Mysql主从复制延时问题

1、如何查看同步延迟状态？

在从服务器上通过 show slave status 查看具体的参数，有几个参数比较重要：

master_log_file:　slave中的IO线程正在读取的主服务器二进制日志文件的名称

read_master_log_pos: 在当前的主服务器二进制日志中，slave中的IO线程已经读取的位置

relay_log_file:　　sql线程当前正在读取和执行的中继日志文件的名称

relay_log_pos:　　在当前的中继日志中，sql线程已经读取和执行的位置

relay_master_log_file: 由sql线程执行的包含多数近期事件的主服务器二进制日志文件的名称

slave_io_running: IO线程是否被启动并成功的连接到主服务器上

slave_sql_running:　　sql线程是否被启动

seconds_behind_master:　从属服务器sql线程和从属服务器IO线程之间的事件差距，单位以秒计

在观察同步延迟的时候，上述的几个参数都是比较重要的，其中有一个最最重要的参数需要引起注意，那就是seconds_behind_master，这个参数就表示当前备库延迟了多长时间，那么这个值是如何计算的呢？

在进行主从复制的时候，需要注意以下几个关键的时刻：

1、主库A执行完成一个事务，写入binlog，我们把这个时刻记为T1;

2、之后传给备库B,我们把备库B接受完这个binlog的时刻记为T2;

3、备库B执行完成这个事务，我们把这个时刻记为T3;

所谓的主备延迟就是同一个事务，在备库执行完成的时间和主库执行完成的时间之间的差值，也就是T3-T1。SBM在进行计算的时候也是按照这样的方式，每个事务的binlog中都有一个时间字段，用于记录主库写入的时间，备库取出当前正在执行的事务的时间字段的值，计算它与当前系统时间的差值，得到SBM。

如果刚刚的流程听明白了，那么下面我们就要开始分析产生这个时间差值的原因有哪些了，以方便我们更好的解决生产环境中存在的问题。

2、主从复制延迟产生的原因有哪些？

1. 在某些部署环境中，备库所在的机器性能要比主库所在的机器性能差。此时如果机器的资源不足的话就会影响备库同步的效率；
2. 备库充当了读库，一般情况下主要写的压力在于主库，那么备库会提供一部分读的压力，而如果备库的查询压力过大的话，备库的查询消耗了大量的CPU资源，那么必不可少的就会影响同步的速度
3. 大事务执行，如果主库的一个事务执行了10分钟，而binlog的写入必须要等待事务完成之后，才会传入备库，那么此时在开始执行的时候就已经延迟了10分钟了
4. 主库的写操作是顺序写binlog，从库单线程去主库顺序读binlog，从库取到binlog之后在本地执行。mysql的主从复制都是**单线程**的操作，但是由于主库是顺序写，所以效率很高，而从库也是顺序读取主库的日志，此时的效率也是比较高的，但是当数据拉取回来之后变成了随机的操作，而不是顺序的，所以此时成本会提高。
5. 从库在同步数据的同时，可能跟其他查询的线程发生锁抢占的情况，此时也会发生延时。
6. 当主库的TPS并发非常高的时候，产生的DDL数量超过了一个线程所能承受的范围的时候，那么也可能带来延迟
7. 在进行binlog日志传输的时候，如果网络带宽也不是很好，那么网络延迟也可能造成数据同步延迟

