

45 | 自增id用完怎么办？

2019-02-25 林晓斌



MySQL里有很多自增的id，每个自增id都是定义了初始值，然后不停地往上加步长。虽然自然数是没有上限的，但是在计算机里，只要定义了表示这个数的字节长度，那它就有上限。比如，无符号整型(unsigned int)是4个字节，上限就是 $2^{32}-1$ 。

既然自增id有上限，就有可能被用完。但是，自增id用完了会怎么样呢？

今天这篇文章，我们就来看看MySQL里面的几种自增id，一起分析一下它们的值达到上限以后，会出现什么情况。

表定义自增值id

说到自增id，你第一个想到的应该就是表结构定义里的自增字段，也就是我在第39篇文章[《自增主键为什么不是连续的？》](#)中和你介绍过的自增主键id。

表定义的自增值达到上限后的逻辑是：再申请下一个id时，得到的值保持不变。

我们可以通过下面这个语句序列验证一下：

```
create table t(id int unsigned auto_increment primary key) auto_increment=4294967295;
insert into t values(null);
//成功插入一行 4294967295
show create table t;
/* CREATE TABLE `t` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4294967295;
*/

insert into t values(null);
//Duplicate entry '4294967295' for key 'PRIMARY'
```

可以看到，第一个insert语句插入数据成功后，这个表的AUTO_INCREMENT没有改变（还是4294967295），就导致了第二个insert语句又拿到相同的自增id值，再试图执行插入语句，报主键冲突错误。

$2^{32}-1$ （4294967295）不是一个特别大的数，对于一个频繁插入删除数据的表来说，是可能会被用完的。因此在建表的时候你需要考察你的表是否有可能达到这个上限，如果有可能，就应该创建成8个字节的bigint unsigned。

InnoDB系统自增row_id

如果你创建的InnoDB表没有指定主键，那么InnoDB会给你创建一个不可见的，长度为6个字节的row_id。InnoDB维护了一个全局的dict_sys.row_id值，所有无主键的InnoDB表，每插入一行数据，都将当前的dict_sys.row_id值作为要插入数据的row_id，然后把dict_sys.row_id的值加1。

实际上，在代码实现时row_id是一个长度为8字节的无符号长整型(bigint unsigned)。但是，InnoDB在设计时，给row_id留的只是6个字节的长度，这样写到数据表中时只放了最后6个字节，所以row_id能写到数据表中的值，就有两个特征：

1. row_id写入表中的值范围，是从0到 $2^{48}-1$ ；
2. 当dict_sys.row_id= 2^{48} 时，如果再有插入数据的行为要来申请row_id，拿到以后再取最后6个字节的话就是0。

也就是说，写入表的row_id是从0开始到 $2^{48}-1$ 。达到上限后，下一个值就是0，然后继续循环。

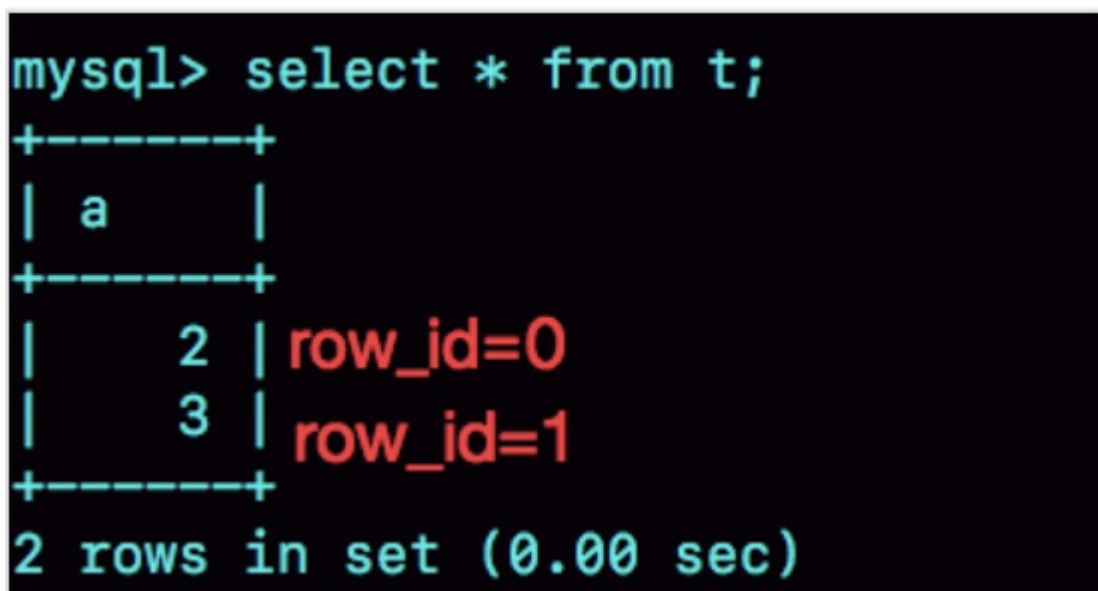
当然， $2^{48}-1$ 这个值本身已经很大了，但是如果一个MySQL实例跑得足够久的话，还是可能达到这个上限的。在InnoDB逻辑里，申请到row_id=N后，就将这行数据写入表中；如果表中已经存

