457

第 10 章

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。下文中我们以 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、网络和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，应用会受限于使用事务、临时表、连接指定的设置和定义用户变量。

459

- 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连到 MySQL 上。通常情况下，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，如果每条连接上只执行一两个快速的查询，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。

460 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，总好过 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漫长（注 1）。

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 使用手册里的说明，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，硬件化的负载平衡器会处理好缓存，Apache 就能很快地结束响应，然后让负载平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。

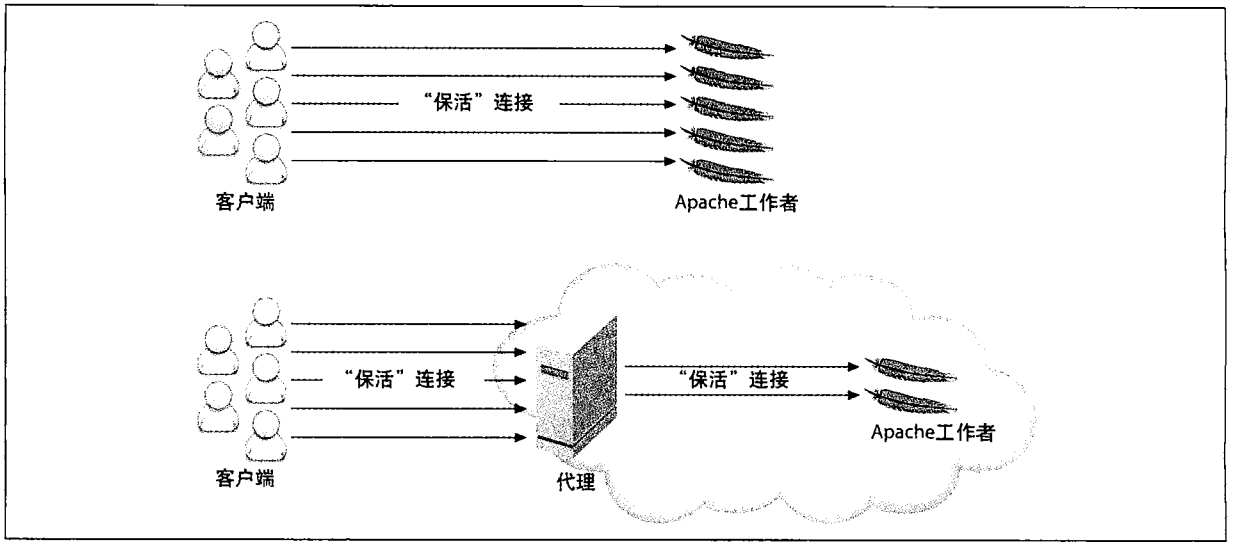

图 10-1：一个代理可以让 Apache 免受长久保持的“保活”连接的负担，从而可以使用更少的 Apache 工作者进程

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。

463

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，进程不总是可以运行的，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，最佳并发数往往会多于 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，但是也不能保证肯定是这样。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接（Mount）选项上开启“访问时间”这个开关项。（实际操作时忽略 noatime 挂接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，你也可以暂时不理会它，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—作废**和**写—更新**。写—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—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。折中的方案是朴素的，一个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

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

除了在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。

10.4 扩展 MySQL
Extending MySQL

470

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去做扩展，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。

10.5 可替代的 MySQL
Alternatives to MySQL

471

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<img>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。