

架构至少能在两年里应付负载的能力。

这里有一些问题，你可以在做伸缩性规划时问问你自己：

- 应用架构是否已经具备了完整的功能？我们建议的许多伸缩性解决方案在某些应用体系里很难实现。如果你应用中一些核心体系还没实现，那就很难看出你要怎么把他们做成具备伸缩性的。同样地，在看到这些体系怎么真正运转起来之前，你也很难为此提供一个伸缩性的解决方案。
- 期望的负载峰值是多少？你的应用需要在峰值负载的时候也能正常工作。如果首页就像是 Yahoo! News 和 Slashdot 一样，应用还能运行吗？即使应用不是这类大众化的网站，你也仍然会有个负载峰值。举例来说，你是个在线零售商，假日期间就是应用达到负载峰值的时候，尤其那些个“恶名昭著”的在线购物的旺季里（圣诞节的前几个星期），在美国，母亲节前的周末就是网上花店的峰值时间。
- 如果系统中的每一部分都要处理负载，那么如果其中一部分故障时会发生什么？比如说，使用复制从服务器（Replication Slaves）来分担你的“读取负载”，当其中一个宕机时，你的系统还能正常处理吗？还是需要关闭部分功能来保持原来的性能？在此，你可以预先准备一些闲余的负载量来缓解这种紧急情况。

9.2.2 在实现伸缩性之前“买”些时间

Buy some Time Before You Buy

在一个完美的世界里，你能够做到计划先行、有足够的开发人员、有花不完的预算等。但在现实世界里，事情就复杂了。你在实现应用的伸缩性时总要做一些妥协，典型的是你常常要把系统的大改动推迟一段时间再实行。在我们开始深入到 MySQL 伸缩性的细节上之前，你要做一些实现伸缩性之前的准备工作：

性能优化

很多时候你对系统的轻微调整都能带来性能上的巨大飞跃，比如正确地建立表索引、更换为别的存储引擎。如果你正面临着性能的极限，首先要做的是开启查询日志，并分析那些运行缓慢的查询，看看哪些查询你还可以做些优化。更多内容请参阅第 64 页的“查询的日志”。

此类调整的回报总会越来越小。在修复了大多数主要问题后，通过查询优化取得的性能提升将会越来越难。每一次新的优化都是事倍功半，反而让你的应用越来越复杂。

购买性能更强的硬件设备

升级你的服务器，或者增加服务器，有时是个有效的办法。特别是对于处于软件生命周期早期的那些应用，购买更多的服务器是个好主意。与之相对应的是，尽量让应用运行在单独一台服务器上。尽管一个漂亮、优雅的设计可以实现这种可能，但是当你需要三个人，花一个月来实现这个设计时，你大概就发现这还不如多买几台服务器来得实用，尤其是在你时间不够用、开发人员不够多的情况下。

当应用比较小，或者开始就是被设计用来分布到各个硬件上时，多购置几台服务器是个行之有效的办法。对于刚上线的应用，效果就很明显，因为它比较小，或者设计比较合理。对于那些更大、更老的应用，购买更多硬件的办法可能就不怎么灵光了，或者说就很费钱了。比如说，服务器从一台增加到三台，这是小事一桩；如果服务器从 100 台增加到 300 台那就是另外一回事了——它们很烧钱的。这样的话，在现有的系统之上多花一些时间和努力来提升性能就很值得了。

9.2.3 向上扩展

Scaling Up

升级能在一段时间内产生效果，但是当你的应用变得越来越大时，这办法就不顶用了。

第一个原因是钱。无论在你的服务器上运行的是什么软件，升级是个糟糕的财务决策。市面上是有一批性价比极好的硬件设备，但是在这个范围之外，硬件设备就变得很独特很不同寻常，相应地也很贵。这就意味着你的升级有个事实上的上限。

从经济角度来看，MySQL 本身在垂直方向并不具备很好的升级条件，因为它难以有效利用多 CPU、多磁盘的硬件环境。应用对硬件设备的有效利用程度取决于你的工作量、使用的硬件和运行的操作系统。大致上说，对于目前版本的 MySQL（注 1），8 个 CPU、14 个磁盘是它的极限了。很多人在达到这个硬件限制之前问题就冒出来了。

415 即使你的主服务器能够有效利用多 CPU，但从服务器几乎不可能比主服务器拥有更强的性能。因为从服务器线程无法充分利用其多 CPU、多磁盘，所以在同样硬件环境下，主服务器能够更轻松地处理沉重的负载。

此外，你不可能无限制地升级，哪怕再强的电脑也有极限。单台服务器的应用往往是先达到它读取性能的极限，尤其是在执行复杂的读取查询时。这些查询在 MySQL 都是以单线程执行的，它们只能使用一个 CPU，所以，金钱也无法帮它们买到更高的性能。目前，最快的服务器级 CPU 也仅仅比商用级 CPU 快两倍，增加更多的 CPU 或 CPU 核也无助于提高那些缓慢查询的运行速度。当查询到的数据结果变得很大，以至于缓存也放不下它们时，服务器也就面临着存储空间的极限了。这也就是常说的现代计算机里磁盘是使用率最高，但也是访问最慢的组成部分。

应用的伸缩性也常常是个问题。由工作量导致的设计方案的选择和局限性制约着应用能否有效地利用多个硬件设备。

基于这些原因，我们建议你不要通过升级来取得伸缩性，至少不能企图无限制地升级。如果你事先知道应用会增长到很大，可以在实行一个更好的伸缩性方案前，先购买一个更强大的服务器来对付一段时间。然而，通常说来，最后还是得通过扩容来解决这个问题。这就是我们在下一节中要讲的内容。

9.2.4 向外扩展

Scaling Out

最常见且最简单的向外扩展方法是把你的数据通过集群的办法分散（Partition）到各个服务器上去，然后使用从服务器来处理读取查询。这个技术方案适用于偏重数据读取的应用。它的缺点是重复缓存，但是如果数据不大的话，这也不算是个严重的问题。这方面的内容我们在前面的章节里已经谈到过，后面还会继续讲到。

向外扩展的另外一个常见的方法是将工作量分布到各个节点上。怎么分割你的工作量是个复杂且颇有难度的决定。回想下上面提过的理想化系统，它能无限制地伸缩——这不是能用 MySQL 轻易做到的。许多大型的 MySQL 应用无法自动分解，至少无法彻底地分解。在这个篇幅里，我们就看一下各种分解的可能性，以及它们的强项和弱项。

注 1：只要你使用 64 位的操作系统和硬件，无论多大的内存都不是问题。但是其中还有些限制，只是看上去不太明显也不严重。我们已经在第 7 章中讨论过内存使用长度了。

一个节点就是你的 MySQL 架构中的一个功能单元。如果你没有做过冗余和高可用性规划，那么它就是一台服务器。如果你正在设计一个带有容错能力的冗余系统，那一个节点通常会由以下几种情况组成。

- 主—主双机拓扑结构，由一台主动服务器和一台被动复制从服务器组成。
- 一台主服务器，和许多台从服务器。
- 一台主动服务器，并有一个分布式数据块复制设备（Distributed Replicated Block Device, DRBD）做备用。
- 一个基于存储区域网络（Storage Area Network, SAN）的集群。

416

在许多案例里，所有服务器都在一个节点上就应该有相同的数据。我们倾向于在两台服务器的主动—被动节点上采用主—主双机架构。更多内容请查看本书第 365 页的“主动—被动模式下的主—主配置”。

功能拆分

功能拆分，或者说职责拆分意味着由各个节点来完成不同的功能。我们在上文中已经提到过一些方法，比如前一章讲到了针对 OLTP 和 OLAP 的工作量怎么来设计不同的服务器。相比于把单独的服务器或节点分配给不同的应用，功能拆分采取的策略往往要比它们更进一步，前者只要在服务器或节点上保留这个应用所需的特定的数据就行了。

在这里，我们多次使用到了“应用”这个词，它所指的不是一个单独的电脑程序，而是相关的一系列程序，它们中的每一个都是能被轻易地相互分离出来并独立运行的程序。举个例子来说，如果有个无需共享数据的网站，可以按照功能领域把它们分离开：门户将各个功能整合起来；通过门户能浏览网站新闻、登录论坛、寻找帮助支持，以及查阅知识库等。每一个板块的数据都可以被存放在一个独立的 MySQL 服务器里。图 9-1 描述了这个组织方式。

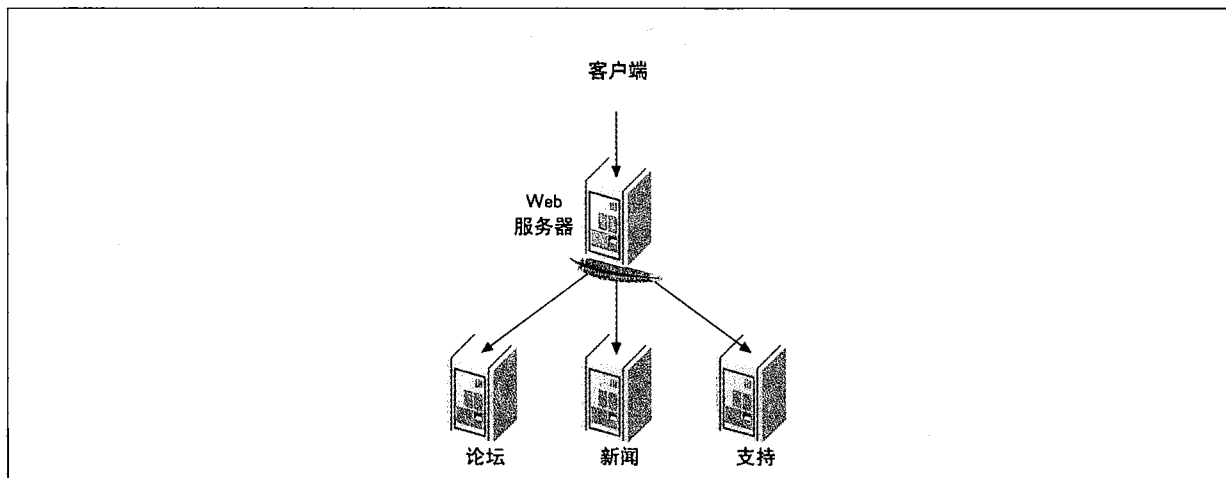


图 9-1：一个门户及 3 个专职于各功能域的点

如果应用的规模非常巨大，那每一个功能区域都可以有其专属的 Web 服务器，但是这种情形并不多见。

417

另一种可能的功能划分方式是把一个单独应用的整块数据按照他们关联程度，划分成几个不会相互关联的集合。当必须要关联的时候，对于那些对性能要求不高的表单，可以在应用中或者采用 Federated 表来做一些关联。这个做法在实际使用中存在着一些不同的“变种”，但它们有个共同属性，就是每一类的数据都可以在一个单独的节点上被找到。数据的划分没有一个通行的方法来做，因为这本身就很难高效地完成，任何一个方法与其他方法相比较，也不会有突出的优点。

最后，你也不可能无限制地通过功能划分来扩展，因为如果每个功能区都紧密绑定在一个 www.zzbaike.com 就只能在垂直方向上进行扩展。假如有一个应用或一个功能区最终会变得过于庞大，那本教程由站长百科收集整理略了，如果你在功能分区上已经走得很远了的话，那它就更难于采用更具有扩展性的设计了。

数据分块

在目前用于扩展大型 MySQL 应用的技术中，数据分块（注 2）是一种最通用且成功的方法。把数据切成一小片，或者说一小块，然后存储在不同的 MySQL 节点上。

分块在功能分割合并时非常好用，但许多标准系统中，总有一些“全局”数据根本无法被分块（比如城市名单）。这些全局数据可以放在一个单独节点上，通常是被加载在一个像 memcached 这样的缓存里。

实际上，许多应用只将其所需要的数据分离出来使用——很显然，整个数据集的这些部分都会增长到巨大的规模。假设你正在构建一个博客服务系统，预计会有 1000 万个用户，这时也可以不把用户注册信息部门分割出来，因为你不会把所有用户（或者是其中活跃的那部分用户）的信息都放在内存里。假如用户数将达到 5 亿，那么你就可能需要数据分块了。用户产生的那些内容，如文章和评论，都需要提取出来放在同一个地方，因为它们现在已经很多了，并且还将更多。

大型应用可能有多个逻辑数据集，处理它们的方法各不相同。你可以把它们放在不同的服务器组上，但未必一定要这么做。同一份数据的分块方式有好多种，具体要根据你访问数据的方式来决定。在下文中，我们会举例说明。

418 分块技术与许多应用的设计理念有着显著的差异，这使得一个应用难以从单块数据存储更改为数据分块式的架构。如果在应用设计时就已经预计到分块，那做起来就容易多了。

许多应用在开始设计时都不是用分块来处理将来的规模扩大的情形。举例来说，你在博客服务系统中使用复制来扩展读查询，直到某一天，它不再奏效了。这时，你就把服务分成三个部分：用户信息、文章和评论，然后，放到不同的服务器上（功能分区），最后，在你的应用中把三者综合起来。图 9-2 显示的是从单一服务器演进为功能分区模式。

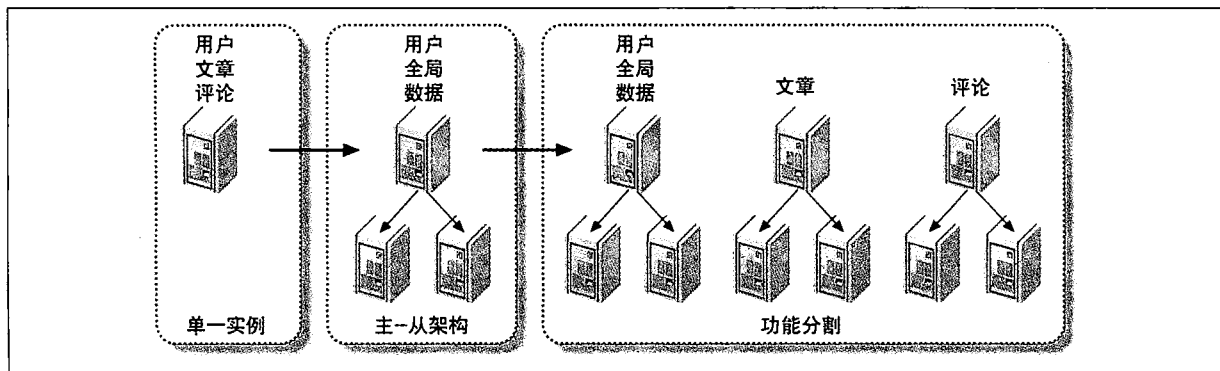


图 9-2：从单一服务器到功能分割的数据存储方式

注 2：“分块”也可以叫做“分裂”、“分割”，在这里我们就只使用“分块”来避免词义混淆。Google 一直把它叫做“分块”，如果 Google 觉得它名副其实的话，那对我们的文章而言也正合适。

最后，你还能按用户 ID 对文章和评论来进行分块，把用户信息放在一个单独的节点上。如果沿 www.zzbaike.com 的配置来做全局节点，用主—主成对的方式来做那些数据分块的节点，这样的数据存储方式如图 4.19 本教程由站长百科收集整理

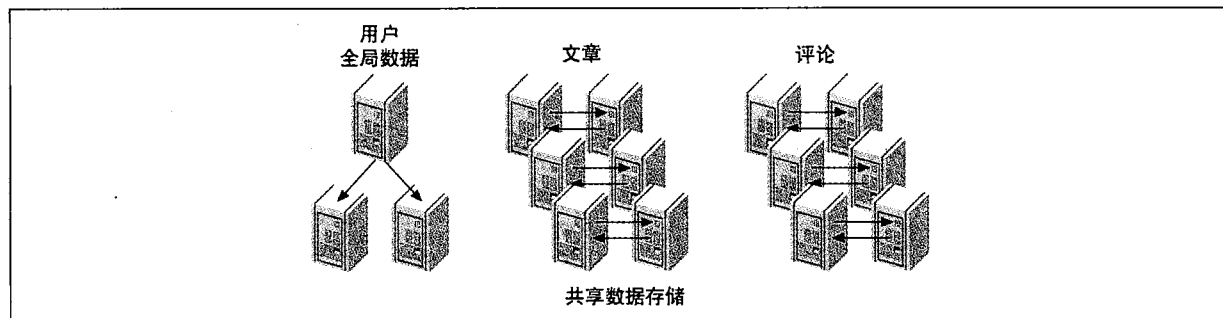


图 9-3: 1 个全局节点+6 个主—主节点的数据存储方式

如果你事先知道应用会扩展到很大的规模，而且也知道功能分区的局限性，就可以跳过这些中间步骤，直接将单一节点升级为数据分块存储方式。

采用数据分块的应用经常有一个数据访问抽象库，它降低了应用与分块数据存储之间的通信复杂度，但无法隐藏所有对分块数据存储的操作，因为应用一般都知道分块数据存储无法执行哪个查询。

太多的抽象会导致效率低下。比如在所有节点中查询一个数据，这数据其实只在一个节点上存在。这就是 MySQL 的 NDB Cluster 存储引擎在 Web 应用中表现差劲的原因之一：它隐瞒了这样一个事实，就是它查询了很多节点，但看起来像是只查询一台服务器。

数据分块存储看起来是个优雅的设计，但实际上很难构建。那为什么还要选择这个架构呢？答案很简单：如果你想扩展应用的写操作的负载能力，必须切分数据。如果只有一台单独的主机，不管从服务器有多少，你都没法扩展写操作的负载能力，对于上述的缺点而言，数据分块正是一个首选的解决方案。

如果存在着一个彻底自动的、高性能的、透明的数据分块方法，并且执行起来跟在一台服务器上一样，那真的是很棒，但是，实际上它还不存在。在将来，MySQL 的 NDB Cluster 存储引擎在这个目标方向上会运行得更快更健壮。

选择分割键

数据分块最重要的挑战是发现和获取数据。你打算怎么对数据分块就怎么去找数据。找数据的方法有很多，其中几个比其他的更好用一些。

数据分块的目标是最重要最常用的查询数据做尽量少的分块。因此，这个过程中有个很重要的方面：怎样为你的数据选择一个（或多个）分割键。分割键决定了数据中的每一行将被分配到哪个数据分块中。如果你知道一个对象的分割键，就回答以下两个问题：

- 这个数据放在哪里？
- 我需要读数据时，从哪里可以获得？

在下文中，将为你展示选择和使用分割键的几种方式。而现在，让我们一起看个示例。假如我们像 MySQL NDB Cluster 一样来操作，用一张由所有表单的主键组成的散列表来对数据进行分块，并以此建立数据分块。这是个很简单的方法，但是它的扩展性很不好，因为它经常要查找所有数据分块来获取你需要的数据。比如说，你要

读取用户 3 的所有博客文章，在哪里可以找到它们呢？这些文章数据很可能被分散到了所有数据分块中，因为它们是按数据的主键值来分割的，而不是按用户。

420

跨分块的查询比单一分块上的查询要糟糕，但是，只要交叉的数据分块不太多，它的情况也不是那么糟。最糟的情况是你不知道你需要的数据放到哪个分块上了，那样你只能扫描每一个数据分块来查找。

一个好的分割键往往是数据库里一个很重要实体的 ID。这些 ID 可以用来区分分块单元。比如你使用用户 ID 或客户 ID 来分割数据，那么数据分块就是用户或客户。

寻找分割键的一个好办法是为你的数据模型画一个实体—关系图，或者使用相应的工具来显示数据模型里的实体和它们之间的关系。把这个关系图全部显示出来，相关的实体尽量地靠近显示。不要只是盯着图看看，而是要仔细检视这个图表，根据应用要执行的查询要求，寻找可作为分割键的字段。如果其中有两个实体之间有着关联，但是应用很少甚至从不使用这个关联，那么就可以断开这个关联以实现所需要的分块。

总有一些数据模型更易于实现数据分块，这取决于实体—关系图中那些实体的连通程度。图 9-4 里左边是一个易于实现分块的数据模型，右边那个则难以实现。

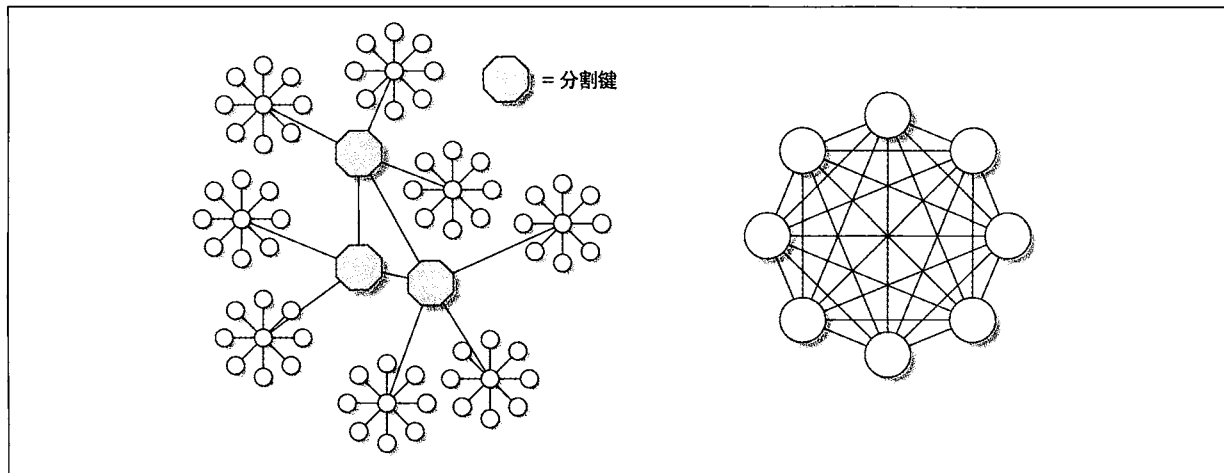


图 9-4：两个数据模型，一个易于分块，另一个则难以分块（注 3）

左边那个数据模型易于分块是因为那些子图之间只有单一连接，你可以从子图间的连接来“切割”数据。右边那个数据模型难以分块是因为它没有所谓的子图。许多数据模型都类似于左边的那个，而右边那种比较少。

421

多个分割键。复杂的数据模型会使数据分割变得困难。许多应用不止一个分割键，特别是当数据中存在两个或更多个“维度”的时候。换句话说，应用需要从多个角度看到一个有效的连贯的数据视图。这就意味着需要在系统里存储某些数据至少两份。

举例说，你需要把博客应用分别用两个方式分块：按用户 ID 和按文章 ID，因为这两个都是应用查找数据的常用方式。试着想象这种情形：你经常要看某个用户发布的所有文章，或者某个文章及其所有评论。如果数据只是根据用户来分块，你就难以查看某文章的所有评论；同样地，如果只是根据文章来分块，就难以查看某用户的所有文章。如果要在一个分块上同时做这两种类型的查询，就得从对应的两个方向上进行分块。

你用不着仅仅为了多个分割键而去设计一个双重冗余数据存储。让我们看一下另外一个例子：一个社交网站下

注 3：感谢 HiveDB 项目和 Britt Crawford 提供这些优雅的图表。

面的读书俱乐部站点，这个站点上的用户能够对书进行评论。站点能够显示任何一本书的所有评论，并显示任何一个用户所读过的书和发表过的评论。

你可以为用户数据建立一个分块数据存储，同时为书本数据建立一个分块数据存储。每一条评论既有用户 ID 又有书本 ID，这就跨越了两个数据分块的边界。无须在每个数据分块中存储一份评论数据，可以把这些评论数据都放在用户数据中，然后把每一条评论的标题和 ID 放在书本数据中。这样正好用来呈现一本书的所有评论，而不必同时访问两个数据分块，直到显示评论的全部内容时，你才需要到用户数据分块中去读取。

跨分块查询

许多基于数据分块的应用总有一些查询需要聚合或联接多个数据分块来完成。比如说，如果读书俱乐部网站要显示最受欢迎或最活跃的用户，它就必须访问每一个数据分块。要让这类查询也能很好的执行，是实现数据分块的最困难的部分。因为从应用层面来看，一条单独的查询会被拆分成多条并行执行的查询语句，每个数据分块一条。一个好的数据库抽象层可以帮你减少这种痛苦，但是这样的查询总归是缓慢的，并且比非数据分块查询昂贵得多，后者有主动缓存可供使用。

某些编程语言，例如 PHP，对于多条查询的并行执行没有很好的支持。对于这种情况，通常的做法是用 C 或 Java 写一个辅助程序，用来执行这类查询和汇集结果。这样 PHP 就可以向这个辅助程序（往往是个 Web 服务）发起查询。

跨分块查询也可以借助汇总表来进行。你遍历所有数据分块并把遍历结果作为冗余数据存放在该分块上。如果觉得同样的数据在一个分块上放两份很浪费的话，可以把汇总表放到一个另外的数据存储上，这样他们就只有一份了。非分块的数据总是被放在一个全局的节点上，并由缓存来分担其负载。

当数据的分布性极其重要，或者实在找不出合适的分割键的时候，一些应用就会采用随机分块的方式。分布式搜索应用就是个很好的例子，在这个应用里跨分段数据的查询和聚合变得很寻常，而不是偶然发生的。

跨分块的查询还不是数据分块要面对的唯一难题，保证数据的一致性也比较困难。外键在各分块之间已经不起作用了，对此的解决办法是在应用中根据需要随时检查其一致性。还有一个可能的办法是使用 XA 事务处理，但这会使经费超支，所以不常用。更多内容请查看 262 页的“分布式 (XA) 事务处理”。

你也可以设计一个间歇执行的清理进程来做数据一致性的事情。比如，如果读书俱乐部网站上一个用户的账号超期了，用不着立即把它移除，你可以写一个周期性执行的任务去删除这个用户在书本数据分块上的所有评论。也可以编写一个检查脚本，定期执行，来确保各个各数据分块之间的数据一致性。

分配数据、数据分块和节点

数据分块与节点间不一定非得是一一对应的关系。一个比较好的主意是把数据分块的大小做得比节点的容量小得多，这样，在一个节点上就可以放多个数据分块了。

数据分块的小块化有助于提高数据的可管理性，备份与恢复将更易于操作，如果表都比较小，那么像格式更改这样的操作就会更加方便。举个例子，你有一个表，其中包含了 100GB 的数据，你可以把它整个地，或者分割成 100 份，每份 1GB 的数据分块放在一个单独的节点上。现在，假设你想往表里添加索引，这样在整个 100GB 的数据分块上耗费的时间肯定是多于在 100 个 1GB 的数据分块上耗费的时间总和，因为 1GB 的数据分块刚好能全部放入到内存进行操作。还有，当你对表做 ALTER TABLE 操作时需要将数据进行冻结，冻结 1GB 的数据要比冻结全部 100GB 的数据要好很多。

小块的数据分块也更易于移动，更易于在节点中间重新分配负载能力，重新配置负载平衡。是个效率低下的过程。你往往要将这个数据分块设置为只读模式（这个功能得先在你的应用实现中），然后把数据分离出来，再把它们放入另外一个节点。在这些步骤里，我们要使用 `mysqldump` 来导出数据，然后使用 `mysql` 来重新加载它们。（如果你使用的数据库格式是 `myisam`，那么可以直接复制对应的数据文件；更多内容请参考第 11 章）。

423 除了移动数据分块之外，也需要考虑在数据分块之间移动数据的事情，最好是不要中断服务以免影响整个应用的运行。如果你的数据分块很大，那在移动一个数据分块里的全部数据时，就很难保持分块间的负载均衡，所以，需要采用另外的办法来移动其中一条单独的数据（比如一个用户的数据）。在数据分块间移动数据比移动数据分块要复杂得多，因此，尽可能不要去移动数据，这也是我们一直强调的数据分块尺寸可管理性的原因。

数据分块的相对尺寸取决于应用的需要。一个粗略的指导思想是：对我们而言的可管理尺寸的意思是数据表的尺寸应该小到能在 5 到 10 分钟内完成执行那些常规的维护指令，像 `ALTER TABLE`、`CHECK TABLE` 或 `OPTIMIZE TABLE` 等。

如果数据分块太小，那你一定会遇上了很多很多的表，太多的表会引起文件系统或 MySQL 内部结构上的问题。这方面内容可参考第 279 页的“表缓存”。过于小的数据分块也会产生大量的跨分块查询。

在节点上布置数据分块。你可能要确定如何在一个节点上部署数据分块。这里有一些常用的方法：

- 每个数据分块使用一个单独的数据库，每个数据库都是用同一个名称。当你应用结构做镜像的时候，这个方法就很有用，在生成许多应用实例时，每个实例只会被关联到一个数据分块。
- 把表从几个数据分块放到一个数据库中，把数据分块的号码写在表的名称里（比如 `bookclub.comments_23`）。通过配置，一个单独的数据库可以支持多个数据分块。
- 每个数据分块都有一个独立的数据库，并把所有应用的表都放在数据库里。把数据分块的号码写在数据库名称里（比如 `bookclub_23.comments`，`bookclub_23.users` 等）。这个方法适用于应用连接到独立的数据库时不会在其查询里指定数据库名称。这样做的好处是用不着针对每一个数据分块的查询分别做个性化处理，也更便于应用从数据分块迁移到单独数据库。
- 每个数据分块都使用独立的数据库。把数据分块的号码既写在数据库名里，又写在表的名称里（比如 `bookclub_23.comments_23`）。

如果你把数据分块的号码写在表名里了，那就需要在查询语句模板里插入一个数据分块的号码。常见的办法是在查询模板里使用神奇的通配符，像 `sprintf()` 这样的格式化函数里的 `%s`，或者用字符串插入技术。在 PHP 里可以像下面这样做：

```
$sql = "SELECT book_id, book_title FROM bookclub_%d.comments_%d... ";
$res = mysql_query(sprintf($sql, $shardno, $shardno), $conn);
```

你也可以只使用字符串插值的方法来做。

```
$sql = "SELECT book_id, book_title FROM bookclub_$shardno.comments_$shardno ...";
$res = mysql_query($sql, $conn);
```

424 在新编写的应用里这样做很容易，但是对于已经运行的应用而言，麻烦就大了。在构建新应用的时候，查询模板不是个问题，我们倾向于每个独立的数据存放一个数据分块，并把数据分块的号码既写在数据库名里又写在表名里。这样的话，执行 `ALTER TABLE` 这种操作会变得复杂，但是也有它特有的优点：

- 如果整个数据分块放在一个独立的数据库里，那使用 `mysqldump` 就可以很方便地移动它。
- 因为数据库在文件系统里是一个目录，那管理这些分块数据的文件就很容易。
- 因为数据分块间互不关联，这样就很方便查看数据分块的规模大小。
- 全局唯一的表名可以避免犯错。整个数据分块范围内的表名是唯一，当你不是在相应的数据分块上做查询时，或者把一个数据分块的数据插入到另外一个数据分块时，都会很快被发现。

现在你可能想确认下应用是否具有一些“亲数据分块”的特性。你可能受益于将一些数据分块放在一起（放在同一台服务器上、同一个网段里、同一个数据中心，或者是同一个交换网络里）去探究其中的数据访问模式。比如说，把数据按照用户来分块，然后再把用户按国家分成几块放在同一个节点上。

对已有应用做数据分块的结果往往是每个节点一个数据分块。这样的简化可以限制对应用里所有查询要做的修改的范围。分块对应用而言就是分裂性的改变，就是对应用的每一部分进行尽可能的简化。如果你在做分块后，每个节点看起来都是整个应用数据的缩微图，那就不必去修改应用里的那些查询了，也用不着费心把那些查询定向到相应的节点上去。

固定分配

给数据分块分配数据有两种方法：**固定分配**和**动态分配**。这两种方法都需要一个分配函数，它读入一行的分配键，然后返回这行数据该被放置的数据分块（注 4）。

固定分配使用的分配函数只依赖于一个分配键，散列函数和模块就是很好的例子。在这些函数里，每个输入值都通过一个分配键被映射到有限数量的存放数据的“桶”里去。

假设你有 100 个“桶”，想找出哪个“桶”来存放用户 111 的数据。如果使用取模的方法，答案就很简单：111 用 100 取模等于 11，所以你要把用户数据放在 11 号数据分块中。

如果你使用 `CRC32()` 函数的方式来做散列，那答案就是 81。

```
mysql> SELECT CRC32(111) % 100;
+-----+
| CRC32(111) % 100 |
+-----+
|                81 |
+-----+
```

固定分配的主要好处是简单、开销低，可以把它硬编码在应用里。

然而，固定分配也存在着一些缺点：

- 如果数据分块很大，并且数量不多，这样就很难在分块间做平衡。
- 固定分配使得你无法决定每一份数据放在哪个分块上，这对于那些在数据分块上没有一致负载的应用很重要。总有一些数据分块比另外一些更“活跃”，如果这些活跃的数据分块恰好是被放在同一个数据分块里，那么固定分配也无法帮你把它们中的一些转移到别的数据分块上。当你的数据都被分成一小份一小份地放在每个数据分块上时，这个问题不会出现，因为大数据量才会让一些稀有的问题暴露出来。

注 4：这里我们使用了具有数学意味的“函数”，用来指代从一个输入值（域）映射到一个输出值（范围）的过程。接下来你会看到，可以用很多方法来创建这样一个函数，包括使用数据库里的查找表。

- 更改一个数据分块常常比较困难，因为它需要重新分配已存在的数据。比如说你开始用 20 取模的散列函数，后来随着数据分块增长，打算改用 200 取模来做，这样一来，就必须把所有数据重新进行散列，随之更新大量的数据，把数据在各个数据分块之间移动。

鉴于这些限制因素，我们更倾向于在新的应用中使用动态分配策略。如果是在对已有的应用做数据分块，那还是用固定分配比较方便，因为它操作起来简单。

我们有时也会在新项目中使用固定分配。一个实例是 BoardReader (<http://www.boardreader.com>) 论坛搜索引擎，这个论坛索引了大量的数据。我们本打算通过网站 ID 来对数据做散列的，这么说的话，所有网站的论坛都会被放置到一个数据分块上，在查找一个网上的一些论坛时候会比较快速——比如说你想找出某网站上几个排名前列的热门论坛。但是，一些网站里有数以千计的论坛，其中包含了好几百万条消息。这样的数据分块就大得难以管理了，因此，我们使用了论坛 ID 来进行散列。

426 动态分配

与固定分配相对应的是动态分配，动态分配是每个数据分块单独处理。举一个例子说，就是有着用户 ID 和分块 ID 两列的表：

```
CREATE TABLE user_to_shard (  
    user_id INT NOT NULL,  
    shard_id INT NOT NULL,  
    PRIMARY KEY (user_id)  
);
```

这个表本身就是个分配函数。给予一个分配键的值（这里就是用户 ID），你就可以找到对应的分块 ID。如果找不到，则可以匹配适合的数据分块，然后把分块 ID 加入到表里。稍后也可以修改它——这就是动态分配的含义。

动态分配在分配函数里会有额外的开销，因为它需要调用外部资源，比如目录服务器（存放映射表的数据存储节点）。这样的架构需要为了效率而多一些分层结构，比如采用分布式缓存系统把目录服务的数据保存在内存里，因为这些数据平时改动都不大。

动态分配的最大好处是对数据的存储可以做精细的控制，能够把数据在分块间均匀地分布，当做一些事先未预计到的变动时更具有弹性。

动态映射可以帮你在简单的键—分块映射之上构建多层次的分块策略。例如，你可以构建一个对偶映射，把每个数据分块关联到一个组（比如读书俱乐部里的一个用户群），然后这些组尽量都放在一个数据分块里。这样应用就可以受益于这些数据分块的机密性，避免过多的跨分块查询。

如果使用动态分配，你会发现数据分块间的不均衡。当你的服务器性能不一样时，这倒是很有用。或者其中几个分块你有特别用途的时候（像做数据归档之类的），如果你有能力让各个数据分块随时保持平衡的话，可以维持分块与节点之间一对一的映射，避免了容量的浪费。一些人还是喜欢一个节点一个数据分块这样的简单配置（但是，要记住小的数据分块好处多多）。

动态分配和灵活运用数据分块的紧密性可以帮你减轻随着应用规模扩大而带来的跨分块查询的问题。设想有一个跨分块查询涉及了 4 个节点上的数据存储，在固定分配情况下，无论怎么查询都要访问所有的分块，但使用动态分配策略的话，只要在其中的 3 个节点上做一次相同的查询就够了。这样的比较看起来差别不大，但是假如你的数据分块达到 400 个时，情况将会变得怎么样：固定分配要查询 400 个分块，而动态分配可能只需要查

询 3 个分块就行了。

动态分配会让你的分块策略想要多复杂就有多复杂，固定分配就没那么多选择了。

427

混合动态和固定分配。你也可以采用固定分配和动态分配混用的方法，这种方法有时很有用，而且是必须的。动态分配在目录映射的规模不大时效果不错，但如果有很多分块单元时，效果就不行了。

举例说明，有个系统是用来保存两个网站之间的链接的。这样的网站存储着几百亿条记录，它所用的分割键结合了源地址和目的地址的 URL（这两个 URL 里的任何一个都有好几百万条链接，因为两个单独的 URL 都不具备做分割键的条件）。然而，把所有源地址和目标地址 URL 都放在映射表里不太可行，因为这样的话记录实在太多，每个 URL 都要占用不少存储空间。

我们的解决方案是把 URL 通过散列放入固定数量的“桶”，然后把“桶”动态地映射到数据分块上。如果你准备了足够多的“桶”——比如说 100 万个——你就能够把大部分数据分配到每个数据分块上了。这样做的结果是你获得了动态分配的好处，同时避免了巨大的映射表。

显式分配

第 3 种分配策略是让应用在创建每一行数据时显式地指定一个所需的数据分块。这种做法在已有的数据上很难做到，因此，在对现有应用做分块改造时很少被采用。然而，它有时也能发挥其优点。

这个主意是把数据分块的号码编码到 ID 中，类似的技术我们已经在主—主复制系统中用来避免重复的键值（更多内容请查看“在主—主复制系统里往两个主服务器里写入数据”，第 398 页）。

举例说，假设你在应用要创建一个用户 3，并把他的数据放在 11 号数据分块上。可以使用一个 BIGINT 类型的字段来存放这两个编号，其中的 8 位用来保存这个数据分块的号码。这样，最终的 ID 就是 $(11 \ll 56) + 3$ ，即 792633534417207299。应用可以很方便地从这个 ID 里分解到用户 ID 和分块 ID，分解的过程像下面一样：

```
mysql> SELECT (792633534417207299 >> 56) AS shard_id,  
-> 792633534417207299 & ~(11 << 56) AS user_id;  
+-----+-----+  
| shard_id | user_id |  
+-----+-----+  
|         11 |         3 |  
+-----+-----+
```

现在假设你要给用户 3 添加一条评论记录，并放在同一个数据分块上。应用会先在这个评论 5 和用户 3 之间做一个关联，然后用同样的方法把评论的 ID 5 和数据分块号码 11 记录在同一个 ID 里。

这种做法的好处是每个对象的 ID 里会一直带有分割键，而其他两个方法里都需要做一次联接或查找来确定分割键。如果你想在数据库里找到某一段评论，你无须知道它是属于哪个用户的；对象 ID 就直接告诉你哪里可以找到该评论。如果这个对象是用用户 ID 动态地分块了，那就不得不先找到该评论的所有者，然后通过目录服务器找到要查找的数据分块。

428

还有个共同存放分割键的解决方案是把该键放在单独的一个字段里。举例来说，你不会单独去引用评论 5，但是评论 5 是属于用户 3 的。这个做法让一些人看来会很开心，因为它不违背第一范式。但是，额外的字段会引起更多的开销、编码任务、以及不便之处（反过来就是我们把两个值放入同一个字段时的好处）。

显式分配的缺点是分块数据去向都是固定的，难以做负载平衡。不过，有个很好的办法，就是让它和动态分配混合着用，原先都是把数据哈希到固定数目的“桶”里，然后再把“桶”映射到节点上，现在代之以把“桶”

作为每个对象的一部分直接编码在里面。这样应用就能控制数据的存放位置了，也就能够在一个数据分块上。

BoardReader 使用的是一个变种的技术：它把分割键编码在 Sphinx 文档 ID 里。这样每个搜索结构的相关数据都能很容易地被找到。更多关于 Sphinx 的内容可以阅读附录 C。

我们在此描述了混合的方式，只是因为它们在某些案例里很管用，但通常情况下我们并不推荐这种做法，倾向于尽可能地使用动态分配的方法，尽量少用显式分配的方法。

数据分块的重平衡

只要是必需的，你就要在各个不同的数据分块间移动数据来重新平衡负载。举例来说，许多读者可能听说过那些大型图片共享网站或热门社交网站的开发人员提到他们使用自己的工具在各个分块间移动用户数据。

在分块间移动数据的好处很明显，具体来说，当你升级硬件时，它可以帮你把用户数据从旧的分块移到新的分块上，而用不着停掉全部数据分块或把它设为只读。

然而，我们也要尽量避免去做重平衡，因为这毕竟会影响用户的使用。同时，要在应用里增加一个在分块间移动数据的功能并非易事，因为其他新功能做起来时都要包含一个重平衡的脚本了。如果你的数据分块一直保持得足够小，那你就用不着操这个心了。如果你确实经常要做重平衡，那么移动全部数据分块比移动其中的一部分容易得多（而且更高效——用每行数据开销这个术语来衡量）。

429 有一种较好的策略是把新数据随机分配到一个数据分块上。当一个数据分块满了的时候，就给它设置一个标志，告诉应用别再往这里放新的数据了。将来在这个数据分块容量扩大后，又可以把这个标志移除。

假设你安装了一个新的 MySQL 节点，上面有 100 个数据分块。开始时，你把他们的标志都设为 1，这样应用就知道它们正在准备接纳新数据。一旦他们中的每一个达到了足够的数据量（比如说有 10 000 个用户），你就把他们的标志都设为 0。之后，当因为节点负载不足而拒绝新账户注册时，又可以打开其中的一些数据分块，来接纳新用户的数据。

如果因为应用升级，或者在应用里添加了新的功能，甚至是由于你算错了负载值，结果导致每个数据分块的查询负载比预期高很多，那就只能把该节点上的部分数据分块移到新的节点上以减轻压力了。这个做法的缺点是当你移动那几个数据分块时，整个数据分块会处于只读模式或者离线状态。这就取决于你和用户是否能接受这个做法了。

生成全局唯一 ID

当你把现有系统转换为数据分块存储结构时，就经常会在不同的机器上生成全局唯一 ID。对于单一数据存储模式，可以使用 AUTO_INCREMENT 字段来取得唯一 ID。默认地，AUTO_INCREMENT 就是被设计用来在单一服务器上方便地生成唯一 ID 的。

