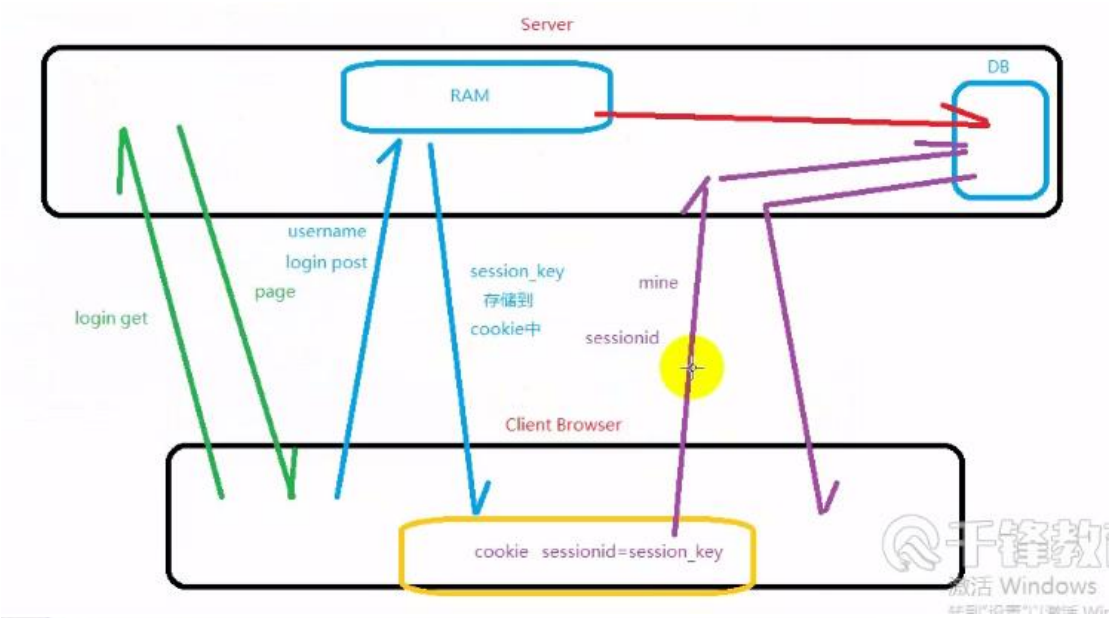
```python
def do_login(request):

    uname = request.POST.get('uname')

    response = HttpResponseRedirect(reverse('app:mine'))

    # response.set_cookie('uname', uname, max_age=60)
    response.set_signed_cookie('content', uname, "Rock")

    return response


def mine(request):

    # uname = request.COOKIES.get("content")

    try:

        uname = request.get_signed_cookie("content", salt="Rock")

        if uname:
            # return HttpResponse(uname)

            return render(request, 'mine.html', context={"uname": uname})
    except Exception as e:
        print("获取失败")

    return redirect(reverse('app:login'))


def logout(request):

    response = redirect(reverse("app:login"))

    response.delete_cookie("content")

    return response
```

- Cookie默认不支持中文
- 可以加盐
  - 加密
  - 获取的时候需要解密

- Session
  - 服务端会话技术
  - 数据存储在服务器中
  - 默认Session存储在内存中
  - Django中默认会把Session持久化到数据库中
  - Django中Session的默认过期时间是14天
  - 主键是字符串
  - 数据是使用了数据安全
    - 使用的base64
    - 在前部添加了一个混淆串
  - Session依赖于Cookie
- Token

编码

- ASCII
- Unicode




```python
def hello(request):
    return HttpResponse("Hello Two")


def login(request):

    if request.method == "GET":

        return render(request, 'two_login.html')
    elif request.method == "POST":

        username = request.POST.get("username")

        request.session["username"] = username

        return HttpResponse("登录成功")


def mine(request):

    username = request.session.get("username")

    return render(request, 'two_mine.html', context=locals())


def logout(request):

    response = redirect(reverse('two:mine'))

    response.delete_cookie('sessionid')

    return response
```

Two/urls.py | views.py | two_mine.html | GP1DjangoView.django_session [GP1DjangoView@localhost]

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Mine</title>
</head>
<body>

<h2>欢迎回来:{{ username }}</h2>

<a href="{% url 'two:logout' %}">退出</a>

</body>
</html>
```

```python
def logout(request):

    response = redirect(reverse('two:mine'))

    del request.session['username']

