第19章、事务简介

标签： MySQL是怎样运行的

事务的起源

对于大部分程序员来说，他们的任务就是把现实世界的业务场景映射到数据库世界。比如银行为了存储人们的账户信息会建立一个`account`表：

```
CREATE TABLE account (
    id INT NOT NULL AUTO_INCREMENT COMMENT '自增id',
    name VARCHAR(100) COMMENT '客户名称',
    balance INT COMMENT '余额',
    PRIMARY KEY (id)
) Engine=InnoDB CHARSET=utf8;
```

狗哥和猫爷是一对好基友，他们都到银行开一个账户，他们在现实世界中拥有的资产就会体现在数据库世界的`account`表中。比如现在狗哥有`11`元，猫爷只有`2`元，那么现实中的这个情况映射到数据库的`account`表就是这样：

```
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |      11 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
```

在某个特定的时刻，狗哥猫爷这些家伙在银行所拥有的资产是一个特定的值，这些特定的值也可以被描述为账户在这个特定的时刻现实世界的一个状态。随着时间的流逝，狗哥和猫爷可能陆续进行向账户中存钱、取钱或者向别人转账等操作，这样他们账户中的余额就可能发生变动，每一个操作都相当于现实世界中账户的一次状态转换。数据库世界作为现实世界的一个映射，自然也要进行相应的变动。不变不知道，一变吓一跳，现实世界中一些看似很简单的状态转换，映射到数据库世界却不是那么容易的。比方说有一次猫爷在赌场赌博输了钱，急忙打电话给狗哥要借10块钱，不然那些看场子的就会把自己剁了。现实世界中的狗哥走向了ATM机，输入了猫爷的账号以及10元的转账金额，然后按下确认，狗哥就拔卡走人了。对于数据库世界来说，相当于执行了下边这两条语句：

```
UPDATE account SET balance = balance - 10 WHERE id = 1;
UPDATE account SET balance = balance + 10 WHERE id = 2;
```

但是这里头有个问题，上述两条语句只执行了一条时忽然服务器断电了咋办？把狗哥的钱扣了，

但是没给猫爷转过去，那猫爷还是逃脱不了被砍死的噩运～ 即使对于单独的一条语句，我们前边唠叨Buffer Pool时也说过，在对某个页面进行读写访问时，都会先把这个页面加载到Buffer Pool中，之后如果修改了某个页面，也不会立即把修改同步到磁盘，而只是把这个修改了的页面加到Buffer Pool的flush链表中，在之后的某个时间点才会刷新到磁盘。如果在将修改过的页刷新到磁盘之前系统崩溃了那岂不是猫爷还是要被砍死？或者在刷新磁盘的过程中（只刷新部分数据到磁盘上）系统奔溃了猫爷也会被砍死？

怎么才能保证让可怜的猫爷不被砍死呢？其实再仔细想想，我们只是想让某些数据库操作符合现实世界中状态转换的规则而已，设计数据库的大叔们仔细盘算了盘算，现实世界中状态转换的规则有好几条，待我们慢慢道来。

原子性（Atomicity）

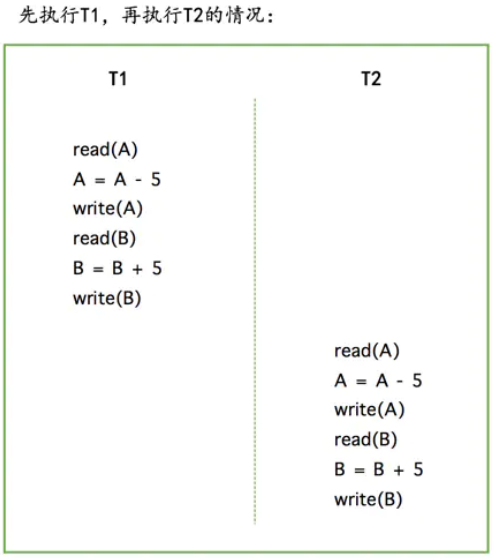
现实世界中转账操作是一个不可分割的操作，也就是说要么压根儿就没转，要么转账成功，不能存在中间的状态，也就是转了一半的这种情况。设计数据库的大叔们把这种要么全做，要么全不做的规则称之为原子性。但是在现实世界中的一个不可分割的操作却可能对应着数据库世界若干条不同的操作，数据库中的一条操作也可能被分解成若干个步骤（比如先修改缓存页，之后再刷新到磁盘等），最要命的是在任何一个可能的时间都可能发生意想不到的错误（可能是数据库本身的错误，或者是操作系统错误，甚至是直接断电之类的）而使操作执行不下去，所以猫爷可能会被砍死。为了保证在数据库世界中某些操作的原子性，设计数据库的大叔需要费一些心机来保证如果在执行操作的过程中发生了错误，把已经做了的操作恢复成没执行之前的样子，这也是我们后边章节要仔细唠叨的内容。

