**MySQL结构设计规范**

[1. 命名规则 1](#_Toc422996499)

[1.1. 语言 1](#_Toc422996500)

[1.2. 基本规范 2](#_Toc422996501)

[1.3. 对象规范 2](#_Toc422996502)

[2. 基本原则 3](#_Toc422996503)

[3. 库、表规则 4](#_Toc422996504)

[4. 字段规则 4](#_Toc422996505)

[4.1.字段数量 4](#_Toc422996506)

[4.2.字段类型 5](#_Toc422996507)

[4.3.NULL属性 11](#_Toc422996508)

[4.4.类型转换 11](#_Toc422996509)

[5. 索引规则 12](#_Toc422996510)

[5.1.基本原则 12](#_Toc422996511)

[5.2.主键(特指InnoDB) 13](#_Toc422996512)

[5.3.唯一键 14](#_Toc422996513)

[5.4.索引字段 15](#_Toc422996514)

[5.5.其他建议 15](#_Toc422996515)

[6. 字符集 16](#_Toc422996516)

[7. 反范式设计 16](#_Toc422996517)

[7.1.基本原则 16](#_Toc422996518)

[7.2.常见方法 17](#_Toc422996519)

[8. 其他建议 17](#_Toc422996520)

## **命名规则**

### 语言

命名采用英文单词，避免使用拼音，特别不应该使用拼音简写。命名不允许使用中文或者特殊字符。英文单词使用用对象本身意义相对或相近的单词。必要时可采取缩写形式。

### 基本规范

* 大小写

名称一律采取小写形式。

* 单词分隔

命名的各单词之间可以使用下划线进行分隔。

* 保留字

命名不允许使用SQL保留字。

* 命名长度

表名、字段名、视图名长度应限制在64个字符内(含前缀)。

* 注释

表、字段等应该有中文名称注释，以及需要说明的内容。

### 对象规范

* 数据库和表名应尽可能和所服务的模块名一致。服务于同一子模块的一类表尽量以子模块名(或部分单词)为前缀或后缀。
* 表名应尽量包含与所存放数据对应的单词。表名前缀为t\_，数据表名称必须以有特征含义的单词或缩写组成，中间可以用“\_”分割。
* 索引名称尽量包含所有的索引键字段名或缩写，且各字段名在索引名中的顺序应与索引键在索引中的索引顺序一致，并尽量包含一个类似于idx的前缀或后缀。
* 字段名称尽量保持和实际数据相对应。
* 约束等其他对象尽可能包含所属表或其他对象的名称，以表明各自关系。

## **基本原则**

* 不在数据库做运算，CPU计算务必移至业务层。
* 尽可能简单应用MySQL，例如MD5()或者ORDER BY RAND()等操作不要在数据库上执行。
* 平衡范式与冗余：为提高效率牺牲范式设计，冗余数据。
* 拒绝3B：大SQL(BIG SQL)、大事务(BIG Transaction)、大批量(BIG Batch)。
* 防止雪崩效应及过载保护。
* 通过多级缓存减少IO访问。

## **库、表规则**

使用InnoDB表，控制表数量。

* 所有库总空间控制在2T以下。
* 单库不超过5000个表。
* 库\*表总数控制在20000以下。
* 控制单表数据量，单表记录控制在2000万。含文本类型较多的表，应该更少一些。
* 提前估算表预计行数（比如：100万条以上），表预计大小（比如：100M以上）；制定历史数据维护策略（分表策略，迁移策略，清理策略等）。
* 没有任何理由考虑使用MYISAM，请使用INNODB类型。

## **字段规则**

### 4.1.字段数量

控制列数量，字段数控制在40以内。如果都是数字类型，可适当多些。合计单行尽量不超过1000字节。这样做的好处是每条记录所占用的空间量减少，这样每个PAGE保存数据的行数增大。每次IO能读取的数据量增大。同时内存中缓存的单位也是PAGE，同理缓存的数据量也会增大。

