# 概述

当我们在使用关系型数据库时，主键（Primary Key）是无法避开的概念，主键的作用就是充当记录的标识符，我们能够通过标识符在一张表中定位到唯一的记录。

在关系型数据库中，我们会选择记录中多个字段的最小子集作为该记录在表中的唯一标识符，根据关系型数据库对主键的定义，我们既可以选择单个列作为主键，也可以选择多个列作为主键，但是主键在整个记录中必须存在并且唯一。最常见的方式当然是使用MySQL默认的自增ID作为主键，虽然使用其他策略设置的主键也是合法的，但是不是通用的以及推荐的做法。

# 分类

主键的主要作用是保证表的完整、保证表数据行的唯一性质。

## 自然主键

1、业务主键（自然主键）：在数据库表中把具有业务逻辑含义的字段作为主键，称为“自然主键(Natural Key)”。

自然主键的含义就是原始数据中存在的不重复字段，直接使用成为主键字段。

这种方式对业务的耦合太强，一般不会使用。

## 逻辑主键

2、逻辑主键（代理主键）：在数据库表中采用一个与当前表中逻辑信息无关的字段作为其主键，称为“代理主键”。

逻辑主键提供了一个与当前表数据逻辑无关的字段作为主键，逻辑主键被广泛使用在业务表、数据表，一般有几种生成方式：uuid、自增。其中使用最多的是自增，逻辑主键成功的避免了主键与数据表关联耦合的问题，与业务主键不同的是，业务主键的数据一旦发生更改，那么那个系统中关于主键的所有信息都需要连带修改，这是不可避免的，并且这个更改是随业务需求的增量而不断的增加、膨胀。

而逻辑主键与应用耦合度低，它与数据无任何必要的关系，你可以只关心：第一条数据；而不用关心：名字是a的那条数据。 某一天名字改成b，你还是只关心：第一条数据。

业务的更改几乎是不可避免的，前期任何产品经理言之凿凿的不修改论调都是不可靠、不切实际的。我们必须考虑主键数据在更改的情况下，数据能否平稳度过危机。

## 复合主键

3、复合主键（联合主键）：通过两个或者多个字段的组合作为主键。

复合主键可以说是业务主键的升级版本，通常一个业务字段不能够确定一条数据的唯一性。

例如：张三的身份证是34123322， 张三这种大众名称100%会出现重复。我们可以用姓名+身份证的方式表示主键，声明一个唯一的记录。

有时候，复合主键是复杂的。姓名+身份证不一定能表示不重复，虽然身份证在17年消除了重复的问题，但是之前的数据呢？可能我们需要新增一个地址作为联合主键，例如：姓名+身份证+联系地址确认一个人的身份。在其他的业务中，例如访问控制，用户+终端+终端类型+站点+页面+时间，可能六个字段的联合才能够去确定一个字段的唯一性，这另复杂度陡升。

另外如果其他表要与该表关联则需要引用复合主键的所有字段，这就不单纯是性能问题了，还有存储空间的问题了，当然你也可以认为这是合理的数据冗余，方便查询，但是感觉有点得不偿失。

使用复合主键的原因可能是：对于关系表来说必须关联两个实体表的主键，才能表示它们之间的关系，那么可以把这两个主键联合组成复合主键即可。

如果两个实体存在多个关系，可以再加一个顺序字段联合组成复合主键，但是这样就会引入业务主键的弊端。当然也可以另外对这个关系表添加一个逻辑主键，避免了业务主键的弊端，同时也方便其他表对它的引用。

# UUID主键

## 优点

**与自动递增整数相比，将UUID用作主键的优点：**

1、适合大规模数据。当你把数据分片（例如一组客户数据）存在多个数据库时，使用UUID意味着ID在所有数据分片中都是唯一，而不仅仅是当前那个分片所在数据库。这使得跨数据库移动更为安全。在我的环境，所有数据库分片都可以简单合并到 Hadoop集群中，不会有主键冲突的问题。

