第3章 文 件

本章将分析构成 MySQL 数据库和 InnoDB 存储引擎表的各种类型文件。这些文件有以下这些。

- 参数文件：告诉 MySQL 实例启动时在哪里可以找到数据库文件，并且指定某些初始化参数，这些参数定义了某种内存结构的大小等设置，还会介绍各种参数的类型。
- 日志文件：用来记录 MySQL 实例对某种条件做出响应时写入的文件，如错误日志文件、二进制日志文件、慢查询日志文件、查询日志文件等。
- socket 文件：当用 UNIX 域套接字方式进行连接时需要的文件。
- pid 文件：MySQL 实例的进程 ID 文件。
- MySQL 表结构文件：用来存放 MySQL 表结构定义文件。
- 存储引擎文件：因为 MySQL 表存储引擎的关系，每个存储引擎都会有自己的文件来保存各种数据。这些存储引擎真正存储了记录和索引等数据。本章主要介绍与 InnoDB 有关的存储引擎文件。

3.1 参数文件

在第 1 章中已经介绍过了，当 MySQL 实例启动时，数据库会先去读一个配置参数文件，用来寻找数据库的各种文件所在位置以及指定某些初始化参数，这些参数通常定义了某种内存结构有多大等。在默认情况下，MySQL 实例会按照一定的顺序在指定的位置进行读取，用户只需通过命令 mysql--help | grep my.cnf 来寻找即可。

MySQL 数据库参数文件的作用和 Oracle 数据库的参数文件极其类似，不同的是，Oracle 实例在启动时若找不到参数文件，是不能进行装载（mount）操作的。MySQL 稍微有所不同，MySQL 实例可以不需要参数文件，这时所有的参数值取决于编译 MySQL 时指定的默认值和源代码中指定参数的默认值。但是，如果 MySQL 实例在默认的数据

库目录下找不到 mysql 架构，则启动同样会失败，此时可能在错误日志文件中找到如下内容：

```
090922 16:25:52  mysqld started
090922 16:25:53  InnoDB: Started; log sequence number 8 2801063211
InnoDB: !!! innodb_force_recovery is set to 1 !!!
090922 16:25:53 [ERROR] Fatal error: Can't open and lock privilege tables:
Table 'mysql.host' doesn't exist
090922 16:25:53  mysqld ended
```

MySQL 的 mysql 架构中记录了访问该实例的权限，当找不到这个架构时，MySQL 实例不会成功启动。

MySQL 数据库的参数文件是以文本方式进行存储的。用户可以直接通过一些常用的文本编辑软件（如 vi 和 emacs）进行参数的修改。

3.1.1 什么是参数

简单地说，可以把数据库参数看成一个键 / 值（key/value）对。第 2 章已经介绍了一个对于 InnoDB 存储引擎很重要的参数 innodb_buffer_pool_size。如我们将这个参数设置为 1G，即 innodb_buffer_pool_size=1G。这里的“键”是 innodb_buffer_pool_size，“值”是 1G，这就是键值对。可以通过命令 SHOW VARIABLES 查看数据库中的所有参数，也可以通过 LIKE 来过滤参数名。从 MySQL 5.1 版本开始，还可以通过 information_schema 架构下的 GLOBAL_VARIABLES 视图来进行查找，如下所示。

```
mysql>  SELECT * FROM
    ->  GLOBAL_VARIABLES
    ->  WHERE VARIABLE_NAME LIKE 'innodb_buffer%'\G;
*************************** 1. row ***************************
 VARIABLE_NAME: INNODB_BUFFER_POOL_SIZE
VARIABLE_VALUE: 1073741824
1 row in set (0.00 sec)

mysql>SHOW VARIABLES LIKE 'innodb_buffer%'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 1073741824
1 row in set (0.00 sec)
```

无论使用哪种方法，输出的信息基本上都一样的，只不过通过视图 GLOBAL_

VARIABLES 需要指定视图的列名。推荐使用命令 SHOW VARIABLES，因为这个命令使用更为简单，且各版本的 MySQL 数据库都支持。

Oracle 数据库存在所谓的隐藏参数（undocumented parameter），以供 Oracle“内部人士”使用，SQL Server 也有类似的参数。有些 DBA 曾问我，MySQL 中是否也有这类参数。我的回答是：没有，也不需要。即使 Oracle 和 SQL Server 中都有些所谓的隐藏参数，在绝大多数的情况下，这些数据库厂商也不建议用户在生产环境中对其进行很大的调整。

3.1.2 参数类型

MySQL 数据库中的参数可以分为两类：

- 动态（dynamic）参数
- 静态（static）参数

动态参数意味着可以在 MySQL 实例运行中进行更改，静态参数说明在整个实例生命周期内都不得进行更改，就好像是只读（read only）的。可以通过 SET 命令对动态的参数值进行修改，SET 的语法如下：

```
SET
| [global | session] system_var_name= expr
| [@@global. | @@session. | @@]system_var_name= expr
```

这里可以看到 global 和 session 关键字，它们表明该参数的修改是基于当前会话还是整个实例的生命周期。有些动态参数只能在会话中进行修改，如 autocommit；而有些参数修改完后，在整个实例生命周期中都会生效，如 binlog_cache_size；而有些参数既可以在会话中又可以在整个实例的生命周期内生效，如 read_buffer_size。举例如下：

```
mysql>SET read_buffer_size=524288;
Query OK, 0 rows affected (0.00 sec)

mysql>SELECT @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql>SELECT @@global.read_buffer_size\G;
*************************** 1. row ***************************
```

```
@@global.read_buffer_size: 2093056
1 row in set (0.00 sec)
```

上述示例中将当前会话的参数 read_buffer_size 从 2MB 调整为了 512KB，而用户可以看到全局的 read_buffer_size 的值仍然是 2MB，也就是说如果有另一个会话登录到 MySQL 实例，它的 read_buffer_size 的值是 2MB，而不是 512KB。这里使用了 set global|session 来改变动态变量的值。用户同样可以直接使用 SET@@globl|@@session 来更改，如下所示：

```
mysql>SET @@global.read_buffer_size=1048576;
Query OK, 0 rows affected (0.00 sec)

mysql>SELECT @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql>SELECT @@global.read_buffer_size\G;
*************************** 1. row ***************************
@@global.read_buffer_size: 1048576
1 row in set (0.00 sec)
```

这次把 read_buffer_size 全局值更改为 1MB，而当前会话的 read_buffer_size 的值还是 512KB。这里需要注意的是，对变量的全局值进行了修改，在这次的实例生命周期内都有效，但 MySQL 实例本身并不会对参数文件中的该值进行修改。也就是说，在下次启动时 MySQL 实例还是会读取参数文件。若想在数据库实例下一次启动时该参数还是保留为当前修改的值，那么用户必须去修改参数文件。要想知道 MySQL 所有动态变量的可修改范围，可以参考 MySQL 官方手册的 Dynamic System Variables 的相关内容。

