华章科技

畅销书全新升级，第1版广获好评；资深MySQL专家撰写，全球知名MySQL数据库服务提供商Percona公司CTO作序推荐，国内多位数据库专家联袂推荐

基于MySQL 5.6，结合源代码，从存储引擎内核角度对InnoDB的整体架构、核心实现和工作机制进行深入剖析

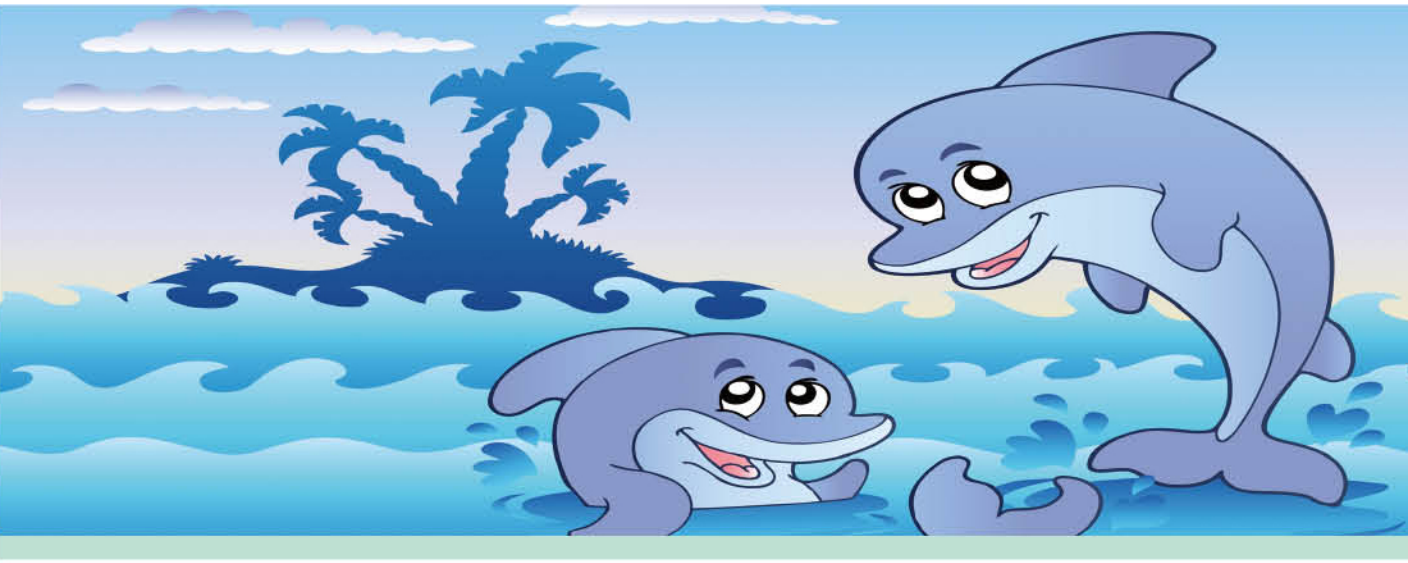
数据库技术丛书

Inside MySQL: InnoDB Storage Engine, Second Edition

MySQL技术内幕

InnoDB存储引擎

第2版



姜承尧◎著

机械工业出版社
China Machine Press

数据库技术丛书

MySQL技术内幕：InnoDB存储引擎

第2版

姜承尧 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

MySQL 技术内幕：InnoDB 存储引擎 / 姜承尧著 .—2 版 . —北京：机械工业出版社，2013.6
（数据库技术丛书）

ISBN 978-7-111-42206-8

I. M…　II. 姜…　III. 关系数据库系统　IV. TP311.138

中国版本图书馆 CIP 数据核字（2013）第 079001 号

本书由国内资深 MySQL 专家亲自执笔，国内外多位数据库专家联袂推荐。作为国内唯一一本关于 InnoDB 的专著，本书的第 1 版广受好评，第 2 版不仅针对最新的 MySQL 5.6 对相关内容进行了全面的补充，还根据广大读者的反馈意见对第 1 版中存在的不足进行了完善，全书大约重写了 50%的内容。本书从源代码的角度深度解析了 InnoDB 的体系结构、实现原理、工作机制，并给出了大量最佳实践，能帮助你系统而深入地掌握 InnoDB，更重要的是，它能为你设计管理高性能、高可用的数据库系统提供绝佳的指导。

全书一共 10 章，首先宏观地介绍了 MySQL 的体系结构和各种常见的存储引擎以及它们之间的比较；接着以 InnoDB 的内部实现为切入点，逐一详细讲解了 InnoDB 存储引擎内部的各个功能模块的实现原理，包括 InnoDB 存储引擎的体系结构、内存中的数据结构、基于 InnoDB 存储引擎的表和页的物理存储、索引与算法、文件、锁、事务、备份与恢复，以及 InnoDB 的性能调优等重要的知识；最后对 InnoDB 存储引擎源代码的编译和调试做了介绍，对大家阅读和理解 InnoDB 的源代码有重要的指导意义。

本书适合所有希望构建和管理高性能、高可用性的 MySQL 数据库系统的开发者和 DBA 阅读。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：姜　影
　　印刷
2013 年 6 月第 1 版第 1 次印刷
186mm×240mm • 27.25 印张
标准书号：ISBN 978-7-111-42206-8
定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

推 荐 序

It's fair to say that MySQL is the most popular open source database. It has a very large installed base and number of users. Let's see what are the reasons MySQL is so popular, where it stands currently, and maybe touch on some of its future (although predicting the future is rarely successful).

Looking at the customer area of MySQL, which includes Facebook, Flickr, Adobe (in Creative Suite 3), Drupal, Digg, LinkedIn, Wikipedia, eBay, YouTube, Google AdSense (source http://mysql.com/customers/ and public resources), it's obvious that MySQL is everywhere. When you log in to your popular forum (powered by Bulleting) or blog (powered by WordPress), most likely it has MySQL as its backend database. Traditionally, two MySQL's characteristics, simplicity of use and performance, were what allowed it to gain such popularity. In addition to that, availability on a very wide range of platforms (including Windows) and built-in replication, which provides an easy scale-out solution for read-only clients, gave more user attractions and production deployments. There is simple evidence of MySQL's simplicity: In 15 minutes or less, you really can get installed, have a working database, and start running queries and store data. From its early stages MySQL had a good interface to most popular languages for Web development - PHP and Perl, and also Java and ODBC connectors.

There are two best known storage engines in MySQL: MyISAM and InnoDB (I don't cover NDB cluster here; it's a totally different story). MyISAM comes as the default storage engine and historically it is the oldest, but InnoDB is ACID compliant and provides transactions, row-level locking, MVCC, automatic recovery and data corruption detection. This makes it the storage engine you want to choose for your application. Also, there is the third-party transaction storage engine PBXT, with characteristics similar to InnoDB, which is included in the MariaDB distribution.

MySQL's simplicity has its own drawback. Just as it is very easy to start working with it, it is very easy to start getting into trouble with it. As soon as your website or forum gets popular, you may figure out that the database is a bottleneck, and that you need special skills and tools to fix it.

The author of this book is a MySQL expert, especially in InnoDB storage engineB. Hence, I highly recommend this book to new users of InnoDB as well as uers who already have well-tuned InnoDB-based applications but need to get internal out of them.

Vadim Tkachenko

全球知名 MySQL 数据库服务提供商 Percona 公司 CTO

知名 MySQL 数据库博客 MySQLPerformanceBlog.com 作者

《高性能 MySQL（第 2 版）》作者之一

前　言

为什么要写这本书

过去这些年我一直在和各种不同的数据库打交道，见证了 MySQL 从一个小型的关系型数据库发展为各大企业的核心数据库系统的过程，并且参与了一些大大小小的项目的开发工作，成功地帮助开发人员构建了可靠的、健壮的应用程序。在这个过程中积累了一些经验，正是这些不断累积的经验赋予了我灵感，于是有了这本书。这本书实际上反映了这些年来我做了哪些事情，其中汇集了很多同行每天可能都会遇到的一些问题，并给出了解决方案。

MySQL 数据库独有的插件式存储引擎架构使其和其他任何数据库都不同。不同的存储引擎有着完全不同的功能，而 InnoDB 存储引擎的存在使得 MySQL 跃入了企业级数据库领域。本书完整地讲解了 InnoDB 存储引擎中最重要的一些内容，即 InnoDB 的体系结构和工作原理，并结合 InnoDB 的源代码讲解了它的内部实现机制。

本书不仅讲述了 InnoDB 存储引擎的诸多功能和特性，还阐述了如何正确地使用这些功能和特性，更重要的是，还尝试了教我们如何 Think Different。Think Different 是 20 世纪 90 年代苹果公司在其旷日持久的宣传活动中提出的一个口号，借此来重振公司的品牌，更重要的是，这个口号改变了人们对技术在日常生活中的作用的看法。需要注意的是，苹果的口号不是 Think Differently，是 Think Different，Different 在这里做名词，意味该思考些什么。

很多 DBA 和开发人员都相信某些“神话”，然而这些“神话”往往都是错误的。无论计算机技术发展的速度变得多快，数据库的使用变得多么简单，任何时候 Why 都比 What 重要。只有真正理解了内部实现原理、体系结构，才能更好地去使用。这正是人类正确思考问题的原则。因此，对于当前出现的技术，尽管学习其应用很重要，但更重要的是，应当正确地理解和使用这些技术。

关于本书，我的头脑里有很多个目标，但最重要的是想告诉大家如下几个简单的观点：

- 不要相信任何的“神话”，学会自己思考；
- 不要墨守成规，大部分人都知道的事情可能是错误的；
- 不要相信网上的传言，去测试，根据自己的实践做出决定；
- 花时间充分地思考，敢于提出质疑。

当前有关 MySQL 的书籍大部分都集中在教读者如何使用 MySQL，例如 SQL 语句的使用、复制的搭建的、数据的切分等。没错，这对快速掌握和使用 MySQL 数据库非常有好处，但是真正的数据库工作者需要了解的不仅仅是应用，更多的是内部的具体实现。

MySQL 数据库独有的插件式存储引擎使得想要在一本书内完整地讲解各个存储引擎变得十分困难，有的书可能偏重对 MyISAM 的介绍，有的可能偏重对 InnoDB 存储引擎的介绍。对于初级的 DBA 来说，这可能会使他们的理解变得更困难。对于大多数 MySQL DBA 和开发人员来说，他们往往更希望了解作为 MySQL 企业级数据库应用的第一存储引擎的 InnoDB，我想在本书中，他们完全可以找到他们希望了解的内容。

再强调一遍，任何时候 Why 都比 What 重要，本书从源代码的角度对 InnoDB 的存储引擎的整个体系架构的各个组成部分进行了系统的分析和讲解，剖析了 InnoDB 存储引擎的核心实现和工作机制，相信这在其他书中是很难找到的。

第 1 版与第 2 版的区别

本书是第 2 版，在写作中吸收了读者对上一版内容的许多意见和建议，同时对于最新 MySQL 5.6 中许多关于 InnoDB 存储引擎的部分进行了详细的解析与介绍。希望通过这些改进，给读者一个从应用到设计再到实现的完整理解，弥补上一版中深度有余，内容层次不够丰富、分析手法单一等诸多不足。

较第 1 版而言，第 2 版的改动非常大，基本上重写了 50% 的内容。其主要体现在以下几个方面，希望读者能够在阅读中体会到。

- 本书增加了对最新 MySQL 5.6 中的 InnoDB 存储引擎特性的介绍。MySQL 5.6 版本是有史以来最大的一次更新，InnoDB 存储引擎更是添加了许多功能，如多线程清理线程、全文索引、在线索引添加、独立回滚段、非递归死锁检测、新的刷新算法、新的元数据表等。读者通过本书可以知道如何使用这些特性、新特性存在的局限性，并明白新功能与老版本 InnoDB 存储引擎之间实现的区别，从而在实际应用中充分利用这些特性。
- 根据读者的要求对于 InnoDB 存储引擎的 redo 日志和 undo 日志进行了详细的分析。读者应该能更好地理解 InnoDB 存储引擎事务的实现。在 undo 日志分析中，通过 InnoSQL 自带的元数据表，用户终于可对 undo 日志进行统计和分析，极大提高了 DBA 对于 InnoDB 存储引擎内部的认知。

- 对第 6 章进行大幅度的重写，读者可以更好地理解 InnoDB 存储引擎特有的 next-key locking 算法，并且通过分析锁的实现来了解死锁可能产生的情况，以及 InnoDB 存储引擎内部是如何来避免死锁问题的产生的。
- 根据读者的反馈，对 InnoDB 存储引擎的 insert buffer 模块实现进行了更为详细的介绍，读者可以了解其使用方法以及其内部的实现原理。此外还增加了对 insert buffer 的升级版本功能——change buffer 的介绍。

读者对象

本书不是一本面向应用的数据库类书籍，也不是一本参考手册，更不会教你如何在 MySQL 中使用 SQL 语句。**本书面向那些使用 MySQL InnoDB 存储引擎作为数据库后端开发应用程序的开发者和有一定经验的 MySQL DBA**。书中的大部分例子都是用 SQL 语句来展示关键特性的，如果想通过本书来了解如何启动 MySQL、如何配置 Replication 环境，可能并不能如愿。不过，在本书中，你将知道 InnoDB 存储引擎是如何工作的，它的关键特性的功能和作用是什么，以及如何正确配置和使用这些特性。

如果你想更好地使用 InnoDB 存储引擎，如果你想让你的数据库应用获得更好的性能，就请阅读本书。从某种程度上讲，技术经理或总监也要非常了解数据库，要知道数据库对于企业的重要性。如果技术经理或总监想安排员工参加 MySQL 数据库技术方面的培训，完全可以利用本书来“充电”，相信你一定不会失望的。

要想更好地学习本书的内容，要求具备以下条件：

- 掌握 SQL。
- 掌握基本的 MySQL 操作。
- 接触过一些高级语言，如 C、C++、Python 或 Java。
- 对一些基本算法有所了解，因为本书会分析 InnoDB 存储引擎的部分源代码，如果你能看懂这些算法，这会对你的理解非常有帮助。

如何阅读本书

本书一共有 10 章，每一章都像一本“迷你书”，可以单独成册，也就说你完全可以从书中任何一章开始阅读。例如，要了解第 10 章中的 InnoDB 源代码编译和调试的知识，就不必先去阅读第 3 章有关文件的知识。当然，如果你不太确定自己是否已经对本书所涉及的内容

完全掌握了，建议你系统性地阅读本书。

本书不是一本入门书籍，不会一步步引导你去如何操作。倘若你尚不了解 InnoDB 存储引擎，本书对你来说可能就显得沉重一些，建议你先查阅官方的 API 文档，大致掌握 InnoDB 的基础知识，然后再来学习本书，相信你会领略到不同的风景。

为了便于大家阅读，本书在提供源代码下载（下载地址：www.hzbook.com）的同时也将源代码附在了书中，因此占去了一些篇幅，还请大家理解。

勘误和支持

由于作者对 InnoDB 存储引擎的认知水平有限，再加上写作时可能存在疏漏，书中还存在许多需要改进的地方。在此，欢迎读者朋友们指出书中存在的问题，并提出指导性意见，不甚感谢。如果大家有任何与本书相关的内容需要与我探讨，请发邮件到 jiangchengyao@gmail.com，或者通过新浪微博 @insidemysql 与我联系，我会及时给予回复。最后，衷心地希望本书能给大家带来帮助，并祝大家阅读愉快！

致谢

在编写本书的过程中，我得到了很多朋友的热心帮助。首先要感谢 Pecona 公司的 CEO Peter Zaitsev 和 CTO Vadim Tkachenko，通过和他们的不断交流，使我对 InnoDB 存储引擎有了更进一步的了解，同时知道了怎样才能正确地将 InnoDB 存储引擎的补丁应用到生产环境。

其次，要感谢网易公司的各位同事们，能在才华横溢、充满创意的团队中工作我感到非常荣幸和兴奋。也因为在这个开放的工作环境中，我可以不断进行研究和创新。

此外，我还要感谢我的母亲，写本书不是一件容易的事，特别是这本书还想传达一些思想，在这个过程中我遇到了很多的困难，感谢她在这个过程中给予我的支持和鼓励。

最后，一份特别的感谢要送给本书的策划编辑杨福川和姜影，他们使得本书变得生动和更具有灵魂。此外还要感谢出版社的其他默默工作的同事们。

姜承尧

目　录

第 1 章　MySQL 体系结构和存储引擎

MySQL 被设计为一个可移植的数据库，几乎在当前所有系统上都能运行，如 Linux，Solaris、FreeBSD、Mac 和 Windows。尽管各平台在底层（如线程）实现方面都各有不同，但是 MySQL 基本上能保证在各平台上的物理体系结构的一致性。因此，用户应该能很好地理解 MySQL 数据库在所有这些平台上是如何运作的。

1.1　定义数据库和实例

在数据库领域中有两个词很容易混淆，这就是“数据库”（database）和“实例”（instance）。作为常见的数据库术语，这两个词的定义如下。

- **数据库**：物理操作系统文件或其他形式文件类型的集合。在 MySQL 数据库中，数据库文件可以是 frm、MYD、MYI、ibd 结尾的文件。当使用 NDB 引擎时，数据库的文件可能不是操作系统上的文件，而是存放于内存之中的文件，但是定义仍然不变。
- **实例**：MySQL 数据库由后台线程以及一个共享内存区组成。共享内存可以被运行的后台线程所共享。需要牢记的是，数据库实例才是真正用于操作数据库文件的。

这两个词有时可以互换使用，不过两者的概念完全不同。在 MySQL 数据库中，实例与数据库的关通常系是一一对应的，即一个实例对应一个数据库，一个数据库对应一个实例。但是，在集群情况下可能存在一个数据库被多个数据实例使用的情况。

MySQL 被设计为一个单进程多线程架构的数据库，这点与 SQL Server 比较类似，但与 Oracle 多进程的架构有所不同（Oracle 的 Windows 版本也是单进程多线程架构的）。这也就是说，**MySQL 数据库实例在系统上的表现就是一个进程**。

在 Linux 操作系统中通过以下命令启动 MySQL 数据库实例，并通过命令 ps 观察 MySQL 数据库启动后的进程情况：

```
[root@xen-server bin]# ./mysqld_safe&

[root@xen-server bin]# ps -ef | grep mysqld
```

```
root     3441  3258  0 10:23 pts/3    00:00:00 /bin/sh ./mysqld_safe
mysql 3578 3441 0 10:23 pts/3 00:00:00
/usr/local/mysql/libexec/mysqld --basedir=/usr/local/mysql
--datadir=/usr/local/mysql/var --user=mysql
--log-error=/usr/local/mysql/var/xen-server.err
--pid-file=/usr/local/mysql/var/xen-server.pid
--socket=/tmp/mysql.sock --port=3306
root     3616  3258  0 10:27 pts/3    00:00:00 grep mysqld
```

注意进程号为3578的进程，该进程就是MySQL实例。在上述例子中使用了mysqld_safe命令来启动数据库，当然启动MySQL实例的方法还有很多，在各种平台下的方式可能又会有所不同。在这里不一一赘述。

当启动实例时，MySQL数据库会去读取配置文件，根据配置文件的参数来启动数据库实例。这与Oracle的参数文件（spfile）相似，不同的是，Oracle中如果没有参数文件，在启动实例时会提示找不到该参数文件，数据库启动失败。而在MySQL数据库中，可以没有配置文件，在这种情况下，MySQL会按照编译时的默认参数设置启动实例。用以下命令可以查看当MySQL数据库实例启动时，会在哪些位置查找配置文件。

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

可以看到，MySQL数据库是按/etc/my.cnf→/etc/mysql/my.cnf→/usr/local/mysql/etc/my.cnf→～/.my.cnf的顺序读取配置文件的。可能有读者会问：“如果几个配置文件中都有同一个参数，MySQL数据库以哪个配置文件为准？”答案很简单，MySQL数据库会以读取到的最后一个配置文件中的参数为准。在Linux环境下，配置文件一般放在/etc/my.cnf下。在Windows平台下，配置文件的后缀名可能是.cnf，也可能是.ini。例如在Windows操作系统下运行mysql--help，可以找到如下类似内容：

```
Default options are read from the following files in the given order:
C:\Windows\my.ini C:\Windows\my.cnf C:\my.ini C:\my.cnf C:\Program Files\
MySQL\M
\MySQL Server 5.1\my.cnf
```

配置文件中有一个参数datadir，该参数指定了数据库所在的路径。在Linux操作系统下默认datadir为/usr/local/mysql/data，用户可以修改该参数，当然也可以使用该路径，不过该路径只是一个链接，具体如下：

```
mysql>SHOW VARIABLES LIKE 'datadir'\G;
*************************** 1. row ***************************
Variable_name: datadir
       Value: /usr/local/mysql/data/
1 row in set (0.00 sec)1 row in set (0.00 sec)

mysql>system ls-lh /usr/local/mysql/data
total 32K
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 bin
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 docs
drwxr-xr-x  3 root mysql 4.0K Aug  6 16:04 include
drwxr-xr-x  3 root mysql 4.0K Aug  6 16:04 lib
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 libexec
drwxr-xr-x 10 root mysql 4.0K Aug  6 16:23 mysql-test
drwxr-xr-x  5 root mysql 4.0K Aug  6 16:04 share
drwxr-xr-x  5 root mysql 4.0K Aug  6 16:23 sql-bench
lrwxrwxrwx  1 root mysql   16 Aug  6 16:05 data -> /opt/mysql_data/
```

从上面可以看到，其实 data 目录是一个链接，该链接指向了 /opt/mysql_data 目录。当然，用户必须保证 /opt/mysql_data 的用户和权限，使得只有 mysql 用户和组可以访问（通常 MySQL 数据库的权限为 mysql：mysql）。

1.2 MySQL 体系结构

由于工作的缘故，笔者的大部分时间需要与开发人员进行数据库方面的沟通，并对他们进行培训。不论他们是 DBA，还是开发人员，似乎都对 MySQL 的体系结构了解得不够透彻。很多人喜欢把 MySQL 与他们以前使用的 SQL Server、Oracle、DB2 作比较。因此笔者常常会听到这样的疑问：

- ❑ 为什么 MySQL 不支持全文索引？
- ❑ MySQL 速度快是因为它不支持事务吗？
- ❑ 数据量大于 1000 万时 MySQL 的性能会急剧下降吗？

……

对于 MySQL 数据库的疑问有很多很多，在解释这些问题之前，笔者认为不管对于使用哪种数据库的开发人员，了解数据库的体系结构都是最为重要的内容。

在给出体系结构图之前，用户应该理解了前一节提出的两个概念：数据库和数据库实例。很多人会把这两个概念混淆，即 MySQL 是数据库，MySQL 也是数据库实例。

这样来理解 Oracle 和 Microsoft SQL Server 数据库可能是正确的，但是这会给以后理解 MySQL 体系结构中的存储引擎带来问题。从概念上来说，数据库是文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合；数据库实例是程序，是位于用户与操作系统之间的一层数据管理软件，用户对数据库数据的任何操作，包括数据库定义、数据查询、数据维护、数据库运行控制等都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库打交道。

如果这样讲解后读者还是不明白，那这里再换一种更为直白的方式来解释：数据库是由一个个文件组成（一般来说都是二进制的文件）的，要对这些文件执行诸如 SELECT、INSERT、UPDATE 和 DELETE 之类的数据库操作是不能通过简单的操作文件来更改数据库的内容，需要通过数据库实例来完成对数据库的操作。所以，用户把 Oracle、SQL Server、MySQL 简单地理解成数据库可能是有失偏颇的，虽然在实际使用中并不会这么强调两者之间的区别。

