课堂主题

Mysql元数据锁、行锁、MySQL事务介绍、InnoDB架构

课堂目标

理解MySQL元数据锁的意义和使用场景

理解MySQL行锁的意义和使用场景

掌握MySQL记录锁和间隙锁的使用区别

掌握死锁的原理和死锁场景

理解事务的概念和四大特征（ACID）

掌握InnoDB的架构和组件作用

理解预写机制、双写机制、RedoLog的作用和日志落盘

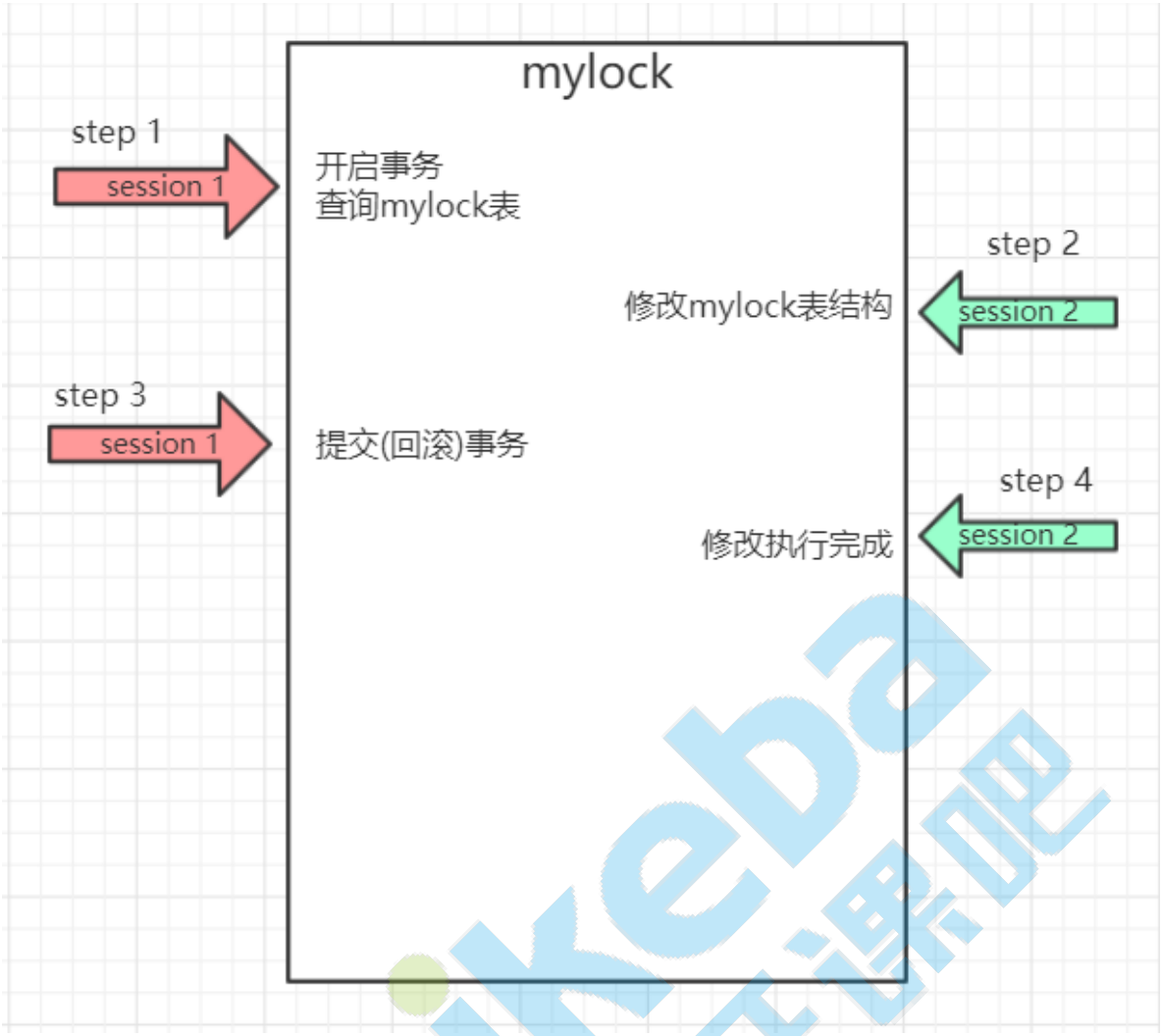
一、元数据锁

元数据锁介绍

MDL (metaDataLock) 元数据：表结构

在 MySQL 5.5 版本中引入了 MDL，当对一个表做增删改查操作的时候，加 MDL 读锁；当要对表做结构变更操作的时候，加 MDL 写锁。

元数据锁演示



session1（Navicat）、session2（mysql）

```
1、session1: begin;--开启事务
           select * from mylock;--加MDL读锁
2、session2: alter table mylock add f int; -- 修改阻塞
3、session1: commit; --提交事务 或者 rollback 释放读锁
4、session2: Query OK, 0 rows affected (38.67 sec)  --修改完成
           Records: 0  Duplicates: 0  Warnings: 0
```

二、行级锁

行级锁介绍

InnoDB存储引擎实现

InnoDB的行级锁，按照锁定范围来说，分为三种：

记录锁（Record Locks）:锁定索引中一条记录。 主键指定 where id=3

间隙锁（Gap Locks）: 锁定记录前、记录中、记录后的行 RR隔离级 （可重复读）-- MySQL默认隔离级

Next-Key 锁: 记录锁 + 间隙锁

行级锁分类

按照功能来说，分为两种：

共享读锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

```
SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE    -- 共享读锁   手动添加
select * from table    -- 无锁
```

排他写锁（X）：允许获得排他写锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁（不是读）和排他写锁。

1、自动加 DML

对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X）;