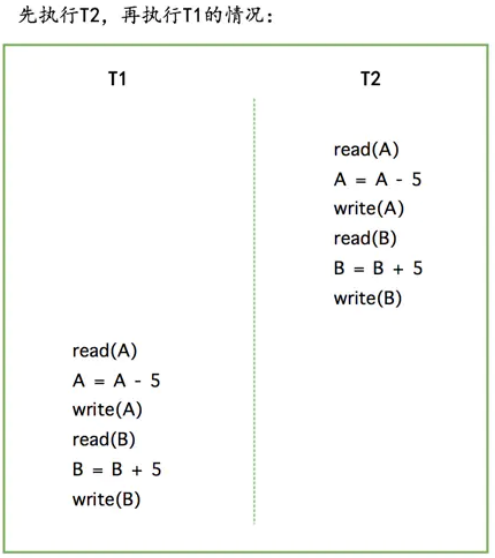
隔离性（Isolation）

现实世界中的两次状态转换应该是互不影响的，比如说狗哥向猫爷同时进行的两次金额为5元的转账（假设可以在两个ATM机上同时操作）。那么最后狗哥的账户里肯定会少10元，猫爷的账户里肯定多了10元。但是到对应的数据库世界中，事情又变的复杂了一些。为了简化问题，我们粗略的假设狗哥向猫爷转账5元的过程是由下边几个步骤组成的：

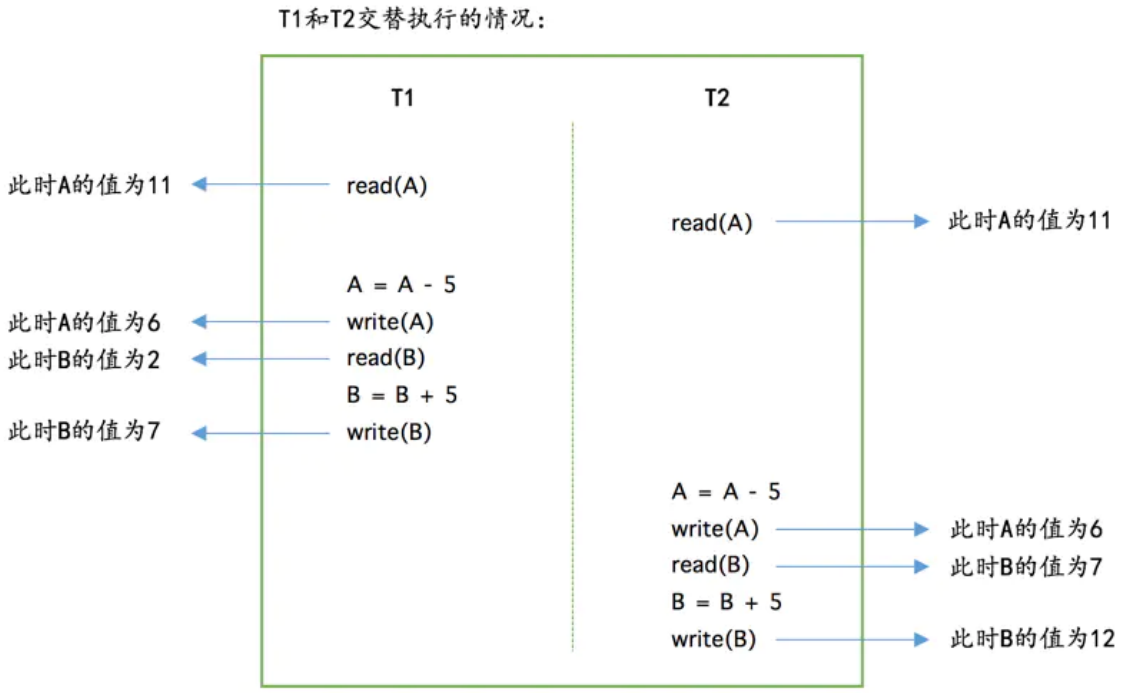
- 步骤一：读取狗哥账户的余额到变量A中，这一步骤简写为`read(A)`。
- 步骤二：将狗哥账户的余额减去转账金额，这一步骤简写为`A = A - 5`。
- 步骤三：将狗哥账户修改过的余额写到磁盘里，这一步骤简写为`write(A)`。
- 步骤四：读取猫爷账户的余额到变量B，这一步骤简写为`read(B)`。
- 步骤五：将猫爷账户的余额加上转账金额，这一步骤简写为`B = B + 5`。
- 步骤六：将猫爷账户修改过的余额写到磁盘里，这一步骤简写为`write(B)`。

