

# 数仓建设保姆级教程

离线和实时一网打尽(理论+实战)

V2.0

本文档来自公众号：五分钟学大数据

微信直接扫码关注

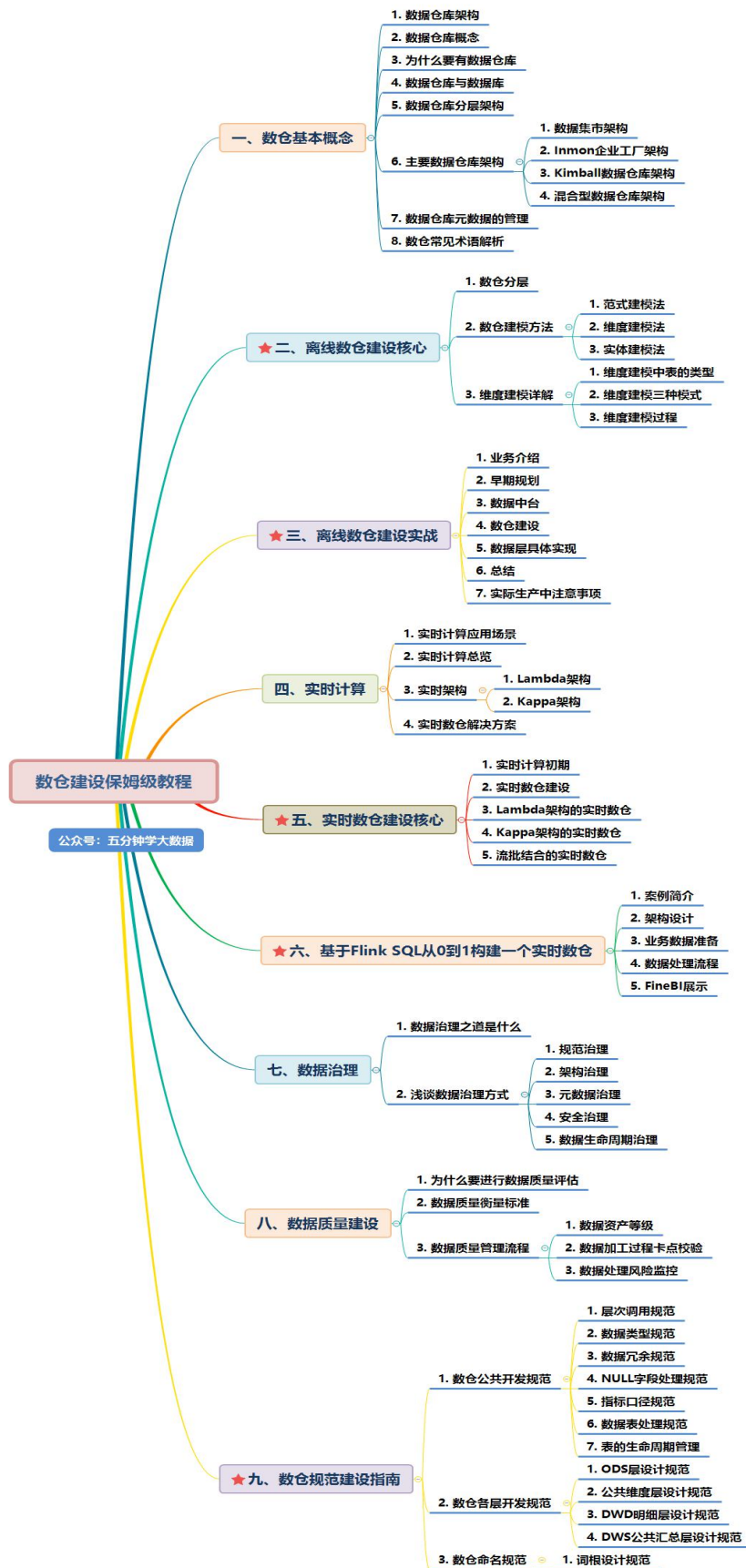


## 目录

本文涉及知识点如下图所示:	4
一、数仓基本概念	5
1. 数据仓库架构	5
2. 数据仓库概念	5
3. 为什么要有数据仓库	8
4. 数据仓库与数据库的区别	8
5. 数据仓库分层架构	9
6. 主要数据仓库架构	11
7. 数据仓库元数据的管理	14
8. 数仓常见术语解析	15
1. 数仓名词解释	16
2. 数仓名词之间关系	20
二、离线数仓建设核心	22
1. 数仓分层	23
2. 数仓建模方法	25
3. 维度建模详解	28
1. 维度建模中表的类型	28
2. 维度建模三种模式	33
3. 维度建模过程	35
三、离线数仓建设实战	37
1. 业务介绍	37
2. 早期规划	37
3. 数据中台	38
4. 数仓建设	41
5. 数据层具体实现	43
6. 总结	46
7. 实际生产中注意事项	47
四、实时计算	47
1. 实时计算应用场景	48
2. 实时计算总览	49
3. 实时架构	52
4. 实时数仓解决方案	55
五、实时数仓建设核心	57
1. 实时计算初期	57
2. 实时数仓建设	58
3. Lambda 架构的实时数仓	60
4. Kappa 架构的实时数仓	61
5. 流批结合的实时数仓	62
六、基于 Flink SQL 从 0 到 1 构建一个实时数仓	62
1. 案例简介	63

2. 架构设计.....	63
3. 业务数据准备.....	63
4. 数据处理流程.....	66
5. FineBI 展示.....	86
七、数据治理.....	86
1. 数据治理之道是什么.....	87
2. 浅谈数据治理方式.....	88
猜你喜欢: .....	95
八、数据质量建设.....	95
1. 为什么要进行数据质量评估.....	96
2. 数据质量衡量标准.....	96
3. 数据质量管理流程.....	98
4. 最后.....	105
九、数仓规范建设指南.....	105
1. 数仓公共开发规范.....	105
2. 数仓各层开发规范.....	111
3. 数仓命名规范.....	115
参考文档: .....	119
最后:.....	119

本文涉及知识点如下图所示：

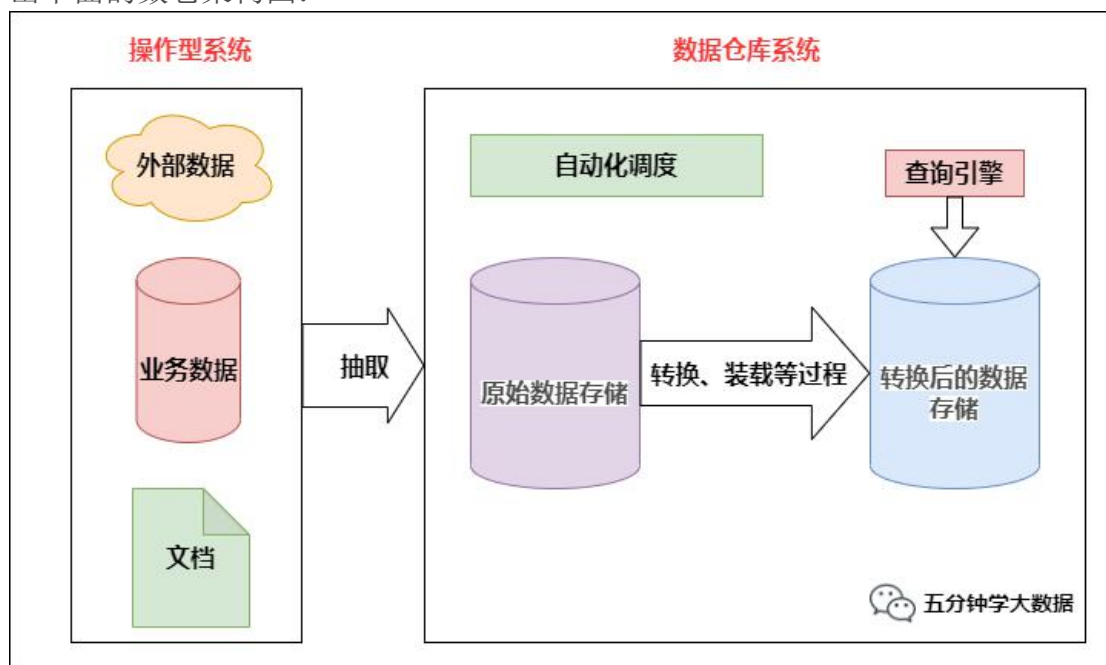


## 一、数仓基本概念

### 1. 数据仓库架构

我们在谈数仓之前，为了让大家有直观的认识，先来谈数仓架构，“**架构**”是什么？这个问题从来就没有一个准确的答案。这里我们引用一段话：在软件行业，一种被普遍接受的架构定义是指系统的一个或多个结构。结构中包括软件的构建（构建是指软件的设计与实现），构建的外部可以看到属性以及它们之间的相互关系。

这里参考此定义，把数据仓库架构理解成构成数据仓库的组件及其之间的关系，画出下面的数仓架构图：



#### 数仓架构

上图中显示的整个数据仓库环境包括操作型系统和数据仓库系统两大部分。操作型系统的数据由各种形式的业务数据组成，这些数据经过抽取、转换和装载(ETL)过程进入数据仓库系统。

任何事物都是随着时间的演进变得越来越完善，当然也是越来越复杂，数仓也不例外。在数据仓库技术演化过程中，产生了几种主要的架构方法，包括**数据集市架构**、**Inmon 企业信息工厂架构**、**Kimball 数据仓库架构**、**混合型数据仓库架构**。这几种架构我们后面再讲，接下来看下数仓的基本概念。

### 2. 数据仓库概念

英文名称为 Data Warehouse，可简称为 DW 或 DWH。数据仓库的目的是构建面向分析的集成化数据环境，为企业提供决策支持（Decision Support）。它出于分析性报告和决策支持目的而创建。

数据仓库本身并不“生产”任何数据，同时自身也不需要“消费”任何的数据，数据来源于外部，并且开放给外部应用，这也是为什么叫“仓库”，而不叫“工厂”的原因。

## 1) 基本特征

数据仓库是面向主题的、集成的、非易失的和时变的数据集合，用以支持管理决策。

### 1. 面向主题：

传统数据库中，最大的特点是面向应用进行数据的组织，各个业务系统可能是相互分离的。而数据仓库则是面向主题的。主题是一个抽象的概念，是较高层次上企业信息系统中的数据综合、归类并进行分析利用的抽象。在逻辑意义上，它是对应企业中某一宏观分析领域所涉及的分析对象。

### 2. 集成性：

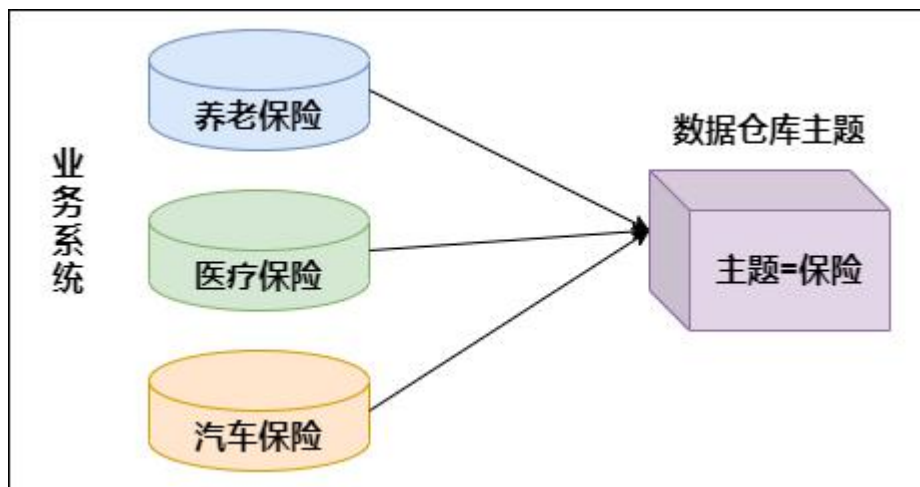
通过对分散、独立、异构的数据库数据进行抽取、清理、转换和汇总便得到了数据仓库的数据，这样保证了数据仓库内的数据关于整个企业的一致性。

数据仓库中的综合数据不能从原有的数据库系统直接得到。因此在数据进入数据仓库之前，必然要经过统一与综合，这一步是数据仓库建设中最关键、最复杂的一步，所要完成的工作有：

- 要统一源数据中所有矛盾之处，如字段的同名异义、异名同义、单位不统一、字长不一致，等等。
- 进行数据综合和计算。数据仓库中的数据综合工作可以在从原有数据库抽取数据时生成，但许多是在数据仓库内部生成的，即进入数据仓库以后进行综合生成的。

下图说明一个保险公司综合数据的简单处理过程，其中数据仓库中与“保险”主题有关的数据来自于多个不同的操作型系统。这些系统内部数据的命名可能不

同，数据格式也可能不同。把不同来源的数据存储到数据仓库之前，需要去除这些不一致。



数仓主题

### 3. 非易失性（不可更新性）：

数据仓库的数据反映的是一段相当长的时间内**历史数据的内容**，是不同时点的数据库快照的集合，以及基于这些快照进行统计、综合和重组的导出数据。

数据非易失性主要是针对应用而言。数据仓库的用户对数据的操作大多是数据查询或比较复杂的挖掘，一旦数据进入数据仓库以后，一般情况下被较长时间保留。数据仓库中一般有大量的查询操作，但修改和删除操作很少。因此，**数据经加工和集成进入数据仓库后是极少更新的，通常只需要定期的加载和更新。**

### 4. 时变性：

数据仓库包含各种粒度的历史数据。数据仓库中的数据可能与某个特定日期、星期、月份、季度或者年份有关。数据仓库的目的是通过分析企业过去一段时间业务的经营状况，挖掘其中隐藏的模式。虽然**数据仓库的用户不能修改数据，但并不是说数据仓库的数据是永远不变的**。分析的结果只能反映过去的情况，当业务变化后，挖掘出的模式会失去时效性。因此数据仓库的数据需要更新，以适应决策的需要。从这个角度讲，数据仓库建设是一个项目，更是一个过程。数据仓库的数据随时间的变化表现在以下几个方面：

- （1） 数据仓库的数据时限一般要远远长于操作型数据的数据时限。
- （2） 操作型系统存储的是当前数据，而数据仓库中的数据是历史数据。
- （3） 数据仓库中的数据是按照时间顺序追加的，它们都带有时间属性。



### 3. 为什么要有数据仓库

先来看下数据仓库的数据从哪里来，最终要到哪里去？

通常数据仓库的数据来自各个业务应用系统。业务系统中的数据形式多种多样，可能是 Oracle、MySQL、SQL Server 等关系数据库里的结构化数据，可能是文本、CSV 等平面文件或 Word、Excel 文档中的数据，还可能是 HTML、XML 等自描述的半结构化数据。这些业务数据经过一系列的数据抽取、转换、清洗，最终以一种统一的格式装载进数据仓库。数据仓库里的数据作为分析用的数据源，提供给后面的即席查询、分析系统、数据集市、报表系统、数据挖掘系统等。

这时我们就想了，为什么不能把业务系统的数据直接拿来供即席查询、分析系统、报表系统等使用呢，为什么要经过数据仓库这一步？实际上在数仓出现之前，确实是这么做的，但是有很多数据分析的先驱者当时已经发现，简单的“直接访问”方式很难良好工作，这样做的失败案例数不胜数。下面列举一些直接访问业务系统无法工作的原因：

- 某些业务数据由于安全或其他因素不能直接访问。
- 业务系统的版本变更很频繁，每次变更都需要重写分析系统并重新测试。
- 很难建立和维护汇总数据来源于多个业务系统版本的报表。
- 业务系统的列名通常是硬编码，有时仅仅是无意义的字符串，这让编写分析系统更加困难。
- 业务系统的数据格式，如日期、数字的格式不统一。
- 业务系统的表结构为事务处理性能而优化，有时并不适合查询与分析。
- 没有适当的方式将有价值的数据合并进特定应用的数据库。
- 没有适当的位置存储元数据。
- 用户需要看到的显示数据字段，有时在数据库中并不存在。
- 通常事务处理的优先级比分析系统高，所以如果分析系统和事务处理运行在同一硬件之上，分析系统往往性能很差。
- 有误用业务数据的风险。
- 极有可能影响业务系统的性能。

尽管需要增加软硬件的投入，但建立独立数据仓库与直接访问业务数据相比，无论是成本还是带来的好处，这样做都是值得的。随着处理器和存储成本的逐年降低，数据仓库方案的优势更加明显，在经济上也更具可行性。

### 4. 数据仓库与数据库的区别

数据库与数据仓库的区别实际讲的是 OLTP 与 OLAP 的区别。



操作型处理，叫联机事务处理 OLTP (On-Line Transaction Processing, )，也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理，像 Mysql, Oracle 等关系型数据库一般属于 OLTP。

分析型处理，叫联机分析处理 OLAP (On-Line Analytical Processing) 一般针对某些主题的历史数据进行分析，支持管理决策。

首先要明白，数据仓库的出现，并不是要取代数据库。数据库是面向事务的设计，数据仓库是面向主题设计的。数据库一般存储业务数据，数据仓库存储的一般是历史数据。

数据库设计是尽量避免冗余，一般针对某一业务应用进行设计，比如一张简单的 User 表，记录用户名、密码等简单数据即可，符合业务应用，但是不符合分析。数据仓库在设计是有意引入冗余，依照分析需求，分析维度、分析指标进行设计。数据库是为捕获数据而设计，数据仓库是为分析数据而设计。

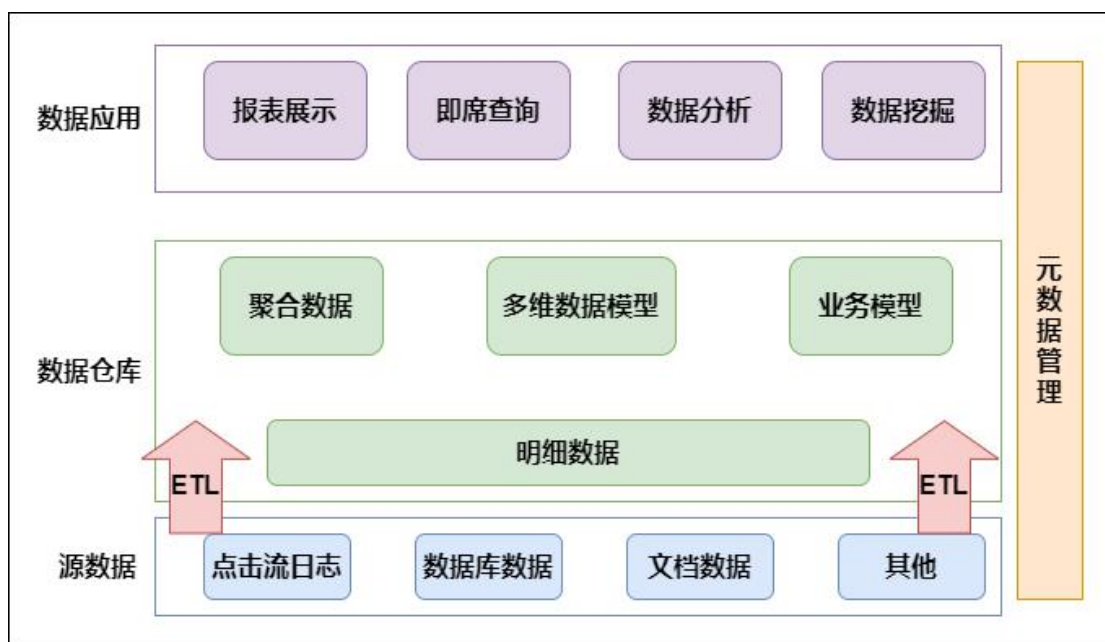
以银行业务为例。数据库是事务系统的数据平台，客户在银行做的每笔交易都会写入数据库，被记录下来，这里，可以简单地理解为用数据库记账。数据仓库是分析系统的数据平台，它从事务系统获取数据，并做汇总、加工，为决策者提供决策的依据。比如，某银行某分行一个月发生多少交易，该分行当前存款余额是多少。如果存款又多，消费交易又多，那么该地区就有必要设立 ATM 了。

显然，银行的交易量是巨大的，通常以百万甚至千万次来计算。事务系统是实时的，这就要求时效性，客户存一笔钱需要几十秒是无法忍受的，这就要求数据库只能存储很短一段时间的数据。而分析系统是事后的，它要提供关注时间段内所有的有效数据。这些数据是海量的，汇总计算起来也要慢一些，但是，只要能够提供有效的分析数据就达到目的了。

数据仓库，是在数据库已经大量存在的情况下，为了进一步挖掘数据资源、为了决策需要而产生的，它决不是所谓的“大型数据库”。

## 5. 数据仓库分层架构

按照数据流入流出的过程，数据仓库架构可分为：源数据、数据仓库、数据应用



## 数据仓库

数据仓库的数据来源于不同的源数据，并提供多样的数据应用，数据自下而上流入数据仓库后向上层开放应用，而数据仓库只是中间集成化数据管理的一个平台。

**源数据：**此层数据无任何更改，直接沿用外围系统数据结构和数据，不对外开放；为临时存储层，是接口数据的临时存储区域，为后一步的数据处理做准备。

**数据仓库：**也称为细节层，DW 层的数据应该是一致的、准确的、干净的数据，即对源系统数据进行了清洗（去除了杂质）后的数据。

**数据应用：**前端应用直接读取的数据源；根据报表、专题分析需求而计算生成的数据。

数据仓库从各数据源获取数据及在数据仓库内的数据转换和流动都可以认为是 ETL（**抽取 Extra**，**转化 Transfer**，**装载 Load**）的过程，ETL 是数据仓库的流水线，也可以认为是数据仓库的血液，它维系着数据仓库中数据的新陈代谢，而数据仓库日常的管理和维护工作的大部分精力就是保持 ETL 的正常和稳定。

## 那么为什么要数据仓库进行分层呢？

- **用空间换时间**，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据；不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大。
- 通过数据分层管理可以**简化数据清洗的过程**，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，

这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

## 6. 主要数据仓库架构

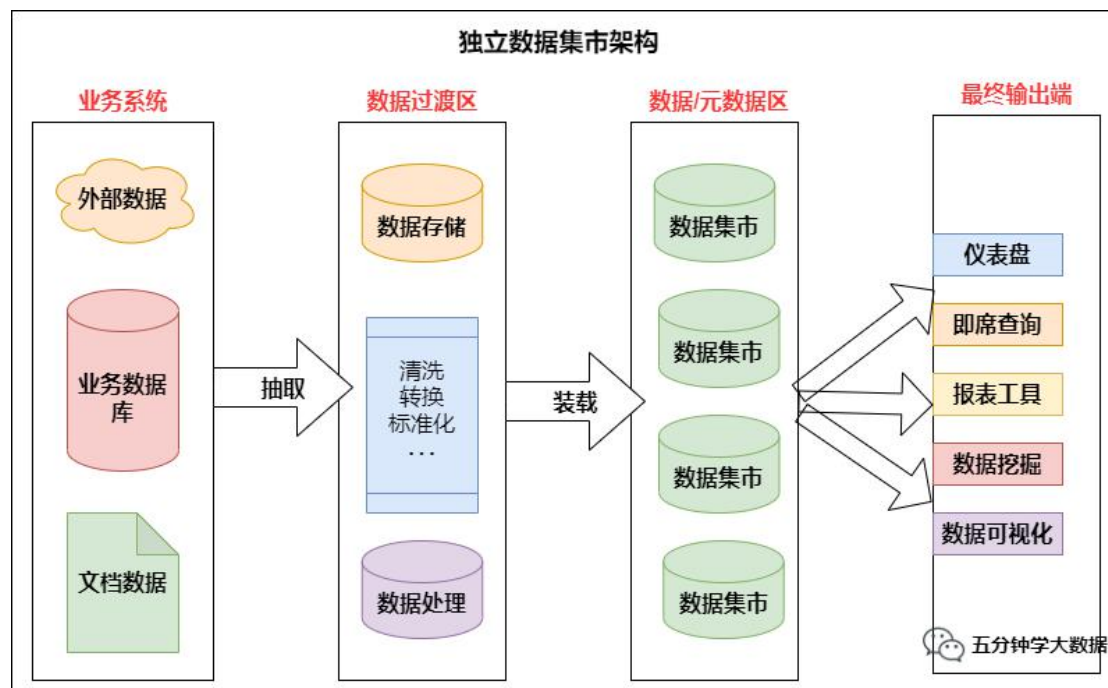
通过上面的内容我们大概了解数仓的概念，接下来就看下数仓的几种演进架构。

### 1. 数据集市架构

数据集市是按主题域组织的数据集合，用于支持部门级的决策。有两种类型的数据集市：**独立数据集市**和**从属数据集市**。

#### 1) 独立数据集市

独立数据集市集中于部门所关心的单一主题域，数据以**部门**为基础部署，无须考虑企业级别的信息共享与集成。例如，制造部门、人力资源部门和其他部门都各自有他们自己的数据集市。



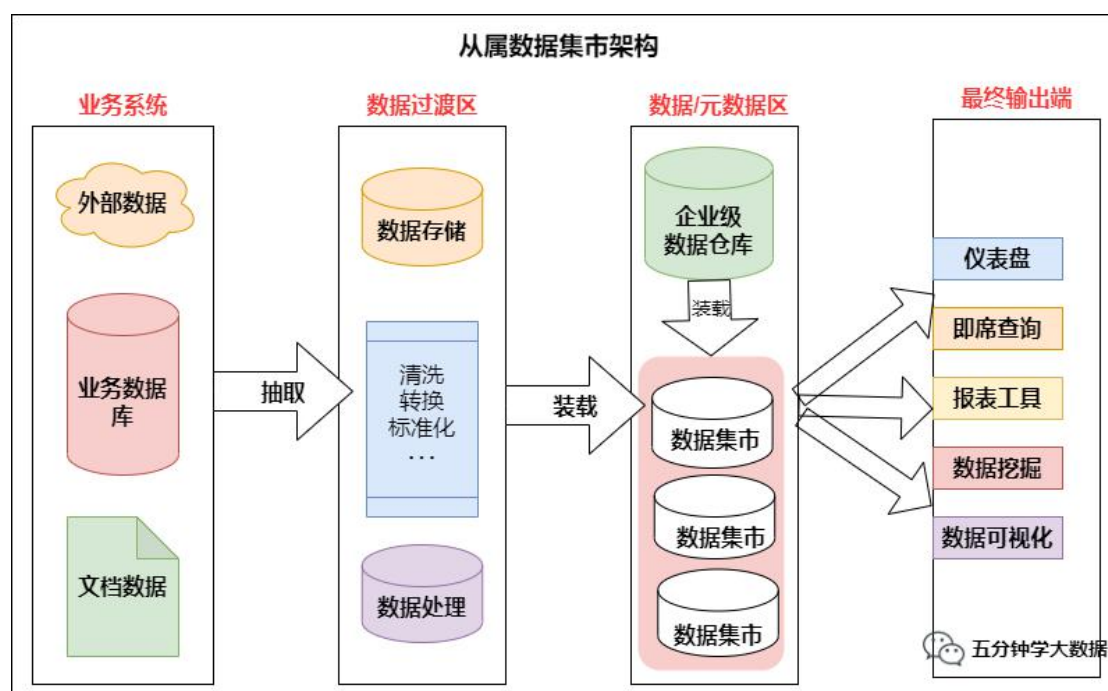
**优点：** 因为一个部门的业务相对于整个企业要简单，数据量也小得多，所以部门的独立数据集市具有周期短、见效快的特点。

**缺点：**

- 从业务角度看，当部门的分析需求扩展，或者需要分析跨部门或跨主题域的数据时，独立数据市场会显得力不从心。
- 当数据存在歧义，比如同一个产品，在 A 部门和 B 部门的定义不同时，将无法在部门间进行信息比较。
- 每个部门使用不同的技术，建立不同的 ETL 的过程，处理不同的事务系统，而在多个独立的数据集市之间还会存在数据的交叉与重叠，甚至会有数据不一致的情况。

## 2) 从属数据集市

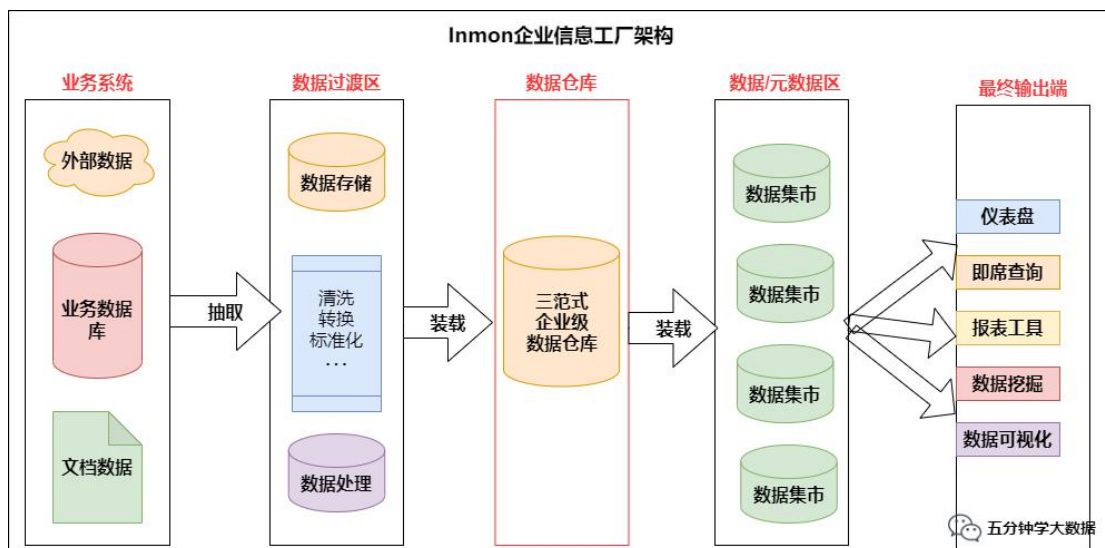
从属数据集市的数据来源于数据仓库。数据仓库里的数据经过整合、重构、汇总后传递给从属数据集市。



建立从属数据集市的好处主要有：

- **性能**：当数据仓库的查询性能出现问题，可以考虑建立几个从属数据集市，将查询从数据仓库移出到数据集市。
- **安全**：每个部门可以完全控制他们自己的数据。
- **数据一致**：因为每个数据集市的数据来源都是同一个数据仓库，有效消除了数据不一致的情况。

## 2. Inmon 企业工厂架构

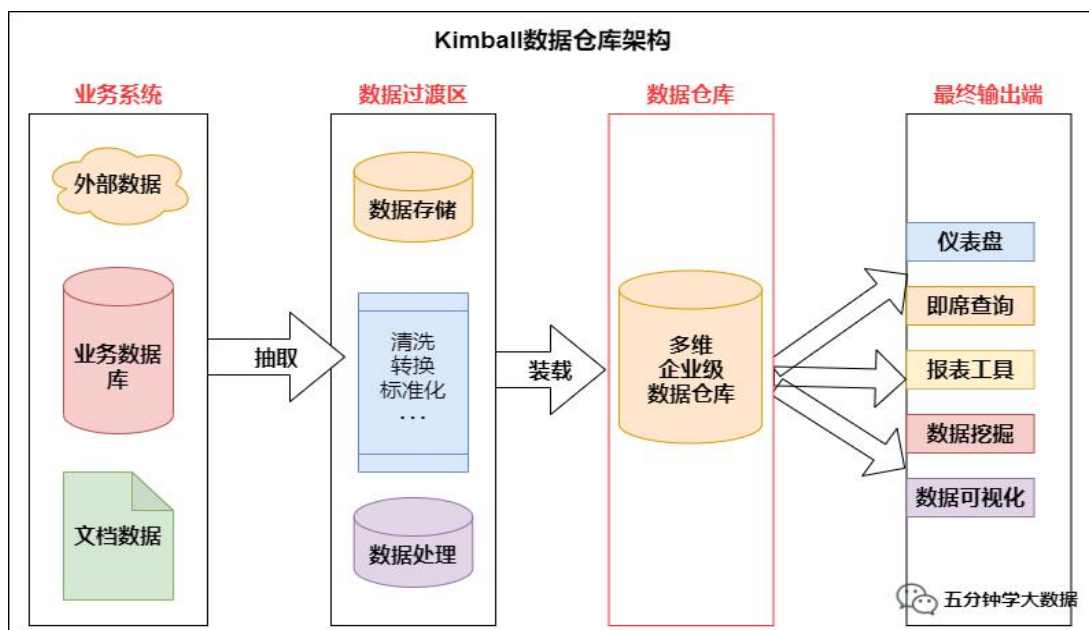


上图的前两步不过多介绍，直接从第三步开始。

**企业级数据仓库**：是该架构中的核心组件。正如 Inmon 数据仓库所定义的，企业级数据仓库是一个细节数据的集成资源库。其中的数据以**最低粒度**级别被捕获，存储在满足三范式设计的关系数据库中。

**部门级数据集市**：是面向主题数据的部门级视图，数据从企业级数据仓库获取。数据在进入部门数据集市时可能进行聚合。数据集市使用多维模型设计，用于数据分析。重要的一点是，所有的报表工具、BI 工具或其他数据分析应用都从**数据集市查询数据**，而不是直接查询企业级数据仓库。

