MySQL 服务器的状态

通过查看 MySQL 的状态，你能回答很多关于 MySQL 服务器的问题。MySQL 主要通过两个途径发布它的内部信息：最新的方法是标准的 INFORMATION_SCHEMA 数据库，传统的方法是一系列的 SHOW 命令（这个方法 MySQL 还继续支持，即使已经有了更好的 INFORMATION_SCHEMA 数据库）。如果你在 INFORMATION_SCHEMA 表里有查不到的信息，那就可以通过 SHOW 命令来得到。

你面临的挑战是要决定哪些信息是跟你的问题相关的，如何得到所需要的信息，以及怎么解读它们。虽然 MySQL 让你看到了大量的服务器内部信息，但是，要利用这些信息也不总是很容易。理解这些信息需要耐心、经验和随时准备查阅 MySQL 使用手册。

有一些工具可以帮助你理解在不同上下文环境下的服务器状态，例如监控和分析。我们在下一章会讲到其中的一些。然而，你仍然需要从一个高层次上来理解这些值——最低限度要知道这些值是怎么分类的——还要知道怎么从服务器上读到它们。

本章要解释许多状态命令和它们的输出结果。如果我们提及的一个主题，在本书的其他地方曾作过详述的话，我们就会将你指引到那个部分去。

13.1 系统变量

MySQL 通过 SHOW VARIABLES SQL 命令显露出许多系统变量。你可以在表达式里使用这些变量，或者在命令行里使用 mysqladmin variables。自从 MySQL 5.1 开始，你也可以通过 INFORMATION_SCHEMA 数据库里的各个表来访问这些变量了。

这些变量代表着多种配置信息，例如服务器默认的存储引擎（Storage_engine）、当前的时区、连接的字符集和启动参数。我们在第 6 章里已经解释过如何设置和使用这些系统变量。

13.2 SHOW STATUS

SHOW STATUS 命令会在一个由两列（名称/值）组成的表格里显示服务器状态变量。跟上一节里我们提到的系统变量不同，这些状态变量都是只读的。你可以像执行 SQL 命令一样执行 SHOW STATUS 来显示这些变量；或者也可以像 shell 命令一样执行 mysqladmin extended-status 来显示。如果你使用的是 SQL 命令，那你也可以用上 LIKE 和 WHERE 来限制显示结果。LIKE 会对变量名做标准的模式匹配。命令执行后会返回一个表格形式的结果，但是，你无法对它进行排序、联接到别的表，也无法完成你原本能在 MySQL 表里做的一些标准操作。

提示：我们使用术语“状态变量”用来指代从 SHOW STATUS 读来的值；术语“系统变量”指代服务器配置变量。

SHOW STATUS 的行为在 MySQL 5.0 里变化比较大，但是，除非你仔细地注意过它，否则你是不会发现这一点的。原先 MySQL 保持着全局变量的一个集合，现在只保持了一部分全部变量和每个连接的一些基本变量。因而，SHOW STATUS 包含的是一个全局变量和会话变量的混合体，它们中的很多具备了两种作用范围：既是全局变量又是会话变量，并且有相同的名称。现在，SHOW STATUS 是默认地显示会话变量，因此，如果你已经习惯于用 SHOW STATUS 查看全局变量的话，你就再也看不到这些了，你只能使用 SHOW GLOBAL STATUS 来代替。(注 1)

在 MySQL 5.1 及更新的版本里，你能直接从 INFORMATION_SCHEMA.GLOBAL_STATUS 和 INFORMATION_SCHEMA.SESSION_STATUS 表里取出值来。

在 MySQL 5.0 服务器里就有好几百个状态变量，更新的版本里还会有更多。它们的大多数要么是计数值，要么是一些状态的当前值。计数器会在每次 MySQL 执行某个操作时将值累加上去，例如初始化一个全表扫描（Select_scan）。度量值，例如服务器上打开着的连接数（Threads_connected），有时会增加有时会减少。有时，某几个变量看上去会是指向同一个东西，例如 Connections（试图联接到服务器的连接数）和 Threads_connected。在这种情况下，这几个变量都是相互关联的，但是类似的名称并不总是意味着它们之间存在着某种联系。

计数器是以无符号整数类型存储的。在 32 位系统里，它占用了 4 个字节；在 64 位系统里是 8 个字节。当达到最大值之后，它们会归零。如果你正在监控那些值的增长，有必要注意它们的归零行为。如果你的服务器已经运行了很长一段时间，你就应该知道，某些值应该要比你所期望的小一些，因为它们已经归零过了。(这种问题在 64 位系统里很少会出现。)

查看这些变量的最好办法是在时长为几分钟的时间段里看它们改变了多少。你可以使用 mysqladmin extended-status -r-i 5 或 innotop 来做这个。

以下是对我们在 SHOW STATUS 中看到的那些变量做一个大致的分类（不是完全的分类）。对于每个指定变量的细节，你要参考 MySQL 使用手册，*http://dev.mysql.com/doc/en/ mysqld-option-tables.html* 的页面内容就是关于这些变量的很有帮助的文档。在下面，当我们讨论一组具有同样前缀的相关变量时，会把它们叫做“<前缀 x>_*变量”。

13.2.1 线程和连接的统计信息

这些变量追踪着尝试连接、退出连接、网络流量和线程的统计信息：

- Connections, Max_used_connections, Threads_connected
- Aborted_clients, Aborted_connects
- Bytes_received, Bytes_sent
- Slow_launch_threads, Threads_cached, Threads_created, Threads_running

注 1：这里有一个小把戏：如果你在新服务器上使用旧版本的 mysqladmin，是无法使用 SHOW GLOBAL STATUS 的，所以就无法显示出“正确”的信息。

如果 Aborted_connects 不为 0，这就意味着你有网络问题或者有人尝试连接并且失败了（也可能是因为他们使用了错误的密码或者连到了一个错误的数据库）。如果这个值太大了，那就会有严重的副作用：它会促使 MySQL 去屏蔽主机。更多细节请查看第 12 章。

Aborted_clients 跟上面这个变量有着相似的名称，却有着完全不同的含义。如果这个值在累加，那就意味着有应用错误在发生，例如程序员在退出程序时没有正确地关闭 MySQL 连接。但是，这个变量也并非总是昭示着大问题的存在。

有一个很有用的度量值是每秒钟创建的线程数目（Threads_created/ Uptime）。如果这个值跟 0 没差太多，这意味着你的线程缓冲区太小了，新来的连接在线程缓冲区里找不到空闲的线程可用。

对于这些变量和度量值，查看服务器最近几分钟里而不是整个正常运行的时间段的值是非常有用的。

13.2.2 二进制日志的状态

Binlog_cache_use 和 Binlog_cache_disk_use 状态变量可以显示当前二进制日志缓存里保存了多少个事务，有多少个事务因为过于庞大而无法放入缓存，只能把它们的语句存储在一个临时文件里。我们已经在第 6 章里解释过如何改变二进制日志缓存的大小了。

13.2.3 命令计数器

Com_*变量记录了已发出的每一种 SQL 或者 C API 命令的数目。举例来说，Com_select 记录的是 SELECT 语句的数目，Com_change_db 记录的是使用 USE 语句或者通过 C API 改变一条连接的默认数据库的次数。Questions 变量记录了服务器收到的查询和命令的总数。然而，因为有查询缓存的命中、关闭和退出连接、及其他可能因素的存在，所以，所有 Com_*变量的总数并不完全相等。

