# 小赢数仓开发手册

### 文档详细信息

|  |  |
| --- | --- |
| 文档名称 | 小赢数仓开发手册 |
| 制定部门 | 大数据平台部 |
| 签发人 | 邹波 |
| 发布时间 | 2019-03-08 |

**文档修订记录**

|  |  |  |
| --- | --- | --- |
| 修订日期 | 版本 | 变更描述 |
| 2019-02-25 | 1.0 | 创建 |
| 2019-03-08 | 1.1 | 丰富数仓开发必备技能 |
| 2019-03-15 | 1.2 | 增加脚本开发规范案例 |

小赢数仓开发手册 1

文档详细信息 1

第一章 前言 4

1.1 文档目的和适用对象 4

1.1.1 文档目的 4

1.1.2 适用对象 4

1.2 术语表 4

第二章系统简介 5

2.1 小赢大数据数仓体系 5

2.2 小赢大数据数仓体架构图 5

第三章命名规范 6

3.1 数仓基本命名规范 6

3.1.1 数据分层代码命名规范 6

3.1.2 数仓各层脚本命名规范 7

3.1.3 数仓数据库命名规范 7

3.1.4 数仓表命名规范 8

3.1.5 周期标志代码命名规范 8

3.1.6 数据表存储类型代码命名规范 8

3.2 临时表与中间表表命名规则 9

3.2.1 临时表命名规范 9

3.2.2 中间表命名规范 9

3.3 备份表命名规则 9

3.4 任务调度azkaban命名规范 9

3.5 建表规范 10

第四章 数据接入 12

4.1 数据接入的五种方式 12

4.2 数据接入 12

4.2.1 业务库表接入规范 12

4.2.2 业务表增加更新字段 13

4.3 数据接入表字段映射关系 13

4.4 业务逻辑和库表变更规范 14

4.5 数据接入具体示例图 14

4.5.1 申请数据接入 14

4.5.2 申请数据列表 15

4.5.3 开始接入数据 16

4.5.4 选择映射关系 17

4.5.5 抽取方式 17

第五章 开发规范 19

5.1 合并小文件 19

5.2 ETL任务加大资源 19

5.3 azkaban调用脚本的问题 20

5.4 脚本开发规范 21

第六章 数仓开发必备技能 23

6.1 高频率开窗函数 23

6.2 大数据SQL与传统关系型数据库SQL的异同点 26

6.2.1 大数据SQL不支持delete 26

6.2.2 大数据SQL不支持update 27

# 第一章 前言

## 1.1 文档目的和适用对象

### 1.1.1 文档目的

本文档将详细描述小赢大数据数仓开发流程以及注意事项，引导数仓开发同事按照正确的流程完成作业。

### 1.1.2 适用对象

产品运营，数据分析，数据仓库开发等同事

## 1.2 术语表

表格 1 术语表

|  |  |
| --- | --- |
| 术语 | 解释 |
| ODS | Operational Data Store操作型存储数据 |
| SNAP | Snapshot快照层数据 |
| DW | data warehouse数据仓库 |
| DM | Data Market数据集市 |
| APP | Application应用层 |

# 第二章系统简介

## 2.1 小赢大数据数仓体系

小赢小赢的大数据数据仓库整体体系架构底层基于Hadoop平台进行构建，采用hive管理数仓元数据。小赢数据仓库主要分为五层，ODS，SNAP，DW，MART，APP层。