### 3. Kimball 数据仓库架构





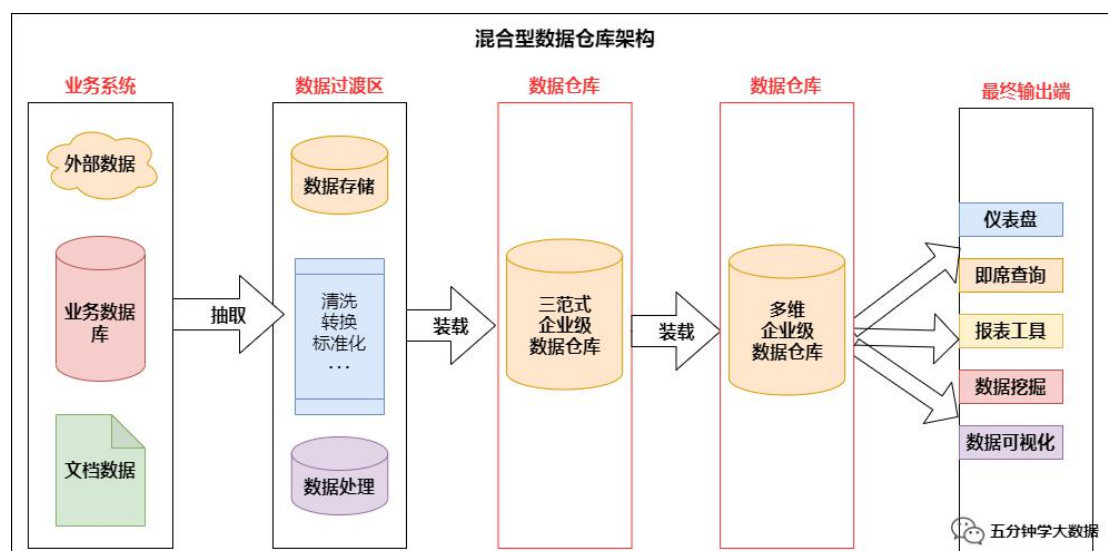
对比上一张图可以看到，Kimball 与 Inmon 两种架构的主要区别在于核心数据仓库的设计和建立。

Kimball 的数据仓库包含高粒度的企业数据，使用多维模型设计，这也意味着**数据仓库由星型模式的维度表和事实表构成**。分析系统或报表工具可以直接访问多维数据仓库里的数据。

在此架构中的数据集市也与 Inmon 中的不同。这里的**数据集市是一个逻辑概念，只是多维数据仓库中的主题域划分**，并没有自己的物理存储，也可以说是虚拟的数据集市。

#### 4. 混合型数据仓库架构

所谓的混合型结构，指的是在一个数据仓库环境中，**联合使用 Inmon 和 Kimball 两种架构**。



从架构图可以看到，这种架构将 Inmon 方法中的数据集市部分替换成了一个多维数据仓库，而数据集市则是多维数据仓库上的逻辑视图。

**使用这种架构的好处是：**既可以利用规范化设计消除数据冗余，保证数据的粒度足够细；又可以利用多维结构更灵活地在企业级实现报表和分析。

#### 7. 数据仓库元数据的管理

元数据 (Meta Date)，主要记录数据仓库中模型的定义、各层级间的映射关系、监控数据仓库的数据状态及 ETL 的任务运行状态。一般会通过元数据资料库

(Metadata Repository) 来统一地存储和管理元数据，其主要目的是使数据仓库的设计、部署、操作和管理能达成协同和一致。

元数据是数据仓库管理系统的重要组成部分，元数据管理是企业级数据仓库中的关键组件，贯穿数据仓库构建的整个过程，直接影响着数据仓库的构建、使用和维护。

- 构建数据仓库的主要步骤之一是 ETL。这时元数据将发挥重要的作用，它定义了源数据系统到数据仓库的映射、数据转换的规则、数据仓库的逻辑结构、数据更新的规则、数据导入历史记录以及装载周期等相关内容。数据抽取和转换的专家以及数据仓库管理员正是通过元数据高效地构建数据仓库。
- 用户在使用数据仓库时，通过元数据访问数据，明确数据项的含义以及定制报表。
- 数据仓库的规模及其复杂性离不开正确的元数据管理，包括增加或移除外部数据源，改变数据清洗方法，控制出错的查询以及安排备份等。

元数据可分为技术元数据和业务元数据。**技术元数据**为开发和管理数据仓库的 IT 人员使用，它描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。而**业务元数据**为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

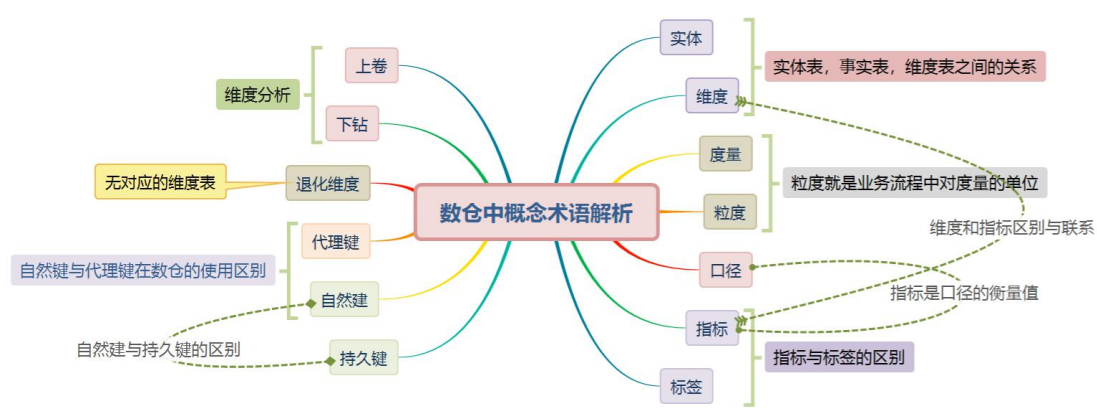
由上可见，元数据不仅定义了数据仓库中数据的模式、来源、抽取和转换规则等，而且是整个数据仓库系统运行的基础，元数据把数据仓库系统中各个松散的组件联系起来，组成了一个有机的整体。

## 8. 数仓常见术语解析

本文档首发于公众号【五分钟学大数据】

本小节结构如下图所示：





## 1. 数仓名词解释

### 1. 实体

实体是指依附的主体，就是我们分析的一个对象，比如我们分析商品的销售情况，如华为手机近半年的销售量是多少，那华为手机就是一个实体；我们分析用户的活跃度，用户就是一个实体。当然实体也可以现实中不存在的，比如虚拟的业务对象，活动，会员等都可看做一个实体。

实体的存在是为了业务分析，作为分析的一个筛选的维度，拥有描述自己的属性，本身具有可分析的价值。

### 2. 维度

维度就是看待问题的角度，分析业务数据，从什么角度分析，就建立什么样的维度。所以维度就是要对数据进行分析时所用的一个量，比如你要分析产品销售情况，你可以选择按商品类别来进行分析，这就构成一个维度，把所有商品类别集合在一起，就构成了维度表。

### 3. 度量

度量是业务流程节点上的一个数值。比如销量，价格，成本等等。

事实表中的度量可分为三类：完全可加，半可加，不可加。

1. 完全可加的度量是最灵活，最有用的，比如说销量，销售额等，可进行任意维度汇总；

2. 半可加的度量可以对某些维度汇总，但不能对所有维度汇总，差额是常见的半可加度量，它除了时间维度外，可以跨所有维度进行加法操作；
3. 还有一种是完全不可加的，例如：比率。对于这类非可加度量，一种好的方法是，**尽可能存储非可加度量的完全可加分量**，并在计算出最终的非可加事实前，将这些分量汇总到最终的结果集中。

## 4. 粒度

粒度就是业务流程中对度量的单位，比如商品是按件记录度量，还是按批记录度量。

在数仓建设中，我们说这是用户粒度的事实表，那么表中**每行**数据都是一个用户，无重复用户；例如还有销售粒度的表，那么表中每行都是一条销售记录。

**选择合适的粒度级别是数据仓库建设好坏的重要关键内容**，在设计数据粒度时，通常需重点考虑以下因素：

1. 要接受的分析类型、**可接受的数据最低粒度**和**能存储的数据量**；
2. 粒度的层次定义越高，就越不能在该仓库中进行更细致的分析；
3. 如果存储资源有一定的限制，就只能采用较高的数据粒度划分；
4. **数据粒度划分策略一定要保证：数据的粒度确实能够满足用户的决策分析需要，这是数据粒度划分策略中最重要的一個准则。**

## 5. 口径

**口径就是取数逻辑（如何取数的）**，比如**要取的数**是 10 岁以下儿童中男孩的平均身高，这就是统计的口径。

## 6. 指标

**指标是口径的衡量值，也就是最后的结果**。比如最近七天的订单量，一个促销活动的购买转化率等。

一个指标具体到计算实施，主要有以下几部分组成：

- 指标加工逻辑，比如 count , sum, avg
- 维度，比如按部门、地域进行指标统计，**对应 sql 中的 group by**

- 业务限定/修饰词，比如以不同的支付渠道来算对应的指标，微信支付的订单退款率，支付宝支付的订单退款率。对应 sql 中的 where。

除此之外，指标本身还可以衍生、派生出更多的指标，基于这些特点，可以将指标进行分类：

- **原子指标**：基本业务事实，没有业务限定、没有维度。比如订单表中的订单量、订单总金额都算原子指标；

业务方更关心的指标，是有实际业务含义，可以直接取数据的指标。比如店铺近 1 天订单支付金额就是一个派生指标，会被直接在产品上展示给商家看。但是这个指标却不能直接从数仓的统一中间层里取数（因为没有现成的事实字段，数仓提供的一般都是大宽表）。需要有一个桥梁连接数仓中间层和业务方的指标需求，于是便有了派生指标

- **派生指标**：维度+修饰词+原子指标。店铺近 1 天订单支付金额中店铺是维度，近 1 天是一个时间类型的修饰词，支付金额是一个原子指标；

维度：观察各项指标的角度；

修饰词：维度的一个或某些值，比如维度性别下，男和女就是 2 种修饰词。

- **衍生指标**：比如某一个促销活动的转化率就是衍生指标，因为需要促销投放人数指标和促销订单数指标进行计算得出。

## 7. 标签

标签是人为设定的、根据业务场景需求，对目标对象运用一定的算法得到的高度精炼的特征标识。可见标签是经过人为再加工后的结果，如网红、白富美、萝莉。对于有歧义的标签，我们内部可进行标签区分，比如：苹果，我们可以定义苹果指的是水果，苹果手机才指的是手机。

## 8. 自然键

由现实中已经存在的属性组成的键，它在业务概念中是唯一的，并具有一定的业务含义，比如商品 ID，员工 ID。

以数仓角度看，来自于业务系统的标识符就是自然键，比如业务库中员工的编号。

## 9. 持久键

保持永久性不会发生变化。有时也被叫做超自然持久键。比如身份证号属于持久键。

**自然键和持久键区别**：举个例子就明白了，比如说公司员工离职之后又重新入职，他的自然键也就是**员工编号**发生了变化，但是他的持久键身份证号是不变的。

## 10. 代理键

就是不具有业务含义的键。代理键有许多其他的称呼：无意义键、整数键、非自然键、人工键、合成键等。

代理键就是简单的以按照顺序序列生产的整数表示。产品行的第 1 行代理键为 1，则下一行的代理键为 2，如此进行。**代理键的作用仅仅是连接维度表和事实表。**

## 11. 退化维度

**退化维度**，就是那些看起来像是事实表的一个维度关键字，但实际上并没有对应的**维度表**，就是维度属性存储到事实表中，这种存储到事实表中的维度列被称为退化维度。与其他存储在维表中的维度一样，退化维度也可以用来进行事实表的过滤查询、实现聚合操作等。

那么究竟怎么定义退化维度呢？比如说订单 id，这种量级很大的维度，没必要用一张维度表来进行存储，而我们进行数据查询或者数据过滤的时候又非常需要，所以这种就冗余在事实表里面，这种就叫退化维度，citycode 这种我们也会冗余在事实表里面，但是**它有对应的维度表，所以它不是退化维度。**

## 12. 下钻

这是在数据分析中常见的概念，下钻可以理解成增加维的层次，从而可以由**粗粒度到细粒度来观察数据**，比如对产品销售情况分析时，可以沿着时间维从年到月到日更细粒度的观察数据。从年的维度可以下钻到月的维度、日的维度等。

## 13. 上卷

知道了下钻，上卷就容易理解了，它俩是相逆的操作，所以上卷可以理解为删掉维的某些层，由细粒度到粗粒度观察数据的操作或沿着维的层次向上聚合汇总数据。

## 14. 数据集市

数据集市（Data Mart），也叫数据市场，数据集市就是满足特定的部门或者用户的需求，按照多维的方式进行存储，包括定义维度、需要计算的指标、维度的层次等，生成面向决策分析需求的数据立方体。其实就是从数据仓库中抽取出来的一个小合集。

## 2. 数仓名词之间关系

### 1. 实体表，事实表，维度表之间的关系

在 Kimball 维度建模中有维度与事实，在 Inmon 范式建模中有实体与关系，如果我们分开两种建模方式看这些概念比较容易理解。但是目前也出现了不少混合建模方式，两种建模方式结合起来看，这些概念是不是容易记忆混乱，尤其事实表和实体表，它们之间到底有怎样区别与联系，先看下它们各自概念：

1. **维度表**：维度表可以看成是用户用来分析一个事实的窗口，它里面的数据应该是对事实的各个方面描述，比如时间维度表，地域维度表，维度表是事实表的一个分析角度。
2. **事实表**：事实表其实就是通过各种维度和一些指标值的组合来确定一个事实的，比如通过时间维度，地域组织维度，指标值可以去确定在某时某地的一些指标值怎么样的事实。事实表的每一条数据都是几条维度表的数据和指标值交汇而得到的。
3. **实体表**：实体表就是一个实际对象的表，实体表放的数据一定是一条条客观存在的事物数据，比如说各种商品，它就是客观存在的，所以可以将其设计一个实体表。实体表只描述各个事物，并不存在具体的事实，所以也有人称实体表是无事实的事实表。

举个例子：比如说手机商场中有苹果手机，华为手机等各品牌各型号的手机，这些数据可以组成一个**手机实体表**，但是表中没有可度量的数据。某天苹果手机卖

了 15 台，华为手机卖了 20 台，这些手机销售数据属于事实，组成一个**事实表**。这样就可以使用**日期维度表**和**地域维度表**对这个事实表进行各种维度分析。

## 2. 指标与标签的区别

- 概念不同

**指标**是用来定义、评价和描述特定事物的一种标准或方式。比如：新增用户数、累计用户数、用户活跃率等是衡量用户发展情况的指标；

**标签**是人为设定的、根据业务场景需求，对目标对象运用一定的算法得到的高度精炼的特征标识。可见标签是经过人为再加工后的结果，如网红、白富美、萝莉。

- 构成不同

**指标名称**是对事物质与量两方面特点的命名；指标取值是指标在具体时间、地域、条件下的数量表现，如人的体重，指标名称是体重，指标的取值就是 120 斤；

**标签名称**通常都是形容词或形容词+名词的结构，标签一般是不可量化的，通常是孤立的，除了基础类标签，通过一定算法加工出来的标签一般都没有单位和量纲。如将超过 200 斤的称为大胖子。

- 分类不同

**对指标的分类：**

按照指标计算逻辑，可以将指标分为原子指标、派生指标、衍生指标三种类型；按照对事件描述内容的不同，分为过程性指标和结果性指标；

**对标签的分类：**

按照标签的变化性分为静态标签和动态标签；

按照标签的指代和评估指标的不同，可分为定性标签和定量标签；

**指标**最擅长的应用是监测、分析、评价和建模。

**标签**最擅长的应用是标注、刻画、分类和特征提取。

特别需要指出的是，由于对结果的标注也是一种标签，所以在自然语言处理和机器学习相关的算法应用场景下，标签对于监督式学习有重要价值，只是单纯的指标难以做到的。而指标在任务分配、绩效管理等领域的作用，也是标签无法做到的。

## 3. 维度和指标区别与联系

维度就是数据的观察角度，即从哪个角度去分析问题，看待问题。

指标就是从维度的基础上去衡算这个结果的值。

维度一般是一个离散的值，比如时间维度上每一个独立的日期或地域，因此统计时，可以把维度相同记录的聚合在一起，应用聚合函数做累加、均值、最大值、最小值等聚合计算。

指标就是被聚合的统计计算，即聚合运算的结果，一般是一个连续的值。

#### 4. 自然键与代理键在数仓的使用区别

数仓工具箱中说**维度表的唯一主键应该是代理键而不应该是自然键**。有时建模人员不愿意放弃使用自然键，因为他们希望与操作型代码查询事实表，而不希望与维度表做连接操作。然而，应该避免使用包含业务含义的多维键，因为不管我们做出任何假设最终都可能变得无效，因为我们控制不了业务库的变动。

**所以数据仓库中维度表与事实表的每个连接应该基于无实际含义的整数代理键。避免使用自然键作为维度表的主键。**

#### 5. 数据集市和数据仓库的关系

**数据集市是企业级数据仓库的一个子集**，他主要面向部门级业务，并且只面向某个特定的主题。为了解决灵活性和性能之间的矛盾，数据集市就是数据仓库体系结构中增加的一种小型的部门或工作组级别的数据仓库。数据集市存储为特定用户预先计算好的数据，从而满足用户对性能的需求。数据集市可以在一定程度上缓解访问数据仓库的瓶颈。

**数据集市和数据仓库的主要区别：**数据仓库是企业级的，能为整个企业各个部门的运行提供决策支持手段；而数据集市则是一种微型的数据仓库，它通常有更少的数据，更少的主题区域，以及更少的历史数据，因此是部门级的，一般只能为某个局部范围内的管理人员服务，因此也称之为部门级数据仓库。

文章都会首发在公众号【五分钟学大数据】

### 二、离线数仓建设核心

**数据仓库的核心是展现层和提供优质的服务。ETL 及其规范、分层等所做的一切都是为了一个更清晰易用的展现层。**

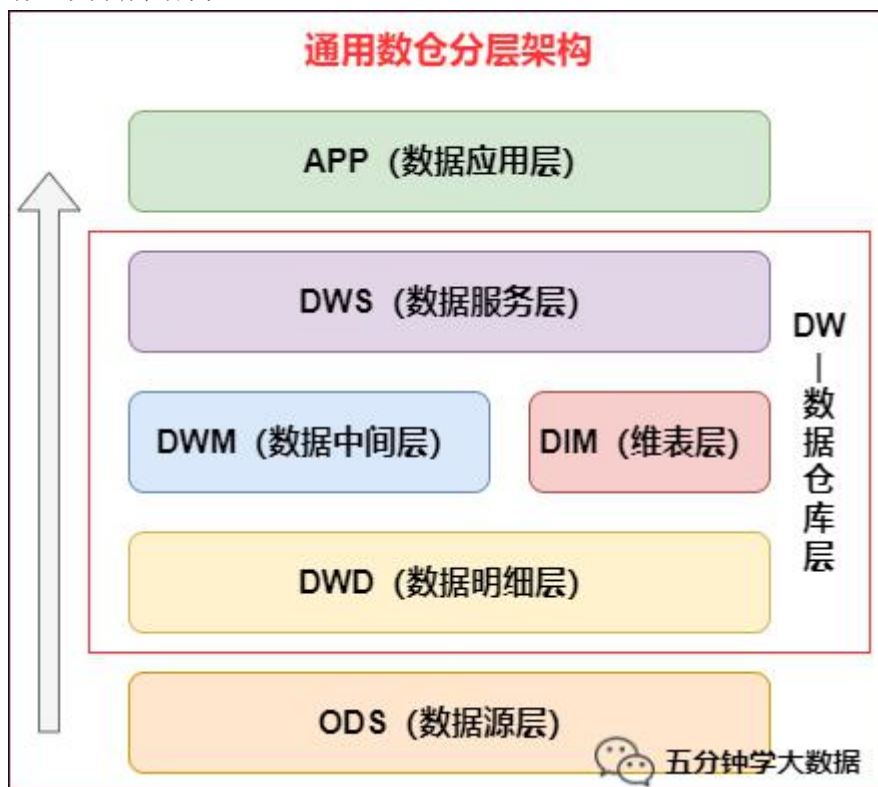


## 1. 数仓分层

### 数仓分层的原则：

1. 为便于数据分析，要屏蔽底层复杂业务，简单、完整、集成的将数据暴露给分析层。
2. 底层业务变动与上层需求变动对模型冲击最小化，业务系统变化影响削弱在基础数据层，结合自上而下的建设方法削弱需求变动对模型的影响。
3. 高内聚松耦合，即主题之内或各个完整意义的系统内数据的高内聚，主题之间或各个完整意义的系统间数据的松耦合。
4. 构建仓库基础数据层，使底层业务数据整合工作与上层应用开发工作相隔离，为仓库大规模开发奠定基础 仓库层次更加清晰，对外暴露数据更加统一。

一般采用如下分层结构：



### 1. 数据源层：ODS (Operational Data Store)

ODS 层，是最接近数据源中数据的一层，为了考虑后续可能需要追溯数据问题，因此对于这一层就不建议做过多的数据清洗工作，原封不动地接入原始数据即可，至于数据的去噪、去重、异常值处理等过程可以放在后面的 DWD 层来做。

## 2. 数据仓库层：DW (Data Warehouse)

数据仓库层是我们在做数据仓库时要核心设计的一层，在这里，从 ODS 层中获得的数据按照主题建立各种数据模型。

DW 层又细分为 **DWD** (Data Warehouse Detail) 层、**DWM** (Data Warehouse Middle) 层和 **DWS** (Data Warehouse Service) 层。

### 1) 数据明细层：DWD (Data Warehouse Detail)

该层一般保持和 ODS 层一样的数据粒度，并且提供一定的数据质量保证。**DWD 层要做的就是将数据清理、整合、规范化、脏数据、垃圾数据、规范不一致的、状态定义不一致的、命名不规范的数据都会被处理。**

同时，为了提高数据明细层的易用性，**该层会采用一些维度退化手法，将维度退化至事实表中，减少事实表和维表的关联。**

另外，在该层也会做一部分的数据聚合，将相同主题的数据汇集到一张表中，提高数据的可用性。

### 2) 数据中间层：DWM (Data Warehouse Middle)

该层会在 DWD 层的数据基础上，数据做轻度的聚合操作，生成一系列的中间表，提升公共指标的复用性，减少重复加工。

直观来讲，**就是对通用的核心维度进行聚合操作，算出相应的统计指标。**

在实际计算中，如果直接从 DWD 或者 ODS 计算出宽表的统计指标，会存在计算量太大并且维度太少的问题，因此一般的做法是，在 DWM 层先计算出多个小的中间表，然后再拼接成一张 DWS 的宽表。由于宽和窄的界限不易界定，也可以去掉 DWM 这一层，只留 DWS 层，将所有数据再放在 DWS 亦可。

### 3) 数据服务层：DWS (Data Warehouse Service)

DWS 层为公共汇总层，会进行轻度汇总，粒度比明细数据稍粗，基于 DWD 层上的基础数据，**整合汇总成分析某一个主题域的服务数据，一般是宽表。**DWS 层应覆盖 80% 的应用场景。又称数据集市或宽表。

按照业务划分，如主题域流量、订单、用户等，生成字段比较多的宽表，用于提供后续的业务查询，OLAP 分析，数据分发等。

一般来讲，该层的数据表会相对比较少，一张表会涵盖比较多的业务内容，由于其字段较多，因此一般也会称该层的表为宽表。

### 3. 数据应用层：APP (Application)

在这里，主要是提供给数据产品和数据分析使用的数据，一般会存放在 ES、PostgreSQL、Redis 等系统中供线上系统使用，也可能存在 Hive 或者 Druid 中供数据分析和数据挖掘使用。比如我们经常说的报表数据，一般就放在这里。

### 4. 维表层：DIM (Dimension)

如果维表过多，也可针对维表设计单独一层，维表层主要包含两部分数据：

**高基数维度数据**：一般是用户资料表、商品资料表类似的资料表。数据量可能是千万级或者上亿级别。

**低基数维度数据**：一般是配置表，比如枚举值对应的中文含义，或者日期维表。数据量可能是个位数或者几千几万。

## 2. 数仓建模方法

数仓建模在哪层建设呢？我们以维度建模为例，**建模是在数据源层的下一层进行建设**，在上节的分层架构中，**就是在 DW 层进行数仓建模**，所以 **DW 层是数仓建设的核心层**。

那数仓建模怎么建呢？其实数据仓库的建模方法有很多种，**每一种建模方法代表了哲学上的一个观点**，代表了一种归纳、概括世界的一种方法。常见的有 **范式建模法、维度建模法、实体建模法**等，**每种方法从本质上将从不同的角度看待业务中的问题**。

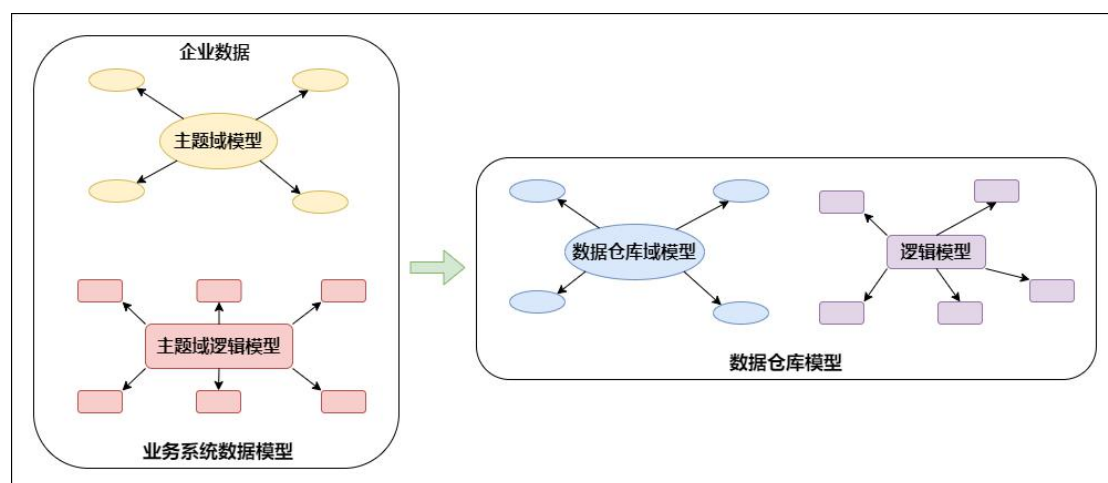
### 1. 范式建模法 (Third Normal Form, 3NF)

范式建模法其实是在构建数据模型常用的一个方法，该方法的主要由 Inmon 所提倡，主要解决关系型数据库的数据存储，利用的一种技术层面上的方法。目前，我们在关系型数据库中的建模方法，大部分采用的是三范式建模法。

范式 是符合某一种级别的关系模式的集合。构造数据库必须遵循一定的规则，而在关系型数据库中这种规则就是范式，这一过程也被称为规范化。目前关系数据库有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、Boyce-Codd 范式（BCNF）、第四范式（4NF）和第五范式（5NF）。

在数据仓库的模型设计中，一般采用第三范式。一个符合第三范式的关系必须具有以下三个条件：

- 每个属性值唯一，不具有多义性；
- 每个非主属性必须完全依赖于整个主键，而非主键的一部分；
- 每个非主属性不能依赖于其他关系中的属性，因为这样的话，这种属性应该归到其他关系中去。

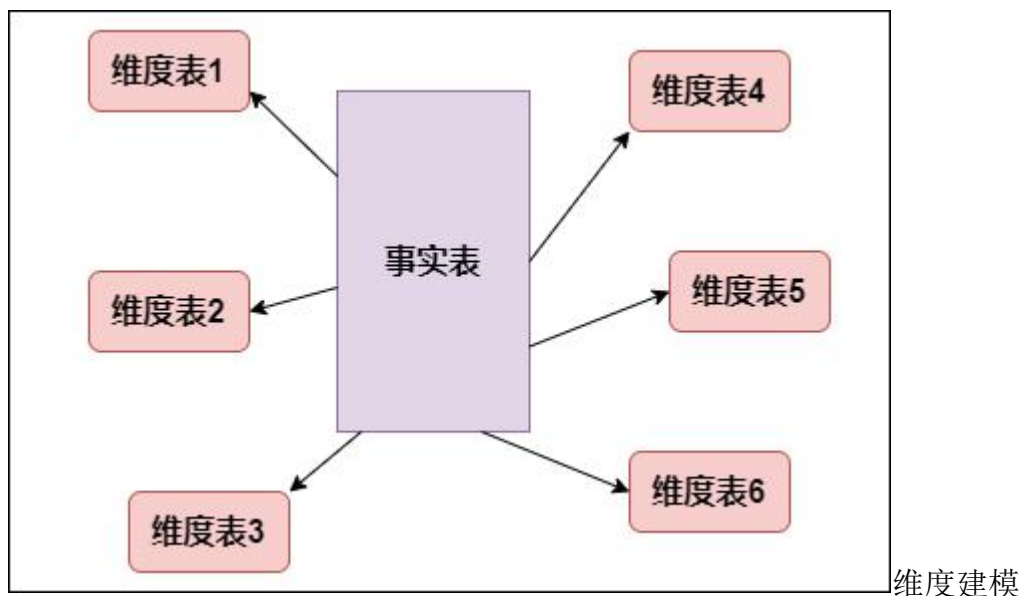


## 范式建模

根据 Inmon 的观点，数据仓库模型的建设方法和业务系统的企业数据模型类似。在业务系统中，企业数据模型决定了数据的来源，而企业数据模型也分为两个层次，即主题域模型和逻辑模型。同样，主题域模型可以看成是业务模型的概念模型，而逻辑模型则是域模型在关系型数据库上的实例化。

## 2. 维度建模法 (Dimensional Modeling)

维度模型是数据仓库领域另一位大师 Ralph Kimall 所倡导，他的《数据仓库工具箱》是数据仓库工程领域最流行的数仓建模经典。维度建模以分析决策的需求出发构建模型，构建的数据模型为分析需求服务，因此它重点解决用户如何更快速完成分析需求，同时还有较好的大规模复杂查询的响应性能。



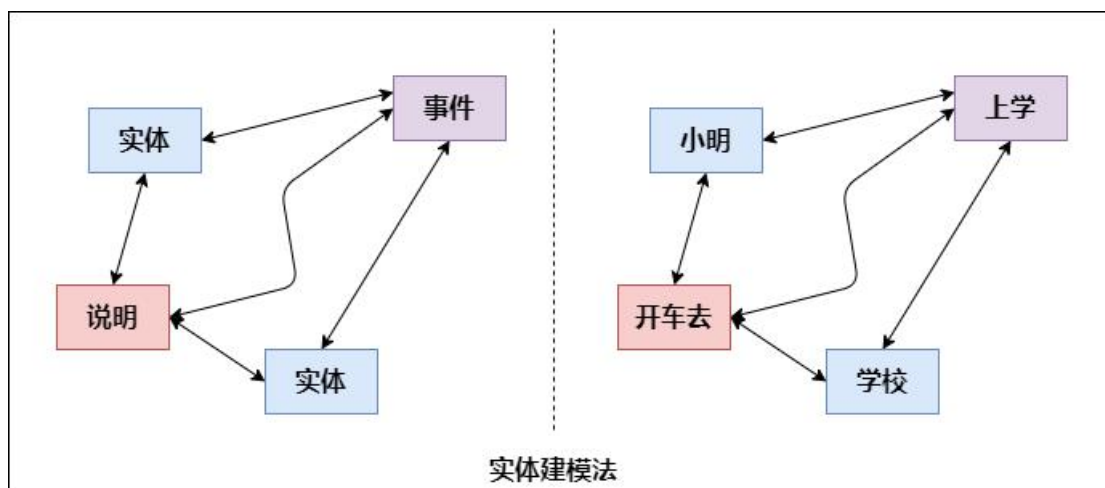
典型的代表是我们比较熟知的星形模型（Star-schema），以及在一些特殊场景下适用的雪花模型（Snow-schema）。

维度建模中比较重要的概念就是 事实表（Fact table）和维度表（Dimension table）。其最简单的描述就是，按照事实表、维度表来构建数据仓库、数据集市。

### 3. 实体建模法（Entity Modeling）

实体建模法并不是数据仓库建模中常见的一个方法，它来源于哲学的一个流派。从哲学的意义上说，客观世界应该是可以细分的，客观世界应该可以分成由一个个实体，以及实体与实体之间的关系组成。那么我们在数据仓库的建模过程中完全可以引入这个抽象的方法，将整个业务也可以划分成一个个的实体，而每个实体之间的关系，以及针对这些关系的说明就是我们数据建模需要做的工作。

虽然实体法粗看起来好像有一些抽象，其实理解起来很容易。即我们可以将任何一个业务过程划分成 3 个部分，**实体**，**事件**，**说明**，如下图所示：



### 实体建模

上图表述的是一个抽象的含义，如果我们描述一个简单的事实：“小明开车去学校上学”。以这个业务事实为例，我们可以把“小明”，“学校”看成是一个实体，“上学”描述的是一个业务过程，我们在这里可以抽象为一个具体“事件”，而“开车去”则可以看出是事件“上学”的一个说明。