Com_admin_commands 状态变量可能会非常大。它不仅记录了管理命令的数目，也记录了发给 MySQL 实例的 ping 请求的数目。这些请求是通过 C API 发出的，一般都是来自客户端代码，例如下面这样的 Perl 代码：

```
my $dbh = DBI->connect(...);
while ( $dbh && $dbh->ping ) {
   # Do something
}
```

这些 ping 请求都是“垃圾”请求。尽管它们不会给服务器增加多少负担，但毕竟也是浪费资源。我们在 ORM 系统里看到过：在每个查询之前都会 ping 服务器，这些都是毫无意义的；我们也在一些数据库抽象库里看到过：在每个查询前都要更改默认的数据库，这会产生数目巨大的 Com_change_db 命令。因此，最好是消除这两种做法。

13.2.4 临时文件和表

你可以查看那些记录了 MySQL 创建临时表和文件次数的变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp%';
```

13.2.5 Handler 操作

Handler API 是 MySQL 与存储引擎之间的接口。Handler_*变量记录了 Handler 的操作次数，例如 MySQL 要求存储引擎从一个索引里读取下一行数据的次数。研读你服务器里的 Handler_*变量能让你看清楚服务器做得最多的是哪几种工作。Handler_*对分析概要查询也很有用。你可以用下面这行命令查看到这些变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Handler_%';
```


13.2.6 MyISAM 索引键缓冲区

Key_*变量包含了 MyISAM 索引键缓冲区的度量值和计数器。你可以用下面这行命令来查看这些变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Key_%';
```

在第 6 章里有关于怎样分析和调优索引键缓存的详细说明。

13.2.7 文件描述符

如果你使用的主要是 MyISAM 存储引擎，那查看文件描述符的统计信息就显得尤为重要，因为它们能告诉你 MySQL 打开每一张表的.frm, .MYI 和.MYD 文件的频度分别是多少。InnoDB 把所有数据都存放在它的表空间文件里，因此，如果你主要使用的是 InnoDB，这些变量就不怎么重要了。你可以这样来查看 Open_*变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Open_%';
```

第 6 章里说明了怎样对影响这些变量的设置项进行调优。

13.2.8 查询缓存

你可以通过 Qcache_*状态变量来检查查询缓存。如果出于性能起见，你的查询都依赖于查询缓存，那么，这一组变量会是相当的重要。要检查它们，就使用：

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache_%';
```

关于怎样调优查询缓存的详细说明在本书的第 5 章。

13.2.9 各种类型的 SELECT

Select_*变量记录了各类型 SELECT 查询的次数。它们可以帮你查看到各种类型查询计划的比例关系。不幸的是，有一些类型的查询没有相应的状态变量，例如 UPDATE 和 REPLACE，然而，你可以通过查看 Handler_*状态变量（刚刚讨论过的）来获知那些非 SELECT 查询的性能情况。要查看所有 Select_*变量，可以这么来做：

```
mysql> SHOW GLOBAL STATUS LIKE 'Select_%';
```

据我们判断，Select_*状态变量可以按照开销的降序来排名：

Select_range

一种联接的数目，该联接在第一个表的索引的指定范围内做扫描。

`Select_scan`

对第一个表作全表扫描的联接数目。如果第一个表里的每一行都被加入了联接的话，它就不会出错。唯一不好的事情是如果你不想加入所有的行，那就没有索引能让你找到那些想要的记录。

`Select_full_range_join`

一种联接的数目，它使用表 n 里的一个值去获取表 n+1 里某个引用索引范围内的行。根据不同的查询，它的开销有时会比 Select_scan 大，有时会小。

`Select_range_check`

一种联接的数目，它会为表 *n* 里的每一行，在表 *n*+1 里对索引重新估值，看看哪个开销最廉价。这通常意味着表 *n*+1 没有索引可用于这个联接。这种查询计划的开销非常高。

`Select_full_join`

显示一个交叉联接，或者一个在表里没有任何标准能匹配到的行的联接。检查过的行的总数就是每个表里行数的乘积。这经常是个很糟糕的事情。

在一个调优过的服务器里，最后两个变量的增长应该不怎么快。比较这两个计数器的值分别与正在执行的查询总数（Com-select）的比值，你有时会发现一个没优化好的工作负荷正在运行。如果其中有一个比值比总数高出几个百分点，这就说明你可能需要优化一下你的查询和/或数据配置了。

还有一个相关的状态变量叫 `Slow_queries`，我们为缓慢查询日志开发的补丁可以帮助你查看某个查询是否需要全联接，是否使用了查询缓存等等。更多的内容请查看第 65 页的“更精细地控制日志”。

13.2.10 排序
Sorts

在第 3 章和第 4 章里，我们讲到了很多关于 MySQL 排序优化的内容，因此，对于排序的工作原理，你应该已经有了一个较好的认识。当 MySQL 无法使用索引去获得预排序的行时，它就不得不做文件排序，从而，会增加 `Sort_*`状态变量的值。只能通过增加可用于排序的索引来影响这些值。`Sort_merge_passes` 依赖于服务器变量 `sort_buffer_size`（不要跟服务器变量 `myisam_sort_buffer_size` 搞混了。）MySQL 使用排序缓冲区存储一批数据行做排序。在完成排序之后，它会把这些已排序的行归并到结果里，并增加 `Sort_merge_passes` 的值，然后又把新的一批数据行填充到缓冲区里进行排序。如果排序缓冲区太小，这个过程会被重复很多次，于是这个状态变量的值将很大。

你可以这样来查看所有 `Sort_*`变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Sort_%';
```

当 MySQL 从文件排序的结果里读出排序过的数据行，并返回给客户端时，它会增加 `Sort_scan` 和 `Sort_range` 两个变量的值。这两个变量的主要区别在于：前者是当查询方案触发了 `Select_scan` 增长时，它也会增长；后者是当 `Select_range` 增长时，它也随之增长。两者之间没有实现和开销方面的差异，主要是为了表明各自对应的查询方式触发排序的次数。

13.2.11 表锁定
Table Locking

Table_locks_immediate 和 Table_locks_waited 变量会告诉你这个时刻有多少个锁立即被授予，又有多少个需要等待。如果你在 SHOW FULL PROCESSLIST 看到有很多线程处于锁定状态，那你就要检查一下这些值。然而，要知道显示出来的仅仅是服务器级锁的统计信息，没包含存储引擎的。关于如何调试锁，请查看附录 D。