2、手动加

```
SELECT * FROM table_name WHERE ... FOR UPDATE
```

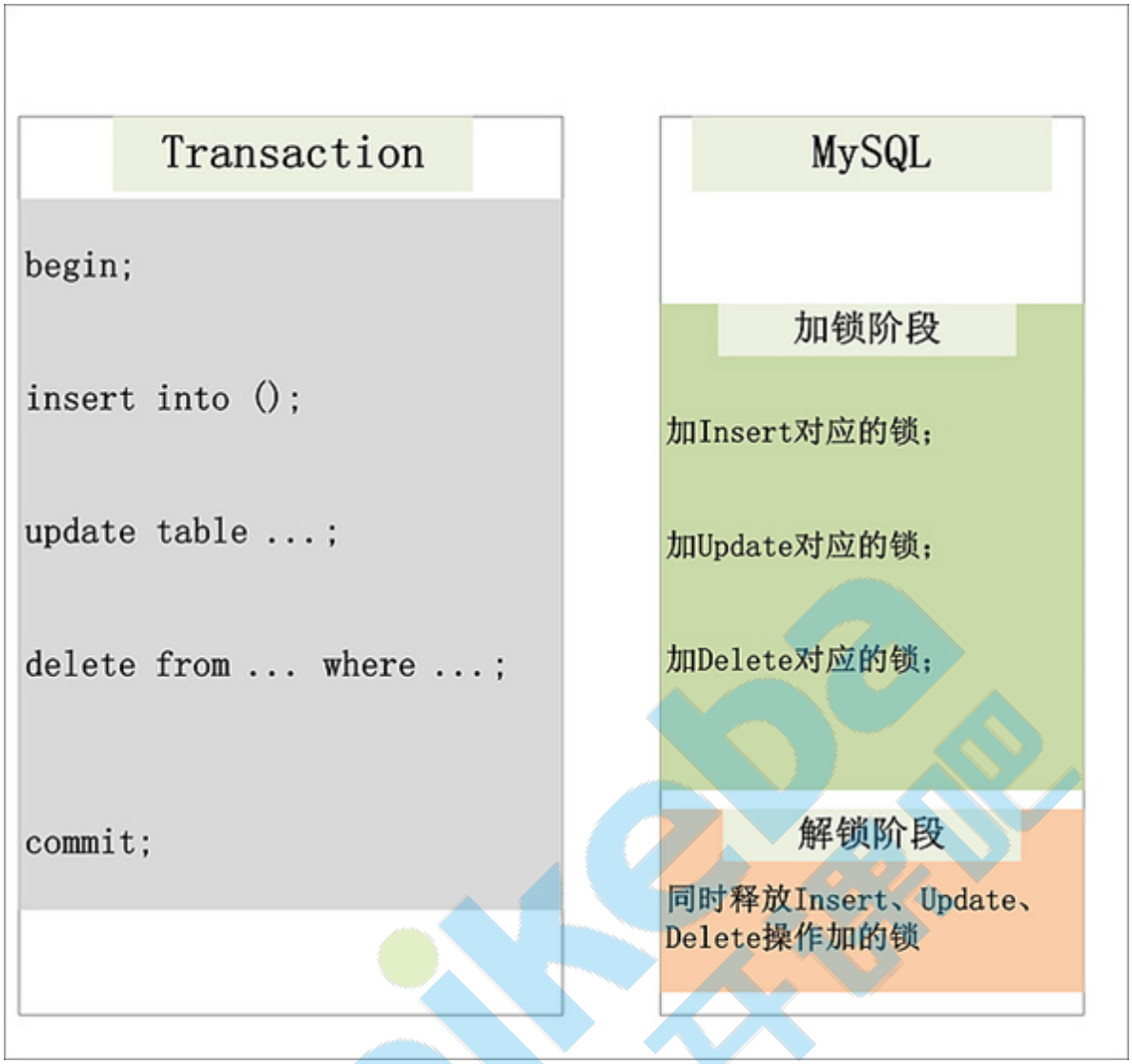
InnoDB也实现了表级锁，也就是意向锁，意向锁是mysql内部使用的，不需要用户干预。

- 意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。
- 意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

  意向锁的主要作用是为了【全表更新数据】时的性能提升。否则在全表更新数据时，需要先检索该表是否某些记录上面有行锁。

	共享锁（S）	**排他锁（X）**	**意向共享锁（IS）**	**意向排他锁（IX）**
共享锁（S）	兼容	冲突	兼容	冲突
排他锁（X）	冲突	冲突	冲突	冲突
意向共享锁（IS）	兼容	冲突	兼容	兼容
意向排他锁（IX）	冲突	冲突	兼容	兼容

两阶段锁（2PL）



锁操作分为两个阶段：加锁阶段与解锁阶段，

加锁阶段与解锁阶段不相交。

加锁阶段：只加锁，不放锁。

解锁阶段：只放锁，不加锁。

行锁演示

InnoDB行锁是通过给索引上的**索引项加锁来实现的**，因此InnoDB这种行锁实现特点意味着：只有通过索引条件检索的数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

where 索引 行锁 否则 表锁

行读锁

session1（Navicat）、session2（mysql）

- Innodb_row_lock_current_waits：当前正在等待锁定的数量；
- Innodb_row_lock_time：从系统启动到现在锁定总时间长度；
- Innodb_row_lock_time_avg：每次等待所花平均时间；
- Innodb_row_lock_time_max：从系统启动到现在等待最常的一次所花的时间；
- Innodb_row_lock_waits：系统启动后到现在总共等待的次数；

```
查看行锁状态  show STATUS like 'innodb_row_lock%';

1、session1: begin;--开启事务未提交
          select * from  mylock  where ID=1 lock in share mode; --手动加id=1的行读锁,使用索引
2、session2: update mylock set name='y' where id=2; -- 未锁定该行可以修改
3、session2: update mylock set name='y' where id=1; -- 锁定该行修改阻塞
    ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
 -- 锁定超时
4、session1: commit; --提交事务 或者 rollback 释放读锁
5、session2: update mylock set name='y' where id=1; --修改成功
            Query OK, 1 row affected (0.00 sec)
            Rows matched: 1  Changed: 1  Warnings: 0

注：使用索引加行锁 ，未锁定的行可以访问
```

