天善智能
TIANSHAN SOFT

标签数据开发

讲师：watermelon

目录

数据仓库基础知识

数据仓库

数据仓库是指一个面向主题的、集成的、稳定的、随时间变化的数据的集合，以用于支持管理决策的过程

（1）面向主题

业务数据库中的数据主要针对事物处理任务，各个业务系统之间是各自分离的。而数据仓库中的数据是按照一定的主题进行组织的

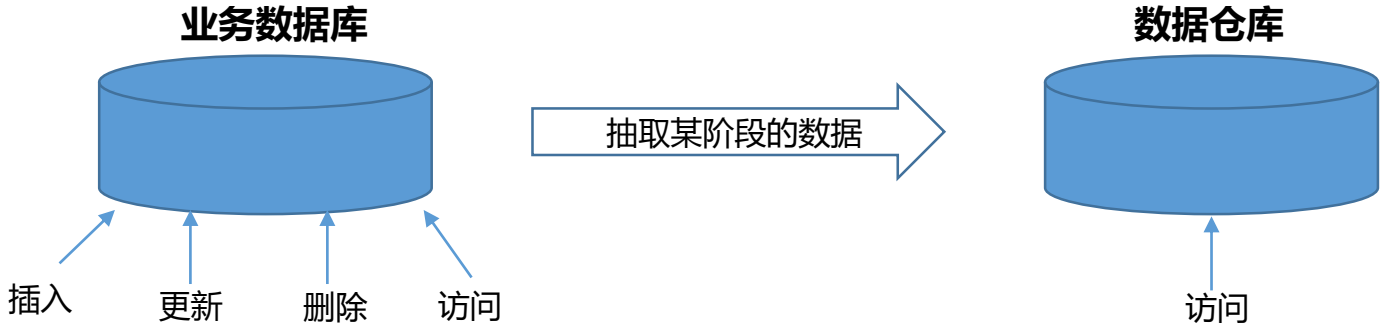
（2）集成

数据仓库中存储的数据是从业务数据库中提取出来的，但并不是原有数据的简单复制，而是经过了抽取、清理、转换（ETL）等工作。业务数据库记录的是每一项业务处理的流水账，这些数据不适合于分析处理，进入数据仓库之前需要经过系列计算，同时抛弃一些分析处理不需要的数据。

（3）稳定

操作型数据库系统中一般只存储短期数据，因此其数据是不稳定的，记录的是系统中数据变化的瞬态。

数据仓库中的数据大多表示过去某一时刻的数据，主要用于查询、分析，不像业务系统中数据库一样经常修改。一般数据仓库构建完成，主要用于访问



OLTP和OLAP

OLTP　　联机事务处理

OLTP是传统关系型数据库的主要应用，主要用于日常事物、交易系统的处理

1、数据量存储相对来说不大

2、实时性要求高，需要支持事物

3、数据一般存储在关系型数据库(oracle或mysql中)

OLAP　　联机分析处理

OLAP是数据仓库的主要应用，支持复杂的分析查询，侧重决策支持

1、实时性要求不是很高，ETL一般都是T+1的数据；

2、数据量很大；

3、主要用于分析决策；

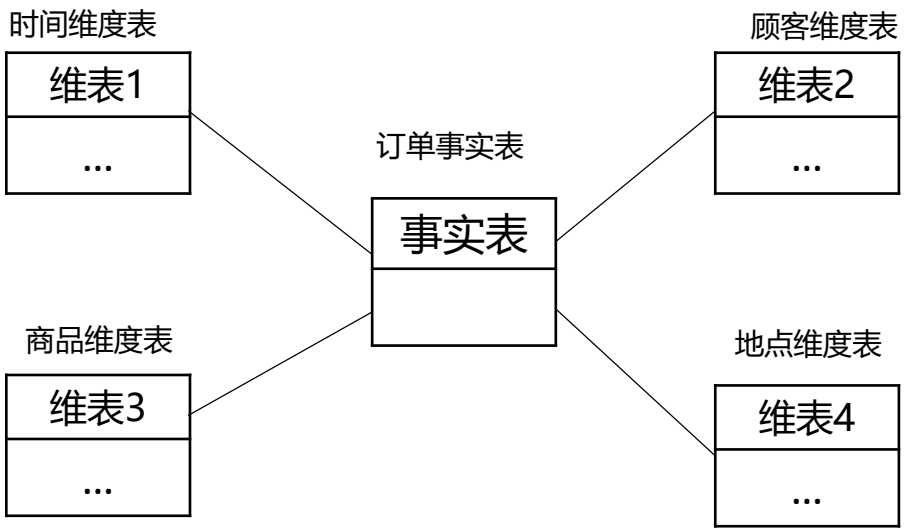
常见多维数据模型—星形模型

星形模型是最常用的数据仓库设计结构。***由一个事实表和一组维表组成***，每个维表都有一个维主键。

该模式核心是事实表，通过事实表将各种不同的维表连接起来，各个维表中的对象通过事实表与另一个维表中的对象相关联，这样建立各个维表对象之间的联系

维表：用于存放维度信息，包括维的属性和层次结构；

事实表：是用来记录业务事实并做相应指标统计的表。同维表相比，事实表记录数量很多

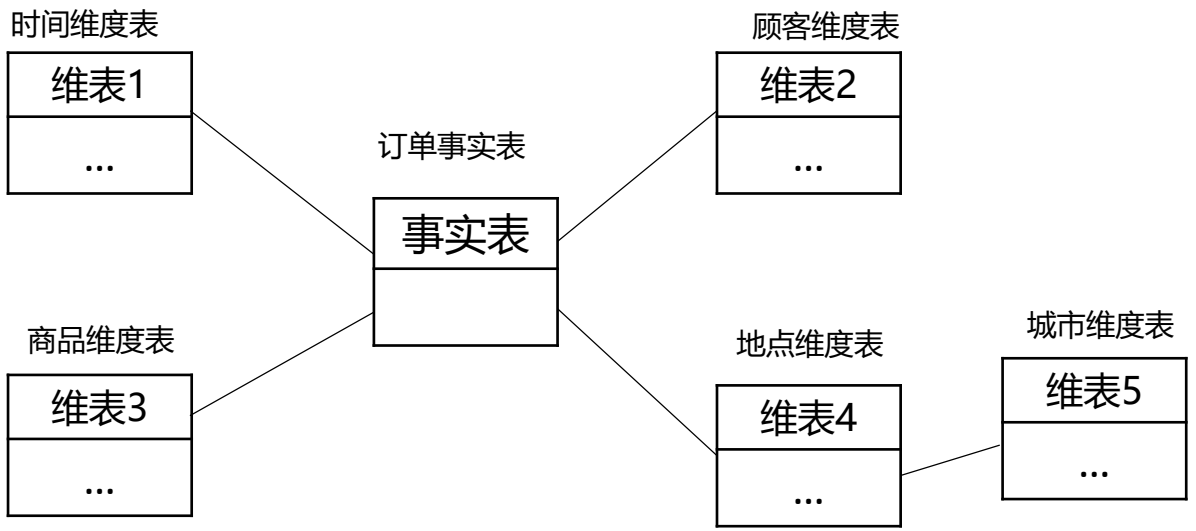


常见多维数据模型—雪花模型

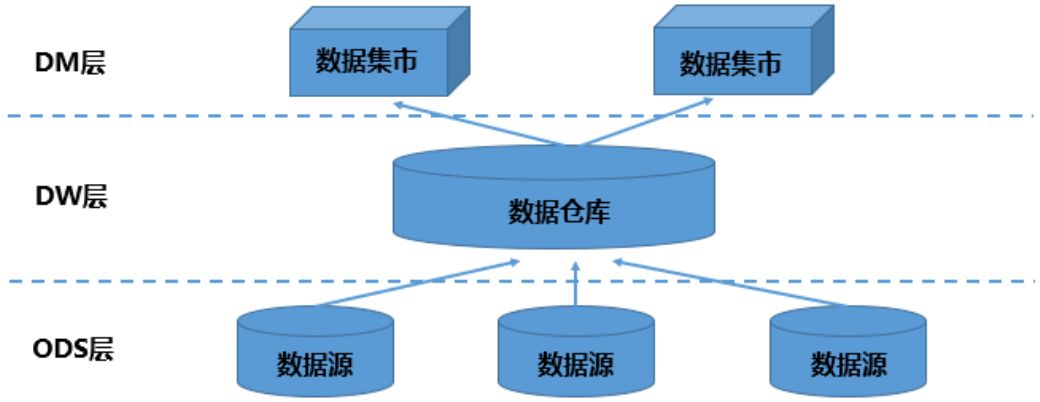
雪花模型是对星形模型的扩展，每一个维表都可以向外连接多个详细类别表。除了具有星形模式中维表的功能外，还连接对事实表进行详细描述的维度，可进一步细化查看数据的粒度

