# Django与后端

Lambda X

2022-07-22

## Why Django

> Django 最初被设计用于具有快速开发需求的新闻类站点，目的是要实现简单快捷的网站开发。

- ~~小学期要用~~
- 开发省时省力，~~方便快速摆烂~~
- Python语言，比较熟悉

## 创建一个简单的应用

**Django 的文档如此友善而清晰以至于我下面的内容大致摘录自 Django 文档。**（~~你不妨现在离开会议然后去阅读 Django 文档~~）

### Helloworld

#### 建立项目

我们开始，先确认一下你的 Python 中是否安装好了 Django

```
python -m django --version
```

现在我们来创建一个 Django 项目

```
django-admin startproject mysite
```

此时你可以看看文件树

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

这些文件的含义参见文档。我们现在可以

```
python manage.py runserver
```

这里的 `manage.py` 是一个用于管理 Django 的命令行工具。许多如运行临时服务器、生成 migration，迁移数据库结构等操作都可以通过 `manage.py` 进行。我们也可以在 `manage.py` 中运行自定义命令。

在运行了上述指令后，Django 会默认监听 `127.0.0.1:8000` 。默认情况下，在修改文件后，这个临时服务器会自动重启。

如果希望 Django 监听其他端口，如 9999，可以使用

```
python manage.py runserver 9999
```

关于 `runserver` 的更多信息，可以参阅 文档

文档指出，这个服务器只能用于开发测试，而不应当用于生产环境。

### 新建应用

下面我们将演示新建应用。

```
python manage.py startapp <app_name>
```

例如，我们按照文档教程中的示例，新建一个名为 `polls` 的应用

```
python manage.py startapp polls
```

应用和项目是多对多的关系。一个应用通常用于完成一个相对独立的功能，可以安装于不同的项目中；而一个项目也可以包含多个应用。

### 创建视图

视图可以根据不同的请求给出 HTTP 响应，我们来为 `polls` 创建个简单的视图。例如我们在 `polls/views.py` 中输入

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

这时，调用 `index` 将返回一个 HTTP 响应，内容为 `Hello, world. You're at the polls index.` 。

### 创建路由

路由配置告诉 Django 哪些的 URL 应该被分配给哪些视图。我们来为我们刚才编写的 `index` 试图配置路由。

首先配置应用一级的路由，在 `polls` 目录下新建 `urls.py` ，输入以下内容

```
from django.urls import path

from . import views

urlpatterns = [
    path('hello', views.index, name='index'),
]
```

然后配置项目一级的路由

```
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls'))
]
```

这里需要注意 `include` 函数可以引用其他位置的 URL 配置，并将截断匹配的部分，将剩余部分分发给后续的 URL 配置进行匹配。这时我们运行临时服务器测试访问这个页面

```
python manage.py runserver
```

访问

```
http://localhost:8000/polls/hello
```

预期观察到的响应为 `Hello, world. You're at the polls index.`

我们回过头来看看我们的请求是如何被分配到 `index` 的。

当我们访问 `http://localhost:8000/polls/hello` ，Django 哪来进行匹配的部分是 `polls/hello`

首先在项目一级匹配，匹配到了 `polls/` ，于是 `polls/` 被去除，剩下 `hello` 子串到应用 `polls` 的 `urls.py` 中进行匹配。

`hello` 子串又匹配到了视图 `index`。此时 HTTP 请求作为第一个参数传给了 `index` 函数进行处理。`index` 函数处理这个请求后，应当返回一个 `HttpResponse` 对象，Django 会将这个对象对应的 HTTP 响应传递给客户端。

## Models

接下来我们来演示如何使用 Django 的 ORM 模型来操作数据库。Django 支持多种数据库。默认情况下，Django使用 SQLite3 作为数据库。这种情形无需配置。我们也可以使用 MySQL 作为数据库。