### 4.2.字段类型

选择合适的字段类型。MySQL的数据类型可以精确到字段，所以当我们需要大型数据库中存放多字节数据的时候，可以通过对不同表不同字段使用不同的数据类型来较大程度减小数据存储量，进而降低 IO 操作次数并提高缓存命中率。

|  |  |  |
| --- | --- | --- |
| 列类型 | 表达的范围 | 存储需求 |
| TINYINT[(M)] [UNSIGNED] [ZEROFILL] | -128到127 或 0到255 | 1个字节 |
| SMALLINT[(M)] [UNSIGNED] [ZEROFILL] | -32768到32767 或 0到65535 | 2个字节 |
| INT[(M)] [UNSIGNED] [ZEROFILL] | -2147483648到2147483647 或 0到4294967295 | 4个字节 |
| BIGINT[(M)] [UNSIGNED] [ZEROFILL] | -9223372036854775808到9223372036854775807  或 0到18446744073709551615 | 8个字节 |
| DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL] | 整数最大位数（M）为65，小数位数最大（D）为30 | 变长 |
| DATE | YYYY-MM-DD | 3个字节 |
| DATETIME | YYYY-MM-DD HH:MM:SS(1001年到9999年的范围) | 8个字节 |
| TIMESTAMP | YYYY-MM-DD HH:MM:SS（1970年到2037年的范围） | 4个字节 |
| CHAR(M) | 0<M<=255（建议CHAR(1)外，超过此长度的用VARCHAR） | M个字符（所占空间跟字符集等有关系） |
| VARCHAR(M) | 0<M<65532/N | M个字符（N大小由字符集，以及是否为中文还是字母数字等有关系） |
| TEXT | 64K个字符 | 所占空间跟字符集等有关系 |

* 浮点数类型

非万不得已不要使用DOUBLE，不仅仅只是存储长度的问题，同时还会存在精确性的问题。同样，固定精度的小数，也不建议使用DECIMAL，建议乘以固定倍数转换成整数存储，可以大大节省存储空间，且不会带来任何附加维护成本。

存储精确浮点数使用DECIMAL替代FLOAT和DOUBLE。

* 整数类型

对数值类型的不同精度和值域范围，就可以选择TINYINT(1Byte)、SMALLINT(2Byte)、MEDIUMINT(3Byte)、INT(4Byte)、BIGINT(8Byte)。如确定不会使用到负数，建议添加unsigned定义。比如取值范围为0-80时，使用TINYINT UNSIGNED。

自增序列类型的字段只能使用INT或者BIGINT，且明确标识出为无符号型(UNSIGNED)，除非确实会出现负数，仅当该字段数字取值会超过42亿，才使用BIGINT类型。

* 字符类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | GBK | UTF8 | | 数据页字节长 |
| TINYBLOB/BLOB/MEDIUMBLOB/LONGBLOG | (2^8/2^16/2^24/2^32)-1字节 | | | 8098/768 |
| TINYTEXT/TEXT/MEDIUMTEXT/LONGTEXT | ((2^8/2^16/2^24/2^32)-1)/2 | | ((2^8/2^16/2^24/2^32)-1)/3 | 8098/768 |
| VARCHAR | 32767字符(65535/2) | | 21845字符(65535/3) | 8098/768 |
| CHAR | 255字符 | | 255字符 | 全部 |
| BINARY/VARBINARY | 255字节 | | 255字节 | 全部 |

BLOB,TEXT,VARCHAR,CHAR,VARBINARY的存储限制及长度

* 表中数据解读:

以UTF8为例，如果（所有的列的长度相加+列的数目）大于21845,创建表会失改，错误信息为“Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs”。而这种情况下，只有当其中某个字段指定的长度大于21845的时候，这时创建表会成功，但这个字段被自动转换为MEDIUMTEXT类型,并报出warning告知” Converting column 'name' from VARCHAR to TEXT”。

* VARCHAR与CHAR的选择

以UTF8或GBK为例,VARCHAR(200)对比CHAR(200),在存储上InnoDB中已将CHAR(200)字段当成变长字段在处理，两者已经不存在区别。但比如我们存放32长度的md5值，这时候就应该用CHAR(32)，因为虽然在InnoDB存储层面与VARCHAR(32)已经没有区别，但使用CHAR(32)有利于传递出这是一个有规律的字段这样的信息。注意在使用varchar的时候也应该尽量使用符合需求的最小长度，因为mysql会分配固定大小的内存来保存内部值，例如在计算过程中自动转化用到临时表的时候，varchar（200）即使只存储了5个字节也会分配200个字符的空间

