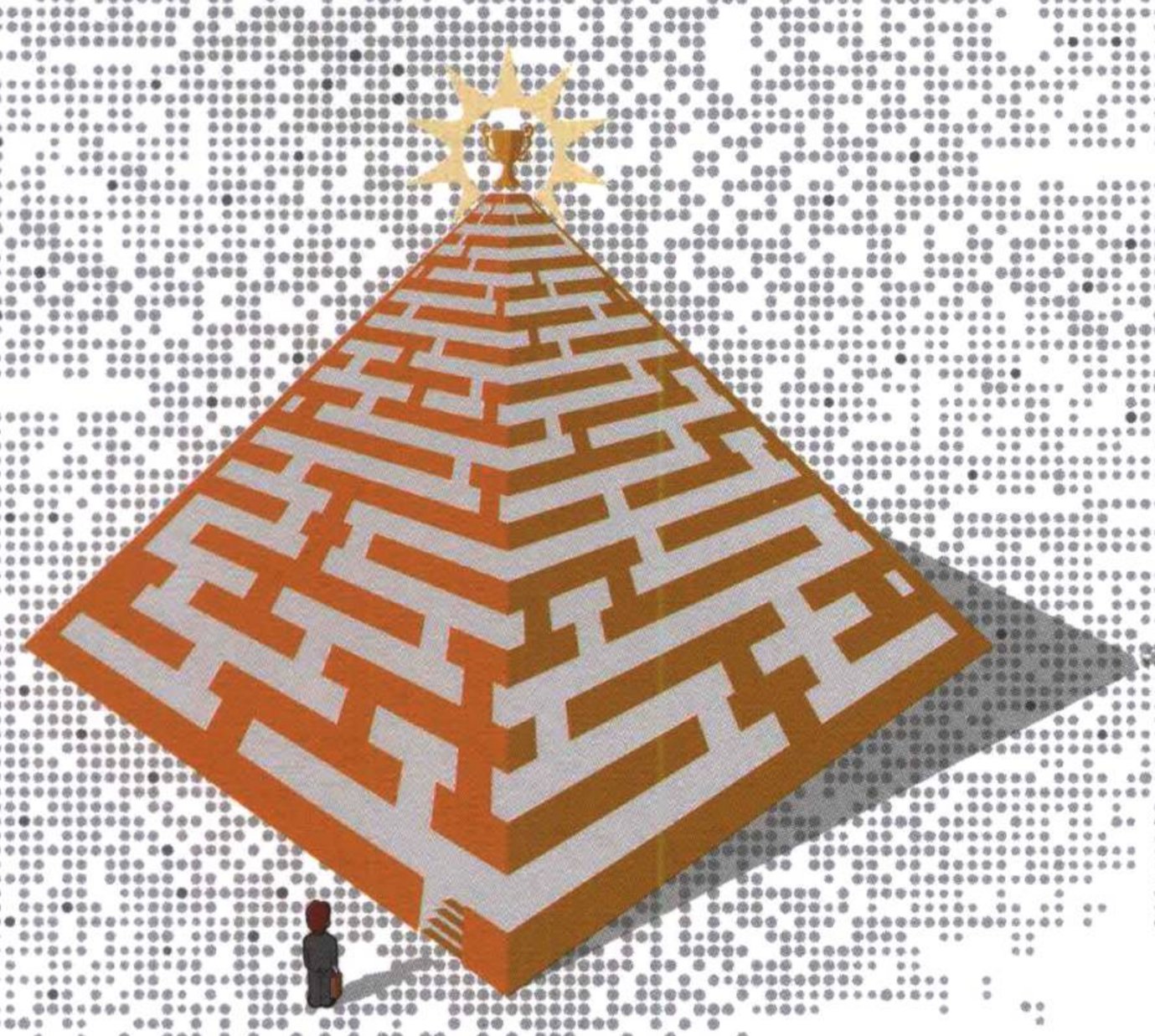
Broadview®
www.broadview.com.cn



十年磨一剑，汇集作者多年MySQL数据库领域的一线实战与教学经验

由浅入深地剖析MySQL的体系结构、备份恢复、复制、高可用集群架构、优化、故障排查、新版本特性、监控、升级及技术面试等知识点

MySQL
王者晋级之路

张甦 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

张 甦

极数学院创始人之一，51CTO知名博主。近十年互联网线上处理及培训经验，专注于MySQL数据库，对MongoDB、Redis等NoSQL数据库以及Hadoop生态圈相关技术有深入研究。

曾就职于数据库服务公司、某大型电商平台，及汽车类网站等大型互联网公司。麾下的学员遍布各大企业。

MySQL
王者晋级之路

张甦 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

目前大部分软件开发平台都基于 Linux，很多互联网公司都把 MySQL 作为后端数据存储的数据库。如果把 MySQL 比喻成数据库界的一条巨龙，则本书涵盖的所有知识点就是这条巨龙的组成部分。

本书深入剖析 MySQL 数据库体系结构，实战演练备份恢复、主从复制，详解高可用集群架构的设计与实践过程，详细梳理优化思路，展现新版本的特性，并与真实生产案例相结合，通过核心原理到“王者”实战，全面覆盖 MySQL 数据库的知识点。

本书适合熟悉 Linux 系统且想提升 MySQL 水平的读者。

图书在版编目（CIP）数据

MySQL 王者晋级之路 / 张甦著. —北京：电子工业出版社，2018.3
ISBN 978-7-121-33679-9

Ⅰ. ①M… Ⅱ. ①张… Ⅲ. ①SQL 语言—程序设计 Ⅳ. ①TP311.132.3

中国版本图书馆 CIP 数据核字（2018）第 028134 号

责任编辑：陈晓猛
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：21.5　　字数：412.9 千字
版　　次：2018 年 3 月第 1 版
印　　次：2018 年 3 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

专家评语

在这个时代，能安安静静去读一本书的机会越来越少，能踏踏实实去深入研究技术的人也越来越少，这就越发显出可以忍耐孤独寂寞、倾心写作的人的难能可贵。在亲身经历过艰苦而漫长的实践之后，我也越来越敬仰那些笔耕不辍、钟情翰墨的作者们。

初识张甦，我就有一见如故的亲切感。当知道他已经在 MySQL 领域默默耕耘多年，自己培养出了一批又一批的学生的时候，我更是敬佩不已。在繁忙的工作中，还能抽出时间完成本书的编写，这是需要勇气和毅力的。

本书是对 MySQL 基础知识的全面解析，也是他多年工作和教学经验的结晶，其内容几乎涵盖了初学者对 MySQL 知识需求的方方面面，是一本非常值得学习的 MySQL 著作，本书将会带你进入 MySQL 的神奇殿堂。在我看来本书更是拙作《MySQL 运维内参》的黄金搭档，一个用于知识点的全面普及，一个用于深入了解 MySQL 的内参指导，但愿二者结合，能够帮助所有愿意学习和使用 MySQL 的人。

周彦伟

《MySQL 运维内参》作者、中国 MySQL 用户组主席

总体来说，市面上 MySQL 类的经典书不多，其中一个重要原因就是 MySQL DBA 的工作内容大而全，充分享受了开源红利，理论学习和工作实践还是存在差别的。无论是学习 MySQL，还是学习 Oracle，数据库体系结构都是重中之重，需要在实践中不断总结，这一点我和作者的观点吻合。

作者收集整理了自己多年的一线经验，以一种轻松明快的文风来讲述 MySQL 体系结构和

运维相关知识。通过本书，不仅能对 MySQL 知识体系有一个全面的了解，还能够一窥一线 DBA 的工作内容和技巧，相信会带给你一些新的思考和方向。

杨建荣

DBAplus 社群发起人，Oracle ACE，《Oracle DBA 工作笔记》作者

感谢张甦邀请，在拿到这本书大纲的时候，我问了张甦一个问题："你写这本书的初心是什么？"答曰："我想将自己近十年的数据库运维经验和授课感悟写给那些刚跨入此领域的学生们。"对此，我深有同感，从业十余年，我发现数据库 DBA 这个圈子大部分都是半路出家的，很少有人在学校里面就决定了自己要做 DBA，要从事数据库相关领域的工作，更多的是随着工作职责的变更，通过自学从而走上这条路的。我认识的朋友中有从开发转 DBA 的，也有从运维转 DBA 的，大部分的情况都是数据库没有专人管理，开始是兼顾，后期慢慢就转型成了专职的 DBA 了。

而在这个过程中，不免会走很多的弯路，尤其是在十年前那个信息匮乏、交流不便的年代，出了问题不知道是什么原因导致的，面对需求没有什么相关的解决方案可以参考，他人的经验更是寥寥无几。目前这个时代正是信息大爆炸的时代，数据库也不再那么神秘，我们可以从各个渠道获取相关的知识，可以看同样的 case 别人是怎么解决的，相同的问题别的公司是怎么处理的，那么我们为什么还需要 MySQL 类的书籍呢？这也是我的第二个疑问，我同样问了张甦，他是这么回答的："这本书包含了我近十年的工作经验总结，涵盖了我能想到的 MySQL 的各个方面，对于初学者来说是很好的入门书籍。"

我翻阅了大纲，这本书从安装下载、部署启动，到参数、索引、锁、事务等 MySQL 常用和不常用的知识点都有详细介绍，描述得非常直白，并配有详细的案例，确实非常适合入门级读者进行学习。我们从业者常见的分享经常会忽略一些基础概念的解释和介绍，默认受众已经知道了这部分内容，而张甦这本书完全假设受众为零基础的读者，深入浅出地介绍了 MySQL 的相关知识，建议作为 MySQL 的入门书籍。

最后，数据库 DBA 在人数上属于较为小众的职业，但是选择了这个职业的人都是比较有趣的，如果你也选择了这个职业，希望我们共同将其发扬光大，让更多的人理解这个职业。

肖鹏

微博研发中心 技术副总监

随着互联网时代的兴起，MySQL 在数据库领域日益显现出举足轻重的地位，它不断扩大的用户群体就是很好的证明。在这样的背景下，需要越来越多的人在知识资源方面为之付出，将自己对 MySQL 数据库的学习过程、运维经验、个人理解等记录下来，为行业中的后来者燃起一盏明灯，让他们少走一些弯路。

这正是本书写作的初衷。书中汇集了作者多年来在 MySQL 运维及教学过程中不断总结、不断思考的成果，内容丰富、体例清晰。不仅有对基础知识的阐述，比如 MySQL 作为一个关系型通用数据库所具备的模块、组织架构、功能及特点等，还有对实际操作的指导，比如告诉你在运维中应该重点关注什么，如何控制数据库的行为，如何解决问题等，旨在让读者对数据库有一个更深入的理解。本书深入浅出，语言平实又不乏幽默，轻快又不失严谨，是一本值得学习的好书。相信开卷有益，每位读者都能在轻松畅快的阅读中有意外的收获。

王竹峰

去哪儿网数据库总监 Oracle MySQL ACE

前　言

找到一份合适的工作，就像在工作的八小时之内有了一个心仪的恋人——MySQL 数据库就是我的甜蜜爱恋。

我上学时特别不爱学习，也从来没有想过，也不敢想，有一天自己会写本书。作为一个在数据库领域摸爬滚打近十年的“老司机”，从一个什么都不会的菜鸟，做到公司的高管兼资深数据库讲师，我很希望把自己这些年积累下来的实战经验和一些学习 MySQL 数据库的心得体会分享给大家。我很能理解那些刚进入数据库领域的同学的困惑，因为自己刚学习 MySQL 的时候，就跟无头苍蝇一样，遇到报错后在网上到处找资料，关键是看完资料后有些问题可能还是解决不了，因为不知道哪种处理方法是对的。当时很崩溃，真心不知道该从哪里下手学习才好。别人还经常推荐一些过于偏向概念性叙述的数据库书籍，越看越晕。我当时就想为什么非要把数据库的知识点说得这么烦琐、深奥，让别人看不懂、理解不了呢？

写作本书的目的

我平常喜欢写一些技术博客，在 51CTO 上面写了一篇“青铜到王者，快速提高 MySQL 数据库段位”的文章，这篇文章指明了 MySQL 数据库的一个学习方向，但没有深入展开讲解里面的核心知识点。有读者发私信和留言说，能不能更具体地展开讲解呢。所以我决定把里面所有核心的技术干货写成书，毫无保留地分享给大家。希望本书对大家在生产中实践 MySQL 时有帮助，可以让有一定基础的、有工作经验的运维人员和 DBA 更加深入地了解 MySQL，使用和维护起来更加得心应手，更希望可以让刚踏入数据库领域的读者快速掌握 MySQL 数据库的核心知识体系，给那些想学习 MySQL 数据库的入门者指明一个正确方向，少走一些弯路。让我们一起朝着技术领域金字塔的塔尖大步前行。

如何阅读本书

本书在知识结构上分为 7 部分。

第 1 部分　倔强青铜篇（第 1～8 章）。包括 MySQL 简介、主流分支版本、数据库安装/启动/关闭、权限管理、MySQL 数据库的内存池结构、存储引擎、线程作用、刷新机制、数据库文件、表管理、字符集、统计信息与数据碎片整理的方法。还包括对索引的详细解读，执行计划的分析、压力测试的展现、事务的介绍，以及对隔离级别的深度讲解，并结合锁一起展开学习。

第 2 部分　秩序白银篇（第 9 章）。介绍生产环境中常用的备份方法、逻辑备份和裸文件备份。针对 mysqldump、select...into outfile、mydumper 和 xtrabackup 等备份恢复工具的原理展开详解及实践演练。

第 3 部分　荣耀黄金篇（第 10～11 章）。介绍主从复制的原理，对复制参数进行详细讲解，对半同步复制、多源复制、GTID 复制进行全面解读与实践，对复制数据一致性的校验和复制管理技巧进行介绍，并解读主从复制中的各种报错故障。

第 4 部分　尊贵铂金篇（第 12～15 章）。介绍互联网主流的 MySQL 高可用集群架构，对 MHA、Keepalived、PXC 原理的解析、维护管理与实践，最后介绍 ProxySQL 这个强大的 MySQL 中间代理层的应用。

第 5 部分　永恒钻石篇（第 16～17 章）。介绍 MySQL 5.7 版本的新特性，以及通过硬件、操作系统、数据库、程序设计这四个维度来全面介绍 MySQL 数据库的优化。

第 6 部分　至尊星耀篇（第 18～19 章）。学习部署 Lepus，监控 MySQL 与版本升级。

第 7 部分　最强王者篇 第 20 章。MySQL 面试总结。

致谢

在我从事数据库工作近十年的道路上，我要感谢那些曾经帮助过我的前辈们，当我遇到困难想要放弃这个行业时，是你们耐心的开导与指引，才使我没有失去方向，一直坚持到今天。我也要感谢 51CTO 这个平台，让我可以把自己工作中的经验分享给大家，感谢 51CTO 的运营经理高阳，不厌其烦地帮我整理博客中的一些素材，才能让我的文章以很好的方式呈现给大家。最后还要感谢电子工业出版社的陈晓猛编辑，感谢你的独具慧眼和对我无条件的支持与鼓励，让我可以非常顺利地完成写作。

技术无国界，让我们一起热爱技术、分享知识，打造出属于自己的一片广阔天空。此书献给所有喜欢技术的朋友们！

联系方式

欢迎大家与我互动，联系方式如下。

博客：sumongodb.blog.51cto.com。

张甦

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33679

目　录

第 1 部分　倔强青铜篇

第 2 部分 秩序白银篇

第 3 部分 荣耀黄金篇

第 5 部分 永恒钻石篇

第 6 部分 至尊星耀篇

第 7 部分　最强王者篇

第 1 部分　倔强青铜篇

本部分主要介绍 MySQL 的体系结构，因为要想学好数据库，首先要把基础打牢。跟玩游戏一样，我们先来看看 MySQL 数据库是什么样子的。

第1章 MySQL简介与主流分支版本

本章主要介绍 MySQL 5.6 和 MySQL 5.7 这两个 Oracle 公司开发的代表作品。为了让大家先对 MySQL 有一个基础的认识，我们先来看一看 MySQL 的发展历程。

1.1 MySQL 简介

1999 年至 2000 年，Monty 成立了 MySQL AB 这家公司。2000 年，MySQL 公布了自己的源代码，并采用 GPL（GNU General Public License）许可协议，正式进入开源的世界。2001 年至 2007 年是 MySQL 开源飞速发展的 7 年，尤其是在 2005 年 10 月发布了一个里程碑式的版本 MySQL 5.0。

5.0 版本中加入了存储过程、服务器端游标、触发器、视图、分布式事务（Xa transactions）、查询优化器的显著改进以及其他的一些特性。这也为 MySQL 5.0 之后的版本奠定了迈向高性能数据库的发展基础。2008 年 1 月 16 号 Sun 收购了 MySQL，花了 10 亿美元。之后不久，2009 年 4 月 20 日 Oracle 收购了 Sun 公司。随之 MySQL 就变成了 Oracle 旗下的一个产品，之后就是我们所熟悉的 MySQL 5.5、5.6、5.7 这些版本了。

确切一点说，MySQL 5.5 应该是 Sun 和 Oracle 之间的一个过渡版本，实际上 MySQL 5.6 才是 Oracle 开发的第一个版本，在 MySQL 5.6 的基础上，Oracle 对 MySQL 进行了一次强悍的加工，才有了 MySQL 5.7 的问世。

1.2　MySQL 主流的分支版本

目前业界的 MySQL 主流分支版本有 Oracle 官方版本的 MySQL、Percona Server、MariaDB。接下来看一下各个分支的特点。

1.2.1　官方版本的 MySQL

目前官网最新的 GA 版就是 MySQL 5.7，也是推荐大家在生产环境中使用的一个版本。它无论是在 InnoDB 存储引擎性能和功能上的提升，还是安全性上的加固、复制功能、sys schema 库的增强等都改进得相当出色。目前还在开发的是 MySQL 8.0，这个版本可能是 MySQL 数据库又一个开拓时代的开始。MySQL 不像 Oracle 数据库的版本跳度那么大，Oracle 是 8i、9i、10g、11g、12c 这样的一个版本迭代速度，而 MySQL 在大版本上一直没有什么变化。从 MySQL 5.0、5.1、5.5、5.6 直到目前最成熟的 MySQL 5.7 都基于 5 这个大版本，升级其小版本。所以这次研发的 8.0 版是一个新时代的开始，虽然 MySQL 8.0 在新特性上没有新元素的加入，但是它对 MySQL 的源代码进行了重构，最突出的一点就是对 MySQL Optimizer 优化器的改进，支持隐藏索引等功能。针对优化器的改进是在之前的版本中从来没有触碰过的。而且 MySQL 8.0 为了对优化器做更多的特性支持，还加入了性能直方图这个新的元素，让 MySQL Server 层和存储引擎层配合得更加紧密。MySQL 8.0 之后我们就真的要和 MyISAM 这个存储引擎说再见了，想想这些改变就让人兴奋，就让我们一起期待着它的问世吧。

MySQL 在 5.1 版本被 Oracle 收购之后，发展到今天有了突飞猛进的变化。官方版本的 MySQL 未来有太多值得大家期待的地方。

让我们来看一下 MySQL 5.5、5.6、5.7 的性能对比图，这样可以更加直观地观察到这些年的发展变化。

在 OLTP 只读模式下，MySQL 5.7 比 MySQL 5.6 快近 3 倍的速度，MySQL 5.6 比 MySQL 5.5 快近 1.5 倍的速度，而且 5.7 有将近 100 万的 QPS（每秒的查询量），如图 1-1 所示。

在 OLTP 读写模式下，MySQL 5.7 比 MySQL 5.6 快近 2.5 倍的速度，比 MySQL 5.5 快近 3 倍的速度，而且 5.7 有近 60 万的 QPS，如图 1-2 所示。

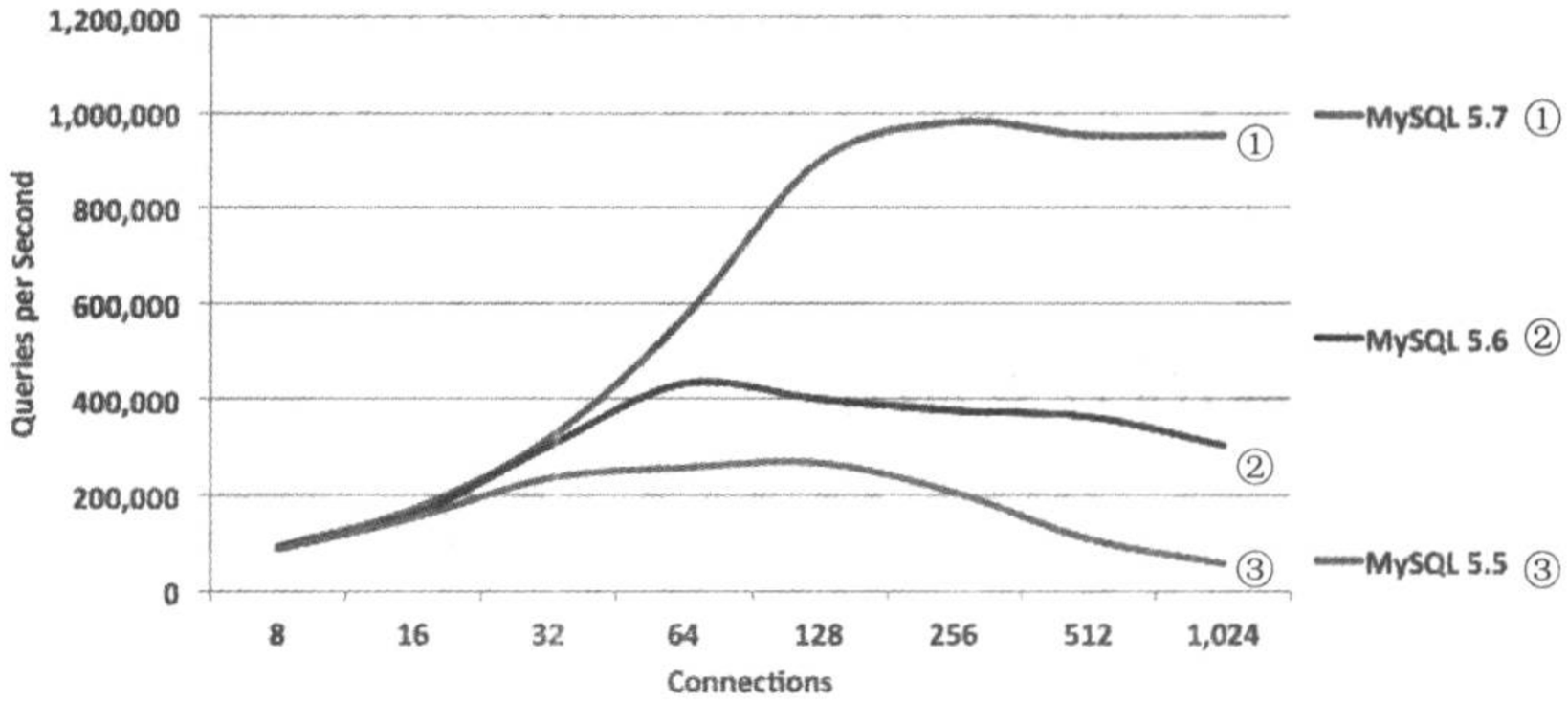


图 1-1　基准测试只读模式下的 QPS 显示图

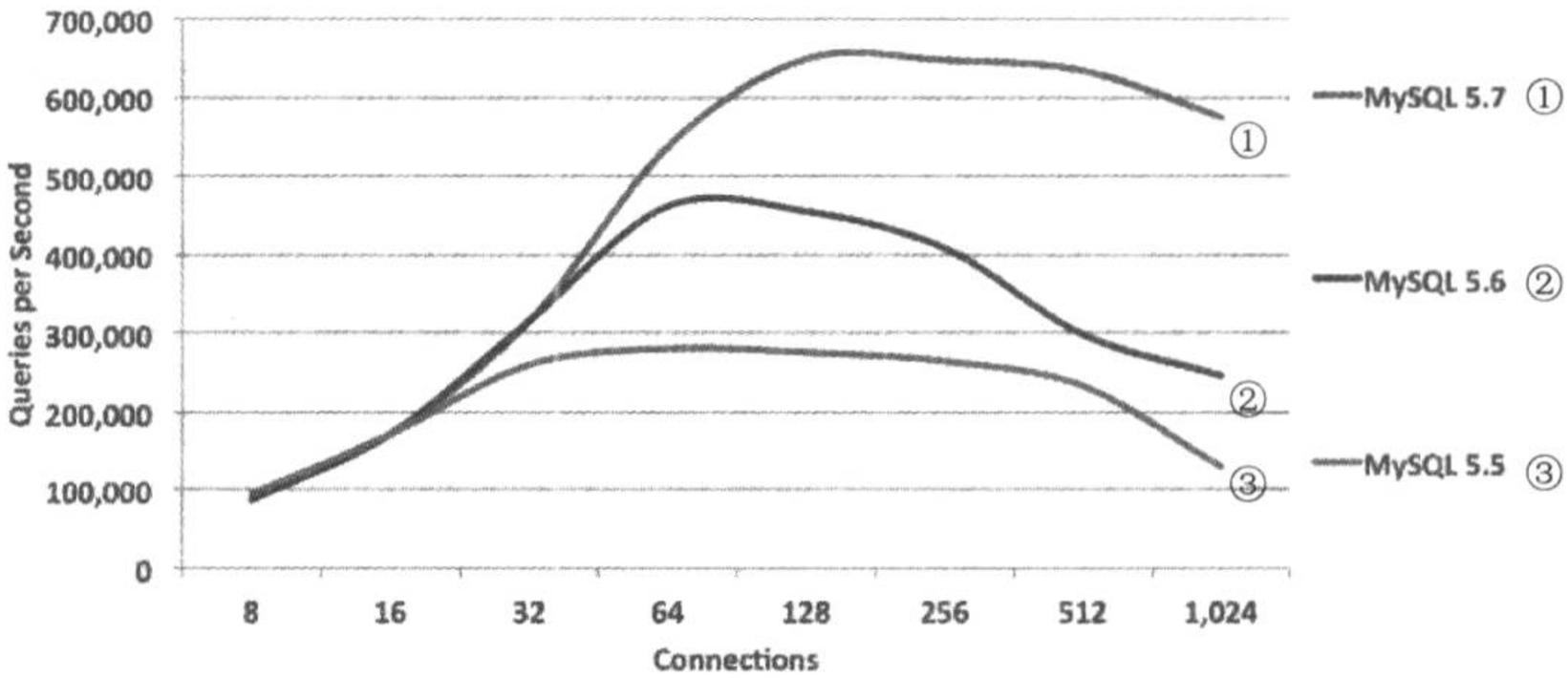


图 1-2　基准测试读写模式下的 QPS 显示图

1.2.2　Percona Server

Percona Server 是 MySQL 重要的分支之一，它基于 InnoDB 存储引擎的基础上，提升了性能和易管理性，最后形成了增强版的 XtraDB 引擎，可以用来更好地发挥服务器硬件上的性能。所以 Percona Server 也可以称为增强的 MySQL 与开源的插件（plugin）的结合。由于官方版本的 MySQL 在一些特性的使用上有一定的局限性，需要收费。所以 Percona Server 就有了一定的市场占有比例，也比较受大家的欢迎。像一些常用的工具包 xtrabackup、percona-toolkit 等，在生产环境中是 DBA 的必备武器。还有像 XtraDB-Cluster 这种支持多点写入的强同步高可用集群架构，真正实现实时同步的过程，解决了 MySQL 主从复制之间经常出现并让人头疼的延迟问题。而且 Percona 还收购了 TokuDB 公司，TokuDB 存储引擎非常优秀，淘宝网、阿里云上大量在使用这款存储引擎。它支持数据压缩，支持 hot scheme modification，它的高扩展性和优秀的查询插入性能都是我们喜欢它的地方。

感兴趣的读者可以去官网了解更详细的特点。

链接地址：https://www.percona.com/software/mysql-database/percona-tokudb。

1.2.3　MariaDB

Mariadb 是由 MySQL 创始人 Monty 创建的，是一款高度兼容的 MySQL 产品，主要由开源社区维护，采用 GPL 授权许可。Oracle 把 MySQL 收购之后，为避免 MySQL 在开源粒度上的下降，MariaDB 由此而生。它不仅仅是 MySQL 的一个替代品，还创新与提高了 MySQL 原有的技术。既包含了 Percona 的 XtraDB 存储引擎，还包含 TokuDB 存储引擎、Spider 水平分片存储引擎等多种存储引擎，并且还有一些复制功能上的新特性，比如基于表的并行复制、Multi-source Replication 多源复制、Galera Cluster 集群。还有比较有意思的一点就是 MariaDB 有一套 Java 的管理系统，可以通过投票机制来决定哪些特性和参数是我们需要的。

第2章

MySQL 数据库的安装、启动和关闭

2.1 下载 MySQL 二进制软件包

本章主要介绍 MySQL 二进制软件包的安装/启动/关闭过程。也许有人要问为什么要选择二进制的安装方式呢？其实答案很简单，官方版本中已经把所有功能都配置好了，我们可以很方便地拿来使用。官网 MySQL 有四个版本，分别为 GA 版、DMR 版、RC 版和 Beta 版。一般情况下，生产环境或者测试环境要选择 GA 版（常规可用的版本，经过 bug 修复测试过）。大家可以去 MySQL 官网下载软件包，官网链接地址为 www.mysql.com。

这里重点强调下面一点。

点击 MySQL Community Server 进入下载软件包的系统平台选择页面，我们这里选择 Linux-Generic，版本选择 X86-64bit，如图 2-1 所示。

软件包下载完成之后，我们可以进入安装流程了。这里最好先对下载下来的软件包进行一次 MD5 校验，保证下载过程中没有任何问题，软件包可以正常使用。

```
[root@node3 local]# md5sum mysql-5.6.16-linux-glibc2.5-x86_64.tar.gz
41b4533f4fcec2d0132794f26f378f2a  mysql-5.6.16-linux-glibc2.5-x86_64.tar.gz
```

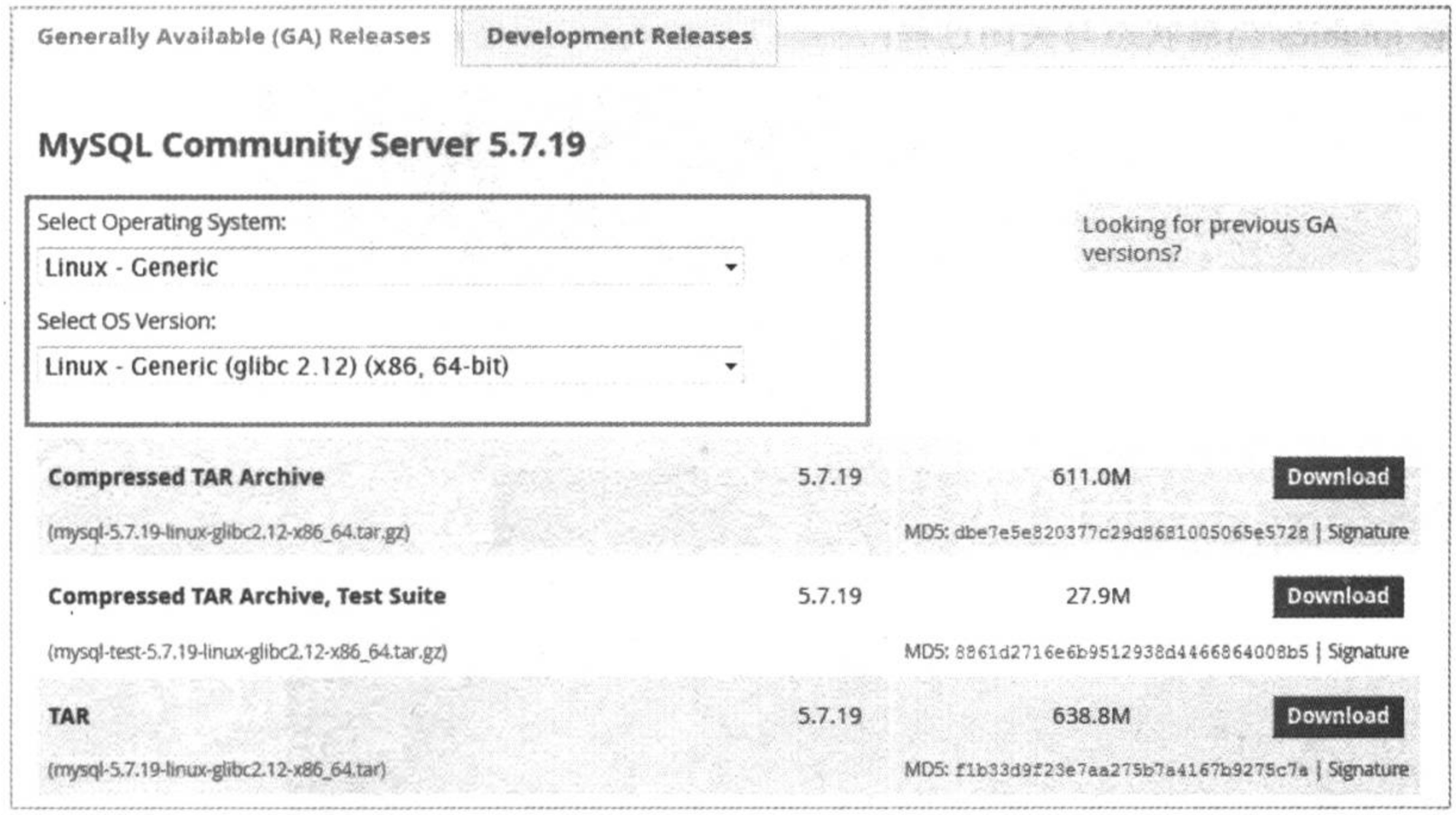


图 2-1　软件下载界面

这里学习 MySQL 5.6 和 MySQL 5.7 版本的安装过程，是因为 MySQL 5.7 之前版本在数据库初始化过程中需要借助于/usr/local/mysql/scripts/ mysql_install_db 命令，但在 MySQL 5.7 之后版本 mysql_install_db 被废弃了。

2.2　安装前系统环境检测

先来看一下 MySQL 5.6 版本的安装，这里我们用到的软件包是 MySQL 5.6.16，不管是 MySQL 哪个版本的安装，为了可以很顺利地把 MySQL 数据库安装好，做好后期数据库的优化工作，在前期进行 Linux 的系统检测是十分有必要的。

这里列举了一些需要注意的事项。

1. SELinux 和系统防火墙 iptables 需要关闭

要把 SELINUX 设置成 disable，设置完成之后需要重启系统：

```
[root@node3 ~]# cat /etc/sysconfig/selinux
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - SELinux is fully disabled.
SELINUX=disabled
# SELINUXTYPE= type of policy in use. Possible values are:
#       targeted - Only targeted network daemons are protected.
#       strict - Full SELinux protection.
SELINUXTYPE=targeted
[root@node3 ~]#
```

查看 iptables 当前状态与关闭过程：

```
[root@node3 mysql]# chkconfig --list|grep iptables
iptables        0:off   1:off   2:on    3:on    4:on    5:on    6:off
[root@node3 mysql]# chkconfig  iptables off
[root@node3 mysql]# chkconfig --list|grep iptables
iptables        0:off   1:off   2:off   3:off   4:off   5:off   6:off
```

2. I/O 调度系统默认是 cfq 模式，这里强烈建议使用 deadline 模式

查看 I/O 调度文件：

```
[root@node3 ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

修改 I/O 调度器，需要在/etc/grub.conf 中加入 elevator=deadline，保证永久生效。

```
[root@node3 ~]# cat /etc/grub.conf
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/sda3
#          initrd /initrd-[generic-]version.img
#boot=/dev/sda
default=0
timeout=5

elevator=deadline
```

3. swap 分区的设置

swappiness 值的大小对如何使用 swap 分区有着很大的影响。

它有 0 和 100 两个极限值，0 代表最大限度地使用物理内存，然后才使用 swap 分区，这种行为有可能导致系统内存溢出，出现 OOM 的错误，从而导致 MySQL 被意外 kill 掉，所以需要谨慎设置。100 则是积极地使用 swap 分区，并且把内存上面的数据及时搬到 swap 分区里（不建议）。这里建议大家不分配 swap，或者分配 4GB 的空间就足够了。

如何查看 swappiness 文件呢？

```
[root@node3 mysql]# cat /proc/sys/vm/swappiness
60
[root@node3 mysql]# sysctl -a|grep swap
vm.swap_token_timeout = 300     0
vm.swappiness = 60
```

想要修改 swappiness 的值，编辑/etc/sysctl.conf，加入 vm.swappiness 的值即可。

4. 文件系统的选择

这里建议大家使用 xfs 文件系统，相比 ext4，它更方便管理，支持动态扩容，删除文件也很方便。

5. 操作系统的限制

先来查看一些当前操作系统的限制情况。

使用 ulimit –a 查看：

```
[root@node3 ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 14865
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 14865
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

这里标记了两个最为重要的参数，一个叫 open files，另一个叫 max user processes。Open files 如果设置不合理，而当前服务器的连接过多或者表过多时，就有可能会出现打不开表或者访问不了表的现象。默认情况下，Linux 的最大句柄数为 1024 个，表示单个进程最多可以访问 1024 个文件句柄。如要超过默认值，就会出现文件句柄超限的错误“too many open files”。

max user processes 参数的用途是，有时候我们可能会跑多实例，但是发现创建不了新的连接，报出“resource temporarily unavailable”的错误，表示没有足够的资源。

为了防止以上两种报错情况，我们可以修改系统的软硬限制。编辑/etc/security/limits.conf，加入限制的相关内容。记得更改完内容之后，需要重启操作系统才能生效。

```
#*               soft    core            0
#*               hard    rss             10000
#@student        hard    nproc           20
#@faculty        soft    nproc           20
#@faculty        hard    nproc           50
#ftp             hard    nproc           0
#@student        -       maxlogins       4

*    soft    nproc    65535
*    hard    nproc    65535
*    soft    nofile    65535
*    hard    nofile    65535
```

6. numa 需要关闭

简单来讲关闭 numa 功能，可以更好地分配内存，不需要采用 swap 的方式来获取内存。因为有经验的系统管理员和 DBA 都知道使用 swap 导致的数据库性能下降有多么的恶心。关闭方

式也分在 BIOS、操作系统中关闭，或者是在数据库启动过程中关闭。

```
    numa  --interleave=all  /usr/local/mysql/bin/mysqld_safe -defaults-file=
/etc/my.cnf  &
```

2.3 MySQL 5.6 版本的安装过程

检查完操作系统环境后，我们就要进行安装 MySQL 的过程演练了。MySQL 的安装总结为“三部曲、一步走”的方式。

1. 第一部曲

创建 MySQL 用户，指定 MySQL 所在的用户组，命令如下：

```
shell>groupadd  mysql
shell>useradd  -g mysql  mysql  -s /sbin/nologin
```

软件包的家目录（basedir）统一规范放在/usr/local/下面：

```
shell>cd /usr/local/
```

需要解压 MySQL 软件包，命令如下：

```
shell>tar -zxvf mysql-5.6.16-linux-glibc2.5-x86_64.tar.gz
```

然后做个软链，方便日后升级：

```
shell>ln -s mysql-5.6.16-linux-glibc2.5-x86_64  mysql
```

别忘了给 MySQL 目录授权：

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

2. 第二部曲

创建 MySQL 数据库的数据目录（datadir），这里可以选择创建在/data/mysql 下面，命令如下：

```
shell>mkdir  -p /data/mysql
```

当然也不能忘记给数据目录授权：

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

3. 第三部曲

由于是二进制的安装方式，这里的数据库配置文件需要自己配置好，到此完成最后一部曲（这里暂时先不需要了解每个参数的含义，后面的章节中会详细讲解）。

```
shell>vim /etc/my.cnf
 [client]
port = 3306
socket = /tmp/mysql.sock
default-character-set=utf8
[mysql]
default-character-set=utf8
[mysqld]
port = 3306
socket = /tmp/mysql.sock
basedir = /usr/local/mysql
datadir = /data/mysql
open_files_limit    = 65535
back_log = 103
max_connections = 512
max_connect_errors = 100000
table_open_cache = 512
external-locking = FALSE
max_allowed_packet = 128M
sort_buffer_size = 2M
join_buffer_size = 2M
thread_cache_size = 51
query_cache_size = 32M
tmp_table_size = 96M
max_heap_table_size = 96M
slow_query_log = 1
slow_query_log_file = /data/mysql/slow.log
log-error = /data/mysql/error.log
long_query_time = 0.5
server-id = 1323306
```

```
log-bin = /data/mysql/mysql-bin
sync_binlog = 1
binlog_cache_size = 4M
max_binlog_cache_size = 128M
max_binlog_size = 1024M
expire_logs_days = 7
key_buffer_size = 32M
read_buffer_size = 1M
read_rnd_buffer_size = 16M
bulk_insert_buffer_size = 64M
character-set-server=utf8
default-storage-engine=InnoDB
binlog_format=row
#gtid_mode=on
#log_slave_updates=1
#enforce_gtid_consistency=1
interactive_timeout=300
wait_timeout=300
transaction_isolation = REPEATABLE-READ
innodb_buffer_pool_size = 1434M
innodb_data_file_path = ibdata1:1024M:autoextend
innodb_flush_log_at_trx_commit = 1
innodb_log_buffer_size = 16M
innodb_log_file_size = 256M
innodb_log_files_in_group = 2
innodb_max_dirty_pages_pct = 50
innodb_file_per_table = 1
innodb_locks_unsafe_for_binlog = 0
[mysqldump]
quick
max_allowed_packet = 32M
```

4. 一步走

接下来完成启动前最后一步走的操作，需要初始化数据库，命令如下：

```
shell>cd /usr/local/mysql/scripts
shell>./mysql_install_db --basedir=/usr/local/mysql --datadir=/data/mysql
--defaults-file=/etc/my.cnf --user=mysql
```

当出现有两个“OK”的时候，证明初始化数据库成功了。

```
Installing MySQL system tables...2017-09-13 10:49:06 0 [Warning] TIMESTAMP with implicit DEFAULT value is depre
ase use --explicit_defaults_for_timestamp server option (see documentation for more details).
OK

Filling help tables...2017-09-13 10:49:50 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Plea
xplicit_defaults_for_timestamp server option (see documentation for more details).
OK
```

初始化完成之后，我们可以启动 MySQL 数据库了，命令如下：

```
shell>cd /usr/local/mysql/bin/
shell>./mysqld_safe  --defaults-file=/etc/my.cnf &
```

MySQL 的启动过程：

```
[root@node3 mysql]# /usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf &
[1] 3171
[root@node3 mysql]# 170913 10:54:00 mysqld_safe Logging to '/data/mysql/error.log'.
170913 10:54:00 mysqld_safe Starting mysqld daemon with databases from /data/mysql
```

查看 MySQL 进程，验证是否启动成功：

```
[root@node3 mysql]# ps -ef|grep mysql
root      3171  3003  0 10:53 pts/1    00:00:00 /bin/sh /usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/m
y.cnf
mysql     3822  3171  1 10:54 pts/1    00:00:00 /usr/local/mysql/bin/mysqld --defaults-file=/etc/my.cnf --based
ir=/usr/local/mysql --datadir=/data/mysql --plugin-dir=/usr/local/mysql/lib/plugin --user=mysql --log-error=/da
ta/mysql/error.log --open-files-limit=3072 --pid-file=/data/mysql/node3.pid --socket=/tmp/mysql.sock --port=330
6
root      3847  3003  0 10:54 pts/1    00:00:00 grep mysql
```

以上就是 MySQL 5.6.16 版本的安装启动过程。

这里还需要强调以下几点。

（1）MySQL 读取配置文件的顺序依次是/etc/my.cnf→/etc/mysql/my.cnf→/usr/local/mysql/etc/my.cnf~/.my.cnf。

```
[root@node3 ~]# mysql --help|grep my.cnf
                      order of preference, my.cnf, $MYSQL_TCP_PORT,
/etc/my.cnf /etc/mysql/my.cnf /usr/local/mysql/etc/my.cnf ~/.my.cnf
```

mysqld 在启动过程中可以自己启动任何位置的配置文件，如果需要指定固定配置文件，就需要使用--defaults-file。

（2）mysqld 启动选项有三个，defaults-file——使用指定位置的配置文件；--defaults-extra-file——除了读取默认的配置文件，还读取额外的配置文件；--no-defaults——忽略所有的配置文件。如果一次性指定多个配置文件，则以最后一次读取的为准。这里使用的是指定位置的配置文件--defaults-file=/etc/my.cnf。

2.4 创建密码

安装完 MySQL 之后，进入数据库的方式是无密码进入的，为了保证数据库的安全性，我们需要为数据库 root 用户创建密码（这里的数据库密码在全书中统一为 root123）。

命令如下：

```
mysql> use mysql;
mysql>  update user set password=password('root123') where user='root';
mysql> flush privileges;
```

低于 MySQL 5.7 版本的数据库需要进行安全加固，只保留数据库中用户为 root、host 为 localhost 的账号。语句如下：

```
mysql> delete from user where user!='root' or host!='localhost';
```

2.5 关闭 MySQL 数据库

关闭 MySQL 数据库有正常关闭和非正常关闭两种。

先看正常关闭方式，命令如下：

```
shell>cd /usr/local/mysql/bin
shell>./mysqladmin -uroot -proot123 shutdown
```

非正常关闭就需要 kill 掉 MySQL 的进程了。

2.6 基础数据库的名称

安装完 MySQL 5.6.16 之后，可以通过 show databases 来查看当前数据库有哪些：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

MySQL 5.7 安装完成之后会再多一个 sys 库。这个库在后期讲新特性时再详细介绍。这里

主要介绍 information_schema 库和 MySQL 库。

information_schema 数据库是在安装 MySQL 过程中的初始化阶段自动生成的。它提供了访问数据库中元数据的所有信息，可以理解为数据字典，它就是 MySQL 的信息库，里面保存着关于 MySQL 服务器所维护的所有其他数据库的信息，如数据库名、数据库里面的表、表的数据类型和访问权限等。但该库是只读库，只能进行 select 操作。我们在 information_schema 下用得比较多的表有：

- tables（记录所有表的基本信息，访问该表可收集表的统计信息）。
- PROCESSLIST（查看当前数据库的连接）。
- GLOBAL_STATUS（查看数据库运行的各种状态值）。
- GLOBAL_VARIABLES（查看数据库中的参数）。
- PARTITIONS（数据库中表分区的情况）。

INNODB_LOCKS、INNODB_TRX、INNODB_LOCK_WAITS 这三张表用来监控数据库中锁的情况。

MySQL 库也是在初始化过程中自动创建的。我们在该库下用得最多的一张表就是 user，用于管理数据库中的用户权限信息。

2.7　MySQL 5.7 版本的安装

我们再来看一下 MySQL 5.7 的安装方式，它唯一区别于 MySQL 5.6 的安装过程就是在初始化数据库那一步了。之前说过 MySQL 5.7 已经废弃了 mysql_install_db 这个初始化命令。

我们安装的是 MySQL 5.7.14 版本。安装过程如下：

```
shell>groupadd  mysql
shell>useradd  -g mysql  mysql  -s /sbin/nologin
shell>cd /usr/local/
shell> tar -zxvf mysql-5.7.14-linux-glibc2.5-x86_64.tar.gz
shell>ln -s mysql-5.7.14-linux-glibc2.5-x86_64  mysql
shell>mkdir  -p /data/mysql
shell>vim /etc/my.cnf
[client]
port = 3306
socket = /tmp/mysql.sock
[mysql]
prompt="\u@db \R:\m:\s [\d]> "
```

```
no-auto-rehash
[mysqld]
user = mysql
port = 3306
basedir = /usr/local/mysql
datadir = /data/mysql/
socket = /tmp/mysql.sock
character-set-server = utf8mb4
skip_name_resolve = 1
open_files_limit    = 65535
back_log = 1024
max_connections = 512
max_connect_errors = 1000000
table_open_cache = 1024
table_definition_cache = 1024
table_open_cache_instances = 64
thread_stack = 512K
external-locking = FALSE
max_allowed_packet = 32M
sort_buffer_size = 4M
join_buffer_size = 4M
thread_cache_size = 768
query_cache_size = 0
query_cache_type = 0
interactive_timeout = 600
wait_timeout = 600
tmp_table_size = 32M
max_heap_table_size = 32M
slow_query_log = 1
slow_query_log_file = /data/mysql/slow.log
log-error = /data/mysql/error.log
long_query_time = 0.5
server-id = 3306100
log-bin = /data/mysql/mysql-binlog
sync_binlog = 1
binlog_cache_size = 4M
max_binlog_cache_size = 1G
max_binlog_size = 1G
```

```
expire_logs_days = 7
master_info_repository = TABLE
relay_log_info_repository = TABLE
gtid_mode = on
enforce_gtid_consistency = 1
log_slave_updates
binlog_format = row
relay_log_recovery = 1
relay-log-purge = 1
key_buffer_size = 32M
read_buffer_size = 8M
read_rnd_buffer_size = 4M
bulk_insert_buffer_size = 64M

lock_wait_timeout = 3600
explicit_defaults_for_timestamp = 1
innodb_thread_concurrency = 0
innodb_sync_spin_loops = 100
innodb_spin_wait_delay = 30
transaction_isolation = REPEATABLE-READ
innodb_buffer_pool_size = 1024M
innodb_buffer_pool_instances = 8
innodb_buffer_pool_load_at_startup = 1
innodb_buffer_pool_dump_at_shutdown = 1
innodb_data_file_path = ibdata1:1G:autoextend
innodb_flush_log_at_trx_commit = 1
innodb_log_buffer_size = 32M
innodb_log_file_size = 2G
innodb_log_files_in_group = 2
innodb_io_capacity = 2000
innodb_io_capacity_max = 4000
innodb_flush_neighbors = 0
innodb_write_io_threads = 8
innodb_read_io_threads = 8
innodb_purge_threads = 4
innodb_page_cleaners = 4
innodb_open_files = 65535
innodb_max_dirty_pages_pct = 50
```

```
innodb_flush_method = O_DIRECT
innodb_lru_scan_depth = 4000
innodb_checksum_algorithm = crc32
innodb_lock_wait_timeout = 10
innodb_rollback_on_timeout = 1
innodb_print_all_deadlocks = 1
innodb_file_per_table = 1
innodb_online_alter_log_max_size = 4G
internal_tmp_disk_storage_engine = InnoDB
innodb_stats_on_metadata = 0
innodb_status_file = 1
innodb_status_output = 0
innodb_status_output_locks = 0
#performance_schema
performance_schema = 1
performance_schema_instrument = '%=on'
#innodb monitor
innodb_monitor_enable="module_innodb"
innodb_monitor_enable="module_server"
innodb_monitor_enable="module_dml"
innodb_monitor_enable="module_ddl"
innodb_monitor_enable="module_trx"
innodb_monitor_enable="module_os"
innodb_monitor_enable="module_purge"
innodb_monitor_enable="module_log"
innodb_monitor_enable="module_lock"
innodb_monitor_enable="module_buffer"
innodb_monitor_enable="module_index"
innodb_monitor_enable="module_ibuf_system"
innodb_monitor_enable="module_buffer_page"
innodb_monitor_enable="module_adaptive_hash"
[mysqldump]
quick
max_allowed_packet = 32M
shell>chown mysql:mysql -R /usr/local/mysql
shell>chown mysql:mysql -R /data/mysql
```

这里用 **mysqld** 命令初始化数据库：

```
shell>cd /usr/local/mysql/bin/
shell>  ./mysqld  --defaults-file=/etc/my.cnf  --basedir=/usr/local/mysql
--datadir=/data/mysql/  --user=mysql -initialize
```

> 注：如果在初始化过程中加上--initialize 参数，表示会生成一个临时的数据库初始化密码，记录在 log-error（错误日志）里面，如果加上--initialize-insecure 参数，代表无密码进入。建议使用生成初始化密码的方式。

启动数据库的过程：

```
shell>cd /usr/local/mysql/bin/
shell>./mysqld_safe --defaults-file=/etc/my.cnf &
```

数据库启动成功之后，进入数据库的初始化密码会在/data/mysql/error.log 下面：

```
cat /data/mysql/error.log |grep password
```

```
[root@node3 mysql]# cat /data/mysql/error.log |grep password
2017-09-13T04:19:46.101301Z 1 [Note] A temporary password is generated for root@localhost: ppA(/UeaB3js
```

使用初始化密码进入数据库之后，需要修改数据库 root 密码，设置为永不过期：

```
mysql>SET PASSWORD = 'root123';
mysql>ALTER USER 'root'@'localhost' PASSWORD EXPIRE NEVER;
mysql>flush privileges;
```

以上就是 MySQL 5.7 版本的安装启动方式。

2.8　MySQL 数据库 root 密码丢失的问题

对于 DBA 来说，丢失超管用户 root 的密码是致命的，我们有什么好的办法可以解决这样的问题呢？可以通过添加--skip-grant-tables 参数来跳过权限表。

忘记 root 密码，进不去数据库：

```
[root@node3 bin]# mysql -uroot -p
Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

这时就需要强制停库，先查看 MySQL 进程号：

```
[root@node3 bin]#
[root@node3 bin]# ps -ef|grep mysql
root      3807  3142  0 12:22 pts/0    00:00:00 /bin/sh ./mysqld_safe --defaults-file=/etc/my.c
mysql     5097  3807  2 12:22 pts/0    00:00:00 /usr/local/mysql/bin/mysqld --defaults-file=/et
cal/mysql --datadir=/data/mysql/ --plugin-dir=/usr/local/mysql/lib/plugin --user=mysql --log-er
--open-files-limit=65535 --pid-file=/data/mysql//db.pid --socket=/tmp/mysql.sock --port=3306
root      5137  3142  0 12:22 pts/0    00:00:00 grep mysql
```

Kill 掉 MySQL 进程，命令如下：

```
Kill -9 5097 3807
```

然后加跳过权限表参数，重启数据库。这样即使不输入密码，也可以进入数据库。

```
[root@node3 bin]# ./mysqld_safe --defaults-file=/etc/my.cnf --skip-grant-tables &
[1] 5149
[root@node3 bin]# 2017-09-13T04:26:31.357606Z mysqld_safe Logging to '/data/mysql/error.log'.
2017-09-13T04:26:31.383690Z mysqld_safe Starting mysqld daemon with databases from /data/mysql/

[root@node3 bin]#
[root@node3 bin]#
[root@node3 bin]# mysql
Welcome to the MySQL monitor.  Commands end with ; or°\g.
Your MySQL connection id is 2
Server version: 5.7.14-log MySQL Community Server (GPL)

Copyright (c) 2000, 2016, 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.

root@db 12:26:  [(none)]>
```

给 root 用户设置新的密码并刷新权限，MySQL 5.7 之后，MySQL 库下的 password 字段用 authentication_string 字段代替。

```
root@db 12:30:  [mysql]> use mysql;
Database changed
root@db 12:31:  [mysql]> update user set authentication_string=password('root') where user='root'
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 1

root@db 12:31:  [mysql]> flush privileges;
Query OK, 0 rows affected (0.01 sec)
```

设置完成之后，重启数据库。注意这时就不用再加--skip-grant-tables 参数了，正常启动服务，输入新的密码可以正常进入数据库了。

```
[root@node3 bin]# ./mysqld_safe --defaults-file=/etc/my.cnf &
[1] 6518
[root@node3 bin]# 2017-09-13T04:34:21.828093Z mysqld_safe Logging to '/data/mysql/error.log'.
2017-09-13T04:34:21.855364Z mysqld_safe Starting mysqld daemon with databases from /data/mysql/

[root@node3 bin]#
[root@node3 bin]# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.14-log

Copyright (c) 2000, 2016, 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.

root@db 12:34:  [(none)]>
```

2.9　MySQL 数据库的连接方式

在 Linux 平台环境下主要有两种连接方式，一种是 TCP/IP 连接方式，另一种就是 Socket 连接。在 Windows 平台下，有 name pipe 和 share memory（不考虑）两种。

TCP/IP 连接是网络中用得最多的一种方式。一般情况下客户端在一台服务器上，而 MySQL 实例在另一台服务器上，两台机器通过一个 TCP/IP 网络连接。

```
mysql -u username -p password  -P port  -h IP
```

通过 TCP/IP 连接 MySQL 实例时，MySQL 会先检查一张权限表，用来判断发起请求的客户端 IP 是否允许连接到 MySQL 实例。该表就是 MySQL 库下面的 user 表。

UNIX Socket 连接方式其实不是一个网络协议，所以只能在 MySQL 客户端和数据库实例在同一台服务器上的情况下使用。可以在配置文件中指定套接字文件的路径，如 socket=/tmp/mysql.sock。

```
mysql -u username  -p password  -S /tmp/mysql.sock
```

> 注：经常使用连接 MySQL 的客户端工具有 sqlyog、navicat。

2.10　用户权限管理

MySQL 数据库的账户安全性对于 DBA 来说是至关重要的。工作中的数据库环境一般分为

测试环境和生产环境。我们经常会给开发人员或者其他人分配权限。所以在任何权限的申请流程上一定要设置规范，让所有人完全按照规则来做事。

MySQL 数据库中的用户一般分为超管权限用户 root 和那些由 root 用户创建的普通用户，它们的权限由 root 用户分配。root 用户避免作为 Web 连接用，超管权限的用户（包括 root 用户和 all privileges 权限的用户）只能归 DBA 管理。查看 MySQL 5.6 数据库中用户的情况：

```
mysql> select user,host,password from user;
+------+-----------+-------------------------------------------+
| user | host      | password                                  |
+------+-----------+-------------------------------------------+
| root | localhost | *FAAFFE644E901CFAFAEC7562415E5FAEC243B8B2 |
+------+-----------+-------------------------------------------+
1 row in set (0.00 sec)
```

查看 MySQL 5.7 的用户情况：

```
root@db 12:41:  [mysql]> use mysql;
Database changed
root@db 12:41:  [mysql]> select user,host,authentication_string from user;
+-----------+-----------+-------------------------------------------+
| user      | host      | authentication_string                     |
+-----------+-----------+-------------------------------------------+
| root      | localhost | *FAAFFE644E901CFAFAEC7562415E5FAEC243B8B2 |
| mysql.sys | localhost | *THISISNOTAVALIDPASSWORDTHATCANBEUSEDHERE |
+-----------+-----------+-------------------------------------------+
2 rows in set (0.00 sec)
```

在创建用户时，我们最好保证专库专账号，不要一个账号管理多个库，库特别多的小公司根据情况特殊对待管理。

创建用户的语法：

```
create user 用户名@主机 ip  identified by '密码';
```

> 注：为了保证数据库的安全性，主机 IP 避免使用%，可以分配一个 IP 地址的网段。

给开发人员的权限分配规则，可以分配只读权限和读写权限的用户，不要给用户超管权限。

> 注：这里不要给用户建表、改表(create、alter)等权限。

只读权限就是只能查询，不能进行 DML 操作。

读写权限包含 insert、update、delete 和 select。

例如，想要为 erp 库创建一个只读用户和读写用户。

只读用户：

```
create user 'erp_read'@'192.168.56.%' identified by 'erp123';
grant select on erp.* to 'erp_read'@'192.168.56.%' identified by 'erp123';
```

切记不要忘记刷新权限：执行 flush privileges。

读写用户：

```
    create user 'erp_user'@'192.168.56.%' identified by 'erp456';
    grant select,insert,update,delete on erp.* to 'erp_user'@'192.168.56.%'
identified by 'erp456';
    flush privileges;
```

第3章 MySQL体系结构与存储引擎

3.1 MySQL体系结构

如果把MySQL数据库比喻成一条龙的话，那么体系结构就是它的龙头部分。学习MySQL数据库，又好比盖房子，如果想把房子盖得特别高，地基一定要稳，基础得牢固。所以学习MySQL数据库就需要先了解它的体系结构，这是学好MySQL数据库的前提。

MySQL的体系结构可以分为两层，MySQL Server层和存储引擎层。在MySQL Server层中又包括连接层和SQL层，如图3-1所示。

应用程序通过接口（如ODBC、JDBC）来连接MySQL。最先连接处理的是连接层，连接层包括通信协议、线程处理、用户名密码认证三个部分。通信协议负责检测客户端版本是否兼容MySQL服务端。线程处理是指每一个连接请求都会分配一个对应的线程，相当于一条SQL对应一个线程，一个线程对应一个逻辑CPU，并会在多个逻辑CPU之间进行切换。用户名密码认证验证创建的账号和密码，以及host主机授权是否可以连接到MySQL服务器。

SQL层包含权限判断、查询缓存、解析器、预处理、查询优化器、缓存和执行计划。

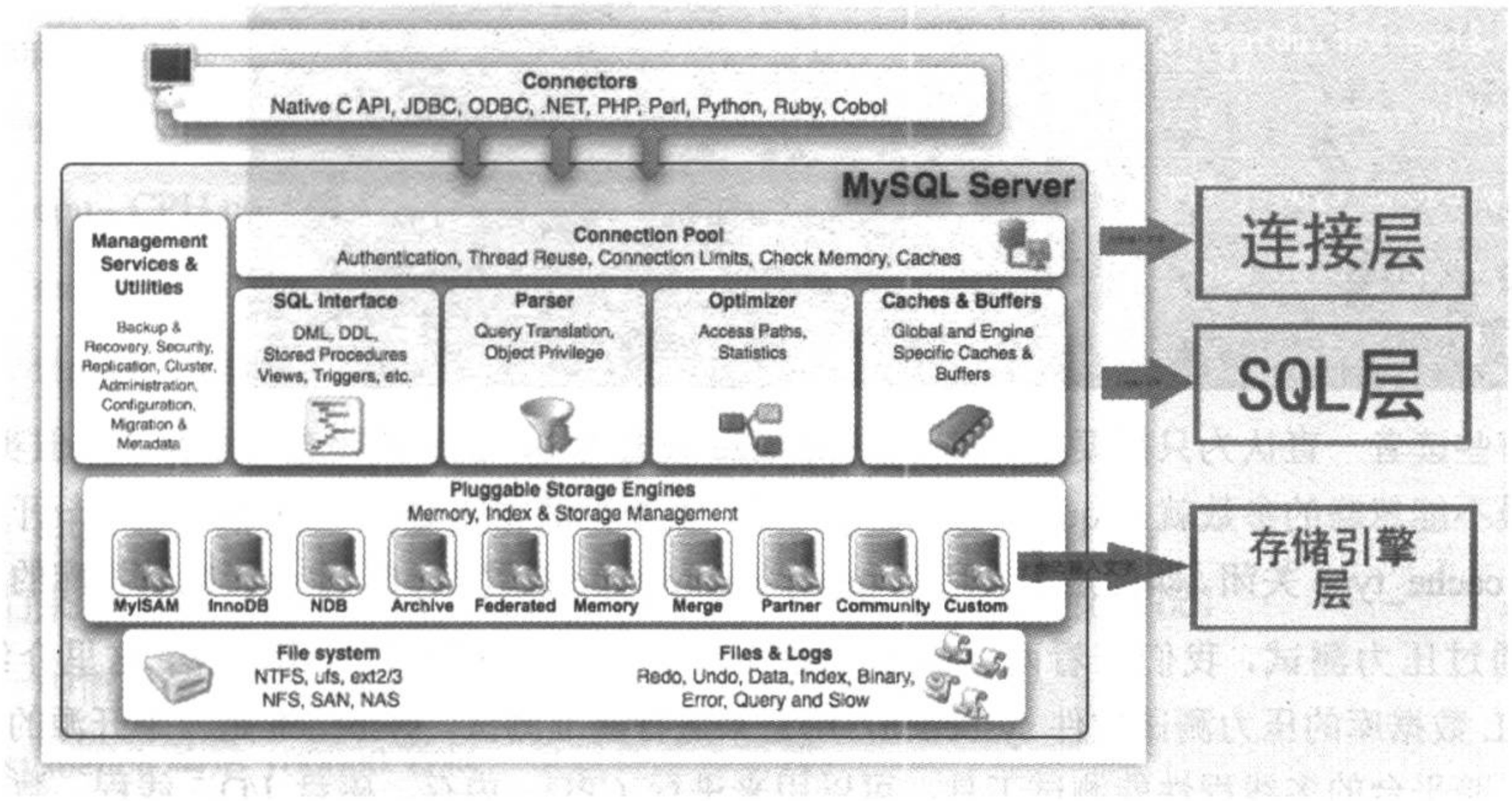


图 3-1　MySQL 体系结构

权限判断可以审核用户有没有访问某个库、某个表，或者表里某行的权限。查询缓存通过 Query Cache 进行操作，如果数据在 Query Cache 中，则直接返回结果给客户端。查询解析器针对 SQL 语句进行解析，判断语法是否正确。预处理器对解析器无法解析的语义进行处理。优化器对 SQL 进行改写和相应的优化，并生成最优的执行计划，就可以调用程序的 API 接口，通过存储引擎层访问数据。

存储引擎层也是 MySQL 数据库区别于其他数据库最核心的一点。

3.2　Query Cache 详解

Query Cache 在生产中建议关闭，因为它只能缓存静态数据信息，一旦数据发生变化，经常读写，Query Cache 就成了“鸡肋”。一般像数据仓库之类的可能会考虑开启 Query Cache。这里再提及一句，MySQL 5.6 之前版本的 Query Cache 默认是开启的，5.6 之后默认是关闭的。

如何彻底关闭 Query Cache 是我们需要关注的。

首先涉及 query_cache 的两个核心参数：

```
mysql> show variables like "%query_cache_size%";
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_size | 0     |
+------------------+-------+
1 row in set (0.00 sec)
```

```
mysql> show variables like "%query_cache_type%";
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | OFF   |
+------------------+-------+
1 row in set (0.00 sec)

mysql>
```

有些读者一直认为只需要把 query_cache_size 设置为 0，就算关闭查询缓存了。但实际上，我们最不能忽略的参数就是 query_cache_type 了。要想彻底关闭 Query Cache，必须一开始就把 query_cache_type 关闭。即便是启动后把 query_cache_type 设置为 off，也会影响数据库的 TPS。

通过压力测试，我们来看两个核心参数在不同设置下 TPS 的对比情况。这里介绍一款 MySQL 数据库的压力测试软件 sysbench，用它来进行基准测试。sysbench 是一个开源的、模块化的、跨平台的多线程性能测试工具，可以用来进行 CPU、内存、磁盘 I/O、线程、数据库的性能测试。目前支持的数据库有 MySQL、Oracle 和 PostgreSQL。

软件包下载地址：https://dev.mysql.com/downloads/benchmarks.html。

sysbench 的安装过程：

```
tar -zvxf sysbench-0.4.8.tar.gz
cd sysbench-0.4.8
./configure  --with-mysql-includes=/usr/local/mysql/include  --with-mysql-libs=
/usr/local/mysql/lib
make && make install
```

sysbench 的主要参数详解如下。

- --num-threads=N：创建测试线程的数目。
- --max-requests=N：请求的最大数目。默认为 10000，0 代表不限制。
- --max-time=N：最大执行时间，单位是 s。默认是 0，不限制。
- --thread-stack-size=SIZE：每个线程的堆栈大小。默认是 32KB。
- --init-rng=[on|off]：在测试开始时是否初始化随机数发生器。默认是 off。
- --test=STRING：指定测试项目名称。
- --debug=[on|off]：是否显示更多的调试信息。默认是 off。
- --validate=[on|off]：在可能情况下执行验证检查。默认是 off。
- --help=[on|off]：帮助信息。

Compiled-in tests：

- fileio - File I/O test——测试 I/O。
- cpu - CPU performance test——测试 CPU。
- memory - Memory functions speed test——测试内存。
- threads - Threads subsystem performance test——测试线程。
- mutex - Mutex performance test——测试互斥性能。
- oltp - OLTP test——测试 OLTP。

我们来看一下在不同 query_cache 参数设置下，TPS 的性能对比情况。

第一种情况是 query_cache_type=0、query_cache_size=0。

用 sysbench 构造 100000 数据，也就是准备阶段：

```
[root@node3 sysbench]# ./sysbench --test=oltp --mysql-table-engine=innodb  --mysql-
-driver=mysql --mysql-socket=/tmp/mysql.sock --mysql-user=root --mysql-host=localho
sysbench v0.4.8:  multi-threaded system evaluation benchmark

Creating table 'sbtest'...
Creating 100000 records in table 'sbtest'...
```

然后进入测试阶段，由于是测试环境，这里设置的线程数量不多，为 16 个，即 num-threads=16。

测试出每秒处理的事务数为 289.24。

```
./sysbench --test=oltp --mysql-table-engine=innodb  --mysql-db=test
--oltp-table-size=100000 --db-driver=mysql --num-threads=16
--mysql-socket=/tmp/mysql.sock --mysql-user=root --mysql-host=localhost
--mysql-password=root123 run
```

```
OLTP test statistics:
    queries performed:
        read:                            140000
        write:                           50000
        other:                           20000
        total:                           210000
    transactions:                        10000  (289.24 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 190000 (5495.54 per sec.)
    other operations:                    20000  (578.48 per sec.)
```

第二种情况是 query_cache_type=1、query_cache_size=0。

测试出每秒处理的事务数为 265.09：

```
OLTP test statistics:
    queries performed:
        read:                            140000
        write:                           50000
        other:                           20000
        total:                           210000
    transactions:                        10000  (265.09 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 190000 (5036.80 per sec.)
    other operations:                    20000  (530.19 per sec.)
```

第三种情况是 query_cache_size=0，把 query_cache_type 从 1 改成 0，测试出每秒处理的事务数为 286.52：

```
OLTP test statistics:
    queries performed:
        read:                            140000
        write:                           50000
        other:                           20000
        total:                           210000
    transactions:                        10000  (286.52 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 190000 (5443.85 per sec.)
    other operations:                    20000  (573.04 per sec.)
```

三种情况下：TPS 的值分别为 289.24、265.09、286.52。可见设置正确的 query_cache 的关闭方式有多么重要。

3.3 存储引擎

MySQL 数据库及其分支版本主要的存储引擎有 InnoDB、MyISAM、Memory、blackhole、TokuDB 和 MariaDB columnstore。

先来看一下主要的存储引擎的特性对比，如表 3-1 所示。

表 3-1 各大存储引擎的特点

存储引擎名称	特　　点	应 用 场 景
InnoDB	支持事务、行锁，支持 MVCC 多版本并发控制，并发性高	应用于 OLTP 业务系统
MyISAM	不支持事务、表锁，MySQL 8.0 之后被废弃了，并发很低，资源利用率也很低	应用于 OLAP 业务系统，建议在生产环境尽量少使用 MyISAM 存储引擎
Memory	表中的数据都在内存中存放，不落地。支持 Hash 和 Btree 索引 数据安全性不高，读取速度快	应用于对数据安全性要求不高的环境下

续表

存储引擎名称	特　　点	应 用 场 景
TokuDB	归 Percona 公司所有。支持事务，支持压缩功能，高速写入功能（比 InnoDB 快 9 倍），在线 Online DDL，不产生索引碎片	应用于海量数据的存储场景下
MariaDB columnstore	列式存储引擎，高压缩功能	数据仓库，OLAP 业务系统
Blackhole	并不存储数据，数据写入时只写 binlog	blackhole 常用来做 binlog 转储或测试

InnoDB 和 MyISAM 是最主流的两个存储引擎，现在数据库版本默认的存储引擎是 InnoDB，并且 MySQL 8.0 宣布 InnoDB 存储数据字典，MyISAM 彻底从 MySQL 数据库中剥离开，被废弃了。但等新版本彻底上线前，还是有不少互联网公司依然在使用 MyISAM 存储引擎。

这里建议大家把线上 MyISAM 的存储引擎表全部转化成 InnoDB 表存储。我们先来比较一下两者之间的主要区别，如表 3-2 所示。

表 3-2　InnoDB 与 MyISAM 对比

区　　别	InnoDB	MyISAM
事务的支持	支持事务	不支持事务
锁粒度	行锁	表锁
并发性	高并发	低并发
构成结构和缓存机制	数据和索引文件都存储在.Ibd 文件里，并且都缓存在内存里	数据文件的扩展名为.MYD (MYData) 索引文件的扩展名是.MYI (MYIndex) 只缓存索引文件，不缓存数据文件
select count(*)	需要扫描全表，统计所有行数	只需要从计数器中读出保存好的行数即可

可以看出 InnoDB 存储引擎的优势很明显。

MySQL 被 Oracle 收购之后，也针对存储引擎层做了相应的改进与优化，Server 层没有太大的变动，主要优化的核心就是 InnoDB 存储引擎，所以我们今后的重心就放在 InnoDB 上面，研究它的体系结构。

3.4　InnoDB 体系结构

3.4.1　数据库和数据库实例

先要弄清楚两个概念，一个是数据库，另一个是数据库实例。MySQL 数据库是一个单进程

多线程模型的数据库。数据库实例就是进程加内存的组合，它就好比我们杯子里面装的水，在操作数据时，是在数据库实例里面进行的，也就是内存；而杯子实际上就是数据库，真实存放水数据的地方。而在内存中的数据，早晚有一天也会根据刷新机制刷到磁盘上，那么谁来帮助数据库刷新呢？就是那些线程。这就是一条数据从内存到磁盘的过程，也就形成了如图 3-2 所示的 InnoDB 的体系结构图。

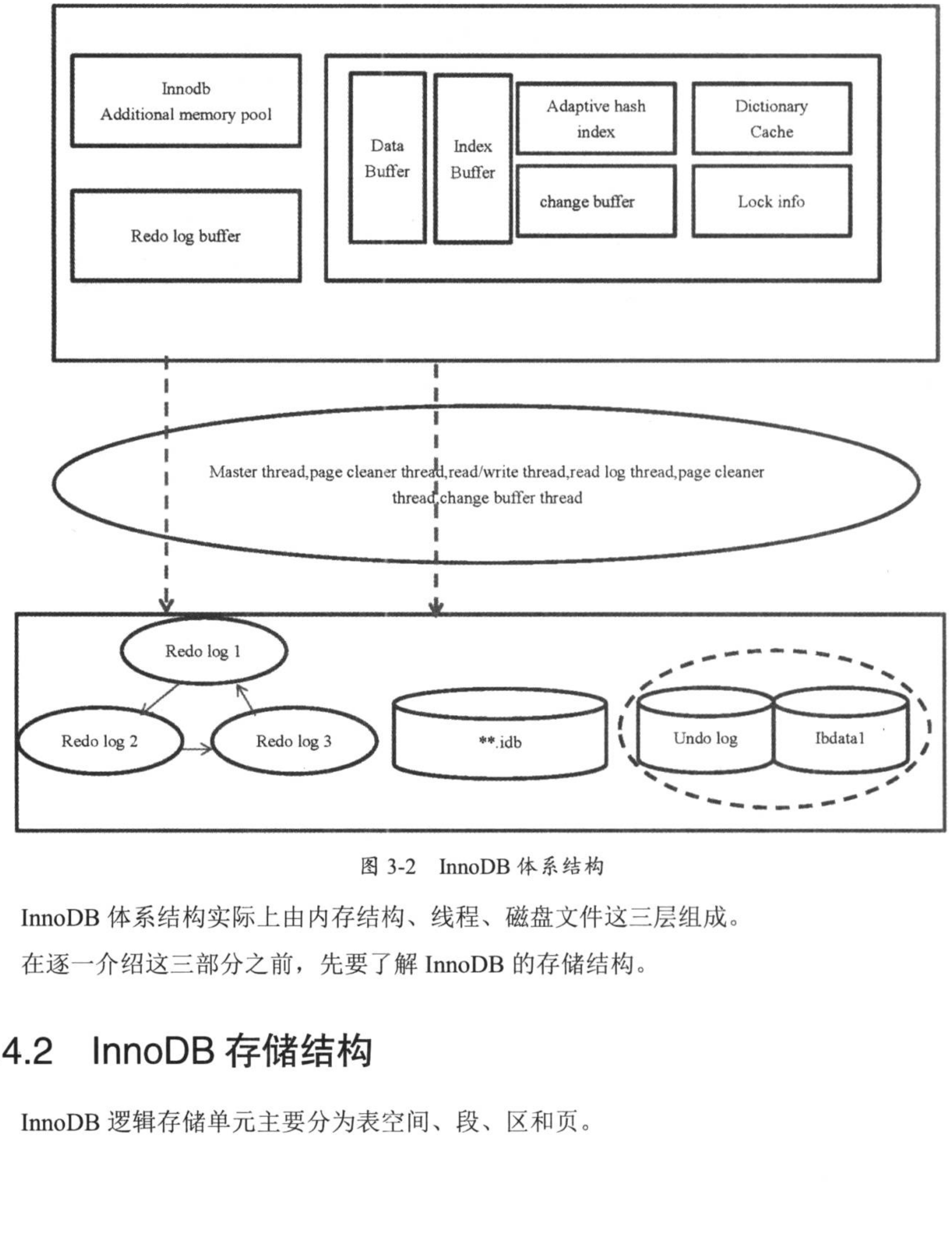


图 3-2　InnoDB 体系结构

InnoDB 体系结构实际上由内存结构、线程、磁盘文件这三层组成。

在逐一介绍这三部分之前，先要了解 InnoDB 的存储结构。

3.4.2　InnoDB 存储结构

InnoDB 逻辑存储单元主要分为表空间、段、区和页。

层级关系为 tablespace→segment→extent（64 个 page，1MB）→page，如图 3-3 所示。

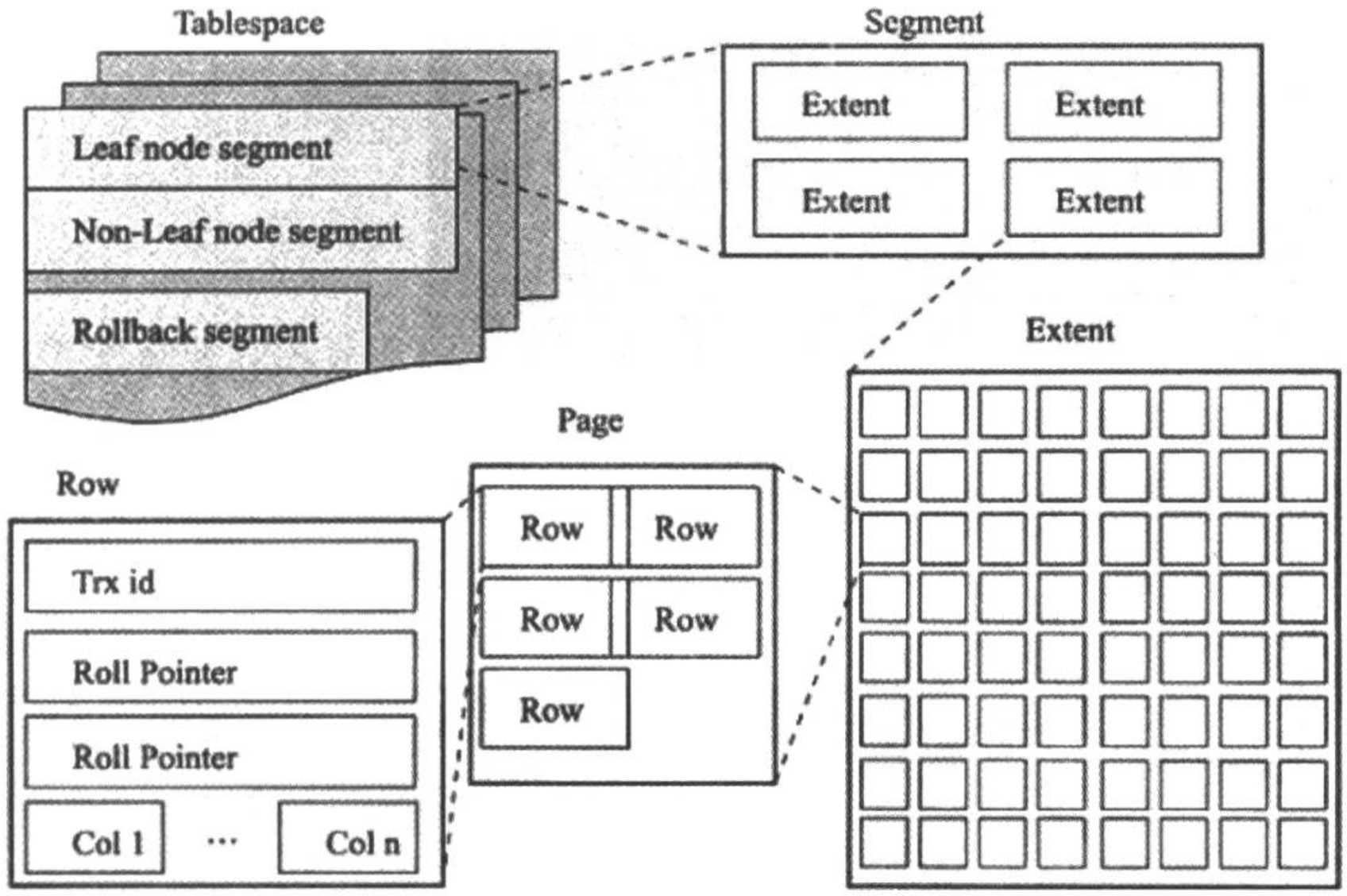


图 3-3　InnoDB 存储结构

1. 表空间

InnoDB 存储引擎表中所有数据都是存储在表空间中的，表空间又分为系统表空间，它以 ibdata1 来命名，在安装数据库初始化数据时就是系统在创建一个 ibdata1 的表空间文件，它会存储所有数据的信息以及回滚段（undo）的信息。在 MySQL 5.6 之后，undo 表空间可以通过参数单独设置存储位置了，可从 ibdata1 中独立出来。Innodb_data_file_path 负责定义系统表空间的路径、初始化大小、自动扩展策略。数据库默认的自动扩展大小是 64MB。

```
mysql> show variables like '%auto%';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| auto_increment_increment    | 1     |
| auto_increment_offset       | 1     |
| autocommit                  | ON    |
| automatic_sp_privileges     | ON    |
| innodb_autoextend_increment | 64    |
| innodb_autoinc_lock_mode    | 1     |
| innodb_stats_auto_recalc    | ON    |
| sql_auto_is_null            | OFF   |
+-----------------------------+-------+
8 rows in set (0.00 sec)
```

数据库默认的 ibdata1 的大小是 10MB，这里建议不要使用 10MB 的默认大小，在遇到高并发事务时，会受到不小的影响。建议把 ibdata1 的初始数值大小调整为 1GB。

```
mysql> show variables like '%innodb_data%';
+-----------------------+-------------------------+
| Variable_name         | Value                   |
+-----------------------+-------------------------+
| innodb_data_file_path | ibdata1:1024M:autoextend |
| innodb_data_home_dir  |                         |
+-----------------------+-------------------------+
2 rows in set (0.00 sec)
```

除了系统表空间，还有独立表空间，设置参数 innodb_file_per_table=1 即可。目前 MySQL 版本中，默认使用的都是独立表空间文件，就是每个表就有自己的表空间文件，而不用存储在 ibdata1 中。独立表空间文件存储对应表的 B+树数据、索引和插入缓冲等信息，其余信息还是存储在默认表空间中，如图 3-4 所示。

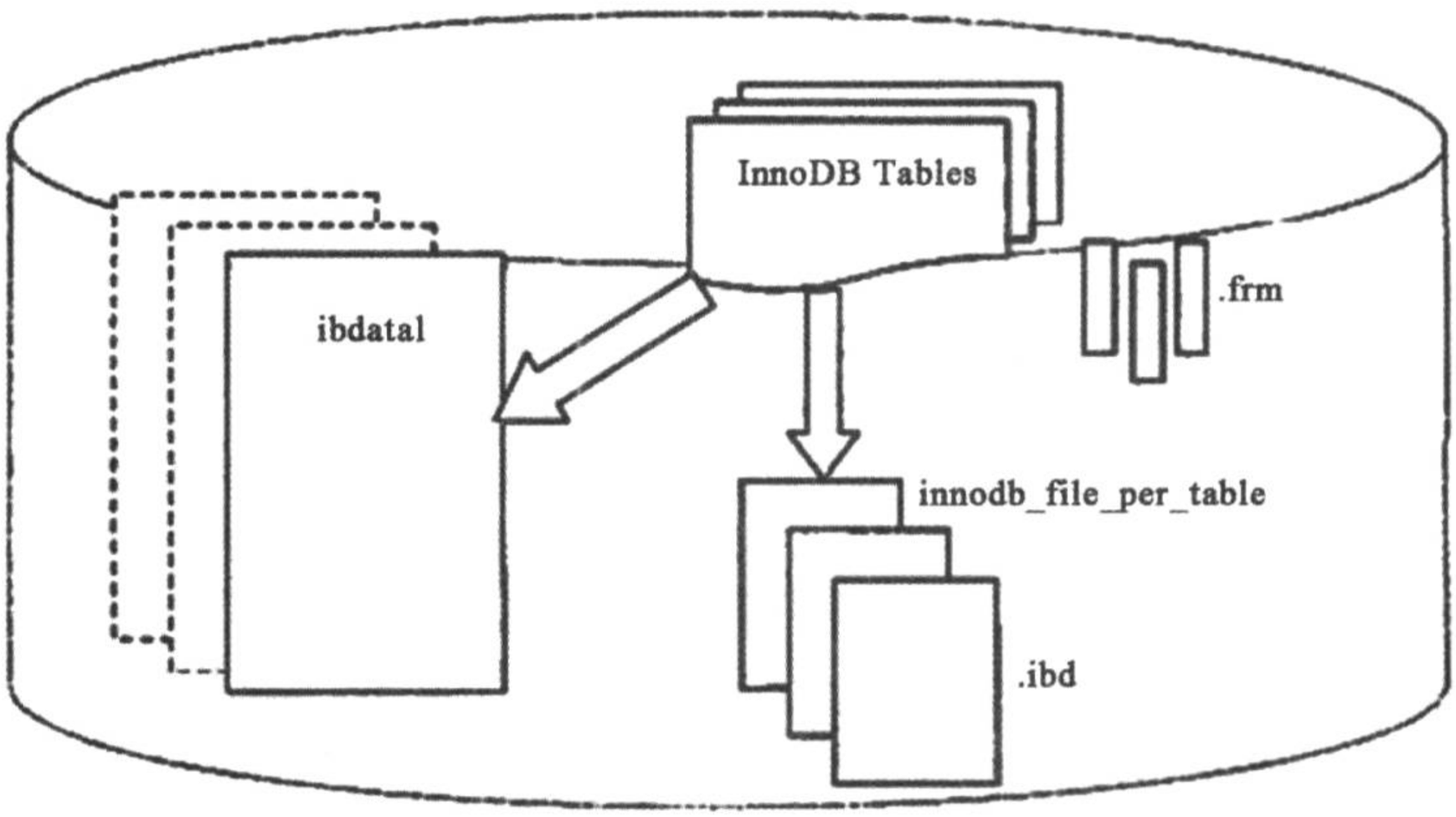


图 3-4　独立表空间

共享表空间和独立表空间有什么区别呢？

独立表空间的每个表都有自己的表空间，并且可以实现表空间的转移，回收表空间也很方便，使用 alter table table_name engine=innodb 或者 pt-online_schema_change 命令即可。但有一个不好的地方在于每个表文件都有.frm 和.ibd 文件两个文件描述符，如果单表增长过快就容易出现性能问题。

共享表空间的数据和文件放在一起方便管理。

但是共享表空间无法在线回收空间，共享表空间想要回收，需要将全部 InnoDB 表中的数据备份、删除原表，然后再把数据导回到与原表结构一样的新表中。特别是统计分析、日志类

系统不太适合用共享表空间。

综合考虑，独立表空间的效率、性能比共享表空间会高一点，目前默认使用的就是独立表空间存储方式。

MySQL 5.7 之后又多临时表空间（temporary tablespace）和通用表空间（general tablespace）的概念。

a. 临时表空间

MySQL 5.7 把临时表的数据从系统表空间中抽离出来，形成自己的独立表空间参数 innodb_temp_data_file_path，并且把临时表的相关检索信息保存在系统信息表的 information_schema 库下的 innodb_temp_table_info 表中。但目前还不能定义临时表空间文件的存放路径，只能与 innodb_data_home_dir 一致。

独立表空间文件名为 ibtmp1，默认大小为 12MB。

