Hands-on 2: Write Ahead Log System (2)

相关知识

Checkpoints

from： **CSE-08-graph2stream P52**

Failure detection: master pings each worker periodically via heartbeat

- Similar in MapReduce

Failure handling

- Checkpointing: at the beginning of a superstep, checkpoint the graph data to a persistent storage (e.g., GFS)
- After failure, can recovery from checkpoint & re-compute
- For either worker or master
- The frequencies of checkpointing is determined by the mean time to failure
- Can measure MTTF in advance

checkpoint是一个内部事件，这个事件激活以后会触发数据库写进程(DBWR)将数据缓冲(**DATA**BUFFER CACHE)中的脏数据块写出到数据文件中。

Following hands-on 1, this hands-on will play with the checkpoint using wal-sys. Start wal-sys with a reset:

```
$ ./wal-sys.py -reset
```

and run the following commands:

```
begin 1
create_account 1 studentA 1000
commit 1
end 1
begin 2
create_account 2 studentB 2000
credit_account 2 studentA 100
begin 3
create_account 3 studentC 3000
checkpoint
commit 2
debit_account 3 studentC 100
show_state
crash
```

Note: we will ask you to crash and recover the system a few times, but you should not run the sequence commands again.

Examine the "LOG" output file. In particular, inspect the CHECKPOINT entry. Also, count the number of entries in the "LOG" file. Run wal-sys again to recover the database.

1. input the command

```
➜  handin git:(lab2) ✗ ./wal-sys.py -reset <cmd2.in
Opening new log
> Beginning id: 1
> Initializing account studentA to 1000
> Committing id: 1
```

```
> Ending id: 1
> Beginning id: 2
> Initializing account studentB to 2000
> Crediting account studentA by 100
> Beginning id: 3
> Initializing account studentC to 3000
> Doing checkpoint
> Committing id: 2
> Debiting account studentC by 100
>
-----------------------
On-disk DB contents:
Account: studentA Value: 1000
-----------------------

-----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
-----------------------
> crashing ...
```

2. see the log

```
➜ vim LOG
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
```

3. recover stage 1

```
➜  handin git:(lab2) ✗ ./wal-sys.py
Recovering the database ..
Starting rollback ...
  The log was rolled back 8 lines
Rollback done
Winners: id: 2  Losers: id: 3  Done: id: 1
Starting forward scan ...
  REDOING: type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
  REDOING: type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
  Logging END records for winners
Forward scan done
Recovery done
> show_statte
invalid command: show_statte
> show_state

----------------------
On-disk DB contents:
Account: studentB Value: 2000
Account: studentA Value: 1100
----------------------

----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
type: END action_id: 2
----------------------
>
```

Note down the action_ids of "Winners", "Losers", and "Done". Use the show_state command to look at the recovered database and verify that the database recovered correctly. Crash the system, and then run wal-sys again to recover the database a second time.

第二次recover

```
➜  handin git:(lab2) ✗ ./wal-sys.py
Recovering the database ..
Starting rollback ...
  The log was rolled back 4 lines
Rollback done
Winners:   Losers: id: 3  Done: id: 1 id: 2
Starting forward scan ...
  Logging END records for winners
Forward scan done
Recovery done
> show_state

-----------------------
On-disk DB contents:
Account: studentB Value: 2000
Account: studentA Value: 1100
-----------------------

-----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
type: END action_id: 2
-----------------------
>
```

Question 1

During checkpoint, wal-sys divides actions into three types: "PENDING", "COMMITTED" and "DONE", what is the meaning of these types?

Answer:

PENDING: 在这个checkpoint 之前还没有commit的action

COMMITTED： 在这个checkpoint之前commit 但是还没有end写入磁盘的action

DONE: 在这个checkpoint 之前已经写入磁盘的end的action

Question 2

What is the relationship between the action categories during checkpoint ("PENDING", "COMMITTED" and "DONE") and action categories during recovery ("Winners", "Losers", and "Done")?