行读锁升级为表锁

session1（Navicat）、session2（mysql）

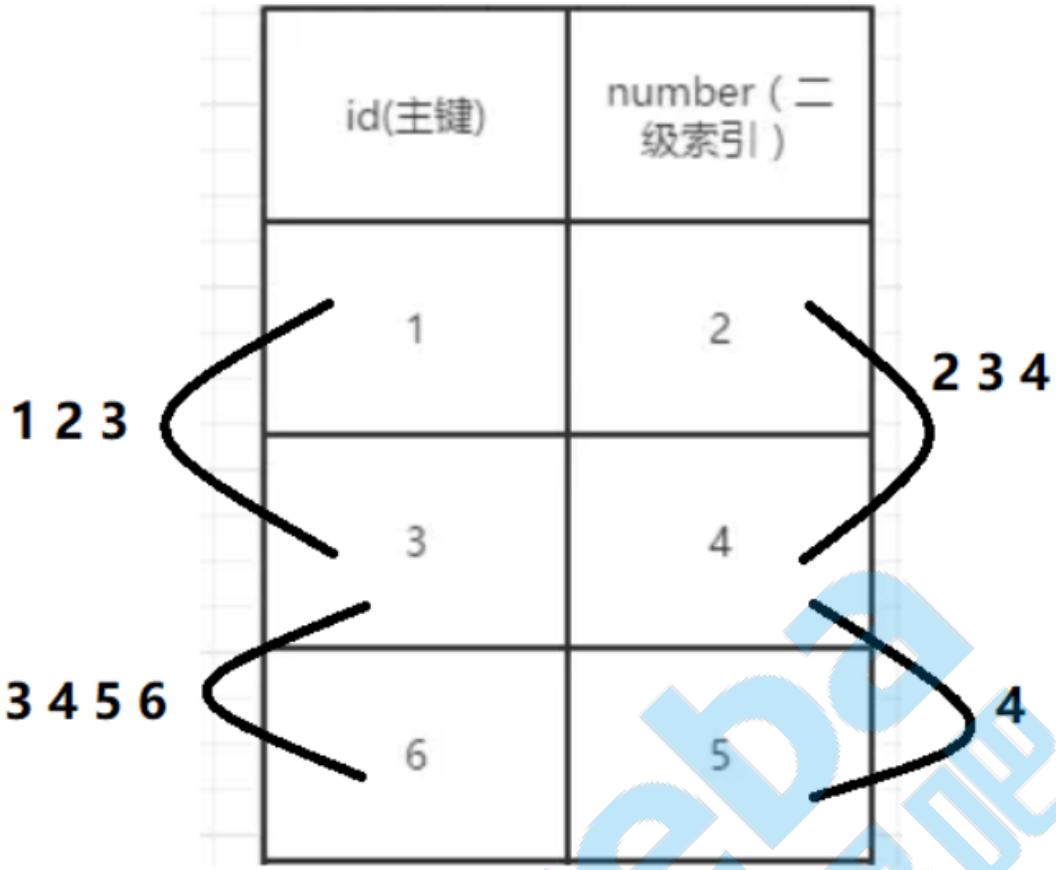
```
1、session1: begin;--开启事务未提交
             --手动加name='c'的行读锁,未使用索引
             select * from  mylock  where name='c' lock in share mode;
2、session2: update mylock set name='y' where id=2; -- 修改阻塞 未用索引行锁升级为表锁
3、session1: commit; --提交事务 或者 rollback 释放读锁
4、session2: update mylock set name='y' where id=2; --修改成功
            Query OK, 1 row affected (0.00 sec)
            Rows matched: 1  Changed: 1  Warnings: 0

注：未使用索引行锁升级为表锁
```

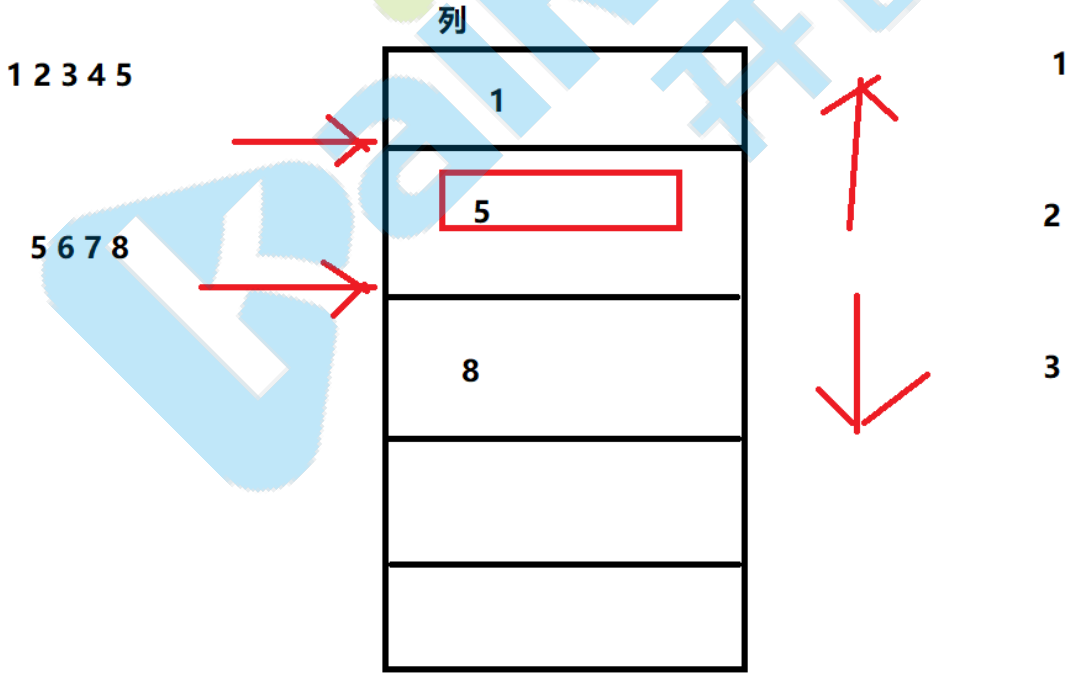
行写锁

session1（Navicat）、session2（mysql）

```
1、session1: begin;--开启事务未提交
             --手动加id=1的行写锁,
             select * from  mylock  where id=1 for update;

2、session2: select * from  mylock  where id=2 ; -- 可以访问
3、session2: select * from  mylock  where id=1 ; -- 可以读 不加锁
    4、session2: select * from  mylock  where id=1 lock in share mode ; -- 加读锁被阻塞
5、session1: commit; -- 提交事务 或者 rollback 释放写锁
5、session2: 执行成功

主键索引产生记录锁
```

间隙锁


id(主键)
number（二级索引）
1
2
3
4
6
5
1 2 3
2 3 4
3 4 5 6
4


列
1 2 3 4 5
5 6 7 8
1
5
8
1
2
3

id(主键)	number（二级索引）
1	2
3	4
6	5
8	5
10	5
13	11

间隙锁防止两种情况

1、防止插入间隙内的数据

2、防止已有数据更新为间隙内的数据

间隙的范围

update news set number=3 where number=4;

number : 2 3 4

id:1 2 3 4 5

间隙情况：

id、number均在间隙内

id、number均在间隙外

id在间隙内、number在间隙外

id在间隙外，number在间隙内

id、number为边缘数据

```
案例演示：
mysql> create table news (id int, number int,primary key (id));
mysql> insert into news values(1,2);
......
--加非唯一索引
mysql> alter table news add index idx_num(number);
```

