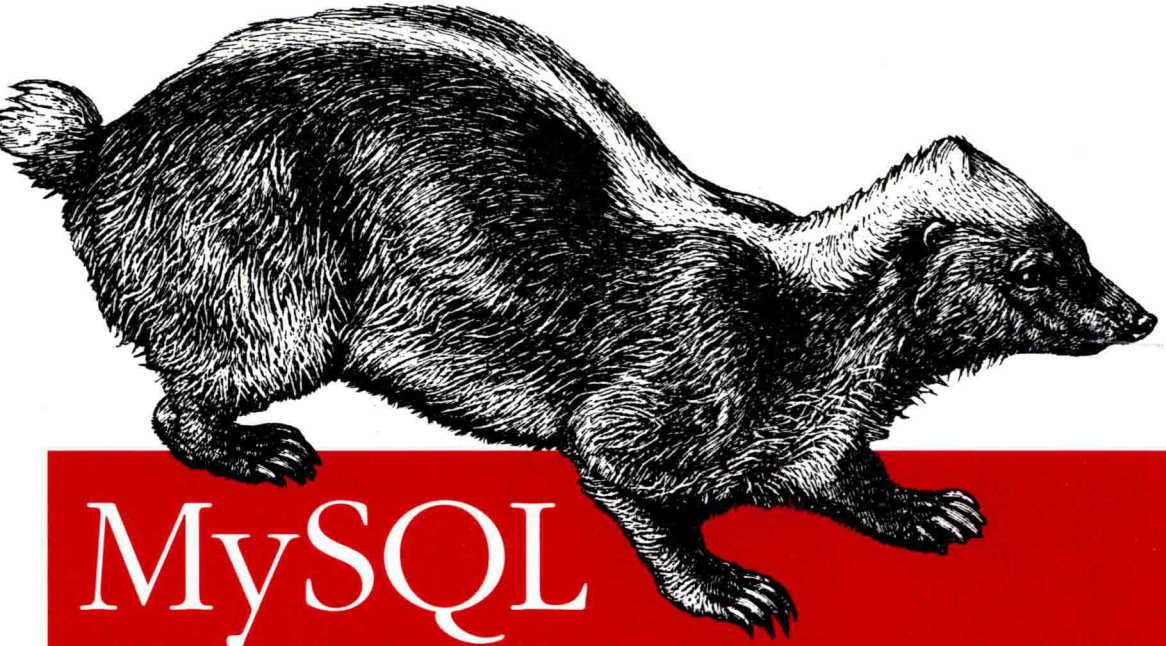
MySQL Troubleshooting



MySQL 排错指南

[美] *Sveta Smirnova* 著
[美] *Dr. Charles Bell* 序
李宏哲 杨挺 译

O'REILLY®
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

MySQL 排错指南

[美] Sveta Smirnova 著

李宏哲 杨挺 译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

MySQL排错指南 /（美）斯米尔诺娃（Smirnova, S.）著 ; 李宏哲, 杨挺译. -- 北京 : 人民邮电出版社, 2015.8
ISBN 978-7-115-39728-7

Ⅰ. ①M… Ⅱ. ①斯… ②李… ③杨… Ⅲ. ①关系数据库系统一指南 Ⅳ. ①TP311.138-62

中国版本图书馆CIP数据核字(2015)第152225号

◆ 著　［美］Sveta Smirnova
译　李宏哲　杨　挺
责任编辑　傅道坤
责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市中晟雅豪印务有限公司印刷

◆ 开本：787×1000　1/16
印张：14.75
字数：297 千字　2015 年 8 月第 1 版
印数：1 – 2 000 册　2015 年 8 月河北第 1 次印刷
著作权合同登记号　图字：01-2013-4953 号

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书由 Oracle 公司的技术支持工程师编写，详细阐述了 MySQL 故障诊断及处理中的知识，教会读者如何深入浅出地定位错误和异常方法，分析并解决各种 MySQL 数据库的故障。

本书共分为 7 章，其内容涵盖了解决 MySQL 问题的基本技巧、MySQL 中的并发问题、服务配置的影响、MySQL 硬件和运行环境相关的问题、复制备份中的故障排除、故障排除使用的技术和工具，以及一些 MySQL 故障排除的最佳实践。此外，本书的附录中还包含了可以帮助读者解决 MySQL 疑难问题的一些有用资源。

本书适合 MySQL 数据库开发及运维人员阅读。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

—— Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

—— Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

—— CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

—— Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

—— Linux Journal

序

解决系统故障可谓是每个系统专家遭遇过的最头疼的问题之一，虽然修复问题或执行解决方案通常是比较容易的部分，但通过诊断分析找出问题的原因才是真正的挑战。

经验丰富的管理员经历过各种尝试和失败，都意识到解决问题最好的方式是通过标准化的步骤来定位问题，列出所有可能的原因，然后依次测试，直到找到解决方案。这种方式听起来初级，但很有效（尽管对于资深系统专家来说不够高效）。

MySQL 是一个专业、复杂、成熟、强大的数据库系统，可以满足大量客户的需求。MySQL 的安装和配置都很简单。事实上，大部分默认安装根本不需要任何配置。然而，MySQL 作为一个拥有众多功能层次的系统，有时也会有故障，会产生警告甚至错误。

有时候，这些警告和错误提供的信息明确（比如，你可能经历过或者在文档中有记录），足以帮助你立即解决问题。也有时候，你遭遇的问题没有已知的解决方案，或者该问题是与你的应用、数据库环境相关的特定问题。寻找这类警告、错误或者其他 MySQL 问题的解决方案可以说是一项让人望而却步的工作。

当遭遇这种问题的时候，数据库专家一般都从各种各样的资料或者记录过相似问题和解决方案的文档中查找线索。大多数时候，你会发现针对问题的建议繁多且相似，或者其中给出的解决方案对你的情况并不适用。

针对这种情形，人们普遍的做法是在互联网上搜索错误消息。通常，你会搜索到各种各样的信息，包括从电子邮件的归档到个人博客，甚至是一些可能与错误消息相关或者不相关的评论。这往往很浪费时间而且容易造成困惑。你需要的其实是一个可以告诉你如何解决 MySQL 问题的参考指南。

这本书不仅仅满足上述需求，还建立了一种几乎适用于解决所有系统问题的方式、方法。书中展示出的方法结构清晰、周密并且可重复运用。结合现实工作中的示例，本书定义了一种合理地分析和修复 MySQL 问题的方法，这是一个标志性的成果。

Sveta 用第一手的经验、丰富的 MySQL 知识与诊断技巧为读者讲解诊断和修复几乎所有可能遇到的 MySQL 问题的基本技能——这也使本书成为 MySQL 专家的必备书籍。

我自认为是一个 MySQL 专家，我的技术来自于丰富的经验。我不敢说我知道 MySQL 的所有细节。读完这本书，我可以说它提升了我的技能。如果一个像我这样经验丰富

的专家都能从此书获益，那么每个 MySQL 用户都应该阅读此书。特别是所有 MySQL 数据库管理员、顾问以及数据库开发人员都应该阅读本书。

Charles Bell 博士，Oracle 公司

MySQL High Availability（O′ Reilly）和 *Expert MySQL*（Apress）图书的作者

前言

我从 2006 年 5 月开始，作为首席技术支持工程师在 MySQL AB 公司 MySQL 支持团队的 bug 校验组工作，然后我到了 Sun 公司，最后是在 Oracle 公司。在日常工作中，我经常遇到用户受困于某个问题而不知所措的情况。虽然有已经被证实可用的方法去定位并快速修复问题，但是用户往往很难从大量的信息中筛选出这些可用的信息。尽管有数以千百计的著名书籍、博文和网页都详细介绍了 MySQL 服务器方方面面的问题，但这正是我感觉困难的地方：这些信息都关注于如何让 MySQL 服务器正常地工作，而忽略了定位错误和异常的方法。

当这些信息综合到一起的时候，它们能够详细解释 MySQL 操作的每个方面。但是，如果你不知道问题是如何发生的，你可能会从文档中提到的大量建议中忽略掉真正的原因。即使你向专家咨询问题产生的原因，他们也可能只会给出很多的建议，你仍需要找出真正的原因。否则，你做出的任何修改可能只是临时解决了问题，甚至反而使问题更糟。

了解问题的来源非常重要，有时对 SQL 语句或者配置选项的一个修改就可以解决问题。掌握错误的原因可以让你永久地修复它并保证以后不再发生。

我写这本书的目的是告诉读者我经常使用什么方法来确定导致 SQL 应用程序或 MySQL 配置错误的原因，以及如何解决这些问题。

本书读者对象

本书是写给具有一定 MySQL 知识基础的读者的。我会尽量使书中的内容对初学者和高级用户都有帮助。读者需要知道 SQL 语句并且简单了解 MySQL 服务器是如何工作的，至少从用户手册或者初学者指南中了解一二。最好读者有实际的使用经验或者已经遇到了难以解决的问题。

我不想重复介绍其他资料中已有的内容，我会更偏重于补充定位错误和异常行为的方法。因此，在本书中你将获得的是如何修复应用程序的指导，而不是有关应用程序和服务器行为的细节介绍。想了解更多信息，可阅读 MySQL Reference Manual（http://dev.mysql.com/doc/refman/5.5/en/index.html）。

如何解决问题

本书将围绕着帮助读者定位问题和寻找原因的目标进行组织。我会介绍我逐步解决问

题的经过，而不会罗列出一大堆不相关的信息或者空想出的方法。

提示：弄清问题是什么非常重要。

例如，当解释 MySQL 安装缓慢的时候，你需要确认哪里缓慢：是仅应用程序相关的部分缓慢，还是所有发送到 MySQL 服务器的查询运行缓慢？你也最好知道，是否同样的安装以前也缓慢，并且该问题是一直存在还是周期性地重现。

另一个例子是关于错误的行为。你需要知道什么是错误的操作，产生了什么结果和你预期的结果是什么。

我现在已经很擅长介绍解决问题的方法。很多问题都有不同的解决方式，最佳的解决方案取决于应用程序和用户的需求。如果我面面俱到地介绍各种解决问题的方法，那么本书的篇幅可能是现在的 10 倍，这可能让你忽略了适合自己的方法。我的目标是使读者从起步就处于正确的道路上，以便可以快速解决各种问题。而修正问题的其他细节可以在各种资料中查到，其中许多资料都会在我们学习的过程中被引用和提及。

本书组织结构

本书包括 7 章和 1 个附录。

第 1 章，“基础”，介绍解决问题的基本技巧，你几乎会在任何场合使用到这些技巧。本章仅覆盖单线程问题，即在隔离条件下，独立连接产生的问题。之所以我从这种隔离的甚至有些理想化的条件下开始介绍，是因为你需要掌握这些在多线程应用中隔离问题的技术。

第 2 章，“你不孤单：并发问题”，介绍应用在多线程环境下运行或者应用与其他应用中的事务有交互的情况下产生的问题。

第 3 章，“配置选项对服务器的影响”，包括两部分内容。第一部分是调试和修复由配置项产生的问题的参考指南。第二部分是关于重要配置项的索引。也就是说，这部分可以根据需要查阅而不用通篇阅读。第二部分也包括解决由于特定配置引发的问题的推荐方法，以及如何测试你是否真正解决了问题。我会尽可能介绍其他参考中没有的技术，并且尽量把所有常见配置项问题集中在一起。我还把它们进行分组，便于你方便地检索到产生问题的原因。

第 4 章，“MySQL 环境”，介绍其他关于硬件和服务器运行环境方面的问题。这是一个大话题，不过大部分的信息是针对操作系统的，而且通常只能由操作系统管理员解决。因此，本章列出了一些 MySQL 数据库管理员（DBA）必须关注的要点。阅读完该章后，你应该知道什么时候是你的环境的问题，以及如何向系统管理员清楚地解释问题。

第 5 章，“复制故障诊断”，本章侧重于复制场景下产生的问题。事实上，本书通篇都在

讨论复制的问题，不过其他章节阐述的是复制和其他问题的关系，本章仅仅针对复制问题。

第 6 章，“故障排查技术与工具”，本章补充介绍在之前解决问题的过程中略过的或者无法详细介绍的故障排除技术和工具。本章的目的是补充前几章遗漏的一些细节，你也可以把它作为参考索引。我先给出原则，然后列出可用工具。我不会列出没有使用过的工具，我列出的都是我自己每天使用的工具，也就是 MySQL 项目组开发的工具（当然现在属于 Oracle 公司）。我也使用第三方工具来帮助我每天处理 bug 和进行服务支持。

第 7 章，“最佳实践”，本章主要介绍安全、高效地解决问题的习惯和方法。本章不会介绍用于设计 MySQL 应用程序的最佳实践，因为这在其他资料中有详细的介绍，而是重点讲述有助于定位问题和避免问题发生的最佳实践。

附录，“资源信息”，本附录包含我日常工作中用到的可以帮助解决疑难问题的资源。当然，本书也用到了其中一部分资源，我已在相应的位置添加了参考信息。

本书的一些选择

在过去的几年，诞生了很多 MySQL 分支，其中最重要的莫过于 Percona 服务器和 MariaDB。但本书不会介绍它们，因为在日常工作中我主要用的都是 MySQL，我无法介绍平常不使用的数据库。然而，由于它们是 MySQL 的分支，因此你也可以使用本书介绍的方法。只有当你用到了分支版本中特有的功能时，你才需要了解针对该产品的内容。

为了节约篇幅，同时也避免介绍一个全新领域的知识，我略过了 MySQL 群集相关的问题。如果你在使用 MySQL 群集的过程中遇到了 SQL 或者应用程序相关的问题，那么你可以采用解决其他存储引擎问题的方式去解决该问题。也就是说，本书也适用于发生在群集环境上的类似问题。但是，解决 MySQL 群集特有的问题需要用到 MySQL 群集的知识，在这里我并没有介绍它们。

不过，本书用大量篇幅介绍与 MyISAM 和 InnoDB 引擎相关的问题。因为它们是目前最受欢迎的存储引擎，安装量巨大。它们分别是过去和现在默认的存储引擎：5.5 版本之前是 MyISAM；5.5 版开始是 InnoDB。

关于本书的示例，我还要多说几句。它们有的是专门为本书设计的，有的是为了讨论我提到过的问题而构造的。尽管有些示例来自于实际支持的案例和 bug 报告，但是所有的代码都是全新的并且不会涉及任何机密。有些地方介绍了一些客户需求，那也不是真实的。不过，这里描述的问题都是真实的，且会多次遇到，只是改用了不同的代码、名称和环境。

我尽可能地使所有示例简单、通用、易懂。因此大部分示例中都使用了 MySQL 命令行客户端。在 MySQL 安装包中，始终都含有此客户端。

这也解释了为何本书不介绍每个安装版本的所有问题；那不可能在一本书中完全涵盖。相反，本书会尽量给出一些引子，你可以在此基础上进行扩展。

我准备用 C 语言 API 去举例说明本书中讨论的功能。做出这种选择并不容易，因为有各种语言的 MySQL API。我不可能在本书中全部使用它们，也不想去猜测哪种语言的 API 更流行。我发现它们中的大部分看起来和 C 的 API 很相似（有很多甚至是在 C 的 API 上进行封装），因此我认为这会是最佳选择。即使你使用完全不同语法的 API，比如 ODBC，这部分也会很有帮助，因为你会知道应该搜索什么。

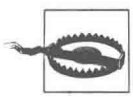
有些示例使用 PHP。我这么做是因为我日常使用的语言是 PHP，这样做可以展示我实际的代码示例。真实的示例往往更适用于展示，因为它们反映了现实生活中读者最可能遭遇的问题。并且，MySQL PHP API 也是基于 C 的 API 封装的，而且使用了同样的名字，因此读者可以很容易与本书中讨论的 C 的函数进行比较[1]。

我没有使用 JDBC 和 ODBC 的示例，因为这些 API 都太特殊。同时，它们的调试技术又很雷同，尽管不是完全一致。它们主要是语法有所不同。我觉得详细介绍这两种连接方式不但不会教会读者更多解决问题的技巧，反而会给读者造成困扰[2]。

本书约定

提示

这个图标用来强调一个提示、建议或一般说明。



警告

这个图标用来说明一个警告或注意事项。

代码示例的使用

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无须获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发 O'Reilly 图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但

1：mysqlnd 使用它自己实现的客户端协议，但是函数名称仍然使用跟 C API 一样的命名方式。

2：你可以从 http://dev.mysql.com/doc/refman/5.5/en/connector -j-reference.html 获取更多的关于 Connector /J（JDBC）的详细信息，从 http://dev.mysql.com/doc/refman/5.5/en/connector-odbc -reference.html 获取关于 Connector/ODBC 的信息。

是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过。一个属性信息通常包括书名、作者、出版社和ISBN。例如："*MySQL Troubleshooting by Sveta Smirnova* (O'Reilly). Copyright 2012 Sveta Smirnova, 978-1-449-31200-8."

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O'Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过如下地址访问该网页：

http://www.oreilly.com/catalog/9781449312008

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站（http://www.oreilly.com），查看更多我们的书籍、课程、会议和最新动态等信息。

Facebook: http://facebook.com/oreilly

Twitter: http://twitter.com/oreillymedia

YouTube: http://www.youtube.com/oreillymedia

致谢

我要感谢所有帮助完成本书的人，没有他们的帮助，本书将无法完成。

首先，我要感谢 Andy Oram 编辑，他给了我很大的帮助，使本书更加具有可读性。他

给我指出了本书中介绍不够细致的地方。他还帮我深入分析了潜在读者的技能储备，建议我增加针对初学者的介绍，同时删除了人人皆知的冗余细节。

我也要感谢整个 MySQL 支持团队。他们将专业知识同团队的每个成员分享，我从中学到了很多很多。这里我不一一列举他们的名字，我想说的是感谢我从 2006 年加入 MySQL 支持团队以来共事过的所有同事，包括那些已经离职的、跳到开发部门的，或者跳到其他公司的同事。

感谢 Charles Bell，是他督促我撰写本书。他对本书进行了审校并提出了很多改进意见。Charls 在 Oracle 公司的 MySQL 复制与备份团队工作，并且是两本 MySQL 专著的作者。他对于本书内容和版式的建议都非常有用。

我要感谢本书所有的审校人员。

- Shane Bester，我在 MySQL 支持团队的同事，他审校了 Gypsy 程序相关的部分并告诉我如何改进示例。
- Alexander（Salle）Keremedarski，他完成了全书的审校并给出了很多非常好的建议。Salle 从 MySQL 支持团队起步，在早期从事 MySQL 支持工作，现在担任 SkySQL EMEA 支持团队的主管。他非常了解普通用户理解上的一些误区，这些经验帮助我弥补了在解释某些问题时细节上的缺陷。这使得每个解决问题的情形读起来就好像是“最佳实践”一样，哪怕有时候它可能并不是。
- Tonci Grgin，审校了关于 MySQL Connector 的部分，建议我补充解释它们的行为。Tonci 曾经跟我在一个组共事，现在就职于 MySQL Connectors 团队。他是我咨询任何 MySQL Connector 问题的首选之人。
- Sinisa Milivojevic，审校了第 3 章和第 4 章以及一部分与 MyISAM 检查、修复工具相关的内容。Sinisa 是另一位从一开始就在 MySQL 支持团队工作的审校人。他是 MySQL 的 2 号雇员，并且目前仍在 Oracle 的 MySQL 支持团队工作。他丰富的经验令人赞叹，让人不禁觉得他了解 MySQL 的每个细节。Sinisa 帮我深入了解了书中所讨论的问题，并且给出很多简洁却非常重要的改进建议。
- Valeriy Kravchuk，他审校了第 2 章和第 4 章以及 6.5.3 节。他也在 MySQL 支持团队工作。Valeriy 在审校中发现了很多不足。他的鞭策促使我努力完善这些内容，当然现在仍有很大的改进空间。
- Mark Callaghan，他在 Facebook 从事数据库服务器运维工作。他审校了整本书。Mark 建议我在解释不清楚的地方增加示例和深入的分析。他给出了第 4 章的示例建议，并指出我之前给出的建议在某些特定的安装包下可能会出现错误，他建议我解释清楚两种场景：我之前给出的建议什么时候适用，什么时候不适用。感谢 Mark，我追加了许多关于这些争议主题的细节信息。

- Alexey Kopytov，他也审校了整本书。他是 SysBench 工具的作者（我在本书中介绍过），曾经在 MySQL 开发部门工作，现就职于 Percona 公司。Alexey 针对 SysBench 部分给出了很改进建议。
- Dimitri（dim）Kravtchuk，Oracle 的首席基准工程师，也审校了全书。他也是我在本书中提到的 dim_STAT 监控解决方案和 db_STRESS 数据库基准测试工具包的作者。他也在一个著名的博客上发表关于 InnoDB 性能和 MySQL 基准测试的文章。他给我了很多关于 InnoDB 介绍、性能架构（schema）和硬件影响部分的改进建议。

最后，感谢我的家人。

- 我的母亲 Yulia Ivanovna，告诉我作为工程师会多么有趣。
- 我的公公 Valentina Alekseevna Lasunova 和婆婆 Nikolay Nikolayevich Lasunov，他们总是在我们需要的时候给予帮助。
- 最后，重点感谢我的丈夫 Sergey Lasunov，他在我的创作中一直给予我支持。

目录

第1章

基础

当解决疑难问题的时候，为了节约时间，你可以从最简单的情况开始，然后一步步从简入繁。在 MySQL 支持团队工作的时候，我每个月解决成百上千的问题。其中的大部分都是从零散的请求信息开始的，最终的解决方案可能也很基础，我们将会从一些示例中看到这点。不过有些时候，我们确实会遭遇很大的挑战。所以，我们应时刻牢记从最基础的开始。

典型的基础类问题不外乎执行一个查询但是返回非预期的结果。这类问题的表现形式可能是很明显的错误，也可能是在你明知有匹配记录的情况下却没有返回结果，或者其他应用程序中罕见的行为。简而言之，本章内容是建立在你充分了解应用程序的运行状况以及查询应该返回结果的基础上的。对于不了解错误来源的情况将在本书后面讨论。

即使你遇到的是最诡异的错误，或者你无法确定应用程序中的错误原因，你也应该从最基础的部分开始。6.6 节会深入讨论这个过程，此过程又称为构造最小化测试用例（creating a minimal test case）。

1.1 语法错误

这个错误听起来十分简单，但仍可能很难发现。我建议你像处理其他问题一样，非常细心地查找可能出现的 SQL 语法错误。

类似如下错误，很容易被发现：

```
SELECT * FRO t1 WHERE f1 IN (1,2,1);
```

在这个示例中，很显然用户少转入了个“m”，错误消息也很清楚（输出结果根据页面设置进行宽度调整）：

```
mysql> SELECT * FRO t1 WHERE f1 IN (1,2,1);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'FRO
t1 WHERE f1 IN (1,2,1)' at line 1
```

遗憾的是，不是所有的语法错误都这么显而易见。我曾经处理过一个问题，它的查询语句是这样的：

```
SELECT id FROM t1 WHERE accessible=1;
```

这是一个版本迁移导致的问题；该语句在 5.0 版本中运行正常，但是在 5.1 版本中出现错误。问题的原因在于，在 5.1 版本中，“accessible”是一个保留字。给该语句加上引号（反引号还是双引号取决于你的 SQL 格式），即可重新正常运行：

```
SELECT `id` FROM `t1` WHERE `accessible`=1;
```

实际场景中的查询语句看起来可能十分繁琐，包括大段的 JOIN 和复杂的 WHERE 条件。所以，即使是简单的错误在其他大量语句的干扰下也很难被查找出来。我们当时第一步的任务就是简化复杂的查询，使它变成像刚才看到的那样只有一行 SELECT 的语句。这就是一个最简化测试的示例。当我们看到简化后只有一行的语句也有同样的 bug 时，我们就能很快意识到原有程序是因为保留字的问题而产生了错误。

- 第一教训就是教你把检查查询的语法错误作为排错的第一步。

但是，当你不知道查询是什么样的时候该怎么办？比如，查询是由应用程序自动生成的，或是在存储库中由第三方库动态生成的。

考虑如下 PHP 代码：

```
$query = 'SELECT * FROM t4 WHERE f1 IN(';
for ($i = 1; $i < 101; $i ++)
$query .= "'row$i,";
$query = rtrim($query, ',');
$query .= ')';
$result = mysql_query($query);
```

从这段脚本中很难直接定位错误。幸运的是，我们可以通过调整代码，使用输出函数打印查询语句。在 PHP 语言中，可以使用 echo 运算符。因此，修改代码，如下所示：

```
…
echo $query;
//$result = mysql_query($query);
```

当程序输出将要提交的语句的时候，问题暴露出来了：

```
$ php ex1.php
SELECT * FROM t4 WHERE f1 IN('row1,'row2,'row3,'row4,'row5,'row6,'row7,'row8,
'row9,'row10,'row11, 'row12,'row13,'row14,'row15,'row16,'row17,'row18,'row19,'row20)
```

如果你仍然没有发现错误，你可以在 MySQL 命令行客户端中尝试执行这个查询：

```
mysql> SELECT * FROM t4 WHERE f1 IN('row1,'row2,'row3,'row4,'row5,'row6,'row7,'row8,
'row9,'row10,'row11,'row12,'row13,'row14,'row15,'row16,'row17,'row18,'row19,'row20);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'row2,
'row3,'row4,'row5,'row6,'row7,'row8,'row9,'row10,'row11, 'row12,'row13,'row' at
line 1
```

问题在于每一行都少了一个右单引号。返回 PHP 代码，需要把：

```
$query .= "'row$i,";
```

改为：

```
$query .= "'row$i',";
```

即可。

- 因此，一个重要的调试技巧是，始终尝试查看 MySQL 服务器最终接收到的查询。不要仅调试应用程序代码，要获取查询语句！

遗憾的是，你不可能始终使用输出函数。比如，我之前提到的那个场景，SQL 语句是在编译好的第三方库中生成的。你的应用程序可能只是使用了库提供的高级的抽象接口，比如 CRUD（新建、读、更新、删除）接口。或者在生产环境下，你不希望用户看到在使用特定参数对特定查询进行测试时的查询。在这种情况下，可以检查 MySQL 的通用查询日志。这里用一个新示例来说明其是如何工作的。

这是一段有问题的 PHP 代码：

```
private function create_query($columns, $table)
{
    $query = "insert into $table set ";
    foreach ($columns as $column) {
        $query .= $column['column_name'] . '=';
        $query .= $this->generate_for($column);
        $query .= ', ';
    }
    return rtrim($query, ',') . ';';
}

private function generate_for($column)
{
    switch ($column['data_type']) {
    case 'int':
        return rand();
    case 'varchar':
    case 'text':
      return "'" . str_pad(md5(rand()), rand(1,$column['character_maximum_length']),
      md5(rand()), STR_PAD_BOTH) . "'";
    default:
        return "''";
    }
}
```

这段代码更新了例 1.1 中定义的表。

例 1.1　一般问题情形的示例表

```
CREATE TABLE items(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    short_description VARCHAR(255),
    description TEXT,
    example TEXT,
    explanation TEXT,
    additional TEXT
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

现在启用通用查询日志。该日志包含 MySQL 服务器接收到的每条独立查询。很多产品不会在日常运行中使用该日志，因为它在高负载的情况下增长十分迅速，并且写入日志会消耗 MySQL 服务器的资源，这些资源很可能用于更重要的用途。从 5.1 版本开始，你可以临时打开通用查询日志，方便随时记录你需要的查询。可以通过下面命令打开日志：

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

也可以将日志记录在表中，这可以帮你方便地分类索引日志文件项，因为你可以像查询其他表一样访问查询日志表：

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

现在可以运行应用程序。经过迭代地执行问题代码后，查询通用日志记录表，以查找有问题的查询：

```
mysql> SELECT * FROM mysql.general_log\G
*************************** 1. row ***************************
  event_time: 2011-07-13 02:54:37
   user_host: root[root] @ localhost []
   thread_id: 27515
   server_id: 60
command_type: Connect
    argument: root@localhost on collaborate2011
*************************** 2. row ***************************
  event_time: 2011-07-13 02:54:37
   user_host: root[root] @ localhost []
   thread_id: 27515
   server_id: 60
command_type: Query
    argument: INSERT INTO items SET id=1908908263,
short_description='8786db20e5ada6cece1306d44436104c',
description='fc84e1dc075bca3fce13a95c41409764',
example='e4e385c3952c1b5d880078277c711c41',
explanation='ba0afe3fb0e7f5df1f2ed3f2303072fb',
additional='2208b81f320e0d704c11f167b597be85',
*************************** 3. row ***************************
  event_time: 2011-07-13 02:54:37
   user_host: root[root] @ localhost []
   thread_id: 27515
   server_id: 60
command_type: Quit
    argument:
```

注意上述代码中 2.row 中的查询语句：

```
INSERT INTO items SET id=1908908263,
short_description='8786db20e5ada6cece1306d44436104c',
description='fc84e1dc075bca3fce13a95c41409764',
example='e4e385c3952c1b5d880078277c711c41',
explanation='ba0afe3fb0e7f5df1f2ed3f2303072fb',
additional='2208b81f320e0d704c11f167b597be85',
```

错误再次显而易见：在语句的结尾有个多余的逗号。这个问题是由下面这部分 PHP 代码产生的：

```
        $query .= ', ';
    }
    return rtrim($query, ',') . ';';
```

如果字符串确实是以逗号结尾的，那么 rtrim 函数本应移除结尾的逗号。但是现在这行实际上是以空格结尾的，因此 rtrim 函数没有移除任何字符。

既然我们已经发现了应用程序中产生错误的原因，我们就可以关闭通用查询日志：

```
mysql> SET GLOBAL general_log='off';
Query OK, 0 rows affected (0.08 sec)
```

在这节，我们学到一些重要的东西：

- 语法错误可能是导致一些现实问题的原因；
- 你应该测试与 MySQL 服务器接收到的请求完全一致的查询；
- 编程语言的输出函数和通用查询日志可以帮助你快速定位由应用程序发送到 MySQL 服务器的查询的问题。

1.2 SELECT 返回错误结果

这是用户反馈的另一个非常常见的问题，主要的现象有：用户看不到更新的结果、展示的顺序错误或者查询到了非预期的结果。

这个问题主要有两方面的原因：一方面是你的 SELECT 查询有误；另一方面是数据库中的数据和你想象的不同。我先介绍第一种情况。

在我规划本节示例的时候，我考虑要么使用真实的示例，要么使用我自己设计的小场景。真实的示例可能占用大量篇幅，但是我自己设计的示例可能对你没有什么帮助，因为没有人会写出那样的代码。因此，我选择使用典型的真实示例作为示例，只是大幅简化了它们。

第一个示例是用户在大量使用 join 时的常见错误。我们将使用之前介绍的例 1-1 中的表。这张表包含了 MySQL 中会引起一些常见使用错误的特性，这些特性是我在 MySQL 支持团队中收集的。每个错误在 items 表中都有一行记录。我还有一张关联资源信息的 links 表。因为条目和关联信息之间是多对多的关系，所以我通过 items_links 关联表把它们联系起来。下面是 items 表和 items_links 表的定义（在这个示例中不需要 links 表）：

```
mysql> DESC items;
+-------------------+--------------+------+-----+---------+----------------+
| Field             | Type         | Null | Key | Default | Extra          |
+-------------------+--------------+------+-----+---------+----------------+
| id                | int(11)      | NO   | PRI | NULL    | auto_increment |
| short_description | varchar(255) | YES  |     | NULL    |                |
| description       | text         | YES  |     | NULL    |                |
| example           | text         | YES  |     | NULL    |                |
| explanation       | text         | YES  |     | NULL    |                |
| additional        | text         | YES  |     | NULL    |                |
+-------------------+--------------+------+-----+---------+----------------+
6 rows in set (0.30 sec)

mysql> DESC items_links;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| iid    | int(11) | YES  | MUL | NULL    |       |
| linkid | int(11) | YES  | MUL | NULL    |       |
+--------+---------+------+-----+---------+-------+
2 rows in set (0.11 sec)
```

我编写的第一条查询正常运行，并且似乎返回了合理的结果：

```
mysql> SELECT count(*) FROM items WHERE id IN (SELECT id FROM items_links);
+----------+
| count(*) |
+----------+
|       10 |
+----------+
1 row in set (0.12 sec)
```

……直到我把返回的数值与关联总数进行比较的时候，我才发现：

```
mysql> SELECT count(*) FROM items_links;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.09 sec)
```

查询到的关联信息数比关联表的记录还多，这怎么可能？

我们再来检查一下我特意编写的这个查询。它很简单，仅仅包含两部分，一个子查询：

```
SELECT id FROM items_links
```

和一个外部查询：

```
SELECT count(*) FROM items WHERE id IN ...
```

子查询是开始错误排查的好切入点，因为它可以独立运行。因此，我们可以预期一个完整的结果集：

```
mysql> SELECT id FROM items_links;
ERROR 1054 (42S22): Unknown column 'id' in 'field list'
```

令人惊讶的是，我们居然有一个输入错误，事实上 items_links 表中并没有 id 字段，而是 iid 字段（代表 items 的 ID）。如果我们重写该条查询，让它使用正确的标识符，它便可正常运行：

```
mysql> SELECT count(*) FROM items WHERE id IN (SELECT iid FROM items_links);
+----------+
| count(*) |
+----------+
|        4 |
+----------+
1 row in set (0.08 sec)
```

- 我们刚刚学习了一个新的调试技巧。如果一个 SELECT 查询没有按预期工作，可以将其拆分成小段语句，然后分析每一部分直到你找到产生错误行为的原因。

提示

如果你通过表名.列名的格式指定完整的列名，那么你可以从一开始就避免这个错误，因为你会立即获得错误：

```
mysql> SELECT count(*) FROM items WHERE items.id IN
       (SELECT items_links.id FROM items_links);
ERROR 1054 (42S22): Unknown column 'items_links.id' in 'field list'
```

MySQL 命令行客户端是一个非常好的测试工具，该工具包含在 MySQL 的安装包中。第 6 章将讨论这个重要的工具。

然而，为什么 MySQL 在执行原始查询语句的时候没有返回同样的错误呢？这是因为在 items 表有一个名为 id 的列，因此 MySQL 认为我们想要执行一个依赖子查询，结果实际上从 items_links 表中查询了 items.id。“依赖子查询”是指引用外部查询中字段的查询。

我们也可以借助 EXPLAIN EXTENDS 命令，通过 SHOW WARNINGS 来查找这个错误。如果我们用该命令运行原始查询，会得到：

```
mysql> EXPLAIN EXTENDED SELECT count(*) FROM items WHERE id IN
(SELECT id FROM items_links)\G
2 rows in set, 2 warnings (0.12 sec)
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: items
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 10
     filtered: 100.00
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
```

```
        table: items_links
         type: index
possible_keys: NULL
          key: iid_2
      key_len: 5
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using where; Using index
2 rows in set, 2 warnings (0.54 sec)

mysql> show warnings\G
*************************** 1. row ***************************
  Level: Note
   Code: 1276
Message: Field or reference 'collaborate2011.items.id' of SELECT #2 was resolved
in SELECT #1
*************************** 2. row ***************************
  Level: Note
   Code: 1003
Message: select count(0) AS `count(*)` from `collaborate2011`.`items` where
<in_optimizer7gt;(`collaborate2011`.`items`.`id`,<exists>(select 1 from
`collaborate2011`.`items_links` where
(<cache>(`collaborate2011`.`items`.`id`) =
`collaborate2011`.`items`.`id`)))
2 rows in set (0.00 sec)
```

EXPLAIN EXTENDED 输出的 2.row 表明该子查询实际上是依赖的：select_type 是 DEPENDENT SUBQUERY。

在结束这个示例之前，我想再介绍一个可以在请求语句涉及很多表的时候，帮助你避免迷茫的小技巧。要知道当你面对 10 个甚至更多表的连接时，即使你很了解它们应该怎么连接，你也会感到迷茫。

上面示例中一个值得注意的地方是 SHOW WARNINGS 的输出信息。MySQL 服务器不是总按照语句输入的顺序执行它，而是调用优化器去构造一个更好的执行计划，因此用户通常都会很快得到返回结果。在 EXPLAIN EXTENDED 之后，SHOW WARNINGS 命令展示的就是优化后的查询。

在该示例中，SHOW WARNINGS 的输出包含两个主要信息。第一个是：

```
Field or reference 'collaborate2011.items.id' of SELECT #2 was resolved in SELECT #1
```

这条信息明确指出服务器是通过 items 表而不是 items_links 表解析 id 的值。

第二条信息包含了优化过的语句：

```
select count(0) AS `count(*)` from `collaborate2011`.`items` where <in_optimizer>
(`collaborate2011`.`items`.`id`,<exists>
(select 1 from `collaborate2011`.`items_links` where
(<cache>(`collaborate2011`.`items`.`id`) = `collaborate2011`.`items`.`id`)))
```

这个输出信息也指出服务器是从 items 表接受 id 的值。

现在我们来对比一下正确的查询和之前列出的错误查询的 EXPLAIN EXTENDED 的结果：

```
mysql> EXPLAIN EXTENDED SELECt count(*) FROM items WHERE id IN
(SELECT iid FROM items_links)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: items
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 10
     filtered: 100.00
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: items_links
         type: index_subquery
possible_keys: iid,iid_2
          key: iid
      key_len: 5
          ref: func
         rows: 1
     filtered: 100.00
        Extra: Using index; Using where
2 rows in set, 1 warning (0.03 sec)

mysql> show warnings\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select count(0) AS `count(*)` from `collaborate2011`.`items` where
<in_optimizer>(`collaborate2011`.`items`.`id`,<exists>
(<index_lookup>(<cache>(`collaborate2011`.`items`.`id`) in
items_links on iid where (<cache>(`collaborate2011`.`items`.`id`) =
`collaborate2011`.`items_links`.`iid`))))
1 row in set (0.00 sec)
```

这次优化过的语句看起来完全不同了，并且确实像我们预期的那样比较 items.id 和 items_links.iid 的值。

- 我们刚刚学习了另一教训：在 EXPLAIN EXTENDED 命令之后使用 SHOW WARNINGS 命令查看查询是如何优化（与执行）的。

在正确的查询中，select_type 的值仍然是 DEPENDENT SUBQUERY。我们已经通过 items_links 表来解析字段的名称了，为什么结果仍是那样？答案从 SHOW WARNINGS 中下面这部分输出开始：

```
where (<cache>(`collaborate2011`.`items`.`id`) =
`collaborate2011`.`items_links`.`iid`)
```

子查询仍然显示是依赖的，因为外部查询子句中的 id 需要子查询去检查与内部查询对应的每行里的 iid 值。这个问题在 MySQL 社区 bug 数据库的 12106 号报告的讨论中提出。

- 这个 bug 报告给我们了另一个重要的教训：如果你怀疑你的查询的执行行为，可以通过有效的资源去获取信息。社区 bug 数据库就是这样的一种资源。

SELECT 查询运行异常可能有很多不同的原因，但是查找问题的一般方法总是相同的。

- 将查询分解成小段，然后依次执行它们直到你发现问题的原因。
- 使用 EXPLAIN EXTENDED，然后使用 SHOW WARNINGS 命令去获得查询执行计划及其实际运行方式的相关信息。
- 如果你不理解 MySQL 服务器的执行状况，可以使用互联网和其他有效的资源去获得信息。附录提供了非常有用的资源列表。

1.3 当错误可能由之前的更新引起时

如果 SELECT 查询返回了非预期的结果集，这并不总是意味着查询语句本身有错误，也有可能是因为你以为已经进行了插入、更新或者删除等操作，而事实上它们并未生效。

在你调查这种可能之前，你应该先完全仔细检查前一节讨论的 SELECT 语句编写错误的问题。在 SELECT 语句编写正确并且能够返回你想要的值的情况下，现在我开始调查由数据本身的问题导致错误的可能性。为了确认问题是由数据本身而非 SELECT 语句产生的，我尝试精简语句，使其变成某个独立表的简单查询。如果是小表，那么移除所有的 WHERE 条件和 GROUP BY 语句，然后通过“野蛮”的 SELECT * FROM table - name 检查所有的数据。对于大表来说，用 WHERE 条件来筛选出你想要的值是明智的选择。如果你仅关心查询结果集的条数是否和预期的一致，也可以考虑用 COUNT(*)来显示条数。

一旦你确定 SELECT 查询工作正常，那就意味着是数据不一致产生的问题，你就需要定位哪里出了问题。有很多可能的原因：使用了错误的备份、错误的 UPDATE 语句，或者从节点与主节点之间同步异常（这里先仅列出这些最常见的可能）。在这一节中，我们会看到一些关于 DELETE 或 UPDATE 操作在随后的 SELECT 查询中没有生效的示例。下一节会介绍一些令人困惑的情况，在这些情况中，问题会在被触发很久后才出现，当然我们也会告诉你如何反向定位这类错误。本章并不涉及事务中的问题，这类问题将在第 2 章中讨论。这里展示的场景都是基于数据库中的数据已经稳定的前提，也就是说，所有使用到的事务都已经完成了。我将继续使用从现实场景中简化过的示例。

我们从可能的最佳情形开始，即错误发生后立即提示数据不一致的问题。我们将使用下面的初始数据集：

```
mysql> CREATE TEMPORARY TABLE t1(f1 INT);
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TEMPORARY TABLE t2(f2 INT);
Query OK, 0 rows affected (0.08 sec)
mysql> INSERT INTO t1 VALUES(1);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM t1;
+------+
| f1   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

在应用程序中，临时表包含从主日志表中查询出来的部分结果集。这是一个保存日常常用数据经常使用的技术手段，当你只需要用到主表中的一小部分数据并且用户不想改变主表中的数据或者锁定主表的时候，可以使用临时表。

所以在这个示例中，当使用完结果集后，用户想要同时删除两个表中的相应行。通常人们很难想象用一个查询语句去做多件事情。不过现实可以与你的设想不同，并且你还会得到非预期的结果或负面影响：

```
mysql> DELETE FROM t1, t2 USING t1, t2;
Query OK, 0 rows affected (0.00 sec)
```

如果用户注意观察输出的 DELETE 语句的相应结果，就会立即发现出了问题。DELETE 操作没有影响到任何行意味着它什么都没做。然而，一条语句的输出通常不是这么显而易见，有时候它并不可见，因为 SQL 语句是在程序或者脚本内部执行的，并且没有人会去监控执行结果。通常情况下，你应该始终检查语句执行的返回信息，从而了解有多少行数据受影响且它们的值是否与你预期的一致。在应用程序中，你必须明确检查信息功能。

继续下面的讨论。如果你立即执行 SELECT 查询，你可能会很惊讶，以为查询语句出现了错误或者查询缓存没有清除：

```
mysql> SELECT * FROM t1;
+------+
| f1   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

如果把 SELECT 语句改为查询行的数量，就可以确认这不是缓存或者其他相关的问题。这个小例子也告诉我们可以通过对同一张表进行不同的查询方式来确认数据的一致性：

```
mysql> SELECT count(*) FROM t1;
+----------+
| count(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

这里 COUNT(*)仍然返回一个正数，这表明表是非空的。细心的用户应该已经注意到，DELETE 操作实际上没有删除任何行。为了找出原因，我们可以将 DELETE 语句改为相应的 SELECT 语句。这样做可以告诉我们哪些行满足了删除条件。

尽管这个简单的示例中没有 WHERE 语句，但是这个技巧对于包含 WHERE 语句的删除和更新操作同样有效。SELECT 语句返回的行即为 DELETE 操作将要删除的行或者 UPDATE 操作将要更新的行：

```
mysql> SELECT * FROM t1, t2;
Empty set (0.00 sec)
```

与之前的结果一致，这里也返回空集合。这就是为什么没有删除任何行！然而，现在仍不清楚产生这个现象的具体原因，但是既然我们有一个 SELECT 查询，就可以利用第一节提到的相关技术。在这个场景中，最佳选择就是用 EXPLAIN 命令执行 SELECT 语句然后分析输出结果：

```
mysql> \W
Show warnings enabled.

mysql> EXPLAIN EXTENDED SELECT * FROM t1, t2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra:
*************************** 2. row ***************************
           id: 2
  select_type: SIMPLE SUBQUERY
        table: t2
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 0
     filtered: 0.00
        Extra: const row not found
2 rows in set, 1 warning (0.03 sec)

Note (Code 1003): select '1' AS `f1`,'0' AS `f2` from `test`.`t1` join `test`.`t2`
```

输出中最后的信息表明查询语句被修饰成了内部连接（inner JOIN），该内部连接仅当另一张表也有满足条件的行时才会同时返回两张表的行。对于 t1 表中的每一行，在 t2 表中应至少有一行的值匹配。在这个示例中，因为 t2 表是空的，自然连接操作返回空

集合。

我们刚刚学习了另一个有助于找出 UPDATE 或 DELETE 语句错误原因的重要技巧：把语句转换成具有相同 JOIN 和 WHERE 条件的 SELECT 语句。针对 SELECT 查询，可以使用 EXPLAIN EXTENDED[1]命令去获取实际的执行计划，同时也可以避免直接操作结果集带来的危险或者修改了错误的行。

这里有一个的使用 UPDATE 的更复杂示例。我们仍使用 items 表：

```
mysql> SELECT SUBSTR(description, 1, 20), additional IS NULL FROM items;
+----------------------------+--------------------+
| substr(description, 1, 20) | additional IS NULL |
+----------------------------+--------------------+
| NULL                       |                  1 |
| NULL                       |                  1 |
| One who has TRIGGER        |                  1 |
| mysql> revoke insert       |                  1 |
| NULL                       |                  0 |
+----------------------------+--------------------+
5 rows in set (0.00 sec)

mysql> SELECT description IS NULL, additional IS NULL FROM items;
+---------------------+--------------------+
| description IS NULL | additional IS NULL |
+---------------------+--------------------+
|                   1 |                  1 |
|                   1 |                  1 |
|                   0 |                  1 |
|                   0 |                  1 |
|                   1 |                  0 |
+---------------------+--------------------+
5 rows in set (0.00 sec)
```

description 和 additional 字段是 TEXT 类型的。在这个示例中，我们将使用一个错误的语句，该语句想要把表中的中 NULL 值替换成更有语义的文本（一个替换成“no description”，另一个替换成“no additional comments”）：

```
mysql> UPDATE items SET description = 'no description' AND
additional = 'no additional comments' WHERE description IS NULL;
Query OK, 3 rows affected, 3 warnings (0.13 sec)
Rows matched: 3  Changed: 3  Warnings: 3
```

该语句会更新一些数据（“影响到 3 行”），让我们检查一下现在表中数据是否合理：

```
mysql> SELECT SUBSTR(description, 1, 20), additional IS NULL FROM items;
+----------------------------+--------------------+
| substr(description, 1, 20) | additional IS NULL |
+----------------------------+--------------------+
| 0                          |                  1 |
```

1：版本 5.6.3 开始，也可以在 UPDATE 和 DELETE 上使用 EXPLAIN 方法，不过把语句转换成 SELECT 查询仍然有效，因为你可以方便地检查和操作实际的结果集，而不是仅使用 EXPLAIN 命令。这尤其适用于复杂的 JOIN 操作，尤其是当 EXPLAIN 输出的检查的行比实际更新的行还要多的时候。

```
| 0                          |                  1 |
| One who has TRIGGER        |                  1 |
| mysql> revoke insert       |                  1 |
| 0                          |                  0 |
+----------------------------+--------------------+
5 rows in set (0.09 sec)
```

正如我们所见，有 3 行记录的 description 字段的值被修改，不过值是 0 而不是我们预期的“no description”。并且，additional 字段的值根本没有改变。为了定位该问题发生的原因，我们应该检查警告。注意服务器返回的这些语句，我们看到有共 3 个警告：

```
Query OK, 3 rows affected, 3 warnings (0.13 sec)
Rows matched: 3  Changed: 3  Warnings: 3

mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------------------+
| Level   | Code | Message                                               |
+---------+------+-------------------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'no description' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'no description' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'no description' |
+---------+------+-------------------------------------------------------+
3 rows in set (0.00 sec)
```

这条消息看起来很奇怪。为什么上述语句执行后，这里会报告关于 DOUBLE 的警告，而 description 和 additional 字段的类型都是 TEXT 的。

```
mysql> SHOW FIELDS FROM items LIKE 'description';
+-------------+------+------+-----+---------+-------+
| Field       | Type | Null | Key | Default | Extra |
+-------------+------+------+-----+---------+-------+
| description | text | YES  |     | NULL    |       |
+-------------+------+------+-----+---------+-------+
1 row in set (0.13 sec)

mysql> SHOW FIELDS FROM items LIKE 'additional';
+------------+------+------+-----+---------+-------+
| Field      | Type | Null | Key | Default | Extra |
+------------+------+------+-----+---------+-------+
| additional | text | YES  |     | NULL    |       |
+------------+------+------+-----+---------+-------+
1 row in set (0.13 sec)
```

我们还想知道为什么 additional 字段完全没有变化，并且我们也没有得到任何警告。

我们把该语句拆分成小段，然后分别检查每段都做了什么：

```
UPDATE items
```

这是 UPDATE 语句惯用的开头，没有什么问题：

```
SET description = 'no description' AND additional = 'no additional comments'
```

该段使用 SET 语句。我们来检查一下它实际做了什么。AND 关键字在这里究竟意味什么？我们在语句中加上圆括号来突出一下运算符号优先级：

```
SET description = ('no description' AND additional = 'no additional comments')
```

所以，实际上这个语句计算了下列表达式：

```
'no description' and additional = 'no additional comments'
```

然后将值赋给 description 字段。计算等式会产生一个布尔类型的结果，表示为 LONGLONG 类型的值。为了证明这点，以--column-type-info 选项打开 MySQL 命令行客户端，然后再次运行 SELECT 查询：

```
$ mysql --column-type-info
mysql> SELECT 'no description' AND additional = 'no additional comments' FROM items;
Field   1:  `'no description' AND additional = 'no additional comments'`
Catalog:    `def`
Database:   ``
Table:      ``
Org_table:  ``
Type:       LONGLONG
Collation:  binary (63)
Length:     1
Max_length: 1
Decimals:   0
Flags:      BINARY NUM


+-------------------------------------------------------------+
| 'no description' AND additional = 'no additional comments' |
+-------------------------------------------------------------+
|                                                           0 |
|                                                           0 |
|                                                           0 |
|                                                           0 |
|                                                           0 |
+-------------------------------------------------------------+
5 rows in set, 5 warnings (0.09 sec)
```

我们可以清楚地看到表达式的结果是 0，这个值随后被插入了 description 字段。并且因为我们对 additional 字段的更新已被这个奇怪的表达式所覆盖了，所以没有值插入该字段中，也就看不到服务器端给出任何关于该字段的信息。

现在可以修改上述语句中的逻辑错误了：

```
UPDATE items SET description = 'no description',
additional = 'no additional comments' WHERE description IS NULL;
```

如果需要你也可以检查 WHERE 语句，不过在这个示例里它没有错误。

这个示例表明返回值和查询执行信息的重要性。我们来进一步讨论它们。

1.4 获取查询信息

正如前一节看到的一样，数据库会返回一些关于每个查询的重要信息，有些信息直接展现在 MySQL 的访问客户端中，而有些信息则需要通过如 SHOW WARNINGS 等命令

才能得到。当从应用程序中调用 SQL 语句的时候，获取这些返回信息并确认没有异常情况发生同样重要。所有语言的 MySQL API 都提供了获取服务器返回信息的接口。本节将讨论这些接口。这里仅涉及 C 的 API，因为我必须选择一种语言的 API，并且大部分其他语言的 API 都是基于 C 的 API 的[1]。

受影响的行数

我们从之前见过的输出开始，每次插入、更新或者删除后都会显示有多少行数据插入、更新或者删除了：

```
Query OK, N rows affected
```

这个信息代表查询正常执行并且修改了 *N* 行数据。

要在应用程序中获取相同的信息，可以调用：

```
mysql_affected_rows()
```

如果有改变发生，该函数会返回一个正数，如果没有改变那么返回 0，-1 代表反生错误。

对于 UPDATE 语句，如果客户端设置了 CLIENT_FOUND_ROWS，那么该函数将会返回满足 WHERE 条件的行数，这个数并不总是和实际更改的行数一致。

提示

对于 Connector/J 来说，默认不启用受影响的行，因为这不是 JDBC 兼容的特性并且会使其 DML 语句依赖于匹配的行数而不是受影响的行数的大多数应用程序产生错误。不过对于 INSERT...ON DUPLICATE KEY UPDATE 类型的语句会返回正确的更新数量。连接字符串属性 useAffectedRows 告诉 Connector/J 在连接到服务器的时候是否设置 CLIENT_FOUND_ROWS 标志。

匹配的行数

输出中表示该数目的字符串是：

```
Rows matched: M
```

该输出表明有多少行满足 WHERE 条件。

下面的 C 函数：

```
mysql_info()
```

以字符串格式返回关于最近的查询的补充信息。

对于更新操作来说，它返回的字符串类似：

```
Rows matched: # Changed: # Warnings: #
```

1：你可以在 http://dev.mysql.com/doc/refman/5.5/en/c.html 找到关于 C API 的详细描述。

其中，每个#依次对应代表匹配的行数、修改的行数和警告数目。可以通过解析该行中的“matched: #”获悉有多少行被查找出来。

被修改的行数

输出中代表该数目的字符串是：

```
Changed: P
```

该输出表明有多少行实际上修改了。需要注意的是，匹配行数 *M* 和修改的行数 *P* 是可以不同的。例如，假设你想要修改的列已经包含你指定的值，在这种情况下，该列会被认为是匹配的而不是修改的。

在应用程序中，像之前一样用 mysql_info()获取信息，不过这次是解析“Changed: #”。

警告：数目和消息

输出中表示这部分信息的字符串是：

```
Warnings: R
```

如果服务器在处理请求过程中探查到一些不寻常的情况或者值得报告的情况，你将会获得警告。不过查询仍然会执行并且会修改数据。无论如何请确保检查警告信息，因为它们会帮助你获悉潜在的问题。

在应用程序中，有很多不同的方式去获取警告。仍可以使用 mysql_info()函数，然后解析“Warinings: #”。也可以调用：

```
mysql_warning_count()
```

如果有警告，可以执行 SHOW WARNINGS 命令去获取关于究竟发生了什么的文本消息。另一个选择是：

```
mysql_sqlstate()
```

该函数将返回最近的 SQL 状态（SQLSTATE）。例如，“42000”代表语法错误，“00000”代表没有错误和警告。

提示

SQLSTATE 的值由 ANSI SQL 标准定义，用于表明语句的执行状态。执行状态被设置成标准中定义的状态码，表明一个请求是成功完成还是返回异常。SQLSTATE 以字符串形式返回。要了解 MySQL 服务器可能返回哪些状态码，可以阅读 MySQL 参数手册中的“服务器错误码和错误消息”一节。

错误

检查错误也总是很有用。下面的函数返回最近 SQL 语句的错误值。

`mysql_errno()`

该函数返回最近一次错误的 MySQL 错误代码。例如，语法错误会生成数字 1064，0 意味着没有错误。

`mysql_error()`

该函数返回最近一次错误的文本信息。对于语法错误，它会返回类似以下的内容。

```
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax
to use near 'FRO t1 WHERE f1 IN (1,2,1)' at line 1
```

这有利于保存存储于不同日志文件中的信息，使你可以在任何时候检查它们。

提示

MySQL 官方文档包含 MySQL 服务器可能返回的错误列表以及客户端错误列表。

通过 perror 获取错误字符串

perror 工具是 MySQL 发行包中携带的一款用于解决问题的非常有用的工具。perror 能够提供与给定错误代码相关联的 MySQL 及其所在操作系统的错误信息。可以从 MySQL 命令行客户端、信息函数或其他错误日志文件中的错误消息中后面括号的部分获取错误代码。下面是一些示例：

```
mysql> CREATE TABLE t2(f1 INT NOT NULL PRIMARY
    -> KEY, f2 INT, FOREIGN KEY(f2) REFERENCES t1(f2)) ENGINE=InnoDB;
ERROR 1005 (HY000): Can't create table 'test.t2' (errno: 150)

mysql> \q

$perror 150
MySQL error code 150: Foreign key constraint is incorrectly formed

$perror 2
OS error code   2:  No such file or directory

$perror 136
MySQL error code 136: No more room in index file
```

当命令产生错误的时候会在 MySQL 客户端输出这些错误代码，这些错误代码可以通过 mysql_error()函数由程序获取。然而，当你面对一个错误码不知所措的时候，可以使用 perror 工具。

1.5 追踪数据中的错误

如果你严格地检查查询和更新的结果，就会发现很多问题，这些问题可能持续数周而未被察觉，然后慢慢变得越来越严重，直到最后无可避免地引发很多让人苦恼的问题。然而，问题确实在慢慢地接近你。有时，SELECT 查询突然开始返回错误的结果，但是

你对该查询的经验使你确信它没有什么问题。

在这种情况下，你应该反向模拟用户操作，直到发现错误的根源。如果幸运，你会一步就发现问题的原因。不过通常你会进行多步操作，有时甚至消耗很长的时间。

大部分这种问题是由于复制环境中主从节点的数据不一致造成的。一个常见的错误情形是期望唯一值的时候出现了重复值（例如，如果用户使用 INSERT ON DUPLICATE KEY UPDATE 语句，但是主从服务器中的表结构是不同的）。在这样的环境设置下，用户往往直到从节点执行 SELECT 语句的时候才会发现问题，而不会在 INSERT 发生时就注意到问题。在循环复制时情况会更糟糕。

为了说明这个问题，我们将使用一个存储过程从保存其他查询结果的临时表向另一个表插入数据。这是另一个常用技巧，用于当用户想要处理大表中的数据，同时担心意外修改数据的风险，或者担心在使用这些大表时对其他应用造成堵塞的情形。

我们来创建表并填充临时数据。在实际应用中，临时表会用于保存等待存入主表的计算结果集：

```
CREATE TABLE t1(f1 INT) ENGINE=InnoDB;
CREATE TEMPORARY TABLE t2(f1 INT) ENGINE=InnoDB;
```

