数据库的表设计之初步设计

屈春河

创建日期：2020-08-11

1 表的分类

对于大多数企业级应用系统而言，数据库是整个应用系统的基石，因此数据库的表设计（也被称为 schema 设计）也就成为整个系统设计的重中之重。表设计的不好或者不合理，不仅会影响系统性能，而且会增加开发和集成的复杂性，甚至埋下隐患，最终会导致一系列的问题，例如数据不一致性问题等。概括而言，需要根据业务需求和系统功能，采用如下步骤设计表。

1) 确定数据库类型。
2) 判断表的类型，并据此构建表的列，确定表名。
3) 设计表的主键以及表之间的关联关系。
4) 优化设计，评估访问性能并根据查询模式添加索引。

应用系统往往涉及很多表，这些表的设计存在先后顺序，而表之间也具有各种关联和依赖关系。因此，表的设计并不是一蹴而就的，步骤 2）、3）和 4）在大多数时候也不是清晰可分的，甚至整个设计也是一个循环迭代的过程，需要在多个步骤之间反复多次，以逐步的细化和优化。

基于所支持应用的不同，数据库可以划分为两大类，分别为面向分析型应用的 Online Analytical Processing（简称为 OLAP）和面向事务型应用的 Online Transaction Processing（简称为 OLTP）。OLAP 用于支持数据挖掘、统计报表、数据预测等统计分析功能，插入、更新和删除这些写入操作较少，但是对于数据读取和数据处理的要求非常高。OLTP 用于支持事务处理功能，需要全面满足操作的原子性（Atomicity）、一致性（Consistency）、事务隔离性（Isolation）和持久性（Durability），即所谓的 ACID 标准。

表设计的第一个步是根据业务需求，确定应用是分析型，还是事务型，从而确定数据库的类型是 OLAP，还是 OLTP。两种类型的数据库在设计理念和设计方法是完全不同的。本文主要讨论针对于 OLTP 的表设计，在下文中如果没有特别指出，那么所指的都是 OLTP 数据库。在大多数场景中，事务型应用也需要数据统计分析功能，但是通常并不会为此构建专门的 OLAP 数据库，而是基于现有的 OLTP 数据库支持这些统计分析功能。为此，需要数据库以支持在线事务处理为主，同时辅助支持统计分析功能。这种情况造成数据库的表往往可以划分为如下三类。

- 业务表，其操作特点是随机增、随机删、随机改、随机查。业务表来源于需要应用系统处理的业务实体，业务实体所包含的基本属性和状态被称为业务属性，其与业务表的列之间存在对应关系。
- 日志表，其操作特点是顺序增、不删、不改、多查。日志表用于记录原始日志或者历史数据，顾名思义，之所以被成为日志表，是因为这些日志和数据一旦生成，就不会被更改，也

不会被删除。

- 汇总表，其操作特点是顺序增、不删、少改、多查。对于数据统计分析功能，如果直接使用日志表无法满足性能要求，则可以引入汇总表。在本质上汇总表是以预先汇总数据和存储汇总结果为代价，来提高数据统计分析功能的访问性能。基于不同的业务场景和业务需求，既可以在将数据插入日志表后，实时地以逐条方式插入或者更新汇总表，也可以间隔一定周期，定时地以批量方式读取日志表中的数据并执行汇总操作，然后批量地将汇总结果插入汇报表。

表的设计步骤 2）和 3）对应于表的初步设计阶段，其是一个从无到有的过程，即从业务需求和系统功能入手，首先根据数据来源、操作特点和所支持的功能，确定表的类型，然后在此基础上逐步地构建表，包括设计列以及确定主键和关联关系。通常而言，在初步设计过程中要依照功能的依赖关系，先设计业务表，再设计日志表，最后设计汇总表。在完成全部表的初步设计后，还需要估算数据规模，并根据各个功能模块对于数据库的访问模式，评估访问性能和添加索引。

