改善性能和扩展规模的具体做法

构建高性能Web站点

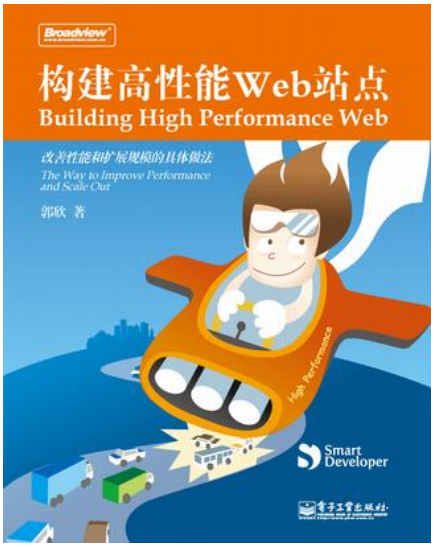
精选版

郭欣 著

InfoQ企业软件开发丛书

免费在线版本

（非印刷免费在线版）



推荐序

你很幸运能拿到这本书，更重要的是，你的网站用户也会很幸运。郭欣在这本书里深入而系统地分享了构建高性能网站技术的方方面面。从后台到前台，从网络传输到数据存储，涉及诸多技术原理和实现细节。通俗的语言，亲切的叙述，仿佛作者在耳边轻轻细语，然而又蕴含着一种技术思想和力量，并且融合了人文思想。

我曾经代表公司面试过许多开发人员，在问及与高性能相关的问题时，大家都能回答出需要负载均衡，需要缓存技术，然而当我进一步询问负载均衡如何实现或如何有效控制缓存命中率时，面试者却无从答起。知其然而不知其所以然是很多浮躁开发者的通病，也因此限制了其技术能力的提升和发展。

这本书将为你提供构建高性能网站的完整解决方案，它会成为每个致力于开发承载百万级用户规模网站开发者的工具箱。郭欣有着架构和开发多个大规模网站的经验，他精通前/后台技术和架构。在知道他将花时间著作一本高性能网站架构的书时，我不禁为国内许多开发者感到高兴。我见过部分知名网站架构师曾经分享过他们网站技术发展的历程，但每每都是停留在抽象层面，而像本书这样全面彻底地进行技术剖析却是头一回。尤其是构建高性能网站的各种技术方案，绝大部分是通过实践总结出来的经验，没有这样的经历，你甚至很难想象为什么会是这样。

不要犹豫了！当你拿起这本书，按照书中所分享的技术方案去实践时，你会发现，原来构建高性能网站就这么简单。中国互联网正在不断地成长，用户规模也在不断地扩大，我相信，越来越多的网站会根据性能这项最基本的用户体验决定其自身的生存能力。本书所提供的技术解决方案，正是在这个发展趋势中的一个基础，拥有它并加以实践，你和你的用户都会更加享受这一切！

为了页面一秒响应的境界，开始阅读吧！

——王速瑜
腾讯 R&D 研发总监（Tencent Director of R&D）
资深技术专家（Senior Technology Expert）
深圳，2009 年 7 月

前　言

从我写出第一个 HTML 网页到现在，已经过去 10 年多的时间了，回顾过去的 Web 开发经历，我曾经尝试过各种不同的技术，与此同时，我和我的团队也犯了很多的错误，但我们为此感到自豪。是的，成长是需要不断付出代价的，每次的挫折都会让我更加深刻地看到隐藏在深处的本质，为什么不把这些内容分享出来呢？于是便有了《构建高性能 Web 站点》这本书。

10 年来，我们见证了互联网有史以来最快速的发展，商业应用层出不穷，业务逻辑不断复杂，对用户体验的要求也不断提升，随之而来的是应用技术和开发语言的日新月异，开发者永不停息地学习新技术。同样，在 Web 站点性能方面，我们一直在跟时间赛跑，社交网站和微博客成为大众的主流应用，带来了更加快速、实时的信息传递，更多的站点意识到开放的重要性，数据访问和计算无处不在，每秒数以万次的数据传递和读写正在我们身边进行。

但是，构建 Web 站点的基础技术几乎多年来从未改变，比如诞生于 20 世纪 80 年代的 TCP，如今依旧是网络数据传输的主宰者，而 HTTP 则更与我们息息相关，可是你真的认真学习过它们吗？人们始终在做的事情就是在这些基础技术之上一层一层地封装概念，不断地诞生新的技术。加上商业化产品的市场竞争和炒作，.NET 和 Java 阵营中的概念让我眼花缭乱却又无可奈何。它们已经成为营销用语，有时候过度会让事情变得更加复杂，让开发者迷失方向。

不论你是一名从事 Web 开发的工程师，还是一名关心 Web 性能的架构师，都应该更多地关注各种技术和架构的本质。

从哲学意义上讲，对本质的研究属于形而上学的范畴，但是在自然科学中，我们从来不缺乏对本质的探索，因为只有认识事物的本质才能做出正确的决策，并且真正地驾驭它们，这是毫无争议的。

也许你曾经被商家的促销活动所打动。是的，我们往往只看到事物的表面现象，而经济学家却看到了事物的本质，这正是他们的高明之处。技术和架构同样如此，你要明白任何收获都是有代价的，天下没有免费的午餐，很多时候，你完全可以用成本经济学的知识来思考技术的合理性，你甚至可以像经济学家一样思考技术问题。

当然，仅仅理解本质是远远不够的，因为在庞大的架构体系中，涉及太多的部件，而影响整体性能的因素究竟有哪些呢？你也许会感到扑朔迷离，但你必须知道瓶颈所在，并且能够意识到何时需要优化性能或者扩展规模。与此同时，系统化的分析方法至关重要，中医理论对人体的系统思辨能力体现了先哲们的智慧，在站点性能不尽如人意的时候，我们能否“对症下药”？这与你对整个系统能否全面把握有着密切的关系。

另一方面，绝对与相对、变化与平衡，是永恒的大道，在很多时候你实际上需要考虑的是如何做出权衡，同时，我们也要铭记变化的道理，系统瓶颈不是一成不变的，久经考验的架构师深知这一点。

道可道，非常道。要将所有的架构之道讲出来实属不易，架构就像艺术品一样，往往无法完全复制，但是独立的技术以及分析的思路是可以学习的，作为优秀的开发者或者架构师，心中的架构才是最有价值的。

如果你希望寻找心中的架构，那么，从本书的绪论开始吧！

读者群

如果你希望学习如何创建一个 Web 站点，那么这本书可能并不适合你，但是当你对站点的性能开始担忧时，欢迎你的归来。

这本书适合以下读者：

- 编写 Web 应用程序、关心站点性能，并且希望自己做得更加出色的开发人员
- 关心性能和可用性的 Web 架构师
- 希望构建高性能 Web 站点的技术负责人

- 实施 Web 站点性能优化或者规模扩展的运维人员
- 与 Web 性能有关的测试人员

的确，整个技术团队的所有成员都适合阅读这本书。另外，高校学生以及个人网站站长也可以阅读，笔者希望本书可以帮助他们开拓视野。

如何阅读

本书涉及大量的软件和工具，由于篇幅有限，书中并没有对它们的具体使用方法展开详细的介绍，你可以通过 Google 查找相关的在线手册来进行自学，同时，在本书的参考文献列表和配套 Blog 中，也会提供一些相关链接。

本书不想在阐释体系上束缚读者的天分，所有内容的安排更像是引导你身处 Web 站点的各个角落，与影响性能的各种因素自由碰撞，并且一步步地思考和分析问题。所以，你也许不会直接找到一个高性能 Web 站点的完整解决方案，但是，当你认真地依次阅读完本书的所有章节后，你也许会找到你更想要的东西，并从中获益，我衷心希望如此。

如果没有什么特殊的原因，还是建议你能够按照章节顺序，心平气和地通读一遍，因为在内容组织上，本书有太多的连贯性设计，我担心随机阅读会让你的收益大打折扣。

创作过程

说到这次写书的过程，我同样感到很有意义，各种全新的尝试让创作过程充满乐趣，它们同样值得分享。

不同于传统使用 Word 编写内容，我使用了快捷的 Google 在线文档，并使用在线表格保存测试数据，它们支持出色的版本管理，并且提供快速的分享和协作功能。当然，最激动人心的莫过于我可以在任何地点通过浏览器继续我的写作过程，甚至当灵感突如其来时打开 Google G1 手机便可以写上两句。

对于几十万字的篇幅，一气呵成绝对是不可能的，多次迭代必然贯穿整个写作过程，从灵感到提纲，再从框架到最终文字，虽然没有完善的过程管理，但是我时刻能感觉到敏捷的火花。

为了尽早地获得读者的反馈，我考虑尽早“部署”，于是选择了讨论组和邮件列表的方式，在 Google Group 上创建了读者讨论组，上传了一些试读章节，收集到了大量的修改意见和想法，这些都是我所需要的，同时也给我带来了鼓励和支持。

整个过程还有很多的花絮，这里就不一一介绍。创作的过程是艰辛的，需要作者的坚持和毅力，虽然创作本身没有捷径，但是我们可以让创作过程更加充满乐趣，让作者和读者更加近距离地接触。

当我将这些过程介绍给一些朋友时，他们感到很有意思，于是我们创立了 SmartDeveloper 系列，希望能够将这种敏捷写作过程进一步整理和完善，当然，《构建高性能 Web 站点》将作为该系列的开山之作。

SmartDeveloper 的具体内容敬请关注以下地址：

http://smartdeveloper.cn

值得一提的是，为本书撰写推荐序的王速瑜先生在敏捷开发领域有着丰富的经验，并在腾讯公司内部积极推广敏捷开发平台和方法。长久以来，我认为我们都是敏捷原住民，骨子里充满了敏捷的思想和战斗力。不可否认，敏捷给我带来了无法估量的收获。幸运的是，他也计划写一本关于敏捷开发的书，总结他的实战经验，并且加入 SmartDeveloper 系列，我也非常期待这本书的问世。

延展阅读

由于篇幅和时间的限制，书中的部分内容点到为止，但是我也希望能够在未来继续和读者进行沟通，所以你也可以访问以下站点，来进一步关注其他的延展阅读资料。

http://highperfweb.com

读者讨论组

作为读者讨论和邮件订阅的最佳途径，我仍然选择了以下讨论组，在本书出版后的任何时间，你都可以来这里提出自己的意见和想法，我会对所有的主题给予回复。

http://groups.google.com/group/highperformanceweb

致谢

感谢我的父母，他们在我读初中的时候送给我第一台电脑（Cyrix1.66G 的兼容机），让我走进了计算机的世界，并且给予我非常多的支持和鼓励，让我毫无顾虑地追逐梦想。

感谢我的 Cris，她为我创作了本书的封面，并且在整个写作过程中毫无抱怨地陪伴我，给予我无尽的支持和灵感。

感谢电子工业出版社的策划编辑李冰和文字编辑江立，她们严谨认真的工作态度以及作为出版商的开放态度让我深感敬佩。

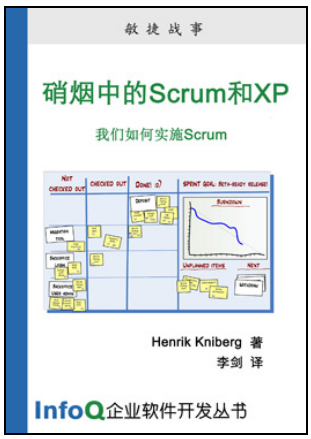
感谢为本书撰写推荐序的王速瑜先生，以及撰写评价的章文嵩博士、张松国先生、沈翔先生、朱鑫先生，他们在繁忙的工作中抽出时间，阅读了本书的样稿并写下了推荐和评价。特别一提的是，章文嵩博士对书中一些内容的理解和建议让我受益匪浅。

