# 背景

MySQL物理备份工具，常用的有两个：MySQL Enterprise Backup（MEB）和XtraBackup（PXB）。

前者常用于MySQL企业版，后者常用于MySQL社区版、Percona Server for MySQL和MariaDB。

所以，如果我们使用的是后三者，在实例较大的情况下，一般都会选择XtraBackup作为备份恢复工具。

## .ibd和.ibdata

两者都是专属于InnoDB存储引擎的数据库文件。

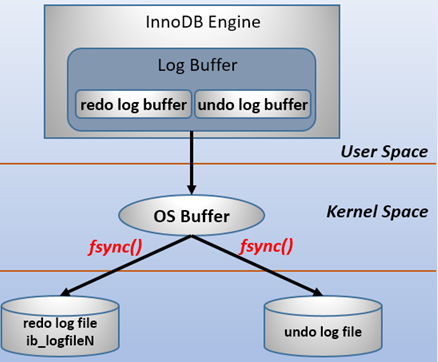
当采用共享表空间时所有InnoDB表的数据均存放在.ibdata中，所以当表越来越多时，这个文件会变得很大；相对应的.ibd就是采用独享表空间时InnoDB表的数据文件。

之所以有两种文件来存放Innodb的数据（包括索引），是因为Innodb的数据存储方式能够通过配置来决定是使用共享表空间存放存储数据，还是独享表空间存放存储数据。独享表空间存储方式使用“.ibd”文件来存放数据，且每个表一个“.ibd”文件，文件存放在和MyISAM数据相同的位置。如果选用共享存储表空间来存放数据，则会使用ibdata文件来存放，所有表共同使用一个（或者多个，可自行配置）ibdata文件。

## redo log

和undo log相反，redo log记录的是**新数据的备份**。**在事务提交前，只要将redo log持久化即可，不需要将数据持久化**。当系统崩溃时，虽然数据没有持久化，但是redo log已经持久化。系统可以根据redo log的内容，将所有数据恢复到最新的状态。

redo log包括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。



# 概述

参考：

<https://zhuanlan.zhihu.com/p/432784679>

<http://mysql.taobao.org/monthly/2016/03/07/>

<https://zhuanlan.zhihu.com/p/419385309>

Percona XtraBackup（简称PXB）是Percona公司开发的一个用于MySQL数据库物理热备的备份工具，具有备份速度快、支持备份数据压缩、自动校验备份数据、支持流式输出、备份过程中几乎不影响业务等特点，是目前各个云厂商普遍使用的MySQL备份工具。支持MySQL（Oracle）、Percona Server和MariaDB，并且全部开源。

当前Xtrabackup存在两个版本：Xtrabackup 2.4.x与8.0.x，分别用于备份MySQL 5.x与MySQL 8.0.x 版本。

## 工具集

软件包安装完后一共有4个可执行文件，如下：

usr

├── bin

│ ├── innobackupex

│ ├── xbcrypt

│ ├── xbstream

│ └── xtrabackup

其中最主要的是innobackupex和xtrabackup，前者是一个perl脚本，后者是C/C++编译的二进制。

xtrabackup是用来备份InnoDB表的，不能备份非InnoDB表，**和mysqld server没有交互**；innobackupex脚本用来备份非InnoDB表，同时会调用xtrabackup命令来备份InnoDB表，**还会和mysqld server发送命令进行交互，如加读锁（FTWRL）、获取位点（SHOW SLAVE STATUS）等**。简单来说，innobackupex在xtrabackup之上做了一层封装。

一般情况下，我们是希望能备份MyISAM表的，虽然我们可能自己不用MyISAM表，但是mysql库下的系统表是MyISAM的，因此备份基本都通过 innobackupex命令进行；另外一个原因是我们可能需要保存位点信息。

另外2个工具相对小众些，xbcrypt是加解密用的；xbstream类似于tar，是Percona自己实现的一种支持并发写的流文件格式。两都在备份和解压时都会用到（如果备份用了加密和并发）。

# 原理

参考：

<https://www.yulan.work/posts/percona-xtrabackup-restore-backup-sc-analysis/>

<https://www.bookstack.cn/read/aliyun-rds-core/fc9c390ec1e4169b.md>

## 通信方式

2个工具之间的交互和协调是通过控制文件的创建和删除来实现的，主要文件有：

xtrabackup\_suspended\_1

xtrabackup\_suspended\_2

xtrabackup\_log\_copied

举个例子，我们来看备份时 xtrabackup\_suspended\_2 是怎么来协调2个工具进程的：

1、innobackupex在启动xtrabackup进程后，会一直等xtrabackup备份完InnoDB文件，方式就是等待xtrabackup\_suspended\_2这个文件被创建出来；xtrabackup在备完InnoDB数据后，就在指定目录下创建出这个文件，然后等这个文件被innobackupex删除；