## 2.2 小赢大数据数仓体架构图

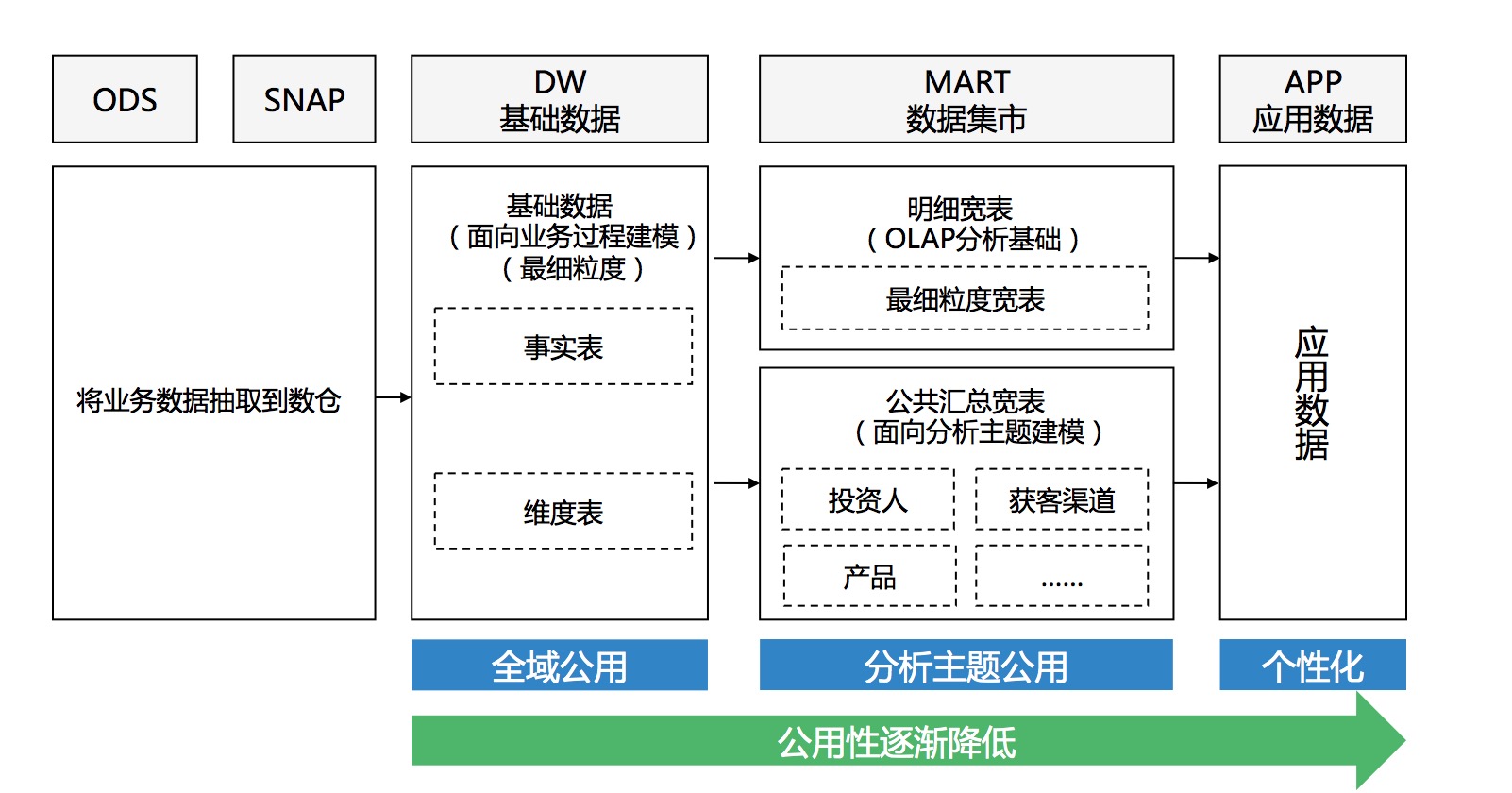


图 1小赢大数据数仓体架构图

# 第三章命名规范

## 3.1 数仓基本命名规范

如下讲解每一层的命名规范

### 3.1.1 数据分层代码命名规范

表格 2 数据分层代码规范

|  |  |  |
| --- | --- | --- |
| 用途 | 约定名 | 说明 |
| ODS | xy\_ods | 数仓贴源层，存放业务库增量或全量数据，操作型数据存储 |
| SNAP | xy\_snap | 数仓快照层，存放业务库某一个时间点的 快照数据，快照数据存储 |
| DW | xy\_dw | 数仓模型层，主要存放事实表以及维度表， 宽表等数仓etl之后的表，数据模型数据存储 |
| Mart | xy\_mart | 数据集市层，主要存储各个集市的主题数据， 数据集市存储 |
| APP | xy\_app | 数据应用层，主要存放各个主题 或者跨主题的指标统计等，数据应用存储 |

### 3.1.2 数仓各层脚本命名规范

表格 3 数仓各层脚本命名规范

|  |  |  |
| --- | --- | --- |
| 数仓层级 | 脚本命名规范 | 示例 |
| ODS | datax\_库名\_db\_表名.sh | datax\_yingzhongtong\_rc\_db\_production.sh |
| SNAP | snap\_库名\_db\_表名.sh | snap\_yingzhongtong\_rc\_db\_production.sh |
| DW | dw\_(fact/dim)\_表名.sh | dw\_fact\_current\_sale.sh,dw\_dim\_sku\_info.sh |
| MART | 主题\_表名.sh | jrb\_investing\_info.sh |
| APP | 主题\_表名.sh | huarui\_report\_loan.sh |

### 3.1.3 数仓数据库命名规范

表格 4 数仓数据库命名规范

|  |  |  |
| --- | --- | --- |
| 数仓层级 | 库命名规范 | 示例 |
| ODS | xy\_层级 | xy\_ods |
| SNAP | xy\_层级 | xy\_snap |
| DW | xy\_层级 | xy\_dw |
| MART | xy\_层级 | xy\_mart |
| bj\_层级 | bj\_mart |
| wallet\_层级 | wallet\_mart |
| APP | xy\_层级 | xy\_app |
| bj\_层级 | bj\_app |
| wallet\_层级 | wallet\_app |
| TEMP | xy\_层级 | xy\_temp |

### 3.1.4 数仓表命名规范

表格 5 数仓表命名规范

|  |  |  |
| --- | --- | --- |
| 数仓层级 | 表命名规范 | 示例 |
| ODS | 源业务库\_db\_业务库表名 | p2p\_db\_t\_loan |
| SNAP | 源业务库\_db\_业务库表名 | p2p\_db\_t\_loan |
| DW | (fact/dim)\_表名 | fact\_sku\_info,dim\_uid\_invitor |
| MART | 主题\_表名 | jrb\_investing\_info |
| APP | 主题\_表名 | huarui\_report\_loan |

### 3.1.5 周期标志代码命名规范

表格 6 周期标志代码命名规范

|  |  |
| --- | --- |
| 属性 | 约定名 |
| 小时(hour) | h |
| 日(day) | d |
| 周(week) | w |
| 月(month) | m |
| 季(quarter) | q |
| 年(year) | y |
| 无周期 | n |

### 3.1.6 数据表存储类型代码命名规范

表格 7 数据表存储类型代码命名规范

|  |  |  |
| --- | --- | --- |
| 存储类型 | 约定名 | 说明 |
| 全量表 | a | 快照表，用户随着时间推移会不断累积，某个时间点只是一个快照 |
| 增量表 | z | 按一定时间周期增量 |

## 3.2 临时表与中间表表命名规则

临时表：短暂性的生成，可以随便被删除的表，一般用于做测试等工作。

中间表：数仓在运算一个结果的过程中，中间生成的一些中间过度表，此表不能被删除，每晚需反复使用到，且里面的数据一般都是被insert overwrite覆盖的。

### 3.2.1 临时表命名规范

临时表建表：需按照以下格式，xy\_temp.temp\_你的名字\_表名，譬如xy\_temp.temp\_zhoubing\_test\_01

### 3.2.2 中间表命名规范

中间表建表：需按照以下格式，xy\_temp.mid\_db\_table\_01

其中db取值（dw,mart,app）,table取值为对应的结果表的表名

譬如xy\_mart层要生成一张结果表叫test，需要生成两个中间表，则表名为xy\_temp.mid\_mart\_test\_01,xy\_temp.mid\_mart\_test\_02

## 3.3 备份表命名规则

用于存储从业务库导入 hive 的表

命名规范为 xy\\_history.库名\\_db\\_表名,如：

xy\\_history.p2p\\_db\_t\\_loan

## 3.4 任务调度azkaban命名规范

表格 8 任务调度azkaban命名规范

|  |  |  |
| --- | --- | --- |
| 数仓层级 | 任务命名规范 | 示例 |
| ODS | task\_src\_to\_xy\_ods\_库名\_db\_表名 | task\_src\_to\_xy\_ods\_revoloan\_db\_t\_loan\_event |
| SNAP | task\_xy\_ods\_to\_xy\_snap\_库名\_db\_表名 | task\_xy\_ods\_to\_xy\_snap\_revoloan\_db\_t\_loan\_event |
| DW | tast\_dw\_(fact/dim)\_表名 | task\_dw\_fact\_current\_sale |
| MART | task\_xy\_mart\_主题 | task\_xy\_mart\_baixin\_pay\_order |
| APP | task\_xy\_app\_主题 | task\_xy\_app\_outbound\_call\_user |

图示例：



图 2 azkaban任务配置

## 3.5 建表规范

\* 表的存储格式需为列式存储ORC格式

\* 金额字段需用bigint或者decimal(20,0)。\*\*重要\*\*（如果不转成这种格式，hive在计算的时候默认会转成double，导致计算精度丢失）

\* 如果是分区表，分区字段名称统一为(xy\_date)

\* 字段名称需要COMMENT注释

\* 表名称需要COMMENT注释

1. **CREATE** **TABLE** xy\_mart.**table**(
2. `fuiloanid` string COMMENT '标的',
3. `FuiLoanUserId` string COMMENT '借款人',
4. `period` **bigint** COMMENT '逾期期数',
5. `plan\_principal` **decimal**(20,0) COMMENT '逾期本金',
6. `plan\_interest` **decimal**(20,0) COMMENT '逾期利息',
7. `overdue\_time` string COMMENT '逾期时间',
8. `uid` string COMMENT '担保方uid',
9. `user\_name` string COMMENT '担保方',
10. `FuiLoanType` string COMMENT '产品类型',
11. `FstrTitle` string COMMENT '产品名称'
12. )
13. COMMENT '表名称作用'
14. PARTITIONED **BY** (xy\_date STRING)
15. STORED **AS** orc;