```
root@db 14:03:  [(none)]> show variables like '%temp%';
+------------------------------+------------------------+
| Variable_name                | Value                  |
+------------------------------+------------------------+
| avoid_temporal_upgrade       | OFF                    |
| innodb_temp_data_file_path   | ibtmp1:12M:autoextend  |
| show_old_temporals           | OFF                    |
+------------------------------+------------------------+
3 rows in set (0.04 sec)
```

b. 通用表空间

多个表放在同一个表空间中，可以根据活跃度来划分表，存放在不同的磁盘上，可以减少 metadata 的存储开销。但目前在生产中很少使用。

2. 段

表空间是由段组成的，也可以把一个表理解为一个段。通常有数据段、回滚段、索引段等。每个段由 *N* 个区和 32 个零散的页组成，段空间扩展是以区为单位进行扩展的。通常情况下，创建一个索引的同时就会创建两个段，分别为非叶子节点和叶子节点段。那一个表有几个段呢？答案是 4 个，是索引个数的 2 倍。

3. 区

区是由连续的页组成的，是物理上连续分配的一段空间，每个区的大小固定是 1MB。

4. 页

InnoDB 的最小物理存储分配单位是 page，有数据页回滚页等。一般情况下，一个区由 64 个连续的页组成，页默认大小是 16KB。

Variable_name	Value
innodb_log_compressed_pages	ON
innodb_max_dirty_pages_pct	50
innodb_max_dirty_pages_pct_lwm	0
innodb_page_size	16384
innodb_stats_persistent_sample_pages	20
innodb_stats_sample_pages	8
innodb_stats_transient_sample_pages	8
large_page_size	0
large_pages	OFF

区就等于 64×16KB=1MB。但 MySQL 数据库从 5.6 开始可以自定义调低 page 的大小，可以从默认的 16KB 调整为 8KB 或 4KB。而 MySQL 5.7 开始可以调高 page 的大小，可以从默认的 16KB 调整为 32KB 或者 64KB。一般情况下，一个 page 页会默认预留 1/16 的空间用于更新数据。真正使用的是 15/16 的空间。一个页最少可以存两行数据，虚拟最小行（infimum records）和虚拟最大行（infimum records），用来限定行记录的范围，以此来保证 B+tree 节点是双向链表结构，如图 3-5 所示。

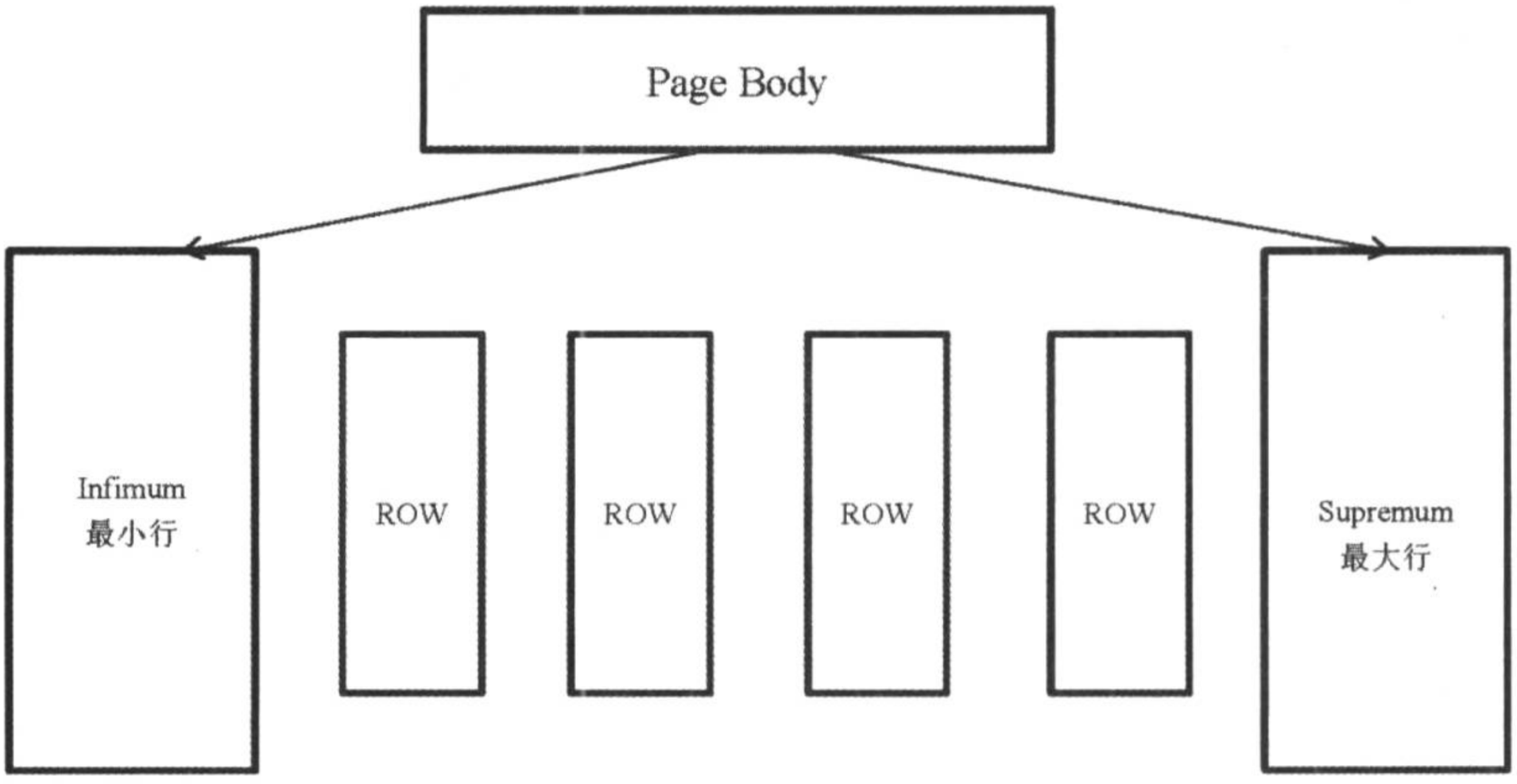


图 3-5　页的结构

整体页的结构如图 3-6 所示。

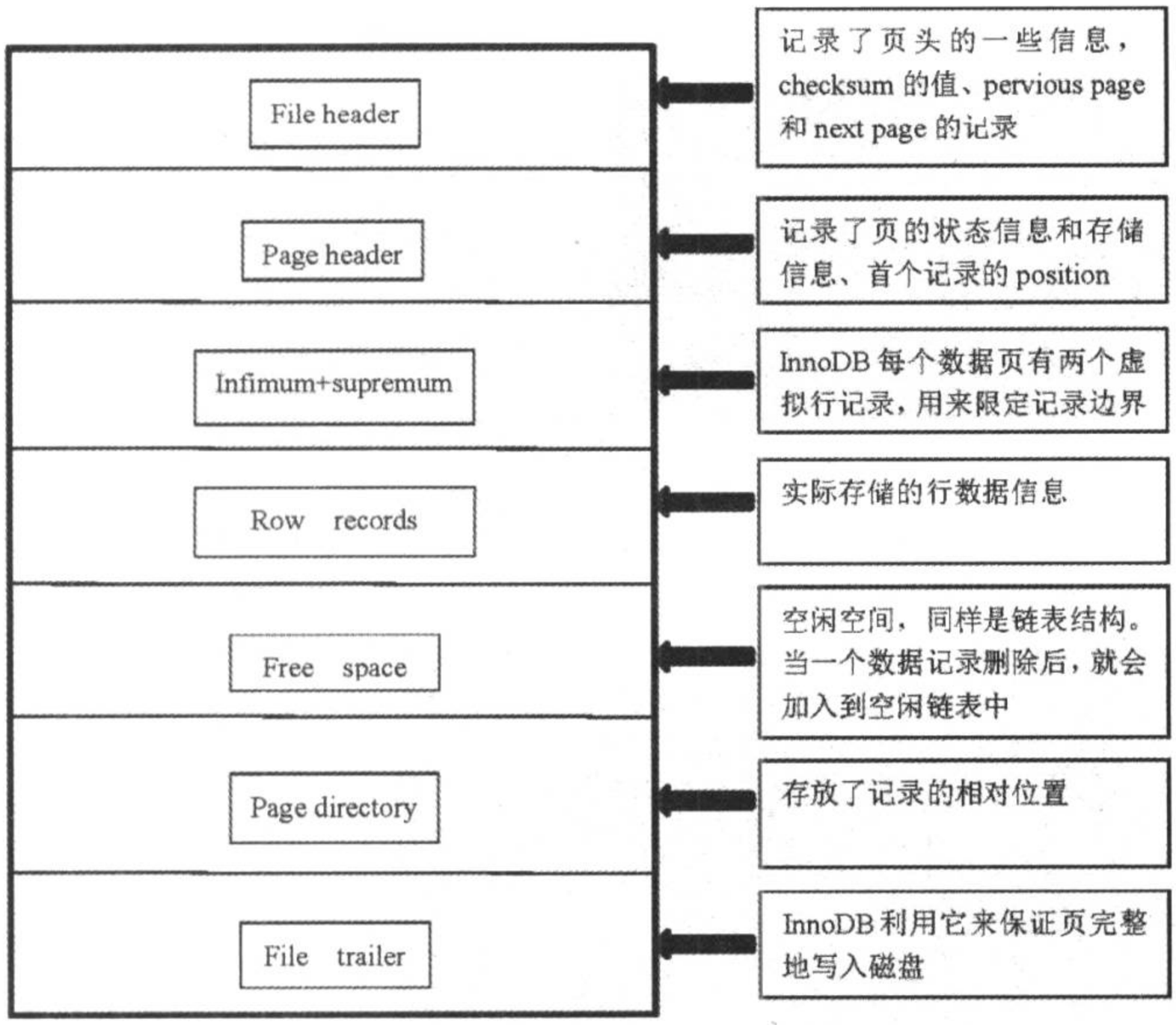


图 3-6　整体页的结构

5. 行

页里面又记录着行记录的信息，InnoDB 存储引擎是面向列的，也就是数据是按照行存储的。行记录数据又是按照行格式进行存放的。InnoDB 存储引擎有两种文件格式，一种叫 Antelope，另外一种叫 Barracuda。在 Antelope 文件格式下，有 compact 和 redundant 两种行记录格式。而在 Barracuda 文件格式下，有 compressed 和 dynamic 两种行记录格式。row_format 这一列对应的就是 t 这张表使用的行格式存储类型。

```
mysql> show table status like '%sbtest%'\G;
*************************** 1. row ***************************
           Name: sbtest
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 98715
 Avg_row_length: 228
    Data_length: 22593536
Max_data_length: 0
   Index_length: 2342912
      Data_free: 4194304
 Auto_increment: 100033
    Create_time: 2017-09-13 12:49:47
    Update_time: NULL
     Check_time: NULL
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
```

文件格式的查看方式：

```
mysql> show variables like '%innodb_file%';
+---------------------------+----------+
| Variable_name             | Value    |
+---------------------------+----------+
| innodb_file_format        | Antelope |
| innodb_file_format_check  | ON       |
| innodb_file_format_max    | Antelope |
| innodb_file_per_table     | ON       |
+---------------------------+----------+
4 rows in set (0.00 sec)
```

行记录格式可以分为四种：Compact、dynamic、redundant、compressed。目前最多使用的就是 Compact 行记录格式，新版本 MySQL 5.7 默认使用 dynamic 行记录格式和 Barracuda 文件格式：

```
root@db 13:29:  [(none)]> show variables like '%row_format%';
+---------------------------+---------+
| Variable_name             | Value   |
+---------------------------+---------+
| innodb_default_row_format | dynamic |
+---------------------------+---------+
1 row in set (0.00 sec)

root@db 13:32:  [(none)]> show variables like '%innodb_file_format%';
+---------------------------+-----------+
| Variable_name             | Value     |
+---------------------------+-----------+
| innodb_file_format        | Barracuda |
| innodb_file_format_check  | ON        |
| innodb_file_format_max    | Barracuda |
+---------------------------+-----------+
3 rows in set (0.00 sec)
```

innodb_default_row_format 参数是 MySQL 5.7 之后新增的。compact 是目前最多使用的一种，而 dynamic 是新版本默认的行记录格式，它们有什么不同呢？生产环境中，我们又应该选择哪一种呢？要解决这样的疑问，首先要先弄明白什么是行溢出。行溢出简单来讲就是需要存储的数据在当前存储页面之外，拆分到多个页进行存储。针对大数据类型 text 或者 blob 存储在其字段中的数据，dynamic 实际采用的数据都存放在溢出的页中（off-page），而数据页只存前 20 个字节的指针。在 compact 行格式下，溢出的列只存放 768 个前缀字节。dynamic 这种行格式模式，针对溢出列所在的新页利用率会更高。所以目前生产环境中建议尽量选择 dynamic 行格式进行存储。redundant 是最早的行记录格式，相比 compact 要消耗更多的存储空间，不建议使用。Compressed 是压缩行格式，是对数据和索引页进行压缩。但只是针对物理存储层面上的压缩，而在内存中是不压缩的.当数据调用到内存中就涉及转换，会有很多无用的 CPU 消耗，而且效率也很低。压缩比例也不高，大概只接近 1/2 的比例。压缩带来的负面影响也很大，数据库的 TPS（每秒中事务的处理）会下降，这就很影响现有的线上业务了，也不建议使用。

3.4.3 内存结构

实际上 MySQL 内存的组成和 Oracle 类似，也可以分为 SGA（系统全局区）和 PGA（程序

缓存区）。数据库内存参数分配，这里是 MySQL 5.7 的配置参数，通过 show variables like '%buffer%'查看：

Variable_name	Value
bulk_insert_buffer_size	67108864
innodb_buffer_pool_chunk_size	134217728
innodb_buffer_pool_dump_at_shutdown	ON
innodb_buffer_pool_dump_now	OFF
innodb_buffer_pool_dump_pct	25
innodb_buffer_pool_filename	ib_buffer_pool
innodb_buffer_pool_instances	8
innodb_buffer_pool_load_abort	OFF
innodb_buffer_pool_load_at_startup	ON
innodb_buffer_pool_load_now	OFF
innodb_buffer_pool_size	1073741824
innodb_change_buffer_max_size	25
innodb_change_buffering	all
innodb_log_buffer_size	33554432
innodb_sort_buffer_size	1048576
join_buffer_size	4194304
key_buffer_size	33554432
myisam_sort_buffer_size	8388608
net_buffer_length	16384
preload_buffer_size	32768
read_buffer_size	8388608
read_rnd_buffer_size	4194304
sort_buffer_size	4194304
sql_buffer_result	OFF

先来介绍系统全局区（SGA）由哪些主要内存区域组成。

（1）innodb_buffer_pool。

用途：用来缓存 InnoDB 表的数据、索引、插入缓冲、数据字典等信息。

（2）innodb_log_buffer。

用途：事务在内存中的缓冲，即 redo log buffer 的大小。

（3）Query Cache。

用途：高速查询缓存，前面已经介绍过，在生产环境中建议关闭。

（4）key_buffer_size。

用途：只用于 MyISAM 存储引擎表，缓存 MyISAM 存储。

引擎表的索引文件（区别于 innodb_buffer_pool 数据和索引都缓存）。

（5）innodb_additional_mem_pool_size。

用途：用来保存数据字典信息和其他内部数据结构的内存池的大小。MySQL 5.7.4 中该参数被移除了。

再来介绍程序缓存区（PGA）中包含的内存区域。

（1）sort_buffer_size。

用途：主要用于 SQL 语句在内存中的临时排序。

（2）join_buffer_size。

用途：表连接使用，用于 BKA。MySQL 5.6 之后开始支持。后面的章节中会介绍。

（3）read_buffer_size。

用途：表顺序扫描的缓存，只能应用于 MyISAM 表存储引擎。

（4）read_rnd_buffer_size。

用途：MySQL 随机读缓冲区大小，用于做 mrr，mrr 是 MySQL 5.6 之后才有的特性。后面章节中会介绍。

再介绍两个特殊的。

（1）tmp_table_size。

用途：SQL 语句在排序或者分组时没有用到索引，就会使用临时表空间。

（2）max_heap_table_size。

用途：管理 heap、memory 存储引擎表。

参数查询：

```
root@db 13:34:  [(none)]> show variables like '%heap%';
+---------------------+----------+
| Variable_name       | Value    |
+---------------------+----------+
| max_heap_table_size | 33554432 |
+---------------------+----------+
1 row in set (0.01 sec)

root@db 13:34:  [(none)]> show variables like '%tmp_table_size%';
+----------------+----------+
| Variable_name  | Value    |
+----------------+----------+
| tmp_table_size | 33554432 |
+----------------+----------+
1 row in set (0.00 sec)
```

建议生产环境中把 tmp_buffer_size 和 max_heap_table_size 设置成一样的值，如果两者值不一样，则会按照两者中小的值来起限制作用。

而且相应的值不要太小，值太小容易出现“converted heap to myisam”的报错。

针对 tmp 还有两个重要的参数：default_tmp_storage_engine 和 MySQL 5.7 新增的 internal_tmp_disk_storage_engine 参数。

- default_tmp_storage_engine 是指临时表默认的存储引擎。
- internal_tmp_disk_storage_engine 是指在磁盘上临时表的管理，由该参数决定（CREATE TEMPORARY TABLE）。

```
root@db 13:34:  [(none)]> show variables like '%tmp%';
+-----------------------------------+----------+
| Variable_name                     | Value    |
+-----------------------------------+----------+
| default_tmp_storage_engine        | InnoDB   |
| innodb_tmpdir                     |          |
| internal_tmp_disk_storage_engine  | InnoDB   |
| max_tmp_tables                    | 32       |
| slave_load_tmpdir                 | /tmp     |
| tmp_table_size                    | 33554432 |
| tmpdir                            | /tmp     |
+-----------------------------------+----------+
7 rows in set (0.01 sec)
```

有些读者经常会问 MyISAM 什么时候就可以彻底被废弃了呢？这个问题之前已经说过了，MySQL 8.0 之后就可以了，因为 8.0 之后 MySQL 库下的表都变成 InnoDB 存储引擎表了，但目前版本中，MySQL 库下还是 MyISAM 引擎表。

3.4.4　Buffer 状态及其链表结构

page 是 InnoDB 磁盘 I/O 的最小单位，数据是存放在 page 中的，那么对应到内存中就是一个个的 buffer。每个 buffer 又分为三种状态。

- free buffer：此状态下的 buffer 从未被使用，像一张白纸。但在实际生产中，数据库很繁忙的情况下，free buffer 的状态基本是不存在的。
- clean buffer：内存中 buffer 里面的数据和磁盘 page 的数据一致。
- dirty buffer：内存中新写入的数据还没有刷新到磁盘，跟磁盘中数据不一致。

buffer 在内存中是需要被组织起来的，由 chain 来管理，也就是链。之前介绍过 InnoDB 是双向链表结构，由三种不同的 buffer 状态衍生出了三条链表。

- free list：把那些 free 状态的 buffer 都串联起来。在数据库真正跑起来的时候，每次把 page 调到内存中，都先会判断 free buffer 的使用情况，如果不够用了，就会从 lru list 和 flush list 链表中释放 free buffer，以获得新的空闲 buffer。
- lru list：lru list 会把那些与磁盘数据一致，并且最近最少被使用的 buffer 串联起来，释放出 free buffer，page 调到内存中便于使用新的可用 buffer。
- flush list：把那些 dirty buffer 串联起来，为了方便刷新线程把脏数据刷到磁盘。推进 checkpoint Lsn，使实例崩溃之后，可以快速恢复。其实 flush list 中也隐藏着一个 lru 的规则。举个例子，假如目前有条数据是 19，下一秒变成了 29，再下一秒又变成了 39，这样的数据情况就是经常被访问、被使用到，暂时不能把它串起来。我们要把那些最近最少被“弄脏”的数据串起来，刷新到磁盘之后，释放出更多 free buffer 供我们使用。

3.4.5 各大刷新线程及其作用

介绍完内存结构之后，我们来了解一下 InnoDB 体系结构中间层的线程部分。InnoDB 存储引擎属于多线程的模型，后台有多种线程，负责处理不同的任务。

先来介绍 master thread 线程，它是后台线程中的主线程，优先级别最高。其内部有四个循环，分别为主循环 loop、后台循环 background loop、刷新循环 flush loop 和暂停循环 suspend loop。根据数据的运行状态会在这四个循环之间进行切换。在 loop 主循环中又包含两种操作，分别为每 1s 和每 10s 的操作。

每 1s 操作：

（1）日志缓冲刷新到磁盘，即使这个事务还没有提交。

（2）刷新脏页到磁盘。

（3）执行合并插入缓冲的操作。

（4）产生 checkpoint。

（5）清除无用的 table cache。

（6）如果当前没有用户活动，就可能切换到 background loop。

每 10s 操作：

（1）日志缓冲刷新到磁盘，即使事务还没有提交。

（2）执行合并插入缓冲的操作。

（3）刷新脏页到磁盘。

（4）删除无用的 undo 页。

（5）产生 checkpoint。

接下来就是四大 I/O 线程，分别是 read thread、write thread、redo log thread 和 change buffer thread。redo log thread 负责把日志缓冲中的内容刷新到 redo log 文件中。change buffer thread 负责把插入缓冲（change buffer）中的内容刷新到磁盘。read/write thread 是数据库的读写请求线程，默认值都是 4 个。如果使用高转速磁盘，可以适当调大该值。

```
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| innodb_read_io_threads | 4     |
| innodb_write_io_threads | 4    |
+------------------------+-------+
2 rows in set (0.01 sec)
```

page cleaner thread 是负责脏页刷新的线程，MySQL 5.7 之后可以增加多个。

```
root@db 13:37:  [(none)]> show variables like '%innodb_page%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| innodb_page_cleaners    | 4     |
| innodb_page_size        | 16384 |
+-------------------------+-------+
2 rows in set (0.00 sec)
```

purge thread 负责删除无用的 undo 页。由于进行 DML 语句的操作都会生成 undo，系统需要定期对 undo 页进行清理，这时就需要 purge 操作。从 MySQL 5.6 开始把 purge thread 单独从 master thread 中分离出来，通过 innodb_purge_thread 参数来控制 purge 的线程个数。默认是 1 个，最大值可以调整为 32 个。

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

checkpoint 线程的作用是在 redo log 发生切换时，执行 checkpoint。redo log 发生切换或者文件快写满时，会触发把脏页刷新到磁盘。还有就是可以确保 redo log 刷新到磁盘，实现真正持久化，避免数据丢失。

- error monitor thread 是负责数据库报错的监控线程。
- lock monitor thread 是负责锁的监控线程。

3.4.6　内存刷新机制

之前提及内存中的数据会刷新到磁盘，那么它的刷新机制是怎样的呢？我们主要来看三个内存部分的刷新情况，redo log buffer、data buffer 和 binlog cache 的刷新机制，如图 3-7 所示。

在 Oracle 和 MySQL 这种关系型数据库中，讲究日志先行策略，就是一条 DML 语句进入数据库之后，都会先写日志，再写数据文件。

1. redo log

先介绍什么是 redo log 文件。redo log 又称重做日志文件，用于记录事务操作的变化，记录的是数据修改之后的值，不管事务是否提交都会记录下来。在实例和介质失败时，重做日志文件就能派上用场，如数据库掉电，InnoDB 存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。

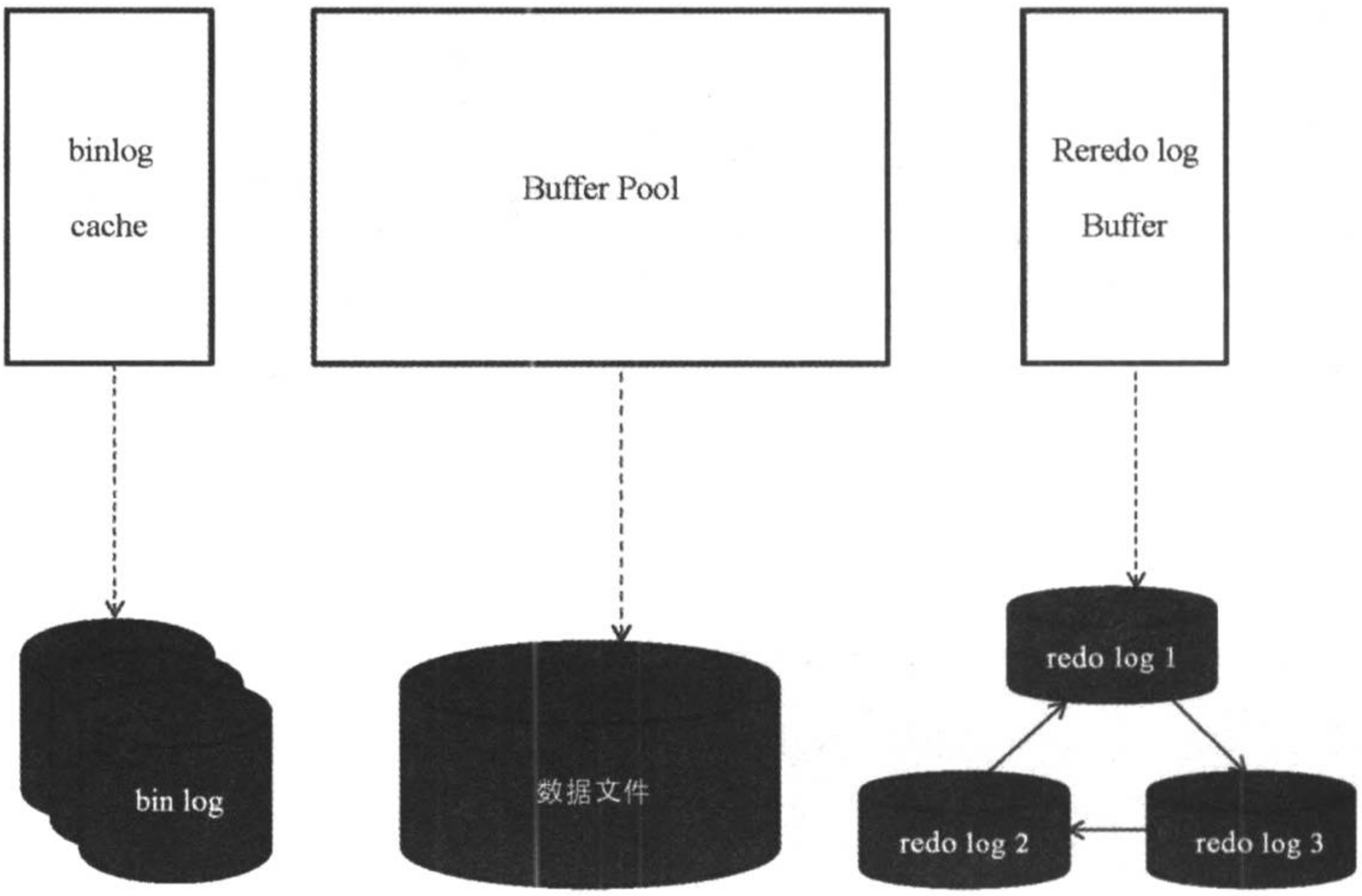


图 3-7　刷新机制

默认情况下至少有两个 redo log 文件，在磁盘上用 ib_logfile（0～N）命名。

```
[root@node3 mysql]# ll -h
total 5.1G
-rw-r-----. 1 mysql mysql    56 Sep 13 12:19 auto.cnf
-rw-r-----. 1 mysql mysql     5 Sep 13 12:34 db.pid
-rw-r-----. 1 mysql mysql   16K Sep 13 12:51 error.log
-rw-r-----. 1 mysql mysql   407 Sep 13 12:19 ib_buffer_pool
-rw-r-----. 1 mysql mysql 1.0G Sep 13 12:34 ibdata1
-rw-r-----. 1 mysql mysql 2.0G Sep 13 12:34 ib_logfile0
-rw-r-----. 1 mysql mysql 2.0G Sep 13 12:19 ib_logfile1
-rw-r-----. 1 mysql mysql  12M Sep 13 12:34 ibtmp1
-rw-r-----. 1 mysql mysql 8.7K Sep 13 12:26 innodb_status.5097
-rw-r-----. 1 mysql mysql 8.8K Sep 13 12:33 innodb_status.6458
-rw-r-----. 1 mysql mysql 8.8K Sep 13 13:40 innodb_status.7812
-rw-r-----. 1 mysql mysql   177 Sep 13 12:19 mybinlog.000001
-rw-r-----. 1 mysql mysql   154 Sep 13 12:26 mybinlog.000002
-rw-r-----. 1 mysql mysql 1.9K Sep 13 12:34 mybinlog.000003
-rw-r-----. 1 mysql mysql   769 Sep 13 12:39 mybinlog.000004
-rw-r-----. 1 mysql mysql   112 Sep 13 12:34 mybinlog.index
drwxr-x---. 2 mysql mysql 4.0K Sep 13 12:19 mysql
-rw-rw----. 1 root  root     5 Sep 13 12:34 mysqld_safe.pid
drwxr-x---. 2 mysql mysql 4.0K Sep 13 12:19 performance_schema
-rw-r-----. 1 mysql mysql 1.3K Sep 13 13:29 slow.log
drwxr-x---. 2 mysql mysql  12K Sep 13 12:19 sys
```

redo log 写的方式是顺序写、循环写。第一个文件写满之后，会按顺序写第二个，第二个写满之后，会继续写第三个。当写满最后一个文件时，会重新从第一个文件开始写。写满日志文件会产生切换操作，并执行 checkpoint，触发脏页的刷新。MySQL 数据库重启过程中，如果参数文件中的 redo log 值大小与当前 redo log 值不一致，会把现有 redo log 删除，并按照参数文件

中设置的大小，重新生成新的 redo log 文件。在磁盘中生成 redo log 文件之前，数据是先写在 redo log buffer 中的。下面讲一下 Redo log buffer 刷到磁盘的条件有哪些。

（1）通过 innodb_flush_log_at_trx_commit 参数来控制。该参数有三个值，分别为 0、1、2。

- 0 的含义：redo log thread 每隔 1s 会将 redo log buffer 中的数据写入 redo log 文件，同时进行刷盘操作，保证数据确实已经写入磁盘。但在该参数值下，每次事务提交并不会触发 redo log thread 将日志缓冲中的数据写入 redo log 文件。
- 1 的含义：每次事务提交时，都会触发 redo log thread 将日志缓冲中的数据写入文件，并“flush”到磁盘。该设置下是最安全的模式。保证数据库在主机断电、OS crash 下不会丢失任何已经提交的数据。
- 2 的含义：每次事务提交时，都会把 redo log buffer 的数据写入 redo log 文件，但是不会同时刷新到磁盘。

三种模式下，0 是性能最好，但是不太安全，MySQL 进程一旦崩溃会导致丢失一秒的数据。1 是安全性最高，但数据库性能最慢。2 是介于两者之间。生产中应该根据公司的业务情况来设置更合理的数值。

（2）master thread：每秒进行刷新。

（3）redo log buffer：使用超过一半的时候会触发刷新。

2. binlog

DML 语句既会写 redo log 文件，也会写 binlog 文件。binlog 在 MySQL 数据库中至关重要，叫作 MySQL 的二进制日志文件。功能应用于备份恢复和主从复制。那么从 binlog cache 刷新到磁盘的 binlog 文件中，需要的刷新条件是什么呢？

通过 sync_binlog 参数来决定，该参数有 *N* 个值，sync_binlog=0，当事务提交之后，MySQL 不做 fsync 之类的磁盘同步指令刷新 binlog_cache 中的信息到磁盘，而让 Filesystem 自行决定什么时候来做同步，或者 cache 满了之后才同步到磁盘。sync_binlog=n，每进行 *n* 次事务提交之后，MySQL 将进行一次 fsync 之类的磁盘同步指令来将 binlog_cache 中的数据强制写入磁盘。为了确保安全性，我们可以将 sync_binlog 设置为 1，为了获得最佳性能，我们可以将 sync_binlog 设置为 0。sync_binlog=1，innodb_flush_log_at_trx_commit=1，这就是数据库中的双一模式，可以确保数据库更加安全。

在了解最后一部分脏页刷新之前，我们有没有这样的疑问，既然 redo log 和 binlog 都记录了对数据真实修改的语句，那它们为什么要并存呢？有一个不就行了吗？面对这样的疑问，我们来看看它们到底有什么区别。

第一：记录内容的不同。

- binlog 是逻辑日志，记录所有数据的改变信息。
- redo log 是物理日志，记录所有 InnoDB 表数据的变化。

第二：记录内容的时间不同。

- binlog 记录 commit 完毕之后的 DML 和 DDL SQL 语句。
- redo log 记录事务发起之后的 DML 和 DDL SQL 语句。

第三：文件使用方式的不同。

- binlog 不是循环使用，在写满或者实例重启之后，会生成新的 binlog 文件。
- redo log 是循环使用，最后一个文件写满之后，会重新写第一个文件。

第四：作用不同。

- binlog 可以作为恢复数据使用，主从复制搭建。
- redo log 作为异常宕机或者介质故障后的数据恢复使用。

了解了 binlog 和 redo log 的区别之后，想一想它们之间有没有什么关联性呢？还有就是如何保证 InnoDB 存储引擎的日志文件（redo、undo）和二进制日志文件保持一致呢？我们都知道 MySQL 主从环境中，从库需要通过二进制日志来应用主库提交的事务，但如果主库 redo log 已经提交而二进制日志没有保持一致，那么就会造成从库数据丢失，主从数据不一致的情况。接下来我们讲一下两阶段提交的过程。

MySQL 两阶段提交过程：

两阶段提交分别为 prepare 和 commit 阶段。

- 准备阶段（transaction prepare）：事务 SQL 语句先写入 redo log buffer，然后做一个事务准备标记，再将 log buffer 中的数据刷新到 redo log。
- 提交阶段（commit）：将事务产生的 binlog 写入文件，刷入磁盘。

再在 redo log 中做一个事务提交的标记，并把 binlog 写成功的标记也一并写入 redo log 文件。

我们结合以下场景分析两阶段提交如何保证数据库的一致性。

场景一

准备阶段，redo log 刷新到磁盘了，但是 binlog 写盘前发生了 MySQL 实例 crash，这时会发生怎样的操作呢？即使 redo log 写盘成功了，但由于 binlog 未写入成功，我们需要执行回滚操作来保证数据库的一致性。

场景二

提交阶段，binlog 写盘成功了，这时 MySQL 实例 crash 了。这时 binlog 已经确保写成功了，我们在重启实例进行恢复的时候，只需要让 redo log 重做一次就可以了。

总结一下：其实只要 binlog 写入完成，那么在主从复制环境中，都会正常完成事务。

最后看一下脏页的刷新条件。

（1）重做日志 ib_logfile 文件写满后，在切换的过程中会执行 checkpoint，会触发脏页的刷新。

（2）通过 innodb_max_dirty_pages_pct 参数的值控制。该参数是指在 buffer pool 中 dirty page 所占的百分比，达到设置的值，就会触发脏页的刷新。

```
mysql> show variables like '%innodb_max_dirty_pages_pct';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| innodb_max_dirty_pages_pct     | 50    |
+--------------------------------+-------+
1 row in set (0.00 sec)
```

生产环境中建议该值可以设置为 25%～50%之间。默认值是 75%，调低是为了避免后期脏页刷新时影响整体数据库的 TPS 性能。

（3）由 innodb_adaptive_flushing 参数控制。该参数影响每秒刷新脏页的数目，它使用一种新的算法，通过 buf_flush_get_desired_flush_reate 函数判断重做日志产生的速度，来确定需要刷新脏页的最合适数量，替换了之前的 innodb_max_dirty_pages_pct，刷新至多 100 个脏页到磁盘。这样即使脏页比例小于 innodb_max_dirty_pages_pct 参数设置的值，也会刷新一定量的脏页。innodb_adaptive_flushing 参数默认是开启的。

```
mysql> show variables like "%innodb_adaptive_flushing";
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_adaptive_flushing      | ON    |
+-------------------------------+-------+
1 row in set (0.00 sec)
```

3.4.7　InnoDB 的三大特性

插入缓冲（change buffer）、两次写（double write）、自适应哈希索引（adaptive hash index）构成了 InnoDB 的三大特性，这些特性让 InnoDB 存储引擎有了更好的性能和可靠性。

（1）插入缓冲。

影响数据库最主要的性能问题就是 I/O，而插入缓冲的作用就是把普通索引上的 DML 操作从随机 I/O 变成顺序 I/O，提高 I/O 效率。它的工作原理也很简单，就是先判断插入的普通索引页是否在缓冲池中，如果在就可以直接插入，如果不在就要先放到 change buffer 中，然后进行 change buffer 和普通索引的合并操作，可以将多个插入合并到一个操作中，一下子就提高了普

通索引的插入性能。涉及以下参数：

```
mysql> show variables like '%change%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_change_buffer_max_size | 25    |
| innodb_change_buffering       | all   |
+-------------------------------+-------+
2 rows in set (0.00 sec)
```

innodb_change_buffer_max_size 的含义：占 innodb_buffer_pool 的最大比例，默认是 25%，最大占 buffer pool 1/4 的大小。建议调整为 50。

innodb_change_buffering：change buffer 的类型。

有如下几种类型：

- all:buffer inserts, delete-marking operations, and purges

  缓冲全部 inserts、delete 标记操作和 purges 操作。

- none:Do not buffer any operations

  关闭 insert buffer。

- inserts:Buffer insert operations

  insert 标记操作。

- deletes:Buffer delete-marking operations

  delete 标记操作。

- changes:Buffer both inserts and delete-marking

  未进行实际 insert 和 delete，只是标记，等待后续 purge。

- purges:Buffer the physical deletion operations that happen in the background

  缓冲后台进程的 purges（物理删除）操作。建议选择默认的 all 就可以了。

（2）两次写（double write）。

插入缓冲带来的是针对普通索引插入性能上的提升，而 double write 就是保证写入的安全性，防止在 MySQL 实例发生宕机时，InnoDB 发生数据页部分页写（partial page write）的问题。有些读者会说，数据库实例崩溃，我们可以通过 redo log 进行恢复，不会有任何问题。但 redo log 文件记录的是页的物理操作，如果页都损坏了，是无法进行任何恢复操作的，巧妇难为无米之炊就是这个道理。所以我们需要页的一个副本，如果实例宕机了，可以先通过副本把原来的页还原出来，再通过 redo log 进行恢复、重做。这就是 double write 的作用。

双写缓冲是一个位于系统表空间中的存储区域，InnoDB 缓冲池中刷出的脏页在被写入数据文件之前，都先会写入 double write buffer。然后从两次写缓冲区分两次，每次将 1MB 大小的数据写入磁盘共享表空间（double write），最后再从 double write buffer 写入数据文件。虽然数据总是写两次，但两次写缓冲并不需要两倍的 I/O 开销，或者两倍的其他 I/O 操作。数据写入双写缓冲后，其本身是一个大型的连续块，会通过一次 fsync()通知操作系统。双写缓冲在大多数场景下都是默认有效的。可以通过设置 innodb_doublewrite 为 0 来关闭双写缓冲。从 MySQL 5.7.4 开始，如果系统表空间文件 ibdata 位于 Fusion-io 设备（支持原子写），双写缓冲自动失效并且 Fusion-io 原子写会被用于所有数据文件。由于双写缓冲设置是全局的，双写缓冲也会在位于非 Fusion-io 硬件的数据文件上失效。该特性仅支持 Fusion-io 硬件，并且只能通过 Linux 系统上的 Fusion-io NVMFS 启用。为了充分利用该特性，O_DIRECT 推荐使用 innodb_flush_method 设置。

（3）自适应哈希索引（自适应哈希索引）。

InnoDB 存储引擎有一个机制，可以监控索引的搜索，如果 InnoDB 注意到查询可以通过建立哈希索引得到优化，那么就会自动完成这件事。可以通过 innodb_adaptive_hash_index 参数来控制。默认情况下是开启的。

```
root@db 13:43:  [(none)]> show variables like '%innodb_adaptive_hash_index';
+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
| innodb_adaptive_hash_index      | ON    |
+---------------------------------+-------+
1 row in set (0.00 sec)
```

从 MySQL 5.7.8 开始，自适应哈希索引搜索系统是分区的。每个索引都会绑定到一个特殊的分区上面，而且各个分区都有自己的锁存器来进行保护。

分区可以通过 innodb_adaptive_hash_index_parts 参数来控制。该参数的默认值为 8 个，最大可以设置为 512。

```
root@db 13:43:  [(none)]> show variables like '%innodb_adaptive_hash_index_parts';
+----------------------------------+-------+
| Variable_name                    | Value |
+----------------------------------+-------+
| innodb_adaptive_hash_index_parts | 8     |
+----------------------------------+-------+
1 row in set (0.00 sec)
```

通过设置分区值，可以降低争用，提高并发性。

还可以通过 SHOW ENGINE INNODB STATUS 命令所输出的 SEMAPHORES 部分来监控自

适应哈希索引的使用及其竞争情况。如果你看到许多线程正在等待一个在 btr0sea.c 中创建的 RW-latch，则它可能被用于禁用自适应哈希索引。

```
------------
SEMAPHORES
------------
OS WAIT ARRAY INFO: reservation count 14
OS WAIT ARRAY INFO: signal count 10
RW-shared spins 0, rounds 8, OS waits 4
RW-excl spins 0, rounds 0, OS waits 0
RW-sx spins 0, rounds 0, OS waits 0
Spin rounds per wait: 8.00 RW-shared, 0.00 RW-excl, 0.00 RW-sx
------------
```

第4章 数据库文件

本章从 MySQL 数据库和存储引擎层面来介绍各种类型的文件。数据库层面的文件有参数文件（my.cnf）、错误日志（error log）、慢查询日志（slow log）、全量日志（general log）、二进制日志（binlog）文件、审计日志（audit log）、中继日志（relay log）、套接字文件（socket）、进程（pid）文件和表结构文件。存储引擎层面有 redo log 和 undo log 日志文件。

4.1 参数文件

前面安装启动 MySQL 时已经介绍过参数文件。

在启动 MySQL 实例的过程中，会按照/etc/my.cnf->/etc/mysql/my.cnf/->/usr/local/mysql/my.cnf->~/.my.cnf 这样的一个优先级别的顺序去读取参数文件。如果想指定默认的参数文件，需要配合--defaults-file 参数。

在 my.cnf 文件中，分为 client section 和 server section 两块。

Client section 是用来配置 MySQL 客户端的参数。

```
[client]
port    = 3306
socket  = /tmp/mysql.sock

[mysql]
prompt="\u@db \R:\m:\s [\d]> "
no-auto-rehash
```

我们下面主要介绍一下参数文件中 MySQL Server 端核心参数的含义。可以通过 show variables like %参数名%;来查看 MySQL 数据库中的参数。

1. innodb_buffer_pool

该缓冲池位于主内存中，InnoDB 用它来缓存被访问过的表和索引文件，使常用数据可以直接在内存中被处理，从而提升处理速度。

建议在服务器只跑数据库一个应用的前提下，该参数可以设置为物理内存的 50%～80%。如果分配过多内存给数据库，有可能会导致系统内存不够，出现 swap 或者 oom 的现象。目前 MySQL 5.7 版本支持在线修改 innodb_buffer_pool 的大小了，所以前期没必要设置太大。

```
root@db 12:08:  [(none)]> show variables like '%innodb_buffer_pool_size'
    -> ;
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| innodb_buffer_pool_size | 1073741824 |
+-------------------------+------------+
1 row in set (0.00 sec)
```

2. innodb_buffer_pool_instance

该参数的默认值为 1，MySQL 5.6.6 之后可以调整为多个，表示 InnoDB 缓冲区可以被划分为多个区域，也可以理解为把 innodb_buffer_pool 划分为多个实例，可以提高并发性，避免在高并发环境下，出现内存的争用问题。设置完成之后，每个缓冲区各自管理自己的数据，互不干涉。通过 show engine innodb status 也可以看到每个 instance 使用内存的情况。

注意，只有当 innodb_buffer_pool 大于 1GB 时，生成的 innodb_buffer_pool 多实例才生效。

```
root@db 12:08:  [(none)]> show variables like '%innodb_buffer_pool_instances';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| innodb_buffer_pool_instances | 8     |
+------------------------------+-------+
1 row in set (0.00 sec)
```

数据库在业务高峰期时，如果实例突然宕机，那么内存中所保存的热点数据就都消失了，只能再从磁盘中读取并调回到内存中，这样对数据库 I/O 的压力太大了，严重影响生产环境的现有业务。那么问题来了，如何把热数据快速加载回来呢？我们可以开启以下两个参数，都设置为 1。默认是关闭状态的。

- innodb_buffer_pool_load_at_startup
- innodb_buffer_pool_dump_at_shutdown

```
root@db 13:47:  [(none)]> show variables like '%innodb_buffer_pool_%';
+---------------------------------------+-------------------+
| Variable_name                         | Value             |
+---------------------------------------+-------------------+
| innodb_buffer_pool_chunk_size         | 134217728         |
| innodb_buffer_pool_dump_at_shutdown   | ON                |
| innodb_buffer_pool_dump_now           | OFF               |
| innodb_buffer_pool_dump_pct           | 25                |
| innodb_buffer_pool_filename           | ib_buffer_pool    |
| innodb_buffer_pool_instances          | 8                 |
| innodb_buffer_pool_load_abort         | OFF               |
| innodb_buffer_pool_load_at_startup    | ON                |
| innodb_buffer_pool_load_now           | OFF               |
| innodb_buffer_pool_size               | 1073741824        |
+---------------------------------------+-------------------+
10 rows in set (0.00 sec)
```

在数据库实例关闭的时候，可以把热数据的元数据信息“dump”出来，保存到 ib_buffer_pool 文件中。

```
root@db 12:39:  [(none)]> show variables like '%innodb_buffer_pool_file%';
+-------------------------------+----------------+
| Variable_name                 | Value          |
+-------------------------------+----------------+
| innodb_buffer_pool_filename   | ib_buffer_pool |
+-------------------------------+----------------+
1 row in set (0.00 sec)
```

当实例再次启动的时候，把元数据信息快速加载回内存。其实所谓的元数据信息就是 space number 和 page number 的列表信息。

```
root@db 13:49:  [information_schema]> select space,page_number from  INNODB_BUFFER_PAGE limit 3;
+-------+-------------+
| space | page_number |
+-------+-------------+
|     0 |           7 |
|     0 |           3 |
|     0 |           2 |
+-------+-------------+
3 rows in set (0.14 sec)
```

- innodb_data_file_path

该参数可以指定系统表空间文件的路径和 ibdata1 文件的大小。默认大小是 10MB，这里建议调整为 1GB。

```
root@db 12:16:  [(none)]> show variables like '%innodb_data_file%';
+-----------------------+-----------------------------+
| Variable_name         | Value                       |
+-----------------------+-----------------------------+
| innodb_data_file_path | ibdata1:1G:autoextend       |
+-----------------------+-----------------------------+
1 row in set (0.00 sec)
```

- innodb_flush_log_at_trx_commit

- sync_binlog
- innodb_max_dirty_pages_pct

前面已经介绍过这三个参数，影响 redo log、binlog、脏页的刷新参数。参考 3.4.6 节内存刷新机制的内容。

- innodb_thread_concurrency

InnoDB 内核最大并发线程数，默认值为 0，代表不受限制。

```
root@db 13:52:  [(none)]> show variables like '%innodb_thread_concurrency%';
+---------------------------+---------+
| Variable_name             | Value   |
+---------------------------+---------+
| innodb_thread_concurrency | 0       |
+---------------------------+---------+
1 row in set (0.00 sec)
```

- interactive_timeout
- wait_timeout

interactive_timeout 是服务器关闭交互式连接前等待活动的时间，默认是 28800s（8 小时）。

wait_timeout 是服务器关闭非交互式连接之前等待活动的时间，默认是 28800s（8 小时）。

这两个参数一定要一起调整，并且值要一致。而且要避免过大的连接时间，默认 8 小时太长了，建议调整为 300～600s。

```
root@db 13:53:  [(none)]> show variables like '%timeout%';
+----------------------------------+----------+
| Variable_name                    | Value    |
+----------------------------------+----------+
| connect_timeout                  | 10       |
| delayed_insert_timeout           | 300      |
| have_statement_timeout           | YES      |
| innodb_flush_log_at_timeout      | 1        |
| innodb_lock_wait_timeout         | 10       |
| innodb_rollback_on_timeout       | ON       |
| interactive_timeout              | 600      |
| lock_wait_timeout                | 3600     |
| net_read_timeout                 | 30       |
| net_write_timeout                | 60       |
| rpl_stop_slave_timeout           | 31536000 |
| slave_net_timeout                | 60       |
| wait_timeout                     | 600      |
+----------------------------------+----------+
13 rows in set (0.01 sec)
```

- innodb_flush_method

这个参数影响 InnoDB 数据文件、redo log 文件的打开刷写模式。

主要有三个值，O_SYNC、O_DSYNC、O_DIRECT。

建议选择 O_DIRECT 模式，数据文件直接从 MySQL InnoDB Buffer 写入到磁盘，不用通过 os buffer。

```
root@db 13:41:  [(none)]> show variables like '%innodb_flush_method%';
+--------------------+----------+
| Variable_name      | Value    |
+--------------------+----------+
| innodb_flush_method | O_DIRECT |
+--------------------+----------+
1 row in set (0.00 sec)
```

有高速 I/O 设备，或者有阵列卡+cache 的话，O_DIRECT 是最理想的选择。

- innodb_old_blocks_time
- innodb_old_blocks_pct

InnoDB 缓冲池用来缓存数据和索引文件，内部由 LRU 链表管理，LRU 链表又进一步可以分为old pages list 和 young pages list。old pages list 里面存放的是长时间未被访问的数据页，young list 中存放那些最新、最近被访问的数据页。当超过 innodb_old_blocks_time 参数所设置的时间时，就会移动到 old list 中，默认是 1000ms。

```
root@db 13:53:  [(none)]> show variables like '%innodb_old_blocks_time';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_old_blocks_time | 1000  |
+-----------------------+-------+
1 row in set (0.01 sec)
```

old pages list 占整个列表的 37%，占整个 buffer pool 的 3/8，由 innodb_old_blocks_pct 参数决定。

```
root@db 13:54:  [(none)]> show variables like '%innodb_old_blocks_pct%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_old_blocks_pct | 37    |
+-----------------------+-------+
1 row in set (0.01 sec)
```

案例分析：生产环境中如果遇到大表扫描或者使用 mysqldump 这样的操作进行备份时，很有可能踢走热数据，给数据库的 I/O 带来压力。这时就可以适当减少 innodb_old_blocks_pct 的值分配，保证更多的热数据不会被冲掉。

- transaction_isolation

MySQL 数据库的事务隔离级别有四种，分别为 READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ 和 SERIALIZABLE。默认采用 REPEATABLE-READ（可重复读），后面讲事务的时候会逐一讲解。

- innodb_open_files

InnoDB 可同时打开的.ibd 文件的个数，最小值为 10，默认值为 300。建议调整为 65535。

- innodb_log_buffer_size
- innodb_log_file_size

innodb_log_buffer_size 是日志缓冲的大小，InnoDB 改变数据的时候，它会把这次改动的记录先写到日志缓冲中。可以通过 show global status like"%Innodb_log_waits%"进行查看，如果 innodb_log_waits（等待日志缓冲刷出的次数）的值大于 0，而且继续增长，就可以增大 log buffer 的大小。取值范围为 16MB～64MB。

innodb_log_file_size 是指 Redo log 日志的大小，该值设置不宜过大也不宜过小，如果设置太大，实例恢复的时候需要较长时间，如果设置太小，会造成 redo log 切换频繁，产生无用的 I/O 消耗，影响数据库性能。

- innodb_log_files_in_group

redo log 文件组中日志文件的数量，默认情况下至少有 2 个。

- max_connections

该参数代表 MySQL 数据库的最大连接数。参数默认值是 151。

案例分析：

这个数值对于并发连接很多的数据库应用是远远不够的，当连接请求大于默认连接数时，就会出现无法连接数据库的错误，在生产环境中我们可能经常会遇到“too many connections”的报错信息。

解决方法：

（1）可以适当增大 max_connections 的值，这里需要注意在调大的时候，要考虑数据库能否承担这么多连接数带来的压力。如果值调得特大，就容易出现实例宕机的情况。所以压力测试也是很重要的一个环节。

（2）调整 InnoDB 内部并发数，通过 innodb_thread_concurrency 参数进行控制。该参数从 MySQL 5.6 版本开始默认值为 0。

```
mysql> show variables like '%innodb_thread_c%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| innodb_thread_concurrency | 0     |
+---------------------------+-------+
1 row in set (0.00 sec)
```

0 代表并发数不受限制，这里建议调整为服务器上逻辑 CPU 核数的两倍。可以先改成 16 或者 64（看服务器压力）。如果非常大，可以先改小一点让服务器的压力下来，之后再慢慢增

大（根据自己的业务而定）。建议先调整为 16 即可。

（3）MySQL 数据库随着连接数的增加性能会下降，可以考虑和开发配合设置 thread pool（连接池），让连接复用。在 MySQL 5.7 的商业版中，引入了 thread pool。

（4）有的监控程序会读取 information_schema 下面的表，可以考虑关闭下面的参数：

- innodb_stats_on_metadata
- set global innodb_stats_on_metadata=0

（5）考虑是不是有死锁的发生。

- expire_logs_days

该参数代表 binlog 的过期时间，单位是天。

- slow_query_log

慢查询日志的开关，该参数等于 1 代表开启慢查询。生产环境要开启慢查询日志。

- long_query_time

慢查询的时间，某条 SQL 语句超过该参数设置的时间，就会记录到慢查询日志中。单位是秒。

- log_queries_not_using_indexes

如果运行的 SQL 语句没有使用索引，则 MySQL 数据库同样会将这条 SQL 语句记录到慢查询日志文件中。生产环境中建议开启此参数：

```
set global  log_queries_not_using_indexes =on
```

- server-id

用于标识 MySQL 在同一组主从结构中的唯一标识。搭建主从环境时，两台机器的 server-id 不能一样。

- binlog_format

该参数代表二进制日志的格式。Binlog 的格式有 statement、row 和 mixed 三种。生产环境中使用 row 这种格式更安全，不会出现跨库复制丢数据的情况。

- lower_case_table_names

表名是否区分大小的参数。默认是值为 0。0 代表区分大小写，1 代表不区分大小写，以小写存储。

- innodb_fast_shutdown

该参数影响表的存储引擎为 InnoDB 在关闭时的行为，有 0、1、2 三个值。

0 代表在 InnoDB 关闭的时候，需要执行 purge all、merge change buffer、flush dirty pages 操作。这是最慢的一种关闭方式，但是 restart 的时候也是最快的。

> 注：0 值为默认值，不需要修改。

1 代表在 InnoDB 关闭的时候，它不需要执行 purge all、merge insert buffer 等操作，只需要执行 flush dirty page。

2 代表不完成 full purge 和 merge insert buffer 等操作，也不刷新脏页到磁盘，只是将日志写入日志文件，虽然不会丢数据，但是下次启动会进行 recovery 操作。

- innodb_force_recovery

该参数影响 InnoDB 存储引擎恢复时的行为，取值为 0、1、2、3、4、5、6。

0 值的含义：表示当需要恢复时执行所有的恢复操作。

> 注：0 是默认值，不需要修改。

1 值的含义：忽略检查到的 corrupt 页。

2 值的含义：阻止主线程的运行，如主线程需要执行 full purge 操作，会导致 crash。

3 值的含义：不执行事务回滚操作。

4 值的含义：不执行插入缓冲的合并操作。

5 值的含义：不查看撤销日志，InnoDB 存储引擎会将未提交的事务视为已提交。

6 值的含义：不执行前滚操作。

- innodb_status_output
- innodb_status_output_locks

这两个参数建议关闭（innodb_status_output=0，innodb_status_output_locks=0），否则会把对数据库监控的信息全部记录到 error log 中，使错误日志增长过快，造成磁盘空间的使用紧张。

- innodb_io_capacity

InnoDB 后台进程最大的 I/O 性能指标，影响刷新脏页和插入缓冲的数量。默认值是 200，在高转速磁盘下，可以根据需要适当提高该参数的值。

auto_increment_increment 表示自增长字段每次递增的量。默认值为 1。

auto_increment_offset 表示自增长字段从哪个值开始。默认值为 1。

4.2　参数类型

MySQL 数据库中把参数分为两类，一类是动态参数，一类是静态参数。

1. 动态参数

MySQL 实例在运行的过程中可以对参数在线修改。可以通过 set global 或者 set session 两个命令，在数据库中完成设置。global 代表全局参数，意味着用其修改完成之后，退出当前会话后仍然生效，但如果重启数据库，那之前设置好的参数值也会失效。session 是只针对当前会话生效，一旦退出，设置的参数就立即失效了。当然，也有些动态参数只能在当前会话中修改。

2. 静态参数

顾名思义，就是在线无法修改参数。

```
mysql> show variables like '%lower_case_table_names%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| lower_case_table_names | 0     |
+------------------------+-------+
1 row in set (0.00 sec)

mysql> set global lower_case_table_names=1;
ERROR 1238 (HY000): Variable 'lower_case_table_names' is a read only variable
```

以上为报出 read only variable 的错误，证明无法在线修改，只能把需要修改的参数写到配置文件中，重启数据库才可以。

4.3　错误日志文件（error log）

错误日志文件一般存放在数据目录下，以 error.log 作为文件名的结尾。

```
root@db 11:13:  [(none)]> show variables like '%log_error';
+---------------+---------------------------+
| Variable_name | Value                     |
+---------------+---------------------------+
| log_error     | /data/mysql/error.log     |
+---------------+---------------------------+
1 row in set (0.01 sec)
```

类似于 Oracle 数据库中的 alert log。错误日志记录着 MySQL 的启动、运行、关闭过程中出现的问题。尤其作为初学者，要学会利用 error log 来定位问题。

错误日志中不光记录着错误的信息，在 MySQL 5.7 初识化数据库中，加上--initialize 参数，会生成一个临时的数据库初始化密码，记录在 log-error（错误日志）中。

当然，错误日志中还有一些 warnings 信息对我们进行一些优化工作也很有帮助。

```
170913 13:07:37 mysqld_safe Starting mysqld daemon with databases from /data/mysql
2017-09-13 13:07:37 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Plea
se use --explicit_defaults_for_timestamp server option (see documentation for more detail
s).
2017-09-13 13:07:37 6577 [Note] Plugin 'FEDERATED' is disabled.
```

这条 warnings 信息提示让你在配置文件中加上 explicit_defaults_for_timestamp 参数。那么该参数有何意义呢？下面来详细介绍一下。

MySQL 中的 TIMESTAMP 类型和其他类型有点不一样（在没有设置 explicit_defaults_for_timestamp=1 的情况下），如果表中的 TIMESTAMP 列没有显式地指明 null 属性，那么该列会被自动加上 not null 属性(而其他类型的列如果没有被显式地指定 not null，那么是允许 null 值的)，如果往这个列中插入 null 值，会自动地设置该列的值为 current timestamp 值。表中的第一个 TIMESTAMP 列，如果没有指定 null 属性或者没有指定默认值，也没有指定 ON UPDATE 语句，那么该列会自动被加上 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP 属性。第一个 TIMESTAMP 列之后的其他 TIMESTAMP 类型的列，如果没有指定 null 属性，也没有指定默认值，那么该列会被自动加上 DEFAULT '0000-00-00 00:00:00'属性。如果 insert 语句中没有为该列指定值，那么该列中插入'0000-00-00 00:00:00'，并且没有 warning。

如果在启动时在配置文件中指定了 explicit_defaults_for_timestamp=1，此时如果 TIMESTAMP 列没有显式地指定 not null 属性，那么默认的该列可以为 null，此时向该列中插入 null 值，会直接记录 null，而不是 current timestamp。如果 TIMESTAMP 列被加上了 not null 属性，并且没有指定默认值，这时如果向表中插入记录，但是没有给该 TIMESTAMP 列指定值，如果 sql_mode=strict 被指定了，那么会直接报错。

实验过程

在未设置 explicit_defaults_for_timestamp=1 的情况下，没有为 timestamp 列指定 not null 属性，也没有为其指定默认值：

```
mysql> create table zs(a timestamp,b timestamp,c timestamp);
Query OK, 0 rows affected (0.62 sec)

mysql> show create table zs\G;
*************************** 1. row ***************************
       Table: zs
Create Table: CREATE TABLE `zs` (
  `a` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `b` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
  `c` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00'
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

从表结构中可以看到，表中三个 timestamp 列都被自动设置为 not null，并且表中第一个

timestamp 列被设置了 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 默认值，其他 timestamp 列被加上了 DEFAULT '0000-00-00 00:00:00'默认值。

往表中插入指定 null 值，则自动会变成当前的系统时间。

```
mysql> insert into zs (a,b,c) values (null,null,null);
Query OK, 1 row affected (0.00 sec)

mysql> select * from zs;
+---------------------+---------------------+---------------------+
| a                   | b                   | c                   |
+---------------------+---------------------+---------------------+
| 2017-09-13 14:07:30 | 2017-09-13 14:07:30 | 2017-09-13 14:07:30 |
+---------------------+---------------------+---------------------+
1 row in set (0.00 sec)
```

插入时没有指定 b、c 列上具体的值。

```
mysql> insert into zs (a) values (null);
Query OK, 1 row affected (0.00 sec)

mysql> select * from zs;
+---------------------+---------------------+---------------------+
| a                   | b                   | c                   |
+---------------------+---------------------+---------------------+
| 2017-09-13 14:07:30 | 2017-09-13 14:07:30 | 2017-09-13 14:07:30 |
| 2017-09-13 14:08:19 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 |
+---------------------+---------------------+---------------------+
2 rows in set (0.01 sec)
```

可见插入时未指定值的 timestamp 列中被插入了 0000-00-00 00:00:00。

在设置 explicit_defaults_for_timestamp=1 的情况下：

```
root@db 14:11:  [zs]> create table zs(a timestamp,b timestamp,c timestamp);
Query OK, 0 rows affected (0.10 sec)

root@db 14:11:  [zs]> show create table zs\G;
*************************** 1. row ***************************
*
       Table: zs
Create Table: CREATE TABLE `zs` (
  `a` timestamp NULL DEFAULT NULL,
  `b` timestamp NULL DEFAULT NULL,
  `c` timestamp NULL DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

没有为 timestamp 列指定 null 属性，也没有为其指定默认值，在表结构中可以看到，三个 timestamp 列都被设置为 null，并且设置了默认值为 null。未显示指定 timestamp 列为 not null 时，能够向 timestamp 列中插入 null 值：

```
root@db 14:11:  [zs]> insert into zs (a,b,c) values (null,null,null);
Query OK, 1 row affected (0.31 sec)

root@db 14:13:  [zs]> select * from zs;
+------+------+------+
| a    | b    | c    |
+------+------+------+
| NULL | NULL | NULL |
+------+------+------+
1 row in set (0.00 sec)
```

插入时没有为 b、c 列指定值时，自动插入 null 值：

```
root@db 14:13:  [zs]> insert into zs (a) values (null);
Query OK, 1 row affected (0.01 sec)

root@db 14:14:  [zs]> select * from zs;
+------+------+------+
| a    | b    | c    |
+------+------+------+
| NULL | NULL | NULL |
| NULL | NULL | NULL |
+------+------+------+
2 rows in set (0.00 sec)
```

为 c 列指定 not null 属性，sql_mode=stric，进行插入测试：

```
root@db 14:14:  [zs]> show variables like '%sql_mode%';
+---------------+-----------------------------------------------------------------------------------------------------------------------------------------------+
| Variable_name | Value                                                                                                                                         |
+---------------+-----------------------------------------------------------------------------------------------------------------------------------------------+
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------------------------------------------+
```

```
root@db 14:16:  [zs]> create table zs (a timestamp,b timestamp,c timestamp not null);
Query OK, 0 rows affected (0.30 sec)

root@db 14:17:  [zs]> show create table zs\G;
*************************** 1. row ***************************
       Table: zs
Create Table: CREATE TABLE `zs` (
  `a` timestamp NULL DEFAULT NULL,
  `b` timestamp NULL DEFAULT NULL,
  `c` timestamp NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)

ERROR:
No query specified

root@db 14:17:  [zs]>
root@db 14:18:  [zs]> insert into zs (a) values(null);
ERROR 1364 (HY000): Field 'c' doesn't have a default value
```

为 c 列指定了 not null 属性，在 sql_mode= stric 时，如果插入时该列没有指定值，会直接报错。

> 注：建议 sql_mode 使用严格模式。

通过以上实验过程，我们既了解了 explicit_defaults_for_timestamp 参数的含义，又明白了不要忽略错误日志中对我们很有帮助的 warnings 信息。

4.4　二进制日志文件（binary log）

之前的章节中也介绍过二进制日志。它和 redo log 一样都用于记录对 MySQL 数据库真正执行更改的所有操作（DML 语句），不包含那些没有修改任何数据的语句，不会记录 select 和 show 这样的语句，如果需要记录，则需要开启全量日志功能。

二进制日志的主要作用：

（1）可以完成主从复制功能。在主服务器上把所有修改数据的操作记录到 binlog 中，通过网路发送给从服务器，从而达到主从同步。

（2）进行恢复操作。数据可以通过 binlog 日志，使用 mysqlbinlog 命令，实现基于时间点和位置的恢复操作。

使用配置参数 log-bin=[filename]可以启动二进制日志。如果没有命名 filename，则以主机名来作为二进制日志的文件名。二进制日志文件默认存储在数据目录下。

```
mysql> show variables like '%log_bin%';
+---------------------------------+---------------------------------+
| Variable_name                   | Value                           |
+---------------------------------+---------------------------------+
| log_bin                         | ON                              |
| log_bin_basename                | /data/mysql/mysql-bin           |
| log_bin_index                   | /data/mysql/mysql-bin.index     |
| log_bin_trust_function_creators | OFF                             |
| log_bin_use_v1_row_events       | OFF                             |
| sql_log_bin                     | ON                              |
+---------------------------------+---------------------------------+
6 rows in set (0.00 sec)
```

可以看到二进制日志是开启的，二进制日志的名字以 mysql-bin 为前缀，日志文件存储在数据目录/data/mysql 下。binlog 的后缀名是它的序列号，号码是按顺序排列的。在磁盘上可以看到：

```
[root@node3 sysbench]# cd /data/mysql
[root@node3 mysql]# ll -h
total 1.4G
-rw-rw---- 1 mysql mysql   56 Sep 13 10:54 auto.cnf
-rw-rw---- 1 mysql mysql  17K Sep 13 13:07 error.log
-rw-rw---- 1 mysql mysql 1.0G Sep 13 14:08 ibdata1
-rw-rw---- 1 mysql mysql 128M Sep 13 14:08 ib_logfile0
-rw-rw---- 1 mysql mysql 128M Sep 13 10:49 ib_logfile1
drwx------ 2 mysql mysql 4.0K Sep 13 10:49 mysql
-rw-rw---- 1 mysql mysql  64K Sep 13 10:49 mysql-bin.000001
-rw-rw---- 1 mysql mysql 1.1M Sep 13 10:49 mysql-bin.000002
-rw-rw---- 1 mysql mysql 6.0M Sep 13 12:53 mysql-bin.000003
-rw-rw---- 1 mysql mysql  14M Sep 13 13:01 mysql-bin.000004
-rw-rw---- 1 mysql mysql  14M Sep 13 13:03 mysql-bin.000005
-rw-rw---- 1 mysql mysql  14M Sep 13 13:07 mysql-bin.000006
-rw-rw---- 1 mysql mysql  14M Sep 13 14:08 mysql-bin.000007
-rw-rw---- 1 mysql mysql  203 Sep 13 13:07 mysql-bin.index
-rw-rw---- 1 mysql mysql    5 Sep 13 13:07 node3.pid
drwx------ 2 mysql mysql 4.0K Sep 13 10:49 performance_schema
-rw-rw---- 1 mysql mysql 6.4M Sep 13 14:05 slow.log
drwx------ 2 mysql mysql 4.0K Sep 13 14:05 test
```

在数据库命令行，执行 show binary logs 也可以列出当前 binlog 文件及值的大小：

```
mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000001 |     65273 |
| mysql-bin.000002 |   1050551 |
| mysql-bin.000003 |   6231106 |
| mysql-bin.000004 |  14089164 |
| mysql-bin.000005 |  13982254 |
| mysql-bin.000006 |  14098210 |
| mysql-bin.000007 |  14201961 |
+------------------+-----------+
7 rows in set (0.00 sec)
```

在数据库命令行，执行 show master status 可以看到 MySQL 当前的日志及状态：

```
mysql> show master status;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000007 | 14201961 |              |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

可以看出当前使用的 binlog 文件是 mysql-bin.000007，文件的偏移量是 14201961 个字节。

数据目录下 mysql-bin.index 文件是二进制日志的索引文件，用来记录产生的二进制日志的序号。

```
[root@node3 mysql]# pwd
/data/mysql
[root@node3 mysql]# cat mysql-bin.index
/data/mysql/mysql-bin.000001
/data/mysql/mysql-bin.000002
/data/mysql/mysql-bin.000003
/data/mysql/mysql-bin.000004
/data/mysql/mysql-bin.000005
/data/mysql/mysql-bin.000006
/data/mysql/mysql-bin.000007
```

涉及 binlog 的相关参数介绍。

（1）max_binlog_size，该参数制定了单个 binlog 的最大值。如果超过该值就会自动生成新的 binlog 文件（重启 MySQL 实例也会生成新的 binlog）。从 MySQL 5.0 开始，max_binlog_size 的默认值为 1GB。

```
mysql> show variables like '%max_binlog_size%';
+-----------------+------------+
| Variable_name   | Value      |
+-----------------+------------+
| max_binlog_size | 1073741824 |
+-----------------+------------+
1 row in set (0.00 sec)
```

> 注：一般情况下，生产环境中我们控制 binlog 的生成时间的最小间隔保持在 2~5 分钟，所以该参数不要太大，可以调整为 256MB。

（2）binlog_cache_size，所有未提交的事务会记录到一个缓存中，等待事务提交时，直接将缓存中的二进制日志写入二进制日志文件，该缓存的大小由 binlog_cache_size 决定，它的默认大小为 32KB，并且 binlog_cache_size 是基于会话的，也就是当一个线程要开始一个事务时，MySQL 会自动分配一个该值大小的缓存。

> 注意：设置此值要小心，如果设置太小就是使用磁盘上的临时文件来记录。可以通过 show global status 命令查看 binlog_cache_use 和 binlog_cache_disk_use 的使用情况，来判断当前的 binlog_cache_size 的设置是否合适。

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

目前是 binlog_cahce_disk_use=0，没有使用磁盘上的临时文件，证明 binlog_cache_size 还是够用的。

生产环境中建议一般设置为 1～4MB。

binlog_format 代表二进制日志的格式。前面的章节中简单介绍过，现在详细讲解一下。

binlog 的格式有 statement、row、mixed 三种。

statement：基于操作的 SQL 语句记录到 binlog 中，简称 SBR。MySQL 5.1 之前都是这种默认格式。不建议在生产环境中使用。

其优点如下：

历史悠久、技术成熟。并不需要记录每一条 SQL 语句和每一行的数据变化，减少了 binlog 日志量，节约了 I/O，提高了性能。

缺点：在某些情况下会导致 master-slave 中的数据不一致。

row：基于行的变更情况记录，会记录行变更前的样子及变更后的内容，简称 RBR。生产环境中推荐使用这种 binlog 存储格式。

row 模式的优点：

不记录每条 SQL 语句的上下文信息，仅仅记录哪条数据被修改了，然后修改成什么样了。任何情况下都可以被复制，这对主从复制来说是最安全可靠的。

MySQL 5.6 之后，新增了一个 binlog_rows_query_log_events 参数，设置该参数，也可以在 row 模式下看见用户的完整 SQL 语句。

缺点：会产生大量的日志。

mixed：混合使用 row 和 statement 格式。在 MySQL 5.1 版本的时候是 statement 和 row 的一个过渡格式。不建议使用。

下面通过 mysqlbinlog 命令来查看，分别在 row 和 statement 的二进制格式下，binlog 记录日志的不同方式。

首先在 row 格式下：

```
mysql> show variables like "%binlog_format%";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| binlog_format | ROW   |
+---------------+-------+
1 row in set (0.00 sec)
```

通过向 t 表中更新一条 SQL 语句来看一下 row 格式下的 binlog 的记录方式。t 表的表结构及数据：

```
mysql> CREATE TABLE `t` (
    ->   `id` int(11) NOT NULL auto_increment,
    ->   `name` varchar(20) DEFAULT NULL,
    ->   `city` varchar(10) default null,
    ->   PRIMARY KEY (`id`),
    ->   KEY `b` (`name`)
    -> ) ENGINE=InnoDB auto_increment=1 DEFAULT CHARSET=utf8
    -> ;
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t (name,city) values ('zs','bj'),('zz','sh'),('tzy','gz');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | zs   | bj   |
|  2 | zz   | sh   |
|  3 | tzy  | gz   |
+----+------+------+
3 rows in set (0.00 sec)
```

然后执行一条 update 语句：

```
mysql> update t set name='zz' where name='tzy';
```

二进制文件用 cat、head，或者 tail 命令是无法查看的。所以需要通过 mysqlbinlog 命令来查看二进制日志，把日志转化格式后，输出到 bin.log 文件中：

```
[root@node3 mysql]#
[root@node3 mysql]# /usr/local/mysql/bin/mysqlbinlog --no-defaults -v -v --base64-output=decode-rows
  /data/mysql/mysql-bin.000007 > bin.log
[root@node3 mysql]#
[root@node3 mysql]#
```

参数含义：

- -v 代表可以看到具体的执行信息；
- --base64-output 把二进制日志文件转化格式。

查看 bin.log 文件内容 vim bin.log：

```
### UPDATE `test`.`t`
### WHERE
###   @1=3 /* INT meta=0 nullable=0 is_null=0 */
###   @2='tzy' /* VARSTRING(60) meta=60 nullable=1 is_null=0 */
###   @3='gz' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
###   @1=3 /* INT meta=0 nullable=0 is_null=0 */
###   @2='zz' /* VARSTRING(60) meta=60 nullable=1 is_null=0 */
###   @3='gz' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 14202775
#170913 14:31:31 server id 3  end_log_pos 14202806 CRC32 0x75d23ca0     Xid = 210434
COMMIT/*!*/;
DELIMITER ;
```

可以看到在 row 模式下，记录的是基于行的变更情况，会记录行变更前的样子及变更后的内容。

可以通过之前所学的 set global binlog_format=statement 来修改二进制格式。

在 statement 格式下，binlog 日志如何记录 SQL 呢？

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

同样先执行一条 update 来更新 SQL 语句：

```
mysql> update t set name='zz' where name='tzy';
```

查看转化格式后的 vim bin.log 文件：

```
BEGIN
/*!*/;
# at 14203581
#170913 15:09:30 server id 3  end_log_pos 14203694 CRC32 0xff481059     Query   thread_id=33
 exec_time=0     error_code=0
SET TIMESTAMP=1505286570/*!*/;
update t set name='zz' where name='tzy'
/*!*/;
# at 14203694
#170913 15:09:30 server id 3  end_log_pos 14203725 CRC32 0x31cd669f     Xid = 210454
COMMIT/*!*/;
DELIMITER ;
# End of log file
```

可以看到在 statement 格式下，记录的是一条完整的 SQL 语句。

（3）sync_binlog，该参数影响 binlog 的刷新，前面介绍过了，这里就不再多讲。

（4）expire_logs_days，二进制日志过期时间（单位是天），一般情况下我们可以设置得时间长一点。

（5）binlog-do-db 或者 binlog-ignore-db，表示需要写入或者忽略写入哪些库的日志，默认值为空，表示将所有库的日志同步到二进制日志。

（6）log_slave_updates，该参数在搭建 m->s1->s2 这样的主从架构时，需要在 s1 上配置该参数，才能实现 s1 到 s2 的同步（log_slave_updates=1）。

（7）binlog_checksum，该参数的目的就是写入 binlog 进行校验。有两个值，一个是 none，另一个是 crc32。MySQL 5.6.6 开始，crc32 作为默认校验算法。

（8）log_bin_use_v1_row_events，该参数代表 binlog 的版本信息，从 MySQL 5.6.6 开始默认使用 Version 2 binary log，log_bin_use_v1_row_events 就显示为 off 状态了。

（9）binlog_row_image，该参数有 full、minimal、noblob 三个值。full 代表全部记录，minimal 代表只记录要修改列的记录，noblob 记录除了 blog 和 text 的所有字段。默认使用 full 值。

4.5　慢查询日志（slow log）

慢查询日志可以把超过参数 long_query_time 时间的所有 SQL 语句记录进来，帮助 DBA 人员优化有问题的 SQL 语句（默认是要开启慢查询日志的）。long_query_time 默认是 10s，从 MySQL 5.1 开始，该参数就以微秒为单位记录 SQL 语句的运行时间了，之前仅仅只能使用秒为单位记录。时间的更加精确与细化可以帮助 DBA 更好地去分析与优化主要影响业务的 SQL 语句。

建议线上生产环境中慢查询日志时间的设置可以不用太小，小数点后面 1 位数就可以了（0.1～0.5s 之间都可以）。慢查询日志存储在数据目录下，可以命名为 slow.log。

```
mysql> show variables like '%slow%';
+---------------------------+----------------------+
| Variable_name             | Value                |
+---------------------------+----------------------+
| log_slow_admin_statements | OFF                  |
| log_slow_slave_statements | OFF                  |
| slow_launch_time          | 2                    |
| slow_query_log            | ON                   |
| slow_query_log_file       | /data/mysql/slow.log |
+---------------------------+----------------------+
5 rows in set (0.00 sec)
```

如何查看慢查询日志呢？随着数据量增大，业务增多，可能慢查询日志中记录的慢 SQL 语句就会越来越多。通过 vi 或者 cat 命令查看不能很直观地反映出有哪些慢 SQL。有些人会说可以使用 mysqldumpslow 命令，但用它查看也并不能更方便地帮助 DBA 工作。这里推荐一款工具，叫作 percona-toolkit（一把锋利的瑞士军刀）。这款工具是 MySQL 一个重要分支 Percona 公司的，它是一组命令的集合。我们通过命令集合下的一个 pt-query-digest 命令来捕获线上的慢 SQL 语句，对其进行分析。以一种生成慢 SQL 报告的形式，来帮助 DBA 进行优化 SQL 的工作。

percona-toolkit 工具包的下载地址可以访问：https://www.percona.com/downloads/percona-toolkit/LATEST/。

下载完成之后，直接解压软件包。命令如下：

```
tar -zxvf percona-toolkit-3.0.3_x86_64.tar.gz
```

由于是二进制版本的软件包，解压完成之后，可以直接进入到 percona-toolkit-3.0.3/bin 目录下使用。

下面介绍 pt-query-digest 的一些重要参数的含义：

- --create-review-table，当使用--review 参数把分析结果输出到表中时，如果没有表就自动创建。
- --create-history-table，当使用--history 参数把分析结果输出到表中时，如果没有表就自动创建。
- --filter，对输入的慢查询按指定的字符串进行匹配过滤后再进行分析。
- --limit，限制输出结果百分比或数量，默认值是 20，即将最慢的 20 条语句输出，如果是 50%，则按总响应时间占比从大到小排序，输出总和达到 50%位置时截止。
- --host，数据库服务器地址。
- --user，数据库用户名。
- --password，数据库用户密码。
- --history，将分析结果保存到表中，分析结果比较详细，下次再使用--history 时，如果存在相同的语句，且查询所在的时间区间和历史表中的不同，则会记录到数据表中，可以通过查询同一 CHECKSUM 来比较某类型查询的历史变化。
- --review，将分析结果保存到表中，这个分析只是对查询条件进行参数化，一个类型的查询为一条记录，比较简单。当下次使用--review 时，如果存在相同的语句分析，就不会记录到数据表中。
- --output，分析结果输出类型，值可以是 report（标准分析报告）、slowlog（Mysql slow log）、json、json-anon，一般使用 report，以便于阅读。
- --since，从什么时间开始分析，值为字符串，可以是指定的某个“yyyy-mm-dd [hh:mm:ss]”格式的时间点，也可以是简单的一个时间值：s（秒）、h（小时）、m（分钟）、d（天），如 12h 就表示从 12 小时前开始统计。
- --until，截止时间，配合--since 可以分析一段时间内的慢查询。

通过如下命令生成慢 SQL 报告：

```
/usr/local/percona-toolkit-3.0.3/bin/pt-query-digest --since=24h /data/mysql/slow.log > query.log
```

接下来分析 query.log 日志，报告分成三个部分。

第一部分：

```
# 140ms user time, 0 system time, 22.59M rss, 149.11M vsz
# Current date: Sun Aug 27 16:16:51 2017
# Hostname: node3
# Files: /data/mysql/slow.log
# Overall: 6 total, 4 unique, 0.01 QPS, 0.07x concurrency _________________
# Time range: 2017-08-27 16:05:52 to 16:16:09
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time            42s     98ms     40s      7s     39s     14s    580ms
# Lock time          388us       0     81us     64us     80us     28us     76us
# Rows sent          97.66k       0  48.83k  16.28k  46.68k  22.00k       0
# Rows examine      195.31k       0  48.83k  32.55k  46.68k  22.00k  46.68k
# Query size           341      16     252   56.83  246.02   85.28   17.65
```

这部分介绍了总体统计之后的结果。

- Overall：总共有多少条查询，上例为总共有 6 个查询。
- Time range：查询执行的时间范围（自己做实验，就临时产生的 SQL）。
- unique：唯一查询数量，即对查询条件进行参数化后，总共有多少个不同的查询，该实验是 4 个。
- total：总计时间（42s）；min：最小时间（98ms）；max：最大时间（40s）；avg：平均时间（7s）。
- 95%：把所有值从小到大排列，位置位于 95%的那个数一般最具有参考价值。
- median：中位数，把所有值从小到大排列，位置位于中间那个数。

第二部分是 SQL 语句的一个占比结果展示：

```
# Profile
# Rank Query ID           Response time Calls R/Call  V/M   Item
# ==== ================== ============= ===== ======= ===== ==============
#    1 0x040ADBE3A1EED0A2 40.3101 95.9%     1 40.3101  0.00 CALL insert_su
# MISC 0xMISC                1.7312  4.1%     5  0.3462   0.0 <3 ITEMS>
```

- Response：总的响应时间。
- time：该查询在本次分析中总的时间占比。
- calls：执行次数，即本次分析总共有多少条这种类型的查询语句。
- R/Call：平均每次执行的响应时间。
- Item：查询对象，即具体的 SQL 语句。