    # response.delete_cookie('sessionid')

    return response
```

将 cookie 和 session 彻底清理掉（Django 的数据库中也没了），进行以下操作：

```python
def logout(request):

    response = redirect(reverse('two:mine'))

    # del request.session['username']

    # response.delete_cookie('sessionid')

    # session cookie一起干掉
    request.session.flush()

    return response
```

服务器端会话技术,依赖于cookie

django中启用SESSION

settings中

INSTALLED_APPS:

'django.contrib.sessions'

MIDDLEWARE:

'django.contrib.sessions.middleware.SessionMiddleware'

每个HttpRequest对象都有一个session属性，也是一个类字典对象

常用操作

get(key,default=None) 根据键获取会话的值
clear() 清楚所有会话
flush() 删除当前的会话数据并删除会话的cookie
delete request['session_id'] 删除会话
session.session_key获取session的key

设置数据
request.session['user'] = username
数据存储到数据库中会进行编码使用的是Base64

令牌

编辑

（信息安全术语）

Token, 令牌，代表执行某些操作的权利的对象

访问令牌（Access token）表示访问控制操作主体的系统对象

邀请码，在邀请系统中使用

Token, Petri 网（Petri net）理论中的Token

密保令牌（Security token），或者硬件令牌，例如U盾，或者叫做认证令牌或者加密令牌，一种计算机身份校验的物理设备

会话令牌（Session token）,交互会话中唯一身份标识符

令牌化技术 (Tokenization), 取代敏感信息条目的处理过程

```
class Student(models.Model):
    s_name = models.CharField(max_length=16, unique=True)
    s_password = models.CharField(max_length=128)
    s_token = models.CharField(max_length=256)
```

```
<form action="{% url 'two:register' %}" method="post">

    <span>用户名:</span><input type="text" name="username" placeholder="请输入用户名">
    <br>
    <span>密码:</span><input type="text" name="password" placeholder="请输入你的银行卡密码">
    <br>

    <button>注册</button>

</form>


</body>
</html>
```

```
urlpatterns = [
    url(r'^hello/', views.hello, name='hello'),

    url(r'^login/', views.login, name='login'),

    url(r'^mine/', views.mine, name='mine'),

    url(r'^logout/', views.logout, name='logout'),

    url(r'^register/', views.register, name='register')
]
```

```python
def register(request):
    if request.method == "GET":
        return render(request, 'student_register.html')
    elif request.method == "POST":
        username = request.POST.get("username")

        password = request.POST.get("password")

        try:

            student = Student()

            student.s_name = username
            student.s_password = password

            student.save()

        except Exception as e:
            return redirect(reverse("two:register"))