例如：地点维表包含属性集{location_id，街道，城市，省，国家}。这种模式通过地点维度表的city_id与城市维度表的city_id相关联，得到如{101，“解放大道10号”，“武汉”，“湖北省”，“中国”}、{255，“解放大道85号”，“武汉”，“湖北省”，“中国”}这样的记录。

星形模型是最基本的模式，一个星形模型有多个维表，只存在一个事实表。在星形模式的基础上，用多个表来描述一个复杂维，构造维表的多层结构，就得到雪花模型

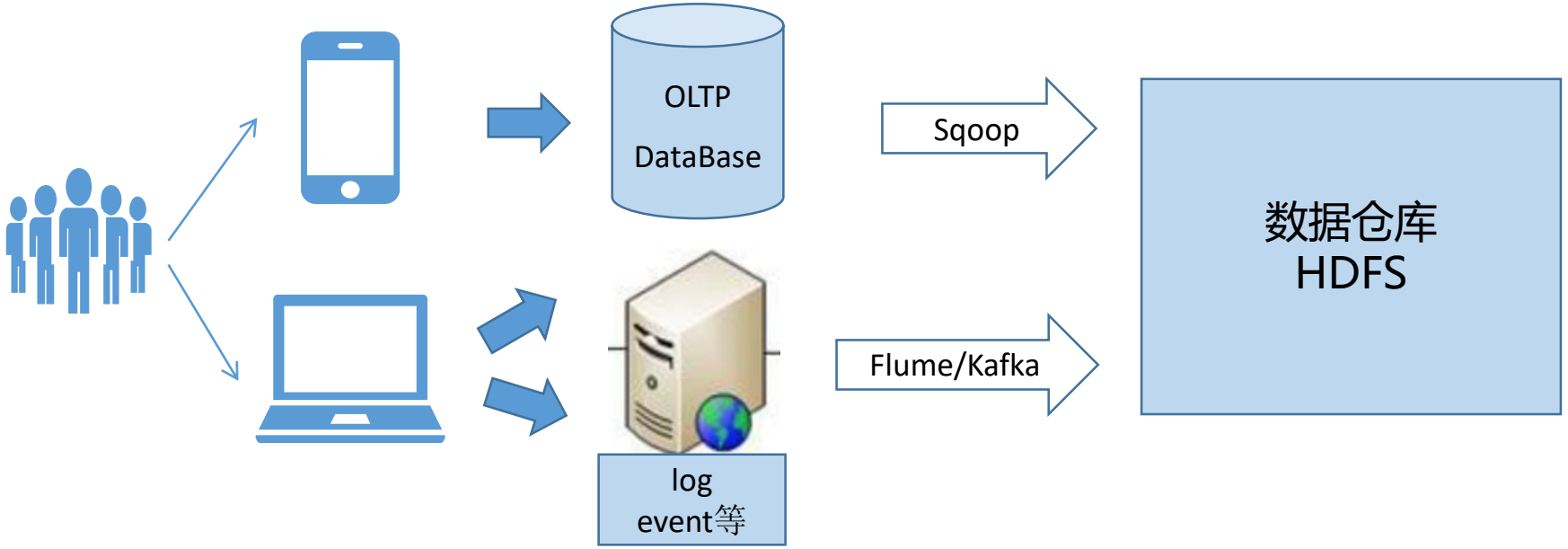


数据仓库分层



- 清晰数据结构：每一个数据分层都有它的作用域，这样我们在使用表的时候能更方便地定位和理解
- 脏数据清洗：屏蔽原始数据的异常
- 屏蔽业务影响：不必改一次业务就需要重新接入数据
- 数据血缘追踪：简单来讲可以这样理解，我们最终给业务呈现的是能直接使用的一张业务表，但是它的来源有很多，如果有一张来源表出问题了，我们希望能够快速准确地定位到问题，并清楚它的危害范围。
- 减少重复开发：规范数据分层，开发一些通用的中间层数据，能够减少极大的重复计算。
- 把复杂问题简单化。将一个复杂的任务分解成多个步骤来完成，每一层只处理单一的步骤，比较简单和容易理解。便于维护数据的准确性，当数据出现问题之后，可以不用修复所有的数据，只需要从有问题的步骤开始修复。

画像数据和数仓数据的关系



- 数据仓库的数据直接对接OLAP或日志类数据，
- 用户画像只是站在用户的角度，对数据仓库数据做进一步的建模加工。因此每天画像标签相关数据的调度依赖上游数据仓库相关任务执行完成。

统计类标签开发案例

回顾一下第四章讲的表结构设计

表结构设计

```
CREATE TABLE `dw.profile_tag_userid`(
`tagid` string COMMENT 'tagid',
`userid` string COMMENT 'userid',
`tagweight` string COMMENT 'tagweight',
`reserve1` string COMMENT '预留1',
`reserve2` string COMMENT '预留2',
`reserve3` string COMMENT '预留3')
COMMENT 'tagid维度userid 用户画像数据'
PARTITIONED BY ( `data_date` string COMMENT '数据日期', `tagtype` string COMMENT '标签主题分类')
```

该表下面记录了标签id、用户id、标签权重等主要字段。按日期和标签主题作为分区。标签主题也作为分区是为了做ETL调度时方便，可以同时计算多个标签插入到该表下面

向hive里面插入几条测试数据，看一下效果

```
--------------------------------插入几条测试数据----------------------------------------
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='user_install_days')
values('A220U029_001','25083679','282','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='user_install_days')
values('A220U029_001','7306783','166','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='user_install_days')
values('A220U029_001','4212236','458','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='user_install_days')
values('A220U029_001','39730187','22','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='user_install_days')
values('A220U029_001','16254215','57','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='userid_all_paid_money')
values('A220U083_001','25083679','800.39','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='userid_all_paid_money')
values('A220U083_001','7306783','311.29','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='userid_all_paid_money')
values('A220U083_001','32171777','129.65','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='userid_all_paid_money')
values('A220U083_001','40382657','602.3','','','');
insert into table dw.profile_tag_userid partition(data_date='20180421', tagtype='userid_all_paid_money')
values('A220U083_001','30765587','465.93','','','');
```