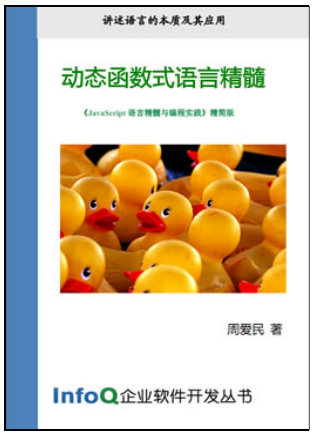
感谢对本书提出宝贵修改意见的朋友，他们是蒋琦、丁吉亮、汤文亮、刘健、朱李、周伟强。

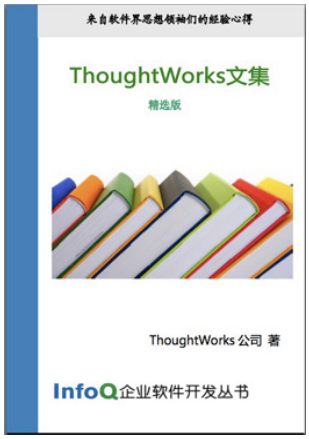
感谢在读者讨论组中所有积极阅读试读内容并提出意见的成员。

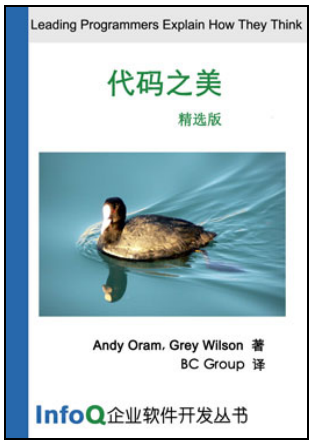
InfoQ企业软件开发丛书

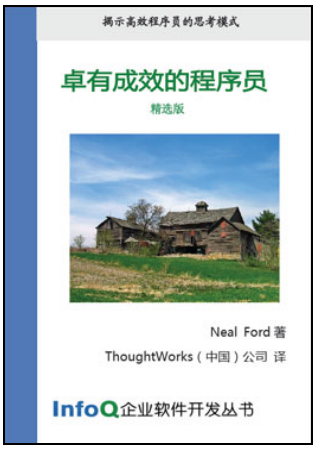
欢迎免费下载

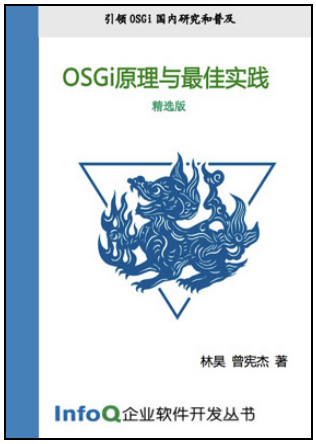












商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com

目录

分布式缓存

说到缓存，你已经非常熟悉了，我们前面曾经探讨了有关动态内容的各种缓存，但基本上都是基于页面缓存，或者整体缓存，比如缓存整个动态图片。无论如何，它们的目的都在于避免重复的慢速计算，比如数据库访问。但是在有些时候，使用页面缓存显得尤为笨重，这可能来自于以下几个原因：

- 一个网页中不同区域的内容，自身更新频率和呈现及时度要求各不相同，如果为了迁就频繁更新的区域，而使整个页面频繁重建缓存，则影响整体吞吐率。
- 即便是采用局部动态缓存，如果局部区域过多，则会使得页面结构过于复杂，而且整合各个局部页面也存在不小的开销。
- 有些计算是无法作为页面来缓存的，比如有些动态内容中需要获取用户的登录状态，并根据不同用户呈现不同的内容。
- 这些页面缓存都只是提高了读数据的速度，并没有提高写数据的速度。

那么，有什么更好的方法呢？

1.1 数据库的前端缓存区

还记得我们曾经介绍过的文件系统内核缓冲区（Buffer Area）吗？它位于物理内存的内核地址空间，除了使用 O_DIRECT 标记打开的文件之外，所有对磁盘文件的读写操作都要经过它，所以你也可以把它看成是磁盘的前端设备。

这块内核缓冲区也称为页高速缓存（Page Cache），实际上它包括以下两部分：

- 读缓存区
- 写缓存区

读缓存区中保存着最近系统从磁盘上读取的数据，一旦下次需要读取这些数据的时候，内核将直接从这里获得，而不需要访问磁盘。

写缓存区的目的主要在于减少磁盘的物理写操作，通常情况下向磁盘中写入数据并不着急，进程不需要因为写操作而等待，内核缓冲区可以将多次写操作的指令累积起来，通过一次物理磁头的移动来完成。当然，写缓存区导致数据真正写入磁盘会产生几秒的延迟，在实际写入磁盘之前，这些数据被称为脏页（Dirty Page）。

其实，从工作职能上看，将写缓存区称为缓冲区更加形象，缓冲区的例子在生活中处处可见，比如城市道路的十字路口，它就像一个写缓冲区，红灯亮起的时候，车辆都停在缓冲区，当变成绿灯后，车辆开始依次前进，这就像内核缓冲区中的数据积累到一定程度时被写入磁盘。

一个有趣的现象是，有些城市会在十字路口增加左转弯等待区，多么美妙的想法啊，它使得缓冲区与目的地的距离更加接近，绿灯时将会有更多的车辆通行，提高了道路的吞吐率。

同样的，类似于页高速缓存，我们也可以在数据库和动态内容之间建立一层缓存区，它可以部署在独立的服务器上，用于加速数据库的读写操作，这个缓存区实际上是由动态内容来控制的。

1.2 使用 memcached

幸运的是，开源社区已经有非常成熟的分布式缓存系统，现如今，几乎没有人不知道 memcached，我们也曾经在前面使用过 memcached 来存储动态内容的页面缓存。可是，你真的能让它工作得愉快吗？

key-value

首先，我们要知道，为了实现高速缓存，我们不会将缓存内容放置在磁盘上，否则将毫无意义。基于这个原则，memcached 使用物理内存来作为缓存区，当我们启动 memcached 的时候，需要指定分配给缓存区的内存大小，比如我们分配了 4GB 的内存来作为缓存区：

```
s-colin:~ # memcached -d -m 4086 -l 10.0.1.12 -p 11711
```

如果要说 memcached 最需要的是什么，毫无疑问，那就是内存。我使用的一台 memcached 服务器已经消耗掉了 2.8GB 的内存空间，而我一共给它分配了 4GB。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3950 root      15   0 2913m 2.8g  384 S    1 35.9   1325:49 memcached
```

memcached 使用 key-value 的方式来存储数据，这是一种单索引的结构化数据组织形式，我们将每个 key 以及对应的 value 合起来称为数据项，所有数据项之间彼此独立，每个数据项都以 key 作为唯一索引，你可以通过 key 来读取或者更新这个数据项。

对于 key-value 这样的存储形式，你无法习惯性地像操作关系数据库那样来控制它，最大的不同在于，所有基于 SQL 语句的条件查询方式在 key-value 形式的存储数据上都无法施展。所以，如何用 key-value 方式来存储数据，一部分取决于你将如何查询这些数据。

在稍具规模的应用中，缓存的数据项可能会非常多，足够达到天文数字，为了在内存中为如此之多的数据项提供高速的查找，memcached 使用了高效的基于 key 的 hash 算法来设计存储数据结构，并且使用了精心设计的内存分配器，它们使得数据项查询的时间复杂度达到 O(1)，这意味着不论你存储多少数据项，查询任何数据项所花费的时间都不变，几乎没有什么理由可以让你不用它。

数据项过期时间

由于缓存区空间是有限的，一旦缓存区没有足够的空间存储新的数据项时，memcached 便会想办法淘汰一些数据项来腾出空间，淘汰机制基于 LRU（Least Recently Used）算法，将最近不常访问的数据项淘汰掉。

当然，我们更愿意为数据项设置过期时间，这样它可以很有面子地离开缓存区。至于过期时间的取值，前面我们已经讨论过很多次，这需要你根据自己的站点来把握平衡，它们的道理都是相似的，你可以回顾一下之前介绍过期时间的内容。

如果你在使用 PHP 来编写动态内容，通过 memcached 的 PECL 扩展，你可以很容易地设置数据项过期时间，比如：

```
<?php
$memcache_obj = memcache_connect('10.0.1.12', 11211);
$memcache_obj->add('item_key', 'item_value', false, 30);
?>
```

这里我们将 item_key 这个数据项的过期时间设置为 30 秒。

网络并发模型

作为分布式缓存系统，memcached 可以运行在独立的服务器上，动态内容通过 TCP Socket 来访问它，这样一来，memcached 本身的网络并发处理能力便显得尤为重要。memcached 使用 libevent 函数库来实现网络并发模型，其中包括我们前面详细介绍过的 epoll，所以你可以在较大并发用户数的环境下仍然放心使用 memcached。

这里，我们不妨做一个简单的性能测试，来看看 memcached 中 set 操作的性能，我们用 PHP 编写了以下的代码：

```
<?php
$key = md5(uniqid());
$value = md5(uniqid());
$memcache = memcache_connect('10.0.1.12', 11711);
$memcache->set($key, $value, false, 0);
$memcache->close();
```

```
?>
```

很简单，它首先随机生成两个 32 字节的字符串，分别赋值给 key 和 value，比如：

```
key -> 87e4f726c1837c98b4719488d9530532
value -> 64f0f23d115dca3b019bea8340cb552e
```

接下来连接到 memcached 服务器，然后通过 set 操作来写入数据，最后关闭连接。显然，我们将这一系列的操作打包为一个动态程序，下面就对这个动态程序进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_memcache_perf3.php
Document Length:        844 bytes
Concurrency Level:      500
Time taken for tests:   2.138 seconds
Complete requests:      5000
Failed requests:        0
Write errors:           0
Total transferred:      5028338 bytes
HTML transferred:       4220000 bytes
Requests per second:    2338.48 [#/sec] (mean)
Time per request:       213.814 [ms] (mean)
Time per request:       0.428 [ms](mean, across all concurrent requests)
Transfer rate:          2296.61 [Kbytes/sec] received
```

从结果中我们知道 memcached 服务器平均每秒处理了大约 2338 个 set 请求，当然，处理过程中不仅仅包括 set 操作，还包括了建立连接和释放连接，它们的开销不可忽视。

接下来，我们希望尽可能地了解 set 操作本身的性能，办法是有的，我们在一个连接中重复进行多次 set 操作就可以了，我们修改一下刚才的 PHP 代码，如下所示：

```
<?php
$key = md5(uniqid());
$value = md5(uniqid());
$memcache = memcache_connect('10.0.1.12', 11711);
for ($i = 0; $i < 10000; ++$i)
{
    $memcache->set($key . $i, $value . $i, false, 0);
}
$memcache->close();
?>
```

与修改之前相比，我们看到 set 操作被一个循环体重复执行了 10000 遍，并且为了让每次 set 操作可以更新不同的节点，更加接近实际运行情况，我们在循环体中对 key 的末尾追加了递增的数值。