我们将狗哥向猫爷同时进行的两次转账操作分别称为`T1`和`T2`，在现实世界中`T1`和`T2`是应该没有关系的，可以先执行完`T1`，再执行`T2`，或者先执行完`T2`，再执行`T1`，对应的数据库操作就像这样：





但是很不幸，真实的数据库中T1和T2的操作可能交替执行，比如这样：



如果按照上图中的执行顺序来进行两次转账的话，最终狗哥的账户里还剩6元钱，相当于只扣了5元钱，但是猫爷的账户里却成了12元钱，相当于多了10元钱，这银行岂不是要亏死了？

所以对于现实世界中状态转换对应的某些数据库操作来说，不仅要保证这些操作以原子性的方式执行完成，而且要保证其它的状态转换不会影响到本次状态转换，这个规则被称之为隔离性。这时设计数据库的大叔们就需要采取一些措施来让访问相同数据（上例中的A账户和B账户）的不同状态转换（上例中的T1和T2）对应的数据库操作的执行顺序有一定规律，这也是我们后边章节要仔细唠叨的内容。

一致性（Consistency）

我们生活的这个世界存在着形形色色的约束，比如身份证号不能重复，性别只能是男或者女，高考的分数只能在0～750之间，人民币面值最大只能是100（现在是2019年），红绿灯只有3种颜色，房价不能为负的，学生要听老师话，吧啦吧啦有点儿扯远了～ 只有符合这些约束的数据才是有效的，比如有个小孩儿跟你说他高考考了1000分，你一听就知道他胡扯呢。数据库世界只是现实世界的一个映射，现实世界中存在的约束当然也要在数据库世界中有所体现。如果数据库中的数据全部符合现实世界中的约束（all defined rules），我们说这些数据就是一致的，或者说符合一致性的。

如何保证数据库中数据的一致性（就是符合所有现实世界的约束）呢？这其实靠两方面的努力：

- 数据库本身能为我们保证一部分一致性需求（就是数据库自身可以保证一部分现实世界的约束永远有效）。

  我们知道MySQL数据库可以为表建立主键、唯一索引、外键、声明某个列为NOT NULL来拒绝NULL值的插入。比如说当我们对某个列建立唯一索引时，如果插入某条记录时该列的值重复了，那么MySQL就会报错并且拒绝插入。除了这些我们已经非常熟悉的保证一致性的功能，MySQL还支持CHECK语法来自定义约束，比如这样：

  ```
  CREATE TABLE account (
      id INT NOT NULL AUTO_INCREMENT COMMENT '自增id',
      name VARCHAR(100) COMMENT '客户名称',
      balance INT COMMENT '余额',
      PRIMARY KEY (id),
      CHECK (balance >= 0)
  );
  ```

  上述例子中的CHECK语句本意是想规定balance列不能存储小于0的数字，对应的现实世界的意思就是银行账户余额不能小于0。但是很遗憾，MySQL仅仅支持CHECK语法，但实际上并没有一点卵用，也就是说即使我们使用上述带有CHECK子句的建表语句来创建account表，那么在后续插入或更新记录时，MySQL并不会去检查CHECK子句中的约束是否成立。

