电影推荐系统

https://github.com/zhurboo/moviesite

2019 年 11 月 22 日

目录

1 项目介绍

1.1 整体介绍

本项目一个结合了 Hadoop[1]、Hbase[2]、Spark[3]、MongoDB[4]、Django[5] 等开源框架的高可用分布式电影推荐系统，并以 Web 页面为用户提供了友好的访问方式。

1.2 功能介绍

本系统提供了以下功能：

- 基本功能
  - 登录注册
  - 个人信息修改
  - 个人喜好修改
  - 按类别电影展示
  - 电影搜索
  - 电影详情查看
  - 电影评分
  - 电影评论
- 后台管理
  - 用户管理
  - 电影管理
- 电影推荐
  - 离线推荐
    - 协同过滤推荐
    - 兴趣推荐
  - 在线推荐
    - 正推荐
    - 负推荐

[1] http://hadoop.apache.org/
[2] https://hbase.apache.org/
[3] https://spark.apache.org/
[4] https://www.mongodb.com/
[5] https://www.djangoproject.com/

1.3 成员分工

本项目的成员分工如表 1 所示。

表 1: 成员分工

姓名	分工
	架构设计、推荐算法、Web 服务器
	数据爬取、数据清洗
	前端页面、前后端数据交互
	数据预处理、数据导入、MongoDB 和 Kafka 的配置
	Hadoop、Hbase 和 Spark 的配置

2 系统架构设计

2.1 体系框架

为保证系统的高可用分布式，本系统基于分布式文件系统、分布式数据库、分布式资源管理系统、分布式计算、分布式存储、分布式文件系统以及分布式 Web 服务器，体系框架如图 1 所示。

2.2 系统架构

本系统的系统架构如图 2 所示，我们的数据基于 MovieLens 20M[6] 数据集，并将该数据集的 rating.csv 存入 HDFS 集群，并从 IMDB[7] 爬取数据集中的电影基本信息与电影图片，电影基本信息经过处理后存入 MongoDB 集群，电影图片存入 Hbaes[8] 集群。计算模块运行于 YARN 集群上，其中 Spark 负责离线推荐计算，Spark Streaming 负责在线推荐计算，离线推荐计算的任务每天执行一次，在线推荐计算的任务来源于 Kafka[9] 消息队列，时间窗口和间隔均为一分钟，计算所得的推荐结果存入 MongoDB 集群。使用 Nginx[10] 处理负载均衡和静态文件 (电影图片、css、js)，使用 uWSGI[11] 启动 Django 服务，并将动静请求分离，Redis[12] 作为 Django 的缓存。Zookeeper[13] 用来保证 Kafka、YARN、HDFS 和 Hbase 的高可用性。接下来，我们将分别介绍本系统中的各个组件。

[6]https://grouplens.org/datasets/movielens/
[7]https://www.imdb.com/
[8]https://hbase.apache.org/
[9]http://kafka.apache.org/
[10]http://nginx.org/en/
[11]https://pypi.org/project/uWSGI/
[12]https://redis.io/
[13]https://zookeeper.apache.org/

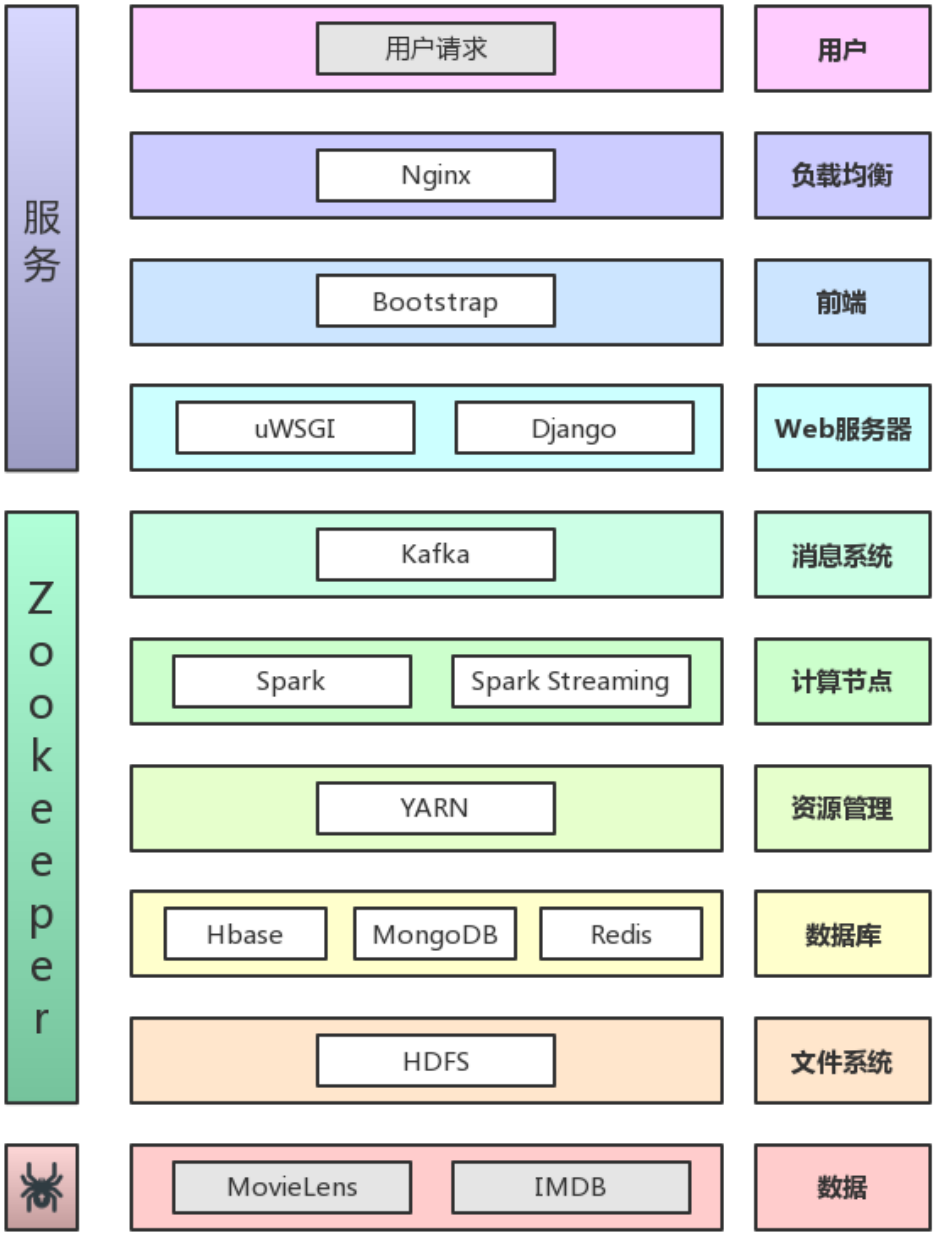
服务
Zookeeper
用户请求
用户
Nginx
负载均衡
Bootstrap
前端
uWSGI
Django
Web服务器
Kafka
消息系统
Spark
Spark Streaming
计算节点
YARN
资源管理
Hbase
MongoDB
Redis
数据库
HDFS
文件系统
MovieLens
IMDB
数据

图 1: 体系框架

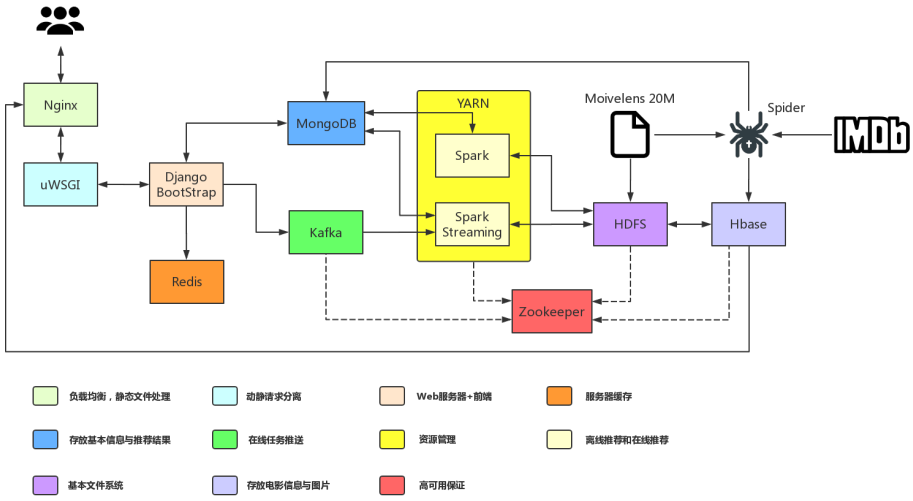

图 2: 系统架构

2.2.1 Zookeeper

ZooKeeper 是一个开源的分布式应用程序协调服务，是一个为分布式提供一致性保证其高可用的分布式系统。Zookeeper 的每个 Server 在内存中存储了一份数据，在 Zookeeper 集群启动时，将通过 Paxos 协议从实例中选举一个 leader，Leader 负责处理数据更新等操作，一个更新操作成功，意味着当且仅当大多数 Server 在内存中成功修改数据。本系统使用 Zookeeper 以保证 Kafka、YARN、HDFS 和 Hbase 的高可用。

