# 数据库设计规范文档

## 一、文档前言

### 1.1 目的

本规范旨在统一数据库设计标准，确保数据库具备高效性、稳定性、安全性、可扩展性及可维护性，降低数据冗余与不一致风险，保障业务系统稳定运行，同时为开发团队提供明确的设计指引，提升协作效率。

### 1.2 适用范围

本规范适用于公司内部所有业务系统的数据库设计工作，包括新系统数据库设计、现有系统数据库优化与迭代，涉及关系型数据库（如 MySQL、Oracle、SQL Server 等），非关系型数据库可参考本规范核心原则结合自身特性调整。

### 1.3 术语定义

* **三范式（3NF）**：指在数据库设计中，满足第一范式（确保每列原子性）、第二范式（消除部分函数依赖）、第三范式（消除传递函数依赖），以减少数据冗余。
* **主键（Primary Key）**：用于唯一标识表中每条记录的字段或字段组合，不可为空且值唯一。
* **外键（Foreign Key）**：用于建立表与表之间关联的字段，其值需与关联表的主键值匹配，维护数据一致性。
* **索引**：数据库中用于加快数据查询速度的数据结构，通过特定算法对字段值进行排序存储，减少查询时的磁盘 I/O 操作。
* **分区表**：将大型表按照特定规则（如时间、地域、范围等）拆分为多个小型子表，提升查询与维护效率。

## 二、数据库设计基本原则

### 2.1 需求导向原则

设计前需通过业务调研、需求评审等方式，充分理解业务场景、数据流转逻辑、用户访问模式及未来 3-5 年的业务扩展需求，确保数据库设计与业务目标高度匹配，避免因需求理解偏差导致后期大规模调整。

### 2.2 标准化原则

遵循数据库设计范式（优先满足第三范式），减少数据冗余与数据不一致问题。若存在性能优化需求（如高频查询场景），可在满足核心业务逻辑的前提下，适当放宽范式要求（如增加冗余字段），但需在设计文档中明确说明原因与影响范围。

### 2.3 性能优先原则

在设计阶段充分考虑性能因素，包括合理设计索引、选择合适的数据类型、优化表结构等，避免因设计缺陷导致后期查询缓慢、系统卡顿。同时，需平衡读写性能，针对读多写少或写多读少的场景，分别制定差异化设计策略（如读多场景增加索引，写多场景减少索引）。

### 2.4 安全性原则

设计时需融入数据安全理念，包括敏感数据加密存储、严格的权限控制、数据备份与恢复机制等，防止数据泄露、篡改或丢失，满足行业合规要求（如隐私保护相关法规）。

### 2.5 可扩展性原则

表结构、字段设计需预留扩展空间，例如采用 “预留字段”（如 reserve1、reserve2，需在文档中说明用途）、使用枚举类型而非硬编码、避免字段长度过度限制等，便于后续业务扩展时无需大规模修改表结构。

### 2.6 一致性原则

数据库设计的命名规则、字段类型、约束条件等需保持统一，例如同一业务含义的字段（如用户 ID）在不同表中需使用相同的字段名与数据类型，避免因设计不一致导致开发混乱与数据集成问题。

## 三、表结构设计规范

### 3.1 命名规范

#### 3.1.1 数据库命名

* 采用 “业务域\_系统名称” 的格式，全小写字母，单词间用下划线分隔，例如 “ecom\_shop”（电商商城系统数据库）、“hr\_manage”（人力资源管理系统数据库）。
* 避免使用数据库保留字（如 “order”“user”“table” 等），若因业务需求必须使用，需用反引号（`）包裹（如 MySQL）或遵循对应数据库的特殊处理规则。

#### 3.1.2 表命名

* 采用 “业务模块\_表含义” 的格式，全小写字母，单词间用下划线分隔，例如 “user\_info”（用户信息表）、“order\_detail”（订单明细表）、“product\_category”（商品分类表）。
* 临时表命名需以 “tmp\_” 为前缀，后跟业务模块与临时表用途，例如 “tmp\_user\_import”（用户导入临时表），临时表使用后需及时清理，避免占用存储空间。
* 历史表命名需以 “his\_” 为前缀，后跟原表名与时间范围（若有），例如 “his\_order\_info\_2024”（2024 年订单信息历史表）。