13.2.12 Secure Sockets Layer (SSL)

Ssl_*变量显示了服务器是如何配置 SSL 的（假设是正确地配置）。你可以这样查看所有 SSL 变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Ssl_%';
```

13.2.13 InnoDB 特有的变量
InnoDB-Specific

Innodb_*变量显示的是包含在 SHOW INNODB STATUS（这个在本章的后半部分里会讲到）里的一些数据。这些变量可以通过名称来分组：Innodb_buffer_pool_*、Innodb_log_*等。当我们检查 SHOW INNODB STATUS 时，会更多地讨论其内部的那些变量。

这些变量在 MySQL 5.0 及更新的版本里可以使用，它们有一个很重要的副作用：它们会创建一个全局锁，在释放锁之前，还会遍历整个 InnoDB 的缓存池，在此期间，其他线程都会阻塞，直到它把锁释放了。因此，它影响到了其他几个状态值，例如 Threads_running，它们会显示出高于平常的数值（可能还会高很多，具体要看你的服务器有多繁忙）。同样的影响也会在你运行 SHOW INNODB STATUS 或者通过 INFORMATION_SCHEMA 表(在 MySQL 5.0 及更新的版本里，SHOW STATUS 和 SHOW VARIABLES 在后台都被映射到 INFORMATION_ SCHEMA 表的查询上)访问这些统计信息时发生。

因此，上面这些操作在这些版本的 MySQL 里会比较昂贵——过于频繁地查看服务器的状态（例如每秒钟一次）引发了数量可观的系统开销。使用 SHOW STATUS LIKE 也没用，因为它也是获取了全部的状态后再做过滤的。

564

13.2.14 Plug-in 特有的变量
Plug-in-Specific

MySQL 5.1 和更新的版本支持可插入的存储引擎，并为存储引擎提供了一个机制，用来注册它们自己的状态和配置变量。如果你正在使用一个插入式存储引擎，就会看到一个跟插入式有关的变量。

13.2.15 其他
Miscellaneous

我们将其他一些状态变量罗列在下面：

Delayed_*, Not_flushed_delayed_rows

这些变量是 INSERT DELAYED 查询的计数器和度量值。

`Last_query_cost`

这个变量显示了查询优化器的查询计划在最近一次执行查询时的开销。

我们在第 4 章里已经讨论过查询计划开销了。

`Ndb_*`

这个变量显示了 NDB Cluster 的配置信息（如果是正确的配置的话）。

`Slave_*`

当服务器是一个复制从服务器时，这个变量就启用了。`Slave_open_temp_tables` 变量对于基于语句的复制非常重要。更多关于复制和临时表的内容请查看第 394 页的“丢失的临时表”。

`Tc_log_*`

这个计数值用于记录该服务器被用作 XA 事务协调器的次数。更多细节请查看第 262 页的“分布式（XA）事务”。

`Uptime`

这个变量显示了服务器的正常运行时间，以秒为单位。

要想了解系统总体工作负荷的最好途径是比较一组相关的状态变量。例如，查看所有 `Select_*`变量，或者 `Handler_*`变量。如果你正在使用 innotop，采用 Command Summary 模式将会更方便做到这一点，但是，你也可以通过像 mysqladmin extended -r -i60 | grep Handler_这样的命令来手动做这个事情。以下就是我们在某台服务器上使用 innotop 检查 `Select_*`变量的结果显示：

```
_____________________ Command Summary ____________________ _ _
Name                    Value    Pct     Last Incr  Pct
Select_scan            756582   59.89%          2  100.00%
Select_range           497675   39.40%          0    0.00%
Select_full_join         7847    0.62%          0    0.00%
Select_full_range_join   1159    0.09%          0    0.00%
Select_range_check          1    0.00%          0    0.00%
```

最前面两列的值是自服务器启动以来的累计数值；后面两列是自最近一次刷新以来的累计数值（在这个例子里，是 10 秒钟之前）。在显示的数据里，百分比超过了显示的数值总和，但是不会超过所有查询的总和。

即使是这台服务器有着百分比相对较低的全联接，它也值得仔细查看一番，查究为什么还会有一些全联接存在。

13.3 SHOW INNODB STATUS

在 `SHOW ENGINE INNODB STATUS`（或者一个相近的命令：`SHOW INNODB STATUS`）的输出里，InnoDB 存储引擎显示出了大量的内部信息。跟大多数 `SHOW` 命令不同的是，它的输出就是单独的字符串，没有行和列。它分为很多个节，每一节对应了 InnoDB 存储引擎不同部分的信息，其中有一些信息对于 InnoDB 开发者来说是非常有用的，但是，许多信息如果你试着去理解，并且应用到高性能 InnoDB 调优的时候，你会发现它们都非常有趣 ——甚至是非常核心的。

提示：InnoDB 经常把 64 位数字分成两部分来打印：高 32 位和低 32 位。有一个例子就是事务 ID，比如 `TRANSACTION 0 3793469`，你可以这么来计算 64 位数字的值：把第一部分往左移动 32 位，然后把它加到第二部分上。我们在后面会展示几个例子。

输出内容包含了一些平均值的统计信息，例如 `fsync()`在每一秒里的调用次数。这些显示的平均值是自上次输出结果生成以来的统计数，因此，如果你正在检查这些值，那就要确保你已经等待了 30 秒左右的时间，使两次采样之间积累起足够多的统计时间。不是所有的输出都会在一个时间点上就能生成，所以，也不是所有显示出

来的平均值会在同一时间间隔里被重新计算一遍。而且，InnoDB 有一个内部复位间隔，它是不可预知的，在各个版本里也是不一样的。您应该检查一下输出，看看有哪些平均值在这个时间段里生成，因为两次采样的时间间隔不必总是相同的。

这里面有足够的信息让你手工计算出大多数你想要的统计信息。但是，如果这时有一款监控工具，例如 innotop——它能为你计算出增量差别和平均值——那将是非常有用的。

13.3.1 头部信息

输出的第 1 段是头部信息，它仅仅是代表输出的开始，其内容包括当前的日期和时间，以及自上次输出以来经过的时长。第 2 行是当前日期和时间；第 4 行显示的是平均值计算的时间间隔，它或者是上次输出以来的时间，或者是距离上次内部复位的时长：

```
1  =====================================
2  070913 10:31:48 INNODB MONITOR OUTPUT
3  =====================================
4  Per second averages calculated from the last 49 seconds
```


13.3.2 SEMAPHORE

如果你有高并发的工作负荷，你就要关注这一段信号量。它包含了两种数据：时间计数器，以及可选的当前等待线程的列表。如果有性能上的瓶颈，你就可以用这些信息来帮你找出瓶颈到底在哪里。不幸的是，要想知道怎么使用这些信息还是有一点点复杂，但是，我们会在本章的后半部里给你一些建议。这里是一些输出样例：

