

优化服务器设置

人们通常会问：“我的服务器有 16GB 内存，100GB 数据，需要怎样的优化配置文件？”这样的文件其实并不存在。具体的配置文件依赖于服务器的硬件、数据量、查询类型、响应时间、事务持久性和连续性等因素。

MySQL 的默认配置不适用于使用大量资源，因为其通用性很高，通常不会假设机器上只安装 MySQL。在默认情况下，配置文件只够启动 MySQL 并且对少量的数据运行简单的查询。如果有较多的数据，那么肯定需要对配置文件进行定制。可以先尝试使用随 MySQL 发布的配置文件，然后按照需要对它进行配置。

不要期望改变配置文件会带来巨大的性能提升。提升的具体大小取决于工作负载，通常可以通过选择适当的配置参数得到两到三倍的性能提升（具体什么参数起作用取决于多种因素）。在这之后，性能提升就是增量的。通过改变一两个配置参数，您可以看到某些原本运行较慢的查询快了一些，但是，服务器的性能通常不会提升一个数量级。为了达到更大的提升，通常需要检查服务器架构、查询及应用程序的架构。

本章先说明了 MySQL 的配置选项如何起作用并且该如何更改它们。接下来讨论了 MySQL 如何使用内存及如何优化内存使用。然后以同样的详细程度讨论了 I/O 及磁盘存储。接下来是基于工作负载的调优，这有助于在特定的负载下得到最佳性能。最后讨论了对某些查询进行动态参数设置的方法。

 提示：关于术语的说明：因为许多 MySQL 的命令行选项都和服务器变量相对应，所以有时不会对它们进行区分。

266

6.1 配置基础知识

<http://www.mysqltutorial.org/configuring/>

本节概述了如何成功地配置 MySQL。我们首先解释了 MySQL 配置实际是如何工作的，然后讨论一些最佳实践。MySQL 通常对配置相当宽容，但是接受这些建议也许能节约您大量的时间和工作。

首先要知道的是 MySQL 从什么地方得到配置信息：命令参数和配置文件中的设置。在 Unix 类系统中，配置文件通常位于 `etc/my.cnf` 或 `/etc/mysql/my.cnf`。如果使用操作系统的启动脚本启动 MySQL，这通常是唯一一个配置选项的地方。如果手动启动 MySQL，也可以通过命令行定义参数。

 提示：大部分变量和相应的命令行选项有同样的命令，但是也有一些例外。比如，`-memlock` 设置了变量 `locked_in_memory`。

任何长久使用的设置都应该被放到全局配置文件中，而不是通过命令行进行定义。否则就有可能会忘记使用它们。把所有的文件放在一个地方也是一个好主意，这样的话就可以很容易地检查它们。



要确认自己知道服务器的配置文件在什么地方。我们曾经见过有人用 MySQL 根本无法读取的文件进行调优。比如在 Debian GNU/Linux 机器上，实际起作用的是 /etc/mysql/my.cnf，而不是 /etc/my.cnf。有时在好几个地方都有文件，这也许是因为前一位系统管理员也被弄糊涂了。如果不知道服务器读取的是哪个文件，可以进行查询：

```
$ which mysqld  
/usr/sbin/mysqld  
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'  
Default options are read from the following files in the given order:  
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

267 这适用于典型安装，即主机上只有一个服务器的情况。可以设计更复杂的配置，但是并没有标准的办法。MySQL 包含了一个叫 mysqlmanager 的程序，它可以使用同一个文件的不同部分运行出多个配置文件的实例（它是较老的 mysqld_multi 脚本的替代品）。但是，许多操作系统没有在启动脚本里面包含这个程序。事实上，许多人根本就不使用 MySQL 提供的启动脚本。

配置文件被分成了若干部分，每部分的第一行都是用方括号括起来的该部分的名字。MySQL 程序会读取和程序名同名的部分，并且许多客户端程序会读取 client 部分，这儿也是放置通用设置的地方。服务器通常会读取 mysqld 部分。要确保自己的设置被放在了正确的部分里面，否则它们就不会有任何作用。

6.1.1 语法、作用域及动态性

SYNTAX, SCOPE, AND DYNAMICISM

配置设置都是小写的，使用下画线或破折号分割单词。下面的两个设置是一样的，在命令行和配置文件中都可使用这两种格式：

```
/usr/sbin/mysqld --auto-increment-offset=5  
/usr/sbin/mysqld --auto_increment_offset=5
```

我们的建议是选择并坚持一种格式。这会让搜索设置容易得多。

配置设置有几种作用域。一些设置在整个服务器内都有效（全局域）；另外一些针对每个连接（会话域）；还有一些只对对象有效。许多会话域的变量和全局变量是一样的，可以认为是全局变量提供了默认值。如果修改了会话域变量的值，它只会在当前连接内有效，连接关闭后值就消失了。下面是一些值得注意的例子：

- query_cache_size 变量是全局性的。
- sort_buffer_size 变量有全局性的默认值，但是可以在会话中进行设置。
- join_buffer_size 有全局性的默认值，并且可以在会话中进行设置。但是联接了几个表的查询可能会为每一个联接都分配一个联接缓冲区，所以每个查询都可能有几个联接缓冲区。

除了在配置文件中设置变量，也可以在服务器运行的时候对某些值（不是全部值）进行设置。MySQL 把它们叫做动态变量。下面的语句显示了几种不同的设置会话和全局变量的方法：