非唯一索引等值

```
-- 非唯一索引的等值
session 1:
start  transaction ;
update news set number=3 where number=4;
session 2:
start  transaction ;
insert into news value(2,3);#（均在间隙内，阻塞）
insert into news value(7,8);#（均在间隙外，成功）
insert into news value(2,8);#（id在间隙内，number在间隙外，成功）
insert into news value(4,8);#（id在间隙内，number在间隙外，成功）
insert into news value(7,3);#（id在间隙外，number在间隙内，阻塞）
insert into news value(7,2);# (id在间隙外，number为上边缘数据，阻塞)
insert into news value(2,2);#（id在间隙内，number为上边缘数据，阻塞）
insert into news value(7,5);#（id在间隙外，number为下边缘数据，成功）
insert into news value(4,5);#（id在间隙内，number为下边缘数据，阻塞）
```

结论：只要number（where后面的）在间隙里（2 3 4），不包含最后一个数（5）则不管id是多少都会阻塞。

主键索引范围

```
--主键索引范围
session 1:
start  transaction ;
update news set number=3 where id>1 and id <6;
session 2:
start  transaction ;
insert into news value(2,3);#（均在间隙内，阻塞）
insert into news value(7,8);#（均在间隙外，成功）
insert into news value(2,8);#（id在间隙内，number在间隙外，阻塞）
insert into news value(4,8);#（id在间隙内，number在间隙外，阻塞）
insert into news value(7,3);#（id在间隙外，number在间隙内，成功）
--id无边缘数据，因为主键不能重复
```

结论：只要id（在where后面的）在间隙里(2 4 5)，则不管number是多少都会阻塞。

非唯一索引无穷大

session1（Navicat）、session2（mysql）

```
--无穷大
session 1:
start  transaction ;
```

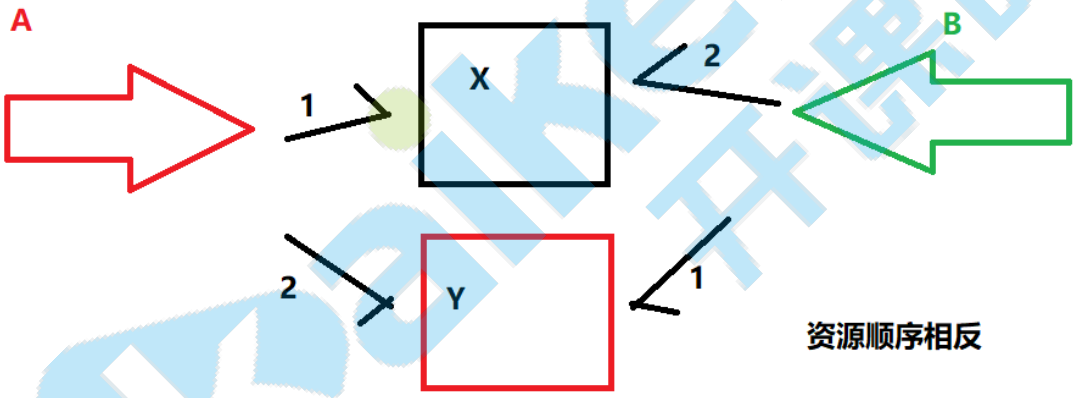
```
update news set number=3 where number=13 ;

session 2:
start  transaction ;
insert into news value(11,5);#(执行成功)
insert into news value(12,11);#(执行成功)
insert into news value(14,11);#(阻塞)
insert into news value(15,12);#(阻塞)
```

检索条件number=13,向左取得最靠近的值11作为左区间，向右由于没有记录因此取得无穷大作为右区间，因此，session 1的间隙锁的范围（11，无穷大）

结论：id和number同时满足
注：非主键索引产生间隙锁，主键范围产生间隙锁

死锁

两个 session 互相等等待对方的资源释放之后，才能释放自己的资源,造成了死锁



session1 （Navicat） 、session2 （mysql）

```
1、session1: begin;--开启事务未提交
              --手动加行写锁 id=1 ， 使用索引
             update mylock set name='m' where id=1;
2、session2: begin;--开启事务未提交
             --手动加行写锁 id=2 ， 使用索引
             update mylock set name='m' where id=2;

3、session1: update mylock set name='nn' where id=2; -- 加写锁被阻塞
4、session2: update mylock set name='nn' where id=1; -- 加写锁会死锁，不允许操作
     ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting
transaction
```

四、事务

事务介绍

在MySQL中的事务是由**存储引擎**实现的，而且支持事务的存储引擎不多，我们主要讲解**InnoDB**存储引擎中的事务。

事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。

事务用来管理`DDL、DML、DCL`操作，比如 insert,update,delete 语句，默认是自动提交的。

事务四大特性(ACID)

- Atomicity（原子性）：构成事务的的所有操作必须是一个逻辑单元，要么全部执行，要么全部不执行。
- Consistency（一致性）：数据库在事务执行前后状态都必须是稳定的或者是一致的。
- **Isolation（隔离性）：事务之间不会相互影响。**