统计类标签开发

案例：用户退货率标签开发

```
#!/usr/bin/env python
# encoding: utf-8

"""
File: userprofile_userid_return_goods_rate.py
Date: 2018/10/01
submit command:
spark-submit --master yarn --deploy-mode client --driver-memory 4g  --executor-memory 8g --executor-cores 2 --num-executors 100 use
"""
from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
import sys
import datetime

# 用户近30日内退货率    B220U071_001
def main():
    start_date = sys.argv[1]
    start_date_str = str(start_date)
    format_1 = "%Y%m%d"

    target_table = 'dw.profile_tag_user'
    strptime, strftime = datetime.datetime.strptime, datetime.datetime.strftime
    old_date_partition_1 = strftime(strptime(start_date_str, format_1) - datetime.timedelta(1), format_1)
    month_day_ago_1 = strftime(strptime(start_date_str, format_1) - datetime.timedelta(30), format_1)

    # 用户近30日订单量
    user_paid_30_orders = " select t1.user_id,
                                   count(distinct t1.order_id) as paid_orders
                              from dw.dw_order_fact t1
                             where t1.pay_status in (1,3)
                               and concat(substr(t1.pay_time,1,4),substr(t1.pay_time,6,2),substr(t1.pay_time,9,2)) >= "+"'"+month_d
                               and concat(substr(t1.pay_time,1,4),substr(t1.pay_time,6,2),substr(t1.pay_time,9,2)) <= "+"'"+start_d
                          group by t1.user_id "

    # 用户近30日订单退货量
    user_paid_30_return = "  select t1.user_id,
                                   count(distinct t1.returned_order_id) as returned_orders
                              from jolly.who_wms_returned_order_info  t1
                             inner join dw.dw_order_fact t2
```

统计类标签开发

案例：购买金额、最近一次购买、累计购买次数等标签开发

```
#!/usr/bin/env python
# encoding: utf-8

"""
File: userprofile_userid_paidinfo.py
Date: 2018/10/01
submit command:
submit command:
spark-submit --master yarn --deploy-mode client --driver-memory 1g  --executor-memory 2g
--executor-cores 2 --num-executors 30 userprofile_userid_paidinfo.py start-date

A220U083_001    累计购买金额
A220U084_001    最近一次购买距今天数
A220U087_001    累计购买次数
A220U088_001    注册未购买
"""

from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
import sys
import datetime

def main():
    start_date = sys.argv[1]
    start_date_str = str(start_date)
    format_1 = "%Y%m%d"
    format_2 = "%Y-%m-%d"
    strptime, strftime = datetime.datetime.strptime, datetime.datetime.strftime
    old_date_partition = strftime(strptime(start_date_str, format_1), format_2)
    target_table = 'dw.profile_tag_user'

    # 累计购买金额
    insert_all_paid_money = " insert overwrite table " + target_table + " partition(data_date="+"'"+start_date_str+"'"+",tagtype='user
                        select 'A220U083_001' as tagid,                         \
                               user_id as userid,                               \
                               sum(order_total_amount) as tagweight,            \
                               '' as tagranking,                                \
                               '' as reserve,                                   \
                               '' as reserve1                                   \
```

规则类标签开发案例

规则类标签开发

案例：用户RFM模型标签开发

```
#!/usr/bin/env python
# encoding: utf-8

"""
File: userprofile_userid_RFM_value.py
Date: 2018/10/01
submit command:
spark-submit --master yarn --deploy-mode client --driver-memory 1g  --executor-memory 2g
--executor-cores 2 --num-executors 50 userprofile_userid_RFM_value.py start-date
"""

# A111U008_001    重要价值用户
# A111U008_002    重要保持用户
# A111U008_003    重要发展用户
# A111U008_004    重要挽留用户
# A111U008_005    一般价值用户
# A111U008_006    一般保持用户
# A111U008_007    一般发展用户
# A111U008_008    一般挽留用户
# dim_user_info
# user_consume_info

from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
import sys
import datetime

def main():
    start_date = sys.argv[1]
    start_date_str = str(start_date)
    date_str = datetime.datetime.strftime(datetime.date.today()-datetime.timedelta(days=1),'%Y-%m-%d')
    format = "%Y%m%d"
    target_table = 'dw.profile_tag_user'


    # 用户RFM维度数据 (用户最后一次购买时间非空)
    user_rfm_info =  "  select t1.user_id,                                    \
                               t2.country,                                    \
                               t1.last_1y_paid_orders,                        \
```