2 表的构建

对于业务表、日志表和汇总表三种不同类型的表，不仅所对应的功能不同，设计目标不同，而且表的构造过程也不尽相同。

业务表的目标是减小冗余数据，以提高数据的完整性（Integrity）和一致性（Consistency），其构造过程如下所示。

1) 归纳和抽象出需要持久化的业务实体。业务实体是指在系统应用中需要依据业务规则/逻辑进行各种处理的对象，其中一部分业务实体需要设计专门的表，以在数据库中存储相应的数据，而其他业务实体，或者衍生自需要持久化的业务实体，或者为临时性业务实体，因此无需为其设计表。
2) 总结相应的业务属性。不同于业务实体，业务属性往往为基本的数据类型，例如小数、整数、字符串等。应用系统依据业务逻辑对业务实体执行一系列的处理，这些处理最终能够被分解为对于业务属性的增、删、改、查操作。
3) 基于业务实体和业务属性，设计对应的表和列。表和表的列分别对应于业务实体和业务属性。根据功能需求，有时还会增加一些额外的列，例如 created_time 和 status 等，用于支持一些辅助性的处理功能。

冗余数据的字面意思是多余的和不必要的数据。冗余数据的一种极端情况是重复数据，可能所在列名不同，甚至所在表也可能不同，但是在两列中对应的数据是完全相同的。冗余数据的另一种常见情况是数据依赖，即一列数据可以由其他一列或者几列数据推导出来。例如商品表中的三列：原价、现价和折扣，因为现价 = 原价 * 折扣，所以其中任何一列都可以由其他两列计算而来，因此存在数据依赖，仅仅需要保留其中两列即可，具体去除那一列需要根据业务需求和使用场景来决定。此外，关系冗余也是一种常见的冗余数据，包括部分依赖和传递依赖。部分依赖是指在使用复合主键时非主键的列仅仅依赖于部分主键，而不是依赖于整个主键。传递依赖是指非主键的列不直接依赖于主键，而是依赖于其他非主键的列。为了消除冗余关系，需

要根据具体情况，或者删除一些相关的列，或者将冗余关系抽取出来独立建表。无论属于那种数据冗余，都需要从业务逻辑和业务含义上来分析和判断，尝试更改一列数据，看看更改此列数据后，是否需要进一步地更改这个表或者其他表的列，以判断是否存在依赖此列的数据。对于业务表而言，一旦出现冗余数据，那么在更新部分冗余数据时，必需同时额外地更新冗余数据的其他部分，否则将会出现数据不一致问题。显而易见，冗余数据将会大大增加数据更新的复杂性，所以必须在业务表中消除各种形式的冗余数据。

设计业务表主键时，采用如下原则。

1) 建议优先使用存在业务含义的、类型为整数的并且具有唯一性的一列或者多列为主键，即现有的一列或者多列为整数类型并且能够唯一地标识一行数据，那么就采用这一列或者多列做主键。如果主键为多列，那么也被称为复合主键。
2) 如果不存在列满足设计原则 1），那么建议创建自增长主键，即设计由关键字 NOT NULL 和 AUTO_INCREMENT 修饰的整数列作为主键。