Answer:

checkpoint 的三种情况：
PENDING：事务在 checkpoint 之前没有commit

COMMITED： 事务在checkpoint 之前commit但是没有end

DONE： 事务在checkpoin 之前end

recovery的三种情况

winner: 事务在checkpoint 之后，crash之前commit，还没有end.

loser: 事务在checkpoint 之后，crash之前没有commit

done: 事务在crash之前已经end

两者之间的关联：其实就是如上所述的时间顺序的关联。

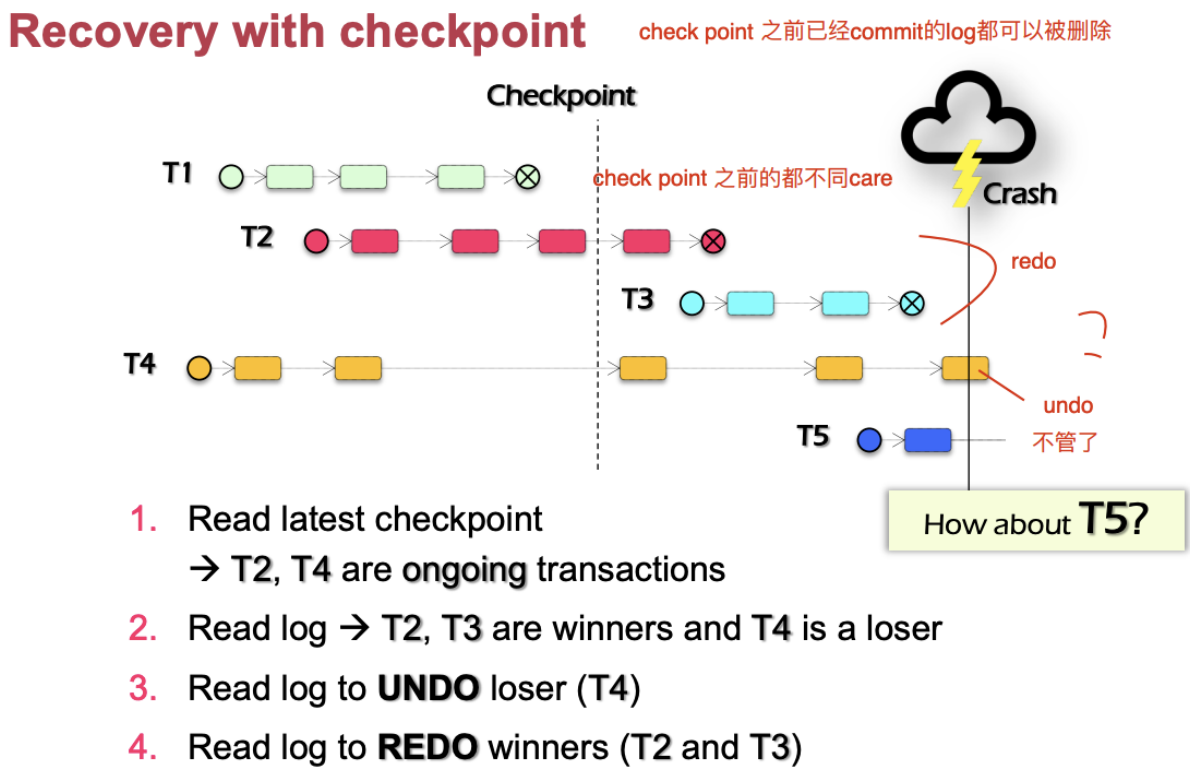
因为checkpoint 发生在crash之前，如果在checkpoint时已经Done了，那发生crash之前也一定done。如这里的**id:1** 。

如果在checkpoint 之前是commit， 分两种情况，一种是在checkpoint和crash之间没有end写入磁盘，那就是winner。如果在中间end了，那就是done.

如果在checkpoint 之前是pending， 分三种情况，一种是在checkpoint和crash之间没有commit , 是loser，**如id: 3**。如果在中间commit 是winner **如id:2**。在中间end, 是done.