        return HttpResponse("注册成功")
```

- Token
  - 服务端会话技术
  - 自定义的Session
  - 如果Web页面开发中，使用起来和Session基本一致
  - 如果使用在移动端或客户端开发中，通常以json形式传输，需要移动端自己存储Token，需要获取Token关联数据的时候，主动传递Token
- Cookie和Session,Token对比
  - Cookie使用更简洁，服务器压力更小，数据不是很安全
  - Session服务器要维护Session，相对安全
  - Token拥有Session的所有优点，自己维护略微麻烦，支持更多的终端

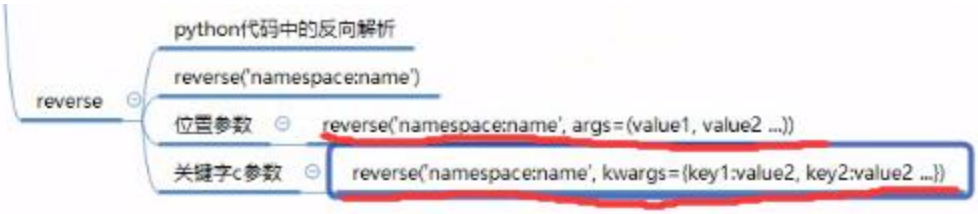
CSRF

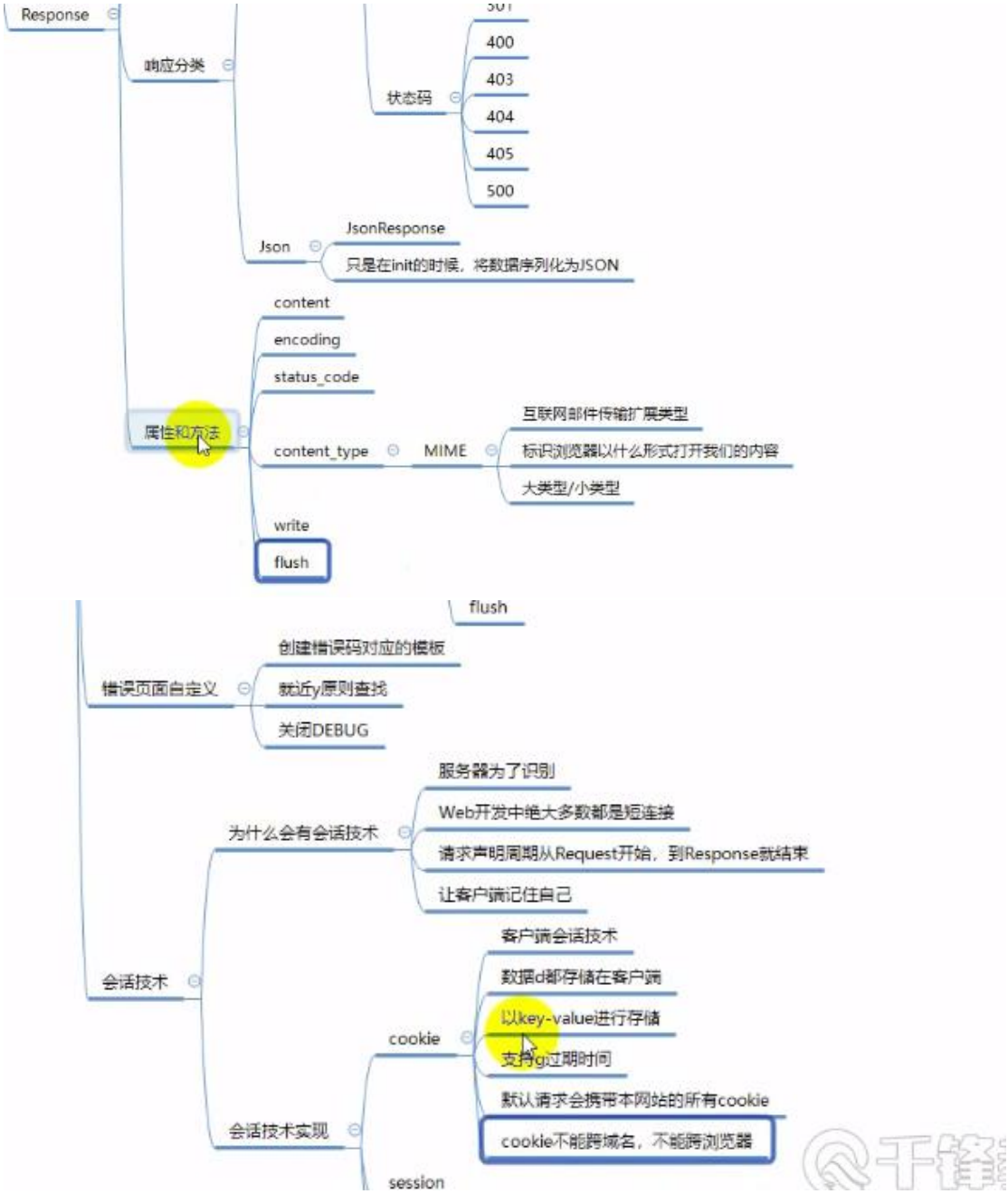
- 防跨站攻击
- 防止恶意注册，确保客户端是我们自己的客户端
- 使用了cookie中csrftoken 进行验证，传输
- 服务器发送给客户端，客户端将cookie获取过来，还要进行 编码转换 （数据安全）
- 如何实现的
  - 在我们 存在csrf_token 标签的页面中，响应会自动设置一个cookie， csrftoken
  - 当我们提交的时候，会自动验证csrftoken
  - 验证通过，正常执行以后流程，验证不通过，直接403

- MTV
  - 基本完成
  - Template不会再讲了
  - Views也不会再讲了
  - Model
    - Model关系
    - Model继承
- 高级
  - 第三方插件
  - 底层的部分原理
    - AOP 面向切面编程
      - 反扒
      - 安全
  - 文件上传
  - 前后端分离
    - RESTful
  - 日志
  - 后台管理
  - 前后端分离
    - RESTful
  - 日志
  - 后台管理
  - 用户角色，用户权限
  - 部署
  - 支付宝支付

算法

- 编码解码
  - Base64
  - urlencode
- 摘要算法，指纹算法，杂凑算法
  - MD5，SHA
    - MD5 默认是128的二进制
    - 32位的十六进制
    - 32位的Unicode
  - 单向不可逆的
  - 不管输出多长，输出都是固定长度
  - 只要输入有任意的变更，输出都会发生巨大的变化
- 加密
  - 对称加密
    - 一把钥匙
    - DES，AES
    - 加密解密效率高
    - 钥匙一旦丢失，所有数据就全玩完了
  - 非对称加密
    - 两把钥匙，成对的
    - 公钥和私钥
    - RSA，PGP
    - 安全性最高
