



数据库领域经典著作全面升级，从实战出发，增补MySQL 5.7和MariaDB 10.1版本的全新内容  
以构建高性能MySQL服务器为核心，从故障诊断与优化、性能调优、备份与恢复、高可用集群架构  
搭建与管理、读写分离和分布式DB架构搭建与管理、性能和服务监控等多角度深入分析管理与维护  
MySQL服务器的技巧和方法，是指导MySQL DBA进阶修炼的最佳实践之作

数据库  
技术丛书

# MySQL管理之道

性能调优、高可用与监控

第2版



贺春阳 著

I

机械工业出版社  
China Machine Press

MySQL管理之道：性能调优、高可用与监控

## 版权信息

书名：MySQL管理之道：性能调优、高可用与监控（第2版）

丛书名：数据库技术丛书

作者：贺春暘

ISBN：9787111547792

本书由北京华章图文信息有限公司授权京东阅读电子版制作与发行

版权所有·侵权必究



## 推荐序：飞跃的第2版

MySQL作为一个开源项目，已经有20年历史了，最近几年在互联网核心系统中的成功使用奠定了其在关系数据库中的地位，也成为让每一个DBA、开发人员、架构师及CTO都不得不考虑的数据库基础软件。

对于数据库爱好者来讲，Oracle是非常值得研究的数据库，因为其历史悠久、功能卓越；而MySQL则是非常适合研究的，因为它的代码、协议及外围配套工具具有开放性。国内有一大批在各个企业成功实施过MySQL的优秀工程师，在完成工作之余，他们还积累了丰富的知识和经验，并提炼总结著成书籍，以帮助其他人，本书作者就是其中之一。

本书的第1版帮助了不少人入门MySQL，我在学习MySQL的过程中也参阅过第1版，以便了解不同企业使用MySQL的业务场景和遇到的技术难题，以及最后所用的解决方案。用心的读者是用心的作者最大的动力和回报。

最近三四年里，MySQL发展极快，包括官方的MySQL版本以及MariaDB分支的发展，更重要的是在企业的各类系统中它的应用也越来越广泛和深入。随着它的发展，架构师及运维主管的工作也更具有挑战性，此时，作者用心编写第2版，不只是简单的完善和改版，还可以理解为重写和飞跃。

现在，数据库管理员已经变成了数据管理员，反映的是理念和架构的变化，第2版中新增的内容更多地印证了这一点，这一版对高可用、自动切换、数据保护等方案的透析更加深入，数据库中间件部分的深入分析更能拓展广大DBA的视野。

如果将一本书比作一个人，那么第1版聚焦于DBA工作，而第2版则聚焦于架构师的思考，并且还可以随时随地联系作者进行深入交流，不再只是局限于本书中的内容！

平民软件 ( <http://www.onexsoft.com> ) 楼方鑫



## 前言

### 为什么要写这本书

首先要感谢读者对第1版的认可。随着技术的更新，第1版的内容已逐渐变老，为了与时俱进，所以准备再写一本关于MariaDB 10和MySQL 5.7的数据库图书，把自己学到的新知识做一个系统性总结来呈现给大家。目前市面上针对相关知识进行介绍的书还寥寥无几，大多数读者只能通过阅读英文手册去获取新的知识，希望本书的出版能对大家有所帮助。

本书以构建高性能MySQL服务器为核心内容，介绍了MariaDB 10和MySQL 5.7的新特性，并从故障诊断与优化、性能调优、备份与恢复、MySQL高可用集群搭建与管理、MySQL服务器性能和服务监控等角度深入讲解了如何去管理与维护MySQL服务器。书中内容均来自于笔者多年实践经验的总结和新知识的拓展，同时也包含很多实用的情景模拟，并针对运维人员、DBA等相关工作者常遇到的有代表性的疑难问题给出了解决方案。不论你目前有没有遇到过此类问题，相信都会有借鉴意义。

### 如何阅读本书

本书的知识结构分四部分：

第一部分（第1章至第2章）介绍MySQL5.7/MariaDB 10的新特性、注意事项、安装和升级方法。

第二部分（第3章至第6章）为故障诊断与优化，涉及生产环境下MySQL故障处理，以及性能调优等内容，包括表设计阶段范式的理解、字段类型的选取、采用表锁还是行锁、MySQL默认的隔离级别与传统SQL Server，以及Oracle数据库默认的隔离级别的区别、SQL语句的优化，以及合理利用索引等。

第三部分（第7章至第10章）为架构篇，内容包括当前互联网流行的高可用架构MHA（Master High Availability）、分库分表中间件Onepoxy和读写分离中间件MariaDB MaxScale，以及Percona/MariaDB Galera Cluster集群管理。

第四部分（第11章）阐述慢SQL管理平台的搭建与维护，主要介绍集中收集慢日志查询。

本书的每个部分都可以单独作为一本迷你书阅读，如果你未接触MySQL5.7/MariaDB 10，建议从第一部分开始阅读。本书提供的脚本和相关软件，请在华章网站（[www.hzbook.com](http://www.hzbook.com)）的本书页面下载。

### 勘误和支持

由于作者的水平有限，编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，不妥之处恳请读者批评指正。你可以将书中的

错误，发送邮件至我的邮箱chunyang\_he@139.com或者通过QQ联系我：3783414，我很期待能够听到你们真挚的反馈。

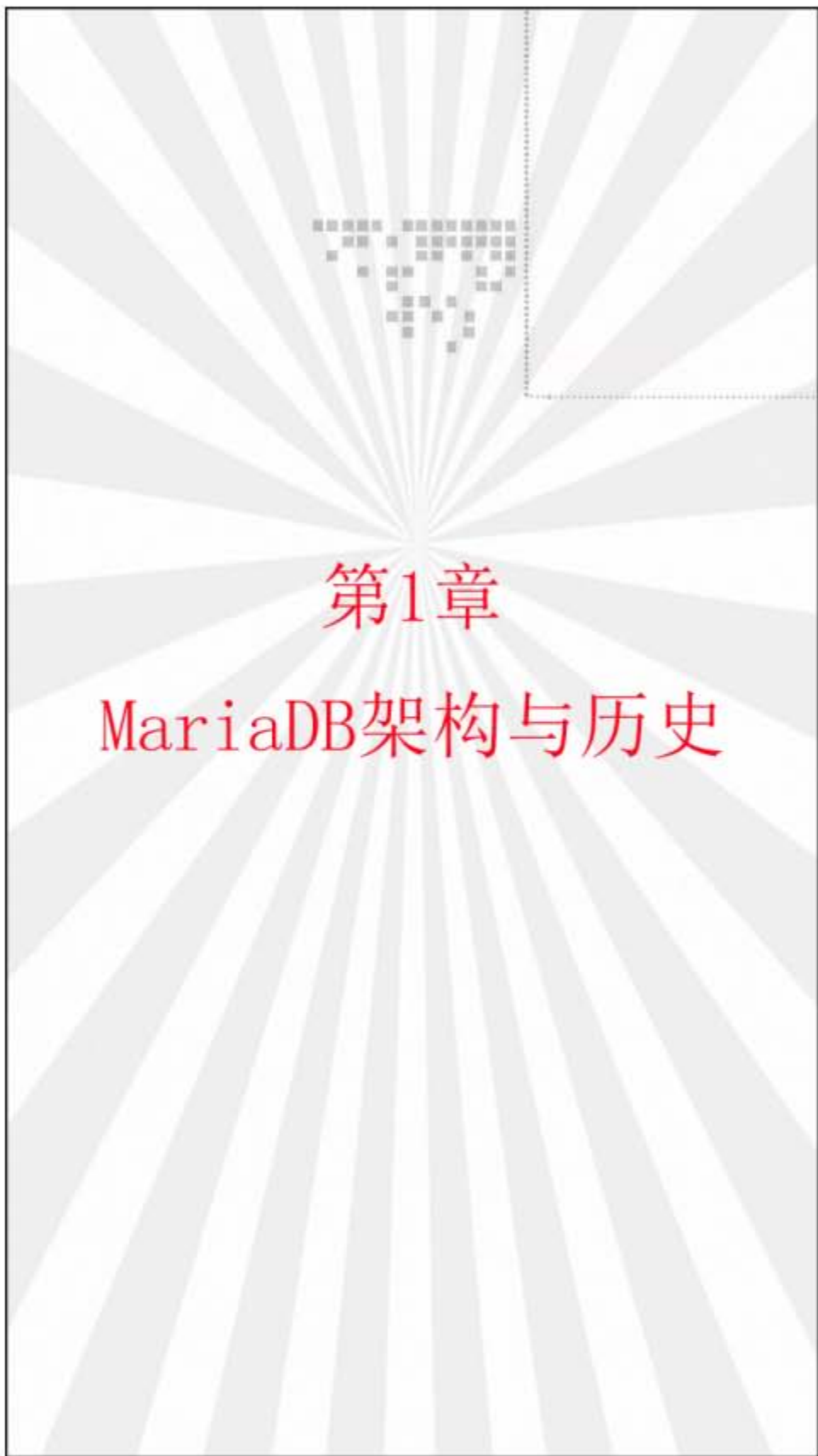
## 致谢

在这里感谢沃趣科技公司高级DBA邱文辉提供《MariaDB 10 Hash Join索引优化》一文。

感谢机械工业出版社华章公司的编辑杨绣国老师，感谢你的魄力和远见，在这一年多的时间中始终支持我的写作，你的鼓励和帮助引导我顺利完成全部书稿。

贺春暘

2016年5月于北京





本书以MariaDB 10.1和MySQL 5.7为主要介绍对象，为了让广大读者了解什么是MariaDB，首先对其进行介绍。



## 1.1 MariaDB的介绍

MariaDB是MySQL源代码的一个分支，主要由开源社区在维护，采用GPL授权许可。开发这个分支的原因之一：甲骨文公司收购了MySQL后，有将MySQL闭源的潜在风险，因此社区采用分支的方式来避开这个风险。MariaDB是完全兼容MySQL的，包括API和命令行，使之能轻松成为MySQL的代替品。在存储引擎方面，使用XtraDB来代替MySQL的InnoDB，XtraDB完全兼容InnoDB。创建一个InnoDB表，内部默认会转换成XtraDB，如图1-1所示。

```
MariaDB [(none)]> show engines;
```

Engine	Support	Comment	Transactions	XA	Savepoints
MRG_MyISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
Aria	YES	Crash-safe tables with MyISAM heritage	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
InnoDB	DEFAULT	Percona-XtraDB, Supports transactions, row-level locking, foreign keys and encryption for tables	YES	YES	YES
SEQUENCE	YES	Generated tables filled with sequential values	YES	NO	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO

8 rows in set (0.06 sec)

```
MariaDB [(none)]> select version();
```

version()
10.1.10-MariaDB-enterprise-log

1 row in set (0.10 sec)

图1-1 默认XtraDB存储引擎

Percona XtraDB是InnoDB存储引擎的增强版，可用来更好地发挥最新的计算机硬件系统性能，同时还包含一些在高性能环境下的新特性。XtraDB存储引擎是完全向下兼容的，在MariaDB中，XtraDB存储引擎被标识为"ENGINE=InnoDB"，这与InnoDB是一样的，所以可以直接用XtraDB替换掉InnoDB，而不会产生任何问题。XtraDB在InnoDB的基础上构建，使XtraDB具有更多的特性、更多的参数指标和更多的扩展。从实践的角度来看，XtraDB在CPU多核的条件下会更有效地使用内存，并且性能更高。从MariaDB 5.1开始就默认使用XtraDB存储引擎。

MariaDB由MySQL的创始人Michael(Monty)Widenius主导开发，早前他曾以10亿美元的价格将自己创建的公司MySQL AB卖给了Sun公司，此后，随着Sun公司被甲骨文公司收购，MySQL的所有权也落入Oracle公司的手中。MariaDB名称来自Michael(Monty)Widenius的女儿Maria的名字。



## MariaDB 10.0和MySQL 5.6的不同之处

MySQL 5.6的代码库的文件结构已经被改动。比如单个代码文件已经被分成多个，又或者某些代码已经被重新归类到不同的文件内。所以要MariaDB去配合现在这个文件结构，一定是一个非常消耗时间的过程。

MariaDB 5.5已经有大量的代码不同于MySQL 5.5的版本，而且也有很多新的特征被整合到MariaDB 5.5中，而这些特征直到MySQL 5.6才出现在MySQL中。所以，在比较同样功能的MySQL和MariaDB的版本，同时在完成设计和QA方面的审核后，一个很明显的结论为MariaDB是一个更好的产品。大多数情况下，当选择MariaDB的时候，人们会更多地考虑功能方面的偏好。

MariaDB不仅仅是MySQL的一个替代品。它的主要目的是创新和提高MySQL的技术，MySQL 5.6不是一个合适的创新基础平台，所以MariaDB团队做了下面的事情。

- 引入了一些新功能（像Multi-source Replication多源复制、基于表的并行复制、Galera Cluster集群、Spider水平分片存储引擎、TokuDB存储引擎等），所以需要一个新版本。

- 下一个版本称为“MariaDB 5.6”是不准确的，因为它不是基于MySQL 5.6的，取而代之，MariaDB团队决定版本号调为10.0。

MariaDB和Percona有什么不同呢？Percona仅是针对InnoDB引擎做了性能上的改善（称为XtraDB），而MariaDB在集成了XtraDB存储引擎之外，还集成了更多的存储引擎，包括Aria、SphinxSE、TokuDB、Cassandra、CONNECT、SEQUENCE及Spider存储引擎等，并且在服务器层上做了大量改进，增加了多源复制和基于表的并行复制等。

## 1.2 MariaDB和MySQL的兼容性

MariaDB和MySQL在绝大多数方面是兼容的，对于前端应用（比如PHP、Perl、Python、Java、.NET、MyODBC、Ruby、MySQL C connector）来说，几乎感觉不到任何不同。目前MariaDB是发展最快的MySQL分支版本，新版本的发布速度已经超过了Oracle公司官方的MySQL版本。



注意 MariaDB 10.0/10.1的GTID复制与MySQL 5.6不兼容。

在Oracle公司控制下的MySQL开发有以下两个主要问题。

- MySQL核心开发团队是封闭的，完全没有Oracle公司之外的成员参加。很多高手即使有心做贡献，也没办法做到。
- MySQL新版本的发布速度在Oracle公司收购Sun公司之后大大减缓。

Michael(Monty)Widenius用数据比较了收购之前和之后新版本的发布速度，并表示有很多bugfix和新的feature都没有及时加入发布版本中。

以上这两个问题，导致了各大公司都开发了自己定制的MySQL版本，包括Yahoo、Facebook、Google、阿里巴巴和淘宝网等。MySQL是开源社区的资产，任何个人/组织都无权据为己有。为了更快速地发展MySQL，另外开分支是必须的。



### 1.3 MariaDB 10.0新增的功能

通过下面官网的对比图（见图1-2~图1-5），可以清晰地看到MariaDB 10新增加的功能。

Compare Products	MySQL 5.6	MariaDB 10
What is it?	MySQL 5.6 is a popular choice of database for use in web applications, and is a central component of the widely used LAMP web application software stack.	MariaDB 10 is a enhanced, high performance, free and open source alternative to MySQL that helps the world's busiest websites deliver more content faster.
SCALABILITY		
Parallel Slave Replication [more]	Single threaded per database	✓
Multi-source Replication [more]		✓
Global Transaction ID [more]	Limited	✓
Sharding - Spider Storage Engine [more]	3rd party	✓
Table Partitioning: Improvements [more]	✓	✓

图1-2 可伸缩性和可扩展性的对比

PERFORMANCE		
TokuDB Storage Engine [more]	3rd party	✓
Improved InnoDB storage engine [more]	✓	✓
Performance Schema [more]	✓	✓
Improved thread pool [more]	MySQL Enterprise only	✓
Fusion-io specific enhancements [more]		✓
Engine Independent Table Statistics [more]		✓
Subquery Optimizations [more]		✓
Histogram Stats for Non-Indexed Columns [more]		✓

图1-3 性能对比

NOSQL CAPABILITIES		
CONNECT storage engine [more]		✓
Sequence storage engine [more]		✓
NoSQL Cassandra Storage Engine [more]		✓
Dynamic Columns [more]		✓
NoSQL Handlersocket interface [more]		✓
NoSQL memcache interface [more]	✓	I

图1-4 NoSQL支持对比

OPERATIONS		
Improved table discovery [more]		✓
SHOW PLUGINS SONAME [more]		✓
SHUTDOWN Command [more]		✓
Kill query by query ID [more]		✓
SHOW EXPLAIN Command [more]		✓
Per-thread Memory Statistics [more]		✓
Improved Error Messages [more]		✓
Online ALTER TABLE [more]	✓	✓
SECURITY & COMPLIANCE		
Role-based access control [more]		✓
Audit Plugin [more]	MySQL Enterprise only	✓
PAM Authentication Plugin [more]	MySQL Enterprise only	✓

图1-5 操作和安全上的对比

从图1-2~图1-5可以看到，在功能上，MariaDB 10.0已经完胜MySQL 5.6，MySQL 5.7才慢慢将缺失的功能追补上来，并且线程池、审计日志等功能是出现在MySQL企业版里的，需要付费。

### 1.3.1 更多的存储引擎

除了包含标准的MyISAM、BLACKHOLE、CSV、MEMORY、ARCHIVE和MERGE等存储引擎外，MariaDB的源代码包和二进制包还包含以下额外的存储引擎。

- Aria ( 增强版的MyISAM )
  - XtraDB ( 增强版的InnoDB )
  - FederatedX
  - OQGRAPH
  - SphinxSE
  - IBMDB2I
  - TokuDB
  - Cassandra
  - CONNECT
  - SEQUENCE
  - Spider
  - PBXT
- SphinxSE是一个可以编译进MySQL5.x版本的MySQL存储引擎，它利用了该版本MySQL的插件式体系结构。尽管SphinxSE被称为“存储引擎”，但其自身并不存储任何数据。它其实是一个允许MySQL服务器与searched交互并获取搜索结果的嵌入式客户端。所有的索引和搜索都发生在MySQL之外。
  - 可以把TokuDB看成是ARCHIVE存储引擎的升级版，典型特征：插入速度快、压缩效率高、支持事务和MVCC版本控制。可以试图把一些LOG日志表改为TokuDB引擎，这样在性能和磁盘空间使用率上都有较大幅度的提升。此外，该引擎的备份工具是收费，所以备份的时候可以采取冷备份。
  - Spider为分库、分表存储引擎，腾讯的Tspider是基于MariaDB官方Spider引擎二次研发的，性能提高了30%，官方原版Spider目前还不稳定，性能有待加强。



### 1.3.2 速度的提升

在MariaDB 5.3版本里，就已经对子查询进行了优化，并采用semi join半连接方式将SQL改写为表关联join，从而提高了查询速度。

在MariaDB 5.3版本里引入group commit for the binary log组提交技术，简单来说，将多个并发提交的事务加入一个队列里，再将这个队列里的事务利用一次I/O合并提交，从而解决写日志时频繁刷磁盘的问题。

在MariaDB 10.0版本里引入基于表的多线程并行复制技术，如果主库上1秒内有10个事务，那么合并一个I/O提交一次，并在binlog里增加一个cid=XX标记，当cid的值一样时，Slave就可以进行并行复制，通过设置多个sql\_thread线程实现。在MySQL 5.5版本里是单进程串行复制，通过sql\_thread线程来恢复主库推送过来的binlog，这样会产生一个问题，主库上有大量的写操作，从库就有可能出现延迟。MySQL 5.6是基于库级别的并行复制，MySQL 5.7是基于表级别的并行复制。

在MariaDB 5.5版本里引入线程池thread pool技术，线程池的连接复用减少了建立连接的开销，减少了CPU上下文切换，非常适合高并发php短连接应用场景（如使用开源电商平台ECSHOP秒杀业务场景）。

处理内部的临时表时，MariaDB用Aria引擎代替了MyISAM引擎，这会使某些GROUP BY和DISTINCT请求速度更快，因为Aria有比MyISAM更好的缓存机制。

### 1.3.3 扩展和新功能

MariaDB 对服务层做了大量改善，增加了很多新的特性，如果一个补丁或功能是有用的、安全的、稳定的，那么 MariaDB 官方会尽一切努力在 MariaDB 发行版包括它。MariaDB 5.3 版本里最显著的特点如下。

#### 1. 时间精确到微秒级别

一个额外的列 `TIME_MS` 已被添加到 `INFORMATION_SCHEMA.PROCESSLIST` 表里，用于查看微秒时间，如图 1-6 所示。

```
MariaDB [test]> select id, time, time_ms, command, state from information_schema.processlist;
```

id	time	time_ms	command	state
4	0	0.769	Query	Filling schema table
3	6320	6320828.952	Sleep	

2 rows in set (0.00 sec)

图1-6 显示微秒时间

#### 2. 提供了虚拟列（函数索引）

在 MariaDB 5.2 版本里就已经提供了虚拟列（函数索引），但直到 MySQL 5.7 版本才支持。

#### 3. kill 命令扩展

在 MariaDB 5.3 版本里又对 kill 命令进行了扩展，可以指定某个 user 用户，杀死所有查询（见图 1-7）。

```
MariaDB [(none)]> show processlist;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host       | db  | Command | Time | State | Info           | Progress |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
|  8 | root | localhost | NULL | Query   |  0   | init  | show processlist | 0.000    |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

MariaDB [(none)]> kill user root;
Query OK, 1 row affected (0.00 sec)

MariaDB [(none)]> show processlist;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id: 9
Current database: *** NONE ***

+----+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host       | db  | Command | Time | State | Info           | Progress |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
|  9 | root | localhost | NULL | Query   |  0   | init  | show processlist | 0.000    |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

MariaDB [(none)]> █
```

图1-7 杀死root账号连接

#### 4.修改表结构可显示执行进度

通过alter table命令可以显示执行进度，如图1-8所示。

```
(root@localhost) [test]> alter table sbtest modify c char(200);
Stage: 1 of 2 'copy to tmp table' 32.1% of stage done
```

图1-8 修改表结构显示执行进度

#### 5.提供了动态列（可以存储JSON格式）

在MariaDB 5.3版本里就已经提供了动态列（可以存储JSON格式），但直到MySQL 5.7版本才支持。

在MariaDB 10.0版本里则有以下显著特点。

- 提供多源复制，但直到MySQL 5.7版本才支持。
- 支持GTID同步复制。



- 创建用户支持创建角色 ( role ) 权限。
- 通过show processlist可以查看内存占用，如图1-9所示。
- 执行create or replace table，等于先执行drop操作，再执行create操作，如图1-10所示。

```
(root@localhost) [(none)]> select * from information_schema.processlist
-> order by memory_used limit 1\G;
***** 1. row *****
      ID: 540940
     USER: root
    HOST: localhost
       DB: NULL
  COMMAND: Query
      TIME: 0
   STATE: Filling schema table
    INFO: select * from information_schema.processlist
order by memory_used limit 1
   TIME_MS: 1.176
     STAGE: 0
  MAX_STAGE: 0
  PROGRESS: 0.000
  MEMORY_USED: 83008
EXAMINED_ROWS: 0
    QUERY_ID: 2923574
1 row in set (0.00 sec)

ERROR: No query specified

(root@localhost) [(none)]>
```

图1-9 显示内存使用

```
(root@localhost) [test]> create or replace table t1(id int);
Query OK, 0 rows affected (0.00 sec)

(root@localhost) [test]> create or replace table t1(id int,name char(10));
Query OK, 0 rows affected (0.00 sec)

(root@localhost) [test]> desc t1;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)   | YES  |     | NULL    |       |
| name  | char(10)  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

图1-10 create语法扩展

执行delete from table returning命令，在删除前，可以返回要删除的记录，以便万一失手，可以拿到原始数据，如图1-11所示。

```
(root@localhost) [test]> select * from t1;
+-----+-----+
| id    | name |
+-----+-----+
| 1     | aa   |
| 2     | bb   |
| 3     | cc   |
+-----+-----+
3 rows in set (0.00 sec)

(root@localhost) [test]> delete from t1 returning id,name;
+-----+-----+
| id    | name |
+-----+-----+
| 1     | aa   |
| 2     | bb   |
| 3     | cc   |
+-----+-----+
3 rows in set (0.00 sec)

(root@localhost) [test]> select * from t1;
Empty set (0.00 sec)

(root@localhost) [test]>
```

图1-11 delete语法扩展

慢查询日志 (slow log) 里增加了explain执行计划。

在my.cnf里添加以下代码：

*log-slow-verbosity = query\_plan, explain*

会在慢日志里把执行计划也打印出来，以方便排查问题，如图1-12所示。



```
[root@master mariadb101]# cat mysql-slow.log
/usr/local/mysql/bin/mysqld, Version: 10.1.10-MariaDB-enterprise-log (MariaDB Enterprise Certified Binary). started with:
Tcp port: 3306 Unix socket: /tmp/mysql.sock
Time          Id Command  Argument
# Time: 160114 12:50:07
# User@Host: root[root] @ localhost []
# Thread_id: 3 Schema: test QC_hit: No
# Query_time: 2.982543 Lock_time: 0.074857 Rows_sent: 1 Rows_examined: 1000000
# Rows_affected: 0
# Full_scan: Yes Full_join: No Tmp_table: No Tmp_table_on_disk: No
# Filesort: No Filesort_on_disk: No Merge_passes: 0 Priority_queue: No
#
# explain: id      select_type      table  type      possible_keys  key      key_len ref      rows      r_rows  filtered      r_filtered      Extra
# explain: 1      SIMPLE  sbtest  index  NULL         k         4      NULL    889247 1000000.00    100.00 100.00 Using index
#
use test;
SET timestamp=1452747007;
select count(*) from sbtest;
[root@master mariadb101]#
```

图1-12 慢日志

## 1.4 如何将MySQL迁移至MariaDB

以MySQL 5.1版本为例，若要升级到MySQL 5.6，需要进行一次全库mysqldump导出再导入，当数据库很大时，比如为100GB，那么升级起来会非常困难。但如果升级为MariaDB 10版本，则会非常轻松，按照官方文档阐述，只需把MySQL卸载掉，并用MariaDB启动，然后通过mysql\_upgrade命令升级即可完成，如图1-13所示。

For all practical purposes, you can view MariaDB as an upgrade of MySQL:

- If you are using Windows, see also Upgrading MariaDB on Windows.
- For upgrading from very old MySQL versions, see Upgrading to MariaDB from MySQL 5.0 (or older version).
- Within the same base version (for example 5.1) you can just uninstall MySQL and install MariaDB and you are good to go. There is no need to dump and restore databases. As with any upgrade, we recommend making a backup of your data beforehand.
- You should run `mysql_upgrade` (just as you would with MySQL) to finish the upgrade. This is needed to ensure that your mysql privilege and event tables are updated with the new fields MariaDB uses. Note that if you use a MariaDB package, `mysql_upgrade` is usually run automatically.
  - After running `mysql_upgrade` you should restart MariaDB so that the new changes takes effect.
- All your old clients and connectors (PHP, Perl, Python, Java, etc.) will work unchanged (no need to recompile). This works because MariaDB and MySQL use the same client protocol and the client libraries are binary compatible. You can also use your old MySQL connector packages with MariaDB if you want.

图1-13 升级步骤

当处理内部的临时表时，MariaDB 5.5/10.0用Aria引擎代替MyISAM引擎，MariaDB 10.1可以通过设置参数 `default_tmp_storage_engine=InnoDB` 作为内部临时表存储引擎，这将使某些GROUP BY和DISTINCT请求速度更快，因为Aria有比MyISAM更好的缓存机制。如果临时表很多，则要增加 `aria_pagecache_buffer_size` 参数的值（缓存数据和索引），默认为128MB（而不是 `tmp_table_size` 参数）。如果没有MyISAM表，那么建议把 `key_buffer_size` 调低，而且要调得非常低，例如64KB，仅提供给MySQL库中的系统表使用，如图1-14所示。



## Upgrading my.cnf

All the options in your original MySQL my.cnf file should work fine for MariaDB.

However as MariaDB has more features than MySQL, there is a few things that you should consider changing in your my.cnf file.

- MariaDB uses by default the Aria storage engine for internal temporary files, instead of MyISAM. If you have a lot of temporary files, you should add and set `aria-pagecache-buffer-size` to the same value as you have for `key-buffer-size`.
- If you don't use MyISAM tables, you can set `key-buffer-size` to a very low value, like 64K.

图1-14 内部临时表存储引擎

什么是Aria呢？Aria是早期MariaDB版本的默认存储引擎，自2007年以来它一直在开发，当前版本是Aria 1.5，下一个版本是Aria 2.0。Aria引擎前身为Maria，后来怕和MariaDB数据库搞混淆，又重新命名，是增强版的MyISAM，解决了MyISAM崩溃安全恢复问题，也就是说，mysqld的进程崩溃后，Aria将恢复所有表。

### 1. Aria引擎的宕机恢复

数据和索引支持崩溃安全恢复，在崩溃后，表数据的变化将回滚语句的开始状态或最后一个LOCK TABLES的命令状态（见图1-15）。

```

[root@MariaDB1 data]# tail -f error.log
140626 19:01:10 mysqld_safe Number of processes running now: 0
140626 19:01:10 mysqld_safe mysqld restarted
140626 19:01:14 [Note] mysqld: Aria engine: starting recovery
recovered pages: 0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100% (10.8 seconds); transactions to roll back: 1 0 (1.3 seconds); tables to
flush: 1 0 (1.3 seconds);
140626 19:01:27 [Note] mysqld: Aria engine: recovery done

```

图1-15 崩溃恢复

### 2. Aria引擎的未来和发展

官方在未来会让Aria全面支持事务，但现在只是加入计划当中，并不在它们内部开发的优先级列表中。目前这个引擎官方都已暂停开发，现在重点都放在了改善MariaDB上。当前的目标是保持稳定和修复所有发现的bug并进行修复。

### 3. 实验验证当前版本不支持事务

下面做个实验验证当前版本是否支持事务，启用关键字TRANSACTIONAL=1。

```

MariaDB [test]> create table t5 (id INT) ENGINE=ARIA TRANSACTIONAL=1;
Query OK, 0 rows affected (0.99 sec)

```

```

MariaDB [test]>begin;
Query OK , 0 rows affected (0.07 sec)
MariaDB [test]>insert into t5 values(1);
Query OK , 1 row affected (0.94 sec)
MariaDB [test]>rollback;
Query OK , 0 rows affected , 1 warning (0.00 sec)
MariaDB [test]>select * from t5;
+-----+
| id |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

```

从以上代码中可以看到并没有进行回滚。



## 1.5 使用二进制包安装MariaDB 10.1企业版

访问MariaDB官方下载企业版，需要先注册一个用户才能下载，下载地址为：<https://mariadb.com/resources/downloads>

安装过程如下：

```
shell>groupadd mysql
shell>useradd -g mysql mysql
shell>cd /usr/local
shell>tar zxvf mariadb-enterprise-10.1.10-linux-x86_64.tar.gz
shell>ln -s mariadb-enterprise-10.1.10-linux-x86_64 mysql
shell>chown -R mysql:mysql mysql/
shell>scripts/mysql_install_db --user=mysql
shell>chown -R mysql:mysql /data/
shell>bin/mysqld_safe --defaults-file=/etc/my.cnf --user=mysql &
```

官方推荐使用jemalloc内存管理器获取更好的性能，如图1-16所示。

- If your applications often connect and disconnect to MariaDB, you should set up `thread-cache-size` to the number of concurrent queries threads you are typically running. This is important in MariaDB as we are using the jemalloc memory allocator. jemalloc usually has better performance when running many threads compared to other memory allocators, except if you create and destroy a lot of threads, in which case it will spend a lot of resources trying to manage thread specific storage. Having a thread cache will fix this problem.

图1-16 MariaDB调用jemalloc内存管理器

```
# yum install jemalloc* -y
```

(Centos系统需要先安装epel.repo源。)

将下面的参数加入my.cnf里，在启动MySQL时使其生效，如图1-17所示。

```
MariaDB [(none)]> show variables like '%malloc%';
+-----+-----+
| Variable_name | Value               |
+-----+-----+
| innodb_use_sys_malloc | ON                  |
| version_malloc_library | bundled jemalloc    |
+-----+-----+
2 rows in set (0.00 sec)
```

图1-17 jemalloc内存管理器已使用

`[mysqld_safe]`

`malloc-lib = /usr/lib64/libjemalloc.so`

内存管理器性能对比如图1-18所示。

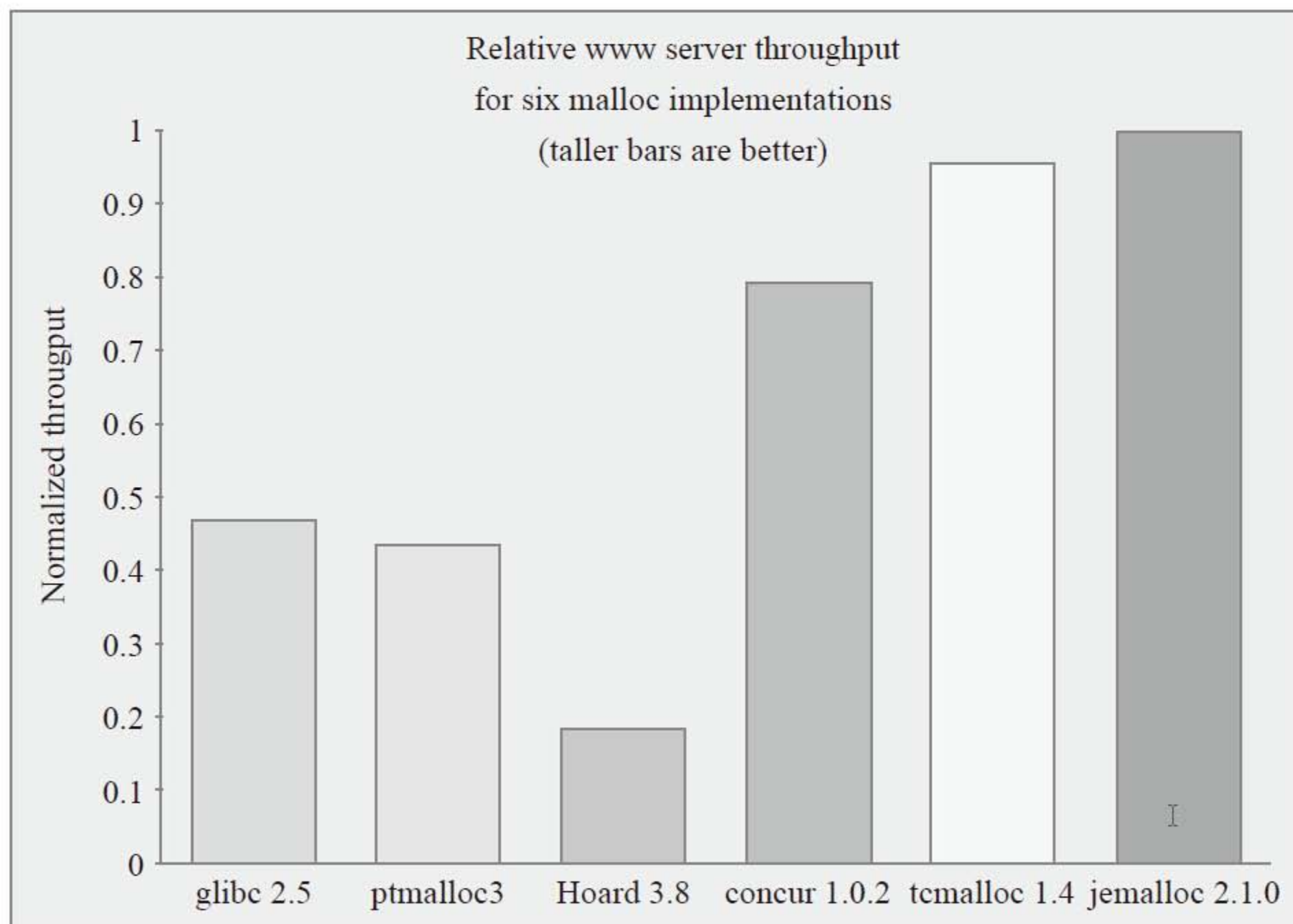


图1-18 性能对比



## 1.6 总结

MariaDB是甲骨文公司MySQL的加强版本，因此已有的系统不需要任何修改就可以运行，就像使用Percona Server一样。

MariaDB社区版和企业版的源代码都是开源的，并且所有功能都是免费开放的，不用担心功能上有删减，但甲骨文公司的MySQL企业版延伸套件采取封闭源代码且需要付费。

此外，相比MySQL，MariaDB拥有更多的功能，速度更快，性能更稳定，Bug修复速度也更快。



MySQL是一个中小型的关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle公司。由于它具有性能高、成本低、可靠性好等特点，近几年已经成为最流行的开源数据库，被广泛地应用在Internet的中小型网站中。而且，随着MySQL的不断成熟，现在它也逐渐用于更多大规模的网站和应用了，比如维基百科、Google和Facebook等。非常流行的开源软件组合“LAMP”中的“M”指的就是MySQL。

这几年，MySQL的版本在不断变更，可以说是有了翻天覆地的变化，在之前的4.0版本中，没有存储过程、触发器、函数、事件，对CPU多核的支持也不好，在经历了5.0和5.1两个过渡版本后，到5.5版，其性能和功能已经得到了很大改善，主要体现在CPU多核处理速度上有了很大提高，宕机恢复时间减少，可快速创建索引，并具有半同步复制功能等方面，目前这个版本很稳定。2015年11月，最新版本5.7已经出了GA版，但这里不推荐直接将其用于生产环境中，因为该版本还有许多未知bug在修复，建议1年后再开始应用，不要充当小白鼠，本书也只会针对该版本部分功能上的新特性进行介绍。

本章主要讲解MySQL 5.7和InnoDB的一些增强性能，这些增强性能极大地提高了系统和MySQL的性能。下面将详细介绍每一个关键的增强性能及其实现过程。

为了不误导读者，保证全文的准确性，下面的内容会结合MySQL 5.7官方手册《What Is New in MySQL 5.7》中的内容来讲解，以帮助读者认识MySQL 5.7中一些较为重要的变化，其中也许会有疏漏的地方，不到之处请大家访问<http://dev.mysql.com/doc/refman/5.7/en/mysql-nutshell.html>参考相关的英文文档。



注意 本章列出了MySQL 5.7和MariaDB 10.1的关键新特性。



## 2.1 性能提升

MySQL 5.7在支持多处理器和高度并发CPU线程的系统上，提供更持续的线性性能和扩展性。实现这一点的关键是通过改进Oracle InnoDB存储引擎的效率和并发性，来消除InnoDB内核中原有的线程争用和互斥锁定的现象。通过这些改进，MySQL可以充分利用当今基于x86的商用硬件先进的多线程处理能力。

在OLTP只读模式下，MySQL 5.7有近100万的QPS（Queries Per Second），比MySQL 5.6性能高3倍，如图2-1所示。

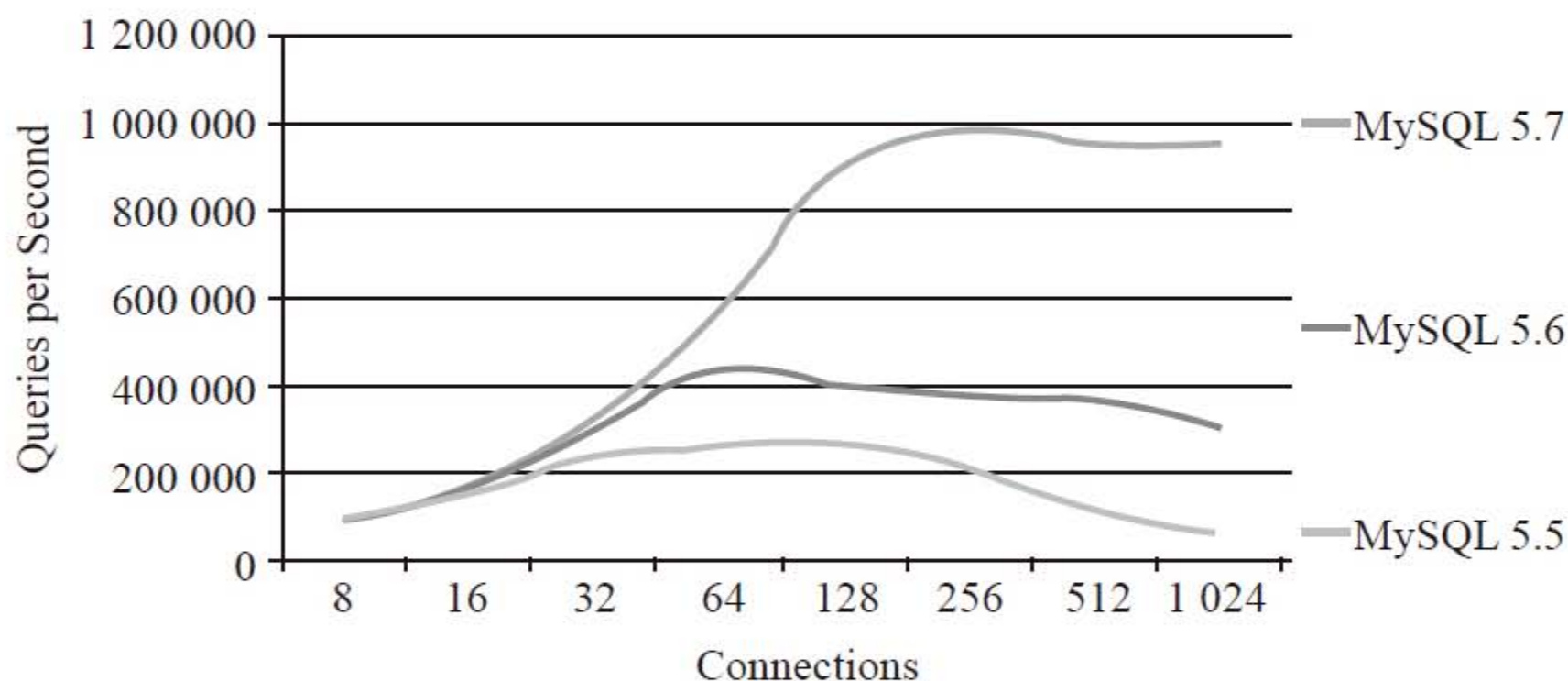


图2-1 sysbench只读模式-每秒查询数

在OLTP读/写模式下，MySQL 5.7压缩到了近60万的TPS，比MySQL 5.6的性能提升了2倍，如图2-2所示。

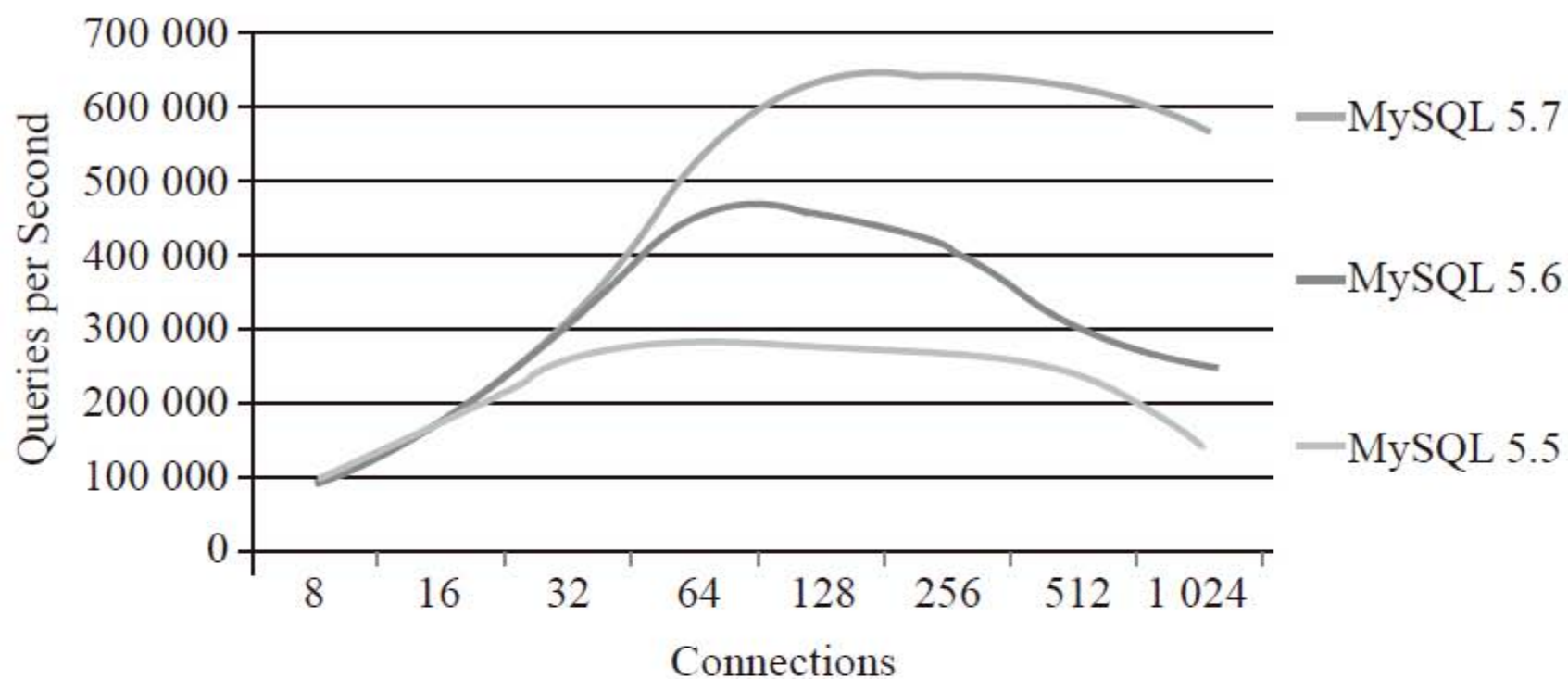


图2-2 sysbench读/写模式-每秒查询数

在多核CPU上，MySQL 5.7在72核上的表现优于MySQL 5.6的，如图2-3所示。

官方服务器的硬件配置如下。

- Intel(R)Xeon(R)CPU E7-8890 v3。
- 4 sockets x 18 cores-HT(144 CPU threads)。
- 2.5 Ghz , 512GB RAM。
- Linux kernel 3.16。

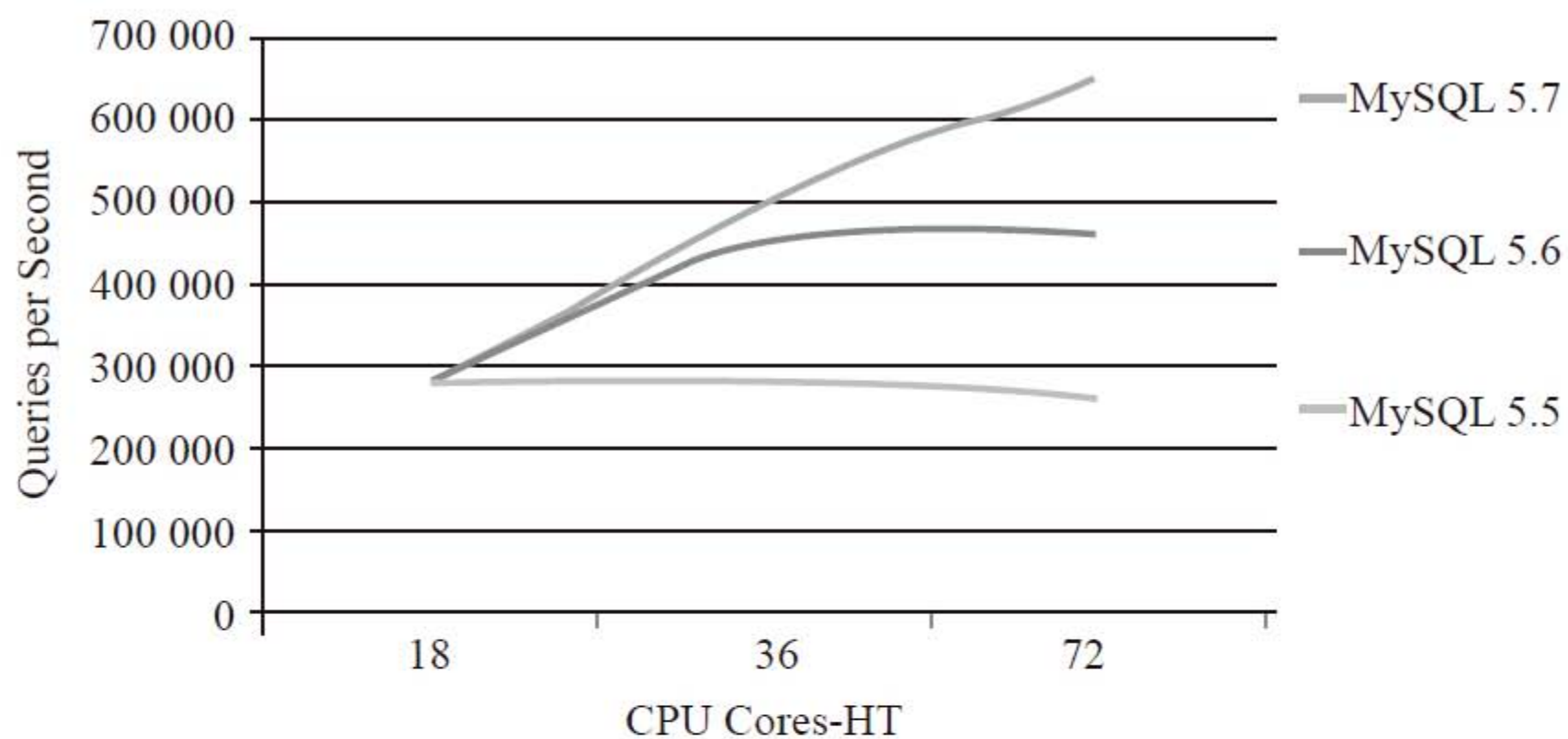


图2-3 多核CPU-每秒查询数

更详细的压测报告，感兴趣的读者可以访问官网，URL地址：<http://www.mysql.com/why-mysql/benchmarks/>。



## 2.2 安全性的提升

### 2.2.1 默认开启SSL

该功能只在MySQL 5.7和Percona5.7版本里支持。为了更容易地支持安全连接，MySQL 5.7在启动的时候，使用OpenSSL可以自动生成SSL和RSA证书和密钥文件。安全套接层（Secure Sockets Layer，SSL）及其继任者传输层安全（Transport Layer Security，TLS）是为网络通信提供安全及数据完整性的一种安全协议。TLS与SSL在传输层对网络连接进行加密，用以保障在Internet上数据传输之安全，利用数据加密（Encryption）技术，可确保数据在网络上之传输过程中不会被截取及窃听。

下面介绍MySQL 5.7的SSL配置与使用方式。

使用二进制包安装MySQL 5.7社区版时，下载地址为[http://dev.mysql.com/get/Downloads/MySQL-5.7/mysql-5.7.10-linux-glibc2.5-x86\\_64.tar.gz](http://dev.mysql.com/get/Downloads/MySQL-5.7/mysql-5.7.10-linux-glibc2.5-x86_64.tar.gz)。

安装过程如下：

```
shell>groupadd mysql
shell>useradd -r -g mysql -s /bin/false mysql
shell>cd /usr/local
shell>tar zxvf mysql-5.7.10-linux-glibc2.5-x86_64.tar.gz
shell>ln -s mysql-5.7.10-linux-glibc2.5-x86_64 mysql
shell>chown -R mysql:mysql mysql/
```

现在已经不用如下的mysql\_install\_db命令安装MySQL 5.7了。

```
shell>bin/mysql_install_db --user=mysql
```

已改为用如下的mysqld--initialize命令安装MySQL 5.7：

```
shell>bin/mysqld --initialize --user=mysql --basedir=/usr/local/mysql
--datadir=/data/mysql/
```

```
[root@slave1 mysql]# bin/mysqld --initialize --user=mysql --basedir=/usr/local/mysql --datadir=/data/mysql/
```

```
2016-01-16T07:53:30.603509Z 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Please use
```



```
--explicit_defaults_for_timestamp server option (see documentation for more details).
2016-01-16T07:53:32.411224Z 0 [Warning] InnoDB: New log files created ,
LSN=45790
2016-01-16T07:53:32.576227Z 0 [Warning] InnoDB: Creating foreign key constraint system tables.
2016-01-16T07:53:32.945296Z 0 [Warning] No existing UUID has been found , so we assume that this is the first time that this
server has been started. Generating a new UUID: 3d6401e5-bc26-11e5-8753-000c2966b647.
2016-01-16T07:53:32.980971Z 0 [Warning] Gtid table is not ready to be used. Table 'mysql.gtid_executed' cannot be opened.
2016-01-16T07:53:33.031378Z 1 [Note] A temporary password is generated for
root@localhost: iFv4pm2<bwQ9
2016-01-16T07:53:35.841432Z 1 [Warning] 'user' entry 'root@localhost' ignored in
--skip-name-resolve mode.
2016-01-16T07:53:35.841461Z 1 [Warning] 'user' entry 'mysql.sys@localhost'
ignored in --skip-name-resolve mode.
2016-01-16T07:53:35.841478Z 1 [Warning] 'db' entry 'sys mysql.sys@localhost'
ignored in --skip-name-resolve mode.
2016-01-16T07:53:35.841495Z 1 [Warning] 'proxies_priv' entry '@ root@localhost'
ignored in --skip-name-resolve mode.
2016-01-16T07:53:35.841534Z 1 [Warning] 'tables_priv' entry 'sys_config
mysql.sys@localhost' ignored in --skip-name-resolve mode.
```

请注意！在初始化安装的时候，会生成一个root用户随机的初始密码，请将其保存好，如下：

```
[Note] A temporary password is generated for root@localhost: iFv4pm2<bwQ9
```

通过如下命令开启SSL加密：

```
shell> bin/mysql_ssl_rsa_setup
```

在执行完mysql\_ssl\_rsa\_setup命令后，会在/data/目录下生成.pem后缀名文件，这就是SSL连接所需要的文件，如图2-4所示。

```
[root@master mysql]# ll *.pem
-rw----- 1 mysql mysql 1679 12月 14 17:08 ca-key.pem
-rw-r--r-- 1 mysql mysql 1074 12月 14 17:08 ca.pem
-rw-r--r-- 1 mysql mysql 1078 12月 14 17:08 client-cert.pem
-rw----- 1 mysql mysql 1675 12月 14 17:08 client-key.pem
-rw----- 1 mysql mysql 1675 12月 14 17:08 private_key.pem
-rw-r--r-- 1 mysql mysql 451 12月 14 17:08 public_key.pem
-rw-r--r-- 1 mysql mysql 1078 12月 14 17:08 server-cert.pem
-rw----- 1 mysql mysql 1675 12月 14 17:08 server-key.pem
[root@master mysql]#
```

图2-4 ssl公私钥文件

将数据目录属性修改为mysql，命令如下：

```
shell>chown -R mysql:mysql /data/
```

并执行下面的命令启动MySQL。

```
shell>bin/mysqld_safe --user=mysql &
```

首先要修改root密码，把之前系统生成的初始密码改掉，可使用命令SET PASSWORD=PASSWORD('123456');完成操作，代码如图2-5所示。

```
[root@slave1 ~]#
[root@slave1 ~]# mysql -uroot -p"iFv4pm2<bwQ9"
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.10-log

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SET PASSWORD = PASSWORD('123456');
Query OK, 0 rows affected, 1 warning (0.40 sec)
```

图2-5 修改初始密码

然后使用密码123456登录，如图2-6所示。



```
[root@slave1 ~]# mysql -uroot -p123456 -e "select version();"
mysql: [Warning] Using a password on the command line interface can be insecure.
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
[root@slave1 ~]#
```

图2-6 登录查询

至此，MySQL 5.7安装完毕。

如果你比较懒，不想每次都输入密码，那么在MySQL 5.6/5.7版本里提供了mysql\_config\_editor工具，是用于用户安全认证的，使用方法如下：

*mysql\_config\_editor set --login-path=client --host=localhost --user=root --password*

回车后输入root密码123456，如图2-7所示。

```
[root@slave1 ~]# mysql_config_editor set --login-path=client --host=localhost --user=root --password
Enter password:
[root@slave1 ~]#
[root@slave1 ~]# pwd
/root
[root@slave1 ~]# cat .mylogin.cnf
S?T[0-RBD...
```

图2-7 生成免密登录认证文件

此时，会在/root/目录下生成一个隐藏文件.mylogin.cnf，查看是乱码的。

然后就可以用mysql--login-path=client来进行免密登录了，如图2-8所示。



注意 在root用户密码变更后，需要重新执行mysql\_config\_editor，否则登录失败。

```
[root@slave1 ~]#
[root@slave1 ~]# mysql --login-path=client
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 5.7.10-log MySQL Community Server (GPL)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.09 sec)
```

图2-8 免密登录

如果觉得输入一长串参数还嫌麻烦，那么加入系统别名里好了，下次直接输入mm即可，如图2-9所示。

```
[root@slave1 ~]# vim .bashrc
[root@slave1 ~]#
[root@slave1 ~]# grep 'mysql' /root/.bashrc
alias mm='/usr/local/mysql/bin/mysql --login-path=client'
[root@slave1 ~]#
[root@slave1 ~]# source /root/.bash_profile
[root@slave1 ~]#
[root@slave1 ~]# mm -e "select version();"
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
```

图2-9 别名方式登录



注意 MariaDB 10.1版本没有这个工具。

当启动MySQL时，可以发现如图2-10所示的状态，代表已经正常工作。

创建用户的时候，需要指定该用户通过SSL连接：可通过命令GRANT ALL PRIVILEGES ON \*.\* TO 'ssluser' @ '%' IDENTIFIED BY '123456' REQUIRE SSL;实现，具体如图2-11所示。

然后使用ssluser用户登录，可通过命令mysql-h192.168.17.128-ussluser-p123456实现，具体如图2-12所示。

从图2-12可以看到，SSL已经显示加密了。MySQL 5.6/MariaDB 10.1同样支持以SSL的方式进行连接，但是操作相对MySQL 5.7较为复杂，用户需要自己通过openssl命令来创建各类公密钥。

```
mysql> show variables like '%ssl%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_openssl  | YES   |
| have_ssl      | YES   |
| ssl_ca        | ca.pem |
| ssl_capath    |       |
| ssl_cert      | server-cert.pem |
| ssl_cipher    |       |
| ssl_crl       |       |
| ssl_crlpath   |       |
| ssl_key       | server-key.pem |
+-----+-----+
9 rows in set (0.35 sec)

mysql> show variables like 'version%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| version       | 5.7.10-log |
| version_comment | MySQL Community Server (GPL) |
| version_compile_machine | x86_64 |
| version_compile_os | linux-glibc2.5 |
+-----+-----+
4 rows in set (0.01 sec)
```

图2-10 SSL已生效

```
mysql>
mysql> GRANT ALL PRIVILEGES ON *.* TO 'ssluser' @ '%' IDENTIFIED BY '123456' REQUIRE SSL;
Query OK, 0 rows affected, 1 warning (0.16 sec)
```

图2-11 授权SSL方式登录



```
mysql> \s
-----
mysql  Ver 14.14 Distrib 5.7.10, for linux-glibc2.5 (x86_64) using EditLine wrapper

Connection id:          33
Current database:
Current user:           ssluser@192.168.17.128
SSL:                    Cipher in use is DHE-RSA-AES256-SHA
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server version:         5.7.10-log MySQL Community Server (GPL)
Protocol version:       10
Connection:             192.168.17.128 via TCP/IP
Server characterset:    utf8
Db      characterset:   utf8
Client characterset:    utf8
Conn.  characterset:    utf8
TCP port:              3306
Uptime:                3 hours 8 min 57 sec
```

图2-12 SSL连接登录

## 2.2.2 不再明文显示用户密码

该功能只在MySQL 5.6/5.7和Percona 5.6/5.7版本里支持。MariaDB 10.1对于binlog中和用户密码相关的操作是不加密的。如果向MySQL发送类似create user, grant user...identified by这样携带初始明文密码的指令，那么会在binlog中原原本本地被还原出来。可通过下面的例子来验证。

MariaDB 10.1在命令行里设置用户密码，会在binlog里明文显示出来，如图2-13所示。

```
MariaDB [test]> grant all on *.* to 'hechunyang'@'%' identified by '123456';  
Query OK, 0 rows affected (0.01 sec)
```

```
MariaDB [test]> show binlog events;
```

Log_name	Pos	Event_type	Server_id	End_log_pos	Info
mysql-bin.000001	4	Format_desc	17129	249	Server ver: 10.1.10-MariaDB-enterprise-log, Binlog ver: 4
mysql-bin.000001	249	Gtid_list	17129	274	[]
mysql-bin.000001	274	Binlog_checkpoint	17129	313	mysql-bin.000001
mysql-bin.000001	313	Gtid	17129	351	GTID 0-17129-1
mysql-bin.000001	351	Query	17129	489	use `test`; grant all on *.* to 'hechunyang'@'%' identified by '123456'

```
5 rows in set (0.06 sec)
```

图2-13 binlog里显示明文密码

MySQL 5.7在命令行里设置用户密码，会在binlog里对明文密码做加密处理。

```
mysql> grant all on *.* to 'hechunyang'@'%' identified by '123456';
Query OK, 0 rows affected, 1 warning (0.12 sec)
```

```
mysql> show binlog events;
```

Log_name	Pos	Event_type	Server_id	End_log_pos	Info
mysql-bin.000001	4	Format_desc	128	123	Server ver: 5.7.10-log, Binlog ver: 4
mysql-bin.000001	123	Previous_gtid	128	154	
mysql-bin.000001	154	Anonymous_Gtid	128	219	SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
mysql-bin.000001	219	Query	128	458	use `information_schema`; GRANT ALL PRIVILEGES ON *.* TO 'hechunyang'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9'

4 rows in set (0.00 sec)

图2-14 binlog里对明文密码加密



### 2.2.3 sql\_mode的改变

MySQL 5.7/MariaDB 10.1默认启用STRICT\_TRANS\_TABLES严格模式。该模式为严格模式，用于进行数据的严格校验，错误数据不能插入，报error（错误），并且事务回滚。

在MySQL 5.6和MariaDB 10.0中，默认SQL\_MODE模式为空，表结构如图2-15所示。

设置sql\_mode模式为空，然后插入数据，如图2-16所示。

由于age字段是int数值整形，因此在插入字符类型时会发出警告，但由于sql\_mode模式为空，故数据仍可以插入，如图2-17所示。

设置sql\_mode模式为STRICT\_TRANS\_TABLES，然后插入数据，如图2-18所示。

由于age字段是int数值整形，因此在插入字符类型时发出警告，并且事务已经回滚，如图2-19所示。

```
mysql> desc t1;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	
name	varchar(2)	YES		NULL	
age	smallint(6)	YES		NULL	

```
3 rows in set (0.01 sec)
```

图2-15 t1表结构

```
mysql> set sql_mode = '';
```

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

```
mysql> insert into t1 values(1,'aa','aaa');
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
mysql> show warnings;
```

Level	Code	Message
Warning	1366	Incorrect integer value: 'aaa' for column 'age' at row 1

```
1 row in set (0.00 sec)
```

图2-16 设置sql\_mode模式为空

```
mysql> insert into t1 values(1,'aa','aaa');
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
mysql> show warnings;
```

Level	Code	Message
Warning	1366	Incorrect integer value: 'aaa' for column 'age' at row 1

```
1 row in set (0.00 sec)
```

```
mysql> select * from t1;
```

id	name	age
1	aa	0

```
1 row in set (0.00 sec)
```

图2-17 插入成功

```
mysql> set sql_mode = 'STRICT_TRANS_TABLES';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
mysql> insert into t1 values(2,'bb','bbb');
ERROR 1366 (HY000): Incorrect integer value: 'bbb' for column 'age' at row 1
mysql>
```

图2-18 插入失败

```
mysql> insert into t1 values(2,'bb','bbb');
ERROR 1366 (HY000): Incorrect integer value: 'bbb' for column 'age' at row 1
mysql>
mysql> select * from t1 where id = 2;
Empty set (0.21 sec)
```

```
mysql> █
```

图2-19 事务回滚

## 2.3 InnoDB存储引擎的提升

### 2.3.1 更改索引名字时不会锁表

该功能只在MySQL 5.7和Percona 5.7版本里支持。这个功能有些鸡肋的感觉，实用性不高，在生产环境上修改索引名字的情况极少发生，其表结构如图2-20所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.37 sec)

mysql> show create table sbtest\G;
***** 1. row *****
      Table: sbtest
Create Table: CREATE TABLE `sbtest` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `k` int(10) unsigned NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` varchar(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k` (`k`)
) ENGINE=InnoDB AUTO_INCREMENT=1000001 DEFAULT CHARSET=utf8
1 row in set (0.25 sec)

ERROR:
No query specified
```

图2-20 sbtest表结构

现在把索引k修改为idx\_k，执行命令如下：  
*alter table sbtest rename index k to idx\_k;*

具体如图2-21所示。

从图2-21可以看到，索引名字已经变更。





注意 MariaDB 10.1并不支持该功能。

```
mysql> alter table sbtest rename index k to idx_k;
Query OK, 0 rows affected (0.26 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> show create table sbtest\G;
***** 1. row *****
      Table: sbtest
Create Table: CREATE TABLE `sbtest` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `k` int(10) unsigned NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` varchar(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `idx_k` (`k`)
) ENGINE=InnoDB AUTO_INCREMENT=1000001 DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-21 修改索引名字

### 2.3.2 在线DDL修改varchar字段属性时不锁表

MySQL 5.7针对DDL功能做了加强，修改varchar字段已不锁表，测试过程如图2-22和图2-23所示。

```
mysql> show create table sbtest\G;
***** 1. row *****
      Table: sbtest
Create Table: CREATE TABLE `sbtest` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `k` int(10) unsigned NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` varchar(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `idx_k` (`k`)
) ENGINE=InnoDB AUTO_INCREMENT=1000001 DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-22 sbtest表结构

```
mysql> select count(*) from sbtest;
+-----+
| count(*) |
+-----+
| 1000000 |
+-----+
1 row in set (5.36 sec)
```

图2-23 表的行数

100万行的表，现在要把pad varchar(60)修改为pad varchar(80)，可以看到，既不用拷贝数据也不用锁表，不到1秒即执行完毕，如图2-24所示。

ALGORITHM用于指定创建或删除索引的算法；COPY表示按照MySQL 5.1版本之前的方法，即创建临时表并全表拷贝数据，原表加全局读锁；INPLACE表示创建字段或删除字段操作不需要创建临时表；DEFAULT表示是通过INPLACE的算法还是COPY的算法。

```
mysql> select count(*) from sbtest;
+-----+
| count(*) |
+-----+
| 1000000 |
+-----+
1 row in set (4.99 sec)

mysql> alter table sbtest ALGORITHM=INPLACE, modify pad varchar(80) not null;
Query OK, 0 rows affected (0.63 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> desc sbtest;
+-----+-----+-----+-----+-----+-----+
| Field | Type                | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id    | int(10) unsigned    | NO   | PRI | NULL    | auto_increment |
| k     | int(10) unsigned    | NO   | MUL | 0       |                |
| c     | char(120)           | NO   |     |         |                |
| pad   | varchar(80)         | NO   |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.16 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.10 sec)
```

图2-24 修改表结构耗时

如果字段属性大于并等于varchar(256)（这里的256是指字节（UTF8占用3字节）或者把varchar(80)减少到varchar(40)），则仍需要拷贝数据且锁表。并且只对varchar类型做了优化，char和int是无效的，分别如图2-25和图2-26所示。



```
mysql> alter table sbtest ALGORITHM=INPLACE, modify pad varchar(256) not null;
ERROR 1846 (0A000): ALGORITHM=INPLACE is not supported. Reason: Cannot change column type INPLACE. Try ALGORITHM=COPY.
mysql>
mysql> alter table sbtest ALGORITHM=INPLACE, modify pad varchar(100) not null;
ERROR 1846 (0A000): ALGORITHM=INPLACE is not supported. Reason: Cannot change column type INPLACE. Try ALGORITHM=COPY.
mysql>
mysql> alter table sbtest ALGORITHM=INPLACE, modify pad char(60) not null;
ERROR 1846 (0A000): ALGORITHM=INPLACE is not supported. Reason: Cannot change column type INPLACE. Try ALGORITHM=COPY.
mysql>
mysql> alter table sbtest ALGORITHM=INPLACE, modify pad int(11) not null;
ERROR 1846 (0A000): ALGORITHM=INPLACE is not supported. Reason: Cannot change column type INPLACE. Try ALGORITHM=COPY.
mysql>
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)
```

图2-25 MySQL 5.7修改字段varchar测试



注意 MariaDB 10.1并不支持对varchar修改表结构的优化。

Online DDL的原理：在执行创建或删除操作的同时，将INSERT/UPDATE/DELETE这类DML操作日志写入一个缓存中。待完成修改表结果后再重做应用到原表上，以此达到数据的一致性。这个缓存大小由innodb\_online\_alter\_log\_max\_size控制，默认为128MB，若用户更新的表比较频繁，在创建过程中伴有大量的写事务，如遇到innodb\_online\_alter\_log\_max\_size的空间不能存放日志，则会抛出错误。对于这个操作，可以调大参数innodb\_online\_alter\_log\_max\_size，一次获得更大的日志缓存空间。

```
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> alter table sbtest ALGORITHM=INPLACE, modify pad varchar(40) not null;
ERROR 1846 (0A000): ALGORITHM=INPLACE is not supported. Reason: Cannot change column type INPLACE. Try ALGORITHM=COPY.
MariaDB [test]>
```

图2-26 MariaDB 10.1修改字段varchar测试

下面重现报错过程步骤：把日志缓存innodb\_online\_alter\_log\_max\_size值调到最小值1，然后添加一个新的name字段，这时，在会话二执行update sbtest set c=' hechunyang' ;时就会出现报错信息，从结果上来看，表结构修改并未成功，如图2-27所示。

```
mysql>
mysql> alter table sbtest add name varchar(100);
ERROR 1799 (HY000): Creating index 'PRIMARY' required more than 'innodb_online_alter_log_max_size' bytes of modification log. Please try again.
mysql>
mysql> desc sbtest;
```

Field	Type	Null	Key	Default	Extra
id	int(10) unsigned	NO	PRI	NULL	auto_increment
k	int(10) unsigned	NO	MUL	0	
c	char(200)	NO		NULL	
pad	varchar(80)	NO		NULL	

```
4 rows in set (0.65 sec)
```

图2-27 修改表结构失败



注意 若临时表的大小超出此上限，则ALTER表的操作会失败，当前所有未提交的DML操作会回滚。因此，较大的值允许在线DDL操作期间有更多的DML被执行，但是过大的值会使DDL操作结束后，应用innodb\_online\_alter\_log日志中的数据会花很长的时间。修改表结构操作，尤其是对一张大表，建议在凌晨业务低峰期执行，另外在执行Online DDL的过程中，数据库吞吐量会下降。

综上所述，Online DDL这个名字很容易误导新手，因为不论什么情况下，修改表结构都不会锁表，理想很丰满，现实很骨感，注意这个



坑！如果开发人员让修改表字段属性，那么建议使用pt-online-schema-change。另外，增加、删除字段或索引不锁全表，删除主键锁全表，如图2-28所示。

当添加字段alter table表时，对该表的增、删、改、查均不会锁表。而在这之前，当该表被访问时，需要等其执行完毕后可以执行alter table，例如在会话一，故意执行一条大结果的查询，然后在会话二执行增加字段name，此时还会出现表锁，如图2-29所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)

mysql> show processlist;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host       | db  | Command | Time | State                | Info                                     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 42 | root | localhost | NULL | Query    | 0    | starting             | show processlist                     |
| 43 | root | localhost | test | Query    | 6    | copy to tmp table    | alter table sbtest modify c char(200) not null |
| 44 | root | localhost | test | Query    | 3    | Waiting for table metadata lock | update sbtest set k='hechunyang' where id = 10 |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.54 sec)
```

图2-28 DDL锁表重现



```
mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host      | db  | Command | Time | State                | Info                                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 14 | root | localhost | test | Query    | 4    | Sending data         | select * from sbtest              |
| 15 | root | localhost | NULL | Query    | 0    | starting             | show processlist                  |
| 18 | root | localhost | test | Query    | 2    | Waiting for table metadata lock | alter table sbtest add name char(10) |
+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.29 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (1.26 sec)
```

图2-29 DDL锁表重现

因此，在凌晨上线时，一定要执行show processlist命令并观察，此刻是否有某个慢SQL对该表进行操作，以免alter table表时出现表锁现象。

### 2.3.3 InnoDB/MyisAM存储引擎支持中文全文索引

该功能只在MySQL 5.7和Percona 5.7版本里支持。之前的版本，全文索引对中文支持不友好，所以更多的公司选择用第三方开源软件Sphinx做全文索引。



注意 Sphinx是一个基于SQL的全文检索引擎，可以结合MySQL、PostgreSQL做全文搜索，它可以提供比数据库本身更专业的搜索功能，让应用程序更容易实现专业化的全文检索。Sphinx特别为一些脚本语言设计搜索API接口，如PHP、Python、Perl、Ruby等，同时为MySQL设计了一个存储引擎插件。

如今在MySQL 5.7版本里，提供了一个内置全文索引支持中文的ngram解析器插件，MariaDB 10.1并不提供此插件。它的工作原理为：例“生日快乐”四个字，当参数设置为ngram\_token\_size=2（默认两个中文单词）时，那么ngram解释器将解释为“生日”“快乐”，ngram\_token\_size参数不可动态修改，所以事先要规划好以几个单词作为搜索条件，避免搜索的结果不准确。

下面对该功能进行测试。这里以三个单词为准，即ngram\_token\_size=3（加入my.cnf配置里），其表结构如图2-30所示。

```
mysql> show create table articles\G;
***** 1. row *****
      Table: articles
Create Table: CREATE TABLE `articles` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `title` varchar(200) DEFAULT NULL,
  `body` text,
  PRIMARY KEY (`id`),
  FULLTEXT KEY `title` (`title`,`body`) /*!50100 WITH PARSER `ngram` */
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8
1 row in set (0.09 sec)

ERROR:
No query specified

mysql> show variables like 'ng%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| ngram_token_size | 3 |
+-----+-----+
1 row in set (0.57 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.10 sec)
```

图2-30 以字段title和body建立联合全文索引



注意 使用ngram解释器，这些参数将失效：innodb\_ft\_min\_token\_size、innodb\_ft\_max\_token\_size、ft\_min\_word\_len和ft\_max\_word\_len。

然后插入测试数据，如图2-31所示。



```
mysql> INSERT INTO articles (title,body) VALUES
->      ('数据库管理','在本教程中我将向你展示如何管理数据库'),
->      ('数据库应用开发','学习开发数据库应用程序');
Query OK, 2 rows affected (0.16 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from articles;
+----+-----+-----+
| id | title                | body                                     |
+----+-----+-----+
|  1 | 数据库管理           | 在本教程中我将向你展示如何管理数据库 |
|  2 | 数据库应用开发       | 学习开发数据库应用程序               |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图2-31 插入数据

现在用如下命令查询数据：

*select \* from articles where MATCH(title , body) AGAINST ('数据库' IN BOOLEAN MODE);*  
 查询后的结果如图2-32所示。

```
mysql> explain select * from articles where MATCH(title,body) AGAINST ('数据库' IN BOOLEAN MODE);
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  1 | SIMPLE      | articles | NULL       | fulltext | title         | title | 0       | const |    1 |   100.00 | Using where; Ft_hints: no_ranking |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.09 sec)

mysql>
mysql> select * from articles where MATCH(title,body) AGAINST ('数据库' IN BOOLEAN MODE);
+----+-----+-----+
| id | title                | body                                     |
+----+-----+-----+
|  1 | 数据库管理           | 在本教程中我将向你展示如何管理数据库 |
|  2 | 数据库应用开发       | 学习开发数据库应用程序               |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图2-32 使用索引全文查询数据

通过执行计划器可以看到已经利用上了title这个索引，并且只需要扫描一行即可查出结果。

但这里的停止词 (stopwords) 设计有一个缺陷。停止词就是指不想让用户在搜索的时候能搜到“李洪志大师”“法轮大法”等词汇，需要事先定义好停止词，这样就不会被搜索到。前面说到的设计缺陷是指你必须事先就定义好，假如日后还想再定义停止词“活摘器官”，则需要重建一次全文索引，如果表很大，那么会很费时间。

下面看例子。

首先定义一张停止词 (stopwords) 表用来保存敏感词汇，表结构如图2-33所示。

```
mysql> show create table my_stopwords\G;
***** 1. row *****
      Table: my_stopwords
Create Table: CREATE TABLE `my_stopwords` (
  `value` varchar(30) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-33 停止词表结构

然后往这个停止词表里插入数据库词汇，并且执行如下语句使其生效。

```
SET GLOBAL innodb_ft_server_stopword_table = 'test/my_stopwords';
```

结果如图2-34所示。

```
mysql> select * from my_stopwords;
+-----+
| value |
+-----+
| 数据库 |
+-----+
1 row in set (0.01 sec)

mysql> SET GLOBAL innodb_ft_server_stopword_table = 'test/my_stopwords';
SET innodb_ft_server_stopword_table = 'test/my_stopwords'; Query OK, 0 rows affected (0.37 sec)
```

图2-34 设置停止词访问的表

再对其中的参数进行说明，这里的test是数据库；my\_stopwords是表。



注意 停止词 (stopwords) 表要和主表放在同一个数据库Schema下。

新开启一个会话终端，再次执行查询语句：

```
select * from articles where MATCH(title, body) AGAINST ('数据库' IN BOOLEAN MODE);
```



但这里的停止词 ( stopwords ) 设计有一个缺陷。停止词就是指不想让用户在搜索的时候能搜到 “李洪志大师” “法轮大法” 等词汇，需要事先定义好停止词，这样就不会被搜索到。前面说到的设计缺陷是指你必须事先就定义好，假如日后还想再定义停止词 “活摘器官”，则需要重建一次全文索引，如果表很大，那么会很费时间。

下面看例子。

首先定义一张停止词 ( stopwords ) 表用来保存敏感词汇，表结构如图2-33所示。

```
mysql> show create table my_stopwords\G;
***** 1. row *****
      Table: my_stopwords
Create Table: CREATE TABLE `my_stopwords` (
  `value` varchar(30) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-33 停止词表结构

然后往这个停止词表里插入数据库词汇，并且执行如下语句使其生效。

*SET GLOBAL innodb\_ft\_server\_stopword\_table = 'test/my\_stopwords';*

结果如图2-34所示。

```
mysql> select * from my_stopwords;
+-----+
| value |
+-----+
| 数据库 |
+-----+
1 row in set (0.01 sec)

mysql> SET GLOBAL innodb_ft_server_stopword_table = 'test/my_stopwords';
SET innodb_ft_server_stopword_table = 'test/my_stopwords'; Query OK, 0 rows affected (0.37 sec)
```

图2-34 设置停止词访问的表

再对其中的参数进行说明，这里的test是数据库；my\_stopwords是表。



注意 停止词 ( stopwords ) 表要和主表放在同一个数据库Schema下。

新开启一个会话终端，再次执行查询语句：

*select \* from articles where MATCH(title, body) AGAINST ('数据库' IN BOOLEAN MODE);*



会发现停止词 ( stopwords ) 并没有生效，如图2-35所示。

```
mysql> show variables like 'innodb_ft_server_stopword_table';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_ft_server_stopword_table | test/my_stopwords |
+-----+-----+
1 row in set (0.35 sec)

mysql> select * from articles where MATCH(title,body) AGAINST ('数据库' IN BOOLEAN MODE);
+----+-----+-----+
| id | title | body |
+----+-----+-----+
| 1 | 数据库管理 | 在本教程中我将向你展示如何管理数据库 |
| 2 | 数据库应用开发 | 学习开发数据库应用程序 |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图2-35 停止词未生效

这时必须重建一次全文索引，使其生效，如图2-36所示。

```
mysql> alter table articles engine = innodb;
Query OK, 2 rows affected (0.55 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> show variables like 'innodb_ft_server_stopword_table';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_ft_server_stopword_table | test/my_stopwords |
+-----+-----+
1 row in set (0.01 sec)

mysql> select * from articles where MATCH(title,body) AGAINST ('数据库' IN BOOLEAN MODE);
Empty set (0.00 sec)
```

图2-36 重新建立全文索引

这样的操作已经查询不到关键字为“数据库”的记录了。

结论：截至MySQL 5.7.10版本，此缺陷并未解决，希望在后面的版本里可以进行调整。

### 2.3.4 InnoDB Buffer Pool预热改进

该功能只支持MySQL 5.7和Percona 5.7版本。当数据库重启时，会面临一个问题，即如何将之前频繁访问的数据重新加载回Buffer中？也就是说，如何对InnoDB Buffer Pool进行预热，以便于快速恢复到之前的性能状态。如果仅靠InnoDB本身去预热Buffer，将会是一个不短的时间周期，在业务高峰时，数据库将面临相当大的考验，I/O的瓶颈会带来糟糕的性能。

在MySQL 5.6/MariaDB 10.0版本里，为了解决上述问题，提供了一个新特性来快速预热Buffer\_Pool缓冲池。只需在my.cnf里加入如下命令即可：

```
innodb_buffer_pool_dump_at_shutdown = 1
```

该命令用于在关闭时把热数据dump到本地磁盘。

以下命令为采用手工方式把热数据dump到本地磁盘。

```
innodb_buffer_pool_dump_now = 1
```

以下命令为在启动时把热数据加载到内存。

```
innodb_buffer_pool_load_at_startup = 1
```

以下命令为采用手工方式把热数据加载到内存。

```
innodb_buffer_pool_load_now = 1
```

关闭MySQL时，会把内存中的热数据保存在磁盘的ib\_buffer\_pool文件中，该文件位于数据目录下，如图2-37所示。

```
120723 17:47:41 InnoDB: Dumping buffer pool(s) to /usr/local/mysql/data/ib_buffer_pool
120723 17:47:41 InnoDB: Buffer pool(s) dump completed at 120723 17:47:41
```

图2-37 导出热数据到本地文件

启动MySQL时，会自动加载热数据到Buffer\_Pool缓冲池里，这样，机器重启后热数据会始终保存在内存中，如图2-38所示。

```
120723 17:49:14 InnoDB: Loading buffer pool(s) from /usr/local/mysql/data/ib_buffer_pool
```

图2-38 数据加载到内存



注意 只有正常关闭MySQL服务或pkill mysql时，才会把热数据dump到内存。机器宕机或者pkill -9 mysql，是不会dump的。



在MySQL 5.7里，一个新的系统参数，`innodb_buffer_pool_dump_pct`，允许DBA控制每个缓冲池最近使用页的百分比来导出，以减缓导出InnoDB Buffer Pool所有页占用过大的磁盘I/O，默认值是25，如果InnoDB Buffer Pool里缓存了100个页，那么会将最近使用的25个页导出，最大值是100，意思为全部导出，如图2-39所示。

• `innodb_buffer_pool_dump_pct`

Introduced	5.7.2	
Command-Line Format	<code>--innodb_buffer_pool_dump_pct=#</code>	
System Variable	Name	<code>innodb_buffer_pool_dump_pct</code>
	Variable Scope	Global
	Dynamic Variable	Yes
Permitted Values (<= 5.7.6)	Type	integer
	Default	100
	Min Value	1
	Max Value	100
Permitted Values (>= 5.7.7)	Type	integer
	Default	25
	Min Value	1
	Max Value	100

图2-39 `innodb_buffer_pool_dump_pct`参数



注意 在MariaDB 10.1里，同样提供了`innodb_buffer_pool_dump_pct`参数，默认值是100。



### 2.3.5 在线调整innodb\_Buffer\_Pool\_Size不用重启mysql进程

在MySQL 5.6/MariaDB 10.1版本里，调整innodb\_buffer\_pool\_size大小必须重启mysql进程才可以生效，而在MySQL 5.7版本里，可以直接动态设置，方便了很多。

这个功能应用的场景如下：

- 机器增加内存，DBA粗心大意忘记调大innodb\_buffer\_pool\_size了。
- 工作交接，新来的DBA发现前任DBA设置的innodb\_buffer\_pool\_size不合理。

需要注意的地方，在调整Buffer\_Pool期间，用户的请求会阻塞，直到调整完毕，所以请勿在白天调整，可在凌晨3~4点低峰期调整。

下面进行一项功能测试，现在把innodb\_buffer\_pool\_size改为256MB（见图2-40）。

```
mysql> show variables like 'innodb_buffer_pool_size';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_buffer_pool_size | 134217728 |
+-----+-----+
1 row in set (0.48 sec)

mysql> set global innodb_buffer_pool_size = 256*1024*1024;
Query OK, 0 rows affected (0.27 sec)

mysql> show variables like 'innodb_buffer_pool_size';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_buffer_pool_size | 268435456 |
+-----+-----+
1 row in set (0.25 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.04 sec)
```

图2-40 在线动态修改innodb\_buffer\_pool\_size为256MB

调整时，内部会把数据页移动到一个新的位置，单位是块。如果想提升移动的速度，则需要调整innodb\_buffer\_pool\_chunk\_size参数的大小，默认是128MB。

innodb\_buffer\_pool\_chunk\_size大小的计算公式是： $\text{innodb\_buffer\_pool\_size} / \text{innodb\_buffer\_pool\_instances}$

若innodb\_buffer\_pool\_size为2GB，innodb\_buffer\_pool\_instances实例为4个，现在要将innodb\_buffer\_pool\_chunk\_size改为1GB，那么系统会自动把innodb\_buffer\_pool\_chunk\_size调整为512MB。

监控InnoDB\_buffer\_pool的调整进程，结果如图2-41所示。

```
mysql> SHOW STATUS WHERE Variable_name='InnoDB_buffer_pool_resize_status';
```

Variable_name	Value
InnoDB_buffer_pool_resize_status	Completed resizing buffer pool at 160119 15:52:16.

```
1 row in set (0.43 sec)
```

图2-41 监控innodb\_buffer\_pool的调整过程

查看错误日志，如图2-42所示。

```
[root@master mysql]# tail -f mysqld.log
2016-01-19T07:52:15.862683Z 0 [Note] InnoDB: Disabling adaptive hash index.
2016-01-19T07:52:15.855444Z 2 [Note] InnoDB: Resizing buffer pool from 134217728 to 268435456 (unit=134217728). (new size: 268435456 bytes)
2016-01-19T07:52:15.895288Z 0 [Note] InnoDB: disabled adaptive hash index.
2016-01-19T07:52:15.895339Z 0 [Note] InnoDB: Withdrawing blocks to be shrunken.
2016-01-19T07:52:15.895351Z 0 [Note] InnoDB: Latching whole of buffer pool.
2016-01-19T07:52:15.908703Z 0 [Note] InnoDB: buffer pool 0 : resizing with chunks 1 to 2.
2016-01-19T07:52:15.977615Z 0 [Note] InnoDB: buffer pool 0 : 1 chunks (8192 blocks) were added.
2016-01-19T07:52:15.980418Z 0 [Note] InnoDB: Completed to resize buffer pool from 134217728 to 268435456.
2016-01-19T07:52:16.023970Z 0 [Note] InnoDB: Re-enabled adaptive hash index.
2016-01-19T07:52:16.024027Z 0 [Note] InnoDB: Completed resizing buffer pool at 160119 15:52:16.
```

图2-42 innodb\_buffer\_pool\_size调整完毕



### 2.3.6 回收（收缩）undo log回滚日志物理文件空间

undo log回滚日志是保存在共享表空间ibdata1文件里的，随着业务的不停运转，ibdata1文件会越来越大，想要回收（收缩空间大小）极其困难和复杂，必须先将mysqldump-A全库数据导出，然后删掉data目录，之后重新初始化安装，最后再把全库的SQL文件导入，这样才可实现ibdata1文件的回收。

在MySQL 5.6版本里，可以把undo log回滚日志分离到一个单独的表空间里；其缺点是不能回收（收缩）空间大小，直到MySQL 5.7版本才支持在线收缩。不过，MariaDB 10.1暂不支持此功能。



注意 安装MySQL时，需要在my.cnf里指定如下参数；否则，等创建数据库以后再指定，就会报错。

·innodb\_undo\_directory=/data2/（指定存放的目录，默认是数据目录）。

·innodb\_undo\_logs=128（指定回滚段128KB）。

·innodb\_undo\_tablespaces=4（指定有多少个undo log文件）。

然后启动MySQL，会出现如图2-43所示的情况。

undo log创建好后，就不能再次修改或增加。

下面对该功能进行测试。

```
-rw-rw---- 1 mysql mysql 10485760 08-26 00:54 undo001
-rw-rw---- 1 mysql mysql 10485760 08-26 00:54 undo002
-rw-rw---- 1 mysql mysql 10485760 08-26 00:54 undo003
-rw-rw---- 1 mysql mysql 10485760 08-26 00:51 undo004
```

图2-43 创建undo log空间

把Undo Log从共享表空间里ibdata1拆分出去，注意在安装MySQL时，需要在my.cnf里指定，否则等数据库已经启动后再指定，就会报错。

下面是涉及的相关参数的解释。

·innodb\_undo\_log\_truncate参数设置为1，即开启在线回收（收缩）undo log日志文件，支持动态设置。

·innodb\_undo\_tablespaces参数必须大于或等于2，即回收（收缩）一个undo log日志文件时，要保证另一个undo log是可用的。

·innodb\_undo\_logs:undo回滚段的数量，至少大于等于35，默认为128。





- innodb\_max\_undo\_log\_size：当超过这个阈值（默认是1GB）时，会触发truncate回收（收缩）动作，truncate后空间缩小到10MB。
- innodb\_purge\_rseg\_truncate\_frequency：控制回收（收缩）undo log的频率。undo log空间在它的回滚段没有得到释放之前不会收缩，想要增加释放回滚区间的频率，就得降低innodb\_purge\_rseg\_truncate\_frequency设定值。

现在对一张有100万行代码的sbtest表进行全表更新4次，如图2-44所示。

然后你会发现undo log空间急速增长，当超过innodb\_max\_undo\_log\_size阈值（默认是1GB）时，会触发truncate回收（收缩）动作，truncate后空间缩小到10MB，如图2-45所示。

```
mysql> update sbtest set c='1';
Query OK, 1000000 rows affected (1 min 1.51 sec)
Rows matched: 1000000  Changed: 1000000  Warnings: 0

mysql> update sbtest set c='2';
Query OK, 1000000 rows affected (59.16 sec)
Rows matched: 1000000  Changed: 1000000  Warnings: 0

mysql> update sbtest set c='3';
Query OK, 1000000 rows affected (59.27 sec)
Rows matched: 1000000  Changed: 1000000  Warnings: 0

mysql> update sbtest set c='4';
Query OK, 1000000 rows affected (1 min 3.37 sec)
Rows matched: 1000000  Changed: 1000000  Warnings: 0
```

图2-44 更新数据

```
2014-11-10T07:31:49.044734Z 0 [Note] InnoDB: Truncating UNDO tablespace with space identifier 2
2014-11-10T07:31:52.509021Z 0 [Note] InnoDB: Completed truncate of UNDO tablespace with space identifier 2
2014-11-10T07:32:07.000002Z 0 [Note] InnoDB: Truncating UNDO tablespace with space identifier 3
2014-11-10T07:32:11.985967Z 0 [Note] InnoDB: Completed truncate of UNDO tablespace with space identifier 3
2014-11-10T07:32:18.745123Z 0 [Note] InnoDB: Truncating UNDO tablespace with space identifier 4
2014-11-10T07:32:22.770921Z 0 [Note] InnoDB: Completed truncate of UNDO tablespace with space identifier 4
2014-11-10T07:32:27.290667Z 0 [Note] InnoDB: Truncating UNDO tablespace with space identifier 1
2014-11-10T07:32:29.818926Z 0 [Note] InnoDB: Completed truncate of UNDO tablespace with space identifier 1
```

图2-45 清空undo log空间

通过观察物理文件，undo log空间已经被回收了，默认为10MB，如图2-46所示。

```
-rw-rw---- 1 mysql mysql 10M 11? 10 15:34 undo001
-rw-rw---- 1 mysql mysql 10M 11? 10 15:34 undo002
-rw-rw---- 1 mysql mysql 10M 11? 10 15:34 undo003
-rw-rw---- 1 mysql mysql 10M 11? 10 15:34 undo004
```

图2-46 undo log空间释放

这个功能出来以后，降低了磁盘空间的使用率，并且加快了xtrabackup热备份的速度。

### 2.3.7 InnoDB提供通用表空间

只在MySQL 5.7和Percona 5.7版本里支持通用表空间（General Tablespaces）这项功能。这项功能类似共享表空间。共享表空间是把所有表都存放在ibdata1文件里，而通用表空间（可以理解为加强版的独立表空间，需要开启参数innodb\_file\_per\_table=1）可以指定某些表存放在同一个share.ibd文件里，如图2-47所示。

```
mysql>
mysql> CREATE TABLESPACE `share` ADD DATAFILE 'share.ibd' Engine=InnoDB;
Query OK, 0 rows affected (0.41 sec)

mysql> CREATE TABLE hcy1 (id INT PRIMARY KEY) TABLESPACE share;
Query OK, 0 rows affected (0.22 sec)

mysql> CREATE TABLE hcy2 (id INT PRIMARY KEY) TABLESPACE share;
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TABLE hcy3 (id INT PRIMARY KEY) TABLESPACE share;
Query OK, 0 rows affected (0.04 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.39 sec)
```

图2-47 创建通用表空间



注意 share.ibd和ibdata1放在同一个目录下，MariaDB 10.1暂时不提供该功能。



### 2.3.8 创建InnoDB独立表空间指定存放路径

只有在MySQL 5.6/5.7、Percona 5.6/5.7和MariaDB 10.0/10.1版本里支持独立表空间这项功能。在MySQL 5.5版本中，采用独立表空间(.ibd)存放数据时，不能更改路径，比如磁盘空间满了，恰巧没做LVM卷组，那么通过CREATE TABLE t2(id int primary key)engine=innodb DATA DIRECTORY="/data2/" 就能把创建t2表的.ibd放到/data2/目录下，如图2-48所示。

```
MariaDB [test]> show tables;
Empty set (0.00 sec)

MariaDB [test]> create table t1(id int primary key);
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> CREATE TABLE t2(id int primary key)engine=innodb DATA DIRECTORY="/data2/";
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)
```

图2-48 指定表存储路径

此时查看数据目录，会发现多了一个t2.isl文件，可以理解为软连接，如图2-49所示。

然后在/data2/目录下可以看到一个t2.ibd数据文件，如图2-50所示。

```
[root@dbbak test]# pwd
/data/mariadb101/test
[root@dbbak test]#
[root@dbbak test]#
[root@dbbak test]# ll -h
总用量 144K
-rw-rw----. 1 mysql mysql 61 1月 25 14:13 db.opt
-rw-rw----. 1 mysql mysql 922 1月 25 14:22 t1.frm
-rw-rw----. 1 mysql mysql 96K 1月 25 14:22 t1.ibd
-rw-rw----. 1 mysql mysql 922 1月 25 14:23 t2.frm
-rw-rw----. 1 mysql mysql 18 1月 25 14:23 t2.isl
[root@dbbak test]#
```

图2-49 查看数据目录

```
[root@dbbak test]# pwd
/data2/test
[root@dbbak test]#
[root@dbbak test]# ll -h
总用量 96K
-rw-rw----. 1 mysql mysql 96K 1月 25 14:23 t2.ibd
[root@dbbak test]#
```

图2-50 查看数据文件

### 2.3.9 迁移单独一张InnoDB表到远程服务器

只有在MySQL 5.6/5.7、Percona 5.6/5.7和MariaDB 10.0/10.1版本里支持迁移单独一张InnoDB表到远程服务器这项功能。我们知道，MyISAM引擎可以单独把\*.MYD和\*.MYI拷贝到远程服务器上，但如果是InnoDB引擎，则受限于版本（MySQL5.5），直接拷贝.ibd到远程服务器是不行的。因为在ibdata1文件里保存有表的字典信息，在ibd文件里保存有事务ID和日志序列号，这些对于服务器来说都是不一样的。所以在MySQL 5.5里，针对InnoDB表迁移只能dump导出再导入，如果是一张50GB的大表，那么会是一个噩梦。

但在MySQL 5.6/5.7和MariaDB 10.0/10.1版本里已解除了限制。下面演示如何在MySQL 5.7里迁移一张表到远程服务器上。条件必须是独立表空间，即set global innodb\_file\_per\_table=1。

首先，执行FLUSH TABLES hechunyang FOR EXPORT;hechunyang表加全局读锁（只能读，不能写，目的是保证数据的一致性），把.cfg数据字典文件导出到磁盘上，如图2-51所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)

mysql> FLUSH TABLES hechunyang FOR EXPORT;
Query OK, 0 rows affected (0.00 sec)
```

图2-51 对表加全局读锁

其次，到数据目录下把hechunyang.cfg和hechunyang.ibd拷贝到另一台机器上，命令如下：

```
scp hechunyang.cfg hechunyang.ibd 192.168.17.129:/tmp
```

再修改属性chown-R mysql:mysql hechunyang.\*。拷贝完成后，执行UNLOCK TABLES;命令，如图2-52所示。

```
-rw-r----- 1 mysql mysql 335 1月 20 01:10 hechunyang.cfg
-rw-r----- 1 mysql mysql 8.4K 1月 20 01:10 hechunyang.frm
-rw-r----- 1 mysql mysql 96K 1月 20 01:10 hechunyang.ibd
```

图2-52 修改数据文件属性为mysql

最后，在另一台机器上创建原表hechunyang的表结构，再执行如下命令：

```
ALTER TABLE hechunyang DISCARD TABLESPACE
```



该命令会关闭hechunyang表的数据空间，并会删除其数据文件。再把拷贝过来的hechunyang.cfg和hechunyang.ibd拷贝到该机器的数据目录下，并执行如下命令：

```
ALTER TABLE hechunyang IMPORT TABLESPACE;
```

就会进行恢复操作，再执行如下命令：

```
check table hechunyang;
```

若没问题，再执行如下命令：

```
select * from hechunyang;
```

就会发现数据恢复了。

如果担心锁表时间很长，则可以采用另一种方法innobackupex，演示如下：

```
innobackupex --user=root --password=123456
```

```
--defaults-file=/etc/my.cnf --tables-file=/tmp/table.txt --export /bak/
```

```
--export
```

导出字典文件.cfg

```
--tables-file
```

只备份某几张表，格式——（库名.表名）

```
# cat /tmp/table.txt
```

```
test.t1
```

恢复的命令如下：

```
innobackupex --user=root --password=123456 --defaults-file=/etc/my.cnf
```

```
--apply-log --tables-file=/tmp/table.txt --export /bak/
```

远程迁移恢复步骤与上述一致。

### 2.3.10 修改InnoDB redo log事务日志文件大小更人性化

在MySQL 5.5版本里，如果想修改ib\_logfile ( redo log ) 文件大小，那么步骤如下：

- 1) 执行set global innodb\_fast\_shutdown=0;命令，将所有的脏页刷到磁盘。
- 2) 执行mysqladmin shutdown命令关闭数据库。
- 3) 在my.cnf文件里修改innodb\_log\_file\_size参数值。
- 4) 执行mv ib\_logfile\*/bak命令，将redo log移动到/bak目录下（不移走，启动会报错）。
- 5) 执行mysqld\_safe--defaults-file=/etc/my.cnf--user=mysql&命令，最后启动MySQL服务。

通过上面的步骤可完成修改redo log文件的大小。

在MySQL 5.6/5.7或MariaDB 10.0/10.1版本里更人性化，上面的第4)步可以忽略，直接启动MySQL即可，如图2-53所示。

在MySQL 5.6/5.7或MariaDB 10.0/10.1版本里，InnoDB redo log的大小从最大4GB提高到512GB，可通过参数innodb\_log\_file\_size来配置，如图2-54所示。

```
2014-02-21 15:35:46 33342 [Warning] InnoDB: Resizing redo log from 2*3072 to 2*6400 pages, LSN=3663762863
2014-02-21 15:35:46 33342 [Warning] InnoDB: Starting to delete and rewrite log files.
2014-02-21 15:35:47 33342 [Note] InnoDB: Setting log file ./ib_logfile101 size to 100 MB
InnoDB: Progress in MB: 100
2014-02-21 15:35:50 33342 [Note] InnoDB: Setting log file ./ib_logfile1 size to 100 MB
InnoDB: Progress in MB: 100
2014-02-21 15:35:52 33342 [Note] InnoDB: Renaming log file ./ib_logfile101 to ./ib_logfile0
2014-02-21 15:35:52 33342 [Warning] InnoDB: New log files created, LSN=3663762956
```

图2-53 自动重置redo log文件大小

• innodb\_log\_file\_size

Command-Line Format	--innodb_log_file_size=#	
System Variable	Name	innodb_log_file_size
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.10)	Type	integer
	Default	50331648
	Min Value	1048576
	Max Value	512GB / innodb_log_files_in_group
Permitted Values (>= 5.7.11)	Type	integer
	Default	50331648
	Min Value	4194304
	Max Value	512GB / innodb_log_files_in_group

图2-54 设置innodb\_log\_file\_size参数最大值

如果innodb\_log\_files\_in\_group设置为3组redo log，那么innodb\_log\_file\_size\*innodb\_log\_files\_in\_group不能超过512GB。设置的值越大，越可以减少checkpoint刷新脏页的频率，这对提升MySQL性能很重要，但风险就是增加了宕机恢复的时间。这里建议TPS在200~300每秒/写的业务场景，通常设置为4GB即可。



### 2.3.11 死锁可以打印到错误日志里

在MySQL 5.6版本中查看死锁，需要执行show engine innodb status\G;命令。

在MySQL 5.6/5.7或MariaDB 10.0/10.1版本中，在my.cnf配置文件里加入：

```
innodb_print_all_deadlocks = 1
```

就可以把死锁信息打印到错误日志里。

### 2.3.12 支持InnoDB只读事务

在READ COMMITTED隔离级别下，无法避免幻读，为了不读取到脏数据，在MySQL 5.6/5.7或MariaDB 10.0/10.1版本中增加了显示开启InnoDB只读事务，如图2-55所示。

只读事务在查询时只能看到事务开始那一刻提交的修改，无论其他Session如何修改，START TRANSACTION READ ONLY事务里的查询结果均不会变化，可以将其理解为一个快照。从图2-55可以看出，只读事务模式下是无法执行插入/删除/更新操作的。

```
(root@localhost) [test]> START TRANSACTION READ ONLY;
Query OK, 0 rows affected (0.00 sec)

(root@localhost) [test]> select * from t1;
+----+-----+-----+
| id | pid | name |
+----+-----+-----+
| 1  | 100 | 贺春阳 |
+----+-----+-----+
1 row in set (0.00 sec)

(root@localhost) [test]> insert into t1 values(2,2,'李四');
ERROR 1792 (25006): Cannot execute statement in a READ ONLY transaction.
(root@localhost) [test]>
(root@localhost) [test]> select version();
+-----+
| version() |
+-----+
| 10.0.21-MariaDB |
+-----+
1 row in set (0.00 sec)
```

图2-55 开启只读事务



### 2.3.13 支持InnoDB表空间数据碎片整理

只有在MariaDB 10.1版本里支持InnoDB表空间数据碎片整理这项功能。对InnoDB进行修改操作时，例如删除一些行，这些行只是被标记为“已删除”，而不是真的从索引中物理删除了，因而空间也没有真的被释放回收。InnoDB的Purge线程会异步地来清理这些没有用的索引键和行，但是依然没有把这些释放出来的空间还给操作系统重新使用，因而会导致页面中存在很多空洞。

为了提升数据库的性能，需要进行一次数据整理。调整以后，数据文件将缩小，数据之间将按照顺序重新整理排序，这对于查询时提升性能会非常有利。

当发现数据物理文件和数据字典统计的数据大小差异过大时，表示要整理碎片了。那如何定期整理数据碎片呢？通常的做法是使用OPTIMIZE TABLE或者ALTER TABLE<table>ENGINE=InnoDB（必须为独立表空间），该方法会对旧表以复制的方式新建一个新表，然后删除旧表，相当于进行一次导出/导入重建新表。

在MariaDB 10.1版本里，合并了Facebook的碎片整理代码，开启新的整理算法需要把下面的配置加到my.cnf配置文件中：

```
[mysqld]
innodb-defragment = 1
#打开或关闭InnoDB碎片整理算法。
innodb_defragment_n_pages = 16
#一次性读取多少个页面进行合并整理操作。取值范围是2~32，默认值是7。
```

这样配置以后，新的碎片整理功能就会替代原有的OPTIMIZE TABLE算法，加快了OPTIMIZE TABLE的操作时间，不会有新的表生成，也不需要把旧表的数据拷贝到新表。新的算法会载入n个页面尝试把上面的记录紧凑地合并到一起，从而让页面存满记录，然后释放掉完全空了的页面。

以下是状态参数说明。

- Innodb\_defragment\_compression\_failures:整理碎片时重新压缩页面失败的次数。
- Innodb\_defragment\_failures:整理操作失败的次数（如没有可压缩的页面）。
- Innodb\_defragment\_count:整理操作的次数。



注意 在经过测试后，将100万行的表删除为50万行，并不会自动回收ibd数据文件空间，通过OPTIMIZE TABLE也并没有回收数据文件空间，必须通过ALTER TABLE<table>ENGINE=InnoDB才可以。



## 2.4 JSON格式的支持

### 2.4.1 支持用JSON格式存储数据

为了兼容传统的SQL语法，MySQL 5.7版本支持原生的JSON格式，即将关系型数据库和文档型NoSQL数据库集于一身，其表结构如图2-56所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.12 sec)

mysql> show create table t1\G;
***** 1. row *****
      Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL,
  `context` json DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

ERROR:
No query specified
```

图2-56 t1表结构

下面进行测试，插入JSON格式数据，如图2-57所示。

获取Key（键）name和age的Value（值）的方法，如图2-58所示。

获取全部Key（键）的方法如图2-59所示。

增加一个Key-Value（键-值）的方法如图2-60所示。

更改一个Key-Value（键-值）的方法如图2-61所示。

删除一个Key-Value ( 键-值 ) 的方法如图2-62所示。

```
mysql>
mysql> insert into t1 values(1, '{"name": "张三", "age": 21}'), (2, '{"name": "李四", "age": 34}');
Query OK, 2 rows affected (0.09 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> select * from t1;
```

id	context
1	{"age": 21, "name": "张三"}
2	{"age": 34, "name": "李四"}

```
2 rows in set (0.00 sec)
```

图2-57 插入JSON格式数据

```
mysql> select id, JSON_EXTRACT(context, '$.name') name, JSON_EXTRACT(context, '$.age') age from t1;
```

id	name	age
1	"张三"	21
2	"李四"	34

```
2 rows in set (0.00 sec)
```

图2-58 获取Key-Value ( 键-值 )

```
mysql> select id, json_keys(context) from t1;
```

id	json_keys(context)
1	["age", "name"]
2	["age", "name"]

```
2 rows in set (0.00 sec)
```

图2-59 获取全部Key ( 键 )

```
mysql> select * from t1;
+----+-----+
| id | context |
+----+-----+
| 1  | {"age": 21, "name": "张三"} |
| 2  | {"age": 34, "name": "李四"} |
+----+-----+
2 rows in set (0.00 sec)

mysql> update t1 set context=JSON_INSERT(context,'$.name','李四','$.address','beijing') where id = 2;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from t1;
+----+-----+
| id | context |
+----+-----+
| 1  | {"age": 21, "name": "张三"} |
| 2  | {"age": 34, "name": "李四", "address": "beijing"} |
+----+-----+
2 rows in set (0.00 sec)
```

图2-60 增加Key-Value (键-值)

```
mysql> select * from t1;
+----+-----+
| id | context |
+----+-----+
| 1  | {"age": 21, "name": "张三"} |
| 2  | {"age": 34, "name": "李四", "address": "beijing"} |
+----+-----+
2 rows in set (0.00 sec)

mysql> update t1 set context=JSON_SET(context,'$.name','隔壁老王') where id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from t1;
+----+-----+
| id | context |
+----+-----+
| 1  | {"age": 21, "name": "隔壁老王"} |
| 2  | {"age": 34, "name": "李四", "address": "beijing"} |
+----+-----+
2 rows in set (0.00 sec)
```

图2-61 更改Key-Value (键-值)



```
mysql> select * from t1;
```

id	context
1	{"age": 21, "name": "隔壁老王"}
2	{"age": 34, "name": "李四", "address": "beijing"}

```
2 rows in set (0.00 sec)
```

```
mysql> update t1 set context=JSON_REMOVE(context,'$.name') where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from t1;
```

id	context
1	{"age": 21}
2	{"age": 34, "name": "李四", "address": "beijing"}

```
2 rows in set (0.00 sec)
```

图2-62 删除Key-Value (键-值)

## 2.4.2 动态列支持用JSON格式存储数据

MariaDB实现动态列 ( Dynamic Columns ) 的方式与MySQL 5.7不一样。

assets的表结构如图2-63所示。

插入JSON格式数据，如图2-64所示。

```
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> show create table assets\G;
***** 1. row *****
      Table: assets
Create Table: CREATE TABLE `assets` (
  `item_name` varchar(32) NOT NULL,
  `dynamic_cols` blob,
  PRIMARY KEY (`item_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-63 assets表结构

```
MariaDB [test]> INSERT INTO assets VALUES ('MariaDB T-shirt', COLUMN_CREATE('color', 'blue', 'size', 'XL'));
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> INSERT INTO assets VALUES ('Thinkpad Laptop', COLUMN_CREATE('color', 'black', 'price', 500));
Query OK, 1 row affected (0.00 sec)
```

图2-64 插入JSON格式数据

获取全部Key-Value ( 键-值 )，如图2-65所示。

```
MariaDB [test]> select * from assets;
```

item_name	dynamic_cols
MariaDB T-shirt	3 sizecolor!XL!blue
Thinkpad Laptop	`colorprice!black`

```
2 rows in set (0.00 sec)
```

图2-65 查询数据乱码

直接通过select查询出来的结果是乱码，需要用以下函数查询，如图2-66所示。

```
MariaDB [test]> SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
```

item_name	COLUMN_JSON(dynamic_cols)
MariaDB T-shirt	{"size": "XL", "color": "blue"}
Thinkpad Laptop	{"color": "black", "price": 500}

```
2 rows in set (0.00 sec)
```

图2-66 通过函数查询

获取全部Key（键），如图2-67所示。

获取Key（键）color的Value（值），如图2-68所示。

增加一个Key-Value（键-值），如图2-69所示。

更改一个Key-Value（键-值），如图2-70所示。

```
MariaDB [test]> SELECT item_name, column_list(dynamic_cols) FROM assets;
```

item_name	column_list(dynamic_cols)
MariaDB T-shirt	`size`,`color`
Thinkpad Laptop	`color`,`price`

```
2 rows in set (0.00 sec)
```

图2-67 获取全部Key（键）



```
MariaDB [test]> SELECT item_name, COLUMN_GET(dynamic_cols, 'color' as char) AS color FROM assets;
```

item_name	color
MariaDB T-shirt	blue
Thinkpad Laptop	black

2 rows in set (0.02 sec)

图2-68 获取Key-Value (键-值)

```
MariaDB [test]> UPDATE assets SET dynamic_cols=COLUMN_ADD(dynamic_cols, 'warranty', '3 years') WHERE item_name='Thinkpad Laptop';
Query OK, 1 row affected (0.03 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
MariaDB [test]> SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
```

item_name	COLUMN_JSON(dynamic_cols)
MariaDB T-shirt	{"size": "XL", "color": "blue"}
Thinkpad Laptop	{"color": "black", "price": 500, "warranty": "3 years"}

2 rows in set (0.00 sec)

图2-69 增加Key-Value (键-值)

```
MariaDB [test]> UPDATE assets SET dynamic_cols=COLUMN_ADD(dynamic_cols, 'color', 'white') WHERE COLUMN_GET(dynamic_cols, 'color' as char)='black';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
MariaDB [test]> SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
```

item_name	COLUMN_JSON(dynamic_cols)
MariaDB T-shirt	{"size": "XL", "color": "blue"}
Thinkpad Laptop	{"color": "white", "price": 500, "warranty": "3 years"}

2 rows in set (0.00 sec)

图2-70 更改Key-Value (键-值)

删除一个Key-Value (键-值)，如图2-71所示。

```

MariaDB [test]> UPDATE assets SET dynamic_cols=COLUMN_DELETE(dynamic_cols, "price") WHERE COLUMN_GET(dynamic_cols, 'color' as char)='white';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
+-----+-----+
| item_name | COLUMN_JSON(dynamic_cols) |
+-----+-----+
| MariaDB T-shirt | {"size": "XL", "color": "blue"} |
| Thinkpad Laptop | {"color": "white", "warranty": "3 years"} |
+-----+-----+
2 rows in set (0.00 sec)

```

图2-71 删除Key-Value (键-值)

虽然MySQL 5.7和MariaDB 10.0/10.1版本对于JSON的支持是比较完整的，但自身没有提供MongoDB的Sharding功能，如果你的生产环境数据量不大，那么可以尝试。

## 2.5 支持虚拟列（函数索引）

### 2.5.1 MySQL 5.7支持函数索引

在MySQL 5.6中，函数索引是无法用到索引的，下面的SQL在执行时会进行全表扫描：  
*select \* from t1 where mod(mod\_id, 10) = 1;*

但在MySQL 5.7里，按如下方式创建表，就可以使用函数索引，其表结构如图2-72所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)

mysql> show create table t2\G;
***** 1. row *****
      Table: t2
Create Table: CREATE TABLE `t2` (
  `id` int(11) NOT NULL,
  `mod_id` int(11) GENERATED ALWAYS AS ((`id` % 10)) VIRTUAL,
  PRIMARY KEY (`id`),
  KEY `IX_mod_id` (`mod_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-72 t2表结构

插入数据时，注意虚拟列的值要为default，否则会报错，如图2-73所示。



```
mysql> insert into t2(id,mod_id) values(55,55);
ERROR 3105 (HY000): The value specified for generated column 'mod_id' in table 't2' is not allowed.
mysql>
mysql> insert into t2(id,mod_id) values(55,default);
Query OK, 1 row affected (0.07 sec)

mysql> select * from t2;
+----+-----+
| id | mod_id |
+----+-----+
| 55 |      5 |
+----+-----+
1 row in set (0.00 sec)
```

图2-73 插入字段值必须为default

通过explain执行计划器可以看到，已经使用到了索引，如图2-74所示。

```
mysql> explain select * from t2 where mod_id=5;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | NULL | ref | IX_mod_id | IX_mod_id | 5 | const | 1 | 100.00 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

图2-74 执行计划返回结果

通过这种方式即可实现函数索引。

## 2.5.2 MariaDB 10.0/10.1支持函数索引

在MySQL 5.6中，函数索引是无法用到索引的，下面的SQL在执行时会进行全表扫描：

```
select * from t1 where mod(mod_id, 10) = 1;
```

但在MariaDB10.0/10.1里，按如下方式创建表，就可以使用函数索引（MariaDB实现的方式与MySQL5.7不一样），其表结构如图2-75所示。

```
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> show create table t2\G;
***** 1. row *****
      Table: t2
Create Table: CREATE TABLE `t2` (
  `id` int(11) NOT NULL,
  `mod_id` int(11) AS ((`id` % 10)) PERSISTENT,
  PRIMARY KEY (`id`),
  KEY `IX_mod_id` (`mod_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图2-75 t2表结构

插入数据时，注意虚拟列的值要为default，否则会报错，如图2-76所示。

```
MariaDB [test]> insert into t2(id,mod_id) values(55,default);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select * from t2;
+----+-----+
| id | mod_id |
+----+-----+
| 55 |      5 |
+----+-----+
1 row in set (0.00 sec)
```

图2-76 插入字段值必须为default

通过explain执行计划器可以看到，已经使用了索引，如图2-77所示。

通过这种方式也可实现函数索引。

```
MariaDB [test]> explain select * from t2 where mod_id=5;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ref	IX_mod_id	IX_mod_id	5	const	1	Using index

```
1 row in set (0.00 sec)
```

图2-77 执行计划返回结果



## 2.6 功能提升

### 2.6.1 支持杀死慢的SQL语句

还在为慢的SQL语句而烦恼吗？MariaDB 10.1和Percona 5.6/5.7解决了这个问题，这个补丁是由Twitter提供的。不过，MySQL5.7不支持这个功能。

通过max\_statement\_time参数控制SQL执行时间（单位为秒），默认为0，不限制SQL的执行时间，假如你定义了超过1秒的慢的SQL语句，由DB自动杀死，那么就要设置：