对于静态变量，若对其进行修改，会得到类似如下错误：

```
mysql>SET GLOBAL datadir='/db/mysql';
ERROR 1238 (HY000): Variable 'datadir' is a read only variable
```

3.2 日志文件

日志文件记录了影响 MySQL 数据库的各种类型活动。MySQL 数据库中常见的日志文件有：

- 错误日志（error log）
- 二进制日志（binlog）
- 慢查询日志（slow query log）
- 查询日志（log）

这些日志文件可以帮助 DBA 对 MySQL 数据库的运行状态进行诊断，从而更好地进行数据库层面的优化。

3.2.1 错误日志

错误日志文件对 MySQL 的启动、运行、关闭过程进行了记录。MySQL DBA 在遇到问题时应该首先查看该文件以便定位问题。该文件不仅记录了所有的错误信息，也记录一些警告信息或正确的信息。用户可以通过命令 SHOW VARIABLES LIKE 'log_error' 来定位该文件，如：

```
mysql> SHOW VARIABLES LIKE 'log_error'\G;
*************************** 1. row ***************************
Variable_name: log_error
        Value: /mysql_data_2/stargazer.log
1 row in set (0.00 sec)

mysql> system hostname
stargazer
```

可以看到错误文件的路径和文件名，在默认情况下错误文件的文件名为服务器的主机名。如上面看到的，该主机名为 stargazer，所以错误文件名为 startgazer.err。当出现 MySQL 数据库不能正常启动时，第一个必须查找的文件应该就是错误日志文件，该文件记录了错误信息，能很好地指导用户发现问题。当数据库不能重启时，通过查错误日志文件可以得到如下内容：

```
[root@nineyou0-43 data]# tail -n 50 nineyou0-43.err
090924 11:31:18  mysqld started
090924 11:31:18  InnoDB: Started; log sequence number 8 2801063331
090924 11:31:19 [ERROR] Fatal error: Can't open and lock privilege tables:
Table 'mysql.host' doesn't exist
090924 11:31:19  mysqld ended
```

这里，错误日志文件提示了找不到权限库 mysql，所以启动失败。有时用户可以直

接在错误日志文件中得到优化的帮助，因为有些警告（warning）很好地说明了问题所在。而这时可以不需要通过查看数据库状态来得知，例如，下面的错误文件中的信息可能告诉用户需要增大 InnoDB 存储引擎的 redo log：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

3.2.2 慢查询日志

3.2.1 小节提到可以通过错误日志得到一些关于数据库优化的信息，而慢查询日志（slow log）可帮助 DBA 定位可能存在问题的 SQL 语句，从而进行 SQL 语句层面的优化。例如，可以在 MySQL 启动时设一个阈值，将运行时间超过该值的所有 SQL 语句都记录到慢查询日志文件中。DBA 每天或每过一段时间对其进行检查，确认是否有 SQL 语句需要进行优化。该阈值可以通过参数 long_query_time 来设置，默认值为 10，代表 10 秒。

在默认情况下，MySQL 数据库并不启动慢查询日志，用户需要手工将这个参数设为 ON：

```
mysql> SHOW VARIABLES LIKE 'long_query_time'\G;
*************************** 1. row ***************************
Variable_name: long_query_time
        Value: 10.000000
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'log_slow_queries'\G;
*************************** 1. row ***************************
```

```
Variable_name: log_slow_queries
        Value: ON
1 row in set (0.00 sec)
```

这里有两点需要注意。首先，设置 long_query_time 这个阈值后，MySQL 数据库会记录运行时间超过该值的所有 SQL 语句，但运行时间正好等于 long_query_time 的情况并不会被记录下。也就是说，在源代码中判断的是大于 long_query_time，而非大于等于。其次，从 MySQL 5.1 开始，long_query_time 开始以微秒记录 SQL 语句运行的时间，之前仅用秒为单位记录。而这样可以更精确地记录 SQL 的运行时间，供 DBA 分析。对 DBA 来说，一条 SQL 语句运行 0.5 秒和 0.05 秒是非常不同的，前者可能已经进行了表扫，后面可能是进行了索引。

另一个和慢查询日志有关的参数是 log_queries_not_using_indexes，如果运行的 SQL 语句没有使用索引，则 MySQL 数据库同样会将这条 SQL 语句记录到慢查询日志文件。首先确认打开了 log_queries_not_using_indexes：

```
mysql> SHOW VARIABLES LIKE 'log_queries_not_using_indexes'\G;
*************************** 1. row ***************************
Variable_name: log_queries_not_using_indexes
        Value: ON
1 row in set (0.00 sec)
```

MySQL 5.6.5 版本开始新增了一个参数 log_throttle_queries_not_using_indexes，用来表示每分钟允许记录到 slow log 的且未使用索引的 SQL 语句次数。该值默认为 0，表示没有限制。在生产环境下，若没有使用索引，此类 SQL 语句会频繁地被记录到 slow log，从而导致 slow log 文件的大小不断增加，故 DBA 可通过此参数进行配置。

DBA 可以通过慢查询日志来找出有问题的 SQL 语句，对其进行优化。然而随着 MySQL 数据库服务器运行时间的增加，可能会有越来越多的 SQL 查询被记录到了慢查询日志文件中，此时要分析该文件就显得不是那么简单和直观的了。而这时 MySQL 数据库提供的 mysqldumpslow 命令，可以很好地帮助 DBA 解决该问题：

```
[root@nh122-190 data]# mysqldumpslow nh122-190-slow.log
Reading mysql slow query log from nh122-190-slow.log
Count: 11  Time=10.00s (110s)  Lock=0.00s (0s)  Rows=0.0 (0), dbother[dbother]@
localhost
  insert into test.DbStatus select now(),(N-com_select)/(N-uptime),(N-com_insert)/
(N-uptime),(N-com_update)/(N-uptime),(N-com_delete)/(N-uptime),N-(N/N),N-(N/N),N.N/
N,N-N/(N*N),GetCPULoadInfo(N) from test.CheckDbStatus order by check_id desc limit N
```

```
    Count: 653  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=0.0 (0), 9YOUgs_SC[9YOUgs_
SC]@[192.168.43.7]
      select custom_name_one from 'low_game_schema'.'role_details' where role_id='S'
    rse and summarize the MySQL slow query log. Options are

      --verbose    verbose
      --debug      debug
      --help       write this text to standard output

      -v           verbose
      -d           debug
      -s ORDER     what to sort by (al, at, ar, c, l, r, t), 'at' is default
                   al: average lock time
                   ar: average rows sent
                   at: average query time
                    c: count
                    l: lock time
                    r: rows sent
                    t: query time
      -r           reverse the sort order (largest last instead of first)
      -t NUM       just show the top n queries
      -a           don't abstract all numbers to N and strings to 'S'
      -n NUM       abstract numbers with at least n digits within names
      -g PATTERN   grep: only consider stmts that include this string
      -h HOSTNAME  hostname of db server for *-slow.log filename (can be wildcard),
                   default is '*', i.e. match all
      -i NAME      name of server instance (if using mysql.server startup script)
      -l           don't subtract lock time from total time
```

