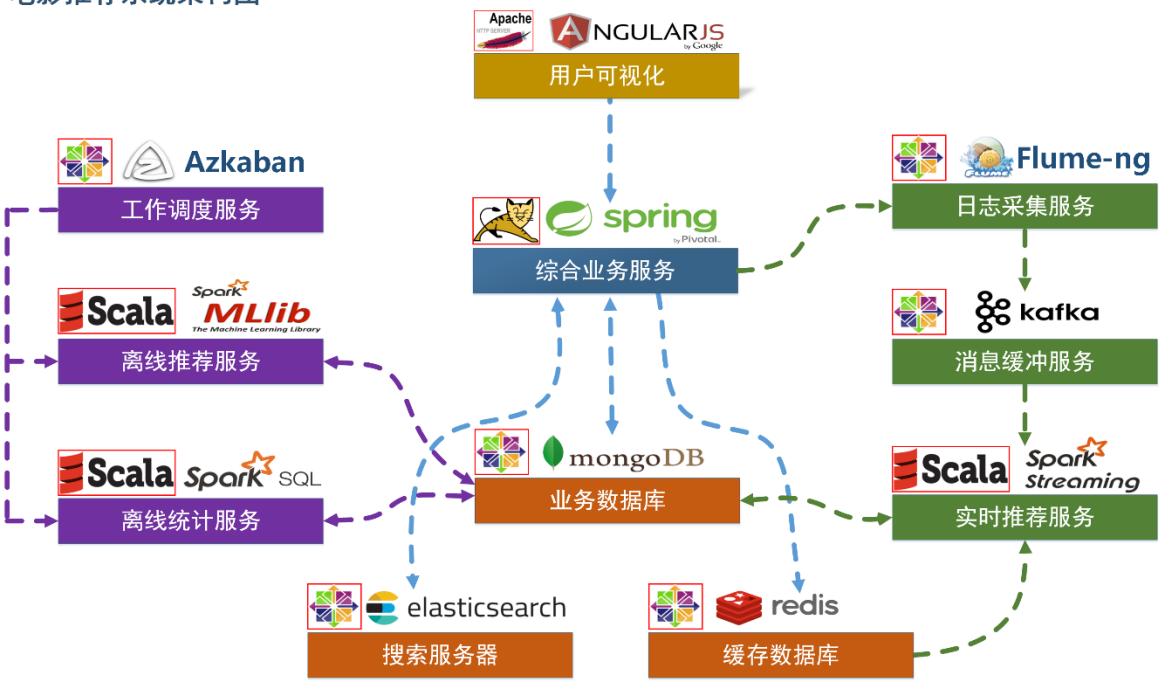
尚硅谷大数据技术之电影推荐系统

第 1 章 项目体系架构设计

1.1 项目系统架构

项目以推荐系统建设领域知名的经过修改过的 MovieLens 数据集作为依托，以某科技公司电影网站真实业务数据架构为基础，构建了符合教学体系的一体化的电影推荐系统，包含了离线推荐与实时推荐体系，综合利用了协同过滤算法以及基于内容的推荐方法来提供混合推荐。提供了从前端应用、后台服务、算法设计实现、平台部署等多方位的闭环的业务实现。

电影推荐系统架构图



用户可视化：主要负责实现和用户的交互以及业务数据的展示，主体采用 AngularJS2 进行实现，部署在 Apache 服务上。

综合业务服务：主要实现 JavaEE 层面整体的业务逻辑，通过 Spring 进行构建，对接业务需求。部署在 Tomcat 上。

【数据存储部分】

业务数据库：项目采用广泛应用的文档数据库 MongDB 作为主数据库，主要负

责平台业务逻辑数据的存储。

搜索服务器：项目爱用 ElasticSearch 作为模糊检索服务器，通过利用 ES 强大的匹配查询能力实现基于内容的推荐服务。

缓存数据库：项目采用 Redis 作为缓存数据库，主要用来支撑实时推荐系统部分对于数据的高速获取需求。

【离线推荐部分】

离线统计服务：批处理统计性业务采用 Spark Core + Spark SQL 进行实现，实现对指标类数据的统计任务。

离线推荐服务：离线推荐业务采用 Spark Core + Spark MLlib 进行实现，采用 ALS 算法进行实现。

工作调度服务：对于离线推荐部分需要以一定的时间频率对算法进行调度，采用 Azkaban 进行任务的调度。

【实时推荐部分】

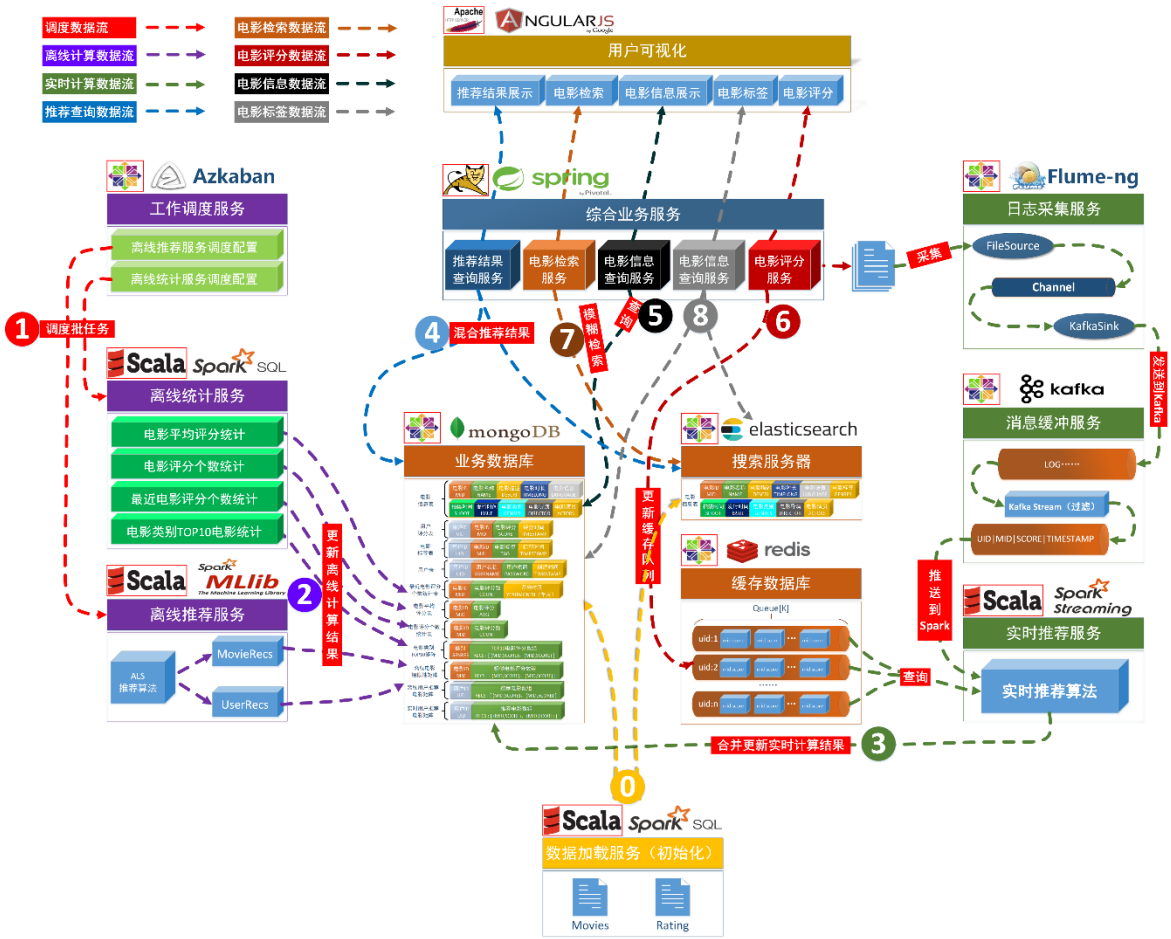
日志采集服务：通过利用 Flume-ng 对业务平台中用户对于电影的一次评分行为进行采集，实时发送到 Kafka 集群。

消息缓冲服务：项目采用 Kafka 作为流式数据的缓存组件，接受来自 Flume 的数据采集请求。并将数据推送到项目的实时推荐系统部分。

实时推荐服务：项目采用 Spark Streaming 作为实时推荐系统，通过接收 Kafka 中缓存的数据，通过设计的推荐算法实现对实时推荐的数据处理，并将结构合并更新到 MongoDB 数据库。

1.2 项目数据流程

电影推荐系统数据流图



【系统初始化部分】

0. 通过 Spark SQL 将系统初始化数据加载到 MongoDB 和 ElasticSearch 中。

【离线推荐部分】

1. 通过 Azkaban 实现对于离线统计服务以离线推荐服务的调度，通过设定的运行时间完成对任务的触发执行。
2. 离线统计服务从 MongoDB 中加载数据，将【电影平均评分统计】、【电影评分个数统计】、【最近电影评分个数统计】三个统计算法进行运行实现，并将计算结果回写到 MongoDB 中；离线推荐服务从 MongoDB 中加载数据，通过 ALS 算法分别将【用户推荐结果矩阵】、【影片相似度矩阵】回写到 MongoDB 中。

【实时推荐部分】

3. Flume 从综合业务服务的运行日志中读取日志更新，并将更新的日志实时推送到 Kafka 中；Kafka 在收到这些日志之后，通过 kafkaStream 程序对获取的日志信息进行过滤处理，获取用户评分数据流【UID|MID|SCORE|TIMESTAMP】，并发送到另外一个 Kafka 队列；Spark Streaming 监听 Kafka 队列，实时获取 Kafka 过滤出来的用户评分数据流，融合存储在 Redis 中的用户最近评分队列数据，提交给实时推荐算法，完成对用户新的推荐结果计算；计算完成之后，将新的推荐结构和 MongDB 数据库

中的推荐结果进行合并。

【业务系统部分】

4. 推荐结果展示部分，从 MongoDB、ElasticSearch 中将离线推荐结果、实时推荐结果、内容推荐结果进行混合，综合给出相对应的数据。
5. 电影信息查询服务通过对接 MongoDB 实现对电影信息的查询操作。
6. 电影评分部分，获取用户通过 UI 给出的评分动作，后台服务进行数据库记录后，一方面将数据推动到 Redis 群中，另一方面，通过预设的日志框架输出到 Tomcat 中的日志中。
7. 项目通过 ElasticSearch 实现对电影的模糊检索。
8. 电影标签部分，项目提供用户对电影打标签服务。

1.3 数据模型

1. Movie【电影数据表】

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
name	String	电影的名称	
descri	String	电影的描述	
timelong	String	电影的时长	
shoot	String	电影拍摄时间	
issue	String	电影发布时间	
language	String	电影语言	
genres	String	电影所属类别	
director	String	电影的导演	
actors	String	电影的演员	

2. Rating【用户评分表】

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
mid	Int	电影的 ID	
score	Double	电影的分值	

timestamp	Long	评分的时间	

3. Tag【电影标签表】

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
mid	Int	电影的 ID	
tag	String	电影的标签	
timestamp	Long	评分的时间	

4. User【用户表】

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
username	String	用户名	
password	String	用户密码	
first	boolean	用于是否第一次登录	
genres	List<String>	用户偏爱的电影类型	
timestamp	Long	用户创建的时间	

5. RateMoreMoviesRecently【最近电影评分个数统计表】

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
count	Int	电影的评分数	
yearmonth	String	评分的时段	yyyymm

6. RateMoreMovies【电影评分个数统计表】

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
count	Int	电影的评分数	

7. AverageMoviesScore【电影平均评分表】

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
avg	Double	电影的平均评分	

8. MovieRecs【电影相似性矩阵】

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
recs	Array[(mid:Int,score:Double)]	该电影最相似的电影集合	

9. UserRecs【用户电影推荐矩阵】

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
recs	Array[(mid:Int,score:Double)]	推荐给该用户的电影集合	

10. StreamRecs【用户实时电影推荐矩阵】

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
recs	Array[(mid:Int,score:Double)]	实时推荐给该用户的电影集合	

11. GenresTopMovies【电影类别 TOP10】

字段名	字段类型	字段描述	字段备注
genres	String	电影类型	
recs	Array[(mid:Int,score:Double)]	TOP10 电影	

第 2 章 工具环境搭建

我们的项目中用到了多种工具进行数据的存储、计算、采集和传输，本章主要简单介绍设计的工具环境搭建。

如果机器的配置不足，推荐只采用一台虚拟机进行配置，而非完全分布式，将该虚拟机 CPU 的内存设置的尽可能大，推荐为 CPU > 4、MEM > 4GB。

2.1 MongoDB（单节点）环境配置

```
// 通过 WGET 下载 Linux 版本的 MongoDB
[bigdata@linux ~]$ wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel62-3.4.3.tgz
// 将压缩包解压到指定目录
[bigdata@linux backup]$ tar -xf mongodb-linux-x86_64-rhel62-3.4.3.tgz -C ~/
// 将解压后的文件移动到最终的安装目录
[bigdata@linux ~]$ mv mongodb-linux-x86_64-rhel62-3.4.3/ /usr/local/mongodb
// 在安装目录下创建 data 文件夹用于存放数据和日志
[bigdata@linux mongodb]$ mkdir /usr/local/mongodb/data/
// 在 data 文件夹下创建 db 文件夹，用于存放数据
[bigdata@linux mongodb]$ mkdir /usr/local/mongodb/data/db/
// 在 data 文件夹下创建 logs 文件夹，用于存放日志
```

```
[bigdata@linux mongodb]$ mkdir /usr/local/mongodb/data/logs/
// 在 logs 文件夹下创建 log 文件
[bigdata@linux mongodb]$ touch /usr/local/mongodb/data/logs/
mongodb.log
// 在 data 文件夹下创建 mongodb.conf 配置文件
[bigdata@linux mongodb]$ touch
/usr/local/mongodb/data/mongodb.conf
// 在 mongodb.conf 文件中输入如下内容
[bigdata@linux mongodb]$ vim ./data/mongodb.conf
   #端口号 port = 27017
   #数据目录
   dbpath = /usr/local/mongodb/data/db
   #日志目录
   logpath = /usr/local/mongodb/data/logs/mongodb.log
   #设置后台运行
   fork = true
   #日志输出方式
   logappend = true
   #开启认证
   #auth = true
```

完成 MongoDB 的安装后，启动 MongoDB 服务器：

```
// 启动 MongoDB 服务器
[bigdata@linux mongodb]$ sudo /usr/local/mongodb/bin/mongod
-config /usr/local/mongodb/data/mongodb.conf
// 访问 MongoDB 服务器
[bigdata@linux mongodb]$ /usr/local/mongodb/bin/mongo
// 停止 MongoDB 服务器
[bigdata@linux mongodb]$ sudo /usr/local/mongodb/bin/mongod
-shutdown -config /usr/local/mongodb/data/mongodb.conf
```