#### 3.1.3 字段命名

* 采用 “业务含义” 或 “业务含义\_属性” 的格式，全小写字母，单词间用下划线分隔，例如 “user\_id”（用户 ID）、“order\_amount”（订单金额）、“product\_name”（商品名称）。
* 同一业务含义的字段在不同表中需保持一致，例如用户 ID 在所有表中均命名为 “user\_id”，而非 “uid”“customer\_id” 等。
* 避免使用保留字（如 “name”“time”“status” 虽非严格保留字，但需结合业务场景明确含义，如 “order\_status” 而非单独 “status”），若必须使用，需在设计文档中注明。

#### 3.1.4 索引命名

* 普通索引：以 “idx\_” 为前缀，后跟 “表名\_字段名（多个字段用下划线连接）”，例如 “idx\_user\_info\_mobile”（用户信息表手机号索引）、“idx\_order\_info\_user\_id\_create\_time”（订单信息表用户 ID + 创建时间联合索引）。
* 唯一索引：以 “uk\_” 为前缀，后跟 “表名\_字段名”，例如 “uk\_user\_info\_email”（用户信息表邮箱唯一索引）。
* 主键索引：无需额外命名，数据库默认生成（如 MySQL 的 PRIMARY KEY）。

### 3.2 字段设计规范

#### 3.2.1 数据类型选择

* 遵循 “最小够用” 原则，选择能满足业务需求且存储空间最小的数据类型，减少磁盘占用与 I/O 操作。
  + 整数类型：根据数值范围选择，如用户 ID 用 INT（范围 - 2147483648~2147483647），若数据量超过 INT 范围，用 BIGINT（范围 - 9223372036854775808~9223372036854775807）；避免用 FLOAT/DOUBLE 存储整数（易产生精度问题）。
  + 字符串类型：短字符串（如手机号、身份证号）用 CHAR（固定长度，查询效率高），长字符串（如商品描述、用户备注）用 VARCHAR（可变长度，节省空间）；避免用 TEXT 存储短文本（TEXT 类型查询效率较低）。
  + 时间类型：优先用 DATETIME（存储范围 1000-01-01 00:00:00~9999-12-31 23:59:59，包含日期与时间）或 DATE（仅存储日期），避免用 VARCHAR 存储时间（无法直接进行时间运算，且易出现格式不一致）；若需存储时间戳，用 TIMESTAMP（范围 1970-01-01 00:00:01~2038-01-19 03:14:07）。
  + 小数类型：涉及金额、数量等精确计算场景，用 DECIMAL（如 DECIMAL (10,2) 表示整数位 8 位、小数位 2 位），避免用 FLOAT/DOUBLE（存在精度丢失风险）。

#### 3.2.2 字段长度设置

* 根据业务实际需求设置字段长度，避免过度冗余或长度不足。
  + 手机号：固定 11 位，用 CHAR (11) 或 VARCHAR (11)。
  + 身份证号：18 位（含 X），用 VARCHAR (18)（因 X 为大写字母，需区分大小写场景需设置对应 collation）。
  + 用户名：一般 2-20 个字符，用 VARCHAR (40)（若支持中文，每个中文字符占 2 个字节，需预留足够长度）。
  + 金额：如订单金额，用 DECIMAL (12,2)（支持最大金额 9999999999.99 元，满足多数业务场景）。

#### 3.2.3 字段约束设置

* **NOT NULL 约束**：核心业务字段（如 user\_id、order\_no、create\_time）需设置 NOT NULL，避免空值导致的查询异常与数据不一致；非核心字段（如 user\_avatar、remark）可允许 NULL，但建议设置默认值（如空字符串、0）。
* **默认值设置**：根据业务场景为字段设置合理默认值，例如：
  + 状态字段（如 order\_status）：默认值 0（表示 “待支付”）。
  + 创建时间（create\_time）：默认值 CURRENT\_TIMESTAMP（自动获取当前时间）。
  + 是否删除（is\_delete）：默认值 0（0 表示未删除，1 表示已删除），用于逻辑删除场景。