## 3. 维度建模详解

目前在互联网公司最常用的建模方法就是维度建模，我们将重点讲解！

维度建模是专门应用于分析型数据库、数据仓库、数据集市建模的方法。数据集市可以理解为是一种“小型数据仓库”。

我们先不着急开始维度建模，先来了解下[维度建模中表的类型](#)和[维度建模的模式](#)之后再开始建模，这样能够让我们深刻理解！

### 1. 维度建模中表的类型

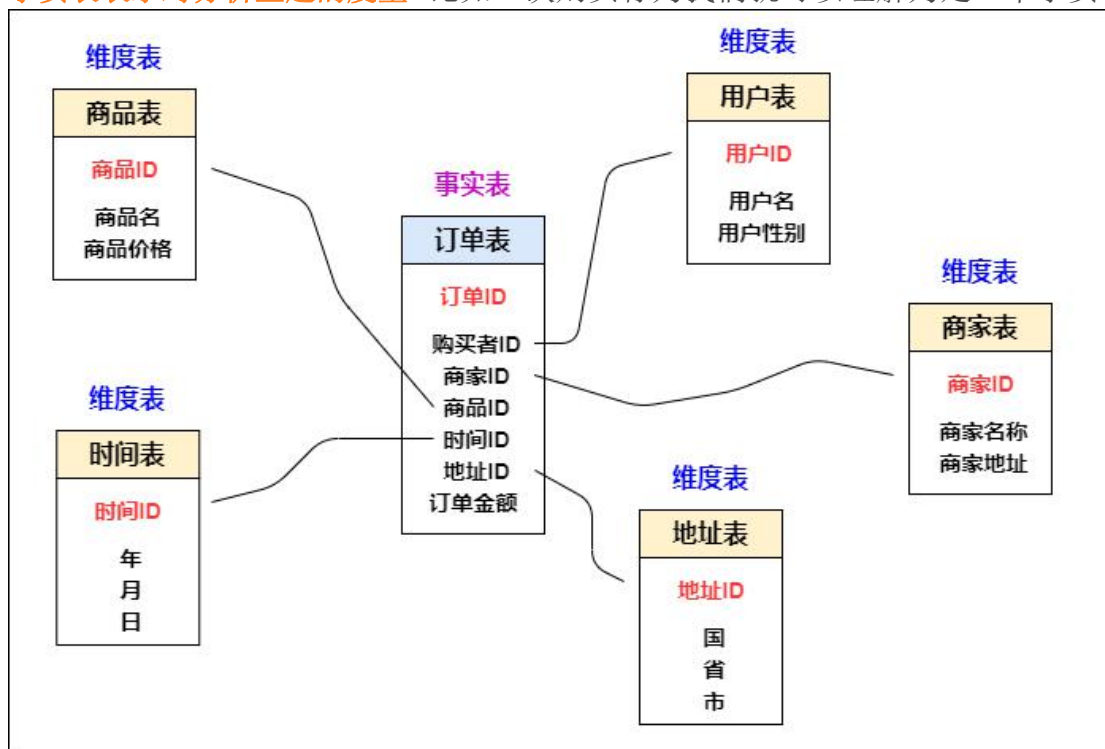
维度建模分为两种表：事实表和维度表：

1. **事实表**：必然存在的一些数据，像采集的日志文件，订单表，都可以作为事实表。  
特征：是一堆主键的集合，每个主键对应维度表中的一条记录，客观存在的，根据主题确定出需要使用的数据
2. **维度表**：维度就是所分析的数据的一个量，维度表就是以合适的角度来创建的表，分析问题的一个角度：时间、地域、终端、用户等角度

### 1. 事实表

发生在现实世界中的操作型事件，其所产生的可度量数值，存储在事实表中。从最低的粒度级别来看，事实表行对应一个度量事件，反之亦然。

**事实表表示对分析主题的度量。**比如一次购买行为我们就可以理解为是一个事实。



## 事实与维度

图中的订单表就是一个事实表，你可以理解他就是在现实中发生的一次操作型事件，我们每完成一个订单，就会在订单中增加一条记录。事实表的特征：表里没有存放实际的内容，他是一堆主键的集合，这些 ID 分别能对应到维度表中的一条记录。事实表包含了与各维度表相关联的外键，可与维度表关联。事实表的度量通常是数值类型，且记录数会不断增加，表数据规模迅速增长。

## 明细表（宽表）：

事实表的数据中，有些属性共同组成了一个字段（糅合在一起），比如年月日时分秒构成了时间，当需要根据某一属性进行分组统计的时候，需要截取拼接之类的操作，效率极低。如：

local_time
2021-03-18 06:31:42

**为了分析方便，可以事实表中的一个字段切割提取多个属性出来构成新的字段，因为字段变多了，所以称为宽表，原来的成为窄表。**

将上述的 local\_time 字段扩展为如下 6 个字段：

year	month	day	hour	m	s
------	-------	-----	------	---	---



year	month	day	hour	m	s
2021	03	18	06	31	42

又因为宽表的信息更加清晰明细，所以也可以称之为明细表。

## 事实表种类

事实表分为以下 6 类：

1. 事务事实表
2. 周期快照事实表
3. 累积快照事实表
4. 无事实的事实表
5. 聚集事实表
6. 合并事实表

简单解释下每种表的概念：

- 事务事实表

表中的一行对应空间或时间上某点的度量事件。就是一行数据中必须有度量字段，什么是度量，就是指标，比如说销售金额，销售数量等这些可加的或者半可加就是度量值。另一点就是事务事实表都包含一个与维度表关联的外键。并且度量值必须和事务粒度保持一致。

- 周期快照事实表

顾名思义，周期事实表就是每行都带有时间值字段，代表周期，通常时间值都是标准周期，如某一天，某周，某月等。粒度是周期，而不是个体的事务，也就是说一个周期快照事实表中数据可以是多个事实，但是它们都属于某个周期内。

- 累计快照事实表

周期快照事实表是单个周期内数据，而累计快照事实表是由多个周期数据组成，每行汇总了过程开始到结束之间的度量。每行数据相当于管道或工作流，有事件的起点，过程，终点，并且每个关键步骤都包含日期字段。如订单数据，累计快照事实表的一行就是一个订单，当订单产生时插入一行，当订单发生变化时，这行就被修改。

- 无事实的事实表

我们以上讨论的事实表度量都是数字化的，当然实际应用中绝大多数都是数字化的度量，但是也可能会有少量的没有数字化的值但是还很有价值的字段，无事实的事实表就是为这种数据准备的，利用这种事实表可以分析发生了什么。

- 聚集事实表

聚集，就是对原子粒度的数据进行简单的聚合操作，目的就是为了提高查询性能。如我们需求是查询全国所有门店的总销售额，我们原子粒度的事实表中每行是每个分店每个商品的销售额，聚集事实表就可以先聚合每个分店的总销售额，这样汇总所有门店的销售额时计算的数据量就会小很多。

- 合并事实表

这种事实表遵循一个原则，就是相同粒度，数据可以来自多个过程，但是只要它们属于相同粒度，就可以合并为一个事实表，这类事实表特别适合经常需要共同分析的多过程度量。

## 2. 维度表

每个维度表都包含单一的主键列。维度表的主键可以作为与之关联的任何事实表的外键，当然，维度表行的描述环境应与事实表行完全对应。维度表通常比较宽，是扁平型非规范表，包含大量的低粒度的文本属性。

维度表示你要对数据进行分析时所用的一个量，比如你要分析产品销售情况，你可以选择按类别来进行分析，或按区域来分析。每个类别就构成一个维度。上图中的用户表、商家表、时间表这些都属于维度表，这些表都有一个唯一的主键，然后在表中存放了详细的数据信息。

总的说来，在数据仓库中不需要严格遵守规范化设计原则。因为数据仓库的主导功能就是面向分析，以查询为主，不涉及数据更新操作。**事实表的设计是以能够正确记录历史信息为准则，维度表的设计是以能够以合适的角度来聚合主题内容为准则。**

- 维度表结构

维度表谨记一条原则，包含单一主键列，但有时因业务复杂，也可能出现联合主键，请尽量避免，如果无法避免，也要确保必须是单一的，这很重要，如果维表主键不是单一，和事实表关联时会出现数据发散，导致最后结果可能出现错误。维度表通常比较宽，包含大量的低粒度的文本属性。

- **跨表钻取**

跨表钻取意思是当每个查询的行头都包含相同的一致性属性时，使不同的查询能够针对两个或更多的事实表进行查询

钻取可以改变维的层次，变换分析的粒度。它包括上钻/下钻：

上钻（roll-up）：上卷是沿着维的层次向上聚集汇总数据。例如，对产品销售数据，沿着时间维上卷，可以求出所有产品在所有地区每月（或季度或年或全部）的销售额。

下钻（drill-down）：下钻是上钻的逆操作，它是沿着维的层次向下，查看更详细的数据。

- **退化维度**

退化维度就是将维度退回到事实表中。因为有时维度除了主键没有其他内容，虽然也是合法维度键，但是一般都会退回到事实表中，减少关联次数，提高查询性能

- **多层次维度**

多数维度包含不止一个自然层次，如日期维度可以从天的层次到周到月到年的层次。所以在有些情况下，在同一维度中存在不同的层次。

- **维度表空值属性**

当给定维度行没有被全部填充时，或者当存在属性没有被应用到所有维度行时，将产生空值维度属性。上述两种情况，推荐采用描述性字符串代替空值，如使用 unknown 或 not applicable 替换空值。

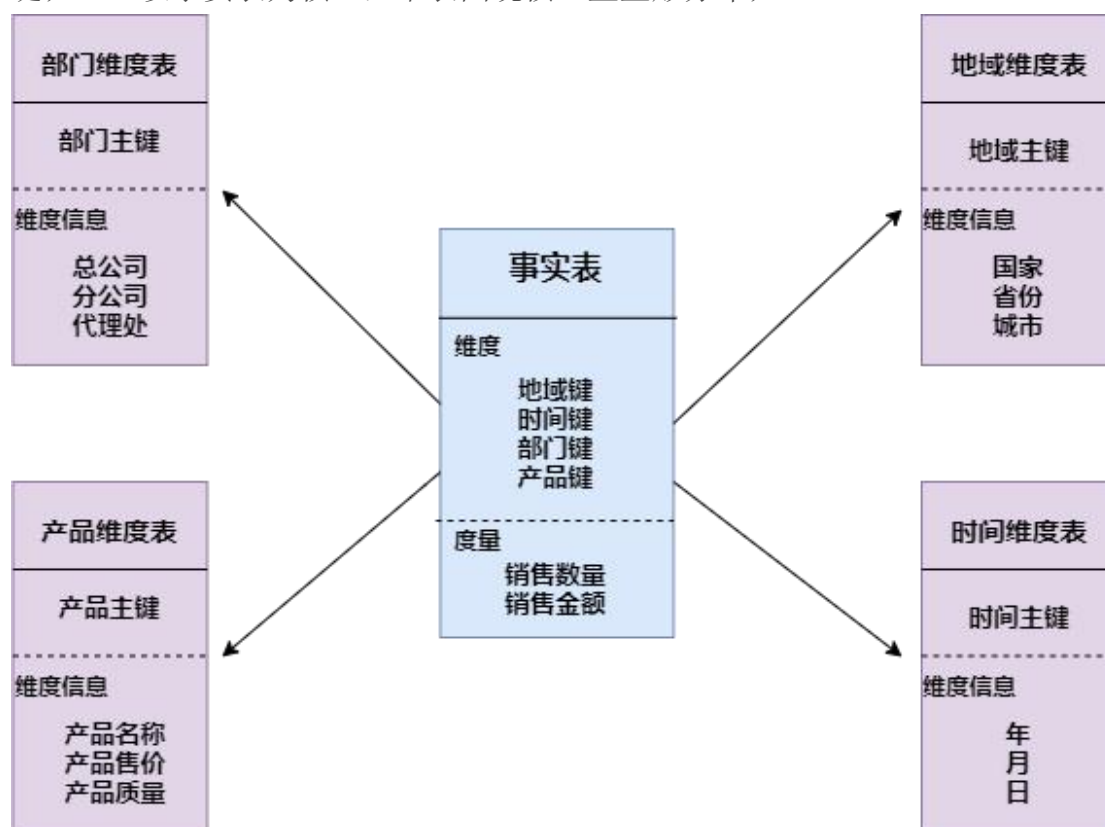
- **日历日期维度**

在日期维度表中，主键的设置不要使用顺序生成的 id 来表示，可以使用更有意义的数字表示，比如将年月日合并起来表示，即 YYYYMMDD，或者更加详细的精度。

## 2. 维度建模三种模式

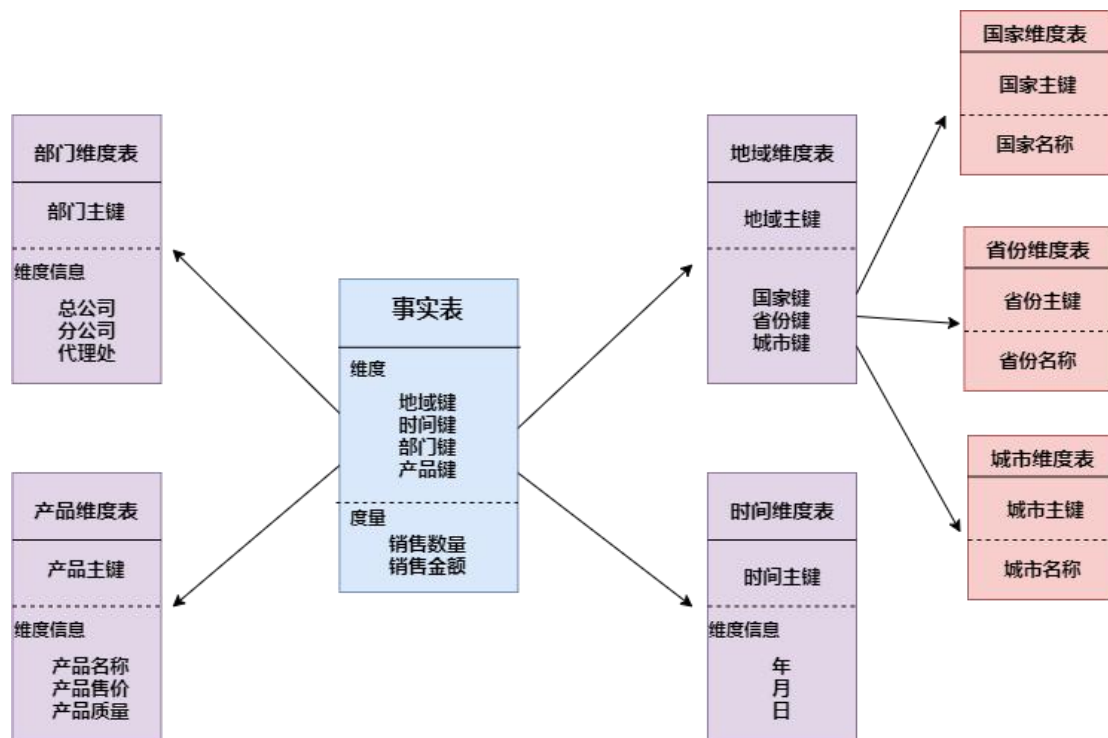
### 1. 星型模式

星形模式 (Star Schema) 是最常用的维度建模方式。**星型模式是以事实表为中心，所有的维度表直接连接在事实表上，像星星一样。** 星形模式的维度建模由一个事实表 and 一组维表成，且具有以下特点： a. 维表只和事实表关联，维表之间没有关联； b. 每个维表主键为单列，且该主键放置在事实表中，作为两边连接的外键； c. 以事实表为核心，维表围绕核心呈星形分布；



### 2. 雪花模式

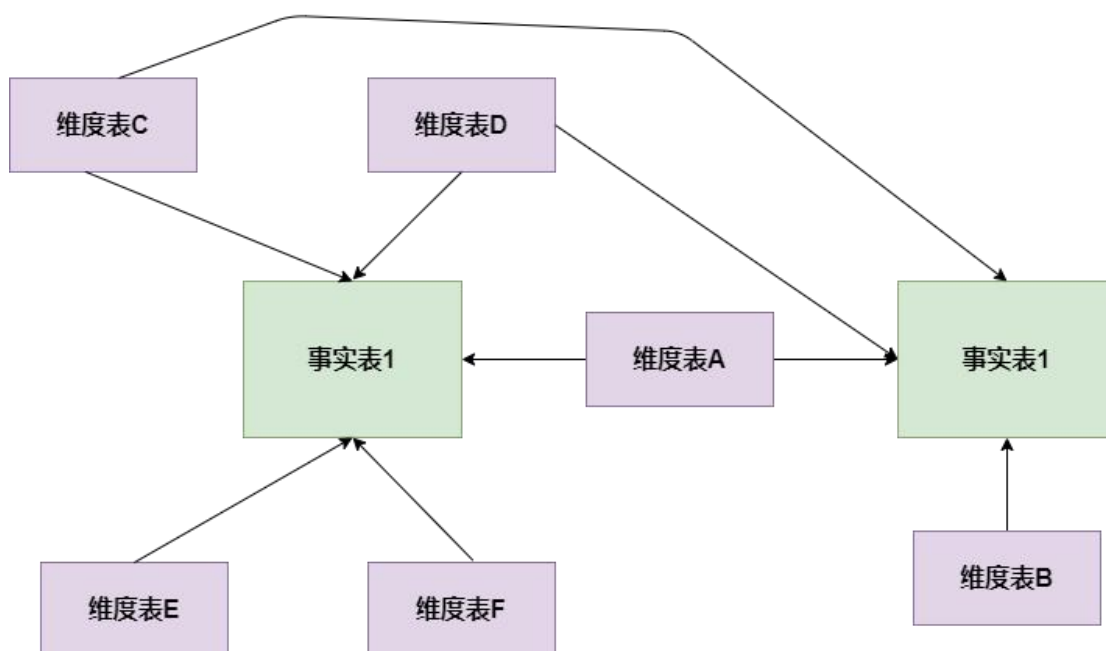
雪花模式(Snowflake Schema)是对星形模式的扩展。**雪花模式的维度表可以拥有其他维度表的**，虽然这种模型相比星型更规范一些，但是由于这种模型不太容易理解，维护成本比较高，而且性能方面需要关联多层维表，性能也比星型模型要低。所以一般不是很常用



雪花模式

### 3. 星座模式

星座模式是星型模式延伸而来，星型模式是基于一张事实表的，而**星座模式是基于多张事实表的，而且共享维度信息**。前面介绍的两种维度建模方法都是多维表对应单事实表，但在很多时候维度空间内的事实表不止一个，而一个维表也可能被多个事实表用到。在**业务发展后期**，绝大部分维度建模都采用的是星座模式。

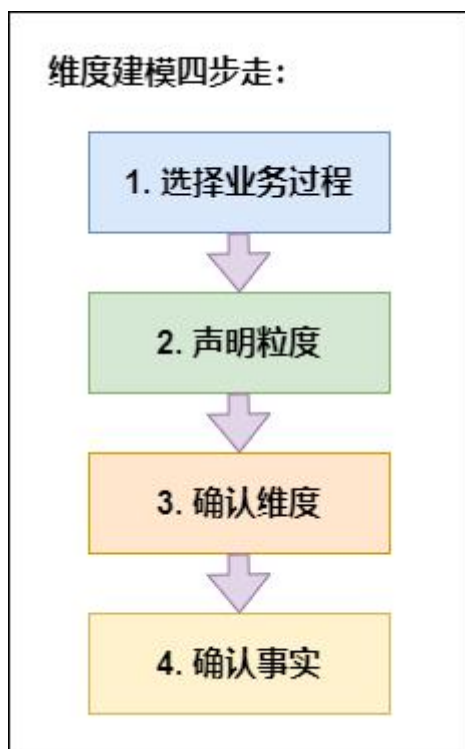


星座模型

### 3. 维度建模过程

我们知道维度建模的表类型有事实表，维度表；模式有星形模型，雪花模型，星座模型这些概念了，但是实际业务中，给了我们一堆数据，我们怎么拿这些数据进行数仓建设呢，数仓工具箱作者根据自身 60 多年的实际业务经验，给我们总结了如下四步，请务必记住！

**数仓工具箱中的维度建模四步走：**



维度建模四步走

请**牢记**以上四步，不管什么业务，就按照这个步骤来，顺序不要搞乱，因为这四步是环环相扣，步步相连。下面详细拆解下每个步骤怎么做

### 1、选择业务过程

维度建模是紧贴业务的，所以必须以业务为根基进行建模，那么选择业务过程，顾名思义就是在整个业务流程中选取我们需要建模的业务，根据运营提供的需求及日后的易扩展性等进行选择业务。比如商城，整个商城流程分为商家端，用户端，平台端，运营需求是总订单量，订单人数，及用户的购买情况等，我们选择业务过程就选择用户端的数据，商家及平台端暂不考虑。业务选择非常重要，因为后面所有的步骤都是基于此业务数据展开的。

### 2、声明粒度

先举个例子：对于用户来说，一个用户有一个身份证号，一个户籍地址，多个手机号，多张银行卡，那么与用户粒度相同的粒度属性有身份证粒度，户籍地址粒度，比用户粒度更细的粒度有手机号粒度，银行卡粒度，存在一对一的关系就是相同粒度。为什么要提相同粒度呢，因为维度建模中要求我们，在**同一事实表**中，必须具有**相同的粒度**，同一事实表中不要混用多种不同的粒度，不同的粒度数据建立不同的事实表。并且从给定的业务过程获取数据时，强烈建议从关注原子粒度开始设计，也就是从最细粒度开始，因为原子粒度能够承受无法预期的用户查询。但是上卷汇总粒度对查询性能的提升很重要的，所以对于有明确需求的数据，我们建立针对需求的上卷汇总粒度，对需求不明朗的数据我们建立原子粒度。



### 3、确认维度

维度表是作为业务分析的入口和描述性标识，所以也被称为数据仓库的“灵魂”。在一堆的数据中怎么确认哪些是维度属性呢，如果该列是对具体值的描述，是一个文本或常量，某一约束和行标识的参与者，此时该属性往往是维度属性，数仓工具箱中告诉我们**牢牢掌握事实表的粒度，就能将所有可能存在的维度区分开**，并且要**确保维度表中不能出现重复数据，应使维度主键唯一**

### 4、确认事实

事实表是用来度量的，基本上都以数量值表示，事实表中的每行对应一个度量，每行中的数据是一个特定级别的细节数据，称为粒度。维度建模的核心原则之一是**同一事实表中的所有度量必须具有相同的粒度**。这样能确保不会出现重复计算度量的问题。有时候往往不能确定该列数据是事实属性还是维度属性。记住**最实用的事实就是数值类型和可加类事实**。所以可以通过分析该列是否是一种包含多个值并作为计算的参与者的度量，这种情况下该列往往是事实。

## 三、离线数仓建设实战

技术是为业务服务的，业务是为公司创造价值的，离开业务的技术是无意义的

### 1. 业务介绍

需要针对不同需求的用户开发不同的产品，所以公司内部有很多条业务线，但是对于数据部门来说，所有业务线的数据都是数据源。对数据的划分不只是根据业务进行，而是结合数据的属性。

### 2. 早期规划

之前开发是不同业务线对应不同的数据团队，每个数据团队互不干扰，这种模式比较简单，只针对自己的业务线进行数仓建设及报表开发即可。

但是随着业务的发展，频繁迭代及跨部门的垂直业务单元越来越多，业务之间的出现耦合情况，这时再采用这种烟囱式开发就出现了问题：

例如权限问题，公司对数据管理比较严格，不同的数据开发组没有权限共享数据，需要其他业务线的数据权限需要上报审批，比较耽误时间；

还有重复开发问题，不同业务线会出现相同的报表需求，如果每个业务方都开发各自的报表，太浪费资源。

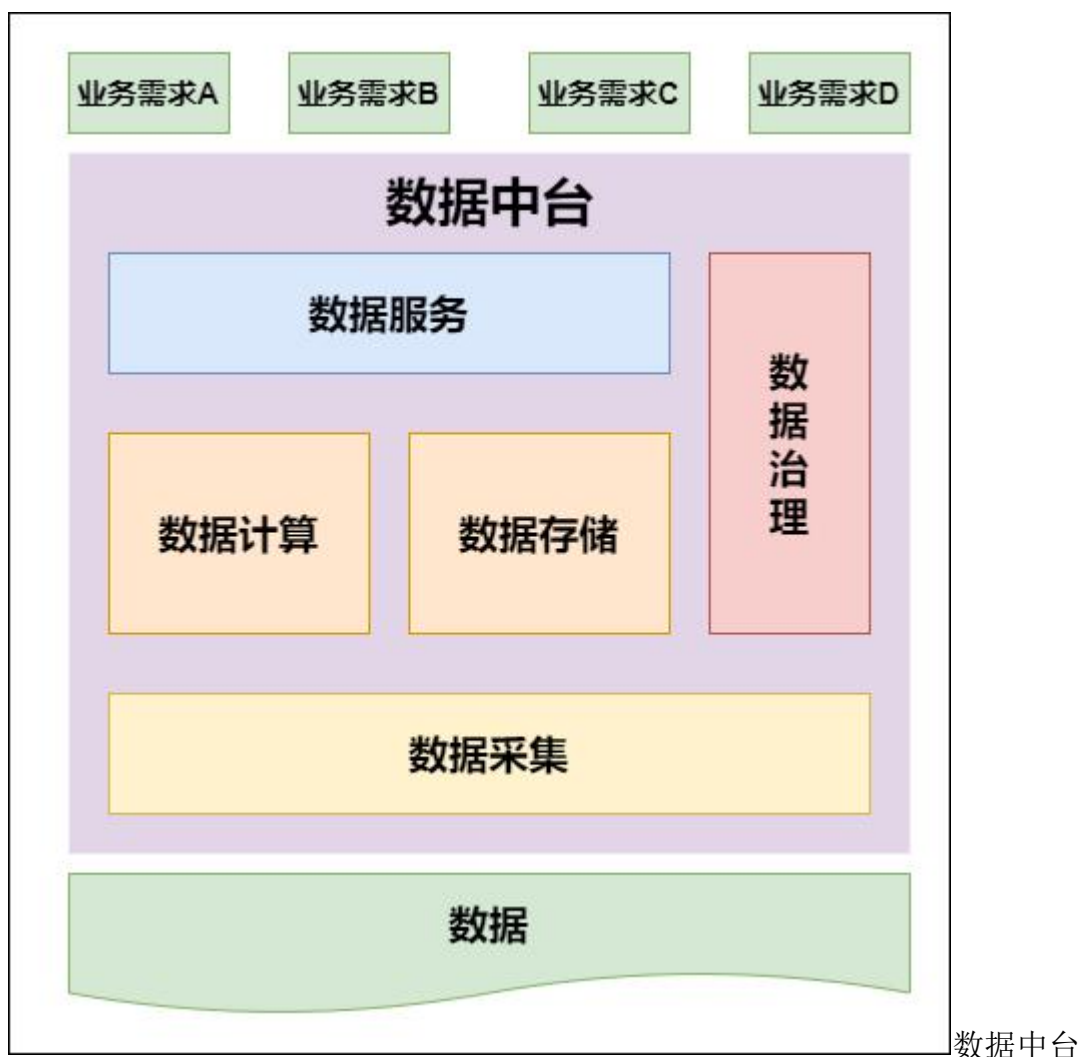
所以对于数据开发而言，需要对各个业务线的数据进行统一管理，所以就有了数据中台的出现。

### 3. 数据中台

我认为数据中台是根据每个公司具体的业务需求而搭建的，不同的业务，对中台的理解有所不同。



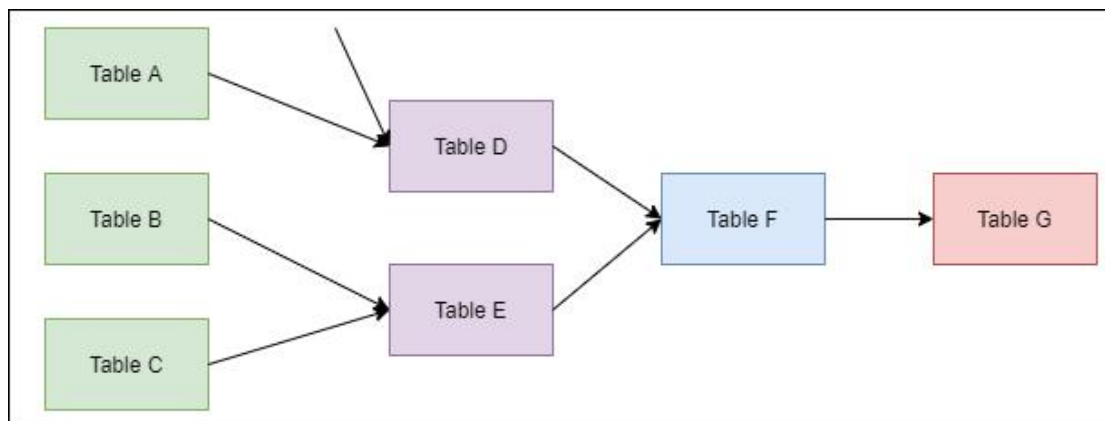
公司内部开发的敏捷数据中台，主要从数据技术和计算能力的复用，到数据资产和数据服务的复用，数据中台以更大价值带宽，快准精让数据直接赋能业务。提供一个统一化的管理，打破数据孤岛，追溯数据血缘，实现自助化及高复用度。如下所示：



以上解释比较抽象，我们以实际项目开发来看下数据中台的便利性。

比如我们之前做报表开发流程，首先是要数据采集，不同的数据源通过 sqoop 等工具采集到大数据平台，然后进行数仓搭建，最后产出报表数据，放到可视化系统展示，最终把整个流程写成脚本放到调度平台进行自动化执行。

而有了数据中台之后就不需要那么繁琐，直接进行数仓搭建，产生报表即可，无需将精力过多放在数据源、可视化展示及调度。并且可以直观的查看数据血缘关系，计算表之间血缘。像下面图中，表之间的依赖关系很明确：



## 数据中台

另一点，数据中台的异构数据系统可以非常简单的进行关联查询，比如 hive 的表关联 mysql 的表。可透明屏蔽异构数据系统异构交互方式，轻松实现跨异构数据系统透明混算。

异构数据系统原理是数据中台提供虚拟表到物理表之间的映射, 终端用户无需关心数据的物理存放位置和底层数据源的特性，可直接操作数据，体验类似操作一个虚拟数据库。



数据中台额外集成可视化展示，提供一站式数据可视化解决方案，支持 JDBC 数据源和 CSV 文件上传，支持基于数据模型拖拽智能生成可视化组件，大屏展示自适应不同大小屏幕。

调度系统是公司内部自写集成到数据中台的，在编写完 sql 语句之后可以直接进行调度。

## 4. 数仓建设

到这才真正到数仓建设，为什么前面我要占那么大篇幅去介绍公司业务及所使用的数据中台系统，因为下面的数仓建设是根据公司的业务发展及现有的数据中台进行，数仓的建设离不开公司的业务。



### 智能数仓规划

**数仓建设核心思想：从设计、开发、部署和使用层面，避免重复建设和指标冗余建设，从而保障数据口径的规范和统一，最终实现数据资产全链路关联、提供标准数据输出以及建立统一的数据公共层。** 有了核心思想，那怎么开始数仓建设，有句话说数仓建设者即是技术专家，也是大半个业务专家，所以采用的方式就是需求推动数据建设，并且因为数据中台，所以各业务知识体系比较集中，各业务数据不再分散，加快了数仓建设速度。

数仓建设主要从两个方面进行，**模型和规范**，所有业务进行统一化

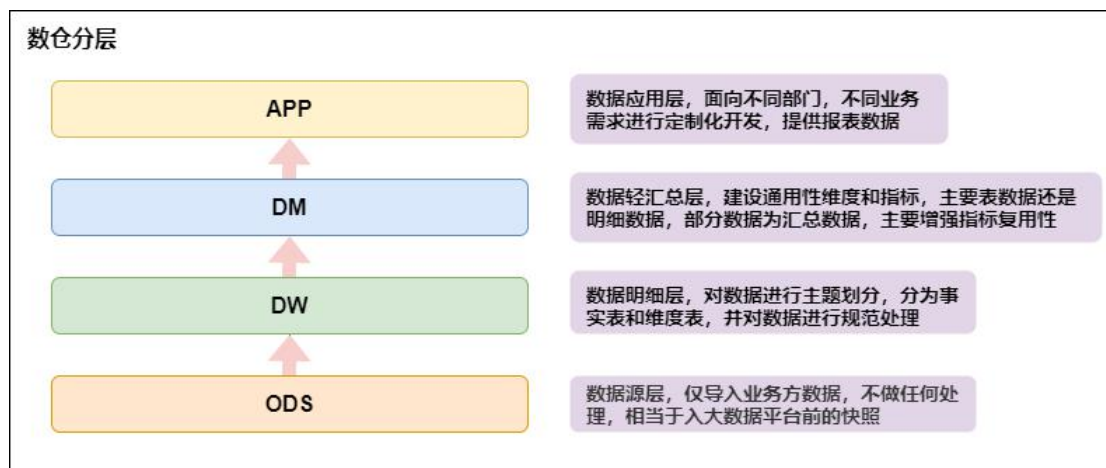
- **模型**

所有业务采用统一的模型体系，从而降低研发成本，增强指标复用，并且能保证数据口径的统一



## • 模型分层

结合公司业务，后期新增需求较多，所以分层不宜过多，并且需要清晰明确各层职责，要保证数据层的稳定又要屏蔽对下游影响，所以采用如下分层结构：



数据分层架构

## • 数据流向

遵循模型开发时分层结构，数据从 ods -> dw -> dm -> app 这样正向流动，可以防止因数据引用不规范而造成数据链路混乱及 SLA 时效难保障等问题，同时保证血缘关系简洁化，能够轻易追踪数据流向。在开发时应避免以下情况出现：