2、在插入数据之前就可以知道PK，这避免了查询DB开销，并简化了事务逻辑，比如在使用该键作为其它表外键（FK）时，需要预先获得这个PK。

3、UUID 不会泄露数据信息，因此在URL中暴露会更安全。如果一个用户ID是12345678，很容易猜到还有用户12345677和1234569，这构成了攻击因素。

## 缺点

**使用UUID的缺点：**

**不直观**

很多人直接使用UUID作为字符串，例如 varchar(36) ，请不要这样做！

除了9倍大小的开销（int的大小为4字节），字符串的排序速度也不如数值型字段快，因为它们依赖于排序规则。

在一家最初决定使用Latin-1字符集的公司中，情况就更麻烦。当我们需要将字符集转换为UTF-8时，几个复合键索引就存不下更改后的字符串。

**UUID的痛苦**

不要低估字段太长无法口头表达及记住的痛苦。

**规划扩展计划**

如果我们的目标是扩展，是真正地扩展，那么首先要承认，int 在许多情况下不够大，4字节最大值限制在20亿左右。我们很多数据库都有超过20亿笔交易。

因此在这些情况下需要使用 bigint，它占用8个字节。还可以使用一些其他策略，诸如PostgreSQL和SQL Server等数据库都有本机类型，可以使用16个字节存储。

这时候，谁会在乎它的大小是 bigint 两倍或 int 的四倍呢？毕竟也只是几个字节，对不对？

**规范化数据库中主键的问题**

如果你拥有一个规范的数据库（就像我们目前所在的公司一样），每个主键连接的外键都存在逐渐增加开销。

不仅体现在磁盘上，还包括连接查询和排序过程中，这些外键都需要存在于内存中。尽管内存越来越便宜，但无论是磁盘还是RAM，它们都是有限的资源，也都不是免费的。

我们的数据库有很多中间表，这些中间表是指向其他表外键的容器，尤其是在一对多关系中。帐户具有多个卡号、地址、电话号码，用户名等。对于具有数十亿用户的一组表中的每个列，外键带来额外空间会迅速增加。

**对随机字符串排序真的很麻烦**

另一个问题是碎片化—**由于UUID是随机的，它们没有自然顺序，因此不能用于聚集索引（clustering index）**。这就是为什么SQL Server实现了newsequentialid()，它适合在聚集索引中使用，并且可能是所有UUID主键的一个正确实现。其他数据库PostgreSQL，MySQL也可能有类似的解决方案。

**主键不应该公开，即使是UUID**

根据定义，主键在其范围内是唯一的。因此，用作用户ID或在URL中当做唯一标识是很自然的事情。

但是别这样做！我认为在任何公共场合暴露主键是一个很不好的做法。

前面提到，简单的使用自动递增ID的问题是，它很容易被猜测。攻击爬虫会不断猜测，直到找到一个存在的为止（即使你转向UUID爬虫也可能会尝试，但猜中的概率就变得非常的低）。

尝试猜测一个UUID可能是愚蠢的事，但是微软也警告过，引入 newsequentialid() 来解决聚集索引问题，它使得它的数字更容易被猜测。

注：所以它的优点之一，可以提前知道逐渐的值就不复存在了，因为这样是有风险的。

**主键不需要修改（直到有人这样做）**

有令人信服的理由，不在任何公共语境暴露主键：比如你修改了数据表结构定义，所有外部引用被打破，想想遍地的“404 Page Not Found”页面吧。

何时需要更改主键？碰巧的是，我们本周正在进行数据迁移，谁能在2003年公司成立之初就能预知现在存在13个海量SQL Server数据库并且还在迅速增长？

不要说“从不用修改”，我已经经历过好几次。事先规划简单，但当你数据万亿级别时候，很难修复。

**我的内部系统是对外的**

我管理 Hadoop 集群每晚都会从所有数据库中接收数据。Hadoop系统连接（绑定）到SQL Server数据库，这也正常—毕竟我们是同一家公司。

尽管如此，为了消除多个数据库中冲突的主键，我们通过连接两个字段：用户ID（基于我们设计，它在所有库中是唯一的），再加上业务表本身的ID（存在多库冲突），生成了一个伪主键。