我们再来进行压力测试，但这次只模拟 10 个并发用户数，同时总请求数为 50，也就是每个用户发送 5 个请求，为什么这么做呢？别忘了我已经让脚本中的 set 操作重复了 10000 遍，我可等不了那么长的时间。测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.liveinmap.com
Server Port:            80
Document Path:          /book_memcache_perf3.php
Document Length:        2 bytes
Concurrency Level:      10
Time taken for tests:   12.248 seconds
Complete requests:      50
Failed requests:        0
Write errors:           0
Total transferred:      10100 bytes
HTML transferred:       100 bytes
Requests per second:    4.08 [#/sec] (mean)
Time per request:       2449.526 [ms] (mean)
Time per request:       244.953 [ms] (mean, across all concurrent requests)
Transfer rate:          0.81 [Kbytes/sec] received
```

这次我们不看结果中的吞吐率，而是记录下总时间，为 12.248 秒。在这段时间内，memcached 一共处理了 50 个请求，而每个请求包括 10000 次 set 操作，那么我们可以算出每秒执行的 set 次数为：

```
10000 × 50 / 12.248 = 40823
```

当然，这是极端前提下的结果，实际情况中根据 TCP 连接复用程度的不同，memcached 的吞吐率会存在上下浮动，这是正常的，你需要根据站点的实际环境来评估它的处理能力。

对象序列化

我们可以在 memcached 的数据项中存储什么格式的内容呢？这要考虑到网络传输，问题也就转换成了“我们可以在网络中传输什么内容”。自然是二进制数据。那么，对于数组或者对象这样的抽象数据类型，是否可以存入 memcached 中呢？

基于序列化（Serialize）机制，我们可以将更高层的抽象数据类型转化为二进制字符串，以便通过网络进入缓存服务器，同时，在读取这些数据的时候，二进制字符串又可以转换回原有的数据类型。

但是，有一点需要清楚的是，当我们试图将一个类的对象（或类的实例）进行序列化时，对象的成员函数是不被序列化的，而被序列化存储的只是对象的数据成员。当需要从持久化数据中恢复对象（反序列化）时，我们首先会实例化另一个新的对象，然后将之前持久化的数据成员依次赋值给新对象的相应数据成员。听起来比较复杂，不过幸运的是，在具有动态特性的脚本语言中，这些具体的过程往往不需要你去实现，你只需要或多或少了解序列化的本质即可。

顺便说一下，我们熟悉的 JSON 格式，便可以很好地应用在序列化中，任何数组和对象都可以很容易地与 JSON 格式的字符串互相转换，而且转换所需的计算量并不大。

对于对象序列化，常见的服务器端动态脚本语言都有相应的扩展支持，比如通过 PHP 的 memcached 扩展，你可以随时将对象写入缓存服务器，并在随后取出。下面是一个简单的例子：

```
<?php
class Person
{
        var $name;
        function setName($name)
        {
                $this->name = $name;
        }
}
$person = new Person();
$person->setName('colin');
$key = 'person.colin';
$memcache = memcache_connect('10.0.1.12', 11711);
$memcache->add($key, $person, false, 0);
$obj = $memcache->get($key);
echo $obj->name;
?>
```

好，我们来运行这个 PHP 脚本，你可以通过浏览器请求它，也可以直接通过 PHP 的命令行方式来执行它，无论如何，它的结果都是一样的，如下所示：

```
colin
```

1.3 读操作缓存

前面简单介绍了 memcached 的一些基本特性，因为它实在是家喻户晓，以至于我觉得不需要太多详细地介绍它本身，否则你会觉得我在浪费你的时间，好，我们还是来看看如何用它来帮助站点提高吞吐率，这也许是你最关心的。

还记得磁盘缓存区中的两部分吗？没错，读缓存区和写缓存区。那么我们也按照这个思路，先将 memcached 作为数据库的读缓存区，来看一个例子。

重复的身份验证

大多数站点都有自己的用户系统，这给我们带来了不少的麻烦，有些事情不可不做，那就是在每个用户请求页面的时候，都需要检查用户的登录状态。

很早以前，我喜欢将登录用户的 ID 直接写入浏览器 cookies，并以此为荣，可是在随后的几年里，破坏者对 cookies 更是情有独钟，他们可以很容易地利用 cookies 的无知，篡改本地 cookies 来冒充其他用户，这让我感到时代在飞速前进，我们必须改变。

新的方式问世了，用户在登录站点的时候会获得一个 ticket 字符串，并将它写入浏览器 cookies，随后每次请求新的页面时，都需要上报这个 ticket，接受重复的身份检查。非常好，它工作得一切正常，破坏者由于不知道我们发给其他登录用户的 ticket，所以无法冒充。补充一点，事实上，永远不要低估破坏者的本领，它们可以通过其他手段获得他们想要的一切东西。

现在，我们将身份检查部分的代码抽取出来，它的工作很简单，读取 cookies 中的 ticket 字符串，然后通过条件 SQL 语句查询数据库，寻找拥有这个 ticket 的用户。我们来看这部分代码，它用 PHP 编写而成。

```
<?php
$user_ticket = '010f06c4c7e74f82fe7fb2aea97c50b2';
$sql = "select * from user_info where user_ticket='" . $user_ticket . "'";
$conn = mysql_connect('localhost', 'root', '', 'db_user');
$res = mysql_query($sql, $conn);
$row = mysql_fetch_array($res, MYSQL_ASSOC);
var_dump($row);
?>
```

以上的代码不难理解，我们这里省略了从 cookies 中获取 ticket 的过程，而直接将 ticket 赋值给$user_ticket 变量，接下来是数据库查询，然后将获得的用户信息进行打印。这里涉及一个数据表 user_info，它有大约 20 个字段，1 万多行的记录。

我们来对它进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /readcache_mysql.php
Document Length:        850 bytes
Concurrency Level:      500
Time taken for tests:   40.263 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      10042546 bytes
HTML transferred:       8500000 bytes
Requests per second:    248.37 [#/sec] (mean)
Time per request:       2013.155 [ms] (mean)
Time per request:       4.026 [ms] (mean, across all concurrent requests)
Transfer rate:          243.58 [Kbytes/sec] received
```

我不认为这个结果很出色，你也许跟我有同样的想法。

数据库索引

不过，我并没有对数据库失去信心，我们来看看以上程序中执行的一段 SQL 语句：

```
select * from user_info where user_ticket='010f06c4c7e74f82fe7fb2aea97 c50b2'
```

它包含一个 where 条件，我想我们可能没有使用索引，通过 MySQL 的查询分析工具来分析这条 SQL 语句，结果如下所示：

```
mysql> explain select * from user_info where user_ticket='010f06c4 c7e74f82fe7fb2aea97c50b2';
+----+-------------+-----------+------+---------------+------+---------+------+------+----------
---+
| id | select_type | table     | type | possible_keys | key  | key_len | ref  | rows | Extra      |
+----+-------------+-----------+------+---------------+------+---------+------+------+----------
---+
|  1 | SIMPLE      | user_info | ALL  | NULL          | NULL | NULL    | NULL | 10807 | Using where
|
+----+-------------+-----------+------+---------------+------+---------+------+------+----------
---+
1 row in set (0.02 sec)
```

的确，请看 key 字段，结果为 NULL，这导致它要遍历所有的 10807 行记录来寻找目标，毫无疑问时间都花在这里了。

我们这里要做的就是为 user_ticket 字段加上索引，然后再来看看压力测试结果：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_readcache_mysql.php
Document Length:        850 bytes
Concurrency Level:      500
Time taken for tests:   5.970 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      10140721 bytes
HTML transferred:       8500000 bytes
Requests per second:    1675.11 [#/sec] (mean)
Time per request:       298.489 [ms] (mean)
Time per request:       0.597 [ms] (mean, across all concurrent requests)
Transfer rate:          1658.86 [Kbytes/sec] received
```

虽然难以置信，但是合乎情理，我们再来看看 SQL 分析：

```
mysql> explain select * from user_info where user_ticket='010f06c4c7 e74f82fe7fb2aea97c50b2';
+----+-------------+-----------+------+---------------+-------------+---------+-------+------+--
-----------+
| id | select_type | table     | type | possible_keys | key         | key_ len | ref   | rows |
Extra       |
+----+-------------+-----------+------+---------------+-------------+---------+-------+------+--
-----------+
|  1 | SIMPLE      | user_info | ref  | user_ticket   | user_ticket | 35      | const |    1 | Using
where |
+----+-------------+-----------+------+---------------+-------------+---------+-------+------+--
-----------+
1 row in set (0.00 sec)
```

这时候 key 的结果为 user_ticket，这意味着此次查询利用了该索引，直接找到目标，的确，rows 为 1。

关于数据库索引的探讨，我们暂且放下，这里之所以提出索引，目的在于希望数据库和 memcached 保持相似的压力测试条件，因为 memcached 也使用了基于 hash 算法的 key 来直接定位目标。另外，这里拿出数据库索引小试牛刀，也算是给下一章做个预热，下一章将详细介绍包括索引在内的一系列数据库优化方法。

缓存用户登录状态

即便使用了索引，查询本身还是存在开销，这很大程度上在于数据库的 I/O 操作，这时候，轮到 memcached 出场了。我们接下来希望将用户状态缓存在 memcached 中，没错，对象序列化派上用场了，它帮你简化了工作量，我们来看看新的代码：

```
<?php
$user_ticket = '010f06c4c7e74f82fe7fb2aea97c50b2';
$memcache = memcache_connect('10.0.1.12', 11711);
$user = $memcache->get($user_ticket);
if ($user !== false)
{
        var_dump($user);
        exit(1);
}
$sql = "select * from user_info where user_ticket='" . $user_ticket . "'";
$conn = mysql_connect('localhost', 'root', '', 'db_user');
$res = mysql_query($sql, $conn);
$user = mysql_fetch_array($res, MYSQL_ASSOC);
$memcache->add($user_ticket, $user, false, 3600);
var_dump($user);
?>
```

这样一来，用户在 1 个小时的缓存有效期内，便不需要访问数据库。有一点需要注意，当用户选择注销时，你必须主动清空 memcached 中该用户的登录状态缓存，否则会让用户感到匪夷所思。

还是老办法，压力测试见分晓，结果如下所示：

```
Server Software:        lighttpd/1.4.20
```

```
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_readcache_memcache.php
Document Length:        844 bytes
Concurrency Level:      500
Time taken for tests:   3.767 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      10036411 bytes
HTML transferred:       8440000 bytes
Requests per second:    2654.64 [#/sec] (mean)
Time per request:       188.349 [ms] (mean)
Time per request:       0.377 [ms] (mean, across all concurrent requests)
Transfer rate:          2601.86 [Kbytes/sec] received
```

相比于前面的访问数据库，这次的吞吐率提高了大约 58%，因为我们使用了数据库的“前置读缓存”。事实上，数据库本身也有查询缓存，后面我们会介绍，但是那属于数据库的控制范围，而我们这里的分布式缓存则由动态内容来控制，它更加灵活。是的，它可以缓存一切。

值得说明的是，以上的压力测试结果都不具备绝对意义，因为有很多不确定的环境因素，比如数据库环境设置、数据表记录数、opcode 缓存、机器硬件配置、当前系统负载等。另一方面，它们的相对意义也有一定的约束条件，也许在某种环境下，你可以让访问数据库获得比访问 memcached 缓存更好的表现，比如将 memcached 服务器部署在另一个数据中心，当然，你不会那么做。总之，你只需要从本质上理解这些对比背后的原因即可，它们往往都是一些简单的道理，类似于索引优于没有索引、内存优于磁盘、近距离优于远距离。

1.4 写操作缓存

对于一个数据库写操作频繁的站点来说，通过引入写缓存来减少写数据库的次数显得至关重要。我们知道，通常的数据写操作包括插入、更新、删除，这些操作又同时可能伴随着条件查找和索引的更新，所以它们的开销往往会令人望而生畏。

直接更新

下面我们来看一个有趣的例子，就拿站点访问量统计功能来说，我们需要记录每个 URL 的累计访问量，所以每次页面刷新都会伴随着一次访问量的增加，我们将访问量数据保存在数据库中，这毫无疑问，因为我们要长久地保存它。

为此，我们编写了一段有代表性的代码，它可以让某个页面的访问量加 1，代码如下所示：