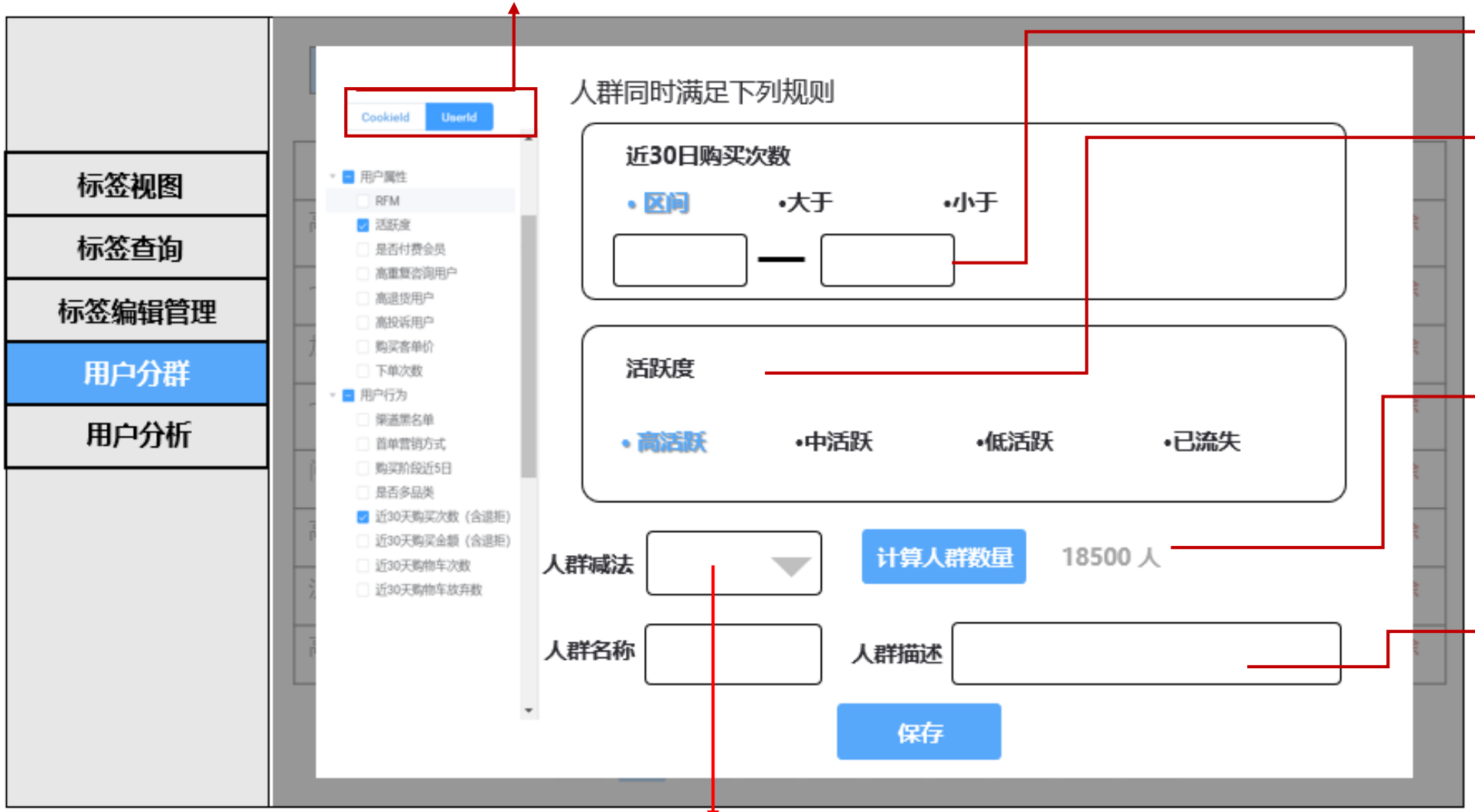
规则类标签开发

案例：ID-MAPPING，根据userid维度的用户RFM模型开发cookieid维度对应的标签

```
# A111H008_008     一般挽留用户
# dim_user_info
# user_consume_info

from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
import sys
import datetime

def main():
    start_date = sys.argv[1]
    start_date_str = str(start_date)
    format = "%Y%m%d"
    target_table = 'dw.profile_tag_user'

    # 用户RFM维度数据 (用户最后一次购买时间非空)
    user_cookie_relation =  " select t.userid,                                                                                \
                                     t.cookie_id as cookieid                                                                  \
                              from (                                                                                          \
                                    select cookie_id,                                                                         \
                                           userid,                                                                            \
                                           row_number() over(partition by cookie_id order by last_time,last_date,userid desc) as rank  \
                                      from dw.cookie_user_relation                                                            \
                                     where cookie_id is not null                                                              \
                                       and userid is not null                                                                 \
                                    ) t                                                                                       \
                              where t.rank =1 "


    insert_table = "insert overwrite table " + target_table + " partition(data_date="+"'"+start_date_str+"'"+",tagtype='cookie_rfm_model')
                          select case when t1.tagid = 'A111U008_001' then 'A111H008_001'     \
                                      when t1.tagid = 'A111U008_002' then 'A111H008_002'     \
                                      when t1.tagid = 'A111U008_003' then 'A111H008_003'     \
                                      when t1.tagid = 'A111U008_004' then 'A111H008_004'     \
                                      when t1.tagid = 'A111U008_005' then 'A111H008_005'     \
                                      when t1.tagid = 'A111U008_006' then 'A111H008_006'     \
                                      when t1.tagid = 'A111U008_007' then 'A111H008_007'     \
                                 else 'A111H008_008' end as tagid,          \
                                 t2.cookieid,                               \
                                 '' as tagweight,                           \
```

标签数据在产品上的应用

对应筛选的用户维度是在userid或cookieid，筛选不同的维度，推送不同的数据到对应的业务系统中



对于统计类型标签，用户可以自定义筛选该标签的权重值

对于分类类型标签，用户不能筛选该标签对应的权重值(该标签只是用户的一个标识，不含有统计数值)

根据用户组合标签定义的人群，计算出该人群覆盖的用户数

对自己组合标签筛选出来的用户群命名，然后保存

下页进行详细介绍“**人群减法**”的功能

挖掘类标签开发案例

判断用户男女性别

脚本在：gender_model_v2.html

```
    6. 20 the most important features

'''
```

In [1]:
```
#设置路径
import os
path = os.path.dirname(os.path.dirname(os.path.abspath('gender_model_v2')))
os.chdir(path)
#print "current directory is {}".format(os.getcwd())
```

```
current directory is /Users/Max/Desktop/gender_project_final
```

In [2]:
```
# basic
import pandas as pd
import numpy as np
from collections import Counter
import warnings
warnings.filterwarnings('ignore')

# machine learning
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
```

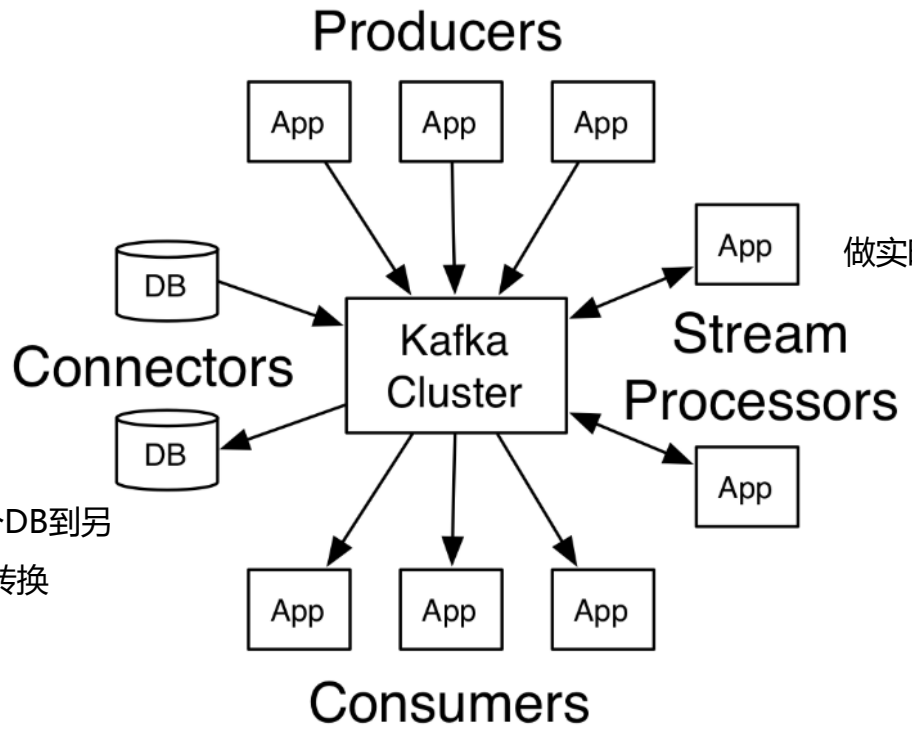
模型评估

```
# analysis report
estimator = clf
print "分析报告生成中.....\n"
print 'performance on training dateset .....  \n\n {}'.format(classification_report(Y_train, estimator.predict(X_train_scaler)))
print 'performance on test dataset .....  \n \n {}'.format(classification_report(Y_val, clf_pred_y))
print 'performance on full dataset .....  \n \n {}'.format(classification_report(np.concatenate((Y_train, Y_val), axis=0), estimator.predict(n
p.vstack((X_train_scaler, X_test_scaler)))))
```

```
分析报告生成中.....

performance on training dateset .....

             precision    recall  f1-score   support

          0       0.87      0.86      0.86     19035
          1       0.79      0.80      0.80     12689

avg / total       0.84      0.83      0.83     31724

performance on test dataset .....

             precision    recall  f1-score   support

          0       0.87      0.85      0.86      4780
          1       0.77      0.80      0.79      3151

avg / total       0.83      0.83      0.83      7931
```

流式计算标签开发—kafka介绍

Kafka基本介绍



做实时流处理

将数据从一个DB到另一个DB转换

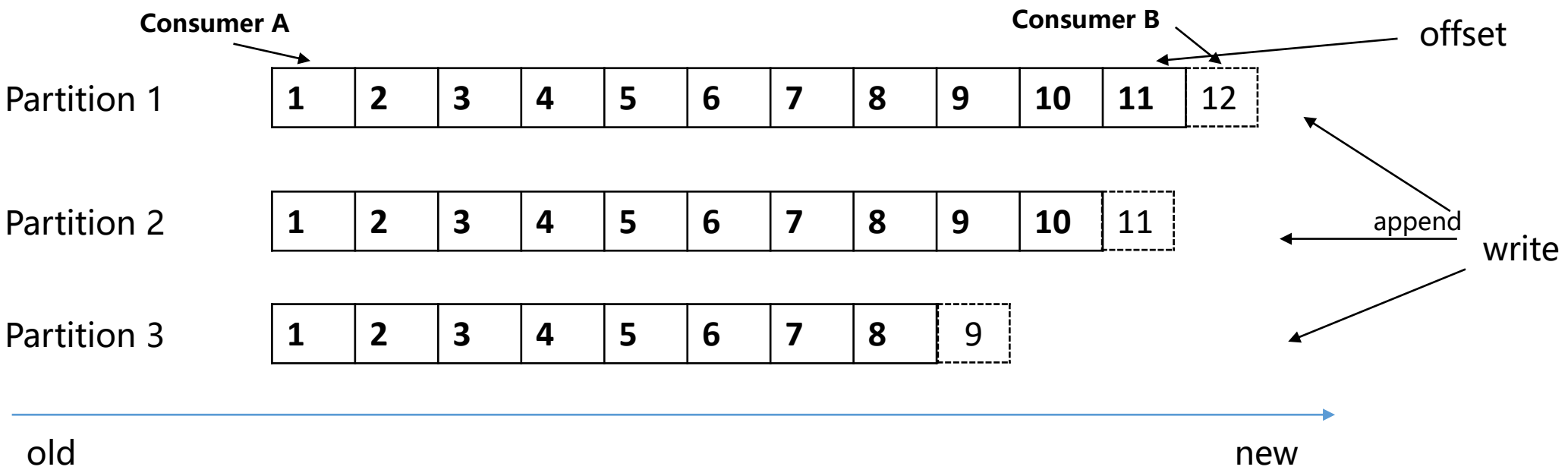
消息中间件

- **producer**：产生信息的主体；
- **consumer**：消费producer产生信息的主体；
- **broker**：消息处理节点，多个broker组成kafka集群；
- topic：是数据主题，是数据记录发布的地方,可以用来区分业务系统；
- **partition**：是topic的分组，每个partition都是一个有序队列
- offset：用于定位消费者在每个partition中消费到的位置