通过这样做，我们在多年的历史用户数据之间建立了紧密而有效的永久绑定。如果RDBMS中的一些主键发生了变化，我们数据也得更改这些主键，否则我们会在前后遇到一些可怕的情况。

## 使用

**两者兼顾：内部整型，外部 UUID？**

简而言之，可采用两者兼顾的方法，我已经在几种不同场景下使用过这种解决方案。

在内部，数据库使用简短高效的连续数字作为主键（int 或 bigint）。

然后添加一个UUID的字段（可以使用插入时的触发器）。在数据库本身的范围内，可以使用常规的主键和外键来管理关系。

当需要将数据的引用公开给外部世界时，即使“外部”对应另外一个内部系统，它们也必须仅依赖UUID。

这样，如果确实必须更改内部主键，则可以确保它仅作用于一个数据库。

我们在另一家公司使用此策略来存储客户数据，只是为了避免“可猜测的”问题。

在另一种情况下，我们将生成一小段文本，从而使 URL 更加人性化。如果有重复项，则只需附加一个哈希值。

即使作为“第二主键”，直接以字符串形式使用UUID也是错误的：我更希望使用内置的数据库机制，将其存储为 8 字节整型数字。

使用整型是因为它们更高效有效。使用 UUID 的数据库实现用于混淆的外部引用。

克里斯·罗素（Chris Russell）正确回应了本文的原始帖子，指出了两个重要的逻辑错误。

首先，即使公开主键的替代品UUID也会暴露信息，这在使用 newsequentialid 函数时尤其如此 — 为了安全，请勿使用UUID。

其次，如果内部的主键是整型值，仍然存在合并两个数据库的键冲突问题，除非所有键都加倍增大。在这种情况下，只使用UUID。

因此，正确的做法是：使用UUID当做主键，并且永远不要公开它们。内外影射的事情留给诸如友好URL处理之类的模块，类似Medium那样使用哈希值。

# 自增列

参考：

<https://www.cnblogs.com/dorothychai/archive/2008/12/19/2268204.html>

<https://www.cnblogs.com/wang666/p/9237583.html>

<https://blog.csdn.net/weixin_39620065/article/details/113464246>

# 原理

参考：<http://mysql.taobao.org/monthly/2021/04/05/>

# 主键不连续

MySQL中默认的AUTO\_INCREMENT属性在多数情况下可以保证主键的连续性，我们通过show create table 命令可以在表的定义中能够看到 AUTO\_INCREMENT属性的当前值，当我们向当前表中插入数据时，它会使用该属性的值作为插入记录的主键，而每次获取该值也都会将它加一。

CREATE TABLE `trades` (  
  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  
 ...  
 `created\_at` timestamp NULL DEFAULT NULL,  
 PRIMARY KEY (`id`),  
 )ENGINE=InnoDB AUTO\_INCREMENT=171 DEFAULT CHARSET=utf8mb4

在很多开发者的认知中，MySQL的主键都应该是单调递增的，但是在我们与MySQL 打交道的过程中会遇到两个问题，首先是记录的主键并不连续，其次是可能会创建多个主键相同的记录，我们将从以下的两个角度回答MySQL不单调和不连续的原因：