2.2 Redis（单节点）环境配置

```
// 通过 WGET 下载 REDIS 的源码
[bigdata@linux ~]$wget
http://download.redis.io/releases/redis-4.0.2.tar.gz
// 将源代码解压到安装目录
[bigdata@linux ~]$ tar -xf redis-4.0.2.tar.gz -C ~/
// 进入 Redis 源代码目录，编译安装
[bigdata@linux ~]$ cd redis-4.0.2/
// 安装 GCC
[bigdata@linux ~]$ sudo yum install gcc
// 编译源代码
[bigdata@linux redis-4.0.2]$ make MALLOC=libc
// 编译安装
[bigdata@linux redis-4.0.2]$ sudo make install
// 创建配置文件
[bigdata@linux redis-4.0.2]$ sudo cp ~/redis-4.0.2/redis.conf
/etc/
// 修改配置文件中以下内容
[bigdata@linux redis-4.0.2]$ sudo vim /etc/redis.conf
daemonize yes   #37 行  #是否以后台 daemon 方式运行，默认不是后台运行
pidfile /var/run/redis/redis.pid   #41 行  #redis 的 PID 文件路径（可选）
bind 0.0.0.0   #64 行  #绑定主机 IP，默认值为 127.0.0.1，我们是跨机器运行，所以需要更改
logfile /var/log/redis/redis.log   #104 行  #定义 log 文件位置，模式 log 信息定向到 stdout，输出到/dev/null（可选）
dir “/usr/local/rdbfile”  #188 行  #本地数据库存放路径，默认为./，编译安装默认存在在/usr/local/bin 下（可选）
```

在安装完 Redis 之后，启动 Redis

```
// 启动 Redis 服务器
```

```
[bigdata@linux redis-4.0.2]$ redis-server /etc/redis.conf
// 连接 Redis 服务器
[bigdata@linux redis-4.0.2]$ redis-cli
// 停止 Redis 服务器
[bigdata@linux redis-4.0.2]$ redis-cli shutdown
```

2.3 ElasticSearch（单节点）环境配置

```
// 通过 Wget 下载 ElasticSearch 安装包
[bigdata@linux ~]$wget
https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.6.2.tar.gz
```

修改 Linux 配置参数：

```
// 修改文件数配置，在文件末尾添加如下配置
[bigdata@linux ~]$sudo vim /etc/security/limits.conf
* soft nofile 65536
* hard nofile 131072
* soft nproc 2048
* hard nproc 4096
// 修改* soft nproc 1024 为 * soft nproc 2048
[bigdata@linux ~]$sudo vim /etc/security/limits.d/90-nproc.conf
* soft nproc 2048      #将该条目修改成 2048
// 在文件末尾添加：
[bigdata@linux ~]$sudo vim /etc/sysctl.conf
vm.max_map_count=655360
// 在文件末尾添加：
[bigdata@linux elasticsearch-5.6.2]$ sudo sysctl -p
```

配置 ElasticSearch：

```
// 解压 ElasticSearch 到安装目录
```

```
[bigdata@linux ~]$ tar -xf elasticsearch-5.6.2.tar.gz
-C ./cluster/
// 进入 ElasticSearch 安装目录
[bigdata@linux cluster]$ cd elasticsearch-5.6.2/
// 创建 ElasticSearch 数据文件夹 data
[bigdata@linux cluster]$ mkdir elasticsearch-5.6.2/data/
// 创建 ElasticSearch 日志文件夹 logs
[bigdata@linux cluster]$ mkdir elasticsearch-5.6.2/logs/
// 修改 ElasticSearch 配置文件
[bigdata@linux
elasticsearch-5.6.2]$ vim ./config/elasticsearch.yml
cluster.name: es-cluster  #设置集群的名称
node.name: es-node    #修改当前节点的名称
path.data: /home/bigdata/cluster/elasticsearch-5.6.2/data  #修改
数据路径
path.logs: /home/bigdata/cluster/elasticsearch-5.6.2/logs  #修改
日志路径
bootstrap.memory_lock: false    #设置 ES 节点允许内存交换
bootstrap.system_call_filter: false    #禁用系统调用过滤器
network.host: linux     #设置当前主机名称
discovery.zen.ping.unicast.hosts: ["linux"]    #设置集群的主机列表
```

完成 ElasticSearch 的配置后：

```
// 启动 ElasticSearch 服务
[bigdata@linux elasticsearch-5.6.2]$ ./bin/elasticsearch -d
// 访问 ElasticSearch 服务
[bigdata@linux elasticsearch-5.6.2]$ curl http://linux:9200/
  {
    "name" : "es-node",
    "cluster_name" : "es-cluster",
    "cluster_uuid" : "VUjWSShBS8KM_EPJdIer6g",
```

```
    "version" : {
      "number" : "5.6.2",
      "build_hash" : "57e20f3",
      "build_date" : "2017-09-23T13:16:45.703Z",
      "build_snapshot" : false,
      "lucene_version" : "6.6.1"
    },
    "tagline" : "You Know, for Search"
  }
// 停止 ElasticSearch 服务
 [bigdata@linux ~]$ jps
   8514 Elasticsearch
   6131 GradleDaemon
   8908 Jps
[bigdata@linux ~]$ kill -9 8514
```

2.4 Azkaban（单节点）环境配置

2.4.1 安装 Git

```
// 安装 GIT
[bigdata@linux ~]$ sudo yum install git
// 通过 git 下载 Azkaban 源代码
[bigdata@linux ~]$ git clone https://github.com/azkaban/azkaban.git
// 进入 azkaban 目录
[bigdata@linux ~]$ cd azkaban/
// 切换到 3.36.0 版本
[bigdata@linux azkaban]$ git checkout -b 3.36.0
```

2.4.2 编译 Azkaban

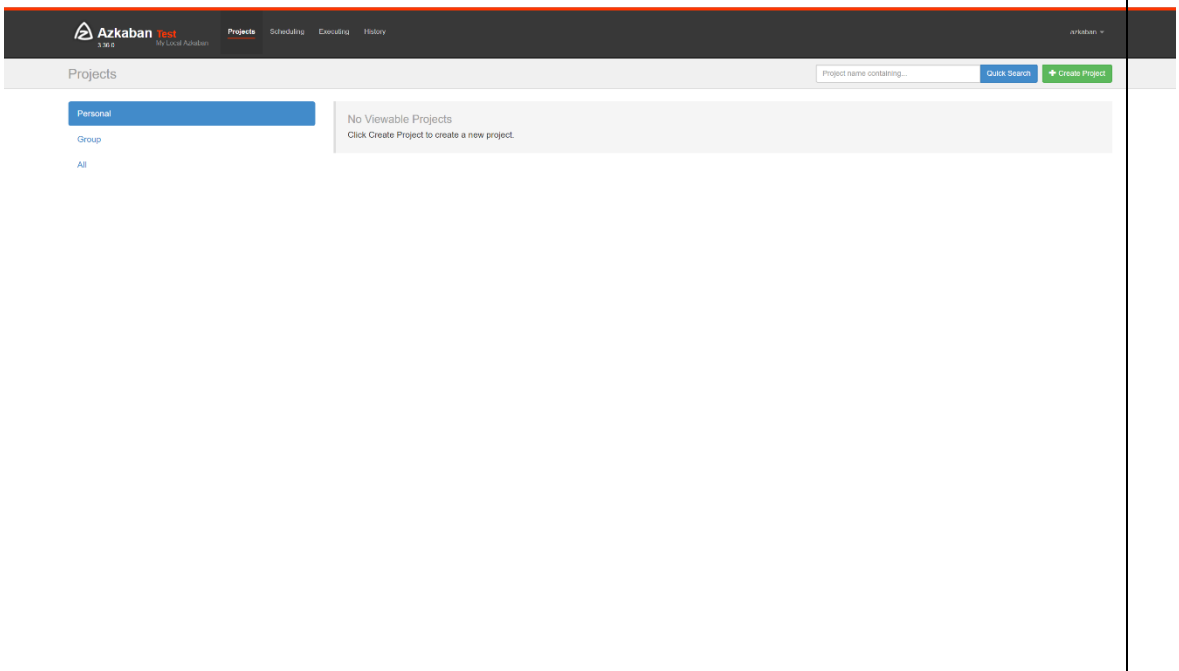
详细请参照：https://github.com/azkaban/azkaban

```
// 安装编译环境
[bigdata@linux azkaban]$sudo yum install gcc
[bigdata@linux azkaban]$sudo yum install -y gcc-c++*
```

```
// 执行编译命令
[bigdata@linux azkaban]$ ./gradlew clean build
```

2.4.3 部署 Azkaban Solo

```
// 将编译好的 azkaban 中的 azkaban-solo-server-3.36.0.tar.gz 拷贝到根目录
[bigdata@linux
azkaban]$ cp ./azkaban-solo-server/build/distributions/azkaban-solo-
server-3.36.0.tar.gz ~/
// 解压 azkaban-solo-server-3.36.0.tar.gz 到安装目录
[bigdata@linux ~]$ tar -xf azkaban-solo-server-3.36.0.tar.gz
-C ./cluster
// 启动 Azkaban Solo 单节点服务
[bigdata@linux azkaban-solo-server-3.36.0]$ bin/azkaban-solo-start.sh
// 访问 azkaban 服务，通过浏览器代开 http://ip:8081，通过用户名：azkaban，密码 azkaban
登录。
```




```
// 关闭 Azkaban 服务
[bigdata@linux
azkaban-solo-server-3.36.0]$ bin/azkaban-solo-shutdown.sh
```

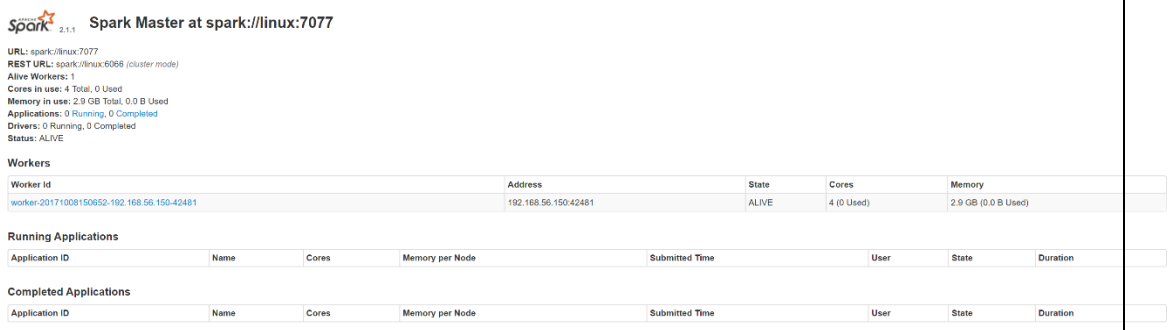
2.5 Spark（单节点）环境配置

```
// 通过 wget 下载 zookeeper 安装包
[bigdata@linux ~]$ wget
https://d3kbcqa49mib13.cloudfront.net/spark-2.1.1-bin-hadoop2.7.tgz
// 将 spark 解压到安装目录
[bigdata@linux ~]$ tar -xf spark-2.1.1-bin-hadoop2.7.tgz -C ./cluster
```

```
// 进入 spark 安装目录
[bigdata@linux cluster]$ cd spark-2.1.1-bin-hadoop2.7/
// 复制 slave 配置文件
[bigdata@linux
spark-2.1.1-bin-hadoop2.7]$ cp ./conf/slaves.template ./conf/slaves
// 修改 slave 配置文件
[bigdata@linux spark-2.1.1-bin-hadoop2.7]$ vim ./conf/slaves
linux  #在文件最后将本机主机名进行添加
// 复制 Spark-Env 配置文件
[bigdata@linux
spark-2.1.1-bin-hadoop2.7]$ cp ./conf/spark-env.sh.template ./conf/s
park-env.sh
SPARK_MASTER_HOST=linux          #添加 spark master 的主机名
SPARK_MASTER_PORT=7077           #添加 spark master 的端口号
```

安装完成之后，启动 Spark

```
// 启动 Spark 集群
[bigdata@linux spark-2.1.1-bin-hadoop2.7]$ sbin/start-all.sh
// 访问 Spark 集群，浏览器访问 http://linux:8080
```




```
// 关闭 Spark 集群
[bigdata@linux spark-2.1.1-bin-hadoop2.7]$ sbin/stop-all.sh
```

2.6 Zookeeper（单节点）环境配置

```
// 通过 wget 下载 zookeeper 安装包
[bigdata@linux ~]$ wget
http://mirror.bit.edu.cn/apache/zookeeper/zookeeper-3.4.10/zookeeper
-3.4.10.tar.gz
// 将 zookeeper 解压到安装目录
[bigdata@linux ~]$ tar -xf zookeeper-3.4.10.tar.gz -C ./cluster
// 进入 zookeeper 安装目录
[bigdata@linux cluster]$ cd zookeeper-3.4.10/
// 创建 data 数据目录
[bigdata@linux zookeeper-3.4.10]$ mkdir data/
// 复制 zookeeper 配置文件
```

```
[bigdata@linux
zookeeper-3.4.10]$ cp ./conf/zoo_sample.cfg ./conf/zoo.cfg
// 修改 zookeeper 配置文件
[bigdata@linux zookeeper-3.4.10]$ vim conf/zoo.cfg
dataDir=/home/bigdata/cluster/zookeeper-3.4.10/data  #将数据目录地址修
改为创建的目录
// 启动 Zookeeper 服务
[bigdata@linux zookeeper-3.4.10]$ bin/zkServer.sh start
// 查看 Zookeeper 服务状态
[bigdata@linux zookeeper-3.4.10]$ bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using                                                    config:
/home/bigdata/cluster/zookeeper-3.4.10/bin/../conf/zoo.cfg
Mode: standalone
// 关闭 Zookeeper 服务
[bigdata@linux zookeeper-3.4.10]$ bin/zkServer.sh stop
```

2.7 Flume-ng（单节点）环境配置

```
// 通过 wget 下载 zookeeper 安装包
[bigdata@linux ~]$ wget
http://www.apache.org/dyn/closer.lua/flume/1.8.0/apache-flume-1.8.0-
bin.tar.gz
// 将 zookeeper 解压到安装目录
[bigdata@linux ~]$ tar -xf apache-flume-1.8.0-bin.tar.gz -C ./cluster
// 等待项目部署时使用
[bigdata@master01 apache-flume-1.7.0-kafka]$ ls
bin  CHANGELOG  conf  DEVNOTES  doap_Flume.rdf  docs  lib  LICENSE  logs  NOTICE  README.md  RELEASE-NOTES  tools
[bigdata@master01 apache-flume-1.7.0-kafka]$ bin/flume-ng agent -c ./conf/ -f ./conf/log-kafka.properties -n agent
```