如果用户希望得到执行时间最长的 10 条 SQL 语句，可以运行如下命令：

```
    [root@nh119-141 data]# mysqldumpslow -s al -n 10 david.log
    Reading mysql slow query log from david.log
    Count: 5  Time=0.00s (0s)  Lock=0.20s (1s)  Rows=4.4 (22), Audition[Audition]@
[192.168.30.108]
      SELECT OtherSN, State FROM wait_friend_info WHERE UserSN = N

    Count: 1  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=1.0 (1), audition-kr[audition-
kr]@[192.168.30.105]
      SELECT COUNT(N) FROM famverifycode WHERE UserSN=N AND verifycode='S'
    ……
```

MySQL 5.1 开始可以将慢查询的日志记录放入一张表中，这使得用户的查询更加方便和直观。慢查询表在 mysql 架构下，名为 slow_log，其表结构定义如下：

```
mysql> SHOW CREATE TABLE mysql.slow_log\G;
*************************** 1. row ***************************
       Table: slow_log
Create Table: CREATE TABLE 'slow_log' (
  'start_time' timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
  'user_host' mediumtext NOT NULL,
  'query_time' time NOT NULL,
  'lock_time' time NOT NULL,
  'rows_sent' int(11) NOT NULL,
  'rows_examined' int(11) NOT NULL,
  'db' varchar(512) NOT NULL,
  'last_insert_id' int(11) NOT NULL,
  'insert_id' int(11) NOT NULL,
  'server_id' int(11) NOT NULL,
  'sql_text' mediumtext NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='Slow log'
1 row in set (0.00 sec)
```

参数 log_output 指定了慢查询输出的格式，默认为 FILE，可以将它设为 TABLE，然后就可以查询 mysql 架构下的 slow_log 表了，如：

```
mysql>SHOW VARIABLES LIKE 'log_output'\G;
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output    | FILE |
+---------------+---------+
1 row in set (0.00 sec)

mysql>SET GLOBAL log_output='TABLE';
Query OK, 0 rows affected (0.00 sec)

mysql>SHOW VARIABLES LIKE 'log_output'\G;
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output  | TABLE |
+---------------+---------+
1 row in set (0.00 sec)

mysql> select sleep(10)\G;
+-----------+
| sleep(10)|
```

```
+-----------+
|         0 |
+-----------+
1 row in set (10.01 sec)

mysql>  SELECT * FROM mysql.slow_log\G;
*************************** 1. row ***************************
    start_time: 2009-09-25 13:44:29
     user_host: david[david] @ localhost []
    query_time: 00:00:09
     lock_time: 00:00:00
     rows_sent: 1
 rows_examined: 0
            db: mysql
last_insert_id: 0
     insert_id: 0
     server_id: 0
      sql_text: select sleep(10)
1 row in set (0.00 sec)
```

参数 log_output 是动态的，并且是全局的，因此用户可以在线进行修改。在上表中人为设置了睡眠（sleep）10 秒，那么这句 SQL 语句就会被记录到 slow_log 表了。

查看 slow_log 表的定义会发现该表使用的是 CSV 引擎，对大数据量下的查询效率可能不高。用户可以把 slow_log 表的引擎转换到 MyISAM，并在 start_time 列上添加索引以进一步提高查询的效率。但是，如果已经启动了慢查询，将会提示错误：

```
mysql>ALTER TABLE mysql.slow_log ENGINE=MyISM;
ERROR 1580 (HY000): You cannot 'ALTER' a log table if logging is enabled

mysql>SET GLOBAL slow_query_log=off;
Query OK, 0 rows affected (0.00 sec)

mysql>ALTER TABLE mysql.slow_log ENGINE=MyISAM;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

不能忽视的是，将 slow_log 表的存储引擎更改为 MyISAM 后，还是会对数据库造成额外的开销。不过好在很多关于慢查询的参数都是动态的，用户可以方便地在线进行设置或修改。

MySQL 的 slow log 通过运行时间来对 SQL 语句进行捕获，这是一个非常有用的优化技巧。但是当数据库的容量较小时，可能因为数据库刚建立，此时非常大的可能是数

据全部被缓存在缓冲池中，SQL 语句运行的时间可能都是非常短的，一般都是 0.5 秒。

InnoSQL 版本加强了对于 SQL 语句的捕获方式。在原版 MySQL 的基础上在 slow log 中增加了对于逻辑读取（logical reads）和物理读取（physical reads）的统计。这里的物理读取是指从磁盘进行 IO 读取的次数，逻辑读取包含所有的读取，不管是磁盘还是缓冲池。例如：

```
    # Time: 111227 23:49:16
    # User@Host: root[root] @ localhost [127.0.0.1]
    # Query_time: 6.081214 Lock_time: 0.046800 Rows_sent: 42 Rows_examined: 727558
Logical_reads: 91584 Physical_reads: 19
    use tpcc;
    SET timestamp=1325000956;
    SELECT orderid,customerid,employeeid,orderdate
    FROM orders
    WHERE orderdate IN
    ( SELECT MAX(orderdate)
    FROM orders
    GROUP BY (DATE_FORMAT(orderdate,'%Y%M'))
    );
```

从上面的例子可以看到该子查询的逻辑读取次数是 91 584 次，物理读取为 19 次。从逻辑读与物理读的比例上看，该 SQL 语句可进行优化。

用户可以通过额外的参数 long_query_io 将超过指定逻辑 IO 次数的 SQL 语句记录到 slow log 中。该值默认为 100，即表示对于逻辑读取次数大于 100 的 SQL 语句，记录到 slow log 中。而为了兼容原 MySQL 数据库的运行方式，还添加了参数 slow_query_type，用来表示启用 slow log 的方式，可选值为：

- 0 表示不将 SQL 语句记录到 slow log
- 1 表示根据运行时间将 SQL 语句记录到 slow log
- 2 表示根据逻辑 IO 次数将 SQL 语句记录到 slow log
- 3 表示根据运行时间及逻辑 IO 次数将 SQL 语句记录到 slow log

3.2.3 查询日志

查询日志记录了所有对 MySQL 数据库请求的信息，无论这些请求是否得到了正确的执行。默认文件名为：主机名 .log。如查看一个查询日志：

