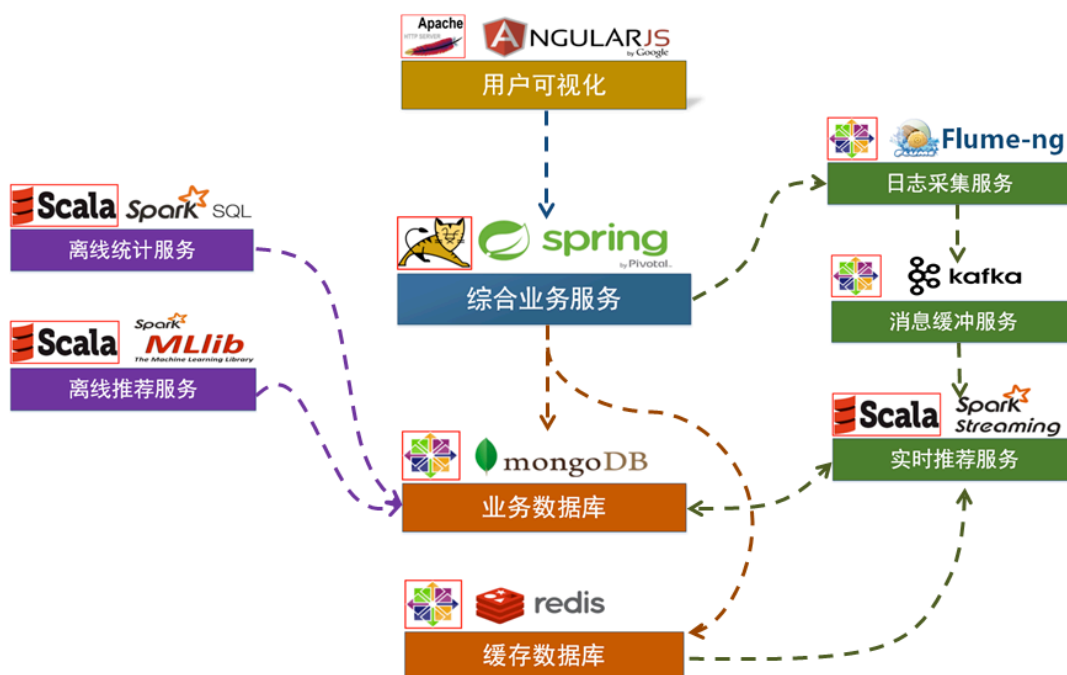


尚硅谷大数据技术之电商推荐系统

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中文亚马逊电商数据集作为依托，以某电商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商推荐系统，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。



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。

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。

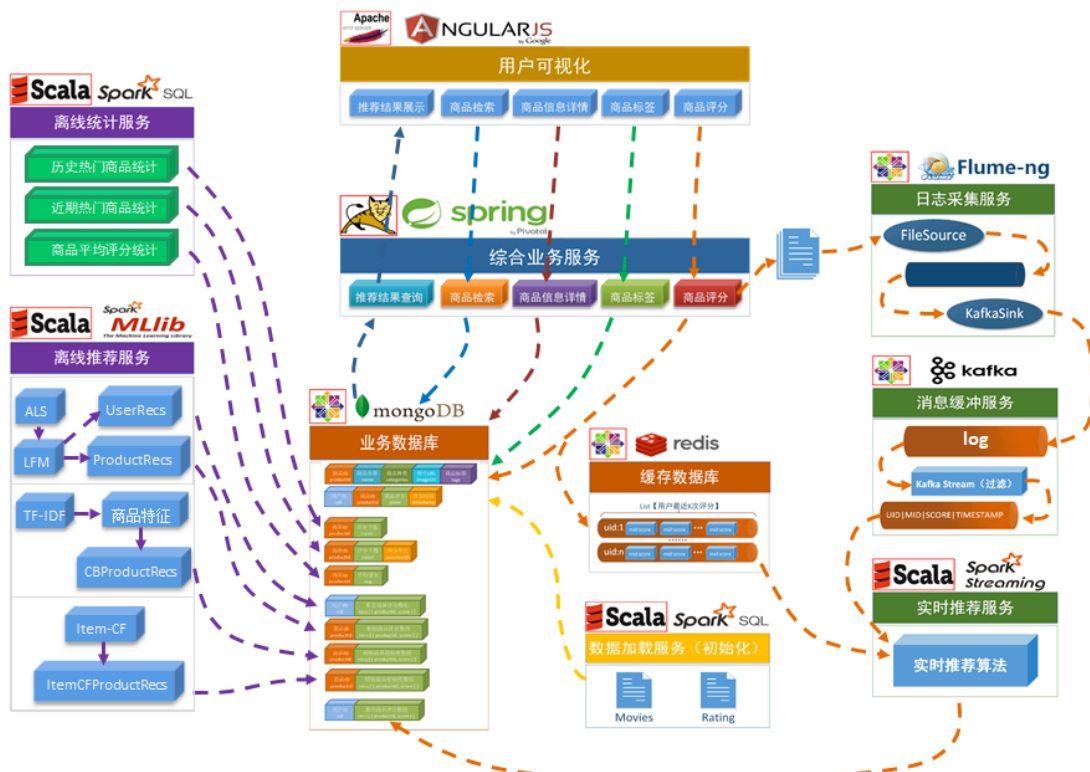
【实时推荐部分】

日志采集服务：通过利用 Flume-ng 对业务平台中用户对于商品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



【系统初始化部分】

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

【离线推荐部分】

1. 可以通过 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【商品平均评分统计】、【商品评分个数统计】、【最近商品评分个数统计】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 MongoDB 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 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. 商品信息查询服务通过对接 MongoDB 实现对商品信息的查询操作。
6. 商品评分部分，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. 商品标签部分，项目提供用户对商品打标签服务。

1.3 数据模型

1. Product【商品数据表】

字段名	字段类型	字段描述	字段备注
productId	Int	商品的 ID	
name	String	商品的名称	
categories	String	商品所属类别	每一项用“ ”分割
imageUrl	String	商品图片的 URL	
tags	String	商品的 UGC 标签	每一项用“ ”分割