```
 1 ----------
 2 SEMAPHORES
 3 ----------
 4 OS WAIT ARRAY INFO: reservation count 13569, signal count 11421
 5 --Thread 1152170336 has waited at ./../include/buf0buf.ic line 630 for 0.00 seconds
   the semaphore:
 6 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
 7 waiters flag 0
 8 wait is ending
 9 --Thread 1147709792 has waited at ./../include/buf0buf.ic line 630 for 0.00 seconds
   the semaphore:
10 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
11 waiters flag 0
12 wait is ending
13 Mutex spin waits 5672442, rounds 3899888, OS waits 4719
14  RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163
```

第 4 行给出了关于操作系统等待阵列的信息，它是一个“插槽”式的阵列，InnoDB 保留了阵列里的一些插槽给信号量使用，操作系统用这些信号量给线程发送信号，使线程可以继续运行，完成它们等着要做的事情。这一行还显示出 InnoDB 使用了多少次操作系统的等待。reservation count 显示了 InnoDB 分配插槽的频度，而 signal count 衡量的是线程通过阵列得到信号的频度。操作系统的等待相对于循环等待（Spin wait）要更昂贵一些，我们即将看到这一点。

第 5 到 12 行显示的是当前正在等待互斥量的 InnoDB 线程。在这个例子里显示出有两个线程正在等待，每一个都是以“-- Thread <num> has waited…”开始的。这一段应该是空的，除非你的服务器运行着高并发的工作负载，

它促使 InnoDB 采取让操作系统等待的措施。如果你对 InnoDB 源代码很熟悉的话，你在这里会看到非常有用的信息：发生线程等待的代码文件名。这就给了你一个提示：在 InnoDB 内部哪里才是热点。举例来说，你看到许多线程都在一个名为 buf0buf.ic 的文件上等待着，这意味着你的系统里存在着缓冲池竞争。这个输入信息还显示了这些线程等待了多长的时间，另外，“waiters flag”显示了有多少个等待者正在等待同一个互斥量，文字“wait is ending”意味着这个互斥量实际上已经被释放了，但是，操作系统还没把线程调度过来运行。

你可能弄不明白 InnoDB 真正等待的是什么。InnoDB 使用了互斥量和信号量来保护代码的临界区，例如限定每次只能有一个线程进入临界区，或者是当有活动的读者时，就限制写者的加入等等。在 InnoDB 代码里有很多临界区，在正确的条件下，它们都会显示在那里。你常常能见到的一个就是获取缓冲池分页的访问权。 567

在等待线程的列表之后，第 13、14 行显示了更多的事件计数器。第 13 行显示的是跟互斥量相关的几个计数器，第 14 行用于显示读/写共享和排斥锁的计数器。在每一个案例里，你都能看到 InnoDB 诉诸于操作系统等待的频度。

InnoDB 有着一个多阶段等待策略。首先，它会试着对锁进行循环等待。如果经过了一个预设的循环等待周期（设置 innodb_sync_spin_loops 配置变量）之后还没有成功，那就会退到更昂贵更复杂的等待阵列里（注 2）。

循环等待的成本相对比较低，但是它们要不停地检查一个资源是否被锁定，还是消耗了 CPU 周期。但是，这没有像它听起来那么糟糕，因为当处理器在等待 I/O 时，一般都有一些空闲的 CPU 周期可用，即使是没有空闲的 CPU 周期，空等也要比其他方式更加廉价一些。然而，当另外一条线程能做一些事情时，循环等待也会独占处理器。

循环等待的替换方案就是让操作系统做上下文切换，这样，当这条线程在等待时，另外一条线程就可以被运行，然后，通过等待阵列里的信号量发出信号，唤醒那条沉睡的线程。通过信号量来发送信号是比较有效率的，但是，上下文切换就很昂贵。你很快就能累加出来：每秒钟几千次的切换会引发大量的系统开销。

你可以通过改变系统变量 innodb_sync_spin_loops 的值，试着在循环等待与操作系统等待之间作一个平衡。不要担心循环等待，除非你在每一秒里看到了许多循环等待（大概是成百上千个）。在第 6 章里，有更多关于如何调优这些变量的建议。

13.3.3 LATEST FOREIGN KEY ERROR

在下一段里，LATEST FOREIGN KEY ERROR 一般不会出现，除非你的服务器上有一个外键错误。在源代码里有许多地方会生成这样的输出，具体还跟错误的类型有关系，有时是一个事务在插入、更新或删除一条记录时要寻找到父行或子行；或者是在改变一个定义了外键的表结构时。

这一段的输出对于调试 InnoDB 常常发生的原因模糊的外键错误非常有帮助。让我们来看几个实例。首先，我们创建两个表，并在两者之间建立一个外键，并插入一些数据： 568

```
CREATE TABLE parent (
   parent_id int NOT NULL,
   PRIMARY KEY(parent_id)
) ENGINE=InnoDB;

CREATE TABLE child (
   parent_id int NOT NULL,
```

注 2：等待阵列在 MySQL 5.1 里变得更加高效了。

```
  KEY parent_id (parent_id),
  CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
) ENGINE=InnoDB;

INSERT INTO parent(parent_id) VALUES(1);
INSERT INTO child(parent_id) VALUES(1);
```


这里有两类基本的外键错误。因为增加、更新或删除数据而违反外键约束的是第一类错误。举例来说，以下就是当我们从父表里删除一行数据时发生的错误：

```
DELETE FROM parent;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
fails (`test/child`, CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES
`parent` (`parent_id`))
```

这个错误信息非常直观易懂。当你增加、更新和删除没有匹配的行时，都会看到跟它相似的错误信息。下面是通过 SHOW INNODB STATUS 看到的结果：

```
1  ------------------------
2  LATEST FOREIGN KEY ERROR
3  ------------------------
4  070913 10:57:34 Transaction:
5  TRANSACTION 0 3793469, ACTIVE 0 sec, process no 5488, OS thread id 1141152064 updating
   or deleting, thread declared inside InnoDB 499
6  mysql tables in use 1, locked 1
7  4 lock struct(s), heap size 1216, undo log entries 1
8  MySQL thread id 9, query id 305 localhost baron updating
9  DELETE FROM parent
10 Foreign key constraint fails for table `test/child`:
11 ,
12  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent`
   (`parent_id`)
13 Trying to delete or update in parent table, in index `PRIMARY` tuple:
14 DATA TUPLE: 3 fields;
15  0: len 4; hex 80000001; asc     ;; 1: len 6; hex 00000039e23d; asc    9 =;; 2: len 7;
   hex 000000002d0e24; asc     - $;;
16
17 But in child table `test/child`, in index `parent_id`, there is a record:
18 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
19  0: len 4; hex 80000001; asc    ;; 1: len 6; hex 000000000500; asc     ;;
```

第 4 行显示了最近的那个外键错误发生的日期和时间。第 5 到第 9 行显示的是那个引发外键错误的事务的细节，我们会在后面解释这一行。第 10 到 19 行显示的是当错误发生时，InnoDB 正在更改的数据。这些输入的大部分是转换为可打印格式的数据行，对此，我们也会在后面做一些说明。

直到现在，一切都还好理解。但是，另外一种外键错误就非常难以调试了。以下就是当我们试着更改父表结构时发生的错误：

```
ALTER TABLE parent MODIFY parent_id INT UNSIGNED NOT NULL;
ERROR 1025 (HY000): Error on rename of './test/#sql-1570_9' to './test/parent'
(errno: 150)
```

这看上去不够清晰，但是，用 SHOW INNODB STATUS 来查看会更明显一些：

```
1  ------------------------
2  LATEST FOREIGN KEY ERROR
3  ------------------------
4  070913 11:06:03 Error in foreign key constraint of table test/child:
5  there is no index in referenced table which would contain
```

```
 6 the columns as the first columns, or the data types in the
 7 referenced table do not match to the ones in table. Constraint:
 8 ,
 9   CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
10 The index in the foreign key in table is parent_id
11 See http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html
12 for correct foreign key definition.
```

这里的错误原因源自不同的数据类型。作为外键的列必须有同样的数据类型，包括任何修饰符的更改（例如在这里是 UNSIGNED 引发的问题）。无论什么时候当你看到 1 025 号错误，不明白它为什么出现时，最好的办法就是查看 SHOW INNODB STATUS 的输出。

13.3.4 LATEST DETECTED DEADLOCK

跟上面的外键节一样，LATEST DETECTED DEADLOCK 也只有当服务器内有死锁时才会出现。

死锁在等待关系图（Waits-for Graph）里就是一个循环，就是一个锁定了行的数据结构又在等待别的锁。这个循环可以任意地大。InnoDB 会立即检测到死锁，因为每当有事务等待行锁的时候，它都会去检查等待关系图里是否有循环。死锁的情况可能会比较复杂，但是，这一段里只显示了最近两个死锁的情况，它们在各自事务里执行的最后一条语句，以及它们在图里形成循环的锁的信息。你看不到在这个循环里的其他事务，也看不到在事务里早先真正获得了锁的语句。尽管如此，你通常还是可以通过查看输入结果来确定到底是什么引起了死锁。

在 InnoDB 里实际上有两种死锁。第一种就是人们常常碰到的那种，它在等待关系图里是一个真正的循环。另外一种就是在一个等待关系图里，因代价昂贵而无法检查它是不是包含了循环。如果 InnoDB 要在图里检查超过 100 万个锁，或者在检查过程中，InnoDB 要重做 200 个以上的事务，那它就会放弃，并宣布这里有一个死锁。这些数值都是硬编码在 InnoDB 代码里的常量，你无法配置它们（如果你愿意的话，可以在代码里更改这些数值，然后重新编译）。当 InnoDB 的检查工作超过这个极限后，它就会引发一个死锁，这时你就在输出里看到一条信息“TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH”。

InnoDB 不仅会打印出事务和事务持有的锁和等待的锁，而且还有记录本身。这些信息对于 InnoDB 开发者来说都是很有用的，但是目前也没有办法关闭它。不幸的是，它会变得很大，超过你为输出结果预留的长度，以至于你都没法看到下面几段输出信息了。对此唯一的补救办法是，制造一个小死锁来替换那个大的死锁，或者使用由本书一个作者提供的补丁，它在 *http://lists.mysql.com/internals/35174* 可以得到。

这里有一个死锁信息的样例：