在row_id=N的行，新写入的行就会覆盖原有的行。

要验证这个结论的话，你可以通过gdb修改系统的自增row_id来实现。注意，用gdb改变量这个操作是为了便于我们复现问题，只能在测试环境使用。

```
mysql> create table table t(a int)engine=innodb;
gdb -p <pid of mysqld> -ex 'p dict_sys.row_id=1' --batch
mysql> inset into t values(1);
gdb -p <pid.mysqld> -ex 'p dict_sys.row_id=281474976710656' --batch
mysql> inset into t values(2);
mysql> inset into t values(3);
mysql> select * from t;
```

图1 row_id用完的验证序列



```
mysql> select * from t;
+-----+
| a     |
+-----+
|     2 | row_id=0
|     3 | row_id=1
+-----+
2 rows in set (0.00 sec)
```

图2 row_id用完的效果验证

可以看到，在我用gdb将dict_sys.row_id设置为 2^{48} 之后，再插入的a=2的行会出现在表t的第一行，因为这个值的row_id=0。之后再插入的a=3的行，由于row_id=1，就覆盖了之前a=1的行，因为a=1这一行的row_id也是1。

从这个角度看，我们还是应该在InnoDB表中主动创建自增主键。因为，表自增id到达上限后，再插入数据时报主键冲突错误，是更能被接受的。

毕竟覆盖数据，就意味着数据丢失，影响的是数据可靠性；报主键冲突，是插入失败，影响的是可用性。而一般情况下，可靠性优先于可用性。

Xid

在第15篇文章 [《答疑文章（一）：日志和索引相关问题》](#) 中，我和你介绍redo log和binlog相配合的时候，提到了它们有一个共同的字段叫作Xid。它在MySQL中是用来对应事务的。

那么，Xid在MySQL内部是怎么生成的呢？

MySQL内部维护了一个全局变量global_query_id，每次执行语句的时候将它赋值给Query_id，然后给这个变量加1。如果当前语句是这个事务执行的第一条语句，那么MySQL还会同时把Query_id赋值给这个事务的Xid。

而global_query_id是一个纯内存变量，重启之后就清零了。所以你就知道了，在同一个数据库实例中，不同事务的Xid也是有可能相同的。

但是MySQL重启之后会重新生成新的binlog文件，这就保证了，同一个binlog文件里，Xid一定是惟一的。

虽然MySQL重启不会导致同一个binlog里面出现两个相同的Xid，但是如果global_query_id达到上限后，就会继续从0开始计数。从理论上讲，还是就会出现同一个binlog里面出现相同Xid的场景。

因为global_query_id定义的长度是8个字节，这个自增值的上限是 $2^{64}-1$ 。要出现这种情况，必须是下面这样的过程：

1. 执行一个事务，假设Xid是A；
2. 接下来执行 2^{64} 次查询语句，让global_query_id回到A；
3. 再启动一个事务，这个事务的Xid也是A。

不过， 2^{64} 这个值太大了，大到你可以认为这个可能性只会存在于理论上。

InnoDB trx_id

Xid和InnoDB的trx_id是两个容易混淆的概念。

Xid是由server层维护的。InnoDB内部使用Xid，就是为了能够在InnoDB事务和server之间做关联。但是，InnoDB自己的trx_id，是另外维护的。

其实，你应该非常熟悉这个trx_id。它就是在我们在第8篇文章 [《事务到底是隔离的还是不隔离的？》](#) 中讲事务可见性时，用到的事务id（transaction id）。

