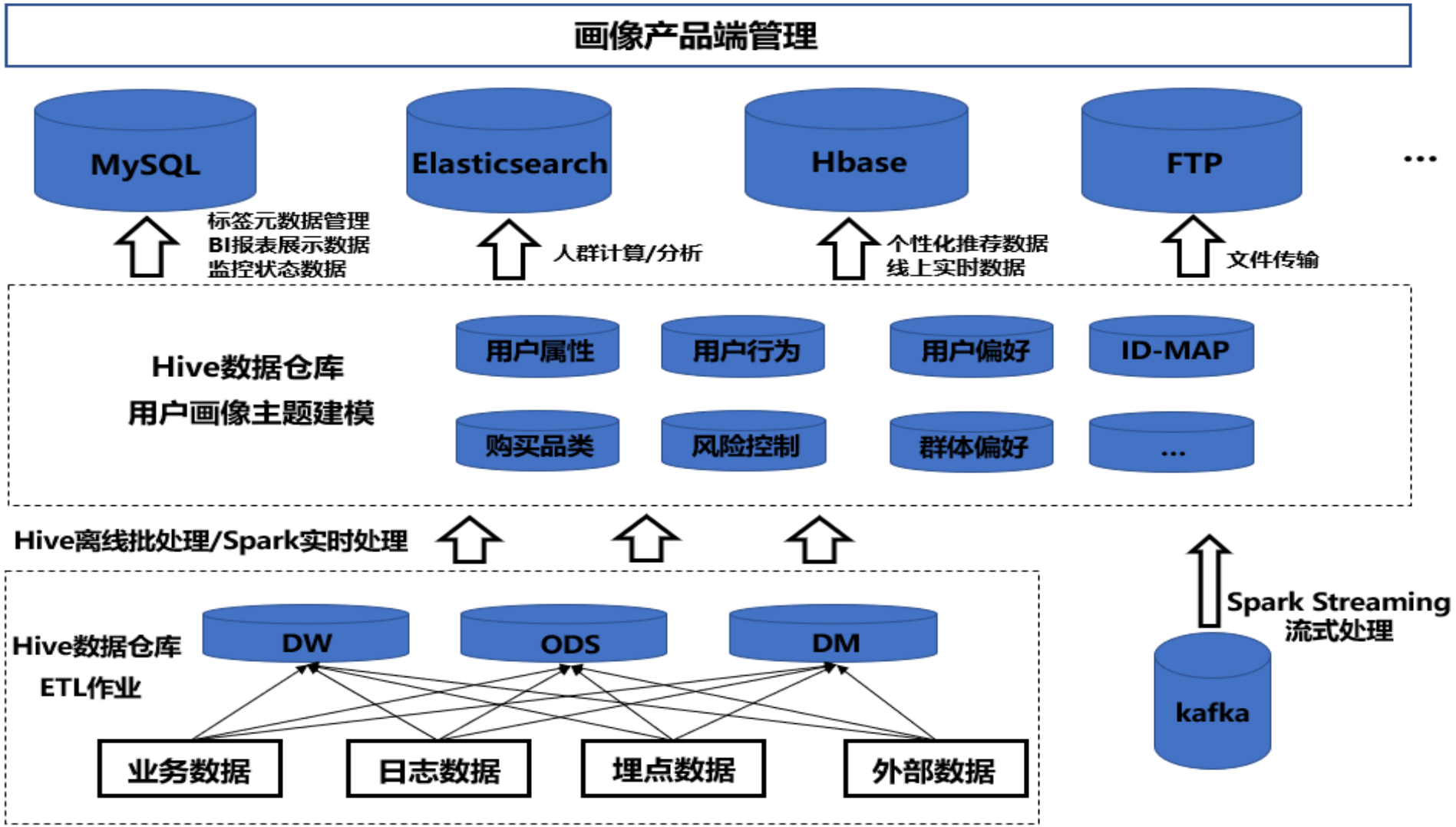
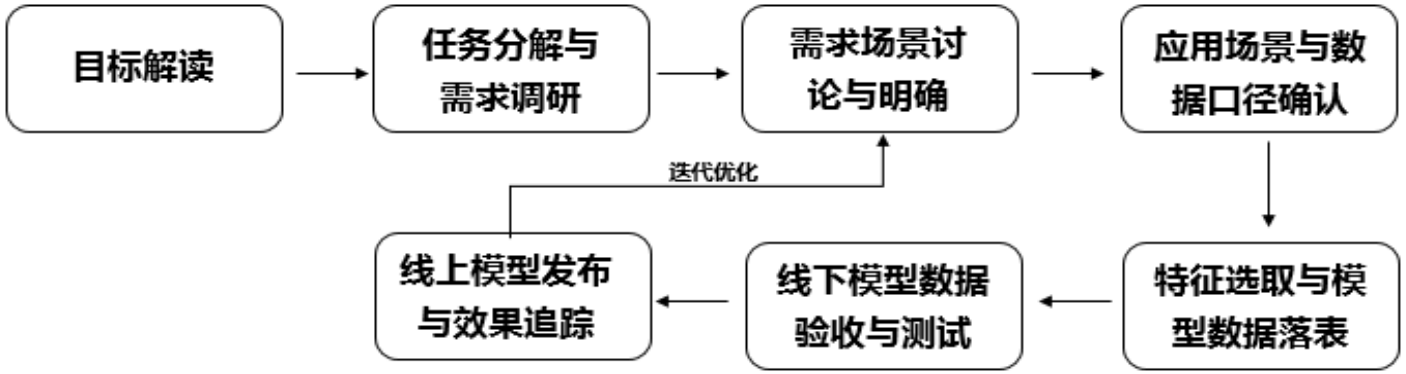
如何从0到1构建用户画像系统

数据架构

从用户画像的数据架构谈需要掌握的大数据模块和开发语言



开发流程



第一阶段：目标解读

在建立用户画像前，首先需要明确用户画像服务于企业的对象，根据业务方需求，未来产品建设目标和用户画像分析之后预期效果；

第二阶段：任务分解与需求调研

经过第一阶段的需求调研和目标解读，我们已经明确了用户画像的服务对象与应用场景，接下来需要针对服务对象的需求侧重点，结合产品现有业务体系和“数据字典”规约实体和标签之间的关联关系，明确分析纬度；

第三阶段：需求场景讨论与明确

在本阶段，数据运营人员需要根据前面与需求方的沟通结果，输出**《产品用户画像规划文档》**，在该文档中明确画像应用场景、最终开发出的标签内容与应用方式，并就该份文档与需求方反复沟通确认无误。

第四阶段：应用场景与数据口径确认

经过第三个阶段明确了需求场景与最终实现的标签纬度、标签类型后，数据运营人员需要结合业务与数据仓库中已有的相关表，明确与各业务场景相关的数据口径。在该阶段中，数据运营方需要输出**《产品用户画像实施文档》**，该文档需要明确应用场景、标签开发的模型、涉及到的数据库与表，应用实施流程；

第五阶段：特征选取与模型数据落表

本阶段中数据分析挖掘人员需要根据前面明确的需求场景进行业务建模，写好HQL逻辑，将相应的模型逻辑写入临时表中，抽取数据校验是否符合业务场景需求。

第六阶段：线下模型数据验收与测试

数据仓库团队的人员将相关数据落表后，设置定时调度任务，进行定期增量更新数据。数据运营人员需要验收数仓加工的HQL逻辑是否符合需求，根据业务需求抽取查看表中数据范围是否在合理范围内，如果发现问题及时反馈给数据仓库人员调整代码逻辑和行为权重的数值。

第七阶段：线上模型发布与效果追踪

经过第六阶段，数据通过验收之后，就可以将数据接口给到搜索、或技术团队部署上线了。上线后通过对用户点击转化行为的持续追踪，调整优化模型及相关权重配置。

各阶段关键产出物

任务类型	任务名称	任务内容	所需时间	重点内容
标签开发	性别标签开发	数据调研、熟悉数据字典，开发标签（包括统计类、算法类、实时类的标签）	xx天	数据调研、和业务方确认数据口径，标签开发上线。初期上线满足应用需求
	会员标签开发		xx天	
	活跃度标签开发		xx天	
	RFM标签开发		xx天	
	……	……	……	
ETL调度开发	任务依赖关系梳理	梳理各任务之间的依赖关系	xx天	满足定时调度，监控预警、失败重试，各调度任务之间的复杂依赖关系
	监控脚本开发	开发标签监控、人群计算监控、服务层监控等相关脚本	xx天	
	调度脚本开发	根据梳理的各任务间调度依赖，开发调度流脚本	xx天	
	上线调度系统	调度流脚本上线调度试运行/正式运行	xx天	
打通服务层接口	push系统业务对接沟通	画像人群数据和push系统的打通方案，开发方式确定	xx天	打通数据仓库数据和各业务系统之间的通路，提供稳健的服务
	外呼系统业务对接沟通	和外呼团队了解外呼业务场景，确定打通方式	xx天	
	广告系统业务对接沟通	和广告团队了解目前广告场景，确定打通方式	xx天	
	客服系统业务对接沟通	和客服团队确认系统打通方式	xx天	
	……	……	……	
画像产品化	产品经理与业务人员、技术开发对接沟通	确定产品功能、画原型、明确开发排期	xx天	产品交互友好，能支持到业务方对用户进行分析、精细运营的需求
	JAVA Web端开发	开发测试、内测	xx天	
	产品上线	通知各业务方使用产品	xx天	
开发调优	标签脚本、调度脚本的重构优化	梳理现有标签开发、调度、校验告警、同步到服务层等相关脚本，明确可以优化的地方，迭代优化	xx天	减少ETL调度时间，降低调度时消耗资源
面向业务方推广应用	写画像使用文档	面向数据分析师、业务人员等群体撰写详细的画像使用文档，包括相关表及元数据、产品使用手册等	xx天	帮助业务人员将画像数据应用到业务中去，提高用户活跃、提高GMV
	提供业务支持	针对业务场景，为业务方提供画像解决方案	xx天	

需要掌握的相关模块

数据开发

Spark

数据存储和查询

Hbase

Hive

MySQL

流式计算

Kafka

Spark Streaming

作业调度(ETL)

crontab

Airflow

需要掌握的其他非技术内容

1、数据分析
如何做数据调研、对于需求方提出的标签如何结合数据情况给出相应的解决方案；
…

2、用户画像工程化
用户标签体系中需要开发哪些标签；
这些标签的调度流是如何构成的；
如何对每天的调度作业进行监控；
哪些数据库可用于存储标签，为何用这些数据库进行存储；
…

3、业务知识
需要开发的标签服务于业务上的哪些应用
….

4、画像产品形态及如何服务与业务
画像产品化后包括哪些模块；
如何评价标签在业务上的应用效果；
…

表结构设计—日全量

日全量数据表中，每天对应的日期分区中插入截止到当天为止的全量数据，用户使用查询时，只需查询最近一天即可获得最新全量数据。下面以一个具体的日全量表结构例子来做说明。

```
CREATE TABLE dw.userprofile_tag_userid (
tagid STRING COMMENT 'tagid',
userid STRING COMMENT 'userid',
tagweight STRING COMMENT 'tagweight',
reserve STRING COMMENT '预留' )
PARTITIONED BY (data_date STRING COMMENT '数据日期' ,tagtype STRING COMMENT '标签主题分类')
```

这里tagid表示标签名称，userid表示用户id，tagweight表示标签权重，reserve表示一个预留字段。分区方式为（日期+标签主题）分区，设置两个分区字段更便于开发和查询数据。该表结构下的标签权重仅考虑统计类型标签的权重，如：历史购买金额标签对应的权重为金额数量，用户近30日访问天数为对应的天数，该权重值的计算未考虑较为复杂的用户行为次数、行为类型、行为距今时间等复杂情况。**通过全连接的方式与前面每天的数据做join关联**

例如，对于主题类型为“会员”的标签，插入‘20180701’日的全量数据，可通过语句：insert overwrite table dw.userprofile_tag_userid partition(data_date=’20180701’，tagtype=’member’)来实现。查询截止到20180701日的被打上会员标签的用户量，可通过语句：select count(*) from dw.userprofile_tag_userid where data_date=’20180701’来实现。

表结构设计—日增量

日增量数据表中，每天的日期分区中插入当天业务运行产生的数据，用户使用查询时需要限制查询的日期范围，可以找出在特定时间范围内被打上特定标签的用户。下面以一个具体的日增量表结构例子来说明。

```
CREATE TABLE  dw.userprofile_useract_tag (
tagid STRING COMMENT '标签id',
userid STRING COMMENT '用户id',
act_cnt int COMMENT '行为次数',
tag_type_id int COMMENT '标签类型编码',
act_type_id int COMMENT '行为类型编码')
comment '用户画像-用户行为标签表’  PARTITIONED BY (data_date STRING COMMENT '数据日期')
```

这里tagid表示标签名称，userid表示用户id，act_cnt表示用户当日行为次数，如用户当日浏览某三级品类商品3次，则打上次数为3，tag_type_id为标签类型,如母婴、3C、数码等不同类型，act_type_id表示行为类型，如浏览、搜索、收藏、下单等行为。分区方式为按日期分区，插入当日数据。

例如，某用户在‘20180701’日浏览某3C电子商品4次（act_cnt），即给该用户（userid）打上商品对应的三级品类标签（tagid），标签类型（tag_type_id）为3C电子商品，行为类型（act_type_id）为浏览。这里可以通过对标签类型和行为类型两个字段以配置维度表的方式，对数据进行管理。例如对于行为类型（act_type_id）字段，可以设定1为购买行为、2为浏览行为、3为收藏行为等，在行为标签表中以数值定义用户行为类型，在维度表中维护每个数值对应的具体含义。

标签类型

用户画像建模其实就是对用户进行打标签，从对用户打标签的方式来看，一般分为三种类型：1、基于统计类的标签；2、基于规则类的标签、3、基于挖掘类的标签。下面我们介绍这三种类型标签的区别：

- **统计类标签**：这类标签是最为基础也最为常见的标签类型，例如对于某个用户来说，他的性别、年龄、城市、星座、近7日活跃时长、近7日活跃天数、近7日活跃次数等字段可以从用户注册数据、用户访问、消费类数据中统计得出。该类标签构成了用户画像的基础；
- **规则类标签**：该类标签基于用户行为及确定的规则产生。例如对平台上“消费活跃”用户这一口径的定义为近30天交易次数>=2。在实际开发画像的过程中，由于运营人员对业务更为熟悉、而数据人员对数据的结构、分布、特征更为熟悉，因此规则类标签的规则确定由运营人员和数据人员共同协商确定；
- **机器学习挖掘类标签**：该类标签通过数据挖掘产生，应用在对用户的某些属性或某些行为进行预测判断。例如根据一个用户的行为习惯判断该用户是男性还是女性，根据一个用户的消费习惯判断其对某商品的偏好程度。该类标签需要通过算法挖掘产生。

在项目工程实践中，一般统计类和规则类的标签即可以满足应用需求，开发中占有较大比例。机器学习挖掘类标签多用于预测场景，如判断用户性别是男是女，判断用户购买商品偏好、判断用户流失意向等。一般地机器学习标签开发周期较长，耗费开发成本较大，因此其开发所占比例较小。

标签命名方式-1

标签主题	标签类型	开发方式	是否互斥关系	用户维度
A：用户属性 B：用户行为 C：用户消费 D：风险控制	1：分类型 2：统计型	1：统计型 2：算法型	1：互斥关系 2：非互斥关系	C：cookieid U：userid

- **标签主题：**用于刻画属于那种类型的标签，如用户属性、用户行为、用户消费、风险控制等多种类型，可用A、B、C、D等字母表示各标签主题；
- **标签类型：**标签类型可划为分类型和统计型这两种类型，其中分类型用于刻画用户属于哪种类型，如是男是女、是否是会员、是否已流失等标签，统计型标签用于刻画统计用户的某些行为次数，如历史购买金额、优惠券使用次数、近30日登陆次数等标签，这类标签都需要对应一个用户相应行为的权重次数；
- **开发方式：**开发方式可分为统计型开发和算法型开发两大开发方式。其中统计型开发可直接从数据仓库中各主题表建模加工而成，算法型开发需要对数据做机器学习的算法处理得到相应的标签；
- **是否互斥标签：**对应同一级类目下（如一级标签、二级标签），各标签之间的关系是否为互斥，可将标签划分为互斥关系和非互斥关系。例如对于男、女标签就是互斥关系，同一个用户不是被打上男性标签就是女性标签，对于高活跃、中活跃、低活跃标签也是互斥关系；
- **用户维度：**用于刻画该标签是打在用户唯一标识（userid）上，还是打在用户使用的设备（cookieid）上。可用U、C等字母分别标识userid和cookieid维度。

标签命名方式-2

示例：

对于用户是男是女这个标签，标签主题是用户属性，标签类型属于分类型，开发方式为统计型，为互斥关系，用户维度为userid。这样给男性用户打上标签“A111U001_001”，女性用户打上标签“A111U001_002”，其中“A111U”为上面介绍的命名方式，“001”为一级标签的id，后面对于用户属性维度的其他一级标签可用“002”、“003”等方式追加命名，“_”后面的“001”和“002”为该一级标签下的标签明细，如果是划分高、中、低活跃用户的，对应一级标签下的明细可划分为“001”、“002”、“003”。

标签id	标签名称	标签汉语	序号	标签主题	一级标签id	一级标签
A111H008_001	important value user	重要价值用户	1	用户属性	8	RFM
A111H008_002	important development use	重要发展用户	2	用户属性	8	RFM
A111H008_003	important maintain user	重要保持用户	3	用户属性	8	RFM
A111H008_004	important detainment user	重要挽留用户	4	用户属性	8	RFM
A111H008_005	general value user	一般价值用户	5	用户属性	8	RFM
A111H008_006	general development user	一般发展用户	6	用户属性	8	RFM
A111H008_007	general maintain user	一般保持用户	7	用户属性	8	RFM
A111H008_008	general detainment user	一般挽留用户	8	用户属性	8	RFM

用户标签相关表结构设计—tag表（1）

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','','','');
```

用户标签相关表结构设计—tag表（2）

数据存储

```
hive> select * from dw.profile_tag_userid;
OK
A220U029_001    25083679        282                             20180421        user_install_days
A220U029_001    7306783 166                             20180421        user_install_days
A220U029_001    4212236 458                             20180421        user_install_days
A220U029_001    39730187        22                              20180421        user_install_days
A220U029_001    16254215        57                              20180421        user_install_days
A220U083_001    25083679        800.39                          20180421        userid_all_paid_money
A220U083_001    7306783 311.29                          20180421        userid_all_paid_money
A220U083_001    32171777        129.65                          20180421        userid_all_paid_money
A220U083_001    40382657        602.3                           20180421        userid_all_paid_money
A220U083_001    30765587        465.93                          20180421        userid_all_paid_money
Time taken: 0.713 seconds, Fetched: 10 row(s)
```

通过设置标签类型这个分区字段，可以同时向该表中插入一个用户的不同类型的标签

存储路径

hdfs://master:9000/root/hive/warehouse/dw.db/profile_tag_userid/data_date=20180421/tagtype=userid_all_paid_money

用户标签在userid和cookieid维度各做了一套，同理适用于cookieid维度的表