  由锁机制和MVCC机制来实现的

  MVCC(多版本并发控制)：优化读写性能（读不加锁、读写不冲突）
- Durability（持久性）：事务执行成功后必须全部写入磁盘。

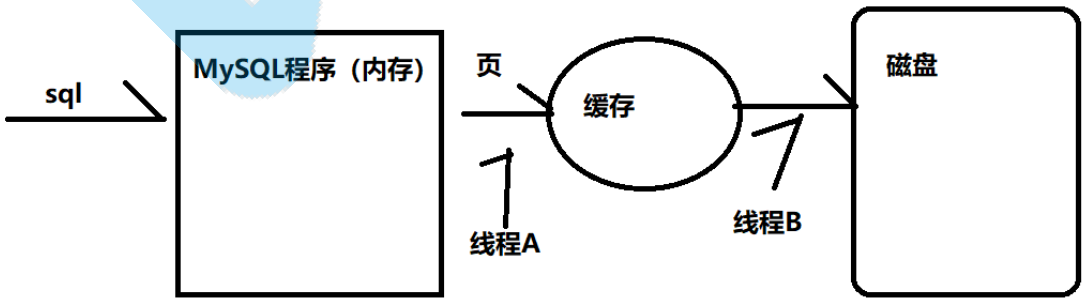
事务开启

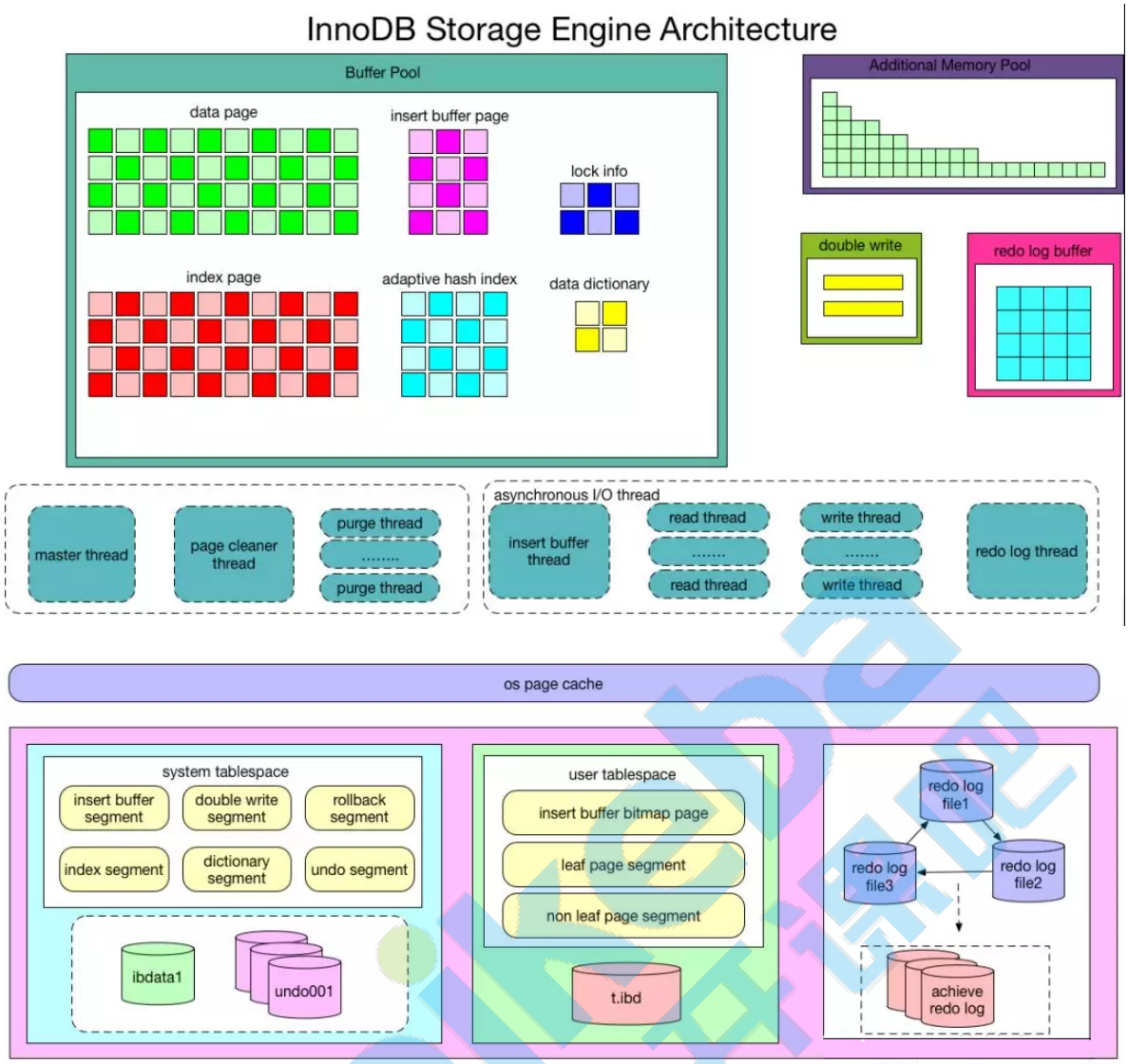
BEGIN`或`START TRANSACTION`；显式地开启一个事务；

`COMMIT`也可以使用`COMMIT WORK`，不过二者是等价的。COMMIT会提交事务，并使已对数据库进行的所有修改称为永久性的；

ROLLBACK`有可以使用`ROLLBACK WORK`，不过二者是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；

InnoDB架构图





上图详细显示了InnoDB存储引擎的体系架构，从图中可见，InnoDB存储引擎由**内存池，后台线程和磁盘文件**三大部分组成。接下来我们就来简单了解一下内存相关的概念和原理。

InnoDB内存结构

Buffer Pool缓冲池

处理数据

数据页和索引页

Page是Innodb存储的最基本结构，也是Innodb磁盘管理的最小单位

做增删改时 缓存里的数据页和磁盘里的数据页不一致，该数据页为脏页

插入缓冲(Insert Buffer)

复杂： 主键排序 索引 树状 插入算法 。。。

自适应哈希索引(Adaptive Hash Index)

hash结构 k-v

InnoDB会根据访问的频率和模式，为热点页建立哈希索引，来提高查询效率

锁信息(lock info)

行锁、 表锁 。。。

数据字典信息(Data Dictionary)

元数据信息 包括表结构、数据库名或表名、字段的数据类型、视图、索引、表字段信息、存储过程、触发器等内容

Redo log Buffer重做日志缓冲

重做日志： Redo Log 如果要存储数据则先存储数据的日志， 一旦内存崩了 则可以从日志找

重做日志保证了数据的可靠性，InnoDB采用了**Write Ahead Log（预写日志）**策略，即当事务提交时，先写重做日志，然后再择时将脏页写入磁盘。如果发生宕机导致数据丢失，就通过重做日志进行数据恢复。