现在向临时表中插入数据：

```
INSERT INTO t2 VALUES(1),(2),(3);
```

存储例程将临时表中的数据移入主表。它在迁移前会先确认数据在临时表中。我们的版本如下：

```
CREATE PROCEDURE p1()
BEGIN
DECLARE m INT UNSIGNED DEFAULT NULL;
CREATE TEMPORARY TABLE IF NOT EXISTS t2(f1 INT) ENGINE=InnoDB;
SELECT MAX(f1) INTO m FROM t2;
IF m IS NOT NULL
THEN
  INSERT INTO t1(f1) SELECT f1 FROM t2;
END IF;
END
|
```

在调用该存储例程时，如果指定的临时表不存在则会创建新的临时表。这样做可以避免由于临时表不存在而产生问题，但同时也会带来新问题。

提示

该示例使用 MAX 函数检查表中是否至少存在一行记录。推荐用 MAX 计数，因为 InnoDB 表不会保存记录的行数，而是在每次调用 COUNT 函数的时候现进行计算。因此，MAX(indexed_field)函数比 COUNT 快。

如果从服务器在第一个插入之后，存储过程调用之前重启，那么从服务器中的临时表将

会是空的并且从服务器上的主表没有任何数据。在这种情况下，我们访问主节点会得到：

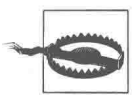
```
mysql> SELECT * FROM t1;
+------+
| f1   |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.03 sec)
```

与此同时，在从服务器上得到：

```
mysql> SELECT * FROM t1;
Empty set (0.00 sec)
```

更糟的是，如果我们在存储过程调用后向 t1 表中插入数据，从服务器中的数据将会完全混乱。

假设我们注意到应用程序中主从表读取数据时的错误。现在我们应该弄清数据是怎么插入从表的：是直接更新从服务器还是从主节点复制的数据？



警告

MySQL 复制不会帮你检查数据一致性，因此对同一个对象，同时使用 SQL 复制线程和从节点上的用户线程更新会使数据与主服务器不同，这会导致随后的复制事件失败。

因为我们在示例中模拟这种情形，所以我们知道发生数据损坏问题的关键点：从服务器在第一次插入之后，存储过程调用之前重启了。在实际场景中，问题一般会在用户执行下面查询的时候被发现：

```
mysql> SELECT * FROM t1;
Empty set (0.00 sec)
```

当你从 SELECT 查询中获得非预期结果时，你需要找出该问题发生的原因，是由于查询本身的问题，还是由于早些时候的一些错误引起的。刚才展示的插入非常简单，除非表损坏了，否则它不可能产生错误，因此我们必须回头检查一下表是如何修改的。

通常的示例是在建立在从服务器只读的复制环境下，因此我们可以确保错误产生有两种可能的原因：要么是主服务器插入了错误的数据，要么是数据在复制时损坏。

所以，首先检查主服务器的数据是否有错误：

```
master> SELECT * FROM t1;
+------+
| f1   |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.03 sec)
```

主服务器数据正常，因此问题的原因在于复制层。然而，这是怎么发生的？复制看起来运行正常[1]，因此我们猜想是主节点有逻辑错误。当发现了这个可能的原因的时候，你应该去分析存储过程并在主节点上调用以寻找修复方案。

如前所述，在向临时表中插入数据完成复制并清空临时表的事件之后，且在调用查询并向主表插入数据的存储过程之前，重启服务器。因此，从服务器仅是重新创建一个空的临时表并且没有插入任何数据。

在这种情况下，可以选择转换成基于行的复制或者重写存储过程，使其不依赖于已经存在的临时表。另一种方法是清空然后重新填充表，这样突然重启不会导致从服务器数据丢失。

有人可能觉得这个示例太过人为了，你不可能预知服务器何时会突然重启。没错，但是重启确实每时每刻都有可能发生。因此，你需要考虑这样的错误。

事实上，从服务器一个接一个地复制二进制日志事件，当数据在一个原子事件（例如，一个事务或者存储过程调用）中产生时，从服务器不会受上述情况的影响。不过回过头来说，这个示例仅仅是为了说明现实生活中发生的事件背后的原理。

- 当你在明知是正确的情况下遇到了一个问题时，请检查在看到这个问题之前你的应用程序的运行情况。

关于复制错误的更多详细信息会在第 5 章进行介绍。

单服务器示例

我曾经处理过一个存储由不同的切割系统产生的度量数据的 Web 应用程序。用户可以添加一个系统，然后编辑保存度量数据的规则。

我第一次遇到错误的时候，我测试了一个含有系统列表的 Web 页面：

```
Existing systems

* System 1
* Test
* test2
* test2
* test2

Enter name of new system:
<>
Description:
<>

<Go!>
```

该列表不该包含重复的系统，因为描述同样的规则两次是没有意义的。因此我非常惊奇地看

1：第 5 章将详细介绍如何解决复制失败的问题，因此这里不再详细解释。

到有很多同名的条目。

输出数据的代码使用的是对象，并且，我无法仅通过阅读代码查看发送到 MySQL 服务器的语句是什么样的：

```
return $this->addParameters(array(Field::ITEMS => DAO::system()->getPlainList()));
```

我通过日志获取了真实的查询，它看来是正确的：

```
SELECT `system`.`id`, `system`.`name`, `system`.`description` FROM `system`
```

接下来，我检查了表的内容：

```
mysql> SELECT * FROM system;
+----+----------+-------------------------------------------------+
| id | name     | description                                     |
+----+----------+-------------------------------------------------+
|  1 | System 1 | Man and woman clothing construction             |
|  2 | Test     | Testing Geometric set                           |
|  3 | test2    | New test                                        |
|  4 | test2    | foobar                                          |
|  8 | test2    |                                                 |
+----+----------+-------------------------------------------------+
5 rows in set (0.00 sec)
```

SELCET 语句准确地返回了表中存在的数据集。我转而检查更新表的代码：

```
$system = System::factory()
->setName($this->form->get(Field::NAME))
->setDescription(
$this->form->get(Field::DESCRIPTION)
);
DAO::system()->take($system);
```

我再次通过日志获取真实的请求语句：

```
INSERT INTO `system` (`id`, `name`, `description`) VALUES ('', 'test2', '')
```

该语句也是正确的！id 似乎是自增的字段，因此会自动设置。

不过同时，该语句也暴露了潜在的问题：它一定在没有检查唯一性的情况下重复执行了。带着这种假设，我决定检查一下表的定义：

```
mysql> SHOW CREATE TABLE system\G
*************************** 1. row ***************************
       Table: system
Create Table: CREATE TABLE `system` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `description` tinytext NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8
1 row in set (0.09 sec)
```

问题的源头很显然了：name 字段没有定义成 UNIQUE（唯一的）。当创建表的时候，我通常使用 id 作为唯一标识符，但是我也会使用 MySQL 的特性在 INSERT 时去给 id 生成一个自增的值，没有什么使我避免重复使用同一个 name。

为了解决这个问题，我手动删除了多余的行并且增加了唯一（UNIQUE）索引。

```
ALTER TABLE system ADD UNIQUE(name)
```

我们已经介绍完了与错误结果相关的问题，接下来将介绍其他一些经常发生的问题。

1.6 慢查询

SQL 应用程序的一个常见问题就是性能退化。这一节将会介绍当你面对性能问题时的一些基本操作。不必担心细节，仅仅关注本质的思想就好。随着知识的深入，你会发现你自己变得更加训练有素，能够更加高效地使用它们。

当我考虑本章应涵盖哪些问题时，我曾犹豫是否应该涉及性能相关的问题。市面上有很多详细介绍性能问题的资料，如 MySQL 参考手册中的“优化”章节以及 O'Reilly 出版的相关书籍。我会在本书的最后简单介绍一些有用的资料。你很容易就会把整个职业生涯都消耗在该问题上面，或者淹没在可用信息之中。

这里将主要介绍 SELECT 查询。本节最后会简单介绍一下如何处理修改数据的慢查询问题。

处理慢查询有 3 个主要技巧：调优查询本身、调优表（包括增加索引）和调优服务器。下面逐一详细介绍。

1.6.1 通过 EXPLAIN 的信息调优查询

最强大的查询调优工具就是我们之前熟知的 EXPLAIN。这个工具为用户提供了服务器实际上如何执行查询的详细信息。MySQL 参考手册已经详细介绍了 MySQL 的 EXPLAIN 工具，因此这里不再赘述。不过，我将会重点介绍输出信息中我认为是最重要、最有用的部分。

第一行中你要注意的是 type，它展示了连接（join）的执行方式；还要注意 rows，它展示了在查询执行过程中检查的行数的估计（例如，如果查询必须扫描整个表，那么 rows 展示的数值和该表中的行数相等）。多表连接需要检查的行数是每个表中检查行数的笛卡儿积。也就是说，如果请求在第一个表中检查 20 行，另一个表中检查 30 行，那么连接一共执行了 600 次检查。EXPLAIN 会包含 JOIN 中每个表中的行。我们将通过下面的示例进行说明。

即使在操作单表时，EXPLAIN 也会报告连接。这可能听起来有些奇怪，因为 MySQL 的内部优化器把每个请求都当成一个连接，哪怕是单表上的连接。

我们来回顾一下前面介绍过的 EXPLAIN 输出：

```
mysql> EXPLAIN EXTENDED SELECT count(*) FROM items WHERE id IN
(SELECT iid FROM items_links)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: items
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 10
     filtered: 100.00
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: items_links
         type: index_subquery
possible_keys: iid,iid_2
          key: iid
      key_len: 5
          ref: func
         rows: 1
     filtered: 100.00
        Extra: Using index; Using where
2 rows in set, 1 warning (0.48 sec)
```

检查的行数是 10 乘以 1，因为子查询对于外部查询的每一行执行一次。第一个查询的类型是 index，这意味着全部索引都将被读取。第二个查询的类型是 index_subquery。这是一个索引查找函数，工作方式类似于 ref 类型。因此，在这个示例中，优化器将从 itmes 表中读取全部索引记录，并且对于从 itmes 表中查询到 10 行记录中的每一行，对应从 items_links 表中读取一行记录。

该如何确认这是一个对该查询合理的执行方式呢？首先，重复查询结果并检查查询的实际执行时间：

```
mysql> SELECT count(*) FROM items WHERE id IN (SELECT iid FROM items_links);
+----------+
| count(*) |
+----------+
|        4 |
+----------+
1 row in set (0.08 sec)
```

MySQL 服务器检查了 10 行然后返回结果是 4。这有多快？为了回答这个问题，统计每个表中的行数：

```
mysql> SELECT count(*) FROM items;
+----------+
| count(*) |
+----------+
|       10 |
+----------+
```

```
1 row in set (0.11 sec)

mysql> SELECT count(*) FROM items_links;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.00 sec)
```

itmes 表中有 10 行记录，每行都有一个唯一的 ID。items_links 表中有 6 行记录，没有唯一的 ID(iid)。对于当前的数据规模来说，这个设计看起来可以，不过与此同时，这也透露出一个潜在的问题。目前，links 数小于 items 数，并且数目的差异不是很大，但是如果数目差距巨大，那就更加值得注意了。

为了验证这个猜想，同时也为了举例说明查询调优的方法，我将向 items 表中插入一些数据。id 字段定义为 INT NOT NULL AUTO_INCREMENT PRIMARY KEY，这样可以确保对新插入的行不会存在关联（link）。这样，我就可以模仿现实中用户想从一个大表中查询少量 link 的场景（在该场景中是 6 个）。下面的语句展现了一个快速构造数据的小技巧，即重复地从表中选择所有行，然后再插入更多行：

```
mysql> INSERT INTO items( short_description , description,
example, explanation, additional) SELECT  short_description , description,
example, explanation, additional FROM items;
Query OK, 10 rows affected (0.17 sec)
Records: 10  Duplicates: 0  Warnings: 0
<Repeat this query few times>
mysql> INSERT INTO items( short_description , description,
example, explanation, additional) SELECT  short_description , description,
example, explanation, additional FROM items;
Query OK, 2560 rows affected (3.77 sec)
Records: 2560  Duplicates: 0  Warnings: 0
```

现在，看一下查询的执行计划是否有所变化：

```
mysql> EXPLAIN EXTENDED SELECT count(*) FROM items WHERE id IN
    -> (SELECT iid FROM items_links)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: items
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 5136
     filtered: 100.00
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: items_links
```

```
        type: index_subquery
possible_keys: iid,iid_2
         key: iid
     key_len: 5
         ref: func
        rows: 1
    filtered: 100.00
       Extra: Using index; Using where
2 rows in set, 1 warning (0.09 sec)
```

查询的执行计划并没有变化——这次为了 6 个 link 检查了 5136 行！有没有什么方式可以重写一下这个查询，使其运行得更快一些呢？

子查询的类型是 index_subquery。这意味着优化器使用索引查询函数完全替代了子查询。SHOW WARNINGS 的输出展示了查询是如何重写的：

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select count(0) AS `count(*)` from `collaborate2011`.`items` where
<in_optimizer>(`collaborate2011`.`items`.`id`,<exists>
(<index_lookup>(<cache>(`collaborate2011`.`items`.`id`) in
items_links on iid where (<cache>(`collaborate2011`.`items`.`id`) =
`collaborate2011`.`items_links`.`iid`))))
1 row in set (0.00 sec)
```

输出信息看起来令人生畏，不过至少在这里可以看到一些连接。如果我们重写查询，让在其上执行连接的列更加明显，那又会怎么样？我们也将重写子查询，使之变成显式 JOIN；在当前版本的 MySQL 中，这个方法能够显著地提高性能：

```
mysql> \W
Show warnings enabled.
mysql> EXPLAIN EXTENDED SELECT count(*) FROM items JOIN
items_links ON (items.id=items_links.iid)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: items_links
         type: index
possible_keys: iid,iid_2
          key: iid_2
      key_len: 5
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: items
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
```

```
          ref: collaborate2011.items_links.iid
         rows: 1
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.05 sec)

Note (Code 1003): select count(0) AS `count(*)` from `collaborate2011`.`items`
join `collaborate2011`.`items_links` where (`collaborate2011`.`items`.`id` =
`collaborate2011`.`items_links`.`iid`)
```

结果看起来挺让人振奋的，因为它没有搜索 items 表中的所有行。不过，这个查询结果正确吗？

```
mysql> SELECT count(*) FROM items JOIN items_links ON
(items.id=items_links.iid);
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.10 sec)
```

我们得到了 6 行，而不是 4 行。这是因为我们要求返回所有匹配的行，这里有相同的 link 被匹配了两次。可以通过添加 DISTINCT 关键字来修复这个问题：

```
mysql> SELECT count(distinct items.id) FROM items JOIN items_links ON
(items.id=items_links.iid);
+--------------------------+
| count(distinct items.id) |
+--------------------------+
|                        4 |
+--------------------------+
1 row in set (0.12 sec)
```

提示

可以通过查询重写技巧来确认是否需要添加 DISTINCT 关键字。只要将 count(*)替换成 items.id，就可以看到重复的值。

加上 DISTINCT 后，该查询一样高效吗？我们再次通过 EXPLAIN 来验证一下：

```
mysql> EXPLAIN EXTENDED SELECT count(distinct items.id) FROM items
JOIN items_links ON (items.id=items_links.iid)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: items_links
         type: index
possible_keys: iid,iid_2
          key: iid_2
      key_len: 5
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using index
```

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: items
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: collaborate2011.items_links.iid
         rows: 1
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): select count(distinct `collaborate2011`.`items`.`id`) AS
`count(distinct items.id)` from `collaborate2011`.`items` join
`collaborate2011`.`items_links` where (`collaborate2011`.`items`.`id` =
`collaborate2011`.`items_links`.`iid`)
```

它仍然检查了 6 行记录。因此，我们可以认为对于这个特定的数据集，该查询得到了优化。本章后面将会解释为何数据结构及其容量会有影响。

在该示例中，数据集是小规模的，所以即使在我的笔记本电脑上，我也无法让其真正执行得特别缓慢。不过，原始的和优化过的查询的执行时间的确有很大不同。下面是原始查询的时间：

```
mysql> SELECT count(*) FROM items  WHERE id IN (SELECT iid FROM
items_links );
+----------+
| count(*) |
+----------+
|        4 |
+----------+
1 row in set (0.21 sec)
```

下面是优化过的查询的时间：

```
mysql> SELECT count(distinct items.id) FROM items JOIN items_links
ON (items.id=items_links.iid);
+--------------------------+
| count(distinct items.id) |
+--------------------------+
|                        4 |
+--------------------------+
1 row in set (0.10 sec)
```

对于如此小的数据集，查询的时间仍然降低了一半！在测试中，虽然仅提高了 0.11 秒，不过如果对于上百万行，那么效率提升效果就会更好。

- 你刚刚学习了一个基本的使用 EXPLAIN 命令的查询调试技巧：阅读当前查询的信息并与你预期的信息进行比较。该过程可以用来调优从最简单到最复杂的任何查询。

1.6.2 表调优和索引

上一节介绍了调优查询的过程。在全部示例中，EXPLAIN 的输出都包括索引信息。那么，如果表没有索引会如何？或者如果没有使用到索引呢？你该如何选择在什么时候、什么地方、添加哪种索引呢？

当结果有限的时候，MySQL 服务器会使用索引。因此，在与 WHERE、JOIN 和 GROUP BY 语句相关的列上添加索引可以加速查询。在与 ORDER BY 语句相关的列上添加索引也会有效果，因为它将使服务器更高效地排序。

在掌握这些规则的前提下，添加索引就成为了一个很简单的工作。考察之前示例中的表，但没有任何索引：

```
mysql> CREATE TEMPORARY TABLE items SELECT * FROM items;
Query OK, 5120 rows affected (6.97 sec)
Records: 5120  Duplicates: 0  Warnings: 0

mysql> CREATE TEMPORARY TABLE items_links SELECT * FROM items_links;
Query OK, 6 rows affected (0.36 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SHOW CREATE TABLE items;
+-------+-----------------------------------------------------------------------+
| Table | Create Table                                                          |
+-------+-----------------------------------------------------------------------+
| items | CREATE TEMPORARY TABLE `items` (
  `id` int(11) NOT NULL DEFAULT '0',
  `short_description` varchar(255) DEFAULT NULL,
  `description` text,
  `example` text,
  `explanation` text,
  `additional` text
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-------+-----------------------------------------------------------------------+
1 row in set (0.10 sec)

mysql> SHOW CREATE TABLE items_links;
+-------------+-----------------------------------------------------------------+
| Table       | Create Table                                                    |
+-------------+-----------------------------------------------------------------+
| items_links | CREATE TEMPORARY TABLE `items_links` (
  `iid` int(11) DEFAULT NULL,
  `linkid` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-------------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

如你所见，没有指定任何索引。我们在这些表上试验一个没有优化过的查询，然后再优化它：

```
mysql> EXPLAIN EXTENDED SELECT count(distinct items.id) FROM items JOIN
items_links ON (items.id=items_links.iid)\G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: items_links
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: items
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5137
     filtered: 100.00
        Extra: Using where; Using join buffer
2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): select count(distinct `collaborate2011`.`items`.`id`) AS
`count(distinct items.id)` from `collaborate2011`.`items` join
`collaborate2011`.`items_links` where (`collaborate2011`.`items`.`id` =
`collaborate2011`.`items_links`.`iid`)
```

类型变成了 ALL，这是最耗时的类型，因为这表示会读取所有行。该查询这次检查了 6 * 5137 = 30 822 行。这甚至比之前示例中我们认为的慢查询还要糟。

来仔细检查一下这个查询：

```
SELECT count(distinct items.id)...
```

这个查询返回结果集中唯一非空值的数目。应该在 items.id 列上添加索引，以使该查询使用索引。

该查询的另一部分是：

```
...FROM items JOIN items_links ON (items.id=items_links.iid)
```

这里有 items 表中 id 字段和 items_links 表中 iid 字段的连接。因此，应该在这两列上添加索引。

```
mysql> ALTER TABLE items ADD INDEX(id);
Query OK, 5120 rows affected (4.78 sec)
Records: 5120  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE items_links ADD INDEX(iid);
Query OK, 6 rows affected (0.04 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

现在看一下添加索引对查询计划的影响：

```
mysql> EXPLAIN EXTENDED SELECT count(distinct items.id) FROM items
JOIN items_links ON (items.id=items_links.iid)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: items_links
         type: index
possible_keys: iid
          key: iid
      key_len: 5
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: items
         type: ref
possible_keys: id
          key: id
      key_len: 4
          ref: collaborate2011.items_links.iid
         rows: 1
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): select count(distinct `collaborate2011`.`items`.`id`) AS
`count(distinct items.id)` from `collaborate2011`.`items` join
`collaborate2011`.`items_links` where (`collaborate2011`.`items`.`id` =
`collaborate2011`.`items_links`.`iid`)
```

这看起来比之前好了很多，只有一点不好： items 表这次的类型是 ref，比之前的 eq_ref 要差。这是因为我们添加的是一个简单索引，而原始表在该列已经有唯一索引了。我们也可以简单地修改该临时表，因为 ID 是唯一的并且也应该如此：

```
mysql> EXPLAIN EXTENDED SELECT count(distinct items.id) FROM items
JOIN items_links ON (items.id=items_links.iid)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: items_links
         type: index
possible_keys: iid
          key: iid
      key_len: 5
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
```

```
          table: items
           type: eq_ref
  possible_keys: id_2,id
            key: id_2
        key_len: 4
            ref: collaborate2011.items_links.iid
           rows: 1
       filtered: 100.00
          Extra: Using index
2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): select count(distinct `collaborate2011`.`items`.`id`) AS
`count(distinct items.id)` from `collaborate2011`.`items` join
`collaborate2011`.`items_links` where (`collaborate2011`.`items`.`id` =
`collaborate2011`.`items_links`.`iid`)
```

现在，当已经使用了执行更快的 eq_ref 类型的时候，可以删除 items.id 字段上多余的索引。这在你关心数据修改的查询速度的时候尤为重要，因为每次更新索引都会消耗时间。下一节会讨论何时该停止优化。

你刚刚学习了索引是如何影响查询执行的以及何时应该添加索引。

选择你自己的执行计划

索引实际上也有减慢查询的时候。在这种情况下，应该删除索引或者使用会忽略索引（IGNORE INDEX）的语句（如果其他的查询还需要用到该索引）。也可以使用强制索引（FORCE INDEX）使优化器知道你想要使用的索引。这些语句对于查询调优也非常有用，比如当你想要了解特定索引对性能会有怎样影响的时候。只需要通过 EXPLAIN 命令执行语句，然后分析输出。

尽管使用 IGNORE INDEX 和 FORCE INDEX 可能听起来不错，但是你应该避免在生产环境中使用，除非你已经准备好在今后的每个升级版本中都逐一检查使用了该语句查询。

因为优化器总是试图选择最佳的执行计划，随着版本的升级，可能对于同一个 JOIN 会使用不同的执行计划，所以这种检查是必要的。当你没有强制使用或者忽略索引的时候，优化器会按照它认为最佳的计划执行。但是，如果你明确指定优化器在多表 JOIN 的某个表中应该如何使用索引，那么这个规则可能会造成其他影响，并且这个最终的执行计划在新版本中可能会比之前要差。

在对单一表的查询中使用 IGNORE INDEX 和 FORCE INDEX 相对安全。对所有其他的情况，必须在更新后检查确保查询的执行计划没有改变。

在产品中使用 IGNORE INDEX 和 FORCE INDEX 的另一个问题是对于指定表的最佳执行计划依赖于其存储的数据。一般的步骤是，优化器检查表的统计数据然后依此调整计划，当然在你使用了 IGNORE INDEX 和 FORCE INDEX 的时候不会这样做。如果你使用这些语句，你就必须定期检查在应用程序的生命周期它们是否还有效。

1.6.3 何时停止调优

前面讨论了简单查询。即使是简单查询，我们仍找到了优化的方向，有时经过一步一步的调优我们获得了更好的结果。当你处理包含很多 JOIN 条件，或者 WHERE 子句和 GROUP BY 字句中包含很多字段的复杂查询时，你就会拥有更多的选择。可以认为你总会找到使性能更好的方法，并且这种优化永无止境。因此，现在问题是，什么时候可以认为查询优化合理并可以停止进一步优化。

深入了解性能优化的技术自然可以帮助你选择合适的解决方案。不过，哪怕你自认不是专家，我们也有一些基本原则可以帮你决定停止优化。

首先，你应该了解查询都做了什么。例如，下面的查询：

```
SELECT * FROM contacts
```

始终会返回表中的所有列，该语句没有什么可优化的空间。

不过，即使你查询所有列，添加 JOIN 语句也会使情况改变：

```
SELECT * FROM child JOIN parent ON (child.pid=parent.id)
```

这会产生优化效果，因为 ON 条件限制了结果集。当然，同样的分析也适用于 WHERE 和 GROUP BY 条件。

其次，你应该通过 EXPLAIN 输出查看连接类型。尽管你想要获得可能的最佳的 JOIN 类型，但是你应时刻意识到你的数据的约束。例如，非唯一的行永远不会产生 eq_ref 或者更好的类型。

当你优化查询的时候，你的数据是非常重要的。对于同样的执行计划，不同的数据可能会产生完全不同的结果。最简单的示例就是比较表中只有一行记录和表中超过 50% 的行都有相同值的结果。在这样情况下，使用索引会降低性能而不是提升性能。

- 另一个规则：不要只依赖于 EXPLAIN 的输出，要衡量实际的查询执行时间。

你需要牢记的另一件事情是索引在修改表时的影响。尽管索引通常会提高 SELECT 查询的性能，但是它会略微降低修改数据的查询的性能，尤其是 INSERT 语句。因此，有些时候为了加快插入的速度，允许 SELECT 查询略慢是明智的。要时刻牢记考察你整个应用程序的性能，而不仅仅是某一个查询的性能。

1.6.4 配置选项的影响

假如你已经对查询进行了完全的优化，找不到进一步优化的方法，但是它仍然很慢，那么还有没有办法可以提高它的性能呢？有的。有很多服务器选项可以让你调节对查询有影响的因素，比如内存中临时表的大小、排序缓冲区等。有些针对特定存储引擎（如 InnoDB）的选项，也会对查询优化很有用。

第 3 章将详细介绍这些选项。这里仅对如何使用它们进行性能优化做一个概述。

调整服务器的配置从某种程度来说是一个影响全局的行为，因为每个修改都可能对该服务器上的每个查询造成影响（对于指定引擎的选项，会影响每个使用该引擎的表上的查询）。不过有些选项是针对特定类型的优化的，如果你的请求没有满足条件，它将没有任何作用。

第一个需要检查的选项是缓冲区大小（buffer size）。每个缓冲区都有其存在的特定原因。一般的规律是大缓冲区意味着高性能——不过仅当请求可以针对该缓冲区扮演的特定角色使用大容量缓存的时候。

当然，增加缓冲区大小是有代价的。下面是一些大缓冲区可能带来的影响。我不是想要阻止你使用大缓冲区，因为在合理的环境下它是提高性能非常有效的手段。你仅需要牢记下面的要素然后合理地调整大小。

交换区（Swapping）

大容量缓冲区可能会导致使用到操作系统级别的交换区从而造成性能缓慢，这取决于系统内存大小。通常情况下，MySQL 服务器在它所需的所有内存都来自物理内存的时候运行最快。当它使用到交换区的时候，性能显著下降。

当为缓冲区分配的内存大小超过服务器的物理内存大小的时候就会使用到交换区。请注意，有一些缓冲区是针对每个用户线程的。要确定这些缓冲区究竟需要多少内存，可以用公式最大连接数 * 缓冲区大小（*max_connections* * *buffer_size*）来计算。计算出所有缓冲区的内存和，并确保和小于 mysqld 服务器可以使用的内存大小。这个计算的值不是决定性的，因为 mysqld 实际上可以分配多于你明确指定大小的内存。

启动时间

mysqld 需要分配的内存越多，其启动时间就越长。

过期数据

我们还会有伸缩性问题，大部分时候是来自线程间的缓存共享。在这些场景中，扩充缓冲区做缓存会产生内存碎片。你通常会在服务器运行数小时后发现内存碎片问题，该问题发生在旧的数据需要从缓冲区中移除以给新数据腾出空间的时候。这会导致高速运转的服务器突然变慢。第 3 章会给出这样的示例。

讨论完缓冲区，第 3 章讨论其他选项。届时，我们不仅关注性能优化选项（如优化器选项），还会关注一些控制高可用性的选项。事务运行得越安全，就需要更多的检查和更慢地执行性能。不过，要注意这些选项；只有在你可以为了性能牺牲安全的时候才调优它们。

当你调优分配的时候，把性能作为整体来考虑尤为重要，因为每个选项都会影响整个

服务器。例如，如果你没有使用特定引擎，针对该存储引擎的选项调优不会有任何作用。这是显而易见的，不过我确实见过很多安装环境下有大量关于 MyISAM 引擎的选项，然而却使用的 InnoDB 存储引擎，或者相反的情况。如果你用一些通用配置作为模板，这些注意点就尤为重要。

1.6.5 修改数据的查询

我们讨论了影响 SELECT 性能的因素，在本节我们开始优化修改数据的查询。UPDATE 和 DELETE 查询可以使用与 SELECT 语句一样的条件去限制受影响的行数。因此，可以使用相同的优化规则。

我们在 1.3 节中学习了如何把 UPDATE 和 DELETE 查询转换成 SELECT 查询，然后使用 EXPLAIN 进行调试。可以在 5.6.3 以下版本的系统上使用该技巧解决性能问题，从 5.6.3 版本开始增加了 EXPLAIN 对 INSERT、UPDATE 和 DELETE 查询的支持，不过，请牢记 UPDATE 和 DELETE 查询有时候与相应的 SELECT 查询的执行方式略有不同。

通过在查询计划前后查询 Handler_%的状态可以检查是否使用了索引：

```
mysql> SHOW STATUS LIKE 'Handler_%'; ❶
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 0     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 0     |
| Handler_read_key           | 0     |
| Handler_read_last          | 0     |
| Handler_read_next          | 0     |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 19    |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 17    |
+----------------------------+-------+
16 rows in set (0.00 sec)
```

❶ 这里使用了 SHOW STATUS 命令，这是 SHOW SEESION STATUS 的同义命令，作用是查看当前会话的变量状态。

提示

在测试前使用 FLUSH STATUS 查询可以方便地重置这些变量。

我们将继续介绍之前列表中的特定变量。你需要注意的是，因为这些是累积的值，所

以它们会随着你的每次查询增长。现在我们开始优化 1.3 节中的查询示例，使其更新可以为空的列：

```
mysql> UPDATE items SET description = 'no description', additional
= 'no additional comments' WHERE description IS NULL;
Query OK, 0 rows affected (6.95 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

这条语句没有修改任何行，因为数据在之前损坏了：现在每个字段中的值是 0 而不是 NULL。但是这个请求执行非常缓慢。我们来看一下处理程序变量：

```
mysql> show status like 'Handler_%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 1     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 1     |
| Handler_read_key           | 2     |
| Handler_read_last          | 0     |
| Handler_read_next          | 0     |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 5140  |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 17    |
+----------------------------+-------+
16 rows in set (0.01 sec)
```

可以看到 Handler_read_rnd_next 的值非常高，该值代表从 datafile 中读取下一个值的频繁程度。过高的值一般代表使用了全表扫描，这对性能是有影响的。Handler_read_key 也是一个相关的变量，表示读取索引的请求数目。正常情况下该值相对于 Handler_read_rnd_next 来说不应该这么低，因为这意味着大部分行的读取都没有使用的索引。此外，Handler_commit 和 Handler_read_first 的值也增长缓慢。它们分别代表事务提交的次数和读索引中第一项的次数。最后，Handler_read_first 的值是 1，表明我们请求服务器读取索引中第一条记录，这可以当作全索引扫描的标志。

希望对这些 Handler_%状态变量的简介可以告诉你如何利用它们去检查查询是怎样执行的。对于该查询是否有提升空间这个问题将作为作业留给读者自己去解答。

我仅将对 INSERT 查询做些说明。它们没有条件去约束受影响的行数，因此表中的索引只会降低插入效率，因为每次插入都需要更新索引。插入的性能需要通过服务器选项调优。特别地，影响 InnoDB 存储引擎的选项会很有作用。

一种加速插入的方式是把多个插入合并成一条语句，这也叫做“批量插入”（bulk insert）：

```
insert into t1 (f1, f2, f3, ...) values (v1, v2, v3, ...), (v1, v2, v3, ...), ...
```

不过，请注意插入会阻塞行甚至是整张表，因此其他查询会在插入的过程中被拒绝访问。我将给出一个通用规则来结束本节内容：

- 在优化任何单个查询的时候，请时刻注意整个应用程序的性能。

1.6.6 没有高招

我们刚刚学习了如何优化服务器选项才能显著提升性能。在本章我们也学习了如何优化特定查询以提升其运行速度。优化查询和优化服务器一般是解决性能问题的两种选择。那么，有没有通用的规则告诉我们该从哪个方向开始优化呢？

我认为恐怕没有。优化服务器选项看起来特别有效以至于许多人认为合理地的改变选项将会使 mysqld 运行得如火箭般高效。如果你也是那么想的，我不得不让你失望了：不好的查询写法仍会耗尽服务器资源。并且你可能在重启服务器后仅仅感受到几个小时的服务器高性能，然后它就又变慢了，因为每个查询都需要消耗很多的资源并且你的缓存将会充满。有时候，服务器会被数以百万计的查询淹没，需要更多的资源。

然后，优化每个查询也不是一个好选择。有些请求很少调用，所以没有必要在这些查询上浪费人力。还有的查询可能查询表中的所有行，这些查询就没有必要去尝试优化了。

我一般推荐“混合”的优化模式。先优化服务器选项，特别注意你使用的存储引擎相关的选项，然后优化查询。当优化完重要的查询后，回头再检查服务器选项，考虑是否有进一步的优化空间，然后再继续优化剩下的查询，反复如此，直到你对性能满意。

你也可以从自己的应用程序中最慢的查询开始或者找到那些可以通过简单的优化获得显著提升的查询，然后优化服务器选项。参考之前展示的状态变量，第 6 章将详细介绍它们。

最后同样重要的是：在性能优化中参考大量的信息以形成你自己的策略。

1.7 当服务器无响应的时候

有时候，MySQL 客户端会收到严重的错误消息“在请求中丢失与服务器的连接”或者“服务器已停止”。尽管我希望你永远不会遇到这个错误，但是有所准备总是有好处的。由 MySQL 安装本身引起的这个问题主要有两个原因：服务器问题（最有可能是崩溃）或者滥用连接选项（通常是超时选项或者 max_allowed_packet）。

第 3 章将讨论连接相关的配置。第 4 章会讨论硬件问题和第三方软件相关的问题。这里简短地介绍一下如果遇到服务器崩溃该做些什么。

首先，确定你的服务器是否真的崩溃。你可以借助进程状态监控器进行确认。如果你在服务器崩溃后运行了 mysqld_safe 或者其他守护进程重启服务器，错误日志将会包含表明服务器已经重启的消息。当 mysqld 启动时，它始终会在错误日志文件中输出类似如下的信息：

```
110716 14:01:44 [Note] /apps/mysql-5.1/libexec/mysqld: ready for connections.
Version: '5.1.59-debug'  socket: '/tmp/mysql51.sock'  port: 3351  Source distribution
```

因此，如果你找到了类似消息，那么服务器已经重启了。如果没有任何消息，并且服务器已经启动并在运行，那么丢失连接的问题最有可能是因为滥用连接选项导致的，这将在第 3 章进行讨论。

提示

如果你记得MySQL服务器原来是何时启动的，你可以使用状态变量uptime，该变量的值代表服务器已启动的时间，单位是秒：

```
mysql> SHOW GLOBAL STATUS LIKE 'uptime';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Uptime        | 10447 |
+---------------+-------+
1 row in set (0.00 sec)
```

该信息也可以帮你检查 mysqld 是否是因为操作系统的重启而失败的。仅需要比较该变量的值和操作系统的启动时间即可。

依赖于错误日志文件是因为我工作经历的原因，比如当客户在服务器崩溃数小时后发现问题，甚至有时候是在 mysqld 计划重启后才发现问题。

如果你确认服务器已经重启了，你应该再次检查错误日志去搜索崩溃本身的信息。通常，你会从错误日志中获取足够的崩溃信息，从而避免发生同样的情况。第 6 章将讨论如何调查你可能会遇到的少数困难情况。现在，我们再次回到错误日志文件，来看一下在服务器崩溃情况下的典型内容示例。这里我将摘录出大量信息：

```
Version: '5.1.39' socket: '/tmp/mysql_sandbox5139.sock' port: 5139
MySQL Community Server (GPL)
091002 14:56:54 - mysqld got signal 11 ;
This could be because you hit a bug. It is also possible that this binary
or one of the libraries it was linked against is corrupt, improperly built,
or misconfigured. This error can also be caused by malfunctioning hardware.
We will try our best to scrape up some info that will hopefully help diagnose
the problem, but since we have already crashed, something is definitely wrong
and this may fail.
key_buffer_size=8384512
read_buffer_size=131072
max_used_connections=1
max_threads=151
threads_connected=1

It is possible that mysqld could use up to
```

```
key_buffer_size + (read_buffer_size + sort_buffer_size)*max_threads = 338301 K
bytes of memory
Hope that's ok; if not, decrease some variables in the equation.
thd: 0x69e1b00
Attempting backtrace. You can use the following information to find out
where mysqld died. If you see no messages after this, something went
terribly wrong...
stack_bottom = 0x450890f0 thread_stack 0x40000
/users/ssmirnova/blade12/5.1.39/bin/mysqld(my_print_stacktrace+0x2e)[0x8ac81e]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(handle_segfault+0x322)[0x5df502]
/lib64/libpthread.so.0[0x3429e0dd40]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN6String4copyERKS_+0x16)[0x5d9876]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Item_cache_str5storeEP4Item+0xc9)
[0x52ddd9]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN26select_singlerow_subselect9send_
dataER4ListI4ItemE+0x45)
[0x5ca145]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x6386d1]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x64236a]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN4JOIN4execEv+0x949)[0x658869]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN30subselect_single_select_
engine4execEv+0x36c)[0x596f3c]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Item_subselect4execEv+0x26)[0x595d96]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN24Item_singlerow_subselect8val_
realEv+0xd)[0x595fbd]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Arg_comparator18compare_real_
fixedEv+0x39)[0x561b89]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN12Item_func_ne7val_intEv+0x23)[0x568fb3]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN4JOIN8optimizeEv+0x12ef)[0x65208f]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z12mysql_selectP3THDPPP4ItemP10TABLE_
LIST jR4ListIS1_ES2_jP8
st_orderSB_S2_SB_yP13select_resultP18st_select_lex_unitP13st_select_lex+0xa0)
[0x654850]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z13handle_selectP3THDP6st_lexP13select_
resultm+0x16c)
[0x65a1cc]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x5ecbda]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z21mysql_execute_commandP3THD+0x602)
[0x5efdd2]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z11mysql_parseP3THDPKcjPS2_+0x357)
[0x5f52f7]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z16dispatch_command19enum_server_
commandP3THDPcj+0xe93)
[0x5f6193]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z10do_commandP3THD+0xe6)[0x5f6a56]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(handle_one_connection+0x246)[0x5e93f6]
/lib64/libpthread.so.0[0x3429e061b5]
/lib64/libc.so.6(clone+0x6d)[0x34292cd39d]`)
Trying to get some variables.
Some pointers may be invalid and cause the dump to abort...
thd->query at 0x6a39e60 = select 1 from `t1` where `c0` <>
(select geometrycollectionfromwkb(`c3`) from `t1`)
thd->thread_id=2
thd->killed=NOT_KILLED
The manual page at http://dev.mysql.com/doc/mysql/en/crashing.html contains
information that should help you find out what
```

表明崩溃原因的关键行是：

```
091002 14:56:54 - mysqld got signal 11 ;
```

这意味着 MySQL 服务器在向操作系统申请资源（例如，访问文件或者内存）后终止了，得到了错误代码 11。在大多数操作系统里，这个信号代表分段错误（segmentation fault）。你可以在你的操作系统的用户手册中获取更详细的信息。对于 UNIX 和 Linux 系统可以执行 man 命令。在 Windows 操作系统里，相似的情况通常会产生类似“mysqld got exception 0xc0000005”的日志消息。查找 Windows 的用户手册可以获取该异常代码的含义。

下面是从某个线程中导致服务器崩溃的请求的相关日志中提取的摘要信息：

```
Trying to get some variables.
Some pointers may be invalid and cause the dump to abort...
thd->query at 0x6a39e60 = SELECT 1 FROM `t1` WHERE `c0` <>
(SELECT geometrycollectionfromwkb(`c3`) FROM `t1`)
thd->thread_id=2
thd->killed=NOT_KILLED
```

为了进一步分析，重新执行查询来检查其是否是崩溃的原因：

```
mysql> SELECT 1 FROM `t1` WHERE `c0` <> (SELECT
geometrycollectionfromwkb(`c3`) FROM `t1`);
ERROR 2013 (HY000): Lost connection to MySQL server during query
```

提示

当我推荐你重现错误的时候，我假定你是在开发服务器而不是生产服务器中进行的。6.3 节将讨论如何安全地在特定环境下进行问题调优。请不要尝试重试示例中的语句，这是一个已知的 bug#47780，已经在当前版本修复了。从 5.0.88、5.1.41、5.5.0 和 6.0.14 版本开始，该 bug 已经修复。

当目前为止，你已经找到并确认崩溃的原因，然而你还需要重写请求，从而避免下次服务器再次崩溃。现在，我们可以从日志的堆栈信息中获得帮助：

```
Attempting backtrace. You can use the following information to find out
where mysqld died. If you see no messages after this, something went
terribly wrong...
stack_bottom = 0x450890f0 thread_stack 0x40000
/users/ssmirnova/blade12/5.1.39/bin/mysqld(my_print_stacktrace+0x2e)[0x8ac81e]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(handle_segfault+0x322)[0x5df502]
/lib64/libpthread.so.0[0x3429e0dd40]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN6String4copyERKS_+0x16)[0x5d9876]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Item_cache_str5storeEP4Item+0xc9)
[0x52ddd9]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN26select_singlerow_subselect9
send_dataER4ListI4ItemE+0x45)
[0x5ca145]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x6386d1]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x64236a]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN4JOIN4execEv+0x949)[0x658869]
```

```
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN3Osubselect_single_select_engine4execEv
+0x36c)[0x596f3c]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Item_subselect4execEv+0x26)[0x595d96]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN24Item_singlerow_subselect8val_realEv
+0xd)[0x595fbd]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Arg_comparator18compare_real_fixedEv
+0x39)[0x561b89]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN12Item_func_ne7val_intEv+0x23)[0x568fb3]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN4JOIN8optimizeEv+0x12ef)[0x65208f]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z12mysql_selectP3THDPPP4ItemP10TABLE_
LISTjR4ListIS1_ES2_jP8
st_orderSB_S2_SB_yP13select_resultP18st_select_lex_unitP13st_select_lex+0xa0)
[0x654850]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z13handle_selectP3THDP6st_lexP13select_
resultm+0x16c)
[0x65a1cc]
/users/ssmirnova/blade12/5.1.39/bin/mysqld[0x5ecbda]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z21mysql_execute_commandP3THD+0x602)
[0x5efdd2]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z11mysql_parseP3THDPKcjPS2_+0x357)
[0x5f52f7]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z16dispatch_command19enum_server_
commandP3THDPcj+0xe93)
[0x5f6193]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_Z10do_commandP3THD+0xe6)[0x5f6a56]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(handle_one_connection+0x246)[0x5e93f6]
/lib64/libpthread.so.0[0x3429e061b5]
/lib64/libc.so.6(clone+0x6d)[0x34292cd39d]`)
```

与错误相关的行是调用 Item_subselect 和 Item_singlerow_subselect 的部分：

```
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN14Item_subselect4execEv+0x26)[0x595d96]
/users/ssmirnova/blade12/5.1.39/bin/mysqld(_ZN24Item_singlerow_subselect8val_realEv
+0xd)[0x595fbd]
```

我是如何确定这是罪魁祸首的呢？在这个示例中，我发现在之前排错过程中的调用。不过经验告诉我，最好还是从头开始排查问题。最前面的几个函数通常是操作系统调用的，这些函数可能会和问题相关，不过在当前环境下没有任何帮助，因为对它们你什么都做不了，接下来是对 MySQL 库的调用。自上而下地检查这些函数，以便找到哪些是与你相关的。例如，对于 String4copy 或 Item_cache_str5store 函数你没什么可做的，但是你可以重写子查询，因此我们从这里开始。

这里，哪怕不去看 mysqld 的源码，我们也可以找到崩溃的原因。推测子查询可能是问题的所在，这是一个不错的假设，因为子查询可以很容易地转换成 JOIN。尝试重写查询然后测试它：

```
mysql> SELECT 1 FROM `t1` WHERE `c0` <> geometrycollectionfromwkb(`c3`);
Empty set (0.00 sec)
```

新查询并没有崩溃，因此你需要做的就是把应用程序中的查询改成等价的形式。

- 你刚刚学到了一些 MySQL 问题排查中的重要事情：当遇到未知的错误时首先应该做的是检查错误日志文件。始终要打开日志。

这里我想补充一点关于 bug 的内容。当你遇到崩溃并确定了原因的时候，应检查一下 MySQL 的 bug 数据库，看看有无类似的问题。如果你找到可能与你相关的 bug，确认它是否修复了。如果已经修复了，那么把你的服务器升级到 bug 已修复的版本（或者更新的版本）。这样做可以节约你的时间，因为你不再需要修改有问题的语句了。

如果你没找到与你相关的 bug，尝试下载最新的 MySQL 版本，然后再执行查询。如果 bug 再次出现，请把它提交给我们。使用最新的稳定版本是非常重要的，因为它包含当前修复的所有 bug，许多老的问题在这里不会重现。第 6 章中会讨论如何在沙盒环境下安全地测试崩溃的情形。

不仅是特定的查询会引起崩溃，而且服务器运行的环境也有可能引起崩溃。最常见的原因就是缺少可用的内存（RAM）。特别当用户分配超大缓冲区时最容易发生。正如我之前提到的那样，mysqld 始终需要比所有缓冲区容量总和略多的内存。通常情况下，错误日志文件包含可用内存的粗略估计。它看起来如下所示：

```
key_buffer_size=235929600
read_buffer_size=4190208
max_used_connections=17
max_connections=2048
threads_connected=13
It is possible that mysqld could use up to
key_buffer_size + (read_buffer_size + sort_buffer_size)*max_connections = 21193712 K
---
21193712K ~= 20G
```

这种估计不是精确的，不过仍然值得确认。上面展示的信息表明 mysqld 可以使用高达 20GB 的内存。虽然现在你可以很容易地获得强大的计算机，但是仍有必要确认你是否真的拥有 20GB 内存。

该环境中的另一个问题是，有其他的应用程序与 MySQL 服务器一起运行。在生产环境中，最好为 MySQL 指定一台专门的服务器，因为其他应用程序可能会占用你希望 MySQL 使用的资源。第 4 章将讨论如何调试其他应用程序对 mysqld 的影响。

1.8 特定于存储引擎的问题及解决方案

实际上，本书讨论的任何问题在你使用的存储引擎下都可能会有细微的差别。这种情况将在本书中一直存在。本节展示一些不依赖于其他问题的存储引擎自身的特性。我们将使用针对 MyISAM 或 InnoDB 存储引擎的工具来解决一些问题，因为这两种引擎是最受欢迎和最频繁使用的存储引擎。如果你使用第三方的存储引擎，那么可以查阅它的用户手册来获取有用的工具。

与存储引擎相关的错误要么会反馈到客户端，要么会记录在错误日志文件中。一般情况下，存储引擎的名字也会出现的错误消息中。偶尔，你可能会获得一个用 perror 工

具也查不到的未知错误码。这一般是表明问题来自某存储引擎的信号。

一个常见的存储引擎问题是数据损坏。这不一定总是存储引擎的错误，也可能是由于磁盘损毁、系统崩溃或者 MySQL 服务器崩溃等外部原因。例如，如果有人使用 kill –9 终止服务器的进程，那么就很有可能招致数据损坏。这里将讨论如果 MyISAM 和 InnoDB 引擎发生数据损坏该怎么做。不会讨论如何修复第三方存储引擎的崩溃；读者可以查询该存储引擎的文档以获得相关指导。作为针对一般情况的建议，可以尝试使用 CHECK TABLE 命令，很多存储引擎都支持该工具（MyISQM 引擎的 CHECK TABLE 工具将在 1.8.1 节中详细介绍）。

数据损坏是一个很难诊断的问题，因为往往直到 MyISQM 服务器访问损坏的表时用户才会发现问题。并且，错误表现出来的特征也有可能产生误导。在最好的情况下，你会得到一条错误消息。然而，问题也可能表现为查询执行错误或者服务器停止。如果问题在某个特定表上突然出现，就始终要检查数据是否损坏。

提示

一旦你怀疑是数据损坏了，就需要修复损坏的表。要养成始终在修复之前备份表文件的习惯，这样你就可以在出错的时候恢复数据了。

1.8.1 MyISAM 损坏

MyISAM 引擎按照三个文件一组保存每张表：table_name.frm 文件包含表的结构（schema），table_name.MYD 文件存储数据，以及 table_name.MYI 文件保存索引。崩溃会损坏数据或者索引文件，或者二者都损坏了。在这种情况下，当访问表时，你就会获得类似“ERROR 126 (HY000): Incorrect key file for table './test/t1.MYI'; try to repair it”或“Table './test/t2' is marked as crashed and last(automatic?) repair failed”的错误消息。错误消息可能各种各样，不过可以检索关键字“repair”或“crashed”来判断是否是表损坏。

SQL 语句 CHECK TABLE 和 REPAIR TABLE 专门针对数据损坏问题。在操作系统 shell 里，也可以使用 myisamchk 工具进行同样的工作。使用 myisamchk 工具的一个好处就是，可以不必访问正在运行的 MySQL 服务器。例如，用户可以在崩溃后，在服务器再次启动前尝试修复表。

1. 通过 SQL 修复 MyISAM 表

CHECK TABLE 命令不加参数可以展示当前表的状态：

```
mysql> CHECK TABLE t2;
+---------+-------+----------+----------------------------------------------------------+
| Table   | Op    | Msg_type | Msg_text                                                 |
+---------+-------+----------+----------------------------------------------------------+
```

```
| test.t2 | check | warning  | Table is marked as crashed and last repair failed    |
| test.t2 | check | warning  | Size of indexfile is: 1806336      Should be: 495616 |
| test.t2 | check | error    | Record-count is not ok; is 780   Should be: 208      |
| test.t2 | check | warning  | Found 780 key parts. Should be: 208                  |
| test.t2 | check | error    | Corrupt                                              |
+---------+-------+----------+------------------------------------------------------+
5 rows in set (0.09 sec)
```

这是一个损坏的表的输出示例。解决问题的第一步应该是执行不带参数的 REPAIR TABLE 命令：

```
mysql> REPAIR TABLE t2;
+---------+--------+----------+---------------------------------------+
| Table   | Op     | Msg_type | Msg_text                              |
+---------+--------+----------+---------------------------------------+
| test.t2 | repair | warning  | Number of rows changed from 208 to 780 |
| test.t2 | repair | status   | OK                                    |
+---------+--------+----------+---------------------------------------+
2 rows in set (0.05 sec)
```

现在很幸运，表修复成功了。可以再次执行 CHECK TABLE 命令进行确认：

```
mysql> CHECK TABLE t2;
+---------+-------+----------+----------+
| Table   | Op    | Msg_type | Msg_text |
+---------+-------+----------+----------+
| test.t2 | check | status   | OK       |
+---------+-------+----------+----------+
1 row in set (0.02 sec)
```

如果单纯地执行 REPAIR TABLE 没有起到什么效果，那么可以选择另外两个选项。REPAIR TABLE EXTENDED 执行速度比 REPAIR TABLE 慢得多，但是可以修复 99% 的错误。作为最后的选择，可以执行 REPAIR TABLE USE_FRM 命令，该命令会不相信索引文件中的信息。它会删除索引并利用 table_name.frm 文件中的描述重建索引，并通过 table_name.MYD 文件填充键对应的值。

提示

为了达到同样的目的，还可以使用 mysqlcheck 工具。该工具通过向服务器发送 CHECK 和 REPAIR 命令进行工作。它还有非常好用的选项，如 --all-databases，该参数可以帮助用户高效地执行表的维护。

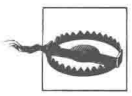
mysqlcheck 像其他客户端一样连接到 MySQL 服务器工作，因此它可以远程使用。

2. 使用 myisamchk 修复 MyISAM 表

所有这些步骤也可以通过使用 myisamchk 完成，该命令包含许多额外的表维护选项。这里不会一一介绍该工具的所有特性，而只是重点介绍与表修复相关的特性。

myisamchk 可以直接访问表文件，而无须启动 MySQL 服务器。在某些情况下，这是非

常有用的。同时，myisamchk需要对表文件保持独立的、排他的访问，因此用户也应该避免在MySQL服务器运行过程中使用该工具。



警告

如果必须在服务器运行期间使用myisamchk工具，那么先执行FLUSH TABLES和LOCK TABLE table_name WRITE语句，然后等待直到最后的查询返回命令提示符，接着再在并行会话中执行myisamchk。如果有除了myisamchk以外的进程在myisamchk运行期间访问表，就可能会产生更糟糕的损坏情况。

一条基本的恢复命令是：

```
$myisamchk --backup --recover t2
- recovering (with sort) MyISAM-table 't2'
Data records: 208
- Fixing index 1
- Fixing index 2
Data records: 780
```

其中，--backup选项通知myisamchk在尝试修复表之前先进行数据文件备份，--recover选项执行实际修复。如果这条命令还不够，可以使用--safe-recover选项。该选项会使用在早期的MySQL版本中就存在的恢复模式进行修复，并且会找到简单的--recover选项无法发现的问题。当然，还有更加严格的选项--extend-check。

也可以使用--sort-recover选项，该选项会使用排序来解析键，甚至在临时文件很大的时候。

在其他选项中，推荐你仔细研究非常有用的--description选项，该选项会输出表的描述信息。结合-v或者其等价的--verbose选项，将会输出额外的信息。可以指定两次甚至三次-v选项去获得更多的信息。

1.8.2 InnoDB数据损坏

InnoDB在共享的表空间中存储其数据和索引。如果服务器在创建表时是以选项--innodb_file_per_table选项启动的，那么它也会有自己的数据文件，不过表的定义仍然在共享的表空间中。理解表文件是如何存储的，将有助于高效地维护数据目录和备份。

InnoDB是带有事务的存储引擎，并且其内部机制会自动修复大部分数据损坏错误。它会在服务器启动时进行修复。下面的摘要信息是在MySQL企业级备份（MEB）中执行mysqlbackup --copy-back命令完成备份后从错误日志中摘录的，它展示了一种典型的恢复情况[1]：

1：MySQL企业级备份（MEB）以前也称作InnoDB热备份，是InnoDB表进行在线热备份和其他存储引擎的表进行在线备份的一个工具。第7章将讨论备份的方法。

```
InnoDB: The log file was created by ibbackup --apply-log at
InnoDB: ibbackup 110720 21:33:50
InnoDB: NOTE: the following crash recovery is part of a normal restore.
InnoDB: The log sequence number in ibdata files does not match
InnoDB: the log sequence number in the ib_logfiles!
110720 21:37:15  InnoDB: Database was not shut down normally!
InnoDB: Starting crash recovery.
InnoDB: Reading tablespace information from the .ibd files...
InnoDB: Restoring possible half-written data pages from the doublewrite
InnoDB: buffer...
InnoDB: Last MySQL binlog file position 0 98587529, file name ./blade12-bin.000002
110720 21:37:15 InnoDB Plugin 1.0.17 started; log sequence number 1940779532
110720 21:37:15 [Note] Event Scheduler: Loaded 0 events
110720 21:37:15 [Note] ./libexec/mysqld: ready for connections.
Version: '5.1.59-debug'  socket: '/tmp/mysql_ssmirnova.sock'  port: 33051
Source distribution
```

不过，有时候数据损坏得很严重并且 InnoDB 无法在没有用户交互的情况下完成修复。在这种情况下，有--innodb_force_recovery 启动选项。该选项可以设置为 0 ~ 6 的任意值（0 意味着不强制修复，1 是最低级别，6 是最高级别）。当修复成功的时候，可以在已修复的表上执行特定类型的请求，不过应该避免执行某些特定的命令。不能执行修改数据的操作，不过该选项仍允许特定的 SELECT 选择语句和 DROP 语句。例如，在级别 6 的情况下，仅可以执行形如 SELECT * FROM table_name 且不带 WHERE 条件、ORDER BY 排序或者其他语句的查询。

如果发生了损坏，可从 1 开始依次尝试每个级别的--innodb_force_recovery 选项，直到可以启动服务器并且可以访问有问题的表为止。你之前的检查应该已经发现了哪个表损坏了。使用 SELECT INTO OUTFILE 将表转储到文件中，然后使用 DROP 和 CREATE 命令重新创建表。最后，用配置--innodb_force_recovery=0 重新启动服务器，然后加载转储的数据。如果问题还存在，尝试找到其他损坏的表然后执行同样的过程直到服务器恢复正常。

当需要在--innodb_force_recovery 选项的值是正数的情况下开始修复数据库时，错误日志通常会有类似下面的明确提示消息：

```
InnoDB: We intentionally generate a memory trap.
InnoDB: Submit a detailed bug report to http://bugs.mysql.com.
InnoDB: If you get repeated assertion failures or crashes, even
InnoDB: immediately after the mysqld startup, there may be
InnoDB: corruption in the InnoDB tablespace. Please refer to
InnoDB: http://dev.mysql.com/doc/refman/5.1/en/forcing-recovery.html
InnoDB: about forcing recovery.
```

你也会从中发现关于自动修复失败和启动失败的信息。

提示

InnoDB 在写实际数据前会立即对数据、索引和日志页写校验和（checksum），并且在从磁盘读数据之后立即确认校验和。这可以避免大多数问题。通常，一旦遭遇 InnoDB 数据损坏，这就意味着磁盘或内存有问题。

1.9 许可问题

MySQL 有复杂的权限方案，这使得你可以精确地设置哪些用户和主机可以或不可以执行这个或那个操作。从 5.5 版本开始，MySQL 也有了可插拔式的身份验证模式。

尽管它有很多优势，但是这个方案很复杂。例如，让 user1@hostA、user2@hostA 和 user1@hostB 不同会很容易混淆它们的权限。当用户名相同而主机名变化的时候更是如此。

MySQL 允许在对象和连接层面设置访问规则。可以限制某个用户对于特定的表、列等的访问权限。

用户通常会遇到两类权限问题：

- 应该有权限连接到服务器的用户无法连接，或者没有权限的用户可以连接；
- 用户可以连接到服务器，但是无法使用他们本应该可以访问的对象，或者可以访问他们无权访问的对象。

在解决这些问题之前，应该确认你是否可以连接到服务器。

当你作为解决问题的用户成功连接到服务器之后（后面的章节将讨论无法连接的情况），执行以下查询：

```
SELECT USER(), CURRENT_USER()
```

USER()函数会返回当用户连接到服务器时使用的连接参数。这些参数通常为指定的用户名和运行客户端的主机名。CURRENT_USER()函数会返回从权限表中选择的与访问权限相关的用户名和主机名对。mysqld 用这些用户名和主机名对来检查数据库对象访问权限。通过比较这些函数的结果，可以找到 mysqld 使用的权限和预期不同的原因。一个典型的问题是对主机名使用通配符%：

```
root> GRANT ALL ON book.* TO sveta@'%';
Query OK, 0 rows affected (0.00 sec)

root> GRANT SELECT ON book.* TO sveta@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

如果此时我以 sveta 身份连接并尝试创建一个表，我就会获得下面的错误：

```
$mysql -usveta book
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 30
Server version: 5.1.52 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

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

```
owners.

