# 数据完整性

关系型数据库系统和文件系统的一个不同点是，关系型数据库本身能保证存储数据的完整性，不需要应用程序的控制，而文件系统一般需要在程序端进行控制。当前几乎所有的关系型数据库都提供了约束机制，该机制提供了一条强大而简易的途径来保证数据库中数据的完整性。

一般来说，数据完整性有以下三种形式：

1、实体完整性保证表中有一个主键。在InnoDB存储引擎表中，用户可以通过定义Primary Key或Unique Key约束来保证实体的完整性。用户还可以通过编写一个触发器来保证数据完整性。

2、域完整性保证数据每列的值满足特定的条件。在InnoDB存储引擎表中，域完整性可以通过以下几种途径来保证：

选择合适的数据类型确保一个数据值满足特定条件

外键约束

编写触发器

还可考虑用DEFAULT约束作为强制域完整性的一个方面

3、参照完整性保证两张表之间的关系。InnoDB存储引擎支持外键，因此允许用户定义外键以强制参照完整性，也可以通过编写触发器以强制执行。

# 概述

约束保证数据的完整性和一致性。

## 约束和索引

当用户创建了一个唯一索引就创建了一个唯一的约束。但是约束和索引还是有所不同的，约束更是一个逻辑的概念，用来保证数据的完整性，而索引是一个数据结构，既有逻辑上的概念，在数据库中还代表着物理存储的方式。

## ENUM和SET约束

## 触发器与约束

## 8.0新特性

### 检查约束

参考：<https://mp.weixin.qq.com/s/9nEDzURmOh5OytCssGO7sA>

检查约束就是在INSERT或UPDATE操作之前，会根据指定条件CHECK要INSERT或UPDATE的字段值是否满足约束。

MySQL在8.0.16之后支持check constraint作为新特性，语法为：

CREATE TABLE t1(id int,  name varchar(10), age int CHECK(age>=18));

在早期版本中，该语法依旧支持，但不会起作用，也就是会被解析，但不会被存储层引用。

除了采用建表语句增加CHECK约束，也可以对已有的表增加CHECK约束：

ALTER TABLE t1 ADD CONSTRAINT AGE\_CHECK CHECK(age>=18)

# 分类

约束分为表级约束和列级约束：

对一个数据列建立的约束，称为列级约束。

对多个数据列建立的约束，称为表级约束。

列级约束既可以在列定义时声明，也可以在列定义后声明。

表级约束只能在列定义后声明。

对于InnoDB存储引擎而言，提供了以下几种约束：

Primary Key（主键约束）

Unique Key（唯一约束）

Foreign Key（外键约束）

Default（默认约束）

NOT NULL（非空约束）

## 主键约束

每张数据表只能存在一个主键

主键保证记录的唯一性

主键自动为NOT NULL

UUID（类似于 GUID）用作数据库表的主键（PK）。

<https://mp.weixin.qq.com/s/ZHBPgS6BsOJ5F0slHJ5I3w>

**使用 UUID 的优点：**

与自动递增整数相比，将 UUID 用作主键的优点很多：

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。（请参见下面的更好的选择）。

使用 UUID 的缺点

1、不直观

很多人直接使用 UUID （类似像70E2E8DE-500E-4630-B3CB-166131D35C21 ）作为字符串，例如 varchar(36) — 请不要这样做！

2、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。  
(见 https://msdn.microsoft.com/en-us/library/ms189786.aspx )

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。

**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。

## 唯一约束

唯一约束可以保证记录的唯一性

唯一约束的字段可以为空值（NULL）

每张数据表可以存在多个唯一约束

注：唯一约束可以存在多个，主键只能是一个，且不为NULL

## 外键约束

### 背景

**如果不使用外检会有什么问题？**

1、潜在的数据完整性问题

缺少外键明显问题是数据库不能强制进行引用完整性检查，如果在高一层没有正确处理，则可能会导致数据不一致（子行没有相应父行）。

2、表格关系不清晰

数据库中缺少外键的另一个不太明显的负面影响是，不了解该模式的人很难找到正确的表并找出表关系。这可能会导致严重的数据库查询和报告问题。

### 概述

在关系型数据库中，外键也被称为**关系键**，它是关系型数据库中提供关系表之间连接的多个列，这一组数据列是当前关系表中的外键，也必须是另一个关系表中的**候选键（Candidate Key）**，我们可以通过候选键在当前表中找到唯一的元素。在通常情况下，我们都会使用关系表中的主键作为其他表中的外键，这样才可以满足关系型数据库对外键的约束。

