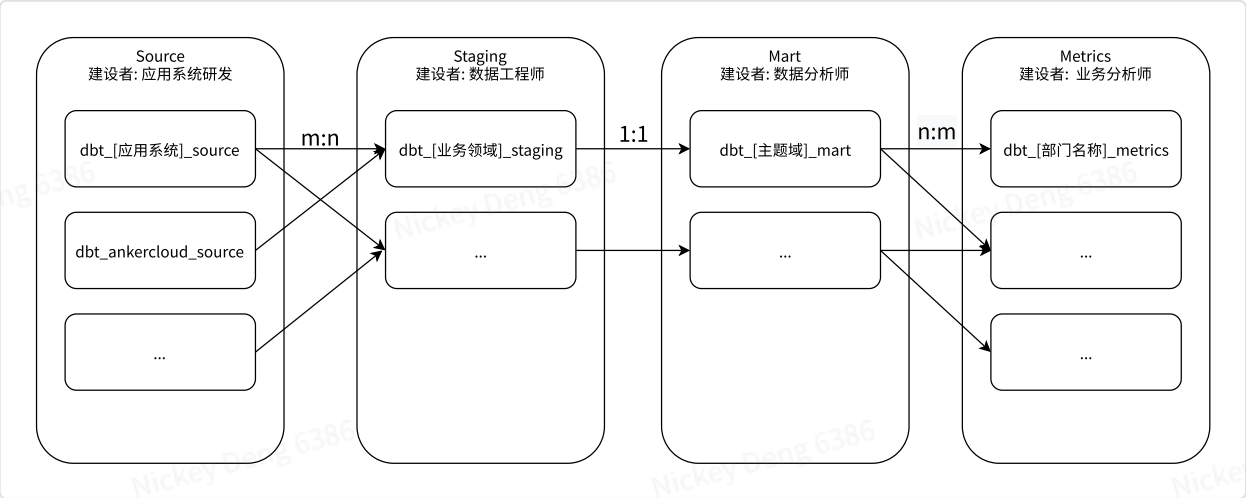
DBT项目规范

项目划分与命名规范



项目命名规范：

- Source: dbt_[应用系统名称]_source
- Staging: dbt_[业务领域]_staging
- Mart: dbt_[主题域]_mart
- Metrics: dbt_[部门名称]_metrics

通过source层来屏蔽源系统数据schema的变化

source层的主要目标有两个：

- 让工作回归到最具备对应知识的人手中，把source层回归到对应业务领域系统技术人员手中，让最懂业务领域数据库的技术人员通过source层来描述并对外发布数据资产
- 隔离业务系统对数据变化的需求与数据变化对下游的影响

因为source层是按应用系统划分，所以我们每个应用系统都形成一个独立的项目。

```CSS
dbt_oms_source
dbt_ankercloud_source
```

每个source项目都至少包含

- orig_<应用名称>.yml用来描述源业务系统中对应表的元数据信息，包括描述、测试等
- 一个 src_< 应用名称 >.yml 文件，其中包含同一目录中模型src_<应用名称>__<useful_name>的测试和文档

- 文档块中的任何额外文档都放置在同一目录中的 docs.md 文件中
- source模型给了一层数据抽象的机会，将orig中对源业务系统中数据对外发布时做了一层隔离，从而减少源业务系统数据库变化对下游数据消费的影响。一般来说，这一层的数据转换逻辑相对很轻，从而不需要一个中间层来保证DRY（Don't Repeat Yourself)
- source模型一般我们将其物化成view，并将它们隐藏在不同的schema中，不会暴露给查询我们仓库的最终用户。
- source 模型应该在数据仓库中物化为view，从而为数据调试提供机会。这些source模型应当放到特定的schema当中对外发布，而对原始orig中的表进行隐藏。
- source层不主动调度运行，应当因staging层的依赖而调度执行。应用系统的技术人员通过source项目定义好本业务领域对外发布的数据之后，此项目仅仅作为定义发布