```
CREATE TABLE `dw.profile_tag_cookieid`(
`tagid` string COMMENT 'tagid',
`cookieid` string COMMENT 'cookieid',
`tagweight` string COMMENT '标签权重',
`reserve1` string COMMENT '预留1',
`reserve2` string COMMENT '预留2',
`reserve3` string COMMENT '预留3')
COMMENT 'tagid维度的用户cookie画像数据'
PARTITIONED BY ( `data_date` string COMMENT '数据日期', `tagtype` string COMMENT '标签主题分类')
```

总结： **可以建立时间分区+标签类型分区的tag表，用于插入每天用户相关的每一个标签到相应的分区下**

用户标签相关表结构设计—人员标签表（1）

表结构设计

```
CREATE TABLE `dw.profile_user_map_userid`(
`userid` string COMMENT 'userid',
`tagsmap` map<string,string> COMMENT 'tagsmap',
`reserve1` string COMMENT '预留1',
`reserve2` string COMMENT '预留2')
COMMENT 'userid 用户画像数据'
PARTITIONED BY (`data_date` string COMMENT '数据日期')
```

数据存储

```
    > select * from dw.profile_user_map_userid;
OK
16254215        {"A220U029_001":"57"}                   20180421
25083679        {"A220U029_001":"282","A220U083_001":"800.39"}                  20180421
30765587        {"A220U083_001":"465.93"}               20180421
32171777        {"A220U083_001":"129.65"}               20180421
39730187        {"A220U029_001":"22"}                   20180421
40382657        {"A220U083_001":"602.3"}                20180421
4212236 {"A220U029_001":"458"}                  20180421
7306783 {"A220U029_001":"166","A220U083_001":"311.29"}                  20180421
Time taken: 0.434 seconds, Fetched: 8 row(s)
```

这里将一个用户身上的所有标签进行聚合

用户标签的聚合在userid和cookieid维度各做了一套，同理适用于cookieid维度的表

```
CREATE TABLE `dw.profile_user_map_cookieid`(
`cookieid` string COMMENT 'tagid',
`tagsmap` map<string,string> COMMENT 'cookieid',
`reserve1` string COMMENT '预留1',
`reserve2` string COMMENT '预留2')
COMMENT 'cookie 用户画像数据'
PARTITIONED BY (`data_date` string COMMENT '数据日期')
```

用户标签相关表结构设计—人员标签表（2）

总结： **从dw.profile_tag_userid 到 dw.profile_user_map_userid，实际是将同一个用户的多个标签聚合在一起。在tag表中，通过设置标签类型这一分区字段的形式，将每个用户身上的多个标签存储在了不同的分区下面。通过tag-map表，将一个用户的全部标签聚合在了一起。**

标签聚合的执行命令

```
insert overwrite table dw.profile_user_map_userid  partition(data_date="20180910")
select userid,
       str_to_map(concat_ws(',',collect_set(concat(tagid,':',tagweight)))) as tagsmap,
       '',
       ''
  from dw.profile_tag_userid
 where data_date="20180910"
group by userid
```

标签聚合的执行过程

```
    > insert overwrite table dw.profile_user_map_userid  partition(data_date="20180421") select userid,str_to_map(concat_ws(',',collect_set(concat(t
agid,':',tagweight)))) as tagsmap,'','' from dw.profile_tag_userid where data_date="20180421" group by userid;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark
, tez) or using Hive 1.X releases.
Query ID = root_20180924001127_53cbe0ef-2e53-4ff1-8390-a278bf085333
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1536058452188_0011, Tracking URL = http://master:8088/proxy/application_1536058452188_0011/
Kill Command = /root/data/hadoop-2.7.5/bin/hadoop job  -kill job_1536058452188_0011
Hadoop job information for Stage-1: number of mappers: 2; number of reducers: 1
2018-09-24 00:11:43,927 Stage-1 map = 0%,  reduce = 0%
2018-09-24 00:11:57,480 Stage-1 map = 50%,  reduce = 0%, Cumulative CPU 4.13 sec
2018-09-24 00:12:03,454 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 8.87 sec
2018-09-24 00:12:12,047 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 12.46 sec
MapReduce Total cumulative CPU time: 12 seconds 460 msec
Ended Job = job_1536058452188_0011
Loading data to table dw.profile_user_map_userid partition (data_date=20180421)
```

用户人群相关表设计

```
CREATE TABLE `dw.profile_usergroup_tag`(
`userid` string,
`tagsmap` map<string,string>,
`reserve1` string,
`reserve2` string)
PARTITIONED BY (`data_date` string, `target` string)
```

用户人群表主要记录了用户的id、人群名称id以及推送到的业务系统

```
val tablename="dw.profile_usergroup_tag"

val dataset = spark.sql(
  s"""
     |    select t1.userid,t2.order_sn,t3.tel
     |      from ${tablename} t1
     | inner join dw.paid_order_fact t2
     |         on t1.userid = t2.user_id
     | inner join dw.order_user_info t3
     |         on t2.order_id = t3.order_id
     | where t1.data_date = '${data_date}'
     |   and t1.target = "100000207486"
     | group by t1.userid,t3.tel
     | having t3.tel <> ''
  """.stripMargin)
```

这里通过人群表t1 join了t2订单表，得到用户的订单编号，然后通过t2订单表join用户收货信息表t3，得到用户的 手机号；

这样通过圈定人群可以得到一批运营用户及他们的手机号，可以推送给外呼中心进行外呼操作

画像标签的元数据管理

元数据信息读取的数据

tag_id	tag_chinese_name	tag_theme	level_1_id	level_1_name	level_2_id	level_2_name	tag_type	develop_type	update_type	developer	sub_developer	idtype
A121H013_002	积分试用会员	用户属性	13	是否付费会员(CO	(Null)		1	2	日	AAA	BBB	cookieid
A121H013_003	赠送会员	用户属性	13	是否付费会员(CO	(Null)		1	2	日	AAA	BBB	cookieid
A121H013_004	补偿会员	用户属性	13	是否付费会员(CO	(Null)		1	2	日	AAA	BBB	cookieid
A121H013_005	历史会员	用户属性	13	是否付费会员(CO	(Null)		1	2	日	AAA	BBB	cookieid
A121H013_006	非会员	用户属性	13	是否付费会员(CO	(Null)		1	2	日	AAA	BBB	cookieid
A121H030_001	未注册	用户属性	10030	注册状态	(Null)		1	2	日	AAA	BBB	cookieid
A121H030_002	已注册	用户属性	10030	注册状态	(Null)		1	2	日	AAA	BBB	cookieid
B120H008_003	首单新人价商品	用户行为	8	首单营销方式(CO	(Null)		1	2	日	AAA	BBB	cookieid
B120H008_004	首单优惠券	用户行为	8	首单营销方式(CO	(Null)		1	2	日	AAA	BBB	cookieid
B120H008_005	首单新人专享优惠商	用户行为	8	首单营销方式(CO	(Null)		1	2	日	AAA	BBB	cookieid
B120H008_006	首单红包	用户行为	8	首单营销方式(CO	(Null)		1	2	日	AAA	BBB	cookieid
B120U008_001	首单正常购买	用户行为	2008	首单营销方式	(Null)		1	2	日	AAA	BBB	userid
B120U008_002	首单免费礼物	用户行为	2008	首单营销方式	(Null)		1	2	日	AAA	BBB	userid
B120U008_003	首单新人价商品	用户行为	2008	首单营销方式	(Null)		1	2	日	AAA	BBB	userid
B120U008_004	首单优惠券	用户行为	2008	首单营销方式	(Null)		1	2	日	AAA	BBB	userid

标签的元数据维护着标签的id、名称、主题、一级二级分类、标签描述等信息

结果集校验

Hive作业完成后，每个标签量级/覆盖率的监控

id	tagid	date	tag_count	tag_total_per	tag_dau_per	t
86	A121H002_003	2018-04-21	581391334	0.0193901	0.030789	
87	A111H001_002	2018-04-21	508867844	0.169713	0.723889	
89	A121H002_002	2018-04-21	1460253600	0.487011	0.657123	
90	B220H026_001	2018-04-21	283956377	0.947052	1.00701	
91	A121H031_002	2018-04-21	25371635	0.846173	0.681642	
92	A121H030_002	2018-04-21	996368533	0.3323	0.51521	
93	A121H030_001	2018-04-21	184326924	0.614752	0.491804	
94	A111H041_002	2018-04-21	315144932	0.105104	0.129659	
95	A111H041_001	2018-04-21	319472833	0.106548	0.0742948	
96	A220H029_001	2018-04-21	28396377	0.947052	1.00701	
97	A121H002_009	2018-04-21	238660590	0.079596	0.0429698	

当日该标签覆盖的用户量

当日该标签覆盖的用户占当日活跃用户的比例

当日该标签与昨日相比的波动比例

Hive同步到hbase后，数据校验

date	service_type	hive_count	hbase_count
2018-09-09	addresstag	[illegible]	[illegible]
2018-09-08	userprofile	[illegible]	[illegible]
2018-09-08	[illegible]	[illegible]	[illegible]
2018-09-08	addresstag	[illegible]	[illegible]
2018-09-07	userprofile	[illegible]	[illegible]
2018-09-07	usergroup2hba[illegible]r	[illegible]	[illegible]
2018-09-07	usergroup2hba[illegible]r	[illegible]	[illegible]
2018-09-07	E[illegible]	[illegible]	[illegible]

存放数据校验标志位

process_date	stage	state	is_online
2018-08-17	Cluster	0	2
2018-08-18	Cluster	1	2
2018-08-19	Cluster	0	2
2018-08-20	Cluster	0	2
2018-08-21	Cluster	0	2
2018-08-22	Cluster	0	2
2018-08-23	Cluster	0	2
2018-08-24	Cluster	0	2
2018-08-25	Cluster	0	2

放置标志位用于判断某些任务是否需要继续执行

同步到业务系统中

在用户画像产品化章节中，当运营人员圈定用户后，需要将该批待运营的用户推送到对应的业务系统中；

不同的业务系统读取的数据库不全都一样，比如说“客服系统”中读取的数据库是关系型数据库

这里，通过sqoop把hive中的数据同步到对应的MySQL库表中

```
os.system("sqoop export --connect jdbc:mysql://xxx.xx.23.142:3307/userprofile --username userprofile
--password userprofile --table tag_tmp_userid --export-dir
hdfs://master:9000/user/hive/warehouse/dw.db/dw_profile_user_tag_service/data_date=20180901/business=push
--input-fields-terminated-by '\001'")
```

同步后的存储结果

cookie_user_id	tagsmap	reserve	reserve1
10005353	{'A121U013_0_006': '', 'B121U031_0_002': '10', 'A120U014_0_001': '3'}		
10005433	{'B121U031_0_001': '1', 'A121U013_0_006': '', 'A120U014_0_001': '2'}		
10005439	{'A121U013_0_005': '', 'B121U031_0_002': '6', 'A120U014_0_001': '2'}		
10005447	{'A121U013_0_006': '', 'B121U031_0_002': '4', 'A120U015_0_001': '0.4', 'A120U014_0_001':		
10005493	{'B121U031_0_002': '2', 'A120U014_0_001': '2', 'A121U013_0_006': ''}		
10005499	{'B121U031_0_002': '3', 'A120U014_0_001': '3', 'A121U013_0_006': ''}		
10005671	{'A121U013_0_006': '', 'B121U031_0_002': '1', 'A120U014_0_001': '2'}		
10005707	{'A121U013_0_006': '', 'B121U031_0_002': '1', 'A120U014_0_001': '2'}		
10005911	{'A121U013_0_006': '', 'B121U031_0_002': '1', 'A120U014_0_001': '2'}		
10005941	{'A121U013_0_005': '', 'B121U031_0_002': '2', 'A120U014_0_001': '2'}		
10005955	{'A121U013_0_004': '', 'B121U031_0_002': '6', 'A120U014_0_001': '2'}		
10005983	{'A121U013_0_004': '', 'B121U031_0_002': '10', 'A120U014_0_001': '3'}		
10006011	{'B121U031_0_001': '1', 'A121U013_0_005': '', 'A120U015_0_001': '0.5555555555555556'}		

这里可以写一个Python脚本，把对应的hive数据同步到MySQL库表下面

Elasticsearch简介

Elasticsearch是一个开源的分布式全文检索引擎，可以近乎实时地存储、检索数据。而且扩展性很好，可以扩展到上百台服务器，处理PB级别数据。对于用户标签查询、用户人群计算、用户群多维透视分析这类对响应时间要求较高的场景，也同样可以考虑选用Elasticsearch进行存储

MySQL	Elastic Search
Database	Index
Table	Type
Row	Document
Column	Field
Schema	Mapping
Index	Everything is indexed
SQL	Query DSL
SELECT * FROM table …	GET http://…
UPDATE table SET …	PUT http://…

数据插入Elasticsearch

userid	tagsmap	data_date
30000591	{"B220U102_001":"153","C120U033_6_006":"1","B220U025_001":"163","C120U033_6_003":"2"}	20181220
20000723	{"D220U004_001":"0.666666","B220U072_001":"0.0","B220U066_001":"497"}	20181220
20000761	{"A121U013_006":"","A220U029_001":"754","A221U024_004":"","A121U031_002":"","A220U091_001":"753"}	20181220
30000835	{"A121U013_006":"","A121U031_002":"","A220U091_001":"753"}	20181220
30000989	{"D220U004_001":"0.0","A111U041_002":"","A220U054_001":"1","A221U024_004":"","B121U101_002":""}	20181220


```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 83500000,
    "max_score": 0,
    "hits": []
  }
}
```



