问题的引入

让我们先来观察几条非常简单的MySQL语句：

mysql> create temporary table tmp(id int, data char(20));

Query OK, 0 rows affected (0.00 sec)

mysql> create table tmp(id int, data char(20));

Query OK, 0 rows affected (0.00 sec)

mysql> drop table tmp;

Query OK, 0 rows affected (0.00 sec)

mysql> drop table tmp;

Query OK, 0 rows affected (0.00 sec)

mysql> drop table tmp;

ERROR 1051 (42S02): Unknown table 'tmp'

这是丁奇提出的引导性的问题，几条语句看似简单，不过接下来我们提出的一连串问题与进  
行的研究可都是围绕它们来的！

看到以上语句，你很容易会产生类似于以下的疑问：

1. 上述语句在一个session中先后创建了两个名为’tmp’的table，只不过一个是temporary  
table，一个是normal table。问题来了：temporary table为何可以与同名的normal table  
共存？

2. 上述语句成功执行了两条DROP TABLE语句，那么每一条语句操作的对象是哪个table呢？  
亦即同名的temporary table与normal table之间的优先级关系是如何的？

很好，有了问题就知道了前进的方向！接下来我们就从这两个问题入手，由浅入深，开始我  
们的探索之旅吧！

单机模式下的同名问题与优先级问题的探究

我们不妨从现象入手，先来验证第二个问题的结果究竟如何，即哪个表拥有较高的优先级？  
为此我们设计如下的语句：

mysql> create temporary table tmp(id1 int, data1 char(20));

Query OK, 0 rows affected (0.00 sec)

mysql> describe tmp;

+-------+----------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+----------+------+-----+---------+-------+

| id1 | int(11) | YES | | NULL | |

| data1 | char(20) | YES | | NULL | |

+-------+----------+------+-----+---------+-------+

2 rows in set (0.00 sec)

mysql> insert into tmp values(1, "Some");

Query OK, 1 row affected (0.00 sec)

mysql> select \* from tmp;

+------+-------+

| id1 | data1 |

+------+-------+

| 1 | Some |

+------+-------+

1 row in set (0.00 sec)

mysql> create table tmp(id2 int, data2 char(20));

Query OK, 0 rows affected (0.00 sec)

mysql> describe tmp;

+-------+----------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+----------+------+-----+---------+-------+

| id1 | int(11) | YES | | NULL | |

| data1 | char(20) | YES | | NULL | |

+-------+----------+------+-----+---------+-------+

2 rows in set (0.00 sec)

mysql> insert into tmp values(2, "Some");

Query OK, 1 row affected (0.00 sec)

mysql> select \* from tmp;

+------+-------+

| id1 | data1 |

+------+-------+

| 1 | Some |

| 2 | Some |

+------+-------+

2 rows in set (0.00 sec)

以上语句做的工作很简单：先创建一个名为’tmp’的temporary table，并insert一个值；  
之后创建一个名为’tmp’的normal table，也insert一个值。最终select时发现，两次  
insert操作均作用于temporary table。

至此我们可以得到初步的印象是，同名的temporary table与normal table共存时，  
temporary table较高的优先级。但是别忘了还存在另一种情况：先创建的表总有着较  
高的优先级。这个猜想是很容易来验证它的对错的，我们只需将刚才的创建表的顺序调  
换一下即可。这里就不再重复代码，直接给出结果：即使temporary table在normal table  
之后创建，诸如select，insert，update等操作仍然优先作用于temporary table之上。  
于是我们可以进一步猜测drop表的时候，先drop的也是temporary table。  
马上来验证一下：

/\* 紧接着之前的代码 \*/

mysql> drop table tmp;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from tmp;

Empty set (0.01 sec)

mysql> describe tmp;

+-------+----------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+----------+------+-----+---------+-------+

| id2 | int(11) | YES | | NULL | |

| data2 | char(20) | YES | | NULL | |

+-------+----------+------+-----+---------+-------+

2 rows in set (0.00 sec)

mysql> drop table tmp;

Query OK, 0 rows affected (0.00 sec)

mysql> show tables;

Empty set (0.00 sec)

mysql> describe tmp;

ERROR 1146 (42S02): Table 'test.tmp' doesn't exist

没问题吧！到这里我们已经从现象得出了初步的结论：在同一个session下同名的  
temporary table与normal table共存时，temporary table总是优先被操作的。那么  
我们可以更进一步提问：为什么temporary table的优先级会高于normal table呢？  
而且别忘了在本段开始时我们还提出了一个问题：为什么在同一session下同名的  
temporary table与normaltable可以共存？众所周知两个同名的temporary table或  
normal table都是不被允许的。我们可以先做出猜想：temporary table与normal table  
是存储在不同的位置的。这个猜想对吗？要回答这些问题，我们必须到MySQL的源码中  
一探究竟，找寻答案了！