```
 1 ------------------------
 2 LATEST DETECTED DEADLOCK
 3 ------------------------
 4 070913 11:14:21
 5 *** (1) TRANSACTION:
 6 TRANSACTION 0 3793488, ACTIVE 2 sec, process no 5488, OS thread id 1141287232 starting
   index read
 7 mysql tables in use 1, locked 1
 8 LOCK WAIT 4 lock struct(s), heap size 1216
 9 MySQL thread id 11, query id 350 localhost baron Updating
10 UPDATE test.tiny_dl SET a = 0 WHERE a <> 0
11 *** (1) WAITING FOR THIS LOCK TO BE GRANTED:
12 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
   'test/tiny_dl' trx id 0 3793488 lock_mode X waiting
13 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
14  0: len 6; hex 000000000501 ...[ omitted ] ...
```

```
15
16 *** (2) TRANSACTION:
17 TRANSACTION 0 3793489, ACTIVE 2 sec, process no 5488, OS thread id 1141422400 starting
   index read, thread declared inside InnoDB 500
18 mysql tables in use 1, locked 1
19 4 lock struct(s), heap size 1216
20 MySQL thread id 12, query id 351 localhost baron Updating
21 UPDATE test.tiny_dl SET a = 1 WHERE a <> 1
22 *** (2) HOLDS THE LOCK(S):
23 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
   'test/tiny_d1' trx id 0 3793489 lock mode S
24 Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
25  0: ... [ omitted ] ...
26
27 *** (2) WAITING FOR THIS LOCK TO BE GRANTED:
28 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
   'test/tiny_dl' trx id 0 3793489 lock_mode X waiting
29 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
30  0: len 6; hex 000000000501 ...[ omitted ] ...
31
32 *** WE ROLL BACK TRANSACTION (2)
```


第 4 行显示的是死锁发生的时间，第 5 到 10 行信息的是死锁里的第一个事务的信息。在下一节里，我们会详尽地解释这些输出的含义。

第 11 到 15 行显示的是当死锁发生时，事务 1 正在等待的锁。我们忽略了其中第 14 行的信息，那是因为这只对调试才有用。这里要特别注意的内容是第 12 行，它告诉你这个事务正在等待的是 `test.tiny_dl` 表里的 `GEN_CLUST_INDEX`（注 3）的排它锁。

第 16 到 21 行显示的是第二个事务的状态，第 22 到 26 行显示的是该事务持有的锁。为了简洁起见，第 25 行的几条记录已经被我们删去了。这些记录里的一条就是第一个事务正在等待的是哪一条记录。最后，第 27 到 31 行显示它正在等待的是哪一个锁。

当一个事务持有了其他事务需要的锁，同时，它又想获取其他事务持有的锁，这时，等待关系图上的循环就产生了。InnoDB 不会显示所有持有的锁和等待的锁，但是，它显示了足够的信息帮你决定：查询操作正在使用的是哪些索引？这对于你是否能避免死锁有着极大的价值。

如果你能使两个查询在同一个索引、同一个方向上进行扫描，你就能降低死锁的数目，因为，查询在同一个顺序上请求锁的时候不会创建循环。有时候，这是很容易做到的，举例来说，如果你要在一个事务里更新许多条记录，就可以在应用程序的内存里把它们按主键进行排序，然后，再用同样的顺序更新到数据库——这样就不会有死锁的发生。但是，在其他一些时候，这个方法也是行不通的（例如当你有两个进程使用了不同的索引区操作同一张表的时候）。

第 32 行显示的是哪个事务被选中成为死锁的牺牲品。InnoDB 会把看上去最容易回滚（就是更新的记录数最少的）的事务选为牺牲品。

这些信息对于监控和日志分析是很有价值的。使用 Maatkit 的 mk-dead- lock-logger 工具来做这个非常方便，对于普通的日志，它也能派得上用场：找出所有与死锁相关的线程在执行的查询语句，看看到底是哪条语句引起了死锁。在下一节里，你会看到如何在死锁的输出信息里找到线程 ID。

注 3：这个索引是当你没有指定主键时，InnoDB 在内部自动创建的。

13.3.5 TRANSACTION

这一段里包含的是 InnoDB 事务的一些摘要信息，它们跟随在当前活动事务目录之后。以下就是前几行（头部信息）：

```
1   ------------
2   TRANSACTIONS
3   ------------
4   Trx id counter 0 80157601
5   Purge done for trx's n:o <0 80154573 undo n:o <0 0
6   History list length 6
7   Total number of lock structs in row lock hash table 0
```

具体的输出信息会根据 MySQL 版本的不同而有变化，但是至少包含了如下这些信息：

- 第 4 行：当前事务的 ID，它是一个系统变量，每创建一个新事务就会累加。
- 第 5 行：这是 InnoDB 清除旧版本 MVCC 行时所用的事务 ID。通过这个值和当前事务 ID 的比较，你会看到有多少老版本的数据已经被清除掉了。在这里，读取这个数字需要准备多大的取值范围才够安全没有硬性规定。如果数据没做过任何更新，一个巨大的数字也不意味着这是未清除的数据，因为实际上所有事务在数据库里查看的都是同一个版本的数据。从另一方面来讲，如果同时有很多行被更新，那每一行就会有一个或多个版本会被留在内存里。减少此类开销的最好办法是确认当事务完成时就立即将它提交，不要让它长时间地处于打开的状态。因为即使只有一个打开的事务不做任何操作，它也会影响到 InnoDB 清除旧版本的行数据。

  同样是在第 5 行里，还有一项 InnoDB 清除进程正在使用的撤销日志编号——如果有的话。如在本例当中一样，如果它是“0 0”，说明清除进程处于空闲状态。

- 第 6 行：历史记录的长度，它就是位于 InnoDB 数据文件的撤销空间里的未清除事务的数目。当一个事务执行了更新并提交后，这个数字就会累加；当清除进程移除一个旧版本数据时，它就会递减。在第 5 行里，清除进程也会更新该数值。
- 第 7 行：锁结构的数目。每一个锁结构经常持有许多个行锁，所以，它跟被锁定行的数目不一样。

头部信息之后就是一个事务列表。当前版本的 MySQL 还不支持嵌套式事务，因此，在某个时间点上，每个客户端连接能拥有的事务数目是有一个上限的，而且每一个事务只能属于单独一个连接。在输出信息里，每一个事务至少占有两行内容。下面这个例子就是你能看到的一个事务所显示的最少的信息：