```
[root@nineyou0-43 data]# tail nineyou0-43.log
090925 11:00:24    44 Connect     zlm@192.168.0.100 on
44 Query       SET AUTOCOMMIT=0
                44 Query       set autocommit=0
                44 Quit
090925 11:02:37  45 Connect    Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:03:51  46 Connect    Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:04:38   23 Query       rollback
```

通过上述查询日志会发现，查询日志甚至记录了对 Access denied 的请求，即对于未能正确执行的 SQL 语句，查询日志也会进行记录。同样地，从 MySQL 5.1 开始，可以将查询日志的记录放入 mysql 架构下的 general_log 表中，该表的使用方法和前面小节提到的 slow_log 基本一样，这里不再赘述。

3.2.4 二进制日志

二进制日志（binary log）记录了对 MySQL 数据库执行更改的所有操作，但是不包括 SELECT 和 SHOW 这类操作，因为这类操作对数据本身并没有修改。然而，若操作本身并没有导致数据库发生变化，那么该操作可能也会写入二进制日志。例如：

```
mysql> UPDATE t SET a = 1 WHERE a = 2;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> SHOW MASTER STATUS\G;
*************************** 1. row ***************************
             File: mysqld.000008
         Position: 383
     Binlog_Do_DB:
 Binlog_Ignore_DB:
Executed_Gtid_Set:
1 row in set (0.00 sec)

mysql> SHOW BINLOG EVENTS IN 'mysqld.000008'\G;
*************************** 1. row ***************************
   Log_name: mysqld.000008
        Pos: 4
 Event_type: Format_desc
  Server_id: 1
```

```
End_log_pos: 120
       Info: Server ver: 5.6.6-m9-log, Binlog ver: 4
*************************** 2. row ***************************
   Log_name: mysqld.000008
        Pos: 120
 Event_type: Query
  Server_id: 1
End_log_pos: 199
       Info: BEGIN
*************************** 3. row ***************************
   Log_name: mysqld.000008
        Pos: 199
 Event_type: Query
  Server_id: 1
End_log_pos: 303
       Info: use 'test'; UPDATE t SET a = 1 WHERE a = 2
*************************** 4. row ***************************
   Log_name: mysqld.000008
        Pos: 303
 Event_type: Query
  Server_id: 1
End_log_pos: 383
       Info: COMMIT
4 rows in set (0.00 sec)
```

从上述例子中可以看到，MySQL 数据库首先进行 UPDATE 操作，从返回的结果看到 Changed 为 0，这意味着该操作并没有导致数据库的变化。但是通过命令 SHOW BINLOG EVENT 可以看出在二进制日志中的确进行了记录。

如果用户想记录 SELECT 和 SHOW 操作，那只能使用查询日志，而不是二进制日志。此外，二进制日志还包括了执行数据库更改操作的时间等其他额外信息。总的来说，二进制日志主要有以下几种作用。

- **恢复**（recovery）：某些数据的恢复需要二进制日志，例如，在一个数据库全备文件恢复后，用户可以通过二进制日志进行 point-in-time 的恢复。
- **复制**（replication）：其原理与恢复类似，通过复制和执行二进制日志使一台远程的 MySQL 数据库（一般称为 slave 或 standby）与一台 MySQL 数据库（一般称为 master 或 primary）进行实时同步。
- **审计**（audit）：用户可以通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入的攻击。

通过配置参数 log-bin［=name］可以启动二进制日志。如果不指定 name，则默认二进制日志文件名为主机名，后缀名为二进制日志的序列号，所在路径为数据库所在目录（datadir），如：

```
mysql> show variables like 'datadir';
+---------------+-----------------------------+
| Variable_name | Value      |
+---------------+-----------------------------+
| datadir       | /usr/local/mysql/data/ |
+---------------+-----------------------------+
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/;
total 2.1G
-rw-rw----  1 mysql mysql 6.5M Sep 25 15:13 bin_log.000001
-rw-rw----  1 mysql mysql   17 Sep 25 00:32 bin_log.index
-rw-rw----  1 mysql mysql 300M Sep 25 15:13 ibdata1
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile0
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile1
drwxr-xr-x  2 mysql mysql 4.0K May  7 10:08 mysql
drwx------  2 mysql mysql 4.0K May  7 10:09 test
```

这里的 bin_log.00001 即为二进制日志文件，我们在配置文件中指定了名字，所以没有用默认的文件名。bin_log.index 为二进制的索引文件，用来存储过往产生的二进制日志序号，在通常情况下，不建议手工修改这个文件。

二进制日志文件在默认情况下并没有启动，需要手动指定参数来启动。可能有人会质疑，开启这个选项是否会对数据库整体性能有所影响。不错，开启这个选项的确会影响性能，但是性能的损失十分有限。根据 MySQL 官方手册中的测试表明，开启二进制日志会使性能下降 1%。但考虑到可以使用复制（replication）和 point-in-time 的恢复，这些性能损失绝对是可以且应该被接受的。

以下配置文件的参数影响着二进制日志记录的信息和行为：

- max_binlog_size
- binlog_cache_size
- sync_binlog
- binlog-do-db
- binlog-ignore-db

- log-slave-update
- binlog_format

参数 max_binlog_size 指定了单个二进制日志文件的最大值，如果超过该值，则产生新的二进制日志文件，后缀名 +1，并记录到 .index 文件。从 MySQL 5.0 开始的默认值为 1 073 741 824，代表 1 G（在之前版本中 max_binlog_size 默认大小为 1.1G）。

当使用事务的表存储引擎（如 InnoDB 存储引擎）时，所有未提交（uncommitted）的二进制日志会被记录到一个缓存中去，等该事务提交（committed）时直接将缓冲中的二进制日志写入二进制日志文件，而该缓冲的大小由 binlog_cache_size 决定，默认大小为 32K。此外，binlog_cache_size 是基于会话（session）的，也就是说，当一个线程开始一个事务时，MySQL 会自动分配一个大小为 binlog_cache_size 的缓存，因此该值的设置需要相当小心，不能设置过大。当一个事务的记录大于设定的 binlog_cache_size 时，MySQL 会把缓冲中的日志写入一个临时文件中，因此该值又不能设得太小。通过 SHOW GLOBAL STATUS 命令查看 binlog_cache_use、binlog_cache_disk_use 的状态，可以判断当前 binlog_cache_size 的设置是否合适。Binlog_cache_use 记录了使用缓冲写二进制日志的次数，binlog_cache_disk_use 记录了使用临时文件写二进制日志的次数。现在来看一个数据库的状态：