第三部分就是 SQL 语句的详细输出结果：

```
# Query 1: 0 QPS, 0x concurrency, ID 0x040ADBE3A1EED0A2 at byte 5194 _____
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.00
# Time range: all events occurred at 2017-08-27 16:06:58
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         16       1
# Exec time     95     40s     40s     40s     40s     40s       0     40s
# Lock time     20     81us    81us    81us    81us    81us      0     81us
# Rows sent      0        0       0       0       0       0       0       0
# Rows examine   0        0       0       0       0       0       0       0
# Query size     6       21      21      21      21      21       0      21
# String:
# Databases    test
# Hosts        localhost
# Users        root
# Query_time distribution
#   1us
#  10us
# 100us
#   1ms
#  10ms
# 100ms
#    1s
#  10s+  ##############################################################
call insert_su(50000)\G
```

- Databases：库名。
- Users：各个用户执行的次数（占比）。
- Query_time distribution：查询时间分布，长短体现区间占比，这个例子中 10s 以上的查询占有比例很高。

4.6 全量日志（general log）

general log 会记录 MySQL 数据库所有操作的 SQL 语句，包含 select 和 show。一般情况下是不会开启该功能的（默认是关闭的），因为 log 的量会非常庞大，但个别情况下可能会临时开一下，用于故障检测。

这里介绍 log_output 参数。

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

log_output：全局动态变量，可取 FILE、TABLE、NONE 三个值。其中 FILE 这种存储方式可以很方便地按条件检索。若指定为 NONE，那么即使 general_log 开启了也不会记录 log。若 log_output 指定为 TABLE，则会在 MySQL 数据库下创建一个 general_log 表。需要注意的是，

该参数不仅仅影响 general 的存储方式，还影响 slow log 的存储方式。这里建议使用 FILE 这种方式存储。

4.7　审计日志（audit log）

数据库审计（简称 DBAudit）能够实时记录网络上的数据库活动，对数据库操作进行细粒度审计的合规性管理，对数据库遭受到的风险行为进行告警，对攻击行为进行阻断。它通过对用户访问数据库行为的记录、分析和汇报，用来帮助用户事后生成合规报告、事故追根溯源，同时加强内外部数据库网络行为记录，提高数据资产安全。

MySQL 数据库官方的收费组件需要购买企业版才可以使用审计功能。下面利用第三方开源审计插件 libaudit_plugin.so 在 MySQL 5.7 社区版中完成审计工作。

插件下载地址：https://bintray.com/mcafee/mysql-audit-plugin/release/1.1.4-725#files

解压插件包：

```
unzip audit-plugin-mysql-5.7-1.1.4-725-linux-x86_64.zip
```

把解压好的插件复制到 MySQLlib 库的插件目录下：

```
cd audit-plugin-mysql-5.7-1.1.4-725/lib
cp libaudit_plugin.so /usr/local/mysql/lib/plugin/
```

数据库命令行安装插件的过程：

```
INSTALL PLUGIN AUDIT SONAME 'libaudit_plugin.so';
```

查看插件功能是否开启：

```
show variables like '%audit%';
```

开启插件功能：

```
set global  audit_json_file=1;
```

在 MySQL 数据目录下，会多出一个 mysql-audit.json 审计日志。

查看 mysql-audit.json 文件，可以找到操作 SQL 语句的用户名、IP 地址，这可以让在数据库上做了坏事又不承认的人无法赖账，起到了对操作数据库很好的监控效果。

4.8 中继日志（relay log）

主从复制中，从服务器上一个很重要的文件。从服务器 I/O 线程将主服务器的二进制日志读取过来并记录到从服务器本地文件（relay log）中，然后从服务器上的 SQL 线程会读取 relay-log 日志的内容并应用到从服务器。这部分内容在讲到主从复制的时候再详细介绍。

4.9 Pid 文件

MySQL 数据库是一个单进程多线程模型的数据库，实例启动完成后，会将自己唯一进程号记录到自己 Pid 文件中。

```
[root@node3 bin]# ps -ef|grep mysql
root      5912  5833  0 13:07 pts/1    00:00:00 /bin/sh /usr/local/mysql/bin/mysqld_safe
ults-file=/etc/my.cnf
mysql     6577  5912  0 13:07 pts/1    00:00:42 /usr/local/mysql/bin/mysqld --defaults-fi
c/my.cnf --basedir=/usr/local/mysql --datadir=/data/mysql --plugin-dir=/usr/local/mysql/l
gin --user=mysql --log-error=/data/mysql/error.log --open-files-limit=3072 --pid-file=/da
ql/node3.pid --socket=/tmp/mysql.sock --port=3306
root      6918  5833  0 15:32 pts/1    00:00:00 grep mysql
[root@node3 bin]#
```

Pid 文件存放在数据目录（/data/mysql/）下。命名规则是将主机名作为前缀。

4.10 Socket 文件

MySQL 数据库有两种连接方式，网络连接和本地连接。

mysql.sock 文件是服务器与本地客户端进行通信的 UNIX 套接字文件，其默认位置是 /tmp/mysql.sock。

```
[root@node3 bin]#
[root@node3 bin]# ps -ef|grep mysql
root      5912  5833  0 13:07 pts/1    00:00:00 /bin/sh /usr/local/mysql/bin/mysqld_safe
ults-file=/etc/my.cnf
mysql     6577  5912  0 13:07 pts/1    00:00:42 /usr/local/mysql/bin/mysqld --defaults-fi
c/my.cnf --basedir=/usr/local/mysql --datadir=/data/mysql --plugin-dir=/usr/local/mysql/l
gin --user=mysql --log-error=/data/mysql/error.log --open-files-limit=3072 --pid-file=/da
ql/node3.pid --socket=/tmp/mysql.sock --port=3306
root      6918  5833  0 15:32 pts/1    00:00:00 grep mysql
[root@node3 bin]#
```

4.11 表结构文件

MySQL 8.0 之前，以.frm 结尾的文件为表结构文件。从 MySQL 8.0 版本开始，frm 表的定义文件被消除掉，把文件中的数据都写到了系统表空间，通过利用 InnoDB 存储引擎实现表 DDL

语句操作的原子性（在之前版本中是无法实现表 DDL 语句操作的原子性的，如 truncate 无法回滚）。

4.12　InnoDB 存储引擎文件

在 InnoDB 存储引擎层面主要分为两种日志，一种是 redo 日志，另外一种就是 undo 日志。前面在介绍 InnoDB 特点的时候说过，InnoDB 支持事务，支持 MVCC 多版本并发控制。InnoDB 的多版本是通过使用 undo 和回滚段来实现的。InnoDB 是索引组织表，每行记录都实现了三个隐藏字段：DB_ROW_ID、DB_TRX_ID、DB_ROLL_PTR。除 DB_ROW_ID 外，DB_TRX_ID 和 DB_ROLL_PTR 分别代表每行记录的事务 ID 和每行记录的回滚指针。InnoDB 有一个全局的事务链表，每个事务的开始都会把事务 ID 放到链表中。DB_ROLL_PTR 指针用于指向 undo 记录，构造多版本。

redo log 用于记录事务操作变化，记录的是数据被修改之后的值。前面的章节中已经介绍它的工作方式：刷新机制。本节主要介绍 undo 日志文件。

undo 日志文件

对记录做变更操作时不仅会产生 redo 记录，也会产生 undo 记录（insert、update、delete）。但 undo 只记录变更前的旧数据，undo 记录默认被记录到系统表空间（ibdata1）中，但是从 MySQL 5.6 开始，就可以使用独立的 undo 表空间了。采用独立 undo 表空间，再也不用担心 undo 会把 ibdata1 文件弄大；也给我们部署不同 I/O 类型的文件位置带来便利，对于并发写入型负载，我们可以把 undo 文件部署到单独的高速存储设备上。

以下为涉及 undo log 的主要参数。

innodb_undo_directory：undo 文件的存储目录。

```
root@db 12:22:  [(none)]>  show variables like '%undo_directory%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_directory    | ./    |
+--------------------------+-------+
1 row in set (0.00 sec)
```

undo 回滚段的数量默认是 128 个。

```
mysql> show variables like '%undo_logs%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| innodb_undo_logs  | 128   |
+-------------------+-------+
1 row in set (0.00 sec)
```

通过 innodb_undo_logs 参数控制，可以将一个大的回滚段拆分成多个小的回滚段。每个 undo

log segments 最多存放 1024 个事务。

innodb_undo_tablespaces 代表 undo tablespace 的个数。表空间中有 undo log 文件，默认大小是 10MB，表空间个数的默认值为 0。

```
root@db 12:06:  [(none)]> show variables like '%undo_tablespaces%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_tablespaces  | 0     |
+--------------------------+-------+
1 row in set (0.01 sec)
```

undo tablespace 的数量最少为 2 个，因为 undo log 的 truncate 操作由 purge 协调线程发起，在 truncate 某个 undo log 表空间的过程中，保证有一个可用的 undo log tablespace 能提供给用户使用，从而实现所谓的在线 truncate。

MySQL 5.7 之后又多了一个 innodb_max_undo_log_size 参数，默认大小是 1GB。

```
root@db 12:06:  [(none)]> show variables like '%innodb_max_undo%';
+--------------------------+------------+
| Variable_name            | Value      |
+--------------------------+------------+
| innodb_max_undo_log_size | 1073741824 |
+--------------------------+------------+
1 row in set (0.00 sec)
```

这个参数用来控制最大 undo tablespace 文件的大小，超过这个阈值，就会触发 truncate undo logs，truncate 后的 undo logs 大小默认恢复为 10MB。而且 MySQL 5.7.5 之后可以支持在线删除无用的 undo logs，默认功能是关闭的。

```
root@db 15:37:  [(none)]> show variables like '%innodb_undo_log_truncate%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_log_truncate | OFF   |
+--------------------------+-------+
1 row in set (0.00 sec)
```

只需要把 innodb_undo_log_truncate=OFF 改成 on 就可以了。

innodb_purge_rseg_truncate_frequency 也是 MySQL 5.7 之后新增的参数，作用是控制回收 undo log 的频率，默认值是 128。

```
root@db 15:37:  [(none)]> show variables like '%purge_rseg%';
+-----------------------------------------+-------+
| Variable_name                           | Value |
+-----------------------------------------+-------+
| innodb_purge_rseg_truncate_frequency    | 128   |
+-----------------------------------------+-------+
1 row in set (0.00 sec)
```

表示 purge undo 轮询 128 次后，进行一次 undo 的 truncate 操作。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 的设定值。

第5章 表

数据库其实就是一个有好多表的大集合。在创建表的时候，首先要保证几点原则，禁止使用中文做字段名；禁止使用字符型做主键；禁止无主建或是唯一索引的表出现。可以把这几点列为开发人员的行为规范。方便后期给开发人员做数据库的相关培训做好准备。在为表设计字段时，就要考虑为该字段选择合适的数据类型。

MySQL 主要的数据类型可以分为整型、浮点型、字符类型和日期时间类型。MySQL 允许我们指定数值字段中的值是否有正负之分或者用零填补。

> 注：在选择数据类型时，要秉承最小、最合适的原则去选择。

5.1 整型

整型类如表 5-1 所示，int、tinyint 是使用最多的整型类型。

表 5-1 整型类型

类　　型	大　　小	范围（有符号）	范围（无符号）	用　　途
Tinyint	1 字节	（-128,127）	（0,255）	小整数值
Smallint	2 字节	（-32768,32767）	（0,65535）	大整数值
Mediumint	3 字节	（-8388608,8388607）	（0,16777215）	大整数值

续表

类　　型	大　　小	范围（有符号）	范围（无符号）	用　　途
Int	4 字节	（-2147483648,2147483647）	（0,4294967295）	大整数值
Bigint	8 字节	（-9233372036854775808,9223372036854775807）	（0,18446744073709551615）	极大整数值

我们经常能看到一些数据表中选择 id 字段作为主键字段，而 id 的数据类型一般都选择 int，基本上不使用 bigint 这种超大整数值，因为 int unsigned 数值范围可以达到 42 亿，肯定够业务上的应用了。为什么非要选择 id 做主键呢？其实选择哪个字段当主键也是有要求的，主键字段要选择那种不经常修改的，尽量要与业务无关的，没有什么具体含义的。因为 InnoDB 表是索引组织表，需要保证索引结构不经常翻转，避免造成性能的消耗。

接下来再问一个问题，int(4)和 int(10)有区别吗？在生产环境中，有些开发人员一会写成 int(1)，一会写成 int(8)。于是我就问，为什么这么写呢？回答：int(1)代表能存 1 位，int(8)代表能存 8 位。其实这种说法是大错特错的。Int(n)括号里面的数字 n 无论写成多少，都是占 4 个字节的空间，最多能存 10 位数。n 不是代表能存多少位数，只是显示宽度。所以 int(4)和 int(10)没有区别。

但如果定义了 zerofill，int(4)中的 4 就有意义了，假如我们写入一个数字 1，它会补充完成写成 0001，int(10)则为 0000000001。

```
mysql> desc tt;
+-------+-----------------------------+------+-----+---------+----------------+
| Field | Type                        | Null | Key | Default | Extra          |
+-------+-----------------------------+------+-----+---------+----------------+
| id    | int(10) unsigned zerofill   | NO   | PRI | NULL    | auto_increment |
| name  | varchar(10)                 | YES  |     | NULL    |                |
+-------+-----------------------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> insert into tt (id,name) values (1,'zs');
Query OK, 1 row affected (0.00 sec)

mysql> select * from tt;
+------------+------+
| id         | name |
+------------+------+
| 0000000001 | zs   |
+------------+------+
1 row in set (0.01 sec)
```

```
mysql> desc tt;
+-------+-----------------------------+------+-----+---------+----------------+
| Field | Type                        | Null | Key | Default | Extra          |
+-------+-----------------------------+------+-----+---------+----------------+
| id    | int(4) unsigned zerofill    | NO   | PRI | NULL    | auto_increment |
| name  | varchar(10)                 | YES  |     | NULL    |                |
+-------+-----------------------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)
```

```
mysql> insert into tt (id,name) values (1,'zs');
Query OK, 1 row affected (0.00 sec)

mysql> select * from tt;
+------+------+
| id   | name |
+------+------+
| 0001 | zs   |
+------+------+
1 row in set (0.00 sec)
```

5.2 浮点型

浮点型如表 5-2 所示。

表 5-2 浮点型

类　　型	大　　小	范围（有符号）	范围（无符号）	用　　途
Float	4 字节	（-3.402 823 466 E+38，1.175 494 351 E-38），0，（1.175 494 351 E-38，3.402 823 466 351 E+38）	0，(1.175 494 351 E-38，3.402 823 466 E+38)	单精度
Double	8 字节	（1.797 693 134 862 315 7 E+308，2.225 073 858 507 201 4 E-308），0，（2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308）	0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308)	双精度
Decimal	DECIMAL(M,D)，如果 M>D，为 M+2，否则为 D+2	依赖于 M 和 D 的值	依赖于 M 和 D 的值	小数值

避免使用浮点类型，因为它属于并不精确的类型，在生产中不建议使用 float 和 double。所以这里就不做过多介绍了。

在生产环境中，我们大多使用 decimal 来存储金钱字段，但是数值在运算过程中还是会转成浮点来运算，而且在运算过程中会出现四舍五入的情况，这样就造成了金额的不准确。这里做一个 decimal 的相关实验，详细介绍一下它的使用。

创建一张测试表 ttt，其中 salary 字段为 decimal(6,2)。

> 注：6 代表整数部分，2 代表小数点后面保留位数。

```
root@db 22:25:  [zs]> CREATE TABLE `ttt` (
    ->   `id` int(11) NOT NULL,
    ->   `salary` decimal(6,2) DEFAULT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.33 sec)

root@db 22:25:  [zs]> desc ttt;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| id     | int(11)      | NO   |     | NULL    |       |
| salary | decimal(6,2) | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

然后往表里插入数据：

```
root@db 22:25:  [zs]> insert into ttt(id,salary) values(1,123456);
ERROR 1264 (22003): Out of range value for column 'salary' at row 1
```

可以看到对 salary 字段插入了长度为 6 的数字，出现了超过范围的报错。如果对 salary 字段插入长度为 4 的数字，插入成功并没有报错，并在末尾补齐两位小数。

```
root@db 22:28:  [zs]> insert into ttt(id,salary) values(1,1234);
Query OK, 1 row affected (0.01 sec)

root@db 22:36:  [zs]> select * from ttt;
+----+---------+
| id | salary  |
+----+---------+
|  1 | 1234.00 |
+----+---------+
1 row in set (0.00 sec)
```

继续往 ttt 表中插入数据，这次是小数点后面 5 位数。

```
root@db 15:50:  [zs]> insert into ttt (id,salary) values (2,1234.12745);
Query OK, 1 row affected, 1 warning (0.01 sec)

root@db 15:51:  [zs]> show warnings;
+-------+------+-----------------------------------------------+
| Level | Code | Message                                       |
+-------+------+-----------------------------------------------+
| Note  | 1265 | Data truncated for column 'salary' at row 1 |
+-------+------+-----------------------------------------------+
1 row in set (0.00 sec)

root@db 15:51:  [zs]> select * from ttt;
+----+---------+
| id | salary  |
+----+---------+
|  1 | 1234.00 |
|  2 | 1234.13 |
+----+---------+
2 rows in set (0.00 sec)
```

可见 salary 字段的数据类型 decimal 定义的是小数点后面保存 2 位小数，但实际插入的是小数点后面 5 位数，结果显示被截断了，并且采用了四舍五入的原则，由于第三位小数是 7，所以四舍五入后，最后插入的值变成了 1234.13。

> 总结：DECIMAL(M,D)中的 D 值是小数部分的位数，若插入的值未指定小数部分或者小数部分不足 D 位，则自动补到 D 位小数，若插入的值小数部分超过了 D，则会发生截断，截取前 D 位小数，并四舍五入。M 值是整数部分加小数部分的总长度，即插入的数字整数部分不能超过 M−D 位，否则不能成功插入，会报超出范围的错误。

对于交易类的平台，这种四舍五入的现象还是要避免出现的。可以使用 int 来存储金钱，让 int 单位为分，这样就不存在四舍五入了，让存的数值更精确。

5.3 时间类型

时间类型如表 5-3 所示。

表 5-3 时间类型

类　　型	大小（字节）	范　　围	格　　式	用　　途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8/5	1000-01-01 00:00:00/ 9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00/2038 年某时	YYYYMMDD HHMMSS	混合日期和时间值，时间戳

我们有时会纠结到底是选择 datetime 还是 timestamp。之前在介绍 explicit_defaults_for_timestamp 参数时，详细说过 timestamp。这里主要强调一下 datetime，datetime 数据类型在 MySQL 5.6 之前是占 8 个字节，5.6 版本之后占 5 个字节。datetime 的可用范围比 timestamp 大，物理存储上仅比 timestamp 多占 1 个字节的空间，整体性能上的消耗并不算太大。因此在生产环境可以使用 datetime 时间类型。当然也可以使用 int 类型来存储时间。可以通过两个函数转换而来：unix_timestamp 和 from_unixtime。

```
mysql> select unix_timestamp('2017-08-29 14:09:11');
+---------------------------------------+
| unix_timestamp('2017-08-29 14:09:11') |
+---------------------------------------+
|                            1503986951 |
+---------------------------------------+
1 row in set (0.01 sec)

mysql> select from_unixtime(1503986951);
+---------------------------+
| from_unixtime(1503986951) |
+---------------------------+
| 2017-08-29 14:09:11       |
+---------------------------+
1 row in set (0.00 sec)
```

注：Timestamp 和 datetime 从 MySQL 5.6 开始都支持自动更新为当前的时间。

5.4 字符串类型

字符串类型如表 5-4 所示。

表 5-4 字符串类型

类　　型	大小（字节）	用　　途
Char	0～255	定长字符串
Varchar	0～255（65535）	变长字符串
Tinyblob	0～255	不超过 255 个字符的二进制字符串
Tinytext	0～255	短文本字符串
Blob	0～65535	二进制形式的长文本数据
TEXT	0～65535	长文本数据
MEDIUMBLOB	0～16777215	二进制形式的中等长度文本数据
MEDIUMTEXT	0～16777215	中等长度文本数据
LOGNGBLOB	0～4 294 967 295	二进制形式的极大文本数据
LONGTEXT	0～4 294 967 295	极大文本数据

text 和 blob 这种存大量文字或者存图片的大数据类型建议不要与业务表放在一起。

> 注：主要业务表切忌出现这样类型的字段。

面试中的问得最多的可能就是 Char 和 Varchar 的区别了。

Char 类型用于定长字符串，并且大小范围为 0～255。如果字符数没有达到定义的位数，会在后面用空格补全存入数据库中；如果超过指定长度大小，会被截断。

Varchar 是变长长度，长度范围为 0～65535，存储时，如果字符没有达到定义的位数，不会在后面补空格；如果超过指定长度，也会被截断。

Varchar 类型可以根据实际内容动态改变存储值的长度，在不能确定字段需要多少字符时，使用 Varchar 类型可以大大地节约磁盘空间，提高存储效率。

> 注：使用 Varchar 时，和输入的字符数有关，会多一到两个字节来记录字节长度，当数据位占用的字节数小于 255 时，用 1 个字节来记录长度，数据位占用字节数大于 255 时，用 2 个字节来记录长度，还有一位用来记录是否为 nul 值。

例如，varchar(100)里面的这个 100 代表的是字符的概念。

- 在 UTF8 字符集下，存储空间为 100×3+1=301 个字节；

- 在 GBK 字符集下，存储空间为 100×2+1=201 个字节。

Char 和 Varchar 的存储空间比较（在 latin1 字符集下）如表 5-5 所示。

表 5-5 Varchar 与 Char 的存储空间区别

Value	Char(4)	存储空间	Varchar(4)	存储空间
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3bytes
'abcd'	'abcd'	4 bytes	'abcd'	5bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5bytes

MySQL 每一行的最大字节数为 65535，使用 UTF8 字符集，每个字符最多占 3 个字节，最大长度不能超过（65535-1-2）/3=21844。

如果使用 GBK 字符集，每个字符最多占 2 个字节，最大长度不能超过（65535-1-2）/2=36766。

IPv4 这样的字段，选择什么数据类型存储合适呢？可能有读者会说选择 Varchar 存储就行了。这里推荐使用 int 类型来存储 IP 字段。可 int 不是存整数的嘛，怎么可以存字符串呢？这里就需要使用到 inet_aton 和 inet_ntoa 两个函数。

```
root@db 23:32:  [zs]> select inet_aton('192.168.56.102');
+-----------------------------------+
| inet_aton('192.168.56.102') |
+-----------------------------------+
|                  3232249958 |
+-----------------------------------+
1 row in set (0.00 sec)

root@db 23:32:  [zs]>
root@db 23:32:  [zs]> select inet_ntoa(3232249958);
+-------------------------+
| inet_ntoa(3232249958) |
+-------------------------+
| 192.168.56.102        |
+-------------------------+
1 row in set (0.00 sec)
```

5.5 字符集

简单地说，字符集就是一套文字符号及其编码，是比较规则的集合。MySQL 数据库字符集包括字符集（character）和校对规则（collation）两个概念。其中，字符集用来定义 MySQL 数据字符串的存储方式，而校对规则则是定义比较字符串的方式。

常用的字符集有 GBK、Latin1、UTF8、UTF8mb4。Latin1 目前已经不使用了，MySQL 5.0 或者 MySQL 5.1 中，Latin1 是数据库默认的字符集，一个汉字或者字母占 1 个字节大小。GBK 占 2 个字节，通用性没有 UTF8 好。UTF8 占 3 个字节，UTF8mb4 是 UTF8 的超集，占 4 个字

节。针对 MySQL 5.7 版本，建议使用 UTF8mb4 字符集。

字符集涉及的范围也很广，像程序连接配置、数据库、服务器、表，还有表中的字段、结果集，都会用到字符集。字符集最容易出现的问题就是中文乱码。

那么如何避免这个问题呢？我们只要保证三线统一就可以了。

首先连接终端的字符集必须是 UTF8。

操作系统的字符集必须是 UTF8，查看 Linux 操作系统字符集的方法：

```
[root@node3 ~]# cat /etc/sysconfig/i18n
LANG="en_US.UTF-8"
SYSFONT="latarcyrheb-sun16"
```

MySQL 数据库的字符集必须是 UTF8，查看方法：

```
root@db 15:56:  [zs]> \s;
--------------
mysql  Ver 14.14 Distrib 5.7.14, for linux-glibc2.5 (x86_64) using  EditLine wrapper

Connection id:          10
Current database:       zs
Current user:           root@localhost
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server version:         5.7.14-log MySQL Community Server (GPL)
Protocol version:       10
Connection:             Localhost via UNIX socket
Server characterset:    utf8mb4
Db     characterset:    utf8mb4
Client characterset:    utf8
Conn.  characterset:    utf8
```

或者可以通过 show variables like "char"来查看数据库字符集的配置，只需要在配置文件中的 mysqld 下加入 character-set-server = utf8mb4 就可以了。

```
root@db 15:57:  [zs]> show variables like '%char%';
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_set_client     | utf8                                                     |
| character_set_connection | utf8                                                     |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_results    | utf8                                                     |
| character_set_server     | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_sets_dir       | /usr/local/mysql-5.7.14-linux-glibc2.5-x86_64/share/charsets/ |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

```
[mysqld]
user    = mysql
port    = 3306
basedir = /usr/local/mysql
datadir = /data/mysql/
socket  = /tmp/mysql.sock
pid-file = db.pid

character-set-server = utf8mb4
```

> 注：想要临时修改数据库字符集的方式，可以在数据库命令行执行 set names 字符集名称，例如 set names utf8。

三者统一都是 UTF8，就不会出现中文乱码的情况。

5.6 表碎片产生的原因

我们有时会在表中删除一些大量的无用数据，但发现数据文件的大小并没有减小，这是因为删除后在数据文件中遗留了大量的数据碎片所导致的。先看一幅图，如图 5-1 所示。

图 5-1 示意图

图 5-1 分为两部分阴影部分和实体部分，分别用 1 和 2 表示。1 是删除（delete）的表中数据，2 是表中剩余的数据。删除之后，当再次扫描全表时，是只遍历第二部分呢，还是会遍历 1+2 部分呢？

其实答案很明显，肯定会全部扫描，可阴影部分 1 不是都删除了吗，为什么还会继续扫描呢？

因为使用 delete 删除数据的时候，MySQL 并不会把数据文件真实删除，而只是将数据文件的标识位删除，也没有整理数据文件，因此不会彻底释放表空间。换句话说，每当我们从表中删除数据时，这段被删除数据的空间就会被留出来，如果又赶上某段时间内对该表进行大量的 delete 操作，那么这部分被删除数据的空间就会越来越大。当有新数据写入时，MySQL 会再次利用这些被删除的区域，但也无法彻底占用。

delete 删除操作会产生数据碎片，这些碎片会占用硬盘空间。这种额外的破碎的存储空间

在读取效率方面比正常占用的空间要低很多。所以我们需要对表进行优化，对表进行碎片整理工作。

5.7 碎片计算方法及整理过程

要整理碎片，首先要了解碎片的计算方法，通过 show table status like"%table_name%"命令来查看：

```
mysql> show table status like '%su%'\G;
*************************** 1. row ***************************
           Name: su
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 98546
 Avg_row_length: 261
    Data_length: 25739264
Max_data_length: 0
   Index_length: 30294016
      Data_free: 5242880
 Auto_increment: 2668201
    Create_time: 2017-09-13 16:05:27
    Update_time: NULL
     Check_time: NULL
```

- 碎片大小=数据总大小-实际表空间文件大小。
- 数据总大小=data_length+index_length= 56033280。
- 实际表空间文件大小=rows×avg_row_length= 25720506。
- 碎片大小=（56033280-25720506）/1024/1024，大约等于 28.9MB。

清除碎片的两种方法：

（1）alter table table_name engine=innodb。

（2）备份原表数据，然后删掉，重新导入到新表中（与原表结构一样）。

第二种整理方式在讲到备份恢复时会用到。

执行第一种方式来整理碎片：

```
mysql> alter table su engine=innodb;
Query OK, 100000 rows affected (6.90 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

这条语句的作用就是重新整理一遍全表数据，整理之后的数据连续性好，全表扫描变快，表空间文件也变小了，节约了磁盘上空间，清除了碎片。缺点是需要先给整表加个写锁，需要经历比较长的时间，例子中表的数据量才 10 万条 SQL 就耗时近 7s，在业务高峰期不建议使用。

这里建议大家使用之前介绍过的 Percona 公司的 percona-toolkit 工具集。使用 pt-query-digest 命令来捕获线上的慢 SQL 语句是一种很好的查看慢查询日志的方式。

下面再介绍工具集中的一个命令 pt-online-schema-change，其可以在线整理表结构、收集碎片、给大表添加字段和索引，避免出现锁表导致阻塞读写的操作。MySQL 5.7 版本不需要使用这个命令了，因为可以直接在线“Online DDL”。

操作过程如下：

```
    ./pt-online-schema-change --user=root --password=root123 --host=localhost
--alter="ENGINE=Innodb" D=test,t=su
    --execute
```

```
Altering `test`.`su`...
Creating new table...
Created new table test._su_new OK.
Altering new table...
Altered `test`.`_su_new` OK.
2017-09-13T16:14:04 Creating triggers...
2017-09-13T16:14:04 Created triggers OK.
2017-09-13T16:14:04 Copying approximately 94674 rows...
2017-09-13T16:14:11 Copied rows OK.
2017-09-13T16:14:11 Analyzing new table...
2017-09-13T16:14:11 Swapping tables...
2017-09-13T16:14:11 Swapped original and new tables OK.
2017-09-13T16:14:11 Dropping old table...
2017-09-13T16:14:11 Dropped old table `test`.`_su_old` OK.
2017-09-13T16:14:11 Dropping triggers...
2017-09-13T16:14:11 Dropped triggers OK.
Successfully altered `test`.`su`.
```

5.8 表统计信息

统计信息就是可以统计每个库的大小、表的大小、数据和索引的大小等。作为一名优秀的 MySQL DBA，需要有很强的 SQL 语句的功底，SQL 语句会按照执行计划去执行，如果统计信息不准确，会导致执行计划是错误的，会给我们优化 SQL 的工作带来不便。所以表统计信息的收集是十分重要的。

以下列举了几个语句，用来统计数据库中的信息情况。

统计每个库大小，SQL 如下：

```
    SELECT TABLE_SCHEMA,SUM(DATA_LENGTH)/1024/1024/1024 as DATA_LENGTH,
SUM(INDEX_LENGTH)/1024/1024/1024
    as INDEX_LENGTH,SUM(DATA_LENGTH+INDEX_LENGTH)/1024/1024/1024 as SUM_DATA_
INDEX FROM information_schema.TABLES
```

```
    WHERE TABLE_SCHEMA!='information_schema' AND TABLE_SCHEMA!='mysql' GROUP BY
TABLE_SCHEMA;
```

```
+--------------+-------------+--------------+---------------+
| TABLE_SCHEMA | DATA_LENGTH | INDEX_LENGTH | SUM_DATA_INDEX |
+--------------+-------------+--------------+---------------+
| performance_schema | 0.000000000000 | 0.000000000000 | 0.000000000000 |
| test | 0.048461914063 | 0.031402587891 | 0.079864501953 |
+--------------+-------------+--------------+---------------+
```

统计库中每个表的大小，SQL 如下所示。这里统计的是 test 库下表的大小。

```
    SELECT TABLE_NAME,DATA_LENGTH,INDEX_LENGTH,SUM(DATA_LENGTH+INDEX_LENGTH)
AS TOTAL_SIZE
    FROM information_schema.TABLES
    WHERE TABLE_SCHEMA='test' GROUP BY TABLE_NAME;
```

```
+------------+-------------+--------------+------------+
| TABLE_NAME | DATA_LENGTH | INDEX_LENGTH | TOTAL_SIZE |
+------------+-------------+--------------+------------+
| sbtest     |    22593536 |      2342912 |   24936448 |
| su         |    26787840 |     31375360 |   58163200 |
| t          |     2637824 |            0 |    2637824 |
| zs         |       16384 |            0 |      16384 |
+------------+-------------+--------------+------------+
4 rows in set (0.00 sec)
```

统计所有数据库的大小，SQL 如下：

```
    select sum(data_length+index_length)/1024/1024/1024 from information_
schema.tables;
```

```
+-----------------------------------------------------+
| sum(data_length+index_length)/1024/1024/1024        |
+-----------------------------------------------------+
|                                      0.080626926385 |
+-----------------------------------------------------+
1 row in set (0.01 sec)
```

5.9 统计信息的收集方法

（1）遍历 information_schema.tables。收集 su 表的统计信息：

> 注：Table_schema 是数据表所属的数据库名，Table_name 是表名称。

```
mysql> select * from information_schema.tables where table_name='su'\G;
*************************** 1. row ***************************
  TABLE_CATALOG: def
   TABLE_SCHEMA: test
     TABLE_NAME: su
     TABLE_TYPE: BASE TABLE
         ENGINE: InnoDB
        VERSION: 10
     ROW_FORMAT: Compact
     TABLE_ROWS: 98456
 AVG_ROW_LENGTH: 272
    DATA_LENGTH: 26787840
MAX_DATA_LENGTH: 0
   INDEX_LENGTH: 31375360
      DATA_FREE: 5242880
 AUTO_INCREMENT: 2668201
    CREATE_TIME: 2017-09-13 16:14:11
    UPDATE_TIME: NULL
     CHECK_TIME: NULL
```

本例是收集 su 这张表的统计信息。可以设置一个定时任务，每天去收集一下表的统计信息。

（2）重启 MySQL 实例。

（3）show table status like‘%table_name%’。

5.10 MySQL 库表常用命令总结

- use database：选择你所创建的数据库；
- show databases：查看所有数据库；
- show tables：查看某库下所有的表；
- create database database_name：创建数据库；
- drop database database_name：删除数据库；
- create table table_name（字段列表）：创建表；
- drop table table_name：删除表（表结构也被删除）；
- delete from table_name (where)或者 truncate table table_name：只删除表数据；
- insert into table_name（字段列表）values（对应字段的值）：往表中插入数据；
- update table_name set:字段名=某值（where）：更新表中某行数据；
- select * from table_name（where）：查看表中数据；
- show create table table_name\G：查看建表语句；
- desc table_name：查看表结构；
- show table status：获取表基础信息；
- show index from table_name：查看当前表下索引的情况；
- show full processlist：查看数据库当前连接的情况。

第6章 索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可以快速访问数据库表中的特定信息。索引就好比书的目录，通过目录你可以快速搜索到想要查找的内容。

在工作中，往往出现这种情况，开发人员很懂业务，但是在项目初期设计表的过程中，根本没有给经常访问的字段添加索引这个概念。而 DBA 的数据库专业知识很强，但不了解业务，也不知道哪个字段会经常用于查询，会经常被访问到。业务人员与 DBA 这两个必须联系起来的伙伴，却彼此谁都不了解谁。我经常对我的学生说，不懂业务的 MySQL DBA 永远不是一个合格的 DBA，职业发展也不会太长远。身为 DBA 要在项目初期就配合开发人员一起参与到表设计当中去，把该建索引的字段加上索引，减少后期出现表数据量增大，再添加索引的不便性。DBA 也要经常给开发人员培训一些索引的知识，让他们知道索引的利弊。索引利用好了就是一辆“法拉利跑车”，使用不好就是一辆破旧的“三轮车”。

学完此章会让大家对索引有一个由浅到深的认识，对索引结构更加了解。MySQL 数据库两个主要的索引是 B+tree 索引和哈希索引。

6.1 二叉树结构

B+tree 是由二叉树→平衡二叉树→B-tree 演化而来的。先来逐一介绍这三种树状结构，以便更好地了解 B+tree 的结构。

二叉树的每个结点至多有两个子结点（二棵子树），二叉树的子树有左右序之分，次序不能

颠倒。在二叉树中，左子树的键值永远比右子树的小，并且小于根键值，如图 6-1 所示。

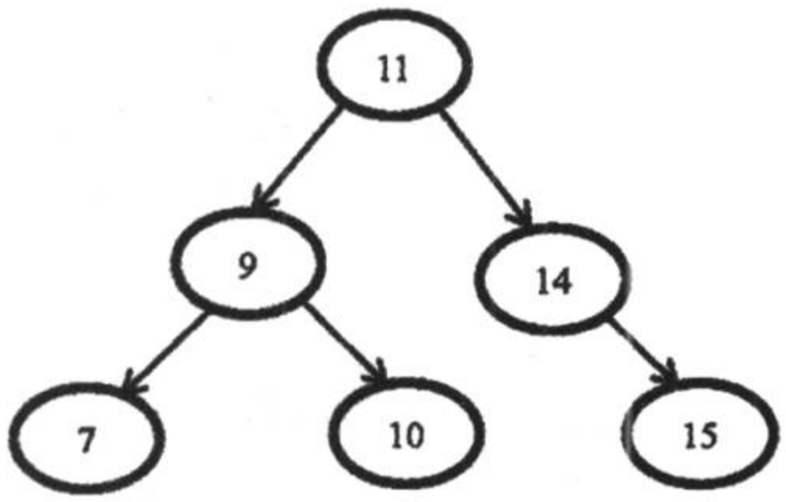


图 6-1　二叉树

6.2　平衡二叉树结构

平衡二叉树是在二叉树基础上的提高，二叉树随着节点的深度加大时，查询的均分复杂度就会上升，为了提供更快的查询速度，平衡树出现了。它是一棵空树，必须满足左右两个子树的高度差的绝对值不超过 1，且它的左子树和右子树都是一棵平衡二叉树。它和二叉树最大的区别在于随时要保证插入后的整棵二叉树是平衡的。它会通过左旋或者右旋来使不平衡的树变成平衡树，如图 6-2、图 6-3 所示。

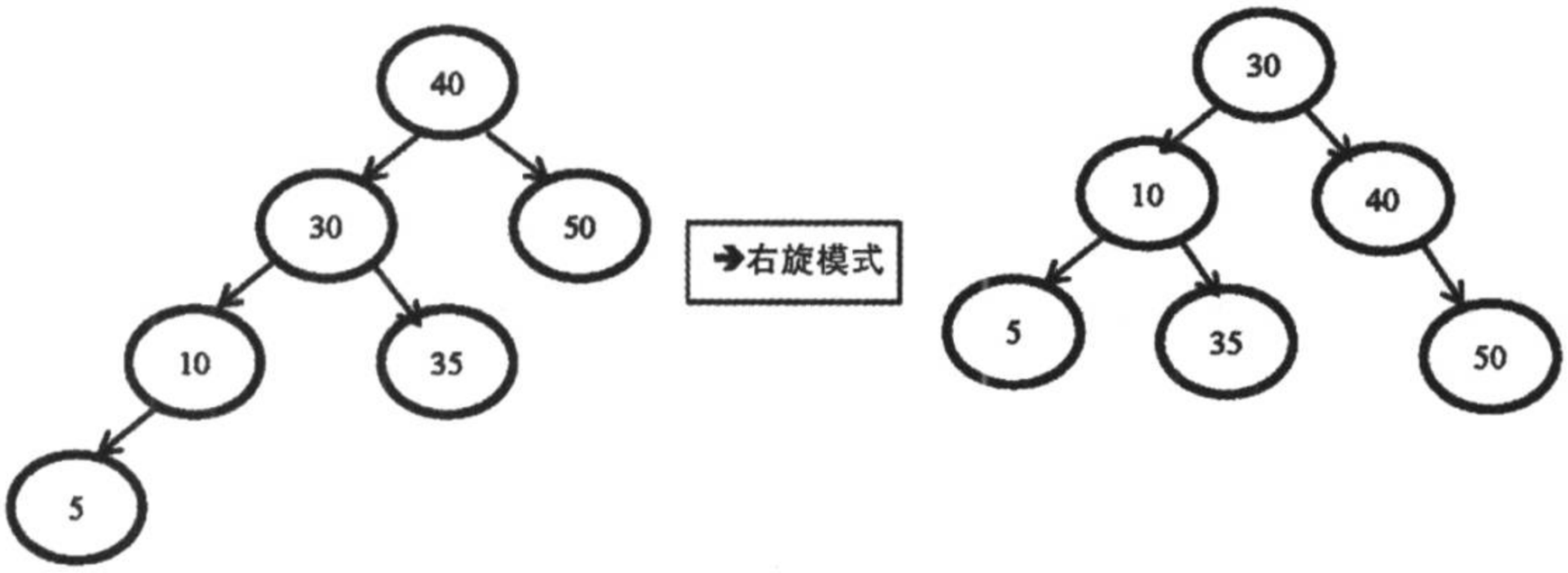


图 6-2　右旋模式

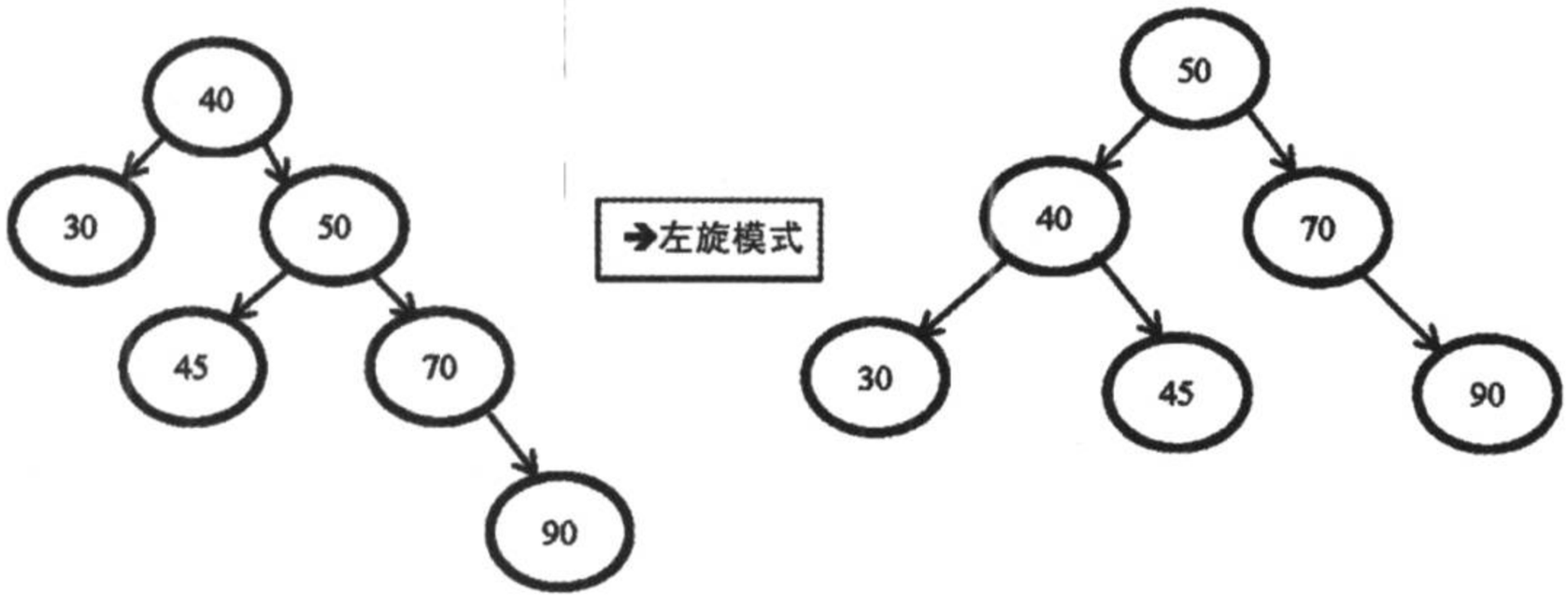


图 6-3　左旋模式

6.3 B-tree 结构

B-tree 结构又称 Btree。有些书中说 MySQL 的索引结构是 Btree，其实并不正确，MySQL 的索引结构是 B+tree，Btree 和 B+tree 有很多不同。我们先来看一下 Btree 的结构，如图 6-4 所示。

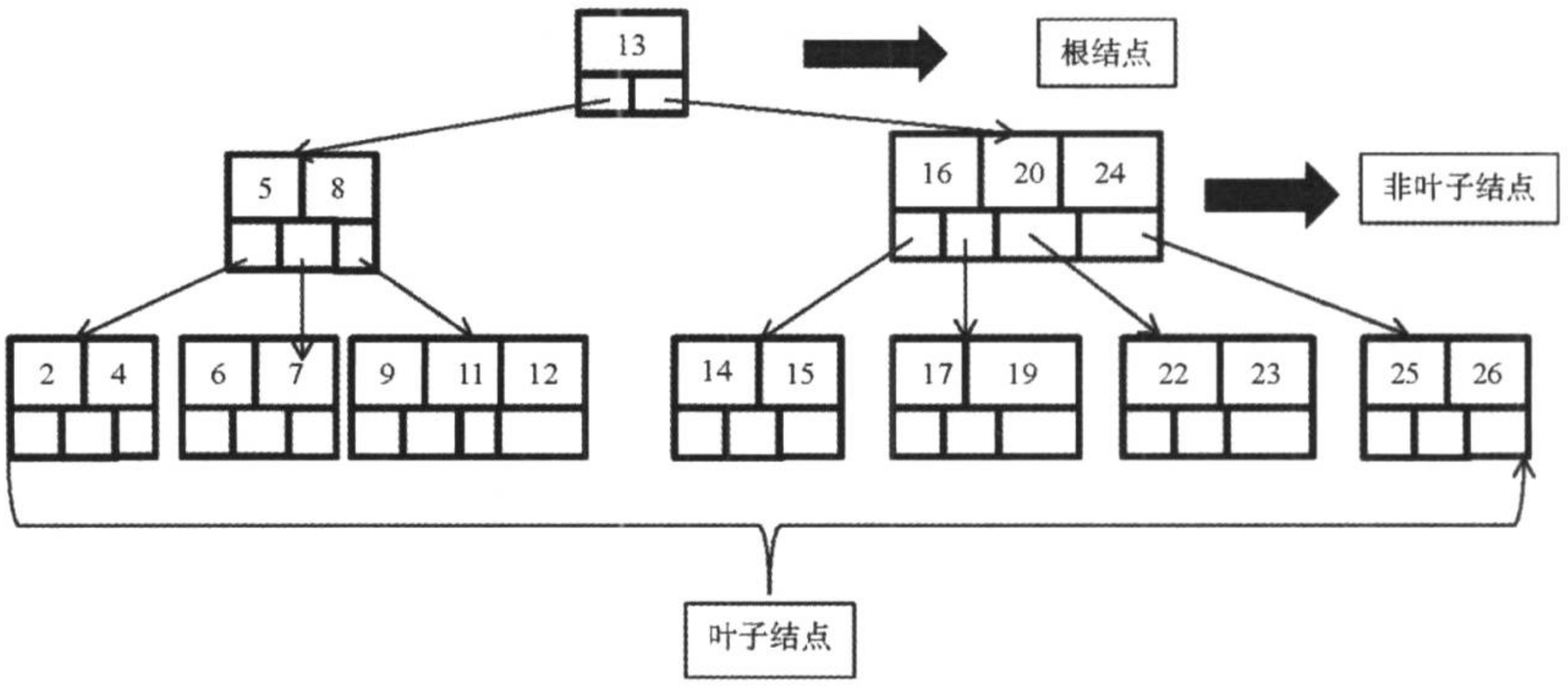


图 6-4 B-tree 的结构

从图 6-4 中可以看出 B 树的结构是一个结点可以拥有多于两个子结点的多叉查找树。

图 6-4 最多子结点数为 4 个，可以理解为这是一棵四阶的 B 树结构。树中每个结点最多含有 4 个子结点。除了根结点和叶子结点，其他每个结点至少有 2 个子结点。若根结点不是叶子结点，则至少有两个子结点。最重要的一点就是所有叶子结点都出现在同一层，叶子结点不包含任何关键字的信息。

6.4 B+tree

B+tree 是 Btree 的变体，也是一种多路搜索树，定义基本与 B-tree 相同。但它所有关键字的信息都出现在叶子结点中，并且包含这些关键字记录的指针，叶子结点可以按照关键字的大小顺序链接。还有就是它所有的数据都保存在叶子结点中，这是区别于 B-tree 结构最主要的特点。MySQL 数据库使用 B+tree 索引结构，如图 6-5 所示。

> 总结：B+tree 索引是双向链表结构，而且用 B+tree 结构做检索要比 B-tree 快，可以看出访问关键字的顺序是连续性的，不用再访问上一个结点，而且叶子结点包含所有数据信息。

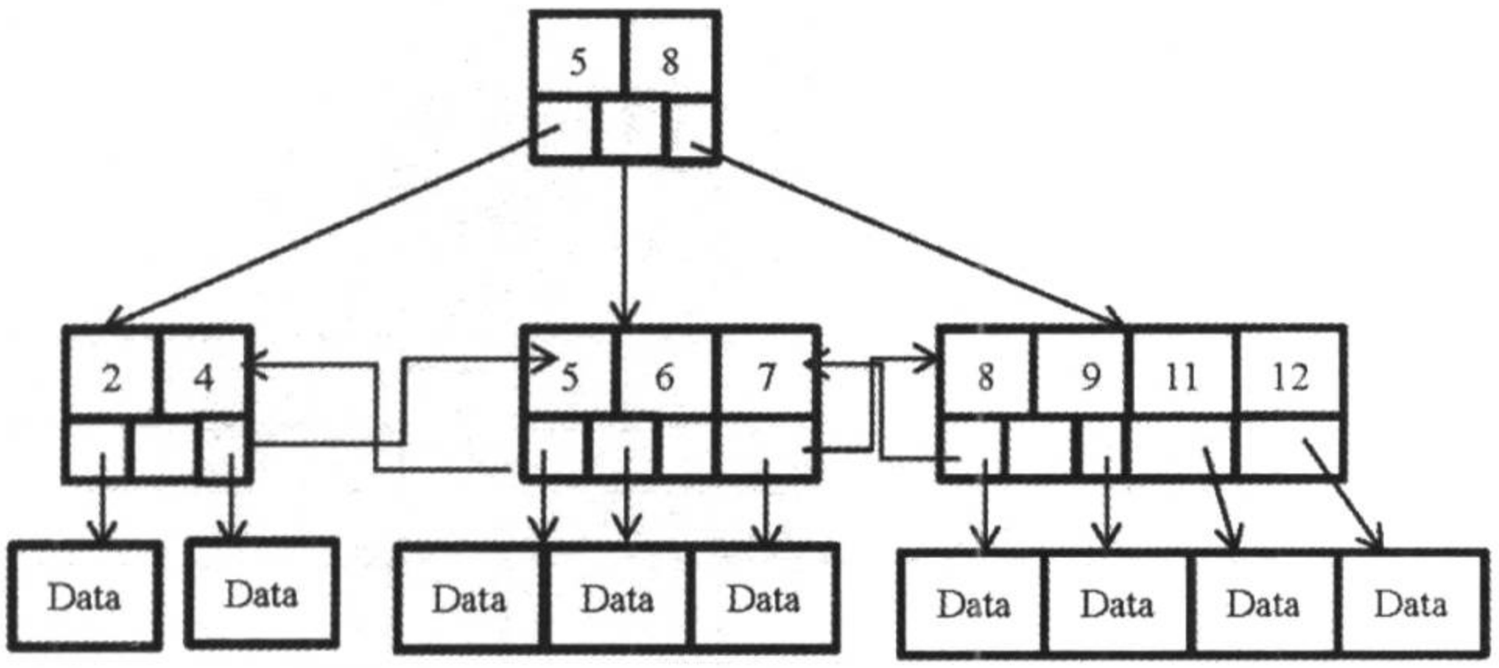


图 6-5　B+tree 索引结构

6.4.1　聚集索引和普通索引

MySQL 数据库的 B+tree 索引其实可以分为两大类，一类叫聚集索引，一类叫非聚集索引（普通索引）。我们知道 InnoDB 存储引擎表是索引组织表，聚集索引其实就是一种索引组织形式，索引键值的逻辑顺序决定了表数据行的物理存储顺序。聚集索引叶子结点存放表中所有行数据记录的信息，所以经常会说数据即索引，索引即数据，这是针对聚集索引来说的。我们在创建一张表时，要显式地为表创建一个主键（聚集索引），如果不主动创建主键，那么 InnoDB 会选择第一个不包含有 null 值的唯一索引作为主键。如果连唯一索引都没有，InnoDB 就会为该表默认生成一个 6 字节的 rowid 作为主键。

普通索引在叶子结点并不包含所有行的数据记录，只是会在叶子结点存有自己本身的键值和主键的值。在检索数据时，通过普通索引叶子结点上的主键来获取到想要查找的行数据记录，如图 6-6 所示。

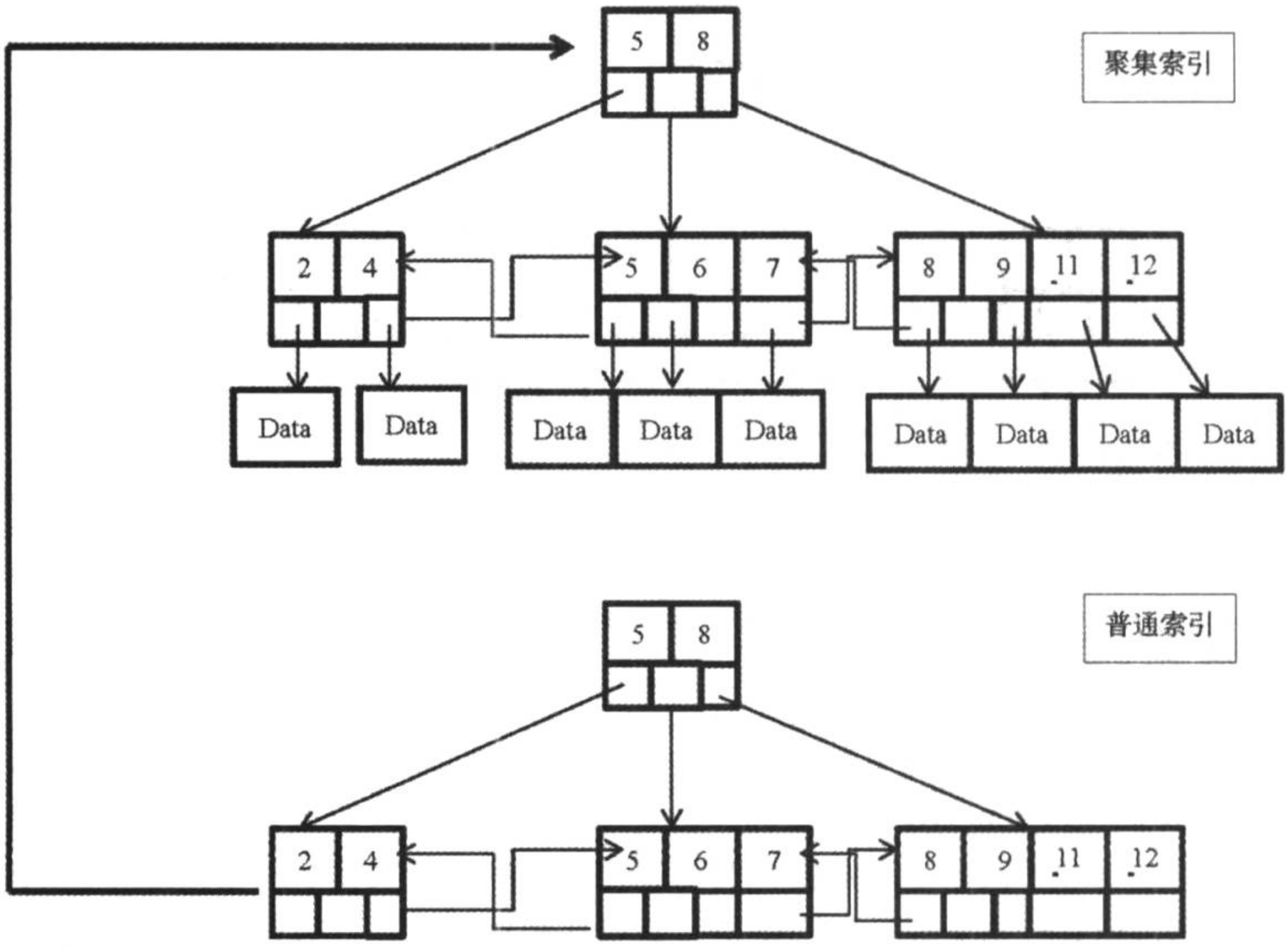


图 6-6　普通索引

普通索引的创建语法：

```
alter table table_name add index index_name(索引字段);
```

或者

```
create index index_name on table_name(索引字段);
```

通常可以使用 show index from table_name 来查看表中有哪些索引。还可以使用 explain 命令来查看 SQL 语句的执行计划，判断添加索引之后，优化器是否生成了更高效的执行计划。

通过实验来深入地理解索引。

首先创建一张表名为 t 的表，表中有 id、name、address 三个字段。id 字段是主键，并通过存储过程往表中灌入 6 万条数据：

```
mysql> desc t;
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| id      | int(11)     | NO   | PRI | NULL    | auto_increment |
| name    | varchar(20) | NO   |     | NULL    |                |
| address | varchar(20) | NO   |     | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql> select count(*) from t;
+----------+
| count(*) |
+----------+
|    60000 |
+----------+
1 row in set (0.02 sec)
```

查询 name=name11 的记录，先看这条 SQL 语句的查询计划：

```
mysql> explain select * from t where name='name11';
+----+-------------+-------+------+---------------+------+---------+------+-------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows  | Extra |
+----+-------------+-------+------+---------------+------+---------+------+-------+-------+
|  1 | SIMPLE      | t     | ALL  | NULL          | NULL | NULL    | NULL | 60180 | Using |
+----+-------------+-------+------+---------------+------+---------+------+-------+-------+
1 row in set (0.00 sec)
```

针对 explain 命令生成的执行计划，这里有一个查看“心法”。我们可以先从查询类型 type 列开始查看，如果出现 all 关键字，后面的内容就都可以不用看了，代表全表扫描。再看 key 列，看是否使用了索引，null 代表没有使用索引。然后看 rows 列，该列用来表示在 SQL 执行过程中被扫描的行数，该数值越大，意味着需要扫描的行数越多，相应的耗时就更长。最后再看 extra 列，在这列中要观察是否有 Using filesort 或者 Using temporary 这样的关键字出现，这些是很影响数据库性能的。MySQL 5.7 的执行计划中会默认添加 filtered 列（MySQL 5.6 使用 explain extended 也会增加此列），它指返回结果的行占需要读到的行（rows 列的值）的百分比。需要注意的是，explain 中输出的 rows 只是一个估算值。本例中该表进行了全表扫描。

该如何优化这条 SQL 语句呢？

这样的问题看似简单，但其实就是一个坑。我也经常问我的学生 SQL 语句如何优化？大部分的答案都是添加索引。其实针对 SQL 的问题，我们要回归到表这一层。下面总结一下面对 SQL 语句如何优化的问题，理清一个正确的解决思路。

（1）先看表的数据类型是否设计得合理，有没有遵守选取数据类型越简单越小的原则。

（2）表中的碎片是否整理。

（3）表的统计信息是否收集，只有统计信息准确，执行计划才可以帮助我们优化 SQL。

（4）查看执行计划，检查索引的使用情况，没有用到索引，考虑创建。

（5）在创建索引之前，还要查看索引的选择性，来判断这个字段是否合适创建索引。

所谓索引的选择性是指不重复的索引值（也称为基数，cardinality）和数据表的记录总数的比值。索引的选择性越高则查询效率越高（越接近于 1 越高），因为选择性高的索引可以让MySQL 在查找时过滤掉更多的行。主键索引和唯一索引的选择性是 1，这是最好的索引选择性，性能也是最好的。

（6）创建索引之后，再查看一下执行计划，对比这两次的结果，看是否查询效率提高了。

做好这 6 步操作，才是一个正确的优化 SQL 的过程。但针对上面这个例子，由于是新创建的表，我们只需要考虑合理地添加索引就可以了。于是又引出了我总结的一个合理创建索引的三个“经常”。

（1）经常被查询的列（一般放在 where 条件后面）。

（2）经常用于表连接的列。

（3）经常排序分组的列（order by 或者 group by 后面的字段）。

可以考虑在这三种情况下建立索引。

给 name 字段创建索引，先查看 name 字段的索引选择性。

```
mysql>  select count(distinct name)/count(*) from t;
+-----------------------------------+
| count(distinct name)/count(*)     |
+-----------------------------------+
|                            0.8333 |
+-----------------------------------+
1 row in set (0.15 sec)
```

可以看到 name 字段选择性为 0.8，代表重复值很少，选择性很高，非常适合建索引。

给 name 字段创建索引，然后再查看执行计划：

```
mysql> create index idx_name on t(name);
Query OK, 0 rows affected (0.58 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain select * from t where name='name11';
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------
-----------------------+
| id | select_type | table | type | possible_keys | key      | key_len | ref   | rows | E
xtra                  |
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------
-----------------------+
|  1 | SIMPLE      | t     | ref  | idx_name      | idx_name | 62      | const |    2 | U
sing index condition |
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------
-----------------------+
1 row in set (0.00 sec)
```

可见建完 name 索引之后，执行计划展现的查询效率提升了很多，证明我们所建的这个索引是有效果的。我们还观察到在执行计划 extra 列出现了一个叫 using index condition 的关键字。这又代表什么意思呢？对性能的影响是好是坏呢？

6.4.2　ICP、MRR 和 BKA

Index Condition Pushdown（ICP）是 MySQL 使用索引从表中检索行数据的一种优化方式，从 MySQL 5.6 开始支持，5.6 之前，存储引擎会通过遍历索引定位基表中的行，然后返回给 Server 层，再去为这些数据行进行 WHERE 后的条件的过滤。MySQL 5.6 之后支持 ICP，如果 WHERE 条件可以使用索引，MySQL 会把这部分过滤操作放到存储引擎层，存储引擎通过索引过滤，把满足的行从表中读取出。ICP 能减少引擎层访问基表的次数和 Server 层访问存储引擎的次数。

MySQL 通过 optimizer_switch 参数中的 index_condition_pushdown 选项来控制，默认是开启的。

```
show variables like '%optimizer_switch%';
```

```
--+
| optimizer_switch | index_merge=on,index_merge_union=on,index_merge_sort_union=on,
merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on mrr=
_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semij
,loosescan=on,firstmatch=on,subquery_materialization_cost_based=on,use_index_extens
n |
+-------------------+-------------------------------------------------------------
```

可通过数据库命令行执行启动或者关闭操作：

```
set optimizer_switch="index_condition_pushdown=on|off";
```

当使用 ICP 优化时，执行计划的 extra 列会显示 Using index condition 的关键字提示。

```
root@db 00:01:  [zs]> explain select * from tt where score>70 and score<90;
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+-
----+----------+-----------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key       | key_len | ref  |
rows | filtered | Extra                             |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+-
----+----------+-----------------------------------+
|  1 | SIMPLE      | tt    | NULL       | range | idx_score     | idx_score | 1       | NULL |
   1 |   100.00 | Using index condition; Using MRR |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+-
----+----------+-----------------------------------+
1 row in set, 1 warning (0.00 sec)
```

MRR 的全称是 Multi-Range Read Optimization。MySQL 的 MRR 特性也是 5.6 之后才有的。通过 optimizer_switch 参数中两个重要的选项来控制，一个是 mrr，另一个是 mrr_cost_based。

参数值默认也是开启状态的（mrr=on 和 mrr_cost_based=on）。

mrr_cost_based 选项表示是否通过基于成本的算法来确定开启 mrr 特性；设置为 on 表示自行判断，要是为 off 则表示强制开启 mrr。

```
| optimizer_switch                    | index_merge=on,index_merge_union=on,index_merge_sort_
union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdow
n=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_c
ost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on |
| optimizer_trace                     | enabled=off,one_line=off
```

可以通过数据库命令行执行启动或者关闭操作：

```
SET  global optimizer_switch='mrr=on|off,mrr_cost_based=on|off';
```

> 注：当 mrr=on、mrr_cost_based=on 时，表示 cost base 的方式还选择启用 MRR 优化,当发现优化后的代价过高时就会不使用该项优化。

当 mrr=on、mrr_cost_based=off 时，表示总是开启 MRR 优化。

当使用 MRR 优化时，执行计划的 extra 列会显示 Using 的关键字提示。

```
root@db 00:01:  [zs]> explain select * from tt where score>70 and score<90;
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+
----------+-----------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key       | key_len | ref  |
rows | filtered | Extra                                   |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+
----------+-----------------------------------------+
|  1 | SIMPLE      | tt    | NULL       | range | idx_score     | idx_score | 1       | NULL |
   1 |   100.00 | Using index condition; Using MRR |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+
----------+-----------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

MRR 的原理很简单，我们之前讲过 MySQL 普通索引获取数据的方式，先是通过索引页的叶子结点找到对应的主键，再通过主键找到相对应的行数据记录。如果在一张表中对某一个字段创建一个普通索引，但这个字段有一些重复的值，那么根据这个字段去做 where 条件时，每次取到的主键值可能不是按顺序的，那么随机 I/O 的行为就会发生。MRR 原理如图 6-7 所示（图片来自 MariaDB 官方文档）。

MRR 的作用就是把普通索引的叶子结点上找到的主键值的集合存储到 read_rnd_buffer 中，然后在该 buffer 中对主键值进行排序，最后再利用已经排序好的主键值的集合，去访问表中的数据，这样就由原来的随机 I/O 变成了顺序 I/O，降低了查询过程中的 I/O 开销。

> 注：在生产环境中，read_rnd_buffer_size 的值可以在 4～8MB 之间调整。

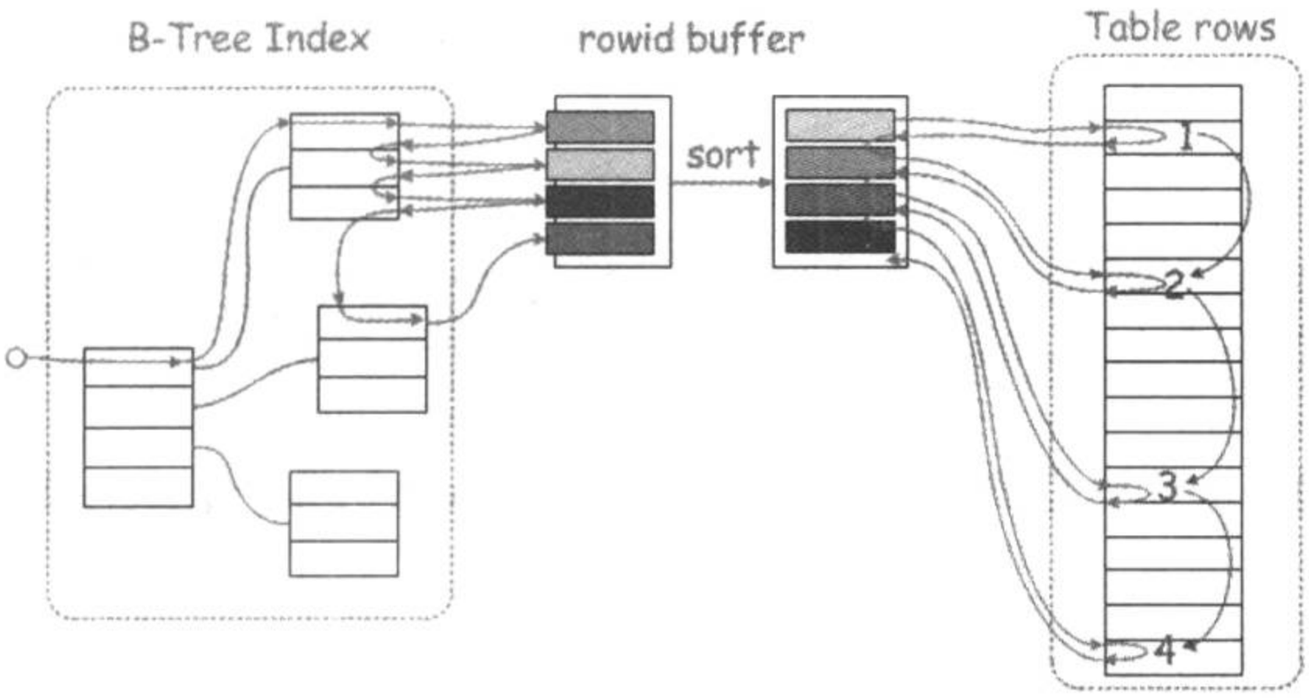


图 6-7　MRR 原理图

BKA 的全称为 Batched Key Access。它是提高表 join 性能的算法，其作用是在读取被 join 表的记录的时候使用顺序 I/O。

BKA 的原理很简单，对于多表 join 语句，当 MySQL 使用索引访问第二个 join 表时，使用一个 join buffer 来收集第一个操作对象生成的相关列值。BKA 构建好 key 后，批量传给引擎层做索引查找。key 是通过 MRR 接口提交给引擎的，这样一来 MRR 使得查询更加高效。

BKA 的原理如图 6-8 所示（图片来自 MariaDB 官方文档）。

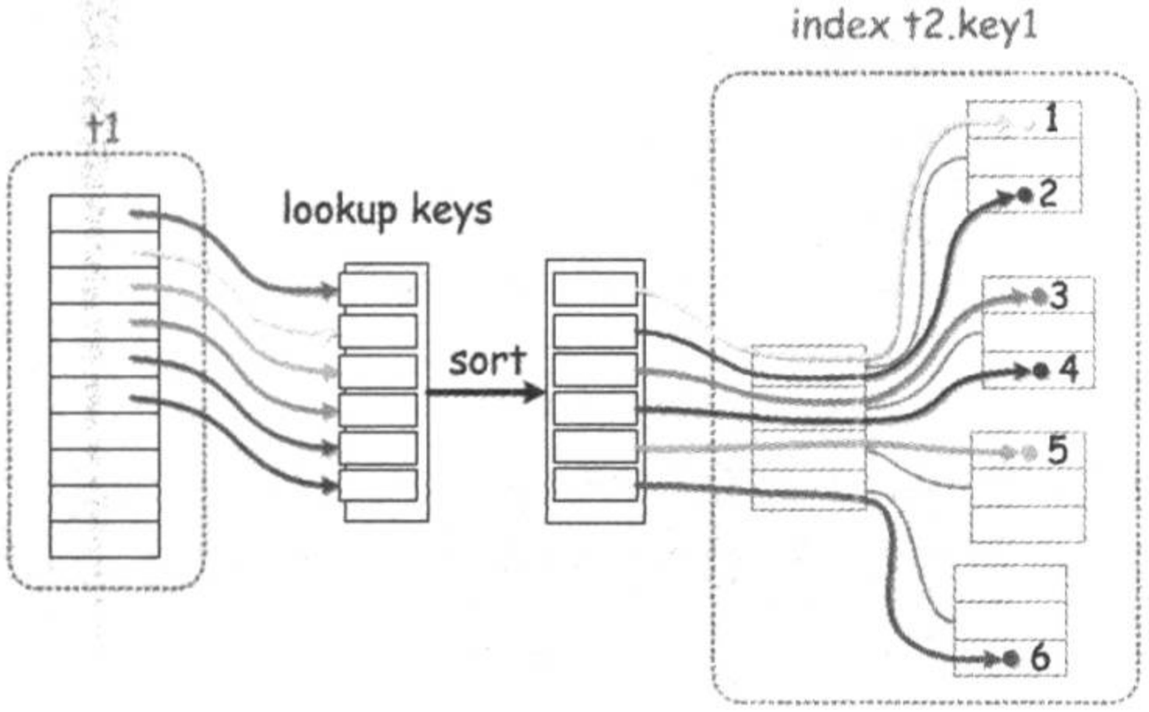


图 6-8　BKA 原理图

MySQL 中通过 optimizer_switch 参数中的 batched_key_access 选项来控制，该选项默认是关闭的。

```
                                                                    |
| optimizer_switch                    | 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, loose
scan=on, firstmatch=on, duplicateweedout=on, subquery_materialization_cost_based=on, use_index_exte
nsions=on, condition_fanout_filter=on, derived_merge=on |
| optimizer_trace                     | enabled=off, one_line=off
```

要想开启该参数，必须先要保证是在强制使用 MRR 的基础上才可以。执行如下操作才算开启 BKA 功能：

```
SET  global optimizer_switch='mrr=on,mrr_cost_based=off';
SET  global optimizer_switch='batched_key_access=on';
```

当 BKA 被使用时，执行计划的 extra 列会显示 Using join buffer (Batched Key Access)的关键字提示。

MRR 与 BKA 之间的关系如图 6-9 所示（图片来自 MariaDB 官方手册）

https://mariadb.com/kb/en/library/multi-range-read-optimization/。

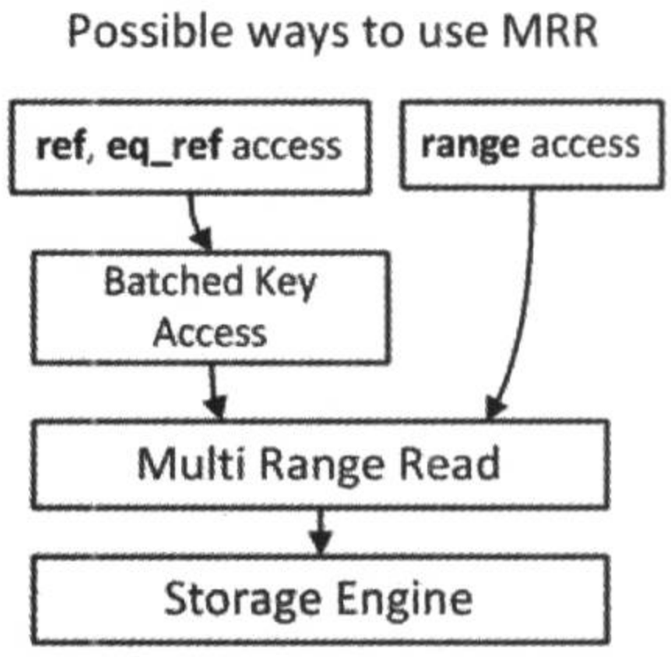


图 6-9　MRR 与 BKA 的关系

存储引擎上端是 MRR，范围扫描（range access）中 MySQL 将扫描到的数据存入 read_rnd_buffer_size，对其按照主键 rowid 进行排序，然后使用排序好的数据进行顺序回表，因为 InnoDB 中的叶节点数据是按照主键 ROWID 进行排列的，这样就转换随机读取为顺序读取了。

而在 BKA 中，则在被连接表使用 ref、eq_ref 索引扫描方式时，第一个表中扫描到的键值放到 join_buffer_size 中，然后调用 MRR 接口进行排序和顺序访问并且通过 join 条件得到数据，这样连接条件也成为了顺序比对。

6.4.3　主键索引和唯一索引

主键索引其实就是聚集索引，每张表中有且仅有一个主键，可以由表中一个或多个字段组成。主键索引必须满足三个条件，主键值必须唯一；不能包含 null 值；一定要保证该值是自增属性。使用自增列做主键，可以保证写入数据的顺序也是自增的，这就在很大程度上提高了存取效率。

创建主键的语法：

```
alter table table_name add primary key(column);
```

唯一索引是约束条件的一种，其实就是不允许有重复的值，但是可以允许有 null 值。上面说过表中只能有一个主键，但唯一索引可以有多个。

创建唯一索引的语法：

```
alter table table_name add unique (column);
```

6.4.4 覆盖索引

MySQL 只需要通过索引就可以返回查询所需要的数据，而不必在查到索引之后再去回表查询数据了，这样就减少了大量的 I/O 操作，查询速度也相当快。在执行计划的 extra 列中会出现 Using index 的关键字。

```
mysql> explain select id from t where name='name11'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ref
possible_keys: idx_name
          key: idx_name
      key_len: 62
          ref: const
         rows: 2
        Extra: Using where; Using index
1 row in set (0.00 sec)

ERROR:
No query specified
```

这条查询语句中，想要检索主键 id 字段，而且在查询条件中 name 字段是普通索引，之前讲过普通索引中包含主键的值，相当于（name,id）索引，那么这条语句就使用了覆盖索引，出现了 Using index。

> 注意：如果使用覆盖索引，一定要让 select 列出所需要的列。坚决不可以直接写出 select *。

6.4.5 前缀索引

对于 BLOB、TEXT，或者很长的 VARCHAR 类型的列，为它们的前几个字符（具体几个字符是在建立索引时指定的）建立索引，这样的索引就叫前缀索引。这样建立起来的索引更小，所以查询更快。但前缀索引也有它的坏处，它不能在 ORDER BY 或 GROUP BY 中使用前缀索引，也不能把它们用作覆盖索引。

创建前缀索引的语法：

```
alter table table_name add key(column_name(prefix_length));
```

> 注意：这里最关键的参数就是 prefix_length，这个值需要根据实际表的内容，来得到合适的索引选择性。

6.4.6 联合索引

联合索引又叫复合索引，是在表中两个或两个以上的列上创建的索引。利用索引中的附加列，可以缩小检索的段池范围，更快地搜索到数据。创建语法跟普通索引的创建一样。例如，为表 t 的 c1、c2 字段。

创建一个联合索引语法：

```
create index idx_c1_c2  on t (c1,c2);
```

联合索引的使用过程中，必须要满足最左前缀原则。一般把选择性高的列放在前面。一条查询语句可以只使用索引中的一部分，但必须从最左侧开始。

可以用到 c1 索引和 c1,c2 索引。

以下查询可以使用到索引：

```
select * from t where c1=某值;
select * from t where c2=某值 and c1=某值;
select * from t where c1=某值 and c2 in(某值, 某值)
select * from t order by c1,c2;
select * from t where c1=某值 order by c2;
```

反之，使用不到索引的情况：

```
select * from t where c2=某值;
select * from t where c2=某值 order by c1;
```

还有一种特殊的情况：

```
select * from t where c1=某值 or c2=某值;
```

虽然 c1 字段在前，但是这种情况是不能使用到索引的。

这种情况可以在 c1、c2 字段上面建两个单列索引。

> 注：尽量在生产环境中，让程序端多做一些判断，不要让数据库做各种运算。尽量避免在 SQL 语句中出现 or 关键字。多列中可以考虑使用 union。

6.5 哈希索引

哈希索引采用哈希算法，把键值换算成新的哈希值，这里需要注意哈希索引只能进行等值查询，不能进行排序、模糊查找、范围查询等。检索时不需要像 B+tree 那样从根结点到叶子结点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，查询速度非常快。例如，select * from zs where city_id=100 这样一条语句。哈希过程如图 6-10 所示。

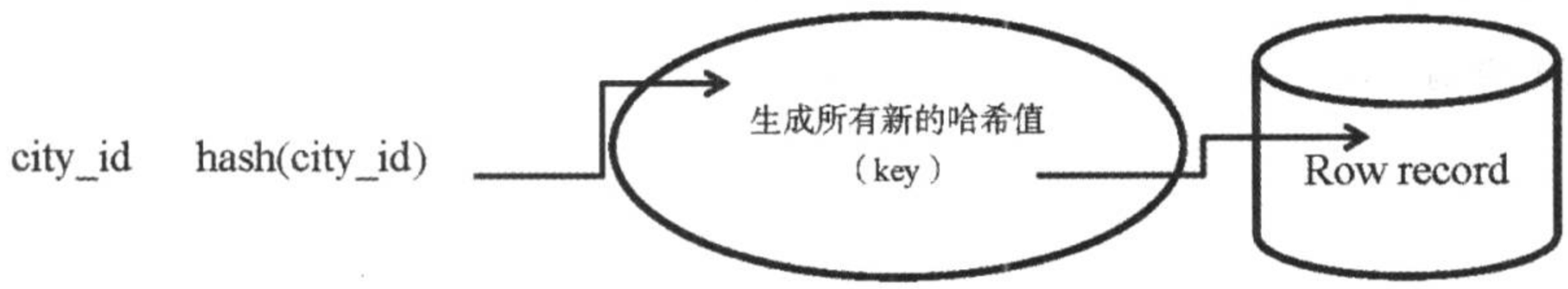


图 6-10 哈希过程

6.6 索引的总结

本章我们学习了 MySQL 数据库的 B+tree 和哈希索引，也深入了解了 B+tree 索引中不同的索引种类。下面总结一下索引的优点，看看哪种情况下，不能使用到索引。

索引的优点：

（1）提高数据检索效率。

（2）提高聚合函数效率。

（3）提高排序效率。

（4）使用覆盖索引可以避免回表。

索引创建的四个不要：

（1）选择性低的字段不要创建索引（例如，性别 sex、状态 status）。

（2）很少查询的列不要创建索引（项目初期就要确定好）。

（3）大数据类型字段不要创建索引。

（4）尽量避免不要使用 NULL，应该指定列为 NOT NULL（在 MySQL 中，含有空值的列很难进行查询优化，它们会使得索引、索引的统计信息及比较运算更加复杂。可以使用空字符串代替空值）。

使用不到索引的情况：

（1）通过索引扫描的行记录数超过全表的 30%，优化器就不会走索引，而变成全表扫描。

（2）联合索引中，第一个查询条件不是最左索引列。

（3）联合索引中，第一个索引列使用范围查询，只能使用到部分索引，有 ICP 出现（范围查询是指 <、=、<=、BETWEEN and）。

（4）联合索引中，第一个查询条件不是最左前缀列。

（5）模糊查询条件列最左以通配符 % 开始（可以考虑放到子查询里面）。

（6）两个单列索引，一个用于检索，一个用于排序。这种情况下只能使用到一个索引。因为查询语句中最多只能使用一个索引，考虑建立联合索引。

（7）查询字段上面有索引，但是使用了函数运算。

第7章 事务

事务其实就是一组 DML（insert、delete、update）语句的集合。MySQL 数据库 InnoDB 存储引擎支持事务，MyISAM 不支持。而且 MySQL 的事务默认是自提交模式，如果想要开启事务，必须以 begin 命令开始，以 commit 或者 rollback 命令结束。

7.1 事务的特性

1. 原子性（Atomicity）

事务的原子性是指事务中包含的所有操作要么都做，要么都不做，保证数据库是一致的。

甲账户向乙账户转账 1000 元，则先将甲账户减少 1000 元，再将乙账户增加 1000，这两个动作要么都提交，要么都回退，不可能发生一个有效另一个无效的情况。

2. 一致性（Consistency）

一致性是指数据库中的数据在事务操作前和事务处理后必须都满足业务规则约束。

甲乙账户的总金额在转账前和转账后必须一致，如有不一致，则必须是短暂的，且只有在事务提交前才会出现的。

3.隔离性（Isolation）

隔离性是数据库允许多个并发事务同时对数据进行读写和修改的能力，隔离性可以防止多

个事务并发执行时由于交叉执行而导致数据的不一致。

在甲、乙之间转账时，丙同时向甲转账，若同时进行，则甲、乙之间的一致性不能得到满足。所以在甲、乙事务执行过程中，其他事务不能访问（修改）当前相关的数值。

4. 持久性（Durability）

事务处理结束后，对数据的修改就是永久的，即便系统发生故障也不会丢失。

在事务的四大特性中，我们主要研究隔离性，知识点就是事务的隔离级别。

7.2 事务语句

事务开启语句是由 begin 或者 start transaction（read write|read only）命令来开始的，或者把自提交特性关掉（执行 set autocommit=0 命令）。事务结束语句通常使用 commit 或者 rollback 显示结束。commit 代表提交事务，使得已对数据库做的所有修改成为永久性。rollback 代表回滚事务，撤销正在进行的所有未提交的修改。

```
mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
+------+------+
3 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (id,name) values (4,'dd');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
+------+------+
4 rows in set (0.00 sec)
```

```
mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
+------+------+
4 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (id,name) values (5,'ee');
Query OK, 1 row affected (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
+------+------+
4 rows in set (0.00 sec)
```

当然事务除了显示提交（commit）和回滚（rollback）的操作，还有隐式提交和回滚。隐式提交可以是 DDL 语句的操作或者再次输入 begin 和 start transaction 命令。隐式回滚可以退出会话、连接超时或者关机等。隐式提交语句展示：

```
mysql>
mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
+------+------+
4 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (id,name) values (5,'ee');
Query OK, 1 row affected (0.00 sec)

mysql> create table tt (id int);
Query OK, 0 rows affected (0.01 sec)

mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
|    5 | ee   |
+------+------+
5 rows in set (0.00 sec)
```

> 注：在 DDL 语句中默认自带一个 commit。

再来看一下隐式回滚的展示：

```
mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
|    5 | ee   |
+------+------+
5 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (id,name) values (6,'ff');
Query OK, 1 row affected (0.00 sec)

mysql> exit
Bye
```

```
[root@node3 ~]# mysql -uroot -proot123 test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.6.16-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> select * from t;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
|    2 | bb   |
|    3 | cc   |
|    4 | dd   |
|    5 | ee   |
+------+------+
5 rows in set (0.00 sec)
```

可见在开始一个事务，执行完一条插入语句后，执行了 exit 退出会话命令。等再次进入数据库之后，发现新插入的语句被回滚了。这就是隐式回滚的方式之一。

接下来讨论一个问题，在 Oracle 数据库中，事务不是自动提交的，而 MySQL 数据库是自提交模式（autocommit=1）。那到底需不需要把 MySQL 的自动提交手动关闭呢？这个问题在我平时讲课过程中，经常被我的学生问到。我们先来想一想关闭自动提交有什么好处，set autocommit=0 之后可以不用一个事务一次提交了，可以多个事务一起提交，提高了每秒处理事务的能力。但如果在这个过程中有某个事务一直没有提交，就会导致行锁等待的现象。其他事务必须得等这个事务提交之后，才可以继续提交，这样就严重影响数据库的 TPS 值了。