insert into xxxx

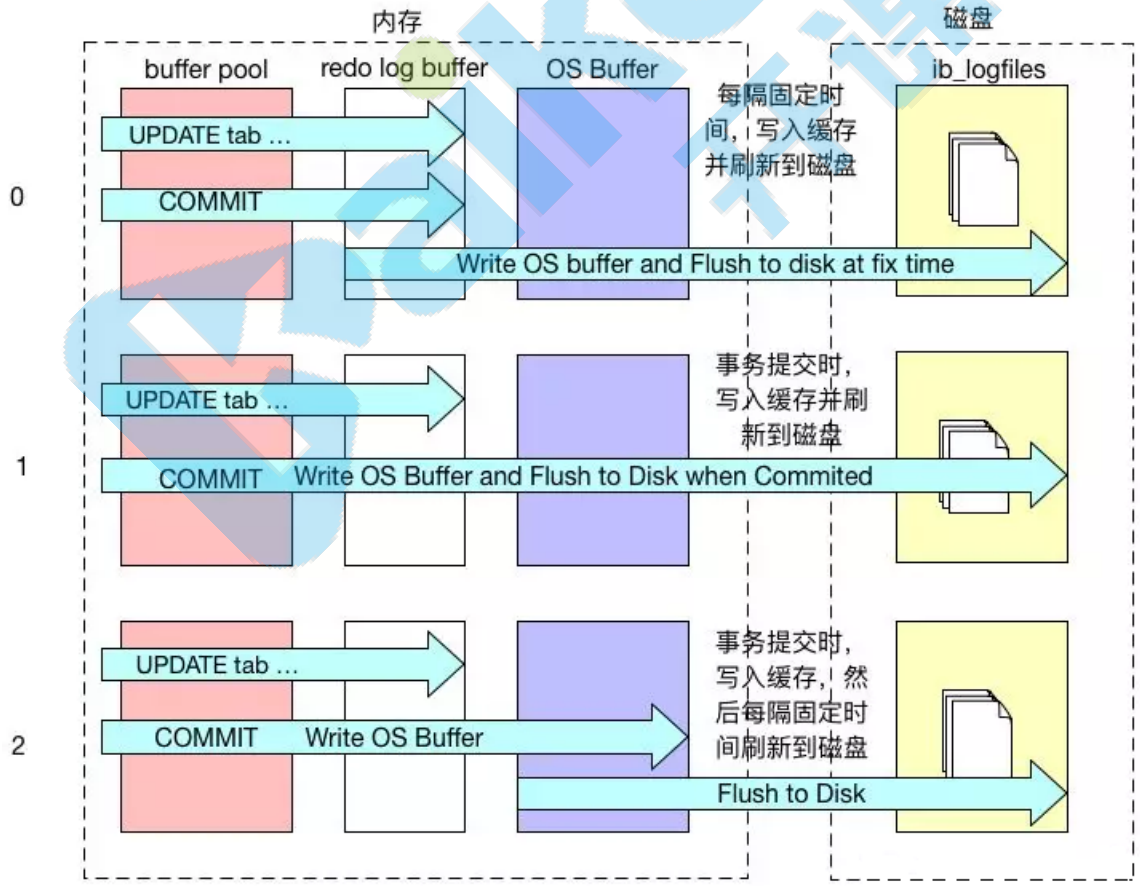
commit; Redo Log File 写成功 则Commit；

Redo Log： ib_logfile0 ib_logfile1 默认为8MB。 可通过配置参数 `innodb_log_buffer_size` 控制

Force Log at Commit机制实现事务的持久性，即当事务提交时，必须先将该事务的所有日志写入到重做日志文件进行持久化，然后事务的提交操作完成才算完成。为了确保每次日志都写入到重做日志文件，在每次将重做日志缓冲写入重做日志后，必须调用一次**fsync**操作（操作系统），将缓冲文件从文件系统缓存中真正写入磁盘。