```
{
  "took": 744,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 100000000,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "tagcounts": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": -99,
          "doc_count": 2500000
        }
      ]
    }
  }
}
```

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

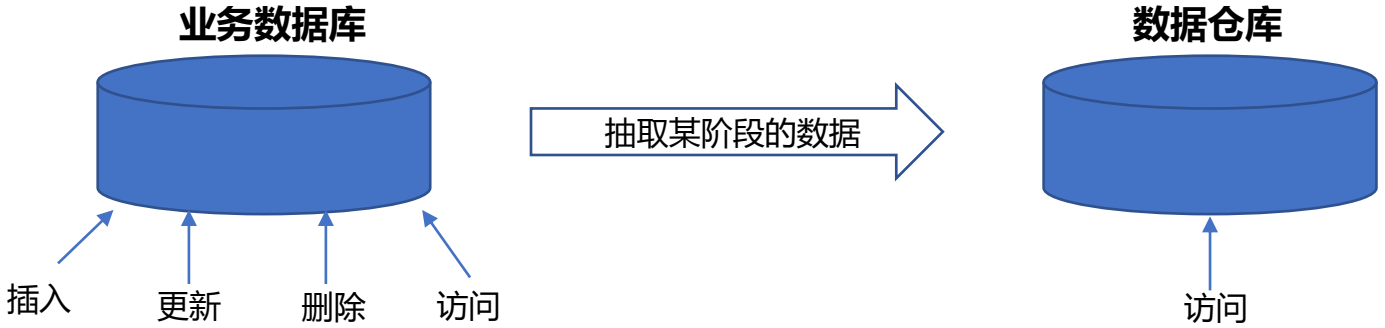
（2）集成

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与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；

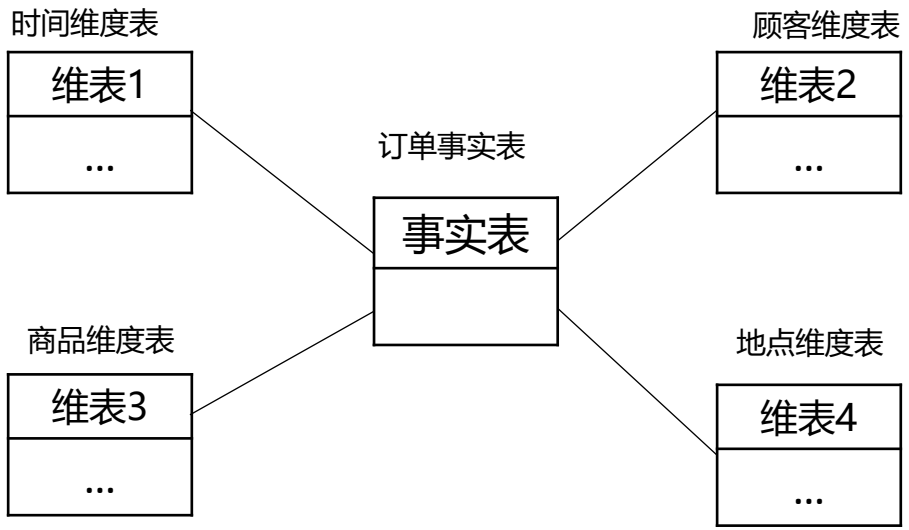
常见多维数据模型—星形模型

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

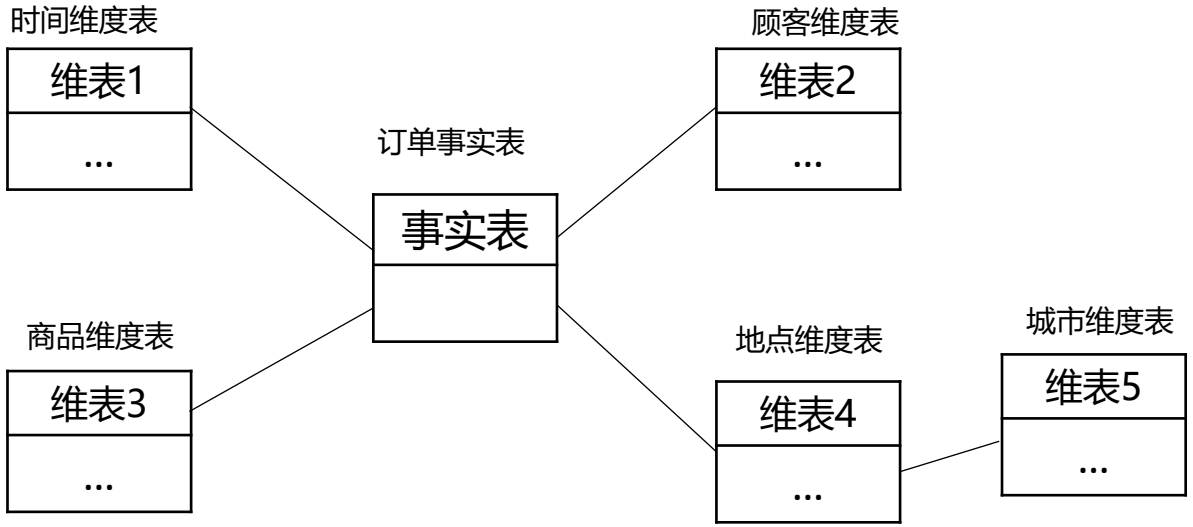


常见多维数据模型—雪花模型

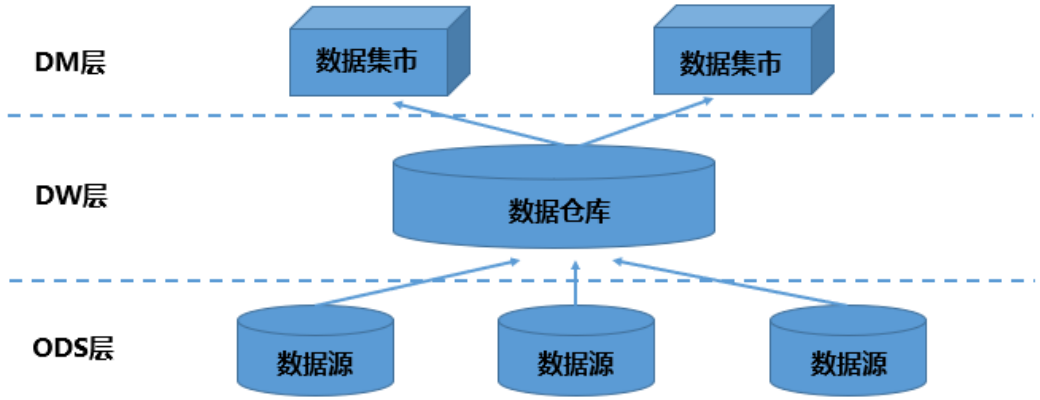
雪花模型是对星形模型的扩展，每一个维表都可以向外连接多个详细类别表。除了具有星形模式中维表的功能外，还连接对事实表进行详细描述的维度，可进一步细化查看数据的粒度

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

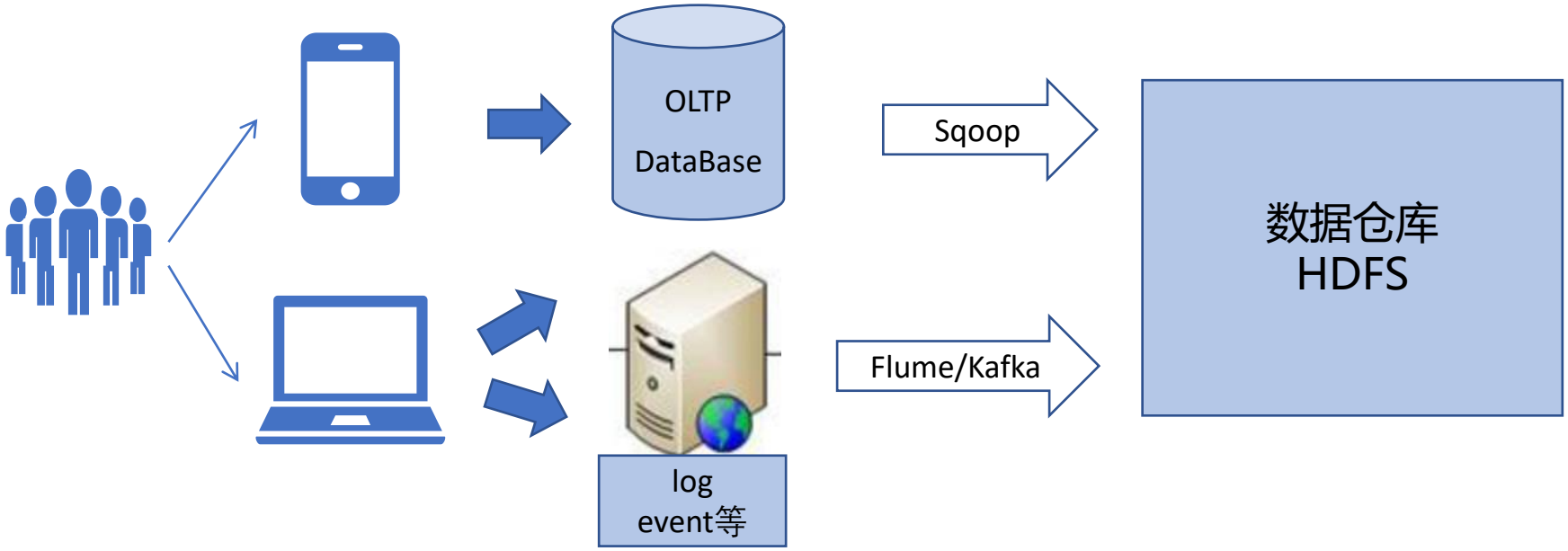


数据仓库分层



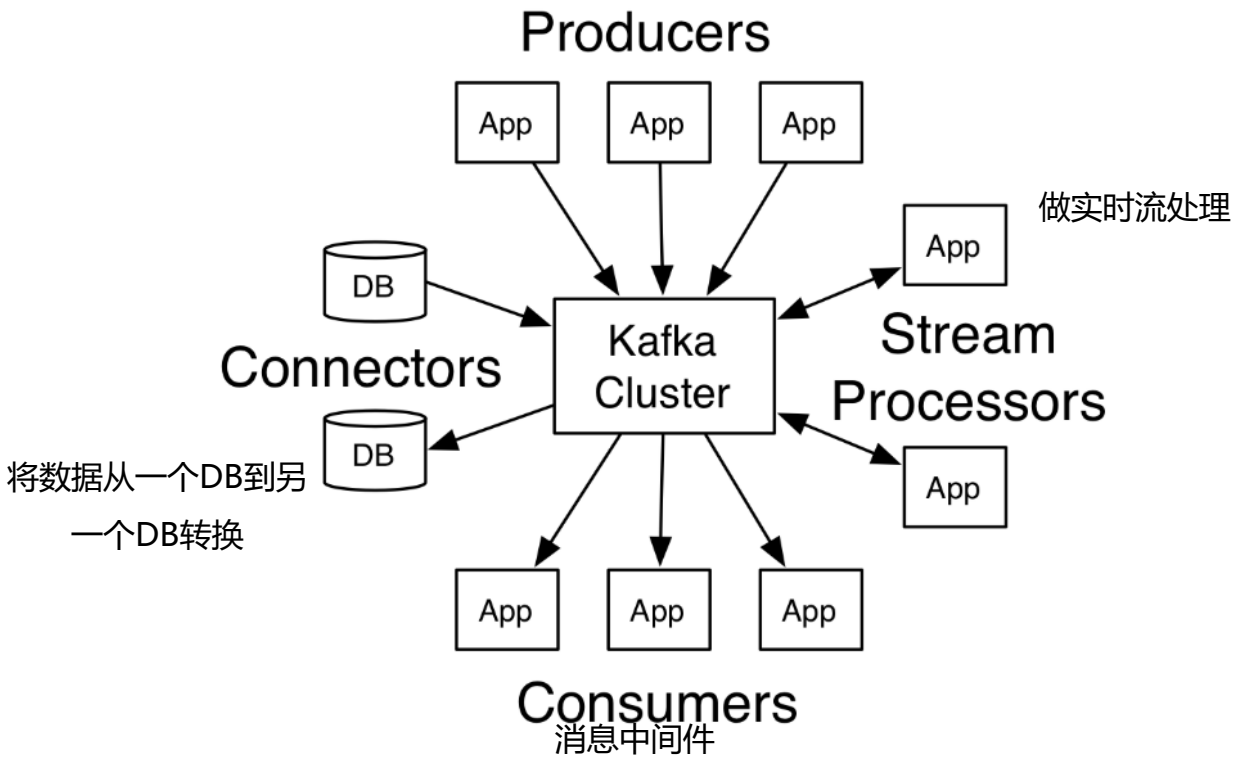
- 清晰数据结构：每一个数据分层都有它的作用域，这样我们在使用表的时候能更方便地定位和理解
- 脏数据清洗：屏蔽原始数据的异常
- 屏蔽业务影响：不必改一次业务就需要重新接入数据
- 数据血缘追踪：简单来讲可以这样理解，我们最终给业务呈现的是能直接使用的一张业务表，但是它的来源有很多，如果有一张来源表出问题了，我们希望能够快速准确地定位到问题，并清楚它的危害范围。
- 减少重复开发：规范数据分层，开发一些通用的中间层数据，能够减少极大的重复计算。
- 把复杂问题简单化。将一个复杂的任务分解成多个步骤来完成，每一层只处理单一的步骤，比较简单和容易理解。便于维护数据的准确性，当数据出现问题之后，可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

画像数据和数据仓库的关系



- 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。

Kafka基本介绍

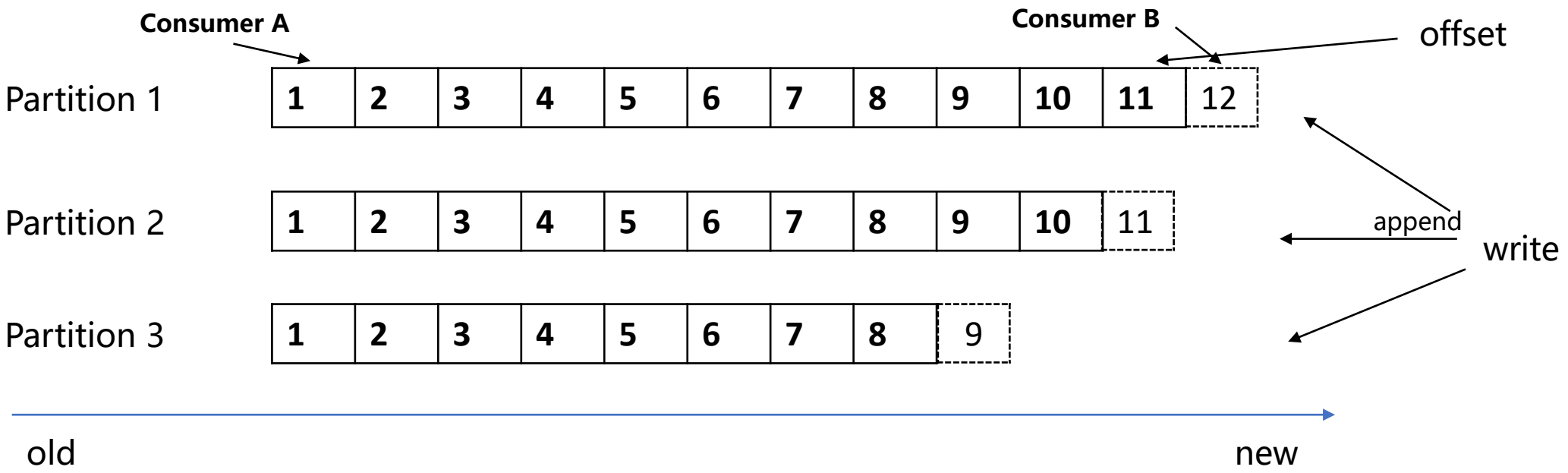


- **producer**：产生信息的主体；
- **consumer**：消费producer产生信息的主体；
- **broker**：消息处理节点，多个broker组成kafka集群；
- topic：是数据主题，是数据记录发布的地方,可以用来区分业务系统；
- **partition**：是topic的分组，每个partition都是一个有序队列
- offset：用于定位消费者在每个partition中消费到的位置

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

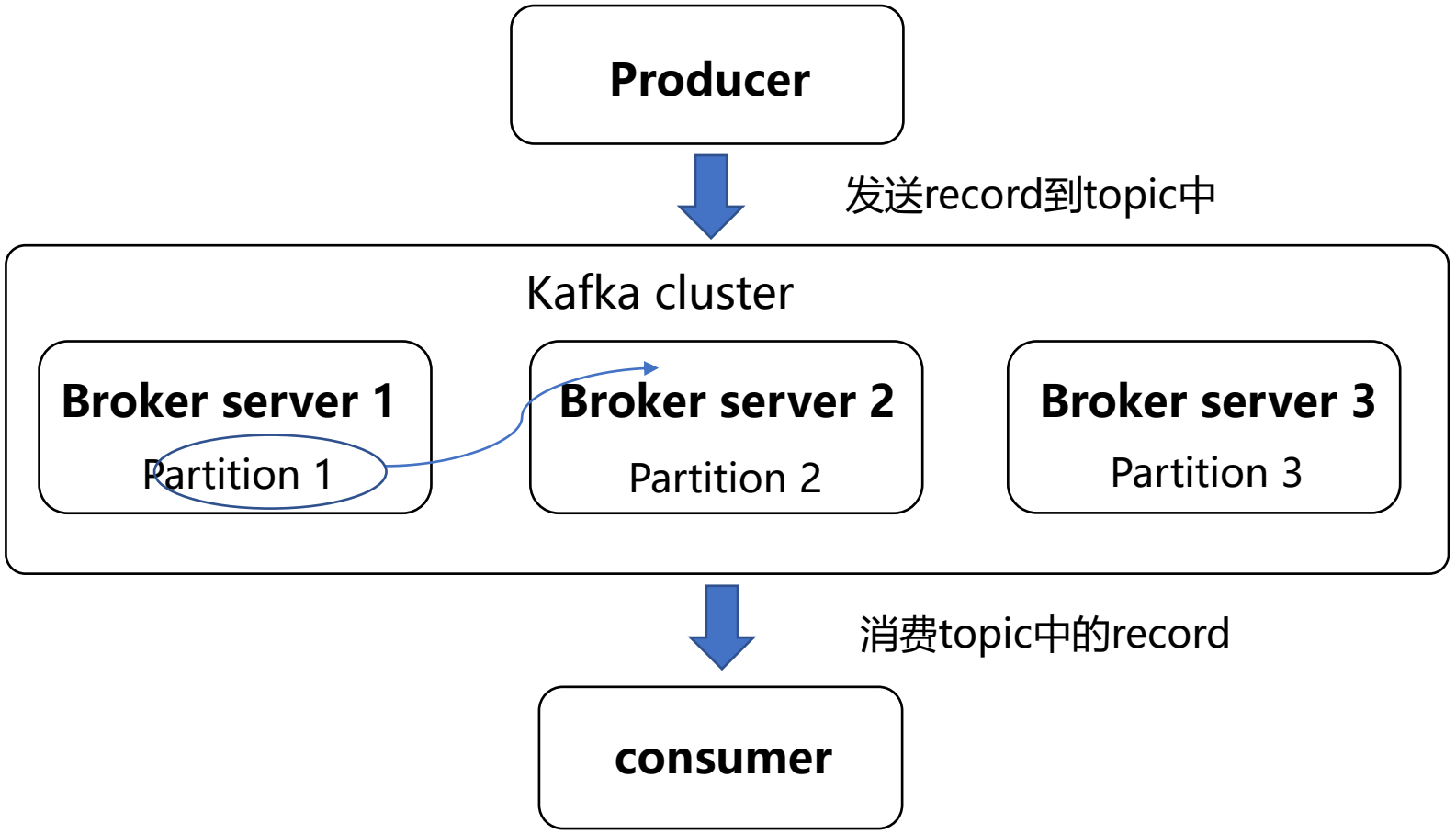
http://spark.apachecn.org/docs/cn/2.2.0/streaming-kafka-0-8-integration.html

Partition和offset（1）



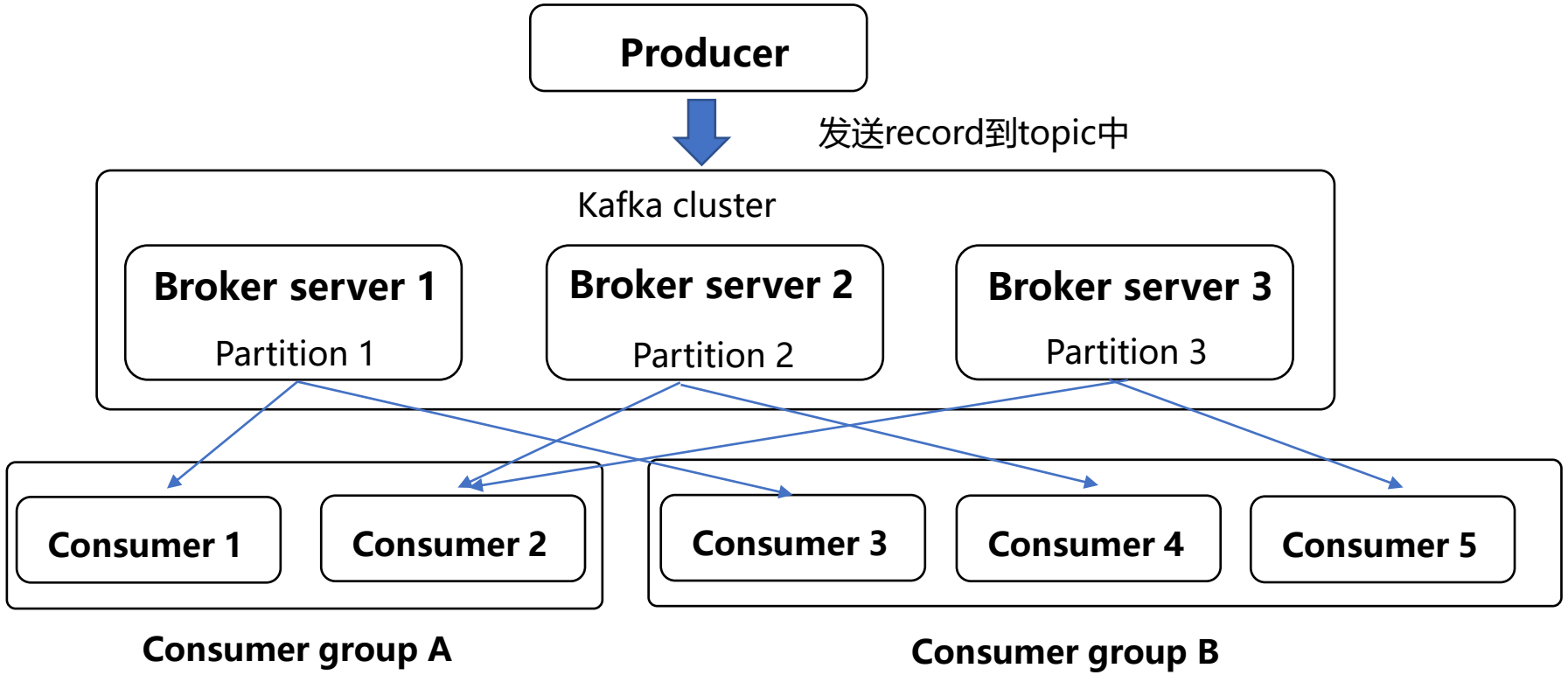
- Topic在broker里面对应着一系列的消息数据，这些消息数据存储在kafka磁盘里面。默认存储7天；
- 对于发过来的消息，是写入到哪个partition上去，是有规则的；
- 每个partition都是一个有序的、不变的record序列。接收到的record追加到这个序列中；
- Offset是偏移量，标识每条消息的唯一标识；
- 一个topic对应的数据是分了好几个区，这些分区都是分布式地分布在多个broker server上，每个分区又备份几份在其他broker上，这样保证了集群的高可用性；
- Offset是consumer控制的，所以consumer可以按照不同需求消费任何位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

Partition和offset（2）



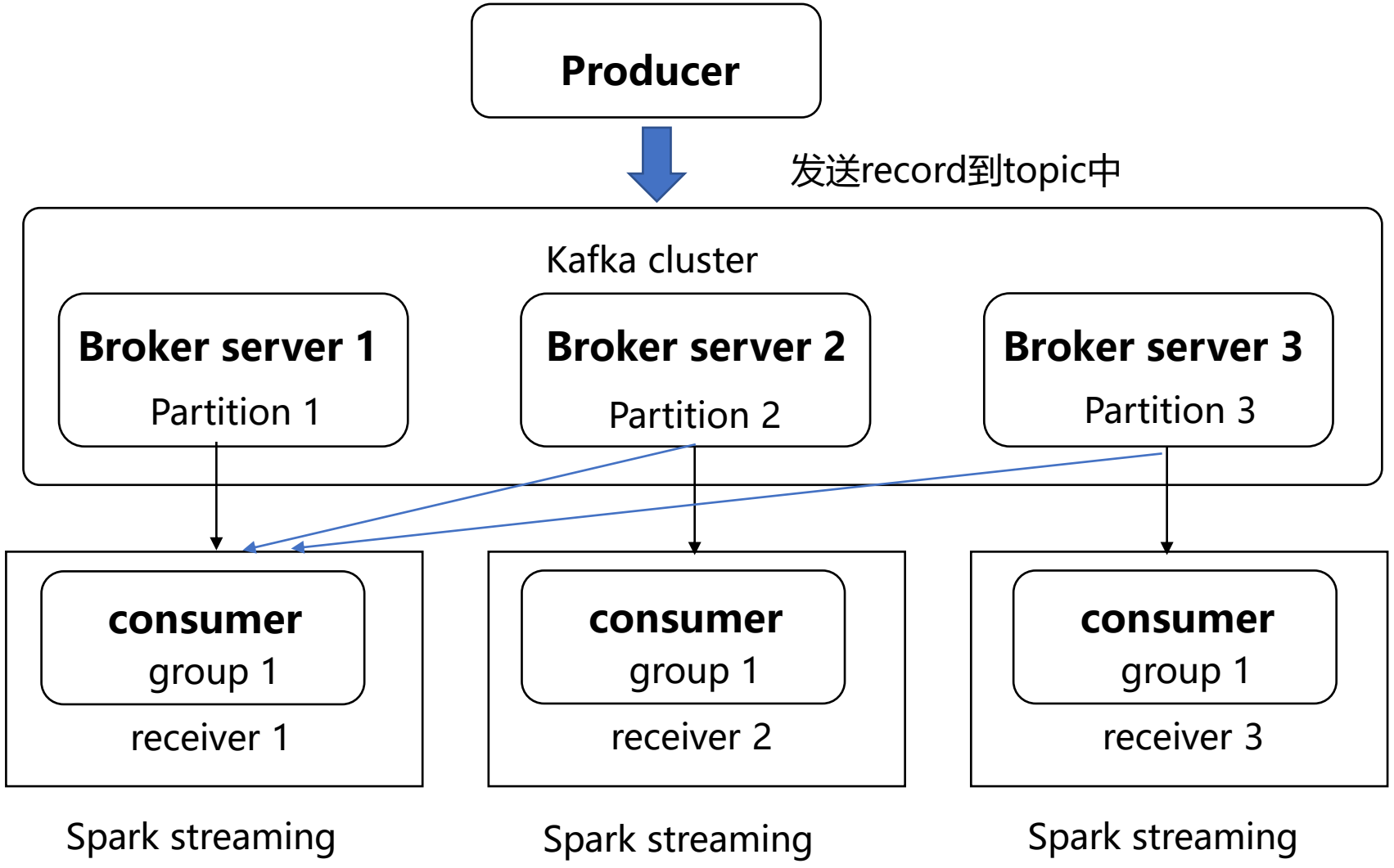
- Producer向topic发消息时，topic以分区形式存在，topic以分区的形式存在broker server中；
- Topic的partition是分布式地分布在多个broker server中；
- 多个接收器接收发送来的数据，这样的吞吐量较高；
- 每个partition都备份到其他broker server中去，有好几个备份。一个partition所在的broker挂掉了，其他上broker上对应的partition同样可对外提供服务，这样partition数据的高可用性得到了保证；

Consumer group



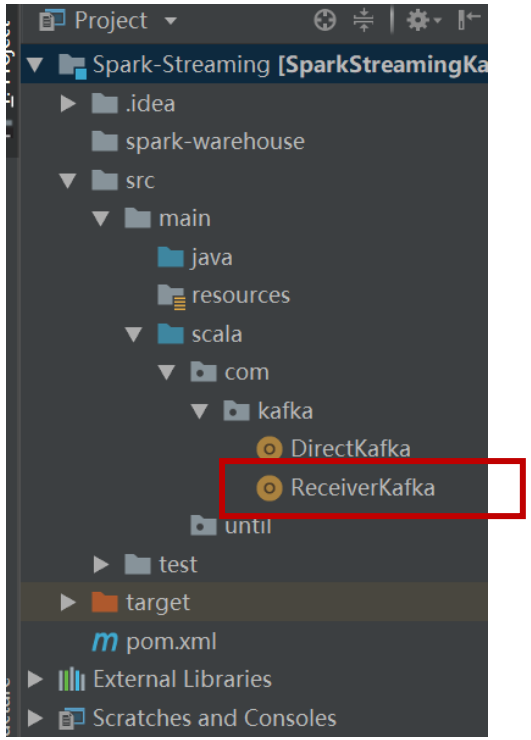
- **Consumer消费消息时候处理需要指定topic外，还需要指定这个consumer属于哪个consumer group。每个consumer group消费topic下的所有partition数据，**每个consumer消费topic中的partitions数据时候是按offset来进行的；
- 每一个consumer都被归为一个consumer group，同时一个consumer group可以包含一个或多个consumer；
- 一个topic中一条record会被所有订阅这个topic的consumer gourp消费。即每一个consumer实例都属于一个consumer 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***

Receiver模式



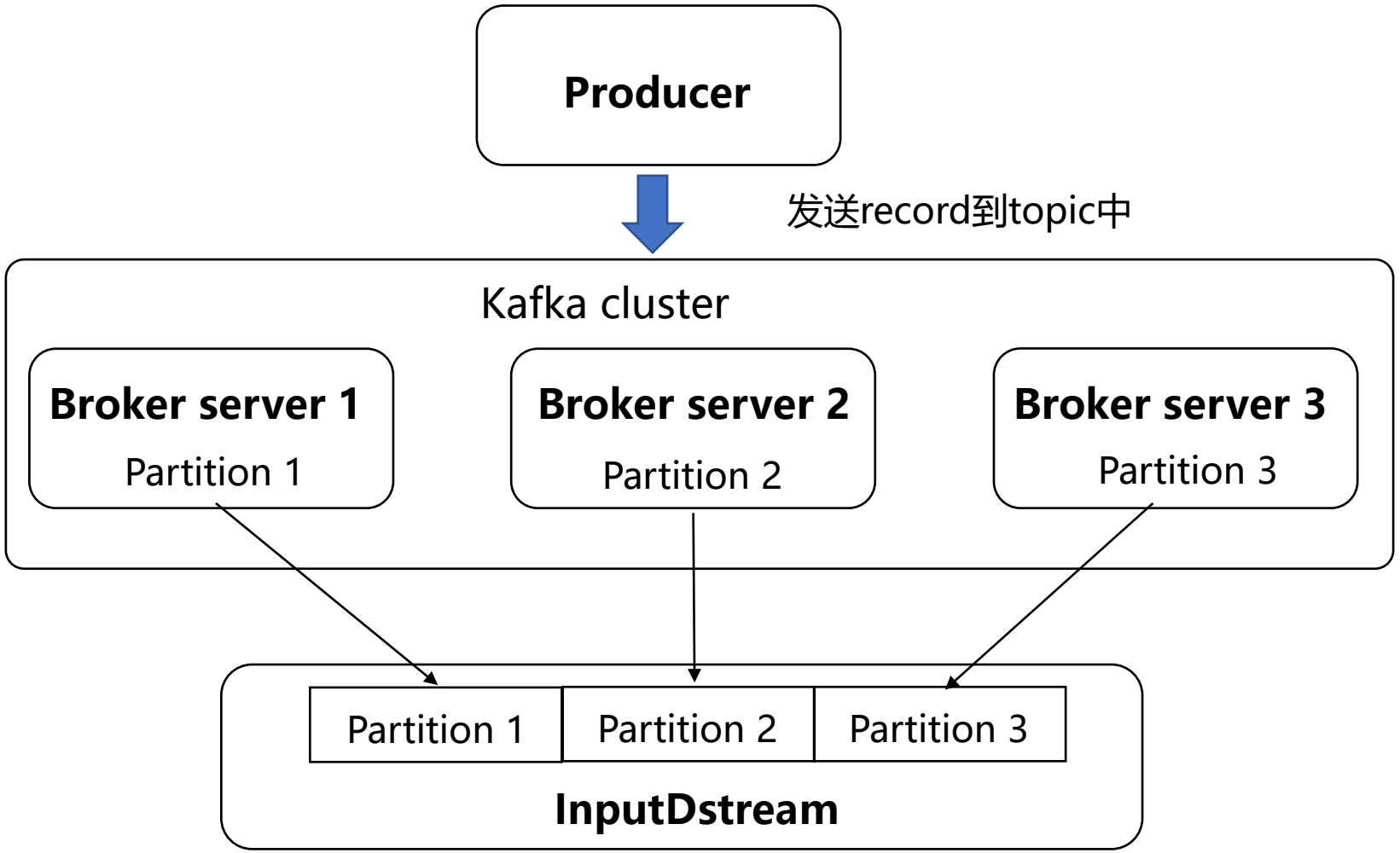
- Receiver实时拉取kafka里面的数据；
- 一个receiver接收数据不过来时，再起其他receiver接收，他们同属于一个consumer group，这样提高了streaming程序的吞吐量；
- kafka中的topic是以partition的方式存在的，Spark中的partition和kafka中的partition并不是相关的，如果我们加大每个topic的partition数量，仅仅是增加线程来处理由单一Receiver消费的主题。但是这并没有增加Spark在处理数据上的并行度

Receiver代码示例



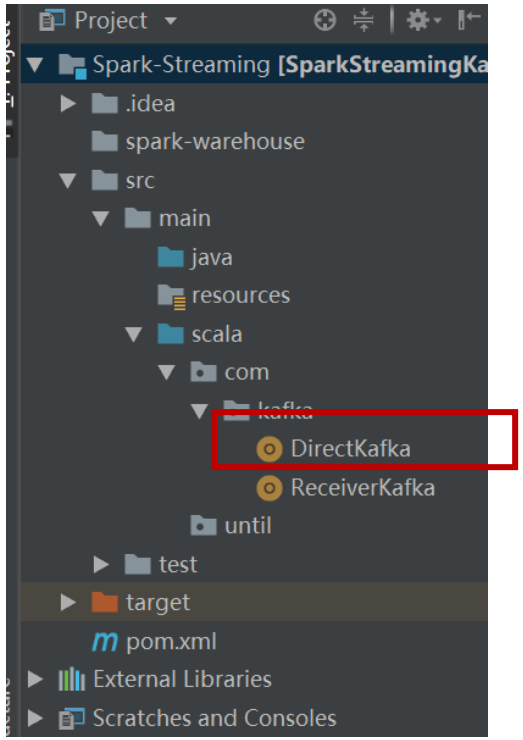