```
set global max_statement_time = 1;
```

下面以MariaDB 10.1为例进行说明，如图2-78所示。

```
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> set max_statement_time = 1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select count(distinct pad) from sbtest;
ERROR 1969 (70100): Query execution was interrupted (max_statement_time exceeded)
MariaDB [test]>
MariaDB [test]> update sbtest set k=1;
ERROR 1969 (70100): Query execution was interrupted (max_statement_time exceeded)
MariaDB [test]>
MariaDB [test]> alter table sbtest add name char(10);
ERROR 1317 (70100): Query execution was interrupted
MariaDB [test]>
```

图2-78 自动终止执行慢的SQL语句

由图2-78可以看到，DML/DDl语句全被杀死。

也可以选择用percona-toolkit工具集的pt-kill命令来实现上述功能，它将从show processlist中获取满足条件的连接杀掉，主要是为了防止执行时间很长的查询，长时间占用系统资源，而对线上业务造成影响。

下面是pt-kill工具的使用方法。

首先，安装percona-toolkit工具集，命令如下：

```
# wget
https://www.percona.com/downloads/percona-toolkit/2.2.16/tarball/percona-toolkit-2.2.16.tar.gz
# tar zxvf percona-toolkit-2.2.16.tar.gz
# cd percona-toolkit-2.2.16
# perl Makefile.PL
# make
# make install
```

如果只需要把select耗时3秒以上的SQL全部杀死，并打印出来，可采用如下命令：

```
# /usr/local/bin/pt-kill -S /tmp/mysql.sock -u root -p 123456 \
--match-info "^(select/SELECT/Select)" \
--busy-time 2 --victim all --interval 1 --kill-query --print --log \
/root/kill.txt --daemonize
```

其中的参数解释如下：

- match-info：DML语句匹配，这里为匹配select|SELECT|Select查询。
- busy-time：定义SQL执行时间，这里为执行时间大于等于2秒。
- interval：运行检查query的间隔（秒）。
- victim：有三个值，分别为oldest（默认）、all和all:but-oldest。
  - oldest：杀掉长时间等待的查询，而不是长时间执行的查询。
  - all：杀掉所有满足条件的查询。
  - all-but-oldest：杀死所有，但最长的保留不杀死。
- kill-query：只杀掉连接执行的语句，但是线程不会被终止。
- kill：杀掉连接并且退出。
- print：打印满足条件的语句。
- log：将慢SQL语句记录在日志文件里。
- daemonize：放在后台以守护进程的形式运行。





如果故意执行一条慢SQL语句，则pt-kill会自动将其杀死，如图2-79所示。

```
mysql> select count(distinct pad) from sbtest;  
ERROR 1317 (70100): Query execution was interrupted  
mysql>
```

图2-79 自动终止执行慢的SQL语句

kill.txt会把慢SQL语句记录下来，如图2-80所示。

如何关闭pt-kill后台进程呢？执行命令kill-9\$(ps-ef|grep pt-kill|grep-v grep|awk'{print\$2}')

综上所述，运行平稳的数据库，如果遇到CPU狂飙，到80%左右，那一定是慢SQL运行导致的，DBA首先要保证的是：数据库别跑挂了，所以要把那些运行慢的SQL杀死并记录到文件里，以便后面的排查与优化。

```
[root@master ~]# cat kill.txt  
# 2016-01-21T02:01:38 KILL QUERY 27 (Query 2 sec) select count(*) from sbtest  
# 2016-01-21T02:02:31 KILL QUERY 37 (Query 2 sec) select count(*) from sbtest  
# 2016-01-21T02:03:06 KILL QUERY 37 (Query 2 sec) select count(distinct pad) from sbtest  
[root@master ~]#
```

图2-80 记录终止慢的SQL语句



## 2.6.2 支持一张表有多个INSERT/DELETE/UPDATE触发器

在MySQL 5.6和MariaDB 10.1版本里，不支持一张表有多个INSERT/DELETE/UPDATE触发器。但在MySQL 5.7版本里，支持一张表上有多个触发器，这样原表已有的触发器也可以支持使用pt-online-schema-change修改表结构了。

来看一个例子，通过代码创建t1表两个INSERT的触发器。第一个触发器的名字为t1\_1，代码如下：

```
DELIMITER $$
USE `test`$$
DROP TRIGGER `t1_1`$$
CREATE
  TRIGGER `t1_1` AFTER INSERT ON `t1`
  FOR EACH ROW BEGIN
INSERT INTO t2(id , NAME) VALUES(new.id , new.name);
  END;
$$
DELIMITER ;
```

第二个触发器的名字为t1\_2，代码如下：

```
DELIMITER $$
USE `test`$$
DROP TRIGGER `t1_2`$$
CREATE
  TRIGGER `t1_2` AFTER INSERT ON `t1`
  FOR EACH ROW BEGIN
INSERT INTO t3(id , NAME) VALUES(new.id , new.name);
  END;
$$
DELIMITER ;
```

以MariaDB 10.1版本为例，执行多个触发器的时候就会报错，如图2-81所示。

而在MySQL 5.7版本里，已经支持一张表有多个INSERT/DELETE/UPDATE触发器，如图2-82所示。

```
MariaDB [test]> DELIMITER $$
MariaDB [test]> USE `test`$$
Database changed
MariaDB [test]> DROP TRIGGER `t1_2`$$
ERROR 1360 (HY000): Trigger does not exist
MariaDB [test]> CREATE
-> TRIGGER `t1_2` AFTER INSERT ON `t1`
-> FOR EACH ROW BEGIN
-> INSERT INTO t3(id,NAME) VALUES(new.id,new.name);
-> END;
-> $$
ERROR 1235 (42000): This version of MariaDB doesn't yet support 'multiple triggers with the same action time and event for one table'
MariaDB [test]> DELIMITER ;
MariaDB [test]>
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)
```

图2-81 MariaDB 10.1版本不支持单表拥有多个触发器

```
mysql> select TRIGGER_SCHEMA, TRIGGER_NAME, EVENT_MANIPULATION, ACTION_STATEMENT, ACTION_TIMING from triggers where TRIGGER_SCHEMA='test';
+-----+-----+-----+-----+-----+
| TRIGGER_SCHEMA | TRIGGER_NAME | EVENT_MANIPULATION | ACTION_STATEMENT | ACTION_TIMING |
+-----+-----+-----+-----+-----+
| test          | t1_1         | INSERT              | BEGIN            |               | |
|               |              | INSERT INTO t2(id,NAME) VALUES(new.id,new.name); |               |               |
|               |              | END | AFTER         |                  |               |
| test          | t1_2         | INSERT              | BEGIN            |               |
|               |              | INSERT INTO t3(id,NAME) VALUES(new.id,new.name); |               |               |
|               |              | END | AFTER         |                  |               |
+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)
```

图2-82 MySQL 5.7版本支持单表拥有多个触发器



### 2.6.3 引入线程池 ( Thread Pool ) 技术

为了满足不断增长的用户、查询和数据通信量对性能和扩展性的持续需求，在MariaDB5.5和MariaDB10.0/10.1版本里，引入了线程池 ( Thread Pool ) 技术，线程池提供了一种具有高度扩展性的线程处理模型，旨在减少管理客户连接和语句执行线程的开销，减少CPU上下文切换，非常适合高并发php短连接应用场景 ( MySQL 5.5.20之后，MySQL 5.6、MySQL 5.7企业版支持 )。



注意 该技术不适用长连接！如果生产环境采用的是阿里巴巴开源的druid连接池，则不需要开启线程池 ( Thread Pool )。

解释线程池的作用：都遇到过堵车吧？所有人都想赶紧过去，都互相抢着先走，结果你不让我，我不让你，那就谁都没法过，都在这堵着，这就是没有线程池，但如果来一个交警指挥交通，按照次序排着队一辆一辆地过，这就是线程池的作用。

没有线程池的情况类似于图2-83。



图2-83 无连接池

有线程池的情况如图2-84所示。

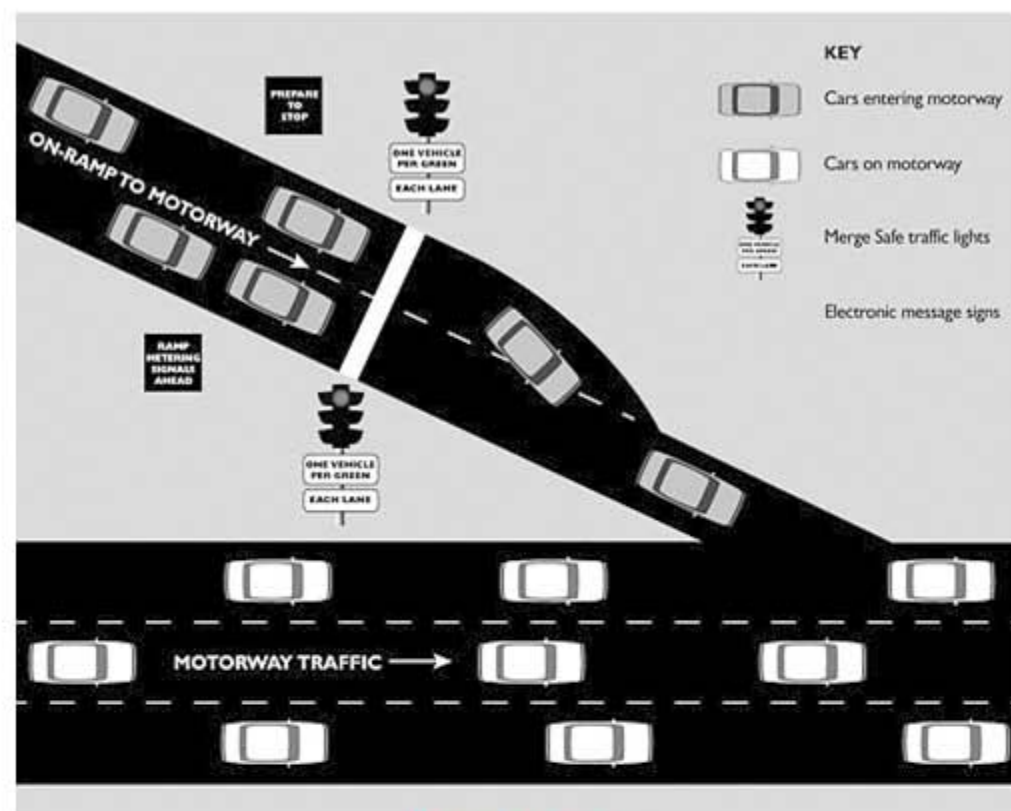


图2-84 有连接池

下面来看看服务不断增长的用户连接数量，以及大通信量的在线应用性能和扩展性持续改善效果。MySQL5.6企业版官方性能压测报告分别如图2-85和图2-86所示。

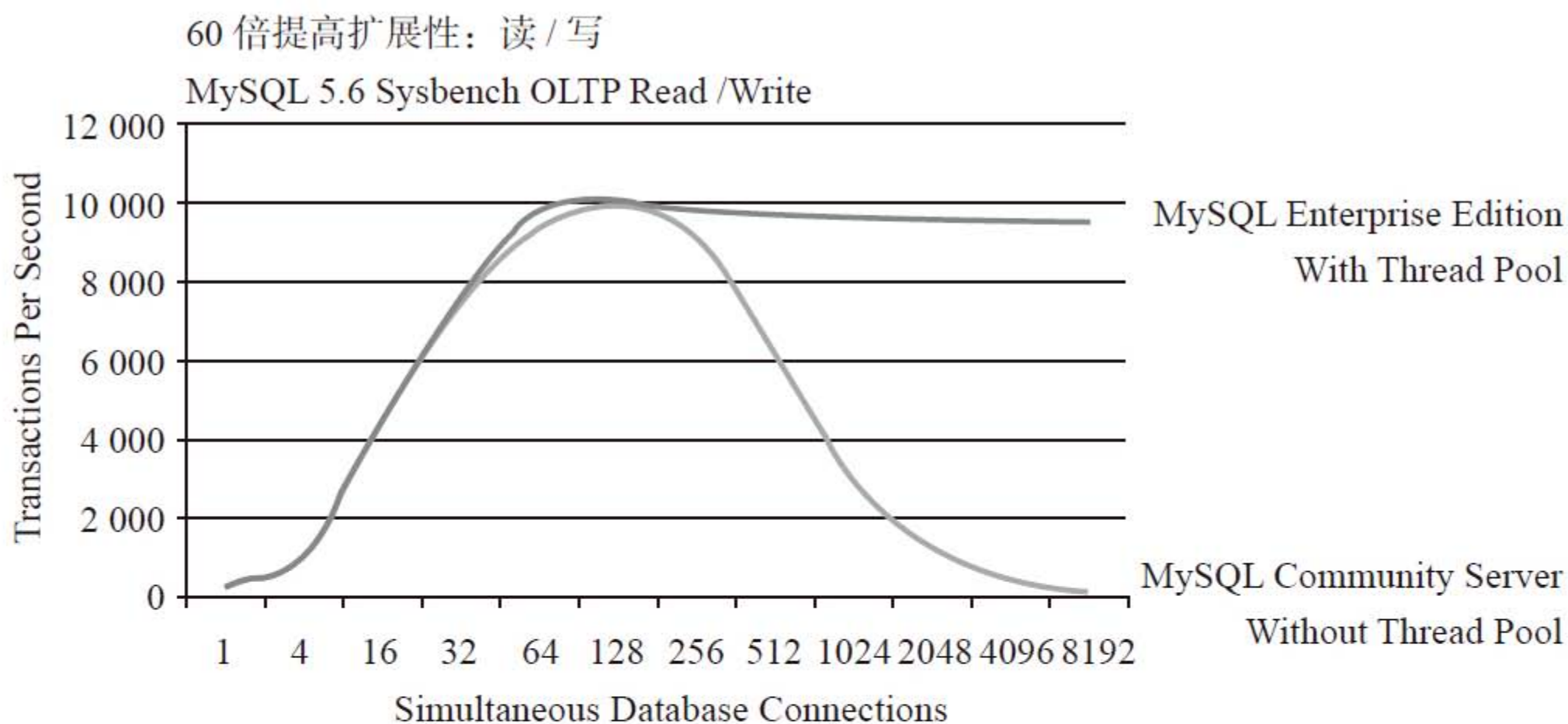


图2-85 sysbench读/写模式——每秒查询数

从图2-85可以看到，与MySQL社区版服务器相比，带MySQL线程池的MySQL企业版可提供60倍的读/写扩展性。



18 倍提高扩展性：只读

MySQL 5.6 Sysbench OLTP Read Only

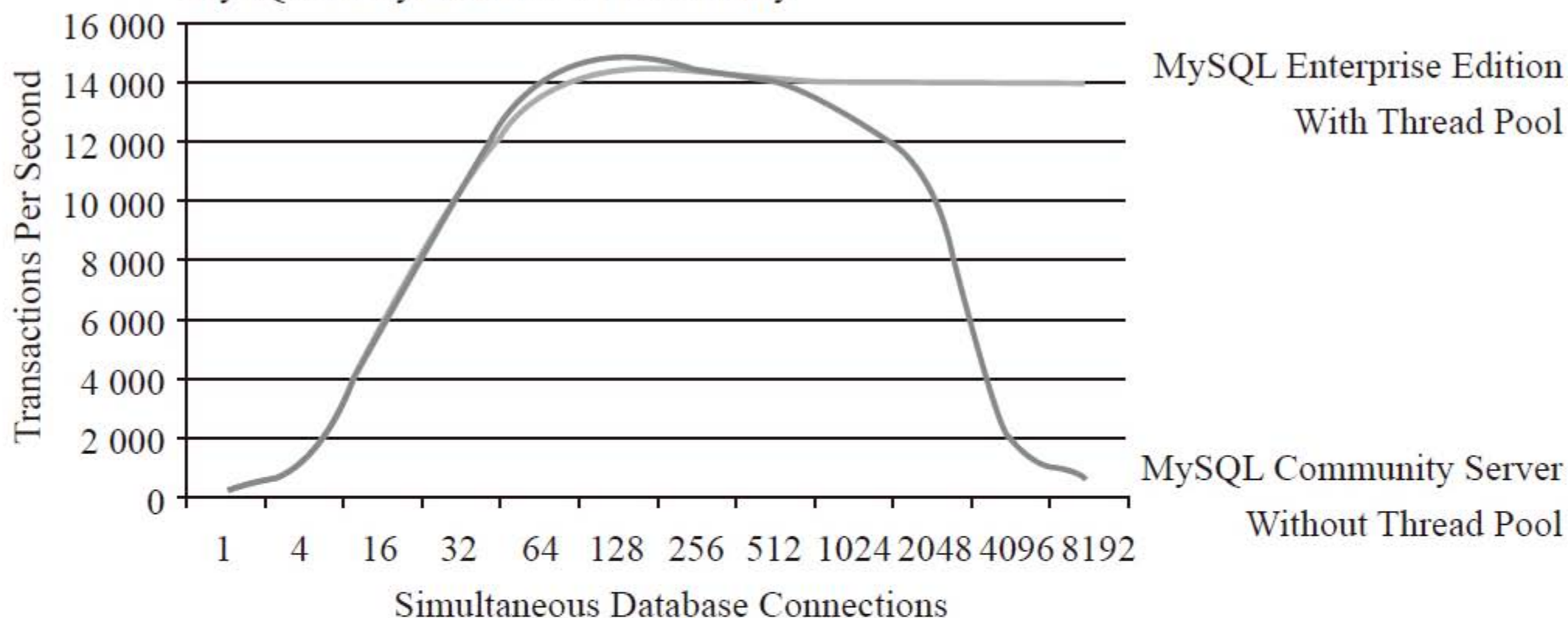


图2-86 sysbench只读模式——每秒查询数

从图2-86可以看到，与MySQL社区版服务器相比，带MySQL线程池的MySQL企业版可提供18倍的只读扩展性。

官方服务器的硬件配置如下：

MySQL 5.6.11

Oracle Linux 6.3 , Unbreakable Kernel 2.6.32

4 sockets , 24 cores , 48 Threads

Intel(R)Xeon® E7540 2GHz CPUs

512GB DDR RAM

参考官网地址为：<http://www.mysql.com/products/enterprise/scalability.html>。

如何开启线程池（以MariaDB 10.0为例）？可在my.cnf配置文件里设置如下参数：

`thread_handling=pool-of-threads。`

·thread\_pool\_max\_threads=500。

默认为500就够用了，自动创建500个线程，如果有促销活动，就改为1000，别忘了修改ulimit文件描述符65535。再重启mysql服务即可，其他参数不用设置。



## 2.6.4 提供审计日志功能

如果有一天数据库里丢失一条记录，总监让DBA查什么时候被谁从哪个IP执行了delete操作，那么在MySQL社区版是无法查到的，该技术主要是在MariaDB 10.0/10.1和Percona 5.6版本里实现。

审计日志，适合互联网金融企业。谁干坏事，全记录在案，日志需要单独存放在一块磁盘上，避免频繁顺序I/O的刷盘操作。该功能在MySQL 5.6/5.7企业版里也支持。

下面介绍在MariaDB 10.1中安装审计Audit Plugin插件的方法，安装时执行如下命令：

```
INSTALL PLUGIN server_audit SONAME 'server_audit.so';
```

具体代码如图2-87所示。



```

MariaDB [(none)]> INSTALL PLUGIN server_audit SONAME 'server_audit.so';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> select * from information_schema.PLUGINS where PLUGIN_NAME like '%audit%'\G;
***** 1. row *****
      PLUGIN_NAME: SERVER_AUDIT
      PLUGIN_VERSION: 1.3
      PLUGIN_STATUS: ACTIVE
      PLUGIN_TYPE: AUDIT
      PLUGIN_TYPE_VERSION: 3.2
      PLUGIN_LIBRARY: server_audit.so
      PLUGIN_LIBRARY_VERSION: 1.11
      PLUGIN_AUTHOR: Alexey Botchkov (MariaDB Corporation)
      PLUGIN_DESCRIPTION: Audit the server activity
      PLUGIN_LICENSE: GPL
      LOAD_OPTION: ON
      PLUGIN_MATURITY: Stable
      PLUGIN_AUTH_VERSION: 1.3.0
1 row in set (0.00 sec)

ERROR: No query specified

MariaDB [(none)]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

```

图2-87 安装审计日志插件

以下是相关参数的解释。

- server\_audit\_events='CONNECT, QUERY, TABLE'：表示会记录连接进来的IP、用户名和密码及表的DML/DDl/DCL操作。
- server\_audit\_logging=ON：表示开启审计日志服务。
- server\_audit\_incl\_users=hechunyang：表示只记录hechunyang用户的所有操作。
- server\_audit\_file\_rotate\_size=1G：表示超过定义的1GB，会自动轮训，切分日志。
- server\_audit\_file\_path=/data/audit/server\_audit.log：表示设置审计日志的路径。

查看效果如图2-88所示。

```
[root@master audit]# cat server_audit.log
20150922 00:13:02, master, root, localhost, 3, 0, CONNECT,,, 0
20150922 00:13:21, master, root, localhost, 3, 0, DISCONNECT,,, 0
20150922 00:13:30, master, hechunyang, localhost, 4, 0, CONNECT,,, 0
20150922 00:13:30, master, hechunyang, localhost, 4, 3, QUERY,, 'select @@version_comment limit 1', 0
20150922 00:13:32, master, hechunyang, localhost, 4, 4, QUERY,, 'SELECT DATABASE()', 0
20150922 00:13:32, master, hechunyang, localhost, 4, 6, QUERY, test, 'show databases', 0
20150922 00:13:32, master, hechunyang, localhost, 4, 7, QUERY, test, 'show tables', 0
20150922 00:13:33, master, hechunyang, localhost, 4, 10, QUERY, test, 'show tables', 0
20150922 00:13:37, master, hechunyang, localhost, 4, 11, READ, test, t1,
20150922 00:13:37, master, hechunyang, localhost, 4, 11, QUERY, test, 'select * from t1', 0
20150922 00:13:43, master, hechunyang, localhost, 4, 12, QUERY, test, 'desc t1', 0
20150922 00:14:00, master, hechunyang, localhost, 4, 13, WRITE, test, t1,
20150922 00:14:00, master, hechunyang, localhost, 4, 13, QUERY, test, 'insert into t1 values(1, default)', 0
20150922 00:14:02, master, hechunyang, localhost, 4, 14, READ, test, t1,
20150922 00:14:02, master, hechunyang, localhost, 4, 14, QUERY, test, 'select * from t1', 0
```

图2-88 审计日志内容

## 2.6.5 支持explain update

在MySQL 5.5版本里，explain只支持select，但在MySQL 5.6/5.7、MariaDB 10.0/10.1版本中，支持explain update/delete，这样就可以更方便查看SQL执行计划了。

在MariaDB 10.1里执行explain update，如图2-89所示。

```
MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)
```

```
MariaDB [test]> explain update sbtest set c='hechunyang' where id = 10;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sbtest | range | PRIMARY | PRIMARY | 4 | NULL | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图2-89 MariaDB 10.1支持explain update

在MySQL 5.7里执行explain update，如图2-90所示。



mysql> select version();

version()
5.7.10-log

1 row in set (0.00 sec)

mysql> explain update sbtest set c='hechunyang' where id = 10;

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	UPDATE	sbtest	NULL	range	PRIMARY	PRIMARY	4	const	1	100.00	Using where

1 row in set (0.27 sec)

图2-90 MySQL 5.7支持explain update

## 2.6.6 在MySQL 5.7中按Ctrl+C组合键不会退出客户端

在MySQL 5.6和MariaDB 10.0/10.1版本中按Ctrl+C组合键会直接终止当前会话并退出客户端，而在MySQL 5.7版本中按Ctrl+C组合键不会退出客户端而是终止当前会话操作，如图2-91所示。

```
mysql> update sbtest set c='nba';
^C^C^C^C -- query aborted
^C^C -- query aborted
^C -- query aborted
ERROR 1317 (70100): Query execution was interrupted
mysql> ^C
mysql> ^C
mysql> ^C
mysql>
mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host      | db  | Command | Time | State   | Info                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 5  | root | localhost | test | Query   | 0    | starting | show processlist    |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.18 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.17 sec)
```

图2-91 在MySQL 5.7版本中按Ctrl+C组合键不会退出客户端

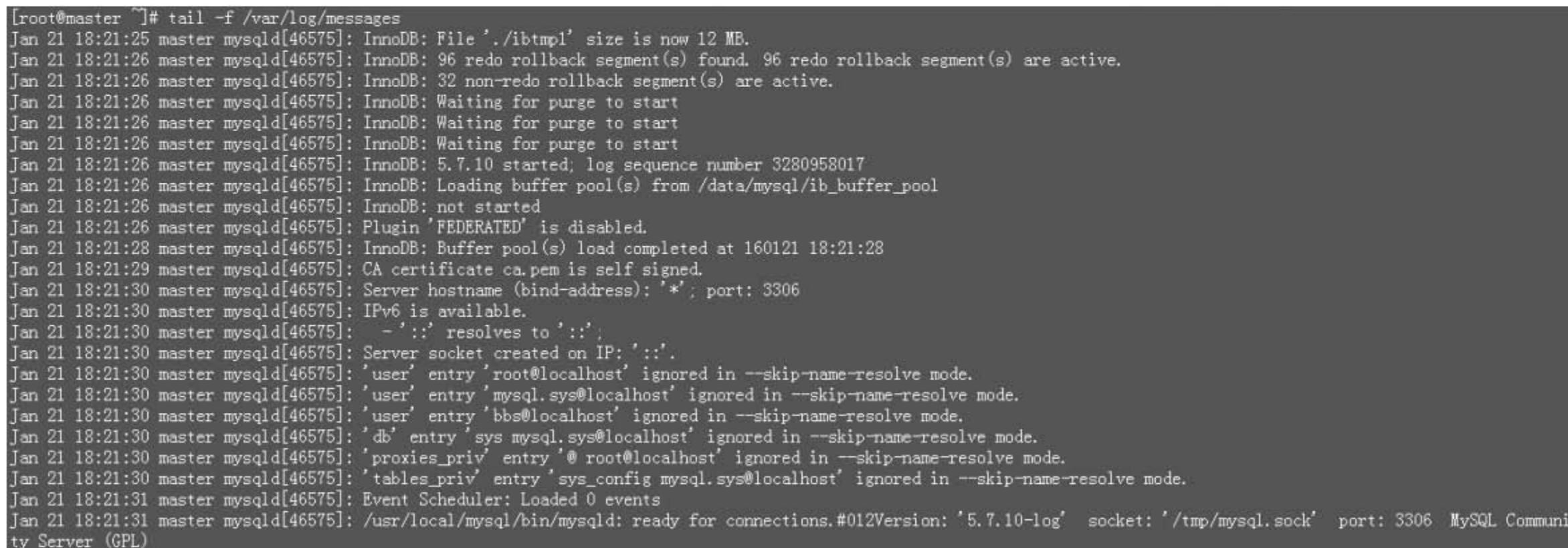
## 2.6.7 可将错误日志打印到系统日志文件中

通常情况下，MySQL的日志信息会输出到数据目录的hostname.err文件中，但MySQL 5.7和MariaDB 10.0/10.1版本可以把日志通过syslog输出到系统的LOG中，便于统一分析处理。

服务启动的时候，mysqld\_safe命令加--syslog参数，如下：

```
/usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf --syslog --user=mysql &
```

查看/var/log/message，即可看到MySQL错误日志，如图2-92所示。



```
[root@master ~]# tail -f /var/log/messages
Jan 21 18:21:25 master mysqld[46575]: InnoDB: File './ibtmp1' size is now 12 MB.
Jan 21 18:21:26 master mysqld[46575]: InnoDB: 96 redo rollback segment(s) found. 96 redo rollback segment(s) are active.
Jan 21 18:21:26 master mysqld[46575]: InnoDB: 32 non-redo rollback segment(s) are active.
Jan 21 18:21:26 master mysqld[46575]: InnoDB: Waiting for purge to start
Jan 21 18:21:26 master mysqld[46575]: InnoDB: Waiting for purge to start
Jan 21 18:21:26 master mysqld[46575]: InnoDB: Waiting for purge to start
Jan 21 18:21:26 master mysqld[46575]: InnoDB: 5.7.10 started; log sequence number 3280958017
Jan 21 18:21:26 master mysqld[46575]: InnoDB: Loading buffer pool(s) from /data/mysql/ib_buffer_pool
Jan 21 18:21:26 master mysqld[46575]: InnoDB: not started
Jan 21 18:21:26 master mysqld[46575]: Plugin 'FEDERATED' is disabled.
Jan 21 18:21:28 master mysqld[46575]: InnoDB: Buffer pool(s) load completed at 160121 18:21:28
Jan 21 18:21:29 master mysqld[46575]: CA certificate ca.pem is self signed.
Jan 21 18:21:30 master mysqld[46575]: Server hostname (bind-address): '*'; port: 3306
Jan 21 18:21:30 master mysqld[46575]: IPv6 is available.
Jan 21 18:21:30 master mysqld[46575]: - '::' resolves to '::'.
Jan 21 18:21:30 master mysqld[46575]: Server socket created on IP: '::'.
Jan 21 18:21:30 master mysqld[46575]: 'user' entry 'root@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:30 master mysqld[46575]: 'user' entry 'mysql.sys@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:30 master mysqld[46575]: 'user' entry 'bbs@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:30 master mysqld[46575]: 'db' entry 'sys mysql.sys@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:30 master mysqld[46575]: 'proxies_priv' entry '@ root@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:30 master mysqld[46575]: 'tables_priv' entry 'sys_config mysql.sys@localhost' ignored in --skip-name-resolve mode.
Jan 21 18:21:31 master mysqld[46575]: Event Scheduler: Loaded 0 events
Jan 21 18:21:31 master mysqld[46575]: /usr/local/mysql/bin/mysqld: ready for connections.#012Version: '5.7.10-log' socket: '/tmp/mysql.sock' port: 3306 MySQL Communi
ty Server (GPL)
```

图2-92 message系统日志显示MySQL信息

另外，通过客户端mysql命令连接时，加--syslog参数，会将交互式语句发送到系统日志（syslog）里，命令如下：

```
mysql -uroot -p123456 -S /tmp/mysql.sock --syslog
```

查看/var/log/message，如图2-93所示。



```
Jan 21 18:32:54 master MysqlClient: SYSTEM_USER:'root', MYSQL_USER:'--', CONNECTION_ID:5, DB_SERVER:'--', DB:'--', QUERY:'select version();'  
Jan 21 18:32:57 master MysqlClient: SYSTEM_USER:'root', MYSQL_USER:'--', CONNECTION_ID:5, DB_SERVER:'--', DB:'--', QUERY:'select now();'  
Jan 21 18:33:01 master MysqlClient: SYSTEM_USER:'root', MYSQL_USER:'--', CONNECTION_ID:5, DB_SERVER:'--', DB:'--', QUERY:'use test;'  
Jan 21 18:33:12 master MysqlClient: SYSTEM_USER:'root', MYSQL_USER:'--', CONNECTION_ID:5, DB_SERVER:'--', DB:'test', QUERY:'select * from sbtest limit 10;'
```

图2-93 message系统日志显示终端执行操作语句

## 2.6.8 支持创建角色

只在MariaDB 10.0/10.1版本里支持创建角色（grant role）这项功能。在数据库中，为了便于对用户及权限进行管理，可以将一组具有相同权限的用户组织在一起，这一组具有相同权限的用户就称为角色（role）。在实际工作中，有大量的用户其权限是一样的。如果让DBA在每次创建完用户后都对每个用户分别授权，则是一件非常麻烦的事情。但如果把具有相同权限的用户集中在角色中进行管理，则会方便很多。

为一个角色进行权限管理，就相当于对该角色中的所有成员进行操作。可以为有相同权限的一类用户建立一个角色，然后为角色授予合适的权限。使用角色的好处是DBA只需对权限的种类进行划分，然后将不同的权限授予不同的角色，而不必关心有哪些具体的用户。而且当角色中的成员发生变化时，比如添加成员或删除成员，系统管理员都无需执行任何关于权限的操作。

该功能只有MariaDB10.0/10.1、MySQL5.7版本尚不支持。

角色授权使用说明如下：

- 1) 创建一个develop角色。
- 2) 给develop角色授予select/insert/update/delete权限。
- 3) 赋予dev@'%' 用户develop角色，并创建密码123456（见图2-94）。

```
MariaDB [(none)]> CREATE ROLE develop;
Query OK, 0 rows affected (0.01 sec)

MariaDB [(none)]> GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO develop;
Query OK, 0 rows affected (0.01 sec)

MariaDB [(none)]> GRANT develop TO 'dev'@'%' identified by '123456';
Query OK, 0 rows affected (0.01 sec)
```

图2-94 创建develop角色并赋权限

- 4) 对dev用户设置develop为默认角色。
- 5) 开启develop角色（见图2-95）。

```

root@master:~ root@master:~ X
[root@master ~]# mysql -h127.0.0.1 -udev -p123456
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 14
Server version: 10.1.6-MariaDB-log MariaDB Server

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> set default role develop;
Query OK, 0 rows affected (0.01 sec)

MariaDB [(none)]> set role develop;
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> use hcy
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [hcy]> create table t2;
ERROR 1142 (42000): CREATE command denied to user 'dev'@'127.0.0.1' for table 't2'
MariaDB [hcy]>
MariaDB [hcy]> select * from t1;
Empty set (0.00 sec)

MariaDB [hcy]> insert into t1 select 1;
Query OK, 1 row affected (0.02 sec)
Records: 1 Duplicates: 0 Warnings: 0

MariaDB [hcy]> update t1 set id=2;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1 Changed: 1 Warnings: 0

```

图2-95 设置develop为默认角色



## 2.6.9 支持TokuDB存储引擎

TokuDB是一个insert高性能、支持事务处理的存储引擎。TokuDB的主要特点是密集压缩，支持事务，MVCC多版本并发控制，聚集索引（主键），并且支持在线DDL操作，可以把它看成是ARCHIVE存储引擎的升级版。MariaDB和Percona都捆绑了TokuDB引擎。

下面针对常见的存储引擎进行对比，分别如表2-1和表2-2所示。

表2-1 压缩对比

存储引擎	压缩算法	表大小 [MB]
InnoDB	none	2272
InnoDB	KEY_BLOCK_SIZE=8	1144
InnoDB	KEY_BLOCK_SIZE=4	584
MyISAM	none	1810
MyISAM	compressed with myisampack	809
Archive	default	211
TokuDB	ZLIB	284
TokuDB	LZMA	208

表2-2 特性对比

特性	Archive	MyISAM (压缩)	InnoDB	TokuDB
DML	Only Insert	No	Yes	Yes
ACID	No	No	Yes	Yes
Index	No	Yes	Yes	Yes
Online DDL	No	No	Yes	Yes

TokuDB的安装说明如下。

1) 关闭Redhat/Centos系统内核Transparent HugePage。

关闭命令如下：

```
# echo never >/sys/kernel/mm/redhat_transparent_hugepage/defrag
# echo never >/sys/kernel/mm/redhat_transparent_hugepage/enabled
# echo never >/sys/kernel/mm/transparent_hugepage/enabled
# echo never >/sys/kernel/mm/transparent_hugepage/defrag
```

再将其加入/etc/rc.local文件里，以便重启后自动生效。



注意 如果不关闭transparent\_hugepage，会导致内存泄露。

2) 在MariaDB 10.1中安装TokuDB插件。

注：mariadb-enterprise-10.1.10-linux-x86\_64.tar.gz二进制安装包里并没有捆绑TokuDB插件，需要下载mariadb-enterprise-10.1.10-linux-glibc\_214-x86\_64.tar.gz二进制安装包。有读者会问，这两个安装包有什么区别？你可以使用rpm -qa | grep glibc命令查看glibc的版本，如果低于2.14版本，那么需要使用mariadb-enterprise-10.1.10-linux-x86\_64.tar.gz二进制安装包。另外，MariaDB-YUM源也提供了TokuDB插件。

笔者使用的系统为Centos 6.6，glibc版本如图2-96所示。

```
[root@slave1 ~]# rpm -qa | grep glibc
glibc-headers-2.12-1.149.el6_6.9.x86_64
glibc-devel-2.12-1.149.el6_6.9.x86_64
glibc-common-2.12-1.149.el6_6.9.x86_64
glibc-2.12-1.149.el6_6.9.x86_64
[root@slave1 ~]#
```



图2-96 查看glibc版本号

这里需要用YUM安装MariaDB，命令如下：

```
# rpm -i
http://downloads.mariadb.com/enterprise/sf99-qnvk/generate/10.1/mariadb-enterprise-repository.rpm
# yum install MariaDB-server MariaDB-client
```

从官方下载MariaDB企业版的yum源，然后进行安装，如图2-97所示。

开启TokuDB的命令如下：

```
MariaDB > install soname 'ha_tokudb';
```

具体如图2-98所示。

在/data/mysql目录查看数据文件的情况如图2-99所示。

创建TokuDB引擎表，如图2-100所示。

依赖关系解决

软件包	架构	版本	仓库	大小
正在安装:				
MariaDB-client	x86_64	10.1.10-1.el6	mariadb-enterprise-main	40 M
replacing mysql.x86_64 5.1.73-5.el6_6				
MariaDB-compat	x86_64	10.1.10-1.el6	mariadb-enterprise-main	2.7 M
replacing mysql-libs.x86_64 5.1.73-5.el6_6				
MariaDB-devel	x86_64	10.1.10-1.el6	mariadb-enterprise-main	6.9 M
replacing mysql-devel.x86_64 5.1.73-5.el6_6				
MariaDB-server	x86_64	10.1.10-1.el6	mariadb-enterprise-main	103 M
为依赖而安装:				
MariaDB-common	x86_64	10.1.10-1.el6	mariadb-enterprise-main	42 k
galera	x86_64	25.3.9-1.rhel6.el6	mariadb-enterprise-main	8.9 M
libpcap	x86_64	14:1.4.0-4.20130826git2dbcaal.el6	base	131 k
nc	x86_64	1.84-24.el6	base	57 k
nmap	x86_64	2:5.51-4.el6	base	2.8 M

事务概要

Install

9 Package(s)

总下载量: 165 M

确定吗? [y/N]: y

下载软件包:

[1/9]: MariaDB-10.1.10-centos6-x86\_64-client.rpm

(1%) 4% [==-

] 235 kB/s | 1.9 MB

02:46 ETA



图2-97 使用YUM源安装MariaDB

```

MariaDB [(none)]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

MariaDB [(none)]> install soname 'ha_tokudb';
Query OK, 0 rows affected (0.80 sec)

MariaDB [(none)]> show engines;
+-----+-----+-----+-----+-----+-----+
| Engine | Support | Comment | Transactions | XA | Savepoints |
+-----+-----+-----+-----+-----+-----+
| MRG_MyISAM | YES | Collection of identical MyISAM tables | NO | NO | NO |
| CSV | YES | CSV storage engine | NO | NO | NO |
| Aria | YES | Crash-safe tables with MyISAM heritage | NO | NO | NO |
| MyISAM | YES | MyISAM storage engine | NO | NO | NO |
| TokuDB | YES | Percona TokuDB Storage Engine with Fractal Tree(tm) Technology | YES | YES | YES |
| MEMORY | YES | Hash based, stored in memory, useful for temporary tables | NO | NO | NO |
| InnoDB | DEFAULT | Percona-XtraDB, Supports transactions, row-level locking, foreign keys and encryption for tables | YES | YES | YES |
| SEQUENCE | YES | Generated tables filled with sequential values | YES | NO | YES |
| PERFORMANCE_SCHEMA | YES | Performance Schema | NO | NO | NO |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)

```

图2-98 安装TokuDB引擎插件

```

[root@slave1 mysql]# pwd
/data/mysql
[root@slave1 mysql]# ll -h | grep tokudb
-rw-rw---- 1 mysql mysql 32K 2月 9 22:57 _test_t1_tokudb_main_3_2_1c.tokudb
-rw-rw---- 1 mysql mysql 16K 2月 9 22:57 _test_t1_tokudb_status_3_1_1c.tokudb
-rw-rw---- 1 mysql mysql 32K 2月 9 22:57 tokudb.directory
-rw-rw---- 1 mysql mysql 16K 2月 9 22:51 tokudb.environment
-rw-rw---- 1 mysql mysql 0 2月 9 22:51 __tokudb_lock_dont_delete_me_data
-rw-rw---- 1 mysql mysql 0 2月 9 22:51 __tokudb_lock_dont_delete_me_environment
-rw-rw---- 1 mysql mysql 0 2月 9 22:51 __tokudb_lock_dont_delete_me_logs
-rw-rw---- 1 mysql mysql 0 2月 9 22:51 __tokudb_lock_dont_delete_me_recovery
-rw-rw---- 1 mysql mysql 0 2月 9 22:51 __tokudb_lock_dont_delete_me_temp
-rw-rw---- 1 mysql mysql 32K 2月 9 22:57 tokudb.rollback

```

图2-99 查看TokuDB数据文件

3 ) 参数调优。

TokuDB调优比InnoDB简单得多，最重要的参数是tokudb\_cache\_size，它类似于innodb\_buffer\_pool\_size，其他参数使用默认值就可

以，运行良好。但如果你的服务器上运行着两种引擎InnoDB和TokuDB，那么就得小心了，别把内存用光了。

```
MariaDB [test]> create table t1_tokudb(id int primary key, name varchar(10)) engine=tokudb;
Query OK, 0 rows affected (0.58 sec)

MariaDB [test]> insert into t1_tokudb values(1,'hechunyang');
Query OK, 1 row affected (0.08 sec)

MariaDB [test]> select * from t1_tokudb;
+----+-----+
| id | name |
+----+-----+
| 1  | hechunyang |
+----+-----+
1 row in set (0.06 sec)
```

图2-100 创建TokuDB引擎表

此外，tokudb\_row\_format参数有以下几个值可以进行设置

注

。

- tokudb\_fast：表示使用quicklz库的压缩模式。
- tokudb\_small：表示使用lzma库的压缩模式。
- tokudb\_zlib：表示使用zlib库的压缩模式，提供了中等级别的压缩比和中等级别的CPU消耗。
- tokudb\_quicklz：表示使用quicklz库的压缩模式，提供了轻量级的压缩比和较低的CPU消耗。
- tokudb\_lzma：表示使用lzma库压缩模式，提供了高压比和高CPU消耗。
- tokudb\_uncompressed：表示不使用压缩模式。



注意 在生产环境上很少有公司将核心业务跑在TokuDB引擎上面，目前的适用场景：可以把一些LOG日志表改为TokuDB引擎，或者将Zabbix监控涉及的大表改为TokuDB引擎，这样在性能和磁盘空间使用率上都有较大幅度的提升。此外，该引擎的备份工具是收费的，所以备份的时候可以采取冷备份。

- 默认使用zlib库进行压缩，未来的版本可能会改变。



## 2.7 优化器改进

### 2.7.1 针对子查询select采用半连接优化

MySQL的子查询一直以来以性能差著称，所以解决的方案是用关联（join）查询代替子查询。子查询在MySQL里，仅看成一个功能，生产环境下很少使用。但如今在MySQL 5.6/5.7和MariaDB 10.0/10.1版本里，子查询终于有了强劲优化，这意味着不改变原有SQL的情况下，通过MySQL内部的优化器把子查询改写成为关联查询。

通常情况下，我们希望由内到外，先完成内表里的查询结果，然后驱动外查询的表，完成最终查询，但是MySQL 5.5版本会先扫描外表中的所有数据，每条数据将会传到内表中与之关联，如果外表很大，那么性能上将会很差。

让我们看一个例子，代码如下：

```
select * from Country where  
    Continent='Europe' and  
    Country.Code in (select City.country  
        from City  
        where City.Population>1*1000*1000);
```

在MySQL 5.5版本里，首先执行外表Country把符合欧洲国家的数据过滤出来，然后将每个符合条件的数据都与内表City进行一次select City.country from City where City.Population>1\*1000\*1000查询，故而性能低下，如图2-101所示。



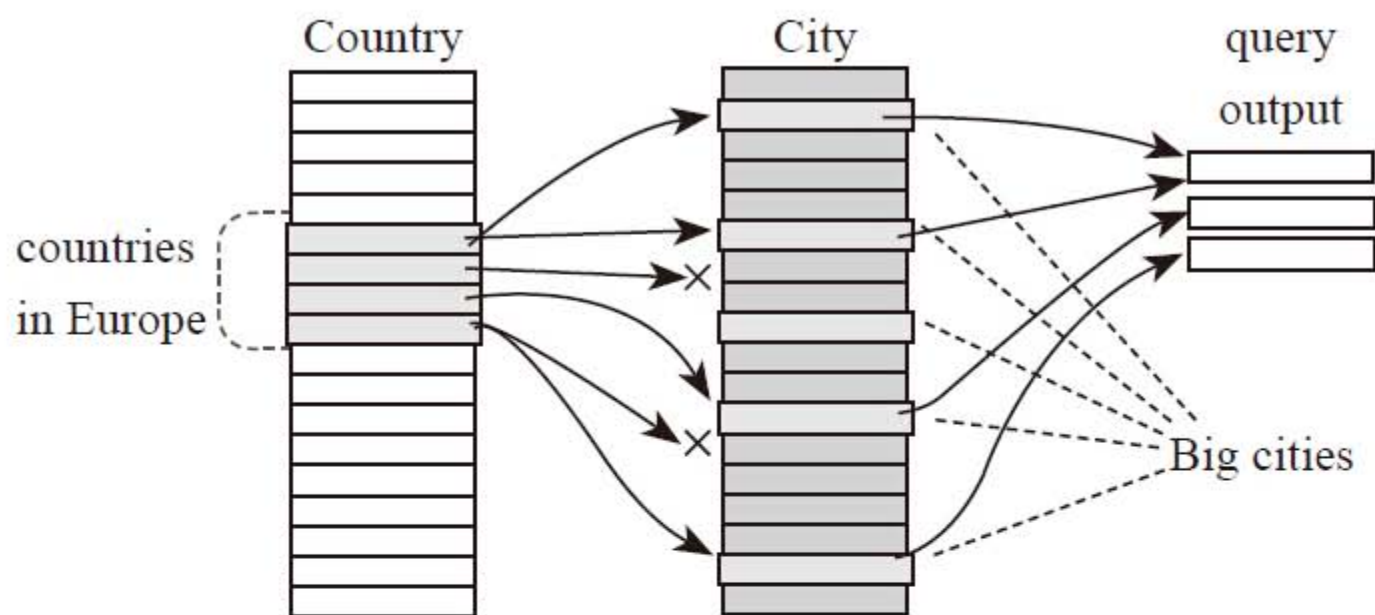


图2-101 MySQL 5.5子查询的执行过程

通过半连接 (semi join) 优化后，执行计划器是这样工作的。

首先，执行内表city，把符合城市人口数量大于100万的数据过滤出来，外表只需要在内表的查询结果中找到匹配的记录即可，这样大大提高了工作效率，如图2-102所示。

该SQL会通过内部优化器重写为：

```
select a.* from Country as a
  join (select City.country from City where City.Population > 1*1000*1000) as b
 on a.Code = b.country
 where a.Continent = 'Europe';
```

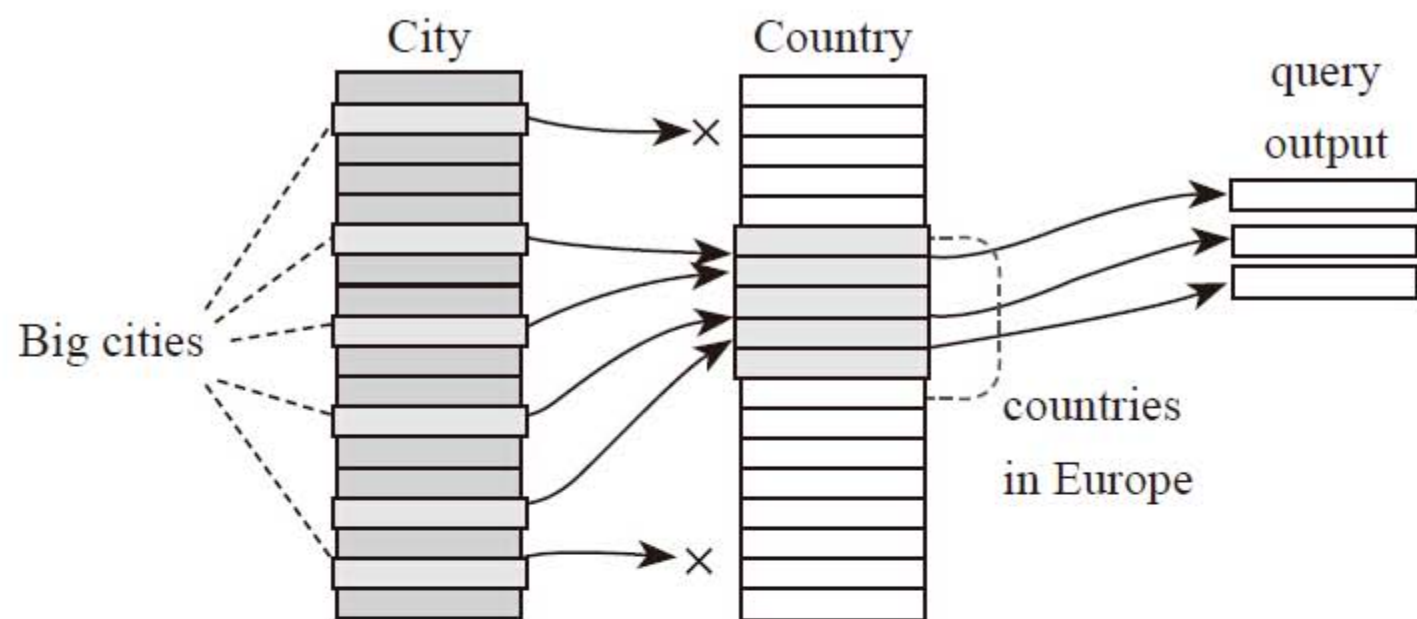


图2-102 MySQL 5.7子查询的执行过程

需要注意的是，MySQL 5.5的子查询执行计划是将in重写为exists，如图2-103所示。

MySQL 5.6/5.7和MariaDB 10.0/10.1的子查询执行计划是将in/exists重写为join，如图2-104所示。

MySQL [test]> explain select \* from sbtest where id in (select id from t1);

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	sbtest	ALL	NULL	NULL	NULL	NULL	5000071	Using where
2	DEPENDENT SUBQUERY	t1	unique_subquery	PRIMARY	PRIMARY	4	func	1	Using index; Using where

2 rows in set (0.00 sec)

MySQL [test]> explain select \* from sbtest where exists (select \* from t1 where t1.id=sbtest.id);

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	sbtest	ALL	NULL	NULL	NULL	NULL	5000071	Using where
2	DEPENDENT SUBQUERY	t1	eq_ref	PRIMARY	PRIMARY	4	test.sbtest.id	1	Using where; Using index

2 rows in set (0.02 sec)

MySQL [test]> select version();

version()
5.5.46-log

1 row in set (0.00 sec)

图2-103 将in重写为exists



```

MariaDB [test]> explain select * from sbtest where id in (select id from t1);
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | t1 | index | PRIMARY | PRIMARY | 4 | NULL | 10 | Using index |
| 1 | PRIMARY | sbtest | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.id | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

MariaDB [test]> explain select * from sbtest where exists (select * from t1 where t1.id=sbtest.id);
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | t1 | index | PRIMARY | PRIMARY | 4 | NULL | 10 | Using index |
| 1 | PRIMARY | sbtest | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.id | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

MariaDB [test]> explain select a.* from sbtest a join t1 b on a.id=b.id;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | b | index | PRIMARY | PRIMARY | 4 | NULL | 10 | Using index |
| 1 | SIMPLE | a | eq_ref | PRIMARY | PRIMARY | 4 | test.b.id | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

MariaDB [test]> select version();
+-----+
| version() |
+-----+
| 10.0.20-MariaDB-log |
+-----+
1 row in set (0.02 sec)

```

图2-104 将in重写为join

半连接子查询优化默认开启，可以通过语句show variables like 'optimizer\_switch'\G;来查看：

```

mysql> show variables like 'optimizer_switch'\G;
***** 1. row *****

```

Variable\_name: optimizer\_switch

Value: index\_merge=on, index\_merge\_union=on, index\_merge\_sort\_union=on, index\_merge\_intersection=on, engine\_condition\_pushdown=on, index\_condition\_pushdown=on, mrr=on, mrr\_cost\_based=on, block\_nested\_loop=on,

*batched\_key\_access=on, materialization=on, semijoin=on, loosescan=on, firstmatch=on, duplicateweedout=on, subquery\_materialization\_cost\_based=on, use\_index\_extensions=on, condition\_fanout\_filter=on, derived\_merge=on*  
1 row in set (0.00 sec)

但是，半连接优化只是针对查询的，针对update/delete的性能仍旧很差。下面通过实验来验证执行语句explain delete from sbtest where id in(select id from t1)，从图2-105中可以看到半连接优化失效了，仍旧还是先查外表再关联内表。

```
mysql> explain delete from sbtest where id in (select id from t1);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	DELETE	sbtest	NULL	ALL	NULL	NULL	NULL	NULL	986400	100.00	Using where
2	DEPENDENT SUBQUERY	t1	NULL	unique_subquery	PRIMARY	PRIMARY	4	func	1	100.00	Using where; Using index

2 rows in set (0.00 sec)

```
mysql> select version();
```

version()
5.7.10-log

1 row in set (0.03 sec)

```
MariaDB [test]> explain delete from sbtest where id in (select id from t1);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	sbtest	ALL	NULL	NULL	NULL	NULL	889247	Using where
2	DEPENDENT SUBQUERY	t1	unique_subquery	PRIMARY	PRIMARY	4	func	1	Using index; Using where

2 rows in set (0.02 sec)

```
MariaDB [test]> select version();
```

version()
10.1.10-MariaDB-enterprise-log

1 row in set (0.00 sec)

图2-105 半连接优化失效

因此，在生产环境中，一定要多加注意！针对update/delete的子查询仍需人工改写为关联查询。



## 2.7.2 优化派生子查询

只在MySQL 5.6/5.7、Percona 5.6/5.7和MariaDB 10.0/10.1版本里支持优化派生子查询这项功能。

### 1. 派生表合并 ( derived table merge ) 优化

先看一个例子，想要查找出人口数量大于1万的城市，并且哪些城市是属于德国的，SQL语句如下：

```
SELECT *  
FROM  
    (SELECT * FROM City WHERE Population > 10*1000) AS big_city  
WHERE  
    big_city.Country='DEU'
```

在MySQL 5.5版本里，首先执行内表找出人口大于1万的城市，查出的结果保存在临时表里。然后在临时表里过滤出哪些城市属于德国，如图2-106所示。

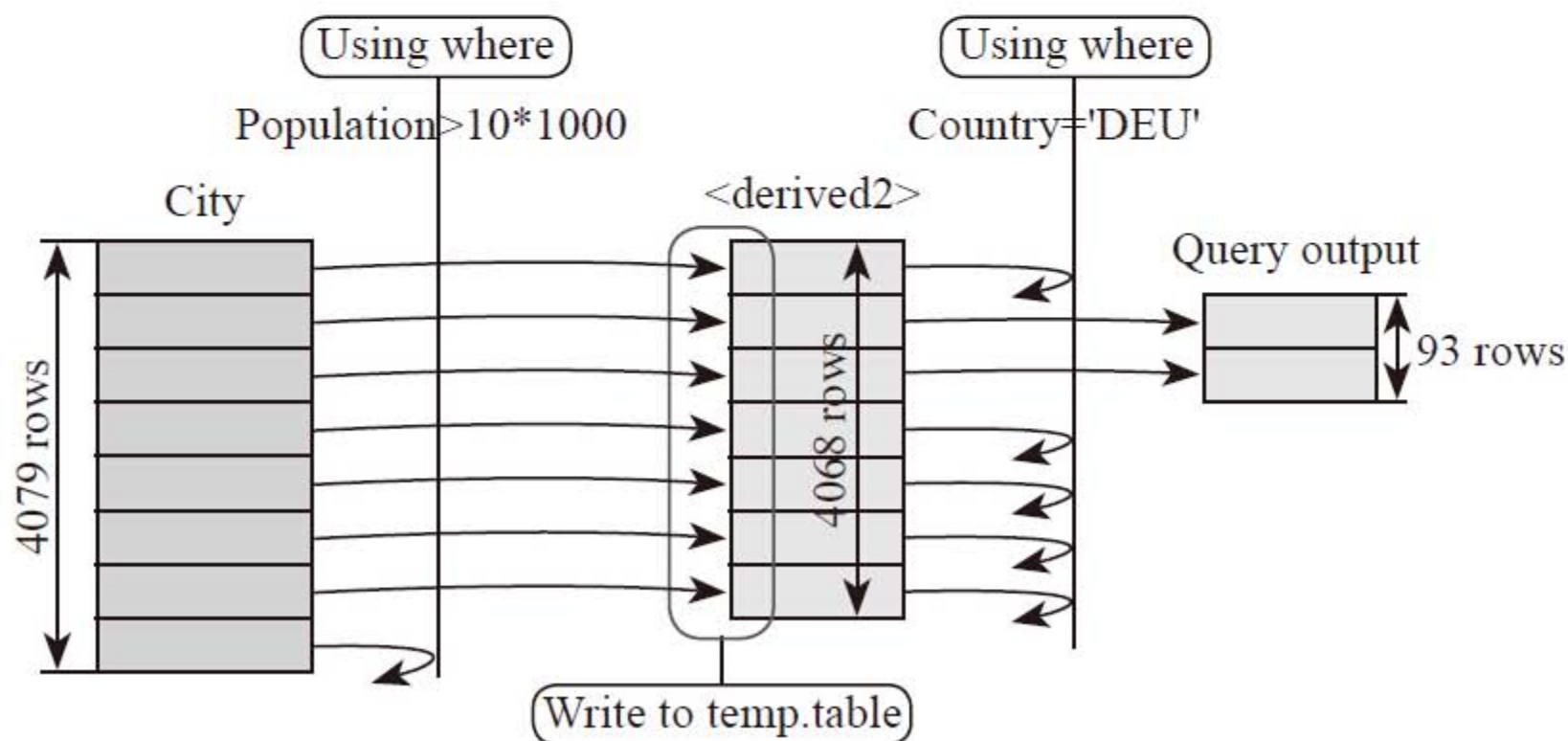


图2-106 在MySQL 5.5版本中派生子查询执行过程

由于对City表读取了大量的记录并且再次筛选过滤出来的临时表记录，这种重复性的工作使得性能非常低下，可通过内部优化器查看执行过程，如图2-107所示。

```
mysql> EXPLAIN SELECT * FROM (SELECT * FROM City WHERE Population > 1*1000) AS big_city WHERE big_city.Country='DEU';↓
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	4068	Using where
2	DERIVED	City	ALL	Population	NULL	NULL	NULL	4079	Using where

2 rows in set (0.60 sec)↓

图2-107 派生表优化前的执行过程

通过派生表合并 (derived table merge) 优化后，执行计划器是这样工作的，如图2-108所示。

```
MariaDB [world]> EXPLAIN SELECT * FROM (SELECT * FROM City WHERE Population > 1*1000) AS big_city WHERE big_city.Country='DEU';↓
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	City	ref	Population, Country	Country	3	const	90	Using index condition; Using where

1 row in set (0.00 sec)↓

图2-108 派生表优化后的执行过程

从上面的输出结果可以看到：

- 派生表已经被合并到父表里。
- 查找属于德国的所有城市，利用索引Country进行检索。
- 过滤大于1万人口的城市，整个查询仅扫描了90行就显示了结果，相比之前从临时表4068行里进行筛选，性能大大提升。

SQL会通过内部优化器重写为：

```
SELECT * FROM City WHERE Country='DEU' AND Population > 10*1000;
```

通过图2-109所示的例子也可以看出来。



```
mysql> explain select * from (select * from sbtest where k=0) tmp where tmp.id>=10 and tmp.id<=20;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sbtest	NULL	range	PRIMARY,idx_k	PRIMARY	4	NULL	11	100.00	Using where

```
1 row in set, 1 warning (0.00 sec)
```

```
mysql> explain select * from sbtest where (id>=10 and id <=20) and k=0;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sbtest	NULL	range	PRIMARY,idx_k	PRIMARY	4	NULL	11	100.00	Using where

```
1 row in set, 1 warning (0.00 sec)
```

```
mysql> select version();
```

version()
5.7.10-log

```
1 row in set (0.00 sec)
```

图2-109 派生表优化后的执行过程

派生表合并 ( derived table merge ) 优化默认开启，可以通过语句show variables like 'optimizer\_switch'\G;来查看：

```
mysql> show variables like 'optimizer_switch'\G;
```

```
***** 1. row *****
```

Variable\_name: optimizer\_switch

Value: index\_merge=on , index\_merge\_union=on , index\_merge\_sort\_union=on , index\_merge\_intersection=on , engine\_condition\_pushdown=on , index\_condition\_pushdown=on , mrr=on , mrr\_cost\_based=on , block\_nested\_loop=on , batched\_key\_access=on , materialization=on , semijoin=on , loosescan=on , firstmatch=on , duplicateweedout=on , subquery\_materialization\_cost\_based=on , use\_index\_extensions=on , condition\_fanout\_filter=on , derived\_merge=on

```
1 row in set (0.00 sec)
```

## 2.派生表索引 ( derived table with key ) 优化

如果派生表不能合并到父表查询，则会把查询的结果存放在一张临时表里，在MySQL 5.5版本里，临时表是无法使用到索引的，只能全表扫描。在MySQL 5.6/5.7和MariaDB 5.3/5.5/10.0/10.1版本里，可采用索引将子查询的结果存进一张临时表，并且利用这张临时表来执行一个连接。



下面看一个例子，执行以下SQL语句：

*explain select \* from sbtest a, (select sum(id) id from sbtest group by k) as b where a.id=b.id and k=1;*

MySQL 5.5版本的执行结果如图2-110所示。

可以看到<derived2>临时表没有使用索引，而是进行了全表扫描。

```
mysql> explain select * from sbtest a, (select sum(id) id from sbtest group by k) as b where a.id=b.id and k=1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	11	
1	PRIMARY	a	eq_ref	PRIMARY,k	PRIMARY	4	b.id	1	Using where
2	DERIVED	sbtest	index	NULL	k	4	NULL	1000075	Using index

3 rows in set (0.24 sec)

```
mysql> select version();
```

version()
5.5.46

1 row in set (0.00 sec)

图2-110 MySQL 5.5版本中派生表优化前的执行过程

MySQL 5.7的执行结果如图2-111所示。

```
mysql> explain select * from sbtest a, (select sum(id) id from sbtest group by k) as b where a.id=b.id and a.k=1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	a	NULL	ref	PRIMARY,idx_k	idx_k	4	const	1	100.00	NULL
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	16	test.a.id	10	100.00	Using where; Using index
2	DERIVED	sbtest	NULL	index	idx_k	idx_k	4	NULL	981507	100.00	Using index

3 rows in set, 1 warning (0.00 sec)

```
mysql> select version();
```

version()
5.7.10-log

1 row in set (0.48 sec)

图2-111 MySQL 5.7版本中派生表优化后的执行过程

可以看到<derived2>临时表自动创建了索引id。

### 2.7.3 优化排序limit

只有在MySQL 5.6/5.7、Percona 5.6/5.7和MariaDB 10.0/10.1版本里支持优化排序（order by）limit这项功能。

在MySQL 5.5版本中，对sbtest表非索引字段进行排序（order by），耗费的时间如图2-112所示。

结果耗时是3分14.78秒。

下面再在MySQL 5.6版本中同样执行这条SQL语句，看看耗费时间为多少，如图2-113所示。

结果耗时是12.53秒。

这是为什么呢？因为在MySQL 5.5版本里对非索引字段排序时，会对表的所有记录进行一次排序操作，然后取出limit 10条记录，余下的记录抛弃。而在MySQL 5.6版本里，只针对limit 10条记录进行排序，余下的记录并不会进行排序，这样就加快了速度，提升了性能。



```
mysql> select * from sbtest order by RAND() desc limit 10;
```

id	k	c	pad
889277	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
656389	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
54426	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
516429	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
854468	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
101418	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
959902	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
945586	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
460104	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt
231384	0		qqqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt

```
10 rows in set (3 min 14.78 sec)
```

```
mysql> select version();
```

version()
5.5.31-0+wheezy1-log

```
1 row in set (0.04 sec)
```

图2-112 优化前的排序时间

```
mysql> select * from sbtest order by RAND() desc limit 10;
```

id	k	c	pad
965153	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
497595	1		aaaaaaaaaaffffffffrrrrrrrrrrreeeeeeeeeeyyyyyyyyy
964875	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
65631	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
960898	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
780991	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
156533	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
283021	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
669436	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt
951731	0		qqqqqqqqqqwwwwwwwwweeeeeeeeerrrrrrrrrrttttttttt

```
10 rows in set (12.26 sec)

mysql> select version();
```

version()
5.6.15-rel63.0-log

```
1 row in set (0.09 sec)
```

图2-113 优化后的排序时间

## 2.7.4 优化IN条件表达式

当执行如下SQL语句时，在MySQL 5.6和Maria DB10.0/10.1版本里会进行全表扫描，无法用到索引：

```
select * from sbtest where (id, k) in ((11, 0), (22, 0));
```

sbtest表结构如下：

```
CREATE TABLE `sbtest` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,  
  `k` int(10) unsigned NOT NULL DEFAULT '0',  
  `c` char(120) NOT NULL DEFAULT '',  
  `pad` char(60) NOT NULL DEFAULT '',  
  PRIMARY KEY (`id`),  
  KEY `k` (`k`)  
) ENGINE=InnoDB AUTO_INCREMENT=1000001  
DEFAULT CHARSET=utf8
```

在MariaDB 10.1版本中的执行结果如图2-114所示。

```
MariaDB [test]> explain select * from sbtest where (id,k) in ((11,0),(22,0));
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sbtest	ALL	NULL	NULL	NULL	NULL	986400	Using where

```
1 row in set (0.00 sec)
```

```
MariaDB [test]> select version();
```

version()
10.1.10-MariaDB-enterprise-log

```
1 row in set (0.00 sec)
```

图2-114 在MariaDB 10.1版本中的执行结果

虽然id是主键，k是索引，但执行以后仍然是全表扫描。



在MySQL 5.7版本中的执行结果如图2-115所示。

```
MySQL [test]> explain select * from sbtest where (id,k) in ((11,0), (22,0));
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sbtest | NULL | range | PRIMARY,k | PRIMARY | 4 | NULL | 2 | 20.00 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

MySQL [test]> select version();
+-----+
| version() |
+-----+
| 5.7.10 |
+-----+
1 row in set (0.00 sec)
```

图2-115 在MySQL 5.7版本中的执行结果

可以看到在MySQL 5.7版本中已经使用了索引。这是因为该SQL语句已经被内部的优化器改写为如图2-116所示的形式。

而MySQL 5.6和MariaDB 10.0/10.1版本内部的优化器无法改写，只能进行全表扫描，造成性能低下。不过可通过人工改写，再来对比修改后的在MariaDB 10.1版本中的执行结果，如图2-117所示。

改写后的SQL语句只需扫描2行就可显示结果，与MySQL 5.7版本内部优化器的性能差不多，再查看查询结果，如图2-118所示。

```
MySQL [test]> explain select * from sbtest where (id=11 and k=0) or (id=22 and k=0);
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sbtest | NULL | range | PRIMARY,k | PRIMARY | 4 | NULL | 2 | 19.00 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

MySQL [test]> select version();
+-----+
| version() |
+-----+
| 5.7.10 |
+-----+
1 row in set (0.00 sec)
```

图2-116 在MySQL 5.7版本中的执行计划

MariaDB [test]> explain select \* from sbtest where (id=11 and k=0) or (id=22 and k=0);

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sbtest	range	PRIMARY,k	PRIMARY	4	NULL	2	Using where

1 row in set (0.00 sec)

MariaDB [test]> explain select \* from sbtest where (id,k) in ((11,0),(22,0));

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sbtest	ALL	NULL	NULL	NULL	NULL	986400	Using where

1 row in set (0.00 sec)

MariaDB [test]> select version();

version()
10.1.10-MariaDB-enterprise-log

1 row in set (0.00 sec)

图2-117 在MariaDB 10.1版本中优化后的执行计划

1 row in set (0.00 sec)

改写后的SQL语句执行0秒就可显示结果。这个特性来自Google/Facebook的补丁。



### 2.7.5 优化union all

在MySQL 5.7和MariaDB 10.1版本里，union all不再创建一张临时表，这在执行大的联合查询时会减少I/O开销，提升查询速度。但对union和在最外层使用order by无效。

例如执行以下SQL语句：

*explain (select id from t1 order by id) union all (select id from sbtest where k=0 order by id );*

在MySQL 5.5版本中的执行结果如图2-119所示。

```
MySQL [test]> explain (select id from t1 order by id) union all (select id from sbtest where k=0 order by id );
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t1	index	NULL	PRIMARY	4	NULL	10	Using index
2	UNION	sbtest	ref	k	k	4	const	500053	Using index
NULL	UNION RESULT	<union1,2>	ALL	NULL	NULL	NULL	NULL	NULL	

3 rows in set (0.00 sec)

```
MySQL [test]> select version();
```

version()
5.5.46-log

1 row in set (0.00 sec)

图2-119 在MySQL 5.5版本中的执行结果

从执行结果来看，是把t1表的查询结果和sbtest表的查询结果合并在了了一张临时表里，然后输出给客户端。

在MySQL 5.7版本中的执行结果如图2-120所示。

MySQL [test]> explain (select id from t1 order by id) union all (select id from sbtest where k=0 order by id );

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	NULL	index	NULL	PRIMARY	4	NULL	10	100.00	Using index
2	UNION	sbtest	NULL	ref	k	k	4	const	493200	100.00	Using index

2 rows in set, 1 warning (0.00 sec)

MySQL [test]> select version();

version()
5.7.10

1 row in set (0.00 sec)

图2-120 在MySQL 5.7版本中的执行结果

从执行结果来看，没有创建临时表，按照顺序，t1表的查询结果首先输出到客户端，然后sbtest表的查询结果再输出到客户端。但对union和在最外层使用order by无效，如图2-121所示。

MySQL [test]> explain (select id from t1 order by id) union all (select id from sbtest where k=0 order by id) order by id desc;

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	NULL	index	NULL	PRIMARY	4	NULL	10	100.00	Using index
2	UNION	sbtest	NULL	ref	k	k	4	const	493200	100.00	Using index
NULL	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary; Using filesort

3 rows in set, 1 warning (0.00 sec)

MySQL [test]> explain (select id from t1 order by id) union (select id from sbtest where k=0 order by id);

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	NULL	index	NULL	PRIMARY	4	NULL	10	100.00	Using index
2	UNION	sbtest	NULL	ref	k	k	4	const	493200	100.00	Using index
NULL	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary

3 rows in set, 1 warning (0.00 sec)

MySQL [test]> select version();

version()
5.7.10

1 row in set (0.00 sec)

图2-121 对union和在最外层使用order by无效



## 2.7.6 支持索引下推优化

支持索引下推优化 ( Index Condition Pushdown , ICP ) 这个特性是MariaDB5.3/MySQL5.6引进的，在解释该特性之前，先看下面的一个例子，该示例的表结构如图2-122所示。

```
mysql> show create table student\G;
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) NOT NULL DEFAULT '0',
  `name` varchar(6) DEFAULT NULL,
  `class` int(11) DEFAULT NULL,
  `score` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `name` (`name`),
  KEY `IX_C_S` (`class`,`score`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec) |
```

图2-122 student表结构

在图2-122中，class和score为联合索引。

在MySQL 5.5版本中未使用ICP的执行计划如图2-123所示。

在MySQL 5.5版本中，未开启索引下推优化，所以首先会根据class=1来查找记录，检索的结果将指向聚集索引，最后根据score>60进行过滤，把最终的结果返回给用户。所以这里看到的是using where，如图2-124所示。

在MySQL 5.6版本中使用ICP的执行计划如图2-125所示。

mysql> select version();

version()
5.5.20-enterprise-commercial-advanced-log

1 row in set (0.00 sec)

mysql> explain select \* from student where class=1 and score > 60;

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	student	range	IX_C_S	IX_C_S	10	NULL	5	Using where

1 row in set (0.00 sec)

图2-123 在MySQL 5.5版本中未使用的ICP执行计划

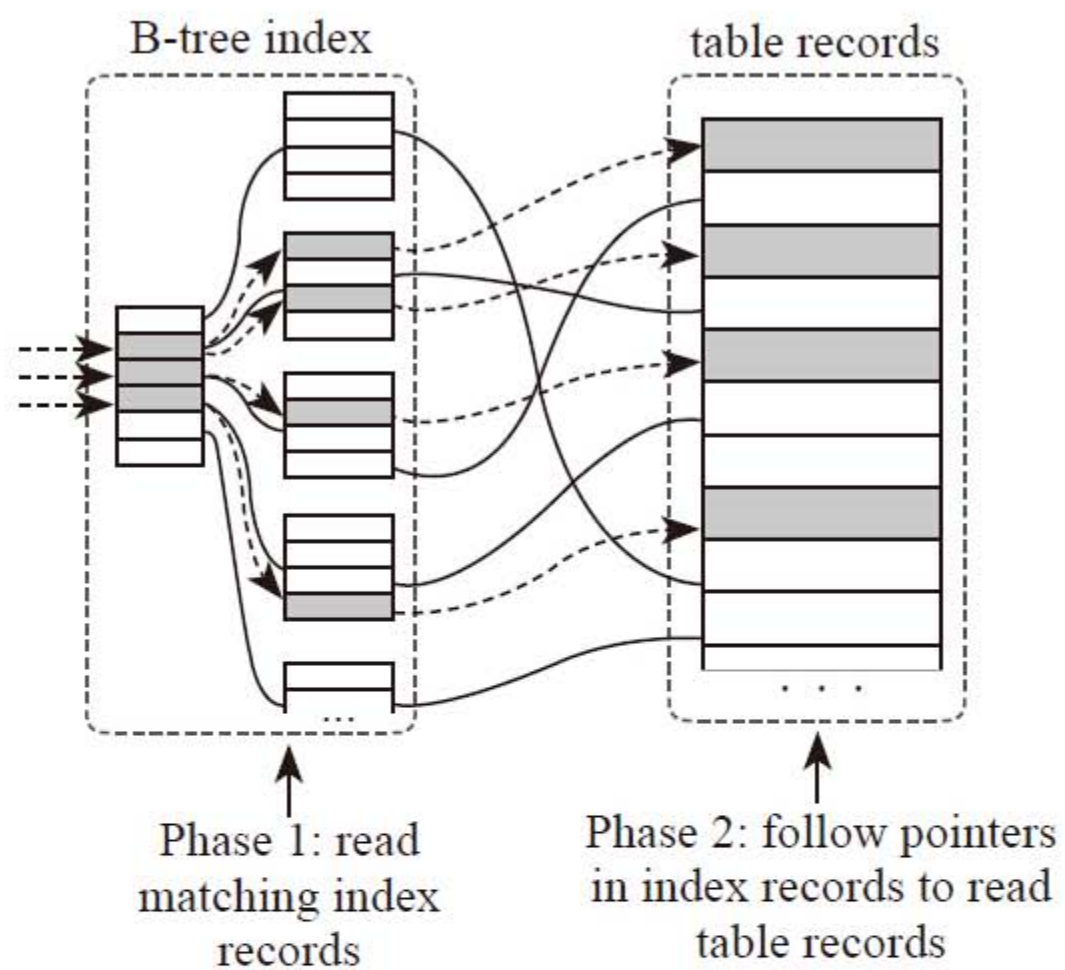


图2-124 未使用ICP的内部原理示意图



```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.04 sec)

mysql> explain select * from student where class=1 and score > 60;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | range | IX_S_C | IX_S_C | 5 | NULL | 14 | Using index condition; Using MRR |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.04 sec)
```

图2-125 在MySQL 5.6版本中使用ICP的执行计划

在MySQL 5.6/5.7和MariaDB 5.3/5.5/10.0/10.1版本中，开启了索引下推优化，在class=1查找记录的同时，会根据score>60进行过滤，然后将检索的结果指向聚集索引，最后返回给用户。所以这里看到的是using index condition，ICP减少了存储引擎访问表的次数，从而提高了数据库的整体性能，如图2-126所示。

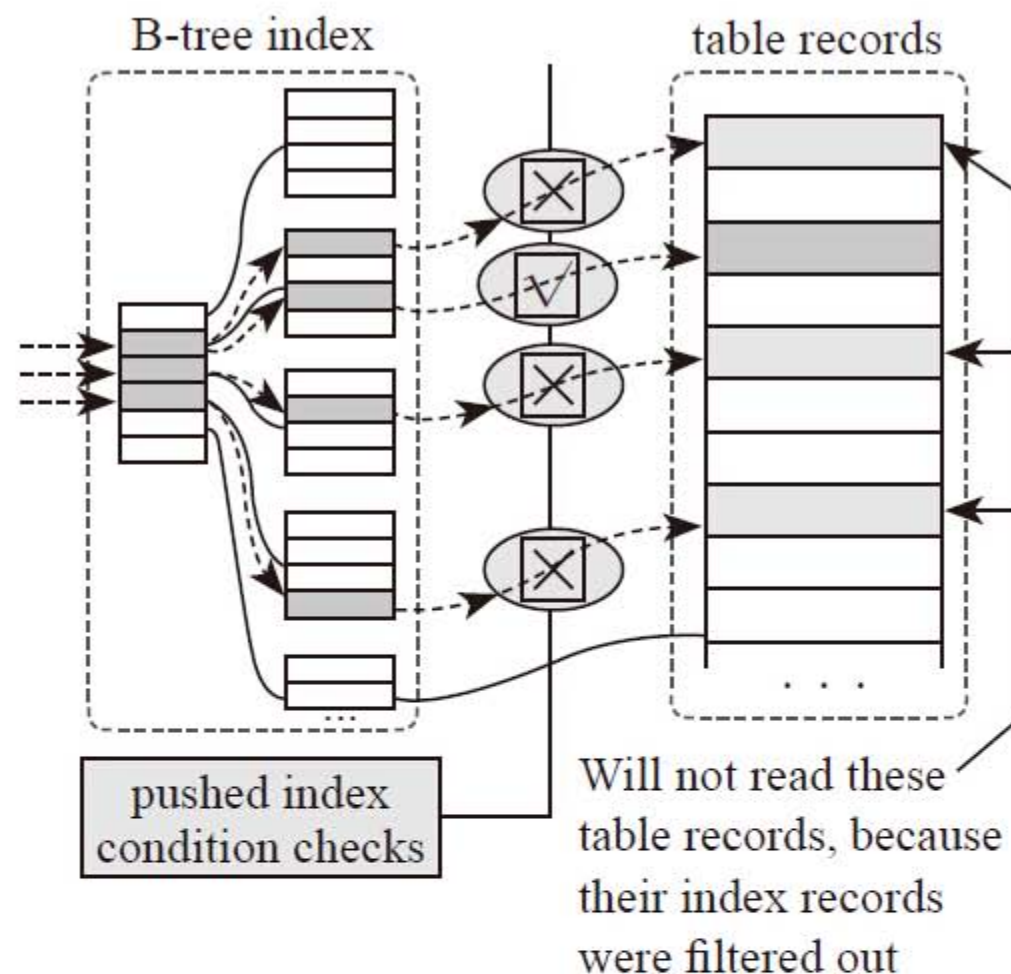


图2-126 使用ICP的内部原理示意图

索引下推优化默认开启，可以通过语句`show variables like 'optimizer_switch'\G`来查看：

```
mysql> show variables like 'optimizer_switch'\G;
```

```
***** 1. row *****
```

```
Variable_name: optimizer_switch
```

```
Value: index_merge=on, index_merge_union=on, index_merge_sort_union=on, index_merge_intersection=on,
engine_condition_pushdown=on, index_condition_pushdown=on, mrr=on, mrr_cost_based=on, block_nested_loop=on,
batched_key_access=on, materialization=on, semijoin=on, loosescan=on, firstmatch=on, duplicateweedout=on,
subquery_materialization_cost_based=on, use_index_extensions=on, condition_fanout_filter=on, derived_merge=on
1 row in set (0.00 sec)
```



## 2.7.7 支持Multi Range Read索引优化

对大表（基于辅助索引）进行范围扫描时，会导致产生许多随机I/O。而对于普通磁盘来说，随机I/O的性能很差，会遇到瓶颈，在MySQL 5.6/5.7和MariaDB 5.3/5.5/10.0/10.1版本里对这种情况进行了优化，一个新的名词Multi Range Read（MRR）出现了，优化器会先扫描索引，然后收集每行的主键，并对主键进行排序，此时就可以用主键顺序访问基表，即用顺序I/O代替随机I/O。

未开启MRR时，在explain中看到的情况如图2-127所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.01 sec)

mysql> explain select * from t_06 where i2 > 2000 and i2 < 4000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t_06 | range | ix_i2 | ix_i2 | 5 | NULL | 1997 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图2-127 未使用MRR时看到的情况

查询*i2*>2000 and *i2*<4000记录时，未开启MRR会产生随机I/O，如图2-128所示。

开启MRR后，在explain中看到的情况如图2-129所示。

查询*i2*>2000和*i2*<4000记录时，已开启MRR，故会对主键排序，将随机I/O转换为顺序I/O，从而提高数据库的整体性能，如图2-130所示。



针对这两种情况进行对比测试，分别如图2-131和图2-132所示。

从图2-131和图2-132中可以明显看到未开启MRR耗时更长，为1分47秒，开启MRR耗时为0.34秒。

Multi Range Read索引优化在MySQL 5.7版本中默认开启，可以通过语句show variables like'optimizer\_switch'\G;来查看：

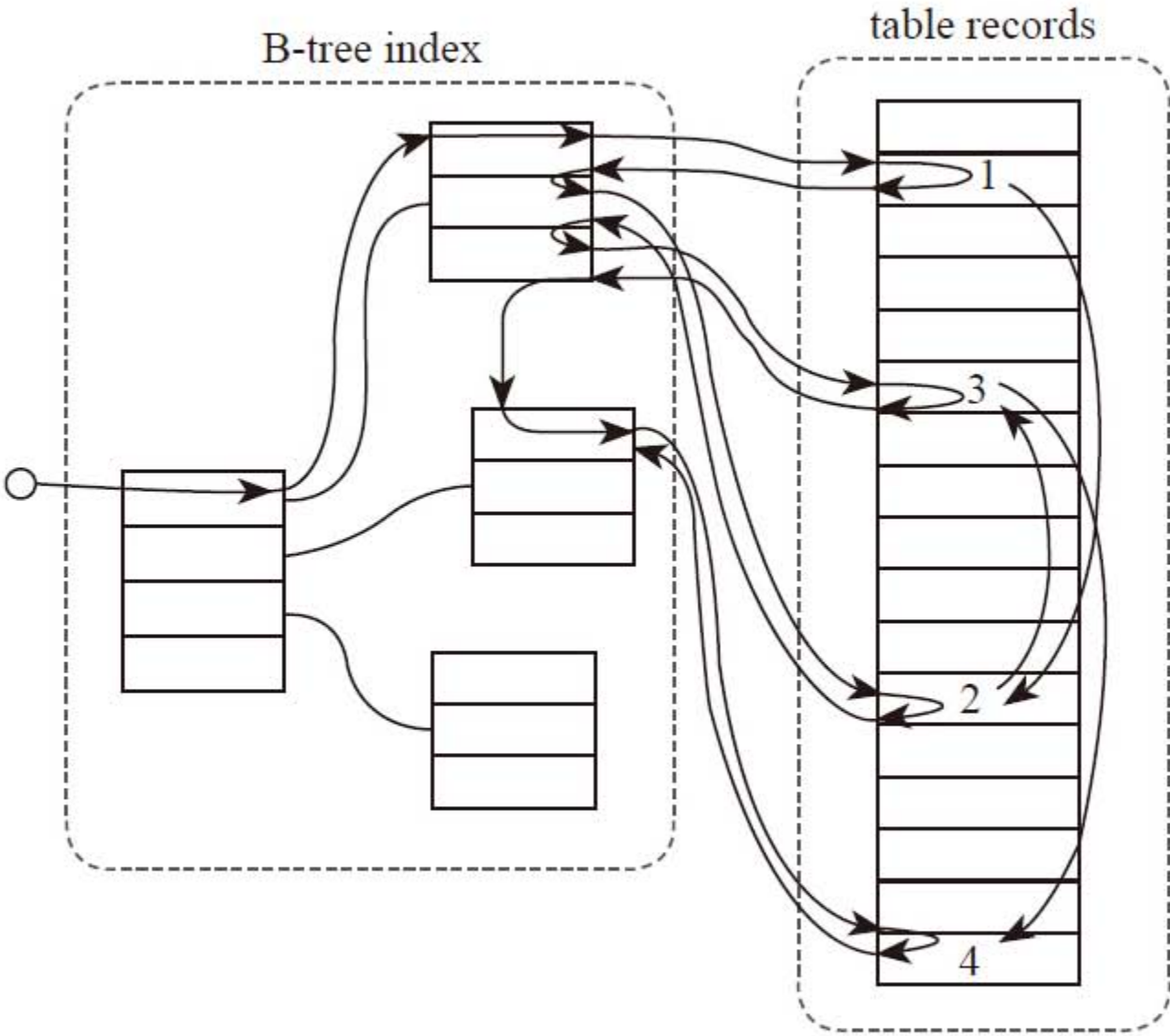


图2-128 未使用MRR的内部原理示意图

mysql> select version();

version()
5.6.6-m9-log

1 row in set (0.08 sec)

mysql> explain select \* from t\_06 where i2 > 2000 and i2 < 4000;

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t_06	range	ix_i2	ix_i2	5	NULL	1997	Using index condition; Using MRR

1 row in set (0.10 sec)

图2-129 已使用MRR时看到的情况

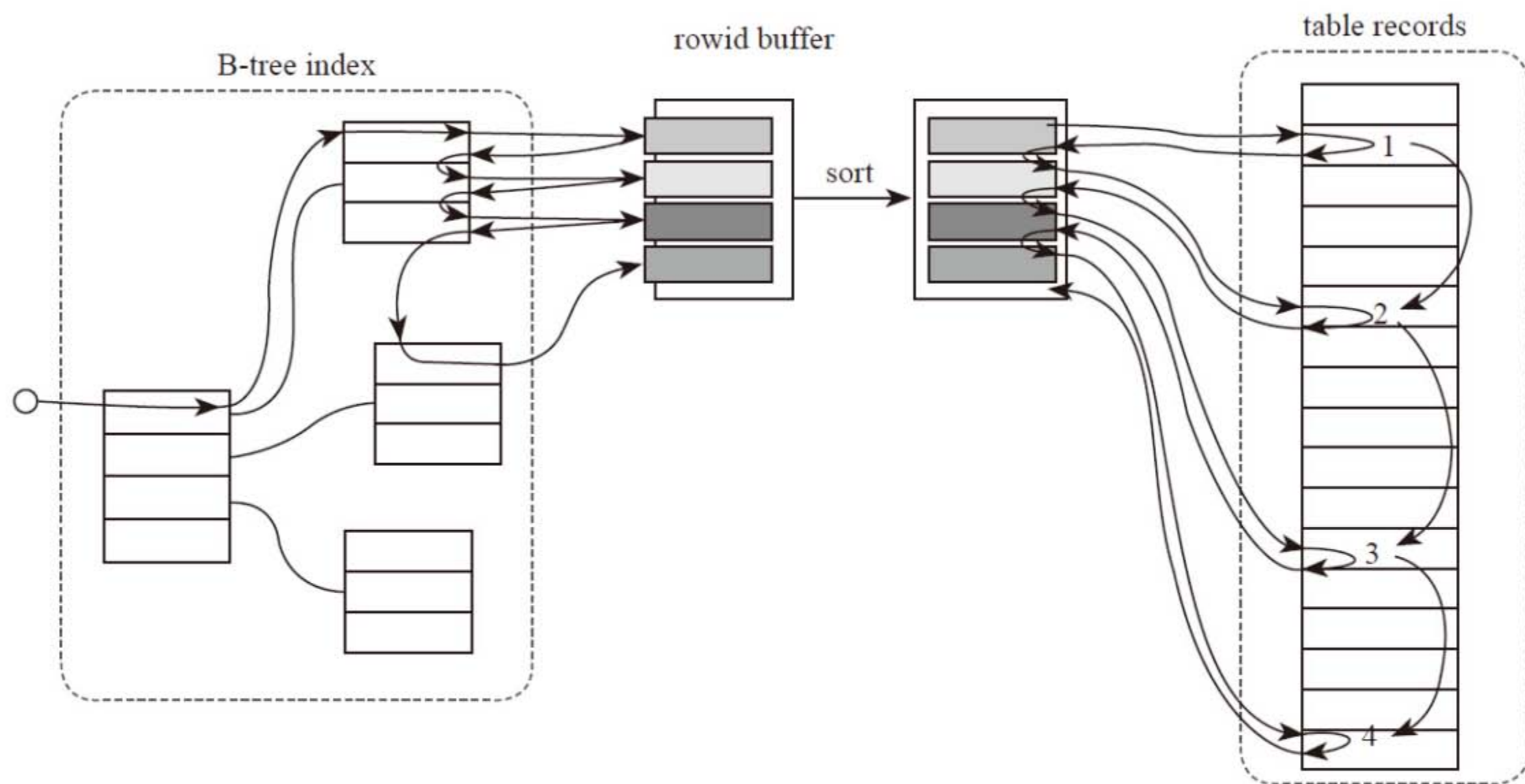


图2-130 使用MRR的内部原理示意图





Value: index\_merge=on , index\_merge\_union=on , index\_merge\_sort\_union=on , index\_merge\_intersection=on , engine\_condition\_pushdown=on , index\_condition\_pushdown=on , mrr=on , mrr\_cost\_based=on , block\_nested\_loop=on , batched\_key\_access=on , materialization=on , semijoin=on , loosescan=on , firstmatch=on , duplicateweedout=on , subquery\_materialization\_cost\_based=on , use\_index\_extensions=on , condition\_fanout\_filter=on , derived\_merge=on  
1 row in set (0.00 sec)

Multi Range Read索引优化在MariaDB 10.1版本中默认是关闭的，可以通过语句show variables like 'optimizer\_switch'\G;来查看：

Variable\_name: optimizer\_switch

Value: index\_merge=on , index\_merge\_union=on , index\_merge\_sort\_union=on , index\_merge\_intersection=on , index\_merge\_sort\_intersection=off , engine\_condition\_pushdown=off , index\_condition\_pushdown=on , derived\_merge=on , derived\_with\_keys=on , firstmatch=on , loosescan=on , materialization=on , in\_to\_exists=on , semijoin=on , partial\_match\_rowid\_merge=on , partial\_match\_table\_scan=on , subquery\_cache=on , mrr=off , mrr\_cost\_based=off , mrr\_sort\_keys=off , outer\_join\_with\_cache=on , semijoin\_with\_cache=on , join\_cache\_incremental=on , join\_cache\_hashed=on , join\_cache\_bka=on , optimize\_join\_buffer\_size=off , table\_elimination=on , extended\_keys=on , exists\_to\_in=on

动态开启MRR的命令如下：

set global optimizer\_switch='mrr=on , mrr\_cost\_based=on , mrr\_sort\_keys=on';

set global mrr\_buffer\_size = 32\*1024\*1024;

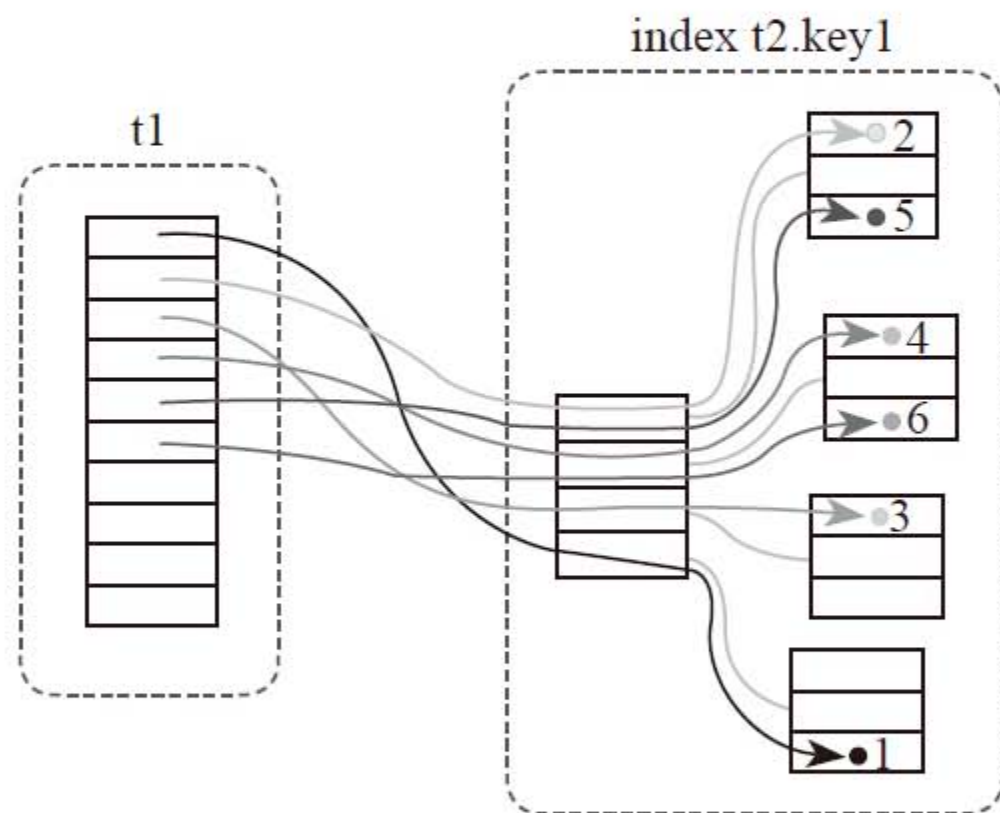


## 2.7.8 支持Batched Key Access ( BKA ) 索引优化

只在MySQL 5.6/5.7、Percona 5.6/5.7和MariaDB 5.3/5.5/10.0/10.1版本里支持Batched Key Access ( BKA ) 索引优化这项功能。

基于Join表连接的算法有Block Nested Loop ( BNL ) 和Batched Key Access ( BKA ) 。

BNL算法的工作原理如图2-133所示。



Regular Nested Loops Join will hit the index at random

图2-133 BNL算法的工作原理

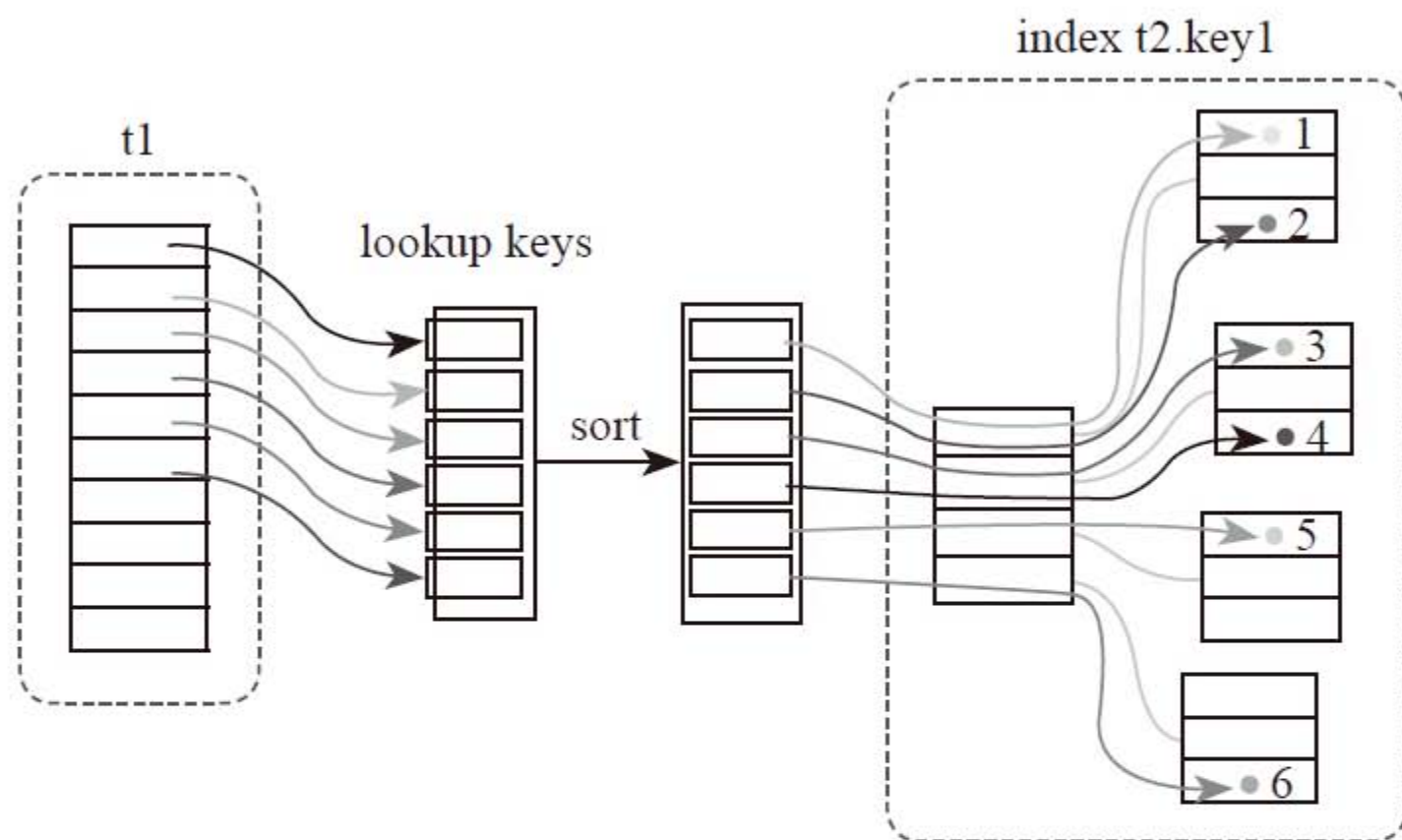
其中，t1表的辅助索引uid关联t2表的辅助索引uid时，BNL将t1.uid记录缓存在join\_buffer\_size里，t2.uid每一行都要跟t1表join\_buffer\_size缓存依次进行关联匹配，最终返回客户端查询结果。

BKA算法的工作原理如图2-134所示。



类似于BNL，只是该算法将t1.uid记录缓存在join\_buffer\_size里时，是使用MRR接口按照t2表uid索引对应的主键顺序进行sort排序的，进而转换为顺序I/O。t2.uid每一行都要跟t1表排序完之后的join\_buffer\_size缓存依次进行关联匹配，最终返回客户端查询结果。

Batched Key Access索引优化在MySQL 5.7.10版本中默认为关闭，可以通过语句show variables like 'optimizer\_switch'\G;来查看：



With *Botched Nested Loops Join and Key-Ordered retrieval*, index lookups are done in one “sweep”

图2-134 BKA算法的工作原理

```
mysql> show variables like 'optimizer_switch'\G;
```

```
***** 1. row *****
```

```
Variable_name: optimizer_switch
```

```
Value:
```

```
index_merge=on, index_merge_union=on, index_merge_sort_union=on, index_merge_intersection=on,  
engine_condition_pushdown=on, index_condition_pushdown=on, mrr=on, mrr_cost_based=on, block_nested_loop=on,  
batched_key_access=off, materialization=on, semijoin=on, loosescan=on, firstmatch=on, duplicateweedout=on,  
subquery_materialization_cost_based=on, use_index_extensions=on, condition_fanout_filter=on, derived_merge=on
```

1 row in set (0.00 sec)

动态开启BKA的命令如下：

```
set global optimizer_switch='batched_key_access=on';
```

Batched Key Access索引优化在MariaDB 10.1版本中默认为关闭，可通过如下命令动态开启：

```
set global join_cache_level=6;
```

join\_cache\_level参数的默认值是2（BNL）算法、4（BNLH）算法、6（BKA）算法和8（BKAH）算法。

通过下面的例子来看BKA算法，如图2-135所示。

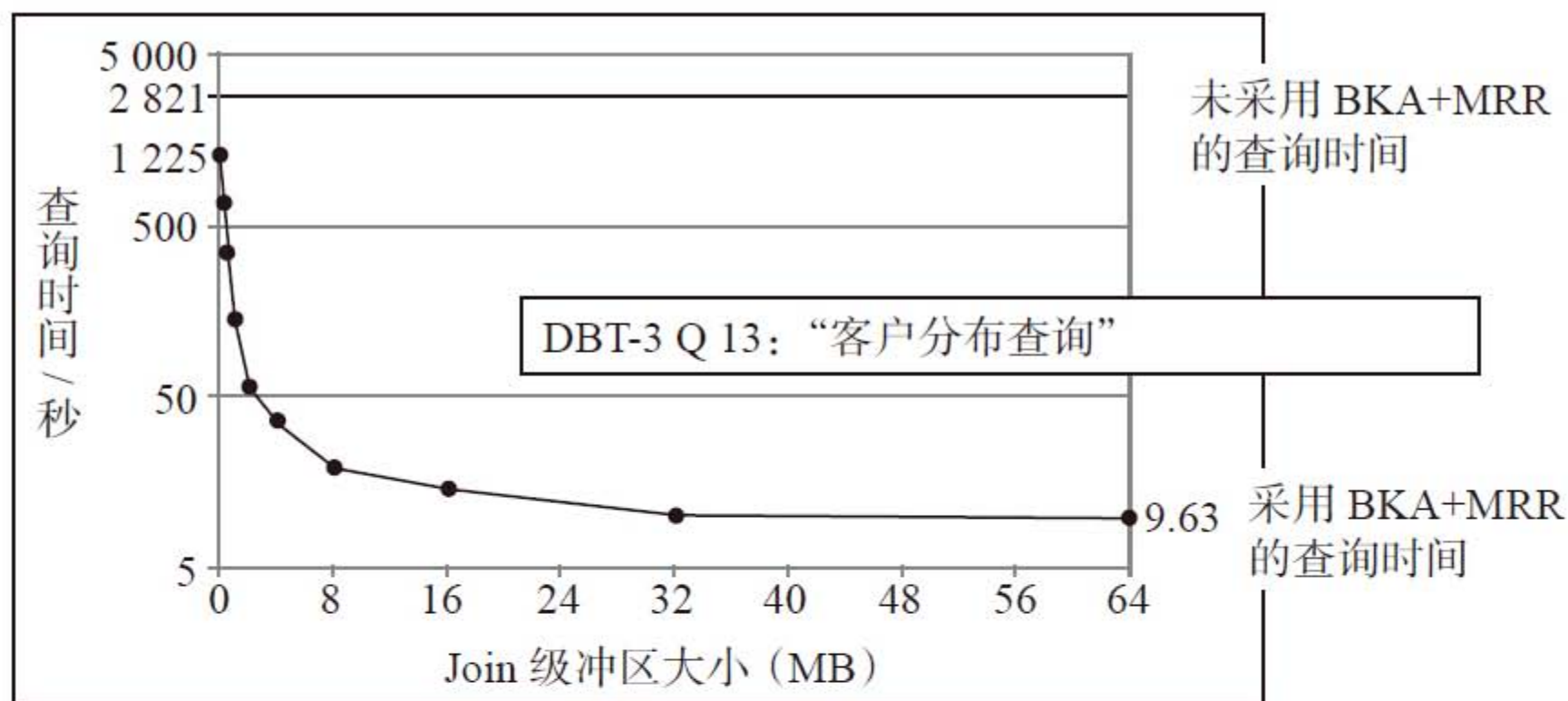
MariaDB [ ]> explain select u.phone from pl join u on pl.user_id = u.id;										
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	pl	index	idx_userid	idx_userid	8	NULL	291677	Using index	
1	SIMPLE	u	eq_ref	PRIMARY	PRIMARY	8	jiea.pl.user_id	1	Using join buffer (flat, BKA join); Key-ordered scan	
2 rows in set (0.00 sec)										
MariaDB [ ]> select version();										
version()										
10.0.21-MariaDB-enterprise-log										
1 row in set (0.00 sec)										
MariaDB [ ]> select @@join_cache_level;										
@@join_cache_level										
6										
1 row in set (0.00 sec)										

图2-135 BKA的执行计划



注意 如果select查询字段是索引，那么根本不需要访问表，直接访问索引树就行了（索引覆盖），因此就不会使用BKA算法了。

甲骨文公司的官方性能测试如图2-136所示。



在 DBT-3 Query 13 和其他针对磁盘的查询基准测试中，BKA 和 MRR 的查询执行速度最高可提升 280 倍。

图2-136 BKA算法性能对比



## 2.7.9 支持Hash Join索引优化

Mariadb 10.0/10.1版本支持块嵌套循环哈希（Block Nested Loop Hash）和块索引哈希批量主键访问（Block Index Hash Join Batch Key Access Hash）两种Hash Join优化方法。

·块嵌套循环哈希连接（Block Nested Loop Hash）首先会选择记录较小的表作为驱动表，读取符合数据计算连接列的哈希值创建一个哈希表；然后根据哈希值去检索被驱动表。

·块索引哈希批量主键访问（Block Index Hash Join Batch Key Access Hash）又叫批量主键哈希连接，它通过多范围读（MRR）的方式访问驱动表主键生成哈希表；然后根据哈希表访问被驱动表并返回结果。

Hash Join算法的工作原理如图2-137所示。

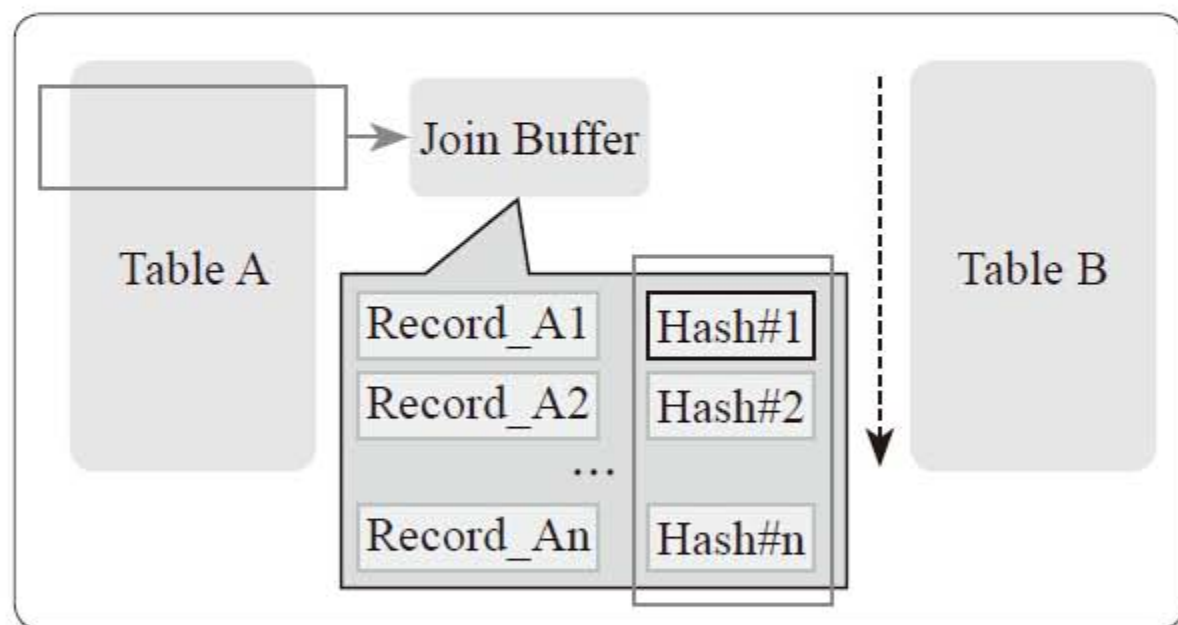


图2-137 Hash Join算法的工作原理

Hash Join索引优化在MariaDB 10.1版本中默认为关闭。

为了支持两种Hash Join，所以要设置为4或8，同时还要打开mrr=on、mrr\_sort\_keys=on这两个特性。

通过下面的例子来看看Hash Join算法，分别如图2-138和图2-139所示。

```
(mariadb10.1.10)root@localhost [join_test]> set join_cache_level = 4;
Query OK, 0 rows affected (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> desc SELECT count(s1.c) FROM sbtest1 s1 JOIN t1 ON s1.k = t1.k WHERE s1.k = 501885;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | idx_k | idx_k | 4 | const | 1 | Using where; Using index |
| 1 | SIMPLE | s1 | hash_range | k_1 | #hash#k_1:k_1 | 4:4 | const | 3000 | Using index condition; Using where; Using join buffer (flat, BNL join) |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.15 sec)

(mariadb10.1.10)root@localhost [join_test]> set optimizer_switch='mrr=on,mrr_cost_based=off,mrr_sort_keys=on';
Query OK, 0 rows affected (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> desc SELECT count(s1.c) FROM sbtest1 s1 JOIN t1 ON s1.k = t1.k WHERE s1.k = 501885;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | idx_k | idx_k | 4 | const | 1 | Using where; Using index |
| 1 | SIMPLE | s1 | hash_range | k_1 | #hash#k_1:k_1 | 4:4 | const | 3000 | Using index condition; Using where; Rowid-ordered scan; Using join buffer (flat, BNL join) |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]>
```

图2-138 块嵌套循环哈希连接的执行计划

```
(mariadb10.1.10)root@localhost [join_test]> set join_cache_level = 8
-> ;
Query OK, 0 rows affected (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> desc SELECT count(s1.c) FROM sbtest1 s1 JOIN t1 ON s1.k = t1.k WHERE s1.k = 501885;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | idx_k | idx_k | 4 | const | 1 | Using index |
| 1 | SIMPLE | s1 | ref | k_1 | k_1 | 4 | const | 3900 | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> set optimizer_switch='mrr=on,mrr_cost_based=off,mrr_sort_keys=on';
Query OK, 0 rows affected (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> desc SELECT count(s1.c) FROM sbtest1 s1 JOIN t1 ON s1.k = t1.k WHERE s1.k = 501885;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | idx_k | idx_k | 4 | const | 1 | Using index |
| 1 | SIMPLE | s1 | ref | k_1 | k_1 | 4 | const | 3900 | Using join buffer (flat, BKAH join), Key-ordered Rowid-ordered scan |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

(mariadb10.1.10)root@localhost [join_test]> █
```

图2-139 块索引哈希的执行计划



注意 Hash Join索引优化这节内容由沃趣科技的邱文辉提供，在此表示感谢！

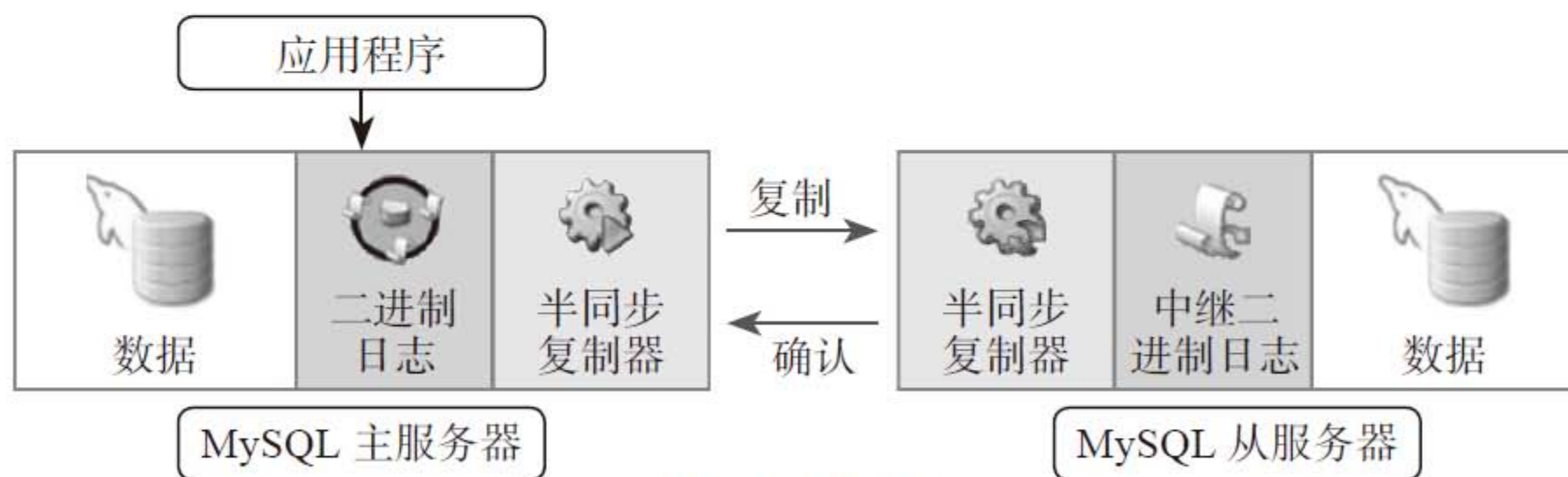


## 2.8 半同步复制改进

### 2.8.1 半同步复制简介

默认情况下，MySQL 5.5/5.6/5.7和MariaDB 10.0/10.1的复制功能是异步的，异步复制可以提供最佳的性能，主库把Binlog日志发送给从库，这一动作就结束了，并不会验证从库是否接收完毕，但这同时也带来了很高的风险，这就意味着当主服务器或从服务器发生故障时，有可能从机没有接收到主机发送过来的Binlog日志，会造成主服务器/从服务器的数据不一致，甚至在恢复时会造成数据丢失。

为了解决上述问题，MySQL 5.5引入了一种半同步复制（Semi Replication）模式，该模式可以确保从服务器接收完主服务器发送的Binlog日志文件并写入自己的中继日志（Relay Log）里，然后会给主服务器一个反馈，告诉对方已经接收完毕，这时主库线程才返回给当前session告知操作完成，如图2-140所示。当出现超时情况时，源主服务器会暂时切换到异步复制模式，直到至少有一台设置为半同步复制模式的从服务器及时收到信息为止（见图2-140）。



半同步复制模式

图2-140 半同步复制模式

简言之，半同步复制在一定程度上可保证提交的事务已经传给了至少一个备库，因此，半同步复制与异步复制相比，进一步提高了数据的完整性。



注意 半同步复制模式必须在主服务器和从服务器同时启用，否则主服务器默认使用异步复制模式。

## 2.8.2 半同步复制的安装配置

这里采用MySQL二进制版本（官方编译好的版本，无需configure、make、make install等步骤），半同步复制插件在目录/usr/local/mysql/lib/plugin下，只需按照下面步骤，即可简单安装完毕：

```
[root@vm01 plugin]# pwd
/usr/local/mysql/lib/plugin
[root@vm01 plugin]#
[root@vm01 plugin]# ll -h semisync_*
-rwxr-xr-x 1 mysql mysql 147K 11-23 23:38 semisync_master.so
-rwxr-xr-x 1 mysql mysql 80K 11-23 23:38 semisync_slave.so
```

在master和slave库首次启动时，安装插件，并开启半同步复制：

```
mysql> install plugin rpl_semi_sync_master soname 'semisync_master.so';
mysql> SET GLOBAL rpl_semi_sync_master_enabled=ON;
mysql> install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
mysql> SET GLOBAL rpl_semi_sync_slave_enabled =ON;
```

在初次加载插件后，MySQL会将该插件记录到系统表mysql.plugin中，下次启动时系统则会自动加载该插件，无需再次执行上面的命令。

另外，在my.cnf配置文件里加入以下命令：

```
rpl_semi_sync_master_enabled = 1
rpl_semi_sync_slave_enabled = 1
```

这样，以后启动MySQL时就会自动开启半同步复制功能。





### 2.8.3 参数说明

半同步复制的配置参数较少，其中，在master主库上有4个相关参数，说明如下。

- `rpl_semi_sync_master_enabled=ON`：表示在master上已经开启半同步复制模式。
- `rpl_semi_sync_master_timeout=10000`：该参数默认为10000毫秒，即10秒，不过，这个参数是动态可调的，它用来表示如果主库在某次事务中的等待时间超过10秒，则降级为异步复制模式，不再等待slave从库。如果主库再次探测到slave从库恢复，则会自动再次回到半同步复制模式。
- `rpl_semi_sync_master_wait_no_slave`：表示是否允许master每个事务提交后都要等待slave的接收确认信号。默认为on，即每一个事务都会等待。如果为off，则slave追赶上后，也不会开启半同步复制模式，需要手工开启。
- `rpl_semi_sync_master_trace_level=32`：表示用于开启半同步复制模式时的调试级别，默认是32。

在slave从库上共有2个配置参数，如下所示。

- `rpl_semi_sync_slave_enabled=ON`：表示在slave上已经开启半同步复制模式。
- `rpl_semi_sync_slave_trace_level=32`：表示用于开启半同步复制模式时的调试级别，默认是32。

## 2.8.4 功能测试

### 1. 如何验证半同步复制是否正常工作

安装完毕后，就要完成主从同步配置了，在MySQL 5.5版中该配置方法与MySQL 5.6/5.7和MariaDB 10.0/10.1版本的一样。

首先，在master库上导出一份全量数据：

```
# /usr/local/mysql/bin/mysqldump -uroot -p123456 -q --single-transaction --master-data=2 -A >alldata.sql
```

然后，把主库上导出的全量数据远程拷贝到从库上，在slave库上导入全量数据完毕后：

```
mysql> CHANGE MASTER TO
MASTER_HOST='192.168.8.22', MASTER_USER='repl', MASTER_PASSWORD='repl', MASTER_PORT=3306,
MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=107;
mysql> start slave;
```

建立主从复制，并开启同步复制。

如何验证半同步复制模式是否已经开始工作了呢？可以在master主库上查看状态，命令如下：

```
mysql> show status like '%semi%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Rpl_semi_sync_master_clients | 1 |
| Rpl_semi_sync_master_net_avg_wait_time | 0 |
| Rpl_semi_sync_master_net_wait_time | 0 |
| Rpl_semi_sync_master_net_waits | 0 |
| Rpl_semi_sync_master_no_times | 0 |
| Rpl_semi_sync_master_no_tx | 0 |
| Rpl_semi_sync_master_status | ON |
| Rpl_semi_sync_master_timefunc_failures | 0 |
| Rpl_semi_sync_master_tx_avg_wait_time | 0 |
| Rpl_semi_sync_master_tx_wait_time | 0 |
```



```

/ Rpl_semi_sync_master_tx_waits / 0 /
/ Rpl_semi_sync_master_wait_pos_backtraverse / 0 /
/ Rpl_semi_sync_master_wait_sessions / 0 /
/ Rpl_semi_sync_master_yes_tx / 1 /
+-----+-----+

```

14 rows in set (0.00 sec)

在上面的状态信息中，有以下参数值得关注。

- Rpl\_semi\_sync\_master\_status：用于指示主服务器是使用异步复制模式，还是半同步复制模式。
- Rpl\_semi\_sync\_master\_clients：用于显示有多少个从服务器配置成了半同步复制模式。
- Rpl\_semi\_sync\_master\_yes\_tx：用于显示从服务器确认的成功提交数量。
- Rpl\_semi\_sync\_master\_no\_tx：用于显示从服务器确认的不成功提交数量。

在slave从库上查看状态，命令如下：

```
mysql> show status like '%semi%';
```

```

+-----+-----+
/ Variable_name / Value /
+-----+-----+
/ Rpl_semi_sync_slave_status / ON /
+-----+-----+

```

1 rows in set (0.00 sec)

在上面的状态信息中，有以下参数值得关注：

Rpl\_semi\_sync\_slave\_status：用于指示从服务器是否启动半同步复制模式，如果状态值为ON，则表示半同步复制模式已经开始工作。

是否开启半同步复制模式，在日志里也有显示，如下所示：

```

120518 11:39:09 [Note] Slave I/O thread: Start semi-sync replication to master 'repl@192.168.8.22:3306' in log 'mysql-bin.000002'
at position 107

```

## 2.半同步复制与异步复制的切换

它的工作原理就是当slave从库的io\_thread线程将binlog日志接收完毕时，要给master主库一个确认信号，如果

rpl\_semi\_sync\_master\_timeout=10000（10秒）超过10秒未收到slave从库的接收确认信号，那么就会自动转换为传统的异步复制模式。

场景一：stop slave。

在slave从库上，停止I/O接收binlog线程，执行以下操作：



```
mysql> stop slave io_thread;
Query OK, 0 rows affected (0.04 sec)
```

该操作会将io\_thread线程关闭，等待10秒后，如果master主库未收到slave从库的接收确认信号，请看下面的程序：

```
mysql> show status like '%semi%';
+-----+
| Variable_name | Value |
+-----+
| Rpl_semi_sync_slave_status | OFF |
+-----+
1 rows in set (0.00 sec)
```

从以上程序可以发现，此时已经转换为异步复制模式。

再次到slave从库上执行以下操作：

```
mysql> start slave;
Query OK, 0 rows affected (0.01 sec)
```

然后查看以下程序：

```
mysql> show status like '%semi%';
+-----+
| Variable_name | Value |
+-----+
| Rpl_semi_sync_slave_status | ON |
+-----+
1 rows in set (0.00 sec)
```

从以上程序可以看到，此时已经由原先的异步复制模式转换为半同步复制模式。

### 场景二：模拟同步报错。

试着这样的操作：先在slave从机上执行drop database test，删除一个test库；然后在master主机上再次执行drop database test，删除一个test库，这时主从同步复制就会报错，代码如下：

```
mysql> show slave status\G;
***** 1. row *****
Slave_IO_State: Waiting for master to send event
```



Master\_Host: 192.168.8.22  
Master\_User: repl  
Master\_Port: 3306  
Connect\_Retry: 60  
Master\_Log\_File: mysql-bin.000003  
Read\_Master\_Log\_Pos: 378  
Relay\_Log\_File: vm02-relay-bin.000020  
Relay\_Log\_Pos: 445  
Relay\_Master\_Log\_File: mysql-bin.000003  
Slave\_IO\_Running: Yes  
Slave\_SQL\_Running: No  
Replicate\_Do\_DB:  
Replicate\_Ignore\_DB:  
Replicate\_Do\_Table:  
Replicate\_Ignore\_Table:  
Replicate\_Wild\_Do\_Table:  
Replicate\_Wild\_Ignore\_Table:  
Last\_Errno: 1008  
Last\_Error: Error 'Can't drop database 'sdf'; database doesn't exist' on query. Default database: 'sdf'. Query: 'drop  
database sdf'  
Skip\_Counter: 0  
Exec\_Master\_Log\_Pos: 299  
Relay\_Log\_Space: 1016  
Until\_Condition: None  
Until\_Log\_File:  
Until\_Log\_Pos: 0  
Master\_SSL\_Allowed: No  
Master\_SSL\_CA\_File:  
Master\_SSL\_CA\_Path:  
Master\_SSL\_Cert:





```

Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 1008
Last_SQL_Error: Error 'Can't drop database 'sdf'; database doesn't exist' on query. Default database: 'sdf'. Query: 'drop
database sdf'
Replicate_Ignore_Server_Ids:
Master_Server_Id: 22
1 row in set (0.00 sec)

```

查看半同步复制状态如下：

```

mysql> show status like '%semi%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Rpl_semi_sync_slave_status | ON |
+-----+-----+
1 rows in set (0.00 sec)

```

从上面的状态信息可以看出，没有转化为异步复制模式，仍然是半同步复制模式，可见半同步复制跟IO\_THREAD是有直接关系的，但跟SQL\_THREAD没关系。也就是说，slave从库接收完二进制日志后会给master主库一个确认信号，但它并不会管中继日志是否执行完。

场景三：提交等待。

先在slave从机上关闭主从复制，为了方便查看效果，调整rpl\_semi\_sync\_master\_timeout=200000毫秒，约等于3分钟。

rpl\_semi\_sync\_master\_wait\_no\_slave=on表示允许master每个事务提交后等待slave的接收确认信号。

下面是操作步骤。

1) 在slave从库，停止同步复制，执行命令为stop slave。

2) 在master主库上执行下面的操作：

```
mysql> use test;
```



Database changed

mysql>begin;

Query OK, 0 rows affected (0.00 sec)

mysql>update t set name='ccd' where id=3;

Query OK, 2 rows affected (0.07 sec)

Rows matched: 2 Changed: 2 Warnings: 0

mysql>select \* from t;

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

| id | name |

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

| 1 | a |

| 1 | a |

| 2 | b |

| 2 | b |

| 3 | ccd |

| 3 | ccd |

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

6 rows in set (0.01 sec)

mysql>commit;

Query OK, 0 rows affected (3 min 20.00 sec)

从以上代码可以看到，一个简单的update需要耗时3分20秒，原因就是半同步复制需要等待slave从库的接收确认，而rpl\_semi\_sync\_master\_timeout设置的是200000毫秒，约等于3分钟，所以提交的速度就变慢了。

## 2.8.5 性能测试

### 1. 半同步复制和异步复制的性能对比

性能测试的目的是对比半同步复制和异步复制在性能上哪个较好，在生产环境中是否可以应用半同步复制。先来看一下测试环境，如下：

Sysbench 192.168.110.19（虚拟机，内存为1GB）

MySQL master 192.168.110.140（虚拟机，内存为1GB）

MySQL slave 192.168.110.141（虚拟机，内存为1GB）

这里采用的是sysbench压力测试，并发100个连接，1万个请求，表2-3是其测试结果。

```
/usr/local/bin/sysbench --test=oltp
--mysql-table-engine=innodb
--oltp-table-size=9000000
--max-requests=10000
--num-threads=100
--mysql-host=192.168.110.140
--mysql-port=3306
--mysql-user=admin
--mysql-password=123456
--mysql-db=test
--mysql-socket=/tmp/mysql.sock run
```

表2-3 半同步复制与异步复制的性能对比

半同步复制	异步复制
参数： rpl_semi_sync_master_timeout = 1000 rpl_semi_sync_master_wait_no_slave = ON	参数： rpl_semi_sync_master_enabled = OFF rpl_semi_sync_master_wait_no_slave = OFF rpl_semi_sync_slave_enabled = OFF
OLTP test statistics: queries performed: read:      140000 write:     50000 other:     20000 total:     210000 transactions:  10000 (50.38 per sec.) deadlocks:     0    (0.00 per sec.) read/write requests: 190000 (957.16 per sec.) other operations:  20000 (100.75 per sec.)	OLTP test statistics: queries performed: read:      140000 write:     50000 other:     20000 total:     210000 transactions:  10000 (65.22 per sec.) deadlocks:     0    (0.00 per sec.) read/write requests: 190000 (1239.26 per sec.) other operations:  20000 (130.45 per sec.)