  > 小贴士： 其它的一些数据库，比如SQL Server或者Oracle支持的CHECK语法是有实实在在的作用的，每次进行插入或更新记录之前都会检查一下数据是否符合CHECK子句中指定的约束条件是否成立，如果不成立的话就会拒绝插入或更新。

  虽然CHECK子句对一致性检查没什么卵用，但是我们还是可以通过定义触发器的方式来自定义一些约束条件以保证数据库中数据的一致性。

  > 小贴士： 触发器是MySQL基础内容中的知识，本书是一本MySQL进阶的书籍，如果你不了解触发器，那恐怕要找本基础内容的书籍来看看了。

- 更多的一致性需求需要靠写业务代码的程序员自己保证。

为建立现实世界和数据库世界的对应关系，理论上应该把现实世界中的所有约束都反应到数据库世界中，但是很不幸，在更改数据库数据时进行一致性检查是一个耗费性能的工作，比方说我们为account表建立了一个触发器，每当插入或者更新记录时都会校验一下balance列的值是不是大于0，这就会影响到插入或更新的速度。仅仅是校验一行记录符不符合一致性需求倒也不是什么大问题，有的一致性需求简直变态，比方说银行会建立一张代表账单的表，里边儿记录了每个账户的每笔交易，每一笔交易完成后，都需要保证整个系统的余额等于所有账户的收入减去所有账户的支出。如果在数据库层面实现这个一致性需求的话，每次发生交易时，都需要将所有的收入加起来减去所有的支出，再将所有的账户余额加起来，看看两个值相不相等。这不是搞笑呢么，如果账单表里有几亿条记录，光是这个校验的过程可能就要跑好几个小时，也就是说你在煎饼摊买个煎饼，使用银行卡付款之后要等好几个小时才能提示付款成功，这样的性能代价是完全承受不起的。

现实生活中复杂的一致性需求比比皆是，而由于性能问题把一致性需求交给数据库去解决这是不现实的，所以这个锅就甩给了业务端程序员。比方说我们的account表，我们也可以不建立触发器，只要编写业务的程序员在自己的业务代码里判断一下，当某个操作会将balance列的值更新为小于0的值时，就不执行该操作就好了嘛！

我们前边唠叨的原子性和隔离性都会对一致性产生影响，比如我们现实世界中转账操作完成后，有一个一致性需求就是参与转账的账户的总的余额是不变的。如果数据库不遵循原子性要求，也就是转了一半就不转了，也就是说给狗哥扣了钱而没给猫爷转过去，那最后就是不符合一致性需求的；类似的，如果数据库不遵循隔离性要求，就像我们前边唠叨隔离性时举的例子中所说的，最终狗哥账户中扣的钱和猫爷账户中涨的钱可能就不一样了，也就是说不符合一致性需求了。所以说，数据库某些操作的原子性和隔离性都是保证一致性的一种手段，在操作执行完成后保证符合所有既定的约束则是一种结果。那满足原子性和隔离性的操作一定就满足一致性么？那倒也不一定，比如说狗哥要转账20元给猫爷，虽然在满足原子性和隔离性，但转账完成了之后狗哥的账户的余额就成负的了，这显然是不满足一致性的。那不满足原子性和隔离性的操作就一定不满足一致性么？这也不一定，只要最后的结果符合所有现实世界中的约束，那么就是符合一致性的。

持久性（Durability）

当现实世界的一个状态转换完成后，这个转换的结果将永久的保留，这个规则被设计数据库的大叔们称为持久性。比方说狗哥向猫爷转账，当ATM机提示转账成功了，就意味着这次账户的状态转换完成了，狗哥就可以拔卡走人了。如果当狗哥走掉之后，银行又把这次转账操作给撤销掉，恢复到没转账之前的样子，那猫爷不就惨了，又得被砍死了，所以这个持久性是非常重要的。

当把现实世界的状态转换映射到数据库世界时，持久性意味着该转换对应的数据库操作所修改的数据都应该在磁盘上保留下来，不论之后发生了什么事故，本次转换造成的影响都不应该被丢失掉（要不然猫爷还是会被砍死）。