```
1   ---TRANSACTION 0 3793494, not started, process no 5488, OS thread id 1141152064
2   MySQL thread id 15, query id 479 localhost baron
```

第 1 行是以事务 ID 和状态开始的。这个事务正处于“not started”状态，这意味着它已经被提交，但是没有发出会影响到其他事务的执行语句，它可能就是空闲着。接下来就是一些进程和线程的信息。第 2 行显示的是 MySQL 的进程 ID，它跟 `SHOW FULL PROCESSLIST` 里显示的 `Id` 列是一致的。紧跟它之后的是一个内部查询号码和一些连接信息（这也跟你在 `SHOW FULL PROCESSLIST` 里发现的一样）。

其实，每一个事务都能打印出比上面更多的信息，这里就是一个更复杂的例子：

```
1   ---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id 1148250464, thread
    declared inside InnoDB 442
2   mysql tables in use 1, locked 0
```

```
3  MySQL thread id 8079, query id 728899 localhost baron Sending data
4  select sql_calc_found_rows * from b limit 5
5  Trx read view will not see trx with id>= 0 80157601, sees <0 80157597
```

本例中的第 1 行显示出这个事务活动了 4 秒钟。它可能的状态是“not started”、“active”、“prepared”和“committed in memory”（一旦被提交到磁盘上，状态就会变为“not started”）。虽然在这个示例里没显示出来，但是，在其他条件下，你也能看到关于事务当前正在做什么的信息，有超过 30 个字符串常量可以作为这类信息显示在这里，例如“fetching rows”、“adding foreign keys”等等。

第 1 行里的“thread declared inside InnoDB 442”的意思是该线程正在 InnoDB 内核里做一些操作，并且，还有 442 张“票子”可以使用。换句话说就是同样的 SQL 查询可以重入 InnoDB 内核 442 次。这“票子”是系统用来限制内核中线程并发操作的手段，防止它在某些平台上运行失常。即使线程的状态是“inside InnoDB”，那线程也可能不必把所有的工作都放在 InnoDB 里面来做。查询大概也就是在服务器级上做一些操作，然后只要通过某个途径跟 InnoDB 内核互动一下就可以了。你会看到这些事务的状态是“sleeping before joining InnoDB queue”或者“waiting in InnoDB queue”。

接下来你看到的这一行显示了当前语句里使用到的和锁定的表有多少。InnoDB 不是以通常方式锁住表，有些语句还是可以在上面执行。当 MySQL 服务器在高于 InnoDB 层次之上将表锁定时，这里也是能够显示出来的。如果事务已经锁定了任意几行数据，这里将会有一行信息显示出锁定结构的数目（再声明一次，这跟行锁是两回事）和堆的大小。具体例子你可以查看上文中死锁的输出信息。在 MySQL 5.1 及更新的版本里，这一行也是显示当前事务持有的行锁的实际数目。

堆的大小指的是为了持有这些行锁而占用的内存大小。InnoDB 是用一种特殊的位图表来实现行锁的，从理论上讲，它将每一个锁定的行表示为一个比特。我们的测试显示，每一个锁通常不超过 4 比特。

574

本例中的第 3 行包含的信息略微多于上一个样例中的第 2 行：在该行的末尾是线程状态“Sending data”，这跟你在 `SHOW FULL PROCESSLIST` 的 `Command` 列上看到的一样。

如果事务正在运行的一个查询，在本例的第 4 行里就会显示出查询的文本（或者，在某些版本的 MySQL 里，显示的是查询的摘要信息）。

第 5 行显示了事务的读视图，它表明了事务 ID 的范围，包括了因为版本关系而产生的对于事务可见的和不可见两种类型。在本例中，四个事务在两个数字之间有一个缺口，这四个事务可以是可见的，也可以是不可见的。当 InnoDB 执行一个查询时，对于那些事务 ID 正好落入这个缺口的行来说，InnoDB 还会检查它们的可见性。

如果事务正在等待一个锁，你就会在查询内容之后看到这个锁的信息。在上文的死锁例子里，这样的信息已经看到过多次了。不幸的是，输出信息并没有说出这个锁正被其他哪个事务**持有着**。

如果输出信息里有很多个事务，InnoDB 就限制了要打印出来的事务数目，以免输出信息变得很长。这时，你就会看到“...truncated...”字样。

13.3.6 FILE I/O

`FILE I/O` 段显示的是 I/O helper 线程的状态，用性能计数器的方式来表示：

```
1  --------
2  FILE I/O
3  --------
4  I/O thread 0 state: waiting for i/o request (insert buffer thread)
```

```
 5 I/O thread 1 state: waiting for i/o request (log thread)
 6 I/O thread 2 state: waiting for i/o request (read thread)
 7 I/O thread 3 state: waiting for i/o request (write thread)
 8 Pending normal aio reads: 0, aio writes: 0,
 9  ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
10 Pending flushes (fsync) log: 0; buffer pool: 0
11 17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs
12 0.20 reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s
```


第 4 到 7 行显示的就是 I/O helper 线程当前状态。第 8 到 10 行显示的是每个 helper 线程还没完成的操作的数目，以及日志和缓冲池线程还没完成的 `fsync()` 操作的数目。其中的缩写词“aio”的意思是“asynchronous I/O”。第 11 行显示了读、写及 `fsync()` 调用的执行次数。这些变量适合用一个图形化的有趋势显示功能的系统来监控，我们会在下一章里谈到这样的系统。变量的绝对值会因工作负载的不同而有变化，因此，监控它们的变化情况显得尤为重要。第 12 行显示的是在头部信息指明的那段时间里，平均每一秒钟执行各种操作的次数。

第 8 到 9 行里显示的未决数值是检测 I/O 密集应用的好办法。如果这些类型的 IO 里大多数都有一些未决的操作，这说明当前的工作负载可能就是 I/O 密集的。

在 Windows 上，你可以通过 `innodb_file_io_threads` 配置变量来调整 I/O helper 线程的数目，这样，你就可以看到超过一条的读和写线程了。但是，在所有平台上，你至少会看到以下四条线程：

插入缓冲区线程

负责插入缓冲区的合并（例如将记录从插入缓冲区合并到表空间里）。

日志线程

负责异步的日志刷新。

读线程

负责读前置（Read-ahead）的操作，预测 InnoDB 将要使用的数据，并将它们预读取进来。

写线程

刷新“脏”缓冲区。

13.3.7 INSERT BUFFER AND ADAPTIVE HASH INDEX

这一节显示的是 `INSERT BUFFER AND ADAPTIVE HASH INDEX` 的状态：

```
1  -------------------------------------
2  INSERT BUFFER AND ADAPTIVE HASH INDEX
3  -------------------------------------
4  Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty
5  Ibuf for space 0: size 1, free list len 887, seg size 889,
6  2431891 inserts, 2672643 merged recs, 1059730 merges
7  Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)
8  2208.17 hash searches/s, 175.05 non-hash searches/s
```

第 4 行显示的信息是插入缓冲区的大小、它的“自由列表”的长度、以及它的分段大小。其中的文本“for space 0”像是说明了多缓冲区（每个表空间一个）的可能性——但是这从没被实现过，而且这行文本在最近的 MySQL 版本里已经被移除了。插入缓冲区只可能有一个，因此，第 5 行是真正的冗余。第 6 行显示了 InnoDB 已经做了多少次缓冲区操作。查看其中合并到插入缓冲区的比例是评判缓冲区效率的好办法。

第7行显示的是自适应散列索引(Adaptive Hash Index)的状态。第8行显示了根据头部信息划定的期限内InnoDB做过的散列索引的次数。散列索引查找次数和非散列索引查找次数的比例是另一项效率度量值，因为散列查找快过非散列查找。这些都是咨询式的信息，你无法配置自适应散列索引。

576

13.3.8 LOG

这一段里显示的InnoDB事务LOG子系统的统计信息：