2.2.2 HDFS

文件系统 HDFS 是 Hadoop 的重要组件。在 HDFS 集群中，NameNode 节点维护文件系统的元数据，接收用户请求，多个 Namenode 保证了 HDFS 的高可用性，任何一台 NameNode 宕机，也不会使得文件系统无法访问。DataNode 节点作为数据节点，保存所有的处理过的和未处理的所有数据。本系使用 HDFS 作为文件系统，用来存储 Spark 和 Spark Streaming 计算过程中所需要的的数据和对 Hbase 的底层支持。

2.2.3 YARN

资源管理系统 YARN 也是 Hadoop 的中药组件。在 YARN 集群中，ResourceManager 节点负责资源分配，在高可用模式下，多个 ResourceManager 保证了 YARN 的高可用性。NodeManage 节点负责监控节点内部的资源使用情况，并向 ResourceManager 节点汇报。本系统使用 YARN 作为资源管理系统，作为 Spark 和 Spark Streaming 的支持平台。

2.2.4 Hbase

Hbase 是一个分布式、面向列的开源数据库。在 Hbase 及群众，HMaster 节点负责为 HRegioServer 分配 region，负责 HRegionServer 的负载均衡，发现失效的 HRegionServer 并重新分配其上的 region，同时也管理用户对 table 的增删改操作。HRegionServer 节点负责维护 region，处理对 region 的 IO 请求，并且负责切分在运行过程中变得过大的 region。本系统使用 Hbase 作为存储电影图片的数据库，以解决 Nginx 服务器单机故障所导致的数据丢失问题。当有新的电影存入时会放在 Hbase 集群中时，会同时与 Nginx 同步，同时，当需要部署新的 Nginx 服务器时，也可从 Hbase 集群中下载电影图片。

2.2.5 Kafka

Kafka 是一个分布式发布 -订阅消息系统。在 Kafka 集群中没有 Master-Slave 的结构，多个节点是平等关系，由 Zookeeper 进行调度控制，在出现宕机的情况下能迅速切换主机对外提供正常服务。根据设置可以在每一个节点上都存储 topic，以保证集群的高可用。本系统使用 Kafka 作为在线推荐任务的消息系统，通过 Django 发布在线推荐任务，并将任务推送至 Spark Streaming 计算模块中。

2.2.6 Spark

Spark 使用 Scala 语言进行实现，它是一种面向对象、函数式编程语言，能够像操作本地集合对象一样轻松地操作分布式数据集，它是一个基于内存的计算框架，有着更快的计算速度，而且有更丰富的机器学习的库，是一个非常适合推荐系统的计算框架。本系统使用 Spark 作为离线推荐的计算框架，使用 Spark Streaming 作为在线推荐的计算框架，使用了 Spark on YARN 模式，以通过 YARN 高效地管理计算资源。

2.2.7 MongoDB

MongoDB 是用 C++ 语言编写的非关系型数据库。具有高性能、易部署、易使用、存储数据十分方便的特点。本系统采用 MongoDB 作为与 Django 的底层数据库，分布式架构采用 Replica Set 模式，其中 Primary 节点负责数据库的维护工作，能进行增删改查等操作，Secondary 节点则仅作为备份与分流节点，能在保存副本集数据的同时进行读操作其采用，在 Primary 节点宕机时，会有一个 Secondary 节点取代其位置，以保证了其高可用性。本系统中，MongoDB 用来存储用户信息、电影信息以及用户的推荐结果。

2.2.8 Redis

Redis 是一个高性能的 key-value 开源数据库，其具有支持数据持久化、支持多种数据结构、支持数据备份的特点。本系统使用 Redis 作为 Django 的缓存数据库，以降低 MongoDB 的负载及提高 Web 节点的响应速度。

2.2.9 Django

Django 是一个基于 Python 的开源 Web 应用框架，其设计模式借鉴了 MVC 框架的思想，将整个 Web 服务器分为三部分 (Model、View、Controller)，具有低耦合性、高重用性、低生命周期成本等特点。本系统使用 Django 作为 Web 服务器，作为与用户请求、数据库与计算模块的桥梁。

2.2.10 uWSGI

uWSGI 是一个 Web 服务器，其实现了 WSGI、uwsgi、http 等协议，把 http 协议转化为语言支持的网络协议，比如把 http 协议转化成 WSGI 规范，让 Python 可以直接使用。本系统使用 uWSGI 启动 Django 服务器，将动静请求分布，静态资源的请求交给 Nginx 服务器，动态请求转发给 Django 服务器。

2.2.11 Nginx

Nginx 是一款高性能的 http 和反向代理服务器，其可提供反向代理、负载均衡、http 等服务，具有高并发、低内存、简单稳定等特点。本系统使用 Nginx 作为负载均衡服务器，以使得 Web 服务器变的高可用，此外，Nginx 还用来处理电影图片等静态资源。

3 环境搭建

3.1 服务器信息

本项目基于 5 台 2 核 4G 内存的服务器，其服务器信息如表 2 所示。

表 2: 服务器信息

名称	内网 ip	外网 ip
cloud1	172.26.154.146	39.99.134.147
cloud2	172.26.240.217	39.98.134.47
cloud3	172.26.154.144	39.99.134.117
cloud4	172.26.154.145	39.99.140.220
cloud5	172.26.240.216	39.100.90.186

3.2 软件信息

本项目使用的软件信息如表 3 所示。

表 3: 软件信息

软件	版本	说明
Python	3.6.8	开发语言
JDK	1.8.0	支持 Apache 软件
Zookeeper	3.5.6	实现 HA
Hadoop	3.2.1	HDFS 和 YARN
Spark	2.4.4	计算离线和在线推荐任务
Hbase	2.2.2	储存图片
Kafka	2.3.1	推送在线推荐任务
MongoDB	4.2.1	基本信息数据库
Redis	5.0.5	Web 服务器缓存
Nginx	1.16.1	负载均衡，静态资源

3.3 节点分布

为了让 5 台服务器的资源得到充分且合理的使用，我们将上述服务分散在各个服务器上，具体节点分布如表 4 所示。我们的所有环境均配置在 \root\cloud302\ 目录下，接下来，我们详细描述将在纯净的 Centos7 服务器上配置整个系统的过程。

表 4: 节点分布

		cloud1	cloud2	cloud3	cloud4	cloud5
Zookeeper	QuorumPeerMain	√	√	√		
HDFS	NameNode	√	√			
	JournalNode	√	√	√		
	DataNode	√	√	√		
	ZFKC	√	√			
YARN	ResourceManager	√	√			
	NodeManager	√	√	√	√	√
Hbase	HMaster	√	√			
	HRegionServer	√	√	√		
	ThriftServer	√	√	√		
Kafka	Worker			√	√	√
Spark	Broker	√	√	√	√	√
MongoDB	-			√	√	√
Django	-				√	√
Redis	-				√	√
Nginx	-				√	√

3.4 环境配置

3.4.1 修改 HOSTNAME

1. 在 cloud1、cloud2、cloud3、cloud4、cloud5 安装基本依赖

```
yum -y install zlib zlib-devel bzip2-devel pcre-devel openssl-devel
    ncurses-devel sqlite-devel readline-devel tk-devel python3-devel
     gcc pcre-devel openssl openssl-deve
```