### 3.3 主键与外键设计

#### 3.3.1 主键设计

* 每个表必须设置主键，确保记录唯一性。
* 优先选择自增整型（如 MySQL 的 INT AUTO\_INCREMENT、Oracle 的 SEQUENCE）作为主键，原因：
  + 自增主键值连续，便于索引维护与查询性能优化。
  + 存储空间小，相比 UUID（字符串类型）更节省磁盘空间。
* 特殊场景（如分布式系统、数据同步场景）可使用 UUID 作为主键，但需注意：
  + UUID 为 36 位字符串，存储空间较大，且无序性可能影响索引性能。
  + 需确保 UUID 生成的唯一性（避免不同节点生成重复 UUID）。
* 禁止使用业务字段（如手机号、订单号）作为主键，避免因业务字段变更（如订单号规则调整）导致主键修改，破坏数据一致性。

#### 3.3.2 外键设计

* 表与表之间存在关联关系时（如订单表与用户表、订单明细表与订单表），需通过外键维护关联，确保数据完整性。
* 外键字段需与关联表的主键字段类型完全一致（如关联表主键为 INT，外键也需为 INT）。
* 外键约束的删除 / 更新策略需根据业务场景设置：
  + 级联删除（ON DELETE CASCADE）：当关联表的主键记录删除时，当前表的外键记录也随之删除（如订单明细表关联订单表，若订单删除，订单明细也需删除，可设置此策略）。
  + 级联更新（ON UPDATE CASCADE）：当关联表的主键值更新时，当前表的外键值也随之更新（较少使用，因主键一般不允许更新）。
  + 禁止删除 / 更新（ON DELETE RESTRICT/ON UPDATE RESTRICT）：若当前表存在外键记录，关联表的主键记录无法删除 / 更新（如用户表与订单表，若用户存在未完成订单，禁止删除用户，可设置此策略）。
* 高并发、高性能要求场景（如秒杀系统订单表），可暂不设置物理外键，改为通过业务代码维护关联关系（减少数据库级别的锁竞争），但需在设计文档中明确说明，并确保代码层面的一致性校验。

## 四、索引设计规范

### 4.1 索引类型选择

* **B 树索引**：默认索引类型，适用于大多数查询场景（如等值查询、范围查询、排序查询），支持多字段联合索引，是数据库索引的核心类型。
* **唯一索引**：用于确保字段值唯一性（如用户邮箱、手机号），同时具备查询加速功能，性能略高于普通 B 树索引（因索引树中无重复值，查询时定位更快）。
* **哈希索引**：适用于等值查询（如 “WHERE user\_id = 123”），查询速度快，但不支持范围查询、排序查询，仅在特定场景（如 Redis 数据库、MySQL 的 Memory 引擎）使用。
* **全文索引**：用于文本内容的模糊查询（如 “WHERE MATCH (product\_desc) AGAINST (' 手机 ')”），适用于长文本字段（如商品描述、文章内容），但性能较低，需结合业务需求谨慎使用（优先考虑 Elasticsearch 等全文检索引擎）。

### 4.2 索引设计原则

#### 4.2.1 索引创建范围

* 优先为高频查询字段（如 WHERE 条件、JOIN 关联、ORDER BY 排序、GROUP BY 分组涉及的字段）创建索引。
* 单表索引数量建议不超过 5 个，过多索引会导致写入操作（INSERT/UPDATE/DELETE）性能下降（每次写入需同步更新所有索引树）。
* 避免为以下字段创建索引：
  + 数据重复率高的字段（如 “性别”“是否删除”，重复率超过 80%，索引过滤效果差，查询时仍需扫描大量数据）。
  + 频繁更新的字段（如 “订单状态”，每次更新需同步更新索引，增加性能开销）。
  + 长度过长的字符串字段（如 “product\_desc”，索引树占用空间大，查询时 I/O 成本高，可考虑前缀索引，如 “idx\_product\_desc (50)”，取前 50 个字符创建索引）。