这些就是可能会造成备库延迟的原因

3、如何解决复制延迟的问题

先说一些虚的东西，什么叫虚的东西呢？就是一听上去感觉很有道理，但是在实施或者实际的业务场景中可能难度很大或者很难实现，下面我们从几个方面来进行描述：

1、架构方面

1、业务的持久化层的实现采用分库架构，让不同的业务请求分散到不同的数据库服务上，分散单台机器的压力

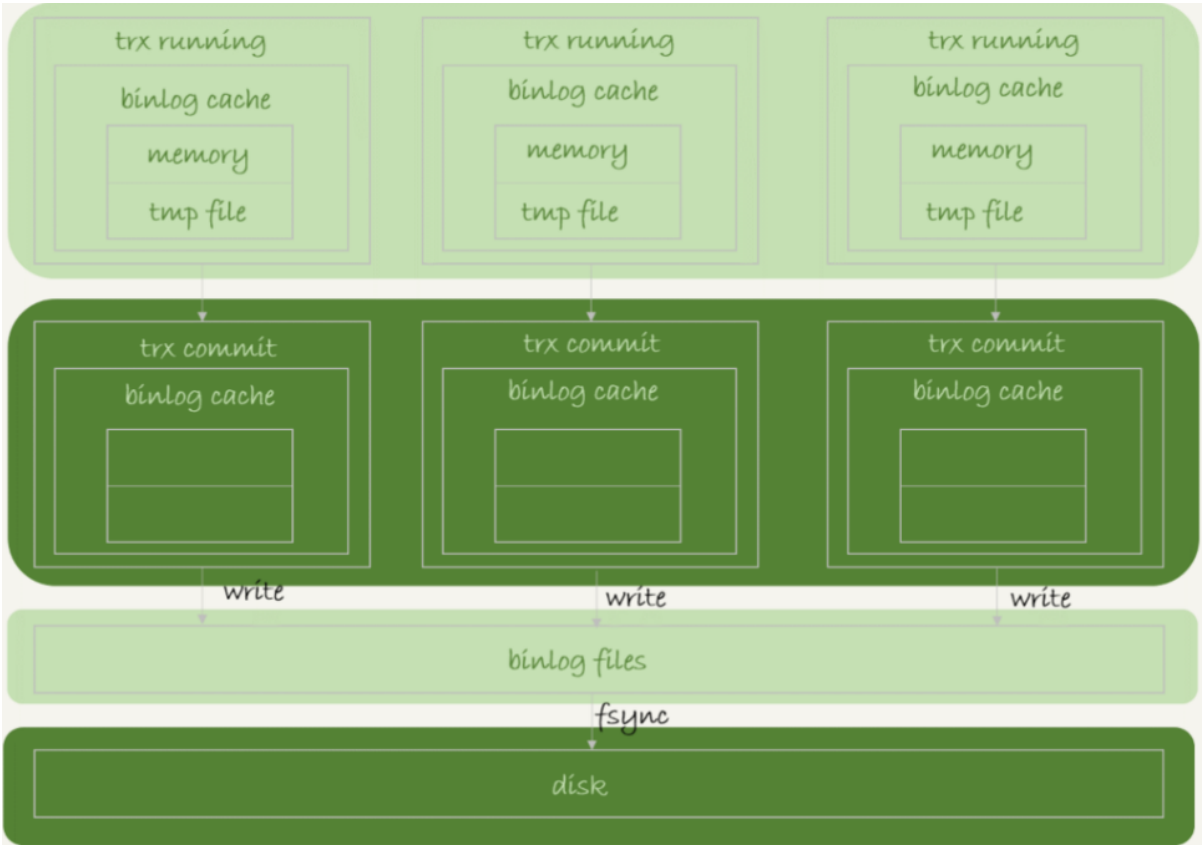
2、服务的基础架构在业务和mysql之间加入缓存层，减少mysql的读的压力，但是需要注意的是，如果数据经常要发生修改，那么这种设计是不合理的，因为需要频繁的去更新缓存中的数据，保持数据的一致性，导致缓存的命中率很低，所以此时就要慎用缓存了

3、使用更好的硬件设备，比如cpu，ssd等，但是这种方案一般对于公司而言不太能接受，原因也很简单，会增加公司的成本，而一般公司其实都很抠门，所以意义也不大，但是你要知道这也是解决问题的一个方法，只不过你需要评估的是投入产出比而已。