1、较早版本的MySQL将 AUTO\_INCREMENT 存储在内存中，实例重启后会根据表中的数据重新设置该值；

2、获取 AUTO\_INCREMENT 时不会使用事务锁，并发的插入事务可能出现部分字段冲突导致插入失败；

## 删除记录

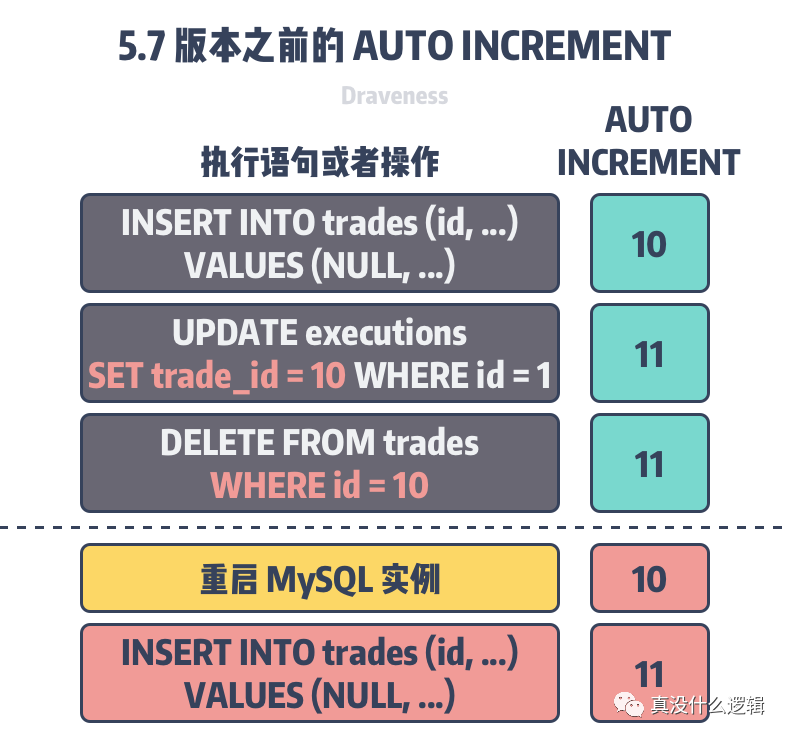
AUTO\_INCREMENT属性虽然在MySQL中十分常见，但是在较早的MySQL版本中，它的实现还比较简陋，InnoDB引擎会在内存中存储一个整数表示下一个被分配到的ID，当客户端向表中插入数据时会获取 AUTO\_INCREMENT 值并将其加一。



因为该值存储在内存中，所以在每次MySQL实例重新启动后，当客户端第一次向table\_name表中插入记录时，MySQL会使用如下所示的SQL语句查找当前表中id 的最大值，将其加一后作为待插入记录的主键，并作为当前表中AUTO\_INCREMENT计数器的初始值。

SELECT MAX(ai\_col) FROM table\_name FOR UPDATE;

如果让作者实现 AUTO\_INCREMENT，在最开始也会使用这种方法。不过这种实现虽然非常简单，但是如果使用者不严格遵循关系型数据库的设计规范，就会出现如下所示的数据不一致的问题：



因为重启了MySQL的实例，所以内存中的 AUTO\_INCREMENT 计数器会被重置成表中的最大值，当我们再向表中插入新的 trades 记录时会重新使用10作为主键，主键也就不是单调的了。在新的trades记录插入之后，executions表中的记录就错误的引用了新的trades，这其实是一个比较严重的错误。

然而这也不完全是MySQL的问题，如果我们严格遵循关系型数据库的设计规范，使用外键处理不同表之间的联系，就可以避免上述问题，因为当前trades记录仍然有外部的引用，所以外键会禁止trades记录的删除，不过多数公司内部的DBA都不推荐或者禁止使用外键，所以确实存在出现这种问题的可能。

然而**在MySQL 8.0中，AUTO\_INCREMENT计数器的初始化行为发生了改变，每次计数器的变化都会写入到系统的重做日志（Redo log）并在每个检查点存储在引擎私有的系统表中**。

In MySQL 8.0, this behavior is changed. The current maximum auto-increment counter value is written to the redo log each time it changes and is saved to an engine-private system table on each checkpoint. These changes make the current maximum auto-increment counter value persistent across server restarts.