```
mysql> show variables like 'binlog_cache_size';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| binlog_cache_size | 32768 |
+-------------------+-------+
1 row in set (0.00 sec)

mysql> show global status like 'binlog_cache%';
+-----------------------+---------------+
| Variable_name         | Value         |
+-----------------------+---------------+
| binlog_cache_disk_use | 0             |
| binlog_cache_use      | 33553         |
+-----------------------+---------------+
2 rows in set (0.00 sec)
```

使用缓冲次数为 33 553，临时文件使用次数为 0。看来 32KB 的缓冲大小对于当前这个 MySQL 数据库完全够用，暂时没有必要增加 binlog_cache_size 的值。

在默认情况下，二进制日志并不是在每次写的时候同步到磁盘（用户可以理解为缓冲写）。因此，当数据库所在操作系统发生宕机时，可能会有最后一部分数据没有写入二进制日志文件中，这会给恢复和复制带来问题。参数 sync_binlog=［N］表示每写缓冲多少次就同步到磁盘。如果将 N 设为 1，即 sync_binlog=1 表示采用同步写磁盘的方式来写二进制日志，这时写操作不使用操作系统的缓冲来写二进制日志。sync_binlog 的默认值为 0，如果使用 InnoDB 存储引擎进行复制，并且想得到最大的高可用性，建议将该值设为 ON。不过该值为 ON 时，确实会对数据库的 IO 系统带来一定的影响。

但是，即使将 sync_binlog 设为 1，还是会有一种情况导致问题的发生。当使用 InnoDB 存储引擎时，在一个事务发出 COMMIT 动作之前，由于 sync_binlog 为 1，因此会将二进制日志立即写入磁盘。如果这时已经写入了二进制日志，但是提交还没有发生，并且此时发生了宕机，那么在 MySQL 数据库下次启动时，由于 COMMIT 操作并没有发生，这个事务会被回滚掉。但是二进制日志已经记录了该事务信息，不能被回滚。这个问题可以通过将参数 innodb_support_xa 设为 1 来解决，虽然 innodb_support_xa 与 XA 事务有关，但它同时也确保了二进制日志和 InnoDB 存储引擎数据文件的同步。

参数 binlog-do-db 和 binlog-ignore-db 表示需要写入或忽略写入哪些库的日志。默认为空，表示需要同步所有库的日志到二进制日志。

如果当前数据库是复制中的 slave 角色，则它不会将从 master 取得并执行的二进制日志写入自己的二进制日志文件中去。如果需要写入，要设置 log-slave-update。如果需要搭建 master=>slave=>slave 架构的复制，则必须设置该参数。

binlog_format 参数十分重要，它影响了记录二进制日志的格式。在 MySQL 5.1 版本之前，没有这个参数。所有二进制文件的格式都是基于 SQL 语句（statement）级别的，因此基于这个格式的二进制日志文件的复制（Replication）和 Oracle 的逻辑 Standby 有点相似。同时，对于复制是有一定要求的。如在主服务器运行 rand、uuid 等函数，又或者使用触发器等操作，这些都可能会导致主从服务器上表中数据的不一致（not sync）。另一个影响是，会发现 InnoDB 存储引擎的默认事务隔离级别是 REPEATABLE READ。这其实也是因为二进制日志文件格式的关系，如果使用 READ COMMITTED 的事务隔离级别（大多数数据库，如 Oracle，Microsoft SQL Server 数据库的默认隔离级别），会出现类似丢失更新的现象，从而出现主从数据库上的数据不一致。

MySQL 5.1 开始引入了 binlog_format 参数，该参数可设的值有 STATEMENT、

ROW 和 MIXED。

（1）STATEMENT 格式和之前的 MySQL 版本一样，二进制日志文件记录的是日志的逻辑 SQL 语句。

（2）在 ROW 格式下，二进制日志记录的不再是简单的 SQL 语句了，而是记录表的行更改情况。基于 ROW 格式的复制类似于 Oracle 的物理 Standby（当然，还是有些区别）。同时，对上述提及的 Statement 格式下复制的问题予以解决。从 MySQL 5.1 版本开始，如果设置了 binlog_format 为 ROW，可以将 InnoDB 的事务隔离基本设为 READ COMMITTED，以获得更好的并发性。

（3）在 MIXED 格式下，MySQL 默认采用 STATEMENT 格式进行二进制日志文件的记录，但是在一些情况下会使用 ROW 格式，可能的情况有：

1）表的存储引擎为 NDB，这时对表的 DML 操作都会以 ROW 格式记录。

2）使用了 UUID()、USER()、CURRENT_USER()、FOUND_ROWS()、ROW_COUNT() 等不确定函数。

3）使用了 INSERT DELAY 语句。

4）使用了用户定义函数（UDF）。

5）使用了临时表（temporary table）。

此外，binlog_format 参数还有对于存储引擎的限制，如表 3-1 所示。

表 3-1 存储引擎对二进制日志格式的支持情况

存储引擎	Row 格式	Statement 格式
InnoDB	Yes	Yes
MyISAM	Yes	Yes
HEAP	Yes	Yes
MERGE	Yes	Yes
NDB	Yes	No
Archive	Yes	Yes
CSV	Yes	Yes
Federate	Yes	Yes
Blockhole	No	Yes

binlog_format 是动态参数，因此可以在数据库运行环境下进行更改，例如，我们可以将当前会话的 binlog_format 设为 ROW，如：

```
mysql>SET @@session.binlog_format='ROW';
```

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

mysql>SELECT@@session.binlog_format;
+-------------------------------+
| @@session.binlog_format |
+-------------------------------+
| ROW                 |
+-------------------------------+
1 row in set (0.00 sec)
```

当然，也可以将全局的 binlog_format 设置为想要的格式，不过通常这个操作会带来问题，运行时要确保更改后不会对复制带来影响。如：

```
mysql>SET GLOBAL binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)

mysql>SELECT @@global.binlog_format;
+-------------------------------+
| @@global.binlog_format |
+-------------------------------+
| ROW             |
+-------------------------------+
1 row in set (0.00 sec)
```

在通常情况下，我们将参数 binlog_format 设置为 ROW，这可以为数据库的恢复和复制带来更好的可靠性。但是不能忽略的一点是，这会带来二进制文件大小的增加，有些语句下的 ROW 格式可能需要更大的容量。比如我们有两张一样的表，大小都为 100W，分别执行 UPDATE 操作，观察二进制日志大小的变化：

```
mysql>SELECT @@session.binlog_format\G;
*************************** 1. row ***************************
@@session.binlog_format: STATEMENT
1 row in set (0.00 sec)

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 106
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql>UPDATE t1 SET username=UPPER(username);
```

```
Query OK, 89279 rows affected (1.83 sec)
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

可以看到，在 binlog_format 格式为 STATEMENT 的情况下，执行 UPDATE 语句后二进制日志大小只增加了 200 字节（306–106）。如果使用 ROW 格式，同样对 t2 表进行操作，可以看到：