2、从库配置方面

1、修改sync_binlog的参数的值

想要合理设置此参数的值必须要清楚的知道binlog的写盘的流程：



可以看到，每个线程有自己的binlog cache，但是共用同一份binlog。

图中的write，指的就是把日志写入到文件系统的page cache，并没有把数据持久化到磁盘，所以速度快

图中的fsync，才是将数据持久化到磁盘的操作。一般情况下，我们认为fsync才占用磁盘的IOPS

而write和fsync的时机就是由参数sync_binlog来进行控制的。

1、当sync_binlog=0的时候，表示每次提交事务都只write，不fsync

2、当sync_binlog=1的时候，表示每次提交事务都执行fsync

3、当sync_binlog=N的时候，表示每次提交事务都write，但积累N个事务后才fsync。

一般在公司的大部分应用场景中，我们建议将此参数的值设置为1，因为这样的话能够保证数据的安全性，但是如果出现主从复制的延迟问题，可以考虑将此值设置为100~1000中的某个数值，非常不建议设置为0，因为设置为0的时候没有办法控制丢失日志的数据量，但是如果是对安全性要求比较高的业务系统，这个参数产生的意义就不是那么大了。

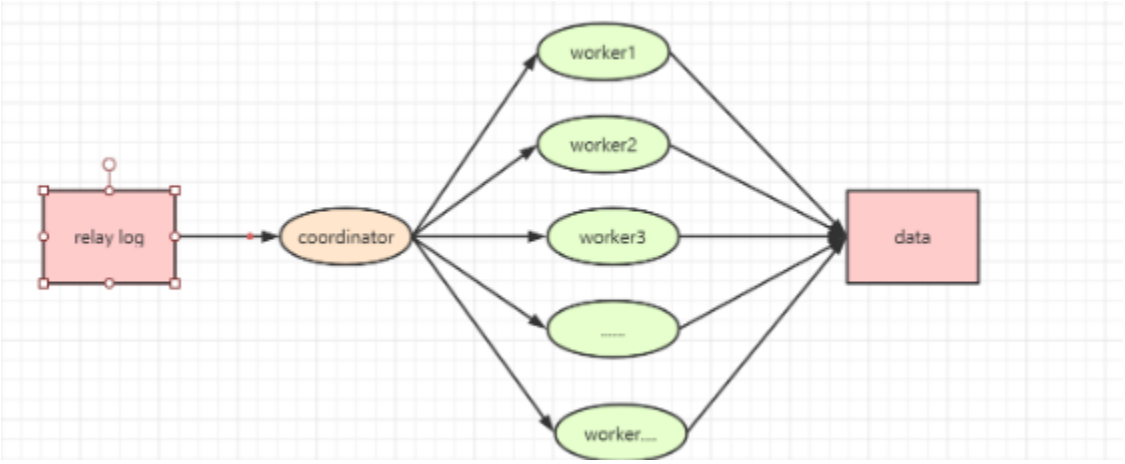
2、直接禁用salve上的binlog，当从库的数据在做同步的时候，有可能从库的binlog也会进行记录，此时的话肯定也会消耗io的资源，因此可以考虑将其关闭，但是需要注意，如果你搭建的集群是级联的模式的话，那么此时的binlog也会发送到另外一台从库里方便进行数据同步，此时的话，这个配置项也不会起到太大的作用。

3、设置innodb_flush_log_at_trx_commit 属性，这个属性在我讲日志的时候讲过，用来表示每一次的事务提交是否需要把日志都写入磁盘，这是很浪费时间的，一共有三个属性值，分别是0（每次写到服务缓存，一秒钟刷写一次），1（每次事务提交都刷写一次磁盘），2（每次写到os缓存，一秒钟刷写一次），一般情况下我们推荐设置成2，这样就算mysql的服务宕机了，卸载os缓存中的数据也会进行持久化。

4、从根本上解决主从复制的延迟问题

很多同学在自己线上的业务系统中都使用了mysql的主从复制，但是大家需要注意的是，并不是所有的场景都适合主从复制，一般情况下是读要远远多于写的应用，同时读的时效性要求不那么高的场景。如果真实场景中真的要求立马读取到更新之后的数据，那么就只能强制读取主库的数据，所以在进行实现的时候要考虑实际的应用场景，不要为了技术而技术，这是很严重的事情。