当MySQL服务被重启或者处于崩溃恢复时，它可以从持久化的检查点和重做日志中恢复出最新的AUTO\_INCREMENT计数器，避免出现不单调的主键也解决了这里提到的问题。

## 并发事务

为了提高事务的吞吐量，MySQL可以处理并发执行的多个事务，但是如果并发执行多个插入新记录的SQL语句，可能会导致主键的不连续。如下图所示，事务1向数据库中插入id = 10 的记录，事务2向数据库中插入id = 11和id = 12的两条记录：



不过如果在最后事务1由于插入的记录发生了唯一键冲突导致了回滚，而事务2没有发生错误而正常提交，在这时我们会发现当前表中的主键出现了不连续的现象，后续新插入的数据也不再会使用 10 作为记录的主键。



这个现象背后的原因也很简单，**虽然在获取 AUTO\_INCREMENT 时会加锁，但是该锁是语句锁，它的目的是保证 AUTO\_INCREMENT 的获取不会导致线程竞争，而不是保证MySQL中主键的连续**。

上述行为是由InnoDB存储引擎提供的 innodb\_autoinc\_lock\_mode 配置控制的，该配置决定了获取 AUTO\_INCREMENT 计时器时需要先得到的锁，该配置存在三种不同的模式，分别是传统模式（Traditional）、连续模式（Consecutive）和交叉模式（Interleaved），其中MySQL使用连续模式作为默认的锁模式：

**传统模式 innodb\_autoinc\_lock\_mode = 0；**

在包含 AUTO\_INCREMENT 属性的表中插入数据时，所有的 INSERT 语句都会获取表级别的 AUTO\_INCREMENT 锁，该锁会在当前语句执行后释放；

**连续模式 innodb\_autoinc\_lock\_mode = 1；**

INSERT ... SELECT、REPLACE ... SELECT  以及 LOAD DATA 等批量的插入操作需要获取表级别的 AUTO\_INCREMENT 锁，该锁会在当前语句执行后释放；

简单的插入语句（预先知道插入多少条记录的语句）只需要获取获取 AUTO\_INCREMENT 计数器的互斥锁并在获取主键后直接释放，不需要等待当前语句执行完成；

**交叉模式 innodb\_autoinc\_lock\_mode = 2；**

所有的插入语句都不需要获取表级别的 AUTO\_INCREMENT 锁，但是当多个语句插入的数据行数不确定时，可能存在分配相同主键的风险；

这三种模式都不能解决MySQL自增主键不连续的问题，**想要解决这个问题的终极方案是串行执行所有包含插入操作的事务，也就是使用数据库的最高隔离级别—可串行化（Serialiable）**。当然直接修改数据库的隔离级别相对来说有些简单粗暴，基于MySQL或者其他存储系统实现完全串行的插入也可以保证主键在插入时的连续，但是仍然不能避免删除数据导致的不连续。

## 总结

早期MySQL的主键既不是单调的，也不是连续的，这些都是在当时工程上做出的一些选择，如果严格地按照关系型数据库的设计规范，MySQL最初的设计造成问题的概率也比较低，只有当被删除的主键被外部系统引用时才会影响数据的一致性，但是今天使用方式的不同却增加出错的可能性，而MySQL也在8.0中持久化了AUTO\_INCREMENT 以避免该问题的出现。

MySQL中不连续的主键又是一个工程设计向性能低头的例子，牺牲主键的连续性来支持数据的并发插入，最终提高了MySQL服务的吞吐量。

MySQL 5.7 版本之前在内存中存储 AUTO\_INCREMENT 计数器，实例重启后会根据表中的数据重新设置，在删除记录后重启就可能出现重复的主键，该问题**在8.0版本使用重做日志解决，保证了主键的单调性**；

MySQL插入数据获取 AUTO\_INCREMENT 时不会使用事务锁，而是会使用互斥锁，并发的插入事务可能出现部分字段冲突导致插入失败，想要保证主键的连续需要串行地执行插入语句。

## 分布式数据库实践

对于分布式数据库改造过程中，对于没有确定主键的表（MySQL是允许不指定主键的，分布式数据库需要指定主键）改造方案是增加一个自增列gdb\_id，但是这种存在一个风险就是并发插入的时候可能会导致自增列的不连续（业务侧不使用这个自增列字段所以不会产生问题，但是如果使用这个字段则会有问题）。

注：我们的分布式数据库没有采用单机MySQL的自增列机制，所有的自增列都是由统一的全局控制模块申请的，能够保证全局的唯一性。

# 使用建议