1. 数据引用链路不正确，如 ods -> dm -> app，出现这种情况说明明细层没有完全覆盖数据；如 ods -> dw -> app，说明轻度汇总层主题划分未覆盖全。减少跨层引用，才能提高中间表的复用度。理想的数仓模型设计应当具备：**数据模型可复用，完善且规范**。
2. 尽量避免一层的表生成当前层的表，如 dw 层表生成 dw 层表，这样会影响 ETL 效率。
3. 禁止出现反向依赖，如 dw 表依赖于 dm 表。

## • 规范

### • 表命名规范

1. 对于 ods、dm、app 层表名：类型\_主题\_表含义，如：dm\_xxsh\_user
2. 对于 dw 层表名：类型\_主题\_维度\_表含义，如：dw\_xxsh\_fact\_users（事实表）、dw\_xxsh\_dim\_city（维度表）

### • 字段命名规范

构建词根，词根是维度和指标管理的基础，划分为普通词根与专有词根

1. 普通词根：描述事物的最小单元体，如：sex-性别。



2. 专有词根：具备行业专属或公司内部规定的描述体，如：xxsh-公司内部对某个产品的称呼。

- 脚本命名规范

脚本名称：脚本类型.脚本功用.[库名].脚本名称，如

hive.hive.dm.dm\_xxsh\_users

脚本类型主要分为以下三类：

1. 常规 Hive sql: hive
2. 自定义 shell 脚本: sh
3. 自定义 Python 脚本: python

- 脚本内容规范

```
#变量的定义要符合python的语法要求
```

```
#指定任务负责人
```

```
owner = "zhangsan@xxx.com"
```

```
#脚本存放目录/opt/xxx
```

```
#脚本名称 hive.hive.dm.dm_xxsh_users
```

```
#source 用来标识上游依赖表，一个任务如果有多个上游表，都需要写进去
```

```
##(xxx_name 是需要改动的，其余不需要改)
```

```
source = {
```

```
    "table_name": {
```

```
        "db": "db_name",
```

```
        "table": "table_name"
```

```
    }
```

```
}
```

```
#如source，但是每个任务target只有一张表
```

```
target = {
```

```
    "db_table": {
```

```
        "host": "hive",
```

```
        "db": "db_name",
```

```
        "table": "table_name"
```

```
    }
```

```
}
```

```
#变量列表
```

```
#$now
```

```
#$now.date 常用，格式示例：2020-12-11
```

```
task = '''
```

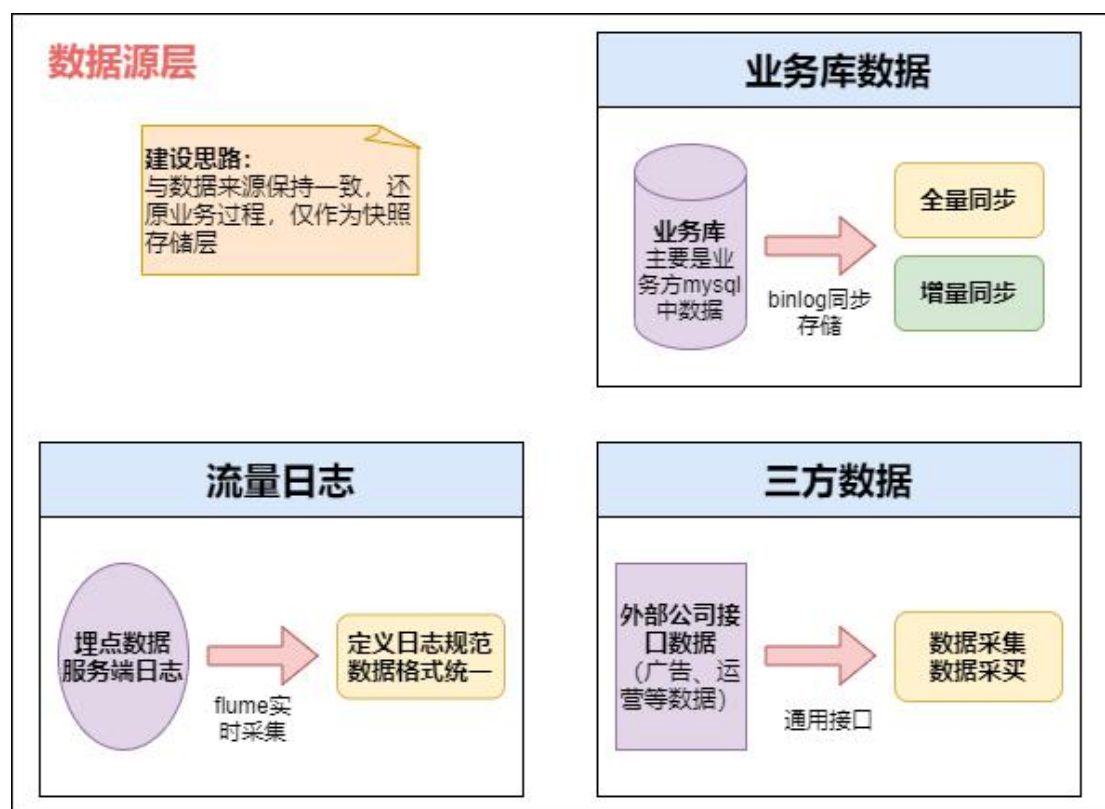
```
写sql代码
```

```
'''
```

## 5. 数据层具体实现

使用四张图说明每层的具体实现

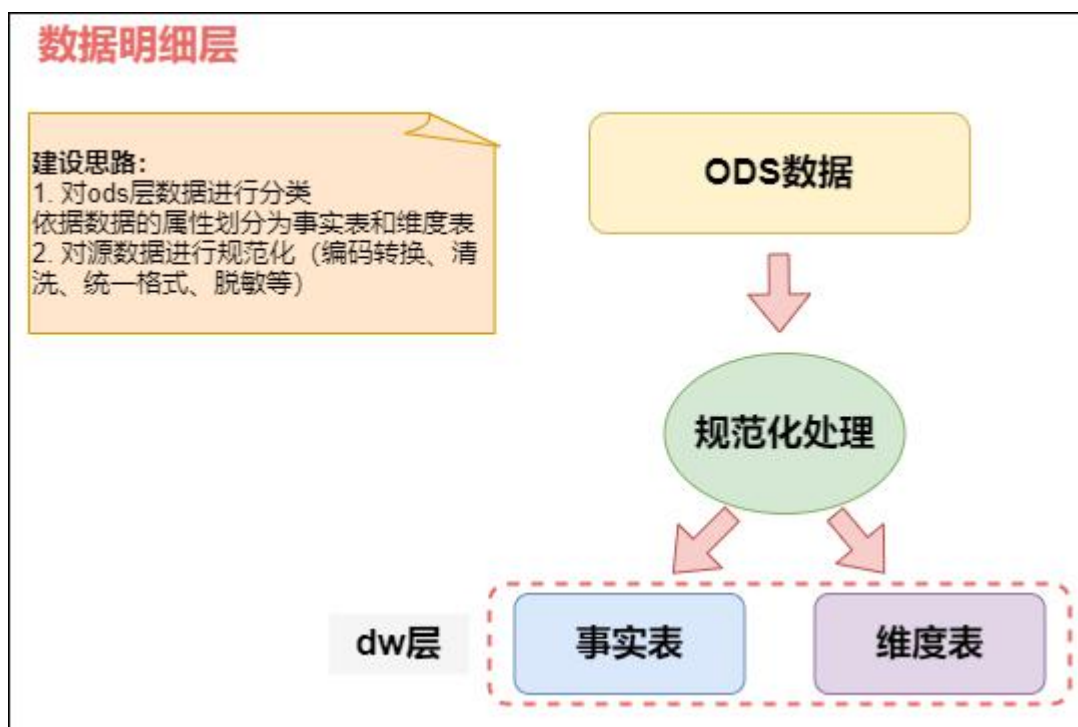
- 数据源层 ODS



数据源层

数据源层主要将各个业务数据导入到大数据平台，作为业务数据的快照存储。

- 数据明细层 DW



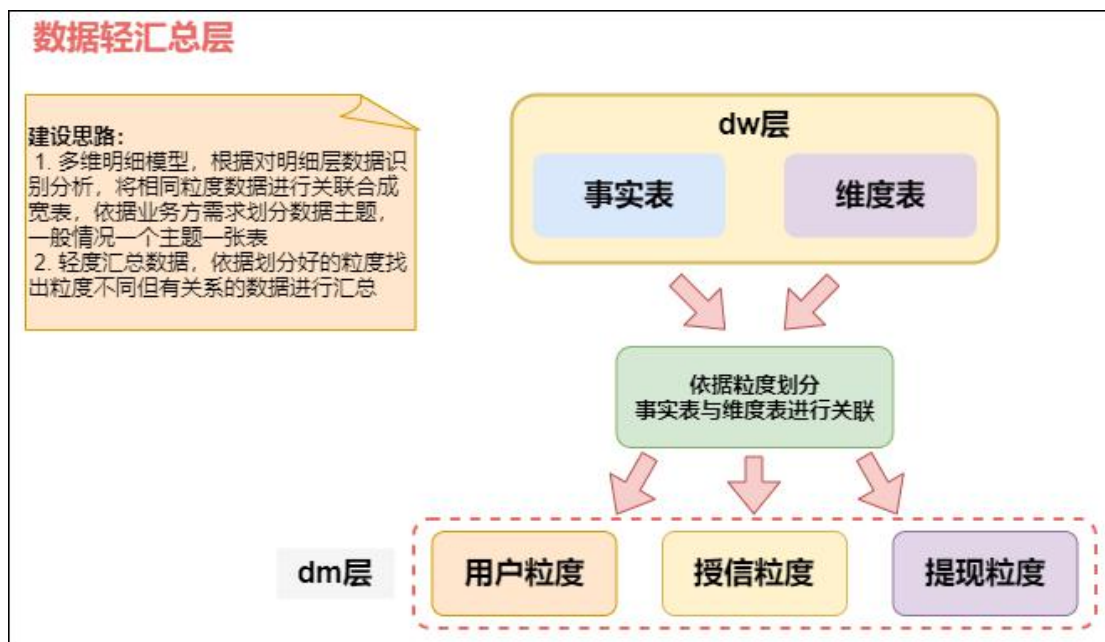
## 数据明细层

事实表中的每行对应一个度量，每行中的数据是一个特定级别的细节数据，称为粒度。维度建模的核心原则之一是**同一事实表中的所有度量必须具有相同的粒度**。这样能确保不会出现重复计算度量的问题。

维度表一般都是单一主键，少数是联合主键，注意维度表不要出现重复数据，否则和事实表关联会出现**数据发散**问题。

有时候往往不能确定该列数据是事实属性还是维度属性。记住**最实用的事实就是数值类型和可加类事实**。所以可以通过分析该列是否是一种包含多个值并作为计算的参与者的度量，这种情况下该列往往是事实；如果该列是对具体值的描述，是一个文本或常量，某一约束和行标识的参与者，此时该属性往往是维度属性。但是还是要结合业务进行最终判断是维度还是事实。

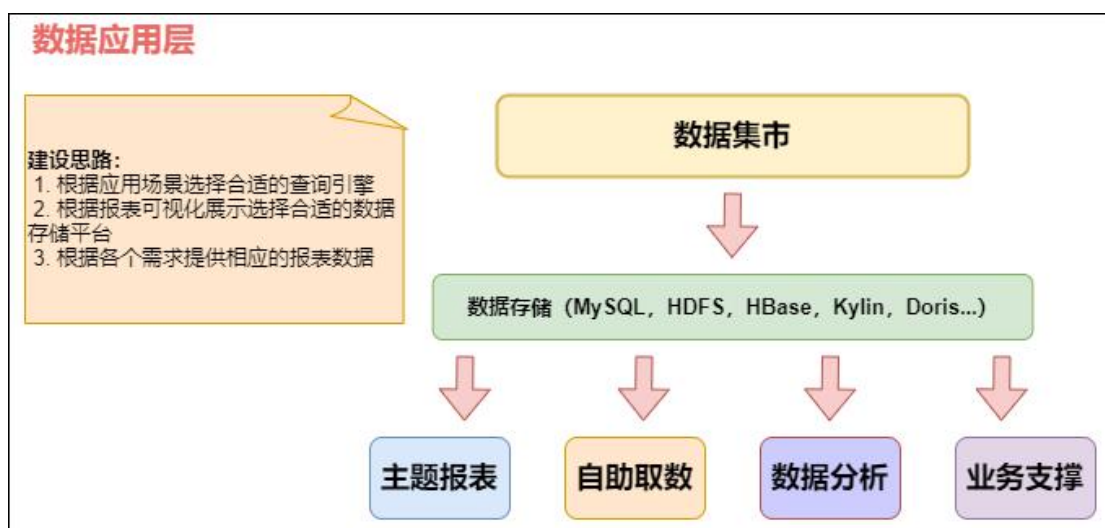
- **数据轻度汇总层 DM**



### 数据轻度汇总层

此层命名为轻汇总层，就代表这一层已经开始对数据进行汇总，但是不是完全汇总，只是对相同粒度的数据进行关联汇总，不同粒度但是有关系的数据也可进行汇总，此时需要将粒度通过聚合等操作进行统一。

- **数据应用层 APP**

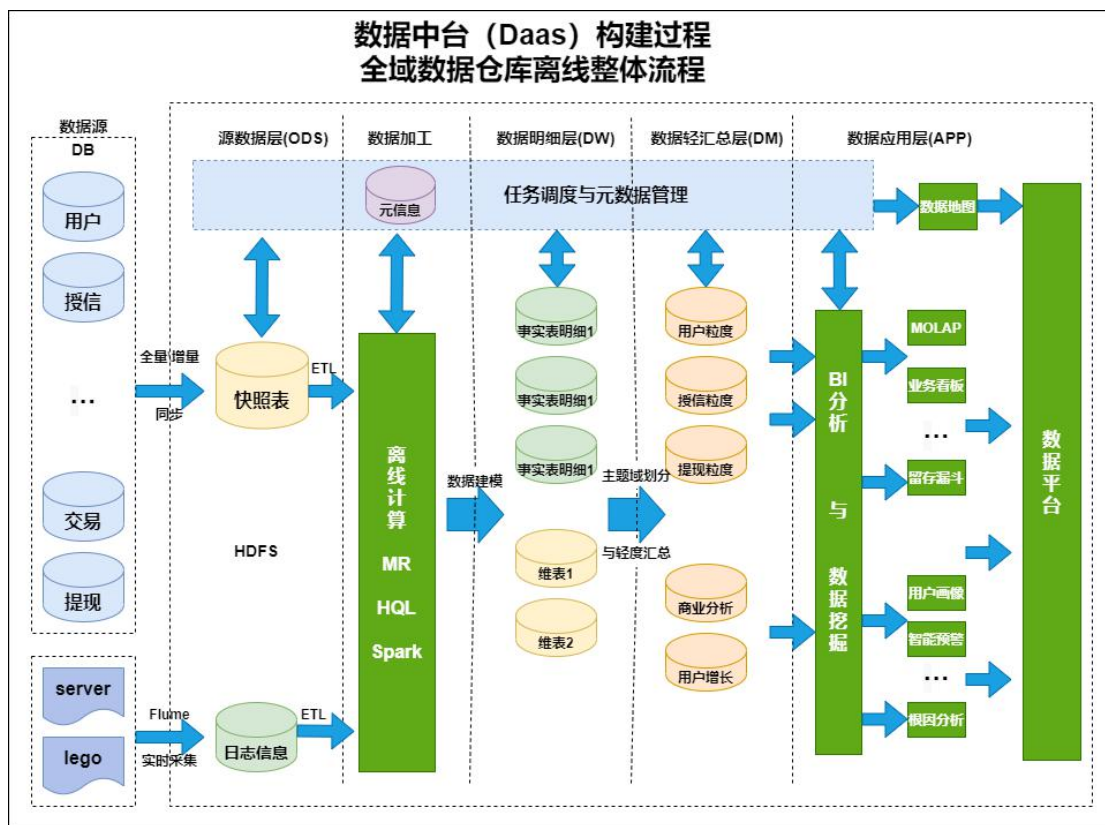


### 数据应用层

数据应用层的表就是提供给用户使用的，数仓建设到此就接近尾声了，接下来就根据不同的需求进行不同的取数，如直接进行报表展示，或提供给数据分析的同事所需的数据，或其他的业务支撑。

## 6. 总结

一张图总结下数据仓库的构建整体流程：



数据中台

## 7. 实际生产中注意事项

生产环境中操作不能像我们自己测试时那样随意，一不小心都可能造成生产事故。所以每步操作都要十分小心，需全神贯注，管好大脑管住右手。

仅列出以下但不限于以下的注意事项：

- 请勿操作自己管理及授权表之外的其它库表；
- 未经授权，请勿操作生产环境中其他人的脚本及文件；
- 在修改生产环境脚本前，请务必自行备份到本地；
- 请确认自己的修改操作能迅速回滚；
- 生产环境中表名及字段等所有命名请遵循命名规则。

推荐阅读：

结合公司业务分析离线数仓建设

数仓建设中最常用模型——Kimball 维度建模详解

## 四、实时计算

实时计算一般都是针对海量数据进行的，并且要求为秒级。由于大数据兴起之初，Hadoop 并没有给出实时计算解决方案，随后 Storm, SparkStreaming, Flink 等实时计算框架应运而生，而 Kafka, ES 的兴起使得实时计算领域的技术越来越完善，而随着物联网，机器学习等技术的推广，实时流式计算将在这些领域得到充分的应用。

实时计算的三个特征：

1. **无限数据**：无限数据指的是一种不断增长的，基本上无限的数据集。这些通常被称为“流数据”，而与之相对的是有限的数据集。
2. **无界数据处理**：一种持续的数据处理模式，能够通过处理引擎重复的去处理上面的无限数据，是能够突破有限数据处理引擎的瓶颈的。
3. **低延迟**：延迟是多少并没有明确的定义。但我们都知道数据的价值将随着时间的流逝降低，时效性将是需要持续解决的问题。

现在大数据应用比较火爆的领域，比如推荐系统在实践之初受技术所限，可能要一分钟，一小时，甚至更久对用户进行推荐，这远远不能满足需要，我们需要更快的完成对数据的处理，而不是进行离线的批处理。

## 1. 实时计算应用场景

随着实时技术发展趋于成熟，实时计算应用越来越广泛，以下仅列举常见的几种实时计算的应用场景：

### 1. 实时智能推荐

智能推荐会根据用户历史的购买或浏览行为，通过推荐算法训练模型，预测用户未来可能会购买的物品或喜爱的资讯。对个人来说，推荐系统起着信息过滤的作用，对 Web/App 服务端来说，推荐系统起着满足用户个性化需求，提升用户满意度的作用。推荐系统本身也在飞速发展，除了算法越来越完善，对时延的要求也越来越苛刻和实时化。利用 Flink 流计算帮助用户构建更加实时的智能推荐系统，对用户行为指标进行实时计算，对模型进行实时更新，对用户指标进行实时预测，并将预测的信息推送给 Web/App 端，帮助用户获取想要的商品信息，另一方面也帮助企业提升销售额，创造更大的商业价值。

### 2. 实时欺诈检测

在金融领域的业务中，常常出现各种类型的欺诈行为，例如信用卡欺诈，信贷申请欺诈等，而如何保证用户和公司的资金安全，是近年来许多金融公司及银行共同面临的挑战。随着不法分子欺诈手段的不断升级，传统的反欺诈手段已经不足



以解决目前所面临的问题。以往可能需要几个小时才能通过交易数据计算出用户的行为指标，然后通过规则判别出具有欺诈行为嫌疑的用户，再进行案件调查处理，在这种情况下资金可能早已被不法分子转移，从而给企业和用户造成大量的经济损失。而运用 Flink 流式计算技术能够在毫秒内就完成对欺诈行为判断指标的计算，然后实时对交易流水进行实时拦截，避免因处理不及时而导致的经济损失。

### 3. 舆情分析

有的客户需要做舆情分析，要求所有数据存放若干年，舆情数据每日数据量可能超百万，年数据量可达到几十亿的数据。而且爬虫爬过来的数据是舆情，通过大数据技术进行分词之后得到的可能是大段的网友评论，客户往往要求对舆情进行查询，做全文本搜索，并要求响应时间控制在秒级。爬虫将数据爬到大数据平台的 Kafka 里，在里面做 Flink 流处理，去重去噪做语音分析，写到 Elasticsearch 里。大数据的一个特点是多数据源，大数据平台能根据不同的场景选择不同的数据源。

### 4. 复杂事件处理

对于复杂事件处理，比较常见的集中于工业领域，例如对车载传感器，机械设备等实时故障检测，这些业务类型通常数据量都非常大，且对数据处理的时效性要求非常高。通过利用 Flink 提供的 CEP 进行时间模式的抽取，同时应用 Flink 的 Sql 进行事件数据的转换，在流式系统中构建实施规则引擎，一旦事件触发报警规则，便立即将告警结果通知至下游通知系统，从而实现对设备故障快速预警检测，车辆状态监控等目的。

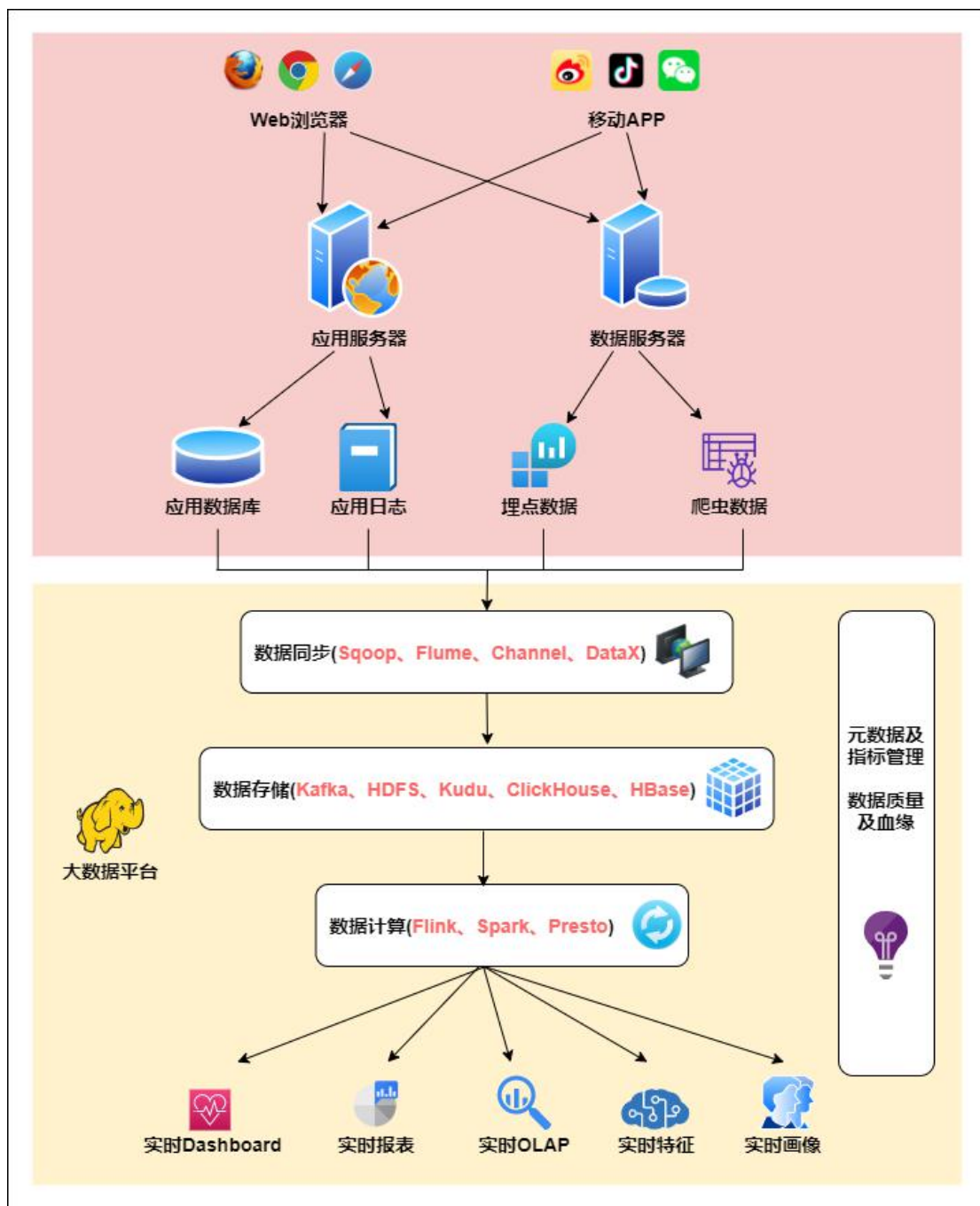
### 5. 实时机器学习

实时机器学习是一个更宽泛的概念，传统静态的机器学习主要侧重于静态的模型和历史数据进行训练并提供预测。很多时候用户的短期行为，对模型有修正作用，或者说是业务判断有预测作用。对系统来说，需要采集用户最近的行为并进行特征工程，然后给到实时机器学习系统进行机器学习。如果动态地实施新规则，或是推出新广告，就会有很大的参考价值。

## 2. 实时计算总览

我们先来看一张大数据平台的实时架构图：





- **数据同步：**

在上面这张架构图中，数据从 Web 平台中产生，通过数据同步系统导入到大数据平台，由于数据源不同，这里的数据同步系统实际上是多个相关系统的组合。数据库同步通常用 Sqoop，日志同步可以选择 Flume 等，不同的数据源产生的数据质量可能差别很大，数据库中的格式化数据直接导入大数据系统即可，而日志和爬虫产生的数据就需要进行大量的清洗、转化处理才能有效使用。

- **数据存储：**

该层对原始数据、清洗关联后的明细数据进行存储，基于统一的实时数据模型分层理念，将不同应用场景的数据分别存储在 Kafka、HDFS、Kudu、Clickhouse、Hbase 等存储中。

- **数据计算：**

计算层主要使用 Flink、Spark、Presto 以及 ClickHouse 自带的计算能力等四种计算引擎，Flink 计算引擎主要用于实时数据同步、流式 ETL、关键系统秒级实时指标计算场景，Spark SQL 主要用于复杂多维分析的准实时指标计算需求场景，Presto 和 ClickHouse 主要满足多维自助分析、对查询响应时间要求不太高的场景。

- **实时应用：**

以统一查询服务对各个业务线数据场景进行支持，业务主要包括实时大屏、实时数据产品、实时 OLAP、实时特征等。

当然一个好的大数据平台不能缺少元数据管理及数据治理：

**1. 元数据及指标管理：**主要对实时的 Kafka 表、Kudu 表、Clickhouse 表、Hive 表等进行统一管理，以数仓模型中表的命名方式规范表的命名，明确每张表的字段含义、使用方，指标管理则是尽量通过指标管理系统将所有的实时指标统一管理起来，明确计算口径，提供给不同的业务方使用；

**2. 数据质量及血缘分析：**数据质量分为平台监控和数据监控两个部分，血缘分析则主要是对实时数据依赖关系、实时任务的依赖关系进行分析。

以上架构只是大数据平台通用的数据模型，如果要具体的建设，需要考虑以下情况，业务需求需要实时还是准实时即可，数据时效性是秒级还是分钟级等。

- 在**调度开销**方面，准实时数据是批处理过程，因此仍然需要调度系统支持，调度频率较高，而实时数据却没有调度开销；
- 在**业务灵活性**方面，因为准实时数据是基于 ETL 或 OLAP 引擎实现，灵活性优于基于流计算的方式；
- 在**对数据晚到的容忍度**方面，因为准实时数据可以基于一个周期内的数据进行全量计算，因此对于数据晚到的容忍度也是比较高的，而实时数据使用的是增量计算，对于数据晚到的容忍度更低一些；
- 在**适用场景**方面，准实时数据主要用于有实时性要求但不太高、涉及多表关联和业务变更频繁的场景，如交易类型的实时分析，实时数据则更适用

于实时性要求高、数据量大的场景，如实时特征、流量类型实时分析等场景。

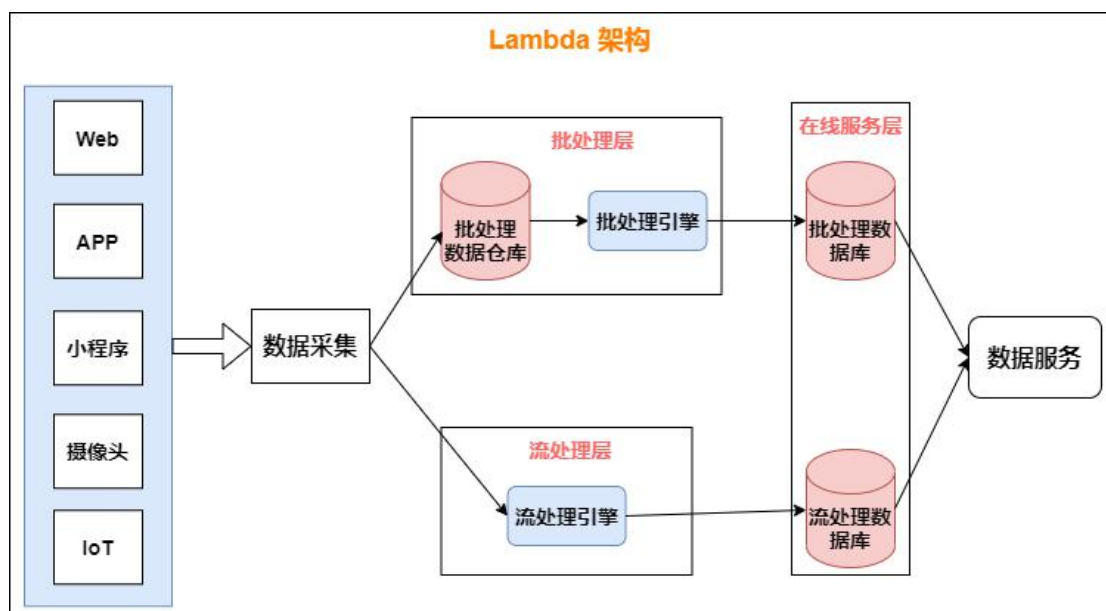
### 3. 实时架构

在某些场景中，数据的价值随着时间的推移而逐渐减少。所以在传统大数据离线数仓的基础上，逐渐对数据的实时性提出了更高的要求。

于是随之诞生了大数据实时数仓，并且衍生出了两种技术架构 Lambda 和 Kappa。

#### 1. Lambda 架构

先来看下 Lambda 架构图：



Lambda 架构图

数据从底层的数据源开始，经过 Kafka、Flume 等数据组件进行收集，然后分成两条线进行计算：

- 一条线是进入流式计算平台（例如 Storm、Flink 或者 SparkStreaming），去计算实时的一些指标；
- 另一条线进入批量数据处理离线计算平台（例如 Mapreduce、Hive，Spark SQL），去计算 T+1 的相关业务指标，这些指标需要隔日才能看见。

#### 为什么 Lambda 架构要分成两条线计算？

假如整个系统只有一个批处理层，会导致用户必须等待很久才能获取计算结果，一般有几个小时的延迟。电商数据分析部门只能查看前一天的统计分析结果，无

法获取当前的结果，这对于实时决策来说有一个巨大的时间鸿沟，很可能导致管理者错过最佳决策时机。

Lambda 架构属于较早的一种架构方式，早期的流处理不如现在这样成熟，在准确性、扩展性和容错性上，流处理层无法直接取代批处理层，只能给用户提供一个近似结果，还不能为用户提供一个一致准确的结果。因此 Lambda 架构中，出现了批处理和流处理并存的现象。

在 Lambda 架构中，每层都有自己所肩负的任务。

### 1. 批处理层存储管理主数据集（不可变的数据集）和预先批处理计算好的视图：

批处理层使用可处理大量数据的分布式处理系统预先计算结果。它通过处理所有的已有历史数据来实现数据的准确性。这意味着它是基于完整的数据集来重新计算的，能够修复任何错误，然后更新现有的数据视图。输出通常存储在只读数据库中，更新则完全取代现有的预先计算好的视图。

### 2. 流处理层会实时处理新来的大数据：

流处理层通过提供最新数据的实时视图来最小化延迟。流处理层所生成的数据视图可能不如批处理层最终生成的视图那样准确或完整，但它们几乎在收到数据后立即可用。而当同样的数据在批处理层处理完成后，在速度层的数据就可以被替代掉了。