```
<?php
$page = 'article_090222.htm';
$sql = "update page_view set view_count=view_count+1 where page='" . $page . "'";
$conn = mysql_connect('localhost', 'root', '', db_page);
mysql_select_db('db_map_main', $conn);
mysql_query($sql, $conn);
?>
```

这段代码很简单，虽然我不知道 article_090222.htm 这个页面目前的访问量是多少，但是我知道它的访问量增加了 1，而且结果将保存在数据库中。我们可以称它为“直接更新”，这来源于之前介绍的文件访问中的直接 I/O 标记（O_Direct），它可以跳过内核写缓存，将数据毫无延迟地直接写入磁盘。

我们对上面的动态程序进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_writecache_mysql.php
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   5.239 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      1690000 bytes
```

```
HTML transferred:       0 bytes
Requests per second:    1908.89 [#/sec] (mean)
Time per request:       261.933 [ms] (mean)
Time per request:       0.524 [ms] (mean, across all concurrent requests)
Transfer rate:          315.04 [Kbytes/sec] received
```

从测试结果可以看出，我们对这个动态程序一共请求了 10000 次，这也意味着 article_090222.htm 这个页面的访问量被增加了 10000。下面我们引入 memcached 作为写缓存，它也许会使得数据更新出现延迟，但是我们可以接受，因为我们并不需要访问量数据实时更新。

线程安全和锁竞争

在此之前，我们先来看一种传统的分布式加运算，以下的代码只是例子中的一个片段，它先从缓存服务器上取回一个数值，然后在本地加 1，接下来写回缓存服务器。

```
<?php
$count = $memcache->get($key);
$count++;
$memcache->set($key, $count, false, 0);
?>
```

看起来没有任何问题，但是，别忘了可能会有多个用户同时触发这样的计算，你一定能想象的到会有什么糟糕的后果，最后的累计访问量总是小于实际访问量。

事实上，这并不涉及 memcached 本身线程安全的问题，而是以上这种加运算的方式不是线程安全的。如果要保证这种加运算可以正常无误地同时进行，那就要考虑一定的事务隔离机制，简单的办法是使用锁竞争，并且将锁保存在 memcached 上，存在竞争关系的动态内容可以争夺这个锁，一旦某个会话抢到锁，那么其他的会话必须等待。

这里要说的不是如何实现这种分布式锁机制，而是并不鼓励这样做，因为锁竞争带来的等待时间是无法容忍的，这将使得引入 memcached 作为写缓存的唯一优势立刻烟消云散。

原子加法

幸运的是，memcached 提供了原子递增操作，事实上，也正是因为它，我们才考虑在访问量递增更新的应用中引入写缓存。

我们再来修改代码，加入 memcached 的支持，如下所示：

```
<?php
$page = 'article_090222.htm';
$memcache = memcache_connect('10.0.1.12', 11711);
$count = $memcache->increment($page, 1);
if ($count === false)
{
        $memcache->add($page, 1, false, 0);
        exit(1);
}
if ($count == 1000)
{
        $memcache->set($page, 0, false, 0);
        $sql = "update page_view set view_count=view_count+" . $count . " where page='" . $page . "'";
        $conn = mysql_connect('localhost', 'root', '', 'db_page');
        mysql_query($sql, $conn);
}
?>
```

在新的代码中，完全改变了之前的“直接更新”方式，当需要增加一次访问量的时候，它做了以下工作：

1. 为 memcached 缓存中的对应数据项加 1，如果该数据项不存在，则创建该数据项，并且赋值为 1，代表这个页面是第一次被访问；

2．如果 memcached 缓存中存在对应数据项，并且累加后的数值为 1000，则将这个数据项置 0，同时更新数据库，将数据库中的对应数值加 1000。

也就是说，改造后的程序每经历 1000 次递增后才写一次数据库，究竟效果如何呢？我们再来进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /writecache_memcache.php
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   3.599 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      1690000 bytes
HTML transferred:       0 bytes
Requests per second:    2778.24 [#/sec] (mean)
Time per request:       179.970 [ms] (mean)
Time per request:       0.360 [ms] (mean, across all concurrent requests)
Transfer rate:          458.52 [Kbytes/sec] received
```

吞吐率提高了大约 46%，这同样是一个不小的飞跃。所以，如果你的数据库因为大量的写操作而繁忙不堪，那么仔细考虑一下，哪些写操作可以缓存在 memcached 中呢？

1.5 监控状态

作为一个分布式缓存系统，memcached 可以非常出色地完成你交给它的工作，但这并不代表你可以对它放任不管，相反，我们需要知道它的运行状况。memcached 提供了这样的协议，可以让你获得它的实时状态，我们通过 PHP 扩展可以十分容易地做到。

以下的代码片段用来获取 memcached 的状态：

```
<?php
$memcache = memcache_connect('10.0.1.12', 11711);
$stats = $memcache ->getStats();
var_dump($stats);
?>
```

我们来看一下包含了 memcached 运行状态的数组，如下所示：

```
array(22)
{
  ["pid"]=> string(4) "3950"
  ["uptime"]=> string(8) "14670598"
  ["time"]=> string(10) "1239857310"
  ["version"]=> string(5) "1.2.2"
  ["pointer_size"]=> string(2) "32"
  ["rusage_user"]=> string(12) "14261.343278"
  ["rusage_system"]=> string(12) "65035.856487"
  ["curr_items"]=> string(8) "24144013"
  ["total_items"]=> string(9) "175459454"
  ["bytes"]=> string(10) "2562440230"
  ["curr_connections"]=> string(2) "13"
  ["total_connections"]=> string(10) "1433357990"
  ["connection_structures"]=> string(3) "164"
  ["cmd_get"]=> string(10) "1414539975"
  ["cmd_set"]=> string(9) "175563032"
  ["get_hits"]=> string(10) "1328926319"
  ["get_misses"]=> string(8) "85613656"
  ["evictions"]=> string(1) "0"
  ["bytes_read"]=> string(13) "1513089335558"
  ["bytes_written"]=> string(13) "4661687357256"
  ["limit_maxbytes"]=> string(10) "4284481536"
  ["threads"]=> string(1) "1"
}
```

真是面面俱到，你可以从这些数据中获得很多的信息，比如 uptime 表示 memcached 持续运行的时间；cmd_get 表示读取数据项的次数；cmd_set 表示更新数据项的次数；get_hits 表示缓存命中的次数；bytes_read 表示读取的总字节数；bytes 表示缓存区已使用空间的大小；limit_maxbytes 表示缓存区空间的总大小。

对于这些丰富的状态信息，我们可以简单地从以下三个方面来看。

空间使用率

> 持续关注缓存空间的使用率，可以让我们知道何时需要为缓存系统扩容，以避免由于缓存空间已满造成的数据被动淘汰，有些数据项在过期之前被 LRU 算法淘汰可能会造成一定的不良后果。

缓存命中率

> 这个话题在此处已经一点都不新鲜了，在前面关于反向代理缓存的章节中，我们曾经详细介绍过影响命中率的因素，它们在这里同样适用。

I/O 流量

> 我们需要关注 memcached 中数据项读写字节数的增长速度，这反映了它的工作量，我们可以从中得知 memcached 是空闲还是繁忙。

同样，我们也希望在监控系统中集成对于 memcached 的监控，如图 1-1 及图 1-2 所示便是在 cacti 监控系统中对 memcached 的某些监控图片，我们会在本书的末尾介绍 cacti 监控系统。

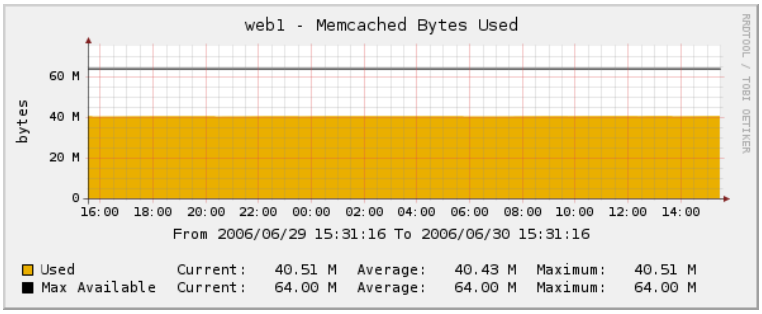


图 1-1　在 cacti 中监控 memcached 的使用率

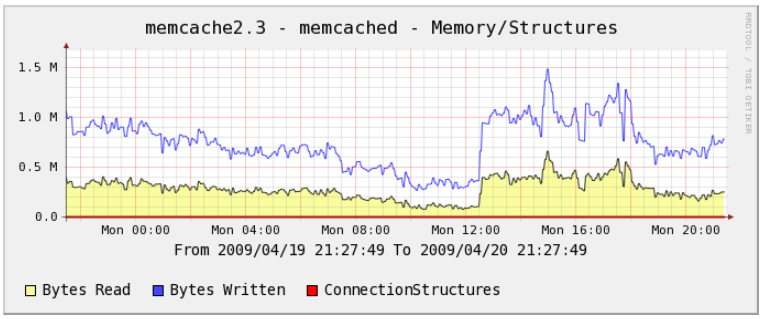


图 1-2　在 cacti 中监控 memcached 的 I/O 流量

1.6　缓存扩展

有很多理由让我们不得不扩展 memcached 的规模，包括并发处理能力和缓存空间容量等，不论是哪个方面达到极限，扩展都在所难免。

对于缓存空间的容量，扩容意味着增加服务器物理内存，这显得不切合实际，而对于并发处理能力，我们知道，memcached

已经在这方面做了很大的努力，这也是它成名的前提。所以，我们只能通过增加新的缓存服务器来达到扩展的目的。

当存在多台缓存服务器后，我们面临的问题是，如何将缓存数据均衡地分布在多台缓存服务器上。现假设我们拥有以下两台缓存服务器，它们的内部 IP 地址如下所示，它们都运行着 memcached。

```
10.0.1.12
10.0.1.13
```

看起来这没什么难的，因为 key-value 类型的数据项本来就相互独立，你可以在连接 memcached 服务器的时候，随便从以上两台服务器中挑选出一台即可。但是，关键在于如何做到“均衡”呢？这个问题归结于如何对数据项进行分区。

相比于关系型数据库的分区设计，key-value 型数据缓存的分区要容易得多，你可以根据数据项的商业逻辑进行分区设计，拿前面的例子来说，我们可以将用户登录状态的缓存部署在独立的一台缓存服务器上，而将访问量统计的缓存部署在另一台独立的服务器上。这样一来，这两台缓存服务器的用途如下：