2. Rating【用户评分表】

字段名	字段类型	字段描述	字段备注
userId	Int	用户的 ID	
productId	Int	商品的 ID	
score	Double	商品的分值	
timestamp	Long	评分的时间	

3. Tag【商品标签表】

字段名	字段类型	字段描述	字段备注
userId	Int	用户的 ID	
productId	Int	商品的 ID	
tag	String	商品的标签	
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
userId	Int	用户的 ID	
username	String	用户名	
password	String	用户密码	
timestamp	Long	用户创建的时间	

5. RateMoreProductsRecently【最近商品评分个数统计表】

字段名	字段类型	字段描述	字段备注
productId	Int	商品的 ID	
count	Int	商品的评分数	
yearmonth	String	评分的时段	yyyymm

6. RateMoreProducts 【商品评分个数统计表】

字段名	字段类型	字段描述	字段备注
productId	Int	商品的 ID	
count	Int	商品的评分数	

7. AverageProductsScore 【商品平均评分表】

字段名	字段类型	字段描述	字段备注
productId	Int	商品的 ID	
avg	Double	商品的平均评分	

8. ProductRecs 【商品相似性矩阵】

字段名	字段类型	字段描述	字段备注
productId	Int	商品的 ID	
recs	Array[(productId:Int,score:Double)]	该商品最相似的商品集合	

9. UserRecs 【用户商品推荐矩阵】

字段名	字段类型	字段描述	字段备注
userId	Int	用户的 ID	
recs	Array[(productId:Int,score:Double)]	推荐给该用户的商品集合	

10. StreamRecs 【用户实时商品推荐矩阵】

字段	字段类型	字段描述	字段备注
----	------	------	------

名			注
userId	Int	用户的 ID	
recs	Array[(productId:Int,score:Double)]	实时推荐给该用户的商品集合	

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 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
```

 Spark Master at spark://linux:7077

URL: spark://linux:7077
REST URL: spark://linux:6066 (cluster mode)
Alive Workers: 1
Cores in use: 4 Total, 0 Used
Memory in use: 2.9 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers				
Worker id	Address	State	Cores	Memory
worker-20171008150652-192.168.56.150-42461	192.168.56.150:42461	ALIVE	4 (0 Used)	2.9 GB (0.0 B Used)

Running Applications							
Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications							
Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

```
// 关闭 Spark 集群
```

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

2.4 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
```

2.5 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 ager
```

2.6 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

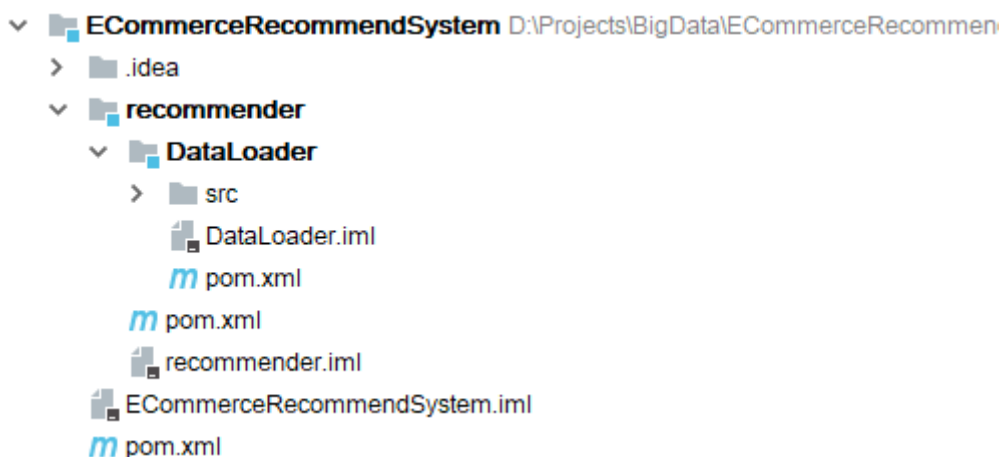
打开 IDEA，创建一个 maven 项目，命名为 ECommerceRecommendSystem。为了方便后期的联调，我们会把业务系统的代码也添加进来，所以我们可以以 ECommerceRecommendSystem 作为父项目，并在其下建一个名为 recommender 的子项目，然后再在下面搭建多个子项目用于提供不同的推荐服务。

3.1.1 项目框架搭建

在 ECommerceRecommendSystem 下新建一个 maven module 作为子项目，命名为 recommender。同样的，再以 recommender 为父项目，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 ECommerceRecommendSystem 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 ECommerceRecommendSystem 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：

ECommerceRecommendSystem/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>
  <redis.version>2.9.0</redis.version>
</properties>
```

```
<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 ECommerceRecommendSystem 中引入公有依赖：

ECommerceRecommendSystem/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>
    </plugin>
  </plugins>
</build>
```

```
<!--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）：

ECommerceRecommendSystem/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 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）：

ECommerceRecommendSystem/recommender/DataLoader/pom.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>
</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 `products.csv`, `ratings.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` 中。

3.2.1 Products 数据集

数据格式：

```
productId,name,categoryIds,amazonId,imageUrl,categories,tags
```

例如：

```
3982^FuhLen 富勒 M8 炫光舞者时尚节能无线鼠标(草绿)(炫光.悦动.时尚炫舞鼠标  
12 个月免换电池 高精度光学寻迹引擎 超细微接收器 10 米传输距  
离)^1057,439,736^B009EJN4T2^https://images-cn-4.ssl-images-amazon.co  
m/images/I/31QPvUDNavL._SY300 QL70_.jpg^外设产品|鼠标|电脑/办公^富勒|鼠  
标|电子产品|好用|外观漂亮
```

Product 数据集有 7 个字段，每个字段之间通过 “^” 符号进行分割。其中的 `categoryIds`、`amazonId` 对于内容特征没有实质帮助，我们只需要其它 5 个字段：

字段名	字段类型	字段描述	字段备注
<code>productId</code>	Int	商品 ID	
<code>name</code>	String	商品名称	
<code>categories</code>	String	商品分类	每一项用 “ ” 分割
<code>imageUrl</code>	String	商品图片 URL	
<code>tags</code>	String	商品 UGC 标签	每一项用 “ ” 分割

3.2.2 Ratings 数据集

数据格式：