### 那 Lambda 架构有没有缺点呢？

Lambda 架构经历多年的发展，其优点是稳定，对于实时计算部分的计算成本可控，批量处理可以用晚上的时间来整体批量计算，这样把实时计算和离线计算高峰分开，这种架构支撑了数据行业的早期发展，但是它也有一些致命缺点，并在大数据 3.0 时代越来越不适应数据分析业务的需求。缺点如下：

- **使用两套大数据处理引擎：**维护两个复杂的分布式系统，成本非常高。
- **批量计算在计算窗口内无法完成：**在 IOT 时代，数据量级越来越大，经常发现夜间只有 4、5 个小时的时间窗口，已经无法完成白天 20 多个小时累计的数据，保证早上上班前准时出数据已成为每个大数据团队头疼的问题。
- **数据源变化都要重新开发，开发周期长：**每次数据源的格式变化，业务的逻辑变化都需要针对 ETL 和 Streaming 做开发修改，整体开发周期很长，业务反应不够迅速。

导致 Lambda 架构的缺点根本原因是同时要同时维护两套系统架构：批处理层和速度层。我们已经知道，在架构中加入批处理层是因为从批处理层得到的结果具有高准确性，而加入速度层是因为它在处理大规模数据时具有低延时性。

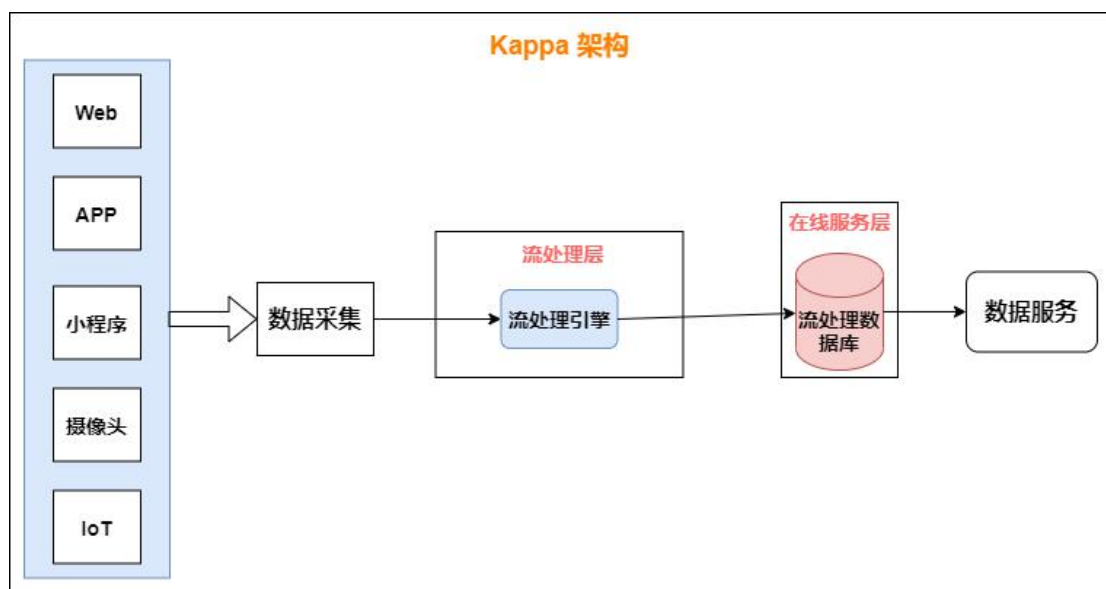
那我们能不能改进其中某一层的架构，让它具有另外一层架构的特性呢？

例如，改进批处理层的系统让它具有更低的延时性，又或者是改进速度层的系统，让它产生的数据视图更具准确性和更加接近历史数据呢？

另外一种在大规模数据处理中常用的架构——Kappa 架构，便是在这样的思考下诞生的。

## 2. Kappa 架构

Kafka 的创始人 Jay Kreps 认为在很多场景下，维护一套 Lambda 架构的大数据处理平台耗时耗力，于是提出在某些场景下，没有必要维护一个批处理层，直接使用一个流处理层即可满足需求，即下图所示的 Kappa 架构：



### Kappa 架构

这种架构只关注流式计算，数据以流的方式被采集过来，实时计算引擎将计算结果放入数据服务层以供查询。可以认为 Kappa 架构是 Lambda 架构的一个简化版本，只是去除了 Lambda 架构中的离线批处理部分；

Kappa 架构的兴起主要有两个原因：

- Kafka 不仅起到消息队列的作用，也可以保存更长时间的历史数据，以替代 Lambda 架构中批处理层数据仓库部分。流处理引擎以一个更早的时间作为起点开始消费，起到了批处理的作用。
- Flink 流处理引擎解决了事件乱序下计算结果的准确性问题。

Kappa 架构相对更简单，实时性更好，所需的计算资源远小于 Lambda 架构，随着实时处理的需求在不断增长，更多的企业开始使用 Kappa 架构。但这并不意味着 Kappa 架构能够取代 Lambda 架构。

Lambda 和 kappa 架构都有各自的适用领域；例如流处理与批处理分析流程比较统一，且允许一定的容错，用 Kappa 比较合适，少量关键指标（例如交易金额、业绩统计等）使用 Lambda 架构进行批量计算，增加一次校对过程。

还有一些比较复杂的场景，批处理与流处理产生不同的结果（使用不同的机器学习模型，专家系统，或者实时计算难以处理的复杂计算），可能更适合 Lambda 架构。



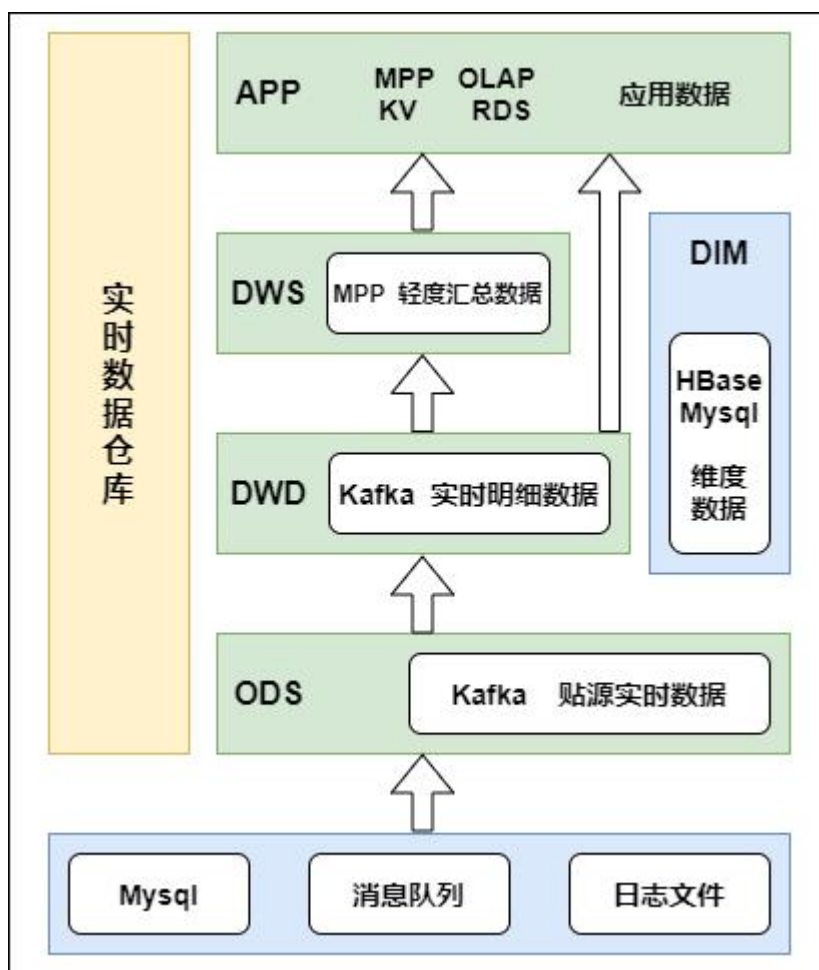
#### 4. 实时数仓解决方案

实时数仓分层架构为了避免面向需求响应的烟囱式构建，**实时数仓也引入了类似于离线数仓的分层理念**，主要是为了提高模型的复用率，同时也要考虑易用性、一致性以及计算成本。

**当然实时数仓的分层架构在设计上并不会像离线数仓那么复杂，避免数据在流转过程中造成的不必要延时响应；**

实时数仓分层架构图：





实时数仓分层架构

1. **ODS 层**：以 Kafka 为支撑，将所有需要实时处理的相关数据放到 Kafka 队列中来实现贴源数据层；
2. **DWD 层**：实时计算订阅业务数据消息队列，然后通过数据清洗、多数据源 join、流式数据与离线维度信息等的组合，将一些相同粒度的业务系统、维表中的维度属性全部关联到一起，增加数据易用性和复用性，得到最终的实时明细数据；
3. **DIM 层**：存放用于关联查询的维度信息，可以根据数据现状来选择存储介质，例如使用 HBase 或者 Mysql
4. **DWS 层**：轻度汇总层是为了便于面向 AdHoc 查询或者 Olap 分析构建的轻度汇总结果集合，适合数据维度、指标信息比较多的情况，为了方便根据自定义条件的快速筛选和指标聚合，推荐使用 MPP 类型数据库进行存储，此层可视场景情况决定是否构建；
5. **APP 层**：面向实时数据场景需求构建的高度汇总层，可以根据不同的数据应用场景决定使用存储介质或者引擎；例如面向业务历史明细、BI 支持等 Olap 分析场景，可以使用 Druid、Greenplum，面向实时监控大屏、高



并发汇总指标等需求，可以使用 KV 模式的 HBase；数据量较小的时候，也可以使用 Mysql 来进行存储。

这里要注意下，其实 APP 层已经脱离了数仓，这里虽然作为了数仓的独立分层，但是实际 APP 层的数据已经分布存储在各种介质中用于使用。

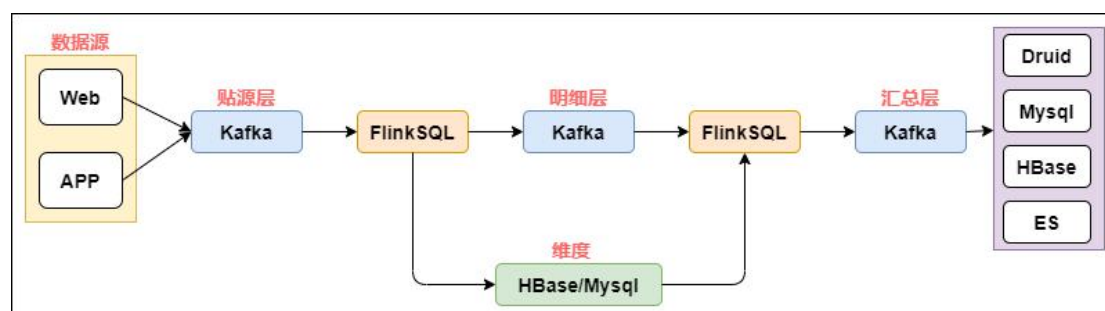
### 基于 Flink 构建的实时数仓

随着业务场景的丰富，更多的实时需求不断涌现，在追求实时任务高吞吐低延迟的同时，对计算过程中间状态管理，灵活时间窗口支持，以及 exactly once 语义保障的诉求也越来越多。

为什么选择 Flink 实时计算平台？之所以选择用 Flink 替代原有 Storm、SparkStreaming 是基于以下原因考虑的，这也是实时数仓关注的核心问题：

1. 高吞吐、低延时；
2. 端到端的 Exactly-once，保证了数据的准确性；
3. 可容错的状态管理，实时数仓里面会进行很多的聚合计算，这些都需要对于状态进行访问和管理；
4. 丰富的 API，对 Streaming/Table/SQL 支持良好，支持 UDF、流式 join、时间窗口等高级用法；
5. 完善的生态体系，实时数仓的构建会涉及多种存储，Flink 在这方面的支持也比较完善。

### 基于 Flink 的实时数仓数据流转过程：



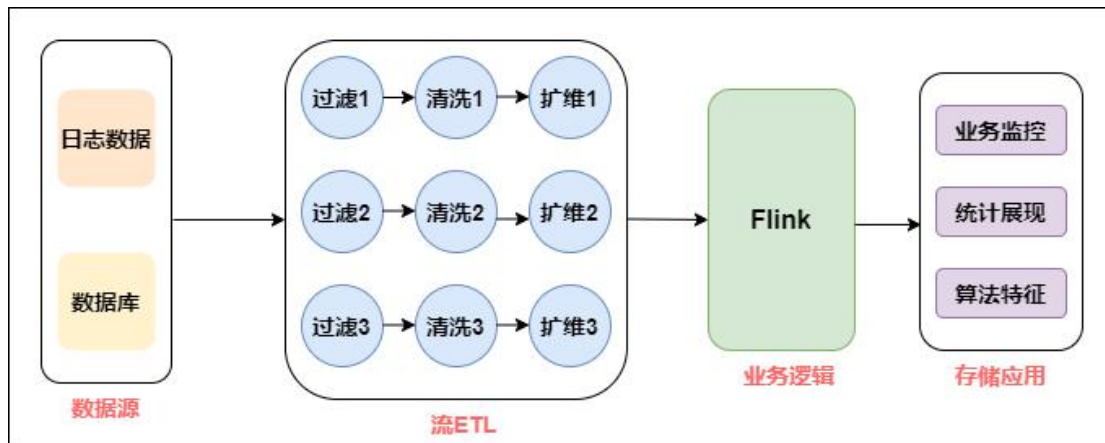
### 实时数仓数据流转过程

数据在实时数仓中的流转过程，实际和离线数仓非常相似，只是由 Flink 替代 Hive 作为了计算引擎，把存储由 HDFS 更换成了 Kafka，但是模型的构建思路与流转过程并没有发生变化。

## 五、实时数仓建设核心

### 1. 实时计算初期

虽然实时计算在最近几年才火起来，但是在早期也有部分公司有实时计算的需求，但是数据量比较少，所以在实时方面形成不了完整的体系，基本所有的开发都是具体问题具体分析，来一个需求做一个，基本不考虑它们之间的关系，开发形式如下：



### 早期实时计算

如上图所示，拿到数据源后，会经过数据清洗，扩维，通过 Flink 进行业务逻辑处理，最后直接进行业务输出。把这个环节拆开来，数据源端会重复引用相同的数据源，后面进行清洗、过滤、扩维等操作，都要重复做一遍，唯一不同的是业务的代码逻辑是不一样的。

随着产品和业务人员对实时数据需求的不断增多，这种开发模式出现的问题越来越多：

1. 数据指标越来越多，“烟囱式”的开发导致代码耦合问题严重。
2. 需求越来越多，有的需要明细数据，有的需要 OLAP 分析。单一的开发模式难以应付多种需求。
3. 每个需求都要申请资源，导致资源成本急速膨胀，资源不能集约有效利用。
4. 缺少完善的监控系统，无法在对业务产生影响之前发现并修复问题。

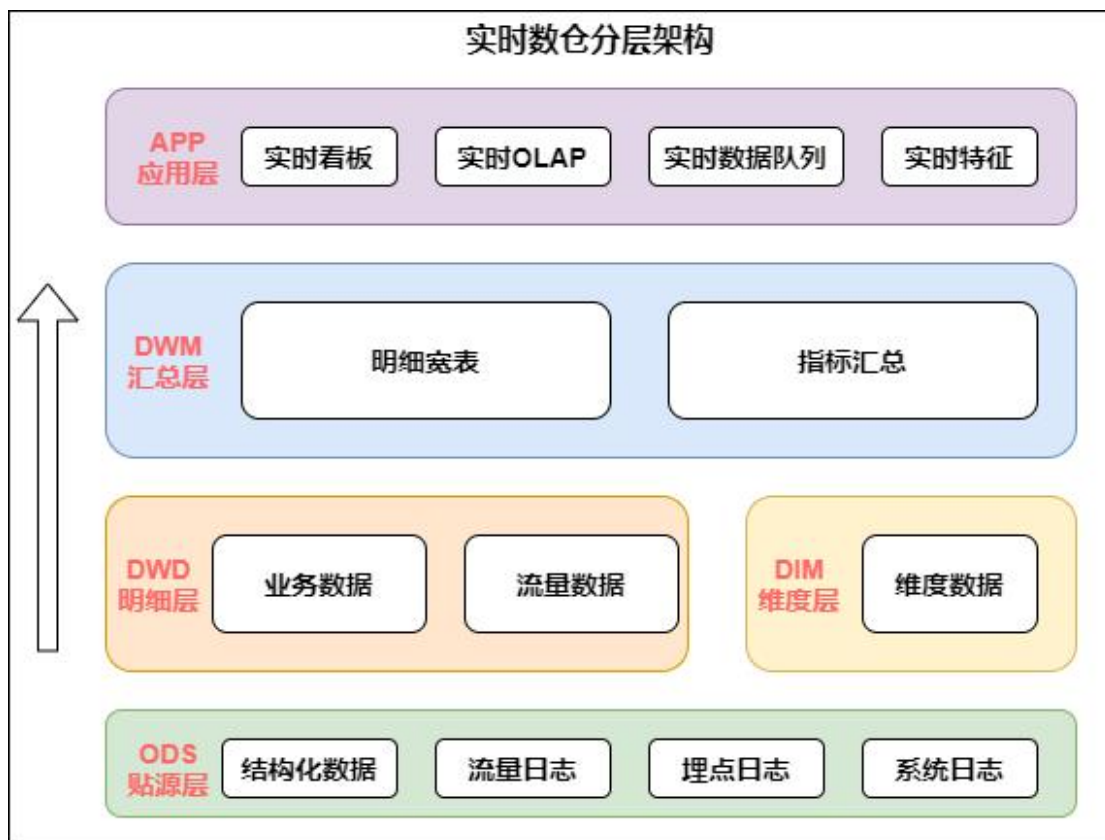
大家看实时数仓的发展和出现的问题，和离线数仓非常类似，后期数据量大了之后产生了各种问题，离线数仓当时是怎么解决的？离线数仓通过分层架构使数据解耦，多个业务可以共用数据，实时数仓是否也可以用分层架构呢？当然是可以的，但是细节上和离线的分层还是有一些不同，稍后会讲到。

## 2. 实时数仓建设

从方法论来讲，实时和离线是非常相似的，离线数仓早期的时候也是具体问题具体分析，当数据规模涨到一定量的时候才会考虑如何治理。分层是一种非常有效

的数据治理方式，所以在实时数仓如何进行管理的问题上，首先考虑的也是分层的处理逻辑。

实时数仓的架构如下图：



实时数仓架构

从上图中我们具体分析下每层的作用：

- 数据源：在数据源的层面，离线和实时在数据源是一致的，主要分为日志类和业务类，日志类又包括用户日志，埋点日志以及服务器日志等。
- 实时明细层：在明细层，为了解决重复建设的问题，要进行统一构建，利用离线数仓的模式，建设统一的基础明细数据层，按照主题进行管理，明细层的目的是给下游提供直接可用的数据，因此要对基础层进行统一的加工，比如清洗、过滤、扩维等。
- 汇总层：汇总层通过 Flink 的简洁算子直接可以算出结果，并且形成汇总指标池，所有的指标都统一在汇总层加工，所有人按照统一的规范管理建设，形成可复用的汇总结果。

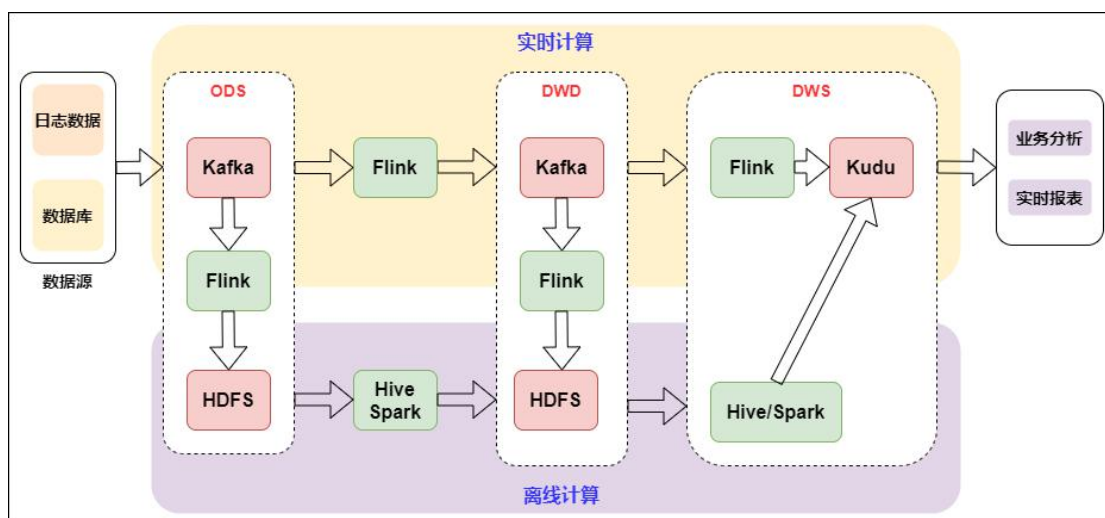
我们可以看出，实时数仓和离线数仓的分层非常类似，比如 数据源层，明细层，汇总层，乃至应用层，他们命名的模式可能都是一样的。但仔细比较不难发现，两者有很多区别：

- 与离线数仓相比，实时数仓的层次更少一些：
  - 从目前建设离线数仓的经验来看，数仓的数据明细层内容会非常丰富，处理明细数据外一般还会包含轻度汇总层的概念，另外离线数仓中应用层数据在数仓内部，**但实时数仓中，app 应用层数据已经落入应用系统的存储介质中，可以把该层与数仓的表分离。**
  - 应用层少建设的好处：**实时处理数据的时候，每建一个层次，数据必然会产生一定的延迟。**
  - 汇总层少建的好处：在汇总统计的时候，往往为了容忍一部分数据的延迟，可能会人为的制造一些延迟来保证数据的准确。举例，在统计跨天相关的订单事件中的数据时，可能会等到 00:00:05 或者 00:00:10 再统计，确保 00:00 前的数据已经全部接受到位了，再进行统计。所以，汇总层的层次太多的话，就会更大的加重人为造成的数据延迟。
- 与离线数仓相比，实时数仓的数据源存储不同：
  - 在建设离线数仓的时候，**基本整个离线数仓都是建立在 Hive 表之上。**但是，在建设实时数仓的时候，同一份表，会使用不同的方式进行存储。比如常见的情况下，**明细数据或者汇总数据都会存在 Kafka 里面，但是像城市、渠道等维度信息需要借助 Hbase, MySQL 或者其他 KV 存储等数据库来进行存储。**

### 3. Lambda 架构的实时数仓

Lambda 和 Kappa 架构的概念已在前文中解释，不了解的小伙伴可点击链接：[一文读懂大数据实时计算](#)

下图是基于 Flink 和 Kafka 的 Lambda 架构的具体实践，上层是实时计算，下层是离线计算，横向是按计算引擎来分，纵向是按实时数仓来区分：

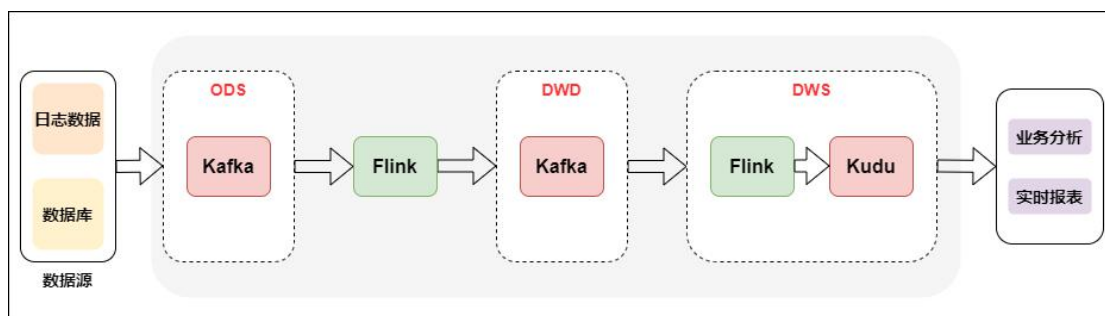


### Lambda 架构的实时数仓

Lambda 架构是比较经典的架构，以前实时的场景不是很多，以离线为主，当附加了实时场景后，由于离线和实时的时效性不同，导致技术生态是不一样的。Lambda 架构相当于附加了一条实时生产链路，在应用层面进行一个整合，双路生产，各自独立。这在业务应用中也是顺理成章采用的一种方式。双路生产会存在一些问题，比如加工逻辑 double，开发运维也会 double，资源同样会变成两个资源链路。因为存在以上问题，所以又演进了一个 Kappa 架构。

## 4. Kappa 架构的实时数仓

Kappa 架构相当于去掉了离线计算部分的 Lambda 架构，具体如下图所示：



### Kappa 架构的实时数仓

Kappa 架构从架构设计来讲比较简单，生产统一，一套逻辑同时生产离线和实时。但是在实际应用场景有比较大的局限性，因为实时数据的同一份表，会使用不同的方式进行存储，这就导致关联时需要跨数据源，操作数据有很大局限性，所以在业内直接用 Kappa 架构生产落地的案例不多见，且场景比较单一。关于 Kappa 架构，熟悉实时数仓生产的同学，可能会有一个疑问。因为我们经常会面临业务变更，所以很多业务逻辑是需要去迭代的。之前产出的一些数据，

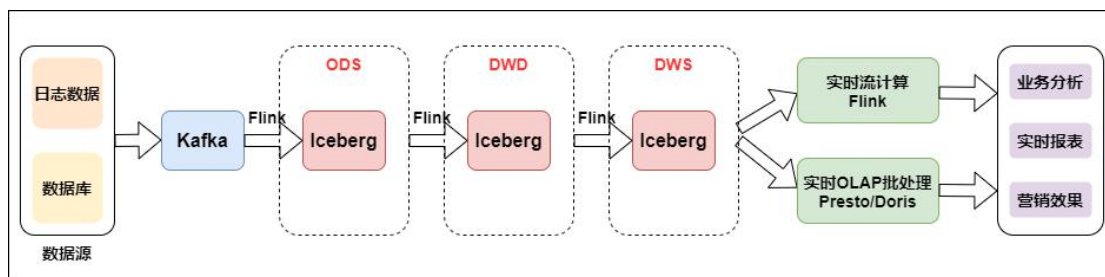
如果口径变更了，就需要重算，甚至重刷历史数据。对于实时数仓来说，怎么去解决数据重算问题？

Kappa 架构在这一块的思路是：首先要准备好一个能够存储历史数据的消息队列，比如 Kafka，并且这个消息队列是可以支持你从某个历史的节点重新开始消费的。接着需要新起一个任务，从原来比较早的一个时间节点去消费 Kafka 上的数据，然后当这个新的任务运行的进度已经能够和现在的正在跑的任务齐平的时候，你就可以把现在任务的下游切换到新的任务上面，旧的任务就可以停掉，并且原来产出的结果表也可以被删掉。

## 5. 流批结合的实时数仓

随着实时 OLAP 技术的发展，目前开源的 OLAP 引擎在性能，易用等方面有了很大的提升，如 Doris、Presto 等，加上数据湖技术的迅速发展，使得流批结合的方式变得简单。

如下图是流批结合的实时数仓：



### 流批结合的实时数仓

数据从日志统一采集到消息队列，再到实时数仓，作为基础数据流的建设是统一的。之后对于日志类实时特征，实时大屏类应用走实时流计算。对于 Binlog 类业务分析走实时 OLAP 批处理。

我们看到流批结合的方式与上面几种架构的存储方式发生了变化，由 Kafka 换成了 Iceberg，Iceberg 是介于上层计算引擎和底层存储格式之间的一个中间层，我们可以把它定义成一种“数据组织格式”，底层存储还是 HDFS，那么为什么加了中间层，就对流批结合处理的比较好了呢？Iceberg 的 ACID 能力可以简化整个流水线的设计，降低整个流水线的延迟，并且所具有的修改、删除能力能够有效地降低开销，提升效率。Iceberg 可以有效支持批处理的高吞吐数据扫描和流计算按分区粒度并发实时处理。

## 六、基于 Flink SQL 从 0 到 1 构建一个实时数仓



注：本小节内容来自公众号大数据技术与数仓！

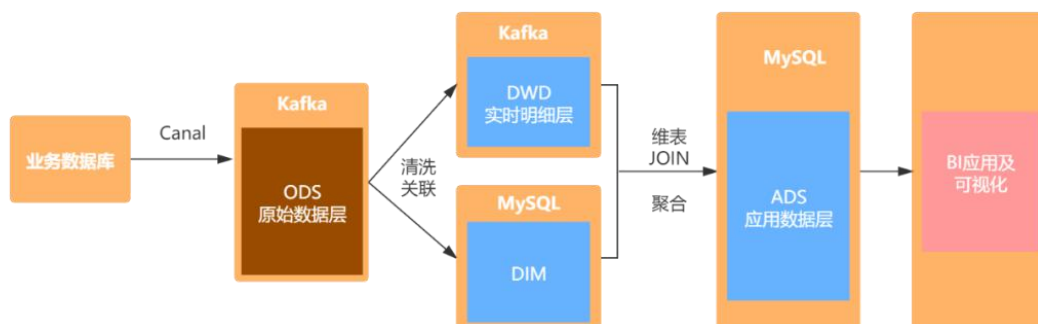
实时数仓主要解决传统数仓数据时效性低的问题，实时数仓通常会用在实时的 OLAP 分析，实时大屏展示，实时监控报警各个场景。虽然关于实时数仓架构及技术选型与传统的离线数仓会存在差异，但是关于数仓建设的基本方法论是一致的。接下来主要介绍 Flink SQL 从 0 到 1 搭建一个实时数仓的 demo，涉及到数据采集、存储、计算、可视化整个流程。

## 1. 案例简介

本文以电商业务为例，展示实时数仓的数据处理流程。另外，本文旨在说明实时数仓的构建流程，所以不会涉及复杂的数据计算。为了保证案例的可操作性和完整性，本文会给出详细的操作步骤。为了方便演示，本文的所有操作都是在 Flink SQL Cli 中完成。

## 2. 架构设计

具体的架构设计如图所示：首先通过 canal 解析 MySQL 的 binlog 日志，将数据存储存储在 Kafka 中。然后使用 Flink SQL 对原始数据进行清洗关联，并将处理之后的明细宽表写入 Kafka 中。维表数据存储在 MySQL 中，通过 Flink SQL 对明细宽表与维表进行 join，将聚合后的数据写入 MySQL，最后通过 FineBI 进行可视化展示。



## 3. 业务数据准备

### 1. 订单表(order\_info)



```
CREATE TABLE `order_info` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',  
  `consignee` varchar(100) DEFAULT NULL COMMENT '收货人',  
  `consignee_tel` varchar(20) DEFAULT NULL COMMENT '收件人电话',  
  `total_amount` decimal(10,2) DEFAULT NULL COMMENT '总金额',  
  `order_status` varchar(20) DEFAULT NULL COMMENT '订单状态',  
  `user_id` bigint(20) DEFAULT NULL COMMENT '用户 id',  
  `payment_way` varchar(20) DEFAULT NULL COMMENT '付款方式',  
  `delivery_address` varchar(1000) DEFAULT NULL COMMENT '送货地址',  
  `order_comment` varchar(200) DEFAULT NULL COMMENT '订单备注',  
  `out_trade_no` varchar(50) DEFAULT NULL COMMENT '订单交易编号 (第三方支付用)',  
  `trade_body` varchar(200) DEFAULT NULL COMMENT '订单描述(第三方支付用)',  
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',  
  `operate_time` datetime DEFAULT NULL COMMENT '操作时间',  
  `expire_time` datetime DEFAULT NULL COMMENT '失效时间',  
  `tracking_no` varchar(100) DEFAULT NULL COMMENT '物流单编号',  
  `parent_order_id` bigint(20) DEFAULT NULL COMMENT '父订单编号',  
  `img_url` varchar(200) DEFAULT NULL COMMENT '图片路径',  
  `province_id` int(20) DEFAULT NULL COMMENT '地区',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='订单表';
```

## 2. 订单详情表(order\_detail)

```
CREATE TABLE `order_detail` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',  
  `order_id` bigint(20) DEFAULT NULL COMMENT '订单编号',  
  `sku_id` bigint(20) DEFAULT NULL COMMENT 'sku_id',  
  `sku_name` varchar(200) DEFAULT NULL COMMENT 'sku 名称 (冗余)',  
  `img_url` varchar(200) DEFAULT NULL COMMENT '图片名称 (冗余)',  
  `order_price` decimal(10,2) DEFAULT NULL COMMENT '购买价格(下单时 sku 价格)',  
  `sku_num` varchar(200) DEFAULT NULL COMMENT '购买个数',  
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='订单详情表';
```