# 第四章 数据接入

## 4.1 数据接入的五种方式

表格 9 数据接入五种方式

|  |  |  |
| --- | --- | --- |
| 源业务库表类型 | 特点 | 接入方式 |
| 事实表 | 增量数据多，数据更新多，表较大 | 根据更新字段增量接入 |
| 维度表 | 增量数据少，数据更新少，表较小 | 全量接入 |
| 流水表 | 增量数据多，数据无更新，表非常大 | 根据创建时间按日期分区接入 |
| 分区表 | 增量数据多，数据有多有少，表较多且量非常大 | 拼接sql收拢分区表，按事实表或者流水表的方式接入 |
| 归档表 | 增量数据多，数据更新多，表大且不定期归档 | 收拢历史归档，按事实表接入 |

## 4.2 数据接入

### 4.2.1 业务库表接入规范

对于需要接入接入数仓的业务库表，务必满足如下需求：

源业务库的表必须有主键

源业务库的表必须有更新字段，此更新字段非业务属性，业务无需关心，由数据库本身维护，而非代码指定，这个更新字段的命名可以统一定义，例如：

源业务库的表在更新字段上需要建立索引，特殊情况数据量非常小可以不建立索引，数据量大的必须建立索引

检查源业务库的表是否有敏感字段，如果有，统一提给安全同学，并要求业务修改，如不能及时修改需要在数据仓库中做脱敏处理，数仓禁止有敏感数据出现

**建议**：DBA统一收拢，加入新建表的规范，在建表时候进行检查，只有符合才能创建新表，并强制执行

### 4.2.2 业务表增加更新字段

对原业务没有更新字段的表，可按照如下模板进行添加更新字段以及索引

1. **alter** **table** db.**table** **add** FstrUpdateTime **timestamp** NOT NULL **DEFAULT** CURRENT\_TIMESTAMP **ON** **UPDATE** CURRENT\_TIMESTAMP COMMENT '更新时间',
2. **add** **index** idx\_update\_time (FstrUpdateTime);

## 4.3 数据接入表字段映射关系

表格 10 数据接入表字段映射关系

|  |  |  |
| --- | --- | --- |
| 源业务表字段类型 | ODS层字段类型 | SNAP层字段类型 |
| 主键( 论字段个数以及类型) | string | string |
| varchar | string | string |
| int | int | int |
| int(11) unsigned | bigint | bigint |
| bigint | bigint | bigint |
| bigint(20) unsigned( 额字段) | string | decimal(20,0) |
| blob | string | string |
| smallint | int | int |
| tinyint | int | int |
| mediumblob | string | string |

## 4.4 业务逻辑和库表变更规范

事后处理

1）DBA负责数据实例管理，有涉及到数据实例迁移、库表、域名等变更，DBA应该提前通知数据侧，进行评估和提前准备

2）业务数据变更情况，如发生以下情况，应第一时间通知数据开发或数据产品，数据核对受影响的范围（比如dw/mart/app基础数据，报表和看板数据等），并进行修复

业务表新增字段（数据侧能自动检查处理）

业务表删除字段（依赖影响比较大，需要提前通知）

经过运维OPS平台提单，修复数据，DBA应该把相应的更新或删除SQL信息同步给数据侧，数据侧在进行相应的处理，此同步过程可以逐渐自动化工具化，目前还是人工处理和通知（时间待定）

不经过OPS平台，业务后台刷数据，数仓无法自动捕获，业务侧应该及时通知数据侧；后面运维统一收拢机器登录权限，或者指定只能在某些机器上进行操作，并对执行脚本进行监控，逐渐工具化

历史数据归档（归档后的新表有结构变更，历史表也需要同步）

业务枚举值变更或新增（反馈沟通+核心业务枚举值监控）

业务侧有重大版本变更，（比如像存管这类项目）涉及到数据的部分，应提前通知数据侧参与前期的需求沟通和审批，数据侧提前做好准备

建议：前期可以通过问题反馈沟通方式把流程推动运行起来，后面可逐渐自动化和工具化

事前接触

3）事前检查（业务开发）：数据侧提供依赖数据库表的check工具，提供页面查询或API接口，业务侧通过tapd固化check流程让开发确认，业务侧有数据变更操作周知数据侧

4）数据前置（数据产品）：业务侧涉及到数据相关的评审，数据团队也一起参加评审，并加入到后续的数据规划和考虑中，优先在重点项目上进行试点，钱包项目会按照这个方式进行执行，看执行效果并推广到其他项目