针对于业务表主键的设计原则，有如下几点说明。其一，之所以建议优先采用具有业务含义的主键，是因为可以获得更好的性能。具有业务含义的列常常作为查询或者更新条件，作为主键可以更加快速地定位数据在磁盘上的存储位置。虽然自增长主键能够实现顺序插入，具有更好的插入性能，但是数据查询和数据更新的操作频率要远远大于数据插入操作，减小查询和更新操作的时延可以获得更好的数据库平均访问性能。其二，在一个表中没有业务含义的自增长主键，在其他表中就存在确定的业务含义，例如在如下代码片段中虽然 member 表的 id 是自增长主键，并没有业务意义，但是在表 last_login 中这个 id（member_id）就具有了业务含义，能够代表一个特定的 member。其三，如果是复合主键，那么在复合主键列中不同列的排列顺序需要仔细考量，一般依据查询模式，越经常作为查询条件的列，在主键中的位置越靠左（越靠前）。其四，ENUM、DATE、DATETIME、TIME 和较短的 CHAR 这些类型所占的存储空间较小，在设计主键时可以等同于整型对待。其五，如果存在一列或者多列，其具有业务含义并且能够唯一标识一行数据，但不是整数（通常为字符串），则有两种解决方案。方案一，如下代码片段中 member 表所示，email 具有唯一性，但不是整数，可以采用自增长主键 + 唯一索引的方式，其中如果为唯一索引为多列，那么在唯一索引中的排列顺序依据复合主键的排列原则来处理。方案二，类似于 member2 和 email 表，将多列中非整数型的列提取出来建立独立字典的表，假设表 t1 中存在三列能够唯一标识每行数据，分别为 c1、c2 和 c3，其中 c1 为整数，c2 和 c3 为 VARCHAR 类型，则从 t1 表中抽离两列 c2 和 c3，分别建立两个新的字典表 t2 和 t3，在新表 t2 和 t3 中采用自增长主键，并且在两个表中 c2 和 c3 所对应的列分别建立唯一索引，而在原表 t1 中采用 t2 表和 t3 表的主键 t2_id 和 t3_id 替换 c2 和 c3 列，并用 c1 和 t2_id、t3_id 三列做复合主键。

```
CREATE TABLE member (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...

  email VARCHAR(63) NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY (email)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE last_login (
  member_id INT UNSIGNED NOT NULL,
  ...

  PRIMARY KEY (member_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE member2 (
  email_id INT UNSIGNED NOT NULL,
  ...

  PRIMARY KEY (email_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE email (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  value VARCHAR(63) NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY (value)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

日志表的目标是尽可能全面的和完整的记录数据，以便于支持现在所需的和未来可能的各种数据分析应用，其构造过程如下所示。

1) 收集和整理需要存储的原始日志或者历史数据。
2) 针对收集到的数据，抽取其中的信息，并关联其他的相关信息。
3) 将信息分解为一系列的信息项，每个信息项都为基本的数据类型。
4) 将这些信息项划分为不同的表。

不同于业务表，日志表可能存在大量的冗余数据。之所以许可日志表有冗余数据，是因为如下几方面原因。其一是为了保存易变的关联数据，例如在订单历史表中每件商品都会记录单价，理论上通过商品信息（商品 id）能够关联得到商品的价格，因此价格是一个冗余数据，但是对于一件商品而言，其价格会随着时间进行调整，也就是说价格会经常变动，实时关联所得到的价格很可能已经不是在商品出售时的价格。其二是为了避免 JOIN 业务表，如果预估日志表和业务表的数据规模都非常庞大，那么在日志表中添加一些相关联的、来自业务表的数据，可以在很多场景中有效地减小日志表与业务表之间的 JOIN 操作，从而避免 JOIN 操作所引起的性能下降或者性能抖动。其三是数据一旦插入日志表中，就不会被更改，因此在一个日志表中每行数据为在数据生成时的关联数据和上下文数据，在每行数据中并不会出现数据不一致问题。

由于大量冗余数据，日志表的列数往往较多。如果列数过多，尤其是 VARCHAR 类型的列过多，那么日志表的访问性能会较差。一方面，当前关系型数据库广泛采用行存储方式，即使仅仅访问其中一列数据，也需要首先从磁盘中读取整行数据。另一方面，大多数的统计分析应用仅仅需要读取日志表中的部分列，而不是全部列。为此，针对于一些行数较多的日志表，可以采用垂直分表，即将这些列合理地拆分到多个相关的日志表中，并通过主键将这些表关联起来。

日志表的一个特点是每行数据或者具有数据的生成时间或者可以添加插入数据库的时间，

结合这个时间戳构建主键可以优化数据存储和数据查询。故而，日志表主键的设计原则如下。

1) 建议优先使用包含时间戳在内的、存在业务含义的、类型为整数（包括 ENUM、DATE、DATETIME、TIME 和较短的 CHAR 等占用存储空间较小的类型）的并且能够唯一标识一行数据的多列作为复合主键。例如在如下代码片段中 login_history 表采用 login_time 和 member_id 做复合主键，其中 login_time 是从格林威治时间 1970 年 1 月 1 日 00 点 00 分 00 秒到当前登陆时刻的总毫秒数。
2) 如果不存在满足设计原则 1）的列，那么建议采用时间戳 + 自增长整数做主键。存在两种具体地实现方案。方案一如下代码片段中表 demo1 所示，采用 64 位无符号整数，其中前 32 位为 Unix 时间戳，表示从格林威治时间 1970 年 1 月 1 日 00 点 00 分 00 秒到当前的总秒数，后 32 位为自增长整数，当然也可以采用分布式 ID 方案 [Chunhe]，以实现分布式入库数据。方案二如表 demo2 所示，采用两列 created_time 和 id 做复合主键，其中 created_time 为 Unix 时间戳，id 为自增长整数。InnoDB 引擎不支持在复合主键中使用关键字 AUTO_INCREMENT，因此需要依靠应用来实现自增长整数。
3) 如果没有其他可选择的主键，那么最后才选择构建一列自增长的整数作为主键。对于自增长主键，如果日志表的数据量比较大，建议定期地将数据转移到专门的历史表中，例如以年或者月为周期分别转移到历史表 ****_yyyy 或者 ****_yyyy_mm 中。