在mysql5.6版本之后引入了一个概念，就是我们通常说的并行复制，如下图：



通过上图我们可以发现其实所谓的并行复制，就是在中间添加了一个分发的环节，也就是说原来的sql_thread变成了现在的coordinator组件，当日志来了之后，coordinator负责读取日志信息以及分发事务，真正的日志执行的过程是放在了worker线程上，由多个线程并行的去执行。

```
-- 查看并行的slave的线程的个数，默认是0.表示单线程
show global variables like 'slave_parallel_workers';
-- 根据实际情况保证开启多少线程
set global slave_parallel_workers = 4;
-- 设置并发复制的方式，默认是一个线程处理一个库，值为database
show global variables like '%slave_parallel_type%';
-- 停止slave
stop slave;
-- 设置属性值
set global slave_parallel_type='logical_check';
-- 开启slave
start slave
-- 查看线程数
show full processlist;
```

通过上述的配置可以完成我们说的并行复制，但是此时你需要思考几个问题

1、在并行操作的时候，可能会有并发的事务问题，我们的备库在执行的时候可以按照轮训的方式发送给各个worker吗？

> 答案是不行的，因为事务被分发给worker以后，不同的worker就开始独立执行了，但是，由于CPU的不同调度策略，很可能第二个事务最终比第一个事务先执行，而如果刚刚好他们修改的是同一行数据，那么因为执行顺序的问题，可能导致主备的数据不一致。

2、同一个事务的多个更新语句，能不能分给不同的worker来执行呢？

> 答案是也不行，举个例子，一个事务更新了表t1和表t2中的各一行，如果这两条更新语句被分到不同worker的话，虽然最终的结果是主备一致的，但如果表t1执行完成的瞬间，备库上有一个查询，就会看到这个事务更新了一半的结果，破坏了事务逻辑的隔离性。

我们通过讲解上述两个问题的最主要目的是为了说明一件事，就是coordinator在进行分发的时候，需要遵循的策略是什么？

> 1、不能造成更新覆盖。这就要求更新同一行的两个事务，必须被分发到同一个worker中。
>
> 2、同一个事务不能被拆开，必须放到同一个worker中。

听完上面的描述，我们来说一下具体实现的原理和过程。

如果让我们自己来设计的话，我们应该如何操作呢？这是一个值得思考的问题。其实如果按照实际的操作的话，我们可以按照粒度进行分类，分为按库分发，按表分发，按行分发。

其实不管按照什么方式进行分发，需要注意的就是在分发的时候必须要满足我们上面说的两条规则，所以当我们进行分发的时候要在每一个worker上定义一个hash表，用来保存当前这个work正在执行的事务所涉及到的表。hash表的key值按照不同的粒度需要存储不同的值：

按库分发：key值是数据库的名字，这个比较简单

按表分发：key值是库名+表名

按行分发：key值是库名+表名+唯一键

1、MySQL5.6版本的并行复制策略

其实从mysql的5.6版本开始就已经支持了并行复制，只是支持的粒度是按库并行，这也是为什么现在的版本中可以选择类型为database，其实说的就是支持按照库进行并行复制。

但是其实用过的同学应该都知道，这个策略的并行效果，取决于压力模型。如果在主库上有多个DB，并且各个DB的压力均衡，使用这个策略的效果会很好，但是如果主库的所有表都放在同一DB上，那么所有的操作都会分发给一个worker，变成单线程操作了，那么这个策略的效果就不好了，因此在实际的生产环境中，用的并不是特别多。