```
SET          sort_buffer_size = <value>;  
SET GLOBAL    sort_buffer_size = <value>;  
SET          @@sort_buffer_size := <value>;  
SET @@session.sort_buffer_size := <value>;  
SET  @@global.sort_buffer_size := <value>;
```

如果动态地设置了变量，那么它们在 MySQL 关闭之后就会丢失。如果想保留这些设置，就应该同时更新配置文件。

如果在服务器正在运行的时候设置了一个全局变量的值，当前会话和其他已有会话的值不会受到影响。这是因为会话的值是在连接创建的时候从全局变量初始化的。可以在每次更改值后使用 `SHOW GLOBAL VARIABLES` 检查一下结果，确保更改已经生效。

变量使用不同的单位，应该要知道每个变量正确的单位是什么。比如，`table_cache` 变量定义了能被缓存的表的数量，而不是被缓存的字节数。`key_buffer_size` 的单位是字节，然而还有其他变量的单位是页面或者其他单位，例如百分比。

许多变量都可以使用前缀，比如 `1MB` 指 1 兆字节。然而，这只在配置文件或命令行参数中有用。当使用 `SET` 命令时，必须使用字面值 `1 048 576` 或者 `1024×1024`。在配置文件中无法使用表达式。

还可以使用 `SET` 给变量赋一个特殊的值：`DEFAULT`。给会话域内的变量赋 `DEFAULT` 值将会使其变成相应的全局变量的值；给一个全局变量赋 `DEFAULT` 将会使其得到编译时的默认值（不是配置文件中的值）。这对重置会话变量的值非常有用。我们建议不要对全局变量使用这个值，因为这有可能和期望的情况不太一样，那是因为它不会将变量设置成服务器刚启动时的值。

6.1.2 设置变量的副作用

Effects of Setting Variables

动态设置变量有出人意料的副作用，比如会清空缓冲区。要注意在线更改的设置，因为它可能会导致服务器做大量的工作。

有时从变量的名字推断变量的行为。比如，`max_heap_table_size` 做的事情和名字一样：定义了隐式的内存临时表能增长到的最大值。但是，命名规范并不完全一致，所以不能总是从名字推测变量的行为。

下面来看看某些重要的变量及动态地改变它们造成的影响：

`key_buffer_size`

269

设置这个变量给键缓冲区（或者说键缓存）分配指定大小的空间。但是操作系统只有在实际用到这些空间的时候才会进行分配。例如，将键缓冲区大小设置为 1GB，并不意味着服务器就会真正地给它分配 1GB 空间（在下一章就会讨论如何观察服务器内存使用情况）。

可以创建多个键缓存，本章稍后将会谈到这个问题。如果对于一个非默认大小的键缓存，将它的值设置为 0，MySQL 就会把每一个索引从特定的缓存移到默认的缓存中，并且在没有对象使用特定的缓存时，就会将其删除掉。给一个不存在的缓存设置这个变量将会创建缓存。

对一个已有的缓存设置非零值将会冲洗缓存，从技术上来说，这是一个在线操作，但是它会阻止所有访问该缓存的动作，直到缓存冲洗完成。

`table_cache_size`

设置这个变量不会立即生效，要等到下一个线程打开表的时候才会生效。当它生效的时候，MySQL 会检查变量的值。如果值大于缓存中表的数量，线程就可以把新打开的表插入到缓存中。如果值小于缓存中表



的数量，MySQL 就会从缓存中删除掉没有使用的表。

thread_cache_size

设置这个变量不会立即生效，生效被延迟到了下一次线程关闭的时候。在那时，MySQL 检查缓存中是否有空间存储线程。如果是，它会把线程缓存起来，供另外一个连接使用。如果不是，它会直接结束掉线程。在这种情况下，缓存中线程的数量，以及线程缓存使用的内存数量不会立即就下降。只有当新连接为了使用线程把它从缓存中移走的时候才会看到下降。（MySQL 只有在连接关闭的时候才会把线程加入缓存，也只有在创建新连接的时候才从缓存中移除线程。）

query_cache_size

在服务器启动的时候，MySQL 会为查询缓存一次性分配变量所定义数量的内存。如果更新了变量（即使是把值设置为当前值），MySQL 会立即删除掉所有缓存的查询，重新把缓存设置为定义的大小，并且重新初始化缓存的内存。

read_buffer_size

MySQL 只有在查询需要的时候才会为该缓存分配内存，但是它会一次性地把指定的大小分配给该缓存。

read_rnd_buffer_size

MySQL 只会在查询需要的时候才会给这个缓冲区分配内存，并且只会分配所需的内存（该变量更精确的名字应该是 max_read_rnd_buffer_size）。

sort_buffer_size

MySQL 只有在查询需要排序的时候才会为这个缓冲区分配内存。但是只要发生了排序，MySQL 会立即分配变量定义的所有内存，不管是否需要这么大的空间。

在本书另外的地方对这些变量有更详细的解释。这儿的目的是简单地展示更改这些重要的变量会发生什么事情。

6.1.3 开始配置

Getting Started

设置变量的时候要小心。更大的值并不总是好事情，如果将值设得太高，很容易引发诸多问题：耗尽内存、导致服务器使用交换区、耗尽地址空间等。

我们建议你在运行服务器之前就开发一套基准测试组件（第 2 章已经讨论过基准测试）。为了优化服务器配置，需要用测试组件来模拟总体工作负载和一些边界条件，比如极其庞大和复杂的查询。如果已经确认了一个特定的问题，比如某个查询运行很慢，那么就可以试着优化这个查询，但是这有可能会在你毫不知情的情况下为其他查询带来负面影响。

应该总是使用监控系统来衡量改动是提升了还是损害了服务器总体性能。基准测试是不够的，因为它们覆盖的范围不够广。如果不衡量服务器的总体性能，就有可能在不知情的情况下伤害性能。我们曾经见过很多例子，某些人改动了服务器的配置，以为可以提高性能，但由于工作负载的变化，实际上却损害了服务器的总体性能。第 14 章讨论了一些监控系统。

最好的方式每次只小幅度地改动一两个设置，并且在每次改动之后都进行基准测试。有时结果会让你惊讶，也许增加一点点就带来了性能提升，但是再增加一点就让性能急剧下降。如果在某次改动后性能下降了，那么有可能是过多地请求了某种资源，比如为某个经常被分配和释放的缓冲区请求了过多内存。也有可能是在服务器和操作系统或硬件之间造成了某种不匹配。例如，`sort_buffer_size` 有可能会受 CPU 缓存的影响，`read_buffer_size` 要和服务器的预读和通用 I/O 子系统相匹配。更大的值并不总是意味着更好。一些变量也会

依赖于其他变量。了解这些东西需要经验，也需要对系统架构有一定的了解。比如，`innodb_log_file_size` 271 的最佳大小就依赖于 `innodb_buffer_pool_size`。

如果习惯做笔记，比如在配置文件中添加注释，这会为你（以及你的继任者）节约大量的工作。一个更好的办法就是对配置文件进行版本控制。这实际上非常有好处，因为可以回滚自己的改动。为了减少管理大量配置文件的复杂性，可以简单地在配置文件中创建一个到中央版本控制库的符号链接。在一些系统管理的书籍中有很多这方面的内容。

在对配置进行调优之前，应该对查询和结构进行调优，进行一些最基本的优化，比如添加索引。如果已经对配置文件做了很多调整，再回过头来更改查询和架构，那么就有可能要重新调整配置文件。要知道调优是一个渐进的过程。除非硬件、工作负载及数据是完全静态的，否则就需要在随后的工作中对配置进行再次调整。这意味着并不需要一次性把服务器的性能调到最好，实际上，对配置文件花费大量的时间也许会收效甚微。我们的建议就是让配置保持“够好”就行了。除非忘记了某项重要的设置，否则就不需要再次改动它。当更改了查询或架构的时候，就可以回过头来再次修改配置文件。

我们通常会为不同的目的开发不同的配置文件样本，然后把它们当成默认设置。这一点在安装大量类似的服务器时尤其有用。但是，正如本章开头所说的那样，并没有一个通用的配置文件适合所有服务器。每个人都需要开发自己的配置文件，因为好的起点依赖于如何使用服务器。

6.2 通用调优原则

General Tuning

可以把调整配置文件看成一个两步的过程：在安装的时候使用适当的初始值，然后基于工作负载进行细节调整。

你也许会使用 MySQL 提供的样本配置作为起始点。考虑服务器的硬件会帮助做出选择。服务器有多少内存、硬盘和 CPU？那些样本文件的名字通常是一目了然的，比如 `my-huge.cnf`、`my-large.cnf`、`my-small.cnf`。但是，这些样本文件通常只适用于 MyISAM 表。如果使用了其他的存储引擎，就需要创建自己的配置文件。

6.2.1 内存使用调优

Tuning Memory Usage

配置 MySQL 正确地使用内存对性能至关重要。对 MySQL 内存使用进行定制基本上是肯定的。可以认为 MySQL 的内存消耗有两种范畴：可以控制的和不可控制的。你不能控制 MySQL 使用多少内存来运行服务器、解析查询及管理内部运行，但是可以控制它为特定工作使用多少内存。妥善使用可控内存并不是难事，但是需要你明白自己配置的是什么。

可以用下面的方式进行内存调优：



1. 决定 MySQL 能使用的内存的绝对上限。
2. 决定 MySQL 会为每个连接使用多少内存，比如排序缓冲区和临时表。
3. 决定操作系统需要多少内存来很好地运行自身，包括机器上的其他程序，比如周期性工作。
4. 假设上面的工作都已完成，就可以把剩余的内存分配给 MySQL 的缓存，比如 InnoDB 的缓存池。

下面几节会依次解释上面的步骤，然后会对 MySQL 不同的缓存的要求做详细讲解。

MySQL 能使用多少内存

在特定的系统上，MySQL 可能使用的内存有一个绝对的上限。起始点是服务器物理内存的数量。如果服务器没有，那 MySQL 肯定就用不了。

也可以考虑操作系统或系统架构的限制，比如 32 位系统能给每一个进程分配的最大内存。MySQL 是一个有多个线程的单进程程序，所以它能使用的内存数量一定会受操作系统的限制。比如，在 32 位 Linux 内核的操作系统上，单个进程能使用的地址空间在 2.5GB 到 2.7GB 之间。耗尽地址空间是很危险的，会导致 MySQL 崩溃。

还有很多其他和操作系统相关的参数也需要考虑进来，不仅仅包括单个进程的限制，还要考虑堆栈大小和其他设置。系统函数库也会对分配造成影响，比如函数库不能一次分配 2GB 以上的内存，那么就不能把 `innodb_buffer_pool` 的值设得比 2GB 高。

即使是在 64 位系统上，一些限制仍然存在。例如，许多我们讨论的缓冲区，比如键缓冲，在 64 位系统上被限制在 4GB。一些限制在 MySQL 5.1 中被放开了，并且以后放松的空间可能会更大，因为 MySQL AB 正在让 MySQL 使用更加强大的硬件。MySQL 手册规定了每个变量的最大值。

单个连接需要的内存

MySQL 只需要很少的内存保持连接开启。它也需要一定的基本内存来执行查询。你需要在 MySQL 工作负载处于顶峰的时候为它分配足够的内存，否则查询就会因为内存不足而变得很慢，甚至失败。

知道 MySQL 在负载处于顶峰的时候需要多少内存是有用的，但是某些使用模式会出人意料地消耗大量的内存，这使得内存消耗变得难以预料。准备语句是一个例子，因为可以一次性打开很多准备语句。另外一个例子就是 InnoDB 的表缓存（稍后有更多这方面的内容）。

在预测最高内存消耗的时候不用假设最坏的场景。例如，如果 MySQL 被配置成可以接受 100 个连接，理论上应该可以同时在这 100 个连接里面运行大的查询，但是实际上这不可能发生。例如，如果把 `myisam_buffer_size` 设置为 256MB，最坏情况下就会消耗 25GB 内存，但是如此高的消耗在实际中不太可能发生。

和计算最坏情况相比，比较好的办法就是观察服务器在真实负载下的内存消耗，可以通过观察进程虚拟内存大小得到相应的数据。在许多 Unix 系统中，可以在 `top` 的 `VIRT` 栏，或者是 `ps` 的 `vsz` 栏看到这一数据。下章会对内存检测做更多说明。

为操作系统保留内存

和为查询指定内存一样，还应当为操作系统保留足够的内存。这样最大的好处就是服务器不会主动把虚拟内存

保存在磁盘上（更多详情请查看第 334 页的“交换”）。

不需要为操作系统保留多于 1GB 或 2GB 的内存，即使服务器安装了很多内存也不需要。添加一点是为了保险，如果机器上有消耗内存的周期性任务（比如备份），那么还可以多给一点。不要为操作系统的缓存添加任何内存，因为这些缓存可能会很大。操作系统通常会为这些缓存使用剩余的内存，并且我们通常认为它们独立于操作系统自身需要的内存。

为缓存分配内存

274

如果服务器是 MySQL 专用的，就不需要为操作系统或用于处理查询的缓存保留任何内存。

MySQL 缓存比其他的东西需要更多的内存。它使用缓存来避免磁盘访问。磁盘访问比内存访问慢几个数量级。操作系统也许会为 MySQL 缓存一些数据（尤其是为 MyISAM），但是 MySQL 自己也需要大量的内存。

对于大部分用户来说，下面的这些缓存是最重要的：

- 操作系统为 MyISAM 的数据提供的缓存。
- MyISAM 键缓存。
- InnoDB 缓存池。
- 查询缓存。

还有另外一些缓存，但是它们通常不会使用太多内存。前一章详细地讨论了查询缓存，所以在接下来的章节将会集中讨论 MyISAM 和 InnoDB 需要的缓存。

如果只使用一个存储引擎，服务器调优就容易得多。如果只使用 MyISAM 表，就可以完全地禁止 InnoDB。如果只使用 InnoDB，就可以只给 MyISAM 分配最少的资源（MySQL 某些内部操作需要 MyISAM）。但是如果混合使用了多个存储引擎，那么就很难在它们之间找到平衡。我们发现最好的方式就是进行有根据的猜测，并且进行基准测试。

6.2.2 MyISAM 键缓存

MyISAM Key Cache

MyISAM 键缓存也被叫做键缓冲区。默认只有一个缓冲区，但是可以创建多个。和 InnoDB 及其他存储引擎不同的是，MyISAM 自身只缓存了索引，没有数据（它让操作系统缓存数据）。如果主要是使用 MyISAM，那么就应该为键缓存分配很多内存。



提示：本节的大部分建议都是假设你只使用 MyISAM。如果混合使用了 MyISAM 和其他存储引擎，比如 InnoDB，就需要考虑所有存储引擎的需要。

最重要的选项是 `key_buffer_size`，它的值应该占到所有保留内存的 25% 到 50%。值得一提的是操作系统缓存，它通常用来保存从 MyISAM 的.MYD 文件中读取出来的数据。对于 MySQL 5.0，不管采用的是什么架构，该变量的最大上限都是 4GB。

275 MySQL 5.1 允许更大的值。请查询文档以确定 MySQL 的版本。

在默认情况下，MyISAM 将所有索引缓存在默认的键缓冲中，但是可以创建多个命名键缓冲区。这样就可以一次在内存中保存 4GB 以上的索引。为了创建名为 key_buffer_1 和 key_buffer_2，大小都为 1GB 的键缓冲区，可以在配置文件中加入下面两行：

```
key_buffer_1.key_buffer_size = 1G  
key_buffer_2.key_buffer_size = 1G
```

现在就有 3 个缓冲区了。两个显式创建的和一个默认的。可以使用 CACHE INDEX 命令把表映射到缓存。也可以通过下面的命令告诉 MySQL 把表 t1 和 t2 的索引保存到 key_buffer_1：

```
mysql> CACHE INDEX t1, t2 IN key_buffer_1;
```

现在当 MySQL 从这些表的索引中读取数据的时候，它就会把数据保存到特定的缓冲区了。还可以使用 LOAD INDEX 把表的索引预加载到缓存中：

```
mysql> LOAD INDEX INTO CACHE t1, t2;
```

可以把这个 SQL 语句放到一个 MySQL 启动时执行的文件里面。文件名字必须在 init_file 选项中定义，并且文件可以包含多个 SQL 命令，每个命令一行，而且不能有注释。MySQL 第一次访问.MYI 文件时，任何没有显式地映射到键缓冲区的索引都会被放在默认缓冲区中。

可以使用 SHOW STATUS 和 SHOW VARIABLES 监视键缓冲区的使用情况和性能。可以通过下面的公式检查缓存命中率和缓冲区使用的百分比：

缓存命中率

```
100 - ( (Key_reads * 100) / Key_read_requests )
```

缓存使用百分比

```
100 - ( (Key_blocks_unused * key_cache_block_size) * 100 / key_buffer_size )
```

 提示：介绍了一些工具，比如 innostop，可以使性能监视更加方便。

了解缓存命中率是很好的，但是它也可能造成误导。例如，99% 和 99.9% 之间的差别看上去很小，但是它实际代表了 10 倍的提升。缓存命中率也依赖于应用程序：一些程序也许只需要 95%，但是另外一些 I/O 密集的程序也许需要 99.9%。使用大小适当的缓存，命中率甚至可以达到 99.99%。

每秒内缓存未命中的数量更有实际意义。假设某个硬盘每秒能执行 100 次随机读取，如果有 5 次未命中不会导致工作成为 I/O 密集型，但是如果每秒 80 次，那就会造成问题。可以使用下面的公式计算它的值：

```
Key_reads / Uptime
```

时间间隔从 10 秒到 100 秒依次增加，计算未命中的次数，就可以了解当前的性能。下面的命令以 10 秒的幅度递增，计算了总的未命中次数：

```
$ mysqladmin extended-status -r -i 10 | grep Key_reads
```

在决定给键缓存分配多少内存的时候，知道 MyISAM 索引使用了多少磁盘空间会比较有帮助。没有必要让键缓



存大于数据的大小。如果使用 Unix 系统，可以使用下面的命令找出存储索引的文件大小：