```
1  ---
2  LOG
3  ---
4  Log sequence number 84 3000620880
5  Log flushed up to   84 3000611265
6  Last checkpoint at  84 2939889199
7  0 pending log writes, 0 pending chkp writes
8  14073669 log i/o's done, 10.90 log i/o's/second
```

第4行显示了当前日志的顺序号，第5行显示了日志已经刷新到的点位。日志顺序号就是已经写入日志文件的字节总数，因此，你能用它来计算日志缓冲区里还有多少数据没被刷新到日志文件里。在本例中，是9 615个字节（13000620880 - 13000611265）。第6行显示了最近的一个检查点（一个检查点代表了一个瞬间，在那一刻里数据和日志文件都处于可知的状态，它可被用来做还原）。第7到8行显示了还未完成的日志操作和统计信息，拿这些跟FILE I/O段的信息比较，你会发现I/O里有多少是跟日志子系统相关的操作触发的。

13.3.9 BUFFER POOL AND MEMORY

这一节里显示的是InnoDB的BUFFER POOL AND MEMORY的统计信息。（关于如何调优缓冲池的信息，请查看第6章。）

```
 1  ----------------------
 2  BUFFER POOL AND MEMORY
 3  ----------------------
 4  Total memory allocated 4648979546; in additional pool allocated 16773888
 5  Buffer pool size   262144
 6  Free buffers       0
 7  Database pages     258053
 8  Modified db pages  37491
 9  Pending reads 0
10  Pending writes: LRU 0, flush list 0, single page 0
11  Pages read 57973114, created 251137, written 10761167
12  9.79 reads/s, 0.31 creates/s, 6.00 writes/s
13  Buffer pool hit rate 999 / 1000
```

第4行显示了InnoDB申请的内存总数，以及其中有多少是在附加内存池里申请到的。

第5到8行显示了缓冲池的度量值，以页为单位。这些度量值包括了缓冲池的总大小、空闲页的数量、用于存储数据库页的页的数量、和“脏”数据库页的数量。InnoDB在缓冲池里把一些页面用作锁的索引、自适应索引散列及其他系统结果，因此，在池里的数据库页的数量永远不会等于池的总大小。

577

第9到10行显示了未完成的读和写的数目(例如InnoDB要为缓冲池做的逻辑读和写的数目)。这些值会跟FILE I/O段里的数据不匹配，因为InnoDB可能会将许多逻辑操作合并为一个单独的物理I/O操作。缩写语LRU代表了“最少最近使用”，它在缓冲池里把不常用的页刷新到次磁盘上，从而为常用的页腾出空间。刷新列表里保

存的一些旧页，它们须由检查点进程来刷新，单页写属于独立的页写操作，它们不会被合并掉。

第 8 行显示了这个缓冲池里容纳了 37 491 个脏页，在某个点上需要被刷新到磁盘上去（它们已在内存里更新，但是磁盘上的还没更新）。然而，第 10 行也显示出在那一刻并没有刷新操作在进行，这不成问题，因为 InnoDB 会在需要的时候刷新它们。

第 11 行显示了 InnoDB 已经读、创建和写了多少页。页读、写的值来自于从磁盘读入到缓冲池的数据，反之亦然。页创建的值来自于 InnoDB 在缓冲池里申请的页，不包括从数据文件读取出来的内容，因为它不关心内容是什么（例如，它们可能属于一个已被删除的表）。

第 13 行报告了缓冲池的命中率，它可以衡量 InnoDB 在缓冲池里找到所需要页的比例，是缓存效率的度量值。它的衡量范围是自最近一次 InnoDB 状态打印以来的命中率，因此，如果自那时以来，服务器一直很安静，那你就会看到“No buffer pool page gets since the last printout.”。因为 InnoDB 就是这么设计的，你不能直接拿 InnoDB 缓冲池的命中率跟 MyISAM 的关键缓冲区命中率作比较。

13.3.10 ROW OPERATIONS

这一段显示的是 ROW OPERATIONS 及其他各色各样的 InnoDB 统计信息：

```
 1 --------------
 2 ROW OPERATIONS
 3 --------------
 4 0 queries inside InnoDB, 0 queries in queue
 5 1 read views open inside InnoDB
 6 Main thread process no. 10099, id 88021936, state: waiting for server activity
 7 Number of rows inserted 143, updated 3000041, deleted 0, read 24865563
 8 0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
 9 ----------------------------
10 END OF INNODB MONITOR OUTPUT
11 ============================
```

第 4 行显示了 InnoDB 内核里有多少条线程（在 TRANSACTIONS 节里，我们已经谈到过这个）。位于队列里的查询就是 InnoDB 出于限制并发执行的线程数的目的，而还未将其置入内核的线程。这些查询在进入队列前也可以进入睡眠状态——我们在前文里已经讨论过这个了。

第 5 行显示了 InnoDB 已经打开了多少个读视图。一个读视图就是在事务开始之时取得的一张源自数据库内容的一致的 MVCC “快照”。在 TRANSACTIONS 节里，你能查看到一个特定的事务是否拥有一个读视图。

第 6 行显示了内核里主线程的状态。在 MySQL 5.0.45 和 5.1.22 里，可能的状态值包括了下面这些：

- archiving log（如果日志归档功能已经开启的话）
- doing background drop tables
- doing insert buffer merge
- flushing buffer pool pages
- flushing log
- making checkpoint
- purging

- reserving kernel mutex
- sleeping
- suspending
- waiting for buffer pool flush to end
- waiting forserveractivity

第 7、8 行显示了行插入、更新、删除和读取次数的统计信息，以及这些数值在每一秒里的平均值。如果你想知道 InnoDB 正在完成多少工作，查看这些数值就是个不错的办法。

SHOW INNODB STATUS 的输出以第 9 到 13 行为结束。如果你没看到这几行文字，说明你的系统里可能有一个非常大的死锁存在，它截断了这些信息的输出。

13.4 SHOW PROCESSLIST

进程列表就是连接的列表，或者是当前正连接到 MySQL 的线程的列表。SHOW PROCESSLIST 列出这些线程时，也显示了线程的当前状态信息。举例来说：

```
mysql> SHOW FULL PROCESSLIST\G
*************************** 1. row ***************************
     Id: 61539
   User: sphinx
   Host: se02:58392
     db: art136
Command: Query
   Time: 0
  State: Sending data
   Info: SELECT a.id id, a.site_id site_id, unix_timestamp(inserted) AS
inserted,forum_id, unix_timestamp(p
*************************** 2. row ***************************
     Id: 65094
   User: mailboxer
   Host: db01:59659
     db: link84
Command: Killed
   Time: 12931
  State: end
   Info: update link84.link_in84 set url_to =
replace(replace(url_to,'&amp;','&'),'%20','+'), url_prefix=repl
```

有几种工具（例如 innotop）可以为你显示进程列表的一个更新情况。

Command 和 State 列就是线程真正显示“状态”的地方。在本例中，第一条线程运行着一个查询，并且正在发送数据；第二条线程已被杀死了，可能是因为它已经执行了太长的时间却还没结束，有人就用 KILL 命令将其终止了。一条线程能处于这种状态好一会儿，因为 KILL 命令不是立即生效的。举例来说，它可能会花费一些时间回滚线程里的事务。

SHOW FULL PROCESSLIST（这里加上了 FULL 关键字）会显示每一条查询的全部文本，在这里只是截取了前面的 100 个字符。

13.5 SHOW MUTEX STATUS

SHOW MUTEX STATUS返回的是InnoDB互斥量的细节信息，通常是用于深入观察系统的伸缩性和并发性的问题。在代码里面，每一个互斥量保护着一块临界区，这在上文里已经说明过了。

这些输出信息会随着不同的MySQL版本和编译选项而有所不同。有时，你能得到互斥量的名称和每一个互斥量的一些列的信息；有时，你只能得到一个文件名、一条线和一个号码。你需要编写一个脚本来聚合这些输出信息，因为输出信息会比较庞大。以下是仅有一个互斥量的输出示例：