```
userId,productId,rating,timestamp
```

例如：

```
4867,457976,5.0,1395676800
```

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
-----	------	------	------

userId	Int	用户 ID	
productId	Int	商品 ID	
score	Double	评分值	
timestamp	Long	评分的时间	

3.2.3 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
```

3.3 数据初始化到 MongoDB

3.3.1 启动 MongoDB 数据库（略）

3.3.2 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.recommender/DataLoader.scala

```
// 定义样例类
case class Product(productId: Int, name: String, imageUrl: String, categories: String,
                    tags: String)
case class Rating(userId: Int, productId: Int, score: Double, timestamp: Int)

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

object DataLoader {
```

```
// 以window下为例, 需替换成自己的路径, Linux下为 /YOUR_PATH/resources/products.csv
val PRODUCT_DATA_PATH = "YOUR_PATH\\resources\\products.csv"
val RATING_DATA_PATH = "YOUR_PATH\\resources\\ratings.csv"

val MONGODB_PRODUCT_COLLECTION = "Product"
val MONGODB_RATING_COLLECTION = "Rating"

// 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"
    )
    // 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Product、Rating数据集加载进来
    val productRDD = spark.sparkContext.textFile(PRODUCT_DATA_PATH)
    // 将ProductRDD转换为DataFrame
    val productDF = productRDD.map(item => {
        val attr = item.split("\\^")
        Product(attr(0).toInt, attr(1).trim, attr(4).trim, attr(5).trim, attr(6).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()

    // 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(productDF, ratingDF)
```

```
// 关闭 Spark
spark.stop()
}
```

3.3.3 将数据写入 MongoDB

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

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

    //新建一个到 MongoDB 的连接
    val mongoClient = MongoClient(MongoClientURI(mongoConfig.uri))

    // 定义通过 MongoDB 客户端拿到的表操作对象
    val productCollection = mongoClient(mongoConfig.db)(MONGODB_PRODUCT_COLLECTION)
    val ratingCollection = mongoClient(mongoConfig.db)(MONGODB_RATING_COLLECTION)

    //如果 MongoDB 中有对应的数据库，那么应该删除
    productCollection.dropCollection()
    ratingCollection.dropCollection()

    //将当前数据写入到 MongoDB
    productDF
        .write
        .option("uri", mongoConfig.uri)
        .option("collection", MONGODB_PRODUCT_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()

    //对数据表建索引
    productCollection.createIndex(MongoDBObject("productId" -> 1))
    ratingCollection.createIndex(MongoDBObject("userId" -> 1))
    ratingCollection.createIndex(MongoDBObject("productId" -> 1))

    //关闭 MongoDB 的连接
}
```

```
mongoClient.close()
}
```

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推荐、基于隐语义模型的协同过滤推荐以及基于内容和基于 Item-CF 的相似推荐。我们这一章主要介绍前两部分，基于内容和 Item-CF 的推荐在整体结构和实现上是类似的，我们将在第 7 章详细介绍。

4.2 离线统计服务

4.2.1 统计服务主体框架

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 Rating(userId: Int, productId: Int, score: Double, timestamp: Int)

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

object StatisticsRecommender {

  val MONGODB_RATING_COLLECTION = "Rating"

  //统计的表的名称
  val RATE_MORE_PRODUCTS = "RateMoreProducts"
  val RATE_MORE_RECENTLY_PRODUCTS = "RateMoreRecentlyProducts"
  val AVERAGE_PRODUCTS = "AverageProducts"

  // 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()

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

//TODO: 不同的统计推荐结果

spark.stop()
}
```

4.2.2 历史热门商品统计

根据所有历史评分数据，计算历史评分次数最多的商品。

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商品，然后按照从大到小排序，将最终结果写入 MongoDB 的 RateMoreProducts 数据集中。

```
//统计所有历史数据中每个商品的评分数
//数据结构 -> productId,count
val rateMoreProductsDF = spark.sql("select productId, count(productId) as count
from ratings group by productId ")

rateMoreProductsDF
    .write
    .option("uri",mongoConfig.uri)
    .option("collection",RATE_MORE_PRODUCTS)
```

```
.mode("overwrite")  
  
.format("com.mongodb.spark.sql")  
  
.save()
```

4.2.3 最近热门商品统计

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商品集合。

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，然后统计每月商品的评分数。统计完成之后将数据写入到 MongoDB 的 RateMoreRecentlyProducts 数据集中。

```
//统计以月为单位每个商品的评分数  
//数据结构 -》 productId,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 转换成年月格式 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 productId, score, changeDate(timestamp) as  
yearmonth from ratings")  
  
// 将新的数据集注册成为一张表  
ratingOfYearMonth.createOrReplaceTempView("ratingOfMonth")  
  
val rateMoreRecentlyProducts = spark.sql("select productId, count(productId) as  
count ,yearmonth from ratingOfMonth group by yearmonth,productId order by yearmonth desc,  
count desc")  
  
rateMoreRecentlyProducts  
  .write  
  .option("uri",mongoConfig.uri)  
  .option("collection",RATE_MORE_RECENTLY_PRODUCTS)  
  .mode("overwrite")  
  .format("com.mongodb.spark.sql")  
  .save()
```