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

mysql> CREATE TABLE t1(f1 INT);
ERROR 1142 (42000): CREATE command denied to user 'sveta'@'localhost' for table 't1'
```

该问题在于，MySQL 服务器认为 sveta 是 sveta@localhost，而不是通配符：

```
mysql> SELECT user(), current_user();
+----------------+----------------+
| user()         | current_user() |
+----------------+----------------+
| sveta@localhost | sveta@localhost |
+----------------+----------------+
1 row in set (0.00 sec)
```

如果你不理解为什么选择一台或另一台主机，可以进行如下查询：

```
mysql> SELECT user, host FROM mysql.user WHERE user='sveta' ORDER
BY host DESC;
+-------+-----------+
| user  | host      |
+-------+-----------+
| sveta | localhost |
| sveta | %         |
+-------+-----------+
2 rows in set (0.00 sec)
```

MySQL 在表中按照从访问最多的主机到访问最少的主机的顺序对行进行排序，然后使用第一个找到的值。因此，它把我当作 sveta@localhost 用户进行连接，此时该用户没有 CREATE 权限。

- USER()、CURRENT_USER()函数和“SELECT user, host FROM mysql.user ORDER BY host DESE”查询语句是遇到权限问题时的首选。

另一种权限问题是你无法作为指定用户进行连接。在这种情况下，通常可以从错误消息中了解问题产生的原因，错误消息一般如下所示：

```
$mysql -usveta books
ERROR 1044 (42000): Access denied for user 'sveta'@'localhost' to database 'books'
```

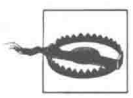
在看到这条消息以后，你了解了用户凭证。作为 root 超级用户进行连接，然后检查该用户是否存在以及是否拥有所需权限：

```
mysql> SELECT user, host FROM mysql.user WHERE user='sveta' ORDER
BY host DESC;
+-------+-----------+
| user  | host      |
+-------+-----------+
| sveta | localhost |
| sveta | %         |
+-------+-----------+
2 rows in set (0.00 sec)

mysql> SHOW GRANTS FOR 'sveta'@'localhost';
```

```
+-------------------------------------------------+
| Grants for sveta@localhost                      |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'sveta'@'localhost'       |
| GRANT SELECT ON `book`.* TO 'sveta'@'localhost' |
+-------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SHOW GRANTS FOR 'sveta'@'%';
+-------------------------------------------------+
| Grants for sveta@%                              |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'sveta'@'%'               |
| GRANT ALL PRIVILEGES ON `book`.* TO 'sveta'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

在这个输出信息中，你可以看到用户'sveta'@'localhost'仅仅对 book 数据库有权限，而对 books 数据库没有权限。现在，可以修复这个错误：赋予 sveta@localhost 用户必要的权限。

前面的示例讨论用户缺失必要权限的情况。对于用户被授予过多权限的情况也可以同样处理；仅需要移除不必要的权限。



警告

MySQL 的权限与其管控对象是分离的：这意味着当你赋予某用户权限时 mysqld 不会检查其是否存在，同时当授权对象被删除时也不会移除相应权限。这样做的好处是允许我们预先授予必要的权限，但同时也有可能在不经意的使用中带来潜在的问题。

作为最佳实践，我推荐你仔细学习 MySQL 的权限工作机制。尤其是在你想要在用户对象级别授予权限的时候，因为你需要理解在一个级别授权是如何影响其他授权的。同样，对于撤消权限情形也一样重要，甚至更重要，因为如果你以为已经撤消了某个权限但它依然存在，这就会造成意外的访问。

第2章

你不孤单：并发问题

MySQL 很少在单用户环境下使用。通常，它会同时处理很多的连接线程，这些线程来自不同的用户，执行不同的任务。这些并行连接可能会访问同样的数据库和表，所以当某个连接发生问题的时候，很难判断数据库的状态。

本章将讨论并行执行任务引发的问题。与第 1 章介绍的排错的场景不同，本章涉及的情况会更加复杂，你可能都不知道是哪个特定查询引发的问题。

并发问题的一个典型特征就是本来优化良好的查询突然变慢。这种变慢可能不是一直发生，不过哪怕偶尔随机出现几次，也应该引起你的注意去检查一下并发问题。

并发问题甚至会影响从服务器的 SQL 线程。为了纠正一个可能的误解，我明确地提出这个问题。人们可能会以为从服务器的 SQL 是单线程的[1]，不需要任何解决并发相关问题的技术。事实并非如此：主从服务器上的复制都会受到并发影响。因此本章会包含与复制相关的问题。

提示

首先，我们先来确认一些术语。对于每个连接的客户端，MySQL 服务器会创建一个独立线程。我使用术语线程、连接或者连接线程来代表处理客户端连接的线程。当根据上下文线程代表其他含义的时候，我会明确指出。我用“线程”来代替“查询”、“语句”和“事务”，因为当你解决问题的时候，你通常需要先隔离有问题的线程，然后再处理可能引发问题的语句。

2.1 锁和事务

MySQL 服务器有内部机制来避免用户损坏其他用户插入的数据。尽管通常情况下，这

1：以后会做修改：5.6.2 版本包含多线程从服务器功能的预览。

些内部机制默默而有效地运转着，以至于人们没有意识到这些安全机制也会引发你的应用程序或他人应用程序问题。因此，我先简单介绍 MySQL 服务器使用的并发控制机制。

MySQL 服务器用锁和事务来处理对其表的并发访问。我将先简单介绍锁的类型和事务处理，然后介绍排错的技术。

当线程请求数据集的时候就会加锁。在 MySQL 中，这可以是表、行、页或者元数据。当线程结束处理特定的数据集之后，它就会释放锁。2.2 节和 2.1.4 节详细介绍 MySQL 的锁设置。不同的章节分别介绍元数据锁，是因为这是一个新特性，并且根据 MySQL 服务器处理并发方式的不同而有所不同。如果你熟悉旧的表锁机制但不了解元数据锁，那么下面的章节可以帮你判断是否在特定情况下遇到了元数据锁。

数据库事务是处理一致性和可靠性工作的最小单位，这使得用户可以避免与其他事务交互时可能产生的风险。事务的隔离等级控制其他并发操作中的变化对本事务是否可见。2.3 小节将详细讨论 MySQL 的事务机制。

2.2 锁

MySQL 服务器和独立的存储引擎都可以设置锁。一般来说，读写操作的锁不同。读锁（或叫共享锁）允许并发线程读取加锁的数据，但禁止写数据。相反，写锁（或叫排他锁）阻止其他线程的读写操作。在存储引擎里，这些锁的实现方式可以不同，不过这些规则的基本原理是稳定的，并且几乎在任何地方都是相同的。

当用 SELECT 语句从表中查询数据或者通过 LOCK TABLE ... READ 语句显式加锁的时候，数据库将会设置读锁。当修改表或者用 LOCK TABLE ...WRITE 语句显式加锁的时候，数据库将会设置写锁。

提示

InnoDB 引擎使用简写的 S 代表读锁/共享锁，用 X 代表写锁/排它锁。你会在它的调试数据中看到这些缩写。

如前所述，MySQL 有 4 种类型的锁：表锁、行锁、页锁和元数据锁。顾名思义，表锁会锁住整个表，因此没有人可以访问表中任何行，直到持有锁的线程解锁该表。行锁的粒度更细一些，仅会锁住一行或者正在被线程访问的任何几行，因此同一个表中的其他行可以被其他并发线程访问。页锁会锁住一页，不过页锁仅在比较少见的 BDB 存储引擎中存在，因此我不会对其进行过多介绍。然而，解决锁问题一般推荐的方法对此类型的锁同样适用。

元数据锁是 MySQL 5.5 版本中的新特性。该锁仅对表中的元数据启用，当有线程开始使用表的时候，元数据锁会锁住表的所有元数据。元数据是 DDL（数据定义语言或叫

数据描述语言）语句的更改信息，如 CREATE、DROP 和 ALTER 等修改方案的语句。在老版本的 MySQL 中引入元数据锁是为了解决线程可以在其他线程中的并发事务使用相同表的情况下修改表定义或是删除表的问题。

在下面的章节会介绍表锁、行锁和元数据锁，以及你的应用程序可能由这些锁引发的问题。

2.2.1 表锁

当设置表锁的时候，整个表都被锁住。这意味着并发线程不能使用表，例如，如果设置的是读锁那么写访问是禁止的，如果设置的是写锁，那么访问读和写访问都是禁止的。当访问表并且该表所使用的存储引擎支持表锁的时候，即会产生表锁，比如 MyISAM 引擎。也可以在任何引擎上显式调用 LOCK TABLES 来产生表锁，在 5.5 之前的 MySQL 版本上也可使用 DDL 操作来产生表锁。

我一贯喜欢用示例来说明概念，这里有一个关于表锁的效果的示例：

```
mysql> SELECT * FROM t;
+-----+
| a   |
+-----+
|   0 |
| 256 |
+-----+
2 rows in set (3 min 18.71 sec)
```

获取两行记录需要 3 分钟？当我在讲座上展示这个示例的时候，我停下来提问是否有人知道原因。那时候，上网本刚开始流行，所有的听众都高喊：“它运行在 Atom CPU 上！”不过这种延迟对于现代处理器来说实在太大了。首先看一下表的定义，然后再次执行同一个请求：

```
mysql> SHOW CREATE TABLE t\G
*************** 1. row ***************
       Table: t
Create Table: CREATE TABLE `t` (
  `a` int(10) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`a`)
) ENGINE=MyISAM AUTO_INCREMENT=257 DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

mysql> SELECT * FROM t;
+-----+
| a   |
+-----+
|   0 |
| 256 |
+-----+
2 rows in set (0.00 sec)
```

现在它几乎没有耗时！

为了查明究竟发生了什么，我们需要在查询运行缓慢的时候执行 SHOW PROCESSLIST 命令。

提示

在实际的应用程序环境中，你要么在忙时手动执行诊断查询，要么采用定时作业来帮助你保存结果。

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1311
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 35
  State: Locked
   Info: SELECT * FROM t
*************************** 2. row ***************************
     Id: 1312
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 36
  State: User sleep
   Info: UPDATE t SET a=sleep(200) WHERE a=0
*************************** 3. row ***************************
     Id: 1314
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
3 rows in set (0.00 sec)
```

输出中的字段解释如下。

Id

MySQL 服务器中运行的连接线程的 ID。

User、Host 和 db

客户端连接到服务器时使用的连接选项。

Command

线程中当前执行的的命令。

Time

从线程开始执行命令到现在消耗的时间。

State

线程的内部状态。

Info

表明线程当前正在进行的工作。如果展示的是查询语句，表明该语句正在执行；如果值是 NULL，表明线程正在休眠，并等待下一条用户命令。

为了查明查询究竟发生了什么，我们需要找到 Info 输出中包含查询文本的行，然后检查查询的状态。在输出信息的最顶端，可以看到查询的状态是锁定的(Locked)，这意味着该查询无法执行，因为其他线程持有该线程正在等待的锁。下面的语句：

```
UPDATE t SET a=sleep(200) WHERE a=0
```

访问相同的表，并且已经执行了 36 秒。因为是同一个表并且再没有其他线程使用该表，所以我们可以推断该更新阻止该查询开始执行。事实上，该查询需要等待 200 秒，直到另一个查询执行完成。

- 你刚刚学到一个新的重要调试技巧：当你怀疑是并发线程影响了查询的时候，使用 SHOW PROCESSLIST 命令查看状态。

2.2.2 行锁

行锁会锁住一些行，而不是整张表。因此，可以修改表中没有被行锁锁住的行。

行锁在存储引擎的级别进行设置。InnoDB 是当前使用行锁的主要的存储引擎。

为了展示行锁和表锁的区别，我们把之前使用过的示例稍作修改：

```
mysql>  CREATE TABLE `t` (
    ->   `a` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    ->   PRIMARY KEY (`a`)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (1.29 sec)

mysql>  INSERT INTO t VALUES();
Query OK, 1 row affected (0.24 sec)

mysql>  INSERT INTO t SELECT NULL FROM t;
Query OK, 1 row affected (0.19 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql>  INSERT INTO t SELECT NULL FROM t;
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql>  INSERT INTO t SELECT NULL FROM t;
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
mysql> INSERT INTO t SELECT NULL FROM t;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t;
+----+
| a  |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  6 |
|  7 |
|  8 |
|  9 |
| 13 |
| 14 |
| 15 |
| 16 |
| 17 |
| 18 |
| 19 |
| 20 |
+----+
16 rows in set (0.00 sec)
```

再次运行同样含有休眠的 UPDATE 查询，比较一下行锁和表锁的不同效果：

```
mysql> UPDATE t SET a=sleep(200) WHERE a=6;
```

当休眠语句执行的时候，我们有充分的时间去使用其他客户端执行查询操作：

```
mysql> SELECT * FROM t;
+----+
| a  |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  6 |
|  7 |
|  8 |
|  9 |
| 13 |
| 14 |
| 15 |
| 16 |
| 17 |
| 18 |
| 19 |
| 20 |
+----+
16 rows in set (0.00 sec)
```

这次立即得到结果。现在尝试更新行：

```
mysql> UPDATE t SET a=23 WHERE a=13;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE t SET a=27 WHERE a=7;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

更新行操作也没有被锁，运行良好。如果我们使用与休眠的 UPDATE 语句中相同的 WHERE 条件更新行，会发生什么？

```
mysql> UPDATE t SET a=26 WHERE a=6;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

新查询等待 innodb_lock_wait_timeout 参数里设置的时间（默认值是 50 秒），然后报错退出。由于数据一致性，我们得到跟表锁一样的结果，不过除非并发线程试图访问正好被锁住的行，否则 InnoDB 的锁不会对并发线程产生影响。

提示

事实上，行锁比我描述的复杂得多。例如，如果我们使用无法通过唯一键（UNIQUE）解析的 WHERE 条件访问表，我们就无法并行修改任何行，因为存储引擎无法判断其余线程是否要更新相同的行。这并不是我在讨论两种级别的锁的时候为节省排错技巧的篇幅略过的唯一细节。可以查阅附录以获取更多关于 MySQL 锁的信息和资料。

现在，查看如何更改进程列表的输出信息。在这个例子中，我会使用 INFORMATION_SCHEMA.PROCESSLIST 表。实际上，该表包含的信息与 SHOW PROCESSLIST 命令展示的信息相同，不过由于这里是保存在表中，因此可以根据需要排序查询结果。在你有很多并发线程的时候，这会带来很大的便利：

```
mysql> SELECT * FROM PROCESSLIST\G
*************************** 1. row ***************************
     ID: 4483
   USER: root
   HOST: localhost
     DB: NULL
COMMAND: Sleep
   TIME: 283
  STATE:
   INFO: NULL
*************************** 2. row ***************************
     ID: 4482
   USER: root
   HOST: localhost
     DB: information_schema
COMMAND: Query
   TIME: 0
  STATE: executing
   INFO: SELECT * FROM PROCESSLIST
*************************** 3. row ***************************
     ID: 4481
```

```
   USER: root
   HOST: localhost
     DB: test
COMMAND: Query
   TIME: 7
  STATE: Updating
   INFO: UPDATE t SET a=26 WHERE a=6
*************************** 4. row ***************************
     ID: 4480
   USER: root
   HOST: localhost
     DB: test
COMMAND: Query
   TIME: 123
  STATE: User sleep
   INFO: UPDATE t SET a=sleep(200) WHERE a=6
4 rows in set (0.09 sec)
```

这里你可以看到另一个与之前示例不同的地方：该查询的状态是 Updating，而不是 Locked。

为了确定在 InnoDB 中一个请求是否阻塞，可以执行 SHOW ENGINE INNODB STATUS 命令，该命令是 InnoDB 监控器机制的一部分。该命令在分析并发多语句事务的作用的时候尤为有用，我们将在本章晚些时候讨论。这里不会展示该工具所有的输出，而仅给出与当前示例有关的部分。2.8.2 节和第 6 章将详细讨论该工具。

```
mysql> SHOW ENGINE INNODB STATUS \G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
110802  2:03:45 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 41 seconds
...
------------
TRANSACTIONS
------------
Trx id counter 0 26243828
Purge done for trx's n:o < 0 26243827 undo n:o < 0 0
History list length 25
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, OS thread id 101514240
MySQL thread id 4483, query id 25022097 localhost root
show engine innodb status
---TRANSACTION 0 26243827, ACTIVE 9 sec, OS thread id 101403136 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 4481, query id 25022095 localhost root Updating
update t set a=26 where a=6
------- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
```

```
trx id 0 26243827 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 0000019072e3; asc     r ;; 2:
 len 7; hex 800000002d0110; asc     -  ;;

------------------
---TRANSACTION 0 26243821, ACTIVE 125 sec, OS thread id 101238272,
thread declared inside InnoDB 500
mysql tables in use 1, locked 1
2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 4480, query id 25022091 localhost root User sleep
update t set a=sleep(200) where a=6
```

我们需要重点关注的部分如下所示。

```
---TRANSACTION 0 26243827, ACTIVE 9 sec, OS thread id 101403136 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 4481, query id 25022095 localhost root Updating
update t set a=26 where a=6
------- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
```

上述信息表明该查询在等待锁。

在回到锁之前，先详细介绍上面的输出信息。

`TRANSACTION 0 26243827`

这是事务的 ID。

`ACTIVE 9 sec`

事务有效的秒数。

`OS thread id 101403136`

执行事务的 MySQL 线程 ID。

`starting index read`

表明事务正在做什么工作。

`mysql tables in use 1, locked 1`

表明有多少表被使用和被锁住。

`LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)`

锁的相关信息。

`MySQL thread id 4481, query id 25022095 localhost root Updating`

MySQL 线程的相关信息，包括：线程 ID、查询 ID、用户凭证和 MySQL 状态。

`update t set a=26 where a=6`

当前执行的查询。

下面是关于锁的详细信息。

```
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26243827 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 0000019072e3; asc     r ;; 2:
 len 7; hex 800000002d0110; asc     -  ;;
```

该信息表明 InnoDB 表空间中阻塞事务的详细信息，包括锁的类型（排他锁，因为我们要进行更新操作）和物理记录的二进制内容。

最后，查看关于执行锁住行的查询的事务的信息：

```
---TRANSACTION 0 26243821, ACTIVE 125 sec, OS thread id 101238272,
thread declared inside InnoDB 500
mysql tables in use 1, locked 1
2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 4480, query id 25022091 localhost root User sleep
update t set a=sleep(200) where a=6
```

既然情况明了，就可以考虑如何修复它。

- 你刚刚学到了另一个重要的排错工具：InnoDB 监控器，该工具可以通过 SHOW ENGINE INNODB STATUS 命令调用。2.8.2 节会详细介绍该工具。

关于 1.6 节中的性能排错问题，我有一点需要说明。那一节提到索引会降低插入的性能，因为在插入数据的同时需要更新索引文件。不过，当使用行锁的时候，索引会提升整个应用程序的性能，尤其当索引唯一的时候，因为当更新这种类型的索引字段的时候，插入不会阻塞对整个表的访问。

下面将暂时从锁的介绍转向事务的介绍。然后将会回到元数据锁。本章如此编排内容，是因为在讨论元数据锁之前我们需要对事务有所了解。

2.3 事务

MySQL 在存储引擎级别提供事务支持。在官方提供的存储引擎中，最受欢迎的 InnoDB 引擎就提供了事务的支持。本书将会讨论如何解决 InnoDB 事务问题。

在 MySQL 中，可以通过 START TRANSACTION 或 BEGIN 语句启动事务；通过 COMMIT 语句提交事务；通过 ROOLBACK 语句回滚事务（取消事务）。

另一种启动多语句事务的方法是将 autocommit 变量的值设置为 0。这将覆盖 MySQL 的默认行为，即在每条语句后发送一个隐式提交指令。当 autocommit 设置为 0 之后，需要显式调用 COMMIT 或 ROLLBACK 命令。在那之后，下一条语句会自动开始新事务。

MySQL 也提供了 SAVAPOINT 和 XA 事务接口。尽管 InnoDB 引擎支持两者，但是本书不打算介绍它们，因为这不会给我们带来任何解决问题的额外技术。换句话说，我介绍的相同技术也适用于这类事务。

2.3.1 隐藏查询

InnoDB 存储引擎把对数据的每个请求都作为事务进行处理。无论事务是单语句或者多语句均可。对于排错来说，可以像 2.2.2 节中介绍的那样处理单个查询事务。你需要知道哪个是当前执行的查询以及哪个锁阻碍了其他查询。

当你的事务是由很多语句组成的时候情况就不同了。在这种情况下，即使你在 SHOW PROCESSLIST 的输出中看不到任何查询，事务也有可能锁住了行。

为了举例说明这个问题，再次修改示例。现在我们甚至无须调用 sleep 命令来产生延迟。在运行这个示例前，我恢复了修改的行，因此该表中的值与初始测试设置中的值相同。

```
mysql1> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql1> UPDATE t SET a=26 WHERE a=6;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

请注意，此时事务没有关闭。从另一个连接启动另一个事务：

```
mysql2> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql2> UPDATE t SET a=36  WHERE a=6;
```

现在，运行 SHOW PROCESSLIST 命令。该查询与行锁示例中的状态一样，都是 Updating，不过这次不清楚是什么阻止了更新操作：

```
mysql3> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 4484
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 104
  State:
   Info: NULL
*************************** 2. row ***************************
     Id: 4485
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 2
  State: Updating
   Info: UPDATE t SET a=36  WHERE a=6
*************************** 3. row ***************************
     Id: 4486
   User: root
   Host: localhost
```

```
     db: test
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
*************************** 4. row ***************************
     Id: 4487
   User: root
   Host: localhost
     db: NULL
Command: Sleep
   Time: 33
  State:
   Info: NULL
4 rows in set (0.09 sec)
```

下面是从 SHOW ENGINE INNODB STATUS 命令的输出中筛选出的信息：

```
mysql> SHOW ENGINE INNODB STATUS\G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
110802 14:35:28 INNODB MONITOR OUTPUT
=====================================
...
------------
TRANSACTIONS
------------
Trx id counter 0 26243837
Purge done for trx's n:o < 0 26243834 undo n:o < 0 0
History list length 2
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, OS thread id 101515264
MySQL thread id 4487, query id 25022139 localhost root
show engine innodb status
---TRANSACTION 0 26243836, ACTIVE 4 sec, OS thread id 101514240
starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 4485, query id 25022137 localhost root Updating
update t set a=36  where a=6
------- TRX HAS BEEN WAITING 4 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26243836 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 0000019072fb; asc     r ;; 2:
 len 7; hex 000000003202ca; asc     2  ;;

------------------
---TRANSACTION 0 26243835, ACTIVE 106 sec, OS thread id 100936704
2 lock struct(s), heap size 320, 1 row lock(s), undo log entries 2
MySQL thread id 4484, query id 25022125 localhost root
```

提示

同样的方法对被 LOCK TABLE 查询锁住的 MyISAM 表也适用，因而这部分信息没有必要显示在 SHOW PROCESSLIST 命令的输出中。InnoDB 会在其状态输出中会输出有关这些表的信息：

```
------------
TRANSACTIONS
------------
Trx id counter B55
Purge done for trx's n:o < B27 undo n:o < 0
History list length 7
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started
MySQL thread id 3, query id 124 localhost ::1 root
show engine innodb status

---TRANSACTION B53, not started
mysql tables in use 1, locked 1

MySQL thread id 1, query id 115 localhost ::1 root
--------
```

锁的信息与我们在 2.2.2 节中看到的类似。

```
------- TRX HAS BEEN WAITING 4 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26243836 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 0000019072fb; asc     r ;; 2:
 len 7; hex 000000003202ca; asc     2  ;;
```

尽管上述信息没有明确指出谁持有锁，但是仍清楚地显示出该事务在等待锁。如果你使用的是 5.0 版本或者 5.1 版本中捆绑的 InnoDB 引擎，你有两种选择：自己找出原因，或者使用 InnoDB 锁监控器。在该示例里，仅有两个事务，所以很容易得到答案，也就是说“自己找到答案”是可行的。不过，如果你有很多连接在使用同一个表中的不同行，那就不会这么简单了。2.8.2 节中将会介绍 InnoDB 锁监控器。这里采用第三种选择，仅适用于 InnoDB 插件的情况。

InnoDB 插件丰富了监控功能，在 INFORMATION_SCHEMA 库中包含一些相关表，分别叫做 INNODB_LOCKS、INNODB_LOCK_WAITS（保存已获取的锁和等待的锁的信息）和 INNODB_TRX（保存正在执行的事务的信息）。

对于该示例，可以查询下面这些表：

```
mysql> SELECT * FROM innodb_locks\G
*************************** 1. row ***************************
    lock_id: 3B86:1120:3:6
lock_trx_id: 3B86
  lock_mode: X
  lock_type: RECORD
 lock_table: `test`.`t`
 lock_index: `PRIMARY`
 lock_space: 1120
  lock_page: 3
```

```
   lock_rec: 6
  lock_data: 6
*************************** 2. row ***************************
    lock_id: 3B85:1120:3:6
lock_trx_id: 3B85
  lock_mode: X
  lock_type: RECORD
 lock_table: `test`.`t`
 lock_index: `PRIMARY`
 lock_space: 1120
  lock_page: 3
   lock_rec: 6
  lock_data: 6
2 rows in set (0.01 sec)
```

这是关于锁的信息。两个事务都对同一条记录设置锁，不过从上述信息中无法得知哪个事务持有锁，哪个事务在等待锁。对于这部分细节信息，可以从 INNODB_LOCK_WAITS 表中获得：

```
mysql> SELECT * FROM innodb_lock_waits\G
*************************** 1. row ***************************
requesting_trx_id: 3B86
requested_lock_id: 3B86:1120:3:6
  blocking_trx_id: 3B85
 blocking_lock_id: 3B85:1120:3:6
1 row in set (0.09 sec)
```

requesting_trx_id 列的值就是我们“挂起”的事务的 ID，blocking_trx_id 列的值为持有锁的事务的 ID，而 requested_lock_id 列和 blocking_lock_id 列的值分别代表被请求和阻塞的锁的 ID 信息。

现在我们所需要知道的就是正在阻塞中的事务的 MySQL 进程的 ID，因此我们需要想点办法。INNODB_TRX 表可以帮助我们找到该 ID：

```
mysql> SELECT * FROM innodb_trx\G
*************************** 1. row ***************************
                    trx_id: 3B86
                 trx_state: LOCK WAIT
               trx_started: 2011-08-02 14:48:51
     trx_requested_lock_id: 3B86:1120:3:6
          trx_wait_started: 2011-08-02 14:49:59
                trx_weight: 2
       trx_mysql_thread_id: 28546
                 trx_query: UPDATE t SET a=36  WHERE a=6
       trx_operation_state: starting index read
         trx_tables_in_use: 1
         trx_tables_locked: 1
          trx_lock_structs: 2
     trx_lock_memory_bytes: 320
           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: 10000
*************************** 2. row ***************************
                  trx_id: 3B85
               trx_state: RUNNING
             trx_started: 2011-08-02 14:48:41
   trx_requested_lock_id: NULL
        trx_wait_started: NULL
              trx_weight: 4
     trx_mysql_thread_id: 28544
               trx_query: NULL
     trx_operation_state: NULL
       trx_tables_in_use: 0
       trx_tables_locked: 0
        trx_lock_structs: 2
   trx_lock_memory_bytes: 320
         trx_rows_locked: 1
       trx_rows_modified: 2
 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: 10000
2 rows in set (0.11 sec)
```

正在阻塞的事务的 ID 是 3B85。所以，输出中的第二行是有关该事务的行，从中可得知 trx_mysql_thread_id 为 28544。可以通过 SHOW PROCESSLIST 命令来确认该信息：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 28542
   User: root
   Host: localhost
     db: information_schema
Command: Sleep
   Time: 46
  State:
   Info: NULL
*************************** 2. row ***************************
     Id: 28544
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 79
  State:
   Info: NULL
*************************** 3. row ***************************
     Id: 28546
   User: root
   Host: localhost
     db: test
```

```
Command: Query
   Time: 1
  State: Updating
   Info: UPDATE t SET a=36  WHERE a=6
*************************** 4. row ***************************
     Id: 28547
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
4 rows in set (0.01 sec)
```

既然我们已经知道了 MySQL 线程的 ID，我们就可以对该阻塞的事务做任何想要的操作：继续等待其完成还是终止该事务均可。如果我们在应用程序中执行该命令，我们还可以分析是什么导致了此类锁定问题，并可以进行修正以避免今后再发生问题。

实际上，INNODB_TRX 表包含了很多关于事务的有用信息。如果回顾该示例，可以看到等待的事务的状态是 trx_state：LOCK WAIT，而正在运行的事务的状态是 trx_state：RUNNING。这里不会过多地讨论这部分信息，不过第 6 章会继续探讨它。

- 我们刚刚学习了一个未提交的事务会持有锁，哪怕该影响到特定行的查询在数小时前已经结束了。

你在编码时应牢记这些知识。我曾见过用户在环境中设置 autocommit=0，从而导致事务运行数小时。这会导致很难发现和理解的问题，特别是当用户没有做好准备的时候。这种环境通常在流行的 Java 的框架中使用，即默认在 URL 中添加 autocommit=0。

- 概括来说，当执行多语句事务的时候，应尽可能及时提交事务。哪怕事务不会修改任何行，也不要在事务最后的更新都已经完成的情况下仍不提交事务。

2.3.2 死锁

死锁是指当两个或多个竞争事务彼此等待对方释放锁，从而导致事务永远无法终止的情况。在行锁级别，死锁是无法 100%避免的。

InnoDB 引擎有内部的死锁探测器。当其发现有死锁的时候，它会回滚其中的一个事务，并会报告一个立即可见的错误。当设计应用程序的时候，你需要对此情况有所准备并合理地处理回滚。

关于死锁的信息可通过 SHOW ENGINE INNODB STATUS 命令获取。我们将通过一个简单的死锁示例来说明这部分内容。

初始化数据如下所示。

```
mysql> CREATE TABLE `t` (
`a` int(10) unsigned NOT NULL AUTO_INCREMENT,
PRIMARY KEY (`a`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.27 sec)

mysql> INSERT INTO t VALUES();
Query OK, 1 row affected (0.16 sec)

mysql> INSERT INTO t SELECT NULL FROM t;
Query OK, 1 row affected (0.11 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT NULL FROM t;
Query OK, 2 rows affected (0.09 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
+---+
4 rows in set (0.00 sec)
```

现在启动两个事务并且在每个事务中插入一行数据：

```
mysql1> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql1> INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)

mysql1> SELECT * FROM t;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 8 |
+---+
5 rows in set (0.00 sec)

mysql2> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql2> INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)

mysql2> SELECT * FROM t;
+---+
| a |
+---+
```

```
| 1 |
| 2 |
| 3 |
| 4 |
| 9 |
+---+
5 rows in set (0.00 sec)
```

目前为止一切正常。两条语句都在自增的字段插入了一个值。现在我们尝试在第一个事务中修改一行记录：

```
mysql1> UPDATE t SET a=9 WHERE a=8;
```

在其等待的过程中，在第二个事务中修改该行记录：

```
mysql2> UPDATE t SET a=8 WHERE a=9;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

该查询立即失败并返回出现死锁的错误消息。与此同时，第二查询正确完成：

```
Query OK, 1 row affected (9.56 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

你刚刚看到了 InnoDB 的死锁探测器是如何工作的。为了弄清楚究竟发生了什么，我们再次检查 SHOW ENGINE INNODB STATUS 命令的输出信息：

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
110803  3:04:34
*** (1) TRANSACTION:
TRANSACTION 3B96, ACTIVE 29 sec, OS thread id 35542016 updating or deleting
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 320, 2 row lock(s), undo log entries 2
MySQL thread id 30446, query id 772 localhost root Updating
update t set a=9 where a=8
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1121 page no 3 n bits 80 index `PRIMARY` of table `test`.`t`
trx id 3B96 lock mode S locks rec but not gap waiting
Record lock, heap no 8 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000009; asc     ;;
 1: len 6; hex 000000003b97; asc     ; ;;
 2: len 7; hex 510000022328d5; asc Q   #( ;;

*** (2) TRANSACTION:
TRANSACTION 3B97, ACTIVE 21 sec, OS thread id 35552256 updating or deleting
mysql tables in use 1, locked 1
3 lock struct(s), heap size 320, 2 row lock(s), undo log entries 2
MySQL thread id 30447, query id 773 localhost root Updating
update t set a=8 where a=9
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 1121 page no 3 n bits 80 index `PRIMARY` of table `test`.`t`
trx id 3B97 lock_mode X locks rec but not gap
Record lock, heap no 8 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000009; asc     ;;
 1: len 6; hex 000000003b97; asc     ; ;;
 2: len 7; hex 510000022328d5; asc Q   #( ;;
```

```
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 1121 page no 3 n bits 80 index `PRIMARY` of table `test`.`t`
trx id 3B97 lock mode S locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000008; asc     ;;
 1: len 6; hex 000000003b96; asc     ; ;;
 2: len 7; hex 50000002221b83; asc P    "  ;;


*** WE ROLL BACK TRANSACTION (2)
```

该输出包含了许多最近一次死锁的信息以及为什么会发生死锁。你需要注意的是WAITING FOR THIS LOCK TO BE GRANTED 相关的部分（该部分表明事务在等待哪个锁）以及 HOLDS THE LOCK(S)部分（该部分表明阻塞事务的锁的信息）。该知识在实际应用中更为重要，比如，查询是由 Web 应用的用户交互产生的，而你无法精确定位在特定的时间执行的是哪个查询。

为了应对潜在的死锁，你需要像第 1 章介绍的一样，在应用程序中添加错误处理功能。如果你得到表明是死锁的错误消息并且已经产生回滚，就需要重启事务。

2.3.3 隐式提交

有些语句在没有显式调用COMMIT语句的时候也会提交事务。这种情况称为隐式提交，并且如果你没意识到你正在提交事务，这很有可能导致一致性问题。

很多语句都会产生隐式提交。这里不一一列举，因为它们会随版本的更替而不同。一般来说，DDL 语句与事务相关的语句和管理语句都会产生隐式提交，而那些操作数据的语句则不会产生隐式提交。

意料之外的隐式提交的一个典型特征就是你会在表中发现非预期的数据，尽管你认为插入数据的语句应该回滚。下面是一个示例：

```
mysql> CREATE TABLE t1(f1 INT) ENGINE=InnoDB;
Query OK, 0 rows affected (0.14 sec)

mysql> SELECT * FROM t1;
Empty set (0.00 sec)

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

mysql> INSERT INTO t1 VALUES(100);
Query OK, 1 row affected (0.03 sec)

mysql> CREATE TABLE t2 LIKE t1;
Query OK, 0 rows affected (0.19 sec)

mysql> INSERT INTO t1 VALUES(200);
Query OK, 1 row affected (0.02 sec)

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

CREATE TABLE 语句产生了隐式提交。所以，即使你以为两条插入语句都进行了回滚，t1 表也包含值为 100 的行数据。第二个值为 200 的插入会如预期一样回滚。

这个示例运行的前提是设置了 autocommit=0，这样才会默认使用多语句事务。当讨论提交的时候，无须再强调 autocommit 的默认值是 1，也就是默认情况下，当没有显式调用 BEGIN 或 START TRANSACTION 命令时不会使用多语句事务。当 autocommit 的值是 1 的时候，每条语句都会立即提交，也就是说，在前面的示例中，两行记录实际上都会写入表中：

```
mysql> SELECT * FROM t1;
+------+
| f1   |
+------+
|  100 |
|  200 |
+------+
2 rows in set (0.00 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
1 row in set (0.00 sec)
```

- 一般来说，为了避免这种问题，应保持事务短小精悍，这样即使你因为错误地使用了导致隐式提交的语句中断了事务，影响也会最小化。

2.4 元数据锁

为了确保数据一致性，在有其他事务使用表的情况下，对该表的 DDL 操作应该阻塞。从 5.5.3 版本开始，MySQL 使用元数据锁来实现这一特性。

当事务开始的时候，它会获取所有需要使用的表上的元数据锁，并在事务结束后释放锁。所有其他想要修改这些表定义的线程都需要等待事务结束。

5.5.3 版本之前的 MySQL 上的 DDL 操作没有并行事务的概念。这会导致类似下面的冲突：

```
mysql1> BEGIN;
Query OK, 0 rows affected (0.08 sec)

mysql1> SELECT * FROM t1;
+------+
| f1   |
+------+
|  100 |
|  200 |
+------+
2 rows in set (0.10 sec)
```

在一个事务中，我们从表中查询数据，并计划在当前事务中使用该查询的结果集。与此同时，另一个线程删除表：

```
mysql2> DROP TABLE t1;
Query OK, 0 rows affected (0.17 sec)
```

DROP 是不可回滚的操作，因此第一个线程因为冲突受到了影响：

```
mysql> SELECT * FROM t1;
ERROR 1146 (42S02): Table 'test.t1' doesn't exist
```

该事务显然不能正常完成。而元数据锁则可以在其他连接执行 DROP 语句之前，让该事务正常完成。为了举例说明这个特点，我们在 5.5.3 或之后的版本上执行同样的例子。

```
mysql1> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql1> SELECT * FROM t1;
+------+
| f1   |
+------+
|  100 |
|  200 |
+------+
2 rows in set (0.00 sec)
```

现在再试图执行 DROP 会被阻塞：

```
mysql2> DROP TABLE t1;
```

在执行完这条命令后，我等待了若干秒，这样你就会知道 drop 执行了多久，然后我才回滚第一个事务：

```
mysql1> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

现在我们可以看一下查询的执行时间，确认 DROP 操作是等待第一个事务执行完成才执行的：

```
mysql2> DROP TABLE t1;
Query OK, 0 rows affected (1 min 0.39 sec)
```

新模型更安全，并且这种方式不再需要掌握任何新的排错技术。不过 MySQL 在元数据锁引入之前已经存在很久了，并且很多用户已经熟悉了老的操作习惯，甚至为其定制了工作方案。因此，我打算增加一些说明来介绍服务器行为中因元数据锁而产生的不同。

元数据锁对比旧的模型

元数据锁的获取不依赖于使用的存储引擎。因此，无论使用的是设置 autocommit=0 的 MyISAM 引擎还是用 BEGIN 或 START TRANSACTION 语句显式声明的事务，连接都会取得元数据锁。可以在 SHOW PROCESSLIST 的输出中清楚地看到这些信息，在输出中你会看到一些状态为“Waiting for table metadata lock”的语句。

用一个小示例来说明元数据锁的使用。第一个线程启动一个事务以访问 MyISAM 引擎的表。这里使用了 BEGIN 语句，不过如果设置 autocommit=0 也会产生同样的效果：

```
mysql1> SHOW CREATE TABLE tm\G
*************************** 1. row ***************************
       Table: tm
Create Table: CREATE TABLE `tm` (
  `a` int(10) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`a`)
) ENGINE=MyISAM AUTO_INCREMENT=5 DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

mysql1> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql1> SELECT * FROM tm;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
+---+
4 rows in set (0.00 sec)
```

同时，另一个线程调用 TRUNCATE 命令，该命令会对表的元数据产生影响：

```
mysql2> TRUNCATE TABLE tm;
```

通过第三个连接来执行 SHOW PROCESSLIST 命令，可以看到前面两个线程的状态：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 30970
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 26
  State:
   Info: NULL
*************************** 2. row ***************************
     Id: 30972
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 9
  State: Waiting for table metadata lock
   Info: TRUNCATE TABLE tm
*************************** 3. row ***************************
     Id: 31005
```

```
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
3 rows in set (0.00 sec)
```

当一个查询在等待元数据锁的时候被阻塞，可以使用 SHOW PROCESSLIST 命令。当迁移到支持元数据锁的 MySQL 版本之后，你会发现 DDL 查询开始变慢了。这是因为当其他事务持有锁的时候它们不得不等待。

从理论上来说，元数据锁会超时。可以通过 lock_wait_timeout 变量指定超时时间。默认是 31 536 000 秒（一年），所以实际上锁住的查询永远不会终止：

```
mysql> truncate table tm;
Query OK, 0 rows affected (5 hours 12 min 52.51 sec)
```

为使超时有效果，可以设置 lock_wait_timeout 为一个较小的值，例如一秒：

```
mysql> set lock_wait_timeout=1;
Query OK, 0 rows affected (0.00 sec)

mysql> truncate table tm;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

2.5 并发如何影响性能

我们刚刚讨论了当并发线程或事务产生冲突从而导致性能问题或者甚至查询中止的情况。你已经了解了 SQL 语句或者存储引擎设置的锁是如何影响并发线程的。这些锁是对用户可见的，尽管它们并非始终容易调试，但是很容易追踪。当应用程序可能产生并发的 MySQL 连接的时候，你需要了解可能的并行竞争的线程。这在 Web 服务器上尤为常见，Web 服务器会打开与 MySQL 服务器的并行连接，因为会有很多用户打开由 Web 服务器提供的网页。

我们也讨论了在应用程序处理错误时难以避免的结果错误或者显著变慢问题。当我为本书搜集示例的时候，我把所有这样的示例都放在 1.2 节中。我把这些问题区别于性能问题，因为你可以立即看到它们的结果，而性能问题通常情况下是先隐蔽起来的，并且你一般是在检查了慢查询日志或者接到关于应用缓慢的用户投诉之后才会注意到它们。

下面处理更为隐蔽的性能问题。如果一个查询突然开始执行缓慢，第一步应该确认它是否是合理优化过的。最简单的确认方式就是在一个隔离的、单线程的环境里去执行该查询。如果该查询仍然执行缓慢，那么它或者需要优化，或者最近的大量更新操作导致索引统计数据过期了（第 1 章包含基本的优化技巧）。

如果一个查询在单线程环境中很快完成但是在多线程环境中执行缓慢，这基本可以确定你遇到了并发问题。本书介绍过的所有应对错误结果的技术也适用于这种情况。慢查询只是相对略微复杂的问题，因为要调试这些问题，你需要重再现你遇到问题时候的情况，而当你想要重现的时候却又很难遇到。

提示

我经常强调重现问题，而不是仅仅移除有问题的查询。对于并发问题来说，这很重要，因为有问题的查询可能仅仅是深层次问题的一个表象。如果你仅是停止执行查询而不是解决掉真正的问题，你很可能会在应用程序的其他部分遭遇同样的问题。

2.5.1 为并发问题监控 InnoDB 事务

如果你正在调试由 InnoDB 事务引起的锁问题，InnoDB 监控器会减轻你的工作。你仅需要打开监控器，然后它便会定期将消息转储到错误日志文件中，这与你使用 SHOW ENGINE INNODB STATUS 命令看到的输出类似。

要打开 InnoDB 监控器，需要在任何数据库中创建一个叫做 innodb_monitor 的表：

```
$mysql test -A
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2624
Server version: 5.1.59-debug Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

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

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

mysql> CREATE TABLE innodb_monitor(f1 INT) ENGINE=InnoDB;
Query OK, 0 rows affected (0.48 sec)
```

MySQL 客户端命令中的-A 选项在你尝试调试与并发有关的问题时非常有用。正常情况下，客户端会请求可用表列表。然后，客户端会被其他连接持有的锁阻塞，这会阻塞客户端调试。选项-A 会阻止表列表请求。

如果你这么做了，InnoDB 引擎会识别出该表并且开始向错误日志文件中输出信息。所以，如果你检查在慢查询运行时记录的事务信息，你可以找出谁持有了阻塞查询的锁。关于 InnoDB 监控器的更多信息可在第 6 章以及 2.8.2 节中看到。

2.5.2 为并发问题监控其他资源

如果你调试的查询没有使用 InnoDB 存储引擎或者你怀疑问题是由于其他不同类型的

锁引起的，那么我们仍然有许多选择。可以使用 SHOW PROCESSLIST 命令，不过更好的选择是定时重复地执行 SELECT… FROM INFORMATION_SCHEMA.PROCESSLIST 命令并且将其输出以及其他信息一起保存到文件或表中。后者的输出信息比 SHOW PROCESSLIST 的输出更好理解和掌握。当通过慢查询找到阻塞查询问题的线索的时候，可以检同时查进程列表信息。

并发查询对性能的影响不总是停留在 SQL 或者存储引擎层。MySQL 服务器中的多个线程也会共享如内存（RAM）和 CPU 等硬件资源。把这些资源中的一部分分配给每个线程，把其他（如临时表等）资源分配给指定类型的操作并且只在必要的时候分配。一些资源被所有线程共享。你在设计应用程序时也需要考虑到操作系统的限制。本节将会介绍这些资源影响性能的一些概念，然后第 3 章将会详细介绍控制资源使用的选项。

从每个线程的内存分配开始。MySQL 服务器有许多选项让你可以设置特定于线程的缓冲区大小。简单来说，分配的内存越多，线程运行得越快。然而这些缓冲区是分配给每个运行的单独线程的，所以给每个线程分配的资源越多，能同时运行的线程数就越少。所以，人们总是寻求能提升性能的值和实际物理内存间的平衡。

分配给指定操作类型的资源也是有限制的。除非需要，不要设置得过大，当然也不要设置得过小。一个好的做法是仅给一小部分需要大缓冲区的会话（连接）设置一个较高的值，而其他的设置为默认值。这种级别的控制可以基于每个会话动态地进行设置：

```
SET SESSION join_buffer_size=1024*1024*1024;
```

第三类资源是被所有线程共享的资源，最常见的是内部缓存。有了这些配置选项，你一般不用担心增加更多的连接会增加内存使用。不过有一个潜在的问题，那就是修改数据会使缓存失效从而导致随后的语句消耗更长的时间。通常这是一个很快的操作，不过如果缓存过大，失效时间可能会很长，从而在线程等待缓存恢复访问的时候会影响用户使用。

最后，性能问题可能会由于操作系统的资源引起，如文件描述符和 CPU。系统上文件描述符的数量会限制服务器可以创建的连接数量，可以同时打开的表的数量，甚至同一个表分区的数量。如果表的分区数大于可用文件操作符的数量，你会发现连打开一个表都不可能。第 4 章会讨论操作系统如何影响 MySQL 服务器。

2.6 其他锁问题

另一些可能影响应用程序的因素包括内部锁和服务器在运行特定操作时的一些互斥请求。不过大多数时候它们保证了数据的完整性。不过有一些异常，比如 InnoDB 互斥和循环锁，你不能也不应该尝试控制它们，不过由于它们中的一部分可能会对用户应用程序可见，因此这里将介绍它们。

事务可能会产生争用情形和导致死锁，当然也可以产生其他情况。当 MySQL 服务器开始使用如文件等资源，或者修改线程间共享的变量的时候，它会锁住资源以避免其他线程对同一个资源的并发访问。这解决了数据一致性的问题。不过同时，这种保护又会带来死锁。

这种死锁很难诊断并且从理论上应该不会反生，不过由于它们在过去确实出现过，因此这里将会介绍当你怀疑是这种情况的时候应该做什么。作为示例，我将会用来自 bug 报告上的一个测试用例来创造一个死锁。该 bug 与排除元数据锁（MDL）中的故障无关，因此我将仅关注调试本身，而不介绍产生死锁的行为。

提示

看起来这似乎是通过人为构造的手段去解释 MySQL 的 bug 导致的问题，而不是来自用户的错误，不过该消息还是很有用的。你不能保证不再遇到 bug，因此做好准备是必要的。所谓有备无患。

“资源”死锁的典型特征与行锁导致的死锁的特征一致：就是查询被挂起。没有内部机制可以发现并终止这类死锁，因此不要指望线程会超时（像 InnoDB 锁那样）或者会立即回滚（像 InnoDB 死锁那样）。SHOW PROCESSLIST 命令会输出如下信息。

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 2
   User: root
   Host: localhost
     db: performance_schema
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
*************************** 2. row ***************************
     Id: 6
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 9764
  State: Waiting for table metadata lock
   Info: SELECT * FROM t1, t2
*************************** 3. row ***************************
     Id: 7
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 9765
  State: Waiting for table metadata lock
   Info: RENAME TABLE t2 TO t0, t4 TO t2, t0 TO t4
*************************** 4. row ***************************
     Id: 8
```

```
    User: root
    Host: localhost
      db: test
 Command: Query
    Time: 9766
   State: Waiting for table level lock
    Info: INSERT INTO t3 VALUES ((SELECT count(*) FROM t4))
*************************** 5. row ***************************
      Id: 10
    User: root
    Host: localhost
      db: test
 Command: Sleep
    Time: 9768
   State:
    Info: NULL
*************************** 6. row ***************************
      Id: 502
    User: root
    Host: localhost
      db: test
 Command: Sleep
    Time: 2
   State:
    Info: NULL
6 rows in set (0.00 sec)
```

该输出表明有很多查询等待不同类型的锁超过了 9000 秒。

SHOW ENGINE INNODB STATUS 命令的 TRANSACTIONS 部分也没有给出任何新的信息：

```
------------
TRANSACTIONS
------------
Trx id counter 4211
Purge done for trx's n:o < 4211 undo n:o < 0
History list length 127
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started, OS thread id 35934208
MySQL thread id 502, query id 124 localhost root
show engine innodb status
---TRANSACTION 0, not started, OS thread id 33726976
MySQL thread id 6, query id 71 localhost root Waiting for table metadata lock
select * from t1, t2
---TRANSACTION 0, not started, OS thread id 35786240
mysql tables in use 2, locked 2
MySQL thread id 8, query id 69 localhost root Waiting for table level lock
insert into t3 values ((select count(*) from t4))
---TRANSACTION 4201, not started, OS thread id 35354624
mysql tables in use 2, locked 4
MySQL thread id 10, query id 68 localhost root
---TRANSACTION 0, not started, OS thread id 35633152
MySQL thread id 7, query id 70 localhost root Waiting for table metadata lock
rename table t2 to t0, t4 to t2, t0 to t4
```

从 5.5 版本开始，可以通过 performance_schema 来获取额外的信息，这将在 2.8.4 节进

行介绍。这里想介绍当我刚刚提到的问题出现时该怎么做。

首先应该检查的表是 MUTEX_INSTANCE，该表会列出自服务器启动以来所有的冲突。其中一部分现在没有使用，所以你应该在 SELECT 查询中略过它们，仅获取 LOCKED_BY_THREAD_ID 不为空的那些记录。

```
mysql> SELECT * FROM MUTEX_INSTANCES WHERE LOCKED_BY_THREAD_ID is
not null\G
*************************** 1. row ***************************
                NAME: wait/synch/mutex/sql/MDL_wait::LOCK_wait_status
OBJECT_INSTANCE_BEGIN: 35623528
  LOCKED_BY_THREAD_ID: 23
*************************** 2. row ***************************
                NAME: wait/synch/mutex/sql/MDL_wait::LOCK_wait_status
OBJECT_INSTANCE_BEGIN: 35036264
  LOCKED_BY_THREAD_ID: 22
*************************** 3. row ***************************
                NAME: wait/synch/mutex/mysys/THR_LOCK::mutex
OBJECT_INSTANCE_BEGIN: 508708108
  LOCKED_BY_THREAD_ID: 24
3 rows in set (0.26 sec)
```

要找出谁在等待这些冲突，可以查询 EVENTS_WAITS_CURRENT 表：

```
mysql> SELECT THREAD_ID, EVENT_ID, EVENT_NAME, SOURCE,
TIMER_START, OBJECT_INSTANCE_BEGIN, OPERATION FROM EVENTS_WAITS_CURRENT WHERE
THREAD_ID IN(SELECT LOCKED_BY_THREAD_ID FROM MUTEX_INSTANCES WHERE
LOCKED_BY_THREAD_ID IS NOT NULL)\G
*************************** 1. row ***************************
         THREAD_ID: 24
          EVENT_ID: 268
        EVENT_NAME: wait/synch/cond/mysys/my_thread_var::suspend
            SOURCE: thr_lock.c:461
       TIMER_START: 128382107931720
OBJECT_INSTANCE_BEGIN: 508721156
         OPERATION: timed_wait
*************************** 2. row ***************************
         THREAD_ID: 22
          EVENT_ID: 44
        EVENT_NAME: wait/synch/cond/sql/MDL_context::COND_wait_status
            SOURCE: mdl.cc:995
       TIMER_START: 130306657228800
OBJECT_INSTANCE_BEGIN: 35036372
         OPERATION: timed_wait
*************************** 3. row ***************************
         THREAD_ID: 23
          EVENT_ID: 42430
        EVENT_NAME: wait/synch/cond/sql/MDL_context::COND_wait_status
            SOURCE: mdl.cc:995
       TIMER_START: 7865906646714888
OBJECT_INSTANCE_BEGIN: 35623636
         OPERATION: timed_wait
3 rows in set (2.23 sec)
```

输出中的 THREAD_ID，是内部 mysqld 分配给线程的实际编号，而不是帮助找到产生

死锁原因的连接线程的编号。为了找到连接线程的编号，可以查询 THREADS 表。这里没有过滤输出，因为我想给你展示在 mysqld 进程内，当给该示例提供 6 个连接服务时所有运行的线程。

```
mysql> SELECT * FROM THREADS\G
*************************** 1. row ***************************
THREAD_ID: 0
       ID: 0
     NAME: thread/sql/main
*************************** 2. row ***************************
THREAD_ID: 24
       ID: 8
     NAME: thread/sql/one_connection
*************************** 3. row ***************************
THREAD_ID: 2
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 4. row ***************************
THREAD_ID: 14
       ID: 0
     NAME: thread/innodb/srv_monitor_thread
*************************** 5. row ***************************
THREAD_ID: 6
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 6. row ***************************
THREAD_ID: 518
       ID: 502
     NAME: thread/sql/one_connection
*************************** 7. row ***************************
THREAD_ID: 12
       ID: 0
     NAME: thread/innodb/srv_lock_timeout_thread
*************************** 8. row ***************************
THREAD_ID: 22
       ID: 6
     NAME: thread/sql/one_connection
*************************** 9. row ***************************
THREAD_ID: 7
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 10. row ***************************
THREAD_ID: 3
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 11. row ***************************
THREAD_ID: 26
       ID: 10
     NAME: thread/sql/one_connection
*************************** 12. row ***************************
THREAD_ID: 9
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 13. row ***************************
THREAD_ID: 16
```

```
       ID: 0
     NAME: thread/sql/signal_handler
*************************** 14. row ***************************
THREAD_ID: 23
       ID: 7
     NAME: thread/sql/one_connection
*************************** 15. row ***************************
THREAD_ID: 1
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 16. row ***************************
THREAD_ID: 4
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 17. row ***************************
THREAD_ID: 5
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 18. row ***************************
THREAD_ID: 8
       ID: 0
     NAME: thread/innodb/io_handler_thread
*************************** 19. row ***************************
THREAD_ID: 15
       ID: 0
     NAME: thread/innodb/srv_master_thread
*************************** 20. row ***************************
THREAD_ID: 18
       ID: 2
     NAME: thread/sql/one_connection
*************************** 21. row ***************************
THREAD_ID: 13
       ID: 0
     NAME: thread/innodb/srv_error_monitor_thread
*************************** 22. row ***************************
THREAD_ID: 10
       ID: 0
     NAME: thread/innodb/io_handler_thread
22 rows in set (0.03 sec)
```

既然我们在没有借助任何其他操作的情况下（例如，如给正在运行的 mysqld 进程挂接调试器），获取了服务器内所有运行状况的信息。在这个示例中，我们实际在上第一次运行 SHOW PROCESSLIST 命令后就获取了我们需要的所有信息。不过，我想展示当跟当前示例中一样状态信息不是很明确时，通过 performance_schema 表可以给调试内部死锁等问题带来何种帮助。

解决死锁问题应该做什么？你需要选择一个连接作为牺牲品，然后终止它。在这种情况下，幸运的话，我们会通过 SHOW ENGINE INNODB STATUS 命令发现一个相对不重要的连接。下面是关于 MySQL 服务器的线程 10 的描述：

