务原理

事务原理：在事务开启之后，所有的操作都会被临时存储到事务日志，事务日志只有在收到commit命令之后，才会将操作同步到数据表，其他任何情况都会清空事务日志，例如突然断开连接、收到rollback命令等。

接下来，我们简单分析一下 MySQL 的操作过程：

Step 1：客户端与服务端建立连接，同时开启一个临时的事务日志，此事务日志只作用于当前用户的当次连接；

Step 2：在客户端用 SQL 语句执行写操作，客户端收到 SQL 语句，执行，将结果直接写入到数据表，并将数据表同步到数据库；

Step 3：我们在客户端开启事务，则服务端原来的操作机制被改变，后续所有操作都会被先写入到临时日志文件；

Step 4：在客户端执行 SQL 语句（例如写操作），服务端收到 SQL 语句，执行，将结果写入到临时日志文件，并不将结果同步到数据库；

Step 5：在客户端执行查询操作，服务端直接从临时日志文件中捞取数据，返回给客户端；

Step 6：在客户端执行commit或者rollback命令，清空临时日志文件，如果是commit命令，则将结果同步到数据库；如果是rollback命令，则不同步。

通过上面的分析，我们就知道了为什么在我们同时开启两个 MySQL 客户端窗口（两次连接）时，当一个窗口开启事务并执行 SQL 操作之后，另一个窗口在查询时并不会收到同步数据。原因就在于，当我们开启事务之后，服务端会将后续的操作都写入到临时日志文件，而另一个窗口在查询的时候，则是直接从数据库捞取数据，并会不走前一个的临时日志文件。

回滚点

回滚点：在某个操作成功完成之后，后续的操作有可能成功也有可能失败，但无论后续操作的结果如何，前一次操作都已经成功了，因此我们可以在当前成功的位置，设置一个操作点，其可以供后续操作返回该位置，而不是返回所有操作，这个点称之为回滚点。关于回滚点的基本语法为，

设置回滚点：savepoint + 回滚点名称;

返回回滚点：rollback to + 回滚点名称;

执行如下 SQL 语句，进行测试：

-- 测试回滚点

-- 查询 bank\_account 表数据

select \* from bank\_account;

-- 开启事务

start transaction;

-- 事务操作 1：给 Charies 发工资 1000 元

update bank\_account set money = money + 10000 where id = 1;

-- 设置回滚点

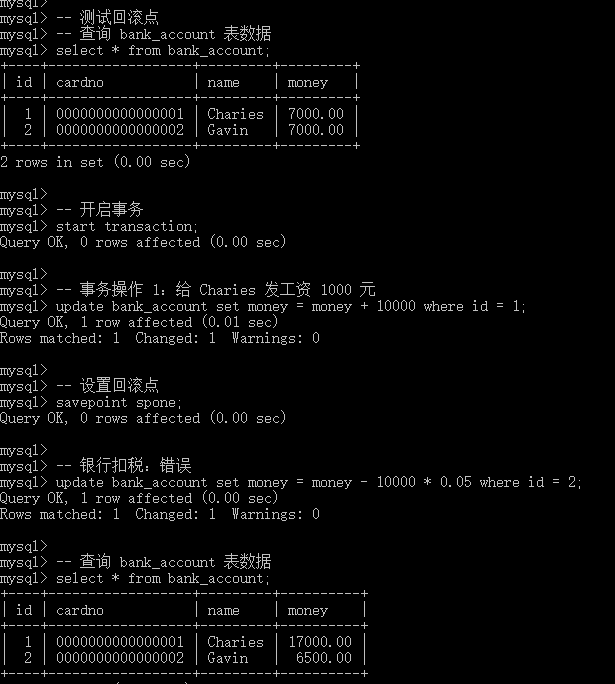
savepoint spone;

-- 银行扣税：错误

update bank\_account set money = money - 10000 \* 0.05 where id = 2;

-- 查询 bank\_account 表数据

select \* from bank\_account;

[](https://camo.githubusercontent.com/c3139a7d12fceb0889b1d23e86150fc4f8967bc6/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313035303239363931)

执行如下 SQL 语句，继续进行测试：

-- 测试回滚点

-- 返回回滚点

rollback to spone;