```
10.0.1.12 -> 用户登录状态缓存
10.0.1.13 -> 访问量统计缓存
```

但是，它们仍然存在以下两个问题：

1. 两台缓存服务器的工作量真的均衡吗？

2. 如果两台缓存服务器仍然不能满足需要，如何继续扩展呢？

第一个问题的答案并不确定，但我想在大多数情况下都是不均衡的，由于以上两种缓存的职责截然不同，所以它们的开销和访问频率也会存在很大的差异，我们当然希望两台缓存服务器的工作量比较对称，达到物尽其用。

对于第二个问题，假如访问量统计缓存需要进一步扩展，我们准备了新的缓存服务器：

```
10.0.1.14
```

接下来我们将访问量统计缓存再次划分，存储在两台缓存服务器上，如何划分呢？我们可以将被统计的所有子站点划分为两部分，让它们分别存储在不同的缓存服务器上，为此，我们需要为每个缓存服务器维护一个一对多的映射表（One To Many Mapping List），里边记录子站点的域名，比如：

```
Group 1 -> www.highperfweb.com, app1.highperfweb.com, app2.highperfweb. com
Group 2 -> app3.highperfweb.com, app4.highperfweb.com, app5.highperfweb. com
```

如此一来，我们重新调整缓存划分方式，三台缓存服务器上的缓存内容如下所示：

```
10.0.1.12 -> 用户登录状态缓存
10.0.1.13 -> 访问量统计缓存 Group 1
10.0.1.14 -> 访问量统计缓存 Group 2
```

但是，假如再次需要扩展，我们还要调整映射表吗？问题是也许你也不知道该怎么划分，实际情况中的映射表可能不像例子中的如此简洁，人工维护这个列表无疑是自找麻烦。另一方面，一个老问题再次浮现出来，你能保证它们的工作量均衡吗？

好吧，我们决定打破这种基于数据项商业逻辑的划分思维，来考虑一种基于 key 的划分方式，这有些类似于后面介绍的数据库水平分区（Sharding）。我们需要设计一种不依赖数据项内容的散列算法，将所有数据项的 key 均衡分配在这三台缓存服务器上。

一个简单而有效的方法是“取余”运算，这就像打扑克时的发牌，让所有数据项按照一个顺序在不同的缓存服务器上轮询，这可以达到较好的相对平衡，想想发牌的动机也正是为了让大家彼此公平。这种方法是一种比较常用的基本散列算法，事实上在很多时候都用得到，而且你可以根据实际情况对它进行改造，达到更好的散列效果。下面我们举个例子。

在“取余”之前，我们先要做一些准备工作，目的是让 key 变成整数，而且尽量唯一。比如对于以下这个 key：

```
article_090222.htm
```

我们先对它进行 md5 运算，这里直接使用 PHP 命令行方式：

```
s-colin:~ # php -r "echo md5('article_090222.htm');"
e6e87fc9f9c2914339a9b7cc4db6055c
```

得到的是一个 32 字节的字符串，同时它也是一个十六进制的长整数，为了减少计算开销，我们取这个字符串的前 5 个字节，然后将它转换为十进制数：

```
s-mat:~ # php -r "echo hexdec('e6e87');"
945799
```

然后将结果进行“模 3”运算：

```
s-mat:~ # php -r "echo 945799 % 3;"
1
```

得到的余数便是缓存服务器的编号，我们的三台缓存服务器应该从 0 开始编号，那么 1 代表了第二台服务器。

看起来真复杂，我们不得不需要一个“缓存连接器”了，我们希望将上面这些运算都放在连接器里，而只需要告诉它 key，接下来选择缓存服务器的事情就拜托给它了。看看连接器的一个例子：

```
<?php
function memcache_connector($key)
{
    $hosts = array(
        '10.0.1.12',
        '10.0.1.13',
        '10.0.1.14'
    );
    $host_index = hexdec(substr(md5($key), 0, 5)) % 3;
    $host = $hosts[$host_index];
    return memcache_connect($host, 11711);
}
?>
```

现在，我们访问缓存的时候只需要这样连接缓存服务器即可：

```
<?php
$memcache = memcache_connector('article_090222.htm');
?>
```

那么，如果还需要继续扩展，一定难不倒你了，将“模 3”的运算变成“模 4”、“模 5”……然后其余的工作就放心地交给连接器去做吧。

这里有一个问题也许你一直在思考，那就是当我们扩展缓存系统后，由于分区算法的改变，会涉及缓存数据需要从一台缓存服务器迁移到另一台缓存服务器的问题，如何迁移呢？事实上，根本不需要考虑分区之间的迁移，因为这是缓存，它应该具备在必要时刻牺牲自己的勇气，当然这是你赋予它的，你必须明白缓存不是持久存储，并且从引入分布式缓存开始就不断地提醒自己。

没错，当调整缓存分区算法后，我们需要时间来等待缓存重建和预热，但这往往并不影响站点的正常运转，前提是你按照前面读缓存和写缓存的理念来进行设计。顺便一提的是，与此相比，数据库规模扩展引发分区（Shard）之间的数据迁移就要复杂得多，后面我们会有专门的章节探讨它。

Web 负载均衡

回顾前面的内容，似乎一直都在回避 Web 规模扩展这个问题，因为我担心过早实施扩展会迷惑我们优化性能的意志。当然，在有些时候进行扩展是显而易见的，比如下载服务由于带宽不足而必须进行的扩展，但是，另一些时候，很多人一看到站点性能不尽如人意，就马上实施负载均衡等扩展手段，真的需要这样做吗？当然这个问题也只有他们自己能回答，除了出于高可用性和就近部署的考虑，大多数情况下这种行为都显得有些过早，也许当你阅读了前面的章节后，你的 Web 服务器已经从 5 台又变回了 1 台，然后你要做的就是回家闭门思过。

那么，是不是一开始就完全不必考虑规模扩展呢？答案完全相反，作为架构师的你，从一开始就要思考未来的扩展计划，并且为扩展而进行架构设计，但是关键在于，你必须能够意识到何时需要实施扩展，并且有足够的数据来证明这种必要性。

值得一提的是，服务器自身硬件的垂直扩展不在我们的讨论之中，这一章我们所谈及的扩展，主要是指水平扩展，我们经常用可扩展性来反映这种扩展能力，所谓可扩展性，实际上是指系统通过扩展规模来提升承载能力的本领，这种本领往往体现在增加物理服务器或者集群节点等方面，可以说，这种本领越强，承载能力可提升的空间就越大。但是，这种本领总是受到或多或少的制约，比如，我们之所以不讨论单机垂直扩展，就是因为单机的扩展能力非常有限，很快就会遇到技术制约，并且随着规模的增大而越来越昂贵，的确，即使最强大的单机也无法满足我们的需要。

> 提示：
>
> 从哲学的角度看，可扩展的能力也是万物生存之道，《道德经》中有“道生一，一生二，二生三，三生万物”，《易经》中有“无极生太极，太极生二仪（阴阳），二仪生四象（太阳、太阴、少阳、少阴），四象生八卦，八卦生六十四艾，六十四艾生宇宙万象”。可见我们的古人早已将无限扩展视为永恒的大道。

2.1 一些思考

对于 Web 站点的水平扩展，负载均衡是一种常见的手段，在介绍负载均衡的多种实现方法之前，我们先来思考一些问题。

我们先将目光转向一个类比的例子，假如某公司有一个小型团队，需要承担一定的工作量，开始的时候，大家各尽其能，非常轻松地就可以完成工作，不亦乐乎。但是，随着公司的发展，这个团队的工作量逐渐增大，超出了团队成员的承受能力，工作完成质量开始下降。

外包

出于一些考虑，这个团队决定将一部分工作任务外包给其他公司来做，以减轻自己的负担，同时由团队中的一个人负责与外包公司进行长期沟通，这里我们称他为外包接口人。

显然，外包使得团队承载能力的扩展成为可能，而且随着工作任务的继续增加，一家外包公司无法应付，这个团队又找到了更多的外包公司同时进行合作，外包接口人负责与这些公司分别进行沟通。

这样一来，公司只需要花费一个人力和一些费用，就可以完成大量持续增加的工作任务。

接口人

突然有一天，外包接口人由于病假没能来公司上班，这下公司着急了，因为与外包公司的沟通工作需要时刻进行，而只有这名外包接口人熟悉这个工作，这使得原本有序的外包工作不得不受到严重影响，多家外包公司暂时停了下来，直到第二天这位外包接口人回到公司后，外包工作才恢复正常。

显然，这位外包接口人非常关键，他的缺席将会影响整个外包工作的正常进行，这也许是致命的，我们将这种情况称为单点故障（Single Point of Failure），如果公司只依赖一个因素、系统、设备或人，就会暴露出单点故障的隐患，所以，我们应该尽量避免单点故障。

为此，公司为这位外包接口人配备了一名助理，全力协助他的工作，并且当他偶尔不在公司的时候，助理可以很好地充当他的角色继续工作。

工作量分配

刚才说到，外包接口人需要跟多个外包公司进行沟通，并且将工作任务持续不断地分配给他们，那么，在任务分配过程中，可能发生这样一些情况：

- 给有些外包公司分配了太多的任务，其中一些任务没能按时完成；
- 有些外包公司比较闲，可是却没能及时给他们分配任务；
- 有些外包公司业务能力差，却给他们分配了高难度的任务，花费了大量的时间，最后还可能无法完成；
- 有些外包公司业务能力强，却给他们分配了非常简单的任务，支出了不必要的高额外包费用。

如此一来，我们看到一个最优化任务分配的问题，当然，这个问题是需要外包接口人来考虑的，同时，他需要借助一些过程管理方法来掌握各外包公司的进度和状态，了解各外包公司的“负载”，以帮助自己更加有效地分配任务，实现外包工作的负载均衡（Load Balancing，LB）。

风险管理

尽管我们已经在一定程度上避免了外包接口人的单点故障，降低了风险，但是外包公司方面仍然有可能出现问题，在必要的时候，我们需要采取行动，将任务快速转移给其他的外包公司，当然，前提是有足够的备用外包公司可以选择。

从避免单点故障到准备备用方案，都是降低外包工作风险的一系列措施，同时也保障了外包工作的不间断运转，或者称为高可用性（High Availability，HA）。

制约

当更多的工作任务需要外包时，这位接口人的工作有点吃不消了，因为：

- 一个接口人负责大量的任务，与多家外包公司分别进行沟通，这几乎花费了他全部的工作时间；
- 一个接口人管理多家外包公司，已经超出了他的管理能力。

显然，这些原因也是制约接口人处理更多外包工作的因素，这些因素限制了外包工作的无限扩展。

这时，公司决定设立更多的外包接口人，成立一个新的团队，其中每位外包接口人分别负责管理一部分外包公司，并跟进相关的外包工作。最终的外包工作关系图如图 2-1 所示。

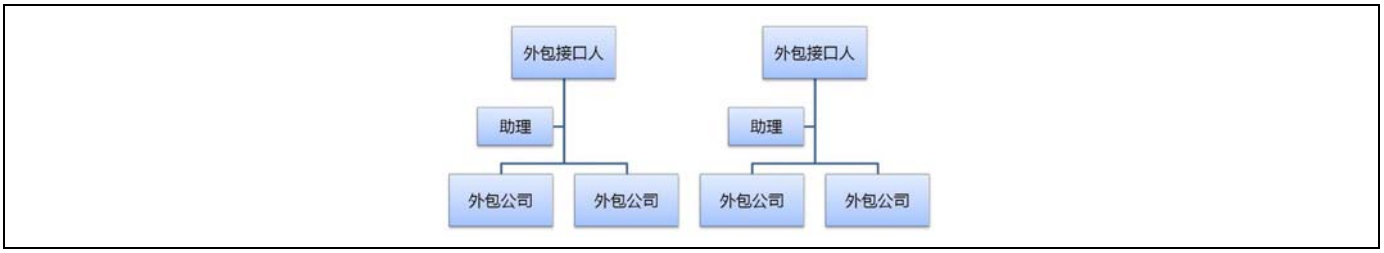


图 2-1　外包工作关系图示例

在以上假想的外包工作场景中，我们遇到了一系列的问题，并提出了解决方案，这些问题在 Web 站点扩展的过程中同样存在，你仍然可以带着这些问题继续前进，接下来我们将回到 Web 站点扩展的主题中，探讨如何实现可扩展的 Web 负载均衡系统，

当然，除了我们关注的性能之外，高可用性也会有所涉及。

2.2 HTTP 重定向

对于 HTTP 重定向，你一定不陌生，它可以将 HTTP 请求进行转移，在 Web 开发中我们经常会用它来完成自动跳转，比如用户登录成功后跳转到相应的管理页面。

这种重定向完全由 HTTP 定义，并且由 HTTP 代理和 Web 服务器共同实现。很简单，当 HTTP 代理（比如浏览器）向 Web 服务器请求某个 URL 后，Web 服务器可以通过 HTTP 响应头信息中的 Location 标记来返回一个新的 URL，这意味着 HTTP 代理需要继续请求这个新的 URL，这便完成了自动跳转。当然，如果你自己写了一个 HTTP 代理，也可以不支持重定向，也就是对于 Web 服务器返回的 Location 标记视而不见，虽然这可能不符合 HTTP 标准，但这完全取决于你的应用需要。

也正是因为 HTTP 重定向具备了请求转移和自动跳转的本领，所以除了满足应用程序需要的各种自动跳转之外，它还可以用于实现负载均衡，以达到 Web 扩展的目的。

熟悉的镜像下载

你也许有过从 php.net 下载 PHP 源代码的经历，那么你是否注意过它是如何实现镜像下载的呢？没错，那就是 HTTP 重定向，而镜像下载的目的便是实现负载均衡，值得一提的是，这里我们暂且认为所有镜像服务器上的内容都是一致的，后续章节中我们会介绍内容分发与同步的一些方法和策略。

我们来看这次下载的重定向过程，首先，记住我们请求的 URL 为：