2. 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，修改 HOSTNAME，如 cloud1

   (a) 设置 HOSTNAME

```
hostnamectl set-hostname cloud1
```

   (b) 修改 hosts 文件

```
vim /etc/hosts
```

```
0 cloud1
172.26.240.217 cloud2
```

```
172.26.154.144 cloud3
172.26.154.145 cloud4
172.26.240.216 cloud5
```

3.4.2 配置免密钥登录

1. 配置 cloud1 免密登录 5 台服务器

   (a) 在 cloud1 上，生成密钥

   ```
   ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
   ```

   (b) 拷贝 cloud1 公钥至 cloud2、cloud3、cloud4、cloud5

   ```
   scp ~/.ssh/id_rsa.pub cloud2:/root/.ssh/id_rsa.pub.cloud1
   scp ~/.ssh/id_rsa.pub cloud3:/root/.ssh/id_rsa.pub.cloud1
   scp ~/.ssh/id_rsa.pub cloud4:/root/.ssh/id_rsa.pub.cloud1
   scp ~/.ssh/id_rsa.pub cloud5:/root/.ssh/id_rsa.pub.cloud1
   ```

   (c) 在 cloud1 上，将公钥写入 authorized_keys 文件

   ```
   cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
   ```

   (d) 在 cloud2、cloud3、cloud4、cloud5 上，将公钥写入 authorized_keys 文件

   ```
   cat ~/.ssh/id_rsa.pub.cloud1 >> ~/.ssh/authorized_keys
   ```

2. 同理，配置 cloud2、cloud3、cloud4、cloud5 免密登录 5 台服务器

3.4.3 安装 Python3

1. 安装 Python3

   ```
   yum install python3
   ```

2. 删除原 Python 和 pip 的软连接

   ```
   cd /usr/bin
   rm python pip
   ```

3. 创建新的软连接

   ```
   ln -s python3 python
   ln -s pip3 pip
   ```

3.4.4 配置 JDK

1. 解压安装包

```
tar -zxvf jdk-8u231-linux-x64.tar.gz -C /usr/cloud302
mv /usr/cloud302/jdk1.8.0_231 /usr/cloud302/java
```

2. 添加环境变量

```
vim /etc/profile
```

```
export JAVA_HOME=/usr/cloud/java
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=${JAVA_HOME}/bin:$PATH
```

```
source /etc/profile
```

3. 检查 JDK

```
java -version
```

3.4.5 配置 Zookeeper

1. 解压安装包

```
tar -zxvf apache-zookeeper-3.5.6-bin.tar.gz -C /usr/cloud302
mv /usr/cloud302/apache-zookeeper-3.5.6-bin /usr/cloud302/zookeeper
```

2. 添加环境变量

```
vim /etc/profile
```

```
export ZOOKEEPER_HOME=/usr/cloud302/zookeeper
export PATH=${ZOOKEEPER_HOME}/bin:$PATH
```

```
source /etc/profile
```

3. 在 cloud1、cloud2、cloud3 上，配置 Zookeeper

   (a) 在 cloud1、cloud2、cloud3 上，创建 data 和 logs 目录

```
mkdir ${ZOOKEEPER_HOME}/{data,logs}
```

(b) 在 cloud1、cloud2、cloud3 上，配置 zoo.cfg 文件

```
mv $ZOOKEEPER_HOME/conf/zoo_sample.cfg $ZOOKEEPER_HOME/conf/zoo.cfg
vim $ZOOKEEPER_HOME/conf/zoo.cfg
```

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/usr/cloud302/zookeeper/data
dataLogDir=/usr/cloud302/zookeeper/logs
clientPort=6181
server.1=cloud1:6888:6889
server.2=cloud2:6888:6889
server.3=cloud3:6888:6889
```

(c) 在 cloud1 上，配置 myid 文件

```
echo '1' > $ZOOKEEPER_HOME/data/myid
```

(d) 在 cloud2 上，配置 myid 文件

```
echo '2' > $ZOOKEEPER_HOME/data/myid
```

(e) 在 cloud3 上，配置 myid 文件

```
echo '3' > $ZOOKEEPER_HOME/data/myid
```

3.4.6 配置 HDFS(HA)

1. 解压安装包

```
tar -zxvf hadoop-3.2.1.tar.gz -C /usr/cloud302
mv /usr/cloud302/hadoop-3.2.1 /usr/cloud302/hadoop
```

2. 添加环境变量

```
vim /etc/profile
```

```
export HADOOP_HOME=/usr/cloud302/hadoop
export PATH=${HADOOP_HOME}/bin:${HADOOP_HOME}/sbin:$PATH
```

```
source /etc/profile
```

3. 在 cloud1、cloud2、cloud3 上，配置 HDFS(HA)

(a) 配置 Zookeeper

(b) 在 cloud1、cloud2、cloud3 上，配置 hadoop-env.sh 文件

```
vim $HADOOP_HOME/etc/hadoop/hadoop-env.sh
```

```
export JAVA_HOME=/usr/cloud302/java
```

(c) 在 cloud1、cloud2、cloud3 上，配置 start-dfs.sh 文件

```
vim $HADOOP_HOME/sbin/start-dfs.sh
```

```
HDFS_DATANODE_USER=root
HDFS_DATANODE_SECURE_USER=hdfs
HDFS_NAMENODE_USER=root
HDFS_ZKFC_USER=root
```

(d) 在 cloud1、cloud2、cloud3 上，配置 stop-dfs.sh 文件

```
vim $HADOOP_HOME/sbin/stop-dfs.sh
```

```
HDFS_DATANODE_USER=root
HDFS_DATANODE_SECURE_USER=hdfs
HDFS_NAMENODE_USER=root
HDFS_ZKFC_USER=root
```

(e) 在 cloud1、cloud2、cloud3 上，配置 core-site.xml 文件

```
vim $HADOOP_HOME/etc/hadoop/core-site.xml
```

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://cluster</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/usr/cloud302/hadoop/tmp</value>
    </property>
    <property>
        <name>hadoop.proxyuser.hduser.hosts</name>
        <value>*</value>
    </property>
    <property>
        <name>hadoop.proxyuser.hduser.groups</name>
```

```
        <value>*</value>
    </property>
    <property>
        <name>ha.zookeeper.quorum</name>
        <value>cloud1:6181,cloud2:6181,cloud3:6181</value>
    </property>
</configuration>
```

(f) 在 cloud1、cloud2、cloud3 上，配置 hdfs-site.xml 文件

```
vim $HADOOP_HOME/etc/hadoop/hdfs-site.xml
```

```
<configuration>
    <property>
        <name>dfs.permissions.enabled</name>
        <value>false</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>usr/cloud302/hadoop/hdfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/usr/cloud302/hadoop/hdfs/data</value>
    </property>
    <property>
        <name>dfs.journalnode.edits.dir</name>
        <value>/usr/cloud302/hadoop/hdfs/journal</value>
    </property>
    <property>
        <name>dfs.nameservices</name>
        <value>cluster</value>
    </property>
    <property>
        <name>dfs.ha.namenodes.cluster</name>
        <value>nn1,nn2</value>
    </property>
    <property>
        <name>dfs.namenode.rpc-address.cluster.nn1</name>
        <value>cloud1:8020</value>
    </property>
    <property>
        <name>dfs.namenode.rpc-address.cluster.nn2</name>
        <value>cloud2:8020</value>
    </property>
```

```
    <property>
        <name>dfs.namenode.http-address.cluster.nn1</name>
        <value>cloud1:9870</value>
    </property>
    <property>
        <name>dfs.namenode.http-address.cluster.nn2</name>
        <value>cloud2:9870</value>
    </property>
    <property>
        <name>dfs.datanode.shared.edits.dir</name>
        <value>qjournal://cloud1:8485;cloud2:8485/cluster</value>
    </property>
    <property>
        <name>dfs.client.failover.proxy.provider.cluster</name>
        <value>org.apache.hadoop.hdfs.server.namenode.ha.
            ConfiguredFailoverProxyProvider</value>
    </property>
    <property>
        <name>dfs.ha.fencing.methods</name>
        <value>sshfence</value>
    </property>
    <property>
        <name>dfs.ha.fencing.ssh.private-key-files</name>
        <value>/root/.ssh/id_rsa</value>
    </property>
    <property>
        <name>dfs.ha.automatic-failover.enabled</name>
        <value>true</value>
    </property>
</configuration>
```

