的。

project和app的关系：

`app` 是 `django` 项目的组成部分。一个 `app` 代表项目中的一个模块，所有 `URL` 请求的响应都是由 `app` 来处理。比如豆瓣，里面有图书，电影，音乐，同城等许许多多的模块，如果站在 `django` 的角度来看，图书，电影这些模块就是 `app` ，图书，电影这些 `app` 共同组成豆瓣这个项目。因此这里要有一个概念，`django` 项目由许多 `app` 组成，一个 `app` 可以被用到其他项目，`django` 也能拥有不同的 `app` 。

URL分发器

视图:

视图一般都写在 `app` 的 `views.py` 中。并且视图的第一个参数永远都是 `request`（一个 `HttpRequest`）对象。这个对象存储了请求过来的所有信息，包括携带的参数以及一些头部信息等。在视图中，一般是完成逻辑相关的操作。比如这个请求是添加一篇博客，那么可以通过 `request` 来接收到这些数据，然后存储到数据库中，最后再把执行的结果返回给浏览器。视图函数的返回结果必须是 `HttpResponseBase` 对象或者子类的对象。示例代码如下：

```
from django.http import HttpResponse
def book_list(request):
    return HttpResponse("书籍列表！")
```

URL映射:

视图写完后，要与URL进行映射，也即用户在浏览器中输入什么 `url` 的时候可以请求到这个视图函数。在用户输入了某个 `url`，请求到我们的网站的时候，`django` 会从项目的 `urls.py` 文件中寻找对应的视图。在 `urls.py` 文件中有一个 `urlpatterns` 变量，以后 `django` 就会从这个变量中读取所有的匹配规则。匹配规则需要使用 `django.urls.path` 函数进行包裹，这个函数会根据传入的参数返回 `URLPattern` 或者是 `URLResolver` 的对象。示例代码如下：

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/', views.book_list)
]
```

URL中添加参数:

有时候，`url` 中包含了一些参数需要动态调整。比如简书某篇文章的详情页的url，是 `https://www.jianshu.com/p/a5aab9c4978e` 后面的 `a5aab9c4978e` 就是这篇文章的 `id`，那么简书的文章详情页的url就可以写成 `https://www.jianshu.com/p/<id>`，其中`id`就是文章的id。那么如何在 `django` 中实现这种需求呢。这时候我们可以在 `path` 函数中，使用尖括号的形式来定义一个参数。比如我现在想要获取一本书籍的详细信息，那么应该在 `url` 中指定这个参数。示例代码如下：

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

```

from django.urls import path
from book import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/', views.book_list),
    path('book/<book_id>', views.book_detail)
]

```

而 `views.py` 中的代码如下：

```

def book_detail(request, book_id):
    text = "您输入的书籍的id是: %s" % book_id
    return HttpResponse(text)

```

当然，也可以通过查询字符串的方式传递一个参数过去。示例代码如下：

```

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/', views.book_list),
    path('book/detail/', views.book_detail)
]

```

在 `views.py` 中的代码如下：

```

def book_detail(request):
    book_id = request.GET.get("id")
    text = "您输入的书籍id是: %s" % book_id
    return HttpResponse(text)

```

以后在访问的时候就是通过 `/book/detail/?id=1` 即可将参数传递过去。

URL中包含另外一个urls模块：

在我们的项目中，不可能只有一个 `app`，如果把所有的 `app` 的 `views` 中的视图都放在 `urls.py` 中进行映射，肯定会让代码显得非常乱。因此 `django` 给我们提供了一个方法，可以在 `app` 内部包含自己的 `url` 匹配规则，而在项目的 `urls.py` 中再统一包含这个 `app` 的 `urls`。使用这个技术需要借助 `include` 函数。示例代码如下：

```

# first_project/urls.py文件:

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

```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',include("book.urls"))
]
```

在 `urls.py` 文件中把所有的和 `book` 这个 `app` 相关的 `url` 都移动到 `app/urls.py` 中了，然后在 `first_project/urls.py` 中，通过 `include` 函数包含 `book.urls`，以后在请求 `book` 相关的 `url` 的时候都需要加一个 `book` 的前缀。

```
# book/urls.py文件:

from django.urls import path
from . import views

urlpatterns = [
    path('list/',views.book_list),
    path('detail/<book_id>/',views.book_detail)
]
```

以后访问书的列表的 `url` 的时候，就通过 `/book/list/` 来访问，访问书籍详情页面的 `url` 的时候就通过 `book/detail/<id>` 来访问。

path函数:

`path` 函数的定义为: `path(route,view,name=None,kwargs=None)`。以下对这几个参数进行讲解。

1. **route 参数:** `url` 的匹配规则。这个参数中可以指定 `url` 中需要传递的参数，比如在访问文章详情页的时候，可以传递一个 `id`。传递参数是通过 `<>` 尖括号来进行指定的。并且在传递参数的时候，可以指定这个参数的数据类型，比如文章的 `id` 都是 `int` 类型，那么可以这样写 `<int:id>`，以后匹配的时候，就只会匹配到 `id` 为 `int` 类型的 `url`，而不会匹配其他的 `url`，并且在视图函数中获取这个参数的时候，就已经被转换成一个 `int` 类型了。其中还有几种常用的类型：
 - **str:** 非空的字符串类型。默认的转换器。但是不能包含斜杠。
 - **int:** 匹配任意的零或者正数的整形。到视图函数中就是一个 `int` 类型。
 - **slug:** 由英文中的横杠 `-`，或者下划线 `_` 连接英文字符或者数字而成的字符串。
 - **uuid:** 匹配 `uuid` 字符串。
 - **path:** 匹配非空的英文字符串，可以包含斜杠。
2. **view 参数:** 可以为一个视图函数或者是类视图 `as_view()` 或者是 `django.urls.include()` 函数的返回值。
3. **name 参数:** 这个参数是给这个 `url` 取个名字的，这在项目比较大，`url` 比较多时候用处很大。

4. `kwargs` 参数: 有时候想给视图函数传递一些额外的参数, 就可以通过 `kwargs` 参数进行传递。这个参数接收一个字典。传到视图函数中的时候, 会作为一个关键字参数传过去。比如以下的 `url` 规则:

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

urlpatterns = [
    path('blog/<int:year>/', views.year_archive, {'foo': 'bar'}),
]
```

那么以后在访问 `blog/1991/` 这个url的时候, 会将 `foo=bar` 作为关键字参数传给 `year_archive` 函数。

re_path函数:

有时候我们在写url匹配的时候, 想要写使用正则表达式来实现一些复杂的需求, 那么这时候我们可以使用 `re_path` 来实现。`re_path` 的参数和 `path` 参数一模一样, 只不过第一个参数也就是 `route` 参数可以为一个正则表达式。

一些使用 `re_path` 的示例代码如下:

```
from django.urls import path, re_path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    re_path(r'articles/(?P<year>[0-9]{4})/', views.year_archive),
    re_path(r'articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/', views.month_archive),
    re_path(r'articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/', views.article_detail),
]
```

以上例子中我们可以看到, 所有的 `route` 字符串前面都加了一个 `r`, 表示这个字符串是一个原生字符串。在写正则表达式中是推荐使用原生字符串的, 这样可以避免在 `python` 这一层面进行转义。而且, 使用正则表达式捕获参数的时候, 是用一个圆括号进行包裹, 然后这个参数的名字是通过尖括号 `<year>` 进行包裹, 之后才是写正则表达式的语法。

include函数:

在项目变大以后, 经常不会把所有的 `url` 匹配规则都放在项目的 `urls.py` 文件中, 而是每个 `app` 都有自己的 `urls.py` 文件, 在这个文件中存储的都是当前这个 `app` 的所有 `url` 匹配规则。然后再统一注册到项目的 `urls.py` 文件中。`include` 函数有多种用法, 这里讲下两种常用的

用法。

1. `include(pattern,namespace=None)` : 直接把其他 app 的 urls 包含进来。示例代码如下:

```
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',include("book.urls"))
]
```

当然也可以传递 `namespace` 参数来指定一个实例命名空间,但是在使用实例命名空间之前,必须先指定一个应用命名空间。示例代码如下:

```
# 主urls.py文件:
from django.urls import path,include
urlpatterns = [
    path('movie/',include('movie.urls',namespace='movie'))
]
```

然后在 `movie/urls.py` 中指定应用命名空间。实例代码如下:

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

# 应用命名空间
app_name = 'movie'

urlpatterns = [
    path('',views.movie,name='index'),
    path('list/',views.movie_list,name='list'),
]
```

2. `include(pattern_list)` : 可以包含一个列表或者一个元组,这个元组或者列表中又包含的是 `path` 或者是 `re_path` 函数。
3. `include((pattern,app_namespace),namespace=None)` : 在包含某个 app 的 urls 的时候,可以指定命名空间,这样做的目的是为了防止不同的 app 下出现相同的 url,这时候就可以通过命名空间进行区分。示例代码如下:

```
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
```

```
path('book/',include(("book.urls",'book')),namespace='book')
]
```

但是这样做的前提是已经包含了应用命名空间。即在 `myapp.urls.py` 中添加一个和 `urlpatterns` 同级别的变量 `app_name` 。

指定默认的参数：

使用 `path` 或者是 `re_path` 的后，在 `route` 中都可以包含参数，而有时候想指定默认的参数，这时候可以通过以下方式来完成。示例代码如下：

```
from django.urls import path

from . import views

urlpatterns = [
    path('blog/', views.page),
    path('blog/page<int:num>/', views.page),
]

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

当在访问 `blog/` 的时候，因为没有传递 `num` 参数，所以会匹配到第一个url，这时候就执行 `view.page` 这个视图函数，而在 `page` 函数中，又有 `num=1` 这个默认参数。因此这时候就可以不用传递参数。而如果访问 `blog/1` 的时候，因为在传递参数的时候传递了 `num`，因此会匹配到第二个 `url`，这时候也会执行 `views.page`，然后把传递进来的参数传给 `page` 函数中的 `num`。

url反转：

之前我们都是通过url来访问视图函数。有时候我们知道这个视图函数，但是想反转回他的url。这时候就可以通过 `reverse` 来实现。示例代码如下：

```
reverse("list")
> /book/list/
```

如果有应用命名空间或者有实例命名空间，那么应该在反转的时候加上命名空间。示例代码如下：

```
reverse('book:list')
> /book/list/
```

如果这个url中需要传递参数，那么可以通过 `kwargs` 来传递参数。示例代码如下：

```
reverse("book:detail",kwargs={"book_id":1})
> /book/detail/1
```

因为 django 中的 reverse 反转 url 的时候不区分 GET 请求和 POST 请求，因此不能在反转的时候添加查询字符串的参数。如果想要添加查询字符串的参数，只能手动的添加。示例代码如下：

```
login_url = reverse('login') + "?next=/"
```

自定义URL转换器：

之前已经学到过一些django内置的 url 转换器，包括有 int 、 uuid 等。有时候这些内置的 url 转换器 并不能满足我们的需求，因此django给我们提供了一个接口可以让我们自己定义自己的url转换器。

自定义 url 转换器按照以下五个步骤来走就可以了：

1. 定义一个类。
2. 在类中定义一个属性 regex ，这个属性是用来保存 url 转换器规则的正则表达式。
3. 实现 to_python(self,value) 方法，这个方法是将 url 中的值转换一下，然后传给视图函数的。
4. 实现 to_url(self,value) 方法，这个方法是在做 url 反转的时候，将传进来的参数转换后拼接成一个正确的url。
5. 将定义好的转换器，注册到django中。

比如写一个匹配四个数字年份的 url 转换器。示例代码如下：

```
# 1. 定义一个类
class FourDigitYearConverter:
    # 2. 定义一个正则表达式
    regex = '[0-9]{4}'

    # 3. 定义to_python方法
    def to_python(self, value):
        return int(value)

    # 4. 定义to_url方法
    def to_url(self, value):
        return '%04d' % value

# 5. 注册到django中
from django.urls import register_converter
register_converter(converters.FourDigitYearConverter, 'yyyy')
urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    # 使用注册的转换器
```

```
    path('articles/<yyyy:year>/', views.year_archive),  
    ...  
]
```

模板

在之前的章节中，视图函数只是直接返回文本，而在实际生产环境中其实很少这样用，因为实际的页面大多是带有样式的HTML代码，这可以让浏览器渲染出非常漂亮的页面。目前市面上有非常多的模板系统，其中最知名最好用的就是DTL和Jinja2。DTL 是 Django Template Language 三个单词的缩写，也就是Django自带的模板语言。当然也可以配置Django支持Jinja2等其他模板引擎，但是作为Django内置的模板语言，和Django可以达到无缝衔接而不会产生一些不兼容的情况。因此建议大家学习好DTL。

DTL与普通的HTML文件的区别：

DTL模板是一种带有特殊语法的HTML文件，这个HTML文件可以被Django编译，可以传递参数进去，实现数据动态化。在编译完成后，生成一个普通的HTML文件，然后发送给客户端。

渲染模板：

渲染模板有多种方式。这里讲下两种常用的方式。

1. `render_to_string`：找到模板，然后将模板编译后渲染成Python的字符串格式。最后再通过 `HttpResponse` 类包装成一个 `HttpResponse` 对象返回回去。示例代码如下：

```
from django.template.loader import render_to_string
from django.http import HttpResponse
def book_detail(request,book_id):
    html = render_to_string("detail.html")
    return HttpResponse(html)
```

2. 以上方式虽然已经很方便了。但是django还提供了一个更加简便的方式，直接将模板渲染成字符串和包装成 `HttpResponse` 对象一步到位完成。示例代码如下：

```
from django.shortcuts import render
def book_list(request):
    return render(request,'list.html')
```

模板查找路径配置：

在项目的 `settings.py` 文件中。有一个 `TEMPLATES` 配置，这个配置包含了模板引擎的配置，模板查找路径的配置，模板上下文的配置等。模板路径可以在两个地方配置。

1. `DIRS`：这是一个列表，在这个列表中可以存放所有的模板路径，以后在视图中使用 `render` 或者 `render_to_string` 渲染模板的时候，会在这个列表的路径中查找模板。

2. `APP_DIRS` : 默认为 `True` , 这个设置为 `True` 后, 会在 `INSTALLED_APPS` 的安装了了的 `APP` 下的 `templates` 文件夹中查找模板。
3. 查找顺序: 比如代码 `render('list.html')` 。先会在 `DIRS` 这个列表中依次查找路径下有没有这个模板, 如果有, 就返回。如果 `DIRS` 列表中所有的路径都没有找到, 那么会先检查当前这个视图所处的 `app` 是否已经安装, 如果已经安装了, 那么就先在当前这个 `app` 下的 `templates` 文件夹中查找模板, 如果没有找到, 那么会在其他已经安装了的 `app` 中查找。如果所有路径下都没有找到, 那么会抛出一个 `TemplateDoesNotExist` 的异常。

DTL模板语法

变量：

模板中可以包含变量，`Django` 在渲染模板的时候，可以传递变量对应的值过去进行替换。变量的命名规范和 `Python` 非常类似，只能是阿拉伯数字和英文字符以及下划线的组合，不能出现标点符号等特殊字符。变量需要通过视图函数渲染，视图函数在使用 `render` 或者 `render_to_string` 的时候可以传递一个 `context` 的参数，这个参数是一个字典类型。以后在模板中的变量就从这个字典中读取值的。示例代码如下：

```
# profile.html模板代码
<p>{{ username }}</p>

# views.py代码
def profile(request):
    return render(request, 'profile.html', context={'username': 'huangyong'})
```

模板中的变量同样也支持点(.)的形式。在出现了点的情况，比如 `person.username`，模板是按照以下方式进行解析的：

1. 如果 `person` 是一个字典，那么就会查找这个字典的 `username` 这个 `key` 对应的值。
2. 如果 `person` 是一个对象，那么就会查找这个对象的 `username` 属性，或者是 `username` 这个方法。
3. 如果出现的是 `person.1`，会判断 `persons` 是否是一个列表或者元组或者任意的可以通过下标访问的对象，如果是的话就取这个列表的第1个值。如果不是就获取到的是一个空的字符串。

不能通过中括号的形式访问字典和列表中的值，比如`dict['key']`和`list[1]`是不支持的！

因为使用点(.)语法获取对象值的时候，可以获取这个对象的属性，如果这个对象是一个字典，也可以获取这个字典的值。所以在给这个字典添加`key`的时候，千万不能和字典中的一些属性重复。比如`items`，`items`是字典的方法，那么如果给这个字典添加一个`items`作为`key`，那么以后就不能再通过`item`来访问这个字典的键值对了。

常用的模板标签：

1. **if 标签**： **if** 标签相当于 Python 中的 **if** 语句，有 **elif** 和 **else** 相对应，但是所有的标签都需要用标签符号（ **{% %}** ）进行包裹。 **if** 标签中可以使用 **==**、**!=**、**<**、**<=**、**>**、**>=**、**in**、**not in**、**is**、**is not** 等判断运算符。示例代码如下：

```
{% if "张三" in persons %}
    <p>张三</p>
{% else %}
    <p>李四</p>
{% endif %}
```

2. **for...in... 标签**： **for...in...** 类似于 Python 中的 **for...in...** 。可以遍历列表、元组、字符串、字典等一切可以遍历的对象。示例代码如下：

```
{% for person in persons %}
    <p>{{ person.name }}</p>
{% endfor %}
```

如果想要反向遍历，那么在遍历的时候就加上一个 **reversed** 。示例代码如下：

```
{% for person in persons reversed %}
    <p>{{ person.name }}</p>
{% endfor %}
```

遍历字典的时候，需要使用 **items** 、 **keys** 和 **values** 等方法。在 **DTL** 中，执行一个方法不能使用圆括号的形式。遍历字典示例代码如下：

```
{% for key,value in person.items %}
    <p>key: {{ key }}</p>
    <p>value: {{ value }}</p>
{% endfor %}
```

在 **for** 循环中， **DTL** 提供了一些变量可供使用。这些变量如下：

- **forloop.counter** ：当前循环的下标。以1作为起始值。
- **forloop.counter0** ：当前循环的下标。以0作为起始值。
- **forloop.revcounter** ：当前循环的反向下标值。比如列表有5个元素，那么第一次遍历这个属性是等于5，第二次是4，以此类推。并且是以1作为最后一个元素的下标。
- **forloop.revcounter0** ：类似于**forloop.revcounter**。不同的是最后一个元素的下标是从0开始。
- **forloop.first** ：是否是第一次遍历。

- `forloop.last` : 是否是最后一次遍历。
 - `forloop.parentloop` : 如果有多个循环嵌套, 那么这个属性代表的是上一级的for循环。
3. `for...in...empty` 标签: 这个标签使用跟 `for...in...` 是一样的, 只不过是在遍历的对象如果没有元素的情况下, 会执行 `empty` 中的内容。示例代码如下:

```
{% for person in persons %}
    <li>{{ person }}</li>
{% empty %}
    暂时还没有任何人
{% endfor %}
```

4. `with` 标签: 在模版中定义变量。有时候一个变量访问的时候比较复杂, 那么可以先把这个复杂的变量缓存到一个变量上, 以后就可以直接使用这个变量就可以了。示例代码如下:

```
context = {
    "persons": ["张三", "李四"]
}

{% with lisi=persons.1 %}
    <p>{{ lisi }}</p>
{% endwith %}
```

有几点需要强烈的注意:

- 在 `with` 语句中定义的变量, 只能在 `{%with%}{%endwith%}` 中使用, 不能在这个标签外面使用。
- 定义变量的时候, 不能在等号左右两边留有空格。比如 `{% with lisi = persons.1%}` 是错误的。
- 还有另外一种写法同样也是支持的:

```
{% with persons.1 as lisi %}
    <p>{{ lisi }}</p>
{% endwith %}
```

5. `url` 标签: 在模版中, 我们经常要写一些 `url`, 比如某个 `a` 标签中需要定义 `href` 属性。当然如果通过硬编码的方式直接将这个 `url` 写死在里面也是可以的。但是这样对于以后项目维护可能不是一件好事。因此建议使用这种反转的方式来实现, 类似于 `django` 中的 `reverse` 一样。示例代码如下:

```
<a href="{% url 'book:list' %}">图书列表页面</a>
```

如果 `url` 反转的时候需要传递参数, 那么可以在后面传递。但是参数分位置参数和关键字参数。位置参数和关键字参数不能同时使用。示例代码如下:

```
# path部分
path('detail/<book_id>/',views.book_detail,name='detail')

# url反转, 使用位置参数
<a href="{% url 'book:detail' 1 %}">图书详情页面</a>

# url反转, 使用关键字参数
<a href="{% url 'book:detail' book_id=1 %}">图书详情页面</a>
```

如果想要在使用 `url` 标签反转的时候要传递查询字符串的参数, 那么必须要手动在后面添加。示例代码如下:

```
<a href="{% url 'book:detail' book_id=1 %}?page=1">图书详情页面</a>
```

如果需要传递多个参数, 那么通过空格的方式进行分隔。示例代码如下:

```
<a href="{% url 'book:detail' book_id=1 page=2 %}">图书详情页面</a>
```

6. `spaceless` 标签: 移除html标签中的空白字符。包括空格、`tab`键、换行等。示例代码如下:

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

那么在渲染完成后, 会变成以下的代码:

```
<p><a href="foo/">Foo</a></p>
```

`spaceless` 只会移除html标签之间的空白字符。而不会移除标签与文本之间的空白字符。看以下代码:

```
{% spaceless %}
  <strong>
    Hello
  </strong>
{% endspaceless %}
```

这个将不会移除 `strong` 中的空白字符。

7. `autoescape` 标签：开启和关闭这个标签内元素的自动转义功能。自动转义是可以将一些特殊的字符。比如 `<` 转义成 `<`，`>` 会被自动转义成 `>`。模板中默认是已经开启了自动转义的。`autoescape` 的示例代码如下：

```
# 传递的上下文信息
context = {
    "info": "<a href='www.baidu.com'>百度</a>"
}

# 模板中关闭自动转义
{% autoescape off %}
    {{ info }}
{% endautoescape %}
```

那么就会显示百度的一个超链接。如果把 `off` 改成 `on`，那么就会显示成一个普通的字符串。示例代码如下：

```
{% autoescape on %}
    {{ info }}
{% endautoescape %}
```

8. `verbatim` 标签：默认在 DTL 模板中是会去解析那些特殊字符的。比如 `{%` 和 `%}` 以及 `{{` 等。如果你在某个代码片段中不想使用 DTL 的解析引擎。那么你可以把这个代码片段放在 `verbatim` 标签中。示例代码如下：

```
{% verbatim %}
    {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

9. 更多标签请参考官方文

档：<https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>

模版常用过滤器

在模版中，有时候需要对一些数据进行处理以后才能使用。一般在 `Python` 中我们是通过函数的形式来完成的。而在模版中，则是通过过滤器来实现的。过滤器使用的是 `|` 来使用。比如使用 `add` 过滤器，那么示例代码如下：

```
{{ value|add:"2" }}
```

那么下面就讲下在开发中常用的过滤器。

add

将传进来的参数添加到原来的值上面。这个过滤器会尝试将 `值` 和 `参数` 转换成整形然后进行相加。如果转换成整形过程中失败了，那么会将 `值` 和 `参数` 进行拼接。如果是字符串，那么会拼接成字符串，如果是列表，那么会拼接成一个列表。示例代码如下：

```
{{ value|add:"2" }}
```

如果 `value` 是等于4，那么结果将是6。如果 `value` 是等于一个普通的字符串，比如 `abc`，那么结果将是 `abc2`。 `add` 过滤器的源代码如下：

```
def add(value, arg):
    """Add the arg to the value."""
    try:
        return int(value) + int(arg)
    except (ValueError, TypeError):
        try:
            return value + arg
        except Exception:
            return ''
```

cut

移除值中所有指定的字符串。类似于 `python` 中的 `replace(args, "")`。示例代码如下：

```
{{ value|cut:" " }}
```

以上示例将会移除 `value` 中所有的空格字符。 `cut` 过滤器的源代码如下：

```
def cut(value, arg):
    """Remove all values of arg from the given string."""
```

```

safe = isinstance(value, SafeData)
value = value.replace(arg, '')
if safe and arg != ';':
    return mark_safe(value)
return value

```

date

将一个日期按照指定的格式，格式化成字符串。示例代码如下：

```

# 数据
context = {
    "birthday": datetime.now()
}

# 模版
{{ birthday|date:"Y/m/d" }}

```

那么将会输出 `2018/02/01`。其中 `Y` 代表的是四位数字的年份，`m` 代表的是两位数字的月份，`d` 代表的是两位数字的日。

还有更多时间格式化的方式。见下表。

格式字符	描述	示例
Y	四位数字的年份	2018
m	两位数字的月份	01-12
n	月份，1-9前面没有0前缀	1-12
d	两位数字的天	01-31
j	天，但是1-9前面没有0前缀	1-31
g	小时，12小时格式的，1-9前面没有0前缀	1-12
h	小时，12小时格式的，1-9前面有0前缀	01-12
G	小时，24小时格式的，1-9前面没有0前缀	1-23
H	小时，24小时格式的，1-9前面有0前缀	01-23
i	分钟，1-9前面有0前缀	00-59
s	秒，1-9前面有0前缀	00-59

default

如果值被评估为 `False`。比如 `[]`，`""`，`None`，`{}` 等这些在 `if` 判断中为 `False` 的值，都会使用 `default` 过滤器提供的默认值。示例代码如下：

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

如果 `value` 是等于一个空的字符串。比如 `""`，那么以上代码将会输出 `nothing`。

default_if_none

如果值是 `None`，那么将会使用 `default_if_none` 提供的默认值。这个和 `default` 有区别，`default` 是所有被评估为 `False` 的都会使用默认值。而 `default_if_none` 则只有这个值是等于 `None` 的时候才会使用默认值。示例代码如下：

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

如果 `value` 是等于 `""` 也即空字符串，那么以上会输出空字符串。如果 `value` 是一个 `None` 值，以上代码才会输出 `nothing`。

first

返回列表/元组/字符串中的第一个元素。示例代码如下：

```
{{ value|first }}
```

如果 `value` 是等于 `['a','b','c']`，那么输出将会是 `a`。

last

返回列表/元组/字符串中的最后一个元素。示例代码如下：

```
{{ value|last }}
```

如果 `value` 是等于 `['a','b','c']`，那么输出将会是 `c`。

floatformat

使用四舍五入的方式格式化一个浮点类型。如果这个过滤器没有传递任何参数。那么只会在小数点后保留一个小数，如果小数后面全是0，那么只会保留整数。当然也可以传递一个参数，标识具体要保留几个小数。

1. 如果没有传递参数：

value	模版代码	输出	---	---	---	34.23234	{{ value\ floatformat }}	34.2		
34.000						{{ value\ floatformat }}	34	34.260	{{ value\ floatformat }}	34.3

2. 如果传递参数：

value	模版代码	输出
34.23234	<code>{{value\ floatformat:3}}</code>	34.232
34.0000	<code>{{value\ floatformat:3}}</code>	34.000
34.26000	<code>{{value\ floatformat:3}}</code>	34.260

join

类似与 Python 中的 join ，将列表/元组/字符串用指定的字符进行拼接。示例代码如下：

```
{{ value|join:"/" }}
```

如果 value 是等于 ['a','b','c'] ，那么以上代码将输出 a/b/c 。

length

获取一个列表/元组/字符串/字典的长度。示例代码如下：

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

如果 value 是等于 ['a','b','c'] ，那么以上代码将输出 3 。如果 value 为 None ，那么以上将返回 0 。

lower

将值中所有的字符全部转换成小写。示例代码如下：

```
{{ value|lower }}
```

如果 value 是等于 Hello World 。那么以上代码将输出 hello world 。

upper

类似于 lower ，只不过是將指定的字符串全部转换成大写。

random

在被给的列表/字符串/元组中随机的选择一个值。示例代码如下：

```
{{ value|random }}
```

如果 value 是等于 ['a','b','c'] ，那么以上代码会在列表中随机选择一个。

safe

标记一个字符串是安全的。也即会关掉这个字符串的自动转义。示例代码如下：

```
{{value|safe}}
```

如果 `value` 是一个不包含任何特殊字符的字符串，比如 `<a>` 这种，那么以上代码就会把字符串正常的输入。如果 `value` 是一串 `html` 代码，那么以上代码将会把这个 `html` 代码渲染到浏览器中。

slice

类似于 `Python` 中的切片操作。示例代码如下：

```
{{ some_list|slice:"2:" }}
```

以上代码将会给 `some_list` 从 2 开始做切片操作。

stringtags

删除字符串中所有的 `html` 标签。示例代码如下：

```
{{ value|striptags }}
```

如果 `value` 是 `hello world`，那么以上代码将会输出 `hello world`。

truncatechars

如果给定的字符串长度超过了过滤器指定的长度。那么就会进行切割，并且会拼接三个点来作为省略号。示例代码如下：

```
{{ value|truncatechars:5 }}
```

如果 `value` 是等于 `北京欢迎您~`，那么输出的结果是 `北京...`。可能你会想，为什么不会 `北京欢迎您...` 呢。因为三个点也占了三个字符，所以 `北京` + 三个点的字符长度就是5。

truncatechars_html

类似于 `truncatechars`，只不过是不会切割 `html` 标签。示例代码如下：

```
{{ value|truncatechars:5 }}
```

如果 `value` 是等于 `<p>北京欢迎您~</p>`，那么输出将是 `<p>北京...</p>`。

自定义模版过滤器

虽然 DTL 给我们内置了许多好用的过滤器。但是有些时候还是不能满足我们的需求。因此 Django 给我们提供了一个接口，可以让我们自定义过滤器，实现自己的需求。

模版过滤器必须要放在 app 中，并且这个 app 必须要在 INSTALLED_APPS 中进行安装。然后再在这个 app 下面创建一个 Python 包 叫做 templatetags 。再在这个包下面创建一个 python 文件 。比如 app 的名字叫做 book ，那么项目结构如下：

```
- book
  - views.py
  - urls.py
  - models.py
  - templatetags
    - my_filter.py
```

在创建了存储过滤器的文件后，接下来就是在这个文件中写过滤器了。过滤器实际上就是python中的一个函数，只不过是把这个函数注册到模板库中，以后在模板中就可以使用这个函数了。但是这个函数的参数有限制，第一个参数必须是这个过滤器需要处理的值，第二个参数可有可无，如果有，那么就意味着在模板中可以传递参数。并且过滤器的函数最多只能有两个参数。在写完过滤器后，再使用 django.template.Library 对象注册进去。示例代码如下：

```
from django import template

# 创建模板库对象
register = template.Library()

# 过滤器函数
def mycut(value,mystr):
    return value.replace(mystr)

# 将函数注册到模板库中
register.filter("mycut",mycut)
```

以后想要在模板中使用这个过滤器，就要在模板中 load 一下这个过滤器所在的模块的名字（也就是这个python文件的名字）。示例代码如下：

```
{% load my_filter %}
```

自定义时间计算过滤器：

有时候经常会在朋友圈、微博中可以看到一条信息发表的时间，并不是具体的时间，而是距离现在多久。比如 刚刚，1分钟前 等。这个功能 DTL 是没有内置这样的过滤器的，因此我们可以自定义一个这样的过滤器。示例代码如下：

```
# time_filter.py文件

from datetime import datetime
from django import template

register = template.Library()

def time_since(value):
    """
    time距离现在的时间间隔
    1. 如果时间间隔小于1分钟以内，那么就显示“刚刚”
    2. 如果是大于1分钟小于1小时，那么就显示“xx分钟前”
    3. 如果是大于1小时小于24小时，那么就显示“xx小时前”
    4. 如果是大于24小时小于30天以内，那么就显示“xx天前”
    5. 否则就是显示具体的时间 2017/10/20 16:15
    """
    if isinstance(value,datetime):
        now = datetime.now()
        timestamp = (now - value).total_seconds()
        if timestamp < 60:
            return "刚刚"
        elif timestamp >= 60 and timestamp < 60*60:
            minutes = int(timestamp / 60)
            return "%s分钟前" % minutes
        elif timestamp >= 60*60 and timestamp < 60*60*24:
            hours = int(timestamp / (60*60))
            return "%s小时前" % hours
        elif timestamp >= 60*60*24 and timestamp < 60*60*24*30:
            days = int(timestamp / (60*60*24))
            return "%s天前" % days
        else:
            return value.strftime("%Y/%m/%d %H:%M")
    else:
        return value

register.filter("time_since",time_since)
```

在模版中使用的示例代码如下：

```
{% load time_filter %}
...
{% value|time_since %}
```

...