(我插几句：作为一个不折不扣的MySQL菜鸟，刚拿到MySQL源码时我就像拿到了天书，  
除了膜拜之外根本不知道从何入手。经过一段时间的摸爬滚打，我高兴的发现我终于  
窥得了其中的端倪，并深感“任务驱动+gdb”是上手的好方法。MySQL完整源码可以从  
以下地址下载：<http://dev.mysql.com/downloads/>)

我们可以从创建一张表的流程入手，来探究这个过程(以下代码中，如果没有特别注明，  
其注释均为原码注释。)

对于语句

create temporary table tmp(id int, data char(20));

create table tmp(id int, data char(20));

定位到./sql/sql\_parse.cc中的mysql\_execute\_command()函数。

case SQLCOM\_CREATE\_TABLE:

{

...

if ((res= create\_table\_precheck(thd, select\_tables, create\_table)))

goto end\_with\_restore\_list;

...

/\* regular create \*/

if (create\_info.options & HA\_LEX\_CREATE\_TABLE\_LIKE)

res= mysql\_create\_like\_table(thd, create\_table, select\_tables,

&create\_info);

else

{

res= mysql\_create\_table(thd, create\_table->db,

create\_table->table\_name, &create\_info,

&alter\_info, 0, 0);

}

...

}

首先我们查看同文件中create\_table\_precheck()函数的实现：

...

/\*

For temporary tables we don't have to check if the created table exists

\*/

if (!(lex->create\_info.options & HA\_LEX\_CREATE\_TMP\_TABLE) &&

find\_table\_in\_global\_list(tables, create\_table->db,

create\_table->table\_name))

{

error= FALSE;

goto err;

}

...

而find\_table\_in\_global\_list()函数实质上调用了./sql/sql\_base.cc文件中的  
find\_table\_in\_list()函数。这个函数的功能就是去内存中的全局table list中遍历，  
确认是否已有同名的normal table存在。注意，对于temporary table，到这里为止是不做  
重名检查的。

继续跟踪到./sql/sql\_talbe.cc中的mysql\_create\_table()函数。  
开头的注释说的很清楚：

/\*

Database and name-locking aware wrapper for mysql\_create\_table\_no\_lock(),

\*/

这个函数实际上是对mysql\_create\_table\_no\_lock()的一个封装，并且处理了一些加锁  
机制。我们继续跟踪到同文件的mysql\_create\_table\_no\_lock()函数。

...

/\* Check if table exists \*/

if (create\_info->options & HA\_LEX\_CREATE\_TMP\_TABLE)

{

path\_length= build\_tmptable\_filename(thd, path, sizeof(path));

create\_info->table\_options|=HA\_CREATE\_DELAY\_KEY\_WRITE;

}

else

{

path\_length= build\_table\_filename(path, sizeof(path) - 1, db, alias, reg\_ext,

internal\_tmp\_table ? FN\_IS\_TMP : 0);

}

...

这里我们看到了一个关键函数build\_tmptable\_filename()，它位于./sql/sql\_table.cc文件  
中，这个函数是为temporary table命名的。在该函数内部我们又看到如下一段关键代码：

...

my\_snprintf(p, bufflen - (p - buff), "/%s%lx\_%lx\_%x%s",

tmp\_file\_prefix, current\_pid,

thd->thread\_id, thd->tmp\_table++, reg\_ext);

...

有了以上这段代码，temporary table的命名规则就非常清楚了，其中current\_pid为16进制  
形式，thd->thread\_id是Client的线程序号，thd->tmp\_table就是临时表序号了，而reg\_ext  
就是形如\*.frm这样的后缀。

现在我们回到函数mysql\_create\_table\_no\_lock()，紧接着刚才的代码：

/\* Check if table already exists \*/

if ((create\_info->options & HA\_LEX\_CREATE\_TMP\_TABLE) &&

find\_temporary\_table(thd, db, table\_name))

{

// 如果找到重名的表，那么执行这里的错误处理代码(非原注释)

}

...

在上面这段代码中我们又看到了一个关键函数find\_temporary\_table()，这个函数内部是大  
有文章的，它会去tmp\_table list中去遍历并检查temporary table是否已经存在。如果一切  
没有问题，那么继续往下执行:

...

if (rea\_create\_table(thd, path, db, table\_name,

create\_info, alter\_info->create\_list,

key\_count, key\_info\_buffer, file))

...