```
---TRANSACTION 4201, not started, OS thread id 35354624
mysql tables in use 2, locked 4
MySQL thread id 10, query id 68 localhost root
```

该线程没有等待任何操作，却锁住了两个表。下面终止它：

```
mysql> KILL 10;
Query OK, 0 rows affected (0.09 sec)
```

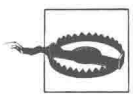
我们很幸运，通过仅终止这个单独的连接解决了问题：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 2
   User: root
   Host: localhost
     db: performance_schema
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
*************************** 2. row ***************************
     Id: 6
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 10361
  State:
   Info: NULL
*************************** 3. row ***************************
     Id: 7
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 10362
  State:
   Info: NULL
*************************** 4. row ***************************
     Id: 8
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 10363
  State:
   Info: NULL
*************************** 5. row ***************************
     Id: 502
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 152
  State:
   Info: NULL
5 rows in set (0.00 sec)

mysql> SELECT * FROM MUTEX_INSTANCES WHERE LOCKED_BY_THREAD_ID IS
NOT NULL\G
```

```
Empty set (0.11 sec)

mysql> SELECT THREAD_ID, EVENT_ID, EVENT_NAME, SOURCE,
TIMER_START, OBJECT_INSTANCE_BEGIN, OPERATION FROM EVENTS_WAITS_CURRENT WHERE
THREAD_ID IN(SELECT LOCKED_BY_THREAD_ID FROM MUTEX_INSTANCES WHERE
LOCKED_BY_THREAD_ID IS NOT NULL)\G
Empty set (1.23 sec)
```

以上是关于系统死锁的简介以及当你遭遇死锁的时候该做什么。基本的程序就是确认有线程确实在一直等待，然后找到一个不是必需的线程并终止它。你不用担心终止一个缓慢的线程，因为终止它们然后手动修复任何造成的错误，比无限期地等待直到mysqld 由于其他原因停止要好得多。

你可能会在应用程序中遭遇其他影响服务器内部锁的情况。有一些甚至会导致服务崩溃。例如，InnoDB 出于各种锁定目的使用一些信号标志，如为了保护 CHECK TABLE 和 OPTIMIZE TABLE，但是如果 InnoDB 信号量等待超过 600 秒，InnoDB 会故意使服务器崩溃。



警告

不要把信号量的长等待时间和持续多于 600 秒的用户会话混为一谈。InnoDB 的信号量用于保护特定的操作。而单个会话可以完全不影响或者影响其中的大部分。

可以监控应用程序的状态从而避免这种情况。因此，存储引擎在其监控器中也会输出关于 InnoDB 的信号量的等待时间的信息：

```
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 179, signal count 177
--Thread 35471872 has waited at trx/trx0rec.c line 1253 for 0.00 seconds the semaphore:
X-lock (wait_ex) on RW-latch at 0x149b124c created in file buf/buf0buf.c line 898
a writer (thread id 35471872) has reserved it in mode  wait exclusive
number of readers 1, waiters flag 0, lock_word: ffffffff
Last time read locked in file buf/buf0flu.c line 1186
Last time write locked in file trx/trx0rec.c line 1253
Mutex spin waits 209, rounds 3599, OS waits 38
RW-shared spins 70, rounds 2431, OS waits 67
RW-excl spins 0, rounds 2190, OS waits 71
Spin rounds per wait: 17.22 mutex, 34.73 RW-shared, 2190.00 RW-excl
```

提示

不用担心 InnoDB 输出需要等待操作的信号量需要花费很长的时间，如在一个大表上的 CHECK TABLE 操作。你需要解决的是在常规操作中长时间等待的情形。

2.7 复制和并发

并发问题的另一个重要场景是在复制环境下。

当排查复制问题的时候，需要记住主服务器总是多线程的，而从服务器在单个线程中执行所有更新[1]。这会影响复制时的性能和一致性，而与你使用的二进制日志格式和选项无关。

与复制相关的问题主要为数据不一致性，也就是从服务器上的数据与主服务器上的数据不同。在大多数情况下，MySQL 复制会小心处理数据一致性问题，不过你仍有时候会遭遇一致性问题，特别是在你使用了基于语句的复制的时候。本节重点关注一致性问题，在最后会简要介绍复制是如何影响性能的。

2.7.1 基于语句的复制问题

从 5.1 版开始，MySQL 开始支持三种二进制日志格式：语句、行和混合日志。数据的不一致性问题大多由基于语句的复制（Statement-Basement Replication，-SBR）引起，并且其使用是语句的二进制日志格式。这种格式与基于行的日志相比有很多优势，并且是历史上唯一支持日志变化的格式，因此其用户群非常庞大而且目前仍然是默认配置。然而，它比基于行的日志格式有更多的风险。

提示

基于行的日志在日志中记录原始数据。因此从服务器不会与主服务器执行相同的查询，而是直接更新表中的行数据。这是最安全的日志格式，因为它不可能在从服务器中插入任何主服务器上不存在的数据。该日志类型甚至可以在调用如 NOW()等不确定的函数的时候保证安全。

基于语句的日志是在日志中记录原始的 SQL 语句，因此从服务器与主服务器执行相同的命令。如果你使用该格式，网络流量通常（虽然不总是如此）比基于行的日志低，因为查询占用的数据会比实际插入的数据小。这在批量更新表中的 BLOB 列的时候尤为显著。该格式的缺点就是需要保证数据的一致性。例如，如果向列中插入了 NOW()函数的返回结果，你很可能会在主从服务器中保存不同的数据。

混合二进制日志综合了基于行和语句的日志的优点：它用语句格式保存大多数执查询，并且当查询不安全的时候改用基于行的格式，即，当你使用了如 NOW()等不确定函数的时候。

1：如 2.7.1 节讨论的那样，这种情况将会改变。

当你的应用程序计划使用 MySQL 复制的时候，需要注意不同的语句对一致性的影响。甚至当主服务器给从服务器提供很多额外信息的时候，后者可能无法处理所有问题。

MySQL 参考手册包含很多在使用基于语句的复制时不安全的语句列表，也就是说使，用这些语句会使主从服务器间结果不同。Charles Bellet 等（O'Reilly）编写的《高可用 MySQL》一书中详细介绍了复制时产生的一致性问题。这里重点介绍与并发相关的问题，如果语句在主服务器的很多线程中执行并且按照特定的顺序在从服务器上执行，则该并发问题可能会导致不同的结果。

由于主服务器是多线程的，那么它会按照一个不确定的顺序执行多个连接请求。但是在从服务器上，目前还是单线程的，那么按照日志记录的顺序重新执行语句就有可能与之前在主服务器上执行的顺序不同。这就有可能导致主从服务器的不同。

多线程从服务器

复制团队现在正在开发多线程的从服务器。该增强特性目前为预览版。多线程从服务器可以将事务分发到不同的线程中。可以通过 slave_parallel_workers 变量调整使用的线程数量。

该特性会影响复制的的扩展性，也就是数据的更新速度，因此它会避免从服务器滞后过多而使一致性受到威胁。不过不要指望它会彻底解决一致性问题。

你可以在 Luis Soares 的博客中查到关于多线程从服务器的更多信息。

为了举例说明顺序问题，我将使用只有一个字段且没有唯一索引的简单表。在实际项目中，类似的情况即发生在当你查询搜索没有使用的唯一索引的行的时候。

例 2.1　由于错误的事务顺序引发的复制问题的示例

```
CREATE TABLE t1(f1 CHAR(2)) ENGINE=InnoDB;
```

现在我要模拟并发事务在批量插入行时的情况：

```
master1> BEGIN;
master1> INSERT INTO t1 VALUES(1);
master1> INSERT INTO t1 VALUES(2);
master1> INSERT INTO t1 VALUES(3);
master1> INSERT INTO t1 VALUES(4);
master1> INSERT INTO t1 VALUES(5);
```

注意，该事务目前还没提交。接下来，我将在另外一个连接中打开一个新事务，然后插入另一些行：

```
master2> BEGIN;
master2> INSERT INTO t1 VALUES('a');
master2> INSERT INTO t1 VALUES('b');
master2> INSERT INTO t1 VALUES('c');
master2> INSERT INTO t1 VALUES('d');
```

```
master2> INSERT INTO t1 VALUES('e');
master2> COMMIT;
```

第二个事务已经提交了。现在我将提交第一个事务：

```
master1> COMMIT;
```

你可能预期表中将会有以 1～5 开头的数据，然后是 a～e。这绝对是正确的，因为第一个事务于第二个事务开始：

```
master1> SELECT * FROM t1;
+------+
| f1   |
+------+
| 1    |
| 2    |
| 3    |
| 4    |
| 5    |
| a    |
| b    |
| c    |
| d    |
| e    |
+------+
10 rows in set (0.04 sec)
```

但是在事务提交前，主服务器没有向二进制日志中写入任务数据。因此，从服务器中的数据将以 a～e 开始，而 1～5 在第二个集合中：

```
slave> SELECT * FROM t1;
+------+
| f1   |
+------+
| a    |
| b    |
| c    |
| d    |
| e    |
| 1    |
| 2    |
| 3    |
| 4    |
| 5    |
+------+
10 rows in set (0.04 sec)
```

目前为止还算正常：尽管顺序不同，不过主从服务器仍包含相同的数据。但是当执行不安全的 UPDATE 时，就会变得不正常了：

```
master1> UPDATE t1 SET f1='A' LIMIT 5;
Query OK, 5 rows affected, 1 warning (0.14 sec)
Rows matched: 5  Changed: 5  Warnings: 1

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
```

```
   Code: 1592
Message: Unsafe statement written to the binary log using statement format since
BINLOG_FORMAT = STATEMENT. The statement is unsafe because it uses a LIMIT
clause. This is unsafe because the set of rows included cannot be predicted.
1 row in set (0.00 sec)
```

如你所见，服务器警告我们查询是不安全的。我们来看看原因。在主服务器上，结尾是：

```
master1> SELECT * FROM t1;
+------+
| f1   |
+------+
| A    |
| A    |
| A    |
| A    |
| A    |
| a    |
| b    |
| c    |
| d    |
| e    |
+------+
```

然而，在从服务器上的数据集是完全不同的：

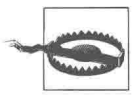
```
slave> SELECT * FROM t1;
+------+
| f1   |
+------+
| A    |
| A    |
| A    |
| A    |
| A    |
| 1    |
| 2    |
| 3    |
| 4    |
| 5    |
+------+
```

这个示例非常简单。在现实生活中，混乱的情况通常复杂得多。在这个示例中，我使用了事务性存储引擎并且使用多语句事务来简单地重现一个错误结果。在非事务的存储引擎中，当你使用的 MySQL 扩展功能延迟了实际数据变化的时候，你可能会看到类似的问题，例如：INSERT DELAYED。

解决这种问题最好的方式就是使用基于行的复制或者混合复制。如果你坚持使用基于语句的复制，那么绝对有必要好好设计应用程序来避免这种情况。请时刻牢记，每个事务都只会在其提交的时候向二进制日志中写入数据。你也需要使用第 1 章介绍的技术去检查警告。如果在生产环境中对每个查询都执行该检查听起来很困难（例如，出于性能考虑），那么至少在开发阶段执行该检查。

2.7.2 混合事务和非事务表

一件很重要的事情就是不要把事务和表非事务的表混合在同一个事务中[1]。一旦修改非事务表，就不能回滚它们，因此如果事务中止或者回滚，数据会变得不一致。



警告

当混合事务表在同一个事务中使用不同的引擎时会产生同样的情况。如果引擎有不同的事务隔离等级或者对会引起隐式提交的语句有不同的规则，你会遭遇与刚才介绍的混合使用有事务表和无事务表相类似的问题。

甚至当你使用单独的 MySQL 服器务的时候也有可能发生这个问题，但是在复制的环境中有时候事情会变得更糟。所以我将本节内容也包含在复制相关的章节里，尽管它也包含无复制环境的内容。

对该示例来说，我们将展示一个使用与 1.5 节中的示例相同概念的存储过程。这里把临时表改为持久表并且添加另外一个存储过程来填充 t2 表中的行。t2 表也将使用 MyISAM 存储引擎。MyISAM 是 5.5 版本之前的默认存储引擎，因此如果用户忘记在 CREATE TABLE 语句中设置 ENGINE 选项，那么这样的临时表很容易创造。用户也有可能想提高性能而错误地使用了 MyISAM 引擎。

```
CREATE TABLE t1(f1 INT) ENGINE=InnoDB;
CREATE TABLE t2(f1 INT) ENGINE=MyISAM;
CREATE TABLE t3(f1 INT) ENGINE=InnoDB;

INSERT INTO t3 VALUES(1),(2),(3);

CREATE PROCEDURE p1()
BEGIN
DECLARE m INT UNSIGNED DEFAULT NULL;
SELECT max(f1) INTO m FROM t2;
IF m IS NOT NULL
THEN
  INSERT INTO t1(f1) SELECT f1 FROM t2;
END IF;
END
|
```

现在我将在事务中向 t2 表中插入值：

```
master>  BEGIN;
Query OK, 0 rows affected (0.00 sec)

master>  INSERT INTO t1 VALUES(5);
Query OK, 1 row affected (0.00 sec)
```

1：该规则的例外是称为“半安全”的异常，如一个非事务表是只读或是只写（例如，日志表在这种上下文中是只写的）的时候。当你与事务表一起使用该类表的时候，需要小心。

```
master> INSERT INTO t2 SELECT f1 FROM t3;
Query OK, 3 rows affected, 1 warning (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 1
```

然后，我将在没有在并行的主连接中提交该事务的情况下，调用 p1()函数：

```
master> BEGIN;
Query OK, 0 rows affected (0.00 sec)

master> CALL p1();
Query OK, 3 rows affected (0.00 sec)

master> COMMIT;
Query OK, 0 rows affected (0.00 sec)

master> SELECT * FROM t1;
+------+
| f1   |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

如你所见，在主服务器上有三行。查看从服务器上有几行数据：

```
slave> SELECT * FROM t1;
Empty set (0.00 sec)
```

从服务器实际上没有数据。这是由于主服务器向事务缓存中写入对无事务表的更新。但是，如我们之前所见，缓存内容仅在事务实际上结束后才会写入二进制日志。在这种情况下，可能相比主服务器，从服务器实际上更符合用户的意图（因为主服务器在 t1 中拥有一个对应一个永远不会完成事务的“幽灵”条目），不过关键是我们由于非事务的表而最终造成了数据不一致性。

提示

3.9.1 节讨论 binlog_direct_non_transactional_updates 选项，该选项控制何时对非事务表的更新会写入二进制日志。

- 因此，不要把事务表和非事务表混合在一个事务中。如果确实有必要这么做，那么使用另外的锁定方法，如 LDCK TABLE，以保证在回滚或崩溃时的一致性。

可以使用基于行的日表或者混合二进制日志解决大多数与复制相关的问题，不过如果主服务器上的数据都不是你想要的那么这些都没有用了。

2.7.3 从服务器上的问题

我们刚才仅讨论了由于主服务器是多线程的而从服务器是单线程的引发的并发问题。如果仅从服务器上的 SQL 线程在运行，那么在从服务器上不会有任何额外的并发问题。

不过在现实生活中，除从主服务器复制外，从服务器还完成其他的任务，因此从服务器上的 SQL 线程所遇到并发问题就跟其他连接线程遇到的一样。

当从服务器上的 SQL 线程向正在被其他线程使用的表中写入数据的时候，跟其他多线程场景一样，它需要获取这些表的所有锁。可以通过执行 SHOW PROCESSLIST 命令，看到这些信息：

```
slave> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 115
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
   Host:
     db: test
Command: Connect
   Time: 16
  State: update
   Info: INSERT INTO t1 VALUES(3)
*************************** 3. row ***************************
     Id: 3
   User: msandbox
   Host: localhost
     db: test
Command: Sleep
   Time: 28
  State:
   Info: NULL
*************************** 4. row ***************************
     Id: 4
   User: msandbox
   Host: localhost
     db: test
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
4 rows in set (0.00 sec)
```

从服务器上的 SQL 线程在第 2 行列出并且是由系统用户调用的，系统用户是一个执行从服务器上查询的特殊用户。现在，在启动从服务器之前，我在并行连接中执行下面的查询以展现 SQL 线程是如何等待其他线程完成后才可以执行更新的：

```
SELECT * FROM t1 FOR UPDATE;
```

在得到 SHOW PROCESSLIST 命令的输出后，我回滚了并行查询，因此该 SQL 线程可以成功地完成查询：

```
slave> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 267
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 168
  State: Slave has read all relay log; waiting for the slave I/O thread to update it
   Info: NULL
...
```

你也可能遭遇到从服务器上的 SQL 线程持有你的应用程序想要获取的行锁的情况。如果想要弄清何时发生了这样的情况，可以检查 SHOW PROCESSLIST 和 SHOW ENGINE INNODB STATUS 的输出。

这两种情况的统一解决方案就是要么等待活动的线程结束，要么在用户事务等待过久的情况下回滚事务。

2.8 高效地使用 MySQL 问题排查工具

为了结束本章的内容，本节打算再介绍一下我们使用过的工具，并补充介绍一下我们之前忽略的其他有用的特性。

2.8.1 SHOW PROCESSLIST 和 INFORMATION_SCHEMA.PROCESSLIST 表

SHOW PROCESSLIST 命令是当你怀疑遇到并发问题时的首选工具。它虽然不会给出多语句事务中语句间的关系，但是它会暴露出问题的特征以确认需要对并发进行更多调查。一个最主要的特征就是线程长时间处于“Sleep”状态。

本章的示例中使用的是 SHOW PROCESSLIST 的精简版本，该版本会删减长查询。SHOW FULL PROCESSLIST 命令展现的是完整的查询，如果你有长查询且很难从查询的开头推测出其完整版本，该命令会比较方便。

从 5.1 版本开始，MySQL 提供了 INFORMATION_SCHEMA.PROCESSLIST 表，该表中的数据与 SHOW FULL PROCESSLIST 的输出相同。在繁忙的服务器中，该表极大地

方便了排错，因为你可以使用 SQL 来缩小希望查看的输出范围：

```
slave2> SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST\G
*************************** 1. row ***************************
     ID: 5
   USER: msandbox
   HOST: localhost
     DB: information_schema
COMMAND: Query
   TIME: 0
  STATE: executing
   INFO: SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST
*************************** 2. row ***************************
     ID: 4
   USER: msandbox
   HOST: localhost
     DB: test
COMMAND: Sleep
   TIME: 583
  STATE:
   INFO: NULL
*************************** 3. row ***************************
     ID: 2
   USER: system user
   HOST:
     DB: NULL
COMMAND: Connect
   TIME: 940
  STATE: Slave has read all relay log; waiting for the slave I/O thread t
   INFO: NULL
*************************** 4. row ***************************
     ID: 1
   USER: system user
   HOST:
     DB: NULL
COMMAND: Connect
   TIME: 1936
  STATE: Waiting for master to send event
   INFO: NULL
4 rows in set (0.00 sec)
```

那么为什么我在大多数例子中使用 SHOW PROCESSLIST 命令而不是 PROCESSLIST 表呢？首先，SHOW PROCESSLIST 命令在所有 MySQL 版本中都支持。当我进行支持工作时，我可以从客户直接申请这些信息而不用事先确认其使用的 MySQL 的版本。另一个原因也是由于我的支持工作：当为客户工作的时候，我们不清楚其环境的所有详细信息，因此查看未经过过滤的进程列表可以给我们提供一些信息以进行深入了解。

当你调试自己的应用程序时就有不同的考虑了。因为你已经知道哪个进程是重要的，所以你可以在查询中通过 WHERE 条件限制输出，例如：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST WHERE TIME > 50
mysql> SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST WHERE INFO LIKE 'my query%'
```

这样做可以节约分析结果的时间。

2.8.2 SHOW ENGINE INNODB STATUS 和 InnoDB 监控器

这些工具展示当你使用 InnnoDB 表时最重要的信息。可以通过 SHOW ENGINE INNODB STATUS 命令和创建 InnoDB 监控表获取相同的信息。这些表不是为用户准备的，而是一种告诉 InnoDB 引擎每隔若干秒向错误日志中写入 InnoDB 的状态信息的方式。

这里将讨论标准监控器和锁监控器。InnoDB 还提供了表空间监控器和表监控器，这两个监控器会分别输出共享表空间和 InnoDB 内部字典中的信息。表空间监控器和表的监控器与并发问题没有直接关系，因此这里略过它们。

标准的 InnoDB 监控器就是执行 SHOW ENGINE INNODB STATUS 命令时获得的信息。与并发相关的，我们关心的是 SEMAPHORES、LATEST DETECTED DEADLOCK 和 TRANSACTIONS。

SEMAPHORES 部分包含了线程等待互斥锁或读写锁的信息。这里需要注意的是等待线程的数量或者长时间等待的线程。不过，长时间等待并不是问题的一个必要特征。例如，在一个超大的表上执行 CHECK TABLE 命令可能会长时间持有信号量。但是如果你发现普通操作持续很长时间，你应该检查一下你安装的版本是否可以处理你拥有的 InnoDB 线程数。降低 innodb_thread_concurrency 的值可以起到作用。

LATEST DETECTED DEADLOCK 部分包含最近检测到的死锁的信息。如果从服务器启动没有发现死锁，那么该部分就是空的。可以通过监控该部分内容确认应用程序中是否有死锁。

当发现有死锁的时候，要么从你的应用程序中移除查询，这样就没有冲突导致死锁，要么添加代码小心地处理死锁。

TRANSACTIONS 部分包含地所有当前正在执行的事务的信息。对于本章的讨论来说，特别需要注意的是，该部分会列出所有活动事务正在等待的所有锁的信息。如果打开 InnoDB 锁监控器，那么该部分还会包含事务持有哪个锁的信息。这对调试锁等待来说是非常有用的信息。

为了说明 InnoDB 锁监控器怎样帮助你调试锁问题，回顾一下 2.3.1 节中的示例。如果打开 InnoDB 锁监控器并且执行同样的查询，我们将会在 SHOW ENGINE INNODB STATUS 的输出中看到一些额外的信息：

```
mysql> SHOW ENGINE INNODB STATUS \G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
```

```
110809 14:03:45 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 6 seconds
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 12, signal count 12
Mutex spin waits 0, rounds 209, OS waits 7
RW-shared spins 10, OS waits 5; RW-excl spins 0, OS waits 0
------------
TRANSACTIONS
------------
Trx id counter 0 26244358
Purge done for trx's n:o < 0 26244356 undo n:o < 0 0
History list length 4
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, OS thread id 101493760
MySQL thread id 219, query id 96 localhost root
show engine innodb status
---TRANSACTION 0 26244357, ACTIVE 1 sec, OS thread id 101357568 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 217, query id 95 localhost root Updating
update t set a=36 where a=6
------- TRX HAS BEEN WAITING 1 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244357 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
 len 7; hex 0000000032081c; asc     2  ;;

------------------
TABLE LOCK table `test`.`t` trx id 0 26244357 lock mode IX
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244357 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
 len 7; hex 0000000032081c; asc     2  ;;

---TRANSACTION 0 26244356, ACTIVE 6 sec, OS thread id 101099008
2 lock struct(s), heap size 320, 1 row lock(s), undo log entries 2
MySQL thread id 184, query id 93 localhost root
TABLE LOCK table `test`.`t` trx id 0 26244356 lock mode IX
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244356 lock_mode X locks rec but not gap
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
len 7; hex 0000000032081c; asc     2  ;;
```

将该输出与你之前看到的没有锁信息的输出进行比较。最重要的区别在于，现在你拥有持有锁的事务的信息：

```
---TRANSACTION 0 26244356, ACTIVE 6 sec, OS thread id 101099008
2 lock struct(s), heap size 320, 1 row lock(s), undo log entries 2
MySQL thread id 184, query id 93 localhost root
TABLE LOCK table `test`.`t` trx id 0 26244356 lock mode IX
```

```
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244356 lock_mode X locks rec but not gap
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
 len 7; hex 0000000032081c; asc     2  ;;
```

我在整个事务信息中识别出关于锁“Record lock, heap no 6”的信息。我们关注以下信息：

```
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
 len 7; hex 0000000032081c; asc     2  ;;
```

这是关于锁记录的物理内容。当你检查正在等待的事务的时候，你可以看到它会等待同一个锁（注意 PHYSICAL RECORD）：

```
update t set a=36 where a=6
------- TRX HAS BEEN WAITING 1 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244357 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907504; asc     u ;; 2:
 len 7; hex 0000000032081c; asc     2  ;;
```

这是非常有用的信息，因为你可以清楚地将正在等待锁的事务与持有锁的事务联系起来。

之前讨论了 InnoDB INFORMATION_SCHEMA 表。为什么使用 InnoDB 监控器而不是直接查看这些表呢？原因就是 INFORMATION_SCHEMA 表仅包含当前信息，而 InnoDB 监控器可以将信息转储到错误日志文件中，这样你可以在稍后分析它。在你想要找到在应用运行程序的时候都发生了什么的时候，这些信息尤为重要。

关于 InnoDB 监控器的输出还有其他有用的部分。当你怀疑一个不确定的死锁或仅是一个长时间的锁的时候，查看 FILE I/O、INSERT BUFFER AND ADAPTIVE HASH INDEX、BUFFER POOL AND MEMORY 以及 ROW OPERATIONS 等信息是很有意义的。不论何时，当读或写操作停止并且线程互相等待的时候，这很有可能是一个锁定问题，哪怕锁住的线程处于 Updating 状态。

2.8.3 INFORMATION_SCHEMA 中的表

从 5.1 版本的 InnoDB 插件开始，MySQL 开始支持新的 InnoDB INFORMATION_SCHEMA 表。它们与并发问题相关的部分是：

INNODB_TRX

包含当前运行的所有事务的列表。

INNODB_LOCKS

包含事务持有的当前锁的相关信息以及每个事务等待的锁的信息。

INNODB_LOCK_WAITS

包含事务正在等待的锁的信息。

这些表易于使用并且可以很快提供有关事务状态和锁的信息。你在本章前面看到的示例解释了我关于这个主题需要说明的一切。

调试并发问题时有用的典型的 INFORMATION_SCHEMA 表查询

关于事务正在等待的所有锁的信息：

```
SELECT * FROM INNODB_LOCK_WAITS
```

阻塞的事务列表：

```
SELECT * FROM INNODB_LOCKS WHERE LOCK_TRX_ID IN
  (SELECT BLOCKING_TRX_ID FROM INNODB_LOCK_WAITS)
```

或

```
SELECT INNODB_LOCKS.* FROM INNODB_LOCKS JOIN INNODB_LOCK_WAITS
  ON (INNODB_LOCKS.LOCK_TRX_ID = INNODB_LOCK_WAITS.BLOCKING_TRX_ID)
```

特定表上的锁的列表：

```
SELECT * FROM INNODB_LOCKS WHERE LOCK_TABLE = 'db_name.table_name'
```

等待锁的事务列表：

```
SELECT TRX_ID, TRX_REQUESTED_LOCK_ID, TRX_MYSQL_THREAD_ID, TRX_QUERY
FROM INNODB_TRX WHERE TRX_STATE = 'LOCK WAIT'
```

2.8.4 PERFORMANCE_SCHEMA 中的表

性能架构允许你从底层监控 MySQL 服务器的运行状态。该架构实现为包含基于 PERFORMANCE_SCHEMA 存储引擎的表的数据库。存储引擎通过服务器源代码中定义的“检测点”收集事件数据。PERFORMANCE_SCHEMA 中的表不使用持久性磁盘存储。

为了调试并发问题，可以使用 COND_INSTANCES、FILE_INSTANCES、MUTEX_INSTANCES 和 RWLOCK_INSTANCES 表以及各种 EVENT_WAITS_*表。THREADS 表有助于建立内部线程和 MySQL 的用户线程之间的关系。

所有的*INSTANCES 表都包含 NAME 和 OBJECT_INSTANCE_BEGIN 字段，这两个字段分别代表实例的名称和对象被检测时的内存地址。

COND_INSTANCES 表包含等待条件列表，这些条件是在服务器启动后生成的。条件（对于学习过并发的程序员来说这会是一个熟悉的术语）是指使一个线程等待其他线程的方式方法。

FILE_INSTANCES 表包含性能架构可见的文件列表。当服务器首次打开文件的时候就

将文件名插入该表，并且在文件从磁盘中删除之前都会保存在表中。目前打开文件会有一个正的 OPEN_COUNT 计数。Number 字段保存当前使用该文件的文件句柄的数量。

MUTEX_INSTANCES 表包含性能架构可见的互斥列表。互斥记录中 LOCKED_BY_THREAD_IS 的值为 NOT NULL 部分是当前锁定的互斥。

RWLOCK_INSTANCE 表包含所有读/写锁实例的列表。WRITE_LOCKED_BY_THREAD_IS 字段代表持有锁的线程 ID。READ_LOCKED_BY_COUNT 字段代表当前在实例上获取了多少读锁。

EVENTS_WAITS_* 系列表包含每个线程等待的事件的信息。

例如，要找出事务正在等待哪种类型的锁，可以使用下面的查询：

```
mysql> SELECT THREAD_ID, EVENT_NAME, SOURCE, OPERATION, PROCESSLIST_ID \
FROM  events_waits_current JOIN threads USING (THREAD_ID) WHERE PROCESSLIST_ID > 0\G
*************************** 1. row ***************************
     THREAD_ID: 36
    EVENT_NAME: wait/synch/mutex/mysys/THR_LOCK::mutex
        SOURCE: thr_lock.c:550
     OPERATION: lock
PROCESSLIST_ID: 20
*************************** 2. row ***************************
     THREAD_ID: 41
    EVENT_NAME: wait/synch/mutex/sql/THD::LOCK_thd_data
        SOURCE: sql_class.cc:3754
     OPERATION: lock
PROCESSLIST_ID: 25
*************************** 3. row ***************************
     THREAD_ID: 40
    EVENT_NAME: wait/synch/mutex/innodb/kernel_mutex
        SOURCE: srv0srv.c:1573
     OPERATION: lock
PROCESSLIST_ID: 24
3 rows in set (0.00 sec)
```

上述信息表明 24 号线程在等待 InnoDB 的 kernel_mutex，而通过 SHOW PROCESSLIST 发现，同一个查询正处于 Updating 状态：

```
mysql> SHOW PROCESSLIST \G
*************************** 1. row ***************************
     Id: 20
   User: root
   Host: localhost
     db: performance_schema
Command: Query
   Time: 0
  State: NULL
   Info: show processlist
*************************** 2. row ***************************
     Id: 24
   User: root
   Host: localhost
     db: sbtest
```

```
Command: Query
   Time: 3
  State: Updating
   Info: update example set f2=f2*2
*************************** 3. row ***************************
     Id: 25
   User: root
   Host: localhost
     db: sbtest
Command: Sleep
   Time: 228
  State:
   Info: NULL
3 rows in set (0.00 sec)
```

THREADS 表包含当前所有正在运行的线程列表。ID 是内部分配的，与连接线程的 ID 完全不同。进一步来说，服务器运行了很多与连接线程无关的内部线程。（例如，从服务器运行一个 SQL 线程和 I/O 线程。）该表包含的 PROCESSLIST_ID 字段记录与每个特定线程相关联的连接线程 ID，前提是如果存在的话。

*_SUMMARY_*表包含被终止事件的聚合信息。

举例来说，要找出哪个表使用得最多，可以尝试下面的查询。该查询对于分别使用独立的文件保存表数据的存储引擎有效，如 MyISAM 引擎或者开启了 innodb_file_per_table 选项的 InnoDB 存储引擎。

```
mysql> SELECT * FROM file_summary_by_instance WHERE file_name \
LIKE CONCAT(@@datadir,'sbtest/%') ORDER BY SUM_NUMBER_OF_BYTES_WRITE DESC, \
SUM_NUMBER_OF_BYTES_READ DESC \G
*************************** 1. row ***************************
               FILE_NAME: /home/ssmirnov/mysql-5.5/data/sbtest/example.ibd
              EVENT_NAME: wait/io/file/innodb/innodb_data_file
              COUNT_READ: 0
             COUNT_WRITE: 8
 SUM_NUMBER_OF_BYTES_READ: 0
SUM_NUMBER_OF_BYTES_WRITE: 196608
*************************** 2. row ***************************
               FILE_NAME: /home/ssmirnov/mysql-5.5/data/sbtest/example.frm
              EVENT_NAME: wait/io/file/sql/FRM
              COUNT_READ: 14
             COUNT_WRITE: 17
 SUM_NUMBER_OF_BYTES_READ: 948
SUM_NUMBER_OF_BYTES_WRITE: 4570
*************************** 3. row ***************************
               FILE_NAME: /home/ssmirnov/mysql-5.5/data/sbtest/sbtest.ibd
              EVENT_NAME: wait/io/file/innodb/innodb_data_file
              COUNT_READ: 5236
             COUNT_WRITE: 0
 SUM_NUMBER_OF_BYTES_READ: 85786624
SUM_NUMBER_OF_BYTES_WRITE: 0
*************************** 4. row ***************************
               FILE_NAME: /home/ssmirnov/mysql-5.5/data/sbtest/sbtest.frm
              EVENT_NAME: wait/io/file/sql/FRM
              COUNT_READ: 7
```

```
            COUNT_WRITE: 0
 SUM_NUMBER_OF_BYTES_READ: 1141
SUM_NUMBER_OF_BYTES_WRITE: 0
*************************** 5. row ***************************
             FILE_NAME: /home/ssmirnov/mysql-5.5/data/sbtest/db.opt
            EVENT_NAME: wait/io/file/sql/dbopt
            COUNT_READ: 0
           COUNT_WRITE: 0
 SUM_NUMBER_OF_BYTES_READ: 0
SUM_NUMBER_OF_BYTES_WRITE: 0
5 rows in set (0.00 sec)
```

2.8.5 日志文件

有两种 MySQL 服务器日志文件有助于处理并发问题：错误日志文件和通用查询日志文件。

错误日志文件包含错误的相关信息。它会包含不安全的复制语句、由于长信号量等待而产生的故意崩溃等相关信息。我在第 1 章已经建议过，错误日志文件是你遇到问题时第一个需要检查的地方。对于并发问题来说，建议相同。当你不确定问题的来源的时候，请先查看错误日志文件。

通用查询日志有助于找到通过其他途径找不到的查询。一个典型例子就是阻塞其他事务的多语句事务。如果这是由一个应用程序调用的，那么有时候很难判断哪个查询导致了该问题。在这种情况下，打开通用查询日志，等待问题重现，然后查询通用日志以找到由特定线程执行的查询。一种典型的排错方式就是查看 InnoDB 监控器的输出，找到可能阻塞其他事务的事务的 MySQL 线程 ID，然后执行以下查询。

```
SELECT argument, event_time FROM mysql.general_log WHERE thread_id =
THIS_THREAD_ID ORDER BY event_time
```

该查询将会返回锁定线程执行的查询列表。你会发现 BEGIN 或者 START TRANSACTION 语句启动了整个多语句事务。一旦你发现了不和谐的事务，你可以研究对应用程序做何改变以避免以后出现类似的锁定。

为了举例说明上面的观点，回顾一下 2.3.1 节中的示例。在 SHOW PROCESSLIST 的输出中，我们看到：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 184
   User: root
   Host: localhost
     db: test
Command: Sleep
   Time: 25
  State:
   Info: NULL
*************************** 2. row ***************************
     Id: 217
   User: root
```

```
     Host: localhost
       db: test
  Command: Query
     Time: 5
    State: Updating
     Info: UPDATE t SET a=36 WHERE a=6
*************************** 3. row ***************************
       Id: 219
     User: root
     Host: localhost
       db: mysql
  Command: Query
     Time: 0
    State: NULL
     Info: SHOW PROCESSLIST
3 rows in set (0.00 sec)
```

SHOW ENGINE INNODB 命令输出：

```
mysql> SHOW ENGINE INNODB STATUS \G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
110809 13:57:21 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 33 seconds
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 5, signal count 5
Mutex spin waits 0, rounds 80, OS waits 2
RW-shared spins 6, OS waits 3; RW-excl spins 0, OS waits 0
------------
TRANSACTIONS
------------
Trx id counter 0 26244354
Purge done for trx's n:o < 0 26243867 undo n:o < 0 0
History list length 3
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, OS thread id 101493760
MySQL thread id 219, query id 86 localhost root
SHOW ENGINE INNODB STATUS
---TRANSACTION 0 26244353, ACTIVE 119 sec, OS thread id 101357568 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320, 1 row lock(s)
MySQL thread id 217, query id 85 localhost root Updating
UPDATE t SET a=36 WHERE a=6
------- TRX HAS BEEN WAITING 1 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 349 page no 3 n bits 88 index `PRIMARY` of table `test`.`t`
trx id 0 26244353 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 3; compact format; info bits 32
 0: len 4; hex 00000006; asc     ;; 1: len 6; hex 000001907500; asc     u ;; 2:
 len 7; hex 00000000320762; asc     2 b;;
```

```
------------------
---TRANSACTION 0 26244352, ACTIVE 139 sec, OS thread id 101099008
2 lock struct(s), heap size 320, 1 row lock(s), undo log entries 2
MySQL thread id 184, query id 79 localhost root
 ...
```

阻塞的是 MySQL 217 号线程中的 26244353 号事务。当前唯一持有锁的事务是 MySQL 184 号线程中的 26244352 号事务。不过在查看通用日志文件之前，我们仍不是很清楚 184 号线程在做什么：

```
mysql> SELECT argument, event_time FROM mysql.general_log WHERE
thread_id=184 ORDER BY event_time;
+-----------------------------+---------------------+
| argument                    | event_time          |
+-----------------------------+---------------------+
| begin                       | 2011-08-09 13:55:58 |
| update t set a=26 where a=6 | 2011-08-09 13:56:09 |
+-----------------------------+---------------------+
2 rows in set (0.15 sec)
```

从该输出中，可以很容易地看到，与阻塞的事务一样，事务正在更新同一个表中的同一行记录。有了这些信息，我们可以重构应用程序。

第3章

配置选项对服务器的影响

MySQL 服务器提供大量选项，可以通过多种方式来对这些选项进行设置，例如：在 my.cnf 配置文件中进行设置，在使用命令行启动服务器的时候进行设置，或者在服务器正在运行的时候使用变量来对它们进行设置。大多数 MySQL 服务器变量都允许动态设置，并且在通常情况下，一个变量对应一个配置选项。

有些选项是 GLOBAL 类型的，一些选项只适用于某个特定的存储引擎，而另外的一些可以称为会话级别，适用于连接和某些特定的活动，比如复制。本章并不是 MySQL 服务器选项的通用指南，但它涉及一些可以创建或能够产生变化的选项，这能够帮助你解决 MySQL 服务器发生的一些问题。

在本章开始前，我们需要在某些方面达成一致。

我将使用变量和选项来表示服务器选项。MySQL 使用独立的语法选项及变量，例如：选项名称的拼写形式通常使用连字符（选项-名称），而对应的变量名称的拼写形式则使用下划线（变量_名称）。通常情况下，这两种拼写形式在 MySQL 服务器的配置文件中和命令行都支持，但变量仅仅支持使用“变量_名称”语法的形式。因此，在本书中，只要涉及的变量支持语法“变量_名称”这种拼写形式，我们就会使用这种拼写形式。

我们可以根据变量的用途来把它们分成多个不同的组：用来设置服务器的配置目录。限制对硬件资源的使用，改变 mysqld 应该如何应对一个或多个场景等。依照它们分配时间的不同，它们也可以分为不同的组，例如：当服务器启动的时候，一个线程连接创建的时候，或者当服务器启动一个特定操作的时候。

3.1 服务器选项

我使用服务器选项选项这个术语，是因为这一个词就能够解释其所有的功能，例如：向服务器指定目录或文件，提醒服务器是否打开一个特定的日志等诸如此类的功能。这些选项通常不会产生什么问题。我只发现过两个典型的故障排除情景是由这种选项导致的：当一个选项指向了一条错误路径或者打开一个特定功能的时候，或者在启动后，改变了 mysqld 命令的运行方式。如果问题发生于第二种情况下，那么你可能很难判断造成 MySQL 问题的根本原因，因为你根本无法知道这以前发生过什么改变。

当某选项使用错误路径的时候，你通常能够在服务器启动的时候注意此类问题。例如，如果你对 datadir 选项指定一条错误路径，那么 mysqld 会拒绝启动并输出有关的错误消息：

```
$./bin/mysqld --datadir=/wrong/path &
[1] 966

$110815 14:08:50 [ERROR] Can't find messagefile
'/users/ssmirnova/blade12/build/mysql-trunk-bugfixing/share/errmsg.sys'
110815 14:08:50 [Warning] Can't create test file /wrong/path/blade12.lower-test
110815 14:08:50 [Warning] Can't create test file /wrong/path/blade12.lower-test
./bin/mysqld: Can't change dir to '/wrong/path/' (Errcode: 2)
110815 14:08:50 [ERROR] Aborting

110815 14:08:50 [Note] Binlog end
110815 14:08:50 [Note]

[1]+  Exit 1                  ./bin/mysqld --datadir=/wrong/path
```

但是，当然，如果你在系统启动文件并用守护进程的方式启动 mysqld，那么你无法在命令行中看到这条消息。在那种情况中，用户通常会注意到这样的问题：在初次尝试连接 mysql 时失败，而随着这个问题出现的错误类似以下内容：

```
$./bin/mysql -uroot -S /tmp/mysql_ssmirnova.sock
ERROR 2002 (HY000): Can't connect to local MySQL server through socket
'/tmp/mysql_ssmirnova.sock' (2)
```

该错误只是向你说明服务器没有运行。在这种情况下，需要检查错误日志文件中的信息，或者，如果没有任何错误日志文件，那么需要检查操作系统日志中有关 mysqld 的消息。MySQL 的错误日志文件之前列出的一样的错误消息。而对于一些自动脚本、操作系统记录的消息则可能不同，比如从 MySQL 安装 mysql.server 然后通过 mysql.server 启动 mysqld 失败的问题。还可以在系统进程清单中检查 MySQL 服务器是否正在运行。这里有一个 Linux 下的例子，这个例子显示 mysqld 没有出现在系统进程列表中的任何位置：

```
$ps -ef | grep mysqld
10149     7076  6722  0 23:35 pts/0    00:00:00 grep mysqld
```

mysqladmin 实用工具有一个 ping 命令，它能够报告 MySQL 服务器当前的状态是运行

还是停止：

```
$mysqladmin -h127.0.0.1 -P3306 ping
mysqladmin: connect to server at '127.0.0.1' failed
error: 'Can't connect to MySQL server on '127.0.0.1' (10061)'
Check that mysqld is running on 127.0.0.1 and that the port is 3306.
You can check this by doing 'telnet 127.0.0.1 3306'
```

有几个选项在指向特定的路径时候不会影响到 MySQL 服务器的启动，但也可以简单地关闭特定选项。例如，让我们来看看 InnoDB 引擎启动失败是什么样的：

```
110815 14:14:45 [Note] Plugin 'FEDERATED' is disabled.
110815 14:14:45 [Note] Plugin 'ndbcluster' is disabled.
110815 14:14:45 [ERROR] InnoDB: syntax error in innodb_data_file_path
110815 14:14:45 [ERROR] Plugin 'InnoDB' init function returned error.
110815 14:14:45 [ERROR] Plugin 'InnoDB' registration as a STORAGE ENGINE failed.
110815 14:14:45 [Note] Event Scheduler: Loaded 0 events
110815 14:14:45 [Note] ./libexec/mysqld: ready for connections.
Version: '5.1.60-debug'  socket: '/tmp/mysql_ssmirnova.sock'  port: 33051
Source distribution
```

服务器已经成功启动，但是 InnoDB 引擎没有成功加载：

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: ndbcluster
     Support: NO
     Comment: Clustered, fault-tolerant tables
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 2. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
```

```
         XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
         XA: NULL
  Savepoints: NULL
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
         XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
         XA: NO
  Savepoints: NO
8 rows in set (0.00 sec)
```

关闭错误日志使我们能够在控制台上看到错误消息，但在生产环境中，错误日志是用来查看错误消息的地方。所以，如果你发现你所需要的某个功能不存在，那么请先检查你的错误日志文件。

了解你所需要的功能是否会影响到服务器的工作，是非常重要的。例如，当 InnoDB 引擎处于一个不可用的状态时，如果 SQL 模式不包含 NO_ENGINE_SUBSTITUTION，我们仍然能够成功创建引擎为 InnoDB 的表：

```
mysql> CREATE TABLE t1(f1 INT) ENGINE=InnoDB;
Query OK, 0 rows affected, 2 warnings (0.01 sec)
```

上面这个例子表明，检查警告信息始终是很有必要的。在这个例子中，我们在创建表的时候使用了错误的存储引擎，因为我们在尝试启动 InnoDB 引擎的时候发生了错误：

```
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1286 | Unknown table engine 'InnoDB'           |
| Warning | 1266 | Using storage engine MyISAM for table 't1' |
+---------+------+-----------------------------------------+
2 rows in set (0.00 sec)
```

如果你没有检查警告信息，那么使用这个表的用户可能只有在当这个问题影响到整个应用程序之后才会发现。从上一章中可以知道，MyISAM 引擎与 InnoDB 引擎的锁定方式是不同的，所以如果一个应用程序基于 InnoDB 引擎的优势编写，却使用了 MyISAM 的表，这将会带来严重的问题。并且我还没谈到不存在事务的情况呢！

- 如果某个依赖的功能出现问题，该功能是否存在于服务器实例上。

其他主要的配置问题涉及的选项是用来更改 MySQL 服务器运行方式的，虽然它们的主要目的并不相同。在对 MySQL 服务器选项进行配置的时候，你可能会希望它能为你提供一个或多个功能，但不要期望它们能够影响你的查询。

一个微不足道的例子是二进制日志对于创建存储函数的影响。当我们启用该功能后，你可以期望二进制日志会存储所有修改数据的事件，但你可能无法意识到它的副作用。

首先，我将演示如何在不使用二进制日志的情况下在 MySQL 服务器上创建一个虚拟的存储函数：

```
root> GRANT ALL ON test.* TO sveta@'%';
Query OK, 0 rows affected (0.01 sec)
```

然后，我用账户 sveta 来连接服务器并创建存储函数：

```
sveta> CREATE FUNCTION f1() RETURNS INT RETURN 1;
Query OK, 0 rows affected (0.02 sec)
```

目前看起来一切正常，但当我使用 log_bin 选项启动 mysqld 服务器后，事情会发生一些变化：

```
$./libexec/mysqld --defaults-file=support-files/my-small.cnf \
--basedir=. --datadir=./data --socket=/tmp/mysql_ssmirnova.sock --port=33051 \
--log_error --log_bin &
[1] 3658
```

然后尝试创建相同的存储函数：

```
sveta> DROP FUNCTION f1;
Query OK, 0 rows affected (0.00 sec)

sveta> CREATE FUNCTION f1() RETURNS INT RETURN 1;
ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL, or READS
SQL DATA in its declaration and binary logging is enabled (you *might* want to
use the less safe log_bin_trust_function_creators variable)
```

错误消息清楚地说明了问题出在哪里。我希望通过这个例子告诉你，选项是如何改变 MySQL 服务器运行方式的，尽管这一选项的主要功能并不会影响到用户查询。通常情况下，当用户遇到这一问题的时候，问题产生的原因并不明确，而且很容易引起混淆。

3.2 可更改服务器运行方式的变量

另一组变量会影响 MySQL 服务器如何处理用户的输入。

我将举一个简单的例子来清楚地说明设置这种类型的变量而带来的效果。在这个例子中，我将 SQL 模式设置为 STRICT_TRANS_TABLES 使无效数据插入事务性表的尝试被拒绝，而不是被忽略。然而，我们可以预计，对于这种非事务表，服务器会尽可能地修复 SQL 语句，而不是将其拒绝：

```
mysql> SET @@sql_mode = 'strict_trans_tables';
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TABLE `myisam` (
    ->           `id` bigint(20) NOT NULL AUTO_INCREMENT,
    ->           `a` varchar(50) NOT NULL,
    ->           `b` varchar(50) NOT NULL,
    ->           PRIMARY KEY (`id`)
    ->         ) ENGINE=MyISAM DEFAULT CHARSET=latin1 ;
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO `myisam` (id,a) VALUES (1,'a');
ERROR 1364 (HY000): Field 'b' doesn't have a default value
```

我故意执行一个错误的 INSERT 操作，省略了字段 b 的值。这种情况下，我期望服务器会为 b 字段写入一个空值。但是，即使该表使用 MyISAM 存储引擎，本次写入依然失败，并输出了对应的错误消息。

MySQL 参考手册说明了此时服务器的行为（参见 http://dev.mysql.com/doc/refman/5.1/en/server-sql-mode.html）。

> STRICT_TRANS_TABLES：如果一个值不能插入一个事务性表，那么中止该语句。对于非事务性表，如果该问题发生在一个影响到表中单行数据的语句中或是发生在一个影响到表中多行数据的语句中的数据的第一行，那么中止该语句。

这里的 INSERT 为单行语句，所以服务器会拒绝纠正它。但这看起来并不直观，这个选项到底是什么？

- 如果你发现你不期望出现的服务器行为，那么请仔细检查你的配置选项。

3.3 有关硬件资源限制的选项

本分类中的配置选项能够对硬件资源的利用施加限制。它们通常有两个用途：优化性能及限制某些操作。当你希望对客户端与服务器之间的流量施加一些限制，或者防止拒绝服务攻击时，刚才提到的两种用途中，后者对该场景非常有用。它能更好地让特定用户得到更优雅的错误，因为资源不足要好过 mysqld 进程终止，因为 mysqld 进程终止后便不能处理所有传入的请求。

本章最后将描述调整这一类选项时应该遵循的策略。这里希望指出的情景是，这些变量的不同设置可能会带来不同和意想不到的服务器行为。与往常一样，我会通过实际的例子来讲解。

在我日复一日的工作中，我发现很多用户忽略了 max_allowed_packet 变量的值。这个变量的设置能够限制服务器和客户端之间传递的单个数据包的字节数。在这个例子中，

为了达到演示效果，我将 max_allowed_packet 值设置为一个低于 1MB 的值。

```
mysql> SELECT repeat('a',1025);
+------------------+
| repeat('a',1025) |
+------------------+
| NULL             |
+------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 1301
Message: Result of repeat() was larger than max_allowed_packet (1024) - truncated
1 row in set (0.00 sec)
```

这一次，错误消息很清晰地告诉了你为什么会发生这一错误，但有时也不是那么清晰。

```
$./my sql test <phpconf2009_1.sql
ERROR 1064 (42000) at line 33: You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right syntax to use
near
'OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO'
at line 2
```

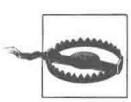
你得到的错误消息是否清晰，取决于处理你发送查询的服务器，通常发生在你从非常大的表或 BLOB 类型的字段中获取数据的时候。

- 如果你开始收到有效查询的语法错误，请检查查询获取的数据量的大小是否超过了 max_allowed_packet。

3.4 使用--no-defaults 选项

用心记住每个变量是什么，是很难的。虽然我每天都在和这些变量打交道，但我依然无法记住它们。我用的一个解决方案是，在选项列表中，一个个排除它们。但这并不简单。MySQL 的 5.1 版拥有 291 个变量，我安装的过时的 5.5 版则有 321 个变量，并且没有人能够保证变量的数量不会在未来继续增多。如果你使用自定义存储引擎，那么这个存储引擎也可以有自己专用的选项。

检查 MySQL 服务器是否没有指定选项是较容易的，即，如果它的选项全部使用默认值。如果你对于 msyqld 如何在未配置自定义选项的场景下运行有一个粗略的猜测，那么你就可以使用--no-defaults 选项启动 MySQL 服务器并与你设置了自定义选项的情况进行比较。



警告

--no-defaults 选项和--defaults-file 选项都必须指定为传递给 mysqld 的第一个选项。否则，服务器将会无法识别它们。

如果结果是不同的，那么可以在添加选项之前逐个使用测试方案，并检查服务器的错误行为是否复现。一旦你找到了这个错误是由哪个变量引起的，你就可以参考配置变量的文档，并进行相应的调整。

3.5 性能选项

这一类选项通常不会引发错误，但它们有可能对服务器的性能产生巨大的影响。通常可以在不同的真实负载的生产服务器上调整这些选项，直到找到一个适合特定环境的性能选项配置组合。

然而，当你选择了这些选项的时候，有一种可能出现的情况会导致错误，所以从配置文件中移除或降低性能配置选项的权重是有意义的。这个场景是：你的服务器遇到一个资源不足错误。最常见的情况涉及缺少内存或者文件描述符。如果服务器存在此类问题，可以使用上一节中的--no-defaults 办法来找明设置了太大的选项。

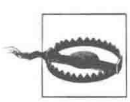
3.6 欲速则不达

这个流行的英文谚语在许多其他语种中有相同含义的说法。俄国有一种说法可以翻译为“慢慢地滑行，进一步到达”。我认为这条睿智的谚语也同样适用于调整 MySQL 服务器的时候，至少是当你无法 100%确定你知道自己在做什么的时候。

- 除非你 100%肯定错误是什么，然后逐个添加选项，并每次都对自己的配置进行测试

这意味着，如果你认为一组配置选项可以改变 MySQL 服务器的行为，并使它更好地为你的应用程序提服务，那么你可以改变一个选项，然后进行测试，如果结果是令你振奋的，那么你可以继续添加其他选项，以此类推，直到你已经检查完毕每一个相关选项。这可能是一个很缓慢的过程，但如果在这一过程中发生错误，你可以放心的回滚到本次变化之前，并能够迅速地让你的服务器回到工作状态。

在你调整内存缓冲区或者其他选项以限制对硬件资源的利用时，这种方式是非常重要的。但这种方式也同样能够用于服务器行为配置选项的修改。即使你对变量到底在做什么有着良好的认知，这也仅仅能让你可以更容易地发现和修复某个错误，而不是在几十种选项择中找出错误的根源。



警告

当你在使用该方法的时候，需要保存每一个测试结果。例如，如果你正在尝试提升服务器的性能，那么你需要运行基准测试或测试查询执行时间在配置选项改变前后的变化，然后重复同样的测试，再去修改每个选项。

3.7 SET 语句

MySQL 支持两种形式的变量：SESSION 及 GLOBAL。会话级别的变量设置只会对当前链接生效，而不会影响其他连接。GLOBAL 变量配置后将应用此后创建的所有连接。但设置一个 GLOBAL 变量，并不会影响当前连接[1]，所以如果需要在当前连接使用一个新的变量值，那么应该同时设置 SESSION 变量和 GLOBAL 变量。

可以通过这样的 SQL 语句来设置 SESSION 变量：

```
SET [SESSION] var_name=value
```

把 SESSION 放在中括号中是因为这样可以省略关键字，这里的 SESSION 指的是 set 命令使用的默认 SESSION。

你可以通过这样的 SQL 语句来设置 GLOBAL 变量：

```
SET GLOBAL var_name=value
```

当测试选项的时候，我建议你尽可能只使用 SESSION 变量来进行测试。当你得到满意的结果后，就可以使用一个 GLOBAL 变量来改变正在运行的服务器的配置，然后再修改对应的配置文件，以便在服务器重新启动后也能够应用此值。

提示

当你想使用一个特定查询求来检查选项的效果的时候，使用 SET SESSION 命令也是很有帮助的。在这种情况下，可以在查询前设置变量，然后再进行测试，使用 SQL 语句 SET [SESSION] variable_name=DEFAULT 来让变量恢复到默认值。

如果多个线程之间共享一个选项，那么可以设置一个 GLOBAL 变量，然后检查服务器的运行状态。在你满意后再修改配置文件，这些修改包括新的变量以及对它们的设置。

这种方式允许你测试变量的改变而无须中断应用程序，因为这会使你所需要的变更对程序的影响推迟到一个预定的服务器重启时间。

存在个别选项是不能动态设置的。在这种情况下，即使你仅仅希望测试一下选项改变带来的效果，你也必须重新启动 MySQL 服务器才能使变更生效。

3.8 如何检查变更是否存在一些影响

一些状态变量能够显示服务器的当前运行状态。相对于配置变量，这些状态变量并不会

1：存在少数例外，例如 SET GLOBAL general_log=1。通常这一类例外没有等效的 SESSION。

影响服务器的运行行为，但它们包含 mysqld 进程正在发生的行为的具体信息。状态变量是只读的，即，它们的改变来自 MySQL 服务器，而不是用户。它们通常会显示比如执行了多少次查询，查询的类型，网络流量，索引是如何使用的（可以在 1.6.5 节中找到一个例子），缓冲区的使用情况，整个实例中有多少表打开，有多少临时表创建了，以及其他很多有用的信息。这里不会描述每一个变量的信息，但如 1.6.5 节那样，在本章稍后介绍状态变量时，我会注意哪个状态变量要监控。

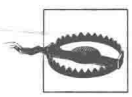
在变量变更上下文中，状态变量主要在 3.9.2 节非常有用，第 6 章会讲述如何从 MySQL 配置中获取状态信息。

像其他变量一样，状态变量可以应用到个别会话以及所有会话（全局）上。会话变量会显示当前会话的状态，而全局变量则会显示自服务器启动后或者自上一次执行了 FLUSH STATUS 命令后的状态。

提示

有些变量不是独立的，例如，当 query_cache_size 变量设置为 0 时，控制查询缓存是否开启的变量是无用的。调整类似选项的时候，需要考虑整组带来的影响，而不是单独只考虑一个变量。

当更改服务器的选项时，它会使状态变量产生变化。例如，如果你修改了表缓存的选项，那么你应该查看一下 Open_tables 和 Opened_tables 这两个状态变量的值。在设置正确的情况下，Opened_tables 的值不应该增长，而所有的 Open_tables 都应该处于缓存中。



警告

某些情况下，由于某个变量的值设置得太大或者太小，会导致该变量的值被服务器丢弃。如果你怀疑你的变更并没有对服务器的运行造成影响，那么你需要通过查询“SHOW [SESSION|GLOBAL] VARIABLES LIKE '变量名称'”或查询“SELECT VARIABLE_VALUE FROM INFORMATION_SCHEMA.[SESSION| GLOBAL]_VARIABLES WHERE VARIABLE_NAME='变量名称'”来对你的设置进行检查。