### `settings.py`

在使用数据库前，我们需要先了解 `settings.py` 中进行一些常用设置

- `DEBUG`

  决定项目是在调试状态还是在生产环境。

  文档指出

  > 永远不要在 `DEBUG` 开启的情况下将网站部署到生产中。

  DEBUG 开启时，产生错误会返回详细的错误报告，其中包含许多敏感信息，不宜在生产环境中泄露。

  DEBUG 关闭时，产生的错误仅有简略说明。

- `ALLOWED_HOSTS`

允许访问的域名列表，其他域名访问将直接被 ban，响应为 HTTP 400

- `INSTALLED_APPS`

  表示该**项目**使用了哪些**应用**。Django会为在其中列出的应用进行注册。会进行启动配置、跟踪 migration 等数据库信息。

- `DATABASES`

  表示使用何种数据库，以及使用数据库时的配置，如主机、端口、用户名密码等等

### 配置 MySQL 数据库

下面按照 Django 文档中的示例，演示为项目配置 MySQL 数据库。

首先安装 `mysqlclient`

```
pip install mysqlclient
```

然后在恰当位置创建 MySQL 数据库的配置文件 `my.cnf` ，输入你的数据库配置

```
[client]
host = 服务器地址
database = 所选数据库
user = 用户名
password = 密码
port = 端口
default-character-set = 字符集，这里推荐 utf8mb4
```

然后，在 settings.py 进行如下修改

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '<my.cnf 的地址>',
        },
    }
}
```

如果填入的地址为相对路径，那么将会是相对运行 `manage.py` 时的工作目录。

## 建立模型

模型对应于数据库中的表，可以用来设计和访问数据库的结构。我们为 `polls` 应用建立模型。编辑 `models.py` 输入以下内容

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.FloatField('date published')

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

上面的代码建立了两个模型，对应于数据库中的两张表。模型是 `models.Model` 的子类，模型的字段是 `models.Field` 的实例。模型的每个字段对应于数据库中的一个字段。

在建立模型后我们需要让 Django 在数据库中建立这些模型，或对模型的修改做出对应数据库结构的调整。因此我们需要首先注册这一应用。在 `settings.py` 中，将我们的应用添加到 `INSTALLED_APPS` 中

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

这时应用 `polls` 就被添加到了我们的 Django 项目中了。

我们运行

```
> python manage.py makemigrations polls
Migrations for 'polls':
  polls\migrations\0001_initial.py
    - Create model Question
    - Create model Choice
```

此时 Django 会检测我们对应用 `polls` 的模型做了哪些修改，在将这些修改操作汇总到一个 Python 文件中。

我们可以看到在 `polls/migrations` 目录中出现了一个 `0001_initial.py` 文件。我们通常无需阅读 `migrations` 中的文件（如果有一天你不得不细细研读它们那你大概是遇到了什么大麻烦

接下来我们需要将修改操作应用到数据库

```
> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
```

```
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying polls.0001_initial... OK
  Applying sessions.0001_initial... OK
```

此时你对模型的修改将会被执行到数据库中。注意到修改包含了大量其他模型的操作，这是因为我们在 `INSTALLED_APPS` 中配置了许多应用，而这些应用的模型尚未建立。

对于应用更改时具体执行的 SQL 语句，我们可以使用

```
python manage.py sqlmigrate polls 0001
```

来查看 migration 0001 对应的 SQL 语句。

```
--
-- Create model Question
--
CREATE TABLE `polls_question` (
    `id` bigint AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `question_text` varchar(200) NOT NULL,
    `pub_date` double precision NOT NULL
);

--
-- Create model Choice
--
CREATE TABLE `polls_choice` (
    `id` bigint AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `choice_text` varchar(200) NOT NULL,
    `votes` integer NOT NULL,
    `question_id` bigint NOT NULL
);