## 4.5 数据接入具体示例图

如下演示一个完整的数据接入案例

### 4.5.1 申请数据接入

登录数据接入地址 http://data.yingzhongtong.com/#/datax-add-index/

填写申请单

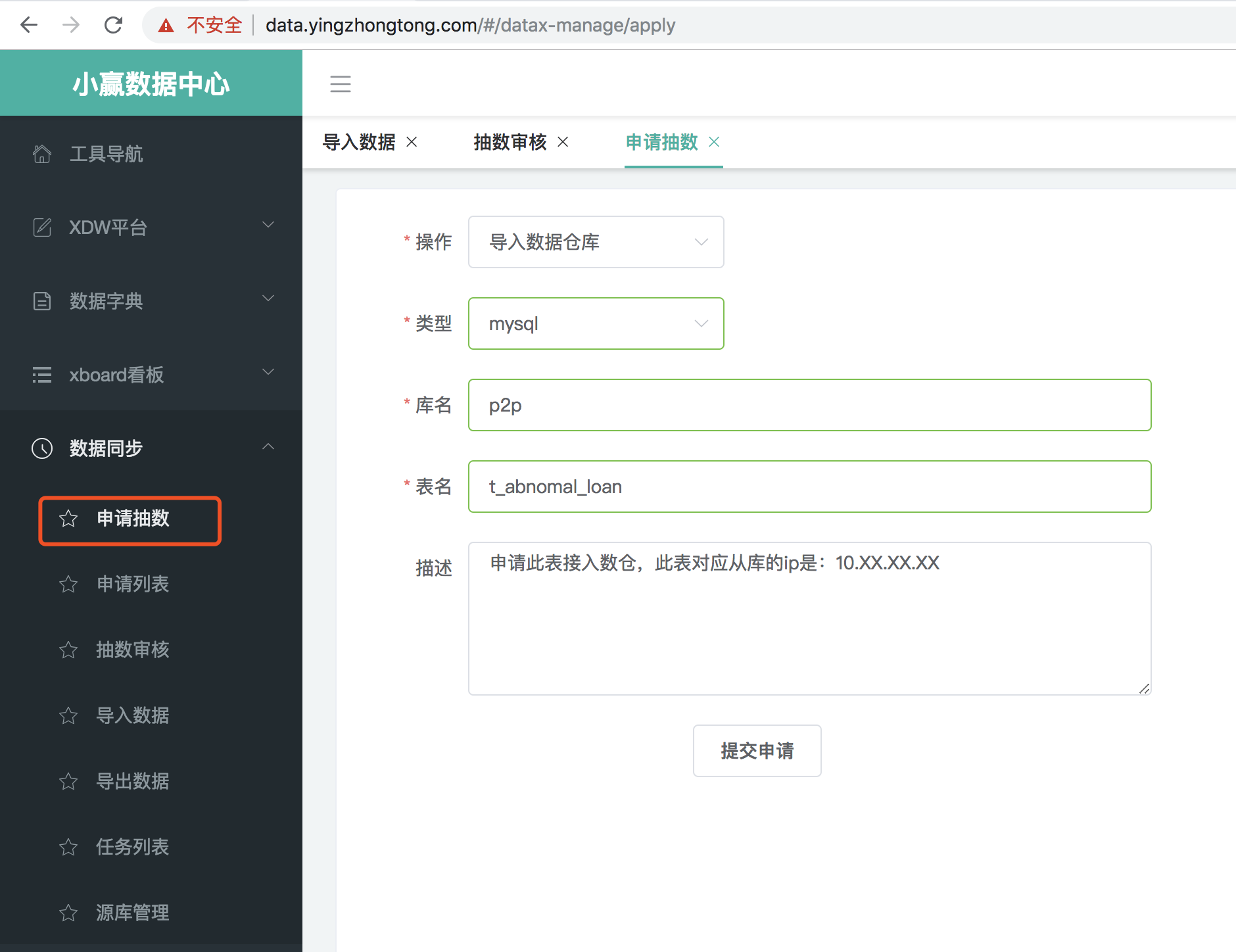


图 3 数据接入申请

### 4.5.2 申请数据列表

填写申请单之后，数仓操作人员可以看到申请列表单

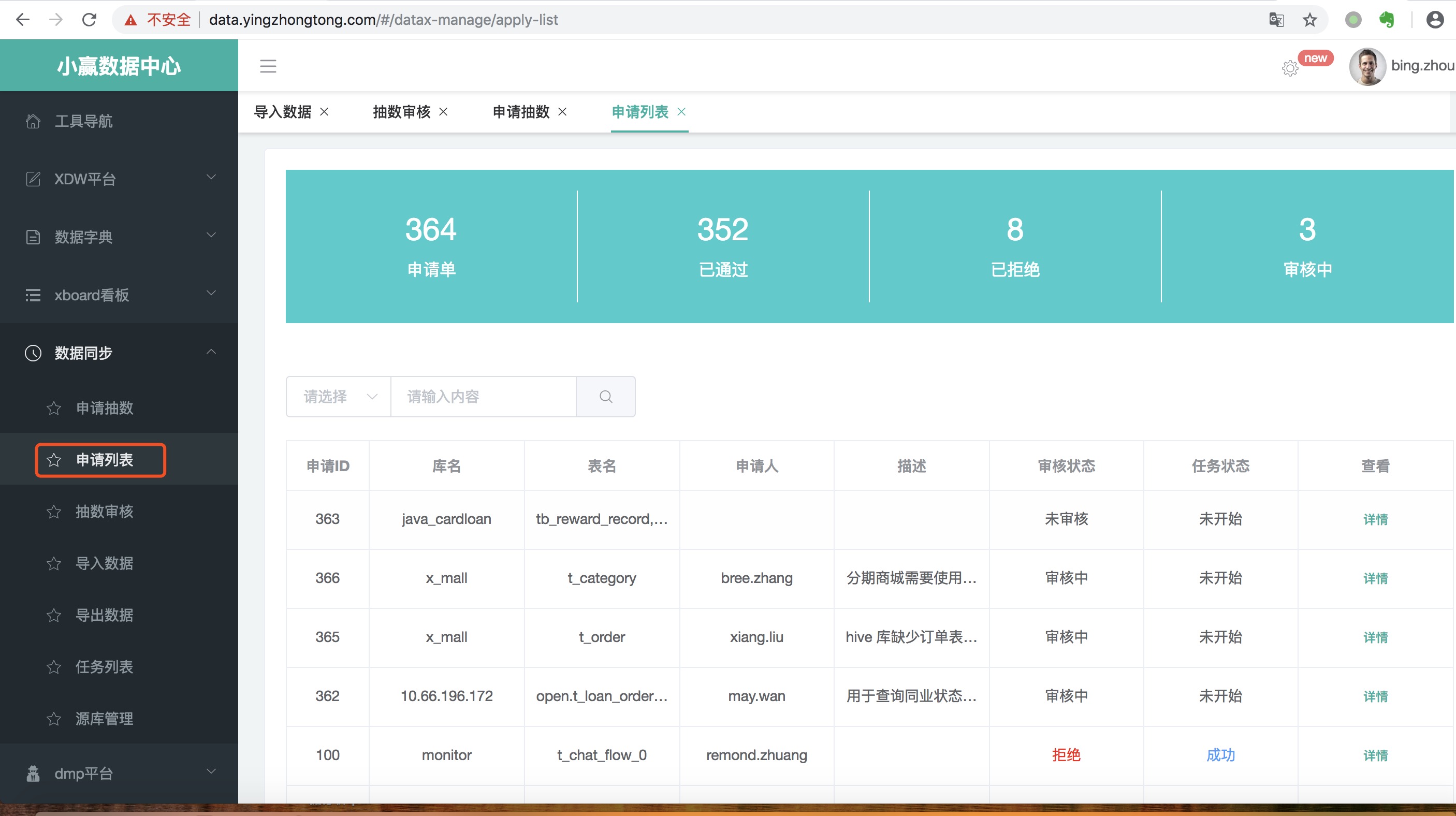


图 4 申请数据列表

### 4.5.3 开始接入数据

选好对应的从库加表名，选择好是天同步还是按小时同步

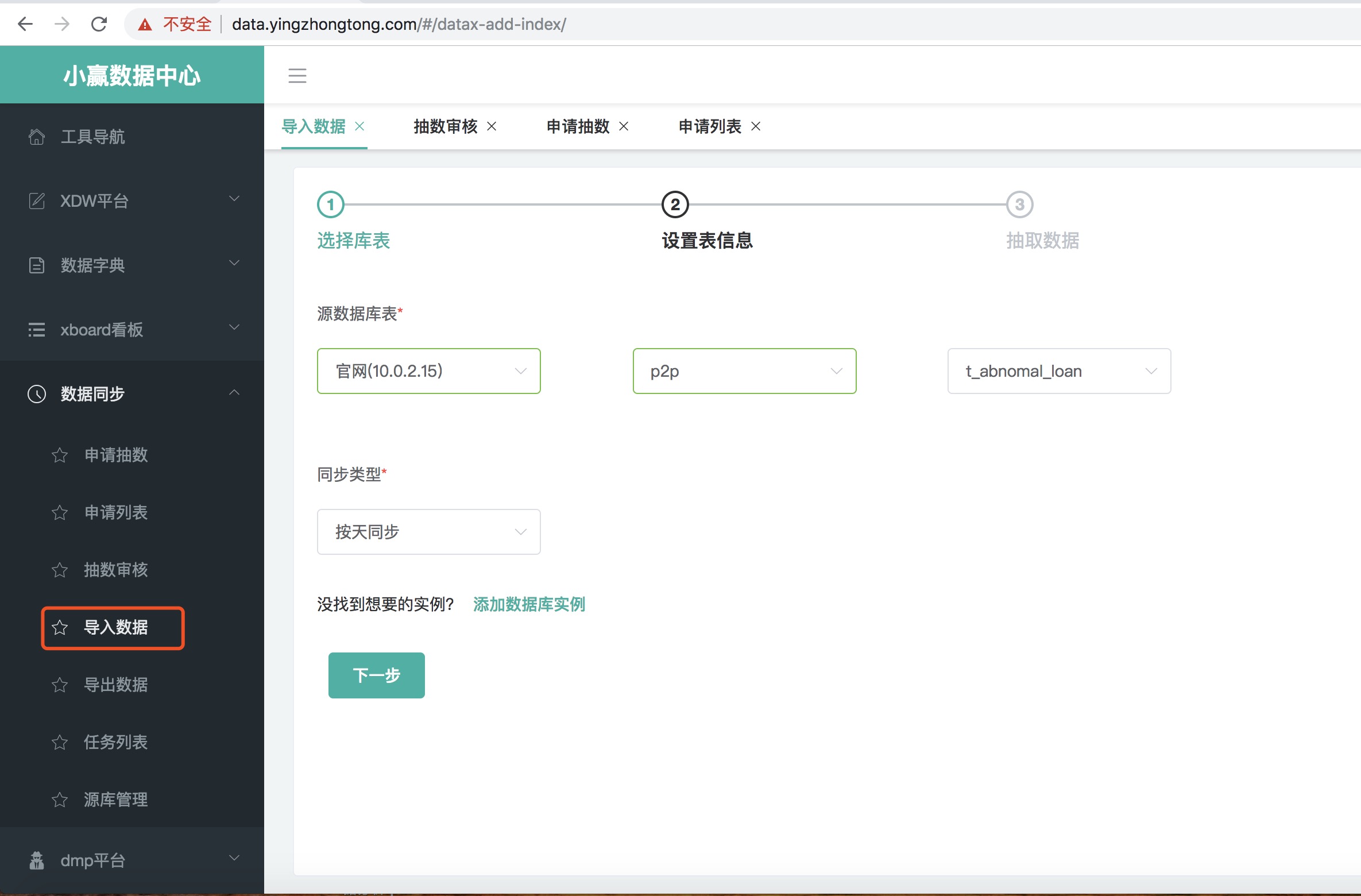


图 5 接入数据

### 4.5.4 选择映射关系

选择表映射关系，具体映射逻辑在4.3已经详细阐述

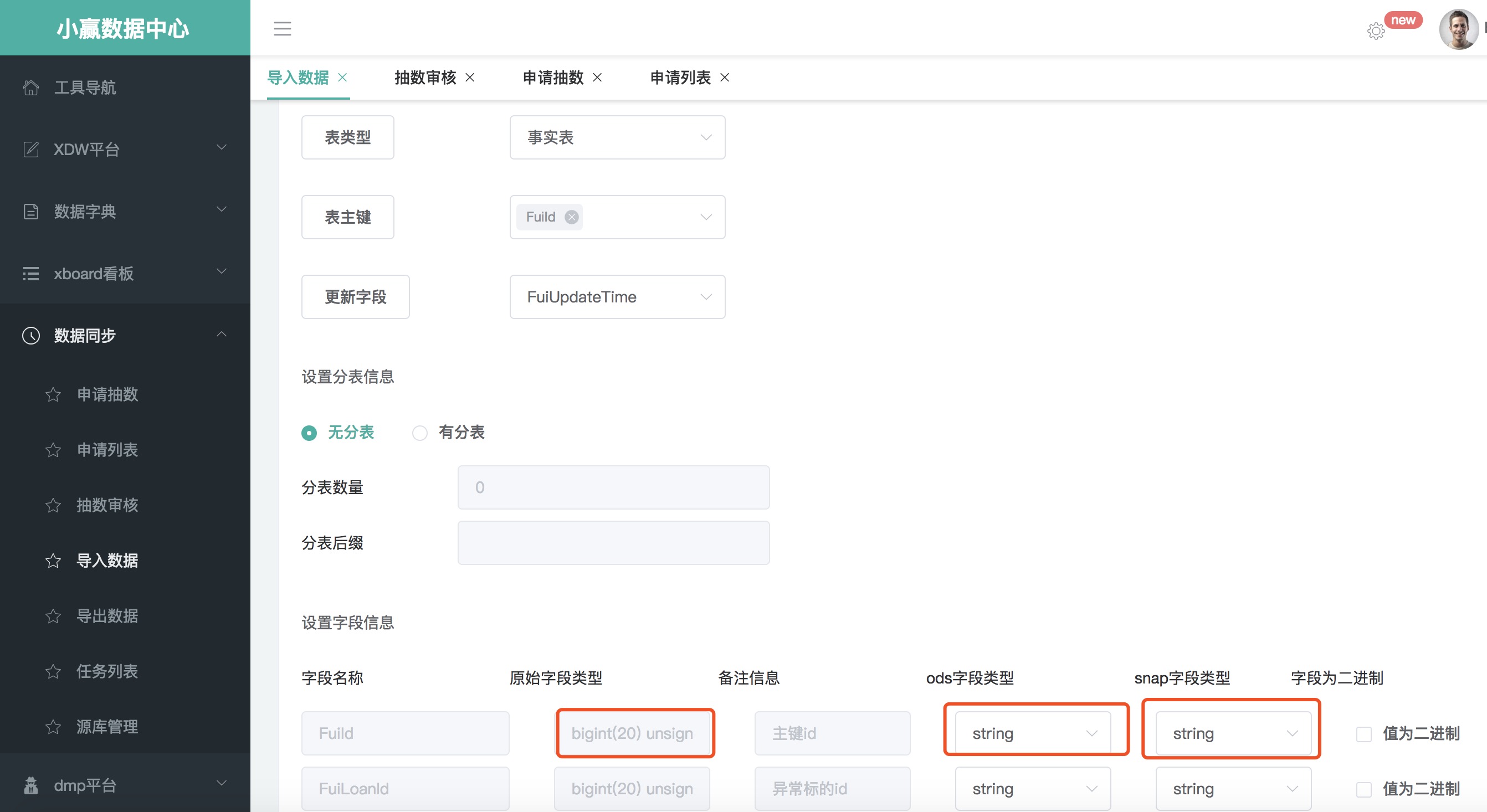


图 6 选择映射关系

### 4.5.5 抽取方式

初始化抽取，需要把存量数据抽取过来，因此要选择第一第二这两个选项

表结构变更抽取，需要创建新的表，因此要选择第一第二第三这三个选项

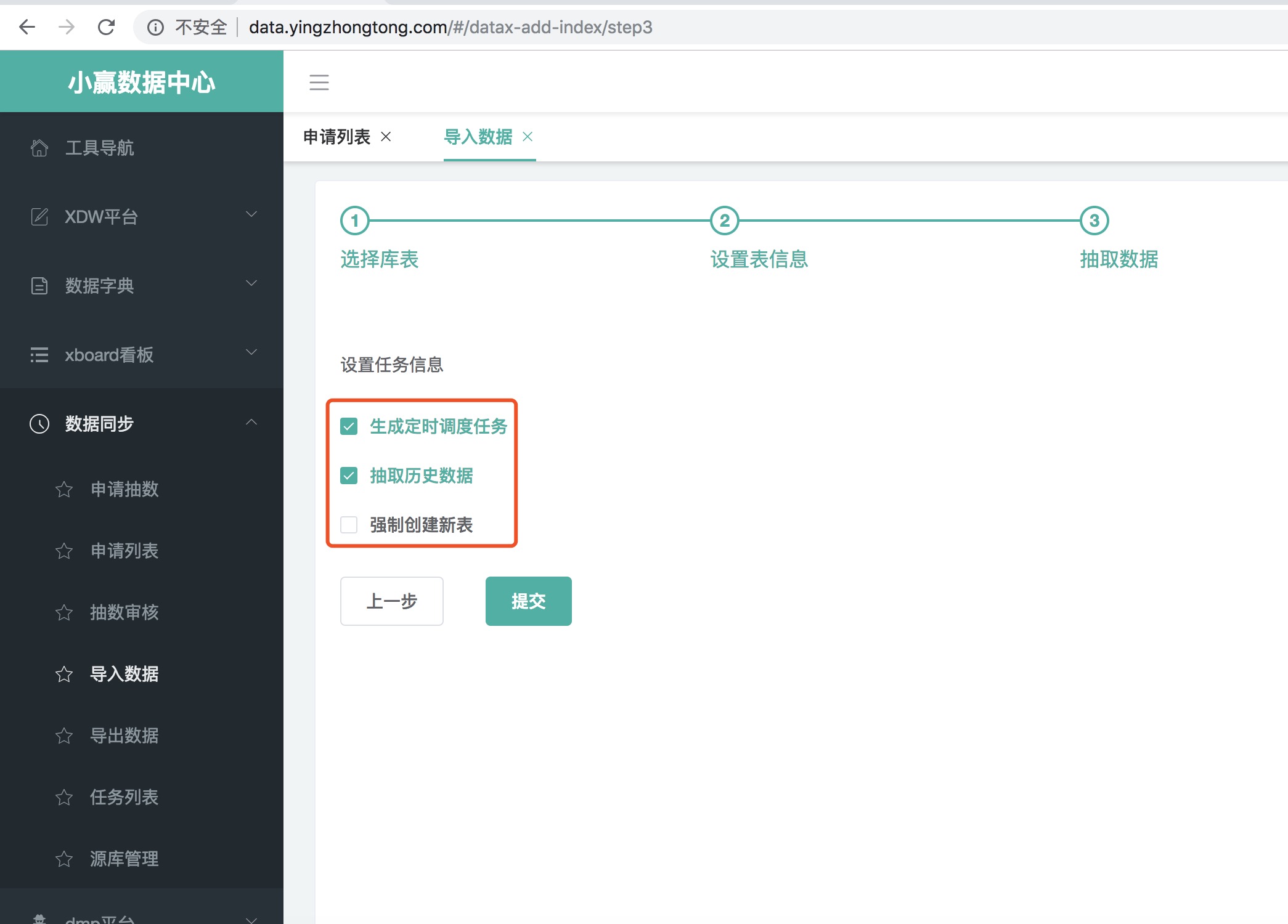


图 7 数据抽取方式

# 第五章 开发规范

## 5.1 合并小文件

数仓脚本执行，默认的引擎是sparksql，由于执行结果会产生较多的小文件，如果不及时处理，会导致查询变慢甚至拖垮整个数仓，因此在生成结果表之后务必对表进行小文件合并。

**分区表合并小文件示例**：

1. hive -hiveconf mapred.**max**.split.**size**=134217728 --hiveconf mapred.min.split.size=134217728 -e
