讲师：李振良（阿良）

今天课题：《Django入门与进阶》下

学院官网：www.ctnrs.com

阿良微信

添加微信好友

DevOps技术栈

关注微信公众号

Django 入门与进阶（下）

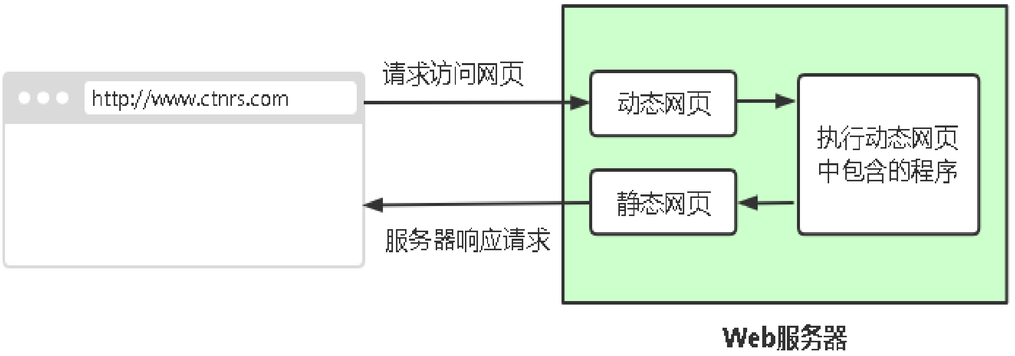
- Django ORM基本使用
- Django 多表操作
- Django 用户认证系统
- Django Session管理
- Django CSRF防御

Django ORM

- 了解静态网站与动态网站
- ORM 是什么
- Model（模型类）
- 使用MySQL数据库
- ORM增删改查
- Django内置管理后台
- 模型中的Meta类与方法
- 模型类常用字段与选项
- QuerySet对象序列化