对于在不同机器上生成全局唯一 ID 的问题，以下是几个解决途径：

使用 `auto_increment_increment` 和 `auto_increment_offset`

假定这里有两个 MySQL 数据库，它们也使用了 AUTO_INCREMENT 字段来取得唯一 ID，为了保证两个数据库同时具备唯一性，我们可以用一个初始值来做 AUTO_INCREMENT 的偏移量。具体来说，在这两个 MySQL

中，它们的自增长幅度设为 2，然后将其中一台的起点设为 1，另外一台设为 2（两个中不能同时为 1，否则就重复了）。这样的话，我们就可以保证一台数据库里的 ID 总是奇数，而另一台总是偶数，绝不会重复。这样的设置可以配置到服务器上的每一个表里。

这种方法很简单，而且也不依赖一个中心节点，因此是生成唯一性 ID 的上佳选择。现有的服务器也很容易用这个方法配置，特别是当你增加服务器或灾难恢复之后。

在全局节点上创建一个表

在一个全局数据库节点上创建一个带有 `auto_increment` 字段的表，应用就从这个表里来取得唯一性 ID。

使用 memcached

在 memcached API 中有个 `incr()` 函数，它能产生一个唯一性 ID 供使用。

批量分配编号

应用从全局节点上一次性地取得一批编号供自己使用，用完后，再申请一批。

使用复合值

你可以使用一个复合的值来做唯一性 ID，比如一个数据分块 ID 和自增长编号，如何生成这样的值可以参见前面单元里讲过的内容。

使用双字段 AUTO_INCREMENT 键

这个只能在 MysISAM 表里使用

```
mysql> CREATE TABLE inc_test(
->   a INT NOT NULL,
->   b INT NOT NULL AUTO_INCREMENT,
->   PRIMARY KEY(a, b)
-> ) ENGINE=MyISAM;
mysql> INSERT INTO inc_test(a) VALUES(1), (1), (2), (2);
mysql> SELECT * FROM inc_test;
+----+-----+
| a | b |
+----+-----+
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |
+----+-----+
```

使用 GUID

你可以使用 `UUID()` 函数来生成全局唯一的 ID。注意，尽管这个 GUID 不能被正确的复制，但是在应用中可用它来选择数据到内存中，并作为字面行的声明。GUID 的值很大且不连续，因为它不适合做 InnoDB 表里的主键，请参考“在 InnoDB 中按主键插入数据”，第 117 页。

MySQL 的开发者已经给 MySQL 创建了一个新的 `UUID_SHORT` 函数，它能返回一个较短的连续的值，也更适于用作主键。在 MySQL 的未来版本里可能会加入这个函数，在此书正在写作的时候，这样的 MySQL 版本还没发布。

如果你使用的是一个全局分配器来产生唯一性 ID，要当心它会成为应用的瓶颈。

虽然 memcached 运行很快（每秒钟几万个值），但它不是持续不断的。每次重启了 memcached 服务，都需要在缓存里初始化那个产生值。这就要求你初始化时，每次都要找出目前在各个数据分块使用的最大值，这个过程相当缓慢而且难以自动执行。

如果你使用 MySQL 里的一个表，有种方法，即创建一个 MyISAM 表，里面只有一个字段：AUTO_INCREMENT 字段。这样外部应用也可以通过事务来读取以提高响应速度。

431 可以让表随着数据的添加而自动增长，也可以使用 REPLACE 让表里的数据值保持一条。

```
CREATE TABLE single_row (
  col1 int NOT NULL AUTO_INCREMENT,
  col2 int NOT NULL,
  PRIMARY KEY(col1),
  UNIQUE KEY(col2)
) ENGINE=MyISAM;
```

你可以像下面这样使用这个表来产生唯一性 ID：

```
mysql> REPLACE INTO single_row(col2) VALUES(1);
```

在这个语句执行后，可以使用 MySQL 的 API: mysql_insert_id() 来获得这个新生成的值。这个方法在不同语言中的实现都不太一样，以下是一个 PERL 的例子：

```
my $sth = $dbh->prepare('REPLACE INTO single_row(col2) VALUES(1)');
while ( my $item = @work_to_do ) {
  $sth->execute( );
  my $id = $dbh->{mysql_insert_id};
  # Do the work...
}
```

用不着再使用另外的查询语句（比如 SELECT LAST_INSERT_ID()）来获取这个值。额外的查询又会在服务器之间多一次往返，这显得很没效率。

数据分块的工具

在设计一个数据分块的应用时，首要的事情是编写能查询多个数据源的代码。

如果没有任何抽象，让应用直接去访问多个数据源，那绝对是个糟糕的设计，因为这会增加许多复杂的代码。一个好办法是，把这些代码都放在一个抽象层里，这个抽象层要完成以下这些任务：

- 联接到正确的数据分块上，并查询数据。
- 分布式一致性检验。
- 跨分块查询后的数据聚合。
- 跨分块连接。
- 锁和事务管理。
- 创建新数据分块（至少是在运行时找到新的数据分块），然后重平衡所有分块（如果你有时间实现它）。

你可能不需要从零开始构建数据分块结构。这里有一些工具和系统，他们或者能提供一些必须的功能，或者能用来实现数据分块的架构。从最基本的层次上讲，可以使用一个工具，比如 MySQL Proxy 来抽象多个数据源的

复杂性。随着 MySQL Proxy 的逐渐演变，它在许多数据分块应用中的位置将越来越关键。

在 Hibernate Shards (<http://shards.hibernate.org>) 里已经有了一个带有分块技术的数据库抽象层，Hibernate Shards 是开源的 Hibernate ORM 库的一个扩展，是使用 Java 编写的。Google 开发了 Hibernate Shards 中 20% 的功能，然后将代码贡献给了社区。它在 Hibernate Core 接口上实现了“分块感知”，这样一来，应用在使用分块技术的时候就无须重新设计了。事实上，应用无须知道自己是不是在使用数据分块。Hibernate Shards 能够透明地从各个数据分块上读取或存入数据，但它本身也没提供一些别的功能框架，像分块重平衡、聚合查询结果等。它使用固定分配策略将数据分配到数据分块。

另一个数据分块系统是 HiveDB (<http://www.hivedb.org>)，一个 MySQL 数据分块的开源框架，它致力于用简洁明了的方式来实现数据分块的核心思想。它有着其他系统所不具备的功能，比如创建数据分块、在分块间移动数据（就是重平衡）。HiveDB 使用的是动态分配策略，它把这样的分块方式叫“水平分割”。

Sphinx 是全文搜索引擎，不是一个数据分块存放系统，但它在一些跨分块的数据查询时非常有用。它能并行地查询多个远程系统，并将查询结果聚合起来返回——这个是一个分块存储系统难以做到的功能（更多关于 Sphinx 的内容请查看附录 C）。

9.2.5 回缩

Scaling Back

对付日益增长的数据量和系统工作负载的最简单的办法是那些不需要的数据归档和删除。根据工作负载，你会发现归档和删除不需要的数据会有显著的成效。这种做法并不是为了替代水平扩展，但是，它可作为争取时间的短期策略，也可能成为应付数据增长的长期策略之一。

当设计归档和删除策略时，这里有一些需要考虑的事情：

对应用的影响

一个设计良好的归档策略，它能够在不影响事务处理的情况下，把数据从重负载的 OLTP 服务上移走。这里的关键是能够有效地找到要移除的行，并一小块一小块地移除。你要确定每次做归档的行的数目，使这个事务在锁竞争和事务负载量之间做到很好的平衡。

哪些需要归档

当你知道一些数据不会再被使用后，就马上可以把它删除或归档，但是你可以设计应用，使它能自动归档那些很少被访问的数据。可以把归档的数据置于核心表的附近，通过视图来访问它们，或者干脆把它们移到别的服务器上。

深度优先还是广度优先

数据间的关联使归档和删除变得复杂。一个设计良好的归档任务能够使数据保持逻辑上的一致性，至少是在应用要访问的数据范围内保持一致性，不会往庞大的事务里再加入多个表。

当表间关系众多时，决定从哪个表开始归档往往是个挑战性任务，如果没有选对开始的表，你在归档时就不必面对“孤儿”或“寡妇”数据行，因此，在归档时，决定是否违背外键将是件麻烦事（可以通过 `FOREIGN_KEY_CHECKS=0` 来关闭 InnoDB 的外键约束），或者暂时把那些“摇摆指针”式的记录放一边去。采用哪种方式取决于应用怎样看待数据。如果应用里从上到下显示一组相关的表里的数据，那归档的次序

也要照它那样。举例来说，应用总是先检查订单然后再检查发票，那你就该先将订单数据归档，就不会看到成“孤儿”的发票数据了，接下来就是将这些发票数据归档。

避免数据丢失

如果你正将一台服务器上的数据归档到另一台上，可能并没用到分布式事务处理，而是把数据归档到 MyISAM 里，或者把另外的非事务数据存储引擎里。因此，为了避免数据丢失，在删除数据源之前要先把数据存放到目标服务器上。把这些归档数据同时写进一个文件倒也是个好主意。你应该把归档任务设计成这个样子：无论何时把它停止或重启，都不会引起数据一致性、索引冲突之类的错误。

解归档

你会经常使用解归档的策略来减少归档数据。即在你难以确定哪些数据需要归档时通过设置一个可回退的选项来做归档。如果你设置了一些检查点让系统来检查是否有数据需要归档——这是个相当容易实现的策略。比如说你将那些不活跃的用户归档，这个检查点应该设在登录验证过程里。如果有一个登录因为不存在这个用户而失败，你就去检查归档数据，看看是否有这个用户存在，如果存在的话，就从归档数据里读取该用户的信息，完成该登录。

Maatkit 中有这样一个工具，它能高效地帮你归档和（或）删除 MySQL 里的表，但是它不提供解归档的功能。

4.3.4 保证归档数据的隔离

即使实际上不是把老数据移到其他别的服务器上，但许多应用还是得益于将活跃数据与不活跃数据相互隔离。隔离之后，缓存的利用就更有效率，而且对于这两类数据可以采用不同的硬件和软件架构来处理。这里例举了几个做法：

将表分成几部分

把表分成几部分是个聪明的办法，尤其是当表无法全部放入内存的时候。举例来说，你可以把用户表分为活跃用户表和不活跃用户表。你可能在想这纯粹是多此一举，数据库自然会把“热门”数据放入缓存里，其实，这个是因不同的数据引擎而异的。如果使用的是 innodb，它的缓存每次只加载一页，你在一页里可以放入 100 个用户，另一方面，假设在你的用户表只有 10% 是活跃的，这样的话，缓存里的每一页有 90% 的空间被浪费了。把这个表分成上述两部分的话，就能显著提高内存的使用率了。

Falcon 存储引擎有个行级别的缓存机制，这使得缓存的使用更高效。但是，这并不意味着 Falcon 的表不能通过活动/非活动的分隔而获得性能的提升。Falcon 的缓存每次对一页做索引，当这个索引里混合了活动和非活动的数据时，还是会降低的索引的效率。因此，在 Falcon 做活动/非活动的分隔时还是很有用的。

MySQL 分区

MySQL 5.1 提供了原生的分区表技术，它能把最近的数据都放在内存里。更多关于分区技术的内容请查看第 253 页的“表的合并和分区”。

以时间为基础的数据分区

如果你的应用要频繁地取得新数据，这样的话，新数据将比旧数据更加活跃。举例来说，BLOG 服务的流量来自于最近 7 天里发布的文章和评论，多数的数据更新也来自这个时间范围，那么他们就把这些数据都放进内存里，并在磁盘上存放一份可恢复的数据副本，供数据出错时调用。其他那些数据就放在别的地方

了。

我们也见过另外一种设计，他们在两个节点上存储着每个用户的信息。新的数据就被放入一个活跃的节点，这个节点配置了巨大的内存和快速磁盘，而且数据也被优化过以支持快速访问。另外一个节点上存储的是旧数据，有着容量巨大且慢速的磁盘。这里假设应用还可能要访问这些旧数据的——这个假设对大多数应用来说都很适用，它们响应的请求里，有 90% 以上是访问那些最新的 10% 的数据。

可以使用动态分配的策略来实现这种分块方式。举例来说，应用了分块目录技术的表的定义可以像下面这样：

```
CREATE TABLE users (  
  user_id          int unsigned not null,  
  shard_new        int unsigned not null,  
  shard_archive    int unsigned not null,  
  archive_timestamp timestamp,  
  PRIMARY KEY (user_id)  
);
```

435

在这个表里，通过一个归档脚本就可以把旧数据从活动节点移到归档节点上，当一份用户数据被移到归档节点上时，就更新它的 `archive_timestamp` 字段，通过 `shard_new` 和 `shard_archive` 字段你可以获知那个编号的分块存放了该份数据。

9.2.6 通过集群来扩展

Scaling by Clustering

集群是另一种在服务器间做分布式负载的扩展技术。“集群”这个术语在计算机领域有着多种含义，通常而言，集群由一个局域网内的几台主机组成，其表现得像一台服务器。有一种集群的变种是联盟——访问远程服务器就像在本地一样，它要创建一个虚拟服务器就像许多服务器的代理。

集群

MySQL 的 NDB Cluster 存储引擎是一个分布式、嵌入内存的、非共享的存储引擎，它还有同步复制和节点间数据自动分区技术。与其他 MySQL 存储相比，它有着完全不同的性能特点。它在特定硬件上有着上佳表现。虽然对应用而言，它存储数据有着很高性能的表现，但对于许多 Web 应用来说，它还不能算是高性能的解决方案。

NDB Cluster 适用那样的应用：数据量相对较少，并执行简单的查询。具体而言，它适用于网站的 session 存储，文件元数据的存储等。当它执行复杂的查询，包括较多的联合时，性能就变得很差，从技术角度来讲就是当任何一条查询不是单表内索引查询而是内部各点间通讯时，性能就下降了。

NDB Cluster 是个事务系统，但是不支持 MVCC，它采用了读锁的方式，而且也不做任何死锁检验。运行当中遇到死锁时，NDB 就以超时返回的形式来处理。以读锁定和超时返回机制相结合的方式，决定了它对于多用户交互的应用和 Web 应用而言不是个好的解决方案。

你可以实现一个变种的集群方案，它基于 MySQL 之上，或者 MySQL 的前端，或者在 MySQL 底层。有个例子可以参考 Continuent (<http://www.continuent.com>) (注 5)，它提供了同步复制负载均衡和通过一个中间件层实现 MySQL 的故障恢复。

注 5：或者是该软件作者的另外一个开源项目——Sequoia，访问 <http://sequoia.continuent.org>。

联合是另外一个含有多种意义的术语。在数据库世界里，它一般是指从一个服务器去查询另外一台服务器的数据。微软的 SQL Server 分布式视图就是这样的东西。

MySQL 通过 Federated 存储引擎对联合提供了有限的支持，该引擎类似于 NDB 集群。它擅长于简单的查找，尽管它在别的服务器上做 INSERT 操作时性能也可以被接受。它目前采用的架构决定它对 DELETE 和 UPDATE 的操作更加低效——在最糟糕的情形下，效率更加低。

Federated 引擎在使用联合和做大数据规模的 SELECT 时表现很差。举例来说，当 GROUP BY 从一个表里取得数据时，或者是当使用 mysql_store_result（注 6）从远程服务器上取数据到本地内存里时。这个缺点在应用的数据规模逐渐增大时会引起大麻烦。Federated 的表也使得复制变得更复杂，因为一个简单的 update 需要在多个服务器里执行。

9.3 负载均衡

Load Balancing

负载均衡的基本思想很简单，就是在一个服务器集中间尽可能地平均地分配工作量。实现这一目的的通常做法是在服务器之前设置一个负载均衡器（往往是一个专门的硬件设备）。这个负载均衡设备会把接入的连接路由到最空闲的可用服务器上。图 9-5 显示的是典型的用于大型网站负载均衡配置，其中一个用于 HTTP 流量的，另一个用于 MySQL 访问。

负载均衡有 5 个基本目标：

可伸缩性

如果系统设计得正确，那可以通过在节点上增加更多的服务器来提高其处理能力。当增加更多服务器时，必须平衡各服务器的实际负载。

高效性

通过控制请求的路由走向使负载均衡帮你更有效率地使用服务器资源。这点特别重要，如果服务器的处理能力各不相同，那你就要把更多的工作量路由到那些强大的服务器上。

可用性

一个灵活的负载均衡方案里选用的服务器都是能保持时刻都可用的服务器。

客户端无须知道是否有负载均衡的存在，也不必关心在负载均衡器后面有多少台服务器，它们的机器名分别是什么。负载均衡器在客户端看来就是一台虚拟服务器。

一致性

如果你的应用是有状态的（如数据库事务、站点 session 等），那负载均衡设备就应该能在重定向客户端请

注 6：更多关于 mysql_store_result 的内容请查看第 161 页的“MySQL 客户端/服务器 协议”。

求时不丢失有关的状态信息。这样,就能使应用不必总是需要了解自己连接的是哪台服务器。

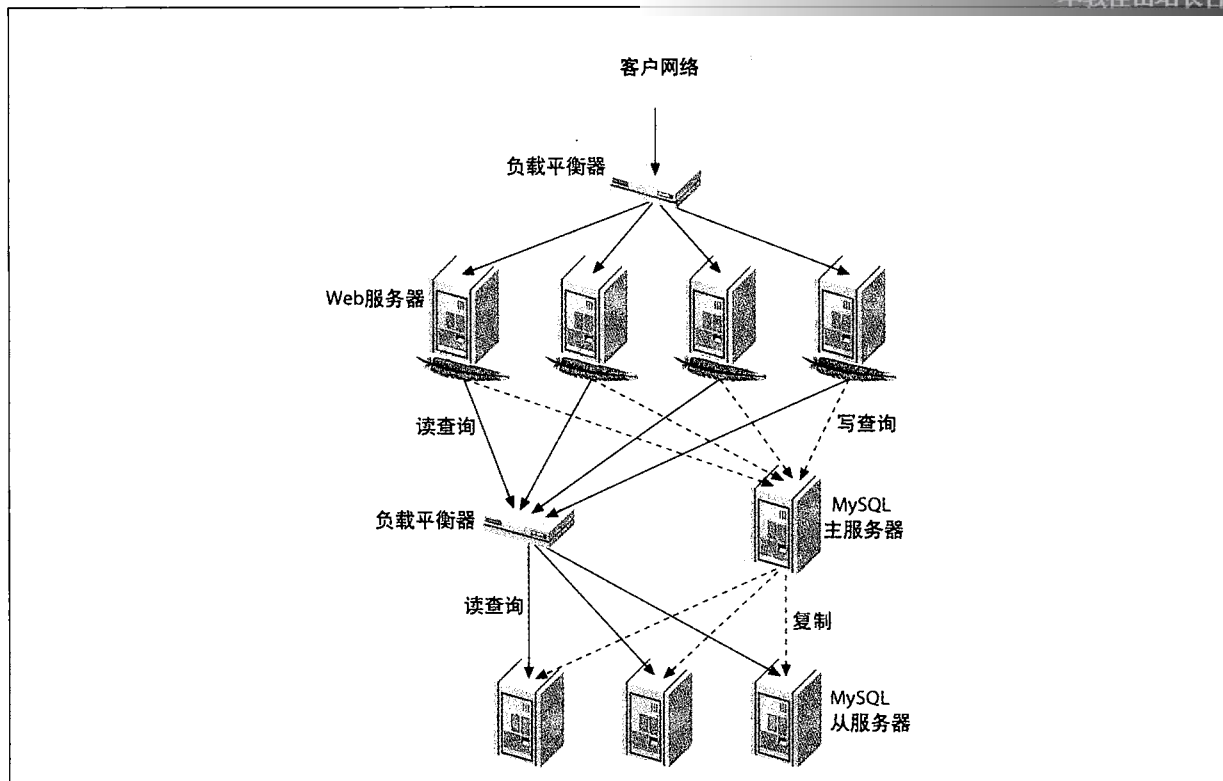


图 9-5: 典型的负载均衡架构, 用于读量较大的网站

在 MySQL 世界里, 负载均衡架构总跟数据分块和复制紧紧地伴随在一起。你可以把负载均衡和高可用性融合在一起, 部署到应用中的任何一个合适的层上。举例来说, 可以在一个 MySQL 集群的多个节点间作负载均衡, 也可以在各个数据中心间作负载均衡, 在每个数据中心里你可以使用一个数据分块的架构, 其中每个节点实际上就是一个附带了许多从存储的主—主复制对, 这些需要再次进行负载均衡。对于高可用性策略而言也是如此, 你可以在这个架构的多个层上实现故障恢复机制。

428

负载均衡有许多微妙之处。举例来说, 有一个挑战是管理读/写策略。有些负载均衡技术本身就实现了这个策略, 但是, 另外一些服务器就需要应用自己来记住哪些节点是可读的, 哪些节点是可写的。

当你构想如何实现负载均衡时, 以上这些因素都需要考虑进去。目前, 有很多类负载均衡解决方案供你参考, 它们的范围从以点为基本的实现 (如 Wackamole (<http://www.backhand.org/wackamole/>)) 到 Domain Name System (DNS)、LVS (Linux Virtual Server; <http://www.linuxvirtualserver.org>), 硬件负载均衡、MySQL Proxy, 以及在应用里管理负载均衡。

9.3.1 直接连接

Connecting Directly

一些人一听到负载均衡就想到那是个夹在应用与 MySQL 数据库之间的东西。然而, 这并不是负载均衡唯一的表现形式。你也可以在保持应用与 MySQL 服务器直接相连的情况下继续使用负载均衡。中央化的负载均衡—

般只有当应用访问一组对等的可替换的服务器时才有良好的效果。如果当一个应用要判断数据是否安全时，它就需要跟服务器直接相连了。

包括一些可能出现的特例逻辑在内，应用中为负载平衡做决策的过程其实是非常高效的。举例来说，假如有两个一样的从服务器，你可以选择其中一个来应付针对某些数据分块的全部查询要求，而另外一个就应付针对其他数据分块的所有查询要求。这样就可以充分使用从服务器的内存，因为它们都只需把数据中的一部分放入内存就可以了。如果其中一台出故障了，另外一台上仍然保留了全部数据分块上的数据以供查询。

以下部分是讨论应用的“直连”技术的几个常用途径，当你评估每个选项时，应该会考虑到其中的一些注意点。

在复制上分离读与写

MySQL 复制使表有多份副本，让你选择是在主服务器还是从服务器上运行一个查询。因为复制过程是异步的，所以，这里的主要困难是怎么处理从服务器里面的旧数据。你可以把从服务器当作是只读的，而主复制能运行读、写查询。

你经常不得不对应用作一些更新，因此你得知道那些令人担心的事情（注 7）。现在，应用可以使用主服务器做读操作，而写操作可以在主服务器和其他从服务器之间分配。在从服务器里访问到的是不太紧要的旧数据，而最新数据就放在主服务器上。

439 如果你使用主—主来实现主动—被动服务器组，同样也要考虑到这些。但是，在这样的配置中，只有主动服务器来接受写操作，而读操作都让被动服务器来承担——如果它上面有那些需要的旧数据的话。

在这里的最大问题是如何避免在读旧数据产生被遗落的“史前古物”，所谓被遗落的“史前古物”是指当一个用户在对旧数据操作时，比如对博客上的某篇文章增加了一条评论后，重新加载了页面，但是看不到自己刚才提交成功的评论，究其原因应用是从一个从服务器上读出博客文章的，而新评论是写在主服务器上的。

下面罗列了一些最常用的分离读/写操作的方法：

以查询为基础的分离

最简单的分离方法是把所有的写操作和一些不接受旧数据的读操作全部放在主动服务器或主服务器上，而其他的读操作都放在从服务器或被动服务器上。这个策略很容易实现，但是在实际运行时会发现从服务器的访问没有预计的那么频繁，因为几乎很少的读操作能够只适用旧数据。

旧数据分离

这个方法是对以查询为基础分离方法的一个小小提升。这些增加的额外工作是要应用去检查从服务器的迟滞程度，然后决定这些数据对读操作是不是够“陈旧”。许多报表类的应用都可以使用这个策略：在晚上数据负荷没有的时候，就把数据复制到从服务器里，应用不会在乎这些数据是不是跟主服务器同步更新的。

以 session 为基础的分离

一个稍微更进一步的方法来决定一个读操作是否在从服务器上执行的方法是在系统标识上用户是否更改了数据。用户用不着一定要看到其他用户的最新数据，但是他必须看到自己更新过的数据。你可以这样来实现：如果用户对数据有过更新，就在 session 层设置一个标志，这样当用户执行读查询时，系统就会到主服务器上读取数据。

注 7：如果你可以使用 MySQL Proxy 来拆分查询，就不需要更改应用了。

可以把这个功能与复制间隔时间监控相结合。如果用户在 10 秒钟前更新了一些数据，并且在 5 秒内都更新过数据了，那么在从服务器读数据就是安全的。一个很好的主意是在从服务器上设计一个用来存储这些 session 相关的数据，否则用户会奇怪地发现有些从服务器的更新数据的速度比其他几台要慢。

以版本为基础的分离

这个方法跟以会话为基础的分离很相似：你通过版本号和（或者）时间戳来跟踪对象，根据从服务器上读取对象的版本号或时间戳来判断这些数据是否足够“新鲜”以供使用。如果从服务器上的数据太旧了，你就可以把它们从主服务器上读过来更新。即使当对象本身没有发生改变，你也可以增加顶层项目的版本号，这样就简化了旧数据的检验（你只要查看一个地方——顶层项目就行了）。举例来说，当用户在博客里发布了新文章，你就可以更新他的版本号，这样数据就会从主服务器上读取了。

在从服务器上读取对象的版本号会增加它的运行开销，只能通过缓存减轻它的影响了。在下一章里，我们会更深入地讨论缓存和对象版本。

全局版本/会话分离

这是以版本为基础和以会话为基础的变种方法。当应用执行一个写操作时，它会在事务提交后运行 `SHOW MASTER STATUS`。它把主服务器的日志坐标作为更新后的对象和（或者）会话版本号放在缓存里。然后，当应用连接到从服务器时，它就运行 `SHOW SLAVE STATUS`，用从服务器上的参数与先前保存的版本相比较。如果从服务器上的版本至少有一个编号高于主服务器提交事务后得到的版本号的话，那说明从服务器上的数据可以被安全地读取。

许多读/写分离解决方案都需要监控从服务器的滞后时间，并以此来决定从哪个服务器上读取数据。如果你也是这样做的，那请注意 `SHOW SLAVE STATUS` 执行后返回的结果里，`Seconds_behind_master` 字段不能用来衡量从服务器的滞后时间。更多细节可查看 379 页的“度量从服务器的滞后时间”。

如果纯可测量性是你的目标，你也不关心要用到多少硬件设备，那事情就变得更简单了，可以不用复制，或者只用它来提高可用性而不是做负载平衡。这样一来就避免了在主服务器与从服务器之间分离读操作的复杂性。有些人认为这么做很合理，也有些人认为这是浪费硬件。不同的目标导致了这个分歧：你是只需要可测量性，还是既要测量性也要高效性？如果你同时需要高效性，那从服务器除了保存数据副本外还要承担其他一些任务，你就不得不处理这些额外增加的复杂性了。

改变应用的配置

还有一个分担负载的方法就是配置你的应用。举例来说，在产生一批大数据量的报表时，可以多配置几台机器来分担这个负载，每台机器都连接到不同的 MySQL 从服务器，分别为对应的第 N 个客户或网站生成报表。

这样的系统通常很容易实现，但是它需要修改一些代码——包括那些配置文件会被改得诘屈聱牙。那些所有硬编码的东西因其固有的限制，需要你逐个服务器去修改，或者在一个中心服务器上修改，然后通过文件或代码控制软件的更新命令“发布”到其他服务器上。如果你的配置都是放在数据库和（或）缓存里的，那就可以避免刚才讲到的麻烦事了。

4.1 更改 DNS 名字

这个负载均衡的方法比较粗陋，但对于一些简单的应用来说，根据不同的目的创建 DNS 名称却很适用。你要创建一个周期性的任务来监控那些 MySQL 服务器，为每个服务器指定一个合适的名字。最简单的实现方法就是给只读的服务器一个 DNS 名字，而给负责写操作的服务器另外一个 DNS 名字。如果从服务器是与该主服务器同步的，那就把只读服务器的名字指给从服务器用，如果它们跟不上同步，你就把这个 DNS 名字给主服务器。

这种 DNS 技术很容易实现，但是错点也不少。最大的问题是你无法把 DNS 完全置于你的控制之下：

- DNS 的更改不是立即生效的，它在网络里传播到每个角落需要比较长的时间。
- DNS 数据在各处缓存起来，它的失效时间是建议性质的，而不是强制性的。
- DNS 更改后可能需要应用或服务器重启后才能生效。
- 多个 IP 地址共用一个 DNS 名称是个不好的主意，具体依赖于负载均衡请求的轮询行为，这个轮询行为不总是预计的。
- DNS 的更改不是个原子操作。
- DBA 可能无权直接访问 DNS 的设置。

除非应用非常简单，依赖于一个不可控的系统总是危险的。你可以通过另外的途径提高一点点对系统的控制权。也可以通过更改/etc/hosts 把你的控制权提升一点。当你对这个文件发布更新时，就知道这些新的设置开始生效了。这比干等着 DNS 缓存信息失效要好得多，但也不是很理想。

我们常常建议人们在构建应用时要实现对 DNS “零依赖”，即使在应用很简单时也适用，因为你不会知道应用将来会扩展到多大的规模。

移动 IP 地址

一些负载均衡的方案依赖于在服务器之间移动虚拟 IP 地址（注 8），这个方法很不错。听上去类似于改变 DNS 名称，其实不是一回事。服务器不会根据一个 DNS 名称去侦听网络流量，但是可以根据一个指定的 IP 地址去侦听网络流量。所以移动 IP 地址，可以保证 DNS 名称静止不变。通过地址转换协议（Address Resolution Protocol ARP），你能迅速而且原子性地使 IP 地址的改变通知到网络的各处。

4.2 有两个系统使用了这种技术：Wackamole 和 LVS。举例来说，它们让你把一个 IP 地址跟一个角色（例如只读操作）关联起来，它们根据实际需要来完成在各个机器之间移动 IP 地址。Wackamole 能够管理许多个 IP 地址，并保证有一台并且只有一台机器监听每一个地址。Wackamole 的服务是以端点为基础的，这可以用来消除一个点故障而带来的影响。

有一个方便的技术是把每个物理上的服务器都配置一个固定 IP。IP 地址就定义在服务器上，不再改变。你对每个逻辑上的“服务”使用虚拟 IP。可以为每个服务都指定一个虚拟 IP 地址，服务可以在各个服务器之间轻易地移动。这样也就可以方便地移动服务和应用实例，而不必重新配置应用了。这是个很棒的架构，你再也不必为了负载均衡和高可用性，把 IP 地址四处移动了。

注 8：虚拟 IP 地址不连接到任何一台指定的电脑或网络接口，它们只在电脑之间“漂浮”。

9.3.2 引入一个中间件

Introducing a Middleware

迄今为止，我们讨论的所有技术都是假定应用跟 MySQL 服务器是直接通信的。然而，许多负载均衡方案里都会引入一个中间件，它的作用就像个网络通信的代理，它在一边接受所有访问请求，又在另一边把这些请求指派到合适的服务器上去处理，然后又把响应结果发回到请求来源的电脑上。这个中间件有时是一个硬件设备，有时是一个软件（注 9）。图 9-6 描述的就是这个架构。这样的架构通常都会运作得很好，要是你把负载均衡设置成冗余，这样就不会因为负载均衡故障而导致整个系统瘫痪。

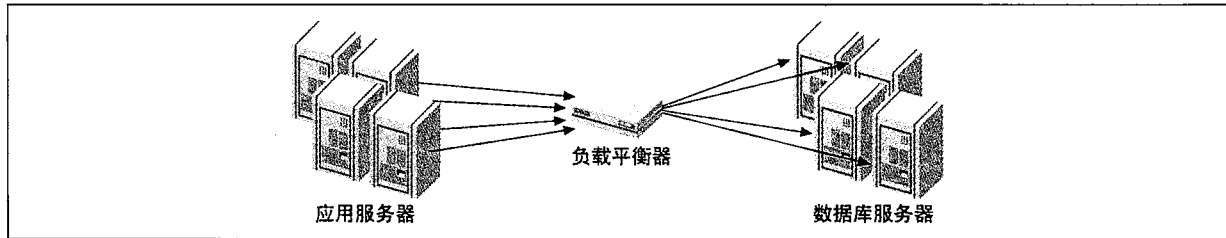


图 9-6：一个负载均衡器，其行为如同一个中间件

负载均衡器

在市场上充斥着各种各样的负载均衡设备，硬件的、软件的都有。但是，其中很少是为 MySQL 服务器专门设计的（注 10）。

Web 服务器更需要负载均衡，许多多用途的负载均衡设备都支持 HTTP，但其他基础架构却不是很全。



提示：一个例外是 MySQL Proxy，它能很好地为一些应用分离读和写操作。它增加了复杂性和一些开销。但它也带有极大的弹性，让你能通过脚本来控制读/写分离。MySQL Proxy 相对比较新，但是已经有很多在线的指南和实例来帮你配置它的负载均衡功能。因为它能深入查看由它转发的会话数据，所以，它做一些很复杂的查询路由。

MySQL 的连接都只是普通的 TCP/IP 连接，所以，可以在 MySQL 上使用多用途负载均衡器，但它们不是为 MySQL 专门设计的，所以，其中也带来了一些局限：

- 除非负载均衡器知道 MySQL 的真实负荷，否则，它不能为分布式请求做到负载均衡，不是所有查询请求都一样的，但多用途负载均衡器将它们一视同仁。
- 许多负载均衡器都知道如何检查 HTTP 请求，把会话“粘贴”到一台服务器上，并把会话状态保存在一台 Web 服务器上。MySQL 的连接也是有状态的，但是这类负载均衡器不知道怎么把所有连接请求从一个 HTTP 会话“粘贴”到一台 MySQL 服务器上。这就导致了效率的下降（如果一个会话里的所有请求都转发到同一台 MySQL 服务器上，那服务器的缓存才会显得有效率）。
- 连接池和持久性连接会阻碍负载均衡器分发连接请求的能力，举例来说，一个连接池打开了其配置数量的连接，负载均衡器把这些连接分配到了现有的 4 台 MySQL 服务器上，现在我们增加了两台 MySQL

注 9：你可以把 LVS 配置成每当应用要创建新的连接时就参与进来，LVS 毕竟不是个中间件。

注 10：在前面章节里，我们提到过一些软件的实现（Sequoia, Continuent），这里还有个语言独立的方案是 DBIx::DBCluster (<http://sqlrelay.sourceforge.net>)。

服务器。因为连接池不会再生成新的连接，那么这两台 MySQL 服务器就处于空闲状态，而那些连接还在那 4 台服务器中间分配着，结果就是一些服务器过载运行，而另外一些闲置着。针对这样的问题，可以通过在不同层面上设置连接池内连接的失效时间来缓解，但是，这个做法比较复杂，也难以实现。连接池解决方案只有当它们自己处理复杂平衡时才会发挥最好的效果。

- 许多多用途负载均衡器能给 HTTP 服务器做健康和负载检查。一个简单的负载均衡器就能校正服务器在 TCP 端口接受的最小连接数，一个更好一点的负载均衡器就能自动发起一个 HTTP 请求，然后通过检查其响应代码来确定这个 Web 服务器是否运行正常。MySQL 服务器在 3306 端口上不会接受任何 HTTP 请求，因此需要你自己来构建一个健康检查方法。可以在 MySQL 服务器上安装一个 HTTP 服务器软件，然后编写个脚本，让负载均衡器来检查 MySQL 服务器的状态，并返回一个适当的状态码。（注 11）其中，最需要检查的是操作系统的负载情况（一般是查看 RPOC/LOADAVG）、复制的状态和 MySQL 的连接数。

负载均衡的算法

有许多种算法来决定哪个服务器来接收下一个连接。每个厂商都用各自不同的算法，下面这个清单罗列了那些可用的方法：

随机

当请求到来时，负载均衡器从可用服务器池里随机选择一台服务器来处理。

轮询

负载均衡器按下面这个循环序列把请求转发给服务器：A、B、C、A、B、C 等。

最少连接优先

下一个连接会被转发到正处理着最少连接的服务器。

最快响应优先

把下一个连接分派给处理请求最快速的服务器。当服务器池里的服务器处理速度有快慢之分时，这个策略就相当有用。然而，当查询复杂度变化较大时，这样的算法就有点难以应付了。即使同样的查询在不同的环境运行也有不同的表现，例如当同样的查询正处于查询缓存上时，或者要查询的数据正好在服务器缓存里时。

散列化

负载均衡器把连接的源 IP 地址散列化，然后映射到服务器池里的某一台服务器上。每次有来自相同 IP 地址的连接，都会被分派到相同的服务器上。只有当池里的服务器数量发生变化时，这个绑定才会随之发生变化。

权重

负载均衡器能够同时使用以上几种算法，并通过权重来得到结果。比如说你有单 CPU 和双 CPU 的机器，双 CPU 机器的性能大概要比单 CPU 高两倍，这样你就可以告诉负载均衡器把多于两倍的请求分派给双 CPU 机器。

注 11：实际上，如果你的编码功夫能够写出一个程序去监听 80 端口，或者你把 xinetd 配置到程序中，你就用不着安装一个 Web 服务器。

哪个才是最好的负载均衡算法要取决于应用的工作量。举例来说，使用最少连接算法在有新服务器加入服务器池时，可能会使大量连接涌入到新服务器上——此时服务器的缓存还没来得及全部激活。本书第一版的作者曾经亲身体验过这样的情况。

你将需要通过实验为系统的工作量找到最好的性能状态，在日常的运行环境之外，也要考虑到一些特殊环境下的情况。即使在那些特殊的环境下——比如高负载、样式改变了、特定的几台服务器离线了，你的系统也应该能做点什么以防止发生大错误。

445

以上我们讨论的都是即时处理请求的算法，还没涉及将连接要求放入等候队列。有些时候那些使用了队列的算法会更有效一些。举例来说，有个算法在数据库服务器上维护着一个指定的并发性，就是说在同一时刻不会有超过 N 个的活动事务存在，如果有太多的活动事务生成，这个算法就会把一个新的请求放入队列中，然后让可用服务器列表里的第一个服务器来处理它。一些连接池就支持队列算法。

在服务器池里增加和减少服务器

在服务器池里增加一台新服务器，往往不是简单地接上一台服务器，然后告诉负载均衡器这个新服务器的到来。你可能想这样也没错，只要不被突然涌入的连接冲掉就行了。但是这不总是正确的，有时候需要慢慢地给服务器增加负载，有些服务器的缓存还是“冷”的，运转很慢，以至于一段时间它根本无法执行任何查询。

当一台服务器的缓存还是“冷”的时候，即使很简单的查询都会等待很长的时间才会结束。如果用户只是查看一个页面视图，而服务器花了 30 秒才返回所需要的数据，那么说明这台服务器还不可用，哪怕是一点点数据量。这样，你只有在通知负载均衡器加入这台新服务器之前，暂时把 SELECT 查询的映射到一台活跃的服务器上。可以在新服务器上读取和转发那台活动服务器的日志文件。

你应该在连接池里配置这些服务器，这样就能保留一些剩余的负载能力，在撤下服务器维护时使用，或者在其他服务器故障时派上用场。你的每台服务器都应该保持多于“足够”的负载能力，以备不时之需。

要确认配置的限度足够高，即使在服务器撤出将服务器池时还是继续工作的。举例来说，如果发现每台 MySQL 服务器典型的连接数为 100，那你就应该将池里的每台服务器的 `max_connections` 设置为 200。这样一来，即使池里的半数服务器都故障了，池还能继续处理同样数量的连接。

9.3.3 主服务器和多台从服务器之间的负载均衡

Load Balancing with a Master-Slave Setup

最常用的复制技术是一台主服务器和多台从服务器，在这个架构上，它们都不可或缺。许多应用都假设只有一个目标地址做所有的写操作，或者假设所有数据在某台服务器一直是可用的。虽说这个架构不太具有扩展性，但还是可以通过一些方法使负载均衡也能在它上面发挥良好的作用。下面的内容就是对其中的一些方法做一次浏览：

功能分区

446

通过为一个特别的用途配置从服务器或从服务器组，也能使负载能力获得一点点的提升。对于常见的那些功能，一般都可以把它们分为报表、分析、数据仓库和全文搜索。在本书 371 页的“自定义复制方案”你可以看到更多的内容。

过滤和数据分区

使用复制过滤技术，可以在各类似的从服务器中间把数据分区（查看 360 页的“复制过滤”）。如果主服务器上的数据已经分离到各个数据库或各个表里，这样做起来就更好了。不幸的是，没有一种内置的过滤方法在单独行的层次上实现这个目的。然而，也可以用这样的方法来实现行级别的过滤：将数据复制到一台分布式主服务器，使用带触发器的 Blackhole 表根据列值把每一行数据插入到不同的表里。

甚至可以采用更奇特的方法，比如复制到 Federated 表里，但是这可能会弄乱数据。Federated 表引入了内置服务器依赖技术来避免这个问题。

即使不把数据分区到各个从服务器上，你也可以通过读操作分离而不是随机分配来提高缓存的效率。举例来说，你可以将名字首字母为 A-M 的用户的所有读操作都分派到其中的一台服务器，而 N-Z 的分派到另外一台服务器。这样就能充分利用两台服务器的缓存，因为每次读数据时，更容易在缓存里找到相关的数据。最好的情况是，没有任何写操作，这样能使用的缓存相当于是两台服务器的缓存之和。作为对比，如果你随机地分配读操作到各个服务器上，那每台服务器的缓存里的数据都几乎一样，所以无论你有多少台从服务器，总的有效缓存数跟一台从服务器一样多。

将部分写操作移到从服务器上

主服务器未必总是处理所有的读操作。你可以从主服务器移走相当数量的冗余工作量到从服务器上，通过分解写操作，把其中一些步骤放到从服务器上去执行。更多内容请查看 299 页的“复制的过度时延”。

保证从服务器跟上更新速度

如果想在从服务器上运行某一个进程，它需要及时知道数据当前在哪个同步点上——哪怕需要等待一会儿后才会到那个点上——你可以使用 MASTER_POS_WAIT() 函数让这个进程等待，只到从服务器赶上主服务器上的那个同步点。另一个办法是，可以使用复制里的“心跳”技术让从服务器定时检查数据更新情况，不过它不提供秒级以下的时间粒度。更多内容请查看第 379 页的“衡量从服务器的时延”。