```scala
val sparkConf = new SparkConf().setAppName("SparkStreaming-ReceivertKafka")
val sc = new SparkContext(sparkConf)
val Array(zkQuorum, group, topics, numThreads) = args
val ssc = new StreamingContext(sc, Seconds(2))
val topicMap = topics.split( regex = ",").map((_, numThreads.toInt)).toMap
val 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

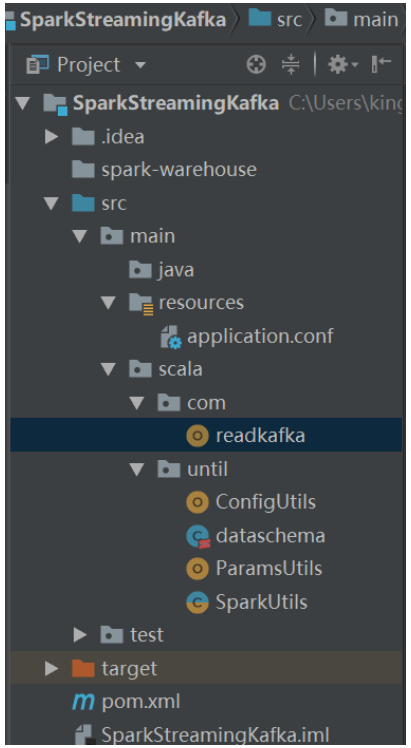
Exactly once	只被处理一次	sparkstreaming自己跟踪消费的offset，是直接消费存储在kafka中的数据

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

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

数据倾斜原因

常见表现：在hive中 map阶段早就跑完了，reduce阶段一直卡在99%。很大情况是发生了数据倾斜，整个任务在等某个节点跑完。

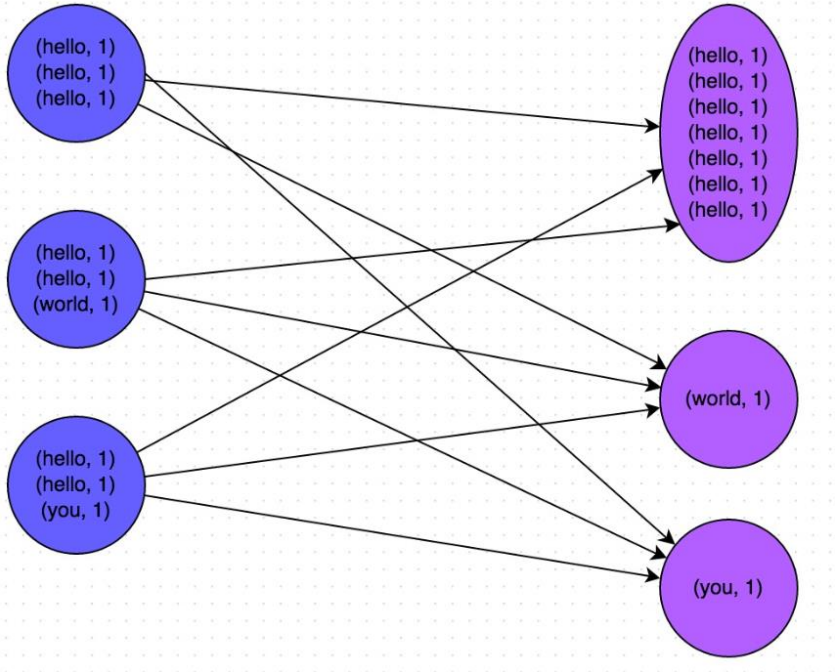
在spark中大部分的task执行的特别快，剩下的一些task执行的特别慢，要几分钟或几十分钟才执行完一个task

Hive中大表join的时候，容易产生数据倾斜问题，spark中产生shuffle类算子的操作，groupbykey、reducebykey、join等操作会引起数据倾斜。

通过stage去定位

数据倾斜原因：

在进行shuffle的时候，*必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理*，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时



数据倾斜解决方案

解决方法1：直接过滤掉那些引起倾斜的key

例如
```
select key1,count(*) as num_1
  from dw.table_a
group by key1
order by num_1 desc limit 20
```

```
select key2,count(*) as num_2
  from dw.table_b
group by key2
order by num_2 desc limit 20
```

比如说，总共有100万个key。只有2 个key，是数据量达到10 万的。其他所有的key，对应的数量都是几十，这样join后会引起倾斜。这个时候，自己可以去取舍，如果业务和需求可以理解和接受的话，在从hive 表查询源数据的时候，直接在sql 中***用where 条件，过滤掉某几个key***。那么这几个原先有大量数据，会导致数据倾斜的key，被过滤掉之后，那么在的spark作业中，自然就不会发生数据倾斜了。

解决方法2：Hive ETL做处理

导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且应用中需要频繁使用Spark对Hive表执行分析操作时，可以使用Hive ETL去做一个预处理

实现方式

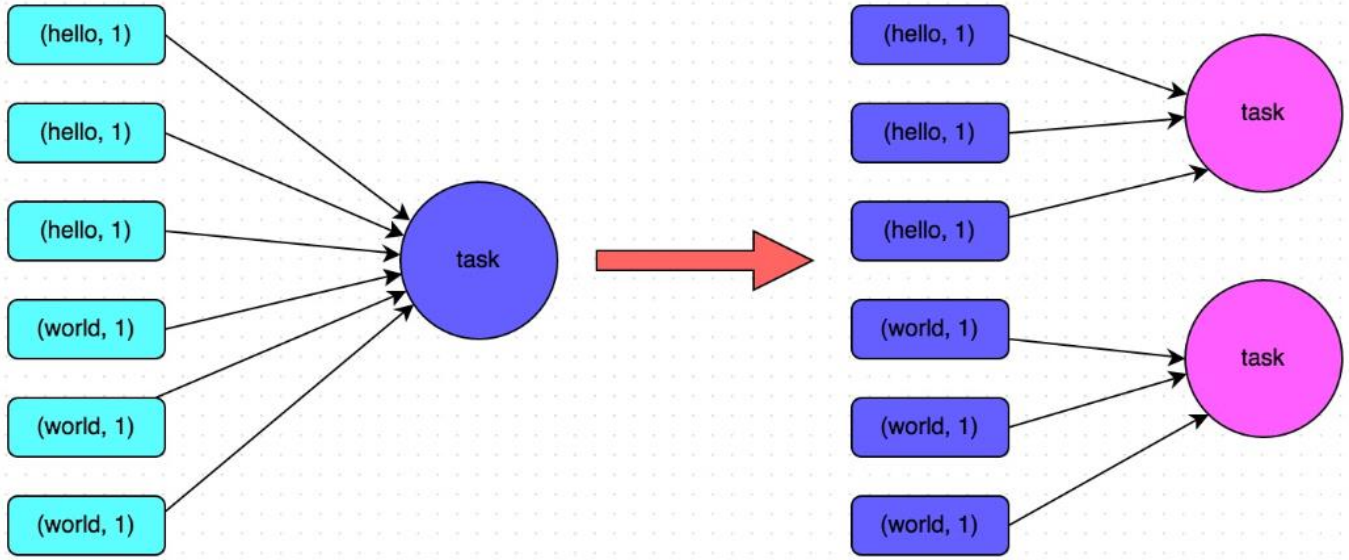
通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join，然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中

数据倾斜解决方案

解决方法3：提高shuffle操作并行度

在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小

原理：增加shuffle read task的数量，可以***让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据***。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了



参考文章：美团技术团队博客

合并小文件

- hadoop fs –ls /文件地址 ： 可以查看Hive表中每个数据文件的大小。小文件一般都几k、几十k的。
- HDFS用于存储大数据的文件，如果Hive中存在过多的小文件会给namenode带来较大的性能压力。同时小文件过多时会影响spark 中job的执行。为了提高namenode的使用效率，在向hdfs加载文件时需要提前对小文件进行合并；
- Spark将job转换成多个task，从hive中拉取数据，对于每个小文件也要分配一个task去处理，每个task只处理很少的数据，这样会起上万个task，非常影响性能；
- 处理大量小文件的速度远远小于处理同样大小的大文件速度，Task启动将耗费大量时间在启动task和释放task上。

为了防止生成小文件，在hive ETL的时候可以通过配置参数在MapReduce过程中合并小文件。

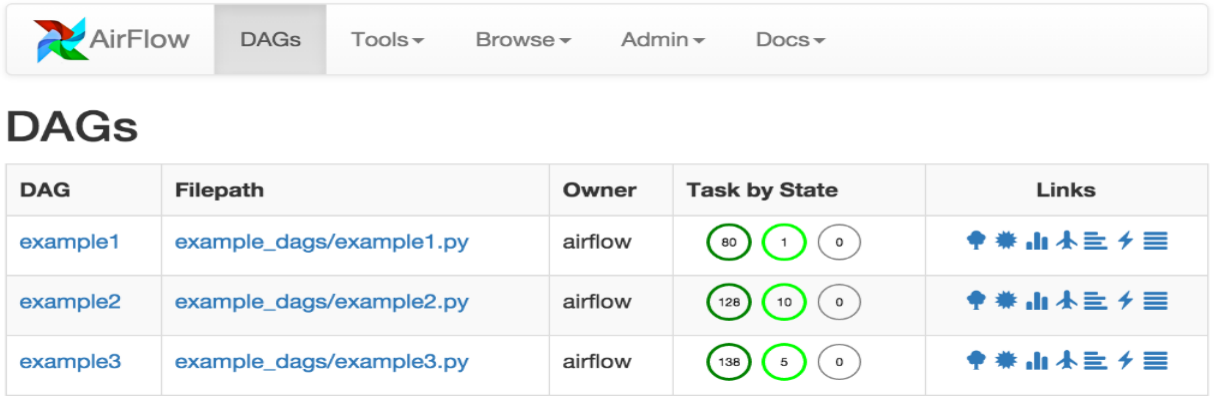
一般在对ods层日志数据进行处理时，如果小文件过多，需要重新ETL合并小文件再重新写入