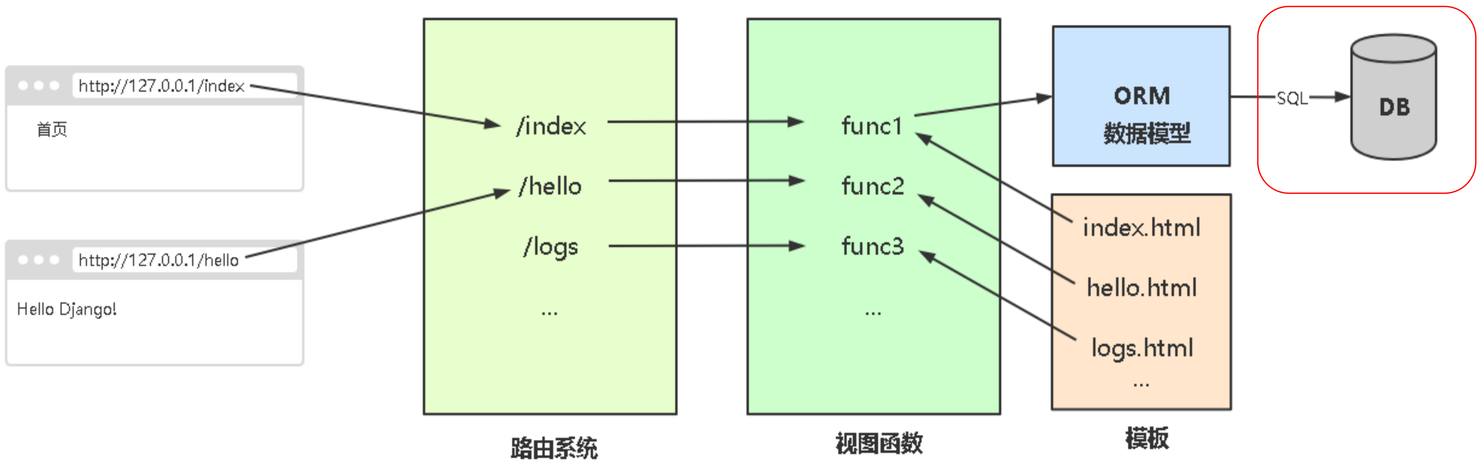
了解静态网站与动态网站

动态网站



静态网站

了解静态网站与动态网站

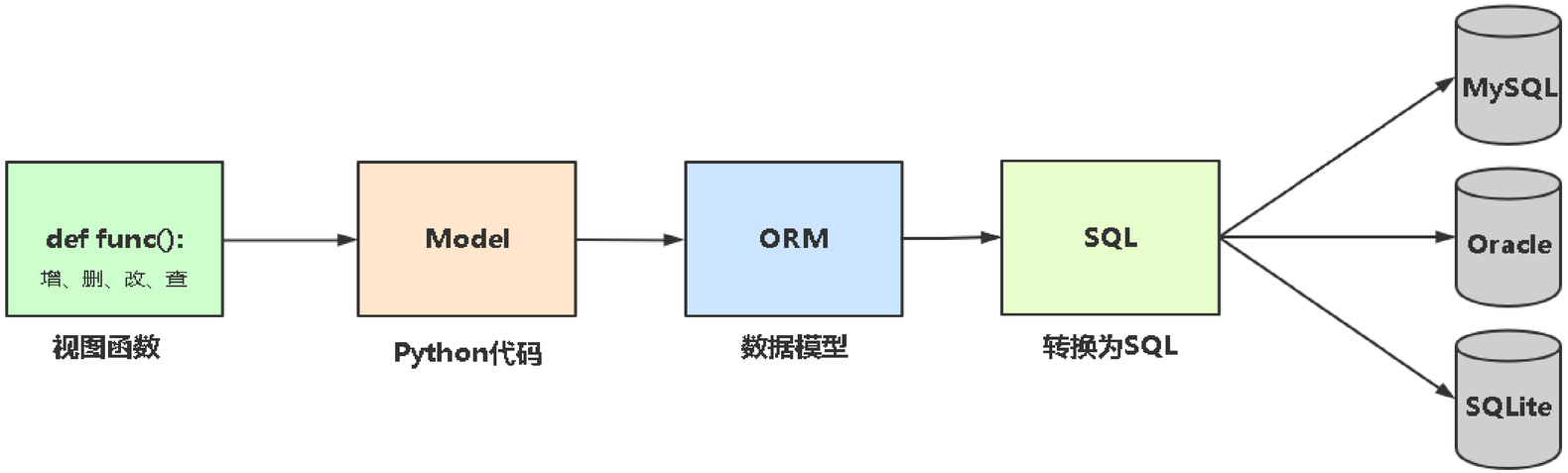


ORM是什么

对象关系映射（Object Relational Mapping，ORM）：是一种程序设计技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。简单来说就是在编程语言中实现的一种虚拟对象数据库。我们对虚拟对象数据库进行操作，它会转换成具体的SQL去操作数据库，这样一来我们就不需要学习复杂的SQL语句了。

ORM优势：不必熟悉复杂的SQL语句，容易上手，避免新手写SQL效率问题。

ORM是什么


def func():
增、删、改、查
视图函数
Model
Python代码
ORM
数据模型
SQL
转换为SQL
MySQL
Oracle
SQLite

Model（模型类）

1、使用模型类定义一个User表，包含多字段

```
# myapp/models.py
class User(models.Model):
    user = models.CharField(max_length=30)    # 用户名
    name = models.CharField(max_length=30)  # 姓名
    sex = models.CharField(max_length=10)      # 性别
    age = models.IntegerField()                        # 年龄
    label = models.CharField(max_length=100) # 标签
```

2、在settings.py配置文件中INSTALLED_APPS列表添加APP名称

```
INSTALLED_APPS = [
    #...
    'myapp',
]
```

3、将模型类生成具体的数据库表

```
# 生成迁移文件
python manage.py makemigrations
# 执行迁移文件生成表
python mange.py migrate
```

4、进入数据库查看表

生成表名的默认格式：应用名_模型类名小写

使用MySQL数据库

1、使用docker启动一个mysql实例

```
docker run -d \
--name db \
-p 3306:3306 \
-v mysqldata:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=123456 \
mysql:5.7 --character-set-server=utf8
```

2、使用pip工具安装pymysql模块

```
pip install pymysql
```

3、修改django默认连接数据库

```
# devops/settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'test',
        'USER': 'root',
        'PASSWORD': '123456',
        'HOST': '192.168.31.62',
        'PORT': '3306',
    }
}
```

4、指定数据库驱动

```
# myapp/__init__.py
import pymysql
pymysql.install_as_MySQLdb()
```

5、执行迁移文件生成表

```
python mange.py migrate
```

ORM增删改查

增：

```
from myapp.models import User
def user_add(request):
    User.objects.create(
        user='aliang',
        name='阿良',
        sex='男',
        age='30',
        label="IT,讲师,老司机"
    )
    return HttpResponse("用户添加成功！")

或者用save方法保存：
obj = User(
    user=user,
    name=name,
    sex=sex,
    age=age,
    label=label
)
obj.save()
```

查：

```
def user_list(request):
    user_list = User.objects.all()
    return render(request, "user.html", {'user_list': user_list})


# 获取所有数据

User.objects.all()

# 加条件获取数据

User.objects.filter(user='amei')

User.objects.filter(age__gt=28)

# 获取单条数据

User.objects.get(id=2)
```

ORM增删改查

改：

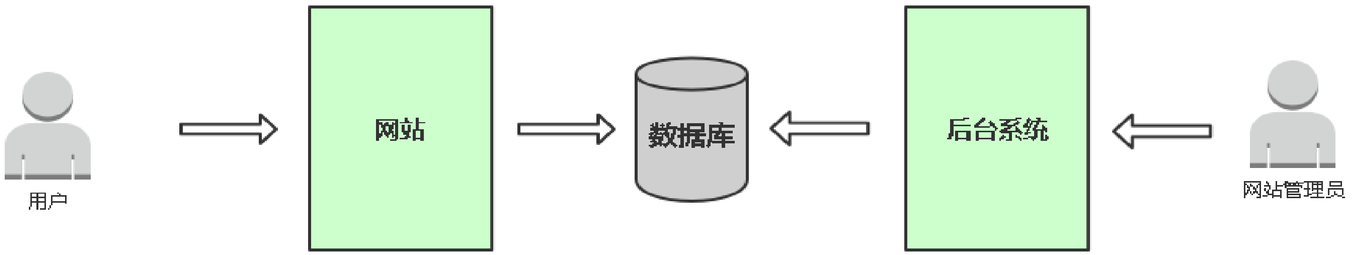
```
User.objects.filter(user='amei').update(age=27,label='公关,漂亮,喜欢购物')
或者
obj = User.objects.get(user='amei')
obj.age = 25
obj.save()
```

删：

```
User.objects.filter(id=3).delete()
或者
obj = User.objects.get(id=2)
obj.delete()
```