外键不仅仅是数据库表中的一个整数，它还提供了额外的一致性保证。因为数据库往往是整个系统的真理之源（Source of Truth），所以保证数据的一致性和正确性非常重要，关系型数据库虽然提供了外键、触发器等特性保证一致性，但是在今天的生产环境中却很少被使用。

引用完整性（Referential Integrity）是数据的属性，如果数据拥有该属性，那么数据中所有的引用都是合法的，在关系型数据库的上下文中，这就意味着关系型数据库中引用另一个表中的值必须存在。

ALTER TABLE posts

ADD CONSTRAINT FOREIGN KEY (author\_id)

REFERENCES authors(id);

上述SQL语句可以向关系表中增加外键约束，该SQL语句的执行前提是posts表中存在author\_id字段。从SQL语句中的CONSTRAINT关键字我们也能推测出外键不是一种数据类型，它是不同关系表之间的约束。

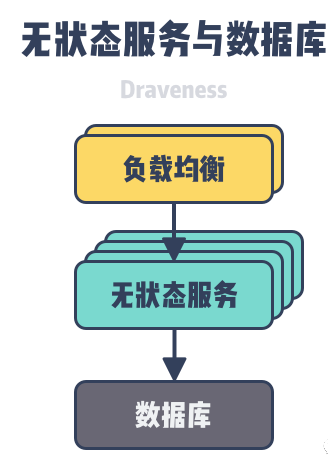


图2 - 无状态服务与数据库

不使用外键的原因其实很简单，MySQL、PostgreSQL等关系型数据库很难水平扩容，但是无状态的服务往往都可以很容易地扩容。**由于外键等特性需要数据库执行额外的工作，而这些操作会占用数据库的计算资源，所以我们可以将大部分的需求都迁移到无状态的服务中完成以降低数据库的工作负载**。

#### 前提

外键约束保持数据一致性，完整性，实现一对一或者一对多关系。

**外键约束的要求：**

1. 父表和子表必须使用相同的存储引擎，而且禁止使用临时表；
2. 数据表的存储引擎只能为InnoDB；
3. 外键列和参照列必须具有相似的数据类型，其中数字的长度或是否有符号位必须相同，而字符的长度则可以不同；
4. 外键列和参照列必须创建索引，如果外键列不存在索引的话，MySQL将自动创建索引。

建立外键的前提：  
 两个表必须是InnoDB表类型。  
 使用在外键关系的域必须为索引型(Index)。  
 使用在外键关系的域必须与数据类型相似。

#### 作用

可以使得两张表关联，保证数据的一致性和实现一些级联操作。

保持数据一致性，完整性，主要目的是控制存储在外键表中的数据。使两张表形成关联，外键只能引用外表中的列的值！

#### 特点

##### 优点

保持数据一致性，完整性，主要目的是控制存储在外键表中的数据。 使两张表形成关联，外键只能引用外表中的列的值！

##### 缺点

在对MySQL做优化的时候类似查询缓存，索引缓存之类的优化对InnoDB类型的表是不起作用的，还有在数据库整体架构中用的同步复制也是对InnoDB类型的表不生效的，像数据库中核心的表类似商品表请大家尽量不要是使用外键，如果同步肯定要同步商品库的，加上了外键也就没法通不了，优化也对它没作用，岂不得不偿失，做外键的目的在于保证数据完整性。因此，请大家最好不要随便用外键，而是通过程序来实现这个目的，切记！  
 外键所产生的性能问题：  
 1、数据库需要维护外键的内部管理；

1. 外键等于把数据的一致性事务实现，全部交给数据库服务器完成；
2. 有了外键，当做一些涉及外键字段的增，删，更新操作之后，需要触发相关操作去检查，而不得不消耗资源；
3. 外键还会因为需要请求对其他表内部加锁而容易出现死锁情况；

### 分类

根据更新和删除时的行为不同，我们可以将外键分成RESTRICT、CASCADE和SET NULL等几种，当我们为关系表中的字段增加外键约束时，需要指定外键的类型，最常见的也就是RESTRICT和CASCADE两种，其中RESTRICT为外键的默认类型，不同类型的外键会带来不同的额外开销，而这些额外开销就是我们不使用外键的理由：