447 同步写入

你也可以使用 MASTER_POS_WAIT() 来确保写操作执行到一台或多台从服务器上。如果应用为了保证数据安全性，需要尽量赶上数据更新，那就可以在各个从服务器上轮流运行 MASTER_POS_WAIT()。这就像创建了一个同步围栏，需要很长时间才会轮一圈，即使各个从服务在数据同步上落后很多。这个办法可以在相对必须的时候采用（你也可以一直等待，直到一台从服务器收到事件信号，如果你的目标只是为了确保某台发出事件信号）。

9.4 高可用性

High Availability

许多人认为如果系统能响应用户的操作就是可用的。然而，可用性要比这个更复杂一些。一个应用是可以响应的，但是它可能处于“降级”模式了，它的一部分功能已经出了故障，只是内部的容错能力让它继续在运行。在做维护或其他特定情况下，你也可以让应用在只读模式下运行。举例来说，有着很多用户的图片共享网站上，用户不会介意总有一段时间里无法上传新照片；但另外一方面，ATM 机用户不希望看到屏幕上显示着“维护性只读模式”。所以，在可用性方面，网站可以这样，ATM 机就不能。

实现高可用性其实也很简单：建立冗余机制，当系统某部分故障时，能在线进行替换。其中，这个过程要快速且可靠。

TopSage.com

9.4.1 高可用性规划

Principles for High Availability

各个应用都有不同的可用性需求。在你开始为某一个正常运行时间的目标而全力以赴之前，要问一下自己：到底什么才是你真正想要达到的目的？可用性每提高一次，它的花费都会超过前一次，可用性的效果和费用是非线性比例关系的。

高可用性里的最重要原则是发现和消除系统里那个会导致全盘崩溃的最薄弱的环节。在头脑里想象你的整个应用，然后试着找出所有这样的环节：它是一块硬盘？一台服务器？一台交换机或者是一台路由器？或者是某个机架上的电源？你的所有机器都放在一个数据中心吗？或者“冗余”数据中心是同一家公司提供的吗？系统内那些没有冗余的部分都会是这样的薄弱环节。还有一些薄弱环节依赖于具体的服务，比如 DNS、单个网络服务供应商（你可以在机器上查看冗余网络连接是不是真的连到不同的骨干网上的）、单独一个电网。

试着去了解所有影响可用性的组件，做一下风险权衡，然后从那些风险最大的组件入手。一些人耗费大量力气在那种可以处理所有硬件故障的软件上，但是，这种软件上的 Bug 会导致更长的停工时间，这远胜于它所节省的时间。还有些人用各种冗余技术来建造“永不沉没”的系统，但是他们忘了数据中心那边可能会断电或者断网，或许他们也忘了恶意攻击者和程序员错误地删除或破坏了数据——比如不小心执行了 DROP TABLE——这些都会导致停工时间。

可以通过风险曝光系数来找出高优先级的风险因素，所谓的风险承担系数=故障的发生概率×故障带来的损失。画一张简单的表格，在纵列上依次是可能性、损失和曝光系数，这样就很容易找出你要优先处理的项目了。

你无法消除所有的“死穴”（Single Point of Failure, SPoF）。增加一个冗余组件并非总是可行的，因为有些限制条件无法克服，比如地理位置、预算和时间等约束。

下一步，假想一下当系统因故障、升级或应用更新时，你把系统切换（或失效转移）到了一个备用系统上。这时，任何导致应用不可用的部分都需要建立一个失效转移计划，同时，你也需要知道一次失效转移需要多长的时间。

一个相关的问题是在一次故障转移后，你替换一个故障的组件需要多长时间。除非恢复了系统的剩余能力，否则你会有更少的冗余和更多的风险。然而，有个备用系统并不是说就用不着及时地替换故障组件。让你准备一台备用服务器，装上操作系统，加载上最新的数据，一共需要多少时间？你有足够的备用服务器吗？你所需的应该不止一台。

另外一个要考虑的事情是：即使应用离线后，你会丢数据吗？如果一台服务器遭遇灾难性故障时，你总会丢失一些数据，比如最近的那些事务，它们刚写入到二进制日志里，还没来得及写到从服务器的转发日志上。你能容忍这样的情形吗？许多应用都可以。对此的替代性方案往往是昂贵的、复杂的，还有更多性能上的开销。举例来说，你可以采用 Google 的同步复制补丁（接下去会更多地提到它），或者把二进制日志放在一台独立的设备上，它通过 DRBD 来复制，这样即使系统彻底故障了，你也不会丢失数据。

一个聪明的应用架构常常能减少你在可用性上的需求，至少对系统的一部分而言是这样的，于是，高可用性的目标就更容易达成了。在你应用中把关键和非关键部分分离开来，能帮你节省大量的工作和金钱，因为为更小的系统提供冗余和高可用性会更容易做到。

通常来讲,过了某一点后,为应用保持高可用性、防止数据丢失会很难而且花费昂贵,所以一个现实的目标,以免这方面的工作做过头。幸运的是,对于大多数应用而言,能达到 99%或 99.9%的正常工作时间已经算是高可用性了。

449 9.4.2 增加冗余

往你的系统里增加冗余有两种形式:增加备用的能力和组件的复制。

事实上增加备用的能力很容易做到——你可以使用任何在本章中提到过的技术。增加可用性的一条途径是创建服务器集群或者服务器池,并增加一个负载平衡方案。当一台服务器故障时,其他服务器可以承担它的负载。还有个好办法是尽可能地让组件留有性能的余地,当服务器因为负载增加,或组件故障时,就有足够的提升空间来应付此时的性能问题。

从各方面来考虑,你需要为组件准备些备用品,当该组件故障时,它能立即顶上。要做个副件很容易,像网卡、路由器或硬盘——及其他所有你想到的很容易坏掉的设备。

要复制整个 MySQL 服务器就有点难了,因为服务器没有数据就没什么用。这意味着你的备用服务器也能访问到主服务器的数据。接下来我们要讨论就是如何做到这一点。

共享存储架构

共享存储是消除一些系统“死穴”的方法之一,常常与 SAN 一起使用(存储区域网,在第 325 页有更多介绍)。在这个策略里,活动的服务器挂接上文件系统,并完成正常操作。如果这台服务器宕机了,那备用服务器就能马上接上同一个文件系统,做一些必要的恢复操作,然后基于刚才那台服务器的文件之上,启动 MySQL。这个过程在逻辑上跟修复那台故障的服务器没什么两样,除了一点就是这个过程的速度要快很多,因为备用服务器已经处于启动状态,随时可以运行。文件系统检查和 InnoDB 的恢复是你可能遇到的最大延时。

共享存储也能用于消除一些数据丢失的场景,但它仍然作为一个“死穴”存在。如果它倒下了,那整个系统也倒下了。如果它故障时损坏了数据文件,那备用服务器也没法将它们恢复过来。我们强烈推荐使用 InnoDB 或别的事务性存储引擎来做共享存储。服务器崩溃最容易损坏的是 MyISAM 表,而修复它们要花很长的时间。

复制磁盘架构

复制磁盘技术是另一种在主服务器灾难性故障时保证数据安全的方法。在 MySQL 中,使用最多的磁盘复制技术是 DRBD (<http://www.drbd.org>),将它和来自 LINUX-HA 项目的工具一起使用(接下来会有更多关于它的内容)。

450 DRBD 是个同步的,块层次的复制技术,它以 Linux 核心模块的形式来实现,它从一个主设备通过网卡将每一块数据复制到另外一台服务器上的块设备(第二个设备),并在把块数据提交到主设备之前把它写下来(注 12)。

DRBD 只能在主动-被动模式下运行。被动设备是热备用机,你无法访问到它——不只是在只读模式下——除

注 12: 实际上你能调节 DRBD 的同步层次。可以将它设为异步的,让它一直等待,直到远程设备上收到了数据,或者让它一直堵塞,直到远程设备把数据写入到磁盘里。在此,强烈建议给 DRBD 配备一块网卡。

非它成为主机后。因为在第二设备上的写必须先于在主设备的写，所以，第二设备的性能至少要和主设备一样好，否则，它就会限制在主设备上写的性能。另外，如果使用 DRBD 来做一个可互换的备用机，那这台备用机的硬件配置必须要跟主服务器一样。

如果活动服务器出故障了，可以将第二设备提升为主设备。因为 DRBD 复制磁盘时是在块层次上完成的，而文件系统又可能变得不稳定，这就意味着最好是使用一个日志文件系统来做快速恢复。一旦服务器恢复了，接下来可能就是 MySQL 做自身的恢复了。如果第一服务器在恢复，它会假定自己是第二服务器的身份，与目前的主设备进行同步。

从实际的故障恢复过程而言，DRBD 跟 SAN 很相似：有一个热备用机器，它从故障机器那里取得相同的数据后开始工作。两者之间最大的差别是 DRBD 是复制存储——不是共享存储——所以，在 DRBD 里，你使用的是数据的一个复制本，而 SAN 使用的是同一个设备上的同一份数据。在以上两种情况下，当启动备用机上的 MySQL 时，它的缓存都是空的。相比之下，一个复制从服务器里有部分缓存可能已经“预热”了。

DRBD 有一些不错的结构和能力能防止集群软件常会遇到的问题。一个例子是“裂脑综合症”，这发生于两个节点同时提升为主服务器时。你可以通过配置 DRBD 来避免这种裂脑综合症的发生。然而，DRBD 也不是一个能满足每一个需求的完美解决方案。下面就让我们看一下它的缺点：

- DRBD 的故障恢复不是次秒级的。它至少需要 5 秒钟把第二设备提升为主设备，这还不包括必需的文件系统恢复和 MySQL 恢复。
- 因为必须在主动—被动模式下运行 DRBD，这使它显得很昂贵。当热备用服务器复制设备处于被动模式时，它不能被用来完成其他任务。这算不算是个缺点要取决于你的出发了。如果你真正需要高可用性，无法容忍主机故障时服务处于降级状态，但你也不能在其中一台服务器上运行两台服务器的负载量，因为如果这么做了，当它们中的一台故障时，你就无法处理这些负载了。你可以用备用机做些别的用途，比如复制从服务器，但是，你仍然是浪费了一些资源。
- 在这里，MyISAM 表特别地无用，因为它们需要很长的时间来检查和修复。MyISAM 在任何需要高可用性的系统里都不是个好选择，代之以 InnoDB 或别的存储引擎，它们会有很好的恢复性能。
- 它无法替代备份。如果是因为恶意干扰、过失、Bug 或硬件故障等原因你的数据被损坏了，DRBD 也帮不上忙：复制的数据只是被损坏的源数据的一个完美拷贝而已。你需要自己做备份（或者是 MySQL 延时复制），以此保证数据远离刚才提到的那些问题。

我们喜欢把 DRBD 用来对存储有二进制日志的服务器做复制。如果活动节点处故障了，你就可以在被动节点上启动日志服务器，然后使用这些恢复了的二进制日志，把从服务器设定到最近的二进制日志位置上（更多内容请查看第 374 页的“创建日志服务器”）。接下来，你就可以选择一个从服务器，把它升级为主服务器来替换崩溃的系统。

MySQL 同步复制

在同步复制中，主机上的事务直到把数据提交到一台或多台从服务器之后才会结束。同步复制也分好几个层次，它们都有各自常用的名字。MySQL 在本书写作的时候，还没提供同步复制功能，但是有许多第三方解决方案可供选择。其中一个就是 Google 的内部补丁。

Google 拥有大量的 MySQL 和 InnoDB 的补丁，给它们增加了许多额外的功能。其中一个就是半同步复制，主机在启动事务后，直到有一台从服务器接收到这个事件才会结束。Google 已经发布了 MySQL 4.0.26 和 5.0.37

的补丁。你可以在 <http://code.google.com/p/google-mysql-tools> 下载到这些补丁和相关工具。

还有个可选方法是 Solid Information Technology 公司的高可用性技术，它将此功能移植到了 solidDB for MySQL 里。这个解决方案在 MySQL 复制时，有几个优点：

- 从服务器不会落后于主服务器。
- Solid 在从服务器上使用多线程来写数据，在许多场合下提高了复制的性能。
- 用户可以自己配置在“安全”层次上的复制。在 1-SAFE 模式下，事务在主机上提交后就立即返回；2-SAFE 模式下，事务直到在从服务器上提交成功后才返回，并提供了多一层的安全保障来应付崩溃发生时的情况。

然而，这个方案只能用在 solidDB 存储引擎下，无法与 MyISAM \ InnoDB 及其他存储引擎一起使用。Solid 公司在将来可能会把更多的高可用性技术移植到 MySQL 上来。

除了以上两种 MySQL 自带的方案外，也可以使用中间件解决方案，比如 Continuent。

9.4.3 故障转移和故障恢复

Failover and Backup

故障转移是移除故障服务器，并用另外一台服务器代替的过程。在高可用性架构里，这是最重要的一部分。

在内容展开之前，让我们来定义一个新术语。在规范的情况下，我们是使用“故障转移”，而有些人把“故障切换”当作了它的同义词。有时，人们也说“主备切换”，它指的是按计划进行的主机和备用机的切换，不是故障之后的应对措施。

我们也使用术语“故障恢复”来说明故障转移后的服务器回迁过程。如果系统具备了故障切换能力，那么故障转移就是一个双向过程：在服务器 A 崩溃后，服务器 B 就替代了它；然后你修复了服务器 A，再把服务器 B 替换回来。

故障转移的缘由各不相同。我们已经在上文中讨论过它们中的一些了，因为负载均衡跟故障转移在许多方面是相似的，它们之间的分界线有点模糊。一般来说，我们想到的一个完整的故障转移方案，它在最小程度上至少能够监控服务器的运行，并在其崩溃时自动地替换掉它。这个过程对于应用而言，应该是透明。负载均衡不需要提供这样的功能。

在 Unix 的世界里，故障转移经常是使用 High Availability Linux 项目 (<http://linux-ha.org>) 提供的工具来完成的。这个项目——暂且不管它创建者的名字——是运行在许多类 Unix 操作系统上的。“心跳”工具提供了监控的功能，其他不同的工具完成 IP 接管和负载均衡等功能。你可以将它们与 DRBD 和（或）LVS 组合起来使用。

故障转移的最重要部分是故障恢复。如果服务器不能切换自如，那么故障转移就是个死胡同，只能加长停工时间。这就是我们喜欢对称复制拓扑学的原因，比如双主机配置，我们不喜欢用三台或更多的伴随主机来做环状复制。如果配置是对称的，那么故障转移和故障恢复在相对的方向上是一样的操作（在此值得一提的是 DRBD 内建了故障恢复的功能）。

在一些应用里，故障转移和故障恢复需要尽可能地快速和原子性。即使当无法达到这个要求时，你仍然最好不要让事情依赖于那些不可控制的因素上，比如 DNS 更改或者应用配置文件。一些最糟糕的问题只有当系统变得很庞大的时候才会显现出来，比如因为某些因素需要重新启动应用时，原子性的需要才开始让你感到困惑。曾

经在许多服务器上做过原子性的代码升级的人都知道这个很困难。

因为负载平衡和故障转移两者联系很近，有些硬件或者软件都是同时为这两个目的而设立的，我们建议你选择负载平衡技术时最好让它同时带有故障转移的能力。这也是我们说要避免用 DNS 和修改代码来做负载平衡的真实原因。如果你采用了这样的策略，就有额外的工作要做了；当你提高了高可用性之后，不得不重写那部分受到影响的代码。

下面部分我们要讨论一些常用的故障转移技术。

提升一个从服务器或者切换角色

将一个从服务器提升为主服务器，或者在主—主复制设置里切换活动和非活动角色，这些都是 MySQL 故障转移方案里很重要的部分。具体操作的细节请查看第 382 页的“改变主服务器”。

虚拟 IP 地址或 IP 接管

你可以通过把一个逻辑 IP 地址指定给一个要提供特定服务的 MySQL 实例来达到高可用性的目的。如果这个 MySQL 实例崩溃的了，你就把这个 IP 地址指向另外一个不同的 MySQL 服务器。这个核心思想跟先前我们写到过的内容相同（请查看第 441 页的“移动 IP 地址”），唯一不同的是现在用它来做故障转移，而不是负载平衡。

这种方法的好处是它对于应用而言是透明的。虽然它会退出当前连接，但是，不需要你去改变应用的配置。移动 IP 地址的过程也可以自动完成，因此，所有应用都能同时看到这一改变。当服务器在可用和不可用状态之间震荡时，这一特性尤其重要。

它的不足之处如下所述：

- 你需要在同一网段里定义好所有 IP 地址，或者使用网络桥接。
- 更改 IP 需要的系统的 root 权限。
- 有时需要更新 ARP 缓存。有些设备会将 ARP 信息存储很长时间，无法在你切换 IP 后立即把 IP 地址绑定到新的 MAC 地址上。
- 你要确保网络硬件设备支持快速的 IP 接管。有些硬件需要 MAC 克隆后才能正常工作。
- 有些服务器在功能部分损坏的情况还会继续占用 IP 地址，这样就需要你从物理上将其关闭或者从网络上断开。

浮动 IP 地址和 IP 接管可以很好地应付出现在两台临近（即同一子网里）的机器之间的故障转移。

等待变化传播开去

经常有这样的情况：当你在某一层定义了冗余之后，你不得不等待更低级的那层对此作出相应的改变。在本章前面的篇幅里，我们指出通过 DNS 来改变服务是种虚弱的方案，因为 DNS 的改变传播很慢。虽然改变 IP 地址给你更多的控制，但是在局域网里 IP 的改变还是要依靠更低级的层——ARP 来传播这一变化。

MySQL 主—主复制管理器

MySQL 主—主复制管理工具 (<http://code.google.com/p/mysql-master-master>), 或者简称为 mmm, 是一系列脚本, 用来执行监控、故障转移和主—主复制配置管理等任务。尽管它的名字如此, 但是它也对其他拓扑架构的服务器做自动的故障转移处理, 比如简单的主—从、主—主下的一个或多个从服务器。它使用抽象的角色概念, 比如读者或作者, 混合了固定 IP 和移动 IP。当它发现一台服务器故障时, 它会按照需要把 IP 地址指定到另外一台服务器, 从而变换了角色。它也能用在规划维护性故障转移或其他任务上。

这个工具通常的安装方法是建立一对副的主 MySQL 服务器, 每台上面运行着 mmm_agent 进程。你需要为它们逐个配置好 IP 地址、用户名、密码等信息。每个 mmm_agent 进程只知道自己所在的点。

这个方案里还有个单独的监控节点存在, 它的硬件配置用不着跟任何一个一样。它监视了那两个节点, 并处理故障转移——就是说移动“作者”角色。这样, 总共会有 3 个虚拟 IP 地址连接到 MySQL 服务器: 两个是“读者”角色, 一个是“作者”角色。可以用 mmm_control 程序显示和控制 MySQL 实例, 并根据实际需要来移动“作者”角色。

你可以将 mmm 跟其他技术混着来用 (比如前面我们提到过的 Google 的半同步复制补丁), 来进一步提高可用性和可靠性。

中间件解决方案

可以使用代理、端口转发、NAT、硬件负载均衡器等来处理故障转移和故障切换。然而, 它们也把自己作为“死穴”引入到了系统里, 你需要为它们准备冗余设备来避免这样的问题。

这个方案里比较好的一点是对应用来说, 远程数据中心就像在同一网络里似的。这使得你可以用浮动 IP 这样的技术让应用与完全不同的数据中心发起通信。你可以配置每个数据中心里的每一个应用服务器, 让它们通过自己的中间件进行连接, 这样, 它们就会访问连接都路由到活动的数据中心。图 9-7 描述了这个配置。

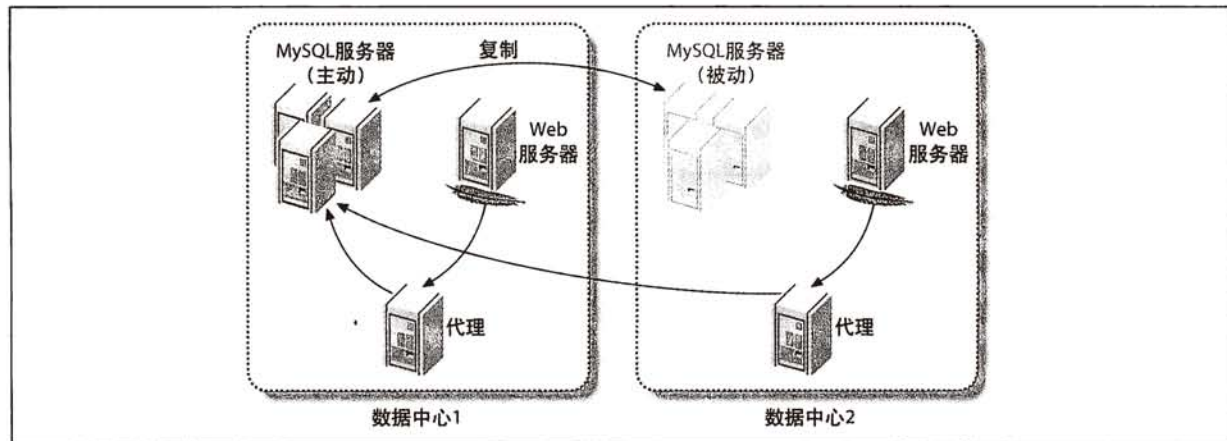


图 9-7: 使用 MySQL Proxy 在各数据中心间分派连接

如果这个活动数据中心的 MySQL 彻底崩溃了, 中间件就会把访问连接定向到另一个数据中心的服务器池, 而应用用不着随这个变化而改变。

这个配置方法的主要缺点是在一个数据中心的 Apache 服务器与另一个数据中心里的 MySQL 服务器活动 MySQL 服务器池的数据中心那里。你也可以使用 HTTP 代理来完成这个事情。

图 9-7 里显示了 MySQL 代理连接到 MySQL 服务器的情形,但是你可以将这个方案与其他中间件架构结合使用,比如 LVS 和硬件负载均衡器。

在应用中处理故障转移

有时候,让应用来处理故障转移会更简单更有弹性。举例来说,如果应用遇到一个错误,而外部观察者在正常情况下无法检测到这个错误,比如表明数据库数据损坏的一个错误消息,那么应用就能自己来执行故障转移过程了。

虽然把故障转移处理过程集成到应用中看上去很吸引人,但它往往运作得不好。许多应用都有着大量的组件,比如 cron 任务、配置文件、用不同语言写就的脚本。

因此把故障转移集成到应用里的做法用得并不广泛,特别是当应用逐渐增长,变得更加复杂的时候。

然而,把监控功能构建在应用里倒是个很好的主意,当它发现有需要,它就能**立即开始**故障转移过程。应用应该也能够管理用户体验,比如提供将功能降级的操作功能,显示相应的消息给用户。

应用层面的优化

本书若不讲解一章关于连接到 MySQL 的应用程序优化的内容，那就不能算完整，因为人们常常把一些性能方面的问题都归咎到 MySQL 身上。书里面我们更多地是讲到 MySQL 的优化，但是，我们不想让你错过这个更大的图景。一个糟糕的应用设计会使你无论怎么优化 MySQL 也弥补不了它带来的损失。实际上，有时候对于这类问题的答案是把它们从 MySQL 上脱离开来，让应用自己或其他工具来做这些事情，这样或许会有较好的性能表现。

本章不是构建高性能应用的参考书，我们只是希望通过阅读这一章让你避免那些常见的会伤及 MySQL 性能的小错误。下文我们以 Web 应用为主要讲解对象，因为 MySQL 主要是用在 Web 应用上的。

10.1 应用程序性能概述

对于更快性能的追求开始时很简单：应用响应请求花费了太长的时间，你总要为此做点什么吧。然而，真正的问题是什么呢？通常的瓶颈是缓慢的查询、锁、CPU 饱和、网络延时和文件 I/O。如果应用配置错误，或者不恰当地使用资源，以上任何一个因素都会引出一个大问题。

10.1.1 找出问题的根源

第一个任务是找出“肇事者”。如果你的应用具备了显示系统运行概况的功能，这做起来就简单了。如果你已经做到了这一步，但还是没法找出引起性能低下的原因，那你就需要增加更多的概况信息的调用，去找出那些要么缓慢要么被多次调用的资源。

如果你的应用因为 CPU 高占用率而一直等待，并且应用里有高并发性，那我们在第 55 页的“分析应用程序”所提到过的“丢失的时间”可能就成了问题了。鉴于此，有些时候在有限的并发条件下生成应用的概况信息是很有用的。

458 网络延时会占用大块的时间，哪怕是在局域网里。应用层面的概况信息已经包括了网络延时，因此，你应该在概况系统里看到网络往返延时带来的影响了。举例来说，如果一个页面执行了 1 000 个查询，即使每次只有 1 毫秒的延时，那累加起来也有 0.5 秒的响应时间，这对高性能应用来说已经是个很大的数目了。

如果应用层面概况信息收集得很充分，那就不难找出问题的根源。如果还没有内置概况功能，那就尽可能地加上它。如果你无法添加这个功能，那也可以试试第 76 页的“当你无法加入概况信息代码时”里提供的那些建议。这个总比钻研像“什么引起应用变慢”那样没头绪的理论设想要更快更容易。

10.1.2 寻找常见问题

Look for Common Problems

同样的问题我在应用里一次又一次地遇到，其原因往往是人们使用了设计糟糕的原有系统，或者采用了简化开发的通用框架。虽然这在某些时候能让你在开发一些功能时变得方便又快速，但它们也给应用增加了风险，因为你不知道它们底下是怎么工作的。这里有一张清单你应该逐个检查一下：

- 在各个机器上的 CPU、磁盘、网络 and 内存资源的使用情况如何？使用率对你而言是否合理？如果不合理，就检查那些影响资源使用的应用的程序基础。配置文件有时就是解决问题的最简单方法，举例来说，如果 Apache 耗光了内存，那是因为它创建 1 000 个工作者进程，每个工作进程需要 50MB 内存，这样，你就可通过配置文件配置这个应用能申请的 Apache 工作者进程数。你也可以配置系统，使之创建进程时少用些内存。
- 应用是否真正使用了它所取得的数据？一个常见的错误是：读取了 1 000 行数据，却只要显示 10 行就够了，其他 990 行就丢弃了（然而，如果应用缓存了余下的 990 条记录供以后使用，那么这可能是特意做的优化）。
- 应用里是否做了本该由数据库来做的处理？反之亦然。有个对应的例子是：读取了所有行的数据，然后在应用里计算它们的总数；以及在数据库里做复杂的字符串处理。数据库擅长于计数，而应用的编程语言擅长于正则表达式。你该使用正确的工具去干正确的活。
- 应用里执行了太多的查询？那些号称能“把程序员从 SQL 代码里解救出来”的 ORM (Object-Relational Mapping) 就因此常被人们责备。数据库服务器是被设计用来匹配多表数据的，因为要移除那些嵌套循环，代之以联接 (Join) 来做同样的查询。
- 应用里执行的查询太少了？我们只知道执行了太多的查询会成为问题。但是，有时“手工的联接”和与其相似的查询是个好主意，因为它们可以更加有效地利用缓存，更少的锁（尤其是 MyISAM），有时当你在应用的代码里使用一个散列联接时（MySQL 的嵌套循环的联接方法往往是低效的），查询的执行速度会更快。
- 应用是不是在毫无必要的时候还连到 MySQL 上去了？如果你能从缓存里读取数据，就不要去连数据库了。
- 应用连接到同一个 MySQL 实例的次数是不是太多了？这可能是因为应用的各个部分都各自开启了自己的数据库连接。有个建议在通常情况下都很对：从头到尾都重用同一个数据库连接。
- 应用是不是做了太多的“垃圾”查询？一个常见的例子是在做查询前才去选择需要的数据库。一个较好的做法是连接到名称明确的数据库，并使用表的全名做查询。（这样做，也便于通过日志或 SHOW PROCESSLIST 去查询情况，因为你可以直接执行这些查询语句，无需再更改数据库）。“准备”数据库连接又是另一个常见的问题，特别是 Java 写的数据库驱动程序，它在准备连接时会做大量的工作，它们中的大多数你都可以关闭。另一种垃圾查询是 SET NAMES UTF8，这纯粹是多此一举（它无法改变客户端连接库的字符集，它只对服务器有影响）。如果你的应用已确定在多数任务下使用的是某一个字符集，那你就可以避免这样无谓的字符集设置命令。
- 应用使用连接池了吗？这既是好事情也是坏事情。它限制了连接的数量，这在连接上查询数不多的情况下 (Ajax 应用就是个典型的例子) 是有利的；然而，它的不好的一面是，应用会受限於使用事务、临时表、连接指定的设置和定义用户变量。

- 应用使用了持久性连接吗？这样做的直接结果是会产生太多的数据库连接到 MySQL。这是个坏主意，除了一种情形：由于慢速的网络导致 MySQL 的连接成本很高，如本教程由站长百科收集整理一两个快速的查询，或者频繁地连接到 MySQL，那样你会很快用完客户端的所有本地端口（更多内容请查看第 328 页的“网络配置”）。如果你正确地配置了 MySQL，根本不需要持久性连接，可以使用“跳过名称解析”来防止 DNS 的查找，并确认该线程的优先级足够高。
- 即使没有使用，应用是不是还打开着连接？如果是，特别是当这些连接连向多台服务器时，它们可能占用了其他进程需要的连接。举例来说，假设你连接到 10 台 MySQL 服务器。由一个 Apache 进程占用 10 个连接数，这不是个问题，但是它们中只有一条连接是在任何指定时间里做着一些操作，而其他 9 条连接绝大多数时间都处于睡眠状态。如果有一台服务器响应变得迟缓，或者网络延时变长，那其他几台服务器就遭殃了，因为它们根本没连接可用。对于这个问题的解决办法是控制应用使用数据库连接的方式。

举例来说，你可以在各个 MySQL 实例中依次做批量操作，在向下一个 MySQL 发起查询前，关闭当前的所有连接。如果你要的是时间消耗很大的操作，比如调用一个 Web Service，可以先关闭与 MySQL 的连接，等这个耗时的调用完成后，再打开 MySQL 的连接，完成剩余的需要在数据库上操作的任务。

持久性连接与连接池的不同点比较模糊。持久性连接有与连接池相同的副作用，因为在各种情况下重新使用的连接往往都带有状态。

然而，连接池并不总是导致许多连接到服务器的联接，因为它们是队列化的，并在各进程间共享这些连接。在另一方面，持久化连接是基于每个进程来创建的，无法被其他进程所使用。

与持久性连接相比，连接池在连接策略上有更多的控制。你可以把一个连接池配置成自动扩充的，但是通常的做法还是当连接池满的时候，新的连接请求都被放在队列里等待。这使得这些请求都在应用服务器上等待，总比 MySQL 因为连接太多而超载。

有太多的方法使查询和连接更加快速，一般性准则是避免把它们放在一起，胜于试着把它们加速。

10.2 Web 服务器的议题

Web Server Issues

Apache 是 Web 应用中使用最广泛的服务器软件。在各种用途下，它都能运行良好，但如果使用得不恰当，它也会占用大量的资源。最常见的一个情况是让它的进程活动了太长的时间，并把它用在各种不同类型的任务下却没有做相应的优化。

Apache 经常在 prefork 配置项里使用 mod_php、mod_perl、mod_python。预分叉（Prefork）是为每个请求分配一个进程。因为 PHP、Perl 和 Python 等脚本语言运行起来很费资源，每个进程占用 50MB 或 100MB 内存的情形也不罕见。当一个请求处理完后，它会把绝大多数内存归还给操作系统，但不会是全部。Apache 会让这个进程保持在运行状态，以处理将要到来的请求。这就意味着如果这个新来的请求只是为了获得一个静态文件，比如一个 CSS 文件或一张图片，你都需要重新启用那个又“肥”又“大”的进程来处理这个简单请求。这也是为什么把 Apache 用作多用途 Web 服务器是件危险的事情。它是多用途的，若你对它进行了有针对性的配置，它才会有更好的性能表现。

另外有个主要的问题是如果你打开了 Keep-Alive 参数项，进程就会长时间地保持忙碌状态。即使你不这么做，

有些进程也会这样。如果内容是像“填鸭”一样传给客户端的，那这个读取数据的过程也会很慢。人们也经常犯这样的错误：按 Apache 默认开启的模块来运行。你可以按照 Apache 使用手册里的优势，把你不需要的模块都关闭掉，做法也很简单：查看 Apache 的配置文件，把不需要的模块都注释掉，然后重启 Apache。你可以从 php.ini 文件中把不需要的 PHP 模块都移除。

如果你创建了一个多用途 Apache 才需要的配置当作 Web 服务器来用，你最后可能会被众多繁重的 Apache 进程所拖垮，这些进程纯粹浪费你的 Web 服务器上的资源。而且，它们会占用大量与 MySQL 的连接，以至于也浪费了 MySQL 的资源。这里有一些方法能给你的服务器“减负”（注 2）：

- 不要把 Apache 用作静态内容的服务，如果一定要用，那也至少要换个另外的 Apache 实例来处理这些事情。常见的替代品有 lighttpd 和 nginx。
- 使用一个缓存代理服务器，比如 Squid 或 Varnish，使用户请求无须抵达 Web 服务器后才能被响应。即使在这个层面上你无法缓存所有的页面，你也能缓存大部分页面，并通过 Edge Side Includes (ESL, <http://www.esi.org>) 技术把页面上的小块动态部分放到缓存的静态部分里。
- 对动态内容和静态内容都设置过期策略。你可以使用缓存代理软件，像 Squid，去验证内容的明确性。Wikipedia 就是用这样的技术在缓存里移除内容已发生变化的文档。
- 有时你可能需要改变一下应用，使它能使用更长的超期时间。举例来说，如果你告诉浏览器要永久缓存 CSS 和 JavaScript 文件，然后又对这个网站静态 HTML 文件做了一些修改，这样这些页面的显示效果可能会变得很糟。对此，你需要使用一个唯一的文件名对每次修订后的页面文件都作一个明确的版本标记。举例来说，你可以自定义你的网站发布脚本，把 CSS 文件复制到/css/123_frontpage.css 目录下，这里的 123 就是 Subversion 里的修订号。你也可以用同样的方法来处理图片文件——不要重用原来的文件名，否则，即使你更新了文件内容，页面不会再被更新，不管浏览器要将原来的页面缓存多久。
- 不要让 Apache 与客户端做“填鸭”式通信。这不仅仅是慢，而且很容易招致拒绝性服务攻击。典型地，硬件化的负载均衡器会处理好缓存，Apache 就能很快地结束响应，然后让负载均衡器从缓存里读出数据去“喂”客户端。你也可以使用 lighttpd、Squid，或者设为事件驱动模式下的 Apache 作为应用的前端。
- 开启 gzip 压缩。现在的 CPU 很廉价，它可以用来节省大量的网络流量。如果你想节省 CPU 周期，那可以使用轻量级的 Web 服务器，比如 lighttpd，来缓存和提供压缩过的页面。
- 不要将 Apache 上的长距离连接配置为“保活”（Keep-Alive）模式，因为它会使 Apache 上臃肿的进程长时间处于运行状态。代替的方案是，用一个服务端的代理来处理“保活”的连接，使服务器免受这类客户端的伤害。如果将 Apache 与代理之间的连接方式设为“保活”，那是不错的主意，因为代理仅使用几个连接从服务器上读取数据。图 10-1 说明了以上两者的差异。

注 1：这种“填鸭”式过程发生在当一个客户端发起一个 HTTP 请求，但无法立即得到请求结果时。直到得到全部数据之前，这个 HTTP 连接及对应的 Apache 进程都将保持忙碌状态。

注 2：有一本关于如何优化 Web 应用的好书，名叫《High Performance Web Sites》，作者是 Steve Sounders (O'Reilly)。虽然它里面的大多数内容是从客户端的角度来讲怎样使网站运行得更快，但是他倡导的实践案例也适用于你的服务器。

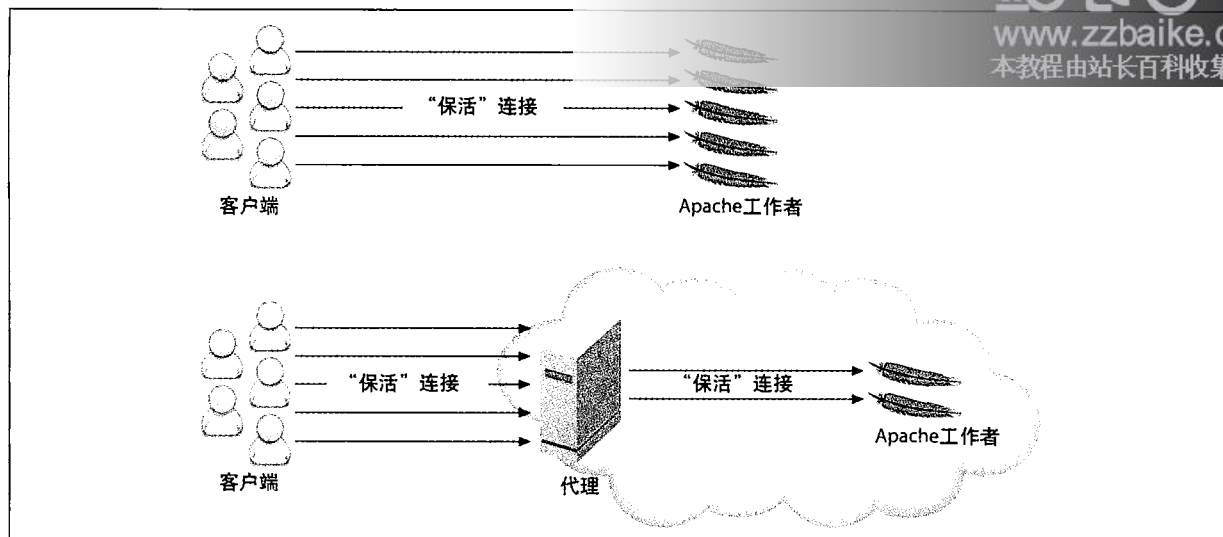


图 10-1：一个代理可以让 Apache 免受长久保持的“保活”连接的负担，从而可以使用更少的 Apache 工作者进程

以上这些策略应该可以帮助 Apache 减少进程的使用数，使你的服务器不会因为太多的进程而崩溃。然而，有些具体的操作仍然会引起 Apache 的进程长时间地运行，吞掉大量的系统资源。有一个例子就是查询外部资源时具有很高的延迟，比如访问一个远程 Web 服务器。这样的问题还是无法用上述那些方法来解决。

10.2.1 找到最佳的并发数

Finding the Optimal Concurrency

每个 Web 服务器都有它的一个**最佳并发数**——它的含义是服务器能同时处理的并发连接数目，它们既能尽可能快地处理客户端请求，又不会使服务器过载。这个“神奇的数目”需要做多次的尝试——失败的反复才能得到，相比于它能带来的好处，这还是值得一做。

对于大流量的网站而言，Web 服务器同时处理几千个连接是件很平常的事情。然而，这些连接中只有很少的一部分需要主动地去处理请求，而其他那些都是读取请求、文件上传、“喂”内容，或者仅仅等待客户端的下一步请求。

当并发数增加时，服务器会在某一点上达到它的吞吐量顶峰，在此之后，吞吐量会变得平稳，往往还会开始下降。更重要的是，系统的响应时间（延迟）开始增加。

想要知道究竟，就要设想如果你只有一颗 CPU，而服务器同时接收到 100 个请求，接下来会发生什么？假如一个 CPU 秒只能处理一个请求，而且你使用了一个完美的操作系统，没有任务调度的开销，也没有上下文切换的开销，那么这些请求总共需要 100 个 CPU 秒才能完成。

那么，怎样去做才是处理这些请求的最好办法？你可以把它们一个接一个放进队列里，或者对它们进行并行处理，每个请求在每一个轮回中都获得一样多的处理时间。这两种方式里，吞吐量都是每一秒一个请求。然而，如果使用队列，平均延迟有 50 秒（并发数=1），如果并行处理，那延迟有 100 秒（并发数=100）。在实际环境下，并发处理方法的平均延迟还会更高，因为其中还有个切换开销。

对于高 CPU 占有率的工作负载而言，其最佳并发数就是 CPU（或者是 CPU 里的核）的数目。然而，高 CPU 占有率的工作负载是可以运行的，因为它们会执行阻塞式调用，比如 I/O、数据库查询和网络请求等。因此，最佳并发数往往多于 CPU 数目。

你可以估计最佳并发数，但是这需要精确的分析模型。通常情况下，还是通过实验的方法比较容易，你尝试着不同的并发数，然后观察系统在降低响应时间前，能达到多大的顶峰吞吐量。

10.3 缓存

Caching

缓存对于高负载的应用而言极其重要。一个典型 Web 应用里，直接提供服务要比使用缓存（包括缓存校验、作废）多生成很多内容，所以，缓存能够将应用的性能提高好几个数量级。这个技巧的关键在于找出缓存粒度和作废策略的最佳结合点。同时，你需要决定缓存哪些内容，在哪里缓存。

一个典型的高负载应用有许多层的缓存。缓存不仅仅发生在你的服务器上：它出现在整个流程的每一个步骤上，包括用户的 Web 浏览器里（这就是网页头部的有关作废设置内容的用途）。通常而言，缓存越靠近客户端，就越能节省更多的资源，更加高效。一副图片从浏览器缓存里读出要好于从 Web 服务器的内存里读取，而后者又好于从服务器的磁盘上读取。每一种缓存都有其独有的特性，比如尺寸、延时等，在接下来的章节里我们将对它们逐一进行叙述。

你可以把缓存想象成两大类：**被动缓存**和**主动缓存**。被动缓存除了保存和返回数据不做其他事情。当你从被动缓存那里请求一些内容时，它要么给你需要的结果，要么告诉你“你要的数据不存在”。一个被动缓存的例子就是 memcached。

464

相反地，主动缓存在找不到请求的数据时，它会做点别的事情。一般就是把你的请求传递给应用的某一部分——它能生成请求所需要的内容，然后主动缓存就会存储这部分内容，并返回给客户端。Squid 缓存代理服务器就是一个主动缓存。

当设计应用时，你总希望你的缓存是主动型（也叫**透明型**）的，因为对于应用，它们可以隐藏“检查—生成—存储”这个逻辑。你可以在被动缓存之上构建你的主动缓存。

缓存并不总是有用

你需要确定缓存是不是真地提高了系统的性能，因为它可能一点用处也没有。举例来说，在实际应用中，从 lighttpd 的内存中读取内容要比从缓存代理那里读取快一些。如果那个代理的缓存是建于磁盘上的，那结论会更明显。