由于上面使用的是虚拟机，压力测试参数没有调得太高，这种情况下，对两种复制模式进行压力测试，从结果表现上来看，异步复制的性能（吞吐率）要稍好于半同步复制。

在半同步复制中，使用命令show processlist可以看到如图2-141所示的情况。



```
| 100 | admin | 192.168.110.19:44345 | test | Execute | 1 | Waiting for semi-sync ACK from slave |
COMMIT
|
| 101 | admin | 192.168.110.19:44346 | test | Execute | 1 | Waiting for semi-sync ACK from slave |
COMMIT
|
| 102 | admin | 192.168.110.19:44347 | test | Execute | 1 | Waiting for semi-sync ACK from slave |
COMMIT
|
| 103 | admin | 192.168.110.19:44348 | test | Execute | 1 | updating
| DELETE from sbtest where id=4497173
|
| 104 | admin | 192.168.110.19:44349 | test | Execute | 1 | Waiting for semi-sync ACK from slave |
COMMIT
|
| 105 | admin | 192.168.110.19:44350 | test | Execute | 1 | Waiting for semi-sync ACK from slave |
COMMIT
|
| 106 | admin | 192.168.110.19:44351 | test | Execute | 0 | statistics
| SELECT c from sbtest where id=?
+-----+
+-----+
103 rows in set (0.05 sec)
```

图2-141 slave机器在等待接收binlog日志

## 2.半同步复制中数据丢失的风险降低

如果你的生产线开启了半同步复制，那么会对数据的一致性要求较高，但在MySQL 5.6和MariaDB 10.0版本里，会存在数据不一致的风险。

MySQL 5.6和MariaDB 10.0版本中半同步复制的工作原理示意图如图2-142所示。

假设有这么一个场景，客户端提交了一个事务，master把binlog发送给slave，在发送的期间，网络出现波动，此时Binlog Dump线程发送就会卡住，要等待slave把binlog写到本地的relay-log里，然后给master一个反馈信号，等待的时间以rpl\_semi\_sync\_master\_timeout参数为准，默认为10秒。在这等待的10秒钟里，在其他会话中，查看刚才的事务是可以看见的，此时一旦master发生宕机，由于binlog没有发送给slave，前端app切换到slave查看，就会发现刚才已提交的事务不见了。这就是该工作原理的一个缺陷，需要引起注意。

在MySQL 5.7或MariaDB 10.1版本中，半同步复制的工作原理示意图如图2-143所示。

主库写入本地binlog文件里，要等待从库一个ACK信号，假如此时出现网络抖动，用户在其他会话里是看不见刚才提交的事务的，因为还没有将事务提交到引擎层。这时，主库宕机，从库提升为新的主库，用户再重新连接新主库时，重新提交就可以了。

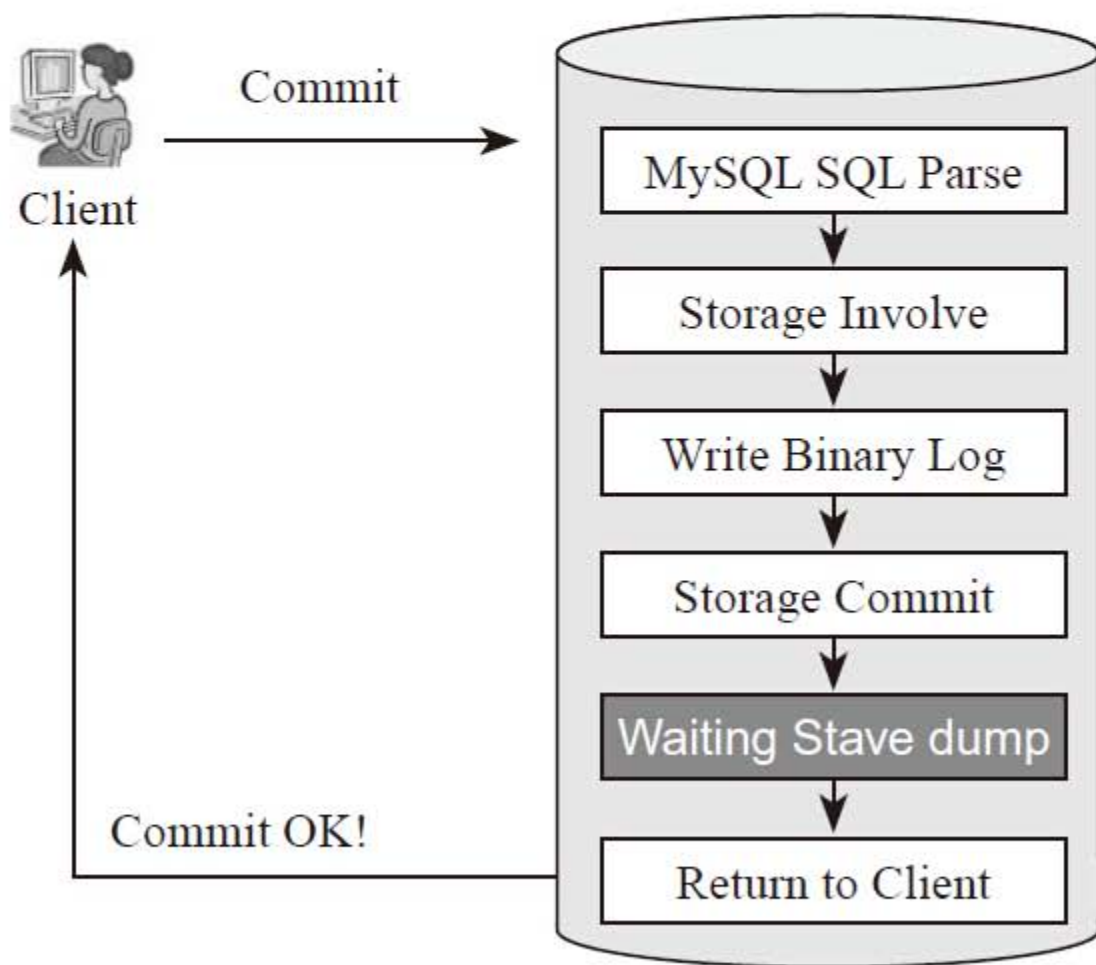


图2-142 优化前半同步复制的工作原理示意图



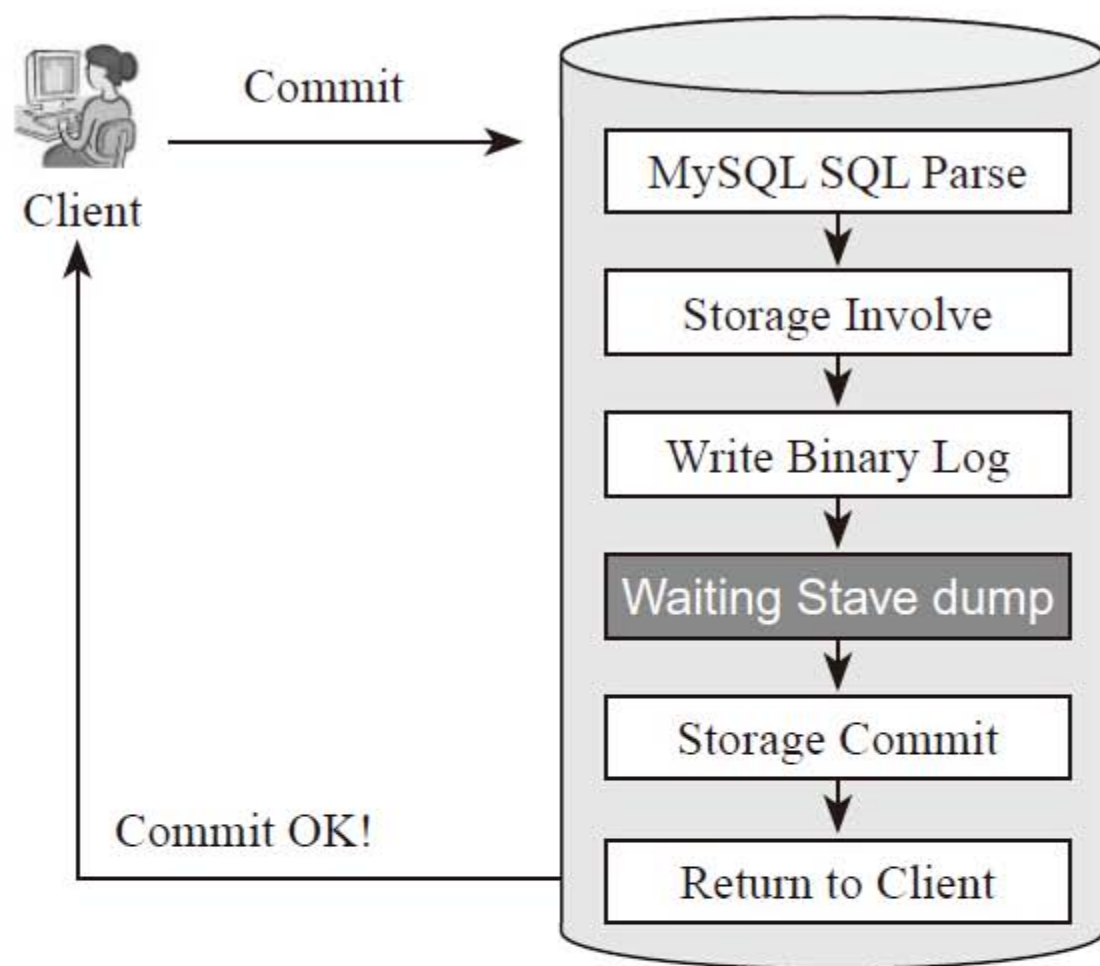


图2-143 优化后半同步复制的工作原理示意图

为了解决这种问题，MySQL 5.7或MariaDB 10.1版本改善了半同步复制这个缺陷。通过`rpl_semi_sync_master_wait_point`这个参数加以控制，MySQL 5.7默认是`AFTER_SYNC`，官方推荐用这个。MariaDB 10.1默认是`AFTER_COMMIT`，这个值是采用老式的MySQL 5.5/5.6或MariaDB 10.0半同步复制工作。

半同步复制的工作原理是：master把binlog发送给slave，只有在slave把binlog写到本地的relay-log里，才提交到存储引擎层，然后把请求返回给客户端，客户端才可以看见刚才提交的事务。如果slave未保存到本地的relay-log里，客户端是看不见刚才的事务的，这样就不会发生上述场景的事情。

半同步复制参数说明如下。

·`rpl_semi_sync_master_wait_point=AFTER_SYNC`：主库把每一个事务写到二进制日志并保存到磁盘上，且发送给从库，主库再等待从库写到自己的relay-log里确认信息。在接到确认信息后，主数据库把事务写到存储引擎里并把相应结果反馈给客户端，客户端将在那时进行处理。

·`rpl_semi_sync_master_wait_point=AFTER_COMMIT`：主库把每一个事务写到二进制日志并保存到磁盘上，且发送给从库，并把事务写到存储引擎里，主



库再等待从库写到自己的relay-log里确认信息。在接到确认信息后，主库把相应结果反馈给客户端，客户端将在那时进行处理。

这两个参数的不同之处在于：当设置为AFTER\_SYNC参数时，所有的客户端可以同时看到提交的数据。在得到从库写到自己的relay-log里的确认信息后，并把事务写到存储引擎里。这样，所有的客户端都可以在主库上看到同样的数据。

主库报错，所有已经写到从库的事务都已经保存到了relay log里。主库的崩溃，HA（High Available，高可用性集群）切换到从库，不会带来任何损失，因为从库的relay-log的数据是最新的。

当设置为AFTER\_COMMIT参数时，发起事务的客户端仅在服务器向存储引擎写入数据并接受从库得到确认信息之后才返回状态。在写入数据后和得到从库确认信息之前，其他的客户端可以看到这一事务。

如果出现了某种错误，比如从库的sql\_thread线程没有执行，那么在主库崩溃和故障转移给从服务器的前提下，有可能这个客户端会丢失那些用户曾经在主库上看到的信息。

此外，MySQL 5.7版本的半同步复制可以通过rpl\_semi\_sync\_master\_wait\_slave\_count参数指定有几台slave接收到了binlog才成功返回客户端请求，默认是一台，但不能指定具体是哪台。



注意 MariaDB 10.1版本没有这个功能。

综上所述，半同步复制是为了保证主从数据的一致性，等待返回的时间长短决定了数据库的更新速度。有利也有弊，利在于数据的一致性；弊在于更新、插入、删除的速度要比传统的异步复制的稍慢一些，因为多了一个回馈确认信息。尤其是在网络受到波动的情况下，这时丢包、ping延时、半同步复制和异步复制就会来回切换，这会致使主库的更新、插入、删除操作受到影响。

## 2.9 GTID复制改进

### 2.9.1 GTID复制概述

主从切换后，在传统的方式里，需要找到binlog和POS点，然后执行命令change master to指向新的主库。对于不是很有经验的运维人员来说，往往会找错，造成主从同步复制报错，在MySQL 5.6版本里，无须再找binlog和POS点，只需要知道master的IP、端口、账号和密码即可，因为同步复制是自动的，MySQL会通过内部机制GTID ( Global Transaction ID ) 自动找点同步。

GTID复制的名词解释如下。

·server\_uuid：服务器身份ID。在第一次启动MySQL时，会自动生成一个server\_uuid并写入数据目录下的auto.cnf文件里，官方不建议修改。并且server\_uuid跟GTID有密切联系。

下面是auto.cnf文件的内容：

```
[root@mysql5_6 data]# pwd
/usr/local/mysql/data
[root@mysql5_6 data]# cat auto.cnf
[auto]
server-uuid=b0869d03-d4a9-11e1-a2ee-000c290a6b8f
```

·GTID：全局事务标识符。使用这个功能时，每次事务提交都会在binlog里生成一个唯一的标识符，它由UUID和事务ID组成。首次提交的事务ID为1，第二次为2，第三次为3，依次类推。

对于事物ID，查看binlog，会看到如图2-144所示的内容。



```
[root@master binlog]# mysqlbinlog -vv mysql-bin.000008 | grep 'GTID_NEXT' | more
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:1' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:2' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:3' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:4' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:5' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:6' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:7' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:8' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:9' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:10' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:11' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:12' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:13' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:14' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:15' /*!*/;
SET @@SESSION.GTID_NEXT= 'd1dd1615-a241-11e5-b087-000c29eddfb6:16' /*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
```

图2-144 GTID事务号

开启GTID时，slave进行同步复制的时候，无须找到binlog日志和POS点，直接change master to master\_auto\_position=1即可，它会自动找点同步。

GTID的工作流程如下。

- 1) 在master上一个事务提交，并写入binlog里。
- 2) binlog日志发送到slave，slave接收完并写入中继日志里，slave读取到这个GTID，并设置gtid\_next的值。例如，  
`SET @@SESSION.GTID_NEXT= 'B0869D03-D4A9-11E1-A2EE-000C290A6B8F:3';`  
然后告诉slave接下来的事务必须使用GTID，并写入它自己的binlog里。
- 3) slave检查并确认这个GTID没有被使用，如果没有被使用，那么开始执行这个事务并写入它自己的binlog里。
- 4) 由于gtid\_next的值不为空，slave不会尝试去生成一个新的gtid，而是通过主从同步来获取GTID。

如何设置MySQL 5.6 GTID方式的主从同步呢？在master和slave上，需要同时在my.cnf文件中加入以下内容：

```
log-bin = mysql-bin
binlog_format = row
log_slave_updates = 1
gtid_mode = ON
enforce_gtid_consistency = ON
```



然后，在master上导出：

```
mysqldump -uroot -p123456 -q --single-transaction -R -E --triggers  
--default-character-set=utf8 --master-data=2 -B yourDB >./yourDB.sql
```

之后进行指向即可，如下所示：

```
mysql> CHANGE MASTER TO  
> MASTER_HOST = master-host ,  
> MASTER_PORT = master-port ,  
> MASTER_USER = repl-user ,  
> MASTER_PASSWORD = repl-password ,  
> MASTER_AUTO_POSITION = 1;
```



注意 如果使用了GTID，就不能再使用传统的binlog和POS方式。

传统的change master to模式如下：

```
CHANGE MASTER TO  
  MASTER_HOST='master2.mycompany.com',  
  MASTER_USER='replication',  
  MASTER_PASSWORD='bigs3cret',  
  MASTER_PORT=3306 ,  
  MASTER_LOG_FILE='master2-bin.001',  
  MASTER_LOG_POS=4 ,  
  MASTER_CONNECT_RETRY=10;
```

否则会报错，如图2-145所示。

GTID的局限性包含以下几方面。

- GTID同步复制是基于事务的，所以MyISAM表不支持，这可能导致多个GTID分配给同一个事务。
- 不支持CREATE TABLE...SELECT语句。因为该语句会被拆分成create table和insert两个事务，并且，如果这两个事务被分配了同一个GTID，则会导致insert被备库忽略掉（见图2-146）。

```
mysql> CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.000008',
-> MASTER_LOG_POS=4;
ERROR 1776 (HY000): Parameters MASTER_LOG_FILE, MASTER_LOG_POS, RELAY_LOG_FILE and RELAY_LOG_POS cannot be set when MASTER_AUTO_POSITION is active.
mysql>
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)
```

图2-145 不支持传统的change master to模式

```
mysql> create table hcy select * from t1;
ERROR 1786 (HY000): Statement violates GTID consistency: CREATE TABLE ... SELECT.
mysql>
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.07 sec)
```

图2-146 不支持CREATE TABLE...SELECT语句

```

Exec_Master_Log_Pos: 309
Relay_Log_Space: 963
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 1032
Last_SQL_Error: Could not execute Delete_rows event on table test.test; Can't find record in 'test', Error_code: 1032;
handler error
HA_ERR_KEY_NOT_FOUND; the event's master log mysql-bin.000001 , end_log_pos 521
Replicate_Ignore_Server_Ids:
Master_Server_Id: 25
Master_UUID: cf716fda-74e2-11e2-b7b7-000c290a6b8f
Master_Info_File: /usr/local/mysql/data/master.info
SQL_Delay: 0
SQL_Remaining_Delay: NULL
Slave_SQL_Running_State:
Master_Retry_Count: 86400
Master_Bind:
Last_IO_Error_Timestamp:
Last_SQL_Error_Timestamp: 130611 23:07:02

```





```

Master_SSL_Crl:
Master_SSL_Crlpath:
Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
Executed_Gtid_Set: 562935a3-74f5-11e2-b830-000c29ba57f2:1-3 ,
cf716fda-74e2-11e2-b7b7-000c290a6b8f:1
Auto_Position: 1

```

1 row in set (0.02 sec)

这里提示删除的主键不存在同步报错，由于是测试机，因此直接跳过：

```
mysql> set global sql_slave_skip_counter=1;
```

*ERROR 1858 (HY000): sql\_slave\_skip\_counter can not be set when the server is running with GTID\_MODE = ON. Instead, for each transaction that you want to skip, generate an empty transaction with the same GTID as the transaction*



注意 由于是运行在GTID模式下，所以不支持sql\_slave\_skip\_counter语法，如果想跳过，就必须把事务ID设置为空值。

```

Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
Executed_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1

```

根据在show slave status\G中获得的信息，观察Retrieved\_Gtid\_Set和Executed\_Gtid\_Set这两行内容，第一行代表接收到的事务，第二行代表已经执行完的事务。也就是说，在执行cf716fda-74e2-11e2-b7b7-000c290a6b8f:2这个事务时报错了，这时，只需跳过这个错误事务就可，如下所示：

```

mysql> stop slave;
Query OK, 0 rows affected (0.07 sec)
mysql> SET GTID_NEXT='cf716fda-74e2-11e2-b7b7-000c290a6b8f:2';
Query OK, 0 rows affected (0.01 sec)
mysql> begin;commit;
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.02 sec)
mysql> SET GTID_NEXT="AUTOMATIC";
Query OK, 0 rows affected (0.02 sec)
mysql> start slave;

```

Query OK, 0 rows affected (0.10 sec)

然后执行show slave status\G;命令确认一下：

mysql>show slave status\G;

\*\*\*\*\* 1. row \*\*\*\*\*

Slave\_IO\_State: Waiting for master to send event

Master\_Host: 192.168.8.25

Master\_User: repl

Master\_Port: 3306

Connect\_Retry: 60

Master\_Log\_File: mysql-bin.000001

Read\_Master\_Log\_Pos: 552

Relay\_Log\_File: M2-relay-bin.000003

Relay\_Log\_Pos: 448

Relay\_Master\_Log\_File: mysql-bin.000001

Slave\_IO\_Running: Yes

Slave\_SQL\_Running: Yes

Replicate\_Do\_DB:

Replicate\_Ignore\_DB: mysql

Replicate\_Do\_Table:

Replicate\_Ignore\_Table:

Replicate\_Wild\_Do\_Table:

Replicate\_Wild\_Ignore\_Table:

Last\_Errno: 0

Last\_Error:

Skip\_Counter: 0

Exec\_Master\_Log\_Pos: 552

Relay\_Log\_Space: 1260

Until\_Condition: None

Until\_Log\_File:

Until\_Log\_Pos: 0



```

Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 0
Last_SQL_Error:
Replicate_Ignore_Server_Ids:
Master_Server_Id: 25
Master_UUID: cf716fda-74e2-11e2-b7b7-000c290a6b8f
Master_Info_File: /usr/local/mysql/data/master.info
SQL_Delay: 0
SQL_Remaining_Delay: NULL
Slave_SQL_Running_State: Slave has read all relay log; waiting for the slave I/O thread to update it
Master_Retry_Count: 86400
Master_Bind:
Last_IO_Error_Timestamp:
Last_SQL_Error_Timestamp:
Master_SSL_Crl:
Master_SSL_Crlpath:
Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
Executed_Gtid_Set: 562935a3-74f5-11e2-b830-000c29ba57f2:1-3 ,
cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
Auto_Position: 1
1 row in set (0.02 sec)

```



OK！已经跳过，同步复制正常。

如果你觉得上述方法有些复杂，那么可以使用pt-slave-restart工具，方法如下：

```
pt-slave-restart -S /tmp/mysql.sock -u root -p 123456 --skip-count 1
```

其中，参数skip-count=1代表跳过一个错误事务。

### 2.9.3 MySQL 5.7中GTID复制的改进

#### 1.支持创建临时表

MySQL 5.7版本可以支持CREATE TEMPORARY TABLE、DROP TEMPORARY TABLE创建临时表，如图2-147所示。

```
mysql> CREATE TEMPORARY TABLE tmp1(t1 int);
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> select @@gtid_mode;
+-----+
| @@gtid_mode |
+-----+
| ON          |
+-----+
1 row in set (0.00 sec)

mysql> SHOW SLAVE HOSTS;
+-----+-----+-----+-----+-----+
| Server_id | Host | Port | Master_id | Slave_UUID |
+-----+-----+-----+-----+-----+
| 17129    |      | 3306 | 128       | 37d202b5-d04b-11e5-9376-000c2966b647 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.00 sec)
```

图2-147 GTID复制模式支持创建临时表

#### 2.开启GTID复制模式，不用开启log\_slave\_updates参数

MySQL 5.6版本的GTID复制模式，必须开启log\_slave\_updates参数，否则启动就报错，因为需要在binlog找到同步复制的信息（UUID:事务号）（见图2-148），如果在密集型写的环境，并且slave从库提供只读服务，这无疑增加了从库不必要的磁盘I/O开销。

```

2014-11-18 16:30:45 5241 [Note] InnoDB: 5.6.20 started; log sequence number 8165949359
2014-11-18 16:30:45 5241 [ERROR] --gtid-mode=ON or UPGRADE_STEP_1 or UPGRADE_STEP_2 requires --log-bin and --log-slave-updates
2014-11-18 16:30:45 5241 [ERROR] Aborting

```

图2-148 未开启log\_slave\_updates参数就报错



注意 开启log\_slave\_updates参数，是把relay-log里的日志内容再记录到slave本地的binlog里。

但在MySQL 5.7版本里，官方进行了调整，用一张gtid\_executed系统表记录同步复制的信息（UUID:事务号），这样就可以不用开启log\_slave\_updates参数，避免两次I/O保存RELAY-LOG和主库的BINLOG，减少了从库的压力（见图2-149）。

3.在线开启/关闭GTID，不用重启mysqld进程（增加了可用性）

MySQL 5.6版本的GTID复制模式，参数gtid\_mode=ON和enforce\_gtid\_consistency=ON不能动态修改，必须写入my.cnf文件里，并且重启主库和从库的mysqld进程才可以生效，在如今的7\*24业务里，是完全无法接受的。所有这些在MySQL 5.7版本里进行了调整，支持在线切换GTID复制模式，大大增强了系统的可用性。

```

mysql> select * from mysql.gtid_executed;
+-----+-----+-----+
| source_uuid | interval_start | interval_end |
+-----+-----+-----+
| d1dd1615-a241-11e5-b087-000c29eddfb6 | 1 | 13 |
| d1dd1615-a241-11e5-b087-000c29eddfb6 | 14 | 14 |
| d1dd1615-a241-11e5-b087-000c29eddfb6 | 15 | 15 |
+-----+-----+-----+
3 rows in set (0.10 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.7.10-log |
+-----+
1 row in set (0.06 sec)

```

图2-149 用gtid\_executed系统表记录同步复制的信息

以下是传统的默认复制切换为GTID复制模式。

开启（主库和从库都按照以下顺序执行）：

```
mysql> set global gtid_mode = 'OFF_PERMISSIVE';
```



```
mysql>set global gtid_mode = 'ON_PERMISSIVE';  
mysql>set global enforce_gtid_consistency = ON;  
mysql>set global gtid_mode = 'ON';
```

以下是GTID复制模式切换为传统的默认复制。

关闭（主库和从库都按照以下顺序执行）：

```
mysql>stop slave;  
mysql>set global gtid_mode = 'ON_PERMISSIVE';  
mysql>set global gtid_mode = 'OFF_PERMISSIVE';  
mysql>CHANGE MASTER TO MASTER_AUTO_POSITION = 0;  
mysql>set global gtid_mode = 'OFF';  
mysql>set global enforce_gtid_consistency = OFF;  
mysql>start slave;
```

## 2.9.4 GTID复制的陷阱

GTID复制是一把双刃剑，虽然很容易搭建主从同步复制，但使用不当，就会掉入“坑”里。

故障重现包含以下几方面。

1) 在slave1上，test库下创建一张hechunyang表。

2) 在slave1上，flush logs刷新binlog，得到切换后的binlog文件名为mysql-bin.000011，然后执行如下命令：

```
PURGE BINARY LOGS TO 'mysql-bin.000011';
```

3) 在slave2上，stop slave停止同步复制，并与slave1建立同步复制，执行如下命令：

```
CHANGE MASTER TO
```

```
MASTER_HOST='slave1', MASTER_USER='repl', MASTER_PASSWORD='repl', MASTER_PORT=3306,  
MASTER_AUTO_POSITION=1, MASTER_CONNECT_RETRY=10;start slave;show slave status\G;
```

就会发现同步复制报错，报错信息如下：

```
Got fatal error 1236 from master when reading data from binary log: 'could not find  
next log; the first event 'mysql-bin.000011' at 857, the last event read from  
'/data/binlog/mysql-bin.000011' at 123, the last byte read from  
'/data/binlog/mysql-bin.000011' at 857.'
```

故障解读包含以下几方面。

slave1上执行了一个事务，同时记录到mysql.gtid\_executed表里，由于在slave上执行了PURGE BINARY LOGS或者超过了参数expire\_logs\_days设置的值，所以binlog日志被删除。当slave2与slave1建立同步复制时，由于找不到mysql.gtid\_executed表里GTID事务对应的binlog文件，所以报上面的1236错误。

在从库执行操作时，一定切记关闭binlog，执行SET SQL\_LOG\_BIN=0，再执行dml/ddl操作；或者为防止被他人误写入，将slave从库设置只读权限read\_only=1（仅对非SUPER权限的用户有效），在MySQL 5.7版本里还提供了super\_read\_only参数，就连root超级用户也无法写入。



注意 MariaDB 10.1版本未提供super\_read\_only参数。





## 2.9.5 MariaDB 10.1中GTID复制的改进

GTID复制出现在MariaDB 10.0版本中，由domain ID-server、ID-sequence number（域ID、server\_id、事务序列号）两部分组成。虽然它与MySQL 5.7的GTID复制原理相同，但实现的方法不同。

如何设置MariaDB 10 GTID方式的主从同步呢？MariaDB 10.0/10.1默认开启GTID复制模式，无需设置任何参数。并不需要像MySQL 5.6版本那样，要在slave从库上设置log\_slave\_updates=1（增加从库的I/O压力），并重启数据库生效那么麻烦。

在master上导出：

```
mysqldump -uroot -p123456 -q --single-transaction -R -E --triggers  
--default-character-set=utf8 --master-data=2 -B yourDB >./yourDB.sql
```

在slave上导入：

```
mysql -uroot -p123456 <./yourDB.sql
```

之后进行指向即可，如下所示：

```
mysql/>CHANGE MASTER TO  
>MASTER_HOST = master-host,  
>MASTER_PORT = master-port,  
>MASTER_USER = repl-user,  
>MASTER_PASSWORD = repl-password,  
>MASTER_USE_GTID = slave_pos;
```

并像MySQL 5.7版本一样，同步复制使用一张mysql.gtid\_slave\_pos系统表记录GTID复制的信息，如图2-150所示。

GTID的局限性包括：GTID同步复制是基于事务的。所以MyISAM表不支持，这可能导致多个GTID分配给同一个事务。

```
MariaDB [(none)]> select * from mysql.gtid_slave_pos;
+-----+-----+-----+-----+
| domain_id | sub_id | server_id | seq_no |
+-----+-----+-----+-----+
|          0 | 132051021 | 17806 | 132050900 |
|          0 | 132051022 | 17806 | 132050901 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

MariaDB [(none)]> select version();
+-----+
| version() |
+-----+
| 10.0.21-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)
```

图2-150 使用gtid\_slave\_pos系统表记录GTID复制的信息



注意 MariaDB 10.0/10.1版本支持CREATE TABLE...SELECT语句。

## 2.9.6 GTID的使用方式不同

### 1. MariaDB支持热切换GTID

无须像MySQL 5.7版本一样要关闭GTID相关参数，MariaDB 10.0/10.1提供了更简单的支持热切换方式。

GTID复制模式切换为传统的复制模式：

```
mysql> CHANGE MASTER TO master_use_gtid = no;
```

传统的复制模式切换为GTID复制模式：

```
mysql> CHANGE MASTER TO master_use_gtid = slave_pos;
```

### 2.可直接跳过同步复制报错

可以像传统的复制模式那样直接跳过报错，如：

```
stop slave;set global sql_slave_skip_counter=1;start slave;
```

但pt-slave-restart工具不支持以下命令：

```
# pt-slave-restart -S /tmp/mariadb.sock --skip-count 1
```

```
DBD::mysql::db selectrow_arrayref failed: Unknown system variable 'gtid_mode' [for Statement "SELECT @@GLOBAL.gtid_mode"]  
at /usr/local/bin/pt-slave-restart line 5008.
```

### 3. MariaDB 10.0/10.1的GTID复制与MySQL 5.6/5.7的不兼容

主库为MySQL 5.7版本，从库为MariaDB 10.1版本，开启GTID复制模式时，同步就会报错，如图2-151所示。



```

Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 1193
Last_IO_Error: The slave I/O thread stops because master does not support MariaDB global transaction id. A fatal error is encountered when it tries to S
ELECT @@GLOBAL.gtid_domain_id. Error: Unknown system variable 'gtid_domain_id'
Last_SQL_Errno: 0
Last_SQL_Error:
Replicate_Ignore_Server_Ids:
Master_Server_Id: 128
Master_SSL_Crl:
Master_SSL_Crlpath:
Using_Gtid: Slave_Pos
Gtid_IO_Pos:
Replicate_Do_Domain_Ids:
Replicate_Ignore_Domain_Ids:
Parallel_Mode: conservative
1 row in set (0.00 sec)

ERROR: No query specified

MariaDB [(none)]> select version();
+-----+
| version() |
+-----+
| 10.1.10-MariaDB-enterprise-log |
+-----+
1 row in set (0.00 sec)

```

图2-151 GTID复制模式不兼容

但使用传统的复制模式可以作为MySQL 5.6/5.7版本的从库。

## 2.10 MySQL 5.6/5.7从库崩溃安全恢复

DBA经常会遇到1032（更新/删除数据找不到）和1062错误（主键冲突），这就是因为从库宕机后，relay-log是以文件形式写盘，没有事务的概念。

我们先看MySQL 5.6的relay-log.info工作原理，如图2-152所示。参数relay\_log\_info\_repository=FILE默认是保存在文件里的。

sql线程每次提交一个事务，就会记录在relay.info文件里。假如在刷盘那一刻宕机，relay-log里没有记录，那么从库重启mysql进程后，就会执行两遍同样的SQL语句，造成同步复制报错。

再来看看把relay-log.info写入表里的情况，命令如下：

```
alter table mysql.slave_relay_log_info engine=innodb;
```

relay\_log\_info\_repository=TABLE的工作原理如图2-153所示。

```
1 | START TRANSACTION;
2 | -- Statement 1
3 | -- ...
4 | -- Statement N
5 | COMMIT;
6 | -- Update replication info files
```

图2-152 中继日志保存在文件里

```
1 | START TRANSACTION;
2 | -- Statement 1
3 | -- ...
4 | -- Statement N
5 | -- Update replication info
6 | COMMIT;
```

图2-153 中继日志保存在表里

这样sql线程执行完事务后，立即会更新slave\_relay\_log\_info表，如果在更新过程中宕机，则事务会回滚，slave\_relay\_log\_info表并不会记录同步的点，下次重新同步复制时，从之前的POS点再次执行。

另外，从MySQL 5.5或MariaDB 10.0版本开始，增加了relay\_log\_recovery参数，这个参数的作用是：当slave从库宕机后，假如relay-log损坏，导致一部分中继日志没有处理，则自动放弃所有未执行的relay-log，并且重新从MASTER上获取日志，这样就保证了relay-log的完整性。默认情况下该功能是关闭的，将relay\_log\_recovery的值设置为1时，可在slave从库上开启该功能，建议开启。

## 2.11 MariaDB 10.0/10.1从库崩溃安全恢复

在MariaDB 10.0.X和10.1.X上不支持`relay_log_info_repository=TABLE`参数，官网建议用GTID复制模式代替传统的复制模式，传统的复制模式是不支持Slave Crash-Safe的，如图2-154所示。

```
[root@master mysql]# scripts/mysql_install_db --defaults-file=/etc/my.cnf --user=mysql
WARNING: The host 'master' could not be looked up with resolveip.
This probably means that your libc libraries are not 100 % compatible
with this binary MariaDB version. The MariaDB daemon, mysqld, should work
normally with the exception that host name resolving will not work.
This means that you should use IP addresses instead of hostnames
when specifying MariaDB privileges !
Installing MariaDB/MySQL system tables in '/data/mysql' ...
2015-08-24 8:14:21 140142845073248 [Warning] 'relay-log-info-repository' is MySQL 5.6 compatible option. Not used or needed in MariaDB.
2015-08-24 8:14:21 140142845073248 [Note] ./bin/mysqld (mysqld 10.1.6-MariaDB-log) starting as process 11396 ...
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: Using mutexes to ref count buffer pool pages
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: The InnoDB memory heap is disabled
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: Memory barrier is not used
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: Compressed tables use zlib 1.2.3
2015-08-24 8:14:21 140142845073248 [Note] InnoDB: Using Linux native AIO
```

图2-154 MariaDB不支持`relay_log_info_repository=TABLE`参数

其工作原理同MySQL 5.6/5.7的`relay_log_info_repository=TABLE`工作原理，这里不再叙述。



## 2.12 slave从库多线程复制

MySQL 5.5版本里是单进程串行复制，通过sql\_thread线程来恢复主库推送过来的binlog，这样会产生一个问题，主库上有大量的写操作，从库就有可能出现延迟。

MySQL 5.6的slave从库多线程复制是基于库（schema）的，设置slave\_parallel\_workers参数，开启基于库的多线程复制。默认是0，不开启，最大并发数为1024个线程。如果用户的MySQL数据库实例中存在多个库，对于slave从库复制的速度可以有比较大的帮助，因为不同的库slave在执行并行复制时，互相没有关联，数据不会不一致。2个库slave就有2个IO/SQL线程，3个库slave就有3个IO/SQL线程，依次类推。

MySQL 5.7的slave从库多线程复制是基于表的，实现的原理基于binlog组提交，简单来说，多个并发提交的事务加入一个队列里，对这个队列里的事务，利用一次I/O合并提交。如果主库上1秒内有10个事务，那么合并一个I/O提交一次，并在binlog里增加一个last\_committed标记（MariaDB的binlog标记是cid），当last\_committed（cid）的值一样时，slave就可以进行并行复制，通过设置多个sql\_thread线程将这10个事务并行恢复，如图2-155所示。

```
[root@dbbak mysql57]# mysqlbinlog -vv mysql-bin.000002 | grep last_committed | more
#160222 14:59:33 server id 17130 end_log_pos 219 CRC32 0x78c11e52 Anonymous_GTID last_committed=0 sequence_number=1
#160222 14:59:33 server id 17130 end_log_pos 1440 CRC32 0x4b092c4b Anonymous_GTID last_committed=0 sequence_number=2
#160222 14:59:33 server id 17130 end_log_pos 2659 CRC32 0x4cc37dba Anonymous_GTID last_committed=0 sequence_number=3
#160222 14:59:33 server id 17130 end_log_pos 3879 CRC32 0xcbafe42a Anonymous_GTID last_committed=0 sequence_number=4
#160222 14:59:33 server id 17130 end_log_pos 5103 CRC32 0x26e754a4 Anonymous_GTID last_committed=0 sequence_number=5
#160222 14:59:33 server id 17130 end_log_pos 6320 CRC32 0x695c3dfc Anonymous_GTID last_committed=0 sequence_number=6
#160222 14:59:33 server id 17130 end_log_pos 7540 CRC32 0xf15bab79 Anonymous_GTID last_committed=0 sequence_number=7
#160222 14:59:33 server id 17130 end_log_pos 8761 CRC32 0xe22b1f7e Anonymous_GTID last_committed=0 sequence_number=8
#160222 14:59:33 server id 17130 end_log_pos 9981 CRC32 0x1355829e Anonymous_GTID last_committed=0 sequence_number=9
#160222 14:59:33 server id 17130 end_log_pos 11201 CRC32 0x01206270 Anonymous_GTID last_committed=0 sequence_number=10
#160222 14:59:33 server id 17130 end_log_pos 12421 CRC32 0x9e23a056 Anonymous_GTID last_committed=1 sequence_number=11
#160222 14:59:33 server id 17130 end_log_pos 13642 CRC32 0xb0676a91 Anonymous_GTID last_committed=1 sequence_number=12
#160222 14:59:33 server id 17130 end_log_pos 14864 CRC32 0x081a70cd Anonymous_GTID last_committed=1 sequence_number=13
#160222 14:59:33 server id 17130 end_log_pos 16085 CRC32 0x86f0068c Anonymous_GTID last_committed=1 sequence_number=14
#160222 14:59:33 server id 17130 end_log_pos 17305 CRC32 0xebdffefb Anonymous_GTID last_committed=1 sequence_number=15
#160222 14:59:33 server id 17130 end_log_pos 18525 CRC32 0x4ba421d2 Anonymous_GTID last_committed=1 sequence_number=16
#160222 14:59:33 server id 17130 end_log_pos 19745 CRC32 0x8a5c92a2 Anonymous_GTID last_committed=1 sequence_number=17
```

图2-155 binlog里增加last\_committed标记

上述last\_committed为0的事务有10个，表示组提交时提交了10个事务，假如设置slave\_parallel\_workers=12（并行复制线程数，根据CPU核数设置），那么这10个事务在slave从库上通过12个线程进行恢复。

通过设置slave-parallel-type=LOGICAL\_CLOCK（基于表的组提交并行复制），默认是slave-parallel-type=DATABASE（基于库的并行



复制)。

MariaDB 10.0/10.1多线程复制的实现原理同MySQL 5.7的,不同的是在binlog里增加了一个cid标记,如图2-156所示。

```
root@master:/data/logs
-rw-rw----. 1 mysql mysql 336 9月 21 13:38 mysql-bin.index
[root@master logs]#
[root@master logs]# mysqlbinlog -vv mysql-bin.000019 | grep GTID | more
#150904 10:48:51 server id 128 end_log_pos 366 GTID 0-128-126343 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 1086 GTID 0-128-126344 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 1808 GTID 0-128-126345 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 2530 GTID 0-128-126346 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 3253 GTID 0-128-126347 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 3974 GTID 0-128-126348 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 4695 GTID 0-128-126349 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 5414 GTID 0-128-126350 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 6134 GTID 0-128-126351 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 6857 GTID 0-128-126352 cid=486 trans
#150904 10:48:51 server id 128 end_log_pos 7579 GTID 0-128-126353 cid=496 trans
#150904 10:48:51 server id 128 end_log_pos 8300 GTID 0-128-126354 cid=496 trans
#150904 10:48:51 server id 128 end_log_pos 9022 GTID 0-128-126355 cid=496 trans
#150904 10:48:51 server id 128 end_log_pos 9743 GTID 0-128-126356 trans
#150904 10:48:51 server id 128 end_log_pos 10465 GTID 0-128-126357 trans
#150904 10:48:51 server id 128 end_log_pos 11185 GTID 0-128-126358 cid=682 trans
#150904 10:48:51 server id 128 end_log_pos 11908 GTID 0-128-126359 cid=682 trans
#150904 10:48:51 server id 128 end_log_pos 12633 GTID 0-128-126360 cid=712 trans
#150904 10:48:51 server id 128 end_log_pos 13354 GTID 0-128-126361 cid=712 trans
#150904 10:48:51 server id 128 end_log_pos 14077 GTID 0-128-126362 cid=751 trans
#150904 10:48:51 server id 128 end_log_pos 14800 GTID 0-128-126363 cid=751 trans
#150904 10:48:51 server id 128 end_log_pos 15523 GTID 0-128-126364 cid=751 trans
#150904 10:48:51 server id 128 end_log_pos 16246 GTID 0-128-126365 cid=822 trans
#150904 10:48:51 server id 128 end_log_pos 16968 GTID 0-128-126366 cid=822 trans
#150904 10:48:51 server id 128 end_log_pos 17691 GTID 0-128-126367 cid=849 trans
#150904 10:48:51 server id 128 end_log_pos 18413 GTID 0-128-126368 cid=849 trans
#150904 10:48:51 server id 128 end_log_pos 19135 GTID 0-128-126369 cid=849 trans
#150904 10:48:51 server id 128 end_log_pos 19849 GTID 0-128-126370 trans
#150904 10:48:51 server id 128 end_log_pos 20571 GTID 0-128-126371 trans
#150904 10:48:51 server id 128 end_log_pos 21290 GTID 0-128-126372 trans
#150904 10:48:51 server id 128 end_log_pos 22012 GTID 0-128-126373 cid=976 trans
```

图2-156 binlog里增加了cid标记

上述cid为486的事务有10个,表示组提交时提交了10个事务,假如设置slave\_parallel\_threads=12(并行复制线程数,根据CPU核数设置),那么这10个事务在slave从库上通过12个线程进行恢复。

官方的性能测试如图2-157所示。

## Parallel Slave Replication

MariaDB 10 solves a long lasting replication challenge that even exists in MySQL 5.8. Until now, with enough writes (INSERT/UPDATE) happening on a master the slaves would not be able to keep up at the same pace and the would lag behind.

With the parallel slave feature in MariaDB 10 this challenge is now gone. The slaves will adapt to the speed of the master and apply binlog events in parallel. Transactions will be applied in parallel if they were excuted in parallel on the master.

Unlike MySQL 5.6, the transactions can concern the same database or even the same table.

sysbench OLTP single database slave tps relative to master`

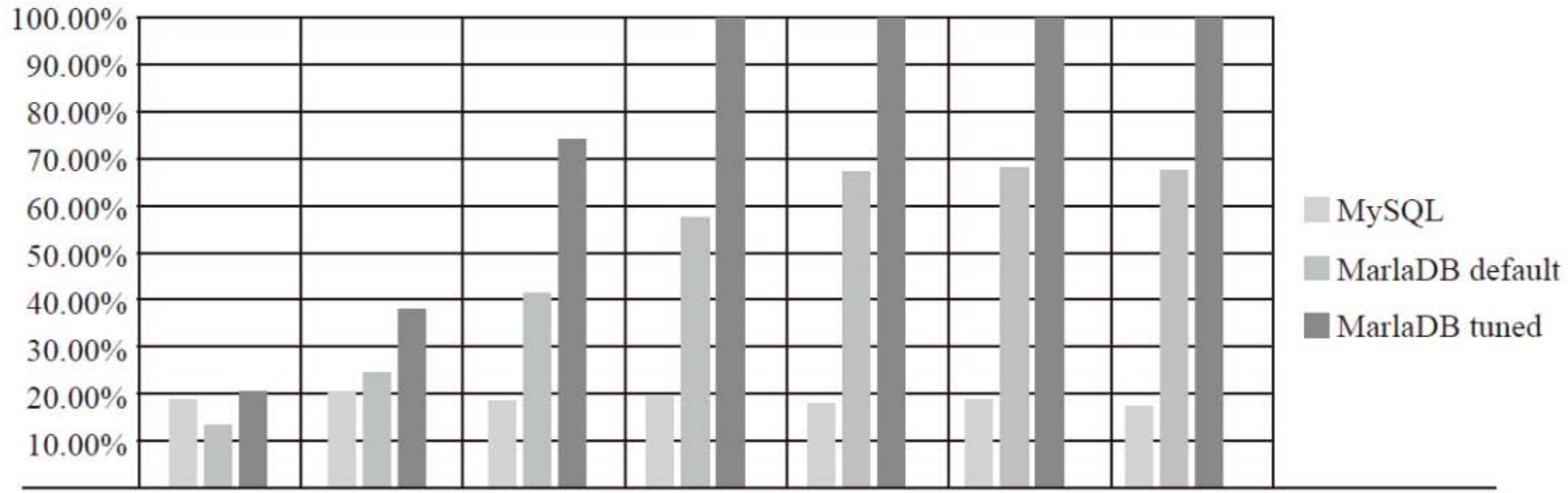


图2-157 单线程和多线程复制的性能对比

从图2-157中可以看出，随着并发线程数的增加，slave从库的恢复速度已经接近master主库的。



## 2.13 slave支持多源复制

在MySQL 5.7或MariaDB 10.0/10.1版本里，一个从库可以支持多个主库，如图2-158所示。

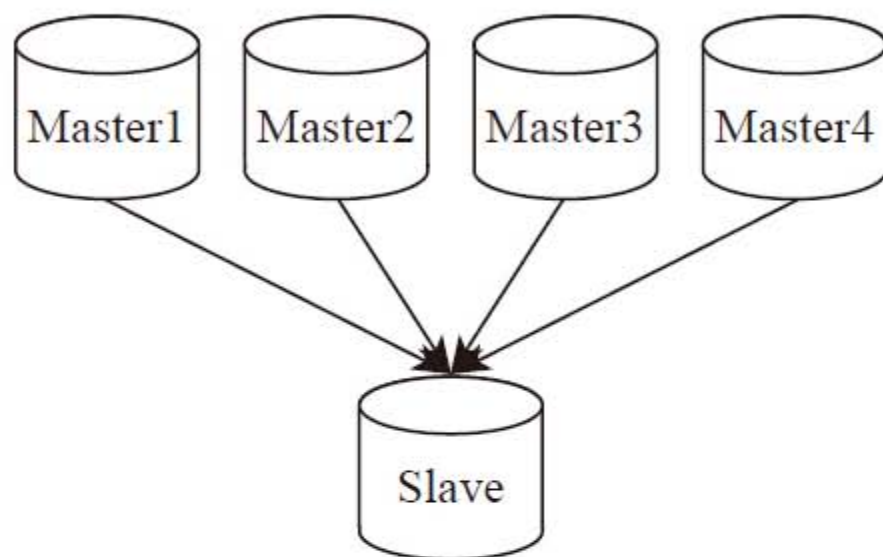


图2-158 多源复制架构

多源复制架构的适用场景：实现数据分析部门的需求，将多个系统的数据汇聚到一台服务器上查询。

### 1. MariaDB 10.0/10.1多源复制搭建

1) 通过如下命令创建通道，命令如下。

```
mysql > SET @@default_master_connection = ${connect_name};
```



注意 `${connect_name}` 为自定义连接名字。

2) 建立同步复制，命令如下：

```
mysql > CHANGE MASTER ${connect_name} TO  
MASTER_HOST='192.168.1.10', MASTER_USER='repl', MASTER_PA  
SSWORD='repl', MASTER_PORT=3306, MASTER_LOG_FILE='mysql-  
bin.000001', MASTER_LOG_POS=4, MASTER_CONNECT_RETRY=1
```

0;

3) 启动多源复制, 命令如下:

```
mysql > START SLAVE ${connect_name};
```

```
mysql > START ALL SLAVES;
```

4) 停止多源复制, 命令如下:

```
mysql > STOP SLAVE ${connect_name};
```

```
mysql > STOP ALL SLAVES;
```

5) 查看状态, 命令如下:

```
mysql > SHOW SLAVE ${connect_name} STATUS;
```

```
mysql > SHOW ALL SLAVES STATUS;
```

6) 清空同步信息和日志, 命令如下:

```
mysql > RESET SLAVE ${connect_name} ALL;
```

7) 刷新Relay logs, 命令如下:

```
mysql > FLUSH RELAY LOGS ${connect_name};
```

## 2. MySQL 5.7多源复制搭建

首先, 从服务器上把master.info和relay.info设置存放在表里, 把如下参数加入my.cnf里:

```
master_info_repository = 'TABLE'
```

```
relay_log_info_repository = 'TABLE'
```

否则在执行CHANGE MASTER TO时会报错:

```
mysql> CHANGE MASTER TO
```

```
MASTER_HOST='192.168.17.128', MASTER_USER='repl', MASTER_PASSWORD='repl', MASTER_PORT=3306,  
MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=4 FOR CHANNEL 'master-1';
```

```
ERROR 3077 (HY000): To have multiple channels, repository cannot be of type  
FILE; Please check the repository configuration and convert them to TABLE.
```

下面开始MySQL 5.7多源复制搭建。

1) 建立同步复制, 命令如下:

```
mysql > CHANGE MASTER TO
```

```
MASTER_HOST='192.168.17.128', MASTER_USER='repl', MASTER_PASSWORD='repl', MASTER_PORT=3306,
```



*MASTER\_LOG\_FILE='mysql-bin.000001', MASTER\_LOG\_POS=4 FOR CHANNEL 'master-1';*

*mysql >CHANGE MASTER TO*

*MASTER\_HOST='192.168.17.129', MASTER\_USER='repl', MASTER\_PASSWORD='repl', MASTER\_PORT=3306 ,  
MASTER\_LOG\_FILE='mysql-bin.000001', MASTER\_LOG\_POS=4 FOR CHANNEL 'master-2';*

2) 启动多源复制, 命令如下:

*mysql >START SLAVE FOR CHANNEL 'master-1';*

*mysql >START SLAVE FOR CHANNEL 'master-2';*

3) 停止多源复制, 命令如下:

*mysql >STOP SLAVE FOR CHANNEL 'master-1';*

*mysql >STOP SLAVE FOR CHANNEL 'master-2';*

4) 查看状态, 命令如下:

*mysql >SHOW SLAVES STATUS FOR CHANNEL 'master-1';*

*mysql >SHOW SLAVES STATUS FOR CHANNEL 'master-2';*

5) 清空同步信息和日志, 命令如下:

*mysql >RESET SLAVE FOR CHANNEL 'master-1';*

*mysql >RESET SLAVE FOR CHANNEL 'master-2';*

6) 刷新Relay logs, 命令如下:

*mysql >FLUSH RELAY LOGS FOR CHANNEL 'master-1';*

*mysql >FLUSH RELAY LOGS FOR CHANNEL 'master-2';*



## 2.14 MySQL 5.7设置同步复制过滤不用重启mysql服务进程

在MySQL 5.5/5.6版本里，设置同步复制过滤，例如，设置忽略掉test库的t2表，你需要在my.cnf配置文件里增加：

*replicate-ignore-table=test.t2*

必须重启mysql服务进程才能生效。

在MySQL 5.7里，通过一个新的命令，可以支持在线动态修改，而无须重启mysql进程就生效。

语法：

```
mysql> CHANGE REPLICATION FILTER REPLICATE_DO_DB=(db1, db2);
```

```
mysql> CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB=(db1, db2);
```

```
mysql> CHANGE REPLICATION FILTER REPLICATE_DO_TABLE=(db1.t1);
```

```
mysql> CHANGE REPLICATION FILTER
```

```
REPLICATE_IGNORE_TABLE=(db2.t2);
```

```
mysql> CHANGE REPLICATION FILTER
```

```
REPLICATE_WILD_DO_TABLE=('db.t%');
```

```
mysql> CHANGE REPLICATION FILTER
```

```
REPLICATE_WILD_IGNORE_TABLE=('db%.a%');
```

```
mysql> CHANGE REPLICATION FILTER
```

```
REPLICATE_REWRITE_DB=((from_db, to_db));
```

通过在线执行CHANGE REPLICATION FILTER REPLICATE\_IGNORE\_TABLE=(test.t2, test.t3);就可以忽略同步复制的表，如图2-159所示。

```
MySQL [test]> CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE=(test.t2, test.t3);
Query OK, 0 rows affected (0.06 sec)

MySQL [test]> start slave;
Query OK, 0 rows affected (0.02 sec)

MySQL [test]> show slave status\G;
***** 1. row *****
      Slave_IO_State: Waiting for master to send event
        Master_Host: 192.168.17.128
        Master_User: repl
        Master_Port: 3316
        Connect_Retry: 60
        Master_Log_File: mysql-bin.000014
    Read_Master_Log_Pos: 2335
        Relay_Log_File: ubuntu2-relay-bin.000014
        Relay_Log_Pos: 464
    Relay_Master_Log_File: mysql-bin.000014
      Slave_IO_Running: Yes
     Slave_SQL_Running: Yes
        Replicate_Do_DB:
    Replicate_Ignore_DB:
        Replicate_Do_Table:
    Replicate_Ignore_Table: test.t3, test.t2
    Replicate_Wild_Do_Table:
    Replicate_Wild_Ignore_Table:
```

图2-159 在线设置不同步的表

增强了易用性，方便了不少。MariaDB10.1并不提供此功能。

## 2.15 小结

MariaDB最大的改善是在server服务层上，InnoDB存储引擎层是集成的Percona的XtraDB，在5.7版本以前，MariaDB在功能上和性能上确实要领先甲骨文MySQL，但随着MySQL 5.7版本的GA出现，这种差距并没有显著的拉大，相反MySQL 5.7在server服务层上的增强，并且在InnoDB引擎层上的改善确实让人眼前一亮，并且截止笔者截稿前，Percona还未发布5.7版本。





## 第3章 故障诊断

在这一章里，将会针对笔者在MySQL服务器管理工作中所遇到的一些常见故障进行汇总，同时还会针对这些故障提供一些分析思路和解决方案，希望能对大家有一定的借鉴作用。作为故障预警，应该尽量做到把问题扼杀在摇篮中，当出现问题时及时处理，不要等到服务器宕机后，再充当“救火队员”，虽然故障解决了，但投诉量也更多了：用户访问不了网站、修复的时间太慢，类似的问题总出现等。所以应该有一套监控流程，当出现报警阈值时，马上去解决问题，把风险降到最低。



## 3.1 影响MySQL性能的因素

影响MySQL InnoDB引擎性能的最主要因素就是磁盘I/O，目前磁盘都是机械方式运作的，主要体现在读写前寻找磁道的过程中。磁盘自带的读写缓存大小，对于磁盘的读写速度至关重要。读写速度快的磁盘，通常都带有较大的读写缓存。磁盘的寻道过程是机械方式，决定了其随机读写速度将明显低于顺序读写。在多进程或多线程并发读取磁盘的情况下，每次执行读写操作，磁道可能存在较大的偏移，磁道寻址时间加大，将会导致磁盘I/O性能急剧下降。

在第1章介绍的新特性来看，几乎都是围绕着如何充分利用内存，如何减少磁盘I/O来展开的，例如，`innodb_io_capacity`参数，可以加大每秒刷新脏页的数量。因此在单块磁盘遇到了I/O瓶颈时，可以把磁盘升级为RAID或SSD固态硬盘来提升性能，SSD固态硬盘的特点是：不用磁头读取数据，寻道时间几乎为0，快速的随机读写，延迟极小，当然价格也很昂贵。目前在生产环境中主要采用RAID10、RAID5，对于数据读写操作频繁的表或数据库，可以适当采用将数据分级存储在SSD固态电子硬盘中的方式，速度会得到较大提升！

影响MySQL InnoDB引擎性能的另一个因素就是内存，InnoDB引擎在设计之初就是考虑用于大型、高负荷、高并发生产环境的，因此内存的大小直接反映了数据库的性能好坏。它在内存中开辟一个Buffer\_Pool缓冲池，然后把数据页和索引页都放在内存缓冲池中读写，因为在内存中的读写要比在磁盘里快得多，InnoDB设计之初就考虑到了这个问题。

在Buffer\_Pool缓冲池中，涉及的参数为`innodb_buffer_pool_size`，它是InnoDB引擎中最重要的参数之一，对InnoDB的性能有决定性的影响。默认的设置只有8 MB，使用默认值时InnoDB的性能很差，远远不能满足生产的需求。在只有InnoDB存储引擎的数据库服务器上面，可以将其设置为60%~80%的内存。如果你有足够的内存，可以将数据量全部放入内存，这时才能达到最佳性能。

上面两个因素是从硬件角度上看的，也就是说因为业务的增长，致使硬件遇到了瓶颈，而另一种常见的情况是，大量的慢SQL是导致性能低下的首要“元凶”，在这种情况下，优化慢SQL是关键，在上线前，应有专门的DBA来审核开发写的SQL语句，通过这样的审核，可避免线上遇到问题。优化一条SQL语句在某种情况下，比增添1条内存管用得多。例如：

```
SELECT * FROM t WHERE id >= '10' and id <= '30';
```

这条语句乍一看没什么问题，可运行后马上就记录在慢日志里了，这是怎么回事？细心的读者可能会发现id是int数值整型，由于加上了引号（"），转化为字符型，于是造成了不能使用索引。可见，审核SQL语句是个很重要的工作。

而如何分辨是硬件性能上遇到了瓶颈，还是SQL自身的问题？这个就要通过日常的监控来确定了，比如，每天早上发一封慢日志邮件来查看SQL的情况，自然就对业务的SQL较为熟悉，再对比最近二到三天内邮件上的慢SQL，这样很容易找出存在的问题。假设某个SQL在昨天慢日志里没有出现，而在今天却出现了，那么尝试着在备机上运行下，如果很快就得到了执行结果，那么就不是SQL的问题，而是业务增长造成的硬件瓶颈。进



一步可以用系统命令来分析，关于分析细节将会在后面章节中具体介绍。



## 3.2 系统性能评估标准

在服务器运维工作中，性能调优是一项富有挑战性的工作，它需要对硬件、操作系统和应用有非常深入的了解。对于MySQL DBA来说，系统性能的实时检测和评估是其需要长期面对的问题，包括上线前各方面的性能测试及上线后整体性能评估，以及随时掌握系统的运行状态是否健康等，对于数据库服务器而言这些工作非常重要。本节将对这方面的知识进行比较全面的介绍。

在运维工作中常用到的性能分析工具包括：vmstat、sar、iostat、netstat、free、ps、top、mpstat以及第三方开发的工具，如dstat、collectl及淘宝的开源监控项目Tsar等。熟悉这些工具能让你清楚当前系统的运行情况，帮助你找出影响系统性能的原因。本书主要针对由第三方开发并开源的工具来进行介绍，书中所有的操作环境都是CenOS(RHEL)平台。



### 3.2.1 影响Linux服务器性能的因素

在操作系统层面影响Linux服务器性能的因素主要就是服务器CPU、内存、磁盘I/O、网络I/O，以及Linux系统本身的内核。

经过几十年的发展，CPU的性能得到了飞速提高，远远超过了计算机系统中的其他组件。虽然说它的性能是整体系统性能中最关键的绝对正确，但是还是有人经常过于高估了它——它也是有负荷的。因此CPU性能的监控与优化也是系统管理员或运维人员必须面对的一个重要问题。不过性能优化核心就是理解系统的运作及系统运行的动态——如果不知道它是如何工作的，就很难对其进行改善。本书不会讲解CPU内部的体系结构、运行机制，因为这超出本书的应用范围和本书的写作意图，但是书中会详细介绍如何实时监控CPU的工作情况及其CPU的性能指标。

人类在理想上都渴望无限大的存储能力，即便是在互联网时代，各类公司还是推举“内存为王”。可以看到，内存对于服务器是多么至关重要。内存是影响MySQL服务器性能好坏的最关键指标，而MySQL的InnoDB引擎中的`innodb_buffer_pool_size`参数可以设置为物理内存的70%~80%。因为把数据放在内存中比存放在磁盘中要快得多，何乐而不为呢？但是话又说回来，也不能随意分配内存，要根据系统的整体情况来做判断。Linux服务器中的内存主要被四类事务所消耗：内核、文件系统高速缓存、应用程序进程、特定预留的共享内存，这点希望读者朋友要有所了解，具体它们之间是怎么使用和调度的请参考《深入理解linux虚拟内存管理》一书，本书将不详细介绍。

磁盘技术发展到今天也有了很大的进步，比如，当今比较火的Fusion-io，它速度已经慢慢接近内存速度，可加速应用级的I/O性能，但是它目前价格相当昂贵，一般的企业无法普遍使用。还有一种叫SSD固态硬盘技术，它是用固态电子存储芯片阵列而制成的硬盘，它功能及使用方法与普通硬盘的完全相同，但是它的成本也比较高。市面上比较常见的固态硬盘有LSISandForce、Indilinx、JMicron、Marvell、Samsung及Intel等多种主控芯片。它的读写速度快，固态硬盘厂商大多会宣称自家的固态硬盘持续读写速度超过了500 MB/s。但目前来说由于技术限制，其容量要比普通硬盘小得多，同时还会有使用寿命限制。

在磁盘技术中应用得最多的还是RAID技术，叫廉价磁盘冗余阵列技术。简单地说，RAID是一种把多块独立的硬盘（物理硬盘）按不同的方式组合起来形成一个硬盘组（逻辑硬盘），从而提供比单个硬盘更高的存储性能和数据备份技术的方式。RAID具有数据冗余备份功能，一旦用户数据发生损坏，利用备份信息即可使损坏数据得以恢复，从而保障了用户数据的安全性。

RAID技术有几种不同的等级，它们可分别提供不同的速度、安全性和性价比，常用的有RAID0、RAID1、RAID10、RAID5、RAID50等RAID技术，但是目前在数据库应用中用得最多还是RAID10。由于篇幅限制，各种RAID技术的相关知识这里不再介绍，请读者朋友查阅相关文档资料。



### 3.2.2 系统性能评估指标

在了解Linux操作系统中的各种调优参数和性能度量工具之前，有必要讨论一下关于系统性能的各种可用指标和它们的意义。由于Linux是一个开源的操作系统，因此有很多开源的性能工具可以使用，不过，这些工具的测量指标都是相同的，理解这些指标能让你使用你所碰到的任何工具。本节将会介绍其中最重要的一些指标。

#### 1.CPU性能指标

从整体上来说，CPU性能指标比较多，因为CPU处理的事物也比较多。常见的指标如下：

##### ·CPU使用率

这可能是最直接的指标了，它表示每个处理器的整体使用率。如果在持续一段时间里CPU的使用率大于80%，这就可能表明CPU出现了瓶颈。

##### ·%us：应用程序（用户空间）

表示用户应用进程所花费的CPU百分比，包括Nice时间。如果用户时间值很高，表明系统正在执行实际的工作。

##### ·%sy：系统（内核空间）

表示内核操作所花费的CPU百分比，包括中断。系统时间值持续很高表明网络或驱动器堆栈可能存在瓶颈。通常，系统只会花费很少时间在内核时间上。

##### ·%wa：I/O等待

等待I/O操作所需的CPU时间总和，系统不应该花费过多的时间等待I/O操作，否则你应该检查一下I/O子系统各方面的性能。

##### ·%id：空闲时间

表示CPU空闲的百分比。这个值越大表明系统CPU的负荷越小。

##### ·%ni：Nice时间

表示花费在执行re-nicing（改变进程的执行顺序和优先级）进程的CPU百分比。

除了上面介绍的这些性能指标以外，还有诸如队列中等待执行的进程数、上下文切换、中断等，这里不再详细描述。

#### 2.内存性能指标

影响内存的性能指标无非就内存大小和虚拟空间大小。内存的性能指标相对比较少，也比较容易理解，常见的主要指标包括：

##### ·空闲内存



与其他操作系统相比，在Linux系统中不必过分在意空闲内存值。因为Linux内核会将大量未使用的内存分配给文件系统来缓存数据，初级运维工作人员用free-m命令看到内存使用率时会感到疑惑，就是这个原因。实际空闲内存应为已用内存扣除用于缓冲和缓存的数量而得到。

#### ·交换空间使用

这个值表示已使用的交换空间大小，相当于Windows系统中的虚拟内存。交换空间的使用只能告诉你Linux在管理内存上是多么的有效。要想确定内存是否存在瓶颈，需要用到Swap In/Out。如果Swap In/Out长时间保持在每秒钟超过200~300页，可能表示内存存在瓶颈。

### 3.磁盘性能指标

磁盘性能对数据库来说至关重要，它严重影响着数据库的读写操作。

#### ·磁盘I/O等待

CPU在等待I/O操作时所花费的时间。如果这个值持续很高，很可能表示I/O存在瓶颈。

#### ·队列平均长度

I/O请求的数量。通常硬盘队列值为2~3时最佳；过高可能表示硬盘I/O存在瓶颈。

#### ·平均等待时间

I/O请求服务所花费的平均时间。等待时间包括实际I/O操作的时间和在I/O队列中等待的时间。单位为毫秒（ms）。

#### ·每秒钟传输的数量

表示每秒钟执行了多少次I/O操作（包括读取和写入）。与每秒钟传输字节数结合可以帮助确定系统平均传输值大小。平均传输值通常要与硬盘子系统的条带大小一致。

#### ·每秒钟读写块的数量

这个指标表示每秒钟读写块的数量，在2.6.XX内核中块的大小为1024字节，早期的内核可以有不同的块大小，其范围可从512字节到4KB。

#### ·每秒钟读写字节的数量

表示块设备读写的实际数据的数量，单位为KB。

上面介绍了服务器系统性能评估指标，这只是个大概的轮廓，还有更多的相关指标请读者朋友通过互联网或从其他书籍中去了解。



### 3.2.3 开源监控和评估工具介绍

注

Linux操作系统的开放和灵活性使其拥有大量的性能监控工具。其中有一些是原来UNIX上知名工具的Linux版，还有一些是专门为Linux设计的。大多Linux性能监控工具是基于虚拟proc文件系统的。要准确评估性能，还需要使用一些适当的基准工具。

本节将给大家介绍一个第三方开源工具。Linux服务器性能监控器：**dstat**。

对于大多数从事运维工作的朋友来说，用得比较多的可能还是要属**sysstat**，它提供了**iostat**、**mpstat**、**cifsio**stat、**sar**等工具，而今天要介绍的**dstat**是一个全能系统信息统计工具，可用来替换**vmstat**、**iostat**、**netstat**、**nfsstat**和**ifstat**这些命令。它是由Python编写的，与**sysstat**相比，**dstat**是以一个彩色的界面动态显示的，这样数据会比较显眼，容易观察，一目了然，而且**dstat**支持即时刷新，可以使用相关参数来指定显示哪些内容！

下面开始进入**dstat**的神秘世界。

官方站点网址：<http://dag.wieers.com/home-made/dstat/#download>

#### 1.dstat安装

安装很简单，只需一个rpm命令即可完成。

```
[root@MySQL56-s ~]# wget http://pkgs.repoforge.org/dstat/dstat-0.7.2-1.el5.rfx.noarch.rpm
[root@MySQL56-s ~]# rpm -ivh dstat-0.7.2-1.el5.rfx.noarch.rpm
warning: dstat-0.7.2-1.el5.rfx.noarch.rpm: Header V3 DSA signature: NOKEY, key ID 6b8d79e6
Preparing... ##### [100%]
1:dstat ##### [100%]
```

#### 2.使用说明

在**dstat**中使用的语法如下：

```
dstat [-afv][options..] [delay [count]]
```

简单执行**dstat**命令，将会得到如图3-1所示的内容：



```
[root@twexdb1 ~]# dstat
You did not select any stats, using -cdngy by default.
-----total-cpu-usage----- -dsk/total- -net/total- ---paging-- ---system--
usr  sys  idl  wai  hiq  siq| read  writ| recv  send|  in   out|  int  csw
 4    0  95    0    0    0|    0    0|    0    0|    0    0| 2783 2759
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1491  678
 1    0  99    0    0    0|8192B  512B|    0    0|    0    0| 2111 1630
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1216  369
 1    0  98    1    0    0|    0    0|    0    0|    0    0| 2409 1965
 1    0  98    0    0    0|    0    0|    0    0|    0    0| 3641 3713
 1    0  99    0    0    0|8192B    0|    0    0|    0    0| 1599  955
 1    0  99    1    0    0|    0    0|    0    0|    0    0| 2281 1616
 0    0  99    0    0    0|    0    0|    0    0|    0    0| 1312 1150
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1394  796
 0    0  99    1    0    0|8192B 3044B|    0    0|    0    0| 1111  457
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1208  340
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1310  524
 1    0  99    0    0    0|    0    0|    0    0|    0    0| 2449 1861
 2    0  97    0    0    0|8192B    0|    0    0|    0    0| 4871 4964
 1    0  99    0    0    0|    0    0|    0    0|    0    0| 2156 1610
 0    0 100    0    0    0|    0    0|    0    0|    0    0| 1306  488
 1    0  99    0    0    0|    0    0|    0    0|    0    0| 2463 1841
 0    0  99    0    0    0|8192B  512B|    0    0|    0    0| 1117  666
 0    0 100    0    0    0|    0    0|7363B    0|    0    0| 2101 1340
 2    0  97    1    0    0|    0    0|    0    0|    0    0| 3816 3696
 1    0  99    0    0    0|    0    0|    0    0|    0    0| 2694 2112
 1    0  98    0    0    0|8192B  512B|    0    0|    0    0| 3220 3087
 2    0  98    0    0    0|    0    0|    0    0|    0    0| 3469 3212
```

图3-1 dstat状态信息

不带任何参数时它只会收集CPU、Disk、Network、Paging、System的状态信息，默认是1 s收集一次。输入dstat相当于输入dstat-cdngy1或dstat-a 1。

下面来看看dstat常用使用的参数，如下所示：

- c, -cpu：显示CPU情况。
- C 0, 3, total include cpu0, cpu3 and total：如果有多个CPU的话，可以指定特定CPU或者所有CPU。
- d, -disk：显示磁盘情况。
- D total, hda include hda and total：显示所有磁盘的情况。
- g, -page enable pagestats：显示页的IN/OUT状态信息。
- i, -int enable interrupt stats：启用网卡中断统计。
- l, -load enable loadstats：显示系统1分钟、5分钟、15分钟的平均负载。
- m, -mem：显示内存情况。
- n, -net：显示网络情况。
- N eth1, total：显示指定网络接口。

- p, -proc enable process stats : 启用进程的状态显示。
- s, -swap : 显示swap情况。
- S swap1, total : 可以指定多个swap。
- t, -time enable timecounte : 显示当前系统时间。
- ipc : 报告IPC消息队列和信号量的使用情况。
- lock enable lockstats : 显示系统各种状态锁的信息。
- raw enable raw stats : 显示裸设备的状态信息。
- tcp enable tcp stats : 显示tcp连接的状态信息。
- udp enable udp stats : 显示ucp连接的状态信息。
- M stat1, stat2 enable internal stats and external plugin stats : 显示一些额外的状态信息，如postfix、sendmail、raw等，具体可以通过dstat-M list列出。
- a, -all : 使用缺省的-cdngy。
- f, -full : 使用-C、-D、-I、-N和-S显示。
- v, -vmstat : 使用-pmgdsc-D显示。
- nocolor disable colors(implies-noupdate) : 关闭颜色显示，只显示一种颜色。
- noheaders : 只显示一次表头以后就不显示了，使用重定向写入文件时很有用。
- output file : 把显示结果写入到CVS文件中。

了解了dstat使用的命令和参数，现在来看看如何使用，它可以帮助我们找到系统的瓶颈。

#### 实例1：dstat sda-D35

可使用dstat sda-D35来协助找系统瓶颈，它只显示sda磁盘的信息。如图3-2所示。

这里“35”的意思跟vmstat35一样，即每隔3秒更新一次，总共更新5次。

```
[root@twexdb1 ~]# dstat -D sda 3 5
You did not select any stats, using -cdngy by default.
-----total-cpu-usage----- --dsk/sda-- --net/total- ---paging-- ---system--
usr  sys  idl  wai  hiq  sig| read  writ| recv  send|  in   out | int   csw
 4    0   95    0    0   0|2048B 128B|    0    0 |    0    0 |2783  2759
 1    0   99    0    0   0|2731B 171B| 43k   40k|    0    0 |1753  1074
 0    0   99    0    0   0|2731B 171B| 23k   87k|    0    0 |1813  1194
 1    0   98    0    0   0|    0    0 | 86k  134k|    0    0 |1782  1112
 1    0   99    0    0   0|2731B 171B| 25k   17k|    0    0 |1875  1361
 1    0   99    0    0   0|2731B 171B| 34k  150k|    0    0 |1858  1184
[root@twexdb1 ~]#
```

图3-2 显示磁盘信息

#### 实例2：dstat-cdlmnpysy

可使用dstat-cdlmnpysy，来协助找系统瓶颈，它显示CPU和内存的信息（如图3-3所示）。



[root@twexdb1 ~]# dstat -cdlmnpys																											
-----total-cpu-usage-----										-dsk/total-		---load-avg---			-----memory-usage-----				-net/total-		---procs---			-----swap----		---system---	
usr	sys	idl	wai	hiq	sig	read	writ	1m	5m	15m	used	buff	cach	free	recv	send	run	blk	new	used	free	int	csw				
4	0	95	0	0	0	k	k	1.43		0.46	2122M	356M	5369M	133M	0	0	0	0	5.5	k	8189M	2783	2759				
4	0	95	0	0	0	0	k	1.43		0.46	2122M	356M	5369M	133M	k	k	1.0	0	0	k	8189M	1354	620				
10	0	90	0	0	0	0	k	1.43		0.46	2122M	356M	5369M	133M	k	k	3.0	1.0	0	k	8189M	3283	4281				
2	0	97	0	0	0	0	k	1.32		0.46	2122M	356M	5369M	133M	k	k	0	0	0	k	8189M	4448	4742				
2	0	98	0	0	0	3192B	512B	1.32		0.46	2125M	356M	5369M	131M	k	k	0	0	39	k	8189M	1303	974				
6	0	93	1	0	0	0	k	1.32		0.46	2124M	356M	5369M	131M	k	k	0	0	2.0	k	8189M	4544	5048				
4	1	95	0	0	0	0	k	1.32		0.46	2124M	356M	5369M	132M	k	k	0	0	1.0	k	8189M	5188	6041				
5	0	95	0	0	0	0	k	1.32		0.46	2124M	356M	5369M	132M	k	k	8.0	0	0	k	8189M	4000	3876				
8	0	91	0	0	0	3192B	k	1.21		0.45	2123M	356M	5369M	132M	k	k	0	0	1.0	k	8189M	4637	4969				
4	0	95	0	0	0	0	k	1.21		0.45	2123M	356M	5369M	132M	k	k	0	0	0	k	8189M	4092	3860				
2	0	97	1	0	0	0	k	1.21		0.45	2123M	356M	5369M	132M	k	k	0	0	2.0	k	8189M	2745	2475				
3	0	97	0	0	0	0	k	1.21		0.45	2123M	356M	5369M	132M	k	k	2.0	0	0	k	8189M	3187	2828				
4	0	95	0	0	0	3192B	k	1.21		0.45	2123M	356M	5369M	133M	k	k	0	0	0	k	8189M	2773	2537				
0	0	100	0	0	0	0	0	1.11		0.45	2123M	356M	5369M	133M	k	k	0	0	0	k	8189M	1482	704				
1	0	99	0	0	0	0	k	1.11		0.45	2122M	356M	5369M	133M	k	k	0	0	4.0	k	8189M	2705	2139				
1	0	99	0	0	0	0	k	1.11		0.45	2123M	356M	5369M	133M	k	k	0	0	2.0	k	8189M	1156	434				
3	0	97	0	0	0	3192B	k	1.11		0.45	2122M	356M	5369M	133M	7576B	k	0	0	0	k	8189M	2471	2053				
0	0	100	0	0	0	0	k	1.11		0.45	2122M	356M	5369M	133M	k	k	0	0	0	k	8189M	1098	241				
0	0	100	0	0	0	0	0	1.02		0.45	2122M	356M	5369M	133M	3887B	k	0	0	0	k	8189M	1009	136				
1	0	99	0	0	0	0	k	1.02		0.45	2122M	356M	5369M	133M	520B	1310B	0	0	1.0	k	8189M	1890	1949				
[root@twexdb1 ~]#																											

图3-3 显示CPU、I/O、load、memory、network、process、swap、system

以上各模块显示的内容跟top、vmstat、iostat等这些工具表现的含意完全相同，比如，与CPU相关的usr代表应用空间也就是应用程序所占用的百分比，注意这里也是百分比，sys表示系统内核空间占用的百分比，idl表示CPU空闲情况，wai表示I/O等待数，hiq和sig则显示服务中断有关的信息。

其他就不再一一说明，这工具的应用还算比较简单，显示也很直观。工具的使用需要靠平时多去练习和观察，这样才能熟能生巧！

- 此节由我的朋友邱治军参与编写，在此表示感谢。



## 3.3 故障与处理

### 3.3.1 连接数过多导致程序连接报错的原因

连接数是直接反应数据库性能好坏的关键指标，连接数过多，很可能有多种原因，比如，被一条SQL查询给堵死了，造成了后面的DML操作等待，又比如，增、删、改、查操作很频繁，磁盘I/O遇到了瓶颈，导致无法处理繁忙的请求等。

也许有人会问，出现了too many connections，直接增大max\_connections值就行了吧？其实，这得看情况，如果你没有设置这个参数，那么将其设置为500~1000，在大多数场合就可以了，但如果是想一直增大（比如增大到10000），那么这就是治标不治本了，因为连接数增大，会导致每个连接所占用的内存也增加，这样一来，机器就很容易因内存不足而死机，所以我们得找到引起连接数升高的本质原因。

在正常情况下，MySQL处理完一条请求后，会根据wait\_timeout值来释放连接（参数含义：服务器关闭非交互连接之前等待活动的秒数），一般设置为100秒即可，如果你没有设置采用默认的28800秒，那么客户端在连接到MySQL Server处理完相应的操作后，要等到28800秒以后才会释放内存，如果你的MySQL Server有大量的闲置连接，不仅会白白消耗内存，而且如果连接一直在累加而不断开，最终肯定会达到MySQL Server的连接上限数，这会报'too many connections'的错误。

下面来看一个因重启服务器导致数据库连接数过多的经典案例。之前我们用的是共享表空间，版本是V5.1，由于运行时间很长，产生了很多的碎片，导致ibdata文件变得很大，所以我们就用导出数据和然后再导入，并且重建ibdata的方式来释放碎片。InnoDB分为共享和独立表空间。alter table TableName engine=innodb通过这条命令可以回收表空间整理碎片。但因是共享表空间，不是独立表空间，这样操作可以回收数据空间，而不是磁盘空间。

这里以书柜的例子来解释整理表空间碎片。书柜上的书摆放的很凌乱，此时你把书都整理好了，缩小了它们在书柜里的位置，但书柜还是那个尺寸，不会因为把书弄整齐了，书柜也就缩小了。而独立表空间是：有多少书就放多大尺寸的书柜里。它既可以回收数据空间，也可以回收磁盘空间。

整理完碎片重启服务器后，在早高峰期用户登录时，连接数满了，导致用户登录时一直等待，此时数据库已经处于无响应状态，如图3-4和图3-5所示。



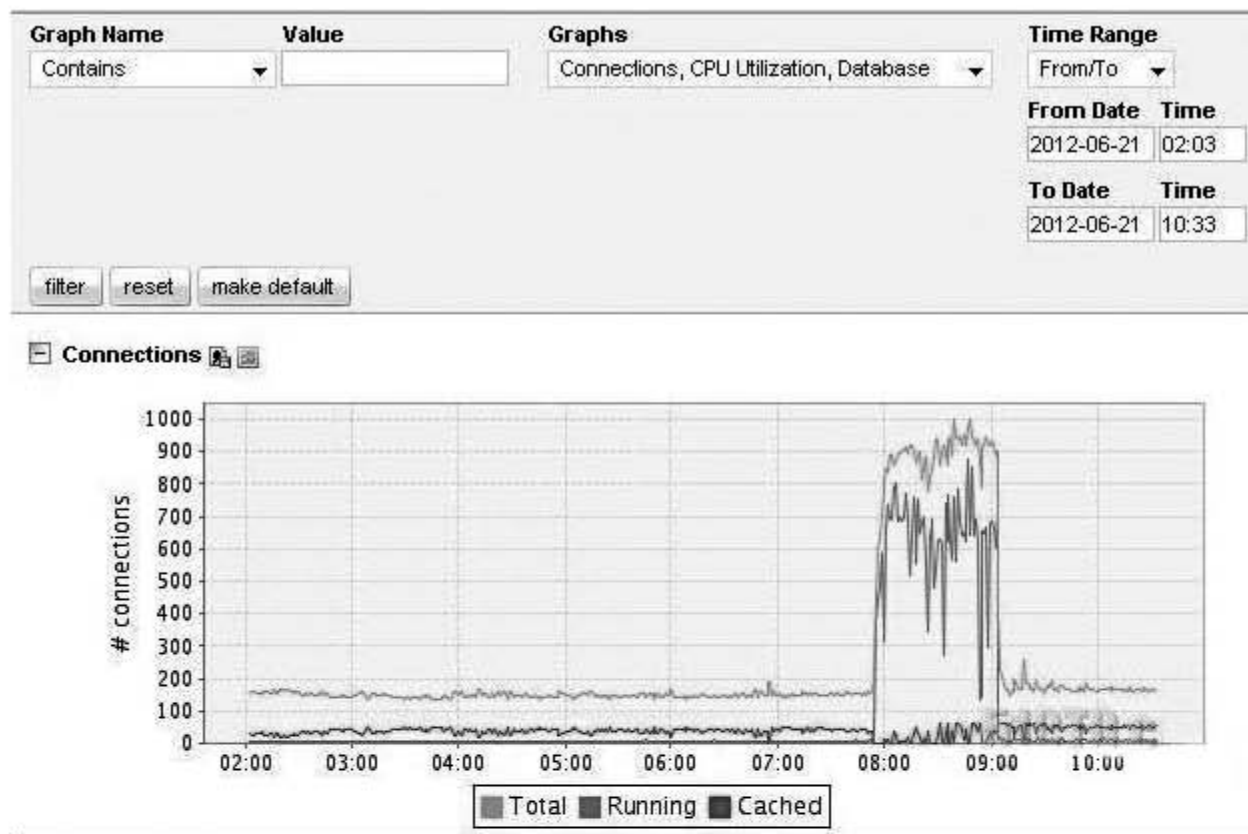


图3-4 连接数

经过分析，发现早高峰连接数升高是这样引起的：由于业务很繁忙，数据库并发高（每秒读操作3000个，写操作500个），当数据库重启后，就会面临一个问题，如何将之前频繁访问的数据重新加载回buffer中，也就是说，如何对InnoDB Buffer Pool进行预热，以便于快速恢复到之前的性能状态。如果仅靠InnoDB本身去预热buffer，将会是一个不短的时间周期，在业务高峰时，数据库将面临相当大的考验，I/O的瓶颈会带来糟糕的性能。早上8点用户登录时，会连接数据库来验证密码，此时先在内存里找，没有找到就会再到磁盘里找，这样相当于走了两个环节，一个用户登录不上，连接就不会释放，后面的用户只能继续排队，导致连接数升高。

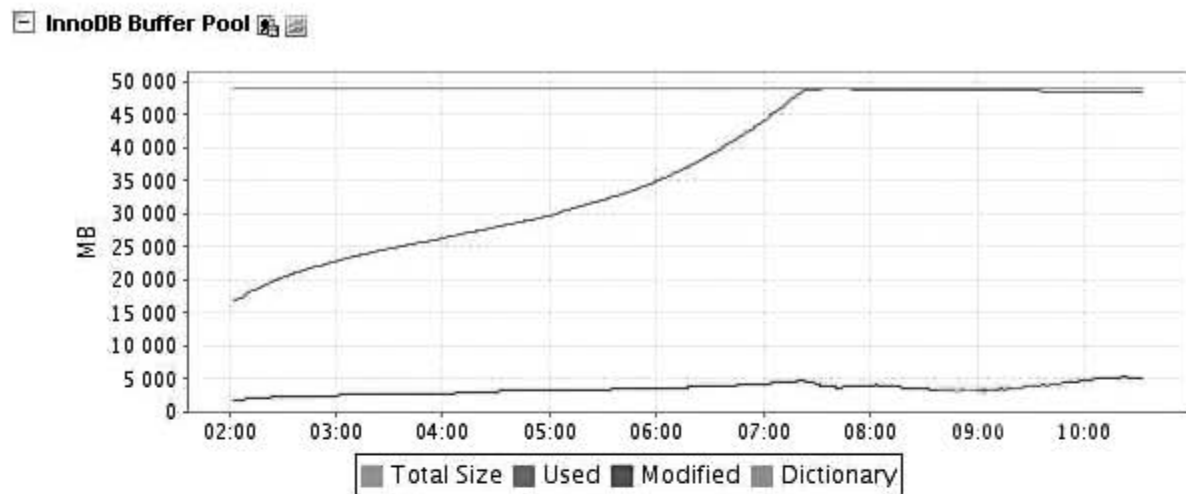


图3-5 InnoDB\_Buffer\_Pool使用情况

解决方法：

在数据库压力很大的情况下，重启完数据库，通过手工执行下列语句，把热数据加载到InnoDB\_Buffer\_Pool缓冲池里进行预热，从而避免了早高峰连接数升高，程序报错。

```
select count(*) from User;
select count(*) from Buddy;
select count(*) from Password;
```

而在MySQL5.6里，为了解决上述问题，提供了一个新特性来快速预热Buffer\_Pool缓冲池。只需在my.cnf里，加入如下命令即可：

```
innodb_buffer_pool_dump_at_shutdown = 1
```

解释：在关闭时把热数据dump到本地磁盘。

```
innodb_buffer_pool_dump_now = 1
```

解释：采用手工方式把热数据dump到本地磁盘。

```
innodb_buffer_pool_load_at_startup = 1
```

解释：在启动时把热数据加载到内存。

```
innodb_buffer_pool_load_now = 1
```

解释：采用手工方式把热数据加载到内存。

在关闭MySQL时，会把内存中的热数据保存在磁盘的ib\_buffer\_pool文件中，该文件位于数据目录下。如图3-6所示。



```
120723 17:47:41 InnoDB: Dumping buffer pool(s) to /usr/local/mysql/data/ib_buffer_pool
120723 17:47:41 InnoDB: Buffer pool(s) dump completed at 120723 17:47:41
```

图3-6 导出热数据到本地文件

在启动MySQL时，会自动加载热数据到Buffer\_Pool缓冲池里，这样，机器重启后热数据会始终保存在内存中，如图3-7所示。

```
120723 17:49:14 InnoDB: Loading buffer pool(s) from /usr/local/mysql/data/ib_buffer_pool
```

图3-7 加载热数据到InnoDB\_Buffer\_Pool



注意 只有在正常关闭MySQL服务，或者pkill mysql时，才会把热数据导出到内存。机器宕机或者pkill-9 mysql，是不会导出的。

下面再来看个“磁盘高负荷把MySQL拖垮”的案例，我们有块业务需要记录大量的LOG日志，插入操作很频繁，这就导致了磁盘高负荷运转，结果就是CPUWait I/O升高，负载加大，来看看CPU的情况，如图3-8所示。

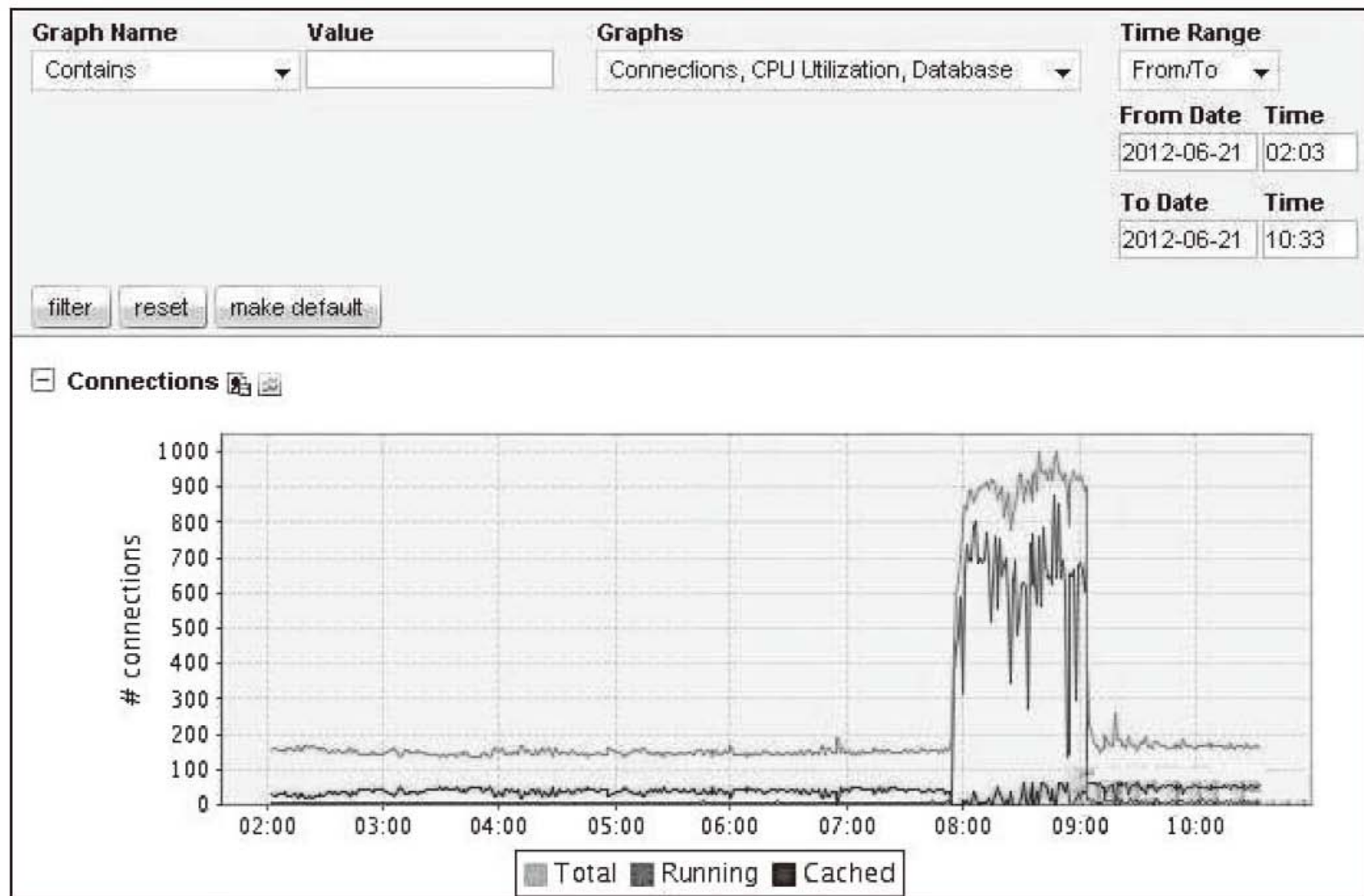


图3-8 CPU的情况

再来看看insert写操作，平均每秒为1000个，可见写操作很大，如图3-9所示。

最后来看看磁盘情况，它处于饱和状态，处理一个I/O请求需要等待15毫秒，如图3-10所示。

此时出现了大量慢日志，而这并不是因为SQL自身存在问题，而是机器压力大造成的。

把情况反馈给了开发，经过程序Bug处理后，再看相应的指标数，如图3-11~图3-14所示。  
可以看到，磁盘I/O已正常，不再处于繁忙状态，处理一个I/O请求只需要等待2.49毫秒。比起之前的15毫秒来，减少了12.51毫秒。

Database Activity

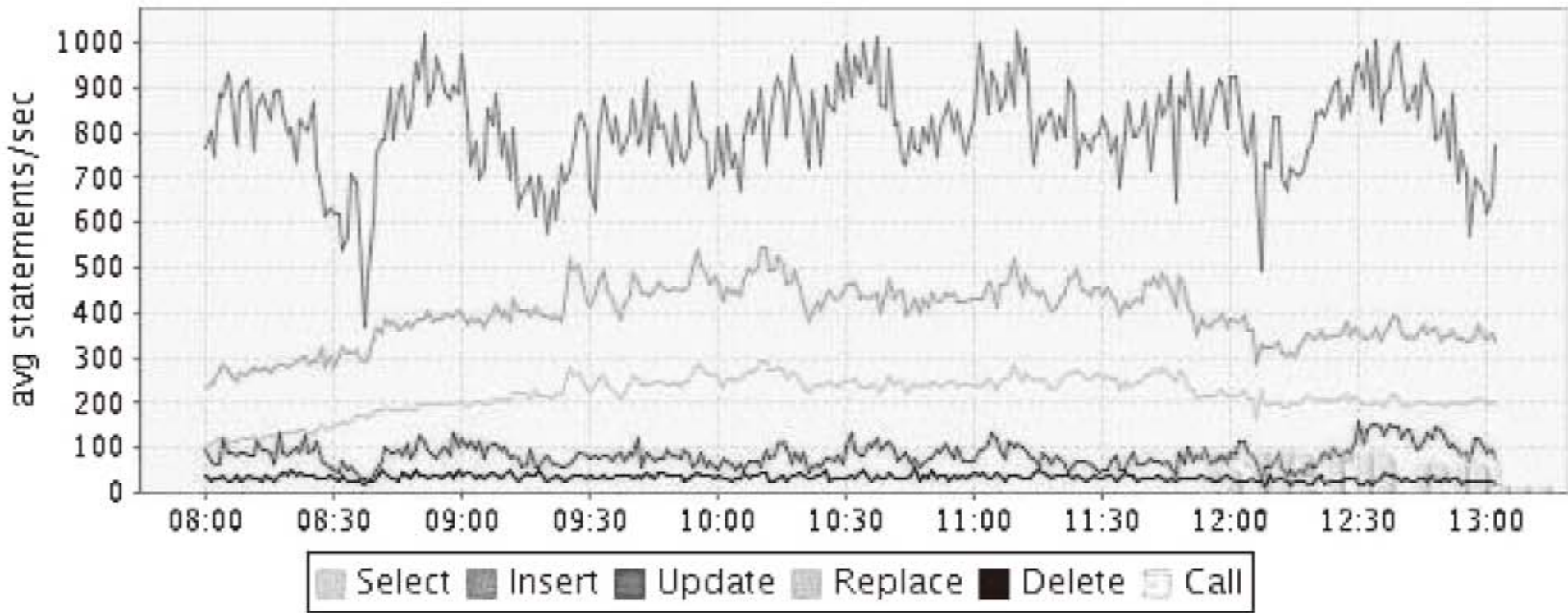


图3-9 数据库性能QPS

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	avgrq-sz	avgqu-sz	await	svctm	%util
cciss/c0d0	0.00	0.00	441.00	199.00	15936.00	24034.00	62.45	10.11	15.24	1.57	100.20
cciss/c0d0p1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
cciss/c0d0p2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
cciss/c0d0p3	0.00	0.00	441.00	199.00	15936.00	24034.00	62.45	10.11	15.24	1.57	100.20

图3-10 磁盘I/O



## CPU Utilization

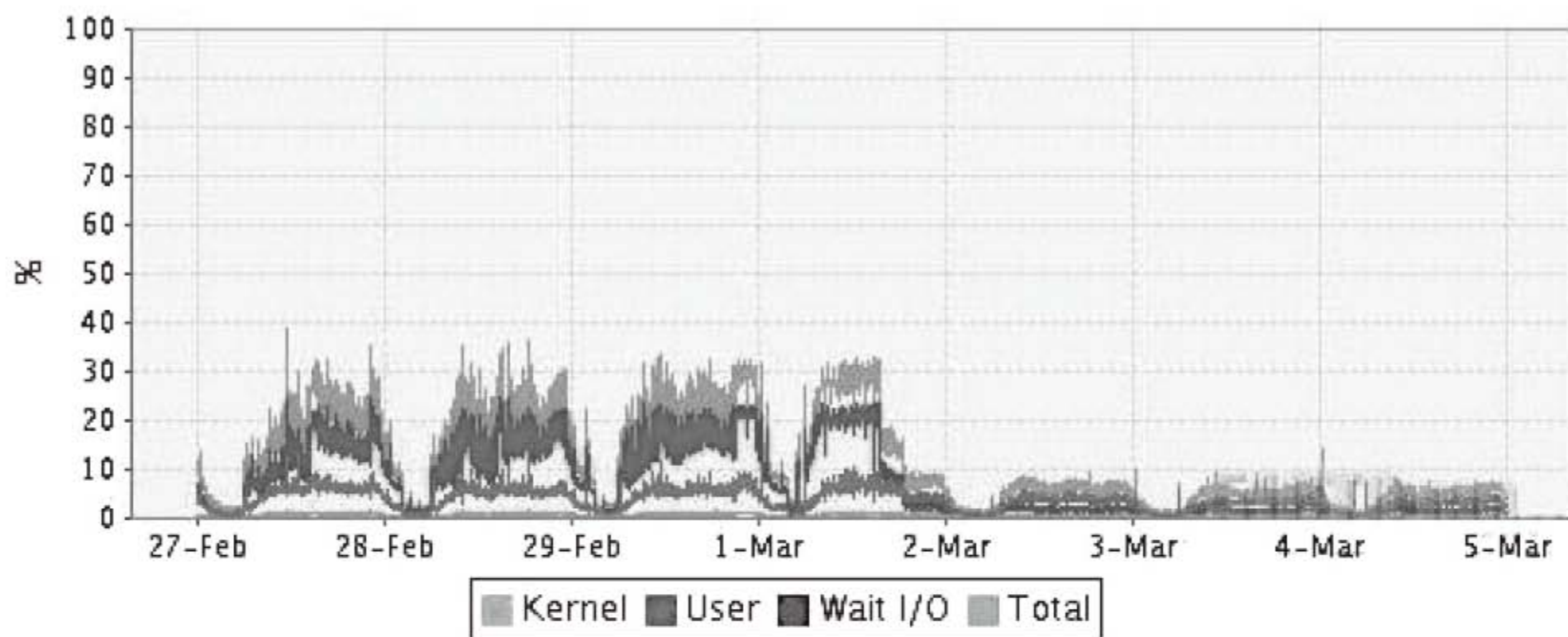


图3-11 CPU的情况

该案例的瓶颈是磁盘I/O繁忙导致CPU负载升高。如果你以后也遇到了这个问题，可以使用一个很简单的方法——加内存，现在内存的价格也很便宜，直接用硬件去解决，把InnoDB\_Buffer\_Pool调整成内存的70%，减少磁盘I/O的压力，当InnoDB\_Buffer\_Pool内存缓冲池大于你的数据量时，此时的性能是最好的，因为所有的数据增删查改操作都是在内存中进行的，内存的速度要比磁盘快得多。

## Database Activity

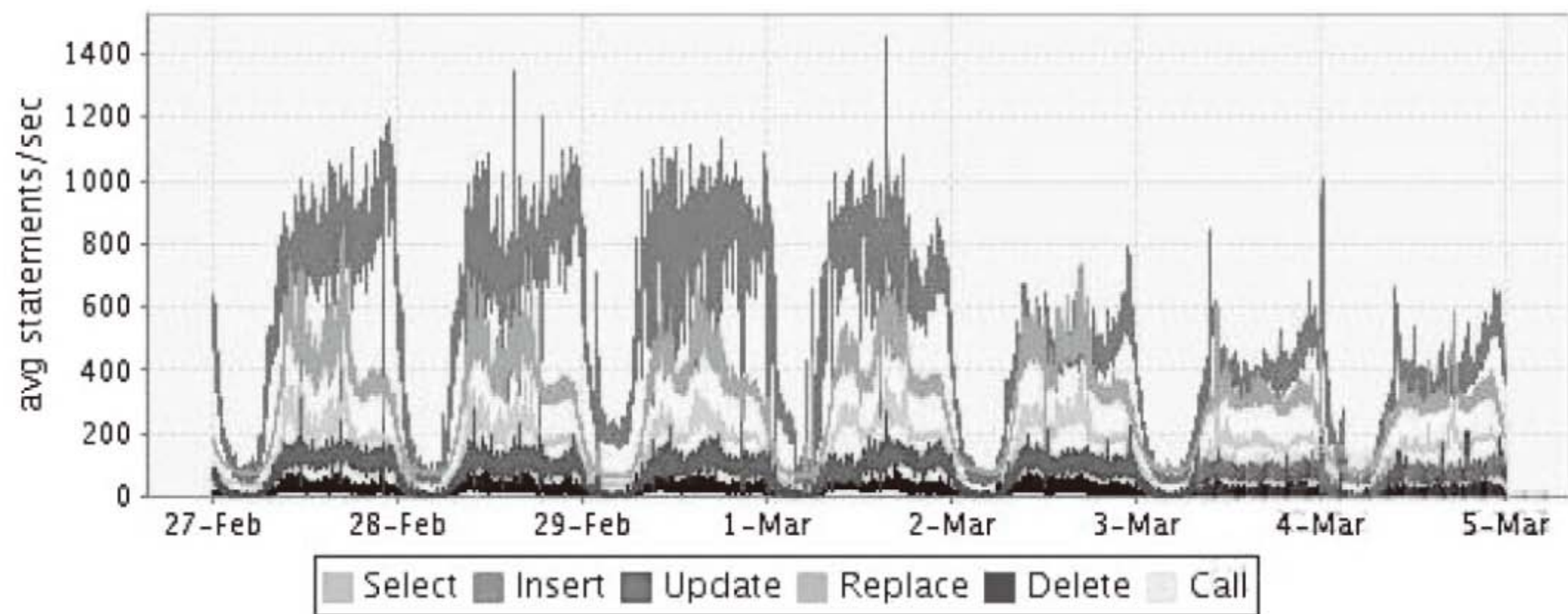


图3-12 数据库性能QPS

## [-] Load Average

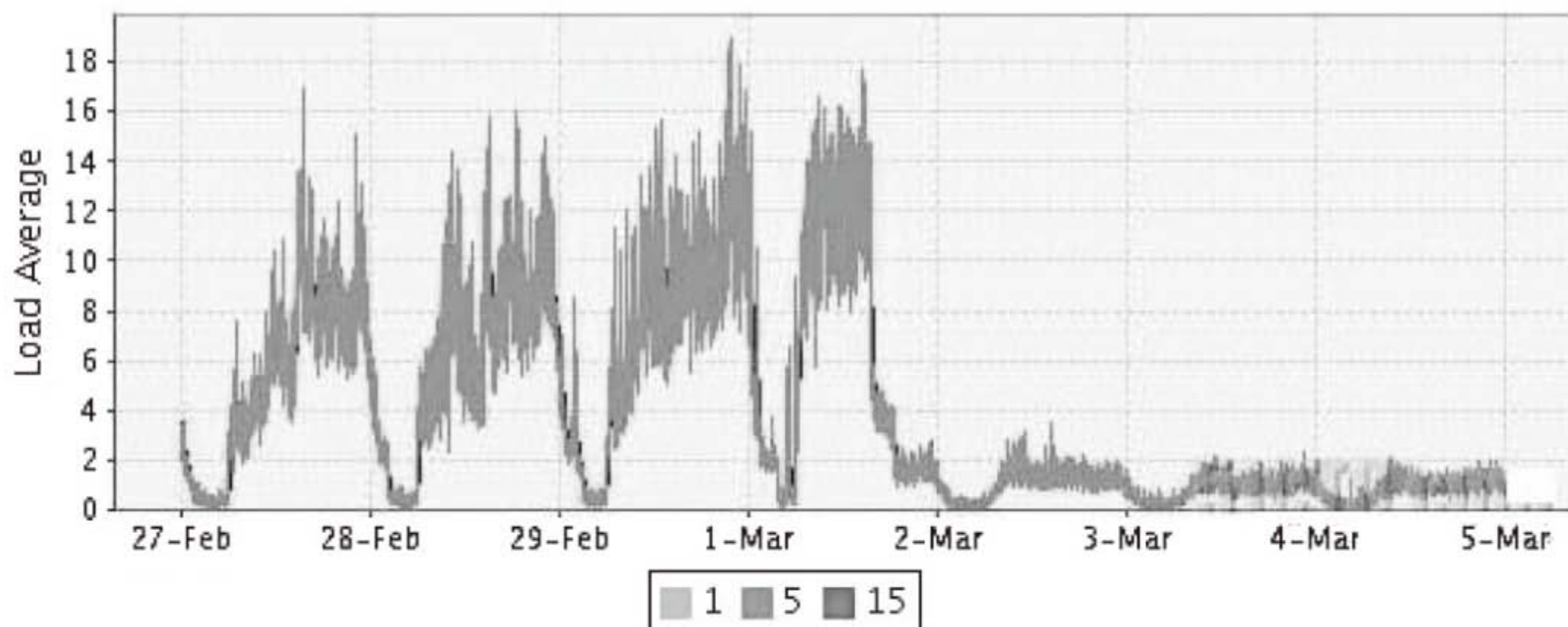


图3-13 CPU负载

```
Device:      rrqm/s  wrqm/s  r/s    w/s    rsec/s  wsec/s  avgrq-sz  avgqu-sz   await  svctm   %util
cciss/c0d0   0.00    71.00  15.00  127.00  480.00  9076.00   67.30     0.35     2.47   2.11   29.90
cciss/c0d0p1 0.00    71.00  0.00   4.00    0.00    600.00   150.00     0.01     1.75   0.75   0.30
cciss/c0d0p2 0.00     0.00  0.00   0.00    0.00     0.00     0.00     0.00     0.00   0.00   0.00
cciss/c0d0p3 0.00     0.00  15.00  123.00  480.00  8476.00   64.90     0.34     2.49   2.14   29.60

[root@CFDB01 ~]#
```



图3-14 磁盘I/O

### 3.3.2 记录子查询引起的宕机

这是2013年5月10日早上10:52分出现的一次故障，由于开发在主库上（生产环境上）运行了一条子查询的慢SQL，导致服务器死机。下面先看下报警信息，如图3-15所示。