3.9 变量介绍

既然你已经熟悉无风险的服务器变量调优方法，我们就准备介绍几个重要的变量。这不是一个完整的指南，但是进一步研究的一个出发点。

你可以以任意顺序阅读本节其余内容：从开始到结尾通读它，检查你当前最感兴趣的特定主题，或甚至跳过它，并且当你遇到问题时作为参考来使用它。我不会介绍每个

选项，但会集中介绍我发现经常误用的变量或者需要扩大认识的变量。

提示

起初，我怀疑我是否需要用本书一节的篇幅来介绍个别变量，因为每一个变量都在其他资源上详细介绍。但 MySQL 参考手册中没有从故障排查方法特别介绍它们，所以我决定对它们提供最简短的描述。

3.9.1 影响服务器与客户端行为的选项

本节介绍与同步复制、连接、存储引擎相关的一般服务器选项。在后面的集合中，我将仅仅涵盖 MyISAM 与 InnoDB 存储引擎。

1. 服务器相关选项

这些选项影响所有的连接与语句。

限制与 max_*变量

在 3.3 节中你已经看到了 max_allowed_packet 如何影响

应用程序。另外一些选项限制了结果集的大小。例如：group_concat_max_len,它限制 GROUP_CONCAT 函数可以返回的字节数。如下比较：

```
mysql> SELECT @@group_concat_max_len;
+------------------------+
| @@group_concat_max_len |
+------------------------+
|                   1024 |
+------------------------+
1 row in set (0.00 sec)

mysql> SELECT group_concat(table_name) FROM tables WHERE
table_schema='mysql'\G
*************************** 1. row ***************************
group_concat(table_name):
columns_priv,db,event,func,general_log,help_category,help_keyword,help_relation,
help_topic,host,ndb_binlog_index,plugin,proc,procs_priv,servers,slow_log,
tables_priv,time_zone,time_zone_leap_second,time_zone_name,time_zone_transition,
time_zone_transition_type,user
1 row in set (0.15 sec)
```

与

```
mysql> SET group_concat_max_len=100; ❶
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT group_concat(table_name) FROM tables WHERE
table_schema='mysql'\G
*************************** 1. row ***************************
```

```
group_concat(table_name): columns_priv,db,event,func,general_log,help_category,
help_keyword,help_relation,help_topic,host,ndb_
1 row in set, 1 warning (0.06 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 1260
Message: Row 11 was cut by GROUP_CONCAT()
1 row in set (0.00 sec)
```

❶ 在本书中，我减少 group_concat_max_len 值来显示一个适合本书的示例。在现实生活中，问题通常发生在当用户在大数据集上执行此函数而其默认值却太小的时候。

我不会介绍 max*组的每一个变量。如果你发现 mysqld 限制了你发送语句或返回结果的大小，你只须检查它们的值。

权限

你需要考虑到另一种可能是，如果语句失败，你的用户是否有执行它的权限，或特定数据库或者表的权限。例如：local_infile 选项可以允许或者禁止执行 LOAD DATA LOCAL INFILE 查询。MySQL 服务器通常会给出明确的错误消息来解释为什么一种或另一种操作是不允许的。

SQL 模式

MySQL 服务器定义的 SQL 模式可以改变服务器如何对待客户端的输入。你已经看到 NO_ENGINE_SUBSTITUTION 与 STRICT_TRANS_TABLES 能影响应用程序。其他模式能改变其他行为。

下面是使用 ANSI_QUOTES 模式的另外一个示例。该模式告诉 MySQL 服务器使用 ANSI SQL 标准中定义的引号，而不使用 MySQL 本身默认的引号。这里介绍的问题不会发生在使用此模式的时刻，但会发生在用户依靠默认空模式希望服务器会错误地拒绝 ANSI 引号的时刻。

```
mysql> SELECT @@sql_mode;
+------------+
| @@sql_mode |
+------------+
|            |
+------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t1 WHERE "f1"=1;
Empty set, 1 warning (0.00 sec)

mysql> SET SQL_MODE='ansi_quotes';

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t1 WHERE "f1"=1;
```

```
+----+----------------------------------+
| f1 | f2                               |
+----+----------------------------------+
|  1 | f9f760a2dc91dfaf1cbc95046b249a3b |
+----+----------------------------------+
1 row in set (0.21 sec)
```

在第一种情况下，默认值起作用，MySQL 将“f1”作为一个字符串然并且将其转换为双精度（DOUBLE）。这是默认 SQL 模式的有效值，但与用户期望的不同。

```
mysql> SHOW WARNINGS;
+---------+------+---------------------------------------+
| Level   | Code | Message                               |
+---------+------+---------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'f1' |
+---------+------+---------------------------------------+
1 row in set (0.09 sec)
```

当转化为双精度时，把 f1 的值转换为 0，0 是不等于 1 的。在 ANSI_QUOTES 模式下，f1 被认为是一个字段名，所以查询能正常执行。这是一个很常见的问题，在包含多个条件的复杂查询中很容易被忽视。

- 当遇到“奇怪”的查询结果时，请检查 SQL 模式并分析它是否影响查询。

本书包含了几个关于 SQL 模式的示例，以显示它们如何使不同的服务器行为有不同的影响。我推荐你研究 SQL 模式列表以及在 MySQL 参考手册中它们是如何做的。

值得一提的一个细节是，从 5.1.38 版本开始，InnoDB 插件有一个 innodb_strict_mode 选项，如果选项开启，它能严格检查插入 InnoDB 表中的数据，此作用与严格的 SQL 模式非常类似，不过略有不同。所以，如果使用 InnoDB 插件，需要检查此变量的介绍。该选项默认是关闭的。

字符集与排序规则

当 latin1 字符集不适合你的需要时，对于必须使用 MySQL 来存储非英语语言数据的用户来说，理解这些变量是至关重要的。

提示

字符集是匹配字符或符号与表示它的字节序列的图。排序规则是一个排序规则，一个字符集可以有多个排序规则。

本主题非常大，所以这里不会面面俱到，但是我会给你几个开始点。

在 MySQL 中支持字符集与排序规则相当好，但是它的很多方面需要调整，所以人们对它们常常感到困惑。当你怀疑一个与字符集相关的错误时候，我推荐你认真研究 MySQL 参考手册中关于字符集与排序规则的章节。通常，在那里能找到问题的答案。

下面的一个示例显示，仅仅更改表的排序规则如何影响数据：

```
mysql> SET NAMES latin1;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TEMPORARY TABLE t1(f1 VARCHAR(255)) DEFAULT
CHARSET=latin1 COLLATE=latin1_german2_ci;
Query OK, 0 rows affected (0.23 sec)

mysql> INSERT INTO t1 VALUES('Sveta'), ('Andy');
Query OK, 2 rows affected (0.29 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> CREATE TEMPORARY TABLE t2 AS SELECT 'Sveta' AS
f1;
Query OK, 1 row affected (0.21 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t1 JOIN t2 USING(f1);
ERROR 1267 (HY000): Illegal mix of collations (latin1_german2_ci,IMPLICIT) and
(latin1_swedish_ci,IMPLICIT) for operation '='
```

为什么连接（join）查询失败？我们在第一个表中指定了 latin1_german2_ci 排序规则，而第二表在连接中使用默认的排序规则。

稍后会回到该示例，但是首先我将显示两个对于诊断此问题非常有用的查询。

```
mysql> SHOW VARIABLES LIKE '%char%';
+--------------------------+--------------------------------------------+
| Variable_name            | Value                                      |
+--------------------------+--------------------------------------------+
| character_set_client     | latin1                                     |
| character_set_connection | latin1                                     |
| character_set_database   | utf8                                       |
| character_set_filesystem | binary                                     |
| character_set_results    | latin1                                     |
| character_set_server     | utf8                                       |
| character_set_system     | utf8                                       |
| character_sets_dir       | /Users/apple/mysql-5.1/share/mysql/charsets/ |
+--------------------------+--------------------------------------------+
8 rows in set (0.09 sec)


mysql> SHOW VARIABLES LIKE '%coll%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | latin1_swedish_ci |
| collation_database   | utf8_general_ci   |
| collation_server     | utf8_general_ci   |
+----------------------+-------------------+
3 rows in set (0.01 sec)
```

每当你怀疑某些关于字符集与排序规则的事情有错误时，请运行上面两个查询，然后分析查询的结果与上下文。通常的安全规则是获取所有的 character_set_* 变

量、collation_*变量，并创建选项相同的任何表与一起协同工作的连接。设置客户端选项最简单方式是用 SET NAMES 语句。当然，当需要不同的字符集与排序规则时，可以使用它，但是你应该明白它们的影响。

如果我们回到 collation_connection 示例中，不同的排序规则是造 JOIN 接查询不能执行的根本原因。可以改变变量的值来证实这一点：

```
mysql> SET COLLATION_CONNECTION='latin1_german2_ci';
Query OK, 0 rows affected (0.00 sec)

mysql> DROP TABLE t2;
Query OK, 0 rows affected (0.04 sec)

mysql> CREATE TEMPORARY TABLE t2 AS SELECT 'Sveta' AS
f1;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t1 JOIN t2 USING(f1);
+--------+
| f1     |
+--------+
| Sveta |
+--------+
1 row in set (0.00 sec)
```

- 当在排序或者比较过程中遇到问题时，请检查字符集选项与表的定义。

*操作系统处理 lower_case*参数*

lower_case_filesystem 与 lower_case_table_names 选项跟字符集选项的作用非常相似。这些变量确定操作系统如何处理数据库对象的大小写情况。

最好不要修改它们的值，特别是操作系统不区分大小写时。这可能导致意外的结果，如下面的示例：

```
mysql> SELECT @@lower_case_table_names;
+--------------------------+
| @@lower_case_table_names |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.10 sec)

mysql> CREATE TABLE Table1(F1 INT NOT NULL AUTO_INCREMENT PRIMARY
KEY) ENGINE=InnoDB;
Query OK, 0 rows affected (0.27 sec)

mysql> CREATE TABLE Table2(F1 INT, CONSTRAINT F1 FOREIGN KEY(F1)
REFERENCES Table1(F1)) ENGINE=InnoDB;
Query OK, 0 rows affected (0.04 sec)

mysql> \q
```

```
Bye

$mysqldump --socket=/tmp/mysql50.sock -uroot test Table1
Table2
-- MySQL dump 10.11
<skipped>
--
-- Table structure for table `Table1`
--

DROP TABLE IF EXISTS `Table1`;
SET @saved_cs_client     = @@character_set_client;
SET character_set_client = utf8;
CREATE TABLE `Table1` (
  `F1` int(11) NOT NULL auto_increment,
  PRIMARY KEY  (`F1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
SET character_set_client = @saved_cs_client;

--
-- Dumping data for table `Table1`
--

LOCK TABLES `Table1` WRITE;
/*!40000 ALTER TABLE `Table1` DISABLE KEYS */;
/*!40000 ALTER TABLE `Table1` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `Table2`
--

DROP TABLE IF EXISTS `Table2`;
SET @saved_cs_client     = @@character_set_client;
SET character_set_client = utf8;
CREATE TABLE `Table2` (
  `F1` int(11) default NULL,
  KEY `F1` (`F1`),

  CONSTRAINT `F1` FOREIGN KEY (`F1`) REFERENCES `table1` (`F1`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;
SET character_set_client = @saved_cs_client;

%lt;skipped>
-- Dump completed on 2008-02-02 21:12:22
```

我已经把输出的重要部分用粗体显示。外键定义指向 table1 表，但是该表名不存在。你不能通过在不区分大小写的操作系统（如 Windows）上的转储在区分大小写的操作系统上（如 Linux）来还原。

在所有语句上同样最好使用一致性表命名方式并且在不区分大小写的操作系统上不混合使用大小写名称。

初始 SQL

这些选项确定服务器在不同的时间是否应该自动执行某些 SQL 语句。

init_file

指向包含在服务器启动时应该执行的 SQL 语句的一个文件。

init_connect

包含在每个客户端连接时需要执行的一个 SQL 字符串。

init_slave

包含当一个服务器作为从服务器启动其 SQL 线程时需要执行的一个 SQL 字符串。

当使用这些选项时，有两种典型的错误用法。

第一个问题是你很容易忘记你所配置的选项。通常情况下，选项用来设置连接的一些默认值。所以当使用默认选项时，如果你得到了与它们应该返回的结果不同的结果，请检查你是否已经设置了这类变量的一个或多个。

同样可以执行 SHOW [GLOBAL] VARIABLES 来找到连接使用的默认值。如果通过编程 API 来连接，请检查使用相同 API 的变量值，因为其他环境中，例如：MySQL 命令行客户端，可能使用不同的配置文件因而有不同的默认值。

init_connect 选项的内容仅仅在连接用户没有 SUPER 权限时才执行。这样做是为了让具有 SUPER 权限的用户即使在 init_connect 内容有错误的情况下也能连接。这是另外一个常见的使用错误，当用户作为 SUPER 用户连接并期望 init_connect 的内容执行时。

open_files_limit

这个重要选项限制 MySQL 服务器同时打开文件句柄的数量。限制数量越高，打开的表文件与临时表越多，因此处理的并发连接量数越多。如果这个限制在你的环境中设置得太低，在你试图连接、打开一个表或者执行一个需要创建临时表的查询时就会出现错误。

因为此地选项的设置反映了硬件的限制，所以下一章将进一步介绍它。

log_warnings

当此选项打开（非零）时，就会在服务器的错误日志文件中写入警告信息。它们不是在 SQL 执行期间发出的警告，而是显示服务器内到底是怎么回事的调试消息。

如果设置为 2，此选项告诉服务器记录连接错误。当你正在对客户端无法连接或正在失去连接的情况做故障排除时，这非常重要。日志并非总是能找到问题所在，但其警告消息往往对要做什么能给出一些启发。当使用同步复制时，在主服务器上开启此选项非常重要，因为你能确定从服务器 I/O 线程何时失去连接。反过来，它是网络故障的一种症状，这在将来可能导致更严重的问题。

当在从服务器上设置为 1（默认值）时，它将输出自己的诊断消息，例如：在二进制日志和中继日志中的位置及其复制状态。从 5.1.38 版本开始，在基于语句模式的同步复制中，需要启用此选项，以便在从服务器输出不安全语句的信息。（5.1.38 版本之前，从服务器在任何情况下都会输出此类消息。）从 5.1.38 版本开始，可以关闭此选项（设置为 0），来丢弃你确定不需要该消息的日志。

2. 复制选项

这些选项确定了主从服务器之间的关系。

*binlog-*与 replicate-*过滤器*

通过 binlog-do-*、replicate-do-*、binlog-ignore-*与 replicate-ignore-* 选项，在复制过程中，MySQL 有能力过滤对象。binlog-*选项减少在主服务器上写入二进制日志文件的事件，而 replicate-*指定在从服务器上记录到二进制日志。从服务器还有 replicate-wild-do-*与 replicate-wild-ignore-*选项，二者允许通过模式匹配，指定哪些应该或哪些不应该同步。

关于这些选项最常见的问题是：

- 大家忘记他们指定了这些选项。
- 在 binlog-do-*、replicate-do-*、binlog-ignore-*与 replicate-ignore-*中指定的过滤器当且仅当显式调用 USE dbname 时才生效。

问题的特征是：

- 某个特定查询没有复制。
- 在从服务器上报告“unknown table xxx on query”错误。
- 当你使用基于语句的复制（SBR）并且发现一些查询要么没有复制要么复制报错，请检查你是否设置了这些选项并且是否使用 USE dbname。

使用这些选项的变体 replicate-wild-*总是好的，因为这些不依赖调用 USE。

二进制日志格式

binlog_format 变量允许你选择复制的格式：STATEMENT、ROW、或 MIXED。

这是一个动态变量，它能在 SESSION 级别调整。对于某个特定查询，如果你不想使用当前默认的模式，你可以使用 binlog_format='row' 或 SET binlog_format='statement' 临时切换格式。

binlog_direct_non_transactional_updates

此选项指定何时非事务表更新应该写入二进制日志。

默认情况下，当使用事务时，MySQL 将非事务表的更新写入事务缓存里，仅当事务提交后，才把缓存刷新到二进制日志里。这样做以便从服务器更有可能与主服务器数据最终一致，即使依赖事务表中的数据来更新非事务表，并且主服务器在许多并发线程中同步更新相同的表。

但这种解决方案会导致一些问题（当另外一个事务造成的更改依赖于使用非事务表中未提交的并行事务修改的数据时）。2.7.2 节中的示例介绍了相似问题。如果遇到了这样的问题，可以打开这个选项。在打开之前，请务必确定非事务表中的数据不被使用事务表的语句更改。

这是一个动态变量，可以在 SESSION 级别改变它，所以，可以在特定语句下使用它。它的工作原理决定了它只能在基于语句模式的复制中才生效。

log_bin_trust_function_creators

这个选项告诉 mysqld 当用户没有 SUPER 权限却试图在主用品创建一个不确定性的函数时，不要触发警告。请参见 3.1 节中默认行为的范例。

binlog_cache_size 与类似选项

此条目包括以下选项：

- binlog_cache_size
- binlog_stmt_cache_size
- max_binlog_cache_size
- max_binlog_stmt_cache_size

在写入二进制日志之前，这些缓存保存在事务期间提交的事务与非事务语句。如果达到了 max_binlog_cache_size，语句将会中止并报告“Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage”错误。

检查 Binlog_cache_use、Binlog_stmt_cache_use、Binlog_cache_disk_use 与 Binlog_stmt_cache_disk_use 状态变量来找出 binlog 缓存使用的频率以及事务大小超过 Binlog_ cache_use 与 Binlog_stmt_cache_use 的频率。当事务大小操作缓存大小时，将会创建临时表来存储事务缓存。

slave_skip_errors

此选项允许从服务器 SQL 线程即使遇到某类错误时还能运行。例如，主服务器运行在宽松的 SQL 模式而从服务器却有一个严格的 SQL 模式，当插入字符串到整数字段时，由于数据格式不一致，而报告 1366 (ERROR 1366 (HY000): Incorrect integer value)错误，可以设置 slave_skip_errors，以便从服务器不会出现故障。

这个选项可能导致主服务器从服务器数据不一致而又很难诊断，所以，如果你遇到这样的问题，请检查该选项是否没有设置。

read_only

该选项使从服务器服务器只读。这意味着，仅仅只有从库 SQL 线程才能更新其数据，而其他连接只能读数据。该选项对于保持从服务器数据的一致很重要。然而，这个选项并不能限制具有 SUPER 权限的用户更改表。另外，所有用户仍允许创建临时表。

3. 引擎选项

本节介绍的选项具体到特定的存储引擎。我将会在此讨论 MyISAM 和 InnoDB 的一些选项。与性能相关的选项在随后的小节里概述。在排除故障的情况下，你应该熟悉并仔细检查你使用的存储引擎的所有选项。

InnoDB 选项。我们将从 InnoDB 存储引擎的选项开始。

innodb_autoinc_lock_mode

此选项定义 InnoDB 在插入自增字段时用到的锁定方式。有三种模式：traditional（在 5.1 版本之前使用）consecutive（从 5.1 版本开始作为默认模式）和 interleaved。consecutive 是最安全的。使用另外两个模式往往以获取更好的性能，traditional 方式也使用在向后兼容时。

这里不说明不同锁类型之间的区别，因为 MySQL 参考手册详细说明了各自是如何工作的。然而牢记一点：如果你的应用程序设置自增值的方式让你感到意外，检查此模式并验证不同值是如何影响自增的。实际上我并不推荐从安全的 consecutive 模式转换至任何其他模式，但在某些场景下是可以接受的。

innodb_file_per_table

InnoDB 默认在共享表空间中存放表和索引数据。使用此选项，你可以告知它将表的索引和数据存放在单独的文件里。共享表空间仍然用来存放表定义。此选项在设置后创建的表上生效；之前创建的表依然使用共享表空间。

使用这个变量实际上是个很好的做法，因为它有助于 InnoDB 表更有效地运作。除了能让你看到表实际占用的空间，它也能让你利用 MySQL 企业备份创建一个不完全备份，甚至在不同的 MySQL 安装版本上恢复一张表（使用 Chris Calender 博客上描述的方式，参见 http://www.chriscalender.com/?p=28）。

innodb_table_locks

此变量定义了 InnoDB 是如何处理 LOCK TABLES 语句发出的表锁请求。默认（当设置了）立刻返回并且内部将表锁住。当关闭时（设置为 0），它会接收 LOCK TABLE 语句，线程直到所有锁释放后才从 LOCK TABLES … WRITE 返回。

innodb_lock_wait_timeout

这是 InnoDB 等待行锁直到放弃的秒数。在 innodb_lock_wait_timeout 秒后，它会

返回错误“ERROR 1205 (HY000):Lock wait timeout exceeded; try restarting transaction”至客户端。我经常看到人们将这个变量设得很大来防止查询失败，这只会导致更严重的问题，因为许多阻塞的事务会互相锁住。尝试在应用程序层处理锁等待错误，并不要将它设置得过高。此参数的最佳值取决于应用程序，通常应该大约为正常事务所消耗的时间。它的默认值是 50 秒，对于需要立即返回结果的应用程序有些大。大多数网店正是如此。

innodb_rollback_on_timeout

当查询因锁等待错误而中断时，只有最后一条语句回滚了，整个事务还没有中止。如果将选项设置为 1 你将会改变此行为。这种情况下事务会在锁等待超时后立刻回滚。

innodb_use_native_aio

在 InnoDB 1.1 版本插件中引入此选项，指定 InnoDB 是否应该使用 Linux 下原生的 AIO 接口，或者是自己来实现，称作“模拟 AIO”。如果设置 innodb_use_native_aio，InnoDB 将分发 I/O 请求至内核。这提高了可扩展性因为比起模拟 AIO 新内核能够处理更多的并行 I/O 请求。

此选项默认开启，在正常操作下不应该改变。如果你遇到操作系统异步 I/O 子系统的问题，阻止 InnoDB 启动，你可以将它关闭。典型的错误消息提示你将此选项关掉：error while loading shared libraries: libaio.so.1: cannot open shared object file: No such file or directory.

Innodb_locks_unsafe_for_binlog

此变量定义 InnoDB 如何使用间隙锁来搜索和扫描索引。默认值（设为 0）下，间隙锁开启。如果设为 1，大多数操作下会禁用间隙锁。其工作原理类似于隔离级别中的 READ COMMITTED，但由于不太好调节应尽量避免。即使它允许你来处理锁问题，当并行事务插入新行至间隙时它也会带来新的问题。所以推荐用 READ COMMITTED 替代它。这个变量不能设置为 SESSION 级别，它会影响所有事务。

MYISAM 选项。这里只讨论两个选项，下一节讨论剩下的部分。

Myisam_data_pointer_size 当不指定 MAX_ROWS 参数创建 MyISAM 表时设置使用的默认指针大小。默认值是 6，允许的范围是 2 ~ 7。指针越大，表能容纳的行越多。默认的 6 允许你创建最大 256TB 的表。如果你使用的 MyISAM 表遇到一个“Table is full”错误，这意味着对于表数据指针设置过小（参考边栏“表可以有多大”）。

表可以有多大？

可以使用 myisamchk–dvi 查看对于特定的指针大小表的精确大小与如果使用 FIXED 行格式它会存储多少行数据。

```
mysql> CREATE TABLE t1(f1 INT, f2 VARCHAR(255)) ENGINE=MyISAM;
Query OK, 0 rows affected (0.16 sec)

mysql> SET GLOBAL myisam_data_pointer_size=2;
Query OK, 0 rows affected (0.00 sec)

mysql> \q
Bye

C:\Program Files\MySQL\MySQL Server 5.5>.\bin\mysql -uroot test
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.5.13-enterprise-commercial-advanced
MySQL Enterprise Server - Advanced Edition (Commercial)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.

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

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

mysql> CREATE TABLE t2(f1 INT, f2 VARCHAR(255)) ENGINE=MyISAM;
Query OK, 0 rows affected (0.13 sec)

mysql> \q
Bye

C:\Program Files\MySQL\MySQL Server 5.5>.\bin\myisamchk.exe -dvi
"C:\ProgramData\MySQL\MySQL Server 5.5\Data\test\t1"

MyISAM file:         C:\ProgramData\MySQL\MySQL Server 5.5\Data\test\t1
Record format:       Packed
Character set:       utf8_general_ci (33)
File-version:        1
Creation time:       2011-11-02 14:43:40
Status:              checked,analyzed,optimized keys,sorted index pages
Data records:                0  Deleted blocks:                0
Datafile parts:              0  Deleted data:                  0
Datafile pointer (bytes):    6  Keyfile pointer (bytes):      3
Datafile length:             0  Keyfile length:             1024
Max datafile length: 281474976710654  Max keyfile length: 17179868159
Recordlength:              774

table description:
Key Start Len Index   Type                     Rec/key         Root  Blocksize

C:\Program Files\MySQL\MySQL Server 5.5>.\bin\myisamchk.exe -dvi
"C:\ProgramData\MySQL\MySQL Server 5.5\Data\test\t2"

MyISAM file:         C:\ProgramData\MySQL\MySQL Server 5.5\Data\test\t2
Record format:       Packed
Character set:       utf8_general_ci (33)
File-version:        1
Creation time:       2011-11-02 14:44:35
Status:              checked,analyzed,optimized keys,sorted index pages
Data records:                0  Deleted blocks:                0
Datafile parts:              0  Deleted data:                  0
Datafile pointer (bytes):    2  Keyfile pointer (bytes):      3
```

```
Datafile length:              0  Keyfile length:            1024
Max datafile length:      65534  Max keyfile length:  17179868159

Recordlength:               774

table description:
Key Start Len Index   Type                    Rec/key         Root  Blocksize
```

myisam_recover_options

此选项告知 MySQL 服务器检查，每次它打开一个 MyISAM 表时，表是否损坏或者没有正常关闭。如果检查未通过，MySQL 在表上运行 CHECK TABLE，如有必要将修复表。可能的值有 OFF、DEFAULT（不是默认选项，但表示恢复方法不使用备份，强制进行一个快速检查）、BACKUP（创建表数据的.MYD 文件的一个备份）。FORCE（指示服务器运行恢复操作，即使有丢失.MYD 文件中一行或多行数据的风险），以及 QUICK（如果未检测到表中有删除的块告知服务器不要运行恢复动作）。你可以同时使用两种或多种选项。此选项最常见的值，设置为 BACKUP、FORCE，因为它能修复所有错误，并且安全因为它创建了一个备份文件。默认这个选项是关闭的。

4. 连接相关的选项

从故障排查角度来看这些选项的首要关注点在超时。我也会讨论跟安全相关并且通常导致（或解决）问题的一些选项。

超时。你已经熟悉了 innodb_lock_wait_timeout，它中断那些等待行锁的查询。

类似的参数有 lock_wait_timeout，它适用于元数据锁。这个锁对所有需要元数据锁的操作有效：DML（数据操纵语言语句，如 INSERT、UPDATE 和 DELETE），DDL、LOCK TABLES 等。默认值是 3 153 600 秒，也就是一年。所以默认情况下，有效 MDL 锁永远不会解除。然而，可以更改值为大于 1 秒的任何值。它是一个动态变量，可以在合适级别更改。

还有一系列超时相关参数，它们和你所运行的查询不相关，限制结果集、客户端数据和授权包的等待时间。这些参数包括以下几个。

connect_timeout

这个超时使用在 MySQL 服务器和客户端交换授权包时。从 5.1 版本开始，此值默认设置为 10 秒。

interactive_timeout

交互式客户端在断开连接之前等待活动多长时间，即服务器等待多久来读取下一条命令。该术语“交互式客户端”是那些直接运行人们发的查询的客户端。例如，

MySQL 命令行客户端（mysql）是交互式的，而 Web 应用程序默认则不是。编写应用程序时，如果你考虑交互性你需要明确地指定。

wait_timeout

在断开连接前等待任何客户端中活动的时间。如果客户端是交互式的并且 interactive_timeout 的值不同于 wait_timeout，则以 interactive_timeout 为准。

net_read_timeout

从客户端写入 MySQL 服务器等待应答的时间。例如，此超时会在客户端执行大的插入操作时起作用。

net_write_timeout

客户端从服务器中读取时等待应答的时间。例如，当客户端发送一个 SELECT 查询读取结果，如果客户端等待一段时间未收到数据，这个超时会断开此连接。如果客户端需要在处理结果前做一些工作，检查工作的持续时间是否长于这个超时。

碰到绝大多数这些限制的现象就是“MySQL server has gone away”错误或者“Lost connection to MySQL server during query”错误。connect_timeout 是例外。如果你遇到了这个限制，你会得到“Lost connection to MySQLserver at 'reading authorization packet'”错误。当从服务器 I/O 线程不能连接主服务器时也会得到类似错误。

如你遇到了前面提到过的限制，不要盲目增加它们；找寻产生问题的真正原因。如果超时是由不稳定的网络造成，你应该修复网络，而不是增加超时时间。这里列出当你怀疑超时问题后可以采取的一些行动：临时增加*timeout 变量，重新运行应用程序。如果超时频率这时变小，你可以肯定是超时时间的问题，但需要找到真正产生错误的原因。这可能是运行很久的应用程序，对大表访问慢或者不稳定的网络。

与安全相关的选项。这些选项控制权限及 MySQL 服务器安全相关的其他方面。

skip-grant-tables

当客户端连接服务器时如果缺乏正确的用户权限另一个授权问题会出现。1.9 节讨论过一点。这里我只想教大家如何在忘记密码的情况下拯救自己。你需要使用 skip-grant-tables 选项启动服务器，手动编辑 mysql 数据库中的权限表，然后执行 FLUSH PRIVILEGES 语句。在这之后，新权限生效。不要忘记去掉 skip-grant-tables 选项重启服务器。否则，在重启后任何人都能连接上你的服务器。要安全地完成这个操作，包含 skip-grant-tables 以及 skip_networking 选项，这样当访问 MySQL 服务器没有限制时只允许本地连接。

safe-user-create

不允许使用 GRANT 语句来创建用户，除非用户有 mysql.user 表的 INSERT 权限才可以。

secure_auth

不允许早于 4.1 版的客户端连接现代服务器。选择 4.1 版本是因为当时新的安全模型添加到了连接协议中。

secure_file_priv

限制 LOAD_FILE 函数以及 LOAD DATA 和 SELECT… INTO OUTFILE 语句只能使用指定目录。

3.9.2 与性能相关的选项

这里会对影响性能的选项进行一个简短的概述。同样，我不会面面俱到，而只介绍那些最常用的。和上一节不同的是，这些选项不会导致不同的结果[1]。

首先讨论影响服务器整体行为的选项，然后是一些特定于引擎的选项。

1. 缓冲区和最大值

第一组选项控制服务器内部使用的内存总量，以及内存使用的上限。

join_buffer_size

这是为连接操作分配的最小缓存大小，这些连接使用普通索引扫描、范围扫描，或者连接不使用索引。两表之间进行全连接时分配缓存。因此，连接两个表的一条查询分配一块缓存，连接 3 个表的一个分配两块查询缓存，以此类推。这个选项可在 SESSION 级使用，能对于特定连接设置。

为了查出是否需要增加 join_buffer_size，可以检查 Select_scan 状态选项，它包括第一张表执行完整扫描的连接数量，同样 Select_full_range_join，它包含使用范围搜索的连接数量。这些状态变量的值不会随着 join_buffer_size 的值的变化而变化，这样你可以利用它们来查出是否需要大的 join_buffer_size，而不是衡量该值改变后的有效性。

net_buffer_length

服务器在客户端连接后立刻创建的缓存大小，用来保存请求和结果。根据需要这个大小可以增长至 max_allowed_packet。正常情况下不用改变默认值(16384 字节)，但当设置 max_connections 选项时要记住此值。

1：有个例外是 EXPLAIN 语句中的优化器参数。

query_prealloc_size

此缓存为语句解析和执行而分配。语句间缓存是不释放的。如果运行复杂查询，增加缓存是合理的，这样 mysqld 不会在执行查询的时候在分配内存上耗时。增加此大小到最大查询的字节数。

read_buffer_size

每个完成顺序扫描线程会分配为每次表扫描此缓存。

read_rnd_buffer_size

此变量用来控制在排序和发送结果至客户端之间存放读取结果的大小。大的值能提高包含 ORDER BY 的查询的性能。

sort_buffer_size

每个线程需要排序的时候会分配此缓存。查明你是否需要增加此缓存的大小，检查 sort_merge_passes 状态变量。也可以检查 sort_range、sort_rows 和 sort_scan 来查明你执行了多少个排序操作。

这些状态变量显示了排序操作的数量。为了找出缓存合适的大小，需要检查一条或多条查询排序的行数，并乘以行大小。或者简单地设置不同的值直到 sort_merge_passes 停止增长。

提示

sort_buffer_size 缓存经常会分配，所以大的 GLOBAL 值会降低性能而不是增加性能。因此，最好不要设置此选项为 GLOBAL 变量，而是当需要时使用 SET SESSION 增加它。

sql_buffer_result

当此变量设置时，服务器会把每条 SELECT 语句的结果缓存到临时表中。当客户端查询需要长时间获取结果时这有助于提前释放表锁。在把结果存储到临时表中之后，服务器会释放原表上的锁，在第一个客户端仍获取结果时让其他线程可访问。

为了找出查询是否在发送结果集上消耗过多时间，执行 SHOW PROCESSLIST 来检查查询在“Sending data”状态下的时间。

提示

在 SHOW PROCESSLIST 输出里“Sending data”状态代表线程正在读取和处理行，并发送数据至客户端。正如你所看到的，这比文字描述的更加复杂，并不一定意味着查询卡在发送数据状态下。

thread_cache_size

为将来使用缓存起来的线程数量。当一个客户端断开连接时，通常其线程也被销毁。如果该选项设置为正值 N，那么连接断开后 N 个线程将被缓存起来。在具有良好线程实现的系统上，该选项不能显著地提高性能，但是，对于一个应用使用成百上千连接的情况，还是很有用的。

thread_stack

每个线程的栈大小。如果该变量设置过小，将会限制 SQL 语句的复杂性、存储过程的递归深度，以及服务器上其他内存消耗型的操作。对于大部分安装来说，默认值（32 位系统是 192KB；64 位系统是 256KB）就可以。如果有类似"Thread stack overrun"的错误消息，请增大该参数。

tmp_table_size

内存中，内部临时表的最大值。服务器默认设置为 max_heap_table_size 和 tmp_table_size 二者中的最小值。如果你有足够的内存，并且 Created_tmp_disk_tables 状态变量在增大，请增大该变量。把需要临时表的所有结果集放在内存中，可以大大提高性能。

query_cache_size

MySQL 服务器存储查询及其结果集的缓存大小。增大该值，可以提高性能，因为查询插入到缓存里，接下来，招行相同的查询不需要查询解析、优化和执行，就可以从缓存中取到结果集。但是，该变量不要设置过大，因为当需要从缓存中删除查询时，即修改表数据，互斥体争用将阻塞并行查询，尤其是多核计算机和高并发环境下，超过 8 个用户会话并行访问查询缓存。该变量的合理值是小于 100MB，尽管可以接受突然宕机而设置得大一点。

提示

最佳实践是 query_cache_size 设置得偏小些，使用 FLUSH QUERY CACHE 定期整理碎片，而不是增大该值。

为了确定查询缓存是否有效，可以查看 Qcache_free_blocks、Qcache_free_memory、Qcache_hits、Qcache_inserts、Qcache_lowmem_prunes、Qcache_not_cached、Qcache_queries_in_cache 和 Qcache_total_blocks 状态变量。

table_definition_cache

存储在缓存中的表定义的数量。当表数量很大时，可以增大该值。如果需要，可以调整该值，以便最近的表刷新（FLUSH TABLES）后，保持 Opened_table_definitions 小于或等于 Open_table_definitions。

table_open_cache

存储在缓存中的表描述符的数量。调整该值，以便 Opened_tables 仍小于或等于 Open_tables。

2. 控制优化器的选项

这些变量可以在 SESSION 级别设置，所以，你可以实验它们是怎样影响特定查询的。

optimizer_prune_level

如果该变量设为 on，优化器删除即时搜索发现的不太有效的计划；如果设置为 off，优化器使用详尽的搜索。默认值为 1（on）。如果你怀疑优化器选择的不是最优计划，你可以改变它的值。

optimizer_search_depth

优化器搜索的最大深度。该值越大，优化器越有可能为复杂的查询找到最优计划。提高该值的代价就是优化器在搜索计划时的时间开销增大。如果设置为 0，服务器会自动选择一个合理的值，默认值为 62（最大值）。

optimizer_switch

该变量控制各种优化器特性，在这里略提一下。合理使用该参数，需要知道优化器的工作原理并且具有丰富的经验。

index_merge

启用或禁用索引合并优化，该优化几个从合并扫描中获取行记录，并把结果合并为一条记录。在 EXPLAIN 的结果输出中，Merge 列显示的就是这个选项。

index_merge_intersection

启用或禁用索引合并交叉访问算法，当 where 从句包含 key 表示的范围条件并且与 AND 关键字时，将会使用该算法。如：

key_col1< 10 AND key_col2 = 'foo'

即使 key_col2 = 'foo'得到唯一值，优化器也把它当做范围条件，参见 MYSQL 手册中的“The Range Access Method for Single-Part Indexes”一节，（http://dev .mysql .com /doc / refman/5.5/en/range-access-single-part.html）。

index_merge_union

启用或禁用索引合并联合访问算法，当 where 从句包含 key 表示的范围条件和 OR 关键字时，将会用到该算法。如：

```
key_col1 = 'foo' OR (key_col2 = 'bar' AND key_col3 = 'baz')
```

index_merge_sort_union

启用或禁用索引合并排序联合访问算法，当 where 从句包含 key 表示的范围条件和 OR 关键字时，将使用该算法，但是，不会应用索引合并联合访问算法，如：

```
(key_col1 > 10 OR key_col2 = 'bar') AND key_col3 = 'baz'
```

max_join_size

对于估计可能超过一定限制的 select 语句，（该选项）阻止优化器进行优化（如，查看多于 max_join_size 的行记录）。当调试找出没有使用索引的查询时，该选项会有很大帮助。

max_length_for_sort_data

当无法使用索引时，如果对 ORDER BY 进行条件优化，MySQL 将使用文件排序算法。该算法有两个变体。原始算法读取所有匹配的行记录，在缓存里存储战对的键值和行指针，该缓存大小受 sort_buffer_size 限制。缓存中的值排序后，该算法再次读取表记录，但是，这次按照一定顺序读取。该算法的缺点是两次读取行记录。

改进后的方法是读取整个行记录到缓冲区，然后排序键值，从缓冲区中读取行记录。该方法的问题是结果集通常超过 sort_buffer_size，所以，对于大数据集，磁盘 I/O 操作使得该算法很慢。max_length_for_sort_data 变量限制键值和行记录指针对的大小，因此原始算法适用于键值和行指针对中额外列的总大小超过了该限制的情况。

磁盘活动多，并且 CPU 活动少是需要调低该变量的一个信号。

更多详情可查看 MYSQL 手册中的“ORDER BY Optimization”部分。

max_seeks_for_key

根据表扫描必须检查的记录行数量，为使用键值而不是表扫描设置阈值。设置该参数为一个较小的值，比如 100，在表扫描时，可以强制优化器优先查看索引。

max_sort_length

设置对 BLOB 或 TEXT 值排序时用到的初始字节数，后面的部分将被忽略。

3. 与引擎相关的选项

这部分变量将会影响特定存储引擎的性能，本书只考虑 InnoDB 和 MyISAM 的选项。

InnoDB 选项。如同之前一样，首先介绍 InnoDB 存储引擎的选项。

innodb_adaptive_hash_index

禁用或启用（默认）InnoDB 自适应散列索引，很多情况下，启用该选项是有好处

的，但是存在少量已知的例外，自适应散列索引可能降低性能。例如，当类似的查询结果数量巨大，并且该索引占用30%或更多缓冲池时。可以在InnoDB监控器的输出里查看该信息，因为InnoDB可能会改进，所以这里不全部描述它们，不过，如果你遭遇到糟糕的性能，建议搜索Wed上的实际测试用例。

innodb_additional_mem_pool_size

该缓冲池存放数据字典和内部数据结构信息。一般情况下，表越多，该选项也应该越大，因为当该选项过小时，InnoDB会在错误日志中记录错误消息，所以你可以等看到错误日志再调整该值。

innodb_buffer_pool_size

InnoDB为存储数据、索引、表结构、自适应散列索引等分配的内存大小，这是影响innodb性能最重要的选项。可以将其设置为物理内存的80%。理想情况下，该缓冲区足够大，以致可以包含所有活动的InnoDB表和额外空间。同时，也要把其他缓冲区计算在内，寻找一个好的平衡。

匹配Innodb_buffer_pool_%的，状态变量显示InnoDB缓冲池的当前状态。

innodb_buffer_pool_instances

该选项设置InnoDB缓冲池应切分的实例数量。每个实例有它自己的空闲列表、刷新列表、使用LRU算法的存储对象的列表，以及其他数据结构，并且受到自身互斥体的保护。设置该变量大于1，可以提高大型系统的并发性能。每个实例的（缓冲区）大小是innodb_buffer_pool_size/innodb_buffer_pool_instances，并且至少是1GB。如果innodb_buffer_pool_size小于1GB，该选项不生效。

innodb_buffer_pool_instances切分缓冲区互斥体，所以，如果有8个或者更多并行SESSION同时访问InnoDB缓冲池，该选项可以设置为4～16。该选项取决于innodb_buffer_pool_size值和可用的内存。

innodb_checksums

默认情况，InnoDB使用校验和验证磁盘上所有页。该选项可以立即确定数据文件是否由于磁盘坏道或者其他原因而损坏。通常需要开启该功能，但是，在很少的情况下，当不关心数据时（比如，只读从服务器，不提供（线上）服务，只做备份），关闭该功能，可以提升性能。

innodb_commit_concurrency

可以同时提交事务的线程数量，默认值为0（没有限制）。

innodb_thread_concurrency

InnoDB内部同时运行的线程数量，不要把它和MySQL服务器创建的连接线程的

数量混淆。默认值是 0：不限制并行或不检查并行。

尽管大量线程并行运行一般意味着高性能，但是，如果同时并行运行很多用户会话，你可能遇到互斥体争用，如果同时运行的用户线程不超过 16 个，通常不用担心该参数。如果有更多用户 SESSION，可以通过查询 Performance Schema 或者 SHOW ENGINE INNODBMUTEX 来监控互斥锁。

如果出现互斥体争用，可以尝试限制该变量为 16 或 32，或者把 mysqld 进程放置到 Linux 的任务集里或者 Solaris 的处理器集里，同时限制它为更少的内核而不是所有内核。最佳实践是使用超过 8 核的系统。或者，也可以使用线程池插件（见以下边栏）。

线程池插件

从 5.5.16 版本开始，MySQL 商业发行版包含了线程池插件。

默认情况下，MySQL 为每个用户连接创建新线程。如果创建大量用户连接，那么并行运行大量线程时，上下文切换的开销变得很高，这会导致资源争用。例如，对 InnoDB 而言，这会增加持有互斥体的时间。

线程池插件为处理线程提供了一种替代方案。它把所有连接线程按组编排，其数量受到 thread_pool_size 变量的制约，这样可以确保在任何时候[1]每组只有一个线程在运行。这个模型可以降低开销和大大提升性能。

可以在 MYSQL 中手册获取线程池插件的更多信息。

---

innodb_concurrency_tickets

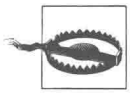
当允许一个线程进入 InnoDB 时，它接收 innodb_concurrency_tickets 张并发“票”（ticket），这些票允许线程离开和重新进入 InnoDB，直到它使用完这些票。

默认值是 500。用完这些票后，把线程放置到等待队列中，以获取一组新票。

innodb_doublewrite

默认情况下，InnoDB 分两次存储数据：第一次写入双写缓冲，第二次写入数据文件。像 innodb_checksums 一样，对于数据安全不是最重要的场景，这个安全选项可以关闭 off，以提升性能。

1：这不是硬性限制，有事每组不止一个线程在执行。



警告

当设置时，innodb_doublewrite 变量可以防止 InnoDB 数据损坏，因此，除非绝对需要，不要设置为 off。

Innodb_dblwr_writes 和 Innodb_dblwr_pages_written 状态变量分别显示两次写操作的数量和写页的数量。

innodb_flush_log_at_trx_commit

定义何时把更改写入（重做）日志文件以及刷新到磁盘。如果设置为 1（默认值），在每个事务提交时更改均会写入和刷新到磁盘。为了得到更好的性能，可以设置该值为 0（每秒写入日志文件和刷新到磁盘，而每个事务提交时，不做操作）或者 2（每次提交事务时写入日志文件，但是，每秒刷新到磁盘）。注意，只有该选项为 1 时才符合 ACID 事务要求的。

Innodb_os_log_fsyncs 状态变量存储 fsync()到日志文件的操作次数。Innodb_os_log_pending_fsyncs 包含挂起的 fsync()写次数。Innodb_log_writes 和 Innodb_os_log_pending_writes 分别包含写入次数和挂起的写次数。

innodb_flush_method

默认情况下，fdatasync()是用来刷新数据文件的，fsync()是用来刷新日志文件到磁盘的，该值可以更改为以下值中的一个：

O_DSYNC

操作系统使用 O_SYNC 打开和刷新日志文件，同时使用 fsync()刷新数据文件。

O_DIRECT

操作系统使用 O_DIRECT 打开数据文件，并且使用 fsync()刷新数据文件。

更改 innodb_flush_method 变量的值，或者提升性能或者降低性能，所以，在（生产）环境中，需要谨慎测试它。

innodb_io_capacity

后台 InnoDB 任务执行的 I/O 活动的上限。对于大多数现代系统来说，默认值 200 是个不错的选择。但是，可以根据系统的 I/O 吞吐量，调整该值。在快速存储器上，增大该值，才有意义。

innodb_log_buffer_size

InnoDB 用来把日志文件写入磁盘的缓冲区大小。当缓冲区满时，必须等待日志刷新到磁盘上后，才能继续进行操作。增大该参数，可以减少磁盘 I/O 操作，但是，只有在存在大量事务时，这才有意义。

状态变量 Innodb_log_waits 包含缓冲区因太小而需要的 I/O 等待次数。

innodb_log_file_size

每个日志文件的大小。大日志文件降低检查点的活动，节省磁盘 I/O。但是，大日志文件显著地延缓崩溃恢复过程[1]。从 1MB 到不超过 innodb_buffer_pool_size/log_files_in_group 的值是有合理的。所有日志文件加起来不能超过 4GB。

最佳实践是在不同的磁盘上存储日志文件、数据文件、如果使用的话，还有二进制日志文件，这样，即使一个设备故障，也不会同时丢失所有文件。

innodb_open_files

仅当使用 innodb_file_per_table 时，该变量才有意义。innodb_open_files 是 InnoDB 可以同时打开的.ibd 文件的数量。默认值是 300，该值增大到 InnoDB 所有的表的数量是有意义的。

innodb_read_io_threads

InnoDB 读操作可以使用的 I/O 线程数。这些操作处理预读：I/O 请求以异步方式将一组页数据装进 InnoDB 的缓冲池，然后清空和插入缓冲操作。默认值是 4。

innodb_write_io_threads

Innodb 从缓冲池中写脏数据的 I/O 线程数量。默认值是 4。

innodb_stats_method

服务器在收集索引值的统计信息时，处理 null 的方式。这会影响索引的基数，优化器因此生成的查询计划。

innodb_stats_on_metadata

启用该变量（默认值），每次执行元信息语句时，比如，SHOW TABLE STATUS 或者 SHOW INDEX，或者当查询 InnodB 表的 INFORMATION_SCHEMA 或统计数据时，innoDB 都会更新统计信息。如果启用该变量，这些查询将和每次查询后执行 ANALYZE TABLE 有一样的效果。如果服务器频繁调用这些语句或者查询具有大量表的数据库时，可以禁用该变量。但是，当禁用该变量时，表统计数据将过时。

innodb_stats_sample_pages

MYSQL 优化器用来计算索引分布统计信息的抽样索引页的数量，例如调用 ANALYZE TABLE。如果你怀疑基数计算不合理，请增大该值（默认值是 8）。但是，如果启用 innodb_stats_on_metadata，增大该值，打开表的时间会增加。

1：这并不总是 100%正确，因为 InnoDB Plugin 1.0.7 引入了加速崩溃恢复措施。

MyISAM 选项。这一节将讨论哪些选项会影响 MyISAM 存储引擎的性能。

myisam_max_sort_file_size

当 MyISAM 重建索引时，使用的临时文件的最大值。默认值是 2GB。如果超过该值，MySQL 会使用健缓存，这会降低重建索引的速度。临时文件是磁盘文件，所以，它受到磁盘空间的限制。

myisam_use_mmap

当设置该变量时，MySQL 服务器在读写 MyISAM 表时使用内存映射机制，默认对于这些操作使用系统调用。尽管 myisam_use_mmap 经常大大提高性能，但是，存在几个已知的 bug，所以，设置该变量后，要测试应用程序。

myisam_mmap_size

内存映射压缩的 MyISAM 表可以使用的最大内存值。默认值很大：对于 32 位系统，4 294 967 295；对于 64 位系统，18 446 744 073 709 547 520。如果使用很多压缩的 MyISAM 表，减小该变量（的值），可以避免交换（内存）。

myisam_sort_buffer_size

REPAIR TABLE、CREATE INDEX 或 ALTER TABLE 操作期间，排序或者创建 MyISAM 索引时，分配的缓冲区大小。

myisam_stats_method

收集索引值统计信息时，服务器处理 NULL 的方式。这影响索引的基数，以及优化器因此生成的查询计划。

bulk_insert_buffer_size

MyISAM 批量插入时使用的特殊树状缓存的大小，包括：INSERT …SELECT、INSERT … VALUES (…), (…), …和 LOAD DATA INFILE 语句。

key_buffer_size

缓存 MyISAM 表的索引块，并在线程之间共享（这些索引块）。该变量控制这个缓冲区的大小。可以创建多个键缓冲区，请在 MySQL 参考手册中查找和阅读该变量的描述信息。

preload_buffer_size

用来预加载索引的缓冲区大小。

3.9.3 计算选项的安全值

通过增大缓冲区或最大值，试图优化服务器性能时，全面考虑内存使用情况是很重要

的。大量缓冲区能因“内存不足”错误而导致服务器崩溃。这一节将提供一些公式，帮助计算是否超出了可用内存。这部分不介绍选项本身，可以参考前面的章节或者MySQL 手册，获取详细介绍。计算依赖该选项何时分配，以及是否共享，所以，本节把它分成几个相关类别。

1. 服务器级选项

这些选项是 GLOBAL 的，影响所有连接和查询，一部分是服务器启动时分配的，而另一部分是后来分配的，比如查询缓存，初始值是 0，（后续）不断增长，直至最大值。MySQL 服务器达到所有的限制，分配所有允许的内存，会花费很长时间。因此，需要计算 mysqld 获得的 RAM 大小，和所有的缓冲区大小保证不要超过它。

下面是服务器级内存缓冲区分配选项列表：

- query_cache_size
- innodb_additional_mem_pool_size
- innodb_buffer_pool_size
- innodb_log_buffer_size
- key_buffer_size

使用下面的公式计算需要为这些缓冲区分配多少 RAM（单位是 MB）。

```
SELECT (@@query_cache_size + @@innodb_additional_mem_pool_size +
@@innodb_buffer_pool_size + @@innodb_log_buffer_size + @@key_buffer_size)/(1024*1024);
```

服务器还有限制文件描述符数量和缓存线程数量的选项，该计算可以忽略它们，因为分配给它们的内存仅仅是系统的一个指针大小乘以分配项的数量，这个数量足够小，以至于在现代系统上可以忽略。在这里列举一下，仅供参考：

- thread_cache_size
- table_definition_cache
- table_open_cache
- innodb_open_files

2. 线程级选项

这些选项是基于每个线程分配（内存）的，这样，服务器分配 max_connections*sum（thread options）。设置 max_connections 和这些选项，保证物理内存总量 – max_connections* sum（thread options）– 服务器级选项大于 0。留一些内存给第三类选项和后台操作，这些操作不受这些变量控制。

线程级选项列表如下：

- net_buffer_length
- thread_stack
- query_prealloc_size
- binlog_cache_size
- binlog_stmt_cache_size

使用下面的公式计算需要为它们分配多少内存（单位是 MB）：

```
SELECT @@max_connections * (@@global.net_buffer_length + @@thread_stack +
@@global.query_prealloc_size + @@binlog_cache_size + @@binlog_stmt_cache_size) /
(1024 * 1024)
```

或者，如果使用 5.5.9 之前的版本（binlog_stmt_cache_size 变量在这个版本中引入）。

```
SELECT @@max_connections * (@@global.net_buffer_length + @@thread_stack +
@@global.query_prealloc_size + @@binlog_cache_size) / (1024 * 1024)
```

3. 为特定操作分配的缓冲区

当服务器执行特殊操作时，根据需要分配的缓冲区。很难计算出分配的内存大小。分析查询，找出哪些需要很多资源，然后像下面这样计算：

缓冲区大小*为特定查询分配的缓冲区数量*并行执行的查询数量

对于所有变量，这样计算，求出总和。

只要对于大多数查询是足够的，保持这些选项（的值）小一点。如果一个特定查询需要更多内存，只对于这个会话增加这个变量的值。例如，对于一个每周执行一次的统计查询，如果需要设置 max_join_size 很高，GLOBAL 级设置它是没有意义的，运行该查询前，请设置该参数。即使这样谨慎，也不要忘记从 GLOBAL 出发使用内存。

以下选项针对每个线程分配一次（缓冲区）：

- read_rnd_buffer_size
- sort_buffer_size
- myisam_mmap_size
- myisam_sort_buffer_size
- bulk_insert_buffer_size
- preload_buffer_size

对于每个线程分配多次（缓冲区）的选项是：

- join_buffer_size
- read_buffer_size
- tmp_table_size

可以使用以下公式计算 MySQL 为每个选项分配的内存的最大数量（单位是 MB）：

```
set @join_tables = YOUR_ESTIMATE_PER_THREAD;
set @scan_tables = YOUR_ESTIMATE_PER_THREAD;
set @tmp_tables = YOUR_ESTIMATE_PER_THREAD;

SELECT @@max_connections * (@@global.read_rnd_buffer_size +
@@global.sort_buffer_size + @@myisam_mmap_size +
@@global.myisam_sort_buffer_size + @@global.bulk_insert_buffer_size +
@@global.preload_buffer_size + @@global.join_buffer_size * IFNULL(@join_tables,
1) + @@global.read_buffer_size * IFNULL(@scan_tables, 1) +
@@global.tmp_table_size * IFNULL(@tmp_tables, 1)) / (1024 * 1024)
```

从这个公式中去除不适合你的环境的选项。

为了总结本章内容，这里给出一个综合的公式，计算 MySQL 安装时最多可以使用的内存（单位是 MB）：

```
set @join_tables = YOUR_ESTIMATE_PER_THREAD;
set @scan_tables = YOUR_ESTIMATE_PER_THREAD;
set @tmp_tables = YOUR_ESTIMATE_PER_THREAD;

SELECT (@@query_cache_size + @@innodb_additional_mem_pool_size +
@@innodb_buffer_pool_size + @@innodb_log_buffer_size + @@key_buffer_size +
@@max_connections * (@@global.net_buffer_length + @@thread_stack +
@@global.query_prealloc_size + @@global.read_rnd_buffer_size +
@@global.sort_buffer_size + @@myisam_mmap_size +
@@global.myisam_sort_buffer_size + @@global.bulk_insert_buffer_size +
@@global.preload_buffer_size + @@binlog_cache_size +
@@binlog_stmt_cache_size + @@global.join_buffer_size * IFNULL(@join_tables,
1) + @@global.read_buffer_size * IFNULL(@scan_tables, 1) +
@@global.tmp_table_size * IFNULL(@tmp_tables, 1))) / (1024 * 1024)
```

或者，对于 5.5.9 之前的版本，对应的公式如下所示。

```
set @join_tables = YOUR_ESTIMATE_PER_THREAD;
set @scan_tables = YOUR_ESTIMATE_PER_THREAD;
set @tmp_tables = YOUR_ESTIMATE_PER_THREAD;

SELECT (@@query_cache_size + @@innodb_additional_mem_pool_size +
@@innodb_buffer_pool_size + @@innodb_log_buffer_size + @@key_buffer_size +
@@max_connections * (@@global.net_buffer_length + @@thread_stack +
@@global.query_prealloc_size + @@global.read_rnd_buffer_size +
@@global.sort_buffer_size + @@myisam_mmap_size +
@@global.myisam_sort_buffer_size + @@global.bulk_insert_buffer_size +
@@global.preload_buffer_size + @@binlog_cache_size +
@@global.join_buffer_size * IFNULL(@join_tables,1) + @@global.read_buffer_size *
IFNULL(@scan_tables, 1) + @@global.tmp_table_size * IFNULL(@tmp_tables, 1))) /
(1024 * 1024)
```

请注意，只有当变量值足够小时，公式才起作用。如果变量值很大，要么转变每个变量为 MB，要么强制把它们转换为无符号整数。如果任何变量超过最大的无符号整数值（18 446 744 073 709 547 5207），即使强制转换为无符号整数也无济于事。不考虑可能的溢出，因为我要可读的和清晰的公式。如果你用不到那些缓冲区或特性，请从公式中去掉，这也是有意义的。例如，不使用 myisam_mmap_size 的默认值，取而代之的是使用线程可用的 MyISAM 表的最大值。