4.2.4 商品平均得分统计

根据历史数据中所有用户对商品的评分，周期性的计算每个商品的平均得分。

实现思路：

通过 Spark SQL 读取保存在 MongoDB 中的 Rating 数据集，通过执行以下 SQL 语句实现对于商品的平均分统计：

```
//统计每个商品的平均评分
val averageProductsDF = spark.sql("select productId, avg(score) as avg from ratings group by productId ")

averageProductsDF
  .write
  .option("uri",mongoConfig.uri)
  .option("collection",AVERAGE_PRODUCTS)
  .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 AverageProducts 集合中。

4.3 基于隐语义模型的协同过滤推荐

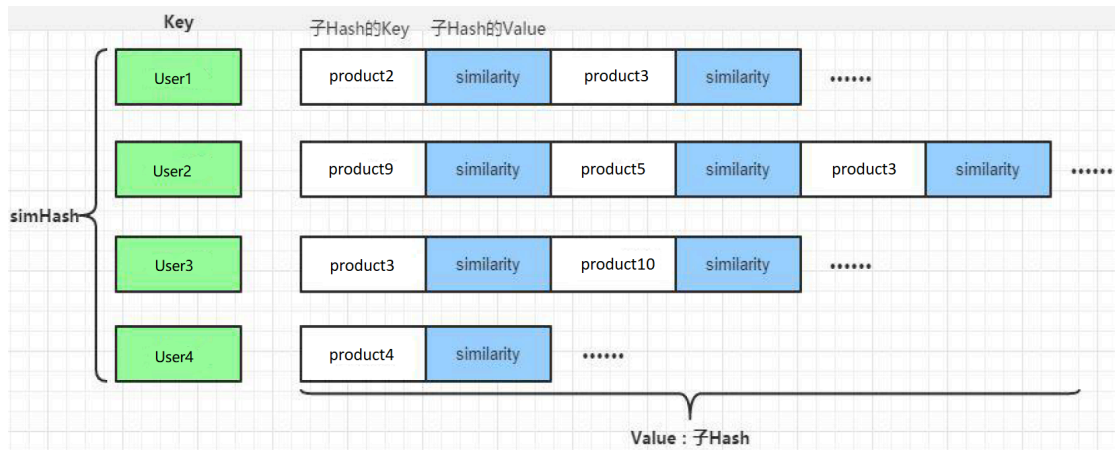
项目采用 ALS 作为协同过滤算法，根据 MongoDB 中的用户评分表计算离线的用户商品推荐列表以及商品相似度矩阵。

4.3.1 用户商品推荐列表

通过 ALS 训练出来的 Model 来计算所有当前用户商品的推荐列表，主要思路如下：

1. userId 和 productId 做笛卡尔积，产生 (userId, productId) 的元组
2. 通过模型预测 (userId, productId) 对应的评分。
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 个商品，作为当前用户的推荐列表。

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 ProductRating(userId: Int, productId: Int, score: Double, timestamp: Int)

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

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

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

// 商品相似度（商品推荐）
case class ProductRecs(productId: Int, recs: Seq[Recommendation])

object OfflineRecommender {

  // 定义常量
  val MONGODB_RATING_COLLECTION = "Rating"

  // 推荐表的名称
  val USER_RECS = "UserRecs"
  val PRODUCT_RECS = "ProductRecs"

  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("OfflineRecommender")
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[ProductRating]
    .rdd
    .map(rating=> (rating.userId, rating.productId, rating.score)).cache()
// 用户的数据集 RDD[Int]
val userRDD = ratingRDD.map(_._1).distinct()
val productRDD = ratingRDD.map(_._2).distinct()

// 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 userProducts = userRDD.cartesian(productRDD)
// model 已训练好, 把 id 传进去就可以得到预测评分列表 RDD[Rating]
(userId,productId,rating)
val preRatings = model.predict(userProducts)

val userRecs = preRatings
    .filter(_._rating > 0)
    .map(rating => (rating.user,(rating.product, rating.rating)))
    .groupByKey()
```

```
.map{
    case (userId,recs) => UserRecs(userId,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: 计算商品相似度矩阵

// 关闭 spark
spark.stop()
}
}
```

4.3.2 商品相似度矩阵

通过 ALS 计算商品相似度矩阵，该矩阵用于查询当前商品的相似商品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、商品分别生成最终的特征矩阵，分别是表示用户特征矩阵的 $U(m \times k)$ 矩阵，每个用户由 k 个特征描述；表示物品特征矩阵的 $V(n \times k)$ 矩阵，每个物品也由 k 个特征描述。

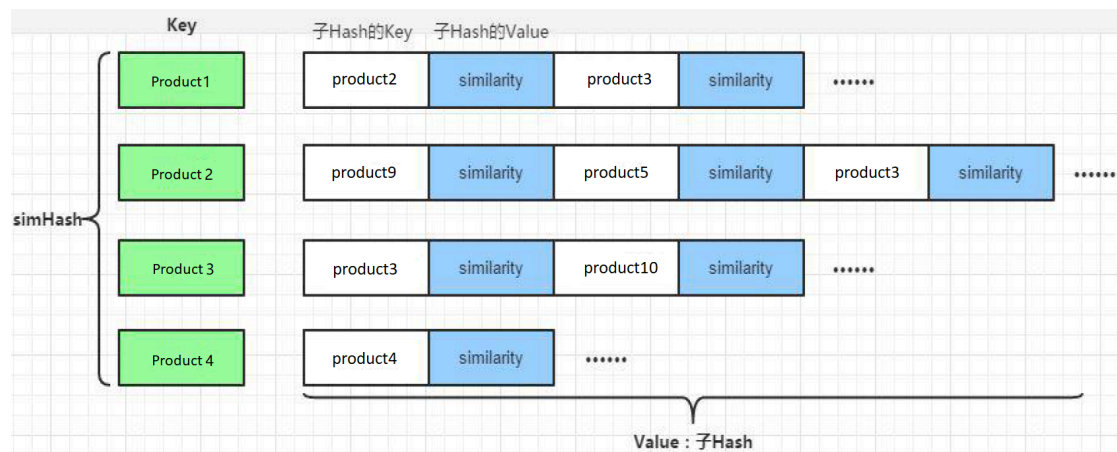
$V(n \times k)$ 表示物品特征矩阵，每一行是一个 k 维向量，虽然我们并不知道每一个维度的特征意义是什么，但是 k 个维度的数学向量表示了该行对应商品的特征。

所以，每个商品用 $V(n \times k)$ 每一行的 $\langle t_1, t_2, t_3, \dots, t_k \rangle$ 向量表示其特征，于是任意两个商品 p ：特征向量为 $V_p = \langle t_{p1}, t_{p2}, t_{p3}, \dots, t_{pk} \rangle$ ，商品 q ：特征向量为 $V_q = \langle t_{q1}, t_{q2}, t_{q3}, \dots, t_{qk} \rangle$ 之间的相似度 $\text{sim}(p, q)$ 可以使用 V_p 和 V_q 的余弦值来表示：