这里我们可以看到rea\_create\_table()函数的功能是创建normal table的实际数据文件。

...

if (create\_info->options & HA\_LEX\_CREATE\_TMP\_TABLE)

{

/\* Open table and put in temporary table list \*/

if (!(open\_temporary\_table(thd, path, db, table\_name, 1)))

{

(void) rm\_temporary\_table(create\_info->db\_type, path);

goto unlock\_and\_end;

}

thd->thread\_specific\_used= TRUE;

}

...

上面这段代码是对temporary table操作的，其中open\_temporary\_table()函数打开一个  
temporary table并将其加入thd->temporary\_table队列。继续往下，在函数末尾看到一  
句代码：

error= write\_create\_table\_bin\_log(thd, create\_info, internal\_tmp\_table);

进入write\_create\_table\_bin\_log()函数，上来就是一段非常清晰的注释：

/\*

Don't write statement if:

- It is an internal temporary table,

- Row-based logging is used and it we are creating a temporary table, or

- The binary log is not open.

Otherwise, the statement shall be binlogged.

\*/

已经说得很明白了，如果是内部创建的temporary table或者Row-based binlog模式下  
创建temporary table或者binlog功能未开启，那么不写binlog，其他情况下都会写。

至此，MySQL一个典型的创建表的流程就走完了。总结上述代码，我们可以回答第一个问题，  
也就是同名normal table与temporary table共存问题。现在我们知道，normal table与  
temporary table保存的位置是不同的，temporary table保存在thd->temporary\_table队列  
中，而normal table是保存在全局的队列中的，这样同名的normal table与temporary table  
就可以共存。并且，temporary table是相对于session的，因为session结束后相应的线程就  
被回收了，那么对应于该线程的temporary table也就被释放了。更进一步，从temporary  
table的命名规则我们可以看到，每个temporary table都对应着独特的客户端线程id，那么  
显然各个Client之间同名的temporary table是允许共存的。而normal table显然是在任何情  
况下都不允许同。

为了回答第二个问题，即优先级问题，我们只需要看一下drop一个表的过程即可，其他操作  
的原理也是类似的。这里我们就不再像刚才那么详细的一步步分析源码，直接给出关键代码  
(位于函数mysql\_rm\_table\_part2()中，该函数位于./sql/sql\_table.cc)

...

error= drop\_temporary\_table(thd, table); // 这里删除临时表(非原注释)

...

error= ha\_delete\_table(thd, table\_type, path, db, table->table\_name,

!dont\_log\_query); // 这里删除表的内容和索引(非原注释)

...

/\* Delete the table definition file \*/

strmov(end,reg\_ext);

// 以下删除表的定义文件(非原注释)

if (!(new\_error=my\_delete(path,MYF(MY\_WME))))

{

some\_tables\_deleted=1;

new\_error= Table\_triggers\_list::drop\_all\_triggers(thd, db,

table->table\_name);

}

...

从以上代码我们不难看出，drop表的过程总是先走temporary table，再走normal table的。  
这也就解释了为何temporary table有着比normal table更高的优先权。

好了，到目前为止我们已经从本质上回答了文章开头提出的两个问题，这样看起来问题已经  
解决的比较圆满了。但是且慢，我们以上所做的探究全部基于同一台服务器下，如果是分布  
式的系统，即主从模式下，又会出现什么样的状况呢？下面一节我们继续探究。

主从模式下temporary table机制的探究

首先我们要说明的是MySQL主从备份的实现机制。我们知道MySQL的众多日志类型中有一种为  
binlog日志类型，凡是涉及到修改数据库的操作都会被记录到binlog日志中。binlog日志本  
身又分为两种记录方式：Statement-based方式，Row-based方式(Mixed方式可以视为这两种  
方式的混合)。在主从模式下，某个特定的分布式服务器群中有两种服务器：Master(主服务  
器)与Slave(从服务器)。Master方将自己的数据修改痕迹以某种方式记录在本机的binlog文  
件中，当有Slave连接到Master时，Master会启动Binlog dump线程来将本地的binlog内容发  
送给Slave方。此时Slave方会启动两个线程：Slave I/O线程和Slave SQL线程。Slave I/O  
线程读取从Master的Binlog dump线程发送过来的binlog内容，并将其写入本机的Relay log  
中。Slave SQL线程则从本地的Relay log读取并且执行需要更新的事件。更具体的实现与配  
置细节可以参考官方文档：<http://dev.mysql.com/doc/refman/5.1/en/replication.html>