第 4 章

MySQL 环境

MySQL 服务器在运行环境中不是孤立的。即使它运行在专用服务器环境中，你依然需要考虑硬件资源和操作系统限制。在共享环境中，MySQL 服务器仍然会受到其他进程的影响。关于 MySQL 操作系统层面的调优，这个主题可以另写一本书了。所以本章不深入讨论，只是从排错角度切入并展开。MySQL 的其中一个优势就是能运行在不同环境中，但这也导致了本章很难将其细化说明。最终我决定写下你需要关心的部分，剩下的留给你去查找操作系统对应的手册来决定如何调优。

4.1 物理硬件限制

对性能有不切实际的期望是通常会犯的错。我们要求 MySQL 服务器在忽略硬件组件延迟的情况下进行优化。因此，理解什么导致了延迟很重要。

下面列出了影响 MySQL 服务器的硬件资源：

- 内存
- CPU
- 内核数量
- 磁盘 I/O
- 网络带宽

我们依次讨论每个细节。

4.1.1 内存

内存对于 MySQL 是非常宝贵的资源。服务器在没有磁盘交换的情况下运行很快。理想情况是数据驻留在内存中。因此，在物理内存的限制内合理配置缓冲区是非常重要的。

关于这一点，1.6.4 小节和 3.9.3 小节提供了更多详细的介绍和指导。

提示

在 Linux/UNIX 上可以通过 vmstat，或者 Windows 上的任务管理器来查看 mysqld 是否有大量磁盘交换。

这里列出 Linux 上一个磁盘交换的例子。重要的部分加粗显示。对于没有磁盘交换的服务器，这些值应该为 0:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 1  0   1936 296828   7524 5045340    4   11   860     0  470  440  4  2 75 18
 0  1   1936 295928   7532 5046432   36   40   860   768  471  455  3  3 75 19
 0  1   1936 294840   7532 5047564    4   12   868     0  466  441  3  3 75 19
 0  1   1936 293752   7532 5048664    0    0   848     0  461  434  5  2 75 18
```

相关章节讨论了影响内存使用的配置变量。首先计算出 MySQL 服务器需要用到的最大内存量，基本原则是要保证它小于你的物理内存。当缓冲区大于实际内存时会增加 MySQL 服务器因为“内存不足”错误而崩溃的风险。

- 前面的观点可以换一种陈述方式：如果你需要更大的缓冲区，则需要购买更多内存。扩展应用程序时这始终是一个很好的做法。
- 使用的内存模块支持扩展错误校正（EEC），这样即使内存有一些损坏，MySQL 服务器也不会因此崩溃。

关于内存使用的更多内容，可以参考 MySQL 参考手册中的“MySQL 如何使用内存”章节。我就不在此处重复，因为它不涉及任何新排错技巧。

很重要的一点是，当查询的行包含 BLOB 字段时，内部缓冲区会增长到能存储该值的点，并且存储引擎在查询结束后不会释放内存。需要执行 FLUSH TABLE 来释放内存。

另一个要点涉及 32 位和 64 位架构的不同。虽然 32 位使用较小的指针大小可以节约内存，但由于操作系统的寻址受限，它在缓冲区大小设置上存在固有的约束。理论上，每个进程在 32 位系统上的最大可用内存为 4GB，实际上在很多系统上这个数值很小。所以如果你想使用的缓冲区在 32 位系统上超过了这个值，考虑转换到 64 位架构下。

4.1.2 处理器与内核

MySQL 的性能不会随着 CPU 的速度线性增长。这并不意味着你不能使用一块快的 CPU，但不要期望提高 CPU 速度能像增加内存那样性能会有大幅提升。

然而，当设置一些影响内部线程并发的选项时，内核的数量很重要。如果你没有足够的内核数增加这些值是没有任何意义的。通过使用一款名叫 sysbench[1]的基准工具能够

1：6.9.1 小节会进一步讨论 sysbench。

轻松说明问题。表 4-1 显示了在四核机器上的一个测试结果。我使用具有 16 个线程的 OLTP sysbench 测试。

表 4-1　不同 innodb_thread_concurrency 值下执行事件消耗的时间（s）

innodb_thread_concurrency	执行时间
1	7.8164
2	4.3959
4	2.5889
8	2.6708
16	3.4669
32	3.4235

如你所见，在启动 8 个线程之前测试越运行越快，但随着数值的增加停止了增长。

4.1.3　磁盘 I/O

快速磁盘对 MySQL 性能很重要。磁盘越快，I/O 性能越好。

关于磁盘，你需要关注磁盘读延迟（每次读访问需要多长时间）和 fsync 延迟（每个 fsync 耗时多少）。

近来固态硬盘（SSD）的性能很好，但不要指望在上面出现奇迹，因为大多数存储引擎是针对硬盘读写优化的。

同样网络存储也是。可以将数据和日志文件放在网络文件系统或者存储上，但这些装置可能会慢于本地磁盘。你需要检查存储是否快速并且可靠，否则，如果由于网络故障引起数据丢失请不要感到意外。

在 Linux/UNIX 上使用 iostat 来判断磁盘 I/O 是否过载。向设备发起的请求平均队列长度正常情况下不会很高。出于同样的目的，你可以在 Windows 平台上使用 perfmon。下面给出 iostat 的一个输出例子：

```
$iostat -x 5
Linux 2.6.18-8.1.1.el5 (blade12)        11/11/2011      _x86_64_

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.27    0.00    0.32    0.65    0.00   97.79

Device:    rrqm/s wrqm/s  r/s   w/s   rsec/s  wsec/s avgrq-sz avgqu-sz await
cciss/c0d0   0.02   7.25  0.50  6.34  14.35   108.73   17.98    0.48   69.58
dm-0         0.00   0.00  0.52 13.59  14.27   108.70    8.72    0.09    6.59

svctm  %util
2.22   1.52
1.08   1.52

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          38.69    0.00    6.43   47.84    0.00    8.64
```

```
Device:    rrqm/s  wrqm/s  r/s     w/s rsec/s   wsec/s avgrq-sz avgqu-sz  await
cciss/c0d0   0.00 5362.40 0.20  713.80   1.60 51547.20    72.20   138.40 193.08
dm-0         0.00    0.00 0.00 6086.00   0.00 48688.00     8.00  1294.74 227.04

svctm %util
1.40 99.88
0.16 99.88

<skipped>

Device:    rrqm/s   wrqm/s  r/s     w/s  rsec/s   wsec/s avgrq-sz avgqu-sz  await
cciss/c0d0   0.00 10781.80 0.00   570.20  0.00  84648.00   148.45   143.58 248.72
dm-0         0.00     0.00 0.00 11358.00  0.00  90864.00     8.00  3153.82 267.57

svctm  %util
1.75 100.02
0.09 100.02

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           8.90    0.00   11.30   75.10    0.00    5.60

Device:    rrqm/s   wrqm/s  r/s     w/s  rsec/s   wsec/s avgrq-sz avgqu-sz  await
cciss/c0d0   0.00 11722.40 0.00  461.60    0.00 98736.00   213.90   127.78 277.04
dm-0         0.00     0.00 0.00 12179.20   0.00 97433.60     8.00  3616.90 297.54

svctm %util
2.14 98.80
0.08 98.80

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          23.55    0.00   23.95   46.19    0.00    7.11

Device:    rrqm/s  wrqm/s  r/s     w/s  rsec/s   wsec/s avgrq-sz avgqu-sz  await
cciss/c0d0   0.00 4836.80 1.00  713.60    8.00 49070.40    68.68   144.28 204.08
dm-0         0.00    0.00 1.00 5554.60    8.00 44436.80     8.00  1321.82 257.28
svctm  %util
1.40 100.02
0.18 100.02
```

以上输出取自 mysqld 空闲并随后启动一个活跃的 I/O 作业时。你可以看到 avgqu-sz 是如何增长的。虽然还不至于带来问题，但我决定在这里通过这个例子来说明当 mysqld 正在处理它的作业时磁盘 I/O 活动是如何发生变化的。

除了速度之外，要牢记的是，存储也可能会丢失数据，可能出现只写入一部分页的情况。如果使用 InnoDB，为了数据安全通过设置 innodb_double_write 使用双写缓冲区。同样，为磁盘准备备用电池是很重要的，它可能防止因为电源故障导致的数据丢失。

4.1.4 网络带宽

客户端几乎总是通过网络连接 MySQL 到服务器的，所以 MySQL 服务器运行在一个快速的网络环境下是很重要的。

除了网络带宽之外，往返时间（RTT）和往返次数也很重要。RTT 指的是客户端发送网

络包到服务端返回应答所消耗的时间。主机间隔的距离越长，RTT 越高。

也正是因为存在网络带宽和 RTT 这些原因，如果可以，推荐将客户端和服务器放在同一个本地网络中。

复制也推荐在本地网络中进行。连接到从服务器时可以使用因特网来替代本地内联网，但预期会出现延迟，甚至中继日志数据损坏带来的报错。上面的错误应该在 bug #26489 bug 修复之后会自动修复，前提是你在 5.5 版本中配置了 relay-log-recovery 选项，或者从 5.6 版本开始使用二进制日志校验和。但由于网络故障主从服务器之间仍然会在重发包上消耗时间。

4.1.5 延迟效应的例子

为了结束本章内容，我用一个小例子来说明硬件延迟是如何影响一条普通的 UPDATE 查询的。我们使用 InnoDB 表并将自动提交打开，同时打开二进制日志文件。

```
UPDATE test_rrbs SET f1 = md5(id*2) WHERE id BETWEEN 200000 AND 300000;
```

这个简单的查询会在下面的情况下遭遇延迟：

客户端发送命令到服务端历经半个 RTT。

执行 UPDATE 的 WHERE 子句，mysqld 读磁盘。

由于自动提交开启 mysqld 会对此事务做一个 fsync 调用。

为写入二进制日志文件 Mysqld 做一次 fsync 调用。

为提交改变 mysqld 做一次 fsync 调用。

客户端收到来自服务器的结果，这是 RTT 的另一半。

4.2 操作系统限制

操作系统在 MySQL 上会存在其他限制。例如，我曾经见过一个案例，服务器故障是因为 Linux 主机设置了 vm.overcommit_ratio = 50。这个参数指定系统经由 malloc()可分配的全部虚拟内存百分比，一个进程尝试分配更多内存时将会失败。50%是许多 Linux 安装时的默认值。所以 mysqld 分配现有内存的 50%，当它试图分配更多的时候就失败了。在多任务配置中，这个选项有利于防止其他关键进程受 MySQL 服务器影响，但在专有服务器环境中这个选项显然是很荒谬的。

另外一个重要的 UNIX 选项是 ulimit，它用来限制用户各方面的资源[1]。当使用 ulimit

1：虽然使用专有的平台工具来设置操作系统的资源限制是正确方式，但值得一提的是内置的 shell 命令 ulimit，它简单易用并且适用于所有用户。运行 ulimit -a 可以显示所有当前限制，或者为当前用户设置软限制。

命令限制资源或者其他系统工具时，记住 MySQL 服务器也是运行在一个用户下，这些限制对任何人都生效。

3.9.1 节提到了操作系统限制是如何影响 open_files_limit 变量的，我在这里举例说明一下。假设服务器的 ulimit 限制打开文件（-n 选项）为 1024，大多数系统的默认值。如果你试图以--open-files-limit=4096 来启动 mysqld，它并不会重写操作系统限制。

```
$ulimit -n
1024

$./bin/mysqld --defaults-file=support-files/my-small.cnf --basedir=.
--datadir=./data --socket=/tmp/mysql_ssmirnova.sock --port=33051 --log-error
--open-files-limit=4096 &
[1] 31833

$./bin/mysql -uroot -S /tmp/mysql_ssmirnova.sock -e "SELECT
@@open_files_limit"
+--------------------+
| @@open_files_limit |
+--------------------+
|               1024 |
+--------------------+
```

如果你有很多表，那么这个选项就非常重要。如果此值设置过小，那么 MySQL 服务器就会在打开表和关闭表操作上消耗大量时间，从而拖慢运行速度。如果服务器由于缺少资源不能打开新表，甚至会出现拒绝尝试连接的情况。

因为每个操作系统都有具体的特性，我不再针对系统限制调优进行深入讨论。我没有提到 Windows，但请放心，它也有限制。

每当你怀疑操作系统限制了 MySQL 服务器，请先检查之前提到过的资源：内存、CPU、网络带宽。检查 mysqld 的资源是否小于硬件提供的。通常，性能问题出现在内存或 CPU 不够用时。另外，检查允许打开的文件数。

如果你发现 mysqld 应该并且可以使用更多的资源，检查操作系统优化的各个方面。比如，操作系统内核选项或者默认用户在可用内存量上没有限制，但是在运行 mysqld 的用户上做了专门限制。当大多数操作系统限制是很大的或者没有设置时，而用户账户的默认值很小，这是种常见的情况。

如果 MySQL 使用了有限的资源，你有时可能会很奇怪出问题时是由于缺乏资源，还是仅仅是没有用到它们，因为当时负载非常低。当服务器故障时，它要么在错误日志文件中输出消息，要么性能开始急剧下降。另一个迹象是，当你试图增加一个选项时，它并没有变。前面的 open_files_limit 例子说明了这点。在这种情况下，如果在启动时设置该选项，那么你就会在错误日志文件中找到这些消息，或者你会在动态设置该选项时发现一条警告。检查可疑变量的真实值也是一个好方法。

4.3 其他软件影响

目前为止我们讨论的所有问题在 MySQL 服务器运行环境中都很重要。理想情况下，MySQL 应该运行在专有环境中，使用主机的所有物理资源。但是一些站点将 MySQL 用在共享环境下。这包括共享的主机，许多实例运行在 mysqld 服务器上，每个实例分别代表不同的客户，系统同时运行 mysqld 以及客户端应用程序和其他进程。

当对安装在这种配置下的 MySQL 进行调优时，需要额外检查两件事：其他进程在正常操作下使用多少资源，在关键时刻需要分配多少资源。

正常负载下，你可以猜到其他程序在预留给它们后还剩多少资源，适当设置 MySQL 的选项。临界负载通常是出乎意料的，会导致突如其来的 MySQL 错误甚至崩溃。这种状况下没有通用的处理方法。记住，第三方软件可以影响 MySQL 安装，并且分析潜在的负载，采取补救措施。

因此，如果你预料到某一时刻一些应用程序拥有临界负载，评估当前负载下的资源使用情况并根据情况调整 MySQL 服务器选项。在这种环境中，使用操作系统级别的限制是合理的，或者使用虚拟化。对比前面的章节，我建议此处增加限制，而不是删除它们。

最坏的情况是遇到了不可预知的高负载。你只有在碰到问题时才会意识到它的存在。因此，如果 MySQL 开始出问题，而发现这个问题并不该发生，始终记住并发进程的影响，检查操作系统日志，查看是否其他应用程序会影响 MySQL 对资源的访问。在 MySQL 初始化时安装监控软件检测所有进程的运行也是很重要的。

- 确定 MySQL 服务器是否受其他操作系统进程影响的一个好方法是，在一个隔离的环境中运行一条有问题的查询。这也是在 2.5 节中推荐的方法。

第5章

复制故障诊断

前面章节已经提到过复制带来的问题，说明了每章提及的问题是如何在复制环境中产生影响的。本章注重讲述复制本身。大多数复制错误是由于错误或性能下降引起的，如从服务器落后于主服务器几个小时。

MySQL的复制是异步的。这意味着主服务器并不关心从服务器上的数据是否是一致的。尽管可配置成多主环形复制，实际意义上就是一串服务器各自同时担当主服务器和从服务器的角色。

MySQL 多主设置

请参考图 5-1，该图对多主复制进行了说明。

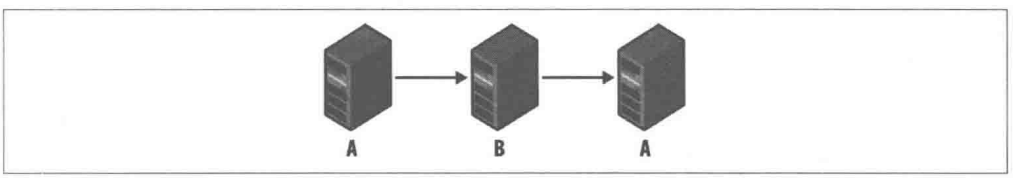


图 5-1　两台服务器互相复制

这里 A 是从服务器 B 的主服务器，同时 B 是从服务器 A 的主服务器。

你可以往这个链中随意增加服务器（见图 5-2）。

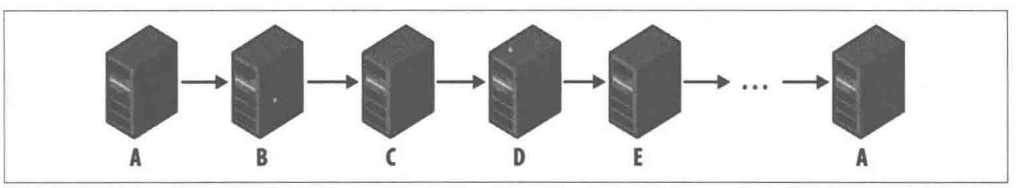


图 5-2　多主环形复制

排查此种配置的步骤，首先挑出一对主从服务器（见图 5-3）。

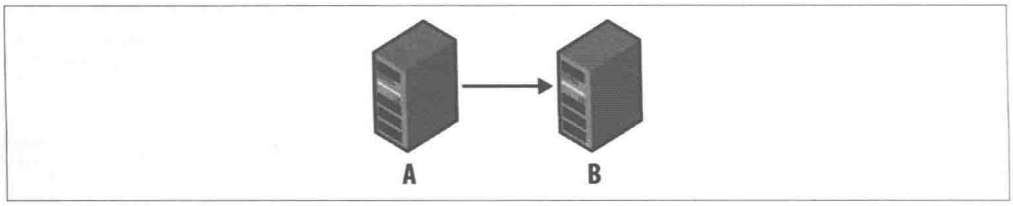


图 5-3　关注多主复制的一个方向

然后像简单的复制环境一样解决。接着挑出另外一对继续。这里不描述其细节，5.3.2 节会简述。

提示

从 5.5 版本开始，MySQL 包中包入了一个半同步复制插件。如果插件开启，主服务器会等待其中一个从服务器发送确认，告知其成功执行了每一个事件。这并不等同于同步复制，因为主服务器并不知道从服务器在执行事件后数据是否相同（2.7.1 节讨论过这种可能性）。此外，如果主服务器连有大量从服务器，并不能保证数据已经复制到所有从服务器上。

排查半同步复制和异步复制一样，唯一的不同体现在具体选项的影响上。这里不针对半同步复制进行具体描述。如果你遇到了问题，按照第 3 章中说的方法做即可。

MySQL 从服务器运行两个与复制相关线程：I/O 线程，它负责处理主服务器过来的所有流量；SQL 线程它重新执行事件，在从服务器上复制结果。两种线程遇到的不同问题，应该使用不同的技巧来解决，在本章分两节来讨论它们。

5.1　查看从服务器状态

在开始对线程排错之前，首先介绍一个查看复制状态信息的有用命令：从库上运行的 SHOW SLAVE STATUS 查询。

下面的例子是在运行正常的从服务器上获取的，当主服务器停止时从服务器上是如何显示错误的。我将分段说明输出。

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Connecting to master
```

这是 I/O 线程的状态。对于一个运行的从服务器，它通常包含 Waiting for master to send event。

```
master to send event:
                  Slave_IO_State: Waiting for master to send event
                     Master_Host: 127.0.0.1
                     Master_User: root
                     Master_Port: 4041
                   Connect_Retry: 60
                 Master_Log_File: mysqld511-bin.000007
```

在这里，“Master_Log_File”这一行显示了主服务器上的二进制日志的文件名，如果此时从服务器发生了 I/O 线程错误，那么这一行将不会显示任何信息。

```
Read_Master_Log_Pos: 106
```

“106”表示从服务器读取的主服务器二进制日志中的位置。

```
Relay_Log_File: mysqld512-relay.000097
```

“Relay_Log_File”是中继日志的名字，是一个在从服务器上包含经主服务器二进制日志转换过内容的文件。

```
Relay_Log_Pos: 255
```

“255”是中继日志的当前位置。

```
Relay_Master_Log_File: mysqld511-bin.000007
     Slave_IO_Running: Yes
```

“Slave_IO_Running”表示 I/O 线程的基本状态，即它是否正在运行。结果或者为 Yes 或者为 No。

```
Slave_SQL_Running: Yes
```

这一次，我们看到的为 SQL 线程的运行状态。同样它或者为 Yes 或者为 No。

```
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 106
```

“Exec_Master_Log_Pos”中的 106 代表执行至主服务器二进制日志的位置。如果从服务器有延迟这个值是不同于 Read_Master_Log_Pos 的。

```
        Relay_Log_Space: 106
        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: 2
```

“Seconds_Behind_Master”显示了从服务器落后主服务器多远。它包含了从服务器上最后执行的事件和中继日志中复制过来的主服务器二进制日志的最后一个事件之间相隔的秒数。理想情况下这个值为0。如果从服务器没有和主服务器连接，这个字段为NULL。

```
Master_SSL_Verify_Server_Cert: No
```

这也是停止的从服务器的输出。

```
Last_IO_Errno: 2013
```

“Last_IO_Errno”要么显示的是I/O线程上最近的错误，要么是0（代表自从服务器启动以来没有产生过错误）。

```
Last_IO_Error: error connecting to master 'root@127.0.0.1:4041'
- retry-time: 60  retries: 86400
```

这两行包含最近的I/O错误的文本。这种情况下，它们包含I/O线程为什么出错的信息。

```
Last_SQL_Errno: 0
```

“Last_SQL_Errno”要么显示的是SQL线程上最近的错误，要么是0（代表自从服务器启动以来没有产生过错误）。

```
Last_SQL_Error:
```

再次说明，SQL错误的文本。虽然这里没有错误，但是这两行也可能包含SQL线程出错的消息。

既然你们已经熟悉了SHOW SLAVE STATUS的输出，下面我们就可以转向排错了。

5.2 与I/O线程有关的复制错误

一般I/O错误包括：

- 从服务器无法连接主服务器；
- 从服务器连接到主服务器，但不断断开连接；
- 从服务器延迟。

当I/O错误出现时，前面章节中我们看到的从服务器状态会变成Slave_IO_Running：No，原因会出现在Last_IO_Errno和Last_IO_Error字段中。错误日志也包含I/O线程出错的消息前提是log_warnings设为1（默认值）。

当从服务器无法连接时，首先在主服务器上检查复制用户是否有正确的权限。复制用户（在开始复制时在CHANGE MASTER查询中指定为master_user的用户）必须在主服务器上拥有REPLICATION SLAVE权限。如果它没有，在主服务器上对该用户授予此权限。

一旦你确认复制用户拥有正确的权限，你就需要检查网络。使用ping工具，来判断主服务器所在的主机是否可以访问。下面是一个例子。

```
$ping 192.168.0.4
PING 192.168.0.4 (192.168.0.4): 56 data bytes
64 bytes from 192.168.0.4: icmp_seq=0 ttl=64 time=0.113 ms
64 bytes from 192.168.0.4: icmp_seq=1 ttl=64 time=0.061 ms
^C
--- 192.168.0.4 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.061/0.087/0.113/0.026 ms
```

如果 ping 无法连接主服务器主机，这清楚地定位了网络有问题并且需要修复它。也可以利用 telnet 来检查 MySQL 服务器是否可以访问。给 telnet 命令指定主服务器的主机和端口参数。

```
$telnet 192.168.0.4 33511
Trying 192.168.0.4...
Connected to apple.
Escape character is '^]'.
>
5.1.59-debug-log}O&i`(D^,#!\o8h%zY0$`;D^]
telnet> quit
Connection closed.
```

例子中，MySQL 服务器是可以访问的：5.1.59-debug-log}O&i`(D^,#!\o8h%zY0$`;D^)是欢迎字符串。如果 ping 正常但 telnet 不能连接到服务器，你需要查明 MySQL 服务器是否正在运行，端口是否可访问，也就是从服务器主机是否可打开主服务器端口，并且主服务器主机是否允许从服务器主机连接到该端口。

如果前面的测试成功，但 I/O 复制线程依然停止，使用复制账户的凭证在 MySQL 命令行连接主服务器，确保能够连接。下面是成功连接上并确认复制用户权限正确的例子：

```
$mysql -h 127.0.0.1 -P 33511 -urepl -preplrepl
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6783
Server version: 5.1.59-debug-log Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

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

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

mysql> SELECT user(), current_user();
+----------------+----------------+
| user()         | current_user() |
+----------------+----------------+
| repl@localhost | repl@localhost |
+----------------+----------------+
1 row in set (0.13 sec)

mysql> SHOW GRANTS\G
*************************** 1. row ***************************
Grants for repl@localhost: GRANT REPLICATION SLAVE ON
*.* TO 'repl'@'localhost' IDENTIFIED BY PASSWORD
'*17125BDFB190AB6350B3AF9B26F9E8F00EA128FE'
1 row in set (0.00 sec)
```

这里 SHOW GRANTS 表明通过从服务器的复制用户的参数可以从主服务器复制数据。

当从服务器可以连接到主服务器但是不停地断开连接时，使用操作系统工具来诊断网络。可以使用 tcpdump 或者 nestat 来监视流量，甚至是经由网络传送大文件并观测进度来确保网络的稳定性。我们的目的是确定主从服务器之间的连接是否有中断。

如果到主服务器的连接已经建立，netstat 的输出如下所示。

```
$netstat -a
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  apple.60344            master.mysql.com.33051 ESTABLISHED
```

tcpdump 将会显示一些包的信息。

```
$tcpdump -i en1 host master.mysql.com and port 33051
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on en1, link-type EN10MB (Ethernet), capture size 96 bytes
22:28:12.195270 IP master.mysql.com.33051 > apple.60344: P
1752426772:1752426864(92) ack 1474226199 win 91 <nop,nop,timestamp 1939999898
649946687>
22:28:12.195317 IP apple.60344 > master.mysql.com.33051: . ack 92 win 65535
<nop,nop,timestamp 649946998 1939999898>
^C
2 packets captured
37 packets received by filter
0 packets dropped by kernel
```

这个例子中主服务器执行了一条查询，它成功地复制了。

当从服务器严重落后主服务器时，这说明从服务器负载过高或者网络慢。本章后面讨论 SQL 线程时再回到过载这个问题上。

为检查网络是否慢，使用 tcpdump 或者发送大文件并观察传送包消耗的时间。也要检查 MySQL 是否利用到了系统提供的所有可用上下行带宽。如果带宽使用量超过了 80%，你有必要购买更快的网络硬件。如果未利用可用带宽，检查是否其他软件占用了同一个网络接口并影响了 MySQL 服务器。如果是其他软件的原因，将它转移到另外一台主机上或者至少更换成另外的硬件网络接口。

另外一个 I/O 线程错误是中继日志损坏。你很可能看到如下 SQL 线程报错。

```
Last_SQL_Errno: 1594
Last_SQL_Error: Relay log read failure: Could not parse relay log event
entry. The possible reasons are: the master's binary log is corrupted (you can
check this by running 'mysqlbinlog' on the binary log), the slave's relay log is
corrupted (you can check this by running 'mysqlbinlog' on the relay log), a
network problem, or a bug in the master's or slave's MySQL code. If you want to
check the master's binary log or slave's relay log, you will be able to know
their names by issuing 'SHOW SLAVE STATUS' on this slave.
```

本节将此讨论这个问题，而不在与 SQL 线程相关的章节，因为造成这个问题的真正原因可能是之前的 I/O 线程错误导致损坏了中继日志。当 SQL 线程在试图执行中继日志中的事件时，一旦遇到损坏的部分就会出现此现象。

在遇到这样一个错误时，首先要做的就是依照错误消息中的指示：使用 mysqlbinlog 工具检查主服务器二进制日志及从服务器中损坏的中继日志。mysqlbinlog 将二进制日志文件转换为可读的格式。只需要这样调用它。

```
$mysqlbinlog /Users/apple/Applications/mysql-5.1/data511/mysqld511-bin.005071
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#110904 16:50:00 server id 511  end_log_pos 106         Start: binlog v 4,
server v 5.1.59-debug-log created 110904 16:50:00
BINLOG '
CIJjTg//AQAAZgAAAGoAAAAAAAQANS4xLjU5LWRlYnVnLWxvZwAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAEzgNAAgAEgAEBAQEEgAAUwAEGggAAAAICAgC
'/*!*/;
# at 106
#110904 16:50:14 server id 511  end_log_pos 192         Query   thread_id=7251
exec_time=0     error_code=0
use test/*!*/;
SET TIMESTAMP=1315144214/*!*/;
SET @@session.pseudo_thread_id=7251/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=1,
@@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=0/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C latin1 *//*!*/;
SET
@@session.character_set_client=8,@@session.collation_connection=8,@@session.
collation_server=33/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
create table t1(f1 int)
/*!*/;
# at 192
#110904 16:50:20 server id 511  end_log_pos 260         Query   thread_id=7251
exec_time=0     error_code=0
SET TIMESTAMP=1315144220/*!*/;
BEGIN
/*!*/;
# at 260
# at 301
#110904 16:50:20 server id 511  end_log_pos 301         Table_map: `test`.`t1`
mapped to number 21
#110904 16:50:20 server id 511  end_log_pos 335         Write_rows: table id 21
flags: STMT_END_F

BINLOG '
HIJjThP/AQAAKQAAAC0BAAAAABUAAAAAAAEABHRlc3QAAnQxAAEDAAE=
HIJjThf/AQAAIgAAAE8BAAAAABUAAAAAAAEAAf/+AQAAAA==
'/*!*/;
# at 335
#110904 16:50:20 server id 511  end_log_pos 404         Query   thread_id=7251
exec_time=0     error_code=0
SET TIMESTAMP=1315144220/*!*/;
```

```
COMMIT
/*!*/;
# at 404
#110904 16:50:36 server id 511  end_log_pos 451         Rotate to
mysqld511-bin.005072  pos: 4
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

此例中我使用了一个有效的二进制日志文件。如果文件被损坏，mysqlbinlog 会明确提示。

```
$mysqlbinlog --verbose --start-position=260 --stop-position=335 \
/Users/apple/Applications/mysql-5.1/data511/mysqld511-bin.000007.corrupted
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
ERROR: Error in Log_event::read_log_event(): 'Found invalid event in binary
log', data_len: 102, event_type: 15
ERROR: Could not read a Format_description_log_event event at offset 4; this
could be a log format error or read error.
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

这里使用了行级二进制日志格式，输出显示行事件看起来大致是这样的。如果使用 binlog_format=statement，所有事件作为 SQL 语句输出。加上—verbose 选项可以看到行级事件的 SQL 表现形式：

```
$mysqlbinlog --verbose --start-position=260 --stop-position=335 \
/Users/apple/Applications/mysql-5.1/data511/mysqld511-bin.005071
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#110904 16:50:00 server id 511  end_log_pos 106         Start: binlog v 4,
server v 5.1.59-debug-log created 110904 16:50:00
BINLOG '
CIJjTg//AQAAZgAAAGoAAAAAAAQANS4xLjU5LWRlYnVnLWxvZwAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAEzgNAAgAEgAEBAQEEgAAUwAEGggAAAAICAgC
'/*!*/;
# at 260
# at 301
#110904 16:50:20 server id 511  end_log_pos 301         Table_map: `test`.`t1`
mapped to number 21
#110904 16:50:20 server id 511  end_log_pos 335         Write_rows: table id 21
flags: STMT_END_F

BINLOG '
HIJjThP/AQAAKQAAACoBAAAAABUAAAAAAAAEABHRlc3QAAnQxAAEDAAE=
HIJjThf/AQAAIgAAAE8BAAAAABUAAAAAAAAEAAf/+AQAAAA==
'/*!*/;
### INSERT INTO test.t1
### SET
```

```
###   @1=1
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

除了--verbose 选项之外，我使用了--start-position 和--stop-position 来说明在日志文件过大的情况下如何限制 mysqlbinlog 输出特定的位置。

可以将 mysqlbinlog 的输出管道至 MySQL 客户端然后执行查询。这适用于 SBR 和基于行的复制（RBR），当你想调试二进制日志事件是如何在从服务器上执行时这也很用。

如果问题出现在主服务器的二进制日志上，找出发生的原因。首先手动重启复制，还原从 Exec_Master_Log_pos 到最近可能的位置的事件，手动执行它们，然后观察 Seconds_Behind_Master 直到 0，比较主从服务器上的表。

如果从损坏到现在变化太大，找出被更改过的行不太现实，你很可能需要备份主服务器，然后在从服务器上加载备份并重启复制。可以对单个表进行复制进行重放。找到二进制日志中数据正确复制的最后一点，然后设置 replicate-wild-do-table 选项并运行：

```
START SLAVE [SQL_THREAD] UNTIL
    MASTER_LOG_FILE = 'log_name', MASTER_LOG_POS = log_pos
START SLAVE [SQL_THREAD] UNTIL
    RELAY_LOG_FILE = 'log_name', RELAY_LOG_POS = log_pos
```

这里 log_pos 是主服务器二进制日志或者中继日志中那张表的最后一个正确位置。在从服务器到达该位置后会停止，移除 replicate-wild-do-table 选项并重启服务器。

如何检查表是否一致

有多种方法来检查表在主从服务器上的一致性。这里做一个简要概述，根据遇到的问题使用其中一种方法。

校验表

顾名思义，此查询返回一个表校验和。这条 MySQL 语句不需要任何额外安装什么，始终可用。

```
mysql> CHECKSUM TABLE test;
+--------------+------------+
| Table        | Checksum   |
+--------------+------------+
| test.test    | 4220395591 |
+--------------+------------+
1 row in set (0.43 sec)
```

当你想检查主从服务器上的表数据是否一致时，在服务器两端分别执行该查询并比较结果。确保 Seconds_Behind_Master 是零，当 CHECKSUM TABLE 运行时保证主服务器上没有对这张表的写入操作。

Mysqldiff

这是随着 MySQL Workbench 捆绑安装的其中一个 MySQL WB 实用工具。此工具读取数据服务器对象的定义，然后使用类似 diff 的方法来判断两个对象是否一致。下面是一个用来对复制排错的例子：

```
$mysqldiff --server1=root@127.0.0.1:33511 --server2=root@127.0.0.1:33512 \
test.t1:test.t1
# server1 on 127.0.0.1: ... connected.
# server2 on 127.0.0.1: ... connected.
# Comparing test.t1 to test.t1                                    [PASS]
Success. All objects are the same.
```

pt-table-checksum

它属于 Percona 工具包集的一部分，是上述讨论中最强大的一个工具。它连接至主服务器和从服务器来比较表结构和表数据是否相同。为了做到这一点，该工具首先创建一张存放主服务器上表的检验和。这个值经复制后，从服务器上再执行 pt-table-checksum 来检验数据。

这里列举一个检查复制的例子：

```
$pt-table-checksum --replicate=test.checksum --create-replicate-table
h=127.0.0.1,P=33511,u=root --databases book
DATABASE TABLE CHUNK HOST      ENGINE   COUNT    CHECKSUM TIME WAIT STAT  LAG
book     t1        0 127.0.0.1 MyISAM       5    42981178    0 NULL NULL NULL
book     ts        0 127.0.0.1 MyISAM      65    aeb6b7a0    0 NULL NULL NULL
```

此条命令计算并保存 book 数据服务器中的每张表的校验和。一旦从服务器启动，我们就可以检查表数据是否相同：

```
$pt-table-checksum --replicate=test.checksum --replicate-check=2
h=127.0.0.1,P=33511,u=root --databases book
Differences on P=33512,h=127.0.0.1
DB    TBL   CHUNK CNT_DIFF CRC_DIFF BOUNDARIES
book  ts        0       -5        1 1=1
```

工具会输出的找到的不同，（如果有）。这里我们可以看到 ts 表在主从服务器上的数据是不一致的，而 t1 表中的数据是一样的。

无论你使用什么工具，保证在主服务器上检查后不要再复制改变。最简单的方法就是在你当前检验的表上加上一个写锁。

mysqldiff 和 pt-table-checksum 能做到的比我刚刚说的更多，但我所介绍的方法可以用来帮助排查复制故障，这是它最重要的一个功能。

如果你在主服务器二进制日志没有发现任何错误，中继日志也没有被损坏，这可能是网络问题或者磁盘损坏的现象。这两种情况下，可以把从服务器上中继日志的位置重新定位至 Exec_Master_Log_Pos，然后重启，按顺序执行下面的查询，STOP SLAVE; CHANGE MASTER master_log_pos=Exec_Master_Log_Pos_Value, master_log_file=Relay_Master_Log_File_Value'; START SLAVE，中继日志会重建。如果损坏是一个异常事件，复制会启动并再次运行。

但不要只清理，忽略了造成问题的可能性。请检查你的磁盘日志和网络问题。

为了查明是否是磁盘问题，检查操作系统的日志文件，利用工具来检查磁盘是否有坏块。如果发现了，修复磁盘。否则，你可能再次遇到类似问题。

在老版本的 MySQL 中网络问题会造成损坏。在 5.0.56 和 5.1.24 版本之前，不稳定的网络经常造成中继日志损坏。在 5.0.56 和 5.1.24 版本中，bug #26489 修复了，这种问题极少见了。从 5.6.2 版本开始，复制校验和引入了。这解决了遗留至今网络中断造成损坏的问题。

这些修正并没有自动恢复损坏的中继日志，但是防止了由于主服务器上问题或者网络问题导致的日志损坏。从 5.5 版本开始，可以配置 relay-log-recovery 选项，当从服务器重启时可自动恢复。

但即使你使用了打过补丁的 MySQL 新版本，你仍然要检查网络。越早定位网络问题，修复它们也越快。即便开启了自动恢复，解决网络问题也需要时间，同样也会拖慢复制。

本节考虑到了一些 SQL 线程错误，它是由于 I/O 线程引起的。下一节将讨论跟 I/O 线程无关的 SQL 线程问题。

5.3 与 SQL 线程有关的问题

2.7.1 节提到过每个从服务器只有单个 SQL 线程，所以它的所有错误都可用单线程的 MySQL 客户端进行测试。即使你运行多线程从服务器预览版，当重现错误时你也可以始终让它使用单线程。如果减少到只激活一个 SQL 线程未能使问题消失，请使用以下方法来修复单线程的逻辑错误，然后再切换至多线程。

重新创建一条导致复制失败的查询很简单：用 MySQL 命令行工具执行它就行了。

当在从服务器上收到 SQL 错误时，复制停止了。SHOW SLAVE STATUS 显示了导致问题的 SQL 线程错误：

```
Last_SQL_Errno: 1146
Last_SQL_Error: Error 'Table 'test.t1' doesn't exist' on query.
Default database: 'test'.
Query: 'INSERT INTO t1 VALUES(1)'
```

错误消息通常包含 SQL 查询的文本及失败的原因。此例中，错误消息很明显（我在从服务器上删除了 t1 表以创建这个例子），但在有疑问的情况下，你可以尝试在 MySQL 命令行下运行同样的命令来查看结果：

```
mysql> INSERT INTO t1 VALUES(1);
ERROR 1146 (42S02): Table 'test.t1' doesn't exist
```

本例中的错误清楚表明你接下来需要做什么来解决问题：创建表

```
mysql> CREATE TABLE t1(f1 INT);
Query OK, 0 rows affected (0.17 sec)
```

在表创建后，可以重启 SQL 线程：

```
mysql> STOP SLAVE SQL_THREAD;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+-------------------------------+
| Level | Code | Message                       |
+-------+------+-------------------------------+
| Note  | 1255 | Slave already has been stopped |
+-------+------+-------------------------------+
1 row in set (0.00 sec)

mysql> START SLAVE SQL_THREAD;
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  <skipped>
              Slave_IO_Running: Yes
             Slave_SQL_Running: Yes
                  <skipped>
                      Last_Errno: 0
                      Last_Error:
                  <skipped>
                   Last_IO_Errno: 0
                   Last_IO_Error:
                  Last_SQL_Errno: 0
                  Last_SQL_Error:
1 row in set (0.00 sec)
```

现在问题解决了，从服务器又成功运行。

5.3.1 当主从服务器上数据不同的时候

如果错误不能简单解决，检查主从服务器上表的定义是否一致。在出问题的查询执行前你也应该检查表中数据是否一样。

提示

MySQL 复制允许你在主从服务器上对表有不同的定义。如果你工作在这种配置下，分析相同的查询在不同表上是怎样运行的。同样也要检查是否使用了不同的存储引擎或者索引，这些都会影响最终结果。

当 SQL 线程停止时，一个很常见的原因就是从服务器的表不同于主服务器。这里不会说明造成错误的所有原因，普遍的有以下几个。

- 在例 2-1 或其他地方我们见到的问题，并发事务更新不能保证数据的一致性。

- INSERT ON DUPLICATE KEY UPDATE，如果随着其他连接上的更新运行在从服务器上，导致和主服务器执行顺序不一致，会更新错误的行，并且跳过主服务器上已经更新的行。
- 不考虑从服务器的存在，在 MyISAM 表上执行并发插入。
- 使用了不确定性函数[1]。

记住一个从服务器是否防崩溃也是非常重要的，因为当 mysqld 发生崩溃后，它能够在重新启动时重复执行崩溃前的事务，从而使主服务器与从服务器有不同的数据。在非事务表更新的中间当从服务器故障时也会发生类似问题，如 MyISAM。

5.3.2 从服务器上的循环复制以及无复制写入

如果你在同步之外对从服务器进行写入，你必须关注数据一致性。两种方法可以避免问题出现，确保写入对象有别于复制改变的数据，并且总是在主从服务器上使用具有不同序列的主键。可使用 AUTO_INCREMENT 主键来保持不同键值，在每个主服务器上设置不同的 auto_increment_offset 选项起点，保持复制环境中 auto_increment_increment 为服务器的个数。

环形复制中，主服务器即为另一台主服务器的从服务器[2]，这样一来也要避免相同的问题，复制数据在从服务器写入的情况下产生冲突。

MySQL 是允许配置环形复制的，但基于异步复制设计不能保证其数据一致性。一致性需要你自己来维护。因此，从排错角度来讲，调试环境复制的错误和其他复制问题一样。记住，双向配置就是服务器互相为主从服务器。一旦你遇到错误，首先确定它们中的主从角色，然后根据情况来应对。很可能你需要测试主从关系，并且互换角色。

为了举例说明技巧，让我们看一下第 5 章开篇（见图 5-4）“MySQL 多主配置”中简单的示例。

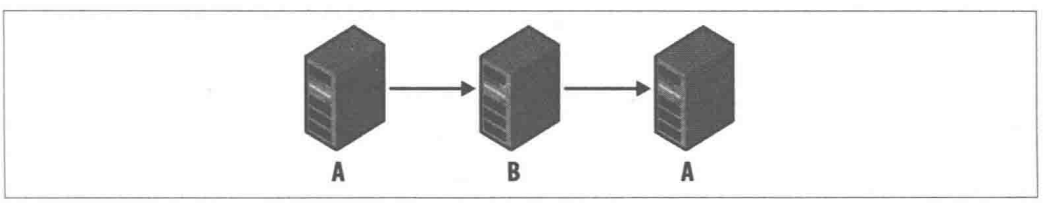


图 5-4 两台服务器互相复制

如果在这种配置中遇到问题，取出一对（像图 5-5 中那样），然后就像简单的主从一样

1：确定性函数指每次执行时使用同样的参数返回的结果是一样的。CONCAT('Hello, ', 'world!')是确定性的，而 NOW()不是。

2：这种复制也叫“多主复制”或“双向复制”。

来解决问题。排错时停止 B 上的所有更新。

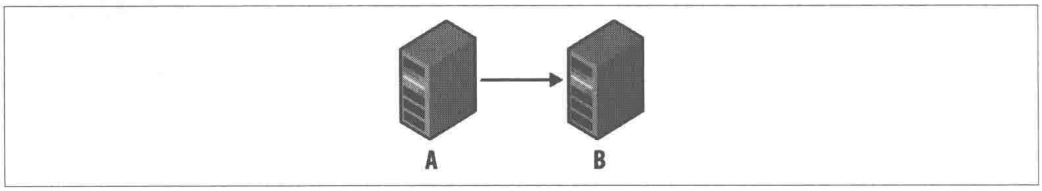


图 5-5　关注多主复制中的一个方向

在问题解决后，临时停止 A 上的更新并转向 B，就像图 5-6 中显示的配置一样。

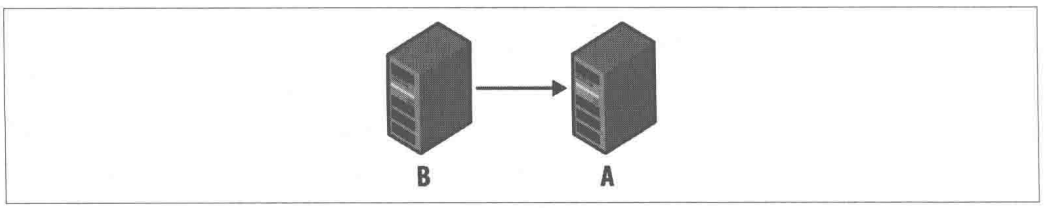


图 5-6　在多主复制中反转方向

如果在这种情况下还有问题，继续解决，然后开始 A 上的更新，现在你就回到了环形多主配置了。

此时，最好分析当时为什么会出现问题，并在两台服务器重启更新前进行修复。

这种方法适用于环中任意数目的服务器。

当搭建环形复制时，你需要清楚地分开查询，以便一台主服务器上的变更不会干扰到另外一台。错误可以使复制中断或导致数据不一致。我不在此说明最佳实践，你可以在由 Charles Bell 等人撰写的《高可用 MySQL》一书（O'Reilly 出版）的第 4 章中找到多主复制搭建的详细介绍。

- 好的设计对于创建一个无故障环形多主复制，是至关重要的。

5.3.3　不完整或被改变的 SQL 语句

如果错误消息没有显示完整的查询，并且错误日志也不包含完整的查询（当查询超过 1024 字节时会发生），你需要使用 mysqlbinlog 工具在主服务器二进制日志或者从服务器的中继日志中获取完整版的查询，然后分析为何执行失败。

这对基于语句的复制生效，因为查询的记录和发送是通过原生可读 SQL 形式进行的。但如果使用行格式你应该怎么做呢？行事件和查询一样是可以在 MySQL 客户端执行的。在 mysqlbinlog 中使用—verbose 选项来获取行事件的 SQL 表示形式。

- 在从服务器上始终使用和主服务器二进制日志事件应用时执行的相同查询。运用

mysqlbinlog 工具去检查运行哪条查询节省时间。二进制日志偶尔会包含一些查询，这些查询和主服务器上原始执行的查询有一些偏差，副作用就凸显出来了。如果你忽视这些影响，你可能花好几个小时去再现问题，但使用主服务器上执行的查询就无法再现。

5.3.4 主从服务器出现的不同错误

另一条让人混淆的状态消息是“Query caused different errors on master and slave...”，消息中通常会含有这种字眼。最让人疑惑的是，这个消息会说从服务器上没有错误——“Error on slave: ‘no error’ (0).”——但主服务器上有一个错误。这可以发生在，例如，主服务器上的错误由触发器产生，但是主表更新成功了。原始表上的查询被写入二进制日志文件中，包含了一个触发器更新失败的错误代码。这种情况下，如果从服务器上的触发器成功执行了或者从服务器上的表根本就没有触发器，那么从服务器上的查询不会返回错误，因此我们得到了这么一条消息。

为了迅速解决这个问题，使用“SET GLOBAL SLAVE_SKIP_COUNTER=1”忽略错误，并继续复制。不要忘了寻找造成问题的真正原因防止再次出现。

如果主服务器上的错误是由于触发器死锁造成的，手动修复从服务器因为主从表产生了不同的数据。要做到这一点，你需要找出哪些表包含不同数据并更新从服务器以保持和主服务器一致。

5.3.5 配置

另一个重点是检查主从服务器的配置选项。理想状况下两者应该是一致的，而有时也有好的理由使它们保持不同，如不同的硬件或负载。当配置不一致并且你开始收到的 SQL 错误不太容易解释时，检查那些能改变服务器行为的配置参数。我们在第 3 章讨论过一些。

我推荐使用--no-defaults 选项来运行 mysqld，就像单服务器配置一样，来检查服务器是否受你的定制选项影响，这里我推荐复制主服务器的配置到从服务器上，使得从服务器拥有和主服务器完全一致的配置。只调整服务器之间需要不同的那些选项，如 server_id，这个必须始终唯一。判断问题是否可以复现。如果不能，你放心一定是配置的一个变量导致了问题。此时你只须找出那个有问题的变量（用 3.6 节提到的技巧），然后做相应调整。

- 始终比较主从服务器上配置的不同。

5.3.6 当从服务器远远落后主服务器时

5.2 节讨论了网络不可靠导致 Seconds_Behind_Master 增大的情形。另一个造成巨大延迟的原因是从服务器执行速度慢于主服务器。

速度慢的硬件，更小的缓冲区，或者自身读和复制线程的资源争用都会导致从服务器更慢。另一个可能性就是主服务器是并行执行语句的，但是从服务器使用单线程一个一个地执行所有二进制日志中的事件。

首先你需要的做的就是找出从服务器落后的真正原因，然后思考如可改善。

对于速度慢的硬件最简单，通过购买性能好的硬件就行。但在花钱之前，分析一下主服务器是否有效利用了硬件资源，及从服务器上与性能相关的选项是否可优化。例如，如果主服务器运行在一个共享环境中，而从服务器是在一台稍慢的专有服务器上，你是有机会去提升它的速度的。计算主服务器实际使用了多少资源，通过调整配置参数从服务器性能就有多少提升。

如果两台服务器硬件一样，或者从服务器配置更好但依然落后，请检查与性能相关的选项。分析其影响并根据情况调整从服务器。将主从服务器上的选项设为相同是个好的起点。这样，可以肯定这些选项对于复制事件是优化的，你只需要调整从服务器上那些提高并发负载的选项就行了。

最坏的情况是，缓慢是由于主服务器并行执行但从服务器是单线程产生的。这种情况下你所能做的，除了使用多线程从服务器预览之外，就是升级从服务器的硬件，尽可能地调整性能相关的选项。

上述例子中，你仍然应该分析在从服务器 SQL 线程执行的同时其查询的影响。第 2 章提到过关于并发的错误排查。

第6章

问题排查技术与工具

前面的章节已经介绍了许多问题排查技术与工具。对其中一部分工具，我进行了详细的剖析，而对于另一部分工具，我仅仅涉及了它们的使用。本章补充了前面章节尚未涉及的详细内容，我会尽量避免内容上的重复。很多技术与工具互相依赖，所以在本章中将把它们结合起来使用。

有太多的工具需要详细介绍，但是篇幅有限，所以这里只介绍那些必需的工具。这些工具中的大部分都是命令行工具，要么来自于 MySQL 发布版，要么就是作为独立的包提供的。我还介绍一些第三方工具及其命令行使用方法。我这么做，并不是因为我不喜欢第三方工具，而是想让大家知道 MySQL 提供了很多非常强大的工具，并且你可以一直使用这些工具。MySQL 工具的优势之一在于，它一直都是可用的，这对于客户来说是非常重要的。一些公司会有规定，禁止他们的员工下载第三方工具。因此，当我们进行技术支持时，我们总是偏爱使用 MySQL 发布版自带的工具。

出于类似原因，这里不会介绍图形工具。命令行工具不需要特别的环境，例如，X Window System 或特定的操作系统，而图形工具却会有更多的要求。

最后，我确信 MySQL 的工具比其他包里类似的工具要更加优秀。假如你发现了更强大的第三方工具，也可以使用它们。但是，知道使用简单的工具能做什么总是非常有用的。

6.1 查询

第 1 章已经介绍了一条查询如何影响整个服务器的性能，以及如何找到哪一条语句出现了问题。这里将会针对这种类型的问题补充几句。

由应用程序产生的有问题的查询，可以通过检查错误日志来定位。这可以通过使用应

用程序中的输出函数来实现，这些输出函数自动记录了发送给 MySQL 服务器的查询，并使用了在应用程序和通用查询日志文件中针对这个目的所编写的一个库。这里介绍如何调整这些日志。

通用日志文件缺少了一些帮助大家调试查询的重要信息，例如，查询执行时间、错误与警告信息、返回集信息。查询执行时间信息可以用来查找慢查询。当然，你可以使用一个自定义库或者通过在应用程序的任何位置简单地添加输出函数来记录所有这些特殊信息。但是在开始调试应用程序之前，你最好使用内置资源：慢查询日志。

6.1.1 慢查询日志

慢查询日志记录运行时间超过 long_query_time 秒的查询，该变量的默认值为 10，但是可以减小它。实际上，可以将这个值设置为 0 来记录所有查询。从 5.1 版本以来，像通用查询日志那样，可以根据需要动态开启或关闭慢日志。也可以把输出从定向到表中，这样就能像其他数据那样查询表。

为了做性能优化，我们需要找到最慢的查询，并单独地一条一条检查，然后改写这些查询语句或者做一些其他必要的改变，例如索引。为了发现更多的查询，可以从 long_query_time 默认值开始，一点一点减少，直至 0。这种方法首先揭示了最慢的查询。

默认情况下，这个选项不会记录管理语句和不用索引的快查询，但是可以分别设置 log-slow-admin-statements 和 log_queries_not_using_indexes 选项来记录这类查询。

慢查询日志的缺点之一是不能忽略你认为不需要优化的查询。把日志写到表中能帮助你过滤出自己不想看到的查询，因为你可以使用 WHERE 子句、分组和排序来把重点放到你认为重要的查询上。

mysqldumpslow

mysqldumpslow 工具可以以一种汇总格式输出慢查询日志中的内容。它将查询进行分组，所以如果两条查询字面上相同，但使用不同的参数，它们与执行次数只输出一次。这意味该工具把 SELECT * FROM t2 WHERE f1=1 与 SELECT * FROMt 2 WHERE f1=2 同样对待，因为 f1 参数的实际值通常不会影响查询的执行时间。如果你想在应用程序中找到同一类型却执行成千上万次的查询，该程序特别方便。

```
$mysqldumpslow /Users/apple/Applications/mysql-5.1/data512/mysqld512-apple-slow.log

Reading mysql slow query log from
/Users/apple/Applications/mysql-5.1/data512/mysqld512-apple-slow.log
Count: 3  Time=0.03s (0s)  Lock=0.03s (0s)  Rows=0.7 (2), root[root]@localhost
  SELECT * FROM t2 WHERE f1=N
```

```
Count: 1  Time=0.03s (0s)  Lock=0.00s (0s)  Rows=1.0 (1), root[root]@localhost
  select @@version_comment limit N

Count: 1  Time=0.02s (0s)  Lock=0.03s (0s)  Rows=3.0 (3), root[root]@localhost
  SELECT * FROM t2

Count: 3  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=0.3 (1), root[root]@localhost
  select TEXT from test where ID=N

Count: 1  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=3.0 (3), root[root]@localhost
  select * from t2
```

需要注意的是，尽管该工具足够智能，可以使用不同的参数来对相似的语句进行分组，但是即使语法上具有微小的差别，它也会将这些语句视作不同的语句——例如，使用不同的大小写字母或不同的空格。

6.1.2 可定制的工具

一般情况，仅仅找到慢查询还不够。你需要知道更多，例如，返回哪些警告或错误，更新或者查询的行数。可以使用 3 种方法来获得这些信息：通过应用程序或编写插件或通过代理。

应用程序可以使用 1.4 节介绍的方法来接收和记录信息；这里不再提供详细说明或者示例，因为它们主要取决于编程语言和其他上下文。如果查询执行时间，那么只需要在应用程序中调用 mysql_query 或 mysql_real_query 方法前后测量它。这种方法的优点是可调式。其缺点是需要修改应用程序，这对于使用第三方软件的用户来说是不可能的。

如果出于审计的目的，你想编写一个 MySQL 服务器插件，那么可以参考 MySQL 参考手册中的“编写审计插件”（http://dev.mysql.com/doc/refman/5.5/en/writing-audit-plugins.html）一节。一旦安装，MySQL 插件将变成 MySQL 服务器的一部分，并且能通过 SQL 查询访问。除了这个优点外，这种解决方案完全独立于应用程序，不需要更改已有代码，并能被多个应用程序使用。缺点是，它必须针对特定的 MySQL 服务器版本进行编译和安装，如果要把插件分发给更多用户，这就会比较麻烦。

第三种解决方案是使用可脚本化的代理。代理是一个守护进程，它位于服务器与客户端之间，可以独立于服务器和客户端进行配置。因为它能获得所有流量，所以可以使用它做你想要做的任何事。这种方法的优点是你完全独立于服务端和客户端，因此无须改变你从别人那里接手的任何事情。其缺点是代理增加了额外的一层处理，所以它会降低应用程序的性能，并在客户端和服务器之间创建一个新的单点故障。

MySQL 代理

MySQL 代理是一个可脚本化的守护进程，它支持 MySQL 协议并位于 MySQL 服务器与应用

程序之间。应用程序需要配置成所有查询经过代理的模式。这通常只需要设定适当的主机名和端口。

MySQL 代理支持 Lua 脚本语言。它允许查询重写和结果集重写、日志记录、负载平衡以及其他内容。这里只讨论日志，因为它能帮助我们调试慢查询。

出于审计目的，你需要编写一个 Lua 脚本来保存必要的信息。以下给出了一个简单的脚本示例，它模拟通用查询日志行为，而且还保存查询执行时间。

```
function read_query( packet )
        if packet:byte() == proxy.COM_QUERY then
                print(os.date("%d%m%y %H:%M:%S") .. "\t"
                .. proxy.connection.server.thread_id
                .."\tQuery\t" .. packet:sub(2))

                proxy.queries:append(1, packet )
                return proxy.PROXY_SEND_QUERY
        end
end

function read_query_result(inj)
        print("Query execution time: " .. (inj.query_time / 1000) .. "ms,\t"
        .. "Response time: " .. (inj.response_time / 1000) .. "ms,\t"
        .. "Total time: " .. ((inj.query_time + inj.response_time) / 1000) .. "ms")
end
```

如下所示调用脚本。