(g) 在 cloud1 上，配置 workers 文件

```
vim /usr/hadoop/hadoop-3.2.1/etc/hadoop/workers
```

```
cloud1
cloud2
cloud3
```

3.4.7 配置 YARN(HA)

1. 完成 Hadoop 环境配置

2. 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，配置 YARN(HA)

   (a) 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，配置 start-yarn.sh 文件

```
vim $HADOOP_HOME/sbin/start-yarn.sh
```

```
YARN_RESOURCEMANAGER_USER=root
HADOOP_SECURE_DN_USRR=yarn
YARN_NODEMANAGER_USER=root
```

   (b) 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，配置 stop-yarn.sh 文件

```
vim $HADOOP_HOME/sbin/stop-yarn.sh
```

```
YARN_RESOURCEMANAGER_USER=root
HADOOP_SECURE_DN_USRR=yarn
YARN_NODEMANAGER_USER=root
```

   (c) 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，配置 mapred-site.xml 文件

```
vim $HADOOP_HOME/etc/hadoop/mapred-site.xml
```

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

   (d) 在 cloud1、cloud2、cloud3、cloud4、cloud5 上，配置 yarn-site.xml 文件

```
vim $HADOOP_HOME/etc/hadoop/yarn-site.xml
```

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.resourcemanager.ha.enabled</name>
        <value>true</value>
    </property>
```

```
    <property>
        <name>yarn.resourcemanager.cluster-id</name>
        <value>cluster-yarn</value>
    </property>
    <property>
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>cloud1</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>cloud2</value>
    </property>
    <property>
        <name>yarn.resourcemanager.recovery.enabled</name>
        <value>true</value>
    </property>
    <property>
        <name>yarn.resourcemanager.store.class</name>
        <value>org.apache.hadoop.yarn.server.resourcemanager.
            recovery.ZKRMStateStore</value>
    </property>
    <property>
        <name>hadoop.zk.address</name>
        <value>cloud1:6181,cloud2:6181,cloud3:6181</value>
    </property>
</configuration>
```

(e) 在 cloud2 上，配置 workers‘ 文件

```
vim /usr/hadoop/hadoop-3.2.1/etc/hadoop/workers
```

```
cloud1
cloud2
cloud3
cloud4
cloud5
```

3.4.8 配置 Hbase(HA)

1. 解压安装包

```
tar -zxvf hbase-2.2.2-bin.tar.gz -C /usr/cloud302
mv /usr/cloud302/hbase-2.2.2 /usr/cloud302/hbase
```

2. 添加环境变量

```
vim /etc/profile
```

```
export HBASE_HOME=/usr/cloud302/hbase
export PATH=${HBASE_HOME}/bin:$PATH
```

```
source /etc/profile
```

3. 在 cloud1、cloud2、cloud3 上，配置 Hbase(HA)

   (a) 配置 HDFS(HA)

   (b) 在 cloud1、cloud2、cloud3 上，配置 hbase-env.sh 文件

```
vim ${HBASE_HOME}/conf/hbase-env.sh
```

```
export JAVA_HOME=/usr/cloud302/java
export HBASE_MANAGES_ZK=false
```

   (c) 在 cloud1、cloud2、cloud3 上，拷贝 HDFS 的配置

```
cp $HADOOP_HOME/etc/hadoop/core-site.xml $HBASE_HOME/conf
cp $HADOOP_HOME/etc/hadoop/hdfs-site.xml $HBASE_HOME/conf
```

   (d) 在 cloud1、cloud2、cloud3 上，配置 hbase-site.xml 文件

```
vim ${HBASE_HOME}/conf/hbase-site.xml
```

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://cluster/hbase</value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
        <property>
```

```
        <name>hbase.zookeeper.quorum</name>
        <value>cloud1:6181,cloud2:6181,cloud3:6181</value>
    </property>
    <property>
        <name>hbase.unsafe.stream.capability.enforce</name>
        <value>false</value>
    </property>
    <property>
        <name>hbase.master.port</name>
        <value>9000</value>
    </property>
    <property>
        <name>hbase.master.info.port</name>
        <value>9001</value>
    </property>
    <property>
        <name>hbase.regionserver.port</name>
        <value>9002</value>
    </property>
    <property>
        <name>hbase.regionserver.info.port</name>
        <value>9003</value>
    </property>
    <property>
        <name>hbase.thrift.server.socket.read.timeout</name>
        <value>86400000</value>
    </property>
    <property>
        <name>hbase.thrift.connection.max-idletime</name>
        <value>86400000</value>
    </property>
</configuration>
```

(e) 在 cloud1 上，配置 regionservers 文件

```
vim ${HBASE_HOME}/conf/regionservers
```

```
cloud1
cloud2
cloud3
```

3.4.9 配置 Kafka(HA)

1. 解压安装包

```
tar -zxvf kafka_2.12-2.3.1.tgz -C /usr/cloud302
mv /usr/cloud302/kafka_2.12-2.3.1 /usr/cloud302/kafka
```

2. 添加环境变量

```
vim /etc/profile
```

```
export KAFKA_HOME=/usr/cloud302/kafka
export PATH=${KAFKA_HOME}/bin:$PATH
```

```
source /etc/profile
```

3. 在 cloud3、cloud4、cloud5 上，配置 Kafka(HA)

   (a) 配置 Zookeeper

   (b) 在 cloud3 上，配置 server.properties 文件

```
vim $KAFKA_HOME/config/server.properties
```

```
bloker.id=3
listeners=PLAINTEXT://cloud3:9092
advertised.listeners=PLAINTEXT://cloud3:9092
num.partitions=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=3
zookeeper.connect=cloud1:6181,cloud2:6181,cloud3:6181
```

   (c) 在 cloud4 上，配置 server.properties 文件

```
vim $KAFKA_HOME/config/server.properties
```

```
bloker.id=4
listeners=PLAINTEXT://cloud4:9092
advertised.listeners=PLAINTEXT://cloud4:9092
num.partitions=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=3
zookeeper.connect=cloud1:6181,cloud2:6181,cloud3:6181
```

(d) 在 cloud5 上，配置 server.properties 文件

```
vim $KAFKA_HOME/config/server.properties
```

```
bloker.id=5
listeners=PLAINTEXT://cloud5:9092
advertised.listeners=PLAINTEXT://cloud5:9092
num.partitions=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=3
zookeeper.connect=cloud1:6181,cloud2:6181,cloud3:6181
```

3.4.10 配置 Spark(On YARN)

1. 解压安装包

```
tar -zxvf spark-2.4.4-bin-hadoop2.7.tgz -C /usr/cloud302
mv /usr/cloud302/spark-2.4.4-bin-hadoop2.7 /usr/cloud302/spark
```

2. 添加环境变量

```
vim /etc/profile
```

```
export SPARK_HOME=/usr/cloud302/spark
export PATH=${SPARK_HOME}/bin:${SPARK_HOME}/sbin:$PATH
```

```
source /etc/profile
```

3.4.11 配置 MongoDB(Replica Set)

1. 解压安装包

```
tar -zxvf mongodb-linux-x86_64-rhel70-4.2.1.tgz -C /usr/cloud302
mv /usr/cloud302/mongodb-linux-x86_64-rhel70-4.2.1 /usr/cloud302/
    mongodb
```

2. 添加环境变量

```
vim /etc/profile
```

```
export MONGODB_HOME=/usr/cloud302/mongodb
export PATH=${MONGODB_HOME}/bin:$PATH
```

```
source /etc/profile
```

3. 在 cloud3、cloud4、cloud5 上，配置 MongoDB(Replica Set)

   (a) 在 cloud3、cloud4、cloud5 上，创建数据文件夹

```
mkdir -p $MONGODB_HOME/data/db
```

   (b) 在 cloud3、cloud4、cloud5 上，配置 mongo.conf 文件

```
vim $MONGODB_HOME/mongo.conf
```

```
systemLog:
    destination: file
    path: "/usr/cloud302/mongodb/mongo.log"
    logAppend: true
storage:
    dbPath: "/usr/cloud302/mongodb/data/db"
    journal:
    enabled: true
replication:
    replSetName: "cloud"
processManagement:
    fork: true
net:
    bindIp: 0.0.0.0
    port: 8018
setParameter:
    enableLocalhostAuthBypass: false
```