这里不建议关闭 MySQL 的自提交模式，采用默认开启即可。

7.3 truncate 和 delete 的区别

truncate 是 DDL 语句操作，delete 是 DML 语句操作，它们的共同点都是清空表内的数据，但 truncate 在事务中不能被回滚，而且 truncate 会清空表的自增属性。

第一点，truncate 不能回滚，delete 可以回滚：

```
mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 300 | a    | bj   |
| 301 | b    | sh   |
| 302 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from t;
Query OK, 3 rows affected (0.00 sec)

mysql> select * from t;
Empty set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 300 | a    | bj   |
| 301 | b    | sh   |
| 302 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)
```

```
mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 300 | a    | bj   |
| 301 | b    | sh   |
| 302 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> truncate table t;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from t;
Empty set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
Empty set (0.00 sec)
```

第二点，truncate 清空表的自增 id 属性，从 1 重新开始记录，而 delete 则不会。

用 delete 删除，结果如下：

```
mysql> desc t;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | YES  | MUL | NULL    |                |
| city  | varchar(10) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

```
mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 300 | a    | bj   |
| 301 | b    | sh   |
| 302 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)

mysql> delete from t;
Query OK, 3 rows affected (0.01 sec)

mysql> select * from t;
Empty set (0.00 sec)
```

```
mysql> insert into t (name,city) values ('a','bj'),('b','sh'),('c','gz');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 303 | a    | bj   |
| 304 | b    | sh   |
| 305 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)
```

用 truncate 删除，结果如下：

```
mysql> select * from t;
+-----+------+------+
| id  | name | city |
+-----+------+------+
| 303 | a    | bj   |
| 304 | b    | sh   |
| 305 | c    | gz   |
+-----+------+------+
3 rows in set (0.00 sec)

mysql> truncate table t;
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t (name,city) values ('a','bj'),('b','sh'),('c','gz');
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
+----+------+------+
3 rows in set (0.00 sec)
```

7.4 事务的隔离级别

MySQL InnoDB 存储引擎实现 SQL 标准的 4 种隔离级别，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。MySQL 数据库通过 show variables like '%tx_isolation%'命令来查看当前数据库的隔离级别。可以通过 set global|session transaction isolation level 命令修改全局或者当前会话的事务隔离级别。

目前 MySQL 版本默认的隔离级别是 REPEATABLE-READ。

```
mysql> show variables like '%tx_isolation%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

（1）读未提交（read uncommitted），简称 RU——在其中一个事务中，可以读取到其他事务未提交的数据变化。这种读取其他会话还没提交的事务，叫作脏读现象。在生产环境中不建议使用。

（2）读已提交（read committed），简称 RC——在其中一个事务中，可以读取到其他事务已经提交的数据变化。这种读取也可以叫作不可重复读，允许幻读现象的发生，是 Oracle 数据库默认的事务隔离级别。

（3）可重复读（repetable read），简称 RR，它是 MySQL 默认的事务隔离级别——在其中一个事务中，直到事务结束前，都可以反复读取到事务刚开始时看到的数据，并一直不会发生变化，避免了脏读、不可重复读和幻读现象的发生。

（4）串行（serializable）——在每个读的数据行上都需要加表级共享锁，在每次写数据时都要加表级排他锁。这就会造成 InnoDB 的并发能力下降，大量的超时和锁竞争就会发生。不建议使用到生产环境中。

7.5 细说脏读、不可重复读、幻读、可重复读现象

7.5.1 脏读

脏读是在事务隔离级别读未提交（RU）中出现的现象。一个事务读取到了其他事务还没有提交的数据。

会话 A，在 RU 隔离级别下，操作如下：

```
mysql> show variables like '%iso%';
+---------------+------------------+
| Variable_name | Value            |
+---------------+------------------+
| tx_isolation  | READ-UNCOMMITTED |
+---------------+------------------+
1 row in set (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
+----+------+------+
```

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (name,city) values ('d','sz');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
+----+------+------+
4 rows in set (0.00 sec)
```

会话 B，在 RU 隔离级别下，操作如下：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
+----+------+------+
4 rows in set (0.00 sec)
```

会话 B 读取到了会话 A 还未提交的事务数据信息，这就是脏读现象。事务隔离级别的名称是读未提交。

7.5.2 不可重复读与幻读

不可重复读是指在其中一个事务中，读取到了其他事务针对旧数据的修改记录（常见的操作就是 update 或者 delete 语句）。

幻读是指在其中一个事务中，读取到了其他事务新增的数据，仿佛出现了幻影现象（常见的操作就是 insert 语句）。这种读的现象允许出现在读已提交的事务隔离级别中。

会话 A，在隔离级别 RC 模式下，在事务中执行一条 insert 语句操作，并且进行了提交：

```
mysql> show variables like '%iso%';
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tx_isolation  | READ-COMMITTED |
+---------------+----------------+
1 row in set (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
+----+------+------+
3 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (name,city) values ('d','sz');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

会话 B，读取到了会话 A 在事务中已经提交的新增数据：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
+----+------+------+
4 rows in set (0.00 sec)
```

以上演示的操作就是幻读现象，在 Oracle 数据库中并没有解决这种情况，MySQL 数据库通过 RR 这个可重复读事务隔离级别很好地避免了此现象的发生。

7.5.3 可重复读

可重复读是 MySQL 数据库默认的事务隔离级别。它消除了脏读、不可重复读、幻读现象，很好地保证了事务的一致性。

会话 A，在 RR 隔离级别下，在事务中新增了一条数据，并且已经对数据进行提交：

```
mysql> show variables like '%iso%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
|  5 | e    | fj   |
+----+------+------+
5 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t (name,city) values ('f','tj');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

会话 B，读取到的还是事务一开始时的数据，并没有读取到新增的那条。如果想要读取到新增数据的信息，可以在查询语句后面加 for update，这样可以查到最新数据版本号的记录，或者执行 commit 操作。

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
|  5 | e    | fj   |
+----+------+------+
5 rows in set (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
|  5 | e    | fj   |
+----+------+------+
5 rows in set (0.00 sec)
```

```
mysql> select * from t for update;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
|  5 | e    | fj   |
|  6 | f    | tj   |
+----+------+------+
6 rows in set (0.00 sec)

mysql>
mysql> commit;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    26
Current database: test

Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | a    | bj   |
|  2 | b    | sh   |
|  3 | c    | gz   |
|  4 | d    | sz   |
|  5 | e    | fj   |
|  6 | f    | tj   |
+----+------+------+
6 rows in set (0.00 sec)
```

以上就是可重复读隔离级别的展示，它可以用于对事务要求比较高的数据库系统，如电子交易类的网站。

第 8 章 锁

数据库锁机制简单来说，就是数据库为了保证数据的一致性，使各种共享资源在被并发访问时变得有序而设计的一种规则。

MySQL 的锁机制比较简单，最显著的特点是不同的存储引擎支持不同的锁机制。我们所知道的，InnoDB 支持行锁，有时也会升级为表锁，MyISAM 只支持表锁。

- 表锁的特点就是开销小、加锁快；不会出现死锁；锁粒度大，发生锁冲突的概率高，并发度相对低。
- 行锁的特点就是开销大，加锁慢；会出现死锁；锁粒度小，发生锁冲突的概率低，并发度也相对行锁较高。

8.1 InnoDB 的锁类型

InnoDB 的行锁类型主要有读锁（共享锁）、写锁（排他锁）、意向锁和 MDL 锁。

8.1.1 读锁

读锁，简称 S 锁，一个事务获取了一个数据行的读锁，其他事务能获得该行对应的读锁，但不能获得写锁，即一个事务在读取一个数据行时，其他事务也可以读，但不能对该数据行进行增删改的操作。

读锁有两种select方式的应用，第一种是自动提交模式下的select查询语句，不需加任何锁，直接返回查询结果，这就是一致性非锁定读。第二种就是通过select….. lock in share mode在被读取的行记录或行记录的范围上加一个读锁，让其他事务可以读，但是要想申请加写锁，那就会被阻塞。

8.1.2 写锁

写锁简称X锁，一个事务获取了一个数据行的写锁，其他事务就不能再获取该行的其他锁，写锁优先级最高。

写锁的应用就很简单了，一些DML语句的操作都会对行记录加写锁。

比较特殊的就是select for update，它会对读取的行记录上加一个写锁，那么其他任何事务就不能对被锁定的行上加任何锁了，要不然会被阻塞。

8.1.3 MDL锁

MySQL 5.5引入了meta data lock，简称MDL锁，用于保证表中元数据的信息。在会话A中，表开启了查询事务后，会自动获得一个MDL锁，会话B就不可以执行任何DDL语句的操作。

不能执行为表中添加字段的操作，会用MDL锁来保证数据之间的一致性。

会话A：

```
root@db 15:17:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 15:17:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

会话B：

```
root@db 15:22:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 15:22:  [zs]> alter table tt add num tinyint(1) not null default 0;
```

```
root@db 17:13:  [(none)]> show full processlist;
+----+------+-----------+------+---------+------+-------------------------------+-----------------------
---+
| Id | User | Host      | db   | Command | Time | State                         | Info
   |
+----+------+-----------+------+---------+------+-------------------------------+-----------------------
---+
| 11 | root | localhost | zs   | Sleep   |  101 |                               | NULL
   |
| 12 | root | localhost | zs   | Query   |   10 | Waiting for table metadata lock | alter table tt add num tiny
0  |
| 13 | root | localhost | NULL | Query   |    0 | starting                      | show full processlist
   |
```

8.1.4 意向锁

在 MySQL 存储引擎 InnoDB 中，意向锁是表级锁。而且有两种意向锁的类型，分别为意向共享锁和意向排他锁。

- 意向共享锁（IS）是指在给一个数据行加共享锁前必须先取得该表的 IS 锁。
- 意向排他锁（IX）是指在给一个数据行加排他锁前必须先取得该表的 IX 锁。

其实意向锁的作用跟 MDL 类似，都是防止在事务进行过程中，执行 DDL 语句的操作而导致数据的不一致。

8.2 InnoDB 行锁种类

InnoDB 在默认的事务隔离级别为 RR，并且参数 innodb_locks_unsafe_for_binlog = 0 的模式下，行锁的种类有三种。

（1）单个行记录的锁（record lock）。

> 注：主键和唯一索引都是行记录的锁模式。在 RC 隔离级别下，只有 record lock 记录锁模式。

（2）间隙锁（GAP Lock）。

（3）记录锁和间隙锁的组合叫作 next-key lock。

> 注：普通索引默认的就是 next-key lock 模式。

8.2.1 单个行记录的锁

我们知道 InnoDB 支持行锁，那么更新同一行数据时会出现锁等待的现象。

tt 这张表中，id 字段是主键，score 字段是普通索引。

会话 A

在会话 A 事务中更新表 tt 字段 score=60 时的 name 字段记录为 aaa。

```
root@db 10:01:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 10:01:  [zs]> show index from tt;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| tt    |          0 | PRIMARY  |            1 | id          | A         |           3 |     NULL | NULL   |      | BTREE      |         |               |
| tt    |          1 | idx_sc   |            1 | score       | A         |           3 |     NULL | NULL   |      | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
```

```
root@db 10:01:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:01:  [zs]> update tt set name='aaa' where score=60;
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

会话 B

在会话 B 的事务中，更新会话 A 同一行的数据。更新表 tt 字段 score=60 时，name 字段的记录为 a，出现了锁等待超时现象。

```
root@db 10:00:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 10:02:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:02:  [zs]> update tt set name='a' where score=60;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

如果要更新不同的行记录，那么就会成功执行，不会出现锁超时现象。

会话 A

在事务中更新表 tt 字段 score=60 时的 name 记录为 aaa。

```
root@db 12:46:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 12:46:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 12:46:  [zs]> update tt set name='aaa' where score=60;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

会话 B

在事务中更新 score=70 时 name 字段的记录，执行成功，并没有出现锁等待。

```
root@db 12:44:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 12:46:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 12:46:  [zs]> update tt set name='a' where score=70;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

接下来针对表结构进行修改，把 score 上面的索引删除，看看会出现什么样的实验效果。

```
root@db 10:06:  [zs]> alter table tt drop index idx_sc  ;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

会话 A

在事务中更新 tt 表字段 score=60 时，name 的值为 aaa。

```
root@db 13:01:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 13:01:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 13:01:  [zs]> update tt set name='aaa' where score=60;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

会话 B

在事务中更新 tt 表字段 score=70 时，name 的值为 a。

```
root@db 13:02:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 13:02:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 13:02:  [zs]> update tt set name='a' where score=70;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

虽然这次更新的是不同行的记录，但也出现了锁超时现象。这是为什么呢？不同点就是把表中 score 字段上的索引删除了，而实验现象就截然不同了。当 score 字段上面没有索引了，在会话 A 的事务中更新 score=60 时，其实是把所有的行记录上了锁。所以会话 B 上更新不同行记录 score=70 也会报锁超时的错误。

通过这个实验，我们可以总结出一个结论，就是 InnoDB 的行锁其实是加在索引项上面的。

8.2.2 间隙锁（Gap lock）

在 RR 这个事务隔离级别，为了避免幻读现象，引入了 Gap lock。但它只锁定行记录数据的范围，不包含记录本身，即不允许在此范围内插入任何数据。

以下会话均在 RR 隔离级别下。

会话 A

事务中查询 tt 表中 score<80 的记录，在上面加了一个共享锁：

```
root@db 10:07:  [zs]> show variables like '%iso%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)

root@db 10:08:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

```
root@db 10:08:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:09:  [zs]> select * from tt where score<80 lock in share mode;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
+----+------+-------+
2 rows in set (0.00 sec)
```

会话 B

在事务中，往 tt 表中插入 score=75 的数据，但没有插入成功，出现了锁超时。在 score<80 的这个区间内，不允许有任何数据插入。间隙锁的功能完美地展现：

```
root@db 10:09:  [zs]> show variables like '%iso%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.01 sec)

root@db 10:09:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

```
root@db 10:09:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:09:  [zs]> insert into tt (name,score) values ('dd',75);
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

换一种隔离级别再来测试，来看看在 RC 隔离级别下，间隙锁还起不起作用。

以下会话在 RC 隔离级别下。

会话 A

同样的操作，在事务中查询 tt 表中 score<80 的记录，在上面加了一个共享锁（lock in share mode）：

```
root@db 10:22:  [zs]> show variables like '%iso%';
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tx_isolation  | READ-COMMITTED |
+---------------+----------------+
1 row in set (0.00 sec)

root@db 10:22:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

```
root@db 10:22:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:22:  [zs]> select * from tt where score<80 lock in share mode;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
+----+------+-------+
2 rows in set (0.00 sec)
```

会话 B

在事务中，往 tt 表中插入 score=75 的数据，结果成功插入了。

```
root@db 10:24:  [zs]> show variables like '%iso%';
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tx_isolation  | READ-COMMITTED |
+---------------+----------------+
1 row in set (0.00 sec)

root@db 10:24:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

```
root@db 10:24:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:24:  [zs]> insert into tt (name,score) values('dd',75);
Query OK, 1 row affected (0.00 sec)
```

发现并没有满足在 score<80 的范围内不允许插入任何数据的现象。可见间隙锁只是针对 RR 隔离级别才管用，它就是来避免幻读现象发生的。

> 注：RC 隔离级别下是允许出现幻读现象的。

8.2.3 Next-key Locks

Next-key Lock 是记录锁（Record Lock）与间隙锁（Gap lock）的组合，当 InnoDB 扫描索引记录时，会先对选中的索引记录加上记录锁（Record Lock），再对索引记录两边的间隙上加上间隙锁（Gap Lock）。

会话 A

在查询 tt 表 score<85 的语句加上了一个写锁：

```
root@db 10:31:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:31:  [zs]> select * from tt where score<85 for update;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

会话 B

往 tt 表中插入 score=85 的记录，发现无法插入成功，出现了锁超时：

```
root@db 10:30:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:31:  [zs]> insert into tt (name,score) values ('dd',85);



ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

证明不光锁定了 score<85 这个范围区间，还包含 85 这个值的本身。

8.3 锁等待和死锁

锁等待是指一个事务过程中产生的锁，其他事务需要等待上一个事务释放它的锁，才能占用该资源。如果该事务一直不释放，就需要持续等待下去，直到超过了锁等待时间，会报一个等待超时的错误。MySQL 中通过 innodb_lock_wait_timeout 参数控制，单位是秒。

```
root@db 14:08:  [zs]> show variables like '%innodb_lock_wait%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| innodb_lock_wait_timeout   | 10    |
+----------------------------+-------+
1 row in set (0.00 sec)
```

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，就是所谓的锁资源请求产生了回路现象，即死循环，如图 8-1 所示。常见的报错是 Deadlock found when trying to get lock; try restarting transaction。

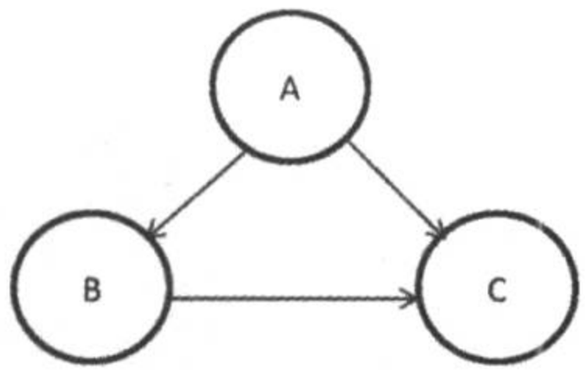


图 8-1 死琐

下面通过一个实验来看一下死锁发生的现象。

表 tt 中，id 是主键，score 字段是索引。

```
root@db 10:34:  [zs]> desc tt;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | NO   |     | NULL    |                |
| score | tinyint(1)  | NO   | MUL | 0       |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

实验过程

在会话 A 的事务中，先执行 update tt set name='aaa' where score=60。

紧接着在会话 B 的事务中，执行 update tt set name='a' where score=70，这时不会出现锁等待，因为 score 字段上有索引，更新的是不同行的数据。

然后在会话 A 的事务中，执行 update tt set name='bb' where score=70，这时就会出现锁等待现象了。

最后在会话 B 的事务中，执行 update tt set name='aa' where score=60，这时就出现了相互等待资源的现象，也就是死锁现象发生了。

InnoDB 存储引擎可以自动检测死锁，并自动回滚该事务。

在会话 A 上操作：

```
root@db 10:35:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)

root@db 10:36:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:36:  [zs]> update tt set name='aaa' where score=60;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

root@db 10:37:  [zs]> update tt set name='bb' where score=70;
```

会话 B 上操作：

```
root@db 10:40:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 10:40:  [zs]> update tt set name='a' where score=70;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

root@db 10:41:  [zs]> update tt set name='a' where score=60;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
root@db 10:41:  [zs]>
```

通过 show engine innodb status 可以查看死锁的展示信息：

```
LATEST DETECTED DEADLOCK
------------------------
2017-09-14 10:41:42 0x7fca74945700
*** (1) TRANSACTION:
TRANSACTION 2860, ACTIVE 49 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 5 lock struct(s), heap size 1136, 4 row lock(s), undo log entries 1
MySQL thread id 13, OS thread handle 140507516528384, query id 112 localhost root Searchi
ng rows for update
update tt set name='bb' where score=70
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 28 page no 4 n bits 72 index idx_sc of table `zs`.`tt` trx id 2860
lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 1; hex c6; asc  ;;
 1: len 4; hex 80000002; asc     ;;
```

```
*** (2) TRANSACTION:
TRANSACTION 2861, ACTIVE 37 sec starting index read
mysql tables in use 1, locked 1
5 lock struct(s), heap size 1136, 4 row lock(s), undo log entries 1
MySQL thread id 14, OS thread handle 140507516000000, query id 113 localhost root Searchi
ng rows for update
update tt set name='a' where score=60
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 28 page no 4 n bits 72 index idx_sc of table `zs`.`tt` trx id 2861
lock_mode X
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 1; hex c6; asc  ;;
 1: len 4; hex 80000002; asc     ;;
```

```
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 28 page no 4 n bits 72 index idx_sc of table `zs`.`tt` trx id 2861
lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 1; hex bc; asc  ;;
 1: len 4; hex 80000001; asc     ;;

*** WE ROLL BACK TRANSACTION (2)
```

其实在生产中，我们要防止锁等待现象的发生，出现死锁的问题并不可怕，这里介绍四种避免死锁的方法。

（1）如果不同程序会并发存取多个表，或者涉及多行记录时，尽量约定以相同的顺序访问表，可以大大降低死锁的机会。

（2）业务中尽量采用小事务，避免使用大事务，要及时提交或者回滚事务，可减少死锁产生的概率。

（3）在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率。

（4）对于非常容易产生死锁的业务部分，可以尝试使用升级锁粒度，通过表锁定来减少死锁产生的概率。

8.4 锁问题的监控

通常情况下，当出现锁问题时，我们习惯性通过 show full processlist 和 show engine innodb status 命令来判断事务中锁问题的情况。其实还有特别重要的三张表，即在 information_schema 库下的 INNODB_TRX、INNODB_LOCKS、INNODB_LOCK_WAITS。这三张表可以更方便地来帮助我们监控当前的事务并分析可能存在的锁问题。

通过实验我们来逐一了解一下这三张表。

会话 A

在事务中，在检索 tt 表字段 score<85 的语句上加了一个写锁：

```
root@db 11:17:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 11:17:  [zs]> select * from tt where score<85 for update;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

会话 B

往 tt 表中插入 score=84 记录，出现锁等待超时：

```
root@db 11:17:  [zs]> begin;
Query OK, 0 rows affected (0.00 sec)

root@db 11:17:  [zs]> insert into tt (name,score) values ('dd',84);

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

我们通过上述三张表来分析出现的锁等待问题。

先来看一下 innodb_trx 这张表：

```
root@db 11:17:  [information_schema]> select * from innodb_trx\G;
*************************** 1. row ***************************
                    trx_id: 2870
                 trx_state: LOCK WAIT
               trx_started: 2017-09-14 11:17:36
     trx_requested_lock_id: 2870:28:3:1
          trx_wait_started: 2017-09-14 11:17:36
                trx_weight: 2
       trx_mysql_thread_id: 25
                 trx_query: insert into tt (name,score) values ('dd',84)
       trx_operation_state: inserting
         trx_tables_in_use: 1
         trx_tables_locked: 1
          trx_lock_structs: 2
     trx_lock_memory_bytes: 1136
           trx_rows_locked: 1
         trx_rows_modified: 0
   trx_concurrency_tickets: 0
       trx_isolation_level: REPEATABLE READ
         trx_unique_checks: 1
    trx_foreign_key_checks: 1
trx_last_foreign_key_error: NULL
 trx_adaptive_hash_latched: 0
 trx_adaptive_hash_timeout: 0
*************************** 2. row ***************************
                    trx_id: 2869
                 trx_state: RUNNING
               trx_started: 2017-09-14 11:17:28
     trx_requested_lock_id: NULL
          trx_wait_started: NULL
                trx_weight: 2
       trx_mysql_thread_id: 24
                 trx_query: NULL
       trx_operation_state: NULL
         trx_tables_in_use: 0
         trx_tables_locked: 1
          trx_lock_structs: 2
     trx_lock_memory_bytes: 1136
           trx_rows_locked: 4
         trx_rows_modified: 0
   trx_concurrency_tickets: 0
       trx_isolation_level: REPEATABLE READ
         trx_unique_checks: 1
    trx_foreign_key_checks: 1
trx_last_foreign_key_error: NULL
 trx_adaptive_hash_latched: 0
 trx_adaptive_hash_timeout: 0
         trx_is_read_only: 0
```

主要的字段介绍。

- trx_id：唯一的事务 id 号，2870 和 2869。
- trx_state：当前事务的状态，本例中 2870 事务号是 lock_wait 锁等待状态。
- trx_wait_started：事务开始等待的时间，本例为 2017-09-14 11:17:36。
- trx_mysql_thread_id：线程 id，与 show full processlist 相对应，本例为 25。

- trx_query：事务运行的 SQL 语句，本例为 insert into tt （name，score）values（‘dd’，84）。
- trx_operation_state：事务运行的状态，本例为 inserting。

然后再通过 innodb_lock_waits 和 innodb_locks 两张表来判断持有锁和锁等待的对象。本例中 2870 是锁等待的对象，2869 是持有锁的对象。

```
root@db 11:17:  [information_schema]> show full processlist;
+----+------+-----------+--------------------+---------+------+----------+-------------------
| Id | User | Host      | db                 | Command | Time | State    | Info
+----+------+-----------+--------------------+---------+------+----------+-------------------
| 23 | root | localhost | information_schema | Query   |    0 | starting | show full processlis
| 24 | root | localhost | zs                 | Sleep   |   19 |          | NULL
| 25 | root | localhost | zs                 | Query   |   11 | update   | insert into tt (name
+----+------+-----------+--------------------+---------+------+----------+-------------------
3 rows in set (0.00 sec)
```

```
root@db 11:17:  [information_schema]> select * from innodb_lock_waits\G;
*************************** 1. row ***************************
requesting_trx_id: 2870
requested_lock_id: 2870:28:3:1
  blocking_trx_id: 2869
 blocking_lock_id: 2869:28:3:1
1 row in set, 1 warning (0.00 sec)
```

```
root@db 11:17:  [information_schema]> select * from innodb_locks\G;
*************************** 1. row ***************************
    lock_id: 2870:28:3:1
lock_trx_id: 2870
  lock_mode: X
  lock_type: RECORD
 lock_table: `zs`.`tt`
 lock_index: PRIMARY
 lock_space: 28
  lock_page: 3
   lock_rec: 1
  lock_data: supremum pseudo-record
*************************** 2. row ***************************
    lock_id: 2869:28:3:1
lock_trx_id: 2869
  lock_mode: X
  lock_type: RECORD
 lock_table: `zs`.`tt`
 lock_index: PRIMARY
 lock_space: 28
  lock_page: 3
   lock_rec: 1
  lock_data: supremum pseudo-record
2 rows in set, 1 warning (0.00 sec)
```

第 2 部分　秩序白银篇

在学习完 MySQL 的体系结构之后，我们对 MySQL 数据库有了一个整体的了解。本部分主要讲解 MySQL 的备份恢复。打好基础的同时，也在逐渐提升自己 MySQL 数据库的“段位”。

随着网络时代的飞速发展，各大公司企业对于信息安全的重要性也越来越重视。但是数据备份的重要性可能经常被忽视。这就好比我们的身体正处于一个亚健康的状态，自我感觉良好，但不知危险正在靠近。如果没有数据库的备份，就没有恢复，当服务器意外宕机或者误操作造成大量数据丢失时，对于数据安全性要求很高的行业（金融类、电商类、游戏类）来说，就会造成重大的经济损失。所以提高数据库的高可用性和遇到灾难的可修复性就显得至关重要了。

如果把 MySQL 比喻成一条龙的话，之前说过体系结构是它的龙头，那么备份恢复就构成了龙的身体。保证数据的安全性和可用性是我们作为 DBA 的最基本素质。备份虽然不能促进公司业务的发展，而且还会消耗一些资源和成本。但当数据损坏，或者由于数据误操作，导致重要数据丢失时，如果有备份的话就可以帮助我们恢复核心数据，备份也就成为了我们 DBA 的一棵“救命稻草”。

本部分主要会围绕生产中经常使用的 mysqldump、mydumper 和 xtrabackup 等工具来进行备份恢复的学习。MySQL 5.7 之后又多了一个 mysqlpump 工具来进行多线程备份，但用得很少，也不是特别安全，所以不作为重点的介绍对象。作为一名合格的 DBA，一定要具备可以根据公司不同业务的情况，制定出详细而又合理备份恢复策略。不言而喻，数据就是我们的“命根”，如果连最基本的数据安全性、稳定性、可用性都无法保证，那真的可以“卷铺盖”回家了。

第9章 备份恢复

9.1 MySQL的备份方式

MySQL数据库按照其服务的运行状态（即停库和非停库）分为冷备和热备。热备份又可以分为逻辑备份和裸文件备份。按照备份后的内容量又可以分为全量备份和增量备份，如图 9-1所示。

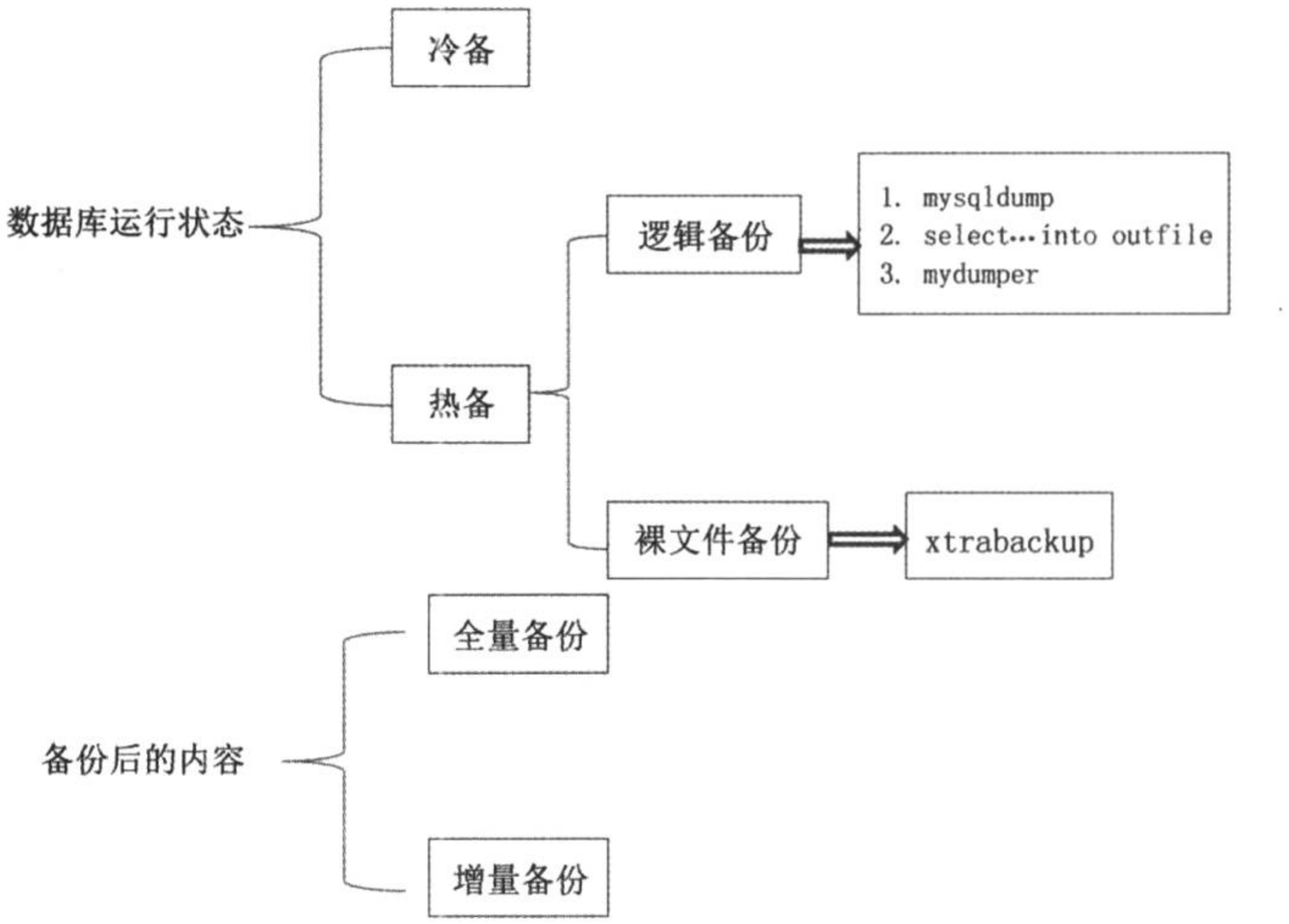


图 9-1　备份的类别

9.2　冷备及恢复

MySQL 备份的方式方法有两种，一种叫冷备，另一种叫热备。

先来介绍一下冷备，顾名思义，冷备就是在数据库处于关闭状态下的备份，好处是可以保证数据库的完整性，备份过程简单并且恢复速度相对快一些，但是数据库的关闭也就意味着会影响现有业务的进行，用户不能再访问你的网站。在一些电商网站店庆搞促销时，如果为了备份，而需要停库，那么带来代价损失将不可估量。所以一般冷备用于不是很重要的、非核心业务上面。

冷备的备份与恢复过程也很简单，仅仅需要如下几步。

首先停掉 MySQL 服务，命令如下：

```
/usr/local/mysql/bin/mysqladmin -uroot -proot123 shutdown
```

备份过程就是复制整个数据目录到远程备份机上或者本地磁盘，命令如下：

```
Scp  -r  /data/mysql/  root@远程备份机 ip:/新的目录
Copy  -r  /data/mysql  /本地新目录
```

恢复过程就更简单了，仅仅需要把已备份的数据目录替换原有的目录就可以了。最后重启 MySQL 服务。

9.3　热备及恢复

与冷备正好相反，热备就是在数据库处于运行状态下的备份，不影响现有业务的正常进行。热备又细分为逻辑备份和裸文件备份。

裸文件备份是基于底层数据文件的 copy datafile。我们利用 Percona 公司发布的一个 XtraBackup 热备份工具来完成裸文件备份。而逻辑备份中有 mysqldump、select ...into outfile、mydumper 三种常用方法来实现备份功能。其实所谓的逻辑备份，就是备份 SQL 语句，在恢复时执行备份 SQL 从而实现数据库数据的重现。备份完成后所形成的文件可以直接编辑。

9.3.1　mysqldump 的备份与恢复

先来介绍 MySQL 数据库中自带的 mysqldump 命令工具。它是最基础的一款备份工具。它的备份过程就是先从 buffer 中找到需要备份的数据进行备份。如果 buffer 中没有，就要去磁盘

中的数据文件中查找并调回到 buffer 里面再备份，最后形成一个可编辑的备份文件。

下面我们来看一看如何利用 mysqldump 来进行备份工作。

首先通过/usr/local/mysql/bin/mysqldump –help 命令来查看这个命令的使用说明。这里介绍几个核心参数，也是生产中用得最多的参数。

- --single-transaction

用于保证 InnoDB 备份数据时的一致性，配合 RR 隔离级别一起使用；当发起事务时，读取一个数据的快照，直到备份结束时，都不会读取到本事务开始之后提交的任何数据（这个参数相当重要）。

- --all-databases（-A）

备份所有的数据库。

- --master-data

该参数有 1 和 2 两个值，如果值等于 1，就会在备份出来的文件中添加一个 CHANGE MASTER 的语句（后期配置搭建主从架构）；如果值等于 2，就会在备份出来的文件中添加一个 CHANGE MASTER 的语句，并在语句前面添加注释符号（后期配置搭建主从架构）。

- --dump-slave

该参数用于在从库端备份数据，在线搭建新的从库时使用。该参数也有 1 和 2 两个值。值为 1 时，也是在备份出来的文件中添加一个 CHANGE MASTER 的语句；值为 2 时，则会在 CHANGE MASTER 命令前增加注释信息。

- --no-create-info（-t）

备份过程中，只备份表数据，并不备份表结构。

- --no-data（-d）

备份过程中，只备份表结构，不备份表数据。

- --complete-insert（-c）

使用完整的 insert 语句会包含表中的列信息，这么做可以提高插入效率。

- --databases（-B）

备份多个数据库。例如：mysqldump –uroot –proot123　--database db1 db2。

- --default-character-set

字符集，MySQL 目前默认的字符集为 UTF8，要与备份出的表的字符集保持一致。

- --quick（-q）

相当于加 sql_no_query，意味着并不会读取缓存中的数据。

- --where=name（-w）

按条件备份出想要的数据。

来看一下备份恢复演示过程，首先备份全库过程，命令如下：

```
/usr/local/mysql/bin/mysqldump  --single-transaction   -uroot  -proot123
-A >all_20170918.sql
```

> 注：这里最好给备份文件标注下时间，方便后期快速找到所需要恢复的文件。如果数据库中已开启 gtid 选项，但备份过程中不想带 gtid 信息，可以加上--set-gtid-purged=OFF 参数。

恢复全库的过程，我们利用 mysql 这个客户端工具来进行，命令如下：

```
/usr/local/mysql/bin/mysql -uroot -proot123 < all_20170918_1447.sql
```

备份单个库 db1 的过程，命令如下：

```
/usr/local/mysql/bin/mysqldump  --single-transaction   -uroot  -proot123
db1 >db1_20170918.sql
```

恢复单库 db1 的过程，命令如下：

```
/usr/local/mysql/bin/mysql -uroot -proot123 db1 < db1_20170918.sql
```

> 注：如果 db1 库存在，则可以直接恢复，如果已经被误删除 drop 掉了，需要在恢复前，先去数据库中创建一个 db1。命令如下：
>
> ```
> create database db1;
> ```
>
> 然后再去做单库的恢复。

备份单表的过程，db1 库下 t 这张表，命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction  -uroot -proot123 db1
t >t_2017_0918.sql
```

恢复表的过程，命令如下：

```
/usr/local/mysql/bin/mysql -uroot -proot123 db1 < t_2017_0918.sql
```

注：恢复单表时，导入符号之前不需要写表的名字，只需要写库的名字即可。

备份 db1 库 t 表中的表结构信息，命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction  -uroot -proot123 db1
t -d >t.sql
```

```
-- Table structure for table `t`
--

DROP TABLE IF EXISTS `t`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
  `city` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `b` (`name`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
```

备份 db1 库 t 表中的数据信息，命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction  -uroot -proot123 db1
t -t >t.sql
```

```
-- Dumping data for table `t`
--

LOCK TABLES `t` WRITE;
/*!40000 ALTER TABLE `t` DISABLE KEYS */;
INSERT INTO `t` VALUES (1,'a','bj'),(2,'b','sh'),(3,'c','gz'),(4,'d','sz'),(5,'e','fj'),(
6,'f','tj');
/*!40000 ALTER TABLE `t` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
```

备份 db1 库 t 表中 id>3 的记录，命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction  -uroot -proot123 db1
t --where="id>3" >t.sql
```

```
--
-- Dumping data for table `t`
--
-- WHERE:  id>3

LOCK TABLES `t` WRITE;
/*!40000 ALTER TABLE `t` DISABLE KEYS */;
INSERT INTO `t` VALUES (4,'d','sz'),(5,'e','fj'),(6,'f','tj');
/*!40000 ALTER TABLE `t` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
```

> 注：WHERE 条件后面记得要加双引号（“”），否则不会被识别。

在使用 mysqldump 备份时，很有可能会遇到数据库性能抖动的问题，出现性能急剧下降的现象。为什么一个简单的备份，会造成数据库的性能下降呢？我们来回忆一下 mysqldump 的备份过程，之前介绍过 mysqldump 备份是先从 buffer 中找想要备份的内容，如果 buffer 中没有，就需要访问磁盘中的数据文件，然后把数据调回到内存，再形成备份文件。问题的关键就在于把数据从磁盘中调回到内存时，就有可能把内存里面的热数据给冲掉了，这样就影响了我们现有业务的访问了。

自 MySQL 5.7 之后，新增了一个 innodb_buffer_pool_dump_pct 参数，用它来控制每个 innodb buffer 中转储活跃使用的 innodb buffer pages 的比例，只有当数据在 1s 内再次被访问时，才能放到热区域内，这样就避免了热数据被冲走的情况。该参数默认值为 25%：

```
root@db 11:38:  [zs]> show variables like '%innodb_buffer_pool_dump_pct%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_buffer_pool_dump_pct   | 25    |
+-------------------------------+-------+
1 row in set (0.00 sec)
```

9.3.2 select ...into outfile

select...into outfile 是我比较喜欢用的一种逻辑备份方法，因为它的恢复速度非常快，比 insert 的插入速度要快很多。它跟有许多备份功能的 mysqldump 不同的是，它只能备份表中的数据，并不能包含表的结构。如果备份完成之后，表被“drop”了，是无法实现恢复操作的。它把备份出的数据导出到一个文本文件中，通过 load data 的方式，实现恢复还原的操作。

常用语法：select col1,col2... from table_name into outfile ‘/path/备份文件名称’。

通过实验来了解一下该备份过程。

把 tt 表中的数据全部导出，备份文件 tt.sql 放到 tmp 目录下：

```
root@db 13:44:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.32 sec)

root@db 13:44:  [zs]> select * from tt into outfile '/tmp/tt.sql';
Query OK, 3 rows affected (0.01 sec)

root@db 13:44:  [zs]> exit
Bye
[root@node3 ~]# cat /tmp/tt.sql
1       aa      60
2       bb      70
3       cc      80
```

如果再次把备份文件导出到 tmp 目录下，并还以 tt.sql 命名，就会报出文件已存在的错误。

```
root@db 13:53:  [zs]> select * from tt into outfile '/tmp/tt.sql';
ERROR 1086 (HY000): File '/tmp/tt.sql' already exists
```

恢复的过程中，把 tt 表中的数据删掉，用 load data 的方式导入已备份的数据文件。

```
root@db 13:55:  [zs]> delete from tt;
Query OK, 3 rows affected (0.35 sec)

root@db 13:55:  [zs]> select * from tt;
Empty set (0.00 sec)
```

```
root@db 13:55:  [zs]> LOAD DATA INFILE '/tmp/tt.sql' INTO TABLE zs.tt;
Query OK, 3 rows affected (0.02 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0

root@db 13:57:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.00 sec)
```

> 小技巧：如果忘记 load data 的语法，可以直接在命令前加“?”。

如：? Load data，这样系统就会把正确的语法展示出来。

```
LOAD DATA INFILE 'data.txt' INTO TABLE db2.my_table;
```

9.3.3　load data 与 insert 的插入速度对比

本次测试用例中，准备 10 万条 SQL 语句进行插入测试演练。

test1 表的数据量为 10 万条 SQL 语句：

```
root@db 13:38:  [zs]> select count(*) from test1;
+----------+
| count(*) |
+----------+
|   100000 |
+----------+
1 row in set (0.02 sec)
```

load data 的测试如下。

首先需要使用 select...into outfile 把 10 万条 SQL 语句“dump”出来：

```
root@db 13:38:  [zs]> select * from test1 into outfile '/tmp/test1.sql';
Query OK, 100000 rows affected (0.07 sec)
```

然后模拟故障，把 test1 数据都删除：

```
root@db 13:40:  [zs]> truncate table test1;
Query OK, 0 rows affected (0.33 sec)
```

最后进行 load data，导入数据：

```
[root@node3 ~]# time mysql -uroot -proot123 -e "LOAD DATA INFILE '/tmp/test1.sql' I
NTO TABLE zs.test1"

real    0m1.402s
user    0m0.003s
sys     0m0.000s
```

可以看到用 load data 导入 10 万条 SQL 语句的时间为 1.4s。

> 注：mysql -e 可以调用数据库的命令行命令。

insert 插入数据的测试如下。

首先利用 sqlyog 工具把 test1 表中的数据全部备份出来，如图 9-2 所示。

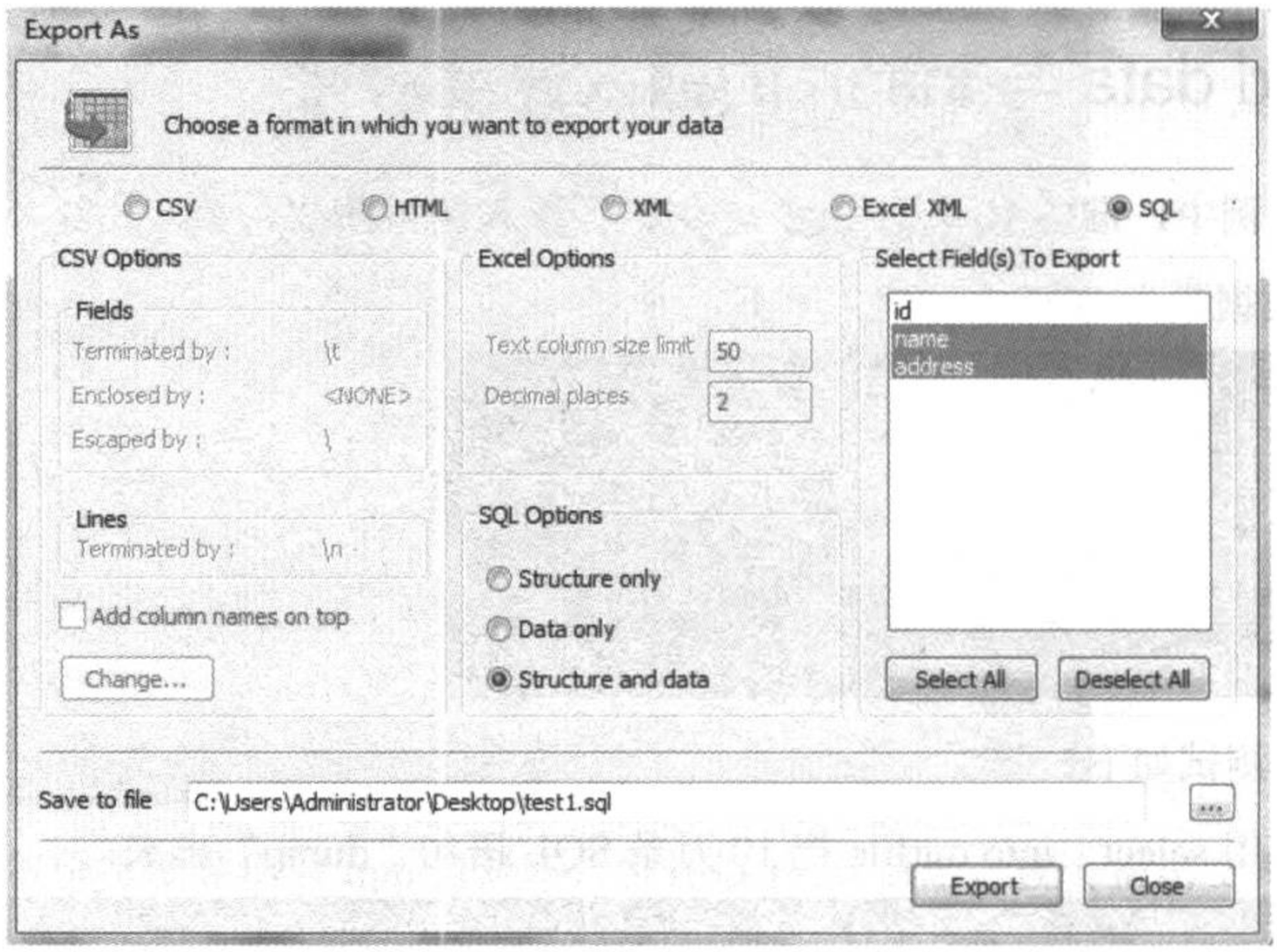


图 9-2　数据备份

备份文件 test1.sql 的内容就是由 10 万条 insert 语句组成的，我们可以用 begin 和 commit 把 10 万条事务语句变成 1 条事务插入：

```
begin;
insert into `test1` (`name`, `address`) values('name1','address1');
insert into `test1` (`name`, `address`) values('name2','address2');
insert into `test1` (`name`, `address`) values('name3','address3');
…
insert into `test1` (`name`, `address`) values('name99998','address99998');
insert into `test1` (`name`, `address`) values('name99999','address99999');
insert into `test1` (`name`, `address`) values('name100000','address100000');
commit
```

然后把 test1 的数据删除：

```
root@db 13:56:  [zs]> truncate table test1;
Query OK, 0 rows affected (0.03 sec)
```

最后导入数据：

```
[root@node3 ~]# time mysql -uroot -proot123 zs < test1.sql


real    0m12.159s
user    0m1.389s
sys     0m0.637s
```

insert 语句的插入耗时大约为 12.2s，load data 方式大概是它的 12 倍。

9.3.4　mydumper

MySQL 自带的 mysqldump 是一个只能支持单线程工作的工具，只能逐个导出表，而 mydumper 是一个针对 MySQL 和 Drizzle 的高性能多线程的备份工具，备份速度远远高于 mysqldump，其备份方式也属于逻辑备份，数据还原时我们使用 myloader 工具。作为 DBA 的我们必须掌握这款备份界的“小钢炮”。

1. 安装 mydumper

下载地址链接：https://launchpad.net/mydumper/+download。

首先准备安装环境，配置好 Yum 源，安装一些依赖的软件包：

```
yum install cmake*
yum install glib2-devel mysql-devel zlib-devel pcre-devel openssl-devel
tar -zxvf mydumper-0.6.2.tar.gz
cd mydumper-0.6.2
cmake.
make
make install
```

2.mydumper 和 myloader 的语法重点参数详解

Mydumper 的重点参数说明。

- -B, --database：需要备份的数据库。
- -T, --tables-list：需要备份的表，多表间用逗号分隔。
- -o, --outputdir：输出文件的目录。
- -s, --statement-size：生成的 insert 语句的字节数，默认为 1000000。
- -r, --rows：将表按行分块时，指定的块行数，指定这个选项会关闭--chunk-filesize。
- -F, --chunk-filesize：将表按大小分块时，指定的块大小，单位是 MB。
- -c, --compress：压缩输出文件。
- -e, --build-empty-files：即使表没有数据，还是会产生一个空文件。
- -x, --regex：正则表达式:'db.table'。
- -i, --ignore-engines：忽略的存储引擎，用逗号分隔。

- -m, --no-schemas：不导出表结构。
- -k, --no-locks：不执行共享读锁，警告，这将导致不一致的备份。
- -l, --long-query-guard：设置长查询时间，默认为 60 秒。
- -K, --kill-long-queries：“kill”掉长时间执行的查询。
- -D, --daemon：启用守护进程模式。
- -I, --snapshot-interval dump：快照间隔时间，默认为 60s，需要在 daemon 模式下。
- -L, --logfile：日志文件。
- -h,--host：MySQL 服务器 IP 地址。
- -u, --user：备份时使用的用户名。
- -p, --password：用户密码。
- -P, --port：数据库的连接端口。
- -S, --socket：套接字文件。
- -t, --threads：使用的线程数，默认为 4 个。
- -C, --compress-protocol：在 MySQL 连接上使用压缩协议。

myloader 的重点参数说明。

- -d, --directory：备份文件的目录。
- -q, --queries-per-transaction：每次事务执行的查询数量，默认是 1000。
- -o, --overwrite-tables：如果要恢复的表存在，则先“drop”掉该表。
- -B, --database：需要还原的数据库。
- -e, --enable-binlog：启用还原数据的二进制日志。
- -h, --host：MySQL 服务器 IP 地址。
- -u, --user：还原时使用的用户。
- -p, --password：用户的密码。
- -P, --port：连接数据库的端口号。
- -S, --socket：套接字文件。
- -t, --threads：还原所使用的线程数，默认是 4 个。
- -C, --compress-protocol：压缩协议。

备份恢复过程展示。

备份全库，命令如下：

```
mydumper -u root -p root123  -o /data/backup/
```

备份 zs 库，命令如下：

```
mydumper -u root -p root123 -B zs -o /data/backup/
```

```
[root@node3 ~]# mkdir -p /data/backup/
[root@node3 ~]# mydumper -u root -p root123 -B zs -o /data/backup/
[root@node3 ~]# cd /data/backup/
[root@node3 backup]# ll
total 24
-rw-r--r--. 1 root root 128 Sep 19 16:19 metadata
-rw-r--r--. 1 root root 306 Sep 19 16:19 zs.tt-schema.sql
-rw-r--r--. 1 root root 165 Sep 19 16:19 zs.tt.sql
-rw-r--r--. 1 root root 188 Sep 19 16:19 zs.ttt-schema.sql
-rw-r--r--. 1 root root 179 Sep 19 16:19 zs.ttt.sql
-rw-r--r--. 1 root root 223 Sep 19 16:19 zs.zs-schema.sql
```

备份目录下会有一个 metadata 文件，该文件记录着当前的 binlog 和 position 号，方便日后搭建 slave 库。

```
[root@node3 backup]# cat metadata
Started dump at: 2017-09-19 16:19:23
SHOW MASTER STATUS:
        Log: mybinlog.000007
        Pos: 744

Finished dump at: 2017-09-19 16:19:23
```

还原 zs 库，命令如下：

```
myloader -u root -p root123 -B zs -d /data/backup/
```

备份 zs 库下 tt 这张表，命令如下：

```
mydumper -u root -p root123 -B zs -T tt -o /data/backup/
```

```
[root@node3 backup]# mydumper -u root -p root123 -B zs -T tt -o /data/backup/
[root@node3 backup]# cd /data/backup/
[root@node3 backup]# ll
total 12
-rw-r--r--. 1 root root 128 Sep 19 16:07 metadata
-rw-r--r--. 1 root root 306 Sep 19 16:07 zs.tt-schema.sql
-rw-r--r--. 1 root root 165 Sep 19 16:07 zs.tt.sql
```

还原 tt 这张表的命令如下：

```
myloader -u root -p root123 -B zs -o tt  -d /data/backup/
```

备份 zs 库下多张表 tt 和 ttt，命令如下：

```
mydumper -u root -p root123 -B zs -T tt,ttt -o /data/backup/
```

```
[root@node3 backup]# mydumper -u root -p root123 -B zs -T tt,ttt -o /data/backup/
[root@node3 backup]# cd /data/backup/
[root@node3 backup]# ll
total 20
-rw-r--r--. 1 root root 129 Sep 19 16:31 metadata
-rw-r--r--. 1 root root 306 Sep 19 16:31 zs.tt-schema.sql
-rw-r--r--. 1 root root 165 Sep 19 16:31 zs.tt.sql
-rw-r--r--. 1 root root 188 Sep 19 16:31 zs.ttt-schema.sql
-rw-r--r--. 1 root root 179 Sep 19 16:31 zs.ttt.sql
```

备份 zs 库下的 tt 表的表的数据，不备份表结构，命令如下：

```
mydumper -u root -p root123 -B zs -T tt -m -o /data/backup/
```

```
[root@node3 backup]# mydumper -u root -p root123 -B zs -T tt -m -o /data/backup/
[root@node3 backup]# cd /data/backup/
[root@node3 backup]# ll
total 8
-rw-r--r--. 1 root root 129 Sep 19 16:36 metadata
-rw-r--r--. 1 root root 165 Sep 19 16:36 zs.tt.sql
[root@node3 backup]# cat zs.tt.sql
/*!40101 SET NAMES binary*/;
/*!40014 SET FOREIGN_KEY_CHECKS=0*/;
/*!40103 SET TIME_ZONE='+00:00' */;
INSERT INTO `tt` VALUES
(1,"aa",60),
(2,"bb",70),
(3,"cc",80);
```

备份 zs 库下 tt 这张表，并进行压缩：

```
[root@node3 backup]# mydumper -u root -p root123 -B zs -T tt -c -o /data/backup/
[root@node3 backup]# ll
total 12
-rw-r--r--. 1 root root 133 Sep 19 17:19 metadata
-rw-r--r--. 1 root root 240 Sep 19 17:19 zs.tt-schema.sql.gz
-rw-r--r--. 1 root root 160 Sep 19 17:19 zs.tt.sql.gz
```

3.mysqldump 和 mydumper 的速度对比

首先使用 mysqldump 进行全库备份，时间消耗为 0.25s，由于是测试环境，数据不是很大：

```
[root@node3 backup]# time /usr/local/mysql/bin/mysqldump --set-gtid-purged=OFF --single-
transaction -uroot -proot123 -A > all.sql
mysqldump: [Warning] Using a password on the command line interface can be insecure.

real    0m0.254s
user    0m0.073s
sys     0m0.013s
```

再使用 mydumper 进行全库备份，时间消耗为 0.18s：

```
[root@node3 backup]# time mydumper -u root -p root123   -o /data/backup/

real    0m0.188s
user    0m0.018s
sys     0m0.030s
```

虽然数据量小，但我们也能看出差距，使用 mydumper 工具进行备份，消耗的时间明显比 mysqldump 要少。

备份过程测试完成之后，我们再看看恢复过程的时间对比。

使用 mysql 工具还原 zs 库的数据，时间消耗为 2.15s：

```
[root@node3 ~]# time mysql -uroot -proot123 zs < zs.sql
mysql: [Warning] Using a password on the command line interface can be insecure.

real    0m2.150s
user    0m0.056s
sys     0m0.003s
```

使用 myloader 工具还原 zs 库的数据，时间消耗为 1.55s：

```
[root@node3 ~]# time myloader -u root -p root123 -B zs   -d /data/backup/

real    0m1.557s
user    0m0.012s
sys     0m0.004s
```

可见使用 myloader 工具进行数据的还原，比 mysql 要快。

4. mydumper 优点总结

- 多线程备份工具。
- 支持文件压缩功能。
- 支持多线程恢复。
- 保证数据的一致性。
- 比 mysqldump 备份速度和恢复速度都要快。

9.3.5　裸文件备份 XtraBackup

热备中两种主要的方式就是逻辑备份和裸文件备份，裸文件备份要比逻辑备份在速度上更快一些，因为它是在底层复制数据文件的，比起一条条 SQL 地插入恢复要快很多。裸文件备份的代表作品就是 XtraBackup，它是 Percona 公司的开源项目，据官方介绍它是世界上唯一一款开源的能够对 InnoDB 和 XtraDB 数据库进行热备的工具。它的优点就是备份与恢复过程的速度很快，安全可靠，而且在备份过程中不会锁表，不影响现有业务。但它目前还是不能对表结构文件和其他非事务类型的表进行备份。XtraBackup 包含了两个主要工具，一个是 xtrabackup，另一个是 innobackupex。在现在最新的版本中，innobackupex 是 XtraBackup 的一个软连接，之

前的版本 innobackupex 是一个 Perl 脚本，它既可以备份 InnoDB 表，也可以满足备份 MyISAM 这类非事务表的需要，但会在备份过程中加锁。

XtraBackup 的原理及安装过程

原理解析：对于 InnoDB 来说，XtraBackup 是基于 InnoDB 的 crash recovery 功能进行备份的。那什么是 crash recovery 呢？

InnoDB 内部维护了一个 redo log，又叫事务日志（transaction log），它包含了 InnoDB 数据的所有更改信息。在 InnoDB 启动时，会先检查 datafile 和 transaction log，然后前滚所有已提交的事务并且回滚所有未提交的事务。

XtraBackup 在备份时并不锁定表，而是一页页地去复制 InnoDB 的数据，这样复制出来的数据是不一致的。要想保证数据的一致性，需要在恢复时使用 crash recovery 进行操作。XtraBackup 还有另外一个线程监视着 redo log，之前章节中介绍过，redo log 大小有限，而且写的方式是循环写和顺序写。当写满最后一个日志后，就会从头开始再写，那就有可能会覆盖之前的数据信息，所以一旦日志发生变化，就复制变化过的 log pages。在复制全部数据文件完之后，停止复制 redo log。

软件包下载地址：https://www.percona.com/downloads/XtraBackup/LATEST/。

> 注：最好选择一个最新版本下载，这样对低版本的 MySQL 也可以进行备份操作，起到了很好的兼容效果。

首先解压软件包：

```
tar -zxvf percona-xtrabackup-2.4.7-Linux-x86_64.tar.gz
```

然后进入工具包的 bin 目录下：

```
cd percona-xtrabackup-2.4.7-Linux-x86_64/bin
```

```
[root@node3 bin]# ll
total 220976
lrwxrwxrwx. 1 root root        10 Sep 21 10:00 innobackupex -> xtrabackup
-rwxr-xr-x. 1 root root   5090481 May 22 20:56 xbcloud
-rwxr-xr-x. 1 root root      3020 May 22 20:55 xbcloud_osenv
-rwxr-xr-x. 1 root root   4934822 May 22 20:56 xbcrypt
-rwxr-xr-x. 1 root root   5041328 May 22 20:56 xbstream
-rwxr-xr-x. 1 root root 211205827 May 22 21:01 xtrabackup
```

可以通过./innobackup –help 来查看各种参数的具体含义。

1. 全备过程

首先创建备份所需要的用户名和密码：

```
root@db 11:05:  [mysql]> create user 'zsbak'@'192.168.56.%' identified by 'zsbak';
Query OK, 0 rows affected (0.01 sec)

root@db 11:05:  [mysql]> grant reload,lock tables,replication client,process,super on *.*
 to 'zsbak'@'192.168.56.%';
Query OK, 0 rows affected (0.00 sec)

root@db 11:06:  [mysql]> flush privileges;
Query OK, 0 rows affected (0.05 sec)
```

然后创建备份目录/data/backup：

```
[root@node3 ~]# mkdir -p /data/backup/
```

开始全备操作，不采用系统默认创建的备份集文件名称，使用--no-timestamp 参数，自己给备份文件命名。这里的备份文件名为 all-20170921bak。

以 zs 数据库中的 tt 为基准，以便后期恢复看到实验效果。

```
root@db 11:28:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.31 sec)
```

全备命令如下：

```
innobackupex    --defaults-file=/etc/my.cnf --no-timestamp --user zsbak
--host 192.168.56.102 --password zsbak  /data/backup/all-20170921bak
```

备份成功之后，会显示 completed OK：

```
170921 11:26:06 Backup created in directory '/data/backup/all-20170921bak/'
MySQL binlog position: filename 'mybinlog.000008', position '1671', GTID of the las
t change 'c64f2211-983a-11e7-b17a-080027cd683a:1-100078'
170921 11:26:06 [00] Writing backup-my.cnf
170921 11:26:06 [00]        ...done
170921 11:26:06 [00] Writing xtrabackup_info
170921 11:26:06 [00]        ...done
xtrabackup: Transaction log of lsn (55386741) to (55386750) was copied.
170921 11:26:06 completed OK!
```

> 注：这里需要注意，在 innobackupex 命令后面需要紧跟--defaults-file 参数，否则会报错。

```
[root@node3 bin]# ./innobackupex --no-timestamp  --defaults-file=/etc/my.cnf  --use
r zsbak --host 192.168.56.102 --password zsbak  /data/backup/all-20170921bak
xtrabackup: Error: --defaults-file must be specified first on the command line
```

我们来看看备份完成后，在/data/backup 目录下都有哪些文件。

```
-rw-r-----. 1 root root        431 Sep 21 11:26 backup-my.cnf
-rw-r-----. 1 root root        407 Sep 21 11:26 ib_buffer_pool
-rw-r-----. 1 root root 1073741824 Sep 21 11:26 ibdata1
drwxr-x---. 2 root root       4096 Sep 21 11:26 mysql
drwxr-x---. 2 root root       4096 Sep 21 11:26 performance_schema
drwxr-x---. 2 root root      12288 Sep 21 11:26 sys
-rw-r-----. 1 root root         67 Sep 21 11:26 xtrabackup_binlog_info
-rw-r-----. 1 root root        115 Sep 21 11:26 xtrabackup_checkpoints
-rw-r-----. 1 root root        630 Sep 21 11:26 xtrabackup_info
-rw-r-----. 1 root root       2560 Sep 21 11:26 xtrabackup_logfile
drwxr-x---. 2 root root       4096 Sep 21 11:26 zs
```

这里有备份的数据库名字，还有几个以 xtrabackup 开头的文件。

xtrabackup_checkpoints 文件记录备份完成时检查点的 lsn 号和该备份文件的类型，此次是一次全量备份，标识为 full-backuped。

```
[root@node3 all-20170921bak]# cat xtrabackup_checkpoints
backup_type = full-backuped
from_lsn = 0
to_lsn = 55386741
last_lsn = 55386750
compact = 0
recover_binlog_info = 0
```

xtrabackup_binlog_info 文件记录当前二进制日志和偏移量，如果开启 gtid 功能，还会记录 gtid 位置的信息，为在线搭建从库做好准备。

```
[root@node3 all-20170921bak]# cat xtrabackup_binlog_info
mybinlog.000008 1671     c64f2211-983a-11e7-b17a-080027cd683a:1-100078
```

xtrabackup_info 文件记录备份的详细信息，如备份命令、备份的开始结束时间、MySQL 版本和 XtraBackup 版本等。

```
uuid = 9acfa106-9e7c-11e7-a0cf-080027cd683a
name =
tool_name = innobackupex
tool_command = --defaults-file=/etc/my.cnf --no-timestamp --user zsbak --host 192.1
68.56.102 --password=... zsbak /data/backup/all-20170921bak
tool_version = 2.4.7
ibbackup_version = 2.4.7
server_version = 5.7.14-log
start_time = 2017-09-21 11:25:46
end_time = 2017-09-21 11:26:06
lock_time = 0
binlog_pos = filename 'mybinlog.000008', position '1671', GTID of the last change '
c64f2211-983a-11e7-b17a-080027cd683a:1-100078'
innodb_from_lsn = 0
innodb_to_lsn = 55386741
```

2. 全备的恢复

我们首先模拟故障，把 zs 库删掉，那么 zs 库里面的 tt 表也就跟着删除了。

```
root@db 12:11:  [(none)]> drop database zs;
Query OK, 4 rows affected (0.15 sec)

root@db 12:11:  [(none)]>
root@db 12:11:  [(none)]>
root@db 12:11:  [(none)]> select * from tt;
ERROR 1046 (3D000): No database selected
root@db 12:11:  [(none)]>
root@db 12:11:  [(none)]>
root@db 12:11:  [(none)]> select * from zs.tt;
ERROR 1146 (42S02): Table 'zs.tt' doesn't exist
```

然后进行恢复操作，恢复时需要加--apply-log 参数，它的作用就是通过回滚未提交的事务及同步已经提交的事务至数据文件，使数据文件处于一致性状态。

命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --user zsbak --host 192.168.56.102
--password zsbak  --apply-log  /data/backup/all-20170921bak
```

```
InnoDB: 5.7.13 started; log sequence number 55387157
xtrabackup: starting shutdown with innodb_fast_shutdown = 1
InnoDB: FTS optimize thread exiting.
InnoDB: Starting shutdown...
InnoDB: Shutdown completed; log sequence number 55387176
170921 12:14:17 completed OK!
```

出现“completed OK”证明备份集校验成功，离最后的数据还原只差一步。

最后一步恢复操作，我们需要先停掉 MySQL 实例，重命名原来的数据目录，改名为/data/mysql_bak，把新的备份集“mv”到/data 下，改名为 mysql，并赋予 mysql 的权限，再重启 MySQL 实例。之前误操作的数据就可以被恢复回来了。命令如下：

```
mysqladmin -uroot -proot123 shutdown
mv /data/mysql /data/mysql_bak
cd  /data/backup/
mv all-20170921bak  /data/
cd /data/
mv all-20170921bak/ mysql
chown mysql:mysql -R mysql
/usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf  &
```

查看恢复结果，zs 库已经恢复成功，也可以查看 tt 表中的数据。

```
root@db 12:32:  [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| zs                 |
+--------------------+
5 rows in set (0.30 sec)
```

```
root@db 12:32:  [(none)]> use zs;
Database changed
root@db 12:32:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.04 sec)
```

恢复完成之后，再查看新的数据目录底下的文件有哪些：

```
-rw-r-----. 1 mysql mysql       67 Sep 21 11:26 xtrabackup_binlog_info
-rw-r--r--. 1 mysql mysql       25 Sep 21 12:13 xtrabackup_binlog_pos_innodb
-rw-r-----. 1 mysql mysql      115 Sep 21 12:13 xtrabackup_checkpoints
-rw-r-----. 1 mysql mysql      630 Sep 21 11:26 xtrabackup_info
-rw-r-----. 1 mysql mysql  8388608 Sep 21 12:13 xtrabackup_logfile
drwxr-x---. 2 mysql mysql     4096 Sep 21 11:26 zs
```

可见这时比原来备份的时候多了一个 xtrabackup_binlog_pos_innodb 文件，来看一下该文件中的内容，发现也记录着 binlog 文件和 position 号：

```
[root@node3 mysql]# cat xtrabackup_binlog_pos_innodb
mybinlog.000007 30775848
```

搭建 MySQL 主从时，我们应该以 xtrabackup_binlog_info 这个文件，还是以 xtrabackup_binlog_pos_innodb 这个文件作为标准来获取 binlog 和 position 号呢？

如果都是 InnoDB 存储引擎的表，可以以 xtrabackup_binlog_pos_innodb 文件中记录的为准。如果是 InnoDB 和其他存储引擎混合操作的表，建议以 xtrabackup_binlog_info 为准。

3. XtraBackup 增备

XtraBackup 增备原理：

在上述实验过程中，我们看到了在全量备份集中有 xtrabackup_checkpoints 文件，里面记录

着备份完成时检查点的 lsn。其实在增量备份文件中也有这个 xtrabackup_checkpoints 文件，在进行新的增量备份时，XtraBackup 会比较表空间中每页的 lsn 是否大于上次备份完成时的 lsn，如果大于，则备份该页，并记录当前检查点的 lsn。

增量备份是基于全备而言的，第一次的增备数据必须要基于上一次的全备，之后的每次增备都是基于上一次的增备，最终达到一致性的增备。

下面开始增备恢复实验演练。

在 2017 年 9 月 21 日这天对数据库进行一次全备，命令如下：

```
    innobackupex   --defaults-file=/etc/my.cnf --no-timestamp --user zsbak
--host 192.168.56.102 --password zsbak  /data/backup/all-20170921bak
```

```
170921 15:30:03 [00] Writing backup-my.cnf
170921 15:30:03 [00]            ...done
170921 15:30:03 [00] Writing xtrabackup_info
170921 15:30:03 [00]            ...done
xtrabackup: Transaction log of lsn (55397612) to (55397621) was copied.
170921 15:30:04 completed OK!
```

备份完成之后查看此时 zs 库下 tt 表的数据信息：

```
root@db 15:30:  [(none)]> use zs;
^[[ADatabase changed
root@db 15:30:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
+----+------+-------+
3 rows in set (0.02 sec)
```

再来看一下 xtrabackup_checkpoints 文件中记录的内容，全备的初始 lsn=0，备份结束时的 lsn=55397612。

```
[root@node3 all-20170921bak]# cat xtrabackup_checkpoints
backup_type = full-backuped
from_lsn = 0
to_lsn = 55397612
last_lsn = 55397621
compact = 0
recover_binlog_info = 0
```

然后往 tt 表中插入一条数据：

```
root@db 15:31:  [zs]> insert into tt (name,score) values ('dd',90);
Query OK, 1 row affected (0.31 sec)
```

假设已经到了 9 月 22 日，然后进行第一次增备操作。增备过程需要要添加--incremental 参数，增备目录文件为/data/backup/all-20170922incr。而且还需要使用--incremental-basedir 参数，它代表在全备的基础上做的增备。

命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --no-timestamp --user zsbak --host
192.168.56.102 --password zsbak  --incremental /data/backup/all-20170922incr
--incremental-basedir=/data/backup/all-20170921bak
```

```
170921 15:33:51 [00] Writing backup-my.cnf
170921 15:33:51 [00]        ...done
170921 15:33:51 [00] Writing xtrabackup_info
170921 15:33:51 [00]        ...done
xtrabackup: Transaction log of lsn (55397953) to (55397962) was copied.
170921 15:33:51 completed OK!
```

出现“completed OK”代表增备成功。

这时我们看一下增备目录底下的 xtrabackup_checkpoints 文件：

```
[root@node3 all-20170922incr]# cat xtrabackup_checkpoints
backup_type = incremental
from_lsn = 55397612
to_lsn = 55397953
last_lsn = 55397962
compact = 0
recover_binlog_info = 0
```

可以看到备份类型属于增备，备份的初始 lsn 是全备结束的 lsn，是 55397612，之后的增备以此类推。

我们再往 tt 表中插入一条数据：

```
root@db 15:35:   [zs]> insert into tt (name,score) values ('ee',100);
Query OK, 1 row affected (0.03 sec)
```

假设已经到了 9 月 23 日，然后进行第二次增备操作。可见 9 月 23 日的增备就是在 9 月 22 日的增备基础上进行的。

命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --no-timestamp --user zsbak --host
192.168.56.102 --password zsbak --incremental /data/backup/all-20170923incr2
--incremental-basedir=/data/backup/all-20170922incr
```

```
170921 15:36:45 [00] Writing backup-my.cnf
170921 15:36:45 [00]        ...done
170921 15:36:45 [00] Writing xtrabackup_info
170921 15:36:45 [00]        ...done
xtrabackup: Transaction log of lsn (55398278) to (55398287) was copied.
170921 15:36:45 completed OK!
```

结果显示第二个增备成功。

查看第二个增备目录下的 xtrabackup_checkpoints 文件，可见该增备的起始 lsn 是第一个增备结束的 lsn：

```
[root@node3 all-20170923incr2]# cat xtrabackup_checkpoints
backup_type = incremental
from_lsn = 55397953
to_lsn = 55398278
last_lsn = 55398287
compact = 0
recover_binlog_info = 0
```

4. 增备的恢复

先来模拟故障，删掉 tt 这张表：

```
root@db 15:38:  [zs]> drop table tt;
Query OK, 0 rows affected (0.04 sec)

root@db 15:38:  [zs]> select * from tt;
ERROR 1146 (42S02): Table 'zs.tt' doesn't exist
```

再进行增备恢复之前，我们先要弄清增备恢复需要涉及的几个过程。首先需要进行全备恢复。然后再进行把增备文件恢复到全备中的操作，前两个过程中注意需要添加--redo-only 参数，它意味着只前滚 XtraBackup 日志中已经提交的事务，并不回滚那些还没有提交的事务信息。最后一个过程就是对整体的全备进行恢复，这时就可以去掉--redo-only 参数了，意味着需要回滚那些还没有提交的事务。

恢复过程展现

首先先恢复全备，命令如下：

```
innobackupex --defaults-file=/etc/my.cnf --user zsbak
--host=192.168.56.102 --password zsbak --apply-log --redo-only
/data/backup/all-20170921bak
```

```
xtrabackup: starting shutdown with innodb_fast_shutdown = 1
InnoDB: Starting shutdown...
InnoDB: Shutdown completed; log sequence number 55397630
InnoDB: Number of pools: 1
170921 15:39:55 completed OK!
```

接下来进行增备恢复，先恢复第一个增备文件，把它恢复到全备文件中，命令如下：

```
    innobackupex --defaults-file=/etc/my.cnf  --user zsbak
--host=192.168.56.102 --password zsbak   --apply-log --redo-only
/data/backup/all-20170921bak --incremental-dir=/data/backup/all-20170922incr
```

```
170921 15:40:57 [00] Copying /data/backup/all-20170922incr//xtrabackup_info to ./xt
rabackup_info
170921 15:40:57 [00]        ...done
170921 15:40:57 completed OK!
```

第一个增备恢复成功。

之后再把第二个增备文件恢复到全备中，命令如下：

```
    innobackupex --defaults-file=/etc/my.cnf  --user zsbak --host=
192.168.56.102 --password zsbak   --apply-log --redo-only /data/backup/
all-20170921bak --incremental-dir=/data/backup/all-20170923incr2
```

```
170921 15:41:45 [00] Copying /data/backup/all-20170923incr2//xtrabackup_info to ./x
trabackup_info
170921 15:41:45 [00]        ...done
170921 15:41:45 completed OK!
```

第二个增备恢复成功。

最后一步，就是把新的全备文件进行一次完全恢复，回滚那些还未提交的数据。注：这里就不需要添加--redo-only 参数了。

命令如下：

```
    ./innobackupex  --defaults-file=/etc/my.cnf --user zsbak  --host=
192.168.56.102 --password zsbak  --apply-log  /data/backup/all-20170921bak
```

```
InnoDB: Waiting for purge to start
InnoDB: 5.7.13 started; log sequence number 55398440
xtrabackup: starting shutdown with innodb_fast_shutdown = 1
InnoDB: FTS optimize thread exiting.
InnoDB: Starting shutdown...
InnoDB: Shutdown completed; log sequence number 55398459
170921 15:42:39 completed OK!
```

这时就可以实现最后的恢复操作了，跟之前讲过的全备恢复一样。我们需要先停掉 MySQL 实例，重命名原来的数据目录改名为/data/mysql_bak，把新的备份集“mv”到/data 下，改名为 mysql，并赋予 mysql 的权限，再重启 MySQL 实例。之前误操作的数据就可以被恢复回来了。命令如下：

```
mysqladmin -uroot -proot123 shutdown
mv /data/mysql /data/mysql_bak
cd  /data/backup/
mv all-20170921bak  /data/
cd /data/
mv all-20170921bak/ mysql
chown mysql:mysql -R mysql
/usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf  &
```

验证数据，查看 tt 表中的数据已经全部恢复成功。

```
root@db 15:46:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.04 sec)
```

9.4 流式化备份

前面介绍了 XtraBackup 的全备和增备的过程。还有一种备份过程叫流式化，可以理解为不用备份到本地磁盘，这样就大大解决了由于备份带来的服务器空间紧张等问题。流式化备份有两种输出格式，一种是基于 tar 的，另一种是基于 xbstream 的。这里主要介绍基于 tar 格式的备份。

9.4.1 非压缩模式的备份

进行流式化备份需要添加一个很重要的参数就是--stream，这里是基于 tar 格式的，所以 stream=tar。然后输出到备份目录/data/backup。

命令如下：

```
innobackupex    --defaults-file=/etc/my.cnf  --no-timestamp  --user  zsbak
--host 192.168.56.102 --password zsbak --stream=tar /tmp >/data/backup/all.tar
```

```
170922 11:10:22 [00] Streaming backup-my.cnf
170922 11:10:22 [00]            ...done
170922 11:10:22 [00] Streaming xtrabackup_info
170922 11:10:22 [00]            ...done
xtrabackup: Transaction log of lsn (55398506) to (55398515) was copied.
170922 11:10:22 completed OK!
```

之后在备份目录下就形成了 all.tar 文件：

```
[root@node3 backup]# ll
total 1075432
-rw-r--r--. 1 root root 1101237248 Sep 22 11:11 all.tar
```

9.4.2 压缩模式的备份

压缩模式的备份只需要在打包模式下加一个管道符号，并在此基础上加一个压缩命令即可。

命令如下：

```
    ./innobackupex  --defaults-file=/etc/my.cnf --no-timestamp --user zsbak
--host 192.168.56.102 --password zsbak --stream=tar /tmp |gzip
- >/data/backup/all-20170922bak.tar.gz
```

```
170922 11:13:27 [00] Streaming backup-my.cnf
170922 11:13:27 [00]            ...done
170922 11:13:27 [00] Streaming xtrabackup_info
170922 11:13:27 [00]            ...done
xtrabackup: Transaction log of lsn (55398506) to (55398515) was copied.
170922 11:13:28 completed OK!
```

备份完成之后，备份目录下多了一个压缩文件：

```
[root@node3 backup]# ll
total 4036
-rw-r--r--. 1 root root 4131921 Sep 22 11:13 all-20170922bak.tar.gz
```

把这个文件解压之后，就是所有的全备文件了。

```
tar -zxvf all-20170922bak.tar.gz
```

```
[root@node3 backup]# ll
total 1052664
-rw-r--r--. 1 root root    4131921 Sep 22 11:13 all-20170922bak.tar.gz
-rw-rw----. 1 root root        431 Sep 22 11:13 backup-my.cnf
-rw-rw----. 1 root root        407 Sep 21 15:30 ib_buffer_pool
-rw-rw----. 1 root root 1073741824 Sep 22 09:33 ibdata1
drwxr-xr-x. 2 root root       4096 Sep 22 11:19 mysql
drwxr-xr-x. 2 root root       4096 Sep 22 11:19 performance_schema
drwxr-xr-x. 2 root root      12288 Sep 22 11:19 sys
-rw-rw----. 1 root root         66 Sep 22 11:13 xtrabackup_binlog_info
-rw-rw----. 1 root root        115 Sep 22 11:13 xtrabackup_checkpoints
-rw-rw----. 1 root root        617 Sep 22 11:13 xtrabackup_info
-rw-rw----. 1 root root       2560 Sep 22 11:13 xtrabackup_logfile
drwxr-xr-x. 2 root root       4096 Sep 22 11:19 zs
```

9.4.3　远程备份

有时需要备份的数据量很大，有可能是几个 T 的量级。这时如果备份到本地磁盘，那么服务器所使用的空间就很紧张了。一般建议本地的磁盘空间不能少于数据库的大小。

使用流式化备份可以把备份文件传到远程备份机上，这样就不会造成由于磁盘空间不足而影响现有数据库的正常运行。

目的：把 192.168.56.102 上的全量备份传递到远程的服务器 192.168.56.101 的/data/backup 上。

基于 SSH 的远程备份操作如下：

```
    ./innobackupex   --defaults-file=/etc/my.cnf --no-timestamp --user zsbak
--host 192.168.56.102 --password zsbak --stream=tar /tmp |gzip |ssh
root@192.168.56.101 "cat - > /data/backup/20170922bak.tar.gz"
```

> 注：192.168.56.101 和 192.168.56.102 两台机器要互信成功，不能输密码。而且先要在 192.168.56.102 上创建好备份目录/data/backup。

在 192.168.56.102 上备份成功：

```
170922 11:56:26 [00] Streaming backup-my.cnf
170922 11:56:26 [00]            ...done
170922 11:56:26 [00] Streaming xtrabackup_info
170922 11:56:26 [00]            ...done
xtrabackup: Transaction log of lsn (55398506) to (55398515) was copied.
170922 11:56:26 completed OK!
```

在 192.168.56.101 上，可以看到压缩好的备份文件已传递成功：

```
[root@node2 ~]# cd /data/backup/
[root@node2 backup]# ll
total 4036
-rw-r--r--. 1 root root 4131925 Sep 22 11:56 20170922bak.tar.gz
```

9.5　表空间传输

从 MySQL 5.6 版本开始，引入了传输表空间这个功能，可以把一张表从一个数据库移到另一个数据库或者另一台机器上。在做数据迁移时，非常方便，尤其是针对一张数据量很大的表来说。相比 mysqldump 的方式，表空间传输要快很多，而且更加灵活。

当然如果想使用表空间传输的功能，也必须满足以下几个条件：

（1）MySQL 版本必须是 5.6 及以上的版本。

（2）使用独立表空间方式，现在版本默认开启 innodb_file_per_table。

（3）源库与目标库之间的 page size 必须一致。

（4）当表做导出操作时，该表只能进行只读操作。

实验演练过程

目的：把 zs 库下面的 tt 这张表的数据传输到 tzy 库下 tt 表中，zs 库 tt 表中的数据如下。

```
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)
```

首先创建 tzy 这个库，并在 tzy 库下创建一张与 zs 库下 tt 表结构一样的 tt 表：

```
root@db 09:45:  [zs]> use tzy;
Database changed
root@db 09:45:  [tzy]>  CREATE TABLE `tt` (
    ->   `id` int(11) NOT NULL AUTO_INCREMENT,
    ->   `name` varchar(20) NOT NULL,
    ->   `score` tinyint(1) NOT NULL DEFAULT '0',
    ->   PRIMARY KEY (`id`)
    -> ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
Query OK, 0 rows affected (0.37 sec)

root@db 09:46:  [tzy]> desc tt;
+-------+------------+------+-----+---------+----------------+
| Field | Type       | Null | Key | Default | Extra          |
+-------+------------+------+-----+---------+----------------+
| id    | int(11)    | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20)| NO   |     | NULL    |                |
| score | tinyint(1) | NO   |     | 0       |                |
+-------+------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

也能看到在数据目录下形成了 tt 表的数据信息：

```
[root@node3 mysql]# cd tzy
[root@node3 tzy]# ll
total 112
-rw-r-----. 1 mysql mysql    67 Sep 24 09:45 db.opt
-rw-r-----. 1 mysql mysql  8618 Sep 24 09:46 tt.frm
-rw-r-----. 1 mysql mysql 98304 Sep 24 09:46 tt.ibd
```

接下来需要卸载 tzy 库下 tt 表的表空间：

```
root@db 09:58:  [tzy]> alter table tt discard tablespace;
Query OK, 0 rows affected (0.04 sec)
```

```
[root@node3 tzy]# ll
total 16
-rw-r-----. 1 mysql mysql   67 Sep 24 09:45 db.opt
-rw-r-----. 1 mysql mysql 8618 Sep 24 09:46 tt.frm
```

结果发现数据目录 tzy 库下 tt.ibd 文件被删除了。

> 注：卸载表空间时要注意，一定要在目标库 tzy 下操作，而且是在数据库命令行下，执行 alter table table_name discard tablespace 命令完成的。绝对不是在系统层面进行 rm 操作，这里一定要切记。

然后在 zs 库下执行表空间导出的操作：

```
root@db 10:07:  [zs]> flush table tt for export;
Query OK, 0 rows affected (0.00 sec)
```

导出完成之后，会发现 zs 库下多了一个 tt.cfg 的文件：

```
[root@node3 zs]# ll
total 13656
-rw-r-----. 1 mysql mysql       67 Sep 21 15:41 db.opt
-rw-r-----. 1 mysql mysql     8622 Sep 21 15:41 test.frm
-rw-r-----. 1 mysql mysql 13631488 Sep 21 15:29 test.ibd
-rw-r-----. 1 mysql mysql      434 Sep 24 10:08 tt.cfg
-rw-r-----. 1 mysql mysql     8618 Sep 21 15:41 tt.frm
-rw-r-----. 1 mysql mysql    98304 Sep 21 15:41 tt.ibd
-rw-r-----. 1 mysql mysql     8590 Sep 21 15:41 ttt.frm
-rw-r-----. 1 mysql mysql    98304 Sep 21 15:29 ttt.ibd
-rw-r-----. 1 mysql mysql     8556 Sep 21 15:41 yy.frm
-rw-r-----. 1 mysql mysql    98304 Sep 21 15:29 yy.ibd
```