```
$mysql-proxy --admin-username=admin --admin-password=foo \
--admin-lua-script=./lib/mysql-proxy/lua/admin.lua \
--proxy-address=127.0.0.1:4040 --proxy-backend-addresses=127.0.0.1:3355 \
--proxy-lua-script=`pwd`/general_log.lua
```

结果将如下所示。

```
$mysql-proxy --admin-username=admin --admin-password=foo \
--admin-lua-script=./lib/mysql-proxy/lua/admin.lua \
--proxy-address=127.0.0.1:4040 --proxy-backend-addresses=127.0.0.1:3355 \
--proxy-lua-script=`pwd`/general_log.lua
031111 01:51:11    20    Query    show tables
Query execution time: 376.57ms, Response time: 376.612ms, Total time: 753.182ms
031111 01:51:19    20    Query    select * from t1
Query execution time: 246.849ms,    Response time: 246.875ms, Total time: 493.724ms
031111 01:51:27    20    Query    select * from t3
Query execution time: 689.772ms,    Response time: 689.801ms, Total time: 1379.573ms
031111 01:51:39    20    Query    select count(*) from t4
Query execution time: 280.751ms,    Response time: 280.777ms, Total time: 561.528ms
```

可以改写这个脚本来满足自己的各种需求。在执行查询之前和之后，MySQL 代理有权访问查询与结果集，与一般的通用日志文件相比，它允许用户记录更多信息：错误、警告、受影响的行数、查询执行时间，甚至完整返回集。

在 MySQL Forge 中，可以发现很多有用的脚本。

6.1.3 MySQL 命令行接口

MySQL 命令行客户端，也称为 MySQL CLI，是在大多数测试情况下使用的首要工具。

当查询无法正常执行时，首先怀疑应用程序中有 bug。但每个查询可能被众多问题影响，尤其是客户端和服务器选项。因此，如果你觉得自己发送了正确的查询却得到了错误的结果，就可以在 MySQL CLI 中测试该查询。这是验证你的猜测最简单且最快速的方法。

当 MySQL Support Bugs Verification Group（MySQL 支持 bug 校验组）的成员怀疑客户端错误或（常见的）配置错误时，我们总是查找那些能够在 MySQL CLI 中测试的查询语句。这里将简单解释一下为什么这一点很重要，以及为什么其他工具不适合用于第一轮故障排除。

提示

除了影响每个 MySQL 客户端应用程序的常用客户端参数外，Connector/J 与 Connector/ODBC 都有它们各自的接口和参数。它们可以通过这些接口进行规则转换来影响查询语句。因此，如果你使用它们中的任意一个，那么在 MySQL CLI 中测试这些有问题的语句就会变得至关重要。

为了测试应用程序返回的结果是否与你期望的一致，使用--column-type-info 选项启动 MySQL CLI，它会输出与数据类型相关的信息：

```
$mysql --column-type-info 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 415
Server version: 5.1.60-debug Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

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

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

mysql> SELECT * FROM t1;
Field   1:  `f1`
Catalog:    `def`
Database:   `test`
Table:      `t1`
Org_table:  `t1`
Type:       LONG
```

```
Collation:  binary (63)
Length:     11
Max_length: 2
Decimals:   0
Flags:      NOT_NULL PRI_KEY AUTO_INCREMENT NUM PART_KEY

Field   2:  `f2`
Catalog:    `def`
Database:   `test`
Table:      `t1`
Org_table:  `t1`
Type:       BLOB
Collation:  latin1_swedish_ci (8)
Length:     65535
Max_length: 32
Decimals:   0
Flags:      BLOB


+----+----------------------------------+
| f1 | f2                               |
+----+----------------------------------+
|  1 | f9f760a2dc91dfaf1cbc95046b249a3b |
|  2 | e81077a403cc27525fdbb587451e7935 |
|  3 | a003a1256c0178e0c4d37a063ad1786b |
|  4 | 2447565c49917f2daeaac192614eabe8 |
|  6 | 6bfb21c57cc3a8de22dc4dbf635fdc77 |
|  7 | 2d9f5350ba5b914a8f4abf31b5ae975c |
|  8 | 57e87a3c55d053b5ab428f5da7f6ba28 |
|  9 | ad2ede5e02ce1da95dcd8f71426d5e7b |
| 13 | 65400ab09cdc725ec5bafe2ac4f5045d |
| 14 | 48f1b1e99041365a74444f75a6689d64 |
| 15 | 1f6c558fe2c492f1da2ebfb42d9f53dc |
| 16 | ff931f7ac8c8035a929dc35fee444332 |
| 17 | f26f6b6e8d16ae5603cf8c02409f4bb5 |
| 18 | 239ca93bf7b5fd82a53e731004cba761 |
| 19 | 1f985d1fe9efa14453a964e2c4657ab5 |
| 20 | 1d599460b91f2d892c024fe5a64f7d6d |
+----+----------------------------------+
16 rows in set (0.09 sec)
```

这个输出显示了与字段有关的基本元信息，如数据类型与排序规则等，可以通过配置选项或应用程序设置来改变这些信息。通过在 MySQL CLI 中运行它，你可以看到查询通常会在服务器上做些什么，并且如果服务器接收的信息与应用程序发出的不同，你就会推测这里出现了问题。现在让我们对输出信息逐行进行分析。

```
mysql> SELECT `name` FROM `t1`;
Field   1:  `name`
```

前面的输出显示了字段名称。

```
Catalog:    `def`
```

目录名称一直都是 def。

```
Database:   `test`
```

当前使用的数据库。

```
Table:      `t1`
```

表名，但当使用 select *field_name* from *table_name* as *alias_name* 语法时，这里显示的是表的别名。

```
Org_table:  `t1`
```

原始表名，当前面行显示的是别名时，知道这个就非常有用。

```
Type:       VAR_STRING
```

前面的行显示字段类型。

```
Collation:  latin1_swedish_ci (8)
```

排序规则。

```
Length:     255
```

表定义中定义的字段长度。

```
Max_length: 5
```

返回结果集中字段长度的最大值。

```
Decimals:   0
```

如果它是一个整数类型，则表示该字段中小数点后的位数。

```
Flags:
```

如果有，这表示标记。例如，主键字段会有 PRI_KEY 与 AUTO_INCREMENT 标记。

```
+--------+
| name   |
+--------+
| sveta  |
+--------+
1 row in set (0.00 sec)
```

查询的结果集。

如果你最喜欢的 MySQL 客户端并不是 MySQL CLI，哪该怎么办？可以在这里进行测试，但是要记住，它可能会带来一些负面影响。特别是 GUI 面客户端。例如，如果客户端使用 JDBC，那么它将受它的配置影响，但它的配置不会影响 MySQL CLI。另一些客户端具有预设置字符集，如 MySQL Workbench，它只支持 UTF-8。这样的设置将阻止你测试其他字符集。一些客户端（如 Workbench）会在每次查询之后断开连接并重新建立连接。而另一些客户端就会受到小的线程缓冲区影响，这对于基于 Web 的客户端来说非常常见。有时你可能会重新配置客户端，但是一旦遇到问题时，就更加容易切换到命令行并尝试在 MySQL CLI 中执行查询。

MySQL CLI 的优势之一就是，与选项相比，它非常透明：你总能查看它的配置并对其进行调优。不可否认，MySQL CLI 像任何软件一样也有 bug，但通常有数以百万计的用户都在大规模使用该工具，并且 Oracle 公司内部也在积极使用它，因此你被 MySQL CLI 的 bug 影响到的概率是非常低的。

提示

除了 Connector/PHP 之外，所有未配置的连接器最初都把 character_set_clients 设置为 UTF8，character_set_results 设置为 NULL。这实际上是一个“调试”模式的字符集，因此不建议在命令行客户端中使用。

这种行为背后的原因是，让驱动程序逻辑确定显示结果到客户端和从客户端存储结果的最佳方式，并且通过防止服务器将文本结果转换成字符集结果来避免非常常见的“双重转换”bug。然而，对于即席查询这一招行不通，例如，SHOW CREATE TABLE，这个语句会把二进制看成 UTF8，或 SELECT varbinary_col FROM some_table，这里本身就是二进制类型，或 SELECT CONCAT（char_field1，1）AS a，这里 a 将会有二进制标记集。

因此，所有连接器都会在它们的连接选项中用某种连接方法来告诉驱动程序用 UTF8 而不是二进制字符来处理函数返回值。同时，尽管每个连接器有自己的默认编码，但它们还是会执行 SET NAMES UTF8。这主要是为了避免 libmysqlclient 库的默认行为，即将所有与字符集相关的变量设置为 latin1。

- 如果你认为一个查询应该执行正常，却返回了非期望的结果，那么在你认为它可能是 MySQL 服务器代码中的一个 bug 之前，请尝试在 MySQL CLI 中执行该语句。

提示

我喜欢自动化。当我为 bug 记录创建测试时，我使用了一个脚本，该脚本在一组发布版 MySQL 服务器中（参见 6.9.3 节）运行 MySQL Test Framework 测试。这能帮助我用一条命令在很多版本中测试一个问题。但是这个习惯曾经让我出了一个冷笑话。我测试一个被提交的 bug，但是我不能重现它，我花了很长时间与报告人沟通，我试了很多选项，遗憾的是，我完全依赖于我们的测试套件，并没有意识到该客户端可能会带来副作用。然后我的同事在 MySQL CLI 中尝试该测试用例，并得到与原报告人彻底相同的结果。该 bug 得到确认并被修复。

这个经验告诉我们，忽视客户端可能的差异是多么危险，同时在做其他事情之前尝试使用 MySQL CLI 是多么重要。

6.2 环境的影响

本书已经介绍过环境的一些影响，例如，并发线程、操作系统、硬件、并发运行软件以及 MySQL 服务器与客户端的选项。但是，一个查询，就算是单个客户运行在专用的 MySQL 服务器上，同样也可能被它所运行的环境所影响。

你从存储过程、存储函数、触发器或事件中调用查询时，在这些场景中，就有可能会用它们自己默认的参数值覆盖当前会话选项。因此，如果你遇到你无法解释的问题，请尝试在该例程之外的环境中执行相同的查询。如果结果不相同，请核对程例的字符集与 SQL 模式。请检查例程中所有的必要对象是否存在，影响查询的变量是否设置。另外一个主要的环境变量是 time_zone，它会影响如 NOW()与 CURDATE()等时间函数的结果。

- 如果你的查询无法正常工作，那么应检查它所运行的环境。

6.3 沙箱

沙箱（sandbox）是一个为运行应用程序而搭建的隔离环境，它不受环境外的其他任何东西影响。在本书中，我一直鼓励大家在数据库上“尝试”各种配置选项与调整。但是这样的一些“尝试”可能会使应用程序变慢，甚至使得应用程序或数据库崩溃。这不是大部分用户想要的。相反，你可以使用沙箱来隔离自己环境中正在测试的系统，这样，你做错什么都不要紧。

在 MySQL 领域，Giuseppe Maxia 通过创建一个名为 MySQL Sandbox 的工具引入了沙箱这个概念。稍后将介绍 MySQL Sandbox 以及将来它如何能帮助我们，但这里先简单地介绍沙箱的一些变体。

在一个表上安全地测试查询的最简单方法就是创建一个副本，这样，原始表就是最安全的，当你对复制数据进行测试时，原始表还可以像往常那样被应用程序使用。你也不用担心如何恢复你无意中改变的数据。

```
CREATE TABLE test_problem LIKE problem;
INSERT INTO test_problem SELECT * FROM problem;
```

这种解决方案的好处之一是，可以通过 WHERE 来限制行数，从而做到只复制其中的一部分数据。例如，假设你正在测试一个复杂的查询，并确定它正确地执行了某个 WHERE 子句。当创建表时，可以限制测试表，以满足该测试的条件，然后你就有一个比较小的表可以用来测试查询。

```
INSERT INTO test_problem SELECT FROM problem WHERE condition]
```

也可以通过去掉一些条件来简化查询。当原始表很大时，这将节约很多时间。当 WHERE 子句运行正常，而 GROUP BY 分组与 ORDER BY 排序运行错误时，这项技术也非常有用。

如果一个查询访问一个以上的表或者只是想在不同表上测试查询，那么创建一个独立的数据库是有意义的。

```
CREATE DATABASE sandbox;
USE sandbox;
CREATE TABLE problem LIKE production.problem;
INSERT INTO problem SELECT * FROM production.problem [WHERE ...]
```

在这种情况，你将有一个与生产环境完全一样的数据库副本。即使你损坏了副本里的某些行，你也不会有任何损失。

对于查询重写以及类似问题，这两种方法都是非常好的。但是如果服务器崩溃或者占用大量资源，则最好不要测试该服务器上的任何东西。而应该出于测试目的专门搭建一个开发服务器并从生产服务器上复制数据。当你正计划升级 MySQL 或者想确定更新版本的 MySQL 是否已经修复了某个特定的 bug 时，这么做同样有帮助。

在应用程序创建之初，你可以在你的开发机器上升级 MySQL 服务器。但如果应用程序运行很长时间并且你需要测试某一特定 MySQL 版本如何影响实际数据，在沙箱中，像这种升级就很难手动创建。在这种情况下，MySQL Sandbox 就是最佳选择。

首先要创建安装，需要一个期望版本且没有安装程序（例如，在 Linux 中，以 tar.gz 结尾）的 MySQL 包，以及 MySQL Sandbox 的一个副本，MySQL Sandbox 可以从 https://launchpad.net/mysql-sandbox 下载。通过如下命令从 MySQL 包中创建沙箱：

```
$make_sandbox mysql-5.4.2-beta-linux-x86_64-glibc23.tar.gz
unpacking /mysql-5.4.2-beta-linux-x86_64-glibc23.tar.gz
...
 The MySQL Sandbox,  version 3.0.05
 (C) 2006,2007,2008,2009 Giuseppe Maxia
installing with the following parameters:
upper_directory  = /users/ssmirnova/sandboxes
...
........ sandbox server started
Your sandbox server was installed in
$HOME/sandboxes/msb_5_4_2
```

一旦安装，你就应该停止服务器，然后更改配置文件，以便能与生产环境的配置做出对比，然后重启它，导入生产数据库的备份。现在你已经可以安全地测试了。当你想在多个 MySQL 版本中快速测试你的应用程序时，例如，为了确定 bug 是否修复，这种方法非常有用。

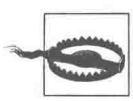
可以创建任意多的沙箱，并且可以在没有额外开销的情况下测试 MySQL 和数据库的不同方面。你甚至可以创建一个复制的沙箱，即一个包含一个主服务器和多个确定的从服务器的沙箱。

```
$make_replication_sandbox mysql-5.1.51-osx10.4-i686.tar.gz

installing and starting master
installing slave 1
installing slave 2
starting slave 1
... sandbox server started
starting slave 2
....... sandbox server started
initializing slave 1
initializing slave 2
replication directory installed in $HOME/sandboxes/rsandbox_5_1_51

$cd $HOME/sandboxes/rsandbox_5_1_51

$./m
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.1.51-log MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

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

master [localhost] {msandbox} ((none)) > \q
Bye

$./s1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.1.51-log MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

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

slave1 [localhost] {msandbox} ((none)) > SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: rsandbox
                  Master_Port: 26366
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000001
          Read_Master_Log_Pos: 1690
               Relay_Log_File: mysql_sandbox26367-relay-bin.000002
                Relay_Log_Pos: 1835
        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: 1690
             Relay_Log_Space: 2003
             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:
1 row in set (0.00 sec)

slave1 [localhost] {msandbox} ((none)) > \q
Bye

$./stop_all
executing "stop" on slave 1
executing "stop" on slave 2
executing "stop" on master
```

一旦沙箱运行正常，就可以根据其选项进行各种测试。

对单个服务器使用 MySQL Sandbox 不可忽视的优点之一就体现在你需要对比多个环境时。如果你正在某类系统上使用某个版本的软件，那么你可以从 MySQL 的备份中只将生产数据加载到开发计算机上。使用复制，这是不行的，因为你将至少需要两个 MySQL 实例。并且，复制的沙箱能极大地节约时间，即使你不关注版本与自定义环境，因为根据你的需要，它仅仅只须花费几分钟的时间来安装和设置尽可能多的 MySQL 实例。



警告

Workbench 实用工具（Workbench Utilities）集中的一些工具可以帮助用户为自己的生产数据库创建一个沙箱副本。

mysqldbcopy

复制一个数据库，不管是在同一个服务器上创建一个不同名的数据库，还是在不同服务器上创建一个同名的或者不同名的数据库。

mysqlreplicate

在两个服务器间，配置与启动复制。

mysqlserverclone

在一个正运行的服务器上启动一个新实例。

6.4 错误与日志

另一个重要的故障排除技术听起来特别简单：从服务器读取和分析信息。这是非常重要的一步。第 1 章介绍了一些工具以及一些示例，它们可以帮助你获取和分析信息，这里增加之前跳过的一些细节。

6.4.1 再论错误信息

错误消息是关键，不应该被忽视。可以在 MySQL 参考手册（http://dev.mysql.com/doc/refman/5.5/en/errorhandling.html 页面）找到错误的相关信息。此网页列出了客户端与服务端的错误消息，却省略了特定予存储引擎的消息。它也无法解释来自操作系统的错误。通过实用程序 perror（见 1.4 节），可以得出描述操作系统错误信息的字符串。

另一个非常重要的工具是 mysqld 错误日志文件，其中包含关于表损坏、服务器崩溃、复制错误，以及更多的信息。它始终都是开启着，当你遇到问题时，可以对其进行分析。应用程序中的日志不能总是取代 mysqld 服务器的错误日志，因为后者能包含对应用程序不可见的一些问题与细节。

6.4.2 崩溃

1.7 节介绍了崩溃与处理这些崩溃的合适顺序。首先使用上一节介绍的方法：查看错误日志文件，并分析其内容。这在大部分情况是可行的，但本节将介绍如果错误日志不包含足够的信息来帮助你解决崩溃，你该怎么办。

最新版本的 MySQL 服务器，甚至是发布版，都会输出回溯信息（backtrace）。因此，如果错误日志文件没有输出回溯信息，则应该检查 mysqld 二进制文件是否被删除了[1]（在类 UNIX 系统上，file 命令能告知可执行文件是否被删除了）。如果它确实被删除了，就可以使用 MySQL 发布版的 mysqld 二进制文件替换原来的。如果是你自己编译 MySQL，请编译一个带有符号表的版本供测试使用。

- 确认 mysqld 二进制文件是否包含符号表，例如：没有被删除。

1：我已经目睹客户人为对 mysqld 二进制文件进行删除，以达到更好的性能，所以我认为在本书中包含这个非常重要。

在某些情况下，可能需要运行调试的二进制文件。这是一个名为 mysqld-debug 的文件，它位于 MySQL 安装根目录的 bin 目录下。

带有调试功能的二进制文件包含断言，它有助于在早期阶段捕捉问题。在这种情况下，你可能会到一条比较好的错误消息，因为当服务器出现问题时，错误消息将被捕捉到，而不用拖到内存泄漏发生时。使用发布的二进制版本，你是得不到错误消息的，除非内存泄漏真正导致了崩溃。

使用调试功能的二进制文件的代价就是性能下降。

如果错误日志文件没有关于崩溃的足够信息来帮助你找到问题的根据，那么可以尝试使用以下介绍的两种方法。就像 1.7 节介绍的从错误日志文件获得回溯信息那样，在任何情况下，工作总是从证据开始。不要只是直观猜测，因为如果你试图用错误的猜测来解决问题，你可能会面临更多的问题。

- 始终进行测试。任何猜测都可能是错误的。

1. 核心文件

核心文件包含一个进程的内存映像，当进程意外终止时会创建它（如果操作系统配置成这样）。可以通过在 MySQL 服务器启动时设置 core 选项来获得一个核心文件，但是首先你必须确定操作系统允许创建核心文件。

为了通过核心文件调试，你需要熟悉 MySQL 源代码。通过 MySQL Forge 上的“MySQL Internals”页面熟悉源代码是一个良好的开始。我同样推荐 Charles A. Bell 博士撰写的《Expert MySQL》（Apress 出版社）一书。同样可以在 Sasha Pachev 撰写的《Understanding MySQL Internals》（O'Reilly 出版社）中以及由 Andrew Hutchings 与 Sergei Golubchik 联手编写的《MySQL 5.1 Plugin Development》（Packt 出版社）中获得非常有用的信息。当然，从某种程度上来说，你有必要深入了解 MySQL 源代码本身。

这里不会详细介绍如何处理核心文件，因为 MySQL 源代码博大精深，即使专门用一本书的篇幅来讨论这个主题也不一定能面面俱到，所以这里只展示一个小示例。

为了创建核心文件，必须使用 core 选项来启动 mysqld，并调整操作系统，以便能够创建大小不受限制的核心文件。不同的操作系统使用不同的工具来控制核心文件的创建。例如，Solaris 使用 coreadm，而在我的 Mac OS X Tiger 系统上，我不得不编辑/etc/hostconfig。在 Window 中，你应该有 mysqld 与操作系统的调试符号表。在类 UNIX 系统中，最简单的方式是使用 ulimit -c 命令，这里它应该被设置为 unlimited，不过，请参考你的 OS 用户手册来确定是否还有其他的配置需要更改。

核心文件创建后，可以通过调试器来读取其内容。这里使用 gdb，但这不是必需的，你可以使用自己喜欢的调试器。

```
$gdb ../libexec/mysqld var/log/main.bugXXXXX/mysqld.1/data/core.21965
```

该命令行包括 gdb 命令的名称，紧随其后是可执行文件 mysqld 的路径与核心文件本身的路径。

```
GNU gdb (GDB) 7.3.1
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html7gt;
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/7gt;...
Reading symbols from /users/ssmirnova/build/mysql-5.1/libexec/mysqld...done.
[New LWP 21984]
[New LWP 21970]
[New LWP 21972]
[New LWP 21974]
[New LWP 21965]
[New LWP 21973]
[New LWP 21967]
[New LWP 21971]
[New LWP 21968]
[New LWP 21969]

warning: Can't read pathname for load map: Input/output error.
[Thread debugging using libthread_db enabled]
Core was generated by `/users/ssmirnova/build/mysql-5.1/libexec/mysqld
--defaults-group-suffix=.1 --de'.
Program terminated with signal 11, Segmentation fault.

#0  0x00832416 in __kernel_vsyscall ()
(gdb)
```

首先我们需要回溯信息：

```
(gdb) bt
#0  0x00832416 in __kernel_vsyscall ()
#1  0x008ce023 in pthread_kill () from /lib/libpthread.so.0
#2  0x085aa6ad in my_write_core (sig=11) at stacktrace.c:310
#3  0x0824f412 in handle_segfault (sig=11) at mysqld.cc:2537
#4  7lt;signal handler called>
#5  0x084bce68 in mach_read_from_2 (b=0xffffffe 7lt;Address 0xffffffe out of
bounds>) at ../../storage/innobase/include/mach0data.ic:68
#6  0x084cfdd6 in rec_get_next_offs (rec=0x0, comp=1) at
../../storage/innobase/include/rem0rec.ic:278
#7  0x084e32c9 in row_search_for_mysql (buf=0xb281d7b0 "\371\001", mode=2,
prebuilt=0xb732de68, match_mode=1, direction=0) at row/row0sel.c:3727
#8  0x08476177 in ha_innobase::index_read (this=0xb281d660, buf=0xb281d7b0
"\371\001", key_ptr=0xb2822198 "", key_len=0, find_flag=HA_READ_KEY_EXACT) at
handler/ha_innodb.cc:4443
#9  0x0838f13c in handler::index_read_map (this=0xb281d660, buf=0xb281d7b0
"\371\001", key=0xb2822198 "", keypart_map=0, find_flag=HA_READ_KEY_EXACT) at
handler.h:1390
#10 0x082dd38f in join_read_always_key (tab=0xb28219e8) at sql_select.cc:11691
```

```
#11 0x082da39f in sub_select (join=0xb2822468, join_tab=0xb28219e8,
end_of_records=false) at sql_select.cc:11141
#12 0x082da79f in do_select (join=0xb2822468, fields=0xb2834954, table=0x0,
procedure=0x0) at sql_select.cc:10898
#13 0x082f1bef in JOIN::exec (this=0xb2822468) at sql_select.cc:2199
#14 0x082090db in subselect_single_select_engine::exec (this=0xb28358a0) at
item_subselect.cc:1958
<skipped>
```

通过以上输出，你已经了解了一些信息。如果你想要详细了解如何使用核心文件，可以转向 man core、调试器文档、MySQL Internals Manual、我推荐的图书以及代源码。

2. 通用日志文件

另外一种捕捉发生了什么的方法就是使用 6.1.2 节提到的：通用日志文件与代理解决方案。这里概念都非常相似，我将展示如何通过通用查询日志来获取错误，并且如果你决定使用代理解决方案，让你自己推断出代理解决方案。我将再次使用 1.7 节介绍的例子，但是在这个案例中，它运行在我自己的 MacBook 上。错误日志包含如下内容。

```
091002 16:49:48 - mysqld got signal 10 ;
This could be because you hit a bug. It is also possible that this binary
or one of the libraries it was linked against is corrupt, improperly built,
or misconfigured. This error can also be caused by malfunctioning hardware.
We will try our best to scrape up some info that will hopefully help diagnose
the problem, but since we have already crashed, something is definitely wrong
and this may fail.
key_buffer_size=8384512
read_buffer_size=131072
max_used_connections=1
max_connections=100
threads_connected=1
It is possible that mysqld could use up to
key_buffer_size + (read_buffer_size + sort_buffer_size)*max_connections = 225784
K
```

此版本不输出回溯信息。如果我处于不能使用调试版本的 MySQL 服务器的情况，我怎么能知道这是怎么回事？

这是通用日志能再次帮助我们的地方。MySQL 在执行每个查询之前会把它写入这个日志。因此，我们能从这个日志中找到关于崩溃的信息。首先，开启日志。

```
mysql> SET GLOBAL general_log=1;
Query OK, 0 rows affected (0.00 sec)
mysql> SET GLOBAL log_output='table';
Query OK, 0 rows affected (0.00 sec)
```

等待，值得崩溃再次发生，然后，查看通用日志的内容。

```
mysql> SELECT argument FROM mysql.general_log ORDER BY event_time
desc \G
*************************** 1. row ***************************
argument: Access denied for user 'MySQL_Instance_Manager'@'localhost'
(using password: YES)
```

```
*************************** 2. row ***************************
argument: select 1 from `t1` where `c0` <> (SELECT geometrycollectionfromwkb(`c3
FROM `t1`)
```

这里输出的第二行就是导致服务器崩溃的语句。

- 如果错误日志不包括关于服务器崩溃的足够信息，请使用通用查询日志。

这种技术唯一一种不能帮助我们的情况是当崩溃发生在 MySQL 服务器写入通用日志时，或者在此之前。当这发生时，你可以尝试记录日志到文件中，而不是记录到表中。代理与应用端解决方案不会受此影响。

6.5 收集信息的工具

信息能指导故障排查。知道服务器进程中发生了什么是非常重要的。本书已经介绍了获取这些信息的方法，但是这里将添加一些关于被讨论工具的详细信息。

6.5.1 Information Schema

INFORMATION_SCHEMA 是提供关于数据库元数据信息的数据库，所有 SHOW 查询现在都映射到 INFORMATION_SCHEMA 表中的 SELECT 语句上。你能像任何其他表那样查询 INFORMATION_SCHEMA 表；这是它能超过其他工具的最大优势。INFORMATION_ SCHEMA 表唯一的缺点就是经过优化后也不能很快执行，所以在它上面的查询都很慢，特别是在包含很多对象信息的表上。

这里不介绍每张表，因为 MySQL 参考手册包含大量关于它们结构的细节（详见：http://dev.mysql.com/doc/refman/5.6/en/information-schema.html）。相反，我将介绍一些查询以证明你可以从 INFORMATION_SCHEMA 得到的那种有用信息。你仍然需要从手册中找到一些细节。这里把链接指向 5.6 MySQL 参表手册，是因为我提到的几个表都在这个版本中有介绍。

为了得到 INFORMATION_SCHEMA 可以做些什么的思路，让我们从当前的使用中提取每个存储引擎有多少个表开始。我从列表中排除了 mysql 数据库。因为其所有表始终都使用 MyISAM 存储引擎。

```
mysql> SELECT count(*), engine FROM tables WHERE table_schema !=
'mysql' GROUP BY engine;
+----------+--------------------+
| count(*) | engine             |
+----------+--------------------+
|      255 | InnoDB             |
|       36 | MEMORY             |
|       14 | MyISAM             |
|       17 | PERFORMANCE_SCHEMA |
+----------+--------------------+
4 rows in set (4.64 sec)
```

这信息非常有用，例如：你想选择一个用于日常备份的策略[1]。

另外一个示例是得到引用特定表的外键列表。当你访问父表并且完全不知道它连接的子表是哪一个时，如果你得到 150 错误，Foreign key constraint is incorrectly formed，这就非常有用：

```
mysql> SELECT KU.CONSTRAINT_SCHEMA, KU.CONSTRAINT_NAME,
KU.TABLE_SCHEMA, KU.TABLE_NAME FROM TABLE_CONSTRAINTS AS TC JOIN
KEY_COLUMN_USAGE AS KU ON(TC.CONSTRAINT_NAME=KU.CONSTRAINT_NAME AND
TC.CONSTRAINT_SCHEMA=KU.CONSTRAINT_SCHEMA) WHERE CONSTRAINT_TYPE='FOREIGN KEY'
AND REFERENCED_TABLE_SCHEMA='collaborate2011' AND REFERENCED_TABLE_NAME='items'
and REFERENCED_COLUMN_NAME='id'\G
*************************** 1. row ***************************
CONSTRAINT_SCHEMA: collaborate2011
  CONSTRAINT_NAME: community_bugs_ibfk_1
     TABLE_SCHEMA: collaborate2011
       TABLE_NAME: community_bugs
*************************** 2. row ***************************
CONSTRAINT_SCHEMA: collaborate2011
  CONSTRAINT_NAME: customers_bugs_ibfk_1
     TABLE_SCHEMA: collaborate2011
       TABLE_NAME: customers_bugs
*************************** 3. row ***************************
CONSTRAINT_SCHEMA: collaborate2011
  CONSTRAINT_NAME: items_links_ibfk_1
     TABLE_SCHEMA: collaborate2011
       TABLE_NAME: items_links
*************************** 4. row ***************************
CONSTRAINT_SCHEMA: collaborate2011
  CONSTRAINT_NAME: mysql_issues_ibfk_1
     TABLE_SCHEMA: collaborate2011
       TABLE_NAME: mysql_issues
*************************** 5. row ***************************
CONSTRAINT_SCHEMA: collaborate2011
  CONSTRAINT_NAME: oracle_srs_ibfk_1
     TABLE_SCHEMA: collaborate2011
       TABLE_NAME: oracle_srs
5 rows in set (9.58 sec)
```

在此输出中，你可以看到，有 5 个表与父表 items 相关联。所以如果在 items 表上执行的一个查询失败并报 150 错误，你就能快速地找出它所有的子表，并修复造成执行语句报那个错误的数据。

既然你对 INFORMATION_SCHEMA 表是什么已经有一定概念了，我们就可以切换到特定表了。

6.5.2 InnoDB 信息概要表

2.8.3 节的并发故障排查场景中，我已经介绍了 INNODB_TRX、INNODB_LOCKS 和

1：SQL 支持不同类型的备份与执行备份的方法。在规划备份时，你需要考虑对表的一些影响，例如：锁，这取决于你使用的存储引擎。7.1 节涉及备份。

INNODB_LOCK_WAITS 表。这里我还会给出其他表的快速概览。

INNODB_TRX 提供了大量有关当前运行事务的详细信息。甚至当锁跟并发并不是同样问题的时候，也可以使用它，但是在锁定问题的上下文里，可以做像找到执行很长时间的事务那样的一些事情（用与你的实际情况相关的时间替换“00:30:00”）：

```
SELECT TRX_ID, TRX_MYSQL_THREAD_ID FROM INNODB_TRX
WHERE TIMEDIFF(NOW(),TRX_STARTED) > '00:30:00';
```

可以找到哪些线程正在等待锁。

```
SELECT TRX_ID, TRX_MYSQL_THREAD_ID, TRX_REQUESTED_LOCK_ID, TRX_WAIT_STARTED
FROM INNODB_TRX
WHERE TRX_STATE = 'LOCK WAIT';
```

或者等待某一锁的时间超过指定时间。

```
SELECT TRX_ID, TRX_MYSQL_THREAD_ID, TRX_REQUESTED_LOCK_ID, TRX_WAIT_STARTED
FROM INNODB_TRX
WHERE TIMEDIFF(NOW(),TRX_WAIT_STARTED) > '00:30:00';
```

为了了解事务有多大，请检索行锁定的数目（TRX_ROWS_LOCKED），内存里锁结构的大小（TRX_LOCK_MEMORY_BYTES），或者更新的行数（TRX_ROWS_MODIFIED）：

```
SELECT TRX_ID, TRX_MYSQL_THREAD_ID, TRX_ROWS_MODIFIED
FROM INNODB_TRX ORDER BY TRX_ROWS_MODIFIED DESC;
```

还可以校验事务的隔离级别，外键检测是否打开与其他相关信息。

提示

注意，只有打开 InnoDB 表后，才会在 INNODB_TRX 表有事务显示。除了以 START TRANSACTION WITH CONSISTENT SNAPSHOT 开始的事务之外，这与 START TRANSACTION 查询，后面跟着 InnoDB 表的 SELECT 的效果一样。

命名以 INNODB_CMP 开头的表，显示了 InnoDB 使用压缩的好坏程度。因此，INNODB_CMP 与 INNODB_CMP_RESET 包含关于压缩表的状态信息，而 INNODB_CMPMEM 与 INNODB_CMPMEM_RESET 包含在 InnoDB 缓冲池里关于压缩页的状态信息。

在这些调用中增加_RESET 版本的唯一额外功能是，在查询后，它们重置所有 INNODB_CMP 表中统计信息为零。因此，如果你想重复统计，就查询_RESET 表，如果你想知道从启动以后的统计信息，仅仅需要查询 INNODB_CMP 和 INNODB_CMPMEM。

自从 5.6.2 版本后，也存在以 INNODB_SYS 和 INNODB_METRICS 开头的表。**INNODB_SYS** 表包含关于在内部词典中如何存储 InnoDB 表以及替换 InnoDB 表监控器信息。在

InnoDB 团队博客可以找到一些非常好的说明与它们使用方法的范例。在一个地方，INNODB_ METRICS 表包含关于性能与资源使用计数的所有数据。为了得到这些统计信息，需要启用一个模块。这些计数器是值得学习的，因为它能帮助你分析 InnoDB 存储引擎里面究竟发生了什么。同样，在 InnoDB 团队博客中提供了说明与范例。

6.5.3 InnoDB 监控

2.8.2 节介绍了 InnoDB 监控。以下对这一节进行了总结，并补充另外一些有用的细节。

为了启用 InnoDB 监控器，需要创建名为 innodb_monitor、innodb_lock_monitor、innodb_table_monitor 与 innodb_tablespace_monitor 的 InnoDB 表。启用这些 InnoDB 监控器就会分别地从标准、锁、表以及表空间监控器定期写入到标准错误（STDERR）输出中。

为这些表定义什么样的表结构或者添加到哪个数据库里，都没有关系，只要它们使用 InnoDB 存储引擎即可[1]。

在数据库停止时会关闭监控器。为了在启动后重新开启它，需要重新创建这些表。如果你希望它们能自动创建，就需要把 DROP 与 CREATE 语句添加到 init-file 选项中。

标准的监控器包含的输出信息类似于以下输出，以下输出来自于 MySQL 5.5 版本。下面将会对输出代码进行分段解释。

```
mysql> SHOW ENGINE INNODB STATUS\G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
110910 14:56:10 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 7 seconds
-----------------
BACKGROUND THREAD
-----------------
```

前面输出中的最后一段文字显示，这个输出主要关注主要后台线程所处理的事。

```
srv_master_thread loops: 95 1_second, 89 sleeps, 7 10_second, 36 background, 36 flush
```

该字据统计从 InnoDB 启动以后的活动。上面输出的 5 个数字分别表示，“每秒一次”循环的迭代次数，“每秒一次”循环调用 sleep 的次数，“每 10 秒一次”循环的迭代次数，称为“background_loop”的循环迭代，其运行在当前没有活跃用户时的后台操作，以及“flush_loop”标签代反弹的循环迭。所有这些循环都被主线程运行，这里主线程主要做清理与其他后台操作。

1：innodb_monitor、innodb_lock_monitor、innodb_table_monitor 与 innodb_tablespace_monitor 不应当作实际表来使用，而是提供一种方法来告诉 InnoDB 记录调试信息到标准错误输出（STDERR）。尽管你能像任何其他表那样使用它们，但是一定要做好当服务器重启后它内容消失的准备。

```
srv_master_thread log flush and writes: 116
```

这里表示写入与刷新日志的次数。

```
----------
SEMAPHORES
----------
```

从这里开始介绍内部信号量。关于这方面，第 2 章已经涉及了一些。高数字在这里表示慢的磁盘 I/O 或高的 InnoDB 争用。在后一种情况中，可以尝试降低

innodb_thread_concurrency，看是否有所改善。需要注意的是，这些数字采集自最近的 InnoDB 启动，因此这里的信息显示有等待不代表真正有等待。你需要查询 Performance Schema 库或者检查互斥锁的状态，来确定当前是否有等待发生。

```
OS WAIT ARRAY INFO: reservation count 519, signal count 476
```

这部分开始显示全局的等待数组信息。第一个数字表示当数组创建后，单元格预留的一个计数，第二个数字表示对象收到通知的次数。

```
Mutex spin waits 212, rounds 6360, OS waits 169
```

上一行显示，在互斥调用上等待的个数，自旋循环迭代个数，以及操作系统调用的等待个数。

```
RW-shared spins 171, rounds 5130, OS waits 171
```

这一行显示在共享（读）锁期间，在读写锁存器（rw-latches）上自旋等待个数，自旋循环迭代个数，以及操作系统调用等待个数。

```
RW-excl spins 55, rounds 5370, OS waits 151
```

这一行显示在排他（写）锁期间，在读写锁存器（rw-latches）上自旋等待个数，自旋循环迭代个数，以及操作系统调用等待个数。

```
Spin rounds per wait: 30.00 mutex, 30.00 RW-shared, 97.64 RW-excl
```

这表示，对于每一个互斥锁，操作系统调用等待的每一个自旋循环迭代个数。

在这一节里，以下是在 UPDATE 查询执行期间，值如何更改的一个范例：

```
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 1197, signal count 1145
--Thread 6932 has waited at trx0rec.c line 1253 for 0.00 seconds the semaphore:
X-lock (wait_ex) on RW-latch at 03CD2028 created in file buf0buf.c line 898
a writer (thread id 6932) has reserved it in mode  wait exclusive
number of readers 1, waiters flag 0, lock_word: ffffffff
Last time read locked in file buf0flu.c line 1292
Last time write locked in file ..\..\..\mysqlcom-pro-5.5.13\storage\innobase\trx\
trx0rec.c line 1253
Mutex spin waits 1163, rounds 33607, OS waits 659
RW-shared spins 248, rounds 7440, OS waits 248
RW-excl spins 47, rounds 8640, OS waits 280
Spin rounds per wait: 28.90 mutex, 30.00 RW-shared, 183.83 RW-excl
```

当查询开始执行时，前面的输出可能出现并尝试预留互斥锁。

```
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 1324, signal count 1246
--Thread 5680 has waited at buf0buf.c line 2766 for 0.00 seconds the semaphore:
Mutex at 038BE990 created file buf0buf.c line 1208, lock var 1
waiters flag 1
Mutex spin waits 1248, rounds 36397, OS waits 745
RW-shared spins 252, rounds 7560, OS waits 252
RW-excl spins 53, rounds 9750, OS waits 310
Spin rounds per wait: 29.16 mutex, 30.00 RW-shared, 183.96 RW-excl
```

在文件 buf0buf.c 的事 *2766* 行定义创建互斥锁时，这个会出现得晚一点。

在信号量部分，你应该检查是否有值变得很大与是否有很多操作等待互斥锁很长时间。

```
------------
TRANSACTIONS
------------
```

2.3 节已经深入介绍过事务。所以这里仅仅只会涉及几个要点。

```
Trx id counter 4602
```

前面一行是下一个事务号。

```
Purge done for trx's n:o < 4249 undo n:o < 0
```

这表示所有编号小于 4249 的事务都已经从历史记录列表中清除了，其中在事务列表中，为了访问相同表中正运行的事务，历史记录列表包含了提供一致性读的记录，但在提交时在修改它们之前。第二个数字表示，有多少撤销编号小于 4249 的记录从历史记录列表被清除。

```
History list length 123
```

这是历史列表的长度（未被清理的已提交事务的撤消日志）。如果这个值变得很大，你可以认为性能下降。这没有线性关系，因为清理性能同样依赖于该列表保留的事务数据总大小，所以很难给出较大值导致性能降低的一个确切例子。这个列表里的大值同样也意味着你有运行很长时间且没有关闭的事务，因为仅当没有事务指向该条目时，这些条目才会被清除。

```
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 4601, not started, OS thread id 33716224
MySQL thread id 6906, query id 123 localhost root
show engine innodb status
```

前面的行当前运行的事务列表开始。2.3 节介绍了详细内容，所以这里不会重复介绍。

```
--------
FILE I/O
--------
```

这一节开始介绍关于执行各种 I/O 操作的 InnoDB 内部线程。可以使用它来找出 InnoDB 执行了多少次 I/O 操作。该速率显示了它们多么有效。

```
I/O thread 0 state: waiting for i/o request (insert buffer thread)
I/O thread 1 state: waiting for i/o request (log thread)
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (read thread)
I/O thread 4 state: waiting for i/o request (read thread)
I/O thread 5 state: waiting for i/o request (read thread)
I/O thread 6 state: waiting for i/o request (write thread)
I/O thread 7 state: waiting for i/o request (write thread)
I/O thread 8 state: waiting for i/o request (write thread)
I/O thread 9 state: waiting for i/o request (write thread)
```

这显示了 InnoDB 内部线程的当前状态。每一行上的值括号内为线程名。

```
Pending normal aio reads: 1 [1, 0, 0, 0] , aio writes: 9 [6, 0, 3, 0] ,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 1
Pending flushes (fsync) log: 0; buffer pool: 0
```

这是关于挂起操作的信息。aio 是异步 IO 的缩写。

```
7204 OS file reads, 10112 OS file writes, 711 OS fsyncs
```

这表示从 InnoDB 启动后的总统计信息。

```
21.71 reads/s, 16384 avg bytes/read, 78.13 writes/s, 3.00 fsyncs/s
```

这表示从最近一次显示后的总统计信息。

```
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
```

顾名思义，这一部分是关于插入缓冲区与自适应散列统计信息的。使用此信息来找出它们多么有效。

```
Ibuf: size 1, free list len 0, seg size 2, 1724 merges
```

它们分别是，在页中插入缓冲索引树的当前大小，空闲列表的长度，在包含插入缓冲树与头信息的文件段中已分配页的个数，被合并页的个数。

```
merged operations:
 insert 15, delete mark 1709, delete 0
```

这表示通过类型区分，索引页被执行合并操作的次数。

```
discarded operations:
 insert 0, delete mark 0, delete 0
```

这表示操作无须合并丢弃操作的数量，因为表空间或索引已被删除。

```
Hash table size 195193, node heap has 1 buffer(s)
```

这表示在自适应散列索引表中单元格的数量与预留缓冲结构的数量。

```
0.00 hash searches/s, 40.71 non-hash searches/s
```

这表示成功使用自适应散列索引查找的数量，与当不能使用自适应散列索引时向下搜索 B 树的次数。这些统计在每次查询后都被重置。

```
---
LOG
---
```

这一部分是关于 InnoDB 日志中活动信息的。

```
Log sequence number 2055193301
Log flushed up to   2055180837
Last checkpoint at  2054187263
```

这表示当前日志序列号（LSN），LSN 日志文件刷新到达的 LSN，最近检查点的 LSN。这个信息允许你通过 Log flushed up to - Last checkpoint at 相减来计算检查点（checkpoint）周期，本例中为 993574。你需要确保检查点周期不能接近 innodb_log_file_size* innodb_log_files_in_group 值的 70%，因为在这个比例，InnoDB 认为，当前 LSN 日志与缓冲池里最老页 LSN 的差值太大，而导致主动的刷新。这可能导致数据库冻结。

```
0 pending log writes, 0 pending chkp writes
357 log i/o's done, 1.29 log i/o's/second
```

这表示挂起的日志写入个数，挂起的检查点写入个数，从 InnoDB 启动后的 IO 操作个数，与从最近一次显示之后的每秒 I/O 操作个数。

```
----------------------
BUFFER POOL AND MEMORY
----------------------
```

这表明开始显示 InnoDB 缓冲池与内存使用情况的信息。使用此信息来评估如何有效地使用 InnoDB 缓冲池。

```
Total memory allocated 49938432; in additional pool allocated 0
```

前一行显示分配的内存总数与分配了多少额外池（additional pool）。

```
Dictionary memory allocated 23269
```

这表示被数据字典表与索引对象所占空间的字节数。

```
Buffer pool size   3008
Free buffers       0
```

这显示页面中缓冲池的个数与其中空闲缓冲区的个数。这里你看到缓冲区已经满了并且有必要增加它。在此情况下，我的计算机上它设置为默认值。所以我还有增加的余地。

```
Database pages     3007
Old database pages 1090
Modified db pages  860
```

InnoDB 缓冲池将对象存储在一个列表里，该列表使用带有中点插入策略的最近最少使用算法（LRU）。当有新块需要添加时，InnoDB 会把它放进列表的中间。为了新块，最近最少使用的块会从列表中移到空闲空间里。这些统计数据显示当前 InnoDB 缓冲区 LRU 队列的长度，旧的 LRU 队列的长度，以及需要刷新的页面的数量。

提示

InnoDB 中间点插入策略实际上管理两个列表：最近访问的新（年轻）块 0 子列表块与最近不访问的旧块子列表。来源于旧块子列表的块将作为被剔除的候选。

```
Pending reads 2
Pending writes: LRU 0, flush list 10, single page 0
```

第一行显示挂起读操作的个数。第二行显示通过 LRU 算法，等待刷新的页数，在 BUF_FLUSH_LIST 列表中等待刷新的页数，在 BUF_FLUSH_SINGLE_PAGE 列表中等待刷新的页数[1]。

```
Pages made young 3508, not young 0
16.71 youngs/s, 0.00 non-youngs/s
```

第一行显示因为最近第一次被访问时，变为新页面的数目，后面为没有变为新页面的数目。第二行显示从最近显示这些值以后每秒的速率。

```
Pages read 7191, created 1871, written 9384
21.43 reads/s, 5.57 creates/s, 74.13 writes/s
```

第一行显示读操作的页面数目，在缓冲池中创建但是没有读取的页面数目，以及写操作的页面数目。第二行显示这些值的每秒速率。

```
No buffer pool page gets since the last printout
```

在我的一个测试输出中，从最近一次显示后我已经没有访问过缓冲池。如果我访问过，在上面的输出中，将会输出更多的信息。

```
Buffer pool hit rate 937 / 1000, young-making rate 49 / 1000 not 0 / 1000
```

此行显示三个比率。第一个是读取到的页面数与获得的缓冲池页面数的比例。第二个是变为新页面的页面数与获得的缓冲池页面的比例。第二个是没有变为新页面的页面数与获取的缓冲池页面的比例。每次查询后，所有这些值都重置。

```
Pages read ahead 0.00/s, evicted without access 0.00/s
```

这表示预读的速率与不通过访问剔除的预读页面的个数。这是从最近显示后测量的每秒的平均值。

```
LRU len: 3007, unzip_LRU len: 0
I/O sum[3937]:cur[1], unzip sum[0]:cur[0]
```

第一行显示的是 LRU 列表的长度与 unzip_LRU 列表的长度。后者是持有压缩文件页与相应的未压缩页面框架的一个常见 LRU 列表子集。第二行显示在普通的 LRU 与 unzip_LRU 列表中，I/O 操作的次数与为当前间隔的 I/O。

```
--------------
ROW OPERATIONS
--------------
```

行操作部分以主线程信息开始。

```
1 queries inside InnoDB, 0 queries in queue
1 read views open inside InnoDB
```

第一行显示当前有多少个正在执行的查询与在 innodb_thread_concurrency 队列中的查

1：当前缓冲区有两个刷新类型。BUF_FLUSH_LIST 根据脏块刷新列表刷新，而 BUF_FLUSH_SINGLE_PAGE 刷新单个页面。

询个数。第二行显示只读视图的数量。

```
Main thread id 4192, state: flushing buffer pool pages
```

上一行显示主线程 ID 及其状态。我是在 Windows 上运行该示例。在 Linux 上，它也会输出线程运行号。

```
Number of rows inserted 0, updated 1759, deleted 0, read 1765
0.00 inserts/s, 5.86 updates/s, 0.00 deletes/s, 5.86 reads/s
```

第一行显示，自从 InnoDB 启动以后，插入、更新、删除、读取的行数。第二行显示从最近一次显示后，每秒的速率。知道哪种查询是你最经常执行的有助于有效地设置 InnoDB 选项。

```
----------------------------
END OF INNODB MONITOR OUTPUT
============================

1 row in set (0.00 sec)
```

2.8.2 节详细介绍了 InnoDB 锁监控器，所以这里就不再介绍它。

还剩两个监控器需要介绍：InnoDB 表空间监控器与 InnoDB 表监控器。

InnoDB 表监控器输出 InnoDB 内部字典的内容。可以使用该监控器查看 InnoDB 是如何存储表的，例如，如果你怀疑它已经损坏。示例输出如下所示。

```
===========================================
110911 15:27:40 INNODB TABLE MONITOR OUTPUT
===========================================
--------------------------------------
TABLE: name collaborate2011/customers_bugs, id 1110, flags 1, columns 5, indexes 3,
appr.rows 0
  COLUMNS: iid: DATA_INT DATA_BINARY_TYPE len 4; bugid: DATA_INT
  DATA_BINARY_TYPE len 4; DB_ROW_ID: DATA_SYS prtype 256 len 6; DB_TRX_ID:
  DATA_SYS prtype 257 len 6; DB_ROLL_PTR: DATA_SYS prtype 258 len 7;
  INDEX: name GEN_CLUST_INDEX, id 2960, fields 0/5, uniq 1, type 1
   root page 3, appr.key vals 0, leaf pages 1, size pages 1
   FIELDS:  DB_ROW_ID DB_TRX_ID DB_ROLL_PTR iid bugid
  INDEX: name iid, id 2961, fields 2/3, uniq 2, type 2
   root page 4, appr.key vals 0, leaf pages 1, size pages 1
   FIELDS:  iid DB_ROW_ID
  FOREIGN KEY CONSTRAINT collaborate2011/customers_bugs_ibfk_1:
  collaborate2011/customers_bugs ( iid )
             REFERENCES collaborate2011/items ( id )
--------------------------------------
TABLE: name collaborate2011/items, id 1106, flags 1, columns 9, indexes 1,
appr.rows 5137
  COLUMNS: id: DATA_INT DATA_BINARY_TYPE DATA_NOT_NULL len 4; short_description:
  DATA_VARMYSQL len 765; description: DATA_BLOB len 10; example: DATA_BLOB len
  10; explanation: DATA_BLOB len 10; additional: DATA_BLOB len 10; DB_ROW_ID:
  DATA_SYS prtype 256 len 6; DB_TRX_ID: DATA_SYS prtype 257 len 6; DB_ROLL_PTR:
  DATA_SYS prtype 258 len 7;
  INDEX: name PRIMARY, id 2951, fields 1/8, uniq 1, type 3
```

```
 root page 3, appr.key vals 5137, leaf pages 513, size pages 545
 FIELDS:  id DB_TRX_ID DB_ROLL_PTR short_description description example
 explanation additional
FOREIGN KEY CONSTRAINT collaborate2011/community_bugs_ibfk_1: collaborate2011/
community_bugs ( iid )
           REFERENCES collaborate2011/items ( id )
FOREIGN KEY CONSTRAINT collaborate2011/customers_bugs_ibfk_1: collaborate2011/
customers_bugs ( iid )
           REFERENCES collaborate2011/items ( id )
FOREIGN KEY CONSTRAINT collaborate2011/items_links_ibfk_1: collaborate2011/
items_links ( iid )
           REFERENCES collaborate2011/items ( id )
FOREIGN KEY CONSTRAINT collaborate2011/mysql_issues_ibfk_1: collaborate2011/
mysql_issues ( iid )
           REFERENCES collaborate2011/items ( id )
FOREIGN KEY CONSTRAINT collaborate2011/oracle_srs_ibfk_1: collaborate2011/
oracle_srs ( iid )
           REFERENCES collaborate2011/items ( id )
```

这个输出显示示例 1-1 中表的信息与同一个数据库里其他表的信息。这个输出不言自明，并且在 MySQL 参考手册里，有关于输出的详细说明，所以这里不介绍这些字段。我只想把它放在这里确保你能熟悉它是什么样。

InnoDB 表空间监控器显示在共享表空间中关于文件段的信息。这些信息能帮助你找到关于表空间的问题，例如：碎片或者损坏。请注意，如果你使用 innodb_file_per_table 选项，该监控器不能显示个别表空间信息。示例输出如下所示。

```
=====================================================
110911 20:33:50 INNODB TABLESPACE MONITOR OUTPUT
=====================================================
FILE SPACE INFO: id 0
size 5760, free limit 5440, free extents 51
not full frag extents 5: used pages 290, full frag extents 3
first seg id not used 857
SEGMENT id 1 space 0; page 2; res 1568 used 1339; full ext 20
fragm pages 32; free extents 0; not full extents 4: pages 27
SEGMENT id 2 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 3 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
...
```

这个输出的意思在 MySQL 参考手册的“InnoDBTablespaceMonitorOutput”一节已经清晰地解释。所以这里同样不会重复一遍。

6.5.4 Performance Schema

2.8.4 节已经介绍了如何使用 Performance Schema 来调研锁的问题，但是它还有许多其他性能相关的用法。这里将介绍一组以 SETUP_名字开头的表，并且让你控制哪些事件是被监控的。下面是一些示例的内容：

```
mysql> SELECT * FROM setup_consumers LIMIT 2;
+----------------------+---------+
| NAME                 | ENABLED |
+----------------------+---------+
| events_waits_current | YES     |
| events_waits_history | YES     |
+----------------------+---------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM setup_instruments LIMIT 2;
+-----------------------------------------+---------+-------+
| NAME                                    | ENABLED | TIMED |
+-----------------------------------------+---------+-------+
| wait/synch/mutex/sql/PAGE::lock         | YES     | YES   |
| wait/synch/mutex/sql/TC_LOG_MMAP::LOCK_sync | YES | YES   |
+-----------------------------------------+---------+-------+
2 rows in set (0.43 sec)

mysql> SELECT * FROM setup_timers;
+------+------------+
| NAME | TIMER_NAME |
+------+------------+
| wait | CYCLE      |
+------+------------+
1 row in set (0.00 sec)
```

系统变量控制有多少事件存储在历史记录表里。

名称以_INSTANCES 结尾的表记录哪个对象正在被检测。对象的类型是每个表名的一部分。

```
mysql> SELECT * FROM FILE_INSTANCES WHERE FILE_NAME LIKE '%ITEMS%'
LIMIT 2\G
*************************** 1. row ***************************
 FILE_NAME: /users/apple/Applications/mysql-trunk/data/collaborate2011/items_links.ibd
EVENT_NAME: wait/io/file/innodb/innodb_data_file
OPEN_COUNT: 1
*************************** 2. row ***************************
 FILE_NAME: /users/apple/Applications/mysql-trunk/data/collaborate2011/items.ibd
EVENT_NAME: wait/io/file/innodb/innodb_data_file
OPEN_COUNT: 1
2 rows in set (0.08 sec)

mysql> SELECT * FROM RWLOCK_INSTANCES LIMIT 2\G
*************************** 1. row ***************************
                    NAME: wait/synch/rwlock/innodb/index_tree_rw_lock
    OBJECT_INSTANCE_BEGIN: 503973272
WRITE_LOCKED_BY_THREAD_ID: NULL
     READ_LOCKED_BY_COUNT: 0
*************************** 2. row ***************************
                    NAME: wait/synch/rwlock/innodb/index_tree_rw_lock
    OBJECT_INSTANCE_BEGIN: 503813880
WRITE_LOCKED_BY_THREAD_ID: NULL
     READ_LOCKED_BY_COUNT: 0
2 rows in set (0.08 sec)
```

```
mysql> SELECT * FROM MUTEX_INSTANCES LIMIT 2\G
*************************** 1. row ***************************
                NAME: wait/synch/mutex/innodb/rw_lock_mutex
OBJECT_INSTANCE_BEGIN: 491583300
  LOCKED_BY_THREAD_ID: NULL
*************************** 2. row ***************************
                NAME: wait/synch/mutex/innodb/rw_lock_mutex
OBJECT_INSTANCE_BEGIN: 345609668
  LOCKED_BY_THREAD_ID: NULL
2 rows in set (0.00 sec)

mysql> SELECT * FROM COND_INSTANCES LIMIT 2\G
*************************** 1. row ***************************
                NAME: wait/synch/cond/innodb/commit_cond
OBJECT_INSTANCE_BEGIN: 10609120
*************************** 2. row ***************************
                NAME: wait/synch/cond/sql/MYSQL_BIN_LOG::update_cond
OBJECT_INSTANCE_BEGIN: 35283728
2 rows in set (0.00 sec)
```

名称以 EVENT_WAITS_开头的表存储关于事件的信息。