## 3. 商品表(sku\_info)

```
CREATE TABLE `sku_info` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'skuid(itemID)',  
  `spu_id` bigint(20) DEFAULT NULL COMMENT 'spuid',  
  `price` decimal(10,0) DEFAULT NULL COMMENT '价格',
```

```
`sku_name` varchar(200) DEFAULT NULL COMMENT 'sku 名称',
`sku_desc` varchar(2000) DEFAULT NULL COMMENT '商品规格描述',
`weight` decimal(10,2) DEFAULT NULL COMMENT '重量',
`tm_id` bigint(20) DEFAULT NULL COMMENT '品牌(冗余)',
`category3_id` bigint(20) DEFAULT NULL COMMENT '三级分类 id (冗余)',
`sku_default_img` varchar(200) DEFAULT NULL COMMENT '默认显示图片(冗余)',
`create_time` datetime DEFAULT NULL COMMENT '创建时间',
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='商品表';
```

#### 4. 商品一级类目表(base\_category1)

```
CREATE TABLE `base_category1` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `name` varchar(10) NOT NULL COMMENT '分类名称',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='一级分类表';
```

#### 5. 商品二级类目表(base\_category2)

```
CREATE TABLE `base_category2` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `name` varchar(200) NOT NULL COMMENT '二级分类名称',
  `category1_id` bigint(20) DEFAULT NULL COMMENT '一级分类编号',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='二级分类表';
```

#### 6. 商品三级类目表(base\_category3)

```
CREATE TABLE `base_category3` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `name` varchar(200) NOT NULL COMMENT '三级分类名称',
  `category2_id` bigint(20) DEFAULT NULL COMMENT '二级分类编号',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='三级分类表';
```

#### 7. 省份表(区域表 (base\_region) base\_province)

```
CREATE TABLE `base_province` (
  `id` int(20) DEFAULT NULL COMMENT 'id',
```

```
`name` varchar(20) DEFAULT NULL COMMENT '省名称',
`region_id` int(20) DEFAULT NULL COMMENT '大区id',
`area_code` varchar(20) DEFAULT NULL COMMENT '行政区位码'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

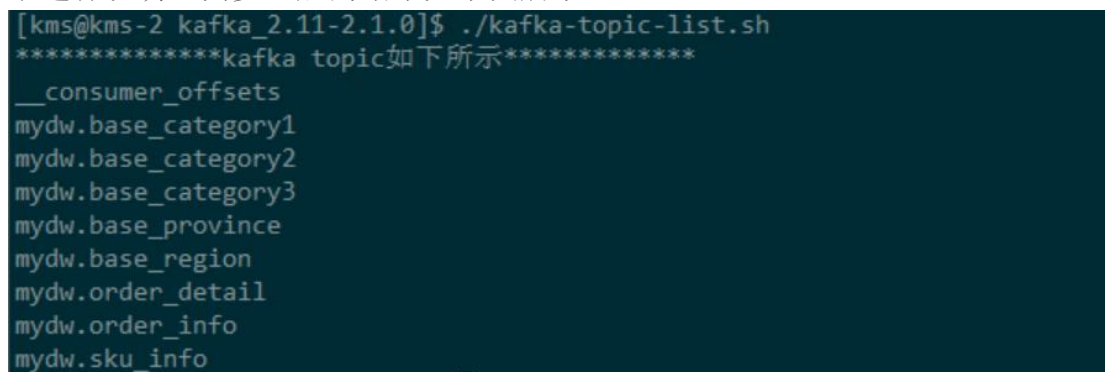
## 8. 区域表(base\_region)

```
CREATE TABLE `base_region` (
  `id` int(20) NOT NULL COMMENT '大区id',
  `region_name` varchar(20) DEFAULT NULL COMMENT '大区名称',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## 4. 数据处理流程

### 1. ods 层数据同步

关于 ODS 层的数据同步这里就不详细展开。主要使用 canal 解析 MySQL 的 binlog 日志，然后将其写入到 Kafka 对应的 topic 中。由于篇幅限制，不会对具体的细节进行说明。同步之后的结果如下图所示：



```
[kms@kms-2 kafka_2.11-2.1.0]$ ./kafka-topic-list.sh
*****kafka topic如下所示*****
__consumer_offsets
mydw.base_category1
mydw.base_category2
mydw.base_category3
mydw.base_province
mydw.base_region
mydw.order_detail
mydw.order_info
mydw.sku_info
```

### 2. DIM 层数据准备

本案例中将维表存储在了 MySQL 中，实际生产中会用 HBase 存储维表数据。我们主要用到两张维表：区域维表和商品维表。处理过程如下：

- 区域维表

首先将 `mydw.base_province` 和 `mydw.base_region` 这个主题对应的数据抽取到 MySQL 中，主要使用 Flink SQL 的 Kafka 数据源对应的 canal-json 格式，注意：在执行装载之前，需要先在 MySQL 中创建对应的表，本文使用的 MySQL 数据库的名字为 `dim`，用于存放维表数据。如下：

```
-- -----
-- 省份
-- kafka Source
-- -----
DROP TABLE IF EXISTS `ods_base_province`;
CREATE TABLE `ods_base_province` (
  `id` INT,
  `name` STRING,
  `region_id` INT,
  `area_code` STRING
) WITH(
  'connector' = 'kafka',
  'topic' = 'mydw.base_province',
  'properties.bootstrap.servers' = 'kms-3:9092',
  'properties.group.id' = 'testGroup',
  'format' = 'canal-json',
  'scan.startup.mode' = 'earliest-offset'
);

-- -----
-- 省份
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `base_province`;
CREATE TABLE `base_province` (
  `id` INT,
  `name` STRING,
  `region_id` INT,
  `area_code` STRING,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/dim',
  'table-name' = 'base_province', -- MySQL 中的待插入数据的表
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe',
  'sink.buffer-flush.interval' = '1s'
```

```
);
--
-- -----
-- 省份
-- MySQL Sink Load Data
-- -----
INSERT INTO base_province
SELECT *
FROM ods_base_province;
--
-- -----
-- 区域
-- kafka Source
-- -----
DROP TABLE IF EXISTS `ods_base_region`;
CREATE TABLE `ods_base_region` (
  `id` INT,
  `region_name` STRING
) WITH(
  'connector' = 'kafka',
  'topic' = 'mydw.base_region',
  'properties.bootstrap.servers' = 'kms-3:9092',
  'properties.group.id' = 'testGroup',
  'format' = 'canal-json',
  'scan.startup.mode' = 'earliest-offset'
);
--
-- -----
-- 区域
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `base_region`;
CREATE TABLE `base_region` (
  `id` INT,
  `region_name` STRING,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/dim',
  'table-name' = 'base_region', -- MySQL 中的待插入数据的表
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe',
  'sink.buffer-flush.interval' = '1s'
```

```
);
-- -----
-- 区域
-- MySQL Sink Load Data
-- -----
INSERT INTO base_region
SELECT *
FROM ods_base_region;
```

经过上面的步骤，将创建维表所需要的原始数据已经存储到了 MySQL 中，接下来就需要在 MySQL 中创建维表，我们使用上面的两张表，创建一张视图：

`dim_province` 作为维表：

```
-- -----
-- DIM 层, 区域维表,
-- 在MySQL 中创建视图
-- -----
DROP VIEW IF EXISTS dim_province;
CREATE VIEW dim_province AS
SELECT
  bp.id AS province_id,
  bp.name AS province_name,
  br.id AS region_id,
  br.region_name AS region_name,
  bp.area_code AS area_code
FROM base_region br
JOIN base_province bp ON br.id= bp.region_id;
```

这样我们所需要的维表：`dim_province` 就创建好了，只需要在维表 join 时，使用 Flink SQL 创建 JDBC 的数据源，就可以使用该维表了。同理，我们使用相同的方法创建商品维表，具体如下：

```
-- -----
-- 一级类目表
-- kafka Source
-- -----
DROP TABLE IF EXISTS `ods_base_category1`;
CREATE TABLE `ods_base_category1` (
  `id` BIGINT,
  `name` STRING
)WITH(
  'connector' = 'kafka',
  'topic' = 'mydw.base_category1',
  'properties.bootstrap.servers' = 'kms-3:9092',
```



```
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset'
);
-- -----
-- 一级类目表
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `base_category1`;
CREATE TABLE `base_category1` (
  `id` BIGINT,
  `name` STRING,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/dim',
  'table-name' = 'base_category1', -- MySQL 中的待插入数据的表
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe',
  'sink.buffer-flush.interval' = '1s'
);
-- -----
-- 一级类目表
-- MySQL Sink Load Data
-- -----
INSERT INTO base_category1
SELECT *
FROM ods_base_category1;
-- -----
-- 二级类目表
-- kafka Source
-- -----
DROP TABLE IF EXISTS `ods_base_category2`;
CREATE TABLE `ods_base_category2` (
  `id` BIGINT,
  `name` STRING,
  `category1_id` BIGINT
) WITH (
  'connector' = 'kafka',
```

```
'topic' = 'mydw.base_category2',
'properties.bootstrap.servers' = 'kms-3:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset'
);

```

```
-- -----
-- 二级类目表
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `base_category2`;
CREATE TABLE `base_category2` (
  `id` BIGINT,
  `name` STRING,
  `category1_id` BIGINT,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/dim',
  'table-name' = 'base_category2', -- MySQL 中的待插入数据的表
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe',
  'sink.buffer-flush.interval' = '1s'
);

```

```
-- -----
-- 二级类目表
-- MySQL Sink Load Data
-- -----
INSERT INTO base_category2
SELECT *
FROM ods_base_category2;

```

```
-- -----
-- 三级类目表
-- kafka Source
-- -----
DROP TABLE IF EXISTS `ods_base_category3`;
CREATE TABLE `ods_base_category3` (
  `id` BIGINT,
  `name` STRING,
  `category2_id` BIGINT

```

```

)WITH(
'connector' = 'kafka',
'topic' = 'mydw.base_category3',
'properties.bootstrap.servers' = 'kms-3:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset'
);

-- -----
-- 三级类目表
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `base_category3`;
CREATE TABLE `base_category3` (
  `id` BIGINT,
  `name` STRING,
  `category2_id` BIGINT,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/dim',
  'table-name' = 'base_category3', -- MySQL 中的待插入数据的表
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe',
  'sink.buffer-flush.interval' = '1s'
);

-- -----
-- 三级类目表
-- MySQL Sink Load Data
-- -----
INSERT INTO base_category3
SELECT *
FROM ods_base_category3;

-- -----
-- 商品表
-- Kafka Source
-- -----

DROP TABLE IF EXISTS `ods_sku_info`;
CREATE TABLE `ods_sku_info` (

```

```

`id` BIGINT,
`spu_id` BIGINT,
`price` DECIMAL(10,0),
`sku_name` STRING,
`sku_desc` STRING,
`weight` DECIMAL(10,2),
`tm_id` BIGINT,
`category3_id` BIGINT,
`sku_default_img` STRING,
`create_time` TIMESTAMP(0)
) WITH(
'connector' = 'kafka',
'topic' = 'mydw.sku_info',
'properties.bootstrap.servers' = 'kms-3:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset'
);
--
-- -----
-- 商品表
-- MySQL Sink
-- -----
DROP TABLE IF EXISTS `sku_info`;
CREATE TABLE `sku_info` (
`id` BIGINT,
`spu_id` BIGINT,
`price` DECIMAL(10,0),
`sku_name` STRING,
`sku_desc` STRING,
`weight` DECIMAL(10,2),
`tm_id` BIGINT,
`category3_id` BIGINT,
`sku_default_img` STRING,
`create_time` TIMESTAMP(0),
PRIMARY KEY (tm_id) NOT ENFORCED
) WITH (
'connector' = 'jdbc',
'url' = 'jdbc:mysql://kms-1:3306/dim',
'table-name' = 'sku_info', -- MySQL 中的待插入数据的表
'driver' = 'com.mysql.jdbc.Driver',
'username' = 'root',
'password' = '123qwe',
'sink.buffer-flush.interval' = '1s'

```

```
);
-- -----
-- 商品
-- MySQL Sink Load Data
-- -----
INSERT INTO sku_info
SELECT *
FROM ods_sku_info;
```

经过上面的步骤，我们可以将创建商品维表的基础数据表同步到 MySQL 中，同样需要提前创建好对应的数据表。接下来我们使用上面的基础表在 MySQL 的 dim 库中创建一张视图：dim\_sku\_info，用作后续使用的维表。

```
-- -----
-- DIM 层, 商品维表,
-- 在MySQL 中创建视图
-- -----
CREATE VIEW dim_sku_info AS
SELECT
    si.id AS id,
    si.sku_name AS sku_name,
    si.category3_id AS c3_id,
    si.weight AS weight,
    si.tm_id AS tm_id,
    si.price AS price,
    si.spu_id AS spu_id,
    c3.name AS c3_name,
    c2.id AS c2_id,
    c2.name AS c2_name,
    c3.id AS c1_id,
    c3.name AS c1_name
FROM
(
    sku_info si
    JOIN base_category3 c3 ON si.category3_id = c3.id
    JOIN base_category2 c2 ON c3.category2_id = c2.id
    JOIN base_category1 c1 ON c2.category1_id = c1.id
);
```

至此，我们所需要的维表数据已经准备好了，接下来开始处理 DWD 层的数据。

### 3. DWD 层数据处理

经过上面的步骤，我们已经将所用的维表已经准备好了。接下来我们将对 ODS 的原始数据进行处理，加工成 DWD 层的明细宽表。具体过程如下：

```
-- -----
--  订单详情
--  Kafka Source
-- -----

DROP TABLE IF EXISTS `ods_order_detail`;
CREATE TABLE `ods_order_detail` (
  `id` BIGINT,
  `order_id` BIGINT,
  `sku_id` BIGINT,
  `sku_name` STRING,
  `img_url` STRING,
  `order_price` DECIMAL(10,2),
  `sku_num` INT,
  `create_time` TIMESTAMP(0)
) WITH(
  'connector' = 'kafka',
  'topic' = 'mydw.order_detail',
  'properties.bootstrap.servers' = 'kms-3:9092',
  'properties.group.id' = 'testGroup',
  'format' = 'canal-json',
  'scan.startup.mode' = 'earliest-offset'
);

-- -----
--  订单信息
--  Kafka Source
-- -----

DROP TABLE IF EXISTS `ods_order_info`;
CREATE TABLE `ods_order_info` (
  `id` BIGINT,
  `consignee` STRING,
  `consignee_tel` STRING,
  `total_amount` DECIMAL(10,2),
  `order_status` STRING,
  `user_id` BIGINT,
  `payment_way` STRING,
  `delivery_address` STRING,
  `order_comment` STRING,
  `out_trade_no` STRING,
  `trade_body` STRING,
```



```

`create_time` TIMESTAMP(0) ,
`operate_time` TIMESTAMP(0) ,
`expire_time` TIMESTAMP(0) ,
`tracking_no` STRING,
`parent_order_id` BIGINT,
`img_url` STRING,
`province_id` INT
) WITH(
'connector' = 'kafka',
'topic' = 'mydw.order_info',
'properties.bootstrap.servers' = 'kms-3:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset'
);

-- -----
-- DWD 层, 支付订单明细表dwd_paid_order_detail
-- -----
DROP TABLE IF EXISTS dwd_paid_order_detail;
CREATE TABLE dwd_paid_order_detail
(
    detail_id BIGINT,
    order_id BIGINT,
    user_id BIGINT,
    province_id INT,
    sku_id BIGINT,
    sku_name STRING,
    sku_num INT,
    order_price DECIMAL(10,0),
    create_time STRING,
    pay_time STRING
) WITH (
    'connector' = 'kafka',
    'topic' = 'dwd_paid_order_detail',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'kms-3:9092',
    'format' = 'changelog-json'
);

-- -----
-- DWD 层, 已支付订单明细表
-- 向dwd_paid_order_detail 装载数据
-- -----
INSERT INTO dwd_paid_order_detail

```

```
SELECT
  od.id,
  oi.id order_id,
  oi.user_id,
  oi.province_id,
  od.skus_id,
  od.skus_name,
  od.skus_num,
  od.order_price,
  oi.create_time,
  oi.operate_time
FROM
  (
    SELECT *
    FROM ods_order_info
    WHERE order_status = '2' -- 已支付
  ) oi JOIN
  (
    SELECT *
    FROM ods_order_detail
  ) od
  ON oi.id = od.order_id;
```

## 4. ADS 层数据

经过上面的步骤，我们创建了一张 `dwd_paid_order_detail` 明细宽表，并将该表存储在了 Kafka 中。接下来我们将使用这张明细宽表与维表进行 JOIN，得到我们 ADS 应用层数据。

- `ads_province_index`

首先在 MySQL 中创建对应的 ADS 目标表：`ads_province_index`

```
CREATE TABLE ads.ads_province_index(
  province_id INT(10),
  area_code VARCHAR(100),
  province_name VARCHAR(100),
  region_id INT(10),
  region_name VARCHAR(100),
  order_amount DECIMAL(10,2),
  order_count BIGINT(10),
  dt VARCHAR(100),
```

```
PRIMARY KEY (province_id, dt)
);
```

向 MySQL 的 ADS 层目标装载数据：

```
-- Flink SQL Cli 操作
-- -----
-- 使用 DDL 创建MySQL 中的ADS 层表
-- 指标: 1. 每天每个省份的订单数
--       2. 每天每个省份的订单金额
-- -----
CREATE TABLE ads_province_index(
  province_id INT,
  area_code STRING,
  province_name STRING,
  region_id INT,
  region_name STRING,
  order_amount DECIMAL(10,2),
  order_count BIGINT,
  dt STRING,
  PRIMARY KEY (province_id, dt) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://kms-1:3306/ads',
  'table-name' = 'ads_province_index',
  'driver' = 'com.mysql.jdbc.Driver',
  'username' = 'root',
  'password' = '123qwe'
);
-- -----
-- dwd_paid_order_detail 已支付订单明细宽表
-- -----
CREATE TABLE dwd_paid_order_detail
(
  detail_id BIGINT,
  order_id BIGINT,
  user_id BIGINT,
  province_id INT,
  sku_id BIGINT,
  sku_name STRING,
  sku_num INT,
  order_price DECIMAL(10,2),
  create_time STRING,
  pay_time STRING
) WITH (
```

```

    'connector' = 'kafka',
    'topic' = 'dwd_paid_order_detail',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'kms-3:9092',
    'format' = 'changelog-json'
);

```

```

-- -----
-- tmp_province_index
-- 订单汇总临时表
-- -----
CREATE TABLE tmp_province_index(
    province_id INT,
    order_count BIGINT, -- 订单数
    order_amount DECIMAL(10,2), -- 订单金额
    pay_date DATE
)WITH (
    'connector' = 'kafka',
    'topic' = 'tmp_province_index',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'kms-3:9092',
    'format' = 'changelog-json'
);

```

```

-- -----
-- tmp_province_index
-- 订单汇总临时表数据装载
-- -----
INSERT INTO tmp_province_index
SELECT
    province_id,
    count(distinct order_id) order_count, -- 订单数
    sum(order_price * sku_num) order_amount, -- 订单金额
    TO_DATE(pay_time, 'yyyy-MM-dd') pay_date
FROM dwd_paid_order_detail
GROUP BY province_id, TO_DATE(pay_time, 'yyyy-MM-dd')
;

```

```

-- -----
-- tmp_province_index_source
-- 使用该临时汇总表，作为数据源
-- -----
CREATE TABLE tmp_province_index_source(
    province_id INT,
    order_count BIGINT, -- 订单数
    order_amount DECIMAL(10,2), -- 订单金额

```

```

    pay_date DATE,
    proctime as PROCTIME() -- 通过计算列产生一个处理时间列
) WITH (
    'connector' = 'kafka',
    'topic' = 'tmp_province_index',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'kms-3:9092',
    'format' = 'changelog-json'
);

-- -----
-- DIM 层, 区域维表,
-- 创建区域维表数据源
-- -----
DROP TABLE IF EXISTS `dim_province`;
CREATE TABLE dim_province (
    province_id INT,
    province_name STRING,
    area_code STRING,
    region_id INT,
    region_name STRING,
    PRIMARY KEY (province_id) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://kms-1:3306/dim',
    'table-name' = 'dim_province',
    'driver' = 'com.mysql.jdbc.Driver',
    'username' = 'root',
    'password' = '123qwe',
    'scan.fetch-size' = '100'
);

-- -----
-- 向ads_province_index 装载数据
-- 维表 JOIN
-- -----

INSERT INTO ads_province_index
SELECT
    pc.province_id,
    dp.area_code,
    dp.province_name,
    dp.region_id,
    dp.region_name,

```

```
pc.order_amount,
pc.order_count,
cast(pc.pay_date as VARCHAR)
FROM
tmp_province_index_source pc
JOIN dim_province FOR SYSTEM_TIME AS OF pc.proctime as dp
ON dp.province_id = pc.province_id;
```

当提交任务之后：观察 Flink WEB UI:



default: INSERT INTO ads\_province\_index SELECT pc.province\_id, dp.area\_code, dp.province\_name, dp.region\_id, dp.region\_name, pc.order\_amount, pc.order\_count, cast(pc.pay\_date as VARCHAR) FROM tmp\_province\_index\_source pc JOIN dim\_province FOR SYSTEM\_TIME AS OF pc.proctime as dp ON dp.province\_id = pc.province\_id

Source: TableSourceScanTable  
 (default catalog, default database, tmp\_province\_index\_source) fields: (province\_id, order\_count, order\_amount, pay\_date) => LookupJoinTableSink (default catalog, default database, dim\_province) joinType: (innerJoin), asyn: false, lookup: (province\_id=province\_id), select: \*

Parallelism: 1

查看 ADS 层的 ads\_province\_index 表数据:

```
Flink SQL> select province_name,region_name,order_amount,order_count ,dt from ads_province_index;
```

province_name	region_name	order_amount	order_count	dt
山西	华北	65400.000...	2	2020-06-18
上海	华东	56284.000...	1	2020-06-18
江苏	华东	15976.000...	1	2020-06-18
浙江	华东	45480.000...	1	2020-06-18
安徽	华东	25384.000...	1	2020-06-18

Received a total of 5 rows

- ads\_sku\_index

首先在 MySQL 中创建对应的 ADS 目标表: ads\_sku\_index

```
CREATE TABLE ads_sku_index
(
sku_id BIGINT(10),
sku_name VARCHAR(100),
weight DOUBLE,
tm_id BIGINT(10),
price DOUBLE,
spu_id BIGINT(10),
c3_id BIGINT(10),
c3_name VARCHAR(100) ,
c2_id BIGINT(10),
```



```
c2_name VARCHAR(100),
c1_id BIGINT(10),
c1_name VARCHAR(100),
order_amount DOUBLE,
order_count BIGINT(10),
sku_count BIGINT(10),
dt varchar(100),
PRIMARY KEY (sku_id,dt)
);
```

向 MySQL 的 ADS 层目标装载数据：

```
-- -----
-- 使用 DDL 创建MySQL 中的ADS 层表
-- 指标： 1. 每天每个商品对应的订单个数
--        2. 每天每个商品对应的订单金额
--        3. 每天每个商品对应的数量
-- -----
CREATE TABLE ads_sku_index
(
    sku_id BIGINT,
    sku_name VARCHAR,
    weight DOUBLE,
    tm_id BIGINT,
    price DOUBLE,
    spu_id BIGINT,
    c3_id BIGINT,
    c3_name VARCHAR,
    c2_id BIGINT,
    c2_name VARCHAR,
    c1_id BIGINT,
    c1_name VARCHAR,
    order_amount DOUBLE,
    order_count BIGINT,
    sku_count BIGINT,
    dt varchar,
    PRIMARY KEY (sku_id,dt) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://kms-1:3306/ads',
    'table-name' = 'ads_sku_index',
    'driver' = 'com.mysql.jdbc.Driver',
    'username' = 'root',
    'password' = '123qwe'
);
```

```

|
|
|-----|
|-- dwd_paid_order_detail 已支付订单明细宽表
|-----|
|CREATE TABLE dwd_paid_order_detail
|
|  detail_id BIGINT,
|  order_id BIGINT,
|  user_id BIGINT,
|  province_id INT,
|  sku_id BIGINT,
|  sku_name STRING,
|  sku_num INT,
|  order_price DECIMAL(10,2),
|  create_time STRING,
|  pay_time STRING
|) WITH (
|  'connector' = 'kafka',
|  'topic' = 'dwd_paid_order_detail',
|  'scan.startup.mode' = 'earliest-offset',
|  'properties.bootstrap.servers' = 'kms-3:9092',
|  'format' = 'changelog-json'
|);
|
|-----|
|-- tmp_sku_index
|-- 商品指标统计
|-----|
|CREATE TABLE tmp_sku_index(
|  sku_id BIGINT,
|  order_count BIGINT,-- 订单数
|  order_amount DECIMAL(10,2), -- 订单金额
|  order_sku_num BIGINT,
|  pay_date DATE
|)WITH (
|  'connector' = 'kafka',
|  'topic' = 'tmp_sku_index',
|  'scan.startup.mode' = 'earliest-offset',
|  'properties.bootstrap.servers' = 'kms-3:9092',
|  'format' = 'changelog-json'
|);
|-----|
|-- tmp_sku_index
|-- 数据装载

```

```

-- -----
INSERT INTO tmp_sku_index
SELECT
    sku_id,
    count(distinct order_id) order_count, -- 订单数
    sum(order_price * sku_num) order_amount, -- 订单金额
    sum(sku_num) order_sku_num,
    TO_DATE(pay_time, 'yyyy-MM-dd') pay_date
FROM dwd_paid_order_detail
GROUP BY sku_id, TO_DATE(pay_time, 'yyyy-MM-dd')
;

-- -----
-- tmp_sku_index_source
-- 使用该临时汇总表，作为数据源
-- -----
CREATE TABLE tmp_sku_index_source(
    sku_id BIGINT,
    order_count BIGINT, -- 订单数
    order_amount DECIMAL(10,2), -- 订单金额
    order_sku_num BIGINT,
    pay_date DATE,
    proctime as PROCTIME() -- 通过计算列产生一个处理时间列
) WITH (
    'connector' = 'kafka',
    'topic' = 'tmp_sku_index',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'kms-3:9092',
    'format' = 'changelog-json'
);

-- -----
-- DIM 层, 商品维表,
-- 创建商品维表数据源
-- -----
DROP TABLE IF EXISTS `dim_sku_info`;
CREATE TABLE dim_sku_info (
    id BIGINT,
    sku_name STRING,
    c3_id BIGINT,
    weight DECIMAL(10,2),
    tm_id BIGINT,
    price DECIMAL(10,2),
    spu_id BIGINT,
    c3_name STRING,

```

```

    c2_id BIGINT,
    c2_name STRING,
    c1_id BIGINT,
    c1_name STRING,
    PRIMARY KEY (id) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://kms-1:3306/dim',
    'table-name' = 'dim_sku_info',
    'driver' = 'com.mysql.jdbc.Driver',
    'username' = 'root',
    'password' = '123qwe',
    'scan.fetch-size' = '100'
);

-- -----
-- 向ads_sku_index 装载数据
-- 维表JOIN
-- -----
INSERT INTO ads_sku_index
SELECT
    sku_id ,
    sku_name ,
    weight ,
    tm_id ,
    price ,
    spu_id ,
    c3_id ,
    c3_name ,
    c2_id ,
    c2_name ,
    c1_id ,
    c1_name ,
    sc.order_amount ,
    sc.order_count ,
    sc.order_sku_num ,
    cast(sc.pay_date as VARCHAR)
FROM
tmp_sku_index_source sc
JOIN dim_sku_info FOR SYSTEM_TIME AS OF sc.proctime as ds
ON ds.id = sc.sku_id;

```

当提交任务之后：观察 Flink WEB UI:



查看 ADS 层的 ads\_sku\_index 表数据:

```
Flink SQL> SELECT sku_name ,c3_name,order_amount,order_count ,sku_count ,dt FROM ads_sku_index;
```

+/-	sku_name	c3_name	order_amount	order_count	sku_count	dt
+	荣耀10青春版 幻影...	手机	4440.0	1	2	2020-06-18
+	TCL 55A950C 55英寸...	平板电视	13284.0	4	4	2020-06-18
+	小米Play 流光渐变...	手机	14420.0	3	10	2020-06-18
+	北院 精制 黄小米...	米面杂粮	1450.0	1	10	2020-06-18
+	荣耀10青春版 幻影...	手机	26401.0	3	17	2020-06-18
+	Apple iPhone XS M...	手机	89000.0	2	10	2020-06-18
+	荣耀10 GT游戏加速...	手机	14712.0	1	6	2020-06-18
+	小米 (MI) 小米路...	路由器	3996.0	3	18	2020-06-18

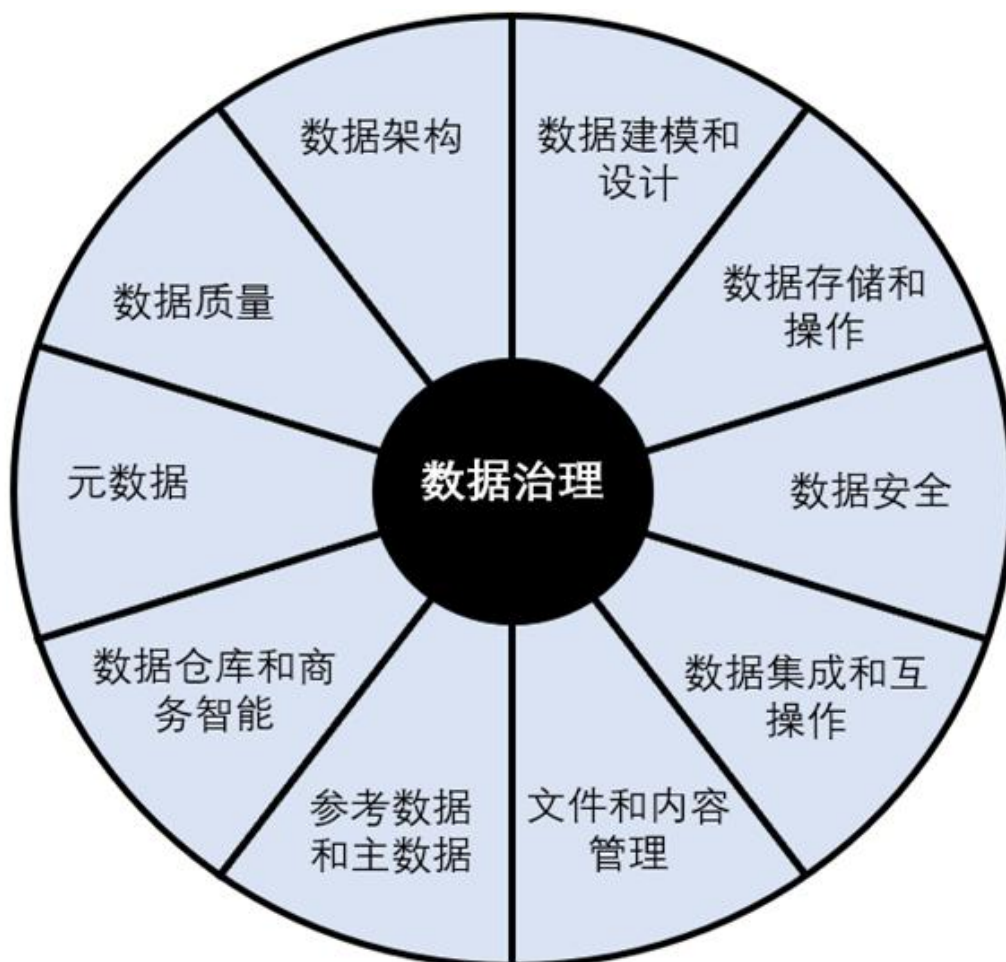
## 5. FineBI 展示



## 七、数据治理

数仓建设真正的难点不在于数仓设计，而在于后续业务发展起来，业务线变的庞大之后的数据治理，包括资产治理、数据质量监控、数据指标体系的建设等。

其实数据治理的范围很广，包含数据本身的管理、数据安全、数据质量、数据成本等。在 DAMA 数据管理知识体系指南中，数据治理位于数据管理“车轮图”的正中央，是数据架构、数据建模、数据存储、数据安全、数据质量、元数据管理、主数据管理等 10 大数据管理领域的总纲，为各项数据管理活动提供总体指导策略。



## 1. 数据治理之道是什么

### 1. 数据治理需要体系建设

为发挥数据价值需要满足三个要素：**合理的平台架构、完善的治理服务、体系化的运营手段。**

根据企业的规模、所属行业、数据量等情况选择合适的平台架构；治理服务需要贯穿数据全生命周期，保证数据在采集、加工、共享、存储、应用整个过程中的



完整性、准确性、一致性和实效性；运营手段则应当包括规范的优化、组织的优化、平台的优化以及流程的优化等等方面。

## 2. 数据治理需要夯实基础

数据治理需要循序渐进，但在建设初期至少需要关注三个方面：数据规范、数据质量、数据安全。规范化的模型管理是保障数据可以被治理的前提条件，高质量的数据是数据可用的前提条件，数据的安全管控是数据可以共享交换的前提条件。

## 3. 数据治理需要 IT 赋能

数据治理不是一堆规范文档的堆砌，而是需要将治理过程中所产生的的规范、流程、标准落地到 IT 平台上，在数据生产过程中通过“以终为始”前向的方式进行数据治理，避免事后稽核带来各种被动和运维成本的增加。

## 4. 数据治理需要聚焦数据

数据治理的本质是管理数据，因此需要加强元数据管理和主数据管理，从源头治理数据，补齐数据的相关属性和信息，比如：元数据、质量、安全、业务逻辑、血缘等，通过元数据驱动的方式管理数据生产、加工和使用。

## 5. 数据治理需要建管一体化

数据模型血缘与任务调度的一致性是建管一体化的关键，有助于解决数据管理与数据生产口径不一致的问题，避免出现两张皮的低效管理模式。

## 2. 浅谈数据治理方式

如上面所说，数据治理的范围非常广，其中最重要的是数据质量治理，而数据质量涉及的范围也很广，贯穿数仓的整个生命周期，从数据产生->数据接入->数据存储->数据处理->数据输出->数据展示，每个阶段都需要质量治理，评价维度包括完整性、规范性、一致性、准确性、唯一性、关联性等。

在系统建设的各个阶段都应该根据标准进行数据质量检测和规范，及时进行治疗，避免事后的清洗工作。

质量检测可参考以下维度：

维度	衡量标准
----	------

维度	衡量标准
完整性	业务指定必须的数据是否缺失，不允许为空字符或者空值等。例如，数据源是否完整、维度取值是否完整、数据取值是否完整等
时效性	当需要使用时，数据能否反映当前事实。即数据必须及时，能够满足系统对数据时间的要求。例如处理（获取、整理、清洗、加载等）的及时性
唯一性	在指定的数据集中数据值是否唯一
参照完整性	数据项是否在父表中有定义
依赖一致性	数据项取值是否满足与其他数据项之间的依赖关系
正确性	数据内容和定义是否一致
精确性	数据精度是否达到业务规则要求的位数
技术有效性	数据项是否按已定义的格式标准组织
业务有效性	数据项是否符合已定义的
可信度	根据客户调查或客户主动提供获得
可用性	数据可用的时间和数据需要被访问时间的比例
可访问性	数据是否便于自动化读取

下面是根据美团的技术文章总结的几点具体治理方式：

## 1. 规范治理

规范是数仓建设的保障。为了避免出现指标重复建设和数据质量差的情况，统一按照最详细、可落地的方法进行规范建设。

### (1) 词根

词根是维度和指标管理的基础，划分为普通词根与专有词根，提高词根的易用性和关联性。

- 普通词根：描述事物的最小单元体，如：交易-trade。

- 专有词根：具备约定成俗或行业专属的描述体，如：美元-USD。

## (2) 表命名规范

### 通用规范

- 表名、字段名采用一个下划线分隔词根(示例:clienttype->client\_type)。
- 每部分使用小写英文单词,属于通用字段的必须满足通用字段信息的定义。
- 表名、字段名需以字母为开头。
- 表名、字段名最长不超过 64 个英文字符。
- 优先使用词根中已有关键字(数仓标准配置中的词根管理),定期 Review 新增命名的不合理性。
- 在表名自定义部分禁止采用非标准的缩写。

### 表命名规则

- **表名称** = 类型 + 业务主题 + 子主题 + 表含义 + 存储格式 + 更新频率 + 结尾, 如下图所示:

存储层-ods	业务库表名			全量-不加 快照-Snapshot-ss 增量-Increment-inc 拉链、缓慢变化维- SlowlyChangingDimen- sions-scd	按小时更新-hourly 按天更新-daily 按周更新-weekly 按月更新-monthly 每年-yearly	[his]			
明细层-dwd	业务主题-m	二级主题简称	自定义表名 [(detail ext)]						
主题宽表层-dwt			维度名[(detail ext)]						
轻度汇总层-dwa									
应用层-app									
维度表-dim	业务表名					序号:01-100			
临时表-temp	业务表名					view			
视图	业务表名					extnl			
外部表	原表名								

例如: dwt\_m\_ord\_order\_ss\_daily

[]: 可选项, 可以省略  
0: 多选项, 以 | 分隔, 必须选填一项

### 统一的表命名规范

## (3) 指标命名规范

结合指标的特性以及词根管理规范, 将指标进行结构化处理。

1. 基础指标词根, 即所有指标必须包含以下基础词根:

基础指标词根	英文全称	Hive数据类型	MySQL数据类型	长度	精度	词根	样例
数量	count	Bigint	Bigint	10	0	cnt	
金额类	amout	Decimal	Decimal	20	4	amt	
比率/占比	ratio	Decimal	Decimal	10	4	ratio	0.9818
.....	.....	.....	.....			.....	