输出合并

合并输出小文件，以减少输出文件的大小，可通过如下参数设置：

```
set  hive.merge.mapfiles=true;      // map only job结束时合并小文件
set  hive.merge.mapredfiles=true;     // 合并reduce输出的小文件
set  hive.merge.size.per.task=64000000;      //合并之后的每个文件大小64M
```

Airflow调度—基础概念



Airflow是Airbnb内部发起的一个工作流管理平台。使用Python编写实现的任务管理、调度、监控工作流平台。

Airflow的调度依赖于crontab命令，与crontab相比Airflow可以方便查看任务的执行状况（执行是否成功、执行时间、执行依赖等），可追踪任务历史执行情况，任务执行失败时可以收到邮件通知，查看错误日志。对于管理调度任务有很大的帮助。

Crontab命令管理调度的方式总结来看存在以下几方面的弊端：

1、在多任务调度执行的情况下，难以理清任务之间的依赖关系；

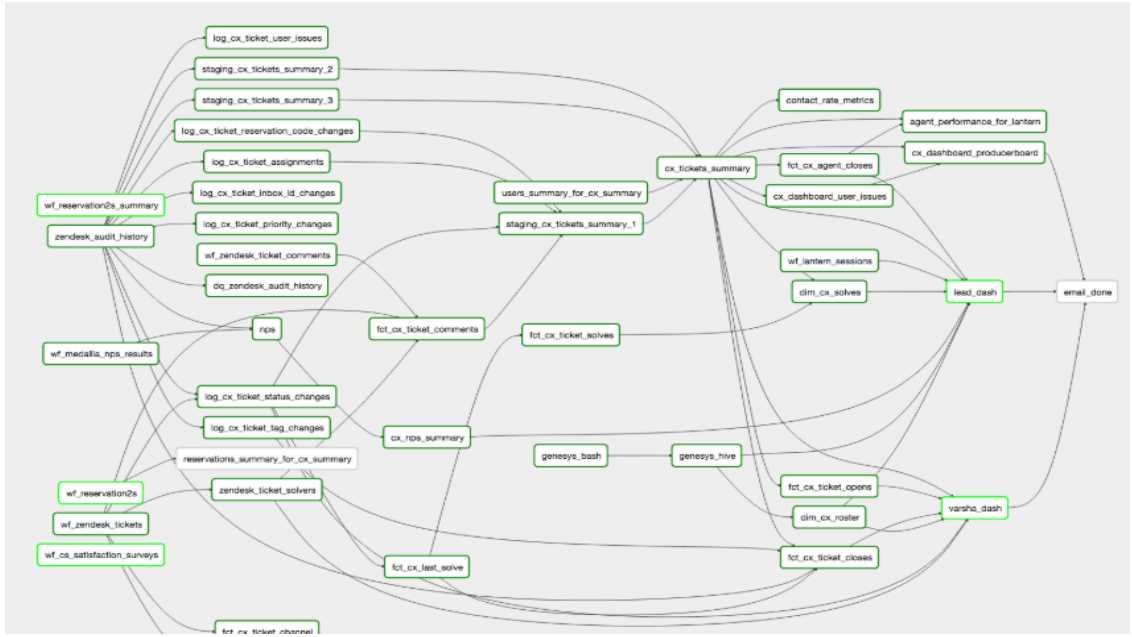
2、不便于查看当前执行到哪一个任务；

3、不便于查看调度流下每个任务执行的起止消耗时间，而这对于优化task作业是非常重要的；

4、不便于记录历史调度任务的执行情况，而这对于优化作业和排查错误是非常重要的；

5、执行任务失败时不便于查看执行日志，不方便定位报错的任务和接收错误告警邮件；

Airflow调度—基础概念

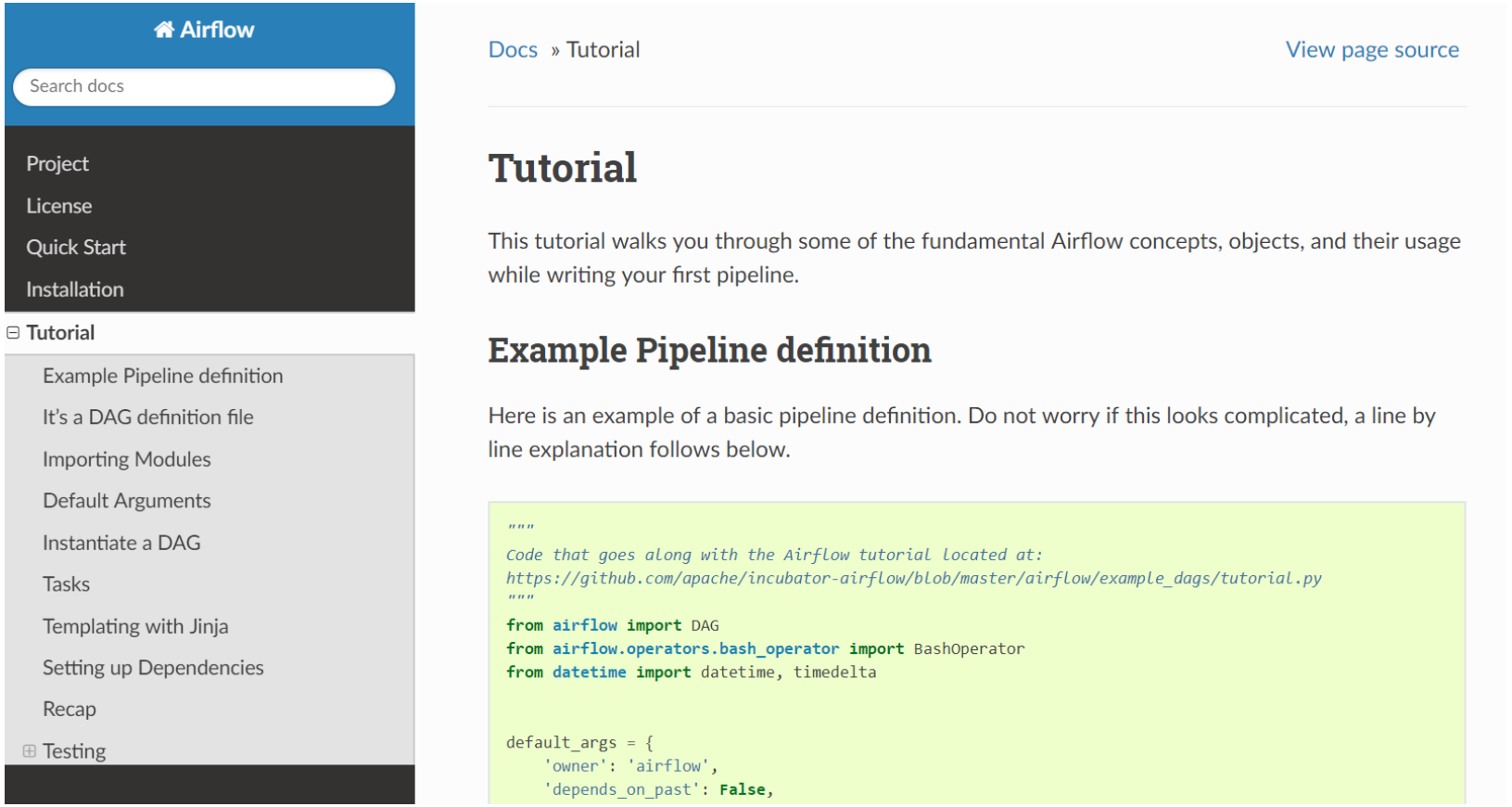


在介绍Airflow这个调度工具前先介绍几个相关的基础概念

- **DAG**：即有向无环图（Directed Acyclic Graph），DAG用于描述数据流的计算过程；
- **Operators**：描述了DAG中一个具体task要执行的任务，如BashOperator为执行一条bash命令，EmailOperator用于发送邮件，HTTPOperator用于发送HTTP请求，PythonOperator用于调用任意的Python函数；
- **Task**：是Operator的一个实例，也就是DAG中的一个节点；
- **Task Instance**：记录task的一次运行。Task Instance有自己的状态，包括“running”、“success”、“failed”、“skipped”、“up for retry”等；
- **Triggher Rules**：指task的触发条件；

Airflow官网

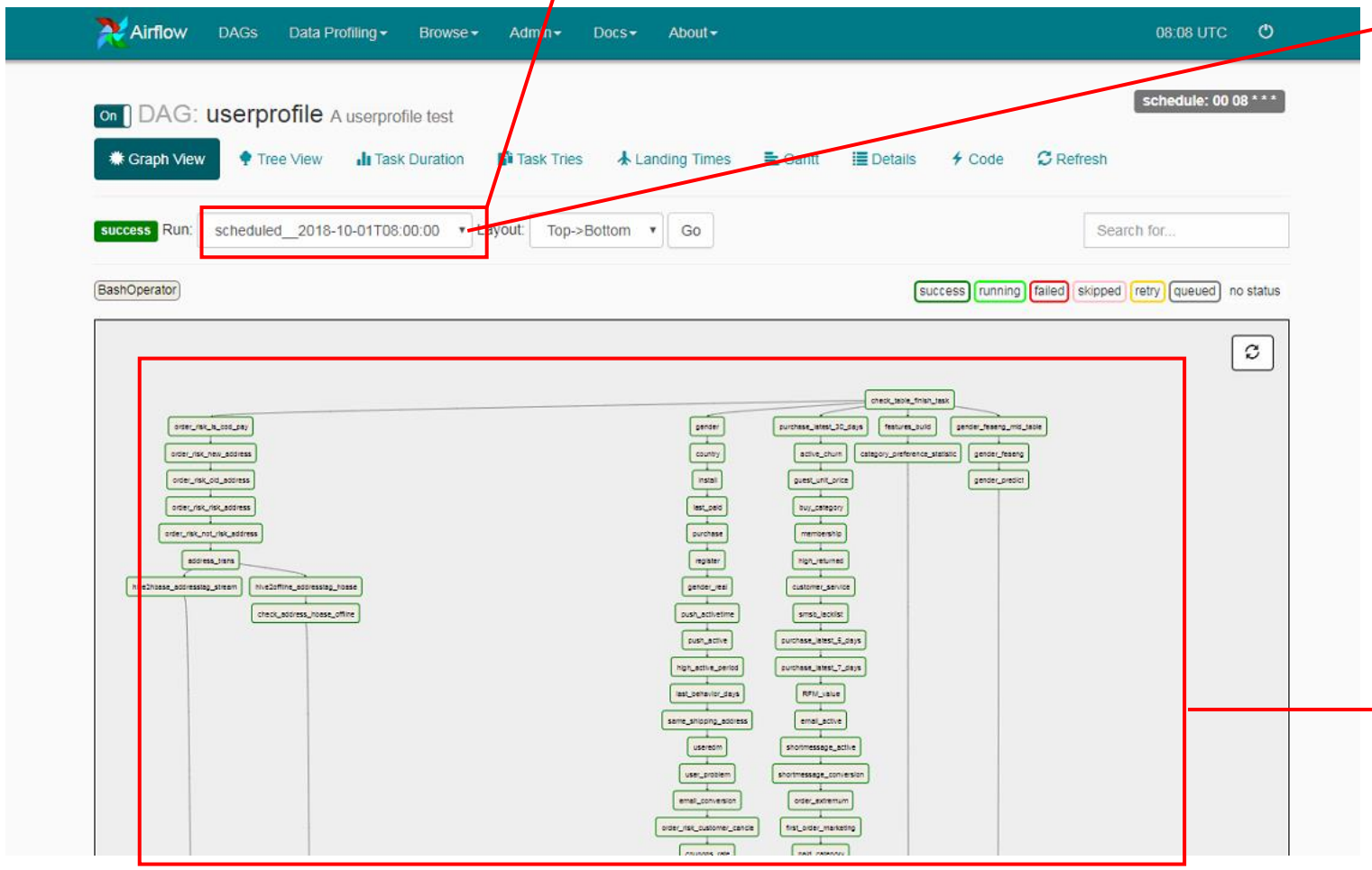
http://airflow.incubator.apache.org/tutorial.html