这个原因很简单：缓存也有自己的运行开销，它们主要检查缓存的开销和提供被命中缓存内容的开销，另外还有将缓存内容作废和保存数据的开销。只有当这些开销的总和小于服务器生成和提供数据所要的开销时，缓存才有用。

如果你知道所有这些操作的总开销，你就能计算缓存能起多大的作用。没有缓存时的开销就是服务器为每个请求生成数据所需要的总开销。有缓存时的开销就是检查缓存的开销，加上缓存没命中的可能性乘以生成这些数据的开销，再加上缓存命中的可能性乘以从缓存里取出这些数据的开销。

如果有缓存时的开销小于没缓存的时候的开销，那使用缓存就可以提高系统性能，但是也有例外。记在脑子里的一个例子就是从 lighttpd 内存里读取内容的开销要比代理从磁盘缓存上读取的开销要小，一些缓存总会比另外一些便宜。

10.3.1 在应用之下的缓存

MySQL 服务器有它自己的内部缓存，你也可以构建你自己的缓存和汇总表。你可以自定义缓存表，以便于更好地将它用于过滤、排序、与其他表做联接、计数，以及其他用途。缓存表比其他应用层的缓存更加持久，因为它们在服务器重启后还会继续存在。

我们在第 3 章、第 4 章里讲到过这些缓存策略，因此在本章里，我们的篇幅主要集中在应用层面和应用之上的缓存。

10.3.2 应用层面的缓存

典型的应用层面的缓存一般都是将数据放在本机内存里，或者放在网络上的另外一台机器的内存里。

应用层面的缓存一般要比更低层面的缓存有更高的效率，因为应用可以把部分计算结果存放在缓存里。因而，缓存对两类工作很有帮助：读取数据和在这些读取数据之上做计算。一个很好的例子是 HTML 文本的各个分块。应用能够产生 HTML 段落，比如头条新闻，然后将它们缓存起来。随后打开的页面里就能将这些被缓存起来的头条新闻直接放到页面上。通常来讲，缓存之前处理的数据越多，使用缓存之后能节省的工作量也越多。

这里有个不足之处就是缓存的命中率越多，要提高它而花费的钱就越多。假如你需要 50 个不同版本的头条新闻，能根据用户所在的不同地域来显示不同的头条。你需要有足够的内存来保存这全部 50 个版本的头条新闻，任何一个给定版本的头条被请求得越少，那它的作废操作也会越复杂。

应用缓存有许多种类型，以下是其中的一部分：

本地缓存

这种缓存一般都比较小，只存在于请求处理时的进程内存空间里。它们可用于避免对同一资源的多次请求。因此，它也没什么精彩之处：它往往只是应用程序代码里的一个变量或一个散列表。举例来说，如果需要显示用户名，而你只知道用户 ID，于是就设计一个函数叫 `get_name_from_id`，把缓存功能放在这个函数里，具体代码如下：

```
<?php
function get_name_from_id($user_id) {
    static $name; // static makes the variable persist
    if ( !$name ) {
        // Fetch name from database
    }
    return $name;
}
?>
```

如果你使用的是 Perl，那么 Memoize 模块就是缓存函数调用结果的标准办法：

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
    my ( $user_id ) = @_;
    my $name = # get name from database
    return $name;
}
```

这类技术都比较简单，但是它们能帮你节省大量工作。

本地共享内存式缓存

466

这种缓存大小中等（几个 GB）、访问快速，同时，难于在各机器间同步。它们适用于小型的、半静态的数据存储。举例来说，像每个州的城市列表、共享数据存储里的分块函数（使用映射表），或者应用了存活时间（Time-to-live, TTL）策略的数据。共享内存的最大好处是访问时非常快速——一般要比任何一种远程缓存要快很多。

分布式内存缓存

分布式内存缓存的最著名的例子是 memcached。分布式缓存比本地共享缓存要大，增长也容易。每一份缓存的数据只被创建一次，因为不会浪费你的内存，当同一份数据在各处缓存时也不会引起数据一致性问题。分布式内存擅长于对共享对象的排序，比如用户信息文件、评论和 HTML 片段。

这种缓存比本地共享缓存有更高的延迟，因此最有效的使用它们的方法是“多取”操作（比如在一次往返时，读取多个对象数据）。它们也要事先规划好怎么加入更多的节点，以及当一个节点崩溃时该怎么做。在这两种情形下，应用都要决定如何在各节点间分布或重新分布缓存对象。

当你在缓存集群里增加或减少一台服务器时，一致性的缓存对于性能问题就显得尤为重要。这里有一个用于 memcached 的一致性缓存库：<http://www.audioscrobbler.net/development/ketama/>。

磁盘缓存

磁盘是慢速的，所以，持久性对象最适合做磁盘缓存。对象往往不适合放在内存里，静态内容也是（比如预生成的自定义图片）。

非常有效地使用磁盘缓存和 Web 服务器的技巧是用 404 错误处理过程来捕捉没命中的缓存。加入你的 Web 应用要在页面的头部显示一个用户自定义的图片，暂且将这个图片命名为 /images/welcomeback/john.jpg。如果这个图片不存在，它就会产生一个 404 错误，同时触发错误处理过程。接着，错误处理过程就生成这个图片，并存放在磁盘上，然后再启动一个重定向，或者仅仅把这个图片“回填”到浏览器里，那么，以后的访问都可以直接从文件里返回这个图片了。

你可以将这项技巧用于许多类型的内容，举例来说，你用不着再缓存那块用来显示最新头条新闻的 HTML 代码了，而把它们放入一个 JavaScript 文件里，然后在页面的头部插入指向这个 js 文件的引用。

缓存失效的操作也很简单：删除这个文件就可以了。你可以通过运行一个周期性的任务，将 N 分钟前创建的文件都删除掉，来实现 TTL 失效策略。

如果想对缓存的尺寸做限制，那你可以实现一个最近最少使用（Least Recently Used, LRU）的失效策略，根据缓存内容的创建时间来删除内容。

467

这个失效策略需要你在文件系统的挂载 (Mount) 选项上开启“访问时间”这个开关 (noatime 挂载选项来达到这个目的)。如果这么做了, 你就应该使用内存文件系统来避免大量的磁盘操作。更多内容请查看第 331 页的“选择文件系统”。

10.3.3 缓存控制策略

Cache Control Policies

缓存引出的问题跟你数据库设计时违背了基本范式一样: 它们包含了重复数据, 这意味更新数据时要更新多个地方, 还要避免读到过期的“坏”数据。以下是几个常用的缓存控制策略:

存活时间

每个缓存的对象都带有一个作废日期, 用一个删除进程定时检查该数据的作废时间是否到达, 如果是就立即删除它, 你也可以暂时不理睬它, 直到下一次访问它时, 如果已经超过作废时间, 那才用一个更新的版本来替换它。这种作废策略最适用于很少变动或几乎不用刷新的数据。

显式作废

如果缓存里的数据过于“陈旧”而无法被接受, 那么更新缓存数据的进程就立即将该旧版本的数据作废。这个策略里有两个变体类型: 写一作废和写一更新。写一作废策略非常简单: 直接将该数据标志为作废 (也可以有从缓存里把它删除掉的选择)。写一更新策略就有更多的工作要做, 因为你还要用最新的数据来替换旧缓存数据。但是, 这个策略非常有用。特别是当生成缓存数据的代价很昂贵时 (这个功能在写的进程里可能已经具备)。更新了缓存之后, 将来的请求就用不着再等应用来生成这份数据了。如果你是在后台执行作废过程的, 比如是基于 TTL 的作废过程, 你可以在一个独立于任何用户请求的进程里生成最新版本的数据去替换缓存里已作废的数据。

读时作废

相对于在改变源数据时使缓存里对应的旧数据作废, 有一个替代性的方法是保存一些信息来帮你判断从缓存里读出的数据是否已经作废。它有个比显式作废更显著的优点: 随着时间的增长, 它开销是固定的。假设你要将一个对象作废, 而缓存里有 100 万个对象依赖于它。如果在写时将它作废, 你就不得不将缓存里的相关 100 万个对象都作废。而 100 万次读的延迟是相当小的, 这样就可以摊薄作废操作的时间成本, 避免了加载时的长时间延迟。

468 采用写时作废策略的最简单的方法是实行对象版本化管理。在这个方法里, 当把对象保存到缓存里时, 你同时要保存该数据所依赖的版本号或时间戳。举例来说, 假设你将一个用户在博客发表的文章的统计信息保存到缓存里, 这些信息包括了发表文章的数量。当将它作为 `blog_stats` 对象缓存时, 你同时也要把该用户当前的版本号也保存起来, 因为这个统计信息依赖于具体某个用户。

无论什么时候你更新了依赖于用户的数据, 也要随之改变用户的版本号。假设用户版本初始为 0, 你生成并缓存这些统计信息。当用户发表了一篇文章后, 你就将用户版本号改为 1 (最好将这个版本号与文章存放在一起, 尽管这个例子我们不必这么做)。那么, 当你需要显示统计信息时, 就先比较缓存的 `blog_stats` 对象的版本和缓存的用户版本, 因为这时用户的版本比这个对象的版本要高, 这样你就知道这份统计信息里的数据已经陈旧, 须要更新了。

这种用于内容作废的方法相当粗糙, 因为它预先假设了缓存里的依赖于用户的数据也跟其他数据进行互动。这个条件并不总是成立。举例来说, 如果用户编辑了一篇文章, 你也会去增加用户的版本号, 这使得缓存里的统

计数据都要作废了，哪怕真正的统计信息（文章的数目）实际上根本没发生变化。折中的方案是简单的缓存作废策略不仅仅要易于实现，还要有更高的效率。

对象版本化管理是**标签式缓存**的一个简化形式，后者可以处理更复杂的依赖关系。一个标签化缓存了解不同类别的依赖关系，并能单独追踪每一个对象的版本号。在上一章的图书俱乐部的例子里，你可以这样给评论做缓存：用用户版本号和书本版本号一起给评论做标签，具体像 `user_ver=1234` 和 `book_ver=5678` 这样。如果其中一个版本发生了变化，你就要刷新缓存。

10.3.4 缓存对象的层次化

David Sifian, *Memcached*

把对象按层次结构存放在缓存中，有助于读取、作废和内存使用的操作。你不仅要将对对象本身缓存起来，还要缓存它们的 ID 和对象分组的 ID，这样就能方便成组地读取它们。

电子商务网站上的搜索结果就是这种技术很好的例子。一次搜索可能返回一个匹配的产品清单，清单里包含了产品的名称、描述、缩略图和价格。如果把整个列表存放到缓存里，那读取时的效率是低下的，因为其他的搜索可能也会包含了同样的某几个产品，这样做的结果就是数据重复、浪费内存。这个策略也难以在产品价格发生变化时到缓存里找到对应的产品并使其作废，因为必须逐个清单地去查看是否存在这个价格变化了的产品。

一个可以代替缓存整个清单的方法是把搜索结果里尽量少的信息缓存起来，比如搜索的结果数目和结果清单里的产品 ID，这样你就可以单独缓存每一个产品资料了。这个方法解决了两个问题：一是消除了重复数据；二是更容易在单独产品的粒度上将缓存数据作废。

这个方法的缺点是你不得不从缓存里读取多个对象数据，而不是立即读取到整个搜索结果。然而，另一方面这也让你能更快地按照产品 ID 对搜索结果进行排序。现在，一次缓存命中就返回一个 ID 列表，如果缓存允许一次调用返回多个对象（Memcached 有一个 `mget()` 调用支持这个功能），你就可以用这些 ID 再到缓存里去读取对应的产品资料。

如果你使用不当，这个方法也会产生古怪的结果。假设你使用 TTL 策略来作废搜索结果，当产品资料发生变化时，明确地将缓存里对应的单个产品资料作废。现在试着想象一个产品的描述发生了变化，它不再包含跟缓存里搜索结果匹配的关键字，而搜索结果还没到作废时间。于是，你的用户就会看到“陈旧”的搜索结果，因为缓存里的这个搜索结果仍然引用了那个描述已经发生变化的产品。

对于多数应用来说，这一般不成为问题。如果你的应用无法容忍这个问题，那么就可以使用以版本为基础的缓存策略，在搜索之后，把产品版本号和搜索结果放在一起。在缓存里找到一个搜索结果后，把结果里的每个产品的版本号跟当前产品的版本号（也是在缓存里的）进行比较，如果发现有版本不符的，就通过重新搜索来获取新的搜索结果。

10.3.5 内容的预生成

Gregg Kinn, *Python*

除了应用层面上缓存数据之外，你还可以使用后台进程向服务器预先请求一些页面，然后将它们转换为静态页面保存在服务器上。如果页面是动态变化的，那你可以预生成页面中的一部分，然后使用一种技术，比如服务端整合，来生成最终页面。这样有助于减少预生成内容的大小和开销，因为本来你要为了各个最终页面上的

细微差别而不得不重复存储大量的内容。

缓存预生成的内容会占用大量空间，也不可能总是去预生成所有东西。无论哪种形式的缓存，预生成内容里的最重要部分就是请求最多的那些内容。因此，像我们在本章的前面提到过的那样，你可以通过 404 错误处理程序来对内容作“按需生成”。这些预生成的内容一般都放在内存文件系统里，避免放在磁盘上。

470 10.4 扩展 MySQL

Extending MySQL

如果 MySQL 完不成你所需要的任务，有一种可能性就是扩展它的能力。在这里，我们不是打算告诉你怎么做扩展，而是要提一下这个可能性里的一些具体途径。如果你有兴趣去深究其中的任何一条途径，那么网上有很多资源可供使用，也有很多关于这个主题的书可以参考。

当我们说“MySQL 完不成你所需要的任务”时，其中包含了两个含义：一是 MySQL 根本做不到，二是 MySQL 能做到，但是使用的办法不够好。无论哪个含义都是我们要扩展 MySQL 的理由。一个好消息是 MySQL 现在变得越来越模块化、多用途了。举例来说，MySQL 5.1 有大量可用的功能插件，它甚至允许存储引擎也是插件形式的，这样你就用不着把它们编译到 MySQL 服务器里了。

使用存储引擎将 MySQL 扩展为特定用途的数据库服务器是个伟大的想法。Brian Aker 已经编写了一个存储引擎的框架和一系列的文章、幻灯片来指导用户如何开发自己的存储引擎。这已经构成了一些主要的第三方存储引擎的基础。如果跟踪 MySQL 的内部邮件列表，你会发现现在有许多公司正在编写他们自己的内置存储引擎。举例来说，Friendster 使用一个特别的存储引擎来做社交图操作，另外，我们还知道有一家公司正在做一个用来做模糊搜索的引擎。编写一个简单的自定义引擎一点也不难。

你也可以把存储引擎直接用作软件某一部分的接口。Sphinx 就是个很好的例子，它直接与 Sphinx 全文检索软件通信（请查看附录 C）。

MySQL 5.1 也允许全文检索解析器插件，如果你能编写 UDF（请查看第 5 章），它擅长处理 CPU 密集的任务，这些任务必须在服务器线程环境下运行，对于 SQL 而言又太慢太笨重。因此，你可以用它们完成系统管理、服务集成、读取操作系统信息、调用 Web 服务、同步数据，以及其他更多相类似的任务。

MySQL 代理另外有一个很棒的选项，可以让你向 MySQL 协议增加你自己的功能。Paul McCullagh 的可扩展大二进制流框架项目 (<http://www.blobstreaming.org>) 为你打通了在 MySQL 里存储大型对象的道路。

因为 MySQL 是免费的、开源的软件，所以当你感觉它功能不够用时，你还可以去查看服务器代码。我们知道一些公司已经扩展了 MySQL 内部解析器的语法。近年来，还有第三方提交的许多有趣的 MySQL 扩展，涵盖了性能概要、扩展及其他新奇的应用。当人们想扩展 MySQL，MySQL 的开发者们总是反应积极，并乐于提供帮助。你可以通过邮件列表 internals@lists.mysql.com（注册用户请访问 <http://lists.mysql.com>）、MySQL 论坛和 IRC 频道 #mysql-dev 跟他们取得联系。

471 10.5 可替代的 MySQL

Alternatives to MySQL

MySQL 不是一个能适用于所有需要的万能解决方案。有些工作全部放到 MySQL 之外会更好，即使 MySQL 在

理论上也能做到。

一个很明显的例子是在传统的文件系统里对数据进行排序而不是在表里。图像文件是又一个经典的案例：你可以把它们都放在 BLOB 字段里，但是这在多数时候都不是个好主意（注 3）。通常的做法是把图像文件或其他大型二进制文件存在文件系统里，然后把文件名放在 MySQL 里。这样，应用就可以在 MySQL 之外读取文件了。在 Web 应用里，你可以把文件名放在 元素的 src 属性里。

全文检索也是应该放在 MySQL 之外处理的任务之一——MySQL 不像 Lucene 或 Sphinx（请查看附录 3）那样擅长于这类检索。

NDB API 可以被用于某一类型的任务。比如，虽然 MySQL 的 NDB Cluster 存储引擎不适合在高性能要求的 Web 应用中作排序操作，但是可以通过直接使用 NDB API 来存储网站的 session 数据或用户注册信息。关于 NDB API，你可以访问 <http://dev.mysql.com/doc/ndbapi/en/index.html> 来获取更多信息。Apache 上也有相应的 NDB 模块，你可以从 <http://code.google.com/p/mod-ndb/> 下载。

最后，对于有些操作，比如图形化的关系、树的遍历，关系数据库并不擅长做这些。MySQL 也不擅长分布式数据处理，因为它缺少并行查询的执行能力。你可能需要使用别的工具（与 MySQL 一起使用）来达到这一目的。

注 3：使用 MySQL 复制功能能快速地将图像文件发布到其他机器上。据我们所知，一些应用使用了这项技术。

备份与还原

Backup and Recovery

人们很容易把重点放在“正经事”上，却忽视了备份和还原。实际上，紧迫的往往不是重要的，同样，重要的也未必显得很紧迫。备份在高性能应用里的重要性跟灾难还原一样，你需要从一开始就规划、设计好备份方案，这样在系统崩溃时，你就可以减少停机时间、性能缩水等负面影响。

如果事先没有规划、部署备份方案，那你就只能在事后创建“插入式”的解决方案了。在那个时候，你才发现之前在系统设计时做的一些决策已经堵住了处理高性能备份的最佳之路。举例来说，你已经架设好了一台服务器，然后又认识到需要一个 LVM 来做文件系统的快照——但为时已晚。你在为系统配置备份参数时，可能也不会注意到那几个会影响到性能的重要选项。同样，如果你没有规划和演练过系统还原预案，那到了真地要用它的时候，进展就不会顺利。

备份系统就像监控报警系统：许多系统管理员已经一次或多次重新“发明”过备份软件，这真非常可惜，因为市面上已经有许多质量上乘、支持完善、扩充性好的备份软件了——其中一些还是开源和免费的，我们鼓励你采用它们中对你有所帮助的那些。

在本章里，我们的内容不会覆盖一个设计良好的备份/还原方案的每一个环节。仅仅是这一章的标题就足够写出一本书——实际上，没几本书是专门讲这个问题的（注 1）。跟本书的第一版相反，我们假定许多读者在 MyISAM 之外还使用 InnoDB，或者用 InnoDB 代替 MyISAM。那种混合使用的状态使某些应用场景下的备份工作变得更复杂。

11.1 概况

Overview

在本章开始前，我们会先回顾几个术语，并讨论在规划自己的备份/还原方案，以及考虑未来潜在的需求时，你要记住的几个要点。然后，我们比较概括地讲述做备份时可供采用的技术，并在还原数据和灾难还原方面做一点探索。接着，我们会讨论如何选择一款适用于备份的工具。在文章的最后，我们用几个实例来讲解如何构建你自己的备份工具。

11.1.1 术语

Terminology

在开始之前，让我们澄清一些关键术语。首先，你将会经常听到热备份、温备份和冷备份这样的说法。人们常常用它们来形容备份对系统的影响，比如，“热”备份不需要任何一台服务器停机。问题是这三个术语对每个人

注 1：我们认为 W. Curtis Preston 的《Backup & Recovery》(O'Reilly)是个不错的选择。

来说并不是指同一件事情。有些工具甚至把“热”放在它们的名称里，但是实际上，它们做不到像热备份表示的那样热备份。我们会尽量避免使用这些术语，直接告诉你一个特定的技术或工具会对系统有多大的影响。

还有两个容易混淆的词：恢复和还原。本章中我们把它们用在特定的途径下。恢复的意思是从备份记录中获取数据，然后把它们加载到 MySQL 里，或者放在 MySQL 能找到的地方。还原在通常情况下的意思是描述援救整个系统或系统一部分的过程，其中就包含了将数据从备份记录中还原出来的操作，以及其他让系统恢复全部功能的必要步骤（比如重启 MySQL、更改配置、预热服务器缓存等）。

对许多人而言，还原仅仅指在服务器崩溃时修复损坏的数据表，不同于还原整个服务器。一个存储引擎的还原还要使数据和日志文件重新统一起来，它要确保只有生效的事务所作的更新才能被写到数据文件里，然后再把日志文件里还未提交的事务重新提交。如果你使用的是事务性存储引擎，上面这个步骤就是整个还原过程的一部分了，甚至是制作备份记录的一部分。然而，这跟还原还不太一样，比如有时在意外的 DROP TABLE 之后，也需要这样来恢复数据。

11.1.2 所有关于还原的内容

It's All About Recovery

如果一切都运转正常，你就永远不需要考虑还原的事情。但是，当你需要还原时，那世界上最好的备份系统也帮不上忙，因为你需要的是一个很棒的还原系统。

但问题是使你的备份系统顺畅运转要比构建一个优良的还原过程和工具更加容易。以下就是具体原因：

- 先要有备份。如果没有事先的备份，你根本无法还原。所以在你构建一个系统的时候，你的注意力很自然地集中在备份上。因此，先对还原进行规划是相当重要的。实际上，在列出系统还原方面的需求之前，你就别想着构建备份系统。
- 备份是例行公事。这使你关注备份过程的自动化和调校，而没去考虑其他问题。在你的备份问题多盯上 5 分钟也不见得有多重要，但是在每一天里你关注过你的还原吗？你应该特意安排演练你的还原过程，直到它像你的备份过程一样顺畅、没有 Bug。
- 备份一般都不是在极大压力之下才做的，但是，还原往往发生在一个紧要关头，因此无论怎么强调还原的重要性都不会过分。
- 安全问题的妨碍。如果正在做离站式备份，你可能会去加密这些备份数据，或者用别的办法去保护它。这时，你一般把注意力放在当你的数据安全性受到危害之后会带来多少损失，而忽视当需要还原这些数据却没有人能够解开你的加密卷时，带来的损失又是多少——或者当你需要从一个巨大的加密文件里解出那个文件时！
- 一个人就能规划、设计和实现备份系统，特别是有优秀的工具辅助时。当灾难来袭时，一个人可能就无法应付了。你需要培训好几个人来为系统还原做准备，这样就不会临时去找不合格的人来做数据还原了。

这里我们举一个取自真实世界的例子：一个客户报告说使用 mysqldump 做备份时，如果加上 -d 参数，备份速度就会快得像闪电一样，所以，他想知道为什么没有人提到用这个参数来加速备份过程的效果有这么大。如果这位客户试着恢复过这些备份记录，就会难以忘记其中的原因：-d 参数没导出数据！这位客户只关注备份，却没在意还原，所以完全没意识到这个问题。

当你开始考虑还原的时候，有个不错的主意是在你采取任何具体措施前先定义好你的需求，要考虑进去：

- 在产生严重后果前，你会丢失多少数据？你需要即时点（Point-In-Time）还原吗？或者说把最近一次正式备份以来的所有数据都丢失了，你能接受吗？有法律上的麻烦吗？
- 还原需要有多快？哪一种故障停工可以被你接受？哪种影响（比如局部不可用）对于你的应用和用户能够被接受？当上述场景发生时，你要构建怎么样的功能来维持系统的原有机能？
- 你确实需要还原吗？通常的需求是还原整个服务器、一个单独的数据、一张表、或者是一些指定的事务或语句。

475 针对这些问题，写下你的答案，并把它们加入系统文档里，在阅读本章的余下部分时，你要时刻牢记这些问题。先做这些练习可以帮助你注意力集中在还原上，如同规划你的备份方案时一样。把它们写在文档里是为了使你将来把这些步骤重新过一遍时更加方便。

备份神话 1：我把复制当备份用

这是我们经常遇到的一种误解。一个复制从服务器不是一个备份设备，RAID 阵列也不是。想知道为什么，就请这么来考虑：如果你在数据库上意外地执行了 DROP DATABASE 命令，你还能找回你的数据吗？RAID 和复制都通不过这个简单测试。它们不是备份，也不是备份的替代。只有备份才能满足备份的要求。

11.1.3 没有涉及的主题

Topics Not Worth Discussing

备份 MySQL 是众多普通的备份/还原方法中比较特殊的一种。我们想专注于高性能 MySQL，但是，不谈及一些其他的话题也会有一点困难，特别是因为我们看到过许多人都在同样的备份/还原问题里挣扎。以下是我们打算涉及的主题：

- 安全问题。（诸如谁能访问到备份记录、谁有恢复数据的权限、哪些文件需要加密。）
- 备份文件该存放在哪里，包括备份记录要离原始数据多远（在不同的磁盘上、不同的服务器上、或者是离站式的），怎么把数据从源移到目的地。
- 保留策略，审核，法律方面的要求及相关主题。
- 存储解决方案和存储媒体、压缩方案及增量备份。
- 存储格式（这里我们要多说一句：避免使用私有版权的备份格式）。
- 备份方案里的监控和报告功能。
- 内建于存储层的备份功能，或者像预制文件系统那样的特殊设备。

这些都是重要的主题。如果你对它们还不太熟悉，那最好去阅读一本关于备份的书籍。

11.1.4 大图景

The Big Picture

在开始引入所有可用选项的诸多细节前，先提供我们关于大多数人在构建一个备份/恢复系统时可能会需要的意见。你可以把这些建议作为一个出发点，或者工作方向的一个指引：

- 裸备份对大型数据而言是特别有必要的。它运行速度很快——这一点很重要。基于快照的备份是我们最喜欢选用的，如果使用的都是 InnoDB 表，那么 InnoDB 热备份就是很好的选择。
- 备份用于即时点还原的二进制日志。
- 多保留几个备份记录，二进制日志要足够长以便于你能从它们那里恢复出数据。
- 定期测试你的备份/还原过程，尤其是整个还原过程。
- 创建逻辑备份。（从效率考虑，可能还是基于你的裸备份来做较好。）
- 如果可能，要测试一下你的裸备份，以确保它们可以被用来做还原。在把它们复制到目的存储地址之前，即在备份的过程中测试它们。
- 对安全问题考虑周全。如果有人危及服务器安全时，他能获得访问备份服务器的权限吗？反之亦然。
- 使用备份工具来监控备份记录和备份过程。你需要外部工具来核实备份操作是没问题的。
- 灵活运用机器间复制文件的方法，有许多复制方法比 scp 和 rsync 更高效。更多内容请查看附录 A。

476

11.1.5 为什么要备份

Why Backup?

如果你正在构建一个依赖于 MySQL 的高性能系统，备份是很重要的。以下就是它的原因：

灾难还原

当你遭遇到硬件崩溃、恶意 Bug 破坏你的数据，或者服务器或数据因为某个其他的原因（这种可能的原因会很多，而且不尽相同——发挥你的想象力去想吧）不能继续使用的时候，你就需要做灾难还原了。以上任何一种灾难发生的可能性相当低，而且，叠加起来一起发生的可能性更低。但是你要为以下这些可能做好准备：某个人碰巧错连到了你的服务器，然后执行了 ALTER TABLE 命令（注 2）；一把火烧毁了你的机房；对你的服务器发动了恶意攻击；MySQL 的一个 Bug。

人们改变了他们的想法

你会很惊奇地发现我们会经常需要把部分数据还原到它们在以前某个时间点上存在过的状态。对于一些应用而言，这样的情况的发生频率要比灾难高很多（比如说一个重要客户以前删除了一些数据，现在又想把它们还原回来）。

477

注 2：Baron 仍然记得当他还在一个电子商务网站做开发人员时，他在另外一个命令窗口里输入了这样一条命令，结果……这件事情也是 DBA 团队的过失，因为他们不该把当前服务器上的这个修改权限开放给开发人员使用。

审查

有时你需要知道数据或式样在过去的某个点上看起来是什么样子的。举例来说，你可能在应用里发现了一个 Bug，现在想看看这段代码在过去运行时产生了什么样的结果（有时仅仅把代码放入版本控制软件还是不够的）。

测试

用真实数据做测试的最简便的方法是定时地把最新的产品数据刷新到测试服务器上。如果你已经做了备份，那就更简单了——使用备份数据就可以。

请检查你所有关于备份的假设。举例来说，你是不是假设共享主机提供商在用你的账号备份 MySQL 服务器？虽然共享主机跟高性能没什么关系，但我们想指出这样一个假设是靠不住的。（为了免得你再怀疑，我们可以说许多主机提供商根本不备份 MySQL 服务器，其他几家也只是在服务器运行的时候，做个数据库文件拷贝而已，这样的数据库文件拷贝很可能是损坏的，没法用的。）

11.2 要权衡的事项

备份 MySQL 要比看上去更难。从最基本的层面看，一个备份记录就是一份数据的副本，但是，诸如你应用所需的、MySQL 存储引擎架构、系统配置等因素使它操作起来变得困难。

11.2.1 你能承受的损失有多大

知道你能承担多大的数据损失可以用来指导你的备份策略。你需要一个即时点还原功能吗？或者说它是否可以被用作昨晚的数据备份，而不管期间丢失的那些数据？如果你需要即时快照式的还原，那你可能需要做常规备份，确保日志功能已经打开，这样，你就可以通过这个日志来恢复备份数据，并将它们还原到你需要的时间点上。

通常而言，你能承担的损失越大，备份就越容易做。如果你有严格要求，那就很难确保能还原所有的东西。不同的即时点还原有着不同的需求：“软”即时点还原的需求意味着你最好能重新创建你的数据，只有这样，数据状态才能跟问题发生时的那个时间点“靠得够近”。“硬”即时点还原的需求意味着你无法容忍任何已提交事务的丢失，不管何种糟糕事情的发生（比如服务器着火了）。这就需要一些特殊的技术，比如说把二进制日志放在一个单独 SAN 卷里，或者使用 DRBD 磁盘复制。更多内容请查看第 9 章。

11.2.2 在线备份还是离线备份

如果可以做到，停掉 MySQL 后再做备份是最简单、最安全的方法，而且还能在最小的“脏数据”和源-备份矛盾的风险下，获得与源数据库相一致的备份记录。如果你停掉 MySQL，你就可以放心去复制数据，不会碰到类似 InnoDB 缓冲池里的“脏缓存”之类的难题，也不用担心复制数据时这些数据被更新了。而且，这时系统没有来自应用的负荷，备份的速度因此会更快。

然而，让一个服务器离线的成本要比看起来高得多。即使把停机时间减到最小，关闭和启动高负载大数据量的

MySQL 也要耗费很长的一段时间：

- 如果在 InnoDB 的缓冲池里有很多“脏缓存”——它的意思是那些数据在内存里已经被更新了但是还没写到磁盘上去——InnoDB 要过很长的一段时间才把这些更新了的数据写到磁盘上。你可以使用 `innodb_fast_shutdown` 配置变量来影响 InnoDB 的关闭时间，这个变量可以控制 InnoDB 处理缓冲池的方式，让它在关闭之前把缓存更新到磁盘上（注 3）。这个方法并不能从根本上解决问题，只是把问题转移了。你仍然不能显著地减少 MySQL 的关闭-开启时间。有时，你可以通过在别的方面配置 InnoDB 来改善这个情况，但这些更改会对性能产生更大的影响。更详细的内容请查看第 281 页的“MySQL I/O 调优”。
- 重启也会消耗很长的时间。打开所有的表、预热缓存都是缓慢的过程，特别是表及其数据很多的时候。如果你设置了 `innodb_fast_shutdown=2` 来加快 InnoDB 的关闭速度，那在下次运行之初，InnoDB 就不得不执行一次还原过程。即使在你的服务器看似已经全部启动好了，它也要花费很长的时间来预热和做一些其他准备工作。

因此，如果你要求高性能构建你的应用，你就必须仔细设计你的备份方案，让你的产品服务器无需离线也能备份。从一致性需求来看，纵然备份可以在线进行，仍然意味着它会很明显地影响原来的系统服务。

举例来说，在许多备份方法里最常见的示例之一就是 `FLUSH TABLES WITH READ LOCK` 开始的。这段代码告诉 MySQL 去刷新（注 4）所有表的缓存然后锁定，同时刷新查询缓存。

这个需要好一会儿才能完成。（很难确切地说需要多长时间。如果全局的读锁不得不等待一个很长的语句执行结束，或者你有许多表，这个时间都会变得更长。）在锁被释放之前，你无法修改服务器上的数据，跟服务器关机相比，`FLUSH TABLES WITH READ LOCK` 的代价也不算昂贵，因为大多数缓存还放在内存里，服务器还是“温热”，但是这个方法相对来说还是带有破坏性的。

如果这构成了问题，那你就需要找到一个可替代的方法。我们使用的一个方法是在复制从服务器上做一个备份，这台服务器是从服务器池里的一台，所以离线-上线的成本很低。让我们回到本章的主题，其他关于在线和离线备份的内容，在本章后面会讲到。现在，我们只能这么说：做在线备份，同时又不影响到服务器上的服务，这很难做到。

11.2.3 逻辑备份还是裸备份

备份 MySQL 数据有两种方法：做一个逻辑备份（也叫做“导出”）和复制裸文件。一个逻辑备份里的数据是 MySQL 能识别的格式，它或者是 SQL，或者是带分界符号的文本（注 5）。裸文件就是原来存储在磁盘上的那些文件。

两种复制数据方法里的每一种都有各自的优缺点。

注 3：插入操作的缓冲区存储在 InnoDB 的表空间文件里，与其他数据放在一起；一个后台进程最终会把这些插入的记录归并到它们对应的表里。

注 4：刷新 MyISAM（不是 InnoDB）里的缓存到磁盘。

注 5：由 `mysqldump` 生成的逻辑备份不总是文本文件。从 SQL 导出的数据包含了许多种不同的字符集，甚至还会包含无法作为可见字符显示的二进制数据。而且，对于许多编辑器而言，它们的行也可能太长。尽管如此，许多备份文件还是可以被文本编辑器打开并阅读的，特别是当你使用带有 `-hex-blob` 选项的 `mysqldump` 生成备份时。

逻辑备份

逻辑备份有以下这些优点：

- 它们都是普通的文件，可以使用编辑器或命令行工具（比如 `grep`、`sed`）来操作和检查。这在恢复数据时很有用，在只是检查一下数据而不做恢复时也一样用得上。
- 它们很容易被恢复。你可以利用管道（Pipe）把它们传入 `mysql` 或使用 `mysqlimport` 导入。
- 可以跨网络进行备份和恢复——不在这台 MySQL 主机上，而在其他机器上进行。
- 它们有很强的兼容性，因为 `mysqldump`——许多人都喜欢采用这个工具（注 6）——能接受许多选项，比如可以用 `where` 语句来限制哪些数据行需要备份。
- 它们独立于具体的存储引擎。因为你是从 MySQL 服务器上把它们取出来的，已经对底层数据引擎的差异做了抽象。因而，只需少许工作就能把从 InnoDB 里备份出来的表恢复到 MyISAM 表里。这在裸文件副本里是做不到的。
- 在许多情况下，如果你使用 `mysqldump` 时指定了正确的选项，你甚至可以把你的逻辑备份记录导入别的数据库服务器，比如 PostgreSQL。
- 它们可以避免数据损坏。如果你复制裸文件时，磁盘驱动器突然坏掉了，你就只能得到一个损坏了的数据备份。除非你特地检查一下这个备份，否则不会注意到这一点，直到将来要用到它的时候。如果这些 MySQL 数据的内存还是完好的，那么，逻辑备份有时还可以得到一个可信赖的备份，而一个完好的裸文件副本则已经是不可能的。

逻辑备份也有如下这些缺点：

- 服务器必须亲自来生成它们，所以会使用更多的 CPU 周期。
- 逻辑备份在某些时候会比原来数据文件更大（注 7）。用 ASCII 表示的数据在效率上有时不及存储引擎存储数据时使用的格式。举例来说，一个整数在存储里是 4 个字节，写成 ASCII 码后，它最多的时候需要 12 个字节。你可以有效地使用压缩功能，但这会使用更多的 CPU 资源。
- 浮点数会因此丢失精度信息，这妨碍了备份记录中数据的精确恢复。（Google 对 MySQL 的一些补丁里就包括了一个针对 `mysqldump` 中此类精度问题的。）
- 从逻辑备份里恢复出数据需要 MySQL 加载和解释这些声明，并重建索引，这将给服务器增添更多的工作量。

这里面最大的不利因素是从 MySQL 导出数据的开销和使用 SQL 语言把数据导回去的开销。

裸备份

裸备份有下面这些优点：

- 裸文件备份仅仅要求你把那些文件复制到其他地方，就算备份了。裸文件不要你做任何额外的工作来生成。
- 恢复裸备份的数据更加简单，不管是基于哪个存储引擎。对于 MyISAM，就是把文件复制到指定的位置。

注 6：也有其他可选择的，有一类工具它们能并行导出和恢复数据，但这个是最常用的工具。

注 7：在我们的经验里，逻辑备份文件往往比裸备份的小，但它们不总是这样的。

对于 InnoDB，它要求你先关闭 MySQL 服务器，然后再完成余下的步骤。

- 裸备份往往便于跨平台、操作系统和 MySQL 版本。
- 恢复裸备份里的数据的速度更快一些，因为 MySQL 不需要执行任何 SQL 语句，也不需要重建索引。如果备份里包含了无法整个放入服务器内存的 InnoDB 数据表，那恢复数据更会快得多。

以下是裸备份的一些不利之处：

- InnoDB 的裸文件经常要比相应的逻辑备份大很多。一般在 InnoDB 的表空间都保留了大量闲置的空间，这里面相当多的是用于存储数据之外的用途（比如插入操作的缓冲区、回滚段等）。
- 裸备份也不总是可以轻易地跨越平台、操作系统和 MySQL 版本。文件名的大小写敏感和浮点数格式都会成为麻烦的根源。你不能把这些备份移到浮点数格式不一样的服务器上去恢复（但是，目前大量的主流处理器都使用 IEEE 的浮点数格式）。

裸备份通常更容易做，也更有效率，但是你不能把它当作长期手段或合理的需求。你至少要定期地做一次逻辑备份。

在你测试之前，不要想当然地认为这个备份（特别是裸备份）是好用的。对于 InnoDB 而言，这意味着要开启 MySQL 实例，然后运行 InnoDB 还原，再运行 CHECK TABLES。你也可以跳过这一步，直接在备份文件上运行 innchecksum——但是，我们不建议这么做。对 MyISAM 而言，你应该运行 CHECK TABLES，或者使用 myisamchk。

一个比较灵活的方法是采用这两种备份混合方式：先做一个裸副本，然后开启一个 MySQL 服务器，在这些副本基础上生成逻辑备份。这样你就获得两者的优点，不会使服务器在导出时被加上过重的负担。当你有文件系统快照的功能时，这个方法会显得特别方便——你可以做一个快照，然后把快照复制到另外一台服务器上释放出来，测试这些裸文件，最后做一个逻辑备份。

11.2.4 要备份什么

What to Back Up

你的还原需求会告诉你要备份哪些东西。最简单的策略是只备份数据和表定义，这算是最最少的备份内容了。通常来讲，为了能还原服务器，你需要备份大量的更多的东西。以下这些是在你做备份时应考虑的备份内容：

不显眼的数据库

不要忘记那些容易被忽视的数据，诸如二进制日志、InnoDB 事务日志。

代码

现代 MySQL 服务器能存储大量代码，比如触发器、存储过程。如果备份了 mysql 的数据库，那你也需要备份这些代码。但是，之后就很难以此还原出一个完整的数据库，因为在那个数据库里的某些“数据”——就像存储过程之类的，实际上只保存在 mysql 数据库里面。

复制的配置信息

如果你正在恢复一台带有复制的服务器，你也要把你需要的所有复制文件备份过来，比如二进制日志、转发日志、日志索引文件及 .info 文件。在最低要求上，你应该包含 SHOW MASTER STATUS 和/或 SHOW SLAVE STATUS 的所有输出内容。使用 FLUSH LOGS 使 MySQL 开启一个全新的日志文件也很有用处，在做即时点

还原时，从日志文件起点开始还原要比从中间开始更加容易。

服务器的配置信息

如果你要将服务器从一个现实世界的灾难还原过来——比如说地震过后，你正在重建数据中心里的一台受损的服务器——这时，你看到备份数据里的服务器配置文件会万分感激。

被选中的操作系统文件

和服务器配置文件一样，备份那些对服务器来说必需的外部配置文件也很重要。在 Unix 服务器上，这些文件包括 cron 任务、用户和组的配置、管理脚本及 sudo 规则。

这些建议在许多场景里会被很快地翻译为“备份所有的东西”。如果你有大量的数据，这么做的代价会变得昂贵，在关于怎么做备份这个问题上，你必须有个灵活的方法。特别地，你可能会希望把不同的数据备份到不同的备份记录里。举例来说，你可以把数据、二进制日志、操作系统及其配置文件都单独备份。

增量备份

对于大量的数据，一种常用的办法就是定期做增量备份。下面是有关这种备份方法的进一步说明：

- 备份二进制日志。这是最简单、使用最广泛、最适合的增量备份的途径。
- 不要备份没有变动的数据表。有一些存储引擎，比如 MyISAM，对每张表的最后一次更新时间都有记录。你可以通过查看磁盘上的文件日期，或者运行 SHOW TABLESTATUS 来得到这个更新时间。如果你使用的是 InnoDB，那么就创建一个触发器，把这些更新时间都写入一个很小的“最近更新时间”表里。这个方法只能用在更新频率不高的那些表上，这样开销会很低。一个自定义的备份脚本能轻松地判定哪些表的数据有过变动。

如果你有“查找”表，其中包含的数据都是诸如用不同语言表示的所有月份的名称，或者是州名或地区名的缩写。这样的表最好是放在一个单独的数据库里，这样你就用不着总是去备份它们。

- 不要备份没发生改变的行。如果一个表是只能 INSERT 的，比如记录每个页面的点击数的表，你就往里面增加一个 TIMESTAMP 字段，每一次备份自上次备份以来新增的那些行就可以了。你也可以使用 Merge 数据引擎，把老的数据放在静态表里。
- 有些数据不要去备份。有时候这个意义重大，举例来说，你有一个数据仓库，它是基于其他数据构建的，因此从技术方面来讲这也属于冗余。在备份时，你可以只备份这部分原始数据，用不着把数据仓库也备份起来。即使在还原数据后需要花时间去重建数据仓库，这个方法也是可行的，因为在备份时候省下的时间和存储空间，会多于全部备份时所需的时间和存储空间。你也可以选择不备份某些临时数据，比如存储网站 session 数据的表。
- 只备份二进制日志里的变化部分。你可以使用 rdiff 命令只获取自日志上次备份后变化的记录（定期的完全备份还是需要的）。还有一个我们用到的工具是 backup，它把 rdiff 和 rsync 合并为一个完整的备份方案。或者你可以在每次备份后都用 FLUSH LOGS 来重新创建一个日志，这样就无需记录日志的变化了。
- 只备份数据文件里的变化部分。这个就像刚才处理二进制日志一样。在 Unix 上常用的工具是 rdiff 和 rdiffbackup。这种方法对于数据量很大但变化不大的数据库非常有用。假如你有 1TB 的数据，而其中只有 50GB 每天会有变动的，那么，现在只要备份其中每天发生变动的那部分就可以了，再偶尔做一次完全备份。这样做的好处是你可以把变动部分转换为一系列磁盘读/写的方式，写入一个完全备份的记录，

比使用二进制日志要快很多。然而，做一次二进制差异备份可能要比完全备份慢一些。

增量备份的缺点是数据还原时的复杂性提高了。如果你正顶着重重压力做系统恢复，相比于还原一个接一个的增量备份，还原一个备份记录会使你轻松许多。如果能做完全备份，那么为简单起见，我们还是建议您做完全备份。

无论如何，你偶尔还是要做一次完全备份的——建议您至少一星期一次。我们不希望使用一个年度的所有增量备份来恢复系统，甚至连一个星期都包含了很大的工作量和风险。

11.2.5 存储引擎和一致性

Storage Engines and Consistency

MySQL 能够选择使用各种存储引擎，这让备份操作变得更加复杂。事情的关键是如何在不同的存储引擎上获得一个与服务器一致的备份记录。

这里的一致性要从两个方面来考虑：**数据一致性和文件一致性**。

数据一致性

484

在备份的时候，你必须保证数据在即时点上是一致的。举例来说，在一个电子商务的数据库里，你需要确保发票和支付记录是相互一致的。当恢复一个支付记录时，若没有其对应的发票，或者反过来，这两个情形都会引起麻烦。

如果你正在做在线备份（在一台正在运行的服务器上），你需要确保获得的是与全部相关表都一致的备份。这就意味着你不能是一次一个表地做锁定-备份的操作。如果你没有使用事务性数据存储引擎，你就别无选择，只有使用 LOCK TABLES 命令锁定所有的表，然后把它们备份到一起，直到所有相关的表都备份完了，你才可以释放锁。

InnoDB 的 MVCC 功能可以帮上点忙。你可以先开启一个事务，然后导出一组相关表，最后提交事务。（这里你使用一个事务来获得一致的备份，就不用 LOCKTABLES 命令了，因为事务会被隐式提交。——细节方面请查看 MySQL 帮助手册）只要你使用的是 REPEATABLE_READ 事务隔离水平，这样的做法就能给你一个完美的一致性。数据的即时点快照也不会阻碍服务器备份完成后的进一步的工作。

然而，这个方法也不能让你避开设计糟糕的应用逻辑。假如你的电子商务站点插入了一条支付记录，然后提交了事务，接着又另外开启一个事务，插入了一条发票记录。你的备份过程可能正好是在这两个事务之间开始的，于是，备份里只有这条支付记录而没有发票记录了。所以说你必须仔细地设计你的事务，把相关的操作都组织起来放入一个事务里。

你也可以使用 mysqldump 来获得逻辑上一致的 InnoDB 表的备份，备份时给 mysqldump 加上 --single-transaction 选项就可以获得上面讲到过的效果了。但是，这会产生一个很长的事务，因此，在某些时候甚至会产生大到难以接受的系统开销。

有一些工具支持“备份集”，比如 ZRM（下文中会讲到）、Maatkit 的 mk-parallel-dump，它们可以帮你轻松备份相关的表集合。

文件一致性

内部文件相互一致也非常重要，比如备份记录反映不出一个文件是否正处于 UPDATE 过程的中间。备份的文件之间相互也要保持一致性，否则当你恢复这些文件时会遇到让你感觉糟糕的意外（这些文件可能都被损坏了）。如果你在不同的时间里复制了这些相关连的文件，那它们就无法一致了。MyISAM 的 .MYD 和 .MYI 就是一个例子。

485 如果使用的是一个非事务类型的数据引擎，例如 MyISAM，你唯一可选择的就是锁定表然后刷新缓冲区。这就意味着你或者使用 LOCK TABLES 和 FLUSH TABLES 的组合，把内存里缓存着的更新写到磁盘上，或者使用 FLUSH TABLES WITH READ LOCK。一旦缓存区刷新后，你就能安全地取得 MyISAM 文件的一份裸副本。

如果使用的是 InnoDB，想要确定磁盘上的文件是否一致就有点困难了。即使你做了 FLUSH TABLES WITH READ LOCK 操作，InnoDB 仍然会在后台运行着：它的插入缓冲区、日志和写线程继续把数据更新写到日志和表空间的文件里。这些线程都被设计成异步的——这样的设计是为了 InnoDB 能达到高并发性——所以，它们独立于 LOCK TABLES。因而，你不仅要确保每个文件的内部一致性，还要在同一时刻复制日志文件和表空间文件。如果做备份时，刚好有个线程在修改文件，或者说备份日志文件的时间点跟备份表空间文件时不一样，那在还原这些文件后得到的还是一个数据损坏了的系统。不过，你可以通过两条途径来避免这个问题：

- 一直等待，直到 InnoDB 里那些清除线程和插入缓冲区合并线程退出。你可以查看 SHOW INNODB STATUS 的输出结果，当看到缓冲区都干净了，未决的写操作也都完成了，你就可以开始复制文件了。然而，这个方法实行起来会花费很长一段时间，而且由于 InnoDB 后台线程的存在，操作时会包含很多猜测，从而显得不怎么可靠。有鉴于此，我们不推荐这个方法。
- 利用系统功能，比如用 LVM 对数据和日志文件做一个一致性快照。这个快照必须使两者成对进行，如果分开来做快照那就不怎么好了。我们将在本章的后面详细讨论 LVM 快照。

一旦把文件复制到了别处，你就可以释放所有的锁，让 MySQL 继续正常运行。

11.2.6 复制

Replication

在普通大众的观念里，用 MySQL 复制做备份的想法是荒谬的。将复制用作完全备份的一部分确实有其特有的优点，但是不能把它当做备份的一切——通常就是这么说的。

从一个从服务器上取得备份的最大好处是它不会妨碍主服务器的运行，也不会给它增加额外的负载。这就是建立一个从服务器的好理由，哪怕你不是拿它来做负载平衡或高可用性。如果不舍得花钱，你也可以把这台备用用途的从服务器派上别的用场，比如做数据报告——只要这些任务不写入数据，也不更改你正在备份的数据。这台从服务器无需专门用作备份，它只要在下次备份到来前，与主服务器的数据同步更新就可以了。

486 当你在一台从服务器上做备份时，要保存所有关于复制过程的信息，比如这台从服务器在主服务器的地位。这个对于以下几种情况很有用：新从服务器的克隆、把二进制日志导入主服务器做即时点还原、把一个从服务器升级为主服务器，以及其他更多的情形。在你停止从服务器之前，要确保没有被打开的临时表，因为它们会阻碍你重启复制功能。关于这方面的更多内容你可以查看第 394 页的“丢失的临时表”。

如果在从服务器里有一台特意设置了延迟复制，那它在一些灾难场景后做还原数据时会很有用。假设你是延迟一小时做复制的，当一条不必要的语句在主服务器上执行之后，你还有一小时的时间去注意到它，并能在从服

务器重复这个操作之前，停止其复制行为。然后，你就把这台从服务器升级为主服务器，跳过刚 www.zzbaike.com 语句，把一些数量较小的相关日志事件重放一遍就好了。这比接下来我们要讨论的即时点还原技术更简单。本教程由站长百科收集整理来自 Maatkit 的 `mk-slave-delay` 脚本可以用在此处。



警告：从服务器的数据可能会跟主服务器的不一样。许多人都假定从服务器上的数据就是主服务器上数据的完全一致的副本。但是，根据我们经验，数据错配在从服务器上很常见，MySQL 也没办法检测到这个问题。备份了错误的或损坏的数据，结果都是得到一个没用的备份记录。更多关于如何确保从服务器数据跟主服务器一致的内容请查看第 380 页的“确定从服务器与主服务器数据一致”。第 8 章也提到了一些避免主-从服务器之间不一致的建议。

保存一份数据复制副本，可以帮你远离例如主机磁盘崩溃等问题，但这也不能保证它一直有效，毕竟，复制不是备份。

11.3 管理和备份二进制日志

Managing and Backing Up Binary Logs

服务器上的二进制日志是你备份的最重要的内容之一。它们对于即时点还原是必不可少的，因为在尺寸上它比数据小，易于经常备份。如果你有某个时间点上的数据备份，还有自那个时间点以来的所有二进制日志，那么你就可以通过重放二进制日志，把自上次完全备份以来所有的数据变动都“前滚”出来。

MySQL 也把二进制日志用作复制。这意味着你的备份/还原策略要经常与你的复制配置相配合。

二进制日志很“特殊”。如果你丢失了数据，又不甘心它们就这么没了，为了让此类事件发生概率降到最小，你可以把它们放在一个单独的卷里。这样即使你想用 LVM 对它做一个快照，也是没问题的。如果想要更多的安全保障，你可以把它们放在 SAN 里，或者通过 DRBD 复制到其他设备上。关于这方面的更多内容，你可以阅读第 9 章。

经常备份二进制日志是个很好的主意。如果你无法承受超过 30 分钟的数据被丢失，那你就该至少每 30 分钟备份一次日志。你也可以使用带 `--log_slave_updates` 选项的只读复制从服务器，以取得额外一层的安全。这样，虽然日志的位置就与主服务器的不匹配，但是，要找到还原的正确位置也不是太难。

以下是我们推荐的用于二进制日志的服务器配置：