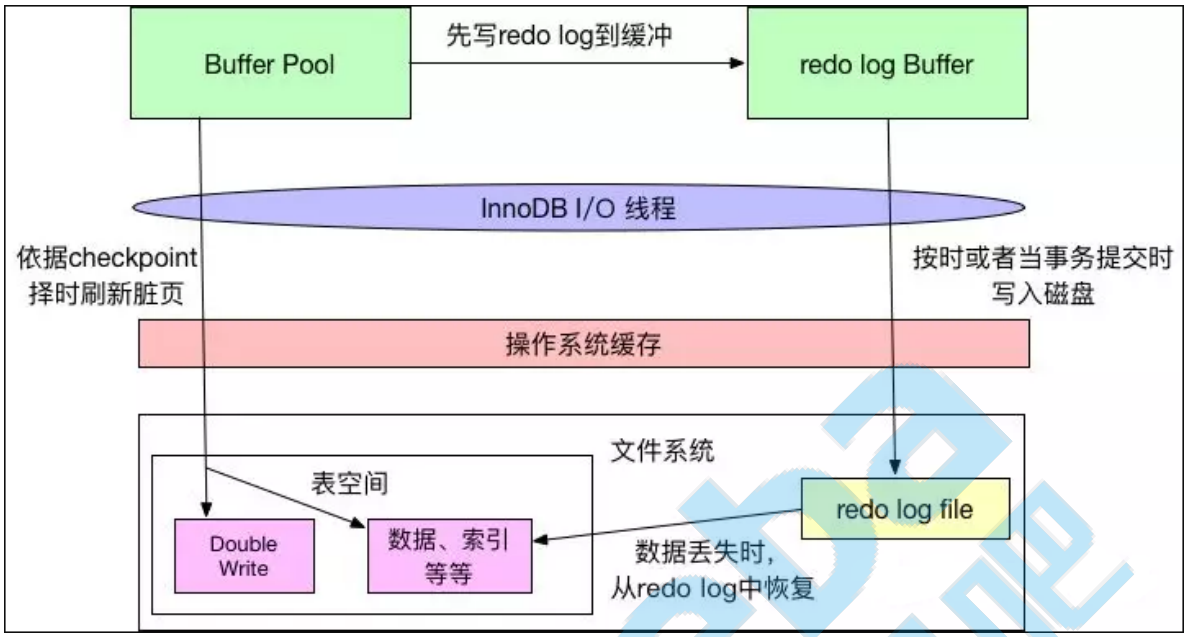
innodb_flush_log_at_trx_commit

重做日志的落盘机制



该参数默认值为1

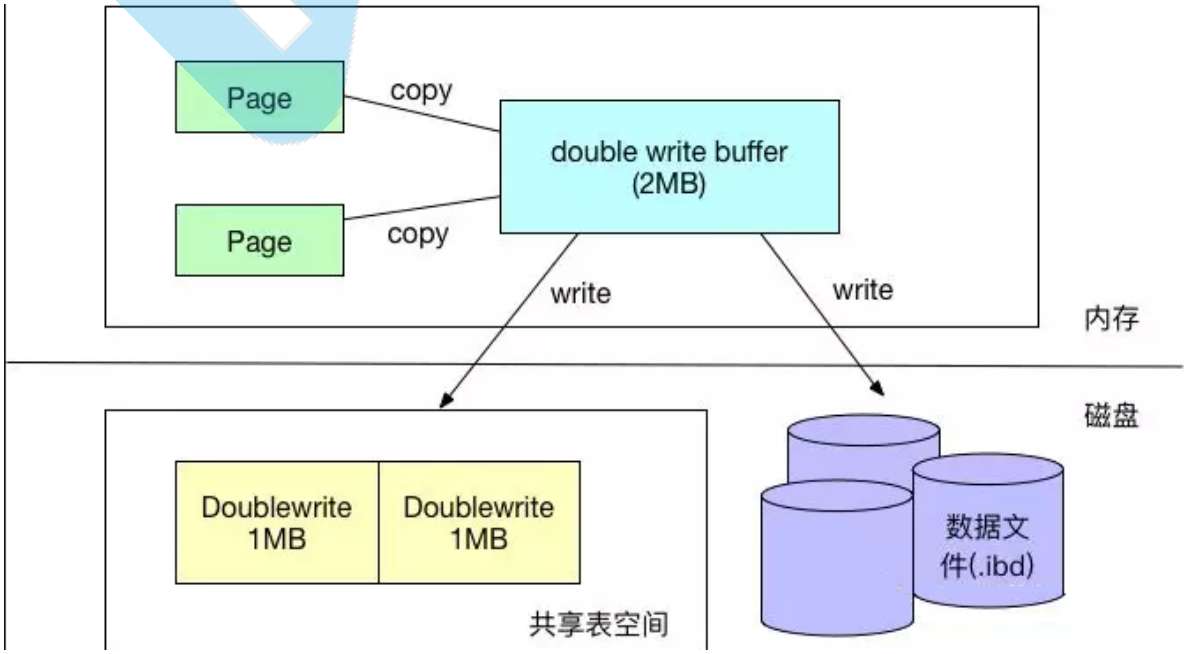
可以通过`innodb_flush_log_at_trx_commit`来控制重做日志刷新到磁盘的策略。该参数默认值为1，表示事务提交必须进行一次fsync操作，还可以设置为0和2。0表示事务提交时不进行写入重做日志操作，该操作只在主线程中完成，2表示提交时写入重做日志，但是只写入文件系统缓存，不进行fsync操作。由此可见，设置为0时，性能最高，但是丧失了事务的一致性。



Double Write双写

Double Write带给InnoDB存储引擎的是数据页的可靠性

如上图所示，**Double Write由两部分组成，一部分是内存中的double write buffer，大小为2MB，另一部分是物理磁盘上共享表空间连续的128个页，大小也为2MB。**在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是通过memcpy函数将脏页先复制到内存中的double write buffer区域，之后通过double write buffer再分两次，每次1MB顺序地写入共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘，避免操作系统缓冲写带来的问题。在完成double write页的写入后，再讲double wirite buffer中的页写入各个表空间文件中。如果操作系统在将页写入磁盘的过程中发生了崩溃，在恢复过程中，InnoDB存储引擎可以从共享表空间中的double write中找到该页的一个副本，将其复制到表空间文件中，再应用重做日志。



CheckPoint： 检查点

检查点，表示脏页写入到磁盘的时机，所以检查点也就意味着脏数据的写入。

1、checkpoint的目的

1、缩短数据库的恢复时间

2、buffer pool空间不够用时，将脏页刷新到磁盘

3、redolog不可用时，刷新脏页

2、检查点分类

1、sharp checkpoint：

完全检查点 数据库正常关闭时，会触发把所有的脏页都写入到磁盘上

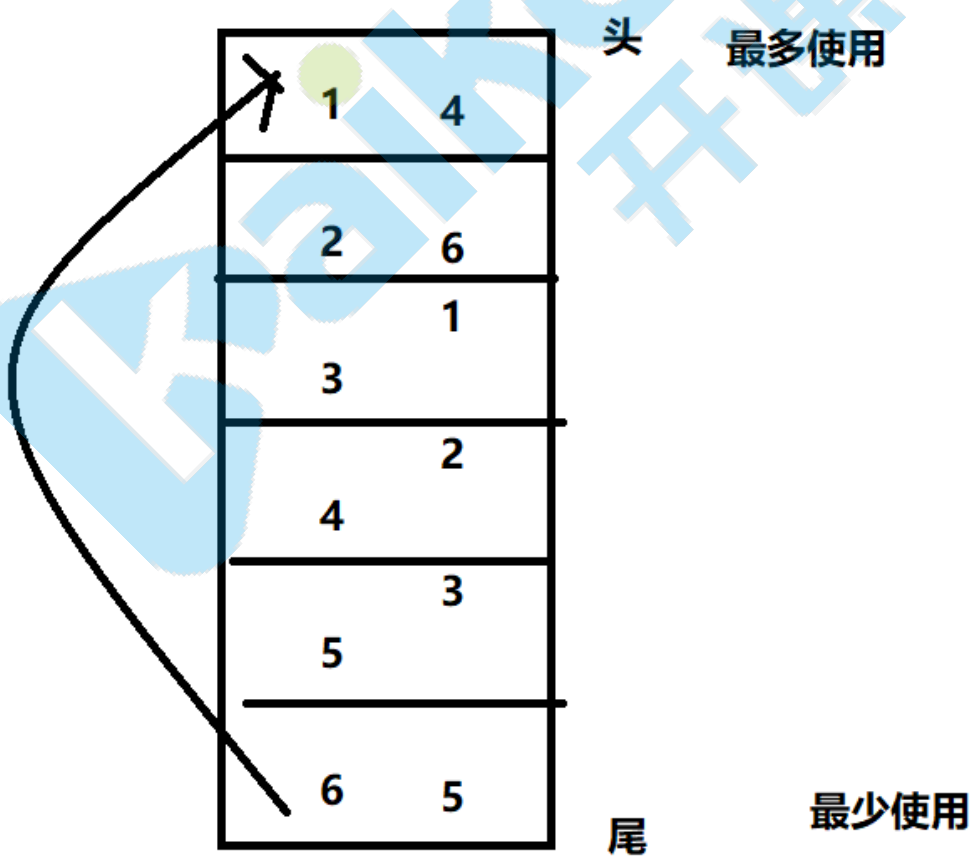
2、fuzzy checkpoint：

正常使用时 模糊检查点，部分页写入磁盘。

master thread checkpoint、flush_lru_list checkpoint、async/sync flush checkpoint、dirty page too much checkpoint。