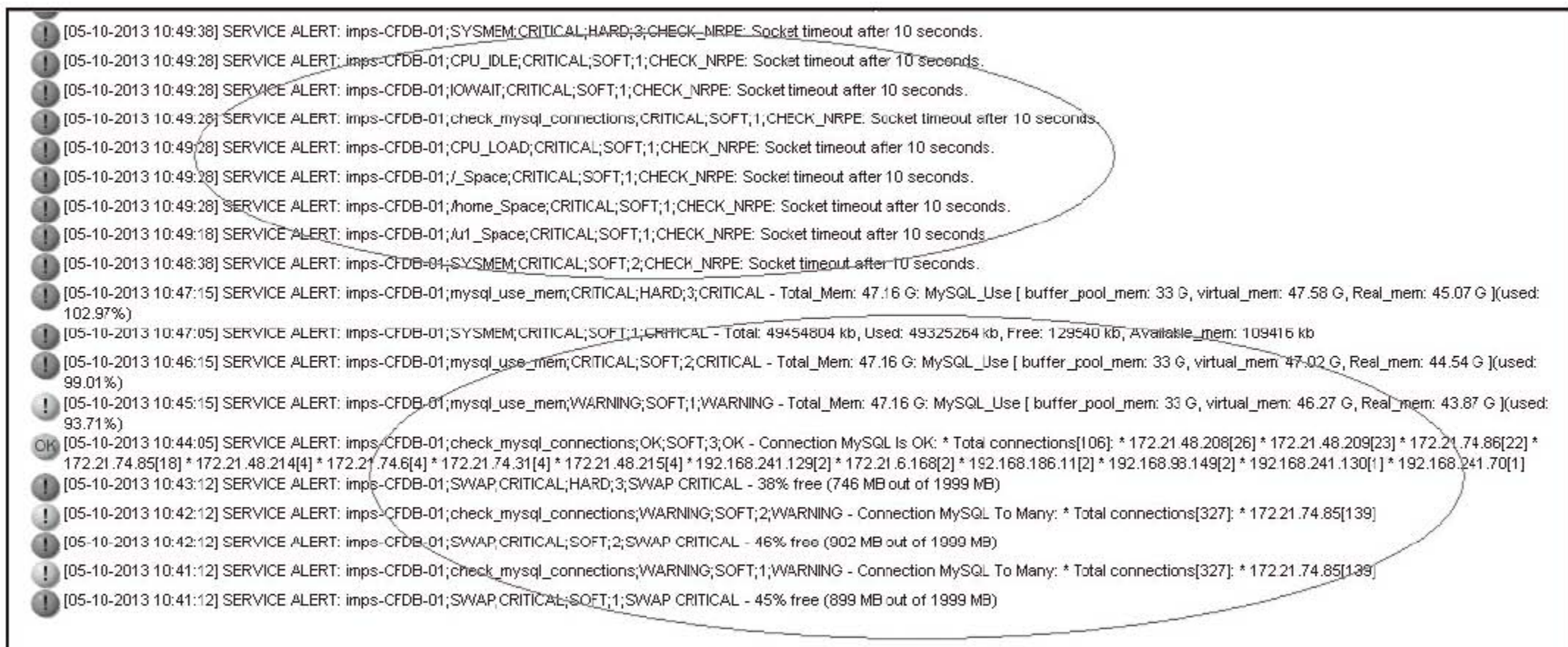


图3-15 Nagios报警

从10:41开始，服务器的swap分区报警，之后内存不足报警，再后来内存耗尽，导致机器死机。通过分析慢日志，我发现是一条统计的SQL语句直接把服务器弄死机了。这肯定是人为执行的，而且是从跳板机98.149上登录SQLYOG上执行的，如图3-16所示。



```
# Time: 130510 10:40:32
# User@Host: admin[admin] @ [192.168.98.149]
# Query_time: 170.078826 Lock_time: 0.000070 Rows_sent: 1 Rows_examined: 12982613
SET timestamp=1368153632;
SELECT * FROM CF_PhysicalStorage
WHERE FileId IN (SELECT FileId FROM CF_TransferCache WHERE CacheId = 'e1998113-0432-4b9c-a4e3-a131a9c25e13');
# Time: 130510 10:40:47
# User@Host: admin[admin] @ [172.21.48.209]
# Query_time: 14.968062 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0
SET timestamp=1368153647;
```

图3-16 慢日志

该SQL语句的执行耗时170秒（猜想是没执行完直接Ctrl+C终止了），而且还是在主库（有业务的机器）上运行的，这导致锁表和其他的进程一直等待，最后越积越多，直接让服务器死机了。

晚上，我在备库上又执行了一次这条SQL，花费了11分26秒（随着这个表的增大，时间还会变慢），如图3-17所示。

```
mysql> SELECT * FROM CF_PhysicalStorage
-> WHERE FileId IN (SELECT FileId FROM CF_TransferCache WHERE CacheId = 'e1998113-0432-4b9c-a4e3-a131a9c25e13');
+-----+-----+-----+
| FileId | ChunkMD5 | PhysicalKey |
| Index | | |
+-----+-----+-----+
| CF41CAC38A5946C60E56952E3DA2EC55 | DF41CAC38A5946C60E56952E3DA2EC55 | group1//MO1/B1/AB/rBUhxFGHIsuufkcOAAEyIk1Qj4k795 |
| 1 | | |
+-----+-----+-----+
1 row in set (11 min 26.91 sec)
```

图3-17 执行时间

为什么这条SQL会执行很慢呢？在MySQL5.1和MySQL5.5的版本里，子查询的性能可谓是相当差，优化器处理不是很好，特别是在WHERE从句中的IN()子查询。对于该查询，MySQL需要先为内层查询语句的查询结果建立一个临时表，然后外层查询语句才能在临时表中查询记录。查询完毕后，MySQL还需要撤销这些临时表。

不过，可以使用连接查询来代替子查询，连接查询不需要建立临时表，其速度比子查询要快。



基于此，上面那条SQL可以这样优化：改用表连接方式（MySQL5.6版本除外），如图3-18所示。

```
mysql> SELECT a.* FROM CF_PhysicalStorage a join
-> (SELECT FileId FROM CF_TransferCache WHERE CacheId = 'e1998113-0432-4b9c-a4e3-a131a9c25e13') b
-> on a.FileId=b.FileId;

+-----+-----+-----+
| FileId | ChunkMD5 | PhysicalKey |
| Index |          |             |
+-----+-----+-----+
| OF41CAC38A5946C60E56952E3DA2EC55 | OF41CAC38A5946C60E56952E3DA2EC55 | group1//M01/B1/AB/rBUhxFGHIsuufkc0AAEyIk1Qj4k7958617 |
| 1 |          |             |
+-----+-----+-----+

1 row in set (0.01 sec)

mysql>
```

图3-18 优化后执行时间

优化后，只需要0.01秒就出结果。

所以，这里要提醒大家一下，凡是这种统计的SQL，千万不要在主库上运行，子查询目前的性能还很差。

一般情况下，开发人员写的SQL里，很多查询都需要使用子查询，子查询可以使查询语句更灵活、易懂，但子查询的执行效率不高，解决的方案是上面所说的用表连接代替子查询。在MySQL5.6以前的版本里，子查询仅仅被看成是一个功能，生产环境下很少使用到。而在MySQL5.6版本里，针对这个问题进行了强劲的优化，这意味着，你可以在生产环境下使用子查询。

下面将分别在MySQL5.5和MySQL5.6里对该功能进行演示，帮助大家了解其特性，如图3-19和图3-20所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.19    |
+-----+
1 row in set (0.01 sec)

mysql> select count(*) from test1 where tid in (select id from test3);
+-----+
| count(*) |
+-----+
|      9992 |
+-----+
1 row in set (1.08 sec)
```

图3-19 MySQL5.5子查询执行时间

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.5-m8   |
+-----+
1 row in set (0.04 sec)

mysql> select count(*) from test1 where tid in (select id from test3);
+-----+
| count(*) |
+-----+
|      9995 |
+-----+
1 row in set (0.18 sec)
```

图3-20 MySQL5.6子查询执行时间

可以看到，在MySQL5.6里，子查询的速度比MySQL5.5快了10倍。之后，再用优化器对其进行优化，效果如图3-21和图3-22所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.19    |
+-----+
1 row in set (0.02 sec)

mysql> explain select count(*) from test1 where tid in (select id from test3);
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | PRIMARY | test1 | index | NULL | tid | 5 | NULL | 10365 | Using where; Using index |
| 2 | DEPENDENT SUBQUERY | test3 | unique_subquery | PRIMARY | PRIMARY | 4 | func | 1 | Using index |
+-----+
2 rows in set (0.05 sec)
```

图3-21 MySQL5.5执行计划



```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.5-m8 |
+-----+
1 row in set (0.01 sec)

mysql> explain select count(*) from test1 where tid in (select id from test3);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | test1 | index | tid | tid | 5 | NULL | 1 | Using where; Using index |
| 1 | SIMPLE | test3 | eq_ref | PRIMARY | PRIMARY | 4 | test.test1.tid | 1 | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.02 sec)
```

图3-22 MySQL5.6执行计划

可以看出，实际上是把子查询改写成了join方式。现在可以放心大胆地去用子查询了。

### 3.3.3 诊断事务量突高的原因

在介绍本节案例之前，先来看看二进制日志相关知识。二进制日志由配置文件的log-bin选项负责启用，MySQL服务器将在数据根目录创建两个新文件XXX-bin.001和xxx-bin.index，若配置选项没有给出文件名，MySQL将使用主机名称命名这两个文件，其中.index文件包含一份全体日志文件的清单。MySQL会把用户对所有数据库的内容和结构的修改情况记入XXX-bin.n文件中，而不会记录SELECT和没有实际更新的UPDATE语句。

二进制日志有两个作用，一是恢复数据，比如，早上9点误删除了一张表，那么可以通过凌晨的全量备份，加上凌晨到9点之前的二进制日志文件做增量恢复。二是实现MySQL的主从复制。

下面进入正题，快下班时收到Nagiso报警短信，说更新和插入阈值报警，于是登录到MySQL Monitor上查看，确实如此，如图3-23所示。

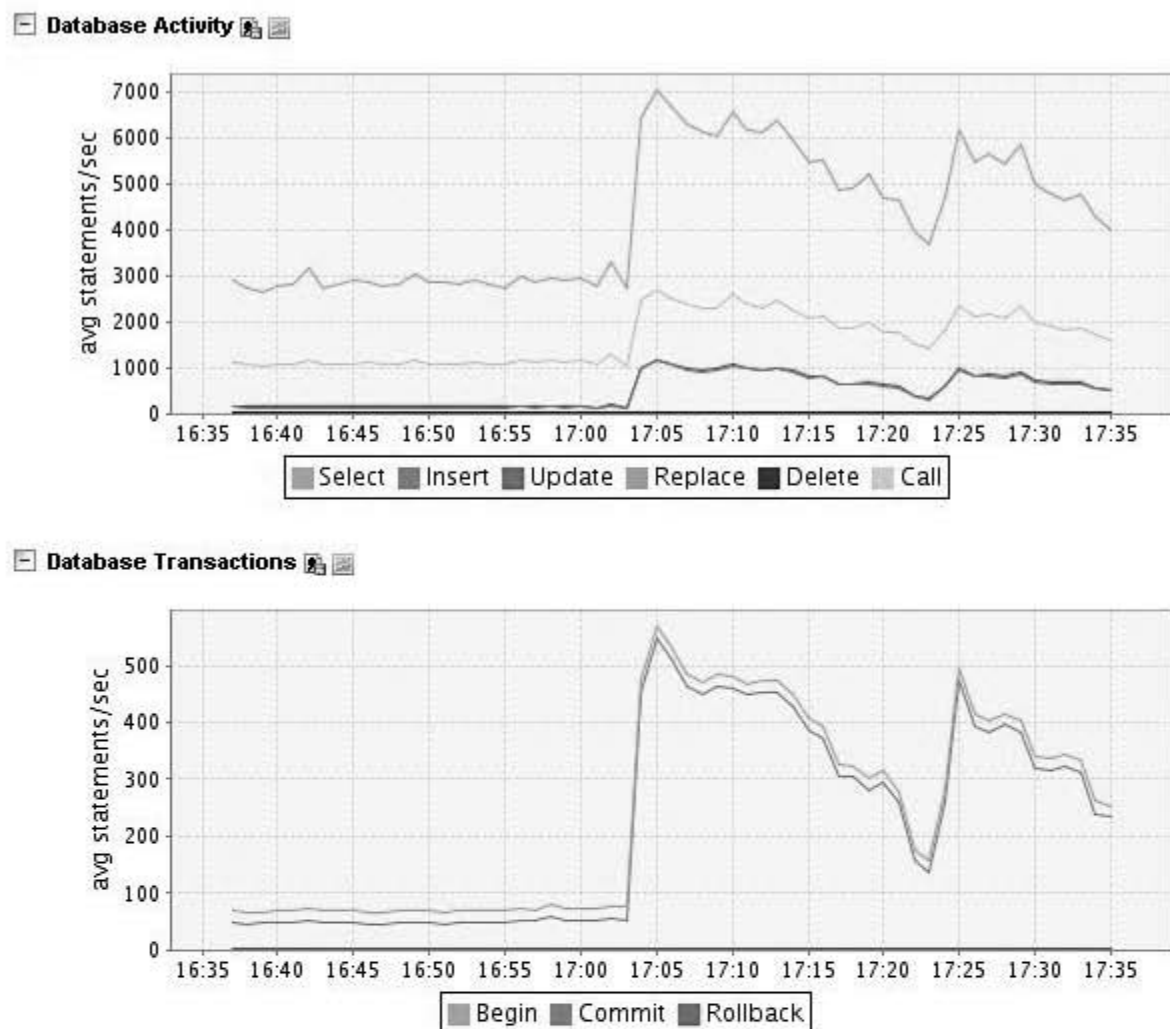


图3-23 数据库性能QPS

看到这个现象，于是登录MySQL服务器上，通过binlog来分析17:05数据库发生抖动之前和之后的表，看看哪个更新较大。

下面是17:05之前的binlog日志：

```
[root@XXX-02 logs]# mysqlbinlog --no-defaults --base64-output=decode-rows -v -v mysql-bin.053373 |more
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE, COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#120802 16:36:47 server id 4713306 end_log_pos 106 Start: binlog v 4, server v 5.1.43sp1-br38368-enterprise-gpl-pro-log c
reated 120802 16:36:47
```



然后用下面这条命令分析，找出写操作频繁的表：

```
[root@XXX-02 logs]# mysqlbinlog --no-defaults --base64-output=decode-rows -v -v mysql-bin.053373 |awk '/###/{if($0~/UPDATE|INSERT|DELETE/)count[$2" "$NF]++}END{for(i in count)print i, "\t", count[i]}' | column -t | sort -k3nr | more
```

```
UPDATE DB.Dynamic 133971
```

```
UPDATE DB.User 54834
```

```
UPDATE DB.Quota 24938
```

```
UPDATE DB.OrderHistory 24482
```

```
UPDATE DB.BOSSOperation 19767
```

```
UPDATE DB.SmsCount 18235
```

```
UPDATE DB.Buddy 10919
```

```
INSERT DB.Buddy_Log 10024
```

```
=====
```

接着查看17:05之后的binlog日志，如下所示：

```
[root@XXX-02 logs]# mysqlbinlog --no-defaults --base64-output=decode-rows -v -v mysql-bin.053375 |more
```

```
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
```

```
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE, COMPLETION_TYPE=0*/;
```

```
DELIMITER /*!*/;
```

```
# at 4
```

```
#120802 17:10:54 server id 4713306 end_log_pos 106 Start: binlog v 4, server v 5.1.43sp1-br38368-enterprise-gpl-pro-log c
```

```
reated 120802 17:10:54
```

```
# at 106
```

然后用这条命令分析，找出写操作频繁的表：

```
[root@XXX-02 logs]# mysqlbinlog --no-defaults --base64-output=decode-rows -v -v mysql-bin.053375 |awk '/###/{if($0~/UPDATE|INSERT|DELETE/)count[$2" "$NF]++}END{for(i in count)print i, "\t", count[i]}' | column -t | sort -k3nr
```

```
INSERT DB.Buddy_Log 194160
```

```
INSERT DB.Buddy 192587
```

```
UPDATE DB.Dynamic 62767
```

```
UPDATE DB.User 30103
```

```
UPDATE DB.OrderHistory 12507
```

*UPDATE DB.Quota 12318*

*UPDATE DB.BOSSOperation 9892*

这样就比较直观地显示出了哪些表的更新较多，之后再找开发人员确认问题，了解是否是业务增长导致的。

这次故障是演示了如何通过binlog日志来分析业务增长量，哪个表写操作频繁，如果以后你也遇到了此类问题，同样可以采用此方法来分析。



注意 此命令只支持MySQL5.1以上版本，innodb引擎、READ-COMMITTED隔离级别、或者binlog\_format的格式为ROW。如果读者没有看明白什么是隔离级别或者binlog的格式，在下一节会有介绍。

### 3.3.4 谨慎设置binlog\_format=MIXED

binlog\_format有三种格式，STATEMENT、ROW和MIXED。

STATEMENT在二进制日志里，记录的是实际的SQL语句，如图3-24所示。

ROW在二进制日志里记录的不是简单的SQL语句，而是实际行的变更，如图3-25所示。

```
# at 175
#130521 15:56:04 server id 140  end_log_pos 263
use test/*!*/;
SET TIMESTAMP=1369122964/*!*/;
insert into t2 values(11)
/*!*/;
# at 263
#130521 15:56:04 server id 140  end_log_pos 290
COMMIT/*!*/;
# at 290
#130521 15:56:06 server id 140  end_log_pos 358
SET TIMESTAMP=1369122966/*!*/;
BEGIN
/*!*/;
# at 358
#130521 15:56:06 server id 140  end_log_pos 446
SET TIMESTAMP=1369122966/*!*/;
insert into t2 values(12)
/*!*/;
# at 446
```

图3-24 STATEMENT格式



```
#130522 15:39:57 server id 140  end_log_pos 336
### UPDATE test.t2
### WHERE
###   @1=77 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=78 /* INT meta=0 nullable=0 is_null=0 */
### UPDATE test.t2
### WHERE
###   @1=88 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=89 /* INT meta=0 nullable=0 is_null=0 */
### UPDATE test.t2
### WHERE
###   @1=11 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=12 /* INT meta=0 nullable=0 is_null=0 */
### UPDATE test.t2
### WHERE
###   @1=12 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=13 /* INT meta=0 nullable=0 is_null=0 */
### UPDATE test.t2
### WHERE
```

图3-25 ROW格式

在二进制日志里，MIXED默认还是采用STATEMENT格式记录的，但在下面这6种情况下会转化为ROW格式，见手册：

### 5.2.4.3. Mixed Binary Logging Format

When running in `MIXED` logging format, the server automatically switches from statement-based to row-based logging under the following conditions:

- When a DML statement updates an `NDBCLUSTER` table.
- When a function contains `UUID()`.
- When one or more tables with `AUTO_INCREMENT` columns are updated and a trigger or stored function is invoked. Like all other unsafe statements, this generates a warning if `binlog_format = STATEMENT`.

For more information, see [Section 16.4.1.1, “Replication and `AUTO\_INCREMENT`”](#).

- When any `INSERT DELAYED` is executed.
- When the body of a view requires row-based replication, the statement creating the view also uses it. For example, this occurs when the statement creating a view uses the `UUID()` function.
- When a call to a UDF is involved.
- If a statement is logged by row and the session that executed the statement has any temporary tables, logging by row is used for all subsequent statements (except for those accessing temporary tables) until all temporary tables in use by that session are dropped.

This is true whether or not any temporary tables are actually logged.

第一种情况：NDB引擎，表的DML操作增、删、改会以ROW格式记录。

第二种情况：SQL语句里包含了UUID()函数。

第三种情况：自增长字段被更新了。

第四种情况：包含了INSERT DELAYED语句。

第五种情况：使用了用户定义函数（UDF）。

第六种情况：使用了临时表。

下面我们来看一个实例，在该实例中，主从都是MySQL5.5，binlog\_format被设置为了MIXED格式，而且使用的默认隔离级别为

REPEATABLE-READ，我们来看看这样设置会有什么问题。

首先来看看master上的数据，如下所示：

```
1. mysql>select * from t2;
2. +-----+
3. | id |
4. +-----+
5. | 1 |
6. | 2 |
7. | 3 |
8. | 4 |
9. | 5 |
10. | 6 |
11. | 7 |
12. | 8 |
13. | 9 |
14. | 10 |
15. +-----+
16. 10 rows in set (0.00 sec)
```

接着看看slave上的数据，如下所示：

```
1. mysql>select * from t2;
2. +-----+
3. | id |
4. +-----+
5. | 1 |
6. | 2 |
7. | 3 |
8. | 4 |
9. | 5 |
10. +-----+
```



11. 5 rows in set (0.00 sec)

那么我在master上执行如下命令：

1. *mysql>update t2 set id=99 where id=9;*

2. *Query OK, 1 row affected (0.01 sec)*

3. *Rows matched: 1 Changed: 1 Warnings: 0*

你说slave上会报错吗？答案是：NO，不会报错，没有任何提示。不信吗？各位可按照上面的内容测试一下。那为啥不会报错呢？因为在二进制日志里虽然采用的是MIXED格式，但默认还是采用的STATEMENT格式记录的，只有几种特殊情况才会以ROW格式记录，所以我们查看binlog会发现记录的是一条update t2 set id=99 where id=9的SQL语句，这条SQL在slave机器上执行后，由于没有发现id=7的这条记录，影响的行数为空，因此主从复制进程并不会中断。但是在这种情况下，主从数据是很不安全的，很容易出现数据不一致的情况，而且MySQL并不会发出任何报警信息。在MySQL5.1以后的版本中，通过采用ROW格式解决了这个问题。

在改为binlog\_format=ROW格式后，再次执行刚才的语句，就会报错，如下所示：

1. *mysql>update t2 set id=99 where id=9;*

2. *Query OK, 1 row affected (0.00 sec)*

3. *Rows matched: 1 Changed: 1 Warnings: 0*

执行结果如图3-26所示。

```
[root@hadoop-datanode5 data]# mysqlbinlog --no-defaults --base64-output=decode-rows -v -v mysql-bin.000002 | grep -A 10 '517'
#121011 16:12:44 server id 140  end_log_pos 517          Update_rows: table id 73 flags: STMT_END_F
### UPDATE test.t2
### WHERE
###   @1=9 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=99 /* INT meta=0 nullable=0 is_null=0 */
# at 517
#121011 16:12:44 server id 140  end_log_pos 544          Xid = 150
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
[root@hadoop-datanode5 data]#
```

图3-26 同步报错

如果你采用的是默认隔离级别REPEATABLE-READ，那么建议设置binlog\_format=ROW。如果是READ-COMMITTED隔离级别，binlog\_format=MIXED和binlog\_format=ROW的效果是一样的，binlog记录的格式都是ROW，很神奇吧？你去试试就知道了。ROW格式对主从复制来说是很安全的参数。

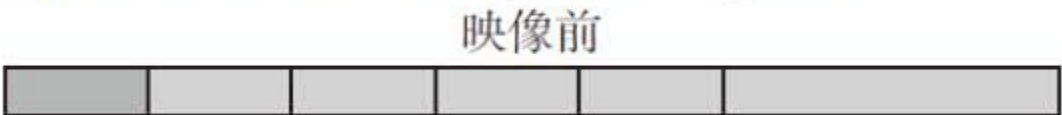
之前有个网友问我，如果binlog\_format为row格式，那么数据量很大的时候，日志量就会很大了，再一个就是写日志所带来的I/O问题是否也要考虑进来呢？

这个问题问得相当好，没错，如果设置为row格式，binlog文件会比STATEMENT格式大很多，而且主从复制是通过binlog来传输的，binlog的增大也增加了网络开销，这就是一把双刃剑，看你如何取舍了，你是要安全稳定性，还是要性能？就我管理的机器上来看，若以15000转希捷硬盘做RAID10，每秒写操作在200个/秒还是没有问题的。

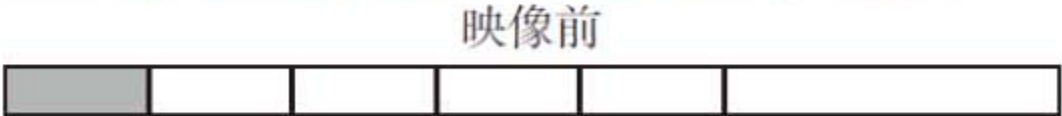
不过，在MySQL5.6里，针对这位网友提出的这个问题进行了优化，可以通过设置binlog\_row\_image=minimal加以解决，如图3-27所示。

也就是说，binlog里只记录影响后的行，从而减少了binlog的增长量大小。下面来演示一下，表里的数据如图3-28所示。注意，我的binlog\_row\_image是默认的FULL，下面执行一条更新语句，如图3-29所示。

MySQL5.6 之前或 binlog-row-imaeg=full



采用 MySQL5.6, 或 binlog-row-imaeg=minimal

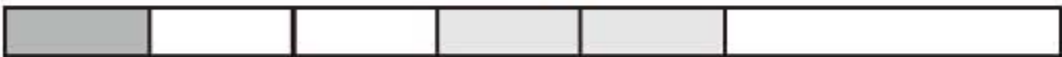


主键

映像后



映像后



更改的列

图3-27 binlog\_row\_image



```
mysql> select * from hcy;
+-----+-----+
| id | name |
+-----+-----+
| 1 | majingjing |
| 2 | wangyanyan |
| 3 | hechunyang |
| 4 | zhaowei |
+-----+-----+
4 rows in set (0.03 sec)

mysql> show variables like 'binlog_row_image'
+-----+-----+
| Variable_name | Value |
+-----+-----+
| binlog_row_image | FULL |
+-----+-----+
1 row in set (0.04 sec)
```

图3-28 binlog\_row\_image设置为FULL

```
mysql> update hcy set name='hcymysql' where id=3;
Query OK, 1 row affected (0.08 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

图3-29 更新

然后再到binlog里去查看，如图3-30所示。

```
# at 820
#130522 18:11:49 server id 25 end_log_pos 386 CRC32 0x67ae4659
### UPDATE `test`.`hcy`
### WHERE
###   @1=3 /* INT meta=0 nullable=1 is_null=0 */
###   @2='hechunyang' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
###   @1=3 /* INT meta=0 nullable=1 is_null=0 */
###   @2='hcymysql' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 386
```



图3-30 binlog信息

可以看到，在binlog里会把所有影响的行都给记录下来。下面把binlog\_row\_image设置为minimal，再执行一条更新语句，如图3-31所示。

然后再到binlog里去查看，如图3-32所示。

如果你仔细和上面的图3-30和图3-32进行对比，就会发现区别，没错，这里只记录了影响的那一行记录。

至于REPEATABLE-READ和READ-COMMITTED有什么区别，将在优化那一章加以介绍。

```
mysql> set global binlog_row_image = 'minimal';
Query OK, 0 rows affected (0.01 sec)

mysql> set binlog_row_image = 'minimal';
Query OK, 0 rows affected (0.02 sec)

mysql> show variables like 'binlog_row_image';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| binlog_row_image | MINIMAL |
+-----+-----+
1 row in set (0.04 sec)

mysql> update hcy set name='WYY' where id=2;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

图3-31 binlog\_row\_image设置为MINIMAL



```
# at 586
#130522 18:16:30 server id 25  end_log_pos 643 CRC32 0x90844821
### UPDATE `test`.`hcy`
### WHERE
###   @1=2 /* INT meta=0 nullable=1 is_null=0 */
###   @2='wangyanyan' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
###   @2='WYY' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 643
#130522 18:16:30 server id 25  end_log_pos 674 CRC32 0x8aa7bf8c
COMMIT/*!*/;
```

图3-32 binlog信息

### 3.3.5 未设置swap分区导致内存耗尽，主机死机

这是一个真实的案例，在2009年和2010年的时候，我负责管理的一个主机多次被Hang死，当时的情况是主机可以ping通，但ssh却连接不上，导致cluster无法切换，只有人工去重启那台被Hang死的机器，才能正常切换。

该主机内存是72 GB，其中MySQL (InnoDB\_Buffer\_Pool) 内存缓冲池所用的内存为48 GB，但在机器正常的情况下，使用free-m命令查看，会发现InnoDB\_Buffer\_Pool内存已经超过了48 GB，为什么会出现这种情况？这是因为其他程序或者系统模块需要额外的内存，比如，cp一个大文件，或者在主库上mysqldump数据库等，当物理内存不够用的时候，操作系统就会把MySQL所拥有的一部分地址空间映射到swap上去。

在这种情况下，我们设置echo 0>/proc/sys/vm/swappiness来禁止使用swap，并且没有建立swap分区，这样做是考虑到将分配给MySQL的地址空间映射到swap上，会导致性能有突起波动。但是如此设置后似乎并没有解决问题，还是存在主机被Hang死的情况。

在百思不得其解的情况下，后来参考了淘宝《MySQL如何避免使用swap》这篇文章

②

，才知道将swappiness设置为0，只能减少使用swap的概率，并不能避免Linux使用swap，所以内存被耗尽后，主机仍旧会被Hang死了。

找到原因后，解决起来也简单，我们后来的解决方案是：

- 1) 增加2 GB的swap分区，避免内存耗尽时机器死机，可以给予些缓冲，重启前端程序释放MySQL压力。
  - 2) 增加内存监控，当内存使用率达到90%，通过重启MySQL来释放内存，避免机器死机（内存Nagios监控脚本会在监控章节介绍）。
- [http://www.taobaodba.com/html/552\\_mysql\\_avoid\\_swap.html](http://www.taobaodba.com/html/552_mysql_avoid_swap.html)。



### 3.3.6 MySQL故障切换之事件调度器注意事项

注

先来普及下基础知识：事件调度器（event）是在MySQL5.1中新增的另一个特色功能，可以作为定时任务调度器，取代部分原先只能用操作系统任务调度器才能完成的定时功能。而且MySQL的事件调度器可以实现每秒钟执行一个任务，这在一些对实时性要求较高的环境下非常实用。不过，在使用这个功能之前必须确保event\_scheduler已开启，可执行如图3-33所示的命令。

```
mysql> set global event_scheduler = 1;
Query OK, 0 rows affected (0.00 sec)

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

图3-33 事件调度器开启

下面针对事件调度器在主从切换时具体会有什么影响，做了一个测试：

在主从架构中，在master创建一个event，如下所示：

```
mysql> show create event 'insert'\G;
```

```
***** 1. row *****
```

```
Event: insert
```

```
sql_mode:
```

```
time_zone: SYSTEM
```

```
Create Event: CREATE DEFINER='root'@'localhost' EVENT 'insert'
```

```
ON SCHEDULE EVERY 1 MINUTE STARTS '2012-11-20 16:10:09'
```

```
ON COMPLETION PRESERVE ENABLE DO BEGIN
```

```
insert into t3(name) values('aa');
```

```
END
```

```
character_set_client: utf8
```

*collation\_connection: utf8\_general\_ci*  
*Database Collation: utf8\_general\_ci*  
*1 row in set (0.02 sec)*

使用slave进行同步，结果是这样的，注意粗斜体字：

*MySQL>show create event 'insert'\G;*

*\*\*\*\*\* 1. row \*\*\*\*\**

*Event: insert*

*sql\_mode:*

*time\_zone: SYSTEM*

*Create Event: CREATE DEFINER='root'@'localhost' EVENT 'insert'*

*ON SCHEDULE EVERY 1 MINUTE STARTS '2012-11-20 16:10:09' ON COMPLETION PRESERVE DISABLE ON slave DO BEGIN*

*insert into t3(name) values('aa');*

*END*

*character\_set\_client: utf8*

*collation\_connection: utf8\_general\_ci*

*Database Collation: utf8\_general\_ci*

*1 row in set (0.02 sec)*

再回过头来看一下事件状态，在master和slave上的事件状态如图3-34和图3-35所示（注意粗斜体内容）。

```
mysql> show events\G;
***** 1. row *****
      Db: test
      Name: insert
      Definer: root@localhost
      Time zone: SYSTEM
      Type: RECURRING
      Execute at: NULL
      Interval value: 1
      Interval field: MINUTE
      Starts: 2012-11-20 16:10:09
      Ends: NULL
      Status: ENABLED
      Originator: 22
character_set_client: utf8
collation_connection: utf8_general_ci
  Database Collation: utf8_general_ci
1 row in set (0.09 sec)
```

图3-34 在master上的事件状态



```
mysql> show events\G;
***** 1. row *****
      Db: test
      Name: insert
      Definer: root@localhost
      Time zone: SYSTEM
      Type: RECURRING
      Execute at: NULL
      Interval value: 1
      Interval field: MINUTE
      Starts: 2012-11-20 16:10:09
      Ends: NULL
      Status: SLAVESIDE_DISABLED
      Originator: 22
character_set_client: utf8
collation_connection: utf8_general_ci
      Database Collation: utf8_general_ci
1 row in set (0.09 sec)
```

图3-35 在slave上的事件状态

也就是说，事件只能在master上触发，在slave上不能触发，如果slave上触发了，同步复制就会坏掉。当主从故障切换之后，VIP漂移到了以前的slave上，此时slave就成了新的master。但这时，事件的状态还是维持着slaveSIDE\_DISABLED，并不是也改成了ENABLED，这样就会造成切换以后，事件无法执行。所以，需要人工重新开启事件状态。命令如下所示：

```
mysql> alter event 'insert' enable;
Query OK, 0 rows affected (0.07 sec)
```

参考手册：

- **Status:** The event status. One of `ENABLED`, `DISABLED`, or `SLAVESIDE_DISABLED`.

`SLAVESIDE_DISABLED` indicates that the creation of the event occurred on another MySQL server acting as a replication master and replicated to the current MySQL server which is acting as a slave, but the event is not presently being executed on the slave.

附上事件状态截图，如图3-36所示。



```
1 DELIMITER $$
2
3 ALTER DEFINER='root'@'localhost' EVENT 'insert'
4 ON SCHEDULE EVERY 1 MINUTE STARTS '2012-11-20 16:10:09'
5 ON COMPLETION PRESERVE ENABLE DO BEGIN
6
7     INSERT INTO t3(NAME) VALUES('aa');
8
9     END$$
10 DELIMITER ;
```

图3-36 事件SQL

- 该问题的发现者是笔者的前同事李冉，本人对其进行了校验与测试。

### 3.3.7 人工误删除InnoDB ibdata数据文件，如何恢复

常在群里看到，有人因为不熟悉InnoDB引擎，而误删了InnoDB ibdata（数据文件）和ib\_logfile（redo log重做事务日志文件），结果导致了悲剧的发生。如果有做主从复制同步，那还好，如果是单机呢？如何恢复？

下面就来模拟生产环境下，人为误删除数据文件和重做日志文件，是如何恢复的。

1) 用Sysbench模拟数据的写入，如下所示：

```
sysbench --test=oltp
--mysql-table-engine=innodb
--oltp-table-size=10000000
--max-requests=10000
--num-threads=90
--mysql-host=192.168.110.140
--mysql-port=3306
--mysql-user=admin
--mysql-password=123456
--mysql-db=test
--oltp-table-name=uncompressed
--mysql-socket=/tmp/mysql.sock run
```

2) 使用rm-f ib\*删除数据文件和重做日志文件。

下面就来具体看看如何恢复。

若此时你发现数据库还可以正常工作，数据照样可以写入，切记，这时千万别把mysqld进程杀死，否则没法挽救。

接下来，先找到mysqld的进程pid，如下所示：

```
# netstat -ntlp | grep mysqld
tcp 0 0 0.0.0.0:3306 0.0.0.0:* LISTEN 30426/mysqld
```

这里是30426。

之后要执行很关键的一步，输入如下命令，并查看结果：



```
# ll /proc/30426/fd | egrep 'ib_|ibdata'
lrwx----- 1 root root 64 9月24 16:51 10 ->/u2/mysql/data/ib_logfile1
lrwx----- 1 root root 64 9月24 16:51 11 ->/u2/mysql/data/ib_logfile2
lrwx----- 1 root root 64 9月24 16:51 4 ->/u2/mysql/data/ibdata1
lrwx----- 1 root root 64 9月24 16:51 9 ->/u2/mysql/data/ib_logfile0
```

其中，10、11、4、9就是我们要恢复的文件。

这时，你可以把前端业务关闭，或者执行：

```
FLUSH TABLES WITH READ LOCK;
```

这一步的作用是让数据库没有写入操作，以便完成后面的恢复工作。

如何验证没有写入操作呢？分以下几步，记住要结合在一起观察。

先输入以下命令，让脏页尽快刷入到磁盘里。

```
set global innodb_max_dirty_pages_pct=0;
```

然后查看binlog日志写入情况，确保File和Position的值没有在变化。

```
mysql> show master status;
```

```
+-----+-----+-----+-----+
| File | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000002 | 107 | | |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

最后再查看InnoDB状态信息，确保脏页已经刷入磁盘。

```
show engine innodb status\G;
```

```
-----
TRANSACTIONS
-----
Trx id counter A21837
Purge done for trx's n:o <A21837 undo n:o <0
## 确保后台Purge进程把undo log全部清除掉，事务ID要一致。
-----
```

## INSERT BUFFER AND ADAPTIVE HASH INDEX

-----  
*Ibuf: size 1 , free list len 65 , seg size 67 , 0 merges*

*## insert buffer合并插入缓存等于1*

## LOG

-----  
*Log sequence number 18158813743*

*Log flushed up to 18158813743*

*Last checkpoint at 18158813743*

*## 确保这3个值不在变化*

## BUFFER POOL AND MEMORY

-----  
*Total memory allocated 643891200; in additional pool allocated 0*

*Dictionary memory allocated 39812*

*Buffer pool size 38400*

*Free buffers 37304*

*Database pages 1095*

*Old database pages 424*

*Modified db pages 0*

*## 确保脏页数量为0*

## ROW OPERATIONS

-----  
*0 queries inside InnoDB , 0 queries in queue*

*1 read views open inside InnoDB*

*Main thread process no. 30426 , id 140111500936976 , state: waiting for server activity Number of rows inserted 0 , updated 0 , deleted 0 , read 0*

*0.00 inserts/s , 0.00 updates/s , 0.00 deletes/s , 0.00 reads/s*



## 确保插入、更新、删除为0

上面一系列确认工作完成之后，就可以进行恢复操作了。还记得刚才我们记录的删除文件吗？如下所示：

```
# ll /proc/30426/fd | egrep 'ib_|ibdata'  
lrwx----- 1 root root 64 9月24 16:51 10 ->/u2/mysql/data/ib_logfile1  
lrwx----- 1 root root 64 9月24 16:51 11 ->/u2/mysql/data/ib_logfile2  
lrwx----- 1 root root 64 9月24 16:51 4 ->/u2/mysql/data/ibdata1  
lrwx----- 1 root root 64 9月24 16:51 9 ->/u2/mysql/data/ib_logfile0
```

把这些文件复制到原来的目录下：

```
#cd /proc/10755/fd  
#cp 10 /u2/mysql/data/ib_logfile1  
#cp 11 /u2/mysql/data/ib_logfile2  
#cp 4 /u2/mysql/data/ibdata1  
#cp 9 /u2/mysql/data/ib_logfile0
```

然后修改用户属性：

```
#cd /u2/mysql/data/  
#chown mysql:mysql ib*
```

现在，只需要重启MySQL即可。重启命令如下：

```
/etc/init.d/mysql restart
```

友情提醒：千万不要在生产环境下测试。



### 3.3.8 update忘加where条件误操作恢复（模拟Oracle闪回功能）

我相信很多人都遇到过忘带where条件，结果执行了update后把整张表的数据都给改了的情况。传统的解决方法是：利用最近的全量备份+增量binlog备份，恢复到误操作之前的状态，但是此方法有一个弊端，那就是随着表的记录增大，binlog的增多，恢复起来会很费时费力。

现在有一个简单的方法，可以恢复到误操作之前的状态。听起来这方法似乎利用的是Oracle的闪回功能，但MySQL目前（包括最新的V5.6版本）还不具有这样的功能，不过，我们完全可以模拟出这一功能来。使用该方法前，记得要将binlog日志设置为binlog\_format=ROW，如果是STATEMENT，这个方法无效的。切记！

下面来演示一下：

现在有一张学生表，要把小于60分的成绩更新成不及格。命令如下：

```
mysql> select * from student;
+----+-----+-----+-----+
| id | name | class | score |
+----+-----+-----+-----+
| 1 | a | 1 | 56 |
| 2 | b | 1 | 61 |
| 3 | c | 2 | 78 |
| 4 | d | 2 | 45 |
| 5 | e | 3 | 76 |
| 6 | f | 3 | 89 |
| 7 | g | 4 | 43 |
| 8 | h | 4 | 90 |
+----+-----+-----+-----+
8 rows in set (0.02 sec)
```

结果，在执行操作时忘带where条件了，如下所示：

```
mysql> update student set score='failure';
Query OK, 8 rows affected (0.11 sec)
Rows matched: 8 Changed: 8 Warnings: 0
```

```
mysql> select * from student;
```

```
+-----+-----+-----+-----+
| id | name | class | score |
+-----+-----+-----+-----+
| 1 | a | 1 | failure |
| 2 | b | 1 | failure |
| 3 | c | 2 | failure |
| 4 | d | 2 | failure |
| 5 | e | 3 | failure |
| 6 | f | 3 | failure |
| 7 | g | 4 | failure |
| 8 | h | 4 | failure |
+-----+-----+-----+-----+
```

```
8 rows in set (0.01 sec)
```

这致使整张表的记录都更新成不及格了。

下面开始恢复：

首先，创建一个普通权限的账号（切记不能是SUPER权限），例如：

```
GRANT ALL PRIVILEGES ON yourDB.* TO 'admin_read_only'@'%' IDENTIFIED BY '123456';
flush privileges;
```

把read\_only打开，设置数据库只读，命令如下：

```
mysql> set global read_only = 1;
Query OK, 0 rows affected (0.01 sec)
```

把刚才创建的admin\_read\_only账号给运维人员，让运维人员把前端程序（PHP/JSP/.NET等）的用户名改一下，然后重启前端程序（PHP/JSP/.NET等），这样再连接进来的用户对数据库进行访问时就只能读不能写了，这是为了保证恢复的一致性。

下面，通过binlog先找到那条语句：

```
[root@M1 data]# /usr/local/mysql/bin/mysqlbinlog
--no-defaults -v -v --base64-output=DECODE-ROWS
mysql-bin.000001 | grep -B 15 'failure' | more
/*!*/;
```



# at 192

#121124 23:55:15 server id25 end\_log\_pos 249 CRC32 0x83a12fbc Table\_map: 'test'. 'student' mapped to number 76

# at 249

#121124 23:55:15 server id 25 end\_log\_pos 549 CRC32 0xcf7d2635 Update\_rows: table id 76 flags: STMT\_END\_F

### UPDATE test.student

### WHERE

### @11=1 /\* INT meta=0 nullable=0 is\_null=0 \*/

### @2='a' /\* VARSTRING(18) meta=18 nullable=1 is\_null=0 \*/

### @3=1 /\* INT meta=0 nullable=1 is\_null=0 \*/

### @4='56' /\* VARSTRING(30) meta=30 nullable=1 is\_null=0 \*/

### SET

### @11=1 /\* INT meta=0 nullable=0 is\_null=0 \*/

### @2='a' /\* VARSTRING(18) meta=18 nullable=1 is\_null=0 \*/

### @3=1 /\* INT meta=0 nullable=1 is\_null=0 \*/

### @4='failure' /\* VARSTRING(30) meta=30 nullable=1 is\_null=0 \*/

### UPDATE test.student

### WHERE

### @1=2 /\* INT meta=0 nullable=0 is\_null=0 \*/

### @2='b' /\* VARSTRING(18) meta=18 nullable=1 is\_null=0 \*/

### @3=1 /\* INT meta=0 nullable=1 is\_null=0 \*/

### @4='61' /\* VARSTRING(30) meta=30 nullable=1 is\_null=0 \*/

### SET

### @1=2 /\* INT meta=0 nullable=0 is\_null=0 \*/

### @2='b' /\* VARSTRING(18) meta=18 nullable=1 is\_null=0 \*/

### @3=1 /\* INT meta=0 nullable=1 is\_null=0 \*/

### @4='failure' /\* VARSTRING(30) meta=30 nullable=1 is\_null=0 \*/

--More--

然后把那条binlog给导出来：

[root@M1 data]# /usr/local/mysql/bin/mysqlbinlog

--no-defaults -v -v --base64-output=DECODE-ROWS





```
mysql-bin.000001 | sed -n '/# at 249/ , /COMMIT/p' >/opt/1.txt
```

```
[root@M1 data]#
```

```
[root@M1 data]# more /opt/1.txt
```

```
# at 249
```

```
#121124 23:55:15 server id 25 end_log_pos 549 CRC32 0xcf7d2635 Update_rows: table id 76 flags: STMT_END_F
```

```
### UPDATE test.student
```

```
### WHERE
```

```
### @11=1 /* INT meta=0 nullable=0 is_null=0 */
```

```
### @2='a' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
```

```
### @3=1 /* INT meta=0 nullable=1 is_null=0 */
```

```
### @4='56' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
```

```
### SET
```

```
### @11=1 /* INT meta=0 nullable=0 is_null=0 */
```

```
### @2='a' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
```

```
### @3=1 /* INT meta=0 nullable=1 is_null=0 */
```

```
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
```

```
### UPDATE test.student
```

```
### WHERE
```

```
### @1=2 /* INT meta=0 nullable=0 is_null=0 */
```

```
### @2='b' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
```

```
### @3=1 /* INT meta=0 nullable=1 is_null=0 */
```

```
### @4='61' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
```

```
### SET
```

```
### @1=2 /* INT meta=0 nullable=0 is_null=0 */
```

```
### @2='b' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
```

```
### @3=1 /* INT meta=0 nullable=1 is_null=0 */
```

```
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
```

```
### UPDATE test.student
```

```
### WHERE
```

```
### @1=3 /* INT meta=0 nullable=0 is_null=0 */
```



```
### @2='c' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=2 /* INT meta=0 nullable=1 is_null=0 */
### @4='78' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=3 /* INT meta=0 nullable=0 is_null=0 */
### @2='c' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=2 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### UPDATE test.student
### WHERE
### @1=4 /* INT meta=0 nullable=0 is_null=0 */
### @2='d' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=2 /* INT meta=0 nullable=1 is_null=0 */
### @4='45' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=4 /* INT meta=0 nullable=0 is_null=0 */
### @2='d' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=2 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### UPDATE test.student
### WHERE
### @1=5 /* INT meta=0 nullable=0 is_null=0 */
### @2='e' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=3 /* INT meta=0 nullable=1 is_null=0 */
### @4='76' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=5 /* INT meta=0 nullable=0 is_null=0 */
### @2='e' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=3 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
```





```

### UPDATE test.student
### WHERE
### @1=6 /* INT meta=0 nullable=0 is_null=0 */
### @2='f' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @33=3 /* INT meta=0 nullable=1 is_null=0 */
### @4='89' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=6 /* INT meta=0 nullable=0 is_null=0 */
### @2='f' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @33=3 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### UPDATE test.student
### WHERE
### @1=7 /* INT meta=0 nullable=0 is_null=0 */
### @2='g' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='43' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=7 /* INT meta=0 nullable=0 is_null=0 */
### @2='g' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### UPDATE test.student
### WHERE
### @1=8 /* INT meta=0 nullable=0 is_null=0 */
### @2='h' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='90' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
### @1=8 /* INT meta=0 nullable=0 is_null=0 */

```



```

### @2='h' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 549
#121124 23:55:15 server id 25 end_log_pos 580 CRC32 0x378c91b0 Xid = 531
COMMIT/*!*/;
[root@M1 data]#

```

其中，这些是误操作之前的数据：

```

### @1=8 /* INT meta=0 nullable=0 is_null=0 */
### @2='h' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='90' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */

```

这些是误操作之后的数据：

```

### @1=8 /* INT meta=0 nullable=0 is_null=0 */
### @2='h' /* VARSTRING(18) meta=18 nullable=1 is_null=0 */
### @3=4 /* INT meta=0 nullable=1 is_null=0 */
### @4='failure' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */

```

这里，@1、@2、@3、@4对应的表字段分别是id、name、class、score。

现在，就要进行最后一步的恢复操作了，把这些binlog转换成SQL语句，然后将其导入进去。

```

[root@M1 opt]# sed '/WHERE/{:a;N;/SET/!ba;s/\([^\n]*\|n\|(.*)\|n\|(.*)\|3|n|2|n|1/}' 1.txt
| sed -r '/WHERE/{:a;N;/@4/!ba;s/### @2.*//g}'
| sed 's/### //g;s/\^.*//g'
| sed '/WHERE/{:a;N;/@1/!ba;s/, /;/g};s/#.*//g;s/COMMIT, //g'
| sed '/^$/d' >./recover.sql
[root@M1 opt]#
[root@M1 opt]# cat recover.sql
UPDATE test.student
SET
  @1=1 ,

```

```

    @2='a' ,
    @3=1 ,
    @4='56' ,
WHERE
    @11=1;
UPDATE test.student
SET
    @1=2 ,
    @2='b' ,
    @3=1 ,
    @4='61' ,
WHERE
    @1=2;
UPDATE test.student
SET
    @1=3 ,
    @2='c' ,
    @3=2 ,
    @4='78' ,
WHERE
    @1=3;
UPDATE test.student
SET
    @1=4 ,
    @2='d' ,
    @3=2 ,
    @4='45' ,
WHERE
    @1=4;
UPDATE test.student

```

SET

@1=5 ,  
@2='e' ,  
@33=3 ,  
@4='76' ,

WHERE

@1=5 ;

UPDATE test.student

SET

@1=6 ,  
@2='f' ,  
@33=3 ,  
@4='89' ,

WHERE

@1=6 ;

UPDATE test.student

SET

@1=7 ,  
@2='g' ,  
@3=4 ,  
@4='43' ,

WHERE

@1=7 ;

UPDATE test.student

SET

@1=8 ,  
@2='h' ,  
@3=4 ,  
@4='90' ,

WHERE



@1=8;

[root@M1 opt]#

再把@1、@2、@3、@4对应的表字段 ( id、name、class、score ) 替换掉，如下所示：

[root@M1 opt]# sed -i 's/@1/id/g;s/@2/name/g;s/@3/class/g;s/@4/score/g' recover.sql

[root@M1 opt]# sed -i -r 's/(score=.\*), \^1/g' recover.sql

[root@M1 opt]#

[root@M1 opt]# cat recover.sql

UPDATE test.student

SET

id=1 ,

name='a' ,

class=1 ,

score='56'

WHERE

id=1 ;

UPDATE test.student

SET

id=2 ,

name='b' ,

class=1 ,

score='61'

WHERE

id=2 ;

UPDATE test.student

SET

id=3 ,

name='c' ,

class=2 ,

score='78'

WHERE

*id=3;*  
*UPDATE test.student*  
*SET*  
*id=4 ,*  
*name='d' ,*  
*class=2 ,*  
*score='45'*  
*WHERE*  
*id=4;*  
*UPDATE test.student*  
*SET*  
*id=5 ,*  
*name='e' ,*  
*class=3 ,*  
*score='76'*  
*WHERE*  
*id=5;*  
*UPDATE test.student*  
*SET*  
*id=6 ,*  
*name='f' ,*  
*class=3 ,*  
*score='89'*  
*WHERE*  
*id=6;*  
*UPDATE test.student*  
*SET*  
*id=7 ,*  
*name='g' ,*  
*class=4 ,*



```
score='43'  
WHERE  
id=7;  
UPDATE test.student  
SET  
id=8 ,  
name='h' ,  
class=4 ,  
score='90'  
WHERE  
id=8;  
[root@M1 opt]#
```

下面进行恢复操作：

```
mysql>select * from student;  
+----+-----+-----+-----+  
/ id / name / class / score /  
+----+-----+-----+-----+  
/ 1 / a / 1 / failure /  
/ 2 / b / 1 / failure /  
/ 3 / c / 2 / failure /  
/ 4 / d / 2 / failure /  
/ 5 / e / 3 / failure /  
/ 6 / f / 3 / failure /  
/ 7 / g / 4 / failure /  
/ 8 / h / 4 / failure /  
+----+-----+-----+-----+
```

8 rows in set (0.02 sec)

```
mysql>source /opt/recover.sql  
Query OK , 1 row affected (0.11 sec)  
Rows matched: 1 Changed: 1 Warnings: 0
```



```
Query OK , 1 row affected (0.95 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.16 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.03 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.80 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.08 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.09 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK , 1 row affected (0.07 sec)
Rows matched: 1 Changed: 1 Warnings: 0
mysql>select * from student;
+----+-----+-----+-----+
| id | name | class | score |
+----+-----+-----+-----+
| 1 | a | 1 | 56 |
| 2 | b | 1 | 61 |
| 3 | c | 2 | 78 |
| 4 | d | 2 | 45 |
| 5 | e | 3 | 76 |
| 6 | f | 3 | 89 |
| 7 | g | 4 | 43 |
| 8 | h | 4 | 90 |
+----+-----+-----+-----+
8 rows in set (0.02 sec)
mysql>
```

友情提醒：千万不要在生产环境下测试。

### 3.3.9 delete忘加where条件误操作恢复（模拟Oracle闪回功能）

如果在delete时忘加where条件了怎么办？可按照上面的方法依葫芦画瓢，下面来演示一下delete操作时忘加where条件的误操作恢复。

表数据为：

```
mysql>select * from qq1;
```

```
+----+-----+-----+-----+
| id | class_id | fist_name | last_name |
+----+-----+-----+-----+
| 1 | 101 | lo | 1gang |
| 2 | 101 | zhou | 2gang |
| 3 | 102 | wong | 3gang |
| 4 | 101 | huo | 4gang |
| 5 | 102 | son | 5gang |
| 6 | 102 | dong | 6gang |
| 7 | 103 | qo | 7gang |
| 8 | 103 | dao | 8gang |
+----+-----+-----+-----+
```

8 rows in set (0.03 sec)

现在要删除id大于5的记录：

```
mysql>delete from qq1 where id >5;
```

Query OK, 3 rows affected (0.00 sec)

```
mysql>select * from qq1;
```

```
+----+-----+-----+-----+
| id | class_id | fist_name | last_name |
+----+-----+-----+-----+
| 1 | 101 | lo | 1gang |
| 2 | 101 | zhou | 2gang |
```



```

/ 3 / 102 / wong / 3gang /
/ 4 / 101 / huo / 4gang /
/ 5 / 102 / son / 5gang /
+---+-----+-----+-----+
5 rows in set (0.00 sec)

```

现在要分析binlog，以便把误操作delete的语句保存到文本里：

```

[root@hadoop-datanode5 data]# mysqlbinlog
--no-defaults --base64-output=decode-rows -v -v mysql-bin.000001
| sed -n '/### DELETE FROM test.qq1/, /COMMIT/p' >/root/delete.txt
[root@hadoop-datanode5 ~]# cat /root/delete.txt
### DELETE FROM test.qq1
### WHERE
### @1=6 /* INT meta=0 nullable=0 is_null=0 */
### @2=102 /* INT meta=0 nullable=1 is_null=0 */
### @3='dong' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### @4='6gang' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### DELETE FROM test.qq1
### WHERE
### @1=7 /* INT meta=0 nullable=0 is_null=0 */
### @2=103 /* INT meta=0 nullable=1 is_null=0 */
### @3='qo' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### @4='7gang' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### DELETE FROM test.qq1
### WHERE
### @1=8 /* INT meta=0 nullable=0 is_null=0 */
### @2=103 /* INT meta=0 nullable=1 is_null=0 */
### @3='dao' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### @4='8gang' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 310
#130526 9:00:02 server id 140 end_log_pos 337 Xid = 88

```



```

COMMIT/*!*/;
### DELETE FROM test.qq1
### WHERE
### @1=6 /* INT meta=0 nullable=0 is_null=0 */
### @2=103 /* INT meta=0 nullable=1 is_null=0 */
### @3='aa' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### @4='aa1' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 688
#130526 9:21:42 server id 140 end_log_pos 715 Xid = 92
COMMIT/*!*/;

```

接下来要把它转换为标准的SQL语句：

```

[root@hadoop-datanode5~]#cat delete.txt | sed -n '/###/p' | sed 's/### //g;s/\^.*$/,/g;s/DELETE FROM/INSERT INTO/g;s/
WHERE/SELECT/g;' | sed -r 's/(@4.*) , \^1;/g' | sed 's/@[1-9]=//g' >insert.sql

```

```

[root@hadoop-datanode5 ~]#

```

```

[root@hadoop-datanode5 ~]# cat insert.sql

```

```

INSERT INTO test.qq1

```

```

SELECT

```

```

6 ,
102 ,
'dong' ,
'6gang';

```

```

INSERT INTO test.qq1

```

```

SELECT

```

```

7 ,
103 ,
'qo' ,
'7gang';

```

```

INSERT INTO test.qq1

```

```

SELECT

```

```

8 ,

```

```
103 ,
'dao' ,
'8gang';
INSERT INTO test.qq1
SELECT
```

```
6 ,
103 ,
'aa' ,
'aa1';
```

[root@hadoop-datanode5 ~]#

之后将其导入数据库里即可，如下所示：

```
mysql>select * from qq1;
+----+-----+-----+-----+
| id | class_id | fist_name | last_name |
+----+-----+-----+-----+
| 1 | 101 | lo | 1gang |
| 2 | 101 | zhou | 2gang |
| 3 | 102 | wong | 3gang |
| 4 | 101 | huo | 4gang |
| 5 | 102 | son | 5gang |
+----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>source /root/insert.sql
Query OK, 1 row affected (0.01 sec)
Records: 1 Duplicates: 0 Warnings: 0
Query OK, 1 row affected (0.00 sec)
Records: 1 Duplicates: 0 Warnings: 0
Query OK, 1 row affected (0.00 sec)
Records: 1 Duplicates: 0 Warnings: 0
mysql>select * from qq1;
```



```
+----+-----+-----+-----+
/ id / class_id / fist_name / last_name /
+----+-----+-----+-----+
/ 1 / 101 / lo / 1gang /
/ 2 / 101 / zhou / 2gang /
/ 3 / 102 / wong / 3gang /
/ 4 / 101 / huo / 4gang /
/ 5 / 102 / son / 5gang /
/ 6 / 102 / dong / 6gang /
/ 7 / 103 / qo / 7gang /
/ 8 / 103 / dao / 8gang /
+----+-----+-----+-----+
8 rows in set (0.00 sec)
```

友情提醒：千万不要在生产环境下测试。





## 第4章

# 同步复制报错故障处理

在发生故障进行切换后，经常遇到的问题就是同步报错，我见过很多DBA都是执行命令“set global sql\_slave\_skip\_counter=1;”跳过并忽略错误，这种方法虽然可以达到让同步复制继续运行的目的，但其实就是掩耳盗铃，因为此时slave数据已经不一致了。

另一种方法就是在主库上MySQLdump导出数据，然后在slave上恢复，当数据很小（比如几个GB）的时候，这样做可以，没有任何问题。但如果你的公司数据量很庞大，大到150~200 GB，此时单纯地用导出和导入方法，太耗费时间，不可取。经过一段时间的摸索，我总结了几种处理方法。本章将针对这些方法进行讲解。

## 4.1 最常见的3种故障

在说明最常见的3种故障之前，先来看一下异步复制和半同步复制的区别：

·异步复制：简单地说就是master把binlog发送过去，不管slave是否接收完，也不管是否执行完，这一动作就结束了。

·半同步复制：简单地说就是master把binlog发送过去，slave确认接收完，但不管它是否执行完，给master一个信号我这边收到了，这一动作就结束了。

（半同步复制patch谷歌写的代码，MySQL5.5上正式应用。）

异步的劣势是：当master上写操作繁忙时，当前POS点，例如，是10，而slave上IO\_THREAD线程接收过来的是3，此时master宕机，会造成相差7个点未传送到slave上而数据丢失。

接下来要介绍的这3种故障是在HA集群切换时产生的，由于是异步复制，且sync\_binlog=0，会造成一小部分binlog没接收完，从而导致同步报错。



### 4.1.1 在master上删除一条记录时出现的故障

笔者曾碰到过这种问题，在master上删除一条记录后，slave上因找不到该记录而报错，报错信息如下：

```
Last_SQL_Error: Could not execute Delete_rows event on table hcy.t1; Can't find record in 't1', Error_code: 1032; handler error
HA_ERR_KEY_NOT_FOUND; the event's master log MySQL-bin.000006, end_log_pos 254
```

出现这种情况是因为主机上已将其删除了，对此，可采取从机直接跳过的方式解决，命令如下：

```
stop slave ;set global sql_slave_skip_counter=1;start slave;
```

对于这种情况，我写了一个脚本skip\_error\_replication.sh来帮助处理，该脚本默认跳过10个错误（只会针对这种情况跳，其他情况还是会输出错误结果，等待处理），这个脚本是参考maakit工具包的mk-slave-restart原理用shell写的，由于mk-slave-restart脚本是不管什么错误一律跳过，这样会造成主从数据不一致，因此我在该脚本的功能方面定义了一些自己的东西，使其不是无论什么错误都一律跳过），脚本如下：

```
#!/bin/bash
#####
##
## 此脚本是用来自动处理同步报错的，默认跳过10次。
##
## 只有Last_SQL_Error: Could not execute Delete_rows event on table hcy.t1; Can't find record ## in 't1', Error_code: 1032;
handler error HA_ERR_KEY_NOT_FOUND; the event's master log ## MySQL-bin.000003, end_log_pos 253
## 这种情况才跳过，其他情况，需要自行处理，以免丢失数据。
##
## by hechunyang
#####
export LANG=zh_CN
. /root/.bash_profile
v_dir=/usr/local/MySQL/bin/
v_user=root
v_passwd=123456
v_log=/home/logs
```



```
v_times=10
if [ -d "${v_log}" ];then
    echo "${v_log} has existed before."
else
    mkdir ${v_log}
fi
echo "" > ${v_log}/slave_status.log
echo "" > ${v_log}/slave_status_error.log
count=1
while true
do
    Seconds_Behind_master=$((${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G;" | awk -F':' '/Seconds_Behind_master/{print $2}'))
    if [ ${Seconds_Behind_master} != "NULL" ];then
        echo "slave is ok!"
        ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G;" >> ${v_log}/slave_status.log
        break
    else
        echo "" >> ${v_log}/slave_status_error.log
        date >> ${v_log}/slave_status_error.log
        echo "" >> ${v_log}/slave_status_error.log
        ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G" >> ${v_log}/slave_status_error.log
        ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G" | egrep 'Delete_rows' >/dev/null 2>&1
        if [ $ = 0 ];then
            ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "stop slave;SET GLOBAL sql_slave_skip_counter=1;start slave;"
        else
            ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G" | grep 'Last_SQL_Error'
            break
        fi
    let count++
done
```



```
if [ $count -gt ${v_times} ];then
```

```
    break
```

```
else
```

```
    ${v_dir}MySQL -u${v_user} -p${v_passwd} -e "show slave status\G" >>${v_log}/slave_status_error.log
```

```
    sleep 2
```

```
    continue
```

```
fi
```

```
fi
```

```
done
```



### 4.1.2 主键重复

主从数据不一致时，slave上已经有该条记录，但我们又在master上插入了同一条记录，此时就会报错，报错信息如下：

```
Last_SQL_Error: Could not execute Write_rows event on table hcy.t1;
```

```
Duplicate entry '2' for key 'PRIMARY',
```

```
Error_code: 1062; handler error HA_ERR_FOUND_DUPP_KEY;
```

```
the event's master log MySQL-bin.000006, end_log_pos 924
```

解决方法：在slave上使用命令“deschcy.t1;”先查看一下表结构，如下所示：

```
MySQL>desc hcy.t1;
```

```
+-----+-----+-----+-----+-----+-----+
```

```
/ Field / Type / Null / Key / Default / Extra /
```

```
+-----+-----+-----+-----+-----+-----+
```

```
/ id / int(11) / NO / PRI / 0 /
```

```
/ name / char(4) / YES / / NULL /
```

```
+-----+-----+-----+-----+-----+-----+
```

```
2 rows in set (0.03 sec)
```

查看该表字段信息，得到主键的字段名。

接着删除重复的主键，命令如下：

```
MySQL>delete from t1 where id=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

然后开启同步复制功能，命令如下：

```
MySQL>start slave;
```

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

```
MySQL>show slave status\G;
```

```
...
```

```
slave_IO_Running: Yes
```

```
slave_SQL_Running: Yes
```

...

```
MySQL>select * from t1 where id=2;
```

完成上述操作后，还要在master上和slave上再分别确认一下，确保执行成功。

### 4.1.3 在master上更新一条记录，而slave上却找不到

主从数据不一致时，master上已经有该条记录，但slave上没有这条记录，之后若在master上又更新了这条记录，此时就会报错，报错信息如下：

```
Last_SQL_Error: Could not execute Update_rows event on table hcy.t1; Can't find
record in 't1', Error_code: 1032; handler error HA_ERR_KEY_NOT_FOUND; the
event's master log MySQL-bin.000010, end_log_pos 794
```

解决方法：在master上，用MySQLbinlog分析一下出错的binlog日志在干什么，如下所示：

```
[root@vm01 data]# /usr/local/MySQL/bin/MySQLbinlog
--no-defaults -v -v --base64-output=DECODE-ROWS
MySQL-bin.000010 | grep -A '10' 794
#120302 12:08:36 server id 22 end_log_pos 794 Update_rows: table id 33 flags: STMT_END_F### UPDATE hcy.t1
### WHERE### @1=2/* INT meta=0 nullable=0 is_null=0 */### @2='bbc'/* STRING(4) meta=65028 nullable=1 is_null=0 */
### SET### @1=2/* INT meta=0 nullable=0 is_null=0 */### @2='BTV'/* STRING(4) meta=65028 nullable=1 is_null=0 */
# at 794
#120302 12:08:36 server id 22 end_log_pos 821 Xid = 60
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by MySQLbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
[root@vm01 data]#
```

从上面的信息来看，是在更新一条记录。

接着，在slave上查找一下更新后的那条记录，应该是不存在的。命令如下：

```
MySQL>select * from t1 where id=2;
Empty set (0.00 sec)
```

然后再到master查看，命令如下：



```
MySQL>select * from t1 where id=2;
```

```
+----+-----+
| id | name |
+----+-----+
| 2 | BTV |
+----+-----+
```

1 row in set (0.00 sec)

可以看到，这里已经找到了这条记录。  
最后把丢失的数据填补到在slave上，命令如下：

```
MySQL>insert into t1 values (2, 'BTV');
```

Query OK, 1 row affected (0.00 sec)

```
MySQL>select * from t1 where id=2;
```

```
+----+-----+
| id | name |
+----+-----+
| 2 | BTV |
+----+-----+
```

1 row in set (0.00 sec)

完成上述操作后，跳过报错即可，命令如下：

```
MySQL>stop slave ;set global sql_slave_skip_counter=1;start slave;
```

Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

```
MySQL>show slave status\G;
```

```
...
slave_IO_Running: Yes
slave_SQL_Running: Yes
...
```

## 4.2 特殊情况：slave的中继日志relay-log损坏

当slave意外宕机时，有可能会损坏中继日志relay-log，再次开启同步复制时，报错信息如下：

*Last\_SQL\_Error: Error initializing relay log position: I/O error reading the header from the binary log*

*Last\_SQL\_Error: Error initializing relay log position: Binlog has bad magic number; It's not a binary log file that can be used by this version of MySQL*

解决方法：找到同步的binlog日志和POS点，然后重新进行同步，这样就可以有新的中继日志了。

下面来看个例子，这里模拟了中继日志损坏的情况，查看到的信息如下：

*MySQL>show slave status\G;*

*\*\*\*\*\* 1. row \*\*\*\*\**

*master\_Log\_File: MySQL-bin.000010*

*Read\_master\_Log\_Pos: 1191*

*Relay\_Log\_File: vm02-relay-bin.000005*

*Relay\_Log\_Pos: 253*

*Relay\_master\_Log\_File: MySQL-bin.000010*

*slave\_IO\_Running: Yes*

*slave\_SQL\_Running: No*

*Replicate\_Do\_DB:*

*Replicate\_Ignore\_DB:*

*Replicate\_Do\_Table:*

*Replicate\_Ignore\_Table:*

*Replicate\_Wild\_Do\_Table:*

*Replicate\_Wild\_Ignore\_Table:*

*Last\_Errno: 1593*

*Last\_Error: Error initializing relay log position: I/O error reading the header from the binary log*

*Skip\_Counter: 1*

*Exec\_master\_Log\_Pos: 821*





其中，涉及几个重要参数：

- slave\_IO\_Running*：接收master的binlog信息
- master\_Log\_File*：正在读取master上binlog日志名。
- Read\_master\_Log\_Pos*：正在读取master上当前binlog日志POS点。
  - slave\_SQL\_Running*：执行写操作。
- Relay\_master\_Log\_File*：正在同步master上的binlog日志名。
- Exec\_master\_Log\_Pos*：正在同步当前binlog日志的POS点。

以*Relay\_master\_Log\_File*参数值和*Exec\_master\_Log\_Pos*参数值为基准。

*Relay\_master\_Log\_File*:MySQL-bin.000010

*Exec\_master\_Log\_Pos*:821

接下来可以重置主从复制了，操作如下：

```
MySQL>stop slave;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
MySQL>CHANGE master TO master_LOG_FILE='MySQL-bin.000010',  
master_LOG_POS=821;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
MySQL>start slave;
```

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

重新建立完主从复制以后，就可以查看一下状态信息了，如下所示：

```
MySQL>show slave status\G;
```

```
***** 1. row *****
```

```
slave_IO_State: Waiting for master to send event
```

```
master_Host: 192.168.8.22
```

```
master_User: repl
```

```
master_Port: 3306
```

```
Connect_Retry: 10
```

```
master_Log_File: MySQL-bin.000010
```

```
Read_master_Log_Pos: 1191
```

```
Relay_Log_File: vm02-relay-bin.000002
```



```

Relay_Log_Pos: 623
Relay_master_Log_File: MySQL-bin.000010
slave_IO_Running: Yes
slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 0
Last_Error:
Skip_Counter: 0
Exec_master_Log_Pos: 1191
Relay_Log_Space: 778
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
master_SSL_Allowed: No
master_SSL_CA_File:
master_SSL_CA_Path:
master_SSL_Cert:
master_SSL_Cipher:
master_SSL_Key:
Seconds_Behind_master: 0
master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 0
Last_SQL_Error:

```

通过这种方法我们已经修复了中继日志。是不是有些麻烦？其实如果你有仔细看完第1章的新特性，会发现MySQL5.5已经考虑到slave宕机中继日志损坏这一问题了，即在slave的配置文件my.cnf里要增加一个参数relay\_log\_recovery=1就可以了，前面已经介绍了，这里不再阐述。

### 4.3 人为失误

这种情况在生产环境中并不常见，下面记录了一次server-id重复导致的主从同步复制报错。

多台slave存在server-id重复

这个错误一般是初级DBA经常遇到的，他们在进行主从配置时，直接把master的my、cnf复制到slave上，却忘记了修改slave上的server-id，报错信息如下：

*slave: received end packet from server , apparent master shutdown:*

*slave I/O thread: Failed reading log event ,*

*reconnecting to retry , log 'MySQL-bin.000012' at position 106*

在这种情况下，同步会一直延时，永远也同步不完，error错误日志里会一直出现上面两行信息。解决方法就是把slave机器上的server-id改成不一致，然后重启MySQL即可。



## 4.4 避免在master上执行大事务

这是一个真实的案例，一张大表大约有70 GB，因为业务需要，要删除一些数据，由于删除的ID关联的数据太多，该删除操作变成了一个大事，一下子就把slave给卡死了，当时的现象是执行“show slave status\G;”时，Exec\_master\_Log\_Pos值一直不发生变化，但Seconds\_Behind\_master的值却越来越大，致使同步落后越来越多。

当时的SQL语句是：

```
delete from bigtable where UserId = v_userid;
```

表数据如图4-1所示。

OwnerId	ContactId	CommonCount
200034797	200034798	161
200034797	200034799	161
200034797	200034800	161
200034797	200034801	161
200034797	200034802	161
200034797	200034803	161
200034797	200034804	161
200034797	200034805	161
200034797	200034806	161
200034797	200034807	161
200034797	200034808	161
200034797	200034809	158
200034797	200034810	158

图4-1 数据

碰到这个问题后，我想出的解决办法是改为用存储过程，每删除1000条事务就提交一次，循环操作直至删除完毕。经过优化，行锁的范围变小了，性能也就变好了。相关代码如下：

```
DELIMITER $$
USE BIGDB$$
DROP PROCEDURE IF EXISTS BIG_table_delete_1k$$
CREATE PROCEDURE BIG_table_delete_1k(IN v_UserId INT)
BEGIN
del_1k:LOOP
delete from BIGDB.BIGTABLE where UserId = v_UserId limit 1000;
select row_count() into @count;
```

```
IF @count = 0 THEN
    select CONCAT('BIGDB.BIGTABLE UserId = ', v_UserId, ' is ', @count, ' rows.') as BIGTABLE_delete_finish;
LEAVE del_1k;
END IF;
    select sleep(1);
END LOOP del_1k;
END$$
DELIMITER ;
```

存储过程上线后，观察一段时间，同步复制时，slave复制正常，问题解决。

## 4.5 slave\_exec\_mode参数可自动处理同步复制错误

大家有没有遇到这种情况，假日里，你正在外面享受生活，突然手机收到一条短信——主从同步报错，会让你原本很好的心情一下子沉入谷底。因为如果你处理不及时，同步复制会因报错而中断，这样就会对业务产生影响（如果你没有做特殊的设置，MySQL是不会自动跳过这个错误的），尤其是开启了读写分离功能后——用户发了一个帖子，半天没找到自己刚才发的帖子，投诉的电话就得在老板的耳旁响起了。

那么该如何解决呢？`slave_exec_mode`这个参数可以帮助你。

你可以动态设置，如下所示：

```
set global slave_exec_mode='IDEMPOTENT';
```

其默认值是STRICT（严格模式），如图4-2所示。

```
mysql> show variables like 'slave_exec_mode';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| slave_exec_mode | STRICT |
+-----+-----+
1 row in set (0.15 sec)

mysql> set global slave_exec_mode='IDEMPOTENT';
Query OK, 0 rows affected (0.03 sec)

mysql> show variables like 'slave_exec_mode';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| slave_exec_mode | IDEMPOTENT |
+-----+-----+
1 row in set (0.04 sec)
```

图4-2 设置global slave\_exec\_mode为IDEMPOTENT



注意 设置完毕后，并不能立即生效，需要重启下复制进程，第一步：关闭同步复制；第二步：开启同步复制，这样才可以。





设置完毕后，当出现的1023错误（记录没找到）和1062错误（主键重复）时，就会自动跳过该错误，并且记录到错误日志里。其实，`slave_exec_mode`参数和`slave_skip_errors`参数的作用是一样的，只不过`slave_skip_errors`参数必须加到配置文件`my.cnf`里然后重启MySQL，而`slave_exec_mode`参数可以动态设置。

另外，你还可以在Nagios主机上做个监控，当出现同步报错时，执行相应的命令：

```
MySQL -uroot -p123456 -h(yourIP) -e "set global slave_exec_mode='IDEMPOTENT';"
```

下面是我实现的一个脚本`skip_slave_error.sh`：

```
#!/bin/bash
user=admin
passwd=123456
port=3306
MySQLpath=/usr/local/MySQL/bin
for hostip in `cat slaveip.txt`
do
    result=$(MySQLpath/MySQL -u$user -p$passwd -h$hostip -P$port \
    -e "show slave status\G" | awk -F": " '{print $2}')
    if [ "$result" != "Yes" ];then
        MySQLpath/MySQL -u$user -p$passwd -h$hostip -P$port \
        -e "set global slave_exec_mode='IDEMPOTENT';"
        MySQLpath/MySQL -u$user -p$passwd -h$hostip -P$port \
        -e "stop slave;"
        MySQLpath/MySQL -u$user -p$passwd -h$hostip -P$port \
        -e "start slave;"
        echo "replication is error and skip" | mail -s "replcation Alert" hechunyang@139.com \
        -- -f nagiosadmin@139.com -F nagiosadmin
        #发送一封邮件并短信通知你同步报错已经跳过了。
    fi
done
```

`slaveip.txt`文件填写你的slave机器的IP，如下面这样一行一行添加，上面的脚本会调用`slaveip.txt`：

```
# cat slaveip.txt
```

192.168.8.22

192.168.8.23

192.168.8.24

192.168.8.25

192.168.8.26

然后将脚本skip\_slave\_error.sh加入到crontab里，每十分钟检查一次。

```
*/10 * * * * bash /home/hechunyang/skip_slave_error.sh
```

这样可以尽可能地减少业务影响的范围扩大，等你到家时再做下一步处理。

参考手册：

- **Binary log execution errors and `slave_exec_mode`.** If `slave_exec_mode` is `IDEMPOTENT`, a failure to apply changes from RBL because the original row cannot be found does not trigger an error or cause replication to fail. This means that it is possible that updates are not applied on the slave, so that the master and slave are no longer synchronized. Latency issues and use of nontransactional tables with RBR when `slave_exec_mode` is `IDEMPOTENT` can cause the master and slave to diverge even further. For more information about `slave_exec_mode`, see Section 5.1.3, “Server System Variables”.



## 4.6 如何验证主从数据一致

Maatkit是一个开源的工具包，为MySQL日常管理提供了帮助。目前，已被Percona公司收购并维护。其中mk-table-checksum是用来检测master和slave上的表结构和数据是否一致的。而mk-table-sync则是在主从数据不一致时，用来修复的。



注意 这两个perl脚本在运行时都会锁表，表的大小取决于执行的快慢，勿在高峰期间运行，可选择凌晨。

其使用方法如下：

```
[root@vm02]# mk-table-checksum h=vm01 , u=admin , p=123456 \  
h=vm02 , u=admin , p=123456 -d hcy -t t1  
Cannot connect to MySQL because the Perl DBI module is not installed  
or not found. Run 'perl -MDBI' to see the directories that Perl  
searches for DBI. If DBI is not installed , try:  
Debian/Ubuntu apt-get install libdbi-perl  
RHEL/CentOS yum install perl-DBI  
OpenSolaris pkg install pkg:/SUNWpmdbi
```

```
[root@vm02]#
```

这里提示缺少perl-DBI模块，那么直接运行yum install perl-DBI。再次执行这一条命令：

```
[root@vm02 bin]# mk-table-checksum h=vm01 , u=admin , p=123456 \  
h=vm02 , u=admin , p=123456 -d hcy -t t1  
DATABASE TABLE CHUNK HOST ENGINE COUNT CHECKSUM TIME  
WAIT STAT LAG  
hcy t1 0 vm02 InnoDB NULL 1957752020 0  
0 NULL NULL  
hcy t1 0 vm01 InnoDB NULL 1957752020 0  
0 NULL NULL  
[root@vm02 bin]#
```

如果表数据不一致，CHECKSUM的值是不相等的。



下面，解释一下输出的意思：

- DATABASE：数据库名
- TABLE：表名
- CHUNK：checksum时的近似数值
- HOST：MySQL的地址
- ENGINE：表引擎
- COUNT：表的行数
- CHECKSUM：校验值
- TIME：所用时间
- WAIT：等待时间
- STAT：master\_POS\_WAIT()返回值
- LAG：slave的延时时间

如果你想过滤出不相等的表，可以用mk-checksum-filter这个工具。只要在-d hcy后面加个管道符就行了。示例如下：

```
[root@vm02 ~]# mk-table-checksum h=vm01, u=admin, p=123456 \  
h=vm02, u=admin, p=123456 -d hcy | mk-checksum-filter  
hcy t2 0 vm01 InnoDB NULL 1957752020 0  
0 NULL NULL  
hcy t2 0 vm02 InnoDB NULL 1068689114 0  
0 NULL NULL
```

在知道有哪些表不一致后，可以用mk-table-sync这个工具来处理（如图4-3所示）。

## Maatkit工具包

MASTER上的t2表数据:

```
mysql> select * from t2;
```

id	name
1	a
2	b
3	ss
4	and
5	ss

5 rows in set (0.00 sec)

```
mysql> \! hostname;
```

vm01

```
mysql>
```

SLAVE上的t2表数据:

```
mysql> select * from t2;
```

id	name
1	a
2	b
3	ss
4	and

4 rows in set (0.00 sec)

```
mysql> \! hostname;
```

vm02

```
mysql>
```

图4-3 主从数据不一致

该工具的使用方法如下：

```
[root@vm02 ~]# mk-table-sync --execute --print --no-check-slave --transaction \
--databases hcy h=vm01 , u=admin , p=123456 h=vm02 , u=admin , p=123456
INSERT INTO 'hcy'. 't2' ('id' , 'name') VALUES ('5' , 'ss') /*maatkit src_db:hcy
src_tbl:t2 src_dsn:h=vm01 , p=... , u=admin dst_db:hcy dst_tbl:t2
dst_dsn:h=vm02 , p=... , u=admin lock:0 transaction:1 changing_src:0 replicate:0
bidirectional:0 pid:3246 user:root host:vm02*/;
```

它的工作原理是：先一行一行地检查主从库的表是否一样，如果发现有哪里不一样，就执行删除、更新、插入等操作，使其达到一致。表的大小决定着执行速度的快慢。



## 4.7 binlog\_ignore\_db引起的同步复制故障

这是我一个同事提供的案例，在MySQL master上使用binlog\_ignore\_db命令忽略了一个库以后，使用MySQL-e执行的所有语句就不写binlog了。听他这样说，我也觉得很奇怪，详细询问当时的情况，原来是在进行主从复制时，有一个库不复制，查了他的my.cnf配置，binlog格式为row模式，跟他要了当时的语句，如下：

```
MySQL -e "create table db.tb like db.tb1"
```

查看手册，了解到是因为忽略某个库的复制有两个参数，一个是binlog\_ignore\_db，另一个是replicate-ignore-db，它们的区别是：

binlog\_ignore\_db参数是设置在主库上的，例如，binlog\_ignore\_db=test，那么针对test库下的所有操作（增、删、改）都不会记录下来，这样slave在接收主库的binlog时文件量就会减少，这样做的好处是可以减少网络I/O，减少slave端I/O线程的I/O量，从而最大程度地优化复制性能。但也存在了一个隐患，对于这个隐患，在后面的演示中会提到。

replicate-ignore-db参数是设置在从库上的，例如，replicate-ignore-db=test，那么针对test库下的所有操作（增、删、改）都不会被SQL线程执行，从性能上来说，它虽然没有binlog\_ignore\_db好（不管是否需要，复制的binlog都会被I/O线程读取到slave端，这样不仅仅增加了网络I/O量，也给slave端的I/O线程增加了RelayLog的写入量），但在安全上，可以保证master和slave数据的一致性。

下面就来演示一下，创建表为什么没有记录在binlog日志里，如图4-4所示。

结果新创建的表在slave上一个都没有。到底是什么原因引起的呢？那就是因为没有使用use库名，如果使用了，就可以记录binlog了，如图4-5所示。



```
mysql> show master status;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000003 | 378      |               | test               |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> create table test1.number3 like test1.number;
Query OK, 0 rows affected (0.07 sec)

mysql> show master status;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000003 | 378      |               | test               |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

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

图4-4 binlog未发生变化

```
mysql> show master status;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000003 | 378      |               | test              |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> use test1;
Database changed
mysql>
mysql> create table test1.number4 like test1.number;
Query OK, 0 rows affected (0.04 sec)

mysql> show master status;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000003 | 486      |               | test              |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> show variables like '%binlog_format%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| binlog_format | ROW   |
+-----+-----+
```

图4-5 binlog发生变化

所以，如果想在slave上忽略一个库的复制，最好不要用binlog\_ignore\_db这个参数，使用replicate-ignore-db=yourdb取代之。

## 4.8 MySQL5.5.19/20同步一个Bug

当我们在用低版本MySQL5.1.43 ( slave ) 向高版本5.5.19 ( master ) 同步复制时，运维组反馈MySQL反复重启，后来，在错误日志里发现了一个Bug信息，如图4-6所示。

```
120803 22:04:06 - mysqld got signal 11 ;  
This could be because you hit a bug. It is also possible that this binary  
or one of the libraries it was linked against is corrupt, improperly built,  
or misconfigured. This error can also be caused by malfunctioning hardware.  
We will try our best to scrape up some info that will hopefully help diagnose  
the problem, but since we have already crashed, something is definitely wrong  
and this may fail.
```

图4-6 Bug信息

后来经过排查，得知低版本向高版本同步复制，只要同步复制的点指错，主机master的MySQL服务就会循环重启，MySQL和Percona版本均是如此，但版本一致的，就不会发生此问题。如果点指对，即使同步复制因为某种原因导致报错，主机master的MySQL服务也不会循环重启。

比如master上的binlog和POS点是：

*MySQL-bin.000001, 107*

那么你在slave上执行如下操作：

*CHANGE master TO master\_LOG\_FILE='MySQL-bin.000001', master\_LOG\_POS=106;*

就会触发那个BUG。有兴趣的朋友可以用那两个版本试试。官方下载地址：<http://downloads.mysql.com/archives.php>，目前MySQL5.5.25a之后的版本已经修复了此BUG。但不推荐低版本向高版本同步，手册上解释：

### 16.4.2. Replication Compatibility Between MySQL Versions

MySQL supports replication from one major version to the next higher major version. For example, you can replicate from a master running MySQL 4.1 to a slave running MySQL 5.0, from a master running MySQL 5.0 to a slave running MySQL 5.1, and so on.

意思为：MySQL支持从高版本向低版本同步（即master是低版本，slave是高版本，那么slave向master同步复制时是兼容的，没有问题）。



Replication from newer masters to older slaves may be possible, but is generally not supported. This is due to a number of factors:

- **Binary log format changes.** The binary log format can change between major releases. While we attempt to maintain backward compatibility, this is not always possible. For example, the binary log format implemented in MySQL 5.0 changed considerably from that used in previous versions, especially with regard to handling of character sets, `LOAD DATA INFILE`, and time zones. This means that replication from a MySQL 5.0 (or later) master to a MySQL 4.1 (or earlier) slave is generally not supported.

This also has significant implications for upgrading replication servers; see [Section 16.4.3, “Upgrading a Replication Setup”](#), for more information.

但反过来，就会存在问题。尤其是字符集设置这块，如图4-7所示。

```
Last_IO_Error:
Last_SQL_Errno: 22
Last_SQL_Error: Error 'Character set '45' is not a compiled character set and is not specified in the '/home/
mysql/mysql-5.1.43spl-br38368/share/charsets/Index.xml' file' on query. Default database: 'IICUPDB'. Query: 'DROP /*!40005 TEM
ORARY */ TABLE IF EXISTS `tt_DelChatFriendIds`'
1 row in set (0.00 sec)

ERROR:
```

图4-7 字符集未识别报错

在低版本V5.1.43里，是不识别高版本这种字符集的，如图4-8所示。

```
mysql> SET @@session.character_set_client=33,@@session.collation_connection=45,@@session.collation_server=33/*!*/;
ERROR 1273 (HY000): Unknown collation: '45'
mysql>
```

图4-8 低版本字符集未识别

再来看下高版本V5.5.19的表现，如图4-9所示。

```
mysql> SET @@session.character_set_client=33,@@session.collation_connection=45,@@session.collation_server=33/*!*/;  
Query OK, 0 rows affected (0.02 sec)  
  
mysql> select version();  
+-----+  
| version() |  
+-----+  
| 5.5.19-log |  
+-----+  
1 row in set (0.41 sec)  
  
mysql>
```

图4-9 高版本字符集识别

这条语句是可以识别的。

所以在这里提醒大家一下，如果master的版本低，slave的版本高，做主从复制是没有问题，可以兼容。但反过来，master的版本高，slave的版本低，就会出现文中这种情形，在日常操作中格外注意。



## 4.9 恢复slave从机上的某几张表的简要方法

在日常工作中，同步报错是数据库管理员遇到最多的一个问题，如果你修复后发现还没有解决，通常的方法就是在master上重新导出一份，然后在slave上恢复。这个方法是针对整个库不是很大的情况的，那如果是较大呢？全部导出再导入耗时就很长。

这时，就要通过特殊的方法恢复某几张表，例如，有a1、b1、c1这三张表的数据跟master上的不一致，操作方法如下：

1) 停止slave复制，命令如下：

```
MySQL>stop slave;
```

2) 在主库上导出这三张表，并记录下同步的binlog和POS点：

```
# MySQLdump -uroot -p123456 -q --single-transaction --master-data=2 yourdb a1 b1 c1 >./a1_b1_c1.sql
```

3) 查看a1\_b1\_c1.sql文件，找出记录的binlog和POS点：

```
# more a1_b1_c1.sql例如master_LOG_FILE='MySQL-bin.002974', master_LOG_POS=55056952;
```

4) 把a1\_b1\_c1.sql复制到slave机器上，并做Change master to指向：

```
MySQL>start slave until master_LOG_FILE='MySQL-bin.002974', master_LOG_POS=55056952;
```

直到sql\_thread线程为NO，这期间的同步报错一律跳过即可，可用如下命令跳过：

```
stop slave ;set global sql_slave_skip_counter=1;start slave;
```



注意 这一步是为了保障其他表的数据不丢失，一直同步，直到同步到那个点为止，a1、b1、c1表的数据在之前的导出已经生成了一份快照，只需要导入后开启同步即可。

5) 在slave机器上导入a1\_b1\_c1.sql，命令如下：

```
# MySQL -uroot -p123456 yourdb <./a1_b1_c1.sql
```

6) 导入完毕后，开启同步即可。

```
MySQL>start slave;
```

这样我们就恢复了3张表，并且同步也修复了。



## 4.10 如何干净地清除slave同步信息

比如，在某些应用场景，我们要下线一台slave从机，一般会执行reset slave来清除show slave status\G里面的同步信息，但当你再次执行时，发现并不是我们想要的结果，如下所示：

```
MySQL>stop slave;
Query OK, 0 rows affected (0.19 sec)
MySQL>reset slave;
Query OK, 0 rows affected (0.17 sec)
MySQL>show slave status\G;
***** 1. row *****

  slave_IO_State:
master_Host: 192.168.8.22
master_User: repl
master_Port: 3306
Connect_Retry: 10
master_Log_File:
Read_master_Log_Pos: 4
  Relay_Log_File: vm02-relay-bin.000001
Relay_Log_Pos: 4
Relay_master_Log_File:
slave_IO_Running: No
slave_SQL_Running: No
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 0
```



```

    Last_Error:
Skip_Counter: 0
Exec_master_Log_Pos: 0
Relay_Log_Space: 126
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
    master_SSL_Allowed: No
    master_SSL_CA_File:
    master_SSL_CA_Path:
    master_SSL_Cert:
master_SSL_Cipher:
    master_SSL_Key:
Seconds_Behind_master: NULL
master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
    Last_SQL_Errno: 0
    Last_SQL_Error:
Replicate_Ignore_Server_Ids:
master_Server_Id: 22
1 row in set (0.02 sec)
ERROR:
No query specified

```

执行reset slave，其实是把master.info和relay-log.info文件给删除，但里面的同步信息还在，如果此时有人误开启了start slave，结果就会又开始从头同步了，有可能还会造成数据丢失。那么可以用下面这个方法，让其清除得更为彻底。

```

MySQL>reset slave all;
Query OK, 0 rows affected (0.04 sec)
MySQL>show slave status\G;
Empty set (0.02 sec)

```



*ERROR:*

*No query specified*



注意 此语句支持在MySQL5.5.20或更高版本。







## 第5章 性能调优

调优，就好比盖楼打地基，地基打得不稳，楼层一高，就会塌方。数据库也是如此，数据少，并发小，隐藏的问题是发现不了的，只要达到一定规模后，所有的问题就会全部暴露出来了，所以前期的设计阶段尤为重要。如今，数据库的大并发查询、写入操作已成为整个应用的性能瓶颈，对于Web应用来说尤为明显。关于数据库的性能，并不只是DBA、运维人员才需要担心的事，更是所有广大开发人员需要重点关注的。

在宏观上来说，调优分为3个部分：硬件、网络、软件，前两个取决于你公司的经济实力，这里不多介绍。软件再细分为表设计（范式、字段类型、存储引擎）、SQL语句与索引、配置文件参数、操作系统、文件系统、MySQL版本、体系架构这几大部分。下面本章将进行逐个阐明。

## 5.1 表设计

在设计关系数据库时，要遵从不同的规范要求来设计出合理的关系型数据库，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。

目前关系数据库有6种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴德斯科范式（BCNF）、第四范式（4NF）和第五范式（5NF，又称完美范式）。满足最低要求的范式是第一范式（1NF）。在第一范式的基础上进一步满足更多规范要求的称为第二范式（2NF），其余范式依次类推。一般说来，数据库只需满足第三范式（3NF）就行了。

### 1. 第一范式（1NF）

第一范式（1NF）：数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整型、字符型、逻辑型、日期型等。在当前的任何关系数据库管理系统（DBMS）中，不可能做出不符合第一范式的数据库，因为这些DBMS不允许你把数据库表的一列再分成二列或多列。因此，你想在现有的DBMS中设计出不符合第一范式的数据库都是不可能的。

来看个例子，表5-1是一个病人信息表。

表5-1 病人信息表



病人编号	姓名	性别	出生日期	就诊记录	联系方式
2003001	王小姐	女	1982.10.14	孕后检查	133XXXXXXXX
2003002	苏小姐	女	1987.5.6	流产	132XXXXXXXX
2003003	合小姐	女	1989.12.3	痛经	135XXXXXXXX
2003004	崔小姐	女	1980.5.3	子宫内膜增厚	133XXXXXXXX
2003005	张小姐	女	1982.1.19	白带增多	158XXXXXXXX
2003006	闪小姐	女	1984.6.26	月经不调	159XXXXXXXX
2003007	赵先生	男	1983.4.14	精子弱	134XXXXXXXX
2003008	高先生	男	1978.2.26	勃起障碍	132XXXXXXXX
2003009	董先生	男	1985.6.16	早泄	131XXXXXXXX
2003010	邱先生	男	1985.9.4	包皮过长	130XXXXXXXX
2003011	李先生	男	1987.2.14	前列腺炎	148XXXXXXXX
2003012	周先生	男	1978.4.18	生殖器感染	154XXXXXXXX

如果把联系方式那列再分为手机和家庭电话，就不符合第一范式，如表5-2所示。

表5-2 病人信息表（不符合第一范式）

病人编号	姓名	性别	出生日期	就诊记录	联系方式	
2003001	王小姐	女	1982.10.14	孕后检查	6402XXXX	133XXXXXXXX
2003002	苏小姐	女	1987.5.6	流产	6402XXXX	132XXXXXXXX
2003003	合小姐	女	1989.12.3	痛经	6402XXXX	135XXXXXXXX
2003004	崔小姐	女	1980.5.3	子宫内膜增厚	6402XXXX	133XXXXXXXX
2003005	张小姐	女	1982.1.19	白带增多	6402XXXX	158XXXXXXXX
2003006	闪小姐	女	1984.6.26	月经不调	6402XXXX	159XXXXXXXX
2003007	赵先生	男	1983.4.14	精子弱	6402XXXX	134XXXXXXXX
2003008	高先生	男	1978.2.26	勃起障碍	6402XXXX	132XXXXXXXX
.....	.....	.....	.....	.....	.....	.....

那必须要把联系方式变更为住宅电话和手机号，该怎么办呢，可如表5-3一样来调整。

表5-3 病人信息表（符合第一范式）

病人编号	姓名	性别	出生日期	就诊记录	住宅电话	手机号
2003001	王小姐	女	1982.10.14	孕后检查	6402XXXX	133XXXXXXXX
2003002	苏小姐	女	1987.5.6	流产	6402XXXX	132XXXXXXXX
2003003	合小姐	女	1989.12.3	痛经	6402XXXX	135XXXXXXXX
2003004	崔小姐	女	1980.5.3	子宫内膜增厚	6402XXXX	133XXXXXXXX
.....	.....	.....	.....	.....	.....	.....

一句话总结：只要是关系数据库都满足第一范式。

## 2.第二范式（2NF）



第二范式 (2NF) 是在第一范式 (1NF) 的基础上建立起来的，即满足第二范式 (2NF) 必须先满足第一范式 (1NF)。第二范式 (2NF) 要求实体的属性完全依赖于主关键字。

还是以上面的病人信息表来举例，下面在上表的基础上新增加了一些字段，如图5-1所示。

病人编号	姓名	性别	出生日期	就诊记录	联系方式	医生编号	姓名	性别	职称	科室编号	科室名称	负责人	诊室号
2003001	王小姐	女	1982.10.14	孕后检查	133XXXX	1	张医生	女	主治医师	101	妇科	王某某	208
2003002	苏小姐	女	1987.5.6	流产	158XXXX	1	张医生	女	主治医师	101	妇科	王某某	208
2003003	合小姐	女	1989.12.3	痛经	158XXXX	1	张医生	女	主治医师	101	妇科	王某某	208
2003004	崔小姐	女	1980.5.3	子宫内膜增厚	138XXXX	2	王医生	女	主治医师	101	妇科	王某某	208
2003005	张小姐	女	1982.1.19	白带增多	159XXXX	2	王医生	女	主治医师	101	妇科	王某某	208
2003006	山小姐	女	1984.6.26	月经不调	137XXXX	2	王医生	女	主治医师	101	妇科	王某某	208
2003007	赵先生	男	1983.4.14	精子弱	132XXXX	3	李医生	男	主治医师	102	男科	贺某某	209
2003008	高先生	男	1978.2.26	勃起障碍	132XXXX	3	李医生	男	主治医师	102	男科	贺某某	209
2003009	董先生	男	1985.6.16	早泄	152XXXX	3	李医生	男	主治医师	102	男科	贺某某	209
2003010	邱先生	男	1985.9.4	包皮过长	186XXXX	4	郭医生	男	主治医师	102	男科	贺某某	209
2003011	李先生	男	1987.2.14	前列腺炎	134XXXX	4	郭医生	男	主治医师	102	男科	贺某某	209
2003012	周先生	男	1978.4.18	生殖器感染	132XXXX	4	郭医生	男	主治医师	102	男科	贺某某	209

图5-1 病人信息表 (增加字段)

图5-1中的信息存在着以下依赖关系：

{病人编号}←{姓名，性别，出生日期，就诊记录，联系方式}

{医生编号}←{姓名，性别，职称，科室编号，科室名称，负责人，诊室号}

这里，出现了两个主关键字，显然不满足第二范式。从表里的数据来看，出现了冗余数据：3名患者找同一个医生看病时，医生编号、姓名、职称等字段就重复了3次。

下面试着把上面的一张表改为三张表，如图5-2~图5-4所示。



病人信息表（主键：病人编号）					
病人编号	姓名	性别	出生日期	就诊记录	联系方式
2003001	王小姐	女	1982.10.14	孕后检查	133XXXX
2003002	苏小姐	女	1987.5.6	流产	158XXXX
2003003	合小姐	女	1989.12.3	痛经	158XXXX
2003004	崔小姐	女	1980.5.3	子宫内膜增厚	138XXXX
2003005	张小姐	女	1982.1.19	白带增多	159XXXX
2003006	山小姐	女	1984.6.26	月经不调	137XXXX
2003007	赵先生	男	1983.4.14	精子弱	132XXXX
2003008	高先生	男	1978.2.26	勃起障碍	132XXXX
2003009	董先生	男	1985.6.16	早泄	152XXXX
2003010	邱先生	男	1985.9.4	包皮过长	186XXXX
2003011	李先生	男	1987.2.14	前列腺炎	134XXXX
2003012	周先生	男	1978.4.18	生殖器感染	132XXXX

图5-2 病人信息表

医生信息表（主键：医生编号）						
医生编号	姓名	性别	科室编号	科室名称	负责人	诊室号
1	张医生	女	101	妇科	王某某	208
2	王医生	女	101	妇科	王某某	208
3	李医生	男	102	男科	贺某某	209
4	郭医生	男	102	男科	贺某某	209

图5-3 医生信息表

病人挂号信息表（主键：挂号单号，外键：病人编号、医生编号）		
挂号单流水号	病人编号	医生编号
20030531000001	2003001	1
20030531000002	2003002	1
20030531000003	2003003	1
20030531000004	2003004	2
20030531000005	2003005	2
20030531000006	2003006	2
20030531000007	2003007	3
20030531000008	2003008	3
20030531000009	2003009	3
20030531000010	2003010	4
20030531000001	2003011	4
20030531000001	2003012	4

图5-4 病人挂号信息表

这样就符合了第二范式，非关键字段都依赖于主键，但这样拆分是不符合第三范式的。

3.第三范式（3NF）

第三范式 (3NF) 是第二范式 (2NF) 的一个子集，即满足第三范式 (3NF) 必须满足第二范式 (2NF)。简而言之，不存在非关键字段对任一候选关键字段的传递函数依赖。

仍以上面的医生信息表为例，在图5-3里是存在着传递依赖关系的，如下所示：

- {医生编号}←{姓名，性别，职称}
- {科室编号}←{科室名称，负责人，诊室号}

可以看到，科室名称依赖着科室编号、科室编号依赖着医生编号，这里存在数据冗余，所以不符合第三范式，将这个表再进行拆分，如图5-5和图5-6所示。

医生信息表（主键：医生编号，外键：科室编号）			
医生编号	姓名	性别	科室编号
1	张医生	女	101
2	王医生	女	101
3	李医生	男	102
4	郭医生	男	102

图5-5 医生信息表（拆分后）

科室信息表（主键：科室编号）			
科室编号	科室名称	负责人	诊室号
101	妇科	王某某	208
102	男科	贺某某	209

图5-6 科室信息表（拆分后）

另外，图5-4也跟着变动，如图5-7所示。

经过上述调整，消除了数据冗余、更新异常、插入异常和删除异常。这些数据库表就符合第一至第三范式了。

E-R图也称实体-联系图，提供了表示实体类型、属性和联系的方法，程序设计初期就需要通过画E-R图来确定实体之间的关系。我们来看看病人和医生信息的E-R图，如图5-8所示。

病人挂号信息表（主键：挂号单号，外键：病人编号、科室编号、医生编号）

挂号单流水号	病人编号	科室编号	医生编号			
20030531000001	2003001	101	1			
20030531000002	2003002	101	1			
20030531000003	2003003	101	1			
20030531000004	2003004	101	2			
20030531000005	2003005	101	2			
20030531000006	2003006	101	2			
20030531000007	2003007	102	3			
20030531000008	2003008	102	3			
20030531000009	2003009	102	3			
20030531000010	2003010	102	4			
20030531000001	2003011	102	4			
20030531000001	2003012	102	4			

图5-7 病人挂号信息表（调整）

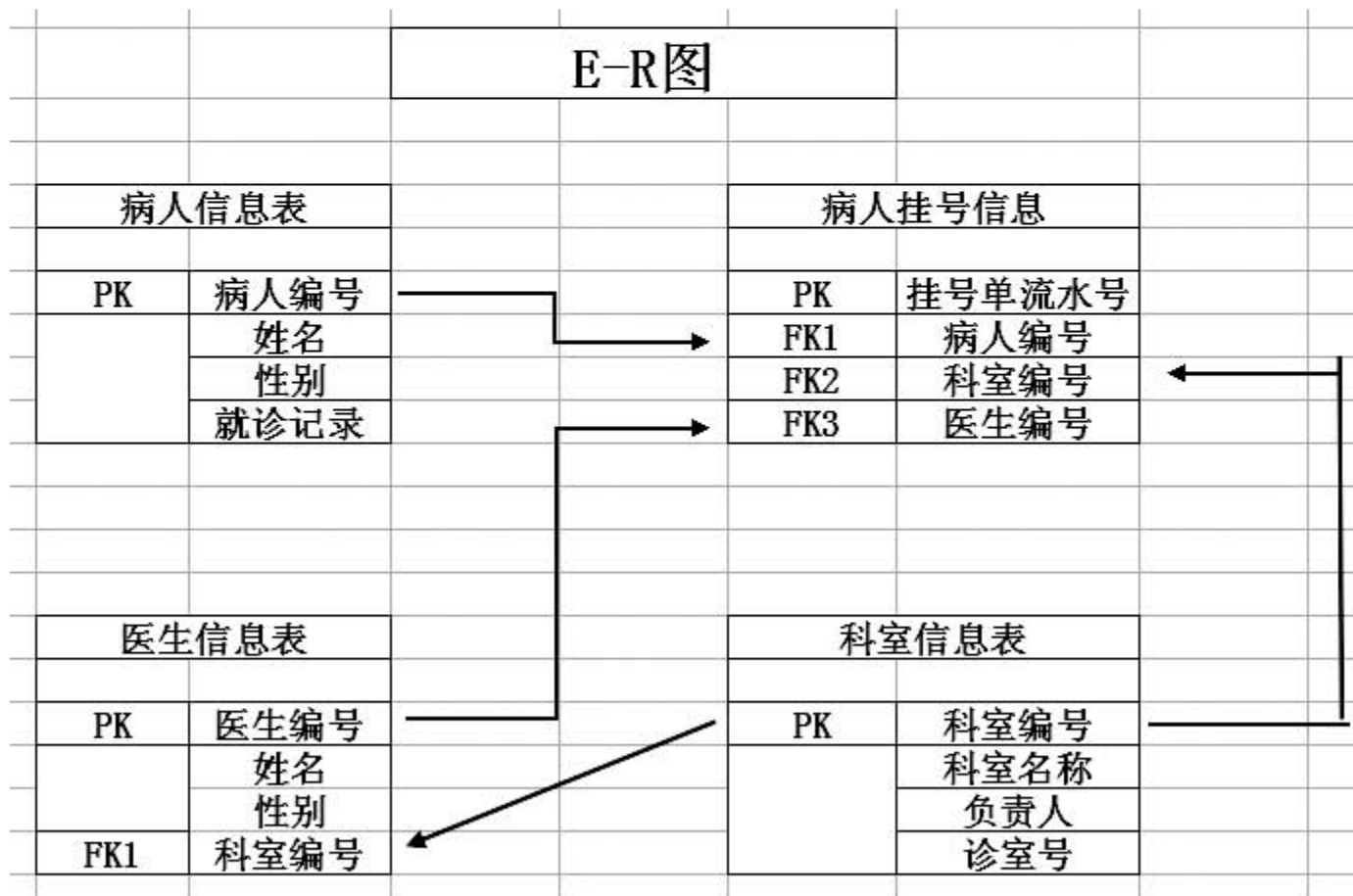


图5-8 实例关系图

小结：在开发应用程序时，设计的数据库要最大程度地遵守三范式，特别是对于OLTP型的系统来说，三范式是必须遵守的规则。当然，三



范式最大的问题在于查询时通常需要join很多表，而这会导致查询效率很低。所以有时候基于性能考虑，我们需要有意违反三范式，适度地做冗余，以达到提高查询效率的目的。注意，这里的反范式是适度的，必须为这种做法提供充分的理由。

## 5.2 字段类型的选取

选择字段的一般原则是保小不保大，能用占用字节少的字段就不用大字段。比如，主键，强烈建议用int整型，不用guid，为什么？省空间啊。空间是什么？空间就是效率！按4个字节和按32个字节定位一条记录，谁快谁慢太明显了。涉及几个表做join时，效果就更明显了。更小的字段类型占用的内存就更少，占用的磁盘空间和磁盘I/O也会更少，而且还会占用更少的带宽。因此，在日常选择字段时必须遵守这一规则。

5.2.1 数值类型

在MySQL中支持的5个主要整数类型是TINYINT、SMALLINT、MEDIUMINT、INT和BIGINT，表5-4列出了各种数值类型以及它们的允许范围和占用的内存空间。

表5-4 数值类型

类型	字节	最小值（有符号 / 无符号）	最大值（有符号 / 无符号）
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32 768	32 767
		0	65 535
MEDIUMINT	3	-8 388 608	8 388 607
		0	16 777 215
INT	4	-2 147 483 648	2 147 483 647
		0	4 294 967 295
BIGINT	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
		0	18 446 744 073 709 551 615



注意 这里的最小值和最大值代表的是宽度。

1.录入手机号带来的问题

朋友公司的开发人员考虑到varchar占用空间大，影响查询性能，于是他把字段varchar改成了int，结果在录入手机号时，溢出了，全部变



成了2147483647。我们知道，手机号是11位的，而int类型的最大宽度不能大于11，否则就溢出，下面来演示一下，表结构如图5-9所示。

```
mysql> create table phone(mobile int);
Query OK, 0 rows affected (0.06 sec)

mysql> show create table phone\G;
***** 1. row *****
      Table: phone
Create Table: CREATE TABLE `phone` (
  `mobile` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图5-9 表结构图

然后插入手机号，发现已经溢出，如图5-10所示。

在这种情况下，可以考虑把int类型转换为bigint，这样就不会存在溢出的情况了，如图5-11和图5-12所示。

```
mysql> insert into phone values (13581549970), (13511111111), (13611111111);
Query OK, 3 rows affected, 3 warnings (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 3

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message                                                                 |
+-----+-----+-----+
| Warning | 1264 | Out of range value for column 'mobile' at row 1 |
| Warning | 1264 | Out of range value for column 'mobile' at row 2 |
| Warning | 1264 | Out of range value for column 'mobile' at row 3 |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> select * from phone;
+-----+
| mobile |
+-----+
| 2147483647 |
| 2147483647 |
| 2147483647 |
+-----+
3 rows in set (0.00 sec)
```

图5-10 字段溢出图

```
mysql> alter table phone modify mobile bigint;
Query OK, 3 rows affected (0.12 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> show create table phone\G;
***** 1. row *****
      Table: phone
Create Table: CREATE TABLE `phone` (
  `mobile` bigint(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图5-11 更改表结构图

```
mysql> insert into phone values (13581549970), (13511111111), (13611111111);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from phone;
+-----+
| mobile |
+-----+
| 2147483647 |
| 2147483647 |
| 2147483647 |
| 13581549970 |
| 13511111111 |
| 13611111111 |
+-----+
6 rows in set (0.00 sec)
```

图5-12 插入数据

可能有人会问，前面那种情况下也可以设置为char(11)啊！这并非不可以，但考虑到占用空间的问题，一般程序的字符集是gbk或utf8，gbk占用2字节，utf8占用3字节，那么11×3就是33字节，而bigint(20)宽度为20，只占用8字节，从性能上考虑，应该设置为bigint。

## 2. IP地址也可采用int整型

MySQL里提供了一个很好用的函数：INET\_ATON()，它负责把IP地址转为数字，而另一个函数INET\_NTOA()负责把数字转换为IP地址，在使用这两个函数时也要注意溢出问题，下面我们来演示一下，如图5-13所示。



```
mysql> show create table ipaddress\G;
***** 1. row *****
      Table: ipaddress
Create Table: CREATE TABLE `ipaddress` (
  `ip` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

ERROR:
No query specified

mysql> insert into ipaddress values(INET_ATON('127.0.0.1'));
Query OK, 1 row affected (0.00 sec)

mysql> insert into ipaddress values(INET_ATON('192.168.1.1'));
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message                                     |
+-----+-----+-----+
| Warning | 1264 | Out of range value for column 'ip' at row 1 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-13 表结构和插入数据

这是什么情况啊？居然溢出了！我们来看看查询结果，如图5-14所示。

```
mysql> select INET_NTOA(ip) from book.ipaddress;
+-----+
| INET_NTOA(ip) |
+-----+
| 127.0.0.1      |
| 127.255.255.255 |
+-----+
2 rows in set (0.01 sec)
```

图5-14 查询结果

从查询结果来看，IP地址不是之前的192.168.1.1，那怎么办呢？还得用char(15)吗？答案是NO，细心的朋友应该注意到了，int整型有符号最大宽度是2147483647，而无符号最大宽度是4294967295，所以要更改下表结构才可以存下，如图5-15所示。

继续插入刚才的IP地址，如图5-16所示。

OK！没有任何问题。

再比如，我们要查询192.168这个网段内的所有IP地址和对应的主机名，可采用如图5-17所示的方式。

```
mysql> alter table ipaddress modify ip int(11) unsigned DEFAULT NULL;
Query OK, 2 rows affected (0.03 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> show create table ipaddress\G;
***** 1. row *****
      Table: ipaddress
Create Table: CREATE TABLE `ipaddress` (
  `ip` int(11) unsigned DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图5-15 更改表结构

```
mysql> insert into ipaddress values(INET_ATON('192.168.1.1'));
Query OK, 1 row affected (0.00 sec)

mysql> insert into ipaddress values(INET_ATON('192.168.254.254'));
Query OK, 1 row affected (0.00 sec)

mysql> select INET_NTOA(ip) from ipaddress;
+-----+
| INET_NTOA(ip) |
+-----+
| 127.0.0.1      |
| 127.255.255.255 |
| 192.168.1.1    |
| 192.168.254.254 |
+-----+
4 rows in set (0.00 sec)
```



图5-16 查询结果

```
mysql> select INET_NTOA(ip),hostname1 from ipaddress where ip >=INET_ATON(192.168);
```

INET_NTOA(ip)	hostname1
192.168.1.1	testA
192.168.254.254	testB

```
2 rows in set (0.00 sec)
```

```
mysql> explain select INET_NTOA(ip),hostname1 from ipaddress where ip >=INET_ATON(192.168);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ipaddress	range	ix_ip	ix_ip	5	NULL	2	Using where

```
1 row in set (0.00 sec)
```

图5-17 查询执行计划

### 3. 根据需求选择最小整数类型

有不少开发人员在设计表字段时，只要是针对数值类型的全部用int，但这不一定合适，就比如用户的年龄，一般来说，年龄大都在1~100岁之间，长度只有3，那么用int就不适合了，可以用tinyint代替。又比如，用户在线状态，0表示离线、1表示在线、2表示离开、3表示忙碌、4表示隐身等，其实类似这样的情况，用int都是没有必要的，浪费空间，采用tinyint完全可以满足需要，int占用的是4字节，而tinyint才占用1个字节。

这里估计有人会问，用enum枚举类型也可以啊，它也占用1字节。但是采用enum枚举类型，会存在扩展的问题，还是用上面用户在线状态的例子，如果此时增加了：5表示请勿打扰、6表示开会中、7表示隐身对好友可见，那么此时只有更改表字段类型了，虽然Percona公司出了pt-online-schema-change在线更改表工具，另外MySQL5.6新特性在线DDL也避免了锁表，可以实现这个功能，但这个动作还是很大，所以建议在表设计之初就把问题考虑周全，免得日后亡羊补牢，这样就得不偿失了。

下面是关于ENUM枚举类型的一个小例子。在该例中，枚举的数值是0、1、2、3、4，如果插入的值不在这个范围里，就会报错，如图5-18和图5-19所示。所以一般用tinyint来代替enum比较合适。



```
mysql> show create table enumtest\G;
***** 1. row *****
      Table: enumtest
Create Table: CREATE TABLE `enumtest` (
  `status` enum('0','1','2','3','4') DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

ERROR:
No query specified

mysql> insert into enumtest values('0'),('1'),('2'),('3'),('4');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

图5-18 表结构和插入数据

```
mysql> insert into enumtest values('6');
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> show warnings;
+-----+-----+-----+
| Level   | Code | Message                                     |
+-----+-----+-----+
| Warning | 1265 | Data truncated for column 'status' at row 1 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from enumtest;
+-----+
| status |
+-----+
| 0      |
| 1      |
| 2      |
| 3      |
| 4      |
+-----+
6 rows in set (0.00 sec)
```

图5-19 查询结果

### 5.2.2 字符类型

char和varchar是日常使用最多的字符类型。char(N)用于保存固定长度的字符串，长度最大为255，比指定长度大的值将被截短，而比指定长度小的值将会用空格进行填补。

varchar(N)用于保存可变长度的字符串，长度最大为65535，只存储字符串实际需要的长度（它会增加一个额外字节来存储字符串本身的长度），varchar使用额外的1~2字节来存储值的长度，如果列的最大长度小于或等于255，则使用1字节，否则就是用2字节。

char和varchar跟字符编码也有密切联系，latin1占用1个字节，gbk占用2个字节，utf8占用3个字节。

#### 1. 计算varchar的最大长度

不同的字符集所占用的存储空间会不同，具体如表5-5~表5-7所示。

表5-5 Latin1字符集存储字节

Latin1（1 个字符 1 个字节）

值	Char(4)	存储的字节	Varchar(4)	存储的字节
"	' '	4 字节	"	1 字节
'ab'	'ab '	4 字节	'ab'	3 字节
'abcd'	'abcd'	4 字节	'abcd'	5 字节
'abcdefgh'	'abcd'	4 字节	'abcd'	5 字节

表5-6 Gbk字符集存储字节



Gbk（1 个字符 2 个字节）

值	Char(4)	存储的字节	Varchar(4)	存储的字节
"	' '	8 字节	"	1 字节
'ab'	'ab '	8 字节	'ab'	5 字节
'abcd'	'abcd'	8 字节	'abcd'	9 字节
'abcdefgh'	'abcd'	8 字节	'abcd'	9 字节

表5-7 Utf8字符集存储字节

Utf8（1 个字符 3 个字节）

值	Char(4)	存储的字节	Varchar(4)	存储的字节
"	' '	12 字节	"	1 字节
'ab'	'ab '	12 字节	'ab'	7 字节
'abcd'	'abcd'	12 字节	'abcd'	13 字节
'abcdefgh'	'abcd'	12 字节	'abcd'	13 字节

引申出一个问题，既然跟字符编码有关系，那么如何计算出varchar(N)的最大长度？下面就经常使用的gbk和utf8来举例说明。

(1) GBK字符集

首先创建一张表，表里采用gbk字符集，可以看到，字段v设置为32766可以正常创建，但超过了就失败。

```
mysql> create table t1(v varchar(32766))charset=gbk;
Query OK, 0 rows affected (0.04 sec)

mysql> create table t2(v varchar(32767))charset=gbk;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
mysql>
```

图5-20 创建表gbk字符集

因为varchar类型长度大于255，因此这里要用2字节存储值的长度。

计算公式： $(65535-2)/2=32766.5$ ，也就是说不能大于32767。

## (2) UTF8字符集

同样创建一张表，表里采用utf8字符集，可以看到，字段v设置为21844可以正常创建，但超过了就失败。

```
mysql> create table t3(v varchar(21844))charset=utf8;
Query OK, 0 rows affected (0.04 sec)

mysql> create table t4(v varchar(21845))charset=utf8;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
mysql>
```

图5-21 创建表字符集为utf8

因为varchar类型长度大于255，因此这里要用3字节存储值的长度。

计算公式： $(65535-2)/3=21844.3$ ，也就是说不能大于21845。

有人会问，一个表里肯定有好几个字段，那么varchar(N)的最大长度还是按照上面的公式计算吗？事实上，这时就需要变通一下了，假设一个表有如下字段：

*(id int, username varchar(20), phone bigint, address varchar(N))*

我们来计算下address的最大长度，这里采用GBK字符集，代码如下：

*create table info*

*(id int, username varchar(20), phone bigint, address varchar(32766))charset=gbk;*

SQL建表语句的执行结果如图5-22所示。

```
mysql> create table info(id int,username varchar(20),phone bigint,address varchar(32766))charset=gbk;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
mysql>
```

图5-22 建表

很奇怪吧？这里输入32766这个长度居然报错了。事实上，MySQL规定一个行的定义长度不能超过65535。若定义的表长度超过这个值，则会报错。而在这个info表中：id字段占用4字节，username字段占用41字节（因为长度小于255，这里要用1字节存储值的长度），phone字段占用8字节，所以计算结果为 $(65535-4-20\times 2+1-8-2)/2=32740$ ，也就是不能大于32740，前面输入的32766明显大于此数，自然会报错了。OK！



我们来验证一下，如图5-23所示。

```
mysql> create table info(id int,username varchar(20), phone bigint,address varchar(32740))charset=gbk;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
mysql>
mysql> create table info(id int,username varchar(20), phone bigint,address varchar(32739))charset=gbk;
Query OK, 0 rows affected (0.04 sec)
```

图5-23 建表

address varchar(32739)，顺利创建成功！

下面采用UTF8字符集来看看该例的情况，同样创建相同的info表，字符集为utf8，依据上面的计算公式，得出address的最在长度是：  
 $(65535 - 4 - 20 \times 3 + 1 - 8 - 2) / 3 = 21820$ ，也就是不能大于21820，OK！我们来验证一下，如图5-24所示。

```
mysql> create table info1 (id int,username varchar(20),phone bigint,address varchar(21820))charset=utf8;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
mysql>
mysql> create table info1 (id int,username varchar(20),phone bigint,address varchar(21819))charset=utf8;
Query OK, 0 rows affected (0.03 sec)
```

图5-24 建表

跟我们预期完全一致。

## 2.在什么情况下使用char和varchar

经常变化的值，如家庭住址，由于每个人地址都不同，有的地址很长有的很短，那么用varchar相对比较合适，因为它只存储字符串实际的长度。

而对于固定长度的值，比如uuid函数，是数字和字母组成的36位长度的字符类型，长度是固定不变的，或者是md5加密后的32位长度的字符类型，那么可以设置为char(36)和char(32)，这样相比于varchar，还节省空间，因为varchar还要用1字节存储值的长度。

我的前同事问过我这么一个问题：既然varchar只存储字符串实际的长度，那么使用varchar(20)或varchar(100)存储'abc'字节都会一样，那我干脆在设计表时就把值定义得大一些，为今后的扩展先预留出来。

对于这个问题，虽然两者存储的空间是一样的，但二者的性能完全不一样，MySQL需要先在内存中分配固定的空间来保存值，这无形中就浪费了内存，而且对表的排序或使用临时表尤其不好，所以只分配真正需要的那部分空间即可。



5.2.3 时间类型

在MySQL中支持的5个时间类型是DATE、TIME、DATETIME、TIMESTAMP和YEAR，表5-8列出了各种数值类型以及它们的允许范围和占用的内存空间。

表5-8 时间类型

数据类型	值	存储的字节	数据类型	值	存储的字节
Date	'0000-00-00'	3 字节	timestamp	'0000-00-00 00:00:00'	4 字节
Time	'00:00:00'	3 字节	Year	0000	1 字节
Datetime	'0000-00-00 00:00:00'	8 字节			

MySQL提供了5种时间类型，datetime和timestamp都可以精确到秒，但datetime占用8字节，而timestamp只占用4字节，在日常建表时应优先选择timestamp类型。timestamp还具有自动更新时间功能，下面来给大家演示一下。

```
首先，建立一张表，命令如下：
mysql> create table t1(id int, ctime timestamp);
Query OK, 0 rows affected (0.20 sec)
然后只针对id插入值，其查询结果如图5-25所示。
```

```
mysql> select * from t1;
Empty set (0.00 sec)

mysql> insert into t1(id) values(1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
|      1 | 2013-06-02 22:05:09 |
+-----+-----+
1 row in set (0.00 sec)
```

图5-25 查询结果

此时ctime字段会自动插入当前的时间，如图5-26所示。

```
mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
|      1 | 2013-06-02 22:05:09 |
+-----+-----+
1 row in set (0.00 sec)

mysql> update t1 set id=id+1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
|      2 | 2013-06-02 22:08:43 |
+-----+-----+
1 row in set (0.00 sec)
```

图5-26 ctime字段自动更新时间

当更新id字段，ctime的时间也会自动更新。

可能有人看到这里会有疑问，我的业务需求并不是这样的，我不想让系统自动更新为当前时间，应该怎么办呢？timestamp在默认的情况下是：

*DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP*

那么只需更改默认值为空，这样就跟datetime类型完全一样了。插入、更新完毕后，都不会自动更改ctime字段为当前时间，如图5-27和图5-28所示。

```
mysql> alter table t1 modify ctime timestamp NULL;
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
| 2     | 2013-06-02 22:08:43 |
| 3     | 2013-06-02 22:40:29 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> insert into t1(id) values(4);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
| 2     | 2013-06-02 22:08:43 |
| 3     | 2013-06-02 22:40:29 |
| 4     | NULL                 |
+-----+-----+
3 rows in set (0.00 sec)
```

图5-27 修改timestamp默认值



```
mysql> update t1 set id=id+10;
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0

mysql> select * from t1;
+-----+-----+
| id    | ctime                |
+-----+-----+
| 12    | 2013-06-02 22:08:43 |
| 13    | 2013-06-02 22:40:29 |
| 14    | NULL                 |
+-----+-----+
3 rows in set (0.00 sec)
```

图5-28 时间未自动更新

### 1.在MySQL5.6中，时间类型timestamp和datetime有了重大改变

在MySQL5.5（或更老的版本MySQL5.1）里，对于timestamp类型，一个表里只允许一个字段既拥有自动插入时间，又拥有自动更新时间。但从MySQL5.6开始，打破了这一传统理念，以上条件均可以出现，并且datetime类型也拥有了timestamp类型的功能。以下是相应的演示内容。

图5-29和图5-30展示的是MySQL5.5的timestamp类型，一个表里只允许一个字段拥有自动插入时间和自动更新时间。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> create table time (id int,t timestamp,t2 timestamp);
Query OK, 0 rows affected (0.04 sec)

mysql> insert into time(id) values(1),(2);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from time;
+----+-----+-----+
| id | t                | t2                |
+----+-----+-----+
| 1  | 2012-09-26 12:23:08 | 0000-00-00 00:00:00 |
| 2  | 2012-09-26 12:23:08 | 0000-00-00 00:00:00 |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图5-29 单个字段自动插入时间

```
mysql> update time set id=id+10;
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> select * from time;
+----+-----+-----+
| id | t                | t2                |
+----+-----+-----+
| 11 | 2012-09-26 12:23:41 | 0000-00-00 00:00:00 |
| 12 | 2012-09-26 12:23:41 | 0000-00-00 00:00:00 |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图5-30 单个字段自动更新时间

图5-31和图5-32是MySQL5.6的timestamp类型，一个表里可以有多个字段拥有自动插入时间和自动更新时间。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.01 sec)

mysql> create table time (id int,t timestamp DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
-> t2 timestamp DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
Query OK, 0 rows affected (0.35 sec)

mysql> insert into time(id) values(1),(2);
Query OK, 2 rows affected (0.09 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> select * from time;
+----+-----+-----+
| id | t                | t2                |
+----+-----+-----+
| 1  | 2012-09-25 22:14:51 | 2012-09-25 22:14:51 |
| 2  | 2012-09-25 22:14:51 | 2012-09-25 22:14:51 |
+----+-----+-----+
2 rows in set (0.02 sec)
```

图5-31 多个字段自动插入当前时间



```
mysql> update time set id=id+10;
Query OK, 2 rows affected (0.07 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> select * from time;
+-----+-----+-----+
| id    | t                | t2                |
+-----+-----+-----+
| 11    | 2012-09-25 22:15:28 | 2012-09-25 22:15:28 |
| 12    | 2012-09-25 22:15:28 | 2012-09-25 22:15:28 |
+-----+-----+-----+
2 rows in set (0.03 sec)
```

图5-32 多个字段自动更新当前时间

MySQL5.5不支持多个字段设置为timestamp类型，如果设置就会报错，如图5-33所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> create table time (id int,t timestamp DEFAULT CURRENT_TIMESTAMP,
-> t2 timestamp ON UPDATE CURRENT_TIMESTAMP);
ERROR 1293 (HY000): Incorrect table definition; there can be only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or ON
UPDATE clause
mysql>
```

图5-33 不允许多个字段设置timestamp时间类型

图5-34和图5-35是MySQL5.6支持多个字段设置timestamp类型。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.01 sec)

mysql> create table time (id int,t timestamp DEFAULT CURRENT_TIMESTAMP,
-> t2 timestamp ON UPDATE CURRENT_TIMESTAMP);
Query OK, 0 rows affected (0.46 sec)

mysql> insert into time(id) values(1),(2);
Query OK, 2 rows affected (0.07 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> select * from time;
+-----+-----+-----+
| id | t | t2 |
+-----+-----+-----+
| 1 | 2012-09-25 23:37:50 | NULL |
| 2 | 2012-09-25 23:37:50 | NULL |
+-----+-----+-----+
2 rows in set (0.03 sec)
```

图5-34 允许多个字段设置timestamp时间类型

```
mysql> update time set id=3 where id=2;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from time;
```

id	t	t2
1	2012-09-25 23:37:50	NULL
3	2012-09-25 23:37:50	2012-09-25 23:38:01

```
2 rows in set (0.02 sec)
```

图5-35 多个字段自动更新当前时间



```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.01 sec)

mysql> create table time (id int, d_time datetime DEFAULT CURRENT_TIMESTAMP,
-> d_time2 datetime ON UPDATE CURRENT_TIMESTAMP);
Query OK, 0 rows affected (0.45 sec)

mysql> insert into time(id) values(1), (2);
Query OK, 2 rows affected (0.06 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> select * from time;
+-----+-----+-----+
| id | d_time | d_time2 |
+-----+-----+-----+
| 1 | 2012-09-25 23:40:56 | NULL |
| 2 | 2012-09-25 23:40:56 | NULL |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

图5-36 datetime类型支持自动插入当前时间

```
mysql> update time set id=3 where id=2;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from time;
+-----+-----+-----+
| id    | d_time                | d_time2                |
+-----+-----+-----+
| 1     | 2012-09-25 23:40:56   | NULL                   |
| 3     | 2012-09-25 23:40:56   | 2012-09-25 23:41:07   |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

图5-37 datetime类型支持自动更新当前时间

图5-36和图5-37展示的是在MySQL5.6中，datetime类型也拥有了timestamp类型的功能。

参考手册：

Previously, at most one **TIMESTAMP** column per table could be automatically initialized or updated to the current date and time. This restriction has been lifted. Any **TIMESTAMP** column definition can have any combination of **DEFAULT CURRENT\_TIMESTAMP** and **ON UPDATE CURRENT\_TIMESTAMP** clauses. In addition, these clauses now can be used with **DATE-TIME** column definitions. For more information, see [Section 11.3.5, “Automatic Initialization and Updating for TIMESTAMP and DATETIME”](#).

2.在MySQL5.6中，year(2)类型会自动转换为year(4)

在MySQL5.6里，对year(2)的类型已经不再识别了，会自动转换为year(4)，如果你插入一条记录12，会自动转换为2012，下面请看演示。

图5-38和图5-39演示的是在MySQL5.6中，year(2)类型自动转换为year(4)。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.02 sec)

mysql> CREATE TABLE y (y2 YEAR(2), y4 YEAR(4));
Query OK, 0 rows affected, 1 warning (0.40 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Warning | 1818 | YEAR(2) column type is deprecated. Creating YEAR(4) column instead. |
+-----+-----+-----+
1 row in set (0.01 sec)

mysql> desc y;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| y2 | year(4) | YES | | NULL | |
| y4 | year(4) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.09 sec)
```

图5-38 表结构信息



```
mysql> insert into y values(12,2012);
Query OK, 1 row affected (0.07 sec)

mysql> select * from y;
+-----+-----+
| y2    | y4    |
+-----+-----+
| 2012  | 2012  |
+-----+-----+
1 row in set (0.03 sec)
```

图5-39 year类型查询结果

再来看一下在MySQL5.5中的情况，如图5-40所示。

```

+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> CREATE TABLE y (y2 YEAR(2), y4 YEAR(4));
Query OK, 0 rows affected (0.04 sec)

mysql> desc y;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| y2    | year(2) | YES  |     | NULL    |       |
| y4    | year(4) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> insert into y values(12,2012);
Query OK, 1 row affected (0.00 sec)

mysql> select * from y;
+-----+-----+
| y2    | y4    |
+-----+-----+
| 12    | 2012  |
+-----+-----+

```

图5-40 year类型查询结果

参见手册：

The `YEAR(2)` data type is now deprecated. `YEAR(2)` columns in existing tables are treated as before, but `YEAR(2)` in new or altered tables are converted to `YEAR(4)`. For more information, see [Section 11.3.4, “YEAR\(2\) Limitations and Migrating to YEAR\(4\)”](#).