InnoDB内部维护了一个max_trx_id全局变量，每次需要申请一个新的trx_id时，就获得max_trx_id的当前值，然后将max_trx_id加1。

InnoDB数据可见性的核心思想是：每一行数据都记录了更新它的trx_id，当一个事务读到一行数

据的时候，判断这个数据是否可见的方法，就是通过事务的一致性视图与这行数据的`trx_id`做对比。

对于正在执行的事务，你可以从`information_schema.innodb_trx`表中看到事务的`trx_id`。

我在上一篇文章的末尾留给你的思考题，就是关于从`innodb_trx`表里面查到的`trx_id`的。现在，我们一起来看一个事务现场：

	session A	session B
T1	<code>begin;</code> <code>select * from t limit 1;</code>	
T2		<code>use information_schema;</code> <code>select trx_id, trx_mysql_thread_id from innodb_trx;</code> <code>/*</code> <code>+-----+-----+</code> <code> trx_id trx_mysql_thread_id </code> <code>+-----+-----+</code> <code> 421578461423440 5 </code> <code>+-----+-----+</code> <code>*/</code>
T3	<code>insert into t values(null);</code>	
T4		<code>select trx_id, trx_mysql_thread_id from innodb_trx;</code> <code>/*</code> <code>+-----+-----+</code> <code> trx_id trx_mysql_thread_id </code> <code>+-----+-----+</code> <code> 1289 5 </code> <code>+-----+-----+</code> <code>*/</code>

图3 事务的`trx_id`

`session B`里，我从`innodb_trx`表里查出的这两个字段，第二个字段`trx_mysql_thread_id`就是线程`id`。显示线程`id`，是为了说明这两次查询看到的事务对应的线程`id`都是5，也就是`session A`所在的线程。

可以看到，`T2`时刻显示的`trx_id`是一个很大的数；`T4`时刻显示的`trx_id`是1289，看上去是一个比较正常的数字。这是什么原因呢？

实际上，在`T1`时刻，`session A`还没有涉及到更新，是一个只读事务。而对于只读事务，`InnoDB`并不会分配`trx_id`。也就是说：

1. 在`T1`时刻，`trx_id`的值其实就是0。而这个很大的数，只是显示用的。一会儿我会再和你说说

这个数据的生成逻辑。

2. 直到session A 在T3时刻执行insert语句的时候，InnoDB才真正分配了trx_id。所以，T4时刻，session B查到的这个trx_id的值就是1289。

需要注意的是，除了显而易见的修改类语句外，如果在select 语句后面加上for update，这个事务也不是只读事务。

在上一篇文章的评论区，有同学提出，实验的时候发现不止加1。这是因为：

1. update 和 delete语句除了事务本身，还涉及到标记删除旧数据，也就是要把数据放到purge 队列里等待后续物理删除，这个操作也会把max_trx_id+1，因此在一个事务中至少加2；
2. InnoDB的后台操作，比如表的索引信息统计这类操作，也是会启动内部事务的，因此你可能看到，trx_id值并不是按照加1递增的。

那么，T2时刻查到的这个很大的数字是怎么来的呢？

其实，这个数字是每次查询的时候由系统临时计算出来的。它的算法是：把当前事务的trx变量的指针地址转成整数，再加上 2^{48} 。使用这个算法，就可以保证以下两点：

1. 因为同一个只读事务在执行期间，它的指针地址是不会变的，所以不论是在 innodb_trx还是在innodb_locks表里，同一个只读事务查出来的trx_id就会是一样的。
2. 如果有并行的多个只读事务，每个事务的trx变量的指针地址肯定不同。这样，不同的并发只读事务，查出来的trx_id就是不同的。

那么，为什么还要再加上 2^{48} 呢？

在显示值里面加上 2^{48} ，目的是要保证只读事务显示的trx_id值比较大，正常情况下就会区别于读写事务的id。但是，trx_id跟row_id的逻辑类似，定义长度也是8个字节。因此，在理论上还是可能出现一个读写事务与一个只读事务显示的trx_id相同的情况。不过这个概率很低，并且也没有什么实质危害，可以不管它。

另一个问题是，只读事务不分配trx_id，有什么好处呢？

- 一个好处是，这样做可以减小事务视图里面活跃事务数组的大小。因为当前正在运行的只读事务，是不影响数据的可见性判断的。所以，在创建事务的一致性视图时，InnoDB就只需要拷贝读写事务的trx_id。
- 另一个好处是，可以减少trx_id的申请次数。在InnoDB里，即使你只是执行一个普通的select 语句，在执行过程中，也是要对一个只读事务的。所以只读事务优化后，普通的查询语句不需要申请trx_id，就大大减少了并发事务申请trx_id的锁冲突。