```
www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
```

接下来，我们在浏览器中打开这个地址，并且通过 HttpWatch 监视 HTTP 请求和响应，如下所示：

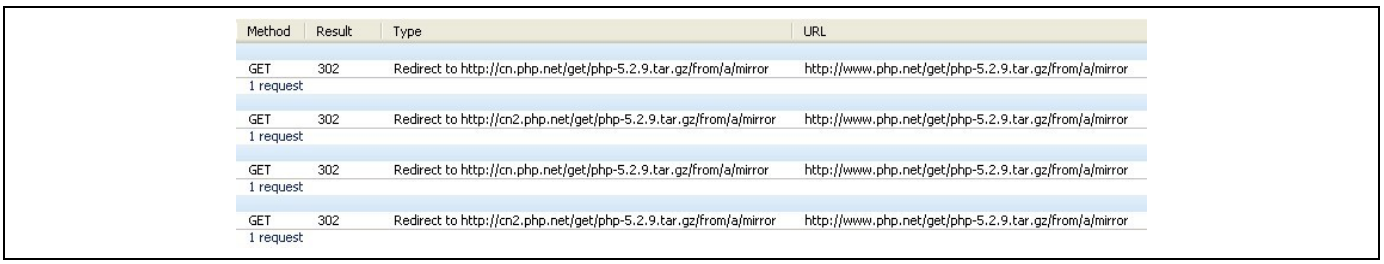
```
GET /get/php-5.2.9.tar.gz/from/a/mirror HTTP/1.1
Accept: image/gif, image/x-xbitmap,image/jpeg,image/pjpeg, application/ x-shockwave-flash,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Referer: http://www.php.net/downloads.php
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB6; CIBA; .NET CLR 2.0.50727)
Host: www.php.net
Connection: Keep-Alive
HTTP/1.1 302 Found
Date: Wed, 13 May 2009 12:08:51 GMT
Server: Apache/1.3.41 (Unix) PHP/5.2.9RC3-dev
X-Powered-By: PHP/5.2.9RC3-dev
Content-language: en
X-PHP-Load: 0.60546875, 0.568359375, 0.5634765625
Location: http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

可以看到，HTTP 响应的状态码为 302，并通过 Location 标记返回了新的 URL，这个 URL 位于 cn.php.net 这个新的域名下，按照命名规则，我们知道这个域名指向部署在中国的服务器，这样一来，我们刚才向 php.net 主站点发出的下载请求便被转移到了国内的服务器上，这相当于主站点将一部分负载转移到了其他服务器上。

接下来，我们再次请求刚才那个位于 php.net 主站点的 URL，这次获得的 HTTP 响应如下所示：

```
HTTP/1.1 302 Found
Date: Wed, 13 May 2009 12:08:51 GMT
Server: Apache/1.3.41 (Unix) PHP/5.2.9RC3-dev
X-Powered-By: PHP/5.2.9RC3-dev
Content-language: en
X-PHP-Load: 0.60546875, 0.568359375, 0.5634765625
Location: http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

这回 Location 中的 URL 发生了变化，域名变成了 cn2.php.net。我们继续重复请求多次，通过 HttpWatch 的截图（如图 2-2 所示）我们可以看到这些请求的重定向过程。



Method	Result	Type	URL
GET	302	Redirect to http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			

图 2-2　php.net 站点的 HTTP 重定向监视

可见，php.net 使用了两台位于国内的镜像服务器，分别对应以下的 URL：

```
cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror
cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror
```

通过图 2-2 我们看到两个镜像地址轮流上阵，但实际上它们并不是严格地交替出现，通过持续观察，你会发现它们似乎是随机出现，但是从整体来看，这已经做到了一定程度上的负载均衡。

在这里，php.net 主站点负责进行地域来源判断以及选择镜像服务器，值得一提的是，这里的重定向方案，在实现负载均衡的同时，也达到了就近访问的目的，加快了用户的下载速度，并且在一定程度上避免了国际带宽的浪费。

当然，通过不同的地域来源来转移请求只是负载均衡的一种策略，在网络 I/O 成为主要瓶颈（比如下载服务）的时候，这种策略的优势表现得尤为突出。但是在其他一些时候，通过地域来源来划分请求并不那么合理，比如来自各个地域的请求到达 Web 服务器的响应时间差异不大，并且各地域的请求比例存在较大差异时，这种策略显然不太适合，它会造成各镜像服务器的负载分配不均衡，从而造成资源的浪费。

用代码来实现

相比于随后要介绍的其他负载均衡方法，基于 HTTP 重定向的方式非常简单，基本上完全由 Web 应用程序即可实现，但是它的性能表现如何呢？我们这里来实现一个基于 HTTP 重定向的负载均衡系统。

假设我们的站点域名为 www.highperfweb.com，它指向的 IP 地址为 10.0.1.100，我们希望将所有对于站点首页的请求随机转移到其他三台实际服务器上，为此，我们进行了以下的域名 DNS 设置：

```
www.highperfweb.com.     IN      A       10.0.1.100
www1.highperfweb.com.    IN      A       10.0.1.101
www2.highperfweb.com.    IN      A       10.0.1.102
www3.highperfweb.com.    IN      A       10.0.1.103
```

可以看到，我们为其他三台服务器准备了新的二级域名，不过看起来毫无内涵，这里我们暂时认为这三台服务器都拥有一致的内容。

这样一来，你完全可以告诉用户直接访问某一台服务器，比如 www2.highperfweb.com，已达到分散主站工作量的目的，但是很显然，这样做不会有任何好处，这等于让用户跳过了我们的负载均衡策略，而且也没有多少用户愿意记住这些古怪的域名，架构师永远不要给用户添麻烦。

提示：

在这个例子中的 IP 地址都是内部地址，但在实际应用中，你必须使用用户可以访问的互联网公开 IP 地址。

接下来，我们编写了首页 index.php 的代码，如下所示：

```
<?php
$domains = array(
    'www1.highperfweb.com',
    'www2.highperfweb.com',
```

```
    'www3.highperfweb.com'
);
$index = substr(microtime(), 5, 3) % count($domains);
$domain = $domains[$index];
header("Location: http://$domain");
?>
```

看看这段代码，很简单，当用户每次访问站点入口，也就是 index.php 的时候，程序会随机挑选三台服务器中的一台，然后返回重定向指令。我们用 curl 来试试看：

```
s-colin:~ # curl -i www.highperfweb.com
HTTP/1.1 302 Found
Date: Thu, 14 May 2009 02:15:59 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Location: www3.highperfweb.com
Content-Length: 0
Content-Type: text/html
```

看来这次 www3.highperfweb.com 这台服务器比较幸运，它被随机选中，在随后的多次请求中，三台镜像服务器对应的域名都分别出现在了 Location 标记中，从整体上看，首页请求转移到三台服务器的次数基本上是相等的，所以我们已经做到了一定程度的均衡。

这里为什么没有采用依次轮询的重定向策略呢？也就是 RR（Round Robin），它是一种基本的调度算法，在这个例子中如果采用 RR 方式的话，站点首页程序必须将所有请求按照顺序依次转移给三台服务器，实现绝对意义上的均衡，但是，与此同时，性能上的代价也是不可避免的，因为：

- HTTP 本身是无状态的，如果要实现按顺序转移请求，我们必须将最后一次转移至的实际服务器序号进行持久化保存，以便在多次 HTTP 请求之间可以共享，比如将它存入 Memcache，显然，这将会带来额外的开销；
- 要实现绝对的按顺序转移，必然会使得主站点请求转移计算的并发性大打折扣，因为转移状态（最后一次转移至的实际服务器序号）是互斥资源，转发程序必须通过一定的锁机制来保证任何时刻只能有一个请求可以修改它。

这就好比一个人做事，独立决策并执行的速度肯定要比上报领导等待批准快。那么，你可能需要在两者之间进行权衡，随后我们将通过测试来比较一下它们的性能差异。

重定向的性能和扩展能力

这里我们指的性能，实际上是针对主站点来说的，因为它负责转移请求到其他服务器，就像前面故事中的外包接口人一样重要，决定着整个负载均衡系统的扩展能力，也意味着整个系统的最大承载能力。

对于刚才的随机转移方案，我们通过 ab 对主站点的首页程序进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   1.553166 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      2440244 bytes
HTML transferred:       0 bytes
Requests per second:    6438.46 [#/sec] (mean)
Time per request:       77.658 [ms] (mean)
Time per request:       0.155 [ms] (mean, across all concurrent requests)
Transfer rate:          1534.29 [Kbytes/sec] received
```