```
$ du -sch `find /path/to/mysql/data/directory/ -name "*.MYI"'
```

要记住 MyISAM 使用操作系统缓存数据文件，它通常大于索引的大小。因此，留给操作系统缓存的内存比留给键缓存的要多也是很正常的。最后，即使没有使用 MyISAM 表，也要给 `key_buffer_size` 设置少量的内存，比如 32MB。MySQL 有时会使用 MyISAM 执行一些内部操作，比如给有 GROUP BY 的查询提供临时表。

MyISAM 数据块大小

键数据块大小是重要的（尤其是对于写入密集型的工作负载），因为它导致了 MyISAM、操作系统缓存，以及文件系统之间的交互。如果键数据块太小，就会遇到写入排队的情况，也就是操作系统只有等读取完成之后才能写入。下面是写入排队的例子，假设操作系统的页面大小是 4KB（这是 X86 架构的典型大小）并键数据块的大小是 1KB：

1. MyISAM 从磁盘请求 1KB 数据。
2. 操作系统从磁盘读取 4KB 数据，并缓存起来，然后把需要的 1KB 数据传给 MyISAM。
3. 操作系统将缓存中的数据丢掉。
4. MyISAM 修改了那 1KB 数据并要求操作系统把它写回磁盘。
5. 操作系统从磁盘读取同样的 4KB 数据，放入缓存中，修改其中的 1KB，并且把这个数据块写回磁盘。

写入等待发生在第 5 步，也就是 MyISAM 要求操作系统只写入其中一部分数据的时候。如果 MyISAM 数据块的大小和操作系统读取的页面大小相同，第 5 步就可以避免（注 1）。277

不幸的是，MySQL 5.0 及其早期版本没有办法配置键数据块的大小。但是，在 MySQL 5.1 及更高版本中，可以使键数据块大小和操作系统相匹配，以避免写入等待。`myisam_block_size` 变量控制了键缓存块的大小。也可以在 CREATE TABLE 或 CREATE INDEX 语句中为每一个键定义 `KEY_BLOCK_SIZE` 选项来控制键的大小。但是由于所有的键都存储在相同的文件中，所以的确需要它们的大小都等于或大于操作系统的值，以避免由于对齐导致的写入等待问题。（比如，一个键数据块大小是 1KB，另外一个是 4KB，4KB 大小可能就无法和系统的页面相匹配。）

6.2.3 InnoDB 缓冲池

The InnoDB Buffer Pool

如果主要是使用 InnoDB，InnoDB 缓冲池也许会比其他的东西需要更多内存。和 MyISAM 键缓存不同，InnoDB 缓冲池不仅仅保存了索引，它还保存了行数据及自适应的哈希索引（查看第 101 页的“哈希索引”）、插入缓冲区、锁及其他内部结构。InnoDB 也使用了缓冲池帮助延迟写入，这样它就可以合并更多的写入然后顺序地执行它们。简言之，InnoDB 严重依赖于缓冲池，并且应该给它分配足够的内存。MySQL 手册建议在专用服务器上把 80% 的物理内存分配给缓冲池。实际上，如果机器有许多内存的话，还可以给它分配更多的内存。和 MyISAM

注 1：理论上，可以把原始的 4KB 数据保存在操作系统的缓存中，从而不需要读取。但是，我们实际上不能控制操作系统把什么数据留在缓存中。可以使用 `fincore` 工具来查看哪个数据块被留在了内存中。可以到 <http://net.doit.wisc.edu/~plonka/fincore/> 下载该工具。



键缓冲区一样，可以使用 SHOW 命令或 innontop 工具来监视 InnoDB 缓冲池的性能和内存使用情况。

InnoDB 没有和 LOAD INDEX INTO CACHE 等价的命令。但是，如果正在给服务器暖身，使它为繁重的负载做好准备的话，可以使用查询进行全表扫描或全索引扫描。

在大部分情况下，应该使 InnoDB 缓冲池和可用内存保持一致。但是，在少数情况下，很大的缓冲池（比如 50GB）会导致长时间的延迟。比如，大型的缓冲池在检查点或插入缓存合并操作的时候会变慢，并且并发也会因为锁定而减少。如果遇到了这些问题，就应该减少缓冲池的大小。

278 可以改变 `innodb_max_dirty_pages_pct` 的值，让 InnoDB 改变保留在缓冲池中被修改的页面的数量。如果允许保留更多被修改的页面，InnoDB 就需要更长的时间来关闭，因为它会在关闭之前把修改的页面写入数据文件。可以强制它快速关闭，但是它在重新启动的时候就会要做更多的恢复工作，所以这实际上不能加快从关闭到启动的周期。如果预先知道需要关闭，就可以把这个变量设置为较小的值，并等待它冲刷线程以清理缓冲池，然后在修改的页面数量变小的时候关闭 InnoDB。可以通过观察状态变量 `Innodb_buffer_pool_pages_dirty` 或使用 innontop 监视 SHOW INNODB STATUS 的值来监测修改过的页面的数量。

降低 `innodb_max_dirty_pages_pct` 的值其实并不会确保 InnoDB 会在缓冲池里面保存较少的修改的页面。相反，它控制了 InnoDB 什么时候开始“变懒”。InnoDB 默认的行为是使用后台线程冲刷修改过的页面、合并写入查询并且顺序地执行写入动作。这种行为是“懒惰的”，因为它只会在其他数据需要空间的时候才会执行冲刷行为。当修改过的页面超过变量规定的值，InnoDB 会尽快地冲刷缓冲池，以保持尽可能少的修改过的页面。这个变量的默认值是 90%，所以在默认情况下，InnoDB 只会在修改过的页面已经占据了缓冲池 90% 空间的时候，才开始冲刷动作。

如果希望改变写入状况，就可以按照自己的工作负载调整这个值。比如，将它降低到 50% 就会让 InnoDB 执行更多的写入操作，因为它会更快地冲刷页面，并且也不会执行批次写入。但是，如果工作负载有很多写入的尖峰，使用更低的值有助于 InnoDB 更好地吸收尖峰。它会有更多的空余内存来保存修改后的页面，所以它不会等待其他修改过的页面被冲刷到磁盘上。

6.2.4 线程缓存

The Thread Cache

线程缓存保存了和当前连接无关的线程。这些线程可以供新连接使用。当一个新连接被创建出来并且缓存中有一个线程的时候，MySQL 就会把这个线程从缓存中删除，并且把它赋给连接。连接关闭时，MySQL 会回收线程，把它放回到缓存中。如果缓存中没空间了，MySQL 就会销毁该线程。只要缓存中有自由的线程，MySQL 就能很快地响应连接请求，因为它不需要为每个连接都创建新的线程。

`thread_cache_size` 定义了 MySQL 能在缓存中保存的线程数量。除非服务器有很多连接请求，否则就不需要改变这个变量的值。可以通过观察 `threads_created` 变量的值，以确定线程缓存是否足够大。如果每秒创建的线程数量少于 10 个，缓存的大小就是足够的。但是要让每秒创建的线程少于 1 个，也是很容易的。

279 一个好办法就是观察 `Threads_connected` 的值，并且把 `thread_cache_size` 的值设得足够大，以处理波动的负载。例如，如果 `Threads_connected` 通常在 100 到 200 之间变化，那么就可以把线程缓存设成 100。如果它在 500 到 700 之间变化，将线程缓存设置为 200 就足够了。要这样思考这个问题：有 700 个连接的时候，缓存中也许根本就没有线程。那么当有 500 个连接的时候，缓存中就有 200 个线程了。

对于大多数情况而言，非常巨大的线程缓存是没有必要的。但是让它很小并不会节约内存，所以那么做没什么



好处。每个在缓存中的线程通常消耗 256KB 内存。这和线程处于活动状态时处理查询所需要的内存相比简直微不足道。通常说来，需要把线程缓存保持得足够大，以使 `Threads_created` 不会经常增加。但是如果它的值非常大（比如好几千），那么就应该把它设置得小一点。这是因为操作系统不能很好地处理极其多的线程，即使它们处于睡眠状态也不行。

6.2.5 表缓存

表缓存和线程缓存是相似的概念。但是它存储了能表示表的对象。缓存中的每个对象都包含了解析表后生成的.frm 文件和其他数据，对象中其他的东西依赖于表的存储引擎。例如，对于 MyISAM 而言，它保存了表的数据或索引文件描述符。对合并表，它保存了大量的描述符，因为合并表有许多下属表。

表缓存有助于复用资源。例如，当查询要求访问 MyISAM 表的时候，MySQL 就可以从缓存中取出一个文件描述符，而不是打开一个文件。表缓存也有助于避免索引头中的 I/O 请求使 MyISAM 表的状态变为“正在使用”（注 2）。

表缓存的设计有一点以 MyISAM 为中心。由于一些历史原因，它属于服务器和存储引擎之间的隔离不是很清晰的区域。对于 InnoDB 来说，表缓存不是那么重要，因为 InnoDB 在很多方面都不会依赖于它（例如保存文件描述符，InnoDB 为此有自己的表缓存）。但是，InnoDB 还是可以从解析后的.frm 文件中获益。

在 MySQL 5.1 中，表缓存被分为了两个部分：一部分为打开表，另一部分为表的定义（通过 `table_open_cache` 和 `table_definition_cache` 定义）。因此，表的定义（解析过的.frm 文件）和其他资源是隔离的，比如文件描述符。打开的表仍然基于每个线程、每个使用的表。但是表定义是全局的，可以在所有的连接中共享。通常可以把 `table_definition_cache` 的值设置得足够高，以缓存所有表的定义。除非有上万的表，否则这项工作是很容易的。208

如果 `Opened_tables` 的值很大或者正在上升，那就说明表缓存不够大，应该增加系统变量 `table_cache` 的值（在 MySQL 中是 `table_open_cache`）。将表缓存变得很大的唯一坏处就是在有很多 MyISAM 表的时候，它会导致较长的关闭时间，因为要冲刷键数据块，而且表要被标记为不再打开。出于同样的原因，它也会导致 `FLUSH TABLES WITH READ LOCK` 需要较长时间才能完成。

如果收到 MySQL 不能打开更多文件的错误提示（使用 perror 工具检查错误码的详情），那就应该增加 MySQL 允许保持开启的文件数量。可以在 my.cnf 文件中设定 `open_files_limit` 解决这个问题。

线程和表缓存其实都不会使用太多内存。它们的好处就在于可以保存资源。尽管创建新线程、打开文件和其他操作比起来并不是非常昂贵的操作，但是在高并发条件下，开销就会上升得很快。缓存线程和表可以提高效率。

注 2：“打开表”的概念有一点点让人困惑。当不同的查询同时访问某个表，或者某个查询在子查询或者自联接中引用了一个表多次，MySQL 就会认为表被打开了很多次。MyISAM 的索引文件包含了一个计数器，当表打开的时候它的值就会增加，关闭的时候就会减少。这使 MyISAM 知道什么时候表没有被干净地关闭掉。如果它第一次打开一个表，但是计数器的值不是 0，那么就说明这个表没有被干净地关闭。



6.2.6 InnoDB 数据字典

The InnoDB Data Dictionary

InnoDB 自己有对每个表的缓存，叫做“表定义缓存”或者“数据字典”，它是不可配置的。当 InnoDB 打开一个表的时候，它就向字典中添加一个相应的对象。每个表可以使用 4KB 或更多的内存（MySQL 5.1 需要的空间要少得多）。当表关闭的时候，它不会被从字典中删除。

主要的性能问题，除了内存需求之外，就是为表打开和计算统计数据。这个操作是很昂贵的，因为它需要大量的 I/O 操作。和 MyISAM 相比，InnoDB 不会一直在表中保存统计数据。它在每次启动的时候都会重新计算。这个操作在当前版本的 MySQL 中使用了全局互斥量，所以它不能并行地操作。如果有许多表的话，服务器也许会需要几个小时来热身，在这期间，它也许除了等待 I/O 操作之外，不能做很多其他的事情。尽管没办法改变这一点，但是还是要知道这一点。

281 这通常只会在有大量（上千或上万）大型表的时候才有问题，它导致 I/O 密集型进程。

如果使用 InnoDB 的 `innodb_file_per_table` 选项（参阅 291 页“配置表空间”），那么对 InnoDB 任何时候能打开的.ibd 文件的数量还有另外一个限制。这是由 InnoDB 存储引擎处理的，而不是 MySQL 服务器，并且它受 `innodb_open_files` 的控制。InnoDB 打开文件的方式和 MyISAM 不一样。MyISAM 使用表缓存来保存打开表的文件描述符。InnoDB 为每个.idb 文件使用了全局文件描述符。如果可以的话，最好把 `innodb_open_files` 设置得足够大，这样服务器就可以保留所有同时打开的.idb 文件。

6.3 MySQL I/O 调优

Tuning MySQL's I/O Behavior

一些配置选项可以影响 MySQL 把数据同步到硬盘和进行恢复的方式。它们通常对性能有很大的影响，因为这其中牵涉了昂贵的 I/O 操作。它们也代表了性能和数据安全性的折中。通常说来，保证数据被立即而连续地写入磁盘的代价是很高的。如果愿意承担数据不能被真正地写入的风险，就可以增加并发并且减少 I/O 等待的时间，但是你需要决定自己到底能承受多大风险。

6.3.1 MyISAM I/O 调优

MyISAM I/O Tuning

让我们从 MyISAM 如何处理索引开始。MyISAM 通常在每次写入之后就会把索引的改变刷写到磁盘上。如果打算对一个表做很多改变，那么把它们组成一个批处理就会快很多。

做到这点的一种办法是使用 `LOCK TABLES`，它可以把写入延迟到对表解锁。这对提高性能是很有价值的技巧，因为它让你精确地控制可以延迟写入的查询及写入的时间。可以在语句中精确地定义要延迟的语句。

还可以使用 `delay_key_write` 变量来延迟索引的写入。如果使用了它，修改过的键缓冲区块只有在表关闭的时候才会被写入磁盘（注 3）。它有下面这些可能的选项：

注 3：表可以因为各种原因被关闭。比如，服务器可能会因为表缓存中没有足够的空间，或者有人执行了 `FLUSH TABLES` 命令，而关闭表。



OFF

MyISAM 在每次写入后就把键缓冲区中修改过的数据块刷写到磁盘上，除非表被 LOCK TABLES 锁住了。

ON

282

延迟键写入被开启，但是只针对使用 DELAY_KEY_WRITE 选项创建的表。

ALL

所有的 MyISAM 表都使用延迟键写入。

延迟键写入在某些情况下是有帮助的，但是它通常不能带来性能的飞跃。在数据量较小、读取命中率较好并且写入命中率较差的时候，它的用处最大。它也有一些缺点：

- 如果服务器崩溃并且数据块没有被刷写到磁盘上，索引就会损坏。
- 如果许多写入被延迟了，MySQL 就会需要更多的时间来关闭表，因为它要等待缓冲区被刷写到磁盘上。这在 MySQL 5.0 中会导致表缓存被长时间地锁住。
- FLUSH TABLES 可能会需要很长的时间，原因同上。这会导致 LVM 快照或其他备份操作运行 FLUSH TABLES WITH READ LOCK 需要更长的时间。
- 键缓冲区中未被刷写的数据块可能不会给将要从磁盘上读取的新块留出空间。因此，查询可能会因为等待键缓冲区释放空间而停止。

除了对 MyISAM 索引 I/O 进行调优，还可以配置 MyISAM 如何从损坏中恢复。myisam_recover 选项控制了 MyISAM 查找和修复错误的方式。可以在配置文件或命令行中设置这个参数。可以使用 SQL 语句参看这个选项的值，但不能对它进行修改（这儿没有写错。这个系统变量和相应的命令行参数的名字不一样）：