好了，在给出上述这些复杂枯燥的定义后，现在可以来看看 MySQL 数据库的体系结构了，其结构如图 1-1 所示（摘自 MySQL 官方手册）。

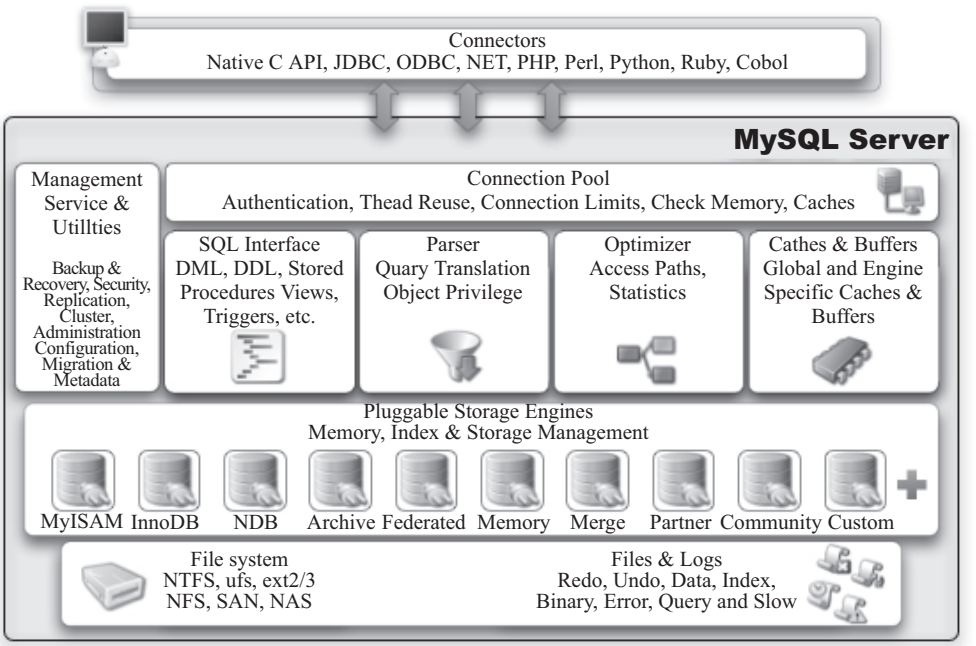


图 1-1　MySQL 体系结构

从图 1-1 可以发现，MySQL 由以下几部分组成：

- 连接池组件
- 管理服务和工具组件
- SQL 接口组件
- 查询分析器组件
- 优化器组件
- 缓冲（Cache）组件
- 插件式存储引擎
- 物理文件

从图 1-1 还可以发现，MySQL 数据库区别于其他数据库的最重要的一个特点就是其插件式的表存储引擎。MySQL 插件式的存储引擎架构提供了一系列标准的管理和服务支持，这些标准与存储引擎本身无关，可能是每个数据库系统本身都必需的，如 SQL 分析器和优化器等，而存储引擎是底层物理结构的实现，每个存储引擎开发者可以按照自己的意愿来进行开发。

需要特别注意的是，存储引擎是基于表的，而不是数据库。此外，要牢记图 1-1 的 MySQL 体系结构，它对于以后深入理解 MySQL 数据库会有极大的帮助。

1.3 MySQL 存储引擎

通过 1.2 节大致了解了 MySQL 数据库独有的插件式体系结构，并了解到存储引擎是 MySQL 区别于其他数据库的一个最重要特性。存储引擎的好处是，每个存储引擎都有各自的特点，能够根据具体的应用建立不同存储引擎表。对于开发人员来说，存储引擎对其是透明的，但了解各种存储引擎的区别对于开发人员来说也是有好处的。对于 DBA 来说，他们应该深刻地认识到 MySQL 数据库的核心在于存储引擎。

由于 MySQL 数据库的开源特性，用户可以根据 MySQL 预定义的存储引擎接口编写自己的存储引擎。若用户对某一种存储引擎的性能或功能不满意，可以通过修改源码来得到想要的特性，这就是开源带给我们的方便与力量。比如，eBay 的工程师 Igor Chernyshev 对 MySQL Memory 存储引擎的改进（http://code.google.com/p/mysql-heap-dynamic-rows/）并应用于 eBay 的 Personalization Platform，类似的修改还有 Google 和 Facebook 等公司。笔者曾尝试过对 InnoDB 存储引擎的缓冲池进行扩展，为其添加了基

于 SSD 的辅助缓冲池[㊀]，通过利用 SSD 的高随机读取性能来进一步提高数据库本身的性能。当然，MySQL 数据库自身提供的存储引擎已经足够满足绝大多数应用的需求。如果用户有兴趣，完全可以开发自己的存储引擎，满足自己特定的需求。MySQL 官方手册的第 16 章给出了编写自定义存储引擎的过程，不过这已超出了本书所涵盖的范围。

由于 MySQL 数据库开源特性，存储引擎可以分为 MySQL 官方存储引擎和第三方存储引擎。有些第三方存储引擎很强大，如大名鼎鼎的 InnoDB 存储引擎（最早是第三方存储引擎，后被 Oracle 收购），其应用就极其广泛，甚至是 MySQL 数据库 OLTP（Online Transaction Processing 在线事务处理）应用中使用最广泛的存储引擎。还是那句话，用户应该根据具体的应用选择适合的存储引擎，以下是对一些存储引擎的简单介绍，以便于读者选择存储引擎时参考。

1.3.1 InnoDB 存储引擎

InnoDB 存储引擎支持事务，其设计目标主要面向在线事务处理（OLTP）的应用。其特点是行锁设计、支持外键，并支持类似于 Oracle 的非锁定读，即默认读取操作不会产生锁。从 MySQL 数据库 5.5.8 版本开始，InnoDB 存储引擎是默认的存储引擎。

InnoDB 存储引擎将数据放在一个逻辑的表空间中，这个表空间就像黑盒一样由 InnoDB 存储引擎自身进行管理。从 MySQL 4.1（包括 4.1）版本开始，它可以将每个 InnoDB 存储引擎的表单独存放到一个独立的 ibd 文件中。此外，InnoDB 存储引擎支持用裸设备（row disk）用来建立其表空间。

InnoDB 通过使用多版本并发控制（MVCC）来获得高并发性，并且实现了 SQL 标准的 4 种隔离级别，默认为 REPEATABLE 级别。同时，使用一种被称为 next-key locking 的策略来避免幻读（phantom）现象的产生。除此之外，InnoDB 储存引擎还提供了插入缓冲（insert buffer）、二次写（double write）、自适应哈希索引（adaptive hash index）、预读（read ahead）等高性能和高可用的功能。

对于表中数据的存储，InnoDB 存储引擎采用了聚集（clustered）的方式，因此每张表的存储都是按主键的顺序进行存放。如果没有显式地在表定义时指定主键，InnoDB 存储引擎会为每一行生成一个 6 字节的 ROWID，并以此作为主键。

㊀ 详见：http://code.google.com/p/david-mysql-tools/wiki/innodb_secondary_buffer_pool

InnoDB 存储引擎是 MySQL 数据库最为常用的一种引擎，而 Facebook、Google、Yahoo ！等公司的成功应用已经证明了 InnoDB 存储引擎具备的高可用性、高性能以及高可扩展性。

1.3.2 MyISAM 存储引擎

MyISAM 存储引擎不支持事务、表锁设计，支持全文索引，主要面向一些 OLAP 数据库应用。在 MySQL 5.5.8 版本之前 MyISAM 存储引擎是默认的存储引擎（除 Windows 版本外）。数据库系统与文件系统很大的一个不同之处在于对事务的支持，然而 MyISAM 存储引擎是不支持事务的。究其根本，这也不是很难理解。试想用户是否在所有的应用中都需要事务呢？在数据仓库中，如果没有 ETL 这些操作，只是简单的报表查询是否还需要事务的支持呢？此外，MyISAM 存储引擎的另一个与众不同的地方是它的缓冲池只缓存（cache）索引文件，而不缓冲数据文件，这点和大多数的数据库都非常不同。

MyISAM 存储引擎表由 MYD 和 MYI 组成，MYD 用来存放数据文件，MYI 用来存放索引文件。可以通过使用 myisampack 工具来进一步压缩数据文件，因为 myisampack 工具使用赫夫曼（Huffman）编码静态算法来压缩数据，因此使用 myisampack 工具压缩后的表是只读的，当然用户也可以通过 myisampack 来解压数据文件。

在 MySQL 5.0 版本之前，MyISAM 默认支持的表大小为 4GB，如果需要支持大于 4GB 的 MyISAM 表时，则需要制定 MAX_ROWS 和 AVG_ROW_LENGTH 属性。从 MySQL 5.0 版本开始，MyISAM 默认支持 256TB 的单表数据，这足够满足一般应用需求。

注意 对于 MyISAM 存储引擎表，MySQL 数据库只缓存其索引文件，数据文件的缓存交由操作系统本身来完成，这与其他使用 LRU 算法缓存数据的大部分数据库大不相同。此外，在 MySQL 5.1.23 版本之前，无论是在 32 位还是 64 位操作系统环境下，缓存索引的缓冲区最大只能设置为 4GB。在之后的版本中，64 位系统可以支持大于 4GB 的索引缓冲区。

1.3.3 NDB 存储引擎

2003 年，MySQL AB 公司从 Sony Ericsson 公司收购了 NDB 集群引擎（见图 1-1）。

NDB 存储引擎是一个集群存储引擎，类似于 Oracle 的 RAC 集群，不过与 Oracle RAC share everything 架构不同的是，其结构是 share nothing 的集群架构，因此能提供更高的可用性。NDB 的特点是数据全部放在内存中（从 MySQL 5.1 版本开始，可以将非索引数据放在磁盘上），因此主键查找（primary key lookups）的速度极快，并且通过添加 NDB 数据存储节点（Data Node）可以线性地提高数据库性能，是高可用、高性能的集群系统。

关于 NDB 存储引擎，有一个问题值得注意，那就是 NDB 存储引擎的连接操作（JOIN）是在 MySQL 数据库层完成的，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢。如果解决了这个问题，NDB 存储引擎的市场应该是非常巨大的。

注意 MySQL NDB Cluster 存储引擎有社区版本和企业版本两种，并且 NDB Cluster 已作为 Carrier Grade Edition 单独下载版本而存在，可以通过 http://dev.mysql.com/downloads/cluster/index.html 获得最新版本的 NDB Cluster 存储引擎。

1.3.4 Memory 存储引擎

Memory 存储引擎（之前称 HEAP 存储引擎）将表中的数据存放在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时表，以及数据仓库中的纬度表。Memory 存储引擎默认使用哈希索引，而不是我们熟悉的 B+ 树索引。

虽然 Memory 存储引擎速度非常快，但在使用上还是有一定的限制。比如，只支持表锁，并发性能较差，并且不支持 TEXT 和 BLOB 列类型。最重要的是，存储变长字段（varchar）时是按照定常字段（char）的方式进行的，因此会浪费内存（这个问题之前已经提到，eBay 的工程师 Igor Chernyshev 已经给出了 patch 解决方案）。

此外有一点容易被忽视，MySQL 数据库使用 Memory 存储引擎作为临时表来存放查询的中间结果集（intermediate result）。如果中间结果集大于 Memory 存储引擎表的容量设置，又或者中间结果含有 TEXT 或 BLOB 列类型字段，则 MySQL 数据库会把其转换到 MyISAM 存储引擎表而存放到磁盘中。之前提到 MyISAM 不缓存数据文件，因此这时产生的临时表的性能对于查询会有损失。

1.3.5 Archive 存储引擎

Archive 存储引擎只支持 INSERT 和 SELECT 操作，从 MySQL 5.1 开始支持索引。Archive 存储引擎使用 zlib 算法将数据行（row）进行压缩后存储，压缩比一般可达 1∶10。正如其名字所示，Archive 存储引擎非常适合存储归档数据，如日志信息。Archive 存储引擎使用行锁来实现高并发的插入操作，但是其本身并不是事务安全的存储引擎，其设计目标主要是提供高速的插入和压缩功能。

1.3.6 Federated 存储引擎

Federated 存储引擎表并不存放数据，它只是指向一台远程 MySQL 数据库服务器上的表。这非常类似于 SQL Server 的链接服务器和 Oracle 的透明网关，不同的是，当前 Federated 存储引擎只支持 MySQL 数据库表，不支持异构数据库表。

1.3.7 Maria 存储引擎

Maria 存储引擎是新开发的引擎，设计目标主要是用来取代原有的 MyISAM 存储引擎，从而成为 MySQL 的默认存储引擎。Maria 存储引擎的开发者是 MySQL 的创始人之一的 Michael Widenius。因此，它可以看做是 MyISAM 的后续版本。Maria 存储引擎的特点是：支持缓存数据和索引文件，应用了行锁设计，提供了 MVCC 功能，支持事务和非事务安全的选项，以及更好的 BLOB 字符类型的处理性能。

1.3.8 其他存储引擎

除了上面提到的 7 种存储引擎外，MySQL 数据库还有很多其他的存储引擎，包括 Merge、CSV、Sphinx 和 Infobright，它们都有各自使用的场合，这里不再一一介绍。在了解 MySQL 数据库拥有这么多存储引擎后，现在我可以回答 1.2 节中提到的问题了。

- 为什么 MySQL 数据库不支持全文索引？不！ MySQL 支持，MyISAM、InnoDB（1.2 版本）和 Sphinx 存储引擎都支持全文索引。
- MySQL 数据库速度快是因为不支持事务？错！虽然 MySQL 的 MyISAM 存储引擎不支持事务，但是 InnoDB 支持。“快”是相对于不同应用来说的，对于 ETL 这种操作，MyISAM 会有其优势，但在 OLTP 环境中，InnoDB 存储引擎的效率

更好。

- 当表的数据量大于 1000 万时 MySQL 的性能会急剧下降吗？不！ MySQL 是数据库，不是文件，随着数据行数的增加，性能当然会有所下降，但是这些下降不是线性的，如果用户选择了正确的存储引擎，以及正确的配置，再多的数据量 MySQL 也能承受。如官方手册上提及的，Mytrix 和 Inc. 在 InnoDB 上存储超过 1 TB 的数据，还有一些其他网站使用 InnoDB 存储引擎，处理插入 / 更新的操作平均 800 次 / 秒。

1.4　各存储引擎之间的比较

通过 1.3 节的介绍，我们了解了存储引擎是 MySQL 体系结构的核心。本节我们将通过简单比较几个存储引擎来让读者更直观地理解存储引擎的概念。图 1-2 取自于 MySQL 的官方手册，展现了一些常用 MySQL 存储引擎之间的不同之处，包括存储容量的限制、事务支持、锁的粒度、MVCC 支持、支持的索引、备份和复制等。

Feature	MyISAM	BDB	Memory	InnoDB	Archive	NDB
Storage Limits	No	No	Yes	64TB	No	Yes
Transactions (commit, rollback, etc.)		✔		✔		
Locking granularity	Table	Page	Table	Row	Row	Row
MVCC/Snapshot Read				✔	✔	✔
Geospatial support	✔					
B-Tree indexes	✔	✔	✔	✔		✔
Hash indexes			✔	✔		✔
Full text search index	✔					
Clustered index				✔		
Data Caches			✔	✔		✔
Index Caches	✔		✔	✔		✔
Compressed data	✔				✔	
Encrypted data (via function)	✔	✔	✔	✔	✔	✔
Storage cost (space used)	Low	Low	N/A	High	Very Low	Low
Memory cost	Low	Low	Medium	High	Low	High
Bulk Insert Speed	High	High	High	Low	Very High	High
Cluster database support						✔
Replication support	✔	✔	✔	✔	✔	✔
Foreign key support				✔		
Backup/Point-in-time recovery	✔	✔	✔	✔	✔	✔
Query cache support	✔	✔	✔	✔	✔	✔
Update Statistics for Data Dictionary	✔	✔	✔	✔	✔	✔

图 1-2　不同 MySQL 存储引擎相关特性比较

可以看到，每种存储引擎的实现都不相同。有些竟然不支持事务，相信在任何一本关于数据库原理的书中，可能都会提到数据库与传统文件系统的最大区别在于数据库是支持事务的。而 MySQL 数据库的设计者在开发时却认为可能不是所有的应用都需要事务，所以存在不支持事务的存储引擎。更有不明其理的人把 MySQL 称做文件系统数据库，其实不然，只是 MySQL 数据库的设计思想和存储引擎的关系可能让人产生了理解上的偏差。

可以通过 SHOW ENGINES 语句查看当前使用的 MySQL 数据库所支持的存储引擎，也可以通过查找 information_schema 架构下的 ENGINES 表，如下所示：

```
mysql>SHOW ENGINES\G;
*************************** 1. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 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)
```

下面将通过 MySQL 提供的示例数据库来简单显示各存储引擎之间的不同。这里将分别运行以下语句，然后统计每次使用各存储引擎后表的大小。

```
mysql>CREATE TABLE mytest Engine=MyISAM
    ->AS SELECT * FROM salaries;
Query OK, 2844047 rows affected (4.37 sec)
Records: 2844047  Duplicates: 0  Warnings: 0

mysql>ALTER TABLE mytest Engine=InnoDB;
Query OK, 2844047 rows affected (15.86 sec)
Records: 2844047  Duplicates: 0  Warnings: 0

mysql>ALTER TABLE mytest Engine=ARCHIVE;
Query OK, 2844047 rows affected (16.03 sec)
Records: 2844047  Duplicates: 0  Warnings: 0
```

通过每次的统计，可以发现当最初表使用 MyISAM 存储引擎时，表的大小为 40.7MB，使用 InnoDB 存储引擎时表增大到了 113.6MB，而使用 Archive 存储引擎时表的大小却只有 20.2MB。该例子只从表的大小方面简单地揭示了各存储引擎的不同。

注意 MySQL 提供了一个非常好的用来演示 MySQL 各项功能的示例数据库，如 SQL Server 提供的 AdventureWorks 示例数据库和 Oracle 提供的示例数据库。据我所知，知道 MySQL 示例数据库的人很少，可能是因为这个示例数据库没有在安装的时候提示用户是否安装（如 Oracle 和 SQL Server）以及这个示例数据库的下载竟然和文档放在一起。用户可以通过以下地址找到并下载示例数据库：http://dev.mysql.com/doc/。

1.5 连接 MySQL

本节将介绍连接 MySQL 数据库的常用方式。需要理解的是，连接 MySQL 操作是一个连接进程和 MySQL 数据库实例进行通信。从程序设计的角度来说，本质上是进程通信。如果对进程通信比较了解，可以知道常用的进程通信方式有管道、命名管道、命名字、TCP/IP 套接字、UNIX 域套接字。MySQL 数据库提供的连接方式从本质上看都是上述提及的进程通信方式。

1.5.1 TCP/IP

TCP/IP 套接字方式是 MySQL 数据库在任何平台下都提供的连接方式，也是网络中使用得最多的一种方式。这种方式在 TCP/IP 连接上建立一个基于网络的连接请求，一般情况下客户端（client）在一台服务器上，而 MySQL 实例（server）在另一台服务器上，这两台机器通过一个 TCP/IP 网络连接。例如用户可以在 Windows 服务器下请求一台远程 Linux 服务器下的 MySQL 实例，如下所示：

```
C:\>mysql -h192.168.0.101 -u david -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18358
```

```
Server version: 5.0.77-log MySQL Community Server (GPL)

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

mysql>
```

这里的客户端是 Windows，它向一台 Host IP 为 192.168.0.101 的 MySQL 实例发起了 TCP/IP 连接请求，并且连接成功。之后就可以对 MySQL 数据库进行一些数据库操作，如 DDL 和 DML 等。

这里需要注意的是，在通过 TCP/IP 连接到 MySQL 实例时，MySQL 数据库会先检查一张权限视图，用来判断发起请求的客户端 IP 是否允许连接到 MySQL 实例。该视图在 mysql 架构下，表名为 user，如下所示：

```
mysql>USE mysql;
Database changed
mysql>SELECT host,user,password FROM user;
*************************** 1. row ***************************
host: 192.168.24.%
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 2. row ***************************
host: nineyou0-43
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 3. row ***************************
host: 127.0.0.1
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 4. row ***************************
host: 192.168.0.100
user: zlm
password: *DAE0939275CC7CD8E0293812A31735DA9CF0953C
*************************** 5. row ***************************
host: %
user: david
password:
5 rows in set (0.00 sec)
```

从这张权限表中可以看到，MySQL 允许 david 这个用户在任何 IP 段下连接该实例，并且不需要密码。此外，还给出了 root 用户在各个网段下的访问控制权限。

1.5.2 命名管道和共享内存

在 Windows 2000、Windows XP、Windows 2003 和 Windows Vista 以及在此之上的平台上，如果两个需要进程通信的进程在同一台服务器上，那么可以使用命名管道，Microsoft SQL Server 数据库默认安装后的本地连接也是使用命名管道。在 MySQL 数据库中须在配置文件中启用 --enable-named-pipe 选项。在 MySQL 4.1 之后的版本中，MySQL 还提供了共享内存的连接方式，这是通过在配置文件中添加 --shared-memory 实现的。如果想使用共享内存的方式，在连接时，MySQL 客户端还必须使用 --protocol=memory 选项。

1.5.3 UNIX 域套接字

在 Linux 和 UNIX 环境下，还可以使用 UNIX 域套接字。UNIX 域套接字其实不是一个网络协议，所以只能在 MySQL 客户端和数据库实例在一台服务器上的情况下使用。用户可以在配置文件中指定套接字文件的路径，如 --socket=/tmp/mysql.sock。当数据库实例启动后，用户可以通过下列命令来进行 UNIX 域套接字文件的查找：

```
mysql>SHOW VARIABLES LIKE 'socket';
*************************** 1. row ***************************
Variable_name: socket
       Value: /tmp/mysql.sock
1 row in set (0.00 sec)
```

在知道了 UNIX 域套接字文件的路径后，就可以使用该方式进行连接了，如下所示：

```
[root@stargazer ~]# mysql -udavid -S /tmp/mysql.sock
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 20333
Server version: 5.0.77-log MySQL Community Server (GPL)

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


mysql>
```

1.6 小结

本章首先介绍了数据库和数据库实例的定义，紧接着分析了 MySQL 数据库的体系

结构，从而进一步突出强调了“实例”和“数据库”的区别。相信不管是 MySQL DBA 还是 MySQL 的开发人员都应该从宏观上了解了 MySQL 体系结构，特别是 MySQL 独有的插件式存储引擎的概念。因为很多 MySQL 用户很少意识到这一点，这给他们的管理、使用和开发带来了困扰。

本章还详细讲解了各种常见的表存储引擎的特性、适用情况以及它们之间的区别，以便于大家在选择存储引擎时作为参考。最后强调一点，虽然 MySQL 有许多的存储引擎，但是它们之间不存在优劣性的差异，用户应根据不同的应用选择适合自己的存储引擎。当然，如果你能力很强，完全可以修改存储引擎的源代码，甚至是创建属于自己特定应用的存储引擎，这不就是开源的魅力吗？

第 2 章　InnoDB 存储引擎

InnoDB 是事务安全的 MySQL 存储引擎，设计上采用了类似于 Oracle 数据库的架构。通常来说，InnoDB 存储引擎是 OLTP 应用中核心表的首选存储引擎。同时，也正是因为 InnoDB 的存在，才使 MySQL 数据库变得更有魅力。本章将详细介绍 InnoDB 存储引擎的体系架构及其不同于其他存储引擎的特性。

2.1　InnoDB 存储引擎概述

InnoDB 存储引擎最早由 Innobase Oy 公司[㊀]开发，被包括在 MySQL 数据库所有的二进制发行版本中，从 MySQL 5.5 版本开始是默认的表存储引擎（之前的版本 InnoDB 存储引擎仅在 Windows 下为默认的存储引擎）。该存储引擎是第一个完整支持 ACID 事务的 MySQL 存储引擎（BDB 是第一个支持事务的 MySQL 存储引擎，现在已经停止开发），其特点是行锁设计、支持 MVCC、支持外键、提供一致性非锁定读，同时被设计用来最有效地利用以及使用内存和 CPU。