2、mariaDB的并行复制策略

在mysql5.7的时候采用的是基于组提交的并行复制，换句话说，slave服务器的回放与主机是一致的，即主库是如何并行执行的那么slave就如何怎样进行并行回放，这点其实是参考了mariaDB的并行复制，下面我们来看下其实现原理。

mariaDB的并行复制策略利用的就是这个特性：

1、能够在同一组里提交的事务，一定不会修改同一行；

2、主库上可以并行执行的事务，备库上也一定是可以并行执行的。

在实现上，mariaDB是这么做的：

1、在一组里面一起提交的事务，有一个相同的commit_id,下一组就是commit_id+1;

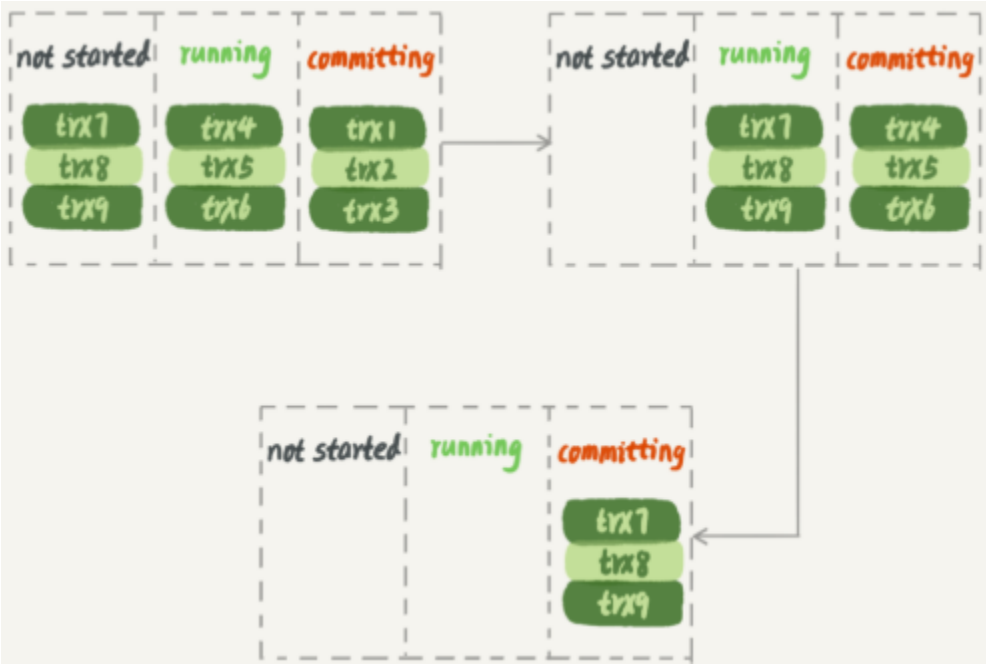
2、commit_id直接写到binlog里面；

3、传到备库应用的时候，相同commit_id的事务会分发到多个worker执行；

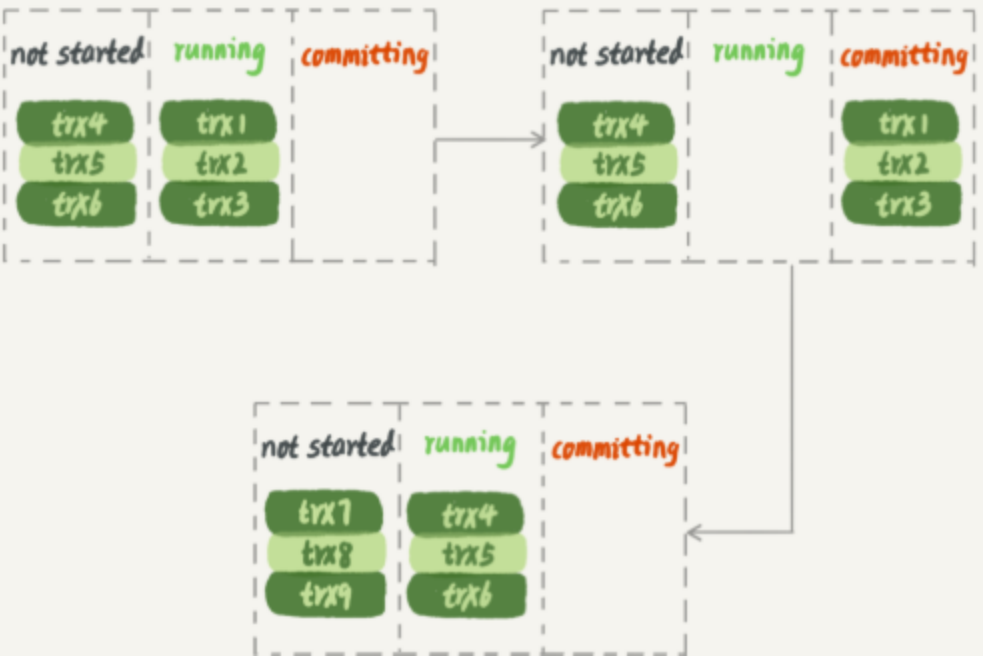
4、这一组全部执行完成后，coordinator再去取下一批。

这是mariaDB的并行复制策略，大体上看起来是没有问题的，但是你仔细观察的话会发现他并没有实现“真正的模拟主库并发度”这个目标，在主库上，一组事务在commit的时候，下一组事务是同时处于“执行中”状态的。

我们真正想要达到的并行复制应该是如下的状态，也就是说当第一组事务提交的是，下一组事务是运行的状态，当第一组事务提交完成之后，下一组事务会立刻变成commit状态。



但是按照mariaDB的并行复制策略，那么备库上的执行状态会变成如下所示：



可以看到，这张图跟上面这张图的最大区别在于，备库上执行的时候必须要等第一组事务执行完成之后，第二组事务才能开始执行，这样系统的吞吐量就不够了。而且这个方案很容易被大事务拖后腿，如果trx2是一个大事务，那么在备库应用的时候，trx1和trx3执行完成之后，就只能等trx2完全执行完成，下一组才能开始执行，这段时间，只有一个worker线程在工作，是对资源的浪费。

3、mysql5.7的并行复制策略

mysql5.7版本的时候，根据mariaDB的并行复制策略，做了相应的优化调整后，提供了自己的并行复制策略，并且可以通过参数slave-parallel-type来控制并行复制的策略：