   (c) 在 cloud3 上，设置集群和创建帐号

```
mongo -host cloud3:8018
```

```
use admin
rs.initiate({_id: 'cloud', members:[{_id: 0, host: 'cloud3:8018'},
    {_id: 2, host: 'cloud4:8018'}, {_id: 2, host: 'cloud4:8018'}]})
```

   (d) 在 MongoDB 集群上，创建帐号

```
mongo -host cloud/cloud3:8018,cloud4:8018,cloud5:8018
```

```
db.createUser({user: "admin", pwd: "123456", roles: [{ role: "
    userAdminAnyDatabase", db: "admin"}]})
```

3.4.12 配置 Redis

1. 安装 Redis

```
yum -y install gcc zlib zlib-devel pcre-devel openssl openssl-devel
tar -zxvf redis-5.0.5.tar.gz -C /usr/cloud302/
mv /usr/cloud302/redis-5.0.5/ /usr/cloud302/redis
make
```

2. 添加环境变量

```
vim /etc/profile
```

```
export REDIS_HOME=/usr/cloud302/redis
export PATH=${REDIS_HOME}/src:$PATH
```

```
source /etc/profile
```

3. 在 cloud4、cloud5 上，配置 redis.conf 文件

```
vim $REDIS_HOME/redis.conf
```

```
daemonize yes
dir /usr/cloud302/redis/
```

3.4.13 配置 uWSGI

1. 在进入 Django 虚拟环境

```
cd /usr/cloud302/moviesite
virtualenv . -p python3
source bin/activate
```

2. 安装 uWSGI

```
pip install uwsgi
```

3. 配置 uwsgi.ini 文件

```
vim uwsgi.ini
```

```
[uwsgi]
http=0:8005
chdir=/usr/cloud302/moviesite
```

```
wsgi-file=moviesite/wsgi.py
daemonize=/usr/cloud302/moviesite/uwsgi.log
pidfile=/usr/cloud302/moviesite/uwsgi.pid
```

3.4.14 配置 Nginx

1. 安装 Nginx

```
yum -y install
tar -zxvf nginx-1.16.1.tar.gz
cd nginx-1.16.1
./configure --prefix=/usr/cloud302/nginx
make & make install
```

2. 添加环境变量

```
vim /etc/profile
```

```
export NGINX_HOME=/usr/cloud302/nginx
export PATH=${NGINX_HOME}/sbin:$PATH
```

```
source /etc/profile
```

3. 配置 ngnix.conf 文件

```
vim $NGINX_HOME/conf/nginx.conf
```

```
user root;
worker_processes  1;
events {worker_connections  1024;}
http {
    upstream cloud{
        server cloud4:8005 weight=3;
        server cloud5:8005 weight=3;}
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       8006;
        server_name  localhost;
        location / { proxy_pass http://cloud;}
        location /static{ alias /usr/cloud302/moviesite/static_new;}
    }
}
```

4 数据采集

4.1 数据介绍

在电影推荐系统中，为了更好地为用户提供高质量的电影推荐，我们需要收集大量的电影信息以及不同用户对这些电影的评分。为了更好地实现这点，我们选择了一个现有的数据集 MovieLens 20M 数据集，改数据集中包含了约 27,000 部电影，每部电影都有对应的标签和上映日期，同时数据集中还有约 138,000 名用户为每部电影的评分，总计约有 20,000,000 条记录。基于这个已有的数据集，我们对电影信息展开了数据采集工作。数据的采集主要可以分成两个部分，分别是基于 MovieLens 20M 数据集的电影信息采集以及最新电影信息采集，在接下来的部分中我们会分别介绍这两个部分。

4.2 IMDB 中 MovieLens 的电影信息采集

为了高效快捷地爬取 MovieLens 20M Dataset 数据集中对应的约 27,000 部电影的信息，我们基于 Python3 的 requests[14]、multiprocessing[15] 和 BeautifulSoup[16] 模块设计了一个多进程爬虫，该爬虫可以应对浏览器识别问题、IP 限制问题、单进程爬取速度慢等一系列问题。在接下来的部分中我们会详细介绍我们的设计思路和具体实现。

4.2.1 爬取的内容

为了更好地构建我们自己的电影数据库，我们需要从类似豆瓣、烂番茄、或者 IMDB 等电影网站中爬取电影的完整信息。由于在 MovieLens 20M 数据集中，有所有电影的 IMDB 编号，这样方便我们去爬取所有的电影，于是在本工作中，我们选择爬取 IMDB 中对应的电影的全部信息。我们主要爬取的内容包括电影名、电影年份、电影类别、电影时长、评分、主要导演、主要编剧、主要演员、语言、出品商、票房、电影简介、最有效的 20 条评论、以及电影对应的封面。为了爬取这些信息，我们针对每一部电影都爬取了 3 个网站，分别对应电影的基本信息、电影的封面、以及用户的评论信息。

4.2.2 代理 IP

在爬虫爬取网页的时候，会遇到这样的情况，同一个 IP 地址短时间内多次访问同一个网站，该网站会将该 IP 地址封禁，阻止其访问网站。我们的爬虫往往需要在短时间内多次访问相同的网站，并且从网站中读取大量的信息，如果我们使用本地的唯一 IP 地址，就会使我们的 IP 地址被网站封禁，于是，为了解决这样的问题，我们利用网上免费的代理 IP 来访问 IMDB，我们从快代理网站中爬取了大量代理 IP，同时尝试着

[14]https://requests.kennethreitz.org/en/master/
[15]https://docs.python.org/3.8/library/multiprocessing.html
[16]https://www.crummy.com/software/BeautifulSoup/

用这些 IP 访问 IMDB 网站，如果在超时前访问成功，我们就保留对应的 IP 地址，反之我们将该 IP 删除，基于剩余的 IP 地址，我们就构建了自己的 IP 池，每次爬取相应的内容时，我们会利用 random.choice 函数随机从 IP 池中抽取一个可用的 IP 地址，利用该地址来访问 IMDB 的网页，这样有效地帮助我们避免了由于重复 IP 地址访问同一网页带来的问题。

4.2.3 伪装浏览器

在爬虫爬取网页的时候，有时候网站会检查浏览器的 header，以便确认访问网站的不是一个爬虫，而是一个真实的浏览器，如果网站发现 header 中没有 User-Agent 的信息，网站就会将访问定义成爬虫爬取，从而将禁止访问行为。于是我们收集了许多浏览器的 User-Agent 信息，构建了 UA 数组，每次访问 IMDB 的时候，我们都利用 random.choice 随机从 UA 数组中随机抽出一个，写入请求语句的 header 中。

4.2.4 多进程爬虫以及延时爬取

在我们的工作中一共设计了两个版本的爬虫，一个版本是单进程的爬虫，另一个版本是多进程的爬虫，我们分别用两个爬虫去爬取 27,000 部电影的信息，单进程版本的爬虫爬取时间极长，所以我们使用了 10 个进程来完成整个爬取工作，需要的时间大大缩短，我们将每个需要爬取的电影分配个 10 个进程中的一个，进程间并行爬取。在每个进程爬取结束之后，我们利用 time.sleep 函数让该进程随机暂停一段时间，考虑到爬虫的速度，我们将时间设定为 0-4 秒钟的一个，之所以随机选择暂停时间，是因为如果将时间确定为一个固定值，会使得访问行为更像一个爬虫的行为，因而我们设定成一个随机值。

4.2.5 错误文件删除

在爬虫爬取的时候，因为各种原因我们会遇到爬虫爬取失败的情况，针对这种情况，我们手动查看了一下对应的网页，发现可能的原因主要有以下几种：

- 爬虫本身问题 (网络等)
- 网页信息缺失

针对爬虫本身问题，我们记录了失败的页面，并进行第二次爬取；针对网页信息缺失，我们将这部分电影丢弃。我们利用 os 模块，遍历我们爬取的所有文件夹，针对那些文件数少于预期值的我们直接删除。