```
log_bin                = mysql-bin
sync_binlog            = 1
innodb_support_xa     = 1    # MySQL 5.0 and newer only
innodb_safe_binlog     # MySQL 4.1 only, roughly equivalent to innodb_support_xa
```

此外，还有别的配置项可用，比如限制日志大小的选项。这方面的更多内容你可以查看 MySQL 的使用手册。

11.3.1 二进制日志的格式

The Binary Log Format

二进制日志里包含了一系列的顺序事件。每个事件都有固定大小的头部信息，其中包含着多种信息，比如当前的时间戳、默认的数据库等。你可以使用 `mysqlbinlog` 工具来查看日志的内容，它能打印出一些头部信息，以下就是一个输出的例子：

```
1 # at 277
2 #071030 10:47:21 server id 3 end_log_pos 369 Query thread_id=13 exec_time=0
  error_code=0
3 SET TIMESTAMP=1193755641/*!*/;
4 insert into test(a) values(2)/*!*/;
```

第 1 行包含了当前记录在日志文件里的偏移字节数（这里是 277）。

第 2 行包含了如下内容：

- 事件发生的日期和时间，MySQL 也会用这个来生成 SET TIMESTAMP 语句。
- 源服务器的服务器 ID，这个对于防止循环复制及其他问题是必需的。
- end_log_pos 是描述下一个事件的偏移字节数。这个值在多语句事务环境下的多数事件记录里都是累加的。在主服务器上发生这些事务时，MySQL 会把这些事件都复制到一个缓存里，但是并不知道下一个日志事件的位置。
- 事件的类型。在本例中，它是一个查询。其实还存在许多不同的类型。
- 在服务器上执行该事件的线程的 ID，跟执行 CONNECTION_ID() 函数一样，这在审核时很重要。
- exec_time，这个项目的真实含义即使对于一些 MySQL 开发人员来说也不太明白。通常来讲，它记录了这个语句执行时所花费的时间，但是，在某些条件下，它又会是些奇怪的值。举例来说，当一个从服务器的 I/O 线程远远落后于主服务器时，这个项目在转发日志上会是一个很大的数值，不管这些语句在主服务器上执行得有多快。所以，不要去依赖这个项目的值。
- 该事件在源服务器上产生的错误代码。如果当从服务器上重放该事件时产生了跟它不一样的错误代码时，作为一项安全预防措施，复制过程就会失败。

其他行里还包含了 SQL 要重放该事件所需的其他数据。还有用户自定义变量、任何其他特殊的设置，例如语句执行时受影响的时间戳也会显示在这里。



提示：如果你正在使用 MySQL 5.1 里的以行为基础的日志，这些事件就不是 SQL 的表现形式了，而是一种无人能读的表更新语句的“映像”。

11.3.2 安全地清除旧日志

Handling Old Binary Logs Safely

现在，你需要制定一个日志作废策略，以免 MySQL 用二进制日志把磁盘都填满了。日志大小增速与 MySQL 的工作量和日志格式有着相互联系（MySQL 5.1 里以行为基础的日志，会有更多的日志条目）。我们建议您的日志只要是有可能被用到，就应该保存下来。这些保存下来的日志可以帮你搭建复制从服务器、分析服务器的工作负荷、审核，以及做上次完全备份以来的即时点还原。当你要在决定该保留多长的日志时，请考虑一下以上那几个需求。

一个常用的设置方法是使用 expire_logs_days 变量来告诉 MySQL 要过多久才可以清除这些日志。这个参数只能用于 MySQL 4.1 以后的版本。对于以前的版本，你就不得不手工删除。然而，你可能也看到过那个建议：使用 cron 条目去删除旧日志，就像下面这样：

```
0 0 * * * /usr/bin/find /var/log/mysql -mtime +N -name "mysql-bin.[0-9]*" | xargs rm
```

对于早于 4.1 版本的 MySQL 来说，这是清除日志的唯一手段了，千万不要在 4.1 及更新的版本里这么做。使用

rm 删除日志文件会使 mysql_bin.index 状态文件跟磁盘上的文件不同步，像 SHOW MASTER LOGS 命令执行时就会毫无反应。即使再手工去修改 mysql_bin.index 文件也无济于事。只能像下面这样使用 cron 命令去做：

```
0 0 * * * /usr/bin/mysql -e "PURGE MASTER LOGS BEFORE CURRENT_DATE - INTERVAL N DAY"
```

expire_logs_days 的设置值在服务器重启，或者 MySQL 轮转日志之后才能生效，所以，如果二进制日志还没填满，也没轮转，服务器仍然不会去清除那些旧的日志条目，它是根据条目的更新时间而不是内容做清除操作的。

11.4 数据备份

11.4 数据备份

在很多主题里，要做个备份总有很多途径，或好或坏——显而易见的方法有时不见得是个好办法。这样的方法往往通过夸大网络、磁盘和 CPU 的承载能力，把备份的速度说得尽可能地快。其实这些因素之间是有一个平衡点的，你要做几次试验才能找到这个“最近听音位置”。

我们很难给出一个特定的建议，所以，在此展示几种很常用的技术。

11.4.1 做一个逻辑备份

11.4.1 做一个逻辑备份

关于逻辑备份，首先要认识到的是它们创建方式是不一样的。在实际应用中，有两种逻辑备份：SQL 导出和限定符文件。

SQL 导出

SQL 导出是很多人所熟悉的，因为这是 mysqldump 默认创建的。举例来说，用默认设置导出一个小的数据表，其输出结果如下所示（已作删减）：

```
$ mysqldump test t1
-- [Version and host comments]

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [More version-specific comments to save options for restore]

--
-- Table structure for table `t1`
--

DROP TABLE IF EXISTS `t1`;
CREATE TABLE `t1` (
  `a` int(11) NOT NULL,
  PRIMARY KEY (`a`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `t1`
--

LOCK TABLES `t1` WRITE;
/*!40000 ALTER TABLE `t1` DISABLE KEYS */;
INSERT INTO `t1` VALUES (1);
```

```
/*!40000 ALTER TABLE `t1` ENABLE KEYS */;  
UNLOCK TABLES;  
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;  
  
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;  
-- [More option restoration]
```

这个导出的文件里包含了表的结构和数据，所有这些都是用可用的 SQL 语句的方式表示的。文件开始的注释部分用来设置 MySQL 各项参数，它们的作用或者是为了还原工作能进行得更有效率，或者是为了兼容性和正确性。接下来，你会看到表的结构，然后就是表内的数据了。最后的那段脚本是将文件开始时修改的参数再修改回来。

导出文件可以直接用来做还原操作，这样显得很方便，但是仅凭 mysqldump 的默认选项还是无法做大规模备份的（在下文里我们会深入讨论 mysqldump 的各个选项）。

mysqldump 不是做 SQL 逻辑备份的唯一工具，例如你也可以使用 phpMyAdmin 来创建备份。我们真正想指出的是使用任何一个特定的工具都不会有太多的问题，但是这里我们把做单一备份记录时的所有缺点都放在了显眼位置，以下就是主要的问题：

样式与数据存放在一起

如果想从单一文件恢复，这是很方便的，但是，如果只恢复一个表，或者只恢复数据，那做起来就麻烦了。导出两次——一次是数据，另一次是样式——可以减轻这个痛苦，但仍然会有下面这个问题。

巨大的 SQL 语句

对于服务器来说，解析、执行所有的 SQL 语句包含了巨大的工作量。通过这样的途径加载数据会相对显得缓慢。

单一的巨大文件

许多文本编辑器无法编辑大文件，在文件的行很长时也无法正常工作。虽然有时你可以使用命令行流编辑器——例如 sed 或 grep——取出你所需要那些数据，但是，与此相比，保持文件小型化才是更好的做法。

逻辑备份成本昂贵

同样也是把数据从 MySQL 里取出来，有很多途径比通过 client/server 协议发送结果集更有效率。

这些限制条件意味着当表变得很大时，SQL 导出的方式就无用武之地了。这里就有了另外一个选择：把数据导出到带定界符号的文件里。

使用定界符文件备份

你可以使用 SELECT INTO OUTFILE SQL 命令来创建一个定界符文件形式的逻辑备份。（你也可以使用带-tab 选项的 mysqldump 来生成，它会帮你调用这个 SQL 命令。）定界符文件里的数据都是用 ASCII 表示的裸数据，没有 SQL 语句，没有注释，也没有列名称。以下就是导出 CSV 格式（Comma-separated values，这种格式是用于表列语言的很好的混合语言）文件的例子：

```
mysql> SELECT * INTO OUTFILE '/tmp/t1.txt'  
-> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY ''  
-> LINES TERMINATED BY '\n'  
-> FROM test.t1;
```

这个结果文件比 SQL 导出文件更加紧凑，更易于用命令行工具生成，但是它最大的优点还是备份和还原的速度。你仍然可以使用 LOAD DATA INFILE 命令将数据加载到表里去，就像导出时一样：

```
mysql> LOAD DATA INFILE '/tmp/t1.txt'
-> INTO TABLE test.t1
-> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
-> LINES TERMINATED BY '\n';
```

这里我们要做一个非正式的测试，用来验证 SQL 文件和界定符文件在备份和还原时各自的速度表现。我们准备了一些合适的数用于此次测试。用来导出的表定义如下：

```
CREATE TABLE load_test (
  col1 date NOT NULL,
  col2 int NOT NULL,
  col3 smallint unsigned NOT NULL,
  col4 mediumint NOT NULL,
  col5 mediumint NOT NULL,
  col6 mediumint NOT NULL,
  col7 decimal(3,1) default NULL,
  col8 varchar(10) NOT NULL default '',
  col9 int NOT NULL,
  PRIMARY KEY (col1,col2)
) ENGINE=InnoDB;
```

这个表里有 1 500 万行数据，在磁盘大致上占用 700MB 空间。表 11-1 比较了两种方式在备份和还原时的性能，你会发现在还原时两者的速度差异非常巨大。

表 11-1：SQL 文件和界定符文件在备份、还原时的耗费时间

| Method | Dump size | Dump time | Restore time |
|----------------|-----------|-----------|--------------|
| SQL dump | 727 MB | 102 sec | 600 sec |
| Delimited dump | 669 MB | 86 sec | 301 sec |

SELECT INTO OUTFILE 方法也存在着一些局限性：

- 你只能在 MySQL 运行的机器上把数据都备份到一个文件里。（你可以编写一个程序，让它读取一次 SELECT 的结果，然后写到磁盘里，这个滚动 SELECT INTO OUTFILE 的方法，我们看到有些人正在这么使用。）
- MySQL 必须有权限访问要生成文件的那个目录，因为 MySQL 服务器——不是用户运行的 SQL 命令——它要把数据写入那个文件。
- 出于安全方面的原因，你不能覆盖已存在的文件，不管这个文件的权限是怎么分配的。
- 你无法直接导出一个压缩文件。

并行导出与还原

在多 CPU 的系统上，用并行方式执行备份和还原会更快。这里“并行”的意思是对多个表同时做导出或还原，不是说多个程序在同一个表上同时做各自的操作。若两个程序同时将数据加载到一个表单里，这效果一般来说都不会好。

你无需使用很特异的工具来做并行的备份/还原，只需要手动运行备份工具的多个实例就可以了。市面上确实有一些工具和脚本是特地为这个用途而设计的，例如 Maatkit 的 mk-parallel-dump 和 mysqldump (<http://www.fr3nd.net/projects/mysqldump/>)。在编写本内容的时候，这些工具相对来说还比较新。基准测试显示 mk-parallel-dump 比单纯使用 mysqldump 备份要快好几倍。

在 MySQL 5.1 里,mysqlimport 支持多线程同时导入。你也能在早期的 MySQL 上使用 5.1 版

如果使用的并行度过高,并行导出与还原实际上可能会耗费更长的时间,而且还会引起更多的数据碎片,从而影响到系统的性能。

TopSage.com

11.4.2 文件系统快照

Filesystem Snapshots

文件系统快照是做在线备份的极佳方法。有快照功能的文件系统能够在瞬间将它的内容做一个一致的镜像,这个镜像你可以用来做备份。有快照功能的文件系统和设备包括 FreeBSD 的文件系统、ZFS 文件系统、GNU/Linux 的逻辑管理器 (Logical Volume Manager, LVM), 以及许多 SAN 系统和文件存储解决方案 (例如 NetApp 存储设备)。

在这里,不要把快照跟备份弄混了。做一个快照只是减少锁定时间的一个简单手段,在释放锁之后,你必须把这些文件都复制到备份里去。事实上,你可以在 InnoDB 上选择做无需锁定的快照。在你所希望的最少锁定甚至不用锁定的前提下,我们为你展示两种使用 LVM 在全 InnoDB 数据库上做备份的方法:



提示: Lenz Grimmer 的 mylvmbakup 是一个现成的 Perl 脚本,它通过 LVM 来创建 MySQL 备份。更多内容请查看本章后面第 511 页的“备份工具”。

LVM 快照是怎么工作的

LVM 使用了写时复制 (Copy-on-write) 技术来创建快照——也就是即时获得全部卷的一个逻辑副本。这有点像数据库里的 MVCC,除了它只保留一份旧版本的数据外。

你应该已经注意到我们没说是一个物理副本。一个逻辑副本看上去跟已做了快照的卷有一模一样的数据,但是在开始之初,它并不包含任何数据。LVM 只是简单地标注了一下你做快照的时间,而不是把数据都复制到快照里。当你需要从快照里读取数据时,它才会到数据来源的卷里把它们都读出来。所以,最初的副本本质上是一个即时的操作,无论你要做快照的卷有多大。

493 当卷里的数据在快照完成之后发生变动时,LVM 就会在变动生效之前,把受影响的数据块复制到一个为快照保留的存储区域里,LVM 不会保留多个“旧版本”的数据,因此,同一个数据块上的多次变动不会让快照再去复制数据了。换句话说,每一个数据库只有在第一次写入时才会触发写时复制,从而把数据复制到保留区域里。

现在,当你要读取快照里这些已被复制过的数据块时,LVM 就会从保留存储区域里读取数据,而不是最初来源的卷里。这样可以使你能在快照里继续读取那份数据,而不会妨碍来源卷里的任何操作。图 11-1 描述了这个安排方式。

快照会在/dev 目录下新建一个逻辑设备,你可以把这个设备挂载 (Mount) 到任何你要挂载的地方。

通过这项技术,在理论上你也可以对非常巨大的卷做快照,而耗费的物理存储空间依然很小。但是,你要预留出足够的空间来容纳当来源卷里的数据更改时要复制过来的数据块。如果保留的写时复制空间不够用,快照会耗尽空间,从而变成不可用,就像拔掉了一个外部驱动器:从中读数据的备份任务都会因 I/O 错误而停止。

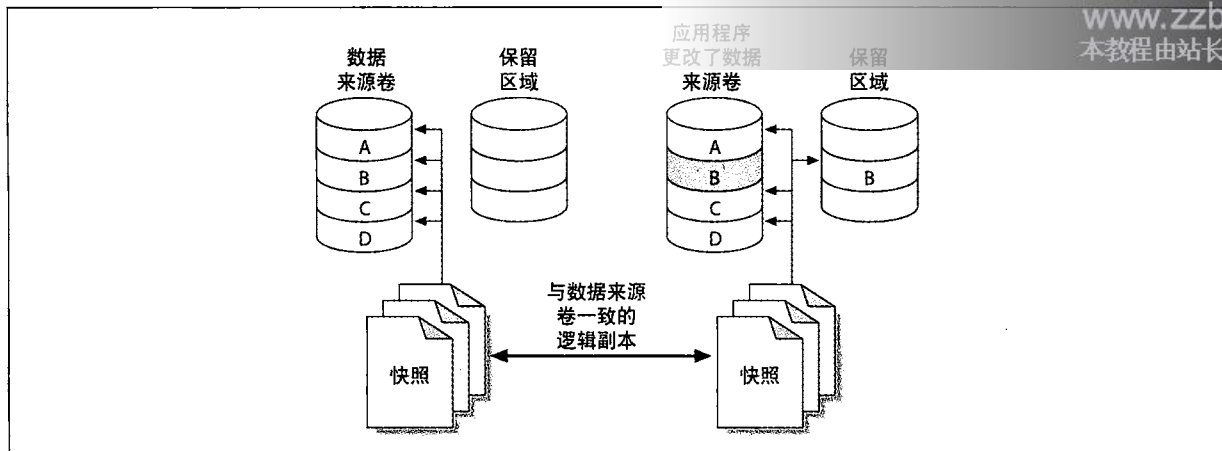


图 11-1：写时复制技术是如何减少快照所需的大小的

先决条件和配置方法

在你掌握磁盘上的分卷信息前来谈创建一个快照几乎是没有什么意义的。你需要先确定你的系统是否已经做好了配置，有了这些配置你才可以在某个即时点上做备份时，得到一个包含所有文件的一致性副本。所以，首先要确认你的系统是否已经符合了以下条件：

- 所有 InnoDB 文件（InnoDB 表空间文件和 InnoDB 事务日志）必须在一个逻辑卷（分区）里。需要即时点上的数据具有一致性，LVM 无法在同一个时间内为一个以上的卷做符合一致性的快照。（这是 LVM 的局限性，另外一些系统没这样的问题。）
- 如果你也需要备份表定义，那么，MySQL 数据目录必须在同一个逻辑卷里。如果你使用另外一个方法来备份表定义，例如把只含有样式的备份记录放入版本控制系统里，就无须担心这样的问题。
- 你在分卷组里必须有足够的空闲空间来创建快照。空间的大小依赖于工作负荷。当你搭建系统时，留一些未分配的空间，这些可以在后来被用作快照存储。

LVM 有一个分卷组（Volume group）的概念，它包含一个或多个逻辑分卷。你可以在你的系统里看到那些分卷组，像下面这样：

```
# vgs
VG      #PV #LV #SN Attr   VSize  VFree
vg      1   4   0 wz--n- 534.18G 249.18G
```

这个输出结果显示一个分卷组有 4 个逻辑卷分布在一个物理卷上，还留有 250GB 的空闲空间。如果你有需要，vgdisplay 命令能给你更多的细节。现在，就让我们看一下系统里的逻辑卷：

```
# lvs
LV      VG      Attr   LSize   Origin Snap%  Move Log Copy%
home    vg      -wi-ao 40.00G
mysql   vg      -wi-ao 225.00G
tmp     vg      -wi-ao 10.00G
var     vg      -wi-ao 10.00G
```

这个输出结果显示 mysql 卷上有 225GB 空间。设备是/dev/vg/mysql，这只是一个名称，尽管看上去像个文件系统里的路径。更使人混淆的是，从这个名称的设备到真实的设备节点上还有一个叫/dev/mapper/vg-mysql 的

符号化的链接。你可以使用 `ls` 和 `mount` 命令看到这个链接：

```
# ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
/dev/mapper/vg-mysql on /var/lib/mysql type reiserfs (rw,noatime,notail)
```

掌握了这些信息以后，你就可以创建文件系统快照了。

创建、挂接和移除 LVM 快照

使用一个单独的命令，你就可以创建快照了。你只需要决定哪里存放快照、给写时复制分配多少存储空间。在分配时要毫不犹豫地比预想的再多分配一些空间。LVM 使用存储空间时不像你指定的那样，它会为未来的使用事先保留起一部分，因此，多保留一些空间总是无害的，除非你需要把这些空间用于同时做的另一些快照。

这里我们就练习一下创建快照。我们给写时复制分配 16GB 的空间，并命名为 `backup_mysql`：

```
# lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
Logical volume "backup_mysql" created
```



提示：我们把卷名叫做 `backup_mysql`，而不是 `mysql_backup`，这样就可以避免 `tab` 自动完成时的二义性，从而也避免了不小心删除 `mysql` 分组卷的可能。类似这样的细节可以帮你避免大灾难。至少，本书的一个作者就曾因为一个草率的 `tab` 自动完成，把 LVM 快照毁掉了。

现在让我们看一下最近创建的分卷状态：

```
# lvs
LV          VG      Attr   LSize   Origin Snap%   Move Log Copy%
backup_mysql vg      swi-a- 16.00G   mysql    0.01
home        vg      -wi-ao 40.00G
mysql       vg      owi-ao 225.00G
tmp         vg      -wi-ao 10.00G
var         vg      -wi-ao 10.00G
```

我们注意到快照的属性已经跟当初的设备不一样了，这些输出信息里还包含了一些额外的信息：那块用于写时复制的 16GB 空间开始时的使用率和当前使用率。在备份的时候监控这些信息的变化是个很好的主意。这样你就能看到空间是否快要满了、是否快要宕机了。你也可以通过监控系统，例如 Nagios 来监控设备的状态：

```
# watch 'lvs | grep backup'
```

就像你刚才在 `mount` 输出信息里看到的那样，`mysql` 卷里包含着一个 ReiserFS 文件系统。这意味着快照卷也是这个文件系统，你可以像其他文件系统那样挂接并使用它：

```
# mkdir /tmp/backup
# mount /dev/mapper/vg-backup_mysql /tmp/backup
# ls -l /tmp/backup/mysql
total 5336
-rw-r----- 1 mysql mysql    0 Nov 17 2006 columns_priv.MYD
-rw-r----- 1 mysql mysql 1024 Mar 24 2007 columns_priv.MYI
-rw-r----- 1 mysql mysql 8820 Mar 24 2007 columns_priv.frm
-rw-r----- 1 mysql mysql 10512 Jul 12 10:26 db.MYD
-rw-r----- 1 mysql mysql 4096 Jul 12 10:29 db.MYI
-rw-r----- 1 mysql mysql 9494 Mar 24 2007 db.frm
... (下略) ...
```

这仅仅是一个练习，所以，最后我们使用 `lvremove` 命令把快照从文件系统上卸下来并删除。

```
# umount /tmp/backup
# rmdir /tmp/backup
# lvremove --force /dev/vg/backup_mysql
Logical volume "backup_mysql" successfully removed
```

496

用于在线备份的 LVM 快照

你已经看到了怎样去创建、挂接和删除快照，现在你能用它们来做备份了。首先，让我们看一下如何在不停止 MySQL 服务器的情况下备份一个 InnoDB 数据库，接着连接到 MySQL 服务器，通过一个全局的读锁把表的缓冲区都刷新到磁盘上，然后得到一个二进制日志的备份记录：

```
mysql> FLUSH TABLES WITH READ LOCK; SHOW MASTER STATUS;
```

记录 `SHOW MASTER STATUS` 的输出信息，确认你还连接着 MySQL 服务器没有释放掉锁。接下来就是做 LVM 快照，做完后立即使用 `UNLOCK TABLES` 或关闭数据库连接把读锁释放掉。最后，把快照挂接上去，复制里面的文件到备份位置上。如果你是用脚本来做的，那么锁定的时间能减少到几秒钟。

这个方法的主要问题是获取读锁可能会花一些时间，特别是当一个很长的查询正在执行时。当数据库连接在等待一个全局读锁时，所有的查询都会被堵塞，也不可能预计这个过程会有多久。

用 LVM 快照做 InnoDB 的无锁备份

无锁备份跟上面的方法相比有点不同。这个差别在于你用不着做一次 `FLUSH TABLES WITH READ LOCK`，这也意味着在你磁盘上的 MyISAM 文件无法保证具有一致性，但是对于 InnoDB 来说，这可能不是问题。你仍然会有一些 MyISAM 表在 `mysql` 的系统数据库里，但是，如果你的备份工作比较典型，这些表在你做快照的时候，可能也不会发生改变。

如果认为那些 `mysql` 系统表也可能会改变，你可以先锁定它们，然后刷新它们的缓冲区。之后，就不要在这些表上做耗时很长的查询了，这样备份起来会快很多：

```
mysql> LOCK TABLES mysql.user READ, mysql.db READ, ...;
mysql> FLUSH TABLES mysql.user, mysql.db, ...;
```

如果你没有得到一个全局读锁，使用 `SHOW MASTER STATUS` 也不能得到什么有用的信息。然而，当在快照上启动 MySQL 时（为了检验备份的完整性），你会在日志文件里看到类似下面这样的信息：

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Progress in percents: 3 4 5 6 ...[omitted]... 97 98 99
InnoDB: Apply batch completed
InnoDB: Last MySQL binlog file position 0 3304937, file name /var/log/mysql/mysql-
bin.000001
070928 14:08:42 InnoDB: Started; log sequence number 0 40817239
```

文件系统快照与 InnoDB

即使在你锁定了所有的表之后，InnoDB 的后台线程还在继续工作，也就是说，当你正在做快照的时候，文件可能还正在被写入。因为 InnoDB 没有执行它的关闭程序，所以，快照里的 InnoDB 文件看上去就像所在的服务器遭遇了意外断电。

497

但这不算个问题，因为 InnoDB 是一个 ACID 系统。在任何时刻（例如你做快照的那刻），事务要么在 InnoDB 的数据文件里，要么在日志文件里。当你恢复了快照，重新启动 MySQL 之后，InnoDB 会运行一个还原过程，就像服务器意外断电之后那样。它会在事务日志里查找到还未更新到数据文件的事务，然后将它们写到数据文件里，这样就可以保证不会丢失任何事务。这就是为什么必须强制在 InnoDB 里的数据和日志文件一起做快照。

这也是为什么在备份之后你要测试备份效果：启动一个 MySQL 的实例，加入那个新做的备份，运行 InnoDB 的还原功能，让它核对所有的表。使用这个方法，你就不会备份损坏的数据，哪怕你事先根本不知道。（文件会有种种原因而损坏。）另外一个好处是将来从备份记录恢复数据的过程会更快，因为你已经运行过还原过程了。

在将快照复制到备份里之前，你可以选择运行这个过程，但是，这也会给系统增加一点开销，这取决于你是否事先做了计划。（更多的内容会在后面展开。）

InnoDB 记录了 MySQL 二进制日志的位置，该位置对应于它被恢复的那个点。这个日志位置可以被用来做即时点还原。

这种使用快照的无锁备份方法在 MySQL 5.0 及更早的版本里有一点扭曲。这些版本的 MySQL 使用 XA 来协调 InnoDB 和二进制日志之间的事务。如果你把备份恢复到不同 `server_id` 的服务器上（这个 `server_id` 是属于备份时的那台服务器的），这台服务器会发现已准备好的事务的 ID 跟它自己的不匹配。对于这种情况，服务器会感到迷惑，当做还原时，那些事务可能会停滞在 `PREPARED` 状态上。这虽然很少发生，但一直存在着这种可能。这就是为什么在你认为备份成功前，最好检验一下备份记录：它可能没法被还原！

如果你正在一个从服务器上做快照，InnoDB 还原会打印出几行信息，类似于下面这个样子：

```
InnoDB: In a MySQL replication slave the last master binlog file
InnoDB: position 0 115, file name mysql-bin.001717
```

在一些版本的 MySQL 里，这几行输出信息告诉你主服务器的二进制日志的定位（跟从服务器上的日志定位），在该点上 InnoDB 做了还原，这个对于从服务器上做备份或从服务器之间的克隆很有用处。然而，在 MySQL 5.0 及更早的版本里，这些数值是靠不住的。

498 规划 LVM 备份

LVM 快照备份对于系统性能并非没有影响，服务器往来源卷里写入的数据越多，备份它们的开销就越大。当服务器以随机次序更新几个完全不同的数据块时，磁盘的磁头必须来回寻找写时复制的空间，并把旧数据写入这个区域。从快照里读数据也需要开销，因为对于大多数数据来说，LVM 还是要真正地到来源卷里去读取。从写时复制里读取只是在需要的时候才会发生；因此，一个从快照里按逻辑顺序读数据的过程，实际上会引起磁盘磁头的来回移动。

你要为此类事情的发生提早做规划。要考虑的事情还有：如果你使用了大量的写时复制空间，那么来源卷和快照的性能会比通常的读写还要糟糕。这不仅能减缓 MySQL 服务器的运行速度，而且还会减缓备份时的文件复制进程。

另一个重要的事情是要为快照分配足够多的空间。我们使用的是下面这个方法：

- 要记住 LVM 只要把更新的数据块复制到快照里一次就行了。当 MySQL 在来源卷里写入数据时，会把原来的数据都复制到快照里去，然后在它的例外表里标注上这个被复制的数据块。因此，对快照的任何写入操作都不会触发“复制到快照”这个操作了。
- 如果你只是用 InnoDB，要考虑到 InnoDB 是怎么写数据的。因为它把所有数据都写两遍，至少有一半的 InnoDB 写 I/O 是写到了双写缓冲区（Doublewrite Buffer）、日志文件和磁盘上的相关的小区域里。这些写入都会一次次地重用磁盘上的同一个数据块空间。因此，它们对快照最初建立的时候会有影响，但是之后，就不会再因快照引发写操作了。
- 下一步是估计在数据块被复制到快照之前，你在数据块上写入时 I/O 的数量会是多少（这里不是指对同一份数据所作的一次又一次的更改）。估算要宽松一点。估算出来的值就是你预计快照要产生额外 I/O 的数量。（还要加上一点点 LVM 进程所要使用的数量。）
- 使用 vmstat 或 iostat 来获取统计信息，以获知服务器在每一秒里在写多少个数据块。有关这方面的工具，请查看第 7 章。
- 测量（或估算）你把备份复制到别的位置上所花费的时间，换句话说，你的 LVM 快照为了被复制要打开多久。

让我们假设你已经估算出写操作的一半会引发快照的写时复制，服务器每秒钟写 10MB 数据，如果要把备份复制到另外一台服务器上，它要花费 1 小时（3 600 秒）的时间。这样，你将需要 $1/2 \times 10\text{MB} \times 3600$ ，即 18GB 的空间用于快照。谨慎地考虑到警告之类的信息，还可以再增加一些空间。

有时很容易算出打开快照时多少数据会被改掉。让我们再回到在别处多次用到过的一个例子，BoardReader 论坛搜索引擎在每个节点上大约有 1TB 的 InnoDB 表，然而，我们知道这里最大的开销是载入新数据。每天大约有 10GB 的新数据会被加进来，因此，对于快照而言，50GB 才算是充分的空间。这个估算也不总是准确的，在一个点上，我们会运行一个耗时很长的 ALTER TABLE 的命令，一个接一个地修改数据分块，每一个分块上的更改还会涉及超过 50GB 的数据。当这个命令运行时，我们就无法做备份了。

其他用途和替代方法

你可以把快照不仅仅用作备份。举例来说，它们可作为一个有潜在危险操作之前的“检查点”。有一些系统，比如 ZFS，能让你将快照升级为原始的文件系统。这有助于你把当前的数据回滚到做过快照的时间点上。

文件系统快照也不是取得数据即时副本的唯一途径。另外还有一个可选择的方法是 RAID 分割：如果有一个 3 碟软件 RAID 镜像，举例来说，你可以将其中一碟从镜像上移下来，然后单独把它挂接上。这里就没有了写时复制时系统资源的“罚款”，如有必要，还能很方便地把这种“快照”升级为主文件。

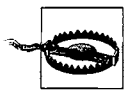
11.5 从备份中还原

还原是真正重要的事情。在这一节里，我们将集中于 MySQL 类型的还原，并且假定你知道如何处理环境里的其他问题。要例行公事地进行“消防演习”，这样在真正的紧急状况发生时，你就知道如何去备份和还原数据。首先要做的是测试备份记录。

怎样还原你的数据取决于你如何备份数据。你可能需要完成下面的全部或部分步骤：

- 关闭 MySQL 服务器。
- 在服务器配置和文件权限上做好记录。
- 把数据从备份记录移到 MySQL 数据目录下。
- 更改配置。
- 更改文件权限。
- 在有限访问权下重新启动服务器，然后等它完全启动。
- 重新加载逻辑备份文件。
- 检查和重放二进制日志。
- 核实恢复出来的内容。
- 使用完全访问来重启服务器。

505 在接下来的各小节里，我们会演示如何按需要来操作每一个步骤。对于某些备份方法或工具，我们会增加注释，表明这些方法或工具在本章的后面部分里会出现。



警告：如果你可能还会需要当前版本的文件，那就不要让还原出来的文件覆盖它们。

举例来说，如果备份记录里包含了二进制日志，你需要重放日志来做即时点还原，那就不要让当前最新的日志被备份记录里的旧版本日志覆盖了。如果必要，就把他们重命名或者移到别的位置上去。