ALTER TABLE `polls_choice`
    ADD CONSTRAINT `polls_choice_question_id_c5b4b260_fk_polls_question_id`
    FOREIGN KEY (`question_id`) REFERENCES `polls_question` (`id`);
```

从这些语句中我们发现 Django 默认为我们建立主键，并添加了外键约束。

总结起来，修改模型并应用修改分为 3 步：

> - 编辑 `models.py` 文件，改变模型。
> - 运行 `python manage.py makemigrations` 为模型的改变生成迁移文件。
> - 运行 `python manage.py migrate` 来应用数据库迁移。

## 一些常见的模型定义

下面以这段代码（魔改自一段软工代码）为例介绍一些模型构建的常用内容

```
class Role(models.TextChoices):
    MEMBER = "member"
    DEV = "dev"
    QA = "qa"
    SYS = "sys"
    SUPERMASTER = "supermaster"

class PendingModifyPasswordEmail(models.Model):
    # 指定主键
    id = models.BigAutoField(primary_key=True)

    # 唯一
    hash1 = models.CharField(max_length=100, unique=True)

    # 枚举类型
    role = models.CharField(max_length=12, choices=Role.choices)

    # 默认函数
    createdAt = models.FloatField(default=getTime.get_timestamp)

    class Meta:
        # 索引
        indexes = [
            models.Index(fields=["hash1", "role"]),
            models.Index(fields=["createdAt"]),
        ]

        # 联合唯一
        unique_together = ["hash1", "createdAt"]
```

## 模型 API

我们可以利用模型 API 完成对数据的操作。下面我们在 Django Shell 中演示一些常见模型 API 的使用方法。当然，我们通常会在视图函数中使用这些 API，使用交互式方式只是为了便于演示。

首先启动 Django Shell

```
python manage.py shell
```

在 Django Shell 中

```
>>> from polls.models import *

# 创建记录
>>> import time
>>> Question.objects.create(question_text="Is it a question?",
pub_date=time.time())
<Question: Question object (1)>
>>> q = Question(question_text="Is it a question?", pub_date=time.time())
>>> q.save()
>>> q.id
2
>>> q.question_text
"Is it a question?"

>>> Question.objects.create(question_text="Is it another question?",
pub_date=time.time())
<Question: Question object (3)>

# 显示所有记录
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>, <Question: Question object (2)>,
<Question: Question object (3)>]>

# 获取指定记录
>>> p = Question.objects.get(pk=2)
>>> p.id, p.question_text, p.pub_date
(2, 'Is it a question?', 1658423046.8407936)

>>> p = Question.objects.get(pk=114514)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\ProgramData\Anaconda3\envs\djangoProject\lib\site-
packages\django\db\models\manager.py", line
85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "C:\ProgramData\Anaconda3\envs\djangoProject\lib\site-
packages\django\db\models\query.py", line 49
6, in get
    raise self.model.DoesNotExist(
polls.models.Question.DoesNotExist: Question matching query does not exist.

# 使用 *_set API
>>> p.choice_set.create(choice_text='A', votes=0)
<Choice: Choice object (1)>
```

```
>>> p.choice_set.create(choice_text='B', votes=0)
<Choice: Choice object (2)>
>>> p.choice_set.create(choice_text='C', votes=0)
<Choice: Choice object (3)>
```

Django 的查询返回一个 QuerySet 对象。QuerySet 对象与 `map`，`zip` 类似，是惰性查询的。

在这里我们发现对象的显示不太方便，我们可以配置相应模型的 `__str__` 方法来自定义模型的显示内容

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.FloatField('date published')

    def __str__(self):
        return f"{self.question_text} @ {self.pub_date}"

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        return f"{self.choice_text} [{self.votes}]"
```

然后重启 Django Shell

```
>>> from polls.models import *

# 自定义显示
>>> Choice.objects.all()
<QuerySet [<Choice: A [0]>, <Choice: B [0]>, <Choice: C [0]>]>

>>> Question.objects.all()
<QuerySet [<Question: Is it a question? @ 1658422997.1246326>,
           <Question: Is it a question? @ 1658423046.8407936>,
           <Question: Is it another question? @ 1658423054.8635635>]>
```