2. "ALTER TABLE xy\_mart.user\_invest PARTITION (xy\_date='${startdate}') concatenate"

**非分区表合并小文件示例**：

1. hive -hiveconf mapred.max.split.size=134217728 --hiveconf mapred.min.split.size=134217728 -e
2. "ALTER TABLE xy\_mart.user\_invest concatenate"

## 5.2 ETL任务加大资源

数仓晚上跑批每个任务，默认分配的资源是一致的，对个别数据量较大计算较为复杂的sql，可酌情加大资源分配，具体参数调试找邹波

**示例，（只是参考模板，具体参数大小需找邹波协调）**

1. 定义变量spark\_cfg:
2. spark\_cfg=" --conf spark.sql.shuffle.partitions=1000 --num-executors 40 --executor-cores 1 --executor-memory 2g --conf spark.yarn.executor.memoryOverhead=1g "
4. 使用变量spark\_cfg:
5. $etl\_sql --name="${spark\_name}" ${spark\_cfg} -hiveconf xy\_date="${startdate}" -hiveconf xy\_today=$(date -d "${startdate} 1days" +"%Y%m%d") -f /opt/xy\_dw/bi\_etl/data\_biz/xy\_mart/a.sql

## 5.3 azkaban调用脚本的问题

azkaban在调用一个shell脚本的时候，只会判断此脚本最后一行的返回值是否是正确，如果正确则认为此脚本是正确的，哪怕此脚本中间会有部分错误的情况出现，因此，在脚本调用sql的之后，一定要紧接一个判断上一行脚本是否正确，如果错误，需要把错误抛出来