Heikki Tuuri（1964 年，芬兰赫尔辛基）是 InnoDB 存储引擎的创始人，和著名的 Linux 创始人 Linus 是芬兰赫尔辛基大学校友。在 1990 年获得赫尔辛基大学的数学逻辑博士学位后，他于 1995 年成立 Innobase Oy 公司并担任 CEO。同时，在 InnoDB 存储引擎的开发团队中，有来自中国科技大学的 Calvin Sun。而最近又有一个中国人 Jimmy Yang 也加入了 InnoDB 存储引擎的核心开发团队，负责全文索引的开发，其之前任职于 Sybase 数据库公司，负责数据库的相关开发工作。

InnoDB 存储引擎已经被许多大型网站使用，如用户熟知的 Google、Yahoo!、Facebook、YouTube、Flickr，在网络游戏领域有《魔兽世界》、《Second Life》、《神兵玄奇》等。我不是 MySQL 数据库的布道者，也不是 InnoDB 的鼓吹者，但是我认为当前实施一个新的 OLTP 项目不使用 MySQL InnoDB 存储引擎将是多么的愚蠢。

㊀ 2006年该公司已经被Oracle公司收购。

从 MySQL 数据库的官方手册可得知，著名的 Internet 新闻站点 Slashdot.org 运行在 InnoDB 上。Mytrix、Inc. 在 InnoDB 上存储超过 1 TB 的数据，还有一些其他站点在 InnoDB 上处理插入 / 更新操作的速度平均为 800 次 / 秒。这些都证明了 InnoDB 是一个高性能、高可用、高可扩展的存储引擎。

InnoDB 存储引擎同 MySQL 数据库一样，在 GNU GPL 2 下发行。更多有关 MySQL 证书的信息，可参考 http://www.mysql.com/about/legal/，这里不再详细介绍。

2.2　InnoDB 存储引擎的版本

InnoDB 存储引擎被包含于所有 MySQL 数据库的二进制发行版本中。早期其版本随着 MySQL 数据库的更新而更新。从 MySQL 5.1 版本时，MySQL 数据库允许存储引擎开发商以动态方式加载引擎，这样存储引擎的更新可以不受 MySQL 数据库版本的限制。所以在 MySQL 5.1 中，可以支持两个版本的 InnoDB，一个是静态编译的 InnoDB 版本，可将其视为老版本的 InnoDB ；另一个是动态加载的 InnoDB 版本，官方称为 InnoDB Plugin，可将其视为 InnoDB 1.0.x 版本。MySQL 5.5 版本中又将 InnoDB 的版本升级到了 1.1.x。而在最近的 MySQL 5.6 版本中 InnoDB 的版本也随着升级为 1.2.x 版本。表 2-1 显示了各个版本中 InnoDB 存储引擎的功能。

表 2-1　InnoDB 各版本功能对比

版　本	功　能
老版本 InnoDB	支持 ACID、行锁设计、MVCC
InnoDB 1.0.x	继承了上述版本所有功能，增加了 compress 和 dynamic 页格式
InnoDB 1.1.x	继承了上述版本所有功能，增加了 Linux AIO、多回滚段
InnoDB 1.2.x	继承了上述版本所有功能，增加了全文索引支持、在线索引添加

在现实工作中我发现很多 MySQL 数据库还是停留在 MySQL 5.1 版本，并使用 InnoDB Plugin。很多 DBA 错误地认为 InnoDB Plugin 和 InnoDB 1.1 版本之间是没有区别的。但从表 2-1 中还是可以发现，虽然都增加了对于 compress 和 dynamic 页的支持，但是 InnoDB Plugin 是不支持 Linux Native AIO 功能的。此外，由于不支持多回滚段，InnoDB Plugin 支持的最大支持并发事务数量也被限制在 1023。而且随着 MySQL 5.5 版本的发布，InnoDB Plugin 也变成了一个历史产品。

2.3 InnoDB 体系架构

通过第 1 章读者已经了解了 MySQL 数据库的体系结构，现在可能想更深入地了解 InnoDB 存储引擎的架构。图 2-1 简单显示了 InnoDB 的存储引擎的体系架构，从图可见，InnoDB 存储引擎有多个内存块，可以认为这些内存块组成了一个大的内存池，负责如下工作：

- 维护所有进程 / 线程需要访问的多个内部数据结构。
- 缓存磁盘上的数据，方便快速地读取，同时在对磁盘文件的数据修改之前在这里缓存。
- 重做日志（redo log）缓冲。

……

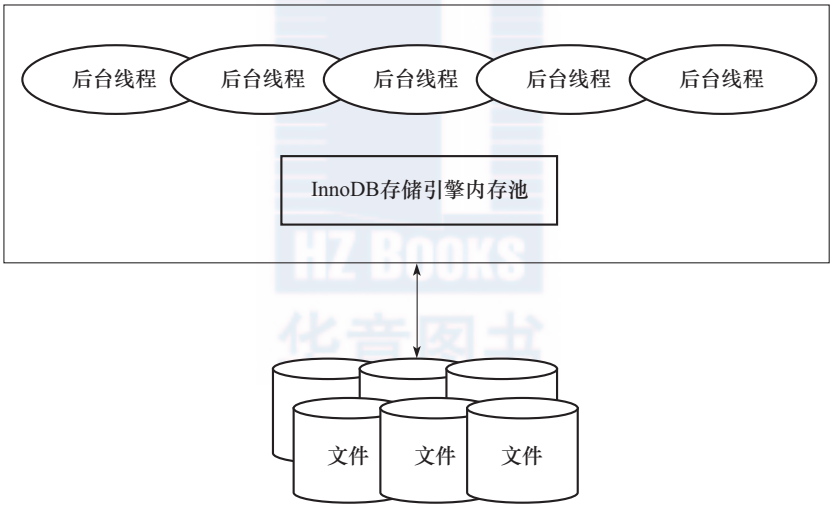


图 2-1 InnoDB 存储引擎体系架构

后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外将已修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常的情况下 InnoDB 能恢复到正常运行状态。

2.3.1 后台线程

InnoDB 存储引擎是多线程的模型，因此其后台有多个不同的后台线程，负责处理不

同的任务。

1. Master Thread

Master Thread 是一个非常核心的后台线程，主要负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲（INSERT BUFFER）、UNDO 页的回收等。2.5 节会详细地介绍各个版本中 Master Thread 的工作方式。

2. IO Thread

在 InnoDB 存储引擎中大量使用了 AIO（Async IO）来处理写 IO 请求，这样可以极大提高数据库的性能。而 IO Thread 的工作主要是负责这些 IO 请求的回调（call back）处理。InnoDB 1.0 版本之前共有 4 个 IO Thread，分别是 write、read、insert buffer 和 log IO thread。在 Linux 平台下，IO Thread 的数量不能进行调整，但是在 Windows 平台下可以通过参数 innodb_file_io_threads 来增大 IO Thread。从 InnoDB 1.0.x 版本开始，read thread 和 write thread 分别增大到了 4 个，并且不再使用 innodb_file_io_threads 参数，而是分别使用 innodb_read_io_threads 和 innodb_write_io_threads 参数进行设置，如：

```
mysql>SHOW VARIABLES LIKE 'innodb_version'\G;
*************************** 1. row ***************************
Variable_name: innodb_version
       Value: 1.0.6
1 row in set (0.00 sec)

mysql>SHOW VARIABLES LIKE 'innodb_%io_threads'\G;
*************************** 1. row ***************************
Variable_name: innodb_read_io_threads
       Value: 4
*************************** 2. row ***************************
Variable_name: innodb_write_io_threads
       Value: 4
2 rows in set (0.00 sec)
```

可以通过命令 SHOW ENGINE INNODB STATUS 来观察 InnoDB 中的 IO Thread：

```
mysql>SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100719 21:55:26 INNODB MONITOR OUTPUT
=====================================
```

```
Per second averages calculated from the last 36 seconds
……
--------
FILE I/O
--------
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)
……
-----------------------------
END OF INNODB MONITOR OUTPUT
=============================


1 row in set (0.01 sec)
```

可以看到 IO Thread 0 为 insert buffer thread。IO Thread 1 为 log thread。之后就是根据参数 innodb_read_io_threads 及 innodb_write_io_threads 来设置的读写线程，并且读线程的 ID 总是小于写线程。

3. Purge Thread

事务被提交后，其所使用的 undolog 可能不再需要，因此需要 PurgeThread 来回收已经使用并分配的 undo 页。在 InnoDB 1.1 版本之前，purge 操作仅在 InnoDB 存储引擎的 Master Thread 中完成。而从 InnoDB 1.1 版本开始，purge 操作可以独立到单独的线程中进行，以此来减轻 Master Thread 的工作，从而提高 CPU 的使用率以及提升存储引擎的性能。用户可以在 MySQL 数据库的配置文件中添加如下命令来启用独立的 Purge Thread：

```
[mysqld]
innodb_purge_threads=1
```

在 InnoDB 1.1 版本中，即使将 innodb_purge_threads 设为大于 1，InnoDB 存储引擎启动时也会将其设为 1，并在错误文件中出现如下类似的提示：

```
120529 22:54:16 [Warning] option 'innodb-purge-threads': unsigned value 4 adjusted to 1
```

从 InnoDB 1.2 版本开始，InnoDB 支持多个 Purge Thread，这样做的目的是为了进一步加快 undo 页的回收。同时由于 Purge Thread 需要离散地读取 undo 页，这样也能更进一步利用磁盘的随机读取性能。如用户可以设置 4 个 Purge Thread：

```
mysql> SELECT VERSION()\G;
*************************** 1. row ***************************
VERSION(): 5.6.6
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'innodb_purge_threads'\G;
*************************** 1. row ***************************
Variable_name: innodb_purge_threads
       Value: 4
1 row in set (0.00 sec)
```

4. Page Cleaner Thread

Page Cleaner Thread 是在 InnoDB 1.2.x 版本中引入的。其作用是将之前版本中脏页的刷新操作都放入到单独的线程中来完成。而其目的是为了减轻原 Master Thread 的工作及对于用户查询线程的阻塞，进一步提高 InnoDB 存储引擎的性能。

2.3.2 内存

1. 缓冲池

InnoDB 存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。因此可将其视为基于磁盘的数据库系统（Disk-base Database）。在数据库系统中，由于 CPU 速度与磁盘速度之间的鸿沟，基于磁盘的数据库系统通常使用缓冲池技术来提高数据库的整体性能。

缓冲池简单来说就是一块内存区域，通过内存的速度来弥补磁盘速度较慢对数据库性能的影响。在数据库中进行读取页的操作，首先将从磁盘读到的页存放在缓冲池中，这个过程称为将页“FIX”在缓冲池中。下一次再读相同的页时，首先判断该页是否在缓冲池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。否则，读取磁盘上的页。

对于数据库中页的修改操作，则首先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。这里需要注意的是，页从缓冲池刷新回磁盘的操作并不是在每次页发生更

新时触发，而是通过一种称为 Checkpoint 的机制刷新回磁盘。同样，这也是为了提高数据库的整体性能。

综上所述，缓冲池的大小直接影响着数据库的整体性能。由于 32 位操作系统的限制，在该系统下最多将该值设置为 3G。此外用户可以打开操作系统的 PAE 选项来获得 32 位操作系统下最大 64GB 内存的支持。随着内存技术的不断成熟，其成本也在不断下降。单条 8GB 的内存变得非常普遍，而 PC 服务器已经能支持 512GB 的内存。因此为了让数据库使用更多的内存，强烈建议数据库服务器都采用 64 位的操作系统。

对于 InnoDB 存储引擎而言，其缓冲池的配置通过参数 innodb_buffer_pool_size 来设置。下面显示一台 MySQL 数据库服务器，其将 InnoDB 存储引擎的缓冲池设置为 15GB。

```
mysql>SHOW VARIABLES LIKE 'innodb_buffer_pool_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 16106127360
1 row in set (0.00 sec)
```

具体来看，缓冲池中缓存的数据页类型有：索引页、数据页、undo 页、插入缓冲（insert buffer）、自适应哈希索引（adaptive hash index）、InnoDB 存储的锁信息（lock info）、数据字典信息（data dictionary）等。不能简单地认为，缓冲池只是缓存索引页和数据页，它们只是占缓冲池很大的一部分而已。图 2-2 很好地显示了 InnoDB 存储引擎中内存的结构情况。

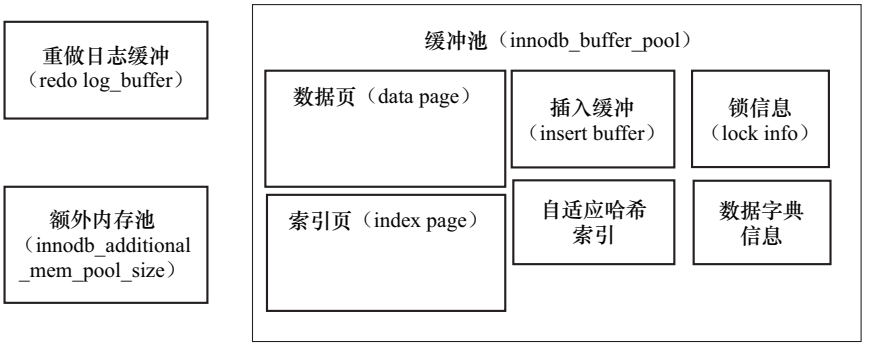


图 2-2 InnoDB 内存数据对象

从 InnoDB 1.0.x 版本开始，允许有多个缓冲池实例。每个页根据哈希值平均分配到不同缓冲池实例中。这样做的好处是减少数据库内部的资源竞争，增加数据库的并发处

理能力。可以通过参数 innodb_buffer_pool_instances 来进行配置，该值默认为 1。

```
mysql> SHOW VARIABLES LIKE 'innodb_buffer_pool_instances'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_instances
       Value: 1
1 row in set (0.00 sec)
```

在配置文件中将 innodb_buffer_pool_instances 设置为大于 1 的值就可以得到多个缓冲池实例。再通过命令 SHOW ENGINE INNODB STATUS 可以观察到如下的内容：

```
mysql> SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
……
----------------------
INDIVIDUAL BUFFER POOL INFO
----------------------
---BUFFER POOL 0
Buffer pool size   65535
Free buffers       65451
Database pages     84
Old database pages 0
Modified db pages  0
Pending reads 0
Pending writes: LRU 0, flush list 0 single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 84, created 0, written 1
9.33 reads/s, 0.00 creates/s, 0.11 writes/s
Buffer pool hit rate 764 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 84, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
---BUFFER POOL 1
Buffer pool size   65536
Free buffers       65473
Database pages     63
Old database pages 0
Modified db pages  0
Pending reads 0
Pending writes: LRU 0, flush list 0 single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
```

```
Pages read 63, created 0, written 0
7.00 reads/s, 0.00 creates/s, 0.00 writes/s
Buffer pool hit rate 500 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 63, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
```

这里将参数 innodb_buffer_pool_instances 设置为 2，即数据库用户拥有两个缓冲池实例。通过命令 SHOW ENGINE INNODB STATUS 可以观察到每个缓冲池实例对象运行的状态，并且通过类似 ---BUFFER POOL 0 的注释来表明是哪个缓冲池实例。

从 MySQL 5.6 版本开始，还可以通过 information_schema 架构下的表 INNODB_BUFFER_POOL_STATS 来观察缓冲的状态，如运行下列命令可以看到各个缓冲池的使用状态：

```
mysql> SELECT POOL_ID,POOL_SIZE,
    -> FREE_BUFFERS,DATABASE_PAGES
    -> FROM INNODB_BUFFER_POOL_STATS\G;
*************************** 1. row ***************************
       POOL_ID: 0
     POOL_SIZE: 65535
  FREE_BUFFERS: 65451
DATABASE_PAGES: 84
*************************** 2. row ***************************
       POOL_ID: 1
     POOL_SIZE: 65536
  FREE_BUFFERS: 65473
DATABASE_PAGES: 63
```

2. LRU List、Free List 和 Flush List

在前一小节中我们知道了缓冲池是一个很大的内存区域，其中存放各种类型的页。那么 InnoDB 存储引擎是怎么对这么大的内存区域进行管理的呢？这就是本小节要告诉读者的。

通常来说，数据库中的缓冲池是通过 LRU（Latest Recent Used，最近最少使用）算法来进行管理的。即最频繁使用的页在 LRU 列表的前端，而最少使用的页在 LRU 列表的尾端。当缓冲池不能存放新读取到的页时，将首先释放 LRU 列表中尾端的页。

在 InnoDB 存储引擎中，缓冲池中页的大小默认为 16KB，同样使用 LRU 算法对缓

冲池进行管理。稍有不同的是 InnoDB 存储引擎对传统的 LRU 算法做了一些优化。在 InnoDB 的存储引擎中，LRU 列表中还加入了 midpoint 位置。新读取到的页，虽然是最新访问的页，但并不是直接放入到 LRU 列表的首部，而是放入到 LRU 列表的 midpoint 位置。这个算法在 InnoDB 存储引擎下称为 midpoint insertion strategy。在默认配置下，该位置在 LRU 列表长度的 5/8 处。midpoint 位置可由参数 innodb_old_blocks_pct 控制，如：

```
mysql> SHOW VARIABLES LIKE 'innodb_old_blocks_pct'\G;
*************************** 1. row ***************************
Variable_name: innodb_old_blocks_pct
        Value: 37
1 row in set (0.00 sec)
```

从上面的例子可以看到，参数 innodb_old_blocks_pct 默认值为 37，表示新读取的页插入到 LRU 列表尾端的 37% 的位置（差不多 3/8 的位置）。在 InnoDB 存储引擎中，把 midpoint 之后的列表称为 old 列表，之前的列表称为 new 列表。可以简单地理解为 new 列表中的页都是最为活跃的热点数据。

那为什么不采用朴素的 LRU 算法，直接将读取的页放入到 LRU 列表的首部呢？这是因为若直接将读取到的页放入到 LRU 的首部，那么某些 SQL 操作可能会使缓冲池中的页被刷新出，从而影响缓冲池的效率。常见的这类操作为索引或数据的扫描操作。这类操作需要访问表中的许多页，甚至是全部的页，而这些页通常来说又仅在这次查询操作中需要，并不是活跃的热点数据。如果页被放入 LRU 列表的首部，那么非常可能将所需要的热点数据页从 LRU 列表中移除，而在下一次需要读取该页时，InnoDB 存储引擎需要再次访问磁盘。

为了解决这个问题，InnoDB 存储引擎引入了另一个参数来进一步管理 LRU 列表，这个参数是 innodb_old_blocks_time，用于表示页读取到 mid 位置后需要等待多久才会被加入到 LRU 列表的热端。因此当需要执行上述所说的 SQL 操作时，可以通过下面的方法尽可能使 LRU 列表中热点数据不被刷出。

```
mysql> SET GLOBAL innodb_old_blocks_time=1000;
Query OK, 0 rows affected (0.00 sec)

# data or index scan operation
……

mysql> SET GLOBAL innodb_old_blocks_time=0;
```

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

如果用户预估自己活跃的热点数据不止 63%，那么在执行 SQL 语句前，还可以通过下面的语句来减少热点页可能被刷出的概率。

```
mysql> SET GLOBAL innodb_old_blocks_pct=20;
Query OK, 0 rows affected (0.00 sec)
```

LRU 列表用来管理已经读取的页，但当数据库刚启动时，LRU 列表是空的，即没有任何的页。这时页都存放在 Free 列表中。当需要从缓冲池中分页时，首先从 Free 列表中查找是否有可用的空闲页，若有则将该页从 Free 列表中删除，放入到 LRU 列表中。否则，根据 LRU 算法，淘汰 LRU 列表末尾的页，将该内存空间分配给新的页。当页从 LRU 列表的 old 部分加入到 new 部分时，称此时发生的操作为 page made young，而因为 innodb_old_blocks_time 的设置而导致页没有从 old 部分移动到 new 部分的操作称为 page not made young。可以通过命令 SHOW ENGINE INNODB STATUS 来观察 LRU 列表及 Free 列表的使用情况和运行状态。

```
mysql> SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
120725 22:04:25 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 24 seconds
……
Buffer pool size   327679
Free buffers       0
Database pages     307717
Old database pages 113570
Modified db pages  24673
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 6448526, not young 0
48.75 youngs/s, 0.00 non-youngs/s
Pages read 5354420, created 239625, written 3486063
55.68 reads/s, 81.74 creates/s, 955.88 writes/s
Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not 0 / 1000
……
```

通过命令 SHOW ENGINE INNODB STATUS 可以看到：当前 Buffer pool size 共有 327 679 个页，即 327679*16K，总共 5GB 的缓冲池。Free buffers 表示当前 Free 列表中页的数量，Database pages 表示 LRU 列表中页的数量。可能的情况是 Free buffers 与 Database pages 的数量之和不等于 Buffer pool size。正如图 2-2 所示的那样，因为缓冲池中的页还可能会被分配给自适应哈希索引、Lock 信息、Insert Buffer 等页，而这部分页不需要 LRU 算法进行维护，因此不存在于 LRU 列表中。

pages made young 显示了 LRU 列表中页移动到前端的次数，因为该服务器在运行阶段没有改变 innodb_old_blocks_time 的值，因此 not young 为 0。youngs/s、non-youngs/s 表示每秒这两类操作的次数。这里还有一个重要的观察变量——Buffer pool hit rate，表示缓冲池的命中率，这个例子中为 100%，说明缓冲池运行状态非常良好。通常该值不应该小于 95%。若发生 Buffer pool hit rate 的值小于 95% 这种情况，用户需要观察是否是由于全表扫描引起的 LRU 列表被污染的问题。

注意 执行命令 SHOW ENGINE INNODB STATUS 显示的不是当前的状态，而是过去某个时间范围内 InnoDB 存储引擎的状态。从上面的例子可以发现，Per second averages calculated from the last 24 seconds 代表的信息为过去 24 秒内的数据库状态。

从 InnoDB 1.2 版本开始，还可以通过表 INNODB_BUFFER_POOL_STATS 来观察缓冲池的运行状态，如：

```
mysql> SELECT POOL_ID,HIT_RATE,
-> PAGES_MADE_YOUNG, PAGES_NOT_MADE_YOUNG
-> FROM information_schema.INNODB_BUFFER_POOL_STATS\G;
*************************** 1. row ***************************
         POOL_ID: 0
        HIT_RATE: 980
 PAGES_MADE_YOUNG: 450
PAGES_NOT_MADE_YOUNG: 0
```

此外，还可以通过表 INNODB_BUFFER_PAGE_LRU 来观察每个 LRU 列表中每个页的具体信息，例如通过下面的语句可以看到缓冲池 LRU 列表中 SPACE 为 1 的表的页类型：

```
mysql> SELECT TABLE_NAME,SPACE,PAGE_NUMBER,PAGE_TYPE
    -> FROM INNODB_BUFFER_PAGE_LRU WHERE SPACE = 1;
```

```
+------------+-------+-------------+--------------------+
| TABLE_NAME | SPACE | PAGE_NUMBER | PAGE_TYPE          |
+------------+-------+-------------+--------------------+
| NULL       |     1 |           0 | FILE_SPACE_HEADER  |
| NULL       |     1 |           1 | IBUF_BITMAP        |
| NULL       |     1 |           2 | INODE              |
| test/t|    |     1 |           3 | INDEX              |
+------------+-------+-------------+--------------------+
4 rows in set (0.00 sec)
```

InnoDB 存储引擎从 1.0.x 版本开始支持压缩页的功能，即将原本 16KB 的页压缩为 1KB、2KB、4KB 和 8KB。而由于页的大小发生了变化，LRU 列表也有了些许的改变。对于非 16KB 的页，是通过 unzip_LRU 列表进行管理的。通过命令 SHOW ENGINE INNODB STATUS 可以观察到如下内容：

```
mysql> SHOW ENGINE INNODB STATUS\G;
……
Buffer pool hit rate 999 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 1539, unzip_LRU len: 156
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
……
```

可以看到 LRU 列表中一共有 1539 个页，而 unzip_LRU 列表中有 156 个页。这里需要注意的是，LRU 中的页包含了 unzip_LRU 列表中的页。