为了更加方便的将函数注册到模版库中当作过滤器。也可以使用装饰器来将一个函数包装成过滤器。示例代码如下：

```
from django import template
register = template.Library()

@register.filter(name='mycut')
def mycut(value, mystr):
    return value.replace(mystr, "")
```

模版结构优化

引入模版

有时候一些代码是在许多模版中都用到的。如果我们每次都重复的去拷贝代码那肯定不符合项目的规范。一般我们可以把这些重复性的代码抽取出来，就类似于Python中的函数一样，以后想要使用这些代码的时候，就通过 `include` 包含进来。这个标签就是 `include` 。示例代码如下：

```
# header.html
<p>我是header</p>

# footer.html
<p>我是footer</p>

# main.html
{% include 'header.html' %}
<p>我是main内容</p>
{% include 'footer.html' %}
```

`include` 标签寻找路径的方式。也是跟 `render` 渲染模板的函数是一样的。

默认 `include` 标签包含模版，会自动的使用主模版中的上下文，也即可以自动的使用主模版中的变量。如果想传入一些其他的参数，那么可以使用 `with` 语句。示例代码如下：

```
# header.html
<p>用户名: {{ username }}</p>

# main.html
{% include "header.html" with username='huangyong' %}
```

模板继承：

在前端页面开发中。有些代码是需要重复使用的。这种情况可以使用 `include` 标签来实现。也可以使用另外一个比较强大的方式来实现，那就是模版继承。模版继承类似于 Python 中的类，在父类中可以先定义好一些变量和方法，然后在子类中实现。模版继承也可以在父模版中先定义好一些子模版需要用到的代码，然后子模版直接继承就可以了。并且因为子模版肯定有自己的不同代码，因此可以在父模版中定义一个**block**接口，然后子模版再去实现。以下是父模版的代码：

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
```

```

<link rel="stylesheet" href="{% static 'style.css' %}" />
<title>{% block title %}我的站点{% endblock %}</title>
</head>

<body>
  <div id="sidebar">
    {% block sidebar %}
    <ul>
      <li><a href="/">首页</a></li>
      <li><a href="/blog/">博客</a></li>
    </ul>
    {% endblock %}
  </div>
  <div id="content">
    {% block content %}{% endblock %}
  </div>
</body>
</html>

```

这个模版，我们取名叫做 `base.html`，定义好一个简单的 `html` 骨架，然后定义好两个 `block` 接口，让子模版来根据具体需求来实现。子模板然后通过 `extends` 标签来实现，示例代码如下：

```

{% extends "base.html" %}

{% block title %}博客列表{% endblock %}

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

```

需要注意的是：**extends** 标签必须放在模版的第一行。

子模板中的代码必须放在 **block** 中，否则将不会被渲染。

如果在某个 `block` 中需要使用父模版的内容，那么可以使用 `{{block.super}}` 来继承。比如上例，`{%block title%}`，如果想要使用父模版的 `title`，那么可以在子模版的 `title block` 中使用 `{{ block.super }}` 来实现。

在定义 `block` 的时候，除了在 `block` 开始的地方定义这个 `block` 的名字，还可以在 `block` 结束的时候定义名字。比如 `{% block title %}{% endblock title %}`。这在大型模版中显得尤其有用，能让你快速的看到 `block` 包含在哪里。

加载静态文件

在一个网页中，不仅仅只有一个 `html` 骨架，还需要 `css` 样式文件，`js` 执行文件以及一些图片等。因此在 `DTL` 中加载静态文件是一个必须要解决的问题。在 `DTL` 中，使用 `static` 标签来加载静态文件。要使用 `static` 标签，首先需要 `{% load static %}`。加载静态文件的步骤如下：

1. 首先确保 `django.contrib.staticfiles` 已经添加到 `settings.INSTALLED_APPS` 中。
2. 确保在 `settings.py` 中设置了 `STATIC_URL`。
3. 在已经安装了的 `app` 下创建一个文件夹叫做 `static`，然后再在这个 `static` 文件夹下创建一个当前 `app` 的名字的文件夹，再把静态文件放到这个文件夹下。例如你的 `app` 叫做 `book`，有一个静态文件叫做 `zhiliao.jpg`，那么路径为 `book/static/book/zhiliao.jpg`。（为什么在 `app` 下创建一个 `static` 文件夹，还需要在这个 `static` 下创建一个同 `app` 名字的文件夹呢？原因是如果直接把静态文件放在 `static` 文件夹下，那么在模版加载静态文件的时候就是使用 `zhiliao.jpg`，如果在多个 `app` 之间有同名的静态文件，这时候可能就会产生混淆。而在 `static` 文件夹下加了一个同名 `app` 文件夹，在模版中加载的时候就是使用 `app/zhiliao.jpg`，这样就可以避免产生混淆。）
4. 如果有一些静态文件是不和任何 `app` 挂钩的。那么可以在 `settings.py` 中添加 `STATICFILES_DIRS`，以后 `DTL` 就会在这个列表的路径中查找静态文件。比如可以设置为：

```
STATICFILES_DIRS = [  
    os.path.join(BASE_DIR, "static")  
]
```

5. 在模版中使用 `load` 标签加载 `static` 标签。比如要加载在项目的 `static` 文件夹下的 `style.css` 的文件。那么示例代码如下：

```
{% load static %}  
<link rel="stylesheet" href="{% static 'style.css' %}">
```

6. 如果不想每次在模版中加载静态文件都使用 `load` 加载 `static` 标签，那么可以在 `settings.py` 中的 `TEMPLATES/OPTIONS` 添加 `'builtins':` `['django.template.tags.static']`，这样以后在模版中就可以直接使用 `static` 标签，而不用手动的 `load` 了。
7. 如果没有在 `settings.INSTALLED_APPS` 中添加 `django.contrib.staticfiles`。那么我们就需要手动的将请求静态文件的 `url` 与静态文件的路径进行映射了。示例代码如下：

```
from django.conf import settings  
from django.conf.urls.static import static  
  
urlpatterns = [
```

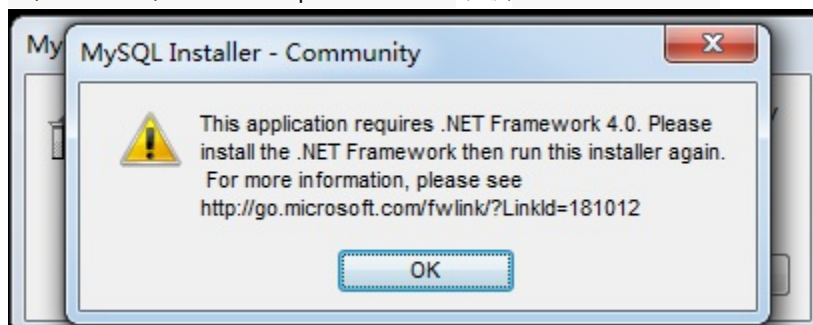
```
# 其他的url映射  
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

MySQL数据库

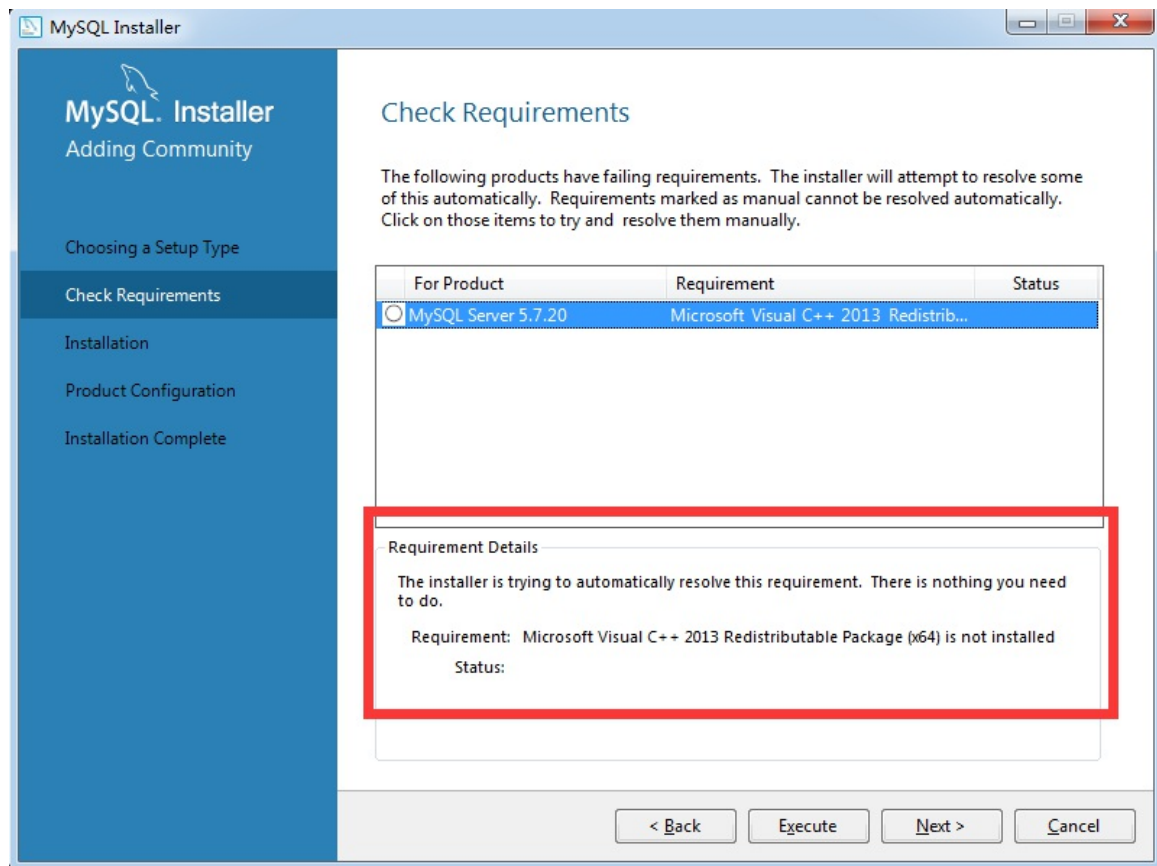
在网站开发中，数据库是网站的重要组成部分。只有提供数据库，数据才能够动态的展示，而不是在网页中显示一个静态的页面。数据库有很多，比如有 SQL Server 、 Oracle 、 PostgreSQL 以及 MySQL 等等。MySQL 由于价格实惠、简单易用、不受平台限制、灵活度高等特性，目前已经取得了绝大多数的市场份额。因此我们在 Django 中，也是使用 MySQL 来作为数据存储。

MySQL数据库安装：

1. 在 MySQL 的官网下载 MySQL 数据库安装文件：<https://dev.mysql.com/downloads/windows/installer/5.7.html> 。
2. 然后双击安装，如果出现以下错误，则到 <http://www.microsoft.com/en-us/download/details.aspx?id=17113> 下载 .net framework 。



3. 在安装过程中，如果提示没有 Microsoft C++ 2013 ，那么就到以下网址下载安装即可：http://download.microsoft.com/download/9/0/5/905DBD86-D1B8-4D4B-8A50-CB0E922017B9/vcredist_x64.exe 。



4. 接下来就是做好用户名和密码的配置即可。

navicat数据库操作软件：

安装完 MySQL 数据库以后，就可以使用 MySQL 提供的终端客户端软件来操作数据库。如下：

```
MySQL 5.7 Command Line Client
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.18-log MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

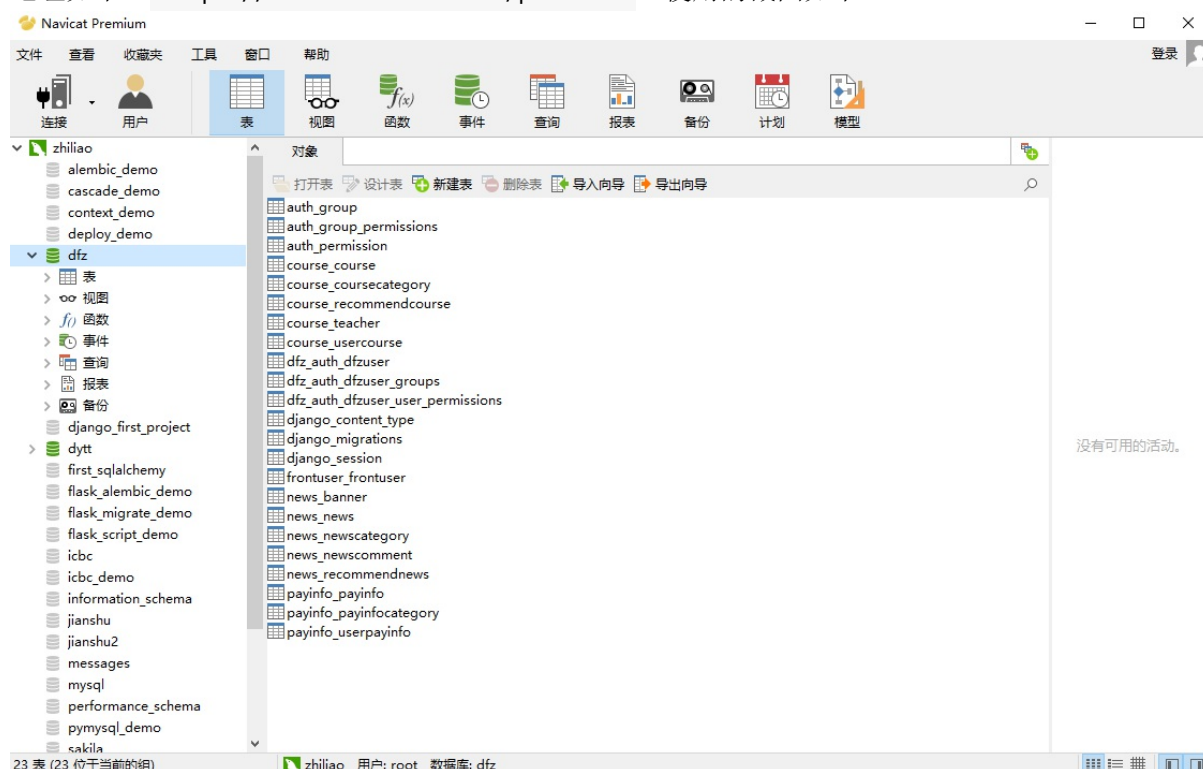
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

这个软件所有的操作都是基于 sql 语言，对于想要熟练 sql 语言的同学来讲是非常合适的。但是对于在企业中可能不是一款好用的工具。在企业中我们推荐使用 mysql workbench 以

及 navicat 这种图形化操作的软件。而 mysql workbench 是 mysql 官方提供的一个免费的软件，正因为是免费，所以在一些功能上不及 navicat 。 navicat for mysql 是一款收费的软件。官网地址如下：<https://www.navicat.com.cn/products> 。使用的截图如下：



MySQL驱动程序安装：

我们使用 Django 来操作 MySQL ，实际上底层还是通过 Python 来操作的。因此我们想要用 Django 来操作 MySQL ，首先还是需要安装一个驱动程序。在 Python3 中，驱动程序有多种选择。比如有 pymysql 以及 mysqlclient 等。这里我们就使用 mysqlclient 来操作。mysqlclient 安装非常简单。只需要通过 `pip install mysqlclient` 即可安装。

常见 MySQL 驱动介绍：

1. MySQL-python ：也就是 MySQLdb 。是对 C 语言操作 MySQL 数据库的一个简单封装。遵循了 Python DB API v2 。但是只支持 Python2 ，目前还不支持 Python3 。
2. mysqlclient ：是 MySQL-python 的另外一个分支。支持 Python3 并且修复了一些 bug 。
3. pymysql ：纯 Python 实现的一个驱动。因为是纯 Python 编写的，因此执行效率不如 MySQL-python 。并且也因为是纯 Python 编写的，因此可以和 Python 代码无缝衔接。
4. MySQL Connector/Python ：MySQL 官方推出的使用纯 Python 连接 MySQL 的驱动。因为是纯 Python 开发的。效率不高。

操作数据库

Django配置连接数据库：

在操作数据库之前，首先先要连接数据库。这里我们以配置 MySQL 为例来讲解。Django 连接数据库，不需要单独的创建一个连接对象。只需要在 `settings.py` 文件中做好数据库相关的配置就可以了。示例代码如下：

```
DATABASES = {
    'default': {
        # 数据库引擎（是mysql还是oracle等）
        'ENGINE': 'django.db.backends.mysql',
        # 数据库的名字
        'NAME': 'dfz',
        # 连接mysql数据库的用户名
        'USER': 'root',
        # 连接mysql数据库的密码
        'PASSWORD': 'root',
        # mysql数据库的主机地址
        'HOST': '127.0.0.1',
        # mysql数据库的端口号
        'PORT': '3306',
    }
}
```

在Django中操作数据库：

在 Django 中操作数据库有两种方式。第一种方式就是使用原生 `sql` 语句操作，第二种就是使用 `ORM` 模型来操作。这节课首先来讲下第一种。

在 Django 中使用原生 `sql` 语句操作其实就是使用 `python db api` 的接口来操作。如果你的 `mysql` 驱动使用的是 `pymysql`，那么你就是使用 `pymysql` 来操作的，只不过 Django 将数据库连接的这一部分封装好了，我们只要在 `settings.py` 中配置好了数据库连接信息后直接使用 Django 封装好的接口就可以操作了。示例代码如下：

```
# 使用django封装好的connection对象，会自动读取settings.py中数据库的配置信息
from django.db import connection

# 获取游标对象
cursor = connection.cursor()
# 拿到游标对象后执行sql语句
cursor.execute("select * from book")
# 获取所有的数据
```

```
rows = cursor.fetchall()
# 遍历查询到的数据
for row in rows:
    print(row)
```

以上的 `execute` 以及 `fetchall` 方法都是 Python DB API 规范中定义好的。任何使用 Python 来操作 MySQL 的驱动程序都应该遵循这个规范。所以不管是使用 `pymysql` 或者是 `mysqlclient` 或者是 `mysqldb`，他们的接口都是一样的。更多规范请参考：<https://www.python.org/dev/peps/pep-0249/>。

Python DB API下规范下cursor对象常用接口：

1. `description`：如果 `cursor` 执行了查询的 `sql` 代码。那么读取 `cursor.description` 属性的时候，将返回一个列表，这个列表中装的是元组，元组中装的分别是 `(name,type_code,display_size,internal_size,precision,scale,null_ok)`，其中 `name` 代表的是查找出来的数据的字段名称，其他参数暂时用处不大。
2. `rowcount`：代表的是在执行了 `sql` 语句后受影响的行数。
3. `close`：关闭游标。关闭游标以后就再也不能使用了，否则会抛出异常。
4. `execute(sql[,parameters])`：执行某个 `sql` 语句。如果在执行 `sql` 语句的时候还需要传递参数，那么可以传给 `parameters` 参数。示例代码如下：

```
cursor.execute("select * from article where id=%s",(1,))
```

5. `fetchone`：在执行了查询操作以后，获取第一条数据。
6. `fetchmany(size)`：在执行查询操作以后，获取多条数据。具体是多少条要看传的 `size` 参数。如果不传 `size` 参数，那么默认是获取第一条数据。
7. `fetchall`：获取所有满足 `sql` 语句的数据。

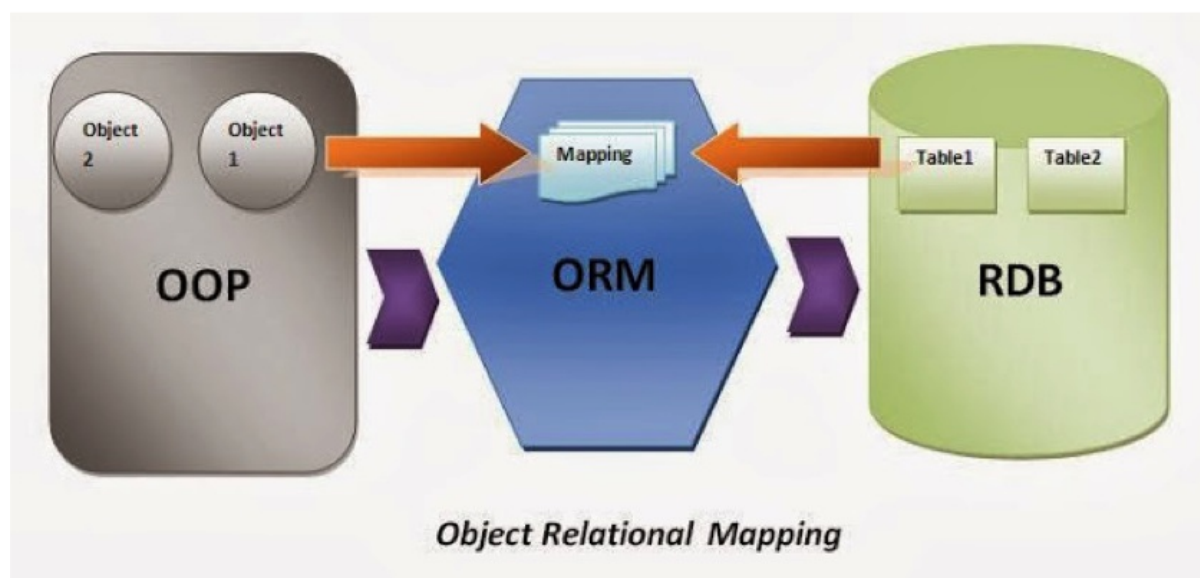
ORM模型介绍

随着项目越来越大，采用写原生SQL的方式在代码中会出现大量的SQL语句，那么问题就出现了：

1. SQL语句重复利用率不高，越复杂的SQL语句条件越多，代码越长。会出现很多相近的SQL语句。
2. 很多SQL语句是在业务逻辑中拼出来的，如果有数据库需要更改，就要去修改这些逻辑，这会很容易漏掉对某些SQL语句的修改。
3. 写SQL时容易忽略web安全问题，给未来造成隐患。SQL注入。

ORM，全称 Object Relational Mapping，中文叫做对象关系映射，通过 ORM 我们可以通过类的方式去操作数据库，而不用再写原生的SQL语句。通过把表映射成类，把行作实例，把字段作为属性，ORM 在执行对象操作的时候最终还是会把对应的操作转换为数据库原生语句。使用 ORM 有许多优点：

1. 易用性：使用 ORM 做数据库的开发可以有效的减少重复SQL语句的概率，写出来的模型也更加直观、清晰。
2. 性能损耗小：ORM 转换成底层数据库操作指令确实会有一些开销。但从实际的情况来看，这种性能损耗很少（不足5%），只要不是对性能有严苛的要求，综合考虑开发效率、代码的阅读性，带来的好处要远远大于性能损耗，而且项目越大作用越明显。
3. 设计灵活：可以轻松写出复杂的查询。
4. 可移植性：Django 封装了底层的数据库实现，支持多个关系数据库引擎，包括流行的 MySQL、PostgreSQL 和 SQLite。可以非常轻松的切换数据库。



创建ORM模型：

ORM 模型一般都是放在 `app` 的 `models.py` 文件中。每个 `app` 都可以拥有自己的模型。并且如果这个模型想要映射到数据库中，那么这个 `app` 必须要放在 `settings.py` 的 `INSTALLED_APP` 中进行安装。以下是写一个简单的书籍 ORM 模型。示例代码如下：

```
from django.db import models
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    author = models.CharField(max_length=20,null=False)
    pub_time = models.DateTimeField(default=datetime.now)
    price = models.FloatField(default=0)
```

以上便定义了一个模型。这个模型继承自 `django.db.models.Model`，如果这个模型想要映射到数据库中，就必须继承自这个类。这个模型以后映射到数据库中，表名是模型名称的小写形式，为 `book`。在这个表中，有四个字段，一个为 `name`，这个字段是保存的是书的名称，是 `varchar` 类型，最长不能超过20个字符，并且不能为空。第二个字段是作者名字类型，同样也是 `varchar` 类型，长度不能超过20个。第三个是出版时间，数据类型是 `datetime` 类型，默认是保存这本书籍的时间。第五个是这本书的价格，是浮点类型。还有一个字段我们没有写，就是主键 `id`，在 `django` 中，如果一个模型没有定义主键，那么将会自动生成一个自动增长的 `int` 类型的主键，并且这个主键的名字就叫做 `id`。

映射模型到数据库中：

将 ORM 模型映射到数据库中，总结起来就是以下几步：

1. 在 `settings.py` 中，配置好 `DATABASES`，做好数据库相关的配置。
2. 在 `app` 中的 `models.py` 中定义好模型，这个模型必须继承自 `django.db.models`。
3. 将这个 `app` 添加到 `settings.py` 的 `INSTALLED_APP` 中。
4. 在命令行终端，进入到项目所在的路径，然后执行命令 `python manage.py makemigrations` 来生成迁移脚本文件。
5. 同样在命令行中，执行命令 `python manage.py migrate` 来将迁移脚本文件映射到数据库中。

模型常用属性

常用字段：

在 Django 中，定义了一些 `Field` 来与数据库表中的字段类型来进行映射。以下将介绍那些常用的字段类型。

AutoField:

映射到数据库中是 `int` 类型，可以有自动增长的特性。一般不需要使用这个类型，如果不指定主键，那么模型会自动的生成一个叫做 `id` 的自动增长的主键。如果你想指定一个其他名字的并且具有自动增长的主键，使用 `AutoField` 也是可以的。

BigAutoField:

64位的整形，类似于 `AutoField`，只不过是产生的数据的范围是从 `1-9223372036854775807`。

BooleanField:

在模型层面接收的是 `True/False`。在数据库层面是 `tinyint` 类型。如果没有指定默认值，默认值是 `None`。

CharField:

在数据库层面是 `varchar` 类型。在 Python 层面就是普通的字符串。这个类型在使用的时候必须要指定最大的长度，也即必须要传递 `max_length` 这个关键字参数进去。

DateField:

日期类型。在 Python 中是 `datetime.date` 类型，可以记录年月日。在映射到数据库中也是 `date` 类型。使用这个 `Field` 可以传递以下几个参数：

1. `auto_now`：在每次这个数据保存的时候，都使用当前的时间。比如作为一个记录修改日期的字段，可以将这个属性设置为 `True`。
2. `auto_now_add`：在每次数据第一次被添加进去的时候，都使用当前的时间。比如作为一个记录第一次入库的字段，可以将这个属性设置为 `True`。

DateTimeField:

日期时间类型，类似于 `DateField`。不仅仅可以存储日期，还可以存储时间。映射到数据库中是 `datetime` 类型。这个 `Field` 也可以使用 `auto_now` 和 `auto_now_add` 两个属性。

TimeField:

时间类型。在数据库中是 `time` 类型。在 `Python` 中是 `datetime.time` 类型。

EmailField:

类似于 `CharField`。在数据库底层也是一个 `varchar` 类型。最大长度是254个字符。

FileField:

用来存储文件的。这个请参考后面的文件上传章节部分。

ImageField:

用来存储图片文件的。这个请参考后面的图片上传章节部分。

FloatField:

浮点类型。映射到数据库中是 `float` 类型。

IntegerField:

整形。值的区间是 `-2147483648—2147483647`。

BigIntegerField:

大整形。值的区间是 `-9223372036854775808—9223372036854775807`。

PositiveIntegerField:

正整形。值的区间是 `0—2147483647`。

SmallIntegerField:

小整形。值的区间是 `-32768—32767`。

PositiveSmallIntegerField:

正小整形。值的区间是 `0—32767`。

TextField:

大量的文本类型。映射到数据库中是`longtext`类型。

UUIDField:

只能存储 `uuid` 格式的字符串。`uuid` 是一个32位的全球唯一的字符串，一般用来作为主键。

URLField:

类似于 `CharField`，只不过只能用来存储 `url` 格式的字符串。并且默认的 `max_length` 是200。

Field的常用参数:

null:

如果设置为 `True`，`Django` 将会在映射表的时候指定是否为空。默认是为 `False`。在使用字符串相关的 `Field` (`CharField/TextField`) 的时候，官方推荐尽量不要使用这个参数，也就是保持默认值 `False`。因为 `Django` 在处理字符串相关的 `Field` 的时候，即使这个 `Field` 的 `null=False`，如果你没有给这个 `Field` 传递任何值，那么 `Django` 也会使用一个空的字符串 `""` 来作为默认值存储进去。因此如果再使用 `null=True`，`Django` 会产生两种空值的情形 (`NULL`或者空字符串)。如果想要在表单验证的时候允许这个字符串为空，那么建议使用 `blank=True`。如果你的 `Field` 是 `BooleanField`，那么对应的可空的字段则为 `NullBooleanField`。

blank:

标识这个字段在表单验证的时候是否可以为空。默认是 `False`。
这个和 `null` 是有区别的，`null` 是一个纯数据库级别的。而 `blank` 是表单验证级别的。

db_column:

这个字段在数据库中的名字。如果没有设置这个参数，那么将会使用模型中属性的名字。

default:

默认值。可以为一个值，或者是一个函数，但是不支持 `lambda` 表达式。并且不支持列表/字典/集合等可变的数据结构。

primary_key:

是否为主键。默认是 `False`。

unique:

在表中这个字段的值是否唯一。一般是设置手机号码/邮箱等。

更多 `Field` 参数请参考官方文档: <https://docs.djangoproject.com/zh-hans/2.0/ref/models/fields/>

模型中 `Meta` 配置:

对于一些模型级别的配置。我们可以在模型中定义一个类,叫做 `Meta`。然后在这个类中添加一些类属性来控制模型的作用。比如我们想要在数据库映射的时候使用自己指定的表名,而不是使用模型的名称。那么我们可以在 `Meta` 类中添加一个 `db_table` 的属性。示例代码如下:

```
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    desc = models.CharField(max_length=100,name='description',db_column="description1")

    class Meta:
        db_table = 'book_model'
```

以下将对 `Meta` 类中的一些常用配置进行解释。

`db_table`:

这个模型映射到数据库中的表名。如果没有指定这个参数,那么在映射的时候将会使用模型名来作为默认的表名。

ordering:

设置在提取数据的排序方式。后面章节会讲到如何查找数据。比如我想在查找数据的时候根据添加的时间排序,那么示例代码如下:

```
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    desc = models.CharField(max_length=100,name='description',db_column="description1")
    pub_date = models.DateTimeField(auto_now_add=True)

    class Meta:
        db_table = 'book_model'
        ordering = ['pub_date']
```

更多的配置后面会慢慢介绍到。 官方文

档: <https://docs.djangoproject.com/en/2.0/ref/models/options/>

外键和表关系

外键：

在 MySQL 中，表有两种引擎，一种是 InnoDB，另外一种是 myisam。如果使用的是 InnoDB 引擎，是支持外键约束的。外键的存在使得 ORM 框架在处理表关系的时候异常的强大。因此这里我们首先来介绍下外键在 Django 中的使用。

类定义为 `class ForeignKey(to,on_delete,**options)`。第一个参数是引用的是哪个模型，第二个参数是在使用外键引用的模型数据被删除了，这个字段该如何处理，比如有 CASCADE、SET_NULL 等。这里以一个实际案例来说明。比如有一个 User 和一个 Article 两个模型。一个 User 可以发表多篇文章，一个 Article 只能有一个 Author，并且通过外键进行引用。那么相关的示例代码如下：

```
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    author = models.ForeignKey("User",on_delete=models.CASCADE)
```

以上使用 ForeignKey 来定义模型之间的关系。即在 article 的实例中可以通过 author 属性来操作对应的 User 模型。这样使用起来非常的方便。示例代码如下：

```
article = Article(title='abc',content='123')
author = User(username='张三',password='111111')
article.author = author
article.save()

# 修改article.author上的值
article.author.username = '李四'
article.save()
```

为什么使用了 ForeignKey 后，就能通过 author 访问到对应的 user 对象呢。因此在底层，Django 为 Article 表添加了一个 属性名_id 的字段（比如author的字段名称是author_id），这个字段是一个外键，记录着对应的作者的主键。以后通过 article.author 访问的时候，实际上是先通过 author_id 找到对应的数据，然后再提取 User 表中的这条数据，形成一个模型。

如果想要引用另外一个 `app` 的模型，那么应该在传递 `to` 参数的时候，使用 `app.model_name` 进行指定。以上例为例，如果 `User` 和 `Article` 不是在同一个 `app` 中，那么在引用的时候的示例代码如下：

```
# User模型在user这个app中
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

# Article模型在article这个app中
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    author = models.ForeignKey("user.User", on_delete=models.CASCADE)
```

如果模型的外键引用的是本身自己这个模型，那么 `to` 参数可以为 `'self'`，或者是这个模型的名字。在论坛开发中，一般评论都可以进行二级评论，即可以针对另外一个评论进行评论，那么在定义模型的时候就需要使用外键来引用自身。示例代码如下：

```
class Comment(models.Model):
    content = models.TextField()
    origin_comment = models.ForeignKey('self', on_delete=models.CASCADE, null=True)
    # 或者
    # origin_comment = models.ForeignKey('Comment', on_delete=models.CASCADE, null=True)
```

外键删除操作：

如果一个模型使用了外键。那么在对方那个模型被删掉后，该进行什么样的操作。可以通过 `on_delete` 来指定。可以指定的类型如下：

1. `CASCADE`：级联操作。如果外键对应的那条数据被删除了，那么这条数据也会被删除。
2. `PROTECT`：受保护。即只要这条数据引用了外键的那条数据，那么就不能删除外键的那条数据。
3. `SET_NULL`：设置为空。如果外键的那条数据被删除了，那么在本条数据上就将这个字段设置为空。如果设置这个选项，前提是要指定这个字段可以为空。
4. `SET_DEFAULT`：设置默认值。如果外键的那条数据被删除了，那么本条数据上就将这个字段设置为默认值。如果设置这个选项，前提是要指定这个字段一个默认值。
5. `SET()`：如果外键的那条数据被删除了。那么将会获取 `SET` 函数中的值来作为这个外键的值。`SET` 函数可以接收一个可以调用的对象（比如函数或者方法），如果是可以调用的对象，那么会将这个对象调用后的结果作为值返回回去。
6. `DO_NOTHING`：不采取任何行为。一切全看数据库级别的约束。

以上这些选项只是 **Django** 级别的，数据级别依旧是 **RESTRICT**！

表关系：

表之间的关系都是通过外键来进行关联的。而表之间的关系，无非就是三种关系：一对一、一对多（多对一）、多对多等。以下将讨论一下三种关系的应用场景及其实现方式。

一对多：

1. 应用场景：比如文章和作者之间的关系。一个文章只能由一个作者编写，但是一个作者可以写多篇文章。文章和作者之间的关系就是典型的多对一的关系。
2. 实现方式：一对多或者多对一，都是通过 `ForeignKey` 来实现的。还是以文章和作者的案例进行讲解。

```
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey("User", on_delete=models.CASCADE)
```

那么以后在给 `Article` 对象指定 `author`，就可以使用以下代码来完成：

```
article = Article(title='abc', content='123')
author = User(username='zhiliao', password='111111')
# 要先保存到数据库中
author.save()
article.author = author
article.save()
```

并且以后如果想要获取某个用户下所有的文章，可以通过 `article_set` 来实现。示例代码如下：

```
user = User.objects.first()
# 获取第一个用户写的所有文章
articles = user.article_set.all()
for article in articles:
    print(article)
```

一对一：

1. 应用场景：比如一个用户表和一个用户信息表。在实际网站中，可能需要保存用户的许多信息，但是有些信息是不经常用的。如果把所有信息都存放到一张表中可能会影响查询效率，因此可以把用户的一些不常用的信息存放到另外一张表中我们叫做 `UserExtension`。但是用户表 `User` 和用户信息表 `UserExtension` 就是典型的一对一了。
2. 实现方式：Django 为一对一提供了一个专门的 `Field` 叫做 `OneToOneField` 来实现一对一操作。示例代码如下：

```
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

class UserExtension(models.Model):
    birthday = models.DateTimeField(null=True)
    school = models.CharField(blank=True,max_length=50)
    user = models.OneToOneField("User", on_delete=models.CASCADE)
```

在 `UserExtension` 模型上增加了一个一对一的关系映射。其实底层是在 `UserExtension` 这个表上增加了一个 `user_id`，来和 `user` 表进行关联，并且这个外键数据在表中必须是唯一的，来保证一对一。

多对多：

1. 应用场景：比如文章和标签的关系。一篇文章可以有多个标签，一个标签可以被多个文章所引用。因此标签和文章的关系是典型的多对多的关系。
2. 实现方式：Django 为这种多对多的实现提供了专门的 `Field`。叫做 `ManyToManyField`。还是拿文章和标签为例进行讲解。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    tags = models.ManyToManyField("Tag",related_name="articles")

class Tag(models.Model):
    name = models.CharField(max_length=50)
```

在数据库层面，实际上 Django 是为这种多对多的关系建立了一个中间表。这个中间表分别定义了两个外键，引用到 `article` 和 `tag` 两张表的主键。

`related_name`和`related_query_name`:

`related_name`:

还是以 `User` 和 `Article` 为例来进行说明。如果一个 `article` 想要访问对应的作者，那么可以通过 `author` 来进行访问。但是如果有一个 `user` 对象，想要通过这个 `user` 对象获取所有的文章，该如何做呢？这时候可以通过 `user.article_set` 来访问，这个名字的规律是 模型名字小写_set。示例代码如下：

```
user = User.objects.get(name='张三')
user.article_set.all()
```

如果不想使用 模型名字小写_set 的方式，想要使用其他的名字，那么可以在定义模型的时候指定 `related_name`。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    # 传递related_name参数，以后在方向引用的时候使用articles进行访问
    author = models.ForeignKey("User", on_delete=models.SET_NULL, null=True, related_name='articles')
```

以后在方向引用的时候。使用 `articles` 可以访问到这个作者的文章模型。示例代码如下：

```
user = User.objects.get(name='张三')
user.articles.all()
```

如果不想使用反向引用，那么可以指定 `related_name='+'`。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    # 传递related_name参数，以后在方向引用的时候使用articles进行访问
    author = models.ForeignKey("User", on_delete=models.SET_NULL, null=True, related_name='+')
```

以后将不能通过 `user.article_set` 来访问文章模型了。

related_query_name:

在查找数据的时候，可以使用 `filter` 进行过滤。使用 `filter` 过滤的时候，不仅仅可以指定本模型上的某个属性要满足什么条件，还可以指定相关联的模型满足什么属性。比如现在想要获取写过标题为 `abc` 的所有用户，那么可以这样写：

```
users = User.objects.filter(article__title='abc')
```

如果你设置了 `related_name` 为 `articles`，因为反转的过滤器的名字将使用 `related_name` 的名字，那么上例代码将改成如下：

```
users = User.objects.filter(articles__title='abc')
```

可以通过 `related_query_name` 将查询的反转名字修改成其他的名字。比如 `article`。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    # 传递related_name参数，以后在方向引用的时候使用articles进行访问
    author = models.ForeignKey("User", on_delete=models.SET_NULL, null=True, related_name=
'articles', related_query_name='article')
```

那么在做反向过滤查找的时候就可以使用以下代码：

```
users = User.objects.filter(article__title='abc')
```


模型的操作：

在 ORM 框架中，所有模型相关的操作，比如添加/删除等。其实都是映射到数据库中一条数据的操作。因此模型操作也就是数据库表中数据的操作。

添加一个模型到数据库中：

添加模型到数据库中。首先需要创建一个模型。创建模型的方式很简单，就跟创建普通的 Python 对象是一摸一样的。在创建完模型之后，需要调用模型的 save 方法，这样 Django 会自动的将这个模型转换成 sql 语句，然后存储到数据库中。示例代码如下：

```
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    desc = models.CharField(max_length=100,name='description',db_column="description1")
    pub_date = models.DateTimeField(auto_now_add=True)

book = Book(name='三国演义',desc='三国英雄! ')
book.save()
```

查找数据：

查找数据都是通过模型下的 objects 对象来实现的。

查找所有数据：

要查找 Book 这个模型对应的表下的所有数据。那么示例代码如下：

```
books = Book.objects.all()
```

以上将返回 Book 模型下的所有数据。

数据过滤：

在查找数据的时候，有时候需要对一些数据进行过滤。那么这时候需要调用 objects 的 filter 方法。实例代码如下：

```
books = Book.objects.filter(name='三国演义')
> [<Book:三国演义>]

# 多个条件
books = Book.objects.filter(name='三国演义',desc='test')
```

调用 `filter`，会将所有满足条件的模型对象都返回。

获取单个对象：

使用 `filter` 返回的是所有满足条件的结果集。有时候如果只需要返回第一个满足条件的对象。那么可以使用 `get` 方法。示例代码如下：

```
book = Book.objects.get(name='三国演义')
> <Book:三国演义>
```

当然，如果没有找到满足条件的对象，那么就会抛出一个异常。而 `filter` 在没有找到满足条件的数据的时候，是返回一个空的列表。

数据排序：

在之前的例子中，数据都是无序的。如果你想在查找数据的时候使用某个字段来进行排序，那么可以使用 `order_by` 方法来实现。示例代码如下：

```
books = Book.objects.order_by("pub_date")
```

以上代码在提取所有书籍的数据的时候，将会使用 `pub_date` 从小到大进行排序。如果想要进行倒序排序，那么可以在 `pub_date` 前面加一个负号。实例代码如下：

```
books = Book.objects.order_by("-pub_date")
```

修改数据：

在查找到数据后，便可以进行修改了。修改的方式非常简单，只需要将查找出来的对象的某个属性进行修改，然后再调用这个对象的 `save` 方法便可以进行修改。示例代码如下：

```
from datetime import datetime
book = Book.objects.get(name='三国演义')
book.pub_date = datetime.now()
book.save()
```

删除数据：

在查找到数据后，便可以进行删除了。删除数据非常简单，只需要调用这个对象的 `delete` 方法即可。实例代码如下：

```
book = Book.objects.get(name='三国演义')
```

```
book.delete()
```

查询操作

查找是数据库操作中一个非常重要的技术。查询一般就是使用 `filter`、`exclude` 以及 `get` 三个方法来实现。我们可以在调用这些方法的时候传递不同的参数来实现查询需求。在 ORM 层面，这些查询条件都是使用 `field + __ + condition` 的方式来使用的。以下将那些常用的查询条件来一一解释。

查询条件

exact:

使用精确的 `=` 进行查找。如果提供的是一个 `None`，那么在 SQL 层面就是被解释为 `NULL`。示例代码如下：

```
article = Article.objects.get(id__exact=14)
article = Article.objects.get(id__exact=None)
```

以上的两个查找在翻译为 SQL 语句为如下：

```
select ... from article where id=14;
select ... from article where id IS NULL;
```

iexact:

使用 `like` 进行查找。示例代码如下：

```
article = Article.objects.filter(title__iexact='hello world')
```

那么以上的查询就等价于以下的 SQL 语句：

```
select ... from article where title like 'hello world';
```

注意上面这个 sql 语句，因为在 MySQL 中，没有一个叫做 `ilike` 的。所以 `exact` 和 `iexact` 的区别实际上就是 `LIKE` 和 `=` 的区别，在大部分 `collation=utf8_general_ci` 情况下都是一样的（`collation` 是用来对字符串比较的）。

contains:

大小写敏感，判断某个字段是否包含了某个数据。示例代码如下：

```
articles = Article.objects.filter(title__contains='hello')
```

在翻译成 SQL 语句为如下：

```
select ... where title like binary '%hello%';
```

要注意的是，在使用 `contains` 的时候，翻译成的 sql 语句左右两边是有百分号的，意味着使用的是模糊查询。而 `exact` 翻译成 sql 语句左右两边是没有百分号的，意味着使用的是精确的查询。

icontains:

大小写不敏感的匹配查询。示例代码如下：

```
articles = Article.objects.filter(title__icontains='hello')
```

在翻译成 SQL 语句为如下：

```
select ... where title like '%hello%';
```

in:

提取那些给定的 `field` 的值是否在给定的容器中。容器可以为 `list` 、 `tuple` 或者任何一个可以迭代的对象，包括 `QuerySet` 对象。示例代码如下：

```
articles = Article.objects.filter(id__in=[1,2,3])
```

以上代码在翻译成 SQL 语句为如下：

```
select ... where id in (1,3,4)
```

当然也可以传递一个 `QuerySet` 对象进去。示例代码如下：

```
inner_qs = Article.objects.filter(title__contains='hello')
categories = Category.objects.filter(article__in=inner_qs)
```

以上代码的意思是获取那些文章标题包含 `hello` 的所有分类。

将翻译成以下 SQL 语句，示例代码如下：

```
select ...from category where article.id in (select id from article where title like '%hello%');
```

gt:

某个 `field` 的值要大于给定的值。示例代码如下：

```
articles = Article.objects.filter(id__gt=4)
```

以上代码的意思是将所有 `id` 大于4的文章全部都找出来。

将翻译成以下 SQL 语句：

```
select ... where id > 4;
```

gte:

类似于 `gt`，是大于等于。

lt:

类似于 `gt` 是小于。

lte:

类似于 `lt`，是小于等于。

startswith:

判断某个字段的值是否是以某个值开始的。大小写敏感。示例代码如下：

```
articles = Article.objects.filter(title__startswith='hello')
```

以上代码的意思是提取所有标题以 `hello` 字符串开头的文章。

将翻译成以下 SQL 语句：

```
select ... where title like 'hello%'
```

istartswith:

类似于 `startswith`，但是大小写是不敏感的。

endswith:

判断某个字段的值是否以某个值结束。大小写敏感。示例代码如下：

```
articles = Article.objects.filter(title__endswith='world')
```

以上代码的意思是提取所有标题以 `world` 结尾的文章。

将翻译成以下 `SQL` 语句：

```
select ... where title like '%world';
```

icontains:

类似于 `endswith`，只不过大小写不敏感。

range:

判断某个 `field` 的值是否在给定的区间中。示例代码如下：

```
from django.utils.timezone import make_aware
from datetime import datetime
start_date = make_aware(datetime(year=2018,month=1,day=1))
end_date = make_aware(datetime(year=2018,month=3,day=29,hour=16))
articles = Article.objects.filter(pub_date__range=(start_date,end_date))
```

以上代码的意思是提取所有发布时间在 `2018/1/1` 到 `2018/12/12` 之间的文章。

将翻译成以下的 `SQL` 语句：

```
select ... from article where pub_time between '2018-01-01' and '2018-12-12'。
```

需要注意的是，以上提取数据，不会包含最后一个值。也就是不会包含 `2018/12/12` 的文章。

而且另外一个重点，因为我们在 `settings.py` 中指定了 `USE_TZ=True`，并且设置

了 `TIME_ZONE='Asia/Shanghai'`，因此我们在提取数据的时候要使

用 `django.utils.timezone.make_aware` 先将 `datetime.datetime` 从 naive 时间转换为 aware 时间。`make_aware` 会将指定的时间转换为 `TIME_ZONE` 中指定的时区的时间。

date:

针对某些 `date` 或者 `datetime` 类型的字段。可以指定 `date` 的范围。并且这个时间过滤，还可以使用链式调用。示例代码如下：

```
articles = Article.objects.filter(pub_date__date=date(2018,3,29))
```

以上代码的意思是查找时间为 2018/3/29 这一天发表的所有文章。
将翻译成以下的 sql 语句：

```
select ... WHERE DATE(CONVERT_TZ(`front_article`.`pub_date`, 'UTC', 'Asia/Shanghai')) =  
2018-03-29
```

注意，因为默认情况下 MySQL 的表中是没有存储时区相关的信息的。因此我们需要下载一些时区表的文件，然后添加到 MySQL 的配置路径中。如果你用的是 windows 操作系统。那么在 http://dev.mysql.com/downloads/tz/2018d_posix.zip 下载 `timezone_2018d_posix.zip` - POSIX standard 。然后将下载下来的所有文件拷贝到 `C:\ProgramData\MySQL\MySQL Server 5.7\Data\mysql` 中，如果提示文件名重复，那么选择覆盖即可。
如果用的是 linux 或者 mac 系统，那么在命令行中执行以下命令：`mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -D mysql -u root -p`，然后输入密码，从系统中加载时区文件更新到 mysql 中。

year:

根据年份进行查找。示例代码如下：

```
articles = Article.objects.filter(pub_date__year=2018)  
articles = Article.objects.filter(pub_date__year__gte=2017)
```

以上的代码在翻译成 SQL 语句为如下：

```
select ... where pub_date between '2018-01-01' and '2018-12-31';  
select ... where pub_date >= '2017-01-01';
```

month:

同 year，根据月份进行查找。

day:

同 year，根据日期进行查找。

week_day:

Django 1.11 新增的查找方式。同 year，根据星期几进行查找。1表示星期天，7表示星期六，2-6 代表的是星期一到星期五。

time:

根据时间进行查找。示例代码如下：

```
articles = Article.objects.filter(pub_date__time=datetime.time(12,12,12));
```

以上的代码是获取每一天中12点12分12秒发表的所有文章。

更多的关于时间的过滤，请参考 Django 官方文

档：<https://docs.djangoproject.com/en/2.0/ref/models/queries/#range>。

isnull:

根据值是否为空进行查找。示例代码如下：

```
articles = Article.objects.filter(pub_date__isnull=False)
```

以上的代码的意思是获取所有发布日期不为空的文章。

将来翻译成 SQL 语句如下：

```
select ... where pub_date is not null;
```

regex和iregex:

大小写敏感和大小写不敏感的正则表达式。示例代码如下：

```
articles = Article.objects.filter(title__regex=r'^hello')
```

以上代码的意思是提取所有标题以 hello 字符串开头的文章。

将翻译成以下的 SQL 语句：

```
select ... where title regexp binary '^hello';
```

iregex 是大小写不敏感的。

根据关联的表进行查询:

假如现在有两个 ORM 模型，一个是 Article ，一个是 Category 。代码如下：

```
class Category(models.Model):
    """文章分类表"""
    name = models.CharField(max_length=100)

class Article(models.Model):
    """文章表"""
```

```
title = models.CharField(max_length=100,null=True)
category = models.ForeignKey("Category",on_delete=models.CASCADE)
```

比如想要获取文章标题中包含"hello"的所有的分类。那么可以通过以下代码来实现：

```
categories = Category.object.filter(article__title__contains("hello"))
```

聚合函数：

如果你用原生 SQL，则可以使用聚合函数来提取数据。比如提取某个商品销售的数量，那么可以使用 Count，如果想要知道商品销售的平均价格，那么可以使用 Avg。

聚合函数是通过 aggregate 方法来实现的。在讲解这些聚合函数的用法的时候，都是基于以下的模型对象来实现的。

```
from django.db import models

class Author(models.Model):
    """作者模型"""
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField()

    class Meta:
        db_table = 'author'

class Publisher(models.Model):
    """出版社模型"""
    name = models.CharField(max_length=300)

    class Meta:
        db_table = 'publisher'

class Book(models.Model):
    """图书模型"""
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.FloatField()
    rating = models.FloatField()
    author = models.ForeignKey(Author,on_delete=models.CASCADE)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)

    class Meta:
```

```

        db_table = 'book'

class BookOrder(models.Model):
    """图书订单模型"""
    book = models.ForeignKey("Book", on_delete=models.CASCADE)
    price = models.FloatField()

    class Meta:
        db_table = 'book_order'

```

1. **Avg** : 求平均值。比如想要获取所有图书的价格平均值。那么可以使用以下代码实现。

```

from django.db.models import Avg
result = Book.objects.aggregate(Avg('price'))
print(result)

```

以上的打印结果是:

```

{"price__avg":23.0}

```

其中 `price__avg` 的结构是根据 `field__avg` 规则构成的。如果想要修改默认的名字，那么可以将 `Avg` 赋值给一个关键字参数。示例代码如下：

```

from django.db.models import Avg
result = Book.objects.aggregate(my_avg=Avg('price'))
print(result)

```

那么以上的结果打印为:

```

{"my_avg":23}

```

2. **Count** : 获取指定的对象的个数。示例代码如下:

```

from django.db.models import Count
result = Book.objects.aggregate(book_num=Count('id'))

```

以上的 `result` 将返回 `Book` 表中总共有多少本图书。

`Count` 类中，还有另外一个参数叫做 `distinct`，默认是等于 `False`，如果是等于 `True`，那么将去掉那些重复的值。比如要获取作者表中所有的不重复的邮箱总共有多少个，那么可以通过以下代码来实现：

```

from django.db.models import Count

```

```
result = Author.objects.aggregate(count=Count('email',distinct=True))
```

3. **Max** 和 **Min** : 获取指定对象的最大值和最小值。比如想要获取 **Author** 表中, 最大的年龄和最小的年龄分别是多少。那么可以通过以下代码来实现:

```
from django.db.models import Max,Min
result = Author.objects.aggregate(Max('age'),Min('age'))
```

如果最大的年龄是88,最小的年龄是18。那么以上的result将为:

```
{"age__max":88,"age__min":18}
```

4. **Sum** : 求指定对象的总和。比如要求图书的销售总额。那么可以使用以下代码实现:

```
from django.db.models import Sum
result = Book.objects.annotate(total=Sum("bookstore__price")).values("name","total")
```

以上的代码 **annotate** 的意思是给 **Book** 表在查询的时候添加一个字段叫做 **total** , 这个字段的数据来源是从 **BookStore** 模型的 **price** 的总和而来。 **values** 方法是只提取 **name** 和 **total** 两个字段的值。

更多的聚合函数请参考官方文

档: <https://docs.djangoproject.com/en/2.0/ref/models/queriesets/#aggregation-functions>

aggregate和annotate的区别:

1. **aggregate** : 返回使用聚合函数后的字段和值。
2. **annotate** : 在原来模型字段的基础之上添加一个使用了聚合函数的字段, 并且在使用聚合函数的时候, 会使用当前这个模型的主键进行分组 (**group by**)。
比如以上 **Sum** 的例子, 如果使用的是 **annotate** , 那么将在每条图书的数据上都添加一个字段叫做 **total** , 计算这本书的销售总额。
而如果使用的是 **aggregate** , 那么将求所有图书的销售总额。

F表达式和Q表达式:

F表达式:

F表达式 是用来优化 **ORM** 操作数据库的。比如我们要将公司所有员工的薪水都增加1000元，如果按照正常的流程，应该是先从数据库中提取所有的员工工资到Python内存中，然后使用Python代码在员工工资的基础之上增加1000元，最后再保存到数据库中。这里面涉及的流程就是，首先从数据库中提取数据到Python内存中，然后在Python内存中做完运算，之后再保存到数据库中。示例代码如下：

```
employees = Employee.objects.all()
for employee in employees:
    employee.salary += 1000
    employee.save()
```

而我们的 **F表达式** 就可以优化这个流程，他可以不需要先把数据从数据库中提取出来，计算完成后保存回去，他可以直接执行 **SQL**语句，就将员工的工资增加1000元。示例代码如下：

```
from django.db.models import F
Employee.object.update(salary=F("salary")+1000)
```

F表达式 并不会马上从数据库中获取数据，而是在生成 **SQL** 语句的时候，动态的获取传给 **F表达式** 的值。

比如如果想要获取作者中，**name** 和 **email** 相同的作者数据。如果不使用 **F表达式**，那么需要使用以下代码来完成：

```
authors = Author.objects.all()
for author in authors:
    if author.name == author.email:
        print(author)
```

如果使用 **F表达式**，那么一行代码就可以搞定。示例代码如下：

```
from django.db.models import F
authors = Author.objects.filter(name=F("email"))
```

Q表达式：

如果想要实现所有价格高于100元，并且评分达到9.0以上评分的图书。那么可以通过以下代码来实现：

```
books = Book.objects.filter(price__gte=100, rating__gte=9)
```

以上这个案例是一个并集查询，可以简单的通过传递多个条件进去来实现。

但是如果想要实现一些复杂的查询语句，比如要查询所有价格低于10元，或者是评分低于9分的图书。那就没有办法通过传递多个条件进去实现了。这时候就需要使用 **Q表达式** 来实现了。示例代码

如下：

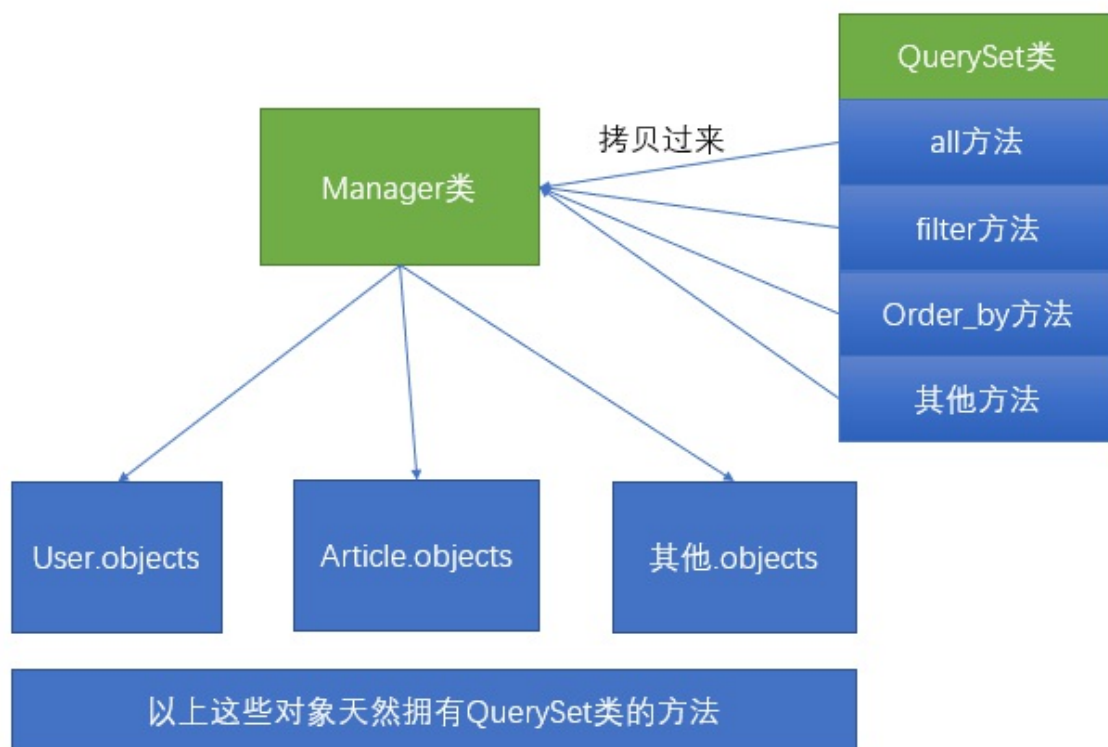
```
from django.db.models import Q
books = Book.objects.filter(Q(price__lte=10) | Q(rating__lte=9))
```

以上是进行或运算，当然还可以进行其他的运算，比如有 `&` 和 `~`（非）等。一些用 `Q` 表达式的例子如下：