2.8 Kafka（单节点）环境配置

```
// 通过 wget 下载 zookeeper 安装包
[bigdata@linux ~]$ wget
http://mirrors.tuna.tsinghua.edu.cn/apache/kafka/0.10.2.1/kafka_2.11-0.10.2.1.tgz
// 将 kafka 解压到安装目录
[bigdata@linux ~]$ tar -xf kafka_2.12-0.10.2.1.tgz -C ./cluster
// 进入 kafka 安装目录
[bigdata@linux cluster]$ cd kafka_2.12-0.10.2.1/
// 修改 kafka 配置文件
[bigdata@linux kafka_2.12-0.10.2.1]$ vim config/server.properties
```

```
host.name=linux                    #修改主机名
port=9092                          #修改服务端口号
zookeeper.connect=linux:2181       #修改 Zookeeper 服务器地址
// 启动 kafka 服务 !!! 启动之前需要启动 Zookeeper 服务
[bigdata@linux kafka_2.12-0.10.2.1]$ bin/kafka-server-start.sh
-daemon ./config/server.properties
// 关闭 kafka 服务
[bigdata@linux kafka_2.12-0.10.2.1]$ bin/kafka-server-stop.sh
// 创建 topic
[bigdata@linux kafka_2.12-0.10.2.1]$ bin/kafka-topics.sh --create
--zookeeper linux:2181 --replication-factor 1 --partitions 1 --topic
recommender
// kafka-console-producer
[bigdata@linux kafka_2.12-0.10.2.1]$ bin/kafka-console-producer.sh
--broker-list linux:9092 --topic recommender
// kafka-console-consumer
[bigdata@linux kafka_2.12-0.10.2.1]$ bin/kafka-console-consumer.sh
--bootstrap-server linux:9092 --topic recommender
```

第 3 章 创建项目并初始化业务数据

我们的项目主体用 Scala 编写，采用 IDEA 作为开发环境进行项目编写，采用 maven 作为项目构建和管理工具。

3.1 在 IDEA 中创建 maven 项目

打开 IDEA，创建一个 maven 项目，命名为 MovieRecommendSystem。为了方便后期的联调，我们会把业务系统的代码也添加进来，所以我们可以以 MovieRecommendSystem 作为父项目，并在其下建一个名为 recommender 的子项目，然后再在下面搭建多个子项目用于提供不同的推荐服务。

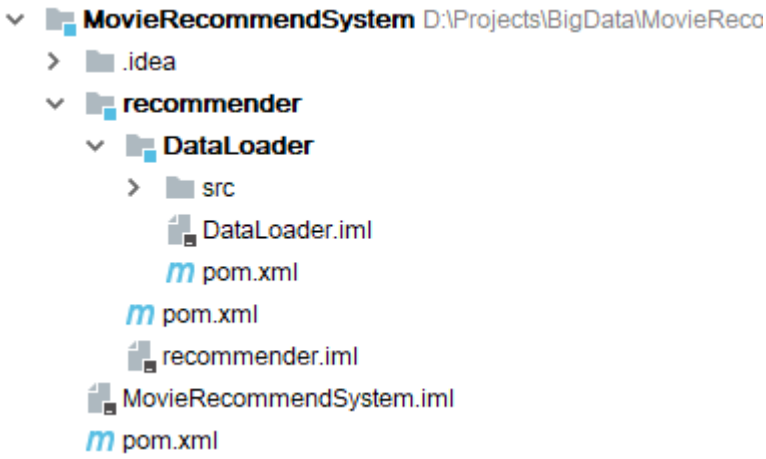
3.1.1 项目框架搭建

在 MovieRecommendSystem 的 pom.xml 文件中加入元素 <packaging>pom</packaging>，然后新建一个 maven module 作为子项目，命名为 recommender。同样的，再以 recommender 为父项目，在它的 pom.xml 中加入 <packing>pom</packaging>，然后新建一个 maven module 作为子项目。我们的第一步是初始化业务数据，所以子项目命名为 DataLoader。

父项目只是为了规范化项目结构，方便依赖管理，本身是不需要代码实现的，

所以 MovieRecommendSystem 和 recommender 下的 src 文件夹都可以删掉。

目前的整体项目框架如下：



3.1.2 声明项目中工具的版本信息

我们整个项目需要用到多个工具，它们的不同版本可能会对程序运行造成影响，所以应该在最外层的 MovieRecommendSystem 中声明所有子项目共用的版本信息。

在 pom.xml 中加入以下配置：

MovieRecommendSystem/pom.xml

```
<properties>
    <log4j.version>1.2.17</log4j.version>
    <slf4j.version>1.7.22</slf4j.version>
    <mongodb-spark.version>2.0.0</mongodb-spark.version>
    <casbah.version>3.1.1</casbah.version>
    <elasticsearch-spark.version>5.6.2</elasticsearch-spark.version>
    <elasticsearch.version>5.6.2</elasticsearch.version>
    <redis.version>2.9.0</redis.version>
    <kafka.version>0.10.2.1</kafka.version>
    <spark.version>2.1.1</spark.version>
    <scala.version>2.11.8</scala.version>
    <jblas.version>1.2.1</jblas.version>
</properties>
```

3.1.3 添加项目依赖

首先，对于整个项目而言，应该有同样的日志管理，我们在 MovieRecommendSystem 中引入公有依赖：

MovieRecommendSystem/pom.xml

```
<dependencies>
    <!--引入共同的日志管理工具 -->
```

```
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>${log4j.version}</version>
    </dependency>
</dependencies>
```

同样，对于 maven 项目的构建，可以引入公有的插件：

```
<build>
    <!--声明并引入子项目共有的插件-->
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.6.1</version>
            <!--所有的编译用 JDK1.8-->
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
    <pluginManagement>
        <plugins>
            <!--maven 的打包插件-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.0.0</version>
```

```
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <!--该插件用于将 scala 代码编译成 class 文件-->
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>3.2.2</version>
            <executions>
                <!--绑定到maven 的编译阶段-->
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</pluginManagement>
</build>
```

然后，在 recommender 模块中，我们可以为所有的推荐模块声明 spark 相关依赖（这里的 dependencyManagement 表示仅声明相关信息，子项目如果依赖需要自行引入）：

MovieRecommendSystem/recommender/pom.xml

```
<dependencyManagement>
    <dependencies>
        <!-- 引入 Spark 相关的 Jar 包 -->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
```

```
            <artifactId>spark-sql_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-mllib_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-graphx_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

由于各推荐模块都是 scala 代码，还应该引入 scala-maven-plugin 插件，用于 scala 程序的编译。因为插件已经在父项目中声明，所以这里不需要再声明版本和具体配置：

```
<build>
    <plugins>
        <!-- 父项目已声明该plugin，子项目在引入的时候，不用声明版本和已经声明的配置 -->
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

对于具体的 DataLoader 子项目，需要 spark 相关组件，还需要 mongodb、elastic search 的相关依赖，我们在 pom.xml 文件中引入所有依赖（在父项目中已声明的不需要再加详细信息）：

MovieRecommendSystem/recommender/DataLoader/pom.xml

```xml
<dependencies>
    <!-- Spark 的依赖引入 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
    </dependency>
    <!-- 引入 Scala -->
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
    </dependency>
    <!-- 加入 MongoDB 的驱动 -->
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>casbah-core_2.11</artifactId>
        <version>${casbah.version}</version>
    </dependency>
    <dependency>
        <groupId>org.mongodb.spark</groupId>
        <artifactId>mongo-spark-connector_2.11</artifactId>
        <version>${mongodb-spark.version}</version>
    </dependency>
    <!-- 加入 ElasticSearch 的驱动 -->
    <dependency>
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>transport</artifactId>
        <version>${elasticsearch.version}</version>
    </dependency>
    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch-spark-20_2.11</artifactId>
        <version>${elasticsearch-spark.version}</version>
        <!-- 将不需要依赖的包从依赖路径中除去 -->
        <exclusions>
            <exclusion>
                <groupId>org.apache.hive</groupId>
                <artifactId>hive-service</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
```

```
</dependencies>
```

至此，我们做数据加载需要的依赖都已配置好，可以开始写代码了。

3.2 数据加载准备

在 src/main/目录下，可以看到已有的默认源文件目录是 java，我们可以将其改名为 scala。将数据文件 movies.csv，ratings.csv，tags.csv 复制到资源文件目录 src/main/resources 下，我们将从这里读取数据并加载到 mongodb 和 elastic search 中。

3.2.1 Movies 数据集

数据格式：

mid,name,descri,timelong,issue,shoot,language,genres,actors,directors

e.g.

1^Toy Story (1995)^ ^81 minutes^March 20, 2001^1995^English ^Adventure|Animation|Children|Comedy|Fantasy ^Tom Hanks|Tim Allen|Don Rickles|Jim Varney|Wallace Shawn|John Ratzenberger|Annie Potts|John Morris|Erik von Detten|Laurie Metcalf|R. Lee Ermey|Sarah Freeman|Penn Jillette|Tom Hanks|Tim Allen|Don Rickles|Jim Varney|Wallace Shawn ^John Lasseter

Movie 数据集有 10 个字段，每个字段之间通过“^”符号进行分割。

字段名	字段类型	字段描述	字段备注
mid	Int	电影的 ID	
name	String	电影的名称	
descri	String	电影的描述	
timelong	String	电影的时长	
shoot	String	电影拍摄时间	
issue	String	电影发布时间	
language	Array[String]	电影语言	每一项用“|”分割
genres	Array[String]	电影所属类别	每一项用“|”分割
director	Array[String]	电影的导演	每一项用“|”分割
actors	Array[String]	电影的演员	每一项用“|”分割

3.1.2 Ratings 数据集

数据格式：

userId,movieId,rating,timestamp

e.g.

1,31,2.5,1260759144

Rating 数据集有 4 个字段，每个字段之间通过“，”分割。

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
mid	Int	电影的 ID	
score	Double	电影的分值	
timestamp	Long	评分的时间	

3.1.3 Tag 数据集

数据格式：

userId,movieId,tag,timestamp

e.g.

1,31,action,1260759144

Rating 数据集有 4 个字段，每个字段之间通过“，”分割。

字段名	字段类型	字段描述	字段备注
uid	Int	用户的 ID	
mid	Int	电影的 ID	
tag	String	电影的标签	
timestamp	Long	标签的时间	

3.1.4 日志管理配置文件

log4j 对日志的管理，需要通过配置文件来生效。在 src/main/resources 下新建配置文件 log4j.properties，写入以下内容：

```
log4j.rootLogger=info, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p --- [%50t] %-80c(line:%5L) : %m%n
```

3.2 数据初始化到 MongoDB

3.2.1 启动 MongoDB 数据库（略）

3.2.2 数据加载程序主体实现

我们会为原始数据定义几个样例类，通过 SparkContext 的 textFile 方法从文件中读取数据，并转换成 DataFrame，再利用 Spark SQL 提供的 write 方法进行数据的分布式插入。

在 DataLoader/src/main/scala 下新建 package，命名为 com.atguigu.recommender，新建名为 DataLoader 的 scala class 文件。

程序主体代码如下：

DataLoader/src/main/scala/com.atguigu.recommerder/DataLoader.scala

```
// 定义样例类
case class Movie(mid: Int, name: String, descri: String, timelong: String, issue: String,
                 shoot: String, language: String, genres: String, actors: String,
                 directors: String)
case class Rating(uid: Int, mid: Int, score: Double, timestamp: Int)
case class Tag(uid: Int, mid: Int, tag: String, timestamp: Int)
case class MongoConfig(uri:String, db:String)
case class ESConfig(httpHosts:String, transportHosts:String, index:String,
                    clustername:String)

object DataLoader {
  // 以window下为例，需替换成自己的路径，linux下为 /YOUR_PATH/resources/movies.csv
  val MOVIE_DATA_PATH = " YOUR_PATH\\resources\\movies.csv"
  val RATING_DATA_PATH = " YOUR_PATH\\resources\\ratings.csv"
  val TAG_DATA_PATH = "YOUR_PATH\\resources\\tags.csv"

  val MONGODB_MOVIE_COLLECTION = "Movie"
  val MONGODB_RATING_COLLECTION = "Rating"
  val MONGODB_TAG_COLLECTION = "Tag"
  val ES_MOVIE_INDEX = "Movie"

  // 主程序的入口
  def main(args: Array[String]): Unit = {
    // 定义用到的配置参数
```

```
val config = Map(
  "spark.cores" -> "local[*]",
  "mongo.uri" -> "mongodb://localhost:27017/recommender",
  "mongo.db" -> "recommender",
  "es.httpHosts" -> "localhost:9200",
  "es.transportHosts" -> "localhost:9300",
  "es.index" -> "recommender",
  "es.cluster.name" -> "elasticsearch"
)
// 创建一个 SparkConf 配置
val sparkConf = new
    SparkConf().setAppName("DataLoader").setMaster(config("spark.cores"))
// 创建一个 SparkSession
val spark = SparkSession.builder().config(sparkConf).getOrCreate()

// 在对 DataFrame 和 Dataset 进行操作许多操作都需要这个包进行支持
import spark.implicits._

// 将 Movie、Rating、Tag 数据集加载进来
val movieRDD = spark.sparkContext.textFile(MOVIE_DATA_PATH)
  //将 MovieRDD 装换为 DataFrame
val movieDF = movieRDD.map(item =>{
  val attr = item.split("\\^")
        Movie(attr(0).toInt,attr(1).trim,attr(2).trim,attr(3).trim,attr(4).trim,
        attr(5).trim,attr(6).trim,attr(7).trim,attr(8).trim,attr(9).trim)
}).toDF()

val ratingRDD = spark.sparkContext.textFile(RATING_DATA_PATH)
  //将 ratingRDD 转换为 DataFrame
val ratingDF = ratingRDD.map(item => {
  val attr = item.split(",")
  Rating(attr(0).toInt,attr(1).toInt,attr(2).toDouble,attr(3).toInt)
}).toDF()

val tagRDD = spark.sparkContext.textFile(TAG_DATA_PATH)
  //将 tagRDD 装换为 DataFrame
val tagDF = tagRDD.map(item => {
  val attr = item.split(",")
  Tag(attr(0).toInt,attr(1).toInt,attr(2).trim,attr(3).toInt)
}).toDF()

// 声明一个隐式的配置对象
implicit val mongoConfig =
```

```
                MongoConfig(config.get("mongo.uri").get,config.get("mongo.db").get)
    // 将数据保存到MongoDB 中
    storeDataInMongoDB(movieDF, ratingDF, tagDF)

    import org.apache.spark.sql.functions._
    val newTag = tagDF.groupBy($"mid")
                .agg(concat_ws("|",collect_set($"tag"))
                .as("tags"))
                .select("mid","tags")
    // 需要将处理后的Tag 数据，和Moive 数据融合，产生新的Movie 数据
    val movieWithTagsDF = movieDF.join(newTag,Seq("mid","mid"),"left")

    // 声明了一个ES 配置的隐式参数
    implicit  val esConfig = ESConfig(config.get("es.httpHosts").get,
                            config.get("es.transportHosts").get,
                            config.get("es.index").get,
                            config.get("es.cluster.name").get)
    // 需要将新的Movie 数据保存到ES 中
    storeDataInES(movieWithTagsDF)
    // 关闭Spark
    spark.stop()
  }
```