```
CREATE TABLE login_history (
  login_time BIGINT UNSIGNED NOT NULL,
  member_id INT UNSIGNED NOT NULL
  ...

  PRIMARY KEY (login_time, member_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE demo1 (
  id BIGINT UNSIGNED NOT NULL,
  ...

  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE demo2 (
  created_time INT UNSIGNED NOT NULL,
  id INT UNSIGNED NOT NULL,
  ...

  PRIMARY KEY (created_time, id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

与自增长主键相比，时间戳不仅表示数据插入的顺序，而且可以作为日志表的查询条件。在绝大多数情况下日志表的查询条件中都会包含时间范围，因此使用时间戳作为复合主键，可以将主键进一步地用于分区（Partition）和优化数据存储，从而大大提升数据库的访问性能。

汇总表是以减小数据库的响应时间为目标，其构造过程如下所示。

1) 挑选出交互性的、高频使用的统计分析需求/功能。汇总数据是有代价的，需要尽可能地避免不必要的数据汇总操作。因此，如果直接访问日志表就能够满足性能要求，那么就无需为此构建汇总表。此外，对于非交互性或者不经常使用的数据分析功能，建议不专门设计汇总表，而是利用已有的汇总表或者直接从日志表中实时地汇总数据。
2) 针对挑选出的统计分析需求/功能，分解和归纳出所需要的信息项。这些信息项可以划分为两类。一类称为统计属性（Attribute）或者统计维度（Dimension），常常用于查询条件，一般为可枚举的类型，例如整型、Date、ENUM 等类型。对于类似国家和地区这类字符串类型的统计维度，在设计表时一般会采用字典表存储，并通过字典表 id 进行关联，从而可以看作一种特殊的 ENUM 类型。另一类称为统计指标（Metric），一般为整型或者小数，统计指标还可以继续划分为基本指标和可由基本指标计算得来的指标（简称为可计算指标）。通常情况下，仅仅需要考虑基本指标即可，只有在数据规模异常庞大并且计算代价异常高昂的情况下，才需要存储可计算指标。
3) 针对于统计维度和基本指标，进行归约和合并，然后划分为不同的表。一个维度可能对应多个不同的粒度（有时也被成为层级），例如时间周期/时间间隔，可能为分钟、小时、天、月等不同的粒度。如果维度相同，但粒度不同，那么可以剔除粒度较大的信息项，仅保留粒度最小的信息项，因为基于最小粒度的数据，通过 SUM 操作就可以得到更大粒度的汇总数据。在剔除维度相同但是粒度较大的信息项后，对于重复的信息项进行合并和剔重，然后根据汇总数据来源或者所支持的功能，将这些信息项划归为不同的表，每个信息项对应于表的一列。
4) 估算数据规模，对汇总进行拆分和组合。对于一个汇总表而言，当统计维度越少、统计粒度越粗时，则数据量越少，查询速度越快，相反地，当统计维度越多、统计粒度越细时，则数据量越大，所支持的功能也越丰富。在上述步骤 3）完成之后，需要预估每个汇总表可能的数据规模，并估算所对应统计分析功能的访问性能。如果预估数据规模庞大并且统计分析功能的访问性能较差，那么可以尝试减小维度和/或增加粒度，以满足性能要求。如果维度和粒度之间还无法兼顾，则可以进一步地针对不同的功能/需求，将汇报表拆分为多个汇总表，即针对不同维度和粒度，建立多个汇总表。一般而言，维度多的汇总表，其粒度较粗，而维度少的表，其粒度较细。在一些极端情况下为了进一步地降低响应时间，许可一个数据数据分析功能就对应一个数据汇总表。