```
from django.db.models import Q
# 获取id等于3的图书
books = Book.objects.filter(Q(id=3))
# 获取id等于3，或者名字中包含文字"记"的图书
books = Book.objects.filter(Q(id=3)|Q(name__contains("记")))
# 获取价格大于100，并且书名中包含"记"的图书
books = Book.objects.filter(Q(price__gte=100)&Q(name__contains("记")))
# 获取书名包含“记”，但是id不等于3的图书
books = Book.objects.filter(Q(name__contains='记') & ~Q(id=3))
```

QuerySet API:

我们通常做查询操作的时候，都是通过 `模型名字.objects` 的方式进行操作。其实 `模型名字.objects` 是一个 `django.db.models.manager.Manager` 对象，而 `Manager` 这个类是一个“空壳”的类，他本身是没有任何的属性和方法的。他的方法全部都是通过 `Python` 动态添加的方式，从 `QuerySet` 类中拷贝过来的。示例图如下：



所以我们如果想要学习 `ORM` 模型的查找操作，必须首先要学会 `QuerySet` 上的一些 `API` 的使用。

返回新的QuerySet的方法：

在使用 `QuerySet` 进行查找操作的时候，可以提供多种操作。比如过滤完后还要根据某个字段进行排序，那么这一系列的操作我们可以通过一个非常流畅的 `链式调用` 的方式进行。比如要从文章表中获取标题为 `123`，并且提取后要将结果根据发布的时间进行排序，那么可以使用以下方式来完成：

```
articles = Article.objects.filter(title='123').order_by('create_time')
```

可以看到 `order_by` 方法是直接在 `filter` 执行后调用的。这说明 `filter` 返回的对象是一个拥有 `order_by` 方法的对象。而这个对象正是一个新的 `QuerySet` 对象。因此可以使用 `order_by` 方法。

那么以下将介绍在那些会返回新的 `QuerySet` 对象的方法。

1. `filter` : 将满足条件的数据提取出来, 返回一个新的 `QuerySet` 。具体的 `filter` 可以提供什么条件查询。请见查询操作章节。

2. `exclude` : 排除满足条件的数据, 返回一个新的 `QuerySet` 。示例代码如下:

```
Article.objects.exclude(title__contains='hello')
```

以上代码的意思是提取那些标题不包含 `hello` 的图书。

3. `annotate` : 给 `QuerySet` 中的每个对象都添加一个使用查询表达式（聚合函数、F表达式、Q表达式、Func表达式等）的新字段。示例代码如下:

```
articles = Article.objects.annotate(author_name=F("author__name"))
```

以上代码将在每个对象中都添加一个 `author_name` 的字段, 用来显示这个文章的作者的年龄。

4. `order_by` : 指定将查询的结果根据某个字段进行排序。如果要倒叙排序, 那么可以在这个字段的前面加一个负号。示例代码如下:

```
# 根据创建的时间正序排序
articles = Article.objects.order_by("create_time")
# 根据创建的时间倒序排序
articles = Article.objects.order_by("-create_time")
# 根据作者的名字进行排序
articles = Article.objects.order_by("author__name")
# 首先根据创建的时间进行排序, 如果时间相同, 则根据作者的名字进行排序
articles = Article.objects.order_by("create_time", 'author__name')
```

一定要注意的一点是, 多个 `order_by` , 会把前面排序的规则给打乱, 而使用后面的排序方式。比如以下代码:

```
articles = Article.objects.order_by("create_time").order_by("author__name")
```

他会根据作者的名字进行排序, 而不是使用文章的创建时间。

5. `values` : 用来指定在提取数据出来, 需要提取哪些字段。默认情况下会把表中所有的字段全都提取出来, 可以使用 `values` 来进行指定, 并且使用了 `values` 方法后, 提取出的 `QuerySet` 中的数据类型不是模型, 而是在 `values` 方法中指定的字段和值形成的字典:

```
articles = Article.objects.values("title", 'content')
for article in articles:
    print(article)
```


以上打印出来的 `article` 是类似于 `{"title":"abc","content":"xxx"}` 的形式。

如果在 `values` 中没有传递任何参数，那么将会返回这个模型中所有的属性。

6. `values_list`：类似于 `values`。只不过返回的 `QuerySet` 中，存储的不是字典，而是元组。示例代码如下：

```
articles = Article.objects.values_list("id","title")
print(articles)
```

那么在打印 `articles` 后，结果为 `<QuerySet [(1,'abc'),(2,'xxx'),...]>` 等。

如果在 `values_list` 中只有一个字段。那么你可以传递 `flat=True` 来将结果扁平化。示例代码如下：

```
articles1 = Article.objects.values_list("title")
>> <QuerySet [("abc",),("xxx",),...]>
articles2 = Article.objects.values_list("title",flat=True)
>> <QuerySet ["abc",'xxx',...]>
```

7. `all`：获取这个 ORM 模型的 `QuerySet` 对象。
8. `select_related`：在提取某个模型的数据的同时，也提前将相关联的数据提取出来。比如提取文章数据，可以使用 `select_related` 将 `author` 信息提取出来，以后再次使用 `article.author` 的时候就不需要再次去访问数据库了。可以减少数据库查询的次数。示例代码如下：

```
article = Article.objects.get(pk=1)
>> article.author # 重新执行一次查询语句
article = Article.objects.select_related("author").get(pk=2)
>> article.author # 不需要重新执行查询语句了
```

`select_related` 只能用在 一对多 或者 一对一 中，不能用在 多对多 或者 多对一 中。比如可以提前获取文章的作者，但是不能通过作者获取这个作者的文章，或者是通过某篇文章获取这篇文章所有的标签。

9. `prefetch_related`：这个方法和 `select_related` 非常的类似，就是在访问多个表中的数据的时候，减少查询的次数。这个方法是为了解决 多对一 和 多对多 的关系的查询问题。比如要获取标题中带有 `hello` 字符串的文章以及他的所有标签，示例代码如下：

```
from django.db import connection
articles = Article.objects.prefetch_related("tag_set").filter(title__contains='hello')
print(articles.query) # 通过这条命令查看在底层的SQL语句
for article in articles:
    print("title:",article.title)
    print(article.tag_set.all())
```

```
# 通过以下代码可以看出以上代码执行的sql语句
for sql in connection.queries:
    print(sql)
```

但是如果在使用 `article.tag_set` 的时候，如果又创建了一个新的 `QuerySet` 那么会把之前的 `SQL` 优化给破坏掉。比如以下代码：

```
tags = Tag.objects.prefetch_related("articles")
for tag in tags:
    articles = tag.articles.filter(title__contains='hello') #因为filter方法会重新生成一个QuerySet，因此会破坏掉之前的sql优化

# 通过以下代码，我们可以看到在使用了filter的，他的sql查询会更多，而没有使用filter的，只有两次sql查询
for sql in connection.queries:
    print(sql)
```

那如果确实是在查询的时候指定过滤条件该如何做呢，这时候我们可以使用 `django.db.models.Prefetch` 来实现，`Prefetch` 这个可以提前定义好 `queryset`。示例代码如下：

```
tags = Tag.objects.prefetch_related(Prefetch("articles", queryset=Article.objects.filter(title__contains='hello'))).all()
for tag in tags:
    articles = tag.articles.all()
    for article in articles:
        print(article)

for sql in connection.queries:
    print('='*30)
    print(sql)
```

因为使用了 `Prefetch`，即使在查询文章的时候使用了 `filter`，也只会发生两次查询操作。

10. `defer`：在一些表中，可能存在很多的字段，但是一些字段的数据量可能是比较庞大的，而此时你又不需要，比如我们在获取文章列表的时候，文章的内容我们是不需要的，因此这时候我们就可以使用 `defer` 来过滤掉一些字段。这个字段跟 `values` 有点类似，只不过 `defer` 返回的不是字典，而是模型。示例代码如下：

```
articles = list(Article.objects.defer("title"))
for sql in connection.queries:
    print('='*30)
    print(sql)
```

在看以上代码的 `sql` 语句，你就可以看到，查找文章的字段，除了 `title`，其他字段都查找出来了。当然，你也可以使用 `article.title` 来获取这个文章的标题，但是会重新执行一个查询的语句。示例代码如下：

```
articles = list(Article.objects.defer("title"))
for article in articles:
    # 因为在上面提取的时候过滤了title
    # 这个地方重新获取title，将重新向数据库中进行一次查找操作
    print(article.title)
for sql in connection.queries:
    print('='*30)
    print(sql)
```

`defer` 虽然能过滤字段，但是有些字段是不能过滤的，比如 `id`，即使你过滤了，也会提取出来。

11. `only`：跟 `defer` 类似，只不过 `defer` 是过滤掉指定的字段，而 `only` 是只提取指定的字段。
12. `get`：获取满足条件的数据。这个函数只能返回一条数据，并且如果给的条件有多条数据，那么这个方法会抛出 `MultipleObjectsReturned` 错误，如果给的条件没有任何数据，那么就会抛出 `DoesNotExist` 错误。所以这个方法在获取数据的只能，只能有且只有一条。
13. `create`：创建一条数据，并且保存到数据库中。这个方法相当于先用指定的模型创建一个对象，然后再调用这个对象的 `save` 方法。示例代码如下：

```
article = Article(title='abc')
article.save()

# 下面这行代码相当于以上两行代码
article = Article.objects.create(title='abc')
```

14. `get_or_create`：根据某个条件进行查找，如果找到了那么就返回这条数据，如果没有查找找到，那么就创建一个。示例代码如下：

```
obj, created= Category.objects.get_or_create(title='默认分类')
```

如果有标题等于 `默认分类` 的分类，那么就会查找出来，如果没有，则会创建并且存储到数据库中。

这个方法的返回值是一个元组，元组的第一个参数 `obj` 是这个对象，第二个参数 `created` 代表是否创建的。

15. `bulk_create`：一次性创建多个数据。示例代码如下：

```
Tag.objects.bulk_create([
    Tag(name='111'),
    Tag(name='222'),
])
```

16. `count` : 获取提取的数据的个数。如果想要知道总共有多少条数据，那么建议使用 `count`，而不是使用 `len(articles)` 这种。因为 `count` 在底层是使用 `select count(*)` 来实现的，这种方式比使用 `len` 函数更加的高效。
17. `first` 和 `last` : 返回 `QuerySet` 中的第一条和最后一条数据。
18. `aggregate` : 使用聚合函数。
19. `exists` : 判断某个条件的数据是否存在。如果要判断某个条件的元素是否存在，那么建议使用 `exists`，这比使用 `count` 或者直接判断 `QuerySet` 更有效得多。示例代码如下：

```
if Article.objects.filter(title__contains='hello').exists():
    print(True)
比使用count更高效:
if Article.objects.filter(title__contains='hello').count() > 0:
    print(True)
也比直接判断QuerySet更高效:
if Article.objects.filter(title__contains='hello'):
    print(True)
```

20. `distinct` : 去除掉那些重复的数据。这个方法如果底层数据库用的是 `MySQL`，那么不能传递任何的参数。比如想要提取所有销售的价格超过80元的图书，并且删掉那些重复的，那么可以使用 `distinct` 来帮我们实现，示例代码如下：

```
books = Book.objects.filter(bookorder__price__gte=80).distinct()
```

需要注意的是，如果在 `distinct` 之前使用了 `order_by`，那么因为 `order_by` 会提取 `order_by` 中指定的字段，因此再使用 `distinct` 就会根据多个字段来进行唯一化，所以就不会把那些重复的数据删掉。示例代码如下：

```
orders = BookOrder.objects.order_by("create_time").values("book_id").distinct()
```

那么以上代码因为使用了 `order_by`，即使使用了 `distinct`，也会把重复的 `book_id` 提取出来。

21. `update` : 执行更新操作，在 `SQL` 底层走的也是 `update` 命令。比如要将所有 `category` 为空的 `article` 的 `article` 字段都更新为默认的分类。示例代码如下：

```
Article.objects.filter(category__isnull=True).update(category_id=3)
```

注意这个方法走的是更新的逻辑。所以更新完成后保存到数据库中不会执行 `save` 方法，因此不会更新 `auto_now` 设置的字段。

22. `delete`：删除所有满足条件的数据。删除数据的时候，要注意 `on_delete` 指定的处理方式。
23. 切片操作：有时候我们查找数据，有可能只需要其中的一部分。那么这时候可以使用切片操作来帮我们完成。`QuerySet` 使用切片操作就跟列表使用切片操作是一样的。示例代码如下：

```
books = Book.objects.all()[1:3]
for book in books:
    print(book)
```

切片操作并不是把所有数据从数据库中提取出来再做切片操作。而是在数据库层面使用 `LIMIT` 和 `OFFSET` 来帮我们完成。所以如果只需要取其中一部分的数据的时候，建议大家使用切片操作。

什么时候 Django 会将 QuerySet 转换为 SQL 去执行：

生成一个 `QuerySet` 对象并不会马上转换为 `SQL` 语句去执行。
比如我们获取 `Book` 表下所有的图书：

```
books = Book.objects.all()
print(connection.queries)
```

我们可以看到在打印 `connection.queries` 的时候打印的是一个空的列表。说明上面的 `QuerySet` 并没有真正的执行。

在以下情况下 `QuerySet` 会被转换为 `SQL` 语句执行：

1. 迭代：在遍历 `QuerySet` 对象的时候，会首先先执行这个 `SQL` 语句，然后再把这个结果返回进行迭代。比如以下代码就会转换为 `SQL` 语句：

```
for book in Book.objects.all():
    print(book)
```

2. 使用步长做切片操作：`QuerySet` 可以类似于列表一样做切片操作。做切片操作本身不会执行 `SQL` 语句，但是如果如果在做切片操作的时候提供了步长，那么就会立马执行 `SQL` 语句。需要注意的是，做切片后不能再执行 `filter` 方法，否则会报错。
3. 调用 `len` 函数：调用 `len` 函数用来获取 `QuerySet` 中总共有多少条数据也会执行 `SQL` 语句。
4. 调用 `list` 函数：调用 `list` 函数用来将一个 `QuerySet` 对象转换为 `list` 对象也会立马执行 `SQL` 语句。
5. 判断：如果对某个 `QuerySet` 进行判断，也会立马执行 `SQL` 语句。

ORM模型迁移

迁移命令：

1. **makemigrations**: 将模型生成迁移脚本。模型所在的 `app`，必须放在 `settings.py` 中的 `INSTALLED_APPS` 中。这个命令有以下几个常用选项：
 - **app_label**: 后面可以跟一个或者多个 `app`，那么就只会针对这几个`app`生成迁移脚本。如果没有任何的`app_label`，那么会检查 `INSTALLED_APPS` 中所有的`app`下的模型，针对每一个`app`都生成响应的迁移脚本。
 - **--name**: 给这个迁移脚本指定一个名字。
 - **--empty**: 生成一个空的迁移脚本。如果你想写自己的迁移脚本，可以使用这个命令来实现一个空的文件，然后自己再在文件中写迁移脚本。
2. **migrate**: 将新生成的迁移脚本。映射到数据库中。创建新的表或者修改表的结构。以下一些常用的选项：
 - **app_label**: 将某个 `app` 下的迁移脚本映射到数据库中。如果没有指定，那么会将所有在 `INSTALLED_APPS` 中的 `app` 下的模型都映射到数据库中。
 - **app_label migrationname**: 将某个 `app` 下指定名字的 `migration` 文件映射到数据库中。
 - **--fake**: 可以将指定的迁移脚本名字添加到数据库中。但是并不会把迁移脚本转换为SQL语句，修改数据库中的表。
 - **--fake-initial**: 将第一次生成的迁移文件版本号记录在数据库中。但并不会真正的执行迁移脚本。
3. **showmigrations**: 查看某个`app`下的迁移文件。如果后面没有`app`，那么将查看 `INSTALLED_APPS` 中所有的迁移文件。
4. **sqlmigrate**: 查看某个迁移文件在映射到数据库中的时候，转换的 `SQL` 语句。

migrations中的迁移版本和数据库中的迁移版本对不上怎么办？

1. 找到哪里不一致，然后使用 `python manage.py --fake [版本名字]`，将这个版本标记为已经映射。
2. 删除指定 `app` 下 `migrations` 和数据库表 `django_migrations` 中和这个 `app` 相关的版本号，然后将模型中的字段和数据库中的字段保持一致，再使用命令 `python manage.py makemigrations` 重新生成一个初始化的迁移脚本，之后再使用命令 `python manage.py makemigrations --fake-initial` 来将这个初始化的迁移脚本标记为已经映射。以后再修改就没有问题了。

更多关于迁移脚本的。请查看官方文档：<https://docs.djangoproject.com/en/2.0/topics/migrations/>

根据已有的表自动生成模型：

在实际开发中，有些时候可能数据库已经存在了。如果我们用 Django 来开发一个网站，读取的是之前已经存在的数据库中的数据。那么该如何将模型与数据库中的表映射呢？根据旧的数据库生成对应的 ORM 模型，需要以下几个步骤：

1. Django 给我们提供了一个 `inspectdb` 的命令，可以非常方便的将已经存在的表，自动的生成模型。想要使用 `inspectdb` 自动将表生成模型。首先需要在 `settings.py` 中配置好数据库相关信息。不然就找不到数据库。示例代码如下：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': "migrations_demo",
        'HOST': '127.0.0.1',
        'PORT': '3306',
        'USER': 'root',
        'PASSWORD': 'root'
    }
}
```

比如有以下表：













◦ article表：

对象 article_article @migrations_...						
新建 保存 另存为 添加字段 插入字段 删除字段 主键 上移 下移						
字段	索引	外键	触发器	选项	注释	SQL 预览
名	类型	长度	小数点	不是 null		
id	int	11	0	<input checked="" type="checkbox"/>		1
title	varchar	100	0	<input checked="" type="checkbox"/>		
content	longtext	0	0	<input type="checkbox"/>		
create_time	datetime	6	0	<input type="checkbox"/>		
author_id	int	11	0	<input type="checkbox"/>		













◦ tag表：

对象 article_tag @migrations_de...						
新建 保存 另存为 添加字段 插入字段 删除字段 主键 上移 下移						
字段	索引	外键	触发器	选项	注释	SQL 预览
名	类型	长度	小数点	不是 null		
id	int	11	0	<input checked="" type="checkbox"/>		1
name	varchar	100	0	<input checked="" type="checkbox"/>		

- article_tag表:

对象  article_article_tags @migrati...						
 新建  保存  另存为  添加字段  插入字段  删除字段  主键  上移  下移						
字段	索引	外键	触发器	选项	注释	SQL 预览
名	类型	长度	小数点	不是 null		
 d	int	11	0	<input checked="" type="checkbox"/>	 1	
article_id	int	11	0	<input checked="" type="checkbox"/>		
tag_id	int	11	0	<input checked="" type="checkbox"/>		

- front_user表:

对象  front_user_frontuser @migr...						
 新建  保存  另存为  添加字段  插入字段  删除字段  主键  上移  下移						
字段	索引	外键	触发器	选项	注释	SQL 预览
名	类型	长度	小数点	不是 null		
 d	int	11	0	<input checked="" type="checkbox"/>	 1	
username	varchar	100	0	<input checked="" type="checkbox"/>		
telephone	varchar	11	0	<input checked="" type="checkbox"/>		

那么通过 `python manage.py inspectdb`，就会将表转换为模型后的代码，显示在终端：

```
from django.db import models

class ArticleArticle(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField(blank=True, null=True)
    create_time = models.DateTimeField(blank=True, null=True)
    author = models.ForeignKey('FrontUserFrontuser', models.DO_NOTHING, blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'article_article'

class ArticleArticleTags(models.Model):
    article = models.ForeignKey(ArticleArticle, models.DO_NOTHING)
    tag = models.ForeignKey('ArticleTag', models.DO_NOTHING)

    class Meta:
        managed = False
        db_table = 'article_article_tags'
        unique_together = (('article', 'tag'),)

class ArticleTag(models.Model):
    name = models.CharField(max_length=100)

    class Meta:
        managed = False
```



```

        db_table = 'article_tag'

class FrontUserFrontuser(models.Model):
    username = models.CharField(max_length=100)
    telephone = models.CharField(max_length=11)

    class Meta:
        managed = False
        db_table = 'front_user_frontuser'

```

以上代码只是显示在终端。如果想要保存到文件中。那么可以使用 `>` 重定向输出到指定的文件。比如让他输出到 `models.py` 文件中。示例命令如下：

```
python manage.py inspectdb > models.py
```

以上的命令，只能在终端执行，不能在 `pycharm->Tools->Run manage.py Task...` 中使用。

如果只是想要转换一个表为模型。那么可以指定表的名字。示例命令如下：

```
python manage.py inspectdb article_article > models.py
```

2. 修正模型：新生成的 ORM 模型有些地方可能不太适合使用。比如模型的名字，表之间的关系等等。那么以下选项还需要重新配置一下：

- 模型名：自动生成的模型，是根据表的名字生成的，可能不是你想要的。这时候模型的名字你可以改成任何你想要的。
- 模型所属app：根据自己的需要，将相应的模型放在对应的app中。放在同一个app中也是没有任何问题的。只是不方便管理。
- 模型外键引用：将所有使用 `ForeignKey` 的地方，模型引用都改成字符串。这样不会产生模型顺序的问题。另外，如果引用的模型已经移动到其他的app中了，那么还要加上这个app的前缀。
- 让Django管理模型：将 `Meta` 下的 `managed=False` 删掉，如果保留这个，那么以后这个模型有任何的修改，使用 `migrate` 都不会映射到数据库中。
- 当有多对多的时候，应该也要修正模型。将中间表注视了，然后使用 `ManyToManyField` 来实现多对多。并且，使用 `ManyToManyField` 生成的中间表的名字可能和数据库中那个中间表的名字不一致，这时候肯定就不能正常连接了。那么可以通过 `db_table` 来指定中间表的名字。示例代码如下：

```

class Article(models.Model):
    title = models.CharField(max_length=100, blank=True, null=True)
    content = models.TextField(blank=True, null=True)
    author = models.ForeignKey('front.User', models.SET_NULL, blank=True, null=True)

```

```

)
# 使用ManyToManyField模型到表，生成的中间表的规则是： article_tags
# 但现在已经存在的表的名字叫做： article_tag
# 可以使用db_table，指定中间表的名字
tags = models.ManyToManyField("Tag",db_table='article_tag')

class Meta:
    db_table = 'article'

```

- 表名：切记不要修改表的名字。不然映射到数据库中，会发生找不到对应表的错误。
3. 执行命令 `python manage.py makemigrations` 生成初始化的迁移脚本。方便后面通过 `ORM` 来管理表。这时候还需要执行命令 `python manage.py migrate --fake-initial`，因为如果不使用 `--fake-initial`，那么会将迁移脚本会映射到数据库中。这时候迁移脚本会新创建表，而这个表之前是已经存在了的，所以肯定会报错。此时我们只要将这个 `0001-initial` 的状态修改为已经映射，而不真正执行映射，下次再 `migrate` 的时候，就会忽略他。
 4. 将 `Django` 的核心表映射到数据库中：`Django` 中还有一些核心的表也是需要创建的。不然有些功能是用不了的。比如 `auth` 相关表。如果这个数据库之前就是使用 `Django` 开发的，那么这些表就已经存在了。可以不用管了。如果之前这个数据库不是使用 `Django` 开发的，那么应该使用 `migrate` 命令将 `Django` 中的核心模型映射到数据库中。

ORM作业:

假设有以下 ORM 模型:

```
from django.db import models

class Student(models.Model):
    """学生表"""
    name = models.CharField(max_length=100)
    gender = models.SmallIntegerField()

    class Meta:
        db_table = 'student'

class Course(models.Model):
    """课程表"""
    name = models.CharField(max_length=100)
    teacher = models.ForeignKey("Teacher", on_delete=models.SET_NULL, null=True)
    class Meta:
        db_table = 'course'

class Score(models.Model):
    """分数表"""
    student = models.ForeignKey("Student", on_delete=models.CASCADE)
    course = models.ForeignKey("Course", on_delete=models.CASCADE)
    number = models.FloatField()

    class Meta:
        db_table = 'score'

class Teacher(models.Model):
    """老师表"""
    name = models.CharField(max_length=100)

    class Meta:
        db_table = 'teacher'
```

使用之前学到过的操作实现下面的查询操作:

1. 查询平均成绩大于60分的同学的id和平均成绩;
2. 查询所有同学的id、姓名、选课的数量、总成绩;
3. 查询姓“李”的老师的个数;
4. 查询没学过“李老师”课的同学的id、姓名;

5. 查询学过课程id为1和2的所有同学的id、姓名；
6. 查询学过“黄老师”所教的“所有课”的同学们的id、姓名；
7. 查询所有课程成绩小于60分的同学的id和姓名；
8. 查询没有学全所有课的同学的id、姓名；
9. 查询所有学生的姓名、平均分，并且按照平均分从高到低排序；
10. 查询各科成绩的最高和最低分，以如下形式显示：课程ID，课程名称，最高分，最低分；
11. 查询没门课程的平均成绩，按照平均成绩进行排序；
12. 统计总共有多少女生，多少男生；
13. 将“黄老师”的每一门课程都在原来的基础之上加5分；
14. 查询两门以上不及格的同学的id、姓名、以及不及格课程数；
15. 查询每门课的选课人数；

ORM作业参考答案：

1. 查询平均成绩大于60分的同学的id和平均成绩；

```
rows = Student.objects.annotate(avg=Avg("score__number")).filter(avg__gte=60).values("id", "avg")
for row in rows:
    print(row)
```

2. 查询所有同学的id、姓名、选课的数、总成绩；

```
rows = Student.objects.annotate(course_nums=Count("score__course"), total_score=Sum("score__number"))
.values("id", "name", "course_nums", "total_score")
for row in rows:
    print(row)
```

3. 查询姓“李”的老师的个数；

```
teacher_nums = Teacher.objects.filter(name__startswith="李").count()
print(teacher_nums)
```

4. 查询没学过“黄老师”课的同学的id、姓名；

```
rows = Student.objects.exclude(score__course__teacher__name="黄老师").values('id', 'name')
for row in rows:
    print(row)
```

5. 查询学过课程id为1和2的所有同学的id、姓名；

```
rows = Student.objects.filter(score__course__in=[1, 2]).distinct().values('id', 'name')
for row in rows:
    print(row)
```

6. 查询学过“黄老师”所教的所有课的同学的学号、姓名；

```
rows = Student.objects.annotate(nums=Count("score__course", filter=Q(score__course__teacher__name='黄老师')))
```

```
.filter(nums=Course.objects.filter(teacher__name='黄老师').count()).values('id','name')
for row in rows:
    print(row)
```

7. 查询所有课程成绩小于60分的同学的id和姓名;

```
students = Student.objects.exclude(score__number__gt=60)
for student in students:
    print(student)
```

8. 查询没有学全所有课的同学的id、姓名;

```
students = Student.objects.annotate(num=Count(F("score__course"))).filter(num__lt=
Course.objects.count()).values('id','name')
for student in students:
    print(student)
```

9. 查询所有学生的姓名、平均分, 并且按照平均分从高到低排序;

```
students = Student.objects.annotate(avg=Avg("score__number")).order_by("-avg").values('name','avg')
for student in students:
    print(student)
```

10. 查询各科成绩的最高和最低分, 以如下形式显示: 课程ID, 课程名称, 最高分, 最低分:

```
courses = Course.objects.annotate(min=Min("score__number"),max=Max("score__number")).values("id","name","min","max")
for course in courses:
    print(course)
```

11. 查询每门课程的平均成绩, 按照平均成绩进行排序;

```
courses = Course.objects.annotate(avg=Avg("score__number")).order_by('avg').values('id','name','avg')
for course in courses:
    print(course)
```

12. 统计总共有多少女生，多少男生；

```
rows = Student.objects.aggregate(male_num=Count("gender",filter=Q(gender=1)),female_num=Count("gender",filter=Q(gender=2)))  
print(rows)
```

13. 将“黄老师”的每一门课程都在原来的基础之上加5分；

```
rows = Score.objects.filter(course__teacher__name='黄老师').update(number=F("number")+5)  
print(rows)
```

14. 查询两门以上不及格的同学的id、姓名、以及不及格课程数；

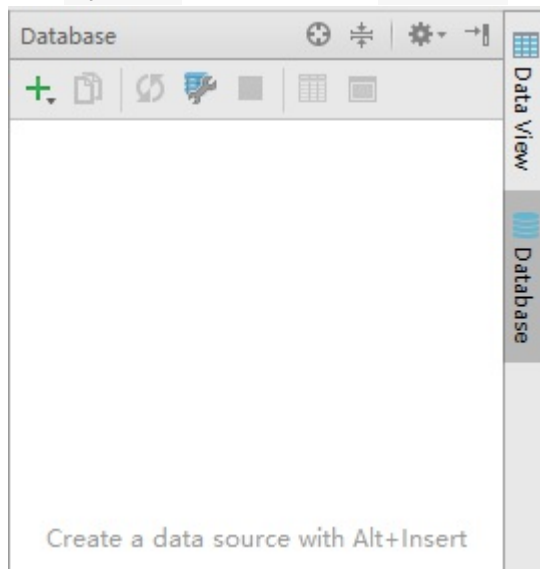
```
students = Student.objects.annotate(bad_count=Count("score__number",filter=Q(score__number__lt=60))).filter(bad_count__gte=2).values('id','name','bad_count')  
for student in students:  
    print(student)
```