3.2.3 将数据写入 MongoDB

接下来，实现 storeDataInMongo 方法，将数据写入 mongodb 中：

```
def storeDataInMongoDB(movieDF: DataFrame, ratingDF:DataFrame, tagDF:DataFrame)
          (implicit mongoConfig: MongoConfig): Unit = {

  //新建一个到MongoDB 的连接
  val mongoClient = MongoClient(MongoClientURI(mongoConfig.uri))
  //如果MongoDB 中有对应的数据库，那么应该删除
  mongoClient(mongoConfig.db)(MONGODB_MOVIE_COLLECTION).dropCollection()
  mongoClient(mongoConfig.db)(MONGODB_RATING_COLLECTION).dropCollection()
  mongoClient(mongoConfig.db)(MONGODB_TAG_COLLECTION).dropCollection()

  //将当前数据写入到MongoDB
  movieDF
    .write
    .option("uri",mongoConfig.uri)
    .option("collection",MONGODB_MOVIE_COLLECTION)
    .mode("overwrite")
    .format("com.mongodb.spark.sql")
    .save()
```

```
  ratingDF
    .write
    .option("uri",mongoConfig.uri)
    .option("collection",MONGODB_RATING_COLLECTION)
    .mode("overwrite")
    .format("com.mongodb.spark.sql")
    .save()
  tagDF
    .write
    .option("uri",mongoConfig.uri)
    .option("collection",MONGODB_TAG_COLLECTION)
    .mode("overwrite")
    .format("com.mongodb.spark.sql")
    .save()

  //对数据表建索引
  mongoClient(mongoConfig.db)(MONGODB_MOVIE_COLLECTION).createIndex(MongoDBObject("mid" -> 1))
  mongoClient(mongoConfig.db)(MONGODB_RATING_COLLECTION).createIndex(MongoDBObject("uid" -> 1))
  mongoClient(mongoConfig.db)(MONGODB_RATING_COLLECTION).createIndex(MongoDBObject("mid" -> 1))
  mongoClient(mongoConfig.db)(MONGODB_TAG_COLLECTION).createIndex(MongoDBObject("uid" -> 1))
  mongoClient(mongoConfig.db)(MONGODB_TAG_COLLECTION).createIndex(MongoDBObject("mid" -> 1))
  //关闭 MongoDB 的连接
  mongoClient.close()
}
```

3.3 数据初始化到 ElasticSearch

3.3.1 启动 ElasticSearch 服务器（略）

3.3.2 将数据写入 ElasticSearch

与上节类似，同样主要通过 Spark SQL 提供的 write 方法进行数据的分布式插入，实现 storeDataInES 方法：

```
def storeDataInES(movieDF:DataFrame)(implicit eSConfig: ESConfig): Unit = {
  //新建一个配置
  val settings:Settings = Settings.builder()
                          .put("cluster.name",eSConfig.clustername).build()
  //新建一个 ES 的客户端
  val esClient = new PreBuiltTransportClient(settings)
```

```
  //需要将 TransportHosts 添加到 esClient 中
  val REGEX_HOST_PORT = "(.+):(\\d+)".r
  eSConfig.transportHosts.split(",").foreach{
    case REGEX_HOST_PORT(host:String,port:String) => {
      esClient.addTransportAddress(new
          InetSocketTransportAddress(InetAddress.getByName(host),port.toInt))
    }
  }
  //需要清除掉 ES 中遗留的数据
  if(esClient.admin().indices().exists(new
          IndicesExistsRequest(eSConfig.index)).actionGet().isExists){
      esClient.admin().indices().delete(new DeleteIndexRequest(eSConfig.index))
  }
  esClient.admin().indices().create(new CreateIndexRequest(eSConfig.index))

  //将数据写入到 ES 中
  movieDF
    .write
    .option("es.nodes",eSConfig.httpHosts)
    .option("es.http.timeout","100m")
    .option("es.mapping.id","mid")
    .mode("overwrite")
    .format("org.elasticsearch.spark.sql")
    .save(eSConfig.index+"/"+ES_MOVIE_INDEX)
}
```

第 4 章 离线推荐服务建设

4.1 离线推荐服务

离线推荐服务是综合用户所有的历史数据，利用设定的离线统计算法和离线推荐算法周期性的进行结果统计与保存，计算的结果在一定时间周期内是固定不变的，变更的频率取决于算法调度的频率。

离线推荐服务主要计算一些可以预先进行统计和计算的指标，为实时计算和前端业务相应提供数据支撑。

离线推荐服务主要分为统计性算法、基于 ALS 的协同过滤推荐算法以及基于 ElasticSearch 的内容推荐算法。

在 recommender 下新建子项目 StatisticsRecommender，pom.xml 文件中只需引入 spark、scala 和 mongodb 的相关依赖：

```
<dependencies>
    <!-- Spark 的依赖引入 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
    </dependency>
    <!-- 引入 Scala -->
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
    </dependency>
    <!-- 加入 MongoDB 的驱动 -->
    <!-- 用于代码方式连接 MongoDB -->
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>casbah-core_2.11</artifactId>
        <version>${casbah.version}</version>
    </dependency>
    <!-- 用于 Spark 和 MongoDB 的对接 -->
    <dependency>
        <groupId>org.mongodb.spark</groupId>
        <artifactId>mongo-spark-connector_2.11</artifactId>
        <version>${mongodb-spark.version}</version>
    </dependency>
</dependencies>
```

在 resources 文件夹下引入 log4j.properties，然后在 src/main/scala 下新建 scala 单例对象 com.atguigu.statistics.StatisticsRecommender。

同样，我们应该先建好样例类，在 main()方法中定义配置、创建 SparkSession 并加载数据，最后关闭 spark。代码如下：

src/main/scala/com.atguigu.statistics/StatisticsRecommender.scala

```
case class Movie(mid: Int, name: String, descri: String, timelong: String, issue: String,
shoot: String, language: String, genres: String, actors: String, directors: String)

case class Rating(uid: Int, mid: Int, score: Double, timestamp: Int)

case class MongoConfig(uri:String, db:String)
```

```
case class Recommendation(mid:Int, score:Double)
case class GenresRecommendation(genres:String, recs:Seq[Recommendation])

object StatisticsRecommender {

  val MONGODB_RATING_COLLECTION = "Rating"
  val MONGODB_MOVIE_COLLECTION = "Movie"

  //统计的表的名称
  val RATE_MORE_MOVIES = "RateMoreMovies"
  val RATE_MORE_RECENTLY_MOVIES = "RateMoreRecentlyMovies"
  val AVERAGE_MOVIES = "AverageMovies"
  val GENRES_TOP_MOVIES = "GenresTopMovies"

  // 入口方法
  def main(args: Array[String]): Unit = {

    val config = Map(
      "spark.cores" -> "local[*]",
      "mongo.uri" -> "mongodb://localhost:27017/recommender",
      "mongo.db" -> "recommender"
    )

    //创建SparkConf 配置
    val sparkConf = new
SparkConf().setAppName("StatisticsRecommender").setMaster(config("spark.cores"))
    //创建SparkSession
    val spark = SparkSession.builder().config(sparkConf).getOrCreate()

    val mongoConfig = MongoConfig(config("mongo.uri"),config("mongo.db"))

    //加入隐式转换
    import spark.implicits._

    //数据加载进来
    val ratingDF = spark
      .read
      .option("uri",mongoConfig.uri)
      .option("collection",MONGODB_RATING_COLLECTION)
      .format("com.mongodb.spark.sql")
      .load()
      .as[Rating]
      .toDF()
```

```
    val movieDF = spark
      .read
      .option("uri",mongoConfig.uri)
      .option("collection",MONGODB_MOVIE_COLLECTION)
      .format("com.mongodb.spark.sql")
      .load()
      .as[Movie]
      .toDF()

    //创建一张名叫ratings 的表
    ratingDF.createOrReplaceTempView("ratings")


    //TODO: 不同的统计推荐结果


    spark.stop()
}
```

4.2 离线统计服务

4.2.4 历史热门电影统计

根据所有历史评分数据，计算历史评分次数最多的电影。

实现思路：

通过 Spark SQL 读取评分数据集，统计所有评分中评分数最多的电影，然后按照从大到小排序，将最终结果写入 MongoDB 的 RateMoreMovies 数据集中。

```
//统计所有历史数据中每个电影的评分数
//数据结构 -》  mid,count
val rateMoreMoviesDF = spark.sql("select mid, count(mid) as count from ratings
group by mid")


rateMoreMoviesDF
  .write
  .option("uri",mongoConfig.uri)
  .option("collection",RATE_MORE_MOVIES)
  .mode("overwrite")
  .format("com.mongodb.spark.sql")
  .save()
```

4.2.2 最近热门电影统计

根据评分，按月为单位计算最近时间的月份里面评分数最多的电影集合。

实现思路：

通过 Spark SQL 读取评分数据集，通过 UDF 函数将评分的数据时间修改为月，然后统计每月电影的评分数。统计完成之后将数据写入到 MongoDB 的 RateMoreRecentlyMovies 数据集中。

```
//统计以月为单位拟每个电影的评分数
//数据结构 -》 mid,count,time

//创建一个日期格式化工具
val simpleDateFormat = new SimpleDateFormat("yyyyMM")

//注册一个UDF 函数，用于将 timestamp 装换成年月格式   1260759144000  => 201605
spark.udf.register("changeDate",(x:Int) => simpleDateFormat.format(new Date(x *
1000L)).toInt)

// 将原来的Rating 数据集中的时间转换成年月的格式
val ratingOfYearMonth = spark.sql("select mid, score, changeDate(timestamp) as yearmonth
from ratings")

// 将新的数据集注册成为一张表
ratingOfYearMonth.createOrReplaceTempView("ratingOfMonth")

val rateMoreRecentlyMovies = spark.sql("select mid, count(mid) as count ,yearmonth from
ratingOfMonth group by yearmonth,mid")

rateMoreRecentlyMovies
  .write
  .option("uri",mongoConfig.uri)
  .option("collection",RATE_MORE_RECENTLY_MOVIES)
  .mode("overwrite")
  .format("com.mongodb.spark.sql")
  .save()
```

4.2.3 电影平均得分统计

根据历史数据中所有用户对电影的评分，周期性的计算每个电影的平均得分。

实现思路：

通过 Spark SQL 读取保存在 MongDB 中的 Rating 数据集，通过执行以下 SQL 语句实现对于电影的平均分统计：

```
//统计每个电影的平均评分
val averageMoviesDF = spark.sql("select mid, avg(score) as avg from ratings group by mid")

averageMoviesDF
  .write
  .option("uri",mongoConfig.uri)
  .option("collection",AVERAGE_MOVIES)
  .mode("overwrite")
  .format("com.mongodb.spark.sql")
  .save()
```

统计完成之后将生成的新的 DataFrame 写出到 MongoDB 的 AverageMoviesScore 集合中。

4.2.4 每个类别优质电影统计

根据提供的所有电影类别，分别计算每种类型的电影集合中评分最高的 10 个电影。

实现思路：

在计算完整个电影的平均得分之后，将影片集合与电影类型做笛卡尔积，然后过滤掉电影类型不符合的条目，将 DataFrame 输出到 MongoDB 的 GenresTopMovies 集合中。

```
//统计每种电影类型中评分最高的10个电影
val movieWithScore = movieDF.join(averageMoviesDF,Seq("mid"))
//所有的电影类别
val genres =
List("Action","Adventure","Animation","Comedy","Crime","Documentary","Drama","Family","Fantasy","Foreign","History","Horror","Music","Mystery"
  ,"Romance","Science","Tv","Thriller","War","Western")

//将电影类别转换成RDD
val genresRDD = spark.sparkContext.makeRDD(genres)

//计算电影类别top10
val genrenTopMovies = genresRDD.cartesian(movieWithScore.rdd)
  .filter{
    case (genres,row) =>
row.getAs[String]("genres").toLowerCase.contains(genres.toLowerCase)
  }
  .map{
    // 将整个数据集的数据量减小，生成RDD[String,Iter[mid,avg]]
```

```
    case (genres,row) => {
      (genres,(row.getAs[Int]("mid"), row.getAs[Double]("avg")))
    }
  }.groupByKey()
  .map{
    case (genres, items) => GenresRecommendation(genres,items.toList.sortWith(_._2 >
_._2).take(10).map(item => Recommendation(item._1,item._2)))
  }.toDF()
// 输出数据到MongoDB
genrenTopMovies
  .write
  .option("uri",mongoConfig.uri)
  .option("collection",GENRES_TOP_MOVIES)
  .mode("overwrite")
  .format("com.mongodb.spark.sql")
  .save()
```

4.3 基于隐语义模型的协同过滤推荐

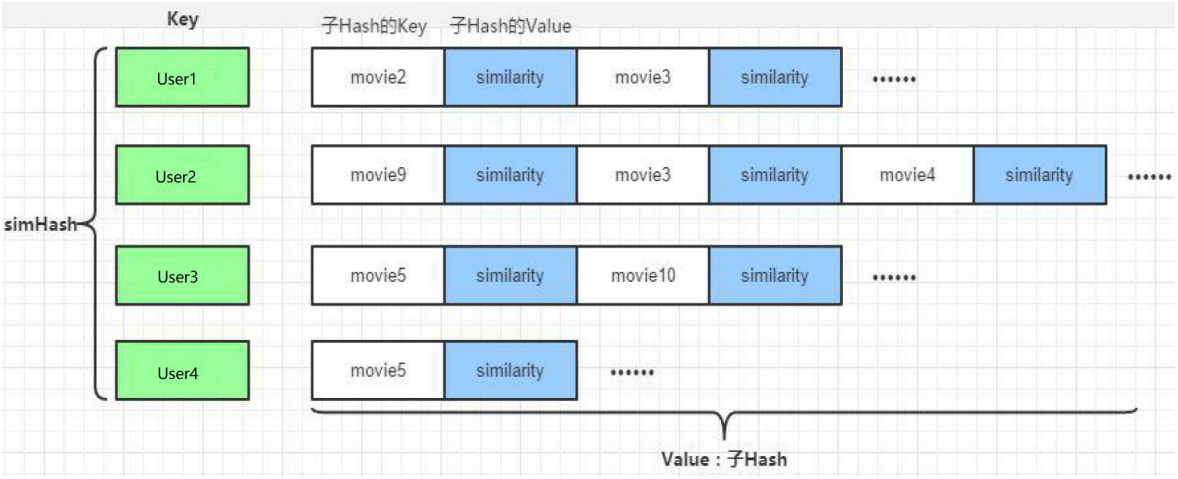
项目采用 ALS 作为协同过滤算法，分别根据 MongoDB 中的用户评分表和电影数据集计算用户电影推荐矩阵以及电影相似度矩阵。

4.3.1 用户电影推荐矩阵

通过 ALS 训练出来的 Model 来计算所有当前用户电影的推荐矩阵，主要思路如下：

1. UserId 和 MovieID 做笛卡尔积，产生（uid，mid）的元组
2. 通过模型预测（uid，mid）的元组。
3. 将预测结果通过预测分值进行排序。
4. 返回分值最大的 K 个电影，作为当前用户的推荐。

最后生成的数据结构如下：将数据保存到 MongoDB 的 UserRecs 表中



新建 recommender 的子项目 OfflineRecommender，引入 spark、scala、mongo 和 jblas 的依赖：

```
<dependencies>

    <dependency>
        <groupId>org.scalanlp</groupId>
        <artifactId>jblas</artifactId>
        <version>${jblas.version}</version>
    </dependency>

    <!-- Spark 的依赖引入 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-mllib_2.11</artifactId>
    </dependency>
    <!-- 引入 Scala -->
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
    </dependency>

    <!-- 加入 MongoDB 的驱动 -->
    <!-- 用于代码方式连接 MongoDB -->
```

```
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>casbah-core_2.11</artifactId>
        <version>${casbah.version}</version>
    </dependency>
    <!-- 用于Spark和MongoDB的对接 -->
    <dependency>
        <groupId>org.mongodb.spark</groupId>
        <artifactId>mongo-spark-connector_2.11</artifactId>
        <version>${mongodb-spark.version}</version>
    </dependency>
</dependencies>
```