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

数据集中任意两个商品间相似度都可以由公式计算得到，商品与商品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 ProductRecs

表中。



核心代码如下：

```
// 计算商品相似度矩阵
// 获取商品的特征矩阵，数据格式 RDD[(scala.Int, scala.Array[scala.Double])]
val productFeatures = model.productFeatures.map{case (productId,features) =>
  (productId, new DoubleMatrix(features))
}

// 计算笛卡尔积并过滤合并
val productRecs = productFeatures.cartesian(productFeatures)
  .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 (productId,items) =>
    ProductRecs(productId,items.toList.map(x => Recommendation(x._1,x._2)))
  }.toDF()

productRecs
  .write
  .option("uri", mongoConfig.uri)
  .option("collection",PRODUCT_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：

```
// 计算两个商品之间的余弦相似度
def consinSim(product1: DoubleMatrix, product2:DoubleMatrix) : Double ={
```

```
product1.dot(product2) / ( product1.norm2() * product2.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。

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

有了 RMSE，我们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[ProductRating]  
        .rdd  
        .map(rating => Rating(rating.userId, rating.productId, 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 userProducts = data.map(item => (item.user,item.product))
    val predictRating = model.predict(userProducts)
    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 ((userId,productId),(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 在某时刻对商品 p 给予了极高的评分，那么在近期一段时候， u 极有可能很喜欢与商品 p 类似的其他商品；而如果用户 u 在某时刻对商品 q 给予了极低的评分，那么在近期一段时候， u 极有可能不喜欢与商品 q 类似的其他商品。所以对于实时推荐，当用户对一个商品进行了评价后，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 实时推荐模型和代码框架

5.2.1 实时推荐模型算法设计

当用户 u 对商品 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 对商品 p 评分，对于用户 u 来说，他与 p 最相似的商品们之间的推荐强度将

发生变化，所以选取与商品 p 最相似的 K 个商品作为候选商品。

每个候选商品按照“推荐优先级”这一权重作为衡量这个商品被推荐给用户 u 的优先级。

这些商品将根据用户 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 ；获取商品 p 的最相似的 K 个商品集合，记为 S ；

然后，对于每个商品 $q \in S$ ，计算其推荐优先级 E_{uq} ，计算公式如下：

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

其中：

R_r 表示用户 u 对商品 r 的评分；

$\text{sim}(q, r)$ 表示商品 q 与商品 r 的相似度，设定最小相似度为 0.6，当商品 q 和商品 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 中商品相似度大于最小阈值的个数；

incount 表示 RK 中与商品 q 相似的、且本身评分较高 (≥ 3) 的商品个数；

recount 表示 RK 中与商品 q 相似的、且本身评分较低 (< 3) 的商品个数；

公式的意义如下：

首先对于每个候选商品 q ，从 u 最近的 K 个评分中，找出与 q 相似度较高 (≥ 0.6) 的 u 已评分商品们，对于这些商品们中的每个商品 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 对商品 q 的评分预测即

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

然后，将 u 最近的 K 个评分中与商品 q 相似的、且本身评分较高 (≥ 3) 的商品个数记为 incount ，计算 $\lg \max\{\text{incount}, 1\}$ 作为商品 q 的“增强因子”，意义

在于商品 q 与 u 的最近 K 个评分中的 n 个高评分(≥ 3)商品相似, 则商品 q 的优先级被增加 $\lg\max\{\text{incount}, 1\}$ 。如果商品 q 与 u 的最近 K 个评分中相似的高评分商品越多, 也就是说 n 越大, 则商品 q 更应该被推荐, 所以推荐优先级被增强的幅度较大; 如果商品 q 与 u 的最近 K 个评分中相似的高评分商品越少, 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 个评分中与商品 q 相似的、且本身评分较低(< 3)的商品个数记为 recount , 计算 $\lg\max\{\text{recount}, 1\}$ 作为商品 q 的“削弱因子”, 意义在于商品 q 与 u 的最近 K 个评分中的 n 个低评分(< 3)商品相似, 则商品 q 的优先级被削减 $\lg\max\{\text{incount}, 1\}$ 。如果商品 q 与 u 的最近 K 个评分中相似的低评分商品越多, 也就是说 n 越大, 则商品 q 更不应该被推荐, 所以推荐优先级被减弱的幅度较大; 如果商品 q 与 u 的最近 K 个评分中相似的低评分商品越少, 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 商品对于 u 的推荐优先级。在计算完每个候选商品 q 的 E_{uq} 后, 将生成一组 \langle 商品 q 的 ID, q 的推荐优先级 \rangle 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 \langle 商品 m, m 的推荐优先级 \rangle 的列表, 其大小也为 K :

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

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 :

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

其中, i 表示 updated_S 与 Rec 的商品集合中的每个商品, topK 是一个函数, 表示从 $\text{Rec} \cup \text{updated_S}$ 中选择出最大的 K 个商品, $\text{cmp} = E_{ui}$ 表示 topK 函数将推荐优先级 E_{ui} 值最大的 K 个商品选出来。最终, NewRec 即为经过用户 u 对商品 p 评分后触发的实时推荐得到的最新推荐结果。

总之, 实时推荐算法流程基本如下:

- (1) 用户 u 对商品 p 进行了评分, 触发了实时推荐的一次计算;
- (2) 选出商品 p 最相似的 K 个商品作为集合 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) 计算商品的推荐优先级，产生 $\langle qID, \rangle$ 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。

5.2.2 实时推荐模块框架

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>
</dependencies>
```

```
<!-- 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(productId:Int, score:Double)

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

// 商品的相似度
case class ProductRecs(productId:Int, recs:Seq[Recommendation])
```

```
object StreamingRecommender {  
  
    val MAX_USER_RATINGS_NUM = 20  
    val MAX_SIM_PRODUCTS_NUM = 20  
    val MONGODB_STREAM_RECS_COLLECTION = "StreamRecs"  
    val MONGODB_RATING_COLLECTION = "Rating"  
    val MONGODB_PRODUCT_RECS_COLLECTION = "ProductRecs"  
  