15. 查询每门课的选课人数；

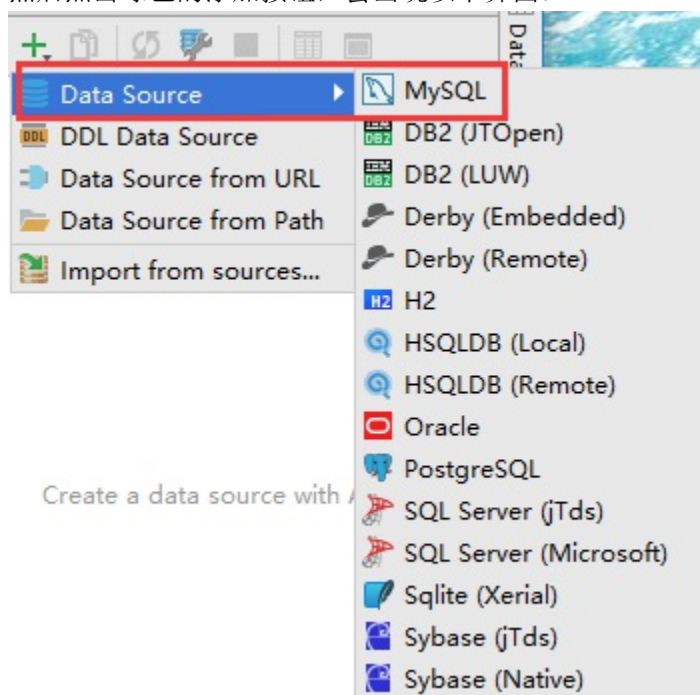
```
courses = Course.objects.annotate(student_nums=Count("score__student")).values('id','name','student_nums')  
for course in courses:  
    print(course)
```

Pycharm配置连接数据库

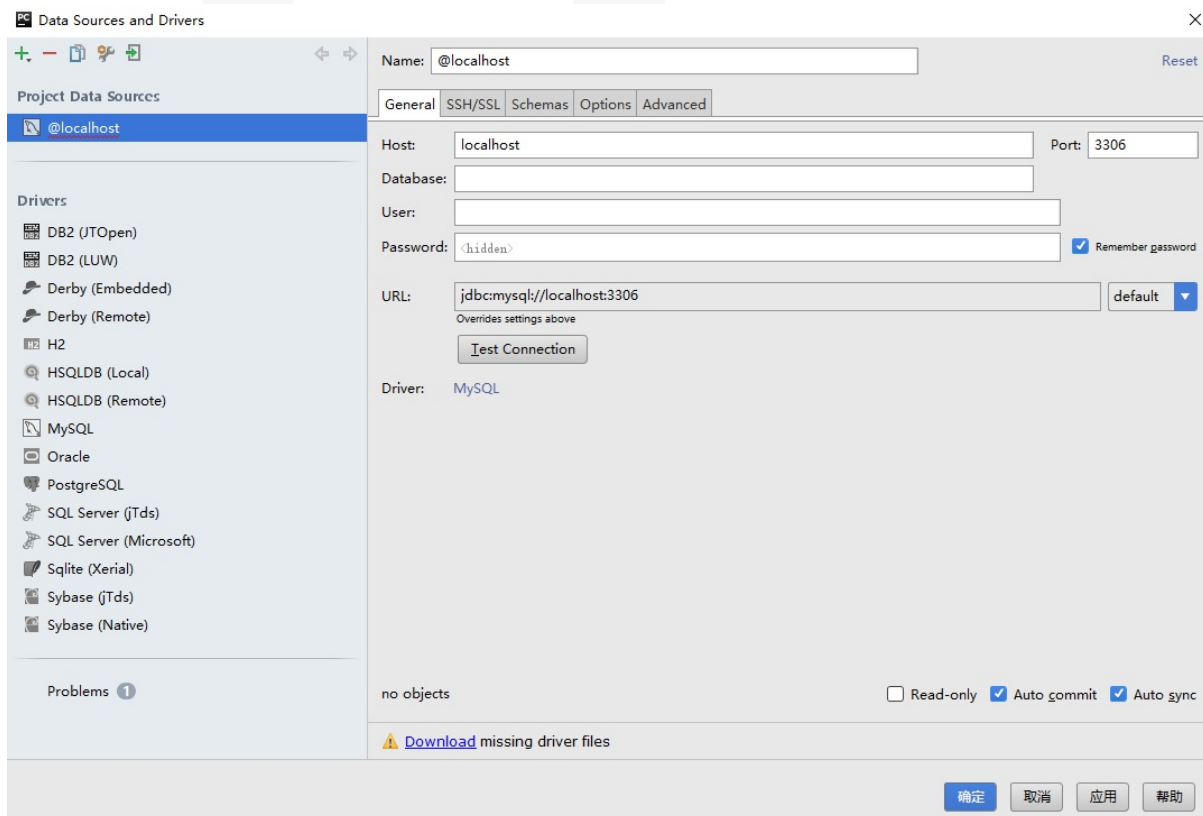
进入 pycharm 后，右边有一个 Database 的选项，点击这个选项会弹出以下界面：



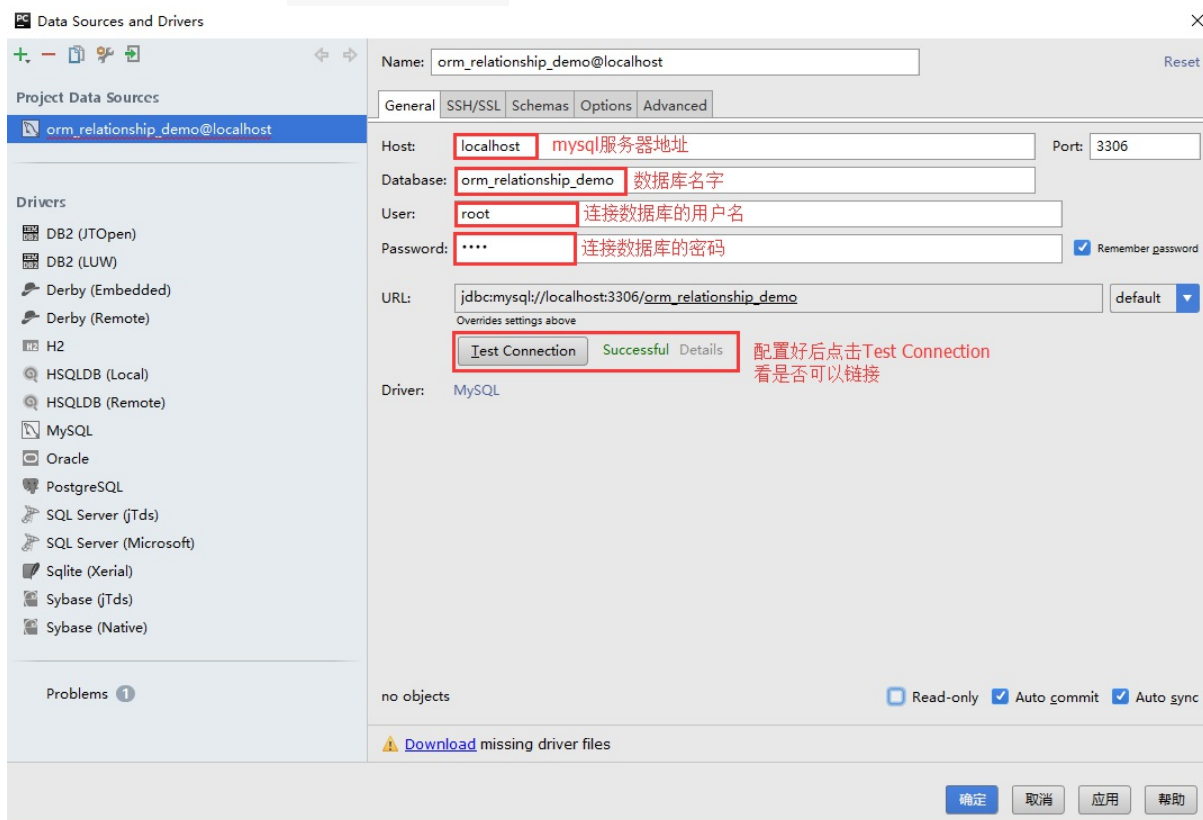
然后点击绿色的添加按钮，会出现以下界面：



这时候我们选择 **MySQL**，然后会弹出以下配置 **MySQL** 的对话框

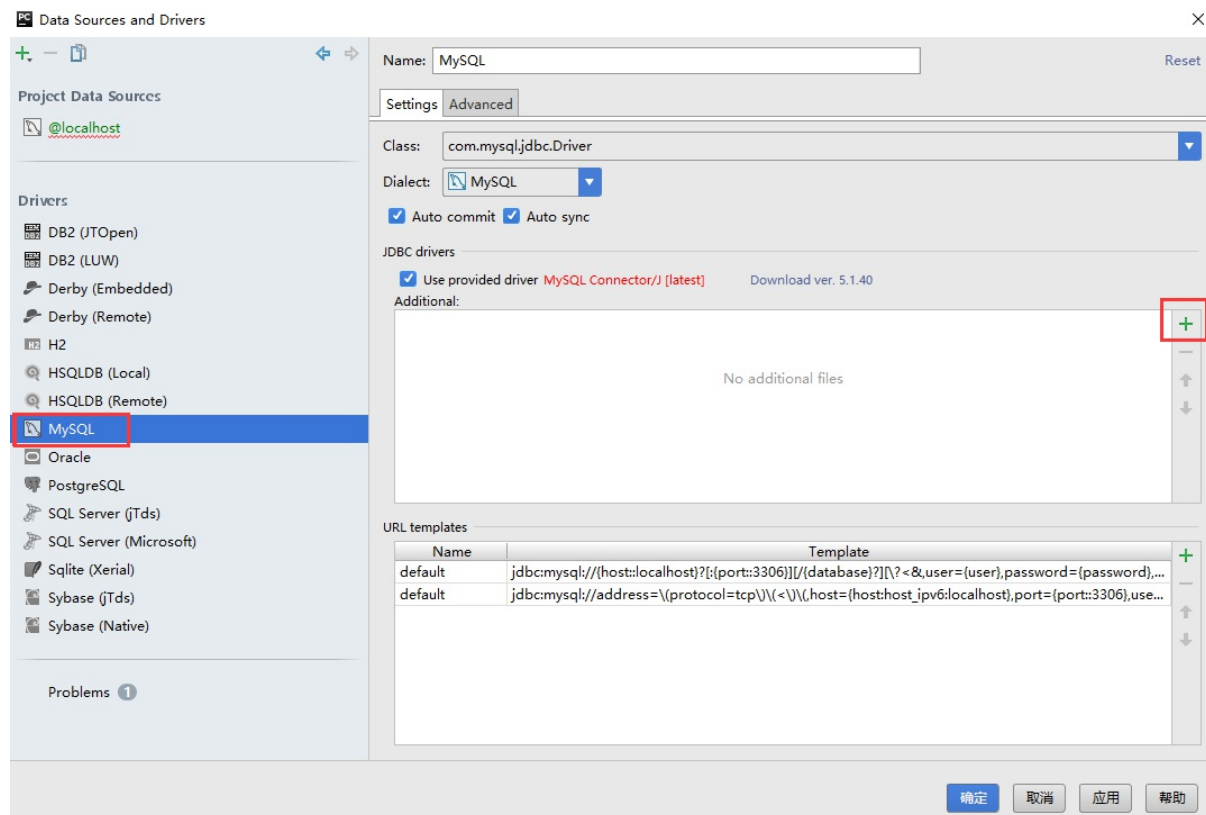


填入相关的信息。然后 **Test Connection** 测试成功后，点击确定即可！

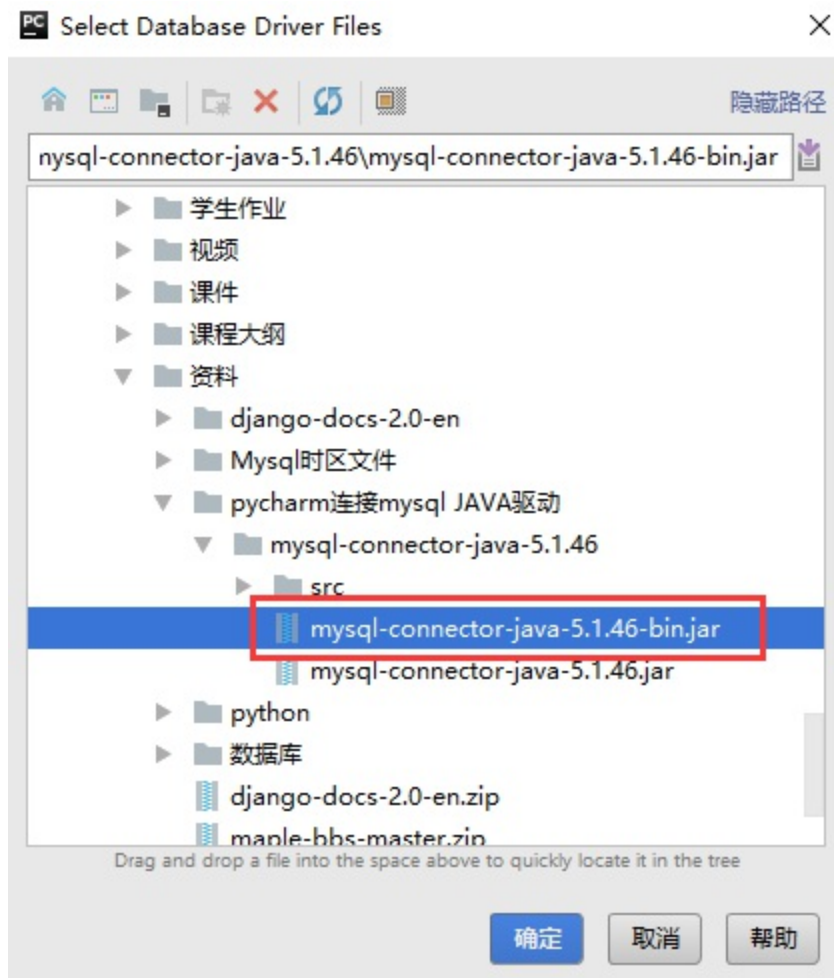


关于没有 **Java Connector Driver**:

Pycharm 是用 java 写的，连接 MySQL 数据库需要一个 driver 文件，从以下链接中下载 mysql-connector-java-5.1.46.zip : <https://dev.mysql.com/downloads/connector/j/> ，然后解压后，来到以下界面



然后点击右边的加号按钮，把刚刚下载的 `mysql-connector-java-5.1.46-bin.jar` 加载进来。



Django限制请求method

常用的请求method:

1. GET请求: GET请求一般用来向服务器索取数据, 但不会向服务器提交数据, 不会对服务器的状态进行更改。比如向服务器获取某篇文章的详情。
2. POST请求: POST请求一般是用来向服务器提交数据, 会对服务器的状态进行更改。比如提交一篇文章给服务器。

限制请求装饰器:

Django 内置的视图装饰器可以给视图提供一些限制。比如这个视图只能通过 GET 的 method 访问等。以下将介绍一些常用的内置视图装饰器。

1. `django.http.decorators.http.require_http_methods` : 这个装饰器需要传递一个允许访问的方法的列表。比如只能通过 GET 的方式访问。那么示例代码如下:

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET"])
def my_view(request):
    pass
```

2. `django.views.decorators.http.require_GET` : 这个装饰器相当于 `require_http_methods(['GET'])` 的简写形式, 只允许使用 GET 的 method 来访问视图。示例代码如下:

```
from django.views.decorators.http import require_GET

@require_GET
def my_view(request):
    pass
```

3. `django.views.decorators.http.require_POST` : 这个装饰器相当于 `require_http_methods(['POST'])` 的简写形式, 只允许使用 POST 的 method 来访问视图。示例代码如下:

```
from django.views.decorators.http import require_POST

@require_POST
def my_view(request):
    pass
```

-
4. `django.views.decorators.http.require_safe` : 这个装饰器相当于 `require_http_methods(['GET', 'HEAD'])` 的简写形式，只允许使用相对安全的方式来访问视图。因为 `GET` 和 `HEAD` 不会对服务器产生增删改的行为。因此是一种相对安全的请求方式。示例代码如下：

```
from django.views.decorators.http import require_safe

@require_safe
def my_view(request):
    pass
```

重定向

重定向分为永久性重定向和暂时性重定向，在页面上体现的操作就是浏览器会从一个页面自动跳转到另外一个页面。比如用户访问了一个需要权限的页面，但是该用户当前并没有登录，因此我们应该给他重定向到登录页面。

- 永久性重定向：**http**的状态码是**301**，多用于旧网址被废弃了要转到一个新的网址确保用户的访问，最经典的就是京东网站，你输入**www.jingdong.com**的时候，会被重定向到**www.jd.com**，因为**jingdong.com**这个网址已经被废弃了，被改成**jd.com**，所以这种情况下应该用永久重定向。
- 暂时性重定向：**http**的状态码是**302**，表示页面的暂时性跳转。比如访问一个需要权限的网址，如果当前用户没有登录，应该重定向到登录页面，这种情况下，应该用暂时性重定向。

在 Django 中，重定向是使用 `redirect(to, *args, permanent=False, **kwargs)` 来实现的。`to` 是一个 `url`，`permanent` 代表的是这个重定向是否是一个永久的重定向，默认是 `False`。关于重定向的使用。请看以下例子：

```
from django.shortcuts import reverse, redirect
def profile(request):
    if request.GET.get("username"):
        return HttpResponse("%s, 欢迎来到个人中心页面！")
    else:
        return redirect(reverse("user:login"))
```

WSGIRequest对象

Django在接收到http请求之后，会根据http请求携带的参数以及报文信息创建一个 `WSGIRequest` 对象，并且作为视图函数第一个参数传给视图函数。也就是我们经常看到的 `request` 参数。在这个对象上我们可以找到客户端上传上来的所有信息。这个对象的完整路径是 `django.core.handlers.wsgi.WSGIRequest`。

WSGIRequest对象常用属性和方法：

WSGIRequest对象常用属性：

`WSGIRequest` 对象上大部分的属性都是只读的。因为这些属性是从客户端上传上来的，没必要做任何的修改。以下将对一些常用的属性进行讲解：

1. `path`：请求服务器的完整“路径”，但不包含域名和参数。比如 `http://www.baidu.com/xxx/yyy/`，那么 `path` 就是 `/xxx/yyy/`。
2. `method`：代表当前请求的 http 方法。比如是 `GET` 还是 `POST`。
3. `GET`：一个 `django.http.request.QueryDict` 对象。操作起来类似于字典。这个属性中包含了所有以 `?xxx=xxx` 的方式上传上来的参数。
4. `POST`：也是一个 `django.http.request.QueryDict` 对象。这个属性中包含了所有以 `POST` 方式上传上来的参数。
5. `FILES`：也是一个 `django.http.request.QueryDict` 对象。这个属性中包含了所有上传的文件。
6. `COOKIES`：一个标准的Python字典，包含所有的 `cookie`，键值对都是字符串类型。
7. `session`：一个类似于字典的对象。用来操作服务器的 `session`。
8. `META`：存储的客户端发送上来的所有 `header` 信息。
9. `CONTENT_LENGTH`：请求的正文的长度（是一个字符串）。
10. `CONTENT_TYPE`：请求的正文的MIME类型。
11. `HTTP_ACCEPT`：响应可接收的Content-Type。
12. `HTTP_ACCEPT_ENCODING`：响应可接收的编码。
13. `HTTP_ACCEPT_LANGUAGE`：响应可接收的语言。
14. `HTTP_HOST`：客户端发送的HOST值。
15. `HTTP_REFERER`：在访问这个页面上一个页面的url。
16. `QUERY_STRING`：单个字符串形式的查询字符串（未解析过的形式）。
17. `REMOTE_ADDR`：客户端的IP地址。如果服务器使用了 `nginx` 做反向代理或者负载均衡，那么这个值返回的是 `127.0.0.1`，这时候可以使用 `HTTP_X_FORWARDED_FOR` 来获取，所以获取 `ip` 地址的代码片段如下：

```
if request.META.has_key('HTTP_X_FORWARDED_FOR'):
    ip = request.META['HTTP_X_FORWARDED_FOR']
```

```
else:
    ip = request.META['REMOTE_ADDR']
```

18. `REMOTE_HOST` : 客户端的主机名。
19. `REQUEST_METHOD` : 请求方法。一个字符串类似于 `GET` 或者 `POST` 。
20. `SERVER_NAME` : 服务器域名。
21. `SERVER_PORT` : 服务器端口号，是一个字符串类型。

WSGIRequest对象常用方法:

1. `is_secure()` : 是否是采用 `https` 协议。
2. `is_ajax()` : 是否采用 `ajax` 发送的请求。原理就是判断请求头中是否存在 `X-Requested-With:XMLHttpRequest` 。
3. `get_host()` : 服务器的域名。如果在访问的时候还有端口号，那么会加上端口号。比如 `www.baidu.com:9000` 。
4. `get_full_path()` : 返回完整的`path`。如果有查询字符串，还会加上查询字符串。比如 `/music/bands/?print=True` 。
5. `get_raw_uri()` : 获取请求的完整 `url` 。

QueryDict对象:

我们平时用的 `request.GET` 和 `request.POST` 都是 `QueryDict` 对象，这个对象继承自 `dict` ，因此用法跟 `dict` 相差无几。其中用得比较多的是 `get` 方法和 `getlist` 方法。

1. `get` 方法: 用来获取指定 `key` 的值，如果没有这个 `key` ，那么会返回 `None` 。
2. `getlist` 方法: 如果浏览器上传上来的 `key` 对应的值有多个，那么就需要通过这个方法获取。

HttpResponse对象

Django服务器接收到客户端发送过来的请求后，会将提交上来的这些数据封装成一个 `HttpRequest` 对象传给视图函数。那么视图函数在处理完相关的逻辑后，也需要返回一个响应给浏览器。而这个响应，我们必须返回 `HttpResponseBase` 或者他的子类的对象。而 `HttpResponse` 则是 `HttpResponseBase` 用得最多的子类。那么接下来就来介绍一下 `HttpResponse` 及其子类。

常用属性：

1. `content`: 返回的内容。
2. `status_code`: 返回的HTTP响应状态码。
3. `content_type`: 返回的数据的MIME类型，默认为 `text/html`。浏览器会根据这个属性，来显示数据。如果是 `text/html`，那么就会解析这个字符串，如果 `text/plain`，那么就会显示一个纯文本。常用的 `Content-Type` 如下：
 - `text/html`（默认的，html文件）
 - `text/plain`（纯文本）
 - `text/css`（css文件）
 - `text/javascript`（js文件）
 - `multipart/form-data`（文件提交）
 - `application/json`（json传输）
 - `application/xml`（xml文件）
4. 设置请求头：`response['X-Access-Token'] = 'xxxx'`。

常用方法：

1. `set_cookie`: 用来设置 `cookie` 信息。后面讲到授权的时候会着重讲到。
2. `delete_cookie`: 用来删除 `cookie` 信息。
3. `write`: `HttpResponse` 是一个类似于文件的对象，可以用来写入数据到数据体（`content`）中。

JsonResponse类：

用来对象 `dump` 成 `json` 字符串，然后返回将 `json` 字符串封装成 `Response` 对象返回给浏览器。并且他的 `Content-Type` 是 `application/json`。示例代码如下：

```
from django.http import JsonResponse
def index(request):
    return JsonResponse({"username": "zhiliao", "age": 18})
```

默认情况下 `JsonResponse` 只能对字典进行 `dump`，如果想要对非字典的数据进行 `dump`，那么需要给 `JsonResponse` 传递一个 `safe=False` 参数。示例代码如下：

```
from django.http import JsonResponse
def index(request):
    persons = ['张三', '李四', '王五']
    return JsonResponse(persons)
```

以上代码会报错，应该在使用 `HttpResponse` 的时候，传入一个 `safe=False` 参数，示例代码如下：

```
return JsonResponse(persons, safe=False)
```

生成CSV文件：

有时候我们做的网站，需要将一些数据，生成有一个 csv 文件给浏览器，并且是作为附件的形式下载下来。以下将讲解如何生成 csv 文件。

生成小的CSV文件：

这里将用一个生成小的 csv 文件为例，来把生成 csv 文件的技术要点讲到位。我们用 Python 内置的 csv 模块来处理 csv 文件，并且使用 HttpResponse 来将 csv 文件返回回去。示例代码如下：

```
import csv
from django.http import HttpResponse

def csv_view(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['username', 'age', 'height', 'weight'])
    writer.writerow(['zhiliao', '18', '180', '110'])

    return response
```

这里再来对每个部分的代码进行解释：

1. 我们在初始化 HttpResponse 的时候，指定了 Content-Type 为 text/csv，这将告诉浏览器，这是一个 csv 格式的文件而不是一个 HTML 格式的文件，如果用默认值，默认值就是 html，那么浏览器将把 csv 格式的文件按照 html 格式输出，这肯定不是我们想要的。
2. 第二个我们还在 response 中添加一个 Content-Disposition 头，这个东西是用来告诉浏览器该如何处理这个文件，我们给这个头的值设置为 attachment；，那么浏览器将不会对这个文件进行显示，而是作为附件的形式下载，第二个 filename="somefilename.csv" 是用来指定这个 csv 文件的名字。
3. 我们使用 csv 模块的 writer 方法，将相应的数据写入到 response 中。

将 csv 文件定义成模板：

我们还可以将 csv 格式的文件定义成模板，然后使用 Django 内置的模板系统，并给这个模板传入一个 Context 对象，这样模板系统就会根据传入的 Context 对象，生成具体的 csv 文件。示例代码如下：

模板文件：

```
{% for row in data %}"{{ row.0|addslashes }}" , "{{ row.1|addslashes }}" , "{{ row.2|addslashes }}" , "{{ row.3|addslashes }}" , "{{ row.4|addslashes }}"
{% endfor %}
```

视图函数:

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    response.write(t.render({"data": csv_data}))
    return response
```

生成大的**CSV**文件:

以上的例子是生成的一个小的 csv 文件, 如果想要生成大型的 csv 文件, 那么以上方式将有可能会发生超时的情况 (服务器要生成一个大型 csv 文件, 需要的时间可能会超过浏览器默认的超时时间)。这时候我们可以借助另外一个类, 叫做 `StreamingHttpResponse` 对象, 这个对象是将响应的数据作为一个流返回给客户端, 而不是作为一个整体返回。示例代码如下:

```
class Echo:
    """
    定义一个可以执行写操作的类, 以后调用csv.writer的时候, 就会执行这个方法
    """
    def write(self, value):
        return value

def large_csv(request):
    rows = ("Row {}".format(idx), str(idx)) for idx in range(655360)
    pseudo_buffer = Echo()
    writer = csv.writer(pseudo_buffer)
    response = StreamingHttpResponse((writer.writerow(row) for row in rows), content_type="text/csv")
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'
    return response
```

这里我们构建了一个非常大的数据集 `rows`，并且将其变成一个迭代器。然后因为 `StreamingHttpResponse` 的第一个参数只能是一个生成器，因此我们使用圆括号 (`writer.writerow(row) for row in rows`)，并且因为我们要写的文件是 `csv` 格式的文件，因此需要调用 `writer.writerow` 将 `row` 变成一个 `csv` 格式的字符串。而调用 `writer.writerow` 又需要一个中间的容器，因此这里我们定义了一个非常简单的类 `Echo`，这个类只实现一个 `write` 方法，以后在执行 `csv.writer(pseudo_buffer)` 的时候，就会调用 `Echo.writer` 方法。

注意：`StreamingHttpResponse` 会启动一个进程来和客户端保持长连接，所以会很消耗资源。所以如果不是特殊要求，尽量少用这种方法。

关于 `StreamingHttpResponse`:

这个类是专门用来处理流数据的。使得在处理一些大型文件的时候，不会因为服务器处理时间过长而到时连接超时。这个类不是继承自 `HttpResponse`，并且跟 `HttpResponse` 对比有以下几点区别：

1. 这个类没有属性 `content`，相反是 `streaming_content`。
2. 这个类的 `streaming_content` 必须是一个可以迭代的对象。
3. 这个类没有 `write` 方法，如果给这个类的对象写入数据将会报错。

注意：`StreamingHttpResponse` 会启动一个进程来和客户端保持长连接，所以会很消耗资源。所以如果不是特殊要求，尽量少用这种方法。

类视图

在写视图的时候， Django 除了使用函数作为视图，也可以使用类作为视图。使用类视图可以使用类的一些特性，比如继承等。

View:

django.views.generic.base.View 是主要的类视图，所有的类视图都是继承自他。如果我们写自己的类视图，也可以继承自他。然后再根据当前请求的 `method`，来实现不同的方法。比如这个视图只能使用 `get` 的方式来请求，那么就可以在这个类中定义 `get(self,request,*args,**kwargs)` 方法。以此类推，如果只需要实现 `post` 方法，那么就只需要在类中实现 `post(self,request,*args,**kwargs)`。示例代码如下：

```
from django.views import View
class BookDetailView(View):
    def get(self,request,*args,**kwargs):
        return render(request,'detail.html')
```

类视图写完后，还应该在 `urls.py` 中进行映射，映射的时候就需要调用 `View` 的类方法 `as_view()` 来进行转换。示例代码如下：

```
urlpatterns = [
    path("detail/<book_id>/",views.BookDetailView.as_view(),name='detail')
]
```

除了 `get` 方法， `View` 还支持以下方法 `['get','post','put','patch','delete','head','options','trace']`。