## 5.2.4 小技巧：快速修改表结构

注

俗话说计划赶不上变化，需求的变更是一天一个样。

我们知道更改一个字段，增加、删除一个字段都会造成锁表，可如今动不动就是需要7×24小时不间断服务的，就算是在低峰期更改表结构，也会引起用户无法写入，会对业务造成影响，为了避免这一问题，本节将给大家介绍一个相关的小技巧。



注意 这个技巧是没有列入手册里的，所以请大家在测试时先备份你的表，免得造成丢失数据！

测试一：修改varchar类型

为了验证此方法，首先创建一张表t1，name字段设置为varchar(5)，如图5-41所示。

```
mysql> create table t1(
  -> id int,name varchar(5),
  -> rmb decimal(9,1));
Query OK, 0 rows affected (0.46 sec)

mysql> insert into t1 values(1,'zhangsan',3.8);
Query OK, 1 row affected, 1 warning (0.05 sec)

mysql> select * from t1;
+-----+-----+-----+
| id    | name  | rmb    |
+-----+-----+-----+
|      1 | zhang | 3.8    |
+-----+-----+-----+
1 row in set (0.03 sec)
```

图5-41 表结构信息

这时变更表结构，把name varchar(5)改为varchar(10)，注意，这里没有用传统的方法alter table modify，而是采用的如下步骤：  
步骤一，创建一张临时表，把varchar设置为10，如图5-42所示。

```
mysql> create table t1_tmp(
  -> id int,name varchar(10),
  -> rmb decimal(9,1));
Query OK, 0 rows affected (0.32 sec)
```

图5-42 创建和原表相同的临时表

步骤二，替换.frm表结构文件，如图5-43所示。

```
mysql> flush tables with read lock;
Query OK, 0 rows affected (0.02 sec)

mysql> system cp /usr/local/mysql/data/book/t1_tmp.frm /usr/local/mysql/data/book/t1.frm
```

图5-43 替换.frm表结构文件



注意 这里一定要先锁定表，防止表被打开，以免数据丢失！  
步骤三，插入数据测试，如图5-44所示。

```
mysql> unlock tables;
Query OK, 0 rows affected (0.02 sec)

mysql> insert into t1 values(2, 'hechunyang', 3.8);
Query OK, 1 row affected (0.11 sec)

mysql> select * from t1;
+-----+-----+-----+
| id    | name      | rmb    |
+-----+-----+-----+
| 1     | zhang     | 3.8    |
| 2     | hechunyang | 3.8    |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

图5-44 插入数据

再看看表结构，如图5-45所示。



```
mysql> show create table t1\G;
***** 1. row *****
      Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(10) DEFAULT NULL,
  `rmb` decimal(9,1) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.05 sec)

ERROR:
No query specified
```

图5-45 验证表结构已经更改

可以看到，已经将name varchar(5)顺利地更改为了name varchar(10)。

不过，你如果做了主从复制同步，这时slave同步会报错，如图5-46所示。

```
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 1677
Last_SQL_Error: Column 1 of table 'book.t1' cannot be converted from type 'varchar(30)' to type 'varchar(5)'
Replicate_Ignore_Server_Ids:
Master_Server_Id: 22
1 row in set (0.00 sec)
```

图5-46 主从复制同步报错信息

解决的方法跟上面的步骤一样，即在slave上也重新执行一遍相关步骤，然后跳过这个错误即可，如图5-47所示。

```
mysql> system cp /usr/local/mysql/data/book/t1_tmp.frm /usr/local/mysql/data/book/t1.frm
mysql>
mysql> flush tables;
Query OK, 0 rows affected (0.02 sec)

mysql> stop slave ;set global sql_slave_skip_counter=1;start slave;
Query OK, 0 rows affected (0.10 sec)

Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.06 sec)

mysql> show slave status\G;
***** 1. row *****
      Slave_IO_State: Waiting for master to send event
        Master_Host: 192.168.8.22
        Master_User: repl
        Master_Port: 3306
        Connect_Retry: 60
        Master_Log_File: mysql-bin.000002
  Read_Master_Log_Pos: 1859
        Relay_Log_File: vm02-relay-bin.000008
        Relay_Log_Pos: 253
  Relay_Master_Log_File: mysql-bin.000002
      Slave_IO_Running: Yes
      Slave_SQL_Running: Yes
      Replicate_Do_DB:
```

图5-47 slave机器上操作步骤

测试二：修改decimal浮点型

看过了上面的varchar类型修改字段值，那么decimal类型是否也可以呢？往下看：

步骤一，创建一张临时表，把decimal设置为(9, 2)，如图5-48所示。

```
mysql> alter table t1_tmp modify rmb decimal(9,2) DEFAULT NULL;
Query OK, 0 rows affected (0.75 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> show create table t1_tmp\G;
***** 1. row *****
      Table: t1_tmp
Create Table: CREATE TABLE `t1_tmp` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(10) DEFAULT NULL,
  `rmb` decimal(9,2) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.03 sec)
```

图5-48 更改表结构信息

步骤二，替换.frm表结构文件，如图5-49所示。



```
mysql> flush tables with read lock;
Query OK, 0 rows affected (0.02 sec)

mysql> select * from t;
ERROR 1146 (42S02): Table 'book.t' doesn't exist
mysql> select * from t1;
+-----+-----+-----+
| id    | name          | rmb    |
+-----+-----+-----+
| 1     | zhang         | 3.8    |
| 2     | hechunyang    | 3.8    |
+-----+-----+-----+
2 rows in set (0.04 sec)

mysql> system cp /usr/local/mysql/data/book/t1_tmp.frm /usr/local/mysql/data/book/t1.frm;
mysql>
mysql> flush tables;
Query OK, 0 rows affected (0.02 sec)
```

图5-49 替换.frm表结构文件

步骤三，进行数据测试，如图5-50所示。

```
mysql> system cp /usr/local/mysql/data/book/tl_tmp.frm /usr/local/mysql/data/book/tl.frm;
mysql>
mysql> flush tables;
Query OK, 0 rows affected (0.02 sec)

mysql> select * from tl;
+-----+-----+-----+
| id    | name      | rmb    |
+-----+-----+-----+
| 1     | zhang     | 3.08   |
| 2     | hechunyang | 3.08   |
+-----+-----+-----+
2 rows in set (0.04 sec)
```

图5-50 查询数据结果

细心的读者会发现原来的人民币3.8元变成了3.08元，改后的值有了误差，这个快速修改表结构的方法对varchar有用，对decimal无效，使用时一定要多测试。

- 本节内容的问题发现者是我的同事吴炳锡，本人参与校验与测试。

### 5.2.5 pt-online-schema-change在线更改表结构

这个工具来自Percona公司，它解决了更改表时不锁表的问题，且不会影响到业务。其原理是：

- 1) 如果存在外键，根据alter-foreign-keys-method参数的值，检测外键相关的表，针对相应的设置进行处理。
- 2) 创建一个新的表，表结构为修改后的数据表，用于从源数据表向新表中导入数据。
- 3) 创建触发器，在复制数据开始之后，将对源数据表继续进行数据修改的操作记录下来，以便在数据复制结束后执行这些操作，保证数据不会丢失。
- 4) 复制数据，从源数据表中复制数据到新表中。
- 5) 修改外键相关的子表，根据修改后的数据，修改外键关联的子表。
- 6) 更改源数据表为old表，把新表更改为源表名，并将old表删除。
- 7) 删除触发器。

该工具的安装命令如下：

```
# tar xzf percona-toolkit-2.2.2.tar.gz
# cd percona-toolkit-2.2.2
# perl Makefile.PL
```

如果这一步报错，需要通过yum安装perl-DBD-MySQL。

```
# make;make install
```

下面就来看看这个工具是如何使用的。

测试一：无主键的表增加字段

下面的测试是针对无主键的表增加字段，看看会发生什么：

```
[root@vm01 ~]# pt-online-schema-change --alter="add column age tinyint" --user=root D=book, t=t1 --execute
Cannot connect to D=book, h=vm02.localdomain, u=root
Operation, tries, wait:
  copy_rows, 10, 0.25
  create_triggers, 10, 1
```



```
drop_triggers , 10 , 1
swap_tables , 10 , 1
update_foreign_keys , 10 , 1
```

Altering 'book'. 't1'...

Creating new table...

Created new table book.\_t1\_new OK.

Altering new table...

Altered 'book'. '\_t1\_new' OK.

Dropping new table...

Dropped new table OK.'book'. 't1' was not altered.

The new table 'book'. '\_t1\_new' does not have a PRIMARY KEY or a unique index which is required for the DELETE trigger.

结论：给没有主键的表增加字段，该工具会报错。

测试二：无主键的表增加主键

下面的测试是给无主键的表添加主键，代码如下：

```
[root@vm01 ~]# pt-online-schema-change --alter="add primary key(id)" --user=root D=book , t=t1 --execute
```

Cannot connect to D=book , h=vm02.localdomain , u=root

Operation , tries , wait:

```
copy_rows , 10 , 0.25
create_triggers , 10 , 1
drop_triggers , 10 , 1
swap_tables , 10 , 1
update_foreign_keys , 10 , 1
```

Altering 'book'. 't1'...

Creating new table...

Created new table book.\_t1\_new OK.

Altering new table...

Altered 'book'. '\_t1\_new' OK.

Creating triggers...

Created triggers OK.

*Copying approximately 2 rows...*

*Copied rows OK.*

*Swapping tables...*

*Swapped original and new tables OK.*

*Dropping old table...*

*Dropped old table 'book'. '\_t1\_old' OK.*

*Dropping triggers...*

*Dropped triggers OK. Successfully altered 'book'. 't1'.*

结论：给无主键的表增加主键，可创建成功。

测试三：在slave上配置了replicate\_do\_table=book.t1，并在master上增加一个字段

回到slave机器上，查看同步复制信息，如下：

*mysql>show slave status\G;*

*\*\*\*\*\* 1. row \*\*\*\*\**

*slave\_IO\_State: Waiting for master to send event*

*master\_Host: 192.168.8.22*

*master\_User: repl*

*master\_Port: 3306*

*Connect\_Retry: 60*

*master\_Log\_File: mysql-bin.000002*

*Read\_master\_Log\_Pos: 6941*

*Relay\_Log\_File: vm02-relay-bin.000010*

*Relay\_Log\_Pos: 1841*

*Relay\_master\_Log\_File: mysql-bin.000002*

*slave\_IO\_Running: Yes*

*slave\_SQL\_Running: No*

*Replicate\_Do\_DB:*

*Replicate\_Ignore\_DB:*

*Replicate\_Do\_Table: book.t1*

*Replicate\_Ignore\_Table:*



```

Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
      Last_Errno: 1017
      Last_Error: Error 'Can't find file: './book/_t1_new.frm' (errno: 2)' on query. Default database: 'book'.
Query: 'RENAME TABLE 'book'. 't1' TO 'book'. '_t1_old', 'book'. '_t1_new' TO 'book'. 't1''
      Skip_Counter: 0
      Exec_Master_Log_Pos: 6154
      Relay_Log_Space: 2783
      Until_Condition: None
      Until_Log_File:
      Until_Log_Pos: 0
      master_SSL_Allowed: No
      master_SSL_CA_File:
      master_SSL_CA_Path:
      master_SSL_Cert:
      master_SSL_Cipher:
      master_SSL_Key:
      Seconds_Behind_Master: NULL
      master_SSL_Verify_Server_Cert: No
      Last_IO_Errno: 0
      Last_IO_Error:
      Last_SQL_Errno: 1017
      Last_SQL_Error: Error 'Can't find file: './book/_t1_new.frm' (errno: 2)' on query. Default database: 'book'.
Query: 'RENAME TABLE 'book'. 't1' TO 'book'. '_t1_old', 'book'. '_t1_new' TO 'book'. 't1''
      Replicate_Ignore_Server_Ids:
      master_Server_Id: 22

1 row in set (0.01 sec)
ERROR:
No query specified

```

结论：在主库上做的修改不会同步到slave上，因为配置了replicate\_do\_table=book.t1，将只会同步主库上的t1表，所以在主库上产生的



临时表并不会复制到从库上，在把临时表改名为原表时，导致主从服务器报错。

测试四：对有外键的表增加字段

以下是表结构信息：

父表：*mysql>show create table parent\G;*

\*\*\*\*\* 1. row \*\*\*\*\*

Table: parent

Create Table: CREATE TABLE 'parent' (

'id' int(11) NOT NULL DEFAULT '0',

'name' varchar(5) DEFAULT NULL,

PRIMARY KEY ('id')

) ENGINE=InnoDB DEFAULT CHARSET=utf8

1 row in set (0.03 sec)子表：

*mysql>show create table child\G;*

\*\*\*\*\* 1. row \*\*\*\*\*

Table: child

Create Table: CREATE TABLE 'child' (

'id' int(11) NOT NULL DEFAULT '0',

'name' varchar(5) DEFAULT NULL,

'pid' int(11) DEFAULT NULL,

PRIMARY KEY ('id'),

KEY 'fk\_pid' ('pid'),

CONSTRAINT 'fk\_pid' FOREIGN KEY ('pid') REFERENCES 'parent' ('id')

) ENGINE=InnoDB DEFAULT CHARSET=utf8

1 row in set (0.05 sec)

对父表增加字段：

*[root@vm01 ~]# pt-online-schema-change --alter-foreign-keys-method=auto --alter="add age tinyint(4)" --user=root D=book, t=parent --execute*

*Cannot connect to D=book, h=vm02.localdomain, u=root*

*Operation, tries, wait:*



*copy\_rows , 10 , 0.25*  
*create\_triggers , 10 , 1*  
*drop\_triggers , 10 , 1*  
*swap\_tables , 10 , 1*  
*update\_foreign\_keys , 10 , 1*

*Child tables:*

*'book'. 'child' (approx. 1 rows)*

*Will automatically choose the method to update foreign keys.*

*Altering 'book'. 'parent'...*

*Creating new table...*

*Created new table book.\_parent\_new OK.*

*Altering new table...*

*Altered 'book'. '\_parent\_new' OK.*

*Creating triggers...*

*Created triggers OK.*

*Copying approximately 1 rows...*

*Copied rows OK.*

*Max rows for the rebuild\_constraints method: 4000*

*Determining the method to update foreign keys...*

*'book'. 'child': 1 rows; can use rebuild\_constraints*

*Swapping tables...*

*Swapped original and new tables OK.*

*Rebuilding foreign key constraints...*

*Rebuilt foreign key constraints OK.*

*Dropping old table...*

*Dropped old table 'book'. '\_parent\_old' OK.*

*Dropping triggers...*

*Dropped triggers OK.*

*Successfully altered 'book'. 'parent'.*

结论：对有外键的表，在父表增加字段，创建成功。





对子表增加字段：

```
[root@vm01 ~]# pt-online-schema-change --alter-foreign-keys-method=auto --alter="add age tinyint(4)" --user=root D=book,  
t=child --execute
```

Cannot connect to D=book, h=vm02.localdomain, u=root

Operation, tries, wait:

copy\_rows, 10, 0.25

create\_triggers, 10, 1

drop\_triggers, 10, 1

swap\_tables, 10, 1

update\_foreign\_keys, 10, 1

No foreign keys reference 'book'. 'child'; ignoring --alter-foreign-keys-method.

Altering 'book'. 'child'...

Creating new table...

Created new table book.\_child\_new OK.

Altering new table...

Altered 'book'. '\_child\_new' OK.

Creating triggers...

Created triggers OK.

Copying approximately 1 rows...

Copied rows OK.

Swapping tables...

Swapped original and new tables OK.

Dropping old table...

Dropped old table 'book'. '\_child\_old' OK.

Dropping triggers...

Dropped triggers OK.

Successfully altered 'book'. 'child'.

结论：对有外键的表，在子表增加字段，创建成功。

测试五：字段含有not null不为空



对一个字段含有not null不为空进行测试，更改表结构，如下：

```
[root@vm01 ~]# pt-online-schema-change --alter="add address varchar(20) not null" --user=root D=book, t=t1 --execute
Cannot connect to D=book, h=vm02.localdomain, u=root
Operation, tries, wait:
  copy_rows, 10, 0.25
  create_triggers, 10, 1
  drop_triggers, 10, 1
  swap_tables, 10, 1
  update_foreign_keys, 10, 1
Altering 'book'. 't1'...
Creating new table...
Created new table book._t1_new OK.
Altering new table...
Altered 'book'. '_t1_new' OK.
Creating triggers...
Created triggers OK.
Copying approximately 3 rows...
Dropping triggers...
Dropped triggers OK.
Dropping new table...
Dropped new table OK.
'book'. 't1' was not altered.
Error copying rows from 'book'. 't1' to 'book'. '_t1_new': Copying rows caused a MySQL error 1364:
  Level: Warning
  Code: 1364
  Message: Field 'address' doesn't have a default value
  Query:  INSERT LOW_PRIORITY IGNORE INTO 'book'. '_t1_new' ('id', 'name', 'age', 'rmb') SELECT
'id', 'name', 'age', 'rmb' FROM 'book'. 't1' LOCK IN SHARE MODE /*pt-online-schema-change 3205 copy table*/
```

从报错信息来看，需要对非空字段增加一个默认值，我们再测试一下：

```
[root@vm01 ~]# pt-online-schema-change --alter="add address varchar(20) not null default 'china'" --user=root D=book, t=t1
```

--execute

Cannot connect to D=book, h=vm02.localdomain, u=root

Operation, tries, wait:

copy\_rows, 10, 0.25

create\_triggers, 10, 1

drop\_triggers, 10, 1

swap\_tables, 10, 1

update\_foreign\_keys, 10, 1

Altering 'book'. 't1'...

Creating new table...

Created new table book.\_t1\_new OK.

Altering new table...

Altered 'book'. '\_t1\_new' OK.

Creating triggers...

Created triggers OK.

Copying approximately 3 rows...

Copied rows OK.

Swapping tables...

Swapped original and new tables OK.

Dropping old table...

Dropped old table 'book'. '\_t1\_old' OK.

Dropping triggers...

Dropped triggers OK.

Successfully altered 'book'. 't1'.

结论：增加的字段为not null，不为空，该工具会报错，需要增加默认值才可以更改成功。

从上面的测试来看，pt-online-schema-change也存在着一些局限性，使用时要特别注意。另外，表的大小决定了pt-online-schema-change执行的快慢，所以一般要在凌晨0:00点以后执行，避开业务高峰期。由于该工具不是MySQL官方的，可能存在一定的风险，在操作之前，建议对数据表进行备份，可以使得操作更安全、可靠。





## 5.2.6 MySQL5.6在线DDL更改表测试

在MySQL5.6之前，InnoDB存储引擎表的许多DDL操作代价是非常昂贵的。许多ALTER TABLE操作的原理是通过创建新的空表，定义被要求的表选项和索引，然后逐行复制已存在的记录到新表中，并且是在插入行时更新索引。在旧表的所有行被复制完之后，旧表会被删除，新表将被重命名为旧的表名。相关命令如下：

```
create table tmp like t1
```

创建一个临时表。

```
insert into tmp select * from t1
```

一行行地把原表数据复制到临时表里，且更新索引。

```
drop table t1;rename table tmp to t1
```

删除原表，并把临时表改名为原表t1。

在这个过程中会对t1表加S锁（共享锁），所以这个代价是很高的。

MySQL5.5和MySQL5.1有了InnoDB Plugin，优化了CREATE INDEX和DROP INDEX，避免了表的复制行为。这个特性被称为Fast index Creation。MySQL5.6加强了ALTER TABLE操作来避免复制表。

这是MySQL5.6的一个重大特性，它参考了pt-online-schema-change的工作原理，之前在MySQL5.5里虽然增加、删除索引时不会锁表，但修改字段还是会锁表，而在MySQL5.6里进一步对其进行了优化，在某一会话中增加字段时，其他会话增、删、改、查均不会受影响，不会锁表。这些功能的组合现在被称为online DDL。

针对上述特性将进行如下测试，用Sysbench生成一张有1000万行内容的数据（数据大一些，可以看出效果来）。

```
sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=10000000 --max-requests=100 --num-threads=16  
--mysql-host=192.168.110.140 --mysql-port=3306 --mysql-user=admin --mysql-password=123456 --mysql-db=test --mysql-socket=/tmp/mysql.sock prepare
```

然后在会话一中执行如下命令：

```
alter table sbtest add name varchar(10) after pad;
```

在会话二中执行如下命令：

```
delete from sbtest where id=1000;
```



*insert into sbtest values(1000, 1, 'abc', 'abc', 'abc');*

*update sbtest set k=11 where id=1000;*

这时你会发现并没有锁表，各操作顺利执行完毕，如图5-51所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.10-log |
+-----+
1 row in set (0.74 sec)

mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host | db | Command | Time | State | Info |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | event_scheduler | localhost | NULL | Daemon | 4472 | Waiting on empty queue | NULL |
| 2 | root | localhost | test | Query | 27 | altering table | alter table sbtest add name varchar(10) |
| 3 | root | localhost | test | Query | 0 | init | show processlist |
+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

图5-51 未锁表状态信息

但在MySQL5.5里，这样的操作是会锁表的，如图5-52所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.02 sec)

mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host | db | Command | Time | State | Info |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 3 | agent | 192.168.110.140:60648 | NULL | Sleep | 14 | | NULL |
| 11 | root | localhost | test | Query | 27 | copy to tmp table | alter table sbtest add name varchar(10) after pad |
| 12 | root | localhost | test | Query | 15 | Waiting for table metadata lock | delete from sbtest where id=1000 |
| 13 | root | localhost | NULL | Query | 0 | NULL | show processlist |
+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

图5-52 锁表状态信息

可能会有人有疑问，MySQL5.6在线DDL就一定不锁表吗？我们再来一个测试，如下：

在会话一中执行如下命令：

```
select * from sbtest;
```

这里故意执行了一条大结果的查询，然后再执行删除操作，删除刚才增加的字段name：

```
alter table sbtest drop name;
```

从图5-53中可以看到，这时就会把表给锁了。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.10-log |
+-----+
1 row in set (0.87 sec)

mysql> show variables like '%iso%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tx_isolation  | READ-COMMITTED |
+-----+-----+
1 row in set (2.84 sec)

mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User          | Host          | db  | Command | Time | State                | Info |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | event_scheduler | localhost    | NULL | Daemon  | 50   | Waiting on empty queue | NULL |
| 2  | root           | localhost    | NULL | Query   | 0    | init                  | show processlist |
| 3  | root           | localhost    | test | Query   | 21   | Sending data          | select * from sbtest |
| 4  | root           | localhost    | test | Query   | 16   | Waiting for table metadata lock | alter table sbtest drop name |
+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.03 sec)

mysql>
```

图5-53 锁表状态信息

也就是说，在执行alter table表时，对该表的增、删、改、查均不会锁表。而如果在这之前，该表有被访问，那需要等其执行完毕后，才可以执行alter table，否则会存在锁表现象。所以在凌晨上线时，一定要观察一下，此时此刻是否有某个慢SQL在对该表进行操作，以免改表时出现锁等待现象。

关于在线DDL，请参考手册：



## 5.5.1. Overview of Online DDL

Historically, many **DDL** operations on **InnoDB** tables were expensive. Many **ALTER TABLE** operations worked by creating a new, empty table defined with the requested table options and indexes, then copying the existing rows to the new table one-by-one, updating the indexes as the rows were inserted. After all rows from the original table were copied, the old table was dropped and the copy was renamed with the name of the original table.

MySQL 5.5, and MySQL 5.1 with the InnoDB Plugin, optimized **CREATE INDEX** and **DROP INDEX** to avoid the table-copying behavior. That feature was known as Fast Index Creation. MySQL 5.6 enhances many other types of **ALTER TABLE** operations to avoid copying the table. Another enhancement allows **SELECT** queries and **INSERT**, **UPDATE**, and **DELETE (DML)** statements to proceed while the table is being altered. This combination of features is now known as **online DDL**.

## 5.3 采用合适的锁机制

MySQL的锁有以下几种形式：

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。MyISAM引擎属于这种类型。
- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。InnoDB引擎属于这种类型。
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。NDB属于这种类型。



### 5.3.1 表锁的演示

MyISAM存储引擎只支持表锁，所以对MyISAM表进行操作，会存在以下情况：

- 对MyISAM表的读操作（加读锁），不会阻塞其他进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其他进程的写操作。
- 对MyISAM表的写操作（加写锁），会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其他进程的读写操作。

下面来演示一下表锁的读取与释放操作：



Session1	Session2
<pre>mysql&gt; show create table t2\G; ***** 1. row *****  Table: t2 Create Table: CREATE TABLE 't2' (   'id' tinyint(3) unsigned NOT NULL AUTO_INCREMENT,   'name' varchar(10) NOT NULL,   PRIMARY KEY ('id') ) ENGINE=MyISAM AUTO_INCREMENT=8 DEFAULT CHARSET=gbk 1 row in set (0.00 sec)</pre>	
<pre>mysql&gt; use test; Database changed mysql&gt; lock table t2 read; Query OK, 0 rows affected (0.00 sec) 对 t2 表加读锁</pre>	
<pre>mysql&gt; select * from t2; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.00 sec)</pre>	<pre>mysql&gt; select * from t2; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.00 sec)</pre>

(续)

Session1	Session2																																								
	mysql> update t2 set name='g1' where id=7; 此时就会锁等待，等待会话一锁的释放																																								
mysql> show processlist; <table><tr><th>Id</th><th>User</th><th>Host</th><th>db</th><th>Command</th><th>Time</th><th>State</th><th>Info</th></tr><tr><td>1</td><td>admin</td><td>192.168.8.1:3269</td><td>test</td><td>Sleep</td><td>535</td><td>NULL</td><td></td></tr><tr><td>2</td><td>admin</td><td>192.168.8.1:3271</td><td>NULL</td><td>Sleep</td><td>552</td><td>NULL</td><td></td></tr><tr><td>4</td><td>root</td><td>localhost</td><td>test</td><td>Query</td><td>0</td><td>NULL</td><td>show processlist</td></tr><tr><td>5</td><td>root</td><td>localhost</td><td>test</td><td>Query</td><td>57</td><td>Waiting for table level lock</td><td>update t2 set name='g1' where id=7</td></tr></table> 4 rows in set (0.00 sec)	Id	User	Host	db	Command	Time	State	Info	1	admin	192.168.8.1:3269	test	Sleep	535	NULL		2	admin	192.168.8.1:3271	NULL	Sleep	552	NULL		4	root	localhost	test	Query	0	NULL	show processlist	5	root	localhost	test	Query	57	Waiting for table level lock	update t2 set name='g1' where id=7	
Id	User	Host	db	Command	Time	State	Info																																		
1	admin	192.168.8.1:3269	test	Sleep	535	NULL																																			
2	admin	192.168.8.1:3271	NULL	Sleep	552	NULL																																			
4	root	localhost	test	Query	0	NULL	show processlist																																		
5	root	localhost	test	Query	57	Waiting for table level lock	update t2 set name='g1' where id=7																																		
mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)																																									
	mysql> update t2 set name='g1' where id=7; Query OK, 1 row affected (27 min 24.11 sec) Rows matched: 1 Changed: 1 Warnings: 0 锁释放后，顺利更新																																								

### 5.3.2 行锁的演示

InnoDB存储引擎是通过给索引上的索引项加锁来实现的，这就意味着：只有通过索引条件检索数据，InnoDB才会使用行级锁，否则，InnoDB将使用表锁。

#### 1. 行锁

下面来演示下行锁的使用情况。



Session1	Session2
<pre>mysql&gt; show create table t\G; ***** 1. row *****        Table: t Create Table: CREATE TABLE 't' (   'id' tinyint(3) unsigned NOT NULL AUTO_INCREMENT,   'name' varchar(10) NOT NULL,   PRIMARY KEY ('id') ) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=gbk 1 row in set (0.00 sec)</pre>	
<pre>mysql&gt; begin; Query OK, 0 rows affected (0.00 sec)</pre>	<pre>mysql&gt; begin; Query OK, 0 rows affected (0.00 sec)</pre>
<pre>mysql&gt; select * from t; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.01 sec)</pre>	<pre>mysql&gt; select * from t; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.01 sec)</pre>
<pre>mysql&gt; update t set name='d1' where id=4; Query OK, 1 row affected (0.03 sec) Rows matched: 1  Changed: 1  Warnings: 0</pre>	
	<pre>mysql&gt; update t set name='d2' where id=4; 此时就会锁等待，因为更新的是同一行记录</pre>
	<pre>mysql&gt; update t set name='e1' where id=5; Query OK, 1 row affected (0.01 sec) Rows matched: 1  Changed: 1  Warnings: 0 而更新下一行记录时，就没事，原因就是 innodb 引擎是行锁，不是表锁</pre>

在并发访问比较高的情况下，如果大量事务因无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重的性能问题，甚至拖垮数据库，这时需要通过设置合适的锁等待超时阈值参数`innodb_lock_wait_timeout`来解决，一般设置为100秒即可。

## 2.行锁转表锁

下面来演示一下行锁转表锁的情况。

Session1	Session2
<pre>mysql&gt; show create table t1\G; ***** 1. row *****  Table: t1 Create Table: CREATE TABLE 't1' (   'id' tinyint(3) unsigned NOT NULL DEFAULT '0',   'name' varchar(10) NOT NULL ) ENGINE=InnoDB DEFAULT CHARSET=gbk 1 row in set (0.00 sec)</pre>	
<pre>mysql&gt; begin; Query OK, 0 rows affected (0.00 sec)</pre>	<pre>mysql&gt; begin; Query OK, 0 rows affected (0.00 sec)</pre>
<pre>mysql&gt; select * from t; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.01 sec)</pre>	<pre>mysql&gt; select * from t; +----+-----+   id   name   +----+-----+   1   a        2   b        3   c        4   d        5   e        6   f        7   g      +----+-----+ 7 rows in set (0.01 sec)</pre>
<pre>mysql&gt; update t1 set name='d1' where id=4; Query OK, 1 row affected (0.01 sec) Rows matched: 1 Changed: 1 Warnings: 0</pre>	
	<pre>mysql&gt; update t1 set name='e1' where id=5; ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction 此时就会锁等待，因为 t1 加上了表锁</pre>



从上面演示的内容可以看出，只有通过索引条件检索数据，InnoDB才会使用行级锁，否则，InnoDB将使用表锁。

### 3.死锁

两个事务都需要获得对方持有的排他锁才能继续完成事务，这种循环锁等待就是典型的死锁。

下面来演示一下死锁的情况。



Session1	Session2
<pre>mysql&gt; select * from t2; +----+-----+   id   name   +----+-----+   1   a         2   b         3   c         4   d         5   e         6   f       +----+-----+ 6 rows in set (0.48 sec)</pre>	<pre>mysql&gt; select * from t2; +----+-----+   id   name   +----+-----+   1   a         2   b         3   c         4   d         5   e         6   f       +----+-----+ 6 rows in set (0.48 sec)</pre>
<pre>mysql&gt; begin; Query OK, 0 rows affected (0.02 sec)</pre>	<pre>mysql&gt; begin; Query OK, 0 rows affected (0.02 sec)</pre>
<pre>mysql&gt; update t2 set name='bb' where id=2; Query OK, 1 row affected (0.25 sec) Rows matched: 1  Changed: 1  Warnings: 0</pre>	
	<pre>mysql&gt; update t2 set name='cc' where id=3; Query OK, 1 row affected (0.07 sec) Rows matched: 1  Changed: 1  Warnings: 0</pre>
<pre>mysql&gt; update t2 set name='cc1' where id=3; (等待 ...)</pre>	
	<pre>mysql&gt; update t2 set name='bb1' where id=2; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction</pre>
<pre>mysql&gt; update t2 set name='cc1' where id=3; Query OK, 1 row affected (6.75 sec) Rows matched: 1  Changed: 1  Warnings: 0</pre>	

发生死锁后，InnoDB一般都能自动检测到，它会让一个事务释放锁并回退，另一个事务则获得锁，继续完成事务。死锁是无法避免的，我们可以通过调整业务的逻辑来尽量减少死锁出现的概率。





### 5.3.3 InnoDB引擎与MyISAM引擎的性能对比

InnoDB和MyISAM作为MySQL数据库中两种最主要、最常用的存储引擎，各有所长。在MySQL5.5之前的版本中，MyISAM是MySQL中默认的存储引擎，而在MySQL5.5之后，MySQL中默认的存储引擎则改为了InnoDB。对于这两种存储引擎的选择，要根据项目应用特点来权衡，而对于复杂的应用系统，也可以根据实际情况来选择多种存储引擎的组合。不过，还是建议尽量不要混合使用多种存储引擎，这样容易带来更复杂的问题。

MyISAM支持全文索引，这是一种基于分词创建的索引，支持一些比较复杂的查询，但不是事务安全的，而且不支持外键。每张MyISAM表存放在3个文件中：frm文件存放表格定义；数据文件是MYD（MYData）；索引文件是MYI（MYIndex）。对于MyISAM表，可以手工或者自动执行检查或修复操作，这一点要注意跟InnoDB的事物恢复区分开来。

InnoDB是事务型引擎，支持回滚，具有崩溃恢复能力，多版本并发控制（MVCC）、支持ACID事务、支持行级锁定（InnoDB表的行锁不是绝对的，如果执行一个SQL语句没有使用到索引，InnoDB表同样会锁全表）。

InnoDB的工作原理：就是把数据捞到内存中，被用户读写，这样大大增加了性能，因为从内存进行读写比磁盘要快得多。当数据全部加载到内存中，这时的性能是最好的。它的设计理论是充分利用内存，减少磁盘I/O使用率，每次版本升级时，改善最多的就是这些方面。它有两种表空间管理模式：一种是共享表空间，一种是独立表空间。共享表空间的数据文件是ibdata1...ibdataN，这个数据文件里保存着元数据、数据、索引、插入合并缓冲、undo log回滚日志，ib\_logfile0...3里保存着redo log重做事务日志。独立表空间把数据、索引、插入合并缓冲从ibdata1拆分了出去，保存在.ibd这种格式的文件里，因为在MySQL5.5里，一些新的特性都是基于独立表空间的。在MySQL5.6里，undo log回滚日志也可以从共享表空间拆分出去，保存在另一块硬盘上，这样做的目的是，充分利用多块磁盘的I/O，比如，可以把undo log放入SSD磁盘上。

MyISAM和InnoDB之间的主要区别有以下几点：

- MyISAM是非事务安全型的，而InnoDB是事务安全型的，也就是ACID事务支持；
- MyISAM锁是表级锁，锁开销最小，而InnoDB支持行级锁定，锁管理开销大，支持更好的并发写操作；
- MyISAM支持全文索引，而InnoDB不支持全文索引，但在最新的5.6版本中已提供支持；
- MyISAM相对简单，管理方便，因此在效率上要优于InnoDB，小型应用可以考虑使用MyISAM；
- MyISAM表是保存成文件的形式，在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦；
- InnoDB表比MyISAM表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表。

这两个存储引擎在第1章已有介绍，篇幅所限，这里就不再多说了。下面将使用Sysbench对这两个引擎进行压力测试，让大家进一步认识



两者的特性。

首先来看看MySQL配置参数：

*innodb\_buffer\_pool\_size=11G*

*sync\_binlog=0*

*innodb\_flush\_log\_at\_trx\_commit = 2*

下面是压力测试参数：

*sysbench --test=oltp --mysql-table-engine=innodb/myisam*

*--oltp-table-size=1000000*

*--max-requests=10000*

*--num-threads=100*

*--mysql-host=192.168.1.11*

*--mysql-port=3306*

*--mysql-user=admin*

*--mysql-password=admin123*

*--mysql-db=test*

*--mysql-socket=/tmp/mysql.sock run*

下面是二者之间的性能对比情况。

InnoDB	MyISAM
OLTP test statistics:	OLTP test statistics:
queries performed:	queries performed:
read: 140014	read: 140000
write: 50005	write: 50000
other: 20002	other: 20000
total: 210021	total: 210000
transactions: 10001 (1549.15 per sec.)	transactions: 10000 (154.54 per sec.)
deadlocks: 0 (0.00 per sec.)	deadlocks: 0 (0.00 per sec.)
read/write requests: 190019 (29433.80 per sec.)	read/write requests: 190000 (2936.22 per sec.)
other operations: 20002 (3098.29 per sec.)	other operations: 20000 (309.08 per sec.)

从测试的结果来看，InnoDB每秒处理的数据为1549个，而MyISAM仅仅为154个。

MyISAM存储引擎的读锁和写锁是互斥的，读写操作是串行的。试想一下，一个进程请求某个MyISAM表的读锁，同时另一个进程也请求同一表的写锁，MySQL该如何处理呢？答案是写进程先获得锁。不仅如此，即使读请求先到锁等待队列，写请求后到，写锁也会插到读锁请求之前！这是因为MySQL认为写请求一般要比读请求重要。这也正是MyISAM表不太适合有大量更新操作和查询操作应用的原因，因为大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞。

InnoDB用于事务处理应用程序，具有众多特性，包括支持ACID事务、行锁等。如果应用中需要执行大量的读写操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。对于MyISAM引擎，在MySQL5.5版本里Oracle公司支持的已经很少了，以后内存数据库是一种趋势，所以建议优先选择InnoDB引擎。



## 5.4 选择合适的事务隔离级别

### 5.4.1 事务的概念

事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。一个逻辑工作单元要成为事务，必须满足所谓的ACID(原子性、一致性、隔离性和持久性)属性：

#### ·原子性

对于数据修改，要么全都执行，要么全都不执行。

#### ·隔离性

在所有的操作没有执行完毕之前，其他会话不能够看到中间改变的过程。

#### ·一致性

事务发生前和发生后，根据数据的规则，总额应该匹配。

#### ·持久性

事务发生前和发生后，根据数据的规则，总额应该匹配。

为了帮助大家进一步理解事务的含义，这里将用一个例子来说明，相信会比专业术语看起来容易明白得多。大家都在ATM取款机取过钱吧？比如，我一次取出2000元，当我按确定等钱出来的时候，取款机主板烧了，此时如果数据库没有保障机制，那么我就损失掉了2000元，所以为了避免这种问题的出现，在我取钱的一刹那，若取款机坏掉了，我这个取钱的动作会自动回滚到先前的状态，保证我不会亏损，那么这个就叫作事务。要么就成功，要么就失败。



## 5.4.2 事务的实现

像其他数据库一样，MySQL在进行事务处理的时候使用的是日志先行的方式来保证事务可快速和持久运行的，也就是在写数据前，需要先写日志。当开始一个事务时，会记录该事务的一个LSN日志序列号；当执行事务时，会往InnoDB\_Log\_Buffer日志缓冲区里插入事务日志（redo log）；当事务提交时，会将日志缓冲区里的事务日志刷入磁盘。这个动作是由innodb\_flush\_log\_at\_trx\_commit这个参数控制的。

innodb\_flush\_log\_at\_trx\_commit=0，表示每个事务提交时，每隔一秒，把事务日志缓存区的数据写到日志文件中，以及把日志文件的数据刷新到磁盘上；它的性能是最好的，同样安全性也是最差的。当系统宕机时，会丢失1秒钟的数据。

innodb\_flush\_log\_at\_trx\_commit=1，表示每个事务提交时，把事务日志从缓存区写到日志文件中，并且刷新日志文件的数据到磁盘上。

innodb\_flush\_log\_at\_trx\_commit=2，表示每个事务提交时，把事务日志数据从缓存区写到日志文件中；每隔一秒，刷新一次日志文件，但不一定刷新到磁盘上，而是取决于操作系统的调度。

图5-54~图5-56是redo log事务日志刷新到磁盘的示意图。

使用命令“show innodb status\G;”可以看到当前刷新事务日志的情况：

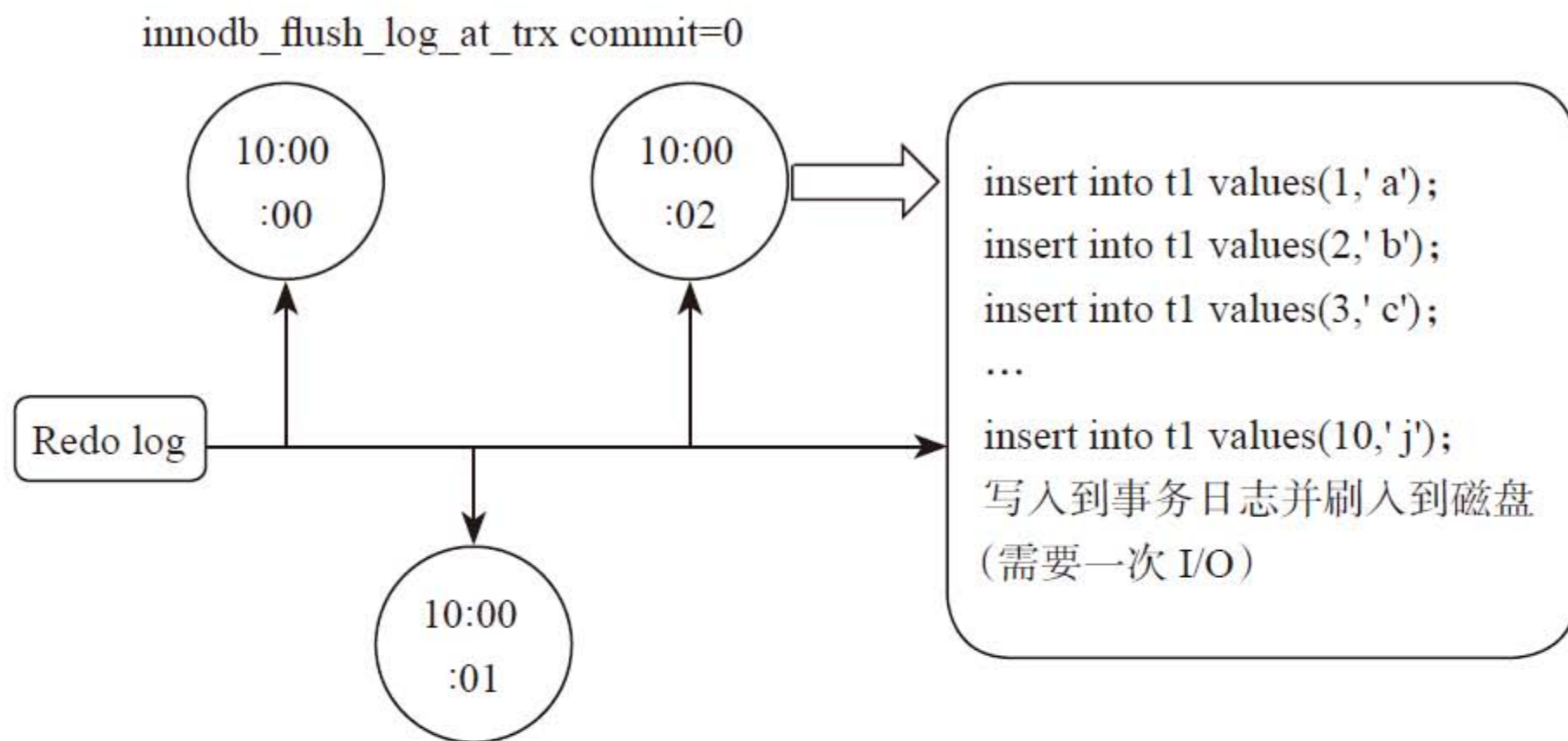


图5-54 innodb\_flush\_log\_at\_trx\_commit=0



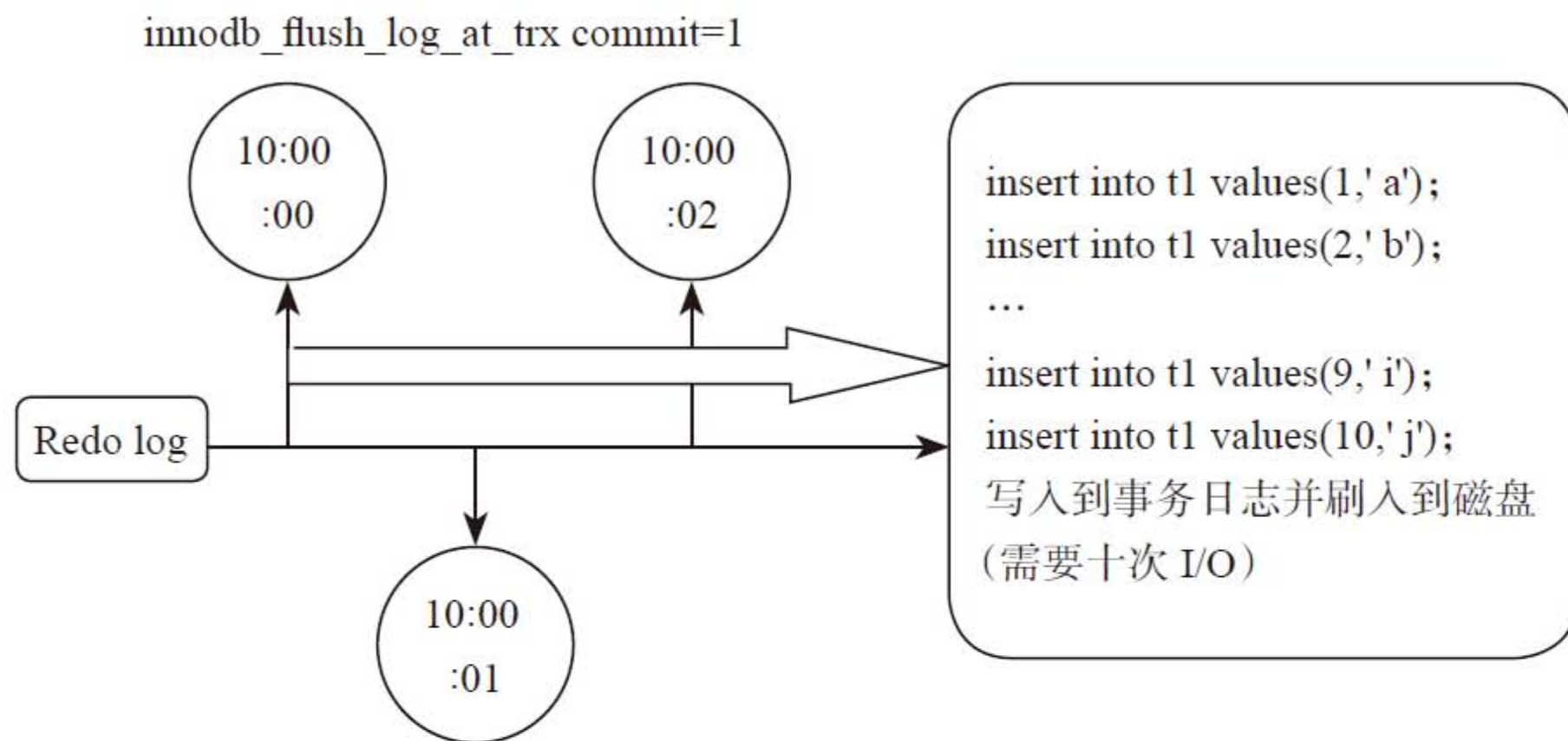


图5-55 innodb\_flush\_log\_at\_trx\_commit=1

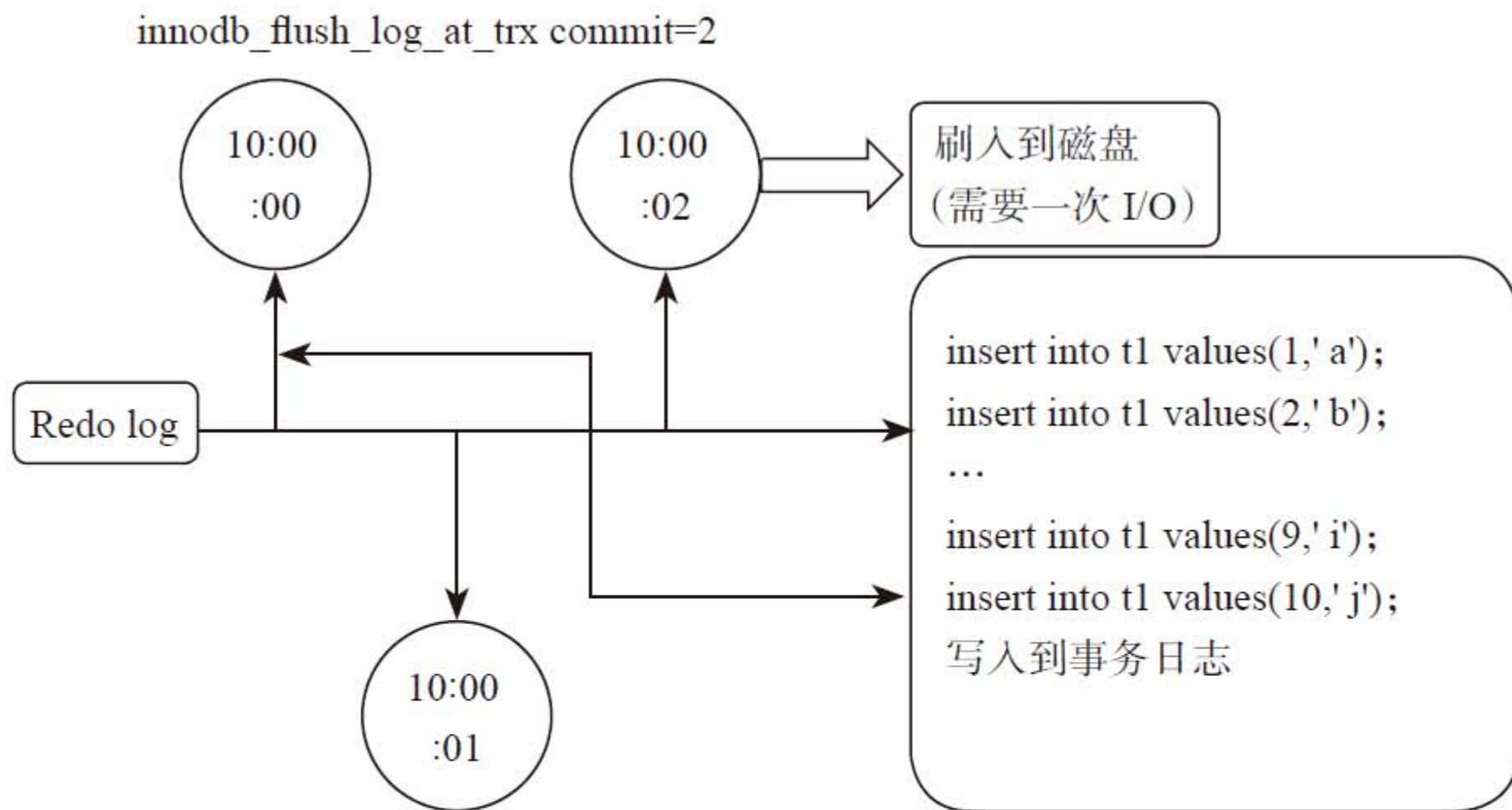


图5-56 innodb\_flush\_log\_at\_trx\_commit=2

## LOG

Log sequence number 184 3577056574 (表示当前的LSN日志序列号)  
 Log flushed up to 184 3576959795 (表示刷新到事务日志的LSN日志序列号)  
 Last checkpoint at 184 3535957301 (表示刷新到磁盘的LSN日志序列号)

除了记录事务日志以外，数据库还会记录一定量的撤销日志 (undo log)，undo与redo正好相反，在对数据进行修改时，由于某种原因失败了，或者人为执行了rollback回滚语句，就可以利用这些撤销日志将数据回滚到修改之前的样子，就如前面举的那个ATM取款机取钱的例子。redo日志保存在ib\_logfile0/1/2里，而undo日志保存在ibdata1里，在MySQL5.6里还可以把undo日志单拆分出去。

在默认情况下，事务都是自动提交的。如果想在程序里自己控制事务，那么开始一个事务前，要写明BEGIN或START TRANSACTION，执

行完以后再COMMIT提交。

比如，下面这个例子：

```
begin;  
insert into t1 values (1, '张三');  
commit;  
begin;  
update t1 set name='李四' where id=1;  
rollback;
```

事务提交以后，首先刷新到binlog，然后再刷新到redo log里。如果在刷新到binlog的时候发生宕机，那么MySQL数据库在下次启动时，由于redo log里没有该事务记录就会回滚，但是二进制日志已经记录了该事务信息，不能被回滚。所以另一端的slave会复制master上的binlog，这样主从数据就会出现不一致，XA事务可以保证InnoDB redo log与MySQL binlog的一致性。可以将innodb\_support\_xa设置为1，但会对性能产生影响。



### 5.4.3 事务隔离级别介绍

在数据库操作中，为了有效保证并发读取数据的正确性，提出了事务隔离级别的概念。InnoDB有四种隔离级别：Read Uncommitted、Read Committed、Repeatable Read、Serializable。ORACLE/SQL SERVER的默认隔离级别是Read Committed（读提交），而MySQL的默认隔离级别是Repeatable Read（可重复读）。

数据库是要被广大客户所共享访问的，那么在数据库操作过程中很可能出现以下几种不确定情况。

·更新丢失（Lost update）：两个事务都同时更新一行数据，但是第二个事务却中途失败退出，导致对数据的两个修改都失效了。这是因为系统没有执行任何的锁操作，因此并发事务并没有被隔离开来。

·脏读（Dirty Reads）：一个事务开始读取了某行数据，另外一个事务已经更新了此数据但没有能够及时提交。这是相当危险的，因为很可能所有的操作都被回滚。

·不可重复读（Non-repeatable Reads）：一个事务对同一行数据重复读取两次，但是却得到了不同的结果。例如，在两次读取的中途，有另外一个事务对该行数据进行了修改，并提交。

·两次更新问题（Second lost updates problem）：无法重复读取的特例。有两个并发事务同时读取同一行数据，然后其中一个对它进行修改提交，而另一个也进行了修改提交。这就会造成第一次写操作失效。

·幻读（Phantom Reads）：事务在操作过程中进行两次查询，第二次查询的结果包含了第一次查询中未出现的数据（这里并不要求两次查询的SQL语句相同）。这是因为在两次查询过程中有另外一个事务插入数据。

为了避免出现上面几种情况，在标准SQL规范中，定义了4个事务隔离级别，不同的隔离级别对事务的处理不同。如下所示：

·未授权读取，也称为读未提交（Read Uncommitted）：允许脏读取，但不允许更新丢失。如果一个事务已经开始写数据，则另外一个数据则不允许同时进行写操作，但允许其他事务读此行数据。该隔离级别可以通过“排他写锁”实现。

·授权读取，也称为读提交（Read Committed）：允许不可重复读取，但不允许脏读取。这可以通过“瞬间共享读锁”和“排他写锁”实现。读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。

·可重复读取（Repeatable Read）：禁止不可重复读取和脏读取，但是有时可能出现幻影数据。这可以通过“共享读锁”和“排他写锁”实现。读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。

·序列化（Serializable）：提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。

通过表5-9会让大家理解得更清晰一些。

表5-9 事务隔离级别

隔离级别	读数据一致性	脏读	不可重复读	幻读
未提交读 (Read uncommitted)	最低级别，只能保证不读取物理上损坏的数据	是	是	是
已提交度 (Read committed)	语句级	否	是	是
可重复读 (Repeatable read)	事务级	否	否	是
可序列化 (Serializable)	最高级别，事务级	否	否	否

下面用表格形式记录了Repeatable-Read可重复读和Read-committed已提交读二者之间的区别。

### 1.Repeatable-Read的演示

开启两个会话端，以下是操作过程：



Session1	Session2
mysql> show variables like '%iso%'; +-----+   Variable_name   Value   +-----+   tx_isolation   REPEATABLE-READ   +-----+ 1 row in set (0.09 sec)	mysql> show variables like '%iso%'; +-----+   Variable_name   Value   +-----+   tx_isolation   REPEATABLE-READ   +-----+ 1 row in set (0.09 sec)
mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     8   +----+ 8 rows in set (0.52 sec)	mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     8   +----+ 8 rows in set (0.52 sec)
START TRANSACTION;	START TRANSACTION;
	mysql> update t1 set id=88 where id=8; Query OK, 1 row affected (0.31 sec) Rows matched: 1 Changed: 1 Warnings: 0
	mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     88   +----+ 8 rows in set (0.04 sec)
	commit;
mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     8   +----+ 8 rows in set (0.05 sec)	



(续)

Session1	Session2
commit;	
mysql> select * from t1;	
+----+	
id	
+----+	
1	
2	
3	
4	
5	
6	
7	
88	
+----+	
8 rows in set (0.03 sec)	

Session2提交后，Session1仍看到的是先前的数据，只有在Session1也提交了，才能看到新的数据，所以这个隔离级别就叫作Repeatable Read（可重复读）。

## 2.Read-Committed的演示

开启两个会话端，以下是操作过程：

Session1	Session2
<pre> Mysql&gt; show variables like '%iso%'; +-----+-----+   Variable_name   Value                  +-----+-----+   tx_isolation    READ-COMMITTED        +-----+-----+ 1 row in set (0.18 sec) </pre>	<pre> mysql&gt; show variables like '%iso%'; +-----+-----+   Variable_name   Value                  +-----+-----+   tx_isolation    READ-COMMITTED        +-----+-----+ 1 row in set (0.18 sec) </pre>
<pre> mysql&gt; select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     88   +----+ 8 rows in set (0.13 sec) </pre>	<pre> mysql&gt; select * from t1; +----+   id   +----+   1     2     3     4     5     6     7     88   +----+ 8 rows in set (0.13 sec) </pre>
START TRANSACTION;	START TRANSACTION;
	<pre> mysql&gt; update t1 set id=77 where id=7; Query OK, 1 row affected (0.23 sec) Rows matched: 1  Changed: 1  Warnings: 0 </pre>

(续)

Session1	Session2
	mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     77     88   +----+ 8 rows in set (0.02 sec)
	commit;
mysql> select * from t1; +----+   id   +----+   1     2     3     4     5     6     77     88   +----+ 8 rows in set (0.02 sec)	





Session2提交后，Session1看到的是新的数据，所以这个隔离级别就叫作Read Committed（读提交）。

3.间隙锁的演示

间隙锁主要是防止幻象读，用在REPEATABLE-READ隔离级别下，指的是当对数据进行条件、范围检索时，对其范围内也许并不存在的值进行加锁。在READ-COMMITTED隔离级别下，没有间隙锁。间隙锁对高并发访问的业务有较大的性能影响。

开启两个会话端，以下是操作过程：

Session1	Session2								
<pre>mysql&gt; show variables like '%iso%';</pre> <table><tr><td>Variable_name</td><td>Value</td></tr><tr><td>tx_isolation</td><td>REPEATABLE-READ</td></tr></table>	Variable_name	Value	tx_isolation	REPEATABLE-READ	<pre>mysql&gt; show variables like '%iso%';</pre> <table><tr><td>Variable_name</td><td>Value</td></tr><tr><td>tx_isolation</td><td>REPEATABLE-READ</td></tr></table>	Variable_name	Value	tx_isolation	REPEATABLE-READ
Variable_name	Value								
tx_isolation	REPEATABLE-READ								
Variable_name	Value								
tx_isolation	REPEATABLE-READ								

(续)

Session1	Session2
begin;	begin;
mysql> select * from t2 where id <8 lock in share mode; +----+   id   +----+   1     2     3     7   +----+ 4 rows in set (0.00 sec)	
	mysql> insert into t2 values(5); ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
	mysql> insert into t2 values(22); Query OK, 1 row affected (0.00 sec)

在上面的示例中，由于会话1的锁定了一个范围（小于8），那么会话2再向其插入小于8的值时，就会被锁住，但大于8的值，是可以插入的，只有默认隔离级别可重复读才会有间隙锁，READ-COMMITTED不会有间隙锁。

下面对READ-COMMITTED再重新进行上面的测试：

Session1	Session2
<pre>mysql&gt; show variables like '%iso%'; +-----+-----+   Variable_name   Value                  +-----+-----+   tx_isolation    READ-COMMITTED        +-----+-----+ 1 row in set (0.00 sec)</pre>	<pre>mysql&gt; show variables like '%iso%'; +-----+-----+   Variable_name   Value                  +-----+-----+   tx_isolation    READ-COMMITTED        +-----+-----+ 1 row in set (0.00 sec)</pre>
begin;	begin;
<pre>mysql&gt; select * from t2 where id &lt;8 lock in share mode; +----+   id   +----+   1      2      3      7    +----+ 4 rows in set (0.00 sec)</pre>	
	<pre>mysql&gt; insert into t2 values(4); Query OK, 1 row affected (0.00 sec) mysql&gt; insert into t2 values(5); Query OK, 1 row affected (0.00 sec)</pre>

可以看到隔离级别改成提交读以后，间隙锁就失效了，可以插入小于8的值。

#### 4.小结

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。对于多数应用程序来说，可以优先考虑把数据库系统的



隔离级别设为Read Committed，它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、虚读和第二类丢失更新这些并发问题，一般来说，还是可以接受的，因为读到的是已经提交的数据，本身并不会带来很大的问题。Oracle、SQL Server默认的隔离级别都是设置为Read Committed的。



## 5.5 SQL优化与合理利用索引

在应用系统的开发初期，由于数据库的数据比较少，在查询SQL语句，复杂视图的编写等方面体会不出SQL语句各种写法的性能优劣情况，但是如果将应用系统提交实际应用后，随着数据库中数据的增加，系统的响应速度就会成为需要解决的最主要问题之一。系统优化中一个很重要的方面就是SQL语句的优化。对于海量数据，劣质SQL语句和优质SQL语句之间的速度差别可以达到上百倍，可见，对于一个系统来说，不是简单地实现其功能就可以了，而是要写出高质量的SQL语句，提高系统的可用性。

### 5.5.1 如何定位执行很慢的SQL语句

数据库上线以后，多多少少会遇到一些问题，比如，常见的慢查询等，对每个运维人员或DBA来说，这也许就是日常工作中的重点，也是难点。在这一节主要讲解一下MySQL的慢日志对系统性能的影响和作用，并说明怎么去定位慢SQL，并根据日志的相关特性得出相应的优化思路。

慢日志带来的直接性能损耗就是数据库系统中最为昂贵的I/O资源。对于MySQL数据库来说，I/O出现瓶颈，会导致连接数增大、锁表，更有可能导致业务访问失败，尤其是在高并发场合下。开启慢查询记录功能带来的好处是可以通过分析慢SQL来优化SQL语句，从而解决因慢SQL引起的各种问题。

开启慢日志功能很简单，在my.cnf配置文件里，加入以下参数：

```
slow_query_log = 1
slow_query_log_file = mysql.slow
long_query_time = 2 (超过2秒的SQL会记录下来)
```

当你的数据库连接数很高时，此时就要注意了，可通过cacti监控软件观察时间点，然后把这一时间段的慢日志截取出来，如：

```
sed -n '/# Time: 110720 16:17:39/, /end/p' mysql.slow > slow.log
```

然后用mysqldumpslow命令取出耗时最长的前10条慢SQL进行分析。

```
mysqldumpslow -s t -t 10 slow.log
```



### 5.5.2 SQL优化案例分析

和设计与调整数据库一样，优化执行SQL语句可以提高应用程序的性能。如果不遵循一些基本的原则，那么无论数据库结构设计得如何合理，无论数据库调整得如何好，也不会得到令用户满意的查询结果。对于SQL查询，应明确要完成的目标，并努力使查询效率最高，以最少的时间准确地检索数据。如果最终用户得到的是一个低速的查询，就好比饥饿者不耐烦地等待迟迟不到的饭菜。大多数查询可有多种方式来完成，不过，查询方式的不同将会导致同一查询执行时间为几秒钟、几分钟甚至是几个小时，所以选择何种查询方式很重要。

我们可通过对MySQL慢日志的监控，找出SQL语句运行慢的主要问题，然后对这些SQL进行优化。这里列出一些SQL语句优化的原则和方法，供参考。

优化的理由：

- SQL语句是对数据库（数据）进行操作的唯一途径；
- SQL语句消耗了70%~90%的数据库资源；
- SQL语句独立于程序设计逻辑，相对于对程序源代码的优化，对SQL语句的优化在时间成本和风险上的代价都很低；
- SQL语句可以有不同的写法；

#### 1.not in子查询优化

下面来看看对not in子查询优化的案例。

案例一：

在将子查询SQL改为JOIN表连接SQL时，子查询性能很差，下面是一个测试：

```
mysql> select SQL_NO_CACHE count(*) from test1 where id not in(select id from test2);
```

```
+-----+
```

```
/ count(*) /
```

```
+-----+
```

```
/ 215203 /
```

```
+-----+
```

```
1 row in set (5.81 sec)
```

```
mysql> select SQL_NO_CACHE count(*) from test1 where not exists (select * from test2 where test2.id=test1.id);
```

```
+-----+
```

```

/ count(*) /
+-----+
/ 215203 /
+-----+
1 row in set (5.25 sec)
mysql>select SQL_NO_CACHE count(*) from test1 left join test2 on test1.id=test2.id where test2.id is null;
+-----+
/ count(*) /
+-----+
/ 215203 /
+-----+
1 row in set (4.63 sec)

```

在生产环境里，应尽量避免使用子查询，可用left join表连接取代之。在前面故障处理一章也有介绍，这里不再多说。

#### 案例二：mysql error 1093解决方法

通过子查询删除已查询的记录时会报错，如下：

```

mysql>delete from t where id in (select id from t where id <5);
ERROR 1093 (HY000): You can't specify target table 't' for update in FROM clause
mysql>

```

对于这种报错情况，更改SQL语句为：

```

mysql>delete from t where id in (select * from (select id from t where id <5) tmp);
Query OK, 4 rows affected (0.00 sec)

```

以这样的形式即可删除。再优化之，改为表连接模式：

```

mysql>delete t from t join (select id from t where id <5) tmp on t.id=tmp.id;
Query OK, 4 rows affected (0.01 sec)

```

#### 2.模式匹配like'%xxx%'优化

在MySQL里，like'xxx%'可以用到索引，但like'%xxx%'却不行。比如，下面这个例子（如图5-57所示）：



```
mysql> explain select * from artist where name like '%Queen%';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | artist | ALL | NULL | NULL | NULL | NULL | 589410 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.02 sec)
```

图5-57 全表扫描

那么这种情况下要如何处理呢，可通过覆盖索引来进一步优化，如图5-58所示。

```
mysql> explain select artist_id from artist where name like '%Queen%';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | artist | index | NULL | name | 257 | NULL | 589410 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.03 sec)
```

图5-58 覆盖索引

这里artist\_id是主键（聚集索引），叶子节点上保存了数据（InnoDB引擎），从索引中就能够取得select的artist\_id列，不必读取数据行（只要你的SELECT字段正好就是索引，那么就用到了覆盖索引），通过覆盖索引，可以减少I/O，提高性能。简单地说，覆盖索引就是：我要在书里查找一个内容，由于目录写得很详细，我在目录中就获取到了，不需要再翻到该页查看。

优化完成之后，我们对SQL的执行时间进行一下对比，图5-59是优化之前的SQL执行时间。

```
mysql> select count(*) from artist where name like '%Queen%';
+-----+
| count(*) |
+-----+
| 280 |
+-----+
1 row in set (17.12 sec)
```



图5-59 优化前耗时

图5-60是优化后的SQL执行时间。

```
mysql> select count(*) from artist a join
-> (select artist_id from artist where name like '%Queen%') b
-> on a.artist_id=b.artist_id;
+-----+
| count(*) |
+-----+
|      280 |
+-----+
1 row in set (8.31 sec)
```

图5-60 优化后耗时

### 3.limit分页优化

在limit分页优化方面，下面会用两个案例来进行说明。

案例一

看下面的limit查询：

```
mysql> select SQL_NO_CACHE * from test1 order by id limit 99999, 10;
```

```
+-----+-----+-----+
/ id / tid / name /
+-----+-----+-----+
/ 100000 / 100000 / abc /
/ 100001 / 100001 / abc /
/ 100002 / 100002 / abc /
/ 100003 / 100003 / abc /
/ 100004 / 100004 / abc /
/ 100005 / 100005 / abc /
/ 100006 / 100006 / abc /
/ 100007 / 100007 / abc /
/ 100008 / 100008 / abc /
```

```
/ 100009 / 100009 / abc /
+-----+-----+-----+
10 rows in set (0.07 sec)
```

在上面的SQL语句中，虽然用上了id索引，但要从第一行开始起定位至99999行，然后再扫描出后10行，相当于进行一个全表扫描，显然效率不高。我们来看看优化方法，如下：

```
mysql>select SQL_NO_CACHE * from test1 where id >= 100000 order by id limit 10;
+-----+-----+-----+
/ id / tid / name /
+-----+-----+-----+
/ 100000 / 100000 / abc /
/ 100001 / 100001 / abc /
/ 100002 / 100002 / abc /
/ 100003 / 100003 / abc /
/ 100004 / 100004 / abc /
/ 100005 / 100005 / abc /
/ 100006 / 100006 / abc /
/ 100007 / 100007 / abc /
/ 100008 / 100008 / abc /
/ 100009 / 100009 / abc /
+-----+-----+-----+
10 rows in set (0.00 sec)
```

第二种写法比第一种快了7倍，利用id索引直接定位100000行，然后再扫描出后10行，相当于一个range范围扫描。

### 案例二：给某房地产网优化

下面是某房地产网站的开发人员写的一条SQL，据了解是为了读取最老的新闻标题（从334570行数据开始读取后面的10条记录）。

```
select id , title , createdate from 表名 order by createdate asc limit 334570 , 10;
```

图5-61是索引情况。



图5-61 索引

优化器explain的执行计划情况如图5-62所示。

```
mysql> explain select id,title,createdate from  order by createdate asc limit 334570,10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	1	ALL	NULL	NULL	NULL	NULL	355069	Using filesort

1 row in set (0.00 sec)

图5-62 全表扫描

先来看一下这个SQL的执行情况，如图5-63所示。



```
mysql> select id,title,createdate from  order by createdate asc limit 334570,10;
```

id	title	createdate
340156	20 即将启动 开启4阅卷 占订	2012-06-10 10:11:36
340157	【汉嘉郡 6月10日	2012-06-10 10:51:11
340158	【海洲景秀世家】	2012-06-10 11:48:48
340159	【新化国际公寓】	2012-06-10 13:16:42
340160	神力 首 航员	2012-06-10 13:29:44
340161	人 动	2012-06-10 14:02:11
340162	5月C	2012-06-10 14:10:01
340163	写 结 去 切	2012-06-10 14:23:07
340164	谁来	2012-06-10 14:25:28
340165	谁来买单 别墅购买群体趋向细分	2012-06-10 14:25:47

10 rows in set (21.87 sec)

图5-63 优化前查询结果耗时时间

可以看到，这条SQL的执行速度是比较慢的，优化思路是先取出334570行后面的1条记录id，然后采用表内连接的方法，取出后10条，在通过这种方式进行优化后，图5-64是优化后的情况。

```
mysql> select a.id,a.title,a.createdate from a
-> join (select id from a order by createdate asc limit 334570,1) b
-> on a.id >=b.id limit 10;
```

id	title	createdate
340156	2012-06-10 10:11:36	2012-06-10 10:11:36
340157	【汉嘉都市森林】晶锐公寓 3月11日开盘	2012-06-10 10:51:11
340158	【海洲景秀世家】88-288平米商铺10日火爆认筹	2012-06-10 11:48:48
340159	【】 75m² 热销	2012-06-10 13:16:42
340160	神九飞天或16日 五大看点揭秘搭载中国女宇航员	2012-06-10 13:29:44
340161	人保部确定退休金领取认证程序将启动	2012-06-10 14:02:11
340162	5月CPI 通胀压力依然沉重	2012-06-10 14:10:01
340163	写字楼综合性价比 专家来支招	2012-06-10 14:23:07
340164	谁来买单 消费群体趋向细分	2012-06-10 14:25:28
340165	谁来买单 消费群体趋向细分	2012-06-10 14:25:47

10 rows in set (13.10 sec)

图5-64 优化后查询结果耗时时间

#### 4.count(\*)统计数据如何加快速度

Count(\*)在innodb里一般都是比较慢的，尤其是数据量很大的情况，所以通过SQL语句的变通，可以达到一定程度速度的提升，例如，下面这个例子。

案例一：count(辅助索引)快于count(\*)

在下面的SQL语句中，第一种写法耗时了6分40秒，改为辅助索引后耗时为2分59秒。

```
mysql>select count(*) from UP_User;
```

```
+-----+
```

```
/ count(*) /
```

```
+-----+
```

```
/ 77515560 /
```

```
+-----+
```

```
1 row in set (6 min 40.36 sec)
```

```
mysql>select count(*) from UP_User where Sid >= 0;
```



```
+-----+
/ count(*) /
+-----+
/ 77515560 /
+-----+
```

1 row in set (2 min 59.14 sec)

通过上面的测试，得出的结论是count（辅助索引）较快，聚集索引是把主键和数据保存在一起，辅助索引不存放数据，而是有一个指针指向对应的数据块。因此，在统计的时候消耗的资源也更少，速度也就更快。

### 案例二：count(distinct)优化

优化distinct最有效的方法是利用索引来做排重操作，先把排重的记录查找出来再通过count统计，这样效果更高，下面请看两个例子，如图5-65和图5-66所示。

```
mysql> explain select count(distinct k) from sbtest;
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | SIMPLE | sbtest | index | NULL | k | 4 | NULL | 1000073 | Using index |
+-----+
1 row in set (0.01 sec)

mysql> explain select count(*) from (select distinct k from sbtest)tmp;
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | PRIMARY | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Select tables optimized away |
| 2 | DERIVED | sbtest | range | NULL | k | 4 | NULL | 19 | Using index for group-by |
+-----+
2 rows in set (0.07 sec)
```

图5-65 explain执行计划情况



```
mysql> select count(distinct k) from sbtest;
+-----+
| count(distinct k) |
+-----+
|                1 |
+-----+
1 row in set (0.50 sec)

mysql> select count(*) from (select distinct k from sbtest)tmp;
+-----+
| count(*) |
+-----+
|        1 |
+-----+
1 row in set (0.00 sec)
```

图5-66 查询耗时时间对比



注意 虽然上述方法加快了速度，但类似统计的SQL语句（如select count(\*)总数、select sum()求和）千万不要在主库上执行。因为生产环境数据库是InnoDB引擎（OLTP联机事务处理），它不像MyISAM引擎（OLAP联机分析处理）那样内置了一个计数器，可在使用select count(\*) from table的时候，直接从计数器中取出数据。InnoDB必须要全表扫描一次方能得到总的数量，且会锁表（表级锁，不是行锁），当数据达到千万级别时，速度会很慢，一个SQL就让数据库挂掉。

#### 5.or条件如何优化

在SQL语句里有or条件，则会用不到索引，下面看这个例子：

下面的SQL语句中，user和age字段都建立索引。

```
mysql> SELECT * FROM USER WHERE name='d' or age=41;
```

```
+----+-----+-----+
| id | name | age |
+----+-----+-----+
| 4 | d | 23 |
```





可以看到已经用到了索引，扫描1行出结果，观察rows那列。

## 6.使用ON DUPLICATE KEY UPDATE子句

MySQL中有一种非常高效的主键冲突处理判断，冲突则执行update，不冲突则执行insert逻辑的语句：ON DUPLICATE KEY UPDATE，比如，下面这个例子：

```
INSERT INTO UP_Relation(OwnerId, ContactId, IsBuddy, IsChatFriend, IsBlackList)VALUES(v_UserId, v_ContactId, 1, 0, 0)
ON DUPLICATE KEY UPDATE IsBuddy=1, IsChatFriend=0;
```

相当于：

```
IF EXISTS(select * from UP_Relation where OwnerId=v_UserId and ContactId= v_ContactId) THEN
    UPDATE UP_Relation
    SET IsBuddy=1, IsChatFriend=0
    WHERE OwnerId=v_UserId AND ContactId= v_ContactId;
ELSE
    INSERT INTO UP_Relation(OwnerId, ContactId, IsBuddy, IsChatFriend, IsBlackList)
    VALUES(v_UserId, v_ContactId, 1, 0, 0);
END IF;
```

前一种写法明显简洁高效，所以在MySQL开发当中，如果有类似逻辑，尽量使用第一种写法，而且把它作为开发规范要求。另外需要注意的是，使用这种方式的表必须是基于主键或唯一索引的操作，否则无法使用。

例子：当插入id值为3发生主键冲突时，把id+1：

```
mysql>select * from gg;
```

```
+----+-----+
```

```
| id | name |
```

```
+----+-----+
```

```
| 1 | a |
```

```
| 2 | b |
```

```
| 3 | c |
```

```
+----+-----+
```

```
3 rows in set (0.00 sec)
```

```
mysql>insert into gg values(3, 'd') ON DUPLICATE KEY UPDATE id=id+1;
```

```
Query OK, 2 rows affected (0.00 sec)
```



```
mysql>select * from gg;
```

```
+-----+-----+
```

```
| id | name |
```

```
+-----+-----+
```

```
| 1 | a |
```

```
| 2 | b |
```

```
| 4 | c |
```

```
+-----+-----+
```

```
3 rows in set (0.00 sec)
```

### 7.不必要的排序

下面这条SQL用于统计子表的记录的条数，那么排序操作在这里是多此一举了，且消耗性能。

```
SELECT count(1) AS rs_count FROM
(
    SELECT a.id , a.title , a.content , b.log_time , b.name
    FROM a , b
    WHERE a.content LIKE 'rc_%' AND a.id = b.id
    ORDER BY a.title DESC
) AS rs_table;
```

修改为：

```
SELECT count(1) AS rs_count FROM
(
    SELECT a.id FROM a JOIN b
    ON a.id = b.id AND a.content LIKE 'rc_%'
) AS rs_table;
```

去掉ORDER BY a.title DESC排序后，性能提升。

### 8.不必要的嵌套select查询

下面这条SQL在子表的查询结果里会再过滤出前30条记录，这样对性能是有所消耗的。

```
SELECT * FROM
(
```

```
SELECT a.id , a.title , a.content , b.log_time , b.name  
FROM a , b  
WHERE a.content LIKE 'rc_ %' AND a.id = b.id  
ORDER BY a.title DESC  
) AS rs_table LIMIT 0 , 30;
```

修改为：

```
SELECT a.id , a.title , a.content , b.log_time , b.name  
FROM a JOIN b  
ON a.id = b.id AND a.content LIKE 'rc_ %'  
ORDER BY a.title DESC LIMIT 0 , 30;
```

这样用一条Select就满足了查询条件，避免了外层的嵌套select查询，性能提升。

#### 9. 不必要的表自身连接

下面的例子中使用了不必要的join表连接，结果数据有重复，不是想要的结果，如图5-67所示。

从图5-67可以看到，查出来的数据有重复记录，修改为图5-68所示的这条SQL。

这样修改后就可以查询到正确的结果了。

#### 10. 用where子句替换having子句

避免使用having子句，having只会在检索出所有记录之后才对结果集进行过滤。这个处理需要排序、总计等操作。如果能通过where子句限制记录的数目，那就能减少这方面的开销。例如，下面这个SQL语句，如图5-69所示。

0  
1  
2  
3  
4  
5

```
SELECT PL.pid, PL.request_date, PL.bean_total FROM paybean_success PL INNER JOIN  
(SELECT pid, MIN(request_date) request_date FROM paybean_success GROUP BY pid) PR  
ON PR.pid = PL.pid AND PR.request_date = PL.request_date
```

1 结果 2 Profiler 3 信息 4 表数据 5 信息

(只读)

pid	request_date	bean_total
3001	2013-03-06 10:55:15	2.20
200591373	2013-03-22 18:48:46	-100.00
200637910	2013-03-27 18:22:49	-2.00
200980253	2013-03-21 14:54:18	100.00
201266578	2013-03-28 14:48:18	-1.00
201669188	2013-03-28 14:46:28	-1.00
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718436	2013-03-03 11:41:25	2.20
201718974	2013-04-02 14:23:31	-1.00
201718977	2013-03-27 15:29:05	-387.00
202803346	2013-03-27 18:26:52	-1.00
202817759	2013-03-27 20:53:52	-1.00
202818090	2013-03-28 14:02:50	-1.00
202818202	2013-03-27 14:54:14	-1.00
202818415	2013-03-27 17:28:21	-1.00
202818454	2013-03-28 15:26:55	-200.00
202818478	2012-10-27 11:41:25	2.20

图5-67 查询结果有重复数据



14  
15  
16  
17

SELECT pid, MIN(request\_date) request\_date, bean\_total FROM paybean\_success GROUP BY pid

1 结果 2 Profiler 3 信息 4 表数据 5 信息

(只读)

☐ pid request\_date bean\_total

☐ 3001 2013-03-06 10:55:15 2.20

☐ 200591373 2013-03-22 18:48:46 -100.00

☐ 200637910 2013-03-27 18:22:49 -2.00

☐ 200980253 2013-03-21 14:54:18 100.00

☐ 201266578 2013-03-28 14:48:18 -1.00

☐ 201669188 2013-03-28 14:46:28 -1.00

☐ 201718436 2013-03-03 11:41:25 2.20

☐ 201718974 2013-04-02 14:23:31 -1.00

☐ 201718977 2013-03-27 15:29:05 -387.00

☐ 202803346 2013-03-27 18:26:52 -1.00

☐ 202817759 2013-03-27 20:53:52 -1.00

☐ 202818090 2013-03-28 14:02:50 -1.00

☐ 202818202 2013-03-27 14:54:14 -1.00

☐ 202818415 2013-03-27 17:28:21 -1.00

☐ 202818454 2013-03-28 15:26:55 -200.00

☐ 202818478 2012-10-27 11:41:25 2.20

☐ 202818530 2013-04-09 11:51:22 8.00

☐ 202818681 2013-04-01 16:43:33 -1.00

☐ 202818701 2013-04-01 15:11:54 -1.00

☐ 202818702 2013-04-01 16:14:39 -1.00

☐ 202818758 2013-04-02 19:07:21 -1.00

☐ 202819111 2013-04-09 17:01:42 -1.00

图5-68 正确的查询结果

如果改用where子句替换having子句，性能就会不一样，如图5-70所示。

一般情况下，having子句中的条件用于对一些集合函数的比较，如count()等。除此以外，都应该写在where子句中。

```
mysql> select * from sbtest where id>40 group by id limit 3;
+----+-----+-----+-----+
| id | k | c | pad |
+----+-----+-----+-----+
| 41 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
| 42 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
| 43 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
+----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> select * from sbtest group by id having id>40 limit 3;
+----+-----+-----+-----+
| id | k | c | pad |
+----+-----+-----+-----+
| 41 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
| 42 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
| 43 | 0 |   | qqqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt |
+----+-----+-----+-----+
3 rows in set (0.00 sec)
```

图5-69 查询结果

```
mysql> explain select * from sbtest group by id having id>40 limit 3;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sbtest	index	NULL	PRIMARY	4	NULL	1000073	

1 row in set (0.00 sec)

```
mysql> explain select * from sbtest where id>40 group by id limit 3;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sbtest	range	PRIMARY	PRIMARY	4	NULL	500036	Using where

1 row in set (0.01 sec)

图5-70 优化器执行结果对比



### 5.5.3 合理使用索引

适当的索引对应用的性能来说至关重要，而且在MySQL中建议使用索引，它的速度是极快的。遗憾的是，索引也有相关的开销。每次向表中写入时（如INSERT、UPDATE或DELETE操作），如果带有一个或多个索引，那么MySQL也要更新各个索引。这样就增加了对各个表写入操作的开销。此外，索引还增加了数据库的规模。只有当某列被用于where子句时，才能享受到索引性能提升的好处。如果不使用索引，它就没有价值，而且会带来维护上的开销。

#### 1. 单列索引和联合索引的对比

图5-71中的表toid是单列索引，datetime也是单列索引，我们看一下explain执行计划。



Autocomplete: [Tab]->Next Tag. [Ctrl+Space]->List Matching Tags. [Ctrl+Enter]->List All Tags.

```
1 EXPLAIN SELECT id, toid, DATETIME, state, cinmessage FROM cin_offlinemessage_387 WHERE toid = 229570387
2 AND state=0 AND DATETIME >= DATE_ADD(NOW(),INTERVAL -7 DAY) ORDER BY toid;
3
```

table	type	possible_keys	key	key_len	ref	rows	Extra
cin_offlinemessage_387	ref	cin_offlinemessage_toid,cin_offlinemessage_datetime	cin_offlinemessage_toid	9	const	653	Using where

图5-71 单列索引

总共扫描了653行，当我们执行查询的时候，MySQL只能使用一个索引，虽然有两个索引，但不能同时都用上，所以最终只选择了它认为最优的toid索引。

那如果把toid字段和datetime字段建立为联合索引会怎样？再来看一下explain执行计划，如图5-72所示。





图5-72 联合索引

可以看出，效率高了许多，仅仅扫描了91行，减少了562行（653行-91行）的扫描。注意，联合索引要遵循最左侧原则，例如，下面这些查询都能够使用这个fname\_lname\_age索引：

*Select peopleid FROM people Where firstname='Mike' AND lastname='Sullivan' AND age='17';*

*Select peopleid FROM people Where firstname='Mike' AND lastname='Sullivan';*

*Select peopleid FROM people Where firstname='Mike';*

下面这些查询不能够使用这个fname\_lname\_age索引：

*Select peopleid FROM people Where lastname='Sullivan';*

*Select peopleid FROM people Where age='17';*

*Select peopleid FROM people Where lastname='Sullivan' AND age='17';*

这里需要特别注意！

2. 字段使用函数，将不能用到索引

先来看两条SQL语句：

*mysql> select createtime from aa where date(createtime)=curdate();*

*+-----+*

*/ createtime /*

*+-----+*

```

/ 2012-07-11 10:03:45 /
+-----+
1 row in set (0.01 sec)

mysql> select createtime from aa where createtime >
DATE_FORMAT(CURDATE(), '%Y-%m-%d');
+-----+
/ createtime /
+-----+
/ 2012-07-11 10:03:45 /
+-----+
1 row in set (0.00 sec)

```

两条SQL语句的结果一样，但写法不同，哪个性能好呢？答案是第2个，因为在where后面的字段使用函数，将不会用到索引，用优化器查看下情况（如图5-73所示）。

```

mysql> explain select createtime from aa where createtime > DATE_FORMAT(CURDATE(), '%Y-%m-%d');
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | SIMPLE | aa | range | createtime | createtime | 9 | NULL | 1 | Using where; Using index |
+-----+
1 row in set (0.00 sec)

mysql> explain select createtime from aa where date(createtime)=curdate();
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | SIMPLE | aa | index | NULL | NULL | 17 | NULL | 1 | Using where; Using index |
+-----+
1 row in set (0.00 sec)

```

图5-73 函数不能用到索引

很明显，“select createtime from aa where date(createtime)=curdate();”语句进行了全表扫描。再看下面这一个例子（如图5-74所示）：



```
mysql> explain SELECT COUNT(*) FROM cdb_forum_post where DateDiff(NOW(),from_unixtime(dateline,'%Y-%m-%d')) = 0;
+-----+
| id | select_type | table          | type | possible_keys | key      | key_len | ref | rows | Extra              |
+-----+
| 1 | SIMPLE      | cdb_forum_post | index | NULL          | dateline | 4       | NULL | 709976 | Using where; Using index |
+-----+
1 row in set (0.00 sec)

mysql> explain SELECT COUNT(*) FROM cdb_forum_post where dateline >= UNIX_TIMESTAMP(DATE_FORMAT(now(),'%Y-%m-%d'));
+-----+
| id | select_type | table          | type | possible_keys | key      | key_len | ref | rows | Extra              |
+-----+
| 1 | SIMPLE      | cdb_forum_post | range | dateline      | dateline | 4       | NULL | 2235 | Using where; Using index |
+-----+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM cdb_forum_post where DateDiff(NOW(),from_unixtime(dateline,'%Y-%m-%d')) = 0;
+-----+
| COUNT(*) |
+-----+
|      4042 |
+-----+
1 row in set (0.50 sec)

mysql> SELECT COUNT(*) FROM cdb_forum_post where dateline >= UNIX_TIMESTAMP(DATE_FORMAT(now(),'%Y-%m-%d'));
+-----+
| COUNT(*) |
+-----+
|      4042 |
+-----+
1 row in set (0.01 sec)
```

图5-74 函数不能用到索引



注意 MySQL目前还不支持函数索引。

3.致命的无引号导致全表扫描，无法用到索引

这是开发人员在日常写SQL语句时容易忽视的一个问题。

先来查看表结构和索引，如图5-75所示。



```
mysql> desc playerinfo ;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
state	int(11)	NO		0	
type	int(11)	NO		0	
accmode	int(11)	NO		0	
terminal	int(11)	NO		0	
regdate	datetime	NO		1960-01-01 00:00:00	
disableddate	datetime	YES		NULL	
lastlogin	datetime	YES		NULL	
sex	int(11)	NO		0	
name	varchar(50)	NO	MUL	NULL	
domain	varchar(10)	NO		feinno	
password	varchar(50)	YES			
portait_id	int(11)	NO		0	

图5-75 表结构

优化器的执行情况如图5-76所示。

```
mysql> explain SELECT * FROM playerinfo where name=1045156967;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	playerinfo	ALL	name_domain, name	NULL	NULL	NULL	1669030	Using where

1 row in set (0.01 sec)

图5-76 全表扫描

由于name是字符型，那么where条件必须要加引号，这条SQL改成下面这样即可用到索引（如图5-77所示）。

```
mysql> explain SELECT * FROM playerinfo where name='1045156967';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	playerinfo	ref	name_domain,name	name_domain	152	const	1	Using where

1 row in set (0.00 sec)

图5-77 使用到了索引

对比一下两个SQL的执行时间（如图5-78所示）。

```
mysql> SELECT * FROM playerinfo where name=1045156967;
Empty set (2.27 sec)
```

```
mysql> SELECT * FROM playerinfo where name='1045156967';
Empty set (0.00 sec)
```

图5-78 查询时间对比

很明显，用数字当字符类型，数字一定要加引号。

4.当取出的数据量超过表中数据的20%，优化器就不会使用索引，而是全表扫描

我们看下面的一个例子，这里要取出大于2012年3月15日的数据，如图5-79所示。

```
mysql> explain select count(id) from cin_offlinemessage where `datetime` >= '2012-03-15' and state=0;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	cin_offlinemessage	ALL	cin_offfflinemessage_datetime	NULL	NULL	NULL	45046637	Using where

1 row in set (0.00 sec)

图5-79 全表扫描

优化器执行的结果是全表扫描。虽然有索引，但没有用上，扫描的行数太多了（8位数），优化器认为全表扫描比索引来得快。下面缩短了时间范围，如图5-80所示。



```
mysql> explain select count(id) from cin_offlinemessage where `datetime` between '2012-03-15 00:00:00' and '2012-03-16 23:59:59' and state=0;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	cin_offlinemessage	range	cin_offfflinemessage_datetime	cin_offfflinemessage_datetime	9	NULL	3534912	Using where

1 row in set (0.00 sec)

图5-80 使用到索引

现在，已经使用到了索引。

### 5.考虑不为某些列建立索引

有时候，进行全表浏览要比必须读取索引和数据表更快，尤其是当索引包含的是平均分布的数据集时更是如此。对此，典型的例子是性别，它有两个均匀分布的值（男和女）。通过性别男或女读取大概一半的记录。在这种情况下进行全表扫描浏览要更快。

### 6.order by、group by优化

我们先看一条SQL，如图5-81所示。

```
mysql> EXPLAIN SELECT * FROM `test_change` WHERE `pid` = 200980253 ORDER BY `change_date`;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	test_change	ref	IX_pid	IX_pid	5	const	2264	Using where; Using filesort

1 row in set (0.00 sec)

图5-81 使用了排序Using filesort

从优化器上来看，里面使用了排序Using filesort，这条SQL优化的关键是看如何增加索引，目前已经有了pid索引，那么给change\_date增加索引吗？我们来试试，如图5-82所示。

很遗憾，还是没有效果。前面已经说过，一条SQL只能有一个索引，如果有多条索引，优化器会选择那个最优的，所以我们可以考虑为pid字段和change\_date建立一个联合索引，再来看看效果（如图5-83所示）。

可以看到，Using filesort排序已经没有了。Group by的优化方法也是一样的。

如果order by后面有多个字段排序，它们的顺序要一致，如果一个是降序，一个是升序，也会出现Using filesort排序，如下面这个例子（如图5-84所示）。



```
mysql> create index IX_change_date on test_change(change_date);
Query OK, 0 rows affected (0.04 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> show index from test_change;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Index_comment
test_change	0	PRIMARY	1	change_id	A	2950	NULL	NULL	
test_change	1	IX_pid	1	pid	A	73	NULL	NULL	
test_change	1	IX_change_date	1	change_date	A	196	NULL	NULL	

```
3 rows in set (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM `test_change` WHERE `pid` = 200980253 ORDER BY `change_date`;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	test_change	ref	IX_pid	IX_pid	5	const	2264	Using where; Using filesort

```
1 row in set (0.00 sec)
```

图5-82 change\_date增加索引

```
mysql> create index IX_pid_change_date on cash_change(pid,change_date);
Query OK, 0 rows affected (0.05 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> EXPLAIN SELECT * FROM `cash_change` WHERE `pid` = 200980253 ORDER BY `change_date`;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type | possible_keys | key      | key_len | ref  | rows | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | cash_change | ref  | IX_pid,IX_p_c | IX_p_c   | 5       | const | 2264 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-83 pid字段和change\_date建立联合索引

```
mysql> EXPLAIN SELECT * FROM `cash_change` force index (IX_p_c_d)
-> WHERE `pid` = 200980253 ORDER BY `change_date` DESC,delta_rmb ASC;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type | possible_keys | key      | key_len | ref  | rows | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | cash_change | ref  | IX_p_c_d      | IX_p_c_d | 5       | const | 2264 | Using where; Using filesort |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-84 排序不一致

改为排序一致即可解决上述问题，如图5-85所示。

```
mysql> EXPLAIN SELECT * FROM `cash_change` force index (IX_p_c_d)
-> WHERE `pid` = 200980253 ORDER BY `change_date` DESC,delta_rmb DESC;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type | possible_keys | key      | key_len | ref  | rows | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | cash_change | ref  | IX_p_c_d      | IX_p_c_d | 5       | const | 2264 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-85 排序一致

在这里，Using filesort排序已经没有了。

### 7.MySQL5.6支持explain update

在MySQL之前的版本中，explain只支持select，但在最新的5.6版本中，它支持explain update/delete了，如图5-86所示。

```
mysql> select @@version;
+-----+
| @@version |
+-----+
| 5.6.5-m8  |
+-----+
1 row in set (0.02 sec)

mysql> explain update user set age=11 where name='h' ;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user | range | PRIMARY, name, age | name | 18 | NULL | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.03 sec)

mysql> 
```

图5-86 explain支持update

### 8.MySQL5.6 InnoDB引擎支持全文索引

在MySQL以前的版本，只有MyISAM引擎有全文索引fulltext，而在5.6版本里，首次在InnoDB引擎上支持全文索引，而且中文支持的还可以。图5-87是相应的截图。



```
mysql> show create table news\G;
***** 1. row *****
      Table: news
Create Table: CREATE TABLE `news` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(2000) NOT NULL,
  `content` text NOT NULL,
  PRIMARY KEY (`id`),
  FULLTEXT KEY `ix_t_c` (`title`,`content`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8
1 row in set (0.03 sec)

ERROR:
No query specified
```

图5-87 InnoDB支持全文索引

### 9. MySQL5.6支持Multi-Range Read索引优化

对大表（基于辅助索引）做范围扫描时，会导致产生许多随机I/O，而对于普通磁盘来说，随机I/O的性能很差，很产生瓶颈，在MySQL5.6里，对这种情况进行了优化，一个新的名词Multi-Range Read出现了，优化器会先扫描索引，然后收集每行的主键，并对主键排序，此时就可以用主键顺序访问基表，即用顺序I/O代替了随机I/O。

在未开启MBR时，在explain中看到的情况是这样（如图5-88所示）。

开启MBR后，在explain中看到的情况是这样（如图5-89所示）。

针对这两种情况，我们进行一下对比测试，如图5-90和图5-91所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.01 sec)

mysql> explain select * from t_06 where i2 > 2000 and i2 < 4000;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t_06 | range | ix_i2 | ix_i2 | 5 | NULL | 1997 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-88 未使用MBR

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.08 sec)

mysql> explain select * from t_06 where i2 > 2000 and i2 < 4000;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t_06 | range | ix_i2 | ix_i2 | 5 | NULL | 1997 | Using index condition; Using MRR |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.10 sec)
```

图5-89 已使用MBR

```

9979 | 999 | qqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt
9980 | 999 | qqqqqqqqqqwwwwwwwwwwweeeeeeeeeerrrrrrrrrrrrtttttttttt
+-----+-----+
9880 rows in set (1.47 sec)

mysql> select version();
+-----+
| version() |
+-----+
| 5.5.19-log |
+-----+
1 row in set (0.03 sec)

```

图5-90 未使用MBR执行时间

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.04 sec)
```

图5-91 开启MBR

可以明显地看到未开启MBR耗时1分47秒，开启MBR耗时0.34秒。

## 10.MySQL5.6优化了Index Merge合并索引

在MySQL5.6中，优化了合并索引，也就是说，一条SQL可以用上两个索引了。

先来看一下在MySQL5.5中索引的表现（如图5-92和图5-93所示）。



```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> show create table t\G;
***** 1. row *****
      Table: t
Create Table: CREATE TABLE `t` (
  `a` int(10) unsigned DEFAULT NULL,
  `b` int(10) unsigned DEFAULT NULL,
  KEY `i_t_a` (`a`),
  KEY `i_t_b` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

图5-92 表结构

```
mysql> explain select * from t where a=1 or b=10;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t | ALL | i_t_a,i_t_b | NULL | NULL | NULL | 40 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-93 执行计划全表扫描

可以看到，在MySQL5.5中无法用到索引，优化后，仍旧只能用到一条索引（如图5-94所示）。

```
mysql> explain select * from t where a=1 union all select * from t where b=10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t	ref	i_t_a	i_t_a	5	const	8	Using where
2	UNION	t	ref	i_t_b	i_t_b	5	const	3	Using where
NULL	UNION RESULT	<union1,2>	ALL	NULL	NULL	NULL	NULL	NULL	

3 rows in set (0.01 sec)

图5-94 or改为union优化

再来看一下在MySQL5.6中的表现（如图5-95所示）。

可以看到，两个索引一起用到了，这是因为采用了索引合并的方式。

但是，如果是三个字段索引，则用不上索引合并（如图5-96所示）。

### 11.MySQL5.6支持Index Condition Pushdown索引优化

这个特性是MySQL5.6新引进的，在解释该特性之前，先看下面的一个例子（如图5-97所示）。

```
mysql> select version();
```

version()
5.6.6-m9-log

```
1 row in set (0.02 sec)
```

```
mysql> explain select * from t where a=1 or b=10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t	index_merge	i_t_a,i_t_b	i_t_a,i_t_b	5,5	NULL	11	Using union(i_t_a,i_t_b); Using where

```
1 row in set (0.03 sec)
```

图5-95 索引合并

```
mysql> explain select * from t where a=1 or b=4 or c=18;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t	ALL	i_t_a,i_t_b,i_t_c	NULL	NULL	NULL	50	Using where

```
1 row in set (0.04 sec)
```

图5-96 三个字段不能索引合并



```
mysql> show create table student\G;
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) NOT NULL DEFAULT '0',
  `name` varchar(6) DEFAULT NULL,
  `class` int(11) DEFAULT NULL,
  `score` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `name` (`name`),
  KEY `IX_C_S` (`class`,`score`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec) |
```

图5-97 表结构

这里class和score为联合索引。在MySQL5.5中的表现如图5-98所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.20-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> explain select * from student where class=1 and score > 60;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | range | IX_C_S | IX_C_S | 10 | NULL | 5 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-98 未使用ICP

在MySQL5.5中，未开启Index Condition Pushdown（简称ICP），所以首先会根据class=1来查找记录，检索的结果将指向聚集索引，

最后根据score>60进行过滤，把最终的结果返回给用户。所以这里看到的是Using where。  
在MySQL5.6中的表现如图5-99所示。

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.6.6-m9-log |
+-----+
1 row in set (0.04 sec)

mysql> explain select * from student where class=1 and score > 60;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | range | IX_S_C | IX_S_C | 5 | NULL | 14 | Using index condition; Using MRR |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.04 sec)
```

图5-99 已使用ICP

在MySQL5.6中，开启了Index Condition Pushdown（简称ICP），在class=1查找记录的同时，会根据score>60进行过滤，然后检索的结果指向聚集索引，最后返回给用户，所以这里看到的是Using index condition，ICP减少了存储引擎访问表的次数，从而提高数据库的整体性能。

## 5.6 my.cnf配置文件调优

在MySQL数据库性能调优中，首先要考虑的就是Schema设计，这一点非常重要，一个糟糕的Schema设计即使是在性能强劲的服务器上运行，也会表现出很差的性能。和Schema相似，查询语句的设计也会影响MySQL的性能，应该避免写出低效的SQL查询。最后要考虑的就是参数优化，MySQL数据库的默认设置性能非常差，仅仅起一个功能测试的作用，不能在生产环境中运行，因此要对一些参数进行调整。比如，无法使用索引的情况下进行全表扫描、全索引扫描等。在这种时候，MySQL会按照数据的存储顺序依次读取数据块，每次读取的数据块首先会暂存在read\_buffer\_size中，当buffer空间被写满或者全部数据读取结束后，再将buffer中的数据返回给上层调用者，以提高效率。



### 5.6.1 per\_thread\_buffers优化

对于per\_thread\_buffers，可以将其理解为Oracle的PGA，为每个连接到MySQL的用户进程分配的内存。其包含如下几个参数：

#### 1.read\_buffer\_size

该参数用于表的顺序扫描，表示每个线程分配的缓冲区大小。比如，在进行全表扫描时，MySQL会按照数据的存储顺序依次读取数据块，每次读取的数据块首先会暂存在read\_buffer\_size中，当buffer空间被写满或者全部数据读取结束后，再将buffer中的数据返回给上层调用者，以提高效率。默认为128 KB。这个参数不要设置过大，一般在128~256 KB即可。

#### 2.read\_rnd\_buffer\_size

该参数用于表的随机读取，表示每个线程分配的缓冲区大小。比如，按照一个非索引字段做order by排序操作时，就会利用这个缓冲区来暂存读取的数据。默认为256 KB。这个参数不要设置过大，一般在128~256 KB即可。

#### 3.sort\_buffer\_size

在表进行order by和group by排序操作时，由于排序的字段没有索引，会出现Using filesort，为了提高性能，可用此参数增加每个线程分配的缓冲区大小。默认为2 MB。这个参数不要设置过大，一般在128~256 KB即可。另外，一般出现Using filesort的时候，要通过增加索引来解决。比如，图5-100所示的这个例子：

```
mysql> explain select * from aa order by name;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aa	ALL	NULL	NULL	NULL	NULL	9	Using filesort

1 row in set (0.03 sec)

```
mysql> create index IX_name on aa(name);
Query OK, 0 rows affected (2.48 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> explain select * from aa order by name;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aa	index	NULL	IX_name	33	NULL	9	Using index

1 row in set (0.02 sec)

图5-100 排序执行计划

#### 4.thread\_stack

该参数表示每个线程的堆栈大小。默认为192 KB。如果是64位操作系统，设置为256 KB即可，这个参数不要设置过大。

#### 5.join\_buffer\_size

表进行join连接操作时，如果关联的字段没有索引，会出现Using join buffer，为了提高性能，可用此参数增加每个线程分配的缓冲区大小。默认为128 KB。这个参数不要设置过大，一般在128~256 KB即可。一般出现Using join buffer的时候，要通过增加索引来解决。比如，图5-101所示的这个例子：

#### 6.binlog\_cache\_size

一般来说，如果数据库中没有什么大事务，写入也不是特别频繁，将其设置为1~2 MB是一个合适的选择。如果有很大的事务，可以适当增加这个缓存值，以获得更好的性能。



```
mysql> explain select aa.* from aa join bb on aa.name=bb.name;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | bb | ALL | NULL | NULL | NULL | NULL | 6 | |
| 1 | SIMPLE | aa | ALL | NULL | NULL | NULL | NULL | 9 | Using where; Using join buffer |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.04 sec)

mysql> create index IX_name on aa(name);
Query OK, 0 rows affected (1.12 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> create index IX_name on bb(name);
Query OK, 0 rows affected (1.13 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> explain select aa.* from aa join bb on aa.name=bb.name;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | bb | index | IX_name | IX_name | 11 | NULL | 6 | Using index |
| 1 | SIMPLE | aa | ref | IX_name | IX_name | 33 | test.bb.name | 1 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.03 sec)
```

图5-101 表连接执行计划

## 7.max\_connections

该参数用来设置最大连接数，默认为100。一般设置为512~1000即可。

上面介绍了各个参数的含义，再来看看per\_thread\_buffers内存的计算公式，如下所示：

$(read\_buffer\_size + read\_rnd\_buffer\_size + sort\_buffer\_size + thread\_stack + join\_buffer\_size + binlog\_cache\_size) * max\_connections$



## 5.6.2 global\_buffers优化

对于global\_buffers，可以理解为Oracle的SGA，用于在内存中缓存从数据文件中检索出来的数据块，可以大大提高查询和更新数据的性能。它主要由以下几个参数组成：

### 1.innodb\_buffer\_pool\_size

这个参数是InnoDB存储引擎的核心参数，默认为128 MB，这个参数要设置为物理内存的60%~70%。

### 2.innodb\_additional\_mem\_pool\_size

该参数用来存储数据字典信息和其他内部数据结构。表越多，需要在这里分配的内存越多。如果InnoDB用光了这个池内的内存，InnoDB开始从操作系统分配内存，并且往MySQL错误日志中写警告信息，默认值是8 MB，当发现错误日志中已经有相关的警告信息时，就应该适当地增加该参数的大小。一般设置为16 MB即可。

### 3.innodb\_log\_buffer\_size

事务日志所使用的缓冲区。InnoDB在写事务日志的时候，为了提高性能，先将信息写入Innodb Log Buffer中，当满足innodb\_flush\_log\_trx\_commit参数所设置的相应条件（或者日志缓冲区写满）时，再将日志写到文件（或者同步到磁盘）中。可以通过innodb\_log\_buffer\_size参数设置其可以使用的最大内存空间。默认为8 MB，一般设置为16~64 MB即可。

### 4.key\_buffer\_size

该参数用来缓存MyISAM存储引擎的索引参数。MySQL5.5默认为InnoDB存储引擎，所以这个参数可以设置小一些，64 MB即可。

### 5.query\_cache\_size

缓存select语句和结果集大小的参数。详细见下一节。

上面介绍了各个参数的含义，现在来看看global\_buffers内存的计算公式，如下所示：

$$\text{innodb\_buffer\_pool\_size} + \text{innodb\_additional\_mem\_pool\_size} + \text{innodb\_log\_buffer\_size} + \text{key\_buffer\_size} + \text{query\_cache\_size}$$



注意 per\_thread\_buffers内存设置+global\_buffers设置不能大于实际物理内存，否则当并发量很高时会造成内存溢出，系统死机。



### 5.6.3 Query Cache在不同环境下的使用

Query Cache的功能就是缓存select语句和结果集。查询缓存会存储一个select查询的文本与被传送到客户端的相应结果。如果之后接收到一个同样的查询，服务器将会从查询缓存中检索结果，而不是再次分析和执行这个同样的查询。

注意！查询缓存绝不返回过期数据。当数据被修改后，在查询缓存中的任何相关词条均会被转储清除。如果有某些表并不经常更改，而你又要对它执行大量的相同查询时，查询缓存将是非常有用的。

如果你的环境中写操作很少，读操作很频繁，那么打开query\_cache\_type=1，会对性能有明显提升。

如果你的环境中写操作很多，那么就不适合打开它了，因为每当表的内容更改过，Query Cache缓存的结果集就要随之刷新，频繁的刷新操作反而会使性能降低很多，在这种情况下，就要关闭它（query\_cache\_type=0），同时设置query\_cache\_size=0，query\_cache\_size=0，下面是一个性能测试对比：

关闭QCACHE

```
query_cache_size = 0
```

```
query_cache_type = 0
```

```
query_cache_limit = 0
```

OLTP test statistics:

queries performed:

read: 140000

write: 50000

other: 20000

total: 210000

transactions: 10000 (36.16 per sec.)

deadlocks: 0 (0.00 per sec.)

read/write requests: 190000 (687.08 per sec.)

other operations: 20000 (72.32 per sec.)