如下示例：

1. #!/bin/bash
2. ....
4. #第一个步骤
5. $etl\_sql --name="${spark\_name}" -hiveconf xy\_date="${xy\_date}" -f /opt/xy\_dw/bi\_etl/data\_biz/bj\_dw/a.sql
7. #第二个步骤
8. if [ $? != 0 ]; then
9. exit 1
10. fi
12. #第三个步骤
13. hive -hiveconf mapred.max.split.size=134217728  --hiveconf mapred.min.split.size=134217728  -e "ALTER TABLE a "

具体分析：

第一个步骤，调用脚本a.sql，第二个步骤，判断上个脚本是否执行正确，第三个步骤，合并小文件。  
如果没有第二个步骤，第一个步骤报错，但是第三个步骤正确的话，任务调度系统azkaban会认为此脚本正确执行。然而整个脚本最核心的第一个部分是最容易报错的地方，因为一定记得加上第二个步骤，对第一个步骤进行校验，如果报错，会把错误抛出给azkaban，提醒程序员

## 5.4 脚本开发规范

查表必须带库名，查询字段必须带表名

1）查表必须带库名，这个自己多试验几次就可以发现，有时候查询默认会跳到default库里面去了，很明显是一个误操作，所以查询必须带库名

2）要重点强调的是查字段必须带表名。

如脚本abc.sh:

1. **select** id,**name** **from** a left join b **on** a.a\_id=b.a\_id

此id字段来自a表，如果别人改了b表，给b增加了一个字段，名称也叫id的话，一般来说，别人测试完a表的脚本，不会来测这个abc.sql的脚本,这个时候你的abc.sh脚本就会报错。