我们看到吞吐率为 6438.46reqs/s，这意味着什么呢？我们知道主站点的这个首页程序只是负责转移请求，刚才也看到了它的代码实现，它的开销并不大，但是位于其他三台服务器上实际首页的吞吐率如何呢？也许它们由于频繁的数据库访问而只有几百的吞吐率，也许通过实施一些缓存策略可以达到几千的吞吐率，你甚至可以花点心思通过完全静态化来达到上万的吞吐

率，前面的章节对此已经有很详细的介绍，但是在这里，这个问题其实并不是我们所关心的，而我们关心的是主站点的最大吞吐率和它的扩展能力有什么关系。

其实这很简单，通过 HTTP 重定向的原理以及我们采取的转移均衡策略，不难得知，当主站点首页的吞吐率达到 6438.46reqs/s 这个极限时，它转移给其他三台服务器的请求均相当于这个吞吐率的 1/3，即 2146.15reqs/s，那么，我们要做的就是保证这三台服务器能够承载这样的压力吗？当然如果你能做到的话最好，但如果不行，那也正常，不要忘了，扩展是把握在我们自己手里的。

我们假设这三台实际服务器的首页能承受的最大吞吐率为 500reqs/s，那么理论上当主站点的实际吞吐率小于 1500reqs/s 的时候，实际服务器可以在承受范围内提供服务。当主站点的实际吞吐率大于 1500reqs/s 后，我们便需要增加实际服务器，但是理论上最多可以增加到 13 台，这取决于主站点首页的最大吞吐率（6438.46reqs/s），它几乎是实际服务器上首页最大吞吐率的 13 倍。

RR 策略下的性能

还记得前面提到的 RR 方式吗？我们现在对主站点首页程序进行修改，希望它能够更加均衡地转移请求到多台实际服务器，代码如下所示：

```
<?php
$memcache = memcache_connect('10.0.1.200', 11711);
$domain_index = $memcache->increment('last_index', 1);
if ($domain_index === false || $domain_index > 100000)
{
        $memcache->set('last_index', 0, null, 0);
        $domain_index = 0;
}
$domains = array(
    'www1.smartdeveloper.cn',
    'www2.smartdeveloper.cn',
    'www3.smartdeveloper.cn'
);
$domain = $domains[$domain_index % count($domains)];
header("Location: http://$domain");
?>
```

可以看到，我们利用 Memcache 服务器创建了一个共享计数器 last_index，在每次转移请求之前，都要对计数器进行递增操作，这是一个原子操作，所以保证了计数器对于所有请求的一致性。我们来对这个新的首页程序进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   2.935200 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      2439868 bytes
HTML transferred:       0 bytes
Requests per second:    3406.92 [#/sec] (mean)
Time per request:       146.760 [ms] (mean)
Time per request:       0.294 [ms] (mean, across all concurrent requests)
Transfer rate:          811.53 [Kbytes/sec] received
```

这次的首页吞吐率只有前边的 53%，而实际服务器的最大数量也由前面的 13 台变为 7 台，这意味着整个系统的承载能力缩水了一半左右。

如此大幅度的性能下降，也印证了前面我们对于顺序转移性能代价的分析，另一方面，也许你觉得这与我们使用 Web 应用程序来实现 RR 调度策略有关，那么，我们来看看 Apache 的 mod_rewrite 模块，它可以轻松地支持 RR 重定向，因为 Web 服务器要维护一个全局资源并不困难。

我们对 Apache 的 Vhost 进行以下配置：

```
<VirtualHost *:80>
    DocumentRoot /data/www/highperfweb/htdocs
    ServerName www.highperfweb.com
    RewriteEngine on
    RewriteMap    lb      prg:/data/www/lb.pl
    RewriteRule   ^/(.+)$ ${lb:$1}
</VirtualHost>
```

这里可以看到我们引入了一个第三方脚本 lb.pl，它是 mod_rewrite 支持的一种可编程模式，这个脚本会跟随 Apache 一起启动，并在自己的逻辑空间中运行，直到 Apache 停止后被释放。我们来看看这个脚本的代码，如下所示：

```
#!/usr/bin/perl
$| = 1;
$name   = "www";
$first  = 1;
$last   = 3;
$domain = "highperfweb.com";
$cnt = 0;
while (<STDIN>)
{
    $cnt = ($cnt + 1) % $last;
    $server = sprintf("%s%d.%s", $name, $cnt + $first, $domain);
    print "http://$server/$_";
}
```

这个脚本不难理解，Apache 每次接入新的请求后，便会触发 while 循环，这时候程序会计算出当前应该使用的实际服务器序号，然后根据事前定义的规则组合成最终实际服务器的域名。我们再次通过 ab 进行同样的压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        223 bytes
Concurrency Level:      500
Time taken for tests:   2.956 seconds
Complete requests:      10000
Failed requests:        4035
   (Connect: 0, Receive: 0, Length: 4035, Exceptions: 0)
Write errors:           0
Non-2xx responses:      10000
Total transferred:      4657753 bytes
HTML transferred:       2246617 bytes
Requests per second:    3383.02 [#/sec] (mean)
Time per request:       147.797 [ms] (mean)
Time per request:       0.296 [ms] (mean, across all concurrent requests)
Transfer rate:          1538.79 [Kbytes/sec] received
```

从结果来看，吞吐率并没有提高，而且更重要的是，在这次压力测试中，失败率非常高，Length 为 4035，达到总请求数的 40% 左右，这意味着 ab 认为有 4035 个请求的响应数据长度可疑，也就是说这些请求的响应数据可能不是预期的正确结果。为了获得失败的处理结果，我们在压力测试的同时，通过其他会话进行请求，偶尔会获得如下所示的结果：

```
s-colin:~ # curl -i http://www.highperfweb.com/
HTTP/1.1 400 Bad Request
Date: Wed, 20 May 2009 08:37:22 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```

可见，这种失败的处理结果是非常严重的。我们将压力测试的模拟并发用户数从 500 降到了 2，失败率仍然为总请求数的 10% 左右，当我们将模拟并发用户数降为 1 的时候，全部请求都处理成功，结果如下所示：

```
Server Software:     Apache/2.2.11
Server Hostname:     www.highperfweb.com
```

```
Server Port:            80
Document Path:          /
Document Length:        223 bytes
Concurrency Level:      1
Time taken for tests:   2.653 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      4740000 bytes
HTML transferred:       2230000 bytes
Requests per second:    3769.66 [#/sec] (mean)
Time per request:       0.265 [ms] (mean)
Time per request:       0.265 [ms] (mean, across all concurrent requests)
Transfer rate:          1744.94 [Kbytes/sec] received
```

的确，我们看到 Apache 在并发环境下对于 mod_rewrite 模块的可编程脚本机制的处理很不理想，当然，这取决于 Apache 在该处的实现。理论上在 Web 服务器中实现专用的重定向策略逻辑，其性能肯定要优于通过 Web 应用程序来实现。也许你可以自己编写专用的 Web 服务器或者插件来实现高效的 RR 重定向，但是，无论如何，顺序调度的性能总是比不上随机调度的性能，特别是在这里，负责转移请求的逻辑本身并没有太大开销，所以更加凸显了顺序调度的性能代价。

如果你使用 RR 策略的目的在于希望做到请求转移的绝对均衡，那么可以肯定的一点是，根据概率统计理论，随着吞吐率的增加，随机调度也会逐渐趋近于顺序调度的均衡效果。所以，你会考虑使用需要一定性能代价的 RR 方式吗？

这里顺便提一下，你也可以考虑使用其他的调度策略，比如根据用户的 IP 地址来散列计算实际服务器序号，这样做同样可以避免 RR 策略的性能代价，事实上凡是 HTTP 请求处理逻辑本身可以独立决策的调度策略，都拥有较高的性能表现。另外，这种根据用户 IP 地址来转移请求的策略可以使得用户每次都访问同一台实际服务器，这种策略在后面我们会再次介绍，它为一些 Web 应用提供了很好的支持。

更多考虑

前面我们费尽心思来实现请求转移次数的均衡，我们知道这样做是没错的，它可以为多台实际服务器平均分配工作量，最大程度地提高利用率。但是在实际环境中，次数的均衡真的代表负载的均衡吗？我们必须思考这个问题。

你是否还记得我们刚才的例子都是针对站点首页的重定向，也就是站点的入口，而站点内页的链接地址我们一般在首页上采用相对地址或者根相对地址，这样一来，一旦用户通过入口时被转移到某台实际服务器后，用户的一系列请求将直接进入这台实际服务器，我们知道不同用户对站点内页的访问深度是不同的，这也是我们无法控制的，这样一来，多台实际服务器的负载差异是不可预料的，而主站点对此却一无所知。

而更加让人无奈的是，你无法保证用户始终从站点首页进入站点，也就是说总有或多或少的用户会跳过你精心设计的主站点调度程序，这的确让人很尴尬，也许用户已经收藏了某个实际服务器上的 URL，比如：

```
http://www3.highperfweb.com/book/load_balancing.php
```

随后即便是你改变了转移策略，但这个 URL 仍然会指向原来的实际服务器，你对它毫无办法，只能想出各种歪门邪道的办法来再次进行重定向，于是策略开始变得混乱和不可控。

所以，大多数情况下通过重定向来实现整个站点的负载均衡，并不那么让人满意，随后我们会探讨其他的扩展方法来实现站点负载均衡。

相比之下，对于文件下载、广告展示等一次性的请求，主站点调度程序可以牢牢地把握控制权，实际服务器的 URL 甚至可以含蓄地隐藏起来，与此同时，这种一次性的请求，也比较容易让多台实际服务器保持均衡的负载，但是也必须考虑一些现实的问题，比如分配给不同实际服务器下载的文件可能尺寸差异较大，我们需要在次数分配均衡的情况尽量保证文件尺寸分配均衡，也就是带宽使用分配均衡，这也许需要借助于应用程序，你可以记录下给每个实际服务器派发的下载文件的尺寸，从而在每次下载转移前挑选你认为比较轻松的实际服务器，但这样做存在风险，你可能是在毫无根据地指手画脚，因为某个用户可能请求下载一个 1GB 的文件，却在下载了 1%后突然终止，而你却毫不知情，仍然以为某台实际服务器在卖力工作，随

后一段时间你对这台服务器实施保护，而它却无所事事。

为了使提供下载服务的多台实际服务器比较均衡地使用带宽，另一个方法值得考虑，事实上它也是负载均衡系统中比较重要的一部分，那就是负载反馈。在这里，我们可以让主站点的定时任务不断获取每个实际服务器的实时流量，这可以通过 SNMP 获得原始数据并计算得出，这些数据将作为下载转移的权重参考。也许你觉得在请求转移逻辑中加入各实际服务器流量权重分析会带来额外的开销，但是，相比于下载的时间开销而言，这些额外的开销实在是九牛一毛。

刚才提到的下载服务只是一个例子，除此之外，对于不同的应用场景，我们仍然需要认真考虑基于重定向的负载均衡是否适用，虽然我们不能一一列举，但是有一点是可以肯定的，我们需要权衡转移请求的开销和处理实际请求的开销，前者相对于后者越小，那么重定向的意义就越大，刚才的下载转移就是个很好的例子。

2.3 DNS 负载均衡

我们知道，DNS 负责提供域名解析服务，当我们访问某个站点时，实际上首先需要通过该站点域名的 DNS 服务器来获取域名指向的 IP 地址，在这一过程中，DNS 服务器完成了域名到 IP 地址的映射，同样，这种映射也可以是一对多的，这时候，DNS 服务器便充当了负载均衡调度器（也称均衡器），它就像前面提到的重定向转移策略一样，将用户的请求分散到多台服务器上，但是它的实现机制完全不同。

多个 A 记录

在 DNS 的各种记录类型中，A 记录你一定不陌生，它负责实现 DNS 的基本功能，用来指定域名对应的 IP 地址，常见的比较成熟的 DNS 系统如 Linux 的 bind，以及 Windows 的 DNS 服务等，都支持为一个域名指定多个 IP 地址，并且可以选择使用各种调度策略，常见的便是 RR 方式。

我们先来看看其他站点的 A 记录设置，这毫无疑问是公开的，不存在任何私密，我们使用 dig 命令来查看 facebook.com 的 A 记录设置，如下所示：