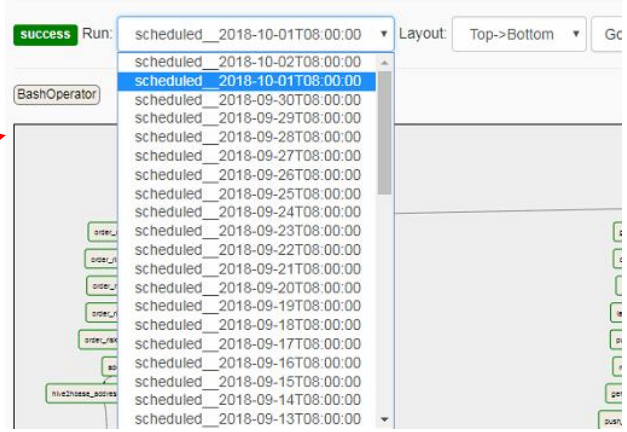


Airflow的安装、配置、使用文档，在airflow的官网中有详细的介绍和demo

Airflow调度—主要功能模块

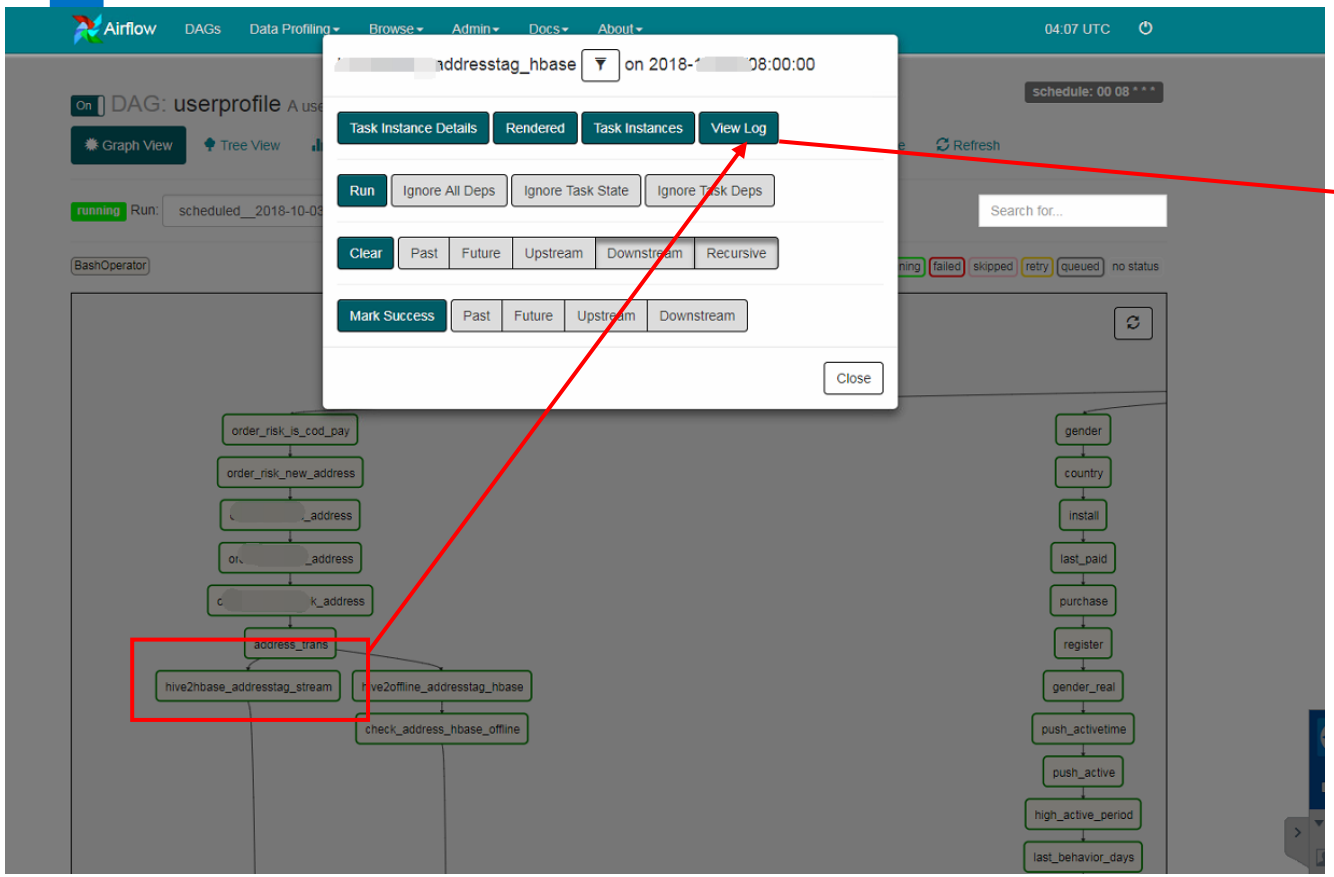
显示该DAG的调度日期，可以追溯查看之前日期的调度情况

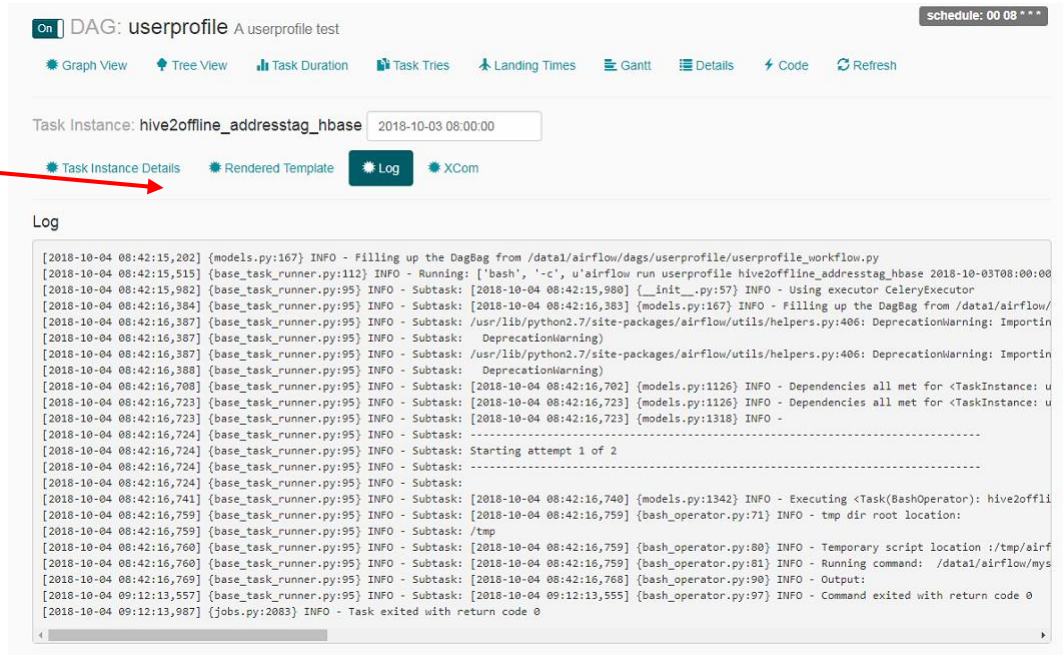




显示当前DAG调度中，各个模块之间调度的依赖和先后顺序

Airflow调度—主要功能模块





查看每个task的运行日志，对应运行报错的task，可根据日志文件进行排错

点击对应任务的task，选择“View log”可以查看该task对应的运行日志

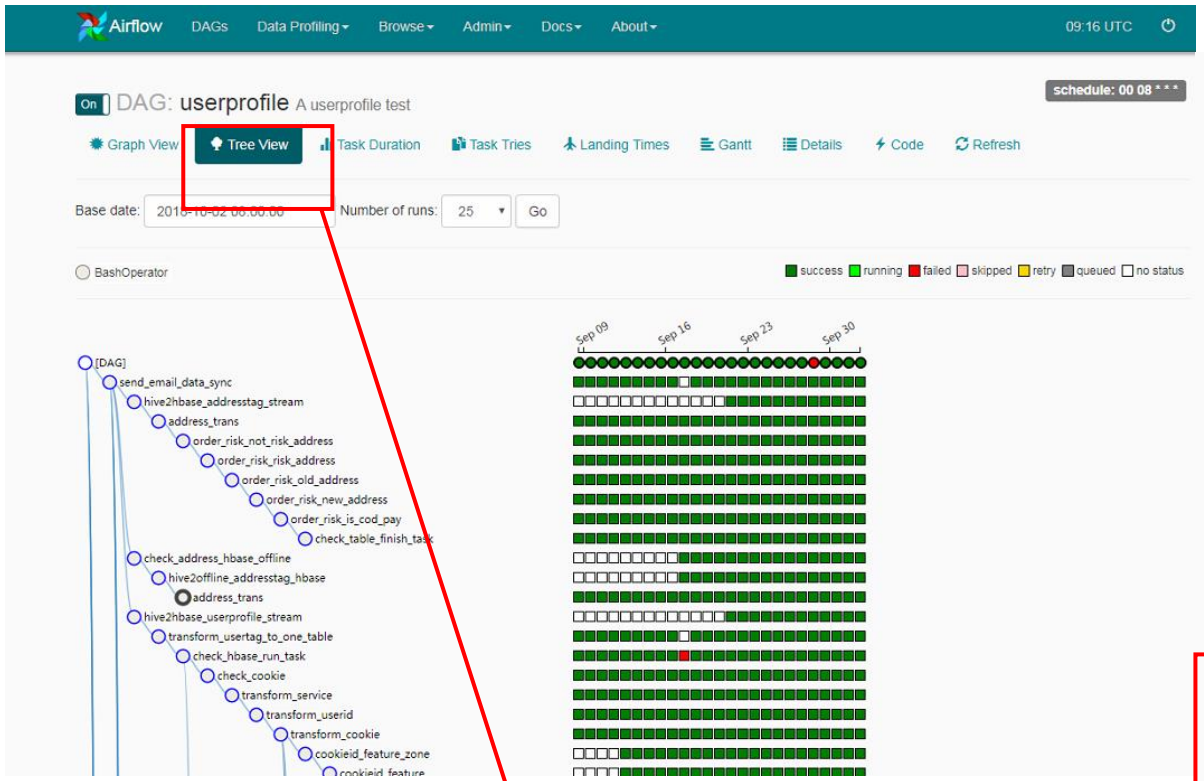
Run：运行当前task任务；

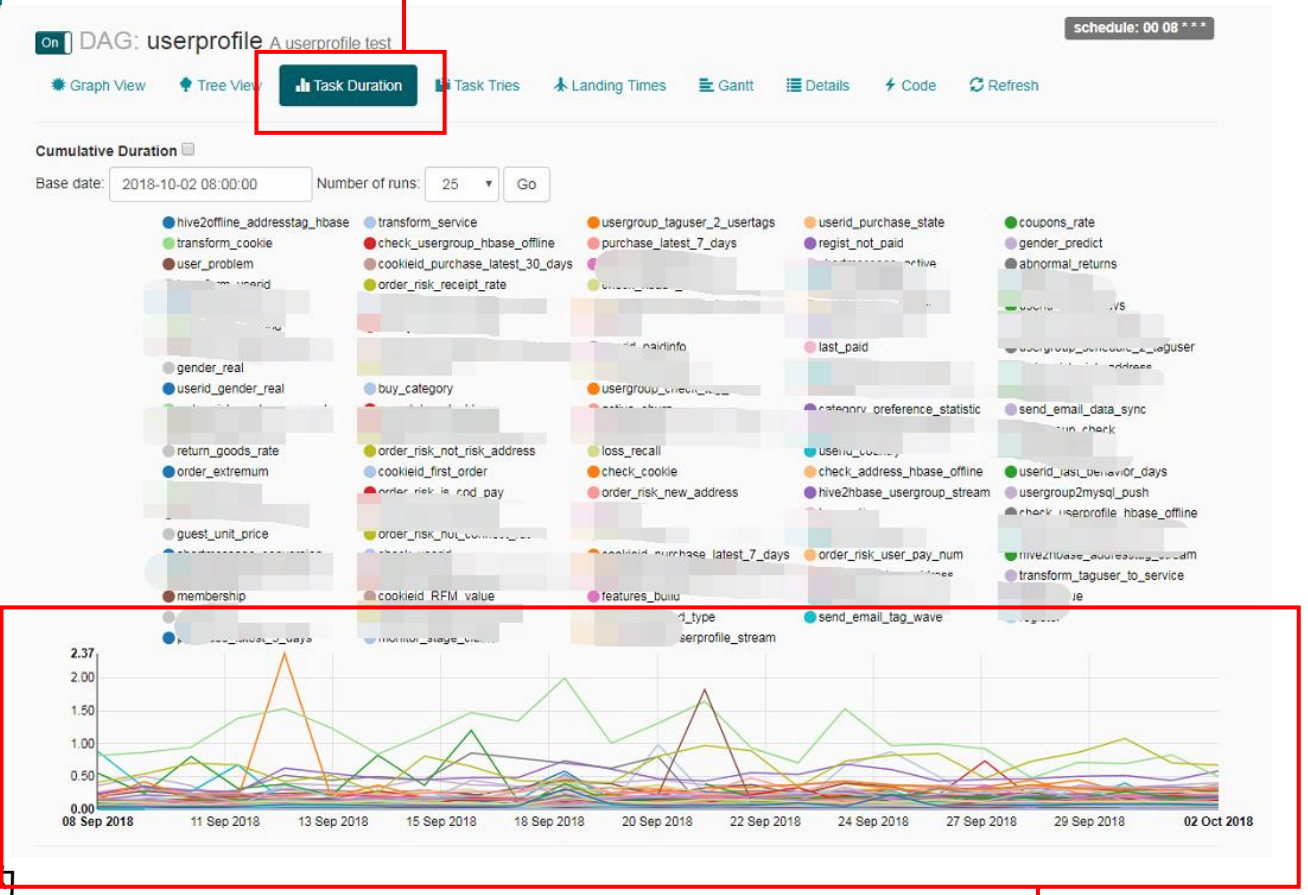
Clear：清除当前task及之后task的任务状态；

Mark Success：标记当前task的状态为成功，对于后续任务依赖前一个的状态为成功的来说，标记成功不影响后续任务运行；

Airflow调度—主要功能模块

显示该DAG调度的持续时间





这个是保留历史状态的DAG树，以树状图的形式展示各个task任务的调度情况（成功/失败/正在运行）

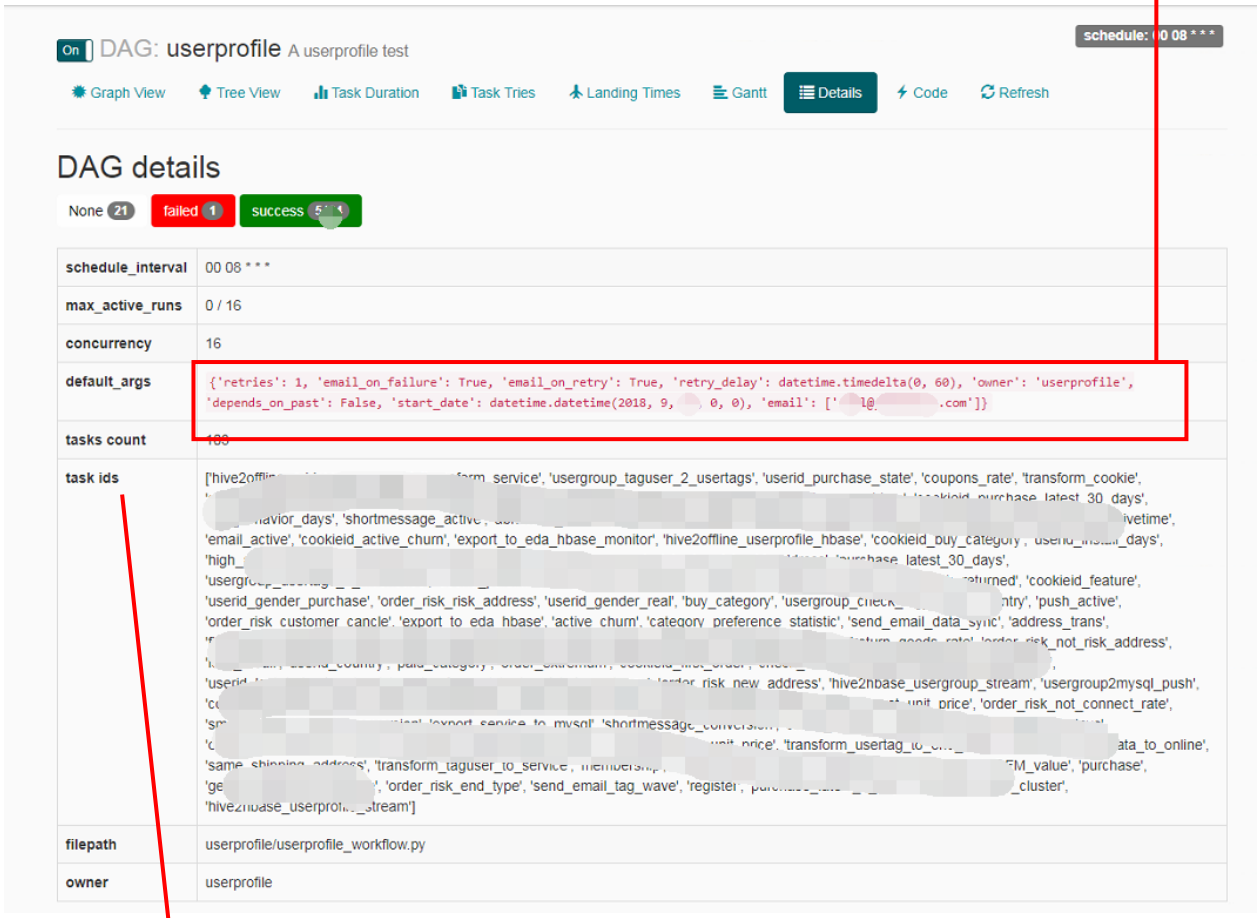
这里显示过去n日，不同任务（task）的持续时间。可以帮助找到异常值，快速理解每个任务的运行时间，以便进行排错和调优

Airflow调度—主要功能模块

以甘特图的形式显示每个任务调度的起止/持续时间

时间轴，将指针放到每个task任务上面，可以看到该任务的起止时间

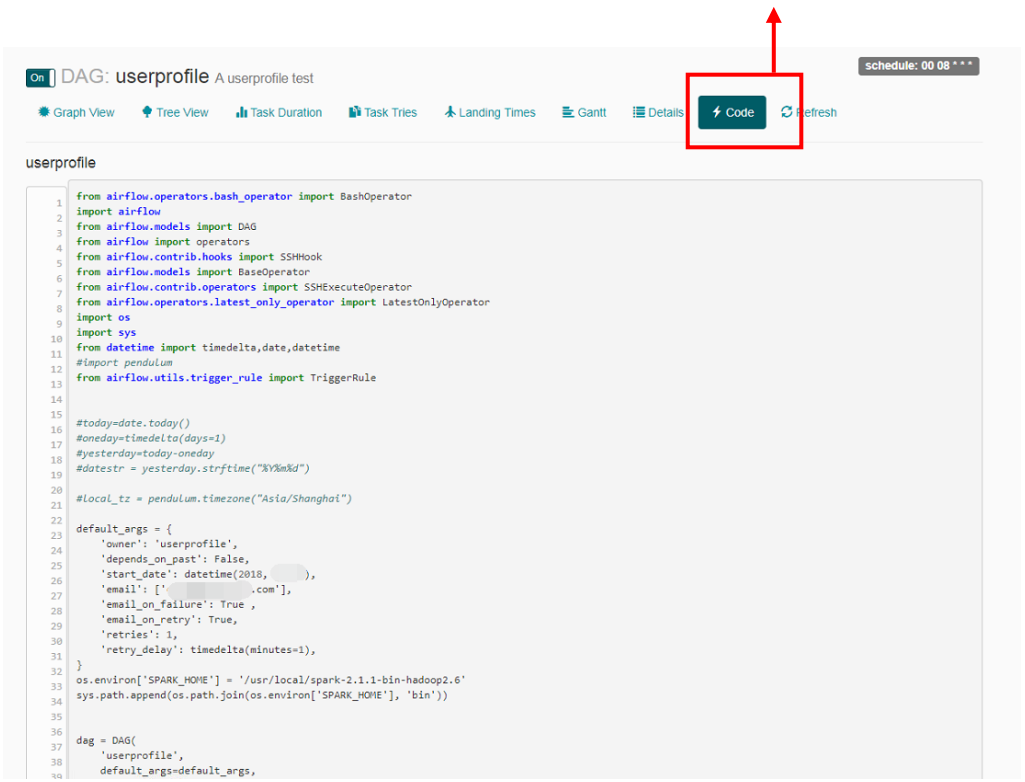
配置DAG运行的默认参数



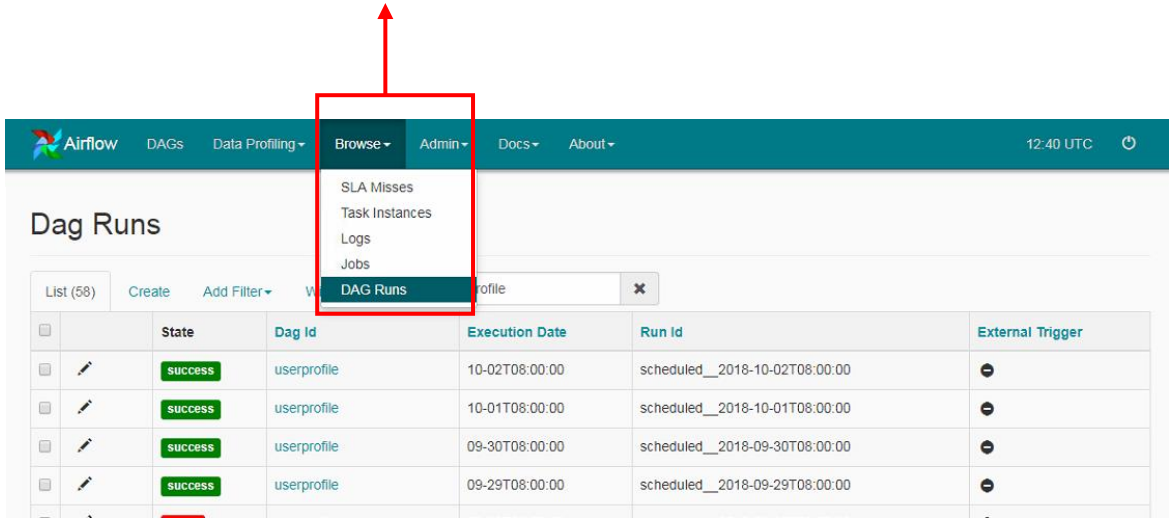
待调度的任务task

Airflow调度—主要功能模块

查看该DAG对应的调度脚本



如果DAG调度出现问题，可以从该模块筛选对应的DAG进行查看和整理



关于DAG的调度有几个地方需要注意一下：

- 配置参数里的Start_date是DAG首次运行的时间，如果配置时间在前面，会把历史任务同时调起来；
- 依赖参数里面常用的上游依赖包括“ALL_SUCCESS”“ ALL_DONE”，前者只有在上游执行成功时才会调起下游任务，后者只要上游任务执行完毕（不论是否执行成功）都可以调起下游任务；

DAG脚本示例（2）

```
high_active_period = BashOperator(
    task_id='high_active_period',
    bash_command='spark-submit  --master yarn --deploy-mode client --driver-memory 4g  --
executor-memory 8g --executor-cores 2 --num-executors 100 cookieid_high_active_period.py
{{ ds_nodash }} ',
    dag=dag,
    trigger_rule=TriggerRule.ALL_DONE)

push_active = BashOperator(
    task_id='push_active',
    bash_command='spark-submit   --master yarn --deploy-mode client --driver-memory 4g  --
executor-memory 8g --executor-cores 2 --num-executors 100 cookieid_push_active.py
{{ ds_nodash }} ',
    dag=dag,
    trigger_rule=TriggerRule.ALL_DONE)

…  # 配置相应用户画像标签脚本的 task，这里省略
buy_category >> userid_edm >> email_active
email_conversion >> paid_category >>sms_blacklist
```

- 左面这段脚本中引入了需要执行的task_id，并对dag进行了实例化。
- 其中以high_active_period这个task_id来说，里面的bash_command参数对于具体执行这个task任务的脚本，cookieid_high_active_period.py文件为执行加工用户高活跃标签对应的脚本。
- Trigger_rule参数为该task任务执行的触发条件，官方文档里面该触发条件有5种状态，一般常用的包括“ALL_DONE”和”ALL_SUCCESS”两种。其中“ALL_DONE”为当上一个task执行完成，该task即可执行，而”ALL_SUCCESS”为只当上一个task执行成功时，该task才能调起执行，执行失败时，本task不执行任务。
- “Bug_category>>userid_edm”命令为task脚本的调度顺序，在该命令中先执行“buy_category”任务后执行“userid_edm”任务。

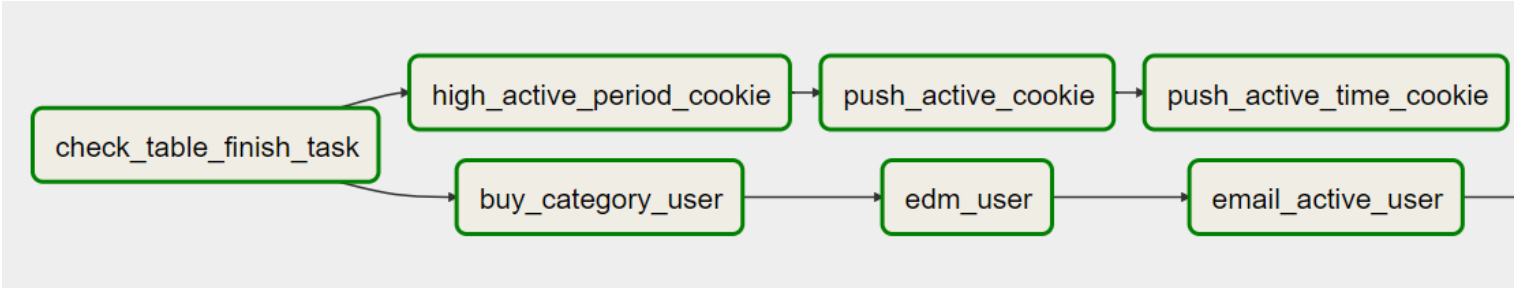
DAG脚本示例（3）

- 一旦Operator被实例化，它被称为“任务”。实例化为在调用抽象Operator时定义一些特定值，参数化任务使之成为DAG中的一个节点

```
op1 >> op2        ==        op1.set_downstream(op2)
op2 << op1        ==        op2.set_upstream(op1)
```