2、innobackupex检测到文件xtrabackup\_suspended\_2被创建出来后，就继续往下走；

3、innobackupex在备份完非InnoDB表后，删除xtrabackup\_suspended\_2这个文件，这样就通知xtrabackup可以继续了，然后等xtrabackup\_log\_copied被创建；

4、xtrabackup检测到xtrabackup\_suspended\_2文件删除后，就可以继续往下了。

是不是感觉有点不可思议，通过文件是否存在来控制进程，这种方式非常的不靠谱，因为非常容易被外部干扰，比如文件被别人误删掉，或者2个正在跑的备份控制文件误放在同一个目录下，备份会乱掉，但是Percona就是这么干的。

之所以这么搞，估计主要是因为perl和C二进制2个进程，没有既好用又方便的通信方式，搞个协议太麻烦。但是官方也觉得这种方式不靠谱，11年就搞了个blueprint要用C重写innobackupex，终于在2.3 版本实现了，innobackupex功能全部集成到xtrabackup里面，只有一个binary，另外为了使用上的兼容考虑，innobackupex作为xtrabackup的一个软链。对于二次开发来说，2.3摆脱了之前2个进程协作的负担，架构上明显要好于之前版本。考虑到 perl + C这种架构的长期存在，大多数读者朋友也基本用的2.3之前版本，本文的介绍也是基于老的架构（2.2版本），但是原理和2.3是一样的，只是实现上的差别。

## 备份过程

整个备份过程如下：

1、innobackupex在启动后，会先fork一个进程，启动xtrabackup进程，然后就等待xtrabackup备份完ibd数据文件；

2、xtrabackup在备份InnoDB相关数据时，是有2种线程的：

redo拷贝线程，负责拷贝redo文件；

ibd拷贝线程，负责拷贝ibd文件。

redo拷贝线程只有一个，在ibd拷贝线程之前启动，在ibd线程结束后结束。

xtrabackup进程开始执行后，先启动redo拷贝线程，从最新的checkpoint点开始顺序拷贝redo日志；然后再启动ibd数据拷贝线程，在xtrabackup拷贝ibd过程中，innobackupex进程一直处于等待状态（等待文件被创建）。

3、xtrabackup拷贝完成idb后，通知innobackupex（通过创建文件），同时自己进入等待（redo线程仍然继续拷贝）;

4、innobackupex收到xtrabackup通知后，执行FLUSH TABLES WITH READ LOCK (FTWRL)，取得一致性位点，然后开始备份非InnoDB文件（包括 frm、MYD、MYI、CSV、opt、par等）。拷贝非InnoDB文件过程中，因为数据库处于全局只读状态，如果在业务的主库备份的话，要特别小心，非InnoDB表（主要是MyISAM）比较多的话整库只读时间就会比较长，这个影响一定要评估到。

5、当innobackupex拷贝完所有非 InnoDB 表文件后，通知xtrabackup（通过删文件），同时自己进入等待（等待另一个文件被创建）；

6、xtrabackup收到innobackupex备份完非InnoDB通知后，就停止redo拷贝线程，然后通知innobackupex redo log拷贝完成（通过创建文件）；

7、innobackupex收到redo备份完成通知后，就开始解锁，执行UNLOCK TABLES；

8、最后innobackupex和xtrabackup进程各自完成收尾工作，如资源的释放、写备份元数据信息等，innobackupex等待 xtrabackup子进程结束后退出。

在上面描述的文件拷贝，都是备份进程直接通过操作系统读取数据文件的，只在执行SQL命令时和数据库有交互，基本不影响数据库的运行，在备份非InnoDB时会有一段时间只读（如果没有MyISAM表的话，只读时间在几秒左右），在备份 InnoDB 数据文件时，对数据库完全没有影响，是真正的热备。

InnoDB和非InnoDB文件的备份都是通过拷贝文件来做的，但是实现的方式不同，前者是以page为粒度做的(xtrabackup)，后者是cp或者tar命令(innobackupex)，xtrabackup 在读取每个page时会校验 checksum 值，保证数据块是一致的，而 innobackupex在cp MyISAM文件时已经做了flush（FTWRL），磁盘上的文件也是完整的，所以最终备份集里的数据文件都是写入完整的。