由于只读事务不分配`trx_id`，一个自然而然的结果就是`trx_id`的增加速度变慢了。

但是，`max_trx_id`会持久化存储，重启也不会重置为0，那么从理论上讲，只要一个MySQL服务跑得足够久，就可能出现`max_trx_id`达到 $2^{48}-1$ 的上限，然后从0开始的情况。

当达到这个状态后，MySQL就会持续出现一个脏读的bug，我们来复现一下这个bug。

首先我们需要把当前的`max_trx_id`先修改成 $2^{48}-1$ 。注意：这个case里使用的是可重复读隔离级别。具体的操作流程如下：

```
mysql> create table t(id int primary key, c int)engine=innodb;
mysql> insert into t values(1,1);
gdb -p <pid.mysql> -ex 'p trx_sys->max_trx_id=281474976710655' --batch
```

	session A	session B
T1	<pre>begin; select * from t; // TA /* +----+-----+ id c +----+-----+ 1 1 +----+-----+ */</pre>	
T2		<pre>update t set c=2 where id=1; begin; update t set c=3 where id=1;</pre>
T3	<pre>select * from t; /* +----+-----+ id c +----+-----+ 1 3 +----+-----+ 脏读 */</pre>	

图 4 复现脏读

由于我们已经把系统的`max_trx_id`设置成了 $2^{48}-1$ ，所以在session A启动的事务TA的低水位就是

$2^{48}-1$ 。

在T2时刻，**session B**执行第一条**update**语句的事务id就是 $2^{48}-1$ ，而第二条**update**语句的事务id就是0了，这条**update**语句执行后生成的数据版本上的**trx_id**就是0。

在T3时刻，**session A**执行**select**语句的时候，判断可见性发现，**c=3**这个数据版本的**trx_id**，小于事务**TA**的低水位，因此认为这个数据可见。

但，这个是脏读。

由于低水位值会持续增加，而事务id从0开始计数，就导致了系统在这个时刻之后，所有的查询都会出现脏读的。

并且，MySQL重启时**max_trx_id**也不会清0，也就是说重启MySQL，这个bug仍然存在。

那么，这个bug也是只存在于理论上吗？

假设一个MySQL实例的TPS是每秒50万，持续这个压力的话，在17.8年后，就会出现这个情况。如果TPS更高，这个年限自然也就更短了。但是，从MySQL的真正开始流行到现在，恐怕都还没有实例跑到过这个上限。不过，这个bug是只要MySQL实例服务时间够长，就会必然出现的。

当然，这个例子更现实的意义是，可以加深我们对低水位和数据可见性的理解。你也可以借此机会再回顾下第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的相关内容。

thread_id

接下来，我们再看看线程id (**thread_id**)。其实，线程id才是MySQL中最常见的一种自增id。平时我们在查各种现场的时候，**show processlist**里面的第一列，就是**thread_id**。

thread_id的逻辑很好理解：系统保存了一个全局变量**thread_id_counter**，每新建一个连接，就将**thread_id_counter**赋值给这个新连接的线程变量。

thread_id_counter定义的大小是4个字节，因此达到 $2^{32}-1$ 后，它就会重置为0，然后继续增加。但是，你不会在**show processlist**里看到两个相同的**thread_id**。

这，是因为MySQL设计了一个唯一数组的逻辑，给新线程分配**thread_id**的时候，逻辑代码是这样的：

```
do {
    new_id= thread_id_counter++;
} while (!thread_ids.insert_unique(new_id).second);
```


这个代码逻辑简单而且实现优雅，相信你一看就能明白。

小结

今天这篇文章，我给你介绍了MySQL不同的自增id达到上限以后的行为。数据库系统作为一个可能需要7*24小时全年无休的服务，考虑这些边界是非常有必要的。

每种自增id有各自的应用场景，在达到上限后的表现也不同：

1. 表的自增id达到上限后，再申请时它的值就不会改变，进而导致继续插入数据时报主键冲突的错误。
2. row_id达到上限后，则会归0再重新递增，如果出现相同的row_id，后写的数据会覆盖之前的数据。
3. Xid只需要不在同一个binlog文件中出现重复值即可。虽然理论上会出现重复值，但是概率极小，可以忽略不计。
4. InnoDB的max_trx_id递增值每次MySQL重启都会被保存起来，所以我们文章中提到的脏读的例子就是一个必现的bug，好在留给我们的时间还很充裕。
5. thread_id是我们使用中最常见的，而且也是处理得最好的一个自增id逻辑了。