对于压缩页的表，每个表的压缩比率可能各不相同。可能存在有的表页大小为 8KB，有的表页大小为 2KB 的情况。unzip_LRU 是怎样从缓冲池中分配内存的呢？

首先，在 unzip_LRU 列表中对不同压缩页大小的页进行分别管理。其次，通过伙伴算法进行内存的分配。例如对需要从缓冲池中申请页为 4KB 的大小，其过程如下：

1）检查 4KB 的 unzip_LRU 列表，检查是否有可用的空闲页；

2）若有，则直接使用；

3）否则，检查 8KB 的 unzip_LRU 列表；

4）若能够得到空闲页，将页分成 2 个 4KB 页，存放到 4KB 的 unzip_LRU 列表；

5）若不能得到空闲页，从 LRU 列表中申请一个 16KB 的页，将页分为 1 个 8KB 的页、2 个 4KB 的页，分别存放到对应的 unzip_LRU 列表中。

同样可以通过 information_schema 架构下的表 INNODB_BUFFER_PAGE_LRU 来观察 unzip_LRU 列表中的页，如：

```
mysql> SELECT
-> TABLE_NAME,SPACE,PAGE_NUMBER,COMPRESSED_SIZE
-> FROM INNODB_BUFFER_PAGE_LRU
-> WHERE COMPRESSED_SIZE <> 0;
+------------+-------+-------------+-----------------+
| TABLE_NAME | SPACE | PAGE_NUMBER | COMPRESSED_SIZE |
+------------+-------+-------------+-----------------+
| sbtest/t   |     9 |         134 |            8192 |
| sbtest/t   |     9 |         135 |            8192 |
| sbtest/t   |     9 |          96 |            8192 |
| sbtest/t   |     9 |         136 |            8192 |
| sbtest/t   |     9 |          32 |            8192 |
| sbtest/t   |     9 |          97 |            8192 |
| sbtest/t   |     9 |         137 |            8192 |
| sbtest/t   |     9 |          98 |            8192 |
 ......
```

在 LRU 列表中的页被修改后，称该页为脏页（dirty page），即缓冲池中的页和磁盘上的页的数据产生了不一致。这时数据库会通过 CHECKPOINT 机制将脏页刷新回磁盘，而 Flush 列表中的页即为脏页列表。需要注意的是，脏页既存在于 LRU 列表中，也存在于 Flush 列表中。LRU 列表用来管理缓冲池中页的可用性，Flush 列表用来管理将页刷新回磁盘，二者互不影响。

同 LRU 列表一样，Flush 列表也可以通过命令 SHOW ENGINE INNODB STATUS 来查看，前面例子中 Modified db pages 24673 就显示了脏页的数量。information_schema 架构下并没有类似 INNODB_BUFFER_PAGE_LRU 的表来显示脏页的数量及脏页的类型，但正如前面所述的那样，脏页同样存在于 LRU 列表中，故用户可以通过元数据表 INNODB_BUFFER_PAGE_LRU 来查看，唯一不同的是需要加入 OLDEST_MODIFICATION 大于 0 的 SQL 查询条件，如：

```
mysql> SELECT TABLE_NAME,SPACE,PAGE_NUMBER,PAGE_TYPE
    -> FROM INNODB_BUFFER_PAGE_LRU
    -> WHERE OLDEST_MODIFICATION> 0;
+------------+-------+-------------+--------------------+
| TABLE_NAME | SPACE | PAGE_NUMBER | PAGE_TYPE          |
+------------+-------+-------------+--------------------+
| NULL       |     0 |          56 | SYSTEM             |
| NULL       |     0 |           0 | FILE_SPACE_HEADER  |
| test/t     |     1 |           3 | INDEX              |
| NULL       |     0 |         320 | INODE              |
```

```
| NULL        |    0  |       325   | UNDO_LOG           |
+-------------+-------+-------------+--------------------+
5 rows in set (0.00 sec)
```

可以看到当前共有 5 个脏页及它们对应的表和页的类型。TABLE_NAME 为 NULL 表示该页属于系统表空间。

3. 重做日志缓冲

从图 2-2 可以看到，InnoDB 存储引擎的内存区域除了有缓冲池外，还有重做日志缓冲（redo log buffer）。InnoDB 存储引擎首先将重做日志信息先放入到这个缓冲区，然后按一定频率将其刷新到重做日志文件。重做日志缓冲一般不需要设置得很大，因为一般情况下每一秒钟会将重做日志缓冲刷新到日志文件，因此用户只需要保证每秒产生的事务量在这个缓冲大小之内即可。该值可由配置参数 innodb_log_buffer_size 控制，默认为 8MB：

```
mysql> SHOW VARIABLES LIKE 'innodb_log_buffer_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_log_buffer_size
       Value: 8388608
1 row in set (0.00 sec)
```

在通常情况下，8MB 的重做日志缓冲池足以满足绝大部分的应用，因为重做日志在下列三种情况下会将重做日志缓冲中的内容刷新到外部磁盘的重做日志文件中。

- Master Thread 每一秒将重做日志缓冲刷新到重做日志文件；
- 每个事务提交时会将重做日志缓冲刷新到重做日志文件；
- 当重做日志缓冲池剩余空间小于 1/2 时，重做日志缓冲刷新到重做日志文件。

4. 额外的内存池

额外的内存池通常被 DBA 忽略，他们认为该值并不十分重要，事实恰恰相反，该值同样十分重要。在 InnoDB 存储引擎中，对内存的管理是通过一种称为内存堆（heap）的方式进行的。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。例如，分配了缓冲池（innodb_buffer_pool），但是每个缓冲池中的帧缓冲（frame buffer）还有对应的缓冲控制对象（buffer control block），这些对象记录了一些诸如 LRU、锁、等待等信息，而这个对象的内存需要从额外内存池中申请。因此，在申请了很大的 InnoDB 缓冲池时，也应考虑相

应地增加这个值。

2.4 Checkpoint 技术

前面已经讲到了，缓冲池的设计目的为了协调 CPU 速度与磁盘速度的鸿沟。因此页的操作首先都是在缓冲池中完成的。如果一条 DML 语句，如 Update 或 Delete 改变了页中的记录，那么此时页是脏的，即缓冲池中的页的版本要比磁盘的新。数据库需要将新版本的页从缓冲池刷新到磁盘。

倘若每次一个页发生变化，就将新页的版本刷新到磁盘，那么这个开销是非常大的。若热点数据集中在某几个页中，那么数据库的性能将变得非常差。同时，如果在从缓冲池将页的新版本刷新到磁盘时发生了宕机，那么数据就不能恢复了。为了避免发生数据丢失的问题，当前事务数据库系统普遍都采用了 Write Ahead Log 策略，即当事务提交时，先写重做日志，再修改页。当由于发生宕机而导致数据丢失时，通过重做日志来完成数据的恢复。这也是事务 ACID 中 D（Durability 持久性）的要求。

思考下面的场景，如果重做日志可以无限地增大，同时缓冲池也足够大，能够缓冲所有数据库的数据，那么是不需要将缓冲池中页的新版本刷新回磁盘。因为当发生宕机时，完全可以通过重做日志来恢复整个数据库系统中的数据到宕机发生的时刻。但是这需要两个前提条件：

- 缓冲池可以缓存数据库中所有的数据；
- 重做日志可以无限增大。

对于第一个前提条件，有经验的用户都知道，当数据库刚开始创建时，表中没有任何数据。缓冲池的确可以缓存所有的数据库文件。然而随着市场的推广，用户的增加，产品越来越受到关注，使用量也越来越大。这时负责后台存储的数据库的容量必定会不断增大。当前 3TB 的 MySQL 数据库已并不少见，但是 3 TB 的内存却非常少见。目前 Oracle Exadata 旗舰数据库一体机也就只有 2 TB 的内存。因此第一个假设对于生产环境应用中的数据库是很难得到保证的。

再来看第二个前提条件 ：重做日志可以无限增大。也许是可以的，但是这对成本的要求太高，同时不便于运维。DBA 或 SA 不能知道什么时候重做日志是否已经接近于磁盘可使用空间的阈值，并且要让存储设备支持可动态扩展也是需要一定的技巧和设备支

持的。

好的，即使上述两个条件都满足，那么还有一个情况需要考虑：宕机后数据库的恢复时间。当数据库运行了几个月甚至几年时，这时发生宕机，重新应用重做日志的时间会非常久，此时恢复的代价也会非常大。

因此 Checkpoint（检查点）技术的目的是解决以下几个问题：

- 缩短数据库的恢复时间；
- 缓冲池不够用时，将脏页刷新到磁盘；
- 重做日志不可用时，刷新脏页。

当数据库发生宕机时，数据库不需要重做所有的日志，因为 Checkpoint 之前的页都已经刷新回磁盘。故数据库只需对 Checkpoint 后的重做日志进行恢复。这样就大大缩短了恢复的时间。

此外，当缓冲池不够用时，根据 LRU 算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行 Checkpoint，将脏页也就是页的新版本刷回磁盘。

重做日志出现不可用的情况是因为当前事务数据库系统对重做日志的设计都是循环使用的，并不是让其无限增大的，这从成本及管理上都是比较困难的。重做日志可以被重用的部分是指这些重做日志已经不再需要，即当数据库发生宕机时，数据库恢复操作不需要这部分的重做日志，因此这部分就可以被覆盖重用。若此时重做日志还需要使用，那么必须强制产生 Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置。

对于 InnoDB 存储引擎而言，其是通过 LSN（Log Sequence Number）来标记版本的。而 LSN 是 8 字节的数字，其单位是字节。每个页有 LSN，重做日志中也有 LSN，Checkpoint 也有 LSN。可以通过命令 SHOW ENGINE INNODB STATUS 来观察：

```
mysql> SHOW ENGINE INNODB STATUS\G;
.....
---
LOG
---
Log sequence number 92561351052
Log flushed up to   92561351052
Last checkpoint at  92561351052
.....
```

在 InnoDB 存储引擎中，Checkpoint 发生的时间、条件及脏页的选择等都非常复杂。

而 Checkpoint 所做的事情无外乎是将缓冲池中的脏页刷回到磁盘。不同之处在于每次刷新多少页到磁盘，每次从哪里取脏页，以及什么时间触发 Checkpoint。在 InnoDB 存储引擎内部，有两种 Checkpoint，分别为：

- Sharp Checkpoint
- Fuzzy Checkpoint

Sharp Checkpoint 发生在数据库关闭时将所有的脏页都刷新回磁盘，这是默认的工作方式，即参数 innodb_fast_shutdown=1。

但是若数据库在运行时也使用 Sharp Checkpoint，那么数据库的可用性就会受到很大的影响。故在 InnoDB 存储引擎内部使用 Fuzzy Checkpoint 进行页的刷新，即只刷新一部分脏页，而不是刷新所有的脏页回磁盘。

这里笔者进行了概括，在 InnoDB 存储引擎中可能发生如下几种情况的 Fuzzy Checkpoint：

- Master Thread Checkpoint
- FLUSH_LRU_LIST Checkpoint
- Async/Sync Flush Checkpoint
- Dirty Page too much Checkpoint

对于 Master Thread（2.5 节会详细介绍各个版本中 Master Thread 的实现）中发生的 Checkpoint，差不多以每秒或每十秒的速度从缓冲池的脏页列表中刷新一定比例的页回磁盘。这个过程是异步的，即此时 InnoDB 存储引擎可以进行其他的操作，用户查询线程不会阻塞。

FLUSH_LRU_LIST Checkpoint 是因为 InnoDB 存储引擎需要保证 LRU 列表中需要有差不多 100 个空闲页可供使用。在 InnoDB1.1.x 版本之前，需要检查 LRU 列表中是否有足够的可用空间操作发生在用户查询线程中，显然这会阻塞用户的查询操作。倘若没有 100 个可用空闲页，那么 InnoDB 存储引擎会将 LRU 列表尾端的页移除。如果这些页中有脏页，那么需要进行 Checkpoint，而这些页是来自 LRU 列表的，因此称为 FLUSH_LRU_LIST Checkpoint。

而从 MySQL 5.6 版本，也就是 InnoDB1.2.x 版本开始，这个检查被放在了一个单独的 Page Cleaner 线程中进行，并且用户可以通过参数 innodb_lru_scan_depth 控制 LRU 列表中可用页的数量，该值默认为 1024，如：

```
mysql> SHOW VARIABLES LIKE 'innodb_lru_scan_depth'\G;
*************************** 1. row ***************************
Variable_name: innodb_lru_scan_depth
       Value: 1024
1 row in set (0.00 sec)
```

Async/Sync Flush Checkpoint 指的是重做日志文件不可用的情况，这时需要强制将一些页刷新回磁盘，而此时脏页是从脏页列表中选取的。若将已经写入到重做日志的 LSN 记为 redo_lsn，将已经刷新回磁盘最新页的 LSN 记为 checkpoint_lsn，则可定义：

```
checkpoint_age = redo_lsn - checkpoint_lsn
```

再定义以下的变量：

```
async_water_mark = 75% * total_redo_log_file_size
sync_water_mark = 90% * total_redo_log_file_size
```

若每个重做日志文件的大小为 1GB，并且定义了两个重做日志文件，则重做日志文件的总大小为 2GB。那么 async_water_mark=1.5GB，sync_water_mark=1.8GB。则：

- 当 checkpoint_age<async_water_mark 时，不需要刷新任何脏页到磁盘；
- 当 async_water_mark<checkpoint_age<sync_water_mark 时触发 Async Flush，从 Flush 列表中刷新足够的脏页回磁盘，使得刷新后满足 checkpoint_age<async_water_mark；
- checkpoint_age>sync_water_mark 这种情况一般很少发生，除非设置的重做日志文件太小，并且在进行类似 LOAD DATA 的 BULK INSERT 操作。此时触发 Sync Flush 操作，从 Flush 列表中刷新足够的脏页回磁盘，使得刷新后满足 checkpoint_age<async_water_mark。

可见，Async/Sync Flush Checkpoint 是为了保证重做日志的循环使用的可用性。在 InnoDB 1.2.x 版本之前，Async Flush Checkpoint 会阻塞发现问题的用户查询线程，而 Sync Flush Checkpoint 会阻塞所有的用户查询线程，并且等待脏页刷新完成。从 InnoDB 1.2.x 版本开始——也就是 MySQL 5.6 版本，这部分的刷新操作同样放入到了单独的 Page Cleaner Thread 中，故不会阻塞用户查询线程。

MySQL 官方版本并不能查看刷新页是从 Flush 列表中还是从 LRU 列表中进行 Checkpoint 的，也不知道因为重做日志而产生的 Async/Sync Flush 的次数。但是 InnoSQL 版本提供了方法，可以通过命令 SHOW ENGINE INNODB STATUS 来观察，如：

```
mysql> SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
……
LRU len: 112902, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
Async Flush: 0, Sync Flush: 0, LRU List Flush: 0, Flush List Flush: 111736
……
1 row in set (0.01 sec)
```

根据上述的信息，还可以对 InnoDB 存储引擎做更为深入的调优，这部分将在第 9 章中讲述。

最后一种 Checkpoint 的情况是 Dirty Page too much，即脏页的数量太多，导致 InnoDB 存储引擎强制进行 Checkpoint。其目的总的来说还是为了保证缓冲池中有足够可用的页。其可由参数 innodb_max_dirty_pages_pct 控制：

```
mysql>SHOW VARIABLES LIKE 'innodb_max_dirty_pages_pct'\G;
*************************** 1. row ***************************
Variable_name: innodb_max_dirty_pages_pct
       Value: 75
1 row in set (0.00 sec)
```

innodb_max_dirty_pages_pct 值为 75 表示，当缓冲池中脏页的数量占据 75% 时，强制进行 Checkpoint，刷新一部分的脏页到磁盘。在 InnoDB 1.0.x 版本之前，该参数默认值为 90，之后的版本都为 75。

2.5 Master Thread 工作方式

在 2.3 节中我们知道了，InnoDB 存储引擎的主要工作都是在一个单独的后台线程 Master Thread 中完成的，这一节将具体解释该线程的具体实现及该线程可能存在的问题。

2.5.1 InnoDB 1.0.x 版本之前的 Master Thread

Master Thread 具有最高的线程优先级别。其内部由多个循环（loop）组成：主循环（loop）、后台循环（backgroup loop）、刷新循环（flush loop）、暂停循环（suspend loop）。Master Thread 会根据数据库运行的状态在 loop、background loop、flush loop 和 suspend

loop 中进行切换。

Loop 被称为主循环，因为大多数的操作是在这个循环中，其中有两大部分的操作——每秒钟的操作和每 10 秒的操作。伪代码如下：

```
void master_thread(){
loop:
for(int i= 0; i<10; i++){
   do thing once per second
   sleep 1 second if necessary
}
do things once per ten seconds
goto loop;
}
```

可以看到，loop 循环通过 thread sleep 来实现，这意味着所谓的每秒一次或每 10 秒一次的操作是不精确的。在负载很大的情况下可能会有延迟（delay），只能说大概在这个频率下。当然，InnoDB 源代码中还通过了其他的方法来尽量保证这个频率。

每秒一次的操作包括：

- 日志缓冲刷新到磁盘，即使这个事务还没有提交（总是）；
- 合并插入缓冲（可能）；
- 至多刷新 100 个 InnoDB 的缓冲池中的脏页到磁盘（可能）；
- 如果当前没有用户活动，则切换到 background loop（可能）。

即使某个事务还没有提交，InnoDB 存储引擎仍然每秒会将重做日志缓冲中的内容刷新到重做日志文件。这一点是必须要知道的，因为这可以很好地解释为什么再大的事务提交（commit）的时间也是很短的。

合并插入缓冲（Insert Buffer）并不是每秒都会发生的。InnoDB 存储引擎会判断当前一秒内发生的 IO 次数是否小于 5 次，如果小于 5 次，InnoDB 认为当前的 IO 压力很小，可以执行合并插入缓冲的操作。

同样，刷新 100 个脏页也不是每秒都会发生的。InnoDB 存储引擎通过判断当前缓冲池中脏页的比例（buf_get_modified_ratio_pct）是否超过了配置文件中 innodb_max_dirty_pages_pct 这个参数（默认为 90，代表 90%），如果超过了这个阈值，InnoDB 存储引擎认为需要做磁盘同步的操作，将 100 个脏页写入磁盘中。

总结上述操作，伪代码可以进一步具体化，如下所示：

```
void master_thread(){
   goto loop;
loop:
for(int i = 0; i<10; i++){
   thread_sleep(1) // sleep 1 second
   do log buffer flush to disk
   if (last_one_second_ios < 5 )
      do merge at most 5 insert buffer
   if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
      do buffer pool flush 100 dirty page
   if ( no user activity )
      goto backgroud loop
}
do things once per ten seconds
background loop:
   do something
   goto loop:
}
```

接着来看每 10 秒的操作，包括如下内容：

- 刷新 100 个脏页到磁盘（可能的情况下）；
- 合并至多 5 个插入缓冲（总是）；
- 将日志缓冲刷新到磁盘（总是）；
- 删除无用的 Undo 页（总是）；
- 刷新 100 个或者 10 个脏页到磁盘（总是）。

在以上的过程中，InnoDB 存储引擎会先判断过去 10 秒之内磁盘的 IO 操作是否小于 200 次，如果是，InnoDB 存储引擎认为当前有足够的磁盘 IO 操作能力，因此将 100 个脏页刷新到磁盘。接着，InnoDB 存储引擎会合并插入缓冲。不同于每秒一次操作时可能发生的合并插入缓冲操作，这次的合并插入缓冲操作总会在这个阶段进行。之后，InnoDB 存储引擎会再进行一次将日志缓冲刷新到磁盘的操作。这和每秒一次时发生的操作是一样的。

接着 InnoDB 存储引擎会进行一步执行 full purge 操作，即删除无用的 Undo 页。对表进行 update、delete 这类操作时，原先的行被标记为删除，但是因为一致性读（consistent read）的关系，需要保留这些行版本的信息。但是在 full purge 过程中，InnoDB 存储引擎会判断当前事务系统中已被删除的行是否可以删除，比如有时候可能还有查询操作需要读取之前版本的 undo 信息，如果可以删除，InnoDB 会立即将其删除。

从源代码中可以发现，InnoDB 存储引擎在执行 full purge 操作时，每次最多尝试回收 20 个 undo 页。

然后，InnoDB 存储引擎会判断缓冲池中脏页的比例（buf_get_modified_ratio_pct），如果有超过 70% 的脏页，则刷新 100 个脏页到磁盘，如果脏页的比例小于 70%，则只需刷新 10% 的脏页到磁盘。

现在我们可以完整地把主循环（main loop）的伪代码写出来了，内容如下：

```
void master_thread(){
   goto loop;
loop:
for(int i = 0; i<10; i++){
   thread_sleep(1) // sleep 1 second
   do log buffer flush to disk
   if (last_one_second_ios < 5 )
      do merge at most 5 insert buffer
   if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
      do buffer pool flush 100 dirty page
   if ( no user activity )
      goto backgroud loop
}
if ( last_ten_second_ios < 200 )
   do buffer pool flush 100 dirty page
do merge at most 5 insert buffer
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
   do buffer pool flush 100 dirty page
else
   buffer pool flush 10 dirty page
goto loop
background loop:
   do something
goto loop:
}
```

接着来看 background loop，若当前没有用户活动（数据库空闲时）或者数据库关闭（shutdown），就会切换到这个循环。background loop 会执行以下操作：

- 删除无用的 Undo 页（总是）；
- 合并 20 个插入缓冲（总是）；
- 跳回到主循环（总是）；

❑ 不断刷新 100 个页直到符合条件（可能，跳转到 flush loop 中完成）。

若 flush loop 中也没有什么事情可以做了，InnoDB 存储引擎会切换到 suspend_loop，将 Master Thread 挂起，等待事件的发生。若用户启用（enable）了 InnoDB 存储引擎，却没有使用任何 InnoDB 存储引擎的表，那么 Master Thread 总是处于挂起的状态。

最后，Master Thread 完整的伪代码如下：

```
void master_thread(){
   goto loop;
loop:
for(int i = 0; i<10; i++){
   thread_sleep(1) // sleep 1 second
   do log buffer flush to disk
   if ( last_one_second_ios < 5 )
      do merge at most 5 insert buffer
   if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
      do buffer pool flush 100 dirty page
   if ( no user activity )
      goto backgroud loop
}
if ( last_ten_second_ios < 200 )
   do buffer pool flush 100 dirty page
do merge at most 5 insert buffer
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
   do buffer pool flush 100 dirty page
else
   buffer pool flush 10 dirty page
goto loop
background loop:
do full purge
do merge 20 insert buffer
if not idle:
goto loop:
else:
   goto flush loop
flush loop:
do buffer pool flush 100 dirty page
if ( buf_get_modified_ratio_pct>innodb_max_dirty_pages_pct )
   goto flush loop
goto suspend loop
```

```
suspend loop:
suspend_thread()
waiting event
goto loop;
}
```

2.5.2 InnoDB1.2.x 版本之前的 Master Thread

在了解了 1.0.x 版本之前的 Master Thread 的具体实现过程后，细心的读者会发现 InnoDB 存储引擎对于 IO 其实是有限制的，在缓冲池向磁盘刷新时其实都做了一定的硬编码（hard coding）。在磁盘技术飞速发展的今天，当固态磁盘（SSD）出现时，这种规定在很大程度上限制了 InnoDB 存储引擎对磁盘 IO 的性能，尤其是写入性能。

从前面的伪代码来看，无论何时，InnoDB 存储引擎最大只会刷新 100 个脏页到磁盘，合并 20 个插入缓冲。如果是在写入密集的应用程序中，每秒可能会产生大于 100 个的脏页，如果是产生大于 20 个插入缓冲的情况，Master Thread 似乎会“忙不过来”，或者说它总是做得很慢。即使磁盘能在 1 秒内处理多于 100 个页的写入和 20 个插入缓冲的合并，但是由于 hard coding，Master Thread 也只会选择刷新 100 个脏页和合并 20 个插入缓冲。同时，当发生宕机需要恢复时，由于很多数据还没有刷新回磁盘，会导致恢复的时间可能需要很久，尤其是对于 insert buffer 来说。