下面为checkpoint 和recovery阶段的提示信息，和分析吻合

```
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
Winners: id: 2  Losers: id: 3  Done: id: 1
```



Question 3

How many lines were rolled back? What is the advantage of using checkpoints?

观察recovery的提示如下，8行.

```
The log was rolled back 8 lines
```

```
----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
type: END action_id: 2
----------------------
>
```

checkpoint 的优势

1、缩短数据库的恢复时间；2、缓冲池不够用时，将脏页刷新到磁盘；3、log 不可用时，刷新脏页。这样可以缩小log占用的空间。

- 当数据库发生宕机时，数据库不需要重做所有的日志，因为Checkpoint之前的页都已经刷新回磁盘。数据库只需对Checkpoint后的重做日志进行恢复，这样就大大缩短了恢复的时间。
- 当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Checkpoint，将脏页也就是页的新版本刷回磁盘。（这个部分是针对mysql这样有缓冲池的设计，本次python 脚本没有涉及可以忽略）
- check point 之前已经结束的事务的log不需要，可以删除。事务的log不是无限长的，这样就可以减少空间，重复利用。

Question 4

Does the second run of the recovery procedure restore "DB" to the same state as the first run? What is this property called?

一样

recover一次和recover两次效果一样 → 幂等性

Question 5

Compare the `action_ids` of "Winners", "Losers", and "Done" from the second recovery with those from the first. The lists are different. How does the recovery procedure guarantee the property from Question 4 even though the recovery procedure can change? (Hint: Examine the "LOG" file).

观察发现，在第一次recover 的时候，log多了一条

`type: END action_id: 2` 也就是说winner在recover的时候redo，并且变成了end的事务被写入了磁盘。然而id：3 的数据是loser，系统undo，于是恢复到最早的undo日志就是NULL的值。在第三次执行的恢复的时候，结果是

`Winners: Losers: id: 3 Done: id: 1 id: 2`

因为id：1 ，2 的事务已经end,已经写入了磁盘，所以和第二次一样。同时id：3 还是loser，所以任然UNDO到null.

因为Winners redo以后事务结束就变成Done,写入数据库，这对数据库的改变是永久性的。所以就算又crash也不会发生变化

Question 6 (Optional)

Wal-sys has a hitherto unmentioned option: if you type wal-sys -undo it will perform undo logging and undo recovery. Try the above sequences again with undo logging to see what changes.

initial stage

```
➜  handin ./wal-sys.py -reset -undo <cmd2.in
Opening new log
> Beginning id: 1
> Initializing account studentA to 1000
> Committing id: 1
> Ending id: 1
> Beginning id: 2
> Initializing account studentB to 2000
> Crediting account studentA by 100
> Beginning id: 3
> Initializing account studentC to 3000
> Doing checkpoint
```

```
> Committing id: 2
> Debiting account studentC by 100
>
----------------------
On-disk DB contents:
Account: studentC Value: 2900
Account: studentB Value: 2000
Account: studentA Value: 1100
----------------------

----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
----------------------
> crashing ...
```

区别：

之前的DB的内容只有student A , 这里三个student都出现了。

recover stage 1

```
➜  handin ./wal-sys.py -undo

Recovering the database ..
Starting rollback ...
  UNDOING: type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
  UNDOING: type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
  The log was rolled back 5 lines
Rollback done
Winners: id: 2  Losers: id: 3  Done: id: 1
 Logging STATUS records for losers
Recovery done
> show_state

----------------------
On-disk DB contents:
Account: studentB Value: 2000
```

```
Account: studentA Value: 1100
----------------------

----------------------
LOG contents:
type: START action_id: 1
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
type: OUTCOME action_id: 3 status ABORTED
----------------------
>
```