同样经过前期的构建样例类、声明配置、创建 SparkSession 等步骤，可以加载数据开始计算模型了。

核心代码如下：

src/main/scala/com.atguigu.offline/OfflineRecommender.scala

```
case class Movie(mid: Int, name: String, descri: String, timelong: String, issue: String,
shoot: String,language: String, genres: String, actors: String, directors: String )

case class MovieRating(uid: Int, mid: Int, score: Double, timestamp: Int)

case class MongoConfig(uri:String, db:String)

// 标准推荐对象，mid,score
case class Recommendation(mid: Int, score:Double)

// 用户推荐
case class UserRecs(uid: Int, recs: Seq[Recommendation])

// 电影相似度（电影推荐）
case class MovieRecs(mid: Int, recs: Seq[Recommendation])

object OfflineRecommmeder {

  // 定义常量
  val MONGODB_RATING_COLLECTION = "Rating"
  val MONGODB_MOVIE_COLLECTION = "Movie"

  // 推荐表的名称
  val USER_RECS = "UserRecs"
  val MOVIE_RECS = "MovieRecs"
```

```
  val USER_MAX_RECOMMENDATION = 20

  def main(args: Array[String]): Unit = {
    // 定义配置
    val config = Map(
      "spark.cores" -> "local[*]",
      "mongo.uri" -> "mongodb://localhost:27017/recommender",
      "mongo.db" -> "recommender"
    )

    // 创建 spark session
    val sparkConf = new
SparkConf().setMaster(config("spark.cores")).setAppName("StatisticsRecommender")
    val spark = SparkSession.builder().config(sparkConf).getOrCreate()

    implicit val mongoConfig = MongoConfig(config("mongo.uri"),config("mongo.db"))

    import spark.implicits._
    //读取 mongoDB 中的业务数据
    val ratingRDD = spark
      .read
      .option("uri",mongoConfig.uri)
      .option("collection",MONGODB_RATING_COLLECTION)
      .format("com.mongodb.spark.sql")
      .load()
      .as[MovieRating]
      .rdd
      .map(rating=> (rating.uid, rating.mid, rating.score)).cache()
    //用户的数据集 RDD[Int]
    val userRDD = ratingRDD.map(_._1).distinct()

   //电影数据集 RDD[Int]
    val movieRDD = spark
      .read
      .option("uri",mongoConfig.uri)
      .option("collection",MONGODB_MOVIE_COLLECTION)
      .format("com.mongodb.spark.sql")
      .load()
      .as[Movie]
      .rdd
      .map(_.mid).cache()

    //创建训练数据集
    val trainData = ratingRDD.map(x => Rating(x._1,x._2,x._3))
```

```
    // rank 是模型中隐语义因子的个数, iterations 是迭代的次数, lambda 是 ALS 的正则化参
    val (rank,iterations,lambda) = (50, 5, 0.01)
    // 调用 ALS 算法训练隐语义模型
    val model = ALS.train(trainData,rank,iterations,lambda)

    //计算用户推荐矩阵
    val userMovies = userRDD.cartesian(movieRDD)
    // model 已训练好，把 id 传进去就可以得到预测评分列表 RDD[Rating] (uid,mid,rating)
    val preRatings = model.predict(userMovies)

    val userRecs = preRatings
      .filter(_.rating > 0)
      .map(rating => (rating.user,(rating.product, rating.rating)))
      .groupByKey()
      .map{
          case (uid,recs) => UserRecs(uid,recs.toList.sortWith(_._2 >
              _._2).take(USER_MAX_RECOMMENDATION).map(x => Recommendation(x._1,x._2)))
      }.toDF()

    userRecs.write
      .option("uri",mongoConfig.uri)
      .option("collection",USER_RECS)
      .mode("overwrite")
      .format("com.mongodb.spark.sql")
      .save()

    //TODO：计算电影相似度矩阵

    // 关闭 spark
    spark.stop()
  }
}
```

4.3.2 电影相似度矩阵

通过 ALS 计算电影见相似度矩阵，该矩阵用于查询当前电影的相似电影并为实时推荐系统服务。

离线计算的 ALS 算法，算法最终会为用户、电影分别生成最终的特征矩阵，分别是表示用户特征矩阵的 U(m x k)矩阵，每个用户由 k 个特征描述；表示物品特征矩阵的 V(n x k)矩阵，每个物品也由 k 个特征描述。

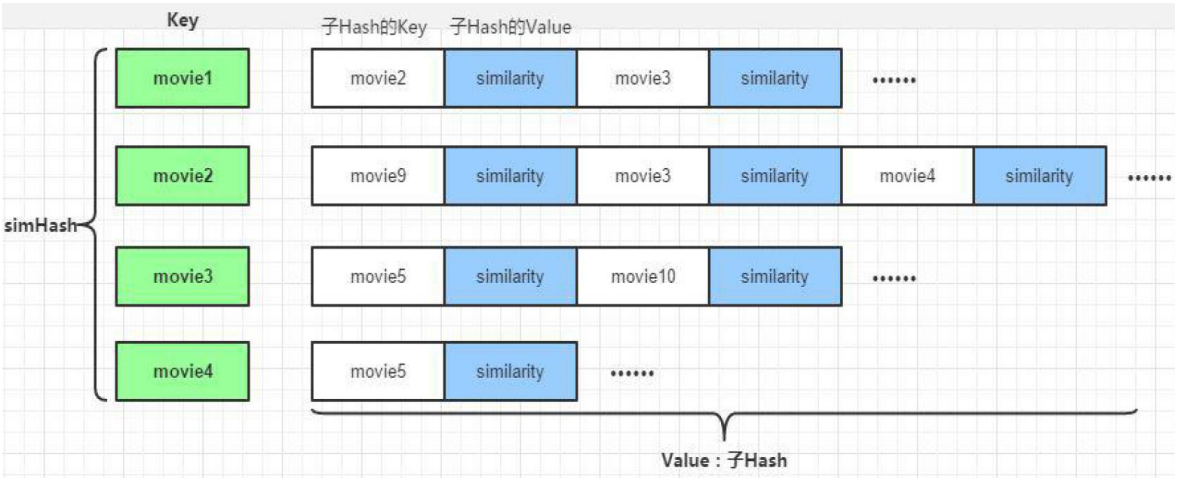
V(n x k)表示物品特征矩阵，每一行是一个 k 维向量，虽然我们并不知道每一

个维度的特征意义是什么，但是 k 个维度的数学向量表示了该行对应电影的特征。

所以，每个电影用 V(n x k)每一行的$<t_1,t_2,t_3,....,t_k>$向量表示其特征，于是任意两个电影 p：特征向量为$V_p=<t_{p1},t_{p2},t_{p3},....,t_{pk}>$，电影 q：特征向量为$V_q=<t_{q1},t_{q2},t_{q3},....,t_{qk}>$之间的相似度 sim(p,q)可以使用$V_p$和$V_q$的余弦值来表示：

$$Sim(p,q)=\frac{\sum_{i=0}^{k}(t_{pi}\times t_{qi})}{\sqrt{\sum_{i=0}^{k}t_{pi}^2}\times\sqrt{\sum_{i=0}^{k}t_{qi}^2}}$$

数据集中任意两个电影间相似度都可以由公式计算得到，电影与电影之间的相似度在一段时间内基本是固定值。最后生成的数据保存到 MongoDB 的 MovieRecs 表中。



核心代码如下：

```
//计算电影相似度矩阵
//获取电影的特征矩阵，数据格式 RDD[(scala.Int, scala.Array[scala.Double])]
val movieFeatures = model.productFeatures.map{case (mid,features) =>
  (mid, new DoubleMatrix(features))
}

// 计算笛卡尔积并过滤合并
val movieRecs = movieFeatures.cartesian(movieFeatures)
  .filter{case (a,b) => a._1 != b._1}
  .map{case (a,b) =>
    val simScore = this.consinSim(a._2,b._2) // 求余弦相似度
    (a._1,(b._1,simScore))
  }.filter(_._2._2 > 0.6)
  .groupByKey()
```

```
  .map{case (mid,items) =>
    MovieRecs(mid,items.toList.map(x => Recommendation(x._1,x._2)))
  }.toDF()

movieRecs
  .write
  .option("uri", mongoConfig.uri)
  .option("collection",MOVIE_RECS)
  .mode("overwrite")
  .format("com.mongodb.spark.sql")
  .save()
```

其中，consinSim 是求两个向量余弦相似度的函数，代码实现如下：

```
//计算两个电影之间的余弦相似度
def consinSim(movie1: DoubleMatrix, movie2:DoubleMatrix) : Double ={
  movie1.dot(movie2) / ( movie1.norm2()  * movie2.norm2() )
}
```

4.3.3 模型评估和参数选取

在上述模型训练的过程中，我们直接给定了隐语义模型的 rank,iterations,lambda 三个参数。对于我们的模型，这并不一定是最优的参数选取，所以我们需要对模型进行评估。通常的做法是计算均方根误差（RMSE），考察预测评分与实际评分之间的误差。

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{t=1}^{N}(observed_t - predicted_t)^2}$$

有了 RMSE，我们可以就可以通过多次调整参数值，来选取 RMSE 最小的一组作为我们模型的优化选择。

在 scala/com.atguigu.offline/下新建单例对象 ALSTrainer，代码主体架构如下：

```
def main(args: Array[String]): Unit = {
  val config = Map(
    "spark.cores" -> "local[*]",
    "mongo.uri" -> "mongodb://localhost:27017/recommender",
    "mongo.db" -> "recommender"
  )
  //创建 SparkConf
  val sparkConf = new
SparkConf().setAppName("ALSTrainer").setMaster(config("spark.cores"))
  //创建 SparkSession
  val spark = SparkSession.builder().config(sparkConf).getOrCreate()
```

```
  val mongoConfig = MongoConfig(config("mongo.uri"),config("mongo.db"))

  import spark.implicits._

  //加载评分数据
  val ratingRDD = spark
    .read
    .option("uri",mongoConfig.uri)
    .option("collection",OfflineRecommender.MONGODB_RATING_COLLECTION)
    .format("com.mongodb.spark.sql")
    .load()
    .as[MovieRating]
    .rdd
    .map(rating => Rating(rating.uid,rating.mid,rating.score)).cache()

  // 将一个 RDD 随机切分成两个 RDD，用以划分训练集和测试集
  val splits = ratingRDD.randomSplit(Array(0.8, 0.2))

  val trainingRDD = splits(0)
  val testingRDD = splits(1)

  //输出最优参数
  adjustALSParams(trainingRDD, testingRDD)

  //关闭 Spark
  spark.close()
}
```

其中 adjustALSParams 方法是模型评估的核心，输入一组训练数据和测试数据，输出计算得到最小 RMSE 的那组参数。代码实现如下：

```
// 输出最终的最优参数
def adjustALSParams(trainData:RDD[Rating], testData:RDD[Rating]): Unit ={
  // 这里指定迭代次数为 5，rank 和 lambda 在几个值中选取调整
  val result = for(rank <- Array(100,200,250); lambda <- Array(1, 0.1, 0.01, 0.001))
    yield {
      val model = ALS.train(trainData,rank,5,lambda)
      val rmse = getRMSE(model, testData)
      (rank,lambda,rmse)
    }
  // 按照 rmse 排序
  println(result.sortBy(_._3).head)
}
```

计算 RMSE 的函数 getRMSE 代码实现如下：

```
def getRMSE(model:MatrixFactorizationModel, data:RDD[Rating]):Double={
  val userMovies = data.map(item => (item.user,item.product))
  val predictRating = model.predict(userMovies)
  val real = data.map(item => ((item.user,item.product),item.rating))
  val predict = predictRating.map(item => ((item.user,item.product),item.rating))
  // 计算RMSE
  sqrt(
    real.join(predict).map{case ((uid,mid),(real,pre))=>
      // 真实值和预测值之间的差
      val err = real - pre
      err * err
    }.mean()
  )
}
```

运行代码，我们就可以得到目前数据的最优模型参数。

第 5 章 实时推荐服务建设

5.1 实时推荐服务

实时计算与离线计算应用于推荐系统上最大的不同在于实时计算推荐结果应该反映最近一段时间用户近期的偏好，而离线计算推荐结果则是根据用户从第一次评分起的所有评分记录来计算用户总体的偏好。

用户对物品的偏好随着时间的推移总是会改变的。比如一个用户 u 在某时刻对电影 p 给予了极高的评分，那么在近期一段时候，u 极有可能很喜欢与电影 p 类似的其他电影；而如果用户 u 在某时刻对电影 q 给予了极低的评分，那么在近期一段时候，u 极有可能不喜欢与电影 q 类似的其他电影。所以对于实时推荐，当用户对一个电影进行了评价后，用户会希望推荐结果基于最近这几次评分进行一定的更新，使得推荐结果匹配用户近期的偏好，满足用户近期的口味。

如果实时推荐继续采用离线推荐中的 ALS 算法，由于算法运行时间巨大，不具有实时得到新的推荐结果的能力；并且由于算法本身的使用的是评分表，用户本次评分后只更新了总评分表中的一项，使得算法运行后的推荐结果与用户本次评分之前的推荐结果基本没有多少差别，从而给用户一种推荐结果一直没变化的感觉，很影响用户体验。

另外，在实时推荐中由于时间性能上要满足实时或者准实时的要求，所以算法

的计算量不能太大，避免复杂、过多的计算造成用户体验的下降。鉴于此，推荐精度往往不会很高。实时推荐系统更关心推荐结果的动态变化能力，只要更新推荐结果的理由合理即可，至于推荐的精度要求则可以适当放宽。

所以对于实时推荐算法，主要有两点需求：

（1）用户本次评分后、或最近几个评分后系统可以明显的更新推荐结果；

（2）计算量不大，满足响应时间上的实时或者准实时要求；

5.2 实时推荐算法设计

当用户 u 对电影 p 进行了评分，将触发一次对 u 的推荐结果的更新。由于用户 u 对电影 p 评分，对于用户 u 来说，他与 p 最相似的电影们之间的推荐强度将发生变化，所以选取与电影 p 最相似的 K 个电影作为候选电影。

每个候选电影按照“推荐优先级”这一权重作为衡量这个电影被推荐给用户 u 的优先级。

这些电影将根据用户 u 最近的若干评分计算出各自对用户 u 的推荐优先级，然后与上次对用户 u 的实时推荐结果的进行基于推荐优先级的合并、替换得到更新后的推荐结果。

具体来说：

首先，获取用户 u 按时间顺序最近的 K 个评分，记为 RK；获取电影 p 的最相似的 K 个电影集合，记为 S；

然后，对于每个电影 q∈S ，计算其推荐优先级E_{uq}，计算公式如下：

$$E_{\text{uq}} = \frac{\sum_{r \in RK} sim(q,r) \times R_r}{sim\_sum} + \lg \max\{incount,1\} - \lg \max\{recount,1\}$$