```
*************************** 1. row ***************************
        Mutex: &(buf_pool->mutex)
       Module: buf0buf.c
        Count: 95
   Spin_waits: 0
  Spin_rounds: 0
     OS_waits: 0
    OS_yields: 0
OS_waits_time: 0
```

你可以使用这些输出信息来确定InnoDB的哪一部分已经成为了瓶颈，例如多CPU也会引起瓶颈。最近，MySQL已经修复了很多在多CPU系统里的伸缩性问题，但是，跟互斥量相关的一些问题还没解决。比较典型的一个就是一些人遇到过的AUTO_INCREMEN锁，它是每张表一个，在表内它属于全局的，在InnoDB和插入缓冲区里有一个互斥量保护着它。哪里有互斥量，哪里就有潜在的竞争。

输出里的列信息说明如下：

`Mutex`

互斥量的名称。

`Module`

互斥量在源代码文件中定义的位置。

`Count`

互斥量被请求过的次数。

`Spin_waits`

InnoDB采用循环等待的方式来获取互斥量的次数。你会回忆起InnoDB首先会采用循环等待的方式，然后再回到一个操作系统等待。

`Spin_rounds`

在InnoDB循环等待时，它检查互斥量是否已被释放的次数。

`OS_waits`

Inno返回到操作系统等待以获取互斥量的次数。

`OS_yields`

线程把互斥量让给其他线程的次数。

`OS_waits_time`

如果`timed_mutexes`设为`1`，这就是已经花费的等待时间，以毫秒为单位。

比较这些计数器的相对大小，你就能发现热点所在。这里有3种办法来缓解瓶颈问题：避开InnoDB的虚弱点、限制并发、或者平衡CPU密集的循环等待和资源密集的操作系统等待。关于如何调优InnoDB并发的更多内容，

请看第 296 页的“InnoDB Concurrency Tuning”。

13.6 复制的状态

MySQL 有几个命令是用于监控复制的。在一个主服务器上，SHOW MASTER STATUS 可以显示出主机的复制状态和配置：

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: mysql-bin.000079
        Position: 13847
    Binlog_Do_DB:
Binlog_Ignore_DB:
```


输出信息包括了主服务器上当前二进制日志的位置。使用 SHOW BINARY LOGS 可以得到一个二进制日志的清单：

```
mysql> SHOW BINARY LOGS
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000044 |     13677 |
...
| mysql-bin.000079 |     13847 |
+------------------+-----------+
36 rows in set (0.18 sec)
```

要想看二进制日志里的事件，就要使用 SHOW BINLOG EVENTS 了。

在一个从服务器上，你可以使用 SHOW SLAVE STATUS 来查看从服务器的状态和配置信息。这里就不把它的输出内容示例包括进来了，因为它们有一点冗长，但是，我们会说明其中的一些要点。首先，你能看到从服务器的 I/O 和 SQL 线程的状态，包括其中的任何错误；其次，你能看到这个从服务器在复制上落后主机多少时间；最后，出于备份和克隆从服务器的目的，有另外三套二进制日志的定位坐标放入输出内容里：

`Master_Log_File/Read_Master_Log_Pos`

I/O 线程在主服务器二进制日志上正在读取的位置。

`Relay_Log_File/Relay_Log_Pos`

SQL 线程在从服务器转发日志上正在执行的位置。

`Relay_Master_Log_File/Exec_Master_Log_Pos`

SQL 线程在主服务器二进制日志上正在执行的位置。它跟 Relay_Log_File/Relay_ Log_Pos 在逻辑位置上是一样的，但是，前者在从服务器的转发日志里，而后者是在主服务器的二进制日志里。换句话说，如果你分别到两个日志里查看对应的位置，会发现那一处的日志事件是相同的。

13.7 INFORMATION_SCHEMA

INFORMATION_SCHEMA 数据库里是一套按照 SQL 标准定义的系统视图。MySQL 实现了许多个标准视图，还增加了其他一些视图。在 MySQL 5.1 里，许多视图对应于 MySQL 的 SHOW 命令，例如，SHOW FULL PROCESS LIST 和 SHOW STATUS，但是，也有一些视图没有可对应的 SHOW 命令。

INFORMATION_SCHEMA 视图的美感在于你可以使用标准的 SQL 查询它们，由此带来的兼容性超过了 SHOW 命令，SHOW 命令只能生成结果，无法再聚合、联接，更无法用标准 SQL 来操作。有了系统视图里的所有这些可用数据，你就能编写出一些有趣又有用的查询。

举例来说，你想知道 Sakila 示例数据库里有哪些表引用了 actor 表吗？一致的命名约定使这个问题显得相对比较简单：


```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='sakila' AND COLUMN_NAME='actor_id'
    -> AND TABLE_NAME <> 'actor';
+------------+
| TABLE_NAME |
+------------+
| actor_info |
| film_actor |
+------------+
```

我们要找出本书几个示例中哪些有多列索引的表，以下就是针对它的查询：

```
mysql> SELECT TABLE_NAME, GROUP_CONCAT(COLUMN_NAME)
    -> FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    -> WHERE TABLE_SCHEMA='sakila'
    -> GROUP BY TABLE_NAME, CONSTRAINT_NAME
    -> HAVING COUNT(*) > 1;
+---------------+--------------------------------------+
| TABLE_NAME    | GROUP_CONCAT(COLUMN_NAME)            |
+---------------+--------------------------------------+
| film_actor    | actor_id,film_id                     |
| film_category | film_id,category_id                  |
| rental        | customer_id,rental_date,inventory_id |
+---------------+--------------------------------------+
```

你也能写出更加复杂的查询，就像你基于任何普通表上做的那样。MySQL Forge（*http://forge.mysql.com*）就是查找和分享这些查询语句的好地方，它拥有很多实例，例如找出重复的或者冗余的索引、用低基数找到索引、以及其他很多很多的查询语句。

这些视图的最大缺点就是跟 SHOW 命令相比，查询它们有时会非常慢。它们典型的工作方式是获取所有数据，存放在一个临时表内，然后在临时表上执行查询。对于许多监控、故障排除和调优等用途而言，与其键入完整的 SQL 语句从视图里取得数据，还不如直接键入 SHOW 命令来得快。

在本书编写的时候，这些视图还是无法被更新的。虽然你可以从它们那里获取到服务器的设置信息，但是，你无法更新它们，也无法从这个途径更改服务器的配置。在实际使用过程中，这些限制条件意味着你还是要使用 SHOW 和 SET 命令来配置服务器，哪怕是 INFORMATION_SCHEMA 的视图对于其他任务非常有用。