事务的概念

为了方便大家记住我们上边唠叨的现实世界状态转换过程中需要遵守的4个特性，我们把原子性

（Atomicity）、隔离性（Isolation）、一致性（Consistency）和持久性（Durability）这四个词对应的英文单词首字母提取出来就是A、I、C、D，稍微变换一下顺序可以组成一个完整的英文单词：ACID。想必大家都是学过初高中英语的，ACID是英文酸的意思，以后我们提到ACID这个词儿，大家就应该想到原子性、一致性、隔离性、持久性这几个规则。另外，设计数据库的大叔为了方便起见，把需要保证原子性、隔离性、一致性和持久性的一个或多个数据库操作称之为一个事务（英文名是：transaction）。

我们现在知道事务是一个抽象的概念，它其实对应着一个或多个数据库操作，设计数据库的大叔根据这些操作所执行的不同阶段把事务大致上划分成了这么几个状态：

- 活动的（active）

  事务对应的数据库操作正在执行过程中时，我们就说该事务处在活动的状态。

- 部分提交的（partially committed）

  当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并没有刷新到磁盘时，我们就说该事务处在部分提交的状态。

- 失败的（failed）

  当事务处在活动的或者部分提交的状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在失败的状态。

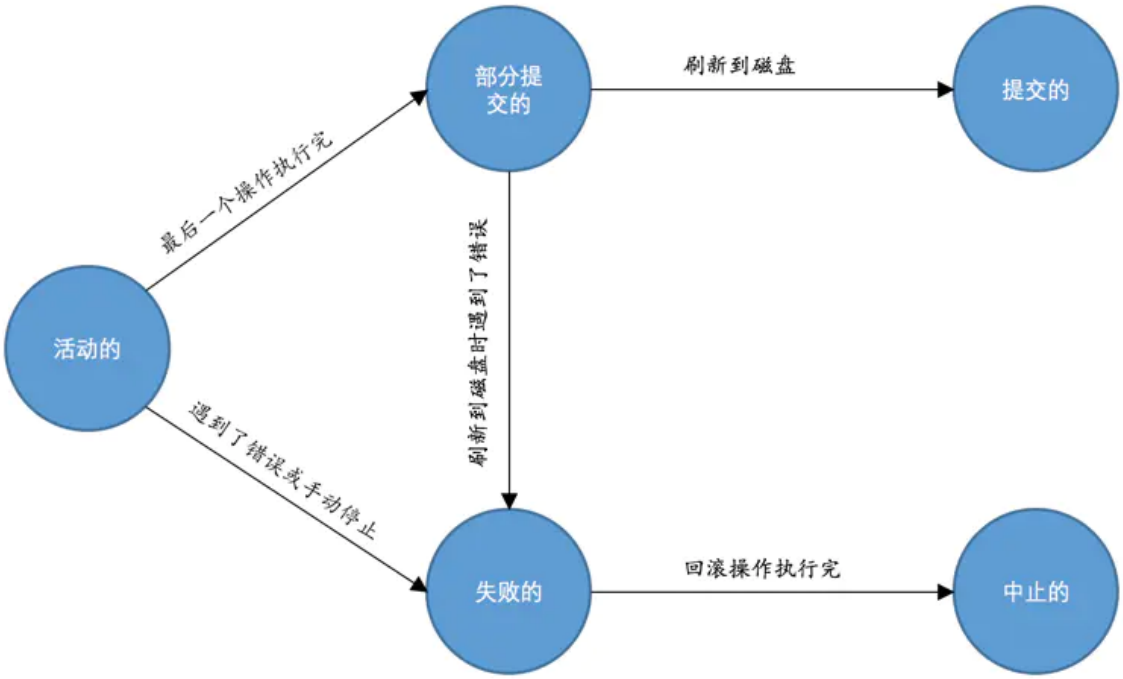
- 中止的（aborted）

  如果事务执行了半截而变为失败的状态，比如我们前边唠叨的狗哥向猫爷转账的事务，当狗哥账户的钱被扣除，但是猫爷账户的钱没有增加时遇到了错误，从而当前事务处在了失败的状态，那么就需要把已经修改的狗哥账户余额调整为未转账之前的金额，换句话说，就是要撤销失败事务对当前数据库造成的影响。书面一点的话，我们把这个撤销的过程称之为回滚。当回滚操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了中止的状态。