其中：

R_r表示用户 u 对电影 r 的评分；

sim(q,r)表示电影 q 与电影 r 的相似度，设定最小相似度为 0.6，当电影 q 和电影 r 相似度低于 0.6 的阈值，则视为两者不相关并忽略；

sim_sum 表示 q 与 RK 中电影相似度大于最小阈值的个数；

incount 表示 RK 中与电影 q 相似的、且本身评分较高（>=3）的电影个数；

recount 表示 RK 中与电影 q 相似的、且本身评分较低（<3）的电影个数；

公式的意义如下：

首先对于每个候选电影 q，从 u 最近的 K 个评分中，找出与 q 相似度较高

（>=0.6）的 u 已评分电影们，对于这些电影们中的每个电影 r，将 r 与 q 的相似度乘以用户 u 对 r 的评分，将这些乘积计算平均数，作为用户 u 对电影 q 的评分预测即

$$\frac{\sum_{r \in RK} sim(q,r) \times R_r}{sim\_sum}$$

然后，将 u 最近的 K 个评分中与电影 q 相似的、且本身评分较高（>=3）的电影个数记为 incount，计算 lgmax{incount,1}作为电影 q 的“增强因子”，意义在于电影 q 与 u 的最近 K 个评分中的 n 个高评分(>=3)电影相似，则电影 q 的优先级被增加 lgmax{incount,1}。如果电影 q 与 u 的最近 K 个评分中相似的高评分电影越多，也就是说 n 越大，则电影 q 更应该被推荐，所以推荐优先级被增强的幅度较大；如果电影 q 与 u 的最近 K 个评分中相似的高评分电影越少，也就是 n 越小，则推荐优先级被增强的幅度较小；

而后，将 u 最近的 K 个评分中与电影 q 相似的、且本身评分较低（<3）的电影个数记为 recount，计算 lgmax{recount,1}作为电影 q 的“削弱因子”，意义在于电影 q 与 u 的最近 K 个评分中的 n 个低评分(<3)电影相似，则电影 q 的优先级被削减 lgmax{incount,1}。如果电影 q 与 u 的最近 K 个评分中相似的低评分电影越多，也就是说 n 越大，则电影 q 更不应该被推荐，所以推荐优先级被减弱的幅度较大；如果电影 q 与 u 的最近 K 个评分中相似的低评分电影越少，也就是 n 越小，则推荐优先级被减弱的幅度较小；

最后，将增强因子增加到上述的预测评分中，并减去削弱因子，得到最终的 q 电影对于 u 的推荐优先级。在计算完每个候选电影 q 的E_{uq}后，将生成一组<电影 q 的 ID, q 的推荐优先级>的列表 updatedList：

$$\text{updatedList} = \bigcup_{q \in S} \{qID, E_{uq}\}$$

而在本次为用户 u 实时推荐之前的上一次实时推荐结果 Rec 也是一组<电影 m,m 的推荐优先级>的列表，其大小也为 K：

$$\text{Rec} = \bigcup_{m \in \text{Rec}} \{mID, E_{um}\}, \quad len(\text{Rec}) = K$$

接下来，将 updated_S 与本次为 u 实时推荐之前的上一次实时推荐结果 Rec

进行基于合并、替换形成新的推荐结果 NewRec：

$$New\ Rec = topK(i \in Rec \cup updatedList, cmp = E_{ui})$$

其中，i 表示 updated_S 与 Rec 的电影集合中的每个电影，topK 是一个函数，表示从 Rec ∪ updated _ S 中选择出最大的 K 个电影，cmp = E_{ui} 表示 topK 函数将推荐优先级E_{ui}值最大的 K 个电影选出来。最终，NewRec 即为经过用户 u 对电影 p 评分后触发的实时推荐得到的最新推荐结果。

总之，实时推荐算法流程流程基本如下：

（1）用户 u 对电影 p 进行了评分，触发了实时推荐的一次计算；

（2）选出电影 p 最相似的 K 个电影作为集合 S；

（3）获取用户 u 最近时间内的 K 条评分，包含本次评分，作为集合 RK；

（4）计算电影的推荐优先级，产生<qID,>集合 updated_S；

将 updated_S 与上次对用户 u 的推荐结果 Rec 利用公式(4-4)进行合并，产生新的推荐结果 NewRec；作为最终输出。

我们在 recommender 下新建子项目 StreamingRecommender，引入 spark、scala、mongo、redis 和 kafka 的依赖:

```
<dependencies>
    <!-- Spark 的依赖引入 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.11</artifactId>
    </dependency>
    <!-- 引入 Scala -->
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
    </dependency>

    <!-- 加入 MongoDB 的驱动 -->
    <!-- 用于代码方式连接 MongoDB -->
```

```
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>casbah-core_2.11</artifactId>
        <version>${casbah.version}</version>
    </dependency>
    <!-- 用于 Spark 和 MongoDB 的对接 -->
    <dependency>
        <groupId>org.mongodb.spark</groupId>
        <artifactId>mongo-spark-connector_2.11</artifactId>
        <version>${mongodb-spark.version}</version>
    </dependency>

    <!-- redis -->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.9.0</version>
    </dependency>

    <!-- kafka -->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.2.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>

</dependencies>
```

代码中首先定义样例类和一个连接助手对象（用于建立 redis 和 mongo 连接），并在 StreamingRecommender 中定义一些常量：

src/main/scala/com.atguigu.streaming/StreamingRecommender.scala

```
// 连接助手对象
object ConnHelper extends Serializable{
  lazy val jedis = new Jedis("localhost")
  lazy val mongoClient =
MongoClient(MongoClientURI("mongodb://localhost:27017/recommender"))
}
```

```
case class MongConfig(uri:String,db:String)

// 标准推荐
case class Recommendation(mid:Int, score:Double)

// 用户的推荐
case class UserRecs(uid:Int, recs:Seq[Recommendation])

//电影的相似度
case class MovieRecs(mid:Int, recs:Seq[Recommendation])

object StreamingRecommender {

  val MAX_USER_RATINGS_NUM = 20
  val MAX_SIM_MOVIES_NUM = 20
  val MONGODB_STREAM_RECS_COLLECTION = "StreamRecs"
  val MONGODB_RATING_COLLECTION = "Rating"
  val MONGODB_MOVIE_RECS_COLLECTION = "MovieRecs"
  //入口方法
  def main(args: Array[String]): Unit = {
  }
}
```

实时推荐主体代码如下：

```
def main(args: Array[String]): Unit = {

  val config = Map(
    "spark.cores" -> "local[*]",
    "mongo.uri" -> "mongodb://localhost:27017/recommender",
    "mongo.db" -> "recommender",
    "kafka.topic" -> "recommender"
  )
  //创建一个 SparkConf 配置
  val sparkConf = new
SparkConf().setAppName("StreamingRecommender").setMaster(config("spark.cores"))
  val spark = SparkSession.builder().config(sparkConf).getOrCreate()
  val sc = spark.sparkContext
  val ssc = new StreamingContext(sc,Seconds(2))

  implicit val mongConfig = MongConfig(config("mongo.uri"),config("mongo.db"))
  import spark.implicits._

  // 广播电影相似度矩阵
  //装换成为 Map[Int, Map[Int,Double]]
```

```
  val simMoviesMatrix = spark
    .read
    .option("uri",config("mongo.uri"))
    .option("collection",MONGODB_MOVIE_RECS_COLLECTION)
    .format("com.mongodb.spark.sql")
    .load()
    .as[MovieRecs]
    .rdd
    .map{recs =>
      (recs.mid,recs.recs.map(x=> (x.mid,x.score)).toMap)
    }.collectAsMap()

  val simMoviesMatrixBroadCast = sc.broadcast(simMoviesMatrix)

  //创建到Kafka 的连接
  val kafkaPara = Map(
    "bootstrap.servers" -> "localhost:9092",
    "key.deserializer" -> classOf[StringDeserializer],
    "value.deserializer" -> classOf[StringDeserializer],
    "group.id" -> "recommender",
    "auto.offset.reset" -> "latest"
  )

  val kafkaStream =
KafkaUtils.createDirectStream[String,String](ssc,LocationStrategies.PreferConsisten
t,ConsumerStrategies.Subscribe[String,String](Array(config("kafka.topic")),kafkaPar
a))

  // UID|MID|SCORE|TIMESTAMP
  // 产生评分流
  val ratingStream = kafkaStream.map{case msg=>
    var attr = msg.value().split("\\|")
    (attr(0).toInt,attr(1).toInt,attr(2).toDouble,attr(3).toInt)
  }

  // 核心实时推荐算法
  ratingStream.foreachRDD{rdd =>
    rdd.map{case (uid,mid,score,timestamp) =>
      println(">>>>>>>>>>>>>>>>")

      //获取当前最近的M 次电影评分
      val userRecentlyRatings =
getUserRecentlyRating(MAX_USER_RATINGS_NUM,uid,ConnHelper.jedis)
```

```
      //获取电影P 最相似的K 个电影
      val simMovies =
getTopSimMovies(MAX_SIM_MOVIES_NUM,mid,uid,simMoviesMatrixBroadCast.value)

      //计算待选电影的推荐优先级
      val streamRecs =
computeMovieScores(simMoviesMatrixBroadCast.value,userRecentlyRatings,simMovies)

      //将数据保存到MongoDB
      saveRecsToMongoDB(uid,streamRecs)

    }.count()
  }

  //启动Streaming 程序
  ssc.start()
  ssc.awaitTermination()
}
```

5.3 实时推荐算法的实现

实时推荐算法的前提：

1. 在 Redis 集群中存储了每一个用户最近对电影的 K 次评分。实时算法可以快速获取。
2. 离线推荐算法已经将电影相似度矩阵提前计算到了 MongoDB 中。
3. Kafka 已经获取到了用户实时的评分数据。

算法过程如下：

实时推荐算法输入为一个评分<userId, movieId, rate, timestamp>，而执行的核心内容包括：获取 userId 最近 K 次评分、获取 movieId 最相似 K 个电影、计算候选电影的推荐优先级、更新对 userId 的实时推荐结果。

5.3.1 获取用户的 K 次最近评分

业务服务器在接收用户评分的时候，默认会将该评分情况以 userId, movieId, rate, timestamp 的格式插入到 Redis 中该用户对应的队列当中，在实时算法中，只需要通过 Redis 客户端获取相对应的队列内容即可。

```
import scala.collection.JavaConversions._
/**
  * 获取当前最近的M 次电影评分
  * @param num  评分的个数
```

```
  * @param uid  谁的评分
  * @return
  */
def getUserRecentlyRating(num:Int, uid:Int,jedis:Jedis): Array[(Int,Double)] ={
  //从用户的队列中取出 num 个评分
  jedis.lrange("uid:"+uid.toString, 0, num).map{item =>
    val attr = item.split("\\:")
    (attr(0).trim.toInt, attr(1).trim.toDouble)
  }.toArray
}
```

5.3.2 获取当前电影最相似的 K 个电影

在离线算法中，已经预先将电影的相似度矩阵进行了计算，所以每个电影 movieId 的最相似的 K 个电影很容易获取：从 MongoDB 中读取 MovieRecs 数据，从 movieId 在 simHash 对应的子哈希表中获取相似度前 K 大的那些电影。输出是数据类型为 Array[Int]的数组，表示与 movieId 最相似的电影集合，并命名为 candidateMovies 以作为候选电影集合。

```
/**
  * 获取当前电影 K 个相似的电影
  * @param num          相似电影的数量
  * @param mid          当前电影的 ID
  * @param uid          当前的评分用户
  * @param simMovies    电影相似度矩阵的广播变量值
  * @param mongConfig   MongoDB 的配置
  * @return
  */
def getTopSimMovies(num:Int, mid:Int, uid:Int,
simMovies:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Double]])(imp
licit mongConfig: MongConfig): Array[Int] ={
  //从广播变量的电影相似度矩阵中获取当前电影所有的相似电影
  val allSimMovies = simMovies.get(mid).get.toArray
  //获取用户已经观看过得电影
  val ratingExist =
ConnHelper.mongoClient(mongConfig.db)(MONGODB_RATING_COLLECTION).find(MongoDBObject
("uid" -> uid)).toArray.map{item =>
    item.get("mid").toString.toInt
  }
  //过滤掉已经评分过得电影，并排序输出
  allSimMovies.filter(x => !ratingExist.contains(x._1)).sortWith(_._2 >
_._2).take(num).map(x => x._1)
}
```

5.3.3 电影推荐优先级计算

对于候选电影集合 simiHash 和 userId 的最近 K 个评分 recentRatings，算法代码内容如下：

```
/**
  * 计算待选电影的推荐分数
  * @param simMovies              电影相似度矩阵
  * @param userRecentlyRatings    用户最近的k次评分
  * @param topSimMovies           当前电影最相似的K个电影
  * @return
  */
def computeMovieScores(
          simMovies:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Doub
          le]],userRecentlyRatings:Array[(Int,Double)],topSimMovies: Array[Int]):
          Array[(Int,Double)] ={

  //用于保存每一个待选电影和最近评分的每一个电影的权重得分
  val score = scala.collection.mutable.ArrayBuffer[(Int,Double)]()

  //用于保存每一个电影的增强因子数
  val increMap = scala.collection.mutable.HashMap[Int,Int]()

  //用于保存每一个电影的减弱因子数
  val decreMap = scala.collection.mutable.HashMap[Int,Int]()

  for (topSimMovie <- topSimMovies; userRecentlyRating <- userRecentlyRatings){
    val simScore = getMoviesSimScore(simMovies,userRecentlyRating._1,topSimMovie)
    if(simScore > 0.6){
      score += ((topSimMovie, simScore * userRecentlyRating._2 ))
      if(userRecentlyRating._2 > 3){
        increMap(topSimMovie) = increMap.getOrDefault(topSimMovie,0) + 1
      }else{
        decreMap(topSimMovie) = decreMap.getOrDefault(topSimMovie,0) + 1
      }
    }
  }

  score.groupBy(_._1).map{case (mid,sims) =>
    (mid,sims.map(_._2).sum / sims.length + log(increMap.getOrDefault(mid, 1)) -
log(decreMap.getOrDefault(mid, 1)))
  }.toArray.sortWith(_._2>_._2)

}
```

其中，getMovieSimScore 是取候选电影和已评分电影的相似度，代码如下：

```
/**
  * 获取当个电影之间的相似度
  * @param simMovies       电影相似度矩阵
  * @param userRatingMovie 用户已经评分的电影
  * @param topSimMovie     候选电影
  * @return
  */
def getMoviesSimScore(
simMovies:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Double]],
userRatingMovie:Int, topSimMovie:Int): Double ={
  simMovies.get(topSimMovie) match {
    case Some(sim) => sim.get(userRatingMovie) match {
      case Some(score) => score
      case None => 0.0
    }
    case None => 0.0
  }
}
```

而 log 是对数运算，这里实现为取 10 的对数（常用对数）：

```
//取 10 的对数
def log(m:Int):Double ={
  math.log(m) / math.log(10)
}
```