内置管理后台

管理后台：一个网站一般都会开发一个后台系统，为管理员提供一种更简单的数据库操作方式。



内置管理后台

1、访问URL

```
from django.contrib import admin # 内建管理后台功能
from django.urls import path
urlpatterns = [
    path('admin/', admin.site.urls),  # 内建管理后台访问地址
]
```

2、创建管理员账号

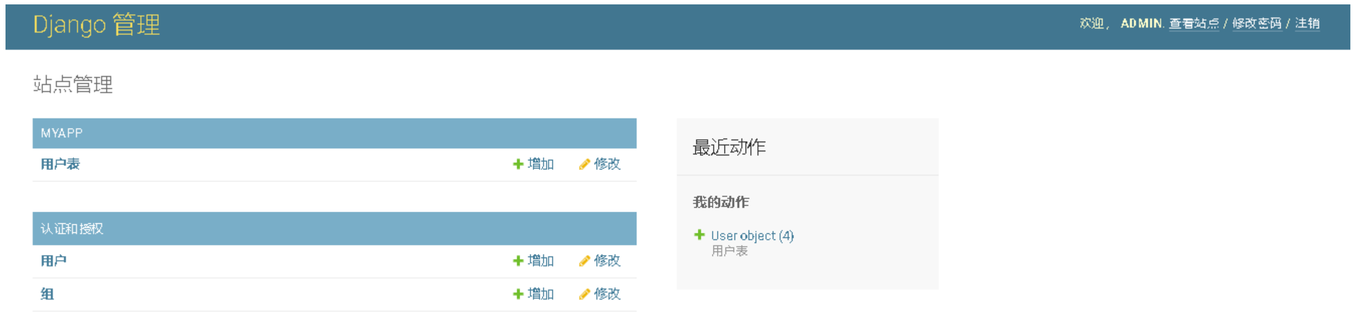
```
python manage.py createsuperuser
```

3、注册模型

```
# myapp/admin.py
from django.contrib import admin
from myapp import models
admin.site.register(models.User)
```

4、设置语言和时区

```
# devops/settings.py
LANGUAGE_CODE = 'zh-hans'
TIME_ZONE = 'Asia/Shanghai'
USE_I18N = True
USE_L10N = True
USE_TZ = False
```



模型中的Meta类与方法

Django模型类的Meta是一个内部类,它用于定义一些Django模型类的行为特性。

以下是该常用属性：

元选项	描述
app_label	指定APP名称，当模型类不在默认的APP的models.py文件中，这时需要指定模型类是属于哪个APP。
db_table	指定生成的数据库表名称，默认是”应用名_模型名”
ordering	对象的默认顺序，值是一个列表或元组。 元素是一个字符串表示字段名，元素前面带减号表示倒序 没有表示升序，问号表示随机排序 例如ordering = ["-sex"]
verbose_name	定义一个易读的模型名称，默认会加一个复数s
verbose_name_plural	定义一个易读的模型名称，不带复数s

模型中的Meta类与方法

```
class User(models.Model):
    user = models.CharField(max_length=30)
    name = models.CharField(max_length=30)
    sex = models.CharField(max_length=30)
    age = models.IntegerField()
    label = models.CharField(max_length=100)

    class Meta:
        app_label = "myapp"         # 指定APP名称
        db_table = "myapp_user"  # 自定义生成的表名
        verbose_name = "用户表"  # 对象的可读名称
        verbose_name_plural = "用户表"    # 名称复数形式
        ordering = ["sex"]       # 对象的默认顺序，用于获取对象列表时
    def __str__(self):
        return self.name      # 返回字段值
```

模型类常用字段

字段类型	描述
AutoField(**options)	ID自动递增，会自动添加到模型中
BooleanField(**options)	布尔值字段（true/false），默认值是None
CharField(max_length=None[,**options])	存储各种长度的字符串
EmailField([max_length=254,**options])	邮件地址，会检查是否合法
FileField([upload_to=None,max_length=100,**options])	保存上传文件。upload_to是保存本地的目录路径
FloatField([**options])	浮点数
IntegerField([**options])	整数
GenericIPAddressField(protocol=’both’, unpack_ipv4=False, **options)	IP地址
TextField([**options])	大文本字符串
URLField([max_length=200,**options])	字符串类型的URL
DateTimeField([auto_now=False,auto_now_add=False,**options])	日期和时间 • auto_now=True时，第二次保存对象时自动设置为当前时间。用于最后一次修改的时间戳，比如更新。 • auto_now_add=True时，第一次创建时自动设置当前时间。用于创建时间的时间戳，比如新增。
DateField([auto_now=False,auto_now_add=False,**options])	日期
TimeField([auto_now=False,auto_now_add=False,**options])	时间

模型类常用字段选项

选项	描述
null	如果为True，字段用NULL当做空值，默认False
blank	如果为True，允许为空，默认False
db_index	如果为True，为此字段建立索引
default	字段的默认值
primary_key	如果为True，设置为主键
unique	如果为True，保持这个字段的值唯一
verbose_name	易读的名称，管理后台会以这个名称显示

QuerySet序列化

序列化：将Python对象转为传输的数据格式

反序列化：将传输的数据格式转为Python对象

ORM查询返回的是QuerySet对象，如果你要提供数据接口，这显然是不行的。

有两种方法可以转为JSON字符串：

- 使用内建函数 serializers
- 遍历QuerySet对象将字段拼接成字典，再通过json库编码

```
from django.core import serializers
obj = User.objects.all()
data = serializers.serialize('json', obj)
```

```
import json
obj = User.objects.all()
d = {}
for i in user_list:
    d['name'] = i.name
    d['user'] = i.user
    d['label'] = i.label
data = json.dumps(d)
```