#### 4.2.2 联合索引设计

* 遵循 “最左前缀原则”，即联合索引的查询效率取决于最左侧字段的使用情况，例如联合索引 “idx\_order\_info (user\_id, create\_time)”，支持 “WHERE user\_id = 123”“WHERE user\_id = 123 AND create\_time > '2024-01-01'” 查询，但不支持 “WHERE create\_time > '2024-01-01'” 查询。
* 联合索引字段顺序需按 “查询频率从高到低、字段区分度从高到低” 排列，例如用户 ID（区分度高、查询频繁）排在前，创建时间（区分度较低、查询频率次之）排在后。
* 避免创建冗余联合索引，例如已创建 “idx\_order\_info (user\_id, create\_time)”，无需再创建 “idx\_order\_info (user\_id)”（前者已包含后者的查询能力）。

#### 4.2.3 覆盖索引设计

* 尽量设计覆盖索引，即查询所需的所有字段（SELECT 子句、WHERE 子句、ORDER BY 子句）均包含在索引中，避免 “回表操作”（查询索引后需再次查询表数据获取其他字段），提升查询效率。
* 示例：若查询 “SELECT order\_no, amount FROM order\_info WHERE user\_id = 123 AND create\_time> '2024-01-01'”，可创建联合索引 “idx\_order\_info (user\_id, create\_time, order\_no, amount)”，查询时直接从索引中获取所有所需字段，无需回表。

### 4.3 索引维护规范

* 定期（如每月）通过数据库工具（如 MySQL 的 EXPLAIN、Oracle 的 EXPLAIN PLAN）分析索引使用情况，删除未使用或使用频率极低的 “冗余索引”“无效索引”。
* 对于数据量较大的表（如超过 1000 万条记录），新增或删除索引需在业务低峰期执行（如凌晨），避免索引操作占用大量资源，影响业务系统运行。
* 索引重建：当表数据大量删除（如删除历史数据）或索引碎片率过高（如 MySQL 的 INFORMATION\_SCHEMA.STATISTICS 表中 DATA\_FREE 字段过大）时，需重建索引（如 MySQL 的 ALTER TABLE ... REBUILD INDEX），优化索引性能。

## 五、数据完整性约束规范

### 5.1 唯一性约束

* 除主键外，需为业务上具有唯一性要求的字段（如用户邮箱、手机号、订单号）设置唯一约束（通过唯一索引实现），防止数据重复录入。
* 若多个字段组合需保持唯一性（如 “用户 ID + 商品 ID” 表示用户对商品的收藏记录，需唯一），需创建联合唯一索引（如 “uk\_collect\_user\_id\_product\_id”）。

### 5.2 检查约束

* 通过 CHECK 约束限制字段值的范围，确保数据符合业务规则，例如：
  + 订单金额（amount）：CHECK (amount> 0)（金额不能为负数）。
  + 用户年龄（age）：CHECK (age BETWEEN 0 AND 150)（年龄范围合理）。
  + 订单状态（order\_status）：CHECK (order\_status IN (0,1,2,3))（状态值仅允许为 “待支付”“已支付”“已发货”“已完成”）。
* 注意：部分数据库（如 MySQL 5.6 及之前版本）对 CHECK 约束支持不完善，需通过业务代码或触发器实现相同逻辑，确保数据完整性。

### 5.3 触发器与存储过程规范

* 触发器用于实现复杂的业务逻辑与数据一致性校验，例如：
  + 当订单表（order\_info）插入记录时，自动更新商品表（product）的库存字段（减少对应商品库存）。
  + 当用户表（user\_info）更新手机号时，自动记录到用户操作历史表（user\_operation\_log）。
* 存储过程用于封装复杂的 SQL 逻辑（如多表关联查询、批量数据处理），提升代码复用性与执行效率

（注：文档部分内容可能由 AI 生成）