5.3.4 将结果保存到 mongoDB

saveRecsToMongoDB 函数实现了结果的保存：

```
/**
  * 将数据保存到MongoDB    uid -> 1,  recs -> 22:4.5|45:3.8
  * @param streamRecs  流式的推荐结果
  * @param mongConfig  MongoDB 的配置
  */
def saveRecsToMongoDB(uid:Int,streamRecs:Array[(Int,Double)])(implicit mongConfig:
MongConfig): Unit ={
  //到 StreamRecs 的连接
  val streaRecsCollection =
ConnHelper.mongoClient(mongConfig.db)(MONGODB_STREAM_RECS_COLLECTION)

  streaRecsCollection.findAndRemove(MongoDBObject("uid" -> uid))
  streaRecsCollection.insert(MongoDBObject("uid" -> uid, "recs" ->
        streamRecs.map( x => MongoDBObject("mid"->x._1,"score"->x._2)) ))
}
```

5.3.5 更新实时推荐结果

当计算出候选电影的推荐优先级的数组 updatedRecommends<movieId, E>后，这个数组将被发送到 Web 后台服务器，与后台服务器上 userId 的上次实时推荐结果 recentRecommends<movieId, E>进行合并、替换并选出优先级 E 前 K 大的电影作为本次新的实时推荐。具体而言：

a.合并：将 updatedRecommends 与 recentRecommends 并集合成为一个新的<movieId, E>数组；

b.替换（去重）：当 updatedRecommends 与 recentRecommends 有重复的电影 movieId 时，recentRecommends 中 movieId 的推荐优先级由于是上次实时推荐的结果，于是将作废，被替换成代表了更新后的 updatedRecommends 的 movieId 的推荐优先级；

c.选取 TopK：在合并、替换后的<movieId, E>数组上，根据每个 movie 的推荐优先级，选择出前 K 大的电影，作为本次实时推荐的最终结果。

5.4 实时系统联调

我们的系统实时推荐的数据流向是：业务系统 -> 日志 -> flume 日志采集 -> kafka streaming 数据清洗和预处理 -> spark streaming 流式计算。在我们完成实时推荐服务的代码后，应该与其它工具进行联调测试，确保系统正常运行。

5.4.1 启动实时系统的基本组件

启动实时推荐系统 StreamingRecommender 以及 mongodb、redis

5.4.2 启动 zookeeper

```
bin/zkServer.sh start
```

5.4.3 启动 kafka

```
bin/kafka-server-start.sh -daemon ./config/server.properties
```

5.4.4 构建 Kafka Streaming 程序

在 recommender 下新建 module，KafkaStreaming，主要用来做日志数据的预处理，过滤出需要的内容。pom.xml 文件需要引入依赖：

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
```

```
        <version>0.10.2.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.2.1</version>
    </dependency>
</dependencies>

<build>
    <finalName>kafkastream</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.atguigu.kafkastream.Application</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

在 src/main/java 下新建 java 类 com.atguigu.kafkastreaming.Application

```
public class Application {
    public static void main(String[] args){

        String brokers = "localhost:9092";
        String zookeepers = "localhost:2181";
```

```
        // 定义输入和输出的 topic
        String from = "log";
        String to = "recommender";

        // 定义 kafka streaming 的配置
        Properties settings = new Properties();
        settings.put(StreamsConfig.APPLICATION_ID_CONFIG, "logFilter");
        settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
        settings.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, zookeepers);

        StreamsConfig config = new StreamsConfig(settings);

        // 拓扑建构器
        TopologyBuilder builder = new TopologyBuilder();

        // 定义流处理的拓扑结构
        builder.addSource("SOURCE", from)
                .addProcessor("PROCESS", () -> new LogProcessor(), "SOURCE")
                .addSink("SINK", to, "PROCESS");

        KafkaStreams streams = new KafkaStreams(builder, config);
        streams.start();
    }
}
```

这个程序会将 topic 为“log”的信息流获取来做处理，并以“recommender”为新的 topic 转发出去。

流处理程序 LogProcess.java

```
public class LogProcessor implements Processor<byte[],byte[]> {
    private ProcessorContext context;

    public void init(ProcessorContext context) {
        this.context = context;
    }

    public void process(byte[] dummy, byte[] line) {
        String input = new String(line);
        // 根据前缀过滤日志信息，提取后面的内容
        if(input.contains("MOVIE_RATING_PREFIX:")){
            System.out.println("movie rating coming!!!!" + input);
            input = input.split("MOVIE_RATING_PREFIX:")[1].trim();
            context.forward("logProcessor".getBytes(), input.getBytes());
```

```
        }
    }
    public void punctuate(long timestamp) {
    }
    public void close() {
    }
}
```

完成代码后，启动 Application。

5.4.5 配置并启动 flume

在 flume 的 conf 目录下新建 log-kafka.properties，对 flume 连接 kafka 做配置：

```
agent.sources = exectail
agent.channels = memoryChannel
agent.sinks = kafkasink


# For each one of the sources, the type is defined
agent.sources.exectail.type = exec
# 下面这个路径是需要收集日志的绝对路径，改为自己的日志目录
agent.sources.exectail.command = tail –f
/mnt/d/Projects/BigData/MovieRecommenderSystem/businessServer/src/main/log/agent.log
agent.sources.exectail.interceptors=i1
agent.sources.exectail.interceptors.i1.type=regex_filter
# 定义日志过滤前缀的正则
agent.sources.exectail.interceptors.i1.regex=.+MOVIE_RATING_PREFIX.+
# The channel can be defined as follows.
agent.sources.exectail.channels = memoryChannel


# Each sink's type must be defined
agent.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkasink.kafka.topic = log
agent.sinks.kafkasink.kafka.bootstrap.servers = localhost:9092
agent.sinks.kafkasink.kafka.producer.acks = 1
```

```
agent.sinks.kafkasink.kafka.flumeBatchSize = 20


#Specify the channel the sink should use
agent.sinks.kafkasink.channel = memoryChannel


# Each channel's type is defined.
agent.channels.memoryChannel.type = memory


# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agent.channels.memoryChannel.capacity = 10000
```

配置好后，启动 flume：

./bin/flume-ng agent -c ./conf/ -f ./conf/log-kafka.properties -n agent -Dflume.root.logger=INFO,console

5.4.6 启动业务系统后台

将业务代码加入系统中。注意在 src/main/resources/ 下的 log4j.properties 中，log4j.appender.file.File 的值应该替换为自己的日志目录，与 flume 中的配置应该相同。

启动业务系统后台，访问 localhost:8088/index.html；点击某个电影进行评分，查看实时推荐列表是否会发生变化。

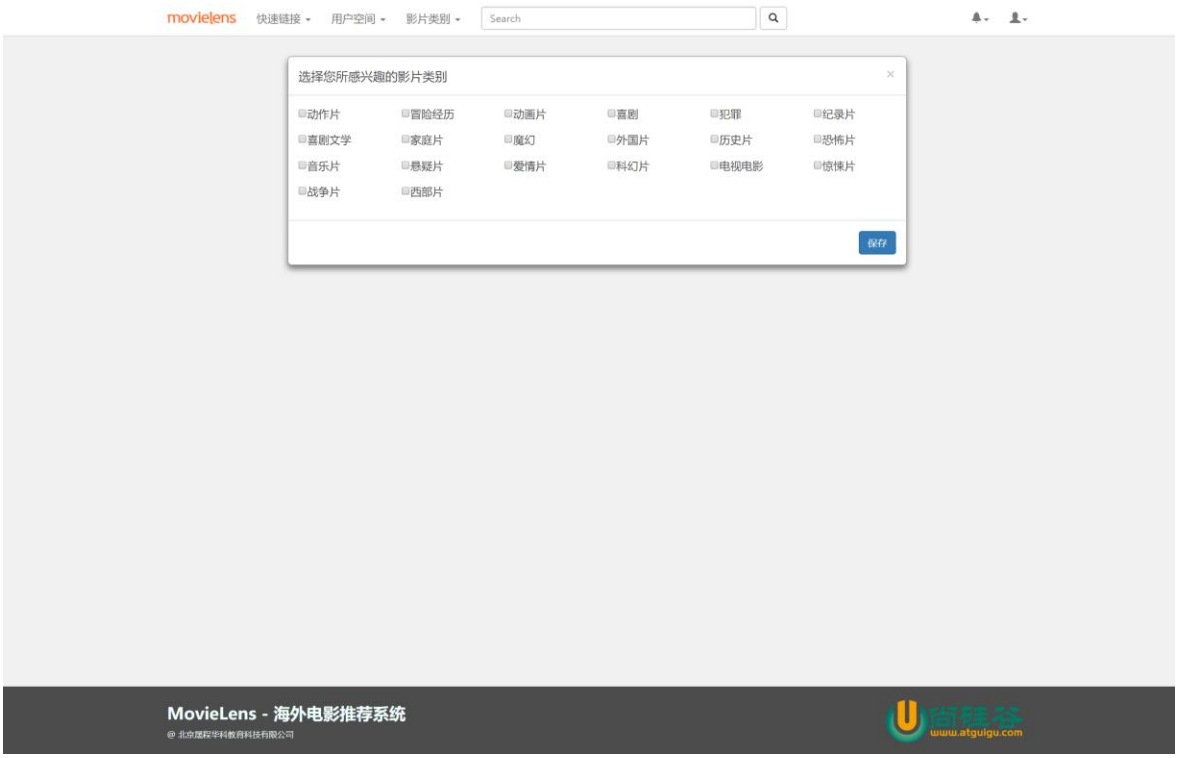
第 6 章 冷启动问题处理

整个推荐系统更多的是依赖于用于的偏好信息进行电影的推荐，那么就会存在一个问题，对于新注册的用户是没有任何偏好信息记录的，那这个时候推荐就会出现问题，导致没有任何推荐的项目出现。

处理这个问题一般是通过当用户首次登陆时，为用户提供交互式的窗口来获取用户对于物品的偏好。

在本项目中，当用户第一次登陆的时候，系统会询问用户对于影片类别的偏好。

如下：



当获取用户的偏好之后，对应于需要通过用户偏好信息获取的推荐结果，则更改为通过对影片的类型的偏好的推荐。

第 7 章 基于内容的推荐服务

7.1 基于内容的推荐服务

原始数据中的 tag 文件，是用户给电影打上的标签，这部分内容想要直接转成评分并不容易，不过我们可以将标签内容进行提取，得到电影的内容特征向量，进而可以通过求取相似度矩阵。这部分可以与实时推荐系统直接对接，计算出与用户当前评分电影的相似电影，实现基于内容的实时推荐。为了避免热门标签对特征提取的影响，我们还可以通过 TF-IDF 算法对标签的权重进行调整，从而尽可能地接近用户偏好。

7.2 基于内容推荐的实现

基于以上思想，加入 TF-IDF 算法的求取电影特征向量的核心代码如下：

```
// 载入电影数据集
val movieTagsDF = spark
  .read
```

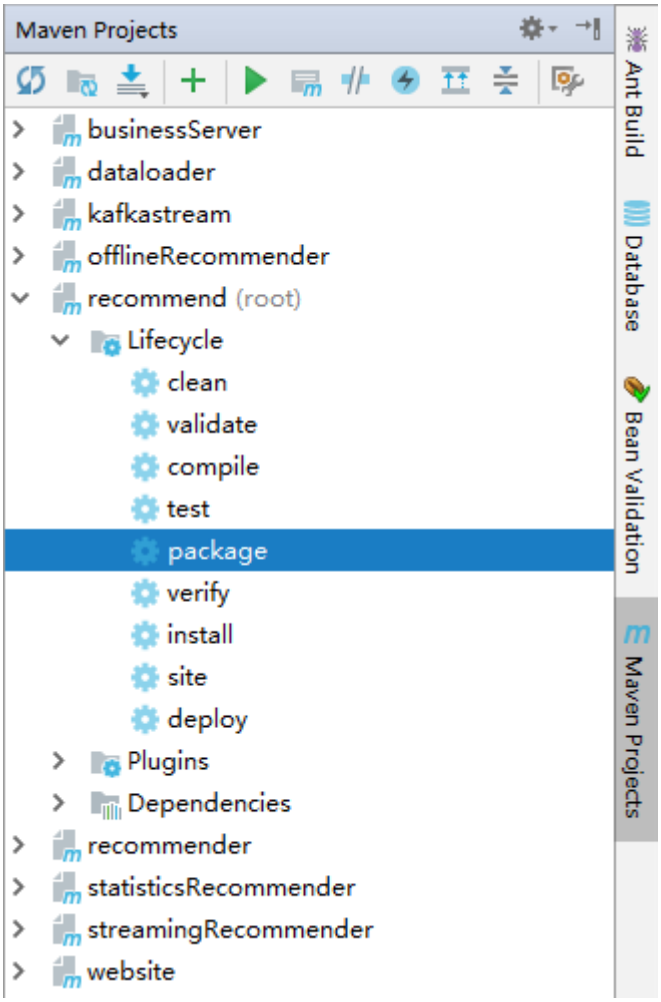
```
  .option("uri",mongoConfig.uri)
  .option("collection",MONGODB_MOVIE_COLLECTION)
  .format("com.mongodb.spark.sql")
  .load()
  .as[Movie]
  .map(x => (x.mid, x.name, x.genres.map(c => if(c == '|') ' ' else c)))
  .toDF("mid", "name", "genres").cache()

// 实例化一个分词器，默认按空格分
val tokenizer = new Tokenizer().setInputCol("genres").setOutputCol("words")

// 用分词器做转换
val wordsData = tokenizer.transform(movieTagsDF)

// 定义一个HashingTF 工具
val hashingTF = new
HashingTF().setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)

// 用 HashingTF 做处理
val featurizedData = hashingTF.transform(wordsData)

// 定义一个IDF 工具
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")

// 将词频数据传入，得到idf 模型（统计文档）
val idfModel = idf.fit(featurizedData)

// 用tf-idf 算法得到新的特征矩阵
val rescaledData = idfModel.transform(featurizedData)

// 从计算得到的 rescaledData 中提取特征向量
val movieFeatures = rescaledData.map{
  case row => ( row.getAs[Int]("mid"), row.getAs[SparseVector]("features").toArray )
}
  .rdd
  .map(x => {
    (x._1, new DoubleMatrix(x._2) )
  })
```

然后通过电影特征向量进而求出相似度矩阵，就可以为实时推荐提供基础，得到用户推荐列表了。可以看出，基于内容和基于隐语义模型，目的都是为了提取出物品的特征向量，从而可以计算出相似度矩阵。而我们的实时推荐系统算法正是基于相似度来定义的。