Django 多表操作

- 一对一
- 一对多
- 多对多

多表关系

常见的数据模型关系有：

- 一对一(one-to-one)，OneToOneField
- 一对多，多对一(one-to-many)，ForeignKey
- 多对多（many-to-many），ManyToManyField

一对一

一对一：一个表中的每条记录对应另一个表中的每条记录，使用OneToOneField建立关系。

例如：一个人对应一个身份证号，一个身份证号也对应一个人

应用场景：当一个表想扩展字段，最常用的方式就是在这个表添加一个对一关系

一对一：创建模型关系

```
class User(models.Model):
    user = models.CharField(max_length=30, verbose_name="用户名")
    name = models.CharField(max_length=30, verbose_name="姓名")
    sex = models.CharField(max_length=30, verbose_name="性别")
    age = models.IntegerField(verbose_name="年龄")
    label = models.CharField(max_length=100, verbose_name="标签")
class IdCard(models.Model):
    number = models.CharField(max_length=20, verbose_name="卡号")
    address = models.CharField(max_length=50, default="北京")
    user = models.OneToOneField(User)  # 定义一对一的模型关系
```

一对一：增删改查

增:

方式1：

```
user_obj = User.objects.create(user='alan',name='阿兰',sex='女',age='25',label="运营,漂亮,喜欢购物")
IdCard.objects.create(user=user_obj, number="456789", address="北京")
```

方式2：

```
user = User()
user.user='xiaoming'
user.name = "阿兰"
user.sex = '女'
user.age = 25
user.label = "运营,漂亮,喜欢购物"
user.save()
```

向已有用户添加身份证信息：

```
user_obj = User.objects.get(user="aliang")
IdCard.objects.create(user=user_obj, number="123456789", address="北京")
```

注：操作模型类记得先导入 from myapp.models import User,IdCard

一对一：增删改查

查：

反向查询：通过用户查到身份证信息（user->idcard）

```
user = User.objects.get(user="aliang")
print(user.idcard.number)
print(user.idcard.address)
```

正向查询：从身份证表查用户（idcard->user）

```
idcard = IdCard.objects.get(user_id=1)
print(idcard.user.user)
print(idcard.user.name)
```

一对一：增删改查

改：

```
user_obj = User.objects.get(user="aliang")
# 修改身份证信息
user_obj.idcard.address="河南"
user_obj.idcard.save()
# 修改用户信息
user_obj.age = 38
user_obj.save()
```

删：

```
User.objects.filter(user="alan").delete()
```

一对多

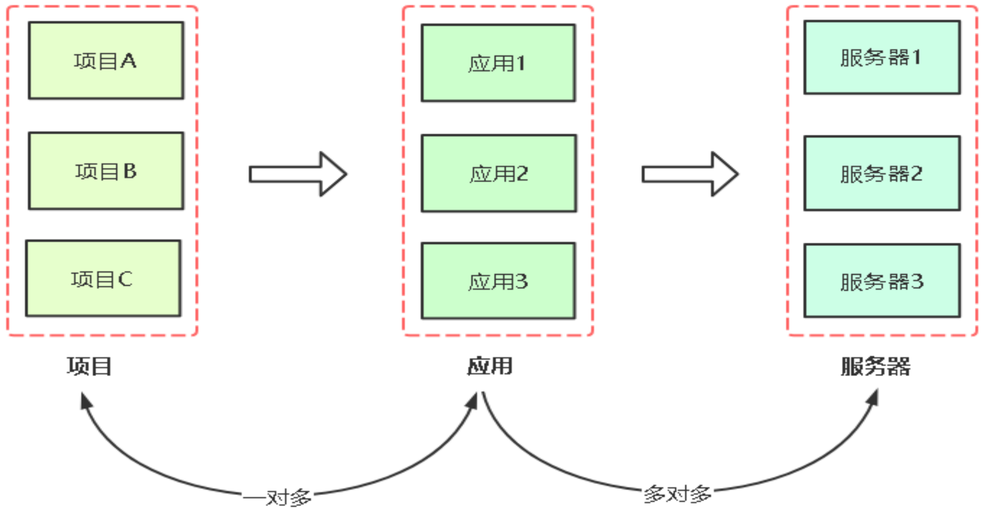
一对一是表与表之间的关系，而一对多、多对多是表与表中数据的关系

一对多：A表中的某个记录对应B表中的多条记录，使用**ForeignKey**建立关系。

例如：项目部署涉及表

一个项目有多个应用，一个应用只能属于一个项目

一个应用部署到多台服务器，一个服务器部署多个应用



一对多：创建模型关系

```
class Project(models.Model):
    name = models.CharField(max_length=30)
    describe = models.CharField(max_length=100, null=True)
    datetime = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return  self.name

    class Meta:
        db_table = 'project'
        verbose_name_plural = '项目'

class App(models.Model):
    name = models.CharField(max_length=30)
    describe = models.CharField(max_length=100, null=True)
    datetime = models.DateTimeField(auto_now_add=True)
    project = models.ForeignKey(Project) # 定义一对多的模型关系
    def __str__(self):
        return  self.name

    class Meta:
        db_table = 'app'
        verbose_name_plural = '应用'

class Server(models.Model):
    hostname = models.CharField(max_length=30)
    ip = models.GenericIPAddressField()
    describe = models.CharField(max_length=100, null=True)
    def __str__(self):
        return  self.hostname

    class Meta:
        db_table = 'server'
        verbose_name_plural = '服务器'
```