这个就是典型的埋雷。不能够控制啥时候报错。

因此正确的写法是：

1. **select** a.id,a.**name** **from** a left join b **on** a.a\_id=b.a\_id

就算别人在b表加了id字段，你的脚本也不会报错。

3）脚本里面的\*号，其实\*，limit这些是禁止上生产的，现为了代码的优雅性，只允许特定的情况写\*号，别的情况禁止出现。

如上面的情况，多表关联，肯定是禁止\*号出现的

1. **select** \* **from** a left join b **on** a.a\_id=b.a\_id

这种脚本也会出现上面同样的错误。

只允许在单个表查询，且查询的字段里面不包含pb，查询的字段占到表字段70%以上才允许使用\*号

如下面这种情况

1. **select** 各种字段 **from**
2. (**select** \* **from** x **where** x.sex=0) a
3. left join b **on** a.id=b.id

上面这种情况是运行使用\*号的，下面这种情况不要使用\*号

比如你只需要查询t\_loan里面的一个id字段，你写成select \* from xy\_snap.p2p\_db\_t\_loan

\*号会把所有的字段捞出来，效率会很低很低。

其中limit是一定禁止上生产，控制行数用rownumber，测试时可用。

# 第六章 数仓开发必备技能

## 6.1 高频率开窗函数

数仓开发人员必会技能开窗函数，用来解决复杂的查询

源表：

表格 11 源表数据

|  |  |  |
| --- | --- | --- |
| area | date | sale |
| nanshan | 1 | 10 |
| nanshan | 2 | 20 |
| nanshan | 3 | 30 |
| futian | 2 | 40 |
| futian | 3 | 30 |

问题1:需要按区域统计每天累加值(当前记录的销售额等于所在区域按照日期排序的累加和)，得到如下的结果表：

表格 12 累加后的结果表

|  |  |  |  |
| --- | --- | --- | --- |
| area | date | sale | total |
| nanshan | 1 | 10 | 10 |
| nanshan | 2 | 20 | 30 |
| nanshan | 3 | 30 | 60 |
| futian | 2 | 40 | 40 |
| futian | 3 | 20 | 60 |

代码示例：

1. **select** t.\*, sum(sale) over (partition **by** area **order** **by** **date**) total
2. **from** t

问题2：当前销售额要等于所在区域内按日期排序的当前销售额，加上一条以及下一条的销售额，得到如下的结果表：

表格 13 跨行累加结果表

|  |  |  |  |
| --- | --- | --- | --- |
| area | date | sale | total |
| nanshan | 1 | 10 | 30 |
| nanshan | 2 | 20 | 50 |
| nanshan | 3 | 30 | 50 |
| futian | 2 | 40 | 60 |
| futian | 3 | 20 | 60 |

PS：第二条的total=50是来自第二条的sale=20加上上一条sale的10加上下一条sale的30

代码示例：

1. **select** t.\*, sum(sale) over (partition **by** area **order** **by** **date** **ROWS** ETWEEN 1 PRECEDING AND 1 FOLLOWING) total
2. **from** t

问题3：取每个区域的第一条销售记录，得到如下的结果表：

表格 14 组内取第一条

|  |  |  |
| --- | --- | --- |
| area | date | sale |
| nanshan | 1 | 10 |
| futian | 2 | 40 |

代码示例：

1. **select** \* **from** (
2. **select** t.\*,rownumber() over(partition **by** area **order** **by** **date** **asc**) rn
3. **from** t
4. )h **where** h.rn=1

问题4：取每一次销售记录，按时间排序，同时查看上一条的销售金额，为空默认补0，得到如下的结果表：

表格 15 跨行取值

|  |  |  |  |
| --- | --- | --- | --- |
| area | date | sale | last\_sale |
| nanshan | 1 | 10 | 0 |
| nanshan | 2 | 20 | 10 |
| nanshan | 3 | 30 | 20 |
| futian | 2 | 40 | 0 |
| futian | 3 | 20 | 40 |

代码示例：

1. **select** \* **from** (
2. **select** t.\*,lag(sale,1,0) over(partition **by** area **order** **by** **date** **asc**) rn
3. **from** t
4. )h **where** h.rn=1

ps:参数说明，1指的是取上一行的记录，写成3的话取上第三行的记录，0是默认值，一般第一个分区里面的的第一条是没有上一条记录的，补0

**常用的分析函数如下所列:**

1. row\_number() over(partition **by** ... **order** **by** ...)
2. rank() over(partition **by** ... **order** **by** ...)
3. dense\_rank() over(partition **by** ... **order** **by** ...)
4. count() over(partition **by** ... **order** **by** ...)
5. **max**() over(partition **by** ... **order** **by** ...)
6. **min**() over(partition **by** ... **order** **by** ...)
7. sum() over(partition **by** ... **order** **by** ...)
8. avg() over(partition **by** ... **order** **by** ...)
9. first\_value() over(partition **by** ... **order** **by** ...)
10. last\_value() over(partition **by** ... **order** **by** ...)
11. lag() over(partition **by** ... **order** **by** ...)
12. lead() over(partition **by** ... **order** **by** ...)

**分析函数经典模板**

1. --显示各部门员工的工资，并附带显示该部分的最高工资。
2. **SELECT** E.DEPTNO,
3. E.EMPNO,
4. E.ENAME,
5. E.SAL,
6. LAST\_VALUE(E.SAL)
7. OVER(PARTITION **BY** E.DEPTNO
8. **ORDER** **BY** E.SAL **ROWS**
9. --unbounded preceding and unbouned following针对当前所有记录的前一条、后一条记录，也就是表中的所有记录
10. --unbounded：不受控制的，无限的
11. --preceding：在...之前
12. --following：在...之后
13. BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) MAX\_SAL
14. **FROM** EMP E;

## 6.2 大数据SQL与传统关系型数据库SQL的异同点

大数据hadoop主要解决了两个问题，大数据量的存储（hdfs）以及大数据量的计算（MapReduce），但是发现用户自己写MapReduce效率极低，所以才对外提供SQL接口，因此大数据的sql与传统sql是存在大量异同的。

### 6.2.1 大数据SQL不支持delete

表A,五条记录

表格 16 源表结构1

|  |
| --- |
| id |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

表B,三条记录

表格 17 源表结构2

|  |
| --- |
| id |
| 1 |
| 2 |
| 3 |

现在希望得到A-B={4，5}

**传统SQL的解决方案**：直接在A表里面删除B表里面的数据，代码如下：

1. **delete** A **from** A,B **where** A.id=B.id

**大数据SQL的解决方案**：A,B两个表关联，过滤掉数据，把结果插入C表，删除A表，把C表rename回A表，代码如下：

1. **create** **table** C stored **as** orc **as**
2. **select**
3. A.id
4. **from** A
5. left join B **on** A.id=B.id
6. **where** B.id **is** null
7. ;
9. 得到的C={4,5}
10. **drop** **table** A;
11. **alter** **table** C rename **to** A ;