注意到Slave方执行事件的线程只有一个，那就是Slave SQL线程。想一想按照我们目前的理  
解，会出现怎样的问题？回忆刚才的MySQL temporary table命名规则，其中有一项是线程  
id。再回忆刚才我们说到，由于temporary table是相对于session的，于是不同的Client可  
以创建同名的temporary table。问题来了：将这个情景移到主从模式下，Master方同时连  
接了两个Client，每一个Client各自创建了一个名为a的temporary table。我们假设此时  
Master的binlog模式被设置为Statement-based，那么这两个建表事件都会被写入binlog。  
现在Slave I/O线程检测并读取了这两个事件，Slave SQL线程要执行这两个事件了。按照  
我们的想法，此时Slave是不能区分这两个temporary table的，因为线程id相同！

但是经过实际验证，MySQL能处理这个问题，而并没有像我们预想的那样会报错。那么MySQL  
内部是如何处理的呢？让我们再仔细读一下建表函数mysql\_create\_table\_no\_lock()中的检  
查temporary table名字冲突的函数find\_temporary\_table()的实现代码。

...

key\_length= create\_table\_def\_key(thd, key, table\_list, 1);

...

显然create\_table\_def\_key()函数是区分每个temporary table的关键，我们继续看这个函数  
内部的细节：

...

int4store(key + key\_length + 4, thd->variables.pseudo\_thread\_id);

...

这里我们看到一个关键信息：thd->variables.pseudo\_thread\_id。如果使用gdb调试，我们发  
现在find\_temporary\_table()函数中thd->variables.pseudo\_thread\_id的值等于Relay-log中  
的线程id，也就是Master的binlog中记录Client的线程id的值。然而注意到Slave SQL线程初  
始化函数handle\_slave\_sql()中调用的 init\_slave\_thread()函数中有这样一句代码：

...

thd->thread\_id= thd->variables.pseudo\_thread\_id= thread\_id++;

...

在这里，thd->variable.pseudo\_thread\_id是被初始化为Slave当前线程id的。那么它是何时被  
修改的呢？继续看代码：

...

while (!sql\_slave\_killed(thd,rli))

{

...

if (exec\_relay\_log\_event(thd,rli))

{

...

}

}

...

以上代码进入了执行relay log的循环。exec\_relay\_log\_event()中调用了函数  
apply\_event\_and\_update\_pos()，而这个函数中调用了ev->apply\_event()，最终调用了  
Query\_log\_event::do\_apply\_event()。在该函数中我们看到：

...

thd->variables.pseudo\_thread\_id= thread\_id; // for temp tables

...

就是在这里，thd->variables.pseudo\_thread\_id已经被置为我们想要看到的值了。很神奇吧！

主从模式下temporary table可能造成的不同步问题

现在我们来考虑另外一个问题，即主从模式下temporary table可能引起的主从间不同步问  
题。

回忆MySQL创建temporary table过程。该过程除了将temporary table信息加入当前线程所  
拥有的temporary table队列之外，还做了一项工作，即在/tmp目录下创建了临时数据文件，  
如：

#sql64d6\_18\_0.frm #sql64d6\_18\_0.ibd (InnoDB下)

考虑以下情形：Master机上创建了一个temporary table，并且此时binlog模式为  
Statement-based。于是Slave上读到了这个事件，并且在Slave上也同步了这个操作，即同样  
建立了一个temporary table。此时由于某种原因，Slave突然意外重启。我们知道服务器  
重启会导致所有/tmp文件夹下的数据文件被清空，那么在Slave上，原先的temporary table  
不复存在。但是此时Master上的原始的temporary table还是好好的！这样，如果我们在  
Master上做任何对该temporary table上的修改操作都会引起Slave端报错，产生类似以下信息：

Error 'Table 'test.tmp' doesn't exist' on query. Default database: 'test'.

Query: 'insert into tmp values(SomeValue)'

我们知道在Slave Server关闭后直到重启前，/tmp目录下的数据文件都是存在的。问题的本质  
在于：Slave Server关闭后，内存中的temporary table链表被回收，导致/tmp下的数据文件  
没有对应的数据结构，那么我们也就无从知晓对应的创建该表的Client到底是哪一个。

解决这个问题的基本思路就是在Slave重启时以某种方式恢复原先内存中的相关信息。其中一种  
思路是，在Slave创建temporary table时，我们额外写一个文件来记录与维护数据文件与客户  
端线程id、表名、数据库名的对应关系。另外一种思路是，在Slave创建temporary table时，  
我们将相应的binlog记录下来，然后在启动的时候重做这些记录。具体的实现这里就不再详细  
展开。