    //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._  
  
    // 广播商品相似度矩阵  
    // 装换成为 Map[Int, Map[Int,Double]]  
    val simProductsMatrix = spark  
        .read  
        .option("uri",config("mongo.uri"))  
        .option("collection",MONGODB_PRODUCT_RECS_COLLECTION)  
        .format("com.mongodb.spark.sql")  
        .load()  
        .as[ProductRecs]  
        .rdd  
        .map{recs =>  
            (recs.productId,recs.recs.map(x=> (x.productId,x.score)).toMap)  
        }.collectAsMap()
```

[illegible]

```
saveRecsToMongoDB(userId, 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商品的 K 次评分。实时算法可以快速获取。
2. 离线推荐算法已经将商品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, productId, rate, timestamp>，而执行的核心内容包括：获取 userId 最近 K 次评分、获取 productId 最相似 K 个商品、计算候选商品的推荐优先级、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, productId, 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 次商品评分
 * @param num 评分的个数
 * @param userId 谁的评分
 * @return
 */
def getUserRecentlyRating(num: Int, userId: Int, jedis: Jedis): Array[(Int, Double)] = {
    //从用户的队列中取出 num 个评分
    jedis.lrange("userId:" + userId.toString, 0, num).map{item =>
        val attr = item.split("\\:")
        (attr(0).trim.toInt, attr(1).trim.toDouble)
    }
}
```



```
} .toArray  
}
```

5.3.2 获取当前商品最相似的 K 个商品

在离线算法中，已经预先将商品的相似度矩阵进行了计算，所以每个商品 productId 的最相似的 K 个商品很容易获取：从 MongoDB 中读取 ProductRecs 数据，从 productId 在 simHash 对应的子哈希表中获取相似度前 K 大的那些商品。输出是数据类型为 Array[Int] 的数组，表示与 productId 最相似的商品集合，并命名为 candidateProducts 以作为候选商品集合。

```
/**  
 * 获取当前商品 K 个相似的商品  
 * @param num          相似商品的数量  
 * @param productId    当前商品的 ID  
 * @param userId       当前的评分用户  
 * @param simProducts  商品相似度矩阵的广播变量值  
 * @param mongConfig   MongoDB 的配置  
 * @return  
 */  
  
def getTopSimProducts(num:Int, productId:Int, userId:Int,  
simProducts:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Double]])(i  
mplicit mongConfig: MongConfig): Array[Int] = {  
  // 从广播变量的商品相似度矩阵中获取当前商品所有的相似商品  
  val allSimProducts = simProducts.get(productId).get.toArray  
  // 获取用户已经观看过得商品  
  val ratingExist =  
ConnHelper.mongoClient(mongConfig.db)(MONGODB_RATING_COLLECTION).find(MongoDBObject  
("userId" -> userId)).toArray.map{item =>  
    item.get("productId").toString.toInt  
  }  
  // 过滤掉已经评分过得商品，并排序输出  
  allSimProducts.filter(x => !ratingExist.contains(x._1)).sortWith(_._2 >  
_._2).take(num).map(x => x._1)  
}
```

5.3.3 商品推荐优先级计算

对于候选商品集合 simiHash 和 userId 的最近 K 个评分 recentRatings，算法代码如下：

```
/**  
 * 计算待选商品的推荐分数  
 * @param simProducts  商品相似度矩阵  
 * @param userRecentlyRatings 用户最近的 k 次评分
```

```
* @param topSimProducts      当前商品最相似的K个商品
* @return
*/
def computeProductScores(
    simProducts:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Double]],userRecentlyRatings:Array[(Int,Double)],topSimProducts: Array[Int]):
    Array[(Int,Double)] ={

    //用于保存每一个待选商品和最近评分的每一个商品的权重得分
    val score = scala.collection.mutable.ArrayBuffer[(Int,Double)]()

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

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

    for (topSimProduct <- topSimProducts; userRecentlyRating <- userRecentlyRatings){
        val simScore =
        getProductSimScore(simProducts,userRecentlyRating._1,topSimProduct)
        if(simScore > 0.6){
            score += ((topSimProduct, simScore * userRecentlyRating._2 ))
            if(userRecentlyRating._2 > 3){
                increMap(topSimProduct) = increMap.getOrElse(topSimProduct,0) + 1
            }else{
                decreMap(topSimProduct) = decreMap.getOrElse(topSimProduct,0) + 1
            }
        }
    }

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

}
```

其中，getProductSimScore 是取候选商品和已评分商品的相似度，代码如下：

```
/**
 * 获取单个商品之间的相似度
 * @param simProducts      商品相似度矩阵
 * @param userRatingProduct 用户已经评分的商品
 * @param topSimProduct    候选商品
```

```
* @return
*/
def getProductsSimScore(
simProducts:scala.collection.Map[Int,scala.collection.immutable.Map[Int,Double]],
userRatingProduct:Int, topSimProduct:Int): Double ={
    simProducts.get(topSimProduct) match {
        case Some(sim) => sim.get(userRatingProduct) 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    userId -> 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(userId:Int,streamRecs:Array[(Int,Double)])(implicit mongConfig:
MongConfig): Unit ={
    //到StreamRecs 的连接
    val streamRecsCollection =
ConnHelper.mongoClient(mongConfig.db)(MONGODB_STREAM_RECS_COLLECTION)