打开QCACHE

```
query_cache_size = 64M
```

```
query_cache_type = 1
```

```
query_cache_limit = 1M
```

OLTP test statistics:

queries performed:

read: 140000

write: 50000

other: 20000

total: 210000

transactions: 10000 (43.10 per sec.)

deadlocks: 0 (0.00 per sec.)

read/write requests: 190000 (818.84 per sec.)

other operations: 20000 (86.19 per sec.)



注意 此压力测试是在虚拟机环境下操作的，如果是真实物理机器，会有明显区别。

在MySQL5.6中，InnoDB数据页16KB与8KB性能对比测试

从MySQL5.6开始，一个新参数innodb\_page\_size可以设置InnoDB数据页为8 KB、4KB，默认为16KB。这个参数在一开始初始化时就要加入到my.cnf里，如果已经创建了表，再修改，启动MySQL会报错。

下面针对InnoDB数据页为16KB和8KB做了一个压力测试。

硬件：R710，72GB内存，6块300GB、15000转做的RAID10，XFS分区。

my.cnf参数如下：

innodb\_buffer\_pool\_size = 48G

innodb\_buffer\_pool\_instances = 8

innodb\_flush\_method = O\_DIRECT

innodb\_file\_per\_table = 1

innodb\_read\_io\_threads = 16

innodb\_write\_io\_threads = 16

innodb\_io\_capacity = 2000

innodb\_log\_files\_in\_group = 3

innodb\_flush\_log\_at\_trx\_commit = 0

innodb\_log\_file\_size = 1024M





*innodb\_max\_dirty\_pages\_pct = 90*

Sysbench参数 ( 读写 ) 如下 :

*sysbench --test=oltp*

*--mysql-table-engine=innodb*

*--oltp-table-size=100000000*

*--max-requests=1000000*

*--num-threads=100*

*--mysql-host=192.168.110.121*

*--mysql-port=3306*

*--mysql-user=admin*

*--mysql-password=123456*

*--mysql-db=test*

*--oltp-table-name=sbtest*

*--mysql-socket=/tmp/mysql.sock run*

sbtest表有1亿条记录, 文件大小为24 GB。

图5-102和图5-103是InnoDB数据页为16 KB的性能图 ( *innodb\_page\_size=16K* ) 。

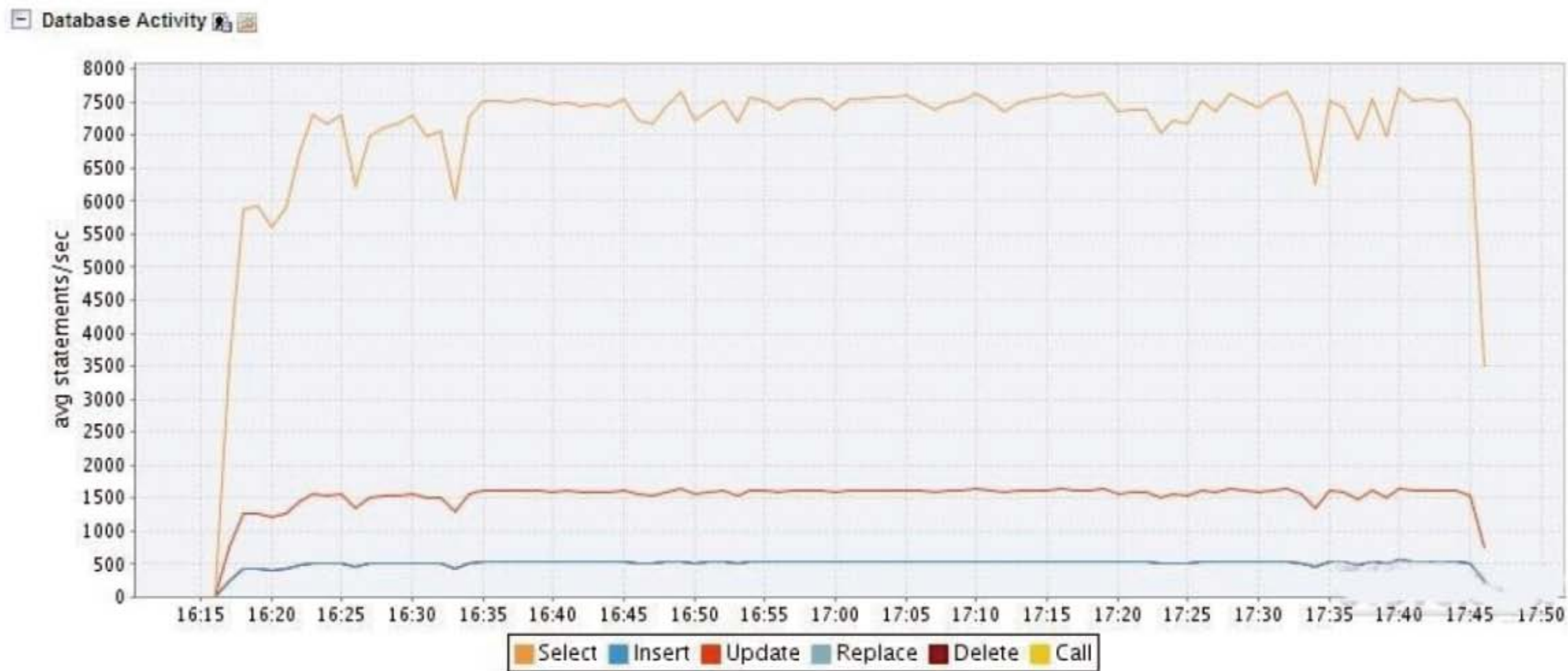


图5-102 数据库状态信息

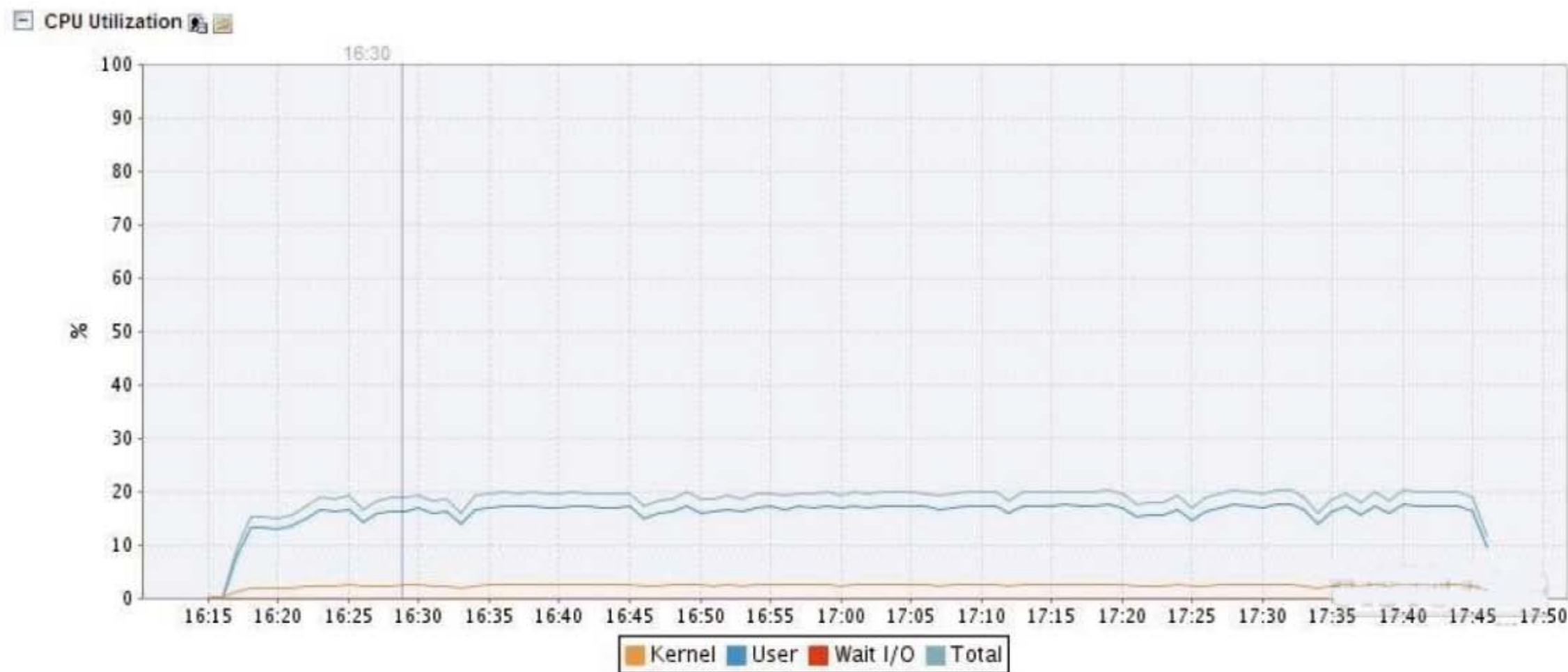


图5-103 CPU

图5-104和图5-105是InnoDB数据页为8 KB的性能图 ( innodb\_page\_size=8k )。





图5-104 数据库状态信息



图5-105 CPU

从上面的对比可以看到，默认页16 KB，对CPU压力较小，平均在20%。8 KB的InnoDB数据页，CPU压力为30%~40%，但select吞吐量要高于16 KB。

再来看看Sysbench参数（只读），如下所示：

```
sysbench --test=oltp
--mysql-table-engine=innodb
--oltp-table-size=10000000
--max-requests=0
--num-threads=100
--oltp-read-only=on
```

```
--mysql-host=192.168.110.121
--mysql-port=3306
--mysql-user=admin
--mysql-password=123456
--mysql-db=test
--oltp-table-name=sbtest
--mysql-socket=/tmp/mysql.sock run
```

其中，sbtest表有1亿条记录，文件大小24 GB。

在只读情况下，我们来看看InnoDB数据页是16 KB和8 KB的性能图，如图5-106和图5-107所示。

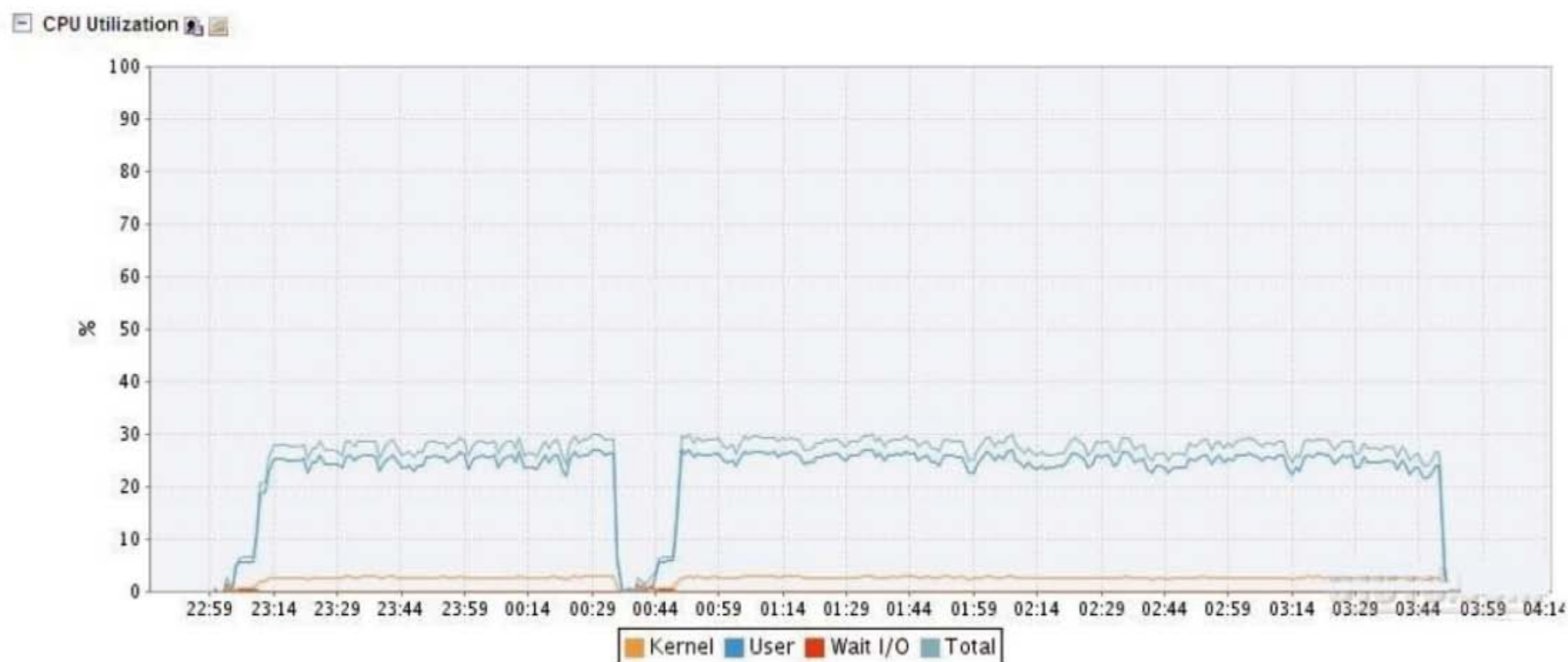


图5-106 CPU



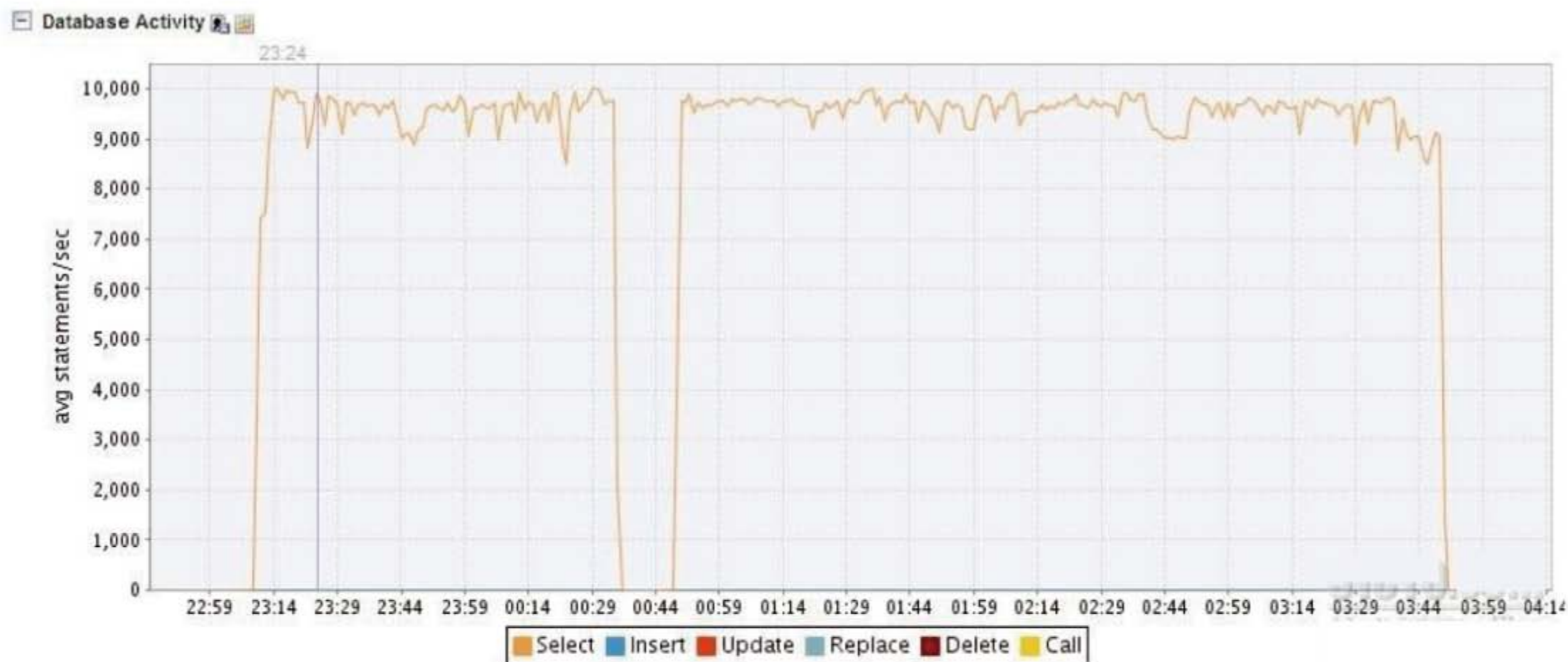


图5-107 数据库状态信息

其中，22:59分-0:29分是16 KB，0:44分-3:44分为8 KB，从图中情况来看，两者的差别不明显。所以，针对读写很频繁的情况，16 KB目前是性能比较好的。

### 5.6.4 tuning-primer.sh性能调试工具的使用

如果你用show status看MySQL的状态，会感觉很难读懂，事实上，你可以用tuning-primer.sh脚本输出可读性更好的报表，它除了提供报表以外，还进一步提供了修改建议。

对于该工具，安装和使用都非常简单，命令如下：

```
# wget http://www.day32.com/MySQL/tuning-primer.sh
# chmod 755 tuning-primer.sh
```

赋予可执行权限后，直接执行./tuning-primer.sh，下面是执行后的结果信息：

```
[root@vm01 ~]# ./tuning-primer.sh
-- MySQL PERFORMANCE TUNING PRIMER --
    - By: Matthew Montgomery -

MySQL Version 5.5.19-log i686
Uptime = 0 days 2 hrs 38 min 30 sec
Avg. qps = 0
Total Questions = 428
Threads Connected = 1
Warning: Server has not been running for at least 48hrs.
It may not be safe to use these recommendations
To find out more information on how each of these
runtime variables effects performance visit:
http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html
Visit http://www.mysql.com/products/enterprise/advisors.html
for info about MySQL's Enterprise Monitoring and Advisory Service
SLOW QUERIES
The slow query log is NOT enabled.
Current long_query_time = 10.000000 sec.
You have 0 out of 449 that take longer than 10.000000 sec. to completeYour long_query_time seems to be fine
```



## *BINARY UPDATE LOG*

*The binary update log is enabled*

*The expire\_logs\_days is not set.*

*The mysqld will retain the entire binary log until RESET MASTER or PURGE MASTER LOGS commands are run manually*

*Setting expire\_logs\_days will allow you to remove old binary logs automatically*

*See <http://dev.mysql.com/doc/refman/5.5/en/purge-master-logs.html>*

*Binlog sync is not enabled, you could lose binlog records during a server crash*

## *WORKER THREADS*

*Current thread\_cache\_size = 0*

*Current threads\_cached = 0*

*Current threads\_per\_sec = 1*

*Historic threads\_per\_sec = 0 Your thread\_cache\_size is fine*

## *MAX CONNECTIONS*

*Current max\_connections = 30*

*Current threads\_connected = 1*

*Historic max\_used\_connections = 2*

*The number of used connections is 6% of the configured maximum. You are using less than 10% of your configured max\_connections.*

*Lowering max\_connections could help to avoid an over-allocation of memory*

*See "MEMORY USAGE" section to make sure you are not over-allocating*

## *INNODB STATUS*

*Current InnoDB index space = 294 M*

*Current InnoDB data space = 276 M*

*Current InnoDB buffer pool free = 0 %*

*Current innodb\_buffer\_pool\_size = 100 M Depending on how much space your innodb indexes take up it may be safe to increase this value to up to 2 / 3 of total system memory*

## *MEMORY USAGE*

*Max Memory Ever Allocated : 128 M*

*Configured Max Per-thread Buffers : 50 M*

*Configured Max Global Buffers : 125 M*



*Configured Max Memory Limit : 175 M*

*Physical Memory : 926 MMax memory limit seem to be within acceptable norms*

#### *KEY BUFFER*

*Current MyISAM index space = 2 M*

*Current key\_buffer\_size = 1 M*

*Key cache miss rate is 1 : 1*

*Key buffer free ratio = 88 %Your key\_buffer\_size seems to be fine*

*QUERY CACHEQuery cache is supported but not enabled*

*Perhaps you should set the query\_cache\_size*

#### *SORT OPERATIONS*

*Current sort\_buffer\_size = 128 K*

*Current read\_rnd\_buffer\_size = 128 K*

*No sort operations have been performedSort buffer seems to be fine*

#### *JOINS*

*Current join\_buffer\_size = 132.00 KYou have had 0 queries where a join could not use an index properly*

*Your joins seem to be using indexes properly*

#### *OPEN FILES LIMIT*

*Current open\_files\_limit = 1024 files*

*The open\_files\_limit should typically be set to at least 2x-3x*

*that of table\_cache if you have heavy MyISAM usage.Your open\_files\_limit value seems to be fine*

#### *TABLE CACHE*

*Current table\_open\_cache = 64 tables*

*Current table\_definition\_cache = 400 tables*

*You have a total of 69 tables*

*You have 64 open tables.*

*Current table\_cache hit rate is 11% , while 100% of your table cache is in use*

*You should probably increase your table\_cache*

#### *TEMP TABLES*

*Current max\_heap\_table\_size = 16 M*

*Current tmp\_table\_size = 16 M*

*Of 555 temp tables , 13% were created on diskCreated disk tmp tables ratio seems fine*

#### **TABLE SCANS**

*Current read\_buffer\_size = 128 K*

*Current table scan ratio = 5 : 1read\_buffer\_size seems to be fine*

#### **TABLE LOCKING**

*Current Lock Wait ratio = 0 : 710Your table locking seems to be fine*

粗体字的信息要重点查看，并按照给出的建议修改相应的值。

### 5.6.5 72 GB内存的my.cnf配置文件

下面是72 GB内存生产环境中my.cnf配置文件，读者可以作为一个优化参考：

```
# MySQL configuration for 72G memory
[client]
port = 3306
socket = /tmp/mysql.sock
# The MySQL server
#####Basic#####
[mysqld]
server-id = 22
port = 3306
user = mysql
basedir = /usr/local/mysql
datadir = /mysqlData/data
tmpdir = /mysqlData/tmp
socket = /tmp/mysql.sock
skip-external-locking
skip-name-resolve
default-storage-engine = INNODB
character-set-server = utf8
wait_timeout = 100
connect_timeout = 20
interactive_timeout = 100
back_log = 500
myisam_recover
event_scheduler = ON
#####binlog#####
```



```
log-bin = /mysqlLog/logs/mysql-bin
binlog_format = row
max_binlog_size = 128M
binlog_cache_size = 2M
expire_logs_days = 5
#####replication#####
slave-net-timeout = 10
rpl_semi_sync_master_enabled = 1
rpl_semi_sync_master_wait_no_slave = 1
rpl_semi_sync_master_timeout = 1000
rpl_semi_sync_slave_enabled = 1
skip-slave-start
log_slave_updates = 1
relay_log_recovery = 1
#####slow log#####
slow_query_log = 1
slow_query_log_file = /mysqlLog/logs/mysql.slow
long_query_time = 2
#####error log#####
log-error = /mysqlLog/logs/error.log
#####per_thread_buffers#####
max_connections=1024
max_user_connections=1000
max_connect_errors=10000
key_buffer_size = 64M
max_allowed_packet = 128M
table_cache = 3096
table_open_cache = 6144
table_definition_cache = 4096
sort_buffer_size = 512K
```

```

read_buffer_size = 512K
read_rnd_buffer_size = 512k
join_buffer_size = 512K
tmp_table_size = 64M
max_heap_table_size = 64M
query_cache_type=0
query_cache_size = 0
bulk_insert_buffer_size = 32M
thread_cache_size = 64
thread_concurrency = 32
thread_stack = 256K
##### InnoDB #####
innodb_data_home_dir = /mysqlData/data
innodb_log_group_home_dir = /mysqlLog/logs
innodb_data_file_path = ibdata1:2G:autoextend
innodb_buffer_pool_size = 50G
innodb_buffer_pool_instances = 8
innodb_additional_mem_pool_size = 16M
innodb_log_file_size = 1024M
innodb_log_buffer_size = 64M
innodb_log_files_in_group = 3
innodb_flush_log_at_trx_commit = 2
innodb_lock_wait_timeout = 10
innodb_sync_spin_loops = 40
innodb_max_dirty_pages_pct = 90
innodb_support_xa = 1
innodb_thread_concurrency = 0
innodb_thread_sleep_delay = 500
innodb_file_io_threads = 4
innodb_concurrency_tickets = 1000

```

```

log_bin_trust_function_creators = 1
innodb_flush_method = O_DIRECT
innodb_file_per_table
innodb_read_io_threads = 16
innodb_write_io_threads = 16
innodb_io_capacity = 2000
innodb_file_format = Barracuda
innodb_purge_threads=1
innodb_purge_batch_size = 32
innodb_old_blocks_pct=75
innodb_change_buffering=all
transaction_isolation = READ-COMMITTED
[mysqldump]
quick
max_allowed_packet = 128M
myisam_max_sort_file_size = 10G
[mysql]
no-auto-rehash
[myisamchk]
key_buffer_size = 64M
sort_buffer_size = 256k
read_buffer = 2M
write_buffer = 2M
[mysqlhotcopy]
interactive-timeout
[mysqld_safe]
open-files-limit = 28192

```

其中具体参数含义请参考第1章。



### 5.6.6 谨慎使用分区表功能

考虑到数据量的增长，很多开发人员在设计表阶段会试图使用分区表功能，但如果没有使用好，会适得其反，下面我们看一个例子。

图5-108给出的是一个表结构。

```
mysql> show create table p1\G;
***** 1. row *****
      Table: p1
Create Table: CREATE TABLE `p1` (
  `id` int(11) NOT NULL,
  `date` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

图5-108 表结构

想一想，如果对‘date’字段进行分区，则不会成功，如图5-109所示。

```
mysql> alter table p1 partition by range columns(date)(
partition p0 values less than ('2010-01-01'),
partition p1 values less than ('2011-01-01'),
partition p2 values less than ('2012-01-01'),
PARTITION p3 VALUES LESS THAN MAXVALUE);
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

图5-109 报错信息

失败的原因是这个字段必须是主键。

重新建立主键以后，顺利创建成功（如图5-110和图5-111所示）。



```
mysql> alter table p1 drop primary key,add primary key(`id`,`date`);
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> alter table p1 partition by range columns(date)(
partition p0 values less than ('2010-01-01'),
partition p1 values less than ('2011-01-01'),
partition p2 values less than ('2012-01-01'),
PARTITION p3 VALUES LESS THAN MAXVALUE);
Query OK, 0 rows affected (0.05 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

图5-110 重建主键

```
mysql> show create table p1\G;
***** 1. row *****
      Table: p1
Create Table: CREATE TABLE `p1` (
  `id` int(11) NOT NULL,
  `date` datetime NOT NULL,
  PRIMARY KEY (`id`,`date`)QFUV
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(`date`)
(PARTITION p0 VALUES LESS THAN ('2010-01-01') ENGINE = InnoDB,
  PARTITION p1 VALUES LESS THAN ('2011-01-01') ENGINE = InnoDB,
  PARTITION p2 VALUES LESS THAN ('2012-01-01') ENGINE = InnoDB,
  PARTITION p3 VALUES LESS THAN (MAXVALUE) ENGINE = InnoDB) */
1 row in set (0.01 sec)
```

图5-111 分区后表结构

现在要增加一个字段name并建立索引，然后插入几条记录测试，如图5-112和图5-113所示。

```
mysql> alter table p1 add name varchar(10) not null;
Query OK, 0 rows affected (0.04 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> alter table p1 add index IX_name(name);
Query OK, 0 rows affected (0.07 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> insert into p1 values(1,'2009-10-1','zhangsan');
Query OK, 1 row affected (0.05 sec)

mysql> insert into p1 values(2,'2010-05-05','lisi');
Query OK, 1 row affected (0.01 sec)

mysql> insert into p1 values(3,'2011-07-08','wangwu');
Query OK, 1 row affected (0.00 sec)

mysql> insert into p1 values(4,'2012-04-27','xuliu');
Query OK, 1 row affected (0.00 sec)

mysql> insert into p1 values(5,'2013-02-14','zhaoqi');
Query OK, 1 row affected (0.01 sec)

mysql> select * from p1;
+----+-----+-----+
| id | date           | name    |
+----+-----+-----+
| 1  | 2009-10-01 00:00:00 | zhangsan |
| 2  | 2010-05-05 00:00:00 | lisi     |
| 3  | 2011-07-08 00:00:00 | wangwu   |
| 4  | 2012-04-27 00:00:00 | xuliu    |
| 5  | 2013-02-14 00:00:00 | zhaoqi   |
+----+-----+-----+
5 rows in set (0.01 sec)
```

图5-112 插入数据



```
mysql> explain partitions select * from p1 where (`date` between '2009-1-1' and '2009-12-31') and name = 'zhangsan';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | p1 | p0 | ref | IX_name | IX_name | 12 | const | 1 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain partitions select * from p1 where (`date` between '2010-1-1' and '2010-12-31') and name = 'lisi';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | p1 | p1 | ref | IX_name | IX_name | 12 | const | 1 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图5-113 已使用到分区

注意，在使用分区时，where后面的字段必须是分区字段，这样才能使用到分区，在图5-113中，2009年使用的分区是p0，2010年使用的分区是p1，那么如果去掉'date'字段，直接写name='zhaoqi'，能行吗？来看看图5-114。

```
mysql> explain partitions select * from p1 where name='zhaoqi';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | p1 | p0,p1,p2,p3 | ref | IX_name | IX_name | 12 | const | 2 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

图5-114 未使用到分区

这里扫描了全部的分区，分区在这里没有一点意义，反而拖累了性能。  
所以说SQL语句一定要带着分区的字段，否则会在所有的分区里全部扫描一遍。

## 5.7 MySQL5.6同步复制新特性详解

继MySQL5.5实现半同步复制后，MySQL5.6又对其进行了优化与改进，其中有两个地方较为重要：

·第一点对运维人员来说应该是一件大喜事，主从切换后，在传统的方式里，需要找到binlog和POS点，然后change master to指向，对于不是很有经验的运维人员来说，往往会找错，造成主从同步复制报错，在MySQL5.6里，无须再找binlog和POS点，你只需要知道master的IP、端口，账号密码即可，因为同步复制是自动的，MySQL会通过内部机制GTID自动找点同步。

·多线程复制基于库。在之前的版本里，同步复制是单线程的、队列的，只能一个个执行，在MySQL5.6里，可以做到多个库之间的多线程复制，例如，在yourDB库里，存放着用户表、商品表、价格表、订单表，那么将每个业务表单独放在一个库里，这时就可以做到多线程复制，但一个库里的表，多线程复制是无效的。

在MySQL5.6里，会涉及一些新名词，下面针对这些新名词进行解释：

·server\_uuid：服务器身份ID。在第一次启动MySQL时，会自动生成一个server\_uuid并写入到数据目录下auto.cnf文件里，官方不建议修改。并且server\_uuid跟GTID有密切联系。

下面是auto.cnf文件内容：

```
[root@mysql5_6 data]# pwd
/usr/local/mysql/data
[root@mysql5_6 data]# cat auto.cnf
[auto]
server-uuid=b0869d03-d4a9-11e1-a2ee-000c290a6b8f
```

·GTID：全局事务标识符。使用这个功能时，每次事务提交都会在binlog里生成一个唯一的标示符，它由UUID和事务ID组成。首次提交的事务ID为1，第二次为2，第三次为3，依次类推。

对于事物ID，查看binlog，会看到如图5-115~图5-118所示的内容。

```
# at 276
#120808 3:31:57 server id 25  end_log_pos 320  GTID [commit=yes]
SET @@SESSION.GTID_NEXT= 'B0869D03-D4A9-11E1-A2EE-000C290A6B8F:1' /*!*/;
```

图5-115 GTID:1

```
# at 496
#120808 3:39:36 server id 25  end_log_pos 540  GTID [commit=yes]
SET @@SESSION.GTID_NEXT= 'B0869D03-D4A9-11E1-A2EE-000C290A6B8F:2' /*!*/;
```



图5-116 GTID:2

```
# at 710
#120808 5:19:07 server id 25  end_log_pos 754  GTID [commit=yes]
SET @@SESSION.GTID_NEXT= 'B0869D03-D4A9-11E1-A2EE-000C290A6B8F:3' /*!*/;
```

图5-117 GTID:3

```
# at 912
#120808 5:19:07 server id 25  end_log_pos 956  GTID [commit=yes]
SET @@SESSION.GTID_NEXT= 'B0869D03-D4A9-11E1-A2EE-000C290A6B8F:4' /*!*/;
```

图5-118 GTID:4

开启GTID时，slave做同步复制的时候，无须找到binlog日志和POS点，直接change master to master\_auto\_position=1即可，它会自动找点同步。

GTID的工作流程是这样的：

1) 在master上一个事务提交，并写入binlog里。

2) binlog日志发送到slave，slave接收完并写入relay log中继日志里，slave读取到这个GTID，并设置gtid\_next的值。例如：  
`SET @@SESSION.GTID_NEXT='B0869D03-D4A9-11E1-A2EE-000C290A6B8F:3';`

然后告诉slave接下来的事务必须使用GTID，并写入到它自己的binlog里。

3) slave检查并确认这个GTID没有被使用，如果没有被使用，那么开始执行这个事务并写入到它自己的binlog里。

4) 由于gtid\_next的值不是空的，slave不会尝试去生成一个新的gtid，而是通过主从同步来获取GTID。

那么，如何设置GTID方式的主从同步呢？

答：在master和slave上，需要同时在my.cnf文件中加入如下内容：

`log-bin = mysql-bin`

`binlog_format = row`

`log_slave_updates`

`gtid-mode = ON`

`disable-gtid-unsafe-statements = ON`注：MySQL5.6.10以后，`disable-gtid-unsafe-statements`参数变更为`enforce-gtid-consistency`  
`= ON`

然后，在master上导出：



```
mysqldump -uroot -p123456 -q --single-transaction -R -E --triggers  
--default-character-set=utf8 -B yourDB >./yourDB.sql
```

在slave上导入：

```
mysql -uroot -p123456 <./yourDB.sql
```

之后做指向即可，如下所示：

```
mysql> CHANGE master TO  
>master_HOST = master-host ,  
>master_PORT = master-port ,  
>master_USER = repl-user ,  
>master_PASSWORD = repl-password ,  
>master_AUTO_POSITION = 1;
```



注意 如果使用了GTID，那么就不能再使用传统binlog和POS方式。

传统change master to模式如下：

```
CHANGE MASTER TO  
MASTER_HOST='master2.mycompany.com',  
MASTER_USER='replication',  
MASTER_PASSWORD='bigs3cret',  
MASTER_PORT=3306 ,  
MASTER_LOG_FILE='master2-bin.001',  
MASTER_LOG_POS=4 ,  
MASTER_CONNECT_RETRY=10;
```

否则会报错，如图5-119所示。

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.8.25', MASTER_USER='repl',
    -> MASTER_PASSWORD='repl', MASTER_LOG_FILE='mysql-bin.000001',
    -> MASTER_LOG_POS=120;
ERROR 1776 (HY000): Parameters MASTER_LOG_FILE, MASTER_LOG_POS, RELAY_LOG_FILE and RELAY_LOG_POS cannot be set when MASTER_AUTO_POSITION is active.
mysql>
```

图5-119 不支持传统change master to模式

GTID的局限性：

- GTID同步复制是基于事务的。所以MyISAM表不支持，这可能导致多个GTID分配给同一个事务。
- 对CREATE TABLE...SELECT语句不支持。因为该语句会被拆分成create table和insert两个事务，并且如果这两个事务被分配了同一个GTID，将会导致insert被备库忽略掉（如图5-120所示）。

```
mysql> create table t2 select * from t1;
ERROR 1786 (HY000): CREATE TABLE ... SELECT is forbidden when DISABLE_GTID_UNSAFE_STATEMENTS = 1.
mysql>
```

图5-120 不支持CREATE TABLE...SELECT语句

- 如果把disable\_gtid\_unsafe\_statements参数关闭，启动mysql时会报错，也就是说开启GTID时，disable\_gtid\_unsafe\_statements参数必须开启（如图5-121所示）。

```
m2 mysqld: 131227 15:02:55 [ERROR] --gtid-mode=ON or UPGRADE_STEP_1 requires --disable-gtid-unsafe-statements
m2 mysqld: 131227 15:02:55 [ERROR] Aborting
```

图5-121 未开启disable\_gtid\_unsafe\_statements参数报错信息

- 不支持CREATE TEMPORARY TABLE、DROP TEMPORARY TABLE临时表操作（如图5-122所示）。

```
mysql> create temporary table t2(id int);
ERROR 1787 (HY000): When DISABLE_GTID_UNSAFE_STATEMENTS = 1, the statements CREATE TEMPORARY TABLE and DROP TEMPORARY TABLE can be executed in a non-transactional context only, and require that AUTOCOMMIT = 1.
mysql>
```

图5-122 不支持创建临时表

下面介绍一下多线程复制基于库的情况。

设置slave\_parallel\_workers参数，开启基于库的多线程复制。默认是0，不开启，最大并发数为1024个线程。

对于命令“set global slave\_parallel\_workers=4;”，当设置为4个线程时，执行命令show processlist，你会发现有4个Waiting for an



event from Coordinator线程 (如图5-123所示)。

```
mysql> show processlist;
```

Id	User	Host	db	Command	Time	State
15	system user		NULL	Connect	211	Waiting for an event from Coordinator
14	system user		NULL	Connect	211	Waiting for an event from Coordinator
13	system user		NULL	Connect	212	Waiting for an event from Coordinator
12	system user		NULL	Connect	212	Waiting for an event from Coordinator
11	system user		NULL	Connect	211	Slave has read all relay log; waiting for the slave I/O thread to update it
10	system user		NULL	Connect	212	Connecting to master
4	root	localhost	NULL	Sleep	299	
22	root	localhost	NULL	Query	0	init

```
show processlist
```

8 rows in set (0.00 sec)

图5-123 状态信息

下面将用Sysbench分别对两个库进行压力测试，如下所示：

```
sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=100000  
--max-requests=1000 --num-threads=16 --mysql-host=localhost --mysql-port=3306 --mysql-user=root --mysql-db=test  
--mysql-socket=/tmp/mysql.sock run  
sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=100000  
--max-requests=1000 --num-threads=16 --mysql-host=localhost --mysql-port=3306 --mysql-user=root --mysql-db=test1  
--mysql-socket=/tmp/mysql.sock run
```

然后调用命令 "select\*from mysql.slave\_worker\_info\G;"，你会发现：



```
mysql>select * from mysql.slave_worker_info\G;
***** 1. row *****
      master_id: 165
      Worker_id: 0
      Relay_log_name:
      Relay_log_pos: 0
      master_log_name:
      master_log_pos: 0
      Checkpoint_relay_log_name:
      Checkpoint_relay_log_pos: 0
      Checkpoint_master_log_name:
      Checkpoint_master_log_pos: 0
      Checkpoint_seqno: 0
      Checkpoint_group_size: 64
      Checkpoint_group_bitmap:
***** 2. row *****
      master_id: 165
      Worker_id: 1
      Relay_log_name:
      Relay_log_pos: 0
      master_log_name:
      master_log_pos: 0
      Checkpoint_relay_log_name:
      Checkpoint_relay_log_pos: 0
      Checkpoint_master_log_name:
      Checkpoint_master_log_pos: 0
      Checkpoint_seqno: 0
      Checkpoint_group_size: 64
      Checkpoint_group_bitmap:
***** 3. row *****
```

```

        master_id: 165
        Worker_id: 2
        Relay_log_name: ./mysql5_6-relay-bin.000009
        Relay_log_pos: 2091034
        master_log_name: mysql-bin.000003
        master_log_pos: 2090832
Checkpoint_relay_log_name: ./mysql5_6-relay-bin.000009
Checkpoint_relay_log_pos: 2082941
Checkpoint_master_log_name: mysql-bin.000003
Checkpoint_master_log_pos: 2082739
Checkpoint_seqno: 5
Checkpoint_group_size: 64
Checkpoint_group_bitmap: 0
***** 4. row *****
        master_id: 165
        Worker_id: 3
        Relay_log_name: ./mysql5_6-relay-bin.000009
        Relay_log_pos: 2954634
        master_log_name: mysql-bin.000003
        master_log_pos: 2954432
Checkpoint_relay_log_name: ./mysql5_6-relay-bin.000009
Checkpoint_relay_log_pos: 2951772
Checkpoint_master_log_name: mysql-bin.000003
Checkpoint_master_log_pos: 2951570
Checkpoint_seqno: 1
Checkpoint_group_size: 64
Checkpoint_group_bitmap:
4 rows in set (0.01 sec)

```

可以看到，有两个Worker\_id的数值在不断变化，也就是说多线程复制就开始起作用了。





注意 如果一个库有N个请求，那么不会使用多线程复制，但是如果两个库有N个请求，那么会使用多线程复制。尽可能地把一个库中的表按照业务逻辑拆分成多个库保存，这样在写操作时，slave上就会开启多线程复制，减少了同步延时。2个库slave就有2个IO/SQL线程，3个库slave就有3个IO/SQL线程，依次类推。

此外，

```
relay_log_info_repository=TABLE
```

```
master_info_repository=TABLE
```

会将master.info和relay.info保存在表中，默认是MyISAM引擎，官方建议用：

```
alter table slave_master_info engine=innodb;
```

```
alter table slave_relay_log_info engine=innodb;
```

```
alter table slave_worker_info engine=innodb;
```

在改为InnoDB引擎后，可防止表损坏，在损坏后也可自行修复。

1. MySQL 5.6 GTID模式，同步复制报错不能跳过解决方法

假设同步复制报错了，信息如下：

```
mysql> show slave status\G;
```

```
***** 1. row *****
```

```
slave_IO_State: Waiting for master to send event
```

```
master_Host: 192.168.8.25
```

```
master_User: repl
```

```
master_Port: 3306
```

```
Connect_Retry: 60
```

```
master_Log_File: mysql-bin.000001
```

```
Read_Master_Log_Pos: 552
```

```
Relay_Log_File: M2-relay-bin.000002
```

```
Relay_Log_Pos: 519
```

```
Relay_Master_Log_File: mysql-bin.000001
```

```
slave_IO_Running: Yes
```





```

slave_SQL_Running: No
Replicate_Do_DB:
Replicate_Ignore_DB: mysql
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 1032
Last_Error: Could not execute Delete_rows event on table test.test; Can't find record in 'test',
Error_code: 1032; handler error HA_ERR_KEY_NOT_FOUND; the event's master log mysql-bin.000001, end_log_pos 521
Skip_Counter: 0
Exec_master_Log_Pos: 309
Relay_Log_Space: 963
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
master_SSL_Allowed: No
master_SSL_CA_File:
master_SSL_CA_Path:
master_SSL_Cert:
master_SSL_Cipher:
master_SSL_Key:
Seconds_Behind_master: NULL
master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 1032
Last_SQL_Error: Could not execute Delete_rows event on table test.test; Can't find record in 'test',
Error_code: 1032; handler error HA_ERR_KEY_NOT_FOUND; the event's master log mysql-bin.000001, end_log_pos 521
Replicate_Ignore_Server_Ids:

```

```

master_Server_Id: 25
master_UUID: cf716fda-74e2-11e2-b7b7-000c290a6b8f
master_Info_File: /usr/local/mysql/data/master.info
SQL_Delay: 0
SQL_Remaining_Delay: NULL
slave_SQL_Running_State:
master_Retry_Count: 86400
master_Bind:
Last_IO_Error_Timestamp:
Last_SQL_Error_Timestamp: 130611 23:07:02
master_SSL_Crl:
master_SSL_Crlpath:
Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
Executed_Gtid_Set: 562935a3-74f5-11e2-b830-000c29ba57f2:1-3 ,
cf716fda-74e2-11e2-b7b7-000c290a6b8f:1
Auto_Position: 1
1 row in set (0.02 sec)

```

这里提示删除的主键不存在同步报错，由于是测试机，于是直接跳过：

```
mysql> set global sql_slave_skip_counter=1;
```

ERROR 1858 (HY000): sql\_slave\_skip\_counter can not be set when the server is running with

GTID\_MODE = ON. Instead, for each transaction that you want to skip, generate an empty transaction with the same GTID as the transaction



提示 由于是运行在GTID模式下，所以不支持sql\_slave\_skip\_counter语法，如果你想跳过，就必须把事务ID设置为空值。

```
Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
```

```
Executed_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1
```

根据在show slave status\G得到的信息，观察Retrieved\_Gtid\_Set和Executed\_Gtid\_Set这两行内容，第一行代表接收到的事务，第二行代表已经执行完的事务。也就是说在执行cf716fda-74e2-11e2-b7b7-000c290a6b8f:2这个事务时报错了，这时，只需跳过这个错误事务就可以



了, 如下所示:

```
mysql> stop slave;  
Query OK, 0 rows affected (0.07 sec)  
mysql> SET GTID_NEXT='cf716fda-74e2-11e2-b7b7-000c290a6b8f:2';  
Query OK, 0 rows affected (0.01 sec)  
mysql> begin;commit;  
Query OK, 0 rows affected (0.01 sec)  
Query OK, 0 rows affected (0.02 sec)  
mysql> SET GTID_NEXT="AUTOMATIC";  
Query OK, 0 rows affected (0.02 sec)  
mysql> start slave;  
Query OK, 0 rows affected (0.10 sec)
```

然后执行 "show slave status\G;" 命令确认一下:

```
mysql> show slave status\G;  
***** 1. row *****  
      slave_IO_State: Waiting for master to send event  
      master_Host: 192.168.8.25  
      master_User: repl  
      master_Port: 3306  
      Connect_Retry: 60  
      master_Log_File: mysql-bin.000001  
      Read_master_Log_Pos: 552  
      Relay_Log_File: M2-relay-bin.000003  
      Relay_Log_Pos: 448  
      Relay_master_Log_File: mysql-bin.000001  
      slave_IO_Running: Yes  
      slave_SQL_Running: Yes  
      Replicate_Do_DB:  
      Replicate_Ignore_DB: mysql  
      Replicate_Do_Table:
```



*Replicate\_Ignore\_Table:*  
*Replicate\_Wild\_Do\_Table:*  
*Replicate\_Wild\_Ignore\_Table:*  
    *Last\_Errno: 0*  
    *Last\_Error:*  
    *Skip\_Counter: 0*  
    *Exec\_master\_Log\_Pos: 552*  
    *Relay\_Log\_Space: 1260*  
    *Until\_Condition: None*  
    *Until\_Log\_File:*  
    *Until\_Log\_Pos: 0*  
    *master\_SSL\_Allowed: No*  
    *master\_SSL\_CA\_File:*  
    *master\_SSL\_CA\_Path:*  
    *master\_SSL\_Cert:*  
    *master\_SSL\_Cipher:*  
    *master\_SSL\_Key:*  
    *Seconds\_Behind\_master: 0*  
*master\_SSL\_Verify\_Server\_Cert: No*  
    *Last\_IO\_Errno: 0*  
    *Last\_IO\_Error:*  
    *Last\_SQL\_Errno: 0*  
    *Last\_SQL\_Error:*

*Replicate\_Ignore\_Server\_Ids:*  
    *master\_Server\_Id: 25*  
    *master\_UUID: cf716fda-74e2-11e2-b7b7-000c290a6b8f*  
    *master\_Info\_File: /usr/local/mysql/data/master.info*  
    *SQL\_Delay: 0*  
    *SQL\_Remaining\_Delay: NULL*  
*slave\_SQL\_Running\_State: slave has read all relay log; waiting for the slave I/O thread to update it*



```

        master_Retry_Count: 86400
        master_Bind:
        Last_IO_Error_Timestamp:
        Last_SQL_Error_Timestamp:
        master_SSL_Crl:
        master_SSL_Crlpath:
        Retrieved_Gtid_Set: cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
        Executed_Gtid_Set: 562935a3-74f5-11e2-b830-000c29ba57f2:1-3 ,
        cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2
        Auto_Position: 1

```

1 row in set (0.02 sec)

OK! 已经跳过去了，同步复制正常。

## 2. MySQL5.6 GTID模式转为传统模式

要将MySQL5.6 GTID模式转为传统模式，首先要在my.cnf配置文件里注释掉以下参数，并重启MySQL数据库：

```
#gtid-mode = ON
```

```
#enforce-gtid-consistency = ON
```

由于之前是GTID同步复制模式，现在要转为传统模式，因此会报错，如下所示：

```

mysql> CHANGE master TO
MASTER_HOST='192.168.8.25', master_USER='repl',
->MASTER_PASSWORD='repl', master_LOG_FILE='mysql-bin.000001',
->MASTER_LOG_POS=120;
ERROR 1776 (HY000): Parameters master_LOG_FILE, master_LOG_POS,
RELAY_LOG_FILE and RELAY_LOG_POS cannot be set when
MASTER_AUTO_POSITION is active.

```

那么，我们要使用一个技巧：

```

mysql> change master to MASTER_AUTO_POSITION = 0;
Query OK, 0 rows affected (0.19 sec)
mysql> CHANGE MASTER TO
MASTER_HOST='192.168.8.25', MASTER_USER='repl', MASTER_PASSWORD='repl', MASTER_LOG_FILE='mysql-bin.000001',

```

*MASTER\_LOG\_POS=120;*

*Query OK, 0 rows affected, 2 warnings (0.16 sec)*

这样就顺利执行了。







## 第6章

### 备份与恢复



随着办公自动化和电子商务的飞速发展，企业对信息系统的依赖性越来越高，数据库作为信息系统的核心担当着重要的角色。尤其是在一些对数据可靠性要求很高的行业如银行、证券、电信等，如果发生意外停机或数据丢失，损失会十分惨重。对此，数据库管理员应针对具体的业务要求制定详细的数据库备份与灾难恢复策略，并通过模拟故障对每种可能的情况进行严格测试，只有这样才能保证数据的高可用性。数据库的备份是一个长期的过程，而恢复只在发生事故后进行，恢复可以看做是备份的逆过程，恢复程度的好坏很大程度上依赖于备份的情况。此外，数据库管理员在恢复时采取的步骤正确与否也会直接影响最终的恢复结果。

MySQL备份种类可分为两种：完全备份和增量备份。

完全备份，是指对某一个时间点上的所有数据或应用进行的一个完全复制。实际应用中就是用一盘磁带对整个系统进行完全备份，包括其中的系统和所有的数据。这种备份方式最大的好处就是只要用一盘磁带，就可以恢复丢失的数据。因此大大加快了系统或数据的恢复时间。然而它的不足之处在于，各个全备份磁带中的备份数据存在大量的重复信息；另外，由于每次需要备份的数据量相当大，因此备份所需的时间也较长。

增量备份，是指在一次全备份或上一次增量备份后，以后每次的备份只需备份与前一次相比增加或被修改的binlog文件。这就意味着，第一次增量备份的对象是进行全备后又增加和修改的binlog文件；第二次增量备份的对象是进行第一次增量备份后所增加和修改的binlog文件，依次类推。这种备份方式最显著的优点就是：没有重复的备份数据，因此备份的数据量不大，备份所需的时间很短。但增量备份的数据恢复是比较麻烦的。您必须具有上一次全备份和所有增量备份的磁带（一旦丢失或损坏其中的一盘磁带，就会造成恢复的失败），并且它们必须沿着从全备份到依次增量备份的时间顺序逐个反推恢复，因此这就极大地延长了恢复时间。

按照备份方式可分为三种：冷备份、热备份和逻辑备份。

·冷备份，此时数据库处于关闭状态，能够较好地保证数据库的完整性。

·热备份，数据库正处于运行状态，这种备份方法依赖于数据库的日志文件。

·逻辑备份，使用mysqldump命令从数据库中提取数据，并将结果写到一个文件上，文件内容为纯文本的SQL语句。

一般情况，在生产环境中会将MySQL配置为一主一从，为了避免影响业务，建议在slave机器上做备份，下面依次介绍一下上面介绍的这三种备份与恢复方式。



## 6.1 冷备份

冷备份一般用于非核心业务，这类业务一般都允许中断，冷备份的特点是速度快，恢复时也最为简单。通常通过直接复制物理文件来实现冷备份。

### 1. 备份过程

第一步，关闭MySQL服务进程。命令如下：

```
/etc/init.d/mysql stop
```

第二步，把data数据目录（包含ibdata1）和日志目录（包含ib\_logfile0，ib\_logfile1，ib\_logfile2）复制到磁带机或者本地的另一块硬盘里。

### 2. 恢复过程

第一步，用复制的数据目录和日志目录替换原有的目录。

第二步，启动MySQL服务进程。命令如下：

```
/etc/init.d/mysql start
```



## 6.2 逻辑备份

逻辑备份一般用于数据迁移或者数据量很小时，逻辑备份采用的是数据导出的备份方式。

### 1. 备份过程

如果需要导出所有数据库，命令如下：

```
mysqldump -q --single-transaction -A >all.sql
```

如果只是要导出其中的某几个数据库，则采用如下命令：

```
mysqldump -q --single-transaction -B test1 test2 >test1_test2.sql
```

如果要导出的是一个库中的某几个表，可采用如下命令：

```
mysqldump -q --single-transaction -B test1 test2 >test1_test2.sql
```

在只需要导出表结构的时候，采用如下命令：

```
mysqldump -q -d --skip-triggers
```

在只需要导出存储过程的时候，采用如下命令：

```
mysqldump -q -Rtdn --skip-triggers
```

如果只需要导出触发器，可采用如下命令：

```
mysqldump -q -tdn --triggers
```

只需要导出事件时，采用如下命令：

```
mysqldump -q -Etdn --skip-triggers
```

只需要导出数据时，采用如下命令：

```
mysqldump -q -single-transaction --skip-triggers -t
```

要想在线建立一台新的slave，请采用如下命令：

```
mysqldump -q --single-transaction --master-data=2 -A >all.sql
```

### 2. 恢复过程

通过以下命令来实现：

```
MySQL -uroot -p123456 <all.sql
```

或者登录到MySQL里，执行“source all.sql;”。

### 6.2.1 mysqldump增加了一个重要参数

在MySQL5.5里，新增加了一个重要参数，即--dump-slave，使用该参数可在slave端dump数据，建立新的slave，其目的是为了防止对主库造成过大压力。

下面来查看同步复制信息，得到复制的binlog和POS点，注意其中的加粗字体：

```
mysql> show slave status\G;
```

```
***** 1. row *****
```

```
    slave_IO_State: Waiting for master to send event
```

```
      master_Host: 192.168.110.216
```

```
      master_User: repl
```

```
      master_Port: 3306
```

```
    Connect_Retry: 10
```

```
    master_Log_File: mysql-bin.002810
```

```
Read_master_Log_Pos: 16550261
```

```
    Relay_Log_File: relay-bin.000419
```

```
    Relay_Log_Pos: 252
```

```
Relay_master_Log_File: mysql-bin.002810
```

```
    slave_IO_Running: Yes
```

```
    slave_SQL_Running: Yes
```

```
    Replicate_Do_DB: WATCDB01 , WATCDB02 , WATCDB03 , WATCDB04
```

```
    Replicate_Ignore_DB:
```

```
    Replicate_Do_Table:
```

```
    Replicate_Ignore_Table:
```

```
    Replicate_Wild_Do_Table:
```

```
    Replicate_Wild_Ignore_Table:
```

```
      Last_Errno: 0
```

```
      Last_Error:
```



```

Skip_Counter: 0
Exec_master_Log_Pos: 16550261
Relay_Log_Space: 547
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
master_SSL_Allowed: No
master_SSL_CA_File:
master_SSL_CA_Path:
master_SSL_Cert:
master_SSL_Cipher:
master_SSL_Key:
Seconds_Behind_master: 0
master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 0
Last_SQL_Error:
Replicate_Ignore_Server_Ids:
master_Server_Id: 2163306

```

1 row in set (0.00 sec)

然后把数据导出来，如下所示：

```

[root@MySQL5 ~]# /usr/local/mysql/bin/mysqldump -A --dump-slave=2 -q --single-transaction >/u1/all.sql
^C
[root@MySQL5 ~]# more /u1/all.sql
-- MySQL dump 10.13 Distrib 5.5.20 , for linux2.6 (x86_64)
--
-- Host: localhost Database:
--
-----
-- Server version 5.5.20-enterprise-commercial-advanced-log

```



...

-- *Position to start replication or point-in-time recovery from (the master of this slave)*

--

-- *CHANGE master TO master\_LOG\_FILE='mysql-bin.002810',  
master\_LOG\_POS=16550261;*

这里会记录slave的那个点，注意CHANGE master相关的加粗字体。

注：--dump-slave用于在slave端dump数据，建立新的slave。



## 6.2.2 取代mysqldump的新工具mydumper

mydumper

注

是一个针对MySQL和Drizzle的高性能多线程的备份和恢复工具。此工具的开发人员分别来自MySQL、Facebook、SkySQL公司，目前已经有一些大型产品业务测试并使用了该工具。我们在恢复数据库时也可使用myloader工具。

Mydumper的主要特性包括：

- 采用轻量级C语言写的代码。
- 相比于mysqldump，其速度快了近10倍。
- 具有事务性和非事务性表一致的快照（适用于0.2.2+）。
- 可快速进行文件压缩（File compression on-the-fly）。
- 支持导出binlog。
- 可多线程恢复（适用于0.2.1+）。
- 可以用守护进程的工作方式，定时扫描和输出连续的二进制日志。

了解了mydumper的特色后，我们再来看看它的安装方法（在CentOS 6.0上测试），安装命令如下所示：

```
# wget http://launchpadlibrarian.net/77098505/mydumper-0.5.2.tar.gz
# yum install glib2-devel mysql-devel zlib-devel pcre-devel
# tar -xzvf mydumper-0.5.2.tar.gz
# cd mydumper-0.5.2
# cmake .
# make;make install
```

Mydumper中的主要参数如下：

- host, -h：连接的MySQL服务器。
- user, -u：用户备份的连接用户。
- password, -p：用户的密码。
- port, -P：连接端口。
- socket, -S：连接socket文件。
- database, -B：需要备份的数据库。



- table-list, -T：需要备份的表，用逗号（，）分隔。
- outputdir, -o：输出的目录。
- build-empty-files, -e：默认无数据则只有表结构文件。
- regex, -x：支持正则表达式，如mysqldumper-regex' ^(?!(mysql|test))'。
- ignore-engines, -i：忽略的存储引擎。
- no-schemas, -m：不导出表结构。
- long-query-guard：长查询，默认60s。
- kill-long-queries, -k：可以设置kill长查询。
- verbose, -v：0=silent, 1=errors, 2=warnings, 3=info，默认是2。
- binlogs, -b：导出binlog。
- daemon, -D：启用守护进程模式。
- snapshot-interval, -l：dump快照间隔时间，默认60s。
- logfile, -L：mysqldumper的日志输出，一般在Daemon模式下使用。

myloader的大多数参数和mysqldumper一样，不同之处如下：

- directory, -d：要还原的数据目录。
- overwrite-tables, -o：指定导出的目录。

现在，可以通过对mysqldump和mysqldumper的对比测试来了解它的性能。先来看看测试环境，XXDB数据库的大小如图6-1所示。

TABLE_SCHEMA	DATA_LENGTH	INDEX_LENGTH	DATA_INDEX
XXDB	3.463668929413	1.068885803223	4.532554732636

图6-1 数据库的大小统计信息

测试过程如下：

用mysqldump导出该XXDB数据库，耗时2分钟16秒，如图6-2所示。

```
[root@XX-XX-01 ~]# time mysqldump -q --single-transaction -B XXDB > /bak/XXDB.sql

real    2m16.068s
user    1m14.014s
sys     0m18.147s
```



图6-2 mysqldump导出数据耗时

用mydumper导出该XXDB数据库，耗时1分钟36秒，如图6-3所示。

```
[root@192-168-01 1]# time mydumper -u root -p 123456 -B XXDB -o /bak/1/

real    1m36.137s
user    0m56.033s
sys     0m14.839s
```

图6-3 mydumper导出数据耗时

下面再来看一下两种工具导入的耗时情况。

用myloader导入该XXDB数据库，耗时12分钟6秒，如图6-4所示。

```
[root@192-168-01 1]# time myloader -u root -p 123456 -d /bak/1/ -o

real    12m6.779s
user    0m22.541s
sys     0m10.304s
```

图6-4 myloader导入耗时

用自带的MySQL导入该XXDB数据库，耗时25分钟4秒，如图6-5所示。

```
[root@192-168-01 bak]# time mysql < ./GRPDB.sql

real    25m4.070s
user    1m38.571s
sys     0m7.265s
```

图6-5 自带MySQL导入耗时

从上面的测试很明显可以看出，dump导出的时间，mydumper要比mysqldump快1倍。而导入的时间，myloader要比自带的MySQL快1倍。

• Mydumper工具的主页：<http://www.mydumper.org/>。

### 6.2.3 逻辑备份全量、增量备份脚本

在采用逻辑备份全量，增量备份脚本的备份方式时，一般情况下，星期一进行全量备份，星期二至星期五则进行增量备份。假设星期五的时候数据被破坏了，恢复时则需要还原星期一的全量备份和从星期二至星期五的所有增量备份。这种备份策略相对来说备份所需的时间较少，但还原数据则会使用较多的时间。

MySQL全量备份脚本如下：

```
#!/bin/bash
# MySQL全量备份脚本，建议在slave从库上运行，并开启log_slave_updates = 1
mkdir /backup
cd /backup
dateDIR='date +%y-%m-%d'
mkdir -p $dateDIR/data
path=/usr/local/mysql/data
for i in `mysql -uroot -p123456 -e "show databases" | grep -v "Database"`
do
mysqldump -uroot -p123456 --default-character-set=utf8 \
-q --lock-all-tables --flush-logs -E -R --triggers -B $i | gzip >
/backup/$dateDIR/data/${i}_`${dateDIR}.sql.gz
done
binlog_rm='tail -n 1 $path/mysql-bin.index | sed 's/.V//''
mysql -uroot -p123456 -e "purge binary logs to '$binlog_rm'"
```

这个全量备份脚本，会在导出的时候锁住全局表，并刷新产生一个新的binlog，期间会有写操作等待，直到导出结束后才会写入新产生的binlog里，然后旧的binlog会被删除掉。一般该备份放在凌晨1:00操作较合适。

MySQL增量备份脚本如下：

```
#!/bin/bash
# MySQL增量备份脚本，建议在slave从库上运行，并开启log_slave_updates = 1
cd /backup
```



```
dateDIR='date +%y-%m-%d'
mkdir -p $dateDIR/data
path=/usr/local/mysql/data
mysqladmin -uroot -p123456 flush-logs
binlog_cp='head -n -1 $path/mysql-bin.index | sed 's/.V///''
for i in $binlog_cp
do
mysql -uroot -p123456 -e " \! cp -p $path/$i /backup/$dateDIR/data/;"
done
binlog_rm='tail -n 1 $path/mysql-bin.index | sed 's/.V///''
mysql -uroot -p123456 -e "purge binary logs to '$binlog_rm'"
```

在执行全量备份脚本以后，就可以执行这个增量备份脚本了，首先会刷新产生一个新的binlog，然后把之前有变化的binlog复制到备份目录下，复制完以后就会把之前的旧binlog删除掉，期间若有写操作，也会写入到新的binlog里。



## 6.3 热备份与恢复

热备份的方式也是直接复制数据物理文件，和冷备份一样，但热备份可以不停机直接复制，一般用于7×24小时不间断的重要核心业务。MySQL社区版的热备份工具InnoDB Hot Backup是付费的，只能试用30天，只有购买企业版才可以得到永久使用权。Percona公司发布了一个xtrabackup热备份工具，和官方付费版的功能一样，支持在线热备份（备份时不影响数据读写），是商业备份工具InnoDB Hot Backup的一个很好的替代品。下面具体介绍一下这个软件的使用方法。

xtrabackup是Percona公司的开源项目，用以实现类似InnoDB官方的热备份工具InnoDB Hot Backup的功能，它能非常快速地备份与恢复MySQL数据库。xtrabackup中包含两个工具：

- xtrabackup是用于热备份InnoDB及XtraDB表中数据的工具，不能备份其他类型的表，也不能备份数据表结构。
- innobackupex是将xtrabackup进行封装的perl脚本，它提供了备份MyISAM表的能力。由于innobackupex的功能更为全面完善，所以一般选择innobackupex来进行备份。

下面来看看xtrabackup的安装方法，安装命令如下：

```
#wget http://www.percona.com/redirect/downloads/XtraBackup/LATEST/binary/Linux/i686/percona-xtrabackup-2.1.3-608.tar.gz
# tar zxvf percona-xtrabackup-2.1.3-608.tar.gz
# cd percona-xtrabackup-2.1.3/bin
[root@M1 bin]# ll -h
total 92M
-rwxr-xr-x 1 root root 109K Apr 11 05:51 innobackupex
lrwxrwxrwx 1 root root 12 Jun 12 15:26 innobackupex-1.5.1 ->innobackupex
-rwxr-xr-x 1 root root 2.0M Apr 11 05:51 xbcrypt
-rwxr-xr-x 1 root root 2.0M Apr 11 05:51 xbstream
-rwxr-xr-x 1 root root 9.8M Apr 11 05:51 xtrabackup
-rwxr-xr-x 1 root root 13M Apr 11 05:34 xtrabackup_55
-rwxr-xr-x 1 root root 66M Apr 11 05:44 xtrabackup_56# cp * /sbin/
```

说明：

- innobackupex是要使用的备份工具。
- xtrabackup是被封装在innobackupex之中的，innobackupex运行时需要调用它。
- xtrabackup\_51 xtrabackup\_55是xtrabackup运行时需要调用的工具。



tar4ibd是以tar流的形式产生备份时用来打包的工具。  
再来了解一下Innobackupex参数，具体说明如图6-6所示。

-defaults-file	同 xtrabackup 的--defaults-file 参数；
-apply-log	对 xtrabackup 的--prepare 参数的封装；
-copy-back	做数据恢复时将备份数据文件拷贝到 MySQL 服务器的 datadir ；
-remote-host	通过 ssh 将备份数据存储在远程服务器上；
-stream	通过指定的数据格式将备份数据输出到标准输出；
-tmpdir	当指定了--remote-host 或--stream 参数后，事务日志需要临时存储在本地磁盘 此参数默认使用 MySQL 服务器的设置；
-use-memory	此参数是给 ibbackup 使用的，类似 xtrabackup 的--use-memory 参数；
-throttle=IOS	同 xtrabackup 的--throttle 参数；
-sleep	是给 ibbackup 使用的，指定每备份 1M 数据，进程停止拷贝多少毫秒，也是为了在备份时尽量减小对正常业务的影响，具体可以查看ibbackup 的手册；
-compress	对备份数据进行压缩，仅支持 ibbackup，xtrabackup 还没有实现；
-include=REGEXP	对 xtrabackup 参数--tables 的封装，也支持 ibbackup；
-databases=LIST	列出需要备份的 databases，如果没有指定该参数，所有包含MyISAM 和 InnoDB 表的 database 都会被备份；
-uncompress	解压备份的数据文件，支持 ibbackup，xtrabackup 还没有实现该功能；
-user=NAME	MySQL 数据库的用户名；
-password=WORD	MySQL 数据库的密码；
-port=PORT	MySQL 数据库监听的端口；
-slave-info	备份 slave 数据库时使用，会将 slave 的相关信息记录到ibbackup_slave_info 文件 便于恢复数据时使用 CHANGE MASTER TO 继续 slave 的同步；
-socket=SOCKET	MySQL 服务器的 socket 文件位置

图6-6 Innobackupex参数说明

### 1.全量备份（不加--databases，默认全部数据库）

进行全量备份的命令如下：

```
innobackupex --user=root --password=123456 --defaults-file=/etc/my.cnf /bak/
```

说明：

- user：指定连接数据库的用户名。
- password：指定连接数据库的密码。
- defaults-file：指定数据库的配置文件。
- /bak/：是备份文件的存放位置。

备份成功后会提示：

```
innobackupex: Backup created in directory '/bak/2013-06-12_15-40-00'  
innobackupex: MySQL binlog position: filename 'mysql-bin.000001', position 552, gtid_executed  
cf716fda-74e2-11e2-b7b7-000c290a6b8f:1-2  
130612 15:41:27 innobackupex: Connection to database server closed  
130612 15:41:27 innobackupex: completed OK!
```

并且会记录当前binlog的文件名和position点，以方便同步复制用。

## 2. 全量恢复

全量恢复的操作步骤如下：

1) 停止MySQL数据库。命令如下：

```
MySQL # /etc/init.d/mysql stop
```

2) 删除老数据库中的数据文件和事务日志文件。

注意：如果没有删除，恢复时将报错。报错信息如下：

```
Original data directory '/usr/local/mysql/data' is not empty! at /sbin/innobackupex line 582.
```

3) 将备份文件中的日志应用到备份文件中的数据文件上，命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --apply-log /bak/2013-06-12_15-40-00
```

事务日志恢复成功后会提示：

```
xtrabackup: starting shutdown with innodb_fast_shutdown = 1  
InnoDB: FTS optimize thread exiting.  
InnoDB: Starting shutdown...  
InnoDB: Shutdown completed; log sequence number 3143306391  
130612 15:51:28 innobackupex: completed OK!
```

4) 将备份文件中的数据恢复到数据库中，命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --copy-back /bak/2013-06-12_15-40-00/
```

物理文件和事务日志恢复成功后会提示：

```
innobackupex: Starting to copy InnoDB system tablespace  
innobackupex: in '/bak/2013-06-12_15-40-00'  
innobackupex: back to original InnoDB data directory '/usr/local/mysql/data'
```





```
innobackupex: Copying '/bak/2013-06-12_15-40-00/ibdata1' to
'/usr/local/mysql/data/ibdata1'
innobackupex: Starting to copy InnoDB undo tablespaces
innobackupex: in '/bak/2013-06-12_15-40-00'
innobackupex: back to '/usr/local/mysql/data'
innobackupex: Starting to copy InnoDB log files
innobackupex: in '/bak/2013-06-12_15-40-00'
innobackupex: back to original InnoDB log directory '/usr/local/mysql/data'
innobackupex: Copying '/bak/2013-06-12_15-40-00/ib_logfile1' to
'/usr/local/mysql/data'
innobackupex: Copying '/bak/2013-06-12_15-40-00/ib_logfile0' to
'/usr/local/mysql/data'
innobackupex: Finished copying back files.
130612 15:59:11 innobackupex: completed OK!
```

5) 数据恢复完成之后，需要修改相关文件的权限，命令如下：

```
chown -R mysql:mysql /usr/local/mysql/data/
```

6) 重新启动MySQL数据库，命令如下：

```
/etc/init.d/mysql start
```

3. 备份到远程服务器

假设：

本地服务器IP：192.168.8.25

目标服务器IP：192.168.8.26

采用如下命令进行备份：

```
innobackupex --defaults-file=/etc/my.cnf --stream=tar /usr/local/mysql/data | ssh
root@192.168.8.26 cat " > " /bak/backup.tar
```

备份成功后会提示：

```
innobackupex: Backup created in directory '/usr/local/mysql/data'
innobackupex: MySQL binlog position: filename 'mysql-bin.000001', position 151
130612 16:16:42 innobackupex: Connection to database server closed
```

*innobackupex: You must use -i (--ignore-zeros) option for extraction of the tar stream.*

*130612 16:16:42 innobackupex: completed OK!*

#### 4. 全量恢复

进行全量恢复时，其操作步骤基本和前面的普通备份恢复类似。

注意：恢复解压缩时，必须使用-i参数：

*tar -ixvf backup.tar*

解压后按普通备份恢复的步骤进行恢复即可。

#### 5. 增量备份

进行增量备份的前提是必须已经做过全量备份，步骤如下：

1) 先进行全量备份，命令如下：

*innobackupex --defaults-file=/etc/my.cnf /bak/fullbak/*

2) 再进行增量备份，命令如下：

*innobackupex --defaults-file=/etc/my.cnf --incremental /bak/incrementbak*

*--incremental-basedir=/bak/fullbak/2013-06-12\_16-46-36/*

备份成功后会提示：

*innobackupex: Backup created in directory '/bak/incrementbak/2013-06-12\_16-59-27'*

*innobackupex: MySQL binlog position: filename 'mysql-bin.000001', position 637, gtid\_executed c25abacf-d336-11e2-9ed0-000c290a6b8f:1-2*

*130612 16:59:56 innobackupex: Connection to database server closed*

*130612 16:59:56 innobackupex: completed OK!*

进入到备份目录，可以看到哪份是全量备份，哪份是增量备份，如下：

*[root@M1 bak]# cd fullbak/2013-06-12\_16-46-36/*

*[root@M1 2013-06-12\_16-46-36]# cat xtrabackup\_checkpoints backup\_type = full-backup*

*from\_lsn = 0*

*to\_lsn = 3143306401*

*last\_lsn = 3143306401*

*compact = 0*

*[root@M1 bak]# cd incrementbak/2013-06-12\_16-59-27 /*



[root@M1 2013-06-12\_16-59-27]#  
[root@M1 2013-06-12\_16-59-27]# cat xtrabackup\_checkpoints backup\_type = incremental  
from\_lsn = 3143306401  
to\_lsn = 3143308222  
last\_lsn = 3143308222  
compact = 0  
[root@M1 2013-06-12\_16-59-27]#

## 6. 增量恢复

进行增量恢复的步骤如下：

1) 先恢复增量事务日志，命令如下：

```
innobackupex --apply-log --redo-only /bak/fullbak/2013-06-12_16-46-36/  
--incremental-dir=/bak/incrementbak/2013-06-12_16-59-27/
```

2) 再恢复全量事务日志，命令如下：

```
innobackupex --apply-log /bak/fullbak/2013-06-12_16-46-36/
```

3) 将备份文件中的数据恢复到数据库中，命令如下：

```
innobackupex --copy-back /bak/fullbak/2013-06-12_16-46-36/
```

4) 数据恢复完成之后，需要修改相关文件的权限，命令如下：

```
chown -R mysql:mysql /usr/local/mysql/data/
```

5) 启动MySQL数据库，命令如下：

```
/etc/init.d/mysql start
```







在第一版中曾介绍过MMM ( Multi-Master Replication Manager ) 架构，随着越来越多的互联网金融公司使用MySQL数据库，对数据的一致性要求也越来越高，这便促成了MHA ( Master High Availability ) 架构的诞生。MHA的作者是Yoshinori Matsunobu，目前就职于Facebook，它与MMM架构都是采用Perl语言编写的，可在宕机的时间内（通常为10~30秒）完成故障切换。此外，还支持在线切换，从当前运行的master切换到另一个新的master上，只需要很短的时间（为0.5~2秒内），此时仅阻塞写操作，并不影响读操作，便于主机的硬件维护，现在国内互联网公司大多采用这种高可用MHA架构。

## 7.1 MHA架构简介

MHA架构和MMM架构有什么区别呢？最大的区别在于：MHA会把丢失的数据在每个save节点上补齐。下面通过一幅图来了解MHA故障转移的工作原理，如图7-1所示。

从图7-1可以看到，当master宕机时，MHA管理机会试图scp丢失的那一部分binlog，然后把该binlog拷贝到最新的slave机器上，再补齐差异的binlog并应用。当最新的slave补齐数据后，再把它的relay-log拷贝到其他的slave上，以识别差异并应用。至此，整个恢复过程结束，从而保证切换后的数据是一致的。

下面通过图7-2更容易理解整个恢复过程。

MHA官网网站为：<https://code.google.com/p/mysql-master-ha/>。

MHA提供了三种故障转移模式，下面分别说明各自的使用场景。



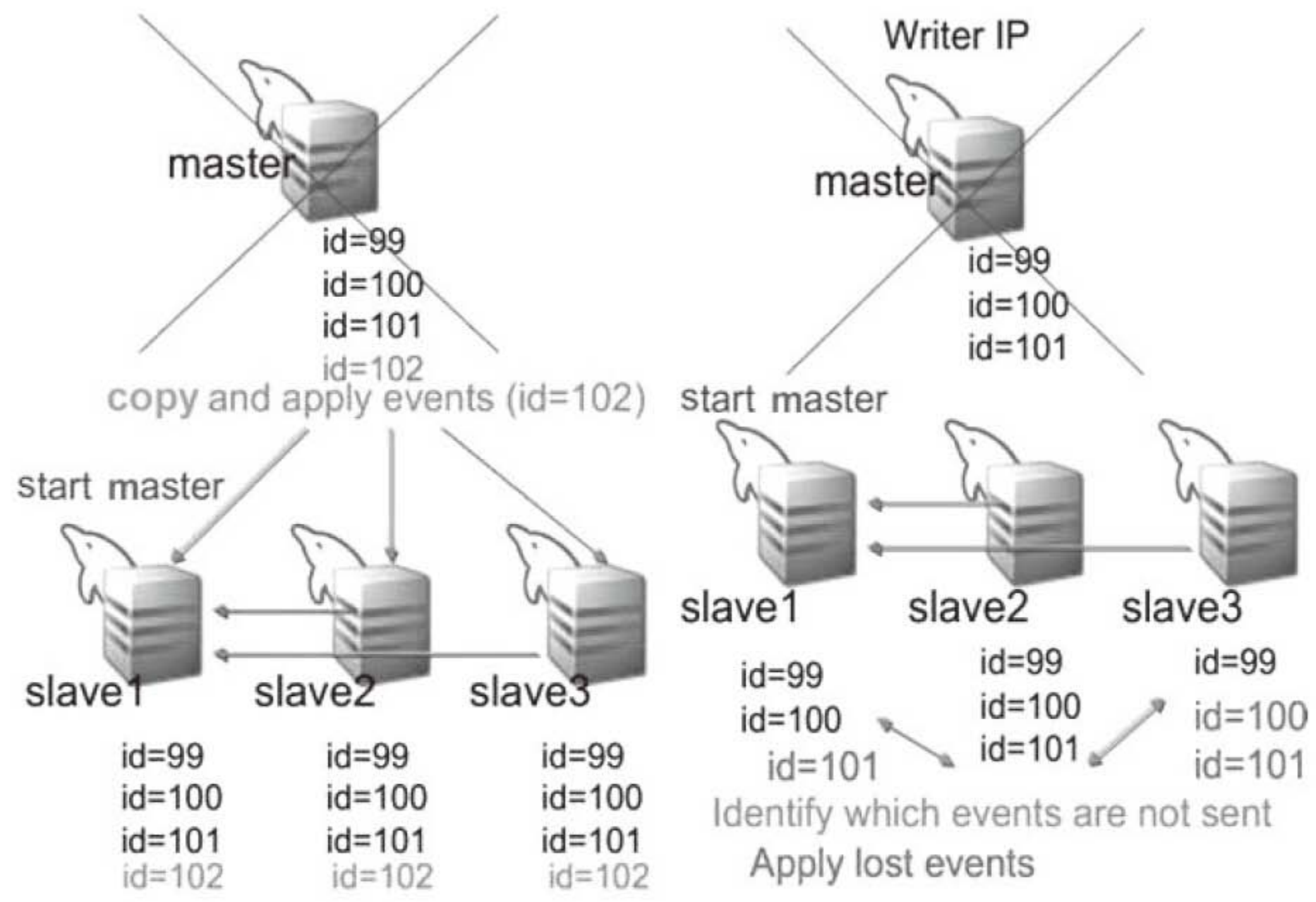


图7-1 MHA故障转移的工作原理

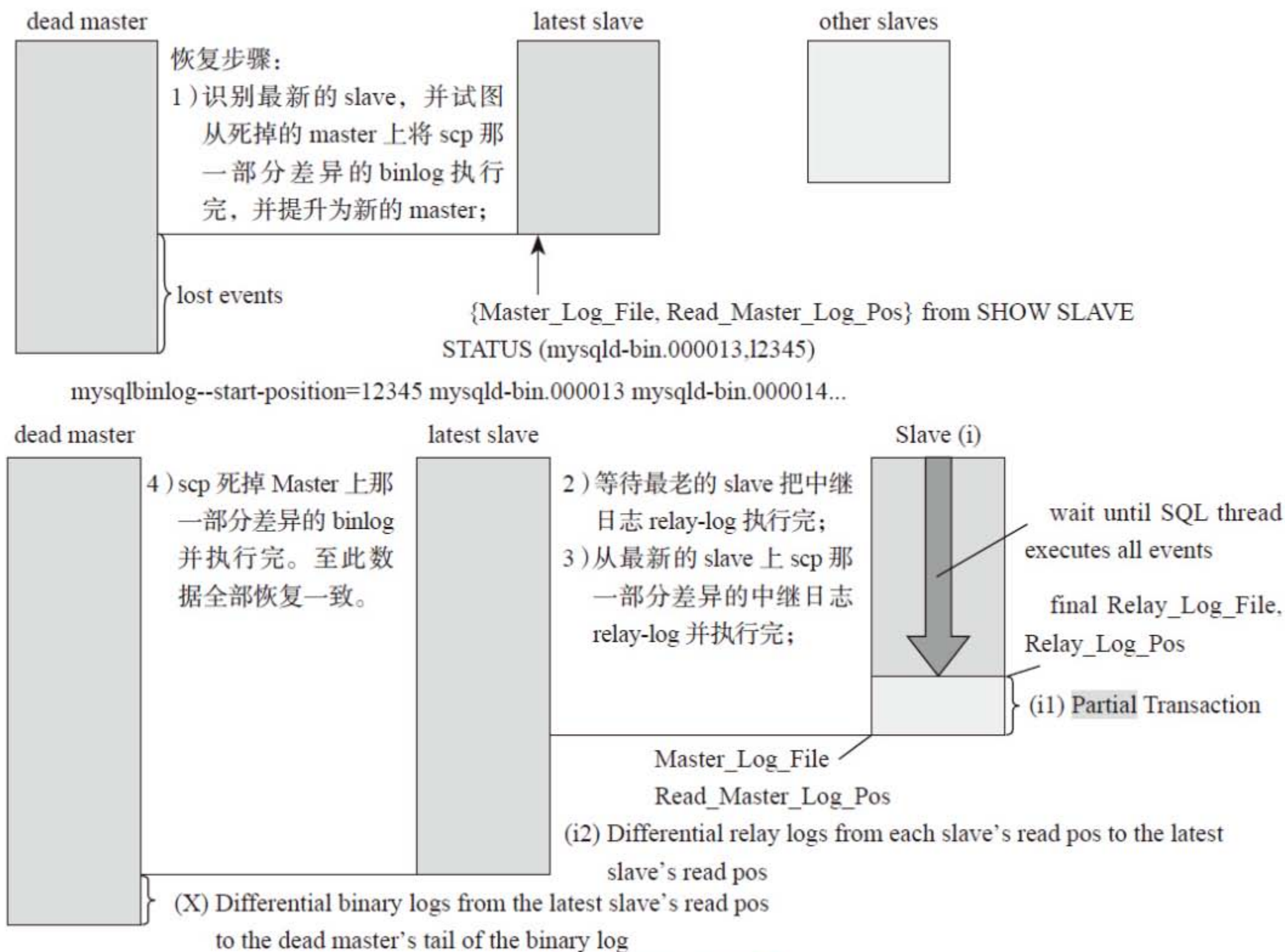


图7-2 缺失binlog补齐并恢复

### 7.1.1 master自动监控和故障转移

在当前已存在的主从复制环境中，MHA可以监控master主机故障，并且故障会自动转移。即使有些slave没有接收新的relay log events，MHA也会从最新的slave自动识别差异的relay log events，并应用差异的event事件到其他slaves。因此所有的slave都是一致的。

如果采用自动切换模式，则需要开启半同步复制（semi replication），以确保slave已经接收到master的binlog，因为master宕机，MHA管理机有可能无法远程拷贝scp那一缺失的binlog，那么数据就会出现不一致。



### 7.1.2 手工处理master故障转移

MHA可以用来只做故障转移，而不监测master，MHA只作为故障转移的交互。

如果采用默认的异步复制模式，由于master的宕机无法得知binlog是否全部发送到slave上，则此时需要等待master的崩溃恢复完成，以便把未发送的binlog同步到slave上。如果这时采用自动切换模式，那么数据就会不一致。

只有在机器长时间启动不了（如硬件主板损坏）或者崩溃恢复时间很长已严重影响业务，并且不在意那一部分丢失的数据的情况下，才可以人工介入手工处理master故障转移。

### 7.1.3 在线平滑切换

如果需要机器的维护，则将master在线切换到其他主机上（例如更换原master坏掉的硬盘），这并不是master崩溃引起的故障转移。在线切换通常需要0.5~2秒，并且会阻塞写（会执行FLUSH TABLESWITHREADLOCK命令加全局读锁）操作，建议在凌晨业务低峰期执行在线切换。



## 7.2 MHA配置安装

一个MHA管理节点可以管理多组集群，如图7-3所示。

MHA的主要功能由下面几个脚本组成。

- masterha\_manager：复制启动MHA进程。
- masterha\_master\_monitor：监控守护进程，判断master是否崩溃。
- masterha\_master\_switch：监控守护进程，将故障的master进行转移。
- filter\_mysqlbinlog：该脚本在新版本里废弃掉了。
- purge\_relay\_logs：删除sql\_thread线程执行完的relay log。
- save\_binary\_logs：保存和拷贝死掉的master上的binlog。
- apply\_diff\_relay\_logs：识别差异binlog和relay log并应用。

MHA配置环境如下。



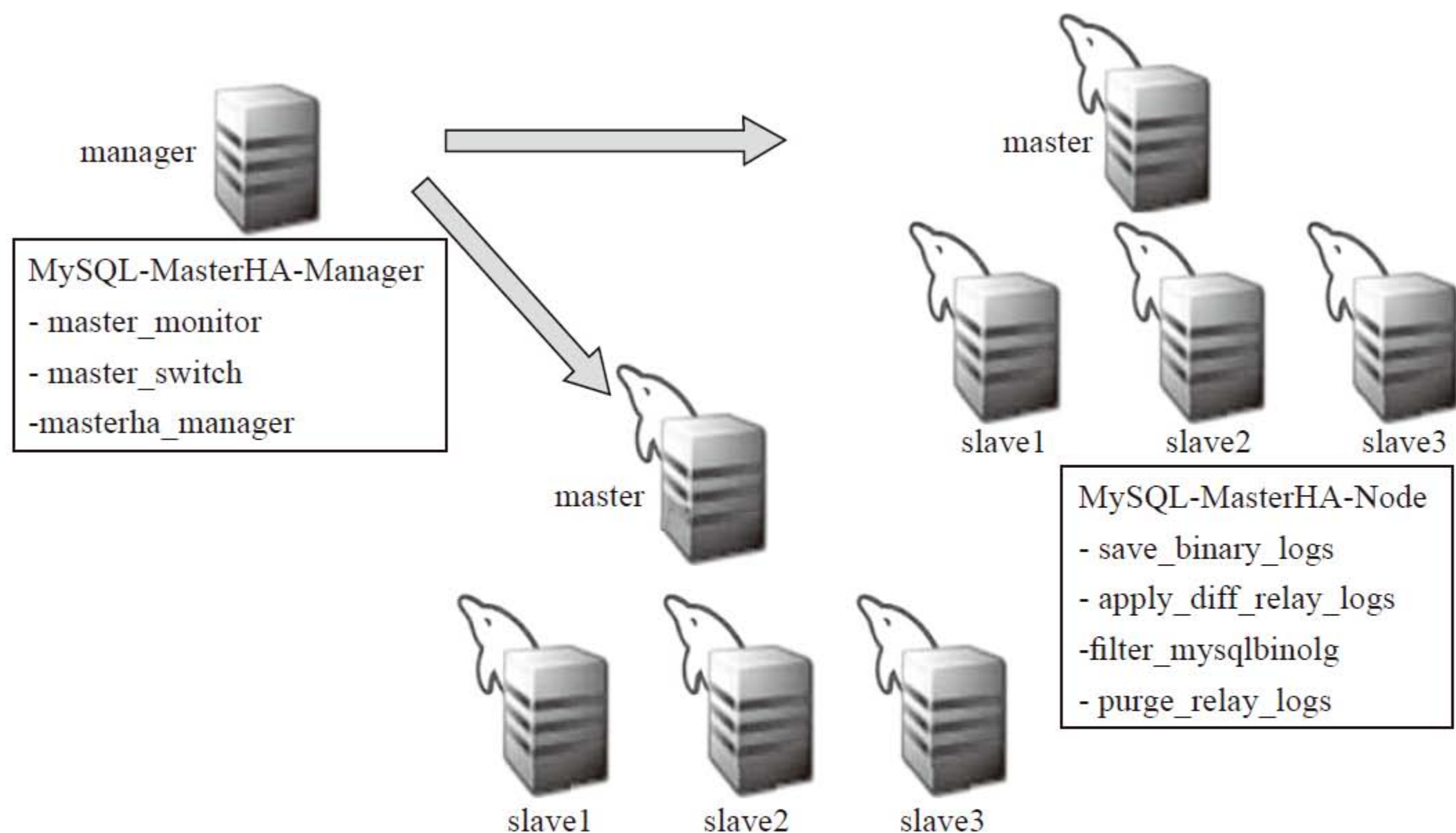


图7-3 管理多套MHA集群

MHA管理节点：192.168.17.131。

master：192.168.17.128。

slave1：192.168.17.129。

slave2：192.168.17.130。

VIP：192.168.17.100。

## 1.MHA架构的安装

下面介绍MHA架构的安装过程。

1) 设置host解析。4台服务器的配置如下：

```
# cat /etc/hosts
192.168.17.128 master
192.168.17.129 slave1
192.168.17.130 slave2
192.168.17.131 MHA
```

2) 建立MHA管理账号。创建MHA监控账号的命令如下：

```
GRANT ALL PRIVILEGES ON *.* TO 'admin'@'%' IDENTIFIED BY '123456';
```

创建主从复制账号的命令如下：

```
GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'repl'@'%' IDENTIFIED BY 'repl';
```

3) 安装MHA-mha4mysql-manager和mha4mysql-node。

在RHEL/CentOS 6上的安装步骤如下。

首先在MHA管理节点、master、slave上均安装perl依赖包，安装命令为：

```
# yum install perl-DBD-MySQL perl-Config-Tiny perl-Log-Dispatch perl-Parallel-ForkManager
```

如果你没有找到rpm包，则要安装epel源，命令如下：

```
# rpm -ivh epel-release-6-8.noarch.rpm
```

然后在管理节点安装MHA管理和客户端安装包，命令如下：

```
# tar zxvf mha4mysql-node-0.56.tar.gz
# cd mha4mysql-node
# perl Makefile.PL
# make
# make install
# tar zxvf mha4mysql-manager-0.56.tar.gz
# cd mha4mysql-manager
# perl Makefile.PL
# make
```



*# make install*

最后，在master和slave节点中安装MHA客户端安装包，命令如下：

```
# tar zxvf mha4mysql-node-0.56.tar.gz
```

```
# cd mha4mysql-node
```

```
# perl Makefile.PL
```

```
# make
```

```
# make install
```

4) 公钥认证，打通SSH无密码远程连接，在管理节点、master和slave上均执行如下命令。

```
# ssh-keygen
```

```
# ssh-copy-id '-p 22 root@192.168.17.128'
```

```
# ssh-copy-id '-p 22 root@192.168.17.129'
```

```
# ssh-copy-id '-p 22 root@192.168.17.130'
```

```
# ssh-copy-id '-p 22 root@192.168.17.131'
```



注意 如果你的ssh端口不是22，修改-p 22为你自己定义的端口。

5) 在slave机器上设置只读权限，并且关闭自动清除执行完毕后的中继日志（relay log）。

将以下参数加入my.cnf里：

```
[mysqld]
```

```
read_only = 1
```

```
relay_log_purge = 0
```



注意 设置read\_only（只读）模式的目的是防止在slave上被人工误写入数据，保证主从数据的一致性。

关闭自动清除执行完毕后的中继日志的目的是假如一台从库没有接受完主库的binlog，那么可以通过MHA把最新的slave上的中继日志发送到最老的slave上，识别差异中继日志并补齐数据。

可以通过定时任务来实现上述功能，该任务写入crontab里，例如，想要每天凌晨5点清除中继日志，那么可以这样：

```
# crontab -l
```





0 5 \* \* \* /usr/local/mysql/bin/mysql -S /tmp/mysql.sock -uroot -p123456 -e "set global relay\_log\_purge = 1;flush logs;set global relay\_log\_purge = 0;flush logs;"

6) 配置MHA监控、管理服务。

在MHA管理机中创建app1.cnf配置文件，内容如下：

```
# cat /etc/app1.cnf
```

```
[server default]
```

```
manager_workdir=/var/log/masterha/
```

```
remote_workdir=/var/log/masterha/
```

以上参数用来定义MHA的管理目录，以便存放日志。

设置ssh账号和端口，命令如下：

```
ssh_user=root
```

```
ssh_port=22
```

设置MHA的管理账号和密码，命令如下：

```
user=admin
```

```
password=123456
```

设置主从复制的账号和密码，命令如下：

```
repl_user=repl
```

```
repl_password=repl
```

如果你的环境是链式复制架构，比如A-B-C，那么设置multi\_tier\_slave=1，如下：

```
multi_tier_slave=1
```

然后，每隔1秒监测一次，命令如下：

```
ping_interval=1
```

以下命令可定义故障切换和在线切换。

```
ping_type=CONNECT
```

```
master_ip_failover_script=/usr/local/bin/master_ip_failover
```

```
master_ip_online_change_script=/usr/local/bin/master_ip_online_change
```

为了防止网络抖动误切换，造成数据不一致，可采用如下命令：

```
secondary_check_script=/usr/local/bin/masterha_secondary_check -s
```

```
192.168.17.131 -s 192.168.17.130 --user=root --master_host=master
--master_ip=192.168.17.128 --master_port=3306
```

其实现采用的是投票机制，当监控管理机无法ping通或无法连接MySQL主库时，会试图从slave2机上去ping通和连接MySQL主库，只有双方都连接失败，才认定MySQL主库宕机。假如有一方可以连接MySQL主库，则不会切换。

由于masterha\_secondary\_check脚本写死了端口，如果你的SSH端口不是22，则要手工修改ssh端口，如下：

```
$ssh_user = "root" unless ($ssh_user);
$ssh_port = 62222 unless ($ssh_port);
$master_port = 3306 unless ($master_port);
```

默认不会在slave存在故障的情况下（SQL Thread已经停止出错）进行master的故障切换。当设置ignore\_fail=1时，MHA会在所有的机器有问题的时候进行故障切换，如下：

```
[server1]
ignore_fail=0
```

默认情况下，当check\_repl\_delay=1时，若一个slave同步延迟超过100M Brelay log，则MHA在进行故障切换时不会选择这个slave作为新的master，因为恢复需要经过很长时间。

```
check_repl_delay = 1
```

设置了check\_repl\_delay=0，MHA将忽略被选择slave上的同步延迟。

以下参数很简单，这里不多加以介绍。

```
hostname=master
ip=192.168.17.128
port=3306
ssh_port=22
master_binlog_dir=/data/logs
```

```
[server2]
ignore_fail=0
check_repl_delay = 1
hostname=slave1
ip=192.168.17.129
port=3306
```



ssh\_port=22  
master\_binlog\_dir=/data/logs  
[server3]  
ignore\_fail=0  
check\_repl\_delay = 1  
hostname=slave2  
ip=192.168.17.130  
port=3306  
ssh\_port=22  
master\_binlog\_dir=/data/logs

7) 在master机器 ( 192.168.17.128 ) 上手工加载虚拟vip, 命令如下:

```
# cat vip_ipaddr.sh  
#!/bin/bash  
ip addr add 192.168.17.100/32 dev eth0
```

8) 启动MHA监控进程, 命令如下:

```
# nohup /usr/local/bin/masterha_manager --conf=/etc/app1.cnf  
--ignore_last_failover </dev/null >/var/log/masterha/manager.log  
2>&1 &
```

可以通过以下命令来查看是否成功启动:

```
# masterha_check_status --conf=/etc/app1.cnf  
app1 (pid:24721) is running(0:PING_OK), master:master
```

上述命令代表监控进程启动成功, 主库目前是master这台机器。

也可以通过如下命令查看状态信息。

```
# masterha_master_monitor --conf=/etc/app1.cnf .....  
Mon Nov 2 07:37:08 2015 - [info]  
master(192.168.17.128:3306) (current master)  
+--slave1(192.168.17.129:3306)  
+--slave2(192.168.17.130:3306)  
Mon Nov 2 07:37:08 2015 - [info] Checking master_ip_failover_script status:
```



```
Mon Nov 2 07:37:08 2015 - [info] /usr/local/bin/master_ip_failover
--command=status --ssh_user=root --orig_master_host=master
--orig_master_ip=192.168.17.128 --orig_master_port=3306
inet 192.168.17.100/32 scope global eth0
INFO: VIP 192.168.17.100 found on Master
Mon Nov 2 07:37:08 2015 - [info] OK.
```

9) 关闭MHA监控进程，命令如下：

```
# masterha_stop --conf=/etc/app1.cnf
Stopped app1 successfully.
```

## 2.MHA架构的注意事项

使用MHA架构时的注意事项包含以下几方面。

1) 防止网络抖动误切换（脑裂）造成数据不一致，其实现原理在上文中已介绍，下面是生产环境的一个例子，如图7-4所示。

2) VIP没有采用Keepalived，就是怕存在网络抖动问题，如果出现脑裂，那么从库也会抢夺VIP，由于主库和从库都持有VIP，因此会造成IP冲突，影响业务。Keepalived只能实现一个节点监控master，而通过自带的脚本可以实现两个节点监控master。

```
Fri Sep 18 17:08:35 2015 - [warning] Connection failed 3 time(s)..
Fri Sep 18 17:08:36 2015 - [warning] Got error on MySQL connect: 2003 (Can't connect to MySQL server on '192.168.111.77' (4))
Fri Sep 18 17:08:36 2015 - [warning] Connection failed 4 time(s)..
Fri Sep 18 17:08:38 2015 - [warning] HealthCheck: Got timeout on checking SSH connection to QCZJ-dbm! at /usr/local/share/perl5/MHA/HealthCheck.pm line 342.
Monitoring server 192.168.111.76 is reachable, Master is not reachable from 192.168.111.76. OK.
ssh: connect to host 192.168.111.79 port 60000: Connection timed out
Monitoring server 192.168.111.79 is NOT reachable!
Fri Sep 18 17:08:43 2015 - [warning] At least one of monitoring servers is not reachable from this script. This is likely a network problem. Failover should not happen.
```

图7-4 网络抖动未切换

修改的以下两个脚本（该脚本在华章网站下载），自带VIP：

```
master_ip_failover_script=/usr/local/bin/master_ip_failover
master_ip_online_change_script=/usr/local/bin/master_ip_online_change
```



```
-----  
# Hardcode stuff now until the next MHA release passes SSH info in here  
MHA::ManagerUtil::exec_ssh_cmd( $new_master_ip , '22' , "ip addr  
add 192.168.17.100/32 dev eth0;arping -q -c 2 -U -I eth0  
192.168.17.100" , undef );  
-----
```



注意 只需把脚本里的VIP地址换成你自己的，ssh端口默认是22换成你自己的，eth0网卡换成你自己的名字即可。

3) mha依赖ssh，ssh有时会出现连接慢或者不通的情况，如何解决？设置ssh连接超时时间(ssh timeout)，分别如图7-5和图7-6所示。

```
[root@QCZJ-dbs1 ~]# grep 'ConnectTimeout' /etc/ssh/ssh_config  
ConnectTimeout 3  
[root@QCZJ-dbs1 ~]#
```

图7-5 设置ssh连接超时时间

从图7-5和图7-6中可以看出，缩短了连接超时的时间，加快了故障切换的速度。

4) 死掉的原master如何与提升为新的master建立同步复制关系？

把下面这段语句复制下来，待死掉的原master恢复好后，执行该语句，即可与新提升的master建立同步复制关系。

```
# grep -i 'change' /var/log/masterha/manager.log  
Sat Sep 5 12:50:34 2015 - [info] All other slaves should start  
replication from here. Statement should be: CHANGE MASTER TO  
MASTER_HOST='QCZJ-dbm or 192.168.111.77',  
MASTER_PORT=3308 , MASTER_LOG_FILE='mysql-bin.000006',  
MASTER_LOG_POS=326 , MASTER_USER='repl',  
MASTER_PASSWORD='xxx';  
Sat Sep 5 12:50:35 2015 - [info] Executed CHANGE MASTER.
```



```
[root@slave2 ~]# time ssh -o ConnectTimeout=3 111.111.111.111
ssh: connect to host 111.111.111.111 port 22: Connection timed out

real    0m3.154s
user    0m0.039s
sys     0m0.010s
[root@slave2 ~]#
[root@slave2 ~]# time ssh 111.111.111.111
ssh: connect to host 111.111.111.111 port 22: Connection refused

real    0m21.043s
user    0m0.029s
sys     0m0.010s
[root@slave2 ~]#
```

图7-6 ssh连接耗时时间

#### 5) MHA结合半同步复制 (semi replication) 时应注意些什么？

互联网金融公司，数据不丢失是第一位的。开启半同步复制势必会影响性能，所以针对这种情况，将DB服务器通过VLAN划分为一个独立网段，与应用相隔离，保证有一个良好的网络环境。

此外，由于MHA是基于ssh公私钥认证的，有些公司禁止ssh互通，以避免黑客入侵后，又跳到其他DB服务器里，所以将DB与应用隔离是必要的。

#### 6) MHA 0.56最新版本不支持MariaDB 10的GTID复制，仅支持甲骨文MySQL的GTID复制。



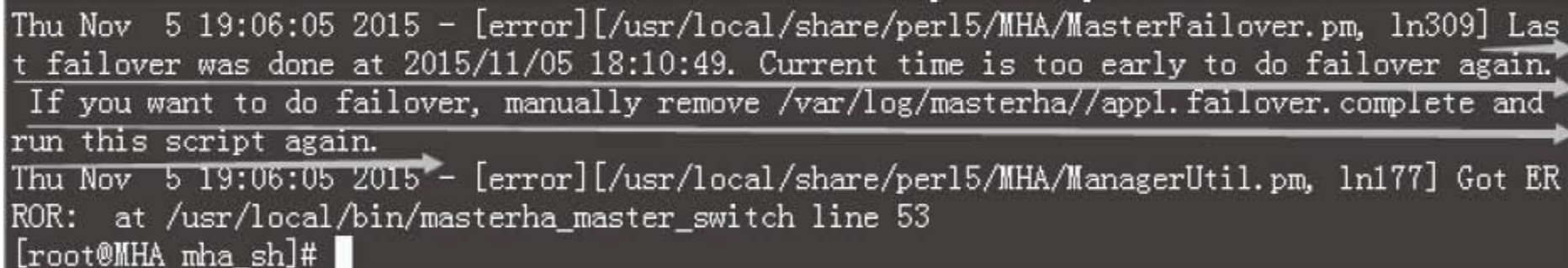
## 7.3 MHA故障切换演示

### 7.3.1 场景一：master自动监控和故障转移

首先启动MHA监控进程，命令如下：

```
# nohup /usr/local/bin/masterha_manager --conf=/etc/app1.cnf
--ignore_last_failover </dev/null >/var/log/masterha/manager.log
2>&1 &
```

这里要注意，如果在默认8小时内连续出现故障，则不会切换，可以通过设置--last\_failover\_minute=(minutes)来缩短时间。但如果设置了--ignore\_last\_failover参数，那么该步骤是被忽略的，如图7-7所示。



```
Thu Nov 5 19:06:05 2015 - [error][/usr/local/share/perl5/MHA/MasterFailover.pm, ln309] Last failover was done at 2015/11/05 18:10:49. Current time is too early to do failover again.
If you want to do failover, manually remove /var/log/masterha/appl.failover.complete and run this script again.
Thu Nov 5 19:06:05 2015 - [error][/usr/local/share/perl5/MHA/ManagerUtil.pm, ln177] Got ER_ROR: at /usr/local/bin/masterha_master_switch line 53
[root@MHA mha_sh]#
```

图7-7 8小时内连续故障不会切换

然后通过mysqladmin shutdown命令关闭主库的mysqld进程，到MHA管理机的/var/log/masterha目录下，使用tail-f manager.log命令查看切换日志。由于日志内容较多，下面逐层分解：

```
Thu Nov 5 15:16:36 2015 - [warning] Got error on MySQL connect ping: DBI
connect(';host=192.168.17.128;port=3306;mysql_connect_timeout=1', 'admin', ...) failed: Lost connection to MySQL server at
'reading initial communication packet', system error: 111 at /usr/local/share/perl5/MHA/HealthCheck.pm line 97
2013 (Lost connection to MySQL server at 'reading initial communication packet', system error: 111)
Thu Nov 5 15:16:36 2015 -[info] Executing SSH check script: save_binary_logs
--command=test --start_pos=4 --binlog_dir=/data/logs
```



```
--output_file=/var/log/masterha//save_binary_logs_test --manager_version=0.56
--binlog_prefix=mysql-bin
Thu Nov 5 15:16:36 2015 - [info] Executing secondary network check script:
/usr/local/bin/masterha_secondary_check -s 192.168.17.131 -s 192.168.17.130 --user=root
--master_host=master --master_ip=192.168.17.128 --master_port=3306 --user=root
--master_host=master --master_ip=192.168.17.128 --master_port=3306
--master_user=admin
--master_password=123456 --ping_type=CONNECT
    Creating /var/log/masterha if not exists.. ok.
    Checking output directory is accessible or not..
        ok.
    Binlog found at /data/logs , up to mysql-bin.000004
Thu Nov 5 15:16:36 2015 - [info] HealthCheck: SSH to master is reachable.
    Monitoring server 192.168.17.131 is reachable , Master is not reachable from 192.168.17.131. OK.
    Monitoring server 192.168.17.130 is reachable , Master is not reachable from 192.168.17.130. OK.
Thu Nov 5 15:16:36 2015 - [info] Master is not reachable from all other monitoring servers. Failover should start.
    Thu Nov 5 15:16:37 2015 - [warning] Got error on MySQL connect: 2013 (Lost connection to MySQL server at 'reading initial
communication packet' , system error: 111)
    Thu Nov 5 15:16:37 2015 - [warning] Connection failed 2 time(s)..
    Thu Nov 5 15:16:38 2015 - [warning] Got error on MySQL connect: 2013 (Lost connection to MySQL server at 'reading initial
communication packet' , system error: 111)
    Thu Nov 5 15:16:38 2015 - [warning] Connection failed 3 time(s)..
    Thu Nov 5 15:16:39 2015 - [warning] Got error on MySQL connect: 2013 (Lost connection to MySQL server at 'reading initial
communication packet' , system error: 111)
    Thu Nov 5 15:16:39 2015 - [warning] Connection failed 4 time(s)..
    Thu Nov 5 15:16:39 2015 - [warning] Master is not reachable from health checker!Thu Nov 5 15:16:39 2015 - [warning] Master
master(192.168.17.128:3306) is not reachable!
    Thu Nov 5 15:16:39 2015 - [warning] SSH is reachable.
    Thu Nov 5 15:16:39 2015 - [info] Connecting to a master server failed. Reading configuration file /etc/masterha_default.cnf
and /etc/app1.cnf again , and trying to connect to all servers to check server status..
```



```

Thu Nov 5 15:16:39 2015 - [warning] Global configuration file /etc/masterha_default.cnf not found. Skipping.
Thu Nov 5 15:16:39 2015 - [info] Reading application default configuration from /etc/app1.cnf.
Thu Nov 5 15:16:39 2015 - [info] Reading server configuration from /etc/app1.cnf.
Thu Nov 5 15:16:39 2015 - [info] GTID failover mode = 0Thu Nov 5 15:16:39 2015 - [info] Dead Servers:
Thu Nov 5 15:16:39 2015 - [info] master(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] Alive Servers:
Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306)
Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306)
Thu Nov 5 15:16:39 2015 - [info] Alive Slaves:
Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306)
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306)
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] Checking slave configurations..
Thu Nov 5 15:16:39 2015 - [info] Checking replication filtering settings..
Thu Nov 5 15:16:39 2015 - [info] Replication filtering check ok.Thu Nov 5 15:16:39 2015 - [info] Master is down!
Thu Nov 5 15:16:39 2015 - [info] Terminating monitoring script.
Thu Nov 5 15:16:39 2015 - [info] Got exit code 20 (Master dead).
Thu Nov 5 15:16:39 2015 - [warning] Global configuration file /etc/masterha_default.cnf not found. Skipping.
Thu Nov 5 15:16:39 2015 - [info] Reading application default configuration from /etc/app1.cnf.
Thu Nov 5 15:16:39 2015 - [info] Reading server configuration from /etc/app1.cnf.Thu Nov 5 15:16:39 2015 - [info]
MHA::MasterFailover version 0.56.
Thu Nov 5 15:16:39 2015 - [info] Starting master failover.

```

若MHA管理端无法连接主库192.168.17.128，则调用如下命令检查128主库的binlog文件是否存在。

```

save_binary_logs --command=test --start_pos=4 --binlog_dir=/data/logs --output_file=/var/log/masterha//save_binary_logs_test
--manager_version=0.56 --binlog_prefix=mysql-bin
Creating /var/log/masterha if not exists.. ok.
Checking output directory is accessible or not..

```



ok.