这里来看一个DAG调度示例

DAG调度流程图



task执行依赖

```
check_table_finish_task >> high_active_period_cookie >> push_active_cookie >> push_active_time_cookie
check_table_finish_task >> buy_category_user >> edm_user >> email_active_user
```

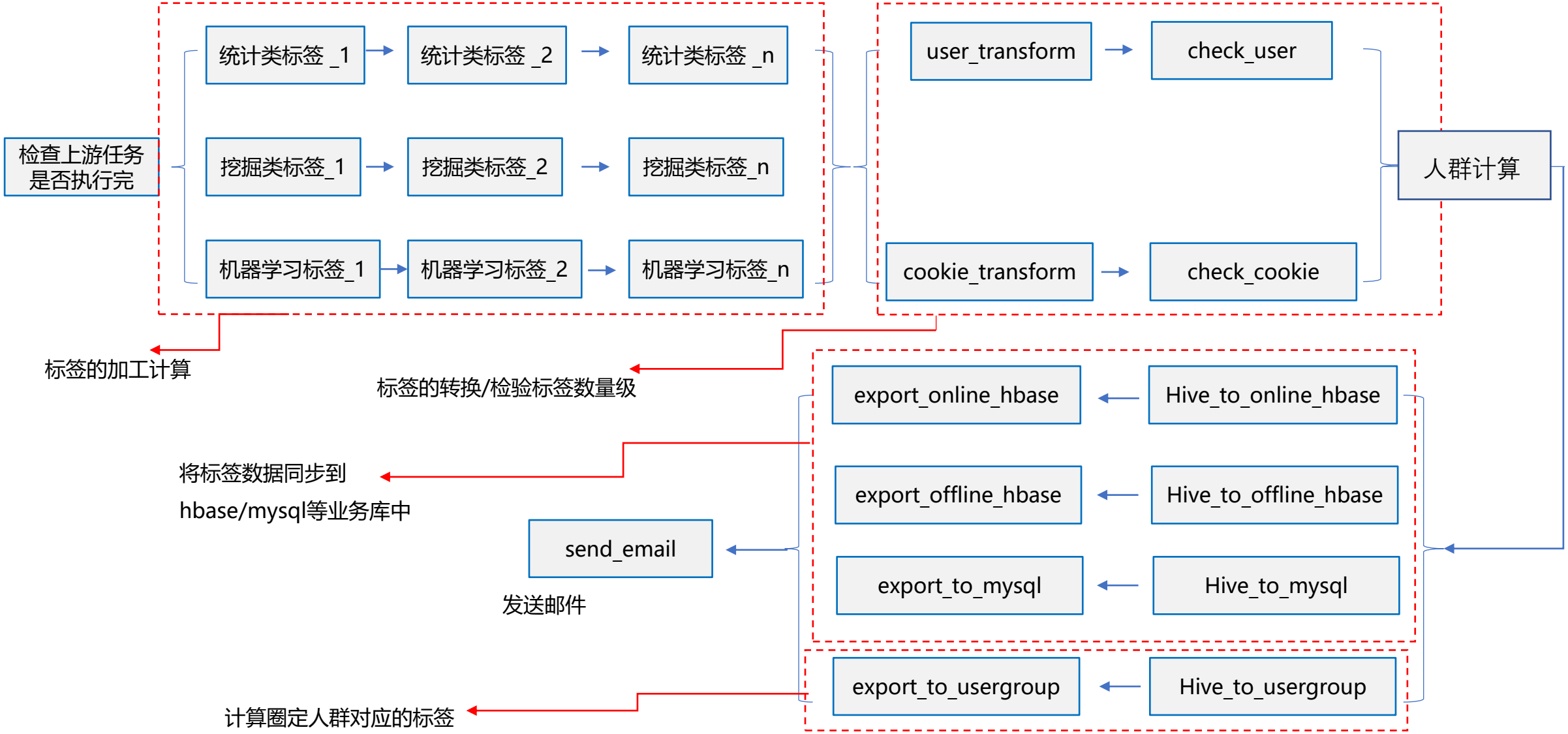
首先执行的脚本，如果执行失败，需要过段时间重试

该脚本执行依赖上游任务成功

后面脚本的执行依赖上游执行完成

工程化调度模块

工程化实践中，每天除了对用户标签执行ETL作业，还需要将与用户标签相关的数据同步到其他数据库或业务系统中，本页内容主要讲整体调度流程，该调度的控制在DAG脚本中即可进行配置



标签查询—视图

输入用户userid或cookieid，可查看该用户各维度信息

输入用户id后，可以查看该用户的属性信息、行为信息、风控属性等信息。从多方位了解一个具体的用户特征

扫一扫二维码关注公众号

1、每日推送人工智能、数据科学等行业最新重磅报告；

2、回复“干货”，获取海量个性化推荐相关技术文档；

3、最大最全的个性化推荐技术与产品社区，欢迎来撩；

个性化推荐技术与产品社区

标签编辑管理—列表页

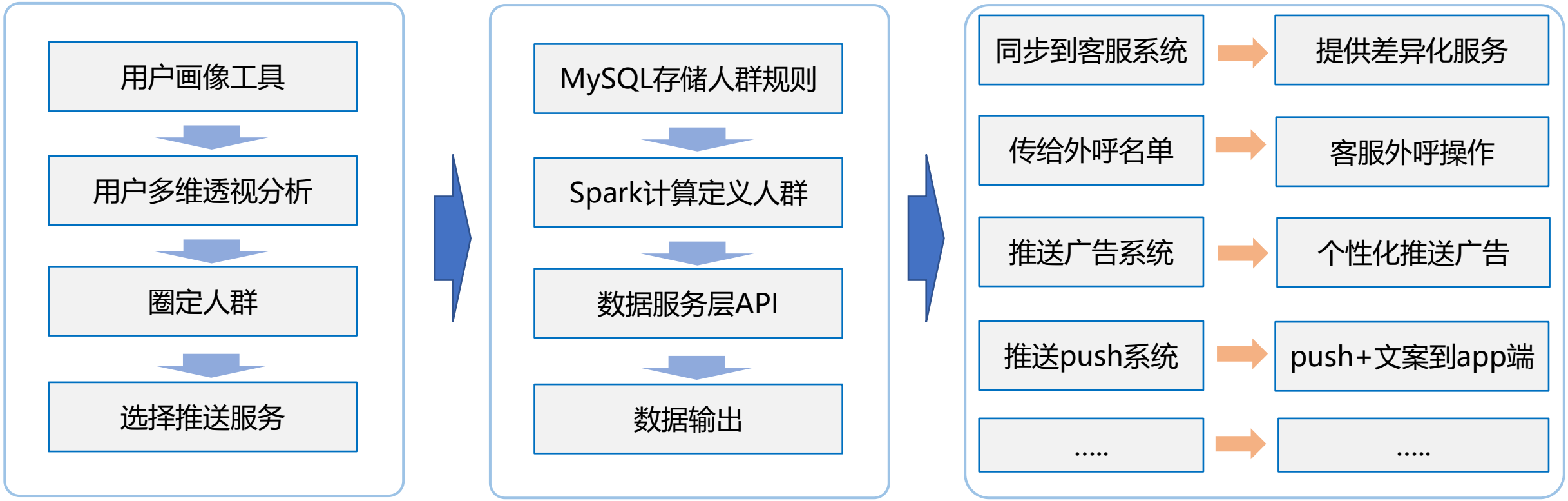
标签编辑管理中，开发人员可在web端编辑画像相关的**元数据**。编辑后，元数据插入到MySQL数据库中

用户点击编辑按钮，可对已经添加的元数据进行重新编辑

用户分群功能介绍

自定义人群提供根据现有用户标签设置圈定用户群体的功能，业务人员可利用“***多维透视分析***”功能进行人群的对比分析，通过预计算对该运营规则圈定的人群数量做测算。保存后将生成圈定人群的规则，而后依据该规则生成圈定人群推送到服务端

下面介绍提供产品化服务的调度流程



Web端的用户画像产品功能

数据计算层逻辑

输出到服务端

用户分群列表页

点击添加分组，可重新添加人群的分组。**支持多标签组合进行个性化定义人群**

点击“编辑”或“删除”，可对已保存的人群分组进行重新编辑，或者删除该人群分组

点击推送到不同的业务系统时，会将对应人群的标签数据推送到相应的数据库中。**如：**外呼系统可通过FTP传输待呼用户的数据文件、客服系统对应MySQL数据库、push系统对应hbase数据库等

筛选透视分析维度

保存后对应的用户群组的名称，点击叉号可以删除该人群

点击新建人群，回到上页内容，组合标签创建对应的用户群组

选择分析该批人群的维度。可供筛选的包括用户属性、用户行为等不同主题下的二级标签维度***或自己创建的人群***；

筛选维度后，可从这些维度透视查看该批用户群的特征

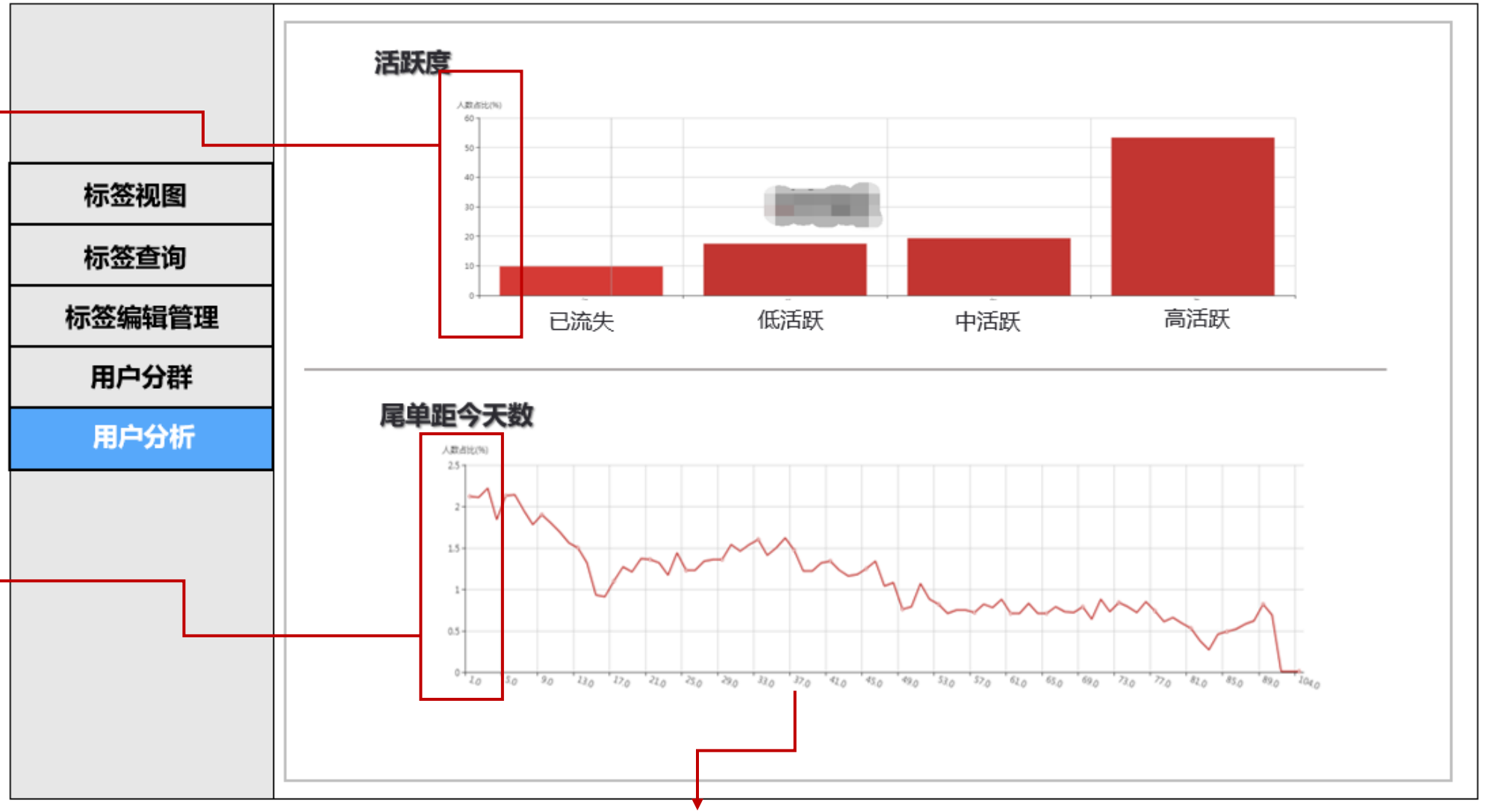
点击可保存该用户群

多维度透视分析筛选的用户群

承接上页ppt，对筛选出来的用户群及分析该批用户群的维度，以可视化的方式展现该批用户群特征

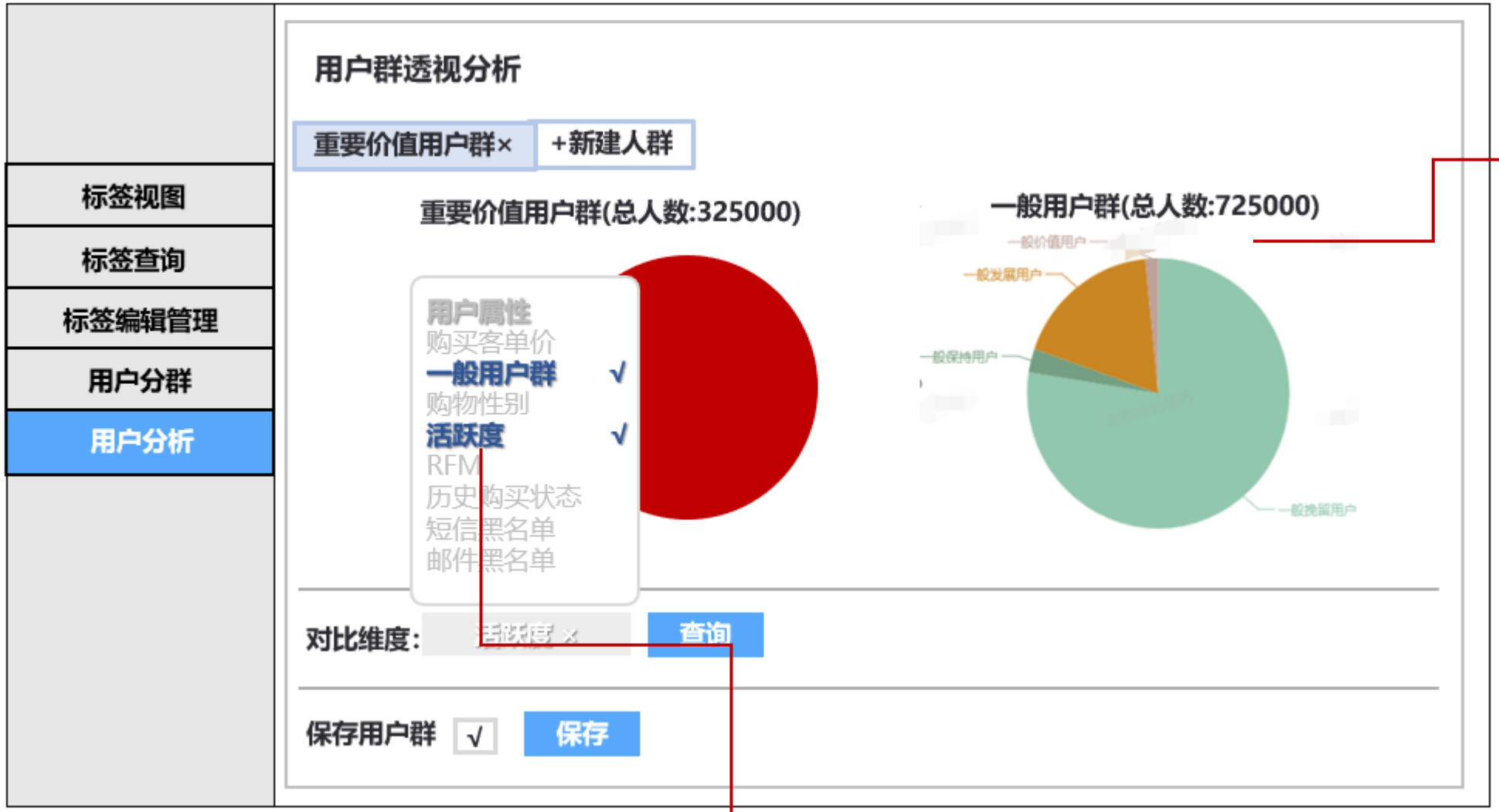
不同活跃度情况对应人数的占比

不同尾单距今天数对应人数的占比



尾单距今天数

透视分析—人群对比分析



新建立的对比分析的用户群

选择对比分析的维度

A/B测试案例：应用背景

本着数据驱动的理念，在正式切换到使用某种规则运营用户前，需要经过A/B测试来看AB哪个组可以带来更高的转化，带来的转化增量是多少。借助画像系统可以很方便地实现对两组人群分别运营效果的对照测试。

某美妆品牌为在大促活动期间取得较好的销量，计划通过push渠道种草产品功效、护肤教程等系列文章，为大促活动造势，激发销量转化。为了精准定位目标人群流量，渠道运营人员现在计划做两版A/B人群效果测试：

1、不同内容标题对流量的影响

2、精准推送相比普通推送带来的流量提升

A/B测试案例：画像切入点

整个项目中需要梳理清楚如何切分AB组流量，如何设计好AB组人群规则和效果监测。下面分步骤介绍画像系统如何切入到AB人群测试中。

1、对AB组流量做切分

为了做A/B组测试，首先需要做好流量的切分，结合平台上cookieid的生成机制，考虑从cookieid尾号入手做流量切分。目前平台上cookieid尾号对应的字符均匀切分为0-9、a-d共14位，每个字符覆盖平台用户数量相同。通过将每个用户的cookieid尾号开发成标签，以控制尾号的形式切分流量，可以将用户划分为A/B人群。

标签开发时，例如对于cookieid为“4957-AABC-DCD2285DB30D”的用户，其对应标签为“cookieid尾号:d”。这样对每个用户打上对应的尾号标签，通过勾选尾号标签可筛选用户群

A/B测试案例：测试方案1

2、测试文案标题对流量影响的方案

某平台渠道运营人员为在大促活动期间召回更多用户来访App，计划在活动预热期选取少量用户做一版文案标题的AB效果测试。

在该测试方案中，控制组A选取了近x天来访过，cookie尾号为a，且近x天内浏览/收藏/加购过美妆类目商品的用户群，给该批用户push美妆类文案A，对照组B选取了近x天来访过，cookie尾号为b，且近x天内浏览/收藏/加购过美妆类目商品的用户群，给该批用户push美妆类文案B。控制组和对照组的用户量相同，但文案不同，后续监控两组人群的点击率大小，进而分析不同文案对用户点击的影响。

A/B测试案例：测试方案2

3、精准推送相比普通推送带来的流量提升的测试方案

在使用画像系统精细化push人群前，某平台对用户采用无差别push消息的形式进行推送。为了测试精细化运营人群相比无差别运营带来的流量提升，渠道运营人员决定在近期重点运营的美妆营销会场做一版A/B效果测试。