将 zs 库下的 tt.ibd 文件和 tt.cfg 文件复制到 tzy 库下，并修改 mysql 权限：

```
[root@node3 zs]# cp tt.{ibd,cfg} /data/mysql/tzy/
```

```
[root@node3 zs]# chown mysql:mysql -R /data/mysql/tzy
[root@node3 zs]# ll /data/mysql/tzy
total 116
-rw-r-----. 1 mysql mysql    67 Sep 24 09:45 db.opt
-rw-r-----. 1 mysql mysql   434 Sep 24 10:13 tt.cfg
-rw-r-----. 1 mysql mysql  8618 Sep 24 09:46 tt.frm
-rw-r-----. 1 mysql mysql 98304 Sep 24 10:13 tt.ibd
```

权限授完之后，因为目前处于只读操作，所有需要执行解锁操作：

```
root@db 10:23:  [zs]> unlock tables;
Query OK, 0 rows affected (0.00 sec)
```

最后在 tzy 库下，执行表空间导入操作：

```
root@db 10:20:  [tzy]> alter table tt import tablespace;
Query OK, 0 rows affected, 1 warning (0.06 sec)

root@db 10:20:  [tzy]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)
```

最终发现已经成功把 zs 库下 tt 的数据传输到 tzy 库下的 tt 表中。

9.6 利用 binlog2sql 进行闪回

在体系结构中已经介绍过 binlog 文件的作用、格式和查看方法。比较好用的工具就是 mysqlbinlog，使用 mysqlbinlog 可以进行基于位置或者时间点的数据恢复操作，可见 binlog 在备份恢复中的作用也是非常重要的。

正所谓“常在河边走，哪有不湿鞋”。在生产环境中经常会遇到误删除、改错数据的情况，现在介绍一款很方便、又省时的 MySQL 闪回工具 binlog2sql，它可以实现数据的快速回滚，从 binlog 中提取 SQL，并能生成回滚 SQL 语句。binlog 以 event 作为单位记录数据库变更的数据信息，闪回就是可以重现这些变化数据信息之前的操作。也就是说，对于 insert 操作，会生成 delete 语句，反之 delete 操作会生成 insert 语句。对于 update 操作，也会生成相反的 update 语句。这款工具只能使用在 binlog 格式为 row 模式下。

binlog2sql 工具的下载地址：

https://github.com/danfengcao/binlog2sql。

环境准备安装各种依赖的工具包——python-pip、PyMySQL、python-mysql-replication、wheel argparse。

解压 binlog2sql 软件：

```
unzip  binlog2sql-master.zip
cd binlog2sql-master
pip install -r requirements.txt
```

可通过 python binlog2sql.py –help 命令查看参数的使用方式。

- -B, --flashback：生成回滚语句。

- --start-file：需要解析的 binlog 文件。
- --start-position：解析 binlog 的起始位置。
- --stop-position：解析 binlog 的结束位置。
- --start-datetime：从哪个时间点的 binlog 开始解析，格式必须为 datetime。
- --stop-datetime：到哪个时间点的 binlog 停止解析，格式必须为 datetime。
- -d, --databases：只输出目标 db 的 SQL。
- -t, --tables：只输出目标 tables 的 SQL。

实验展示过程

首先删除 zs 库下 tt 表中的数据：

```
root@db 15:38:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)

root@db 15:38:  [zs]> delete from tt;
Query OK, 5 rows affected (0.01 sec)

root@db 15:38:  [zs]> select * from tt;
Empty set (0.00 sec)
```

需要创建一个闪回用户：

```
root@db 15:38:  [zs]> GRANT SELECT,REPLICATION SLAVE,REPLICATION CLIENT ON *.* TO 'zs_test'@'%'
 identified by 'zs_test';
Query OK, 0 rows affected, 1 warning (0.04 sec)

root@db 15:38:  [zs]> flush privileges;
Query OK, 0 rows affected (0.00 sec)

root@db 15:39:  [zs]>
```

然后就可以进行恢复操作了，首先确认当前的 binlog 文件是什么：

```
root@db 15:39:  [zs]> show master status;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-binlog.000001 |     1494 |              |                  | 088f82f9-9ea1-11e7-8
0027cd683a:1-5 |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

可以看到当前 binlog 文件是 mysql-binlog.000001。

再定位误操作 SQL 的 binlog 位置，预估一下大致的误操作时间，时间范围应该在下午 3 点 20 分到 3 点 40 分之间，最后再根据时间过滤数据。

命令如下：

```
python binlog2sql.py -h192.168.56.102 -P3306 -uzs_test -pzs_test -dzs -ttt
--start-file='mysql-binlog.000001'   --start-datetime='2017-09-24   15:20:00'
--stop-datetime='2017-09-24 15:40:00'
```

```
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (60, 1, 'aa'); #start 503 end 700 tim
e 2017-09-24 15:38:01
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (70, 2, 'bb'); #start 503 end 700 tim
e 2017-09-24 15:38:01
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (80, 3, 'cc'); #start 503 end 700 tim
e 2017-09-24 15:38:01
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (90, 4, 'dd'); #start 503 end 700 tim
e 2017-09-24 15:38:01
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (100, 5, 'ee'); #start 503 end 700 ti
me 2017-09-24 15:38:01
DELETE FROM `zs`.`tt` WHERE `score`=60 AND `id`=1 AND `name`='aa' LIMIT 1; #start 796 end
 993 time 2017-09-24 15:38:12
DELETE FROM `zs`.`tt` WHERE `score`=70 AND `id`=2 AND `name`='bb' LIMIT 1; #start 796 end
 993 time 2017-09-24 15:38:12
DELETE FROM `zs`.`tt` WHERE `score`=80 AND `id`=3 AND `name`='cc' LIMIT 1; #start 796 end
 993 time 2017-09-24 15:38:12
DELETE FROM `zs`.`tt` WHERE `score`=90 AND `id`=4 AND `name`='dd' LIMIT 1; #start 796 end
 993 time 2017-09-24 15:38:12
DELETE FROM `zs`.`tt` WHERE `score`=100 AND `id`=5 AND `name`='ee' LIMIT 1; #start 796 en
d 993 time 2017-09-24 15:38:12
```

从解析结果中我们了解到，误操作 SQL 的位置是在 796 到 993 之间。这样就可以进一步过滤，使用 flashback 模式生成回滚 SQL。

命令如下：

```
python binlog2sql.py -h192.168.56.102 -P3306 -uzs_test -pzs_test -dzs -ttt
--start-file='mysql-binlog.000001'  -start-position=796 --stop-position=993
-B >tt_rollback.sql
cat tt_rollback.sql
```

```
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (100, 5, 'ee'); #start 796 end 993 time 201
7-09-24 15:38:12
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (90, 4, 'dd'); #start 796 end 993 time 2017
-09-24 15:38:12
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (80, 3, 'cc'); #start 796 end 993 time 2017
-09-24 15:38:12
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (70, 2, 'bb'); #start 796 end 993 time 2017
-09-24 15:38:12
INSERT INTO `zs`.`tt`(`score`, `id`, `name`) VALUES (60, 1, 'aa'); #start 796 end 993 time 2017
-09-24 15:38:12
```

最后应用回滚 SQL 语句，完成恢复操作：

```
root@db 16:08:  [zs]> source /root/binlog2sql-master/binlog2sql/tt_rollback.sql;
Query OK, 1 row affected (0.35 sec)

Query OK, 1 row affected (0.04 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.00 sec)

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

```
root@db 16:08:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)
```

zs 库下 tt 表中的数据已经恢复成功。

binlog2sql 的注意事项：

（1）要保证 MySQL 服务是开启的，离线无法进行分析 binlog。

（2）binlog_format 必须是 row 格式。

（3）DDL 语句无法做到闪回，只能解析 DML 语句。

9.7 binlog server

binlog 在备份中起着至关重要的作用，备份 binlog 文件时，只能先在本地备份，然后才能传送到远程服务器上。从 MySQL 5.6 版本以后，可以利用 mysqlbinlog 命令把远程机器的日志备份到本地目录，这样就更加方便快捷地实现 binlog 日志的安全备份。

环境介绍：192.168.56.100 是备份服务器，192.168.56.101 是正在运行的 MySQL 数据库。

重点参数介绍。

- -R --read-from-remote-server：代表从远程 MySQL 服务器上读取 binlog。
- -raw：以 binlog 格式存储日志，方便后期使用。
- --stop-never：连接到远程的 MySQL 服务器上读取日志，直到远程的服务关闭后才会退出，或是被 kill 掉。

- mysql-bin.***：代表从那个日志开始备份。
- --stop-never-slave-server-id mysqlbinlog：相当于从库拉取主库的日志，所以需要 server-id 来做一个唯一的标识。

操作如下：

先在 56.100 备份服务器上创建一个 binlog 的备份目录。

```
mkdir  -p /data/binbak
```

再在 192.168.56.100 备份服务器上执行远程复制 56.101 上的 binlog 的命令，从 mysql-binlog.000008 开始备份。

```
/usr/local/mysql/bin/mysqlbinlog     --raw     --read-from-remote-server
--stop-never  --host=192.168.56.101  --port=3306  --user=zs  --password=123456
mysql-binlog.000008
```

最后在 56.100 上查看备份的结果，发现 binlog 都已经备份过来了：

```
[root@node1 binbak]# ll
total 32
-rw-r-----. 1 root root   451 Oct  8 10:37 mysql-binlog.000008
-rw-r-----  1 root root 21874 Oct  8 10:37 mysql-binlog.000009
-rw-r-----. 1 root root   123 Oct  8 10:37 mysql-binlog.000010
```

9.8 总结

本章讲解了 MySQL 数据库中不同的备份方式——热备和冷备。热备中包含逻辑备份、裸文件备份。还有根据内容量划分的全备和增备。通过 binlog 在备份中的作用，使用 binlog2sql 完成日志分析和闪回操作，binlog server 的应用以及传输表空间等操作。生产环境中，我们会根据实际情况选择合适的备份工具。而备份策略的制定也可以根据真实的数据量，考虑是否一天一全备，或者一周一全备、一天一增备。总之，维护数据的安全性和保证公司业务的稳定性是 DBA 最基本的工作，踏踏实实地做好本职工作，把基础知识打牢，才能不断地提高自己。

第 3 部分　荣耀黄金篇

黄金篇主要介绍 MySQL 的主从复制。逐渐从一个“菜鸟”，通过一步步“打怪升级”，不断修炼自己的能力。

第 10 章

主从复制概述

MySQL 主从复制也可以称为 MySQL 主从同步，它是构建数据库高可用集群架构的基础。它通过将一台主机的数据复制到其他一台或者多台主机上，并重新应用日志（relay log）中的 SQL 语句来实现复制功能。MySQL 支持单向、双向、链式级联、异步复制，5.5 版本之后加入的半同步复制，5.6 版本之后的 GTID 复制，MySQL 5.7 的多源复制、并行复制、loss-less 复制。复制过程中一台服务器充当主库（master），而一个或者多个服务器充当从库（slave）。

10.1 常见的几种主从架构模式图

单向主从模式如图 10-1 所示。

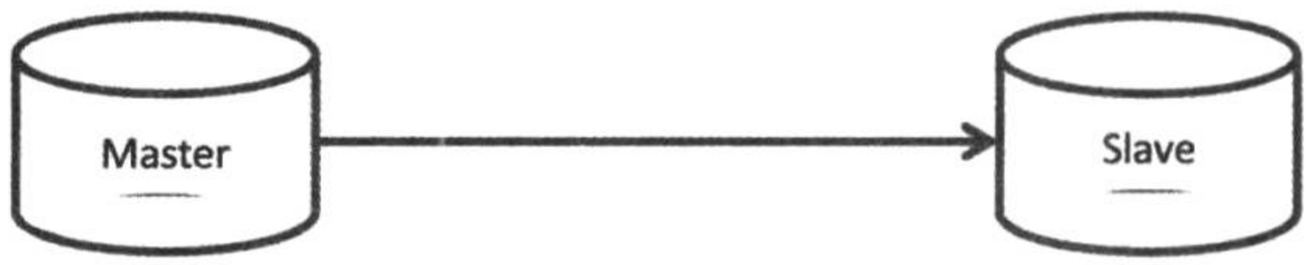


图 10-1 单向主从模式

双向主从模式如图 10-2 所示。

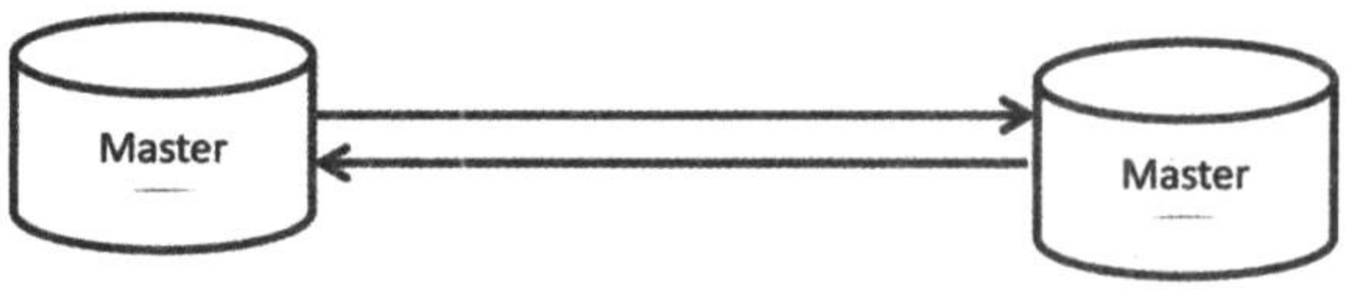


图 10-2 双向主从模式

级联主从模式如图 10-3 所示。

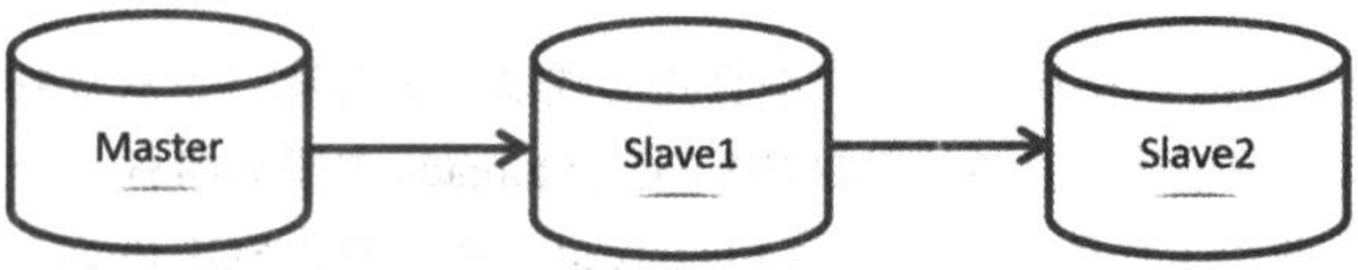


图 10-3　级联主从模式

一主多从模式如图 10-4 所示。

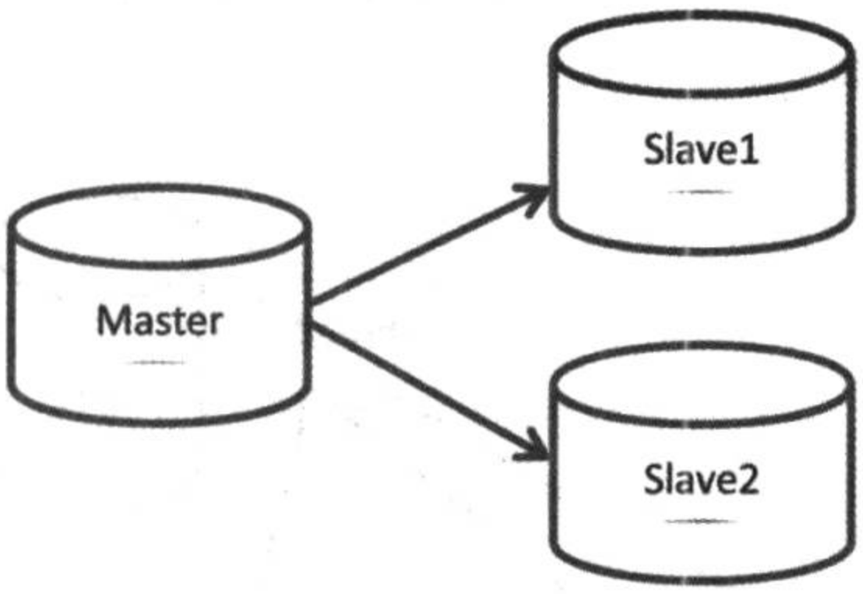


图 10-4　一主多从模式

多主一从模式（MySQL5.7 版本之后才支持）如图 10-5 所示。

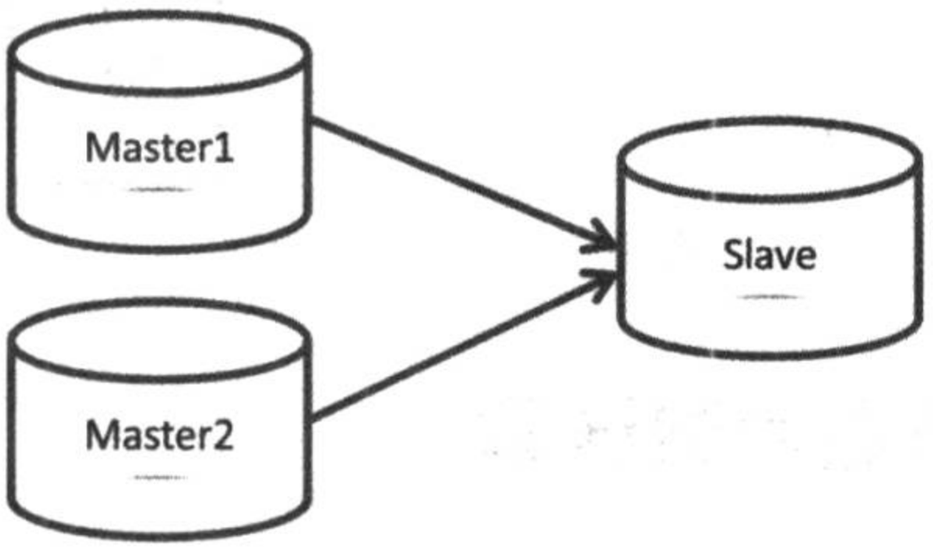


图 10-5　多主一从模式

10.2　主从复制功能

利用 MySQL 主从复制的功能，我们可以实时灾备，让从库随时接管有故障的主库。还可以让从库分担主库的读压力，做读写分离，提供查询服务。并且可以让从库做一些特殊 SQL 的统计任务。最后也可以利用从库做备份来减少对公司现有业务的影响，以及利用从库来完成 MySQL 平滑的版本升级操作。

10.3 主从复制原理

我们先不着急学习搭建复制的过程，在真正实战前，先弄清楚主从复制的原理，便于我们日后排查复制故障。首先介绍主从复制过程中工作的 thread。主服务器有一个工作线程 I/O dump thread，从服务器有两个工作线程，一个是 I/O thread，另一个是 SQL thread。

主库把外界接收的 SQL 请求记录到自己的 binlog 日志中，从库的 I/O thread 去请求主库的 binlog 日志，并将得到的 binlog 日志写到自己的 Relay log（中继日志）文件中。然后在从库上重做应用中继日志中的 SQL 语句。主库通过 I/O dump thread 给从库 I/O thread 传送 binlog 日志。

主从复制原理如图 10-6 所示。

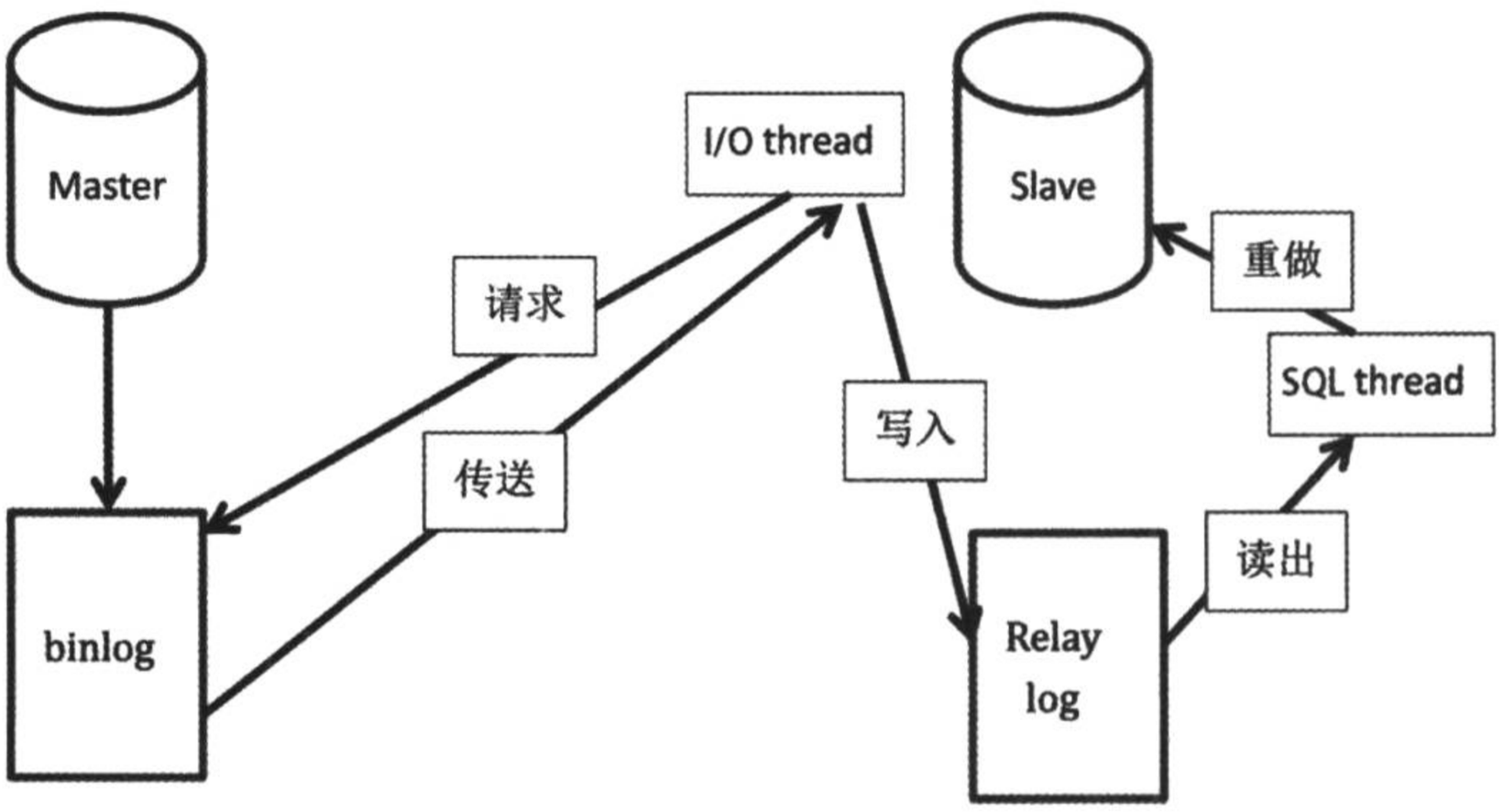


图 10-6 主从复制原理

10.4 复制中的重点参数详解

在第一部分已经介绍了很多关于 MySQL 数据库的参数及其设置使用规范，请参考书中的第 4 章的内容。

本节介绍在主从复制过程中需要考虑的重要参数。

- log-bin：搭建主从复制，必须开启二进制日志。
- server-id：MySQL 在同一组主从结构中的唯一标识（主从服务器上该参数不能一致）。
- server-uuid：从 MySQL 5.6 开始有了这个参数，在数据库启动过程中自动生成，每台机器的 server-uuid 是不一样的。uuid 存放在数据目录的 auto.cnf 文件下。
- read only：设置从库为只读状态，避免在从库上进行写操作。注：对超管权限（super）

账号没有效果。但 MySQL 5.7 之后新增了一个 super_read_only 参数，开启该参数，连超管都没有权限进行写入操作。

- binlog_format：二进制日志的格式，必须使用 row 模式。
- log_slave_updates：作用是将从 master 服务器上获取数据变更的信息记录到从服务器的二进制日志文件中。
- binlog_error_action：该参数用来控制当不能写 binlog 文件时，MySQL server 将会怎样。该参数是 MySQL 5.7 之后新增的，有 ABORT_SERVER 和 IGNORE_ERROR 两个值。ABORT_SERVER 代表会使 MySQL Server 在写 binlog 遇到磁盘满或者文件系统不可写入时退出；而 IGNORE_ERROR 则代表如果遇到 binlog 无法写入的情况，MySQL Server 会在错误日志中记录错误，并且还会强制关闭 binlog 功能。这样就会影响从库获取主库上 binlog 的过程，从而导致主从数据不一致。MySQL 5.7.7 之后默认使用 binlog_error_action=ABORT_SERVER。
- binlog-do-db：使用该参数可选择性复制数据库（在主库上使用），如 binlog-do-db=zs，意味着除了 zs 库，其他库都不复制。但建议大家不要使用该参数，尽量保证复制的过滤规则不在主库上面添加。
- binlog-ignore-db：该参数就是忽略某个库的复制。如 binlog-ignore-db=zs，意味着除了 zs 库，其他库都复制。
- gtid_mode：决定 gtid 模式是否开启。使用 gtid 模式，设置 gtid_mode=on。
- enforce-gtid-consistency：使用 GTID 复制模式时，要开启该参数，用来保证 GTID 的一致性。设置 enforce-gtid-consistency=on。
- gtid_next：该参数是 session 级别的变量，下一个 gtid。默认是 AUTOMATIC。
- gtid_purged：丢弃掉的 GTID。
- relay log：记录从库的 I/O thread 从主库读取而来的 binlog 内容。
- replicate_do_table：只复制指定的表，在从库上使用。
- replicate_ignore_table：不复制指定的表，在从库上使用。
- replicate_do_db：只复制指定的库，在从库上使用。
- replicate_ignore_db：不复制指定的库，在从库上使用。
- replicate-wild-do-table：使用通配符复制指定的表，如复制 zs 表下 tt 开头的表，--replicate-wild-do-table=zs.tt%。
- replicate-wild-ignore-table：使用通配符不复制指定的表。
- master_info_repository：把 master.info（主从状态，配置信息）记录下来，默认记录到

file 里，建议使用表记录。

```
master_info_repository=table
```

- relay_log_info_repository sql thread：应用二进制日志中的内容，并将 binlog 应用到的位置记录到 relay.info，默认记录到 file 里，建议使用表记录。

```
relay_log_info_repository=table
```

- relay_log_recovery：为了让从库是 crash safe 的，必须要设置 relay_log_recovery=1。该参数的含义是：当从库发生崩溃或者重启时，它会把那些未执行完的中继日志删除，并会向主库重新获取 binlog，再次生成 relay log 来完成中继日志的恢复。建议在从库上开启 relay_log_recovery 参数，默认情况下是关闭的。
- relay_log_purge：清除已经执行过的 relay log。建议在从库开启该参数。
- slave_net_timeout：该参数是设置在多少秒没收到主库传来的 binlog 之后，从库认为是网络超时，从库的 I/O thread 会重新连接主库。该值从 MySQL 5.7.7 开始的默认值为 60s。
- slave_parallel_type 该参数是从 MySQL 5.7.2 引入的，有两个值，一个是 DATABASE，另一个是 LOGICAL_CLOCK。在 MySQL 5.7 中引入了基于组提交的并行复制。通过设置参数 slave_parallel_workers>0 并且 slave_parallel_type='LOGICAL_CLOCK' 实现。
- slave_parallel_workers：设置多个线程来并发执行 relay log 中主库提交的事务，最大值为 1024。

第11章 复制原理及实战演练

MySQL 主从复制方式有默认的异步复制，5.5 版本之后的半同步复制，5.6 版本新增的 GTID 复制，包括 5.7 版本的多源复制，基于组提交的并行复制和增强半同步复制功能。本章介绍复制方式的原理、搭建过程、复制数据校验、复制延迟以及故障问题的处理，让大家更好地掌握 MySQL 复制功能，为后期构建高性能、高可用的数据库集群架构打下基础。

11.1 异步复制

异步复制是 MySQL 默认的复制方式，其原理很简单，就是在主库写入 binlog 日志后即可成功返回客户端，无须等待 binlog 日志传递给从库的过程。但这样一旦主库发生宕机，就有可能出现丢失数据的情况。

下面讲解 MySQL 5.6 版本的异步复制在线搭建过程，这里是非 GTID 的模式，我们基于 binlog 和 position 方式来搭建一主一从的架构。先介绍一下环境：

- 192.168.56.132（作为 master）；
- 192.168.56.133（作为 slave）。

搭建主从的几个必要条件：

（1）主库的 server-id=1323306，从库的 server-id=1333306，保证两者不一致。

（2）主库开启 binlog 功能。

> 注：建议从库也开启 binlog，并且开启 log_slave_updates 参数，让从库也写 binlog，方便后期扩展架构。

（3）为了后期不出现数据不一致的情况，保证 binlog 格式为 row 模式来实施搭建过程，分为在主库上的操作和在从库上操作。

在主库（192.168.56.132）上的操作如下：

（1）创建一个主从复制的账号。

```
mysql> create user 'bak'@'192.168.56.%' identified by 'bak123';
Query OK, 0 rows affected (0.00 sec)

mysql> grant replication slave on *.* to 'bak'@'192.168.56.%';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

（2）初始化数据，让从库与主库在某一 position 位置时达到同步。这里就需要先从主库中导出数据，用到了之前在备份恢复章节所学的 mysqldump，或者使用 XtraBackup 做数据的导出操作。

命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction -uroot -proot123
--master-data=2 -A > all.sql
```

> 注：必须加参数--master-data=2,让备份出来的文件中记录备份这一时刻的binlog文件与 position 号，为搭建主从环境做准备。查看 all.sql 备份文件，已记录当前 binlog 文件和 position 号。

```
--
-- Position to start replication or point-in-time recovery from
--

-- CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.000019', MASTER_LOG_POS=1257;
```

（3）把备份出来的 all.sql 文件传递到从库服务器（192.168.56.133）上。

```
[root@node3 ~]# scp all.sql 192.168.56.133:/root/
root@192.168.56.133's password:
all.sql                                       100%   15MB  14.7MB/s   00:01
```

到此，主库上的操作已结束。

在从库（192.168.56.133）上操作如下：

（1）恢复从主库传递过来的数据。

命令如下：

```
mysql -uroot -proot123 < all.sql
```

（2）在数据库命令行执行配置主从命令。

```
CHANGE MASTER TO
MASTER_HOST='192.168.56.132',
MASTER_USER='bak',
MASTER_PASSWORD='bak123',
MASTER_PORT=3306,
MASTER_LOG_FILE='mysql-bin.000019',
MASTER_LOG_POS=1257;
```

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.56.132',MASTER_USER='bak',MASTER_PASSWORD='b
ak123',MASTER_LOG_FILE='mysql-bin.000019',MASTER_LOG_POS=1257;
Query OK, 0 rows affected, 2 warnings (0.01 sec)
```

参数详解。

- MASTER_HOST：指向主库的 IP 地址。
- MASTER_USER：之前创建的复制用户。
- MASTER_PASSWORD：复制用户的密码。
- MASTER_PORT：数据库端口号。
- MASTER_LOG_FILE：从备份文件中获取的当前二进制文件。
- MASTER_LOG_POS：从备份文件中获取的 position 号。

（3）执行开始主从复制的命令。

```
start slave;
```

（4）查看主从复制状态。

```
show slave status\G;
```

```
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.132
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000019
          Read_Master_Log_Pos: 1257
               Relay_Log_File: node4-relay-bin.000002
                Relay_Log_Pos: 283
        Relay_Master_Log_File: mysql-bin.000019
             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: 1257
```

当从库的 I/O thread 和 SQL thread 都呈现 Yes 状态，代表主从复制开始工作了。此时在从库上面的所有操作就已经完成了。

show slave status 命令展现出了几个重要的参数，下面对重点参数进行详解。

- Master_Log_File：当前主库的二进制文件名，本例为 mysql-bin.000019。
- Read_Master_Log_Pos：正在读取主库当前二进制日志的 position 位置，本例为 1257。
- Exec_Master_Log_Pos：执行到主库二进制日志中的 position 位置。本例为 1257。

目前 Master_log_file=relay_master_log_file， Read_Master_Log_Pos=Exec_Master_Log_Pos，证明目前没有主从延迟状态。

- Slave_IO_Running：从库上 I/O thread 负责请求和接收主库传递来的 binlog 信息。
- Slave_SQL_Running：从库上 SQL thread 负责应用 relay 中 binlog 的信息。

接下来验证主从是否可以正常同步数据了。

在主库 192.168.56.132 上的 test 库下，往 t 表中插入一条 SQL：

```
[root@node3 ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:51:d3:cc brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.132/24 brd 192.168.56.255 scope global eth0
    inet6 fe80::a00:27ff:fe51:d3cc/64 scope link
       valid_lft forever preferred_lft forever
3: sit0: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
```

```
mysql> desc t;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | YES  | MUL | NULL    |                |
| city  | varchar(10) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql> select * from t;
Empty set (0.00 sec)

mysql> insert into t (name,city) values ('zs','bj');
Query OK, 1 row affected (0.00 sec)
```

在从库上查看 t 表，可以查到主库新插入的数据：

```
[root@node4 ~]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:a0:bd:4b brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.133/24 brd 192.168.56.255 scope global eth0
    inet6 fe80::a00:27ff:fea0:bd4b/64 scope link
       valid_lft forever preferred_lft forever
3: sit0: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
```

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | zs   | bj   |
+----+------+------+
1 row in set (0.00 sec)
```

证明主从复制成功。

主从复制的管理命令。

- show slave status\G：在从库上查看主从复制状态。
- show master status：查看主库的 binlog 文件和 position 位置，以及开启 GTID 模式下记录的 gtid。
- change master to：在从库上配置主从过程。
- start slave：开启主从同步。
- stop slave：关闭主从同步。
- reset slave all：清空从库的所有配置信息。

11.2 主从复制故障处理

这里总结了几种常见的主从复制故障，方便今后在生产环境中再遇到相似问题时，可以轻松解决。对于 DBA 来说，遇到再棘手的问题，我们都不要慌张，想想知识点的原理，理清解决问题的思路，任何问题都能找到答案，咱们得有一种跟报错“死磕到底”的精神！

1. 主从故障之主键冲突，错误代码为 1062

原因：由于误操作，在从库上执行了写入操作，导致再在主库执行相同操作时，由于主键冲突，主从复制状态会报错。所以生产环境中建议在从库上开启 read only，避免在从库执行写入操作。

模拟故障

先在从库服务器 192.168.56.133 上向 test 库下的 t 表插入一条 SQL：

```
mysql> insert into t (name,city) values ('tzy','sh');
Query OK, 1 row affected (0.01 sec)
```

再在主库服务器 192.168.56.132 上向 test 库下的 t 表插入相同的 SQL 语句：

```
mysql> insert into t (name,city) values ('tzy','sh');
Query OK, 1 row affected (0.31 sec)
```

这时在从库上执行 show slave status 查看主从状态就有报错：

```
            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: 1062
                  Last_Error: Could not execute Write_rows event on table test.t; Duplic
ate entry '2' for key 'PRIMARY', Error_code: 1062; handler error HA_ERR_FOUND_DUPP_KEY; t
he event's master log mysql-bin.000019, end_log_pos 2155
```

错误代码是 1062，代表主键冲突。

解决办法：

遇到这样的问题，可以直接通过 percona-toolkit 工具集中的 pt-slave-restart 命令在从库跳过错误。

```
[root@node4 bin]# ./pt-slave-restart -uroot -proot123
2017-09-29T13:56:07 p=...,u=root node4-relay-bin.000002        1012 1062
```

再次查看主从状态，已经恢复正常：

```
mysql> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.132
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000019
          Read_Master_Log_Pos: 2186
               Relay_Log_File: node4-relay-bin.000002
                Relay_Log_Pos: 1212
        Relay_Master_Log_File: mysql-bin.000019
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

2. 主从故障之主库更新数据，从库找不到而报错，错误代码为 1032

上一个错误是主从库都有相同的数据，我们可以直接通过工具跳过主从错误。但如果从库上少数据，我们就不能直接跳过错误了。需要找到缺少的数据，在从库上重新执行一遍。

故障原因：由于误操作，在从库上执行 delete 删除操作，导致主从数据不一致。这时再在主库执行同条数据的更新操作时，由于从库已经没有该数据，SQL 无法在从库实现。

模拟故障

先在从库服务器的 test 库下的 t 表中，执行 delete 删除语句操作：

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | zs   | bj   |
|  2 | tzy  | sh   |
+----+------+------+
2 rows in set (0.01 sec)

mysql> delete from t where name='tzy';
Query OK, 1 row affected (0.32 sec)
```

再在主库服务器的 test 库下的 t 表中，执行相同数据的 update 更新语句操作：

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | zs   | bj   |
|  2 | tzy  | sh   |
+----+------+------+
2 rows in set (0.00 sec)

mysql> update t set name='zyn' where name='tzy';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

这时在从库上执行 show slave status 命令查看主从状态，就会展现主从的报错。

```
            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: 1032
                  Last_Error: Could not execute Update_rows event on table test.t; Can't find record in 't', E
rror_code: 1032; handler error HA_ERR_KEY_NOT_FOUND; the event's master log mysql-bin.000019, end_log_pos 2368
                Skip_Counter: 0
        Exec_Master_Log_Pos: 2186
            Relay_Log_Space: 1598
```

解决方法：

根据报错信息所知道的 binlog 文件和 position 号，在主库上，通过 mysqlbinlog 命令，找到在主库上执行的哪条 SQL 语句导致的主从报错。

命令如下：

```
/usr/local/mysql/bin/mysqlbinlog --no-defaults -v -v --base64-output=
decode-rows /data/mysql/mysql-bin.000019 |grep -A 10 2368 > sql.log
```

```
[root@node3 mysql]# cat sql.log
#170929 14:08:45 server id 1323306  end_log_pos 2368 CRC32 0xc3b565b6   Update_rows: tabl
e id 112 flags: STMT_END_F
### UPDATE `test`.`t`
### WHERE
###   @1=2 /* INT meta=0 nullable=0 is_null=0 */
###   @2='tzy' /* VARSTRING(60) meta=60 nullable=1 is_null=0 */
###   @3='sh' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
### SET
###   @1=2 /* INT meta=0 nullable=0 is_null=0 */
###   @2='zyn' /* VARSTRING(60) meta=60 nullable=1 is_null=0 */
###   @3='sh' /* VARSTRING(30) meta=30 nullable=1 is_null=0 */
# at 2368
#170929 14:08:45 server id 1323306  end_log_pos 2399 CRC32 0x70536fc7   Xid = 535
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

接下来只需把从库上丢失的这条数据补上，然后再执行跳过错误，主从复制功能就恢复正常了。

> 注：如果在生产环境中，从库缺失了多条数据，建议重新搭建主从环境来确保数据的一致性。

```
mysql> select * from t;
+----+------+------+
| id | name | city |
+----+------+------+
|  1 | zs   | bj   |
+----+------+------+
1 row in set (0.01 sec)

mysql> insert into t (name,city) values ('zyn','sh');
Query OK, 1 row affected (0.00 sec)
```

```
[root@node4 bin]# ./pt-slave-restart -uroot -proot123
2017-09-29T14:19:05 p=...,u=root node4-relay-bin.000002        1212 1032
```

```
mysql> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.132
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000019
          Read_Master_Log_Pos: 2399
               Relay_Log_File: node4-relay-bin.000002
                Relay_Log_Pos: 1425
        Relay_Master_Log_File: mysql-bin.000019
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

3. 主从故障之主从 server-id 一致

原因：数据库日常的巡检没有做到位，导致在搭建主从复制的过程中，两台服务器拥有了相同的 server-id。

```
                Last_IO_Errno: 1593
                Last_IO_Error: Fatal error: The slave I/O thread stops because master and
 slave have equal MySQL server ids; these ids must be different for replication to work (
or the --replicate-same-server-id option must be used on slave but this does not always m
ake sense; please check the manual before using it).
```

解决方法：不同机器设置不同的 server-id。

> 注：搭建主从复制时，server-id 是重点的检查项。

4. 主从故障之跨库操作，丢失数据

原因：在主库中设置 binlog-do-db 参数，使用的 binlog 记录格式为 statement 模式，导致在主库上执行跨库操作时，从库没有复制成功，丢失数据。

故障操作描述：

在主库的参数文件中添加 binlog-do-db=zs，代表只复制 zs 这个库，并且主库的 binlog_format

设置为 statement。

在主库上执行：

```
use tzy;
insert into zs.t (name, city) values('zz','gz');
```

就是在 tzy 库下，往 zs 库的 t 表中插入数据。

结果这条语句不会同步到从库，这就是由于使用 binlog-do-db 进行跨库操作，并且在 statement binlog 格式下，数据不能被复制到从库上，导致主从数据的不一致现象。

> 注：如果在 row 格式下进行该操作，就不会出现从库不能复制数据的现象。

解决办法：在主库上尽量避免使用库复制的过滤规则，可以在从库上使用 replicate-do-db 或者 replicate-ignore-db 等参数，最重要的一点就是一定要让 binlog 的格式为 row 模式，可以使复制更加安全。

11.3 半同步复制

MySQL 数据库复制默认的方式是异步复制，但异步复制的不足之处就在于，当主库把 event 写入二进制日志之后，并不知道从库是否已经接收并应用了。在异步模式下的复制，如果主库崩溃，很有可能在主库中已经提交的事务，并没有传到任何一台从库机器上。在高可用集群架构下做主备切换，就会造成新的主库丢失数据的现象。

MySQL 5.5 版本之后引入了半同步复制功能，主从服务器必须同时安装半同步复制插件，才能开启该复制功能。在该功能下，确保从库接收完主库传递过来的 binlog 内容已经写入到自己的 relay log 里面了，才会通知主库上面的等待线程，该操作完毕。如果等待超时，超过 rpl_semi_sync_master_timeout 参数设置的时间，则关闭半同步复制，并自动转换为异步复制模式，直到至少有一台从库通知主库已经接收到 binlog 信息了为止。

半同步复制原理如图 11-1 所示。

半同步复制提升了主从之间数据的一致性，让复制更加安全可靠。在 MySQL 5.7 版本中又增加了 rpl_semi_sync_master_wait_point 参数，用来控制半同步模式下主库在返回给 session 事务成功之前的事务提交方式。

该参数有两个值：

（1）AFTER_COMMIT。

该值是 MySQL 5.6 版本的默认值，含义为：主库将每个事务写入 binlog，并传递给从库，

刷新到中继日志中，同时主库提交事务。之后主库开始等待从库的反馈，只有收到从库的回复之后，master 才将“commit OK”的结果反馈给客户端。

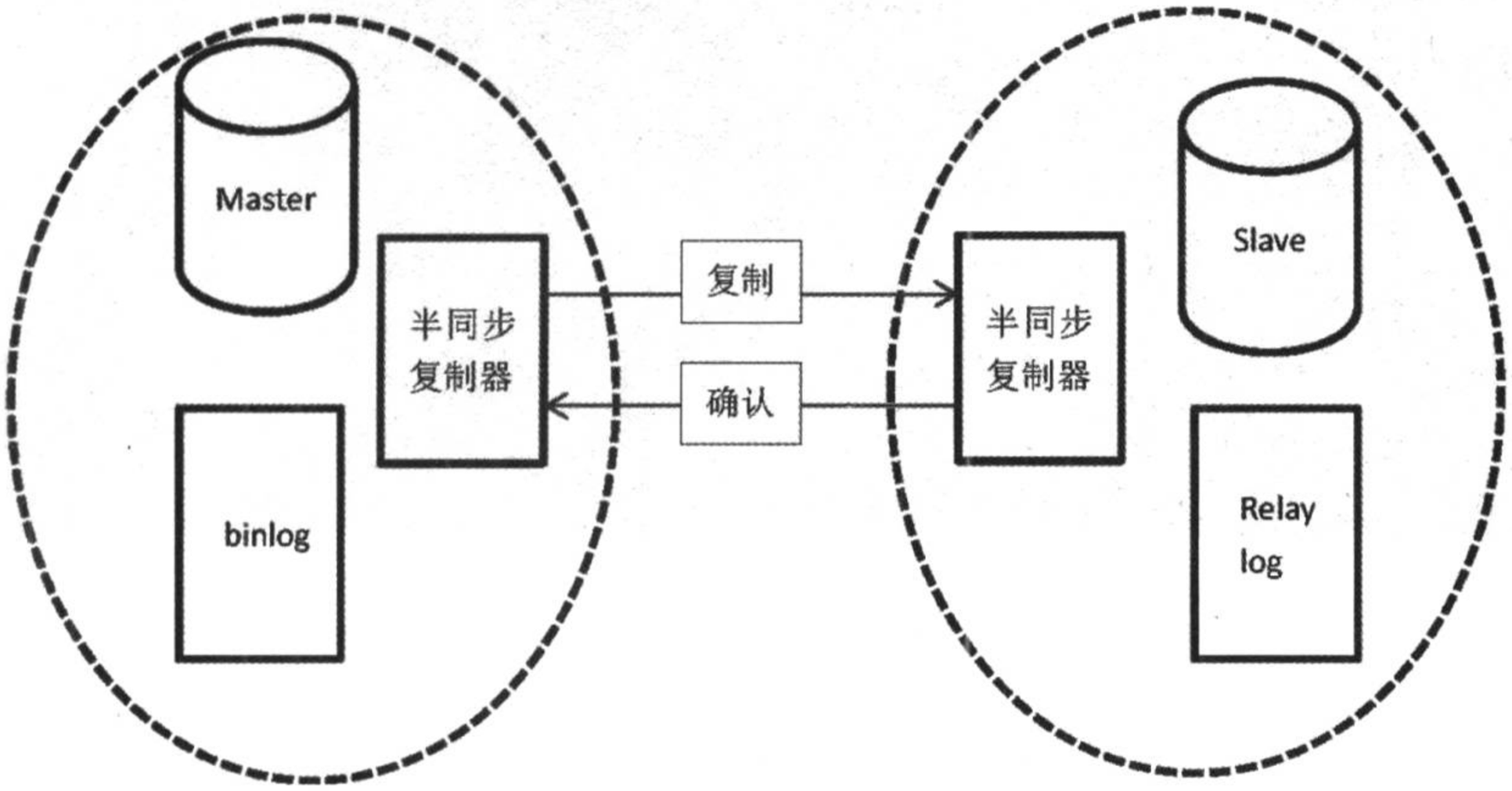


图 11-1　半同步复制原理

（2）AFTER_SYNC。

该值是 MySQL 5.7 版本之后新增的，也是 MySQL 5.7 默认的半同步复制方式。含义为：主库将每个事务写入 binlog，并传递给从库，刷新到中继日志中，主库开始等待从库的反馈，接收到从库的回复之后，再提交事务并且返回“commit OK”结果给客户端。

> 注：可以通过 rpl_semi_sync_master_wait_for_slave_count 参数来控制主库接收多少个从库写事务成功反馈，才返回成功给客户端。生产环境中使用半同步复制方式，当从库出现故障，等待超时的时间又很长，导致主库无法接收从库信息而无法正常写入时，可通过该参数剔除故障从库。

在 after_sync 模式下，即使主库宕机，所有在主库上已经提交的事务都能保证已经同步到从库的中继日志中，不会丢任何数据。

半同步复制的搭建很简单，它基于异步复制的基础上，安装半同步复制插件就可以了。异步复制的过程我们之前已经搭建过。

目前我们需要先在主库中安装半同步复制插件和开启半同步复制功能：

```
install plugin rpl_semi_sync_master soname 'semisync_master.so';
set global  rpl_semi_sync_master_enabled=on;
```

```
root@db 10:30:  [(none)]> install plugin rpl_semi_sync_master soname 'semisync_master.so'
;
Query OK, 0 rows affected (0.07 sec)

root@db 10:31:  [(none)]> set global  rpl_semi_sync_master_enabled=on;
Query OK, 0 rows affected (0.00 sec)

root@db 10:31:  [(none)]> show variables like '%semi%';
+-------------------------------------------+-------+
| Variable_name                             | Value |
+-------------------------------------------+-------+
| rpl_semi_sync_master_enabled              | ON    |
| rpl_semi_sync_master_timeout              | 10000 |
| rpl_semi_sync_master_trace_level          | 32    |
| rpl_semi_sync_master_wait_for_slave_count | 1     |
| rpl_semi_sync_master_wait_no_slave        | ON    |
| rpl_semi_sync_master_wait_point           | AFTER_SYNC |
+-------------------------------------------+-------+
6 rows in set (0.04 sec)
```

> 注：还有一个比较重要的参数就是 rpl_semi_sync_master_timeout，单位是毫秒。它表示如果主库等待从库回复消息的时间超过该值，就自动切换为异步复制模式。建议不要取默认值 10s，该值可以调整得很大，禁止向异步复制切换来保证数据复制的安全性。MySQL 5.7 半同步复制默认的方式就是 after_sync 模式。

可通过 show plugins 确认插件已经加载成功：

```
| rpl_semi_sync_master          | ACTIVE   | REPLICATION        | semisync_master.so | GPL
```

然后再在从库中安装半同步复制插件和开启半同步复制功能：

```
install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
set global  rpl_semi_sync_slave_enabled=on;
```

```
root@db 10:30:  [(none)]> install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
Query OK, 0 rows affected (0.04 sec)

root@db 10:33:  [(none)]> set global  rpl_semi_sync_slave_enabled=on;
Query OK, 0 rows affected (0.00 sec)

root@db 10:33:  [(none)]> show variables like '%semi%';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| rpl_semi_sync_slave_enabled    | ON    |
| rpl_semi_sync_slave_trace_level | 32   |
+--------------------------------+-------+
2 rows in set (0.04 sec)
```

通过 show plugins 确认插件已经加载成功：

```
| rpl_semi_sync_slave         | ACTIVE   | REPLICATION        | semisync_slave.so | GPL
```

> 注：为了以后可以开机自启动半同步复制功能，我们可以把 rpl_semi_sync_slave_enabled=on、rpl_semi_sync_master_enabled=on 两个参数加载到 my.cnf 配置文件中。

由于之前已经建立了成功的异步复制模式，现在只是成功加载半同步复制的插件和开启半同步复制功能，还没有真正实现半同步复制，所以需要最后一步，重启从库的 I/O 线程，激活半同步复制。

```
stop slave io_thread;
start slave io_thread;
```

在主库上查看半同步复制是否正常运行：

```
root@db 10:43:  [(none)]> show global 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                | 0     |
+--------------------------------------------+-------+
14 rows in set (0.01 sec)
```

Rpl_semi_sync_master_clients 参数代表已经有一个从库连接到了主库，并且是半同步复制方式。

Rpl_semi_sync_master_status 参数是 ON（开启）的状态，代表已经是半同步复制模式了。

> 注：还有几个参数需要多加注意。

- Rpl_semi_sync_master_no_tx：代表没有成功接收 slave 提交的次数。
- Rpl_semi_sync_master_yes_tx：代表成功接收 slave 事务回复的次数。

在从库上查看半同步复制状态：

```
root@db 10:55:  [(none)]> show global status like '%semi%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Rpl_semi_sync_slave_status | ON    |
+----------------------------+-------+
1 row in set (0.00 sec)
```

Rpl_semi_sync_slave_status 参数等于 on 代表从库也开启了半同步复制模式。至此 MySQL 半同步复制搭建成功。

11.4 半同步复制和异步复制模式的切换

半同步复制的原理是从库的 I/O thread 接收完主库的 binlog，并把它写入 relay 中后，会给主库一个回馈。但如果主库等待从库的回复时间超过 rpl_semi_sync_master_timeout 参数设置的时间，也会自动切换为异步复制方式。目前测试过程中等待时间为 10s。

```
root@db 11:04:  [(none)]>  show variables like '%rpl_semi_sync_master_timeout%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| rpl_semi_sync_master_timeout  | 10000 |
+-------------------------------+-------+
1 row in set (0.00 sec)
```

首先需要先把从库的 I/O thread 关掉：

```
stop slave io_thread;
```

查看从库的半同步状态已经为 OFF，关闭了。

```
root@db 11:10:  [zs]> show global status like '%semi%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Rpl_semi_sync_slave_status | OFF   |
+----------------------------+-------+
1 row in set (0.00 sec)
```

这时查看主库的半同步复制状态，还是开启的，并且还有连接的从库。

```
root@db 11:08:  [zs]> show global 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            | 3     |
| 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     | 1008  |
| Rpl_semi_sync_master_tx_wait_time         | 3024  |
| Rpl_semi_sync_master_tx_waits             | 3     |
| Rpl_semi_sync_master_wait_pos_backtraverse| 0     |
| Rpl_semi_sync_master_wait_sessions        | 0     |
| Rpl_semi_sync_master_yes_tx               | 3     |
+-------------------------------------------+-------+
```

然后在主库中往 zs 库下 tt 表中插入一条 SQL 语句：

```
root@db 11:10:  [zs]> insert into tt (name,score) values ('ff','99');
Query OK, 1 row affected (10.00 sec)
```

这条 SQL 语句的执行时间为 10s，很慢。主库一直在等待从库的回复，直到超过默认的等待时间 10s。

这时再查看主库的半同步复制状态，发现主库已经没有可连接的从库了，而且主库的半同步复制状态也已经是 OFF 了。证明已经完全从半同步复制自动切换为异步复制模式了。

```
root@db 11:11:  [zs]> show global status like '%semi%';
+-------------------------------------------+-------+
| Variable_name                             | Value |
+-------------------------------------------+-------+
| Rpl_semi_sync_master_clients              | 0     |
| Rpl_semi_sync_master_net_avg_wait_time    | 0     |
| Rpl_semi_sync_master_net_wait_time        | 0     |
| Rpl_semi_sync_master_net_waits            | 3     |
| Rpl_semi_sync_master_no_times             | 1     |
| Rpl_semi_sync_master_no_tx                | 1     |
| Rpl_semi_sync_master_status               | OFF   |
| Rpl_semi_sync_master_timefunc_failures    | 0     |
| Rpl_semi_sync_master_tx_avg_wait_time     | 1008  |
| Rpl_semi_sync_master_tx_wait_time         | 3024  |
| Rpl_semi_sync_master_tx_waits             | 3     |
| Rpl_semi_sync_master_wait_pos_backtraverse| 0     |
| Rpl_semi_sync_master_wait_sessions        | 0     |
| Rpl_semi_sync_master_yes_tx               | 3     |
+-------------------------------------------+-------+
14 rows in set (0.01 sec)
```

> 注：在生产环境中不建议将半同步复制切换到异步复制模式。

因为这样对数据的安全性没有保证。所以一些公司将 rpl_semi_sync_master_timeout 参数的值设置得很大。

如果再想从异步复制模式切换为半同步复制，只需重新开启从库的 I/O thread。

命令：start slave io_thread。

从库半同步复制开启：

```
root@db 11:10:  [zs]> start slave io_thread;
Query OK, 0 rows affected (0.00 sec)

root@db 11:19:  [zs]> show global status like '%semi%';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| Rpl_semi_sync_slave_status  | ON    |
+-----------------------------+-------+
1 row in set (0.00 sec)
```

主库半同步复制开启成功。

```
root@db 11:14:  [zs]> show global 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             | 4     |
| Rpl_semi_sync_master_no_times              | 1     |
| Rpl_semi_sync_master_no_tx                 | 1     |
| Rpl_semi_sync_master_status                | ON    |
| Rpl_semi_sync_master_timefunc_failures     | 0     |
| Rpl_semi_sync_master_tx_avg_wait_time      | 1008  |
| Rpl_semi_sync_master_tx_wait_time          | 3024  |
| Rpl_semi_sync_master_tx_waits              | 3     |
| Rpl_semi_sync_master_wait_pos_backtraverse | 0     |
| Rpl_semi_sync_master_wait_sessions         | 0     |
| Rpl_semi_sync_master_yes_tx                | 3     |
+--------------------------------------------+-------+
14 rows in set (0.00 sec)
```

11.5 GTID 复制

11.5.1 GTID 原理介绍

GTID 又叫全局事务 ID（Global Transaction ID），是一个已提交事务的编号，并且是一个全局唯一的编号。MySQL 5.6 版本之后在主从复制类型上新增了 GTID 复制。

GTID 是由 server_uuid 和事务 id 组成的，即 GTID = server_uuid:transaction_id。 server_uuid

是在数据库启动过程中自动生成的，每台机器的 server-uuid 不一样。uuid 存放在数据目录的 auto.cnf 文件下。而 transaction_id 就是事务提交时由系统顺序分配的一个不会重复的序列号。

11.5.2　GTID 存在的价值

（1）GTID 使用 master_auto_position=1 代替了基于 binlog 和 position 号的主从复制搭建方式，更便于主从复制的搭建。

（2）GTID 可以知道事务在最开始是在哪个实例上提交的。

（3）GTID 方便实现主从之间的 failover，再也不用不断地去找 position 和 binlog 了。

11.5.3　主从复制中 GTID 的管理与维护

GTID 带来最方便的一点就是主从复制的搭建过程了。它跟异步复制、半同步复制类似，只不过不再利用传统复制模式的 binlog 文件和 position 号了，而是在从库“change master to”时使用 master_auto_position=1 的方式进行搭建，这就让操作变得更加方便和可靠。

GTID 搭建过程中的注意事项

主从库需要设置的参数如下。

主库配置：

```
gtid_mode=on
enforce_gtid_consistency=on
log_bin=on
```

server-id 不能与从库一样。

```
binlog_format=row
```

从库配置：

```
gtid_mode=on
enforce_gtid_consistency=on
log_slave_updates=1
```

虽然在 MySQL 5.7 版本之后可以关闭掉该参数，使用 gtid_executed 这张表。但还是建议在从库中开启。

server-id 不能与主库一样。

配置好参数之后，主库也创建了复制账号，如果是新搭建的主从环境，就可以直接在从库执行 change master to 语句了。如果是已经运行了一段期间的主库，还需要利用备份方式从主库“dump”出数据到从库中，先完成基于某个点的 GTID 复制，然后从库从那个点之后再开始追主库。利用 mysqldump 备份，备份后的文件中会有 SET @@GLOBAL.GTID_PURGED= ***，利用 xtrabackup 工具备份，备份后的文件中会直接记录需要跳过的 GTID。启动复制之后，从库会直接跳过已经执行过 GTID 的范围，直接从主库获取新的 GTID 信息。

在主库执行 show master status 命令，通过 Executed_Gtid_Set 来查看执行过的 GTID。

```
root@db 15:51:  [(none)]> show master status;
+-------------------------+----------+--------------+------------------+-------------------------------------------+
| File                    | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set                         |
+-------------------------+----------+--------------+------------------+-------------------------------------------+
| mysql-binlog.000005     |      194 |              |                  | 088f82f9-9ea1-11e7-8bb2-080027cd683a:1-100025 |
+-------------------------+----------+--------------+------------------+-------------------------------------------+
1 row in set (0.00 sec)
```

在 MySQL 5.7 版本之后，gtid_executed 这个值持久化了。在 MySQL 库下新增了一张表 gtid_executed：

```
root@db 16:07:  [mysql]> desc gtid_executed;
+----------------+-----------+------+-----+---------+-------+
| Field          | Type      | Null | Key | Default | Extra |
+----------------+-----------+------+-----+---------+-------+
| source_uuid    | char(36)  | NO   | PRI | NULL    |       |
| interval_start | bigint(20)| NO   | PRI | NULL    |       |
| interval_end   | bigint(20)| NO   |     | NULL    |       |
+----------------+-----------+------+-----+---------+-------+
3 rows in set (0.00 sec)

root@db 16:07:  [mysql]> select * from gtid_executed;
+--------------------------------------+----------------+--------------+
| source_uuid                          | interval_start | interval_end |
+--------------------------------------+----------------+--------------+
| 088f82f9-9ea1-11e7-8bb2-080027cd683a |              1 |       100025 |
+--------------------------------------+----------------+--------------+
1 row in set (0.00 sec)
```

该表会记录已经执行的 GTID 集合的信息，有了这张表，就不用再像 MySQL 5.6 版本时，必须开启 log_slave_updates 参数，从库才可以进行复制。GTID 信息会保存在 gtid_executed 表中，可以关闭从库的 binlog，节约 binlog 的记录开销。在执行 reset master 时，会清空表内所有的数据。

MySQL 5.7 还有一个 gtid_executed_compression_period 参数，用来控制 gtid_executed 表的压缩。该参数默认值为 1000，意味着表压缩在执行完 1000 个事务之后开始。

```
root@db 10:29:  [(none)]> show variables like '%gtid_executed%';
+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
| gtid_executed_compression_period | 1000  |
+---------------------------------+-------+
1 row in set (0.00 sec)
```

从 MySQL 5.7.6 开始，gtid_mode 支持动态修改，gtid_mode 可取值为：

- OFF——不支持 GTID 的事务；
- OFF_PERMISSIVE——新的事务是匿名的，同时允许复制的事务可以是 GTID，也可以是匿名的；
- ON_ PERMISSIVE——新的事务使用 GTID，同时允许复制的事务可以是 GTID，也可以是匿名的；
- ON——支持 GTID 的事务。

在生产环境中，可能有把传统复制改为 GTID 的复制模式的需求。这里特意强调一点，gtid_mode 虽然支持动态修改，但不支持跳跃式修改。从 ON_ PERMISSIVE 修改为 OFF 是不可以的。下面会有实验来展示传统复制与 GTID 复制之间的切换过程。

在从库上可通过 show slave status 命令来获取接收的 gtid（retrieve_gtid_set）和执行的 gtid（execute_gtid_set）。

11.5.4　GTID 复制与传统复制的切换

前面已经搭建好了 MySQL 5.7 版本的 GTID 复制模式，下面先来操作从 GTID 复制模式切换为传统复制模式的过程。

环境介绍：主库为 192.168.56.101，从库为 192.168.56.102。

当前主从状态展示：

```
                Master_Host: 192.168.56.101
                Master_User: bak
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-binlog.000002
        Read_Master_Log_Pos: 1141
             Relay_Log_File: node3-relay-bin.000006
              Relay_Log_Pos: 720
      Relay_Master_Log_File: mysql-binlog.000002
           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: 1141
            Relay_Log_Space: 1876
Retrieved_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:1-4
 Executed_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:1-4
```

实施过程如下：

（1）先在从库中执行 stop slave，停掉主从复制。然后调整为传统复制模式，让 master_auto_position=0。

执行如下命令：

```
CHANGE MASTER TO master_auto_position=0,Master_Host='192.168.56.101',
MASTER_USER='bak',MASTER_PASSWORD='bak123','Master_Log_File='mysql-binlog.00
0002',MASTER_LOG_POS=1141;
```

执行完成之后，开启复制功能 start slave。

（2）需要在主从服务器上同时调整 GTID 模式为 on_permissive。

```
root@db 14:40:  [(none)]> set global gtid_mode=on_permissive;
Query OK, 0 rows affected (0.04 sec)
```

（3）需要在主从服务器上同时调整 GTID 模式为 off_permissive。

```
root@db 14:41:  [(none)]> set global gtid_mode=off_permissive
;
Query OK, 0 rows affected (0.05 sec)
```

（4）需要在主从服务器上同时关闭 GTID 功能。

```
root@db 15:32:  [zs]> set global enforce_gtid_consistency=off;
Query OK, 0 rows affected (0.00 sec)

root@db 15:32:  [zs]> set global gtid_mode=off;
Query OK, 0 rows affected (0.00 sec)
```

（5）然后把 gtid_mode=off 和 enforce_gtid_consistency=off 写入配置文件 my.cnf 中，下次重启直接生效。

（6）测试是否切换成功。

首先向主库的 zs 库下的 tt 表中插入一条数据：

```
root@db 14:43:  [zs]> insert into tt (name,score) values ('gg',88);
Query OK, 1 row affected (0.04 sec)
```

查看从库，这条数据同步成功：

```
root@db 15:49:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
|  6 | ff   |    99 |
|  7 | gg   |    88 |
+----+------+-------+
7 rows in set (0.00 sec)
```

然后在从库中执行 show slave status 查看主从复制状态，发现 GTID 的值没有增加，证明切换成功：

```
Retrieved_Gtid_Set:
 Executed_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:1-4
```

然后再操作从传统复制模式切换为 GTID 复制模式的过程。

实施过程如下：

（1）在主从库上同时修改参数 enforce_gtid_consistency=warn，确保在 error log 中不会出现警告信息。如果有，需要先修复，才能往后继续执行。

```
root@db 15:56:  [zs]> set global enforce_gtid_consistency=warn;
Query OK, 0 rows affected (0.00 sec)
```

（2）在主从服务器上把 enforce_gtid_consistency 改为 on，保证 GTID 的一致性。

```
root@db 15:56:  [zs]> set global enforce_gtid_consistency=on;
Query OK, 0 rows affected (0.00 sec)
```

（3）在主从服务器上同时调整 GTID 模式为 off_permissive。

```
root@db 16:01:  [zs]> set global gtid_mode=off_permissive;
Query OK, 0 rows affected (0.32 sec)
```

（4）在主从服务器上同时调整 GTID 模式为 on_permissive。

```
root@db 16:03:  [zs]> set global gtid_mode=on_permissive;
Query OK, 0 rows affected (0.02 sec)
```

（5）确认从库的 Ongoing_anonymous_transaction_count 参数是否 0，如果为 0，意味着没有等待的事务，可以直接进行下一步操作了。

```
root@db 16:06:  [zs]> show global status like 'ongoing_anonymous_%';
+----------------------------------------+-------+
| Variable_name                          | Value |
+----------------------------------------+-------+
| Ongoing_anonymous_transaction_count    | 0     |
+----------------------------------------+-------+
1 row in set (0.01 sec)
```

（6）在主从库上同时设置 gtid_mode=on。

```
root@db 16:08:  [zs]> set global gtid_mode=on;
Query OK, 0 rows affected (0.04 sec)
```

查看 GTID 参数设置，目前都是开启状态：

```
root@db 16:11:  [zs]> show variables like '%gtid%';
+---------------------------------+-----------+
| Variable_name                   | Value     |
+---------------------------------+-----------+
| binlog_gtid_simple_recovery     | ON        |
| enforce_gtid_consistency        | ON        |
| gtid_executed_compression_period| 1000      |
| gtid_mode                       | ON        |
| gtid_next                       | AUTOMATIC |
| gtid_owned                      |           |
| gtid_purged                     |           |
| session_track_gtids             | OFF       |
+---------------------------------+-----------+
8 rows in set (0.00 sec)
```

（7）把传统复制模式改为 GTID 复制。先要把原有的传统复制停掉，执行 stop slave 操作，然后再执行 change master to master_auto_position=1。

执行完 stop slave，查看当前主从的状态为：

```
root@db 16:16:  [zs]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State:
                  Master_Host: 192.168.56.101
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000008
          Read_Master_Log_Pos: 194
               Relay_Log_File: node3-relay-bin.000014
                Relay_Log_Pos: 373
        Relay_Master_Log_File: mysql-binlog.000008
             Slave_IO_Running: No
            Slave_SQL_Running: No

Retrieved_Gtid_Set:
 Executed_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:1-4
```

然后再执行 change master to master_auto_position=1，开启主从复制 start slave。

（8）验证是否切换成功。

首先向主库的 zs 库下的 tt 表中插入一条数据：

```
root@db 16:35:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
|  6 | ff   |    99 |
|  7 | gg   |    88 |
+----+------+-------+
7 rows in set (0.00 sec)

root@db 16:35:  [zs]> insert into tt (name,score) values ('hh',85);
Query OK, 1 row affected (0.02 sec)
```

查看从库，这条数据同步成功：

```
root@db 16:33:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
|  6 | ff   |    99 |
|  7 | gg   |    88 |
|  8 | hh   |    85 |
+----+------+-------+
8 rows in set (0.00 sec)
```

然后在从库中执行 show slave status 查看主从复制状态，发现 GTID 的值增加了。证明开启了 GTID 复制方式，切换成功。

```
  Retrieved_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:5
   Executed_Gtid_Set: 1bdeac40-a4ee-11e7-a814-08002783b39d:1-5
```

11.5.5　GTID 使用中的限制条件

GTID 复制是针对事务来说的，一个事务只对应一个 GTID，好多的限制就在于此。

（1）不能使用 create table table_name select * from table_name。

（2）在一个事务中既包含事务表的操作又包含非事务表。

（3）不支持 CREATE TEMPORARY TABLE or DROP TEMPORARY TABLE 语句操作。

（4）使用 GTID 复制从库跳过错误时，不支持执行 sql_slave_skip_counter 参数的语法。

11.6 多源复制

所谓多源复制，就是把多台主库的数据同步到一台从库服务器上，从库会创建通往每个主库的管道。在 MySQL 5.7 之前的版本中，只能实现一主一从、一主多从或者多主多从的复制架构，如果想要实现多主一从的复制，只能使用 MariaDB。MySQL 5.7 版本已经可以实现多主一从的复制。它的搭建过程支持 GTID 模式和 binlog+position 方式。

多主一从的复制架构如图 11-2 所示。

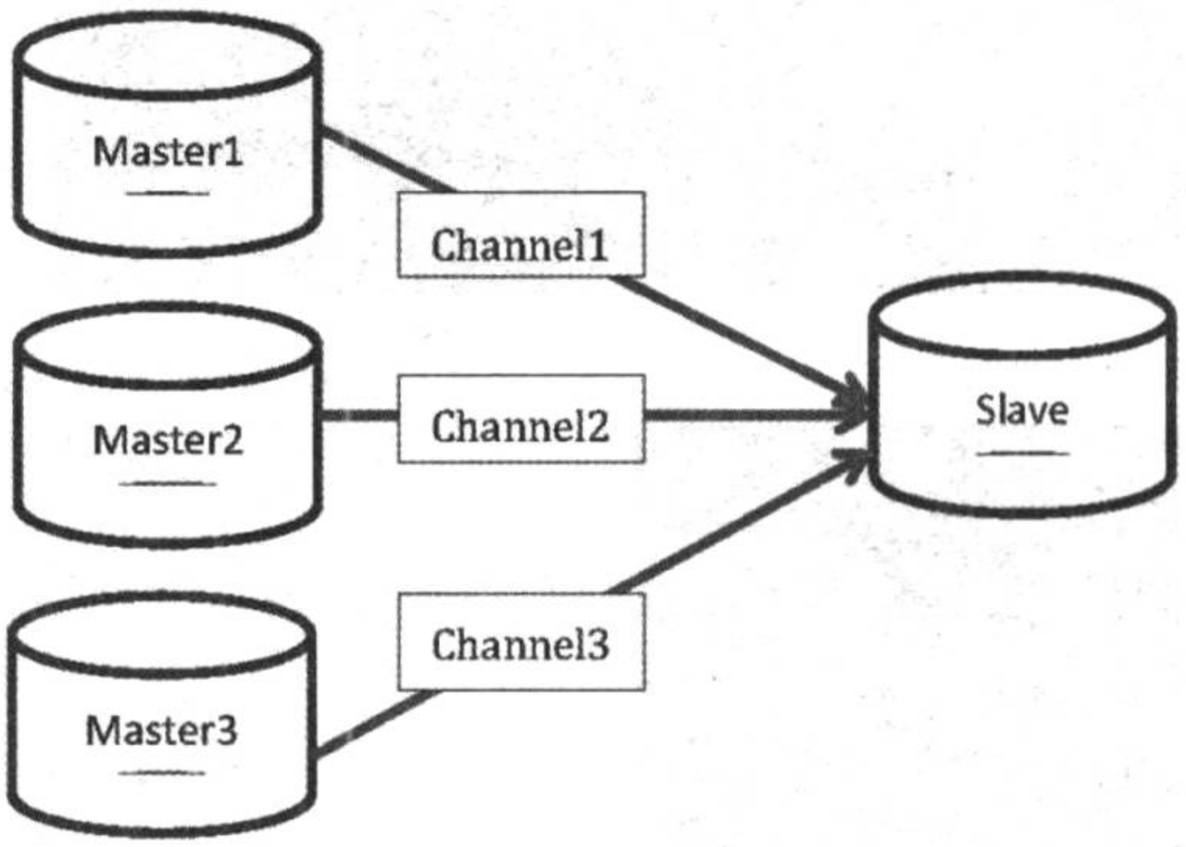


图 11-2　多主一从的复制架构

多源复制的优势：

（1）可以集中备份，在从库上备份，不会影响线上的数据库正常运行。

（2）节约了购买从库服务器的成本，只需要一个服务器即可。

（3）数据都汇总在一起，方便后期做数据统计。

多源复制搭建之前的环境参数介绍：

这里使用的两主一从的架构基于 MySQL 5.7 版本的 GTID 多源复制。

```
MasterA: 192.168.56.100。
MasterB: 192.168.56.101。
Slave: 192.168.56.102。
```

搭建中的注意事项：

MasterA 和 MasterB 不能拥有相同的数据库名字，否则就会在从库出现数据覆盖的现象。

MasterA→slave 与 MasterB→slave 要拥有不同的复制账号。

三台机器之间的数据库参数配置跟 GTID 复制的一样。保证开启 GTID 功能，server-id 之间不一致。binlog 格式为 row 模式。

这里需要特别强调的一点就是从库的参数需要配置：

```
master_info_repository=table
relay_log_info_repository=table
```

这是多源复制中需要注意的，主从间的复制信息需要记录到表中。

开始实施搭建过程：

（1）分别在 MasterA 和 MasterB 上创建复制账号。

MasterA：

```
root@db 09:50:  [(none)]> create user 'bak'@'192.168.56.%' identified by 'bak123';
Query OK, 0 rows affected (0.00 sec)

root@db 09:50:  [(none)]> grant replication slave on *.* to 'bak'@'192.168.56.%';
Query OK, 0 rows affected (0.01 sec)

root@db 09:50:  [(none)]> flush privileges;
Query OK, 0 rows affected (0.04 sec)
```

MasterB：

```
root@db 10:47:  [(none)]> create user 'repl'@'192.168.56.%' identified by 'repl123'
Query OK, 0 rows affected (0.04 sec)

root@db 10:47:  [(none)]> grant replication slave on *.* to 'repl'@'192.168.56.%';
Query OK, 0 rows affected (0.01 sec)
```

（2）分别在 MasterA 和 MasterB 中使用 mysqldump 工具导出需要备份的 tzy 和 zs 数据库，传递到从库机器上。

MasterA 的操作命令：

```
    /usr/local/mysql/bin/mysqldump -uroot -proot123 --master-data=2
--single-transaction tzy >tzy.sql
    scp tzy.sql 192.168.56.102:/root/
```

MasterB 的操作命令：

```
    /usr/local/mysql/bin/mysqldump -uroot -proot123 --master-data=2
--single-transaction zs >zs.sql
    scp zs.sql 192.168.56.102:/root/
```

（3）在从库上进行数据库的恢复操作。

```
mysql -uroot -proot123 tzy < tzy.sql
mysql -uroot -proot123 zs < zs.sql
```

（4）在从库上分别配置 MasterA→slave 和 MasterB→slave 的同步过程。

```
    CHANGE MASTER TO MASTER_HOST='192.168.56.100',MASTER_USER='bak',
MASTER_PASSWORD='bak123',master_auto_position=1 FOR CHANNEL 'm1';

     CHANGE MASTER TO MASTER_HOST='192.168.56.101',MASTER_USER='repl',
MASTER_PASSWORD='repl123',master_auto_position=1 FOR CHANNEL 'm2';
```

> 注：创建 FOR CHANNEL 'm1'、FOR CHANNEL 'm2'来管理从库通往主库的通道。这里有 m1 和 m2 两个通道。

（5）开启主从复制，可以通过 start slave 命令开启所有复制，也可以通过 start slave for channel 来分别开启。

```
root@db 10:34:  [(none)]> start slave for channel 'm1';
Query OK, 0 rows affected (0.02 sec)
```

```
root@db 10:35:  [(none)]> start slave for channel 'm2';
Query OK, 0 rows affected (0.02 sec)
```

通过 show slave status for channel 'm1'\G、show slave status for channel 'm2'\G 命令分别查看复制源 m1 和 m2 的主从同步状态信息；也可以通过 performance_schema 库下 replication_connection_configuration 表来查看复制配置信息，replication_connection_status 表中的内容用来监控主从复制状态。

```
root@db 11:57:  [performance_schema]> select * from replication_connection_status\G;
*************************** 1. row ***************************
             CHANNEL_NAME: m1
               GROUP_NAME:
              SOURCE_UUID: f0c8973e-a709-11e7-8a5c-080027f1fd08
                THREAD_ID: 136
            SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 6
 LAST_HEARTBEAT_TIMESTAMP: 2017-10-02 11:57:31
 RECEIVED_TRANSACTION_SET: f0c8973e-a709-11e7-8a5c-080027f1fd08:1-27
        LAST_ERROR_NUMBER: 0
       LAST_ERROR_MESSAGE:
     LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
```

```
*************************** 2. row ***************************
             CHANNEL_NAME: m2
               GROUP_NAME:
              SOURCE_UUID: 1bdeac40-a4ee-11e7-a814-08002783b39d
                THREAD_ID: 138
            SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 7
 LAST_HEARTBEAT_TIMESTAMP: 2017-10-02 11:57:31
 RECEIVED_TRANSACTION_SET:
        LAST_ERROR_NUMBER: 0
       LAST_ERROR_MESSAGE:
     LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
2 rows in set (0.00 sec)
```

（6）验证数据是否同步。

在 MasterA 上往 tzy 库下的 t1 表中插入一条数据：

```
root@db 15:05:  [tzy]> insert into t1 select 1;
Query OK, 1 row affected (0.05 sec)
Records: 1  Duplicates: 0  Warnings: 0

root@db 15:05:  [tzy]> select * from t1;
+------+
| id   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

在 MasterB 上往 zs 库下的 t1 表中插入一条数据：

```
root@db 15:07:  [zs]> insert into tt (name,score) values ('aa',60);
Query OK, 1 row affected (0.07 sec)

root@db 15:07:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
| 10 | aa   |    60 |
+----+------+-------+
1 row in set (0.00 sec)
```

在从库 slave 上查看数据都已经同步：

```
root@db 15:07:  [zs]> use zs;
Database changed
root@db 15:10:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
| 10 | aa   |    60 |
+----+------+-------+
1 row in set (0.00 sec)
```

```
root@db 15:10:  [zs]> use tzy;
Database changed
root@db 15:11:  [tzy]> select * from t1;
+------+
| id   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

MySQL 两主一从架构搭建成功。

11.7 主从延迟的解决方案及并行复制

在了解完 MySQL 主从复制的原理之后，我们知道主库可以并发写入，但从库只能通过单 SQL thread 完成任务（MySQL 5.7 之前），这是出现主从延迟的最核心原因。下面先介绍如何监控主从延迟。

这里推荐 percona-toolkit 工具集中的一个工具 pt-heartbeat。

它的监控原理就是先在主库创建一张 hearbeat 表，表中有个时间戳字段。主库上 pt-heartbeat 的 update 线程会在指定时间间隔更新时间戳。而从库上的 pt-heartbeat 的 monitor 线程会检查复制的心跳记录，这个记录就是主库修改的时间戳。然后和当前系统时间进行对比，得出时间上的差异，差异值就是延迟的时间大小。由于 heartbeat 表中有 server_id 字段，在监控某个从库的延迟时指定参考主库的 server_id 即可。

> 注：实施监控之前，一定要确保主从的时间要同步。

监控操作如下：

```
192.168.56.100 master  server-id=3306100
192.168.56.101 slave  server-id=3306101
```

pt-heartbeat 工具主要参数介绍。

- --daemonize：执行时，放入到后台执行。
- --create-table：在主库上创建心跳监控的表。
- --user　-u：连接数据库的账号。
- --database　-D：连接数据库的名称。
- --host　-h：连接的数据库地址。

- --password　-p：连接数据库的密码。
- --port　-P：连接数据库的端口。
- --monitor：持续监控从库的延迟情况。
- --master-server-id：指定主库的 server_id，若没有指定，则该工具会连到主库上查找其 server_id。
- --update：更新主库上的心跳表。

首先在主库上创建 heartbeat 心跳表，通过 update 执行更新时间戳操作，心跳表指定建立在 zs 库下。命令如下：

```
./pt-heartbeat -uroot -proot123 --database zs --update --create-table -
daemonize
```

```
root@db 08:56:  [zs]> select * from heartbeat;
+----------------------------+-----------+--------------------+----------+-----------------
---+--------------------+
| ts                         | server_id | file               | position | relay_master_log_fi
le | exec_master_log_pos |
+----------------------------+-----------+--------------------+----------+-----------------
---+--------------------+
| 2017-11-11T08:56:25.000840 |   3306100 | mysql-binlog.000010 |     8797 | NULL
   |                NULL |
+----------------------------+-----------+--------------------+----------+-----------------
---+--------------------+
1 row in set (0.00 sec)
```

然后就可以对从库进行监控工作了。指定参考主库的 server-id 值即可。从库操作命令如下：

```
[root@node2 bin]# ./pt-heartbeat --master-server-id=3306100 --monitor --database zs  -uroot -proot123
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 ]
```

结果表示当前没有主从延迟。中括号中的数值分别表示 1m、5m、15m 的平均值。

其他的延迟原因可以总结为：

（1）MySQL 主从之间的同步本来就不是实时同步的，而是异步的同步，也就是说，主库提交事务之后，从库才再执行一遍。

（2）在主库上对没有索引大表的列进行 delete 或者 update 的操作。

（3）从库的硬件配置没有主库的好，经常忽略从库的重要性。

（4）网络抖动导致 I/O 线程复制延迟。

针对延迟的解决办法：

（1）使用 MySQL 5.7 的并行复制功能。在 5.6 版本中就有了并行的概念，但其中的并行复制是基于库级别的，即 slave_parallel_type=database。但在这种模式下，只是基于多库少表的情况，并不适用于真实的生产环境下。在 MySQL 5.7 版本中，真正实现了基于组提交的并行复制，简单说就是主库并行执行 SQL 语句，从库也可以通过多个 workers 线程并发执行 relay log 中主库提交的事务。想要开启 MySQL 5.7 的并行复制，可以在从库设置参数 slave_parallel_workers>0，并把 5.7 版本中新添加的 slave_parallel_type 参数设置为 LOGICAL_CLOCK。该参数有 DATABASE 和 LOGICAL_CLOCK 两个值。MySQL 5.6 默认是 database。

```
root@db 16:01:  [(none)]> show variables like '%slave_parallel_%';
+------------------------+---------------+
| Variable_name          | Value         |
+------------------------+---------------+
| slave_parallel_type    | LOGICAL_CLOCK |
| slave_parallel_workers | 16            |
+------------------------+---------------+
2 rows in set (0.00 sec)
```

（2）可以采用 Percona 公司的 percona-xtradb-cluster（简称 PXC 架构），这种架构下可以实现多节点写入，达到实时同步。

（3）业务初期规划时就要选择合适的分库、分表策略，避免单表或者单库过大，带来额外的复制压力，从而带来主从延迟的问题。

（4）避免一些无用的 I/O 消耗，可以增加高转速的磁盘、SSD 或者 PCIE-SSD 设备。

（5）阵列级别要选择 RAID10，raid cache 策略要使用 WB，坚决不要使用 WT。

（6）I/O 调度要选择 deadline 模式。

（7）适当调整 buffer pool 的大小。

（8）避免让数据库进行各种大量运算，要记住数据库只是用来存储数据的，让应用端多分担些压力，或者可以通过缓存来完成。

11.8 主从复制的数据校验

在工作中接触和处理的最多的问题就是 MySQL 的主从同步了，有时发生主库宕机，我们就面临着主从切换的问题，要把从库提升为主库。但主从库之间的数据一致性不能保证，所以就会利用 percona-toolkit 工具集中的 pt-table-checksum 工具来检查主从数据的一致性，然后再通过 pt-table-sync 工具来修复不一致的数据信息。

之前的章节已经介绍过 percona-toolkit 的安装方法，所以我们直接来看命令的使用。

pt-table-checksum 用于校验主从数据的一致性，该命令在主库上执行校验查询，然后对复制的一致性进行检查，来对比主从之间的校验值，并输出对比结果。

使用方法：可以通过 perldoc ./pt-table-checksum 来查看它的使用手册。

重要的参数解析。

- --[no]check-replication-filters

  是否检查复制的过滤器，默认是 yes，建议启用不检查模式。

- --databases

  指定需要被检查的数据库，多个库之间可以用逗号分隔。

- --[no]check-binlog-format

  是否检查 binlog 文件的格式，默认值是 yes，建议开启不检查。因为在默认的 row 格式下会出错。

- --replicate

  把 checksum 的信息写入到指定表中。

- --replicate-check-only

  只显示不同步信息。

- --tables

  指定需要被检查的表，多个表之间用逗号分隔。

- --where

  大表校验的过程，可以使用 where 来过滤一些条件。

主库为 192.168.56.101，从库为 192.168.56.102。

校验 zs 库下 tt 表的数据信息：

```
pt-table-checksum --no-check-binlog-format --replicate=zs.checksums
--databases=zs --tables=tt -uzs -p123456 -h 192.168.56.101
```

```
[root@node2 bin]# ./pt-table-checksum --no-check-binlog-format --replicate=zs.check
sums --databases=zs --tables=tt -uzs -p123456 -h 192.168.56.101
            TS ERRORS  DIFFS     ROWS  CHUNKS SKIPPED    TIME TABLE
10-02T17:41:31      0      1        2       1       0   0.303 zs.tt
```