```
mysql> SHOW VARIABLES LIKE 'myisam_recover_options';
```

开启这个选项告诉 MySQL 在打开 MyISAM 表的时候检查它的错误，并且如果发现了错误，就尝试修复它。可以给它设置下面的值：

DEFAULT (或者不设置)

MySQL 会尝试修复所有被标记为崩溃及没有被标记为干净地关闭的表。默认设置除了恢复之外，不会做任何事情。和其他大部分变量不一样，DEFAULT 并不会把它设置为编译时给定的值，它只是表示“没有设置”。

BACKUP

283

让 MySQL 把数据文件备份到一个.BAK 文件中，可以方便随后进行检查。

FORCE

即使.MYD 丢失的数据多于一行，恢复也会继续。

QUICK

除非有被删除的数据块才跳过恢复。这些被删除的行仍然占据空间，并且可以被以后的 INSERT 语句复用。这是有用的，因为对大表进行 MyISAM 恢复通常需要很长的时间。

可以使用多重设置，中间用逗号隔开。比如，BACKUP、FORCE 将强制恢复并且创建备份。

我们推荐开启这些选项，特别是在仅有一些小的 MyISAM 表的时候。使用被损坏了的 MyISAM 表运行服务器是很危险的，因为它们有时候会损坏更多的数据，甚至让服务器崩溃。然后，如果有大表的话，自动恢复是不切实际的。当它们被打开的时候，就会导致服务器检查和修复所有的 MyISAM 表，这样效率太低。在修复的这段时间内，MySQL 会阻止所有的连接做任何事情。如果有许多 MyISAM 表，那么在启动之后使用 CHECK TABLES 或 REPAIR TABLES 可能会更好一些。这两种方式对于检查和修复表都是很重要的。

启动到数据文件的内存映射访问是另外一种有用的调优手段。内存映射是 MyISAM 能够通过操作系统的页面缓存直接访问到.MYD 文件，避免了代价较高的系统调用。在 MySQL 5.1 及以上版本中，可以使用 myisam_use mmap 选项打开内存映射。较老版本的 MySQL 只对压缩的 MyISAM 表使用了内存映射。

6.3.2 InnoDB I/O 调优

InnoDB I/O Flow

InnoDB 比 MyISAM 复杂得多。与之相对的是，不仅可以控制它如何恢复，还可以控制它如何打开表及刷写数据，它们极大地影响了恢复和总体性能。InnoDB 的恢复过程是自动的并且在 InnoDB 启动的时候总会运行，但还是可以影响它的行为。更多内容请参阅第 11 章。

先不考虑恢复，假设没有发生任何崩溃或错误，InnoDB 还是有很多值得配置的东西。它有复杂的链式缓冲区和文件，使它可以改进性能并且保证 ACID 属性，并且链上的每一个环节都是可配置的。图 6-1 显示了这些文件和缓冲区。

对于普通使用，一些很重要的配置是 InnoDB 日志文件大小、InnoDB 如何刷写日志缓冲区，以及 InnoDB 如何执行 I/O。

284

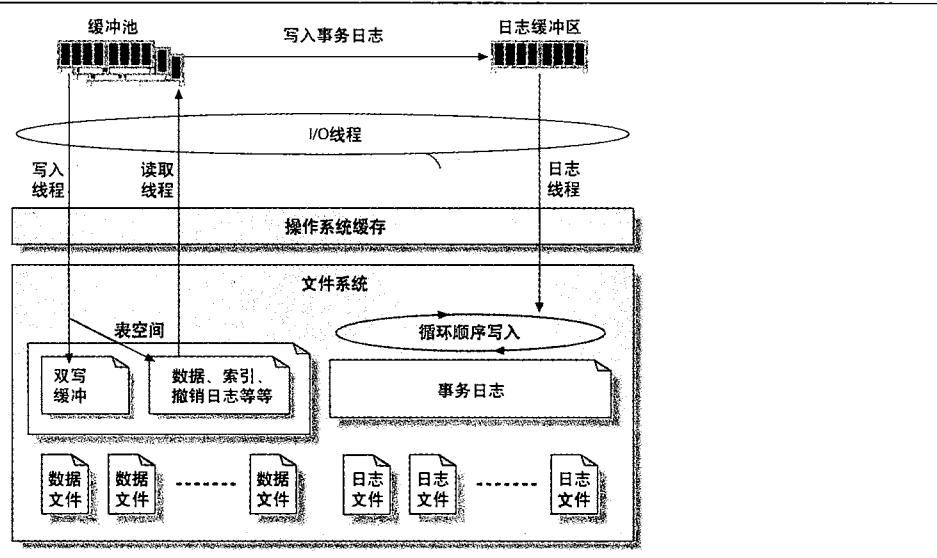


图 6-1：InnoDB 的缓冲区和文件

InnoDB 使用日志来减少提交事务的开销。它不是在每次事务提交的时候就把缓冲池刷写到磁盘上，而是记录了事务。事务对数据和索引做出的改变通常会被映射到表空间的随机位置，所以将这些改变写到磁盘上就会引起随机 I/O。作为一条原则，随机 I/O 比顺序 I/O 开销要高得多，因为它需要时间在磁盘上寻找正确的位置，并且还要等磁头移到相应的位置上。

InnoDB 使用自身的日志把随机 I/O 转换为顺序 I/O。一旦日志被记录到磁盘上，事务就是持久的了，尽管这时候改变还没有被写到数据文件中。如果一些坏事发生了（比如断电），InnoDB 可以回放日志并恢复提交了的事务。

当然，InnoDB 最终要把改变写到数据文件中，因为日志的大小是固定的。它以循环的方式写日志，当记录到达日志的底部，就会又从顶部开始。它不会覆盖改变没有被应用到数据文件的记录，因为这会消除提交的事务唯一持久性的记录。

InnoDB 使用后台线程智能地把改变写入到文件中。该线程可以把写入集中在一起，然后以效率更高的顺序写入的方式执行。实际上，事务日志把随机数据文件 I/O 转换为顺序日志文件和数据文件 I/O。把刷写工作变成后台进行可以让查询完成得更迅速，并且在查询负载很大的时候可以对 I/O 系统进行缓冲。285

日志文件总体大小由 `innodb_log_file_size` 和 `innodb_log_files_in_group` 控制，并且它们对写入的性能影响极大。这两个文件默认大小都是 5MB，总计为 10MB。对于高性能的负载，这个大小是不够的。日志文件总大小的上限是 4GB，但是即使是写入负载极高的查询也需要几百 MB（比如总共 256MB）。接下来的章节解释了如何为负载找到合适的大小。

InnoDB 用多个文件组成一个循环日志系统。通常不用改变默认的日志数量，只须改变每个日志文件的大小就可以了。为了改变日志文件的大小，先关闭 MySQL，然后移走旧日志，再重新配置大小，最后重新启动 MySQL 就行了。要确保干净地关闭 MySQL，否则日志文件中就有可能还保留着需要被应用到数据文件的记录。当重新启动服务器的时候要注意查看错误日志。在重新启动之后，就可以删除掉老的日志。

日志文件大小和日志缓冲。为了决定日志文件的理想大小，需要衡量通常数据改变的开销和崩溃时恢复的时间。如果日志太小，InnoDB 将会设置更多的检查点，并且导致更多日志写入。在极端情况下，写入查询有可能会停下来，等待日志上的记录被应用到数据文件上。另一方面，如果日志太大，InnoDB 在恢复的时候可能就会做大量的工作。它会极大地增加恢复的时间。

数据大小和访问模式也会影响恢复时间。假设有 1TB（译注 1）数据并且有 16GB 缓冲池，总日志是 128MB。如果在缓冲池中有许多不干净的页面（比如，更改没有被应用到数据文件的页面），并且它们在整个 1TB 数据中均匀分布，那么从崩溃中恢复就会花很长时间。InnoDB 将会不得不扫描日志、检查数据文件，并且按照需要把改动应用到文件上。这意味着大量的读写！在另一方面，如果更改是局部化的，比如说只有几 GB 数据经常被改动，恢复就会很快，即使数据和日志文件很大也是这样。恢复时间也依赖于典型更改的大小，它和数据行的平均长度有关。较短的行可以在日志中存储更多的改动，所以 InnoDB 在恢复的时候就会回放更多的改动。

在 InnoDB 改变数据的时候，它会把这次改动的记录写到日志缓冲里面。日志缓冲被保存在内存中。缓冲写满、事务提交或每一秒钟，不管那种情况先发生，InnoDB 都会把缓冲区写到磁盘上的日志文件中。如果有大型事务，就可以增加缓存文件（默认是 1MB）来减少 I/O 动作。控制缓冲大小的变量叫 `innodb_log_buffer_size`。286

译注 1：1024GB。



不需要把缓冲区变得很大。推荐值是 1MB 到 8MB。除非要写入大量的巨型 BLOB 记录，否则这个大小就足够了。日志相对 InnoDB 的正常数据要紧凑得多。它们不是基于页面的，所以它们不会在存储数据的时候浪费整个页面。InnoDB 也会使日志记录尽可能短。它们有时甚至会被像 C 函数的函数名和参数那样存储。

可以通过检查 `SHOW INNODB STATUS` 命令的 `LOG` 部分检测 InnoDB 的日志和日志缓冲 I/O 性能，还可以通过观察 `Innodb_os_log_written` 的值了解 InnoDB 向日志文件写入了多少数据。一个好的法则就是观察 10 秒到 100 秒时间间隔内的数据，并且注意最大值。可以使用这个值来判断日志缓冲大小是否合适。例如，如果最大数据是每秒写入 100KB，那么 1MB 的日志缓存可能就足够了。

也可以使用这个指标来决定日志文件的合适大小。如果最大值是每秒 100KB，256MB 日志文件就可以存储至少 2560 秒的日志记录，这很可能足够了。参阅第 565 页的“显示 INNODB 状态”了解如何监控和解释日志及缓冲区的状态。

InnoDB 如何刷写日志缓冲。当 InnoDB 把日志缓冲刷写到磁盘上的日志文件时，它会用一个互斥量锁定缓存，把缓存写到应有的位置，然后把剩下的记录移到缓存的前端。一种可能性就是当互斥量被释放时，不止一个事务打算刷写自己的日志。InnoDB 有群体提交特性，它可以利用一次 I/O 操作把所有的请求提交到日志中。但是对于 MySQL 5.0，如果二进制日志被激活，这个功能就失效了。

日志缓冲区必须被刷写到持久性存储中，以保证提交了的事务能完全持久化。如果比起持久性，更在意性能，就可以改变 `innodb_log_at_trx_commit` 的值来控制日志缓存被刷写到什么地方及刷写的频率。可能的设置如下：

- 0 把日志缓存写到日志文件中，并且每秒钟刷写一次，但是在事务提交的时候不进行任何动作。
- 1 将日志缓冲写到日志文件中，并且在事务提交的时候把缓存刷写到持久性存储中。这是默认（最安全）的设置。它保证不会遗失任何提交的事务，除非磁盘或操作系统“假冒”了刷写操作。
- 2 在每次提交的时候把日志缓冲写到日志文件中，但是不进行清理。InnoDB 安排每秒钟清理一次。它和 0 最大的区别（这也使得 2 更好）是 2 在 MySQL 进程崩溃的时候不会丢失任何事务。但是如果日志服务器崩溃或失去电力，还是有可能丢失事务。

187

重要的是知道把日志缓冲写入日志文件和把日志刷写到持久性存储中的区别。在大多数操作系统中，把缓冲写入日志只是简单地数据从 InnoDB 的内存缓冲区移到操作系统的缓存中，它也在内存中。它实际上不会把数据写入持久性存储中。因此，设置 0 和 2 通常导致在崩溃或电力故障的时候最多失去一秒钟的数据，因为这时数据可能只在操作系统的缓存中。之所以说“通常”，原因是 InnoDB 会每秒钟把日志文件写到磁盘上一次，但是在某些情况下也有可能失去多于 1 秒的数据，例如刷写被停住了。