这个问题最初由 Google 的工程师 Mark Callaghan 提出，之后 InnoDB 官方对其进行了修正并发布了补丁（patch）。InnoDB 存储引擎的开发团队参考了 Google 的 patch，提供了类似的方法来修正该问题。因此 InnoDB Plugin（从 InnoDB1.0.x 版本开始）提供了参数 innodb_io_capacity，用来表示磁盘 IO 的吞吐量，默认值为 200。对于刷新到磁盘页的数量，会按照 innodb_io_capacity 的百分比来进行控制。规则如下：

- 在合并插入缓冲时，合并插入缓冲的数量为 innodb_io_capacity 值的 5%；
- 在从缓冲区刷新脏页时，刷新脏页的数量为 innodb_io_capacity。

若用户使用了 SSD 类的磁盘，或者将几块磁盘做了 RAID，当存储设备拥有更高的 IO 速度时，完全可以将 innodb_io_capacity 的值调得再高点，直到符合磁盘 IO 的吞吐量为止。

另一个问题是，参数 innodb_max_dirty_pages_pct 默认值的问题，在 InnoDB 1.0.x 版本之前，该值的默认为 90，意味着脏页占缓冲池的 90%。但是该值“太大”了，因

为 InnoDB 存储引擎在每秒刷新缓冲池和 flush loop 时会判断这个值，如果该值大于 innodb_max_dirty_pages_pct，才刷新 100 个脏页，如果有很大的内存，或者数据库服务器的压力很大，这时刷新脏页的速度反而会降低。同样，在数据库的恢复阶段可能需要更多的时间。

在很多论坛上都有对这个问题的讨论，有人甚至将这个值调到了 20 或 10，然后测试发现性能会有所提高，但是将 innodb_max_dirty_pages_pct 调到 20 或 10 会增加磁盘的压力，系统的负担还是会有所增加的。Google 在这个问题上进行了测试，证明 20 并不是一个最优值[㊀]。而从 InnoDB 1.0.x 版本开始，innodb_max_dirty_pages_pct 默认值变为了 75，和 Google 测试的 80 比较接近。这样既可以加快刷新脏页的频率，又能保证了磁盘 IO 的负载。

InnoDB 1.0.x 版本带来的另一个参数是 innodb_adaptive_flushing（自适应地刷新），该值影响每秒刷新脏页的数量。原来的刷新规则是：脏页在缓冲池所占的比例小于 innodb_max_dirty_pages_pct 时，不刷新脏页；大于 innodb_max_dirty_pages_pct 时，刷新 100 个脏页。随着 innodb_adaptive_flushing 参数的引入，InnoDB 存储引擎会通过一个名为 buf_flush_get_desired_flush_rate 的函数来判断需要刷新脏页最合适的数量。粗略地翻阅源代码后发现 buf_flush_get_desired_flush_rate 通过判断产生重做日志（redo log）的速度来决定最合适的刷新脏页数量。因此，当脏页的比例小于 innodb_max_dirty_pages_pct 时，也会刷新一定量的脏页。

还有一个改变是：之前每次进行 full purge 操作时，最多回收 20 个 Undo 页，从 InnoDB 1.0.x 版本开始引入了参数 innodb_purge_batch_size，该参数可以控制每次 full purge 回收的 Undo 页的数量。该参数的默认值为 20，并可以动态地对其进行修改，具体如下：

```
mysql> SHOW VARIABLES LIKE 'innodb_purge_batch_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_purge_batch_size
        Value: 20

mysql> SET GLOBAL innodb_purge_batch_size=50;
Query OK, 0 rows affected (0.00 sec)
```

㊀ 有兴趣的读者可参考：http://code.google.com/p/google-mysql-tools/wiki/InnodbIoOltpDisk。

通过上述的讨论和解释我们知道，从 InnoDB 1.0.x 版本开始，Master Thread 的伪代码必将有所改变，最终变成：

```
void master_thread(){
   goto loop;
loop:
for(int i = 0; i<10; i++){
   thread_sleep(1) // sleep 1 second
   do log buffer flush to disk
   if ( last_one_second_ios < 5% innodb_io_capacity )
      do merge 5% innodb_io_capacity insert buffer
   if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
      do buffer pool flush 100% innodb_io_capacity dirty page
   else if enable adaptive flush
      do buffer pool flush desired amount dirty page
   if ( no user activity )
      goto backgroud loop
}
if ( last_ten_second_ios <innodb_io_capacity)
   do buffer pool flush 100% innodb_io_capacity dirty page
do merge 5% innodb_io_capacity insert buffer
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
   do buffer pool flush 100% innodb_io_capacity dirty page
else
   dobuffer pool flush 10% innodb_io_capacity dirty page
goto loop
background loop:
do full purge
do merge 100% innodb_io_capacity insert buffer
if not idle:
goto loop:
else:
   goto flush loop
flush loop:
do buffer pool flush 100% innodb_io_capacity dirty page
if ( buf_get_modified_ratio_pct>innodb_max_dirty_pages_pct )
   go to flush loop
   goto suspend loop
suspend loop:
suspend_thread()
waiting event
goto loop;
}
```

很多测试都显示，InnoDB 1.0.x 版本在性能方面取得了极大的提高，其实这和前面提到的 Master Thread 的改动是密不可分的，因为 InnoDB 存储引擎的核心操作大部分都集中在 Master Thread 后台线程中。

从 InnoDB 1.0.x 开始，命令 SHOW ENGINE INNODB STATUS 可以查看当前 Master Thread 的状态信息，如下所示：

```
mysql>SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
090921 14:24:56 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 6 seconds
----------
BACKGROUND THREAD
----------
srv_master_thread loops: 45 1_second, 45 sleeps, 4 10_second, 6 background, 6 flush
srv_master_thread log flush and writes: 45  log writes only: 69
……
```

这里可以看到主循环进行了 45 次，每秒挂起（sleep）的操作进行了 45 次（说明负载不是很大），10 秒一次的活动进行了 4 次，符合 1∶10。background loop 进行了 6 次，flush loop 也进行了 6 次。因为当前这台服务器的压力很小，所以能在理论值上运行。如果是在一台压力很大的 MySQL 数据库服务器上，看到的可能会是下面的情景：

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
091009 10:14:34 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 42 seconds
----------
BACKGROUND THREAD
----------
srv_master_thread loops: 2188 1_second, 1537 sleeps, 218 10_second, 2
background, 2 flush
```

```
srv_master_thread log flush and writes: 1777  log writes only: 5816
……
```

可以看到当前主循环运行了 2188 次，但是循环中的每秒挂起（sleep）的操作只运行了 1537 次。这是因为 InnoDB 对其内部进行了一些优化，当压力大时并不总是等待 1 秒。因此，并不能认为 1_second 和 sleeps 的值总是相等的。在某些情况下，可以通过两者之间差值的比较来反映当前数据库的负载压力。

2.5.3 InnoDB 1.2.x 版本的 Master Thread

在 InnoDB 1.2.x 版本中再次对 Master Thread 进行了优化，由此也可以看出 Master Thread 对性能所起到的关键作用。在 InnoDB 1.2.x 版本中，Master Thread 的伪代码如下：

```
if InnoDB is idle
  srv_master_do_idle_tasks();
else
  srv_master_do_active_tasks();
```

其中 srv_master_do_idle_tasks() 就是之前版本中每 10 秒的操作，srv_master_do_active_tasks() 处理的是之前每秒中的操作。同时对于刷新脏页的操作，从 Master Thread 线程分离到一个单独的 Page Cleaner Thread，从而减轻了 Master Thread 的工作，同时进一步提高了系统的并发性。

2.6 InnoDB 关键特性

InnoDB 存储引擎的关键特性包括：

- 插入缓冲（Insert Buffer）
- 两次写（Double Write）
- 自适应哈希索引（Adaptive Hash Index）
- 异步 IO（Async IO）
- 刷新邻接页（Flush Neighbor Page）

上述这些特性为 InnoDB 存储引擎带来更好的性能以及更高的可靠性。

2.6.1 插入缓冲

1. Insert Buffer

Insert Buffer 可能是 InnoDB 存储引擎关键特性中最令人激动与兴奋的一个功能。不过这个名字可能会让人认为插入缓冲是缓冲池中的一个组成部分。其实不然，InnoDB 缓冲池中有 Insert Buffer 信息固然不错，但是 Insert Buffer 和数据页一样，也是物理页的一个组成部分。

在 InnoDB 存储引擎中，主键是行唯一的标识符。通常应用程序中行记录的插入顺序是按照主键递增的顺序进行插入的。因此，插入聚集索引（Primary Key）一般是顺序的，不需要磁盘的随机读取。比如按下列 SQL 定义表：

```
CREATE TABLE t (
    a INT AUTO_INCREMENT,
    b VARCHAR(30),
    PRIMARY KEY(a)
);
```

其中 a 列是自增长的，若对 a 列插入 NULL 值，则由于其具有 AUTO_INCREMENT 属性，其值会自动增长。同时页中的行记录按 a 的值进行顺序存放。在一般情况下，不需要随机读取另一个页中的记录。因此，对于这类情况下的插入操作，速度是非常快的。

注意 并不是所有的主键插入都是顺序的。若主键类是 UUID 这样的类，那么插入和辅助索引一样，同样是随机的。即使主键是自增类型，但是插入的是指定的值，而不是 NULL 值，那么同样可能导致插入并非连续的情况。

但是不可能每张表上只有一个聚集索引，更多情况下，一张表上有多个非聚集的辅助索引（secondary index）。比如，用户需要按照 b 这个字段进行查找，并且 b 这个字段不是唯一的，即表是按如下的 SQL 语句定义的：

```
CREATE TABLE t (
    a INT AUTO_INCREMENT,
    b VARCHAR(30),
    PRIMARY KEY(a),
    key(b)
);
```

在这样的情况下产生了一个非聚集的且不是唯一的索引。在进行插入操作时，数据

页的存放还是按主键 a 进行顺序存放的，但是对于非聚集索引叶子节点的插入不再是顺序的了，这时就需要离散地访问非聚集索引页，由于随机读取的存在而导致了插入操作性能下降。当然这并不是这个 b 字段上索引的错误，而是因为 B+ 树的特性决定了非聚集索引插入的离散性。

需要注意的是，在某些情况下，辅助索引的插入依然是顺序的，或者说是比较顺序的，比如用户购买表中的时间字段。在通常情况下，用户购买时间是一个辅助索引，用来根据时间条件进行查询。但是在插入时却是根据时间的递增而插入的，因此插入也是“较为”顺序的。

InnoDB 存储引擎开创性地设计了 Insert Buffer，对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个 Insert Buffer 对象中，好似欺骗。数据库这个非聚集的索引已经插到叶子节点，而实际并没有，只是存放在另一个位置。然后再以一定的频率和情况进行 Insert Buffer 和辅助索引页子节点的 merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。

然而 Insert Buffer 的使用需要同时满足以下两个条件：

- 索引是辅助索引（secondary index）；
- 索引不是唯一（unique）的。

当满足以上两个条件时，InnoDB 存储引擎会使用 Insert Buffer，这样就能提高插入操作的性能了。不过考虑这样一种情况：应用程序进行大量的插入操作，这些都涉及了不唯一的非聚集索引，也就是使用了 Insert Buffer。若此时 MySQL 数据库发生了宕机，这时势必有大量的 Insert Buffer 并没有合并到实际的非聚集索引中去。因此这时恢复可能需要很长的时间，在极端情况下甚至需要几个小时。

辅助索引不能是唯一的，因为在插入缓冲时，数据库并不去查找索引页来判断插入的记录的唯一性。如果去查找肯定又会有离散读取的情况发生，从而导致 Insert Buffer 失去了意义。

用户可以通过命令 SHOW ENGINE INNODB STATUS 来查看插入缓冲的信息：

```
mysql>SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
```

```
  Type: InnoDB
  Name:
Status:
=====================================
100727 22:21:48 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 44 seconds
……
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 7545, free list len 3790, seg size 11336,
8075308 inserts, 7540969 merged recs, 2246304 merges
……
-----------------------------
END OF INNODB MONITOR OUTPUT
=============================


1 row in set (0.00 sec)
```

seg size 显示了当前 Insert Buffer 的大小为 11336×16KB，大约为 177MB ； free list len 代表了空闲列表的长度 ； size 代表了已经合并记录页的数量。而黑体部分的第 2 行可能是用户真正关心的，因为它显示了插入性能的提高。Inserts 代表了插入的记录数 ；merged recs 代表了合并的插入记录数量 ； merges 代表合并的次数，也就是实际读取页的次数。merges:merged recs 大约为 1 : 3，代表了插入缓冲将对于非聚集索引页的离散 IO 逻辑请求大约降低了 2/3。

正如前面所说的，目前 Insert Buffer 存在一个问题是 ：在写密集的情况下，插入缓冲会占用过多的缓冲池内存（innodb_buffer_pool），默认最大可以占用到 1/2 的缓冲池内存。以下是 InnoDB 存储引擎源代码中对于 insert buffer 的初始化操作：

```
/** Buffer pool size per the maximum insert buffer size */
#define IBUF_POOL_SIZE_PER_MAX_SIZE       2
ibuf->max_size = buf_pool_get_curr_size() / UNIV_PAGE_SIZE
            / IBUF_POOL_SIZE_PER_MAX_SIZE;
```

这对于其他的操作可能会带来一定的影响。Percona 上发布一些 patch 来修正插入缓冲占用太多缓冲池内存的情况，具体可以到 Percona 官网进行查找。简单来说，修改 IBUF_POOL_SIZE_PER_MAX_SIZE 就可以对插入缓冲的大小进行控制。比如将 IBUF_

POOL_SIZE_PER_MAX_SIZE 改为 3，则最大只能使用 1/3 的缓冲池内存。

2. Change Buffer

InnoDB 从 1.0.x 版本开始引入了 Change Buffer，可将其视为 Insert Buffer 的升级。从这个版本开始，InnoDB 存储引擎可以对 DML 操作——INSERT、DELETE、UPDATE 都进行缓冲，他们分别是：Insert Buffer、Delete Buffer、Purge buffer。

当然和之前 Insert Buffer 一样，Change Buffer 适用的对象依然是非唯一的辅助索引。

对一条记录进行 UPDATE 操作可能分为两个过程：

- 将记录标记为已删除；
- 真正将记录删除。

因此 Delete Buffer 对应 UPDATE 操作的第一个过程，即将记录标记为删除。Purge Buffer 对应 UPDATE 操作的第二个过程，即将记录真正的删除。同时，InnoDB 存储引擎提供了参数 innodb_change_buffering，用来开启各种 Buffer 的选项。该参数可选的值为：inserts、deletes、purges、changes、all、none。inserts、deletes、purges 就是前面讨论过的三种情况。changes 表示启用 inserts 和 deletes，all 表示启用所有，none 表示都不启用。该参数默认值为 all。

从 InnoDB 1.2.x 版本开始，可以通过参数 innodb_change_buffer_max_size 来控制 Change Buffer 最大使用内存的数量：

```
mysql> SHOW VARIABLES LIKE 'innodb_change_buffer_max_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_change_buffer_max_size
        Value: 25
1 row in set (0.00 sec)
```

innodb_change_buffer_max_size 值默认为 25，表示最多使用 1/4 的缓冲池内存空间。而需要注意的是，该参数的最大有效值为 50。

在 MySQL 5.5 版本中通过命令 SHOW ENGINE INNODB STATUS，可以观察到类似如下的内容：

```
mysql> SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
  Type: InnoDB
……
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
```

```
-------------------------------------
Ibuf: size 1, free list len 34397, seg size 34399, 10875 merges
merged operations:
 insert 20462, delete mark 20158, delete 4215
discarded operations:
 insert 0, delete mark 0, delete 0
……
```

可以看到这里显示了 merged operations 和 discarded operation，并且下面具体显示 Change Buffer 中每个操作的次数。insert 表示 Insert Buffer；delete mark 表示 Delete Buffer；delete 表示 Purge Buffer；discarded operations 表示当 Change Buffer 发生 merge 时，表已经被删除，此时就无需再将记录合并（merge）到辅助索引中了。

3. Insert Buffer 的内部实现

通过前一个小节读者应该已经知道了 Insert Buffer 的使用场景，即非唯一辅助索引的插入操作。但是对于 Insert Buffer 具体是什么，以及内部怎么实现可能依然模糊，这正是本节所要阐述的内容。

可能令绝大部分用户感到吃惊的是，Insert Buffer 的数据结构是一棵 B+ 树。在 MySQL 4.1 之前的版本中每张表有一棵 Insert Buffer B+ 树。而在现在的版本中，全局只有一棵 Insert Buffer B+ 树，负责对所有的表的辅助索引进行 Insert Buffer。而这棵 B+ 树存放在共享表空间中，默认也就是 ibdata1 中。因此，试图通过独立表空间 ibd 文件恢复表中数据时，往往会导致 CHECK TABLE 失败。这是因为表的辅助索引中的数据可能还在 Insert Buffer 中，也就是共享表空间中，所以通过 ibd 文件进行恢复后，还需要进行 REPAIR TABLE 操作来重建表上所有的辅助索引。

Insert Buffer 是一棵 B+ 树，因此其也由叶节点和非叶节点组成。非叶节点存放的是查询的 search key（键值），其构造如图 2-3 所示。

图 2-3 Insert Buffer 非叶节点中的 search key

search key 一共占用 9 个字节，其中 space 表示待插入记录所在表的表空间 id，在 InnoDB 存储引擎中，每个表有一个唯一的 space id，可以通过 space id 查询得知是哪张表。space 占用 4 字节。marker 占用 1 字节，它是用来兼容老版本的 Insert Buffer。offset 表示页所在的偏移量，占用 4 字节。

当一个辅助索引要插入到页（space，offset）时，如果这个页不在缓冲池中，那么InnoDB 存储引擎首先根据上述规则构造一个 search key，接下来查询 Insert Buffer 这棵B+ 树，然后再将这条记录插入到 Insert Buffer B+ 树的叶子节点中。

对于插入到 Insert Buffer B+ 树叶子节点的记录（如图 2-4 所示），并不是直接将待插入的记录插入，而是需要根据如下的规则进行构造：

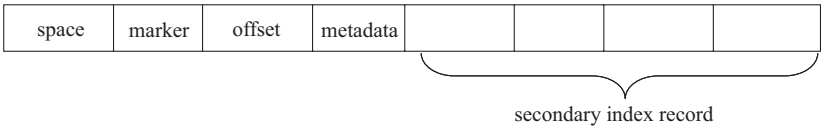


图 2-4 Insert Buffer 叶子节点中的记录

space、marker、page_no 字段和之前非叶节点中的含义相同，一共占用 9 字节。第4 个字段 metadata 占用 4 字节，其存储的内容如表 2-2 所示。

表 2-2 metadata 字段存储的内容

名 称	字 节
IBUF_REC_OFFSET_COUNT	2
IBUF_REC_OFFSET_TYPE	1
IBUF_REC_OFFSET_FLAGS	1

IBUF_REC_OFFSET_COUNT 是保存两个字节的整数，用来排序每个记录进入Insert Buffer 的顺序。因为从 InnoDB1.0.x 开始支持 Change Buffer，所以这个值同样记录进入 Insert Buffer 的顺序。通过这个顺序回放（replay）才能得到记录的正确值。

从 Insert Buffer 叶子节点的第 5 列开始，就是实际插入记录的各个字段了。因此较之原插入记录，Insert Buffer B+ 树的叶子节点记录需要额外 13 字节的开销。

因为启用 Insert Buffer 索引后，辅助索引页（space，page_no）中的记录可能被插入到 Insert Buffer B+ 树中，所以为了保证每次 Merge Insert Buffer 页必须成功，还需要有一个特殊的页用来标记每个辅助索引页（space，page_no）的可用空间。这个页的类型为 Insert Buffer Bitmap。

每个 Insert Buffer Bitmap 页用来追踪 16384 个辅助索引页，也就是 256 个区（Extent）。每个 Insert Buffer Bitmap 页都在 16384 个页的第二个页中。关于 Insert Buffer Bitmap 页的作用会在下一小节中详细介绍。

每个辅助索引页在 Insert Buffer Bitmap 页中占用 4 位（bit），由表 2-3 中的三个部分

组成。

表 2-3 每个辅助索引页在 Insert Buffer Bitmap 中存储的信息

名 称	大小（bit）	说 明
IBUF_BITMAP_FREE	2	表示该辅助索引页中的可用空间数量，可取值为： ❑ 0 表示无可用剩余空间 ❑ 1 表示剩余空间大于 1/32 页（512 字节） ❑ 2 表示剩余空间大于 1/16 页 ❑ 3 表示剩余空间大于 1/8 页
IBUF_BITMAP_BUFFERED	1	1 表示该辅助索引页有记录被缓存在 Insert Buffer B+ 树中
IBUF_BITMAP_IBUF	1	1 表示该页为 Insert Buffer B+ 树的索引页

4. Merge Insert Buffer

通过前面的小节读者应该已经知道了 Insert/Change Buffer 是一棵 B+ 树。若需要实现插入记录的辅助索引页不在缓冲池中，那么需要将辅助索引记录首先插入到这棵 B+ 树中。但是 Insert Buffer 中的记录何时合并（merge）到真正的辅助索引中呢？这是本小节需要关注的重点。

概括地说，Merge Insert Buffer 的操作可能发生在以下几种情况下：

❑ 辅助索引页被读取到缓冲池时；

❑ Insert Buffer Bitmap 页追踪到该辅助索引页已无可用空间时；

❑ Master Thread。

第一种情况为当辅助索引页被读取到缓冲池中时，例如这在执行正常的 SELECT 查询操作，这时需要检查 Insert Buffer Bitmap 页，然后确认该辅助索引页是否有记录存放于 Insert Buffer B+ 树中。若有，则将 Insert Buffer B+ 树中该页的记录插入到该辅助索引页中。可以看到对该页多次的记录操作通过一次操作合并到了原有的辅助索引页中，因此性能会有大幅提高。

Insert Buffer Bitmap 页用来追踪每个辅助索引页的可用空间，并至少有 1/32 页的空间。若插入辅助索引记录时检测到插入记录后可用空间会小于 1/32 页，则会强制进行一个合并操作，即强制读取辅助索引页，将 Insert Buffer B+ 树中该页的记录及待插入的记录插入到辅助索引页中。这就是上述所说的第二种情况。

还有一种情况，之前在分析 Master Thread 时曾讲到，在 Master Thread 线程中每秒或每 10 秒会进行一次 Merge Insert Buffer 的操作，不同之处在于每次进行 merge 操作的页的数量不同。

在 Master Thread 中，执行 merge 操作的不止是一个页，而是根据 srv_innodb_io_capactiy 的百分比来决定真正要合并多少个辅助索引页。但 InnoDB 存储引擎又是根据怎样的算法来得知需要合并的辅助索引页呢？

在 Insert Buffer B+ 树中，辅助索引页根据（space，offset）都已排序好，故可以根据（space，offset）的排序顺序进行页的选择。然而，对于 Insert Buffer 页的选择，InnoDB 存储引擎并非采用这个方式，它随机地选择 Insert Buffer B+ 树的一个页，读取该页中的 space 及之后所需要数量的页。该算法在复杂情况下应有更好的公平性。同时，若进行 merge 时，要进行 merge 的表已经被删除，此时可以直接丢弃已经被 Insert/Change Buffer 的数据记录。

2.6.2 两次写

如果说 Insert Buffer 带给 InnoDB 存储引擎的是性能上的提升，那么 doublewrite（两次写）带给 InnoDB 存储引擎的是数据页的可靠性。

当发生数据库宕机时，可能 InnoDB 存储引擎正在写入某个页到表中，而这个页只写了一部分，比如 16KB 的页，只写了前 4KB，之后就发生了宕机，这种情况被称为部分写失效（partial page write）。在 InnoDB 存储引擎未使用 doublewrite 技术前，曾经出现过因为部分写失效而导致数据丢失的情况。

有经验的 DBA 也许会想，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚地认识到，重做日志中记录的是对页的物理操作，如偏移量 800，写 'aaaa' 记录。如果这个页本身已经发生了损坏，再对其进行重做是没有意义的。这就是说，在应用（apply）重做日志前，用户需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做，这就是 doublewrite。在 InnoDB 存储引擎中 doublewrite 的体系架构如图 2-5 所示。

doublewrite 由两部分组成，一部分是内存中的 doublewrite buffer，大小为 2MB，另一部分是物理磁盘上共享表空间中连续的 128 个页，即 2 个区（extent），大小同样为 2MB。在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过 memcpy 函数将脏页先复制到内存中的 doublewrite buffer，之后通过 doublewrite buffer 再分两次，每次 1MB 顺序地写入共享表空间的物理磁盘上，然后马上调用 fsync 函数，同步磁盘，避免缓冲写带来的问题。在这个过程中，因为 doublewrite 页是连续的，因此这个过程是顺序

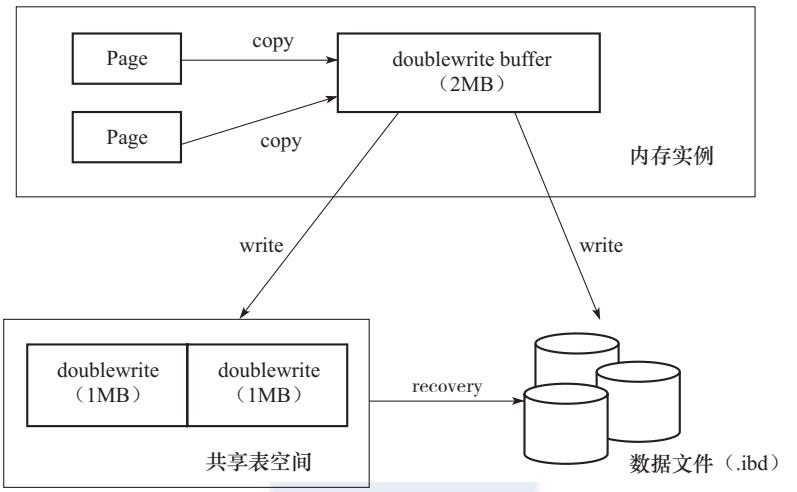


图 2-5　InnoDB 存储引擎 doublewrite 架构

写的，开销并不是很大。在完成 doublewrite 页的写入后，再将 doublewrite buffer 中的页写入各个表空间文件中，此时的写入则是离散的。可以通过以下命令观察到 doublewrite 运行的情况：