在一个汇总表中不同统计维度之间通常情况下是相互独立的。如果需要多个不同粒度的汇总数据，那么理论上仅仅需要在汇总表保留最细粒度的统计维度即可，较高粒度的数据可以通过细粒度的数据汇总得来。例如地区由粗到细可以划分为三个粒度，分别为国家、省/州/自治区和市，三个粒度分别对应三个表 country、subdivision 和 city，主键分别为 country_code、subdivision_id 和 city_id，并且 city 表和 subdivision 表分别通过 subdivision_id 和 country_code 关联到 subdivision 表和 country 表。如果同时需要上述三个粒度的订单地区统计表，那么仅仅需要市粒度的汇总数据，即订单地区统计表仅仅需要 city_id 列，国家和省/州/自治区两个粒度的订单统计数据可以由城市粒度的数据通过 JOIN 和 SUM 得到。然而，如果一个维度的不同粒度所对应的数据规模异常庞大或者为了避免 JOIN 操作，那么在一个汇总表中许可使用同一个维度的不同粒度作为列，也就是说可能存在相互依赖的多个列。假设国家、省/州/自治区和市三个粒度所对应的行数分别

为千万、亿和十亿级别，那么在原有的市粒度（city_id）基础上，在订单地区统计表中再增加国家（country_code）和省/州/自治区（subdivision_id）两个粒度（需要分别添加两列作为索引）。这虽然会造成地区统计维度的重复和列之间的依赖，但是却可以大大提高包含国家和省/州/自治区条件的数据汇总性能。

对于汇总表的主键，建议遵循如下设计原则。

1) 如果统计维度之间相互独立，那么选择统计维度的列作为复合主键。复合主键的多列按照如下规则排列：a）如果统计维度中存在时间周期/时间间隔，那么时间周期/时间间隔所在的列位于复合主键的最左面（最前面）；b）越经常作为查询条件的统计维度，在复合主键中的位置越靠左（越靠前）。
2) 如果存在一个统计维度的不同粒度，那么仅仅选择最细粒度的列作为复合主键，并且复合主键的多列遵守设计原则 1）中的规则进行排列。

类似于日志表，在汇总表中建议将时间周期/时间间隔置于复合主键的最左面，是因为在访问日志表时时间范围往往是条件之一，从而可以根据时间范围进行分区和优化查询性能。基于周期/间隔的不同，时间周期/时间间隔可以选择不同的定义格式。对于天，建议采用 DATE 类型，但是也许可使用 yyyymmdd 格式的 INT UNSIGNED 类型。对于小时，既可以采用两列，并且两列的类型分别为 DATE 和 TINYINT UNSIGNED，分别用于存储日期和小时，也可以采用 INT UNSIGNED 定义的一列，以 yyyymmddHH 格式同时存储日期和小时。对于分钟或者秒粒度的周期/间隔，建议采用 INT UNSIGNED 定义的 UNIX 时间戳，例如对于 30 秒的周期/间隔 [t, t+30)，可以采用时间 t 对应的 UNIX 时间戳。

对于三种类型的表，无论那种类型，在其初步设计过程中都不可避免地需要重复地尝试对表进行拆分和合并，即将一个表拆分为多个表或者将多个表合并为一个表。在表的分拆与合并过程中需要参考如下几方面因素。其一是业务逻辑，基于业务含义或者业务意义，尽量将业务相关性强和业务关联紧密的列放到一个表中。其二是读取模式，尽量将经常同时一起访问的列放到一个表中，以优化读取性能。其三是更新模式，尽量将那些频繁更新的列放到一个表中，以优化数据更新性能。

3 表的关联

顾名思义，关联就是将位于一个表或者多个表内相关的多行数据相互联系起来，也就是说，通过一个表的一行数据，可以访问到此表或者其他表中与此行数据相关的一行或者多行数据，甚至被关联的数据还可以继续关联其他数据，从而基于一行数据就可以获得与之相关的完整数据。

表的关联来自如下几个方面。

- 业务实体关系。业务实体之间的关系各种各样，例如包含、依赖、从属、组成等等。这些种类繁多的关系在数据库层面最终都会表示为表的关联。例如雇员（employee）和部门（department），又如地区的三个粒度国家（country）、省/州/自治区（subdivision）和市（city）。虽然从业务层面看上这些业务实体之间的关系迥异，但是在数据库层面上都能通过统一的表关联关系来支持。
- 字典表的应用。对于 VARCHAR 类型或者较长 CHAR 类型定义的列，如果不同行的取值可