输出结果分析。

- TS：完成检查的时间。
- ERRORS：检查时发生错误和警告的数量。

- DIFFS：0 表示没有不一致发生，1 表示出现不一致。当指定--no-replicate-check 时会一直为 0，当指定--replicate-check-only 时会显示不同的信息。本例中值为 1，代表出现了数据不一致的情况。
- ROWS：表的行数。本例中 tt 表有 2 条记录。
- CHUNKS：被划分到表中的块的数目。
- SKIPPED：由于错误或警告或过大，跳过块的数目。
- TIME：执行的时间。
- TABLE：被检查的表名。本例为 zs 库下的 tt 表。

在主库执行完命令之后，还会在主从服务器的 zs 库下分别生成一张 checksums 表。

在主库（192.168.56.101）上查看 zs.checksums：

```
root@db 17:49:  [zs]> select * from checksums \G;
*************************** 1. row ***************************
            db: zs
           tbl: tt
         chunk: 1
    chunk_time: 0.026494
   chunk_index: NULL
lower_boundary: NULL
upper_boundary: NULL
      this_crc: ea79efcb
      this_cnt: 2
    master_crc: ea79efcb
    master_cnt: 2
            ts: 2017-10-02 17:41:31
1 row in set (0.00 sec)
```

在从库（192.168.56.102）上查看 zs.checksums：

```
root@db 17:55:  [zs]> select * from checksums \G;
*************************** 1. row ***************************
            db: zs
           tbl: tt
         chunk: 1
    chunk_time: 0.026494
   chunk_index: NULL
lower_boundary: NULL
upper_boundary: NULL
      this_crc: 7ae23197
      this_cnt: 4
    master_crc: ea79efcb
    master_cnt: 2
            ts: 2017-10-02 17:41:31
1 row in set (0.00 sec)
```

pt-table-checksum 的原理就是针对某张表中的所有字段进行 hash 函数运算，在主库上经运算后，把得到的值记录为 master_crc、master_cnt，在从库上经运算后，把得到的值记录为 this_crc、

this_cnt。最后通过比较从库 this 的值和主库 master 的值来判断主从之间的数据一致性。

本例中，主库 master_crc= ea79efcb，master_cnt= 2；从库 this_crc= 7ae23197，this_cnt=4。

显然 this 和 master 的值不一致。

通过在从库中执行如下 SQL 语句，可以找到数据不一致的地方：

```
SELECT db, tbl, SUM(this_cnt) AS total_rows, COUNT(*) AS chunks FROM
zs.checksums WHERE (  master_cnt <> this_cnt  OR master_crc <> this_crc  OR
ISNULL(master_crc) <> ISNULL(this_crc)) GROUP BY db, tbl;
```

```
+----+-----+------------+--------+
| db | tbl | total_rows | chunks |
+----+-----+------------+--------+
| zs | tt  |          4 |      1 |
+----+-----+------------+--------+
1 row in set (0.00 sec)
```

不一致的数据是 zs 库下的 tt 这张表，表中总共有 4 条数据。

在主库中执行 select count(*) from zs.tt，发现只有 2 条数据，证明从库 zs 库下的 tt 表比主库 zs 库的 tt 表多 2 条记录。

```
root@db 18:16:  [zs]> select count(*) from tt;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)
```

接下来就该使用 pt-table-sync 工具来修复主从数据不一致的位置。pt-table-sync 核心参数解析如下。

- --replicate：指定通过 pt-table-checksum 工具得到的表。本例中是放在 zs 库下。
- --print：打印但不执行命令。
- --execute：执行命令。

在主库上通过执行 pt-table-sync 命令，先打印出需要执行修复的 SQL 语句。这里配合--print 参数：

```
pt-table-sync --replicate=zs.checksums h=192.168.56.101,u=zs,p=123456 -
print
```

```
DELETE FROM `zs`.`tt` WHERE `id`='12' LIMIT 1 /*percona-toolkit src_db:zs src_tbl:t
t src_dsn:h=192.168.56.101,p=...,u=zs dst_db:zs dst_tbl:tt dst_dsn:h=192.168.56.102
,p=...,u=zs lock:1 transaction:1 changing_src:zs.checksums replicate:zs.checksums b
idirectional:0 pid:4857 user:root host:node2*/;
DELETE FROM `zs`.`tt` WHERE `id`='13' LIMIT 1 /*percona-toolkit src_db:zs src_tbl:t
t src_dsn:h=192.168.56.101,p=...,u=zs dst_db:zs dst_tbl:tt dst_dsn:h=192.168.56.102
,p=...,u=zs lock:1 transaction:1 changing_src:zs.checksums replicate:zs.checksums b
idirectional:0 pid:4857 user:root host:node2*/;
```

接着再在主库上执行 pt-table-sync 命令，这次使用--execute 参数来真正修复主从服务器的 zs 库不一致的数据，命令如下：

```
./pt-table-sync  --replicate=zs.checksums  h=192.168.56.101,u=zs,p=123456
--execute
```

最后校验主从数据是否一致。

在主库重新执行一次如下命令：

```
./pt-table-checksum --no-check-binlog-format --replicate=zs.checksums
--databases=zs --tables=tt -uzs -p123456 -h 192.168.56.101
```

```
            TS ERRORS  DIFFS     ROWS  CHUNKS SKIPPED    TIME TABLE
10-02T18:35:10      0      0        2       1       0   1.168 zs.tt
```

结果显示 diff 记录为 0，没有差异，zs 库下的 tt 表主从数据一致。

11.9 总结

本章介绍了复制的原理，各种复制方式的搭建过程，复制故障处理，延迟问题及其主从数据校验等。生产中建议使用 GTID+rows 复制模式，直接根据 GTID 找到同步的位置，不需要再去查找 binlog 和 position 的位置，对于维护主从架构太方便了。学好 MySQL 的复制也是为了后期的高可用集群架构打下基础。在实战中才能发现自己的问题，理论一定要配合实践。

第 4 部分　尊贵铂金篇

我们又晋升了段位，已经来到铂金了，离最终的王者又进了一步。学习 MySQL 和玩游戏差不多，前期都是熟悉的过程。熟悉之后就需要接触更主流的技术，以及和高水平的人一起交流经验，让自己进入高阶层面，绝对不能停滞不前。因为学习技术，需要的就是不断学习，有一种开拓进取不服输的精神才行。本部分介绍 MySQL 的高可用集群架构，主要包括 MHA、Keepalived+双主、PXC 以及中间件 ProxySQL 等知识点。

MySQL 高可用集群

所谓高可用性其实就是应用可以提供持续不间断服务的能力。有些读者对高可用和负载均衡两者的概念经常混淆。负载均衡是指后端服务器没有状态，可以任意分配，它们之间是同时工作的，可以通过增加机器来增加系统的吞吐量。而高可用是指后端服务器是有状态的，通常是一主一备的结构，它们之间是一个工作而另一个不工作等待着接管服务，以此来保证系统的可用性。一般系统高可用性用可用率来评价，100%的可用性是不存在的。在生产中我们努力让

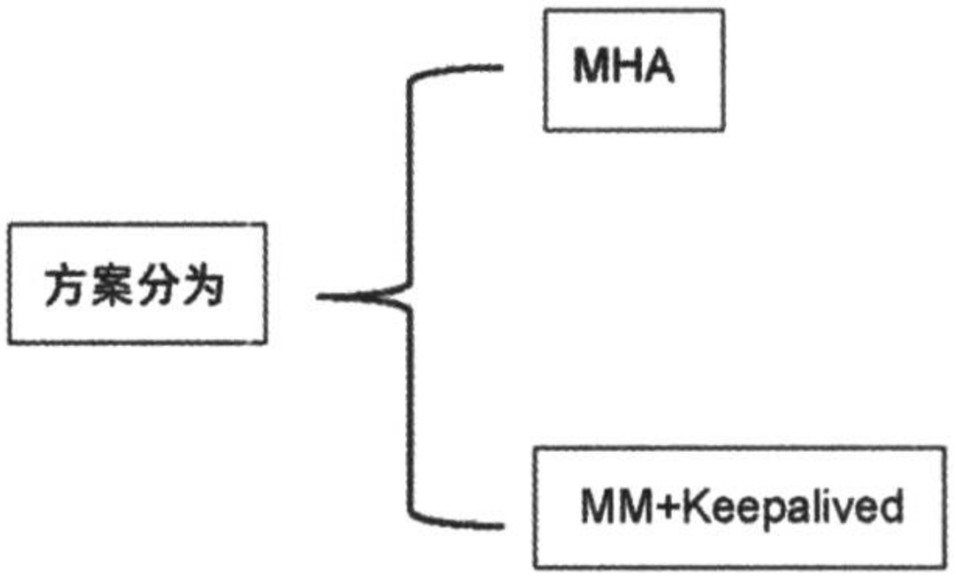
系统的可用性达到 4 个 9 的状态（99.99%），即全年业务不可用的时间大概只有 52 分钟。可用性 9 原则列表如下表所示。

可　用　率	全年系统不可用时间
99%	365×24×0.01=87.6 小时
99.9%	365×24×0.001=8.76 小时
99.99%	365×24×60×0.0001=52.56 分钟
99.999%	365×24×60×0.00001=5.256 分钟

从表中可以看到，要达到两个 9 或者三个 9 的目标还是非常容易的，一般的公司都可以达到。随着可用性的提高，所花费的成本也会随之提高。所以要权衡好一个最合适的高可用方案，如果钱没少花，但收益并不明显，那就事与愿违了。

造成业务不可用的原因有很多，可能是系统软硬件的故障，应用程序代码不完善存在 bug，或是遭到外界的恶意攻击等。实现高可用性最有效的手段就是冗余策略。针对数据库服务在冗余实现和可用性上要考虑数据和服务两个方面。MySQL 可选的高可用实现方案多种多样，这里主要介绍以下几种基础的高可用方案。

（1）最常用的就是基于主从复制的方案。



（2）基于 Galera 协议的 Percona XtraDB Cluster（简称 PXC），实现真正意义上同步复制。

（3）基于中间件 Proxy，主要介绍 ProxySQL。

第 12 章 MHA

12.1 MHA 简介

MHA，即 MasterHigh Availability Manager and Toolsfor MySQL，是日本的一位 MySQL 专家采用 Perl 语言编写的一个脚本管理工具，该工具仅适用于 MySQL Replication 环境，目的在于维持 master 主库的高可用性。MHA（Master High Availability）是自动的 master 故障转移和 slave 提升的软件包，基于标准的 MySQL 复制（异步/半同步）。

目前 MHA 在生产环境中使用得比较多，它包含两个组成部分：MHA Manager（管理节点）和 MHA Node（数据节点）。

下载地址：https://github.com/yoshinorim/mha4mysql-manager；

https://github.com/yoshinorim/mha4mysql-node。

12.1.1 MHA 部署

MHA Manager 管理节点可以单独部署在一台独立服务器上管理多个 master-slave 集群，也可以部署在一台 slave 上。MHA Manager 探测集群中的 node 节点，当发现 master 出现故障时，它可以自动将具有最新数据的 slave 提升为新的 master，然后将所有其他 slave 导向新的 master 上。整个故障转移过程对应用程序是透明的。MHA node 数据节点可以运行在每台 MySQL 服务

器上（master/slave/manager），它通过监控具备解析和清理 logs 功能的脚本来加快故障转移。

12.1.2 MHA 原理

MHA 的目的在于维持 MySQL Replication 中 master 库的高可用性，其最大特点是可以修复多个 slave 之间的差异日志，最终使所有 slave 保持数据一致，然后从中选择一个充当新的 master，并将其他 slave 指向它。当 master 出现故障时，可以通过对比 slave 之间 I/O thread 读取主库 binlog 的 position 号，选取最接近的 slave 作为备选主库（备胎）。其他的从库可以通过与备选主库对比生成差异的中继日志，在备选主库上应用从原来 master 保存的 binlog，同时将备选主库提升为 master。最后在其他 slave 上应用相应的差异中继日志并从新的 master 开始复制。

12.1.3 MHA 的优缺点

1. 优点

（1）故障切换时，可以自行判断哪个从库与主库的数据最接近，然后切换到上面，可以减少数据的丢失，保证数据的一致性。

（2）支持 binlog server，可提高 binlog 的传送效率，进一步减少数据丢失的风险。

（3）结合 MySQL 5.7 的增强半同步功能，确保故障切换时数据不丢失。

2. 缺点

（1）自动切换的脚本太简单了，而且比较老化，建议后期逐渐完善。

（2）搭建 MHA 架构，需要开启 Linux 系统互信协议，所以对于系统安全性来说是个不小的考验。

12.1.4 MHA 工具包的功能

1. Manager 管理工具

- #masterha_check_ssh：检查 MHA 的 SSH 配置。
- #masterha_check_repl：检查 MySQL 数据库的主从复制功能。
- #masterha_manager：启动 MHA 服务。
- #masterha_check_status：检测当前 MHA 运行状态。
- #masterha_master_monitor：监测 master 是否宕机。

- #masterha_master_switch：控制故障转移（自动或手动）。
- #masterha_conf_host：添加或删除配置的 server 信息。

2. Node 数据节点工具

- #save_binary_logs：保存和复制 master 的二进制日志。
- #apply_diff_relay_logs：识别差异的中继日志事件并应用于其他 slave。
- #filter_mysqlbinlog：去除不必要的 ROLLBACK 事件（MHA 已不再使用这个工具）。
- #purge_relay_logs：清除中继日志（不会阻塞 SQL 线程）。

12.2　实战演练

环境介绍：

使用三台机器来完成本次 MHA 的搭建过程。

192.168.56.100　master node（作为主库，并需要在上面安装数据节点）

192.168.56.101　slave1 node（作为第一个从库，并需要在上面安装数据节点）

192.168.56.102　slave2　manager，node（作为第二个从库，在其上面既安装管理节点，又安装数据节点）

vip 192.168.56.123　虚拟 IP 地址

实验拓扑如图 12-1 所示。

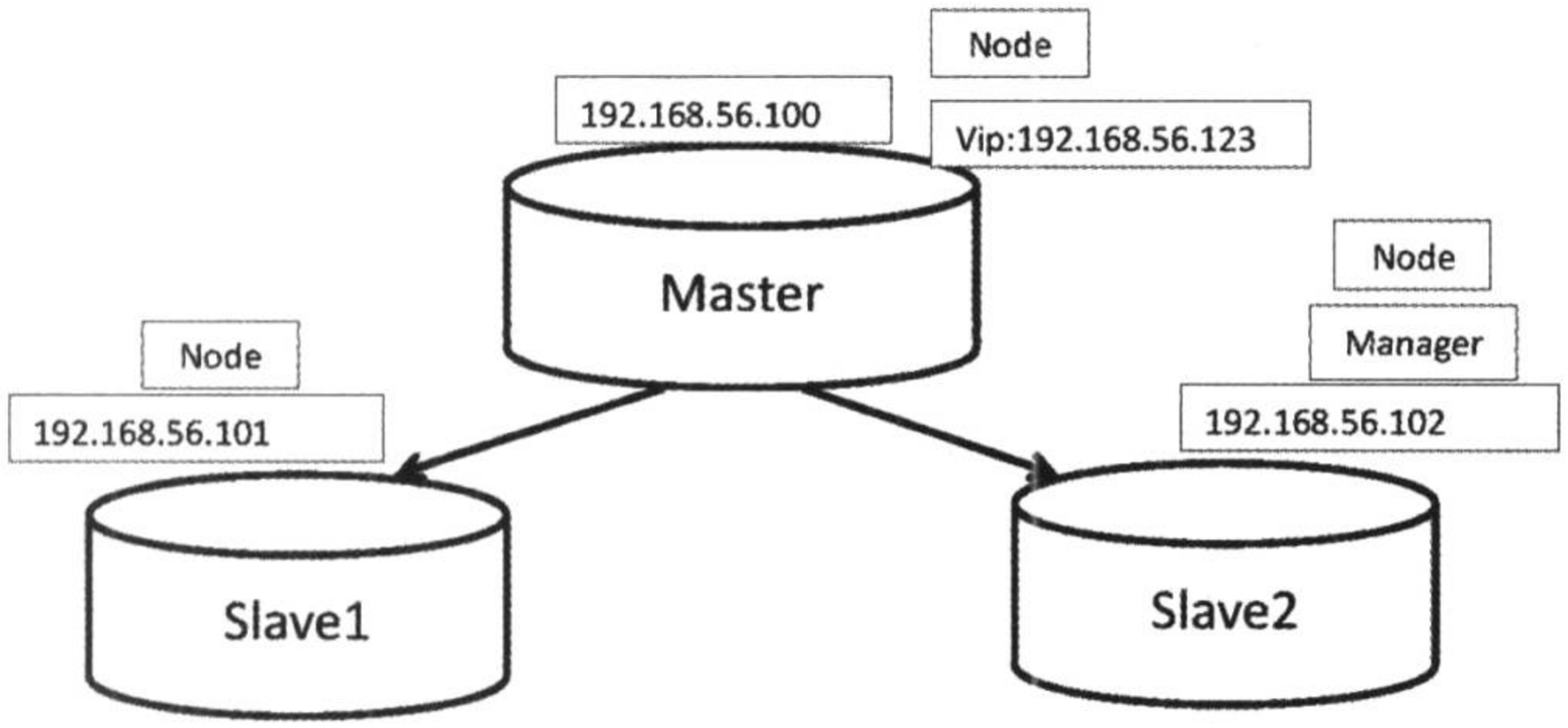


图 12-1　实验拓扑图

实验第一步

配置三台机器的互信：

首先在主库（192.168.56.100）上执行生成密钥操作。命令如下：

```
ssh-keygen -t dsa -P '' -f id_dsa
```

id_dsa.pub 为公钥，id_dsa 为私钥，紧接着将公钥文件复制成 authorized_keys 文件，这个步骤是必需的，过程如下：

```
cat id_dsa.pub >> authorized_keys
```

其次再从库（192.168.56.101）上执行生成密钥操作。命令如下：

```
ssh-keygen -t dsa -P '' -f id_dsa
```

也需要将公钥文件复制成 authorized_keys 文件。过程如下：

```
cat id_dsa.pub >> authorized_keys
```

然后在管理节点（192.168.56.102）上生成密钥。命令如下：

```
ssh-keygen -t dsa -P '' -f id_dsa
```

同样需要把公钥文件复制成 authorized_keys 文件。过程如下：

```
cat id_dsa.pub >> authorized_keys
```

在主库（192.168.56.100）上执行接收密钥的过程：

```
scp 192.168.56.101:/root/.ssh/id_dsa.pub ./id_dsa.pub.101
scp 192.168.56.102:/root/.ssh/id_dsa.pub ./id_dsa.pub.102
```

接收完成后执行合并密钥的命令：

```
cat id_dsa.pub.101 >> authorized_keys
cat id_dsa.pub.102 >> authorized_keys
```

在主库（192.168.56.100）上传送合成的密钥给其他两台机器：

```
scp authorized_keys 192.168.56.101:/root/.ssh/
scp authorized_keys 192.168.56.102:/root/.ssh/
```

在三台服务器上编辑/etc/hosts 文件，然后分别加入三台主机的 hostname。三台机器上都需要完成此步：

```
vim /etc/hosts
node1 192.168.56.100
node2 192.168.56.101
node3 192.168.56.102
```

验证主机名登录，互信验证过程：

分别在三台机器上执行以下操作。

在 192.168.56.100 执行：

```
ssh node2
ssh node3
```

在 192.168.56.101 上执行：

```
ssh node1
ssh node3
```

在 192.168.56.102 上执行：

```
ssh node1
ssh node2
```

都不需要输入密码，直接登录；代表主机互信配置成功。

实验第二步

搭建主从环境，这里配置一主两从环境。使用的是 MySQL 5.7 版本，基于 GTID+row 模式进行搭建。

在所有机器上都要执行以下操作。

创建主从复制账号：

```
create user 'repl'@'192.168.56.%' identified by 'repl';
grant replication slave on *.* to 'repl'@'192.168.56.%';
flush privileges;
```

```
root@db 15:15:  [(none)]> create user 'repl'@'192.168.56.%' identified by 'repl';
Query OK, 0 rows affected (0.00 sec)

root@db 15:15:  [(none)]> grant replication slave on *.* to 'repl'@'192.168.56.%';
Query OK, 0 rows affected (0.00 sec)

root@db 15:16:  [(none)]> flush privileges;
Query OK, 0 rows affected (0.02 sec)
```

创建管理账号：

```
create user  'zs'@'192.168.56.%' identified by '123456';
grant all privileges on *.* to 'zs'@'192.168.56.%' ;
flush privileges;
```

```
root@db 15:20:  [mysql]> create user  'zs'@'192.168.56.%' identified by '123456';
Query OK, 0 rows affected (0.00 sec)

root@db 15:20:  [mysql]> grant all privileges on *.* to 'zs'@'192.168.56.%' ;
Query OK, 0 rows affected (0.00 sec)

root@db 15:20:  [mysql]> flush privileges;
Query OK, 0 rows affected (0.01 sec)
```

在主库（192.168.56.100）上执行复制数据到两个从库，完成在某个 GTID 时的同步。使用 mysqldump 完成此复制数据的过程。由于是 GTID 方式的复制，就不需要添加--master-data 参数了。命令如下：

```
/usr/local/mysql/bin/mysqldump --single-transaction -uroot -proot123  -A > all.sql
```

把 all.sql 传递给其他两个从库，并执行恢复操作。

```
[root@node1 bin]# scp all.sql 192.168.56.101:/root/
all.sql                                       100%  751KB 750.7KB/s   00:00
[root@node1 bin]# scp all.sql 192.168.56.102:/root/
all.sql                                       100%  751KB 750.7KB/s   00:00
```

```
[root@node2 ~]# mysql -uroot -proot123 < all.sql
[root@node2 ~]#
```

```
[root@node3 ~]# mysql -uroot -proot123 < all.sql
[root@node3 ~]#
```

在两个从库中分别配置主从复制命令并开启主从同步。

命令如下：

```
    CHANGE MASTER TO MASTER_HOST='192.168.56.100',MASTER_USER='repl',
MASTER_PASSWORD='repl',MASTER_AUTO_POSITION=1;
    start slave;
```

最后在两台从库上查看主从同步状态，SQL 和 I/O thread 都正常运行，呈现 Yes 状态。

```
root@db 15:40:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.100
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000007
          Read_Master_Log_Pos: 3180
               Relay_Log_File: node2-relay-bin.000002
                Relay_Log_Pos: 423
        Relay_Master_Log_File: mysql-binlog.000007
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
```

至此，一主两从复制架构搭建成功。

实验第三步：安装 MHA-node 节点

分别在主库（192.168.56.100）、slave1（192.168.56.101）和 slave2 （192.168.56.102）上面安装数据节点。

首先安装 MySQL 依赖的 Perl 环境：

```
yum install perl-DBD-MySQL
```

解压数据节点的包：

```
tar -zxvf mha4mysql-node-0.57.tar.gz
```

安装 perl-cpan 软件包：

```
cd mha4mysql-node-0.57
yum -y install perl-CPAN*
perl Makefile.PL
make && make install
```

实验第四步：安装配置 MHA-manager 管理节点

以下操作都是在 192.168.56.102 上完成的。

在 slave2 机器上（192.168.56.102）安装管理节点。

先要安装环境需要的介质包：

```
yum install -y perl-DBD-MySQL*
rpm -ivh perl-Params-Validate-0.92-3.el6.x86_64.rpm
rpm -ivh perl-Config-Tiny-2.12-1.el6.rfx.noarch.rpm
rpm -ivh perl-Log-Dispatch-2.26-1.el6.rf.noarch.rpm
rpm-ivh perl-Parallel-ForkManager-0.7.5-2.2.el6.rf.noarch.rpm
```

然后安装管理节点：

```
tar -zxvf mha4mysql-manager-0.57.tar.gz
perl Makefile.PL
make
make install
```

进行管理节点 MHA 的配置过程。

创建 MHA 家目录，编辑启动配置文件。

```
mkdir -p /usr/local/mha
mkdir -p /etc/mha
cd /etc/mha/
vim mha.conf
[server default]
user=zs
password=123456
manager_workdir=/usr/local/mha
manager_log=/usr/local/mha/manager.log
remote_workdir=/usr/local/mha
ssh_user=root
repl_user=repl
repl_password=repl
ping_interval=1
master_ip_failover_script=/usr/local/scripts/master_ip_failover
master_ip_online_change_script=/usr/local/scripts/master_ip_online_change
```

```
[server1]
hostname=192.168.56.100
ssh_port=22
master_binlog_dir=/data/mysql
candidate_master=1
port=3306
[server2]
hostname=192.168.56.101
ssh_port=22
master_binlog_dir=/data/mysql
candidate_master=1
port=3306
[server3]
hostname=192.168.56.102
ssh_port=22
master_binlog_dir=/data/mysql
no_master=1
port=3306
```

配置参数详解。

- User：MySQL 管理员命令，建议赋予 all privileges 权限的用户名。
- Password：上面 MySQL user 的密码。
- manager_workdir：MHA manager 使用的工作目录。
- manager_log：MHA manager 生成日志的绝对路径。
- remote_workdir：每一个 MHA node 生成日志文件的工作路径，该路径是绝对路径。
- ssh_user：MHA manager 和 MHA node 通过这个用户连接到 MySQL Server 的操作系统上。
- repl_user：复制时使用的用户名。
- repl_password：复制时使用的用户名的密码。
- ping_interval：MHA manager ping 主库的间隔，当连续三次 ping 失败之后，MHA manager 认为这个 MySQL 主库宕机，单位是秒。
- master_ip_failover_script：failover 切换的脚本。
- master_ip_online_change_script：在线切换脚本。
- hostname：目标实例的主机名或者 IP 地址。

- ssh_port：目标数据库的 SSH 端口，默认是 22。
- master_binlog_dir：master 上生成 binlog 的绝对路径。
- candidate_master：从不同的从库服务器中，提升一个可靠的库作为新主库。如果设置 candidate_master 的值为 1，那么这个 server 会优先成为 master。本例中 192.168.56.101 优先成为主库。
- no_master：no_master=1 是默认值，意味着这个 server 从来不会成为新的 master，这个参数用来标记某些从来不用成为新主的服务器。

本例中 192.168.56.102 永远不会成为主库。

创建 failover，online 脚本的目录：

```
mkdir -p /usr/local/scripts
```

编辑 failover 切换脚本：

```
vim master_ip_failover
#!/usr/bin/env perl

use strict;
use warnings FATAL => 'all';

use Getopt::Long;
my (
    $command,          $ssh_user,        $orig_master_host, $orig_master_ip,
    $orig_master_port, $new_master_host, $new_master_ip,    $new_master_port
);

my $vip = '192.168.56.123/24';
my $key = '0';
my $ssh_start_vip = "/sbin/ifconfig eth0:$key $vip";
my $ssh_stop_vip = "/sbin/ifconfig eth0:$key down";

GetOptions(
    'command=s'          => \$command,
    'ssh_user=s'         => \$ssh_user,
    'orig_master_host=s' => \$orig_master_host,
    'orig_master_ip=s'   => \$orig_master_ip,
```

```
    'orig_master_port=i' => \$orig_master_port,
    'new_master_host=s'  => \$new_master_host,
    'new_master_ip=s'    => \$new_master_ip,
    'new_master_port=i'  => \$new_master_port,
);

exit &main();

sub main {

    print "\n\nIN SCRIPT TEST====$ssh_stop_vip==$ssh_start_vip===\n\n";

    if ( $command eq "stop" || $command eq "stopssh" ) {

        my $exit_code = 1;
        eval {
            print "Disabling the VIP on old master: $orig_master_host \n";
            &stop_vip();
            $exit_code = 0;
        };
        if ($@) {
            warn "Got Error: $@\n";
            exit $exit_code;
        }
        exit $exit_code;
    }
    elsif ( $command eq "start" ) {

        my $exit_code = 10;
        eval {
            print "Enabling the VIP - $vip on the new master - $new_master_host \n";
            &start_vip();
            $exit_code = 0;
        };
        if ($@) {
            warn $@;
            exit $exit_code;
        }
```

```
        exit $exit_code;
    }
    elsif ( $command eq "status" ) {
        print "Checking the Status of the script.. OK \n";
        exit 0;
    }
    else {
        &usage();
        exit 1;
    }
}


sub start_vip() {
    `ssh $ssh_user\@$new_master_host \" $ssh_start_vip \"`;
}
sub stop_vip() {
     return 0  unless  ($ssh_user);
    `ssh $ssh_user\@$orig_master_host \" $ssh_stop_vip \"`;
}

    sub usage {
        print
        "Usage: master_ip_failover --command=start|stop|stopssh|status --orig_
master_host=host --orig_master_ip=ip
                --orig_master_port=port --new_master_host=host --new_master_ip=
ip --new_master_port=port\n";
    }
```

编辑 online_change 的脚本，代码请见 www.broadview.com.cn/33679 中下载资源。

创建完这两个脚本，记得赋予执行权限：

```
chmod +x master_ip_failover
chmod +x master_ip_online_change
```

利用 MHA 工具检测 SSH。

安装需要的环境包：

```
yum -y  install perl-Time-HiRes
```

执行检测命令：

```
/usr/local/bin/masterha_check_ssh --conf=/etc/mha/mha.conf
```

输出显示结果：

```
    Tue Oct 10 10:13:14 2017 - [warning] Global configuration file
/etc/masterha_default.cnf not found. Skipping.
    Tue Oct 10 10:13:14 2017 - [info] Reading application default configuration
from /etc/mha/mha.conf..
    Tue Oct 10 10:13:14 2017 - [info] Reading server configuration from
/etc/mha/mha.conf..
    Tue Oct 10 10:13:14 2017 - [info] Starting SSH connection tests..
    Tue Oct 10 10:13:15 2017 - [debug]
    Tue Oct 10 10:13:14 2017 - [debug]  Connecting via SSH from
root@192.168.56.100(192.168.56.100:22) to
root@192.168.56.101(192.168.56.101:22)..
    Tue Oct 10 10:13:14 2017 - [debug]   ok.
    Tue Oct 10 10:13:14 2017 - [debug]  Connecting via SSH from
root@192.168.56.100(192.168.56.100:22) to
root@192.168.56.102(192.168.56.102:22)..
    Tue Oct 10 10:13:15 2017 - [debug]   ok.
    Tue Oct 10 10:13:15 2017 - [debug]
    Tue Oct 10 10:13:14 2017 - [debug]  Connecting via SSH from
root@192.168.56.101(192.168.56.101:22) to
root@192.168.56.100(192.168.56.100:22)..
    Tue Oct 10 10:13:15 2017 - [debug]   ok.
    Tue Oct 10 10:13:15 2017 - [debug]  Connecting via SSH from
root@192.168.56.101(192.168.56.101:22) to
root@192.168.56.102(192.168.56.102:22)..
    Tue Oct 10 10:13:15 2017 - [debug]   ok.
    Tue Oct 10 10:13:16 2017 - [debug]
    Tue Oct 10 10:13:15 2017 - [debug]  Connecting via SSH from
root@192.168.56.102(192.168.56.102:22) to
root@192.168.56.100(192.168.56.100:22)..
```

```
    Tue Oct 10 10:13:15 2017 - [debug]   ok.
    Tue Oct 10 10:13:15 2017 - [debug]  Connecting via SSH from
root@192.168.56.102(192.168.56.102:22) to
root@192.168.56.101(192.168.56.101:22)..
    Tue Oct 10 10:13:16 2017 - [debug]   ok.
    Tue Oct 10 10:13:16 2017 - [info] All SSH connection tests passed successfully.
```

检测结果显示都为“OK”，代表 SSH 检测成功。

接着检测主从结构，命令如下：

```
masterha_check_repl --conf=/etc/mha/mha.conf
```

输出显示结果：

```
    Tue Oct 10 10:19:27 2017 - [warning] Global configuration file
/etc/masterha_default.cnf not found. Skipping.
    Tue Oct 10 10:19:27 2017 - [info] Reading application default configuration
from /etc/mha/mha.conf..
    Tue Oct 10 10:19:27 2017 - [info] Reading server configuration from
/etc/mha/mha.conf..
    Tue Oct 10 10:19:27 2017 - [info] MHA::MasterMonitor version 0.57.
    Tue Oct 10 10:19:27 2017 - [info] GTID failover mode = 1
    Tue Oct 10 10:19:27 2017 - [info] Dead Servers:
    Tue Oct 10 10:19:27 2017 - [info] Alive Servers:
    Tue Oct 10 10:19:27 2017 - [info]   192.168.56.100(192.168.56.100:3306)
    Tue Oct 10 10:19:27 2017 - [info]   192.168.56.101(192.168.56.101:3306)
    Tue Oct 10 10:19:27 2017 - [info]   192.168.56.102(192.168.56.102:3306)
    Tue Oct 10 10:19:27 2017 - [info] Alive Slaves:
    Tue Oct 10 10:19:27 2017 - [info]   192.168.56.101(192.168.56.101:3306)
Version=5.7.14-log (oldest major version between slaves) log-bin:enabled
    Tue Oct 10 10:19:27 2017 - [info]    GTID ON
    Tue Oct 10 10:19:27 2017 - [info]    Replicating from
192.168.56.100(192.168.56.100:3306)
    Tue Oct 10 10:19:27 2017 - [info]    Primary candidate for the new Master
(candidate_master is set)
    Tue Oct 10 10:19:27 2017 - [info]  192.168.56.102(192.168.56.102:3306)
Version=5.7.14-log (oldest major version between slaves) log-bin:enabled
    Tue Oct 10 10:19:27 2017 - [info]    GTID ON
```

```
    Tue Oct 10 10:19:27 2017 - [info]    Replicating from
192.168.56.100(192.168.56.100:3306)
    Tue Oct 10 10:19:27 2017 - [info]    Not candidate for the new Master
(no_master is set)
    Tue Oct 10 10:19:27 2017 - [info] Current Alive Master:
192.168.56.100(192.168.56.100:3306)
    Tue Oct 10 10:19:27 2017 - [info] Checking slave configurations..
    Tue Oct 10 10:19:27 2017 - [info]  read_only=1 is not set on slave
192.168.56.101(192.168.56.101:3306).
    Tue Oct 10 10:19:27 2017 - [info]  read_only=1 is not set on slave
192.168.56.102(192.168.56.102:3306).
    Tue Oct 10 10:19:27 2017 - [info] Checking replication filtering settings..
    Tue Oct 10 10:19:27 2017 - [info]  binlog_do_db= , binlog_ignore_db=
    Tue Oct 10 10:19:27 2017 - [info]  Replication filtering check ok.
    Tue Oct 10 10:19:27 2017 - [info] GTID (with auto-pos) is supported. Skipping
all SSH and Node package checking.
    Tue Oct 10 10:19:27 2017 - [info] Checking SSH publickey authentication
settings on the current master..
    Tue Oct 10 10:19:27 2017 - [info] HealthCheck: SSH to 192.168.56.100 is
reachable.
    Tue Oct 10 10:19:27 2017 - [info]
    192.168.56.100(192.168.56.100:3306) (current master)
     +--192.168.56.101(192.168.56.101:3306)
     +--192.168.56.102(192.168.56.102:3306)

    Tue Oct 10 10:19:27 2017 - [info] Checking replication health on
192.168.56.101..
    Tue Oct 10 10:19:27 2017 - [info]  ok.
    Tue Oct 10 10:19:27 2017 - [info] Checking replication health on
192.168.56.102..
    Tue Oct 10 10:19:27 2017 - [info]  ok.
    Tue Oct 10 10:19:27 2017 - [info] Checking master_ip_failover_script status:
    Tue Oct 10 10:19:27 2017 - [info]   /usr/local/scripts/master_ip_failover
--command=status --ssh_user=root --orig_master_host=192.168.56.100
--orig_master_ip=192.168.56.100 --orig_master_port=3306


    IN SCRIPT TEST====/sbin/ifconfig eth0:0 down==/sbin/ifconfig eth0:0
```

```
192.168.56.123/24===

    Checking the Status of the script.. OK
    Tue Oct 10 10:19:27 2017 - [info]  OK.
    Tue Oct 10 10:19:27 2017 - [warning] shutdown_script is not defined.
    Tue Oct 10 10:19:27 2017 - [info] Got exit code 0 (Not master dead).

MySQL Replication Health is OK.
```

显示结果为主从同步状态 OK。

> 注：这一步至关重要，好多报错都是在这一步出现的，不过不用担心，从报错信息中也能找到解决问题的答案。

实验第五步：添加 vip

在主库（192.168.56.100）上执行添加 vip 的过程（第一次手动添加）：

```
ip addr add 192.168.56.123 dev eth0
```

```
[root@node1 ~]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:f1:fd:08 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.100/24 brd 192.168.56.255 scope global eth0
    inet 192.168.56.123/32 scope global eth0
    inet6 fe80::a00:27ff:fef1:fd08/64 scope link
       valid_lft forever preferred_lft forever
```

实验第六步：启动 MHA 服务

在管理节点（192.168.56.102）上执行 MHA 的启动：

```
    nohup masterha_manager --conf=/etc/mha/mha.conf > /tmp/mha_manager.log  <
/dev/null 2>&1 &
```

验证启动成功的命令并查看显示状态：

```
    masterha_check_status --conf=/etc/mha/mha.conf
```

```
[root@node3 ~]# masterha_check_status --conf=/etc/mha/mha.conf
mha (pid:3477) is running(0:PING_OK), master:192.168.56.100
```

MHA 已经正常开始工作了。

之后可以模拟主库故障，查看是否自动切换。

在主库（192.168.56.100）上执行停掉 MySQL 服务操作：

```
mysqladmin -uroot -proot123 shutdown
```

从库 192.168.56.101 自动获取 vip：

```
[root@node2 ~]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:83:b3:9d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global eth0
    inet 192.168.56.123/24 brd 192.168.56.255 scope global secondary eth0:0
    inet6 fe80::a00:27ff:fe83:b39d/64 scope link
       valid_lft forever preferred_lft forever
```

转换为新的主库，192.168.56.102 自动指向新的主库：192.168.56.101

```
root@db 10:46:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.101
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000002
          Read_Master_Log_Pos: 3100
               Relay_Log_File: node3-relay-bin.000002
                Relay_Log_Pos: 423
        Relay_Master_Log_File: mysql-binlog.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

切换后，MHA 进程会自动停止运行。

在管理节点上查看：

```
masterha_check_status --conf=/etc/mha/mha.conf
```

```
[root@node3 ~]#  masterha_check_status --conf=/etc/mha/mha.conf
mha is stopped(2:NOT_RUNNING).
```

恢复操作，把宕掉的主库 192.168.56.100 恢复起来：

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

然后重新指向到新的主库，配置主从：

```
root@db 10:55:  [(none)]> CHANGE MASTER TO MASTER_HOST='192.168.56.101',MASTER_USER
='repl',MASTER_PASSWORD='repl',MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 2 warnings (0.07 sec)

root@db 10:58:  [(none)]> start slave;
Query OK, 0 rows affected (0.03 sec)
```

```
root@db 10:58:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.101
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000002
          Read_Master_Log_Pos: 3100
               Relay_Log_File: node1-relay-bin.000002
                Relay_Log_Pos: 423
        Relay_Master_Log_File: mysql-binlog.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

新的一主两从结构完成。

如果还想把原来的 192.168.56.100 变成主库，就需要手动在线执行切换工作。

在管理节点上执行如下命令：

```
masterha_master_switch  --conf=/etc/mha/mha.conf    --master_state=alive
--new_master_host=192.168.56.100 --orig_master_is_new_slave
```

参数详解。

- --master_state：代表当前主库的状态为 alive。本例中 192.168.56.101 为当前主库。
- --new_master_host：代表切换后的新主库是 192.168.56.100。
- --orig_master_is_new_slave：将原来的主库变为从节点。该参数必须加。

命令输出结果较长，见 www.broadview.com.cn/33679 中下载资源。

结果显示输出成功。

验证切换后的结果。

在 192.168.56.100 上查看 vip 是否切换：

```
[root@node1 ~]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:f1:fd:08 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.100/24 brd 192.168.56.255 scope global eth0
    inet 192.168.56.123/32 scope global eth0
    inet6 fe80::a00:27ff:fef1:fd08/64 scope link
       valid_lft forever preferred_lft forever
```

vip 已经从 56.101 切换到 56.100 上。

在两个从库上查看主从状态，也已经都指向新的主库 192.168.56.100

```
root@db 11:25:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.100
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000009
          Read_Master_Log_Pos: 194
               Relay_Log_File: node2-relay-bin.000002
                Relay_Log_Pos: 373
        Relay_Master_Log_File: mysql-binlog.000009
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

MHA 常用的管理命令。

- masterha_check_repl：检查主从结构是否创建 OK。
- masterha_check_ssh：检查 SSH 连接是否成功。
- masterha_check_status：查看 MHA 进程状态。
- masterha_manager：启动 MHA 服务。
- masterha_master_switch：手动执行在线切换。
- masterha_stop：停止 MHA 的服务。

第 13 章 Keepalived+双主架构

双 master 配合 Keepalived 这种 MySQL 高可用架构设计也是基于主从复制的原理而搭建的。使用 MySQL 双主复制技术+Keepalived 是一种简单、便捷的解决方案，在高可用集群环境中，Keepalived 使用 vip，利用 Keepalived 自带的服务监控功能和自定义脚本来实现 MySQL 故障时自动切换。

13.1 Keepalived 介绍

Keepalived 是基于 VRRP 协议的，VRRP 的全称为 Virtual Redundent Routing Protocol，虚拟冗余路由协议。协议的目的就是为了解决静态路由器单点故障引起的网络失效问题而设计的一套主备协议。Keepalived 利用 VRRP 协议可以实现 MySQL 高可用集群方案，避免单点故障。两台互为主备的 MySQL 服务器运行 Keepalived，master 会向 backup 节点发送广播信号，当 backup 节点接收不到 master 发送的 VRRP 包时，会认为 master 宕机，这时会根据 VRRP 的优先级来选举出一个 backup 来充当 master，则这个 master 就会持有 vip（vip 是对外应用连接的 IP 地址），从而保证了线上现有业务的正常运行，高可用性完美地展现出来。

所谓 VRRP 的优先级就是它会根据优先级来确定其在集群中的地位，用 0～255 来表示，数字越小则表示优先级越低，数值越大则表明优先级越高。VRRP 优先级可选的取值范围为 1 到 254，因为当值为 0 时，代表 master 放弃持有 vip，该值为 255 时，表示当前 master 的优先级最高并持有 vip。另外在 VRRP 中还有一个概念就是 vrid，它是虚拟路由器的标识，同组机器内

的 vrid 必须一致，通常用 0～255 来标识。

详细信息可以查看 Keepalived 的官网地址：

http://keepalived.org/documentation.html。

我们来看一张来自官网的图片，如图 13-1 所示。

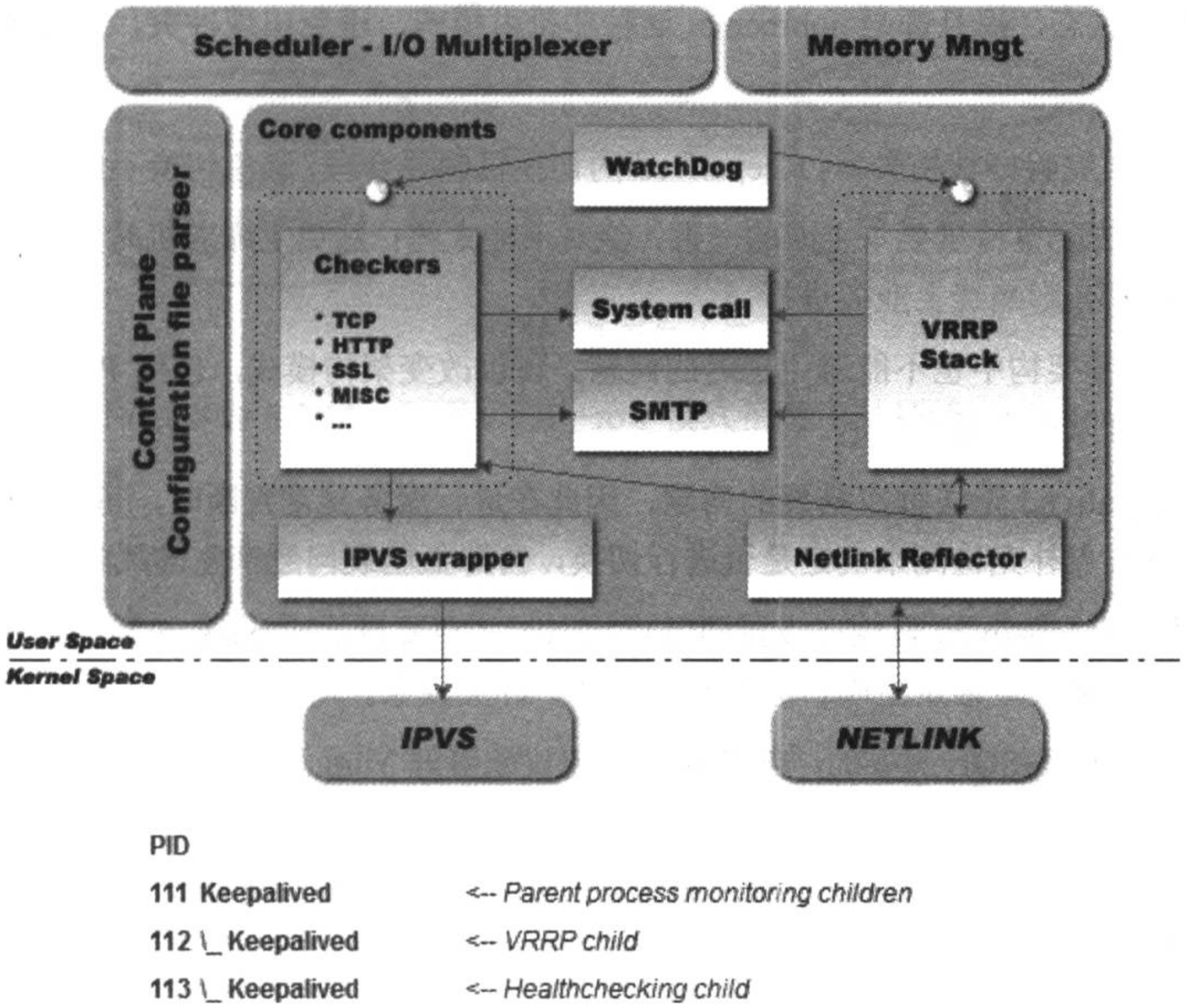


图 13-1　Keepalived 官网图片

从图中我们可以看出，启动完 Keepalived 服务之后，会有 watch dog、vrrp 和 Health-check 三个工作的进程，watch dog 是负责监控 check 和 VRRP 进程的看护者；VRRP 负责当 master 上的服务不可用时通过 VRRP 将其切换到 backup 服务器上；Health-check 是检测服务器的健康状态。

13.2　集群搭建思路及建议

搭建架构之前要理清思路：

（1）需要装有两台 MySQL 的数据库服务器，两者之间互为主从模式，都可读写。其实就只有一台服务器 A 负责数据的写入工作，而另一台服务器 B 作为备用数据库。

（2）安装 Keepalived 的软件包，建议使用 Yum 安装就可以，也很方便。当然我们需要知道 Yum 源的配置方法和安装之后的软件路径在什么位置。

（3）整理好 Keepalived 的配置文件，理清 Keepalived 的三种状态信息。还要准备一个监控 MySQL 的脚本，便于检测到宕机，从而顺利发生切换。

（4）在两台服务器 A 和 B 配置 Keepalived 的参数文件中，要注意两台机器的 state 都要采用 backup 这种状态，而且都是 nopreempt 这种非抢占模式，避免出现冲突，发生脑裂现象。

建议：

（1）一定要完善好切换脚本，Keepalived 的切换机制要合理，避免切换不成功的现象发生。

（2）从库的配置尽量要与主库的一致，绝对不能太差；避免主库宕机时发生切换，新的主库（原来的从库）影响线上业务进行。

（3）在这套架构中也不能避免延迟的问题。可以改变架构模式，使用 PXC 完成实时同步功能，基本上可以实现没有延迟。

（4）Keepalived 无法解决脑裂的问题，因此在进行服务异常判断时，可以修改判断脚本，通过对第三方节点补充检测来决定是否进行切换，可降低脑裂问题产生的风险。

（5）采用 Keepalived 架构在设置两节点状态时，都要设置成 backup 状态而且是不抢占模式，通过优先级来决定谁是主库。避免发生脑裂、冲突现象。

（6）安装好 MySQL 需要的一些依赖包；建议配置好 Yum 源，用 Yum 安装 Keepalived 即可。

一般中小型公司都使用这种架构，搭建比较方便简单；可以采用主从或者主主模式，在 master 节点发生故障后，利用 Keepalived 高可用机制实现快速切换到 slave 节点，原来的从库变成新的主库。

> 注：在云平台环境下，此架构不能搭建。因为 VRRP 协议已经被禁用。

13.3 实验部署演练

集群架构图如图 13-2 所示。

实验部署环境介绍：

```
192.168.56.100 主 masterA
192.168.56.101 备库 masterB
vip: 192.168.56.111
```

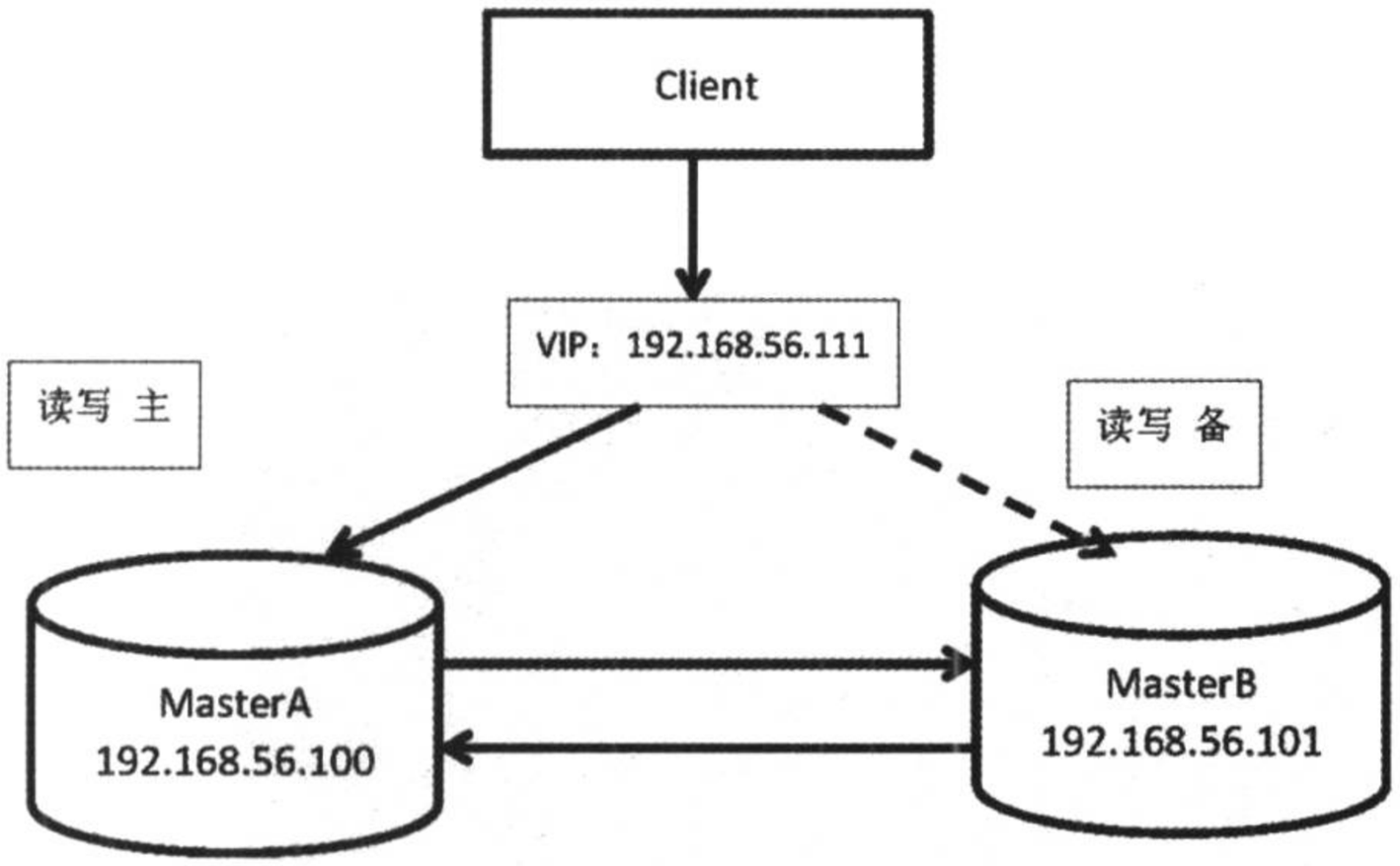


图 13-2　集群架构图

MySQL 数据库版本为 5.7.14，采用 GTID+row 模式搭建主从环境。

搭建注意事项：

- 两台机器的防火墙必须是关闭状态。
- 两台 MySQL 数据库配置文件中 server-id 绝对不能一样，否则会报 1593 这个主从同步的错误，导致搭建不成功。

首先要搭建两台 MySQL 数据库为互为主从的架构模式。这两台服务器都是新机器，没有任何数据。如果主库已经跑了一段时间，需要使用 mysqldump 或者 XtraBackup 工具将数据复制传送到从库。跟之前章节中讲过的操作一致。

第一步：分别在主备库上创建主从同步复制账号。

命令如下：

```
create  user 'bak'@'192.168.56.%' identified by '123456';
grant replication slave on *.* to 'bak'@'192.168.56.%' ;
flush privileges;
```

先在备库 192.168.56.101 上配置同步信息：

```
CHANGE MASTER TO MASTER_HOST='192.168.56.100',MASTER_USER='bak',
MASTER_PASSWORD='123456',master_auto_position=1;
```

打开主从同步开关：

```
start slave;
```

查看主从状态：

```
show slave status\G;
```

```
root@db 12:46:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.100
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000010
          Read_Master_Log_Pos: 1892
               Relay_Log_File: node2-relay-bin.000002
                Relay_Log_Pos: 423
        Relay_Master_Log_File: mysql-binlog.000010
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

从库两个工作的线程 I/O、SQL 都为 yes，代表同步搭建成功

- Slave_IO_Running：Yes。
- Slave_SQL_Running：Yes。

之后再在主库 192.168.56.100 上配置主从同步信息：

```
    CHANGE MASTER TO MASTER_HOST='192.168.56.101',MASTER_USER='bak',
MASTER_PASSWORD='123456',master_auto_position=1;
```

开启主从开关：

```
    start slave;
```

查看主从同步状态：

```
    show slave status
```

```
root@db 12:55:  [(none)]> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.101
                  Master_User: bak
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-binlog.000003
          Read_Master_Log_Pos: 3212
               Relay_Log_File: node1-relay-bin.000002
                Relay_Log_Pos: 1763
        Relay_Master_Log_File: mysql-binlog.000003
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

- Slave_IO_Running：Yes。
- Slave_SQL_Running：Yes。

从库两个工作的线程 I/O、SQL 都为 yes，代表同步搭建成功。

第二步：分别两台机器上安装 Keepalived 的软件包，可以使用 Yum 安装方式。

命令如下：

```
yum -y install keepalived;
```

输入结果：

```
Loaded plugins: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
 * c6-media:
file:///media/CentOS/repodata/repomd.xml: [Errno 14] Could not open/read
file:///media/CentOS/repodata/repomd.xml
Trying other mirror.
file:///media/cdrecorder/repodata/repomd.xml: [Errno 14] Could not
open/read file:///media/cdrecorder/repodata/repomd.xml
Trying other mirror.
c6-media                                                 | 4.0 kB     00:00 ...
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package keepalived.x86_64 0:1.2.7-3.el6 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package         Arch              Version                 Repository      Size
================================================================================
Installing:
 keepalived      x86_64            1.2.7-3.el6             c6-media        174 k

Transaction Summary
================================================================================
Install       1 Package(s)
```

```
Total download size: 174 k
Installed size: 526 k
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : keepalived-1.2.7-3.el6.x86_64                    1/1
  Verifying  : keepalived-1.2.7-3.el6.x86_64                    1/1

Installed:
  keepalived.x86_64 0:1.2.7-3.el6

Complete!
```

在两台机器上分别证明 Keepalived 软件已经存在：

```
[root@node1 ~]# rpm -qa|grep keepalived
keepalived-1.2.7-3.el6.x86_64
```

```
[root@node2 ~]# rpm -qa|grep keepalived
keepalived-1.2.7-3.el6.x86_64
```

第三步：在两台机器上分别配置检测 MySQL 数据库的脚本。

通过判断 MySQL 服务是否宕机，然后停止 Keepalived 服务来进行切换操作。

首先进入 Yum 安装后的软件目录下：

```
cd /etc/keepalived
vim checkmysql.sh
#!/bin/bash
mysqlstr=/usr/local/mysql/bin/mysql
host=192.168.56.100
user=zs
password=123456
port=3306
##MySQL 服务状态正常为 1，否则为 0
mysql_status=1
```

```
    ####check mysql status######
    $mysqlstr -h $host -u $user -ppassword -Pport -e "show status;" >/dev/null
2>&1
    if [ $? = 0 ] ;then
        echo "mysql_status=1"
        exit 0
    else
    /etc/init.d/keepalived stop
    Fi
```

别忘记给脚本赋予执行权限：

```
chmod +x /etc/keepalived/checkMySQL.sh
```

第四步：在两台机器上修改 Keepalived 的配置文件。

在 192.168.56.100 上编辑：

```
    vim /etc/keepalived/keepalived.conf
    ###设置一个脚本来检测 MySQL 的状态，脚本执行间隔设置为 10s 比较合理
    vrrp_script vs_mysql_100 {
        script "/etc/keepalived/checkmysql.sh"
        interval 10
    }
    vrrp_instance VI_100 {  ##集群的名称###
        state BACKUP  ##指定 Keepalived 的角色，master 为主 backup 为备，这里两台机器
都要设置为 backup 角色
        nopreempt ##并且需要设置为不抢占模式
        interface eth0  ##vip 绑定的网卡位置 eth0
        virtual_router_id 100 ##vrid 的值为 100，两台机器必须一致
        priority 100 ##代表优先级，56.100 的优先级比 56.101 的优先级高，56.101 是 90
        advert_int 5 ##主备之间同步检查的时间间隔单位是秒，这里设置为 5s
        authentication {
            auth_type PASS
            auth_pass 1314 ##验证密码，主备密码要保持一致
        }
        track_script {
            vs_mysql_100 ##执行监控的服务
        }
```

```
    virtual_ipaddress {
        192.168.56.111 ##虚拟 IP 地址即 vip
    }
}
```

解释一下脚本中参数的作用：

```
vrrp_instance VI_100 {  ##集群的名称###
    state BACKUP  ##指定 Keepalived 的角色，master 为主 backup 为备，这里两台机器都要设置为 backup 角色
    nopreempt ##并且需要设置为不抢占模式
    interface eth0  ##vip 绑定的网卡位置 eth0
    virtual_router_id 100 ##vrid 的值为 100，两台机器必须一致
    priority 100 ##代表优先级，56.100 的优先级比 56.101 的优先级高，56.101 是 90
    advert_int 5 ##主备之间同步检查的时间间隔单位是秒，这里设置为 5s
    authentication {
        auth_type PASS 验证方式是通过密码
        auth_pass 1314 ##验证密码，主备密码要保持一致
    }
    track_script {
        vs_mysql_100 ##执行监控的服务
    }
    virtual_ipaddress {
        192.168.56.111 ##虚拟 IP 地址即 vip
```

192.168.56.101 上操作：

```
vim /etc/keepalived/keepalived.conf
vrrp_script vs_mysql_101 {
    script "/etc/keepalived/checkmysql.sh"
    interval 10
}
vrrp_instance VI_101 {
    state BACKUP
    nopreempt
    interface eth0
    virtual_router_id 100
    priority 90
    advert_int 5
```

```
    authentication {
        auth_type PASS
        auth_pass 1314
    }
    track_script {
        vs_mysql_101
    }
    virtual_ipaddress {
        192.168.56.111
    }
}
```

总结：可以看到两台机器的 state 都是 backup，并且都是非抢占模式 nopreempt，通过优先级的高低来决定谁是主库（这里 192.168.56.100 是主库）。还要注意，virtual_router_id（虚拟路由 id）要保持一致。

接下来启动两台机器的 Keepalived 进程。

在 192.168.56.100 上执行：

```
/etc/init.d/keepalived start
```

```
[root@node1 keepalived]# /etc/init.d/keepalived start
Starting keepalived:                                       [  OK  ]
```

```
[root@node1 keepalived]# ps -ef|grep keepalived
root      5315     1  0 10:17 ?        00:00:00 /usr/sbin/keepalived -D
root      5317  5315  0 10:17 ?        00:00:00 /usr/sbin/keepalived -D
root      5318  5315  0 10:17 ?        00:00:00 /usr/sbin/keepalived -D
root      5435  3884  0 10:37 pts/1    00:00:00 grep keepalived
```

查看启动日志文件的输出结果：

```
cat /var/log/messages
Oct 14 10:17:02 node1 Keepalived_healthcheckers[5317]: Configuration is
using : 4443 Bytes
Oct 14 10:17:02 node1 Keepalived_vrrp[5318]: VRRP_Script(vs_mysql_100)
succeeded
Oct 14 10:17:02 node1 Keepalived_healthcheckers[5317]: Using LinkWatch
kernel netlink reflector...
Oct 14 10:17:18 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100)
Transition to MASTER STATE
Oct 14 10:17:23 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) Entering
```

```
MASTER STATE
    Oct 14 10:17:23 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) setting
protocol VIPs.
    Oct 14 10:17:23 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) Sending
gratuitous ARPs on eth0 for 192.168.56.111
    Oct 14 10:17:23 node1 Keepalived_healthcheckers[5317]: Netlink reflector
reports IP 192.168.56.111 added
    Oct 14 10:17:28 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) Sending
gratuitous ARPs on eth0 for 192.168.56.111
```

从启动完成之后的日志结果中我们可以分析出，由于 192.168.56.100 机器上的优先级高，Keepalived 的状态已经从 backup 升级为 master，并且发送了一个广播协议，证明 192.168.56.111 已经在本台机器上，其他机器就不要再使用了。

```
[root@node1 keepalived]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:f1:fd:08 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.100/24 brd 192.168.56.255 scope global eth0
    inet 192.168.56.111/32 scope global eth0
    inet6 fe80::a00:27ff:fef1:fd08/64 scope link
       valid_lft forever preferred_lft forever
```

在 192.168.56.101 备库上启动 Keepalived 进程：

```
[root@node2 keepalived]# /etc/init.d/keepalived start
Starting keepalived:                                       [  OK  ]
```

```
[root@node2 keepalived]# ps -ef|grep keepalived
root      4689     1  0 10:38 ?        00:00:00 /usr/sbin/keepalived -D
root      4691  4689  0 10:38 ?        00:00:00 /usr/sbin/keepalived -D
root      4692  4689  0 10:38 ?        00:00:00 /usr/sbin/keepalived -D
root      4698  3177  0 10:38 pts/0    00:00:00 grep keepalived
```

查看启动日志的输出结果：

```
    Oct 14 10:38:15 node2 Keepalived_vrrp[4692]: Configuration is using : 62721
Bytes
    Oct 14 10:38:15 node2 Keepalived_vrrp[4692]: Using LinkWatch kernel netlink
reflector...
    Oct 14 10:38:15 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) Entering
BACKUP STATE
    Oct 14 10:38:15 node2 Keepalived_healthcheckers[4691]: Configuration is
using : 4573 Bytes
```

```
    Oct 14 10:38:15 node2 Keepalived_healthcheckers[4691]: Using LinkWatch
kernel netlink reflector...
    Oct 14 10:38:15 node2 Keepalived_vrrp[4692]: VRRP sockpool: [ifindex(2),
proto(112), fd(10,11)]
    Oct 14 10:38:15 node2 Keepalived_vrrp[4692]: VRRP_Script(vs_mysql_101)
succeeded
```

从日志中分析得出，备库的 Keepalived 的状态就是 backup，时刻准备接管主库的服务，并没有 vip 绑定：

```
[root@node2 keepalived]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:83:b3:9d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global eth0
    inet6 fe80::a00:27ff:fe83:b39d/64 scope link
       valid_lft forever preferred_lft forever
```

测试使用 vip 是否可以连接到主库。

使用另外一台机器进行登录测试，在 192.168.56.102 上使用 vip 连接主库：

```
mysql -uzs -p123456 -h192.168.56.111
```

可正常连接，使用 show status 查看当前数据库状态，明显可以看到是通过 vip192.168.56.111 连接的主库。

```
zs@db 11:16:  [(none)]> \s
--------------
/usr/local/mysql/bin/mysql  Ver 14.14 Distrib 5.7.14, for linux-glibc2.5 (x86_64) u
sing  EditLine wrapper

Connection id:          9
Current database:
Current user:           zs@192.168.56.102
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server version:         5.7.14-log MySQL Community Server (GPL)
Protocol version:       10
Connection:             192.168.56.111 via TCP/IP
```

模拟主库宕机的故障切换。

在主库 192.168.56.100 上执行关闭 MySQL 服务操作：

```
mysqladmin -uroot -proot123 shutdown
```

这时再查看一下 Keepalived 日志的情况：

```
cat /var/log/messages
```

```
Oct 14 10:59:02 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) sending 0 priori
ty
Oct 14 10:59:02 node1 Keepalived_vrrp[5318]: VRRP_Instance(VI_100) removing protoco
l VIPs.
Oct 14 10:59:02 node1 Keepalived[5315]: Stopping Keepalived v1.2.7 (02/21,2013)
```

已经把 vip removed 了，并且停掉了 Keepalived 的服务。

在备库 192.168.56.101 上查看 Keepalived 日志：

```
Oct 14 10:59:02 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) Transition to MA
STER STATE
Oct 14 10:59:07 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) Entering MASTER
STATE
Oct 14 10:59:07 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) setting protocol
 VIPs.
Oct 14 10:59:07 node2 Keepalived_healthcheckers[4691]: Netlink reflector reports IP
 192.168.56.111 added
Oct 14 10:59:07 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) Sending gratuito
us ARPs on eth0 for 192.168.56.111
Oct 14 10:59:12 node2 Keepalived_vrrp[4692]: VRRP_Instance(VI_101) Sending gratuito
us ARPs on eth0 for 192.168.56.111
```

发现备库的 Keepalived 状态已经从 backup 升级为 master，并且 vip 已经成功切换到了备库上。

```
[root@node2 keepalived]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:83:b3:9d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global eth0
    inet 192.168.56.111/32 scope global eth0
    inet6 fe80::a00:27ff:fe83:b39d/64 scope link
       valid_lft forever preferred_lft forever
```

不影响客户端的连接状态，会短时间内出现间断。可继续使用 vip 连接当前主库：

```
mysql -uzs -p123456 -h192.168.56.111
```

```
zs@db 11:19:   [(none)]> \s
--------------
/usr/local/mysql/bin/mysql  Ver 14.14 Distrib 5.7.14, for linux-glibc2.5 (x86_64) u
sing  EditLine wrapper

Connection id:          7
Current database:
Current user:           zs@192.168.56.102
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server version:         5.7.14-log MySQL Community Server (GPL)
Protocol version:       10
Connection:             192.168.56.111 via TCP/IP
```

第 14 章 PXC

本章介绍基于 Galera 协议的 MySQL 高可用集群架构。Galera 产品是以 Galera Cluster 方式为 MySQL 提供高可用集群解决方案的。Galera Cluster 就是集成了 Galera 插件的 MySQL 集群。Galera replication 是 Codership 提供的 MySQL 数据同步方案，具有高可用性，方便扩展，并且可以实现多个 MySQL 节点间的数据同步复制与读写，可保障数据库的服务高可用及数据强一致性。

基于 Galera 的高可用方案主要有 MariaDB Galera Cluster 和 Percona XtraDB Cluster，简称 PXC，目前 PXC 架构在生产环境中用得更多而且更成熟一些。

PXC 属于一套近乎完美的 MySQL 高可用集群解决方案，相比那些比较传统的基于主从复制模式的集群架构 MHA 和 MM+Keepalived，Galera Cluster 最突出特点就是解决了诟病已久的数据复制延迟问题，基本上可以达到实时同步。而且节点与节点之间，它们相互的关系是对等的。本身 Galera Cluster 也是一种多主架构，如图 14-1 所示（该图来自官网）。

要搭建 PXC 架构至少需要三个 MySQL 实例来组成一个集群，三个实例之间不是主从模式，而是各自为主，所以三者是对等关系，不分从属，这就叫 multi-master 架构。客户端写入和读取数据时，连接哪个实例都是一样的。读取到的数据是相同的，写入任意一个实例之后，集群自己会将新写入的数据同步到其他实例上，这种架构不共享任何数据，是一种高冗余架构。

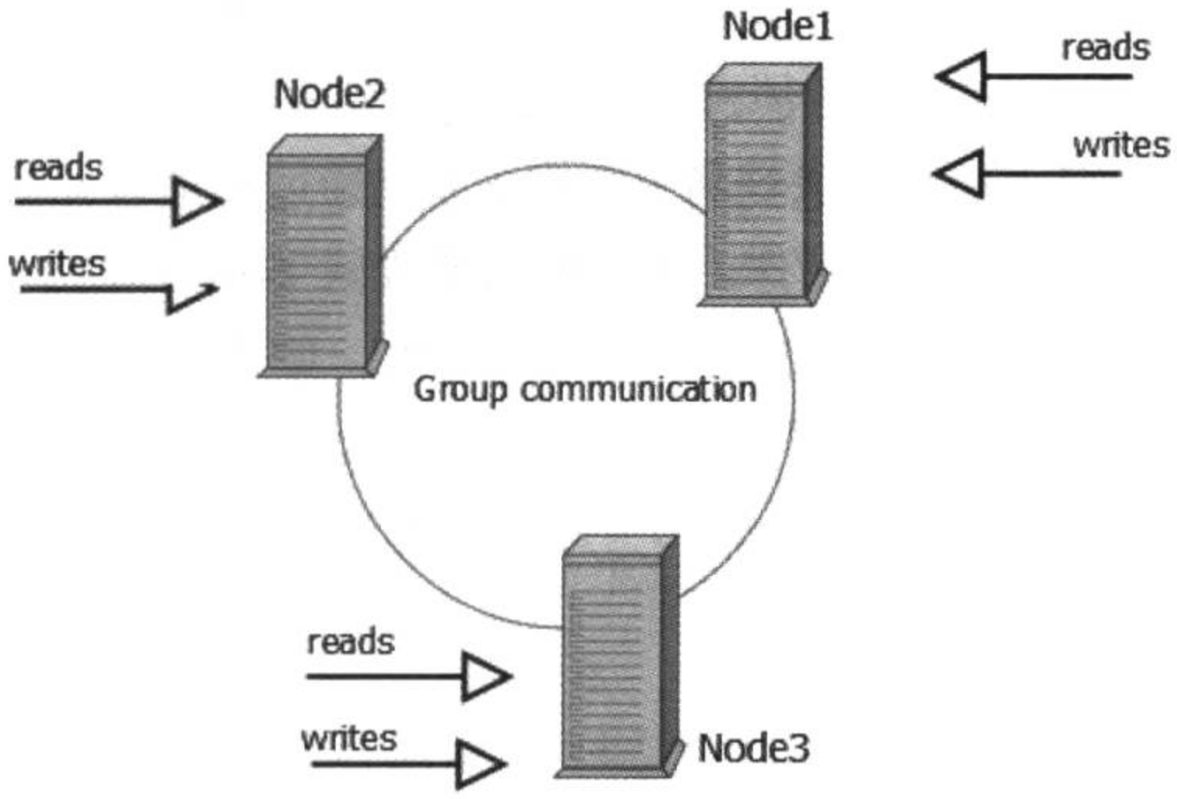


图 14-1　Galera Cluster 架构

14.1　PXC 原理

PXC 最常使用以下 4 个端口号：

- 3306——数据库对外服务的端口号。
- 4444——请求 SST 的端口（注：SST 是指数据库一个备份全量文件的传输）。
- 4567——组成员之间进行沟通的一个端口号。
- 4568——用于传输 IST（注：相对于 SST 来说的一个增量）。

PXC 的原理如图 14-2（该图来自官方手册）。

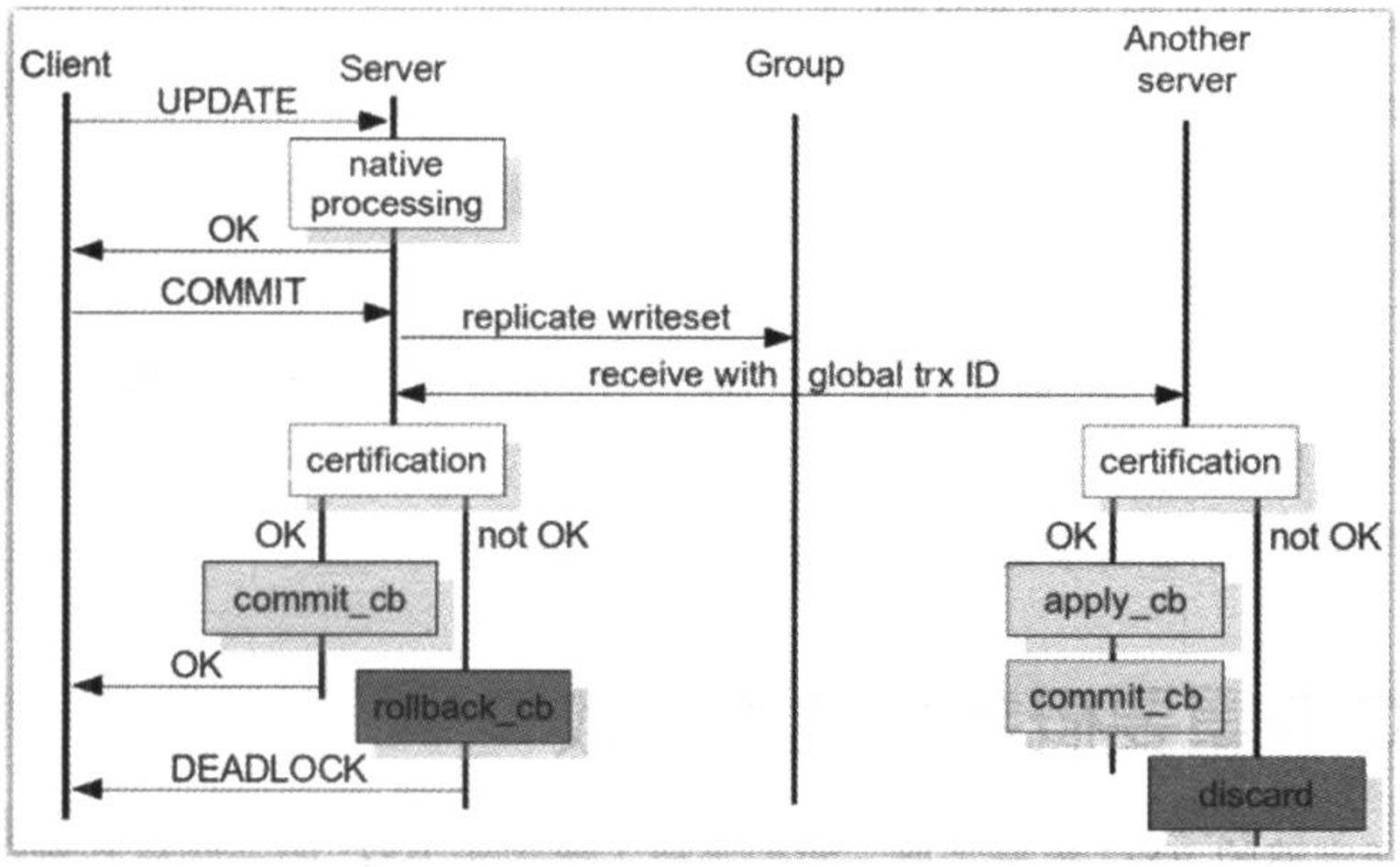


图 14-2　PXC 原理图

从图中可以看出 PXC 的操作流程。首先客户端先发起一个事务，该事务先在本地执行，执行完成之后就要发起对事务的提交操作了。在提交之前需要将产生的复制写集广播出去，然后获取到一个全局的事务 ID 号，一并传送到另一节点上面。通过验证合并数据之后，发现没有冲突数据，执行 apply_cb 和 commit_cb 动作，否则就需要取消（discard）此次事务的操作。而当前 server 节点通过验证之后，执行提交操作，并返回 OK，如果验证没通过，则执行回滚。当然在生产中至少要有三个节点的集群环境，如果其中有一个节点没有验证通过，出现了数据冲突，那么此时采取的方式就是将出现不一致的节点踢出集群环境，而且它自己会执行 shutdown 命令，自动关机。

14.2 PXC 架构的优缺点

优点如下：

（1）实现 MySQL 数据库集群架构的高可用性和数据的强一致性。

（2）完成了真正的多节点读写的集群方案。

（3）改善了传统意义上的主从复制延迟的问题，基本上达到了实时同步。

（4）新加入的节点可以自动部署，无须提供手动备份，维护起来很方便。

（5）由于是多节点写入，所以数据库故障切换很容易。

缺点如下：

（1）新加入的节点开销大，需要复制完整的数据。采用 sst 传输开销太大。

（2）任何更新事务都需要全局验证通过，才会在每个节点库上执行。集群性能受限于性能最差的节点，也就是经常说的短板效应。

（3）因为需要保证数据的一致性，所以在多节点并发写时，锁冲突问题比较严重。

（4）存在写扩大问题，所有的节点上都会发生写操作。

（5）只支持 InnoDB 存储引擎表。

（6）没有表级别的锁定，执行 DDL 语句操作会把整个集群锁住，而且也“kill”不了（注：建议使用 osc 操作）。

（7）所有的表必须含有主键，不然操作数据时会报错。

14.3 PXC 中重要概念和重点参数

在搭建一套完整的 PXC 集群之前，我们需要了解关于 PXC 的一些核心参数和重要概念，

便于后期集群的维护。首先要规范集群中节点的数量，整个集群中节点数控制在最少 3 个、最大 8 个的范围内。最少 3 个节点是为了防止出现脑裂现象，因为只有在两个节点下才会出现此现象。脑裂现象的标志就是输入任何命令，返回结果都是 unkown command。节点在集群中，会因新节点的加入或者故障，同步失效等而发生状态的切换。

节点状态变化阶段：

- open——节点启动成功，尝试连接到集群。
- primary——节点已处于集群中，在新节点加入时，选取 donor 进行数据同步时会产生的状态。
- joiner——节点处于等待接收同步文件时的状态。
- joined——节点完成数据同步的工作，尝试保持和集群进度一致。
- synced——节点正常提供服务的状态，表示已经同步完成并和集群进度保持一致。
- doner——节点处于为新加入的节点提供全量数据时的状态。

> 注：doner 节点就是数据的贡献者，如果一个新节点加入集群，此时又需要大量数据的 SST 传输，就有可能因此而拖垮整个集群的性能。所以在生产环境中，如果数据量小，还可以使用 SST 全量传输，但如果数据量很大就不建议使用这种方式了。可以考虑先建立主从关系，再加入集群。

PXC 有两种节点的数据传输方式，一种叫 SST 全量传输，另一种叫 IST 增量传输。SST 传输有 xtrabackup、mysqldump 和 rsync 三种方法，而增量传输就一种方法就是 xtrabackup。但生产环境中一般数据量不大的时候，可以使用 SST 全量传输，但也只使用 xtrabackup 方法。

搭建 PXC 集群过程中，需要在 PXC 配置文件中设置好如下参数；

```
wsrep_cluster_name  ##标识该集群的名字
wsrep_cluster_address=gcomm: ##列出集群中的成员
wsrep_node_address ##当前节点的 IP 地址
wsrep_provider=/usr/local/mysql/lib/libgalera_smm.so
##这个参数指定 Galera 库的路径和文件名
wsrep_sst_method=xtrabackup-v2  ##传输数据的方法，现在版本中都使用 xtrabackup
v2 方式
wsrep_sst_auth=sst:zs ##节点的数据库用户的账号和密码
```

在 PXC 中还有一个特别重要的模块就是 GCache。它的核心功能就是每个节点缓存当前最新的写集。如果有新节点加入进来，就可以把新数据的增量传递给新节点，而不需再使用 SST

方式了。这样可以让节点更快地加入集群中。涉及如下参数：

- gcache.size 代表用来缓存写集增量信息的大小。它的默认大小是 128MB，通过 wsrep_provider_options 参数设置。建议调整为 2GB～4GB 范围，足够的空间便于缓存更多的增量信息。
- gcache.mem_size 代表 Gcache 中内存缓存的大小，适度调大可以提高整个集群的性能。
- gcache.page_size 可以理解为如果内存不够用（Gcache 不足），就直接将写集写入磁盘文件中。

14.4 PXC 架构搭建实战

环境介绍如表 14-1 所示.

表 14-1 环境介绍

192.168.56.100	Node1	Master1
192.168.56.101	Node2	Master2
192.168.56.102	Node3	Master3

安装之前的注意事项：

首先要保证三台机器的防火墙 iptables、selinux 都要关闭，三台机器的 server-id 不能一样。

使用的是 PXC 5.7.14 版本。

PXC 软件包的下载地址：

https://www.percona.com/downloads/Percona-XtraDB-Cluster-LATEST/。

接下来三台机器上都需要针对以下基础软件包进行安装，使用 Yum 安装即可，解决依赖性。

```
perl-IO-Socket-SSL.noarch
perl-DBD-MySQL.x86_64
perl-Time-HiRes
openssl
openssl-devel
socat
```

当然还需要使用 XtraBackup 的 SST 传输方式；所以也要安装 percona-xtrabackup。

软件包下载地址：

https://www.percona.com/downloads/XtraBackup/LATEST/。

在节点一（192.168.56.100）上执行操作。

先解压软件包并赋予权限：

```
cd /usr/local/
tar -zxvf Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101.tar.gz
ln -s Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101 mysql
chown mysql:mysql -R mysql
mkdir -p /data/mysql
chown mysql:mysql -R /data/mysql
```

配置 PXC 的参数文件。

注意事项：binlog 的格式必须 row。

```
#PXC
default_storage_engine=Innodb
innodb_autoinc_lock_mode=2
wsrep_cluster_name=pxc_zs ----------------集群的名字
wsrep_cluster_address=gcomm://192.168.56.100,192.168.56.101,192.168.56.1
02（集群中节点的 IP）
wsrep_node_address=192.168.56.100------当前机器的 IP 地址
wsrep_provider=/usr/local/mysql/lib/libgalera_smm.so
wsrep_provider_options="gcache.size=1G"
wsrep_sst_method=xtrabackup-v2 （SST 传输方法）
wsrep_sst_auth=sst:zs （账号权限）
```

然后就可以启动第一个节点了。

首先初始化数据，命令如下：

```
cd /usr/local/mysql/bin
./mysqld --defaults-file=/etc/my.cnf --basedir=/usr/local/mysql --datadir=
/data/mysql/  --user=mysql  --initialize
```

启动第一个节点服务，命令如下：

```
cd /usr/local/mysql/support-files
cp mysql.server /etc/init.d/mysql、
/etc/init.d/mysql bootstrap-pxc
```

```
[root@node1 support-files]# /etc/init.d/mysql bootstrap-pxc
Bootstrapping PXC (Percona XtraDB Cluster)Starting MySQL (P[  OK  ]traDB Cluster)..
```

启动成功之后，创建超管用户和 PXC SST 传输账号：

```
create user 'zs'@'192.168.56.%' identified by 'zs';
grant all privileges on *.* to 'zs'@'192.168.56.%';
create user 'sst'@'localhost' identified by 'zs';
grant all privileges on *.* to 'sst'@'localhost';
flush privileges;
```

在节点二（192.168.56.101）上执行操作：

```
cd /usr/local/
tar -zxvf Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101.tar.gz
ln -s Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101 mysql
chown mysql:mysql -R mysql
mkdir -p /data/mysql
chown mysql:mysql -R /data/mysql
```

编辑 PXC 的配置文件：

```
#pxc
default_storage_engine=Innodb
innodb_autoinc_lock_mode=2
wsrep_cluster_name=pxc_zs ----------------集群的名字
wsrep_cluster_address=gcomm://192.168.56.100,192.168.56.101,192.168.56.1
02（集群中节点的 IP）
wsrep_node_address=192.168.56.101------当前机器的 IP 地址
wsrep_provider=/usr/local/mysql/lib/libgalera_smm.so
wsrep_provider_options="gcache.size=1G"
wsrep_sst_method=xtrabackup-v2 （SST 传输方法）
wsrep_sst_auth=sst:zs （账号权限）
```

然后就可以启动第二个节点了。

首先初始化数据，命令如下：

```
cd /usr/local/mysql/bin
./mysqld --defaults-file=/etc/my.cnf --basedir=/usr/local/mysql
--datadir=/data/mysql/  --user=mysql  --initialize
```

启动第二个节点服务，命令如下：

```
/etc/init.d/mysql start
```

```
[root@node2 bin]# /etc/init.d/mysql start
MySQL (Percona XtraDB Cluster) is not running, but lock fi[FAILED]lock/subsys/mysql) exists
Starting MySQL (Percona XtraDB Cluster)....State transfer in progress, setting sleep higher
.....                                                      [  OK  ]
```

在第三个节点（192.168.56.102）上执行以下操作：