一对多：增删改查

向项目表添加已知的项目名称：

```
Project.objects.create(name="电商项目",describe="电商项目描述...")
Project.objects.create(name="在线教育项目",describe="在线教育项目描述...")
Project.objects.create(name="大数据项目",describe="大数据项目描述...")
```

创建新应用并加入到项目中：

```
project_obj = Project.objects.get(name="电商项目")
App.objects.create(name="product",describe="商品服务",project=project_obj)
```

注：操作模型类记得先导入 from myapp.models import Project,App

一对多：增删改查

正向查询：通过应用名称查询所属项目（app->project）

查询某个应用所属项目：

```
app = App.objects.get(name="product") # 获取应用
app.project.name  # 根据获取的应用，查询对应项目名称
```

查询所有应用所属项目：

```
app_list = App.objects.all()
for i in app_list:
print(i.name, i.project.name, i.project.describe)
```

示例：

```
def release(request):
    app = App.objects.all()
    return  render(request, "release.html", {"app":app})
```

```
{% for i in app_list %}
    {{ i.id }}
    {{ i.name }}
    {{ i.describe }}
    {{ i.project_id }}
    {{ i.project.name }}
{% endfor %}
```

一对多：增删改查

反向查询：通过项目名称查询有哪些应用（project->app）

查询某个项目有哪些应用：

```
project = Project.objects.get(name="电商项目") # 获取项目
project.app_set.all()  # 根据获取的项目，查询所有应用
```

查询所有项目有哪些引用：

```
project = Project.objects.all()
for i in project:
    print(i.name, i.app_set.all())
```

示例：

```
def app(request):
    project_list = Project.objects.all()
    return render(request, "app.html", {"project_list": project_list})
```

```
{% for i in project_list %}
    {{ i }}
    {% for x in i.app_set.all %}
        {{ x }}
    {% endfor %}
{% endfor %}
```

一对多：应用案例

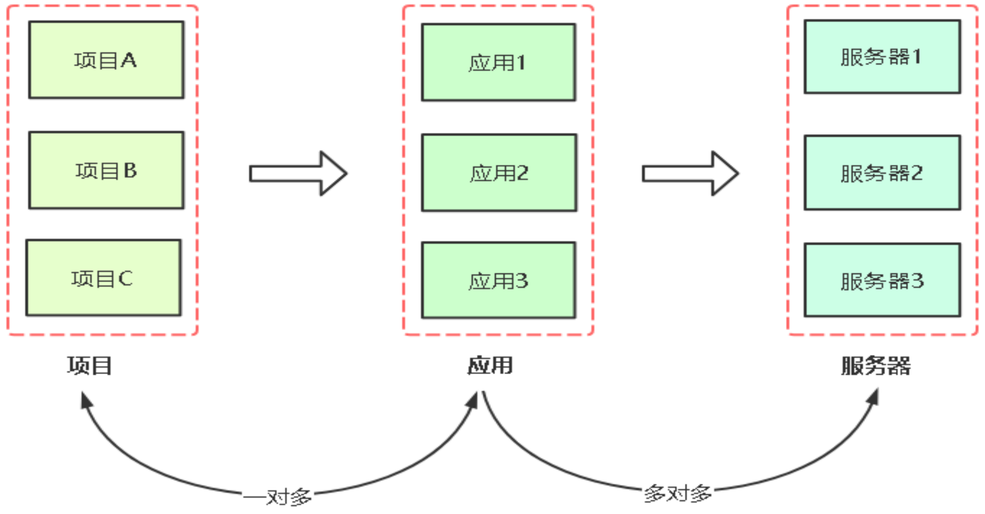
页面开发案例：

- 页面可添加应用并可选项目
- 展示所有应用名称、应用描述、所属项目

多对多

多对多：A表中的某个记录对应B表中的多条记录，B表中的某个记录对应A表中多条记录。使用**ManyToManyField**建立关系。

例如：一个应用部署到多台服务器，一个服务器部署多个应用



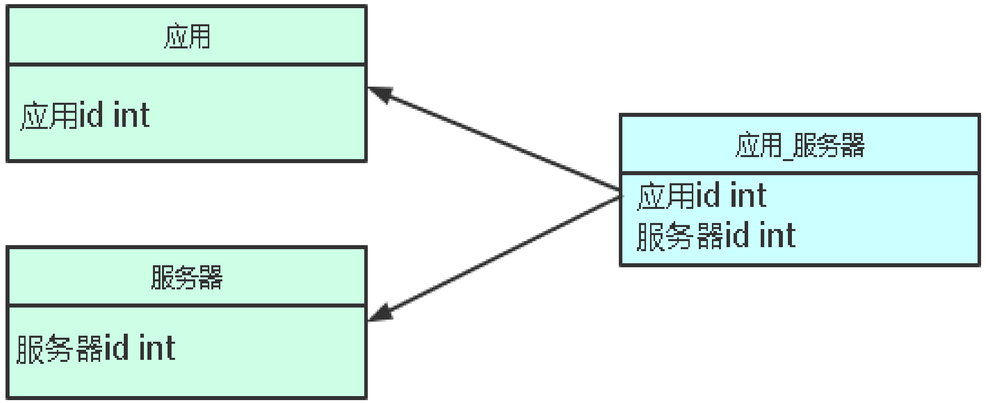
多对多：创建模型关系

```
class Server(models.Model):
    hostname = models.CharField(max_length=30)
    ip = models.GenericIPAddressField()
    describe = models.CharField(max_length=100, null=True)
    app = models.ManyToManyField(App)
    def __str__(self):
        return  self.hostname

    class Meta:
        db_table = 'server'
        verbose_name_plural = '服务器'
```