**ps:传统的数据库OLTP讲究的是用事务去控制毫秒级别的sql响应，大数据OLAP讲究的是大数据量批次化的挪动与转化关系。**

### 6.2.2 大数据SQL不支持update

传统SQL的解决方案：直接使用update更新一条记录

大数据SQL的解决方案：通过case when去更新记录，生成新的结果表C

1. **create** **table** C **as**
2. **select**
3. A.id,
4. (case **when** b.id **is** not null **then** a.age+1 **else** a.age **end** ) age
5. **from** A
6. left join B **on** A.id=B.id
7. ;
8. **drop** **table** A;
9. **alter** **table** C rename **to** A

6.2.3 数据倾斜

传统数据库查询关注sql是否索引失效导致全表扫描，大数据SQL关注是否数据倾斜，没有充分利用集群的优势一起计算，导致个别机器单独在计算别的机器都在挂起。

什么是数据倾斜？（也可直接跳到最后面看解决方案）  
数据倾斜在MapReduce编程模型中十分常见,用最通俗易懂的话来说,数据倾斜无非就是大量的相同key被partition分配到一个分区里,造成了'一个人累死,其他人闲死'的情况,这种情况是我们不能接受的,这也违背了并行计算的初衷,首先一个节点要承受着巨大的压力,而其他节点计算完毕后要一直等待这个忙碌的节点,也拖累了整体的计算时间,可以说效率是十分低下的。  
用户侧的感知情况： sql运行很久，甚至几个小时，后台一直卡在99%

数据倾斜发生的场景?  
在join 后，group by后等情况下，同一个key值数量太多导致数据倾斜

具体分析：

1. **select** id,usename
2. **from** a
3. left join b **on** a.username=b.username

这个sql，如果 a.username 里面存在大量的null，就会发生数据倾斜。  
传统的mr解决方式有两种：  
第一种：set hive.groupby.skewindata=true(默认关闭)，此时hive的执行在MR后台会存在两个map一个reduce，第一个map本质上就是先对数据进行shuffle，第二个map就可以对shuffle之后的数据进行操作。  
建议用这种，如果sql卡在99%很久没有完成的情况下。因为不同的开发人员水平不一样，使用了这个之后，虽然整体性能降低了，但是可以让开发人员无感知的去完成查询且不出现数据倾斜，不长久占用集群资源。  
第二种：null值导致倾斜，巧妙利用sql解决

方法一：user\_id为空的不参与关联

1. **Select** \*
2. **From** log a
3. Join bmw\_users b **On** a.user\_id **is** not null And a.user\_id = b.user\_id
4. **Union** all
5. **Select** \*
6. **from** log a
7. **where** a.user\_id **is** null;

方法二：赋与空值分新的key值

1. **Select** \*
2. **from** log a
3. left outer join bmw\_users b **on** case **when** a.user\_id **is** null **then** concat('dp\_hive',rand() ) **else** a.user\_id **end** = b.user\_id;

但是，我们现在的集群不是跑的MapReduce，而是sparksql，它不支持这种case when a.user\_id is null then concat('dp\_hive',rand() ) else a.user\_id end = b.user\_id; 语法需要稍作修改。

解决方案：

倾斜的sql：

1. **Select** \*
2. **From** log a
3. Join b **On** a.user\_id = b.user\_id
4. 正确的使用方式：
5. **select** \* **from** (
6. **select**
7. a.id,
8. a.username,
9. nvl(a.user\_id,concat('xy\_dw',rand())) join\_key1
10. **from** log a
11. ) t
12. Join b **on** t.join\_key1=b.user\_id;

总结：

其实为null的值，压根与后面的表是关联不上的，但是程序是死的，它并不知道自己去判别，所以只能手动给它生成随机值让后面的表的id与前面的表关联不上即可，打散key值分布，让每台机器都运作起来。

6.2.4 sparksql禁止使用not in或者in大量集合

sparksql禁止使用in或者not in（除非在10个以内的集合），否则尽量使用join之后is null或者is not null的方式去解决

如下示例是使用了not in 无法跑出结果的情况：

1. **select**
2. user\_id,
3. CASE **WHEN** user\_source=1 **THEN** '卡贷' **WHEN** user\_source=2 **THEN** '易贷' **WHEN** user\_source=4 **THEN** '卡贷' **WHEN** user\_source=5 **THEN** '卡贷'
4. **WHEN** user\_source=6 **THEN** '易贷' **WHEN** user\_source=7 **THEN** '薪贷' **WHEN** user\_source=10 **THEN** '精英贷' **else** null **END** **as** product,
5. channel\_id
6. **from** xy\_snap.java\_cardloan\_db\_tb\_user\_info t
7. **where** t.user\_id not in
8. ( **select** fuiuid **from** xy\_dw.dim\_user\_registered\_products t **where** t.fstrregisteruser !='小赢普惠' ) and user\_source in (1,2,3,4,5,6,7,8,9,10)

优化后的解决方案：

1. **select**
2. user\_id,
3. CASE **WHEN** user\_source=1 **THEN** '卡贷' **WHEN** user\_source=2 **THEN** '易贷' **WHEN** user\_source=4 **THEN** '卡贷' **WHEN** user\_source=5 **THEN** '卡贷'
4. **WHEN** user\_source=6 **THEN** '易贷' **WHEN** user\_source=7 **THEN** '薪贷' **WHEN** user\_source=10 **THEN** '精英贷' **else** null **END** **as** product,
5. channel\_id
6. **from** xy\_snap.java\_cardloan\_db\_tb\_user\_info t
7. left join xy\_dw.dim\_user\_registered\_products b **on** b.fstrregisteruser !='小赢普惠' and t.user\_id=b.fuiuid
8. **where** user\_source in (1,2,3,4,5,6,7,8,9,10)  and b.fuiuid **is** null

6.2.5 敏捷开发，合理使用分组字段

对于较复杂的sql，部分分组字段取数复杂的情况下，可以使用group by 1,2这种语法提升代码的可读性

源sql:

1. **select**
2. case **when** col='1' **then** 'a'
3. **when** col='2' **then** 'b'
4. **when** col='3' **then** 'c'
5. **when** col='4' **then** 'd'
6. **end** xx,
7. sum(money) total\_money
8. **from** t
9. **group** **by**
10. case **when** col='1' **then** 'a'
11. **when** col='2' **then** 'b'
12. **when** col='3' **then** 'c'
13. **when** col='4' **then** 'd'
14. **end**

优化后的sql:

1. **select**
2. case **when** col='1' **then** 'a'
3. **when** col='2' **then** 'b'
4. **when** col='3' **then** 'c'
5. **when** col='4' **then** 'd'
6. **end** xx,
7. sum(money) total\_money
8. **from** t
9. **group** **by** 1

总结，在特别复杂的sql下可以使用这种情况使自己的代码变的易看易用且简洁。

6.2.6 数仓金额字段运算

数仓金额字段运算，务必使用decimal类型

譬如：

1. **select** A/B   ;

sparksql内部会把A跟B转成double去运算，这样会精度丢失，得出的结果不正确。  
正确的做法是：

1. **select** cast(A **as** **DECIMAL**(38,5)) / cast(B **as** **DECIMAL**(38,5)) ;