```
mysql>SHOW GLOBAL STATUS LIKE 'innodb_dblwr%'\G;
*************************** 1. row ***************************
Variable_name: Innodb_dblwr_pages_written
       Value: 6325194
*************************** 2. row ***************************
Variable_name: Innodb_dblwr_writes
       Value: 100399
2 rows in set (0.00 sec)
```

可以看到，doublewrite 一共写了 6 325 194 个页，但实际的写入次数为 100 399，基本上符合 64∶1。如果发现系统在高峰时的 Innodb_dblwr_pages_written:Innodb_dblwr_writes 远小于 64∶1，那么可以说明系统写入压力并不是很高。

如果操作系统在将页写入磁盘的过程中发生了崩溃，在恢复过程中，InnoDB 存储引擎可以从共享表空间中的 doublewrite 中找到该页的一个副本，将其复制到表空间文件，再应用重做日志。下面显示了一个由 doublewrite 进行恢复的情况：

```
090924 11:36:32  mysqld restarted
090924 11:36:33  InnoDB: Database was not shut down normally!
InnoDB: Starting crash recovery.
InnoDB: Reading tablespace information from the .ibd files...
InnoDB: Crash recovery may have failed for some .ibd files!
```

```
InnoDB: Restoring possible half-written data pages from the doublewrite
InnoDB: buffer...
```

若查看MySQL官方手册，会发现在命令SHOW GLOBAL STATUS中Innodb_buffer_pool_pages_flushed变量表示当前从缓冲池中刷新到磁盘页的数量。根据之前的介绍，用户应该了解到，在默认情况下所有页的刷新首先都需要放入到doublewrite中，因此该变量应该和Innodb_dblwr_pages_written一致。然而在MySQL 5.5.24版本之前，Innodb_buffer_pool_pages_flushed总是为Innodb_dblwr_pages_written的2倍，而此Bug直到MySQL5.5.24才被修复。因此用户若需要统计数据库在生产环境中写入的量，最安全的方法还是根据Innodb_dblwr_pages_written来进行统计，这在所有版本的MySQL数据库中都是正确的。

参数skip_innodb_doublewrite可以禁止使用doublewrite功能，这时可能会发生前面提及的写失效问题。不过如果用户有多个从服务器（slave server），需要提供较快的性能（如在slaves erver上做的是RAID0），也许启用这个参数是一个办法。不过对于需要提供数据高可靠性的主服务器（master server），任何时候用户都应确保开启doublewrite功能。

注意 有些文件系统本身就提供了部分写失效的防范机制，如ZFS文件系统。在这种情况下，用户就不要启用doublewrite了。

2.6.3 自适应哈希索引

哈希（hash）是一种非常快的查找方法，在一般情况下这种查找的时间复杂度为O(1)，即一般仅需要一次查找就能定位数据。而B+树的查找次数，取决于B+树的高度，在生产环境中，B+树的高度一般为3～4层，故需要3～4次的查询。

InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引（Adaptive Hash Index，AHI）。AHI是通过缓冲池的B+树页构造而来，因此建立的速度很快，而且不需要对整张表构建哈希索引。InnoDB存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引。

AHI有一个要求，即对这个页的连续访问模式必须是一样的。例如对于（a，b）这

样的联合索引页，其访问模式可以是以下情况：

- WHERE a=xxx
- WHERE a=xxx and b=xxx

访问模式一样指的是查询的条件一样，若交替进行上述两种查询，那么 InonDB 存储引擎不会对该页构造 AHI。此外 AHI 还有如下的要求：

- 以该模式访问了 100 次
- 页通过该模式访问了 *N* 次，其中 *N*= 页中记录 *1/16

根据 InnoDB 存储引擎官方的文档显示，启用 AHI 后，读取和写入速度可以提高 2 倍，辅助索引的连接操作性能可以提高 5 倍。毫无疑问，AHI 是非常好的优化模式，其设计思想是数据库自优化的（self-tuning），即无需 DBA 对数据库进行人为调整。

通过命令 SHOW ENGINE INNODB STATUS 可以看到当前 AHI 的使用状况：

```
mysql>SHOW ENGINE INNODB STATUS\G;
*************************** 1. row ***************************
Status:
=====================================
090922 11:52:51 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 15 seconds
……
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 2249, free list len 3346, seg size 5596,
374650 inserts, 51897 merged recs, 14300 merges
Hash table size 4980499, node heap has 1246 buffer(s)
1640.60 hash searches/s, 3709.46 non-hash searches/s
……
```

现在可以看到 AHI 的使用信息了，包括 AHI 的大小、使用情况、每秒使用 AHI 搜索的情况。值得注意的是，哈希索引只能用来搜索等值的查询，如 SELECT*FROM table WHERE index_col='xxx'。而对于其他查找类型，如范围查找，是不能使用哈希索引的，因此这里出现了 non-hash searches/s 的情况。通过 hash searches:non-hash searches 可以大概了解使用哈希索引后的效率。

由于 AHI 是由 InnoDB 存储引擎控制的，因此这里的信息只供用户参考。不过用户可以通过观察 SHOW ENGINE INNODB STATUS 的结果及参数 innodb_adaptive_hash_

index 来考虑是禁用或启动此特性，默认 AHI 为开启状态。

2.6.4 异步 IO

为了提高磁盘操作性能，当前的数据库系统都采用异步 IO（Asynchronous IO，AIO）的方式来处理磁盘操作。InnoDB 存储引擎亦是如此。

与 AIO 对应的是 Sync IO，即每进行一次 IO 操作，需要等待此次操作结束才能继续接下来的操作。但是如果用户发出的是一条索引扫描的查询，那么这条 SQL 查询语句可能需要扫描多个索引页，也就是需要进行多次的 IO 操作。在每扫描一个页并等待其完成后再进行下一次的扫描，这是没有必要的。用户可以在发出一个 IO 请求后立即再发出另一个 IO 请求，当全部 IO 请求发送完毕后，等待所有 IO 操作的完成，这就是 AIO。

AIO 的另一个优势是可以进行 IO Merge 操作，也就是将多个 IO 合并为 1 个 IO，这样可以提高 IOPS 的性能。例如用户需要访问页的（space，page_no）为：

（8，6）、（8，7），（8，8）

每个页的大小为 16KB，那么同步 IO 需要进行 3 次 IO 操作。而 AIO 会判断到这三个页是连续的（显然可以通过（space，page_no）得知）。因此 AIO 底层会发送一个 IO 请求，从（8，6）开始，读取 48KB 的页。

若通过 Linux 操作系统下的 iostat 命令，可以通过观察 rrqm/s 和 wrqm/s，例如：

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           4.70    0.00    1.60   13.20    0.00   80.50

Device:    rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz
await  svctm  %util
sdc        3905.67   172.00 6910.33  466.67   168.81    18.15    51.91    19.17
2.59   0.13  97.73
```

在 InnoDB1.1.x 之前，AIO 的实现通过 InnoDB 存储引擎中的代码来模拟实现。而从 InnoDB 1.1.x 开始（InnoDB Plugin 不支持），提供了内核级别 AIO 的支持，称为 Native AIO。因此在编译或者运行该版本 MySQL 时，需要 libaio 库的支持。若没有则会出现如下的提示：

```
/usr/local/mysql/bin/mysqld: error while loading shared libraries: libaio.so.1:
cannot open shared object file: No such file or directory
```

需要注意的是，Native AIO 需要操作系统提供支持。Windows 系统和 Linux 系统都

提供 Native AIO 支持，而 Mac OSX 系统则未提供。因此在这些系统下，依旧只能使用原模拟的方式。在选择 MySQL 数据库服务器的操作系统时，需要考虑这方面的因素。

参数 innodb_use_native_aio 用来控制是否启用 Native AIO，在 Linux 操作系统下，默认值为 ON：

```
mysql> SHOW VARIABLES LIKE 'innodb_use_native_aio'\G;
*************************** 1. row ***************************
Variable_name: innodb_use_native_aio
        Value: ON
1 row in set (0.00 sec)
```

用户可以通过开启和关闭 Native AIO 功能来比较 InnoDB 性能的提升。官方的测试显示，启用 Native AIO，恢复速度可以提高 75%。

在 InnoDB 存储引擎中，read ahead 方式的读取都是通过 AIO 完成，脏页的刷新，即磁盘的写入操作则全部由 AIO 完成。

2.6.5 刷新邻接页

InnoDB 存储引擎还提供了 Flush Neighbor Page（刷新邻接页）的特性。其工作原理为：当刷新一个脏页时，InnoDB 存储引擎会检测该页所在区（extent）的所有页，如果是脏页，那么一起进行刷新。这样做的好处显而易见，通过 AIO 可以将多个 IO 写入操作合并为一个 IO 操作，故该工作机制在传统机械磁盘下有着显著的优势。但是需要考虑到下面两个问题：

- 是不是可能将不怎么脏的页进行了写入，而该页之后又会很快变成脏页？
- 固态硬盘有着较高的 IOPS，是否还需要这个特性？

为此，InnoDB 存储引擎从 1.2.x 版本开始提供了参数 innodb_flush_neighbors，用来控制是否启用该特性。对于传统机械硬盘建议启用该特性，而对于固态硬盘有着超高 IOPS 性能的磁盘，则建议将该参数设置为 0，即关闭此特性。

2.7 启动、关闭与恢复

InnoDB 是 MySQL 数据库的存储引擎之一，因此 InnoDB 存储引擎的启动和关闭，更准确的是指在 MySQL 实例的启动过程中对 InnoDB 存储引擎的处理过程。

在关闭时，参数 innodb_fast_shutdown 影响着表的存储引擎为 InnoDB 的行为。该参数可取值为 0、1、2，默认值为 1。

- 0 表示在 MySQL 数据库关闭时，InnoDB 需要完成所有的 full purge 和 merge insert buffer，并且将所有的脏页刷新回磁盘。这需要一些时间，有时甚至需要几个小时来完成。如果在进行 InnoDB 升级时，必须将这个参数调为 0，然后再关闭数据库。
- 1 是参数 innodb_fast_shutdown 的默认值，表示不需要完成上述的 full purge 和 merge insert buffer 操作，但是在缓冲池中的一些数据脏页还是会刷新回磁盘。
- 2 表示不完成 full purge 和 merge insert buffer 操作，也不将缓冲池中的数据脏页写回磁盘，而是将日志都写入日志文件。这样不会有任何事务的丢失，但是下次 MySQL 数据库启动时，会进行恢复操作（recovery）。

当正常关闭 MySQL 数据库时，下次的启动应该会非常“正常”。但是如果没有正常地关闭数据库，如用 kill 命令关闭数据库，在 MySQL 数据库运行中重启了服务器，或者在关闭数据库时，将参数 innodb_fast_shutdown 设为了 2 时，下次 MySQL 数据库启动时都会对 InnoDB 存储引擎的表进行恢复操作。

参数 innodb_force_recovery 影响了整个 InnoDB 存储引擎恢复的状况。该参数值默认为 0，代表当发生需要恢复时，进行所有的恢复操作，当不能进行有效恢复时，如数据页发生了 corruption，MySQL 数据库可能发生宕机（crash），并把错误写入错误日志中去。

但是，在某些情况下，可能并不需要进行完整的恢复操作，因为用户自己知道怎么进行恢复。比如在对一个表进行 alter table 操作时发生意外了，数据库重启时会对 InnoDB 表进行回滚操作，对于一个大表来说这需要很长时间，可能是几个小时。这时用户可以自行进行恢复，如可以把表删除，从备份中重新导入数据到表，可能这些操作的速度要远远快于回滚操作。

参数 innodb_force_recovery 还可以设置为 6 个非零值：1 ～ 6。大的数字表示包含了前面所有小数字表示的影响。具体情况如下：

- 1(SRV_FORCE_IGNORE_CORRUPT)：忽略检查到的 corrupt 页。
- 2(SRV_FORCE_NO_BACKGROUND)：阻止 Master Thread 线程的运行，如 Master Thread 线程需要进行 full purge 操作，而这会导致 crash。
- 3(SRV_FORCE_NO_TRX_UNDO)：不进行事务的回滚操作。

- 4(SRV_FORCE_NO_IBUF_MERGE)：不进行插入缓冲的合并操作。
- 5(SRV_FORCE_NO_UNDO_LOG_SCAN)：不查看撤销日志（Undo Log），InnoDB 存储引擎会将未提交的事务视为已提交。
- 6(SRV_FORCE_NO_LOG_REDO)：不进行前滚的操作。

需要注意的是，在设置了参数 innodb_force_recovery 大于 0 后，用户可以对表进行 select、create 和 drop 操作，但 insert、update 和 delete 这类 DML 操作是不允许的。

现在来做一个实验，模拟故障的发生。在第一个会话中（session），对一张接近 1 000 万行的 InnoDB 存储引擎表进行更新操作，但是完成后不要马上提交：

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

mysql>UPDATE Profile SET password='';
Query OK, 9587770 rows affected (7 min 55.73 sec)
Rows matched: 9999248  Changed: 9587770  Warnings: 0
```

START TRANSACTION 语句开启了事务，同时防止了自动提交（auto commit）的发生，UPDATE 操作则会产生大量的 UNDO 日志（undo log）。这时，人为通过 kill 命令杀掉 MySQL 数据库服务器：

```
[root@nineyou0-43 ~]# ps -ef | grep mysqld
root     28007     1  0 13:40 pts/1    00:00:00 /bin/sh./bin/mysqld_safe --datadir =/
usr/local/mysql/data --pid-file=/usr/local/mysql/data/nineyou0-43.pid
mysql    28045 28007 42 13:40 pts/1    00:04:23 /usr/local/mysql/bin/mysqld -- basedir
=/usr/local/mysql  --datadir=/usr/local/mysql/data  --user=mysql  --pid-file=/usr/
local/mysql/data/nineyou0-43.pid --skip-external-locking --port=3306 --socket=/tmp/
mysql.sock
root    28110 26963  0 13:50 pts/11    00:00:00 grep mysqld
[root@nineyou0-43 ~]# kill -9 28007
[root@nineyou0-43 ~]# kill -9 28045
```

通过 kill 命令可以模拟数据库的宕机操作。下次 MySQL 数据库启动时会对之前的 UPDATE 事务进行回滚操作，而这些信息都会记录在错误日志文件（默认后缀名为 err）中。如果查看错误日志文件，可得如下结果：

```
090922 13:40:20  InnoDB: Started; log sequence number 6 2530474615
InnoDB: Starting in background the rollback of uncommitted transactions
090922 13:40:20  InnoDB: Rolling back trx with id 0 5281035, 8867280 rows to undo

InnoDB: Progress in percents: 1090922 13:40:20
```

```
    090922 13:40:20 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
    Version: '5.0.45-log'  socket: '/tmp/mysql.sock'  port: 3306  MySQL Community
Server (GPL)
     2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 95 96 97 98 99 100
    InnoDB: Rolling back of trx id 0 5281035 completed
    090922 13:49:21  InnoDB: Rollback of non-prepared transactions completed
```

可以看到，采用默认的策略，即将 innodb_force_recovery 设为 0，InnoDB 会在每次启动后对发生问题的表进行恢复操作。通过错误日志文件，可知这次回滚操作需要回滚 8867280 行记录，差不多总共进行了 9 分钟。

再做一次同样的测试，只不过这次在启动 MySQL 数据库前，将参数 innodb_force_recovery 设为 3，然后观察 InnoDB 存储引擎是否还会进行回滚操作。查看错误日志文件，可得：

```
    090922 14:26:23  InnoDB: Started; log sequence number 7 2253251193
    InnoDB: !!!innodb_force_recovery is set to 3 !!!
    090922 14:26:23 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
    Version: '5.0.45-log'  socket: '/tmp/mysql.sock'  port: 3306  MySQL Community
Server (GPL)
```

这里出现了“!!!”，InnoDB 警告已经将 innodb_force_recovery 设置为 3，不会进行回滚操作了，因此数据库很快启动完成了。但是用户应该小心当前数据库的状态，并仔细确认是否不需要回滚事务的操作。

2.8 小结

本章对 InnoDB 存储引擎及其体系结构进行了概述，先给出了 InnoDB 存储引擎的历史、InnoDB 存储引擎的体系结构（包括后台线程和内存结构）；之后又详细介绍了 InnoDB 存储引擎的关键特性，这些特性使 InnoDB 存储引擎变得更具“魅力”；最后介绍了启动和关闭 MySQL 时一些配置文件参数对 InnoDB 存储引擎的影响。

通过本章的铺垫，读者在学习后面的内容时就会对 InnoDB 引擎理解得更深入和更全面。第 3 章开始介绍 MySQL 的文件，包括 MySQL 本身的文件和与 InnoDB 存储引擎本身有关的文件。之后本书将介绍基于 InnoDB 存储引擎的表，并揭示内部的存储构造。

第3章 文　　件

本章将分析构成 MySQL 数据库和 InnoDB 存储引擎表的各种类型文件。这些文件有以下这些。

- 参数文件：告诉 MySQL 实例启动时在哪里可以找到数据库文件，并且指定某些初始化参数，这些参数定义了某种内存结构的大小等设置，还会介绍各种参数的类型。
- 日志文件：用来记录 MySQL 实例对某种条件做出响应时写入的文件，如错误日志文件、二进制日志文件、慢查询日志文件、查询日志文件等。
- socket 文件：当用 UNIX 域套接字方式进行连接时需要的文件。
- pid 文件：MySQL 实例的进程 ID 文件。
- MySQL 表结构文件：用来存放 MySQL 表结构定义文件。
- 存储引擎文件：因为 MySQL 表存储引擎的关系，每个存储引擎都会有自己的文件来保存各种数据。这些存储引擎真正存储了记录和索引等数据。本章主要介绍与 InnoDB 有关的存储引擎文件。

3.1 参数文件

在第 1 章中已经介绍过了，当 MySQL 实例启动时，数据库会先去读一个配置参数文件，用来寻找数据库的各种文件所在位置以及指定某些初始化参数，这些参数通常定义了某种内存结构有多大等。在默认情况下，MySQL 实例会按照一定的顺序在指定的位置进行读取，用户只需通过命令 mysql--help | grep my.cnf 来寻找即可。

MySQL 数据库参数文件的作用和 Oracle 数据库的参数文件极其类似，不同的是，Oracle 实例在启动时若找不到参数文件，是不能进行装载（mount）操作的。MySQL 稍微有所不同，MySQL 实例可以不需要参数文件，这时所有的参数值取决于编译 MySQL 时指定的默认值和源代码中指定参数的默认值。但是，如果 MySQL 实例在默认的数据

库目录下找不到 mysql 架构，则启动同样会失败，此时可能在错误日志文件中找到如下内容：

```
090922 16:25:52  mysqld started
090922 16:25:53  InnoDB: Started; log sequence number 8 2801063211
InnoDB: !!! innodb_force_recovery is set to 1 !!!
090922 16:25:53 [ERROR] Fatal error: Can't open and lock privilege tables:
Table 'mysql.host' doesn't exist
090922 16:25:53  mysqld ended
```

MySQL 的 mysql 架构中记录了访问该实例的权限，当找不到这个架构时，MySQL 实例不会成功启动。

MySQL 数据库的参数文件是以文本方式进行存储的。用户可以直接通过一些常用的文本编辑软件（如 vi 和 emacs）进行参数的修改。

3.1.1 什么是参数

简单地说，可以把数据库参数看成一个键 / 值（key/value）对。第 2 章已经介绍了一个对于 InnoDB 存储引擎很重要的参数 innodb_buffer_pool_size。如我们将这个参数设置为 1G，即 innodb_buffer_pool_size=1G。这里的“键”是 innodb_buffer_pool_size，“值”是 1G，这就是键值对。可以通过命令 SHOW VARIABLES 查看数据库中的所有参数，也可以通过 LIKE 来过滤参数名。从 MySQL 5.1 版本开始，还可以通过 information_schema 架构下的 GLOBAL_VARIABLES 视图来进行查找，如下所示。

```
mysql>  SELECT * FROM
    ->  GLOBAL_VARIABLES
    ->  WHERE VARIABLE_NAME LIKE 'innodb_buffer%'\G;
*************************** 1. row ***************************
 VARIABLE_NAME: INNODB_BUFFER_POOL_SIZE
VARIABLE_VALUE: 1073741824
1 row in set (0.00 sec)

mysql>SHOW VARIABLES LIKE 'innodb_buffer%'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 1073741824
1 row in set (0.00 sec)
```

无论使用哪种方法，输出的信息基本上都一样的，只不过通过视图 GLOBAL_

VARIABLES 需要指定视图的列名。推荐使用命令 SHOW VARIABLES，因为这个命令使用更为简单，且各版本的 MySQL 数据库都支持。

Oracle 数据库存在所谓的隐藏参数（undocumented parameter），以供 Oracle“内部人士”使用，SQL Server 也有类似的参数。有些 DBA 曾问我，MySQL 中是否也有这类参数。我的回答是：没有，也不需要。即使 Oracle 和 SQL Server 中都有些所谓的隐藏参数，在绝大多数的情况下，这些数据库厂商也不建议用户在生产环境中对其进行很大的调整。

3.1.2 参数类型

MySQL 数据库中的参数可以分为两类：

- 动态（dynamic）参数
- 静态（static）参数

动态参数意味着可以在 MySQL 实例运行中进行更改，静态参数说明在整个实例生命周期内都不得进行更改，就好像是只读（read only）的。可以通过 SET 命令对动态的参数值进行修改，SET 的语法如下：