## 增量备份

PXB是支持增量备份的，但是只能对InnoDB做增量，InnoDB每个page有个LSN号，LSN是全局递增的，page被更改时会记录当前的LSN号，page中的LSN越大，说明当前page越新（最近被更新）。每次备份会记录当前备份到的LSN（xtrabackup\_checkpoints文件中），增量备份就是只拷贝LSN大于上次备份的page，比上次备份小的跳过，每个ibd文件最终备份出来的是增量delta 文件。

MyISAM是没有增量的机制的，每次增量备份都是全部拷贝的。

增量备份过程和全量备份一样，只是在ibd文件拷贝上有不同。

### XtraBackup

XtraBackup的main函数定义在storage/innobase/xtrabackup/src/xtrabackup.cc文件中。

可以看到，对于--backup选项，会调用xtrabackup\_backup\_func函数。

int main(int argc, char \*\*argv)

{

...

/\* --backup \*/

if (xtrabackup\_backup) {

xtrabackup\_backup\_func();

}

/\* --stats \*/

if (xtrabackup\_stats) {

xtrabackup\_stats\_func(server\_argc, server\_defaults);

}

/\* --prepare \*/

if (xtrabackup\_prepare) {

xtrabackup\_prepare\_func(server\_argc, server\_defaults);

}

if (xtrabackup\_copy\_back || xtrabackup\_move\_back) {

if (!check\_if\_param\_set("datadir")) {

msg("Error: datadir must be specified.\n");

exit(EXIT\_FAILURE);

}

mysql\_mutex\_init(key\_LOCK\_keyring\_operations,

&LOCK\_keyring\_operations, MY\_MUTEX\_INIT\_FAST);

if (!copy\_back(server\_argc, server\_defaults)) {

exit(EXIT\_FAILURE);

}

mysql\_mutex\_destroy(&LOCK\_keyring\_operations);

}

...

msg\_ts("completed OK!\n");

exit(EXIT\_SUCCESS);

}

#### xtrabackup\_backup\_func

下面重点看看xtrabackup\_backup\_func函数的处理逻辑。

xtrabackup\_backup\_func

该函数同样位于xtrabackup.cc文件中。

void

xtrabackup\_backup\_func(void)

{

...

/\* start back ground thread to copy newer log \*/

/\* 创建redo log拷贝线程 \*/

os\_thread\_id\_t log\_copying\_thread\_id;

datafiles\_iter\_t \*it;

...

/\* get current checkpoint\_lsn \*/

/\* Look for the latest checkpoint from any of the log groups \*/

/\* 获取最新的checkpoint lsn \*/

mutex\_enter(&log\_sys->mutex);

err = recv\_find\_max\_checkpoint(&max\_cp\_group, &max\_cp\_field);

if (err != DB\_SUCCESS) {

ut\_free(log\_hdr\_buf\_);

exit(EXIT\_FAILURE);

}

log\_group\_header\_read(max\_cp\_group, max\_cp\_field);

buf = log\_sys->checkpoint\_buf;

checkpoint\_lsn\_start = mach\_read\_from\_8(buf + LOG\_CHECKPOINT\_LSN);

checkpoint\_no\_start = mach\_read\_from\_8(buf + LOG\_CHECKPOINT\_NO);

...

/\* copy log file by current position \*/

/\* 从最新的checkpoint lsn开始拷贝redo log \*/

if(xtrabackup\_copy\_logfile(checkpoint\_lsn\_start, FALSE))

exit(EXIT\_FAILURE);

mdl\_taken = true;

log\_copying\_stop = os\_event\_create("log\_copying\_stop");

debug\_sync\_point("xtrabackup\_pause\_after\_redo\_catchup");

os\_thread\_create(log\_copying\_thread, NULL, &log\_copying\_thread\_id);

/\* Populate fil\_system with tablespaces to copy \*/

/\* 获取ibdata1，undo tablespaces及所有的ibd文件 \*/

err = xb\_load\_tablespaces();

if (err != DB\_SUCCESS) {

msg("xtrabackup: error: xb\_load\_tablespaces() failed with"

"error code %lu\n", err);

exit(EXIT\_FAILURE);

}

...