2. 业务修饰词, 用于描述业务场景的词汇, 例如 trade-交易。

3. 日期修饰词, 用于修饰业务发生的时间区间。

日期类型	全称	词根	备注
日	daily	d	
周	weekly	w	
.....	.....	.....	

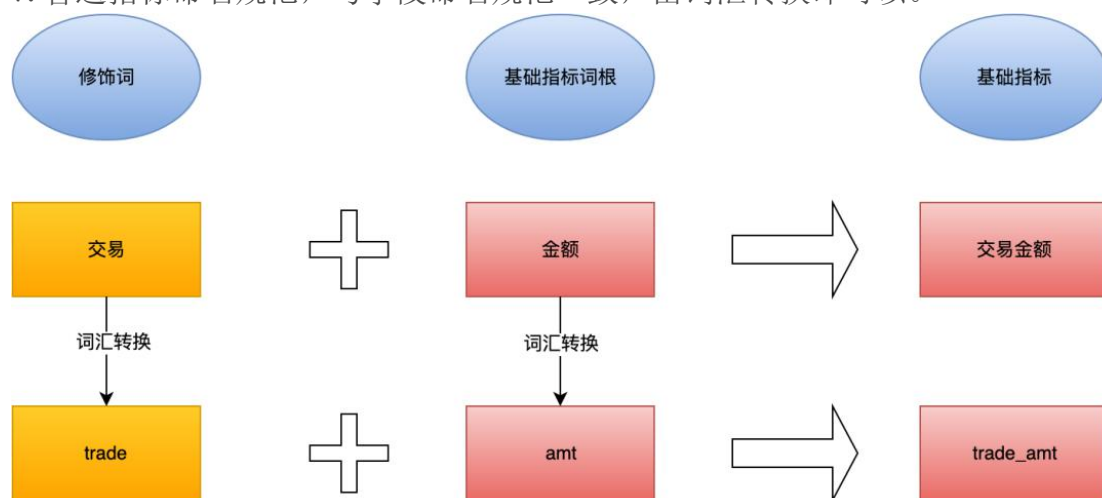
4. 聚合修饰词，对结果进行聚集操作。

聚合类型	全称	词根	备注
平均	average	avg	
周累计	wtd	wtd	本周一截止到当天累计
.....	.....	.....	

5. 基础指标，单一的业务修饰词+基础指标词根构建基础指标，例如：交易金额-trade\_amt。

6. 派生指标，多修饰词+基础指标词根构建派生指标。派生指标继承基础指标的特性，例如：安装门店数量-install\_poi\_cnt。

7. 普通指标命名规范，与字段命名规范一致，由词汇转换即可以。



## 2. 架构治理

### (1) 数据分层

优秀可靠的数仓体系，往往需要清晰的数据分层结构，即要保证数据层的稳定又要屏蔽对下游的影响，并且要避免链路过长，一般的分层架构如下：



## (2) 数据流向

稳定业务按照标准的数据流向进行开发，即 ODS→DWD→DWA→APP。非稳定业务或探索性需求，可以遵循 ODS→DWD→APP 或者 ODS→DWD→DWT→APP 两个模型数据流。在保障了数据链路的合理性之后，又在此基础上确认了模型分层引用原则：

- 正常流向：ODS→DWD→DWT→DWA→APP，当出现 ODS→DWD→DWA→APP 这种关系时，说明主题域未覆盖全。应将 DWD 数据落到 DWT 中，对于使用频度非常低的表允许 DWD→DWA。
- 尽量避免出现 DWA 宽表中使用了 DWD 又使用了（该 DWD 所归属主题域）DWT 的表。
- 同一主题域内对于 DWT 生成 DWT 的表，原则上要尽量避免，否则会影响 ETL 的效率。
- DWT、DWA 和 APP 中禁止直接使用 ODS 的表，ODS 的表只能被 DWD 引用。
- 禁止出现反向依赖，例如 DWT 的表依赖 DWA 的表。

## 3. 元数据治理

元数据可分为技术元数据和业务元数据：

**技术元数据**为开发和管理数据仓库的 IT 人员使用，它描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。

常见的技术元数据有：

- 存储元数据：如表、字段、分区等信息。

- 运行元数据：如大数据平台上所有作业运行等信息：类似于 Hive Job 日志，包括作业类型、实例名称、输入输出、SQL、运行参数、执行时间，执行引擎等。
- 数据开发平台中数据同步、计算任务、任务调度等信息：包括数据同步的输入输出表和字段，以及同步任务本身的节点信息：计算任务主要有输入输出、任务本身的节点信息 任务调度主要有任务的依赖类型、依赖关系等，以及不同类型调度任务的运行日志等。
- 数据质量和运维相关元数据：如任务监控、运维报警、数据质量、故障等信息，包括任务监控运行日志、告警配置及运行日志、故障信息等。

**业务元数据**为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

- 常见的业务元数据有维度及属性(包括维度编码，字段类型，创建人，创建时间，状态等)、业务过程、指标(包含指标名称, 指标编码，业务口径，指标类型，责任人，创建时间，状态，sql 等)，安全等级，计算逻辑等的规范化定义，用于更好地管理和使用数据。数据应用元数据，如数据报表、数据产品等的配置和运行元数据。

元数据不仅定义了数据仓库中数据的模式、来源、抽取和转换规则等，而且是整个数据仓库系统运行的基础，**元数据把数据仓库系统中各个松散的组件联系起来，组成了一个有机的整体。**

**元数据治理主要解决三个问题：**

1. 通过建立相应的组织、流程和工具，推动业务标准的落地实施，实现指标的规范定义，消除指标认知的歧义；
2. 基于业务现状和未来的演进方式，对业务模型进行抽象，制定清晰的主题、业务过程和分析方向，构建完备的技术元数据，对物理模型进行准确完善的描述，并打通技术元数据与业务元数据的关系，对物理模型进行完备的刻画；
3. 通过元数据建设，为使用数据提效，解决“找数、理解数、评估”难题以及“取数、数据可视化”等难题。

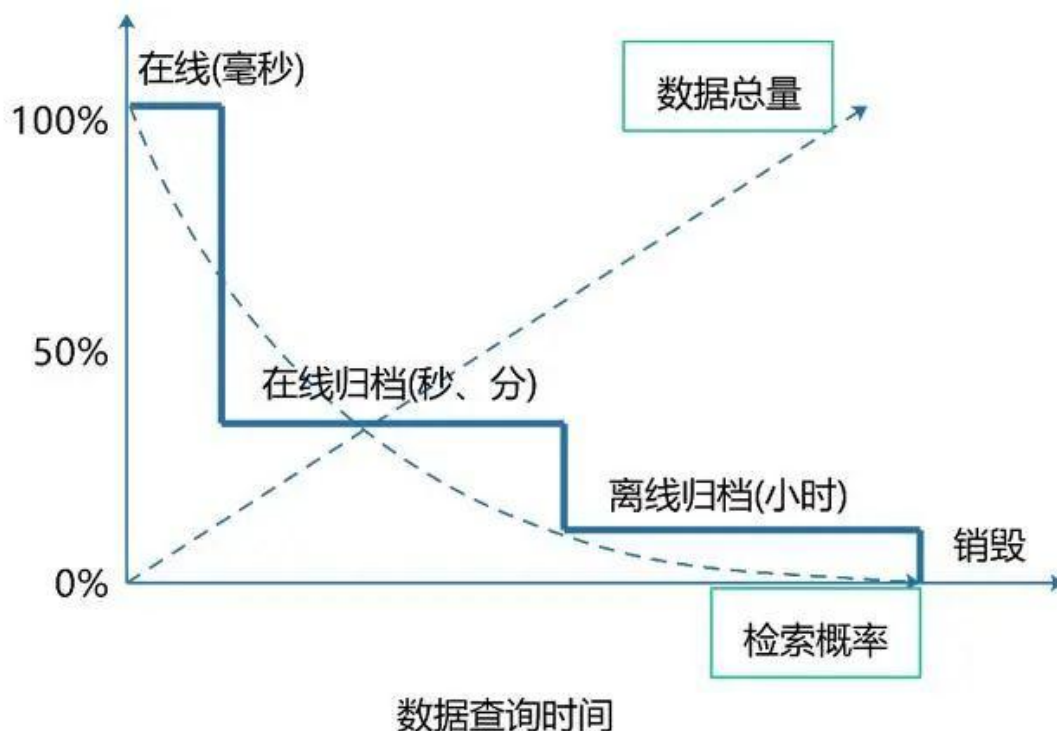
#### 4. 安全治理



围绕数据安全标准，首先要有数据的分级、分类标准，确保数据在上线前有着准确的密级。第二，针对数据使用方，要有明确的角色授权标准，通过分级分类和角色授权，来保障重要数据拿不走。第三，针对敏感数据，要有隐私管理标准，保障敏感数据的安全存储，即使未授权用户绕过权限管理拿到敏感数据，也要确保其看不懂。第四，通过制定审计标准，为后续的审计提供审计依据，确保数据走不脱。

## 5. 数据生命周期治理

任何事物都具有一定的生命周期，数据也不例外。从数据的产生、加工、使用乃至消亡都应该有一个科学的管理办法，将极少或者不再使用的数据从系统中剥离出来，并通过核实的存储设备进行保留，不仅能够提高系统的运行效率，更好的服务客户，还能大幅度减少因为数据长期保存带来的储存成本。数据生命周期一般包含在线阶段、归档阶段（有时还会进一步划分为在线归档阶段和离线归档阶段）、销毁阶段三大阶段，管理内容包括建立合理的数据类别，针对不同类别的数据制定各个阶段的保留时间、存储介质、清理规则和方式、注意事项等。



从上图数据生命周期中各参数间的关系中我们可以了解到，数据生命周期管理可以使得高价值数据的查询效率大幅提升，而且高价格的存储介质的采购量也可以

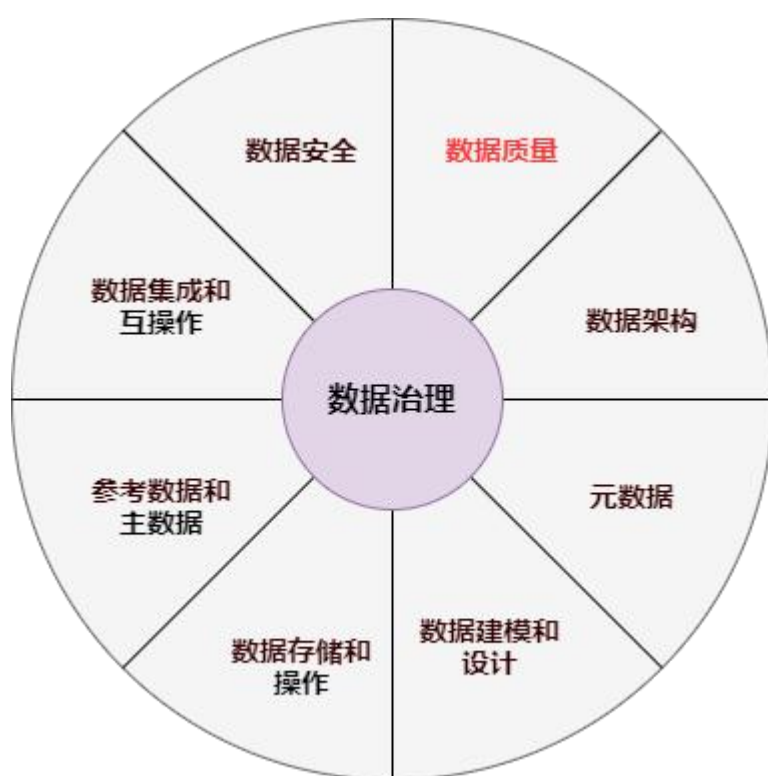
减少很多；但是随着数据的使用程度的下降，数据被逐渐归档，查询时间也慢慢的变长；最后随着数据的使用频率和价值基本没有了之后，就可以逐渐销毁了。

猜你喜欢：

1. 美团数据平台及数仓建设实践，超十万字总结
2. 上百本优质大数据书籍，附必读清单(大数据宝藏)
3. 五万字 | 耗时一个月整理出这份 Hadoop 吐血宝典

## 八、数据质量建设

数据治理的范围非常广，包含数据本身的管理、数据安全、数据质量、数据成本等。在这么多治理内容中，大家想下最重要的治理是什么？当然是**数据质量治理**，因为数据质量是数据分析结论有效性和准确性的基础，也是这一切的前提。所以如何保障数据质量，确保数据可用性是数据仓库建设中不容忽视的环节。



数据质量涉及的范围也很广，贯穿数仓的整个生命周期，从**数据产生->数据接入->数据存储->数据处理->数据输出->数据展示**，每个阶段都需要质量治理。在系统建设的各个阶段都应该根据标准进行数据质量检测和规范，及时进行治疗，避免事后的清洗工作。

本文档首发于公众号【五分钟学大数据】，完整的数据治理及数仓建设文章公众号上都有！

## 1. 为什么要进行数据质量评估

很多刚入门的数据人，拿到数据后会立刻开始对数据进行各种探查、统计分析等，企图能立即发现数据背后隐藏的信息和知识。然而忙活了一阵才颓然发现，并不能提炼出太多有价值的信息，白白浪费了大量的时间和精力。比如和数据打交道的过程中，可能会出现以下的场景：

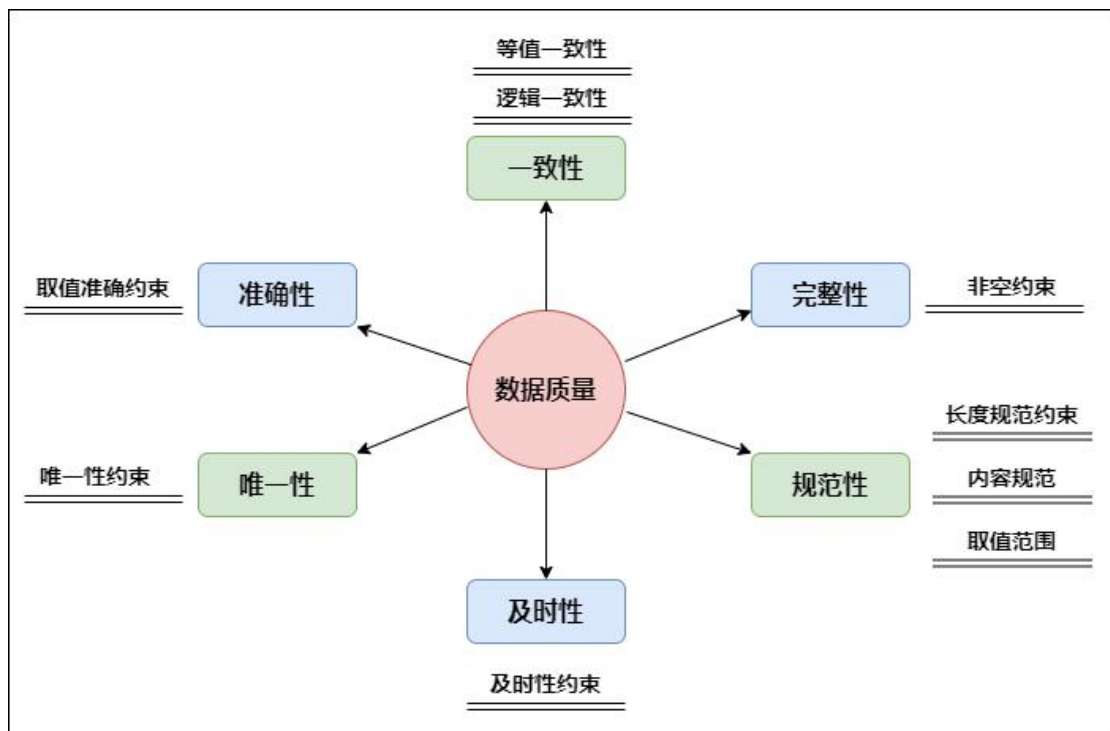
**场景一：**作为数据分析人员，要统计一下近 7 天用户的购买情况，结果从数仓中统计完发现，很多数据发生了重复记录，甚至有些数据统计单位不统一。

**场景二：**业务看报表，发现某一天的成交 gmV 暴跌，经过排查发现，是当天的数据缺失。

造成这一情况的一个重要因素就是忽视了对数据质量的客观评估，没有制定合理的衡量标准，导致没有发现数据已出现问题。所以，进行科学、客观的数据质量衡量标准是非常必要且十分重要的。

## 2. 数据质量衡量标准

如何评估数据质量的好坏，业界有不同的标准，我总结了以下六个维度进行评估，包括**完整性**、**规范性**、**一致性**、**准确性**、**唯一性**、**及时性**。



## 1. 数据完整性

完整性指的是数据信息**是否存在缺失**的状况，数据缺失的情况可能是整个数据记录缺失，也可能是数据中某个字段信息的记录缺失。

## 2. 数据规范性

规范性指的是描述数据**遵循预定的语法规则**的程度，是否符合其定义，比如数据的类型、格式、取值范围等。

## 3. 数据一致性

一致性是指数据是否遵循了**统一的规范**，数据集合是否保持了统一的格式。数据质量的一致性主要体现在数据记录的规范和数据是否符合逻辑，一致性并不意味着数值上的绝对相同，而是数据收集、处理的方法和标准的一致。常见的一致性指标有：ID 重合度、属性一致、取值一致、采集方法一致、转化步骤一致。

## 4. 数据准确性

准确性是指数据记录的信息**是否存在异常或错误**。和一致性不一样，存在准确性问题的数据不仅仅只是规则上的不一致，更为常见的数据准确性错误就如乱码，其次异常的大或者小的数据也是不符合条件的数据。常见的准确性指标有：缺失值占比、错误值占比、异常值占比、抽样偏差、数据噪声。

## 5. 数据唯一性

唯一性指的是数据库的**数据不存在重复**的情形。比如真实成交 1 万条，但数据表有 3000 条重复了，成了 1.3 万条成交记录，这种数据不符合数据唯一性。

## 6. 数据及时性

及时性是指数据从产生到可以查看的时间间隔，也叫数据的延时时长。比如一份数据是统计离线今日的，结果都是第二天甚至第三天才能统计完，这种数据不符合数据及时性。

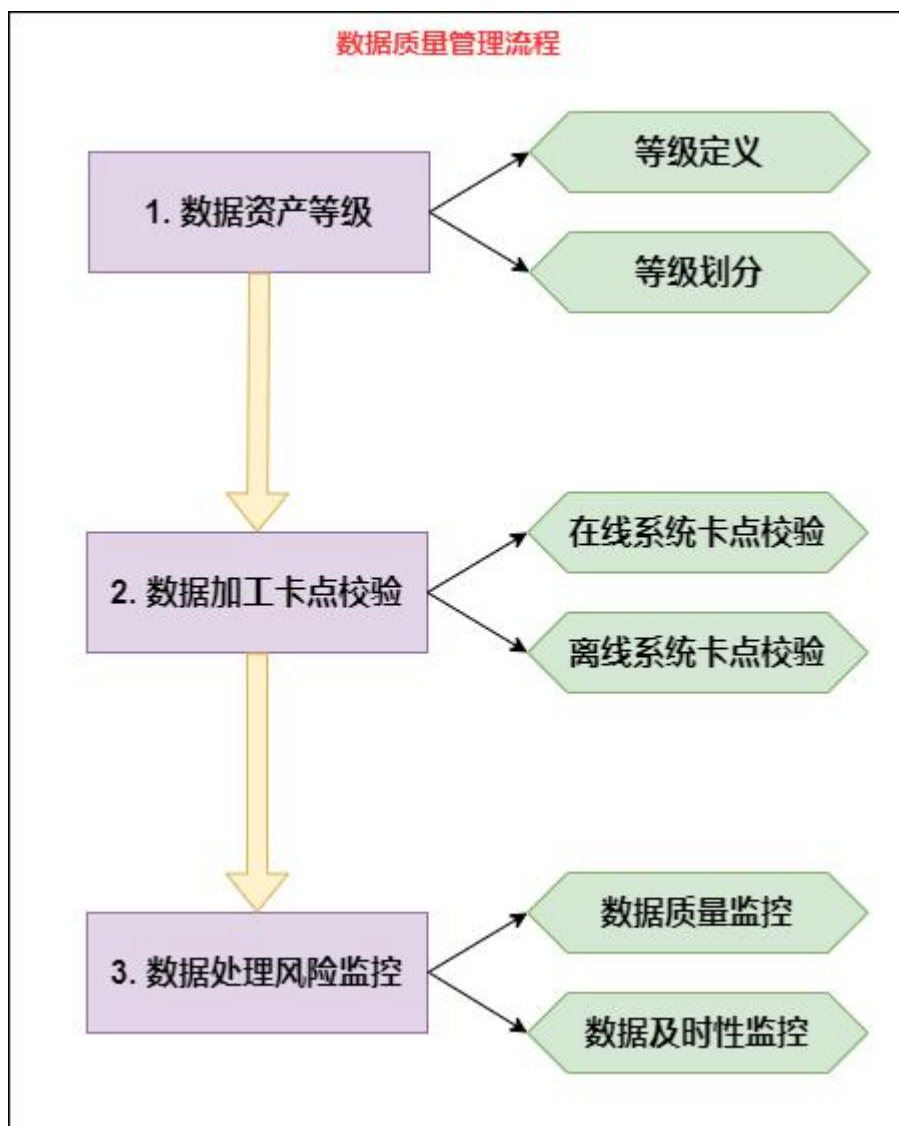
还有一些其他的衡量标准，在此简单列出：

维度	衡量标准
参照完整性	数据项是否在父表中有定义

维度	衡量标准
依赖一致性	数据项取值是否满足与其他数据项之间的依赖关系
正确性	数据内容和定义是否一致
精确性	数据精度是否达到业务规则要求的位数
技术有效性	数据项是否按已定义的格式标准组织
业务有效性	数据项是否符合已定义的
可信度	根据客户调查或客户主动提供获得
可用性	数据可用的时间和数据需要被访问时间的比例
可访问性	数据是否便于自动化读取

### 3. 数据质量管理流程

本节流程如下图所示：



## 1. 数据资产等级

### 1) 等级定义

根据当数据质量不满足完整性、规范性、一致性、准确性、唯一性、及时性时，对业务的影响程度大小来划分数据的资产等级。

- 毁灭性**：数据一旦出错，会引起巨大的资产损失，面临重大收益受损等。标记为 L1
- 全局性**：数据用于集团业务、企业级效果评估和重要决策任务等。标记为 L2

3. **局部性**：数据用于某个业务线的日常运营、分析报告等，如果出现问题会给该业务线造成一定的影响或影响其工作效率。标记为 L3
4. **一般性**：数据用于日常数据分析，出现问题的带来的影响很小。标记为 L4
5. **未知性质**：无法追溯数据的应用场景。标记为 Lx

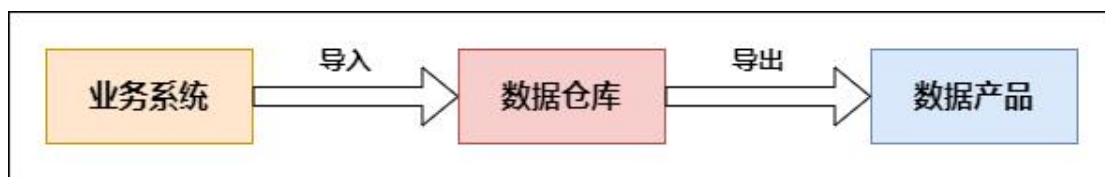
**重要程度**：L1>L2>L3>L4>Lx。如果一份数据出现在多个应用场景中，则根据其最重要程度进行标记。

## 2) 等级划分

定义数据资产等级后，我们可以从数据流程链路开始进行数据资产等级标记，完成数据资产等级确认，给不同的数据定义不同的重要程度。

### 1. 分析数据链路：

数据是从业务系统中产生的，经过同步工具进入数据仓库系统中，在数据仓库中进行一般意义上的清洗、加工、整合、算法、模型等一系列运算后，再通过同步工具输出到数据产品中进行消费。而从业务系统到数据仓库再到数据产品都是以表的形式体现的，其流转过程如下图所示：



### 2. 标记数据资产等级：

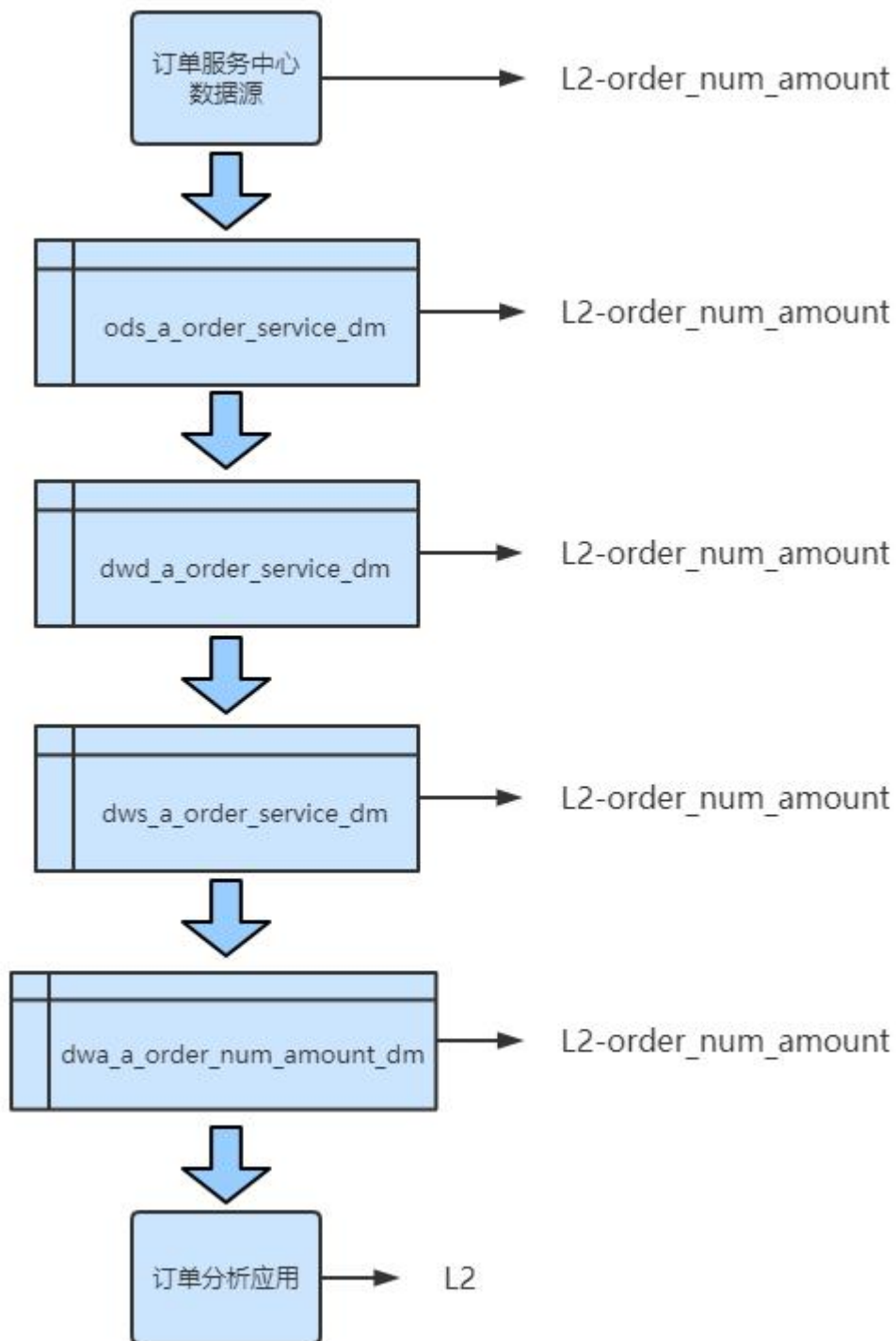
在所有数据链路上，整理出消费各个表的应用业务。通过给这些应用业务划分数据资产等级，结合数据的上下游依赖关系，将整个链路打上某一类资产等级标签。

#### 举例：

假设公司有统一的订单服务中心。应用层的应用业务是按照业务线，商品类型和地域统计公司的订单数量和订单金额，命名为 `order_num_amount`。

假设该应用会影响到整个企业的重要业务决策，我们可以把应用定级为 L2，从而整个数据链路上的表的数据等级，都可以标记为 `L2-order_num_amount`，一直标记到源数据业务系统，如下图所示：





## 2. 数据加工过程卡点校验

### 1) 在线系统数据校验

在线业务复杂多变，总是在不断地变更，每一次变更都会带来数据的变化，数据仓库需要适应这多变的业务发展，及时做到数据的准确性。

基于此，在线业务的变更如何高效地通知到离线数据仓库，同样也是需要考虑的问题。为了保障在线数据和离线数据的一致性，我们可以通过**工具+人员管理并行的方式**来尽可能的解决以上问题：既要在工具上自动捕捉每一次业务的变化，同时也要求开发人员在意识上自动进行业务变更通知。