-- 银行扣税：正确

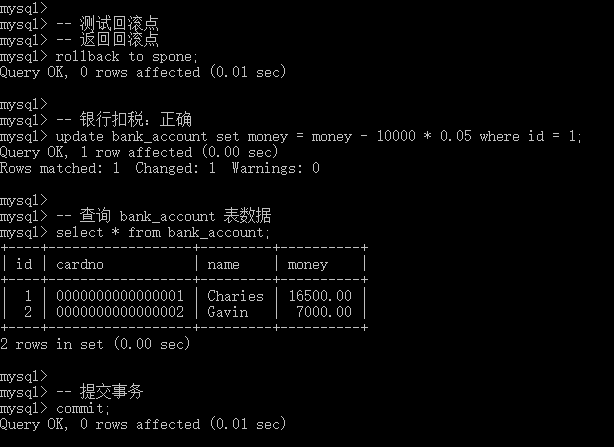
update bank\_account set money = money - 10000 \* 0.05 where id = 1;

-- 查询 bank\_account 表数据

select \* from bank\_account;

-- 提交事务

commit;

[](https://camo.githubusercontent.com/57ed5845a7f6a5e75b55a8c65823080ea2eaa6bf/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313035323238323330)

如上图所示，显然在执行返回回滚点的操作之后，我们之前的错误操作得到了修正。

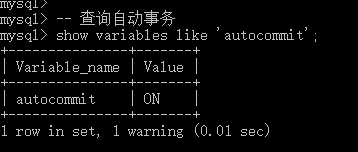
自动事务

在 MySQL 中，默认的都是自动事务处理，即用户在操作完成之后，其操作结果会立即被同步到数据库中。

自动事务是通过autocommit变量控制的，我们可以通过如下 SQL 语句，进行查看：

-- 查询自动事务

show variables like 'autocommit';

[](https://camo.githubusercontent.com/4058390aa172b841969093f039088e650b6397ac/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313035383330353833)

如上图所示，此为 MySQL 的默认设置。实际上，我们可以自己选择是否开启自动事务处理，其基本语法为，

开启自动事务处理：set autocommit = on / 1;

关闭自动事务处理：set autocommit = off / 0;

在此，我们以关闭自动事务处理为例，进行演示：

-- 关闭自动事务处理

set autocommit = 0;

-- 查看自动事务处理

show variables like 'autocommit';

-- 查看 bank\_account 表数据

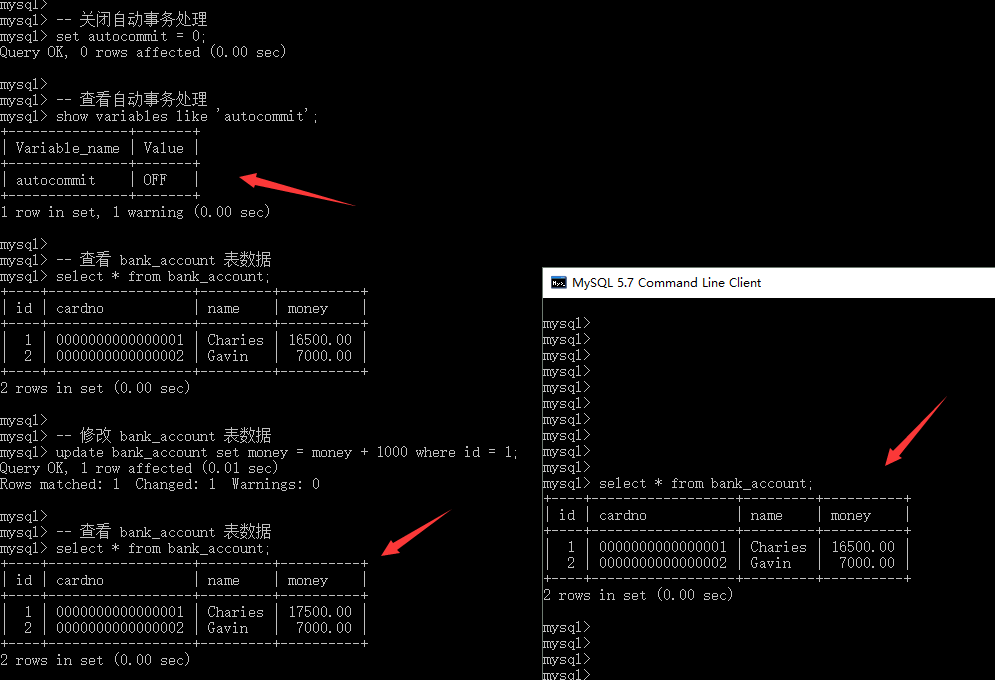
select \* from bank\_account;