master thread checkpoint： 以每秒或每十秒的速度从缓冲池的脏页列表中刷新一定比例的页回磁盘，这个过程是异步的，

flush_lru_list checkpoint： 读取lru (Least Recently Used) list，找到脏页，写入磁盘。 最近最少使用

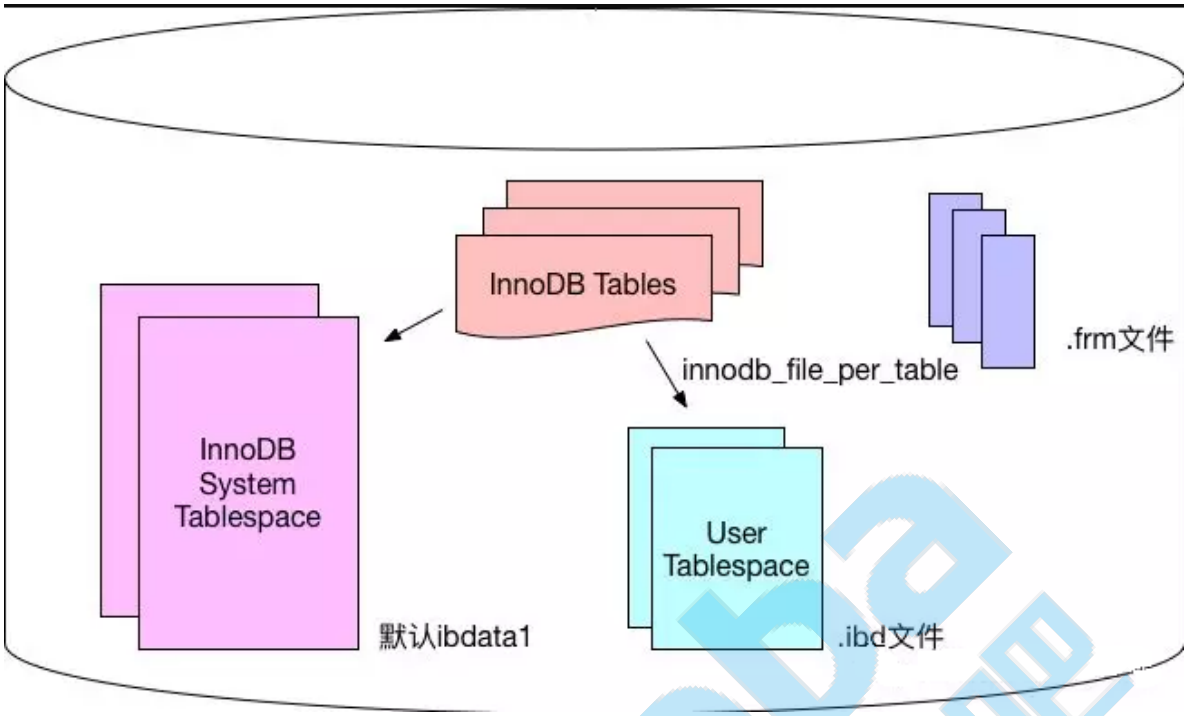


async/sync flush checkpoint： redo log file快满了，会批量的触发数据页回写，这个事件触发的时候又分为异步和同步，不可被覆盖的redolog占log file的比值：75%--->异步、90%--->同步。

dirty page too much checkpoint： 默认是脏页占比75%的时候，就会触发刷盘，将脏页写入磁盘

InnoDB磁盘文件

系统表空间和用户表空间



`innodb_data_file_path`的格式如下：

```
innodb_data_file_path=datafile1[,datafile2]...
```

用户可以通过多个文件组成一个表空间，同时制定文件的属性：

```
innodb_data_file_path = /db/ibdata1:1000M;/dr2/db/ibdata2:1000M:autoextend
```

这里将/db/ibdata1和/dr2/db/ibdata2两个文件组成系统表空间。如果这两个文件位于不同的磁盘上

系统表空间（共享表空间）

1、数据字典(data dictionary)：记录数据库相关信息

2、doublewrite write buffer：解决部分写失败（页断裂）

3、insert buffer：内存insert buffer数据，周期写入共享表空间，防止意外宕机

4、回滚段(rollback segments)

5、undo空间：undo页

用户表空间（独立表空间）

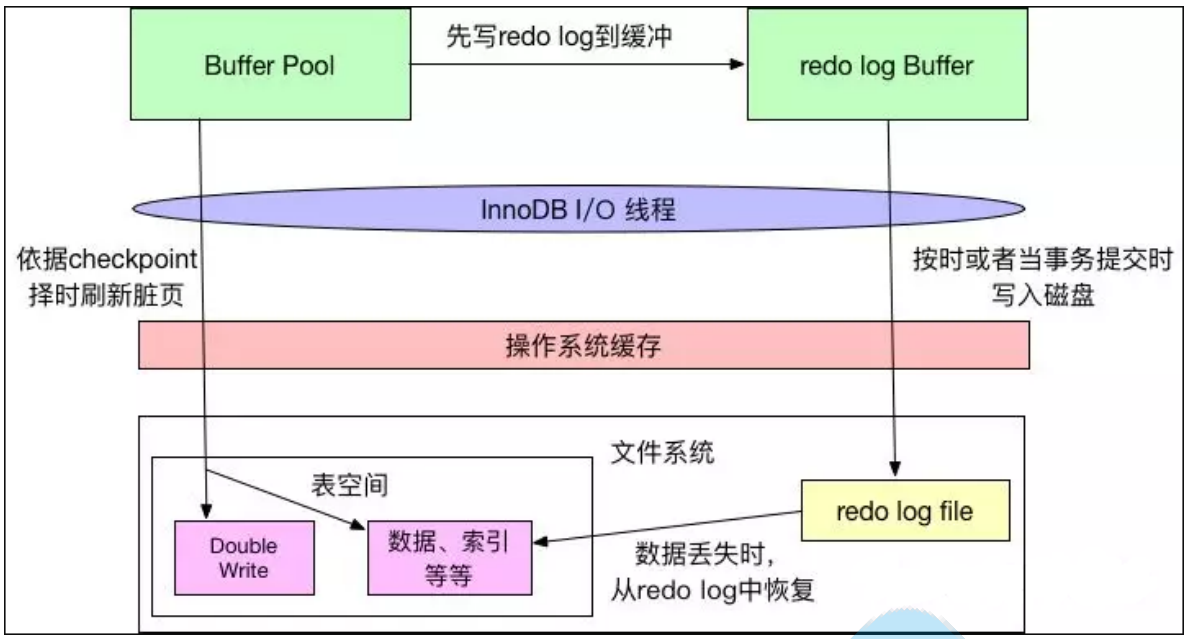
1、每个表的数据和索引都会存在自己的表空间中

2、每个表的结构

3、undo空间：undo页 （需要设置）

4、doublewrite write buffer

重做日志文件和归档文件



ib_logfile0 和 ib_logfile1

在日志组中每个重做日志文件的大小一致，并以【循环写入】的方式运行。

InnoDB存储引擎先写入重做日志文件1，当文件被写满时，会切换到重做日志文件2，再当重做日志文件2也被写满时，再切换到重做日志文件1。