1、当配置的值为DATABASE的时候，表示使用5.6版本的按库并行策略；

2、当配置的值为LOGICAL_CLOCK的时候，表示跟mariaDB相同的策略。

此时，大家需要思考一个问题：**同时处于执行状态的所有事务，是否可以并行？**

答案是不行的，因为多个执行中的事务是有可能出现锁冲突的，锁冲突之后就会产生锁等待问题。

在mariaDB中，所有处于commit状态的事务是可以并行，因为如果能commit的话就说明已经没有锁的问题，但是大家回想下，我们mysql的日志提交是两阶段提交，如下图，其实只要处于prepare状态就已经表示没有锁的问题了。

因此，mysql5.7的并行复制策略的思想是：

1、同时处于prepare状态的事务，在备库执行是可以并行的。

2、处于prepare状态的事务，与处于commit状态的事务之间，在备库上执行也是可以并行的。

基于这样的处理机制，我们可以将大部分的日志处于prepare状态，因此可以设置

1、binlog_group_commit_sync_delay 参数，表示延迟多少微秒后才调用 fsync;

2、binlog_group_commit_sync_no_delay_count 参数，表示累积多少次以后才调用 fsync。

5、基于GTID的主从复制问题

在我们之前讲解的主从复制实操中，每次想要复制，必须要在备机上执行对应的命令，如下所示：

```
change master to master_host='192.168.85.11',master_user='root',master_password='123456',m
aster_port=3306,master_log_file='master-bin.000001',master_log_pos=154;
```

在此配置中我们必须要知道具体的binlog是哪个文件，同时在文件的哪个位置开始复制，正常情况下也没有问题，但是如果是一个主备主从集群，那么如果主机宕机，当从机开始工作的时候，那么备机就要同步从机的位置，此时位置可能跟主机的位置是不同的，因此在这种情况下，再去找位置就会比较麻烦，所以在5.6版本之后出来一个基于GTID的主从复制。