Spark Streaming + Kafka 集成指南 (Kafka broker version 0.8.2.1 or higher)

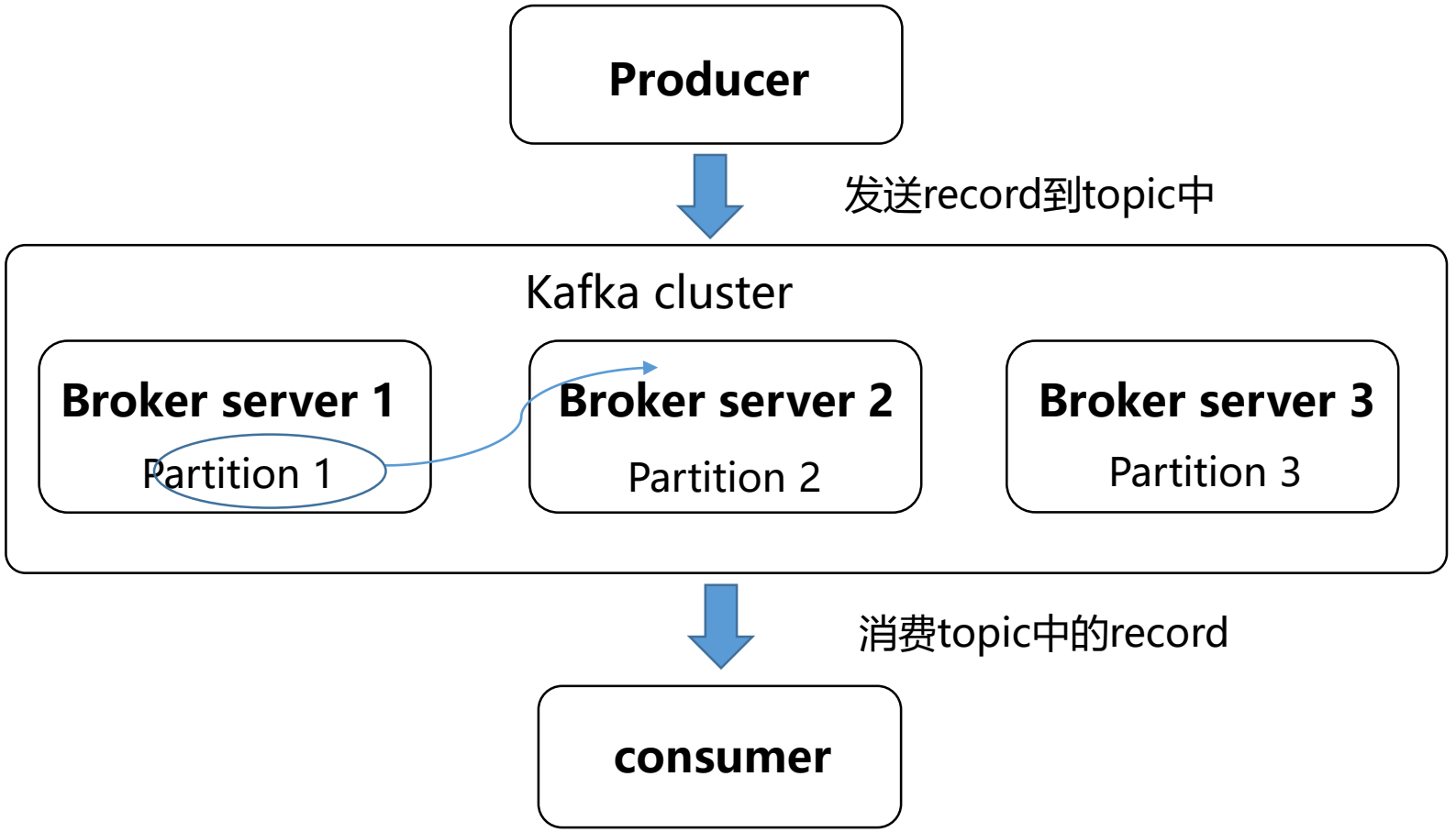
http://spark.apachecn.org/docs/cn/2.2.0/streaming-kafka-0-8-integration.html

partition和offset（1）



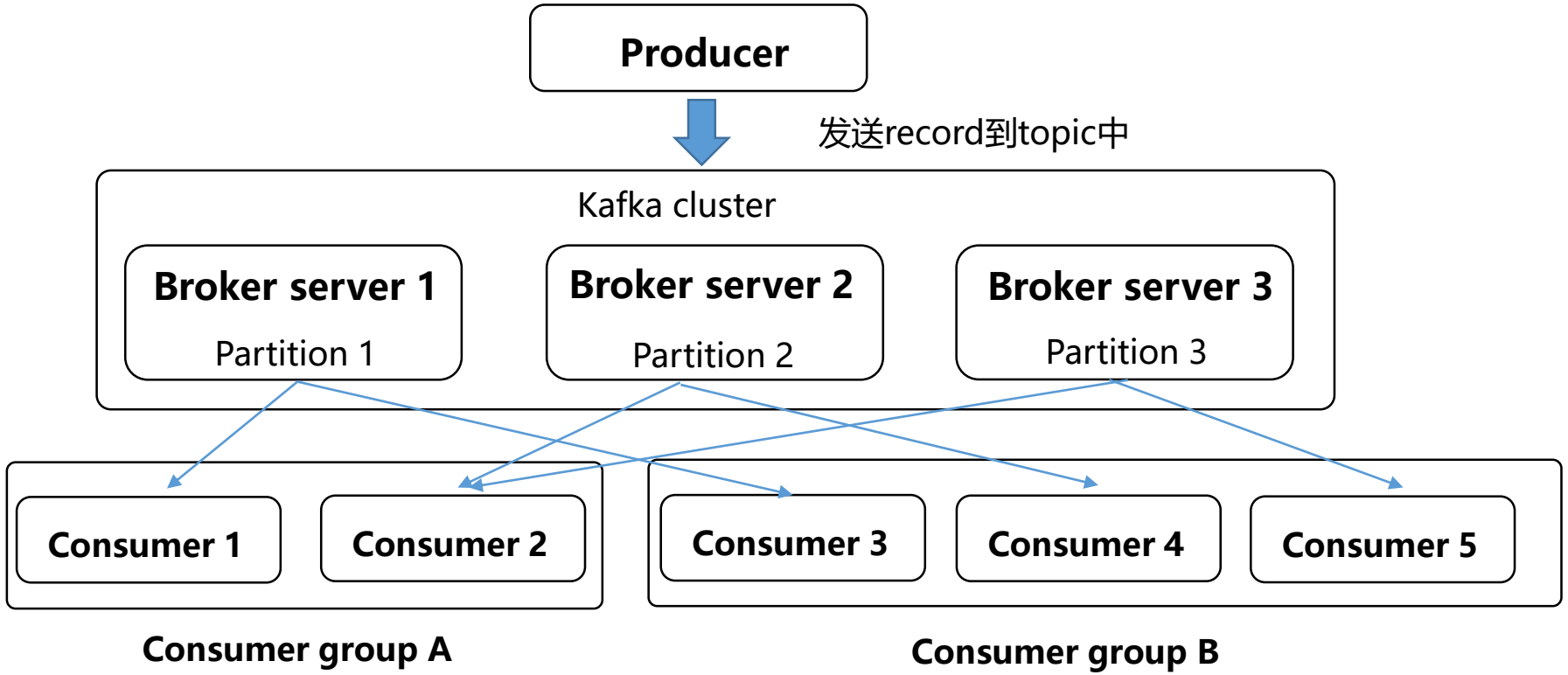
- Topic在broker里面对应着一系列的消息数据，这些消息数据存储在kafka磁盘里面。默认存储7天；
- 对于发过来的消息，是写入到哪个partition上去，是有规则的；
- 每个partition都是一个有序的、不变的record序列。接收到的record追加到这个序列中；
- Offset是偏移量，标识每条消息的唯一标识；
- 一个topic对应的数据是分了好几个区，这些分区都是分布式地分布在多个broker server上，每个分区又备份几份在其他broker上，这样保证了集群的高可用性；
- Offset是consumer控制的，所以consumer可以按照不同需求消费任何位置的数据；