    - 算法复杂，需要时间长
    - 支付宝，微信都是RSA




Response
响应分类
状态码
400
403
404
405
500
Json
JsonResponse
只是在init的时候，将数据序列化为JSON
属性和方法
content
encoding
status_code
content_type
MIME
互联网邮件传输扩展类型
标识浏览器以什么形式打开我们的内容
大类型/小类型
write
flush
flush
错误页面自定义
创建错误码对应的模板
就近y原则查找
关闭DEBUG
会话技术
为什么会有会话技术
服务器为了识别
Web开发中绝大多数都是短连接
请求声明周期从Request开始，到Response就结束
让客户端记住自己
会话技术实现
cookie
客户端会话技术
数据d都存储在客户端
以key-value进行存储
支持g过期时间
默认请求会携带本网站的所有cookie
cookie不能跨域名，不能跨浏览器
session


cookie默认不支持中文
会话技术实现
session
服务端会话技术
数据d都存储在服务端
支持过期
在django被持久化到了数据库中
默认做了数据安全，使用了BASE64
依赖于cookies
token
会话技术
自定义的Session
数据存储在服务器中
如果用Web网站开发中，效果基本等同于Session
主要应用场景是在移动端和专用客户端开发中，不支持Cookie的设备中


常见算法
编码
Base64
摘要，指纹，杂凑
单项不可逆
输出长度固定
输入变更一点点，输出发生很大的变更
MD5，SHA
hashlib
加密
对称加密
一把钥匙
非对称加密
一对钥匙
公钥
私钥
RSA，PGP
防跨站攻击
CSRF
防跨站攻击
django中内置CSRF插件
实现机制
页面中存在{% csrf_token %}时
在渲染的时候，会向Response中添加 csrftoken的Cookie
在t提交的时候，会被添加到请求体中，会被验证有效性

迁移

- 分两步实现
  - 生成迁移文件
  - 执行迁移文件
- 迁移文件的生成
  - 根据models文件生成对应的迁移文件
  - 根据models和已有迁移文件差别 生成新的迁移文件
- 执行迁移文件
  - 先去迁移记录查找，哪些文件未迁移过
    - app_label + 迁移文件名字
  - 执行未迁移的文件
  - 执行完毕，记录执行过的迁移文件
- 重新迁移
  - 删除迁移文件
  - 删除迁移文件产生的表
  - 删除迁移记录

Django模型

模型的对应关系

1 : 1

1 : N

M : N

常见的几种数据关系，django都提供了很好的支持

1 : 1

使用models.OneToOneField()进行关联