GTID(global transaction id)是对于一个已提交事务的编号，并且是一个全局唯一的编号。GTID实际上是由UUID+TID组成的，其中UUID是mysql实例的唯一标识，TID表示该实例上已经提交的事务数量，并且随着事务提交单调递增。这种方式保证事务在集群中有唯一的ID，强化了主备一致及故障恢复能力。

1、基于GTID的搭建

1、修改mysql配置文件，添加如下配置

```
gtid_mode=on
enforce-gtid-consistency=true
```

2、重启主从的服务

3、从库执行如下命令

```
change master to master_host='192.168.85.111',master_user='root',master_password='123456'
,master_auto_position=1;
```

4、主库从库插入数据测试。

2、基于GTID的并行复制

无论是什么方式的主从复制其实原理相差都不是很大，关键点在于将组提交的信息存放在GTId中。

```
show binlog events in 'lian-bin.000001';
```

previous_gtids:用于表示上一个binlog最后一个gtid的位置，每个binlog只有一个。

gtid:当开启gtid的时候，每一个操作语句执行前会添加一个gtid事件，记录当前全局事务id，组提交信息被保存在gtid事件中，有两个关键字段，last_committed,sequence_number用来标识组提交信息。

上述日志看起来可能比较麻烦，可以使用如下命令执行：

其中last_committed表示事务提交的时候，上次事务提交的编号，如果事务具有相同的last_committed值表示事务就在一个组内，在备库执行的时候可以并行执行。同时大家还要注意，每个last_committed的值都是上一个组事务的sequence_number值。

看到此处，大家可能会有疑问，如果我们不开启gtid，分组信息该如何保存呢？

其实是一样的，当没有开启的时候，数据库会有一个Anonymous_Gtid，用来保存组相关的信息。

如果想看并行的效果的话，可以执行如下代码：

```
package com.test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Date;

public class ConCurrentInsert  extends Thread{
    public void run() {
        String url = "jdbc:mysql://192.168.85.111/lian2";
        String name = "com.mysql.jdbc.Driver";
```

```
        String user = "root";
        String password = "123456";
        Connection conn = null;
        try {
            Class.forName(name);
            conn = DriverManager.getConnection(url, user, password);//获取连接
            conn.setAutoCommit(false);//关闭自动提交，不然conn.commit()运行到这句会报错
        } catch (Exception e1) {
            e1.printStackTrace();
        }
        // 开始时间
        Long begin = new Date().getTime();
        // sql前缀
        String prefix = "INSERT INTO t1 (id,age) VALUES ";
        try {
            // 保存sql后缀
            StringBuffer suffix = new StringBuffer();
            // 设置事务为非自动提交
            conn.setAutoCommit(false);
            // 比起st，pst会更好些
            PreparedStatement pst = (PreparedStatement) conn.prepareStatement("");//准备执
行语句
            // 外层循环，总提交事务次数
            for (int i = 1; i <= 10; i++) {
                suffix = new StringBuffer();
                // 第j次提交步长
                for (int j = 1; j <= 10; j++) {
                    // 构建SQL后缀
                    suffix.append("(" +i*j+","+i*j+"),");
                }
                // 构建完整SQL
                String sql = prefix + suffix.substring(0, suffix.length() - 1);
                // 添加执行SQL
                pst.addBatch(sql);
                // 执行操作
                pst.executeBatch();
                // 提交事务
                conn.commit();
                // 清空上一次添加的数据
                suffix = new StringBuffer();
            }
            // 头等连接
            pst.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        // 结束时间
        Long end = new Date().getTime();
        // 耗时
        System.out.println("100万条数据插入花费时间 : " + (end - begin) / 1000 + " s"+"  插入
完成");
    }
```

```
    public static void main(String[] args) {
        for (int i = 1; i <=10; i++) {
            new ConCurrentInsert().start();
        }
    }
}
```