能出现较多的重复，那么建议增加字典表用于存储字符串，并且在原来的表中使用字典表的整数型主键替换原来的字符串。例如在上文代码片段中表 member2 和表 email，其中表 email 充当了字典表，并且为了确保取值的唯一性，value 添加了唯一索引，而在 member2 中采用字典表 email 的主键作为一列 email_id，以取代具体的 email 值。

- 范式的正规化。范式（Normal Forms）是从实践中提炼出来的一系列基本的标准或者准则，用于对表进行正规化或者规范化（Normalization）。目前广泛使用的范式包括第一范式（1NF）、第二范式（2NF）、第三范式（3NF）和 Boyce-Codd 范式（BCNF）4 个范式，其中后三个范式用于避免部分依赖或者传递依赖。一旦违反后三个范式，就需要对表进行拆分，从而在被拆分出来的表和原来的表之间形成关联。
- 优化访问性能。关系型数据库广泛采用行存储方式，也就是说，数据以行作为在磁盘上的存储和访问单位，即使读取或者更新一行数据中的一列，也需要将整行数据从磁盘中读取出来。为了减小不必要的 I/O 操作和优化访问性能，需要尽量避免一个表具有过多的列。对于列数过多的表，首先根据业务逻辑和业务含义的相关性，从业务层面进行拆分。如果无法从业务层面进行拆分，则可以根据读取模式或者更新模式，将那些频繁读取或者更新的列拆分出来。拆分之后，通常需要在拆分出来的多个表之间建立关联关系。

一个关联关系在大多数情况下会连接两方数据，这两方数据既可以分别位于两个不同的表，也可以位于一个表中的不同行。需要指出的是，表的关联关系是双向的，即从两方中的任何一方都可以访问到两方的完整数据。根据关联两方行数的对应关系，表的关联可以划分为如下三类。

- 一对一关系（One-to-One Relationship）。在关联的两方中任何一方的一行数据仅仅对应另一方的一行数据，反之亦然。例如在上文代码片段中的表 member 和表 last_login 就是一对一的关联关系，member 表中的一行数据（一个 member）对应 last_login 表中的一行数据（最后一次登录记录）。相反方向，last_login 表中的一条最后一次登录记录（如果存在的话）也对应 member 表的一个 member。
- 一对多关系（One-to-Many Relationship）。在关联的两方中从一个方向上，一方的一行数据对应另一方的多行数据，而从相反方向上，另一方的一行数据却仅仅对应一行数据。例如国家（country）和省/州/自治区（subdivision）就是一对多的关系，country 表的一行数据（一个国家）对应 subdivision 表的多行数据（多个省/州/自治区），而 subdivision 表中的一个省/州/自治区仅仅属于 country 表中的一个国家。
- 多对多关系（Many-to-Many Relationship）。在关联的两方中任何一方的一行数据对应另一方的多行数据，反之亦然。例如学生（student）和课程（course）就是多对多的关系。因为一个学生通常会选择多门课程，因此 student 表中的一行数据对应 course 表的多行，而多个学生会选择同一门课程，因此一行 course 表中的数据也会对应多行 student 表中的数据。

表的关联在数据库层面需要设计专门的列，用于额外存储所关联数据的主键。在被关联的两方中，任何一方通过本方的一行数据（或者一个主键）就可以访问这个主键的拷贝，进而获得所需要的另一方面数据。对于上述三种类型的关联关系，在具体实现如何存储主键的拷贝以及如何访问到这个主键拷贝上有较大差异。