如果用户访问了 `View` 中没有定义的方法。比如你的类视图只支持 `get` 方法，而出现了 `post` 方法，那么就会把这个请求转发给 `http_method_not_allowed(request,*args,**kwargs)`。示例代码如下：

```
class AddBookView(View):
    def post(self,request,*args,**kwargs):
        return HttpResponse("书籍添加成功！")

    def http_method_not_allowed(self, request, *args, **kwargs):
        return HttpResponse("您当前采用的method是: %s, 本视图只支持使用post请求！" % request.method)
```

`urls.py` 中的映射如下：

```
path("addbook/", views.AddBookView.as_view(), name='add_book')
```

如果你在浏览器中访问 `addbook/`，因为浏览器访问采用的是 `get` 方法，而 `addbook` 只支持 `post` 方法，因此以上视图会返回您当前采用的 `method` 是：`GET`，本视图只支持使用 `post` 请求！。

其实不管是 `get` 请求还是 `post` 请求，都会走 `dispatch(request,*args,**kwargs)` 方法，所以如果实现这个方法，将能够对所有请求都处理到。

TemplateView:

django.views.generic.base.TemplateView，这个类视图是专门用来返回模版的。在这个类中，有两个属性是经常需要用到的，一个是 `template_name`，这个属性是用来存储模版的路径，`TemplateView` 会自动的渲染这个变量指向的模版。另外一个 `get_context_data`，这个方法是用来返回上下文数据的，也就是在给模版传的参数的。示例代码如下：

```
from django.views.generic.base import TemplateView

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['username'] = "黄勇"
        return context
```

在 `urls.py` 中的映射代码如下：

```
from django.urls import path

from myapp.views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

如果在模版中不需要传递任何参数，那么可以直接只在 `urls.py` 中使用 `TemplateView` 来渲染模版。示例代码如下：

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
```

```
path('about/', TemplateView.as_view(template_name="about.html")),
]
```

ListView:

在网站开发中，经常会出现需要列出某个表中的一些数据作为列表展示出来。比如文章列表，图书列表等等。在 Django 中可以使用 `ListView` 来帮我们快速实现这种需求。示例代码如下：

```
class ArticleListView(ListView):
    model = Article
    template_name = 'article_list.html'
    paginate_by = 10
    context_object_name = 'articles'
    ordering = 'create_time'
    page_kwarg = 'page'

    def get_context_data(self, **kwargs):
        context = super(ArticleListView, self).get_context_data(**kwargs)
        print(context)
        return context

    def get_queryset(self):
        return Article.objects.filter(id__lte=89)
```

对以上代码进行解释：

1. 首先 `ArticleListView` 是继承自 `ListView` 。
2. `model`：重写 `model` 类属性，指定这个列表是给哪个模型的。
3. `template_name`：指定这个列表的模板。
4. `paginate_by`：指定这个列表一页中展示多少条数据。
5. `context_object_name`：指定这个列表模型在模板中的参数名称。
6. `ordering`：指定这个列表的排序方式。
7. `page_kwarg`：获取第几页的数据的参数名称。默认是 `page` 。
8. `get_context_data`：获取上下文的数据。
9. `get_queryset`：如果你提取数据的时候，并不是要把所有数据都返回，那么你可以重写这个方法。将一些不需要展示的数据给过滤掉。

Paginator和Page类:

`Paginator` 和 `Page` 类都是用来做分页的。他们在 Django 中的路径为 `django.core.paginator.Paginator` 和 `django.core.paginator.Page` 。以下对这两个类的常用属性和方法做解释：

Paginator常用属性和方法:

1. `count` : 总共有多少条数据。
2. `num_pages` : 总共有多少页。
3. `page_range` : 页面的区间。比如有三页, 那么就 `range(1,4)` 。

Page常用属性和方法:

1. `has_next` : 是否还有下一页。
2. `has_previous` : 是否还有上一页。
3. `next_page_number` : 下一页的页码。
4. `previous_page_number` : 上一页的页码。
5. `number` : 当前页。
6. `start_index` : 当前这一页的第一条数据的索引值。
7. `end_index` : 当前这一页的最后一条数据的索引值。

给类视图添加装饰器:

在开发中, 有时候需要给一些视图添加装饰器。如果用函数视图那么非常简单, 只要在函数的上面写上装饰器就可以了。但是如果想要给类添加装饰器, 那么可以通过以下两种方式来实现:

装饰dispatch方法:

```
from django.utils.decorators import method_decorator

def login_required(func):
    def wrapper(request, *args, **kwargs):
        if request.GET.get("username"):
            return func(request, *args, **kwargs)
        else:
            return redirect(reverse('index'))
    return wrapper

class IndexView(View):
    def get(self, request, *args, **kwargs):
        return HttpResponse("index")

    @method_decorator(login_required)
    def dispatch(self, request, *args, **kwargs):
        super(IndexView, self).dispatch(request, *args, **kwargs)
```

直接装饰在整个类上:

```

from django.utils.decorators import method_decorator
def login_required(func):
    def wrapper(request,*args,**kwargs):
        if request.GET.get("username"):
            return func(request,*args,**kwargs)
        else:
            return redirect(reverse('login'))
    return wrapper

@method_decorator(login_required,name='dispatch')
class IndexView(View):
    def get(self,request,*args,**kwargs):
        return HttpResponse("index")

    def dispatch(self, request, *args, **kwargs):
        super(IndexView, self).dispatch(request,*args,**kwargs)

```

错误处理

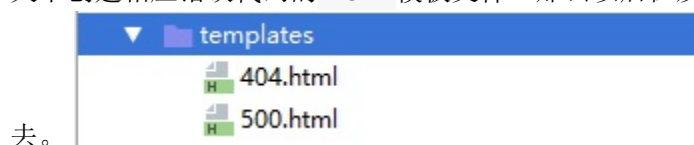
在一些网站开发中。经常会需要捕获一些错误，然后将这些错误返回比较优美的界面，或者是将这个错误的请求做一些日志保存。那么我们本节就来讲讲如何实现。

常用的错误码：

- 404 : 服务器没有指定的url。
- 403 : 没有权限访问相关的数据。
- 405 : 请求的 method 错误。
- 400 : bad request ， 请求的参数错误。
- 500 : 服务器内部错误，一般是代码出bug了。
- 502 : 一般部署的时候见得比较多，一般是 nginx 启动了，然后 uwsgi 有问题。

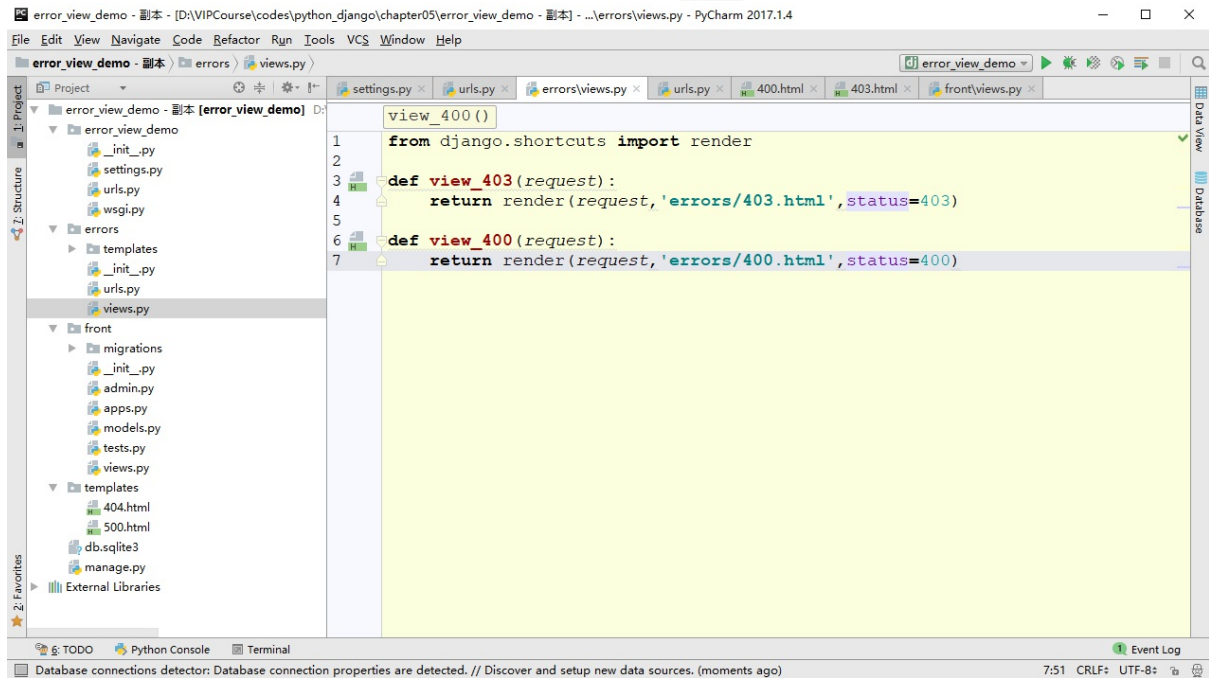
自定义错误模板：

在碰到比如 404 ， 500 错误的时候，想要返回自己定义的模板。那么可以直接在 templates 文件夹下创建相应错误代码的 html 模板文件。那么以后在发生相应错误后，会将指定的模板返回回



错误处理的解决方案：

对于 404 和 500 这种自动抛出的错误。我们可以直接在 `templates` 文件夹下新建相应错误代码的模板文件。而对于其他的错误，我们可以专门定义一个 `app`，用来处理这些错误。



分页

在 Django 中实现分页功能非常简单。因为 Django 已经内置了两个处理分类的类。分别是 `Paginator` 和 `Page`。`Paginator` 用来管理整个分类的一些属性，`Page` 用来管理当前这个分页的一些属性。通过这两个类，就可以轻松的实现分页效果。以下对这两个类进行讲解。

`Paginator` 类:

`Paginator` 是用来控制整个分页的逻辑的。比如总共有多少页，页码区间等等。都可以从他上面来获取。

创建 `Paginator` 对象:

`class Paginator(object_list, per_page, orphans=0, allow_empty_first_page=True)`，其中的参数解释如下：

1. `object_list`：列表，元组，`QuerySet` 或者是任何可以做切片操作的对象。会将这个里面的对象进行分页。
2. `per_page`：分页中，一页展示多少条数据。
3. `orphans`：用来控制最后一页元素的个人如果少于 `orphans` 指定的个数的时候，就会将多余的添加到上一页中。
4. `allow_empty_first_page`：如果 `object_list` 没有任何数据，并且这个参数设置为 `True`，那么就会抛出 `EmptyPage` 异常。

常用属性和方法:

1. `Paginator.page(number)`：获取 `number` 这页的 `Page` 对象。
2. `count`：传进来的 `object_list` 总共的数量。
3. `num_pages`：总共的页数。
4. `page_range`：页码的列表。比如 `[1,2,3,4]`。

`Page` 类:

常用属性和方法:

1. `has_next()`：是否还有下一页。
2. `has_previous()`：是否还有上一页。
3. `next_page_number()`：下一页的页码。
4. `previous_page_number()`：上一页的页码。

5. `object_list` : 在当前这页上的对象列表。
6. `number` : 当前的页码。
7. `paginator` : 获取 `Paginator` 对象。

表单：

HTML中的表单：

单纯从前端的 html 来说，表单是用来提交数据给服务器的，不管后台的服务器用的是 Django 还是 PHP 语言还是其他语言。只要把 input 标签放在 form 标签中，然后再添加一个提交按钮，那么以后点击提交按钮，就可以将 input 标签中对应的值提交给服务器了。

Django中的表单：

Django 中的表单丰富了传统的 HTML 语言中的表单。在 Django 中的表单，主要做以下两件事：

1. 渲染表单模板。
2. 表单验证数据是否合法。

Django中表单使用流程：

在讲解 Django 表单的具体每部分的细节之前。我们首先先来看下整体的使用流程。这里以一个做一个留言板为例。首先我们在后台服务器定义一个表单类，继承自 `django.forms.Form`。示例代码如下：

```
# forms.py
class MessageBoardForm(forms.Form):
    title = forms.CharField(max_length=3, label='标题', min_length=2, error_messages={"min_length": '标题字段不符合要求! '})
    content = forms.CharField(widget=forms.Textarea, label='内容')
    email = forms.EmailField(label='邮箱')
    reply = forms.BooleanField(required=False, label='回复')
```

然后在视图中，根据是 GET 还是 POST 请求来做相应的操作。如果是 GET 请求，那么返回一个空的表单，如果是 POST 请求，那么将提交上来的数据进行校验。示例代码如下：

```
# views.py
class IndexView(View):
    def get(self, request):
        form = MessageBoardForm()
        return render(request, 'index.html', {'form': form})

    def post(self, request):
        form = MessageBoardForm(request.POST)
        if form.is_valid():
            title = form.cleaned_data.get('title')
```

```

        content = form.cleaned_data.get('content')
        email = form.cleaned_data.get('email')
        reply = form.cleaned_data.get('reply')
        return HttpResponse('success')
    else:
        print(form.errors)
        return HttpResponse('fail')

```

在使用 GET 请求的时候，我们传了一个 `form` 给模板，那么以后模板就可以使用 `form` 来生成一个表单的 `html` 代码。在使用 POST 请求的时候，我们根据前端上传上来的数据，构建一个新的表单，这个表单是用来验证数据是否合法的，如果数据都验证通过了，那么我们可以通过 `cleaned_data` 来获取相应的数据。在模板中渲染表单的 HTML 代码如下：

```

<form action="" method="post">
    <table>

        <tr>
            <td></td>
            <td><input type="submit" value="提交"></td>
        </tr>
    </table>
</form>

```

我们在最外面给了一个 `form` 标签，然后在里面使用了 `table` 标签来进行美化，在使用 `form` 对象渲染的时候，使用的是 `table` 的方式，当然还可以使用 `ul` 的方式（`as_ul`），也可以使用 `p` 标签的方式（`as_p`），并且在后面我们还加上了一个提交按钮。这样就可以生成一个表单了

用表单验证数据

常用的Field:

使用 `Field` 可以是对数据验证的第一步。你期望这个提交上来的数据是什么类型，那么就使用什么类型的 `Field`。

CharField:

用来接收文本。

参数:

- `max_length`: 这个字段值的最大长度。
- `min_length`: 这个字段值的最小长度。
- `required`: 这个字段是否是必须的。默认是必须的。
- `error_messages`: 在某个条件验证失败的时候，给出错误信息。

EmailField:

用来接收邮件，会自动验证邮件是否合法。

错误信息的key: `required`、`invalid`。

FloatField:

用来接收浮点类型，并且如果验证通过后，会将这个字段的值转换为浮点类型。

参数:

- `max_value`: 最大的值。
- `min_value`: 最小的值。

错误信息的key: `required`、`invalid`、`max_value`、`min_value`。

IntegerField:

用来接收整形，并且验证通过后，会将这个字段的值转换为整形。

参数:

- `max_value`: 最大的值。
- `min_value`: 最小的值。

错误信息的key: `required`、`invalid`、`max_value`、`min_value`。

URLField:

用来接收 url 格式的字符串。

错误信息的key: `required`、`invalid`。

常用验证器:

在验证某个字段的时候,可以传递一个 `validators` 参数用来指定验证器,进一步对数据进行过滤。验证器有很多,但是很多验证器我们其实已经通过这个 `Field` 或者一些参数就可以指定了。比如 `EmailValidator`, 我们可以通过 `EmailField` 来指定, 比如 `MaxValueValidator`, 我们可以通过 `max_value` 参数来指定。以下是一些常用的验证器:

1. `MaxValueValidator`: 验证最大值。
2. `MinValueValidator`: 验证最小值。
3. `MinLengthValidator`: 验证最小长度。
4. `MaxLengthValidator`: 验证最大长度。
5. `EmailValidator`: 验证是否是邮箱格式。
6. `URLValidator`: 验证是否是 URL 格式。
7. `RegexValidator`: 如果还需要更加复杂的验证,那么我们可以通过正则表达式的验证器: `RegexValidator`。比如现在要验证手机号码是否合格,那么我们可以通过以下代码实现:

```
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678]\d{9}",message='请输入正确格式的手机号码!')])
```

自定义验证:

有时候对一个字段验证,不是一个长度,一个正则表达式能够写清楚的,还需要一些其他复杂的逻辑,那么我们可以对某个字段,进行自定义的验证。比如在注册的表单验证中,我们想要验证手机号码是否已经被注册过了,那么这时候就需要在数据库中进行判断才知道。对某个字段进行自定义的验证方式是,定义一个方法,这个方法的名字定义规则是: `clean_fieldname`。如果验证失败,那么就抛出一个验证错误。比如要验证用户表中手机号码之前是否在数据库中存在,那么可以通过以下代码实现:

```
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678]\d{9}",message='请输入正确格式的手机号码!')])

    def clean_telephone(self):
        telephone = self.cleaned_data.get('telephone')
        exists = User.objects.filter(telephone=telephone).exists()
        if exists:
            raise forms.ValidationError("手机号码已经存在!")
```

```
return telephone
```

以上是对某个字段进行验证，如果验证数据的时候，需要针对多个字段进行验证，那么可以重写 `clean` 方法。比如要在注册的时候，要判断提交的两个密码是否相等。那么可以使用以下代码来完成：

```
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678]\d{9}",
message='请输入正确格式的手机号码! ')])
    pwd1 = forms.CharField(max_length=12)
    pwd2 = forms.CharField(max_length=12)

    def clean(self):
        cleaned_data = super().clean()
        pwd1 = cleaned_data.get('pwd1')
        pwd2 = cleaned_data.get('pwd2')
        if pwd1 != pwd2:
            raise forms.ValidationError('两个密码不一致!')
```

提取错误信息：

如果验证失败了，那么有一些错误信息是我们需要传给前端的。这时候我们可以通过以下属性来获取：

1. `form.errors`：这个属性获取的错误信息是一个包含了 `html` 标签的错误信息。
2. `form.errors.get_json_data()`：这个方法获取到的是一个字典类型的错误信息。将某个字段的 `name` 作为 `key`，错误信息作为值的一个字典。
3. `form.as_json()`：这个方法是将 `form.get_json_data()` 返回的字典 `dump` 成 `json` 格式的字符串，方便进行传输。
4. 上述方法获取的字段错误值，都是一个比较复杂的数据。比如以下：

```
{'username': [{'message': 'Enter a valid URL.', 'code': 'invalid'}, {'message': 'Ensure this value has at most 4 characters (it has 22).', 'code': 'max_length'}]}
```

那么如果我只想把错误信息放在一个列表中，而不要再放在一个字典中。这时候我们可以定义一个方法，把这个数据重新整理一份。实例代码如下：

```
class MyForm(forms.Form):
    username = forms.URLField(max_length=4)

    def get_errors(self):
        errors = self.errors.get_json_data()
        new_errors = {}
        for key,message_dicts in errors.items():
```

```
messages = []
for message in message_dicts:
    messages.append(message['message'])
new_errors[key] = messages
return new_errors
```

这样就可以把某个字段所有的错误信息直接放在这个列表中。

ModelForm:

大家在写表单的时候，会发现表单中的 `Field` 和模型中的 `Field` 基本上是一模一样的，而且表单中需要验证的数据，也就是我们模型中需要保存的。那么这时候我们就可以将模型中的字段和表单中的字段进行绑定。

比如现在有个 `Article` 的模型。示例代码如下：

```
from django.db import models
from django.core import validators
class Article(models.Model):
    title = models.CharField(max_length=10,validators=[validators.MinLengthValidator(limit_value=3)])
    content = models.TextField()
    author = models.CharField(max_length=100)
    category = models.CharField(max_length=100)
    create_time = models.DateTimeField(auto_now_add=True)
```

那么在写表单的时候，就不需要把 `Article` 模型中所有的字段都一个个重复写一遍了。示例代码如下：

```
from django import forms
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = "__all__"
```

`MyForm` 是继承自 `forms.ModelForm`，然后在表单中定义了一个 `Meta` 类，在 `Meta` 类中指定了 `model=Article`，以及 `fields="__all__"`，这样就可以将 `Article` 模型中所有的字段都复制过来，进行验证。如果只想针对其中几个字段进行验证，那么可以给 `fields` 指定一个列表，将需要的字段写进去。比如只想验证 `title` 和 `content`，那么可以使用以下代码实现：

```
from django import forms
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = ['title','content']
```

如果要验证的字段比较多，只是除了少数几个字段不需要验证，那么可以使用 `exclude` 来代替 `fields`。比如我不想验证 `category`，那么示例代码如下：

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
```

```
exclude = ['category']
```

自定义错误消息:

使用 `ModelForm`，因为字段都不是在表单中定义的，而是在模型中定义的，因此一些错误消息无法在字段中定义。那么这时候可以在 `Meta` 类中，定义 `error_messages`，然后把相应的错误消息写到里面去。示例代码如下：

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ['category']
        error_messages = {
            'title': {
                'max_length': '最多不能超过10个字符!',
                'min_length': '最少不能少于3个字符!'
            },
            'content': {
                'required': '必须输入content!'
            }
        }
```

save方法:

`ModelForm` 还有 `save` 方法，可以在验证完成后直接调用 `save` 方法，就可以将这个数据保存到数据库中了。示例代码如下：

```
form = MyForm(request.POST)
if form.is_valid():
    form.save()
    return HttpResponse('succes')
else:
    print(form.get_errors())
    return HttpResponse('fail')
```

这个方法必须要在 `clean` 没有问题后才能使用，如果在 `clean` 之前使用，会抛出异常。另外，我们在调用 `save` 方法的时候，如果传入一个 `commit=False`，那么只会生成这个模型的对象，而不会把这个对象真正的插入到数据库中。比如表单上验证的字段没有包含模型中所有的字段，这时候就可以先创建对象，再根据填充其他字段，把所有字段的值都补充完成后，再保存到数据库中。示例代码如下：

```
form = MyForm(request.POST)
if form.is_valid():
```

```
    article = form.save(commit=False)
    article.category = 'Python'
    article.save()
    return HttpResponseRedirect('succes')
else:
    print(form.get_errors())
    return HttpResponseRedirect('fail')
```

文件上传：

文件上传是网站开发中非常常见的功能。这里详细讲述如何在 Django 中实现文件的上传功能。

前端HTML代码实现：

1. 在前端中，我们需要填入一个 `form` 标签，然后在这个 `form` 标签中指定 `enctype="multipart/form-data"`，不然就不能上传文件。
2. 在 `form` 标签中添加一个 `input` 标签，然后指定 `input` 标签的 `name`，以及 `type="file"`。

以上两步的示例代码如下：

```
<form action="" method="post" enctype="multipart/form-data">
    <input type="file" name="myfile">
</form>
```

后端的代码实现：

后端的主要工作是接收文件。然后存储文件。接收文件的方式跟接收 `POST` 的方式是一样的，只不过是通过 `FILES` 来实现。示例代码如下：

```
def save_file(file):
    with open('somefile.txt','wb') as fp:
        for chunk in file.chunks():
            fp.write(chunk)

def index(request):
    if request.method == 'GET':
        form = MyForm()
        return render(request,'index.html',{'form':form})
    else:
        myfile = request.FILES.get('myfile')
        save_file(myfile)
        return HttpResponse('success')
```

以上代码通过 `request.FILES` 接收到文件后，再写入到指定的地方。这样就可以完成一个文件的上传功能了。

使用模型来处理上传的文件：

在定义模型的时候，我们可以给存储文件的字段指定为 `FileField`，这个 `Field` 可以传递一个 `upload_to` 参数，用来指定上传上来的文件保存到哪里。比如我们让他保存到项目的 `files` 文件夹下，那么示例代码如下：

```
# models.py
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.FileField(upload_to="files")

# views.py
def index(request):
    if request.method == 'GET':
        return render(request, 'index.html')
    else:
        title = request.POST.get('title')
        content = request.POST.get('content')
        thumbnail = request.FILES.get('thumbnail')
        article = Article(title=title, content=content, thumbnail=thumbnail)
        article.save()
        return HttpResponse('success')
```

调用完 `article.save()` 方法，就会把文件保存到 `files` 下面，并且会将这个文件的路径存储到数据库中。

指定 `MEDIA_ROOT` 和 `MEDIA_URL`：

以上我们是使用了 `upload_to` 来指定上传的文件的目录。我们也可以指定 `MEDIA_ROOT`，就不需要在 `FileField` 中指定 `upload_to`，他会自动的将文件上传到 `MEDIA_ROOT` 的目录下。

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

然后我们可以在 `urls.py` 中添加 `MEDIA_ROOT` 目录下的访问路径。示例代码如下：

```
from django.urls import path
from front import views
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('', views.index),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

如果我们同时指定 `MEDIA_ROOT` 和 `upload_to`，那么会将文件上传到 `MEDIA_ROOT` 下的 `upload_to` 文件夹中。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.FileField(upload_to="%Y/%m/%d/")
```

限制上传的文件拓展名：

如果想要限制上传的文件的拓展名，那么我们就需要用到表单来进行限制。我们可以使用普通的 `Form` 表单，也可以使用 `ModelForm`，直接从模型中读取字段。示例代码如下：

```
# models.py
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.FileField(upload_to='%Y/%m/%d/', validators=[validators.FileExtensionValidator(['txt', 'pdf'])])

# forms.py
class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = "__all__"
```

上传图片：

上传图片跟上传普通文件是一样的。只不过是上传图片的时候 `Django` 会判断上传的文件是否是图片的格式（除了判断后缀名，还会判断是否是可用的图片）。如果不是，那么就会验证失败。我们首先来定义一个包含 `ImageField` 的模型。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.ImageField(upload_to="%Y/%m/%d/")
```

因为要验证是否是合格的图片，因此我们还需要用一个表单来进行验证。表单我们直接就使用 `ModelForm` 就可以了。示例代码如下：

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
```

```
fields = "__all__"
```

注意：使用**ImageField**，必须要先安装**Pillow**库：**pip install pillow**

cookie和session

1. **cookie**: 在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。`cookie` 的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（`cookie`）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的 `cookie` 数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。`cookie` 存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用 `cookie` 只能存储一些小量的数据。
2. **session**: `session`和`cookie`的作用有点类似，都是为了存储用户相关的信息。不同的是，`cookie` 是存储在本地浏览器，`session` 是一个思路、一个概念、一个服务器存储授权信息的解决方案，不同的服务器，不同的框架，不同的语言有不同的实现。虽然实现不一样，但是他们的目的都是服务器为了方便存储数据的。`session` 的出现，是为了解决 `cookie` 存储数据不安全的问题的。
3. **cookie和session使用**: `web` 开发发展至今，`cookie` 和 `session` 的使用已经出现了一些非常成熟的方案。在如今的市场或者企业里，一般有两种存储方式：
 - 存储在服务端：通过 `cookie` 存储一个 `sessionid`，然后具体的数据则是保存在 `session` 中。如果用户已经登录，则服务器会在 `cookie` 中保存一个 `sessionid`，下次再次请求的时候，会把该 `sessionid` 携带上来，服务器根据 `sessionid` 在 `session` 库中获取用户的 `session` 数据。就能知道该用户到底是谁，以及之前保存的一些状态信息。这种专业术语叫做 `server side session`。`Django` 把 `session` 信息默认存储到数据库中，当然也可以存储到其他地方，比如缓存中，文件系统中等。存储在服务器的数据会更加的安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，一些 `session` 信息还是绰绰有余的。
 - 将 `session` 数据加密，然后存储在 `cookie` 中。这种专业术语叫做 `client side session`。`flask` 框架默认采用的就是这种方式，但是也可以替换成其他形式。

在django中操作cookie和session:

操作cookie:

设置cookie:

设置 `cookie` 是设置值给浏览器的。因此我们需要通过 `response` 的对象来设置，设置 `cookie` 可以通过 `response.set_cookie` 来设置，这个方法的相关参数如下：

1. `key` : 这个 `cookie` 的 `key`。
2. `value` : 这个 `cookie` 的 `value`。
3. `max_age` : 最长的生命周期。单位是秒。

4. `expires` : 过期时间。跟 `max_age` 是类似的，只不过这个参数需要传递一个具体的日期，比如 `datetime` 或者是符合日期格式的字符串。如果同时设置了 `expires` 和 `max_age`，那么将会使用 `expires` 的值作为过期时间。
5. `path` : 对域名下哪个路径有效。默认是对域名下所有路径都有效。
6. `domain` : 针对哪个域名有效。默认是针对主域名下都有效，如果只要针对某个子域名才有效，那么可以设置这个属性。
7. `secure` : 是否是安全的，如果设置为 `True`，那么只能在 `https` 协议下才可用。
8. `httponly` : 默认是 `False`。如果为 `True`，那么在客户端不能通过 `JavaScript` 进行操作。

删除cookie:

通过 `delete_cookie` 即可删除 `cookie`。实际上删除 `cookie` 就是将指定的 `cookie` 的值设置为空的字符串，然后使用将他的过期时间设置为 `0`，也就是浏览器关闭后就过期。

获取cookie:

获取浏览器发送过来的 `cookie` 信息。可以通过 `request.COOKIEs` 来或者。这个对象是一个字典类型。比如获取所有的 `cookie`，那么示例代码如下：

```
cookies = request.COOKIEs
for cookie_key,cookie_value in cookies.items():
    print(cookie_key,cookie_value)
```

操作session:

`django` 中的 `session` 默认情况下是存储在服务器的数据库中的，在表中会根据 `sessionid` 来提取指定的 `session` 数据，然后再把这个 `sessionid` 放到 `cookie` 中发送给浏览器存储，浏览器下次在向服务器发送请求的时候会自动的把所有 `cookie` 信息都发送给服务器，服务器再从 `cookie` 中获取 `sessionid`，然后再从数据库中获取 `session` 数据。但是我们在操作 `session` 的时候，这些细节压根就不用管。我们只需要通过 `request.session` 即可操作。示例代码如下：

```
def index(request):
    request.session.get('username')
    return HttpResponse('index')
```

`session` 常用的方法如下：

1. `get` : 用来从 `session` 中获取指定值。
2. `pop` : 从 `session` 中删除一个值。
3. `keys` : 从 `session` 中获取所有的键。