相反的是，将日志刷写到持久性存储中意味着 InnoDB 要求操作系统把数据刷到缓存外部并且确保写入到磁盘上。这会阻止 I/O 调用，直到数据被完全写入。由于将数据写到磁盘比较慢，所以当 `innodb_flush_log_at_trx_commit` 被设置到 1 时，它会显著地降低 InnoDB 每秒可以提交的事务。现在的高速驱动器（注 4）每秒只能执行几百次真正的磁盘事务，这是由驱动器旋转和寻址时间所决定的。

有时硬盘控制器或操作系统会假冒清写动作，它会把数据放入另外一个缓存中，比如磁盘自己的缓存。这样做速度较快但是很危险，因为数据在驱动器失去电力的时候还是会丢失。这甚至比把 `innodb_flush_log_at_trx_commit` 设置为 1 之外的值更糟糕，因为它不仅仅是导致丢失事务，还能导致数据损坏。

注 4：我们谈论的是普通的磁碟式硬盘，而不是固态硬盘。固态硬盘的性能特点完全不一样。

把 `innodb_flush_log_at_trx_commit` 设置为 1 之外的值能导致丢失事务。但是，如果不在意持久性(Durability ACID 中的 D)，把它设置为其他值也是有用的。也许你只需要 InnoDB 的其他特性，比如聚集索引、防止数据损坏及行级锁。完全为了性能原因使用 InnoDB 代替 MyISAM 也是常见的。

高性能事务的最佳配置是把 `flush_log_at_trx_commit` 设置为 1，并且将日志文件放在有备用电池的写入缓存的 RAID 上。这不仅安全，速度也很快。更多 RAID 的内容请参阅第 317 页的“RAID 性能优化”。

InnoDB 如何打开并清写日志和数据文件

288

`innodb_flush_method` 选项让你可以配置 InnoDB 实际与文件系统进行交互的方式。除了写数据之外，它还可以影响 InnoDB 如何读取数据。Windows 和非 Windows 上这个选项的值是互斥的。在 Windows 系统上，只能使用 `async_unbuffered`、`unbuffered` 和 `normal`，不能使用其他任何值。在 Windows 系统上的默认值是 `unbuffered`，其他系统是 `fdatasync`。（如果 `SHOW GLOBAL VARIABLES` 显示该变量的值为空，那就意味着它被设置成了默认值。）



警告：改变 InnoDB 如何执行 I/O 操作会极大地影响性能，要仔细地评测它们。

下面是一些可能的值：

`fdatasync`

它是非 Windows 系统上的默认值。InnoDB 使用 `fsync()` 函数来刷写数据和日志文件。

InnoDB 通常使用 `fsync()` 来代替 `fdatasync()`，尽管它们的意思好像是相反的。`fdatasync()` 和 `fsync()` 相似，除了它只刷写文件的数据，而不包括元数据（最后修改时间等）。因此，`fsync()` 能导致更多的 I/O 操作。但是 InnoDB 的开发人员非常保守，并且他们发现 `fdatasync()` 在某些情况下会导致崩溃。InnoDB 决定哪种方法能被安全地使用，一些选项在编译时就被设置好了，有些在运行时进行设置。它会使用最快的方法。

使用 `fsync()` 的缺点就是操作系统至少会缓存一些数据在自己的缓存中。在理论上，这是有点浪费的双缓冲，因为 InnoDB 能比操作系统更智能地管理自己的缓冲区。但是，最终的影响和系统及文件系统非常相关。如果双缓冲可以让文件系统更智能地做 I/O 计划和进行批处理，那它也不是什么坏事。一些文件系统和操作系统能累计写入并一起执行它们，还能按照效率进行排序，或者并行地把它们写到多个设备中。它们也会做读取前的优化，比如指导磁盘预读下一个顺序块。

有时这些优化是有帮助的，但是有时又不是。如果好奇自己机器上的 `fsync()` 到底是什么版本，可以使用 `fsync(2)` 读取系统的指南页 (Manpage)。

`innodb_file_per_table` 导致每个文件都被单独地使用了 `fsync()` 函数，这意味着想多个表写入不能被合并到单个 I/O 操作中。这可能需要 InnoDB 执行较多的 `fsync()` 操作。

`0_DIRECT`

289

InnoDB 根据系统和数据文件使用 `0_DIRECT`，或者 `directio()` 标志。该选项不会影响日志文件并且不是在所有的 Unix 系统上都可用。至少 Linux、FreeBSD 和 Solaris (5.0 及以上版本) 支持它。和 `0_DSYNC` 标志不同，它会影响读写。这个设置还是使用 `fsync()` 把文件刷写到磁盘，但是它告诉操作系统不要缓存数据，也不要使用提前读取。它完全禁止了操作系统的缓存并且使所有的读写动作直接到存储设备，避免了



双缓冲。

在大多数系统上，它通过调用 `fcntl()` 在文件描述符中设置 `O_DIRECT` 标志。所以可以通过 `fcntl(2)` 读取指南页了解详细情况。在 Solaris 上，该选项使用了 `directio()`。

该设置不能禁止 RAID 卡的提前读取功能。它只能禁止操作系统或文件系统的提前读取功能。

在使用 `O_DIRECT` 的时候通常不应该禁止 RAID 卡的写入缓存，因为这是保持良好性能的唯一途径。当在 InnoDB 和实际存储设备之间没有缓存的时候使用 `O_DIRECT` 会极大地降低性能。

这个设置能导致服务器热身时间增加，特别是操作系统缓存非常巨大的时候。它也能使小缓存池（例如，默认大小的缓存池）比缓冲过的 I/O 慢很多。这是因为操作系统通过把数据保存在自己的缓存中以“帮助”数据移出。如果需要的数据不在缓存池中，InnoDB 将会直接从磁盘上读取它。

该设置不会对使用 `innodb_file_per_table` 有更多的危害。

`O_DSYNC`

该选项在对日志文件调用 `open()` 使用了 `O_SYNC` 标志。这使所有的写入都是同步的，换句话说，写入动作只有在数据被写入磁盘的时候才会返回。该选项不会影响数据文件。

在 `O_SYNC` 和 `O_DIRECT` 标志之间的区别是 `O_SYNC` 不会在操作系统层面禁止缓存。因此，它不会避免双缓冲，并且它不会直接写入磁盘。使用 `O_SYNC`，写入修改了缓存中的数据，然后把它发送到磁盘上。

尽管使用 `O_SYNC` 同步写入听上去很像 `fsync()`，但是它们在操作系统和硬件层面的实现是很不一样的。当使用 `O_SYNC` 的时候，操作系统会把“使用同步 I/O”的标志传递到硬件层，告诉设备不要使用缓存。在另一方面，`fsync()` 告诉操作系统把修改过的缓存写到设备上，然后告诉设备如果可能的话就清理掉自己的缓存。所以它确保数据已经被记录到了物理介质上。另外一个区别就是使用 `O_SYNC`，每个 `write()` 或者 `pwrite()` 操作在完成之前，都会阻塞调用过程。相反地，不使用 `O_SYNC` 标志调用 `fsync()` 会在缓存中累计写入（这使每个写入更快），然后再一次性地进行刷写。

还有，该选项设置了 `O_SYNC` 标志，而不是 `O_DSYNC` 标志，因为 InnoDB 开发人员发现了使用 `O_DSYNC` 的问题。`O_SYNC` 和 `O_DSYNC` 类似于 `fsync()` 和 `fdatasync()`：`O_SYNC` 同步数据和元数据，但是 `O_DSYNC` 只同步数据。

`Async_unbuffered`

这是 Windows 系统的默认值。该选项让 InnoDB 对大部分写入都使用无缓冲的 I/O。例外情况就是它在 `innodb_flush_log_at_trx_commit` 被设置到 2 的时候对日志文件使用了有缓冲的 I/O。

该选项使 InnoDB 在 Windows 2000、Windows XP 及以上版本中对读写都使用操作系统原生的异步（层叠）I/O。在较老版本的 Windows 上，InnoDB 使用自己的异步 I/O，它使用了线程。

`Unbuffered`

只适用于 Windows。该选项和 `async_unbuffered` 类似，但是不使用原生异步 I/O。

`Normal`

只适用于 Windows。该选项让 InnoDB 不使用原生异步 I/O 或者无缓冲 I/O。

Nosync and littlesync

仅能用于开发。这两个选项没有文档支持，对产品服务也不安全，不应该使用它们。

如果你的 RAID 控制器的写入缓存有备用电池，我们推荐使用 `O_DIRECT`。如果不是，默认值或 `O_DIRECT` 都有可能是最佳选择，这依赖于应用程序。

可以配置 Windows 中 I/O 线程的数量，但是在其他平台上都不行。把 `innodb_file_io_threads` 设置为大于 4 将会导致 InnoDB 为数据 I/O 创建更多的读写线程。一般只有 1 个插入缓冲线程和 1 个日志线程。所以如果它的值是 8 的话，那就有 1 个插入缓冲线程、1 个日志线程、3 个读取线程和 3 个写入线程。

InnoDB 表空间

InnoDB 把数据保存在表空间中。表空间实际上是跨越了磁盘上的一个或多个文件的虚拟文件系统。InnoDB 出于很多考虑使用了表空间，而不仅仅是为了存储表和索引。它保留了自己的撤销日志（老的数据行的版本）、插入缓存、双写缓存（下一节将会介绍），以及表空间的其他内部结构。201

配置表空间。可以使用 `innodb_data_file_path` 定义表空间文件。这些文件都在 `innodb_data_home_dir` 定义的目录中，下面是一个例子：