接下来我们看看如何修改对象。和创建类似地，有两种方法

```
# save object: 修改特定对象
>>> ch = Choice.objects.get(pk=2)
>>> ch.votes = 114514
>>> ch.save()

# update: 修改整个 Query Set
>>> Choice.objects.filter(id=3).update(votes=255)
1

# 查看修改效果
>>> Choice.objects.all()
<QuerySet [<Choice: A [0]>, <Choice: B [114514]>, <Choice: C [255]>]>
```

然后是一些基本的查询方法

```
# 条件：votes == 0
>>> Choice.objects.filter(votes=0)
<QuerySet [<Choice: A [0]>]>

# 条件：votes >= 255
>>> Choice.objects.filter(votes__gte=255)
<QuerySet [<Choice: B [114514]>, <Choice: C [255]>]>

# 条件：question_text LIKE 'Is it another%'
>>> Question.objects.filter(question_text__startswith="Is it another")
<QuerySet [<Question: Is it another question? @ 1658423054.8635635>]>

# 排除
>>> Question.objects.exclude(question_text__startswith="Is it another")

# 特别地，一些多表关系操作变得十分容易
>>> Choice.objects.filter(question__id=2)
<QuerySet [<Choice: A [0]>, <Choice: B [114514]>, <Choice: C [255]>]>
>>> Choice.objects.filter(question__id=1)
<QuerySet []>

# 级联
>>> Choice.objects.filter(question__question_text__contains="Is")
<QuerySet [<Choice: A [0]>, <Choice: B [114514]>, <Choice: C [255]>]>

# COUNT
>>> Question.objects.get(pk=2).choice_set.count()

# ORDER, LIMIT
>>> Choice.objects.order_by("-id")
<QuerySet [<Choice: C [255]>, <Choice: B [114514]>, <Choice: A [0]>]>
>>> Choice.objects.order_by("-id")[1:2]
<QuerySet [<Choice: B [114514]>]>

# SELECT 指定列
>>> Choice.objects.values("id", "votes")
<QuerySet [{'id': 1, 'votes': 0}, {'id': 2, 'votes': 114514}, {'id': 3, 'votes':
255}]>
```

```
# 使用 Q 函数完成复杂逻辑
>>> from django.db.models import Q
>>> Choice.objects.filter(Q(question__id=2) & (~Q(choice_text__in=["A", "B"]) |
Q(id=1)))
<QuerySet [<Choice: A [0]>, <Choice: C [255]>]>

# 使用 F 函数来在查询中嵌入列名
>>> from django.db.models import F
>>> ch = Choice.objects.get(pk=1)
>>> ch.votes = F("votes") + 1
```