    streamRecsCollection.findAndRemove(MongoDBObject("userId" -> userId))
    streamRecsCollection.insert(MongoDBObject("userId" -> userId, "recs" ->
        streamRecs.map( x => MongoDBObject("productId"->x._1,"score"->x._2)) ))
}
```

5.3.5 更新实时推荐结果

当计算出候选商品的推荐优先级的数组 updatedRecommends<productId, 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<productId, E>进行合并、替换并选出优先级 E 前 K 大的商品

作为本次新的实时推荐。具体而言：

a. 合并：将 `updatedRecommends` 与 `recentRecommends` 并集成为一个新的 `<productId, E>` 数组；

b. 替换（去重）：当 `updatedRecommends` 与 `recentRecommends` 有重复的商品 `productId` 时，`recentRecommends` 中 `productId` 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 `productId` 的推荐优先级；

c. 选取 TopK：在合并、替换后的 `<productId, E>` 数组上，根据每个 `product` 的推荐优先级，选择出前 K 大的商品，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>
```

```
</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("PRODUCT_RATING_PREFIX:")){
            System.out.println("product rating coming!!!!" + input);
            input = input.split("PRODUCT_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/ECommerceRecommenderSystem/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=.+PRODUCT_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`；点击某个商品进行评分，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户的偏好信息进行商品的推荐，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題，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，让用户勾选预设的兴趣标签。

当获取用户的偏好之后，就可以直接给出相应类型商品的推荐。

第 7 章 其它形式的离线相似推荐服务

7.1 基于内容的相似推荐

原始数据中的 tag 文件，是用户给商品打上的标签，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，得到商品的内容特征向量，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商品的相似商品，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。

基于以上思想，加入 TF-IDF 算法的求取商品特征向量的核心代码如下：

```
// 载入商品数据集
val productTagsDF = spark
    .read
    .option("uri",mongoConfig.uri)
    .option("collection",MONGODB_PRODUCT_COLLECTION)
    .format("com.mongodb.spark.sql")
    .load()
    .as[Product]
    .map(x => (x.productId, x.name, x.genres.map(c => if(c == '|') ' ' else c)))
    .toDF("productId", "name", "tags").cache()

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

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

// 定义一个HashingTF 工具
val hashingTF = new
HashingTF().setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(200)

// 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 productFeatures = rescaledData.map{
  case row =>
    ( row.getAs[Int]("productId"), row.getAs[SparseVector]("features").toArray )
}
  .rdd
  .map(x => {
    (x._1, new DoubleMatrix(x._2) )
  })
```

然后通过商品特征向量进而求出相似度矩阵，就可以在商品详情页给出相似推荐了；通常在电商网站中，用户浏览商品或者购买完成之后，都会显示类似的推荐列表。

得到的相似度矩阵也可以为实时推荐提供基础，得到用户推荐列表。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。

7.2 基于物品的协同过滤相似推荐

基于物品的协同过滤（Item-CF），只需收集用户的常规行为数据（比如点击、收藏、购买）就可以得到商品间的相似度，在实际项目中应用很广。

我们的整体思想是，如果两个商品有同样的受众（感兴趣的人群），那么它们就是有内在相关性的。所以可以利用已有的行为数据，分析商品受众的相似程度，进而得出商品间的相似度。我们把这种方法定义为物品的“同现相似度”，公式如下：

$$w_{ij} = \frac{|N_i \cap N_j|}{\sqrt{|N_i| |N_j|}}$$

其中， N_i 是购买商品 i （或对商品 i 评分）的用户列表， N_j 是购买商品 j 的用户列表。

核心代码实现如下：

```
val ratingDF = spark.read
  .option("uri", mongoConfig.uri)
  .option("collection", MONGODB_RATING_COLLECTION)
  .format("com.mongodb.spark.sql")
  .load()
```

```
.as[Rating]
.map(x=> (x.userId, x.productId, x.score) )
.toDF("userId", "productId", "rating")

// 统计每个商品的评分个数，并通过内连接添加到 ratingDF 中
val numRatersPerProduct = ratingDF.groupBy("productId").count()
val ratingWithCountDF = ratingDF.join(numRatersPerProduct, "productId")

// 将商品评分按 userId 两两配对，可以统计两个商品被同一用户做出评分的次数
val joinedDF = ratingWithCountDF.join(ratingWithCountDF, "userId")
.toDF("userId", "product1", "rating1", "count1", "product2", "rating2", "count2")
.select("userId", "product1", "count1", "product2", "count2")
joinedDF.createOrReplaceTempView("joined")
val cooccurrenceDF = spark.sql(
  """
  |select product1
  |, product2
  |, count(userId) as coocount
  |, first(count1) as count1
  |, first(count2) as count2
  |from joined
  |group by product1, product2
  """.stripMargin
).cache()

val simDF = cooccurrenceDF.map{ row =>
  // 用同现的次数和各自的次数，计算同现相似度
  val coocSim = cooccurrenceSim( row.getAs[Long]("coocount"),
row.getAs[Long]("count1"), row.getAs[Long]("count2") )
  ( row.getAs[Int]("product1"), ( row.getAs[Int]("product2"), coocSim ) )
}
.rdd
.groupByKey()
.map{
  case (productId, recs) =>
    ProductRecs( productId,
      recs.toList
        .filter(x=>x._1 != productId)
        .sortWith(_._2>_. _2)
        .map(x=>Recommendation(x._1,x._2))
        .take(MAX_RECOMMENDATION)
    )
}
.toDF()
```

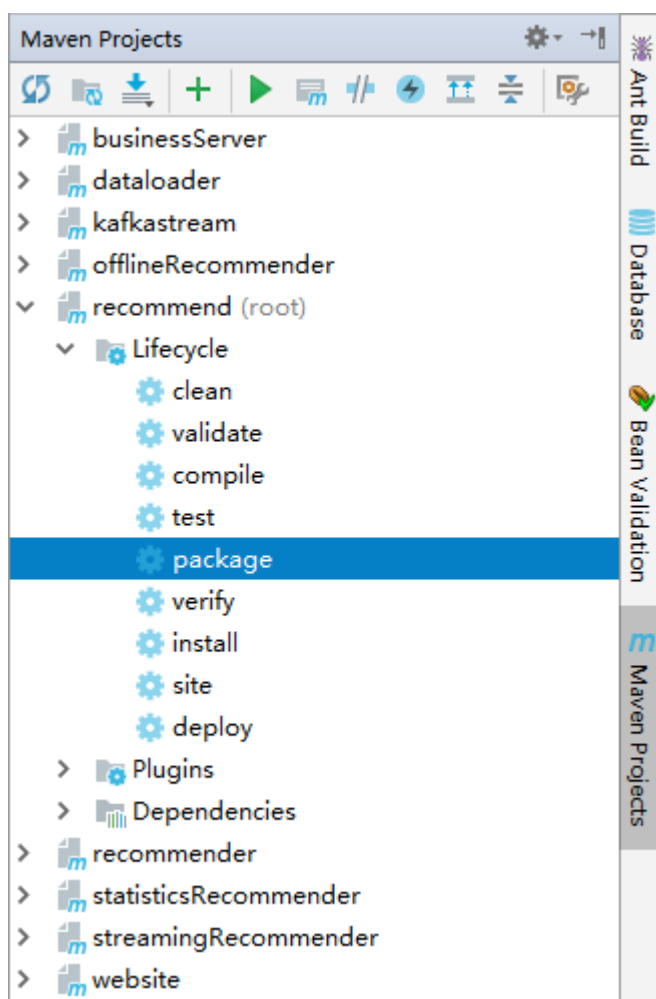
其中，计算同现相似度的函数代码实现如下：