```
innodb_data_home_dir = /var/lib/mysql/  
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

上面的例子在 3 个文件中创建了 3GB 的表空间。有时人们在想是否可以跨越多个驱动器使用文件来分摊负载，就像这样：

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

这确实把文件放在了不同的目录中，本例中是不同的驱动器。InnoDB 实际上只是把这些文件连接了起来。因此，用这种办法不会有什么好处。InnoDB 会先把第 1 个文件写满，然后是第 2 个，依次类推。负载其实并没有被分摊开。RAID 控制器能更智能地分摊负载。

为了使表空间在空间耗尽之后可以自动增长，可以用下面的命令让最后一个文件自动延伸：

```
...ibdata3:1G:autoextend
```

默认行为是创建 10MB 的单个延伸文件。如果让文件可以自动延伸，一种不错的方式就是给表空间的值设定一个上限，使之不会增长得过大，因为它一旦长大了，就不会自动减小。例如，下面的例子限制延伸文件最大为 2GB：

```
...ibdata3:1G:autoextend:max:2G
```

管理单个表空间也许是一件费力气的事情，尤其是它能自动增长，而你又想回收空间（出于这个原因，我们建议你禁止自动延伸）。回收空间的唯一办法就是将数据进行转储、关闭 MySQL、删除所有的文件、改变配置、重新启动、让 InnoDB 创建新文件并且恢复数据。InnoDB 对表空间的控制是很严的，不能简单地移除文件或改变大小。如果表空间崩溃了的话，InnoDB 就不会启动。这和它对日志文件的严格限制一样。如果你习惯了在 MyISAM 中随便移动数据，在这儿可要当心些。

在 MySQL 4.1 及以上版本中，`Innodb_file_per_table` 选项使 InnoDB 为每一个表使用一个文件。它在数据库目录中以“表名.ibd”保存数据。这使得删除表后回收数据变得比较容易，并且它对于把表分布到多个磁盘上也

292

很有用处。但是，将数据放在多个文件中能导致浪费更多的存储空间，因为它把单个 InnoDB 表空间的碎片都放在了.ibd 文件中。

这对于很小的表尤其会成为一个问题，因为 InnoDB 的页面大小是 16KB。即使表只有 1KB 数据，它也会需要至少 16KB 的磁盘空间。

即使开启了 `innodb_file_per_table` 选项，还是需要为撤销日志和其他系统数据定义主表空间。（如果没有在里面保存所有的数据，它会比较小。但是关闭自动延伸还是一个好主意，因为只有重新加载数据才能减少文件大小。）同样，不能简单地通过拷贝文件来移动、备份或恢复表。也有可能做到这一点，但是需要额外的步骤，并且肯定不能在服务器之间拷贝数据。更多详情请参阅第 500 页的“恢复原始文件”。

一些人喜欢使用 `innodb_file_per_table`，仅仅是因为它的可管理性和可见性。例如，通过检查文件得到表的大小比通过 `SHOW TABLE STATUS` 要快得多，后者需要锁定表并且扫描缓冲池，以了解有多少页面被分配给了表。

我们也应该注意到实际上不用把 InnoDB 文件保存在传统的文件系统中。和很多传统的数据库服务器一样，InnoDB 提供了使用原始设备，比如未格式化的分区来存储数据的能力。然而，现在的文件系统能高效地处理大型文件，所以没有必要使用这种能力。使用原始设备可能会把性能提高几个百分点，但是我们认为这点性能提升可以弥补不能按照文件来操作数据的缺点。当把数据保存在原始分区（Raw Partition）中时，不能对它使用 `mv`、`cp` 或者其他的工具。我们也考虑了快照能力，比如由 GNU/Linux 的逻辑卷管理器（LVM，Logical Volume Manager）提供的功能，有很大的好处。可以把原始设备放到逻辑卷中，但这违反了初衷，因为这已经不是真正的原始设备了。最后要说的是，由使用原始设备带来的一点点性能提升并不划算。

旧数据版本和表空间。 InnoDB 的表空间在写入负荷很重的环境中会增长得很大。如果事务长时间处于打开状态（即使它没有做任何工作），并且它们正在使用默认的 REPEATABLE READ 事务隔离层，InnoDB 将不能删除老的数据，因为未提交的事务还需要它们。InnoDB 将老的数据保存在表空间中，所以当更多的数据被更新时，它就会继续增长。有时问题不是出在没有提交的事务上，而是出在工作负载上：清理进程是单线程的，它跟不上需要被清理的老数据的数量。

在这两种情况下，`SHOW INNODB STATUS` 的输出有助于锁定问题。查看 `TRANSACTIONS` 的第 1 和第 2 行，它会显示当前事务的数量及指出哪个的清理工作已经完成了。如果它们的区别很大，那么就说明有许多没被清理的事务，下面是个例子：

```
-----  
TRANSACTIONS  
-----  
Trx id counter 0 80157601  
Purge done for trx's n:o < 0 80154573 undo n:o < 0 0
```

293

事务标识符是一个由两个 32 位整数组成的 64 位整数，所以需要做一点点数学计算来了解它们的区别。在这个例子中，计算非常容易。数字的高位都是 0，所以就有 $80157601 - 80154573 = 3028$ 个潜在的未被清理的事务（`Innotop` 可以帮你做这个数学计算）。这儿说“潜在的”，是因为大的差别并不一定是指有很多未被清理的行。只有改变了数据的事务才会创建旧数据，有可能很多事务没有改变数据（同样，有可能一个事务就改变了很多数据）。

如果有很多未被清理的事务并且表空间因它而增长，就可以强制 MySQL 变慢，以使清理线程能跟上数据的变化。这听上去不怎么好，但是也没什么替代的办法。否则，InnoDB 就会不停地写入数据并且填充磁盘，直到耗尽磁盘空间或让表空间达到规定的上限。



为了减缓写入，可以把 `innodb_max_purge_lag` 变量设置为 0 之外的值。这个值表示在等待清理的事务的最大数量，一旦超过这个值，InnoDB 就会延迟更新数据的事务，要知道自己的负载才能决定这个值的最佳大小。举个例子，如果事务平均影响 1KB 数据，并且能容忍 100MB 未清理的数据，那么这个值就是 1 000 000。

要记住未被清理的行会影响所有的查询，因为它会让表和索引很快变大。如果清理线程跟不上节奏，性能就会大打折扣。设置 `innodb_max_purge_lag` 也会降低性能，但是它的危害比前者小。

双写缓存

InnoDB 在对页面进行部分写入的时候使用了双缓冲，以防止数据损坏。部分写入发生在磁盘写入没有全部完成，并且只有 16KB 页面的一部分被写入的时候。有许多原因（崩溃、缺陷等）会导致数据被部分写入。双写缓存在这种情况下保护了数据。

双缓冲是表空间中一个特殊的保留区域，大小足够在一个连续块中容纳 100 个页面。它在本质上是最近写入页面的备份。当 InnoDB 把页面从缓冲池中清写到磁盘上时，它会先把它们写入（或者清写）到双缓冲中，然后再写入到真正的地方。这确保每次写入的原子性和可持续性。

这不是意味着每个页面都被写了两次吗？是的，确实是两次，但是因为 InnoDB 顺序地把一些页面写入双缓冲，然后再调用 `fsync()` 把它们同步到磁盘上，所以对性能的影响比较小——通常是几个百分点。更重要的是，这个策略使日志文件高效得多。因为双写缓冲为 InnoDB 保证数据不受破坏提供了强有力的保障，所以 InnoDB 的日志记录就不需要包含完整的页面，而更像是页面的二进制增量数据。24

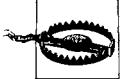
如果有部分页面被写入了双写缓存，那么原始页面还将继续留在磁盘的本来位置上。当 InnoDB 恢复的时候，它会使用原始页面来代替双写缓存中被破坏的页面。但是，如果双写缓存成功并且写到页面的真实位置失败了，InnoDB 就会在恢复的时候使用双写缓存中的拷贝。InnoDB 知道每个被损坏了的页面，因为每个页面的最后都会有一个校验码。校验码最后才会被写入页面。所以如果页面的内容和校验码不匹配，那么就说明页面已经被损坏了。因此，在恢复的时候，InnoDB 只是读取双写缓存中的页面并且验证校验码。如果页面的校验码不正确，它就会从原始位置读取页面。

在某些情况下，双写缓存并不是必需的，例如在从服务器上可以禁用它。同时，一些文件系统（比如 ZFS）自身会做同样的事情，那么 InnoDB 就不用做重复的工作了。可以通过把 `innodb_doublewrite` 设置为 0 禁用双写缓存。

另外的 I/O 调优

`sync_binlog` 选项控制了 MySQL 如何把二进制日志刷写到磁盘。它的默认值是 0，这意味着 MySQL 不会执行任何刷写操作，并且何时把日志写到持久性存储设备上取决于操作系统。如果该值大于 0，它就规定了在把二进制日志刷写到磁盘期间可以运行多少次写入（如果 `autocommit` 被设置为 1，那么每次写入就是单条语句，否则就是一个事务）。一般很少把它设置为 0 或 1 之外的值。

如果没有把 `sync_binlog` 设置为 1，那么就可能导致二进制日志和事务的数据不同步。这会破坏复制并且使按时间点进行恢复变得不可能。然而，把它设置为 1 带来的安全性需要很高的代价。同步二进制日志和事务日志要求 MySQL 把两个文件刷写到两个不同的磁盘位置上。这会导致相对较慢的磁盘搜索。