/\* Create data copying threads \*/

/\* 创建数据拷贝线程 \*/

data\_threads = (data\_thread\_ctxt\_t \*)

ut\_malloc\_nokey(sizeof(data\_thread\_ctxt\_t) \*

xtrabackup\_parallel);

count = xtrabackup\_parallel;

mutex\_create(LATCH\_ID\_XTRA\_COUNT\_MUTEX, &count\_mutex);

/\* 拷贝物理文件，其中，xtrabackup\_parallel是拷贝并发线程数，由--parallel参数指定 \*/

for (i = 0; i < (uint) xtrabackup\_parallel; i++) {

data\_threads[i].it = it;

data\_threads[i].num = i+1;

data\_threads[i].count = &count;

data\_threads[i].count\_mutex = &count\_mutex;

data\_threads[i].error = &data\_copying\_error;

os\_thread\_create(data\_copy\_thread\_func, data\_threads + i,

&data\_threads[i].id);

}

/\* 循环等待，直到拷贝结束 \*/

/\* Wait for threads to exit \*/

while (1) {

os\_thread\_sleep(1000000);

mutex\_enter(&count\_mutex);

if (count == 0) {

mutex\_exit(&count\_mutex);

break;

}

mutex\_exit(&count\_mutex);

}

mutex\_free(&count\_mutex);

ut\_free(data\_threads);

datafiles\_iter\_free(it);

if (data\_copying\_error) {

exit(EXIT\_FAILURE);

}

if (changed\_page\_bitmap) {

xb\_page\_bitmap\_deinit(changed\_page\_bitmap);

}

}

/\* 调用backup\_start函数，这个函数会加全局读锁，拷贝非ibd文件 \*/

if (!backup\_start()) {

exit(EXIT\_FAILURE);

}

if(opt\_lock\_ddl\_per\_table && opt\_debug\_sleep\_before\_unlock){

msg\_ts("Debug sleep for %u seconds\n",

opt\_debug\_sleep\_before\_unlock);

os\_thread\_sleep(opt\_debug\_sleep\_before\_unlock \* 1000000);

}

/\* 读取最新的checkpoint lsn，用于后续的增量备份 \*/

/\* read the latest checkpoint lsn \*/

latest\_cp = 0;

{

log\_group\_t\* max\_cp\_group;

ulint max\_cp\_field;

ulint err;

mutex\_enter(&log\_sys->mutex);

err = recv\_find\_max\_checkpoint(&max\_cp\_group, &max\_cp\_field);

if (err != DB\_SUCCESS) {

msg("xtrabackup: Error: recv\_find\_max\_checkpoint() failed.\n");

mutex\_exit(&log\_sys->mutex);

goto skip\_last\_cp;

}

log\_group\_header\_read(max\_cp\_group, max\_cp\_field);

xtrabackup\_choose\_lsn\_offset(checkpoint\_lsn\_start);

latest\_cp = mach\_read\_from\_8(log\_sys->checkpoint\_buf +

LOG\_CHECKPOINT\_LSN);

mutex\_exit(&log\_sys->mutex);

msg("xtrabackup: The latest check point (for incremental): "

"'" LSN\_PF "'\n", latest\_cp);

}

skip\_last\_cp:

/\* 停止redo log拷贝线程. 将备份的元数据信息记录在XTRABACKUP\_METADATA\_FILENAME中，即xtrabackup\_checkpoints \*/

/\* stop log\_copying\_thread \*/

log\_copying = FALSE;

os\_event\_set(log\_copying\_stop);

msg("xtrabackup: Stopping log copying thread.\n");

while (log\_copying\_running) {

msg(".");

os\_thread\_sleep(200000); /\*0.2 sec\*/

}

msg("\n");

os\_event\_destroy(log\_copying\_stop);

if (ds\_close(dst\_log\_file)) {

exit(EXIT\_FAILURE);

}

if (!validate\_missing\_encryption\_tablespaces()) {

exit(EXIT\_FAILURE);

}

if(!xtrabackup\_incremental) {

strcpy(metadata\_type, "full-backuped");

metadata\_from\_lsn = 0;

} else {

strcpy(metadata\_type, "incremental");

metadata\_from\_lsn = incremental\_lsn;

}

metadata\_to\_lsn = latest\_cp;

metadata\_last\_lsn = log\_copy\_scanned\_lsn;

if (!xtrabackup\_stream\_metadata(ds\_meta)) {

msg("xtrabackup: Error: failed to stream metadata.\n");

exit(EXIT\_FAILURE);

}

/\* 调用backup\_finish函数，这个函数会释放全局读锁 \*/

if (!backup\_finish()) {

exit(EXIT\_FAILURE);

}

...

