1. MySQL数据库方式

1. Redis Incr方式分配自增ID
2. 数据库方式3和Redis Incr分配自增ID的性能对比

1. MySQL数据库方式

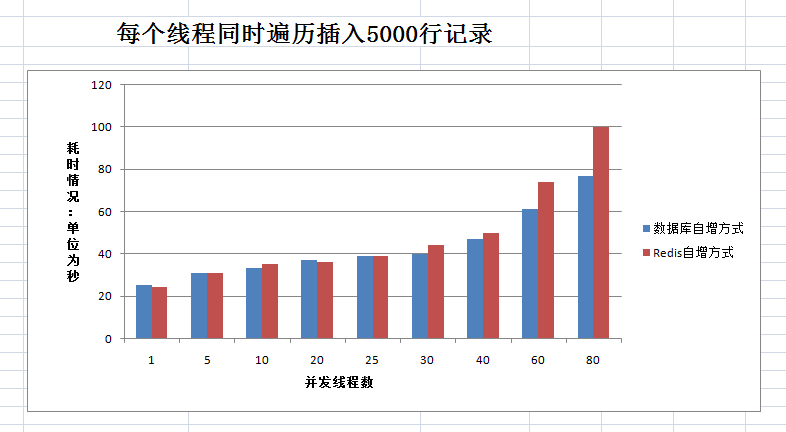
|  |  |  |  |
| --- | --- | --- | --- |
| 数据库方式 |  | 优点、缺点 | 性能对比 |
| 方式1 | 自增ID表  CREATE TABLE `ticket\_mutex` (  `id` bigint(20) unsigned NOT NULL auto\_increment,  `stub` char(1) NOT NULL default '',  PRIMARY KEY (`id`),  UNIQUE KEY `stub` (`stub`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  获取ID  REPLACE INTO ticket\_mutex (stub) VALUES ('a');  SELECT LAST\_INSERT\_ID(); | 优点：  采用了Unique Key的限制，确保这个表永远只有一行  缺点：  并发场景，会产生死锁 | 5个并发线程，每个线程遍历插入5000行记录，其中有2-3线程被死锁回滚掉，测不出导入25000行记录完整的耗时。 |
| 方式2 | 自增ID表  CREATE TABLE `ticket\_mutex\_2` (  name char(8) NOT NULL PRIMARY KEY,  value bigint(20) UNSIGNED NOT NULL  )Engine=InnoDB DEFAULT CHARSET=utf8mb4;  INSERT INTO ticket\_mutex\_2(name, value) values('detail', 1);  获取ID  UPDATEticket\_mutex\_2 SET value=LAST\_INSERT\_ID(value+1) WHERE name='detail';  SELECT LAST\_INSERT\_ID(); | 优点：  1. 采用了Primary Key的限制，确保这个表永远只有一行。  2. 一个业务逻辑一行，多个业务逻辑可以共用1个表来分配自增ID。  缺点：  只有更新操作，每次update会锁住对应的行记录，相当于是串行执行，并发情况下会有大量的行锁等待。 | 5个并发线程，每个线程遍历插入5000行记录，耗时118秒，产生24995次行锁等待。 |
| 方式3 | 自增ID表  CREATE TABLE `ticket\_mutex\_3` (  `id` bigint(20) unsigned NOT NULL auto\_increment,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  获取ID  INSERT INTO ticket\_mutex\_3 () values ();  SELECT LAST\_INSERT\_ID(); | 优点：  只会在插入数据的时候申请自增长锁，申请完就释放锁，并发性最好，效率最高。  缺点：  每获取一次自增ID，都会产生一行，长时间跑，导致该表越来越大。  经过测试，当该表达到1亿行记录，物理大小为3GB。 | 5个并发线程，每个线程遍历插入5000行记录，耗时31秒，没有行锁等待。 |
| 3种方式的对比 | 经过测试，个人觉得方式1和方式2不可靠；  个人觉得在数据库的3种方式中，方式3可用，该自增ID表可以在每次停服更新的时候，把该表删除后再重新建立表。 | | |

2. Redis Incr方式分配自增ID

生产环境上Redis 需要有高可用的保障，避免单点问题，比如：Sentinel+Redis 的高可用

3. 数据库方式3和Redis Incr分配自增ID的性能对比

环境：MySQL 和 Redis 都是用 39.108.193.40服务器上的。



前25个线程数，两种方式的性能不相上下。

并发线程数达到30以上的时候，Redis 自增方式就没有MySQL这么优秀了。