```
mysql>SET SESSION binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql>UPDATE t2 SET username=UPPER(username);
Query OK, 89279 rows affected (2.42 sec)
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 13782400
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

这时会惊讶地发现，同样的操作在 ROW 格式下竟然需要 13 782 094 字节，二进制日志文件的大小差不多增加了 13MB，要知道 t2 表的大小也不超过 17MB。而且执行时间也有所增加（这里我设置了 sync_binlog=1）。这是因为这时 MySQL 数据库不再将逻辑的 SQL 操作记录到二进制日志中，而是记录对于每行的更改。

上面的这个例子告诉我们，将参数 binlog_format 设置为 ROW，会对磁盘空间要求有一定的增加。而由于复制是采用传输二进制日志方式实现的，因此复制的网络开销也有所增加。

二进制日志文件的文件格式为二进制（好像有点废话），不能像错误日志文件、慢查询日志文件那样用 cat、head、tail 等命令来查看。要查看二进制日志文件的内容，必须通过 MySQL 提供的工具 mysqlbinlog。对于 STATEMENT 格式的二进制日志文件，在使用 mysqlbinlog 后，看到的就是执行的逻辑 SQL 语句，如：

```
[root@nineyou0-43 data]# mysqlbinlog --start-position=203 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
….
#090927 15:43:11 server id 1  end_log_pos 376   Query   thread_id=188   exec_
time=1     error_code=0
SET TIMESTAMP=1254037391/*!*/;
update t2 set username=upper(username) where id=1
/*!*/;
# at 376
#090927 15:43:11 server id 1  end_log_pos 403   Xid = 1009
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

通过 SQL 语句 UPDATE t2 SET username=UPPER（username）WHERE id=1 可以看到，二进制日志的记录采用 SQL 语句的方式（为了排版的方便，省去了一些开始的信息）。在这种情况下，mysqlbinlog 和 Oracle LogMiner 类似。但是如果这时使用 ROW 格式的记录方式，会发现 mysqlbinlog 的结果变得“不可读”（unreadable），如：

```
[root@nineyou0-43 data]# mysqlbinlog  --start-position=1065 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
……
# at 1135
# at 1198
#090927 15:53:52 server id 1  end_log_pos 1198  Table_map: 'member'.'t2' mapped
to number 58
#090927 15:53:52 server id 1  end_log_pos 1378  Update_rows: table id 58 flags:
STMT_END_F

BINLOG '
```

```
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
# at 1378
#090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

这里看不到执行的 SQL 语句，反而是一大串用户不可读的字符。其实只要加上参数 -v 或 -vv 就能清楚地看到执行的具体信息了。-vv 会比 -v 多显示出更新的类型。加上 -vv 选项，可以得到：

```
[root@nineyou0-43 data]# mysqlbinlog -vv  --start-position=1065 test.000004
……
BINLOG '
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
### UPDATE member.t2
### WHERE
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
###   @2='david' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
###   @3='family' /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
### SET
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
```

```
###   @2='DAVID' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
###   @3=family /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
# at 1378
#090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

现在 mysqlbinlog 向我们解释了它具体做的事情。可以看到，一句简单的 update t2 set username=upper(username)where id=1 语句记录了对于整个行更改的信息，这也解释了为什么前面更新了 10W 行的数据，在 ROW 格式下，二进制日志文件会增大 13MB。

3.3 套接字文件

前面提到过，在 UNIX 系统下本地连接 MySQL 可以采用 UNIX 域套接字方式，这种方式需要一个套接字（socket）文件。套接字文件可由参数 socket 控制。一般在 /tmp 目录下，名为 mysql.sock：

```
mysql>SHOW VARIABLES LIKE 'socket'\G;
*************************** 1. row ***************************
Variable_name: socket
        Value: /tmp/mysql.sock
1 row in set (0.00 sec)
```

3.4 pid 文件

当 MySQL 实例启动时，会将自己的进程 ID 写入一个文件中——该文件即为 pid 文件。该文件可由参数 pid_file 控制，默认位于数据库目录下，文件名为主机名 .pid：

```
mysql> show variables like 'pid_file'\G;
*************************** 1. row ***************************
Variable_name: pid_file
        Value: /usr/local/mysql/data/xen-server.pid
1 row in set (0.00 sec)
```

3.5 表结构定义文件

因为 MySQL 插件式存储引擎的体系结构的关系，MySQL 数据的存储是根据表进行的，每个表都会有与之对应的文件。但不论表采用何种存储引擎，MySQL 都有一个以 frm 为后缀名的文件，这个文件记录了该表的表结构定义。

frm 还用来存放视图的定义，如用户创建了一个 v_a 视图，那么对应地会产生一个 v_a.frm 文件，用来记录视图的定义，该文件是文本文件，可以直接使用 cat 命令进行查看：

```
[root@xen-server test]# cat v_a.frm
TYPE=VIEW
query=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
md5=4eda70387716a4d6c96f3042dd68b742
updatable=1
algorithm=0
definer_user=root
definer_host=localhost
suid=2
with_check_option=0
timestamp=2010-08-04 07:23:36
create-version=1
source=select * from a
client_cs_name=utf8
connection_cl_name=utf8_general_ci
view_body_utf8=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
```

3.6 InnoDB 存储引擎文件

之前介绍的文件都是 MySQL 数据库本身的文件，和存储引擎无关。除了这些文件外，每个表存储引擎还有其自己独有的文件。本节将具体介绍与 InnoDB 存储引擎密切相关的文件，这些文件包括重做日志文件、表空间文件。

3.6.1 表空间文件

InnoDB 采用将存储的数据按表空间（tablespace）进行存放的设计。在默认配置下会有一个初始大小为 10MB，名为 ibdata1 的文件。该文件就是默认的表空间文件（tablespace file），用户可以通过参数 innodb_data_file_path 对其进行设置，格式如下：

```
innodb_data_file_path=datafile_spec1[;datafile_spec2]...
```

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，如：