```
class Card(models.Model):
        person = models.OneToOneField(Person)
```

绑定卡与人的一对一关系，默认情况下，当人被删除的情况下，与人绑定的卡就也删除了，这个可以使用on_delete进行调整

```
on_delete
        models.CASCADE      默认值
        models.PROTECT      保护模式
        models.SET_NULL     置空模式
        models.SET_DEFAULT  置默认值
        models.SET()        删除的时候重新动态指向一个实体
访问对应元素   person.pcard
```

模型关系

- 1:1
  - 应用场景
    - 用于复杂表的拆分
    - 扩展新功能
  - Django中 OneToOneField
    - 使用的时候，关系声明还是有细微差别的
  - 实现
    - 使用外键实现的
    - 对外键添加了唯一约束
  - 数据删除
    - 级联表
      - 主表
      - 从表
      - 谁声明关系谁就是从表
      - 在开发中如何确认主从
        - 当系统遭遇不可避免毁灭时，只能保留一张表，这个表就是你的主表
- 1:M
- M:N
      - 从表
      - 谁声明关系谁就是从表
      - 在开发中如何确认主从
        - 当系统遭遇不可避免毁灭时，只能保留一张表，这个表就是你的主表
    - 默认特性（CASECADE）
      - 从表数据删除，主表不受影响
      - 主表数据删除，从表数据直接删除
    - PROTECT 受保护
      - 开发中为了防止误操作，我们通常会设置为此模式
      - 主表如果存在级联数据，删除动作受保护，不能成功
      - 主表不存在级联数据，可以删除成功
    - SET
      - SET_NULL
        - 允许为NULL
      - SET_DEFAULT
        - 存在默认值
      - SET()
        - 指定值
  - 1:M
  - M:N

- 指定值
    - 级联数据获取
        - 主获取从 隐性属性 默认就是级联模型的名字
        - 从获取主，显性属性，就是属性的名字
- 1:M
    - ForeignKey
    - 主从获取
        - 主获取从 隐性属性 级联模型_set
            - student_set Manager的子类
                - all
                - filter
                - exclude
                - Manager上能使用的函数都能使用
        - 从获取主
            - 显性属性
- M:N

- M:N
    - 实际上最复杂
    - 开发中很少直接使用多对多属性，而是自己维护多对多的关系
    - 产生表的时候会产生单独的关系表
        - 关系表中存储关联表的主键，通过多个外键实现的
        - 多个外键值不能同时相等
    - 级联数据获取
        - 从获取主
            - 使用属性，属性是一个Manager子类
        - 主获取从
            - 隐性属性
                - 也是Manager子类，操作和从操作主完全一样
    - 级联数据
        - add
        - remove
        - clear
        - set

- ManyRelatedManager
    - 函数中定义的类
    - 并且父类是一个参数
    - 动态创建

Models的面向对象

django中的数据库模块提供了一个非常不错的功能，就是支持models的面向对象，可以在models中添加Meta，指定是否抽象，然后进行继承

```
class Animal(models.Model):
    xxx
    class Meta:
        abstract = True/False

class Dog(Animal):
    xxx
```

- Django中模型支持继承
- 默认继承是会将通用字段放到父表中，特定字段放在自己的表中，中间使用外键连接
  - 关系型数据库关系越复杂，效率越低，查询越慢
  - 父类表中也会存储过多的数据
- 使用元信息来解决这个问题
  - 使模型抽象化
  - 抽象的模型就不会在数据库中产生映射了
  - 子模型映射出来的表直接包含父模型的字段

在企业中开发中

- model -> sql
  - 都可以使用
- sql -> model
  - django也提供了很好的支持
  - python manage.py inspectdb
    - 可以直接根据表生成模型
    - 元信息中包含一个属性 manage=False
- 如果自己的模型不想被迁移系统管理，也可以使用 manage=False进行声明

```
managed = False
db_table = 'Book'
```

```
(GP1) rock@Rock:~/GP1/Day05/ModelToSQL$ python manage.py inspectdb > App/models.py
```

ython Console | Terminal | 6: TODO

```
class UserModel(models.Model):
    u_name = models.CharField(max_length=16)
    # upload_to 相对路径，  相对于的是MEDIA_ROOT  媒体根目录
    u_icon = models.ImageField(upload_to='%Y/%m/%d/icons')
```

静态文件配置

在settings.py中最底下有一个叫做static的文件夹，主要用来加载一些模板中用到的资源，提供给全局使用

这个静态文件主要用来配置CSS,HTML,图片,字体文件等

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR,'static')
]
```