● 使用RESTRICT会在更新或者删除记录时对外键对应的记录是否存在进行一致性检查；

● 使用CASCADE会在更新或者删除记录时触发级联更新或者删除操作；

注意：MySQL中的NO ACTION和RESTRICT具有相同的语义。

**外键约束的参照操作：**

1. CASCARD：从父表删除或更新且自动删除或更新子表中匹配的行
2. SET NULL：从父表删除或更新行，并设置子表中的外键列为NULL，如果使用该项，必须保证子表列没有指定NOT NULL；
3. RESTRICT：拒绝对父表的删除或更新操作；
4. NOT ACTION：标准SQL的关键字，在MySQL中与RESTRICT相同

#### 一致性检查

当我们使用默认的外键类型RESTRICT时，在创建、修改或者删除记录时都会检查引用的合法性。想要在MySQL等数据库中触发外键的一致性检查其实非常容易，假设我们的数据库中包含posts(id, author\_id, content)和authors(id, name)两张表，在执行如下所示的操作时都会触发数据库对外键的检查：

● 向posts表中插入数据时，检查author\_id是否在authors表中存在；

● 修改posts表中的数据时，检查author\_id是否在authors表中存在；

● 删除authors表中的数据时，检查posts中是否存在引用当前记录的外键；

作为专门用于管理数据的系统，数据库与应用服务相比能够更好地保证完整性，而上述的这些操作都是引入外键带来的额外工作，不过这也是数据库保证数据完整性的必要代价。上述的这些分析都是理论上的定性分析，我们其实可以简单的定量分析一下引入外键对性能的影响。

在这里我们在数据库中同时创建authors、posts和foreign\_key\_posts三种表，如下所示，其中posts和foreign\_key\_posts两个表中的列完全相同，只是foreign\_key\_posts表为author\_id字段增加了RESTRICT类型的外键约束：



我们先在authors表中插入一条记录，随后分别在posts和foreign\_key\_posts中插入多条新数据列引用该条记录，前者不会检查外键的合法性，而后者会做额外的检查。你可以在 这里 找到作者用来测试外键额外开销的Go语言代码，经过多次基准测试，我们可以得到如下所示的结果：

BenchmarkBaseline-8 3770 309503 ns/op

BenchmarkForeignKey-8 3331 317162 ns/op

BenchmarkBaseline-8 3192 315506 ns/op

BenchmarkForeignKey-8 3381 315577 ns/op

BenchmarkBaseline-8 3298 312761 ns/op

BenchmarkForeignKey-8 3829 345342 ns/op

BenchmarkBaseline-8 3753 291642 ns/op

BenchmarkForeignKey-8 3948 325239 ns/op

作者执行了4次外键的基准测试，虽然4次测试的结果不是特别稳定，但是使用外键的用例在每次测试中都明显弱于不使用外键的用例，外键带来的额外开销分别为~2.47%、~0.02%、~10.41%和~11.52%。这里的基准测试只是一个比较简单的定量分析，但是我们也可以从结果中看到大概的趋势—外键的完整性检查确实会带来额外的性能开销，而这些开销在高并发的服务中需要慎重考虑。

想要在应用程序中模拟数据库外键的功能其实比较容易，我们只需要遵循以下的几个准则：

● 向表中插入数据或者修改表中的数据时，都应该执行额外的SELECT语句确保它引用的数据在数据库中存在；

● 在删除数据之前需要执行额外的SELECT语句检查是否存在当前记录的引用；

需要注意的是为了保证一致性，我们需要在事务中执行上述的查询和修改语句，这样才能完整模拟外键的功能；当我们向posts表中插入或者修改数据时，需要的处理相对比较简单，我们只需要执行有限的SELECT语句并按照如下所示的模式执行对应的操作就可以了：

BEGIN

SELECT \* FROM authors WHERE id = <post.author\_id> FOR UPDATE;

-- INSERT INTO posts ... / UPDATE posts ...