区别：

回滚的机制发生了变化

之前只做redo,现在只做undo, 之前回滚8条，现在回滚5条, 也就是从 id:3 开始的五条。

之前需要对winnder log END record.现在需要对loser log status, 即为ABORTED

recover stage 2

```
➜  handin ./wal-sys.py -undo

Recovering the database ..
Starting rollback ...
  The log was rolled back 4 lines
Rollback done
Winners: id: 3 id: 2  Losers:   Done: id: 1
 Logging STATUS records for losers
Recovery done
> show_state

----------------------
On-disk DB contents:
Account: studentB Value: 2000
Account: studentA Value: 1100
----------------------

----------------------
LOG contents:
type: START action_id: 1
```

```
type: UPDATE action_id: 1 variable: studentA redo: "1000" undo: NULL
type: OUTCOME action_id: 1 status: COMMITTED
type: END action_id: 1
type: START action_id: 2
type: UPDATE action_id: 2 variable: studentB redo: "2000" undo: NULL
type: UPDATE action_id: 2 variable: studentA redo: "1100" undo: "1000"
type: START action_id: 3
type: UPDATE action_id: 3 variable: studentC redo: "3000" undo: NULL
type: CHECKPOINT PENDING: id: 3 id: 2 COMMITTED:  DONE: id: 1
type: OUTCOME action_id: 2 status: COMMITTED
type: UPDATE action_id: 3 variable: studentC redo: "2900" undo: "3000"
type: OUTCOME action_id: 3 status ABORTED
----------------------
>
```

区别：

Winners: Losers: id: 3 Done: id: 1 id: 2 → Winners: id: 3 id: 2 Losers: Done: id: 1

这个说明由于第一次的log 更新了id：3的状态，所以第二次recover的时候，id:3 变成了winner

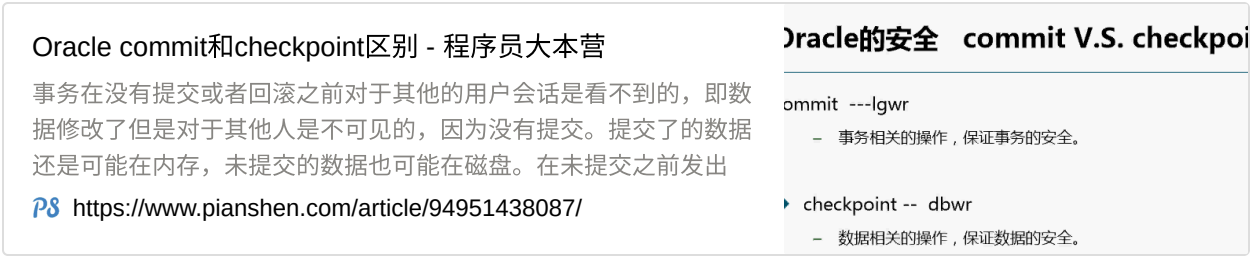
总结：

1. -undo 表示每次操作写log的同时都会写磁盘，通过undo执行recovery
2. 之前对于winner需要做redo把数据刷到磁盘， 现在由于最新的数据在磁盘上了，所以需要把不应该写入的loser的数据给undo了。
3. 第一种情况下多次crash不会改变loser和winner的状态，但是-undo的时候因为有了ABORTED，使得loser在第二次crash恢复的时候变成了winner。

reference

commit 和checkpoint 的区别

讲的很好



事务在没有提交或者回滚之前对于其他的用户会话是看不到的，即数据修改了但是对于其他人是不可见的，因为没有提交。提交了的数据还是可能在内存，未提交的数据也可能在磁盘。在未提交之前发出alter system checkpoint，那么所有修改了的数据块都写到磁盘上面了，虽然未提交，但是数据还是写到磁盘上面了，因为未提交，其他会话依旧看不到数据修改的变化。

对于一个事务的提交与否和在磁盘，内存没有任何关系。对于commit来说是来保证数据的永久的改变，这些改变在磁盘还是在内存改变都不重要，总之就是改变了。

Checkpoint就是将内存凡是修改过的数据块就写到磁盘上面，修改的数据块有两种情况，一种是修改完提交，一种是修改完未提交。所以checkpoint只管将修改了的数据块写到磁盘上面。