4. `items` : 从 `session` 中获取所有的值。
5. `clear` : 清除当前这个用户的 `session` 数据。
6. `flush` : 删除 `session` 并且删除在浏览器中存储的 `session_id` , 一般在注销的时候用得比较多。
7. `set_expiry(value)` : 设置过期时间。
 - 整形: 代表秒数, 表示多少秒后过期。
 - `0` : 代表只要浏览器关闭, `session` 就会过期。
 - `None` : 会使用全局的 `session` 配置。在 `settings.py` 中可以设置 `SESSION_COOKIE_AGE` 来配置全局的过期时间。默认是 `1209600` 秒, 也就是2周的时间。
8. `clear_expired` : 清除过期的 `session` 。 `Django` 并不会清除过期的 `session` , 需要定期手动的清理, 或者是在终端, 使用命令行 `python manage.py clearsessions` 来清除过期的 `session` 。

修改session的存储机制:

默认情况下, `session` 数据是存储到数据库中的。当然也可以将 `session` 数据存储到其他地方。可以通过设置 `SESSION_ENGINE` 来更改 `session` 的存储位置, 这个可以配置为以下几种方案:

1. `django.contrib.sessions.backends.db` : 使用数据库。默认就是这种方案。
2. `django.contrib.sessions.backends.file` : 使用文件来存储session。
3. `django.contrib.sessions.backends.cache` : 使用缓存来存储session。想要将数据存储到缓存中, 前提是你必须要在 `settings.py` 中配置好 `CACHES` , 并且是需要使用 `Memcached` , 而不能使用纯内存作为缓存。
4. `django.contrib.sessions.backends.cached_db` : 在存储数据的时候, 会将数据先存到缓存中, 再存到数据库中。这样就可以保证万一缓存系统出现问题, `session`数据也不会丢失。在获取数据的时候, 会先从缓存中获取, 如果缓存中没有, 那么就会从数据库中获取。
5. `django.contrib.sessions.backends.signed_cookies` : 将 `session` 信息加密后存储到浏览器的 `cookie` 中。这种方式要注意安全, 建议设置 `SESSION_COOKIE_HTTPONLY=True` , 那么在浏览器中不能通过 `js` 来操作 `session` 数据, 并且还需要对 `settings.py` 中的 `SECRET_KEY` 进行保密, 因为一旦别人知道这个 `SECRET_KEY` , 那么就可以进行解密。另外还有就是在 `cookie` 中, 存储的数据不能超过 `4k` 。

上下文处理器

上下文处理器是可以返回一些数据，在全局模板中都可以使用。比如登录后的用户信息，在很多页面中都需要使用，那么我们可以放在上下文处理器中，就没有必要在每个视图函数中都返回这个对象。

在 `settings.TEMPLATES.OPTIONS.context_processors` 中，有许多内置的上下文处理器。这些上下文处理器的作用如下：

1. `django.template.context_processors.debug`：增加一个 `debug` 和 `sql_queries` 变量。在模板中可以通过他来查看到一些数据库查询。
2. `django.template.context_processors.request`：增加一个 `request` 变量。这个 `request` 变量也就是在视图函数的第一个参数。
3. `django.contrib.auth.context_processors.auth`：Django 有内置的用户系统，这个上下文处理器会增加一个 `user` 对象。
4. `django.contrib.messages.context_processors.messages`：增加一个 `messages` 变量。
5. `django.template.context_processors.media`：在模板中可以读取 `MEDIA_URL`。比如想要在模板中使用上传的文件，那么这时候就需要使用 `settings.py` 中设置的 `MEDIA_URL` 来拼接 `url`。示例代码如下：

```
<img src="" />
```

6. `django.template.context_processors.static`：在模板中可以使用 `STATIC_URL`。
7. `django.template.context_processors.csrf`：在模板中可以使用 `csrf_token` 变量来生成一个 `csrf token`。

自定义上下文处理器：

有时候我们想要返回自己的数据。那么这时候我们可以自定义上下文处理器。自定义上下文处理器的步骤如下：

1. 你可以根据这个上下文处理器是属于哪个 `app`，然后在这个 `app` 中创建一个文件专门用来存储上下文处理器。比如 `context_processors.py`。或者是你也可以专门创建一个 `Python`包，用来存储所有的上下文处理器。
2. 在你定义的上下文处理器文件中，定义一个函数，这个函数只有一个 `request` 参数。这个函数中处理完自己的逻辑后，把需要返回给模板的数据，通过字典的形式返回。如果不需要返回任何数据，那么也必须返回一个空的字典。示例代码如下：

```
def frontuser(request):
    userid = request.session.get("userid")
    userModel = models.FrontendUser.objects.filter(pk=userid).first()
    if userModel:
        return {'frontuser': userModel}
```

```
else:  
    return {}
```


中间件

中间件是在 `request` 和 `response` 处理过程中的一个插件。比如在 `request` 到达视图函数之前，我们可以使用中间件来做一些相关的事情，比如可以判断当前这个用户有没有登录，如果登录了，就绑定一个 `user` 对象到 `request` 上。也可以在 `response` 到达浏览器之前，做一些相关的处理，比如想要统一在 `response` 上设置一些 `cookie` 信息等。

自定义中间件：

中间件所处的位置没有规定。只要是放到项目当中即可。一般分为两种情况，如果中间件是属于某个 `app` 的，那么可以在这个 `app` 下面创建一个 `python` 文件用来存放这个中间件，也可以专门创建一个 `Python` 包，用来存放本项目的所有中间件。创建中间件有两种方式，一种是使用函数，一种是使用类，接下来对这两种方式做个介绍：

使用函数的中间件：

```
def simple_middleware(get_response):
    # 这个中间件初始化的代码

    def middleware(request):
        # request到达view的执行代码

        response = get_response(request)

        # response到达浏览器的执行代码

        return response

    return middleware
```

使用类的中间件：

```
class SimpleMiddleware(object):
    def __init__(self, get_response):
        self.get_response = get_response
        # 这个中间件初始化的代码
    def __call__(self, request):
        # request到达view之前执行的代码

        response = self.get_response(request)

        # response到达用户浏览器之前执行的代码
```

```
return response
```

在写完中间件后，还需要在 `settings.MIDDLEWARES` 中配置写好的中间件才可以使用。比如我们写了一个在 `request` 到达视图函数之前，判断这个用户是否登录，如果已经登录就绑定一个 `user` 对象到 `request` 上的中间件，这个中间件放在当前项目的 `middlewares.users` 下：

```
def user_middleware(get_response):
    # 这个中间件初始化的代码

    def middleware(request):
        # request到达view的执行代码
        userid = request.session.get("userid")
        userModel = FrontUser.objects.filter(pk=userid).first()
        if userModel:
            setattr(request, 'frontuser', userModel)

        response = get_response(request)

        # response到达浏览器的执行代码

        return response

    return middleware
```

那么就可以在 `settings.MIDDLEWARES` 下做以下配置：

```
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',
    'middlewares.users.user_middleware'
]
```

中间件的执行是有顺序的，他会根据在 `MIDDLEWARE` 中存放的顺序来执行。因此如果有些中间件是需要基于其他中间件的，那么就需要放在其他中间件的后面来执行。

Django内置的中间件：

1. `django.middleware.common.CommonMiddleware`：通用中间件。他的作用如下：
 - 限制 `settings.DISALLOWED_USER_AGENTS` 中指定的请求头来访问本网

站。 `DISALLOWED_USER_AGENT` 是一个正则表达式的列表。示例代码如下：

```
import re
DISALLOWED_USER_AGENTS = [
    re.compile(r'^\s$|^$'),
    re.compile(r'.*PhantomJS.*')
]
```

- 如果开发者在定义 `url` 的时候，最后有一个斜杠。但是用户在访问 `url` 的时候没有提交这个斜杠，那么 `CommonMiddleware` 会自动的重定向到加了斜杠的 `url` 上去。
- 2. `django.middleware.gzip.GZipMiddleware` : 将响应数据进行压缩。如果内容长度少于200个长度，那么就不会压缩。
- 3. `django.contrib.messages.middleware.MessageMiddleware` : 消息处理相关的中间件。
- 4. `django.middleware.security.SecurityMiddleware` : 做了一些安全处理的中间件。比如设置 `XSS` 防御的请求头，比如做了 `http` 协议转 `https` 协议的工作等。
- 5. `django.contrib.sessions.middleware.SessionMiddleware` : `session` 中间件。会给 `request` 添加一个处理好的 `session` 对象。
- 6. `django.contrib.auth.middleware.AuthenticationMiddleware` : 会给 `request` 添加一个 `user` 对象的中间件。
- 7. `django.middleware.csrf.CsrfViewMiddleware` : `CSRF` 保护的中间件。
- 8. `django.middleware.clickjacking.XFrameOptionsMiddleware` : 做了 `clickjacking` 攻击的保护。`clickjacking` 保护是攻击者在自己的病毒网站上，写一个诱惑用户点击的按钮，然后使用 `iframe` 的方式将受攻击的网站（比如银行网站）加载到自己的网站上去，并将其设置为透明的，用户就看不到，然后再把受攻击的网站（比如银行网站）的转账按钮定位到病毒网站的按钮上，这样用户在点击病毒网站上按钮的时候，实际上点击的是受攻击的网站（比如银行网站）上的按钮，从而实现了在不知不觉中给攻击者转账的功能。
- 9. 缓存中间件：用来缓存一些页面的。
 - `django.middleware.cache.UpdateCacheMiddleware` 。
 - `django.middleware.cache.FetchFromCacheMiddleware` 。

内置中间件放置的顺序：

1. `SecurityMiddleware` : 应该放到最前面。因为这个中间件并不需要依赖任何其他的中间件。如果你的网站同时支持 `http` 协议和 `https` 协议，并且你想让用户在使用 `http` 协议的时候重定向到 `https` 协议，那么就没有必要让他执行下面一大串中间件再重定向，这样效率更高。
2. `UpdateCacheMiddleware` : 应该在 `SessionMiddleware`, `GZipMiddleware`, `LocaleMiddleware` 之前。
3. `GZipMiddleware` 。
4. `ConditionalGetMiddleware` 。
5. `SessionMiddleware` 。
6. `LocaleMiddleware` 。
7. `CommonMiddleware` 。
8. `CsrfViewMiddleware` 。

9. `AuthenticationMiddleware` ◦
10. `MessageMiddleware` ◦
11. `FetchFromCacheMiddleware` ◦
12. `FlatpageFallbackMiddleware` ◦
13. `RedirectFallbackMiddleware` ◦

CSRF攻击:

CSRF攻击概述:

CSRF (Cross Site Request Forgery, 跨站域请求伪造) 是一种网络的攻击方式, 它在 2007 年曾被列为互联网 20 大安全隐患之一。其他安全隐患, 比如 SQL 脚本注入, 跨站域脚本攻击等在近年来已经逐渐为众人熟知, 很多网站也都针对他们进行了防御。然而, 对于大多数人来说, CSRF 却依然是一个陌生的概念。即便是大名鼎鼎的 Gmail, 在 2007 年底也存在着 CSRF 漏洞, 从而被黑客攻击而使 Gmail 的用户造成巨大的损失。

CSRF攻击原理:

网站是通过 cookie 来实现登录功能的。而 cookie 只要存在浏览器中, 那么浏览器在访问这个 cookie 的服务器的时候, 就会自动的携带 cookie 信息到服务器上去。那么这时候就存在一个漏洞了, 如果你访问了一个别有用心或病毒网站, 这个网站可以在网页源代码中插入js代码, 使用js代码给其他服务器发送请求 (比如ICBC的转账请求)。那么因为在发送请求的时候, 浏览器会自动的把 cookie 发送给对应的服务器, 这时候相应的服务器 (比如ICBC网站), 就不知道这个请求是伪造的, 就被欺骗过去了。从而达到在用户不知情的情况下, 给某个服务器发送了一个请求 (比如转账)。

防御CSRF攻击:

CSRF攻击的要点就是在向服务器发送请求的时候, 相应的 cookie 会自动的发送给对应的服务器。造成服务器不知道这个请求是用户发起的还是伪造的。这时候, 我们可以在用户每次访问有表单的页面的时候, 在网页源代码中加一个随机的字符串叫做 csrf_token, 在 cookie 中也加入一个相同值的 csrf_token 字符串。以后给服务器发送请求的时候, 必须在 body 中以及 cookie 中都携带 csrf_token, 服务器只有检测到 cookie 中的 csrf_token 和 body 中的 csrf_token 都相同, 才认为这个请求是正常的, 否则就是伪造的。那么黑客就没办法伪造请求了。在 Django 中, 如果想要防御 CSRF 攻击, 应该做两步工作。第一个是在 settings.MIDDLEWARE 中添加 CsrfMiddleware 中间件。第二个是在模版代码中添加一个 input 标签, 加载 csrf_token。示例代码如下:

- 服务器代码:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.middleware.gzip.GZipMiddleware',  
    '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'
]
```

- 模版代码:

```
<input type="hidden" name="csrfmiddlewaretoken" value="{ csrf_token }"/>
```

或者是直接使用 `csrf_token` 标签, 来自动生成一个带有 `csrf token` 的 `input` 标签:

```
{% csrf_token %}
```

使用ajax处理csrf防御:

如果用 `ajax` 来处理 `csrf` 防御, 那么需要手动的在 `form` 中添加 `csrfmiddlewaretoken`, 或者是在请求头中添加 `X-CSRFToken`。我们可以从返回的 `cookie` 中提取 `csrf token`, 再设置进去。示例代码如下:

```
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie !== '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) === (name + '=')) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}

var myajax = {
    'get': function (args) {
        args['method'] = 'get';
        this.ajax(args);
    },
    'post': function (args) {
        args['method'] = 'post';
        this._ajaxSetup();
        this.ajax(args);
    },
    'ajax': function (args) {
        $.ajax(args);
    }
}
```

```

    },
    '_ajaxSetup': function () {
        $.ajaxSetup({
            beforeSend: function(xhr, settings) {
                if (!/^(GET|HEAD|OPTIONS|TRACE)$/.test(settings.type) && !this.crossDomain) {
                    xhr.setRequestHeader("X-CSRFToken", getCookie('csrftoken'));
                }
            }
        });
    }
};

$(function () {
    $("#submit").click(function (event) {
        event.preventDefault();
        var email = $("input[name='email']").val();
        var money = $("input[name='money']").val();

        myajax.post({
            'url': '/transfer/',
            'data':{
                'email': email,
                'money': money
            },
            'success': function (data) {
                console.log(data);
            },
            'fail': function (error) {
                console.log(error);
            }
        });
    })
});

```

iframe相关知识:

1. `iframe` 可以加载嵌入别的域名下的网页。也就是说可以发送跨域请求。比如我可以在我自己的网页中加载百度的网站，示例代码如下：

```

<iframe src="http://www.baidu.com/">
</ifrmæ>

```

2. 因为 `iframe` 加载的是别的域名下的网页。根据[同源策略](#)，`js` 只能操作属于本域名下的代码，因此 `js` 不能操作通过 `iframe` 加载来的 DOM 元素。
3. 如果 `ifrmæ` 的 `src` 属性为空，那么就没有同源策略的限制，这时候我们就可以操作 `iframe` 下面的代码了。并且，如果 `src` 为空，那么我们可以在 `iframe` 中，给任何域名都

可以发送请求。

4. 直接在 `iframe` 中写 `html` 代码，浏览器是不会加载的。

XSS攻击

XSS（Cross Site Script）攻击又叫做跨站脚本攻击。他的原理是用户在使用具有 xss 漏洞的网站的时候，向这个网站提交一些恶意的代码，当用户在访问这个网站的某个页面的时候，这个恶意的代码就会被执行，从而来破坏网页的结构，获取用户的隐私信息等。

XSS攻击场景：

比如 A网站 有一个发布帖子的入口，如果用户在提交数据的时候，提交了一段 js 代码比如： `<script>alert("hello world");</script>`，然后 A网站 在渲染这个帖子的时候，直接把这个代码渲染了，那么这个代码就会执行，会在浏览器的窗口中弹出一个模态对话框来显示 `hello world`！如果攻击者能成功的运行以上这么一段 js 代码，那他能做的事情就有很多很多了！

XSS攻击防御：

1. 如果不需要显示一些富文本，那么在渲染用户提交的数据的时候，直接进行转义就可以了。在 Django 的模板中默认就是转义的。也可以把数据在存储到数据库之前，就转义再存储进去，这样以后在渲染的时候，即使不转义也不会有安全问题，示例代码如下：

```
from django.template.defaultfilters import escape
from .models import Comment
from django.http import HttpResponse
def comment(request):
    content = request.POST.get("content")
    escaped_content = escape(content)
    Comment.objects.create(content=escaped_content)
    return HttpResponse('success')
```

2. 如果对于用户提交上来的数据包含了一些富文本（比如：给字体换色，字体加粗等），那么这时候我们在渲染的时候也要以富文本的形式进行渲染，也即需要使用 `safe` 过滤器将其标记为安全的，这样才能显示出富文本样式。但是这样又会存在一个问题，如果用户提交上来的数据存在攻击的代码呢，那将其标记为安全的肯定是有问题的。示例代码如下：

```
# views.py
def index(request):
    message = "<span style='color:red;'>红色字体</span><script>alert('hello world')</script>";
    return render_template(request, 'index.html', context={"message":message})
```

```
# index.html
```

那么这时候该怎么办呢？这时候我们可以指定某些标签我们需要的（比如：**span**标签），而某些标签我们是不需要的（比如：**script**）那么我们在服务器处理数据的时候，就可以将这些需要的标签保留下来，把那些不需要的标签进行转义，或者干脆移除掉，这样就可以解决我们的问题了。这个方法是可行的，包括很多线上网站也是这样做的，在 **Python** 中，有一个库可以专门用来处理这个事情，那就是 **sanitizer**。接下来讲下这个库的使用。

bleach 库：

bleach 库是用来清理包含 **html** 格式字符串的库。他可以指定哪些标签需要保留，哪些标签是需要过滤掉的。也可以指定标签上哪些属性是可以保留，哪些属性是不需要的。想要使用这个库，可以通过以下命令进行安装：

```
pip install bleach
```

这个库最重要的一个方法是 **bleach.clean** 方法，**bleach.clean** 示例代码如下：

```
import bleach
from bleach.sanitizer import ALLOWED_TAGS, ALLOWED_ATTRIBUTES

@require_http_methods(['POST'])
def message(request):
    # 从客户端中获取提交的数据
    content = request.POST.get('content')

    # 在默认的允许标签中添加img标签
    tags = ALLOWED_TAGS + ['img']
    # 在默认的允许属性中添加src属性
    attributes = {**ALLOWED_ATTRIBUTES, 'img': ['src']}

    # 对提交的数据进行过滤
    cleaned_content = bleach.clean(content, tags=tags, attributes=attributes)

    # 保存到数据库中
    Message.objects.create(content=cleaned_content)

    return redirect(reverse('index'))
```

相关介绍如下：

1. **tags**：表示允许哪些标签。
2. **attributes**：表示标签中允许哪些属性。
3. **ALLOWED_TAGS**：这个变量是 **bleach** 默认定义的一些标签。如果不符合要求，可以对其进行增加或者删除。
4. **ALLOWED_ATTRIBUTES**：这个变量是 **bleach** 默认定义的一些属性。如果不符合要求，可以对

其进行增加或者删除。

bleach更多资料:

1. github地址: <https://github.com/mozilla/bleach>
2. 文档地址: <https://bleach.readthedocs.io/>

clickjacking攻击:

clickjacking 攻击又称作点击劫持攻击。是一种在网页中将恶意代码等隐藏在看似无害的内容（如按钮）之下，并诱使用户点击的手段。

clickjacking攻击场景:

场景一:

如用户收到一封包含一段视频的电子邮件，但其中的“播放”按钮并不会真正播放视频，而是链入一购物网站。这样当用户试图“播放视频”时，实际是被诱骗而进入了一个购物网站。

场景二:

用户进入到一个网页中，里面包含了一个非常有诱惑力的 按钮A，但是这个按钮上面浮了一个透明的 iframe 标签，这个 iframe 标签加载了另外一个网页，并且他将这个网页的某个按钮和原网页中的 按钮A 重合，所以你在点击 按钮A 的时候，实际上点的是通过 iframe 加载的另外一个网页的按钮。比如我现在有一个百度贴吧，想要让更多的用户来关注，那么我们可以准备以下一个页面：

```
<!DOCTYPE html>
<html>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<head>
<title>点击劫持</title>
<style>
    iframe{
        opacity:0.01;
        position:absolute;
        z-index:2;
        width: 100%;
        height: 100%;
    }
    button{
        position:absolute;
        top: 345px;
        left: 630px;
        z-index: 1;
        width: 72px;
        height: 26px;
    }
</style>
</head>
```

```
<body>
  这个合影里面怎么会有你?
  <button>查看详情</button>
  <iframe src="http://tieba.baidu.com/f?kw=%C3%C0%C5%AE"></iframe>
</body>
</html>
```

页面看起来比较简陋，但是实际上可能会比这些更精致一些。当这个页面通过某种手段被传播出去后，用户如果点击了“查看详情”，实际上点击到的是关注的按钮，这样就可以增加了一个粉丝。

clickjacking防御:

像以上场景1，是没有办法避免的，受伤害的是用户。而像场景2，受伤害的是百度贴吧网站和用户。这种场景是可以避免的，只要设置百度贴吧不允许使用 `iframe` 被加载到其他网页中，就可以避免这种行为了。我们可以通过在响应头中设置 `X-Frame-Options` 来设置这种操作。`X-Frame-Options` 可以设置以下三个值：

1. `DENY`：不让任何网页使用 `iframe` 加载我这个页面。
2. `SAMEORIGIN`：只允许在相同域名（也就是我自己的网站）下使用 `iframe` 加载我这个页面。
3. `ALLOW-FROM origin`：允许任何网页通过 `iframe` 加载我这个网页。

在 Django 中，使用中间件 `django.middleware.clickjacking.XFrameOptionsMiddleware` 可以帮我们堵上这个漏洞，这个中间件设置了 `X-Frame-Option` 为 `SAMEORIGIN`，也就是只有在自己的网站下才可以使用 `iframe` 加载这个网页，这样就可以避免其他别有心机的网页去通过 `iframe` 去加载了。

SQL注入

所谓SQL注入，就是通过把SQL命令插入到表单中或页面请求的查询字符串中，最终达到欺骗服务器执行恶意的SQL命令。具体来说，它是利用现有应用程序，将（恶意的）SQL命令注入到后台数据库引擎执行的能力，它可以通过在Web表单中输入（恶意）SQL语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行SQL语句。比如先前的很多影视网站泄露VIP会员密码大多就是通过WEB表单递交查询字符暴出的。

场景：

比如现在数据库中有一个 `front_user` 表，表结构如下：

```
class User(models.Model):
    telephone = models.CharField(max_length=11)
    username = models.CharField(max_length=100)
    password = models.CharField(max_length=100)
```

然后我们使用原生 `sql` 语句实现以下需求：

1. 实现一个根据用户 `id` 获取用户详情的视图。示例代码如下：

```
def index(request):
    user_id = request.GET.get('user_id')
    cursor = connection.cursor()
    cursor.execute("select id,username from front_user where id=%s" % user_id)
    rows = cursor.fetchall()
    for row in rows:
        print(row)
    return HttpResponse('success')
```

这样表面上看起来没有问题。但是如果用户传的 `user_id` 是等于 `1 or 1=1`，那么以上拼接后的 `sql` 语句为：

```
select id,username from front_user where id=1 or 1=1
```

以上 `sql` 语句的条件是 `id=1 or 1=1`，只要 `id=1` 或者是 `1=1` 两个有一个成立，那么整个条件就成立。毫无疑问 `1=1` 是肯定成立的。因此执行完以上 `sql` 语句后，会将 `front_user` 表中所有的数据都提取出来。

2. 实现一个根据用户的 `username` 提取用户的视图。示例代码如下：

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

```

username = request.GET.get('username')
cursor = connection.cursor()
cursor.execute("select id,username from front_user where username='%s'" % user
name)
rows = cursor.fetchall()
for row in rows:
    print(row)
return HttpResponse('success')

```

这样表面上看起来也没有问题。但是如果用户传的 `username` 是 `zhiliao' or '1=1`，那么以上拼接后的 `sql` 语句为：

```
select id,username from front_user where username='zhiliao' or '1=1'
```

以上 `sql` 语句的条件是 `username='zhiliao'` 或者是一个字符串，毫无疑问，字符串的判断是肯定成立的。因此会将 `front_user` 表中所有的数据都提取出来。

sql注入防御：

以上便是 `sql` 注入的原理。他通过传递一些恶意的参数来破坏原有的 `sql` 语句以便达到自己的目的。当然 `sql` 注入远远没有这么简单，我们现在讲到的只是冰山一角。那么如何防御 `sql` 注入呢？归类起来主要有以下几点：

1. 永远不要信任用户的输入。对用户的输入进行校验，可以通过正则表达式，或限制长度；对单引号和双引号进行转换等。
2. 永远不要使用动态拼装 `sql`，可以使用参数化的 `sql` 或者直接使用存储过程进行数据查询存取。比如：

```

def index(request):
    user_id = "1 or 1=1"
    cursor = connection.cursor()
    cursor.execute("select id,username from front_user where id=%s",(user_id,))
    rows = cursor.fetchall()
    for row in rows:
        print(row)
    return HttpResponse('success')

```

3. 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。
4. 不要把机密信息直接存放，加密或者hash掉密码和敏感的信息。
5. 应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装。

在Django中如何防御 sql 注入：

1. 使用 ORM 来做数据的增删改查。因为 ORM 使用的是参数化的形式执行 sql 语句的。
2. 如果万一要执行原生 sql 语句，那么建议不要拼接sql，而是使用参数化的形式。

验证和授权概述

Django 有一个内置的授权系统。他用来处理用户、分组、权限以及基于 `cookie` 的会话系统。Django 的授权系统包括验证和授权两个部分。验证是验证这个用户是否是他声称的人（比如用户名和密码验证，角色验证），授权是给予他相应的权限。Django 内置的权限系统包括以下方面：

1. 用户。
2. 权限。
3. 分组。
4. 一个可以配置的密码哈希系统。
5. 一个可插拔的后台管理系统。

使用授权系统：

默认中创建完一个 `django` 项目后，其实就已经集成了授权系统。那哪些部分是跟授权系统相关的配置呢。以下做一个简单列表：

INSTALLED_APPS :

1. `django.contrib.auth` : 包含了一个核心授权框架，以及大部分的模型定义。
2. `django.contrib.contenttypes` : `Content Type` 系统，可以用来关联模型和权限。

中间件：

1. `SessionMiddleware` : 用来管理 `session` 。
2. `AuthenticationMiddleware` : 用来处理和当前 `session` 相关联的用户。

User模型

User 模型是这个框架的核心部分。他的完整的路径是在 `django.contrib.auth.models.User` 。以下对这个 User 对象做一个简单了解：

字段：

内置的 User 模型拥有以下的字段：

1. `username` : 用户名。150个字符以内。可以包含数字和英文字符，以及 `_`、`@`、`+`、`.` 和 `-` 字符。不能为空，且必须唯一！
2. `first_name` : 歪果仁的 `first_name`，在30个字符以内。可以为空。
3. `last_name` : 歪果仁的 `last_name`，在150个字符以内。可以为空。
4. `email` : 邮箱。可以为空。
5. `password` : 密码。经过哈希过后的密码。
6. `groups` : 分组。一个用户可以属于多个分组，一个分组可以拥有多个用户。`groups` 这个字段是跟 `Group` 的一个多对多的关系。
7. `user_permissions` : 权限。一个用户可以拥有多个权限，一个权限可以被多个用户所有用。和 `Permission` 属于一种多对多的关系。
8. `is_staff` : 是否可以进入到 `admin` 的站点。代表是否是员工。
9. `is_active` : 是否是可用的。对于一些想要删除账号的数据，我们设置这个值为 `False` 就可以了，而不是真正的从数据库中删除。
10. `is_superuser` : 是否是超级管理员。如果是超级管理员，那么拥有整个网站的所有权限。
11. `last_login` : 上次登录的时间。
12. `date_joined` : 账号创建的时间。

User模型的基本用法：

创建用户：

通过 `create_user` 方法可以快速的创建用户。这个方法必须要传递 `username`、`email`、`password`。示例代码如下：

```
from django.contrib.auth.models import User
user = User.objects.create_user('zhiliao', 'hynever@zhiliao.com', '111111')
# 此时user对象已经存储到数据库中了。当然你还可以继续使用user对象进行一些修改
user.last_name = 'abc'
user.save()
```

创建超级用户：

创建超级用户有两种方式。第一种是使用代码的方式。用代码创建超级用户跟创建普通用户非常的类似，只不过是使用 `create_superuser` 。示例代码如下：

```
from django.contrib.auth.models import User
User.objects.create_superuser('admin', 'admin@163.com', '111111')
```

也可以通过命令行的方式。命令如下：

```
python manage.py createsuperuser
```

后面就会提示你输入用户名、邮箱以及密码。

修改密码：

因为密码是需要经过加密后才能存储进去的。所以如果想要修改密码，不能直接修改 `password` 字段，而需要通过调用 `set_password` 来达到修改密码的目的。示例代码如下：

```
from django.contrib.auth.models import User
user = User.objects.get(pk=1)
user.set_password('新的密码')
user.save()
```

登录验证：

Django 的验证系统已经帮我们实现了登录验证的功能。通过 `django.contrib.auth.authenticate` 即可实现。这个方法只能通过 `username` 和 `password` 来进行验证。示例代码如下：

```
from django.contrib.auth import authenticate
user = authenticate(username='zhiliao', password='111111')
# 如果验证通过了，那么就会返回一个user对象。
if user is not None:
    # 执行验证通过后的代码
else:
    # 执行验证没有通过的代码。
```

扩展用户模型：

Django 内置的 `User` 模型虽然已经足够强大了。但是有时候还是不能满足我们的需求。比如在验证用户登录的时候，他用的是用户名作为验证，而我们有时候需要通过手机号码或者邮箱来进行验证。还有比如我们想要增加一些新的字段。那么这时候我们就需要扩展用户模型了。扩展用户模型有多种方式。这里我们来一一讨论下。

1. 设置Proxy模型:

如果你对 Django 提供的字段，以及验证的方法都比较满意，没有什么需要改的。但是只是需要在他原有的基础之上增加一些操作的方法。那么建议使用这种方式。示例代码如下：