partition和offset（2）



- Producer向topic发消息时，topic以分区形式存在，topic以分区的形式存在broker server中；
- Topic的partition是分布式地分布在多个broker server中；
- 多个接收器接收发送来的数据，这样的吞吐量较高；
- 每个partition都备份到其他broker server中去，有好几个备份。一个partition所在的broker挂掉了，其他上broker上对应的partition同样可对外提供服务，这样partition数据的高可用性得到了保证；

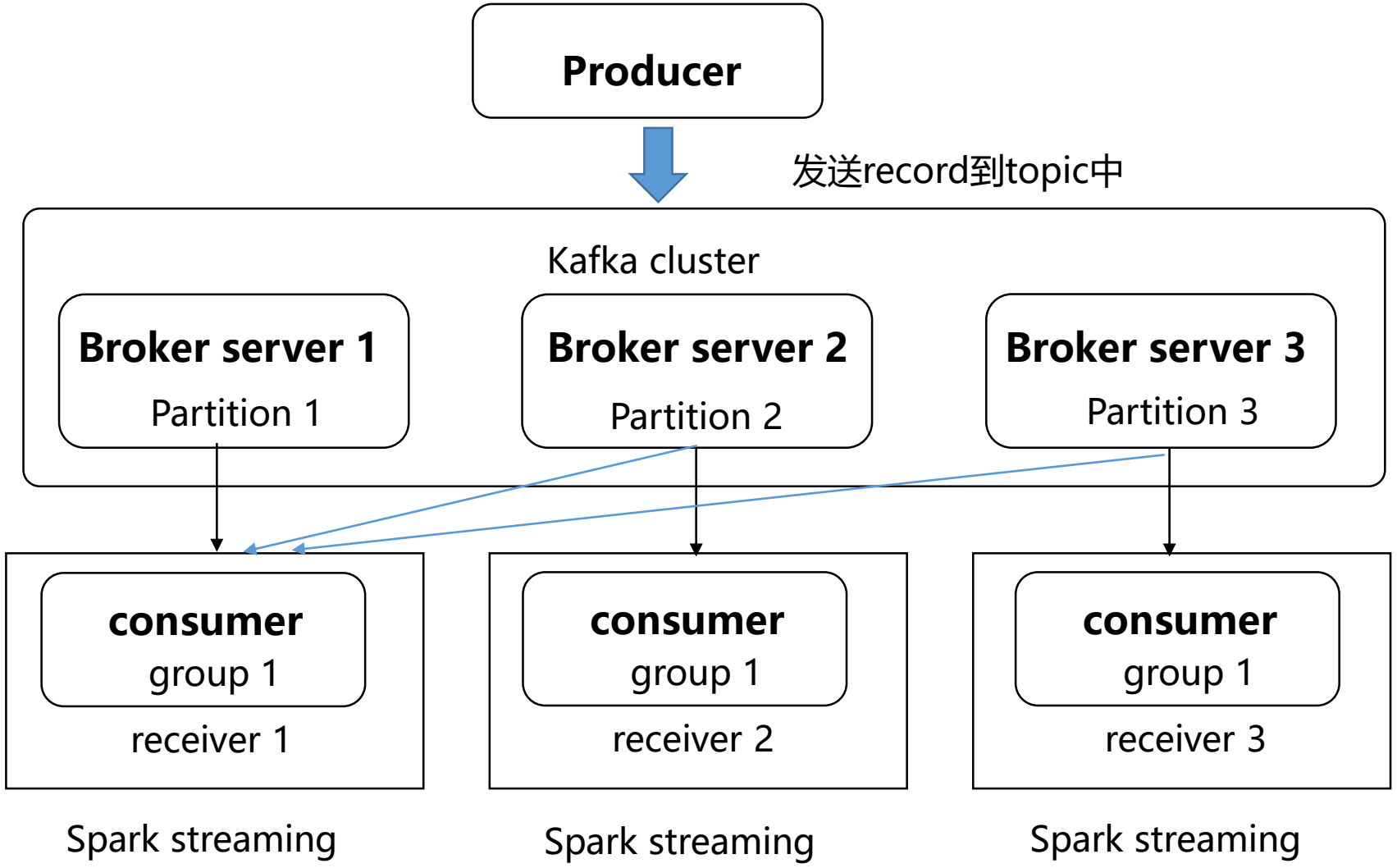
consumer group



- **Consumer消费消息时候处理需要指定topic外，还需要指定这个consumer属于哪个consumer group。每个consumer group消费topic下的所有partition数据，**每个consumer消费topic中的partitions数据时候是按offset来进行的；
- 每一个consumer都被归为一个consumer group，同时一个consumer group可以包含一个或多个consumer；
- 一个topic中一条record会被所有订阅这个topic的consumer gourp消费。即每一个consumer实例都属于一个consumer group，每一条消息只会被同一个consumer group里的一个consumer实例消费。不同的consumer group可以同时消费同一条数据；
- ***一个consumer group会消费一个topic里所有的数据***