该测试方案中，控制组A选取了近x天来访过，cookie尾号为1，没有类目偏好的用户群，对照组B选取了近x天来访过，cookie尾号为2，且近x天内浏览/收藏/加购过美妆类目商品的用户群。对AB组用户群都push相同的文案，后续监控两组人群的点击率大小，进而分析精准营销推送带来的增长点大小。

案例1：短信营销-业务规则

业务人员可根据要营销的用户量和营销内容，建立规则筛选需要推送短信/邮件的人群

做AB测试的时候，可以根据cookieid尾号对应的标签做AB组测试

业务类型	用户组名称	人群筛选预定规则	推送时间	推送链接	短信内容	推送覆盖人数
业务A	xxxx用户组	规则1+规则2+规则3 （对用户进行规则的限制，可通过组合标签实现）	2018.1.3	https://xxxxxx	短信文案配置 https://xxxxxx	50000
业务B	xxxx用户组	规则1+规则2+规则3 （对用户进行规则的限制，可通过组合标签实现）	2018.1.4	https://xxxxxx	短信文案配置 https://xxxxxx	40000
业务C	xxxx用户组	规则1+规则2+规则3 （对用户进行规则的限制，可通过组合标签实现）	2018.1.5	https://xxxxxx	短信文案配置 https://xxxxxx	50000
业务D	xxxx用户组	规则1+规则2+规则3 （对用户进行规则的限制，可通过组合标签实现）	2018.1.6	https://xxxxxx	短信文案配置 https://xxxxxx	45000
业务E	xxxx用户组			.....	短信文案配置 https://xxxxxx	50000

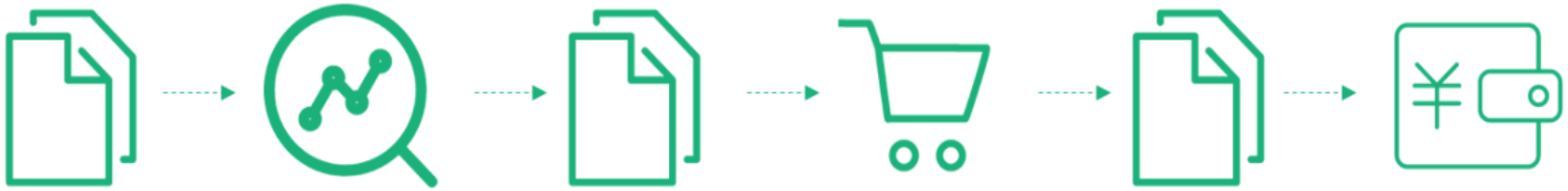
案例2：广告弹窗-应用效果

运营人员在画像系统（第7章中介绍）中根据业务规则定义组合用户标签筛选出用户群，并将该人群上线到广告系统中

将待运营的用户群上线到广告系统

Session分析介绍

用户进入电商类网站或APP的一个典型流程包括，进入首页后搜索关键词、点击商品板块或点击推荐商品进入详情页，在详情页浏览点击加购后退出该页面搜索其它商品继续浏览，最后进入订单页进行支付，或浏览途中退出APP。这系列行为就是用户的行为轨迹，对于用户这样的访问行为，我们称为session。



Session分析介绍

Session中记录了用户在什么时间点、通过什么样的行为、浏览了什么页面/商品。一般session的切割为固定时长，如定义APP端session的切割时长为5分钟时，用户每次访问行为如果距离上一次访问行为在5分钟之内，则记为同一次访问，如果距离上次访问大于5分钟则记为两次不同的访问。通过session_id可用来标识用户的访问，同一次连续访问的session_id相同，否则不同。

基于session对用户进行分析具有非常重要的作用，可以从用户的访问次数、访问路径、访问商品品类等多个维度分析用户特征。进一步地，分析用户首次访问的session对于挖掘影响用户购买行为具有重要的意义

Session分析特征构建

字段名称	中文名称	备注	示例
cookieid	用户id		d4a6-41e5-a670-85d217d055a6
eventkey1	第一次行为事件	浏览/收藏/加购等行为	add_to_bag
eventkey2	第二次行为事件	浏览/收藏/加购等行为	add_to_bag
eventkey3	第三次行为事件	浏览/收藏/加购等行为	add_to_bag
eventkey4	第四次行为事件	浏览/收藏/加购等行为	add_to_bag
eventkey5	第五次行为事件	浏览/收藏/加购等行为	add_to_bag
.....	.....	.....	.....
level3_name	3级品类名称	用户行为浏览/收藏/加购等行为次数最多对应商品所属三级品类	运动鞋
goods_event_num	商品数量	用户行为对应商品数量	11
goods_list	商品清单	用户行为对应商品明细，记录商品id及对应行为次数	{001:3;002:5;003:11;...}
avg_origin_price	商品平均原价	浏览/收藏/加购等行为对应商品	66
avg_promote_price	商品平均促销价	浏览/收藏/加购等行为对应商品	59
price_range	价格区间	浏览/收藏/加购行为对应商品	[50,100)
access_time	浏览时长	访问从开始到结束时间	600
click_num	点击次数		32
goods_number	商品数量		16
is_paid	是否下单	记录用户是否下订单	1

Session分析数据

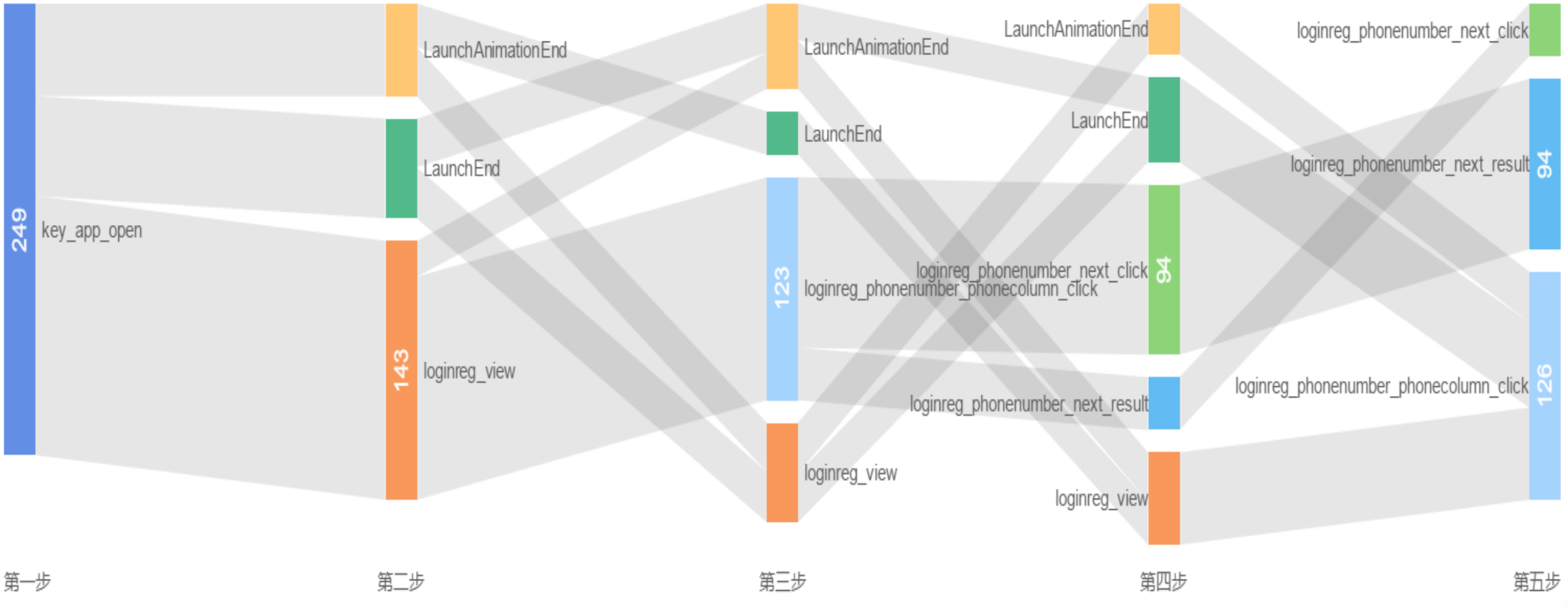
cookieid	eventkey	level3_name	ods_event_nu	goods_list	avg_origin_price	avg_promote_price	price_range	access_time	click_num	goods_number	is_paid	data_date
6945-4AAE-9911-FC1525850DA1	goodsdetail_view	洗发水	3	{"3766198":"1"}	310	294	>200	180	34	4	0	20181225
4C5E-402F-8D8F-D955ADFF7FB3	addtobag_click	牙膏	3	{"2226516":"1"}	9.99	0	[0,20]	788	91	1	0	20181225
8B2D-4715-BCF7-86A04A807107	addtobag_click	沐浴露	6	{"1001174":"1","530748":"1"}	32.99	10	(20,40]	553	84	4	0	20181225
E0D4-464A-BA5B-99779B0161CF	goodsdetail_view	香皂	4	{"4389212":"1"}	15.99	7.99	[0,20]	1620	216	13	0	20181225
E0D4-464A-BA5B-99779B0161CF	goodsdetail_view	礼盒	4	{"4389212":"1"}	15.99	7.99	[0,20]	1620	216	13	0	20181225
C86B-4FBE-A0A0-D9D441F02B9A	goodsdetail_view	洗手液	8	{"4365470":"1","4946434":"1"}	9.99	0	[0,20]	359	84	12	0	20181225
C86B-4FBE-A0A0-D9D441F02B9A	goodsdetail_view	洗手液	8	{"4365470":"1","4946434":"1"}	9.99	0	[0,20]	359	84	12	0	20181225
DACC-4413-B22F-2249E32051C1	goodsdetail_view	卫生纸	3	{"3582844":"1"}	24.99	0	(20,40]	945	109	8	0	20181225
FCCC-4068-97C8-FA0B60C73E4C	goodsdetail_view	厨房用纸	12	{"1980724":"2","2946772":"1","4217	10.99	0.5	[0,20]	1161	167	35	1	20181225
9800-431B-8812-31D435AE5E0E	goodsdetail_view	洗洁精	11	{"3685230":"1","1751462":"1","4774	139.72	23.03	(100,150]	825	169	47	0	20181225
D44D-4754-9BBE-213329C489E5	goodsdetail_view	洁厕剂	6	{"172567":"1","596632":"1"}	26.49	13.49	(20,40]	897	135	25	0	20181225
8744-4D90-B28C-ADA1560A6078	addtobag_click	洁厕剂	3	{"3661866":"1"}	9.99	4.99	[0,20]	799	152	2	0	20181225
8744-4D90-B28C-ADA1560A6078	addtobag_click	洗衣液	3	{"3661866":"1"}	9.99	4.99	[0,20]	799	152	2	0	20181225
C28A-4B1B-82E0-4F747E819ED8	goodsdetail_view	彩妆工具	3	{"2518138":"1"}	71	0	(60,80]	264	61	1	0	20181225
84E8-4F27-93D4-085863EAB225	goodsdetail_view	漱口水	4	{"3272696":"1"}	23.99	0	(20,40]	248	65	4	0	20181225
B240-44DC-BBAC-37151BDBF7B6	goodsdetail_view	美发工具	3	{"1656288":"3"}	769	0	>200	1512	197	11	0	20181225
2795-4ED7-B15B-3BE334687CE2	goodsdetail_view	香皂	6	{"4336200":"1","4072692":"1"}	71.5	34.5	(60,80]	489	114	14	1	20181225
067F-4F0A-B609-BACB981A2BE2	goodsdetail_view	礼盒	3	{"3624142":"1"}	34.99	28.99	(20,40]	1220	98	18	0	20181225
7627-40FB-949D-CAEC4A3B4023	goodsdetail_view	洗手液	7	{"4622712":"2","197939":"1"}	21.7	15.7	(20,40]	269	29	3	0	20181225
7627-40FB-949D-CAEC4A3B4023	goodsdetail_view	洗手液	7	{"4622712":"2","197939":"1"}	21.7	15.7	(20,40]	269	29	3	0	20181225
E020-4F91-996A-A07D70C1FEF1	goodsdetail_view	面膜	17	{"4562986":"3","3479876":"6","4942	21.4	9.64	(20,40]	484	146	21	0	20181225
E020-4F91-996A-A07D70C1FEF1	goodsdetail_view	精华	17	{"4562986":"3","3479876":"6","4942	21.4	9.64	(20,40]	484	146	21	0	20181225
5029-4AA9-8E5E-076D4C75602C	goodsdetail_view	防晒	3	{"4190982":"2"}	52	38.99	(40,60]	307	73	6	0	20181225
6624-452B-B7E7-1B97F47462BF	goodsdetail_view	爽肤水	3	{"2790564":"1"}	30.99	6.99	(20,40]	413	40	4	0	20181225
6624-452B-B7E7-1B97F47462BF	goodsdetail_view	美甲	3	{"2790564":"1"}	30.99	6.99	(20,40]	413	40	4	1	20181225
32DC-4B99-8687-71B143E40671	goodsdetail_view	男士香水	4	{"4656856":"1"}	37.99	22.99	(20,40]	419	66	6	0	20181225
7114-4AC8-82A6-383198C069F6	goodsdetail_view	垃圾袋	3	{"3291054":"1"}	6.99	2.99	[0,20]	255	89	6	0	20181225
6847-47A2-8A01-93AB0153C1BF	goodsdetail_view	卫生巾	4	{"4496272":"1"}	304	288.99	>200	158	60	1	0	20181225
6847-47A2-8A01-93AB0153C1BF	goodsdetail_view	女生香水	4	{"4496272":"1"}	304	288.99	>200	158	60	1	0	20181225
6FBE-4646-ADD8-6EFF5D2CF919	goodsdetail_view	洗手液	3	{"977190":"4"}	33.99	27.99	(20,40]	135	59	4	1	20181225
FD17-420A-962B-3A2832517FC9	addtobag_click	洗手液	3	{"2225960":"2"}	35.99	0	(20,40]	807	257	2	0	20181225
AD1E-482F-A55A-423D347991CE	goodsdetail_view	面膜	19	{"1327832":"2","3634942":"2","1577	32.36	18.71	(20,40]	635	194	25	0	20181225
F085-4302-8EF8-5DEC8CBDBDF2	goodsdetail_view	精华	3	{"2557358":"1"}	80	37.99	(60,80]	504	72	5	0	20181225
DB98-4CD4-A3E9-517CC055387F	goodsdetail_view	防晒	4	{"4830686":"1"}	19.43	14.9	[0,20]	144	35	3	0	20181225
9442-44BF-89C2-68361176C2DD	addtobag_click	垃圾袋	4	{"4480258":"1"}	22.99	9.99	(20,40]	497	88	2	0	20181225
4C79-4D80-9688-A06926B98BE1	goodsdetail_view	卫生巾	27	{"231855":"1","2535354":"1","23337	33.58	19.77	(20,40]	1592	249	34	0	20181225
2B19-490C-991A-4390120D3C1D	addtobag_click	女生香水	4	{"4764064":"1"}	13.04	11.18	[0,20]	326	83	1	0	20181225
31D6-408E-ABDE-5ABDD812DDFE	goodsdetail_view	洗手液	4	{"4705066":"1"}	29.99	0	(20,40]	178	91	2	0	20181225
D7F6-4FA5-AE54-57AF407D808A	goodsdetail_view	洗手液	32	{"3259126":"1","4864050":"1","4861	11.18	2.56	[0,20]	1399	178	40	0	20181225
D7F6-4FA5-AE54-57AF407D808A	goodsdetail_view	卫生纸	32	{"3259126":"1","4864050":"1","4861	11.18	2.56	[0,20]	1399	178	40	1	20181225
83DB-4549-ACE6-7DA822DD685F	goodsdetail_view	厨房用纸	14	{"4382552":"1","2809962":"1","8556	41.44	12.63	(40,60]	380	89	5	0	20181225
0AEE-415A-A40D-88F1D145C1CD	goodsdetail_view	洗洁精	4	{"5010028":"1"}	52	27.99	(40,60]	1280	78	21	0	20181225

Session分析—用户行为路径

根据用户的访问路径分析，对于产品设计的改进有很大帮助，分析用户从登陆、搜索、浏览详情页到购买的行为路径，根据用户在各环节的转化率发现用户行为偏好和影响订单转化的主要因素。

eventLst	num
key_app_open,loginreg_view,null,null,null	136
key_app_open,loginreg_view,loginreg_phonenumber_phonecolumn_click,loginreg_phonenumber_next_click,loginreg_phonenumber_next_result	94
key_app_open,loginreg_view,loginreg_phonenumber_phonecolumn_click,loginreg_phonenumber_next_result,loginreg_phonenumber_next_click	29
key_app_open,LaunchEnd,loginreg_view,LaunchAnimationEnd,loginreg_phonenumber_phonecolumn_click	28
key_app_open,LaunchEnd,LaunchAnimationEnd,loginreg_view,loginreg_phonenumber_phonecolumn_click	27
key_app_open,LaunchAnimationEnd,loginreg_view,LaunchEnd,loginreg_phonenumber_phonecolumn_click	27
key_app_open,loginreg_view,loginreg_phonenumber_phonecolumn_click,null,null	26
key_app_open,null,null,null,null	25
key_app_open,LaunchAnimationEnd,LaunchEnd,loginreg_view,loginreg_phonenumber_phonecolumn_click	24
key_app_open,loginreg_view,LaunchAnimationEnd,LaunchEnd,loginreg_phonenumber_phonecolumn_click	20
key_app_open,loginreg_view,LaunchEnd,LaunchAnimationEnd,loginreg_phonenumber_phonecolumn_click	18
key_app_open,LaunchEnd,loginreg_view,LaunchAnimationEnd,guidepage_skip	17
key_app_open,key_app_open,loginreg_view,loginreg_view,loginreg_phonenumber_phonecolumn_click	17
key_app_open,loginreg_view,loginreg_phonenumber_phonecolumn_click,loginreg_phonenumber_next_click,loginreg_verification_sendsms...	15
key_app_open,LaunchAnimationEnd,null,null,null	15
key_app_open,loginreg_view,loginreg_phonenumber_next_click,null,null	14
key_app_open,LaunchAnimationEnd,LaunchEnd,loginreg_view,guidepage_skip	13
key_app_open,LaunchAnimationEnd,loginreg_view,LaunchEnd,guidepage_skip	13
key_app_open,LaunchEnd,LaunchAnimationEnd,loginreg_view,guidepage_skip	12
key_app_open,loginreg_view,LaunchEnd,LaunchAnimationEnd,guidepage_skip	11

Session分析—用户行为路径



用户访问session流量桑基图

应用效果评估标准

精准营销是数据价值的一个重要出口，但如何评估效果的好坏，不同业务线的人员有不同的关注重点。总体来看，可分为流量提升导向和GMV提升导向两种情况。

有的业务线人员背的kpi指标是流量，因此关注的重点是流量提升，例如负责push业务线的人员。这种情况下，对效果的分析会对比使用圈定人群进行精准推送方式带来的点击率，与没有使用用户画像进行无差别普通推送带来的点击率相比，是否有所提升，提升多少个百分点。

有的业务线人员背的kpi指标是gmv，因此关注的重点是roi的转化，例如短信营销、外呼营销的业务线人员。这种情况下，对效果的分析会关注营销活动中营销了多少用户、实际触达了多少用户、有多少用户实际付费以及带来的gmv，对比实际营销成本（短信、外呼电话的成本），分析营销的roi。