* BLOB,TEXT不推荐使用

大字段意味着业务可能需要存放，读取更多的数据。会带来更多的网络消耗。

IO性能：当一个表中某一行，TEXT或BLOB字段有数据存放在BLOB页，那么读取的时候，不仅需要读取数据页，还需要去BLOB页中读取在数据页中这个字段768字节余下的存放在BLOB中的数据。

当大家准备使用TEXT字段类型的时候,首先需要与业务方确认，是否真的需要存放这么长的数据，一般情况下所需要存放的长度远没有我们想像中的长。

* VARCHAR存放多长数据合适

前面已经提到过一行记录的最大长度的阀值是8098。以UTF8为例,则为8098/3=2699。结合表定义中所有字段情况，当一行所有字段长度小于2699的时候最佳，不会有数据存放在BLOB页中。当然，这只是一个参考值。如果能在这个范围内当然最好不过。如果需要大于这个值，以UTF8为例，VARCHAR最长也就20000字符多点，再大就如前面的情况自动转成MEDIUMTEXT了。结合QPS计算出查询的网络消耗，再由DBA综合评估便可。就通常情况来说业务需求不会超过7000字符长度。

UTF8字符集下，VARCHAR类型长度建议2699以下；GBK字符集下，VARCHAR类型长度建议4049以下。

* VARBINARY,BINARY的选择

线上比较典型的应用是使用VARBINARY存放二维码图像二进制信息（这种情况相对比较特殊，线上MySQL不允许用来存放图片、音、视频等）。

* 日期时间类型

根据精度，选择适当的日期时间类型。

尽量使用TIMESTAMP类型，因为其存储空间只需要 DATETIME 类型的一半。对于只需要精确到某一天的数据类型，建议使用DATE类型，因为它的存储空间只需要3个字节，比TIMESTAMP还少。不建议通过INT类型类存储一个unix timestamp 的值，因为这太不直观，会给维护带来不必要的麻烦，同时还不会带来任何好处。

存储年使用YEAR类型。存储日期使用DATE类型。存储时间（精确到秒）使用TIMESTAMP类型或INT。使用时间字段作为查询条件，尤其是以时间段查询时，应使用INT保存。

对于频繁更新且需要根据更新时间查询数据的，建议增加update\_time NOT NULL default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP 以及相应索引

同一张表中只允许一个字段含有current\_timestamp属性

* 枚举类型

不建议使用enum和set类型：

1. 不利于扩展，扩展属于表结构变更,会导致表级别锁
2. 对于弱类型语言例如php，enum('0', '1', '2')这样的设置会导致使用上容易产生歧义
3. 建议使用TINYINT来代替。需在COMMENT信息中标明被枚举的含义。

* 大对象类型

强烈反对在数据库中存放 LOB 类型数据，虽然数据库提供了这样的功能，但这不是他所擅长的，我们更应该让合适的工具做他擅长的事情，才能将其发挥到极致。

如果表中存在类似于 TEXT 或者是很大的 VARCHAR类型的大字段的时候，如果我们大部分访问这张表的时候都不需要这个字段，我们就该义无反顾的将其拆分到另外的独立表中，以减少常用数据所占用的存储空间。这样做的一个明显好处就是每个数据块中可以存储的数据条数可以大大增加，既减少物理 IO 次数，也能大大提高内存中的缓存命中率。

### 4.3.NULL属性

如字段为非空，一定要显式定义NOT NULL属性。原因如下：

* 含有NULL的查询，很难进行查询优化。
* NULL列加索引，需要额外空间。

### 4.4.类型转换

* 字符=>数字

INET\_ATON()

INET\_NTOA()

举例：用无符号INT存储IP，而非CHAR(15)

SELECT INET\_ATON('192.168.0.1');

+--------------------------+

| INET\_ATON('192.168.0.1') |

+--------------------------+

| 3232235521 |

+--------------------------+

SELECT INET\_NTOA('3232235521');

+-------------------------+

| INET\_NTOA('3232235521') |

+-------------------------+

| 192.168.0.1 |

+-------------------------+

* 日期=>数字

FROM\_UNIXTIME()

UNIX\_TIMESTAMP()