```
class Person(User):
    class Meta:
        proxy = True

    def get_blacklist(self):
        return self.objects.filter(is_active=False)
```

在以上，我们定义了一个 Person 类，让他继承自 User，并且在 Meta 中设置 proxy=True，说明这个只是 User 的一个代理模型。他并不会影响原来 User 模型在数据库中表的结构。以后如果你想方便的获取所有黑名单的人，那么你就可以通过 Person.get_blacklist() 就可以获取到。并且 User.objects.all() 和 Person.objects.all() 其实是等价的。因为他们都是从 User 这个模型中获取所有的数据。

2. 一对一外键:

如果你对用户验证方法 authenticate 没有其他要求，就是使用 username 和 password 即可完成。但是想要在原来模型的基础之上添加新的字段，那么可以使用一对一外键的方式。示例代码如下：

```
from django.contrib.auth.models import User
from django.db import models
from django.dispatch import receiver
from django.db.models.signals import post_save

class UserExtension(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name='extension')
    birthday = models.DateField(null=True, blank=True)
    school = models.CharField(max_length=100)

@receiver(post_save, sender=User)
def create_user_extension(sender, instance, created, **kwargs):
    if created:
        UserExtension.objects.create(user=instance)
    else:
        instance.extension.save()
```

以上定义一个 UserExtension 的模型，并且让她和 User 模型进行一对一的绑定，以后我们新增的字段，就添加到 UserExtension 上。并且还写了一个接受保存模型的信号处理方法，只要是 User 调用了 save 方法，那么就会创建一个 UserExtension 和 User 进行绑定。

3. 继承自 `AbstractUser` :

对于 `authenticate` 不满意, 并且不想要修改原来 `User` 对象上的一些字段, 但是想要增加一些字段, 那么这时候可以直接继承自 `django.contrib.auth.models.AbstractUser`, 其实这个类也是 `django.contrib.auth.models.User` 的父类。比如我们想要在原来 `User` 模型的基础之上添加一个 `telephone` 和 `school` 字段。示例代码如下:

```
from django.contrib.auth.models import AbstractUser
class User(AbstractUser):
    telephone = models.CharField(max_length=11,unique=True)
    school = models.CharField(max_length=100)

    # 指定telephone作为USERNAME_FIELD, 以后使用authenticate
    # 函数验证的时候, 就可以根据telephone来验证
    # 而不是原来的username
    USERNAME_FIELD = 'telephone'
    REQUIRED_FIELDS = []

    # 重新定义Manager对象, 在创建user的时候使用telephone和
    # password, 而不是使用username和password
    objects = UserManager()

class UserManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self,telephone,password,**extra_fields):
        if not telephone:
            raise ValueError("请填入手机号码! ")
        user = self.model(telephone=telephone,*extra_fields)
        user.set_password(password)
        user.save()
        return user

    def create_user(self,telephone,password,**extra_fields):
        extra_fields.setdefault('is_superuser',False)
        return self._create_user(telephone,password)

    def create_superuser(self,telephone,password,**extra_fields):
        extra_fields['is_superuser'] = True
        return self._create_user(telephone,password)
```

然后再在 `settings` 中配置好 `AUTH_USER_MODEL=youapp.User`。

这种方式因为破坏了原来`User`模型的表结构, 所以必须要在第一次 `migrate` 前就先定义好。

4. 继承自 `AbstractBaseUser` 模型:

如果你想修改默认的验证方式，并且对于原来 `User` 模型上的一些字段不想要，那么可以自定义一个模型，然后继承自 `AbstractBaseUser`，再添加你想要的字段。这种方式会比较麻烦，最好是确定自己对 `Django` 比较了解才推荐使用。步骤如下：

1. 创建模型。示例代码如下：

```
class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    username = models.CharField(max_length=150)
    telephone = models.CharField(max_length=11, unique=True)
    is_active = models.BooleanField(default=True)

    USERNAME_FIELD = 'telephone'
    REQUIRED_FIELDS = []

    objects = UserManager()

    def get_full_name(self):
        return self.username

    def get_short_name(self):
        return self.username
```

其中 `password` 和 `last_login` 是在 `AbstractBaseUser` 中已经添加好了的，我们直接继承就可以了。然后我们再添加我们想要的字段。比如 `email`、`username`、`telephone` 等。这样就可以实现自己想要的字段了。但是因为我们重写了 `User`，所以应该尽可能的模拟 `User` 模型：

- `USERNAME_FIELD`：用来描述 `User` 模型名字字段的字符串，作为唯一的标识。如果没有修改，那么会使用 `USERNAME` 来作为唯一字段。
 - `REQUIRED_FIELDS`：一个字段名列表，用于当通过 `createsuperuser` 管理命令创建一个用户时的提示。
 - `is_active`：一个布尔值，用于标识用户当前是否可用。
 - `get_full_name()`：获取完整的名字。
 - `get_short_name()`：一个比较简短的用户名。
2. 重新定义 `UserManager`：我们还需要定义自己的 `UserManager`，因为默认的 `UserManager` 在创建用户的时候使用的是 `username` 和 `password`，那么我们要替换成 `telephone`。示例代码如下：

```
class UserManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self, telephone, password, **extra_fields):
        if not telephone:
            raise ValueError("请填入手机号码！")
        user = self.model(telephone=telephone, *extra_fields)
```

```

        user.set_password(password)
        user.save()
        return user

    def create_user(self, telephone, password, **extra_fields):
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(telephone, password)

    def create_superuser(self, telephone, password, **extra_fields):
        extra_fields['is_superuser'] = True
        return self._create_user(telephone, password)

```

3. 在创建了新的 `User` 模型后，还需要在 `settings` 中配置好。配置 `AUTH_USER_MODEL='appname.User'`。
4. 如何使用这个自定义的模型：比如以后我们有一个 `Article` 模型，需要通过外键引用这个 `User` 模型，那么可以通过以下两种方式引用。
第一种就是直接将 `User` 导入到当前文件中。示例代码如下：

```

from django.db import models
from myauth.models import User
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)

```

这种方式是可以行得通的。但是为了更好的使用性，建议还是将 `User` 抽象出来，使用 `settings.AUTH_USER_MODEL` 来表示。示例代码如下：

```

from django.db import models
from django.conf import settings
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

```

这种方式因为破坏了原来 `User` 模型的表结构，所以必须要在第一次 `migrate` 前就先定义好。

权限和分组

登录、注销和登录限制：

登录

在使用 `authenticate` 进行验证后，如果验证通过了。那么会返回一个 `user` 对象，拿到 `user` 对象后，可以使用 `django.contrib.auth.login` 进行登录。示例代码如下：

```
user = authenticate(username=username, password=password)
if user is not None:
    if user.is_active:
        login(request, user)
```

注销：

注销，或者说退出登录。我们可以通过 `django.contrib.auth.logout` 来实现。他会清理掉这个用户的 `session` 数据。

登录限制：

有时候，某个视图函数是需要经过登录后才能访问的。那么我们可以通过 `django.contrib.auth.decorators.login_required` 装饰器来实现。示例代码如下：

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

# 在验证失败后，会跳转到/accounts/login/这个url页面
@login_required(login_url='/accounts/login/')
def my_view(request):
    pass
```

权限：

`Django` 中内置了权限的功能。他的权限都是针对表或者说是模型级别的。比如对某个模型上的数据是否可以增删改查操作。他不能针对数据级别的，比如对某个表中的某条数据能否进行增删改查操作（如果实现数据级别的，考虑使用 `django-guardian` ）。创建完一个模型后，针对这

个模型默认就有三种权限，分别是增/删/改/。可以在执行完 `migrate` 命令后，查看数据库中的 `auth_permission` 表中的所有权限。

id	content_type_id	codename	name
1	1	add_logentry	Can add log entry
2	1	change_logentry	Can change log entry
3	1	delete_logentry	Can delete log entry
4	2	add_permission	Can add permission
5	2	change_permission	Can change permission
6	2	delete_permission	Can delete permission
7	3	add_group	Can add group
8	3	change_group	Can change group
9	3	delete_group	Can delete group
10	4	add_contenttype	Can add content type
11	4	change_contenttype	Can change content type
12	4	delete_contenttype	Can delete content type
13	5	add_session	Can add session
14	5	change_session	Can change session
15	5	delete_session	Can delete session
16	6	add_user	Can add user
17	6	change_user	Can change user
18	6	delete_user	Can delete user

其中的 `codename` 表示的是权限的名字。 `name` 表示的是这个权限的作用。

通过定义模型添加权限：

如果我们想要增加新的权限，比如查看某个模型的权限，那么我们可以在定义模型的时候在 `Meta` 中定义好。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(get_user_model(), on_delete=models.CASCADE)

    class Meta:
        permissions = (
            ('view_article', 'can view article'),
        )
```

通过代码添加权限：

权限都是 `django.contrib.auth.Permission` 的实例。这个模型包含三个字段，`name`、`codename` 以及 `content_type`，其中的 `content_type` 表示这个 `permission` 是属于哪个 `app` 下的哪个 `models`。用 `Permission` 模型创建权限的代码如下：

```
from django.contrib.auth.models import Permission, ContentType
from .models import Article
content_type = ContentType.objects.get_for_model(Article)
permission = Permission.objects.create(name='可以编辑的权限', codename='edit_article', content_type=content_type)
```

用户与权限管理：

权限本身只是一个数据，必须和用户进行绑定，才能起到作用。`User` 模型和权限之间的管理，可以通过以下几种方式来管理：

1. `myuser.user_permissions.set(permission_list)`：直接给定一个权限的列表。
2. `myuser.user_permissions.add(permission, permission, ...)`：一个个添加权限。
3. `myuser.user_permissions.remove(permission, permission, ...)`：一个个删除权限。
4. `myuser.user_permissions.clear()`：清除权限。
5. `myuser.has_perm('<app_name>.<codename>')`：判断是否拥有某个权限。权限参数是一个字符串，格式是 `app_name.codename`。
6. `myuser.get_all_permissions()`：获取所有的权限。

权限限定装饰器：

使用 `django.contrib.auth.decorators.permission_required` 可以非常方便的检查用户是否拥有这个权限，如果拥有，那么就可以进入到指定的视图函数中，如果不拥有，那么就会报一个 `400` 错误。示例代码如下：

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

@permission_required('front.view_article')
def my_view(request):
    ...
```

分组：

权限有很多，一个模型就有最少三个权限，如果一些用户拥有相同的权限，那么每次都要重复添加。这时候分组就可以帮我们解决这种问题了，我们可以把一些权限归类，然后添加到某个分组中，之后再和把需要赋予这些权限的用户添加到这个分组中，就比较好管理了。分组我们使用的

是 `django.contrib.auth.models.Group` 模型，每个用户组拥有 `id` 和 `name` 两个字段，该模型在数据库被映射为 `auth_group` 数据表。

分组操作：

1. `Group.object.create(group_name)`：创建分组。
 2. `group.permissions`：某个分组上的权限。多对多的关系。
 - `group.permissions.add`：添加权限。
 - `group.permissions.remove`：移除权限。
 - `group.permissions.clear`：清除所有权限。
 - `user.get_group_permissions()`：获取用户所属组的权限。
 3. `user.groups`：某个用户上的所有分组。多对多的关系。
-

在模板中使用权限：

在 `settings.TEMPLATES.OPTIONS.context_processors` 下，因为添加了 `django.contrib.auth.context_processors.auth` 上下文处理器，因此在模板中可以直接通过 `perms` 来获取用户的所有权限。示例代码如下：

```
{% if perms.front.add_article %}
    <a href='/article/add/'>添加文章</a>
{% endif %}
```

memcached

什么是memcached:

1. memcached 之前是 danga 的一个项目，最早是为LiveJournal服务的，当初设计师为了加速LiveJournal访问速度而开发的，后来被很多大型项目采用。官网是 www.danga.com 或者是 memcached.org。
2. Memcached 是一个高性能的分布式的内存对象缓存系统，全世界有不少公司采用这个缓存项目来构建大负载的网站，来分担数据库的压力。Memcached 是通过在内存里维护一个统一的巨大的hash表，memcached 能存储各种各样的数据，包括图像、视频、文件、以及数据库检索的结果等。简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。
3. 哪些情况下适合使用 Memcached：存储验证码（图形验证码、短信验证码）、登录session等所有不是至关重要的数据。

安装和启动 memcached：

1. windows:

- 安装: `memcached.exe -d install`。
- 启动: `memcached.exe -d start`。

2. linux (ubuntu)：

- 安装: `sudo apt install memcached`
- 启动:

```
cd /usr/local/memcached/bin
./memcached -d start
```

3. 可能出现的问题:

- 提示你没有权限：在打开cmd的时候，右键使用管理员身份运行。
- 提示缺少 pthreadGC2.dll 文件：将 pthreadGC2.dll 文件拷贝到 windows/System32。
- 不要放在含有中文的路径下面。

4. 启动 memcached：

- `-d`：这个参数是让 memcached 在后台运行。
- `-m`：指定占用多少内存。以 M 为单位，默认为 64M。
- `-p`：指定占用的端口。默认端口是 11211。
- `-l`：别的机器可以通过哪个ip地址连接到我这台服务器。如果是通过 `service memcached start` 的方式，那么只能通过本机连接。如果想要让别的机器连接，就必须设置 `-l 0.0.0.0`。

如果想要使用以上参数来指定一些配置信息，那么不能使用 `service memcached start`，而应该使用 `/usr/bin/memcached` 的方式来运行。比如 `/usr/bin/memcached -u memcache -m 1024 -p 11222 start`。

telnet 操作 memcached：

telnet ip地址 [11211]

1. 添加数据：

◦ set：

■ 语法：

```
set key fls(是否压缩) timeout value_length
value
```

■ 示例：

```
set username 0 60 7
zhiliao
```

◦ add：

■ 语法：

```
add key fls(0) timeout value_length
value
```

■ 示例：

```
add username 0 60 7
xiaotuo
```

set 和 add 的区别：add 是只负责添加数据，不会去修改数据。如果添加的数据的 key 已经存在了，则添加失败，如果添加的 key 不存在，则添加成功。而 set 不同，如果 memcached 中不存在相同的 key，则进行添加，如果存在，则替换。

2. 获取数据：

◦ 语法：

```
get key
```

◦ 示例：

```
get username
```

3. 删除数据:

- 语法:

```
delete key
```

- 示例:

```
delete username
```

- `flush_all` : 删除 `memcached` 中的所有数据。

4. 查看 `memcached` 的当前状态:

- 语法: `stats` 。

通过 `python` 操作 `memcached` :

1. 安装: `python-memcached` : `pip install python-memcached` 。

2. 建立连接:

```
import memcache  
mc = memcache.Client(['127.0.0.1:11211', '192.168.174.130:11211'], debug=True)
```

3. 设置数据:

```
mc.set('username', 'hello world', time=60*5)  
mc.set_multi({'email': 'xxx@qq.com', 'telephone': '111111'}, time=60*5)
```

4. 获取数据:

```
mc.get('telephone')
```

5. 删除数据:

```
mc.delete('email')
```

6. 自增长:

```
mc.incr('read_count')
```

7. 自减少:

```
mc.decr('read_count')
```

memcached的安全性:

memcached 的操作不需要任何用户名和密码, 只需要知道 memcached 服务器的ip地址和端口号即可。因此 memcached 使用的时候尤其要注意他的安全性。这里提供两种安全的解决方案。分别来进行讲解:

1. 使用 `-l` 参数设置为只有本地可以连接: 这种方式, 就只能通过本机才能连接, 别的机器都不能访问, 可以达到最好的安全性。
2. 使用防火墙, 关闭 11211 端口, 外面也不能访问。

```
ufw enable # 开启防火墙
ufw disable # 关闭防火墙
ufw default deny # 防火墙以禁止的方式打开, 默认是关闭那些没有开启的端口
ufw deny 端口号 # 关闭某个端口
ufw allow 端口号 # 开启某个端口
```

在Django中使用memcached:

首先需要在 `settings.py` 中配置好缓存:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

如果想要使用多台机器, 那么可以在 `LOCATION` 指定多个连接, 示例代码如下:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

配置好 memcached 的缓存后, 以后在代码中就可以使用以下代码来操作 memcached 了:

```
from django.core.cache import cache

def index(request):
    cache.set('abc', 'zhiliao', 60)
```

```
print(cache.get('abc'))
response = HttpResponse('index')
return response
```

需要注意的是，`django` 在存储数据到 `memcached` 中的时候，不会将指定的 `key` 存储进去，而是会对 `key` 进行一些处理。比如会加一个前缀，会加一个版本号。如果想要自己加前缀，那么可以在 `settings.CACHES` 中添加 `KEY_FUNCTION` 参数：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'KEY_FUNCTION': lambda key, prefix_key, version: "django:%s"%key
    }
}
```


redis教程:

概述

redis 是一种 nosql 数据库,他的数据是保存在内存中,同时 redis 可以定时把内存数据同步到磁盘,即可以将数据持久化,并且他比 memcached 支持更多的数据结构(string , list列表[队列和栈] , set[集合] , sorted set[有序集合] , hash(hash表))。相关参考文

档: <http://redisdoc.com/index.html>

redis使用场景:

1. 登录会话存储: 存储在 redis 中, 与 memcached 相比, 数据不会丢失。
2. 排行版/计数器: 比如一些秀场类的项目, 经常会有一些前多少名的主播排名。还有一些文章阅读量的技术, 或者新浪微博的点赞数等。
3. 作为消息队列: 比如 celery 就是使用 redis 作为中间人。
4. 当前在线人数: 还是之前的秀场例子, 会显示当前系统有多少在线人数。
5. 一些常用的数据缓存: 比如我们的 BBS 论坛, 板块不会经常变化的, 但是每次访问首页都要从 mysql 中获取, 可以在 redis 中缓存起来, 不用每次请求数据库。
6. 把前200篇文章缓存或者评论缓存: 一般用户浏览网站, 只会浏览前面一部分文章或者评论, 那么可以把前面200篇文章和对应的评论缓存起来。用户访问超过的, 就访问数据库, 并且以后文章超过200篇, 则把之前的文章删除。
7. 好友关系: 微博的好友关系使用 redis 实现。
8. 发布和订阅功能: 可以用来做聊天软件。

redis 和 memcached 的比较:

	memcached	redis
类型	纯内存数据库	内存磁盘同步数据库
数据类型	在定义value时就要固定数据类型	不需要
虚拟内存	不支持	支持
过期策略	支持	支持
存储数据安全	不支持	可以将数据同步到dump.db中
灾难恢复	不支持	可以将磁盘中的数据恢复到内存中
分布式	支持	主从同步
订阅与发布	不支持	支持

redis 在 ubuntu 系统中的安装与启动

1. 安装:

```
sudo apt-get install redis-server
```

2. 卸载:

```
sudo apt-get purge --auto-remove redis-server
```

3. 启动: redis 安装后, 默认会自动启动, 可以通过以下命令查看:

```
ps aux|grep redis
```

如果想自己手动启动, 可以通过以下命令进行启动:

```
sudo service redis-server start
```

4. 停止:

```
sudo service redis-server stop
```

对 redis 的操作

对 redis 的操作可以用两种方式, 第一种方式采用 `redis-cli`, 第二种方式采用编程语言, 比如 `Python`、`PHP` 和 `JAVA` 等。

1. 使用 redis-cli 对 redis 进行字符串操作:

2. 启动 redis :

```
sudo service redis-server start
```

3. 连接上 redis-server :

```
redis-cli -h [ip] -p [端口]
```

4. 添加:

```
set key value  
如:  
set username xiaotuo
```

将字符串值 `value` 关联到 `key`。如果 `key` 已经持有其他值，`set` 命令就覆写旧值，无视其类型。并且默认的过期时间是永久，即永远不会过期。

5. 删除：

```
del key
如：
del username
```

6. 设置过期时间：

```
expire key timeout(单位为秒)
```

也可以在设置值的时候，一同指定过期时间：

```
set key value EX timeout
或：
setex key timeout value
```

7. 查看过期时间：

```
ttl key
如：
ttl username
```

8. 查看当前 `redis` 中的所有 `key`：

```
keys *
```

9. 列表操作：

- 在列表左边添加元素：

```
lpush key value
```

将值 `value` 插入到列表 `key` 的表头。如果 `key` 不存在，一个空列表会被创建并执行 `lpush` 操作。当 `key` 存在但不是列表类型时，将返回一个错误。

- 在列表右边添加元素：

```
rpush key value
```

将值`value`插入到列表`key`的表尾。如果`key`不存在，一个空列表会被创建并执行`RPUSH`操作。当`key`存在但不是列表类型时，返回一个错误。

- 查看列表中的元素：

```
lrange key start stop
```

返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定,如果要左边的第一个到最后的一个 `lrange key 0 -1` 。

- 移除列表中的元素：

- 移除并返回列表 `key` 的头元素：

```
lpop key
```

- 移除并返回列表的尾元素：

```
rpop key
```

- 移除并返回列表 `key` 的中间元素：

```
lrem key count value
```

将删除 `key` 这个列表中， `count` 个值为 `value` 的元素。

- 指定返回第几个元素：

```
lindex key index
```

将返回 `key` 这个列表中，索引为 `index` 的这个元素。

- 获取列表中的元素个数：

```
llen key  
如：  
llen languages
```

- 删除指定的元素：

```
lrem key count value  
如：  
lrem languages 0 php
```

根据参数 `count` 的值，移除列表中与参数 `value` 相等的元素。`count` 的值可以是以下几种：

- `count > 0`：从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count`。
- `count < 0`：从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。
- `count = 0`：移除表中所有与 `value` 相等的值。

10. `set` 集合的操作：

◦ 添加元素：

```
sadd set value1 value2....  
如：  
sadd team xiaotuo datuo
```

◦ 查看元素：

```
smembers set  
如：  
smembers team
```

◦ 移除元素：

```
srem set member...  
如：  
srem team xiaotuo datuo
```

◦ 查看集合中的元素个数：

```
scard set  
如：  
scard team1
```

◦ 获取多个集合的交集：

```
sinter set1 set2  
如：  
sinter team1 team2
```

◦ 获取多个集合的并集：

```
sunion set1 set2  
如：  
sunion team1 team2
```

◦ 获取多个集合的差集：

```
sdiff set1 set2
如：
sdiff team1 team2
```

11. hash 哈希操作：

- 添加一个新值：

```
hset key field value
如：
hset website baidu baidu.com
```

将哈希表 `key` 中的域 `field` 的值设为 `value` 。

如果 `key` 不存在，一个新的哈希表被创建并进行 `HSET` 操作。如果域 `field` 已经存在于哈希表中，旧值将被覆盖。

- 获取哈希中的 `field` 对应的值：

```
hget key field
如：
hget website baidu
```

- 删除 `field` 中的某个 `field`：

```
hdel key field
如：
hdel website baidu
```

- 获取某个哈希中所有的 `field` 和 `value`：

```
hgetall key
如：
hgetall website
```

- 获取某个哈希中所有的 `field`：

```
hkeys key
如：
hkeys website
```

- 获取某个哈希中所有的值：

```
hvals key
```

```
如：  
hvals website
```

- 判断哈希中是否存在某个 `field` :

```
hexists key field  
如：  
hexists website baidu
```

- 获取哈希中总共的键值对:

```
hlen field  
如：  
hlen website
```

12. 事务操作：Redis事务可以一次执行多个命令，事务具有以下特征：

- 隔离操作：事务中的所有命令都会序列化、按顺序地执行，不会被其他命令打扰。
- 原子操作：事务中的命令要么全部被执行，要么全部都不执行。
- 开启一个事务：

```
multi
```

以后执行的所有命令，都在这个事务中执行的。

- 执行事务：

```
exec
```

将会在 `multi` 和 `exec` 中的操作一并提交。

- 取消事务：

```
discard
```

会将 `multi` 后的所有命令取消。

- 监视一个或者多个 `key` :

```
watch key...
```

监视一个(或多个)`key`，如果在事务执行之前这个(或这些) `key` 被其他命令所改动，那么事务将被打断。

- 取消所有 key 的监视:

```
unwatch
```

13. 发布/订阅操作:

- 给某个频道发布消息:

```
publish channel message
```

- 订阅某个频道的消息:

```
subscribe channel
```

14. 持久化: redis 提供了两种数据备份方式, 一种是 RDB, 另外一种 AOF, 以下将详细介绍这两种备份策略:

|| RDB | AOF || --- | --- | --- || 开启关闭 | 开启: 默认开启。关闭: 把配置文件中所有的save都注释, 就是关闭了。 | 开启: 在配置文件中 appendonly yes 即开启了 aof, 为 no 关闭。

|| 同步机制 | 可以指定某个时间内发生多少个命令进行同步。比如1分钟内发生了2次命令, 就做一次同步。 | 每秒同步或者每次发生命令后同步 || 存储内容 | 存储的是redis里面的具体的值 | 存储的是执行的更新数据的操作命令 || 存储文件的路径 | 根据dir以及dbfilename来指定路径和具体的文件名 | 根据dir以及appendfilename来指定具体的路径和文件名 || 优点 | (1) 存储数据到文件中会进行压缩, 文件体积比aof小。(2) 因为存储的是redis具体的值, 并且会经过压缩, 因此在恢复的时候速度比AOF快。(3) 非常适用于备份。 | (1) AOF的策略是每秒钟或者每次发生写操作的时候都会同步, 因此即使服务器故障, 最多只会丢失1秒的数据。

(2) AOF存储的是Redis命令, 并且是直接追加到aof文件后面, 因此每次备份的时候只要添加新的数据进去就可以了。(3) 如果AOF文件比较大了, 那么Redis会进行重写, 只保留最小的命令集合。 || 缺点 | (1) RDB在多少时间内发生了多少写操作的时候就会出发同步机制, 因为采用压缩机制, RDB在同步的时候都重新保存整个Redis中的数据, 因此你一般会设置在最少5分钟才保存一次数据。在这种情况下, 一旦服务器故障, 会造成5分钟的数据丢失。(2) 在数据保存进RDB的时候, Redis会fork出一个子进程用来同步, 在数据量比较大的时候, 可能会非常耗时。 | (1) AOF文件因为没有压缩, 因此体积比RDB大。(2) AOF是在每秒或者每次写操作都进行备份, 因此如果并发量比较大, 效率可能有点慢。(3) AOF文件因为存储的是命令, 因此在灾难恢复的时候Redis会重新运行AOF中的命令, 速度不及RDB。 || 更多 | <http://redisdoc.com/topic/persistence.html#redis> ||

15. 安全: 在配置文件中, 设置 requirepass password, 那么客户端连接的时候, 需要使用密码:

```
> redis-cli -p 127.0.0.1 -p 6379
redis> set username xxx
(error) NOAUTH Authentication required.
redis> auth password
redis> set username xxx
```


OK

Python操作redis

1. 安装 `python-redis` :

```
pip install redis
```

2. 新建一个文件比如 `redis_test.py` , 然后初始化一个 `redis` 实例变量, 并且在 `ubuntu` 虚拟机中开启 `redis` 。比如虚拟机的 `ip` 地址为 `192.168.174.130` 。示例代码如下:

```
# 从redis包中导入Redis类
from redis import Redis
# 初始化redis实例变量
xtredis = Redis(host='192.168.174.130',port=6379)
```

3. 对字符串的操作: 操作 `redis` 的方法名称, 跟之前使用 `redis-cli` 一样, 现就一些常用的来做个简单介绍, 示例代码如下(承接以上的代码):

```
# 添加一个值进去, 并且设置过期时间为60秒, 如果不设置, 则永远不会过期
xtredis.set('username','xiaotuo',ex=60)
# 获取一个值
xtredis.get('username')
# 删除一个值
xtredis.delete('username')
# 给某个值自增1
xtredis.set('read_count',1)
xtredis.incr('read_count') # 这时候read_count变为2
# 给某个值减少1
xtredis.decr('read_count') # 这时候read_count变为1
```

4. 对列表的操作: 同字符串操作, 所有方法的名称跟使用 `redis-cli` 操作是一样的:

```
# 给languages这个列表往左边添加一个python
xtredis.lpush('languages','python')
# 给languages这个列表往左边添加一个php
xtredis.lpush('languages','php')
# 给languages这个列表往左边添加一个javascript
xtredis.lpush('languages','javascript')

# 获取languages这个列表中的所有值
print xtredis.lrange('languages',0,-1)
> ['javascript','php','python']
```

5. 对集合的操作:

```
# 给集合team添加一个元素xiaotuo
xtredis.sadd('team','xiaotuo')
# 给集合team添加一个元素datuo
xtredis.sadd('team','datuo')
# 给集合team添加一个元素slice
xtredis.sadd('team','slice')

# 获取集合中的所有元素
xtredis.smembers('team')
> ['datuo','xiaotuo','slice'] # 无序的
```

6. 对哈希(hash)的操作:

```
# 给website这个哈希中添加baidu
xtredis.hset('website','baidu','baidu.com')
# 给website这个哈希中添加google
xtredis.hset('website','google','google.com')

# 获取website这个哈希中的所有值
print xtredis.hgetall('website')
> {"baidu":"baidu.com","google":"google.com"}
```

7. 事务(管道)操作: redis 支持事务操作, 也即一些操作只有统一完成, 才能算完成。否则都执行失败, 用 python 操作 redis 也是非常简单, 示例代码如下:

```
# 定义一个管道实例
pip = xtredis.pipeline()
# 做第一步操作, 给BankA自增长1
pip.incr('BankA')
# 做第二步操作, 给BankB自减少1
pip.desc('BankB')
# 执行事务
pip.execute()
```

以上便展示了 `python-redis` 的一些常用方法, 如果想深入了解其他的方法, 可以参考 `python-redis` 的源代码 (查看源代码 `pycharm` 快捷键提示: 把鼠标光标放在 `import Redis` 的 `Redis` 上, 然后按 `ctrl+b` 即可进入)。