```
[mysqld]
innodb_data_file_path = /db/ibdata1:2000M;/dr2/db/ibdata2:2000M:autoextend
```

这里将 /db/ibdata1 和 /dr2/db/ibdata2 两个文件用来组成表空间。若这两个文件位于不同的磁盘上，磁盘的负载可能被平均，因此可以提高数据库的整体性能。同时，两个文件的文件名后都跟了属性，表示文件 idbdata1 的大小为 2000MB，文件 ibdata2 的大小为 2000MB，如果用完了这 2000MB，该文件可以自动增长（autoextend）。

设置 innodb_data_file_path 参数后，所有基于 InnoDB 存储引擎的表的数据都会记录到该共享表空间中。若设置了参数 innodb_file_per_table，则用户可以将每个基于 InnoDB 存储引擎的表产生一个独立表空间。独立表空间的命名规则为：表名 .ibd。通过这样的方式，用户不用将所有数据都存放于默认的表空间中。下面这台 MySQL 数据库服务器设置了 innodb_file_per_table，故可以观察到：

```
mysql>SHOW VARIABLES LIKE 'innodb_file_per_table'\G;
*************************** 1. row ***************************
Variable_name: innodb_file_per_table
        Value: ON
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/member/*
-rw-r-----  1 mysql mysql  8.7K  2009-02-24    /usr/local/mysql/data/member/
Profile.frm
-rw-r-----  1 mysql mysql 1.7G  9月 25 11:13 /usr/local/mysql/data/member/
Profile.ibd
-rw-rw----  1 mysql mysql 8.7K  9月 27 13:38 /usr/local/mysql/data/member/
t1.frm
-rw-rw----  1 mysql mysql  17M  9月 27 13:40 /usr/local/mysql/data/member/
t1.ibd
-rw-rw----  1 mysql mysql 8.7K  9月 27 15:42 /usr/local/mysql/data/member/
t2.frm
```

```
-rw-rw----  1 mysql mysql  17M  9月 27 15:54 /usr/local/mysql/data/member/
t2.ibd
```

表 Profile、t1 和 t2 都是基于 InnoDB 存储的表，由于设置参数 innodb_file_per_table=ON，因此产生了单独的 .ibd 独立表空间文件。需要注意的是，这些单独的表空间文件仅存储该表的数据、索引和插入缓冲 BITMAP 等信息，其余信息还是存放在默认的表空间中。图 3-1 显示了 InnoDB 存储引擎对于文件的存储方式：

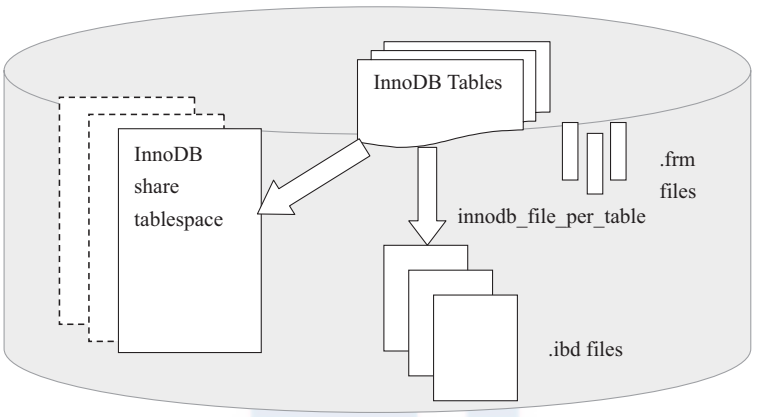

图 3-1 InnoDB 表存储引擎文件

3.6.2 重做日志文件

在默认情况下，在 InnoDB 存储引擎的数据目录下会有两个名为 ib_logfile0 和 ib_logfile1 的文件。在 MySQL 官方手册中将其称为 InnoDB 存储引擎的日志文件，不过更准确的定义应该是重做日志文件（redo log file）。为什么强调是重做日志文件呢？因为重做日志文件对于 InnoDB 存储引擎至关重要，它们记录了对于 InnoDB 存储引擎的事务日志。

当实例或介质失败（media failure）时，重做日志文件就能派上用场。例如，数据库由于所在主机掉电导致实例失败，InnoDB 存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。

每个 InnoDB 存储引擎至少有 1 个重做日志文件组（group），每个文件组下至少有 2 个重做日志文件，如默认的 ib_logfile0 和 ib_logfile1。为了得到更高的可靠性，用户可以设置多个的镜像日志组（mirrored log groups），将不同的文件组放在不同的磁盘上，以此提高重做日志的高可用性。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 存储引擎先写重做日志文件 1，当达到文件的最后时，会切换至重做日志文件 2，再当重做日志文件 2 也被写满时，会再切换到重做日志文件 1 中。图 3-2 显示了一个拥有 3 个重做日志文件的重做日志文件组。

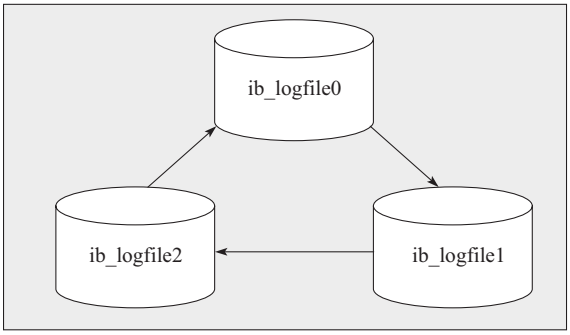

图 3-2 日志文件组

下列参数影响着重做日志文件的属性：

- innodb_log_file_size
- innodb_log_files_in_group
- innodb_mirrored_log_groups
- innodb_log_group_home_dir

参数 innodb_log_file_size 指定每个重做日志文件的大小。在 InnoDB1.2.x 版本之前，重做日志文件总的大小不得大于等于 4GB，而 1.2.x 版本将该限制扩大为了 512GB。

参数 innodb_log_files_in_group 指定了日志文件组中重做日志文件的数量，默认为 2。参数 innodb_mirrored_log_groups 指定了日志镜像文件组的数量，默认为 1，表示只有一个日志文件组，没有镜像。若磁盘本身已经做了高可用的方案，如磁盘阵列，那么可以不开启重做日志镜像的功能。最后，参数 innodb_log_group_home_dir 指定了日志文件组所在路径，默认为 ./，表示在 MySQL 数据库的数据目录下。以下显示了一个关于重做日志组的配置：