```
SET
| [global | session] system_var_name= expr
| [@@global. | @@session. | @@]system_var_name= expr
```

这里可以看到 global 和 session 关键字，它们表明该参数的修改是基于当前会话还是整个实例的生命周期。有些动态参数只能在会话中进行修改，如 autocommit ；而有些参数修改完后，在整个实例生命周期中都会生效，如 binlog_cache_size ；而有些参数既可以在会话中又可以在整个实例的生命周期内生效，如 read_buffer_size。举例如下：

```
mysql>SET read_buffer_size=524288;
Query OK, 0 rows affected (0.00 sec)

mysql>SELECT @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql>SELECT @@global.read_buffer_size\G;
*************************** 1. row ***************************
```

```
@@global.read_buffer_size: 2093056
1 row in set (0.00 sec)
```

上述示例中将当前会话的参数 read_buffer_size 从 2MB 调整为了 512KB，而用户可以看到全局的 read_buffer_size 的值仍然是 2MB，也就是说如果有另一个会话登录到 MySQL 实例，它的 read_buffer_size 的值是 2MB，而不是 512KB。这里使用了 set global|session 来改变动态变量的值。用户同样可以直接使用 SET@@globl|@@session 来更改，如下所示：

```
mysql>SET @@global.read_buffer_size=1048576;
Query OK, 0 rows affected (0.00 sec)

mysql>SELECT @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql>SELECT @@global.read_buffer_size\G;
*************************** 1. row ***************************
@@global.read_buffer_size: 1048576
1 row in set (0.00 sec)
```

这次把 read_buffer_size 全局值更改为 1MB，而当前会话的 read_buffer_size 的值还是 512KB。这里需要注意的是，对变量的全局值进行了修改，在这次的实例生命周期内都有效，但 MySQL 实例本身并不会对参数文件中的该值进行修改。也就是说，在下次启动时 MySQL 实例还是会读取参数文件。若想在数据库实例下一次启动时该参数还是保留为当前修改的值，那么用户必须去修改参数文件。要想知道 MySQL 所有动态变量的可修改范围，可以参考 MySQL 官方手册的 Dynamic System Variables 的相关内容。

对于静态变量，若对其进行修改，会得到类似如下错误：

```
mysql>SET GLOBAL datadir='/db/mysql';
ERROR 1238 (HY000): Variable 'datadir' is a read only variable
```

3.2 日志文件

日志文件记录了影响 MySQL 数据库的各种类型活动。MySQL 数据库中常见的日志文件有：

- 错误日志（error log）
- 二进制日志（binlog）
- 慢查询日志（slow query log）
- 查询日志（log）

这些日志文件可以帮助 DBA 对 MySQL 数据库的运行状态进行诊断，从而更好地进行数据库层面的优化。

3.2.1 错误日志

错误日志文件对 MySQL 的启动、运行、关闭过程进行了记录。MySQL DBA 在遇到问题时应该首先查看该文件以便定位问题。该文件不仅记录了所有的错误信息，也记录一些警告信息或正确的信息。用户可以通过命令 SHOW VARIABLES LIKE 'log_error' 来定位该文件，如：

```
mysql> SHOW VARIABLES LIKE 'log_error'\G;
*************************** 1. row ***************************
Variable_name: log_error
        Value: /mysql_data_2/stargazer.log
1 row in set (0.00 sec)

mysql> system hostname
stargazer
```

可以看到错误文件的路径和文件名，在默认情况下错误文件的文件名为服务器的主机名。如上面看到的，该主机名为 stargazer，所以错误文件名为 startgazer.err。当出现 MySQL 数据库不能正常启动时，第一个必须查找的文件应该就是错误日志文件，该文件记录了错误信息，能很好地指导用户发现问题。当数据库不能重启时，通过查错误日志文件可以得到如下内容：

```
[root@nineyou0-43 data]# tail -n 50 nineyou0-43.err
090924 11:31:18  mysqld started
090924 11:31:18  InnoDB: Started; log sequence number 8 2801063331
090924 11:31:19 [ERROR] Fatal error: Can't open and lock privilege tables:
Table 'mysql.host' doesn't exist
090924 11:31:19  mysqld ended
```

这里，错误日志文件提示了找不到权限库 mysql，所以启动失败。有时用户可以直

接在错误日志文件中得到优化的帮助，因为有些警告（warning）很好地说明了问题所在。而这时可以不需要通过查看数据库状态来得知，例如，下面的错误文件中的信息可能告诉用户需要增大 InnoDB 存储引擎的 redo log：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

3.2.2 慢查询日志

3.2.1 小节提到可以通过错误日志得到一些关于数据库优化的信息，而慢查询日志（slow log）可帮助 DBA 定位可能存在问题的 SQL 语句，从而进行 SQL 语句层面的优化。例如，可以在 MySQL 启动时设一个阈值，将运行时间超过该值的所有 SQL 语句都记录到慢查询日志文件中。DBA 每天或每过一段时间对其进行检查，确认是否有 SQL 语句需要进行优化。该阈值可以通过参数 long_query_time 来设置，默认值为 10，代表 10 秒。

在默认情况下，MySQL 数据库并不启动慢查询日志，用户需要手工将这个参数设为 ON：

```
mysql> SHOW VARIABLES LIKE 'long_query_time'\G;
*************************** 1. row ***************************
Variable_name: long_query_time
       Value: 10.000000
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'log_slow_queries'\G;
*************************** 1. row ***************************
```

```
Variable_name: log_slow_queries
        Value: ON
1 row in set (0.00 sec)
```

这里有两点需要注意。首先，设置 long_query_time 这个阈值后，MySQL 数据库会记录运行时间超过该值的所有 SQL 语句，但运行时间正好等于 long_query_time 的情况并不会被记录下。也就是说，在源代码中判断的是大于 long_query_time，而非大于等于。其次，从 MySQL 5.1 开始，long_query_time 开始以微秒记录 SQL 语句运行的时间，之前仅用秒为单位记录。而这样可以更精确地记录 SQL 的运行时间，供 DBA 分析。对 DBA 来说，一条 SQL 语句运行 0.5 秒和 0.05 秒是非常不同的，前者可能已经进行了表扫，后面可能是进行了索引。

另一个和慢查询日志有关的参数是 log_queries_not_using_indexes，如果运行的 SQL 语句没有使用索引，则 MySQL 数据库同样会将这条 SQL 语句记录到慢查询日志文件。首先确认打开了 log_queries_not_using_indexes：

```
mysql> SHOW VARIABLES LIKE 'log_queries_not_using_indexes'\G;
*************************** 1. row ***************************
Variable_name: log_queries_not_using_indexes
        Value: ON
1 row in set (0.00 sec)
```

MySQL 5.6.5 版本开始新增了一个参数 log_throttle_queries_not_using_indexes，用来表示每分钟允许记录到 slow log 的且未使用索引的 SQL 语句次数。该值默认为 0，表示没有限制。在生产环境下，若没有使用索引，此类 SQL 语句会频繁地被记录到 slow log，从而导致 slow log 文件的大小不断增加，故 DBA 可通过此参数进行配置。

DBA 可以通过慢查询日志来找出有问题的 SQL 语句，对其进行优化。然而随着 MySQL 数据库服务器运行时间的增加，可能会有越来越多的 SQL 查询被记录到了慢查询日志文件中，此时要分析该文件就显得不是那么简单和直观的了。而这时 MySQL 数据库提供的 mysqldumpslow 命令，可以很好地帮助 DBA 解决该问题：

```
[root@nh122-190 data]# mysqldumpslow nh122-190-slow.log
Reading mysql slow query log from nh122-190-slow.log
Count: 11  Time=10.00s (110s)  Lock=0.00s (0s)  Rows=0.0 (0), dbother[dbother]@
localhost
  insert into test.DbStatus select now(),(N-com_select)/(N-uptime),(N-com_insert)/
(N-uptime),(N-com_update)/(N-uptime),(N-com_delete)/(N-uptime),N-(N/N),N-(N/N),N.N/
N,N-N/(N*N),GetCPULoadInfo(N) from test.CheckDbStatus order by check_id desc limit N
```

```
    Count: 653  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=0.0 (0), 9YOUgs_SC[9YOUgs_
SC]@[192.168.43.7]
      select custom_name_one from 'low_game_schema'.'role_details' where role_id='S'
    rse and summarize the MySQL slow query log. Options are

      --verbose    verbose
      --debug      debug
      --help       write this text to standard output

      -v           verbose
      -d           debug
      -s ORDER     what to sort by (al, at, ar, c, l, r, t), 'at' is default
                   al: average lock time
                   ar: average rows sent
                   at: average query time
                    c: count
                    l: lock time
                    r: rows sent
                    t: query time
      -r           reverse the sort order (largest last instead of first)
      -t NUM       just show the top n queries
      -a           don't abstract all numbers to N and strings to 'S'
      -n NUM       abstract numbers with at least n digits within names
      -g PATTERN   grep: only consider stmts that include this string
      -h HOSTNAME  hostname of db server for *-slow.log filename (can be wildcard),
                   default is '*', i.e. match all
      -i NAME      name of server instance (if using mysql.server startup script)
      -l           don't subtract lock time from total time
```

如果用户希望得到执行时间最长的 10 条 SQL 语句，可以运行如下命令：

```
    [root@nh119-141 data]# mysqldumpslow -s al -n 10 david.log
    Reading mysql slow query log from david.log
    Count: 5  Time=0.00s (0s)  Lock=0.20s (1s)  Rows=4.4 (22), Audition[Audition]@
[192.168.30.108]
      SELECT OtherSN, State FROM wait_friend_info WHERE UserSN = N

    Count: 1  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=1.0 (1), audition-kr[audition-
kr]@[192.168.30.105]
      SELECT COUNT(N) FROM famverifycode WHERE UserSN=N AND verifycode='S'
    ……
```

MySQL 5.1 开始可以将慢查询的日志记录放入一张表中，这使得用户的查询更加方便和直观。慢查询表在 mysql 架构下，名为 slow_log，其表结构定义如下：

```
mysql> SHOW CREATE TABLE mysql.slow_log\G;
*************************** 1. row ***************************
       Table: slow_log
Create Table: CREATE TABLE 'slow_log' (
  'start_time' timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
  'user_host' mediumtext NOT NULL,
  'query_time' time NOT NULL,
  'lock_time' time NOT NULL,
  'rows_sent' int(11) NOT NULL,
  'rows_examined' int(11) NOT NULL,
  'db' varchar(512) NOT NULL,
  'last_insert_id' int(11) NOT NULL,
  'insert_id' int(11) NOT NULL,
  'server_id' int(11) NOT NULL,
  'sql_text' mediumtext NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='Slow log'
1 row in set (0.00 sec)
```

参数 log_output 指定了慢查询输出的格式，默认为 FILE，可以将它设为 TABLE，然后就可以查询 mysql 架构下的 slow_log 表了，如：

```
mysql>SHOW VARIABLES LIKE 'log_output'\G;
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output    | FILE |
+---------------+---------+
1 row in set (0.00 sec)

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

mysql>SHOW VARIABLES LIKE 'log_output'\G;
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output  | TABLE |
+---------------+---------+
1 row in set (0.00 sec)

mysql> select sleep(10)\G;
+-----------+
| sleep(10)|
```

```
+-----------+
|         0 |
+-----------+
1 row in set (10.01 sec)

mysql>  SELECT * FROM mysql.slow_log\G;
*************************** 1. row ***************************
    start_time: 2009-09-25 13:44:29
     user_host: david[david] @ localhost []
    query_time: 00:00:09
     lock_time: 00:00:00
     rows_sent: 1
 rows_examined: 0
            db: mysql
last_insert_id: 0
     insert_id: 0
     server_id: 0
      sql_text: select sleep(10)
1 row in set (0.00 sec)
```

参数 log_output 是动态的，并且是全局的，因此用户可以在线进行修改。在上表中人为设置了睡眠（sleep）10 秒，那么这句 SQL 语句就会被记录到 slow_log 表了。

查看 slow_log 表的定义会发现该表使用的是 CSV 引擎，对大数据量下的查询效率可能不高。用户可以把 slow_log 表的引擎转换到 MyISAM，并在 start_time 列上添加索引以进一步提高查询的效率。但是，如果已经启动了慢查询，将会提示错误：

```
mysql>ALTER TABLE mysql.slow_log ENGINE=MyISM;
ERROR 1580 (HY000): You cannot 'ALTER' a log table if logging is enabled

mysql>SET GLOBAL slow_query_log=off;
Query OK, 0 rows affected (0.00 sec)

mysql>ALTER TABLE mysql.slow_log ENGINE=MyISAM;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

不能忽视的是，将 slow_log 表的存储引擎更改为 MyISAM 后，还是会对数据库造成额外的开销。不过好在很多关于慢查询的参数都是动态的，用户可以方便地在线进行设置或修改。

MySQL 的 slow log 通过运行时间来对 SQL 语句进行捕获，这是一个非常有用的优化技巧。但是当数据库的容量较小时，可能因为数据库刚建立，此时非常大的可能是数

据全部被缓存在缓冲池中，SQL 语句运行的时间可能都是非常短的，一般都是 0.5 秒。

InnoSQL 版本加强了对于 SQL 语句的捕获方式。在原版 MySQL 的基础上在 slow log 中增加了对于逻辑读取（logical reads）和物理读取（physical reads）的统计。这里的物理读取是指从磁盘进行 IO 读取的次数，逻辑读取包含所有的读取，不管是磁盘还是缓冲池。例如：

```
    # Time: 111227 23:49:16
    # User@Host: root[root] @ localhost [127.0.0.1]
    # Query_time: 6.081214 Lock_time: 0.046800 Rows_sent: 42 Rows_examined: 727558
Logical_reads: 91584 Physical_reads: 19
    use tpcc;
    SET timestamp=1325000956;
    SELECT orderid,customerid,employeeid,orderdate
    FROM orders
    WHERE orderdate IN
    ( SELECT MAX(orderdate)
    FROM orders
    GROUP BY (DATE_FORMAT(orderdate,'%Y%M'))
    );
```

从上面的例子可以看到该子查询的逻辑读取次数是 91 584 次，物理读取为 19 次。从逻辑读与物理读的比例上看，该 SQL 语句可进行优化。

用户可以通过额外的参数 long_query_io 将超过指定逻辑 IO 次数的 SQL 语句记录到 slow log 中。该值默认为 100，即表示对于逻辑读取次数大于 100 的 SQL 语句，记录到 slow log 中。而为了兼容原 MySQL 数据库的运行方式，还添加了参数 slow_query_type，用来表示启用 slow log 的方式，可选值为：

- 0 表示不将 SQL 语句记录到 slow log
- 1 表示根据运行时间将 SQL 语句记录到 slow log
- 2 表示根据逻辑 IO 次数将 SQL 语句记录到 slow log
- 3 表示根据运行时间及逻辑 IO 次数将 SQL 语句记录到 slow log

3.2.3 查询日志

查询日志记录了所有对 MySQL 数据库请求的信息，无论这些请求是否得到了正确的执行。默认文件名为：主机名 .log。如查看一个查询日志：

```
[root@nineyou0-43 data]# tail nineyou0-43.log
090925 11:00:24   44 Connect     zlm@192.168.0.100 on
44 Query       SET AUTOCOMMIT=0
                44 Query       set autocommit=0
                44 Quit
090925 11:02:37  45 Connect    Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:03:51  46 Connect    Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:04:38  23 Query       rollback
```

通过上述查询日志会发现，查询日志甚至记录了对 Access denied 的请求，即对于未能正确执行的 SQL 语句，查询日志也会进行记录。同样地，从 MySQL 5.1 开始，可以将查询日志的记录放入 mysql 架构下的 general_log 表中，该表的使用方法和前面小节提到的 slow_log 基本一样，这里不再赘述。

3.2.4 二进制日志

二进制日志（binary log）记录了对 MySQL 数据库执行更改的所有操作，但是不包括 SELECT 和 SHOW 这类操作，因为这类操作对数据本身并没有修改。然而，若操作本身并没有导致数据库发生变化，那么该操作可能也会写入二进制日志。例如：

```
mysql> UPDATE t SET a = 1 WHERE a = 2;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> SHOW MASTER STATUS\G;
*************************** 1. row ***************************
             File: mysqld.000008
         Position: 383
     Binlog_Do_DB:
 Binlog_Ignore_DB:
Executed_Gtid_Set:
1 row in set (0.00 sec)

mysql> SHOW BINLOG EVENTS IN 'mysqld.000008'\G;
*************************** 1. row ***************************
   Log_name: mysqld.000008
        Pos: 4
 Event_type: Format_desc
  Server_id: 1
```

```
End_log_pos: 120
       Info: Server ver: 5.6.6-m9-log, Binlog ver: 4
*************************** 2. row ***************************
   Log_name: mysqld.000008
        Pos: 120
 Event_type: Query
  Server_id: 1
End_log_pos: 199
       Info: BEGIN
*************************** 3. row ***************************
   Log_name: mysqld.000008
        Pos: 199
 Event_type: Query
  Server_id: 1
End_log_pos: 303
       Info: use 'test'; UPDATE t SET a = 1 WHERE a = 2
*************************** 4. row ***************************
   Log_name: mysqld.000008
        Pos: 303
 Event_type: Query
  Server_id: 1
End_log_pos: 383
       Info: COMMIT
4 rows in set (0.00 sec)
```

从上述例子中可以看到，MySQL 数据库首先进行 UPDATE 操作，从返回的结果看到 Changed 为 0，这意味着该操作并没有导致数据库的变化。但是通过命令 SHOW BINLOG EVENT 可以看出在二进制日志中的确进行了记录。

如果用户想记录 SELECT 和 SHOW 操作，那只能使用查询日志，而不是二进制日志。此外，二进制日志还包括了执行数据库更改操作的时间等其他额外信息。总的来说，二进制日志主要有以下几种作用。

- **恢复**（recovery）：某些数据的恢复需要二进制日志，例如，在一个数据库全备文件恢复后，用户可以通过二进制日志进行 point-in-time 的恢复。
- **复制**（replication）：其原理与恢复类似，通过复制和执行二进制日志使一台远程的 MySQL 数据库（一般称为 slave 或 standby）与一台 MySQL 数据库（一般称为 master 或 primary）进行实时同步。
- **审计**（audit）：**用户可以通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入的攻击**。

通过配置参数 log-bin［=name］可以启动二进制日志。如果不指定 name，则默认二进制日志文件名为主机名，后缀名为二进制日志的序列号，所在路径为数据库所在目录（datadir），如：

```
mysql> show variables like 'datadir';
+---------------+-----------------------------+
| Variable_name | Value     |
+---------------+-----------------------------+
| datadir       | /usr/local/mysql/data/ |
+---------------+-----------------------------+
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/;
total 2.1G
-rw-rw----  1 mysql mysql 6.5M Sep 25 15:13 bin_log.000001
-rw-rw----  1 mysql mysql   17 Sep 25 00:32 bin_log.index
-rw-rw----  1 mysql mysql 300M Sep 25 15:13 ibdata1
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile0
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile1
drwxr-xr-x  2 mysql mysql 4.0K May  7 10:08 mysql
drwx------  2 mysql mysql 4.0K May  7 10:09 test
```

这里的 bin_log.00001 即为二进制日志文件，我们在配置文件中指定了名字，所以没有用默认的文件名。bin_log.index 为二进制的索引文件，用来存储过往产生的二进制日志序号，在通常情况下，不建议手工修改这个文件。

二进制日志文件在默认情况下并没有启动，需要手动指定参数来启动。可能有人会质疑，开启这个选项是否会对数据库整体性能有所影响。不错，开启这个选项的确会影响性能，但是性能的损失十分有限。根据 MySQL 官方手册中的测试表明，开启二进制日志会使性能下降 1%。但考虑到可以使用复制（replication）和 point-in-time 的恢复，这些性能损失绝对是可以且应该被接受的。

以下配置文件的参数影响着二进制日志记录的信息和行为：

- ❑ max_binlog_size
- ❑ binlog_cache_size
- ❑ sync_binlog
- ❑ binlog-do-db
- ❑ binlog-ignore-db

- log-slave-update
- binlog_format

参数 max_binlog_size 指定了单个二进制日志文件的最大值，如果超过该值，则产生新的二进制日志文件，后缀名 +1，并记录到 .index 文件。从 MySQL 5.0 开始的默认值为 1 073 741 824，代表 1 G（在之前版本中 max_binlog_size 默认大小为 1.1G）。

当使用事务的表存储引擎（如 InnoDB 存储引擎）时，所有未提交（uncommitted）的二进制日志会被记录到一个缓存中去，等该事务提交（committed）时直接将缓冲中的二进制日志写入二进制日志文件，而该缓冲的大小由 binlog_cache_size 决定，默认大小为 32K。此外，binlog_cache_size 是基于会话（session）的，也就是说，当一个线程开始一个事务时，MySQL 会自动分配一个大小为 binlog_cache_size 的缓存，因此该值的设置需要相当小心，不能设置过大。当一个事务的记录大于设定的 binlog_cache_size 时，MySQL 会把缓冲中的日志写入一个临时文件中，因此该值又不能设得太小。通过 SHOW GLOBAL STATUS 命令查看 binlog_cache_use、binlog_cache_disk_use 的状态，可以判断当前 binlog_cache_size 的设置是否合适。Binlog_cache_use 记录了使用缓冲写二进制日志的次数，binlog_cache_disk_use 记录了使用临时文件写二进制日志的次数。现在来看一个数据库的状态：

```
mysql> show variables like 'binlog_cache_size';
+-------------------+-------+
| Variable_name   | Value |
+-------------------+-------+
| binlog_cache_size | 32768 |
+-------------------+-------+
1 row in set (0.00 sec)

mysql> show global status like 'binlog_cache%';
+-----------------------+--------------+
| Variable_name  | Value |
+-----------------------+---------------+
| binlog_cache_disk_use | 0     |
| binlog_cache_use     | 33553 |
+-----------------------+---------------+
2 rows in set (0.00 sec)
```

使用缓冲次数为 33 553，临时文件使用次数为 0。看来 32KB 的缓冲大小对于当前这个 MySQL 数据库完全够用，暂时没有必要增加 binlog_cache_size 的值。

在默认情况下，二进制日志并不是在每次写的时候同步到磁盘（用户可以理解为缓冲写）。因此，当数据库所在操作系统发生宕机时，可能会有最后一部分数据没有写入二进制日志文件中，这会给恢复和复制带来问题。参数 sync_binlog=［N］表示每写缓冲多少次就同步到磁盘。如果将 N 设为 1，即 sync_binlog=1 表示采用同步写磁盘的方式来写二进制日志，这时写操作不使用操作系统的缓冲来写二进制日志。sync_binlog 的默认值为 0，如果使用 InnoDB 存储引擎进行复制，并且想得到最大的高可用性，建议将该值设为 ON。不过该值为 ON 时，确实会对数据库的 IO 系统带来一定的影响。

但是，即使将 sync_binlog 设为 1，还是会有一种情况导致问题的发生。当使用 InnoDB 存储引擎时，在一个事务发出 COMMIT 动作之前，由于 sync_binlog 为 1，因此会将二进制日志立即写入磁盘。如果这时已经写入了二进制日志，但是提交还没有发生，并且此时发生了宕机，那么在 MySQL 数据库下次启动时，由于 COMMIT 操作并没有发生，这个事务会被回滚掉。但是二进制日志已经记录了该事务信息，不能被回滚。这个问题可以通过将参数 innodb_support_xa 设为 1 来解决，虽然 innodb_support_xa 与 XA 事务有关，但它同时也确保了二进制日志和 InnoDB 存储引擎数据文件的同步。