警告：如果在 MySQL 5.0 及更高的版本上使用二进制日志和 InnoDB，并且尤其是在从低版本升级到高版本的情况下，那么就要特别小心 XA 事务这一新特性。它被设计成用于在不同存储引擎和二进制日志之间同步事务提交，但是它也禁用了 InnoDB 的群体提交功能。这会极大地降低性能，因为提交事务的时候可能会导致大量的 `fsync()` 调用。可以通过禁用二进制日志和把 `innodb_support_xa=0` 设置为 0 禁用 XA 支持来解决这个问题。如果使用了有备用电池的 RAID 缓存，每个 `fsync()` 调用都会更快，所以这可能不会成为一个问题。

和 InnoDB 日志文件一样，把二进制日志文件放入有备用电池的 RAID 卷会极大地提高性能。

有一点和性能无关的提示：如果想使用 `expire_logs_days` 选项自动地删除老的二进制日志，就不要使用 `rm` 来删除它们。否则服务器会弄不清楚到底是哪一个起作用，于是就不会自动地删除它们，并且 `PURGE MASTER LOGS` 也会停止工作。在发现自己有这个问题的时候，解决的办法就是手动地使用磁盘上的文件列表和 `hostname-bin.index` 文件进行重新同步。

在第 7 章会深入讨论 RAID，但是在这儿有一点值得重申，那就是质量优良的 RAID 控制器，在具有备用电池的写入缓存并且使用了回写策略的情况下，每秒可以处理上千次写入，并且还能提供其他存储服务。数据被写入到了有电池的缓存中，所以即使系统失去了电力，数据也不会丢失。当电力供应恢复的时候，RAID 控制器就会在磁盘可用之前把数据写入磁盘。因此，有足够写入缓存的 RAID 控制器能极大地提高性能并且这会是很划算的投资。

6.4 MySQL 并发调优

Tuning MySQL Concurrency

当 MySQL 在高并发条件下工作的时候，你可能会遇到以前未曾遇到过的性能瓶颈。接下来的章节解释了如何检测问题，并且如何在 InnoDB 和 MyISAM 遇到高并发的时候得到最佳性能。

6.4.1 MyISAM 并发调优

MyISAM Concurrency Tuning

需要很仔细地控制同时进行的读写，以避免读取到不连续的数据。MyISAM 在某些条件下允许并发插入和读取，并且它让你可以“调度”某些操作，以尽可能少地阻止工作。

在了解 MyISAM 的并发设置之前，重要的是要了解 MyISAM 如何删除和插入行。删除操作不会重新安排整个表，它们只是把行标记为已经删除，并且在表中留下了一些“洞”。MyISAM 在可能的情况下会优先使用这些“洞”，为插入复用空间。如果表是完整的，它就会把新的行拼接在表的最后。

即使 MyISAM 有表级别的锁，它也能在读取的同时把行拼接到表尾。它通过禁止读取最后一行做到了这一点。这避免了不连续的读取。

但是，当表中间的数据改变的时候，要提供连续读取就困难得多。MVCC 是最通用的解决这个问题的办法：它在创建新版本数据的同时提供老版本数据读取。

MyISAM 不支持 MVCC，所以它只有在到达表尾的时候才允许并发插入。

可以使用 `concurrent_insert` 变量配置 MyISAM 的并发插入行为，它有下面的值：

- 0 MyISAM 不允许并发插入，每一次插入都会把表锁住。
- 1 默认值。只要表中没有空缺，MyISAM 就允许并发插入。
- 2 该值在 MySQL 5.0 及更高的版本可用。它强制并发插入到表尾，即使表有空缺也不例外。如果没有线程从表中读取数据，MySQL 就会把新数据插入到空缺中。使用了该设置，表的碎片会增多，所以就需要更经常地对表进行优化。

可以配置 MySQL 把一些操作延迟，然后合并到一起执行。例如，可以使用 `delay_key_write` 延迟写入索引。这会带来一些明显的矛盾：立即写入索引（安全但是代价很高），或者等待写入并希望在写入前不要断电（更快，但是如果断电的话就会导致大规模的索引损坏，因为索引文件已经明显过期了）。也可以使用 `low_priority_updates` 让 `INSERT`、`REPLACE`、`DELETE` 及 `UPDATE` 比 `SELECT` 的优先级更低。这等同于全局地给 `UPDATE` 使用 `LOW_PRIORITY` 修饰符。更多内容请参阅第 195 页的“查询优化器提示”。

最后，尽管 InnoDB 的扩展性问题被提及的很多，但 MyISAM 实际在很长时间内都有互斥量的问题。在 MySQL 4.0 和更早版本中，有全局互斥量保护键缓冲的任何 I/O 行为，它在多 CPU 和多磁盘的时候带来了扩展性问题。MySQL 4.1 的键缓冲代码已经被改进过，并且不再有这个问题，但是它还是为每一个键缓冲保持了一个互斥量。当一个线程从键缓冲把键数据块拷贝到本地存储中，而不是从磁盘读取时，它会成为一个问题。磁盘的瓶颈已经没有了，但是在访问键缓冲中的数据时瓶颈依然存在。有时可以使用多个键缓冲绕过这个问题，但是这种方法并不总是奏效。例如，在只包含了一个索引的时候就没办法解决这个问题。这样的话，多 CPU 机器上的并发 `SELECT` 查询会比单 CPU 的机器慢很多，即使只有这一个查询在运行也是如此。

6.4.2 InnoDB 并发调优

InnoDB Concurrency Tuning

InnoDB 是为高并发设计的，但它并不完美。InnoDB 的结构仍然基于有限内存、单 CPU 和单磁盘系统。InnoDB 某些方面的性能在高并发条件下下降得很快，并且唯一的解决办法就是限制并发。可以经常检查 `SHOW INNODB STATUS` 输出中的 `SEMAPHORES` 部分来确认是否发生了并发问题。更多内容请参阅第 566 页的“SEMAPHORES”。

297

InnoDB 用自己的“线程调度”程序来控制线程如何进入 InnoDB 的内核访问数据，以及一旦进入内核之后可以执行的动作。控制并发最基本的方式是使用 `innodb_thread_concurrency` 变量，它限制了一次有多少线程能进入内核。0 表示不限制进入内核的数量。如果有 InnoDB 并发问题，该变量是最重要的配置项。

没有办法为所有的架构和工作负载确定最佳的并发数量，在理论上，可以用下面的公式进行计算：

$$\text{并发} = \text{CPU 的数量} \times \text{磁盘的数量} \times 2$$

但是在实际中，使用更小的值往往会更好。为了知道适合自己系统的最佳值，需要进行试验和测试。

如果内核中已经有了允许数量的线程，那么线程就不能再进入内核了。InnoDB 采用了一种两阶段的过程来保证线程可以尽可能高效地进入内核。这种策略减少了操作系统引起的上下文切换带来的开销。线程首先睡眠 `innodb_thread_sleep_delay` 所规定的微秒数，然后再次进行尝试。如果还是不能进入，它就会进入一个等待线程的队列中并且把控制权交给操作系统。



第一阶段默认的睡眠时间是 10 000 微秒。当有很多线程都处于“正在等待进入队列”这一状态时，改变这个值有助于高并发性系统。默认值在有大量小查询的时候会太大了，因为它给查询增加了 10 毫秒延时。

一旦线程进入了内核，它就会得到一个确定的数字作为“凭据”，它再次进入内核的时候，就不会再进行任何的并发检查。该数字限定了它再次回到等待队列之前能做多少工作。`Innodb_concurrency_tickets` 选项控制了凭据的数量。除非有大量的运行极长时间查询，否则我们极少改动这个选项。凭据只是为每个查询授权，而不是为每个事务授权。一旦查询结束，凭据就会被丢掉。

除了缓冲池和其他结构的瓶颈，在提交阶段还有另外一种形式的并发瓶颈，也就是刷写操作造成的密集 I/O 操作。`Innodb_commit_concurrency` 变量决定了某一时刻有多少线程能进行提交。当 `innodb_thread_concurrency` 被设置到了一个较低的值，造成大量线程状况不佳时，设置该选项会有帮助。

InnoDB 团队正在解决这些问题，在 MySQL 5.0.30 和 MySQL 5.0.32 中有了较大的改进。

298 6.5 基于工作负载调优

熟悉你的服务器的工作负载

对服务器调优的最终目的是按照特定的工作负载进行定制。这不仅仅需要了解查询，还需要深刻了解服务器各种活动的数据、类型及频率，以及其他的一些活动，比如到服务器的连接和刷写数据表。还需要了解如何监控和解释 MySQL 以及操作系统的行为和状态。更多内容可以参阅第 7 章和第 14 章。

第一件要做的事情就是熟悉服务器。了解运行在它上面的查询。使用 `innotop` 或其他工具对它进行监控。这不仅仅有助于知道服务器的总体情况，也有助于知道查询正在做什么。获取这种知识的一种方式就是使用一个脚本 (`Innotop` 内建了该功能) 聚合 `SHOW PROCESSLIST` 输出中 `COMMAND` 栏的内容，或者就用肉眼进行观测。要找到在某种特殊情况下花费了大量时间的线程。

在服务器满负荷运行的时候试着查看一下进程列表是了解何种类型的查询遇到了最大困难的最佳时机。例如，有许多查询把结果拷贝到了临时表吗？或者正在对结果进行排序？如果是的话，那么就应该检查临时表和排序缓冲的配置（也许也要优化查询）。

我们常常推荐我们为 MySQL 日志开发的补丁，它能显示查询所做的事情的大量信息，并且还能让你更详细地分析负载。这些补丁被包含在 MySQL 官方发布中，所以它们可能已经在你的机器中了。具体内容请参阅第 65 页的“更好地控制日志”。

6.5.1 优化 BLOB 和 TEXT 负载

优化 BLOB 和 TEXT 的方法

BLOB 和 TEXT 对于 MySQL 来说是特殊的负载（为了简单起见，我们把 BLOB 和 TEXT 统称为 BLOB，因为它们属于同一种数据类型）。BLOB 有一些局限，所以服务器不得不特殊地处理它们。一个最重要的考虑就是服务器不能为 BLOB 使用内存中的临时表。因此，如果某个查询需要为 BLOB 值使用数据表，不管它是否很小，都会在磁盘上进行。这效率很低，尤其是对小而快的查询而言，临时表可能是查询开销中最大的部分。

有两种办法可以解决这个问题：(1) 利用 `SUBSTRING` 函数把值转换为 `VARCHAR`（具体内容请参阅第 84 页的“字符串类型”）；(2) 让临时表运行得更快。



这减少了一些开销，但仍然比使用内存中的表慢很多。使用基于内存的文件系统有一些帮助，因为操作系统会避免向磁盘写入数据（注 5）。普通的文件系统也被缓存到了内存中，但是操作系统会几秒钟就向磁盘写一次数据。Tmpfs 文件系统永远都不会刷写数据。它也是为较低开销和简单性而设计的。例如，这种文件系统不需要为恢复做准备，这使它运行得更快。

控制临时表位置的服务器设置是 tempdir。要监控文件系统被写满的程度，以保证临时表有足够的空间。如果需要，可以定义几个临时表位置，MySQL 会循环使用它们。

如果 BLOB 列非常大，并且使用的是 InnoDB，那么就应该增加 InnoDB 的缓冲区大小。本章前面的节有更多这方面的内容。

对于很长的长度可变的列（比如，BLOB、TEXT 和很长的字符列），InnoDB 会在页面中存储一个 768 字节的前缀（注 6）。如果列的值大于前缀的长度，InnoDB 可能就会分配外部的存储空间来保存其余的数据。InnoDB 会为数据分配 16KB 大小的整个页面，并且每一个列都会有自己的页面（列不会共享页面）。InnoDB 一次为一个列分配一个页面，直到页面达到 32 个，然后它就会一次性分配 64 个页面。