-- 修改 bank\_account 表数据

update bank\_account set money = money + 1000 where id = 1;

-- 查看 bank\_account 表数据

select \* from bank\_account;

[](https://camo.githubusercontent.com/3f4a222b735f4dc9354b248677dee0fc82e63478/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313130383233363838)

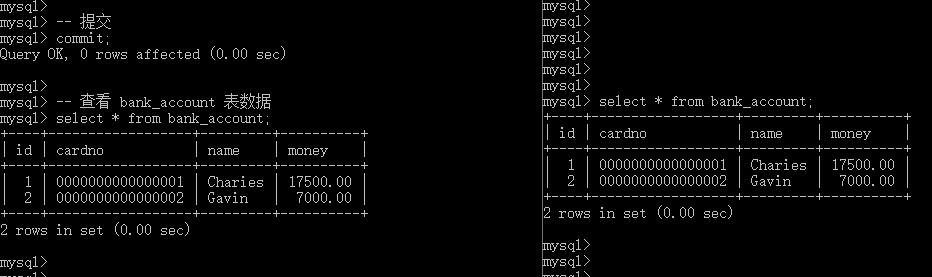
如上图所示，我们并没有开启事务，仅是关闭了自动事务处理，但是我们发现，在我们修改了bank\_account表中数据之后，其结果并不会立即同步到数据库。实际上，这就是关闭了自动事务处理的正常现象。在我们执行commit命令之后，上述操作的结果即可进行同步：

-- 提交

commit;

-- 查看 bank\_account 表数据

select \* from bank\_account;

[](https://camo.githubusercontent.com/900987c36eea07fa5f2cdfab17bdc9a43061ef48/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313131333232303934)

当然，如果我们不执行commit命令，而是执行rollback命令，那么之前的所用操作都会回滚到初始的状态。在此，我们需要注意的是：通常情况下，我们是应该开启自动事务处理的，否则的话，每次操作完成之后都需要我们手动提交，那岂不是要被累死了？

事务特性

事务的特性，可以简单的概括为ACID，具体为：

原子性：Atomic，表示事务的整个操作是一个整体，是不可分割的，要么全部成功，要么全部失败；

一致性：Consistency，表示事务操作的前后，数据表中的数据处于一致状态；

隔离性：Isolation，表示不同的事务操作之间是相互隔离的，互不影响；

持久性：Durability，表示事务一旦提交，将不可修改，永久性的改变数据表中的数据。

对于上述事务的四个特性，其中原子性、一致性、持久性比较容易理解，但是隔离性却需要格外注意。例如，开启两个客户端窗口，分别执行如下 SQL 语句，进行测试：

-- 演示隔离性操作：窗口 1

-- 开始事务

start transaction;

-- 修改 id 为 1 的数据

update bank\_account set money = money + 666 where id = 1;

-- 查看 bank\_account 表数据

select \* from bank\_account;

--------- 万人迷分割线 ---------

-- 演示隔离性操作：窗口 2

-- 开始事务

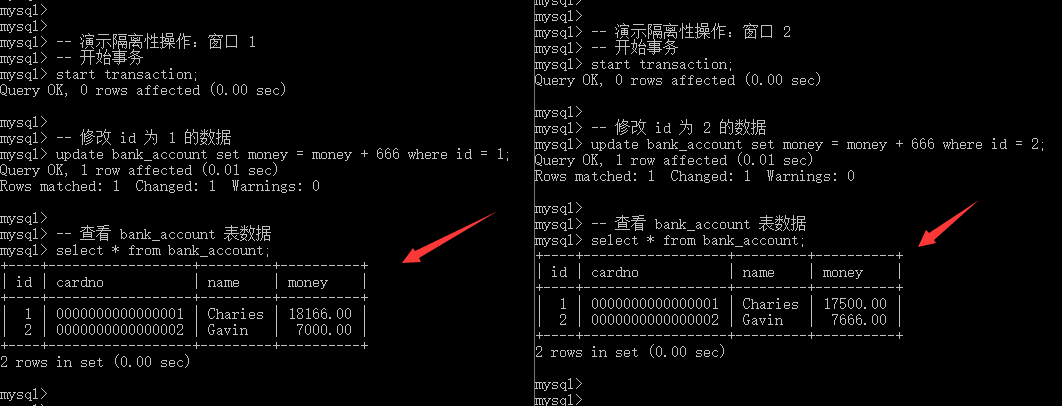
start transaction;