第 8 章 程序部署与运行

8.1 发布项目

编译项目：执行 root 项目的 clean package 阶段



编译完成如下：

```
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\Administrator\Desktop\RecommendSystem\3.code\RecommendSystem\businessServer\src\main\webapp]
[INFO] Webapp assembled in [743 msecs]
[INFO] Building war: C:\Users\Administrator\Desktop\RecommendSystem\3.code\RecommendSystem\businessServer\target\BusinessServer.war
[INFO] WEB-INF\web.xml already added, skipping
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] recommend ..................................... SUCCESS [  0.002 s]
[INFO] website ....................................... SUCCESS [ 31.780 s]
[INFO] recommender ................................... SUCCESS [  0.639 s]
[INFO] offlineRecommender ............................ SUCCESS [ 14.184 s]
[INFO] streamingRecommender .......................... SUCCESS [  8.239 s]
[INFO] statisticsRecommender ......................... SUCCESS [ 55.802 s]
[INFO] dataloader .................................... SUCCESS [01:14 min]
[INFO] kafkastream ................................... SUCCESS [  8.033 s]
[INFO] businessServer ................................ SUCCESS [  6.945 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 03:19 min
[INFO] Finished at: 2017-10-12T15:19:17+08:00
[INFO] Final Memory: 238M/1765M
[INFO] ------------------------------------------------------------------------
```

8.2 安装前端项目

将 website-release.tar.gz 解压到/var/www/html 目录下，将里面的文件放在根目录，如下：

```
[bigdata@linux html]$ pwd
/var/www/html
[bigdata@linux html]$ ll
total 23272
drwxr-xr-x     3 root root    4096 Oct 12 14:39 assets
-rw-r--r--     1 root root   38078 Oct 12 14:39 favicon.ico
-rw-r--r--     1 root root   18028 Oct 12 14:39 glyphicons-halflings-regular.448c34a56d699c29117a.woff2
-rw-r--r--     1 root root  108738 Oct 12 14:39 glyphicons-halflings-regular.89889688147bd7575d63.svg
-rw-r--r--     1 root root   45404 Oct 12 14:39 glyphicons-halflings-regular.e18bbf611f2a2e43afc0.ttf
-rw-r--r--     1 root root   20127 Oct 12 14:39 glyphicons-halflings-regular.f4769f9bdb7466be6508.eot
-rw-r--r--     1 root root   23424 Oct 12 14:39 glyphicons-halflings-regular.fa2772327f55d8198301.woff
-rw-r--r--     1 root root 1219868 Oct 12 14:39 icomoon.0a75ccae458aa5a90871.eot
-rw-r--r--     1 root root 1219780 Oct 12 14:39 icomoon.25e7839e4ae2737001b8.woff
-rw-r--r--     1 root root 4133880 Oct 12 14:39 icomoon.d50bc6cbfbd940af3c8b.svg
-rw-r--r--     1 root root 1219704 Oct 12 14:39 icomoon.d9033dd8aa810bff2e29.ttf
drwxr-xr-x 27280 root root  565248 Oct  7 11:29 images
-rw-r--r--     1 root root    1252 Oct 12 14:39 index.html
-rw-r--r--     1 root root    5898 Oct 12 14:39 inline.bundle.js
-rw-r--r--     1 root root    5964 Oct 12 14:39 inline.bundle.js.map
-rw-r--r--     1 root root  105676 Oct 12 14:39 main.bundle.js
-rw-r--r--     1 root root  105787 Oct 12 14:39 main.bundle.js.map
-rw-r--r--     1 root root  204036 Oct 12 14:39 polyfills.bundle.js
-rw-r--r--     1 root root  248279 Oct 12 14:39 polyfills.bundle.js.map
-rw-r--r--     1 root root  807669 Oct 12 14:39 scripts.bundle.js
-rw-r--r--     1 root root  987921 Oct 12 14:39 scripts.bundle.js.map
-rw-r--r--     1 root root  468093 Oct 12 14:39 styles.bundle.js
-rw-r--r--     1 root root  671809 Oct 12 14:39 styles.bundle.js.map
-rw-r--r--     1 root root 5221686 Oct 12 14:39 vendor.bundle.js
-rw-r--r--     1 root root 6331539 Oct 12 14:39 vendor.bundle.js.map
```

启动 Apache 服务器，访问 http://IP:80

8.3 安装业务服务器

将 BusinessServer.war，放到 tomcat 的 webapp 目录下，并将解压出来的文件，放到 ROOT 目录下：

```
[bigdata@linux ROOT]$ pwd
/home/bigdata/cluster/apache-tomcat-8.5.23/webapps/ROOT
[bigdata@linux ROOT]$ ll
total 12
-rw-r----- 1 bigdata bigdata  304 Oct  1 13:27 index.html
drwxr-x--- 3 bigdata bigdata 4096 Oct 12 14:25 META-INF
drwxr-x--- 4 bigdata bigdata 4096 Oct 12 14:25 WEB-INF
```

启动 Tomcat 服务器。

8.4 Kafka 配置与启动

启动 Kafka

在 kafka 中创建两个 Topic，一个为 log，一个为 recommender

启动 kafkaStream 程序，用于在 log 和 recommender 两个 topic 之间进行数据格式化。

```
[bigdata@linux ~]$ java -cp kafkastream.jar
com.atguigu.kafkastream.Application linux:9092 linux:2181 log
recommender
```

8.5 Flume 配置与启动

在 flume 安装目录下的 conf 文件夹下，创建 log-kafka.properties

```
agent.sources = exectail
```

```
agent.channels = memoryChannel
agent.sinks = kafkasink

# For each one of the sources, the type is defined
agent.sources.exectail.type = exec
agent.sources.exectail.command = tail -f
/home/bigdata/cluster/apache-tomcat-8.5.23/logs/catalina.out
agent.sources.exectail.interceptors=i1
agent.sources.exectail.interceptors.i1.type=regex_filter
agent.sources.exectail.interceptors.i1.regex=.+MOVIE_RATING_PREFIX.+
# The channel can be defined as follows.
agent.sources.exectail.channels = memoryChannel

# Each sink's type must be defined
agent.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkasink.kafka.topic = log
agent.sinks.kafkasink.kafka.bootstrap.servers = linux:9092
agent.sinks.kafkasink.kafka.producer.acks = 1
agent.sinks.kafkasink.kafka.flumeBatchSize = 20


#Specify the channel the sink should use
agent.sinks.kafkasink.channel = memoryChannel

# Each channel's type is defined.
agent.channels.memoryChannel.type = memory

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agent.channels.memoryChannel.capacity = 10000
```

启动 flume

```
[bigdata@linux apache-flume-1.7.0-kafka]$ bin/flume-ng agent -c ./conf/
-f ./conf/log-kafka.properties -n agent
```

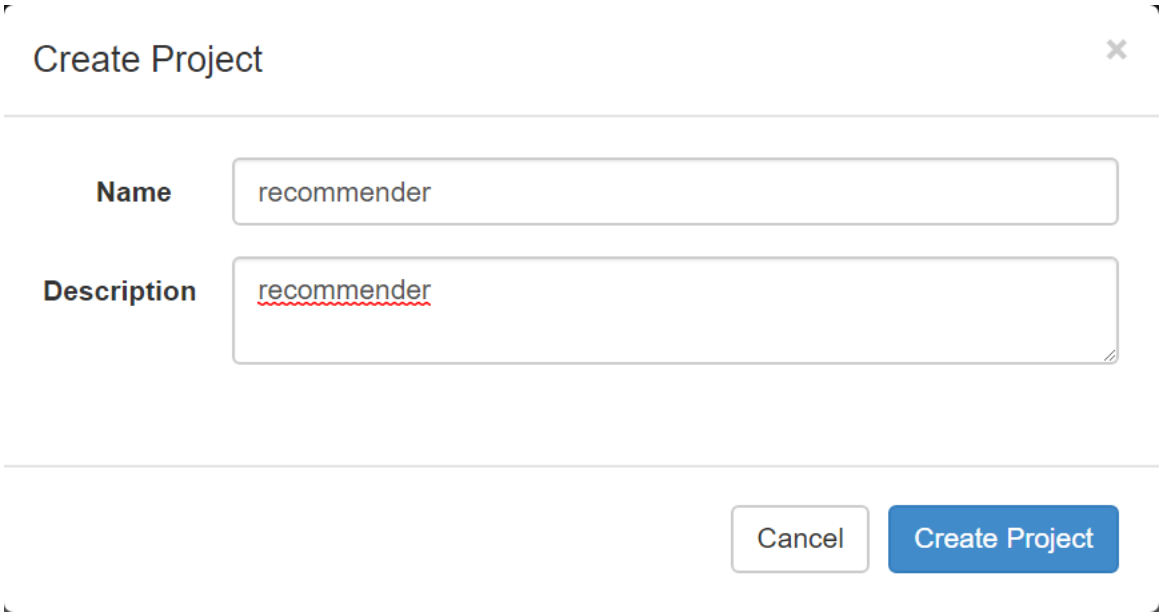
8.6 部署流式计算服务

提交 SparkStreaming 程序：

```
[bigdata@linux spark-2.1.1-bin-hadoop2.7]$ bin/spark-submit --class
com.atguigu.streamingRecommender.StreamingRecommender
streamingRecommender-1.0-SNAPSHOT.jar
```

8.7 Azkaban 调度离线算法

创建调度项目



创建两个 job 文件如下：

Azkaban-stat.job:

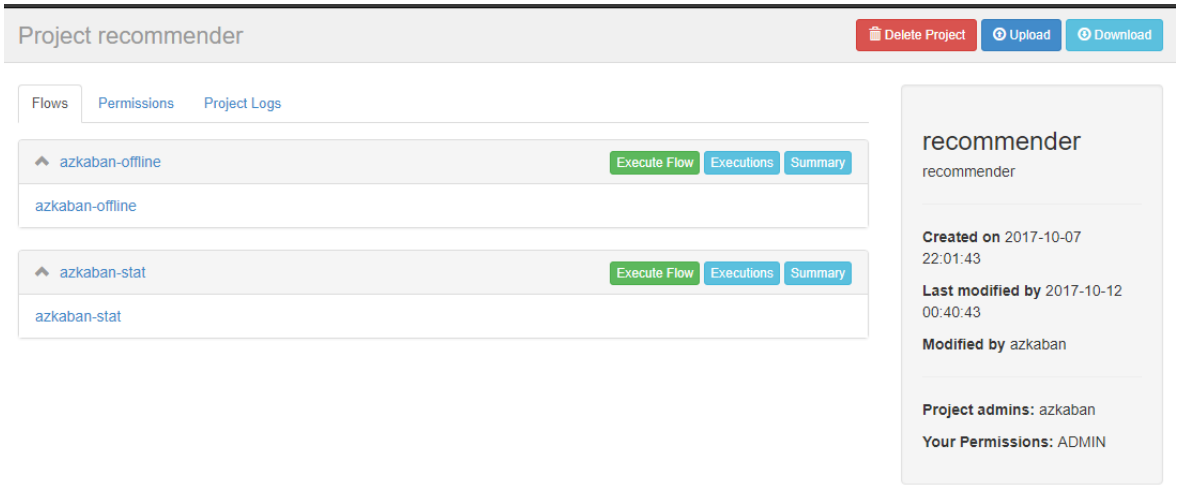
```
type=command
command=/home/bigdata/cluster/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class com.atguigu.offline.RecommenderTrainerApp
 offlineRecommender-1.0-SNAPSHOT.jar
```

Azkaban-offline.job:

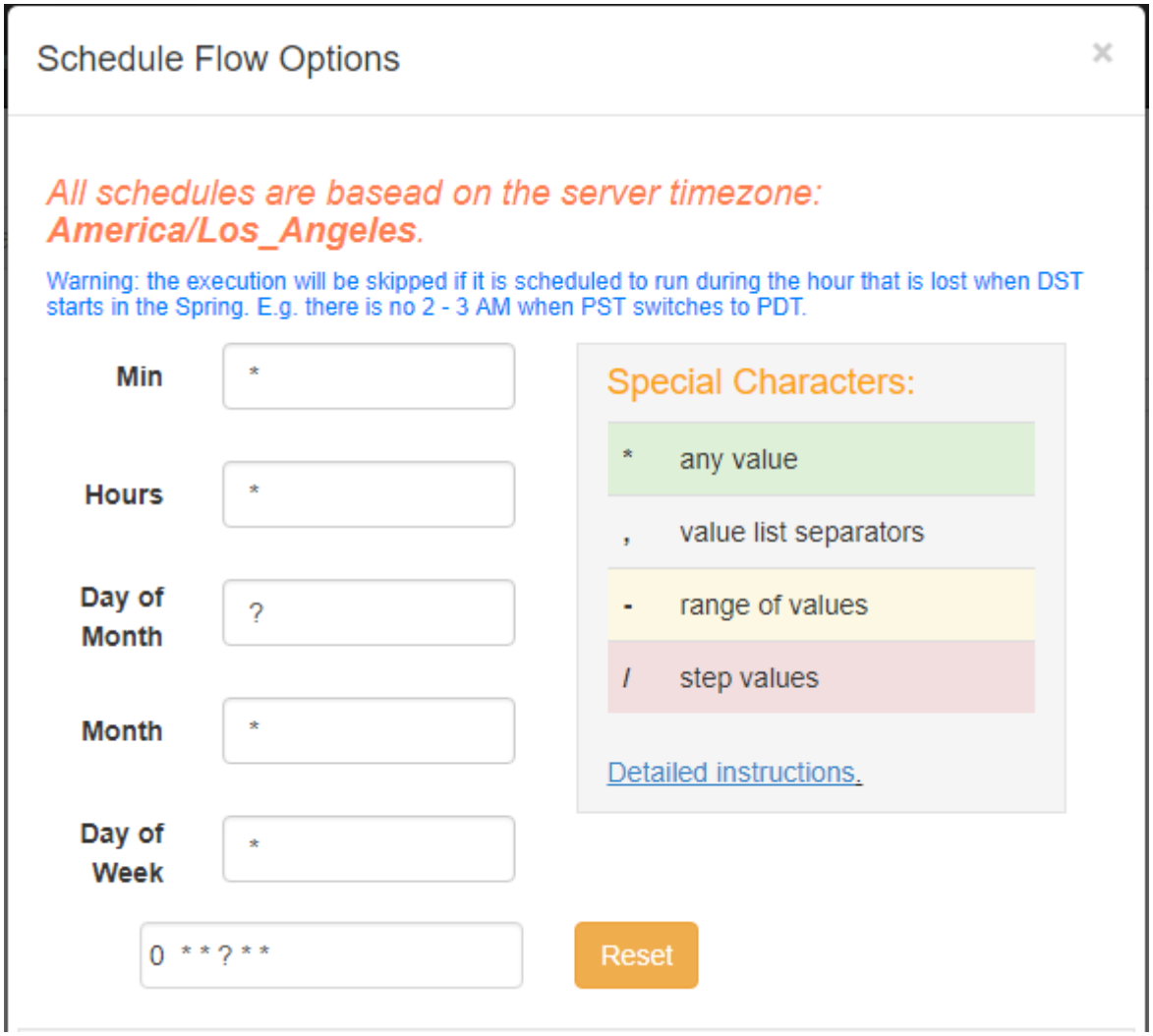
```
type=command
command=/home/bigdata/cluster/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class com.atguigu.statisticsRecommender.StatisticsApp
 statisticsRecommender-1.0-SNAPSHOT.jar
```

将 Job 文件打成 ZIP 包上传到 azkaban：

如下：



分别为每一个任务定义指定的时间，即可：



定义完成之后，点击 Scheduler 即可。