因此，一个 source 项目可能最终看起来像：

CSS

```
dbt_oms_source
    ├──── dbt_project.yml
    └──── models
          ├──── orig_oms.yml
          ├──── docs.md
          ├──── src_oms.yml
          ├──── src_oms__customers.sql
          ├──── src_oms__orders.sql
```

通过staging层来标准化源系统数据

staging层的目标是基于业务领域创建3NF模型。它们通常按业务单元（业务领域）分组：marketing、finance、product等，这些将跟业务系统的领域驱动设计中的业务领域一一对应。staging模型获取每个业务领域的原始数据，按3NF标准对其进行清理并为进一步分析(BI, AI, DS, 图模型，标签体系等）做好准备。对于查询数据仓库的用户，带有 stg_ 前缀的模型表示：

- 按业务领域的能用语言字段以一致性的方式重命名和重转换，
- 标准化每个字段的数据类型和命名取值范围
- 已发生轻度清理，例如用 NULL 值替换空字符串
- 对半结构化数据进行扁平化
- 每个模型都有一个既唯一又不为空（并经过测试）的主键
- 定义好每个模型的主外键关系，并根据主外键关系设计好物理存储的分区键和排序键

staging模型可以在其中加入join，以便为上下文或丰富内容添加额外的列；通过union添加行并通过filter删除它们；对natural key进行重复数据删除或通过hash生成surrogate key。

因为staging层是按业务领域划分，所以我们每个业务领域都形成一个独立的项目，每个业务领域的staging层都有可以获取多个source层的数据。

```CSS
dbt_retailsales_staging
dbt_inventory_staging
dbt_accounting_staging
dbt_finance_staging
```

每个staging项目至少包含：

- 一个 stg_<业务领域>.yml 文件，其中包含同一目录中模型stg_<业务领域>__<useful_name>的测试和文档
- 文档块中的任何额外文档都放置在同一目录中的 docs.md 文件中
- 在与base模型相同的目录中的 base.yml 文件中进行测试
- staging 模型应该在数据仓库中物化为表以提高查询性能。默认情况下，我们使用table实现，在性能需要的地方，我们使用incremental实现。
- 获取staging模型所需的中间转换放置在嵌套的 models/base 目录中。它们被命名为 base_<业务领域>__<useful_name>__<transformation_in_past_tense>.sql (<transformation_in_past_tense>包括：renamed, recasted, unioned, filtered等）。前缀为base和使用双下划线表明这些是中间模型，但为了测试目的，我们将其物化成view，并将它们隐藏在不同的schema中，不会暴露给查询我们仓库的最终用户。

因此，一个 stagings 项目可能最终看起来像：

```CSS
dbt_marketing_staging
    ├─── dbt_project.yml
    └─── models
        ├─── docs.md
        ├─── stg_marketing.yml
        ├─── stg_marketing__customers.sql
        ├─── stg_marketing__orders.sql
        └─── base
            ├─── base_marketing__big_customers.sql
            ├─── base_marketing__vip_customers.sql
            ├─── base_marketing__orders_renamed.sql
            ├─── base.yml
```

通过mart层描述业务实体和过程

mart层存储描述业务实体和过程的模型。它们通常按业务单元（主题域）分组：marketing、finance、product。而那些被所有业务单元（主题域）共享的模型被分组在dbt_core_mart中，由于设计上解耦调度的依赖关系，所有的marts项目不产生项目上的上下游依赖关系，marts层之间的联合去构建指标发生在后续的metrics层中。

```CSS
dbt_core_mart
dbt_retailsales_mart
dbt_inventory_mart
dbt_procurement_mart
dbt_ordermgt_mart
dbt_accounting_mart
dbt_crm_mart
dbt_finance_mart
...
```

我们的目标是构建基于星形模型的fact和dimension和宽表模型（仅限于同主题域内的fact与dimension之间连接），这些模型是从它们所依赖的源数据中抽象出来的：