## **索引规则**

### 5.1.基本原则

* 控制索引数量。
* 对于区分度不大的字段，不建立索引。
* 不使用外键，通过应用解决。
* 索引中的字段数不超过5个。

### 5.2.主键(特指InnoDB)

* 业务主键vs代理主键

如果使用的是InnoDB并且不需要特殊的聚簇。定义一个代理键（surrogate key）是个好的主意。意思就是这个主键并不是来自于你的应用程序的数据（与业务逻辑无关，而应用程序的数据如果有唯一的候选列可以做成唯一键），最简单的方法就是使用AUTO\_INCREMENT列。这能保证数据插入保持着连续的顺序并且对于使用主键连接会获得更好的性能。最好避免使用随机的聚簇键。 对每张表，最重要的就是一定要有主键。

InnoDB表都设计一个无业务用途的自增列做主键，对于绝大多数场景都是如此，真正纯只读用InnoDB表的并不多，真如此的话还不如用TokuDB来得划算。

* 主键类型

推荐使用自增长列，不建议使用字符串作为主键。

* 主键长度

控制主键长度，因为其他的所有都保存主键，长度过长空间消耗过大。

* 主键生成方法
* UUID

从性能的角度来说，使用UUID是个最不好的方法：它使聚簇索引的插入是随机的。这是最不好的场景了。并且对于数据的聚集也没有什么帮助。

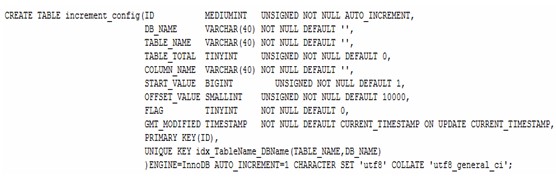
UUID\_SHORT()所占用的存储空间比UUID要小。（UUID\_SHORT()可能要使用bigint占用8个字节，而UUID可能要使用字符串用char(32)）。另外uuid\_short()是顺序的，这个也解决了随机导致的问题，但是uuid\_short()也有一些限制和bug。

* 自增列

简单实用，需要注意GAP LOCK问题，高并发下有一定性能瓶颈。

* 程序控制

避免自增列引起的一些锁问题，统一管理，并发性更高。



* 中间件

更高的并发性，可以考虑从数据库中剥离，使用自己开发或第三方中间件，如：<https://github.com/twitter/snowflake>

### 5.3.唯一键

* 唯一键由3个以下字段组成，并且字段都是整形时，使用唯一键作为主键。
* 没有唯一键或者唯一键不符合条件时，使用自增（或者通过发号器获取）ID作为主键。注意避免多列唯一键作为主键的长度过长问题。
* 唯一键不和主键重复。

### 5.4.索引字段

* 索引字段的顺序需要考虑字段值去重之后的个数，个数多的放在前面。选择率高的字段放在索引前面。
* ORDER BY，GROUP BY，DISTINCT的字段需要添加在索引的后面。这样可以利用索引，避免排序的产生。

### 5.5.其他建议

* 使用EXPLAIN判断SQL语句是否合理使用索引，尽量避免extra列出现：Using File Sort，Using Temporary。
* UPDATE、DELETE语句需要根据WHERE条件添加索引。
* 对长度大于50的VARCHAR字段建立索引时，按需求恰当的使用前缀索引，或使用其他方法。通常取其50%（甚至更小）左右长度创建前缀索引就足以满足80%以上的查询需求了，没必要创建整列的全长度索引。
* 合理创建联合索引（避免冗余），(a,b,c) 相当于 (a) 、(a,b) 、(a,b,c)。
* 合理利用覆盖索引。

## **字符集**

* 字符集直接决定了数据在MySQL中的存储编码方式，由于同样的内容使用不同字符集表示所占用的空间大小会有较大的差异，所以通过使用合适的字符集，可以帮助我们尽可能减少数据量，进而减少IO操作次数。
* 纯拉丁字符能表示的内容，没必要选择 latin1 之外的其他字符编码，因为这会节省大量的存储空间。

如果我们可以确定不需要存放多种语言，就没必要非得使用UTF8或者其他UNICODE字符类型，这回造成大量的存储空间浪费。

## **反范式设计**

### 7.1.基本原则

* 不创建外键，减少多表关联查询。
* 便于分布式设计，允许适度冗余，为了容量扩展允许适度开销。
* 基于业务自由优化，基于I/O或查询设计，无须遵循范式结构设计。

### 6.2.常见方法

* 增加冗余字段

## **其他建议**

* 不建议在MySQL中使用视图、分区表、存储过程触发器等结构，尽量简化MySQL的使用方法。