```
cd /usr/local/
tar -zxvf Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101.tar.gz
ln -s Percona-XtraDB-Cluster-5.7.14-rel8-26.17.1.Linux.x86_64.ssl101 mysql
chown mysql:mysql -R mysql
mkdir -p /data/mysql
chown mysql:mysql -R /data/mysql
```

编辑 PXC 的配置文件：

```
#pxc
default_storage_engine=Innodb
innodb_autoinc_lock_mode=2
wsrep_cluster_name=pxc_zs ----------------集群的名字
wsrep_cluster_address=gcomm://192.168.56.100,192.168.56.101,192.168.56.1
02（集群中节点的 IP）
wsrep_node_address=192.168.56.102------当前机器的 IP 地址
wsrep_provider=/usr/local/mysql/lib/libgalera_smm.so
wsrep_provider_options="gcache.size=1G"
wsrep_sst_method=xtrabackup-v2 （SST 传输方法）
wsrep_sst_auth=sst:zs （账号权限）
```

然后就可以启动第三个节点了。

首先初始化数据，命令如下：

```
cd /usr/local/mysql/bin
./mysqld --defaults-file=/etc/my.cnf --basedir=/usr/local/mysql
--datadir=/data/mysql/  --user=mysql  --initialize
```

启动第三个节点服务，命令如下：

```
/etc/init.d/mysql start
```

```
[root@node3 support-files]# /etc/init.d/mysql start
MySQL (Percona XtraDB Cluster) is not running, but lock fil[FAILED]lock/subsys/mysql) exi
Stale sst_in_progress file in datadir
Starting MySQL (Percona XtraDB Cluster)State transfer in progress, setting sleep higher
.....                                                      [  OK  ]
```

> 注：节点 2 和节点 3 的 PXC 启动方式不是/etc/init.d/mysql bootstrap-pxc，而换成了/etc/init.d/mysql start 方式。只有节点 1 才使用/etc/init.d/mysql bootstrap-pxc 方式启动。

至此，三个节点都已经启动成功，这样在任意一个节点上执行一条 DML 语句的操作，都会同步到另外两个节点上。

```
node1:
insert into tt (name,score) values ('aa','80');
```

```
mysql> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  2 | aa   |    80 |
+----+------+-------+
1 row in set (0.00 sec)
```

node2、node3 都可以查到该条记录：

```
mysql> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  2 | aa   |    80 |
+----+------+-------+
1 row in set (0.00 sec)
```

14.5 PXC 集群状态的监控

集群搭建成功之后，我们还要做好相关的监控工作，便于后期处理问题。

使用 show global status like 'wsrep%'命令来查看集群中的参数状态。下面列出几个重要的参数，便于诊断问题。

- wsrep_cluster_state_uuid：集群中所有的节点值应该是相同的，如果有不同值的节点，说明其没有连接入集群。

- wsrep_cluster_size：如果这个值跟预期的节点数一致，则所有的集群节点已经连接。
- wsrep_cluster_status：集群组成的状态。如果不为“Primary”，说明出现“分区”或“脑裂”现象。
- wsrep_local_state：值为 4 表示正常。节点状态有如下 4 个值。
  - joining——表示节点正在加入集群；
  - donor——当前节点是数据奉献者，正在为新加入的节点同步数据；
  - joined——当前节点已经成功加入集群；
  - synced——表示当前节点与整个集群是同步状态。
- wsrep_last_committed：最后提交的事务数目。
- wsrep_ready：值为 ON 表示当前节点可以正常服务，如果值为 OFF，则该节点可能发生脑裂或者网络问题。

14.6　从节点在线转化为 PXC 节点

之前的章节中讲过，将一个新节点加入 PXC 集群，还需要 SST 全量备份传输，这样就很有可能整体拉垮集群性能。所以我们可以考虑让即将加入集群中的节点成为 PXC 集群中某个节点的从库节点，在线快速通过 IST 方式加入集群中。这样的操作可给 DBA 节省出了大量的休息时间。接下来我们看看具体如何操作。

环境介绍

目前 PXC 集群中有两个节点。

- node1：192.168.56.100；
- node2：192.168.56.101。

node3：192.168.56.102 为即将新加入到集群中的节点。上面已经装好 PXC 的软件，服务已经启动。

首先我们需要先让 node3 节点成为 node2 的从节点。搭建主从架构操作如下所述。

在 node2 上先创建主从同步账号：

```
create user 'bak'@'192.168.56.%' identified by '123456';
grant replication slave on *.* to 'bak'@'192.168.56.%';
flush privileges;
```

复制 node2 的数据，传递到 node3 上，执行恢复操作：

```
    /usr/local/mysql/bin/mysqldump --single-transaction -uroot -proot123 -A
--master-data=2 > all.sql
    scp all.sql 192.168.56.102:/root/
```

在 node3 上执行恢复数据的操作：

```
    mysql -uroot -proot123 < all.sql
```

在 node3 上执行配置主从并开启主从同步命令：

```
    CHANGE MASTER TO MASTER_HOST='192.168.56.101',MASTER_USER='bak',MASTER_
PASSWORD='123456',MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=1794;
    start slave;
    mysql> show slave status\G;
    *************************** 1. row ***************************
                Slave_IO_State: Waiting for master to send event
                   Master_Host: 192.168.56.101
                   Master_User: bak
                   Master_Port: 3306
                 Connect_Retry: 60
               Master_Log_File: mysql-bin.000001
           Read_Master_Log_Pos: 2056
                Relay_Log_File: node3-relay-bin.000004
                 Relay_Log_Pos: 582
         Relay_Master_Log_File: mysql-bin.000001
              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: 2056
              Relay_Log_Space: 789
              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: 1013306
                  Master_UUID: 1dfdf5cd-b3b3-11e7-aed9-08002783b39d
             Master_Info_File: /data/mysql/master.info
                    SQL_Delay: 0
          SQL_Remaining_Delay: NULL
      Slave_SQL_Running_State: Slave has read all relay log; waiting for more
updates
           Master_Retry_Count: 86400
                  Master_Bind:
      Last_IO_Error_Timestamp:
     Last_SQL_Error_Timestamp:
               Master_SSL_Crl:
           Master_SSL_Crlpath:
           Retrieved_Gtid_Set:
            Executed_Gtid_Set:
                Auto_Position: 0
         Replicate_Rewrite_DB:
                 Channel_Name:
           Master_TLS_Version:
1 row in set (0.00 sec)
```

可见当前从库同步到主库的位置是二进制文件 mysql-bin.000001，position 号是 2056。

接下来关闭 node3 上的 MySQL 服务：

```
/etc/init.d/mysql stop
```

```
[root@node3 mysql]# /etc/init.d/mysql stop
Shutting down MySQL (Percona XtraDB Cluster)..                [  OK  ]
```

然后把 PXC 的相关配置参数加入到/etc/my.cnf 下：

```
#pxc
default_storage_engine=Innodb
innodb_autoinc_lock_mode=2

wsrep_cluster_name=pxc_zs
wsrep_cluster_address=gcomm://192.168.56.100,192.168.56.101,192.168.56.102
wsrep_node_address=192.168.56.102
wsrep_provider=/usr/local/mysql/lib/libgalera_smm.so
wsrep_provider_options="gcache.size=1G"
wsrep_sst_method=xtrabackup-v2
wsrep_sst_auth=sst:zs
```

在 node2 节点上确认 PXC 需要同步的位置。

执行如下命令：

```
/usr/local/mysql/bin/mysqlbinlog -v -v mysql-bin.000001 |grep Xid
```

```
[root@node2 mysql]# /usr/local/mysql/bin/mysqlbinlog -v -v mysql-bin.000001 |grep Xid
#171018 13:14:01 server id 1003306  end_log_pos 1181 CRC32 0x0394ffec   Xid = 12
#171018 15:12:48 server id 1013306  end_log_pos 2056 CRC32 0x022b0e9f   Xid = 16
```

确认 position 2056 对应的 Xid 为 16。

在继续往下操作之前，这里先介绍一下 PXC 集群中的一个很重要的 grastate.dat 文件。

查看文件中的内容：

```
[root@node2 mysql]# cat grastate.dat
# GALERA saved state
version: 2.1
uuid:    3d6362eb-b3b1-11e7-a264-06dfc8e21616
seqno:   -1
cert_index:
```

文件中的 uuid 就是集群中的 wsrep_cluster_state_uuid。seqno 是集群中 wsrep_last_committed 的值，根据这个值可以直接判断下次节点启动时做增量传输的位置。本例中是 node2 节点的 grastate.dat 文件，node2 是正在运行的状态，所以 seqno=-1。

> 注：如果是非正常关闭的状态，值也是-1。

接下来将 node2 上的 grastate.dat 复制到 node3 对应的目录下。

命令如下：

```
scp grastate.dat 192.168.56.102:/data/mysql/
```

最后在 node3 节点上设置同步开始位置：

```
cd /data/mysql
```

编辑 grastate.dat 文件，修改 seqno 的值，将 seqno: -1 改成刚才获取的 position 2056 对应的 Xid 的值 16，这个就是 PXC 同步开始的位置。

```
[root@node3 mysql]# vim grastate.dat

# GALERA saved state
version: 2.1
uuid:    3d6362eb-b3b1-11e7-a264-06dfc8e21616
seqno:   16
cert_index:
```

> 注：别忘记给传来的文件赋予 MySQL 权限，否则启动后会报错误。

```
chown mysql:mysql grastate.dat
```

在 node3 节点上启动 PXC 服务：

```
/etc/init.d/mysql start
```

```
[root@node3 mysql]# /etc/init.d/mysql start
Starting MySQL (Percona XtraDB Cluster)..                  [  OK  ]
```

检测新加入节点的状态：

```
show global status like '%ws%';
```

wsrep_local_state	4
wsrep_local_state_comment	Synced
wsrep_cert_index_size	0
wsrep_cert_bucket_count	22
wsrep_gcache_pool_size	1320
wsrep_causal_reads	0
wsrep_cert_interval	0.000000
wsrep_incoming_addresses	192.168.56.101:3306,192.168.56.100:3306,192.168.56.102:3306

```
wsrep_cluster_conf_id          | 13
wsrep_cluster_size             | 3
wsrep_cluster_state_uuid       | 3d6362eb-b3b1-11e7-a264-06dfc8e21616
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         | 3.17(r)
wsrep_ready                    | ON
```

证明新加入的节点已经正常工作了。

第 15 章

ProxySQL

ProxySQL 是 MySQL 的一款中间件的产品，是灵活强大的 MySQL 代理层，可以实现读写分离，支持 Query 路由功能，支持动态指定某个 SQL 进行缓存，支持动态加载配置、故障切换和一些 SQL 的过滤功能。还有一些同类产品，比如 DBproxy、MyCAT、OneProxy 等。但经过反复对比和测试之后，决定给大家介绍一款性能不错的 MySQL 中间件产品 ProxySQL。

有关 ProxySQL 更多的详细信息可访问：

https://github.com/sysown/proxysql/wiki。

接下来通过实战来全面了解一下 ProxySQL 的特性和使用场景，先介绍一下环境，我们的系统是 CentOS 6.7，MySQL 版本是 5.7.14，准备一主两从架构来配合 ProxySQL，如表 15-1 所示。

表 15-1　环境配置

192.168.56.100	Master（node1）	server-id：3306100
192.168.56.101	Slave1（node2）	server-id：3306101
192.168.56.102	Slave2（node3）	server-id：3306102
192.168.56.103	Proxysql 中间件	server-id：3306103

注：两个从库都要开启 read_only=on。

实验架构如图 15-1 所示。

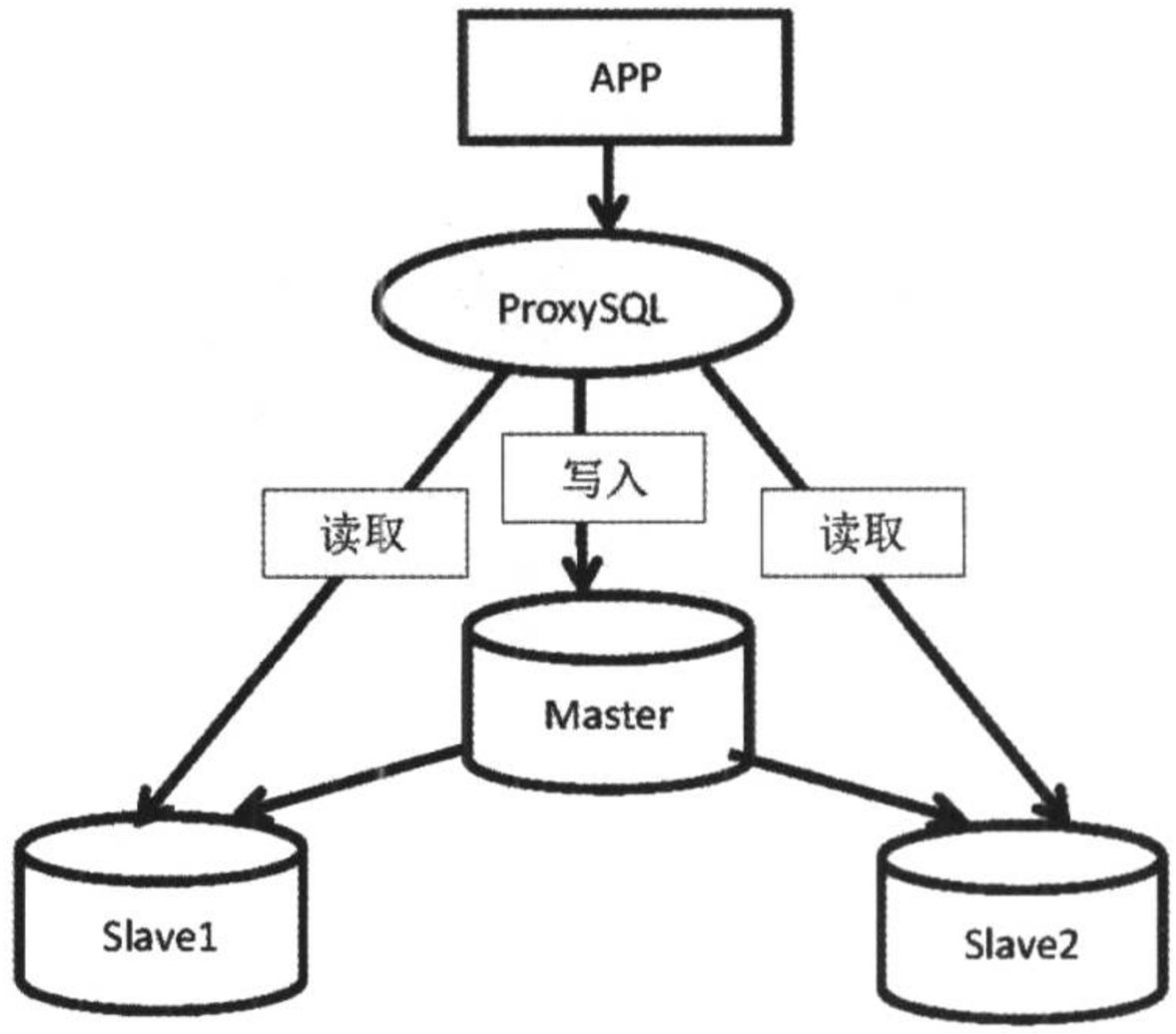


图 15-1　实验架构

15.1　ProxySQL 的安装与启动

首先要安装一些依赖的软件包，配置好 Yum 源进行安装即可。

在 192.168.56.103 上执行如下操作：

```
yum -y install perl-DBD-MySQL
yum -y install perl-DBI
yum -y install perl-Time-HiRes
yum -y install perl-IO-Socket-SSL
```

ProxySQL 软件包的两个下载地址。

- GitHub 官网：https://github.com/sysown/proxysql/releases。
- Percona 官网：https://www.percona.com/downloads/proxysql/。

安装 ProxySQL：

```
rpm -ivh proxysql-1.3.9-1-centos67.x86_64.rpm
```

配置文件路径为/etc/proxysql.cnf。

启动 ProxySQL：

```
service proxysql start
```

```
[root@proxysql ~]# service proxysql start
Starting ProxySQL: DONE!
```

```
[root@proxysql ~]# netstat -anlp|grep proxysql
tcp        0      0 127.0.0.1:6032          0.0.0.0:*               LISTEN      2
9758/proxysql
tcp        0      0 0.0.0.0:6033            0.0.0.0:*               LISTEN      2
9758/proxysql
unix  2      [ ACC ]     STREAM     LISTENING     40193  29758/proxysql       /tmp/proxysq
l_admin.sock
unix  2      [ ACC ]     STREAM     LISTENING     40189  29758/proxysql       /tmp/proxysq
l.sock
```

注：6032 是 ProxySQL 的管理端口号，6033 是对外服务的端口号。

用户名和密码都是默认的 admin。

关闭 ProxySQL：

```
service proxysql stop
```

查看安装版本：

```
[root@proxysql ~]# proxysql --version
ProxySQL version 1.3.9-0-gd09ce23, codename Truls
```

管理员登录命令：

```
/usr/local/mysql/bin/mysql -uadmin -padmin -h 127.0.0.1 -P 6032
```

```
admin@db 10:41:  [(none)]> show databases;
+-----+---------+-------------------------------+
| seq | name    | file                          |
+-----+---------+-------------------------------+
| 0   | main    |                               |
| 2   | disk    | /var/lib/proxysql/proxysql.db |
| 3   | stats   |                               |
| 4   | monitor |                               |
+-----+---------+-------------------------------+
4 rows in set (0.00 sec)
```

可见有四个库：main、disk、stats 和 monitor。分别说明一下这四个库的作用。

main：内存配置数据库，即 MEMORY，表里存放后端 db 实例、用户验证、路由规则等信息。main 库中有如下信息：

```
admin@db 10:47:  [main]> show tables;
+-----------------------------------+
| tables                            |
+-----------------------------------+
| global_variables                  |
| mysql_collations                  |
| mysql_query_rules                 |
| mysql_replication_hostgroups      |
| mysql_servers                     |
| mysql_users                       |
| runtime_global_variables          |
| runtime_mysql_query_rules         |
| runtime_mysql_replication_hostgroups |
| runtime_mysql_servers             |
| runtime_mysql_users               |
| runtime_scheduler                 |
| scheduler                         |
+-----------------------------------+
13 rows in set (0.00 sec)
```

库下的主要表：

- mysql_servers——后端可以连接 MySQL 服务器的列表。
- mysql_users——配置后端数据库的账号和监控的账号。
- mysql_query_rules——指定 Query 路由到后端不同服务器的规则列表。

> 注：表名以 runtime_开头的表示 ProxySQL 当前运行的配置内容，不能通过 DML 语句修改。只能修改对应的不以 runtime 开头的表，然后“LOAD”使其生效，“SAVE”使其存到硬盘以供下次重启加载。

disk 库：持久化磁盘的配置。

stats 库：统计信息的汇总。

monitor 库：一些监控的收集信息，包括数据库的健康状态等。

15.2 配置 ProxySQL 监控

首先在 master（192.168.56.100）上创建 ProxySQL 的监控账户和对外访问账户并赋予权限。命令如下：

```
create user 'monitor'@'192.168.56.%' identified by 'monitor';
grant all privileges on *.* to 'monitor'@'192.168.56.%' with grant option;
create user 'zs'@'192.168.56.%' identified by 'zs';
grant all privileges on *.* to 'zs'@'192.168.56.%' with grant option;
```

```
flush privileges;
```

15.3　ProxySQL 的多层配置系统

ProxySQL 有一套很完整的配置系统，方便 DBA 对线上的操作。整套配置系统分为三层，顶层为 RUNTIME，中间层为 MEMORY，底层也就是持久层为 DISK 和 CONFIG FILE。

配置结构如图 15-2 所示。

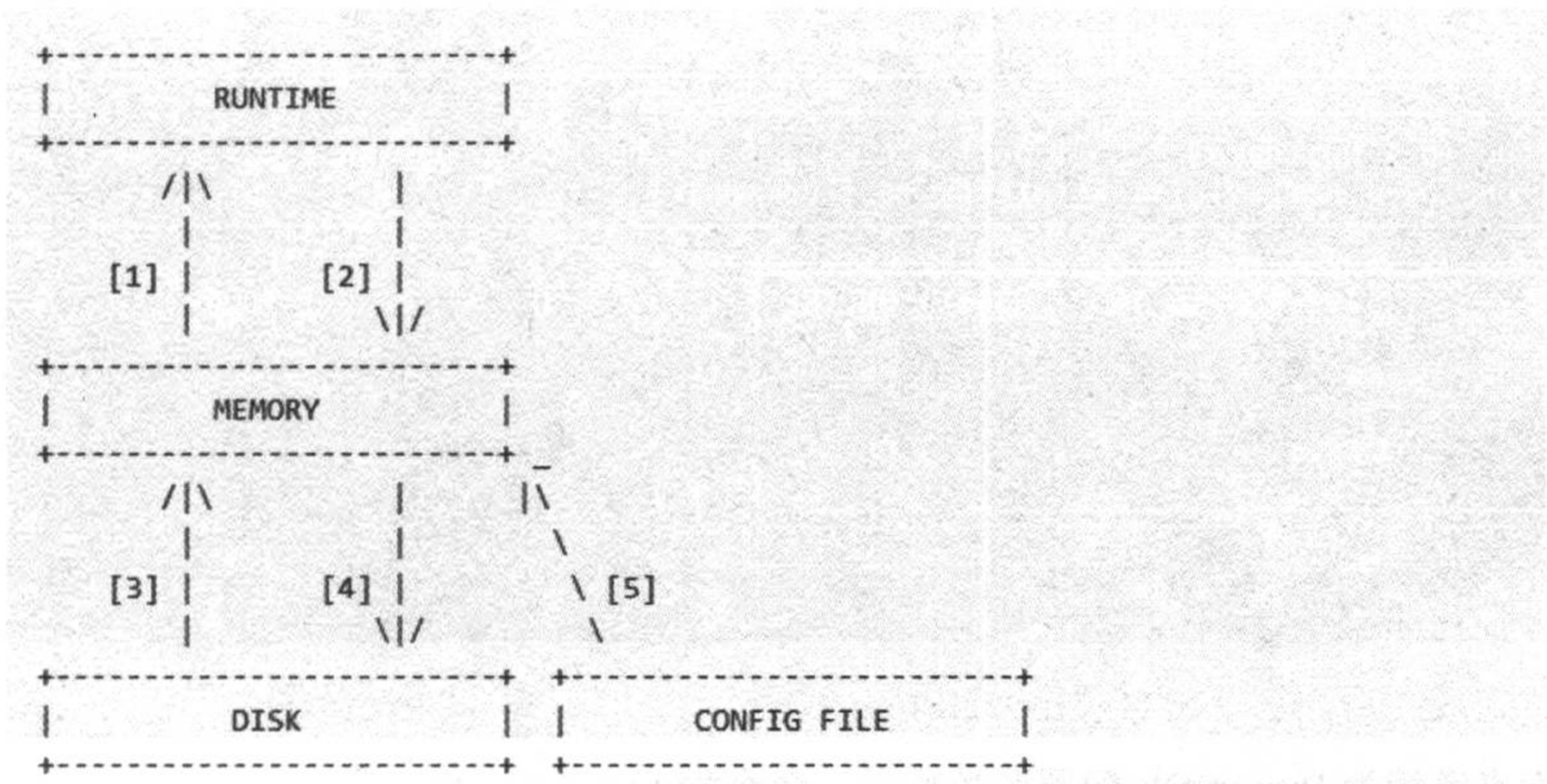


图 15-2　配置结构（图来自 https://github.com/sysown/proxysql/wiki/Multi-layer-configuration-system）

- RUNTIME：代表 ProxySQL 当前生效的正在使用的配置，无法直接修改这里的配置，必须要从下一层“load”进来。
- MEMORY：MEMORY 层上面连接 RUNTIME 层，下面连接持久化层。在这层可以正常操作 ProxySQL 配置，随便修改，不会影响生产环境。修改一个配置一般都是先在 MEMORY 层完成的，确认正常之后再加载到 RUNTIME 和持久化到磁盘上。
- DISK 和 CONFIG FILE：持久化配置信息，重启后内存中的配置信息会丢失，所以需要将配置信息保留在磁盘中。重启时，可以从磁盘快速加载回来。

介绍完这三层配置系统之后，用超管用户登录 ProxySQL 来添加主从服务器列表。

命令如下：

```
insert into mysql_servers(hostgroup_id,hostname,port) values
(10,'192.168.56.100',3306);
```

```
    insert into mysql_servers(hostgroup_id,hostname,port) values
(10,'192.168.56.101',3306);
    insert into mysql_servers(hostgroup_id,hostname,port) values
(10,'192.168.56.102',3306);
    load mysql servers to runtime;
    save mysql servers to disk;
```

登录 ProxySQL 之后，凡是进行任何操作，都需要运行 load **to runtime，从 memory 加载到 runtime，再执行 save ** to disk 持久化到磁盘。

```
admin@db 15:22:  [(none)]> select * from mysql_servers;
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+-----
---------------+---------+----------------+---------+
| hostgroup_id | hostname       | port | status | weight | compression | max_connections | max_
replication_lag | use_ssl | max_latency_ms | comment |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+-----
---------------+---------+----------------+---------+
| 10           | 192.168.56.100 | 3306 | ONLINE | 1      | 0           | 1000            | 0
               | 0       | 0              |         |
| 10           | 192.168.56.101 | 3306 | ONLINE | 1      | 0           | 1000            | 0
               | 0       | 0              |         |
| 10           | 192.168.56.102 | 3306 | ONLINE | 1      | 0           | 1000            | 0
               | 0       | 0              |         |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+-----
---------------+---------+----------------+---------+
3 rows in set (0.00 sec)
```

加载完成之后，三台机器都是 ONLINE 状态。

接下来继续为 ProxySQL 配置监控账号，命令如下：

```
set mysql-monitor_username='monitor';
set mysql-monitor_password='monitor';
load mysql variables to runtime;
save mysql variables to disk;
```

之后验证监控信息：

```
admin@db 15:22:  [(none)]> select * from monitor.mysql_server_connect_log order by time_start_u
s desc limit 6;
+----------------+------+------------------+-------------------------+---------------+
| hostname       | port | time_start_us    | connect_success_time_us | connect_error |
+----------------+------+------------------+-------------------------+---------------+
| 192.168.56.102 | 3306 | 1508484232894194 | 1604                    | NULL          |
| 192.168.56.101 | 3306 | 1508484232883109 | 1378                    | NULL          |
| 192.168.56.100 | 3306 | 1508484232871933 | 1390                    | NULL          |
| 192.168.56.102 | 3306 | 1508484172892776 | 1602                    | NULL          |
| 192.168.56.101 | 3306 | 1508484172882580 | 1480                    | NULL          |
| 192.168.56.100 | 3306 | 1508484172871707 | 1339                    | NULL          |
+----------------+------+------------------+-------------------------+---------------+
6 rows in set (0.01 sec)
```

```
admin@db 15:24:  [(none)]> select * from monitor.mysql_server_ping_log order by time_start_us d
esc limit 6;
+---------------+------+------------------+----------------------+------------+
| hostname      | port | time_start_us    | ping_success_time_us | ping_error |
+---------------+------+------------------+----------------------+------------+
| 192.168.56.102 | 3306 | 1508484273284892 | 498                  | NULL       |
| 192.168.56.101 | 3306 | 1508484273282976 | 499                  | NULL       |
| 192.168.56.100 | 3306 | 1508484273281073 | 709                  | NULL       |
| 192.168.56.102 | 3306 | 1508484263285064 | 496                  | NULL       |
| 192.168.56.101 | 3306 | 1508484263283073 | 773                  | NULL       |
| 192.168.56.100 | 3306 | 1508484263280089 | 575                  | NULL       |
+---------------+------+------------------+----------------------+------------+
6 rows in set (0.00 sec)
```

监控信息都已正常，没有任何报错。

15.4　配置 ProxySQL 主从分组信息

这里会用到一张表 mysql_replication_hostgroups：

```
admin@db 15:24:  [(none)]> show create table mysql_replication_hostgroups\G;
*************************** 1. row ***************************
       table: mysql_replication_hostgroups
Create Table: CREATE TABLE mysql_replication_hostgroups (
    writer_hostgroup INT CHECK (writer_hostgroup>=0) NOT NULL PRIMARY KEY,
    reader_hostgroup INT NOT NULL CHECK (reader_hostgroup<>writer_hostgroup AND reader_hostgrou
p>0),
    comment VARCHAR,
    UNIQUE (reader_hostgroup))
1 row in set (0.00 sec)
```

里面的 writer_hostgroup 是写入组的编号，reader_hostgroup 是读取组的编号。实验中使用 10 作为写入组，20 作为读取组编号。

```
insert into mysql_replication_hostgroups values (10,20,'proxy');
load mysql servers to runtime;
save mysql servers to disk;
```

```
admin@db 15:05:  [(none)]> select * from mysql_replication_hostgroups;
+------------------+------------------+---------+
| writer_hostgroup | reader_hostgroup | comment |
+------------------+------------------+---------+
| 10               | 20               | proxy   |
+------------------+------------------+---------+
1 row in set (0.00 sec)
```

ProxySQL 会根据 server 的 read_only 的取值将服务器进行分组。read_only=0 的 server，master 被分到编号为 10 的写组，read_only=1 的 server，slave 则被分到编号为 20 的读组。

```
admin@db 15:29:  [(none)]> select * from mysql_servers
    -> ;
+--------------+----------------+------+--------+--------+-------------+-----------------+---------+
| hostgroup_id | hostname       | port | status | weight | compression | max_connections | max_    |
replication_lag | use_ssl | max_latency_ms | comment |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------+
| 10           | 192.168.56.100 | 3306 | ONLINE | 1      | 0           | 1000            | 0       |
                | 0       | 0              |         |
| 20           | 192.168.56.102 | 3306 | ONLINE | 1      | 0           | 1000            | 0       |
                | 0       | 0              |         |
| 20           | 192.168.56.101 | 3306 | ONLINE | 1      | 0           | 1000            | 0       |
                | 0       | 0              |         |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------+
3 rows in set (0.00 sec)
```

配置对外访问账号，默认指定主库，并对该用户开启事务持久化保护。

> 注：mysql_users 表中的 transaction_persistent 字段默认为 0，建议在创建完用户之后设置为 1，避免发生脏读、幻读等现象，命令如下：

```
insert into mysql_users(username,password,default_hostgroup) values
('zs','zs',10);
update mysql_users set transaction_persistent=1 where username='zs';
load mysql users to runtime;
save mysql users to disk;
```

验证登录的服务器就是主库：

```
[root@proxysql ~]# /usr/local/mysql/bin/mysql -uzs -pzs -h 192.168.56.103 -P 6033 -e "show slav
e hosts"
mysql: [Warning] Using a password on the command line interface can be insecure.
+-----------+------+------+-----------+--------------------------------------+
| Server_id | Host | Port | Master_id | Slave_UUID                           |
+-----------+------+------+-----------+--------------------------------------+
|   3306101 |      | 3306 |   3306100 | ce6d8333-b539-11e7-9e81-08002783b39d |
|   3306102 |      | 3306 |   3306100 | 460e5f90-b53a-11e7-a1a7-080027cd683a |
+-----------+------+------+-----------+--------------------------------------+
```

```
[root@proxysql ~]# /usr/local/mysql/bin/mysql -uzs -pzs -h 192.168.56.103 -P 6033 -e "select @@
hostname"
mysql: [Warning] Using a password on the command line interface can be insecure.
+------------+
| @@hostname |
+------------+
| node1      |
+------------+
```

> 注：对外端口号需要指定为 6033。

15.5　配置读写分离策略

配置读写分离策略需要使用 mysql_query_rules 表。表中的 match_pattern 字段就是代表设置的规则，destination_hostgroup 字段代表默认指定的分组，apply 代表真正执行应用规则。

把所有以 select 开头的语句全部分配到编号为 20 的读组中。select for update 会产生一个写锁，对数据查询的时效性要求高，把它分配到编号为 10 的写组中，其他所有操作都会默认路由到写组中。

命令如下：

```
    insert into mysql_query_rules(active,match_pattern,destination_hostgroup,
apply) VALUES(1,'^SELECT.*FOR UPDATE$',10,1);
    insert into mysql_query_rules(active,match_pattern,destination_hostgroup,
apply) VALUES(1,'^SELECT',20,1);
    load mysql query rules to runtime;
    save mysql query rules to disk;
```

15.6　测试读写分离

通过创建的对外账户 zs 连接 ProxySQL 登录数据库。

命令如下：

```
    /usr/local/mysql/bin/mysql -uzs -pzs -h 192.168.56.103 -P 6033
```

查看 zs 库下 tt 的数据：

```
zs@db 17:23:  [(none)]> use zs;
Database changed, 2 warnings
zs@db 17:23:  [zs]> select * from tt;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)
```

然后登录管理端口，通过查询 stats_mysql_query_digest 这张表来监控查询状态，命令如下：

```
    select * from stats_mysql_query_digest;
```

```
admin@db 17:24:  [(none)]> select * from stats_mysql_query_digest;
+-----------+------------+----------+--------------------+-----------------+-----------+-----
------+-----------+-----------+-----------+
| hostgroup | schemaname | username | digest             | digest_text     | count_star | firs
t_seen | last_seen  | sum_time | min_time | max_time |
+-----------+------------+----------+--------------------+-----------------+-----------+-----
------+-----------+-----------+-----------+
| 20        | zs         | zs       | 0x33DB410ED8C2EE1E | select * from tt | 1          | 1508
491493 | 1508491493 | 933      | 933      | 933      |
+-----------+------------+----------+--------------------+-----------------+-----------+-----
------+-----------+-----------+-----------+
1 row in set (0.00 sec)
```

可见这条 select 语句自动路由到编号为 20 的读组，即 slave 库上。

然后继续测试，通过 ProxySQL 登录到数据库：

```
/usr/local/mysql/bin/mysql -uzs -pzs -h 192.168.56.103 -P 6033
```

执行 select * from zs.tt for update 和 update tt set name='ff' where score=100 的语句操作：

```
zs@db 17:24:  [zs]> select * from tt for update;
+----+------+-------+
| id | name | score |
+----+------+-------+
|  1 | aa   |    60 |
|  2 | bb   |    70 |
|  3 | cc   |    80 |
|  4 | dd   |    90 |
|  5 | ee   |   100 |
+----+------+-------+
5 rows in set (0.00 sec)
```

```
zs@db 17:33:  [zs]> update tt set name='ff' where score=100;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

这时再登录管理端口，监控查询状态，发现都已经成功路由到了编号为 10 的写组，即主库，证明读写分离设置成功。

```
admin@db 17:28:  [(none)]> select * from stats_mysql_query_digest;
+-----------+------------+----------+--------------------+-----------------------------+-------
-----+-----------+-----------+-----------+-----------+
| hostgroup | schemaname | username | digest             | digest_text                 | count_
star | first_seen | last_seen  | sum_time | min_time | max_time |
+-----------+------------+----------+--------------------+-----------------------------+-------
-----+-----------+-----------+-----------+-----------+
| 10        | zs         | zs       | 0x967436551350F0CD | select * from tt for update | 1
     | 1508491710 | 1508491710 | 1803     | 1803     | 1803     |
+-----------+------------+----------+--------------------+-----------------------------+-------
-----+-----------+-----------+-----------+-----------+
1 row in set (0.00 sec)
```

```
admin@db 17:33:  [(none)]> select * from stats_mysql_query_digest;
+-----------+-----------+----------+--------------------+--------------------------------+
| hostgroup | schemaname | username | digest             | digest_text                    |
| count_star | first_seen | last_seen | sum_time | min_time | max_time |                                |
+-----------+-----------+----------+--------------------+--------------------------------+
| 10        | zs         | zs       | 0x505AB5588AD20A31 | update tt set name=? where score=? |
| 1         | 1508492022 | 1508492022 | 50577    | 50577    | 50577    |                                |
+-----------+-----------+----------+--------------------+--------------------------------+
1 row in set (0.00 sec)
```

读写分离设置成功之后，我们可以调整权重，让某台机器承受更多的读操作。这些技巧都可以用在运维 ProxySQL 上面。

调整 192.168.56.102 node2 节点的查询权重，让更多的读请求路由到这台机器上面。

命令如下：

```
update mysql_servers set weight=10 where hostname='192.168.56.102';
load mysql servers to runtime;
save mysql servers to disk;
```

```
admin@db 11:07:  [main]> select * from mysql_servers;
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+---------+----------------+---------+
| hostgroup_id | hostname       | port | status | weight | compression | max_connections | max_replication_lag | use_ssl | max_latency_ms | comment |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+---------+----------------+---------+
| 10           | 192.168.56.100 | 3306 | ONLINE | 1      | 0           | 1000            | 0                   | 0       | 0              |         |
| 20           | 192.168.56.102 | 3306 | ONLINE | 10     | 0           | 1000            | 0                   | 0       | 0              |         |
| 20           | 192.168.56.101 | 3306 | ONLINE | 1      | 0           | 1000            | 0                   | 0       | 0              |         |
+--------------+----------------+------+--------+--------+-------------+-----------------+---------------------+---------+----------------+---------+
3 rows in set (0.00 sec)
```

15.7　总结

本章讲解了 MySQL 在生产中常用到各类集群架构。作为 DBA，无论是初学者还是已经从业多年的“老司机”，都不要急于去把每个 MySQL 集群架构搭建出来。在学习的过程中，一些读者总是存在一个误区，就是觉得我会搭建所有的数据库架构就非常厉害了。其实并不是这样的，架构搭建并不是我们的最终目的，作为 DBA 要先了解清楚自己公司的现有业务，看看公司的业务场景适合什么样的架构，要做好相应的数据库架构设计。了解好该架构的优缺点，以及在今后应用中可能出现的问题，提前做好能解决问题的预案。知己知彼，注重细节，才能避

免没日没夜地加班处理那些不该发生的问题。

下面总结了五条 MySQL 架构设计中的经验。

（1）根据公司现有业务设计合理架构。

（2）选择成熟的架构方案。

（3）因地制宜，根据实际设备情况做出选择。

（4）考虑方案的可行性。

（5）越简单越好，越适合公司越好。

第 5 部分　永恒钻石篇

如果把学习 MySQL 数据库比喻成一次马拉松，那现在可以为冲刺做好准备了。本部分将学习 MySQL 5.7 版本的新特性，以及通过硬件、操作系统、数据库、程序设计四个维度来全面介绍 MySQL 数据库的优化。机遇与挑战并存，怀揣着对技术的热爱，让我们开启新篇章的学习。

第 16 章

MySQL 5.7 新特性

在“外人”看来技术是枯燥与乏味的，每天还面临着诸多需要解决的问题，排查故障，让人提不起一点兴趣。但这就好比两人谈恋爱，在一起久了都会缺乏新鲜感，需要新元素的加入才能让人提起兴趣。身处在技术这个圈子里，从事 MySQL 数据库相关的工作，它的任何变化都吸引着我们去探究。针对新版本 MySQL 5.7，它到底改进了什么，引入了哪些新的功能，对数据库性能的优化又有哪些帮助，都需要进一步学习。其实前面章节的内容中就涉及了大量新特性的知识点，本章会选择一些特别实用的特性做个总结介绍，也希望大家可以通过本章的学习，更能激发对 MySQL 数据库的热爱，更好地把所学的知识应用到实际工作中去。

MySQL 数据库在 5 这个大版本上待了 10 年之久，我们都知道它的体系结构分为 server 层和存储引擎层，其实在 server 层并没有太大的变化，主要的更新与改变呈现在存储引擎层上，也就是目前最火的 InnoDB 存储引擎，功能和性能上都有所提升。

InnoDB 存储引擎的增强主要有两个部分，如图 16-1 所示。

接下来逐一对图 16-1 所示的这些特性展开介绍。

16.1 InnoDB 存储引擎的增强

1. Online ALTER TABLE

MySQL 5.7 版本之前不支持对索引的重命名，这次新增了索引重命名的语法，是 in place 方式，不需要再执行 table copy 的操作了。而且 5.7 版本之后也支持在线调整 varchar 列的大小。

索引重命名的语法：

```
alter table table_name rename index old_index_name to new_index_name;
```

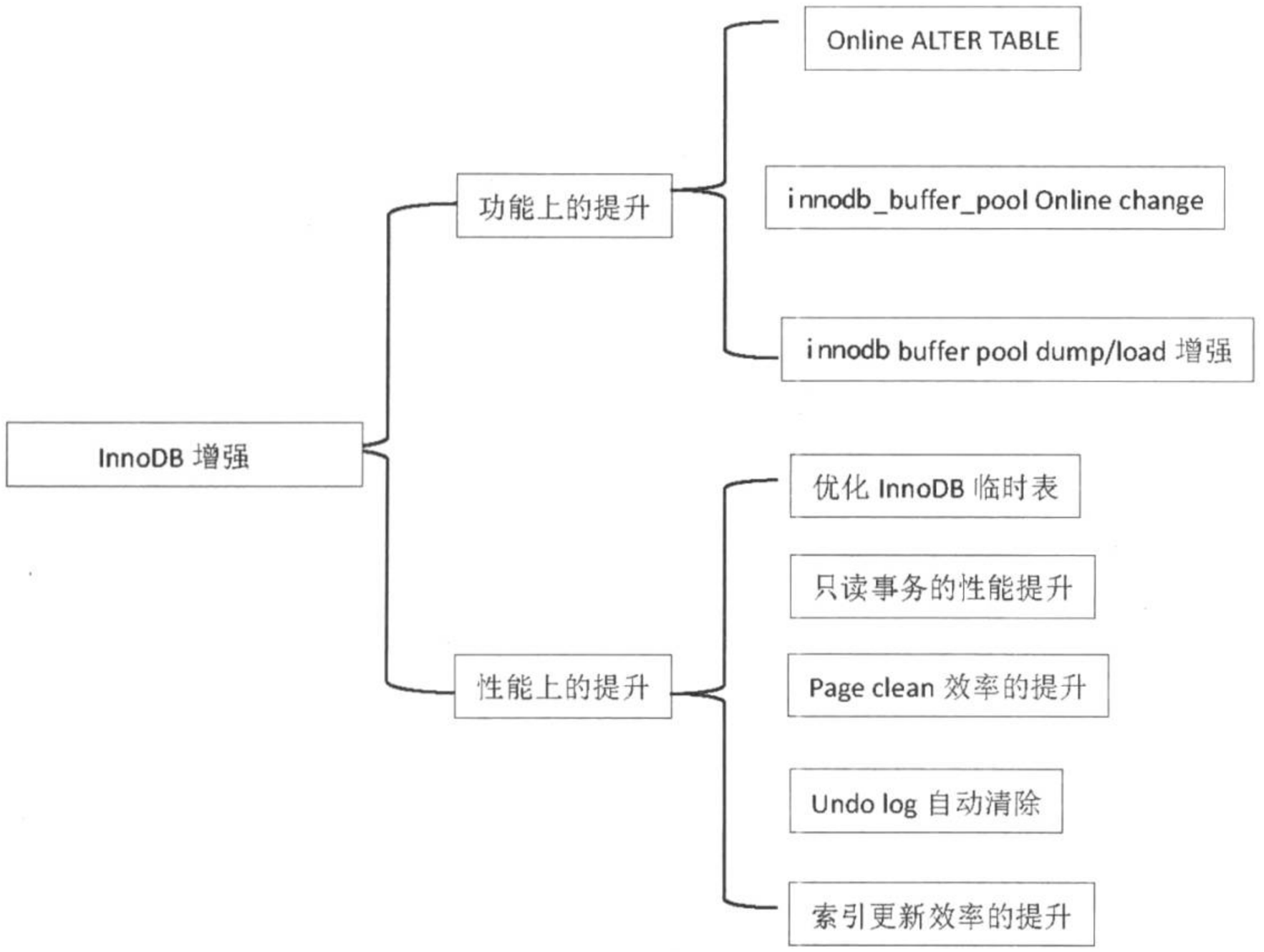


图 16-1　InnoDB enhancements

```
root@db 09:48:  [zs]> show create table tt \G;
*************************** 1. row ***************************
       Table: tt
Create Table: CREATE TABLE `tt` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL,
  `score` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  KEY `idx_name` (`name`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)

ERROR:
No query specified
```

```
root@db 09:48:  [zs]> alter table tt rename index idx_name to index_name ;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
root@db 09:49:  [zs]> show create table tt \G;
*************************** 1. row ***************************
       Table: tt
Create Table: CREATE TABLE `tt` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL,
  `score` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  KEY `index_name` (`name`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

varchar 列的大小在线调整的方法：

```
ALTER TABLE t1 ALGORITHM=INPLACE, CHANGE COLUMN c1 c1 VARCHAR(255);
```

> 注：这里需要注意的是，修改 varchar 长度只支持两种情况，一种是长度在 0~255 字节之间增加，另一种就是 256 到更大范围的。但并不支持之前的一个长度跳到 256 之后的长度（比如 80~512），当然也不支持缩减长度。不支持的条件还是使用 copy 的算法。

2. innodb_buffer_pool Online change

从 MySQL 5.7.5 之后，为了支持在线动态调整 innodb_buffer 的大小，引入了 chunk 的概念，每个 chunk 默认大小为 128MB。buffer pool 以 innodb_buffer_pool_chunk_size 为单位进行动态增大和缩小。通过设置动态参数 innodb_buffer_pool_size 来修改 buffer pool 的大小，此过程不会消耗太高的代价。innodb_buffer_pool_size 的大小要求是 innodb_buffer_pool_chunk_size* innodb_buffer_pool_instances 的倍数，如果不是，将适当调大 innodb_buffer_pool_size。值从大改小的过程需要释放内存。

还能通过新增状态参数 innodb_buffer_pool_resize_status 来监控 buffer pool 的 resize 过程。

```
root@db 11:02:  [(none)]> show variables like '%innodb_buffer_pool_size%';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| innodb_buffer_pool_size | 1073741824 |
+-------------------------+------------+
1 row in set (0.00 sec)

root@db 11:02:  [(none)]> set global innodb_buffer_pool_size=2147483648;
Query OK, 0 rows affected (0.01 sec)

root@db 11:02:  [(none)]> show variables like '%innodb_buffer_pool_size%';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| innodb_buffer_pool_size | 2147483648 |
+-------------------------+------------+
1 row in set (0.00 sec)
```

```
root@db 11:03:  [(none)]> show status like '%innodb_buffer_pool_resize%';
+---------------------------------+-----------------------------------------------+
| Variable_name                   | Value                                         |
+---------------------------------+-----------------------------------------------+
| Innodb_buffer_pool_resize_status | Completed resizing buffer pool at 171028 11:02:42. |
+---------------------------------+-----------------------------------------------+
1 row in set (0.00 sec)
```

3. innodb_buffer_pool dump 和 load 的增强

innodb buffer pool 的 dump 和 load 操作加强是通过设置新增参数 innodb_buffer_pool_dump_pct 实现的，即 dump 的百分比。只导出最热的那部分数据的 page，当系统繁忙时，可以

通过 innodb_io_capacity 参数限制 buffer pool load 的过程。

```
root@db 11:27:  [(none)]> show variables like '%innodb_buffer_pool_dump_pct%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_buffer_pool_dump_pct   | 25    |
+-------------------------------+-------+
1 row in set (0.00 sec)
```

```
root@db 11:32:  [(none)]> show variables like '%innodb_io_capacity';
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| innodb_io_capacity | 4000  |
+--------------------+-------+
1 row in set (0.01 sec)
```

4. InnoDB 临时表优化

临时表不再记录 redo log，MySQL 5.7 把临时表的数据从系统表空间中抽离出来，形成了自己的独立表空间，并且把临时表的相关检索信息保存在系统信息表中 information_schema 库下 innodb_temp_table_info 表中。但目前还不能定义临时表空间文件的存放路径，只能与 innodb_data_home_dir 一致。

独立表空间文件名 ibtmp1，默认大小为 12MB：

```
root@db 12:01:  [(none)]> show variables like '%temp_data_file_path%';
+-------------------------------+-------------------------+
| Variable_name                 | Value                   |
+-------------------------------+-------------------------+
| innodb_temp_data_file_path    | ibtmp1:12M:autoextend   |
+-------------------------------+-------------------------+
1 row in set (0.00 sec)
```

5. page clean 的效率提升

page cleaner 线程不再只有一个，可以通过新增的参数 innodb_page_cleaners 来指定 page cleaner 线程的数量，提高脏页的刷新效率：

```
root@db 12:08:  [(none)]> show variables like '%innodb_page_clea%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_page_cleaners  | 4     |
+-----------------------+-------+
1 row in set (0.00 sec)
```

6. Undo log 自动清除

启动新增的 innodb_undo_log_truncate 参数（默认是关闭状态），设置 innodb_undo_log_truncate=1 开启。

当 undo log 的大小超过 innodb_max_undo_log_size 参数指定的最大值时，undo log 就会自

动清除，以防止磁盘空间产生消耗。

```
root@db 12:18:  [(none)]> show variables like '%innodb_undo_log_tr%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_log_truncate | OFF   |
+--------------------------+-------+
1 row in set (0.00 sec)

root@db 12:18:  [(none)]> set global innodb_undo_log_truncate=1;
Query OK, 0 rows affected (0.00 sec)

root@db 12:19:  [(none)]> show variables like '%innodb_undo_log_tr%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_log_truncate | ON    |
+--------------------------+-------+
1 row in set (0.00 sec)
```

MySQL 5.7 除了 InnoDB 的特性增强，还有其他方面的改进。

16.2 其他方面的增强

1. 安全性的增强

安装库初始化时，废弃了 mysql_install_db 命令，改为使用 mysqld 配合--initialize 的方式。root 用户的密码不再默认为空，而是随机产生一个密码，保存在错误日志里面。要求 MySQL 库下 user 表中的 plugin 列为非空，默认值是 mysql_native_password，而不是 mysql_old_password，不再支持旧密码格式了。密码不再记录在 password 字段中，而是使用 authentication_string 字段记录。而且默认安装完成之后已经没有 test 库。还新增了 super_read_only 参数，来禁止超管的写操作权限。增加了账户密码有自动过期策略，密码过期之后必须强制进行修改，还增加了用户锁定限制，可以锁定或者解锁用户来更好地进行登录控制，通过 MySQL 库下 user 表中新增的 account_locked 列来表示锁定状态。如果 MySQL Server 使用 OpenSSL 编译，则 MySQL Server 可以自动创建 SSL、RSA 证书和 key 文件来支持安全连接。

2. sql_mode 的变化

默认开启严格模式的 sql mode（STRICT_TRANS_TABLES）。在 5.7 版本之前默认是 NO_ENGINE_SUBSTITUTION。启用严格模式下，如果遇见 SQL 书写有问题，就会直接抛出错误，不会出现超长内容自动被截断的现象。而且不能在 grant 命令中直接创建用户了，需要使用 create user 命令。

3. sys schema 功能的增强

sys schema 是 MySQL 5.7.7 引入的一个系统库，包含了一系列的视图、函数、存储过程。

sys schema 的数据来源主要来自 performance_schema，其目的就是为了降低查询 performance_schema 的复杂度，可以让 DBA 和开发人员诊断问题。在没有这个库之前，我们可能只能根据经验去排查数据库的问题，但现在可以通过 sys schema 了解哪些语句使用了临时表，哪个用户请求了最多的 I/O，哪个线程占用了最多的内存，哪些索引是无用索引等。

sys 库下有两种表：一种是以字母开头的，提供更好的阅读体验；另一种是以 x$开头的，提供了原始数据，适合工具采集数据。

sys 库下重点视图介绍。

- 以 host 开头的视图：记录主机相关的统计信息。
- 以 innodb 开头的视图：记录 innodb buffer 相关的信息。
- 以 io 开头的视图：记录 I/O 相关的信息，像等待 I/O、I/O 的使用等。
- 以 memory 开头的视图：记录从各种维度来展示内存的使用情况。
- metrics 视图：记录数据库内部的统计数值。
- processlist 和 session 视图：记录连接会话相关的信息。
- 以 schema 开头的视图：记录表的统计信息。
- 以 statement 开头的视图：基于语句的统计信息。
- 以 user 开头的视图：记录统计用户执行语句、I/O 使用等信息。
- 以 wait 开头的视图：记录等待事件的相关情况。

利用 sys 库查询生产环境中需要的信息。

查看数据库中索引的使用情况语句：

```
select index_name,rows_selected,rows_inserted,rows_deleted,rows_updated
from schema_index_statistics where table_schema='DB_name' and table_name=
'**' and index_name='**';
```

```
root@db 14:19:  [sys]> select index_name,rows_selected,rows_inserted,rows_deleted,rows_updated
 from schema_index_statistics where table_schema='zs' and table_name='tt' and index_name='name_
idx';
+------------+---------------+---------------+--------------+--------------+
| index_name | rows_selected | rows_inserted | rows_deleted | rows_updated |
+------------+---------------+---------------+--------------+--------------+
| name_idx   |             3 |             0 |            0 |            0 |
+------------+---------------+---------------+--------------+--------------+
1 row in set (0.00 sec)
```

查看数据库中冗余索引的 SQL 语句：

```
select * from sys.schema_redundant_indexes;
```

```
root@db 14:23:  [sys]> select * from sys.schema_redundant_indexes\G;
*************************** 1. row ***************************
              table_schema: zs
                table_name: tt
      redundant_index_name: name_idx
    redundant_index_columns: name
redundant_index_non_unique: 1
        dominant_index_name: na_sc_idx
     dominant_index_columns: name,score
 dominant_index_non_unique: 1
           subpart_exists: 0
           sql_drop_index: ALTER TABLE `zs`.`tt` DROP INDEX `name_idx`
1 row in set (0.00 sec)

ERROR:
No query specified
```

查看数据库的未使用到的索引：

```
select * from sys.schema_unused_indexes;
```

```
root@db 14:23:  [sys]> select * from sys.schema_unused_indexes;
+---------------+-------------+-------------+
| object_schema | object_name | index_name  |
+---------------+-------------+-------------+
| zs            | tt          | name_idx    |
| zs            | tt          | na_sc_idx   |
+---------------+-------------+-------------+
2 rows in set (0.00 sec)
```

查看 I/O 使用最多的表：

```
select * from io_global_by_file_by_bytes limit 1;
```

```
root@db 14:32:  [sys]> select * from io_global_by_file_by_bytes limit 1;
+-------------------+------------+------------+-----------+-------------+---------------+-------------+-----------+
| file              | count_read | total_read | avg_read  | count_write | total_written | avg_write   | total
| write_pct |
+-------------------+------------+------------+-----------+-------------+---------------+-------------+-----------+
| @@datadir/ibtmp1  |          0 | 0 bytes    | 0 bytes   |        2845 | 119.27 MiB    | 42.93 KiB   | 119.27 MiB
|    100.00 |
+-------------------+------------+------------+-----------+-------------+---------------+-------------+-----------+
1 row in set (0.03 sec)
```

查看实例消耗的内存：

```
select * from sys.memory_global_total;
```

```
root@db 14:23:  [sys]> select * from sys.memory_global_total;
+-----------------+
| total_allocated |
+-----------------+
| 1.33 GiB        |
+-----------------+
1 row in set (0.00 sec)
```

查看每个库占用多少 buffer pool：

```
root@db 14:35:  [sys]> select * from innodb_buffer_stats_by_schema;
+---------------+-----------+------------+-------+--------------+-----------+-------------+
| object_schema | allocated | data       | pages | pages_hashed | pages_old | rows_cached |
+---------------+-----------+------------+-------+--------------+-----------+-------------+
| InnoDB System | 11.69 MiB | 10.62 MiB  |   748 |            0 |         0 |       12167 |
| zs            | 4.42 MiB  | 3.41 MiB   |   283 |          110 |         0 |       50119 |
| mysql         | 240.00 KiB| 2.23 KiB   |    15 |            1 |         0 |          31 |
| sys           | 16.00 KiB | 338 bytes  |     1 |            0 |         0 |           6 |
+---------------+-----------+------------+-------+--------------+-----------+-------------+
4 rows in set (0.47 sec)
```

前五位占用最多 buffer pool 的表：

```
root@db 14:40:  [sys]> select * from innodb_buffer_stats_by_table order by pages desc limit 5;
+---------------+--------------+-----------+-----------+-------+--------------+-----------+-------------+
| object_schema | object_name  | allocated | data      | pages | pages_hashed | pages_old | rows_cached |
+---------------+--------------+-----------+-----------+-------+--------------+-----------+-------------+
| InnoDB System | SYS_TABLES   | 11.58 MiB | 10.63 MiB |   741 |            0 |         0 |       36450 |
| zs            | tt           | 4.42 MiB  | 3.41 MiB  |   283 |          110 |         0 |       50119 |
| InnoDB System | SYS_FOREIGN  | 32.00 KiB | 0 bytes   |     2 |            0 |         0 |           0 |
| InnoDB System | SYS_COLUMNS  | 16.00 KiB | 8.83 KiB  |     1 |            0 |         0 |         136 |
| InnoDB System | SYS_DATAFILES| 16.00 KiB | 1.08 KiB  |     1 |            0 |         0 |          21 |
+---------------+--------------+-----------+-----------+-------+--------------+-----------+-------------+
5 rows in set (0.16 sec)
```

4. 复制功能的增强

在复制章节中，我们已经介绍了 5.7 版本中复制功能的提升。主要分为以下几点。

- 并行复制：基于 logical-clock（5.7 版本引入），一个组内提交的内事务都可以并行，可以达到接近主库并发效果。
- 多源复制：支持由多个 master 向一个 slave 复制。用于将多个服务器备份到单个服务器上。可用于异地容灾，集中备份。
- 增强半同步：是在 MySQL 5.5 半同步复制基础上的增强，在集群架构切换时可以保证数据的一致性。由 after_commit 模式变成了 after_sync，提高了复制的效率和数据的可靠性。
- 组复制（MGR）：有点像 Oracle 里面的 RAC 集群，可以保证多节点并行写入数据，比较类似于 PXC 架构。建议：目前不是很成熟，先不建议使用。

5. 设置查询 SQL 的超时（max_execution_time）

MySQL 5.7.4 刚引入时的名字是 max_statement_time，后来改成 max_execution_time。这个参数很实用，是一种自我保护的措施。

防止因为一条 SQL 语句的长时间执行，导致数据库雪崩。

6. 执行计划的增强

MySQL 5.7 可以直接查看正在运行的 SQL 语句的执行计划。

通过 show full processlist 查看正在执行 SQL 的线程号，然后利用 explain for connetion **（SQL 线程号）查看当前语句的执行计划。

```
root@db 15:17:  [(none)]> show full processlist;
+----+------+-----------------------+------+---------+------+----------+-----------------------------------------------------------------------------------+
| Id | User | Host                  | db   | Command | Time | State    | Info                                                                              |
+----+------+-----------------------+------+---------+------+----------+-----------------------------------------------------------------------------------+
| 17 | root | localhost:54801       | zs   | Query   |    3 | update   | insert into tt(name,score) values(concat('name',i),FLOOR(50 + (RAND() * 50))) |
| 18 | root | localhost             | NULL | Query   |    0 | starting | show full processlist                                                             |
| 22 | zs   | 192.168.56.101:54803  | zs   | Query   |    3 | query end | update tt set name='bb'                                                          |
+----+------+-----------------------+------+---------+------+----------+-----------------------------------------------------------------------------------+
3 rows in set (0.00 sec)
```

```
root@db 15:17:  [(none)]> explain for connection 22;
+----+-------------+-------+------------+------+---------------+---------+---------+------+-------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref  | rows  | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+---------+---------+------+-------+----------+-----------------+
|  1 | UPDATE      | tt    | NULL       | index | NULL          | PRIMARY | 4       | NULL | 70292 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+---------+---------+------+-------+----------+-----------------+
1 row in set (0.00 sec)
```

7. 优化器的提升

MySQL 5.7 之前，优化器是性能瓶颈点。5.7 之后改善了很多功能。比如 in 语句子查询优化，MySQL 5.7 的 in 查询能够使用 index range scan 方式，Union all 不再产生临时表，新增了更多的 HINTS，并提供新的 HINTS 语法。排序效率上也有所提升。

第 17 章

MySQL 全面优化

随着公司业务的增长，MySQL 数据库上跑的应用也会越来越多，随之带来的就是数据库逐渐会遇到性能瓶颈的问题。数据库是所有架构中不可缺少的一环，一旦数据库出现性能问题，会给整个系统带来灾难性的后果。像一些电商、游戏类的公司，由于性能问题，导致业务受影响，带来的经济损失将不可估量。所以如何优化 MySQL 数据库的性能，就需要我们花费大量精力去研究。

众所周知，绝大多数使用 Linux 操作系统的大中小型互联网网站都在使用 MySQL 作为其后端的数据库存储，从大型的 BAT 门户，到电商平台、游戏、分类门户、SNS 等无一例外地都使用 MySQL 数据库，它可以很好地配合 Linux、PHP、Apache、Nginx，比如 LAMP 和 LNMP 架构。想优化 MySQL，首先要先了解它的特点。在使用 MySQL 的过程中，我们要注意以下几点。

- 不要错误得把它当作一个文件存储，诸如图片、附近之类的都放在 MySQL 数据库中，这样就很容易导致表空间很庞大，磁盘 I/O 的读写性能很差。
- 也不要把 MySQL 数据库当成一个计算器，在其中进行大量复杂的运算。
- 更不要把它当成一个全文检索工具，我们就单纯地把 MySQL 当成一个可以处理并发事务，保证数据一致性的数据库就可以了。

为了更加清楚地了解 MySQL 的特点，下面从三个方面讲解。首先从 CPU 开始介绍，MySQL 5.1 版本之前的多核支持能力很差，随着后期版本的升级，用到的 CPU 核数也越来越多。

MySQL 5.6 版本时已经可以用到 64 个核了。所用建议定期升级 MySQL 线上的版本到当前最新的稳定版本。当每个连接进入数据库时，MySQL 都会创建一个线程来响应此连接的请求。每个 SQL 语句只能用一个 CPU 核心，如果此 SQL 语句执行得很缓慢，就容易出现锁等待、排队等现象。所有设计的业务逻辑不要过于复杂，避免写那种很多表连接的 SQL 语句，尽量让 SQL 语句快速执行成功。还有在项目初期设计表时，数据类型的选择也很重要，一般建议类型选取方案越简单越好，请参考之前介绍的表章节中的内容。

接下来是内存，实际上 MySQL 内存的组成和 Oracle 类似，也可以分为 SGA（系统全局区）和 PGA（程序缓存区）。每个连接会话连到数据库时，都会为其分配内存，分配的内存大小一定要控制好，不宜过大，防止当业务高峰，连接数过高时，有可能会出现 OOM 现象（内存溢出）。相比 Oracle 的内存管理机制，MySQL 的内存管理还是很简单的。在 InnoDB 存储引擎中实现了自己的内存池系统和内存堆分配系统，在 InnoDB 的内存管理系统中，大致分为基础的内存块分配管理、内存伙伴分配器和内存堆分配器三个部分。InnoDB 定义和实现内存池的主要目的是提供内存的使用率和效率，防止内存碎片和内存分配跟踪和调试。内存的主要作用就是缓存热点数据，尽量保证所有的读取操作都在内存中完成，避免产生过多的物理 I/O。在业务高峰期，高并发的环境下，我们可以适当地增加物理内存的大小，来提高数据库的并发性能。说到内存，就不得不说 innodb_buffer_pool，它的作用是缓存 InnoDB 表的数据、索引、插入缓冲、数据字典等信息。如果是单实例，且数据库中绝大多数是 InnoDB 存储引擎表，则它的大小可考虑设置为物理内存的 50%～80%。还有一个需要引起注意的就是 query cache，在第一部分已经介绍过了，通常情况下建议关闭它，不要使用。有些人一直误认为只需要把 query_cache_size 大小调整为 0 就算彻底关闭了，但实际上需要把 query_cache_size 和 query_cache_type 两个参数同时设置为 0，才算关闭 Query Cache 功能。

> 注：在 MySQL 5.6 之前的版本 Query Cache 默认是开启的，5.6 版本之后默认是关闭的。

建议可以选择 Redis、memcache 或者 MongoDB 来作为缓存热点数据层。

最后介绍 MySQL 利用磁盘的方式，来更加深入地了解 MySQL。MySQL 的 binlog 文件、存储引擎文件 undo 和 redo 都是顺序写入的方式。数据文件是伴随着顺序写和随机写一起的。OLTP 这类业务系统都是以随机 I/O 为主，可以通过增大内存来提升数据库读写性能。

我们对 MySQL 利用 CPU、内存和磁盘的特点有了进一步的了解，但这只是做到了想要优化 MySQL 性能的第一步。想要做好优化工作，只知道 MySQL 数据库的知识体系是远远不够的，还要熟悉 Linux 操作系统、硬件、程序设计这三个层面。接下来从这四个维度去总结优化数据库的细节，如图 17-1 所示。

先从第一个维度硬件设备来谈针对 MySQL 的优化。

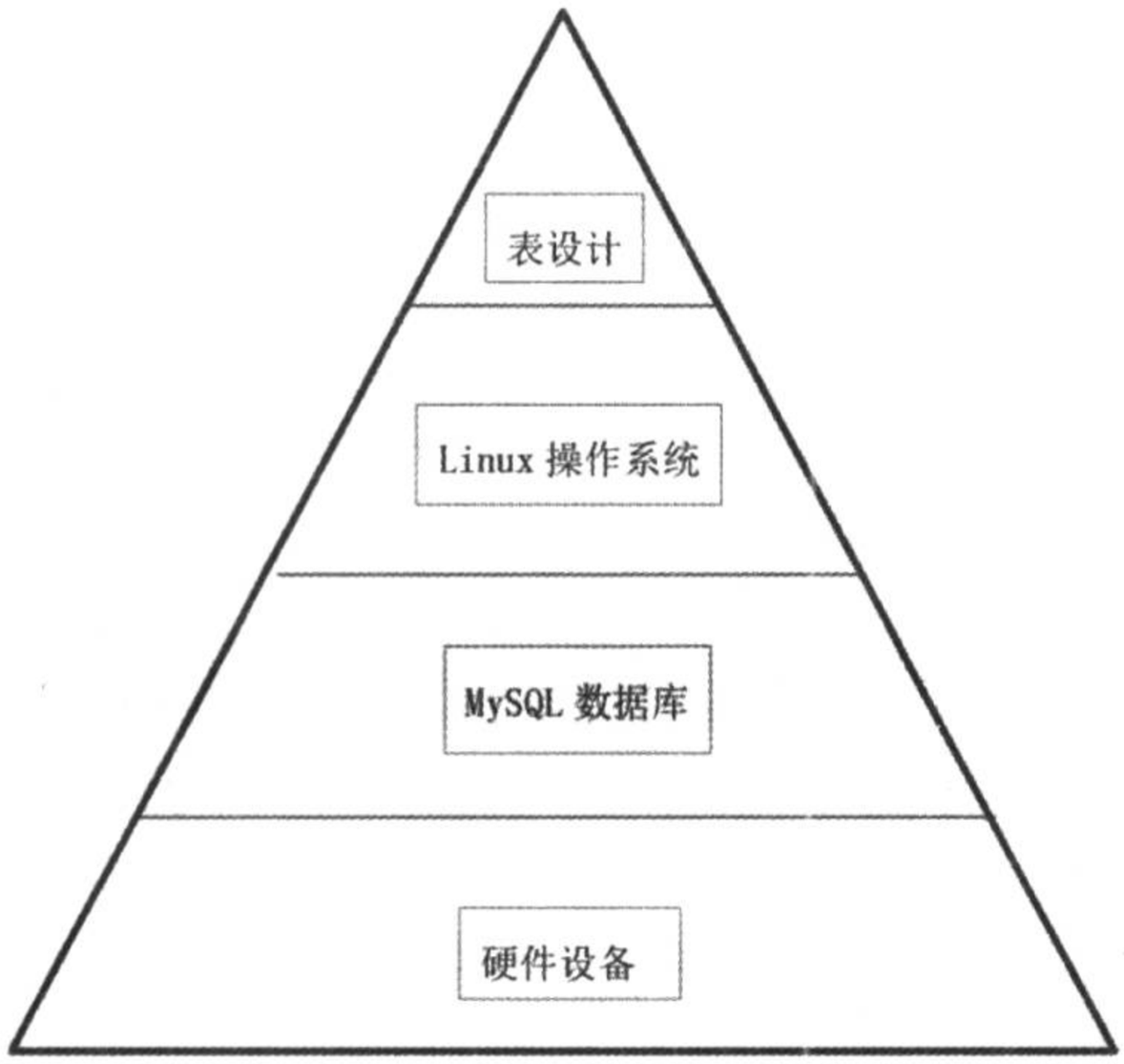


图 17-1　优化维度

17.1　硬件优化

影响数据库最大的性能问题就是磁盘 I/O，为了提升数据库的 IOPS 性能，可以使用 SSD 或者 PCIE-SSD 高速磁盘设备，至少可以获得上百倍或者上万倍的 IOPS 性能提升。当数据库系统 TPS 过高或者业务量较高时，一定要配置阵列卡，而且阵列卡一直要配备 cache 模块，cache 模块还要配置 BBU 模块来提供持续电量。防止发生断电情况时，出现数据丢失的现象，因为 cache 模块可以继续工作。在选择阵列卡的 cache 策略时，为了提高 I/O 写数据性能，强烈建议设置为 Write Back（WB）。这种策略是将数据先写到 cache 中，然后再通过阵列卡控制，刷回到磁盘里面。千万不要采用 Write Through 策略，不经过 cache 来直接写盘。在阵列卡中 cache 的大小不是很大，建议关闭其中的预读功能，让 cache 起到缓存的作用就可以了。针对阵列级别的选择，建议使用 RAID 1+0，而不要选择其他模式。

在服务器 BIOS 层面也可以对 MySQL 数据库的优化进行一些配置参数的设置。CPU 建议采用最大的性能模式，选择 performance per watt optimized 来充分发挥 CPU 的最大功耗性能，同时建议关闭 C1E 和 C stats 这类节能选项，因为当负载增加，或者访问量变大时，有可能会造成数据库响应不了太大的请求，从而出现数据库性能变慢、卡住的现象，甚至还有可能 MySQL 服务宕机。

内存方面也要选择最大性能模式 maximum performance。尽量在 BIOS 里就禁掉 NUMA 功能，将 Node Interleaving 设置为 Enabled 模式，让内存在多个 CPU 节点之间进行循环使用，这样可以更好地利用内存。无论 OS 层面的 numa 功能是否打开，都不会影响数据库的性能。

接下来讨论 MySQL 中重要的配置参数优化。

17.2 配置参数优化

- innodb_buffer_pool_size：这个参数已经在之前的章节中讲过很多次了，如果是单实例且绝大多数是 InnoDB 引擎表，则可以设置为物理内存的 50%～80%左右。
- innodb_flush_log_at_trx_commit 和 sync_binlog：分别是 redo log 刷新和 binlog 刷新的参数。如果要求数据不能丢失，建议把这两个参数都设置为 1。这就是数据库中的双一概念，可以保证主从架构中数据的一致性。但想要保证其强一致性，主从故障切换过程中不丢数据，可以考虑使用 MySQL 5.7 的增强半同步功能。
- innodb_max_dirty_pages_pct：该参数是指脏页占 innodb buffer pool 的比例。当比例到达所设置的值时，触发刷脏页到磁盘。不要将该参数设置太大，脏页过多也会影响数据库的 TPS。该值建议调整为 25%～50%。
- innodb_io_capacity：InnoDB 后台进程最大的 I/O 性能指标，影响刷新脏页和插入缓冲的数量。默认值是 200，在高转速磁盘下，可以适当提高该参数的值。SSD 磁盘配置下可以调整该值为 5000～20000，PCIE-SSD 可以调整得更高（50000 左右）。
- innodb_data_file_path = ibdata1:1G:autoextend：该参数不要使用默认的 10MB 大小，一般设置为 1GB。防止在高并发情况下，数据库受到影响。
- long_query_time：该参数在 5.5 版本以上已经可以设置为小于 1 了，建议设置该参数值为 0.1～0.5s。记录那些执行较慢的 SQL，便于后续的优化性能排查。
- binlog_format：建议 binlog 的记录格式设置为 row 模式，让数据更加安全可靠，复制过程中不会出现丢数据的情况。
- interactive_timeout,wait_timeout：两个参数分别代表交互式等待时间和非交互式等待时间，两个参数设置的值要一致，且必须同时修改，建议调整为 300～500s，不要取默认值 8 小时。
- max_connections：数据库最大连接数，不要盲目去调大连接数的数量，应该注意优化业务中的 SQL 语句，让 SQL 快速执行完成，可以释放掉连接。一味地提高连接数的值，容易发生 OOM 现象，从而可能“kill”掉 MySQL 服务，所以一定谨慎调高该值。在调高参数的同时，还应该调低 interactive_timeout 和 wait_timeout 的值。
- innodb_log_file_size：redo log 的值不要太大，如果值太大，当实例恢复时，会消耗大量时间；值太小也会造成日志切换过于频繁。
- gerenal log：全量日志建议关闭，否则该日志文件会越来越大，造成磁盘空间的紧张，MySQL 的性能也会逐渐下降。

17.3　从 Linux 操作系统层面来谈对 MySQL 的优化

在 Linux 操作系统上，主要介绍几个关键点，第一点就是 I/O 调度问题，建议选择 deadline 或者 noop 模式。千万不要使用 cfq，因为会严重影响数据库的性能。

查看 I/O 调度的方式（中括号里面的就是当前使用的 I/O 调度方式）：

```
cat /sys/block/sda/queue/scheduler
```

```
[root@node2 ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory [deadline] cfq
```

第二点就是文件系统的选择。文件系统推荐采用 xfs，其次可以选择 ext4，基本上可以放弃 ext3。

第三点涉及一个内核参数 vm.swappiness，该参数表示使用 swap 的意向。如果设置该参数值较高，则意味着当内存快用尽时，倾向于使用 swap，而不特意释放内存中的数据。如果设置该值较低，则意味着不倾向于使用 swap，需要释放内存中的数据，以便用于后续新数据在内存中的读写。该参数可以在 1～10 范围内取值，不建议设置为 0，因为这种行为有可能会导致系统内存溢出（OOM），从而导致 MySQL 被意外“kill”掉。

查看 swappiness 的参数设置：

```
cat /proc/sys/vm/swappiness
```

```
[root@node2 ~]# cat /proc/sys/vm/swappiness
10
```

想要修改 swappiness 的值，编辑/etc/sysctl.conf，加入 vm.swappiness 的值即可。

swappiness 参数取值详解如表 17-1 所示。

表 17-1　swappiness 参数取值

swappiness 取值	swap 取值含义
vm.swappiness=0	禁用 swap，可能会出现 OOM 现象
vm.swappiness=1	尽量不使用 swap
vm.swappiness=60	默认值，不考虑这样去设置
vm.swappiness=100	积极得使用 swap，很影响性能

还有两个内核参数需要我们关注：vm.dirty_background_ratio 和 vm.dirty_ratio。

- vm.dirty_background_ratio：这个参数指定了当文件系统缓存脏页数量达到系统内存的百分比例多少时，就会触发 pdflush/flush/kdmflush 等后台回写进程运行，将一定缓存的脏页异步地刷入磁盘。建议该参数的取值不要超过 10 就可以了。

- vm.dirty_ratio：这个参数指定了当文件系统缓存脏页数量达到系统内存百分比例多少时，系统不得不开始处理缓存脏页。

> 注：由于脏页数量已经比较多，为了避免数据丢失，需要将一定脏页刷入磁盘。

在此过程中可能很多应用进程需要去处理文件 I/O 而出现阻塞现象。

建议该参数的取值不要最好超过 20。

最后谈谈表设计层面和一些我们忽略的细节点相关的优化工作。

17.4 表设计及其他优化

作为 DBA 的我们，在工作中可能会面对开发人员的各种要求，但是他们是否知道在操作数据库时需要注意哪些事项呢？下面总结了 18 点建议：

（1）在创建业务表时，库名、表名、字段名必须使用小写字母，采用“_”分割。

（2）MySQL 数据库中，通过 lower_case_table_names 参数来区分表名的大小写，默认为 0，代表大小写敏感。如果是 1，代码大小写不敏感，以小写存储。为字段选取数据类型时，要秉承着简单、够用的原则。表中的字段和索引数量都不宜过多，要保证 SQL 语句查询的高效性，快速执行完，避免出现堵塞、排队现象。

（3）表的存储引擎一定要选择使用 InnoDB，之前的章节中已经反复强调过了 InnoDB 和 MyISAM 的区别。MySQL 5.7 基本已经废弃 MyISAM，从 8.0 版本开始，系统表也彻底与 MyISAM 告别了。

（4）要显式地为表创建一个使用自增列 INT 或者 BIGINT 类型作为主键，可以保证写入顺序是自增的，和 B+tree 叶子节点分裂顺序一致。写入更加高效，TPS 性能会更高，存储效率也是最高的。

（5）金钱、日期时间、IPv4 尽量使用 int 来存储。用 int 来存储金钱，让 int 单位为分，这样就不存在四舍五入了，存储的数值更精确。

日期时间可以选择使用 datetime，datetime 的可用范围比 timestamp 大，物理存储上仅比 timestamp 多占 1 个字节的空间，整体性能上的消耗并不算太大。因此在生产环境可以使用 datetime 时间类型。当然也可以使用 int 来存储时间，通过转换函数 from_unixtime 和 unix_timestamp 来实现。

```
mysql> select unix_timestamp('2017-08-29 14:09:11');
+---------------------------------------+
| unix_timestamp('2017-08-29 14:09:11') |
+---------------------------------------+
|                            1503986951 |
+---------------------------------------+
1 row in set (0.01 sec)

mysql> select from_unixtime(1503986951);
+---------------------------+
| from_unixtime(1503986951) |
+---------------------------+
| 2017-08-29 14:09:11       |
+---------------------------+
1 row in set (0.00 sec)
```

IPv4 字段基本上可以不使用 char(15)来存储，使用 int 来存储，通过转换函数 inet_aton 和 inet_ntoa 来实现。

```
root@db 23:32:  [zs]> select inet_aton('192.168.56.102');
+-----------------------------+
| inet_aton('192.168.56.102') |
+-----------------------------+
|                  3232249958 |
+-----------------------------+
1 row in set (0.00 sec)

root@db 23:32:  [zs]>
root@db 23:32:  [zs]> select inet_ntoa(3232249958);
+---------------------------+
| inet_ntoa(3232249958)     |
+---------------------------+
| 192.168.56.102            |
+---------------------------+
1 row in set (0.00 sec)
```

有些字段一看就知道该选择什么数据类型进行存储，比如性别 sex 字段、状态 status 字段，基本上选择 tinyint 就可以了。

（6）text 和 blob 这种存大量文字或者存图片的大数据类型，建议不要与业务表放在一起。注：主要业务表切忌出现这样大类型的字段。

SQL 语句中尽量避免出现 or 子句，这种判断的子句可以让程序自行完成，不要交给数据库判断。也要避免使用 union，尽量采用 union all，减少去重和排序的工作。

（7）用 select 查询表时只需要获取必要的字段，避免使用 select *。这样可以减少网络带宽消耗，还有可能利用到覆盖索引。

建立索引时不要在选择性低的字段上创建，比如 sex、status 这种字段。

索引的选择性计算方法：

```
select count(distinct col1)/count(*) from table_name;
```

越接近 1，证明选择性越高，越适合创建索引。

（8）很长的字符串列可以考虑创建前缀索引，提高索引利用率。

单表索引数量不要太多，一般建议不要超过 4～5 个（根据实际业务表再确定）。当执行 DML 语句操作时，也会对索引进行更新，如果索引数量太多，则会造成索引树的分裂，性能也会下降。

（9）所有字段定义中，默认都加上 not null 约束，避免出现 null。在对该字段进行 select count() 统计计数时，可以让统计结果更准确，因为值为 null 的数据不会被计算进去。

（10）表的字符集默认使用 UTF8，必要时可申请使用 UTF8mb4 字符集。因为它的通用性比 GBK、Latin1 都要好。UTF8 字符集存储汉字占用 3 个字节，如果遇到表情存储的需求，就可以使用 UTF8mb4

（11）建议模糊查询 select...like '%**%' 的语句不要出现在数据库中，可以使用搜索引擎 sphinx 代替。

（12）索引字段上面不要使用函数，否则使用不到索引，也不要创建函数索引。

（13）join 列类型要保持一致，其中包括长度、字符集都要一致。

（14）当在执行计划中的 extra 项看到 Using filesort，或者看到 Using temporary 时，也要优先考虑创建排序索引和分组索引。

> 注：排序、分组字段上都需要创建索引。

（15）limit 语句上的优化，建议使用主键来进行范围检索，缩短结果集大小，使查询更高效。

（16）通常情况下，可以使用 MySQL 自带的工具 mysqldumpslow 或者最常使用的第三方工具软件 pt-query-digest 来捕获线上的影响业务的慢查询 SQL 语句。

（17）还可以使用 MySQL 提供的可以用来分析当前会话中语句执行资源消耗情况的命令 show profile。

首先查看 profile 参数（默认是关闭的）：

```
root@db 14:29:  [(none)]> show variables like 'profiling';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| profiling     | OFF   |
+---------------+-------+
1 row in set (0.00 sec)
```

我们需要开启该参数，执行 set profiling=1（开启）。

然后执行一条建立索引 SQL 语句的过程：

```
root@db 14:34:  [zs]> create index name_idx on tt(name);
Query OK, 0 rows affected (0.44 sec)
Records: 0  Duplicates: 0  Warnings: 0

root@db 14:34:  [zs]> select count(*) from tt;
+----------+
| count(*) |
+----------+
|    50000 |
+----------+
1 row in set (0.01 sec)
```

接着通过 show profiles 查看 SQL 运行结果。

Query_ID 代表执行语句的编号，1 代表执行建立索引语句的编号：

```
root@db 14:34:  [zs]> show profiles;
+----------+-------------+-------------------------------------+
| Query_ID | Duration    | Query                               |
+----------+-------------+-------------------------------------+
|        1 | 0.43702950 | create index name_idx on tt(name)   |
|        2 | 0.01030900 | select count(*) from tt             |
+----------+-------------+-------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

最后通过 show profile for query 1 查看建立索引语句的资源消耗情况：

Status	Duration
starting	0.000069
checking permissions	0.000010
init	0.000006
Opening tables	0.000084
setup	0.000030
creating table	0.004851
After create	0.000165
System lock	0.000011
preparing for alter table	0.006230
altering table	0.406017
committing alter table to stor	0.015046
end	0.000029
query end	0.004285
Waiting for semi-sync ACK from	0.000017
query end	0.000018
closing tables	0.000012
freeing items	0.000137
cleaning up	0.000017

可见这条 SQL 的消耗时间大概是 0.4s 左右。

还可以通过 show profile block io,cpu for query 1 语句来查看。

这条创建索引的 SQL 语句消耗 CPU 和磁盘 I/O 的情况：

```
root@db 14:39:  [zs]> show profile block io,cpu for query 1;
```

```
+------------------------------+----------+----------+------------+------------+-------------+
| Status                       | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+------------------------------+----------+----------+------------+------------+-------------+
| starting                     | 0.000069 | 0.000000 |   0.000000 |          0 |           0 |
| checking permissions         | 0.000010 | 0.000000 |   0.000000 |          0 |           0 |
| init                         | 0.000006 | 0.000000 |   0.000000 |          0 |           0 |
| Opening tables               | 0.000084 | 0.000000 |   0.000000 |          0 |           0 |
| setup                        | 0.000030 | 0.000000 |   0.000000 |          0 |           0 |
| creating table               | 0.004851 | 0.000000 |   0.001000 |          0 |          24 |
| After create                 | 0.000165 | 0.000000 |   0.000000 |          0 |           0 |
| System lock                  | 0.000011 | 0.000000 |   0.000000 |          0 |           0 |
| preparing for alter table    | 0.006230 | 0.000000 |   0.000000 |          0 |           8 |
| altering table               | 0.406017 | 0.065990 |   0.009998 |          0 |        3912 |
| committing alter table to stor | 0.015046 | 0.007999 |   0.001000 |          0 |          40 |
| end                          | 0.000029 | 0.000000 |   0.000000 |          0 |           0 |
| query end                    | 0.004285 | 0.000000 |   0.000000 |          0 |          16 |
| Waiting for semi-sync ACK from | 0.000017 | 0.000000 |   0.000000 |          0 |           0 |
| query end                    | 0.000018 | 0.000000 |   0.000000 |          0 |           0 |
| closing tables               | 0.000012 | 0.000000 |   0.000000 |          0 |           0 |
| freeing items                | 0.000137 | 0.000000 |   0.001000 |          0 |           0 |
| cleaning up                  | 0.000017 | 0.000000 |   0.000000 |          0 |           0 |
+------------------------------+----------+----------+------------+------------+-------------+
```