之后在模板中，首先加载静态文件，之后调用静态，就不用写绝对全路径了

静态文件配置

模板中的声明
{% load static%} 或 {% load staticfiles %}

在引用资源的时候使用
{% static 'xxx' %} xxx 就是相对于staticfiles_dirs的一个位置

图片上传

文件数据存储在request.FILES属性中

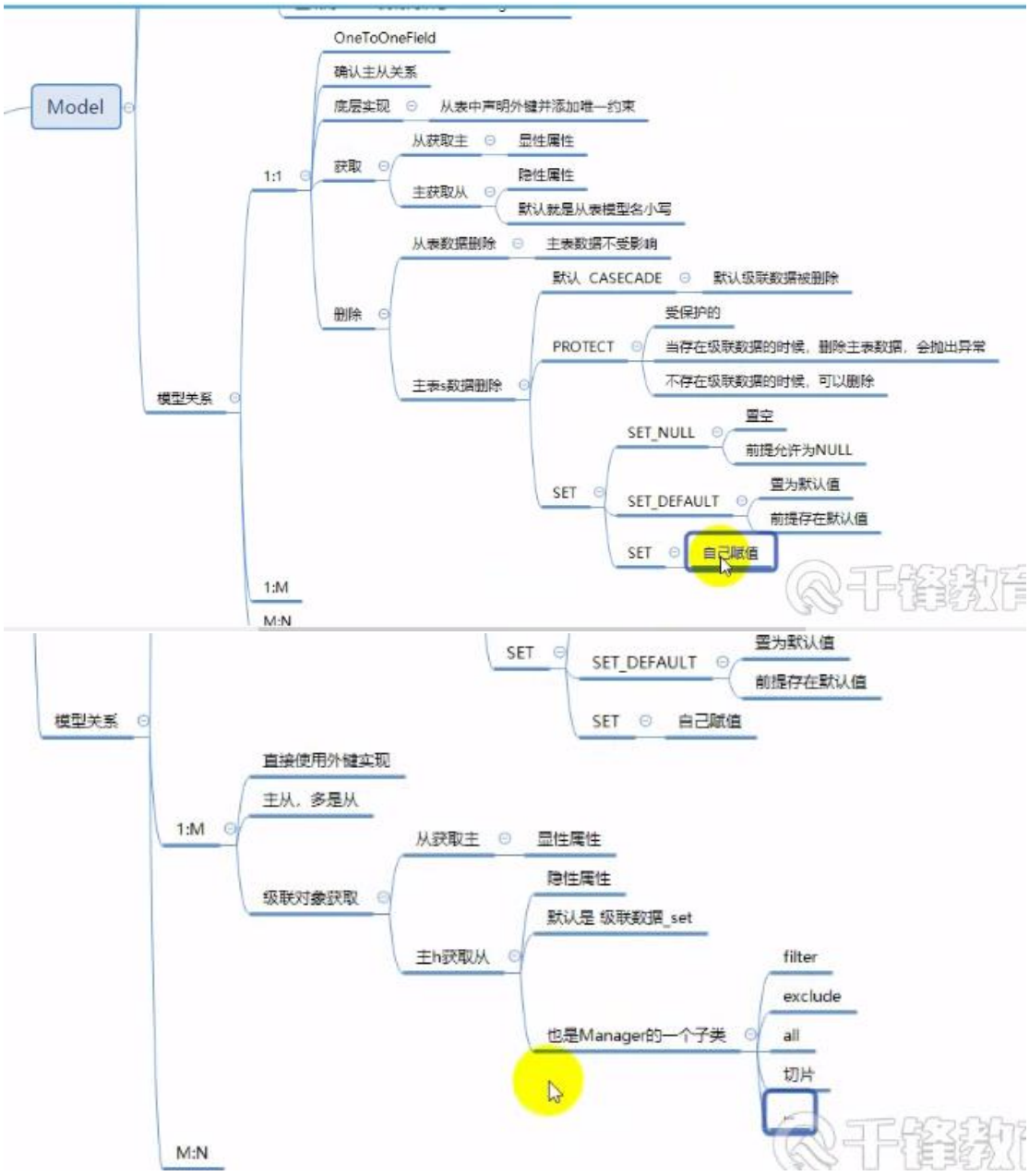
form表单上传文件需要添加enctype='multipart/form-data'
文件上传必须使用POST请求方式