- fct_<主题域>__<动词>：一个高而窄的表格，代表已经发生或正在发生的现实世界的过程。这些模型的核心通常是一个不可变的事件流：会话、交易、订单、故事、投票。
- dim_<主题域>__<名词>：一张宽而短的表格，每一行代表一个人、地方或事物；识别和描述组织中各种实体时的最终真相来源。它们是可变的，虽然在缓慢变化：客户、产品、候选人、建筑物、员工。
- bp_<主题域>__[<名词>]_<动词>_[<名词>]：一张将原子的fact表和dimension表join后进行where, groupby, window等操作后并带有指标计算公式的宽表，但bp表只限join本主题域的fact和dimension表。

在staging模型的工作是基于3NF清理和准备一致性数据的情况下，事实表是实质性数据转换的产物：选择（和减少）维度、date-spinning、执行业务逻辑以及做出明智、自信的决策。

这一层模型是根据组织的灵活分析需求高度定制的。因此，当涉及到这些模型时，我们的惯例要少得多，但我们有一些可用的模式是：

- fct_ 、dim_和bp_ 模型应该在数据仓库中物化为表以提高查询性能。默认情况下，我们使用table实现，在性能需要的地方，我们使用incremental实现。
- 获取事实或维度模型所需的中间转换放置在嵌套的 models/intermediate 目录中。它们被命名为 int_<主题域>__<useful_name>__<transformation_in_past_tense>.sql (<transformation_in_past_tense>包括：grouped, joined, windowed, filtered等）。前缀为int和使用双下划线表明这些是中间模型，不值得信任，我们经常使用ephemeral模式，因此它们不会暴露给查询我们仓库的最终用户
- 模型在与模型相同的目录中的 mart_<主题域>.yml 文件中进行测试和记录。
- 文档块 中的任何额外文档都放置在同一目录中的 mart_<主题域>.md 文件中。

因此，一个 marts 目录可能最终看起来像：

```
dbt_marketing_mart
     ├─── dbt_project.yml
     └─── models
          ├─── docs.md
          ├─── mart_marketing.yml
          ├─── dim_marketing__customers.sql
          ├─── fct_marketing__orders.sql
          ├─── bp_marketing__customers_pay_orders.sql
          └─── intermediate
               ├─── int_marketing__customer_orders__grouped.sql
               ├─── int_marketing__customer_payments__grouped.sql
               ├─── intermediate.yml
               └─── int_marketing__order_payments__joined.sql
```

通过metrics / BI 层构建指标体系

metrics层从marts层的fact, dimension和bp表中抽取数据来精确定义指标，直接用于时序报告，并且结构紧密以便与目标和预测进行一对一的比较。主要用来存储描述指标及被指标依附的模型。它们通常按业务部门分组：oso、rso、marketing等。

```
dbt_d3_metrics
dbt_oso_metrics
dbt_rso_metrics
dbt_marketing_metrics
dbt_cn_metrics
dbt_jp_metrics
...
```

metrics层与BI工具共同构建公司和部门的指标体系。一般来说，metrics层模型的设计的驱动来自业务部门指标根据相关性聚类的结果。所有的指标都应该落在metrics层的某个模型之上。前缀为tmp和使用双下划线表明这些是中间模型，不值得信任，我们经常使用ephemeral模式，因此它们不会暴露给查询我们仓库的最终用户。用户真正消费是metrics模型物化优先使用view方式。层层剥开，最后所有的metrics层模型都由marts层模型构成，整个过程保证DRY (Don't Repeat Yourself）。

CSS

```
dbt_d3_metrics
     ├─── dbt_project.yml
     └─── models
          ├─── docs.md
          ├─── metrics_d3.yml
          ├─── bi_d3__customers.sql
          ├─── bi_d3__orders.sql
          ├─── bi_d3__customers_pay_orders.sql
          └─── tmp
               ├─── tmp_d3__customer_orders__grouped.sql
               ├─── tmp_d3__customer_payments__grouped.sql
               ├─── tmp.yml
               └─── tmp_d3__order_payments__joined.sql
```