-- 修改 id 为 2 的数据

update bank\_account set money = money + 666 where id = 2;

-- 查看 bank\_account 表数据

select \* from bank\_account;

[](https://camo.githubusercontent.com/b02ad32b381efff972cc7fd8710ae5cc570bde19/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313135383435383732)

如上图所示，其完美的展示了事务隔离性的效果，即窗口 1 的中的事务操作，没有影响到窗口 2 的事务操作；窗口 2 的中的事务操作，也没有影响到窗口 1 的事务操作。But，在我们执行下面的 SQL 语句之后，我们将会看到不同的效果：

-- 演示隔离性操作：窗口 1

-- 开始事务

start transaction;

-- 修改 name 为 Charies 的数据

update bank\_account set money = money + 666 where name = 'Charies';

-- 查看 bank\_account 表数据

select \* from bank\_account;

--------- 万人迷分割线 ---------

-- 演示隔离性操作：窗口 2

-- 开始事务

start transaction;

-- 修改 name 为 Gavin 的数据

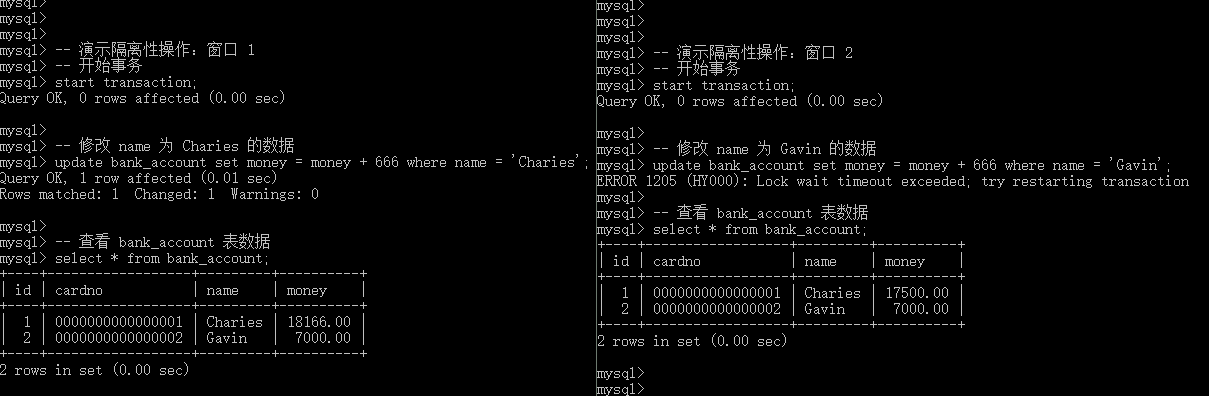
update bank\_account set money = money + 666 where name = 'Gavin';

-- 查看 bank\_account 表数据

select \* from bank\_account;

[](https://camo.githubusercontent.com/4d5f313fd7e81d1a6d1dc10221232b8b908608c3/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313230363138353337)

如上图所示，窗口 1 的事务可以正常执行，但是窗口 2 的事务开启成功，但是在修改数据的时候被“卡”住了，并且在持续一段时间之后，报出了一个 Lock wait timeout exceeded的错误：

[](https://camo.githubusercontent.com/af39fa558920149b679a3eefb0578a5fa83eb7c9/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313731313138313230373539383436)

那么到底是什么原因导致了上述错误的发生呢？这就是涉及到了数据库的另外一个知识点 锁机制 啦！

实际上，MySQL 使用的默认存储引擎是 InnoDB，而 InnoDB 默认使用的锁机制是 行锁（锁住操作的当前行），但是如果在事务操作的过程中，我们没有使用索引字段，那么系统就会自动进行全表检索，也就是其自动将行锁升级为 表锁（锁住操作的当前表）。

现在回想一下，我们在第一次测试的时候，使用的条件id为主键索引，所以两个事务可以表示出很好的隔离性，互不影响；在第二次测试的时候，我们将条件换为name，而name并不是索引字段，因此在第二次测试的时候，窗口 1 的事务使用了表锁，锁住了整张表，而在事务提交或回滚之前，其并不释放锁，所以所有试图修改被锁住表的数据的操作，都会陷入等待状态。等待超时，自然就报错啦！

对于锁机制，在「[基础教程](https://github.com/guobinhit/mysql-tutorial)」篇，我们并不做过多的介绍，在后续的「[性能优化](https://github.com/guobinhit/mysql-tutorial)」篇中在详细的进行讨论。