4.3 IMDB 中最新电影信息采集

为了实时的展示最新的电影，我们还对最新电影进行了实时爬取，我们爬取的页面是 https://www.imdb.com/showtimes/location/US/90001/2019-11-15，最后的日期是我们想要查询的日期。我们从这个页面中提取了所有的所有新电影的 id，并且利用已有的爬虫，对新电影展开多进程爬取，我们在爬取之后还对错误的文件进行了删除工作。

5 数据处理

5.1 数据处理简介

Django 直接从 MongoDB 中获取数据展示至网页，这里采取两层结构存储电影信息。第一层结构是所有电影的 genres、writers、directors、actors、languages 数据与电影条目数据 (包括 IMDB、名称、年份、评分等，还有一个唯一的 movieId)；第二层结构是每一个电影 (记为 x) 与包含的 genres、writers、directors、actors、languages 数据 (记为 y) 的关联表，每一个 x-y 记录为一行，便于后续的推荐算法使用。

5.2 数据处理过程

这一部分的主要内容是按照数据处理的要求对爬虫获取的数据进行格式化、切分并写入数据库，为系统提供源数据；原始数据是爬虫爬取的电影数据，每个电影数据文件以 IMDB 号为文件名，包含电影海报、txt 类型的接送格式数据文本。

处理时，首先读取 links.csv 文件 (指明 movieId 与 IMDB 的映射)，利用 ditc 结构进行存储；然后连接数据库并测试连接情况，做好准备工作。接下来依次读取路径下所有的电影数据文件，对源数据每一个电影文件进行处理：将 txt 文件内容转 json 化，按照特定的规则构造每一个电影数据的 list 元素，最后一并写入数据库的 moviemovie 表中。在此过程中，读取电影的 genres、writers、directors、actors、languages 详细数据 (这一部分数据每一个类别同一个电影可能包含多个，且不同电影存在交集)，分别放入 set 中，然后指定 id、构造元素并写入 movie 相应的表中。

最后然后处理关联表：对每一个电影文件，genres、writers、directors、actors、languages 等类型数据可能有不止一个条目，因此对于一条电影数据，从源文件中找出其相应类型的具体信息，将每一个条目都在其数据表中查找对应的 id，然后与电影的 id 建立一个唯一的 movieId-itemId 对应表，写入数据库。在此过程中，需要注意对异常数据的处理。例如，有的电影 writers 可能以 writers 或 writer 体现，同时可能为空，因此需要判断并对应处理。

6 算法开发

6.1 算法介绍

本系统提供了两种类型的推荐算法，分别为离线推荐算法和在线推荐算法。其中，离线推荐算法包括协同过滤、兴趣推荐和无评价推荐。在线推荐算法包括正推荐和负推荐。如图 3 所示，离线推荐基于 Spark，在线推荐基于 Spark Streaming，它们的最终目的均为改变 MongoDB 中的用户推荐电影。在 MongoDB 数据库中，每位用户的推荐电影数目为 30 个，其包括电影 ID 和推荐权重，在 Web 端，系统仅展示权重最大的 12 部电影，剩余的未使用电影则作为推荐算法的缓冲。接下来，我们将分别来介绍离线推介算法和在线推荐算法。

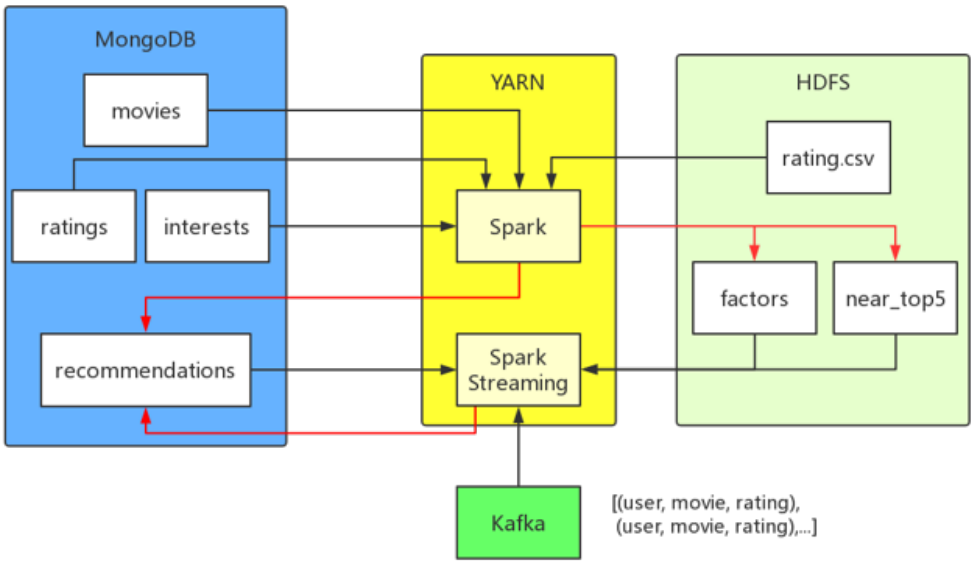

图 3: 推荐算法

6.2 离线推荐算法

6.2.1 协同过滤

推荐系统的目的是联系用户和信息，一方面帮助用户发现对自己有价值的信息, 而另一方面让信息能够展现在对它感兴趣的用户面前从而实现信息消费者和信息生产者的

双赢。协同过滤是推荐系统中的一种重要算法，其分为基于用户的协同过滤和基于项目的协同过滤。本系统采用了基于用户的协同过滤，其假设喜欢类似物品的用户可能有相同或相似的偏好，基于用户对物品的偏好找到相邻令居用户，然后将邻居用户的喜欢推荐给当前用。我们使用了 Spark 中的 ml[17] 库，使用其提供的 ALS 模型，我们只需要输入用户对电影评分的稀疏矩阵，即可得到每位评分用户的推荐电影。需要注意的是，这里我们的评分矩阵是基于 MovieLens 20M 数据集和本系统的用户评分，但我们提供推荐服务的用户仅仅是本系统的真实用户。我们取该算法推荐的前 30 部电影，此外，该模型还提供了用户和电影的特征向量，其中电影特征向量正是我们后面需要用到的。在在线推荐算法中，我们需要快速地计算任意一部电影与的最相邻的几部电影，我们的系统需要同时考虑空间与时间的开销，若我们将所有的相似度计算留到在线推荐算法中，那么在线推荐算法便会奇慢无比，若我们计算且储存所有电影两两间的相似度矩阵，那么内存便会不足。经过衡量，我们决定储存每部电影的相似 Top-5 和所有电影的特征向量，以在时间和空间做一个均衡处理。下面我们将展示 ALS 模型中的核心代码。

```
als  = ALS(maxIter=5, regParam=0.01)
model = als.fit(all_ratings_df)
recommendations_top_rdd = model.recommendForUserSubset(true_ratings_df,30)\
                .rdd.flatMap(lambda x: [(x[0],v[0],v[1]) for v in x[1]])
factors_rdd = model.itemFactors.rdd.map(lambda x:[x[0]]+x[1])
```

6.2.2 兴趣推荐

我们的 Web 系统提供了用户感兴趣电影的选择，针对填写了感兴趣电影的用户，我们将提供兴趣推荐服务。同样，我们使用协同过滤模型中的 ALS 模型，获得其推荐的前 60 部电影，在这 60 电影中，我们再次过滤用户不感兴趣的题材，以保留用户感兴趣的题材，并将其权重乘以 1.1 。最后，将这部分电影与协同过滤推荐的 30 部电影做并操作，剔除重复电影。

6.2.3 无评价推荐

我们的系统每星期将获取一批最新电影，在系统中，这部分新电影的是无用户评价的，那么 ALS 算法将无法推荐此部分电影。无法被推荐到，就更难以被用户评价，这会陷入一个恶性循环。针对这种现象，我们不仅提供了最新电影的展示，还在离线推荐算法中考虑了这种无评价电影的推荐，系统从 MongoDB 数据库中获取所有电影，减去评分表中已出现的电影，即可得到未评价的电影，在这部分未评价的电影中，我们随机的选取 5 部，并赋予一个随机的权重。至此，我们将离线推荐算法中三部分的结果结合，消去重复元素，并最终排序取前 30 写入 MongoDB 的用户推荐电影列表。