- 提交的（committed）

  当一个处在部分提交的状态的事务将修改过的数据都同步到磁盘上之后，我们就可以说该事务处在了提交的状态。

随着事务对应的数据库操作执行到不同阶段，事务的状态也在不断变化，一个基本的状态转换图如下所示：



从图中大家也可以看出了，只有当事务处于提交的或者中止的状态时，一个事务的生命周期才算是结束了。对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，对于处于中止状态的事务，该事务对数据库所做的所有修改都会被回滚到没执行该事务之前的状态。

> 小贴士： 此贴士处纯属扯犊子，与正文没啥关系，纯属吐槽。大家知道我们的计算机术语基本上全是从英文翻译成中文的，事务的英文是transaction，英文直译就是交易，买卖的意思，交易就是买的人付钱，卖的人交货，不能付了钱不交货，交了货不付钱把，所以交易本身就是一种不可分割的操作。不知道是哪位大神把transaction翻译成了事务（我想估计是他们也想不出什么更好的词儿，只能随便找一个了），事务这个词儿完全没有交易、买卖的意思，所以大家理解起来也会比较困难，外国人理解transaction可能更好理解一点吧~

MySQL中事务的语法

我们说事务的本质其实只是一系列数据库操作，只不过这些数据库操作符合ACID特性而已，那么MySQL中如何将某些操作放到一个事务里去执行的呢？我们下边就来重点唠叨唠叨。

开启事务

我们可以使用下边两种语句之一来开启一个事务：

- `BEGIN [WORK];`

  `BEGIN`语句代表开启一个事务，后边的单词`WORK`可有可无。开启事务后，就可以继续写若干条语句，这些语句都属于刚刚开启的这个事务。

  ```
  mysql> BEGIN;
  ```

```
Query OK, 0 rows affected (0.00 sec)

mysql> 加入事务的语句...
```

- `START TRANSACTION;`

  `START TRANSACTION`语句和`BEGIN`语句有着相同的功效，都标志着开启一个事务，比如这样：

  ```
  mysql> START TRANSACTION;
  Query OK, 0 rows affected (0.00 sec)

  mysql> 加入事务的语句...
  ```

  不过比`BEGIN`语句牛逼一点儿的是，可以在`START TRANSACTION`语句后边跟随几个`修饰符`，就是它们几个：

  - `READ ONLY`：标识当前事务是一个只读事务，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。

    > 小贴士：其实只读事务中只是不允许修改那些其他事务也能访问到的表中的数据，对于临时表来说（我们使用CREATE TMEPORARY TABLE创建的表），由于它们只能在当前会话中可见，所以只读事务其实也是可以对临时表进行增、删、改操作的。

  - `READ WRITE`：标识当前事务是一个读写事务，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。

  - `WITH CONSISTENT SNAPSHOT`：启动一致性读（先不用关心啥是个一致性读，后边的章节才会唠叨）。

  比如我们想开启一个只读事务的话，直接把`READ ONLY`这个修饰符加在`START TRANSACTION`语句后边就好，比如这样：

  ```
  START TRANSACTION READ ONLY;
  ```

  如果我们想在`START TRANSACTION`后边跟随多个`修饰符`的话，可以使用逗号将`修饰符`分开，比如开启一个只读事务和一致性读，就可以这样写：

  ```
  START TRANSACTION READ ONLY, WITH CONSISTENT SNAPSHOT;
  ```

  或者开启一个读写事务和一致性读，就可以这样写：

  ```
  START TRANSACTION READ WRITE, WITH CONSISTENT SNAPSHOT
  ```

  不过这里需要大家注意的一点是，`READ ONLY`和`READ WRITE`是用来设置所谓的事务`访问模式`的，就是以只读还是读写的方式来访问数据库中的数据，一个事务的访问模式不能同时既设置为`只读`的也设置为`读写`的，所以我们不能同时把`READ ONLY`和`READ WRITE`放到`START TRANSACTION`语句后边。另外，如果我们不显式指定事务的访问模式，那么该事务的访问模式就是`读写`模

式。

提交事务

开启事务之后就可以继续写需要放到该事务中的语句了，当最后一条语句写完了之后，我们就可以提交该事务了，提交的语句也很简单：

```
COMMIT [WORK]
```