END

但是如果我们要删除authors表中的数据，就需要查询所有引用authors数据的表；如果有10个表都有指向authors表的外键，我们就需要在10个表中查询是否存在对应的记录，这个过程相对比较麻烦，不过也是为了实现完整性的必要代价，不过这种模拟外键方法其实远比使用外键更消耗资源，它不仅需要查询关联数据，还要通过网络发送更多的数据包。

#### 级联操作

当我们在关系型数据库中创建外键约束时，如果使用如下所示的SQL语句指定更新或者删除记录时使用CASCADE行为，那么在客户端更新或者删除数据时就会触发级联操作：

ALTER TABLE posts

ADD CONSTRAINT FOREIGN KEY (author\_id)

REFERENCES authors(id)

ON UPDATE CASCADE

ON DELETE CASCADE;

当客户端更新 authors 表中记录的主键时，数据库会同时更新 posts 表中所有引用该记录的外键；

当客户端删除 authors 表中的记录时，数据库会删除所有与 authors 表关联的记录；

不过无论是执行更新还是删除操作，数据库都可以保证各个关系表之间引用的一致性和合法性不会出现引用到不存在记录的情况，与 RESTRICT 行为一样，所有外键的更新和删除行为都可以通过执行额外的检查和操作保证数据的一致。

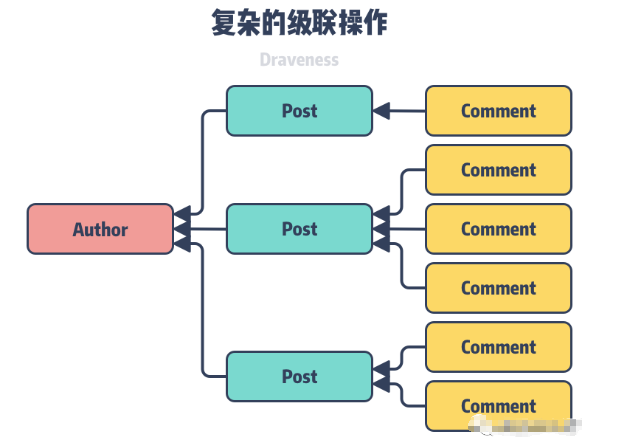


图 4 - 复杂的级联操作

虽然级联删除的出发点也是保证数据的完整性，但是在设计关系表之间的不同关系时，我们也需要注意级联删除引起的数据大规模删除的问题。如上图所示，当客户端想要在数据库中删除 authos 表中的数据时，如果我们同时在 authors 和 posts 中指定了级联删除的行为，那么数据库会同时删除所有关联的 posts 记录以及与 posts 表关联的 comments 数据。

这种涉及多级的级联删除行为在数据量较小的数据库中不会导致问题，但是在数据量较大的数据库中删除关键数据可能会引起雪崩，一条记录的删除可能会被放大到几十倍甚至上百倍，这些对磁盘的随机读写会带来巨大的开销，是我们想要尽可能避免的情况。如果我们能够较好地设计各个表之间的关系并且慎用 CASCADE 行为，这对于保证数据库中数据的合法性有着很重要的意义，使用该特性可以避免数据库中出现过期的、不合法的数据，但是在使用时也要合理预估可能造成的最坏情况。

手动实现数据库的级联删除操作是可行的，如果我们在一个事务中按照顺序删除所有的数据，确实可以保证数据的一致性，但是这与外键的级联删除功能没有太大的区别，反而会有更差的表现。如果我们能够接受在一个时间窗口内的数据不一致，就可以将一个大号的删除任务拆成多个子任务分批执行，降低对数据库影响的峰值。

DELETE FROM posts WHERE author\_id = 1 LIMIT 100;

DELETE FROM posts WHERE author\_id = 1 LIMIT 100;

...

DELETE FROM authors WHERE id = 1;

与数据库外键的 CASCADE 相比，这种方式会带来更大的额外开销，只是我们能降低对数据库性能的瞬时影响。

### 操作

#### 创建

指定主键关键字：foreign key(列名)  
 引用外键关键字：references <外键表名>(外键列名)

**示例：**

ALTER TABLE course ADD CONSTRAINT FK\_course\_teacher FOREIGN KEY(Tid) REFERENCES teacher(Tid);

ALTER TABLE sc ADD CONSTRAINT FK\_sc\_student FOREIGN KEY(S) REFERENCES student(S);

ALTER TABLE sc ADD CONSTRAINT FK\_sc\_course FOREIGN KEY(C) REFERENCES course(Cid);