[17]https://spark.apache.org/docs/latest/api/python/pyspark.ml.html

6.3 在线推荐算法

6.3.1 在线推荐介绍

本系统需要根据用户近期内的动作行为，推测用户的心理，以推荐用户所感兴趣的电影，在 Django 服务器端，我们收集了用户的电影点击行为和电影评分行为，并实时地通过 Kafka 将这种行为传给 Spark Streaming，其形式如 (用户 ID，电影 ID，分数)。需要注意的是，这是一种批处理的过程，即每一次在线推荐算法包含了多条记录，我们取一份作为滑动的窗口和采样的频率，即每一分钟通过所有用户操作更新一次用户推荐列表。其中，我们将在线推荐任务分为正推荐和负推荐。正推荐包括：评分大于 3 的用户评分行为和电影点击行为 (视为评分 3.5)，服推荐包括评分小于 3 的用户评分行为。在线推荐算法的输入为用户已有的推荐列表、电影特征矩阵、电影 Top-5 矩阵以及用户行为，输出为用户推荐列表的更新。

6.3.2 在线正推荐

对于在线正推荐任务，我们获取的信息是用户对某部电影比较感兴趣，此时，我们需要从所有电影库中，寻找与之相似的电影，电影 Top-5 矩阵便发挥了作用，我们从矩阵中直接获取与此电影最相似的 5 部电影，并根据评分高低赋予推荐的权重，最后将这些电影与用户已有的推荐列表结合。

6.3.3 在线负推荐

对于在线负推荐任务，我们获取的信息是用户对某部电影不敢兴趣，此时，我们需要从用户已有的推荐列表中下手，我们使用电影特征矩阵来计算该电影与用户推荐列表中所有电影的相似性。类似的，我们取其中的前 5，即用户在其推荐列表中最可能不感兴趣的前 5 部电影，我们降低其推荐权重，具体降低情况视评分而定。最终，在经过在线正推荐和在线负推荐的过程后，我们重新排列了每个用户的推荐列表，且最终的数量往往大于 30，此时，还需要截取前 30 部推荐电影，更新 MongoDB 数据库。

7 Web 服务端开发

7.1 Web 服务端介绍

Django 服务器采用 MVC 模式，其中包括模型、视图和控制器，我们的系统提供的主要功能有用户的基本模块和后台管理模块，包括登录注册、个人信息修改、个人喜好修改、按类别电影展示、推荐电影展示、电影搜索、电影详情查看、电影评分、电影评论、后台管理。接下来我们将分别介绍该系统的模型和视图。

7.2 模型

本系统的模型如图 4 所示，该系统主要分为 User 和 Movie 两大模块，其中 Movie 中的题材、编剧、导演、演员、语言均为所对多的关系，因此在 Djnago 中，重新为他们定义了新的模型。每个 User 实例对应多个 Recommendation 实例，即多部推荐电影。每个 User 可对每部 Movie 评论和评分，故 User 对 Comment、User 对 Rating、Movie 对 Comment、Movie 对 Rating 均是一对多的关系。

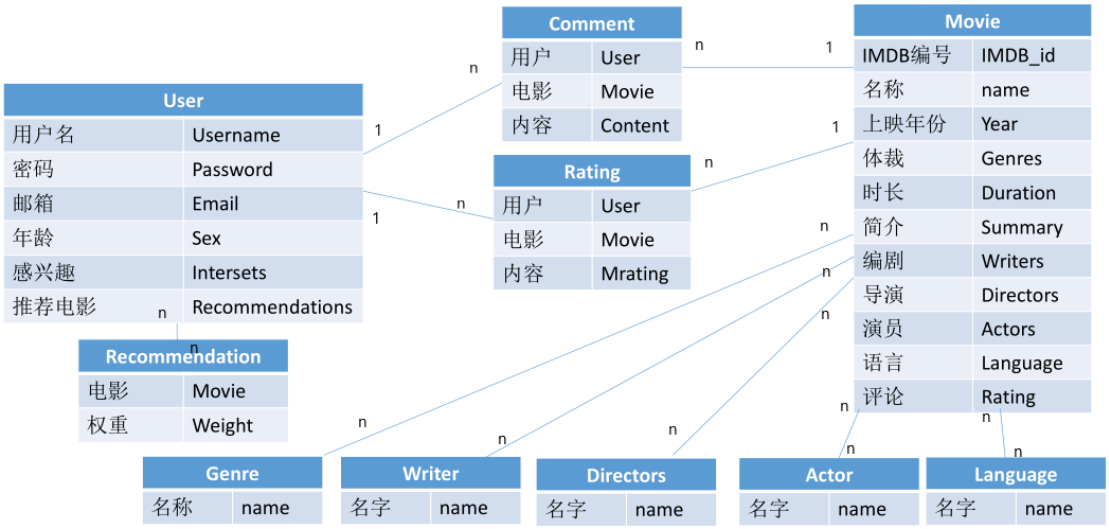

图 4: 模型

7.3 视图

本系统的视图也比较简单，在 User 部分，我们使用了 django-allauth[18] 模块，其提供了基本的用户登录、注册、修改密码等操作，另外还需要我们实现个人信息展示与个人信息修改。在修改信息页面，若为 GET 请求，则展示用户已有的个人信息，若为 POST 请求，则修改用户的个人信息。在个人信息中，我们加入了感兴趣的题材多选框，以提供离线推荐算法中的感兴趣推荐支持。

[18]https://pypi.org/project/django-allauth/

```
def profile_view(request)
def change_profile_view(request)
```

在 Movie 部分，需要实现以下视图，其中主要包括推荐列表、各种分类列表、电影搜索已经电影详细页面的展示。其中电影搜索通过 GET 方法传递需要搜索的电影名字，采用的是一种模糊搜索的方式。电影详细页面可添加评论与评分，添加评论与评分需要用户先登录，此数据是通过 POST 方法传递。

```
def movie_recommendation_list(request)
def movie_hot_list(request)
def movie_new_list(request)
def movie_action_list(request)
def movie_adventure_list(request)
def movie_science_list(request)
def movie_thriller_list(request)
def movie_romance_list(request)
def movie_amimation_list(request)
def movie_children_list(request)
def movie_comedy_list(request)
def movie_documentary_list(request)
def movie_search(request)
def movie_detail(request, movie_id)
```

7.4 分页

在各个列表页面，由于电影数量过多，我们需要对每个页面的电影做分页处理，这里用到了 Django 的分页器 Paginator[19]，将大数量电影按制定数量分成每页的电影。前端通过 GET 参数指定访问的页码，Web 服务端返回对应页面的电影。

7.5 缓存

在各个列表页面，每次访问一个列表的一页，Django 就会遍历整个电影数据库，此行为的效率很低。另外，由于电影信息处于长期稳定的状态，更新操作并不频繁，因此，我们给各个列表加上了缓存操作，这里用到了单机的 Redis 数据库，且该数据库与 Django 服务器部署在同一节点，以降低网络延时。

[19]https://docs.djangoproject.com/en/dev/ref/paginator/

8 前端开发

8.1 前端简介

使用了 Boostrap4 前端框架，主要设计了登录主页页面、电影列表页面以及电影详情页面。使用 POST 方法将用户在页面上填写的信息上传到后台，利用后台传到前端的数据动态的展示页面。

8.2 账号管理

如图 5 所示。

登录：点击网页的右上角登录按钮可进入登录页面，若忘掉了密码可以使用邮箱找回。需要注意的是，只有用户登录之后才能评价电影，否则只能浏览，不能做任何修改。

注册：点击网页右上角的注册按钮可以进入注册页面，需要填写用户名、密码、邮箱。

个人主页：登录之后可以简单的现实个人信息，包括年龄、性别等。

(a) 登录　　(b) 注册

图 5: 账号管理

8.3 主页

如图 6 所示。

可以看到当前的电影列表，已经相应的登录和注册入口可以进行相应的用户登录和注册，并且在页面的右上方给出了电影的几种分类：推荐、热门、最新、动作、冒险、科幻、恐怖等。并且提供了搜索栏可以输入关键字进行搜索。该页面接收后台传入的电影列表，动态的展示所有的电影信息。