COMMIT语句就代表提交一个事务，后边的WORK可有可无。比如我们上边说狗哥给猫爷转10元钱其实对应MySQL中的两条语句，我们就可以把这两条语句放到一个事务中，完整的过程就是这样：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE account SET balance = balance + 10 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

手动中止事务

如果我们写了几条语句之后发现上边的某条语句写错了，我们可以手动的使用下边这个语句来将数据库恢复到事务执行之前的样子：

```
ROLLBACK [WORK]
```

ROLLBACK语句就代表中止并回滚一个事务，后边的WORK可有可无类似的。比如我们在写狗哥给猫爷转账10元钱对应的MySQL语句时，先给狗哥扣了10元，然后一时大意只给猫爷账户上增加了1元，此时就可以使用ROLLBACK语句进行回滚，完整的过程就是这样：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> UPDATE account SET balance = balance + 1 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

这里需要强调一下，ROLLBACK语句是我们程序员手动的去回滚事务时才去使用的，如果事务在执行过程中遇到了某些错误而无法继续执行的话，事务自身会自动的回滚。

> 小贴士： 我们这里所说的开启、提交、中止事务的语法只是针对使用黑框框时通过mysql客户端程序与服务器进行交互时控制事务的语法，如果大家使用的是别的客户端程序，比如JDBC之类的，那需要参考相应的文档来看看如何控制事务。

支持事务的存储引擎

MySQL中并不是所有存储引擎都支持事务的功能，目前只有InnoDB和NDB存储引擎支持（NDB存储引擎不是我们的重点），如果某个事务中包含了修改使用不支持事务的存储引擎的表，那么对该使用不支持事务的存储引擎的表所做的修改将无法进行回滚。比方说我们有两个表，tbl1使用支持事务的存储引擎InnoDB，tbl2使用不支持事务的存储引擎MyISAM，它们的建表语句如下所示：

```
CREATE TABLE tbl1 (
    i int
) engine=InnoDB;

CREATE TABLE tbl2 (
    i int
) ENGINE=MyISAM;
```

我们看看先开启一个事务，写一条插入语句后再回滚该事务，tbl1和tbl2的表现有什么不同：

```
mysql> SELECT * FROM tbl1;
Empty set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tbl1 VALUES(1);
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM tbl1;
Empty set (0.00 sec)
```

可以看到，对于使用支持事务的存储引擎的tbl1表来说，我们在插入一条记录再回滚后，tbl1就恢复到没有插入记录时的状态了。再看看tbl2表的表现：

```
mysql> SELECT * FROM tbl2;
Empty set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tbl2 VALUES(1);
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SELECT * FROM tbl2;
+------+
| i    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

可以看到，虽然我们使用了ROLLBACK语句来回滚事务，但是插入的那条记录还是留在了tbl2表中。

自动提交

MySQL中有一个系统变量autocommit：

```
mysql> SHOW VARIABLES LIKE 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.01 sec)
```

可以看到它的默认值为ON，也就是说默认情况下，如果我们不显式的使用START TRANSACTION或者BEGIN语句开启一个事务，那么每一条语句都算是一个独立的事务，这种特性称之为事务的自动

提交。假如我们在狗哥向猫爷转账10元时不以START TRANSACTION或者BEGIN语句显式的开启一个事务，那么下边这两条语句就相当于放到两个独立的事务中去执行：

```
UPDATE account SET balance = balance - 10 WHERE id = 1;
UPDATE account SET balance = balance + 10 WHERE id = 2;
```

当然，如果我们想关闭这种自动提交的功能，可以使用下边两种方法之一：

- 显式的的使用START TRANSACTION或者BEGIN语句开启一个事务。

  这样在本次事务提交或者回滚前会暂时关闭掉自动提交的功能。

- 把系统变量autocommit的值设置为OFF，就像这样：