```
mysql>SHOW VARIABLES LIKE 'innodb%log%'\G;
……
*************************** 4. row ***************************
Variable_name: innodb_log_file_size
        Value: 5242880
*************************** 5. row ***************************
Variable_name: innodb_log_files_in_group
        Value: 2
*************************** 6. row ***************************
Variable_name: innodb_log_group_home_dir
        Value: ./
*************************** 7. row ***************************
Variable_name: innodb_mirrored_log_groups
        Value: 1
7 rows in set (0.00 sec)
```

重做日志文件的大小设置对于 InnoDB 存储引擎的性能有着非常大的影响。一方面

重做日志文件不能设置得太大，如果设置得很大，在恢复时可能需要很长的时间；另一方面又不能设置得太小了，否则可能导致一个事务的日志需要多次切换重做日志文件。此外，重做日志文件太小会导致频繁地发生 async checkpoint，导致性能的抖动。例如，用户可能会在错误日志中看到如下警告信息：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

上面错误集中在 InnoDB:ERROR:the age of the last checkpoint is 9433645，InnoDB:which exceeds the log group capacity 9433498。这是因为重做日志有一个 capacity 变量，该值代表了最后的检查点不能超过这个阈值，如果超过则必须将缓冲池（innodb buffer pool）中脏页列表（flush list）中的部分脏数据页写回磁盘，这时会导致用户线程的阻塞。

也许有人会问，既然同样是记录事务日志，和之前介绍的二进制日志有什么区别？

首先，二进制日志会记录所有与 MySQL 数据库有关的日志记录，包括 InnoDB、MyISAM、Heap 等其他存储引擎的日志。而 InnoDB 存储引擎的重做日志只记录有关该存储引擎本身的事务日志。

其次，记录的内容不同，无论用户将二进制日志文件记录的格式设为 STATEMENT 还是 ROW，又或者是 MIXED，其记录的都是关于一个事务的具体操作内容，即该日志是逻辑日志。而 InnoDB 存储引擎的重做日志文件记录的是关于每个页（Page）的更改的物理情况。

此外，写入的时间也不同，二进制日志文件仅在事务提交前进行提交，即只写磁盘一次，不论这时该事务多大。而在事务进行的过程中，却不断有重做日志条目（redo entry）被写入到重做日志文件中。

在 InnoDB 存储引擎中，对于各种不同的操作有着不同的重做日志格式。到 InnoDB 1.2.x 版本为止，总共定义了 51 种重做日志类型。虽然各种重做日志的类型不同，但是它们有着基本的格式，表 3-2 显示了重做日志条目的结构：

表 3-2 重做日志条目结构

redo_log_type	space	page_no	redo_log_body

从表 3-2 可以看到重做日志条目是由 4 个部分组成：

- redo_log_type 占用 1 字节，表示重做日志的类型
- space 表示表空间的 ID，但采用压缩的方式，因此占用的空间可能小于 4 字节
- page_no 表示页的偏移量，同样采用压缩的方式
- redo_log_body 表示每个重做日志的数据部分，恢复时需要调用相应的函数进行解析

在第 2 章中已经提到，写入重做日志文件的操作不是直接写，而是先写入一个重做日志缓冲（redo log buffer）中，然后按照一定的条件顺序地写入日志文件。图 3-3 很好地诠释了重做日志的写入过程。

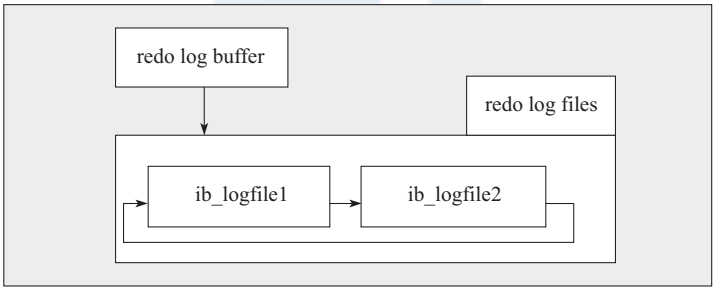

图 3-3 重做日志写入过程

从重做日志缓冲往磁盘写入时，是按 512 个字节，也就是一个扇区的大小进行写入。因为扇区是写入的最小单位，因此可以保证写入必定是成功的。因此在重做日志的写入过程中不需要有 doublewrite。

前面提到了从日志缓冲写入磁盘上的重做日志文件是按一定条件进行的，那这些条件有哪些呢？第 2 章分析了主线程（master thread），知道在主线程中每秒会将重做日志缓冲写入磁盘的重做日志文件中，不论事务是否已经提交。另一个触发写磁盘的过程是由参数 innodb_flush_log_at_trx_commit 控制，表示在提交（commit）操作时，处理重做日志的方式。

参数 innodb_flush_log_at_trx_commit 的有效值有 0、1、2。0 代表当提交事务时，并不将事务的重做日志写入磁盘上的日志文件，而是等待主线程每秒的刷新。1 和 2 不同的地方在于：1 表示在执行 commit 时将重做日志缓冲同步写到磁盘，即伴有 fsync 的调用。2 表示将重做日志异步写到磁盘，即写到文件系统的缓存中。因此不能完全保证在执行 commit 时肯定会写入重做日志文件，只是有这个动作发生。

因此为了保证事务的 ACID 中的持久性，必须将 innodb_flush_log_at_trx_commit 设置为 1，也就是每当有事务提交时，就必须确保事务都已经写入重做日志文件。那么当数据库因为意外发生宕机时，可以通过重做日志文件恢复，并保证可以恢复已经提交的事务。而将重做日志文件设置为 0 或 2，都有可能发生恢复时部分事务的丢失。不同之处在于，设置为 2 时，当 MySQL 数据库发生宕机而操作系统及服务器并没有发生宕机时，由于此时未写入磁盘的事务日志保存在文件系统缓存中，当恢复时同样能保证数据不丢失。

3.7 小结

本章介绍了与 MySQL 数据库相关的一些文件，并了解了文件可以分为 MySQL 数据库文件以及与各存储引擎相关的文件。与 MySQL 数据库有关的文件中，错误文件和二进制日志文件非常重要。当 MySQL 数据库发生任何错误时，DBA 首先就应该去查看错误文件，从文件提示的内容中找出问题的所在。当然，错误文件不仅记录了错误的内容，也记录了警告的信息，通过一些警告也有助于 DBA 对于数据库和存储引擎进行优化。

二进制日志的作用非常关键，可以用来进行 point in time 的恢复以及复制（replication）环境的搭建。因此，建议在任何时候时都启用二进制日志的记录。从 MySQL 5.1 开始，二进制日志支持 STATEMENT、ROW、MIX 三种格式，这样可以更好地保证从数据库与主数据库之间数据的一致性。当然 DBA 应该十分清楚这三种不同格式之间的差异。

本章的最后介绍了和 InnoDB 存储引擎相关的文件，包括表空间文件和重做日志文件。表空间文件是用来管理 InnoDB 存储引擎的存储，分为共享表空间和独立表空间。重做日志非常的重要，用来记录 InnoDB 存储引擎的事务日志，也因为重做日志的存在，才使得 InnoDB 存储引擎可以提供可靠的事务。