11.5.1 限制访问 MySQL

Limiting Access to MySQL

在还原时，除了还原进程之外不让其他一切程序来访问 MySQL，有时也很重要。在复杂的系统里，这一点就很难被保证。我们喜欢在启动 MySQL 时使用 `--skip-networking` 和 `--socket=/tmp/mysql_recover.sock` 选项，这可以保证任何现存的应用不会来访问 MySQL，直到我们对 MySQL 检查完毕并上线。这对于逻辑备份尤其重要，因为它是分成一块块被加载的。

11.5.2 还原裸文件

Restoring Raw Files

还原裸文件相当的直接——也就是说它没有太多选项可以用。这是好事情也是坏事情，具体要根据还原的要求而定，通常是把文件复制到指定位置就可以了。

是否需要关闭 MySQL 取决于使用的存储引擎。MyISAM 的文件一般都是相互独立的，只要复制每个表的 .frm、.MYI 和 .MYD 文件就可以了，哪怕服务器仍在运行。一旦有人要查询这些表，或者服务器要查找表时（比如执行 `SHOW TABLES` 命令），服务器就会自动发现这些新加进来的表。如果在复制表文件的时候，那个表正打开着，那可能会引起点麻烦，因此，在复制表文件之前，你应该把这些表删除掉，或者重命名，或者干脆使用 `LOCK TABLES` 和 `FLUSH TABLES` 把它们关闭。

InnoDB 是另外一个麻烦。如果你正在还原一个通常的 InnoDB 数据库，其中的表全都存放在一个单独的表空间里。这样，你必须先关闭 MySQL，把这些文件复制或移动到另外的地方，然后重启 MySQL。你也需要确认 InnoDB

的事务日志文件与表空间文件是相匹配的。如果两者不匹配——例如你替换了表空间文件，但是没替换事务日志文件——InnoDB 会拒绝启动。这就是数据文件跟日志文件一起备份的紧要性之一。

如果你正在使用的是新版 InnoDB 一表一文件结构 (innodb_file_per_table)，InnoDB 会把数据和索引存放在一个 .ibd 文件里，这个文件类似于 MyISAM 里的 .MYI 和 .MYD 文件的合体。现在，你也可以在服务器还在运行时，通过复制文件来对单独的表做备份和还原操作了。但是，它还是没有 MyISAM 那么简单，那个单独文件不能完全独立于 InnoDB 而存在。每一个 .ibd 文件包含了一些内部信息，它们告诉 InnoDB 如何将它们与主（共享）表空间联系起来。当你还原这样一个文件时，必须要告诉 InnoDB 去“导入”这个文件。

501

在这个还原过程里有很多约束条件，具体的可以阅读 MySQL 帮助手册里的关于使用“一表一个表空间”的章节。这些约束条件里，最大的约束是你只能将表还原到备份出来的服务器上。用这样的配置条件来备份和还原表也不是不可能，只是要比你想象的更麻烦一些。

所有这些复杂性意味着还原裸文件是枯燥乏味的，而且还容易出错。一个很好的经验法则是：还原过程越艰难复杂，你就越要保护好逻辑备份。在裸文件备份出了点问题，不能确定是否可以供 MySQL 使用的情况下，事先保存一份逻辑备份是个不错的主意。

在还原裸文件后启动 MySQL

在做完还原之后，启动 MySQL 服务器之前，你还需要做一些事情。

首要的，也是最容易被遗忘的问题之一，就是检查服务器配置，在启动 MySQL 服务器之前，确认那些还原的文件有正确的属主 (Owner) 及权限。这些属性必须正确，否则 MySQL 就没法启动了。这些属性在服务器之间各不相同，因此要翻一下笔记，看看是否都是你要设置的属性。典型地，你要让 mysql 的用户和组拥有这些文件和目录，只有他们才能读和写，不能有其他入。

我们也建议在服务器启动时，查看 MySQL 错误日志。在一个 Unix 类型的系统里，你可以这样来查看日志：

```
$ tail -f /var/log/mysql/mysql.err
```

错误日志的确切位置在各系统里有所不同。一旦能够监控到日志，你就可以开始启动 MySQL，然后查看是否有错误发生。如果一切进展顺利，那你就拥有了一台被完美还原的服务器。

在新版的 MySQL 里，查看错误日志更显得重要。老版本的 MySQL 在 InnoDB 有错误时，根本不会启动，但是在新版里，服务器仍然会启动，它只是把 InnoDB 关闭了。即使服务器看起来没有任何问题地启动了，你还是应该在每个数据库里运行 SHOW TABLE STATUS 查看一下，然后再检查一遍错误日志。

11.5.3 还原逻辑备份

502

如果你使用逻辑备份来还原，而不是裸文件，那你就要用 MySQL 服务器本身把数据加载到表里去，不再是简单地用操作系统把文件复制到某个地方。

在加载那个导出的文件之前，你需要花些时间考虑一下它有多大、需要多久才能加载上去，以及其他任何需要在加载前考虑的问题（例如通知用户、关闭应用的一部分功能）。临时关闭二进制日志也是个好主意，除非你要把还原过程复制到从服务器上去：一个巨大的导出文件很难被服务器加载，更何况把它写到日志里，这会增加更多（很可能是不必要）的系统开销。对于某些存储引擎来说，加载大文件也有个顺序问题。举例来说，在一

个单独的事务里把 100GB 的数据加载到 InnoDB 里绝不是个好主意，因为这会导致一个巨大的事务。把文件分成一块块容易管理的备份数据，然后每一块一个事务，最终把文件全部加载上去。

你可能会用到两种还原方法以对应于你做的逻辑备份的类型。

加载 SQL 文件

如果你有一个 SQL 导出文件，文件里包含了可执行的 SQL 语句。你要做的就是运行它们。假定你把 Sakila 示例数据库和式样备份到一个单独的文件里，下面这行典型的命令就是你用来还原的：

```
$ mysql < sakila-backup.sql
```

你也可以在 mysql 命令行客户端上用 SOURCE 命令来加载 SQL 文件。对于同一个事情，虽然这显得很不一样，但是它让一些事情变得简单了。举例来说，如果你是 MySQL 上的管理员用户，你可以直接关闭日志（原先还需要通过客户端连接来关闭），然后就可以加载文件而无需重新启动 MySQL 服务器：

```
mysql> SET SQL_LOG_BIN = 0;  
mysql> SOURCE sakila-backup.sql;  
mysql> SET SQL_LOG_BIN = 1;
```

如果你使用 SOURCE，那就要注意在执行过程中如果发生一个错误，不会使整批语句退出执行，默认情况是只有当你把文件重定向到 mysql 后才会退出。

如果压缩了备份记录，不要分开来做解压缩和加载。它们可以用一个单独的操作来完成，这样会快很多：

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

如果你用 SOURCE 命令加载一个压缩了的文件，请阅读下一节里关于命名管道的讨论。

若只还原一张单独的表（例如 actor 表）又会怎样？如果数据没有换行符，而且样式已经固定下来，那也不难还原：

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

或者，文件是压缩的：

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`' | mysql sakila
```

如果还原数据时要创建表，而那个单独文件又包含了全部的数据库，这样，你就不得不编辑这个文件了。这就是为什么很多人喜欢把每个表导出到各自对应的文件里。许多编辑器无法处理这样的巨大文件，特别是当它们被压缩过的时候。更何况，你不是真地想要编辑那个文件——你只想取出相关的几行——这样就可以用命令行来做余下的工作了。使用 grep 能很容易地从指定的表里取出 INSERT 语句，就像我们在前一个命令里做的那样。但是，取得 CREATE TABLE 语句会更难一些。这里有一个 sed 脚本可以帮你取出想要的段落：

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

我们承认这个命令的语义看上去很模糊。如果你非得用这种方式来还原数据，那备份记录一定是设计糟糕的。稍微有一点计划性，就可能防止你陷入这样的境地：你恐慌万分，拼命想找出 sed 是怎样工作的。其实只要事先将每个表备份到它们自己的文件里就行了，或者，更好一点，把数据和样式分开来备份。

加载定界符文件

如果你是通过 SELECT INTO OUTFILE 来导出数据的，就必须用带同样参数的 LOAD DATA INFILE 来做还原。你也可以使用 mysqlimport——它是 LOAD DATA INFILE 的一个封装，依赖于命名惯例来决定从哪里加载文件的数据。

我们希望你导出数据样式，而不仅仅是数据。如果这样做了，它就是一个 SQL 导出文件了，你可以用上一节讲到的技术来还原它。

在 LOAD DATA INFILE 里有一个巨大的你可以用到的优化。它必须从文件里直接读取，因此，你可能会认为必须在加载前先做解压缩。这其实是个非常缓慢，又是磁盘密集的操作过程。但是这里还有一个变通的办法——至少在支持 FIFO “命名管道”文件的系统上（例如 GNU/Linux）是可行的。首先创建一个命名管道和流，并把数据压缩到里面：

```
$ mkfifo /tmp/backup/default/sakila/payment.fifo
$ chmod 666 /tmp/backup/default/sakila/payment.fifo
$ gunzip -c /tmp/backup/default/sakila/payment.txt.gz > /tmp/backup/default/sakila/
payment.fifo
```

要注意到：在这里，我们使用了一个大于号 (>) 把解压缩结果重定向到了 payment.fifo 文件——这不是一个管道记号（如果是这种记号，就会在两个程序之间创建匿名管道）。payment.fifo 本身就是一个命名管道，所以这里不需要使用匿名管道。

这个管道会一直等待，直到某个程序在另一端把它打开，读取数据。这是很清晰的一个过程：MySQL 能够从管道里读取压缩了的数据，就像从其他文件里读取一样。如果考虑周全，就不要忘记关闭二进制日志：

504

```
mysql> SET SQL_LOG_BIN = 0; -- Optional
-> LOAD DATA INFILE '/tmp/backup/default/sakila/payment.fifo'
-> INTO TABLE sakila.payment;
Query OK, 16049 rows affected (2.29 sec)
Records: 16049 Deleted: 0 Skipped: 0 Warnings: 0
```

一旦 MySQL 加载完数据，gunzip 就退出了，然后，你可以删除那个命名管道。你可以使用同样的技术在 MySQL 命令行客户端上用 SOURCE 命令来加载压缩文件。

为什么要测试备份？

本书的作者之一最近为了节省空间，让数据处理能更快一些，把表里某列的数据类型从 DATETIME 改为 TIMESTAMP——就像在第 3 章里推荐的那样。这种表最终的定义方式如下：

```
CREATE TABLE tbl1 (
  col1 timestamp NOT NULL,
  col2 timestamp NOT NULL default CURRENT_TIMESTAMP
    on update CURRENT_TIMESTAMP,
  ... more columns ...
);
```

此表的定义在 MySQL 5.0.40（就是创建这个表时所在的服务器）里会引起一个语法错误。于是，你可以导出它，但是无法重新加载它。像这样怪异、无法预料的错误就是要测试备份记录的很重要的原因之一。你永远无法知道有什么来妨碍你还原数据。

11.5.4 即时点还原

Point-In-Time Recovery

MySQL 上最常用的即时点还原是还原最近的一个完全备份记录，然后重放这个备份记录以来的二进制日志（有时把这个做法叫做“前滚还原”）。有了二进制日志，你就可以在任何想要的点上来做还原。你甚至可以在没太多麻烦的情况下还原整个数据库。

一个常见的场景是在一个有害的操作后，要消除它所带来的影响，例如 DROP TABLE 操作。让我们举一个简化了的例子来说明只用 MyISAM 表是怎么做到这一点的。假定事情发生在午夜，备份进程正在运行一个命令（相当于下面这样一些语句），它的用途就是把数据库复制到同一服务器的另外一个位置上：

```
mysql> FLUSH TABLES WITH READ LOCK;
-> server1# cp -a /var/lib/mysql/sakila /backup/sakila;
mysql> FLUSH LOGS;
-> server1# mysql -e "SHOW MASTER STATUS" --vertical > /backup/master.info;
mysql> UNLOCK TABLES;
```

第二天，假设某人运行了下面这样的语句：

```
mysql> USE sakila;
mysql> DROP TABLE sakila.payment;
```

为了便于说明，我们假设能在数据库隔离的状态下还原它（这也就是说该数据库里没有表涉及跨数据库查询）。我们还假设要在一段时间后才发现有人执行了这段不良的语句。这个例子的目标是还原受这段语句影响的所有数据，我们必须保留其他表的更新操作，包括在执行了这段语句后的那些操作。

这还不是所有难以处理的部分。首先，我们要停掉 MySQL 以防止有新的数据操作发生，然后只从备份里还原出 Sakila 数据库：

```
server1# /etc/init.d/mysql stop
server1# mv /var/lib/mysql/sakila /var/lib/mysql/sakila.tmp
server1# cp -a /backup/sakila /var/lib/mysql
```

我们在 my.cnf 里加入下面这两行参数来关闭那些普通的数据库连接：

```
skip-networking
socket=/tmp/mysql_recover.sock
```

现在就可以安全地启动服务器了：

```
server1# /etc/init.d/mysql start
```

接下来的任务就是在日志中找到我们要重放和要略过的语句。事实上，既然备份是在午夜时分生成的，服务器就只创建了一个日志。我们可以用 grep 来检查这个日志文件，找出那段不良的语句来：

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215 | grep -B
3 -i 'drop table sakila.payment'
# at 352
#070919 16:11:23 server id 1 end_log_pos 429 Query thread_id=16 exec_time=0
error_code=0
SET TIMESTAMP=1190232683/*!*/;
DROP TABLE sakila.payment/*!*/;
```

我们要略过的语句在日志的 352 位置上，紧接着的语句是在 429 位置上。于是，我们可以把日志从开始重放到 352 位置，然后再从 429 开始到最后。具体命令如下：

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215  
--stop-position=352 | mysql -uroot -p  
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215  
--start-position=429 | mysql -uroot -p
```

现在我们要做的就是检查一下以确保结果正确，然后关闭服务器，把 my.cnf 的配置改回来，再重新启动服务器。 506

11.5.5 更高级的还原技术

复制和即时点还原使用的是同一个机制：服务器的二进制日志。这意味着从一些不太明显的角度来看，复制在还原过程中也可以是个很有帮助的工具。在这一节里，我们会展示其中的一些可能性。这不是一个面面俱到的目录，但是它可以在你设计还原过程时提供一些参考。你要记得把在还原过程中要做的事情，先写下来演练一遍。

把延时复制用于快速还原

如同我们在本章前面所提到的那样，如果你能在从服务器执行那些可恶的语句之前发现这个问题，使用延迟复制从服务器能够使即时点还原更快、更容易。

这个过程跟前面小节里讲的有一点点不同，但是，理念还是一样的。你要先停掉从服务器，然后使用 START SLAVE UNTIL 来重放事件，直到那个要略过的语句之前。接着，你执行 SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1 来跳过这个坏语句。如果你要跳过好几个语句，就把 1 改成更大的数字（或者可以简单地使用 CHANGE MASTER TO 在日志里移位）。

接下来你要做的就是执行 START SLAVE，让从服务器运行起来，直到把日志都重放完毕。到此，你的从服务器已经做完了即时点还原里的所有脏活累活。现在，你可以把从服务器提升为主服务器了，刚才的还原也只有很少一点中断服务的时间。

你甚至都不需要一个延迟复制从服务器来加速还原过程，从服务器本身就很有用，因为从主服务器上获取日志，放在了另外一台机器上。如果主服务器的磁盘发生故障，那么从服务器上的重放日志就可能是唯一能信得过的最新的主服务器日志副本了。（把二进制日志放在一个 SAN 里，或者用 DRBD 复制它们，会显得更加安全，这在第 9 章里已经讨论过了。）

用日志服务器来还原

这里是本书作者将复制用于还原的方法：搭建一台日志服务器。（具体做法的细节请查看第 374 页的“构建一台日志服务器”。）

日志服务器有更好的兼容性，用在还原上比 mysqlbinlog 更方便。这不仅是因为有 START SLAVE UNTIL 选项，还因为你可以设置复制规则（例如 replicate-do-table）。使用日志服务器，你能做比原来在做的还要复杂的过滤。

举例来说，日志服务器可以让你很轻松地恢复一个单独的表。这本来对于 mysqlbinlog 和命令行工具来说很难做到的——事实上，它确实很难，我们也建议你不要去尝试了。 507

让我们假设之前有一个粗心的开发人员把一个表删除了，现在我们想把它还原出来，但又不能恢复到昨天晚上的那个备份状态。下面就是如何用日志服务器来处理这个事情：

1. 把你要还原的那台服务器叫做服务器 1。
2. 把昨晚的备份还原到另外一台服务器上（就叫它服务器 2）。在这台服务器上运行还原过程，以免在还原过程中出错把事情搞得更糟。
3. 按照第 374 页的“搭建日志服务器”指引，搭建一台针对于服务器 1 日志的日志服务器。（为更加小心，有个好主意是：把日志复制到另外的服务器上，然后在那里搭建日志服务器。）
4. 将服务器 2 的配置文件作如下修改：

```
replicate-do-table=sakila.payment
```
5. 重新启动服务器 2，使用 `CHANGE MASTER TO` 命令，将它作为日志服务器的从服务器。将它配置为从昨晚备份记录的日志定位位置上读取数据，这时还不能运行 `START SLAVE`。
6. 检查服务器 2 上 `SHOW SLAVESTATUS` 的输出项，核对数据是否都正确了。在这方面一定要三思而后行。
7. 找到那段不良语句在日志里的位置，然后在服务器 2 上执行 `START SLAVE UNTIL` 来重放事件，直到那个位置之前。
8. 用 `STOP SLAVE` 停止服务器 2 上的进程。现在，那个被删除的表应该已经还原回来了。
9. 把这个表从服务器 2 复制到服务器 1 上。

这个过程只有当服务器不涉及多表 `UPDATE`、`DELETE` 和 `INSERT` 操作时才是可能的。以上 3 种操作的任何一种都会使数据库产生不同的状态，跟原来单独一个表操作时产生的日志不一样。因此，这个表还原出来的数据可能会跟它应有的不尽相同。

11.5.6 InnoDB 还原

InnoDB 在每次启动时都会检查数据和日志文件，以确定是否需要还原。然而，InnoDB 的还原跟我们在本章里讲的还原又是两码事。它不是还原原先备份的数据，而是根据日志里的事务状态，把数据更新到数据文件，或者作回滚。

InnoDB 真正在做的还原工作其实要比描述的更复杂。我们关注的焦点是当 InnoDB 发生严重问题时，该怎么去做还原。

多数时候，InnoDB 都善于自己修复问题。除非 MySQL 遇到了 Bug 或硬件故障，你一般都无须做超乎寻常的事情，甚至包括预防服务器断电。InnoDB 在启动时会执行一个通常的还原过程，将一切都恢复正常。在它的日志文件里，你会看到像下面这样的消息：

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
```

InnoDB 会在日志文件里以百分比的形式打印出还原进程的进度消息。也有些人报告说没看到这些消息，直到整个过程完成之后。请耐心等待，没有办法能加快这个过程。把它关闭再重启只会使这个过程更加漫长。

当你的硬件出现一个严重的问题（例如内存或磁盘错误）时，或者在 MySQL 或 InnoDB 里碰到了 Bug 时，你就不得不进行干预，或者强制还原，或者防止从那个错误处开始还原。

InnoDB 损坏的根源

InnoDB 通常情况下是相当健壮的。它是建立在可靠性之上的，有许多内建的合理性检验来防止、发现和修复被损坏的数据——比其他存储引擎要多得多。然而，它也不可能让自己“刀枪不入”。

从最低限度上来讲，InnoDB 依赖于无缓冲 I/O 调用和 `fsync()` 调用，在数据被安全写入物理媒体之前，它们都不会返回。如果你的硬件没有真正写入这些数据，那 InnoDB 也无法保护数据。一次服务器崩溃就会引起数据损坏。

许多 InnoDB 损坏问题都是与硬件相关连的（例如在损坏的页上写入是由掉电或坏内存引发的）。然而，在我们的经历里，配置不当的硬件也是一个很大的问题来源。常见的配置不当包括在 RAID 卡里开启了回写 (Writeback) 缓存，而 RAID 卡又没有后备电源；或者在硬盘驱动器上开启了回写缓存。这些错误都会使控制器或驱动器谎报 `fsync()` 已经完成，实际上，那些数据还只在回写缓存里，而不是在磁盘上。换句话说，硬件无法提供 InnoDB 所需要的保证，来保障数据的安全。

有时，这些不当的配置都是机器默认配置好的，因为这有更好的性能表现——这对一些用途而言确实是好的，但是对事务数据库服务器来说却不好。如果服务器不是你设置的，你最好经常地检查一下这些配置。

如果你在网络附属存储 (Network-attached Storage, NAS) 上运行 InnoDB，你也会得到被损坏的数据，因为在该类设备上完成 `fsync()` 操作，只是意味着设备已经接收到该数据。这时，如果 InnoDB 崩溃了，那数据还是安全的，但是，如果 NAS 设备崩溃了，那数据就可能没了。

有时数据损坏会比其他问题糟糕好几倍。严重的损坏会使 InnoDB 或 MySQL 崩溃。但是，轻微的损坏仅仅意味着一些事务的丢失，因为日志文件不是真正与磁盘同步的。

怎么还原损坏的 InnoDB 数据

509

InnoDB 的损坏有 3 种主要类型，每一种对数据还原有着不同层面的影响：

二级索引损坏

通常而言，你可以使用 `OPTIMIZE TABLE` 来修复损坏了的二级索引。同样地，你可以使用 `SELECT INTO OUTFILE`，删除后再重建这张表，然后使用 `LOAD DATA INFILE` 来完成修复。这些修复过程都是通过建立新表，进而重建受到影响的索引。

集群索引损坏

你可能要用到 `innodb_force_recovery` 设置来导出这个表（更多内容在后面展开）。有时，这个导出过程会使 InnoDB 崩溃，如果确实崩溃了，那你就需要导出某个范围内的行，跳过那个会引起崩溃的页。一个损坏了的集群索引要比一个损坏的二级索引严重得多，前者会影响到数据所在的行，但是，在许多情况下，修复这些受影响的表也是有可能的。

损坏的系统结构

这里指的系统结构包括了 InnoDB 事务日志、表空间里的撤销日志区域、数据目录。这类损坏可能需要做整个数据库的导出和还原，因为这些结构影响到了 MySQL 内部绝大部分的工作任务。

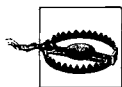
你一般可以修复一个损坏的二级索引而不丢失数据。然而，另有两种情形会引起一些数据的丢失。如果有一个备份，那最好还是从备份记录里还原数据，而不要试着从损坏的文件里去提取数据。

如果你必须从损坏的文件里提取数据，那一般要让 InnoDB 运行起来，然后使用 `SELECT INTO OUTFILE` 导出数据。如果服务器已经崩溃了，无法再启动 InnoDB，你可以配置 InnoDB，防止它做常规还原，并阻止其后台处理过程的运行。这也许可以帮你启动服务器，在缺少检查或没有检查的情况下做一个逻辑备份出来。

`InnoDB_force_recovery` 参数用来控制 InnoDB 在启动和做常规操作时要做哪一种类型的操作。通常情况下这个值是 0，最高可以提高到 6。MySQL 使用手册里记录了每个数值会产生的操作行为。在这里，我们不再重复这段信息，但是，我们要告诉你：在有点危险的前提下，你可以把这个数值调高到 4。使用这个设置时，若数据页有损坏，你就会丢失一些数据；如果将数值设得更高一点，你会从损坏的页里提取到坏掉的数据，但是提高了执行 `SELECT INTO OUTFILE` 的崩溃风险。换句话说，层次 4 对你的数据没有损害，但会丧失修复问题的机会；层次 5 和层次 6 会更主动地修复问题，但是损害数据的风险会很大。

510 当你把 `innodb_force_recovery` 设为大于 0 的某个值时，InnoDB 在本质上就是只读的，但是你仍然可以创建和删除表。这可以阻止进一步的损害，InnoDB 会放松一些常规检查，因为当发现坏数据时，它不会特意崩溃。在常规操作中，这就是一个安全保障。但是当你在还原时，就不会希望这样了。如果你需要强迫 InnoDB 还原，有个好主意就是配置 MySQL，使它在操作完成之前，不允许接受常规的连接请求。

如果 InnoDB 的数据损坏到了根本不能启动 MySQL 的程度，你可以使用 InnoDB 还原工具箱从表空间的数据页里直接取出数据。这些工具由本书的几个作者开发，可以在 <http://code.google.com/p/innodb-tools/> 网站上免费获取。



警告：我们通常不会提到 MySQL 中具体的错误，但是在许多版本的 MySQL 里，在定义 `innodb_force_recovery` 之后，会有一个非常严重的 Bug 阻止你执行还原过程。您可以在 <http://bugs.mysql.com/28604> 上追踪到这个 Bug 的状态。如果你在导出一个损坏了的 InnoDB 表时看到 “Incorrect key file” 错误信息，你就需要阅读一下这个 Bug 的报告，看看是否确为这个原因。如果是，可以使用 MySQL 5.0.22 来导出数据。作为一个美好愿望，希望你永远都不需要担心此类问题。

11.6 备份和还原的速度

Backup and Recovery Speed

在正确性之后，速度就是高性能系统备份/还原里的最重要因素了。以下是要考虑的与速度有关的因素：

锁定时间

在备份的时候，你的锁会保持多长时间，例如全局的 `FLUSH TABLES WITH READ LOCK` 锁？

备份时间

把备份复制到目标位置上要多长时间？

备份的负载

当备份复制到目标位置时会给服务器的性能带来多大的影响？

还原时间

还原时间包括了把备份记录从存放位置复制到 MySQL 服务器的时间、重放二进制日志的时间，以及其他相关时间。

这里最大的权衡在于备份时间和备份负载。通常，你可以用一项开销来抵消另一项开销，例如更多服务器性能的代价来优先做备份。

你可以设计备份以利用负载模式的优点。例如，如果服务器在夜间有 8 小时处于 50% 负载量，那你可以试着把备份方案设计为给服务器增加少于 50% 的加载，但仍然保证在 8 小时内完成。

你可以通过多种方法来达到这个效果：例如，你可以用 `ionice` 和 `nice` 优先执行复制和压缩操作，使用不同级别的压缩方式，或者在备份服务器上压缩数据，而不是在 MySQL 服务器上。你也可以是用 `O_DIRECT` 或 `madvise` 使复制操作绕过操作系统的缓存，这样它们就不会去“污染”服务器的缓存了。

通常来讲，从裸备份里获取数据的速度会更快一点，相关工作也更少。然而，逻辑备份是一个重要的补充，因为裸文件不够轻便，也不是永远都可以还原的，而且当其中有损坏时也难以被发现。如果定期从裸文件副本里做逻辑备份，你花一点额外的精力就可以获得两者之长。

11.7 备份工具

Backup Tools

除停止服务器、还原数据、重启服务器之外的那些事情都是相当复杂的，这是演练和脚本化这些操作的根本原因。在本小节接下来的篇幅里，我们要介绍一些在备份/还原时很有用的工具，它们可以用来生成操作脚本，也可以用来做进一步到位（One-shot）的导出和还原。

本书的第一版里说到过“如果你有复杂的配置，或者有不同寻常的需要，可能没有一个方案能为你完成预定的任务。这时，你需要构建一个自定义方案了。”时间已经过去了很久，今天我们建议不要使用脚本来操作备份工具，除非真地必须。如果某个现存的工具能很好地满足你的需要，那真是个不错的机会；如果没有，你也能修改其中的一个来完成你要做的事情。

尽管如此，有一些更复杂的备份场景还是要自定义的脚本来做，因此，我们在本章末尾部分包含了一些基本的 how-to 建议。

11.7.1 Mysqldump

mysqldump

用来创建数据和式样逻辑备份的工具，最流行的应该是 `mysqldump` 了。`Mysqldump` 是随着 MySQL 服务器一起提供的，所以，你一般都用不着安装它。它是个多用途工具，可以用来完成很多任务，例如将表从一台服务器复制到另外一台：

```
$ mysqldump --host=server1 test t1 | mysql --host=server2 test
```

在本章里，我们已经展示过好几个使用 `mysqldump` 创建逻辑备份的例子了。在默认条件下，它输出的是一个包含了所有建表和填充数据命令的脚本。它也有一些选项用来输出视图、存储过程和触发器。以下是一些典型用途的例子：

- 将服务器上的所有东西都备份到一个单独的文件里：

```
$ mysqldump --all-databases > dump.sql
```


- 只将 Sakila 示例数据库做逻辑备份：
`$ mysqldump --databases sakila > dump.sql`
- 只将 sakila.actor 表做逻辑备份：
`$ mysqldump sakila actor > dump.sql`

你可以用--result-file 选项来指定一个输出文件，这个选项能帮你防止在 Winodwn 上出现换行转换：

```
$ mysqldump sakila actor --result-file=dump.sql
```

mysqldump 的默认选项并不对所有有重要目的的备份都适合。你可能想通过指定一些选项来明确地控制输出结果。这里有一些常用的选项，可以使 mysqldump 更加有效率，输出的结果更容易使用：

--opt

启用一组选项，它们会关闭缓冲区（这个会使服务器耗尽内存），导出数据时把更多的数据写在更少的 SQL 语句里，这样在加载的时候就更有效率了，同时可以做其他一些有用的事情。更多细节可以阅读版本帮助文件。如果关闭了这组选项，mysqldump 会在把表写到磁盘之前，把它们都导出到内存里，这对于大型的表而言是不切实际的。

--allow-keywords, --quote-names

使用户能在导出和恢复表时，使用保留字作为表的名字。

--complete-insert

使用户能在带有不同字段的表之间移动数据。

--tz-utc

使用户能在不同时区的服务器之间移动数据。

--lock-all-tables

使用 FLUSH TABLES WITH READ LOCK 获取一个全局一致的备份。

--tab

使用 SELECT INTO OUTFILE 导出数据，使备份和还原速度更快。

--skip-extended-insert

使每一行数据都有自己的 INSERT 语句。必要时这可以帮你有选择地还原某些行。它的代价是大文件导入到 MySQL 时，开销会更大。因此，只有在你需要时，才能启用它。

如果你在 mysqldump 上使用-databases 或--all-databases 选项，那么最终导出的数据会是每个数据库里都一致的，因为 mysqldump 会同一时间锁定并导出一个数据库里的所有表。然而，来自不同数据库的各个表就未必是相互一致的。使用--lock-all-tables 选项可以解决这个问题。

11.7.2 mysqlhotcopy

MySQLhotcopy 是一个 Perl 脚本，包含在标准版 MySQL 服务器下载包里。它是为 MyISAM 表设计的，在我们的可选范围里，它不会做“热”备份，因为它在复制表的时候要锁定所有表。虽然它曾是在活动服务器上做备份的最流行的工具之一，但是如今它已经不怎么流行了。许多高性能安装包正在远离 MyISAM，甚至于你能使用的只有 MyISAM 表，而文件系统快照常常缺少插入性，因为它们锁定数据的时间可以更短。

作为一个例子，我们用所有的 MyISAM 表创建了 Sakila 示例数据库的一个副本。为了将这个数据库复制到另一个数据库，我们运行如下命令：

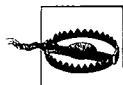
```
$ mysqlhotcopy sakila_myisam /tmp
```

这个命令在/tmp 下创建了一个名叫 sakila_myisam 的子目录，里面包含了从那个数据库里复制出来的所有表：

```
$ ls -l /tmp/sakila_myisam/
total 3632
-rw-rw---- 1 mysql mysql 8694 2007-09-28 09:57 actor.frm
-rw-rw---- 1 mysql mysql 5016 2007-09-28 09:57 actor.MYD
-rw-rw---- 1 mysql mysql 7168 2007-09-28 09:57 actor.MYI
... omitted ...
-rw-rw---- 1 mysql mysql 8708 2007-09-28 09:57 store.frm
-rw-rw---- 1 mysql mysql 18 2007-09-28 09:57 store.MYD
-rw-rw---- 1 mysql mysql 4096 2007-09-28 09:57 store.MYI
```

它复制了数据库里每一个表的数据、索引和表定义文件。为了节省空间，你可以使用 `noindices` 选项让它对每个 .MYI 文件（这个文件是 MySQL 用来重建索引的）只备份 2048 个字节数的数据。如果你使用了这个选项，就要在还原文件之后重建相关表的索引。你可以使用带 `recover` 选项的 `myisamchk` 来做，或者使用 `REPAIR TABLE` 这个 SQL 命令。

Mysqldhotcopy 相当复杂，并不是所有人都喜欢这种兼容性，因此，许多人运行自己编写的脚本，用一种稍微有所差别的方式，来完成本质上完全相同的工作。



警告：用 `innodb_file_per_table` 配置 InnoDB 后，`mysqlhotcopy` 会复制 .ibd 数据文件，但这是没有用的。不要相信这个虚假的安全感，这不是安全备份 InnoDB 数据的途径。

11.7.3 InnoDB 热备份

InnoDB Hot Backup

InnoDB 热备份——`ibbackup`，是一个由 InnoDB (Innobase) 发布的商业化工具。使用它时，不用停止 MySQL，设置锁，或者中断常规的数据库活动（虽然它会给服务器添加一些额外的负载）。它也可以压缩备份记录。

配置 `ibbackup` 的方法是给它提供一个与服务器上 `my.cnf` 文件相匹配的配置文件，并且放在不同的数据目录下。这个工具会读取这两个配置文件，然后把 InnoDB 文件从服务器复制到第 2 个配置文件指定的位置上：

514

```
$ ibbackup /etc/my.cnf /etc/ibbackup.cnf
```

还原备份记录时，先关闭 MySQL，然后运行下面这个命令：

```
$ ibbackup --restore /etc/ibbackup.cnf
```

这里会有一个小问题：`ibbackup` 只复制了 InnoDB 文件，不包括表的定义或其他必要的文件。Innobase 也提供了一个免费的 `innobackup` 助手脚本，它把文件复制、表锁定和 `ibbackup` 封装在一个单独的命令里，这样就能像复制 InnoDB 文件一样，复制表的定义和 MyISAM 文件了。不像 `ibbackup` 那样，这个脚本会中断 MySQL 的常规处理进程，因为它里面使用了 `FLUSH TABLES WITH READ LOCK`。

在我们的可选范围内，使用 LVM 的快照功能备份 InnoDB 比 `ibbackup` 更加便利和有用。LVM 最大便利之一是不需要在文件系统上再给数据做一次复制——你可以创建快照，按你的意愿执行 InnoDB 备份，然后直接把它

发送到备份目的地。

LVM 和 ibbackup 在通常情况下都有不相上下的性能，具体要看如何配置备份，以及是否有写密集的工作负载。在那种情形下，LVM 会有很多写时复制的系统开销。在另一方面，ibbackup 不会随着数据尺寸的增大而呈线性增长，它的工作原理是逐页复制数据文件，然后在复制过去的文件上重放日志文件来“还原”，直到备份过程结束。

11.7.4 mk-parallel-dump

这个工具是 Maatkit (<http://maatkit.sourceforge.net>) 的一部分。它能同时执行几个备份操作。

在默认情况下，mk-parallel-dump 的行为就像对 mysqldump 的多线程封装，但是，它也能使用 SELECT INTO outfile 来输出 tab 分隔文件。mk-parallel-dump 的默认配置是每个 CPU 一个线程，因此服务器上的 CPU 越多，它工作起来也越快。它也可以按预定尺寸一块一块地备份每一个表——这在还原 InnoDB 表时更加快速，其中的好处是你可以避免还原时出现的巨大事务。巨大的事务会使 InnoDB 的表空间增长到非常庞大的规模，若有错误发生，回滚的时间会延长很多。

这个工具还有一些漂亮的功能，例如能够做增量备份；将一些表组成群放入逻辑备份集合里。基准测试显示：并行地进行逻辑备份可以显著提高整体备份速度。

Maatkit 也包含了 mk-parallel-restore——一个做多线程导入时的伴侣程序。以上两者都善于利用 Unix 的经典技术（例如管道、FIFO 设备）来减少压缩/解压缩文件时对系统性能产生的影响。

11.7.5 Mylvmbackup

Lenz Grimmer 的 mylvmbackup (<http://lenz.homelinux.org/mylvmbackup/>) 是一个 Perl 脚本，在使用 LVM 快照做 MySQL 备份时，可以用它来自动化整个过程。它得到一个全局的读锁，然后创建一个快照，再释放锁。接着使用 tar 压缩数据，删除快照，用备份发生时的时间戳来命名这个 tar 压缩包。

11.7.6 Zmanda Recovery Manager

Zmanda Recovery Manager for MySQL，或者叫 ZRM (<http://www.zmanda.com>)，是我们在这里提到的备份/还原工具里面应用最广泛的。它有免费 (GPL) 和商业两种许可方式。它的企业版里带有一个管理控制台，提供了一个基于 Web 的图形化界面，可用于配置、备份、校验、还原、报告和调度，也能用于 MySQL 集群的备份。它具备一切通常所见的优点（例如后台支持）。

开源版本在基本功能上没有缩水，但它没有那些额外的好东西，例如基于 Web 的控制台。如果你善于使用命令行，它还是非常可用的。例如，你可以在命令提示符之后，做调度和检查备份。

ZRM 实际上不仅仅是个单一的工具，它更像个“备份协调员”。它在标准的工具和技术之上封装了自己的功能，例如 mysqldump 和 LVM 快照。它用标准的格式存储数据，因此，就没有必要购买专用软件来做还原。它强大功能之一是统一还原机制——无论备份是如何做成的，它都可以用一样的途径来还原。

图 11-2 显示了企业版里的日历界面，它能概览 MySQL 备份记录和日志分析器——Zmanda 称之为“数据库事件显示器”。本质上，它就是一个日志搜索工具，你可以使用普通搜索语法来找出想要找的事件，便于将备份还

原到某个日志事件或某个即时点。

安装和测试 ZRM

Zmanda 的网站上宣称：安装、执行和检验一个备份；设置和校验一个每日计划；执行一次还原、总共需要 15 分钟！作为一个检验，我们在一台运行着 Ubuntu 的笔记本上从头开始安装 ZRM。要下载的 ZRM 包很小，我们使用 `sudo dpkg -i mysql-zrm_1.2.1_all.deb` 将它安装好。这里还有几个先决条件，但是，我们使用 `sudo apt-get -f install` 很容易地就安装好了，整个过程花费不到 1 分钟。

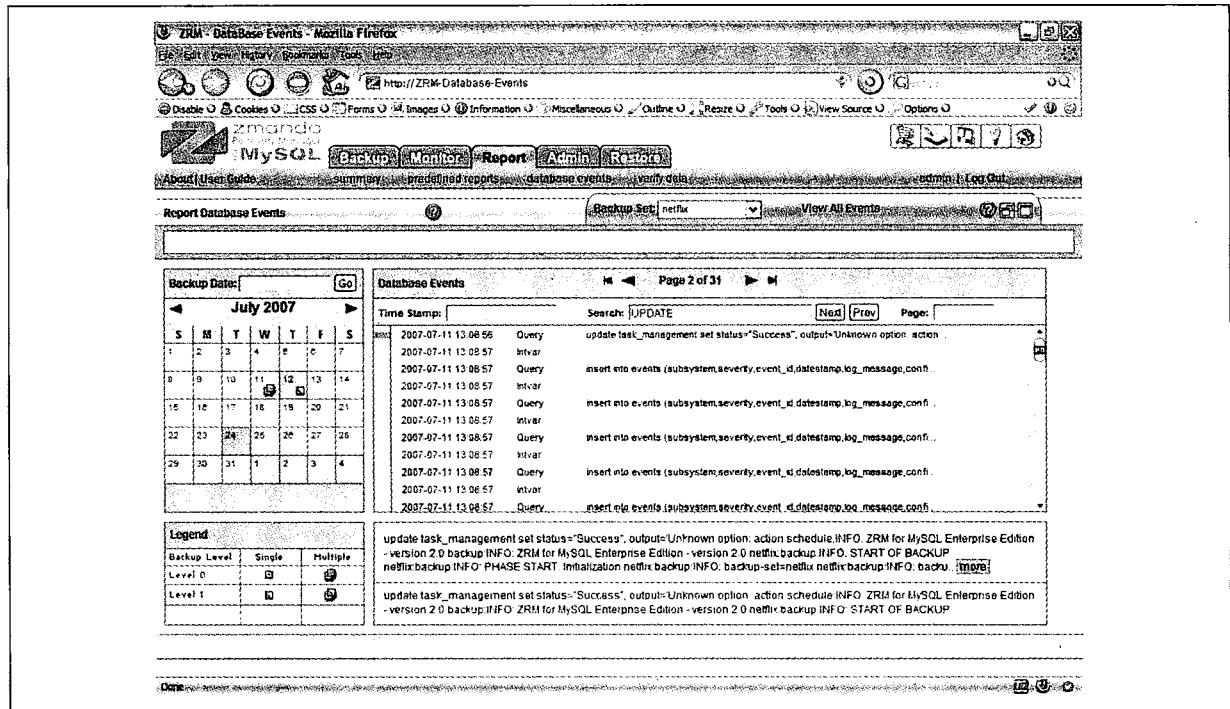


图 11-2: ZRM 的备份日历和日志搜索界面

我们按网站上的指令来配置备份集，它现在是 Sakila 示例数据库的一个逻辑备份。这大概花费了 3 分钟。然后输入以下这个命令来开始备份：

```
# mysql-zrm-scheduler --now --backup-set dailyrun
```

备份过程就只有一会儿，结果文件存放在 `/var/lib/mysql-zrm/dailyrun`。然后，我们再运行一遍备份，并故意给 ZRM 制造出一些错误来，例如杀掉它的一些子进程，给它错误的登录参数。它可以正确地检测到这些错误，然后写入备份通知邮件里发送出去。这里有个要注意的细节是日志会写到系统日志预期的位置上。

最后，我们删除了 sakila 数据库，又使用下述命令从最近一次成功的备份记录里将它还原出来：

```
# mysql-zrm-reporter --show restore-info --where backup-set=dailyrun
# mysql-zrm-restore --backup-set dailyrun --source-directory
/var/lib/mysql-zrm/dailyrun/20070930134242/
```

通常而言，ZRM 是一个精心设计的系统，具备了良好的错误检查功能，它能自动化许多繁琐的备份/还原工作，而且，正如其名称所显示的那样，它是自下而上都为还原而设计的。