我们可以使用 `.query` 来查看 QuerySet 对应的 SQL 语句

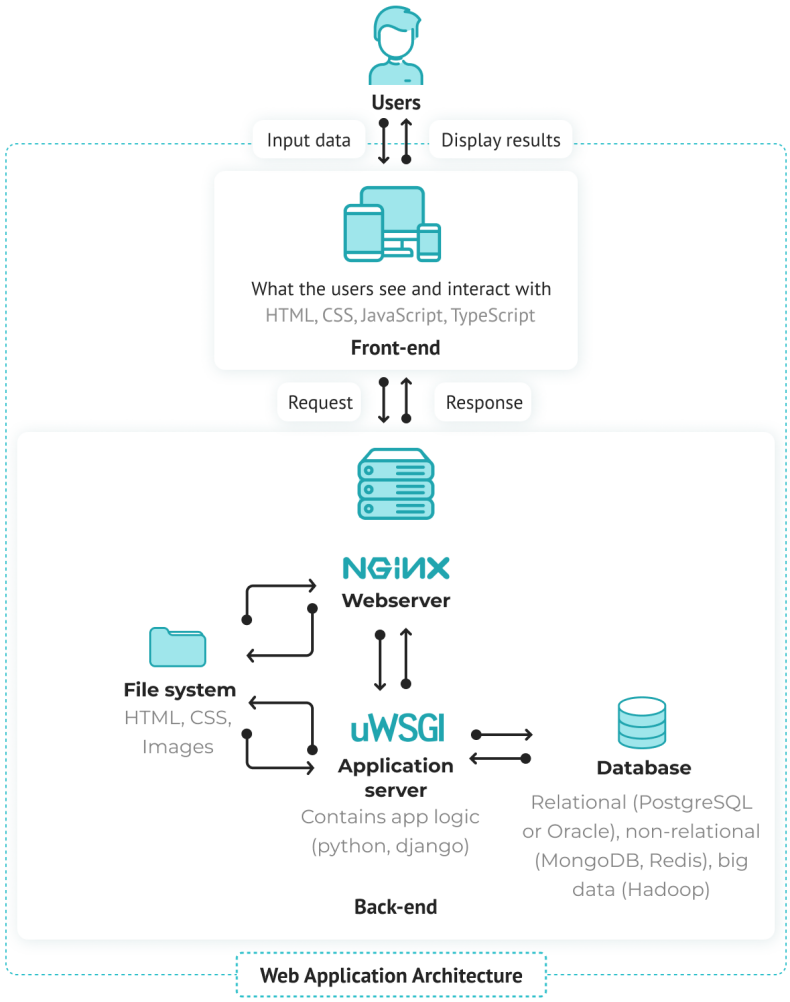
```
>>> print(
...     Choice.objects.filter(
...         Q(question__id=2) & (~Q(choice_text__in=["A", "B"]) | Q(id=1))
...     ).query
... )
SELECT `polls_choice`.`id`, `polls_choice`.`question_id`,
`polls_choice`.`choice_text`, `polls_choice`.`votes` FROM `polls_choice` WHERE
(`polls_choice`.`question_id` = 2 AND (NOT (`polls_choice`.`choice_text` IN (A,
B)) OR `polls_choice`.`id` = 1))
```

在查询复杂（或者开发者对 ORM接口不熟悉）的情况下也可以直接裸写 SQL。由于不同 SQL 实现的版本的差异，这样的代码并不想 ORM 那样具有较为良好的迁移性。

```
>>> Choice.objects.raw("SELECT * FROM ...")
```

当然，并不是说如果使用 ORM 就一定能保证代码在不同数据库之间无损转换。例如 Postgre SQL 的 DISTINCT ON 接口就会导致一部分代码迁移到 MySQL 后不能使用。而如果要把一部分数据从一种数据库迁移到另一种数据库，这通常也十分令人头大。

## 前后端分离的开发模式




对于 Web 应用来说，可以将其简单视为 2 部分。

- 前端：用户可以看到并与之交互的部分
- 后端：用户看不到，但承担着一些如数据存储、计算等等功能的部分。

对于前后端的互动，也可以简单分成两种模式

- 服务端渲染：服务端将用户看到的界面计算好，然后整体以 HTML 的形式返回
- 用户端渲染：服务端先将不含数据，但包含交互逻辑的前端传给客户端。客户端需要时向服务端请求数据，利用请求的数据来完成页面的渲染。

对于当前日趋复杂的 Web 应用来说，前后端分离的是一种日趋流行的趋势

- 前后端分离应用的页面在交互的过程中，只传递了数据，而不是渲染好的整个视图，更加高效、灵活
- 后端人员与前端人员的工作解耦，两者只需要通过约定的 API 进行协作，不需要代码的直接交互，不会出现后端开发里面嵌入了前端代码这种难以维护情形。

可能还需要说明的是，在学会前后端分离后，并不一定所有的任务都一定要上前后端分离，有些非常简单的任务用耦合的架构大概就能做的非常好了

## Templates & Forms

Django 对前后端耦合的开发模式有深入的支持，但模板和表单在前后端分离的开发中使用较少，不是本节的重点，在这里附上文档链接，供有兴趣的同学查阅。

(其实小学期用模板摆一下可能还是挺省时间的)