Binlog found at /data/logs , up to mysql-bin.000004

然后调用如下命令，通过事先定义好的IP地址192.168.17.131和192.168.17.130同时访问主库128机器。如果都无法连接，就认定128主库宕机。

```
masterha_secondary_check -s 192.168.17.131 -s 192.168.17.130
--user=root --master_host=master --master_ip=192.168.17.128
--master_port=3306 --user=root --master_host=master
--master_ip=192.168.17.128 --master_port=3306 --master_user=admin
--master_password=123456 --ping_type=CONNECT
```

继续看下面的日志：

```
Thu Nov 5 15:16:39 2015 - [info] * Phase 1: Configuration Check Phase..
Thu Nov 5 15:16:39 2015 - [info]
Thu Nov 5 15:16:39 2015 - [info] GTID failover mode = 0Thu Nov 5 15:16:39 2015 - [info] Dead Servers:
Thu Nov 5 15:16:39 2015 - [info] master(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] Checking master reachability via MySQL(double check)...
Thu Nov 5 15:16:39 2015 - [info] ok.
Thu Nov 5 15:16:39 2015 - [info] Alive Servers:
Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306)
Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306)Thu Nov 5 15:16:39 2015 - [info] Alive Slaves:
Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306) Version=10.1.8-MariaDB-log (oldest major version between slaves)
log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306) Version=10.1.8-MariaDB-log (oldest major version between slaves)
log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] Starting Non-GTID based failover.
Thu Nov 5 15:16:39 2015 - [info]
Thu Nov 5 15:16:39 2015 - info] ** Phase 1: Configuration Check Phase completed.
```

这里是第一阶段：配置检查。从上面的日志信息中可以得知死掉的主库是192.168.17.128，存活的从库是192.168.17.129/130，版本为



10.1.8-MariaDB-log , 且没有开启GTID同步复制。

*Thu Nov 5 15:16:39 2015 - [info] \* Phase 2: Dead Master Shutdown Phase..*

*Thu Nov 5 15:16:39 2015 - [info]*

*Thu Nov 5 15:16:39 2015 - [info] Forcing shutdown so that applications never connect to the current master..*

*Thu Nov 5 15:16:39 2015 - [info] Executing master IP deactivation script:Thu Nov 5 15:16:39 2015 - [info] /usr/local/bin/*

*master\_ip\_failover*

*--orig\_master\_host=master --orig\_master\_ip=192.168.17.128 --orig\_master\_port=3306*

*--command=stopssh --ssh\_user=root*

*Thu Nov 5 15:16:39 2015 - [info] done.*

*Thu Nov 5 15:16:39 2015 - [warning] shutdown\_script is not set. Skipping explicit shutting down of the dead master.*

*Thu Nov 5 15:16:39 2015 - [info] \* Phase 2: Dead Master Shutdown Phase completed.*

第二阶段是把死掉的master从虚拟的VIP ( 192.168.17.100 ) 中移除。

继续看下面的日志：

*Thu Nov 5 15:16:39 2015 - [info] \* Phase 3: Master Recovery Phase..*

*Thu Nov 5 15:16:39 2015 - [info]*

*Thu Nov 5 15:16:39 2015 - [info] \* Phase 3.1: Getting Latest Slaves Phase..*

*Thu Nov 5 15:16:39 2015 - [info]*

*Thu Nov 5 15:16:39 2015 - [info] The latest binary log file/position on all slaves is  
mysql-bin.000004:327*

*Thu Nov 5 15:16:39 2015 - [info] Latest slaves (Slaves that received relay log files to the latest):*

*Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306)*

*Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled*

*Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)*

*Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306)*

*Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled*

*Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)Thu Nov 5 15:16:39 2015 - [info] The oldest  
binary log file/position on all slaves is  
mysql-bin.000004:327*



```
Thu Nov 5 15:16:39 2015 - [info] Oldest slaves:
Thu Nov 5 15:16:39 2015 - [info] slave1(192.168.17.129:3306)
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
Thu Nov 5 15:16:39 2015 - [info] slave2(192.168.17.130:3306)
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled
Thu Nov 5 15:16:39 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)
```

第三阶段是进行最新的slave数据恢复。在此阶段中，首先要检查最新的slave和最老的slave所接收的master上的binlog是否都为mysql-bin.000004，position位置是否为327。

继续看下面的日志：

```
Thu Nov 5 15:16:39 2015 - [info] * Phase 3.2: Saving Dead Master's Binlog Phase..
Thu Nov 5 15:16:39 2015 - [info]
Thu Nov 5 15:16:39 2015 - [info] Fetching dead master's binary logs..
Thu Nov 5 15:16:39 2015 - [info] Executing command on the dead mastermaster(192.168.17.128:3306): save_binary_logs
--command=save
--start_file=mysql-bin.000004 --start_pos=327 --binlog_dir=/data/logs
--output_file=/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog --handle_raw_binlog=1
--disable_log_bin=0 --manager_version=0.56
Creating /var/log/masterha if not exists.. ok.
Concat binary/relay logs from mysql-bin.000004 pos 327 to mysql-bin.000004 EOF into /var/log/masterha//
saved_master_binlog_from_master_3306_20151105151639.binlog ..
Dumping binlog format description event , from position 0 to 249.. ok.
Dumping effective binlog data from /data/logs/mysql-bin.000004 position 327 to tail(346).. ok.
Concat succeeded.
Thu Nov 5 15:16:40 2015 - [info] scp from
root@192.168.17.128:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog to
local:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog succeeded.
Thu Nov 5 15:16:40 2015 - [info] HealthCheck: SSH to slave1 is reachable.
Thu Nov 5 15:16:40 2015 - [info] HealthCheck: SSH to slave2 is reachable.
```



这里抓取了主库上未发送的binlog，通过如下命令在主库192.168.17.128上找到名为mysql-bin.000004、position位置为327的binlog日志，把该日志之后的binlog保存到本地/var/log/masterha/目录下的saved\_master\_binlog\_from\_master\_3306\_20151105151639.binlog文件中。

```
save_binary_logs --command=save
--start_file=mysql-bin.000004 --start_pos=327
--binlog_dir=/data/logs
--output_file=/var/log/masterha/saved_master_binlog_from_master_3306_20151105151639.binlog --handle_raw_binlog=1
--disable_log_bin=0 --manager_version=0.56
```

接着，将saved\_master\_binlog\_from\_master\_3306\_20151105151639.binlog文件拷贝到MHA管理机本地/var/log/masterha/目录下。

继续看下面的日志：

```
Thu Nov 5 15:16:41 2015 - [info] * Phase 3.3: Determining New Master Phase..
Thu Nov 5 15:16:41 2015 - [info]
Thu Nov 5 15:16:41 2015 - [info] Finding the latest slave that has all relay logs for recovering other slaves..
Thu Nov 5 15:16:41 2015 - [info] All slaves received relay logs to the same position. No need to resync each other.
Thu Nov 5 15:16:41 2015 - [info] Searching new master from slaves..
Thu Nov 5 15:16:41 2015 - [info] Candidate masters from the configuration file:
Thu Nov 5 15:16:41 2015 - [info] Non-candidate masters:
Thu Nov 5 15:16:41 2015 - [info] New master is slave1(192.168.17.129:3306)
Thu Nov 5 15:16:41 2015 - [info] Starting master failover..
Thu Nov 5 15:16:41 2015 - [info] From:
master(192.168.17.128:3306) (current master)
+--slave1(192.168.17.129:3306)
+--slave2(192.168.17.130:3306)To:
slave1(192.168.17.129:3306) (new master)
+--slave2(192.168.17.130:3306)
Thu Nov 5 15:16:41 2015 - [info]
Thu Nov 5 15:16:41 2015 - [info] * Phase 3.3: New Master Diff Log Generation Phase..
Thu Nov 5 15:16:41 2015 - [info]
```



```
Thu Nov 5 15:16:41 2015 - [info] This server has all relay logs. No need to generate diff files from the latest slave.
Thu Nov 5 15:16:41 2015 - [info] Sending binlog..
Thu Nov 5 15:16:41 2015 - [info] scp from
local:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog to
root@slave1:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639. binlog succeeded.
```

这里提升一个新的master。由于slave1和slave2接收的binlog是一致的，数据都是最新的，因此不需要做数据补齐操作。检查MHA的app1.cnf配置文件是否设置了candidate\_master=1参数，如果设置了，则将该机器提升为新的master；如果未设置，则按照IP的顺序，将192.168.17.129提升为新的master。

继续看下面的日志：

```
Thu Nov 5 15:16:41 2015 - [info] * Phase 3.4: Master Log Apply Phase..
Thu Nov 5 15:16:41 2015 - [info]
Thu Nov 5 15:16:41 2015 - [info] *NOTICE: If any error happens from this phase , manual recovery is needed.
Thu Nov 5 15:16:41 2015 - [info] Starting recovery on slave1(192.168.17.129:3306)..
Thu Nov 5 15:16:41 2015 - [info] Generating diffs succeeded.
Thu Nov 5 15:16:41 2015 - [info] Waiting until all relay logs are applied.
Thu Nov 5 15:16:41 2015 - [info] done.
Thu Nov 5 15:16:41 2015 - [info] Getting slave status..
Thu Nov 5 15:16:41 2015 -[info] This slave(slave1)'s Exec_Master_Log_Pos equals to Read_Master_Log_Pos(mysql-bin.000004:327).
No need to recover from
Exec_Master_Log_Pos.
Thu Nov 5 15:16:41 2015 - [info] Connecting to the target slave host slave1 , running recover script..
Thu Nov 5 15:16:41 2015 -[info] Executing command: apply_diff_relay_logs
--command=apply --slave_user='admin' --slave_host=slave1 --slave_ip=192.168.17.129
--slave_port=3306
--apply_files=/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog --workdir=/var/log/masterha/
--target_version=10.1.8-MariaDB-log
--timestamp=20151105151639 --handle_raw_binlog=1 --disable_log_bin=0
--manager_version=0.56 --slave_pass=xxx
Thu Nov 5 15:16:41 2015 - [info]
```



MySQL client version is 10.1.8. Using --binary-mode. Applying differential binary/relay log files /var/log/masterha/saved\_master\_binlog\_from\_master\_3306\_20151105151639.binlog on slave1:3306. This may take long time...

Applying log files succeeded.

Thu Nov 5 15:16:41 2015 - [info] All relay logs were successfully applied.

Thu Nov 5 15:16:41 2015 - [info] Getting new master's binlog name and position..

Thu Nov 5 15:16:41 2015 - [info] mysql-bin.000018:327

Thu Nov 5 15:16:41 2015 - [info] All other slaves should start replication from here. Statement should be: CHANGE MASTER TO MASTER\_HOST='slave1 or 192.168.17.129',

MASTER\_PORT=3306, MASTER\_LOG\_FILE='mysql-bin.000018',

MASTER\_LOG\_POS=327, MASTER\_USER='repl', MASTER\_PASSWORD='xxx';

Thu Nov 5 15:16:41 2015 - [info] Executing master IP activate script: Thu Nov 5 15:16:41 2015 - [info] /usr/local/bin/master\_ip\_failover --command=start

--ssh\_user=root --orig\_master\_host=master --orig\_master\_ip=192.168.17.128

--orig\_master\_port=3306 --new\_master\_host=slave1 --new\_master\_ip=192.168.17.129

--new\_master\_port=3306 --new\_master\_user='admin' --new\_master\_password='123456'

Set read\_only=0 on the new master.

Thu Nov 5 15:16:42 2015 - [info] OK.

Thu Nov 5 15:16:42 2015 - [info] \*\* Finished master recovery successfully.

Thu Nov 5 15:16:42 2015 - [info] \* Phase 3: Master Recovery Phase completed.

这里识别差异binlog，并补齐数据。假如slave1的数据比主库上的老，那么把MHA本地的/var/log/masterha/saved\_master\_binlog\_from\_master\_3306\_20151105151639.binlog文件拷贝到slave1的/var/log/masterha/目录下，通过apply\_diff\_relay\_logs命令识别差异日志并应用之。

生成以下同步复制语句：

CHANGE MASTER TO

MASTER\_HOST='slave1 or 192.168.17.129',

MASTER\_PORT=3306, MASTER\_LOG\_FILE='mysql-bin.000018',

MASTER\_LOG\_POS=327, MASTER\_USER='repl',

MASTER\_PASSWORD='xxx';



通过master\_ip\_failover命令将故障转移，将虚拟VIP（192.168.17.100）漂移到slave1（192.168.17.129）机器上，从而使其提升为新的master，并在slave1（新提升的master）机器上，关闭只读参数set global read\_only=0;。

继续看下面的日志：

```
Thu Nov 5 15:16:42 2015 - [info] * Phase 4: Slaves Recovery Phase..
Thu Nov 5 15:16:42 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] * Phase 4.1: Starting Parallel Slave Diff Log Generation Phase..
Thu Nov 5 15:16:42 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] -- Slave diff file generation on host
slave2(192.168.17.130:3306) started , pid: 4423. Check tmp log
/var/log/masterha//slave2_3306_20151105151639.log if it takes time..
Thu Nov 5 15:16:42 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] Log messages from slave2 ...
Thu Nov 5 15:16:42 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] This server has all relay logs. No need to generate diff files from the latest slave.
Thu Nov 5 15:16:42 2015 - [info] End of log messages from slave2.
Thu Nov 5 15:16:42 2015 - [info] -- slave2(192.168.17.130:3306) has the latest relay log events.
Thu Nov 5 15:16:42 2015 - [info] Generating relay diff files from the latest slave succeeded.
```

第四阶段是恢复最老的slave数据。如果最老的slave的sql\_thread还没执行完，那么要先等它执行完毕。然后检查最老的slave执行完成的relay log和position，并与最新的slave进行对比。

继续看下面的日志：

```
Thu Nov 5 15:16:42 2015 - [info] * Phase 4.2: Starting Parallel Slave Log Apply Phase..
Thu Nov 5 15:16:42 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] -- Slave recovery on host slave2(192.168.17.130:3306) started , pid: 4425. Check tmp log /var/
log/masterha//slave2_3306_20151105151639.log if it takes time..
Thu Nov 5 15:16:43 2015 - [info]
Thu Nov 5 15:16:43 2015 - [info] Log messages from slave2 ...
Thu Nov 5 15:16:43 2015 - [info]
Thu Nov 5 15:16:42 2015 - [info] Sending binlog..
Thu Nov 5 15:16:43 2015 - [info] scp from
```



```

local:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog to
root@slave2:/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog succeeded.
Thu Nov 5 15:16:43 2015 - [info] Starting recovery on slave2(192.168.17.130:3306)..
Thu Nov 5 15:16:43 2015 - [info] Generating diffs succeeded.
Thu Nov 5 15:16:43 2015 - [info] Waiting until all relay logs are applied.
Thu Nov 5 15:16:43 2015 - [info] done.
Thu Nov 5 15:16:43 2015 - [info] Getting slave status..
Thu Nov 5 15:16:43 2015 - [info] This slave(slave2)'s Exec_Master_Log_Pos equals to Read_Master_Log_Pos
(mysql-bin.000004:327). No need to recover from Exec_Master_Log_Pos.
Thu Nov 5 15:16:43 2015 - [info] Connecting to the target slave host slave2 , running recover script..
Thu Nov 5 15:16:43 2015 - [info] Executing command: apply_diff_relay_logs
--command=apply --slave_user='admin' --slave_host=slave2 --slave_ip=192.168.17.130
--slave_port=3306 --apply_files=/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog
--workdir=/var/log/masterha/ --target_version=10.1.8-MariaDB-log
--timestamp=20151105151639 --handle_raw_binlog=1 --disable_log_bin=0
--manager_version=0.56 --slave_pass=xxx
Thu Nov 5 15:16:43 2015 - [info]
MySQL client version is 10.1.8. Using --binary-mode.Applying differential binary/relay log files
/var/log/masterha//saved_master_binlog_from_master_3306_20151105151639.binlog on
slave2:3306. This may take long time...
Applying log files succeeded.
Thu Nov 5 15:16:43 2015 - [info] All relay logs were successfully applied.Thu Nov 5 15:16:43 2015 - [info] Resetting slave slave2
(192.168.17.130:3306) and starting replication from the new master slave1(192.168.17.129:3306)..
Thu Nov 5 15:16:43 2015 - [info] Executed CHANGE MASTER.
Thu Nov 5 15:16:43 2015 - [info] Slave started.
Thu Nov 5 15:16:43 2015 - [info] End of log messages from slave2.
Thu Nov 5 15:16:43 2015 - [info] -- Slave recovery on host slave2(192.168.17.130:3306) succeeded.
Thu Nov 5 15:16:43 2015 - [info] All new slave servers recovered successfully.

```

当最老的slave把最新的slave上的relay log补齐后，MHA将master缺失的那一部分binlog发送给最老的slave，然后通过apply\_diff\_relay\_logs命令将数据补齐，之后重置同步复制关系，重新提升master。



继续看下面的日志：

*Thu Nov 5 15:16:43 2015 - [info] \* Phase 5: New master cleanup phase..*

*Thu Nov 5 15:16:43 2015 - [info]*

*Thu Nov 5 15:16:43 2015 - [info] Resetting slave info on the new master..*

*Thu Nov 5 15:16:43 2015 - [info] slave1: Resetting slave info succeeded.*

*Thu Nov 5 15:16:43 2015 - [info] Master failover to slave1(192.168.17.129:3306) completed successfully.*

*Thu Nov 5 15:16:43 2015 - [info]*

*----- Failover Report -----app1: MySQL Master failover master(192.168.17.128:3306) to slave1(192.168.17.129:3306) succeeded*

*Master master(192.168.17.128:3306) is down!*

*Check MHA Manager logs at MHA for details.*

*Started automated(non-interactive) failover.*

*Invalidated master IP address on master(192.168.17.128:3306)*

*The latest slave slave1(192.168.17.129:3306) has all relay logs for recovery.*

*Selected slave1(192.168.17.129:3306) as a new master.*

*slave1(192.168.17.129:3306): OK: Applying all logs succeeded.*

*slave1(192.168.17.129:3306): OK: Activated master IP address.*

*slave2(192.168.17.130:3306): This host has the latest relay log events.*

*Generating relay diff files from the latest slave succeeded.*

*slave2(192.168.17.130:3306): OK: Applying all logs succeeded. Slave started , replicating from slave1(192.168.17.129:3306)slave1  
(192.168.17.129:3306): Resetting slave info succeeded.*

*Master failover to slave1(192.168.17.129:3306) completed successfully.*

最后就是打印故障切换报告了。



### 7.3.2 场景二：master手工故障转移

通过以下命令可以实现手工故障切换：

```
#masterha_master_switch --conf=/etc/app1.cnf --master_state=dead  
--ignore_last_failover --dead_master_host=master  
--dead_master_ip=192.168.17.128 --dead_master_port=3306
```

这里的切换日志跟自动故障转移是一样的，为了节省篇幅，就不再介绍。

### 7.3.3 场景三：在线平滑切换

通过以下命令可以实现在线平滑切换：

```
# masterha_master_switch --conf=/etc/app1.cnf --master_state=alive  
--orig_master_is_new_slave
```

因为日志内容较多，下面逐层分解：

```
Tue Nov 10 00:13:29 2015 - [info] MHA::MasterRotate version 0.56.  
Tue Nov 10 00:13:29 2015 - [info] Starting online master switch..  
Tue Nov 10 00:13:29 2015 - [info]  
Tue Nov 10 00:13:29 2015 - [info] * Phase 1: Configuration Check Phase..  
Tue Nov 10 00:13:29 2015 - [info]  
Tue Nov 10 00:13:29 2015 - [warning] Global configuration file /etc/masterha_default.cnf not  
found. Skipping.  
Tue Nov 10 00:13:29 2015 - [info] Reading application default configuration from /etc/app1.cnf..  
Tue Nov 10 00:13:29 2015 - [info] Reading server configuration from /etc/app1.cnf..  
Tue Nov 10 00:13:29 2015 - [info] GTID failover mode = 0  
Tue Nov 10 00:13:29 2015 - [info] Current Alive Master: master(192.168.17.128:3306)  
Tue Nov 10 00:13:29 2015 - [info] Alive Slaves:  
Tue Nov 10 00:13:29 2015 - [info] slave1(192.168.17.129:3306)  
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled  
Tue Nov 10 00:13:29 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)  
Tue Nov 10 00:13:29 2015 - [info] slave2(192.168.17.130:3306)  
Version=10.1.8-MariaDB-log (oldest major version between slaves) log-bin:enabled  
Tue Nov 10 00:13:29 2015 - [info] Replicating from 192.168.17.128(192.168.17.128:3306)It is better to execute FLUSH  
NO_WRITE_TO_BINLOG TABLES on the master before switching. Is it ok to execute on master(192.168.17.128:3306) (YES/no): YES  
Tue Nov 10 00:13:46 2015 - [info] Executing FLUSH NO_WRITE_TO_BINLOG TABLES. This may take long time..  
Tue Nov 10 00:13:47 2015 - [info] ok.  
Tue Nov 10 00:13:47 2015 - [info] Checking MHA is not monitoring or doing failover..
```







*Tue Nov 10 00:13:47 2015 - [info] Checking replication health on slave1..*

*Tue Nov 10 00:13:47 2015 - [info] ok.*

*Tue Nov 10 00:13:47 2015 - [info] Checking replication health on slave2..*

*Tue Nov 10 00:13:47 2015 - [info] ok.*

*Tue Nov 10 00:13:47 2015 - [info] Searching new master from slaves..*

*Tue Nov 10 00:13:47 2015 - [info] Candidate masters from the configuration file:*

*Tue Nov 10 00:13:47 2015 - [info] Non-candidate masters:*

*Tue Nov 10 00:13:47 2015 - [info] From:*

*master(192.168.17.128:3306) (current master)*

*+--slave1(192.168.17.129:3306)*

*+--slave2(192.168.17.130:3306)To:*

*slave1(192.168.17.129:3306) (new master)*

*+--slave2(192.168.17.130:3306)*

*+--master(192.168.17.128:3306)Starting master switch from master(192.168.17.128:3306) to slave1(192.168.17.129:3306) (yes/NO): yes*

*Tue Nov 10 00:13:57 2015 - [info] Checking whether slave1(192.168.17.129:3306) is ok for the new master..*

*Tue Nov 10 00:13:57 2015 - [info] ok.*

*Tue Nov 10 00:13:57 2015 - [info] master(192.168.17.128:3306): SHOW SLAVE STATUS returned empty result. To check replication filtering rules , temporarily executing CHANGE*

*MASTER to a dummy host.*

*Tue Nov 10 00:13:57 2015 - [info] master(192.168.17.128:3306): Resetting slave pointing to the dummy host.*

*Tue Nov 10 00:13:57 2015 - [info] \*\* Phase 1: Configuration Check Phase completed.*

这是第一阶段：配置检查。首先应检测到当前存活主机master(192.168.17.128:3306)、slave1(192.168.17.129:3306)和slave2(192.168.17.130:3306)。

在输入yes后，只要在原master上执行FLUSH NO\_WRITE\_TO\_BINLOG TABLES操作，就会强制把打开的表关闭，这一步会耗费很长时间，尤其是业务繁忙的时候，请务必在凌晨执行。

之后会询问是否要把master(192.168.17.128:3306)切换到slave1(192.168.17.129:3306)，输入yes。

继续看下面的日志：



Tue Nov 10 00:13:57 2015 - [info] \* Phase 2: Rejecting updates Phase..

Tue Nov 10 00:13:57 2015 - [info]

Tue Nov 10 00:13:57 2015 - [info] Executing master ip online change script to disable write on the current master:

Tue Nov 10 00:13:57 2015 - [info] /usr/local/bin/master\_ip\_online\_change

--command=stop--orig\_master\_host=master --orig\_master\_ip=192.168.17.128 --orig\_master\_port=3306

--orig\_master\_user='admin' --orig\_master\_password='123456' --new\_master\_host=slave1

--new\_master\_ip=192.168.17.129 --new\_master\_port=3306 --new\_master\_user='admin'

--new\_master\_password='123456' --orig\_master\_ssh\_user=root

--new\_master\_ssh\_user=root

--orig\_master\_is\_new\_slave

ARGS: \$VAR1 = [

'--command=stop',

'--orig\_master\_host=master',

'--orig\_master\_ip=192.168.17.128',

'--orig\_master\_port=3306',

'--orig\_master\_user=admin',

'--orig\_master\_password=123456',

'--new\_master\_host=slave1',

'--new\_master\_ip=192.168.17.129',

'--new\_master\_port=3306',

'--new\_master\_user=admin',

'--new\_master\_password=123456',

'--orig\_master\_ssh\_user=root',

'--new\_master\_ssh\_user=root',

'--orig\_master\_is\_new\_slave'

];

Unknown option: orig\_master\_ssh\_user

Unknown option: new\_master\_ssh\_user

Unknown option: orig\_master\_is\_new\_slave

Tue Nov 10 00:13:57 2015 595003 Set read\_only on the new master.. ok.





*bind: Cannot assign requested address*

*Tue Nov 10 00:13:57 2015 770128 Set read\_only=1 on the orig master.. ok.*

*Tue Nov 10 00:13:57 2015 775882 Killing all application threads..*

*Tue Nov 10 00:13:57 2015 775924 done.*

*STOP ARGS: \$VAR1 = [];*

*Tue Nov 10 00:13:57 2015 - [info] ok.*

*Tue Nov 10 00:13:57 2015 - [info] Locking all tables on the orig master to reject updates from everybody (including root):*

*Tue Nov 10 00:13:57 2015 - [info] Executing FLUSH TABLES WITH READ LOCK..*

*Tue Nov 10 00:13:57 2015 - [info] ok.*

*Tue Nov 10 00:13:57 2015 - [info] Orig master binlog:pos is mysql-bin.000003:307127.*

*Tue Nov 10 00:13:57 2015 - [info] Waiting to execute all relay logs on  
slave1(192.168.17.129:3306)..*

*Tue Nov 10 00:13:57 2015 - [info] master\_pos\_wait(mysql-bin.000003:307127)completed on slave1(192.168.17.129:3306). Executed  
0 events.*

*Tue Nov 10 00:13:57 2015 - [info] done.*

*Tue Nov 10 00:13:57 2015 - [info] Getting new master's binlog name and position..*

*Tue Nov 10 00:13:57 2015 - [info] mysql-bin.000022:327*

*Tue Nov 10 00:13:57 2015 - [info] All other slaves should start replication from here. Statement should be: CHANGE MASTER TO  
MASTER\_HOST='slave1 or 192.168.17.129',*

*MASTER\_PORT=3306, MASTER\_LOG\_FILE='mysql-bin.000022', MASTER\_LOG\_POS=327, MASTER\_USER='repl',  
MASTER\_PASSWORD='xxx';*

这是第二阶段：拒绝更新，防止原主库被写入数据。

切换过程包含以下几方面。

- 1) 将原master上的虚拟VIP清除。
- 2) 设置原master为只读模式set global read\_only=1。
- 3) kill掉所有应用连接的线程。
- 4) 执行FLUSH TABLES WITH READ LOCK全局读锁。
- 5) 在原master上执行select master\_pos\_wait(mysql-bin.000003:307127)，等待slave1和slave2把relay log执行完。

继续看下面的日志：

Tue Nov 10 00:13:57 2015 - [info] /usr/local/bin/master\_ip\_online\_change

--command=start

--orig\_master\_host=master --orig\_master\_ip=192.168.17.128 --orig\_master\_port=3306

--orig\_master\_user='admin' --orig\_master\_password='123456' --new\_master\_host=slave1

--new\_master\_ip=192.168.17.129 --new\_master\_port=3306 --new\_master\_user='admin'

--new\_master\_password='123456' --orig\_master\_ssh\_user=root

--new\_master\_ssh\_user=root

--orig\_master\_is\_new\_slave

ARGS: \$VAR1 = [

'--command=start',

'--orig\_master\_host=master',

'--orig\_master\_ip=192.168.17.128',

'--orig\_master\_port=3306',

'--orig\_master\_user=admin',

'--orig\_master\_password=123456',

'--new\_master\_host=slave1',

'--new\_master\_ip=192.168.17.129',

'--new\_master\_port=3306',

'--new\_master\_user=admin',

'--new\_master\_password=123456',

'--orig\_master\_ssh\_user=root',

'--new\_master\_ssh\_user=root',

'--orig\_master\_is\_new\_slave'

];

Unknown option: orig\_master\_ssh\_user

Unknown option: new\_master\_ssh\_user

Unknown option: orig\_master\_is\_new\_slave

Tue Nov 10 00:13:58 2015 022670 Set read\_only=0 on the new master.

Tue Nov 10 00:13:58 2015 - [info] ok.

Tue Nov 10 00:13:58 2015 - [info]



Tue Nov 10 00:13:58 2015 - [info] \* Switching slaves in parallel..

Tue Nov 10 00:13:58 2015 - [info]

Tue Nov 10 00:13:58 2015 - [info] -- Slave switch on host slave2(192.168.17.130: 3306) started , pid: 1362

Tue Nov 10 00:13:58 2015 - [info]

Tue Nov 10 00:13:58 2015 - [info] Log messages from slave2 ...

Tue Nov 10 00:13:58 2015 - [info]

Tue Nov 10 00:13:58 2015 - [info] Waiting to execute all relay logs on slave2(192.168.17.130:3306)..

Tue Nov 10 00:13:58 2015 - [info] master\_pos\_wait(mysql-bin.000003:307127) completed on slave2(192.168.17.130:3306).  
Executed 0 events.

Tue Nov 10 00:13:58 2015 - [info] done.

Tue Nov 10 00:13:58 2015 - [info] Resetting slave slave2(192.168.17.130:3306) and starting replication from the new master slave1(192.168.17.129:3306)..

Tue Nov 10 00:13:58 2015 - [info] Executed CHANGE MASTER.

Tue Nov 10 00:13:58 2015 - [info] Slave started.

Tue Nov 10 00:13:58 2015 - [info] End of log messages from slave2 ...

Tue Nov 10 00:13:58 2015 - [info]

Tue Nov 10 00:13:58 2015 - [info] -- Slave switch on host slave2 (192.168.17.130:3306) succeeded.

Tue Nov 10 00:13:58 2015 - [info] Unlocking all tables on the orig master:

Tue Nov 10 00:13:58 2015 - [info] Executing UNLOCK TABLES..

Tue Nov 10 00:13:58 2015 - [info] ok.

Tue Nov 10 00:13:58 2015 - [info] Starting orig master as a new slave..

Tue Nov 10 00:13:58 2015 - [info] Resetting slave master(192.168.17.128:3306) and starting replication from the new master slave1(192.168.17.129:3306)..

Tue Nov 10 00:13:58 2015 - [info] Executed CHANGE MASTER.

Tue Nov 10 00:13:58 2015 - [info] Slave started.

Tue Nov 10 00:13:58 2015 - [info] All new slave servers switched successfully.

Tue Nov 10 00:13:58 2015 - [info]

Tue Nov 10 00:13:58 2015 - [info] \* Phase 5: New master cleanup phase..

Tue Nov 10 00:13:58 2015 - [info]

*Tue Nov 10 00:13:58 2015 - [info] slave1: Resetting slave info succeeded.*

*Tue Nov 10 00:13:58 2015 - [info] Switching master to slave1(192.168.17.129:3306) completed successfully.*

这里的切换过程包含以下几方面。

- 1) 将VIP切换到slave1上。
- 2) 设置slave1 (新提升的master) 为读写模式set global read\_only=0。
- 3) 在slave2上执行CHANGE MASTER TO slave1。
- 4) 在原master上解除锁表UNLOCK TABLES。
- 5) 在原master上执行CHANGE MASTER TO slave1。
- 6) 在slave1 (新提升的master) 上执行reset slave all, 清空之前的同步复制信息
- 7) 整个切换流程结束。



## 7.4 MHA高可用架构总结

至此，关于MHA的内容就介绍完了，该技术目前已广泛使用于互联网公司。在切换后保证数据一致性上获得了很好的解决方案，并且是Percona首要推荐，如图7-8所示。

- M-S / M-M with automated failover, continued
  - MMM – Multi-Master Replication Manager
    - Agent-based system. Unreliable agent communication
    - Not sure if it's even still actively being developed. Don't use.
  - MHA – Master High Availability for MySQL
    - Tries very hard to ensure data consistency when promoting a new slave into the master role.
    - Can be dropped into an existing MySQL topology without extensive reconfiguration.
    - The preferred choice of Percona's Remote DBA team.
  - MySQL Utilities
    - New tools from Oracle designed to work with MySQL 5.6 and GTID-based replication. Have yet to see this in the wild.

[www.percona.com](http://www.percona.com)

图7-8 MHA和MMM功能对比

下面是MHA的参数说明，详细内容如表7-1所示。

- Local：表示指每一个配置块内部。Local功能的参数需要放置在[server\_xxx]块下面。
- App：表示参数作用于master/slave。这些参数需要配置在[server\_default]块的下面。
- Global：表示作用于master/slave。Global级别的参数用于管理多组master/slave结构，可以统一化管理一些参数。

表7-1 MHA参数说明



Parameter Name	Required?	Parameter Scope	Default Value	Example
hostname	Yes	Local Only	-	hostname=mysql_server1, hostname=192.168.0.1, etc
ip	No	Local Only	gethostbyname (\$hostname)	ip=192.168.1.3
port	No	Local/App/Global	3306	port=3306

(续)

Parameter Name	Required?	Parameter Scope	Default Value	Example
ssh_host	No	Local Only	same as hostname	ssh_host=mysql_server1, ssh_host=192.168.0.1, etc
ssh_ip	No	Local Only	gethostbyname(\$ssh_host)	ssh_ip=192.168.1.3
ssh_port	No	Local/App/Global	22	ssh_port=22
ssh_connection_timeout	No	Local/App/Global	5	ssh_connection_timeout=20
ssh_options	No	Local/App/Global	""(empty string)	ssh_options="-i /root/.ssh/id_dsa2"
candidate_master	No	Local Only	0	candidate_master=1
no_master	No	Local Only	0	no_master=1
ignore_fail	No	Local Only	0	ignore_fail=1
skip_init_ssh_check	No	Local Only	0	skip_init_ssh_check=1
skip_reset_slave	No	Local/App/Global	0	skip_reset_slave=1
user	No	Local/App/Global	root	user=mysql_root
password	No	Local/App/Global	""(empty string)	password=rootpass
repl_user	No	Local/App/Global	Master_User value from SHOW SLAVE STATUS	repl_user=repl
repl_password	No	Local/App/Global	- (current replication password)	repl_user=replpass
disable_log_bin	No	Local/App/Global	0	disable_log_bin=1
master_pid_file	No	Local/App/Global	""(empty string)	master_pid_file=/var/lib/mysql/master1.pid
ssh_user	No	Local/App/Global	current OS user	ssh_user=root
remote_workdir	No	Local/App/Global	/var/tmp	remote_workdir=/var/log/masterha/app1
master_binlog_dir	No	Local/App/Global	/var/lib/mysql	master_binlog_dir=/data/mysql1./data/mysql2
log_level	No	App/Global	info	log_level=debug
manager_workdir	No	App	/var/tmp	manager_workdir=/var/log/masterha
client_bindir	No	App	-	client_bindir=/usr/mysql/bin
client_libdir	No	App	-	client_libdir=/usr/lib/mysql
manager_log	No	App	STDERR	manager_log=/var/log/masterha/app1.log
check_repl_delay	No	App/Global	1	check_repl_delay=0
check_repl_filter	No	App/Global	1	check_repl_filter=0
latest_priority	No	App/Global	1	latest_priority=0

(续)

Parameter Name	Required?	Parameter Scope	Default Value	Example
multi_tier_slave	No	App/Global	0	multi_tier_slave=1
ping_interval	No	App/Global	3	ping_interval=5
ping_type	No	App/Global	SELECT	ping_type=CONNECT
secondary_check_script	No	App/Global	null	secondary_check_script= masterha_secondary_check -s remote_dc1 -s remote_dc2
master_ip_failover_script	No	App/Global	null	master_ip_failover_script=/usr/ local/custom_script/master_ip_ failover
master_ip_online_change_script	No	App/Global	null	master_ip_online_change_ script= /usr/local/custom_script/ master_ip_online_change
shutdown_script	No	App/Global	null	shutdown_script= /usr/local/ custom_script/master_shutdown
report_script	No	App/Global	null	report_script= /usr/local/ custom_script/report
init_conf_load_script	No	App/Global	null	report_script= /usr/local/ custom_script/init_conf_loader



## 第8章

### MySQL架构演进： “一主多从、读/写分离”

随着网站业务的扩展、数据不断增加、用户也越来越多，单台数据库的压力也就越来越大，只通过数据库参数调整或者SQL优化基本已无法满足要求，这时可以采用读/写分离的策略来改变现状。

数据库层面通常采用的读/写分离技术为一个master数据库（以下简称master库），多个slave数据库（以下简称slave库）。master库负责数据更新和实时数据查询，slave库负责非实时数据查询。在实际的应用中，因为数据库都是读多写少（读取数据的频率高，更新数据的频率相对较少），而读取数据通常耗时比较长，占用的数据库服务器CPU较多，因此也会影响用户体验。对此通常的解决办法就是把查询从主库中抽取出来，采用多个从库，使用负载均衡，减轻每个从库的查询压力。

采用读/写分离技术的目标是：既有效减轻master库的压力，又可以把用户查询数据的请求分发到不同的slave库，从而保证系统的健壮性。

读/写分离的基本原理是：让master库处理事务增、删、改操作（INSERT、DELETE、UPDATE），而让slave库处理SELECT查询操作，replication数据库负责把数据变更同步到集群的slave库中，如图8-1所示。

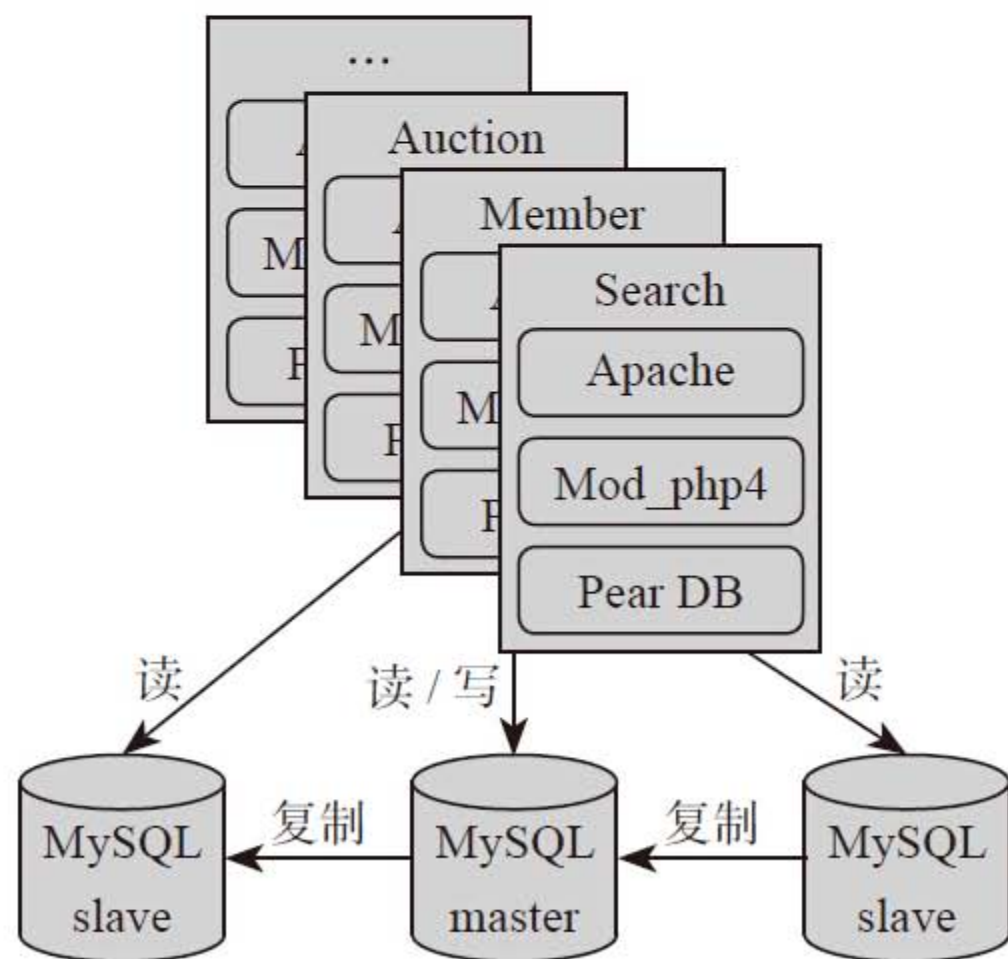


图8-1 读/写分离架构图





## 8.1 实现读/写分离的两种方式

实现读/写分离有两种方式，第一种是通过客户端方式实现，比如PHP Yii框架，第二种是通过Proxy解析SQL的方式实现。通过Yii框架实现读/写分离非常简单，只需要在配置文件中写几个配置参数即可。

首先，配置db.php文件，如图8-2所示。

```

3 return [
4     'class' => 'yii\db\Connection',
5     'charset' => 'utf8',
6     'tablePrefix' => 'pro_',
7     'masterConfig' => [
8         'username' => 'root',
9         'password' => '',
10        'attributes' => [
11
12            PDO::ATTR_TIMEOUT => 10,
13        ],
14    ],
15
16    'masters' => [
17        ['dsn' => 'mysql:host=127.0.0.1;dbname=yii2basic_master'],
18    ],
19
20    'slaveConfig' => [
21        'username' => 'root',
22        'password' => '',
23        'attributes' => [
24
25            PDO::ATTR_TIMEOUT => 10,
26        ],
27    ],
28
29    'slaves' => [
30        ['dsn' => 'mysql:host=127.0.0.1;dbname=yii2basic_slave'],
31    ],
32 ];

```

图8-2 PHP实现读/写分离的代码

在PHP实现读/写分离的过程中，要考虑如下几个问题。

·在主从架构中，如果主从延时、主从数据不一致，怎么办？如果有延迟，能否不把延迟（N秒）的请求转发给这台slave？

·如果是一主多从，那么从库的load balance负载均衡如何实现？当某台slave宕机时，能否不把请求转发给这台slave？当所有的slave不可用时，如何把请求转发给master？

采用PHP Yii框架实现读/写分离时，是需要考虑以上问题的，且需要借助第三方负载均衡软件HAProxy解决这些问题。

HAProxy提供了高可用性、负载均衡以及基于TCP和HTTP应用的代理，它支持虚拟主机，是一种免费、快速并且可靠的解决方案。对于那些负载特大的Web站点来说，HAProxy特别适用。这些站点通常要实现会话保持或七层处理，HAProxy运行在当前的硬件上，完全可以支持数以万计的并发连接。

HAProxy具体以下特性：

- 1) 免费开源，稳定性非常好。
- 2) 根据官方文档，HAProxy使用Myricom厂商的万兆网卡可以将10Gb/s的网络带宽跑满，这个数值作为软件级负载均衡器是相当惊人的。

大多数公司的架构是通过客户端方式实现的：一个主库，多个从库。主库负责写，从库负责查询，主库的高可用性通过MHA ( Master High Availability ) 实现，从库读的负载均衡通过LVS或者HAProxy实现 ( Java框架和PHP框架实现读/写分离 )，如图8-3所示。



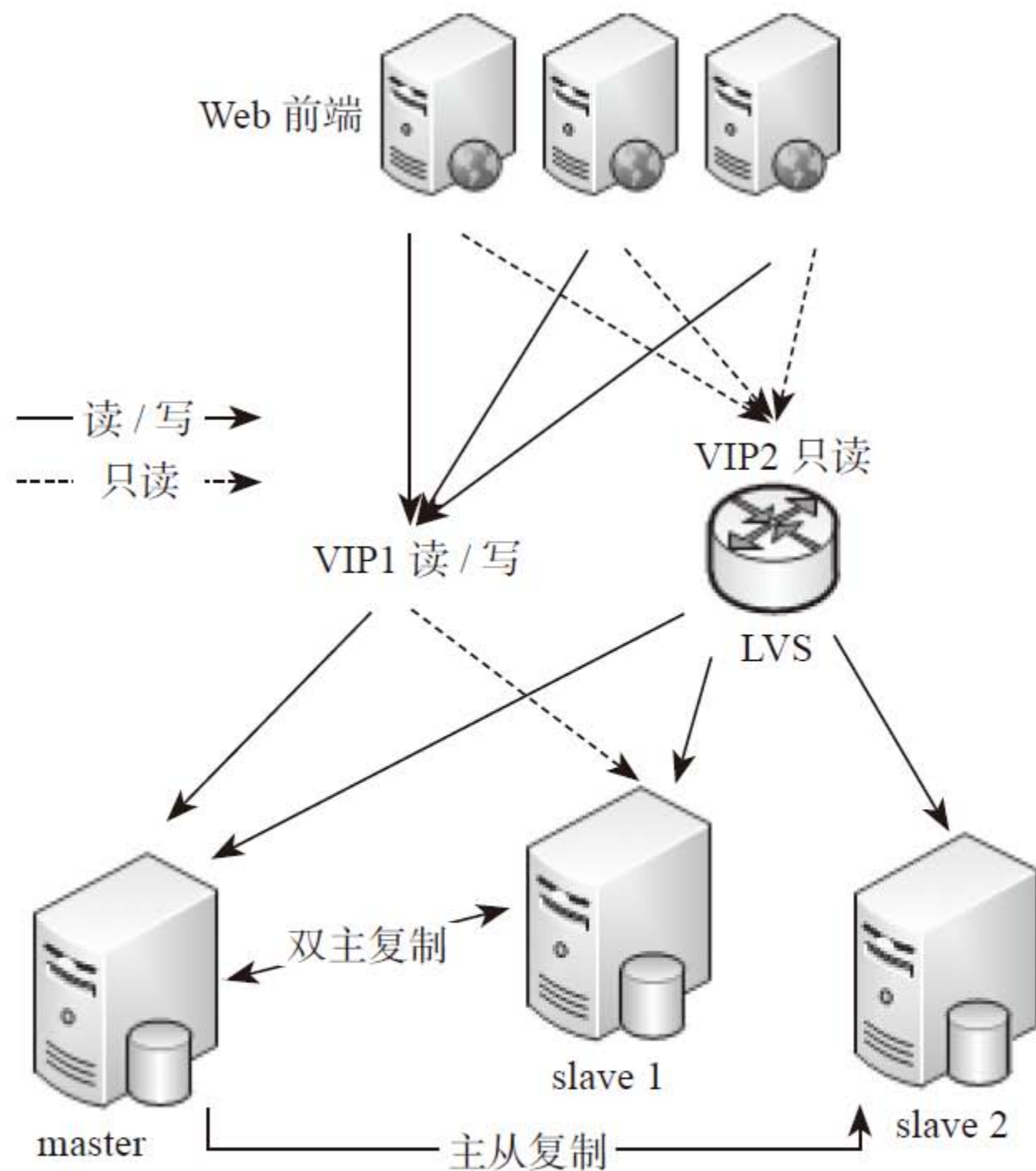


图8-3 MHA+LVS架构

通过HAProxy代理，基于connect方式，以及自定义脚本，可以解决slave延迟或宕机故障转移。

再来看看实现读/写分离的第二种方式，即通过Proxy解析SQL的方式实现。早前，甲骨文公司官方提供了MySQL Proxy，但由于近几年一直没有正式版本，所以无法用在生产上，如图8-4所示。然而，MariaDB于2015年1月14日宣布其旗下的MaxScale发布GA版本。

MaxScale使用C语言开发，利用Linux下的异步I/O功能，使用epoll作为事件驱动框架。它是MariaDB开发的一个数据库智能代理服务，允许根据数据库SQL语句将请求路由到多个服务器，且可设定各种复杂的转向规则。MaxScale可用于透明地提供数据库的负载均衡和高可用性，同时也可提供高度可伸缩和灵活的架构，支持不同的协议和路由决策。

This documentation covers MySQL Proxy 0.8.5. MySQL Proxy contains third-party code. For license information on third-party code, see Licenses for Third-Party Components.



#### Warning

MySQL Proxy is currently an Alpha release and should not be used within production environments.

图8-4 MySQL Proxy不能使用在生产环境中

MaxScale有两种方式实现读/写分离。一种是基于connect的，类似于HAProxy，不解析SQL语句，可以通过PHP Yii框架或Java Mybatis框架实现。在此方式中，用MaxScale做多台slave的负载均衡，并且支持主从同步延迟检测功能，如图8-5所示。

另一种是基于statement的，要解析SQL语句。在这种方式里，前端程序不需要修改，通过MaxScale对SQL语句进行解析，把读/写请求自动路由到后端数据库节点上，从而实现读/写分离。商业软件OneProxy中间件也是基于statement方式实现读/写分离的。

这种方式的好处是不修改程序代码，减少了复杂度，可平滑迁移，无感知；缺点是解析SQL势必会增加CPU的性能损耗，性能没有基于connect的方式好，如图8-6所示。

## MariaDB MaxScale - 基于 connect

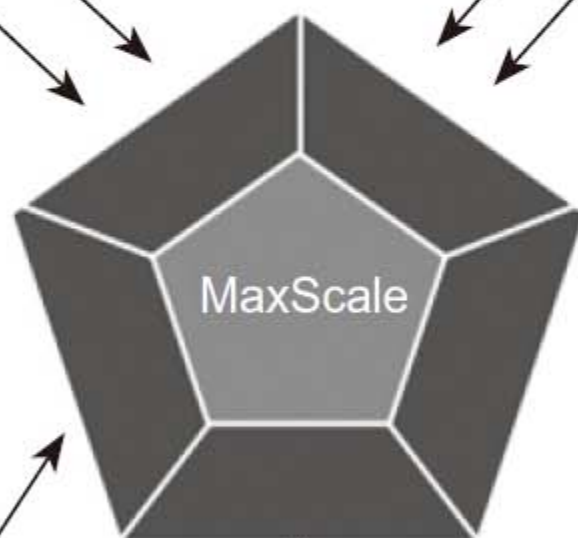


Load Balancing



MySQL Replication+  
Connection Load Balancing

Each application server uses  
2 connections: 1 R/W, 1R



MaxScale connects the R/W  
client connections to the master  
and the R connections are  
load-balanced to all slaves

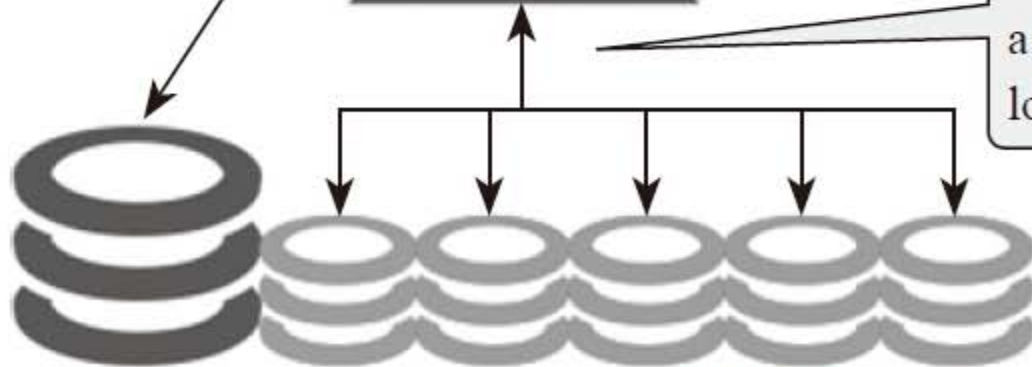


图8-5 基于connect

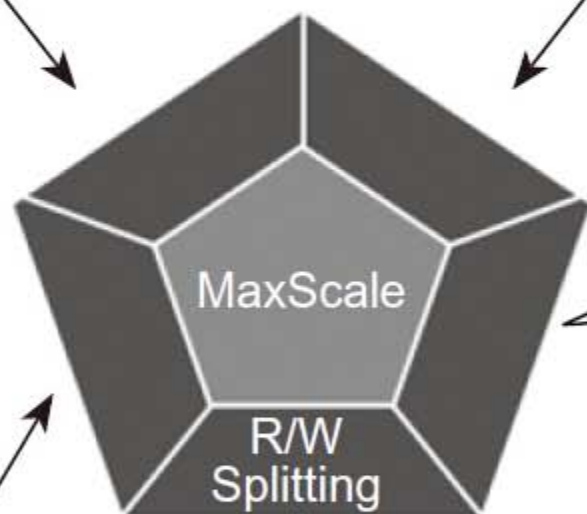




Read/Write Splitting



MySQL Replication+R/W Split routing



Each application server uses only 1 connection

MaxScale monitors the state of each node and only applies operations on available slaves

MaxScale creates 2 connections, one for R/W on the master node and one for R/O load balanced on the slave nodes

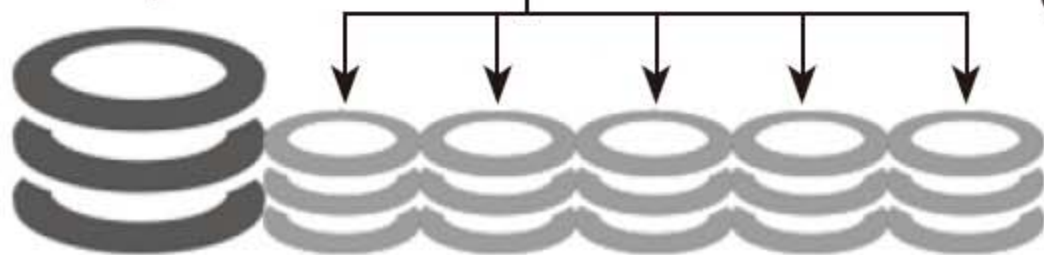


图8-6 基于statement

## 8.2 主从同步延迟的判断标准

在MySQL复制环境中，master将binlog推送到slave（通过io\_thread线程接收并保存在本地relay-log）上，然后通过sql\_thread线程将binlog重放，而Seconds\_Behind\_Master表示本地relay-log中未被执行完的那部分的差值。

我们经常会通过show slave status\G命令查看Seconds\_Behind\_Master的值是否为0，那么Seconds\_Behind\_Master是如何计算的呢？其计算方法是通过当前系统的时间戳减去sql\_thread线程正在执行的binlog event上的时间戳，得到的差值就是Seconds\_Behind\_Master的值。

通过下面的mysqlbinlog命令解析binlog，可以看到SET TIMESTAMP=1447782553命令设置的时间戳。

```
# mysqlbinlog -vv mysql-bin.000143 | grep -C 10 'TIMESTAMP'
### @9=2015-11-18 01:48:36 /* DATETIME meta=0 nullable=0 is_null=0 */
# at 1059
#151118 1:49:13 server id 17708 end_log_pos 1086 Xid = 173726
COMMIT/*!*/;
# at 1086
#151118 1:49:13 server id 17708 end_log_pos 1124 GTID 0-17708-9582
/*!100001 SET @@session.gtid_seq_no=9582/*!*/;
# at 1124
#151118 1:49:13 server id 17708 end_log_pos 1217 Query thread_id=3202492
exec_time=0 error_code=0
use `test`/*!*/;SET TIMESTAMP=1447782553/*!*/;
SET @@session.pseudo_thread_id=3202492/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0,
@@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=2097152/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C utf8 *//*!*/;
SET @@session.character_set_client=33, @@session.collation_connection=33, @@session.collation_server=33/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
```



*TRUNCATE TABLE table1*

*/!\*/;*

事实上，这样的计算是不准确的，可通过以下几个例子进行验证。

示例一：调整系统时间

首先使用sysbench生成一张1000万行的表：

```
# yum install sysbench
# sysbench --test=oltp --mysql-table-engine=innodb
--oltp-table-size=10000000 --mysql-host=127.0.0.1 --mysql-port=3306
--mysql-user=root --mysql-password=123456 --mysql-db=test
--mysql-socket=/tmp/mysql.sock --db-driver=mysql prepare
```

然后在master上全表更新，命令如下：

```
# update sbtest set c='[';
```

执行完毕后，slave就会产生延迟，反复执行show slave status\G去观察Seconds\_Behind\_Master的值，然后在这台slave上把系统时间改成2020年1月1日，会发现Seconds\_Behind\_Master的值瞬间会变得巨大；如果把系统时间改成2010年1月1日，会发现Seconds\_Behind\_Master的值永远为0。

示例二：模拟网络中断

首先，在VMware虚拟机控制台上关闭网卡，如图8-7所示。



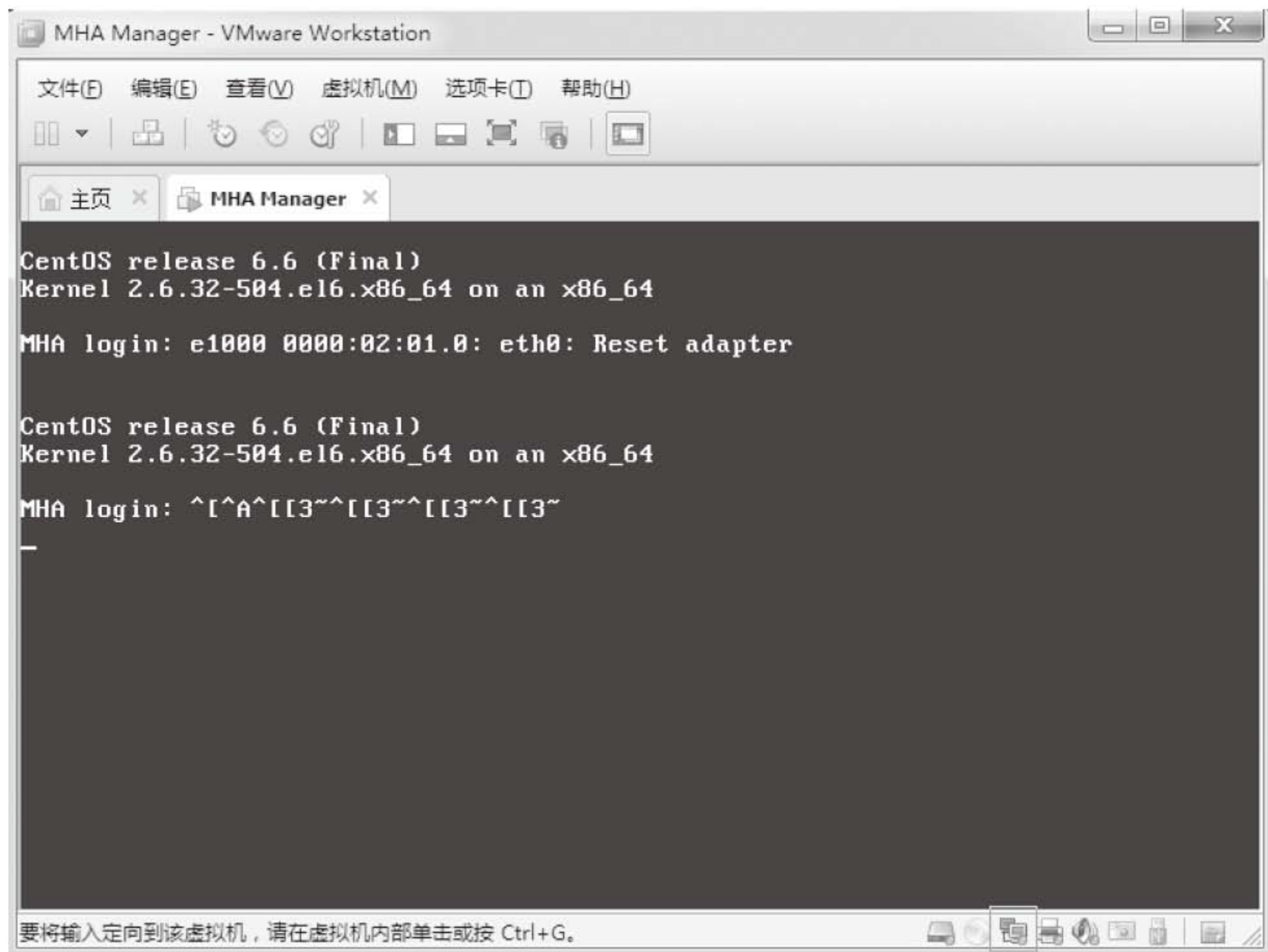


图8-7 VMware断开网卡连接

然后，在master上插入一条数据。

接着，在slave上反复执行show slave status\G。此时观察Seconds\_Behind\_Master的值，发现仍为0，并且显示IO/SQL线程都是正常

的。但由于关闭了网卡，网络不通，master上的binlog是无法推送到slave上的，因此此时的数据是不一致的。

出现上述问题的原因是什么呢？由于默认的参数slave\_net\_timeout是3600秒（1小时），因此只有在master一个小时都没有binlog发送过来时，slave才会尝试重连主库。所以在生产环境下，建议把slave\_net\_timeout调小（比如10秒），以避免出现这种问题。

基于这种情况，可采用Percona工具集pt-heartbeat对主从复制延迟进行检测，其工作方式如下。

·在master上创建一张heartbeat表，按照一定的时间频率更新该表的字段（把时间更新进去）。

·连接到slave上检查复制的时间记录，与slave的当前系统时间进行比较，得出时间的差异。

下面给出pt-heartbeat工具使用的步骤。

1) 在主库上开启守护进程，通过更新heartbeat表来获取主从延迟的差距，命令如下：

```
# pt-heartbeat -S /tmp/mysql.sock --user root --password 123456 --database test --update --create-table --interval=1 --daemonize
```

参数解释如下：

# --create-table

指在主库上创建heartbeat表，默认是不存在的。

Percona官方建议使用MEMORY存储引擎来保存heartbeat表数据，由于heartbeat表每秒都在更新，所以放入内存里是最快的，如图8-8所示。

You must either manually create the heartbeat table on the master or use `--create-table`. See `--create-table` for the proper heartbeat table structure. The **MEMORY** storage engine is suggested, but not required of course, for MySQL.

图8-8 建议使用MEMORY存储引擎

heartbeat表结构如下：

```
CREATE TABLE heartbeat (
  ts varchar(26) NOT NULL,
  server_id int unsigned NOT NULL PRIMARY KEY,
  file varchar(255) DEFAULT NULL, -- SHOW MASTER STATUS
  position bigint unsigned DEFAULT NULL, -- SHOW MASTER STATUS
  relay_master_log_file varchar(255) DEFAULT NULL, -- SHOW SLAVE STATUS
  exec_master_log_pos bigint unsigned DEFAULT NULL -- SHOW SLAVE STATUS
```



);

*alter table heartbeat engine=MEMORY;*

heratbeat表一直在更改ts和position，而ts是检查复制延迟的关键。

再看下面的参数。

*# --interval=1* 检查、更新的间隔时间，默认是1s

*# --daemonize* 执行时，放入后台执行

2) 在从库上执行如下命令：

*# pt-heartbeat -S /tmp/mysql.sock --user root --password 123456 --database test --monitor --master-server-id 128*

其中的参数说明如下：

---monitor代表一直执行，不退出。

---master-server-id后面跟master的server\_id。

---check代表执行一次就退出。

命令执行的效果如图8-9所示。

```
[root@slave1 ~]# /usr/local/bin/pt-heartbeat -S /tmp/mysql.sock --database test --monitor --master-server-id 128
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
0.00s [ 0.00s, 0.00s, 0.00s ]
```

图8-9 延迟监控

其中，0表示从没有延迟。[0.00s, 0.00s, 0.00s]分别表示1m、5m、15m的平均值。可以通过--frames去设置。

3) 用参数--stop关闭master上执行的后台进程，命令如下：

*# pt-heartbeat -S /tmp/mysql.sock --user root --stop*



*Successfully created file /tmp/pt-heartbeat-sentinel*

这样就把在master上开启的进程杀掉了，如果要继续开启后台进程，需要把/tmp/pt-heartbeat-sentinel文件删除，否则启动不了。

通过pt-heartbeat工具可以很好地弥补默认主从延迟的问题，而默认的Seconds\_Behind\_Master值是通过将服务器当前的时间戳与二进制日志中的事件时间戳相对比得到的，所以只有在执行事件时才能报告延时。

## 8.3 HAProxy感知MySQL主从同步延迟

先来看看HaProxy+MySQL主从配置环境，如下。

HAProxy : 192.168.17.131

Master : 192.168.17.128

Slave1 : 192.168.17.129

Slave2 : 192.168.17.130

下面介绍安装过程。

1) 设置host解析。4台服务器的配置如下：

```
# cat /etc/hosts
```

```
192.168.17.128 master
```

```
192.168.17.129 slave1
```

```
192.168.17.130 slave2
```

```
192.168.17.131 HAProxy
```

2) 安装HAProxy，命令如下：

```
# yum install haproxy -y
```

3) 安装rsyslog，命令如下：

```
# yum install rsyslog -y
```

```
# vim /etc/rsyslog.conf
```

之后定义HAProxy的输出log，在最后一行加入如下内容：

```
local2.* /var/log/HAProxy.log
```

```
$ModLoad imudp
```

```
$UDPServerRun 514
```

```
# /etc/init.d/rsyslog restart
```

重启rsyslog服务。

4) 修改文件描述符65535。

```
# vim /etc/security/limits.conf
```

```
* soft nofile 65535
* hard nofile 65535
# vim /etc/sysctl.conf
fs.file-max=655350
net.ipv4.ip_local_port_range = 1025 65000
net.ipv4.tcp_tw_reuse = 1
# 修改完毕后，重启服务器生效
```

5) 配置HAProxy服务。其中包括以下5大部分。

- global：全局配置参数，进程级的，用来控制HAProxy启动前的一些进程及系统设置。
- defaults：配置一些默认的参数，可以被frontend、backend、listen段继承使用。
- frontend：用来匹配接收客户所请求的域名、uri等，并针对不同的匹配做不同的请求处理。
- backend：定义后端服务器集群，以及对后端服务器的一些权重、队列、连接数等选项进行设置，笔者将其理解为Nginx中的upstream块。
- listen：理解为frontend和backend的组合体。

配置文件说明如下。

```
# cat /etc/HAProxy/HAProxy.cfg
global
# 全局参数的设置
log 127.0.0.1 local2
# 全局的日志配置，使用log关键字，指定使用127.0.0.1上syslog服务中的local2日志设备，记录日志等级为warning的日志
maxconn 65535
# 定义haproxy进程的最大连接数
user haproxy
group haproxy
# 设置运行HAProxy的用户和组
daemon
# 以守护进程的方式运行
nbproc 24
# 设置HAProxy启动时的进程数，该值应该和服务器的CPU核心数一致，比如常见的2颗12核CPU的服务器共有24核，则可以将其值设置为：
<=24，创建多个进程数，可以减少每个进程的任务队列，但是过多的进程数也可能导致进程的崩溃。
defaults
```



# 配置默认的参数

log global

# 继承global中log的定义

option tcplog

# 启用日志记录TCP请求

option dontlognull

# 启用该项，日志中将不会记录空连接。空连接指在上游的负载均衡器或监控系统中，为了探测该服务是否存活可用而定期连接或获取某一固定的组件或页面，或者探测扫描端口是在监听还是处于开放状态等；官方文档中标注，如果该服务上游没有其他的负载均衡器，那么建议不要使用该参数，因为使用后互联网上的恶意扫描或其他动作就不会被记录下来

retries 3

# 定义连接后端服务器的失败重连次数，连接失败次数超过此值后会将对应后端服务器标记为不可用

option redispatch

# 若使用了cookie，HAProxy会将其请求的后端服务器的serverID插入cookie中，以保证会话的SESSION持久性；此时，如果后端的服务器宕机，那么客户端的cookie是不会刷新的。如果设置此参数，则会将客户的请求强制定向到另外一个后端server上，以保证服务的正常。

option abortonclose

# 当服务器负载很高时，自动结束当前队列处理比较久的链接

timeout connect 10s

# 设置成功连接到一台服务器的最长等待时间

timeout client 2m

# 设置客户端发送数据时的最长等待时间

timeout server 2m

# 设置服务器端回应客户端发送数据时的最长等待时间

timeout queue 1m

# 设置一个请求在队列里的超时时间

frontend mysqlcluster-front

# 定义一个名为mysqlcluster-front的前端部分

bind \*:3320

# 应用端PHP/JAVA连接HAProxy的端口号

mode tcp

# tcp是四层模式

```
default_backend mysqlcluster-back
# 定义一个名为mysqlcluster-back的后端部分
frontend stats-front
# 定义一个名为stats-front的前端部分
bind *:80
# HAProxy监控页面的端口号
mode http
# http是七层模式
default_backend stats-back
# 定义一个名为stats-back的后端部分
backend mysqlcluster-back
mode tcp
balance roundrobin
# 轮询模式
option httpchk
# 开启对后端服务器的健康检测
server 192.168.17.128 192.168.17.128:3306 check port 9200 inter 2000 rise 3 fall 3 backup
# 正常的服务器全部都宕机时，才会启用备份服务器[backup]
server 192.168.17.129 192.168.17.129:3306 check port 9200 inter 2000 rise 3 fall 3 weight 10
# port 9200端口是调用后端DB的服务，用来感知MySQL同步延迟
# weight 10代表权重，值越小，转发的请求就越少
server 192.168.17.130 192.168.17.130:3306 check port 9200 inter 2000 rise 3 fall 3 weight 10
backend stats-back
mode http
stats uri /HAProxy/stats
# 定义监控页面URL地址
stats auth admin:123456
# 定义页面访问的用户名和密码
stats refresh 3s
# 定义每3秒自动刷新页面
```



6) 启动HAProxy服务，命令如下：

```
# /etc/init.d/haproxy start
```

7) 访问监控页面URL。打开浏览器，输入URL：<http://192.168.17.131/haproxy/stats>，第一次会让你输入用户名和密码，如图8-10所示。

这里输入刚才定义好的用户名，为admin，密码为123456。



图8-10 输入用户名和密码

图8-11是进入后的监控页面，这里看到从库192.168.17.129/130是DOWN状态，是因为HAProxy在调用后端slave的9200端口服务，我们还没有在该机器里配置同步监测脚本，所以这里才显示DOWN状态。



## HAProxy version 1.5.4, released 2014/09/02

### Statistics Report for pid 2605

#### > General process information

pid = 2605 (process #1, ntprec = 1)  
 uptime = 0c 0h04m25s  
 system limits: memmax = unlimited; ulimit-n = 131085  
 maxsock = 131085; maxconn = 65535; maxpipes = 0  
 current conns = 1; current pipes = 0/0; conn rate = 1/sec  
 Running tasks: 1/0; idle = 100 %

active UP      backup UP  
 active UP, going down      backup UP, going down  
 active DOWN, going up      backup DOWN, going up  
 active or backup DOWN      not checked  
 active or backup DOWN for maintenance (MAINT)  
 active or backup SOFT STOPPED for maintenance  
 Note: 'NOLB'/'DRAIN' = UP with load-balancing disabled.

Display option:

• Scope:   
 • Hide 'DOWN' servers  
 • Disable refresh  
 • Refresh now  
 • CSV export

External resources:

• [Primary site](#)  
 • [Updates \(v1.5\)](#)  
 • [Online manual](#)

#### mysqlcluster-front

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bsk	Chk	Dwn	Downtime	Thrtle
Frontend				0	0		0	0		65 836	0		0	0	0	0	0					OPEN								

#### stats-front

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bsk	Chk	Dwn	Downtime	Thrtle
Frontend				1	1		1	1		65 506	2		264	262	0	0	0					OPEN								

#### mysqlcluster-back

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bsk	Chk	Dwn	Downtime	Thrtle
192.168.17.128	0	0	-	0	0		0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	4m25s UP	* L7OW/200 in 990ms	1	-	Y	0	0	0s	-
192.168.17.129	0	0	-	0	0		0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	4m23s DOWN	L7STS/503 in 1002ms	10	Y	-	1	1	4m23s	-
192.168.17.130	0	0	-	0	0		0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	4m22s DOWN	L7STS/503 in 1002ms	10	Y	-	1	1	4m22s	-
Backend	0	0		0	0		0	0	6 504	0	0	?	0	0	0	0	0	0	0	0	0	4m25s UP		1	0	1		0	0s	

#### stats-back

	Queue			Session rate			Sessions					Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bsk	Chk	Dwn	Downtime	Thrtle
Backend	0	0		1	1		1	1	6 504	2	0	0s	264	262	0	0	0	0	0	0	0	4m25s UP		0	0	0		0		

图8-11 HAProxy监控页面

8) 配置slave的同步检测9200端口服务，命令如下：

```
# yum install xinetd -y
```

```
# vim /etc/services
```

增加9200服务时，要在最后一行加入如下内容：

```
mysqlchk 9200/tcp # mysqlchk
```

```
# vim /etc/xinetd.d/mysqlchk
```

修改mysqlchk文件，添加如下内容：

```
# default: on
```

```
# description: mysqlchk
```

```

service mysqlchk
{
# this is a config for xinetd , place it in /etc/xinetd.d/
  disable = no
  flags = REUSE
  socket_type = stream
  port = 9200
  wait = no
  user = nobody
  server = /usr/bin/replication_check
  log_on_failure += USERID
  only_from = 0.0.0.0/0
  # recommended to put the IPs that need
  # to connect exclusively (security purposes)
  per_source = UNLIMITED
}

```

然后创建如下同步延迟检测脚本，这个脚本就是HAProxy调用的。

```
# vim /usr/bin/replication_check
```

这里定义的是延迟小于10秒时，为OK状态，超过10秒认定该slave宕机，不会把请求转发给它。修改replication\_check脚本，添加如下内容：

```

#!/bin/bash
Master_server_id=128
Seconds_Behind_Master=$(/usr/local/bin/pt-heartbeat -S /tmp/mysql.sock --user root --password 123456 --database test
--check --master-server-id $Master_server_id)
result=`echo ${Seconds_Behind_Master%. *}`
if [ $result -lt 10 ]
then
  # mysql is fine , return http 200
  /bin/echo -e "HTTP/1.1 200 OK \n"

```

```

/bin/echo -e "Content-Type: Content-Type: text/plain \n"
/bin/echo -e "\r\n"
/bin/echo -e "MySQL is running. \n"
/bin/echo -e "\r\n"
else
    # mysql is fine , return http 503
/bin/echo -e "HTTP/1.1 503 Service Unavailable \n"
/bin/echo -e "Content-Type: Content-Type: text/plain \n"
/bin/echo -e "\r\n"
/bin/echo -e "MySQL is *down*. \n"
/bin/echo -e "\r\n"
fi

```

赋予replication\_check可执行权限，命令如下：

```

# chmod 755 /usr/bin/replication_check
# /usr/bin/replication_check
HTTP/1.1 200 OK
Content-Type: Content-Type: text/plain
MySQL is running.
# /etc/init.d/xinetd start

```

从图8-12可以看到端口已经启动。

```

[root@slave1 ~]# netstat -ntlp | grep 9200
tcp        0      0 :::9200          :::*              LISTEN      16791/xinetd
[root@slave1 ~]#

```

图8-12 xinetd服务启动

通过监控页面可以看到全部slave均正常，如图8-13所示。



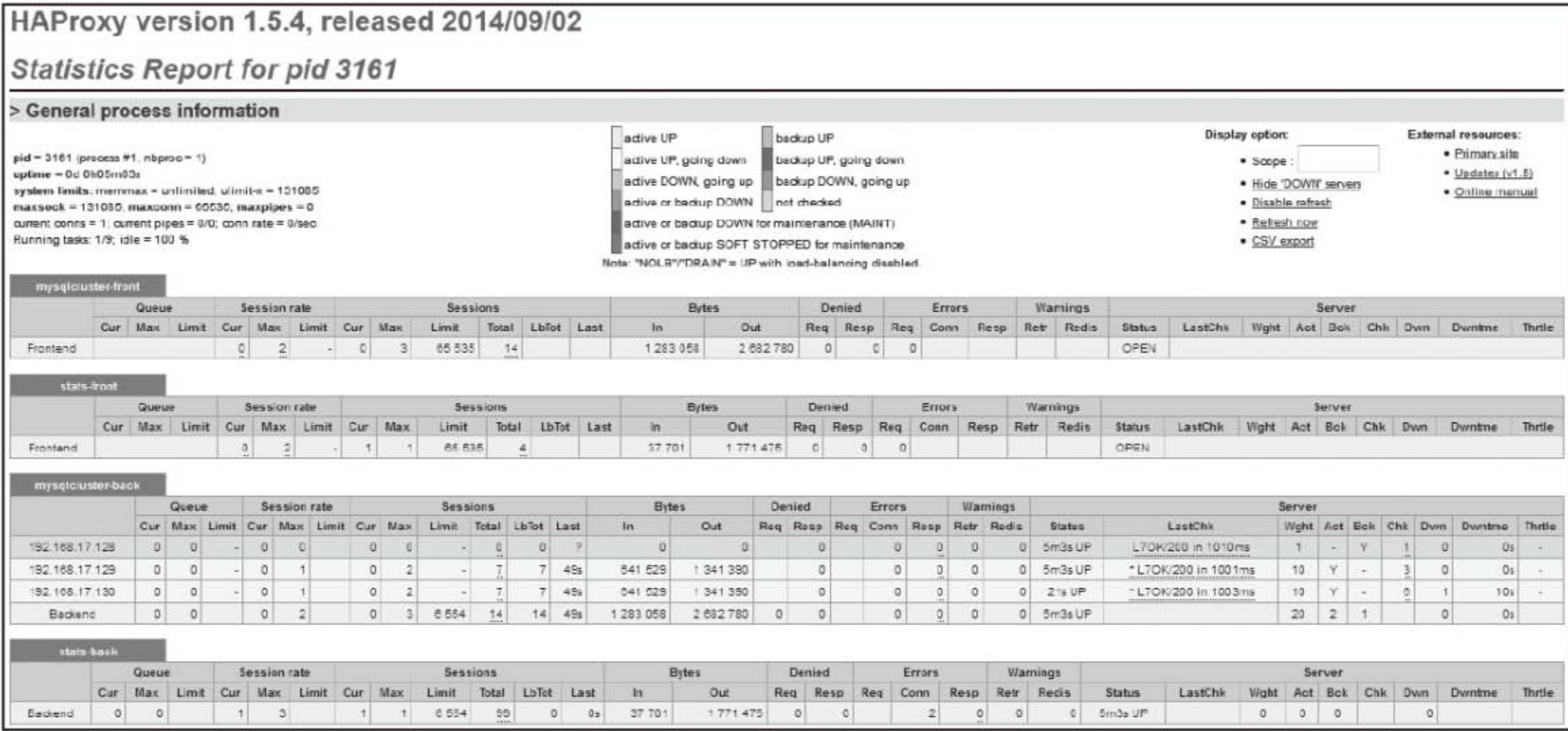


图8-13 HAProxy监控页面

同步延迟和故障切换测试过程如下。

测试之前，要在master ( 192.168.17.128 )、slave1 ( 192.168.17.129 )、slave2 ( 192.168.17.130 ) 上均打开General Log，命令如

下：

Set global general\_log=1;

( 1 ) 模拟slave延迟

在slave2 ( 192.168.17.130 ) 上执行flush tables with read lock全局读锁，通过客户端连接HAProxy 3320端口，命令如下：

mysql -h192.168.17.131 -uadmin -p123456 -P3320 test -e "select \* from t1 where id =1;"

这时通过查看General Log，可以发现复制延迟超过10秒（/usr/bin/replication\_check脚本里定义）的客户端请求不会转发到slave2上。

### （2）模拟一台slave故障

在slave2（192.168.17.130）上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接HAProxy 3320端口，命令如下：  
*mysql -h192.168.17.131 -uadmin -p123456 -P3320 test -e "select \* from t1 where id =1;"*

这时通过查看General Log可以发现客户端请求不会转发到slave2上。

### （3）模拟全部的slave故障

在slave1（192.168.17.129）和slave2（192.168.17.130）上执行stop slave io\_thread或者stop slave sql\_thread，通过客户端连接HAProxy 3320端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P3320 test -e "select \* from t1 where id =1;"*

这时通过查看General Log可以发现所有的请求都只会转发给master（192.168.17.128），slave上并不会会有请求转发，实现了平滑故障转移。

综合来看，这个架构是存在一定问题的，虽然解决了slave延迟和宕机的问题，但细心的读者会发现，主从延迟判断是依赖pt-heartbeat的，master上需要启动一个pt-heartbeat守护进程。假如master（server\_id=128）宕机，那么新的slave（server\_id=129）提升为master，此时就需要人工再次启动pt-heartbeat守护进程，而slave2（server\_id=130）还需要修改/usr/bin/replication\_check脚本，将里面的pt-heartbeat--master-server-id 128改为--master-server-id 129。但在这之前，由于master的宕机，slave（server\_id=129）通过如下脚本检测到同步复制失败，因此HAProxy会将两台slave（server\_id=128/129）进行下线处理，此时整个业务瘫痪，无法做到无感知平滑故障切换。

*# pt-heartbeat -S /tmp/mysql.sock --user root --password 123456 --database test --check --master-server-id 128*



## 8.4 读/写分离MariaDB MaxScale架构搭建演示

### 8.4.1 配置环境及安装介绍

先来看看搭建该架构的配置环境，如下。

MaxScale : 192.168.17.131

Master : 192.168.17.128

Slave1 : 192.168.17.129

Slave2 : 192.168.17.130

下面介绍安装过程。

1) 设置hosts解析。4台服务器的配置如下：

```
# cat /etc/hosts
```

```
192.168.17.128 master
```

```
192.168.17.129 slave1
```

```
192.168.17.130 slave2
```

```
192.168.17.131 MaxScale
```

2) 安装MaxScale。其官网下载地址为：[https://mariadb.com/my\\_portal/download/maxscale](https://mariadb.com/my_portal/download/maxscale)。

安装命令如下：

```
# rpm -ivh maxscale-1.2.1-1.rhel6.x86_64.rpm
```

3) 创建加密密钥，命令如下：

```
# maxkeys /var/lib/maxscale/
```

密钥文件.secrets存放在/var/lib/maxscale/目录下。

4) 创建加密密码，命令如下：

```
# maxpasswd /var/lib/maxscale/ 123456
```

```
D0E432F8DC9919A6F1F8C7D0AB57E981
```





# 这里是对密码123456做加密

5) 修改文件描述符65535以达到tcp的最大连接数, 命令如下:

```
# vim /etc/security/limits.conf
```

```
* soft nofile 65535
```

```
* hard nofile 65535
```

```
# vim /etc/sysctl.conf
```

```
fs.file-max=655350
```

```
net.ipv4.ip_local_port_range = 1025 65000
```

```
net.ipv4.tcp_tw_reuse = 1
```

修改完毕后, 通过reboot重启服务器使之生效。

6) 配置MaxScale服务, 配置文件如下:

```
# cat /etc/maxscale.cnf
```

```
[maxscale]
```

```
threads=1
```

```
## 如果CPU是24核的, 那么线程设置为12核即可
```

```
[MySQL Monitor]
```

```
type=monitor
```

```
module=mysqlmon
```

```
servers=server1, server2, server3
```

```
user=admin
```

```
# MaxScale监控账号
```

```
passwd=6590A334C68DE06B41847F38AF7E8F24
```

```
monitor_interval=10000
```

```
# 默认每隔10秒执行监控检查
```

```
detect_stale_master=1
```

```
# 当所有的slave都不可用时, select查询请求会转发给master, 默认为关闭, 设置为1开启。
```

```
detect_replication_lag=1
```

```
# 开启同步复制延迟检查, 默认为关闭, 设置为1开启。
```

```
[RW Split Router]
```



```

# 基于statement SQL解析的方式
type=service
router=readwritesplit
servers=server1 , server2 , server3
user=rw
# 应用读/写分离账号
passwd=6590A334C68DE06B41847F38AF7E8F24
max_slave_replication_lag=5
# 定义超过延迟5秒，把请求转发给其他slave
max_slave_connections=100%
# 所有的slave提供select查询服务
#weightby=serv_weight
# 定义权重
use_sql_variables_in=all
#如果程序需要预先设置会话变量，如Discuz论坛程序，那么每次连接数据库都要执行如下命令。
#SET character_set_connection=utf8 , character_set_results=utf8 ,
#character_set_client=binary , sql_mode=""
# 默认是all，会在后端master和slave上执行，设置参数为master
# 将只会路由到master上执行
# enable_root_user=1
# 默认禁止root超级权限用户访问，设置为1表示开启
[Read Connection Router]
# 基于connect的方式
type=service
router=readconnroute
servers=server1 , server2 , server3
user=rw
passwd=6590A334C68DE06B41847F38AF7E8F24
router_options=slave
max_slave_replication_lag=5

```

```
# 定义超过延迟5秒，则把请求转发给其他slave
[CLI]
type=service
router=cli
[RW Split Listener]
type=listener
service=RW Split Router
protocol=MySQLClient
port=4006
# 读/写分离端口，应用连接这个端口
[Read Connection Listener]
type=listener
service=Read Connection Router
protocol=MySQLClient
port=4008
# slave负载均衡的端口，应用连接这个端口
[CLI Listener]
type=listener
service=CLI
protocol=maxscaled
port=6603
# MaxScale后台管理端口
[server1]
type=server
address=192.168.17.128
port=3306
protocol=MySQLBackend
#serv_weight=1
# 权重自定义
[server2]
```



```
type=server
address=192.168.17.129
port=3306
protocol=MySQLBackend
#serv_weight=10
# 权重自定义
[server3]
type=server
address=192.168.17.130
port=3306
protocol=MySQLBackend
#serv_weight=10
# 权重自定义
```

7) 启动服务，命令如下：

```
/etc/init.d/maxscale start
```

安装完成之后，再来说明前端PHP/Java程序的接入注意事项。

以下情况，select查询将会在master上执行。

·当事务里有SQL语句 ( begin;select;commit; ) 时。

·在Java预编译语句prepared statement中执行SQL时，如下代码所示：

```
protected boolean updateSalary(Connection conn, BigDecimal x, String ID) throws SQLException{
    PreparedStatement pstmt = null;
    try {
        pstmt = conn.prepareStatement("UPDATE EMPLOYEES SET SALARY = WHERE ID = ");
        pstmt.setBigDecimal(1, x);
        pstmt.setString(2, ID);
        return true;
    } finally
    if (pstmt != null) {
        pstmt.close();
    }
}
```

```
}
```

```
}
```

·执行存储过程或者函数时。



注意 目前MaxScale 1.2.1版本不支持水平分库分表，将在未来的版本里支持。

## 8.4.2 基于connect方式的测试

测试之前，在slave1 ( 192.168.17.129 )、slave2 ( 192.168.17.130 ) 上打开General Log，命令如下：

*Set global general\_log=1;*

( 1 ) 从库进行Load Balance负载均衡测试

通过客户端连接MaxScale 4008端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4008 test -e "select \* from t1 where id =1;"*

查看General Log，得知把请求轮训转发给了slave1和slave2。

( 2 ) 模拟slave延迟

在slave2 ( 192.168.17.130 ) 上执行flush tables with read lock全局读锁，通过客户端连接MaxScale 4008端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4008 test -e "select \* from t1 where id =1;"*

查看General Log，得知复制延迟超过5秒的客户端请求仍旧会转发到slave2上。基于connect方式在MaxScale 1.2.1版本里，无法实现延迟检测功能。

( 3 ) 模拟一台slave故障

在slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接MaxScale 4008端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4008 test -e "select \* from t1 where id =1;"*

这时通过查看General Log得知客户端请求将不会转发到slave2上。

( 4 ) 模拟全部的slave故障

在slave1 ( 192.168.17.129 ) 和slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接MaxScale 4008端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4008 test -e "select \* from t1 where id =1;"*

这时通过查看General Log得知所有的请求将只转发给master ( 192.168.17.128 )，slave上并不会会有请求转发，实现了平滑故障转移。





### 8.4.3 基于statement方式 (SQL解析) 的测试

在master ( 192.168.17.128 )、slave1 ( 192.168.17.129 )、slave2 ( 192.168.17.130 ) 上均打开General Log，命令如下：

*Set global general\_log=1;*

#### (1) 读/写分离测试

通过客户端连接MaxScale 4006端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4006 test -e "select \* from t1 where id =1;"*

*mysql -h192.168.17.131 -uadmin -p123456 -P4006 test -e "insert into t1 values(10);"*

查看General Log得知select语句会把请求转发给slave1或slave2。insert语句会把请求转发给master。

#### (2) 模拟slave延迟

在slave2 ( 192.168.17.130 ) 上执行flush tables with read lock全局读锁，通过客户端连接MaxScale 4006端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4006 test -e "select \* from t1 where id =1;"*

查看General Log，得知复制延迟超过5秒的客户端请求将不会转发给slave2，而是转发给没有延迟的slave1，当slave1和slave2都出现延迟超过5秒时，请求会强制转发给master。

#### (3) 模拟一台slave故障

在slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接MaxScale 4006端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4006 test -e "select \* from t1 where id =1;"*

这时通过查看General Log，得知客户端请求将不会转发到slave2上。

#### (4) 模拟全部的slave故障

在slave1 ( 192.168.17.129 ) 和slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接MaxScale 4006端口，命令如下：

*mysql -h192.168.17.131 -uadmin -p123456 -P4006 test -e "select \* from t1 where id =1;"*

这时通过查看General Log，得知所有的请求将只转发给master ( 192.168.17.128 )，slave上并不会会有请求转发，实现了平滑故障转移。



注意 MaxScale无法对master进行故障转移切换，这里需要借助第三方工具，如MHA。但故障切换后，MaxScale可以自动识别哪台机器是master。另外，该架构必须是一主带N从，不能是双主带N从。



#### 8.4.4 MaxScale延迟检测

可能有读者会问，MaxScale延迟检测是如何计算的？类似pt-heartbeat那样，在主库上创建一张表并且更新它，命令如下：

```
4612 Query CREATE DATABASE IF NOT EXISTS maxscale_schema
4612 Query CREATE TABLE IF NOT EXISTS
maxscale_schema.replication_heartbeat (maxscale_id INT NOT NULL , master_server_id INT NOT NULL , master_timestamp INT
UNSIGNED NOT NULL , PRIMARY KEY
( master_server_id , maxscale_id )) ENGINE=MYISAM DEFAULT CHARSET=latin1
4612 Query BEGIN
4612 Query DELETE FROM maxscale_schema.replication_heartbeat
WHERE master_timestamp < 1448163255
4612 Query COMMIT
4612 Query BEGIN
181 Query SELECT master_timestamp FROM
maxscale_schema.replication_heartbeat WHERE maxscale_id = 6603 AND
master_server_id = 128
4612 Query UPDATE maxscale_schema.replication_heartbeat SET
master_timestamp = 1448336055 WHERE master_server_id = 128 AND maxscale_id = 6603
4612 Query COMMIT
```

MaxScale会对master和slave上replication\_heartbeat表字段的master\_timestamp时间戳进行对比，相减得出差异，该差值即为MySQL主从同步复制的延迟值。

登录MaxScale后台管理命令如下，后台前的所有命令如图8-14所示。

```
# maxadmin -uadmin -pmariadb -P6603
```



```
MaxScale> help
Available commands:
  add user
  clear server
  disable [heartbeat|log|sessionlog|root|feedback]
  enable [heartbeat|log|sessionlog|root|feedback]
  flush [log|logs]
  list [clients|dcbs|filters|listeners|modules|monitors|services|servers|sessions|threads]
  reload [config|dbusers]
  remove user
  restart [monitor|service]
  set [server|pollsleep|nbpolls]
  show [dcbs|dcb|dbusers|epoll|eventq|eventstats|feedbackreport|filter|filters|modules|monitor|monitors|server|servers|serversjson|services|service|session|sessions|tasks|threads|users]
  shutdown [maxscale|monitor|service]
```

图8-14 MaxScale后台管理所有命令

这里列出了帮助信息，比如想要查看后端DB的信息，输入图8-15所示的命令，可以看见当前的连接数是2个，master和slave运行正常。通过图8-16所示的命令可查看后端DB更详细的信息。

```
MaxScale> list servers
Servers.
```

Server	Address	Port	Connections	Status
server1	192.168.17.128	3306	2	Master, Running
server2	192.168.17.129	3306	2	Slave, Running
server3	192.168.17.130	3306	2	Slave, Running

图8-15 查看主从运行状况

```
MaxScale> show servers
Server 0x1c571d0 (server1)
  Server: 192.168.17.128
  Status: Master, Running
  Protocol: MySQLBackend
  Port: 3306
  Server Version: 10.1.8-MariaDB-log
  Node Id: 128
  Master Id: -1
  Slave Ids: 129, 130
  Repl Depth: 0
  Last Repl Heartbeat: 1448459901
  Number of connections: 7
  Current no. of conns: 0
  Current no. of operations: 0
Server 0x1c570c0 (server2)
  Server: 192.168.17.129
  Status: Slave, Running
  Protocol: MySQLBackend
  Port: 3306
  Server Version: 10.1.8-MariaDB-log
  Node Id: 129
  Master Id: 128
  Slave Ids:
  Repl Depth: 1
  Slave delay: 0
  Last Repl Heartbeat: 1448459901
  Number of connections: 7
  Current no. of conns: 0
  Current no. of operations: 0
Server 0x1bb3a10 (server3)
  Server: 192.168.17.130
  Status: Slave, Running
  Protocol: MySQLBackend
```

图8-16 后端DB的详细信息

通过图8-17所示的命令查看定义的读/写分离和从库负载均衡服务。

```
MaxScale> list services
Services.
```

Service Name	Router Module	#Users	Total Sessions
RW Split Router	readwritesplit	3	4
Read Connection Router	readconnroute	1	1
CLI	cli	2	2

图8-17 查看服务

通过图8-18所示的命令查看更详细的服务信息。

```
MaxScale> show services
Service 0x1c58480
  Service: RW Split Router
  Router: readwritesplit (0x7f8c78a23720)
  State: Started
  Number of router sessions: 7
  Current no. of router sessions: 0
  Number of queries forwarded: 10008
  Number of queries forwarded to master: 0
  Number of queries forwarded to slave: 10008
  Number of queries forwarded to all: 8
  Master/Slave percentage: 0.00%
  Started: Wed Nov 25 17:17:11 2015
  Root user access: Disabled
  Backend databases
    192.168.17.130:3306 Protocol: MySQLBackend
    192.168.17.129:3306 Protocol: MySQLBackend
    192.168.17.128:3306 Protocol: MySQLBackend
  Users data: 0x1c5a660
  Total connections: 8
  Currently connected: 1
  SSL: Disabled
```



图8-18 查看服务信息

通过图8-19所示的命令查看客户端连接MaxScale的IP地址。

```
MaxScale> list clients
```

Client Connections			
Client	DCB	Service	Session
127.0.0.1	0x1c6f670	CLI	0x1c6f860
192.168.17.131	0x1c6fa30	RW Split Router	0x1c700d0
192.168.17.131	0x1c6fd00	RW Split Router	0x1c63310

图8-19 查看客户端的IP地址

对于这部分内容，就不提供案例了，在如下地址中有一个MariaDB MaxScale平滑接入Discuz论坛/ECshop电商购物平台的案例：<http://edu.51cto.com/lesson/id-80861.html>。

## 8.5 读/写分离OneProxy介绍及架构搭建演示

本节内容以会员信息为例进行说明。假设你的业务突飞猛进，现在每天有上千万的页面访问量或会员登录，这时若还只用一台机器来提供会员信息的服务可能会顶不住或运行极不稳定。由于会员信息具有以读为主的业务特征，因此可以考虑复制多份会员信息，将读流量分担出去，从而进行架构的横向扩展，以保证业务的稳定。如果希望对应用完全透明，那么后端的读节点可以根据需要进行透明伸缩。

OneProxy可在自动单点切换的功能基础上进行扩展，实现对应用透明的读/写分离功能。它所具有的内置故障检测机制可以在1秒钟内发现后端故障，并主动踢除出问题的数据库，无须向应用推送后端数据库的运行状态，以实现轻松运行维护。目前已在Zabbix、PHP、PHPWind、Discuz和Java JDBC上验证过，越来越多的用户正在选择用OneProxy来保障以读为主的互联网业务。

### 8.5.1 OneProxy简介

OneProxy平民软件是完全自主开发的分布式数据访问层，可帮助用户在MySQL/PostgreSQL集群上快速搭建支持分库分表的分布式数据库中间件，也是一款具有SQL白名单（防SQL注入）及IP白名单功能的SQL防火墙软件。它采用与MySQL Proxy一致的反向协议输出模式，应用非常简单且透明，这让用户畏惧的分库分表（Horizontal Partitioning）工作变得极其简单可控！它基于Libevent机制实现，单个实例可以实现25万个的SQL转发能力，用一个OneProxy节点可以带动整个MySQL集群。



## 8.5.2 OneProxy的功能及安装介绍

从信息服务的角度来看，数据库层对上的服务有三种行为：读取、存入和计算，比如页面显示主要是读取，交易处理主要是写入，而报表汇总则是计算。基本的信息业务都可以分解为由这三种行为的一定比例构成。对任何一种行为来讲，都需要考虑服务能力的扩展（Capacity）及服务的可靠性（Availability），比如在京东、淘宝、支付宝等网站上购买东西时，可能需要先登录，登录过程其实是一个信息的读取行为，考虑到他们的用户量，一定要求做到系统可以随时扩容，并且当一台物理机器发生故障时，服务能不受影响。对于以读取信息为主的服务，可以考虑将数据复制多份，并且当用户登录时，可以从任何一份复制的数据里进行检索，这样既扩展了系统的信息读取能力，也不会让任何一份数据复制失效从而产生大的影响。

数据库本身都提供了数据复制的便捷手段，比如Oracle的Active DataGuard、MySQL的Replication等，为了简化事务管理，数据库的复制节点通常是只读的，称为从节点或备节点。而用于写入的那个原始节点则称为主节点，主节点和从节点之间构成一套主从节点集群或主备节点集群。由于从节点或备节点只能读取不能写入，所以要扩展读能力，则先要做读/写分离，要实现该功能首先要梳理和修改应用代码，进行详尽的线下测试，再进行仔细的应用发布，然后修改应用程序对读取操作的数据源管理和选择方式，这可能是一个非常复杂的软件项目。

当后台用的是MySQL数据库或兼容MySQL协议的数据库时，就可以不用修改应用程序，使用OneProxy透明地实现读/写分离功能，因为OneProxy可以识别MySQL协议，可以清晰地识别协议包中的SQL语句，可对不同类型的语句进行透明地转发处理，自然也就可以实现对应用透明的读/写分离了，完全不需要启动前面所说的复杂的软件项目，架构如图8-20所示。

OneProxy+MySQL的配置环境如下。

OneProxy : 192.168.17.131

Master : 192.168.17.128

Slave1 : 192.168.17.129

Slave2 : 192.168.17.130

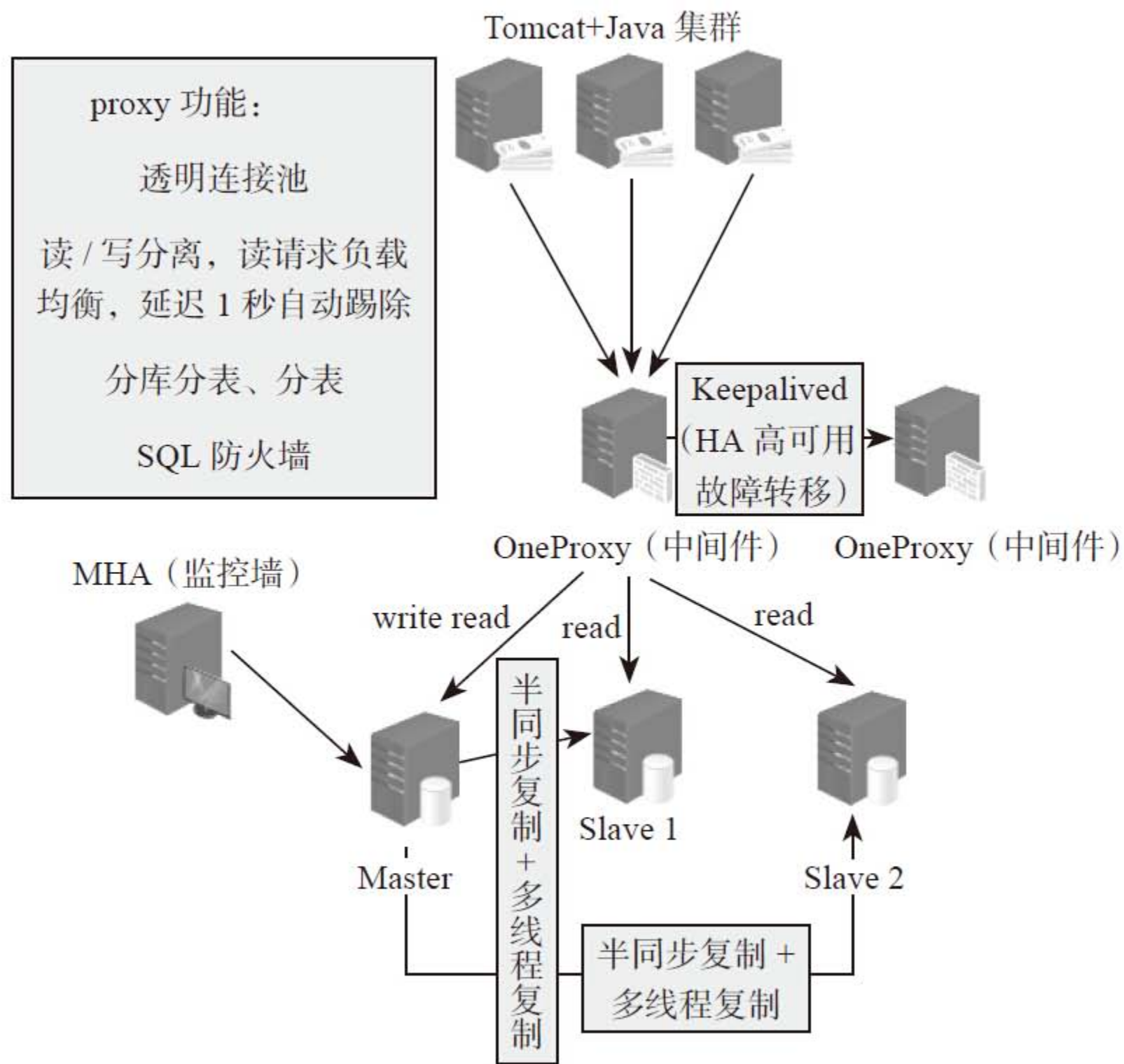


图8-20 OneProxy架构图

下面介绍一下安装过程。



1) 设置host解析。4台服务器的配置如下：

```
# cat /etc/hosts
192.168.17.128 master
192.168.17.129 slave1
192.168.17.130 slave2
192.168.17.131 oneproxy
```

2) 安装OneProxy ( 一个可执行文件无需安装 ) , 其官网下载地址为：<http://www.onexsoft.com/>。

安装命令如下：

```
# wget
http://www.onexsoft.cn/software/oneproxy-rhel6-linux64-v5.8-ga.tar.gz
# tar zxvf oneproxy-rhel6-linux64-v5.8-ga.tar.gz
```

3) 修改文件描述符65535，命令如下：

```
# vim /etc/security/limits.conf
* soft nofile 65535
* hard nofile 65535
# vim /etc/sysctl.conf
fs.file-max=655350
net.ipv4.ip_local_port_range = 1025 65000
net.ipv4.tcp_tw_reuse = 1
```

#修改完毕后，重启服务器生效。

4) 创建加密密码，命令如下：

```
# ./mysqlpwd 123456
9D7E55EAF8912CCBF32069443FAC452794F8941B
# 这里是对密码123456做加密
```

5) 配置OneProxy服务，启动脚本如下：

```
# cat start.sh
#!/bin/bash
export ONEPROXY_HOME=/root/oneproxy
${ONEPROXY_HOME}/oneproxy --keepalive --proxy-address=:3317 \
```



```
--proxy-master-addresses=192.168.17.128:3306@test |
--proxy-slave-addresses=192.168.17.129:3306@test |
--proxy-slave-addresses=192.168.17.130:3306@test |
--readmin-username=repl |
--readmin-password=066C934D687741280278A1F2409BA9FD6EF94541 |
--proxy-replication-check |
--proxy-group-slavedelay=test:5 |
--proxy-group-policy=test:read-slave |
--proxy-user-list=admin/9D7E55EAF8912CCBF32069443FAC452794F8941B@test |
--admin-address=:4041 |
--event-threads=4 --proxy-group-security=test:0 |
--log-file=${ONEPROXY_HOME}/oneproxy.log |
--pid-file=${ONEPROXY_HOME}/oneproxy.pid
```

参数说明如下。

- keepalive：如果OneProxy进程宕机，则自动重启。
- proxy-address：定义前端Java、PHP程序连接的端口，这里是3317。
- proxy-master-addresses：定义主库的IP地址为端口：组名（可以随便定义，主要是为了分库分表后的路由分组）。
- proxy-slave-addresses：定义从库的IP地址为端口：组名（需要和主库的组名一致）。
- readmin-username：同步复制的用户名，这里是repl。
- readmin-password：同步复制的密码，这里通过mysqlpwd工具加密。
- proxy-replication-check：开启同步复制延迟检测，这里的实现方式跟MaxScale一样，更新一张表，对比时间戳差值。如果不开启这个参数，默认以Seconds\_Behind\_Master为准。打开general\_log，可以看到如下信息。

```
151126 2:22:34 294 Query BEGIN
          294 Query REPLACE INTO test.oneproxy_replication_check
VALUES (1, UNIX_TIMESTAMP(NOW()))
          294 Query COMMIT /* implicit, from Xid_log_event */
240 Query SHOW MASTER STATUS
240 Query SHOW SLAVE STATUS
240 Query SELECT id, col2 FROM test.oneproxy_replication_check
where id = 1
```



- proxy-group-slavedelay：设置延迟多少秒不转发请求，这里为5秒，test是组名。
- proxy-group-policy：设置读/写分离的策略，总共支持8种策略，如图8-21所示。其中的4个重点策略如下。
- read\_slave：读在从库上，当从库宕机或者同步延迟时，强制请求走主库。
- read\_balance：读在主库和从库上。
- big\_slave：复杂SQL，如count(\*)、group by、join等查询访问从库。
- big\_balance：与read\_balance同理。
- proxy-user-list：定义前端Java、PHP程序连接的用户名、密码、数据库名。
- admin-address：定义OneProxy后台管理端口。
- proxy-group-security：可针对某一个Server Group来指定安全级别，默认值为0，即没有任何设置。设置为1，表示禁止通过OneProxy来执行DDL操作；设置为2，表示必须要有Where条件；设置为3，表示只允许只读的操作。
- event-threads：设置并发线程数，最大允许48个线程，不要超过CPU的核数。

```
MySQL [(none)]> LIST POLICY;
```

POLICY	MEMO
master_only	read and write on master database only
read_failover	read and write on master, read filover to slave if master fail
read_slave	write on master, read on slaves
read_balance	write on master, read on master and slaves
write_failover	write on one master, read on masters and slaves
write_balance	write on any masters, read on masters and slaves
big_slave	write and simple query on master, big query on slaves
big_balance	write and simple query on master, big query on masters and slaves

8 rows in set (0.00 sec)

图8-21 读/写分离策略

- 6) 启动服务，命令如下：  
`/bin/bash start.sh`
  - 7) 关闭服务。客户端连接OneProxy 4041后台管理端口，命令如下：  
`mysql -h192.168.17.131 -uadmin -pOneProxy -P4041 -e "shutdown force"`
- 安装完成，同样来看看前端PHP/Java程序的接入注意事项。

所有的后端MySQL节点，都需要有相同的用户名及口令，具体由proxy-user-list选项决定，若不是选项中指定的用户，则不能用来登录OneProxy，不能为每个不同的后端实例单独指定登录信息。

如果是Java应用程序，只需要在JDBC连接时指定自动重连选项即可，配置文件如下：

```
jdbc:mysql://localhost:3306/test characterEncoding=gbk&autoReconnect=true&failOverReadOnly=false
```

以下情况select查询将会在master上执行。

- 当事务里有SQL语句 ( begin;select;commit; ) 时。
- 增加了注释/\*master\*/时，如select/\*master\*/salary from money where name='张三';。
- 不支持slave的读取权重功能时。



### 8.5.3 OneProxy读/写分离接入限制

OneProxy读/写分离接入有如下限制。

·不支持预编译语句Prepared Statement，不支持Bind、Execute调用接口，分别如图8-22和图8-23所示。

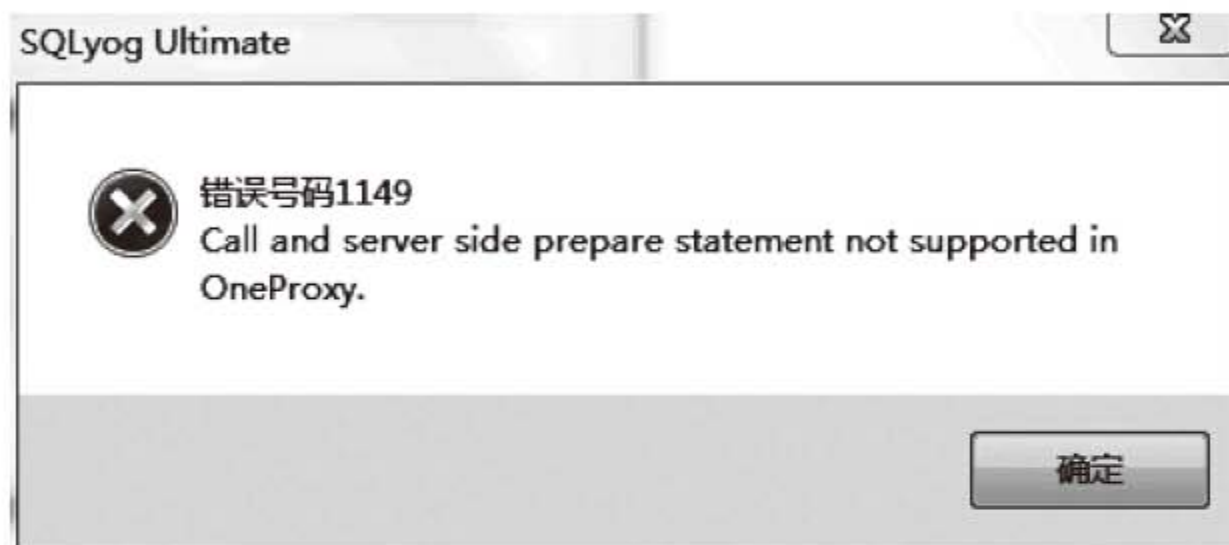


图8-22 Prepared Statement限制 ( 1 )

```
FATAL: mysql_stmt_prepare() failed
FATAL: MySQL error: 1149 "prepared statement not supported in OneProxy"
FATAL: thread#0: failed to prepare statements for test
```

图8-23 Prepared Statement限制 ( 2 )

·不支持使用use命令来切换后端数据库，默认连接的数据库由proxy-database（对所有用户指定）和proxy-user-list选项（可对不同用户指定不同的默认数据库）的值来决定。use命令可以执行，但其含义是切换到不同的MySQL主备集群，OneProxy在支持分库分表功能后，就将一个主备集群视为一个数据库了，连接OneProxy时如果指定了数据库名，则需要替换成Server Group的名字。

·禁止使用set命令，任何set命令都会直接返回成功，而不做任何处理，必须保持连接池里的每一个连接处于同样的会话配置状态。后端个别数据库参数的调整，需要直接访问后端数据库进行操作。

·默认禁止CALL、PREPARE、EXECUTE、DEALLOCATE命令，不支持存储过程和函数。

基于statement方式（SQL解析）的测试过程如下。

测试之前，在master（192.168.17.128）、slave1（192.168.17.129）、slave2（192.168.17.130）上均打开General Log，命令如下：



Set global general\_log=1;

#### (1) 读/写分离测试

让客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "select * from t1 where id =1;"
```

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "insert into t1 values(10);"
```

查看General Log，得知select语句会把请求转发给slave1或slave2，insert语句会把请求转发给master。

#### (2) 模拟slave延迟

在slave2 ( 192.168.17.130 ) 上执行flush tables with read lock全局读锁，通过客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "select * from t1 where id =1;"
```

查看General Log，得知复制延迟超过5秒的客户端请求将不会转发给slave2，而是会转发给没有延迟的slave1。当slave1和slave2都出现延迟超过5秒时，请求会强制转发给master。

#### (3) 模拟一台slave故障

在slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "select * from t1 where id =1;"
```