参数 binlog-do-db 和 binlog-ignore-db 表示需要写入或忽略写入哪些库的日志。默认为空，表示需要同步所有库的日志到二进制日志。

如果当前数据库是复制中的 slave 角色，则它不会将从 master 取得并执行的二进制日志写入自己的二进制日志文件中去。如果需要写入，要设置 log-slave-update。如果需要搭建 master=>slave=>slave 架构的复制，则必须设置该参数。

binlog_format 参数十分重要，它影响了记录二进制日志的格式。在 MySQL 5.1 版本之前，没有这个参数。所有二进制文件的格式都是基于 SQL 语句（statement）级别的，因此基于这个格式的二进制日志文件的复制（Replication）和 Oracle 的逻辑 Standby 有点相似。同时，对于复制是有一定要求的。如在主服务器运行 rand、uuid 等函数，又或者使用触发器等操作，这些都可能会导致主从服务器上表中数据的不一致（not sync）。另一个影响是，会发现 InnoDB 存储引擎的默认事务隔离级别是 REPEATABLE READ。这其实也是因为二进制日志文件格式的关系，如果使用 READ COMMITTED 的事务隔离级别（大多数数据库，如 Oracle，Microsoft SQL Server 数据库的默认隔离级别），会出现类似丢失更新的现象，从而出现主从数据库上的数据不一致。

MySQL 5.1 开始引入了 binlog_format 参数，该参数可设的值有 STATEMENT、

ROW 和 MIXED。

（1）STATEMENT 格式和之前的 MySQL 版本一样，二进制日志文件记录的是日志的逻辑 SQL 语句。

（2）在 ROW 格式下，二进制日志记录的不再是简单的 SQL 语句了，而是记录表的行更改情况。基于 ROW 格式的复制类似于 Oracle 的物理 Standby（当然，还是有些区别）。同时，对上述提及的 Statement 格式下复制的问题予以解决。从 MySQL 5.1 版本开始，如果设置了 binlog_format 为 ROW，可以将 InnoDB 的事务隔离基本设为 READ COMMITTED，以获得更好的并发性。

（3）在 MIXED 格式下，MySQL 默认采用 STATEMENT 格式进行二进制日志文件的记录，但是在一些情况下会使用 ROW 格式，可能的情况有：

1）表的存储引擎为 NDB，这时对表的 DML 操作都会以 ROW 格式记录。

2）使用了 UUID()、USER()、CURRENT_USER()、FOUND_ROWS()、ROW_COUNT() 等不确定函数。

3）使用了 INSERT DELAY 语句。

4）使用了用户定义函数（UDF）。

5）使用了临时表（temporary table）。

此外，binlog_format 参数还有对于存储引擎的限制，如表 3-1 所示。

表 3-1 存储引擎对二进制日志格式的支持情况

存储引擎	Row 格式	Statement 格式
InnoDB	Yes	Yes
MyISAM	Yes	Yes
HEAP	Yes	Yes
MERGE	Yes	Yes
NDB	Yes	No
Archive	Yes	Yes
CSV	Yes	Yes
Federate	Yes	Yes
Blockhole	No	Yes

binlog_format 是动态参数，因此可以在数据库运行环境下进行更改，例如，我们可以将当前会话的 binlog_format 设为 ROW，如：

```
mysql>SET @@session.binlog_format='ROW';
```

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

mysql>SELECT@@session.binlog_format;
+------------------------------+
| @@session.binlog_format |
+------------------------------+
| ROW                 |
+------------------------------+
1 row in set (0.00 sec)
```

当然，也可以将全局的 binlog_format 设置为想要的格式，不过通常这个操作会带来问题，运行时要确保更改后不会对复制带来影响。如：

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

mysql>SELECT @@global.binlog_format;
+------------------------------+
| @@global.binlog_format |
+------------------------------+
| ROW              |
+------------------------------+
1 row in set (0.00 sec)
```

在通常情况下，我们将参数 binlog_format 设置为 ROW，这可以为数据库的恢复和复制带来更好的可靠性。但是不能忽略的一点是，这会带来二进制文件大小的增加，有些语句下的 ROW 格式可能需要更大的容量。比如我们有两张一样的表，大小都为 100W，分别执行 UPDATE 操作，观察二进制日志大小的变化：

```
mysql>SELECT @@session.binlog_format\G;
*************************** 1. row ***************************
@@session.binlog_format: STATEMENT
1 row in set (0.00 sec)

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 106
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql>UPDATE t1 SET username=UPPER(username);
```

```
Query OK, 89279 rows affected (1.83 sec)
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

可以看到，在 binlog_format 格式为 STATEMENT 的情况下，执行 UPDATE 语句后二进制日志大小只增加了 200 字节（306–106）。如果使用 ROW 格式，同样对 t2 表进行操作，可以看到：

```
mysql>SET SESSION binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql>UPDATE t2 SET username=UPPER(username);
Query OK, 89279 rows affected (2.42 sec)
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql>SHOW MASTER STATUS\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 13782400
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

这时会惊讶地发现，同样的操作在 ROW 格式下竟然需要 13 782 094 字节，二进制日志文件的大小差不多增加了 13MB，要知道 t2 表的大小也不超过 17MB。而且执行时间也有所增加（这里我设置了 sync_binlog=1）。这是因为这时 MySQL 数据库不再将逻辑的 SQL 操作记录到二进制日志中，而是记录对于每行的更改。

上面的这个例子告诉我们，将参数 binlog_format 设置为 ROW，会对磁盘空间要求有一定的增加。而由于复制是采用传输二进制日志方式实现的，因此复制的网络开销也有所增加。

二进制日志文件的文件格式为二进制（好像有点废话），不能像错误日志文件、慢查询日志文件那样用 cat、head、tail 等命令来查看。要查看二进制日志文件的内容，必须通过 MySQL 提供的工具 mysqlbinlog。对于 STATEMENT 格式的二进制日志文件，在使用 mysqlbinlog 后，看到的就是执行的逻辑 SQL 语句，如：

```
[root@nineyou0-43 data]# mysqlbinlog --start-position=203 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
….
#090927 15:43:11 server id 1  end_log_pos 376   Query   thread_id=188   exec_
time=1     error_code=0
SET TIMESTAMP=1254037391/*!*/;
update t2 set username=upper(username) where id=1
/*!*/;
# at 376
#090927 15:43:11 server id 1  end_log_pos 403   Xid = 1009
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

通过 SQL 语句 UPDATE t2 SET username=UPPER（username）WHERE id=1 可以看到，二进制日志的记录采用 SQL 语句的方式（为了排版的方便，省去了一些开始的信息）。在这种情况下，mysqlbinlog 和 Oracle LogMiner 类似。但是如果这时使用 ROW 格式的记录方式，会发现 mysqlbinlog 的结果变得“不可读”（unreadable），如：

```
[root@nineyou0-43 data]# mysqlbinlog  --start-position=1065 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
……
# at 1135
# at 1198
#090927 15:53:52 server id 1  end_log_pos 1198  Table_map: 'member'.'t2' mapped
to number 58
#090927 15:53:52 server id 1  end_log_pos 1378  Update_rows: table id 58 flags:
STMT_END_F

BINLOG '
```

```
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
# at 1378
#090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

这里看不到执行的 SQL 语句，反而是一大串用户不可读的字符。其实只要加上参数 -v 或 -vv 就能清楚地看到执行的具体信息了。-vv 会比 -v 多显示出更新的类型。加上 -vv 选项，可以得到：

```
[root@nineyou0-43 data]# mysqlbinlog -vv  --start-position=1065 test.000004
……
BINLOG '
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
### UPDATE member.t2
### WHERE
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
###   @2='david' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
###   @3='family' /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
### SET
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
```

```
    ###   @2='DAVID' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
    ###   @3=family /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
    ###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
    ###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
    ###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
    ###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
    ###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
    ###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
    ###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
    # at 1378
    #090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
    COMMIT/*!*/;
    DELIMITER ;
    # End of log file
    ROLLBACK /* added by mysqlbinlog */;
    /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

现在 mysqlbinlog 向我们解释了它具体做的事情。可以看到，一句简单的 update t2 set username=upper(username)where id=1 语句记录了对于整个行更改的信息，这也解释了为什么前面更新了 10W 行的数据，在 ROW 格式下，二进制日志文件会增大 13MB。

3.3 套接字文件

前面提到过，在 UNIX 系统下本地连接 MySQL 可以采用 UNIX 域套接字方式，这种方式需要一个套接字（socket）文件。套接字文件可由参数 socket 控制。一般在 /tmp 目录下，名为 mysql.sock：

```
mysql>SHOW VARIABLES LIKE 'socket'\G;
*************************** 1. row ***************************
Variable_name: socket
        Value: /tmp/mysql.sock
1 row in set (0.00 sec)
```

3.4 pid 文件

当 MySQL 实例启动时，会将自己的进程 ID 写入一个文件中——该文件即为 pid 文件。该文件可由参数 pid_file 控制，默认位于数据库目录下，文件名为主机名 .pid：

```
mysql> show variables like 'pid_file'\G;
*************************** 1. row ***************************
Variable_name: pid_file
        Value: /usr/local/mysql/data/xen-server.pid
1 row in set (0.00 sec)
```

3.5 表结构定义文件

因为 MySQL 插件式存储引擎的体系结构的关系，MySQL 数据的存储是根据表进行的，每个表都会有与之对应的文件。但不论表采用何种存储引擎，MySQL 都有一个以 frm 为后缀名的文件，这个文件记录了该表的表结构定义。

frm 还用来存放视图的定义，如用户创建了一个 v_a 视图，那么对应地会产生一个 v_a.frm 文件，用来记录视图的定义，该文件是文本文件，可以直接使用 cat 命令进行查看：

```
[root@xen-server test]# cat v_a.frm
TYPE=VIEW
query=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
md5=4eda70387716a4d6c96f3042dd68b742
updatable=1
algorithm=0
definer_user=root
definer_host=localhost
suid=2
with_check_option=0
timestamp=2010-08-04 07:23:36
create-version=1
source=select * from a
client_cs_name=utf8
connection_cl_name=utf8_general_ci
view_body_utf8=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
```

3.6 InnoDB 存储引擎文件

之前介绍的文件都是 MySQL 数据库本身的文件，和存储引擎无关。除了这些文件外，每个表存储引擎还有其自己独有的文件。本节将具体介绍与 InnoDB 存储引擎密切相关的文件，这些文件包括重做日志文件、表空间文件。

3.6.1 表空间文件

InnoDB 采用将存储的数据按表空间（tablespace）进行存放的设计。在默认配置下会有一个初始大小为 10MB，名为 ibdata1 的文件。该文件就是默认的表空间文件（tablespace file），用户可以通过参数 innodb_data_file_path 对其进行设置，格式如下：

```
innodb_data_file_path=datafile_spec1[;datafile_spec2]...
```

用户可以通过多个文件组成一个表空间，同时制定文件的属性，如：

```
[mysqld]
innodb_data_file_path = /db/ibdata1:2000M;/dr2/db/ibdata2:2000M:autoextend
```

这里将 /db/ibdata1 和 /dr2/db/ibdata2 两个文件用来组成表空间。若这两个文件位于不同的磁盘上，磁盘的负载可能被平均，因此可以提高数据库的整体性能。同时，两个文件的文件名后都跟了属性，表示文件 idbdata1 的大小为 2000MB，文件 ibdata2 的大小为 2000MB，如果用完了这 2000MB，该文件可以自动增长（autoextend）。

设置 innodb_data_file_path 参数后，所有基于 InnoDB 存储引擎的表的数据都会记录到该共享表空间中。若设置了参数 innodb_file_per_table，则用户可以将每个基于 InnoDB 存储引擎的表产生一个独立表空间。独立表空间的命名规则为：表名 .ibd。通过这样的方式，用户不用将所有数据都存放于默认的表空间中。下面这台 MySQL 数据库服务器设置了 innodb_file_per_table，故可以观察到：

```
mysql>SHOW VARIABLES LIKE 'innodb_file_per_table'\G;
*************************** 1. row ***************************
Variable_name: innodb_file_per_table
       Value: ON
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/member/*
-rw-r-----   1 mysql mysql  8.7K   2009-02-24    /usr/local/mysql/data/member/
Profile.frm
-rw-r-----   1 mysql mysql 1.7G   9月 25 11:13 /usr/local/mysql/data/member/
Profile.ibd
-rw-rw----   1 mysql mysql 8.7K   9月 27 13:38 /usr/local/mysql/data/member/
t1.frm
-rw-rw----   1 mysql mysql   17M   9月 27 13:40 /usr/local/mysql/data/member/
t1.ibd
-rw-rw----   1 mysql mysql 8.7K   9月 27 15:42 /usr/local/mysql/data/member/
t2.frm
```

```
-rw-rw----  1 mysql mysql  17M  9月 27 15:54 /usr/local/mysql/data/member/
t2.ibd
```

表 Profile、t1 和 t2 都是基于 InnoDB 存储的表，由于设置参数 innodb_file_per_table=ON，因此产生了单独的 .ibd 独立表空间文件。需要注意的是，这些单独的表空间文件仅存储该表的数据、索引和插入缓冲 BITMAP 等信息，其余信息还是存放在默认的表空间中。图 3-1 显示了 InnoDB 存储引擎对于文件的存储方式：

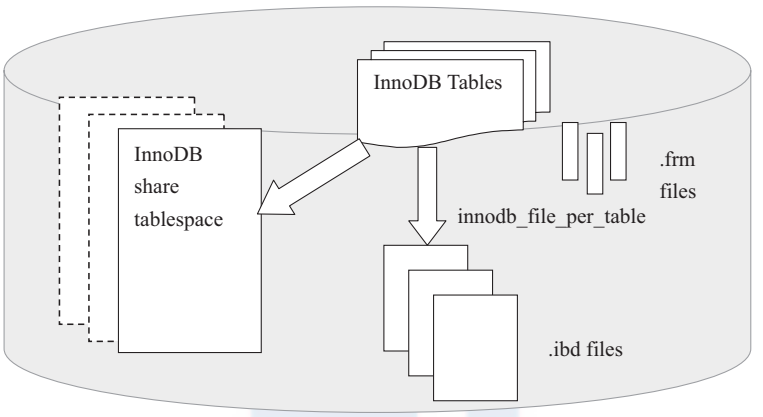


图 3-1 InnoDB 表存储引擎文件

3.6.2 重做日志文件

在默认情况下，在 InnoDB 存储引擎的数据目录下会有两个名为 ib_logfile0 和 ib_logfile1 的文件。在 MySQL 官方手册中将其称为 InnoDB 存储引擎的日志文件，不过更准确的定义应该是重做日志文件（redo log file）。为什么强调是重做日志文件呢？因为重做日志文件对于 InnoDB 存储引擎至关重要，它们记录了对于 InnoDB 存储引擎的事务日志。

当实例或介质失败（media failure）时，重做日志文件就能派上用场。例如，数据库由于所在主机掉电导致实例失败，InnoDB 存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。

每个 InnoDB 存储引擎至少有 1 个重做日志文件组（group），每个文件组下至少有 2 个重做日志文件，如默认的 ib_logfile0 和 ib_logfile1。为了得到更高的可靠性，用户可以设置多个的镜像日志组（mirrored log groups），将不同的文件组放在不同的磁盘上，以此提高重做日志的高可用性。在日志组中每个重做日志文件的大小一致，并以循环写

入的方式运行。InnoDB 存储引擎先写重做日志文件 1，当达到文件的最后时，会切换至重做日志文件 2，再当重做日志文件 2 也被写满时，会再切换到重做日志文件 1 中。图 3-2 显示了一个拥有 3 个重做日志文件的重做日志文件组。

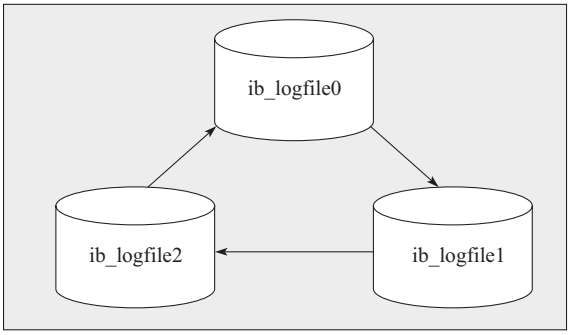


图 3-2 日志文件组

下列参数影响着重做日志文件的属性：

- innodb_log_file_size
- innodb_log_files_in_group
- innodb_mirrored_log_groups
- innodb_log_group_home_dir

参数 innodb_log_file_size 指定每个重做日志文件的大小。在 InnoDB1.2.x 版本之前，重做日志文件总的大小不得大于等于 4GB，而 1.2.x 版本将该限制扩大为了 512GB。

参数 innodb_log_files_in_group 指定了日志文件组中重做日志文件的数量，默认为 2。参数 innodb_mirrored_log_groups 指定了日志镜像文件组的数量，默认为 1，表示只有一个日志文件组，没有镜像。若磁盘本身已经做了高可用的方案，如磁盘阵列，那么可以不开启重做日志镜像的功能。最后，参数 innodb_log_group_home_dir 指定了日志文件组所在路径，默认为 ./，表示在 MySQL 数据库的数据目录下。以下显示了一个关于重做日志组的配置：

```
mysql>SHOW VARIABLES LIKE 'innodb%log%'\G;
……
*************************** 4. row ***************************
Variable_name: innodb_log_file_size
        Value: 5242880
*************************** 5. row ***************************
Variable_name: innodb_log_files_in_group
        Value: 2
*************************** 6. row ***************************
Variable_name: innodb_log_group_home_dir
        Value: ./
*************************** 7. row ***************************
Variable_name: innodb_mirrored_log_groups
        Value: 1
7 rows in set (0.00 sec)
```

重做日志文件的大小设置对于 InnoDB 存储引擎的性能有着非常大的影响。一方面

重做日志文件不能设置得太大，如果设置得很大，在恢复时可能需要很长的时间；另一方面又不能设置得太小了，否则可能导致一个事务的日志需要多次切换重做日志文件。此外，重做日志文件太小会导致频繁地发生 async checkpoint，导致性能的抖动。例如，用户可能会在错误日志中看到如下警告信息：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

上面错误集中在 InnoDB:ERROR:the age of the last checkpoint is 9433645，InnoDB:which exceeds the log group capacity 9433498。这是因为重做日志有一个 capacity 变量，该值代表了最后的检查点不能超过这个阈值，如果超过则必须将缓冲池（innodb buffer pool）中脏页列表（flush list）中的部分脏数据页写回磁盘，这时会导致用户线程的阻塞。

也许有人会问，既然同样是记录事务日志，和之前介绍的二进制日志有什么区别？

首先，二进制日志会记录所有与 MySQL 数据库有关的日志记录，包括 InnoDB、MyISAM、Heap 等其他存储引擎的日志。而 InnoDB 存储引擎的重做日志只记录有关该存储引擎本身的事务日志。

其次，记录的内容不同，无论用户将二进制日志文件记录的格式设为 STATEMENT 还是 ROW，又或者是 MIXED，其记录的都是关于一个事务的具体操作内容，即该日志是逻辑日志。而 InnoDB 存储引擎的重做日志文件记录的是关于每个页（Page）的更改的物理情况。

此外，写入的时间也不同，二进制日志文件仅在事务提交前进行提交，即只写磁盘一次，不论这时该事务多大。而在事务进行的过程中，却不断有重做日志条目（redo entry）被写入到重做日志文件中。

在 InnoDB 存储引擎中，对于各种不同的操作有着不同的重做日志格式。到 InnoDB 1.2.x 版本为止，总共定义了 51 种重做日志类型。虽然各种重做日志的类型不同，但是它们有着基本的格式，表 3-2 显示了重做日志条目的结构：

表 3-2 重做日志条目结构

redo_log_type	space	page_no	redo_log_body

从表 3-2 可以看到重做日志条目是由 4 个部分组成：

- redo_log_type 占用 1 字节，表示重做日志的类型
- space 表示表空间的 ID，但采用压缩的方式，因此占用的空间可能小于 4 字节
- page_no 表示页的偏移量，同样采用压缩的方式
- redo_log_body 表示每个重做日志的数据部分，恢复时需要调用相应的函数进行解析

在第 2 章中已经提到，写入重做日志文件的操作不是直接写，而是先写入一个重做日志缓冲（redo log buffer）中，然后按照一定的条件顺序地写入日志文件。图 3-3 很好地诠释了重做日志的写入过程。

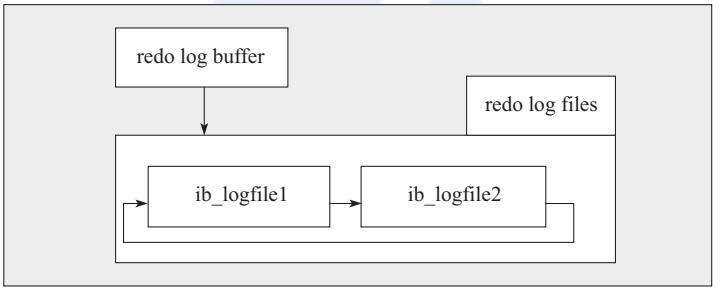


图 3-3 重做日志写入过程

从重做日志缓冲往磁盘写入时，是按 512 个字节，也就是一个扇区的大小进行写入。因为扇区是写入的最小单位，因此可以保证写入必定是成功的。因此在重做日志的写入过程中不需要有 doublewrite。

前面提到了从日志缓冲写入磁盘上的重做日志文件是按一定条件进行的，那这些条件有哪些呢？第 2 章分析了主线程（master thread），知道在主线程中每秒会将重做日志缓冲写入磁盘的重做日志文件中，不论事务是否已经提交。另一个触发写磁盘的过程是由参数 innodb_flush_log_at_trx_commit 控制，表示在提交（commit）操作时，处理重做日志的方式。

参数 innodb_flush_log_at_trx_commit 的有效值有 0、1、2。0 代表当提交事务时，并不将事务的重做日志写入磁盘上的日志文件，而是等待主线程每秒的刷新。1 和 2 不同的地方在于：1 表示在执行 commit 时将重做日志缓冲同步写到磁盘，即伴有 fsync 的调用。2 表示将重做日志异步写到磁盘，即写到文件系统的缓存中。因此不能完全保证在执行 commit 时肯定会写入重做日志文件，只是有这个动作发生。

因此为了保证事务的 ACID 中的持久性，必须将 innodb_flush_log_at_trx_commit 设置为 1，也就是每当有事务提交时，就必须确保事务都已经写入重做日志文件。那么当数据库因为意外发生宕机时，可以通过重做日志文件恢复，并保证可以恢复已经提交的事务。而将重做日志文件设置为 0 或 2，都有可能发生恢复时部分事务的丢失。不同之处在于，设置为 2 时，当 MySQL 数据库发生宕机而操作系统及服务器并没有发生宕机时，由于此时未写入磁盘的事务日志保存在文件系统缓存中，当恢复时同样能保证数据不丢失。

3.7 小结

本章介绍了与 MySQL 数据库相关的一些文件，并了解了文件可以分为 MySQL 数据库文件以及与各存储引擎相关的文件。与 MySQL 数据库有关的文件中，错误文件和二进制日志文件非常重要。当 MySQL 数据库发生任何错误时，DBA 首先就应该去查看错误文件，从文件提示的内容中找出问题的所在。当然，错误文件不仅记录了错误的内容，也记录了警告的信息，通过一些警告也有助于 DBA 对于数据库和存储引擎进行优化。

二进制日志的作用非常关键，可以用来进行 point in time 的恢复以及复制（replication）环境的搭建。因此，建议在任何时候时都启用二进制日志的记录。从 MySQL 5.1 开始，二进制日志支持 STATEMENT、ROW、MIX 三种格式，这样可以更好地保证从数据库与主数据库之间数据的一致性。当然 DBA 应该十分清楚这三种不同格式之间的差异。

本章的最后介绍了和 InnoDB 存储引擎相关的文件，包括表空间文件和重做日志文件。表空间文件是用来管理 InnoDB 存储引擎的存储，分为共享表空间和独立表空间。重做日志非常的重要，用来记录 InnoDB 存储引擎的事务日志，也因为重做日志的存在，才使得 InnoDB 存储引擎可以提供可靠的事务。