8.4 电影详情页

如图 7 所示。

点击任意一个电影介绍图像，可以进入电影详情页，该页面主要显示电影的封面，详情

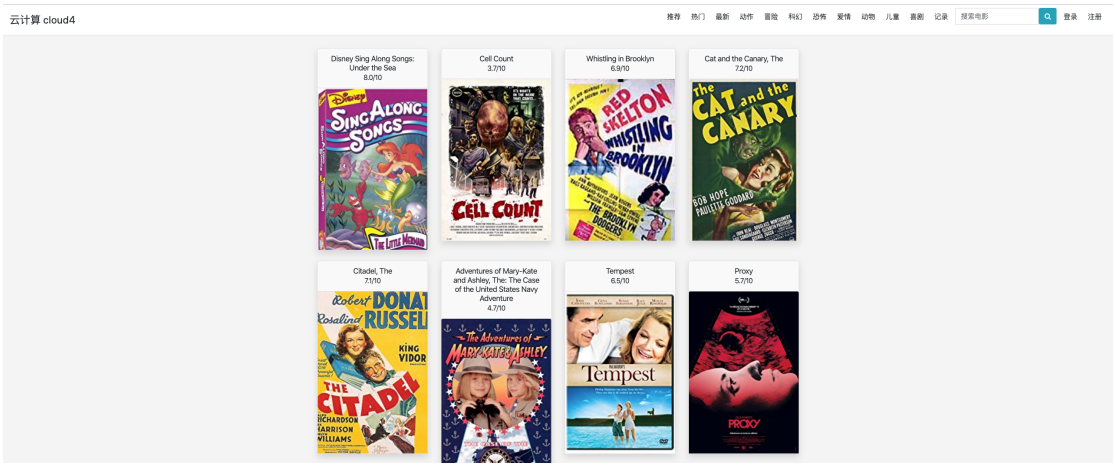

图 6: 主页

信息以及剧情介绍，提高了打分和评价功能。详情信息主要包括了上映年份、时长、编剧、导演、演员、语言以及评分。

图 7: 电影详情页

8.5 其他

其他功能页面：定义的邮箱发送信息页面、密码重置页面、重复登录提示页面等。网页基本页面：无论进入哪个页面，页面上部分是不变的。单独写成 html 文件，可以方便的被其他页面引用。项目的页面定义在项目的 moviesite/template 文件夹下，并且在 moviesite/static/base.css 文件下定义了常用的 css 样式。

9 系统集成

9.1 模块间交互

9.1.1 Nginx、uWSGI、Django 的交互

Nginx 负责负载均衡，用户请求传到 uWSGI 服务器，若为静态资源请求，则将请求转回 Nginx，返回 Nginx 服务器上的静态资源；若为动态请求，则将请求信息传给 Django，由 Django 处理后，将返回信息传回给 Nginx。

9.1.2 Django 与 Redis 的交互

Django 直接指定缓存服务器，来访问 Redis 数据库，已实现 Django 内部的缓存 API。

9.1.3 Django 与 MongoDB 的交互

官方不能很好地支持 Django 使用 MongoDB 数据库，我们使用了第三方库 djongo[20] 使得 Django 完美支持 MongoDB 数据库。

9.1.4 Django 与 Kafka 的交互

在 Django 中操作 Kafka，直接使用了支持 python 的 kafka-python[21] 库, 简单实用。

9.1.5 MongoDB 与 Spark 的交互

MongoDB 与 Spark 的交互使用了官方的 Spark Connector[22]。

9.1.6 其他

其余模块，如 python 访问 MongoDB 使用的 pymongo[23]、python 访问 HDFS 使用的 pyhdfs[24]、python 访问 Hbase 使用的 happybase[25] 等，均有较为详细的使用文档，此处不再详细列举。

[20]https://github.com/nesdis/djongo
[21]https://pypi.org/project/kafka-python/
[22]https://docs.mongodb.com/spark-connector/current/python-api/index.html
[23]https://api.mongodb.com/python/current/
[24]https://pyhdfs.readthedocs.io/en/latest/pyhdfs.html
[25]https://happybase.readthedocs.io/en/latest/

9.2 系统集成

我们将如表 4 所示的 5 台服务器上的服务器部署到 Docker 上，环境基于 Centos:7 镜像，并在内部配置了 SSH 免登录等。最终实现了多个分布式组件分散在 5 台服务器上，保证了其中任意一台服务器的宕机，整个系统均能正常运行 (在 Docker 内部的集群还有些 BUG)。

10 安装部署

10.1 部署说明

已经将 5 台服务器上的模块打包成 5 个 Docker image，其大小分别为 2G、2.3G、3.4G、4.5G、3.9G，由于存放了电影基本信息 (MongoDB 3 份)、电影图片 (Nginx 2 份) 以及评分矩阵 (HDFS 3 份)，故镜像体积有些大。5 台服务器的 Docker 载入命令已写到对应的 shell 文件里。

10.2 部署方法

1. 在每台服务器上载入 docker 镜像，如

```
docker load -i cloud1.tar
```

2. 在每台服务器上执行对应的 bash 文件，以启动 container，如

```
bash cloud1.sh
```

3. 在每台服务器上，打开 ssh 服务

```
/usr/sbin/sshd -D &
```

4. 在 cloud1、cloud2、cloud3 开启 Zookeeper

```
zkServer.sh start
```

5. 在 cloud1、cloud2、cloud3 开启 HDFS
   在 cloud1、cloud2、cloud3 启动 journalnode

```
hdfs --daemon start journalnode
```

   在 cloud1 启动 HDFS 集群

```
start-dfs.sh
```

此时可在 cloud1、cloud2 的外网 9870 端口访问两个 namenode 页面。

6. 在 cloud1、cloud2、cloud3 开启 YARN

   在 cloud2 启动 YARN 集群

```
start-yarn.sh
```

   此时可在 cloud1、cloud2 的外网 8088 端口访问两个 resourcemanager 界面。

7. 在 cloud1、cloud2、cloud3 开启 Hbase

   清除 Hbase 没清理干净的 log，不然会出错

```
rm -rf /usr/cloud302/hbase/logs/* /usr/cloud302/hbase/pid/*
```

   在 cloud1 启动 Hbase 集群

```
start-hbase.sh
```

   在 cloud2 启动第二个 hmaster 节点

```
hbase-daemon.sh start master
```

   此时可在 cloud1、cloud2 的外网 9001 端口访问两个 hmaster 界面。

8. 在 cloud3、cloud4、cloud5 启动 Kafka

```
kafka-server-start.sh -daemon $KAFKA_HOME/config/server.properties
```

9. 在 cloud3、cloud4、cloud5 启动 MongoDB

```
mongod -f $MONGODB_HOME/mongo.conf
```

10. 在 cloud4、cloud5 启动 Redis

```
redis-server /$REDIS_HOME/redis.conf
```

11. 在 cloud4、cloud5 启动 Django

```
cd /usr/cloud302/moviesite
source bin/activate
python manage.py runserver 0:8004
```

    此时可在 cloud4、cloud5 的外网 8004 端口访问 Web 界面。

    管理员账号 root 密码 62625536

    后台界面.../admin

12. 在 cloud4、cloud5 使用 uWSGI 启动 Django

    或者不经过第 11 步，在 cloud4、cloud5 采用 uWSGI 启动 Django 服务，再通过 Nginx 实现负载均衡，其中静态文件交由 Nginx 处理，首先需要收集静态文件。

```
source bin/activate
python manage.py collectstatic
```

```
uwsgi --ini /usr/cloud302/moviesite/uwsgi.ini
```

```
nginx
```

    此时可在 cloud4、cloud5 的外网 8006 端口访问 Nginx 界面。

    这时候的后台界面无法正常加载静态文件，此 bug 未修复。

13. 提交推荐任务

    在 cloud4 上可提交推荐任务，首先

```
source /etc/profile
```

    分为离线推荐 offline.py 和在线推荐 inline.py，有以下三种提交方式：

```
\\cluster 模式，无具体输出
spark-submit --master yarn --deploy-mode cluster --jars spark-streaming-
    kafka.jar,mongo-spark-connector.jar ...py

\\client 模式，有具体输出
spark-submit --master yarn --deploy-mode client --jars spark-streaming-
    kafka.jar,mongo-spark-connector.jar ...py

\\local 模式
spark-submit --jars spark-streaming-kafka.jar,mongo-spark-connector.jar
    ...py
```

    提交离线推荐，前两者会因为内存原因存在崩掉的可能性。