事务发出一个commit之后，这个时候Oracle就认为将这个数据块永久的改变了。commit表示事务的结束，事务的结束就意味着别人对操数据的结果可见。哪怕修改了100W条记录没有commit，对于其他的用户依然查看不到修改了的数据。commit标识事务的结束，标识修改数据的生效，生效是在内存生效还是磁盘生效都不重要。

总结：commit就是让事务提交，让事务生效，让修改过后的数据对于其他用户可见。如果生效的数据在磁盘上面，别的用户查的时候就去磁盘上读取出来，如果生效的数据在内存里面，那么在内存里面就是可以看到的。

commit之后不是将修改过后的脏数据块写到磁盘上面，redo log开始是放在内存里面的，commit之后会强迫log buffer里面的redo log无条件的必须写到磁盘上面。只有磁盘写成功了commit才会返回给客户端提交完成的信息。在Oracle里面数据是由redo来保护的，为什么不将修改了的数据直接写到磁盘上面，为什么还要让redo去写。因为写redo速度快，redo是顺序写的，是从一个文件从头写到尾。而要将数据块写到磁盘上面，先要去磁盘上面找数据块的位置，然后再写入，这样非常耗时。一旦redo信息写到磁盘上面就没有问题了，即使数据块丢失了，也可以通过redo找回来。

redo一写到磁盘上，这个数据就安全了，既然安全了为什么还要checkpoint呢？

第一：就是给别人让位置，脏数据会越来越多的，最后内存会放不下，所以脏数据最后还是得刷到磁盘上面给其他新产生的脏数据让出位置。

第二：在恢复的时候减少时间，redo是可以将数据保护起来，如果有大量的数据都在内存里面，比如说有好几个G的数据在内存里面没有刷到磁盘上面，数据库坏了恢复的时候因为大量数据的丢失到导致恢复的时候需要很长时间。所以checkpoint就是将一些数据及时的写到磁盘上面，一旦写到磁盘上面就不需要恢复了。

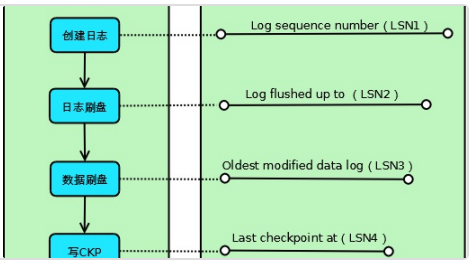
那么是否要通过checkpoint经常来将脏数据写到磁盘上面呢？效率的问题，因为频繁的写到磁盘上面是随机写I/O，效率低。

【mysql】关于checkpoint机制

【mysql】关于checkpoint机制

一、简介 思考一下这个场景：如果重做日志可以无限地增大，同时缓冲池也足够大，那么是不需要将缓冲池中页的新版本刷新回磁盘。因为当发生宕机时，完全可以通过重做日志来恢复整个数据库系统中

知 https://zhuanlan.zhihu.com/p/86601861



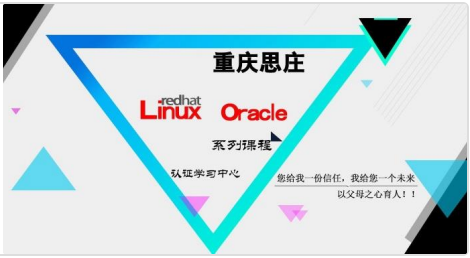
chekcpoint 之后所有的redo log都可以删掉了。

redo和undo区别讨论

redo和undo区别讨论 - 程序员大本营

英文解释： 名词：两种流程， redo重做流程， undo撤销还原流程；或则是redo日志与undo段的简称 动词：redo即重做，undo即撤销还原。 翻译有时候为了简单，常把动词和名称混用。不同场景不同的

PS https://www.pianshen.com/article/1839673328/



事务的四个属性

原子性（Atomicity）原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

一致性（Consistency）事务必须使**数据库**从一个一致性状态变换到另外一个一致性状态。

隔离性（Isolation）事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

持久性（Durability）持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

https://www.cnblogs.com/1995hxt/p/5550855.html