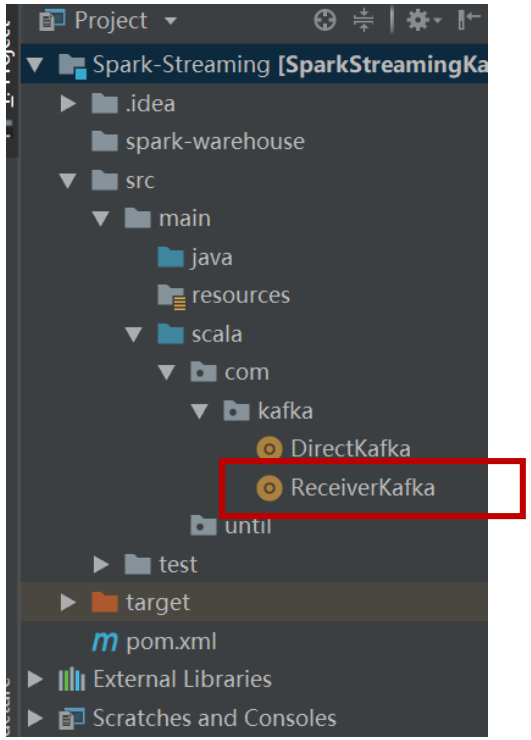
流式计算标签开发—streaming的Receiver模式与Direct模式

Receiver模式



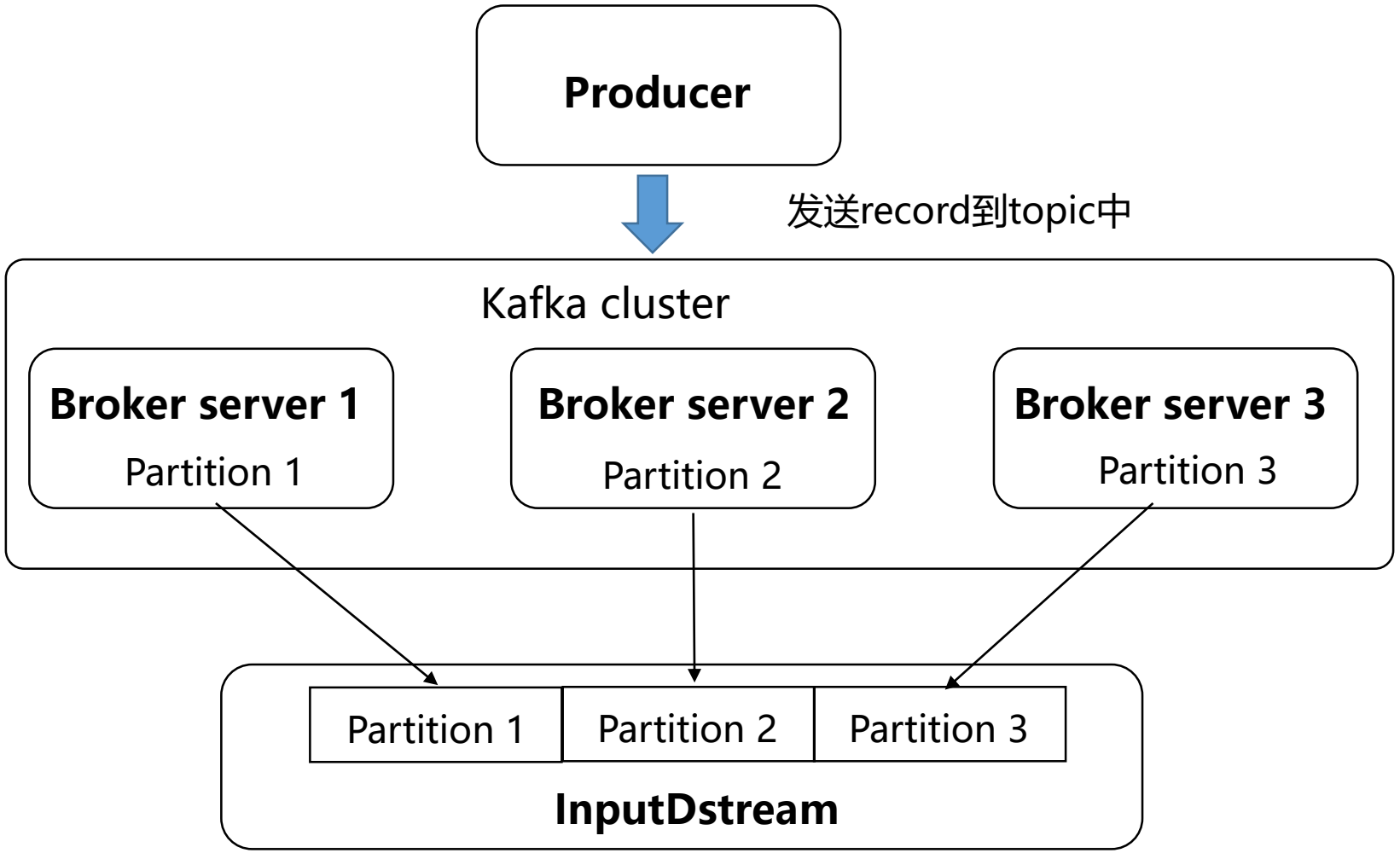
- Receiver实时拉取kafka里面的数据；
- 一个receiver接收数据不过来时，再起其他receiver接收，他们同属于一个consumer group，这样提高了streaming程序的吞吐量；
- kafka中的topic是以partition的方式存在的，Spark中的partition和kafka中的partition并不是相关的，如果我们加大每个topic的partition数量，仅仅是增加线程来处理由单一Receiver消费的主题。但是这并没有增加Spark在处理数据上的并行度

Receiver代码示例



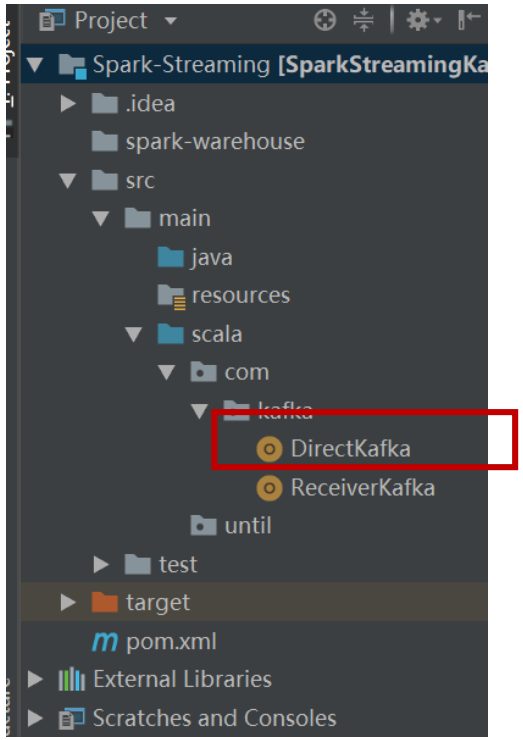

```scala
val sparkConf = new SparkConf().setAppName("SparkStreaming-ReceivertKafka")
val sc = new SparkContext(sparkConf)
val Array(zkQuorum, group, topics, numThreads) = args
val ssc = new StreamingContext(sc, Seconds(2))
val topicMap = topics.split( regex = ",").map((_, numThreads.toInt)).toMap
val kafkaParams = Map[String, String](
  elems = "zookeeper.connect" -> zkQuorum, "group.id" -> group,
  "zookeeper.connection.timeout.ms" -> "10000",
  "auto.offset.reset" -> "largest")

val numStreams = 3
val kafkaStreams = (1 to numStreams).map { _ =>
  KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
    ssc, kafkaParams, topicMap, StorageLevel.MEMORY_AND_DISK_SER_2) }
val unifiedStream = ssc.union(kafkaStreams)

val messages = unifiedStream.map(_._2)
val words = messages.flatMap(_.split( regex = " "))
val wordcounts = words.map(x => (x, 1L)).
  reduceByKeyAndWindow(_ + _, _ - _,Minutes(1), Seconds(2), numPartitions = 2)
wordcounts.print()
ssc.start()
ssc.awaitTermination()
```

Direct模式



- 会周期性查询kafka，获得每个topic、partition的offset；
- 不需要创建多个输入Dstream然后进行union操作；
- 会创建和kafka partition一样多的RDD partition，并行从kafka读取数据；

Direct代码示例




```scala
val sparkConf = new SparkConf().setAppName("SparkStreaming-DirectKafka")
val sc = new SparkContext(sparkConf)

val Array(brokers, topics) = args
val ssc = new StreamingContext(sc, Seconds(2))
val topicset = topics.split( regex = ",").toSet
val KafkaParams = Map[String,String]( elems = "metadata.broker.list" -> brokers)
val directKafkaStream = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
  ssc, KafkaParams, topicset)

var offsetRanges = Array.empty[OffsetRange]
directKafkaStream.print()

directKafkaStream.transform { rdd =>
  offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  rdd
}.map(_._2)
  .flatMap(_.split( regex = " "))
  .map(x => (x, 1L))
  .reduceByKey(_ + _)
  .foreachRDD { rdd =>
    for (o <- offsetRanges) {
      println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")
    }
    rdd.take( num = 10).foreach(println)
  }
```

Receiver模式与Direct模式对比

Receiver模式

KafkaReceiver	at most once	最多被处理一次	会丢失数据
ReliableKafkaReceiver	at least once	最少被处理一次	不会丢失数据