多对多：创建模型关系

Django会自动创建一个表来管理多对多关系，称为**中间表**；这个中间表的名称使用多对多的名称和包含这张表的模型的名称生成，也可以使用db_table选项指定这个中间表名称。



多对多：增删改查

添加服务器：

```
Server.objects.create(hostname="ec-test1", ip="192.168.1.10", describe="电商项目测试服务器1")
Server.objects.create(hostname="ec-test2", ip="192.168.1.11", describe="电商项目测试服务器2")
Server.objects.create(hostname="bigdata-test1", ip="192.168.1.11", describe="大数据项目测试服务器1")
```

部署一个应用到指定服务器：

```
project_obj = Project.objects.get(name="电商项目")
app = App.objects.create(name="portal",describe="前端服务",project=project_obj)
server = Server.objects.get(hostname="ec-test1")
server.app.add(app)  # 将服务器关联到应用
```

多对多：增删改查

正向查询：查询服务器部署了哪些应用（server->app）

查询某台服务器部署了哪些应用：

```
server = Server.objects.get(hostname="ec-test1")
server.app.all()
```

查询所有服务器部署了哪些引用：

```
server_list = Server.objects.all()
for i in server_list:
    print(i.hostname, i.app.all())
```

示例：

```
def server(request):
    server_list = Server.objects.all()
    return render(request, "server.html", {"server_list": server_list})
```

```
{% for i in server_list %}
    {{ i }}
    {% for x in i.app.all %}
    {{ x.name }}
    {% endfor %}
{% endfor %}
```

多对多：增删改查

反向查询：查看某个应用部署到哪些服务器通过项目名称查询有哪些应用（app->server)

查询某个应用部署到哪些服务器：

```
app = App.objects.get(name="portal")
app.server_set.all()
```

查询所有应用部署到哪些服务器：

```
for i in app_list:
    print(i.name, i.server_set.all())
```

示例：

```
def app_server(request):
    app_list = App.objects.all()
    return render(request, "app.html", {"app_list": app_list})
```

```
{% for i in app_list %}
    {{ i }}
    {% for x in i.server_set.all %}
        {{ x }}
    {% endfor %}
{% endfor %}
```

多对多：中间表关系操作

增加：

server = Server.objects.get(hostname= “ec-test1") # 获取已有的服务器

server.app.add(3) # 将应用id3关联该服务器

server.app.add(1,2,3) # 将应用id1、2、3关联该服务器

删除：

server.app.remove(3) # 将应用id3与该服务器取消关联

清空:

server.app.clear() # 将该服务器取消所有应用关联

多对多：综合案例

页面开发案例：展示所有应用、所属项目、部署服务器列表

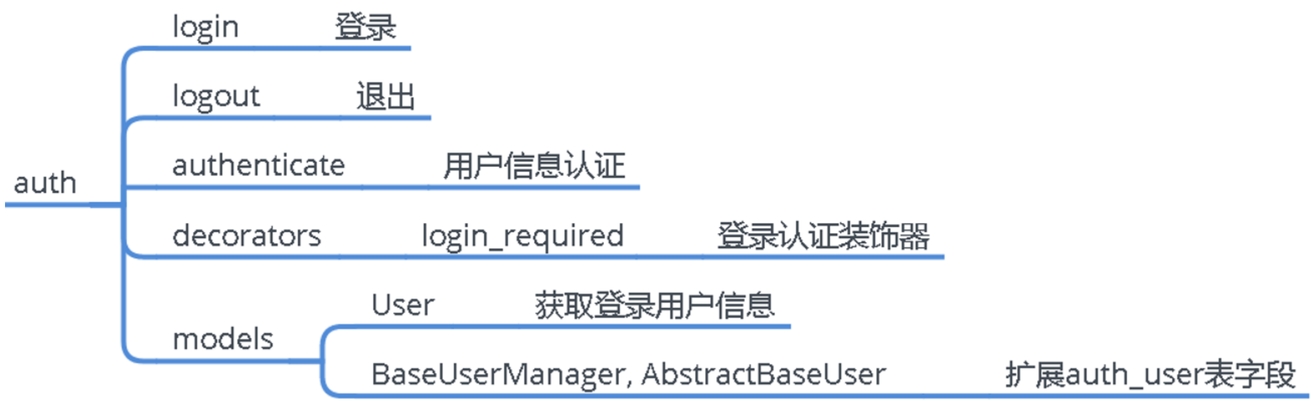
Django 用户认证系统

- 内置用户认证系统
- auth模块使用

用户认证系统

Django内置一个用户认证系统，使用auth模块实现。

auth模块提供了登录、注册、效验、修改密码、注销、验证用户是否登录等功能。



用户认证系统

Django默认创建的数据库表：

- auth_user： 用户表
- auth_user_groups： 用户所属组的表
- auth_user_user_permissions： 用户权限表
- auth_group： 用户组表
- auth_group_permissions： 用户组权限表
- auth_permission ： 存放全部权限的表，其他的表的权限都是从此表中外键连接过去的
- django_session： 保存HTTP状态
- django_migrations： 数据库迁移记录

auth模块：login()