}

该函数的处理流程如下：

1、创建redo log拷贝线程，从最近的checkpoint lsn开始拷贝redo log。

2、创建数据文件拷贝线程，拷贝ibdata1，undo tablespaces及所有的ibd文件。

这里可通过设置--parallel进行多线程备份，提高物理文件的拷贝效率。不设置则默认为1。

3、ibd文件拷贝完成后，调用backup\_start函数。

4、停止redo log拷贝线程。

5、调用backup\_finish函数。

接下来重点看看backup\_start和backup\_finish这两个函数的实现逻辑。

##### backup\_start

backup\_start该函数位于backup\_copy.cc文件中。

bool

backup\_start()

{

/\* opt\_no\_lock指的是--no-lock参数 \*/

if (!opt\_no\_lock) {

/\* 如果指定了--safe-slave-backup，会关闭SQL线程，等待Slave\_open\_temp\_tables变量为0。

如果使用的是statement格式，且使用了临时表，建议设置--safe-slave-backup。对于row格式，无需指定该选项 \*/

if (opt\_safe\_slave\_backup) {

if (!wait\_for\_safe\_slave(mysql\_connection)) {

return(false);

}

}

/\* 调用backup\_files函数备份非ibd文件，加了全局读锁还会调用一次。这一次，实际上针对的是--rsync方式 \*/

if (!backup\_files(fil\_path\_to\_mysql\_datadir, true)) {

return(false);

}

history\_lock\_time = time(NULL);

/\* 加全局读锁，如果支持备份锁，且没有设置--no-backup-locks，会优先使用备份锁 \*/

if (!lock\_tables\_maybe(mysql\_connection,

opt\_backup\_lock\_timeout,

opt\_backup\_lock\_retry\_count)) {

return(false);

}

}

/\* 备份非ibd文件 \*/

if (!backup\_files(fil\_path\_to\_mysql\_datadir, false)) {

return(false);

}

// There is no need to stop slave thread before coping non-Innodb data when

// --no-lock option is used because --no-lock option requires that no DDL or

// DML to non-transaction tables can occur.

if (opt\_no\_lock) {

if (opt\_safe\_slave\_backup) {

if (!wait\_for\_safe\_slave(mysql\_connection)) {

return(false);

}

}

}

/\* 如果设置了--slave-info，会将SHOW SLAVE STATUS的相关信息，记录在xtrabackup\_slave\_info中 \*/

if (opt\_slave\_info) {

/\* 如果之前使用了备份锁，这里会先锁定Binlog（LOCK BINLOG FOR BACKUP）\*/

lock\_binlog\_maybe(mysql\_connection, opt\_backup\_lock\_timeout,

opt\_backup\_lock\_retry\_count);

if (!write\_slave\_info(mysql\_connection)) {

return(false);

}

}

/\* The only reason why Galera/binlog info is written before

wait\_for\_ibbackup\_log\_copy\_finish() is that after that call the xtrabackup

binary will start streamig a temporary copy of REDO log to stdout and

thus, any streaming from innobackupex would interfere. The only way to

avoid that is to have a single process, i.e. merge innobackupex and

xtrabackup. \*/

if (opt\_galera\_info) {

if (!write\_galera\_info(mysql\_connection)) {

return(false);

}

write\_current\_binlog\_file(mysql\_connection);

}

/\* 如果--binlog-info设置的是ON（默认是AUTO），则会将SHOW MASTER STATUS的相关信息，记录在xtrabackup\_binlog\_info中 \*/

if (opt\_binlog\_info == BINLOG\_INFO\_ON) {

lock\_binlog\_maybe(mysql\_connection, opt\_backup\_lock\_timeout,

opt\_backup\_lock\_retry\_count);

write\_binlog\_info(mysql\_connection);

}

if (have\_flush\_engine\_logs) {

msg\_ts("Executing FLUSH NO\_WRITE\_TO\_BINLOG ENGINE LOGS...\n");

xb\_mysql\_query(mysql\_connection,

"FLUSH NO\_WRITE\_TO\_BINLOG ENGINE LOGS", false);

}

return(true);

}

该函数的处理流程如下：

1、调用lock\_tables\_maybe函数加全局读锁。lock\_tables\_maybe函数的处理逻辑会在下篇文章介绍。

2、调用backup\_files函数备份非ibd文件。

具体来说，会备份以下面这些关键字作为后缀的文件。

const char \*ext\_list[] = {"frm", "isl", "MYD", "MYI", "MAD", "MAI",

"MRG", "TRG", "TRN", "ARM", "ARZ", "CSM", "CSV", "opt", "par",

NULL};