```
def cooccurrenceSim(cooCount: Long, count1: Long, count2: Long): Double = {  
    cooCount / math.sqrt( count1 * count2 )  
}
```

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 glyphs-halflings-regular.448c34a56d699c29117a.woff2
-rw-r--r-- 1 root root 108738 Oct 12 14:39 glyphs-halflings-regular.89889688147bd7575d63.svg
-rw-r--r-- 1 root root 45404 Oct 12 14:39 glyphs-halflings-regular.e18bbf611f2a2e43afc0.ttf
-rw-r--r-- 1 root root 20127 Oct 12 14:39 glyphs-halflings-regular.f4769f9bdb7466be6508.eot
-rw-r--r-- 1 root root 23424 Oct 12 14:39 glyphs-halflings-regular.f2772327f55d8198301.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.d50bc6cbfd940af3c8b.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=.+PRODUCT_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

Create Project

Name

recommender

Description

recommender

Cancel

Create Project

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-su  
bmit --class com.atguigu.offline.RecommenderTrainerApp  
offlineRecommender-1.0-SNAPSHOT.jar
```

Azkaban-offline.job:

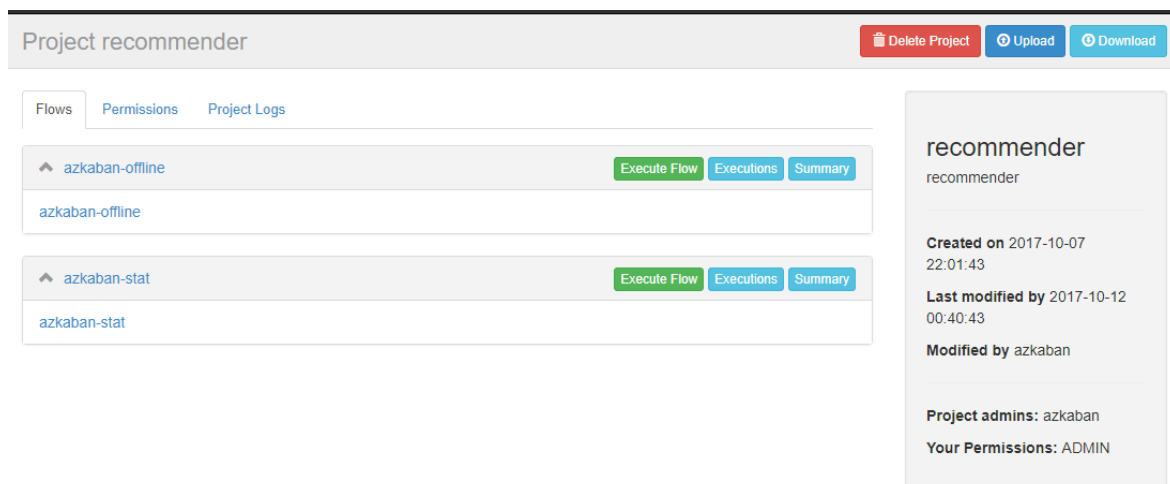
```
type=command  
command=/home/bigdata/cluster/spark-2.1.1-bin-hadoop2.7/bin/spark-su  
bmit --class com.atguigu.statisticsRecommender.StatisticsApp  
statisticsRecommender-1.0-SNAPSHOT.jar
```

将 Job 文件打成 ZIP 包上传到 azkaban:



The image shows a web-based dialog box titled "Upload Project Files" with a close button (X) in the top right corner. Below the title bar, there is a section labeled "Job Archive". To the right of this label is a file selection interface containing a button labeled "选择文件" (Select File) and the text "未选择任何文件" (No file selected). At the bottom right of the dialog, there are two buttons: "Cancel" and "Upload".

如下:



The image displays the "Project recommender" interface in Azkaban. At the top, there is a header bar with the title "Project recommender" and three action buttons: "Delete Project" (red), "Upload" (blue), and "Download" (blue). Below the header, there are three tabs: "Flows", "Permissions", and "Project Logs". The "Flows" tab is active, showing a list of project flows. The list contains two entries: "azkaban-offline" and "azkaban-stat". Each entry has a small upward arrow icon, a name, and three buttons: "Execute Flow" (green), "Executions" (blue), and "Summary" (blue). To the right of the flow list, there is a detailed view for the selected project, "recommender". This view includes the project name "recommender", its creation date and time "Created on 2017-10-07 22:01:43", the last modification date and time "Last modified by 2017-10-12 00:40:43", the modifier "Modified by azkaban", the project administrators "Project admins: azkaban", and the user's permissions "Your Permissions: ADMIN".

分别为每一个任务定义指定的时间，即可：

Schedule Flow Options

*All schedules are basead on the server timezone:
America/Los_Angeles.*

Warning: the execution will be skipped if it is scheduled to run during the hour that is lost when DST starts in the Spring. E.g. there is no 2 - 3 AM when PST switches to PDT.

Min

*

Hours

*

Day of Month

?

Month

*

Day of Week

*

0 * * ? * *

Special Characters:

*

any value

,

value list separators

-

range of values

/

step values

[Detailed instructions.](#)

Reset

定义完成之后，点击 Scheduler 即可。