- 模板：https://docs.djangoproject.com/zh-hans/4.0/intro/tutorial03/
- 表单，Generic View：https://docs.djangoproject.com/zh-hans/4.0/intro/tutorial04/

## 使用 Django 收发请求

作为一个后端来说，最为常规的操作可能读取请求参数和收发各种 JSON。

### GET

例如我们希望实现这样一个接口，请求

```
/polls/questions/2
```

返回 `id=2` 的问题文本、创建时间和所有选项，此时我们可以写如下一个视图函数

```
from django.http import HttpRequest, JsonResponse
from .models import *
from django.forms.models import model_to_dict


def question_detail(req: HttpRequest, qid: int):
    res = Question.objects.filter(id=qid).first()
    if res:
        return JsonResponse({
            "code": 0,
            "data": {
                **model_to_dict(res),
                "choices": [
                    model_to_dict(ch, exclude=["question"])
                    for ch in res.choice_set.all()
                ]
            }
        })
    else:
        return JsonResponse({
            "code": -1
        }, status=404)
```

其接受一个 HttpRequest 对象和被查询问题的 `id`，使用 `model_to_dict` 将问题对象转换为 `dict`，再将问题对应所有的 `Choice` 转化也都转化为 `dict`。最后将这个 `dict` 与 `list` 的复合物封装成 JSON 返回。

然后我们在 `polls/urls.py` 中为其配置路由。

```
from django.urls import path

from . import views

urlpatterns = [
    path('question/<int:qid>', views.question_detail),
]
```

这里我们遇到一种新的路由格式。其在匹配到这一项后相当于调用了

```
views.question_detail(request=<HttpRequest object>, qid=<int:qid>)
```

更多 URL Pattern 写法详见文档

这时我们可以使用 Postman 对该接口进行验证和调试。

此外，对于 GET 请求中附带的参数，可以使用 `req.GET` 获取，例如添加上

```
print(req.GET)
```

即可获取 GET 请求的参数

## POST

我们考虑另一种情形，就是给指定问题的指定选项投票，类似于以下场景

```
// 请求
{
    "qid": 2,
    "cid": 0
}

// 响应
{
    "code": 0,
    "choices": [...]
}
```

这时我们可以撰写类似这样的一个视图函数

```
def vote(req: HttpRequest):
    info = json.loads(req.body)
    try:
        qid = info["qid"]
        cid = info["cid"]
        ch = Choice.objects.get(pk=cid)
        if ch.question.id != qid:
            raise ValueError("Mismatched question and choice pair")
        ch.votes += 1
        ch.save()
        return JsonResponse({
            "code": 0,
            "choices": [
                model_to_dict(ch, exclude=["question"])
                for ch in Choice.objects.filter(question__id=qid)
            ]
        })
    except:
        if settings.DEBUG:
            raise
        return JsonResponse({
            "code": -1
```

```
        }, status=400)
```

然后添加一条路由

```
path('vote', views.vote)
```

这时在 Postman 中查询，发现响应为 HTTP 403，原因是缺少 CSRF token

不妨直接添加装饰器关闭 CSRF token 检查。

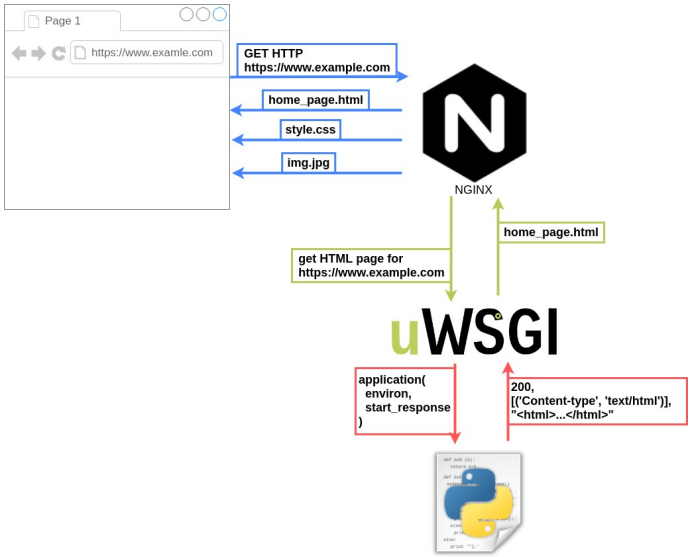
```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def vote(req: HttpRequest):
    ...
```