存储

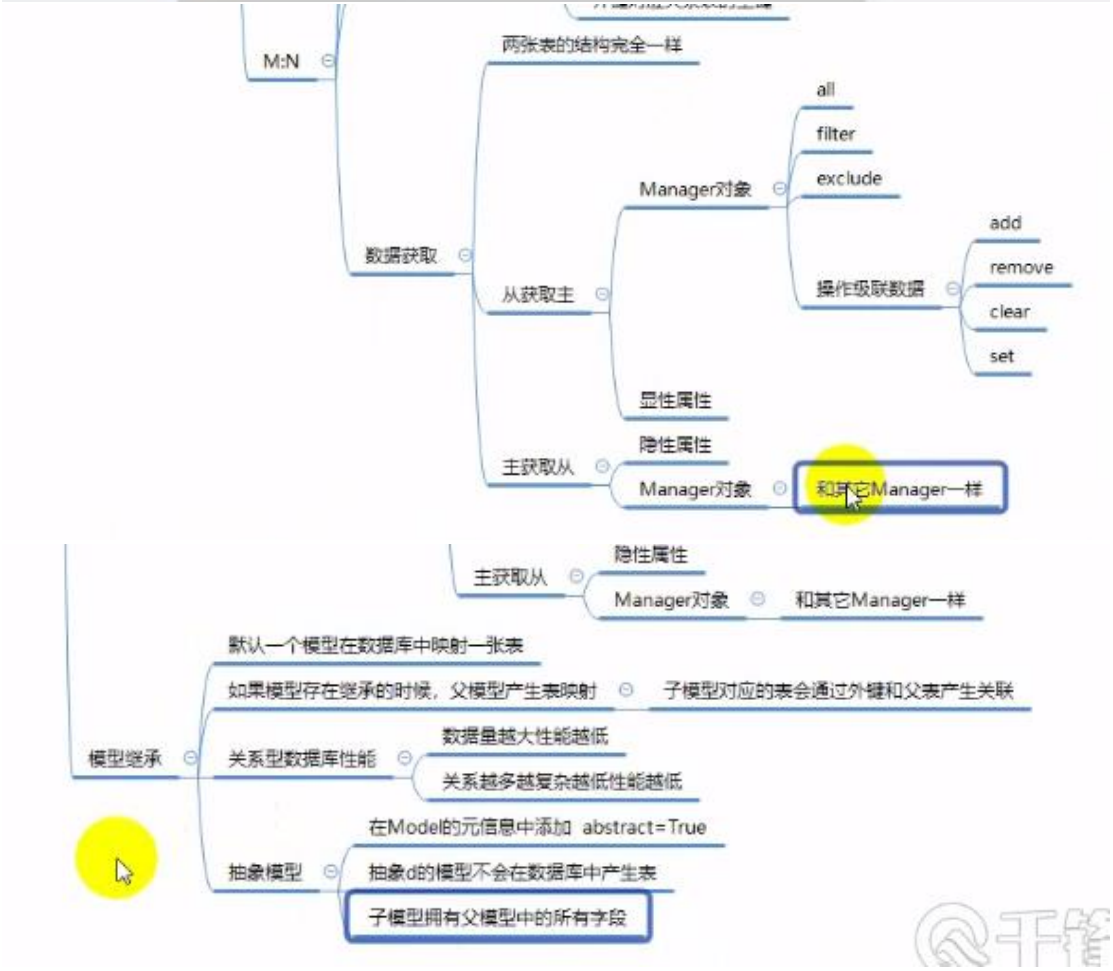
在static文件夹下创建uploadefiles用与存储接收上传的文件
在settings中配置，MEDIA_ROOT=os.path.join(BASE_DIR,r'static/uploadefiles')

在开发中通常是存储的时候，我们要存储到关联用户的表中

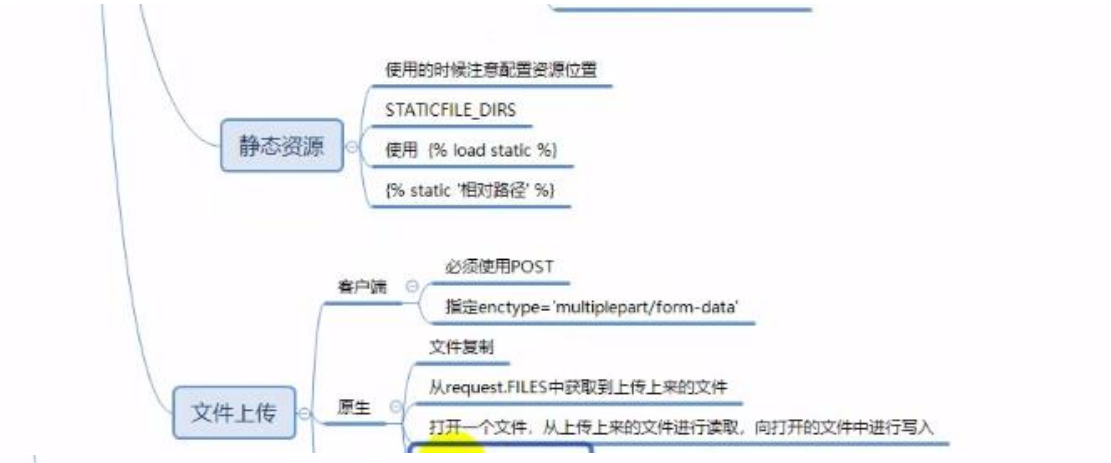
```
<form action="{% url 'app:upload_file' %}" method="post".enctype="multipart/form-data">
    {% csrf_token %}
    <span>文件:</span>
    <input type="file" name="icon">
    <br>
    <input type="submit" value="上传">
</form>
```