示例：登录认证

```
from django.contrib import auth

def login(request):
    if reqeust.method == 'GET':
        return render(request, 'login.html')
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')

        # 1. 对用户数据验证
        user = auth.authenticate(username=username, password=password)
        # 如果效验成功，返回一个用户对象，否则返回一个None

        if user:
            # 2. 验证通过后，将request与用户对象（包含session）传给login()函数
            auth.login(request, user)
            # 3. 跳转到首页
            return redirect("/")
        else:
            msg = "用户名或密码错误！"
            return render(request, 'login.html',{'msg': msg})
```

登录表单：

```
<form method="post">
    用户名: <input type="text" name="username"><br>
    密码: <input type="text" name="password"><br>
    <button type="submit">登录</button>
    <span style="color: red">{{ msg }}</span>
</form>
```

auth模块：logout()

示例：退出登录

```
from django.contrib.auth import login,logout

def logout(request):
    # 清除当前用户的session信息
    auth.logout(request)
    return redirect('/login')
```

auth模块：login_required装饰器

login_required装饰器：判断用户是否登录，如果没有登录引导至登录页面，登录成功后跳转到目的页面。

示例：

```
from django.contrib.auth.decorators import login_required
@login_required()
def index(request):
    return render(request, 'index.html')
```

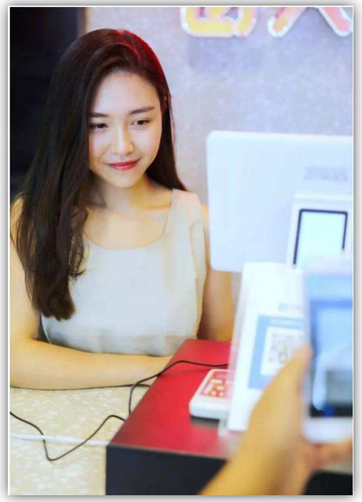
在settings.py文件设置没有登录默认跳转页面：

```
LOGIN_URL = '/login/'
```

Django Session管理

- Session与Cookie是什么
- Django使用Session
- 自己实现用户登录认证

Session与Cookie是什么



带女朋友买衣服场景

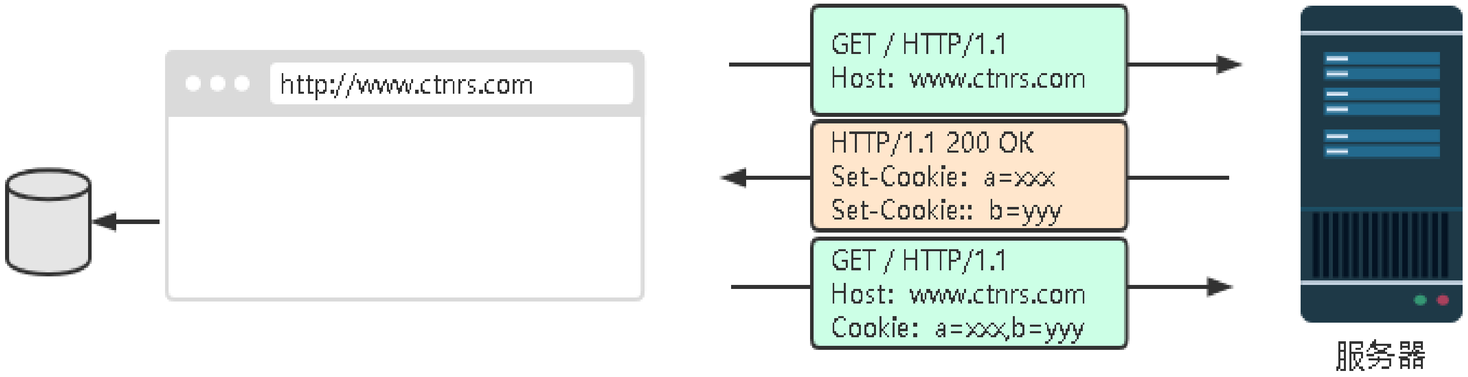
Session与Cookie是什么

就像你去电商平台购物一样，而网站采用是HTTP协议，它本身就是一个无状态的，是记不住你上次来做了什么事，那怎么记住每个用户呢。
于是，服务器给每个用户贴了一个小纸条，上面记录了服务器给我们返回的一些信息。然后服务器看到这张小纸条就知道我们是谁了。

这个小纸条就是Cookie。那么Cookie怎么工作的呢?

1. 浏览器第一次访问服务器时，服务器此时肯定不知道它的身份，所以创建一个独特的身份标识数据，格式为key=value，放入到Set-Cookie字段里，随着响应报文发给浏览器。
2. 浏览器看到有Set-Cookie字段以后就知道这是服务器给的身份标识，于是就保存起来，下次请求时会自动将此key=value值放入到Cookie字段中发给服务器。
3. 服务器收到请求报文后，发现Cookie字段中有值，就能根据此值识别用户的身份然后提供个性化的服务。

Session与Cookie是什么



有了Cookie实现了有状态这一需求，那为什么又来一个Session呢?