  ```
  SET autocommit = OFF;
  ```

  这样的话，我们写入的多条语句就算是属于同一个事务了，直到我们显式的写出COMMIT语句来把这个事务提交掉，或者显式的写出ROLLBACK语句来把这个事务回滚掉。

隐式提交

当我们使用START TRANSACTION或者BEGIN语句开启了一个事务，或者把系统变量autocommit的值设置为OFF时，事务就不会进行自动提交，但是如果我们输入了某些语句之后就会悄悄的提交掉，就像我们输入了COMMIT语句了一样，这种因为某些特殊的语句而导致事务提交的情况称为隐式提交，这些会导致事务隐式提交的语句包括：

- 定义或修改数据库对象的数据定义语言（Data definition language，缩写为：DDL）。

  所谓的数据库对象，指的就是数据库、表、视图、存储过程等等这些东西。当我们使用CREATE、ALTER、DROP等语句去修改这些所谓的数据库对象时，就会隐式的提交前边语句所属于的事务，就像这样：

  ```
  BEGIN;

  SELECT ... # 事务中的一条语句
  UPDATE ... # 事务中的一条语句
  ... # 事务中的其它语句

  CREATE TABLE ... # 此语句会隐式的提交前边语句所属于的事务
  ```

- 隐式使用或修改mysql数据库中的表

  当我们使用ALTER USER、CREATE USER、DROP USER、GRANT、RENAME USER、REVOKE、SET PASSWORD等语句时也会隐式的提交前边语句所属于的事务。

- 事务控制或关于锁定的语句

  当我们在一个事务还没提交或者回滚时就又使用START TRANSACTION或者BEGIN语句开启了另一

个事务时，会隐式的提交上一个事务，比如这样：

```
BEGIN;

SELECT ... # 事务中的一条语句
UPDATE ... # 事务中的一条语句
... # 事务中的其它语句

BEGIN; # 此语句会隐式的提交前边语句所属于的事务
```

或者当前的autocommit系统变量的值为OFF，我们手动把它调为ON时，也会隐式的提交前边语句所属的事务。

或者使用LOCK TABLES、UNLOCK TABLES等关于锁定的语句也会隐式的提交前边语句所属的事务。

- 加载数据的语句

  比如我们使用LOAD DATA语句来批量往数据库中导入数据时，也会隐式的提交前边语句所属的事务。

- 关于MySQL复制的一些语句

  使用START SLAVE、STOP SLAVE、RESET SLAVE、CHANGE MASTER TO等语句时也会隐式的提交前边语句所属的事务。

- 其它的一些语句

  使用ANALYZE TABLE、CACHE INDEX、CHECK TABLE、FLUSH、 LOAD INDEX INTO CACHE、OPTIMIZE TABLE、REPAIR TABLE、RESET等语句也会隐式的提交前边语句所属的事务。

> 小贴士： 上边提到的一些语句，如果你都认识并且知道是干嘛用的那再好不过了，不认识也不要气馁，这里写出来只是为了内容的完整性，把可能会导致事务隐式提交的情况都列举一下，具体每个语句都是干嘛用的等我们遇到了再说哈。

保存点

如果你开启了一个事务，并且已经敲了很多语句，忽然发现上一条语句有点问题，你只好使用ROLLBACK语句来让数据库状态恢复到事务执行之前的样子，然后一切从头再来，总有一种一夜回到解放前的感觉。所以设计数据库的大叔们提出了一个保存点（英文：savepoint）的概念，就是在事务对应的数据库语句中打几个点，我们在调用ROLLBACK语句时可以指定会滚到哪个点，而不是回到最初的原点。定义保存点的语法如下：

```
SAVEPOINT 保存点名称;
```

当我们想回滚到某个保存点时，可以使用下边这个语句（下边语句中的单词WORK和SAVEPOINT是可有可无的）：

```
ROLLBACK [WORK] TO [SAVEPOINT] 保存点名称;
```

不过如果ROLLBACK语句后边不跟随保存点名称的话，会直接回滚到事务执行之前的状态。

如果我们想删除某个保存点，可以使用这个语句：

```
RELEASE SAVEPOINT 保存点名称;
```

下边还是以狗哥向猫爷转账10元的例子展示一下保存点的用法，在执行完扣除狗哥账户的钱10元的语句之后打一个保存点：

```
mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |      11 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SAVEPOINT s1;    # 一个保存点
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |       1 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)

mysql> UPDATE account SET balance = balance + 1 WHERE id = 2; # 更新错了
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> ROLLBACK TO s1;  # 回滚到保存点s1处
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |       1 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)
```