（18）在 MySQL 数据库中，可以通过使用 show global status 命令来查看数据库的运行状态，通过得出的数值优化 MySQL 运行效率。

接下来细说一下 show global status 输出的重点参数项。

- Aborted_clients：由于客户端没有正确关闭连接导致客户端终止而中断的连接数。比如连接时间超过设置的连接超时时间（wait_timeout 和 interactive_timeout）就会退出，会使 Aborted_clients 计算器加 1。
- Aborted_connects：试图连接到 MySQL 服务器而失败的连接数。例如，输错密码，无法连接数据库，就会在 Aborted_connects 该参数值上加 1。
- Binlog_cache_disk_use：使用临时二进制日志缓存但超过 binlog_cache_size 值并使用临时文件来保存事务中语句的事务数量。
- Binlog_cache_use：使用临时二进制日志缓存的事务数量。
- Binlog_stmt_cache_disk_use：当非事务语句使用二进制日志缓存，但是超出

binlog_stmt_cache_size 的大小时，就会使用一个临时文件来存放这些语句。

- Binlog_stmt_cache_use：使用二进制日志缓存文件的非事务语句数量。
- Created_tmp_disk_tables：服务器执行语句时在硬盘上自动创建的临时表的数量。

注：排序过程中，内存不够用，就需要在磁盘上创建临时表来完成排序工作。

- Created_tmp_tables：服务器执行语句时自动创建的内存中的临时表的数量。如果 Created_tmp_disk_tables 参数值较大，可能要增加 tmp_table_size 值，使临时表基于内存而不基于磁盘。
- Handler_commit：内部提交的语句数量。执行 update 或者 delete 语句使该值数量加 2，执行 select 语句使该值加 1。

```
root@db 14:11:  [zs]> show global status like 'Handler_commit%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Handler_commit  | 37734 |
+-----------------+-------+
1 row in set (0.00 sec)
```

```
root@db 14:11:  [zs]> update tt set score=81 where score=90;
Query OK, 2218 rows affected (0.44 sec)
Rows matched: 2218  Changed: 2218  Warnings: 0

root@db 14:12:  [zs]> show global status like 'Handler_commit%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Handler_commit  | 37736 |
+-----------------+-------+
1 row in set (0.01 sec)
```

```
root@db 14:12:  [zs]> show global status like 'Handler_commit%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Handler_commit  | 37736 |
+-----------------+-------+
1 row in set (0.01 sec)

root@db 14:12:  [zs]> select * from tt limit 1;
+----+------+-------+
| id | name | score |
+----+------+-------+
| 20 | cc   |    50 |
+----+------+-------+
1 row in set (0.00 sec)

root@db 14:14:  [zs]> show global status like 'Handler_commit%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Handler_commit  | 37737 |
+-----------------+-------+
```

- Handler_rollback：内部 ROLLBACK 语句的数量。
- Handler_read_rnd：根据固定位置读一行的请求数。如果正执行大量查询并需要对结果进行排序，则该值较高。你可能使用了大量需要 MySQL 扫描整个表的查询或连接没有正确使用索引。比如针对分页查询优化时，使用 select …limit 100,1。
- Handler_read_rnd_next：在数据文件中读下一行的请求数。如果正进行大量的表扫描操作，该值较高，通常说明表索引不正确或写入的查询没有利用索引。
- Handler_read_first：索引中第一条被读的次数。如果较高，它建议服务器正执行大量全索引扫描。例如，使用 select 字段 A from table，假定字段 A 有索引。
- Handler_read_key：根据索引读一行的请求数。如果较高，说明查询和表的索引正确。

下面介绍 MySQL InnoDB 存储引擎中三个重要的等待事件。

- innodb_buffer_pool_wait_free：一般情况下，通过后台向 InnoDB 缓冲池写。但如果需要读或创建页，并且没有干净的页可用，则它还需要先等待页面清空，该计数器对等待实例进行记数。如果已经适当设置了缓冲池大小，该值应小。

```
root@db 14:37:  [zs]> show global status like 'Innodb_buffer_pool_wait_free';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| Innodb_buffer_pool_wait_free | 0     |
+--------------------------------+-------+
1 row in set (0.00 sec)
```

如果 innodb_buffer_pool_wait_free 参数值大于 0，就需要添加 innodb buffer pool 的大小了。

- innodb_log_waits：必须等待的时间，因为日志缓冲区太小，在继续前必须先清空它。

```
root@db 14:48:  [zs]> show global status like 'Innodb_log_waits';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Innodb_log_waits | 0     |
+-----------------+-------+
1 row in set (0.00 sec)
```

如果 innodb_log_waits 参数不为 0，则证明当前的 redo log buffer size 太小了，需要增大。

- innodb row lock waits：当前等待行锁的数量。如果该参数值较高，需要通过 show engine innodb status\G 或者利用 information_schema 库下 INNODB_LOCKS、INNODB_TRX、INNODB_LOCK_WAITS 进一步分析锁等待过程。
- Open_tables：当前打开表的数量。
- Opened_tables：已经打开表的数量。

```
root@db 15:09:  [sys]> show global status like 'open%tables%';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 50    |
| Opened_tables | 161   |
+---------------+-------+
2 rows in set (0.00 sec)
```

如果 Opened_tables 的数值非常大，说明 table_open_cache 值太小，导致要频繁地“open table”，可以查看当前的 table_open_cache 设置：

```
root@db 15:12:  [sys]> show variables like '%table_open_cache%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| table_open_cache          | 1024  |
| table_open_cache_instances | 64    |
+---------------------------+-------+
2 rows in set (0.00 sec)
```

MySQL 5.6 之后多了一个 table_open_cache_instances 参数。该参数代表表缓存实例数，作用就是对 table cache 进行划分，减少锁竞争。

- Threads_cached：线程缓存内的线程的数量。
- Threads_connected：当前打开的连接的数量。
- Threads_created：创建用来处理连接的线程数。如果 Threads_created 较大，可能要增加 thread_cache_size 值。
- Threads_running：激活的（非睡眠状态）线程数。

```
root@db 15:18:  [sys]> show global status like 'thread%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Threads_cached    | 1     |
| Threads_connected | 2     |
| Threads_created   | 3     |
| Threads_running   | 2     |
+-------------------+-------+
4 rows in set (0.00 sec)
```

- Select_full_join：没有使用索引的连接的数量。如果该值不为 0，应仔细检查表的索引。
- Select_full_range_join：在引用的表中使用范围搜索的连接的数量。
- Select_range：在第一个表中使用范围的连接的数量。一般情况下不是关键问题，即使该值相当大。
- Select_range_check：在每一行数据后对键值进行检查，检查不带键值的连接的数量。如果不为 0，应仔细检查表的索引。
- Select_scan：对第一个表进行完全扫描的连接的数量。

```
root@db 15:40:  [sys]> show global status like 'select%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Select_full_join       | 0     |
| Select_full_range_join | 0     |
| Select_range           | 1     |
| Select_range_check     | 0     |
| Select_scan            | 290   |
+------------------------+-------+
5 rows in set (0.00 sec)
```

- Sort_merge_passes：排序算法已经执行的合并的数量。如果这个变量值较大，应考虑增加 sort_buffer_size 系统变量的值。
- Sort_range：在范围内执行的排序的数量。
- Sort_rows：已经排序的行数。
- Sort_scan：通过扫描表完成的排序的数量。

```
root@db 15:37:  [sys]> show global status like 'sort%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Sort_merge_passes | 0     |
| Sort_range        | 0     |
| Sort_rows         | 0     |
| Sort_scan         | 2     |
+-------------------+-------+
4 rows in set (0.00 sec)
```

17.5 整体管理优化总结

通过本章的学习，我们知道想要做好 MySQL 的优化工作，绝对不是简简单单地调整几个参数，加强点硬件设备就能解决问题。需要通过表设计、Linux 操作系统、MySQL 数据库配置和硬件四个维度来真正了解怎样才能做好 MySQL 的优化。作为 DBA 也绝对不要在一个新项目上线的后期，当出现数据库性能问题时才想起来该做 MySQL 的优化工作。DBA 要在项目初期，设计表的过程中就参与到业务设计中，把一些优化工作提前做好。遇到性能问题，也不必慌张。第一步先要确认问题出在哪里，之后了解清楚问题的瓶颈点是什么，通过技术商议整理出一套可执行的解决问题的方案，然后需要对技术方案进行实战测试演练，验证方案的可执行性。如果执行没有问题，就在线上环境运行此方案。最后一定要做好相应问题处理过程的记录工作，以防下次再遇到同类问题时束手无策。知识的累积不是一两天完成的，我们需要通过每天的努力与成长，逐渐积累起知识的堡垒，向着最终的目标迈进。

第 6 部分　至尊星耀篇

现在进入了学习 MySQL 的冲刺阶段，我们一定不要忽视每一个知识点的细节。不积跬步，无以至千里；不积小流,无以成江海。在 MySQL 数据库知识的体系中怎么能少得了对 MySQL 监控的学习，监控可以帮助我们排查 MySQL 的性能瓶颈，方便我们进行调优工作。本部分主要学习如何通过部署 Lepus 来监控 MySQL 数据库，还会进行 MySQL 版本升级的实战演练。

第 18 章 Lepus 之 MySQL 监控

随着企业业务的扩展，数据量日益剧增，性能分析和监控预警工作变得越来越重要。有些人总觉得监控没有意义，还得消耗资源成本，不能带来一点实用价值。无论我们做什么行业，这种想法都是不对的。不要总着眼于眼前这点利益，要把眼光放得更长远一些，这样才有可能带来更大的收益。监控的意义主要可以分为两个方面，第一个就是它可以监控数据库服务和业务的可用性，用来保证线上业务 7×24 小时的正常运行。但我们要知道服务的可用性并不代表数据的准确，所以另一个意义就是监控数据的可靠性。监控工作对我们来说至关重要。

工欲善其事，必先利其器。使用什么监控工具来监控 MySQL 数据库比较方便呢？用得比较多的无非就是 Zabbix、Cacti 等监控工具。这里推荐一款在生产环境中经常使用的监控工具 Lepus（天兔），它是一个由 Python+PHP 开发的数据库企业级监控系统。

18.1 Lepus 简介

Lepus 数据库企业监控系统是一套针对互联网企业开发的企业数据库监控管理系统，企业通过Lepus可以对数据库的实时健康和各种性能指标进行全方位的监控。目前已经支持MySQL、Oracle、MongoDB、Redis 数据库的全面监控。Lepus 可以在数据库出现故障或者潜在性能问题时，根据用户设置及时将数据库的异常进行报警，通知数据库管理员以进行处理和优化，帮助企业解决数据库性能监控问题，及时发现性能和瓶颈，避免由数据库潜在问题造成的直接经济损失。Lepus 能够查看各种实时性能状态指标，并且对监控、性能数据进行统计分析，从运维

者到决策者多个层面的视角查看相关报表。帮助决策者对未来数据库容量进行更好的规划，从而降低了硬件成本。

Lepus 目前主要有以下功能和特性（摘自 Lepus 官网）：

- 无须 Agent，远程监视云中数据库；
- Web 直观的管理和监视数据库；
- 实时 MySQL 健康监视和告警；
- 实时 MySQL 复制监视和告警；
- 实时 MySQL 资源监视和分析；
- 实时 MySQL 缓存等性能监视；
- 实时 InnoDB I/O 性能监控；
- MySQL 表空间增长趋势分析；
- 可视化 MySQL 慢查询在线分析；
- MySQL 慢查询自动推送功能；
- MySQL AWR 在线性能分析；
- 实时 Oracle 健康监控和报警；
- 实时 Oracle 表空间使用监控；
- 实时 Oracle 性能监控；
- 实时 MongoDB 健康监控和报警；
- 实时 MongoDB 索引性能监控；
- 实时 MongoDB 内存使用监控；
- 实时 Redis 健康监控和报警；
- 实时 Redis 性能监控；
- 实时 OS 主机 CPU/内存/磁盘/网络/IO 监控；
- 可视化告警系统，邮件发送告警，短信接口支持；
- 严格的权限认证系统；
- 丰富的健康性能分析图表；
- 多维的对比和性能分析；

更多详细资料可以访问 Lepus 官网地址 http://www.lepus.cc，或者查看在线手册 http://www.lepus.cc/manual/index/。

接下来学习 Lepus 的实战部署过程。

18.2 实战部署

环境配置如表 18-1 所示。

表 18-1 环境配置

192.168.56.100	主库	被监控服务器
192.168.56.101	从库	被监控服务器
192.168.56.103	部署 lepus	监控服务器

第一步，安装 Xampp

Lepus 的安装需要 LAMP（Linux+Apache+MySQL+PHP）环境，这里我们不推荐使用 Yum 和 Rpm 安装方式，建议使用 Xampp 集成环境包进行安装。Xampp 是一个可靠的稳定的 LAMP 套件，目前已被诸多公司用于生产服务器的部署，目前 Lepus 的开发环境、测试环境以及线上官网的 Web 环境都运行在 Xampp 环境下面，并且一直都很稳定。

Xampp 的下载地址：https://www.apachefriends.org/download.html。

下载完成之后，执行如下安装命令，调出图形安装界面：

```
chmod +x xampp-linux-x64-1.8.2-5-installer.run
./xampp-linux-x64-1.8.2-5-installer.run
```

一路点击“next”，直到出现安装完成的界面提示。

第二步，启动 xampp 服务

命令如下：

/opt/lampp/lampp start，因为默认安装在/opt/lampp 目录下。

```
[root@proxysql lepus_v3.7]# /opt/lampp/lampp start
Starting XAMPP for Linux 1.8.2-5...
XAMPP: Starting Apache...ok.
XAMPP: Starting MySQL...ok.
XAMPP: Starting ProFTPD...ok.
```

各服务显示 ok，代表启动成功。

第三步，安装 Python 基础模块

MySQLdb 为 Python 连接和操作 MySQL 的类库，如果准备使用 Lepus 系统监控 MySQL 数据库，那么该模块必须安装。

Python 模块下载地址：

http://cdn.lepus.cc/cdncache/software/MySQLdb-python.zip。

解压 Python 软件包：

```
unzip MySQLdb-python.zip
```

进入解压之后的目录下：

```
cd MySQLdb1-master/
```

确认 mysql_config 命令的位置：

```
which mysql_config
```

```
[root@proxysql MySQLdb1-master]# which mysql_config
/opt/lampp/bin/mysql_config
```

编辑 site.cfg 文件 vi site.cfg。

加入 mysql_config = /opt/lampp/bin/mysql_config：

```
# The path to mysql_config.
# Only use this if mysql_config is not on your PATH, or you have some weird
# setup that requires it.

mysql_config = /opt/lampp/bin/mysql_config
```

执行安装 Python 模块的脚本之前，需要先安装好如下这些包：

```
yum -y install openssl-devel
yum -y install python-devel
yum -y install gcc
rpm -ivh libffi-devel-3.0.5-3.2.el6.x86_64.rpm
```

```
[root@node3 ~]# rpm -ivh libffi-devel-3.0.5-3.2.el6.x86_64.rpm
warning: libffi-devel-3.0.5-3.2.el6.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID c105b9de: NOKEY
Preparing...                ########################################### [100%]
   1:libffi-devel           ########################################### [100%]
```

```
yum -y install urpmi xterm
```

安装 Python 模块的脚本：

```
[root@node3 MySQLdb1-master]# python setup.py build
```

输出结果：

```
running build
running build_py
copying MySQLdb/release.py -> build/lib.linux-x86_64-2.6/MySQLdb
running build_ext
building '_mysql' extension
    gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2
-fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -mtune=generic
-D_GNU_SOURCE -fPIC -fwrapv -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2
-fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -mtune=generic
-D_GNU_SOURCE -fPIC -fwrapv -fPIC -Dversion_info=(1,2,4,'final',1)
-D__version__=1.2.4 -I/usr/include/mysql -I/usr/include/python2.6 -c _mysql.c
-o build/temp.linux-x86_64-2.6/_mysql.o -g -pipe -Wp,-D_FORTIFY_SOURCE=2
-fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -D_GNU_SOURCE
-D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -fno-strict-aliasing -fwrapv -fPIC
-DUNIV_LINUX -DUNIV_LINUX
    In file included from /usr/include/mysql/my_config.h:14,
                     from _mysql.c:44:
    /usr/include/mysql/my_config_x86_64.h:1082:1: warning: "HAVE_WCSCOLL"
redefined
    In file included from /usr/include/python2.6/pyconfig.h:6,
                     from /usr/include/python2.6/Python.h:8,
                     from _mysql.c:29:
    /usr/include/python2.6/pyconfig-64.h:808:1: warning: this is the location
of the previous definition
    gcc -pthread -shared build/temp.linux-x86_64-2.6/_mysql.o
-L/usr/lib64/mysql -L/usr/lib64 -lmysqlclient_r -lz -lpthread -lcrypt -lnsl -lm
-lpthread -lssl -lcrypto -lpython2.6 -o build/lib.linux-x86_64-2.6/_mysql.so

    [root@node3 MySQLdb1-master]# python setup.py install
```

输出结果：

```
running install
running bdist_egg
running egg_info
creating MySQL_python.egg-info
writing MySQL_python.egg-info/PKG-INFO
```

```
writing top-level names to MySQL_python.egg-info/top_level.txt
writing dependency_links to MySQL_python.egg-info/dependency_links.txt
writing manifest file 'MySQL_python.egg-info/SOURCES.txt'
reading manifest file 'MySQL_python.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'MySQL_python.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_py
copying MySQLdb/release.py -> build/lib.linux-x86_64-2.6/MySQLdb
running build_ext
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/MySQLdb
copying build/lib.linux-x86_64-2.6/MySQLdb/release.py ->
build/bdist.linux-x86_64/egg/MySQLdb
copying build/lib.linux-x86_64-2.6/MySQLdb/connections.py ->
build/bdist.linux-x86_64/egg/MySQLdb
copying build/lib.linux-x86_64-2.6/MySQLdb/times.py ->
build/bdist.linux-x86_64/egg/MySQLdb
copying build/lib.linux-x86_64-2.6/MySQLdb/converters.py ->
build/bdist.linux-x86_64/egg/MySQLdb
copying build/lib.linux-x86_64-2.6/MySQLdb/__init__.py ->
build/bdist.linux-x86_64/egg/MySQLdb
creating build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/FLAG.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/FIELD_TYPE.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/REFRESH.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/ER.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/__init__.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/CR.py ->
build/bdist.linux-x86_64/egg/MySQLdb/constants
copying build/lib.linux-x86_64-2.6/MySQLdb/constants/CLIENT.py ->
```

```
build/bdist.linux-x86_64/egg/MySQLdb/constants
    copying build/lib.linux-x86_64-2.6/MySQLdb/cursors.py ->
build/bdist.linux-x86_64/egg/MySQLdb
    copying build/lib.linux-x86_64-2.6/_mysql_exceptions.py ->
build/bdist.linux-x86_64/egg
    copying build/lib.linux-x86_64-2.6/_mysql.so ->
build/bdist.linux-x86_64/egg
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/release.py to
release.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/connections.py to
connections.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/times.py to times.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/converters.py to
converters.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/__init__.py to
__init__.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/constants/FLAG.py to
FLAG.pyc
    byte-compiling
build/bdist.linux-x86_64/egg/MySQLdb/constants/FIELD_TYPE.py to
FIELD_TYPE.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/constants/REFRESH.py
to REFRESH.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/constants/ER.py to
ER.pyc
    byte-compiling
build/bdist.linux-x86_64/egg/MySQLdb/constants/__init__.py to __init__.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/constants/CR.py to
CR.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/constants/CLIENT.py
to CLIENT.pyc
    byte-compiling build/bdist.linux-x86_64/egg/MySQLdb/cursors.py to
cursors.pyc
    byte-compiling build/bdist.linux-x86_64/egg/_mysql_exceptions.py to
_mysql_exceptions.pyc
    creating stub loader for _mysql.so
    byte-compiling build/bdist.linux-x86_64/egg/_mysql.py to _mysql.pyc
    creating build/bdist.linux-x86_64/egg/EGG-INFO
```

```
    copying MySQL_python.egg-info/PKG-INFO ->
build/bdist.linux-x86_64/egg/EGG-INFO
    copying MySQL_python.egg-info/SOURCES.txt ->
build/bdist.linux-x86_64/egg/EGG-INFO
    copying MySQL_python.egg-info/dependency_links.txt ->
build/bdist.linux-x86_64/egg/EGG-INFO
    copying MySQL_python.egg-info/top_level.txt ->
build/bdist.linux-x86_64/egg/EGG-INFO
    writing build/bdist.linux-x86_64/egg/EGG-INFO/native_libs.txt
    zip_safe flag not set; analyzing archive contents...
    creating dist
    creating 'dist/MySQL_python-1.2.4-py2.6-linux-x86_64.egg' and adding
'build/bdist.linux-x86_64/egg' to it
    removing 'build/bdist.linux-x86_64/egg' (and everything under it)
    Processing MySQL_python-1.2.4-py2.6-linux-x86_64.egg
    creating
/usr/lib64/python2.6/site-packages/MySQL_python-1.2.4-py2.6-linux-x86_64.egg
    Extracting MySQL_python-1.2.4-py2.6-linux-x86_64.egg to
/usr/lib64/python2.6/site-packages
    Adding MySQL-python 1.2.4 to easy-install.pth file

    Installed
/usr/lib64/python2.6/site-packages/MySQL_python-1.2.4-py2.6-linux-x86_64.egg
    Processing dependencies for MySQL-python==1.2.4
    Finished processing dependencies for MySQL-python==1.2.4
```

第四步，安装 Lepus 采集器

下载 Lepus 采集器：

http://www.lepus.cc/soft/index。

解压 Lepus 软件包：

```
unzip  Lepus 3.7.zip
```

第五步，创建监控数据库并授权

监控数据库上需要创建的权限，命令如下：

```
create database lepus default character set utf8;
grant select,insert,update,delete,create on lepus.* to
```

```
'lepus_user'@'192.168.56.%' identified by 'root123';
```

被监控库需要创建的权限，命令如下：

```
    create user 'lepus_monitor'@'192.168.56.%' identified by 'root123';
    grant select,super,process,reload,show databases,replication client on *.*
to'lepus_monitor'@'192.168.56.%';
    flush privileges;
```

第六步，进入 lepus_v3.7 目录下，导入初始化数据

导入表结构和表数据文件：

```
[root@proxysql sql]# ll
total 116
-rw-r--r--. 1 root root 34988 Feb  9  2015 lepus_data.sql
-rw-r--r--. 1 root root 74677 Feb  9  2015 lepus_table.sql
```

```
/opt/lampp/bin/mysql -uroot  lepus < lepus_table.sql
/opt/lampp/bin/mysql -uroot  lepus < lepus_data.sql
```

第七步，安装 Lepus 程序

切换到安装包 Python 目录下执行安装操作。

首先给 install.sh 脚本授权，然后执行脚本。

授权命令：chmod +x install.sh。

执行过程：

```
[root@proxysql python]# ./install.sh
[note] lepus will be install on basedir: /usr/local/lepus
[note] /usr/local/lepus directory does not exist,will be created
[note] /usr/local/lepus directory created success.
[note] wait copy files.......
[note] change script permission.
[note] create links.
[note] install complete.
```

修改配置文件 vi /usr/local/lepus/etc/config.ini：

```
###监控机MySQL数据库连接地址###
[monitor_server]
host="192.168.56.103"
port=3306
user="lepus_user"
passwd="root123"
dbname="lepus"
```

- user：数据库用户名，上面设置监控权限时创建的用户名是 lepus_user。
- passwd：数据库密码，上面设置监控权限时创建的密码是 root123。

启动 Lepus 服务：

```
[root@proxysql lepus]# lepus start
nohup: appending output to `nohup.out'
lepus server start success!
```

Lepus 日志的存放位置：

```
[root@proxysql ~]# tail -f /usr/local/lepus/logs/lepus.log
2017-11-13 14:34:41 [INFO] check os controller finished.
2017-11-13 14:34:44 [INFO] lepus controller start.
2017-11-13 14:34:45 [INFO] check mysql controller started.
2017-11-13 14:34:45 [WARNING] check mysql: not found any servers
2017-11-13 14:34:45 [INFO] check mysql controller finished.
2017-11-13 14:34:57 [INFO] check os controller started.
2017-11-13 14:34:57 [WARNING] check os: not found any servers
2017-11-13 14:34:57 [INFO] check os controller finished.
2017-11-13 14:35:00 [INFO] alarm controller started.
2017-11-13 14:35:00 [INFO] alarm controller finished.
```

第八步，配置 Web 管理平台

```
mkdir /opt/lampp/htdocs/lepus
```

进入 lepus_v3.7/php 目录下，复制所有内容到/opt/lampp/htdocs/lepus 目录。

命令如下：

```
cp -rf * /opt/lampp/htdocs/lepus
```

编辑 database.php 文件，修改监控主机的 IP 地址、用户名密码：

```
vi /opt/lampp/htdocs/lepus/application/config/database.php
```

```
$db['default']['hostname'] = '192.168.56.103';
$db['default']['port']     = '3306';
$db['default']['username'] = 'lepus_user';
$db['default']['password'] = 'root123';
$db['default']['database'] = 'lepus';
$db['default']['dbdriver'] = 'mysql';
$db['default']['dbprefix'] = '';
$db['default']['pconnect'] = TRUE;
$db['default']['db_debug'] = TRUE;
$db['default']['cache_on'] = FALSE;
$db['default']['cachedir'] = '';
$db['default']['char_set'] = 'utf8';
$db['default']['dbcollat'] = 'utf8_general_ci';
$db['default']['swap_pre'] = '';
$db['default']['autoinit'] = TRUE;
$db['default']['stricton'] = FALSE;
```

第九步，修改完 database.php 之后，重启 Xampp 服务

```
[root@proxysql ~]# /opt/lampp/lampp restart
Restarting XAMPP for Linux 1.8.2-5...
XAMPP: Stopping Apache...ok.
XAMPP: Stopping MySQL...ok.
XAMPP: Stopping ProFTPD...ok.
XAMPP: Starting Apache...ok.
XAMPP: Starting MySQL...ok.
XAMPP: Starting ProFTPD...ok.
```

第十步，输入 http://192.168.56.103/lepus，登录 Lepus 的监控界面

默认的管理员账号和密码是 admin/Lepusadmin。

下面就可以实现监控 MySQL 服务器的工作了。

18.3 监控 MySQL 服务器

先添加需要被监控的 MySQL 服务器。

这里需要监控的是（192.168.56.100 和 192.168.56.101），如图 18-1、图 18-2 所示。

图 18-1 被监控的 MySQL 服务器 1

图 18-2 被监控的 MySQL 服务器 2

添加完之后会在左边栏 MySQL 监控项的健康监控中显示主机的基础信息，如图 18-3 所示。

图 18-3　监控信息

也可以通过监控图表来观察 MySQL 的连接情况、TPS、QPS、每秒执行 DML 语句等监控信息，如图 18-4～图 18-7 所示。

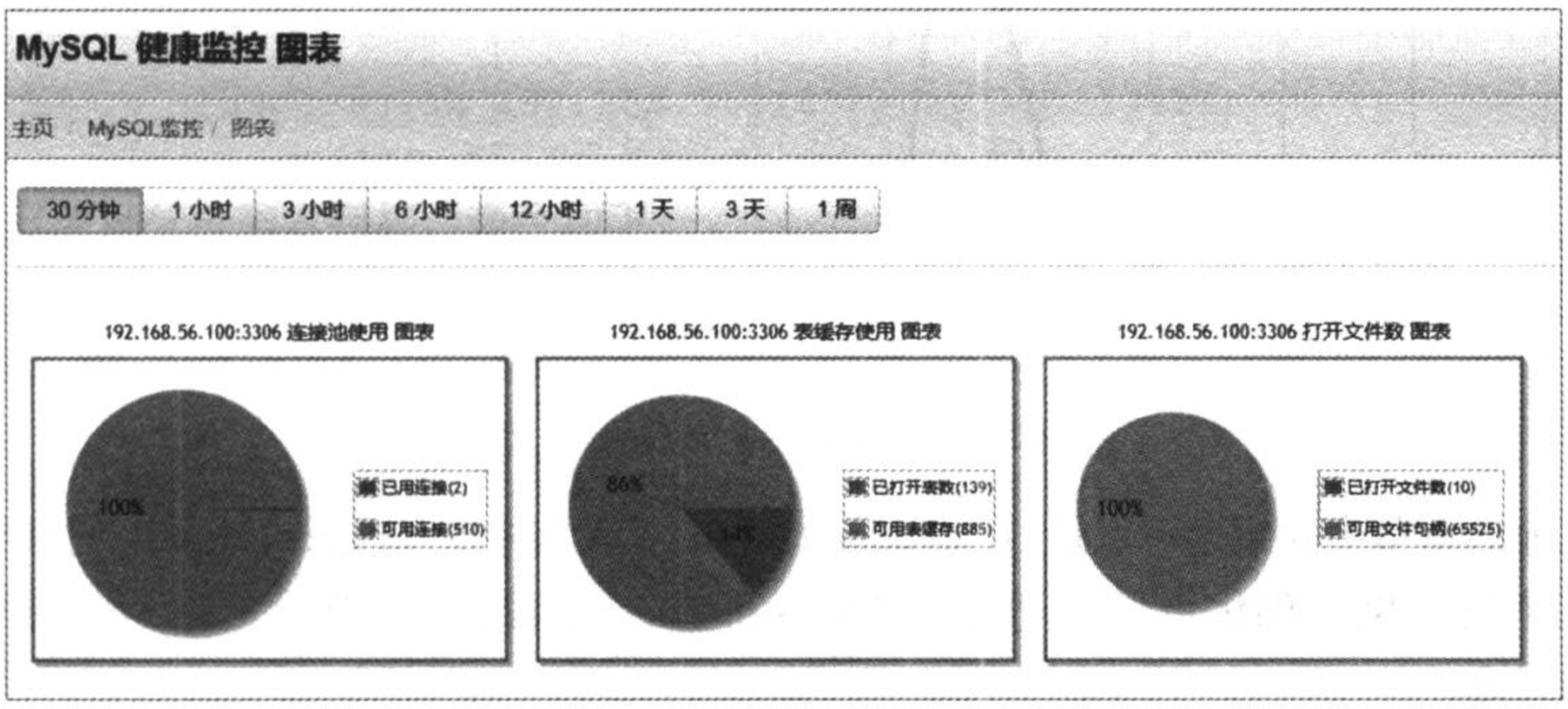


图 18-4　监控的连接情况

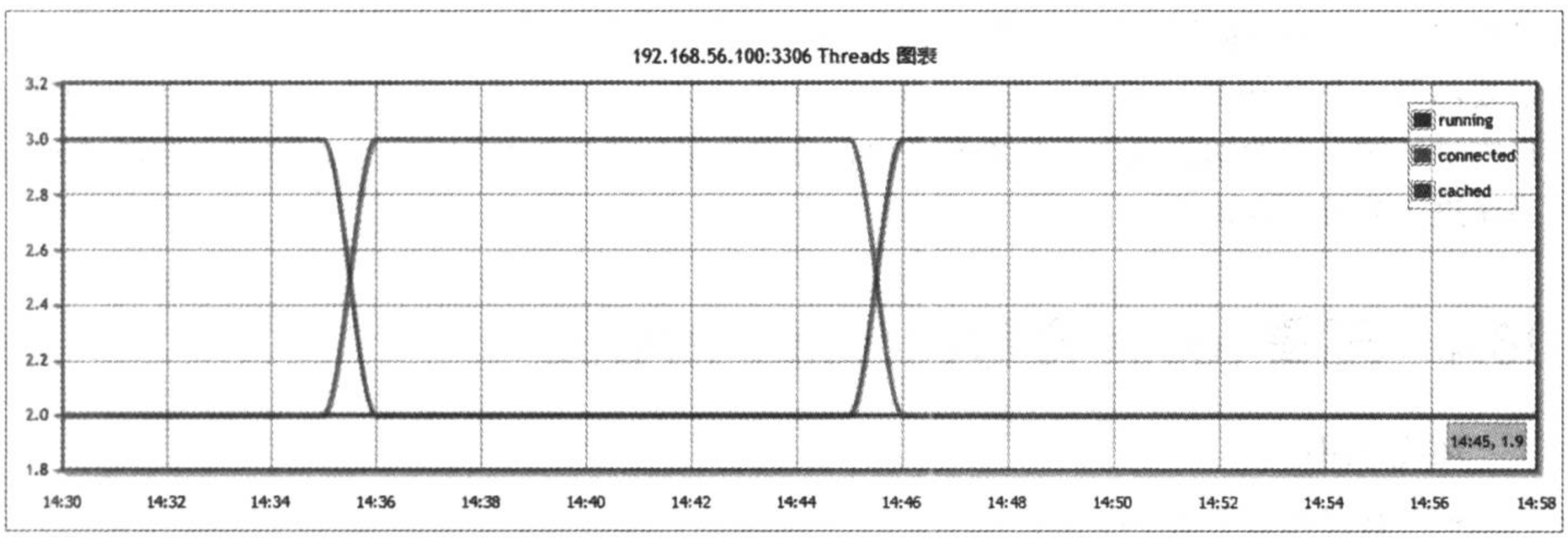


图 18-5　监控的 Threads 信息

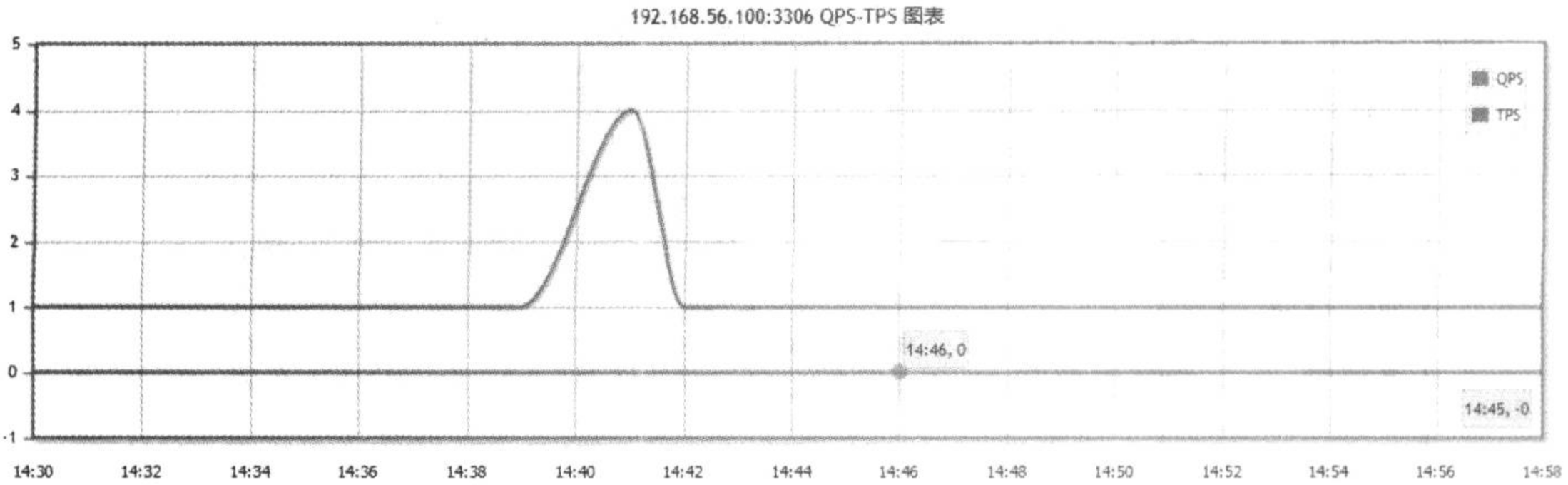


图 18-6　监控的 TPS 信息

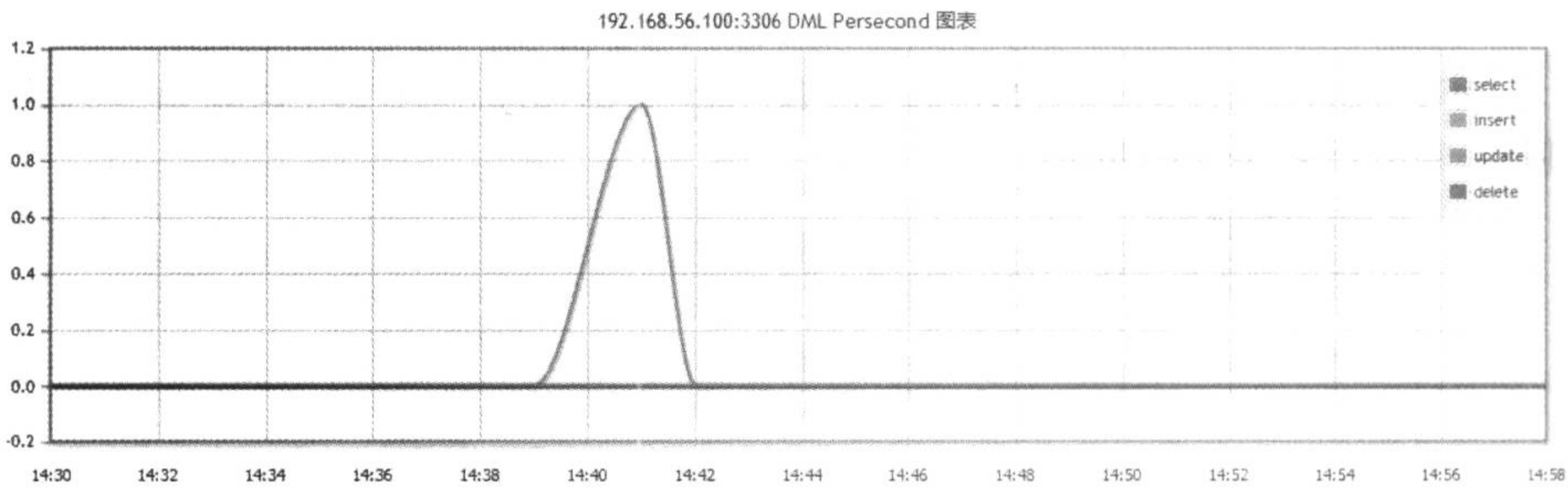


图 18-7　监控的 DML Persecond 信息

在之前的章节中介绍过可以通过 percona-toolkit 工具集中的 pt-query-digest 捕获线上的慢 SQL 语句来进行 SQL 的优化工作。在学完 Lepus 部署之后，看看如何利用 Lepus 来监控 MySQL 的慢查询日志。

本章将介绍生产环境中 Lepus 的慢查询分析平台是如何部署的。Lepus 的慢查询分析平台是独立于监控系统的模块，该功能需要使用 percona-toolkit 工具来采集和记录慢查询日志，并且需要部署一个 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，并对慢查询日志进行按小时的切割操作，收集慢查询日志的数据到监控机数据库。之后就可以通过 Lepus 系统进行分析慢查询了。

18.4　部署 Lepus 慢查询分析平台实战

环境介绍：

- 被监控服务器 192.168.56.100；
- 监控服务器 192.168.56.103。

第一步，需要在被监控服务器上安装 percona-toolkit 工具。

先用 Yum 安装环境需要的软件包：

```
yum -y install perl-IO-Socket-SSL
yum -y install perl-DBI
yum -y install perl-DBD-MySQL
yum -y install perl-Time-HiRes
```

解压下载的 percona-toolkit 软件包，解压目录放在/usr/local 下：

```
tar -zxvf percona-toolkit-3.0.3_x86_64.tar.gz -C /usr/local
```

第二步，从监控服务器的/usr/local/lepus/client/mysql/目录复制 Lepus 提供的慢查询分析脚本 lepus_slowquery.sh 到被监控服务器上。命令如下：

```
scp lepus_slowquery.sh 192.168.56.100:/usr/local/sbin/
```

之后需要对分析脚本授权，命令如下：

```
chmod +x /usr/local/sbin/lepus_slowquery.sh
```

第三步，修改脚本里面的配置信息。

```
vim /usr/local/sbin/lepus_slowquery.sh
#!/bin/bash
#*************************************************************#
# ScriptName: /usr/local/sbin/lepus_slowquery.sh
# Create Date: 2014-03-25 10:01
# Modify Date: 2014-03-25 10:01
#*************************************************************#
#config lepus database server
lepus_db_host="192.168.56.103"
lepus_db_port=3306
lepus_db_user="lepus_user"
lepus_db_password="root123"
lepus_db_database="lepus"

#config mysql server
```

```
mysql_client="/usr/local/mysql/bin/mysql"
mysql_host="192.168.56.100"
mysql_port=3306
mysql_user="lepus_monitor"
mysql_password="root123"

#config slowqury
slowquery_dir="/data/mysql/"
    slowquery_long_time=0.05
    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/percona-toolkit-3.0.3/bin/pt-query-digest"

    #config server_id
    lepus_server_id=1

    #collect mysql slowquery log into lepus database
    $pt_query_digest --user=$lepus_db_user --password=$lepus_db_password
--port=$lepus_db_port --review
h=$lepus_db_host,D=$lepus_db_database,t=mysql_slow_query_review  --history
h=$lepus_db_host,D=$lepus_db_database,t=mysql_slow_query_review_history
--no-report --limit=100% --filter=" \$event->{add_column} =
length(\$event->{arg}) and \$event->{serverid}=$lepus_server_id "
$slowquery_file > /tmp/lepus_slowquery.log

    ##### set a new slow query log ###########
    tmp_log=`$mysql_client -h$mysql_host -P$mysql_port -u$mysql_user
-p$mysql_password -e "select
concat('$slowquery_dir','slowquery_',date_format(now(),'%Y%m%d%H'),'.log');"
|grep log|sed -n -e '2p'`

    #config mysql slowquery
    $mysql_client -h$mysql_host -P$mysql_port -u$mysql_user -p$mysql_password
-e "set global slow_query_log=1;set global
long_query_time=$slowquery_long_time;"
    $mysql_client -h$mysql_host -P$mysql_port -u$mysql_user -p$mysql_password
-e "set global slow_query_log_file = '$tmp_log'; "
```

```
#delete log before 7 days
cd $slowquery_dir
/usr/bin/find ./ -name 'slowquery_*' -mtime +7|xargs rm -rf ;

####END####
```

下面详细介绍脚本中配置参数的含义。

- #config lepus database server：监控服务器中的配置信息。
- lepus_db_host="192.168.56.103"：监控服务器的 IP 地址。
- lepus_db_port=3306：监控服务器的数据库端口号。
- lepus_db_user="lepus_user"：上面创建的监控服务器的用户名。
- lepus_db_password="root123"：上面创建的监控服务器的密码。
- lepus_db_database="lepus"：上面创建的监控服务器的数据库。
- #config mysql server：被监控服务器中的配置信息。
- mysql_client="/usr/local/mysql/bin/mysql"：客户端命令的绝对路径。
- mysql_host="192.168.56.100"：被监控的 MySQL 服务器。
- mysql_port=3306：被监控的 MySQL 服务器的端口号。
- mysql_user="lepus_monitor"：上面创建的被监控服务器用户名。
- mysql_password="root123"：上面创建的被监控服务器密码。
- #config slowqury：代表慢查询的配置信息。
- slowquery_dir="/data/mysql/"：慢查询日志的目录位置。
- slowquery_long_time=0.05：慢查询日志的时间，单位是秒。
- 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}'`：获取慢查询日志的名称（/data/mysql/slow.log）。
- pt_query_digest="/usr/local/percona-toolkit-3.0.3/bin/pt-query-digest"：被监控服务器的 pt-query-digest 的命令所在位置。
- #config server_id：代表被监控服务器的 ID 号。
- lepus_server_id=1：ID 号从 Lepus 监控图中获取，如图 18-8 所示。

图 18-8　监控信息

> 注：编辑完脚本的内容之后，需要改变当前的文件格式为 unix，否则无法执行脚本。在 vi 中执行 set fileformat=unix 命令。

第四步，脚本配置成功之后，就可以加入定时任务了。每 10 分钟进行一次采集慢 SQL 过程。

```
*/10 * * * * sh /usr/local/sbin/lepus_slowquery.sh > /dev/null 2>&1
```

从监控图中可以获取到慢 SQL 语句的信息，如图 18-9、图 18-10 所示。

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
8661637915225417606	+set global slow_query_log_file = ?;	zs	lepus_monitor	2017-11-14 07:36:01	11	0.224	0.072	0.301	0	0	0
17124965243867073762	+update tt set score=? where score=?;	zs	root	2017-11-14 06:36:28	5	0.235	0.136	0.304	0.0005	0.00010	0.00012
15221788837014604330	+select * from tt;	zs	root	2017-11-14 06:36:11	5	0.145	0.071	0.251	0.0804	0.00009	0.03970
14181556060046122434	+select count(*) from tt;	zs	root	2017-11-14 06:14:34	1	0.149	0.149	0.149	0.0001	0.00010	0.00010

图 18-9　慢 SQL 语句的信息 1

数据库	zs		用户	lepus_monitor		
校验值	8661637915225417606		次数	5		
首次出现	2017-11-14 06:22:01		最近时间	2017-11-14 07:36:01		
	set global slow_query_log_file = ?					
	set global slow_query_log_file = '/data/mysql/slowquery_2017111414.log'					
查询时间	Query_time_sum	Query_time_min	Query_time_max	Query_time_pct_95	Query_time_stddev	Query_time_median
	1.09505	0.072802	0.301822	0.292981	0.0841761	0.241036
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	0	0	0	0	0
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
	0	0	0	0	0	0

图 18-10　慢 SQL 语句的信息 2

18.5　监控总结

在监控数据库的过程中，我们只需关注那几个重点项就可以了，不可能面面俱到。系统层面监控项必须包括 CPU、I/O、内存。数据库层面的监控需要从 MySQL 的可用性和性能上综合考虑。当然在公司中最主要的还是业务层面的监控，比如业务状态是否正常，TPS 和 QPS 的量值等。有时虽然数据库的状态是“ok”，但业务无法响应，那么一切监控工作都是没有任何意义的。合理地使用监控工具也是 DBA 重要的本领之一。我们不能放过工作中任何一个细节，这样才能使业务系统稳定可靠地运行。

第 19 章

MySQL 版本升级

MySQL 版本升级这项工作对 DBA 来说是十分必要的，因为每个 MySQL 版本都有自己支持的生命周期，在超过生命周期之后，官方就不再提供服务支持。而且新版本的性能与特性比老版本的 MySQL 提升得太多了。针对新版本增加的特性，更是便于 DBA 去管理优化 MySQL 数据库的工作。但是在没有充分的测试之前，就应用到线上生产环境中也是非常危险的，因为升级可能会让应用不能正常运作，也可能引起性能问题。所以我们要经常去关注 MySQL 官方发布的信息，不要冒险去做第一个吃“螃蟹”的人，多收集资料，多测试，最后升级到一个稳定的发布版本。

在开始进行升级工作之前，别先忙于执行升级命令操作而忽略了升级过程中一个很重要的环节，那就是需要结合官方手册中 Upgrading MySQL 章节中 Changes Affecting Upgrades to 新版本的一些变化。本章学习升级中的两种方法，从 MySQL 5.6 升级到 MySQL 5.7 版本。官方文档中列出了 Configuration Changes、System Table Changes、Server Changes、InnoDB Changes、SQL Changes 这些变化可能会影响升级到 MySQL 5.7 版本。当然在这些变化中，我们着重关注 InnoDB Changes，因为如果使用 In-Place Upgrade 升级方法，后期会涉及调整 innodb_fast_shutdown 参数的值。

19.1 升级方式

MySQL 升级方式有两种，一种叫 In-Place Upgrade，另一种叫 Logical Upgrade（逻辑升级

方式）。

Logical Upgrade：利用 mysqldump 来直接导出 SQL 文件，然后导入到新库中，适应于跨大版本的升级方案，做法相对安全，并能整理表中碎片。但如果有数据量较大的库需要 mysqldump 导出，时间上的消耗就会很大，升级效率就会受到影响。

In-Place Upgrade：它的工作方法简单快速，就是直接替换掉原来版本 MySQL 的安装目录和 my.cnf 配置文件，利用 mysql_upgrade 脚本来完成系统表的升级。

> 注：跨小版本升级可以使用 in-place 这种方式。

接下来先来演练利用 in-place 的升级方案，从 MySQL 5.6.16 升级到 MySQL 5.7.14 的过程。

环境介绍

- 当前服务器 IP：192.168.56.132。
- 数据目录：/data/mysql。
- 安装目录：/usr/local/mysql。
- 配置文件：/etc/my.cnf。
- 当前版本：5.6.16。

```
[root@node3 ~]# /usr/local/mysql/bin/mysql -V
/usr/local/mysql/bin/mysql  Ver 14.14 Distrib 5.6.16, for linux-glibc2.5 (x86_64)
```

19.2　实战演练

第一步，需要将 innodb_fast_shutdown 参数设置为 0。

```
mysql> set global innodb_fast_shutdown=0;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%fast%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_fast_shutdown     | 0     |
+--------------------------+-------+
1 row in set (0.00 sec)
```

> 注：innodb_fast_shutdown 有 0、1、2 三个值。参数值为 0 代表 MySQL 关闭，InnoDB 需要完成所有的 full purge 和 merge insert buffer 操作，这个过程需要一定的时间，有时可能会花上几个小时。参数值为 1 是该参数的默认值，表示关闭 MySQL 时不完成 full

> purge 和 Merge insert buffer 操作，但是缓冲池中的脏页还是会写到磁盘中。参数值为 2 时，表示既不完成 full purge 和 Merge insert buffer 操作，也不将缓冲池中的脏页刷新到磁盘，而是将日志写入日志文件中。

第二步，关闭 MySQL 服务，命令如下：

```
mysqladmin -uroot -proot123 shutdown
```

第三步，替换 MySQL 的安装文件：

```
lrwxrwxrwx  1 mysql mysql        34 Oct 30 10:59 mysql -> mysql-5.6.16-linux-glibc2.5-x86_64
drwxr-xr-x 13 root  root       4096 Oct 30 11:04 mysql-5.6.16-linux-glibc2.5-x86_64
-rw-r--r--  1 root  root  304788904 Nov  1  2016 mysql-5.6.16-linux-glibc2.5-x86_64.tar.gz
```

需要执行 unlink mysql 命令。

解压新版本的 MySQL 软件包，然后重新做连接并赋予 MySQL 权限。命令如下：

```
tar -zxvf mysql-5.7.14-linux-glibc2.5-x86_64.tar.gz
ln -s mysql-5.7.14-linux-glibc2.5-x86_64 mysql
chown -R mysql:mysql mysql
```

```
lrwxrwxrwx 1 mysql mysql        34 Nov 12 17:34 mysql -> mysql-5.7.14-linux-glibc2.5-x86_64
drwxr-xr-x 9  7161 31415      4096 Jul 12  2016 mysql-5.7.14-linux-glibc2.5-x86_64
-rw-r--r-- 1 root  root  642694570 Nov 12 16:43 mysql-5.7.14-linux-glibc2.5-x86_64.tar.gz
```

第四步，把 MySQL 5.6 的配置文件替换成 5.7 版本的 my.cnf，并启动 MySQL 实例。

在启动过程中要注意的是需要加上--skip-grant-tables 和--skip-networking 参数，来保证没有任何的应用连接，让升级过程更加安全。

```
/usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf
--skip-grant-tables --skip-networking &
```

进程存在，已经启动成功：

```
[root@node3 ~]# ps -ef|grep mysql
root      7974  3079  0 17:50 pts/2    00:00:00 /bin/sh /usr/local/mysql/bin/mysqld_safe --defa
ults-file=/etc/my.cnf --skip-grant-tables --skip-networking
mysql     9330  7974  1 17:50 pts/2    00:00:00 /usr/local/mysql/bin/mysqld --defaults-file=/et
c/my.cnf --basedir=/usr/local/mysql --datadir=/data/mysql/ --plugin-dir=/usr/local/mysql/lib/pl
ugin --user=mysql --skip-grant-tables --skip-networking --log-error=/data/mysql/error.log --ope
n-files-limit=65535 --pid-file=/data/mysql//db.pid --socket=/tmp/mysql.sock --port=3306
```

但在错误日志中，依然可以捕捉到很多错误信息，证明还没有升级系统表信息。报错信息如下：

```
[ERROR] Native table 'performance_schema
[ERROR] Native table 'performance_schema
[ERROR] Native table 'performance_schema
[ERROR] Native table 'performance_schema
[ERROR] Native table 'performance_schema
```

第五步，升级系统表数据字典信息，命令如下：

```
/usr/local/mysql/bin/mysql_upgrade
```

输出结果：

```
Checking if update is needed.
Checking server version.
Running queries to upgrade MySQL server.
Checking system database.
mysql.columns_priv                                 OK
mysql.db                                           OK
mysql.engine_cost                                  OK
mysql.event                                        OK
mysql.func                                         OK
mysql.general_log                                  OK
mysql.gtid_executed                                OK
mysql.help_category                                OK
mysql.help_keyword                                 OK
mysql.help_relation                                OK
mysql.help_topic                                   OK
mysql.innodb_index_stats                           OK
mysql.innodb_table_stats                           OK
mysql.ndb_binlog_index                             OK
mysql.plugin                                       OK
mysql.proc                                         OK
mysql.procs_priv                                   OK
mysql.proxies_priv                                 OK
mysql.server_cost                                  OK
mysql.servers                                      OK
mysql.slave_master_info                            OK
mysql.slave_relay_log_info                         OK
```

```
mysql.slave_worker_info                    OK
mysql.slow_log                             OK
mysql.tables_priv                          OK
mysql.time_zone                            OK
mysql.time_zone_leap_second                OK
mysql.time_zone_name                       OK
mysql.time_zone_transition                 OK
mysql.time_zone_transition_type            OK
mysql.user                                 OK
Upgrading the sys schema.
Checking databases.
sys.sys_config                             OK
test.sbtest                                OK
test.su                                    OK
test.t1                                    OK
test.tb_1                                  OK
test.tt                                    OK
test.tzy                                   OK
Upgrade process completed successfully.
Checking if update is needed.
```

没有报错，表示系统表升级成功。

第六步，正常启动 MySQL 数据库，不要使用--skip-grant-tables 和--skip-networking 参数。

首先停掉 MySQL 服务，命令如下：

```
mysqladmin -uroot -proot123 shutdown
```

然后正常启动 MySQL 服务，命令如下：

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

目前已经是 MySQL 5.7 版本，证明升级成功：

```
[root@node3 ~]# /usr/local/mysql/bin/mysql -V
/usr/local/mysql/bin/mysql  Ver 14.14 Distrib 5.7.14, for linux-glibc2.5 (x86_64)
ne wrapper
```

MySQL 5.7 版本中的 sys 库也存在：

```
root@db 19:15:  [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

接下来再使用 Logical 逻辑升级的方式来实战 MySQL 5.6.16 升级到 MySQL 5.7.14 的过程。生产环境中根据实际情况来选择哪种方法更适用于目前的升级预案。

第一步，利用 mysqldump 导出数据：

```
/usr/local/mysql/bin/mysqldump -uroot -proot123 --single-transaction
--master-data=2 -A > all.sql
```

第二步，将旧版本 MySQL 5.6 的数据库关闭并且备份原来的数据目录。

建立新版本的数据目录并授权：

```
/usr/local/mysql/bin/mysqladmin -uroot -proot123 shutdown
mv /data/mysql/ /data/mysql_bak
mkdir -p /data/mysql
chown mysql:mysql -R  /data
```

第三步，替换 MySQL 的安装文件：

```
lrwxrwxrwx  1 mysql mysql        34 Oct 30 10:59 mysql -> mysql-5.6.16-linux-glibc2.5-x86_64
drwxr-xr-x 13 root  root       4096 Oct 30 11:04 mysql-5.6.16-linux-glibc2.5-x86_64
-rw-r--r--  1 root  root  304788904 Nov  1  2016 mysql-5.6.16-linux-glibc2.5-x86_64.tar.gz
```

需要执行 unlink mysql 命令。

解压新版本的 MySQL 软件包，然后重新做连接并赋予 MySQL 权限。命令如下：

```
tar -zxvf mysql-5.7.14-linux-glibc2.5-x86_64.tar.gz
ln -s mysql-5.7.14-linux-glibc2.5-x86_64 mysql
chown -R mysql:mysql mysql
```

```
lrwxrwxrwx 1 mysql mysql        34 Nov 12 17:34 mysql -> mysql-5.7.14-linux-glibc2.5-x86_64
drwxr-xr-x 9  7161 31415      4096 Jul 12  2016 mysql-5.7.14-linux-glibc2.5-x86_64
-rw-r--r-- 1 root  root  642694570 Nov 12 16:43 mysql-5.7.14-linux-glibc2.5-x86_64.tar.gz
```

第四步，替换之前 MySQL 的配置文件为 MySQL 5.7 版本的 my.cnf 文件，并初始化数据库。命令如下：

```
    /usr/local/mysql/bin/mysqld --defaults-file=/etc/my.cnf
--basedir=/usr/local/mysql --datadir=/data/mysql -initialize
```

第五步，启动新版本 MySQL。

> 注：在启动过程中要注意的是，需要添加--skip-grant-tables 和--skip-networking 参数，来保证没有任何的应用连接，让升级过程更加安全。

```
    /usr/local/mysql/bin/mysqld_safe --defaults-file=/etc/my.cnf
--skip-grant-tables --skip-networking &
```

第六步，导入之前导出的文件到新版本数据库中，命令如下：

```
    /usr/local/mysql/bin/mysql  < all.sql
```

第七步，需要升级数据字典信息，命令如下：

```
    /usr/local/mysql/bin/mysql_upgrade
```

输出报告如下：

```
Checking if update is needed.
Checking server version.
Running queries to upgrade MySQL server.
Checking system database.
mysql.columns_priv                                 OK
mysql.db                                           OK
mysql.engine_cost                                  OK
mysql.event                                        OK
mysql.func                                         OK
mysql.general_log                                  OK
mysql.gtid_executed                                OK
mysql.help_category                                OK
mysql.help_keyword                                 OK
mysql.help_relation                                OK
mysql.help_topic                                   OK
mysql.innodb_index_stats                           OK
```

```
mysql.innodb_table_stats                          OK
mysql.ndb_binlog_index                            OK
mysql.plugin                                      OK
mysql.proc                                        OK
mysql.procs_priv                                  OK
mysql.proxies_priv                                OK
mysql.server_cost                                 OK
mysql.servers                                     OK
mysql.slave_master_info                           OK
mysql.slave_relay_log_info                        OK
mysql.slave_worker_info                           OK
mysql.slow_log                                    OK
mysql.tables_priv                                 OK
mysql.time_zone                                   OK
mysql.time_zone_leap_second                       OK
mysql.time_zone_name                              OK
mysql.time_zone_transition                        OK
mysql.time_zone_transition_type                   OK
mysql.user                                        OK
The sys schema is already up to date (version 1.5.1).
Found 0 sys functions, but expected 22. Re-installing the sys schema.
Upgrading the sys schema.
Checking databases.
sys.sys_config                                    OK
test.sbtest                                       OK
test.su                                           OK
test.t                                            OK
test.zs                                           OK
Upgrade process completed successfully.
Checking if update is needed.
```

升级成功。

第八步，正常启动 MySQL 数据库，不要使用--skip-grant-tables 和--skip-networking 参数。

首先停掉 MySQL 服务，命令如下：

```
mysqladmin -uroot -proot123 shutdown
```

然后正常启动 MySQL 服务，命令如下：

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

目前已经是 MySQL 5.7 版本，证明升级成功：

```
[root@node4 ~]# /usr/local/mysql/bin/mysql -V
/usr/local/mysql/bin/mysql  Ver 14.14 Distrib 5.7.14, for linux-glibc2.5 (x86_64)
```

MySQL 5.7 版本中的 sys 库也存在：

```
root@db 15:20:  [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

第 7 部分　最强王者篇

我们经历了整个 MySQL 数据库知识框架的学习之后，终于来到了 MySQL 的最后一个段位“最强王者”。前面 6 个部分的学习是知识的积累与沉淀。我们已经有了一个量的积累，剩下来的就是质的飞跃。最后一部分总结在面试过程中面临的各种技术性的问题。俗话说得好，是骡子是马拉出来遛遛。我们也应该对所学的知识做一个系统的总结和一个自我检测了。由于作者一直从事数据库相关的培训工作，所以总是会遇到有些学生平时学得很好，而且掌握知识的能力也很强，但一去面试就发挥不出自己应有的水平，薪资也自然达不到自己的预期。学生给我的反馈就是看见那些面试官就胆怯，一问问题就不知道该说些什么，之前学过的那些知识也自然就忘得一干二净了。其实也可以理解这种情况，因为刚跨入这个行业的人员对于数据库领域的理解还相当陌生，即使学完所有核心的理论知识，心里可能还是对自己没有信心。其实对于我们这些做技术的人来说，学习能力固然很重要，但更为重要的是要有一个良好的心态。作为 DBA，今后可能都会遇到误删数据、服务器宕机，以及各种疑难故障问题，这些状况就跟家常便饭一样。如果没有一颗大心脏，那么遇到一点困难，就会自己慌了神，感觉手足无措，不知道该如何下手。这些心态上的问题，需要有一个克服的过程。希望大家不管遇到什么问题，要相信 MySQL 数据库没有那么脆弱，更要相信自己的能力，把自己所学的所有知识都轻松地展现出来就好。拥有一颗平常心在工作和面试中就显得格外重要了。

第 20 章
MySQL 面试宝典

面试就是一个检验自己学习水平的平台，有些人可以借助这个平台去知名大公司展现自我价值，有些人可以借助这个平台拿到自己期望中的高薪，而有些人只能被更优秀的竞争者淘汰。虽说我们可以在面试中各取所需，但这些需求的背后，也需要我们付出相应的努力。这里重点强调的就是面试中的自我介绍环节。有些人总有理解上的误区，总觉得自我介绍就是一个流程，没有必要太重视，简单说说就行了。其实对于面试官来说，通过简单的自我介绍，从你的衣着、长相、言行举止等可以快速获取对你的第一印象。此外，通过自我介绍，可以为后面的面试提供相对应的话题。作为 DBA 的我们需要衔接各种部门的人员，测试、开发、运营、运维，等等，所以语言组织能力、逻辑思维能力、沟通能力，都将作为面试官审核你是否达标的依据。

20.1 自我介绍

在自我介绍中有三个要素一定要知晓。第一个就是要有自我认知的能力，意思是就是要对自己有个全方位的认识，可以从自己的过往工作经历来介绍自己，不要觉得简历上都写清楚了，为什么还非得再说一遍呢？面试官看重的东西其实很简单，主要就看你对这次面试的重视程度，你连自我介绍都可以漫不经心，那对交给你工作的态度更是需要打个问号了。短短的几分钟聊天就能让对方对你有个大致的认识，所以耐心地介绍一遍自己的履历并不是什么难事。大多数面试官最喜欢问上家公司的一些事情，你就可以从上一家公司所处理过的事情中抽出一件印象深刻的来讲给他听，营造一种轻松的聊天氛围，既可以让自己紧张的心情放松下来，又会给自己整个的面试过程加不少分。当然这都是要建立在真诚的基础上。

第二点就是对岗位的认知能力。在各大招聘网站上我们都会看到对某一岗位的职责描述，比如作为 DBA，我们要掌握优化数据库性能瓶颈的本领。你就可以从之前所处理过的优化案列中找出一件，阐述给面试官在你遇到数据库某一性能瓶颈问题时，如何屡清优化思路，利用资源进行相应的优化操作来解决问题。这种阐述就是一种推销自己的面试，可以让面试官快速从众多竞聘者中记住你，也会向高层介绍你突出的处理问题的能力，间接说明你很适合当前这个岗位。

第三点就是分清主次，捡重要的说。在面试中我们先要学会听清对方所提问的问题，不要在不了解问题的基础上畅所欲言。而且在回答问题时，要学会摸清对方的心理，捡重点来阐述。面试官看重的是你是否是一个有大局观、测分清主次的人。今后当你向领导汇报问题时，可以协助领导以最快的速度了解目前的现状，早下决策。

以上就是自我介绍的三点要素，希望可以让大家重新重视起面试中自我介绍的重要性。机会永远留给有准备的人。

除了自我介绍的重要性，面试官可能还会对你的过往经历比较感兴趣，看看你每段工作的时间，是否是那种频繁跳槽的人。一些人认为频繁跳槽可以开拓视野，让自己多角度思考问题，年轻人就得折腾，多经历一些事情不是坏事。但作为 DBA 的我们，一定要不忘初心，牢记使命！维护公司核心数据库的安全与业务的稳定性是我们最核心的任务。如果出现经常跳槽的现象，是没有公司敢要你的，技术更新换代很迅速，而且学习的过程是需要积累与沉淀的，跳槽太多不利于你静下心来深入研究技术知识，你可能永远停留在最初级的阶段，那么很多优秀的岗位、诱惑的薪资都将与你擦肩而过。所以作者希望在数据库领域的这些年轻人，不要太浮躁，踏实做事认真做人，稳步前行。

最后也可以谈谈你对未来进入公司之后的一些计划。这个可以从比较专业的角度去阐述。可以先简单了解一下公司现有业务模式（面试之前就要做好功课），提出相关在此业务模式基础之上数据库架构设计方案的建议。针对创业型公司，我们也可以谈针对未来组建运维、数据库团队的规划，让面试官看到你加入公司的决心。

上面说了这么多，都是从非技术的角度去了解面试中的一些环节和注意事项。接下来我们从专业的技术角度总结一下在 MySQL DBA 面试过程中，最常被问及的一些技术问题。这里总结了 15 个问题及其相关解答思路，方便大家在今后的面试中，可以非常顺利地通过技术面试，得到自己期望中的高薪资。

20.2　技术问答

问题 1：

你目前接触的 MySQL 版本是什么？除了官方版本，还接触过其他的 MySQL 分支版本吗？

解答思路：首先可以谈谈产生分支的原因。许多开发人员认为有必要将其拆分成其他项目，并且每个分支项目都有自己的专长。该需求和 Oracle 对核心产品增长缓慢的担忧，导致出现了许多开发人员感兴趣的子项目和分支。

介绍一下三个流行的 MySQL 分支：Drizzle、MariaDB 和 Percona Server（包括 XtraDB 引擎）。

更细节化展开说明各个分支的特点。MariaDB 不仅是 MySQL 的替代品，主要是创新和提高了 MySQL 自有技术。新功能介绍如下：multi-source replication 多源复制、表的并行复制（MySQL 5.7 版本中也加入该特性）、galera cluster 集群、spider 水平分片和 TokuDB 存储引擎。

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，还包含在高性能模式下的新特性。它可以向下兼容，因为它是在 InnoDB 基础上构建的，所以有更多的指标和扩展功能。而且它在 CPU 多核的条件下，可以更好地使用内存，将数据库性能提到更高。

Drizzle 与 MySQL 的差别就比较大了，并且不能兼容，如果想运行此环境，就需要重写一些代码了。

问题 2：

MySQL 主要的存储引擎 MyISAM 和 InnoDB 的不同之处？

解答思路：可以从五个方向去介绍。事务的支持不同（InnoDB 支持事务、MyISAM 不支持事务）；锁粒度（InnoDB 行锁应用、MyISAM 表锁）；存储空间（InnoDB 既缓存索引文件，又缓存数据文件，MyISAM 只能缓存索引文件）；存储结构（MyISAM：数据文件的扩展名为.MYD myData，索引文件的扩展名是.MYI myIndex；InnoDB：所有的表都保存在同一个数据文件里面，即.Ibd）；统计记录行数（MyISAM：保存有表的总行数，select count(*) from table 会直接取出该值；InnoDB：没有保存表的总行数，select count(*) from table 会遍历整个表，消耗相当大）。

问题 3：

介绍一下 InnoDB 的体系结构。

解答思路：谈及 InnoDB 的体系结构，首先要考虑 MySQL 的体系结构，分为 MySQL 的 server 层和存储引擎层两部分。

先要跟面试官聊清楚 MySQL 的整体方向，然后再去涉及 InnoDB 体系结构。建议从三方面介绍 InnoDB 体系结构：内存、线程、磁盘。

内存中包含 insert_buffer、data_buffer、index_buffer、redo_log_buffer、double_write。

内存刷新到磁盘的机制：redo log buffer、脏页、binlog cache 的刷新条件。

各种线程的作用：master_thread、purge_thread、redo log thread、read thread、write thread、

page cleaner thread。

磁盘中存放的数据文件：redo log、undo log、binlog。

问题 4：

MySQL 有哪些索引类型？

解答思路：可以从三个角度去谈。首先从数据结构角度上可以分为 B+tree 索引、hash 索引、fulltext 索引（InnoDB、MyISAM 都支持）。

其次从存储角度上可以分为聚集索引和非聚集索引。

最后从逻辑角度上可以分为 primary key、normal key、单列、复合和覆盖索引。

问题 5：

MySQL binlog 有几种格式？生产中你用哪种？各自有什么特点？

解答思路：可以对各种格式的优缺点分别介绍。

第一种，statement 格式。

优点：不需要记录每一行的变化，减少了 binlog 日志量，节约了 I/O，提高了性能。

缺点：当使用一些特殊函数，或者跨库操作时容易丢失数据。

注：在生产中不建议使用。

第二种，row 格式。

优点：清晰记录每行的数据信息，不会出现跨库丢数据的情况，安全性非常高。

缺点：当内容记录到日志中时，都将以每行的修改来记录，会产生大量的 binlog，网络开销也比较大。

注：生产中推荐使用。

第三种，mixed 格式。

MySQL 5.1 的一个过渡版本，DDL 语句会记录成 statement，DML 会记录成 row。

注：生产中不建议使用。

问题 6：

MySQL 主从复制的具体原理是什么？

解答思路：直接阐述原理即可，表达一定要清楚。主服务器把数据更新记录到二进制日志中，从服务器通过 I/O thread 向主库发起 binlog 请求，主服务器通过 I/O dump thread 把二进制日志传递给从库，从库通过 I/O thread 记录到自己的中继日志中。然后通过 SQL thread 应用中继日志中 SQL 的内容。

问题 7：

MySQL 主从延迟的原理是什么？如何监控主从延迟，如何解决主从延迟问题？

解答思路：这个问题是上一个问题的延伸。我们可以先从最核心的延迟问题讲解。我们知道主库可以并发写入，但从库只能通过单 SQL thread 完成任务（MySQL 5.7 之前），这是出现主从延迟的最核心原因。

再从其他方面来总结主从延迟原因：

（1）MySQL 主从之间的同步本来就不是实时同步的，是异步的同步，也就是说，主库提交事务之后，从库才再执行一遍。

（2）在主库上对没有索引大表的列进行 delete 或者 update 的操作。

（3）从库的硬件配置没有主库的好，经常忽略从库的重要性。

（4）网络抖动导致 I/O 线程复制延迟。

针对如何监控 MySQL replication 复制延迟的问题。我们可以通过第三方工具（业界中的瑞士军刀 percona-toolkit）中的 pt-heartbeat 命令进行主从延迟监控。

传统方法，通过比较主从服务器之间的 position 号的差异值。

还可以通过查看 seconds_behind_master 估算一下主从延迟时间。

介绍完诸多引起延迟的原因之后，可以再进行展开延迟的解决方法的讨论。

（1）使用 MySQL 5.7 的并行复制功能。在 5.6 版本中就有了并行的概念，但其中的并行复制是基于库级别的，即 slave_parallel_type=database。但这种模式下，只是基于多库少表的情况，并不适用于真实的生产环境下。在 MySQL 5.7 版本中，真正实现了基于组提交的并行复制，简单说就是主库并行执行 SQL 语句，从库也可以通过多个 workers 线程并发执行 relay log 中主库提交的事务。想要开启 MySQL 5.7 的并行复制，可以在从库设置参数 slave_parallel_workers>0，并把 5.7 版本中新添加的 slave_parallel_type 参数设置为 LOGICAL_CLOCK。该参数有 DATABASE 和 LOGICAL_CLOCK 两个值。MySQL 5.6 默认是 DATABASE。

（2）可以采用 Percona 公司的 percona-xtradb-cluster（简称 PXC 架构），这种架构下可以实现多节点写入，达到实时同步。

（3）业务初期规划时，就要选择合适的分库、分表策略，避免单表或者单库过大，带来额外的复制压力，从而带来主从延迟的问题。

（4）避免一些无用的 I/O 消耗，可以增加高转速的磁盘、SSD 或者 PCIE-SSD 设备。

（5）阵列级别要选择 RAID10，raid cache 策略要采用 WB，坚决不要采用 WT。

（6）I/O 调度要选择 deadline 模式。

（7）适当调整 buffer pool 的大小。

（8）避免让数据库进行各种大量运算，要记住数据库只是用来存储数据的，让应用端多分担些压力，或者可以通过缓存来完成。

问题 8：

数据库中的双一是什么？

解答思路：遇到这样的问题，可以从两个参数着手分析。一个是 sync_binlog=1，另一个就是 innodb_flush_log_at_trx_commit=1。

innodb_flush_log_at_trx_commit 和 sync_binlog 两个参数是控制 MySQL 磁盘写入策略以及数据安全性的关键参数。

innodb_flush_log_at_trx_commit 设置为 1，每次事务提交时，MySQL 都会把 log buffer 的数据写入 log file，并且刷到磁盘中。

sync_binlog =N（$N>0$），MySQL 在每写 N 次二进制日志 binary log 时，会使用 fdatasync() 函数将它的写二进制日志 binary log 同步到磁盘中。

问题 9：

如何实施大表 DDL 语句才能把性能影响降到最低？

解答思路：我们可以先通过传统方法导入/导出数据，新建一张与原表一样的表结构，把需要执行的 DDL 语句在无数据的新表中执行，然后把老表中的数据导入到新表中，最后把新表改成老表的名字。

也可以通过第三方工具（percona-toolkit）中的 pt-online-schema-change 命令进行在线操作，但对于 MySQL 5.7 版本可以直接在线“online ddl”。

还可以介绍一下 MySQL 对大表进行 drop table 操作时，可以对数据文件建立硬连接，这样可以缩短执行时间。依赖的原理：OS HARD LINK。当多个文件名同时指向同一个 INODE 时，这个 INODE 的引用数 $N>1$，删除其中任何一个文件名只是删除了一个指针，不会删除数据文件。当 INODE 的引用数 N=1 时，删除文件时需要把这个文件相关的所有数据块清除，所以会比较耗时。

问题 10：

为什么要为 InnoDB 表设置自增列做主键？

解答思路：使用自增列做主键，写入顺序是自增的，和 B+数叶子节点分裂顺序一致。InnoDB 表的数据写入顺序能和 B+树索引的叶子节点顺序一致时，存取效率是最高的。

问题 11：

如何优化一条慢 SQL 语句？

解答思路：针对 SQL 语句的优化，我们不要一上来就回答添加索引，这样显得太不专业。我们可以从如下几个角度去分析：

（1）回归到表的设计层面，数据类型选择是否合理。

（2）大表碎片的整理是否完善。

（3）表的统计信息是不是准确的。

（4）审查表的执行计划，判断字段上面有没有合适的索引。

（5）针对索引的选择性，建立合适的索引（就又涉及大表 DDL 的操作问题。所以说，我们要有能力把各个知识点联系起来）。

问题 12：

服务器负载过高或者网页打开缓慢，简单说说你的优化思路？

解答思路：我们可以通过前面讲过的优化思路中的四维度模型去阐述。

首先要发现问题的过程，通过操作系统、数据库、程序设计、硬件角度四个维度找到问题所在。先找到瓶颈点的位置，制定好优化方案，形成处理问题的体系模型。体系制定好之后，在测试环境进行优化方案的测试。测试环境下如果优化效果很好，再实施到生产环境上。最后做好处理问题的记录。好记性不如烂笔头，多做总结，方可大步前进。

问题 13：

你接触过哪些 MySQL 的主流架构？架构应用中有哪些问题需要考虑？

解答思路：先整体介绍一下你所知道的集群架构：

（1）M-S。

（2）MHA。

（3）MM+Keepalived。

（4）PXC。

（5）利用中间件 ProxySQL 配合 PXC 架构。

再总结各种集群的优缺点，可以先从 MHA 开始说起。

MHA 优点总结：

（1）故障切换时，可以自行判断哪个从库与主库的数据最接近，就切换到上面，可以减少数据丢失的风险，保证数据的一致性。

（2）支持 binlog server，可提高 binlog 传送效率，进一步减少数据丢失风险。

（3）结合 MySQL 5.7 的增强半同步功能，来确保故障切换时的数据不丢失。

缺点：

（1）自动切换的脚本太简单了，而且比较老化，建议后期逐渐完善。

（2）搭建 MHA 架构，需要开启 Linux 系统互信协议，所以对于系统安全性来说是个不小的考验。

再介绍一下 MM+Keepalived 集群的建议：

（1）一定要完善好切换脚本，Keepalived 的切换机制要合理，避免发生切换不成功的现象。

（2）从库的配置尽量要与主库的一致，绝对不能太差；避免主库宕机时发生切换，新的主库（原来的从库）影响线上业务进行。

（3）对于延迟的问题，在这套架构中也不能避免。可以改变架构模式，使用 PXC 完成实时同步功能，基本上可以达到没有延迟。

（4）Keepalived 无法解决脑裂的问题，因此在进行服务异常判断时，可以修改判断脚本，通过对第三方节点补充检测来决定是否进行切换，可降低脑裂问题产生的风险。

（5）采用 Keepalived 架构，在设置两节点状态时，都要设置成 backup 状态，而且还都是不抢占模式，通过优先级来决定谁是主库。避免发生脑裂、冲突现象。

（6）安装好 MySQL 需要的一些依赖包；建议配置好 Yum 源，用 Yum 安装 Keepalived 即可。

最后可以介绍一下 PXC 架构的优缺点。

优点如下：

（1）实现 MySQL 数据库集群架构的高可用性和数据的强一致性。

（2）完成了真正的多节点读写的集群方案。

（3）改善了传统意义上的主从复制延迟的问题，基本上达到了实时同步。

（4）对于新加入的节点可以自动部署，无须手动备份，维护起来很方便。

（5）由于是多节点写入，所以发生数据库故障时切换很容易。

缺点如下：

（1）新加入的节点开销大，需要复制完整的数据，采用 SST 传输开销太大。

（2）任何更新事务都需要全局验证通过，才会在每个节点库上执行。集群性能受限于性能最差的节点，也就是经常说的短板效应。

（3）因为需要保证数据的一致性，所以在多节点并发写时，锁冲突问题比较严重。

（4）存在写扩大问题，所有的节点上都会发生写操作。

（5）只支持 InnoDB 存储引擎表。

（6）没有表级别的锁定，执行 DDL 语句操作会把整个集群锁住，而且也“kill”不了（注：建议使用 OSC 操作）。

（7）所有的表必须含有主键，否则操作数据时会报错。

问题 14：

什么是死锁？锁等待？通过数据库哪些表可以监控？

解答思路：死锁是指两个或多个事务在同一资源上互相占用，并请求加锁时，而导致的恶性循环现象。当多个事务以不同顺序试图加锁同一资源时，就会产生死锁。

锁等待：MySQL 数据库中，不同 session 在更新同行数据时，会出现锁等待的现象。

重要的三张锁的监控表：innodb_trx、innodb_locks 和 innodb_lock_waits。

问题 15：

你之前处理过 MySQL 的哪些案例？

解答思路：说到案例，逃离不了 MySQL 的五大知识模块：体系结构、数据的备份恢复、复制、高可用集群架构和优化。我们可以从这五个方面着手考虑，比如：

（1）MySQL 版本的升级。

（2）处理集群架构中的各种“坑”和问题（你遇到过的就可以）。

（3）根据公司业务类型，合理设计 MySQL 库、表和后期架构。

（4）定期进行灾备恢复演练。

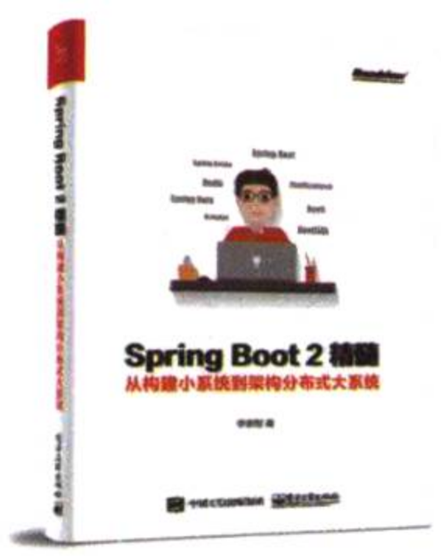
（5）恢复误删除的数据信息。

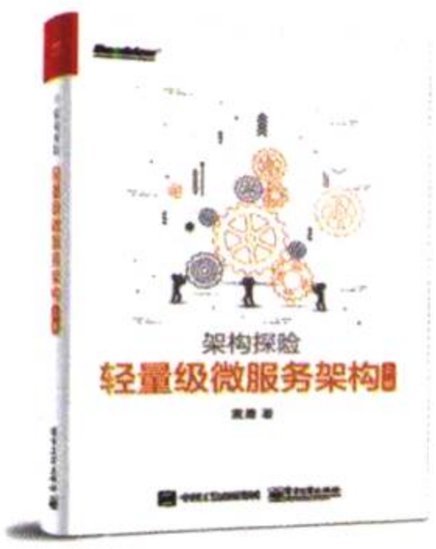
面试问题先总结到这里，可能还会有超出介绍范围之外的面试问题，希望今后可以多多探讨。

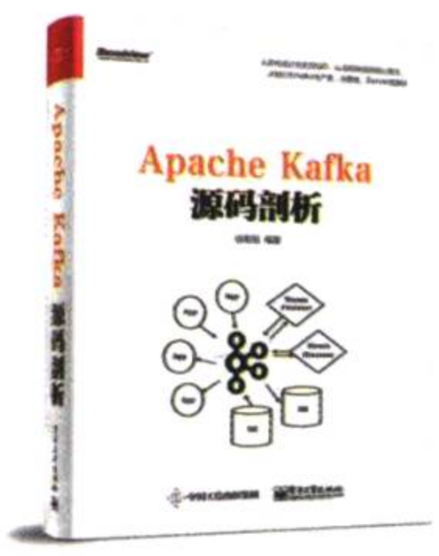
至此，全书的内容就结束了，作者按照从青铜到王者，分级别、分层次地介绍了 MySQL 数据库中涉及的核心知识点，全书采用原理配合生产实战的讲解方式，深入剖析 MySQL 数据库，就是希望可以在工作中对大家有所帮助，可以帮助那些刚进入该领域的年轻人。我知道现在网上的学习资料很多，很少有人再愿意看书了。但我还是希望，大家抽闲时可以读一读，也许就可以从中找到解决问题的灵感。

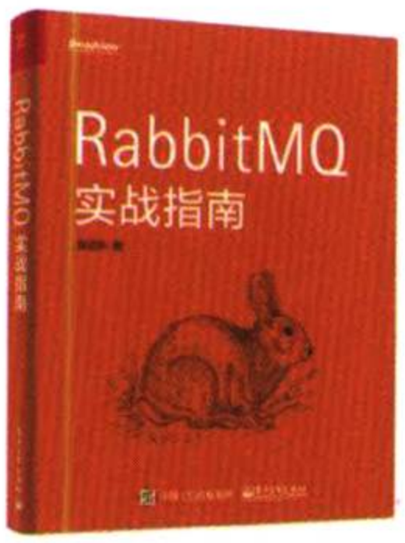
雷霆雨露，俱是天恩。在工作生活中遇到的再难的事，都是上天对我们的“恩赐”，不要着急，总会有解决问题的办法。最后祝大家都可以找到理想的工作，薪资翻倍！我们一起努力！

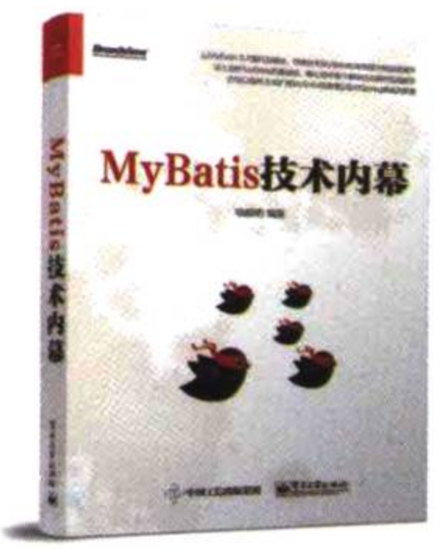
好书分享











拒绝堆砌臃肿，支持纯正原创

欢迎投稿：chenxm@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

初始张甦就有一见如故的亲切感，本书是对MySQL基础知识的全面解析，也是张甦多年工作和教学经验的结晶，其内容几乎涵盖了初学者对MySQL知识需求的方方面面，它将会带你进入MySQL的神奇殿堂。

周彦伟

《MySQL运维内参》作者、中国MySQL用户组主席

书中汇集了张甦多年来在MySQL运维及教学过程中不断总结和思考的成果，内容丰富，体例清晰。

本书深入浅出，语言平实又不乏幽默，轻快又不失严谨，确是一本好书。

王竹峰

去哪儿网数据库总监 Oracle MySQL ACE

张甦收集整理了自己多年一线的经验，以一种轻松明快的文风来讲述MySQL体系结构和运维相关知识，通过本书，不仅可以对MySQL知识体系有全面的了解，还可以掌握DBA工作的核心技巧，相信会带给你一些新的思考和方向。

杨建荣

DBAplus社群发起人，Oracle ACE，《Oracle DBA工作笔记》作者

张甦通过自己近十年的数据库运维经验和授课感悟，把MySQL大部分知识点描述得非常直白，并伴有详细举例，非常适合刚入门的同学来学习。

本书完全假设受众为零基础的同学，深入浅出地介绍了MySQL相关知识，建议作为MySQL的入门书籍。

肖鹏

微博研发中心 技术副总监

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：陈晓猛
封面设计：李　玲

上架建议：计算机-数据库

ISBN 978-7-121-33679-9

定价：79.00元