11.7.7 R1Soft

R1Soft (<http://www.r1soft.com>) 提供了一个名叫 Continuous Data Protection 的产品, 这是个商业软件。它与文件系统快照类似, 只是当把快照复制到另外一台服务器上时, 它只能复制其中更改过的那部分数据。你可以用它把数据回滚到多个过去的版本。

11.7.8 MySQL 在线备份

MySQL Online Backup

MySQL 在线备份不是一个工具, 它只是 MySQL 5.2 (现在还是个 alpha 版) 里开发的一个功能, 可能会在 MySQL 6.0 里正式发布。

这个功能的界面就是一个新的 BACKUP DATABASE 语句, 它能以很高的速度对每个表做快照, 然后写入对应的文件里。它或者使用一个默认的驱动来备份所有类型的存储引擎, 或者为特定的一个存储引擎实现驱动以提高备份的效率。默认的驱动会堵塞其他 SQL 语言的运行, 但是原生驱动在备份时就不会这样。还原功能也包含在这个语句里了。

在本文还在写作的时候, 该项目在 5.2 版本的代码里已经有了一个初步的实现, 一些主要的功能已经实现。但是还有很多功能有待完成, 例如 MyISAM 的原生驱动、跨数据引擎的一致性备份。

在线备份是很值得期待的, 当它完成之后, 可能就是 MySQL 最重要的备份工具之一了。

11.7.9 备份工具的比较

Comparison of Backup Tools

表 11-2 提供了本章讨论到的各种备份工具的快速摘要信息。

表 11-2: 各备份工具的特点

| 特点 | mylvmbackup | mysqldump | mk-parallel- dump | mysqlhotcopy | ibbackup |
|----------|-------------|-----------|-------------------|-----------------|----------|
| 是否有数据块处理 | 可选 | 是 | 是 | 是 | 否 |
| 逻辑备份或裸备份 | 裸备份 | 逻辑备份 | 逻辑备份 | 裸备份 | 裸备份 |
| 数据引擎 | 全部 | 全部 | 全部 | MyISAM/ Archive | InnoDB |
| 速度 | 非常好 | 慢 | 好 | 非常好 | 非常好 |
| 能否远程备份 | 否 | 是 | 是 | 否 | 否 |
| 可用性 | 免费 | 免费 | 免费 | 免费 | 商业 |
| 许可证 | GPL | GPL | GPL | GPL | 私有版权 |

11.8 脚本化备份

Scripting Backups

我们建议如果你有现成的系统可用, 就不要重新发明轮子。但是, 你仍然有必要运行自己的脚本, 或者修改一个已有的脚本。下面是我们在工作中看到过的几种备份配置:

- 把许多服务器上的数据备份到几个备份服务器上，这些备份服务器拥有容量巨大、廉价（具体位置要看哪个服务器上有足够空间）它也要保证不同的备份生成到不同的服务器上，这样一来，即使丢了任何一台服务器也不是什么大问题。
- 把备份存档记录切分成很多份，对它们进行加密，然后存储到外部的数据中心——像 Amazon 的 S3 服务或其他大型存储服务。
- 把还原与复制集成起来，这样你就能从备份记录里克隆出一个从服务器。

我们展现一个示例程序如下，它带有一些必不可少的占有页面大量篇幅的脚手架代码，我们将列出这样一个典型的备份脚本的各组成部分，并用 Perl 脚本来显示其中的代码片段。你可以把它们当作基石，用它们来创建你自己的脚本。我们将按你使用它们时的大致次序来罗列：

健全性检查

让你和你同事的人生更简单一些——打开严格的错误检查功能，使用英文变量名：

```
use strict;
use warnings FATAL => 'all';
use English qw(-no_match_vars);
```

如果脚本在 Bash 上，你可以启用更严格的变量检查。当有个未定义的变量用于替换，或者程序退出产生错误时，运行以下语句会出现一个错误信息：

```
set -u;
set -e;
```

命令行参数

每个脚本都要接受命令行参数。如果你发现已经把诸如用户、密码之类的信息硬编码在配置里了，那你真地应该将它提升到更高的层次。

```
use Getopt::Long;
Getopt::Long::Configure('no_ignore_case', 'bundling');
GetOptions( .... );
```

连接到 MySQL

标准的 Perl DBI 库几乎到处都有，它提供许多强大而富有兼容性的功能。关于使用它的细节请阅读 Perldoc（可以在 <http://search.cpan.org> 上获得）。

```
use DBI;
$dbh = DBI->connect(
    'DBI:mysql;:host=localhost', 'user', 'pass', {RaiseError => 1});
```

关于命令行脚本，请阅读标准 mysql 程序的 -help 显示的文本，其中有大量选项可增进脚本的友好度。举例来说，以下是 Bash 环境下的枚举数据库列表：

```
for DB in `mysql --skip-column-names --silent --execute 'SHOW DATABASES'`
do
    echo $DB
done
```

停止和启动 MySQL

停止和启动 MySQL 的最好方法是使用操作系统上的方法，例如运行 /etc/init.d/mysql 下的初始化脚本。如果在 Windows 平台上，可以使用管理工具里的“服务”。这也不是唯一的途径。你可以用 Perl 通过一个已存在的数据库连接来关闭数据库。

```
$dbh->func("shutdown", 'admin');
```

当这行命令完成后，你也不能确信 MySQL 已经被关闭了，它也许只在进程表里被关闭。可以通过下面这行命令来关闭 MySQL：

```
$ mysqladmin shutdown
```

取得数据库和表的目录

每一个备份脚本都会向 MySQL 请求数据库和表的目录。要知道每一个数据库条目并不是真正的数据库，例如有些日志文件系统里的 lost+found 目录和 INFORMATION_SCHEMA。你也要确认脚本能够处理视图。如果 InnoDB 里有很多数据，执行 SHOW TABLE STATUS 会耗费很长一段时间：

```
mysql> SHOW DATABASES;
mysql> SHOW /*!50002 FULL*/ TABLES FROM <database>;
mysql> SHOW TABLE STATUS FROM <database>;
```

表的锁定、缓存刷新和解锁

你一定会需要锁定一个或多个表，或者根据表名锁定需要的表，或者干脆全局性地锁定所有对象；有时还要对它们的缓存进行刷新：

```
mysql> LOCK TABLES <database.table> READ [, ...];
mysql> FLUSH TABLES;
mysql> FLUSH TABLES <database.table> [, ...];
mysql> FLUSH TABLES WITH READ LOCK;
mysql> UNLOCK TABLES;
```

当你在锁定表并取得目录时，要当心竞争条件的发生。新表可以被创建，或者现存的表可以被删除或重命名。如果每锁定一个表就做备份，那你就无法得到一个一致性的备份。

刷新二进制日志

要服务器创建一个新的日志会比较难（这要在锁定表之后，开始备份之前来做）：

```
mysql> FLUSH LOGS;
```

如果你不考虑从日志文件的中间开始执行，这会使还原和增量备份做起更容易。这样做也有一些副作用：在刷新和重新打开日志时，会潜在破坏旧的日志条目。因此，你要小心，别丢掉你需要的数据。

获取二进制日志的位置

你的脚本应该能获得和记录主服务器和从服务器的状态——哪怕服务器只是一台主服务器，或者只是一台从服务器：

```
mysql> SHOW MASTER STATUS;
mysql> SHOW SLAVE STATUS;
```

这两条语句都能显示内容，并忽略你得到的任何错误，因此，你的脚本有可能读到所有信息。

导出数据

你有两个最佳的选择：使用 mysqldump，或者使用 SELECT INTO OUTFILE。

复制数据

使用本章里我们提到过的方法里的一种就可以。

以上这些是所有备份脚本的基石。其中，最困难的部分是编写还原的脚本。如果你想获得一些这方面的灵感，可以去看 ZRM 的源代码。它的脚本能做一些灵活的事情，例如把元数据与备份保存在一起，使还原更容易操作。

安全

MySQL 安全

保证 MySQL 安全是保护数据完整性和私密性的关键。就像保护 Unix 或 Windows 的登录账号一样，你要确保 MySQL 账号都有足够强度的密码和适当的权限。因为 MySQL 经常是放在网络上使用的，所以，你也要确保 MySQL 所在主机的安全性：谁能访问到它？如果有人嗅探你的网络数据流，他能得到什么信息？

MySQL 具备了一套非标准的安全和权限系统，它能帮你完成很多特定的任务。它基于一套简单的规则来实现，但是，仍然有很多复杂的例外和特殊案例要处理，因此，会显得有点难以理解。在本章里，我们将先看一下 MySQL 的许可机制是如何工作的，告诉你怎么控制哪些人来访问你的数据。因为在 MySQL 的帮助手册里有完整的关于权限的文档可供查阅，因此，我们在此就只解释那些难以理解的概念，并教你如何去做那些看似难以掌握的普通任务。然后，我们会讲到一些基本的操作系统和网络的安全措施，你可以用它们把“坏家伙”阻挡在数据库之外。最后，我们将讨论一下加密，以及如何让 MySQL 运行在高限制条件的环境下。

12.1 术语

Terminology

在正式开始之前，让我们把一些容易混淆的术语定义一下。在本章里，我们用它们来指代那些特定的事情：

认证

你是谁？MySQL 使用用户名、密码和来源主机来认证你的身份。这是赋予你具体权限的先决条件。

授权

你被允许做些什么事情？例如你要有 SHUTDOWN 权限，才能关闭服务器。在 MySQL 里，授予的是全局性的权限，无法与任何特定对象相关联（例如表或者数据库）。

访问控制

哪些数据允许你查看和/或控制？当你试着读取或更新数据时，MySQL 会检查你是否已被授权查看或更改当前选择的列。跟全局权限不一样的是访问控制应用到特定的对象，例如特定的数据库、表和列。

权限和许可

这两个术语大致上指同一个事情，一个权限或许可就是 MySQL 表示的一项授权或访问许可。

522

12.2 账号的基本知识

MySQL Basics

MySQL 账号跟大多数系统里的不太一样，因为 MySQL 认为登录的来源也是认证的一部分，相反，Unix 登录常常是只验证用户名和密码。换句话说，Unix 账号的主键是用户名，而 MySQL 的主键是用户名和来源（常常是主机名、IP 地址或通配符）的组合。

如同我们看到的那样，与位置相关的账号会给原本简单的系统增加复杂性。用户 joe 从 *joe.example.com* 登录到系统的情况会跟他从 *sally.example.com* 登录到系统的不一样。从 MySQL 角度来看，他们是完全不同的用户，有着不同的密码和权限；从另一方面来讲，他们又是相同的用户。具体要看你是怎么配置账号的。

12.2.1 权限

Privileges

MySQL 使用你的帐户信息（用户名、密码和位置）来验证你的身份。一旦验证通过，它就要决定你被允许做的事情有哪些。这时，它就会向你的**权限系统**咨询，权限通常是在执行一条被允许的 SQL 查询语句后取得的。例如，你需要有权限表的 SELECT 权限，从里面取出权限数据。

权限也分为两种：与对象(例如表、数据库和视图)相关的和与对象无关的。**对象指定权限**赋予你访问特定对象的权限。例如，它们可以控制你是否能够从表里获取数据、是否可以更改表结构、是否可以在数据库里创建视图、是否能够创建触发器等。MySQL 5.0 及更新的版本有许多额外的对象相关权限，因为它引入了视图、存储过程及其他新的功能。

另外一方面，**全局权限**可以让你执行一些功能，例如关闭服务器、执行 FLUSH 命令、执行各种 SHOW 命令和查看其他用户的查询情况；基于对象的权限能让你操作服务器里的内容（虽然两者之间的差异并非像定义的那样严格）。每一个全局权限都有着联系广泛的安全含义，所以，将它们授予用户时要极其谨慎。

523 MySQL 的权限是布尔型的：一个权限要么被赋予，要么没有。不像其他数据库系统，MySQL 没有明确拒绝权限的概念。撤销一项权限不是去禁止用户执行某个操作，而是把权限从用户那里移除了——如果原来有这项权限的话。

MySQL 的权限也是层次化的，但是有一点别扭。我们将会说明这一点的。

12.2.2 授权表

The Grant Tables

MySQL 使用了一整套的**授权表**来保存用户和他们的权限。这些表都是 mysql 数据库里的普通的 MyISAM 表(注 1)。将安全信息存储在授权表里意义重大，但是这也意味如果服务器配置不当，任何用户都能通过调整这些数据来修改安全设置了。

MySQL 的授权表是整个安全系统的核心。MySQL 提供给你 GRANT、REVOKE 和 DROP USER 的权限（关于这一点，我们稍后继续讨论），使你几乎可以完全控制所有的安全设置。然而，操作授权表一般就意味着执行某些

注 1：它们必须使用 MyISAM 表，千万不要把它们存储引擎改成其他类别的。

任务。举例来说，在旧版本的 MySQL 里，想要彻底删除一个用户的唯一方法是从 `user` 表里 `DELETE` 然后执行 `FLUSH PRIVILEGES`。

我们不建议你直接修改授权表，但是，你仍然应该知道它们是怎么工作的，这样就可以调试那些意外行为了。我们鼓励你使用 `DESCRIBE` 或 `SHOW CREATE TABLE` 去检查授权表的结构，特别是当你刚使用 `GRANT` 和 `REVOKE` 修改了权限之后。跟仅仅使用它们相比，阅读它们会让你获得更多的知识。

以下是所有的授权表，显示的次序就是当 MySQL 检查一个验证通过的用户是否允许使用某个操作时的查询顺序：

User

每一行就是一个用户账号（用户名、主机名和加密后的密码）以及用户的全部权限。MySQL5.0 又增加一个可选的用户限制项，例如该用户可以保持的最大连接数。

Db

每一行包含了某个用户在数据库级权限。

Host

每一行包含了用户从指定主机登录过来时它在一个数据库里的所有权限。当检查数据库层面的访问时，这些条目会与 `db` 表里的条目“合并”起来使用。虽然它是作为授权表罗列出来的，但是无法使用 `GRANT`、`REVOKE` 等命令修改这个主机表，你只能手动增加和删除其中的条目。

我们建议你不要动用这张表，以防止造成 MySQL 的维护问题和各种怪异行为。

tables_priv

每一行包含了指定用户和表的表级别上的权限，也包括了视图的权限在内。

columns_priv

每一行包含了指定用户和列的列级别上的权限。

procs_priv (MySQL 5.0 里新引入的)

每一行包含了指定用户和存储程序（存储过程和函数）的权限。

12.2.3 如何检查权限

How MySQL Checks Privileges

MySQL 在授权表里检查权限的次序就是我们在上一节里列出的那样。服务器会在找到匹配的功能授权后停止检查。这就是说，如果它在 `db` 表里找到了与当前要使用的权限相匹配的条目时，就不会再去查询 `process_priv` 表。图 12-1 描述了这个过程。

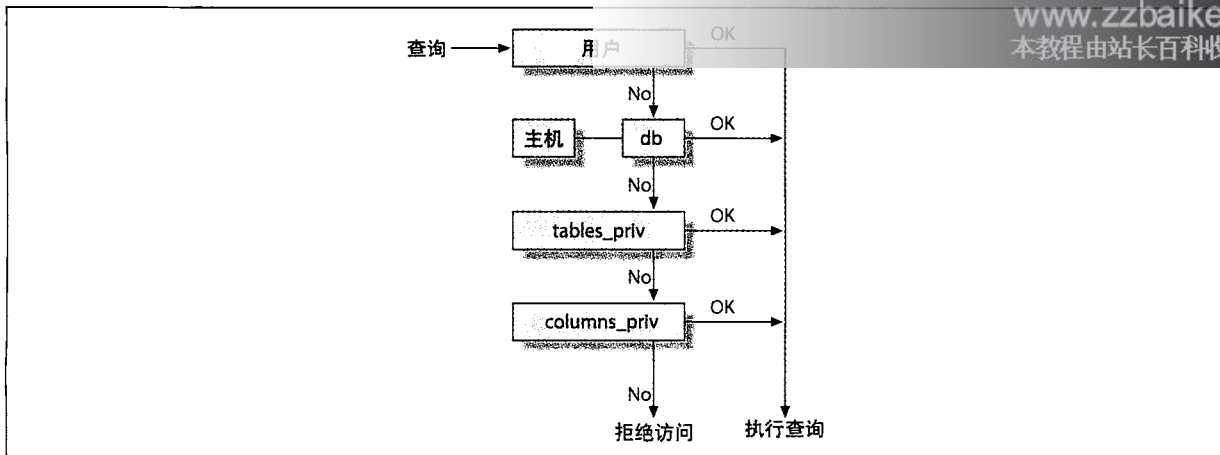


图 12-1: MySQL 是如何检查权限的

525

MySQL 相当于是在缓存的授权表里做 SELECT 操作来决定采用哪个权限。这个虚拟语句里的 WHERE 子句包含着每个表里的主键字段。有些列允许模式匹配，它们中的大部分在有特殊的列值时，还具有“神奇的”含义，例如当它们为空的时候。更多的细节可以查阅 MySQL 使用手册。

你大概会花费大量时间在学习授权表和它们的工作方式上，那些知识可能在偶然间唾手可得。但是，我们不建议你这么去做，除非是必须，与此相比，你更应该阅读下一节内容。只有当你觉得无法（或者说是因为它太复杂而不能够）用 GRANT 和 REVOKE 构建安全系统时，深入钻研授权表才是有价值的。

12.2.4 增加、删除和查看授权

我们建议在 MySQL 里增加用户账号、增加和删除权限都要通过 GRANT 和 REVOKE 命令来做，这两个命令在 MySQL 使用手册里都有完备的文档可供参考。它们提供了一个简单的语法，可做许多种变换，却无须知道底下的授权表和它们不同的匹配规则。你可以使用 GRANT 来增加新用户账号，或增加一项权限；但 REVOKE 只能删除权限，无法删除账号。想删除用户账号，你只能用 DROP USER 来做。

你可以使用 SHOW GRANTS 来查看一个用户的授权情况，其结果的显示内容就是根据该用户当前的权限重新创建用户时所用的语句。例如，下面是默认安装的 Debian 系统里 root 登录后的显示：

```

mysql> SHOW GRANTS;
+-----+
| Grants for root@localhost |
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+-----+

```

这个语句显示了默认情况下该用户所得到的授权，因此，这也是查看你的登录名和权限的捷径。这里显示的用户拥有了所有的权限，但是没有显示密码，这意味着不用输入密码，这个账户也可以登录。（注 2）这是非常不安全的！当你刚安装完一个全新的 MySQL 后，检查这些安全设置就是你首先要做的事情之一。

注 2：唯一可以减轻问题严重性的因素就是这个用户无法从其他任何一台主机上登录到系统，但是，这不能作为一项安全设置来考量。

如果你想看到其他用户的授权信息，你需要给这个特定的用户指定好用户名和主机名。举例来说，在 Debian 系统里，user 表里有下面这样一些条目：

| user | host |
|------------------|-----------|
| repl | % |
| root | 127.0.0.1 |
| root | kanga |
| debian-sys-maint | localhost |
| root | localhost |

526

要注意到这里有三个 root 账号！如果你想查看其中特定一个的授权情况，就必须指定它的用户名和主机名。默认的主机名是%，所以，遗漏主机名会引发一个错误：

```
mysql> SHOW GRANTS FOR root;
ERROR 1141 (42000): There is no such grant defined for user 'root' on host '%'
```

如果你要给一个用户授权而不指定主机名，可以给该用户授权一个@%'（即任何主机）。

没有什么能阻止你使用标准的 INSERT、UPDATE、DELETE 命令直接修改授权表，但是，坚持使用 GRANT 和 REVOKE 命令可以把你跟这些表格变化区分。若直接修改这些表，容易犯非常严重的错误。例如，MySQL 不会阻止你把它无法解释的数据写入这些表里去。GRANT 和 REVOKE 就是被推荐用于权限管理的最好途径，而且一直会这样。

如果你决定要手工修改这些授权表，而不是用 GRANT 和 REVOKE 命令，你必须通过执行 FLUSH PRIVILEGES 命令告诉 MySQL 你已经这么做了，MySQL 就会从表里重新获取和缓存账户、权限信息。你使用 INSERT 或其他普通命令在授权表里做的更改，要等到你重启服务器或者运行 FLUSH PRIVILEGES 之后，才会被 MySQL 知晓。

12.2.5 设置 MySQL 权限

Let's Do Another Privilege

让我们来看一下如何为一个虚拟的组织——widgets.example.com 创建一系列适当的用户和权限。我们假定你已经登录到了一个刚刚安装完成的 MySQL 实例上，而且已经使用 DROP USER 把所有默认账户都删除了。你要检查一下 mysql.user 表看看是否已经做了这些。

MySQL 不支持你在别的数据服务器上熟识的角色和分组。MySQL 只支持用户。

在这里，基本的习惯用法就是把三条命令合在一起使用：

```
GRANT [privileges] ON [objects] TO [user];
GRANT [privileges] ON [objects] TO [user] IDENTIFIED BY [password];
REVOKE [privileges] ON [objects] FROM [user];
```

以下是你要创建的各种用户类型以及要设置的各种权限的概况：

密码安全

我们特意使用了一个可爱的“p4ssword”来达到说明的目的，但是，在实际情况下，这不是一个好的密码。你不能因为 MySQL 存储的不是明文，而不注意密码的复杂度。任何能连接到你 MySQL 服务器的人都可以启动一次强力攻击来试着找出你的密码，而 MySQL 这边不像其他类型的密码系统（例如 Unix 密码）那样，

527

能够有许多种常用的手段来检测并防止此类攻击的发生。MySQL 也没有提供任何一种方法让用户使用一种好的密码标准。你不能使用 libcrack 去连接 MySQL，并要求密码都要符合它的标准，不管这种想法有多酷。外面有许多好的工具和网站能帮你和你的用户生成足够强的密码——我们建议你采用其中的一种。

系统管理员账号

在许多大型组织里，你有两个重要的管理员角色。system administrators 管理的是“物理”服务器，包括操作系统、Unix 登录账号等等；database administrators 专注于数据库服务器的管理。你怎么分配管理账号要取决于你的需要——你可能想让事情简单一些，要求任何要做管理任务的人登录到 MySQL 后都是超级用户；或者是你可能想为每一个需要管理访问的人分别创建一个单独的账号。我们也尽量能简单地开始，只创建了一个名叫 root（像传统的 Unix 里的超级用户一样）的超级权限用户：

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost'  
-> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

数据库管理员账号

当有超过一个 DBA 需要访问 MySQL 时，为他们分别创建一个单独的账号要好过让他们共用一个 root 账号。这样的设置提供了更强的责任性和可审查性：

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'john'@'localhost'  
-> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

每个雇员一个账号

widgets.example.com 里的典型雇员就是客户服务代表，负责登记来自电话的订单、更新现有的订单等等。就让我们假设 Tera 就是一个客户服务代表，她登录到一个客户应用里，后者会把 Tera 的用户名和密码传送到 MySQL 服务器里验证和授权。创建 Tera 账号的命令如下：

```
mysql> GRANT INSERT,UPDATE PRIVILEGES ON widgets.orders  
-> TO 'tera'@'%.widgets.example.com'  
-> IDENTIFIED BY 'p4ssword';
```

Tera 必须提供她的用户名和密码给应用程序，应用程序通过验证后就能让她增加新订单或者更新现有订单，但是，她无法再返回去删除自己刚才创建的订单了。在这个配置里，每一个 widgets.example.com 的雇员都是使用他们各自的数据访问权限登录到系统里输入订单。他们不是共享一个“应用程序账号”，而是每个雇员都用自己的用户名和权限对订单做各种操作。

模拟组

MySQL 没有提供用户组或角色这样的功能，但这些在其他数据服务器上是很普遍的。有时，创建一个账号，将它的名称命名在一个特定的雇员或应用角色名称例如 custserv 或者 analyst 后面，这非常有意义，但在这里我们不打算这么做。

日志，只写访问

把 MySQL 用作各种数据类型的日志的后台也很常见。不管你是否正在用 Apache 记录 MySQL 上的每一个请求，或者当你的门铃响起的时候，你就一直在追踪，日志就是一个只有写入的应用，它可能只需要对一个单独的数据库或表进行写入。你可以像下面这样创建一个日志账号：

```
mysql> GRANT INSERT ON logs.* TO 'logger'@'%.widgets.example.com'  
-> IDENTIFIED BY 'p4ssword';
```

这条命令会在 user 表里新增加一行，但是，我们没有指定给它全局权限，因此，在这行里的所有权限列都是 N。添加这一行的唯一用途就是让用户在提供了密码的前提下从任何一个主机都可以登录进来。因为我们针对某个数据库指定了一项权限，所以就会有一些有趣的数据保存到 db 表里。在 db 表里，这一行的各个列也都是 N，除 the Insert_priv 列是 Y。

备份

一个备份用户会通过 mysqldump 来做备份，他一般只需要用到 SELECT 和 LOCK TABLES 两项权限。如果用户是使用带-tab 选项的 mysqldump 来做 tab 分界符文件的导出，或者是用 SELECT INTO OUTFILE，那么，你还要给这个用户赋予一个 FILE 权限。下面是一个备份用户的示例，他只能在本地主机上登录：

```
mysql> GRANT SELECT, LOCK TABLES, FILE ON *.* TO 'backup'@'localhost'  
-> IDENTIFIED BY 'p4ssword';
```

为了保证许多备份操作的一致性，这里还会用到 FLUSH TABLES WITH READ LOCK，这就需要 RELOAD 的权限了。这个权限也允许用户做其他几个常见的操作，例如 FLUSH LOGS。

操作和监控

可能有时你会想让某人或某些东西（例如一个用户，或者网络操作中心里一些监控软件，或者 NOC）来访问 MySQL 服务器，帮你维护系统或者修复故障。这个用户账号要求能够连到数据库，可以使用 KILL 和 SHOW 命令，还可以关闭服务器。因为这些功能都非常强大，所以，需要将它限制在一台单独的主机上使用。这就意味着即使是未授权的用户掌握了密码，他也必须在 NOC 里做这些操作。以下这条语句就可以实现这些要求：

```
mysql> GRANT PROCESS, SHUTDOWN on *.*  
-> TO 'noc'@'monitorserver.noc.widgets.example.com'  
-> IDENTIFIED BY 'p4ssword';
```

你可能还要把 SUPER 赋予他，这可以让他能执行 SHOW INNODB STATUS。

12.2.6 权限在 MySQL 4.1 里的变化

Changes in MySQL 4.1

MySQL 4.1 里引入了一个新的更加安全的密码散列方式，但是，你仍然可以在新版本里使用旧的密码散列方式（甚至在 MySQL 5.0 及更新的版本里继续使用）。我们建议您还是少用旧的密码散列方式，因为它很容易被破解。如果你注重安全问题的话，就要使用 MySQL 4.1 或更新的版本，并采用新的散列方式。

有一些 GNU/Linux 分发版为了能和老的客户端程序兼容，就将 MySQL 默认地配置为使用旧的密码散列方式。你要检查一下你的默认配置里是否也选择了旧的密码方式。如果你希望你的 MySQL 服务器能拒绝任何使用不安全的旧风格密码的尝试，你可以在服务器的配置文件里设置 secure_auth 选项。另外，还有一个类似的选项可用于客户端程序，它能阻止它们使用旧格式发送密码给服务器——哪怕是服务器要求它这么做。这是个好主意，因为旧格式很容易被嗅探和破解。

新风格的密码是以星号开始的，这样你就能用目测的方法把它们区别开来。在大多数情况下，从旧 MySQL 版本升级过来的用户账号都能正常验证。但是，当以前的 MySQL 4.1 客户端程序试图用新格式的密码连接到更新的 MySQL 服务器时，将无法实现。为了修补这个问题，你可以手工使用 OLD_PASSWORD() 把账号的密码改回

到旧的散列方式，或者升级客户端程序的 MySQL 客户端库。

12.2.7 权限在 MySQL 5.0 里的变化

Privilege Changes in MySQL 5.0

TopSage.com

MySQL 5.0 增加了一些新的权限，也略微改变了一些现有的系统安全行为。这一节对这些变化做一个概述。在你升级到任何新版的 MySQL 之前，应该阅读一下它的发布声明，了解一下其中增加了什么新内容、更改了哪些东西。

530 存储程序

我们在第 5 章里讲到过 MySQL 5.0 增加了对存储程序的支持。这些程序能在两种安全上下文里执行：作为定义者（例如那个定义程序的用户）和作为调用者（例如那个调用程序的用户）。

存储程序一般被用作代理人，把特定的权限赋予那些无法直接访问表的用户。常用的方法是创建一个授权用户，然后，作为定义者创建一些列的存储程序，并给予它们 SQL SECURITY DEFINER 特性。表 12-1 说明了一个存储过程是如何允许用户使用另外一个用户的权限来执行语句的。

表 12-1：一个存储过程里的安全上下文语句

| 调用过程的用户 | 安全上下文语句 | |
|-----------------|-------------------------|--|
| | 使用 SQL SECURITY INVOKER | 使用 SQL SECURITY DEFINER 和 DEFINER=LegalStaff |
| LegalStaff | LegalStaff | LegalStaff |
| HumanResources | HumanResources | LegalStaff |
| CustomerService | CustomerService | LegalStaff |

采用这个方法使你能根据用户的具体身份授予或取消他对表的访问权，同时，当你不希望这个用户直接访问表的时候，就可以授予他在表里执行特定的操作——这些操作就封装在存储过程里。举例来说，你有一些私密的法律数据（例如与某外部当事人签订的合约的履行情况）存放在一组表里，它们只对法务人员可见，但是你的客户服务代表需要能够更新这些表里的某一个特定的列。这时，你要取消除了法律人员之外的其他人在这些表里的 SELECT 权限，然后，编写一个存储过程，让其他人能更新他们需要的字段，并使用 SQL SECURITY DEFINER 让存储过程以 LegalStaff 的权限运行。这跟在 Unix 风格的操作系统里使用的 SUID 权限非常相像。

存储程序的命名空间是每个数据库一个，所以，你可以同时有 db1.func_1() 和 db2.func_1()，而不会引起任何命名冲突。

MySQL 会检查存储程序里每条语句的所需权限，执行该程序的权限不会覆盖到其中的语句里，所以，核准这些语句的执行权限要么是根据定义者，要么是根据调用者，具体要看你创建程序时是使用了 SQL SECURITY DEFINER 还是 SQL SECURITY INVOKER。

触发器

MySQL 5.0 还增加了触发器的支持。如果触发器没有用 SQL SECURITY DEFINER 特性来定义，那么在执行的时候就需要专用的权限。这就会产生令人迷惑的影响，例如在一个表里运行 UPDATE、INSERT 或 DELETE 时，会

531

得到下面这样的一个错误信息：

```
mysql> INSERT INTO ...;  
ERROR 1142 (42000): Access denied; you need the SUPER privilege for this operation
```

如果触发器不是使用 SQL SECURITY DEFINER 特性创建的，那么，往表里插入数据的用户必须具有 SUPER 权限才能执行这个触发器，这也是为什么刚才那段错误信息看起来像是在说插入表的操作需要 SUPER 权限。(MySQL 5.1 包含了一个 TRIGGER 权限，它会使错误信息少一些迷惑性。)

MySQL 会检查触发器里语句的所需权限，如同它在存储程序上做的那样。

视图

就像存储过程和触发器一样，你可以使用定义者或调用者的权限来执行视图。定义者权限可以让你访问到视图，但是不能访问视图下面的各个表。

这可以让你实现行级安全，但是也可以限制对列的访问。我们相信这是比使用 GRANT 来指定行级权限更好的解决方案，也更易于维护。如果把视图放在一个单独的数据库里，你可以简单地赋予数据库级的权限给用户就行了，用不着去维护单独的表和视图上的权限。图 12-2 显示了用 GRANT 赋予特定行的访问权和把相关列放入一个视图两种方法之间的差异。

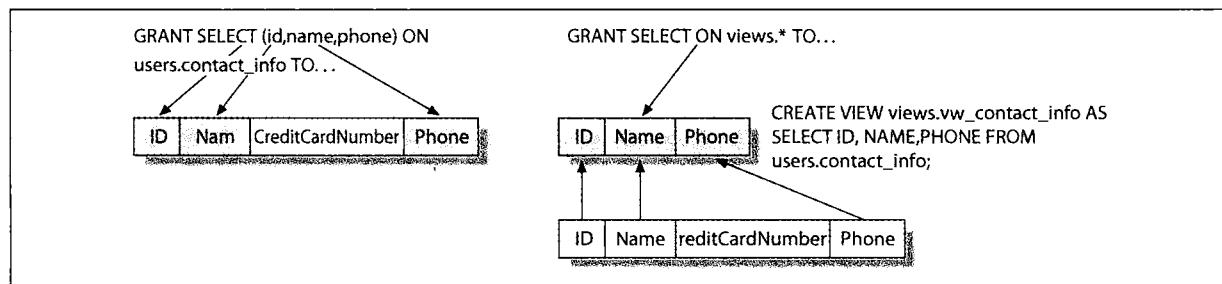


图 12-2：通过定义视图简化对特定列的访问

在图 12-2 的左边，DBA 运行了一条 GRANT 语句，给那些单独行赋予了粒度适当的访问权限。这会减慢所有数据库的访问速度，并要为每一个要求列级许可核查的表单独执行 GRANT 语句。

在图 12-2 的右边，DBA 创建了一个名叫 views 的数据库，用来存放那一系列的视图。然后，他又创建了一个视图，其中包含的是那个他希望用户拥有合适权限粒度的表的各个列。任何这样的视图都可以被存放在 views 数据库里，而单独的 GRANT 语句会使访问授权更有效率。

INFORMATION_SCHEMA 表的权限

官方的 SQL 标准里定义了一整套视图，它们作为一个整体，以名为 INFORMATION_SCHEMA 表而为大家所知，它们可以为你提供数据库、表以及数据库服务器其他部分的信息，MySQL 尽可能地沿袭这些标准。作为努力的结果，服务器能够自动管理这些表的权限，最好不要显式地定义它们的权限。如果某个用户没有适当的权限去访问这些表里的行或者值，MySQL 也就不会显示行给他看，只返回 NULL 形式的结果值。例如，一个用户不能看到 INFORMATION_SCHEMA.TABLES 视图里的表，除非他在这些表上有一定的权限。这也类似于 MySQL 的 SHOW TABLES 的行为：MySQL 不会显示出该用户无权查看的表。

12.2.8 权限与性能

Privileges and Performance

权限看上去跟性能没什么太大关系，但是，它们在某些环境下确实会引发性能问题。其中有一些事情需要考虑：

权限太多

如果在授权表里，权限条目过多，它们的开销会很显著。当数据库检查一个用户是否有权限执行当前语句时，它要核对每一条添加进来的权限条目。同时，权限的存储也会消耗内存。

权限的粒度过细

在 MySQL 中如果权限层级里的每一层（用户、数据库、主机、表以及列）都做权限核对显然过于昂贵。核对全局权限相对来说比较快速，但是，即使你仅仅定义一个列的权限，服务器就会潜在地用全局的、数据库的、表的和列的权限去检查每一个查询（回想一下第 524 页的“如何检查权限”，服务器从最高层次开始，持续查找，直到找到跟当前所需权限匹配的授权）。

列的权限和查询缓冲区

在本书写作时，带有列权限的查询访问一个表时，还无法受益于查询缓冲区。像上一节里我们讨论过的那样，我们建议用视图代替列权限，从而避免这里以及其他由列权限带来的问题。

在默认情况下，MySQL 认证用户时，会做前向和逆向的 DNS 查询。在你的 my.cnf 表增加一项 `skip_name_resolve` 就可以关闭这个查询。这对于安全和性能两者都很有好处，因为它提高了连接速度，减轻了对 DNS 服务器的依赖，降低了遭受拒绝式服务攻击的可能性。

这种改变的副作用是它阻止了你在 Host 列里用主机名定义用户。这类用户定义会停止工作，作为替换方案，你必须使用 IP 地址（你仍然可以使用通配符，例如 192.%），也可以使用一个特定的值——localhost，哪怕 `skip_name_resolve` 选项已经开启了。

12.2.9 常见问题和解决方案

Common Problems and Solutions

MySQL 帮助手册对权限有着详尽的说明，因此，我们把上一节的篇幅缩减掉了一些，用在本节里来讨论常见的需求、小技巧 and 意外行为，你可以把它用作快速参考或问题排除手册。接下来，我们就开始描述常被问及的问题、常见的任务和我们遇到过的困境。

连接时出错

许多因为连接 MySQL 服务器时遇到麻烦的用户把邮件列表、论坛和 IRC 频道塞得人满为患。对于这些问题有多种原因可以解释，从因为 my.cnf 里定义了 `skip_networking` 而导致 TCP 连接失败，到 `bind_address` 把一个不匹配的 IP 地址设在了服务器上，再到错误的 GRANT 语句，一直到防火墙为止。在此，我们无法讲到所有的原因，但是，在 MySQL 帮助手册里有一节是专门讲这个主题的。

通过 localhost 还是 127.0.0.1 来连接

主机名 localhost 一般就是 IP 地址 127.0.0.1 的别名。但是，在 MySQLd 默认行为里，两者的处理方式有轻微的差别。当你把 localhost 作为连接的一项参数时，MySQL 会默认地使用 Unix socket（注 3）去连接，而不是你所期望的 TCP/IP。因而，下面这行命令就是通过一个 Unix socket 连接的：

```
$ mysql --host=localhost
```

这是一个让人有点遗憾的设计决策，因为它不像人们期望的那样运作，要把它改回来也为时已晚，因为这会打破与旧的应用和客户端库的兼容性。如果你想通过 TCP/IP 去连到你正在运行的机器，你有两种选择：指定一个 IP 地址而不是主机名，或者明确地指定通信协议。以下两行命令里的任何一个都能通过 TCP/IP 去连接：

```
$ mysql --host=127.0.0.1
$ mysql --host=localhost --protocol=tcp
```

在一个相关的记录里，如果在建立 SSH 隧道时，试着用 localhost 去连接 TCP 转发端口，你会发现它不会工作。因为必须使用 TCP 去连接端口，所以，你一定要用 IP 地址 127.0.0.1 来代替。我们会在本章的后面讨论 SSH 隧道。

主机名在另一方面也是特殊的：MySQL 不会拿 localhost 作通配符匹配。换句话说，同时指定 user@'%'和 user@localhost 不是多余的。

安全使用临时表

MySQL 对于临时表没有设置特别的权限，除了 CREATE TEMPORARY TABLE。一个临时表一旦被创建，就被加入该用户的标准的表级权限。这意味着一个用户可能可以创建临时表，却没权限增加更多的列、修改表和添加索引（甚至无法 SELECT）。然而，如果把这些权限授予该用户，又会使他危及其他真正的表——这可不是你想要的结果。

这个问题的解决办法是除了在特地为临时表准备的数据库里，其他地方都不接受用户的这些权限：

```
mysql> CREATE DATABASE temp;
mysql> GRANT SELECT, INSERT, UPDATE, DELETE, DROP, ALTER, INDEX,
-> CREATE TEMPORARY TABLES ON temp.* TO analyst@'%';
```

不允许没有密码的访问

MySQL 本身是允许没有密码访问的。一个没有密码的账号在 user 表里的就是 password 列为空字符串的一行记录。你可以通过不带 IDENTIFIED BY 子句的 GRANT 命令来创建这样一个账号。

你无法彻底禁止 MySQL 的无密码访问，但是，如果你能控制用户连接发起的那台机器，你就可以往它的 my.cnf 文件的 [client] 节里加入下面这么一行条目：

```
password
```

这就能让默认读取该配置文件的程序（这包括了所有 MySQL 分发的程序，除非它们已被配置成别的方式。）一直提示用户名和密码。在 MySQL 5.0 和更新的版本里，你可以把服务器的 SQL 模式设为 NO_AUTO_CREATE_USER 来防止创建没有密码的用户，但是，对于一个执着的用户，他还是可能通过变通的方法来达到此目的。

注 3：localhost 在 Windows 上不显得特殊，但也可以说有点特殊，因为它意味着将通过命名管道去连接。

要记住，在 `mysql.user` 表里用空字符串做密码的用户就是没有密码的用户，没有一个用户会

关闭匿名用户

535

MySQL 也是允许匿名用户的：授权表里 `User` 列为空字符串的条目就是匿名用户使用的权限。对于这些条目要小心，因为 `SHOW GRANTS` 不会显示出这些权限来，我们认为最好还是移除这些条目。你可以运行 MySQL 提供的 `mysql_secure_installation` 程序来做这个。

记得给主机名单单独加引号

用户很容易忘记用户名和主机名是要分别加引号的。下面这条命令无法做到它看上去可以做的事情：

```
mysql> GRANT USAGE ON *.* TO 'fred@%';
```

它看起来是创建一个名叫 `fred` 的账号，他可以从任何地址登录进来，但是，事实上，它是创建了一个名叫 `fred@%` 的用户。正确的语法格式要像下面这样（注意，用户和主机名要分开来加引号）：

```
mysql> GRANT USAGE ON *.* TO 'fred'@'%';
```

不要重用用户名

MySQL 认为相同用户名但是不同主机名就是完全不同的用户。这看起来有助于你对同一用户从不同地方来连接时赋予完全不同的权限，但是，根据我们的经验，这样的做法并不是个好主意，其中潜在的混乱和问题远远超过它能带来的好处。跟他们连接后能做什么相比，确保用户名的唯一性，并用主机名来限制用户能从哪里连接进来，会更加简单。举例来说，你可能已经决定所有连接只能来自本地机器，或者本地网络，或者是一个指定的子网。这是一个很合理的安全防范措施，尽管防火墙是更加安全的连接限制方式（我们过一会儿将进一步讨论）。

MySQL 有着许多兼容性措施，但是，这并不意味着你的生活会更加容易，我们认为最好还是让它保持简单。

赋予 SELECT 权限，允许执行 SHOW CREATE TABLE

给用户授予 `SELECT` 权限，让他能执行 `SHOW CREATE TABLE`，显示出重建一个表的 SQL 语句。这通常情况下是个好办法，但是有时也会泄露出敏感信息。最明显的例子就是 MySQL 5.0 里的 `Federated` 表：用户能够看到数据引擎连接到远程服务器时使用的用户名和密码。（MySQL 5.1 增加了一个单独的机制，用于管理 `Federated` 表的远程连接。）

不要授予 mysql 数据库的权限

536

如果你把 `mysql` 数据库的权限授予了用户，那么，用户就能自己提升自己的权限、查看其他用户的权限（这也就打开了猜密码攻击的方便之门）、甚至是重命名或更改 MySQL 运行时所需要的表。因此，根本不要把这些表的任何访问权授予普通用户——哪怕是只读访问，这意味着下面这行命令就是个坏主意，因为它授予了全局的权限：

```
mysql> GRANT ... ON *.* ...;
```