当然，在MySQL里还有别的自增id，比如table_id、binlog文件序号等，就留给你去验证和探索了。

不同的自增id有不同的上限值，上限值的大小取决于声明的类型长度。而我们专栏声明的上限id就是45，所以今天这篇文章也是我们的最后一篇技术文章了。

既然没有下一个id了，课后也就没有思考题了。今天，我们换一个轻松的话题，请你来说说，读完专栏以后有什么感想吧。

这个“感想”，既可以是你读完专栏前后对某一些知识点的理解发生的变化，也可以是你积累的学习专栏文章的好方法，当然也可以是吐槽或者对未来的期望。

欢迎你给我留言，我们在评论区见，也欢迎你把这篇文章分享给更多的朋友一起阅读。

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言



Continue

👍 12

跟着学了三个多月，受益匪浅，学到了很多新的知识和其中的原理！

2019-02-25

作者回复

早☺

2019-02-25



克劳德

👍 7

本人服务端工程师，在学习这门课之前数据库一直是我的短板，曾听朋友说MySQL或数据库中涉及了很多方面的知识点，每一个拿出来展开讲几乎都能出一本书了，对数据库是越来越忌惮，同时也因为工作上并没有过多接触，水平便一直停留在编写简单SQL层面。

在面试中被问到数据库问题，只能无奈的说这块不太清楚，也曾在网上自学过，但网上的文章知识点比较零散，很多都是给出一些结论性的观点，由于不了解其内部原理，记忆很难深刻。老实说，当初报这门课的时候就像买技术书籍一样，我相信大家都有这样的体会，以为买到了就等于学到了，所以有一段时间没有点开看过，以至于后面开始学的时候都是在追赶老师和大家的进度，唯一遗憾的地方就是没能跟老师及时留言互动。

这门课虽然是文字授课，但字里行间给我的感觉就是很亲切很舒服，为什么呢，因为老师可以把晦涩的知识变得通俗易懂，有时我在思考，如果让我来讲一个自己擅长的领域是否也能做到这一点，如果要做到的话需要什么样的知识储备呢。

最后真要感谢老师的这门课，让我从心里不再惧怕数据库问题，不管是工作还是面试中信心倍增，现在时不时都敢和我们DBA“切磋切磋”了，哈哈。

祝好~

2019-02-25

作者回复

切磋切磋

留言不会“过时”哈，在对应的章节下面提出相关的问题，我会持续关注评论区

2019-02-25



三胖

3

老师，我才学了四分之一的课程，但是这门课已经更新完了，我是直接跑到最后一节技术篇来留言的！很想知道，后来者比如我在学到后面的课程时遇到问题留言，老师还会看会回复吗？
(老师的课程超值！！)

2019-02-25

作者回复

会看的

后台系统是按照留言时间显示的

而且我在这事情上有强迫症，一定会让“未处理问题”变成0的

只是说如果是其他同学评论区问过的问题，我可能就不会重复回复了

2019-02-25



某、人

2

很遗憾没能坚持到最后,但是也很庆幸能遇到这么好的专栏。以前了解mysql都是一些零散的知识点,通过学习完专栏,不论是mysql整体架构还是基础的知识点,都有了更深的认识。以后就把老师的文档当官方文档查,出现问题先来看看专栏。
感触特别深的是,老师对于提到的每一个问题,都会严谨又认真的去回答,尽量帮助每一位同学都能有所收获。要做到这一点,是特别耗费精力的。
感谢老师的传道授业解惑,希望以后有机会能当面向老师请教问题。期待老师下一部杰作

2019-02-26

作者回复

刚过完年都是很忙的，找时间补上哈，等你的评论区留言^_^

2019-02-26



夜空中最亮的星（华仔）

2

不知道是最后一篇，否则的话就慢些读完了；
我是一名运维，公司也没有DBA，所以MySQL库也归我收拾；
读了老师的专栏，操作起数据库来，心情更好了；
老师的课，让我有了想看完《高性能MySQL》的兴趣；
听了老师的课，开发都来问我数据库的问题了，高兴；
老师你会有返场吗？我猜会
可否透漏下接下来的安排，会有续集吗？进阶吗？
不想这一别就是一生。