这时通过查看General Log，得知客户端请求将不会转发到slave2上。

#### (4) 模拟全部的slave故障

在slave1 ( 192.168.17.129 ) 和slave2 ( 192.168.17.130 ) 上执行stop slave io\_thread或stop slave sql\_thread，通过客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "select * from t1 where id =1;"
```

这时通过查看General Log，得知所有的请求将只转发给master ( 192.168.17.128 )，slave上并不会有请求转发，实现了平滑故障转移。



注意 OneProxy支持master进行故障转移切换，但建议采用流行的高可用方案MHA实现。故障切换后，OneProxy可以自动识别哪台机器是master。另外，架构必须是一主带N从，不能是双主带N从。

## 第9章

# Codership Galera Cluster 集群架构搭建与管理



Galera Cluster是由第三方公司Codership所研发的一套免费开源的集群高可用性方案，实现了数据零丢失，官网地址为<http://galeracluster.com/>。其在MySQL InnoDB存储引擎基础上打了wsrep（虚拟全同步复制）补丁，Percona/MariaDB已捆绑在各自的发行版里，目前只可在Linux系统下使用。

MySQL异步复制及semi-sync半同步复制都是基于MySQL binlog的，原生复制则是完全异步的。异步复制的工作机制是：master无需保证slave接收并执行了binlog，异步复制能够使master获取到最大性能，但是slave可能存在延迟，主从数据无法保证一致性，在不停止服务的前提下，如果master宕机后提升slave为新的主库，就会丢失数据。semi-sync在异步复制基础上增加了数据保护的步骤，这样，master必须确认slave收到binlog后（但不保证slave执行了事务）才会最终提交事务，若再结合MHA高可用架构，那么master挂掉之后，slave可以在应用完所有relay log后再切换成master提供的读/写服务。

相对于MySQL原生复制和semi-sync半同步复制，Galera Cluster全同步复制的差异如下。

- 同步复制，主备无延迟，一个节点宕机后，其他两个节点可以立即提供服务，而semi-sync需要应用（执行）完所有relay log，并依赖第三方高可用软件实现数据不丢失。
- 事务冲突检测可保证数据的一致性，多个节点可以同时读/写数据，可以极大简化数据访问。
- Galera Cluster有行级别并行复制，MySQL 5.7/MariaDB 10.0版本之前，slave SQL线程只有一个，这也是导致slave落后master的主要原因，饱受诟病。

## 9.1 Codership Galera Cluster的特性和优缺点

Codership Galera Cluster具有如下特性。

- 全同步复制，事务要么在所有节点都提交，要么都回滚。
- 多主复制，可以在任意节点进行写操作。
- 在从服务器上并行应用事件，是真正意义上的并行复制。
- 节点自动配置，故障节点自动从集群中移除。当故障节点再次加入集群时，无需手工备份当前数据库并拷贝至故障节点。
- 同步无延迟，可保证数据的一致性。
- 具有应用程序的兼容性，无需更改应用程序，具有原生的MySQL接口。
- 每个节点都包含完整的数据副本。
- 各个节点的同步复制，不是通过binlog实现的，而是通过galera.cache实现的。

Codership Galera Cluster也有一些需要注意的地方，比如，在生产环境下，建议集群配置3个节点，否则很容易产生脑裂。

Galera Cluster架构图如图9-1所示。



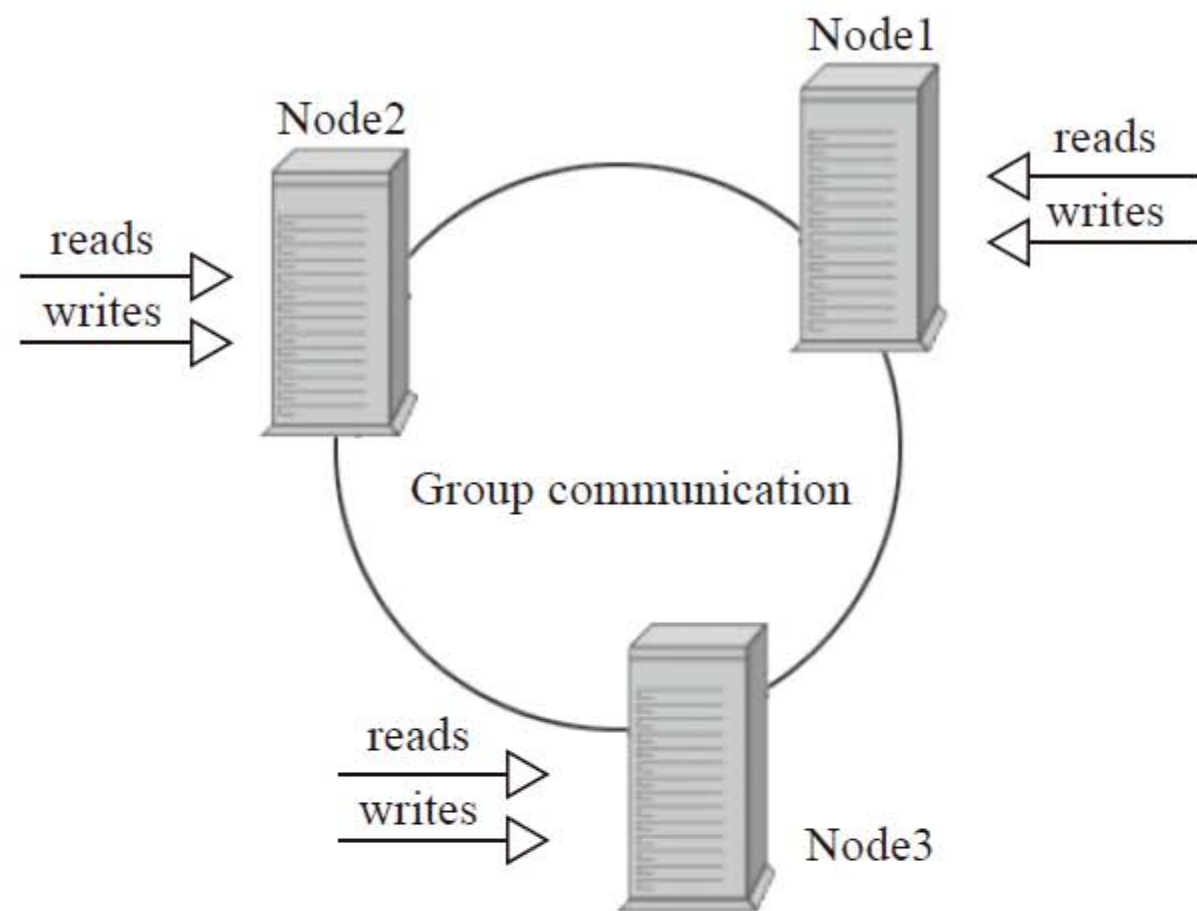


图9-1 Galera Cluster 集群多点读/写

该架构具有如下优点。

- 是真正的多主架构，任何节点都可以进行读/写操作，无需进行读/写分离。
- 无集中管理，可以在任何时间点失去任何一个节点，集群的正常工作不受影响。
- 节点宕机不会导致数据丢失。
- 对应用透明，无需更改应用或只需要进行极小的更改。

该架构的缺点也很明显，如下：

- 加入新节点时开销大，需要复制完整的数据。
- 不能有效地解决写扩展问题，所有的写操作都将发生在所有节点上。
- 有多少个节点就有多少重复的数据。
- 由于事务提交需要跨节点通信（分布式事务），因此写入会比主从复制慢很多，随着集群节点的增加，写入也会变得越来越慢，自然死锁和回滚也会更加频繁。



·对网络要求非常高，如果网络出现波动或机房被ARP攻击，则令造成两个节点失联，Galera Cluster集群会发生脑裂，服务将不可用。





## 9.2 Codership Galera Cluster的局限性

Codership Galera Cluster的局限性主要包含以下几方面。

- 目前的复制仅支持InnoDB存储引擎。任何写入其他引擎的表，包括mysql.\*表都不会被复制。但是DDL语句会被复制，因此创建用户将会被复制，但是insert into mysql.user...在user表插入数据时不会被复制。
- DELETE操作不支持没有主键的表。没有主键的表在不同节点上的顺序将不同，如果执行SELECT...LIMIT...，将出现不同的结果集。
- LOCK/UNLOCK TABLES/FLUSH TABLES{explicit table list}WITH READ LOCK不支持单表锁，以及锁函数GET\_LOCK()、RELEASE\_LOCK()，但FLUSH TABLES WITH READ LOCK支持全局表锁。
- General Query Log日志不能保存在表中。如果开启查询日志，则只能保存到文件中。
- 不能有大事务写入，不能超过wsrep\_max\_ws\_rows=131072（行），且写入集不能超过wsrep\_max\_ws\_size=1073741824（1G），否则客户端直接报错。
- 由于集群是乐观锁并发控制，因此，在commit阶段会有事务冲突发生。如果有两个事务在集群中的不同节点上对同一行写入并提交，则失败的节点将回滚，客户端返回死锁报错。
- XA分布式事务不支持Codership Galera Cluster，在提交时可能会回滚。
- 整个集群的写入吞吐量是由最弱的节点限制的，如果有一个节点变得缓慢，比如硬盘故障（RAID10坏了一块盘），那么整个集群将是缓慢的。为了达到稳定的高性能要求，所有的节点应使用统一的硬件。

### 9.3 Codership Galera Cluster的工作原理

Galera Cluster使用的是基于认证的复制，其流程分别如图9-2和图9-3所示。

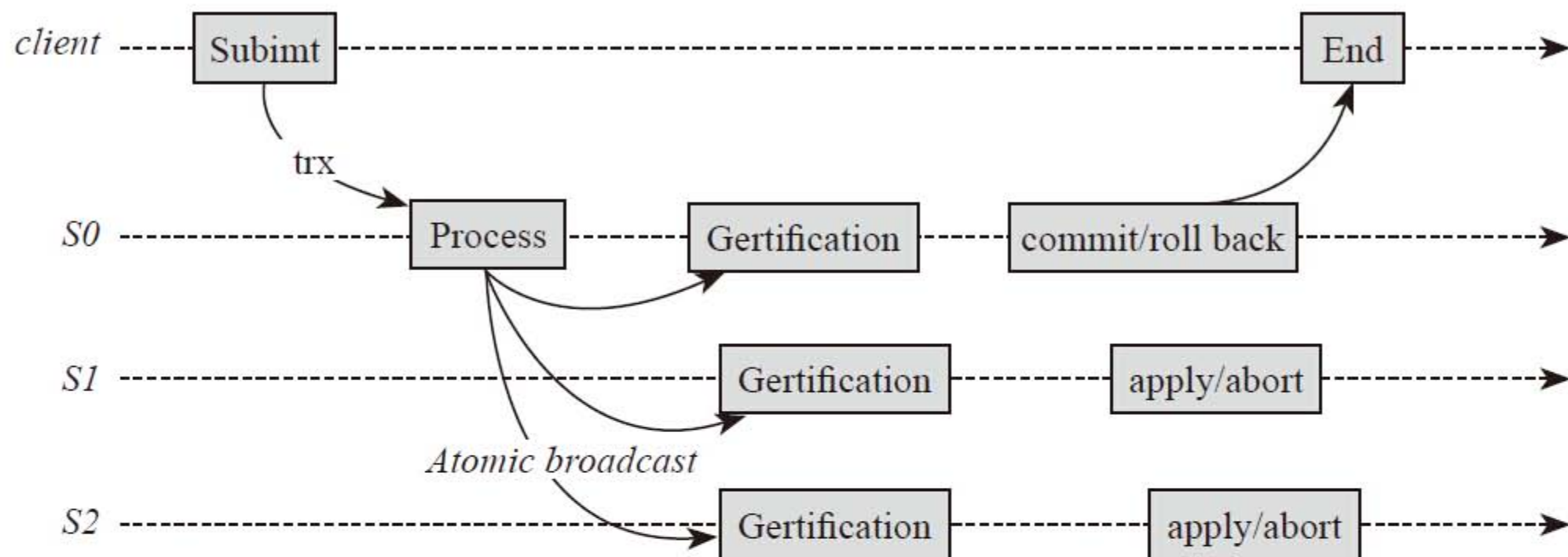


图9-2 认证复制工作流程

当客户端发起commit命令时（此时仍然没有发生真正的commit），事务内更改的行都会被搜集到一个写入集（writese）中，该写入集随后会被复制到其他节点，并且会在每个节点上使用搜索到的主键进行确认性认证测试，从而判断该写入集是否可以被应用。如果认证测试失败，写入集会被丢弃并且原始事务会被回滚；如果认证成功，事务会被提交并且写入集会在剩余节点上被应用。事务提交响应时间是由网络往返时间、远端节点认证时间决定的。



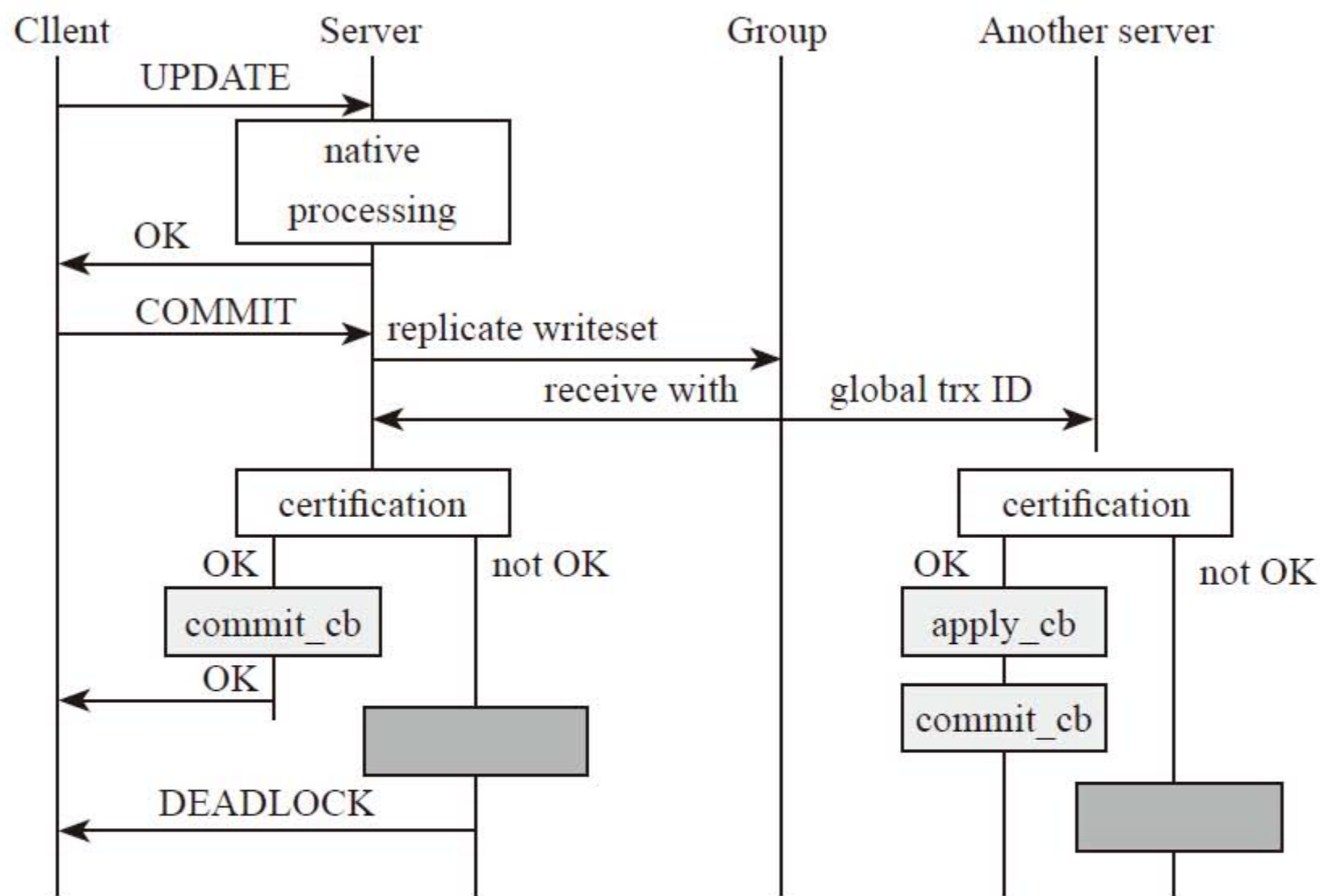


图9-3 认证复制工作原理

即使当前数据并没有真正地写入当前节点，只要其他节点验证成功了，就会返回成功的信号。因此这里的全同步复制，其实是虚拟的全同步复制，这段时间内，数据是有延迟的，但很小，如果应用程序访问的是远端节点，读到的数据是未改变之前的旧数据，如图9-4所示。这里可以通过设置参数`wsrep_causal_reads=ON`来避免，这种情况下需要等待远端节点应用完事务后，再返回客户端读取请求，这将增加读取的响应时间。而真正意义上的全同步复制，是要等所有节点事务都提交落地，才成功返回客户端。因此虚拟全同步复制的性能会更好一些。

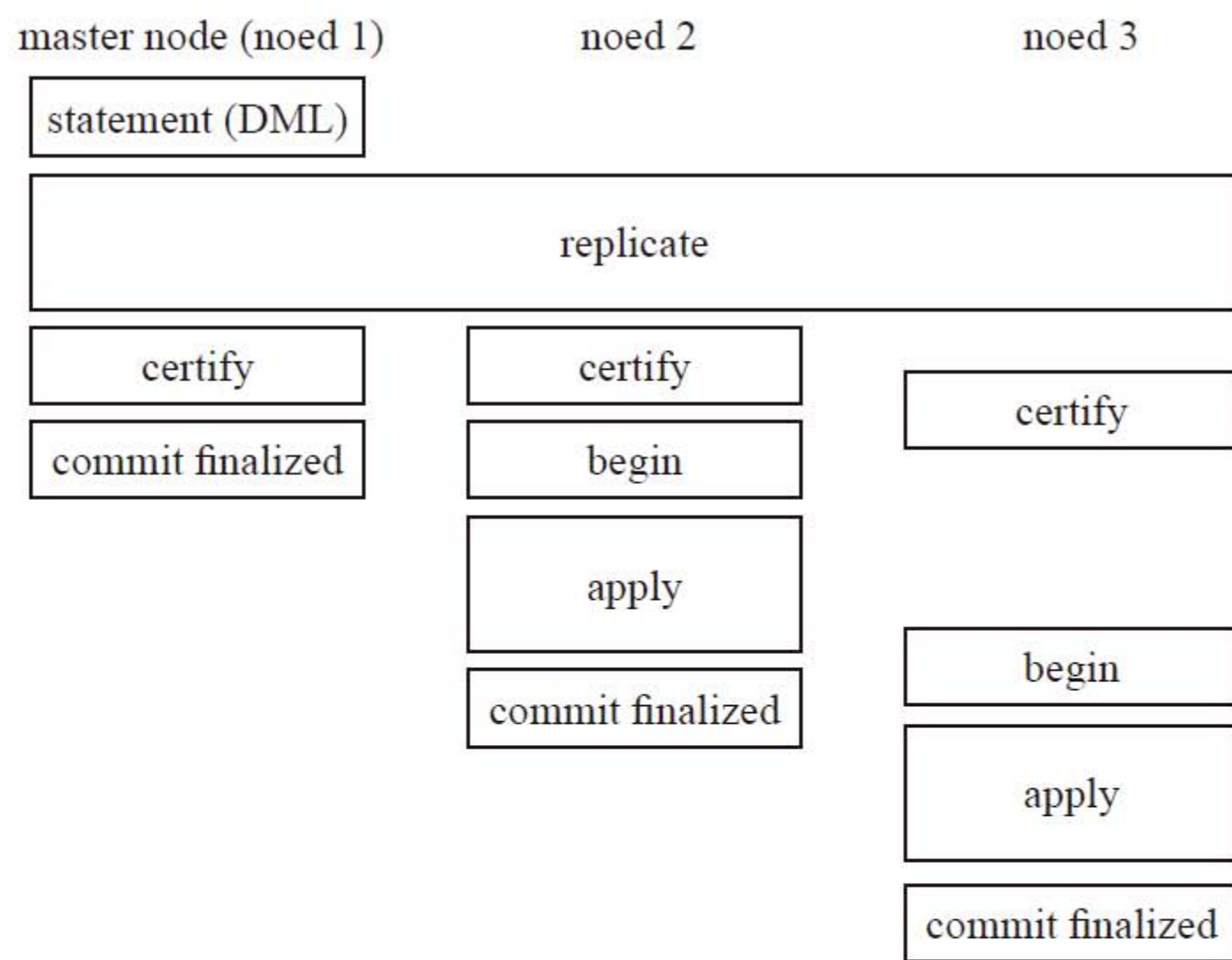


图9-4 认证复制工作原理

Galera Cluster内部实现了flow control限流措施，作用就是协调各个节点，保证所有节点执行事务的速度大于队列增长速度，从而避免丢失事务。实现原理很简单：在整个Galera Cluster集群中，同一时间只有一个节点可以广播消息（数据），每个节点都会获得广播消息的机会（获得机会后也可以不广播），当慢节点的待执行队列超过一定长度后，它会广播一个FC\_PAUSE消息，所以节点收到消息后都会暂缓广播消息并不提供写操作，直到该慢节点的待执行队列长度减小到一定长度后，Galera Cluster数据同步又开始恢复。

下面介绍Galera Cluster涉及的变量参数。

- wsrep\_provider\_options=gcs.fc\_limit：待执行队列长度超过该值时，flow control被触发，默认值是16。
- wsrep\_provider\_options=gcs.fc\_factor：待执行队列长度小于(gcs.fc\_factor\*gcs.fc\_limit)该值时，集群恢复同步复制，gcs.fc\_factor默认值是1。

下面是状态参数的说明。



- wsrep\_flow\_control\_paused：表示集群是否暂停复制，值大于0代表集群已被限流，将无法写入，但可以查询。
- wsrep\_flow\_control\_sent：表示该节点已经停止发送事务多少次。
- wsrep\_flow\_control\_recv：表示该节点已经停止接收事务多少次。
- wsrep\_local\_send\_queue\_avg：表示发送队列的平均长度，值大于0代表集群已被限流，将无法写入，但可以查询。
- wsrep\_local\_recv\_queue\_avg：表示接收队列的平均长度，值大于0代表节点压力过大，集群将进入限流措施，无法写入，但可以查询。

当查看error.log日志时，可以发现如下信息，表示已经暂停复制。

```
151221 17:55:45 [Note] WSREP: Provider paused at  
c9a11bfb-9cf5-11e5-a2f0-3f9309531dc3:3314 (4)
```

新节点加入流程的示意图如图9-5所示。

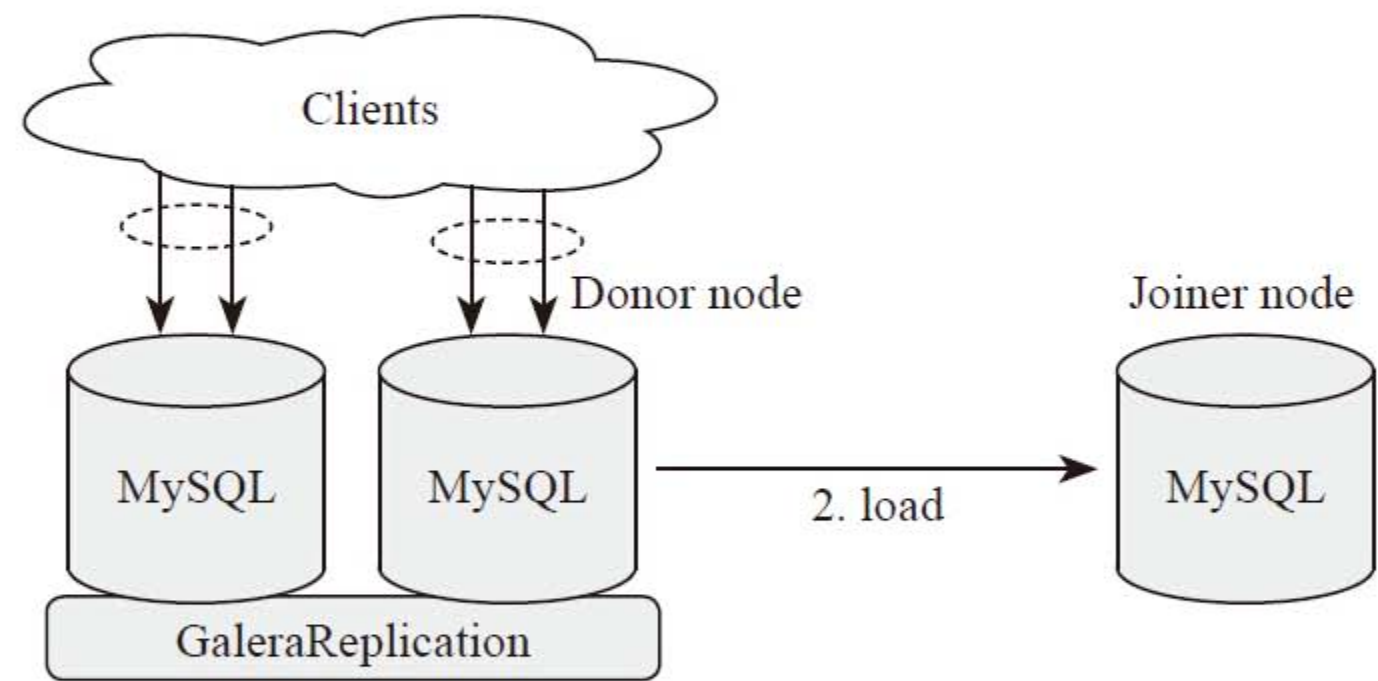


图9-5 新节点加入

在图9-5中，新加入的节点叫Joiner，给Joiner提供复制的节点叫Donor。在该过程中，首先会检查本地grastate.dat文件的seqno事务号是否在远端donor节点（集群中被选中作为数据源的节点）galera.cache文件里，如果存在，那么进行Incremental State Transfer（IST）增量同步复制，把剩余的事务发送过去；如果不存在，那么进行State Snapshot Transfer（SST）全量同步复制。SST有三种全量拷贝方式：mysqldump、rsync、xtrabackup，这三种方式的说明如表9-1所示。

表9-1 三种全量拷贝方式的对比



备份工具	mysqldump	rsync	xtrabackup
是否锁表	锁表	锁表	只备份 .frm 表结构和 MyISAM 引擎锁表
性能	速度慢	速度快	速度较快
可靠性	不推荐	不推荐	推荐

在XtraBackup 2.1.X版本里，使用innobackupex备份时，备份的流程如下。

- 1) 拷贝InnoDB表数据。
- 2) 执行全局表读锁FLUSH TABLES WITH READ LOCKS。
- 3) 拷贝.frm和MyISAM表数据。
- 4) 得到当前binlog文件名和Position点。
- 5) 完成redo log事务日志的后台复制。
- 6) 解锁UNLOCK TABLES。

从上面的流程可知，若备份的是一张或几张MyISAM的大表，就会造成锁表，在业务高峰期将严重影响业务。因此，在XtraBackup2.2.X版本里进行了改进。

在Percona 5.6.16版本里，首次引入了备份锁（MySQL/MariaDB没有备份锁）：

```
LOCK TABLES FOR BACKUP;
LOCK BINLOG FOR BACKUP;
```

备份锁只针对非InnoDB引擎，比如MyISAM、ARCHIVE、CSV引擎等，这可降低锁的粒度，备份的流程如下。

- 1) 拷贝InnoDB表数据。
- 2) 执行表备份锁。
- 3) 拷贝.frm和MyISAM表数据。
- 4) 执行binlog文件锁，确保日志不发生变化。
- 5) 完成redo log事务日志的后台复制。
- 6) 解表锁。
- 7) 得到当前binlog文件名和Position点。

8) 解binlog锁。



## 9.4 Codership Galera Cluster的配置

### 9.4.1 Codership Galera Cluster的配置环境及安装

三个节点IP如下。

node1 : 192.168.17.128

node2 : 192.168.17.129

node3 : 192.168.17.130



注意 在生产环境下，一定要配置三个节点，这样可以阻止脑裂问题，更多的节点不会对写入有很大的提升；相反，节点越多，写入性能越差！

Codership Galera Cluster的安装过程如下。

(1) 设置host解析

三台服务器的配置如下：

```
# cat /etc/hosts
```

```
192.168.17.128 node1
```

```
192.168.17.129 node2
```

```
192.168.17.130 node3
```

(2) 修改文件描述符65535

命令如下：

```
# vim /etc/security/limits.conf
```

```
* soft nfile 65535
```

```
* hard nfile 65535
```

```
# vim /etc/sysctl.conf
```



*fs.file-max=655350*  
*net.ipv4.ip\_local\_port\_range = 1025 65000*  
*net.ipv4.tcp\_tw\_reuse = 1*

修改完毕后，重启服务器生效。

### (3) 安装Percona XtraBackup热备份工具

下载地址为：[https://www.percona.com/downloads/XtraBackup/Percona-XtraBackup-2.3.2/binary/tarball/percona-xtrabackup-2.3.2-Linux-x86\\_64.tar.gz](https://www.percona.com/downloads/XtraBackup/Percona-XtraBackup-2.3.2/binary/tarball/percona-xtrabackup-2.3.2-Linux-x86_64.tar.gz)。

安装命令如下：

```
# tar zxvf percona-xtrabackup-2.3.2-Linux-x86_64.tar.gz  
# cd percona-xtrabackup-2.3.2-Linux-x86_64/bin/  
# cp -a * /usr/bin/
```

### (4) 安装依赖包

命令如下：

```
# yum install lsof -y  
# yum install socat -y  
# yum install tar -y  
# yum install openssl* -y
```

### (5) 创建XtraBackup备份时的用户名和密码

创建命令如下：

```
# GRANT ALL PRIVILEGES ON *.* TO 'admin'@'localhost'  
IDENTIFIED BY '123456';
```

### (6) 安装MariaDB Galera Cluster集群软件



注意 Percona/MariaDB把Galera Cluster集群套件都捆绑在各自的发行版里，也可以选择用官方Codership捆绑的MySQL Galera Cluster，它的安装配置参数是一样的，这里选择用MariaDB Galera Cluster做演示！

Codership的下载地址为：[http://releases.galeracluster.com/binary/mysql-wsrep-5.6.27-25.12-linux-x86\\_64.tar.gz](http://releases.galeracluster.com/binary/mysql-wsrep-5.6.27-25.12-linux-x86_64.tar.gz)

Percona的下载地址为：[https://www.percona.com/downloads/Percona-XtraDB-Cluster-56/Percona-XtraDB-Cluster-5.6.26-25.12/binary/tarball/Percona-XtraDB-Cluster-5.6.26-rel74.0-25.12.1.Linux.x86\\_64.tar.gz](https://www.percona.com/downloads/Percona-XtraDB-Cluster-56/Percona-XtraDB-Cluster-5.6.26-25.12/binary/tarball/Percona-XtraDB-Cluster-5.6.26-rel74.0-25.12.1.Linux.x86_64.tar.gz)

MariaDB的下载地址为：[http://mirrors.accretive-networks.net/mariadb//mariadb-galera-10.0.22/bintar-linux-x86\\_64/mariadb-galera-10.0.22-linux-x86\\_64.tar.gz](http://mirrors.accretive-networks.net/mariadb//mariadb-galera-10.0.22/bintar-linux-x86_64/mariadb-galera-10.0.22-linux-x86_64.tar.gz)

安装命令如下：

```
# useradd mysql
# tar zxvf mariadb-galera-10.0.22-linux-x86_64.tar.gz -C /usr/local/
# cd /usr/local/
# ln -s mariadb-galera-10.0.22-linux-x86_64 mysql
# chown -R mysql:mysql mysql/
# 修改环境变量
# vim /root/.bash_profile
PATH=/usr/local/mysql/bin:$PATH:$HOME/bin:/usr/local/bin
# source /root/.bash_profile
# which mysql
/usr/local/mysql/bin/mysql
```

(7) 配置参数

三个节点均按照如下方式设置。

```
# vim /etc/my.cnf
[mysqld]
server_id=128
# 三个节点的server_id要修改成不一样的
datadir=/data/galera
socket=/tmp/mysql.sock
user=mysql
skip-external-locking
skip-name-resolve
character-set-server = utf8
#####
```



```
##Galera Cluster
wsrep_provider = /usr/local/mysql/lib/libgalera_smm.so
# 设置galera类库
wsrep_cluster_address="gcomm://192.168.17.128 , 192.168.17.129 , 192.168.17.130"
# 设置集群IP
wsrep_provider_options = "gcache.size=4G"
# 设置gcache (同步复制缓冲池) 大小，默认是128MB，并根据你的内存定义。建议生产环境设置大一些，避免新加入集群采取State
Snapshot Transfer (SST) 全量同步复制，造成集群抖动。
wsrep_cluster_name = MariaDB Galera Cluster
# 设置集群的名字，这里随便定义
wsrep_sst_auth = admin:123456
# XtraBackup备份时的用户名和密码
wsrep_sst_method = xtrabackup-v2
# 默认是rsync全量拷贝，但需要在donor节点上执行全局读锁 ( flash tables with read lock )，这里采用XtraBackup热备份方式，只有在备
份.frm表结构文件才会锁表
wsrep_node_name = node1
# 设置集群节点的名字，在第二个节点上改为node2，在第三个节点上改为node3
wsrep_slave_threads = 24
# 开启并行复制线程，根据CPU核数设置
wsrep_on = ON
# 开启全同步复制模式
wsrep_causal_reads = ON
# 节点应用完事务才返回查询请求
wsrep_certify_nonPK= ON
# 为没有显示申明主键的表生成一个用于certification test的主键，默认为ON
#log-bin = /data/galera/mysql-bin
# log_slave_updates = 1
# 如果不需要后面接从库，则注释掉log-bin和log_slave_updates
binlog_format = ROW
# binlog设置为ROW行格式
```



```
innodb_flush_log_at_trx_commit = 0
```

# 因为Galera Cluster的节点恢复是通过远端节点，这里为了性能考虑，设置为0，即事务提交每隔1秒刷盘

```
innodb_autoinc_lock_mode=2
```

# 将主键自增模式修改为交叉模式

```
query_cache_size = 0
```

# 关闭查询缓存

### (8) 三个节点初始化安装

安装命令如下：

```
# cd /usr/local/mysql/
```

```
# scripts/mysql_install_db --defaults-file=/etc/my.cnf --user=mysql
```

### (9) 在node1节点上通过bootstrap引导启动

启动命令如下：

```
# /usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf
```

```
--user=mysql --wsrep-new-cluster &
```

注：第一次启动一定要加--wsrep-new-cluster参数，再次启动就不需要了。

### (10) 在node2和node3节点启动

启动命令如下：

```
# /usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf
```

```
--user=mysql &
```

至此安装结束，登录三个节点，执行show global status like 'ws%' \G;命令，如果界面如图9-6所示，则代表集群已经正常工作。

wsrep_local_state	4
wsrep_local_state_comment	Synced
wsrep_cert_index_size	0
wsrep_causal_reads	0
wsrep_cert_interval	0.000000
wsrep_incoming_addresses	192.168.17.128:3306, 192.168.17.129:3306, 192.168.17.130:3306
wsrep_evs_delayed	
wsrep_evs_evict_list	
wsrep_evs_repl_latency	0/0/0/0/0
wsrep_evs_state	OPERATIONAL
wsrep_gcomm_uuid	e9b0b897-aed1-11e5-96fc-bf319e89d493
wsrep_cluster_conf_id	15
wsrep_cluster_size	3
wsrep_cluster_state_uuid	03e94e33-b234-11e5-a7c4-7b1665b324a8
wsrep_cluster_status	Primary
wsrep_connected	ON
wsrep_local_bf_aborts	0
wsrep_local_index	2
wsrep_provider_name	Galera
wsrep_provider_vendor	Codership Oy <info@codership.com>
wsrep_provider_version	25.3.9(r3387)
wsrep_ready	ON
wsrep_thread_count	3

图9-6 集群状态

以下是图9-6中的参数解释。

- wsrep\_cluster\_size为3，代表集群有三个节点。
- wsrep\_cluster\_status为Primary，代表节点为主节点，可正常读/写。
- wsrep\_ready为ON，代表集群已正常运行。

## 9.4.2 功能测试

### 1. 集群基准测试

首先在node1节点上执行插入、删除和升级操作，如图9-7所示。

然后在node2/node3节点上查看（见图9-8和图9-9），可以看到数据都是一致的。

```
MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node1              |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> insert into t1 values(1,'a');
Query OK, 1 row affected (0.01 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
| 1  | a    |
+----+-----+
1 row in set (0.00 sec)
```

图9-7 node1节点写入数据



```

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node2              |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
| 1  | a    |
+----+-----+
1 row in set (0.07 sec)

```

图9-8 在node2节点上查看数据一致性

```

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node3              |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
|  1 | a    |
+----+-----+
1 row in set (0.08 sec)

```

图9-9 在node3节点上查看数据一致性

## 2. 模拟节点故障测试

先关闭node1节点，执行show global status like 'ws%' \G;命令，如图9-10所示。

此时集群会自动剔除故障节点node1（192.168.17.128），并且正常提供服务。

wsrep_incoming_addresses	192.168.17.129:3306, 192.168.17.130:3306
wsrep_evs_delayed	
wsrep_evs_evict_list	
wsrep_evs_repl_latency	0.000436132/0.000666624/0.000985251/0.000199124/4
wsrep_evs_state	OPERATIONAL
wsrep_gcomm_uuid	19c96696-ae53-11e5-a2cb-531b06418189
wsrep_cluster_conf_id	18
wsrep_cluster_size	2
wsrep_cluster_state_uuid	03e94e33-b234-11e5-a7c4-7b1665b324a8
wsrep_cluster_status	Primary
wsrep_connected	ON
wsrep_local_bf_aborts	0
wsrep_local_index	0
wsrep_provider_name	Galera
wsrep_provider_vendor	Codership Oy <info@codership.com>
wsrep_provider_version	25.3.9(r3387)
wsrep_ready	ON
wsrep_thread_count	3

57 rows in set (0.43 sec)

```
MariaDB [test]> select @@wsrep_node_name;
```

@@wsrep_node_name
node2

图9-10 集群节点目前是两个

### 3. 模拟脑裂后的处理

下面模拟在网络发生丢包抖动的情况下，两个节点失联导致脑裂。首先，在node1节点和node2节点上分别执行以下命令：

```
# iptables -A INPUT -p tcp --sport 4567 -j DROP
# iptables -A INPUT -p tcp --dport 4567 -j DROP
```

用于禁止wsrep全同步复制4567端口通信。



然后在node3节点上执行show global status like 'ws%' \G;命令，如图9-11所示。

wsrep_local_state_comment	Initialized
wsrep_cert_index_size	0
wsrep_causal_reads	0
wsrep_cert_interval	0.000000
wsrep_incoming_addresses	192.168.17.130:3306
wsrep_evs_delayed	2be2f613-ae68-11e5-a2e9-03f921a5b113:tcp://192.168.17.129:4567:2,afefc6de-b249-11e5-bbd7-17abc49e4199:tcp://192.168.17.128:4567:1
wsrep_evs_evict_list	
wsrep_evs_repl_latency	0/0/0/0/0
wsrep_evs_state	OPERATIONAL
wsrep_gcomm_uuid	2ad0c274-b053-11e5-acd3-c2430aa51805
wsrep_cluster_conf_id	18446744073709551615
wsrep_cluster_size	1
wsrep_cluster_state_uuid	03e94e33-b234-11e5-a7c4-7b1665b324a8
wsrep_cluster_status	non-Primary
wsrep_connected	ON
wsrep_local_bf_aborts	0
wsrep_local_index	0
wsrep_provider_name	Galera
wsrep_provider_vendor	Codership Oy <info@codership.com>
wsrep_provider_version	25.3.9(r3387)
wsrep_ready	OFF
wsrep_thread_count	3

57 rows in set (0.00 sec)

```
MariaDB [test]> select * from t1;
ERROR 1047 (08S01): WSREP has not yet prepared node for application use
MariaDB [test]> select @@wsrep_node_name;
ERROR 1047 (08S01): WSREP has not yet prepared node for application use
MariaDB [test]>
```

图9-11 脑裂

现在已经出现脑裂，并且集群无法执行任何操作，此时除了等待网络恢复就没别的方法了吗？答案是NO，可以通过设置：

*set global wsrep\_provider\_options="pc.bootstrap=true";*

来强制恢复出现脑裂的机器，如图9-12所示。

```

MariaDB [test]> select * from t1;
ERROR 1047 (08S01): WSREP has not yet prepared node for application use
MariaDB [test]> select @@wsrep_node_name;
ERROR 1047 (08S01): WSREP has not yet prepared node for application use
MariaDB [test]>
MariaDB [test]> set global wsrep_provider_options="pc.bootstrap=true";
Query OK, 0 rows affected (0.39 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
| 1  | a    |
+----+-----+
1 row in set (0.00 sec)

MariaDB [test]> show global status like 'wsrep_cluster_status';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_cluster_status | Primary |
+-----+-----+
1 row in set (0.00 sec)

MariaDB [test]> 

```

图9-12 强制恢复脑裂

从图9-13可以看到，集群已经恢复工作了。

#### 4.避免脏读

前面已经介绍，Galera Cluster不是真正意义上的全同步复制，因此存在延迟。我们在node3上节点上执行flush tables with read lock;全局读锁，然后在node1上insert插入，模拟延迟。

在node3节点上执行命令的截图如图9-13所示。

```
MariaDB [test]> flush tables with read lock;  
Query OK, 0 rows affected (0.00 sec)  
  
MariaDB [test]> select @@wsrep_node_name;  
+-----+  
| @@wsrep_node_name |  
+-----+  
| node3              |  
+-----+  
1 row in set (0.00 sec)
```

图9-13 在node3节点上执行全局读锁

在node1节点上上执行插入命令的截图如图9-14所示。

现在，再在node3节点上查看数据，如图9-15所示。



```

MariaDB [test]> insert into t1 values(2,'b');
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
|  1 | a    |
|  2 | b    |
+----+-----+
2 rows in set (0.00 sec)

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node1              |
+-----+
1 row in set (0.00 sec)

```

图9-14 在node1节点上插入数据1

```

-----
MariaDB [test]> flush tables with read lock;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node3              |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
| 1  | a    |
+----+-----+
1 row in set (0.00 sec)

```

图9-15 在node3节点上查看数据1

此时读取的就是脏数据。通过设置set global wsrep\_causal\_reads=1；来查看效果，如图9-16和图9-17所示。

```

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node1              |
+-----+
1 row in set (0.00 sec)

MariaDB [test]> insert into t1 values(3,'c');
Query OK, 1 row affected (0.01 sec)

MariaDB [test]> select * from t1;
+----+-----+
| id | name |
+----+-----+
|  1 | a    |
|  2 | b    |
|  3 | c    |
+----+-----+
3 rows in set (0.00 sec)

```

图9-16 在node1节点上查看数据

从图9-17可以看到，现在已经无法读取数据，从而避免了脏读。

## 5. 限流测试

首先，在node3节点上执行flush tables with read lock;全局读锁，然后在node1上使用sysbench打流量：

```

# sysbench --test=oltp --mysql-table-engine=innodb
--oltp-table-size=1000000 --max-requests=1000 --num-threads=4
--mysql-host=192.168.17.128 --mysql-port=3306 --mysql-user=admin --mysql-password=123456 --mysql-db=test
--db-driver=mysql
--mysql-engine-trx=yes run

```



```

MariaDB [test]> select @@wsrep_node_name;
+-----+
| @@wsrep_node_name |
+-----+
| node3              |
+-----+
1 row in set (0.01 sec)

MariaDB [test]> set global wsrep_causal_reads = 1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> set wsrep_causal_reads = 1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> flush tables with read lock;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from t1;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
MariaDB [test]> █

```

图9-17 在node3节点上查看数据2

再到node3节点上查看日志：

```

160101 22:14:36 [Note] WSREP: Provider paused at
03e94e33-b234-11e5-a7c4-7b1665b324a8:128 (192)

```

可以看到已经限流，集群停止。

下面在node1节点上再开启一个会话，执行插入操作，如图9-18所示。

```
MariaDB [test]> select @@wsrep_node_name;
```

```
+-----+
| @@wsrep_node_name |
+-----+
| node1              |
+-----+
```

```
1 row in set (0.00 sec)
```

```
MariaDB [test]> show processlist;
```

Id	User	Host	db	Command	Time	State	Info	Progress
1	system user		NULL	Sleep	427218	committed 25	NULL	0.000
2	system user		NULL	Sleep	25296	wsrep aborter idle	NULL	0.000
4	system user		NULL	Sleep	NULL	NULL	NULL	0.000
15	admin	192.168.17.128:64961	test	Execute	81	init	COMMIT	0.000
16	admin	192.168.17.128:64962	test	Execute	81	init	COMMIT	0.000
17	admin	192.168.17.128:64963	test	Execute	81	init	COMMIT	0.000
18	admin	192.168.17.128:64964	test	Execute	81	init	COMMIT	0.000
19	root	localhost	test	Query	14	query end	insert into t1 values(5, 'e')	0.000
20	root	localhost	test	Query	0	init	show processlist	0.000

```
9 rows in set (0.17 sec)
```

```
MariaDB [test]> █
```

图9-18 在node1节点上插入数据2

从图8-18可以看到，集群此时无法写入，解锁unlock tables，待数据追上时，限流关闭，集群正常运行。

综上所述，在生产环境下应避免使用大事务，不建议在高并发写入场景下使用Galera Cluster架构（如电商秒杀场景），会导致集群限流，从而引起整个集群hang住，出现重大故障。对数据一致性要求较高，并且数据写入不频繁、数据库容量也不大（50GB左右）、网络状况良好（极少出现丢包，建议机房交换机做双机热备，数据库和应用服务器已经通过VLAN做了网络隔离）的，Galera Cluster集群架构会是一个好的选择。

## 9.5 HAProxy结合Galera Cluster实现无单点秒级故障切换

前面已经介绍了Galera Cluster的安装和测试，那么Java/PHP应用如何访问呢？这里借助第三方代理软件HAProxy进行讲解，这也是Percona官方推荐的架构，它们可实现无单点秒级故障切换。在前端应用如Java里的jdbc配置文件中，把IP改为HAProxy，通过HAProxy把读/写请求轮训转发到后端Galera Cluster的三个节点上，当有一个节点宕掉时，通过HAProxy检测，并让该节点进行下线处理，因此不会影响业务，保证业务7×24小时不间断访问，架构如图9-19所示。



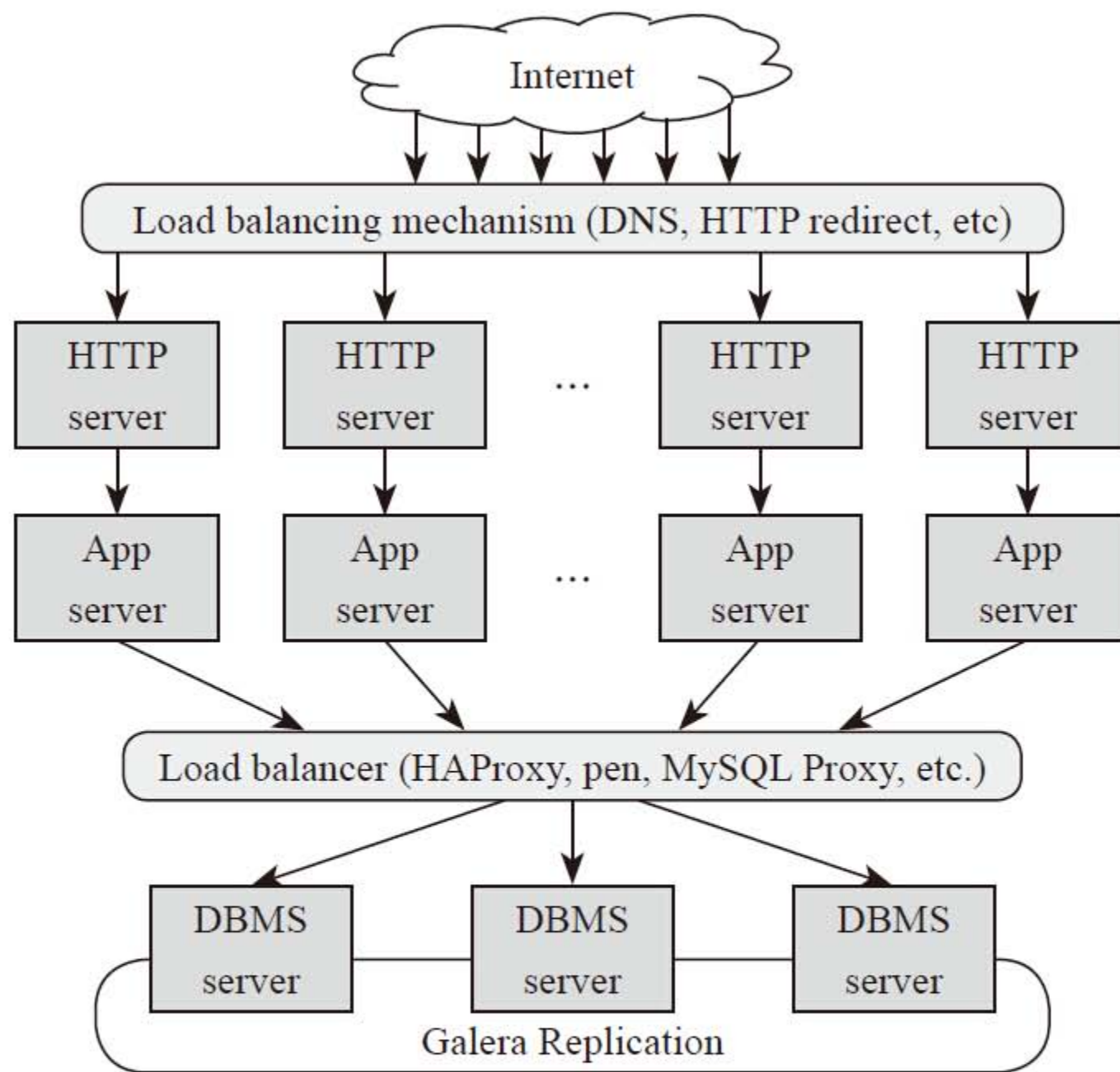


图9-19 HAProxy+Galera Cluster架构图

下面介绍HAProxy结合Galera Cluster实现无单点秒级故障切换时的安装和配置过程。

#### (1) 安装HAProxy

安装命令如下：

```
# yum install haproxy -y
```

## (2) 安装rsyslog

安装命令如下：

```
# yum install rsyslog -y  
# vim /etc/rsyslog.conf
```

然后定义HAProxy的输出log，在最后一行加入：

```
local2.* /var/log/HAProxy.log  
$ModLoad imudp  
$UDPServerRun 514  
# /etc/init.d/rsyslog restart
```

重启系统日志服务。

## (3) 修改文件描述符65535

命令如下：

```
# vim /etc/security/limits.conf  
* soft nofile 65535  
* hard nofile 65535  
# vim /etc/sysctl.conf  
fs.file-max=655350  
net.ipv4.ip_local_port_range = 1025 65000  
net.ipv4.tcp_tw_reuse = 1
```

修改完毕后，重启服务器生效。

## (4) 配置HAProxy服务

HAProxy配置中包括以下五大部分。

- global：全局配置参数，进程级的，用来控制Haproxy启动前的一些进程及系统设置。
- defaults：配置一些默认的参数，可以被frontend、backend、listen段继承使用。
- frontend：用来匹配接收客户所请求的域名、uri等，并针对不同的匹配进行不同的请求处理。
- backend：定义后端服务器集群，以及对后端服务器的一些权重、队列、连接数等选项的设置，笔者将其理解为Nginx中的upstream块。
- listen：笔者理解为frontend和backend的组合体。

配置HAProxy服务的命令如下：

```
# cat /etc/HAProxy/HAProxy.cfg
```



```

global
# 全局参数的设置
log 127.0.0.1 local2
# 全局的日志配置，使用log关键字，指定使用127.0.0.1上的syslog服务中的local2日志设备，记录日志等级为warning的日志
maxconn 65535
# 定义haproxy进程的最大连接数
user haproxy
group haproxy
# 设置运行HAProxy的用户和组
daemon
# 以守护进程的方式运行
nbproc 24
# 设置HAProxy启动时的进程数，该值的设置应该和服务器的CPU核心数一致，即常见的2颗12核CPU的服务器，共有24核，则可以将其值设置为<=24，创建多个进程数，可以减少每个进程的任务队列，但是过多的进程数也可能导致进程的崩溃。

defaults
# 配置默认的参数
log global
# 继承global中log的定义
option tcplog
# 启用日志记录TCP请求
option dontlognull
# 启用该项，日志中将不会记录空连接。所谓空连接，就是上游的负载均衡器或者监控系统为了探测该服务是否存活可用，需要定期连接或者获取某一固定的组件或页面，或者探测扫描端口是否在监听或开放等动作。官方文档中标注，如果该服务上游没有其他的负载均衡器，建议不要使用该参数，因为互联网上的恶意扫描或其他动作就不会被记录下来

retries 3
# 定义连接后端服务器的失败重连次数，连接失败次数超过此值后，将对应后端服务器标记为不可用
option redispatch
# 当使用cookie时，haproxy会将其请求的后端服务器的serverID插入cookie中，以保证会话的SESSION持久性。此时，如果后端的服务器宕掉，那么客户端的cookie是不会刷新的；如果设置此参数，则会将客户的请求强制定向到另外一个后端server上，以保证服务的正常。
option abortonclose

```



```
# 当服务器负载很高的时候，自动结束掉当前队列处理比较久的链接
timeout connect 10s
# 设置成功连接到一台服务器的最长等待时间
timeout client 2m
# 设置客户端发送数据时的最长等待时间
timeout server 2m
# 设置服务器端回应客户端发送数据时的最长等待时间
timeout queue 1m
# 设置一个请求在队列里的超时时间
frontend mysqlcluster-front
# 定义一个名为mysqlcluster-front的前端部分
bind *:3320
# 应用端php/java连接haproxy的端口号
mode tcp
# tcp是四层模式
default_backend mysqlcluster-back
# 定义一个名为mysqlcluster-back的后端部分
frontend stats-front
# 定义一个名为stats-front的前端部分
bind *:80
# haproxy监控页面的端口号
mode http
# http是七层模式
default_backend stats-back
# 定义一个名为stats-back的后端部分
backend mysqlcluster-back
mode tcp
balance roundrobin
# 轮询模式
option httpchk
```



# 开启对后端服务器的健康检测

```
server 192.168.17.128 192.168.17.128:3306 check port 13306 inter 2000 rise 3 fall 3 weight 10
```

```
server 192.168.17.129 192.168.17.129:3306 check port 13306 inter 2000 rise 3 fall 3 weight 10
```

# port 13306端口是调用后端DB的服务，用来感知Galera Cluster节点wsrep\_cluster\_status状态值是否为Primary

# weight 10代表权重，值越小，转发的请求就越少

```
server 192.168.17.130 192.168.17.130:3306 check port 13306 inter 2000 rise 3 fall 3 weight 10
```

```
backend stats-back
```

```
mode http
```

```
stats uri /HAProxy/stats
```

# 定义监控页面URL地址

```
stats auth admin:123456
```

# 定义页面访问的用户名和密码

```
stats refresh 3s
```

# 定义每3秒自动刷新页面

(5) 启动HAProxy服务

启动命令如下：

```
# /etc/init.d/haproxy start
```

(6) 访问监控页面URL

打开浏览器，输入URL：<http://192.168.17.131/haproxy/stats>，第一次会让你输入用户名和密码，如图9-20所示。



图9-20 HAproxy登录页面

这里输入刚才定义好的用户名为admin，密码为123456。

图9-21是进入后的监控页面，这里可以看到从库192.168.17.129/130是DOWN状态，这是因为HAProxy在调用后端slave的9200端口服务，我们还没有在该机器里配置，所以这里显示DOWN状态。





图9-21 HAProxy监控状态

### (7) 配置Galera Cluster的复制，检测13306端口服务

命令如下：

# yum install xinetd -y

### (8) 安装xinetd服务

命令如下：

# vim /etc/services

增加13306服务，在最后一行加入：

Galera\_Cluster\_check 13306/tcp # Galera\_Cluster\_check

然后修改Galera\_Cluster\_check文件，命令如下：

```
# vim /etc/xinetd.d/Galera_Cluster_check
# default: on
# description: Galera_Cluster_check
service Galera_Cluster_check
{
    # this is a config for xinetd , place it in /etc/xinetd.d/
    disable = no
    flags = REUSE
    socket_type = stream
    port = 13306
    wait = no
    user = nobody
    server = /usr/bin/galera_check
    log_on_failure += USERID
    only_from = 0.0.0.0/0
    # recommended to put the IPs that need
    # to connect exclusively (security purposes)
    per_source = UNLIMITED
}
```

再创建Galera Cluster的复制检测脚本，这个脚本就是HAProxy调用的，命令如下：

```
# vim /usr/bin/galera_check
#!/bin/bash
BASEDIR=/usr/local/mysql/bin
MYSQL_USERNAME=root
MYSQL_PASSWORD=123456
WSSREP_STATUS=`$BASEDIR/mysql --host=localhost --user=$MYSQL_USERNAME
--password=$MYSQL_PASSWORD -e "show status like 'wsrep_local_state';" | awk '{if
(NR!=1){print $2}}' 2>/dev/null`
if [ "$WSSREP_STATUS" == "4" ]
then
```





```

# mysql is fine , return http 200
/bin/echo -e "HTTP/1.1 200 OK \n"
/bin/echo -e "Content-Type: Content-Type: text/plain \n"
/bin/echo -e "\r\n"
/bin/echo -e "MySQL is running. \n"
/bin/echo -e "\r\n"
else
# mysql is fine , return http 503
/bin/echo -e "HTTP/1.1 503 Service Unavailable \n"
/bin/echo -e "Content-Type: Content-Type: text/plain \n"
/bin/echo -e "\r\n"
/bin/echo -e "MySQL is *down*. \n"
/bin/echo -e "\r\n"
fi
```

这里定义了wsrep\_local\_state是否等于4，如果等于，则返回OK状态；如果不等于4，则认定该node节点宕机，不会把请求转发给它。图9-22列举了四种状态。

Num	Comment	Description
1	Joining	Node is joining the cluster
2	Donor/Desynced	Node is the donor to the node joining the cluster
3	Joined	Node has joined the cluster
4	Synced	Node is synced with the cluster

图9-22 wsrep\_local\_state状态

下面赋予/usr/bin/galera\_check可执行权限，命令如下：



`# chmod 755 /usr/bin/galera_check`

执行下面的脚本，如看到如下信息代表节点运行正常。

`# /usr/bin/galera_check`

`HTTP/1.1 200 OK`

`Content-Type: Content-Type: text/plain`

`MySQL is running.`

`# /etc/init.d/xinetd start`

可以看到端口已经启动了，如图9-23所示。

```
[root@master ~]# netstat -ntlp | grep 13306
tcp        0      0 :::13306                :::*                    LISTEN      1122/xinetd
[root@master ~]#
```

图9-23 查看端口

通过监控页面可以看到全部node节点均正常，如图9-24所示。

# HAProxy version 1.5.4, released 2014/09/02

## Statistics Report for pid 1583

### > General process information

pid = 1583 (process #1, nbproc = 1)  
 uptime = 0d 0h00m27s  
 system limits: memmax = unlimited; ulimit-n = 131085  
 maxsock = 131085, maxconn = 65535, maxpipes = 0  
 current conns = 1, current pipes = 0/0; conn rate = 0/sec  
 Running tasks: 1/9; idle = 100 %

active UP      backup UP  
 active UP, going down      backup UP, going down  
 active DOWN, going up      backup DOWN, going up  
 active or backup DOWN      not checked  
 active or backup DOWN for maintenance (MAINT)  
 active or backup SOFT STOPPED for maintenance  
 Note: "NO LB"/"DRAIN" = UP with load-balancing disabled.

Display option:   
 External resources:  
 • [Primary site](#)  
 • [Updates \(v1.5\)](#)  
 • [Online manual](#)  
 • Scope:   
 • [Hide DOWN servers](#)  
 • [Disable refresh](#)  
 • [Refresh now](#)  
 • [CSV export](#)

mysqlcluster-front																															
	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrftle	
Frontend				0	0	-	0	0	65 535	0			0	0	0	0	0					OPEN									

stats-front																															
	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrftle	
Frontend				0	2	-	1	1	65 535	4			15 318	389 247	0	0	0					OPEN									

mysqlcluster-back																															
	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrftle	
192.168.17.128	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	27s UP	L7OK/200 in 36ms	10	Y	-	0	0	0s	-	
192.168.17.129	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	27s UP	L7OK/200 in 16ms	10	Y	-	0	0	0s	-	
192.168.17.130	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	27s UP	L7OK/200 in 20ms	10	Y	-	0	0	0s	-	
Backend	0	0		0	0		0	0	6 554	0	0	?	0	0	0	0		0	0	0	0	27s UP		30	3	0		0	0s		

stats-back																															
	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrftle	
Backend	0	0		1	3		1	1	6 554	24	0	0s	18 310	389 247	0	0		2	0	0	0	27s UP		0	0	0		0			

图9-24 HAProxy监控图

通过图9-25所示的基准测试可以看到，请求已经轮询分发。

```
[root@MHA ~]# mysql -h192.168.17.131 -uadmin -p123456 -P3320 -e "show variables like 'wsrep_node_name';"
Warning: Using a password on the command line interface can be insecure.
```

Variable_name	Value
wsrep_node_name	node1

```
[root@MHA ~]#
[root@MHA ~]# mysql -h192.168.17.131 -uadmin -p123456 -P3320 -e "show variables like 'wsrep_node_name';"
Warning: Using a password on the command line interface can be insecure.
```

Variable_name	Value
wsrep_node_name	node2

```
[root@MHA ~]#
[root@MHA ~]# mysql -h192.168.17.131 -uadmin -p123456 -P3320 -e "show variables like 'wsrep_node_name';"
Warning: Using a password on the command line interface can be insecure.
```

Variable_name	Value
wsrep_node_name	node3

图9-25 轮询转发请求





随着网站的壮大，MySQL数据库架构一般会经历如图10-1所示的演进过程。

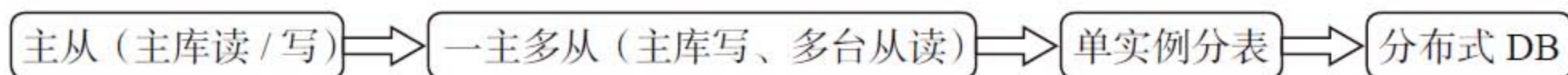


图10-1 MySQL数据库架构的演进

当数据很小时，只用一台机器也许就能扛住访问压力；当数据量变大时，最初可以通过增加硬件（比如，加内存、换SSD硬盘，或者采购性能很强劲的小型机）的方法去解决。如果数据量越来越大，则最好从架构层出发进行改进，可以采用读/写分离的方式，同时采用多台slave备机提供读取业务，这样就降低了数据库的负载。

随着业务的进一步发展，一台主库的数据写入将成为瓶颈，如电商秒杀场景。依靠表的user\_id取模，把数据平均分散到不同的小表，再分布到各台机器上的方式，可以看成是数据迁移。

为什么要分库分表？原因如下：

- 单个库的数据容量太大，单个DB存储空间不够。
- 单个库表太多，查询时，打开表操作也消耗系统资源。
- 单个表容量太大，查询时，扫描行数过多，磁盘I/O大，查询缓慢。
- 单个库能承载的访问量有限，可高的访问量只能通过分库分表实现。

当一个表太大不利于维护时，可考虑将大表拆分成小表，当然，这些表是属于同一个数据库的，这种技术称为分表；当一个数据库的处理能力不够支撑业务，增加CPU的作用也十分有限时，就可能需要将部分表移到别的数据库，以增加系统处理能力，这种技术称为分库；通过精心的数据模型设计，将大的业务表拆分成小表，再将一系列小表分到不同的服务器，使得每台服务器都能独立处理部分业务，这种技术称为水平拆分，俗称分库分表。分表的数量可以和物理的机器数不一致，分表数量称为逻辑份数，分库的数量称为物理份数，当逻辑份数大于物理份数时，就可以迅速获得水平扩展能力。

当使用传统商业数据库时，必须通过应用层修改代码来实现，现在流行的做法是将应用开发语言统一（比如用统一的Java框架）起来，然后编写统一的数据访问层（比如TDDL、ZDAL等）。这种做法在大并发量下的性能上有一定的优势，可以减少一次网络交互，但在开发上绑定了特定的开发语言，需要有强大的配置推送体系，并且需要有强大的运维团队来支持。当后台使用的是MySQL数据库，或兼容MySQL协议的数据库时，就可以不用修改应用程序，使用OneProxy来实现TDDL、ZDAL的功能，将后端的多台MySQL虚拟成一台MySQL提供给上层应用，对应用相对透明地实现分库分表的需求，从而快速获得MySQL上的水平扩展能力。

OneProxy可以轻松实现MySQL的横向扩展，突破单台MySQL的限制，将数据库的同库分区表（范围分区、列表分区、哈希分区）扩展到跨库的分区表（将不同的分区放到不同的MySQL主从集群上），通过对单库事务及绝大部分跨库查询的透明支持，实现对应用相对非常透明的分布



式数据库的架构支持，使得应用在进行少量修改的情况下就可以切换到分库分表的分布式互联网架构上，OneProxy内置的故障检测机制，可以让应用的开发变得非常简单。

对程序来讲，就是访问一张User表。后面怎么划分，按照什么规则划分，都不用管。例如，客户端插入user\_id：1-10十条数据，会通过OneProxy内置的SQL解释器分析SQL，按照range规则或hash规则，将客户端的请求自动分发到后端DB上，智能实现透明分库分表，每台DB上单独保存着分片后的数据。

分表后的示意图如图10-2所示。

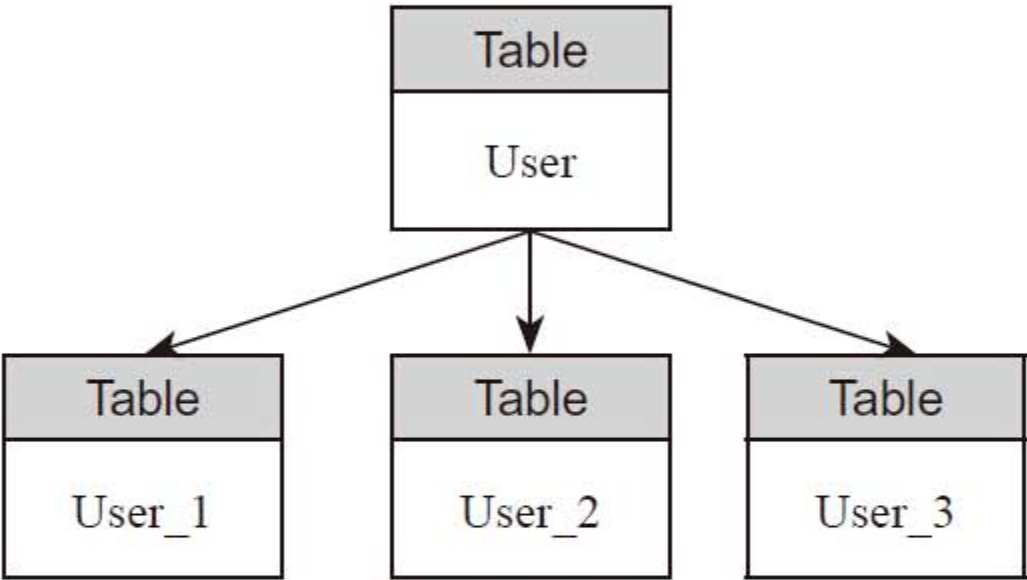


图10-2 拆分User表



## 10.1 OneProxy分库分表的搭建

### 10.1.1 配置与安装

OneProxy分库分表的配置环境如下。

OneProxy : 192.168.17.131

Master1 : 192.168.17.128

Master2 : 192.168.17.129

下面介绍安装过程。

(1) 设置host解析

3台服务器的配置命令如下：

```
# cat /etc/hosts
```

```
192.168.17.128 master1
```

```
192.168.17.129 master2
```

```
192.168.17.131 oneproxy
```

(2) 安装OneProxy (无需安装可执行文件)

首先在官网下载安装文件，下载地址为<http://www.onexsoft.com/>。安装命令如下：

```
# wget
```

```
http://www.onexsoft.cn/software/oneproxy-rhel6-linux64-v5.8-ga.tar.gz
```

```
# tar zxvf oneproxy-rhel6-linux64-v5.8-ga.tar.gz
```

(3) 修改文件描述符65535

修改文件描述符65535命令如下：

```
# vim /etc/security/limits.conf
```

```
* soft nofile 65535
```

```
* hard nofile 65535
```



```
# vim /etc/sysctl.conf
fs.file-max=655350
net.ipv4.ip_local_port_range = 1025 65000
net.ipv4.tcp_tw_reuse = 1
```

修改完毕后，通过reboot命令重启服务器生效。

#### (4) 创建加密密码

创建加密密码的命令如下：

```
# ./mysqlpwd 123456
9D7E55EAF8912CCBF32069443FAC452794F8941B
```

这里是对密码123456进行加密。

#### (5) 配置OneProxy服务

配置OneProxy服务的命令如下：

```
# cat sharding_start.sh
#!/bin/bash
export ONEPROXY_HOME=/root/oneproxy
${ONEPROXY_HOME}/oneproxy --keepalive --proxy-address=:3317 \
    --proxy-master-addresses=192.168.17.128:3306@group1 \
    --proxy-master-addresses=192.168.17.129:3306@group2 \
    --proxy-user-list=admin/9D7E55EAF8912CCBF32069443FAC452794F8941B@test \
    --admin-address=:4041 \
    --proxy-part-tables=${ONEPROXY_HOME}/part.txt \
    --proxy-parallel-degree=8 \
    --event-threads=8 \
    --proxy-group-policy=group1:0 --proxy-group-policy=group2:0 \
    --proxy-group-security=group1:0 --proxy-group-security=group2:0 \
    --log-file=${ONEPROXY_HOME}/oneproxy.log \
    --pid-file=${ONEPROXY_HOME}/oneproxy.pid
```

其中，参数解释如下。

---keepalive：表示在oneproxy进程宕掉，则自动重启。



- proxy-address：表示定义前端Java、PHP程序连接的端口，这里是3317。
- proxy-master-addresses：表示定义主库的IP地址：端口：组名（可以随便定义，主要是为了分库分表后的路由分组）。
- proxy-group-policy：表示设置读/写分离的策略，总共支持8种策略，如图10-3所示。其中4种重点策略如下。

```
MySQL [(none)]> LIST POLICY;
```

POLICY	MEMO
master_only	read and write on master database only
read_failover	read and write on master, read filover to slave if master fail
read_slave	write on master, read on slaves
read_balance	write on master, read on master and slaves
write_failover	write on one master, read on masters and slaves
write_balance	write on any masters, read on masters and slaves
big_slave	write and simple query on master, big query on slaves
big_balance	write and simple query on master, big query on masters and slaves

8 rows in set (0.00 sec)

图10-3 读/写分离策略

- read\_slave：读在从库上，当从库发生宕机或者同步延迟，强制请求走主库。
- read\_balance：读在主库和从库上。
- big\_slave：复杂的SQL如count(\*)、group by、join等查询访问从库。
- big\_balance：功能与read\_balance类似。
- proxy-user-list：定义前端Java、PHP程序连接的用户名、密码和数据库名。
- admin-address：定义OneProxy后台管理端口。
- proxy-group-security：可针对某一个Server Group来指定安全级别，默认值为0，即没有任何设置。设置为1，表示禁止通过OneProxy来执行DDL操作；设置为2，表示必须要有Where条件；设置为3则只允许只读的操作。
- event-threads：设置并发线程数，最大允许48个线程，不要超过CPU的核数。
- proxy-part-tables：设置分库分表规则。OneProxy分库分表原理为首先检查客户端执行的SQL，判断分区键值HASH取模后的结果；然后在part.txt规则文件里取出插入的表名和组名，并将SQL路由到指定的后端DB中。
- proxy-parallel-degree：当where条件带分区列时，直接命中该表，如果未带分区列，则会逐一扫描所有分表（单线程），考虑性能问题，增加并行查询（多线程，这里设置为8个线程），比如将SQL改为select/\*parallel\*/from t1 where name='李四';并行查询会增加额外的CPU消耗。



## (6) 设置分库分表配置文件

设置分库分表配置文件的命令如下：

```
# cat part.txt
```

```
[
{
    "table": "t1",
    "pkey": "cid",
    "type": "int",
    "method": "hash",
    "partitions":
    [
        { "suffix": "_0", "group": "group1" },
        { "suffix": "_1", "group": "group2" },
        { "suffix": "_2", "group": "group1" },
        { "suffix": "_3", "group": "group2" }
    ]
}
]
```

这里分库分表采用的规则是以hash方式（取模）将表分为4张，以test库下的t1表cid字段作为分区键，当mod(1, 4)得到取模后的结果等于1时，会将客户端的请求路由到组group2（192.168.17.129），并执行insert插入操作。

## (7) 启动服务

启动服务的命令如下：

```
/bin/bash sharding_start.sh
```

## (8) 关闭服务

客户端连接OneProxy 4041后台管理端口即可关闭服务，命令如下：

```
mysql -h192.168.17.131 -uadmin -pOneProxy -P4041 -e "shutdown force"
```

### 10.1.2 前端PHP/Java程序接入事项

所有的后端MySQL节点，都需要有相同的用户名及口令，由proxy-user-list选项决定，不在选项中指定的用户，既不能用来登录OneProxy，也不能为每个不同的后端实例单独指定登录信息。

如果是Java应用程序，只需要在JDBC连接时指定自动重连选项，防止OneProxy连接异常中断时业务请求终止，如下所示：  
*jdbc:mysql://localhost:3306/test characterEncoding=gbk&autoReconnect=true&failOverReadOnly=false*

## 10.2 OneProxy分库分表接入限制

OneProxy分库分表接入有如下限制。

·不支持预编译语句Prepared Statement，不支持Bind、Execute调用接口，分别如图10-4和图10-5所示。

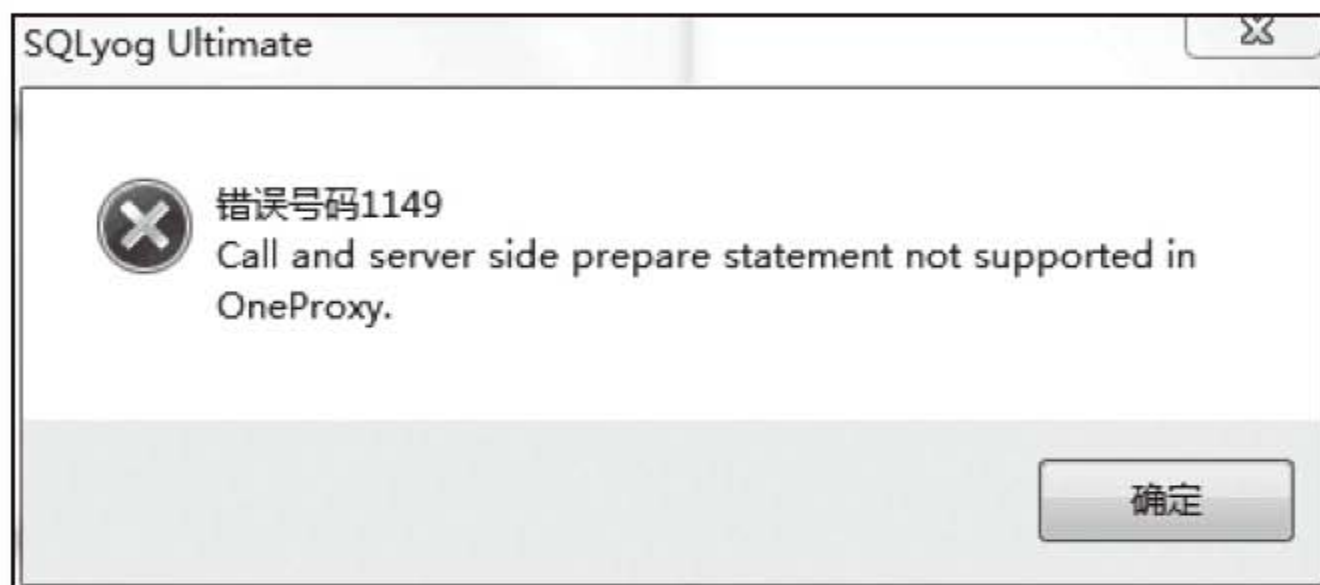


图10-4 不支持Prepared Statement ( 1 )

```
FATAL: mysql_stmt_prepare() failed
FATAL: MySQL error: 1149 "prepared statement not supported in OneProxy"
FATAL: thread#0: failed to prepare statements for test
```

图10-5 不支持Prepared Statement ( 2 )

·不支持使用use命令来切换后端数据库，默认连接的数据库由proxy-database（对所有用户指定）和proxy-user-list选项（可对不同用户指定不同的默认数据库）的值来决定。use命令可以执行，但它的含义是切换到不同的MySQL主备集群。OneProxy支持分库分表功能后，将一个主备集群视为一个数据库，连接OneProxy时如果指定数据库名，需要替换成Server Group的名字。

·禁止使用set命令，任何set命令都会直接成功返回，而不做任何处理，必须保持连接池里的每一个连接处于同样会话配置的状态。后端个别数据库参数的调整，需要直接访问后端数据库进行操作。

·默认禁止CALL、PREPARE、EXECUTE、DEALLOCATE命令，不支持存储过程和函数。

·单库（单实例）分表（比如insert/update/delete）要加分区字段名，如insert into t1(id, name)values(1, '张三');。

·在单库（单实例）分表的情况下，如果目前分了N张表，以自增id做关联查询，那么每张表的自增id都是从1开始的，在与其他表join关联查询时，数据会不



准确。

·在单库（单实例）分表的情况下，当where条件有分区列时，值不能有函数转换，也不能有算术表达式，必须是原子值，否则结果会不准确。

·在分库分表（多实例）的情况下，不支持垮库join，例如user\_0表在10.0.0.1机器里，现在要join关联查询10.0.0.2机器里的money\_detail表，则会不支持。

·在分库分表（多实例）的情况下，不支持分布式事务，例如user\_0表在10.0.0.1机器里，user\_1表在10.0.0.2机器里，现在想同时更新两张表，则会不支持。

·用于分区的列，可以是int或char类型，int对应到MySQL中各种整数类型，char对应到各种定长和变长字符串类型，日期类型在SQL中按字符串处理就可以支持。

·跨库的分页操作会在OneProxy的内存中进行，Limit中的Offset在10000以内，结果是精确的，超过10000时会做一些模糊的处理。

## 10.3 OneProxy分库分表基本测试

### 10.3.1 分库分表的功能测试

首先，让客户端连接OneProxy 3317端口，在test库下创建t1表，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "
```

```
CREATE TABLE `t1` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `cid` int(11) DEFAULT NULL,  
  `name` char(10) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `cid` (`cid`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;"
```

然后登录后端DB（192.168.17.128和192.168.17.129），这时通过如下命令会看到创建的表已经自动分表（见图10-6和图10-7）。

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test -e "show tables;"
```

```
mysql -h192.168.17.129 -uadmin -p123456 -P3306 test -e "show tables;"
```

Tables_in_test
t1_0
t1_2

2 rows in set (0.00 sec)

图10-6 后端DB-128表已拆分



Tables_in_test
t1_1
t1_3
2 rows in set (0.14 sec)

图10-7 后端DB-129表已拆分

之后，让客户端连接OneProxy 3317端口，插入测试数据，如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "
```

```
insert into t1(cid, name) values(1, 'a');
insert into t1(cid, name) values(2, 'b');
insert into t1(cid, name) values(3, 'c');
insert into t1(cid, name) values(4, 'd');
insert into t1(cid, name) values(5, 'e');
insert into t1(cid, name) values(6, 'f');
insert into t1(cid, name) values(7, 'g');
insert into t1(cid, name) values(8, 'h');
insert into t1(cid, name) values(9, 'i');
insert into t1(cid, name) values(10, 'j');"
```

并通过OneProxy查看t1表的数据，如图10-8所示。



```
MySQL [(none)]> select * from t1;
```

id	cid	name
1	4	d
2	8	h
1	1	a
2	5	e
3	9	i
1	2	b
2	6	f
3	10	j
1	3	c
2	7	g

10 rows in set (0.01 sec)

图10-8 t1表数据汇总

再次登录后端DB（192.168.17.128和192.168.17.129），通过如下命令会看到数据已经通过hash取模，规则分布在不同的表里（见图10-9和图10-10所示）。

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test
```

```
mysql -h192.168.17.129 -uadmin -p123456 -P3306 test
```

如果以上的操作运行没有问题，那么代表分库分表已经工作正常。

```
mysql> select * from t1_0;
+----+-----+-----+
| id | cid  | name |
+----+-----+-----+
| 1  | 4    | d    |
| 2  | 8    | h    |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from t1_2;
+----+-----+-----+
| id | cid  | name |
+----+-----+-----+
| 1  | 2    | b    |
| 2  | 6    | f    |
| 3  | 10   | j    |
+----+-----+-----+
3 rows in set (0.00 sec)
```

图10-9 后端DB-128表数据已拆散1

```
mysql> select * from t1_1;
+----+-----+-----+
| id | cid | name |
+----+-----+-----+
| 1  | 1   | a    |
| 2  | 5   | e    |
| 3  | 9   | i    |
+----+-----+-----+
3 rows in set (0.00 sec)

mysql> select * from t1_3;
+----+-----+-----+
| id | cid | name |
+----+-----+-----+
| 1  | 3   | c    |
| 2  | 7   | g    |
+----+-----+-----+
2 rows in set (0.00 sec)
```

图10-10 后端DB-129表数据已拆散1

现在，在客户端连接OneProxy 4041后台管理端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -pOneProxy -P4041
```

可以查看到分库分表规则，如图10-11所示。



```
MySQL [(none)]> LIST TABLES;
```

TABLENAME	KEY	TYPE	METHOD	PARTITIONS	KEYCACHE
t1	cid	int	hash	4	0

1 row in set (0.00 sec)

```
MySQL [(none)]> LIST PARTITIONS;
```

TABLENAME	PARTID	PARTITION	GROUP	VALUES	STATUS
t1	0	t1_0	group1	NULL	RW
t1	1	t1_1	group2	NULL	RW
t1	2	t1_2	group1	NULL	RW
t1	3	t1_3	group2	NULL	RW

4 rows in set (0.00 sec)

图10-11 在后台查看分表规则

查看分区表t1的QPS情况，如图10-12所示。

其中：

- Select代表总共执行的次数。
- Selrow代表总共查询的行数。
- Seltim代表总共执行的时间（毫秒）。

查看后端DB的请求，如图10-13所示。

```
MySQL [(none)]> LIST TABSTATS;
```

TABLE	INSERT	INSROW	INSTIM	UPDATE	UPDROW	UPDTIM	DELETE	DELROW	DELTIM	SELECT	SELROW	SELTIM
t1	0	0	0	0	0	0	0	0	0	22	51	150

1 row in set (0.00 sec)

图10-12 每秒请求数统计

```
MySQL [(none)]> list backend;
```

INDX	ADDRESS	TYPE	STATUS	MARKUP	REQUESTS	DEGREE	GROUP
1	127.0.0.1:3317	RW/Master	UP	0	0	0	
2	192.168.17.128:3306	RW/Master	UP	1	11	0	group1
3	192.168.17.129:3306	RW/Master	UP	1	12	0	group2

3 rows in set (0.00 sec)

图10-13 后端数据库访问请求统计

其中：

- REQUESTS代表有多少请求过来。
- STATUS下的UP代表主机存活，DOWN代表主机宕机。

可能有读者会问，今后再对新的表做分库分表时，如何动态加载呢？同样也是在后台里完成设置，如图10-14所示。

```
MySQL [(none)]> load tables '/root/oneproxy/part.txt';
Query OK, 0 rows affected (0.00 sec)

MySQL [(none)]> list tables;
+-----+-----+-----+-----+-----+-----+
| TABLENAME | KEY | TYPE | METHOD | PARTITIONS | KEYCACHE |
+-----+-----+-----+-----+-----+-----+
| t1          | cid | int  | hash  | 4           | 0         |
| t2          | cid | int  | hash  | 4           | 0         |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

MySQL [(none)]> list partitions
-> ;
+-----+-----+-----+-----+-----+-----+
| TABLENAME | PARTID | PARTITION | GROUP | VALUES | STATUS |
+-----+-----+-----+-----+-----+-----+
| t1          | 0      | t1_0      | group1 | NULL    | RW     |
| t1          | 1      | t1_1      | group2 | NULL    | RW     |
| t1          | 2      | t1_2      | group1 | NULL    | RW     |
| t1          | 3      | t1_3      | group2 | NULL    | RW     |
| t2          | 0      | t2_0      | group1 | NULL    | RW     |
| t2          | 1      | t2_1      | group2 | NULL    | RW     |
| t2          | 2      | t2_2      | group1 | NULL    | RW     |
| t2          | 3      | t2_3      | group2 | NULL    | RW     |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

图10-14 动态加载分区规则

通过load tables命令即可实现动态加载，并不用重新启动OneProxy服务进程。也可以通过Explain命令查看后端DB的命中情况。当where条件带分区列时，直接命中该表，如图10-15所示。



```
MySQL [(none)]> explain select * from t1 where cid=3;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	IX_cid	IX_cid	5	const	1	NULL

1 row in set (0.01 sec)

图10-15 查询单个分区表

如果where条件未带分区列，则会逐一扫描后端DB所有分表，如图10-16所示。

```
MySQL [(none)]> explain select * from t1 where name='c' ;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	IX_name	IX_name	33	const	1	Using index condition
1	SIMPLE	t1	ref	IX_name	IX_name	33	const	1	Using index condition
1	SIMPLE	t1	ref	IX_name	IX_name	33	const	1	Using index condition
1	SIMPLE	t1	ref	IX_name	IX_name	33	const	1	Using index condition

4 rows in set (0.40 sec)

图10-16 扫描所有分区表

### 10.3.2 分库分表的二级分区测试

二级分区的适用场景：先对订单号取模分4张表，再对时间字段range扩容，按照时间拆分数据，历史订单归档，小表增删改查的性能要比大表快得多。

首先，修改分库分表配置文件，命令如下：

```
# cat part.txt
[{
    "table": "order_info",
    "pkey": "order_sn",
    "type": "int",
    "method": "hash",
    "partitions":
    [
        { "suffix": "_0", "group": "group1" },
        { "suffix": "_1", "group": "group2" },
        { "suffix": "_2", "group": "group1" },
        { "suffix": "_3", "group": "group2" }
    ],
    "subpkey": "paytime",
    "subtype": "timestamp",
    "submethod": "range",
    "subpartitions":
    [
        { "suffix": "_2015", "group": "group1", "value": "2016-01-01 00:00:00" },
        { "suffix": "_2016", "group": "group2", "value": "2017-01-01 00:00:00" },
        { "suffix": "_2017", "group": "group1", "value": "2018-01-01 00:00:00" },
        { "suffix": "_2018", "group": "group2", "value": null }
    ]
}]
```



]

}}

这里的分库分表采用的规则是以hash方式（取模）将表分为4张，以test库下的order\_info订单表order\_sn订单号字段作为分区键，并将paytime订单支付字段按照时间range规则分为4张表，小于2016年1月1日00:00:00放入历史归档表\_2015，依此类推。当mod(1, 4)得到取模后的结果等于1时，会将客户端的请求路由到组group2（192.168.17.129），并执行insert插入操作。

然后，在客户端连接OneProxy 3317端口，在test库下创建order\_info表，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "
```

```
CREATE TABLE `order_info` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `order_sn` int(11) DEFAULT NULL,  
  `user_id` int(11) DEFAULT NULL,  
  `product` varchar(100) DEFAULT NULL,  
  `paytime` datetime DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `IX_order_sn` (`order_sn`),  
  KEY `IX_paytime` (`paytime`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

现在登录后端DB（192.168.17.128和192.168.17.129），通过如下命令会看到创建的表已经自动分表（见图10-17和图10-18）。

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test -e "show tables;"
```

```
mysql -h192.168.17.129 -uadmin -p123456 -P3306 test
```



```
mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| order_info_0_2015 |
| order_info_0_2016 |
| order_info_0_2017 |
| order_info_0_2018 |
| order_info_2_2015 |
| order_info_2_2016 |
| order_info_2_2017 |
| order_info_2_2018 |
+-----+
8 rows in set (0.00 sec)
```

图10-17 后端DB-128查看二级分区表

```
mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| order_info_1_2015 |
| order_info_1_2016 |
| order_info_1_2017 |
| order_info_1_2018 |
| order_info_3_2015 |
| order_info_3_2016 |
| order_info_3_2017 |
| order_info_3_2018 |
+-----+
8 rows in set (0.00 sec)
```

图10-18 后端DB-129查看二级分区表

之后，在客户端连接OneProxy 3317端口，插入测试数据，如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test -e "
insert into order_info(order_sn , user_id , product , paytime) values(101 , 1 , '手机' , '2015-12-31 16:00:00');
insert into order_info(order_sn , user_id , product , paytime) values(102 , 2 , '电脑' , '2016-1-1 00:00:00');
insert into order_info(order_sn , user_id , product , paytime) values(103 , 3 , '笔记本' , '2016-2-14 16:00:00');
insert into order_info(order_sn , user_id , product , paytime) values(104 , 4 , '书' , '2017-10-1 16:00:00');
insert into order_info(order_sn , user_id , product , paytime) values(105 , 5 , '咖啡' , '2018-7-1 16:00:00');
insert into order_info(order_sn , user_id , product , paytime) values(106 , 6 , '茶叶' , '2019-5-1 16:00:00');
```

并通过OneProxy查看order\_info表的数据，如图10-19所示。

```
MySQL [(none)]> select * from order_info;
```

id	order_sn	user_id	product	paytime
1	104	4	书	2017-10-01 16:00:00
1	101	1	手机	2015-12-31 16:00:00
1	105	5	咖啡	2018-07-01 16:00:00
1	102	2	电脑	2016-01-01 00:00:00
1	106	6	茶叶	2019-05-01 16:00:00
1	103	3	笔记本	2016-02-14 16:00:00

6 rows in set (0.02 sec)

图10-19 order\_info表数据汇总

再次登录后端DB（192.168.17.128和192.168.17.129），通过如下命令会看到数据已经通过hash取模规则分布在不同的表里（见图10-20和图10-21）。

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test
```



```
mysql> select * from order_info_0_2017;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1  |      104 |      4  | 书      | 2017-10-01 16:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from order_info_2_2016;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1  |      102 |      2  | 电脑    | 2016-01-01 00:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from order_info_2_2018;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1  |      106 |      6  | 茶叶    | 2019-05-01 16:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图10-20 后端DB-128表数据已拆散2

```
mysql -h192.168.17.129 -uadmin -p123456 -P3306 test
```

```
mysql> select * from order_info_1_2015;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1 | 101 | 1 | 手机 | 2015-12-31 16:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from order_info_1_2018;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1 | 105 | 5 | 咖啡 | 2018-07-01 16:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from order_info_3_2016;
+----+-----+-----+-----+-----+
| id | order_sn | user_id | product | paytime |
+----+-----+-----+-----+-----+
| 1 | 103 | 3 | 笔记本 | 2016-02-14 16:00:00 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图10-21 后端DB-129表数据已拆散2

以上的操作顺利进行，代表分库分表已经工作正常。  
有一个需要注意的地方，参见图10-22。

```
MySQL [(none)]> select * from order_info;
```

id	order_sn	user_id	product	paytime
1	104	4	书	2017-10-01 16:00:00
1	101	1	手机	2015-12-31 16:00:00
1	105	5	咖啡	2018-07-01 16:00:00
1	102	2	电脑	2016-01-01 00:00:00
1	106	6	茶叶	2019-05-01 16:00:00
1	103	3	笔记本	2016-02-14 16:00:00

```
6 rows in set (0.02 sec)
```

```
MySQL [(none)]>
MySQL [(none)]> select * from order_info where paytime >='2015-01-01';
ERROR 1044 (42000): Partitioned tables should choose at least one partition!
MySQL [(none)]> select * from order_info where paytime >='2016-01-01';
ERROR 1044 (42000): Partitioned tables should choose at least one partition!
MySQL [(none)]> select * from order_info where paytime >='2017-01-01';
ERROR 1044 (42000): Partitioned tables should choose at least one partition!
MySQL [(none)]>
MySQL [(none)]> select * from order_info where paytime <='2017-01-01';
```

id	order_sn	user_id	product	paytime
1	101	1	手机	2015-12-31 16:00:00

```
1 row in set (0.01 sec)
```

图10-22 执行报错

图10-22所示的结果完全不是我们想得到的数据，而且执行SQL报错了。对此，必须修改VALUE值，使paytime<'2020-01-0100:00:00'，这样才可以正常执行，如图10-23所示。



### 10.3.3 分库分表的聚合测试

360公司的开源产品Atlas是不支持分库分表聚合统计的，参考官网截图10-24。

#### 关于支持的语句

Atlas sharding只对sql语句提供有限的支持, 目前支持基本的Select, insert/replace, delete, update语句, 支持全部的Where语法(SQL-92标准), 不支持DDL(create drop alter)以及一些管理语句, DDL请直连MYSQL执行, 请只在Atlas上执行Select, insert, delete, update(CRUD)语句, 对于以下语句, 如果语句命中了多台dbgroup, Atlas均未做支持(如果语句只命中了一个dbgroup, 如select count(\*) from test where id < 1000, 其中dbgroup0范围是0 - 1000, 那么这些特性都是支持的)

- Limit Offset(支持Limit)
- Order by
- Group by
- Join
- ON
- Count, Max, Min等函数

请不要在Sharding的表上使用这些特性, 如果对这种特性有需求请不要让此表sharding.

图10-24 不支持聚合函数

但OneProxy部分支持以上这些语句，跨库查询结果集的合并、排序、分组汇总操作会在OneProxy的内存中进行，在生产环境下需要考虑给OneProxy机器配置更大的内存。

聚合测试步骤如下。

首先在客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

然后执行如图10-25所示的语句。

```
MySQL [(none)]> select count(*) from t1;
+-----+
| count(*) |
+-----+
|        10 |
+-----+
1 row in set (0.10 sec)

MySQL [(none)]> select sum(cid) from t1;
+-----+
| sum(cid) |
+-----+
|         55 |
+-----+
1 row in set (0.11 sec)

MySQL [(none)]> select max(cid) from t1;
+-----+
| max(cid) |
+-----+
|         10 |
+-----+
1 row in set (0.13 sec)

MySQL [(none)]> select min(cid) from t1;
+-----+
| min(cid) |
+-----+
|          1 |
+-----+
1 row in set (0.01 sec)
```



```
MySQL [(none)]> select cid,name,count(*) from t1 group by cid order by cid limit 10;
```

cid	name	count(*)
1	a	1
2	b	1
3	c	1
4	d	1
5	e	1
6	f	1
7	g	1
8	h	1
9	i	1
10	j	1

```
10 rows in set (0.00 sec)
```

图10-25 OneProxy支持简单聚合函数

如果同时执行group by cid having count(\*)语句，是不被支持的，如图10-26所示。

```
MySQL [(none)]>
MySQL [(none)]> select cid,name,count(*) from t1 group by cid having count(*)>=1;
ERROR 1044 (42000): Unsafe count distinct/average/group having/limit condition for partitioned tables!
MySQL [(none)]>
```

图10-26 OneProxy不支持复杂聚合函数



### 10.3.4 分库分表的插入测试

当执行insert、update、delete操作时，后面要加分区字段名。来看看不加分区字段名的情况。

首先，在客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

然后执行如图10-27所示的语句。

```
MySQL [(none)]> insert into t1 values(11,11,'hechunyang');  
ERROR 1044 (42000): Partitioned tables should choose at least one partition!  
MySQL [(none)]>
```

```
MySQL [(none)]> update t1 set name='nba' where name='b';  
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!  
MySQL [(none)]>  
MySQL [(none)]> delete from t1 where name='b';  
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!  
MySQL [(none)]>
```

图10-27 未加分区字段报错

从图10-27可以看到，没有加分区字段，执行报错。

加上分区字段，则会执行正常，如图10-28所示。

```
MySQL [(none)]> insert into t1(id,cid,name) values(11,11,'hechunyang');  
Query OK, 1 row affected (0.02 sec)
```

图10-28 加分区字段正常执行

### 10.3.5 分库分表不支持跨库join的测试

首先登录后端DB ( 192.168.17.128 ) , 创建一张t2表, 命令如下:

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test -e "
```

```
CREATE TABLE `t2` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `cid` int(11) DEFAULT NULL,  
  `name` char(10) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `cid` (`cid`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;"
```

然后, 在客户端连接OneProxy 3317端口, 命令如下:

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

执行如图10-29所示的语句。

```
MySQL [(none)]> select t1.* from t1 join t2 on t1.id = t2.id;  
ERROR 1146 (42S02): Table 'test.t2' doesn't exist  
MySQL [(none)]>
```

图10-29 不支持跨库join

此时会报错, 提示t2表找不到, 但实际上刚才已经创建了t2表, 在业务测试时需要特别注意。

### 10.3.6 分库分表不支持分布式事务的测试

首先，在客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

然后执行如图10-30所示的语句。



```
MySQL [(none)]> select * from t1;
```

id	cid	name
1	4	d
2	8	h
1	1	a
2	5	e
3	9	i
1	2	b
2	6	f
3	10	j
1	3	c
2	7	g
11	11	hechunyang

```
11 rows in set (0.00 sec)
```

```
MySQL [(none)]> delete from t1 where cid < 15;
```

```
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!
```

```
MySQL [(none)]>
```

```
MySQL [(none)]> delete from t1;
```

```
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!
```

```
MySQL [(none)]>
```

```
MySQL [(none)]> update t1 set name='nba' where cid in (1,2,3);
```

```
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!
```

```
MySQL [(none)]>
```

图10-30 不支持分布式事务

由于数据被分散到后端DB上的4张子表里 (t1\_0/t1\_1/t1\_2/t1\_3)，要让delete操作同时删除四张表里的数据是无法实现的，如图10-30所示。

此外，也不支持insert...select...操作，如图10-31所示。

```
MySQL [test]>  
MySQL [test]> insert into t2 select * from t1;  
ERROR 1044 (42000): Partitioned tables should choose only one partition for DML queries!  
MySQL [test]>
```

图10-31 不支持分布式事务



### 10.3.7 分库分表不支持存储过程的测试

首先，登录后端DB ( 192.168.17.128 )，创建t1\_proc存储过程，命令如下：

```
mysql -h192.168.17.128 -uadmin -p123456 -P3306 test
DELIMITER $$
USE `test`$$
DROP PROCEDURE IF EXISTS `t1_proc`$$
CREATE DEFINER=`admin`@`%` PROCEDURE `t1_proc`()
BEGIN
    SELECT * FROM t1 WHERE cid = 1;
END$$
DELIMITER ;
```

然后，在客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

并执行如图10-32所示的语句。

```
MySQL [test]> call t1_proc();
ERROR 1149 (42000): Call and server side prepare statement not supported in OneProxy.
MySQL [test]>
```

图10-32 不支持存储过程

若执行存储过程报错，则表示不支持存储过程。

综上所述，分库分表后，会有很多的限制条件，应用接入时需要在线下重点测试！



## 10.4 搭建OneProxy高可用故障切换HA

由于OneProxy内置了HA功能，因此可以不需要安装第三方软件实现高可用故障切换，也可以使用KeepAlived实现。如果将OneProxy部署在多台机器上，构成一个集群，那么应用就可以在程序中实现错误尝试机制，或者使用F5、HAProxy、LVS等软硬件做端口转发，这样，就可以根据一定的策略转发到任何一个OneProxy节点，从而做到OneProxy无单点服务。

### 1. 通过OneProxy内置功能实现高可用HA

假设两台OneProxy机器上的网卡名称为“eth0”，那么只需要在两台机器启动OneProxy的命令中新增一个参数“--vip-address=VIP地址/eth0:1”就可以了，OneProxy会自动检测VIP地址。如果一台机器重启，那么另一台机器会在1~2秒内自动接管VIP地址，以确保系统高可用，完全省去了多个复杂的集群软件的安全和配置，启动脚本如下：

```
# cat sharding_start.sh
#!/bin/bash
export ONEPROXY_HOME=/root/oneproxy
${ONEPROXY_HOME}/oneproxy --keepalive --proxy-address=:3317 \
--proxy-master-addresses=192.168.17.128:3306@group1 \
--proxy-master-addresses=192.168.17.129:3306@group2 \
--proxy-user-list=admin/9D7E55EAF8912CCBF32069443FAC452794F8941B@test \
--admin-address=:4041 \
--proxy-part-tables=${ONEPROXY_HOME}/part.txt \
--proxy-parallel-degree=8 \
--event-threads=8 \
--proxy-group-policy=group1:0 --proxy-group-policy=group2:0 \
--proxy-group-security=group1:0 --proxy-group-security=group2:0 \
--vip-address=192.168.17.200/eth0:1 \
--log-file=${ONEPROXY_HOME}/oneproxy.log \
--pid-file=${ONEPROXY_HOME}/oneproxy.pid
```

启动脚本以后，通过ifconfig命令，可以查看到多了一个192.168.17.200 VIP地址，如图10-33所示。

```
[root@MHA ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:A0:0B:E6
          inet addr:192.168.17.131  Bcast:192.168.17.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fea0:be6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3577 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3228 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:347112 (338.9 KiB)  TX bytes:264834 (258.6 KiB)

eth0:1    Link encap:Ethernet  HWaddr 00:0C:29:A0:0B:E6
          inet addr:192.168.17.200  Bcast:192.168.17.255  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:9737 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9737 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:566557 (553.2 KiB)  TX bytes:566557 (553.2 KiB)
```

图10-33 多了一个VIP地址

## 2.故障转移测试

当kill掉oneproxy进程，或者把机器重启后，VIP会自动飘移到OneProxy备机上，切换时间为1~2秒。这里需要注意的是：故障切换是不打印日志的，如果想通过日志查看是何时切换的，目前的OneProxy 5.8.0版本（libevent:2.0.22-stable）还不支持。



## 10.5 OneProxy黑名单SQL防火墙搭建测试

前几年，国内某网站的用户数据库被黑客公开发布，几百万用户的登录名及密码被公开泄露，随后又有多家网站的用户名及密码被流传于网络，因此引发众多网民对自己账号、密码等互联网信息被盗取的普遍担忧，触动了每一位用户的神经，网络安全成为现在互联网的焦点。下面先了解什么是SQL注入。

SQL注入（SQL Injection），就是通过把SQL命令插入Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。具体来说，它是利用现有应用程序将（恶意的）SQL命令注入后台数据库引擎执行的能力，它可以通过在Web表单中输入（恶意）SQL语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行SQL语句。

对于这类情况，可利用OneProxy自带的SQL防火墙将事先定义好的黑名单SQL语句直接拦截，从而保护数据库的安全。来看看具体步骤。

### （1）开启OneProxy防火墙

启动脚本如下：

```
# cat sharding_start.sh
#!/bin/bash
export ONEPROXY_HOME=/root/oneproxy
${ONEPROXY_HOME}/oneproxy --keepalive --proxy-address=:3317 \
--proxy-master-addresses=192.168.17.128:3306@group1 \
--proxy-master-addresses=192.168.17.129:3306@group2 \
--proxy-user-list=admin/9D7E55EAF8912CCBF32069443FAC452794F8941B@test \
--admin-address=:4041 \
--proxy-part-tables=${ONEPROXY_HOME}/part.txt \
--proxy-parallel-degree=8 \
--event-threads=2 \
--proxy-group-policy=group1:0 --proxy-group-policy=group2:0 \
--proxy-group-security=group1:0 --proxy-group-security=group2:0 \
--vip-address=192.168.17.200/eth0:1 \
--proxy-enable-firewall \
--proxy-firewall-watchonly \
--proxy-black-query=${ONEPROXY_HOME}/black_sql.txt \
```



```
--log-file=${ONEPROXY_HOME}/oneproxy.log \  
--pid-file=${ONEPROXY_HOME}/oneproxy.pid
```

参数解释如下。

- proxy-enable-firewall：开启内置SQL防火墙。
- proxy-firewall-watchonly：开启SQL防火墙的观察模式。
- proxy-black-query：自定义黑名单SQL。

这里要说明一下什么是黑名单SQL。黑名单SQL的文件名为black\_sql.txt。其内容如下：

```
# cat black_sql.txt  
1select * from t1  
3update t1 set name =  
4delete from t1  
4delete from t1 where cid >
```

参数解释如下。

- ?号代表变量（任意值）。
- 数字1代表select，数字2代表insert，数字3代表update，数字4代表delete。

## （2）OneProxy防火墙黑名单拦截演示

在客户端连接OneProxy 3317端口，命令如下：

```
mysql -h192.168.17.131 -uadmin -p123456 -P3317 test
```

然后执行黑名单中的SQL语句，如图10-34所示。

```

MySQL [(none)]> select * from t1;
ERROR 1045 (28000): Statement was blocked by OneProxy SQL Firewall
MySQL [(none)]>
MySQL [(none)]> update t1 set name='hechunyang';
ERROR 1045 (28000): Statement was blocked by OneProxy SQL Firewall
MySQL [(none)]>
MySQL [(none)]> delete from t1;
ERROR 1045 (28000): Statement was blocked by OneProxy SQL Firewall
MySQL [(none)]>
MySQL [(none)]> delete from t1 where cid > 10;
ERROR 1045 (28000): Statement was blocked by OneProxy SQL Firewall
MySQL [(none)]>
MySQL [(none)]> select * from t1 where cid = 1;
+----+-----+-----+
| id | cid | name |
+----+-----+-----+
| 1  | 1  | a    |
+----+-----+-----+
1 row in set (0.00 sec)

```

图10-34 黑名单SQL被防火墙拦截

从图10-34可以看到，黑名单SQL直接被拦截，而合法的SQL是正常执行的。

如果在客户端连接OneProxy 4041后台管理端口，则可以列出定义的黑名单SQL，连接命令如下：

```
mysql -h192.168.17.131 -uadmin -pOneProxy -P4041
```

结果如图10-35所示。

```
MySQL [(none)]> LIST BLACKSQL
-> ;
```

HASHCODE	EXECS	SQLTEXT
55778364	0	delete from t1
1769265575	0	select * from t1
2269172827	0	update t1 set name = ?
2114990179	0	delete from t1 where cid > ?

4 rows in set (0.00 sec)

图10-35 管理后台查看拦截的SQL

若再次添加黑名单SQL，如何动态加载呢？同样，也是在后台完成设置，如图10-36所示。

```
MySQL [(none)]> load blacksql '/root/oneproxy/black_sql.txt';
Query OK, 0 rows affected (0.00 sec)
```

```
MySQL [(none)]> LIST BLACKSQL;
```

HASHCODE	EXECS	SQLTEXT
55778364	0	delete from t1
1769265575	0	select * from t1
2269172827	0	update t1 set name = ?
186087535	0	select * from t1 cid > ?
219957192	0	select * from t1 where l=1
2114990179	0	delete from t1 where cid > ?
2028718327	1	select * from t1 where cid > ?

7 rows in set (0.00 sec)

图10-36 动态加载黑名单规则



通过load blacksql命令即可实现动态加载，并不用重新启动OneProxy服务进程。

要注意的是，这里有个bug，就是在执行select\*from t1 where 1=1;防火墙失效时，仍旧可以跳过黑名单规则正常执行，但该规则已经在黑名单里设置了，如图10-37所示。

也可以动态关闭SQL防火墙，登录后台执行如图10-38所示的命令。

这样就可以关闭SQL防火墙了。

```
[root@MHA oneproxy]# admin -e "list blacksql"
```

HASHCODE	EXECS	SQLTEXT
55778364	0	delete from t1
1769265575	0	select * from t1
2269172827	0	update t1 set name = ?
186087535	0	select * from t1 cid > ?
219957192	0	select * from t1 where 1=1
2114990179	0	delete from t1 where cid > ?
2028718327	1	select * from t1 where cid > ?

```
[root@MHA oneproxy]#
[root@MHA oneproxy]# op -e "select * from t1 where 1=1;"
```

id	cid	name
1	4	d
2	8	h
1	1	a
2	5	e
3	9	i
1	2	b
2	6	f
3	10	j
1	3	c
2	7	g

```
[root@MHA oneproxy]# op -e "select * from t1;"
ERROR 1045 (28000) at line 1: Statement was blocked by OneProxy SQL Firewall
[root@MHA oneproxy]# _
```

图10-37 防火墙bug

```
MySQL [(none)]> SET SQL_FIREWALL OFF;
Query OK, 0 rows affected (0.00 sec)
```

```
MySQL [(none)]> LIST OPTION;
```

OPTION	VALUE
SQLSTATS	ON
TABSTATS	ON
USERSTATS	OFF
TRANS_DEBUG	OFF
LOG_SQLERROR	OFF
SQL_DEBUG	OFF
TOKEN_DEBUG	OFF
SECURITY_LEVEL	0
SQL_FIREWALL	OFF
DML_FIREWALL	OFF
WATCH_MODE	ON
SQL_SIGNATURE	OFF
IP_FIREWALL	OFF
LOGIN_RESTRICT	OFF
STRICT_POOLMAX	OFF
UNSAFE_GROUP	OFF
MULTI_INSERTS	OFF
PARALLEL_DEGREE	8
PARALLEL_REQUESTS	64
MAX_CACHE_ROWS	100
MAX_IDLE_TIME	300

```
21 rows in set (0.00 sec)
```

图10-38 动态关闭防火墙





## 第11章

# Lepus慢日志 分析平台搭建与维护

DBA在分析慢查询日志时，通常是登录服务器后采用mysqldumpslow工具进行分析的，但这样很不直观，而Lepus作为一款免费开源的慢日志分析平台系统，可将系统收集的慢查询SQL语句展示在网页里，并通过邮件形式推送给开发者或DBA进行优化。

Lepus是基于LAMP环境（Linux+Apache+MySQL+PHP）搭建的。首先要把Lepus监控搭建起来，再才能部署慢日志分析平台。

Lepus目前主要是有以下功能和特性。

- 无需Agent即可实现远程监控云中数据库。
- 在Web上直观地管理和监控数据库。
- 实时对MySQL的健康进行监视和报警。
- 实时对MySQL的复制进行监视和报警。
- 实时对MySQL的资源进行监控和分析。
- 实时对MySQL的缓存等性能进行监控。
- 实时对InnoDB IO的性能进行监控。
- MySQL表空间增长趋势分析。
- 可视化MySQL慢查询在线分析。
- MySQL慢查询自动推送。
- MySQL AWR在线性能分析。

## 11.1 Lepus基础组件的安装

首先，安装Apache、PHP、MySQL，以及PHP连接MySQL库组件的接口，命令如下：

```
# yum install -y httpd php php-devel mysql mysql-server php-mysql
```

然后，安装Python基础模块，命令如下：

```
# wget http://cdn.lepus.cc/cdn-cache/software/MySQLdb-python.zip
```

```
# unzip MySQLdb-python.zip
```

```
# cd MySQLdb1-master/
```

```
# which mysql_config
```

```
/usr/local/mysql/bin/mysql_config
```

对vim site.cfg进行修改，改为mysql\_config绝对路径，命令如下：

```
mysql_config = /usr/local/mysql/bin/mysql_config
```

```
# python setup.py build
```

```
# python setup.py install
```

最后，安装Lepus监控系统，具体步骤如下。

1) 上传软件包到监控机服务器并解压缩软件到系统。

```
# lepus.zip请在华章网站下载
```

```
# unzip lepus.zip
```

2) 在监控机中创建监控数据库并授权。

```
mysql> create database lepus default character set utf8;
```

```
mysql> grant select , insert , update , delete , create on lepus.* to
```

```
'lepus_user'@'localhost' identified by 'lepus';
```

```
mysql> flush privileges;
```

导入SQL文件夹里的SQL文件（表结构和数据文件），如下：

```
# mysql -uroot -plepus lepus <sql/lepus_table.sql
```

```
# mysql -uroot -lepus lepus <sql/lepus_data.sql
```

3) 安装Lpeus程序。

通过如下命令进入软件包的python文件夹。



*# cd python/*

授予install.sh可执行权限，命令如下：

*# chmod 755 install.sh*

执行安装，命令如下：

*# ./install.sh*

4) 修改配置文件。

通过如下命令进入安装目录，默认为/usr/local/lepus。

*# cd /usr/local/lepus/*

*# vim etc/config.ini*

修改config.ini配置文件，将host改为监控机MySQL数据库的连接地址。

*[monitor\_server]*

*host="localhost"*

*port=3306*

*user="lepus\_user"*

*passwd="lepus"*

*dbname="lepus"*

5) 启动Lepus，命令如下：

*# /usr/local/lepus/lepus start*

现在可以安装Web管理台了，命令如下：

*# cp -rf php/\* /var/www/html/htdocs/*

打开application\config\database.php文件，修改PHP连接监控服务器的数据库信息：

*\$db['default']['hostname'] = 'localhost';*

*\$db['default']['username'] = 'lepus\_user';*

*\$db['default']['password'] = 'lepus';*

*\$db['default']['database'] = lepus;*

*\$db['default']['dbdriver'] = 'mysql';*

登录添加主机和监控。通过浏览器输入IP地址或域名打开监控界面，即可登录系统（见图11-1）。默认管理员账号和密码为admin/Lepusadmin，登录后请修改管理员密码，增加普通账号。

监控 / 健康监控最新监控时间:20

请输入标签

连接线程数

活动线程数

排序

正序

Q 搜索

C 重置

刷新

服务器	基本信息					线程			网络		查
标签	连接	角色	运行时间	版本		连接线程数	活动线程数	等待线程数	接收	发送	每秒查询
192.168.111.78:3306 _111_144	失败	—	—	—		—	—	—	—	—	—
192.168.111.77:3306 _111_145	成功	S	41天	10.0.21-MariaDB-log		2	3	3	115KB	115KB	545
192.168.111.79:3304 master	成功	M	70天	10.0.21-MariaDB-enterprise-log		39	3	0	28KB	228KB	746
192.168.111.77:3306 master	成功	S	61天	10.0.21-MariaDB-enterprise-log		2	1	17	2KB	13KB	11
192.168.111.77:3306 slave1	成功	M	82天	10.0.21-MariaDB-enterprise-log		38	3	0	9KB	47KB	41

服务器			线程					二进制日志		主库位置			
主机	标签	角色	Gtid模式	只读	IO	SQL	备库延时	当前文件	位置	日志文件	位置	空间	图表
192.168.111.78:3306	-slave1	master	OFF	OFF	---	---	---	---	---	mysql-bin.002747	100386680	6GB	
192.168.111.77:3306	master	slave	OFF	ON	Run	Run	00:00:00	mysql-bin.002747	100386680	mysql-bin.002747	100386680	---	
192.168.111.79:3306	slave2	master/slave	OFF	ON	Run	Run	00:00:00	mysql-bin.002747	100386680	mysql-bin.002747	100386680	---	
192.168.111.77:3304	master	master	OFF	OFF	---	---	---	---	---	mysql-bin.000461	42485207	20GB	

图11-1 Lepus主从复制监控



## 11.2 安装percona-toolkit工具

Lepus的慢查询分析平台是独立于监控系统的模块的，它使用percona-toolkit工具来采集和记录慢查询日志，并且会部署一个Lepus提供的shell脚本来进行数据采集。该脚本会自动开启数据库的慢查询日志，并且按分钟对慢查询日志进行切割，然后将收集的慢查询日志的数据发送到监控机数据库，再通过Lepus系统就可以分析慢查询了。

下面是percona-toolkit的安装步骤。

1) 安装percona-toolkit，命令如下：

```
# yum -y install perl-IO-Socket-SSL
# yum -y install perl-DBI
# yum -y install perl-DBD-MySQL
# wget
https://www.percona.com/downloads/percona-toolkit/2.2.16/tarball/percona-toolkit-2.2.16.tar.gz
# tar zxvf percona-toolkit-2.2.16.tar.gz
# cd percona-toolkit-2.2.16
# perl Makefile.PL
# make
# make install
```

2) 从监控机拷贝Lepus提供的慢查询分析脚本到MySQL端。

```
# scp /usr/local/lepus/client/mysql/lepus_slowquery.sh
root@192.168.111.78:/usr/local/sbin/
# chmod 755 /usr/local/sbin/lepus_slowquery.sh
```

3) 编辑文件修改配置，在这里需要指定Lepus监控机数据库的地址、本地MySQL地址，以及存储慢查询的路径和慢查询时间，另外还需要配置一个Lepus主机的server\_id，具体如下。

```
#vim /usr/local/sbin/lepus_slowquery.sh
#config lepus database server
lepus_db_host="192.168.111.82"
lepus_db_port=3306
lepus_db_user="admin"
```



```

lepus_db_password="123456"
lepus_db_database="lepus"
#config mysql server
mysql_client="/usr/local/mysql/bin/mysql"
mysql_host="192.168.111.78"
mysql_port=3306
mysql_user="admin"
mysql_password="123456"
#config slowquery
slowquery_dir="/data/mysql/slowlog/"
slowquery_long_time=1
slowquery_file=`$mysql_client -h$mysql_host -P$mysql_port
-u$mysql_user -p$mysql_password -e "show variables like
'slow_query_log_file'"/grep log/awk '{print $2}'`
pt_query_digest="/usr/local/bin/pt-query-digest"
#config server_id
lepus_server_id=271

```

注意，lepus\_server\_id的值需要从系统中获取。进入MySQL服务器配置，在部署脚本的主机前查询到当前ID即为主机的server\_id。

lepus\_server\_id必须和MySQL服务器配置里的服务器ID一一对应，否则可能无法查询到该主机的慢查询，如图11-2所示。



图11-2 服务器ID

配置完成后保存，并加入计划任务。因为慢查询做了按分钟的切割设置，因此建议计划任务时间间隔在5分钟之内，命令如下：

```
# crontab -l
*/5 * * * * /bin/bash /usr/local/sbin/lepus_slowquery.sh >/dev/null 2>&1
```

配置完成后，稍等片刻，即可在慢查询分析平台查看该库的慢查询日志（见图11-3）。

#### 4) 开启慢查询自动推送功能。

慢查询自动推送是指定时将系统收集到的慢查询TOP数据推送给相应开发人员进行优化，无需在每台数据库上部署脚本。MySQL慢查询的自动推送功能将通过任务完成。如果需要推送，则得在监控机部署如下任务。

```
# crontab -l
*/1 * * * * curl
http://192.168.111.82/mysqlmonitor/index.php/task/send_mysql_slowquery_mail >/dev/null 2>&1
```

任务部署完成后，则会按照配置的时间将慢查询推送给对应人员。慢查询以邮件的方式推送给哪些人需要在MySQL主机配置中设置（见图11-4），若邮箱为空，则该数据库主机不会发送慢查询推送。

# MySQL 慢查询分析

主页 / MySQL监控 / 慢查询分析

主机		▼ 时间	2016-02-16 16:00	-	2016-02-23 16:00	排序	最近时间	▼	倒序	▼	Q 搜索	C 重置
SQL		查询时间			锁等待时间							
校验值	抽象语句 展开所有	数据库	用户	最近时间	次数	平均	最小	最大	总计	最小	最大	
11295828619216035301	+select count(pl.id) from p_loan pl :		devs	2016-02-23 15:47:00	31178	2.413	2.000	5.455	2.8327	0.00003	0.00083	
2589655896978760154	+select pl.id, pl.loan_no as loanno,:		devs	2016-02-23 15:40:12	68	2.097	2.002	2.420	0.0152	0.00009	0.00039	
12894737561460943613	+select count(?) from user u left jo:		devs	2016-02-23 15:34:42	219	2.134	2.001	2.620	0.0237	0.00004	0.00019	
15216983452305686868	+select pl.id, pl.loan_no as loanno,:		devs	2016-02-23 13:57:41	2343	3.772	2.005	4.683	0.3957	0.00008	0.00103	
9919018602010958573	+select id as id, id_num as idnum, a:		devs	2016-02-23 12:20:51	13	2.053	2.004	2.126	0.0071	0.00042	0.00072	
14799804637239350472	+select count(cqw.id) from credit_qu:		_dev	2016-02-23 12:03:25	15	8.468	3.285	37.58	0.0068	0.00009	0.00501	
2776260014704128821	+select count(cqw.id) from credit_qu:		dev	2016-02-23 11:52:41	41	12.56	2.487	79.73	0.0040	0.00004	0.00016	
9316725730261614066	+select pl.id, pl.loan_no as loanno,:		devs	2016-02-23 11:46:57	2	2.097	2.049	2.145	0.0005	0.00024	0.00028	
4844085500606352726	+select pl.id, pl.loan_no as loanno,:		devs	2016-02-23 10:59:28	1	2.109	2.109	2.109	0.0002	0.00024	0.00024	
12510289504428231164	+select pl.id, pl.loan_no as loanno,:		_devs	2016-02-23 10:32:56	30	6.347	2.087	14.70	0.0060	0.00009	0.00032	



MySQL 慢查询分析

主页 / MySQL监控 / 慢查询分析

返回列表

关闭

数据库			用户	_devs		
校验值	7688044299177213205		次数	1		
首次出现	2016-02-19 18:59:25		最近时间	2016-02-19 18:59:25		
update i, p_loan p set i.live_province = p.region where p.id = i.loan_id and i.live_province is ?						
update i, p_loan p set i.live_province = p.region where p.id = i.loan_id and i.live_province is null						
查询时间	Query_time_sum	Query_time_min	Query_time_max	Query_time_pct_95	Query_time_stddev	Query_time_median
	31.0225	31.0225	31.0225	31.0225	0	31.0225
锁等待时间	Lock_time_sum	Lock_time_min	Lock_time_max	Lock_time_pct_95	Lock_time_stddev	Lock_time_median
	0.000166	0.000166	0.000166	0.000166	0	0.000166
发送行数	Rows_sent_sum	Rows_sent_min	Rows_sent_max	Rows_sent_pct_95	Rows_sent_stddev	Rows_sent_median
	0	0	0	0	0	0
扫描行数	Rows_examined_sum	Rows_examined_min	Rows_examined_max	Rows_examined_pct_95	Rows_examined_stddev	Rows_examined_median
	900524	900524	900524	900524	0	900524

图11-3 Lepus慢查询日志展示

监控

开

发送邮件

开

告警邮件收件人

@

hechunyang@puhuifinance.com

发送短信

关

告警短信收件人

@

多人请用;分隔

慢查询

开

慢查询推送邮箱

@

hechunyang@puhuifinance.com

图11-4 设置慢查询发送邮件人

邮件

*邮件协议	smtp
*邮件类型	html
*SMTP主机	smtp.126.com
*SMTP端口	25
*SMTP账号	noreplymail
*SMTP密码	
*SMTP超时时间	10
*邮件发件人	noreplymail@126.com

图11-4 (续)