有时我们希望指定请求的方法，可以使用

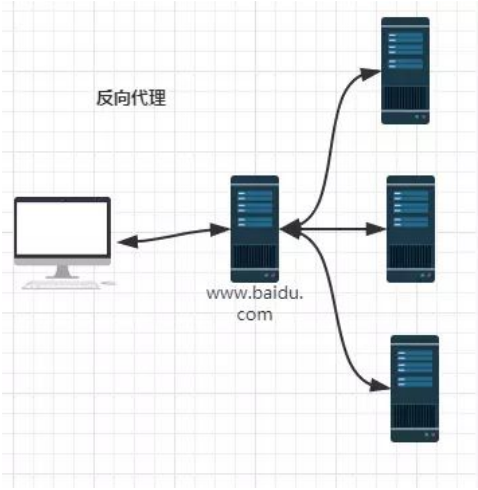
```
from django.views.decorators.http import require_http_methods as method

@method(["GET"])
def leaderboard(req: HttpRequest):
    ...
```

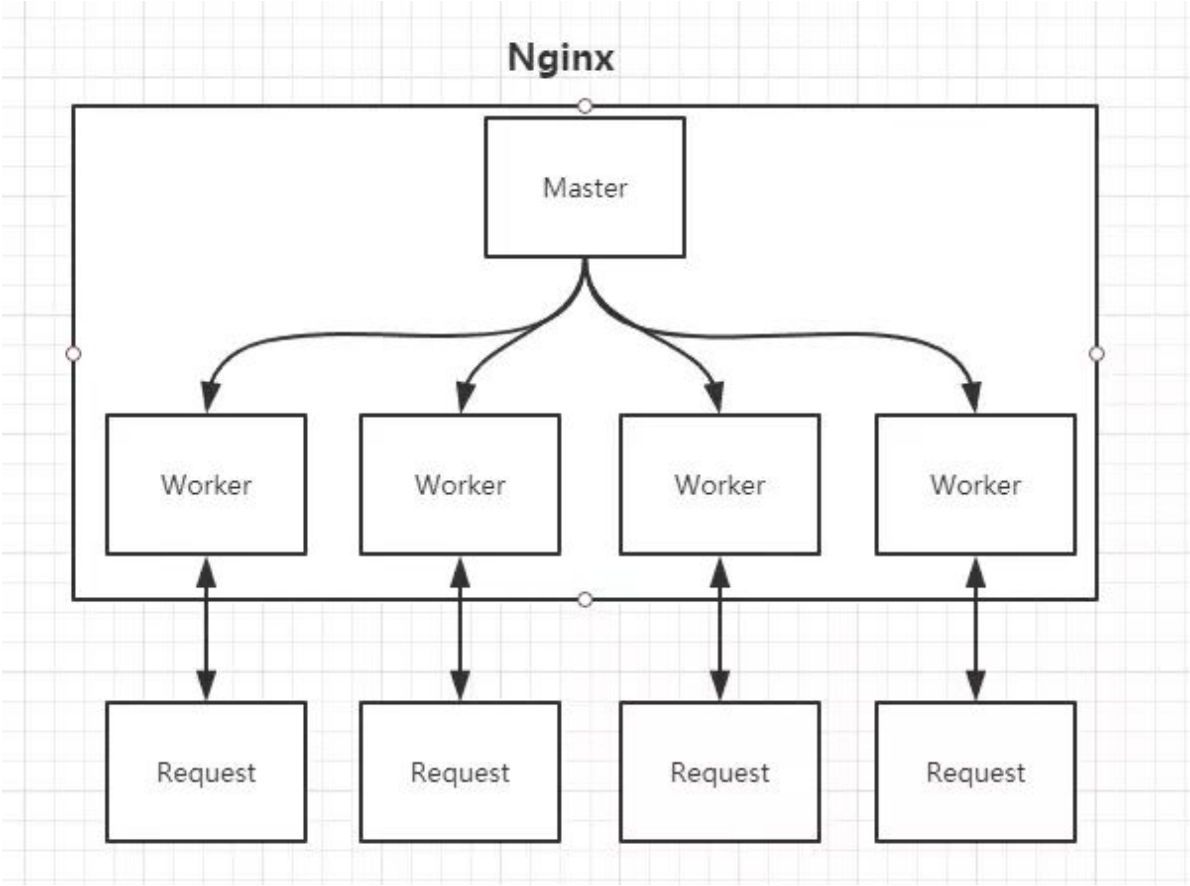
# 直接部署 Django

一种典型的部署方式如下图所示




## Nginx




首先启动 Nginx

## Deploy Django

现在我们可以准备将 Django 部署到服务器上了。

先将文件转移到服务器的某个文件夹中，这里就直接选择 `/home/train/site`

在课前准备中，我们已经安装了 conda 并配置好了虚拟环境，确认了 Nginx 可以正常启动。

进入网站目录，启动虚拟环境，图我们可以将 uWSGI 代理到 `10001` 端口

```
uwsgi --http :10001 --module mysite.wsgi
```

其中 module 参数是指出相应的 Python 模块。

现在我们就可以从外网通过端口来访问这个 Django 应用了。

我们也可以使用 配置文件来进行操作

```
[uwsgi]
socket = :9999
chdir = /home/train/site
module = mysite.wsgi
master = true
processes = 9
threads = 9
vacuum = true
```

然后

```
uwsgi --ini u.ini
```

其中，如果直接使用 uWSGI 应该指定 http，否则指定 socket。

关于 Django + uWSGI 的更多配置，详见 如何用 uWSGI 托管 Django 。

## 配置 Nginx 反向代理

我们可以先来个极简配置，代理一波 uWSGI

```
server{
listen 10002;
server_name 59.66.131.240;
  location / {
        include uwsgi_params;