您的从未谋面的学生。

2019-02-25

作者回复

谢谢你

“开发都来问我数据库的问题了”，当年我也是这么开始“入坑”，加油

2019-02-25



极客时间

2

通過這個專欄的系統學習，梳理很多知識點、擴展了我對MySQL的認識及以後使用。感謝老師的諄諄教導！

2019-02-25



NoDBA

1

低版本thread_id超过 $2^{32}-1$ 后，在general log显示是负数，高版本貌似没有这个问题，是否高版本的thread_id是8字节呢？

2019-02-27

作者回复

主要不是定义的问题，而是打印的时候代码问题，按照这个代码输出的：

```
"%5ld ", (long) thread_id
```

是个bug，超过 2^{31} 就变成负数了，

新版本改了

好问题

2019-02-28



kun

1

感觉戛然而止哈 没学够，后面还会再回顾，老师辛苦！

2019-02-26



亮

1

老师，sql的where里 < 10001 和 ≤ 10000 有什么区别吗？

2019-02-25

作者回复

这要看你关注的是什么

你这么问，应该这个字段是整型吧？

从查询结果可能是一样的，

不过锁的范围不同，你可以看下21篇

2019-02-25



IceGeek17

1

问题四、还是我文中的建议，如果都用NLJ或BKA算法的join其实还好，所以看看explain。

降低join表数量的方法，基本上行就是冗余字段和拆成多个语句这两个方向了

2019-02-25



Leon

1

跟着老师终于学到了最后，每天的地铁时间无比充实，我对mysql的基本原理和 workflow 大致有了初步的了解，而不是以前的增删查改，打算以后抽时间再二刷三刷，等全部搞懂后，再去看看高性能mysql这本书，如果时间允许，打算再去自己参照教程实现一个简易的DB，课程虽然结束了，仍然感觉意犹未尽，希望老师拉一个倍洽群，大家一起在里面讨论和学习

2019-02-25

作者回复

👍

评论区一直会开放

大家到对应的文章去提相关问题

二刷三刷我也一直在哦

2019-02-25



Dkey

1

当前系统并无其他事务存在时，启动一个只读事务时（意味没有事务id），它的低高水位是怎么样的老师。

2019-02-25

作者回复

假设当前没有其他事务存在，假设当前的max_trx_id=N，

这时候启动一个只读事务，它的高低水位就都是N。

2019-02-25



shawn

1

受益匪浅，最后几讲还想了解下null值如何建立索引，由于null直接不能比较和排序，MySQL能区分出每一个null值吗

2019-02-25

作者回复

可以，因为普通索引上都有主键值对吧，

所以其实是 (null, id1), (null, id2)

2019-02-25



亢星东

0

id是有上限的，这个的id上限是45，这个结局可以，讲的不错，学到很多

2019-03-13



Bamboo

0

今天终于读完了，从对MySQL只停留在CRUD操作的水平，慢慢开始对MySQL底层的机制有了一些认识，在遇到问题时，会首先从底层原理去分析，并结合explain来验证自己的分析，一次很nice的学习之旅。感谢大神老师这么认真负责，节假日都不休息，哈哈！

2019-03-12

作者回复

👍

2019-03-13



ArtistLu

👍 0

相遇恨晚👍，安慰下自己，种树的最好时机是十年前，其次是现在!!! 谢谢老师

2019-03-08

作者回复

👍

2019-03-09



fighting

👍 0

已经二刷了，准备三刷四刷

2019-03-07

作者回复

👍👍

2019-03-09



沙漠里的骆驼

👍 0

讲的非常好，是我遇到课程讲授最好的了。今天刚和池老师说，希望可以有线下的课程，比如完成一个数据库的完整设计，从最上层的sql语法解析器到底层的文件调度系统。在集中的时间里比如1个月或者2个月，线下组织大家一起，每个人都完成一个tiny_db的工程。我想这是最好的成长了。不知道老师是否也有这方面的想法？

不管如何，真的很感谢老师。如此娓娓道来，所谓的如沐春风便是如此吧。

2019-03-06

作者回复

谢谢你。

后面只要还是在评论区继续和大家交流👍

2019-03-07



芬

👍 0

学习到了很多平时没有关注到的小细节，很赞！当然 师傅领进门 修行靠个人。剩下的就是自己好好消化应用了，谢谢老师

2019-02-28



封建的风

👍 0

之前很多知识点有点粗浅，尤其在行版本可见性，redo log&bin log关系，加锁的原理章节，深入浅出，受益匪浅。感谢老师精品专栏，后期再二刷

2019-02-27