Direct模式

Exactly once	只被处理一次	sparkstreaming自己跟踪消费的offset，是直接消费存储在kafka中的数据

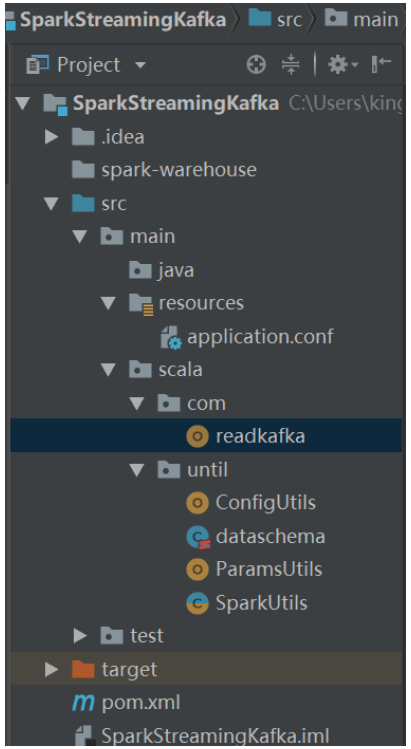
Receiver方式是通过zookeeper来连接kafka队列，Direct方式是直接连接kafka节点来获取数据。Direct模式消除了与zk不一致的情况，Receiver模式消费的kafka topic的offset是保存在zk中的。所以基于Direct模式可以使得spark streaming应用完全达到Exactly once语义的情况。

SparkStreaming对kafka集成有两个版本，一个0.8一个是0.10以后的。0.10以后只保留了direct模式

流式计算标签开发—记录消费的offset

Direct模式保存offset

为了保证Spark Streaming在不丢失数据，在Direct模式下需要记录消费的offset数据。




```scala
val size = dauOffsetList.size

val inpDS: InputDStream[(String, String)] = if (size < 0) {
  //这个会将 kafka 的消息进行 transform，最终 kafka 的数据都会变成 (topic_name, message) 这样的 tuple
  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder, (String, String)](
    ssc, ParamsUtils.kafka.KAFKA_PARAMS, dauFromOffsets, dauMessageHandler)
} else {
  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
    ssc, ParamsUtils.kafka.KAFKA_PARAMS, ParamsUtils.kafka.KAFKA_TOPIC)
}
```

Direct模式保存offset

```
// messages 从kafka获取数据,将数据转为RDD
messages.foreachRDD((rdd, batchTime) => {
  import org.apache.spark.streaming.kafka.HasOffsetRanges
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges   // 获取偏移量信息
  /**
    * OffsetRange 是对topic name，partition id，fromOffset(当前消费的开始偏移)，untilOffset(当前消费的结束偏移)的封装。
    * *  所以OffsetRange 包含信息有：topic名字，分区Id，开始偏移，结束偏移
    */
  println("==========================> count: " + rdd.map(x => x + "1").count())
 // offsetRanges.foreach(offset => println(offset.topic, offset.partition, offset.fromOffset, offset.untilOffset))
  for (offset <- offsetRanges) {
    // 遍历offsetRanges,里面有多个partition
    println(offset.topic, offset.partition, offset.fromOffset, offset.untilOffset)
    DBs.setupAll()
    // 将partition及对应的untilOffset存到MySQL中
    val saveoffset = DB localTx {
      implicit session =>
       sql"DELETE FROM offsetinfo WHERE topic = ${offset.topic} AND partitionname = ${offset.partition}".update.apply()
        sql"INSERT INTO offsetinfo (topic, partitionname, untilOffset) VALUES (${offset.topic},${offset.partition},${offset.untilOffset})".update.apply()
    }
  }
```

将offset循环写入到MySQL中

Direct模式保存offset

MySQL中记录消费的topic、partition、offset

topic	partitionname	untilOffset
countly_imp	0	1628669
countly_imp	1	1654888
countly_imp	10	1628693
countly_imp	11	1628686
countly_imp	2	1628680
countly_imp	3	1628685
countly_imp	4	1628667
countly_imp	5	1654895
countly_imp	6	1628696
countly_imp	7	1628682
countly_imp	8	1628671
countly_imp	9	1654890

流式计算标签开发—Streaming上线工程化

读取kafka数据

本章案例的程序对应 Spark-Streaming 文件夹

```
val dauTopic = topics.head // ParamsUtils.kafka.KAFKA_TOPIC.toString  //"countly_imp"
val dauOffsetList = List((dauTopic,1,1651983),
  (dauTopic,2,1625775),
  (dauTopic,3,1625780),
  (dauTopic,4,1625761),
  (dauTopic,5,1651990),
  (dauTopic,6,1625791),
  (dauTopic,7,1625776),
  (dauTopic,8,1625766))
 val dauFromOffsets = readFromOffsets(dauOffsetList)
val dauMessageHandler = (mmd: MessageAndMetadata[String, String]) => (mmd.key, mmd.message)

val size = dauOffsetList.size

val inpDS: InputDStream[(String, String)] = if (size < 0) {
  //这个会将 kafka 的消息进行 transform，最终 kafka 的数据都会变成 (topic_name, message) 这样的 tuple
  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder, (String, String)](
    ssc, ParamsUtils.kafka.KAFKA_PARAMS, dauFromOffsets, dauMessageHandler)
} else {
  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
    ssc, ParamsUtils.kafka.KAFKA_PARAMS, ParamsUtils.kafka.KAFKA_TOPIC)
}
```

解析RDD数据-解析json

```
// 处理从kafka获取的message信息
val parameter = messages.flatMap(line => {
  //获取服务端事件日期 reqts_day
  val reqts_day = try {
    new DateTime(JSON.parseObject(line._2).getJSONObject( key = "i").getLong( key = "timestamp") * 1000).toDateTime.toString(
  } catch {
    case ex: Exception => "(unknown)"
  }

  //获取 设备号
  val cookieid = try {
    JSON.parseObject(line._2).getJSONObject( key = "d").get("d")    //将Json字符串转化为相应的对象   .getString("kid")
  } catch {
    case ex: Exception => "(unknown)"
  }

  //组合成一个字符串
  val data = reqts_day + "##" + cookieid
  Some(data)        //some是一定有值的, some.get获取值,如果没有值,会报异常
}).map(_.split( regex = "##")).map(x => (x(0),x(1)))
```

存储offset偏移数据

```
// messages 从kafka获取数据,将数据转为RDD
messages.foreachRDD((rdd, batchTime) => {
  import org.apache.spark.streaming.kafka.HasOffsetRanges
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges    // 获取偏移量信息
  /**
    * OffsetRange 是对topic name，partition id，fromOffset(当前消费的开始偏移)，untilOffset(当前消费的结束偏移)的封装。
    * *  所以OffsetRange 包含信息有：topic名字，分区Id，开始偏移，结束偏移
    */
  println("============================> count: " + rdd.map(x => x + "1").count())
 // offsetRanges.foreach(offset => println(offset.topic, offset.partition, offset.fromOffset, offset.untilOffset))
  for (offset <- offsetRanges) {
    // 遍历offsetRanges,里面有多个partition
    println(offset.topic, offset.partition, offset.fromOffset, offset.untilOffset)
    DBs.setupAll()
    // 将partition及对应的untilOffset存到MySQL中
    val saveoffset = DB localTx {
      implicit session =>
       sql"DELETE FROM offsetinfo WHERE topic = ${offset.topic} AND partitionname = ${offset.partition}".update.apply()
        sql"INSERT INTO offsetinfo (topic, partitionname, untilOffset) VALUES (${offset.topic},${offset.partition},${offset
    }
  }
})
```

将kafka数据存储到MySQL中

```
parameter.foreachRDD{ rdd =>

  val sqlContext = new org.apache.spark.sql.SQLContext(sc)
  import sqlContext.implicits._
  // 转换成DataFrame
  val SaveParameter = rdd.map(w => dataschema(w._1.toString,w._2.toString)).toDF( colNames = "data_date","cookies_num")
  // 注册视图
  SaveParameter.createOrReplaceTempView( viewName = "dau_tmp_table")
  val insertsql =sqlContext.sql( sqlText = "select * from dau_tmp_table")
  insertsql.write.mode(SaveMode.Append).jdbc( url = "jdbc:mysql://localhost:3306/userprofile_test", table = "dau_tmp_table",Pa

}

messages.print()
ssc.start()
ssc.awaitTermination()
```