3、如果命令行中指定了 --slave-info ，则会执行 SHOW SLAVE STATUS 获取复制的相关信息。

4、如果命令行中指定了 --binlog-info ，则会执行 SHOW MASTER STATU 获取 Binlog 的位置点信息。

binlog-info无需显式指定，因为它的默认值为AUTO，如果开启了Binlog，则为ON。

##### backup\_finish

backup\_finish该函数位于backup\_copy.cc文件中。

bool

backup\_finish()

{

/\* release all locks \*/

/\* 释放所有锁，如果锁定了Binlog，还会解锁Binlog \*/

if (!opt\_no\_lock) {

unlock\_all(mysql\_connection);

history\_lock\_time = time(NULL) - history\_lock\_time;

} else {

history\_lock\_time = 0;

}

/\* 如果设置了--safe-slave-backup，且SQL线程停止了，会开启SQL线程 \*/

if (opt\_safe\_slave\_backup && sql\_thread\_started) {

msg("Starting slave SQL thread\n");

xb\_mysql\_query(mysql\_connection,

"START SLAVE SQL\_THREAD", false);

}

/\* Copy buffer pool dump or LRU dump \*/

/\* 拷贝ib\_buffer\_pool和ib\_lru\_dump文件 \*/

if (!opt\_rsync) {

if (opt\_dump\_innodb\_buffer\_pool) {

check\_dump\_innodb\_buffer\_pool(mysql\_connection);

}

if (buffer\_pool\_filename && file\_exists(buffer\_pool\_filename)) {

const char \*dst\_name;

dst\_name = trim\_dotslash(buffer\_pool\_filename);

copy\_file(ds\_data, buffer\_pool\_filename, dst\_name, 0);

}

if (file\_exists("ib\_lru\_dump")) {

copy\_file(ds\_data, "ib\_lru\_dump", "ib\_lru\_dump", 0);

}

if (file\_exists("ddl\_log.log")) {

copy\_file(ds\_data, "ddl\_log.log", "ddl\_log.log", 0);

}

}

msg\_ts("Backup created in directory '%s'\n", xtrabackup\_target\_dir);

if (mysql\_binlog\_position != NULL) {

msg("MySQL binlog position: %s\n", mysql\_binlog\_position);

}

if (!mysql\_slave\_position.empty() && opt\_slave\_info) {

msg("MySQL slave binlog position: %s\n",

mysql\_slave\_position.c\_str());

}

/\* 生成配置文件，backup-my.cnf \*/

if (!write\_backup\_config\_file()) {

return(false);

}

/\* 将备份的相关信息记录在xtrabackup\_info文件中 \*/

if (!write\_xtrabackup\_info(mysql\_connection)) {

return(false);

}

return(true);

}

该函数的处理流程如下：

1、释放全局读锁。

2、拷贝ib\_buffer\_pool和ib\_lru\_dump文件。

3、将备份的相关信息记录在xtrabackup\_info文件中。

如果设置了--history，还会将备份信息记录在PERCONA\_SCHEMA库下的xtrabackup\_history表中。

### XtraBackupex

## 恢复过程

如果看恢复备份集的日志，会发现和mysqld启动时非常相似，其实备份集的恢复就是类似mysqld crash后，做一次crash recover。

恢复的目的是把备份集中的数据恢复到一个一致性位点，所谓一致就是指原数据库某一时间点各引擎数据的状态，比如MyISAM中的数据对应的是15:00时间点的，InnoDB中的数据对应的是15:20 的，这种状态的数据就是不一致的。PXB备份集对应的一致点，就是备份时FTWRL的时间点，恢复出来的数据，就对应原数据库FTWRL时的状态。

因为备份时FTWRL后，数据库是处于只读的，非InnoDB数据是在持有全局读锁情况下拷贝的，所以非InnoDB数据本身就对应FTWRL时间点；InnoDB的ibd文件拷贝是在FTWRL前做的，拷贝出来的不同 ibd 文件最后更新时间点是不一样的，这种状态的ibd文件是不能直接用的，但是redo log是从备份开始一直持续拷贝的，最后的redo日志点是在持有FTWRL后取得的，所以最终通过redo应用后的ibd数据时间点也是和FTWRL一致的。

所以恢复过程只涉及InnoDB文件的恢复，非InnoDB数据是不动的。备份恢复完成后，就可以把数据文件拷贝到对应的目录，然后通过mysqld来启动了。