注意到我们只是说 InnoDB 可能会分配外部存储空间。如果行的总长度，包括长列的完整值，小于 InnoDB 的最大行长度（稍小于 8KB），那么即使长列的值超过了前缀长度，InnoDB 也不会分配外部存储空间。

最后，当 InnoDB 更新外部存储空间中的长列时，它不会对它进行原地更新。相反地，它会把值放到外部存储空间中的新地方，然后把老的数据删除。

这些行为会造成下面的后果：

- 长列可能会浪费很多 InnoDB 空间。例如，如果列的值只比行的最大长度多一个字节，InnoDB 也会用整个页面来存储其余的数据，这样就浪费了页面的大部分空间。同样地，如果值大小稍大于 32 个页面，它在磁盘上就实际会用 96 个页面。
- 外部存储禁用了适应性哈希索引，它需要比较列的完全长度来验证数据正确性。（哈希让 InnoDB 能很快地找到自己“猜测”的结果，但是它必须检测自己的“猜测”是否正确。）因为适应性哈希索引全部在内存中并且它直接从缓存池中最经常被访问的数据上构造出来，它不能使用外部存储。
- 长值会让无法使用索引的有 WHERE 子句的查询运行缓慢。MySQL 在应用 WHERE 子句之前会读取所有的列，所以它可能会让 InnoDB 读取大量的外部存储，然后检查 WHERE 子句并把读取的数据丢掉。选择不需要的列永远都不是一个好主意，但长值是特殊情况，尤其需要避免这种行为。如果发现查询受到了这个局限的影响，可以试着使用覆盖索引。更多内容请参阅第 120 页的“覆盖索引”。
30
- 如果在单个表里面有很多长列，那么把这些数据合并到一列中也许会更好，也可以使用 XML 文档。这让所有的数据可以共享外部存储，而不是每列都有自己的页面。
- 有时可以通过把长列存储在 BLOB 中并且使用 COMPRESS() 压缩它们得到性能上和空间上的好处，也可以在把它们发送给 MySQL 之前在应用程序中进行压缩。

注 5：如果操作系统交换了数据，它还是会被写到磁盘上。

注 6：这个长度足够在列上创建 255 字节的索引，即使它是 utf-8 也可以。每个 utf-8 字符可能需要 3 个字节。



为文件排序进行优化

MySQL 有两个变量可以控制如何进行文件排序。

回想一下第 176 页的“排序优化”，MySQL 有两种文件排序算法。如果需要进行排序的列的总大小加上 ORDER BY 列的大小超过了 max_length_for_sort_data 定义的字节，MySQL 就会使用双路排序。当任何需要的列——甚至不是用于 ORDER BY 的列——是 BLOB 或 TEXT 列的时候，也会使用双路排序。(可以使用 SUBSTRING() 把这些列转换为可以使用单路排序的列。)

可以通过改变 max_length_for_sort_data 变量的值来影响 MySQL 选择的算法。因为单路排序算法为将要排序的每一行创建了固定大小的缓冲区，VARCHAR 列的最大长度是 max_length_for_sort_data 规定的值，而不是排序数据的实际大小。这就是我们为什么推荐只把列设为需要的大小。

当 MySQL 不得不对 BLOB 或 TEXT 列进行排序时，它只会使用前缀并会忽略掉剩余的值。这是因为它不得不分配固定大小的结构来容纳数据并且从外部存储中将前缀拷贝回结构中。可以使用 max_sort_length 定义前缀应该是多大。

不幸的是，MySQL 不会真正地显示它使用的排序算法。如果增加了 max_length_for_sort_data 的值，并且磁盘的使用率上升、CPU 使用率下降、sort_merge_passes 的值比以前增加的更快，也许就该强制更多的排序使用单路排序算法。

更多关于 BLOB 和 TEXT 的内容请参阅第 84 页的“字符串类型”。

301

6.5.2 检测 MySQL 服务器状态变量

~~Inspecting MySQL Server Status Variables~~

按照工作负载对 MySQL 进行调优的最有生产率的方式是检查 SHOW GLOBAL STATUS 的输出，以了解哪些设置需要改变。如果你刚刚才开始对服务器进行调优并且熟悉 mysqlreport，那么就该运行这个报告，并且检查它的输出，这会节约大量的时间。这个报告可以帮你锁定有问题的地方，并且可以使用 SHOW GLOBAL STATUS 更仔细地检查相关的变量。如果发现某些可以被改进的值，就可以对它进行调优。然后检查 mysqladmin extended -r -i60 产生的增量输出，以确定改变的影响。为了得到最佳的结果，应该检查变量的绝对值以及它随时间的变化。

在第 13 章中有可以使用 SHOW GLOBAL STATUS 进行检测的详细变量列表。下面仅仅显示了某些最值得检查的变量：

Aborted_clients

如果这个变量随时间增加，那么就要确定是否优雅地关闭了连接。如果不是，那就要检查网络性能，并且检查 max_allowed_packet 配置变量，超过了 max_allowed_packet 的查询会被强制地中断。

Aborted_connections

这个值应该接近于 0。不是的话，就可能是网络问题。有几个被中断的连接是正常的。例如，当某些人试着从错误的主机连接、使用了错误的用户名和密码，或者定义了无效的数据库，就会发生这样的情况。

Binlog_cache_disk_use 和 Binlog_cache_use

如果 Binlog_cache_disk_use 和 Binlog_cache_use 之间的比率很大，那么就应该增加 binlog_cache_



size 的值。大部分事务最好都落在二进制日志缓存里面，但是偶尔有一个发生在磁盘上也无妨。

没有非常确定方式可以减少二进制日志缓存未中 (Cache Miss)。最好的办法是增加 binlog_cache_size 并且观察缓存未中率是否下降了。一旦未中率下降到了某一个点，就不会再从加大缓存中受益。假设每秒的未中是一个，并且增加了缓存的大小，让未中减少了到每分钟一次。这已经足够好了，不可能再让它下降，即使是下降了，也不会再从中得到更多的好处，所以还不如把内存节约下来做别的事情。

Bytes_received 和 Bytes_sent

302

这两个值可以帮助你考察问题是由发送服务器的数据过多引起的，还是从服务器读取的数据太多引起的（注 7）。它们也许会指出代码中其他的问题，比如查询提取了超出自己需要的数据（更多内容请查阅第 161 页的“MySQL 客户端/服务器协议”）。

Com_*

应该注意不要让诸如 Com_rollback 这样不常见的变量的值超过预期值。一种检查合理的值的快捷的方式是 innostop 的命令总结模式 (Command Summary Mode)（更多关于 innostop 的内容参阅第 14 章）。

Connections

这个变量表示了连接意图的数量（不是当前连接的数量，它是 Threads_connected）。如果它的值快速增长，例如，每秒几百，那么就应该检查连接次或调整操作系统的网络堆栈（更多关于网络配置的内容见下一章）。

Created_tmp_disk_tables

如果这个值较高，有两件事情可能发生了错误：(1) 查询在选择 BLOB 或 TEXT 列的时候创建了临时表；(2) tmp_table_size 和 max_heap_table_size 可能不够大。

Created_tmp_tables

该值较高的唯一处理办法是优化查询。查询优化提示请参阅第 3 章和第 4 章。

Handler_read_rnd_next

Handler_read_rnd_next / Handler_read_rnd 显示了全表扫描的大致平均值。如果值较大，那么就应该优化架构、索引和查询。

Key_blocks_used

如果 Key_blocks_used * key_cache_block_size 的值远小于已经充分热身的服务器上的 key_buffer_size 值，那么就意味着 key_buffer_size 的值太大了，内存被浪费了。

Key_reads

要注意观察每秒钟发生的读取次数，并且将这个值和 I/O 系统进行匹配，以了解有多接近 I/O 限制。更多内容请参阅第 7 章。

注 7：即使网络容量足够大，也不要认为它不会导致性能瓶颈，网络延迟会让性能变差。



303 Max_used_connections

如果该值和 `max_connections` 相同，那么可能是 `max_connections` 被设置的过低或者最大负载超过了服务器的上限。但是不要假设应该增加 `max_connections`。它是保证服务器不会被太多负载压垮的警戒线。如果看到需求激增，那么就应该检查应用程序的行为是不是正常、服务器调优是否正确、服务器架构是否设计良好。这都比简单地增加 `max_connections` 要好。

Open_files

注意它不应该和 `open_files_limit` 的值接近。如果接近了，那么就应该增加 `open_files_limit`。

Open_tables 和 opened_tables

应该将该值和 `table_cache` 进行对照。如果每秒有太多 `opened_tables`，那么说明 `table_cache` 还不够大。表缓存没有被完全利用上时，显式的临时表也能导致 `opened_tables` 增加。所以说也许可以不用担心它。

Qcache_*

更多信息请参阅第 204 页的“MySQL 查询缓存”。

Select_full_join

全联接是无索引联接，它是真正的性能杀手。最好能避免全联接，即使是每分钟一次也太多了。如果联接没有索引，那么最好能优化查询和索引。

Select_full_range_join

如果该变量过高，那么就说明运行了许多使用了范围查询联接表。范围查询比较慢，同时也是一个较好的优化点。

Select_range_check

该变量记录了在联接时，对每一行数据重新检查索引的查询计划的数量，它的性能开销很大。如果该值较高或正在增加，那么就意味着一些查询没有找到好索引。

Slow_launch_threads

该变量较大说明了某些因素正在延迟联接的新线程。它说明了服务器有一些问题，但是它并没有说明真正的原因。它通常表示系统过载，导致操作系统不能给新创建的线程分配时间片。

Sort_merge_passes

该变量较大说明应该增加 `sort_buffer_size`，也许仅仅只是为某些查询。检查查询并且查明哪一个导致了文件排序。最好的办法是优化查询。

Table_locks_waited

该变量显示了有多少表被锁住了并且导致了服务器级的锁等待（等待存储引擎锁，比如 InnoDB 的行级锁，不会使该变量增加）。如果这个值较高并且正在增加，那么就说明了严重的并发瓶颈。这时候就应该考虑下面的优化手段：使用 InnoDB 或另外的使用了行级锁定的存储引擎；手动对大表进行分区或者使用



MySQL 5.1 或以上版本的内建分区机制，优化查询；启用并发插入或者对锁定设置进行优化。

MySQL 不会显示等待时间是多长。在写本书的时候，最好的办法也许是使用微秒粒度的慢速查询日志。更多内容请参阅第 63 页的“MySQL 剖析”。

Threads_created

如果该变量较大或正在增加，那么也许就应该增加 `thread_cache_size` 的值。可以通过检查 `threads_cached` 知道有多少线程已经在缓存中了。

6.6 每连接 (Per-Connection) 设置调优

除非确信自己是正确的，否则就不要全局性地增加每连接设置的值。即使不需要某些缓存，它们也会一次性地被分配好。所以庞大的全局设置可能会成为巨大的浪费。相反，应该在查询需要的时候才增加设置的值。

对于平时需要保持较小值，只在需要的时候才进行增加的设置来说，最常见的例子就是 `sort_buffer_size`，它控制了用于文件排序的缓存大小。即使排序的数据量很小，它也会按照设置分配全部的空间，所以如果它的值大于排序需要的空间，那么就意味着浪费。

当发现查询需要较大的排序缓冲区时，就可以在查询之前提高它的值，并且在查询之后就把值恢复为 `DEFAULT`。下面是一个例子：

```
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := DEFAULT;
```

把这种代码放到函数中也许会更方便。其他的应该基于每个连接进行设置的变量是 `read_buffer_size`、`read_rnd_buffer_size`、`tmp_table_size`，以及 `myisam_sort_buffer_size`（用于修复表）。

如果需要保持并且恢复定制的值，可以按照下面的方式进行：

```
SET @saved_<unique_variable_name> := @@session.sort_buffer_size;
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := @saved_<unique_variable_name>;
```