### 1. 业务上线发布平台：

监控业务上线发布平台上的重大业务变更，通过订阅这个发布过程，及时将变更内容通知到数据部门。

由于业务系统复杂多变，若日常发布变更频繁，那么每次都通知数据部门，会造成不必要的资源浪费。这时，我们可以使用**之前已经完成标记的数据资产等级标签**，针对涉及高等级数据应用的数据资产，整理出哪些类型的业务变更会影响数据的加工或者影响数据统计口径的调整，则这些情况都必须及时通知到数据部门。如果公司没有自己的业务发布平台，那么就需要与业务部门约定好，**针对高等级的数据资产的业务变更，需要以邮件或者其他书面的说明及时反馈到数据部门。**

### 2. 操作人员管理：

工具只是辅助监管的一种手段，而使用工具的人员才是核心。数据资产等级的上下游打通过程需要通知给在线业务系统开发人员，使其知道哪些是重要的核心数据资产，哪些暂时还只是作为内部分析数据使用，提高在线开发人员的数据风险意识。

可以通过培训的方式，**把数据质量管理的诉求，数据质量管理的整个数据加工过程，以及数据产品的应用方式及应用场景告知在线开发人员，使其了解数据的重要性、价值及风险。**确保在线开发人员在完成业务目标的同时，也要考虑数据的目标，保持业务端和数据段一致。

## 2) 离线系统数据校验

数据从在线业务系统到数据仓库再到数据产品的过程中，需要在数据仓库这一层完成数据的清洗、加工。正是有了数据的加工，才有了数据仓库模型和数据仓库代码的建设。如何保障数据加过程中的质量，是离线数据仓库保障数据质量的一个重要环节。

在这些环节中，我们可以采用以下方式保障数据质量：

### 1. 代码提交核查：

开发相关的规则引擎，辅助代码提交校验。规则分类大致为：

- **代码规范类规则**：如表命名规范、字段命名规范、生命周期设置、表注释等；
- **代码质量类规则**：如分母为 0 提醒、NULL 值参与计算提醒等；
- **代码性能类规则**：如大表提醒、重复计算监测、大小表 join 操作提醒等。

## 2. 代码发布核查：

加强测试环节，测试环境测试后再发布到生成环境，且生成环境测试通过后才算发布成功。

## 3. 任务变更或重跑数据：

在进行数据更新操作前，需要通知下游数据变更原因、变更逻辑、变更时间等信息。下游没有异议后，再按照约定时间执行变更发布操作。

## 3. 数据处理风险监控

风险点监控主要是针对数据在日常运行过程中容易出现的风险进行监控并设置报警机制，主要包括**在线数据**和**离线数据**运行风险点监控。

### 1) 数据质量监控

**在线业务系统**的数据生产过程需要保证数据质量，主要根据业务规则对数据进行监控。

比如交易系统配置的一些监控规则，如订单拍下时间、订单完结时间、订单支付金额、订单状态流转等都配置了校验规则。订单拍下时间肯定不会大于当天时间，也不会小于业务上线时间，一旦出现异常的订单创建时间，就会立刻报警，同时报警给到多人。通过这种机制，可以及时发现并解决问题。

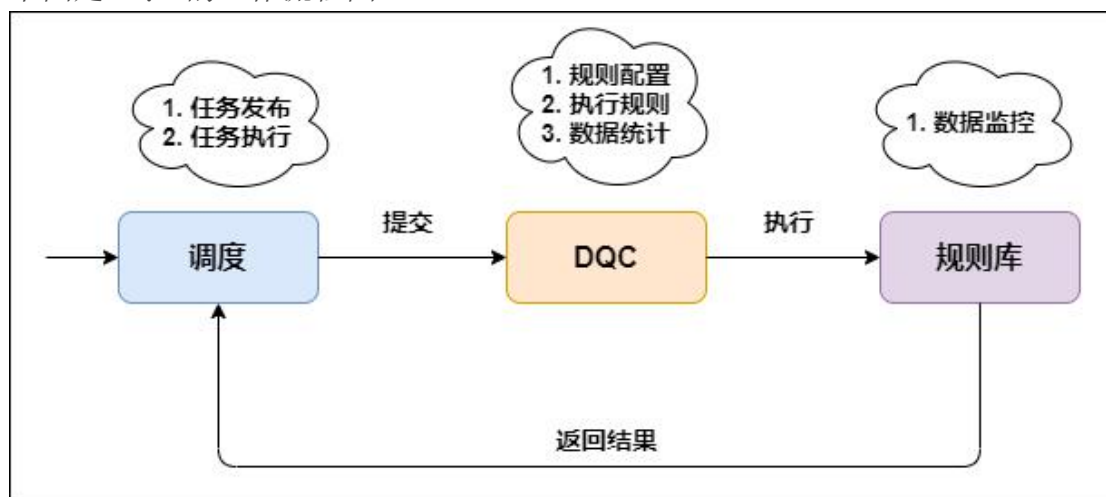
随着业务负责程度的提升，会导致规则繁多、规则配置的运行成本增大，这时可以**按照我们之前的数据资产等级有针对性的进行监控**。

**离线数据**风险点监控主要包括对数据准确性和数据产出及时性的监控。对数据调度平台上所有数据处理调度进行监控。

我们以阿里的 DataWorks 数据调度工具为例，DataWorks 是基于 MaxCompute 计算引擎的一站式开发工场，帮助企业快速完成数据集成、开发、治理、质量、安全等全套数据研发工作。

DataWorks 中的 DQC 通过配置数据质量校验规则，实现离线数据处理中的数据质量监控报警机制。

下图是 DQC 的工作流程图：



DQC 数据监控规则有强规则和弱规则：

- 强规则：一旦触发报警就会阻断任务的执行（将任务置为失败状态，使下游任务不会被触发执行）。
- 弱规则：只报警但不阻断任务的执行。

DQC 提供常用的规则模板，包括表行数较 N 天前波动率、表空间大小较 N 天前波动率、字段最大/最小/平均值相比 N 天前波动率、字段空值/唯一数等。

DQC 检查其实也是运行 SQL 任务，只是这个任务是嵌套在主任务中的，一旦检查点太多自然就会影响整体的性能，因此还是依赖数据产等级来确定规则的配置情况。比如 L1、L2 类数据监控率要达到 90% 以上，规则类型需要三种及以上，而不重要的数据资产则不强制要求。

## 2) 数据及时性监控

在确保数据准确性的前提下，需要进一步让数据能够及时地提供服务，否则数据的价值将大幅度降低，甚至没有价值，所以确保数据及时性也是保障数据质量重中之重的一环。

### 1. 任务优先级：

对于 DataWorks 平台的调度任务，可以通过智能监控工具进行优先级设置。DataWorks 的调度是一个树形结构，当配置了叶子节点的优先级，这个优先级会传递到所有的上游节点，而叶子节点通常就是服务业务的消费节点。因此，在优先级的设置上，要先确定业务的资产等级，**等级越高的业务对应的消费节点优先级越高**，优先调度并占用计算资源，确保高等级业务的准时产出。总之，就是按照数据资产等级优先执行高等级数据资产的调度任务，优先保障高等级业务的数据需求。

## 2. 任务报警：

任务报警和优先级类似，通过 DataWorks 的智能监控工具进行配置，只需要配置叶子节点即可向上游传递报警配置。任务执行过程中，可能出错或延迟，为了保证最重要数据（即资产等级高的数据）产出，需要立即处理出错并介入处理延迟。

## 3. DataWorks 智能监控：

DataWorks 进行离线任务调度时，提供智能监控工具，对调度任务进行监报告警。根据监控规则和任务运行情况，智能监控决策是否报警、何时报警、如何报警以及给谁报警。智能监控会自动选择最合理的报警时间、报警方式以及报警对象。

## 4. 最后

要想真正解决数据质量问题，就要**明确业务需求并从需求开始控制数据质量，并建立数据质量管理机制**。从业务出发做问题定义，由工具自动、及时发现问题，明确问题责任人，通过邮件、短信等方式进行通知，保证问题及时通知到责任人。跟踪问题整改进度，保证数据质量问题全过程的管理。

# 九、数仓规范建设指南

## 1. 数仓公共开发规范

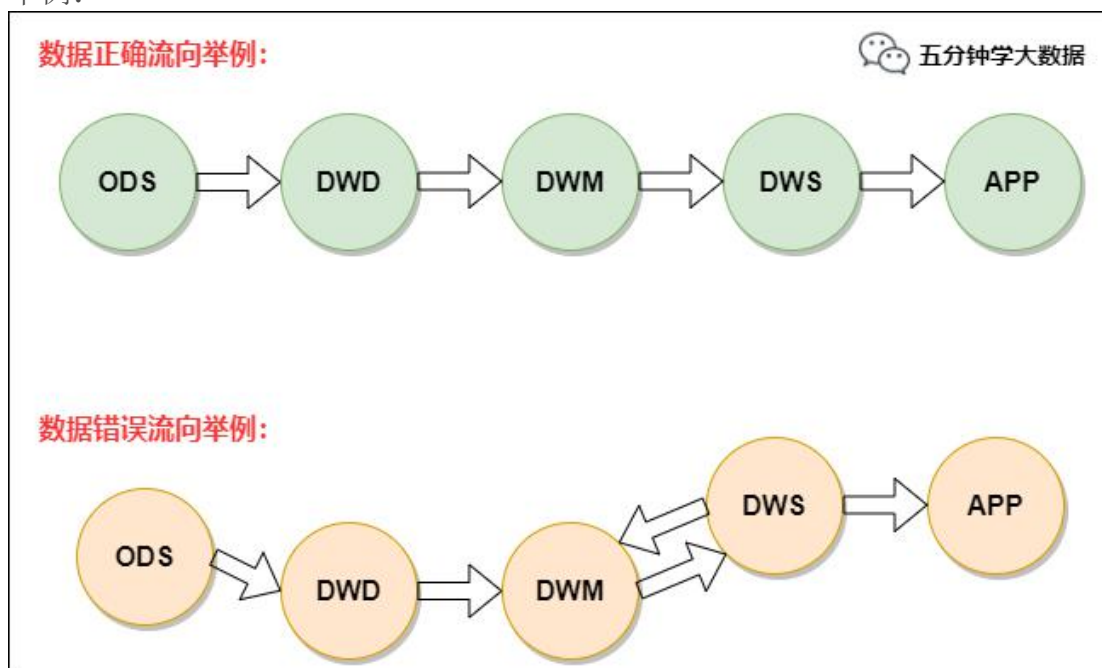
### 1. 层次调用规范

**稳定业务**按照标准的数据流向进行开发，即 ODS -> DWD -> DWS -> APP。**非稳定业务**或探索性需求，可以遵循 ODS -> DWD -> APP 或者 ODS -> DWD -> DWM -> APP 两个模型数据流。

在保障了数据链路的合理性之后，也必须保证模型分层引用原则：

- 正常流向：ODS -> DWD -> DWM -> DWS -> APP，当出现 ODS -> DWD -> DWS -> APP 这种关系时，说明主题域未覆盖全。应将 DWD 数据落到 DWM 中，对于使用频度非常低的表允许 DWD -> DWS。
- 尽量避免出现 DWS 宽表中使用 DWD 又使用（该 DWD 所归属主题域）DWM 的表。
- 同一主题域内对于 DWM 生成 DWM 的表，原则上要尽量避免，否则会影响 ETL 的效率。
- DWM、DWS 和 APP 中禁止直接使用 ODS 的表，ODS 的表只能被 DWD 引用。
- **禁止出现反向依赖**，例如 DWM 的表依赖 DWS 的表。

举例：



## 2. 数据类型规范

需统一规定不同的数据的数据类型，严格按照规定的数据类型执行：

1. **金额**：double 或使用 decimal(11,2) 控制精度等，**明确单位是分还是元**。
2. **字符串**：string。
3. **id 类**：bigint。
4. **时间**：string。
5. **状态**：string

### 3. 数据冗余规范

宽表的冗余字段要确保：

1. 冗余字段要使用高频，下游 3 个或以上使用。
2. 冗余字段引入不应造成本身数据产生过多的延后。
3. 冗余字段和已有字段的重复率不应过大，原则上不应超过 60%，如需要可以选择 join 或原表拓展。

### 4. NULL 字段处理规范

- 对于维度字段，需设置为-1
- 对于指标字段，需设置为 0

### 5. 指标口径规范

保证主题域内，指标口径一致，无歧义。

通过数据分层，提供统一的数据出口，统一对外输出的数据口径，避免同一指标不同口径的情况发生。

#### 1) 指标梳理

指标口径的不一致使得数据使用的成本极高，经常出现口径打架、反复核对数据的问题。在数据治理中，我们将需求梳理到的所有指标进行进一步梳理，明确其口径，如果存在两个指标名称相同，但口径不一致，先判断是否是进行合并，如需要同时存在，那么在命名上必须能够区分开。

#### 2) 指标管理

指标管理分为原子指标维护和派生指标维护。

原子指标：

- 选择原子指标的归属产线、业务板块、数据域、业务过程
- 选择原子指标的统计数据来源于该业务过程下的原始数据源
- 录入原子指标的英文名称、中文名称、概述
- 填写指标函数



- 系统根据指标函数自动生成原子指标的定义表达式
- 系统根据指标定义表达式以及数据源表生成原子指标 SQL

派生指标：

- 在原子指标的基础之上选择了一些维度或者修饰限定词。

## 6. 数据表处理规范

### 1) 增量表

新增数据，增量数据是上次导出之后的新数据。

1. 记录每次增加的量，而不是总量；
2. 增量表，只报变化量，无变化不用报；
3. 每天一个分区。

### 2) 全量表

每天的所有的最新状态的数据。

1. 全量表，有无变化，都要报；
2. 每次上报的数据都是所有的数据（变化的 + 没有变化的）；
3. 只有一个分区。

### 3) 快照表

按日分区，记录截止数据日期的全量数据。

1. 快照表，有无变化，都要报；
2. 每次上报的数据都是所有的数据（变化的 + 没有变化的）；
3. 一天一个分区。

### 4) 拉链表

记录截止数据日期的全量数据。

1. 记录一个事物从开始，一直到当前状态的所有变化的信息；
2. 拉链表每次上报的都是历史记录的最终状态，是记录在当前时刻的历史总量；

3. 当前记录存的是当前时间之前的所有历史记录的最后变化量（总量）；
4. 只有一个分区。

## 7. 表的生命周期管理

这部分主要是要通过对历史数据的等级划分与对表类型的划分生成相应的生命周期管理矩阵。

### 1) 历史数据等级划分

主要将历史数据划分 P0、P1、P2、P3 四个等级，其具体定义如下：

- **P0**：非常重要的主题域数据和非常重要的应用数据，具有不可恢复性，如交易、日志、集团 KPI 数据、IPO 关联表。
- **P1**：重要的业务数据和重要的应用数据，具有不可恢复性，如重要的业务产品数据。
- **P2**：重要的业务数据和重要的应用数据，具有可恢复性，如交易线 ETL 产生的中间过程数据。
- **P3**：不重要的业务数据和不重要的应用数据，具有可恢复性，如某些 SNS 产品报表。

### 2) 表类型划分

#### 1. 事件型流水表（增量表）

事件型流水表（增量表）指数据无重复或者无主键数据，如日志。

#### 2. 事件型镜像表（增量表）

事件型镜像表（增量表）指业务过程性数据，有主键，但是对于同样主键的属性会发生缓慢变化，如交易、订单状态与时间会根据业务发生变更。

#### 3. 维表

维表包括维度与维度属性数据，如用户表、商品表。

#### 4. Merge 全量表

Merge 全量表包括业务过程性数据或者维表数据。由于数据本身有新增的或者发生状态变更，对于同样主键的数据可能会保留多份，因此可以对这些数据根据主键进行 Merge 操作，主键对应的属性只会保留最新状态，历史状态保留在前一天分区中。例如，用户表、交易表等都可以进行 Merge 操作。

## 5. ETL 临时表

ETL 临时表是指 ETL 处理过程中产生的临时表数据，一般不建议保留，最多 7 天。

## 6. TT 临时数据

TT 拉取的数据和 DbSync 产生的临时数据最终会流转到 DS 层，ODS 层数据作为原始数据保留下来，从而使得 TT&DbSync 上游数据成为临时数据。这类数据不建议保留很长时间，生命周期默认设置为 93 天，可以根据实际情况适当减少保留天数。

## 7. 普通全量表

很多小业务数据或者产品数据，BI 一般是直接全量拉取，这种方式效率高，对存储压力也不是很大，而且表保留很长时间，可以根据历史数据等级确定保留策略。

通过上述历史数据等级划分与表类型划分，生成相应的生命周期管理矩阵，如下表所示：

公众号：五分钟学大数据		P0	P1	P2	P3
ODS层	事件型流水表（增量表）	永久保留	3年	365天	180天
	事件型流水表（增量表）	永久保留	3年	365天	180天
	维表（全量表）	33天+极限存储	33天+极限存储	33天+极限存储	33天+极限存储
	Merge 全量表	2天	2天	2天	2天
	普通全量表	3年	3年	3年	3年
	新同步全量表	3天	3天	3天	3天
DWD层	事件型流水表（增量表）	永久保留	3年	365天	180天
	事件型流水表（增量表）	永久保留	3年	365天	180天
	维表（全量表）	33天+极限存储	33天+极限存储	33天+极限存储	33天+极限存储
	普通全量表	3年	365天	365天	180天
DWS层	各粒度数据	永久保留	3年	3年	3年
临时存储区	ETL临时表	7天	3天	3天	3天
	TT临时表	7天	7天	7天	7天
应用层	运营报表	永久保留	——	——	——
	对外数据	7年	——	——	——
	内部产品	3年	——	——	——
本表依据来源《大数据之路》					

## 2. 数仓各层开发规范

### 1. ODS 层设计规范

#### 同步规范：

1. 一个系统源表只允许同步一次；
2. 全量初始化同步和增量同步处理逻辑要清晰；
3. 以统计日期和时间进行分区存储；
4. 目标表字段在源表不存在时要自动填充处理。

#### 表分类与生命周期：

##### 1. ods 流水全量表：

- 不可再生的永久保存；
- 日志可按留存要求；
- 按需设置保留特殊日期数据；
- 按需设置保留特殊月份数据；

##### 2. ods 镜像型全量表：

- 推荐按天存储；
- 对历史变化进行保留；
- 最新数据存储在最大分区；
- 历史数据按需保留；

##### 3. ods 增量数据：

- 推荐按天存储；
- 有对应全量表的，建议只保留 14 天数据；
- 无对应全量表的，永久保留；

##### 4. ods 的 etl 过程中的临时表：

- 推荐按需保留；
- 最多保留 7 天；
- 建议用完即删，下次使用再生成；

## 5. BDSync 非去重数据:

- 通过中间层保留，默认用完即删，不建议保留。

## 数据质量:

1. 全量表必须配置唯一性字段标识;
2. 对分区空数据进行监控;
3. 对枚举类型字段, 进行枚举值变化和分布监控;
4. ods 表数据量级和记录数做环比监控;
5. ods 全表都必须要有注释;

## 2. 公共维度层设计规范

### 1) 设计准则

1. 一致性

**共维度在不同的物理表中的字段名称、数据类型、数据内容必须保持一致**（历史原因不一致，要做好版本控制）

### 2. 维度的组合与拆分

- **组合原则:**

将维度与关联性强的字段进行组合, 一起查询, 一起展示, 两个维度必须具有天然的关系, 如: 商品的基本属性和所属品牌。

无相关性: 如一些使用频率较小的杂项维度, 可以构建一个集合杂项维度的特殊属性。

行为维度: 经过计算的度量, 但下游当维度处理, 例: 点击量 0-1000, 100-1000 等, 可以做聚合分类。

- **拆分与冗余:**

针对重要性, 业务相关性、源、使用频率等可分为核心表、扩展表。

数据记录较大的维度, 可以适当冗余一些子集。

### 2) 存储及生命周期管理

建议按天分区。

1. 3 个月内最大访问跨度 $\leq 4$  天时，建议保留最近 7 天分区；
2. 3 个月内最大访问跨度 $\leq 12$  天时，建议保留最近 15 天分区；
3. 3 个月内最大访问跨度 $\leq 30$  天时，建议保留最近 33 天分区；
4. 3 个月内最大访问跨度 $\leq 90$  天时，建议保留最近 120 天分区；
5. 3 个月内最大访问跨度 $\leq 180$  天时，建议保留最近 240 天分区；
6. 3 个月内最大访问跨度 $\leq 300$  天时，建议保留最近 400 天分区；

### 3. DWD 明细层设计规范

#### 1) 存储及生命周期管理

建议按天分区。

1. 3 个月内最大访问跨度 $\leq 4$  天时，建议保留最近 7 天分区；
2. 3 个月内最大访问跨度 $\leq 12$  天时，建议保留最近 15 天分区；
3. 3 个月内最大访问跨度 $\leq 30$  天时，建议保留最近 33 天分区；
4. 3 个月内最大访问跨度 $\leq 90$  天时，建议保留最近 120 天分区；
5. 3 个月内最大访问跨度 $\leq 180$  天时，建议保留最近 240 天分区；
6. 3 个月内最大访问跨度 $\leq 300$  天时，建议保留最近 400 天分区；

#### 2) 事务型事实表设计准则

- 基于数据应用需求的分析设计事务型事实表，结合下游较大的针对某个业务过程和分析指标需求，可考虑基于某个事件过程构建事务型实时表；
- 一般选用事件的发生日期或时间作为分区字段，便于扫描和裁剪；
- 冗余子集原则，有利于降低后续 IO 开销；
- 明细层事实表维度退化，减少后续使用 join 成本。

#### 3) 周期快照事实表

- 周期快照事实表中的每行汇总了发生在某一标准周期，如某一天、某周、某月的多个度量事件。
- 粒度是周期性的，不是个体的事务。
- 通常包含许多事实，因为任何与事实表粒度一致的度量事件都是被允许的。

#### 4) 累积快照事实表

- 多个业务过程联合分析而构建的事实表，如采购单的流转环节。
- 用于分析事件时间和时间之间的间隔周期。
- 少量的且当前事务型不支持的，如关闭、发货等相关的统计。

## 4. DWS 公共汇总层设计规范

数据仓库的性能是数据仓库建设是否成功的重要标准之一。**聚集**主要是通过**汇总明细粒度数据**来获得改进查询性能的效果。通过访问聚集数据，可以减少数据库在响应查询时必须执行的工作量，能够快速响应用户的查询，同时有利于减少不同用访问明细数据带来的结果不一致问题。

### 1) 聚集的基本原则

- **一致性**。聚集表必须提供与查询明细粒度数据一致的查询结果。
- **避免单一表设计**。不要在同一个表中存储不同层次的聚集数据。
- **聚集粒度可不同**。聚集并不需要保持与原始明细粒度数据一样的粒度，聚集只关心所需要查询的维度。

### 2) 聚集的基本步骤

#### 第一步：确定聚集维度

在原始明细模型中会存在多个描述事实的维度，如日期、商品类别、卖家等，这时候需要确定根据什么维度聚集，如果只关心商品的交易额情况，那么就可以根据商品维度聚集数据。

#### 第二步：确定一致性上钻

这时候要关心是按月汇总还是按天汇总，是按照商品汇总还是按照类目汇总，如果按照类目汇总，还需要关心是按照大类汇总还是小类汇总。当然，我们要做的只是了解用户需要什么，然后按照他们想要的进行聚集。

#### 第三步：确定聚集事实

在原始明细模型中可能会有多个事实的度量，比如在交易中有交易额、交易数量等，这时候要明确是按照交易额汇总还是按照成交数量汇总。

### 3) 公共汇总层设计原则



除了聚集基本的原则外，公共汇总层还必须遵循以下原则：

- **数据公用性**。汇总的聚集会有第三者使用吗？基于某个维度的聚集是不是经常用于数据分析中？如果答案是肯定的，那么就有必要把明细数据经过汇总沉淀到聚集表中。
- **不跨数据域**。数据域是在较高层次上对数据进行分类聚集的抽象。如以业务
- **区分统计周期**。在表的命名上要能说明数据的统计周期，如 `_Id` 表示最近 1 天，`_td` 表示截至当天，`_nd` 表示最近 N 天。

### 3. 数仓命名规范

#### 1. 词根设计规范

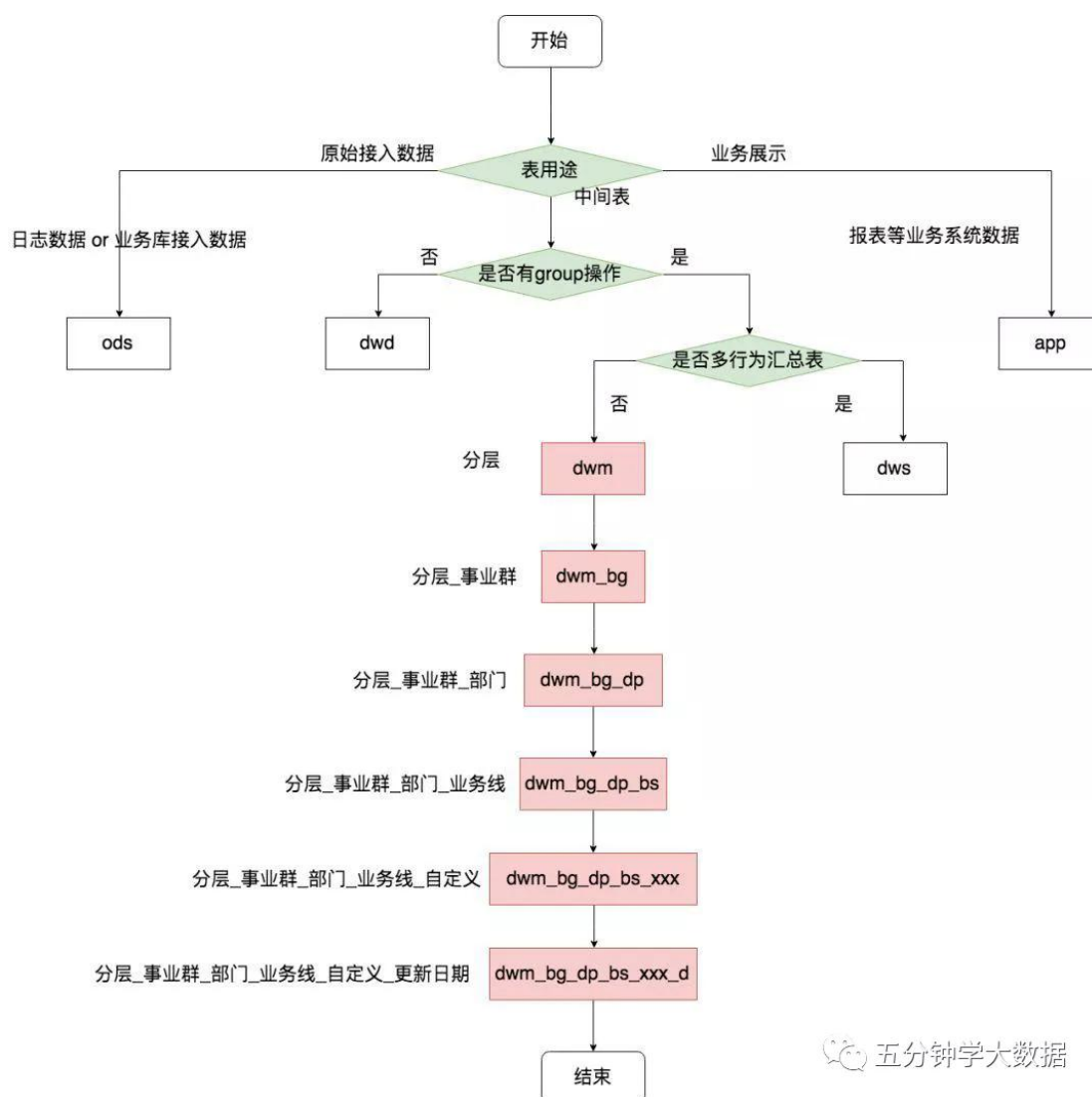
词根属于数仓建设中的规范，属于元数据管理的范畴，现在把这个划到数据治理的一部分。完整的数仓建设是包含数据治理的，只是现在谈到数仓偏向于数据建模，而谈到数据治理，更多的是关于数据规范、数据管理。

表命名，其实在很大程度上是对元数据描述的一种体现，表命名规范越完善，我们能从表名获取到的信息就越多。比如：一部分业务是关于货架的，英文名是：rack，rack 就是一个词根，那我们就在所有的表、字段等用到的地方都叫 rack，不要叫成 别的什么。这就是词根的作用，用来统一命名，表达同一个含义。

指标体系中有很多“率”的指标，都可以拆解成 XXX+率，率可以叫 rate，那我们所有的指标都叫做 XXX+rate。

**词根：可以用来统一表名、字段名、主题域名等等。**

举例：以流程图的方式来展示，更加直观和易懂，本图侧重 dwm 层表的命名规范，其余命名是类似的道理：



第一个判断条件是表的用途，是中间表、原始日志还是业务展示用的表 如果该表被判断为中间表，就会走入下一个判断条件：表是否有 group 操作 通过是否有 group 操作来判断该表该划分在 dwd 层还是 dwm 和 dws 层 如果不是 dwd 层，则需要判断该表是否是多个行为的汇总表（即宽表） 最后再分别填上事业群、部门、业务线、自定义名称和更新频率等信息即可。

**分层：**表的使用范围

**事业群和部门：**生产该表或者该数据的团队

**业务线：**表明该数据是哪个产品或者业务线相关

**主题域：**分析问题的角度，对象实体

**自定义：**一般会尽可能多描述该表的信息，比如活跃表、留存表等

**更新周期：**比如说天级还是月级更新

**数仓表的命名规范如下：**

1. 数仓层次：

公用维度：dim

DM 层：dm

ODS 层：ods

DWD 层：dwd

DWS 层：dws

## 2. 周期/数据范围：

日快照：d

增量：i

全量：f

周：w

拉链表：l

非分区全量表：a

## 2. 表命名规范

### 1) 常规表

常规表是我们需要固化的表，是正式使用的表，是目前一段时间内需要去维护去完善的表。

**规范：分层前缀[dwd|dws|ads]\_部门\_业务域\_主题域\_XXX\_更新周期|数据范围**

业务域、主题域我们都可以用词根的方式枚举清楚，不断完善。

更新周期主要的是时间粒度、日、月、年、周等。

### 2) 中间表

中间表一般出现在 Job 中，是 Job 中临时存储的中间数据的表，中间表的作用域只限于当前 Job 执行过程中，Job 一旦执行完成，该中间表的使命就完成了，是可以删除的（按照自己公司的场景自由选择，以前公司会保留几天的中间表数据，用来排查问题）。

**规范：mid\_table\_name\_[0~9|dim]**

table\_name 是我们任务中目标表的名字，通常来说一个任务只有一个目标表。

这里加上表名，是为了防止自由发挥的时候表名冲突，而末尾大家可以选择自由

发挥，起一些有意义的名字，或者简单粗暴，使用数字代替，各有优劣吧，谨慎选择。

通常会遇到需要补全维度的表，这里使用 dim 结尾。

如果要保留历史的中间表，可以加上日期或者时间戳。

### 3) 临时表

临时表是临时测试的表，是临时使用一次的表，就是暂时保存下数据看看，后续一般不再使用的表，是可以随时删除的表。

规范：tmp\_xxx

只要加上 tmp 开头即可，其他名字随意，注意 tmp 开头的表不要用来实际使用，只是测试验证而已。

### 4) 维度表

维度表是基于底层数据，抽象出来的描述类的表。维度表可以自动从底层表抽象出来，也可以手工来维护。

规范：dim\_xxx

维度表，统一以 dim 开头，后面加上，对该指标的描述。

### 5) 手工表

手工表是手工维护的表，手工初始化一次之后，一般不会自动改变，后面变更，也是手工来维护。

一般来说，手工的数据粒度是偏细的，所以暂时统一放在 dwd 层，后面如果有目标值或者其他类型手工数据，再根据实际情况分层。

规范：dwd\_业务域\_manual\_xxx

手工表，增加特殊的主题域，manual，表示手工维护表。

## 3. 指标命名规范

### 公共规则

- 所有单词小写
- 单词之间下划线分割（反例：appName 或 AppName）

- 可读性优于长度（词根，避免出现同一个指标，命名一致性）
- 禁止使用 sql 关键字，如字段名与关键字冲突时 +col
- 数量字段后缀 \_cnt 等标识...
- 金额字段后缀 \_price 标识
- 天分区使用字段 dt，格式统一（yyyymmdd 或 yyyy-mm-dd）
- 小时分区使用字段 hh，范围（00-23）
- 分钟分区使用字段 mi，范围（00-59）
- 布尔类型标识：is\_{业务}，不允许出现空值

### 参考文档：

1. [上百本优质大数据书籍，附必读清单\(大数据宝藏\)](#)
2. [最强最全面的数仓建设规范指南](#)
3. [美团数据平台及数仓建设实践，超十万字总结](#)
4. [五万字 | 耗时一个月整理出这份 Hadoop 吐血宝典](#)

### 最后：

第一时间获取最新大数据技术，尽在公众号：[五分钟学大数据](#)

搜索公众号：[五分钟学大数据](#)，学更多大数据技术！

微信直接扫码关注：



微信搜一搜



五分钟学大数据