事件触发限制：on delete和on update，可设参数cascade(跟随外键改动)，restrict(限制外表中的外键改动)，set Null(子表设空），set Default（设默认值），no action[默认]。  
     mysql> create table temp(  
 id int,  
 name char(20),  
 foreign key(id) references outTable(id) on delete cascade on update cascade);     
 说明：把id列 设为外键，参照外表outTable的id列，当外键的值删除，本表中对应的纪录删除，当外键的值改变，本表中对应的纪录改变。

#### 删除

**语法：**

ALTER TABLE ... DROP FOREIGN KEY ...;

**示例：**

ALTER TABLE course DROP FOREIGN KEY tid

### 查询

参考：

<http://www.manongjc.com/article/35162.html>

SELECT C.TABLE\_SCHEMA 拥有者,

C.REFERENCED\_TABLE\_NAME 父表名称,

C.REFERENCED\_COLUMN\_NAME 父表字段,

C.TABLE\_NAME 子表名称,

C.COLUMN\_NAME 子表字段,

C.CONSTRAINT\_NAME 约束名,

T.TABLE\_COMMENT 表注释,

R.UPDATE\_RULE 约束更新规则,

R.DELETE\_RULE 约束删除规则

FROM INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE C

JOIN INFORMATION\_SCHEMA.TABLES T

ON T.TABLE\_NAME = C.TABLE\_NAME

JOIN INFORMATION\_SCHEMA.REFERENTIAL\_CONSTRAINTS R

ON R.TABLE\_NAME = C.TABLE\_NAME

AND R.CONSTRAINT\_NAME = C.CONSTRAINT\_NAME

AND R.REFERENCED\_TABLE\_NAME= C.REFERENCED\_TABLE\_NAME

WHERE C.REFERENCED\_TABLE\_NAME IS NOT NULL ;

### 使用建议

参考：

<https://mp.weixin.qq.com/s/bxdBfAdV30EGEWkw7Q4vPA>

**不推荐使用外键：**

1、潜在的数据完整性问题

缺少外键明显问题是数据库不能强制进行引用完整性检查，如果在高一层没有正确处理，则可能会导致数据不一致（子行没有相应父行）。

2、表格关系不清晰

数据库中缺少外键的另一个不太明显的负面影响是，不了解该模式的人很难找到正确的表并找出表关系。这可能会导致严重的数据库查询和报告问题。

**为什么数据库可以没有外键？**

1、性能

在表上拥有活动的外键可以提高数据质量，但会影响插入、更新和删除操作的性能。**在这些任务之前，数据库需要检查它是否违反数据完整性**。这就是为什么一些架构师和DBA完全放弃外键的原因。数据仓库和分析数据库尤其如此，这些数据仓库和分析数据库不以交易方式（一次一行）处理数据，而是批量处理数据。性能是数据仓库和商业智能的一切。

2、传统数据

许多数据库在设计时需要存储来自旧数据库和遗留数据，这些数据可能对数据质量和完整性没有那么严格。为了能够容纳旧的脏数据，架构师可以选择

1. 清理和转换遗留数据（昂贵的练习）
2. 放弃在数据库级别上强制执行参照完整性。一些打包的ERP和CRM应用程序也使用这种方法。

3、全表重新加载

一些数据库，如数据仓库，分段或接口数据库，需要经常从外部重新加载数据。这会导致重新加载时数据不一致（在父表为空的情况下，子表可能已满载）。这可以通过在重新加载时禁用外键来绕过。然而，这引入了额外的逻辑和复杂性以及另一个失败点。如上所述，对性能有负面影响。通常，成本大于收益，开发人员不用担心外键。

4、更高层次的框架

一些应用程序使用编程框架，在物理数据库之上创建另一个逻辑层。开发人员不使用插入或更新语句来修改数据，而使用API或者框架在后台执行所有操作。ORM（对象关系映射）框架或Ruby on Rails框架就是这种情况。这些工具负责参照完整性，并与RDBMS一起创建更高级别的数据库引擎。这些框架可以自己创建数据库表，而不总是创建外键。使用这些工具的开发人员很少会干扰自动生成的模式，并且不需要外键。

5、跨数据库关系

这可能不是数据库没有外键的正确理由，一些数据库跨越更多的物理数据库甚至引擎，并且在技术上可能不能创建跨越数据库的它不能在同一台服务器上的两个数据库上创建key。SQL Server就是一个很好的例子 - 它不能在同一台服务器上的两个数据库上创建key。而且这种架构在大型系统中很常见。