试想一下，如果将用户账户的一些信息都存入Cookie中的话，一旦信息被拦截，那么所有的账户信息都会可能被泄露丢，这是不安全的。所以就出现了Session，在一次会话中将重要信息保存在Session中，浏览器只记录SessionId一个SessionId对应一次会话请求。

Session与Cookie是什么

Django使用Session

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

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',
]
```

Django使用Session

在settings.py配置文件中设置客户端Cookie：

参数	描述
SESSION_COOKIE_NAME ＝ "sessionid"	Session的cookie保存在浏览器上时的key 即：sessionid＝随机字符串（默认）
SESSION_COOKIE_PATH ＝ "/"	Session的cookie保存的路径（默认）
SESSION_COOKIE_DOMAIN = None	Session的cookie保存的域名（默认）
SESSION_COOKIE_SECURE = False	是否Https传输cookie（默认）
SESSION_COOKIE_HTTPONLY = True	是否Session的cookie只支持http传输（默认）
SESSION_COOKIE_AGE = 1209600	Session的cookie失效日期（2周）（默认）
SESSION_EXPIRE_AT_BROWSER_CLOSE = False	是否关闭浏览器使得Session过期（默认）
SESSION_SAVE_EVERY_REQUEST = False	是否每次请求都保存Session，默认修改之后才保存（默认）

在视图中操作Session：

参数	描述
request.session['key'] = value	向Session写入键值
request.session.get('key',None)	获取Session中键的值
request.session.flush()	清除Session数据
request.session.set_expiry(value)	Session过期时间

自己实现用户登录认证

案例：自己实现登录认证机制

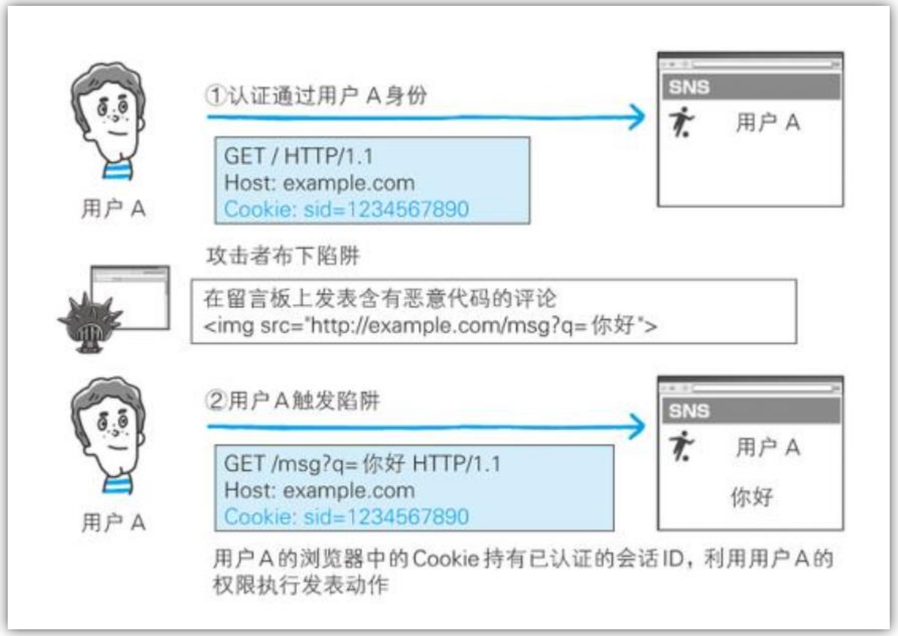
- 登录
- 视图验证登录
- 退出登录
- 装饰器

Django CSRF防护

- CSRF是什么
- Django CSRF工作原理
- 使用CSRF防护机制

CSRF是什么

CSRF（Cross Site Request Forgery）：跨站请求伪造，实现的原理是CSRF攻击者在用户已经登录目标网站之后，诱使用户访问一个攻击页面，利用目标网站对用户的信任，以用户身份在攻击页面对目标网站发起伪造用户操作的请求，达到攻击目的。



Django CSRF工作原理

Django怎么验证一个请求是不是CSRF?

Django处理客户端请求时，会生成一个随机Token，放到Cookie里一起返回，然后需要前端每次POST请求时带上这个Token，可以放到POST数据里键为csrfmiddlewaretoken，或者放到请求头键为X-CSRFToken，Django从这两个位置取，每次处理都会拦截验证，通过比对两者是否一致来判断这个请求是不是非法，非法就返回403状态码。

使用CSRF防护机制

常见有三种方法可以携带CSRF Token发送给服务端：

- from表单添加{% csrf_token %}标签，表单会携带一同提交
- 如果你是Ajax请求，需要把csrf token字符串（也是通过拿{% csrf_token %}标签产生的值）放到data里一起提交，并且键名为csrfmiddlewaretoken或者放到请求头传递服务端
- 指定取消某函数视图CSRF防护

使用CSRF防护机制

方法1：

```
<form method="post">
    {% csrf_token %}
    用户名: <input type="text" name="username"><br>
    密码: <input type="text" name="password"><br>
    <button type="submit">登录</button>
</form>
```

方法2：

```
var csrf_token = $("[name='csrfmiddlewaretoken']").val();
var data = {'id': '123', 'csrfmiddlewaretoken': csrf_token};
$.ajax({
    type: "POST",
    url: "/api",
    data: data,
    dataType: 'json'
})
```

方法3：

```
from django.views.decorators.csrf import csrf_exempt
@csrf_exempt
def index(request):
      return render(request, 'index.html')
```

谢谢

阿良微信

添加微信好友

DevOps技术栈

关注微信公众号

DevOps实战学院：www.ctnrs.com