Model
模型关系
1:1
OneToOneField
确认主从关系
底层实现
从表中声明外键并添加唯一约束
获取
从获取主
显性属性
主获取从
隐性属性
默认就是从表模型名小写
删除
从表数据删除
主表数据不受影响
主表s数据删除
默认 CASECADE
默认级联数据被删除
PROTECT
受保护的
当存在级联数据的时候，删除主表数据，会抛出异常
不存在级联数据的时候，可以删除
SET
SET_NULL
置空
前提允许为NULL
SET_DEFAULT
置为默认值
前提存在默认值
SET
自己赋值
1:M
M:N
模型关系
SET
SET_DEFAULT
置为默认值
前提存在默认值
SET
自己赋值
1:M
直接使用外键实现
主从，多是从
级联对象获取
从获取主
显性属性
主h获取从
隐性属性
默认是 级联数据_set
也是Manager的一个子类
filter
exclude
all
切片
M:N


...
数据删除同一对一
ManyToManyField
两张以上的表的对应关系
会产生额外的关系表
表中使用多个外键实现
外键对应关系表的主键
M:N
两张表的结构完全一样
数据获取
从获取主
Manager对象
all
filter
exclude
操作级联数据
add
remove
clear
set
显性属性


M:N
两张表的结构完全一样
数据获取
从获取主
Manager对象
all
filter
exclude
操作级联数据
add
remove
clear
set
显性属性
主获取从
隐性属性
Manager对象
和其它Manager一样

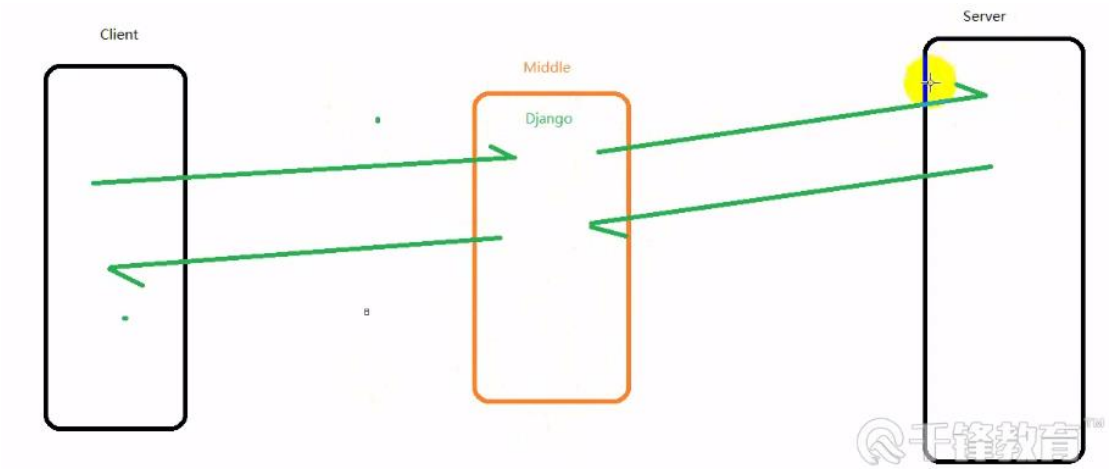
主获取从
隐性属性
Manager对象
和其它Manager一样
模型继承
默认一个模型在数据库中映射一张表
如果模型存在继承的时候，父模型产生表映射
子模型对应的表会通过外键和父表产生关联
关系型数据库性能
数据量越大性能越低
关系越多越复杂越低性能越低
抽象模型
在Model的元信息中添加 abstract=True
抽象d的模型不会在数据库中产生表
子模型拥有父模型中的所有字段


静态资源
使用的时候注意配置资源位置
STATICFILE_DIRS
使用 {% load static %}
{% static '相对路径' %}
文件上传
客户端
必须使用POST
指定enctype='multiplepart/form-data'
原生
文件复制
从request.FILES中获取到上传上来的文件
打开一个文件，从上传上来的文件进行读取，向打开的文件中进行写入


文件上传
客户端
必须使用POST
指定enctype='multiplepart/form-data'
原生
文件复制
从request.FILES中获取到上传上来的文件
打开一个文件，从上传上来的文件进行读取，向打开的文件中进行写入
每次写入记得 flush
Django内置
ImageField
依赖于 pillow
配置使用
settings中指定 MEDIA_ROOT
指定字段的upload_to
相对于MEDIA_ROOT的路径
支持时间格式化
%Y
%m
%d
%h
...


requests
请求库
实际上是urllib封装
python内置模块
使用场景
爬虫
MCS架构
添加中间层服务器

MCS 架构


Client
Middle
Django
Server