6、数据库平台不可知论者

类似于前一个，一些应用程序被设计为数据库平台（DBMS）不可知的，并能够在Oracle，SQL Server，DB/2或Sybase等各种数据库上工作。设计人员不想绑定到任何特定的平台，并将所有逻辑推送到应用程序层，尽可能清楚地离开数据库层。

7、对更改开放

Oracle电子商务套件-就是它被设计成尽可能定制。Oracle提供了坚实的基础，使实施团队具有弹性，可以尽可能多地决定设计。至少这是他们所说的。也许这个原因和以前一样，或者是下一个原因：

8、懒惰的架构师

在创建数据库时，如果要存储数据，则需要创建一些表和列。这是最低限度。但是，您不必创建保持数据一致性的结构，如主键，唯一键，外键或约束。这需要一些努力，但是却没有带来直接的好处。一些架构师和数据库管理员只是忽略了这一部分。

9、保持模型的秘密

也许这是一个很遥远的问题，但也许有时候是因为人们不希望别人知道太多太容易。一般来说，人们希望被需要和不可替代。

### 总结

外键提供的几种在更新和删除时的不同行为都可以帮助我们保证数据库中数据的一致性和引用合法性，但是外键的使用也需要数据库承担额外的开销，在大多数服务都可以水平扩容的今天，高并发场景中使用外键确实会影响服务的吞吐量上限。在数据库之外手动实现外键的功能是可能的，但是却会带来很多维护上的成本或者需要我们在数据一致性上做出一些妥协。我们可以从可用性、一致性几个方面分析使用外键、模拟外键以及不使用外键的差异：

不使用外键牺牲了数据库中数据的一致性，但是却能够减少数据库的负载；

模拟外键将一部分工作移到了数据库之外，我们可能需要放弃一部分一致性以获得更高的可用性，但是为了这部分可用性，我们会付出更多的研发与维护成本，也增加了与数据库之间的网络通信次数；

使用外键保证了数据库中数据的一致性，也将全部的计算任务全部交给了数据库；

在大多数不需要高并发或者对一致性有较强要求的系统中，我们可以直接使用数据库提供的外键帮助我们对数据进行校验，但是在对一致性要求不高的、复杂的场景或者大规模的团队中，不使用外键也确实可以为数据库减负，而大团队也有更多的时间和精力去设计其他的方案，例如：分布式的关系型数据库。

当我们考虑应不应该在数据库中使用外键时，需要关注的核心我们的数据库承担这部分计算任务后会不会影响系统的可用性，在使用时也不应该一刀切的决定用或者不用外键，应该根据具体的场景做决策，我们在这里介绍了两个使用外键时可能遇到的问题：

RESTRICT外键会在更新和删除关系表中的数据时对外键约束的合法性进行检查，保证外键不会引用到不存在的记录；

CASCADE外键会在更新和删除关系表中的数据时触发对关联记录的更新和删除，在数据量较大的数据库中可能会有数量级的放大效果；

我们在很多时候其实并不能选择是否使用外键，大多数公司的 DBA 都会对数据库系统的使用有比较明确的规定，但是我们要清楚做出使用外键和不使用外键这一抉择的原因。

# 操作

## 检查约束

## 添加约束

添加主键约束：

ALTER TABLE tb\_name ADD [CONSTRAINT [symbol]] PRIMARY KEY [index\_type](index\_col\_name,…)

添加唯一约束：

ALTER TABLE tb\_name ADD [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY] [index\_name] [index\_type](index\_col\_name,…)

添加外键约束：

ALTER TABLE tb\_name ADD [CONSTRAINT [symbol]] FOREIGN KEY [index\_name] (index\_col\_name,…) reference\_definition

添加/删除默认约束：

ALTER TABLE tb\_name ALTER [COLUMN] col\_name {SETDEFAULT literal | DROP DEFAULT}

## 删除约束

删除主键约束：

ALTER RABLE tb\_name DROP PRIMARY KEY

删除唯一约束：

ALTER RABLE tb\_name DROP {INDEX|KEY} index\_name

删除外键约束：

ALTER RABLE tb\_name DROP FOREIGN KEY fk\_symbol