如果一个用户拥有更新 mysql 数据库里所有表的许可，那他也应该有 GRANT 的选项。否则，他就无法删除行来删除权限，而且无法再加回来。本书的作者之一就曾因此意外地删除了系统里的每一个数据库。本教程由站长百科收集整理，得不开 MySQL 服务器，再使用带--skip_grant_tables 选项来重启服务器、还原所有用户。

不要随便授予 SUPER 权限

正如你所期望的那样，SUPER 权限可以让用户做超级用户的操作（例如更改服务器上的只读数据），但是，这里也有一个额外的行为：MySQL 会为有 SUPER 权限的用户保留一条连接，哪怕是服务器已经达到了 max_connections 的极限。这可以让你在服务器无法再接受普通的客户端连接时，还能接入并管理服务器。

你应该尽量避免把 SUPER 权限授予太多的用户，但是，当有一些其他的普通用途（例如清除 master 日志）也需要用到它时，就很难把握了。

用通配符数据库授予权限

MySQL 的数据库模式匹配方式让你无法设定“所有数据库都要这样”。这意味着要删除 mysql 数据的权限将是一个枯燥的过程。使用一个良好的命名约定就能够有所帮助：用一个通用的前缀来命名所有数据库，然后就能使用通配符把权限授予所有匹配的数据库。下面就一个例子：

```
mysql> GRANT ... ON `analysis%`.* TO 'analyst' ...;
```

不幸的是，MySQL 并没有真正可用于解决此类问题的方案，虽然使用命名约定也能缓解一些痛苦。值得注意的是在 GRANT 命令里使用时必须给数据库名加上引号来作为一个标识符（带括号的）。

建立共享主机环境也是一种有用的技术。这样的环境可以限制用户使用他的用户名和一个下划线来访问数据。下划线就是一个通配符模式，所以，你必须在 GRANT 语句里将它忽略掉。举例来说，你可以使用下面这样一个命令来为名叫 sunny 的用户建立一个新的主机账号：

```
mysql> GRANT ... ON `sunny\_`.* TO 'sunny' ...;
```

你不要把 SHOW DATATABLES 权限授予一个共享主机环境。这样可以确保用户看不到他无权访问的数据库——就是说让他们知道得越少越好。

取消特定的权限

如果你授予权限是全局化的，那你就无法用非全局化的方式来取消它们：

```
mysql> GRANT SELECT ON *.* TO 'user';
mysql> SHOW GRANTS FOR user;
+-----+
| Grants for user@% |
+-----+
| GRANT SELECT ON *.* TO 'user'@'%' |
+-----+
mysql> REVOKE SELECT ON sakila.film FROM user;
ERROR 1147 (42000): There is no such grant defined for user 'user' on host '%' on table 'film'
```

这项权限是被全局化授予的，也就只能全局化地取消。如果试着在一个特定表上取消它，那 MySQL 就会产生一个错误：没有表级权限匹配这个指定的标准。

用户甚至在 REVOKE 之后还能连接到数据库

假定你取消了一个用户的所有权限：

```
mysql> REVOKE ALL PRIVILEGES ON...;
```

这个用户还是可以连接进来的，因为 REVOKE 没有删除用户账号，它只删除了权限。你必须使用 DROP USER 彻底删除他的账号（或者，在老版本的 MySQL 里，从 user 表里删除一行）。

如果你仅仅取消了权限，SHOW GRANTS 会显示出该用户仍然具有 USAGE 权限。你无法取消这个权限，因为这个就是“没有权限”的同义词，仅仅意味着他还是能连接到 MySQL。

当无法授予或取消一项权限时

除了 GRANT 选项之外，你必须拥有要授予或取消的那个权限。这项安全措施可以防止用户相互之间逐级提高自己的权限。如果你试着要取消所有权限，那你必须要有 CREATE USER 权限。

不可见的权限

SHOW GRANT 并不能真正显示出一个用户的所有权限：它仅仅是显示出那些明确授予该用户的权限。一个用户也可以拥有别的许可，可能是由于这些许可被授权给了匿名用户。举例来说，默认的 MySQL 安装程序会把 test 数据库和以 test_开头的其他数据库的权限授予每一个用户。首先，让我们以 root 的名义登录，然后创建一个没有权限的用户：

```
mysql> GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY 'p4ssword';
mysql> SHOW GRANTS FOR restricted;
+-----+
| Grants for restricted@% |
+-----+
| GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY |
| PASSWORD '*544F2E9C6390E7D5A5E0A508679188BBF7467B57' |
+-----+
```

看仔细了，这个用户看上去是只能连接到数据库，不能做其他任何事情。但这还不是整个故事的全部。为了证明这一点，我们只要以这个用户的身份登录，然后运行 SHOW DATABASES 就知道了：

```
$ mysql -u restricted -pp4ssword
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| test |
+-----+
```

这个服务器里也包含了一个 Sakila 示例数据库，这里没有显示出来的原因是该用户没有 SHOW DATABASES 权限，但是 test 已经在列表里了。事实上，如同你会在下面看到的一样，这个新用户拥有这个数据库以及表的所有权限：

```
mysql> USE test;
mysql> SHOW TABLES;
+-----+
| Tables_in_test |
```

```
+-----+
| heartbeat |
+-----+
mysql> SELECT * FROM heartbeat;
+-----+
| id | ts |
+-----+
| 1 | 2007-10-28 21:31:08 |
+-----+
```

这个用户账号只是不能从表里读数据，但是他拥有绝大多数别的权限。实际上，他甚至可以创建一个新的数据库：

```
mysql> CREATE DATABASE test_muah_ha_ha;
Query OK, 1 row affected (0.01 sec)
```

这个问题的罪魁祸首是 mysql.db 里的两行记录：

```
mysql> SELECT * FROM mysql.db\G
***** 1. row *****
          Host: %
          Db: test
          User:
          Select_priv: Y
... 以下省略 ...
***** 2. row *****
          Host: %
          Db: test\_%
          User:
          Select_priv: Y
... 以下省略...
```

539

请注意 User 列是空的，这意味着匿名用户——实际上包括了所有用户拥有这些权限，哪怕是它们没有出现在 SHOW GRANT（注 4）的输出结果里。我讲这个故事的寓意是 SHOW GRANT 无法把所有的一切都显示给你看。有时你仍然要知道怎么去解读授权表里的那些信息。

这也不是用户权限里唯一的奇怪行为。因为主机名和数据库匹配时，首先匹配最具体的名称，不具体的匹配项的权限会被隐藏起来，哪怕它们的名字更加随意。

设想一下下面这样的场景：相比于采纳我们先前建议的命名约定，最小权限原则会使一个懒惰的 DBA 决定采用的逆向方法。他把所有权限都授予一个用户，然后在 mysql 数据库里，把不希望给他的权限都用权限列为 N 的行将它覆盖掉。

```
mysql> GRANT USAGE ON *.* TO 'gotcha'@'%' IDENTIFIED BY 'p4ssword';
mysql> GRANT ALL PRIVILEGES ON `%.*` TO 'gotcha'@'%';
mysql> INSERT INTO mysql.db(Host, DB, User) VALUES('%', 'mysql', 'gotcha');
mysql> FLUSH PRIVILEGES;
```

因为在 mysql 模式里，更具体的名称优先于%模式，这个懒惰的 DBA 就会考虑：那用户不能拥有像 mysql 数据库里的 SELECT 那样的权限。下面这行命令就说明这个情况：

```
mysql> SELECT * FROM mysql.user;
ERROR 1142 (42000): SELECT command denied to user 'gotcha'@'localhost' for table 'user'
```

注 4：这个例子说明了 MySQL 权限表里的“神奇”值之一。在 User 列里的空字符串代表了一个匿名用户，也就是当你用不存在的用户名连接到服务器时，MySQL 会采用的验证方式，或者意味着这个权限对任何人都可用。

这么做的问题在于权限方案给任何要在将来修改该用户权限的人设置了一个陷阱。在删除 `mysql` 数据库权限的严谨性。换句话说，删除权限实际上会授予更多权限。方案远非它看上去那样聪明，实际上，这非常危险。它也不会 `SHOW GRANT` 时出现，因此很容易被遗漏。

你也可以按照这样的次序，用主机名匹配来玩类似的游戏。举例来说，如果你想允许那个 `gotcha` 用户从除了某特定主机名称以外的任何主机连接到服务器，你不能采用“否定主机名模式”来达到这个目的，因为 MySQL 上没这种东西。唯一的途径就是用同样的用户名创建一个用户，然后指定一个要阻止的主机名和一个假密码：

```
mysql> GRANT USAGE ON *.* TO 'gotcha'@'denied.com' IDENTIFIED BY 'b0gus';
```

现在，当 `gotcha` 从这个主机连接到 MySQL 时，MySQL 会试着用 `user` 表里的 `gotcha@denied.com` 行来验证他，然后因为密码不匹配而拒绝了他的登录。然而，如果有人认为表里的这个条目是错误的，并把它移除了，或者因为性能原因关闭了主机查找功能，或者认为可能有用户危及到了反向 DNS，那么，这个“解决方法”就会显得非常危险，以上情况中的任何一种都会使这个用户无法用 `gotcha@%` 登录。

我们建议你避免这类隐藏的权限，转而采用“聪明的方式”，例如我们展示过的那些方案，要多使用符合常识的方法，不要试着用你不需要的权限去做任何花哨的东西，并遵循最佳实践的指引，例如最小权限原则。

废弃的权限

当你删除对象时，MySQL 不会去清除那些旧权限。举例来说，让我们假定你运行了下面这样一条命令：

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO analyst;
```

然后，又运行了这条命令：

```
$ mysqladmin drop my_db
```

如果 MySQL 能清除这个表有关的 `GRANT` 那是再好不过了，但是，实际上，那些权限还留在 `db` 表里。如果你随后又用同样的名字创建了一个数据库，而那些权限也仍然存在。这样问题就产生了，你可能都记不起 `analyst` 账号有过哪些权限。

在 MySQL 5.0 及更新的版本里，`INFORMATION_SCHEMA` 表能帮你找出那些陈旧的权限。举例来说，你可以使用一个排除联接查询来找到那些引用不存在的数据库的权限：

```
mysql> SELECT d.Host, d.Db, d.User
-> FROM mysql.db AS d
-> LEFT OUTER JOIN INFORMATION_SCHEMA.SCHEMATA AS s
-> ON s.SCHEMA_NAME LIKE d.Db
-> WHERE s.SCHEMA_NAME IS NULL;
+-----+-----+-----+
| Host | Db       | User |
+-----+-----+-----+
| %    | test\_% |      |
+-----+-----+-----+
```

你可以针对 `INFORMATION_SCHEMA` 的其他表写出类似的查询。在更早的 MySQL 版本里，你必须手工去查找那些陈旧的权限，或者写一个脚本为你做这件事情。

MySQL 可以让你为不存在的数据库创建数据库级的权限，但是，不会让你为不存在的表创建表级的权限。如果你需要这么做，那你就不得不直接往 `mysql.tables_priv` 表里插入行。

12.3 操作系统安全

Improving System Security

如果一个攻击者取得了服务器的 root 权限，那即使最周全的考虑和最安全的授权表都帮不上你的忙了。有了无限制的访问权，用户可以很方便地把你的数据文件复制到另外一台运行着 MySQL 的机器上。（注 5）这样，攻击者就能很快地得到一份跟你的数据库一模一样的副本。

虽然数据偷窃不是安全预防措施所面临的唯一威胁，一个有创造性的攻击者会认为在几个星期甚至几个月里，持续对你的数据作轻微的修改会更加有效。根据你保留备份的时间长短，以及多久后才发现数据损坏，这样的攻击会极具破坏性。

12.3.1 指导方针

Guidelines

在这里讨论的一般性指导方针不是一个关于系统安全的全面指导。如果你对安全非常重视——你本来就应该这样——我们建议你去阅读 Simson Garfinkel 和其他一些人合写的《Practical Unix and Internet Security》（O'Reilly）。也就是说：这里只是一些关于如何维持数据库服务器良好安全状态的意见：

不要用特权账号运行 MySQL

Unix 上的 root 用户和 Windows 上的系统（管理员）用户对系统有完全的控制权。如果有人在 MySQL 上发现一个安全方面的 Bug，而你又是作为特权账号在运行，那么攻击者就能获得对服务器的广泛的访问权。对于这个问题，安装指示里也写得很清楚，但是值得再重复一遍：创建一个单独的账号（一般就是叫 mysql），专门用来运行 MySQL。

让你的操作系统随时保持更新

所有的操作系统厂商（Microsoft, Sun, Red Hat, Novell 等）在有安全相关的更新可用时，都会通知用户。现在，你要找出厂商的邮件列表，并订阅它，同时，还要特别关注 MySQL 本身的安全列表。对于那些跟数据库直接交互的软件，例如 PHP 或 Perl，你也要留意一下它们的更新。

在数据库主机上限制登录

是不是每个基于 MySQL 开发应用程序的人都需要一个服务器上的账号？当然不！只有系统管理员和数据库管理员需要机器上的账号。所有开发人员只需要能够使用 TCP/IP 远程登录到数据库做一些查询就可以了。

将产品与其他任何东西隔离开

把你的生产环境和开发环境、测试环境隔离开来。最好是使用完全不同的物理服务器。生产服务器的安全和访问需求跟开发服务器的那些设置完全不一样，因此从物理上隔绝它们很有意义。这也可以防止错误发生，使管理和维护更加方便。它要求从一开始就要创建一些适当的过程和工具，例如建立在数据库之间传输数据的途径。

审查你的服务器

许多大型组织都有内部审查员，他们能评估一台服务器的安全性，然后提出一些改善建议。如果你不幸联系不到审查员，那你可以雇佣一个安全咨询师来做这些审查工作。

使用最强大的就意味着管用

你可以采用一些技术，例如 chroot、jail、zone；或者用虚拟服务器，以及其他更多的手段来隔离 MySQL。

注 5：记住：MyISAM 数据文件便于跨越操作系统和 CPU 架构（提供同样的 CPU 浮点数格式）。

把你的备份记录放在另外一台服务器上是一项重要的安全措施。如果有人侵入过你的服务器，你可以从干净干净的源开始重装操作系统。一旦完成这个步骤后，你就面临着还原所有数据的任务。你可以将被侵入服务器上的数据和已知的完好的数据备份相比较，试着找出攻击者是怎么攻入的。

12.4 网络安全

Network Security

隔离你的服务器，让它们难以被访问到，这一直是最好的方法，但是，你总会有一台 MySQL 服务器，它需要被不在同一台主机上的客户端访问到。下面例举一些可以用来限制服务器“曝光”的技术。

哪怕是只在你自己单位的内部网络上使用你的服务器，你也要采取措施让你的数据远离窥探的眼睛。毕竟，公司里的一些最严重的安全威胁实际上都是来自内部。

要牢记记住这里提到的信息只是确保 MySQL 服务器受到良好保护的出发点。市面上有许多优秀的网络安全书籍可供参考，它们包括 Elizabeth D. Zwicky 等人写的《Building Internet Firewalls》和 Craig Hunt(也是来自 O'Reilly)写的《TCP/IP Network Administration》。如果你对网络安全问题非常关切，你就帮自己一个忙，读一本关于此主题的书（在读完本书之后）。

如同在操作系统安全里做的那样，让第三方来审查你的网络也很有用，使薄弱点在别人利用之前就把它们找出来。

12.4.1 只有 Localhost 的连接

Localhost-Only Connections

如果你在一个应用里使用了 MySQL，两者位于同一台主机上（通常就是一个小型或中型的网站），那你就有机会禁止任何来自网络的 MySQL 访问了。消除了接受外部连接的需要，就可以减少许多攻击者入侵 MySQL 服务器的途径。

禁止了网络访问也限制了你远程做管理更改(例如增加用户、日志周转等)的能力，因此，你或者需要通过 SSH 登录到 MySQL 服务器，或者安装一个基于 Web 的应用程序，通过它，你可以完成那些更改操作。在一个 Windows 系统上，远程登录的必要条件会有些困难，但是，还有别的远程访问方法可供选择。有一种解决方案就是安装 phpMyAdmin。但是，你要当心，因为众所周知它是有安全隐患的。

skip_networking 选项告诉 MySQL 不要监听任何 TCP socket，但是它仍然会允许 Unix socket 连接。启动不带网络支持的 MySQL 也很简单，只要把下列选项放在 my.cnf 文件的 [mysqld] 里就可以了：

```
[mysqld]
skip_networking
```

skip_networking 选项有一些不方便的副作用：它会阻止你使用诸如 stunnel 这样的工具来做安全的远程连接和复制，它也不让 Java 应用程序连接到数据库（Connector/J 只能通过 TCP/IP 来连接）。有一个变通的方法是像下面这样配置 MySQL：

```
[mysqld]
bind_address=127.0.0.1
```

这就开启了 TCP 连接，但是，只能来自本地机器上，因此就兼顾了安全和便利。有一些流行的数据库已经将此设为默认配置。



提示：一台配置了 `skip_networking` 的 MySQL 从服务器会很有趣。因为它发起连接到主服务器，从服务器仍然可以得到所有数据的更新，但是，因为它不允许有 TCP 连接，你就会有一个更加安全的“备份复制”，它不会被远程污染。你也无法将它用在容错配置里：没有客户端能够连接到它。

12.4.2 防火墙

54

在任何基于网络的服务里，你要只允许认证过的主机才能连接到服务器，这一点是非常重要的。你可以使用 MySQL 的 `GRANT` 命令来限制用户连接过来的主机，但是，多一层保护总归是个不错的主意。使用多重过滤连接的方法，就意味着一个孤立的错误，例如 `GRANT` 命令里的拼写错误，都会使未经认证的主机无法连接到服务器。在网络层上使用防火墙来过滤连接可以给予你额外的安全保障。（注 6）

在许多组织里，网络安全是由其他非开发组的人员来管理的。这有助于进一步减少因为个人的改变而导致服务器暴露的风险。

可供采用的最安全的办法是给一台机器增加防火墙时，先默认地拒绝所有连接，然后再增加规则，给予那些要访问服务器上服务的主机以访问权。对 MySQL 服务器要做的限制是你应该只允许 TCP 的 3306 端口（这是 MySQL 默认的）上的连接，可能还要开通一个远程登录服务，例如 SSH（一般是 TCP 的 22 端口）。

没有默认的路由

要考虑到装备了防火墙的 MySQL 服务器上不能有默认的路由配置。这样的话，即使防火墙的配置被损害，有些人从外部连接到了你的 MySQL，那些数据包也不会发回到他们那里，永远不能离开你的局域网。

让我们假设你的 MySQL 服务器的 IP 是 192.168.1.10，局域网的掩码是 255.255.255.0。在这样一个配置里，任何来自 192.168.0.0/24 的数据包都被认为是“本地”的，因为它可以通过当前网络接口（大概是 `eth0`，或者主机操作系统上类似的东西）直接到达。来自其他地址的网络流量都将不得不定向到一个网关以到达最后的目的地。因为没有配置默认路由，所以，那些数据包都无法找到它们自己的网关，从而无法到达目的地。

如果你必须允许一些选择好的外部主机来访问你那台带防火墙的服务器，那么就给它们添加静态路由。做这个时候你要确保服务器能够响应尽量少的外部主机。

不配置默认路由的措施也不是万无一失的，它更多的是保护你免于因防火墙的配置错误而带来安全威胁，而不是当整个防火墙受到危害的时候。无论如何，它在各个细小方面都是有所帮助的。

注 6：在我们的用意里，防火墙仅仅是一个设备，用于过滤（可能还有路由）流经它的那些网络流量，不管它是一个“真正”的防火墙、路由器，或者是一个老的 486 机器都没关系。

545 12.4.3 在 DMZ 里的 MySQL

对于许多安装要求而言，仅仅给 MySQL 服务器加个防火墙，还是不够安全。如果你的 Web 或者应用服务器之一受到危害了，一个攻击者就能使用这台服务器来直接攻击 MySQL 服务器。一旦攻击者在带有防火墙网络里获得了其中一台电脑的访问权，他也就能够在相对较少的限制条件下（在许多配置里都是这样）访问到网络里的其他服务器。（注 7）

现在就把 MySQL 服务器移到它们自己单独的网段里，使外界无法访问到，从而提高安全性。举例来说，想象一下有一个局域网，其中有若干台 Web 或者应用服务器和一个防火墙。防火墙的后面是一台或多台 MySQL 服务器，它们位于一个不同的物理网段并且是不同的逻辑子网里。应用服务器已经被限制访问 MySQL 服务器：它的所有流量要先通过防火墙——这个你可以用严格的设置来做到。如果有人获得了应用服务器的访问权，但是，防火墙只允许应用服务器通过 MySQL 服务器的 3306 端口进行相互通信，因此，入侵者无法对 MySQL 上运行的其他服务（例如 SSH）发起攻击。

你可能要把应用服务器放在 DMZ 里，或者放在它们自己单独的 DMZ 里。这是不是做得过头了？可能。作为安全问题里总是会遇到的情况，你也需要从方便性上衡量一下安全措施，然而，在着手做的时候，你应该知道其中包含的风险。

12.4.4 连接加密和隧道技术

在任何你需要通过公共的（如 Internet）或向网络嗅探开放的（如无线网络）网络去访问 MySQL 的时候，你都要考虑到某种形式的加密问题。这样一来就可以给那些企图拦截连接和嗅探或发送欺骗数据的人制造更大的困难。

作为一个额外的好处，许多加密算法的结果都是压缩了的数据流，因此，不仅你的数据更安全，而且带宽的使用效率也会提高很多。

虽然我们讨论的焦点是一个访问 MySQL 服务器的客户端，但是，客户端也可以是另外一台 MySQL 服务器。当使用 MySQL 的内建复制时，这就很常见了：每一个从服务器使用跟普通 MySQL 客户端一样的协议连接到主服务器。

546 虚拟私有网络

如果一个公司在很远的地方有两个或更多的办公室，那就可以使用多种技术在办公室之间架设一个虚拟私有网络（VPN）。一种常见的解决方案是每一间办公室里都有一个外部路由器，它会加密办公室之间的网络通信。在这样一种情势下，就没什么好担心的了，无论连接到办公室的网络是公有的还是私有的，所有的流量都会被加密。

VPN 排除了使用一个 MySQL 专用的解决方案的必要性吗？也不一定。当 VPN 必须因为某个原因而关闭时，若能继续保持 MySQL 网络流量的私密性，那是再好不过了。通过配置 MySQL 使它只接受来自 VPN IP 地址的连

注 7：这也不完全是对的。许多现代网络交换机可以让你在一个物理网络里配置多个虚拟局域网（VLAN）。不在同一个 VLAN 里的机器相互之间不能通信。

接就可以解决该问题：如果 VPN 被关闭，MySQL 就将无法被访问到。

MySQL 里的 SSL

在 4.1 版本里，MySQL 对 Secure Sockets Layer (SSL) 有着原生的支持——同样的技术被用于当你在 Amazon.com 上买书的时候，或者在你喜欢的旅游网站订购机票时，保证你的信用卡号码的安全。特别值得一提的是：MySQL 采用的是可免费使用的 yaSSL 库（或者在更老的版本里是 OpenSSL）。

有些二进制版本的 MySQL 在默认情况下没有开启 SSL。为了核对你的服务器，你只要检查一下 `have_openssl` 变量就可以了：

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_openssl  | NO    |
+-----+-----+
```

如果它显示的是 NO，那你就需要编译你自己的 MySQL 服务器，或者换另外一个版本。如果它显示的是 YES，那么，数据库访问安全的所有新层次都会开放给管理员。至于你怎么使用它们，还是要看你那些特定的应用在全局方面有什么样的需求。

在大多数基本层次上，你可以只允许加密的会话，让 SSL 协议来保护用户的密码。通过 GRANT 命令里的可选项，你也能要求用户通过 SSL 来连接：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE SSL;
```

然而，GRANT 无法对客户端所使用的 SSL 证书设置任何限制。只要客户端和 MySQL 服务器能够商定一个 SSL 会话，MySQL 就不会去检查客户端证书的合法性。

你可以使用 REQUIRE x509 选项对客户端证书做最低程度上的检查：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE x509;
```

这是要求客户端证书至少能通过 MySQL 服务器上已经设置承认的 CA 的认证。

547

另有一种提高安全度的方法是只允许一个特定的客户端证书访问数据库。你可以用 REQUIRE SUBJECT 语法来做到：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com";
```

可能你并不关心到底使用的是哪个特定的客户端许可证，只要它是你自己组织签发的 CA 证书就行。在这种情况下，你就可以像下面这样使用 REQUIRE ISSUER 语法来实现这个需求：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/emailAddress=admin@example.com";
```

在最终的认证里，你可以合并这两个子句用来要求签发者和标题都是预定的值。举例来说，你可以要求 Raymond 使用由你组织签发的特定的 CA 证书，就像下面这样：


```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'  
-> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets  
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com"  
-> AND ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets  
Inc./CN=cacert.example.com/ emailAddress=admin@example.com";
```

另外有一个跟 SSL 相关的次要选项就是 CIPHER 要求选项，它可以让管理者只允许使用了“可信任的”（高强度的）加密方法的密码。SSL 跟密码是相互独立的，如果一个真正的弱密码被用来保护传输的数据，那么，潜在的高强度 SSL 加密方法就会无效。你可以将选择限制在一组你认为安全的协议里，这个命令像下面这样：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'  
-> REQUIRE CIPHER "EDH-RSA-DES-CBC3-SHA";
```

管理单独的客户端证书看上去是极好的安全方式，但是，它也会成为管理员的噩梦。当你创建了一个客户端证书后，你就必须给它设定一个失效日期——最好是在不久的将来。你希望它的寿命足够长，这样就不会使你被迫经常性地重新生成新的证书。如果寿命足够短，那即使是它落入了敌对方的手里，他们访问你的数据的时间也长不了。

在一个只有几名雇员的小环境里，持续追踪单个证书的所有权是件非常容易的事情。但是，当你的组织规模扩大到几百乃至几千名雇员的时候，要追踪哪个证书已经过期，并确保证书在过期前被更新掉，将会相当的繁琐。

一些组织解决此类问题的方法是结合 REQUIRE ISSUER 和一系列月度客户端证书——这些证书是通过可信任的发布渠道（例如企业内部局域网）来发布的。使用这些够用一个月或两个月的证书，客户端就能下载和连接到 MySQL 服务器。这样，如果一个雇员丢失了公司内部局域网的访问权，或者一个合作人不再拥有月度 key 的访问权，甚至是管理员没被告知要删除某个用户的访问权，那他的连接权限也会根据预订的时间表自然地失效。

SSH 隧道技术

如果你用的是一个老版本的 MySQL，或者仅仅不想被 SSL 的设置所烦扰，那就可以考虑用 SSH 来代替。如果正使用 Linux 或 Unix，那么其实你已经在用 SSH 登录到远程服务器上了。（注 8）许多人所不知道的是使用 SSH 还可以在两个主机之间建立一条加密的隧道。

SSH 隧道最好用一个例子来说明。让我们假定你想在一台 GNU/Linux 工作站和 *db.example.com* 上的 MySQL 服务器之间建立一个加密连接。在工作站这边，你需要执行下述命令：（注 9）

```
$ ssh -N -f -L 4406:db.example.com:3306
```

这在工作站的 TCP 4406 端口与 *db.example.com* 的 3306 端口之间建立了一个隧道。现在，你就能通过隧道从工作站连接到 MySQL 了，就像这样：

```
$ mysql -h 127.0.0.1 -P 4406
```

SSH 是一种非常强大的工具，它所能做的远超过这个简单示例所表明的。Stunnel 是另一种用于创建安全隧道的工具，但是，它没有登录/shell 组件。在一些情形下，它也是替代 VPN 的好工具。

注 8：有一种 OpenSSH 变体可用于 Windows 客户端，Putty (<http://www.chiark.greenend.org.uk/~sgtatham/putty/>) 的使用也比较广泛。<http://www.vbmysql.com/articles/security/sshtunnel.html> 是一个完整的教程，它能指导你如何建立一个 SSH 隧道去连接 MySQL 服务器。

注 9：假设 SSH v2 已经安装了。SSH v1 没有 -N 选项。更多细节可以查看 SSH 文档。

12.4.5 TCP 封装

你可以在 Unix 系统里把 TCP 封装的支持编译到 MySQL 里。如果没有使用一个成熟的防火墙，那么 TCP 封装也能提供最基本的防御功能：在不必更改你的授权表的情况下，你可以对那些 MySQL 要连接或不连接的主机施加更多一层的控制。有一些操作系统，例如 Debian GNU/Linux，在默认配置下就是这么构建 MySQL 的。

为了能使用 TCP 封装，你需要构建 MySQL 的源码，把 `--with-libwrap` 选项传给 `configure`，这样它就会知道在操作系统上的哪个地方才能找到合适的头文件：

```
$ ./configure --with-libwrap=/usr/local/tcp_wrappers
```

假设在你的 `/etc/hosts.deny` 文件里有一个条目，默认条件下是拒绝所有连接：

```
# deny all connections
ALL: ALL
```

你可以显式地把 MySQL 加入到你的 `/etc/hosts.allow` 文件里：

```
# allow mysql connections from hosts on the local network
mysqld: 192.168.1.0/255.255.0.0 : allow
```

唯一还要做的是为 MySQL 在 `/etc/services` 里增加一个合适的条目。如果你还没设置过，那就加入下面一行：

```
mysql    3306/tcp  # MySQL Server
```

如果你是在一个非标准端口上运行 MySQL——不是在 3306 端口上。

TCP 封装会增加一些系统开销，例如逆向 DNS 查找——这会在 DNS 子系统里创建一个依赖——这不是你所希望看到的。

12.4.6 自动主机封锁

MySQL 在防止基于网络的攻击方面提供了一些帮助：如果它注意到有太多的有害连接来自一个特定的主机，它就会封锁所有来自该主机的连接。服务器变量 `max_connection_errors` 决定了 MySQL 在开始封锁主机前可以允许有多少个有害连接。“有害连接”指的是那些永不完成的连接（结果是一直占用着一个可用的 MySQL 会话）。拙劣的密码常常是有害连接的始作俑者，但是，网络问题也会导致有害连接的产生。

当 MySQL 封锁了一个主机时，它会把相应的消息写进日志，消息内容如下：

```
Host 'host.badguy.com' blocked because of many connection errors.
Unblock with 'mysqladmin flush-hosts'
```

如同消息显示的那样，在你弄明白了被封锁主机的连接问题，并采取了相应的措施之后，你可以使用 `mysqladmin flush-hosts` 命令让主机取消该封锁，`mysqladmin flush hosts` 命令仅仅执行了一个 `FLUSH HOSTS` 的 SQL 命令，它会清空 MySQL 的主机缓存表，还会解除所有被封锁主机的封锁，对于单独一个主机它无法做到。

出于某种原因，如果你发现有害连接的起因是一个普通问题，你就可以把 `my.cnf` 文件里的 `max_connection_errors` 变量设置为一个相对比较大的数值，以避免被封锁：

max_connection_errors=999999999

你不可能通过把 max_connection_errors 设为 0 来彻底关闭连接检查，而且通过任何其他办法你也无法做到。你最好还是找出并解决其背后的问题。

12.5 数据加密

Data Encryption

在应用里，常常存放着一些敏感数据，例如银行记录，你可能希望能以加密的形式来保存数据。对于那些未经授权的用户而言，使用这些加密了的数据将会很困难，哪怕他们能在物理上访问到你的服务器。全面讨论加密算法和技术会超出本书的内容范围，但是，我们会快速浏览一些跟它相关的主题。

12.5.1 密码散列

Creating Passwords

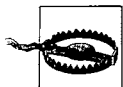
在一些不怎么敏感的应用里，你需要保护的只有一些数量较少的信息，例如另外一个应用的密码数据库。密码不应该用明文来保存，因此，它们在应用里一般都是被加密的。但是，除了加密之外，更明智的做法是效仿大多数的 Unix 系统，甚至 MySQL 本身：对密码使用散列算法，把结果保存在你的表里。

不同于传统的加密方法——它们都是可逆的，一个优秀的散列函数就是一个单向过程，是不可逆的。破解密码的唯一方法就是使用需要昂贵计算能力的暴力攻击生成一个个特定散列值进行匹配，穷尽所有可能的数值。

MySQL 提供了 3 个用户函数可用于密码的散列：ENCRYPT()、SHA1() 和 MD5()。(注 10) 查看每个函数执行结果的最好方式就是把它们放在同一段源代码试运行，让我们看一下字符串 p4ssword 经过三个函数散列后的结果是怎么样的：

```
mysql> SELECT MD5('p4ssword'), ENCRYPT('p4ssword'), SHA1('p4ssword')\G
***** 1. row *****
MD5('p4ssword'): 93863810133ebebe6e4c6bbc2a6ce1e7
ENCRYPT('p4ssword'): dDCjeBzIycENk
SHA1('p4ssword'): fbb73ec5afd91d5b503ca11756e33d21a9045d9d
```

551 每一个函数都会返回一个固定长度的包含了字母和数字的字符串，你可以把它存储在一个 CHAR 列里。为了能应付使用 ENCRYPT() 而带来的混合了大小写字母的结果，这个列的类型最好还是 CHAR BINARY。



警告：千万不要在应用里使用 MySQL 内部的 PASSWORD() 函数，它的结果在各个版本的 MySQL 里各不相同。

存储散列数据非常简单，就像这样：

```
mysql> INSERT INTO user_table (user, pass) VALUES ('user', MD5('p4ssword'));
```

要验证一个密码，就运行一个 SELECT 查询，看看提供的用户名和密码是否匹配。若以 Perl 为例，你可以这样来做：

注 10：MySQL 的 ENCRYPT() 仅仅是调用了 C 语言库里的 crypt() 函数。在一些 Unix 里，crypt() 是一个 MD5 的实现，使它跟 MD5() 没什么区别。在其他操作系统里，它就是传统的 DES 加密算法。

```
my $sth = $dbh->prepare('SELECT * FROM user_table '
    . 'WHERE user = ? AND pass = MD5(?)');
$sth->execute($username, $password);
```

密码散列易于使用，而且相对比较安全，保存在数据库里也不容易被还原。再做一点改进就能使字典攻击更加困难：把用户名和密码放在一起做散列。这将使密码的产生要依赖更多的变量：

```
my $sth = $dbh->prepare('SELECT * FROM user_table '
    . 'WHERE user = ? AND pass = SHA1(CONCAT(?, ?))');
$sth->execute($username, $username, $password);
```

这里唯一还有安全风险的是用明文将密码发送给 MySQL，明文密码会被写入到磁盘上的日志文件里，并且在进程列表里是可见的。对此，你可以把密码作为用户变量存储起来，从而减少一点风险，或者把散列过程移到应用里（在大多数编程语言里，都有专门的加密函数和库），这样就彻底避免了泄密问题。在下文中，我们将会简短地浏览一下应用层的加密。

12.5.2 加密的文件系统

Database File Encryption

MySQL 里的各种存储引擎都把它们的数据以普通文件的形式存储，不管你使用的是什么文件系统。所以，你也可以使用加密的文件系统。大多数流行的操作系统都至少有一种加密的文件系统可供使用（或者是免费的或者是商业的）。

这种方法的主要好处是你无须为了 MySQL 能利用加密文件系统的优点而特地去做些什么，因为所有加解密动作都是在 MySQL 之外完成的。MySQL 只要进行读写就可以，用不着任何关于底层操作细节的知识。所有你要做的事情就是确定把你的数据和日志文件都存放在正确的文件系统里。从你的应用那一端来看，是不是采用了这样的文件系统都一样，没什么区别的。

采用加密文件系统也有几个缺点。首先，因为你加密了所有数据、索引和日志，所以，在加解密这些信息的时候会产生大量的 CPU 开销。如果你正在考虑使用一个加密文件系统，那就一定要做个全面的基准测试，这样你就会了解你的服务器在重负载时的表现。

其次，要确保你做数据备份的时候，不会将这些数据都解密了。这不是个硬性规定，但是常常会忘记去做。

最后要知道的是一个加密文件系统对于那些能访问到数据所在的服务器的人来说，是没有任何保护的。因为文件系统的解密对于服务器来说是透明的，任何有访问服务器权限的人，也就能读到这些数据——甚至可以将这些数据做成一个解密了的副本。

12.5.3 应用层的加密

Application-Level Encryption

一种更常用的方法是把加密功能构建到应用（或中间件）里。当应用需要存储敏感数据的时候，它首先会加密数据，然后再把结果存放到 MySQL 里。从相反的方向来说，当应用从 MySQL 那里接收一份加密数据后，它会先将它解密了。

这种方法提供了许多弹性。它不会把你跟特定的文件系统、操作系统、甚至是数据库（如果你的代码是以一种通用的方式编写的话）捆绑在一起。这也使应用设计者能够自由选择加密算法，让选中的加密算法可以更加适

合存储的数据（在加解密速度和加密强度之间平衡）。

因为数据是被加密的，所以备份非常容易。无论是从哪里复制数据，它都是被加密的。然而，这也意味着这些数据必须通过一个知道怎么解密的软件才能访问到。你再也不能打开一个命令行工具，然后发出查询要求了。

应用层加密一般都是个不错的解决方案，但是，它也有一些缺点。举例来说，MySQL 会难以高效地对加密数据做索引；当你在处理机密数据的时候，要想优化 MySQL 的性能也更加难了。

设计要点

我们在这里提到的自由和弹性跟数据设计还有着一些有趣的牵连。一个要点是你必须确保你使用的列的类型要跟加密类型相适合。有些算法产生的数据块有一个最小尺寸。这意味着你的列必须有 256 字节大小，刚好能容纳一块加密后的数据——它的尺寸小于加密前的尺寸。还有，许多流行的加密库都是产生二进制数据，所以，你创建的列也要能存储二进制数据。作为一个替代办法，你可以把二进制数据转换为一个十六进制或 base-64 来表示，但这会要求更多的空间和时间。

553 要决定哪些数据要加密哪些不要加密也不容易。你要在安全和表内信息查询的便利性之间做好权衡。举例来说，你有一个账号表，它代表了一系列的银行账号，每一行包含了下列这些字段：

- id
- type
- status
- balance
- overdraft_protection
- date_established

哪些列有加密的需要呢？如果你加密了“balance”，这看上去有点道理，但是会难以应付常用的查询问题。例如，你编写了下面这样一个查询，要在这些账号里找出最大、最小和平均的余额数：

```
mysql> SELECT MIN(balance), MAX(balance), AVG(balance)
-> FROM account GROUP BY type;
```

查询的结果将会毫无意义。MySQL 不知道这些加密了的 balance 列是何含义，因此，仅仅是针对这些加密数据执行了那几个函数。

对此的解决方案是从 account 表里读取所有的行，然后为你自己需要的报表做数学运算。这不是非常的难，只是有点令人讨厌，因为你不仅需要重新实现 MySQL 已提供的函数，而且还会减缓处理速度。

所有这些都可以归结为在安全与关系数据库优点之间的权衡。包含了加密数据的列对 MySQL 内嵌函数而言都是无用的，因为它们需要在未加密的数据上进行操作。同时，还有一个类似的问题，就是查询优化。举例来说，在一个未加密的数据表里，你可以很方便地找出所有余额大于\$100 000 的账号：

```
mysql> SELECT * FROM account WHERE balance > 100000;
```

如果在 balance 列上有一个未加密的索引，MySQL 就能利用这个索引来找到所需要的行。但是，如果数据是被加密了的，你就不得不把所有行都读到你的应用里，把它们解密后再进行过滤。

MySQL 内部的加密和解密

这就是说，你可以把加密后的值存放在 MySQL 里，并根据需要使用内嵌函数对它们进行加密、解密。在这方面用途里的最好的函数是 `AES_ENCRYPT()` 和 `AES_DECRYPT()`，它们能把一个字符串转换成加密的二进制字符串，或者反过来做。它们的加密算法是对称的：你加密时使用的 key 就是你解密时要用到的 key。例如：

```
mysql> SET @key      := 's3cret';
mysql> SET @encrypted := AES_ENCRYPT('sensitive data', @key);
mysql> SELECT AES_DECRYPT(@encrypted, @key);
+-----+
| AES_DECRYPT(@encrypted, @key) |
+-----+
| sensitive data                |
+-----+
```

我们没有显示出加密后的数值是因为它的二进制格式的结果显示出来就是一些乱码。

然而，这个方法也无法解决我们提及的所有问题。比如说它无法回避索引问题；还有你的加密数据在 SQL 查询语句里仍旧是明文形式，并被写入到系统日志里（假设日志功能是开启的）。但是，我们还是要给你指出一个步骤，可以降低别的用户看到你的加密数据的风险：把你的加密 key 存放在一个用户变量里，而保证用户变量安全性的方法就有很多了。例如，你可以把这个变量放在一个存储过程里，通过这个存储过程来设置它的值，同样，也能限制对这个存储过程的访问。这就能让其他用户难以确定这个 key 的值。

12.5.4 源代码修改

如果你想寻找一个比加密文件系统或基于应用的加密技术更具弹性的方法，那你可以自己来构建一个自定义的解决方案。MySQL 的源代码在 GNU General Public License 授权下可以自由使用。

这类工作要求你自己懂 C++，或者雇佣某个人来帮你做这个。满足了这个前提后，你就可以使用原生的加密支持来创建自己的存储引擎了，或者更简单一点，用加密支持扩展一个已有的存储引擎。

12.6 在 Chroot 环境里使用 MySQL

在 Unix 系统里，让服务器运行在 chroot 环境中能极大地提高整个系统的安全性。chroot 环境会建立一个独立的环境，指定目录以外的文件都无法被外界访问到。通过这种方法，即使服务器上有一个安全漏洞被发现并利用，其潜在的破坏也将被限定在那个目录的范围即那个特定的应用所使用的全部文件内。

如果想让你的 MySQL 应用运行在一个 chroot 环境里，你就必须这么开始：编译 MySQL 源代码，或者是解包并安装 MySQL AB 提供的二进制包。许多管理员认为这么做是理所当然的，但是这对于一个 chroot 化的应用来说却是必须的：许多预打包了的 MySQL 安装程序会把一些文件放在 `/usr/bin` 下，还有一些放在 `/var/lib/mysql` 下，等等；但是所有在一个 chroot 安装里的文件都要被放置在同一个目录结构下面。

我们将要做的是创建一个 `/chroot` 目录，所有 chroot 化的应用都会安装在那里。为了实现这个目的，你要像下面这样配置你的 MySQL 安装脚本：

```
$ ./configure --prefix=/chroot/mysql
```