```
mysql> SELECT COUNT(*), EVENT_NAME FROM EVENTS_WAITS_CURRENT GROUP
BY EVENT_NAME;
+----------+----------------------------------------+
| count(*) | EVENT_NAME                             |
+----------+----------------------------------------+
|        1 | wait/io/table/sql/handler              |
|        6 | wait/synch/mutex/innodb/ios_mutex      |
|        1 | wait/synch/mutex/innodb/kernel_mutex   |
|        1 | wait/synch/mutex/innodb/log_sys_mutex  |
|        1 | wait/synch/mutex/innodb/rw_lock_mutex  |
|        1 | wait/synch/mutex/innodb/thr_local_mutex |
|        2 | wait/synch/mutex/sql/LOCK_thread_count |
+----------+----------------------------------------+
7 rows in set (0.26 sec)
```

这里使用 COUNT，因为知道有多少事件被执行可以帮助你找到它们对于你的 MySQL 负载的贡献。

名称以_HISTORY 结尾的表存储关于哪个事件发生了的信息，名称以_SUMMARY 结尾的表包含基于各种参数的事件总结。

这里将给出使用这些表的示例。例如，你可以找到哪些实例使用最多的时间或者锁定最长的时间。这可以揭示某些方面的性能，而这些性能可以提升。

```
mysql> SELECT COUNT(*), (TIMER_END-TIMER_START) AS TIME,
EVENT_NAME FROM EVENTS_WAITS_HISTORY_LONG GROUP BY EVENT_NAME ORDER BY TIME
DESC;
+----------+---------+----------------------------------------+
| count(*) | time    | EVENT_NAME                             |
+----------+---------+----------------------------------------+
|     9967 | 3289104 | wait/io/table/sql/handler              |
|       10 | 2530080 | wait/synch/mutex/innodb/log_sys_mutex  |
|        5 | 2439720 | wait/synch/mutex/innodb/kernel_mutex   |
```

```
|         2 | 1481904 | wait/synch/mutex/mysys/THR_LOCK::mutex   |
|         2 | 1102392 | wait/synch/rwlock/sql/MDL_lock::rwlock   |
|         1 | 1036128 | wait/synch/rwlock/sql/LOCK_grant         |
|         2 |  789144 | wait/synch/mutex/mysys/THR_LOCK_lock     |
|         2 |  457824 | wait/synch/mutex/sql/LOCK_plugin         |
|         5 |  415656 | wait/synch/mutex/sql/THD::LOCK_thd_data  |
|         2 |  343368 | wait/synch/mutex/sql/MDL_map::mutex      |
|         2 |  325296 | wait/synch/mutex/sql/LOCK_open           |
+-----------+---------+------------------------------------------+
11 rows in set (0.26 sec)
```

6.5.5 Show [GLOBAL] STATUS

第 3 章介绍了与配置选项相关的状态变量。这里会补充其他状态变量的信息。像服务器变量，状态变量可以为全局与单个会话。当客户端连接时，会话变量会设置为 0。全局变量显示从服务器启动或最近执行 FLUSH STATUS 查询之后的状态。

当通过状态变量来进行故障诊断时，不能仅仅查看单独的值，而应该随着时间的推移持续观察。一个非常大的值本身并不意味着任何情况；也许你足够幸运经历多年的正常运行。如果它很大并且增长速度很快，你可能就能看到一个问题的迹象。

在临界负载下，我建议我们的客户在每 5 ~ 10 分钟的时间间隔里，通过 SHOW GLOBAL STATUS 采集其输出，然后对比不同时间段的变量值。这是找到有用信息最简单的方式。

下面的列表主要集中在一些特定类型的状态变量上。

Com_*状态变量

这些变量包含的数量执行的各种类型。例如，Com_select 显示有多少 SELECT 查询执行，Com_begin 显示有多少事务开始。

使用这些变量能得到负载的一个概述。例如，如果你得到一个大的 Com_select 值，与之相关的 Com_insert、Com_update、与 Com_delete 是 0，你就可以针对偏爱的 SELECT 查询来调整配置选项。

Handler_*、Select_*和 Sort_*变量

Handler_*变量显示当查询执行时，表内部发生了什么。例如，Handler_delete 显示实际上有多少行被删除。可以使用此变量来观察在当前运行的一个大表上，执行 DELETE 的进度。

Select_*变量显示使用到的不同连接的数目。Sort_*变量显示关于排序的信息。它们能帮你找到你的查询在性能方面的效果如何。

1.6.5 节介绍了这些变量，所以不会在这里花更多的时间，但是它们确实值得详细研究。

Innodb_*变量

顾名思义，这些变量显示了 InnoDB 存储引擎的内部状态。当使用 InnoDB 引擎时，研究它们并确定它们如何被不同的 InnoDB 选项所影响。

Performance_schema_*变量

performance schema 提供关于“监测点”对象的信息，其中监测点是在 MySQL 服务器或者存储引擎源代码中创建的。这些变量显示有多少监测不能加载或创建。

Ssl_*变量

这些显示关于 SSL 连接的统计信息。

*open*和*create*变量

包含这些关键字的变量显示有多少不同类型的对象被打开或关闭。

其他状态变量的目的或者可以通过它们的名字能推断出来或者在第 3 章中能找到。

6.6 本地化问题（最小化测试用例）

1.5 节已经介绍了最小化测试用例的价值，其中把不正确的 SELECT 化简为再现错误参数的 CREATE TABLE。这里将从更通用的方面介绍，最小测试案例的原则。

作为一个例子，将执行如下大的查询[1]。

```
SELECT
IF(TABLE1.FIELD1 = 'R' AND TABLE1.FIELD2 IS NOT NULL AND TABLE1.FIELD3 = '1' AND
TABLE2.FIELD4 = TABLE2.FIELD5 AND TABLE3.FIELD6 = TABLE4.FIELD6, TABLE3.FIELD7,
TABLE4.FIELD7) AS ALIAS1,

IF(TABLE1.FIELD1 = 'R' AND TABLE1.FIELD2 IS NOT NULL AND TABLE1.FIELD3 = '1' AND
TABLE2.FIELD4 = TABLE2.FIELD5 AND TABLE3.FIELD6 = TABLE4.FIELD6, TABLE3.FIELD8,
TABLE4.FIELD8) AS ALIAS2,

SUM(
IF (
(SELECT TABLE5.FIELD7 FROM TABLE4 ALIAS3, TABLE2 ALIAS4, TABLE4 ALIAS5 WHERE
TABLE5.FIELD5 = ALIAS4.FIELD4 AND ALIAS4.FIELD5 = ALIAS5.FIELD5 AND
ALIAS5.FIELD7 = FIELD9 AND TABLE5.FIELD6 = TABLE6.FIELD7 LIMIT 1 ) IS NULL, 0,
TABLE7.FIELD10/TABLE7.FIELD11)
) AS ALIAS11

FROM TABLE4 ,TABLE4 ALIAS6, TABLE8 , TABLE4 ALIAS7, TABLE9 , TABLE7 , TABLE2 ,
TABLE1 FORCE INDEX(FIELD12)
```

1：这个示例基于社区 bug #33794。

```
TABLE1 FORCE INDEX(FIELD12)
LEFT JOIN TABLE1 ALIAS8 ON TABLE1.FIELD13 = TABLE10.FIELD13
LEFT JOIN TABLE1 ALIAS9 ON ALIAS9.FIELD13 = TABLE10.FIELD2
LEFT JOIN TABLE4 ALIAS10 ON ALIAS10.DFIELD5 = TABLE3.FIELD5

WHERE TABLE1.FIELD14 > DATE_sub(now(), INTERVAL 16 DAY)
and TABLE1.FIELD1 IN ('P', 'R','D')
AND TABLE1.DFIELD5 = TABLE4.FIELD5
AND TABLE1.FIELD15 = TABLE8.FIELD16
AND TABLE6.FIELD7 = TABLE8.FIELD17
AND TABLE4.FIELD18 = TABLE9.FIELD18
AND TABLE1.FIELD19 = TABLE7.FIELD19
AND TABLE1.FIELD20 = TABLE2.FIELD21
AND TABLE4.FIELD6 = TABLE11.FIELD7

GROUP BY TABLE4.FIELD6, FIELD9;
```

可以将问题隔离到更小的等价语句中：

```
SELECT
IF(T1.F1 = 'R', A1.F2, T2.F2) AS A4,
IF(T1.F1 = 'R' , A1.F3, T2.F3) AS F3,
SUM( IF ( (SELECT A7.F2
FROM T2 A7, T4 A2,  T2 A3
WHERE
A7.F4 = A2.F10
AND   A3.F2 = A4
LIMIT 1 ) IS NULL, 0, T3.F5)) AS A6

FROM T2, T3, T1
JOIN T2 A1 ON T1.F9 = A1.F4

GROUP BY  A4;
```

使用这条等价语句，测试问题是很容易的。后面的章节将介绍我如何创建如此小的测试用例，然后再讨论如何使用这些测试用例。

6.7 故障排除的一般步骤

下面是我定位问题时采取的步骤。它们并非完全线性，因为针对不同的问题有不同的解决方法。但该序列可以作为一份检查列表。

尝试确定造成问题的实际查询。

我已经介绍了很多方法来确定查询或者，在并发场景下，共同导致问题的查询集合。一旦你获得了查询及其环境，你就成功了一半了。或者问题是可重复的，或者与它与并发问题相关，你都能追查这些问题。

检查以确保查询的语法正确。

最简单的方法是在 MySQL 客户端里执行语句。如果查询在语法上不正确，它将返

回一个错误。在任何情况下，一旦你找出哪一部分语法是错误的，你就能找到问题的根源，否则，继续一步。

确认查询里有问题。

如果问题是一个错误的结果或者一个性能问题，请检查，确保在 MySQL CLI 能重现。

你可能会注意到，当列出需要遵循的步骤时，我在朝不同方向扩展。因为问题的差异，最有效方法找到问题的原因，所以这些问题始终无法通过单一方法来解决。

如果查询返回错误数据时，请其尝试重写它以得到正确的结果，也就是说，你期望获得的那些结果。

如果问题是一些可重复的单个查询，它总是可以重写来反映你的需求。这个时候，你可以让你的查询变短，减少数据集并测试你做出的修改。修改查询时，不要强制单个查询产生最终结果而犯一些典型的错误。为了得到必要的结果集，一个查询的解决方案并不会总是比多个查询组成的序列要快。

如果重写没有帮助，你可以检查服务器选项并尝试确定它们是否影响结果。

此方法可以与上一个联合起来使用。有经验的用户知道对配置选项的影响何时产生怀疑。当有疑惑时，在简化语句后，尽量尝试配置选项调整。

如果你确定查询是正确的，当查询执行时，请向后查找直到找到损坏数据的一个语句或动作。

如果你确定查询是绝对正确的并且没有配置选项影响其结果，那么抱怨查询已经没有任何意义。请检查数据是否损坏并找出最初的根本原因。这可能是应用程序或者外部因素，例如：物理磁盘损坏，或操作系统级别数据库文件修改。

如果问题不能在 MySQL CLI 重现，请检查它是否是并发问题。

使用第 2 章的所有方法来找到哪里造成的问题。我从 SHOW PROCESSLIST 开始，到应用程序级别技术结尾。

如果该问题会导致系统崩溃或挂起，首先检查错误日志。

虽然最新的崩溃信息日志位于文件末尾，但请使用完整的文件，也请多关注旧的消息。它们可能包含表的损坏信息或过去发生的类似相关问题，从而使你更了解发生了什么事。

如果错误日志文件不能给出一些线索，请试着找到崩溃前的最后查询。

如前所述，可以使用通用查询日志、应用程序级别日志或代理完成该任务。

使用 mysqld-debug，从发生故障的服务器中产生核心文件，然后分析它。

如果这是一个可重复性的挂起而不是崩溃，请把调试器连接到正在运行的进程上。

分析并调整配置选项。

选项也有可能导致服务挂起而不崩溃。分析每一个可能影响的选项，然后相应地调整它们。

利用操作系统工具来找到哪些影响 mysqld 的外部进程。

在你尝试与 MySQL 服务器本身相关的任何事情并确定它是正常运行后，请检查其运行环境。

这一般的行动计划是将大的、未知的问题分解成更小的部分以便进行分析与解决。执行路径并不总是需要遵循顺序，因为有时候你能很快找到问题的原因并能马上修复。但是当原因很难确定并且你需要一些引导来找到它时，这些顺序就能显示出作用。这里跳过了实际的修复过程，因为本书前面部分已经详细介绍它们了。

在信息不全的环境中减少测试用例

在很多情况下寻找一个问题的最小测试用例是一个普遍策略，而不仅仅在 SQL 应用程序中。当我开始写这本书时，我发现我没有在自己最喜欢的编辑器上安装 XML 插件。我打开插件器管理器，发现很多插件已经过期。

这里我犯了一个错：我下载并安装了所有的插件。

这是一个糟糕的主意，其确切原因是我最喜欢的编辑器运行在 Java 虚拟机中，并且为了运行日常任务需要的一个程序，我确是需要一个过时的版本。

但自去年以来，编辑器与它的插件都切换到最新版本的 Java。故下一次启动编辑器时，我就得到很多关于插件版本的错误，最终程序就挂起了。

我确实不想重新安装所有的插件，因为我不想失去我当前的设置。我确信，问题的根源只是一个插件，而不是编辑器本身。但是安装几十个插件后，我很难确定是哪个插件阻止了编辑器启动。

于是，我打开编辑器的选项目录，然后将插件子目录的内容复制到一个安全的地方。我很惊讶地看到该编辑器还不能启动。

接下来，当我最后一次打开编辑器时，我确定哪些文件被涉及，然后也把它们也挪到另一个安全的地方。

这一次，编辑器启动了，并且重新创建了它的环境。这是不错的，但是我还是想回到偏爱的

选项。

所以我开始一个一个添加回选项文件，一次接一个地重启编辑器，直接我发现那个损坏的文件。幸运的是，它不包含在我需要恢复的选项，所以我仅仅删除它，并让编辑器重新创建它。

现在又轮到插件了，我再次把它们一个个地加到目录中，直接我启动的时候报错。检查所有插件后，我再次得到了我的工作安装环境了。

- 最小化测试用例的原则甚至可以用在一些信息有限的环境中。

6.8 测试方法

创建一个最小化测试用例可以确认问题，但通过它，你可以做更多的事。最小化测试用例，可以很容易地查明问题的原因。当你查看通过复杂的 WHERE 条件连接多个表的一个 JOIN 时，很难明确是什么错误。但减少查询到几个表可以大大缩小可能性。在最小化查询的同时有时候你可以解决问题。

- 最小化测试用例通常能揭示和解决问题。

在本节中，如果你创建的测试无法解决问题，我们就会考虑需要做什么。这可能有两个原因：MySQL 源码 bug 或对某些功能工作原理的错误理解。我假设你已经阅读了 MySQL 参考手册中有关问题的章节，并没有发现答案。但仍然有一些资源你可以尝试。

6.8.1 在新版本中尝试查询

尝试 MySQL 新版本的理由是，假设你的问题确实由服务器中的 bug 导致，你可能发现它已经修复了。安装和使用新版本 MySQL 听起来很复杂，但是如果你遵循 6.3 节中建议并有一个沙箱，它就非常容易，特别是如果你有一个最小化的数据集，并且能把沙箱与最小化测试用例结合起来。

6.8.2 检查已知的 bug

如果你尝试使用一个新版本也起作用，那么请在 bug 数据库中检查已知的 bug。你甚至可以在尝试新的 MySQL 版本前，就考虑查找 bug 报告。随着你遇到更多的问题，你会变得更有经验，你将会有先做哪个事情的灵感。你可以遵循的一个简单规则是：对于你容易创建的数据集，如果该问题可以重现，那么请先尝试新版本的 MySQL，否则，请首先查找 bug 数据库。

你开始可以从社区版 bug 数据库中搜索。如果你是 Oracle 的客户，在支持的门户网站

里，你可以使用内部的 bug 数据库。如果你发现一些案例似乎符合自己的工作场景，这将告诉你，是否认为它是一个 bug 和它是否已经修正。

如果这是一个 bug，要么下载和测试已经修正 bug 的版本，要么找到一些变通方法直到 bug 被修正。如果有一些已知的变通方法，你将会因为大众的评论而找到它。对于寻找变通的解决方法，搜索引擎也是非常有用的。

如果不认为该问题是一个 bug，bug 报告将指向 MySQL 参考手册中描述正确使用方法的那一部分。我们不详细解释，为什么一个功能是 bug 而另一个功能却不是 bug，因为这不是 MySQL bug 数据库的目的，但是学习 MySQL 软件正确的使用方法有助于找到一个解决方案。

如果你不能在 bug 数据库中找到问题，请使用你最喜欢的搜索引擎来找到被提及的类似问题。如果在你的逻辑思维中，你仍不能找到有什么问题，并且在最新的 MySQL 版本中能重现该问题，那么是时候报告这个 bug 了。

6.8.3 变通方法

如果前面章节不能帮你解决问题，你可以尝试创建你自己的变通方法。重写查询，以排除造成问题的部分，并且把查询分解成能正确执行的小查询。

下面是基于#47650 bug（目前已经修正）的一个示例，它能描述这个概念。首先，我们来看看一个触发 bug 报告错误行为的简单版本。

```
mysql> CREATE TABLE `t1` (
    ->   `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    ->   PRIMARY KEY (`id`)
    -> ) ENGINE=MyISAM;
Query OK, 0 rows affected (0.04 sec)

mysql> CREATE TABLE `t2` (
    ->   `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    ->   `t1_id` BIGINT(20) DEFAULT NULL,
    ->   PRIMARY KEY (`id`)
    -> ) ENGINE=MyISAM;
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO `t1` VALUES
(1),(2),(3),(4),(5),(6),(7),(8);
Query OK, 8 rows affected (0.00 sec)
Records: 8 Duplicates: 0 Warnings: 0
mysql> INSERT INTO `t2` VALUES
(1,1),(2,1),(3,1),(4,2),(5,2),(6,2),(7,3),(8,3);
Query OK, 8 rows affected (0.01 sec)
Records: 8 Duplicates: 0 Warnings: 0

mysql> SELECT t1.id AS t1_id, COUNT(DISTINCT t2.id) AS cnt FROM t1
LEFT JOIN t2 ON t1.id = t2.t1_id
```

```
    -> WHERE t1.id = 1 GROUP BY t1.id WITH ROLLUP LIMIT 100;
+-------+-----+
| t1_id | cnt |
+-------+-----+
|     1 |   8 |
|  NULL |   8 |
+-------+-----+
2 rows in set (0.01 sec)
```

当 wheret1_id=1 时，为什么我们得到的不是 8 行？仅插入了 t1_id=1 的 3 行

```
mysql> INSERT INTO `t2` VALUES (1,1),(2,1),(3,1),
(4,2),(5,2),(6,2),(7,3),(8,3);
Query OK, 8 rows affected (0.01 sec)
Records: 8 Duplicates: 0 Warnings: 0
```

如果删除 GROUP BY 子句，问题就能清晰可见了。

```
mysql> SELECT t1.id AS t1_id, t2.id FROM t1 LEFT JOIN t2 ON t1.id =
t2.t1_id WHERE t1.id = 1;
+-------+------+
| t1_id | id   |
+-------+------+
|     1 |    1 |
|     1 |    2 |
|     1 |    3 |
+-------+------+
3 rows in set (0.00 sec)
```

这个清单显示这个数据是正确的并且 GROUP BY 子句造成了问题。

乍一看，解决该问题的唯一办法是把一个查询分成几个查询。但是我们可以尝试另一种变通方法，了解一些有关优化器的事情。如果 GROUP BY 使用索引，那么查询执行计划可能会改变，所以让我们尝试添加一个索引：

```
mysql> ALTER TABLE t2 ADD INDEX(t1_id);
Query OK, 8 rows affected (0.05 sec)
Records: 8 Duplicates: 0 Warnings: 0
```

现在问题解决了：

```
mysql> SELECT t1.id AS t1_id, COUNT(DISTINCT t2.id) AS cnt FROM t1 LEFT JOIN t2
ON t1.id = t2.t1_id WHERE t1.id = 1 GROUP BY t1.id WITH ROLLUP LIMIT 100;
+-------+-----+
| t1_id | cnt |
+-------+-----+
|     1 |   3 |
|  NULL |   3 |
+-------+-----+
2 rows in set (0.02 sec)
```

所以玩转 SQL 来看是否能规避 bug。我展示的示例不适合一般的应用程序，但是它表明了能从细微的改变来帮助我们解决问题。这也再次显示出使用一个沙箱的优势了。

6.9 专用的测试工具

当你测试一个有多个解决方案的问题时，如一个性能问题或 SQL 应用程序的设计，你需要测试每一个单元如何满足需求。这一节提供能帮助你做此类测试的工具快速概览。

6.9.1 基准工具

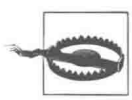
基准工具用来测试应用程序的速度。MySQL 基准工具通常用来测试 MySQL 的安装，虽然这与测试应用程序并不相同，但对于测试某一组特定的选项它们仍然很有用。如果基准工具允许你使用专门为应用程序编写的自定义查询，那么你也可以在自己的数据集上运行测试。

sysbench 和 mysqlslap 是最受欢迎的 MySQL 基准工具。下面将分别介绍它们。

1. mysqlslap

mysqlslap 是 MySQL 发布版中自带的一个负载模拟客户端。在相似查询上，它让并发负载测试变得很容易。无论是从一个文件或者为它指定一个参数，它都与 SQL 脚本一起运行。

```
$ mysqlslap  --socket=/tmp/mysql51.sock --user=root --delimiter=";" \
--create-schema=mstest --create="CREATE TABLE mstest(id INT NOT NULL \
AUTO_INCREMENT PRIMARY KEY, f1 VARCHAR(255)) ENGINE=InnoDB" --query="INSERT INTO \
mstest(f1) VALUES(MD5(RAND())); SELECT f1 FROM mstest;" --concurrency=10 \
--iterations=1000
Benchmark
        Average number of seconds to run all queries: 0.039 seconds
        Minimum number of seconds to run all queries: 0.025 seconds
        Maximum number of seconds to run all queries: 0.173 seconds
        Number of clients running queries: 10
        Average number of queries per client: 2
```



警告

请注意，如果你为 mysqlslap 指定 create 选项，在 create-schema 选项指定的架构将被删除。如果你使用的是 5.1.57 或 5.5.12 之前的 mysqlslap，即使你不使用 create 选项，这同样也会发生。

- 仅仅在新的空架构上使用该工具。

2. SysBench

该基准工具可以从 Launchpad 网站获得，它可以用来测量整个系统的性能，包括测试 CPU、文件 I/O、操作系统调度程序、POSIX 线程性能、内存分配、传输速度与数据

库服务器性能。这里重点讨论数据库服务器性能。

在早期版本中，可以仅仅使用预定义的在线事务处理（OLTP）测试来测试数据库服务器选项，它将创建一个表，并基于这个表运行并行测试。

```
$sysbench --test=./sysbench/tests/db/oltp.lua
--mysql-table-engine=innodb --oltp-table-size=1000000
--mysql-socket=/tmp/mysql60.sock --mysql-user=root prepare
sysbench 0.5:  multi-threaded system evaluation benchmark

Creating table 'sbtest1'...
Inserting 1000000 records into 'sbtest1'

$sysbench --test=./sysbench/tests/db/oltp.lua
--mysql-table-engine=innodb --oltp-table-size=1000000
--mysql-socket=/tmp/mysql60.sock --mysql-user=root --num-threads=16
--max-requests=100000 run
sysbench 0.5:  multi-threaded system evaluation benchmark
Running the test with following options:
Number of threads: 16
Random number generator seed is 0 and will be ignored


Threads started!

OLTP test statistics:
    queries performed:
        read:                            1400154
        write:                           400044
        other:                           200022
        total:                           2000220
    transactions:                        100011 (50.84 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 1800198 (915.16 per sec.)
    other operations:                    200022 (101.68 per sec.)

General statistics:
    total time:                          1967.0882s
    total number of events:              100011
    total time taken by event execution: 31304.4927s
    response time:
         min:                                  18.10ms
         avg:                                 313.01ms
         max:                               13852.37ms
         approx.  95 percentile:              595.43ms

Threads fairness:
    events (avg/stddev):           6250.6875/22.15
    execution time (avg/stddev):   1956.5308/0.65
```

以上是使用最新版本的 MySQL 的输出信息，它允许你把 sysbench 指向一个自定义测试，但这种预定义的测试行为与早期版本中使用的预定义测试是相同的。

从 0.5 版本开始，可以通过 Lua 编程语言来编写自己的测试用例。开始时最简单的方法是将 oltp.lua 作为模板。以下简单脚本说明了需要定义哪些功能：

```
$cat sysbench.lua
function prepare()
        local i

        db_connect()

        print("Creating table 'test'")
        db_query("CREATE TABLE test(id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        f1 VARCHAR(255))")
        print("Inserting 1000 rows")
        for i = 1, 1000 do
                db_query("INSERT INTO test(f1) VALUES(1000*rand(1000))")
        end

end

function cleanup()
        print("Dropping table 'test'")
        db_query("DROP TABLE test")
end

function event(thread_id)
        db_query("SELECT * FROM test WHERE f1 = " ..  sb_rand(1, 1000))
end
```

在实际的基准中，可以编写代码来适应更复杂的场景。

上述测试结果与默认的 OLTP 测试相似。

```
$sysbench
--test=/Users/apple/Documents/web_project/MySQL/examples/sysbench.lua
--mysql-table-engine=innodb
--mysql-socket=/tmp/mysql60.sock --mysql-user=root --num-threads=16
--max-requests=100000 prepare
sysbench 0.5:  multi-threaded system evaluation benchmark

Creating table 'test'
Inserting 1000 rows

$sysbench
--test=/Users/apple/Documents/web_project/MySQL/examples/sysbench.lua
--mysql-table-engine=innodb
--mysql-socket=/tmp/mysql60.sock --mysql-user=root --num-threads=16
--max-requests=100000 run
sysbench 0.5:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 16
Random number generator seed is 0 and will be ignored


Threads started!

OLTP test statistics:
    queries performed:
        read:                            100001
```

```
        write:                           0
        other:                           0
        total:                           100001
    transactions:                        0      (0.00 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 100001 (37.08 per sec.)
    other operations:                    0      (0.00 per sec.)

General statistics:
    total time:                          2697.2491s
    total number of events:              100001
    total time taken by event execution: 43139.9169s
    response time:
         min:                                  7.54ms
         avg:                                431.39ms
         max:                               2304.82ms
         approx.  95 percentile:             913.27ms

Threads fairness:
    events (avg/stddev):           6250.0625/27.35
    execution time (avg/stddev):   2696.2448/0.57
```

6.9.2 Gypsy

Gypsy 同样可以从 Launchpad 获得，它是由 MySQL 支持工程师 Shane Bester 开发的一款负载测试工具。在测试并发负载时，我们会积极地使用该工具。它不是基准工具，但是它能找到锁定问题或其他并发问题。

Gypsy 是可以脚本化的。其查询文件的语法非常简单。

```
i|1|DROP TABLE IF EXISTS t1|
i|1|CREATE TABLE t1( id INT, f1 INT, PRIMARY KEY(id)) ENGINE=InnoDB|
i|1|SET GLOBAL SQL_MODE='strict_trans_tables'|
n|100|INSERT INTO t1 SET id = ?, f1 = 1 ON DUPLICATE KEY UPDATE f1 = f1 + 1|tinyint
```

标有 i 的行是初始设置的一部分，并且只运行一次。标有 n 的行表示模拟负载的查询。可以像如下这样执行 Gypsy：

```
$gypsy --host=127.0.0.1:3351 --user=root --password= --database=test
--queryfile=bug42644.query --threads=2 --duration=100
[INFO]   04:08:15 [0290] 2684407808 - 32-bit version of Gypsy
[INFO]   04:08:15 [0291] 2684407808 - sizeof(long long int) = 8
[INFO]   04:08:15 [0300] 2684407808 - using 1 hosts
[WARN]   04:08:15 [2950] 2684407808 - setting statement on line 1 to non-prepared by
default
[WARN]   04:08:15 [2950] 2684407808 - setting statement on line 2 to non-prepared by
default
[WARN]   04:08:15 [2950] 2684407808 - setting statement on line 3 to non-prepared by
default
[INFO]   04:08:15 [0362] 2684407808 - client library version: 5.0.92
[ALWAYS] 04:08:15 [0376] 2684407808 - server 00: '5.1.60-debug', host:
'127.0.0.1 via TCP/IP',  SSL: 'NULL', protocol: 10, charset: latin1
[ALWAYS] 04:08:15 [0414] 2684407808 - thrd = 2
[INFO]   04:08:15 [0459] 2684407808 - read 4 valid queries from query file
[INFO]   04:08:15 [0556] 2684407808 - spawning data generation thread
```

```
[INFO]   04:08:15 [0693] 25182208 - creating new char data for the first time
[INFO]   04:08:15 [0711] 25182208 - refreshing char data
[INFO]   04:08:15 [0718] 25182208 - char data has been generated, char_increment=2
[INFO]   04:08:15 [0603] 2684407808 - now running for 100 seconds.
[INFO]   04:08:15 [0609] 2684407808 - running initialization queries
[INFO]   04:08:15 [1443] 25183232 - thread 0 connecting to host 0
[INFO]   04:08:15 [1456] 25183232 - thread 0 has 1 alive hosts connected
[WARN]   04:08:16 [2182] 25183232 - thread[00] didn't complete entire query
file. Might need longer --duration=
[INFO]   04:08:16 [0636] 2684407808 - about to create all 'populate' scripts from I_S
[INFO]   04:08:16 [0691] 2684407808 - spawning database stats thread
[ALWAYS] 04:08:16 [0708] 2684407808 - spawning 2 new thread(s)
[INFO]   04:08:16 [1443] 25184256 - thread 0 connecting to host 0
[INFO]   04:08:16 [0957] 25183232 - writing server status variables to
'report_18098_host_00.txt'
[INFO]   04:08:16 [1456] 25184256 - thread 0 has 1 alive hosts connected
[INFO]   04:08:16 [1443] 25188352 - thread 1 connecting to host 0
[INFO]   04:08:16 [1456] 25188352 - thread 1 has 1 alive hosts connected
[INFO]   04:08:17 [0736] 2684407808 - completed spawning new database worker threads
[INFO]   04:08:28 [0777] 2684407808 - 02 threads running, 0030487 successful
queries.  0000000 failed queries (2540.583333 QPS).
[INFO]   04:08:39 [0777] 2684407808 - 02 threads running, 0059212 successful
queries.  0000000 failed queries (2393.750000 QPS).
[INFO]   04:08:50 [0777] 2684407808 - 02 threads running, 0084904 successful
queries.  0000000 failed queries (2141.000000 QPS).
[INFO]   04:09:01 [0777] 2684407808 - 02 threads running, 0110477 successful
queries.  0000000 failed queries (2131.083333 QPS).
[INFO]   04:09:12 [0777] 2684407808 - 02 threads running, 0133212 successful
queries.  0000000 failed queries (1894.583333 QPS).
[INFO]   04:09:23 [0777] 2684407808 - 02 threads running, 0148816 successful
queries.  0000000 failed queries (1300.333333 QPS).
[INFO]   04:09:34 [0777] 2684407808 - 02 threads running, 0165359 successful
queries.  0000000 failed queries (1378.583333 QPS).
[INFO]   04:09:45 [0777] 2684407808 - 02 threads running, 0178743 successful
queries.  0000000 failed queries (1115.333333 QPS).
[ALWAYS] 04:09:56 [0792] 2684407808 - waiting for threads to finish
[INFO]   04:09:56 [0808] 2684407808 - running cleanup queries
[INFO]   04:09:56 [1443] 25188352 - thread 0 connecting to host 0
[INFO]   04:09:56 [1456] 25188352 - thread 0 has 1 alive hosts connected
[WARN]   04:09:56 [2182] 25188352 - thread[00] didn't complete entire query file.
Might need longer --duration=
[INFO]   04:09:56 [0835] 2684407808 - now about to tell stats thread to exit
[INFO]   04:09:56 [0842] 2684407808 - now about to tell data generation thread to exit
[ALWAYS] 04:09:56 [0884] 2684407808 - done!!!
[ALWAYS] 04:09:56 [0885] 2684407808 - press a key to continue!!
```

6.9.3 MySQL 测试框架

MySQL 测试框架也称为 MTR，MTR 是它的主要命令 mysql-test-run 的缩写。它是 MySQL 开发人员使用的自动测试包，并且在所有 MySQL 包都包含它。在概念上，它类似于单元测试。它涉及测试用例，其中每个测试用例都由一个测试文件和一个结果文件组成。测试文件包含几组 MySQL 查询和特定的 MTR 命令，结果文件包含预期的结果。

这里给出一个测试文件示例，它创建一个表，并插入一条数据，然后再选择它。

```
--source include/have_innodb.inc

CREATE TABLE t1(f1 int NOT NULL AUTO_INCREMENT PRIMARY KEY, f2 VARCHAR(255))
ENGINE=InnoDB;
INSERT INTO t1 (f2) VALUES('test');
SELECT f1, f2 FROM t1;
DROP TABLE t1;
```

我们假定你已经将以上信息保存在 MRT 的 t 子目录下一个名为 book.test 的文件中。如果你确定自己的 MySQL 版本运行良好，就可以通过以下命令来自动记录一个结果文件。

```
$./mtr --record book
Logging: ./mtr  --record book
110915  3:58:38 [Warning] Setting lower_case_table_names=2 because file system
for /tmp/PrqusdwLQa/ is case insensitive
110915  3:58:39 [Note] Plugin 'FEDERATED' is disabled.
110915  3:58:39 [Note] Plugin 'ndbcluster' is disabled.
MySQL Version 5.1.60
Checking supported features...
 - skipping ndbcluster
 - SSL connections supported
 - binaries are debug compiled
Collecting tests...
vardir: /users/apple/bzr/mysql-5.1/mysql-test/var
Checking leftover processes...
Removing old var directory...
Creating var directory '/users/apple/bzr/mysql-5.1/mysql-test/var'...
Installing system database...
Using server port 56970

==========================================================================

TEST                                      RESULT   TIME (ms)
------------------------------------------------------------

worker[1] Using MTR_BUILD_THREAD 300, with reserved ports 13000..13009
main.book                                 [ pass ]    124
------------------------------------------------------------
The servers were restarted 0 times
Spent 0.124 of 20 seconds executing testcases

Completed: All 1 tests were successful.
```

现在结果文件的内容如下所示。

```
$cat r/book.result
CREATE TABLE t1(f1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY, f2 VARCHAR(255))
ENGINE=InnoDB;
INSERT INTO t1 (f2) VALUES('test');
SELECT f1, f2 FROM t1;
f1      f2
1       test
DROP TABLE t1;
```

当不使用--record 选项来执行相同的测试时，MTR 会将实际结果与结果文件的内容进行比较，如果两者有差异，就会失败。也可以把测试文件与结果文件移到其他服务器中

进行测试，并查看执行结果是否会改变。

从来自于 MySQL 5.1 服务器以及更高 MySQL 版本的 2.0 版本开始，MTR 允许用户为单独产品创建套件，所以，就可以像以下这样创建自己的测试套件：

```
$./mtr --suite=./suite/book book
Logging: ./mtr  --suite=book book
110915  4:05:29 [Warning] Setting lower_case_table_names=2 because file system
for /tmp/7npx97ZLbz/ is case insensitive
110915  4:05:29 [Note] Plugin 'FEDERATED' is disabled.
110915  4:05:29 [Note] Plugin 'ndbcluster' is disabled.
MySQL Version 5.1.60
Checking supported features...
 - skipping ndbcluster
 - SSL connections supported
 - binaries are debug compiled
Collecting tests...
vardir: /users/apple/bzr/mysql-5.1/mysql-test/var
Checking leftover processes...
Removing old var directory...
Creating var directory '/users/apple/bzr/mysql-5.1/mysql-test/var'...
Installing system database...
Using server port 56998

========================================================================

TEST                                      RESULT   TIME (ms)
------------------------------------------------------------

worker[1] Using MTR_BUILD_THREAD 300, with reserved ports 13000..13009
book.book                                 [ pass ]     72
------------------------------------------------------------
The servers were restarted 0 times
Spent 0.072 of 12 seconds executing testcases

Completed: All 1 tests were successful.
```

除了开发 MySQL 以外，在自动检测具有不同选项或不同 MySQL 版本的安装是如何执行自定义语句时，这个工具是非常有用的。

可以在 MTR 用户指南中找到更多信息。

6.10 维护工具

在 DBA 日常工作中，这一节中的工具都非常有用，不仅仅是在故障发生时。如果你经常使用它们，那么你可以避免最麻烦的情况。这就是为什么我要在这里介绍它们：我撰写本书的原因之一是为了帮助你顺畅地安装 MySQL。

像参考手册那样，这些工具都有非常好的用户指南，所以这里仅仅介绍它们而不会介绍更多的细节。

来源于MySQL发布版的工具

可以在MySQL安装包的bin目录下找到MySQL附带的工具集。MySQL参考手册的“MySQL 程序”（*http://dev.mysql.com/doc/refman/5.5/en/programs.html*）一节介绍了这些工具，请掌握它们。

Percona Toolkit

这个工具包可以从Percona网站获得，它的Aspersa与Maatkit发布版联合组成。它包含大量控制服务器和表的强大工具。

MySQL WB 实用工具

尽管它们是MySQL Workbench安装包的一部分，但是它们在命令行模式下运行。它们与MySQL Workbench完全独立并且它们能运行在MySQL Workbench不能运行的平台上。这些工具使用Python语言编写。它们主要与数据库结构一起使用，而不是数据，并且它们可以让一些工作自动化，例如复制搭建和权限迁移。

监控工具

监控系统中有关日常操作的统计是非常重要的。这里介绍两个工具来帮助你监控MySQL安装。

MySQL Enterprise Manager

MySQL团队提供MySQL Enterprise Manager（MEM），主要目的是为了监控。

MEM是为企业客户提供的一个商业软件。它作为服务器运行在专用的机器上，与MEM一起的是一个轻量级代理，它与需要被监控且正在运行的MySQL服务器运行在相同的机器上。这些代理收集MySQL和操作系统的信息，然后发到MEM服务器上。通过基于Web的图形界面，DBA可以得到一个跨越多台服务器的图形概述。在服务器设置中，MEM还能给出改进建议。

可以在Charles Bell等撰写的*MySQL High Availability*（O'Reilly出版社）一书或者官方文档中了解到关于MEM的更多细节。

dim_STAT

dim_STAT是一个监控UNIX、Linux与Solaris系统的性能工具。它收集操作系统监控工具，如vmstat、iostat的输出，它本身拥有多个插件，可以用于收集不同的统计信息、绘制图表等。dim_STAT支持多主机。

dim_STAT同样有MySQL插件件来监控MySQL与InnoDB的使用。

可以检查统计信息和使用基于 Web 的 GUI 查看图。dim_STAT 同样也有一个 CLI 解决方案，可以通过给定的数据库、收集 ID 和时间间隔来生成单幅 PNG 格式的图。

可以从 http://dimitrik.free.fr/下载 dim_STAT。

第7章

最佳实践

本章总结了一些可以帮助你用最有效、最安全的解决 MySQL 问题的最佳实践。它们不是问题排查方法或工具本身，但是它们能极大地影响问题排查。整本书都介绍这些实战技巧，但本章把它们结合起来应用，以强调其价值。

7.1 备份

本书中的很多程序都可以对数据库做出变更，这就是我鼓励你在沙箱中运行测试的原因。然而，在生产数据库中做出这些变更可能会为自己带来灾难。备份可以在这种情况下提供帮助。

备份是一种状态，可以在任何一个阶段返回这个状态。如果在测试或应用程序故障期间，某些东西被破坏，你就可以恢复全部最近更改的数据。当你因为测试从备份中加载数据到沙箱时，备份也非常有用。如果在适当的时候进行备份，那么无须等到主服务器上负载低的时候才进行，而且还可以对正在运行的实例进行快照。在不中断正常操作的情况下，可以只将备份复制到沙箱中，并开始测试。

当然，仅最近的备份才是有用的。在有大量写操作的应用程序中，一个月前的备份是没有多大帮助的。所以定期全备、频繁的增量备份是有必要的。开启二进制日志也是非常好的，这样在故障发生时，你就拥有所有的更改操作记录。

Baron Schwartz 等人撰写的 *High Performance MySQL* 和 Charles Bell 等人撰写的 *MySQL High Availability*（这两本书都由 O'Reilly 出版）介绍了如何进行备份，包括何时、为什么、如何使用可用的工具来进行备份的描述。以下小节仅仅给出计划备份时一些需要牢记的注意事项。

7.1.1 计划备份

在计划备份时，必须思考如何完成备份，这样在任何时间点你都能进行完整还原。例如，你不应仅仅依赖复制的从服务器作为你的备份。从服务器可能远远落后于它的主服务器，并因此有过期的数据。从服务器可能与包含了与主服务器不一样的数据，特别是在基于语句的复制。第 5 章已经介绍了发生这种情况的原因。因此，不要依赖于将从服务器作为你的唯一备份解决方案。

我喜欢每周完整备份，每天都增量备份以及保存所有的二进制日志。当然，你可以根据实际写入负载来调整计划。我只是不建议在两次备份之间间隔很长时间，因为在这种情况下，你就可能会因为硬件故障而面临丢失大量数据的风险。所以规划一定要合理。

每次还原数据时，首先从最新的完整备份恢复，然后在完整备份后，如果增量备份存在，就按顺序应用它们，最后从二进制日志中加载任何剩余的改变记录。

7.1.2 备份类型

本节介绍完整备份与增量备份。备份的第三要素是保存二进制日志，简单概括为：最近一次备份后，不删除新创建的日志，如果备份之间的时间间隔很大，就需要定期把它们复制到安全的地方。

备份可以根据几个不同维度进行分组。

依据格式

逻辑备份

保存转储的结构与数据。尽管这种备份方式很慢，但是它非常有用，因为它的文件可以由人阅读和手动编辑。

物理备份

保存二进制文件，这通常很快。如果因为表某种程度的损坏而不能复现问题，这种备份方法就非常重要。在这种情况下，使用单个表的二进制备份并把它移到测试服务器中是非常意义的。

与 MySQL 服务器交互

在线备份

在 MySQL 服务器运行时备份。

离线备份

在 MySQL 服务器停止时备份。

与 MySQL 服务器对象交互

冷备份

在 MySQL 服务器不允许操作时进行备份。服务器或者是停止修改他自己的文件了，或者是被阻塞而不能修改自己的文件。这种备份方式的优点是它非常快。

温备份

在 MySQL 服务器运行期间备份，在备份时仅仅禁止了对象上的少数操作。如果并行线程使用数据库对象，并不总是可能有一个一致的备份，因此，所有的备份方法会使用某种类型的智能来保证正在备份的对象是安全的。这种方法仅仅涉及当前备份对象的写锁，而允许其他连接修改其他对象。在这种备份方式期间，读操作访问通常也是允许的。

热备分

在 MySQL 服务器运行期间备份，备份对象上的所有操作都是允许的。在在线备份中，这是最快的方法。

依据内容

完整备份

备份所有对象。

增量备份

只备份特定时间后的更改记录，通常是以前备份的时间。

部分备份

仅仅复制特定对象，例如，一个数据库中的几个表。

7.1.3 工具

本节不会介绍所有可用的 MySQL 备份工具，恰恰相反，会介绍用于说明上一节所示备份类型的几个工具。无论它是否在列表中，你都可以根据自己的喜好自由使用。

mysqldump

来自于 MySQL 发布版，它能进行逻辑备份。

MySQL Enterprise Backup（MEB）

为企业客户提供的一个单独产品。它能为 InnoDB 表创建热备份，为其他存储引擎创建温备份，以及冷备份。

Percona XtraBackup

功能与 MEB 相似的一个开源产品。它能为 Xtradb 与 InnoDB 表创建热备份，能为

其他引擎创建温备份。

cp

复制文件的基本 UNIX shell 命令。能用它创建冷的离线备份。

文件系统快照工具（如 LVM）

创建文件系统快照。它应该用在当 MySQL 服务器停止或者 MySQL 服务器被阻止写自己文件的时候。

表 7-1 给出了不同的备份类型与支持这些备份类型的工具之间的对应关系。

表 7-1　　本节所讨论的各种工具支持的不同备份类型

备份类型/工具	mysqldump	MEB	XtraBackup	cp	LVM
逻辑	是	否	否	否	否
物理	否	是	是	是	是
在线	是	是	是	否	否
离线	否	是	否	是	是
冷	否	是	否	是	是
温	是	是	是	否	否
热	否	是	是	否	否
全部	是	是	是	是	是
增加	否	是	是	否	否
部分	是	是	是	是	否

使用此表，可以确定哪个工具最适合你的需求。如果你决定使用不包含增量备份的方案，请通过二进制日志完成增量备份。关于备份的详细信息与最佳实践，可以参考 7.1 节。

7.2　收集需要的信息

信息是成功排查故障的关键。除了在你自己的故障排查中使用信息之外，当你打开一个支持案例时信息也是很关键的。所以不要忽视收集问题报告的来源与工具，如错误日志。

也就是说，你不能一直记录所有信息。日志记录会增加服务器的负担。所以你需要在一直保存需要的信息与忽略信息直到真正问题发生之间找到一个平衡。

我建议你一直打开并不耗费大量资源的工具，例如，错误日志文件与操作系统日志。

如果只需要少量资源，例如没有查询优化器的 MEM，记录功能也应该长期运行。可以试图通过关闭一些工具和放弃在关键问题期间能提供帮助的工具，来少量减轻服务器的负载，这并不能节约资源。

关于报告工具，他可以明显地降低性能，虽然可以使用它们，但是将其关闭，直到问题被触发。这些工具可以被当作是应用程序中的一些选项来实现。

所有这些意味着什么

如果你不理解，就算是千兆字节（GB）信息也是无用的。所以阅读一些错误消息、从 MEM 中得到一些建议，等等。如果你不理解它们，可以参考 MySQL 参考手册、书籍、博客、论坛或其他的信息来源。

搜索引擎是一位好朋友。只要在搜索框中输入错误消息，通常就会出现大量包含其他用户解决相同问题信息的链接。在大多数情况下，你并不是孤单的（有不少遇到相同问题的用户）。

当然，你可以随时在公共论坛或者 IRC 上提问题。最后，你可以购买服务支持。

7.3 测试

当你提出一个关于正在发生什么和如何解决问题的假设之后，请测试这个假设并考虑结果。

在我的工作中，我遇到一些用户，他们非常害怕测试。主要有两个原因导致他们抵触它：过度自信（用户认为正确的猜测是不需要测试的）和缺乏自信（一些用户害怕破坏一些东西）。

但即使非常有经验的用户都会犯错误，所以仅仅依赖猜测是有风险的。请务必检测你所做任何更改的结果。早发现就能早解决。

不要害羞。只需要考虑这一点：如果你需要解决一些问题，那就说明你已经遇到问题了。如果某个测试或其他测试使得问题变得更糟，那么测试至少使你避免了一次重大更改，这种重大更改需要花很长时间才能修复。而在测试过程中，产生的错误可以在几秒钟内进行回滚。

此外，发现一个猜测是错误的，这是非常重要的，因为你可以将搜索范围缩小。保留较少选项，这样在下一次测试中，就会有更多的机会来证明是正确的。

如果你不想冒任何风险来损坏生产环境，那么请在沙箱中进行测试。特别是当测试需要更改数据时，我建议这样，但当你只需要修改一些配置选项时，它同样也有用。只需要创建一个可重复的测试案例并在沙箱中运行它。

不要偷懒！当你正在找一个最好且最短的方法来解决问题时，懒惰可能是件好事，但测试时，它可能闹出一些不好的笑话。

由于太懒惰而不愿意从原始表将数据复制到沙箱中，由于不足的测试数据而无法重现问题，可能会导致许多无效尝试来重现你的问题。更糟的是，你可能最终实施了错误的解决方案。因此，如果问题对于小测试数据集无法重现，那么就应将全部表复制到沙箱中，然后再在这个基础上进行实验。

7.4 预防

故障排查的最好方法是提前预防问题的发生。消除所有可能的问题的唯一方法是完全阻止访问 MySQL 服务器，这意味着它不会做任何有意义的工作。所以我们往往只能采用折中的办法。

7.4.1 权限

预防的一个重要方面是权限。尽管本书中的许多测试用例使用 root 用户，但这仅仅是在不造成任何伤害的测试环境中才被接受。当运行在生产服务器中时，每个用户都应该有尽可能少的权限。理想情况下，用户应该只拥有他将使用的这些对象的权限。例如，如果一个 Web 应用程序仅仅从几个数据库中查询数据，那么就不要给 Web 应用程序的用户账号写入和读取 MySQL 数据库的权限。如果你需要维护数据库，请为它创建一个单独的用户。

这种预防措施将使你免受不良干预，甚至可以减少 SQL 注入攻击。成功的攻击造成的损害也将是有限的。

7.4.2 环境

另一个重要方面是服务器运行的环境：MySQL 服务器选项和它运行的外部环境。
当调整选项时，分析其造成的影响时。仅仅从你确信会造成影响的选项开始，然后逐一添加它们并分析它们如何改变事情。在这种场景下，如果出现错误，你可以在几秒钟内还原可以接受工作环境。
当规划 MySQL 安装时，分析环境如何影响它。不要期望在硬件和操作系统因其他进程过载或遭受磁盘损坏时，MySQL 服务器还能顺畅、快速地运行。相应地规划并在问题出现第一迹象时检查硬件。

在规划复制或其他复杂环境时，尝试提前诊断该设置如何影响数据库活动。一个简单的示例是在分别使用行模式或语句模式的二进制日志记录时查询行为的差异。始终做出分析并制订相应的计划。

7.5 三思而后行！

我想以强调推理重要性作为本书的结束。找出问题根源的能力不仅来源于实践，同时还来源于分析问题的技巧。

当你遇到问题时，我建议你在选择你认为最适合这特定问题的故障排查技术前，彻底地思考一下。

如果一开始就能选择一个好的行动计划，就能节省大量时间。而且，一个全面的诊断不仅能预防是你眼下意识到的问题，还能预防日后出现因同样原因造成的其他问题。

附录

信息资源

在本书中，我已经指出了一些信息资源来帮助大家排错。这里给出了一个信息资源的简短列表，并对其按照用途来分组。一如既往，我非常喜欢这些资源，并且经常使用。

包含有用信息的资源

MySQL 官方参考手册（The official MySQL Reference Manual）

这是搜索信息的第一去处，它记录了 MySQL 的特性是如何工作的。

搜索引擎

如果在 MySQL 参考手册中找不到足够的细节，可以尝试一下搜索引擎。在大多数情况下，可以复制和粘贴一个错误消息到搜索引擎中，然后得到关于该问题的大量信息。世界这么小，我们很难捕获到一个别人没有发现过的问题。

Bug 和知识数据库

社区 bug 数据库（The Community Bug Database）

如果你不理解 MySQL 的行为，但是相信它的行为是错误的，则可以搜索 bug 数据库。你很有可能会找到同一问题的一个报告。如果你使用的 MySQL 版本较老，甚至可以发现这个问题是否已经被修复。

Oracle 客户 bug 数据库（Oracle Customer's Bug Database）Oracle 使用其内部的 bug 数据库来跟踪内部报告的 bug 或者是客户报告的 bug。如果你是 Oracle 客户，你可以访问

该数据库，找到社区 bug 数据库还没有报告的 bug。该数据库大多含有真正的 bug，你很少会遇到标识为“非 bug（not a bug）”的报告。这是因为在将这些 bug 包含到该数据库之前，它们会经过仔细的审查。

Oracle 知识管理数据库（Oracle's Knowledge Management database）

Oracle 维护着一个定期更新的知识数据库，以供其客户访问。它包含了产品与问题的描述，而且详细程序远甚于 MySQL 参考手册。其中的大多数文章都是从客户的使用案例中创建的，因此你可能会从中找到你遇到的问题。有些文章是在某一特性被发布之时的同一天发表的，因此该数据库是一个用来查询真实消息的好资源。

专家知识在线（Expert Knowledge Online）

如果你正在搜索与某个特性相关的详细文章，可以试试下述资源。

MySQL Forge

MySQL Forge 项目旨在为 MySQL 开发者提供一个用于信息共享的 wiki 库。这里有 MySQL 内核、插件开发、公共工作记录表（worklogs）和社区项目相关的信息。你在这里可以找到描述许多特性内部工作机制是 wiki、MySQL 内核文档、社区插件和公共工作表（开发 MySQL 后续版本的计划）。

MySQL Planet

这是一个关于 MySQL 的博客聚合器，其语言为英语。由活跃社区成员写作的博客都汇聚到这里。在这里可以找到详细描述某些特性的博文，也可以找到深入剖析内核的博文。这是一个很棒的资源，在这一个地方就能找到与 MySQL 相关的大量信息。MySQL Planet 还有特定语言的版本，你可以查看一下是否有自己母语的 MySQL Planet。

我想在这里强调两个博客，这两个博客都可以在 MySQL Planet 站点找到。

MySQL Performance Blog

Percona 的工程师在该博客上发表博文，它包含了大量与性能调优相关的信息。当你遇到性能相关的问题时，它应该是你最先查看的资源。

InnoDB Team Blog

从名字中就可以猜出来，该博客由 InnoDB 团队的成员来写作。你在这里可以找到 InnoDB 开发相关的详情、新特性和内核的使用技巧。对于使用 InnoDB 存储引擎的人来说，这是一个相当不错的资源。

可以求助的地点

论坛、社区、用户组

由于论坛太多，我在这里不提供某些论坛的链接。你以从 MySQL forum 起步，也开始使用最喜欢的本地论坛（我就是这样做的）。只需要在你所在的国家找到一个好的论坛就好。

IRC，尤其是#mysql at Freenode

许多 MySQL 专家都在这里，只需登录进去然后提问问题即可。这里还有大量特定的通道，比如#mysql-dev，你可以在这里咨询插件开发相关或者是扩展 MySQL 相关的问题；还有#mysql-ndl，很多 NDB 专家会在该通道出没。对于排错问题，最合适的通道是#mysql。

图书

还有大量与 MySQL 相关的图书。我们先从 O'Reilly 图书开始介绍（http://shop.oreilly.com/category/browse-subjects/data/relational-databases.do），这是因为 O'Reilly 总是与业内顶尖的专家进行合作。在这个打开的链接中，你可以发现 MySQL 源代码的作者，以及一些顶级的 MySQL 专家。

我还要推荐 Charles A. Bell 写作的 *Expert MySQL*（Apress 出版），该书介绍了如何调试和修改 MySQL 代码的大量信息。我在本书的 1.7 节和 6.4.2 节的“Core file”小节提到了这些主题，如果你想知道更多细节，请咨询 Bell 博士。

作者介绍

Sveta Smirnova 是 Oracle 公司 MySQL 支持团队 Bug 验证小组的首席技术支持工程师。

封面图片介绍

本书封面上的动物为马来西亚臭獾，也称为爪哇臭獾、巽他臭獾或印尼臭獾。臭獾属还包括另外一个物种——巴拉望臭獾。长期以来，臭獾一直被认为是獾家族的一部分，但是最近的 DNA 研究表明，它们与臭鼬的关系更为密切。

臭獾有棕黑色的皮毛，而且有白色或黄色的嘴，背部后面有一条类似于臭鼬的长条纹。它们的长嘴上面有像猪一样的鼻子，在前脚上长着细长而弯曲的爪子。它们一般有 12～20 英寸长（包括一条很短的尾巴在内），体重可达 8 磅。

臭獾主要分布在印尼、马来西亚以及菲律宾，它们生活在森林里以及附近的开阔地带，以及多山岛屿的高海拔地带。作为夜行动物，它们生活在地下的洞穴中，它们能自己挖洞，也会与豪猪共用洞穴。它们以蛋类、昆虫、植物和腐肉为食，它们的爪子和嘴用来挖掘蚯蚓。它们通常一窝幼崽有 2～3 只，但是人们对它们的生活习性以及繁殖情况知之甚少。

Lydekker 在他的 *Royal Natural History*（《皇家自然史》）中，将这种物种从后部腺体发出来的“臭气”称为具有极端臭味的喷雾。臭獾的这种分泌物用来防御肉食动物，其中包括爪哇鹰雕、野猫和老虎。

Databases

MySQL排错指南

还在与bug、性能问题、程序崩溃、数据损坏以及令人费解的输出等问题死磕？如果你是一名数据库程序员或DBA，你将每天都要与这些问题打交道。它们的应对之策是知道如何进行迅速的恢复。本书采用独特的视角，通过大量案例来演示如何处理MySQL中遇到的棘手问题。

本书由Oracle的首席技术支持工程师编写，它提供了用于解决各种问题（从简单到复杂）的相关背景、工具和步骤。无论是你插入的数据无法在查询中出现，还是因为服务器故障而导致整个数据库被损坏，只要本书在手，你都可以轻松应对这些问题。

- 即使问题的解决方案很简单，也要理解问题产生的根源
- 当应用程序在多个线程上运行时，处理所发生的问题
- 调试和修复由配置选项引发的问题
- 探究操作系统调优如何影响服务器
- 使用特定的排错技术来重现问题
- 参考其他排错技术和工具，其中包括第三方解决方案
- 学习可实现安全、高效排错以及预防问题的最佳实践

Sveta Smirnova是Oracle公司MySQL部门bug分析支持团队的首席技术支持工程师，每天的工作是处理棘手的支持问题和MySQL软件bug。Sveta是开源社区的一名积极参与者。

“迄今为止出版的独一无二的MySQL排错图书。在我看来，Sveta的这本著作在该领域具有里程碑意义。我之前从来没有遇到过这样的图书，这是一本彻头彻尾的工具类图书。相信本书很快就会成为所有MySQL DBA和支持人员的案头必备。”

——Charles Bell博士

O'REILLY®
oreilly.com.cn

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/数据库/MySQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39728-7

ISBN 978-7-115-39728-7

定价：49.00 元

[General Information]
书名=MySQL排错指南
作者=（美）斯米尔诺娃著
页数=217
SS号=13803990
DX号=
出版日期=2015.08
出版社=北京人民邮电出版社