        uwsgi_pass 127.0.0.1:9999;
        uwsgi_read_timeout 15;
  }
}
```

另外可以使用 Nginx 配置静态文件的代理。例如我们想部署一下这次后端作业的样例前端。

```
server {
listen 10001;
server_name 59.66.131.240;

location / {
        root /home/nana/build;
        index index.html;
}
```

有时我们会遇到 Nginx 没有权限读取内容的问题，表现为 HTTP 403，此时可以给相关文件加上权限，chown，甚至可以直接找到 `/etc/nginx/nginx.conf` ，将其首部的 `user` 改为 `root`，~~暴力解决一切问题~~。

在配置 Nginx 时，常用

```
sudo nginx -t
sudo cat /var/log/nginx/error.log
```

定位错误

例如我配置文件搞错了或者端口被占用，在这两个地方可以看到问题所在。

# 扩展阅读

其实 Django 的文档写得十分详细而友好。今天的课程主要是我就个人使用 Django 的经验进行的一些分享，要想了解 Django，还得自己来读文档。

还有些我认为比较常用的主题，今天大概没有时间了，把连接放到这里

- UnitTest：https://docs.djangoproject.com/zh-hans/4.0/intro/tutorial05/
- Middle Ware：https://docs.djangoproject.com/zh-hans/4.0/topics/http/middleware/
- Command：https://docs.djangoproject.com/zh-hans/4.0/howto/custom-management-commands/

有时你在使用 Django 作为纯后端的时候会感到有点难受

对于 Django API 开发来说，Django REST Framework 可以让开发流程方便不少。如果想要用 Django 摆烂软工等课程的**后端**，不妨使用 Django REST Framework。[Quick Start]