对于一对一的关联关系，存在如下三种实现方式。第一种方式是相关联的两方数据采用相同的主键，例如上文代码片段中的 member 表和 last_login 表，由于先有 member 数据，后有

last_login 数据，因此 last_login 表采用 member 表的主键作为自己的主键 member_id。第二种方式是在一方的表中采用专门的、非主键的列存储另一方数据的主键，并且为了确保一对一的对应关系，此列通常会添加唯一索引。如下代码片段中表 last_login2 所示，该表存在独立的自增长主键，并通过列 member_id 存储 member 表主键的拷贝。第三种方式是采用专门的关联表来存储两方数据的主键，并且在通常情况下两个主键所对应的列会分别作为这个关联表的主键和唯一索引，如下代码片段中表 last_login3_member_association 就是一个专门的关联表，用于存储表 last_login3 和表 member 之间的关联关系。在上述三种方式中第一种实现方式较为简单，但是如果采用 AUTO_INCREMENT 自增长主键，则需要满足如下条件：相互关联的两方数据具有严格的插入顺序，即总是在一方的数据先插入数据库之后，另一方的数据才能插入数据库。如果相关联的两方数据需要几乎同时插入数据库，则需要由应用来生成主键或者使用具有业务含义的主键。如果相关联的两方数据的插入是相互独立的，也就是说无法确保两方数据的插入顺序，那么建议采用第三种方式来实现一对一的关联。

```
CREATE TABLE last_login2 (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  member_id INT UNSIGNED NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY (member_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE last_login3 (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE last_login3_member_association (
  member_id INT UNSIGNED NOT NULL,
  last_login_id INT UNSIGNED NOT NULL,
  PRIMARY KEY (member_id),
  UNIQUE KEY (last_login_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对于一对多的关联关系，存在两种实现方式。第一种方式是在一对多关系中对应多的一方添加专门的列用于额外存储另一方（在一对多关联中对应一的一方）的主键，例如 country 和 subdivision 是一对多的关系，如下代码片段中 subdivision 表采用 country_code 列来存储 country 表的主键。此种方式的一种特殊情况是相关联的两方数据都位于一个表中。对于这种特殊情况，也会在表中构建专门的列，但是对于一对多关系中对应一的那些行，此列采用特殊的取值，以区别于正常的关联关系。例如，在企业组织架构中除了极少数 empolyee 之外，每个 employee 都会向一个特定 leader 汇报工作，而 leader 本身也是一个 employee，并且可能也会存在其对应的 leader。如下代码片段中 employee 表，每个 employee 对应表中的一行，并通过 leader_id 关联到表中其 leader，而对于极少数没有 leader 的 empoyee，其 leader_id 设置为 0，因为不存在 id 为 0

的 employee。第二种方式是采用专门的关联表来存储关联两方的主键，并且一般情况会采用多的一方所对应的列作为关联表的主键，如下代码片段中 employee_department_association 表就是一个专门的关联表，用于存储 employee 和 department 之间的关联关系，并且使用 emplyee 表的主键作为自己的主键 employee_id。

```
CREATE TABLE country (
  code CHAR(2) NOT NULL,
  ...
  PRIMARY KEY (code)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE subdivision (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  country_code CHAR(2) NOT NULL,
  PRIMARY KEY (id),
  KEY (country_code)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE employee (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  leader_id INT UNSIGNED NOT NULL DEFAULT 0,
  PRIMARY KEY (id),
  KEY (leader_id),
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE department (
  id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE employee_department_association (
  employee_id INT UNSIGNED NOT NULL,
  department_id SMALLINT UNSIGNED NOT NULL,
  PRIMARY KEY (employee_id),
  KEY (department_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对于多对多的关联关系，会采用专门的关联表来存储关联两方的主键。在关联表中常常联合双方的主键作为复合主键。如果不使用复合主键，而是采用自增长主键，那么要联合双方的主键作为唯一索引，以保证关联数据的唯一性，避免出现重复的数据。如下代码片段中 student 表和 course 表数据为多对多的关系，enrollment 表是关联表，存储 student 表和 course 表的主键，并使用这两个表的主键作为自己的复合主键。

```
CREATE TABLE student (
```

```
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE course (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ...
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE enrollment (
  student_id INT UNSIGNED NOT NULL,
  course_id INT UNSIGNED NOT NULL,
  PRIMARY KEY (course_id, student_id),
  KEY (student_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

参考文献

CHUNHE Q. 分布式 ID 的应用和考量[EB/OL]. https://github.com/QuChunhe/blogs/blob/master/files/2020-06-14_%E5%88%86%E5%B8%83%E5%BC%8FID%E7%9A%84%E5%BA%94%E7%94%A8%E5%92%8C%E8%80%83%E9%87%8F.pdf.