```
s-colin:~ # dig facebook.com
; <<>> DiG 9.3.2 <<>> facebook.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 48314
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 4, ADDITIONAL: 4
;; QUESTION SECTION:
;facebook.com.                  IN      A
;; ANSWER SECTION:
facebook.com.           2254    IN      A       69.63.176.140
facebook.com.           2254    IN      A       69.63.178.11
facebook.com.           2254    IN      A       69.63.184.142
;; AUTHORITY SECTION:
facebook.com.           34766   IN      NS      dns04.sf2p.tfbnw.net.
facebook.com.           34766   IN      NS      dns05.sf2p.tfbnw.net.
facebook.com.           34766   IN      NS      ns1.facebook.com.
facebook.com.           34766   IN      NS      ns2.facebook.com.
;; ADDITIONAL SECTION:
ns1.facebook.com.       54020   IN      A       204.74.66.132
ns2.facebook.com.       54020   IN      A       204.74.67.132
dns04.sf2p.tfbnw.net.   44300   IN      A       69.63.176.8
dns05.sf2p.tfbnw.net.   44300   IN      A       69.63.176.9
;; Query time: 1 msec
;; SERVER: 202.102.152.3#53(202.102.152.3)
;; WHEN: Thu May 14 10:26:21 2009
;; MSG SIZE  rcvd: 232
```

这里我们首先看粗体部分，可以看到 facebook.com 拥有三个 A 记录，分别指向三个不同的 IP 地址，这意味着什么呢？接下来我们通过 ping 命令来测试 facebook.com 域名的 A 记录解析，连续三次的测试结果如下所示：

```
s-mat:~ # ping facebook.com
PING facebook.com (69.63.176.140) 56(84) bytes of data.
s-mat:~ # ping facebook.com
PING facebook.com (69.63.184.142) 56(84) bytes of data.
s-mat:~ # ping facebook.com
PING facebook.com (69.63.178.11) 56(84) bytes of data.
```

果然，DNS 服务器使用三个不同的 IP 地址来轮流解析 facebook.com 域名，实现了 RR 调度策略。

也许你觉得这里 DNS 服务器的职能有点类似于前面提到的重定向调度程序，的确有点，我们现在就对前面的 DNS 设置做一番修改，让 DNS 来取代所谓主站点的工作，修改后的 DNS 设置如下所示：

```
www1.highperfweb.com.    IN      A       10.0.1.101
www2.highperfweb.com.    IN      A       10.0.1.102
www3.highperfweb.com.    IN      A       10.0.1.103
www.highperfweb.com.     IN      CNAME   www1.highperfweb.com.
www.highperfweb.com.     IN      CNAME   www2.highperfweb.com.
www.highperfweb.com.     IN      CNAME   www3.highperfweb.com.
```

这里我们仍然保留了 www1、www2 和 www3 的 A 记录，同时增加了三个别名（CNAME）记录，可以清楚地看到，www.highperfweb.com 指向了这三个 A 记录，从实际效果来看，以上设置也等同于：

```
www.highperfweb.com.     IN      A       10.0.1.101
www.highperfweb.com.     IN      A       10.0.1.102
www.highperfweb.com.     IN      A       10.0.1.103
```

这两种设置几乎没有什么性能上的差别，但是前一种设置也许可以满足你的一些需要，比如：

- 如果你之前使用了重定向方案，那么你可能希望在一段时期内兼顾到老用户，如果他们愿意，仍然可以通过收藏的类似 www3 等域名来访问站点，并期待你引导他们放弃旧的 URL。
- 当你拥有很多 DNS 记录的时候，这样做使你的维护更加容易，比如你希望多个二级域名都指向同一个 IP 地址，那么你可以用一个别名来代替 IP 地址，随后当你需要变更 IP 地址的时候，只需要修改别名的 A 记录即可。当然，你可以起一些有意义的别名，而不像这里的 www1、www2 和 www3。

可见，DNS 负载均衡的实现主要依赖于 DNS 服务器的设置，如果你的站点拥有自己的 DNS 服务器，那么以上的设置对于 DNS 管理员来说并不困难，但是，大多数站点仍然使用第三方 DNS 服务商，幸运的是，现在有很多 DNS 服务商完全支持多个 A 记录的轮询设置，你可以根据需要来挑选。

扩展能力和可管理性

和前面基于重定向的负载均衡方式相比，基于 DNS 的方案完全节省了所谓的主站点，或者说 DNS 服务器已经充当了主站点的职能。

作为调度器，DNS 服务器本身的性能我们几乎不用担心，因为事实上，DNS 记录可以被用户浏览器或者互联网接入服务商的各级 DNS 服务器缓存，只有当缓存过期后才会重新向该域名的 DNS 服务器请求解析，所以即便是采用了 RR 调度策略，我们也几乎不会遇到 DNS 服务器成为性能瓶颈的问题。

另一方面，我们一般会配备至少两台以上的 DNS 服务器来提高可用性，还记得刚才我们通过 dig 命令获得 facebook.com 的 DNS 服务器列表吗？通过 NS 类型记录我们可以看到，它使用了 4 台 DNS 服务器。

既然负载均衡调度器不存在性能的制约，那么在这种方案下，整个系统的扩展能力理论上将可以被无限放大，比如我们通过 dig 命令看到 www.sina.com.cn 指向了 16 台服务器（甚至更多，但这已经足够了）。

```
www.sina.com.cn.         IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.     IN      CNAME   dorado.sina.com.cn.

dorado.sina.com.cn.      IN      A       60.215.128.148
dorado.sina.com.cn.      IN      A       60.215.128.149
dorado.sina.com.cn.      IN      A       60.215.128.123
dorado.sina.com.cn.      IN      A       60.215.128.124
dorado.sina.com.cn.      IN      A       60.215.128.125
dorado.sina.com.cn.      IN      A       60.215.128.126
dorado.sina.com.cn.      IN      A       60.215.128.127
dorado.sina.com.cn.      IN      A       60.215.128.128
dorado.sina.com.cn.      IN      A       60.215.128.129
dorado.sina.com.cn.      IN      A       60.215.128.130
dorado.sina.com.cn.      IN      A       60.215.128.131
dorado.sina.com.cn.      IN      A       60.215.128.132
dorado.sina.com.cn.      IN      A       60.215.128.133
```

```
dorado.sina.com.cn.      IN      A       60.215.128.134
dorado.sina.com.cn.      IN      A       60.215.128.135
dorado.sina.com.cn.      IN      A       60.215.128.136
```

当你不必为扩展担忧的时候，另一方面的问题便开始暴露，如何管理这么多的服务器呢？诸如内容同步、数据共享、状态监控等问题都尤为重要，不过还好，在后续章节中我们会详细探讨这些内容，在这里，它们还不至于阻碍我们对 DNS 负载均衡的热衷。

智能解析

尽管基于 HTTP 重定向的负载均衡系统受到主站点性能的制约，但是不可否认这种方案中的调度策略具有非常好的灵活性，你完全可以通过 Web 应用程序实现任何你能想到的调度策略。

相比之下，为 DNS 服务器开发自定义的调度策略就不那么容易了，但幸运的是，类似 bind 这样的 DNS 服务器软件提供了丰富的调度策略供你选择，其中最常用的就是根据用户 IP 来进行智能解析，这意味着 DNS 服务器可以在所有可用的 A 记录中寻找离用户最近的一台服务器。

当然，如何利用这种策略，完全取决于你，你可以为用户比较集中的一些城市提供专用的服务器，接入城市核心节点，也可以为各互联网运营商网络中的用户提供专用的服务器，并接入该运营商骨干节点，要做到这些，你还需要收集到相应的网络地址分布数据，并添加到 DNS 服务器的智能解析策略中。

拿刚才的 www.sina.com.cn 域名来说，我们用另一台位于其他城市的服务器再次进行 dig 命令操作，结果如下所示：

```
s-web:~ # dig www.sina.com.cn
; <<>> DiG 9.3.2 <<>> www.sina.com.cn
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37334
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;www.sina.com.cn.              IN      A
;; ANSWER SECTION:
www.sina.com.cn.        59      IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.    239     IN      A       218.30.66.101
;; Query time: 6 msec
;; SERVER: 218.30.19.40#53(218.30.19.40)
;; WHEN: Fri May 22 14:50:38 2009
;; MSG SIZE  rcvd: 71
```

这次看到的 A 记录指向了一个新的 IP 地址，这说明 DNS 服务器根据我们的来源进行了智能解析。有趣的是，我们再次更换一个城市，结果又发生了变化，如下所示：

```
colin-mini:~ root# dig www.sina.com.cn
; <<>> DiG 9.4.2-P2 <<>> www.sina.com.cn
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43203
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;www.sina.com.cn.              IN      A
;; ANSWER SECTION:
www.sina.com.cn.        1       IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.    336     IN      A       202.108.33.32
;; Query time: 10 msec
;; SERVER: 10.0.1.1#53(10.0.1.1)
;; WHEN: Fri May 22 14:58:09 2009
;; MSG SIZE  rcvd: 71
```

的确，IP 地址又变了。那么这些策略是否达到了“就近”解析的目的呢？这个任务还是交给你吧，你可以用 ping 命令来测试你的 PC 到这些 IP 地址的响应时间，看看 DNS 服务器返回给你的 IP 地址是否就是最快的那个。

最后，如果你的站点在使用第三方 DNS 服务，也没有关系，有很多提供 DNS 轮询设置的服务商也都提供了智能解析服务，你可以灵活地使用它们。

故障转移

对于负载均衡调度器后端的多台实际服务器，我们完全可以通过监控系统来实时了解它们的状态，一旦发现某台服务器出现故障，这时候就需要立刻将它从调度策略中拿掉，也就是暂停指向该服务器的 DNS 记录，以免用户访问到发生故障的服务器而感到莫名其妙。

对于基于 DNS 的负载均衡系统，要做到这一点的确让人非常头疼，因为有一个现实的问题是，我们一般不会将 DNS 记录的 TTL 设置为 0，这使得所有对 DNS 记录的修改都需要一定时间才能生效，比如一个 DNS 记录的 TTL 为 3600 秒，那么对它的更新最多要过一个小时才会生效，这是我们无法容忍的，当然，用户更无法容忍。

另一方面，如何在意识到故障后的第一时间修改 DNS 记录，也是我们需要考虑的问题，在迫不得已需要容忍 DNS 记录更新延迟的情况下，我们唯一能做的就是尽早修改 DNS 记录。

听起来一点也不难，也许你为站点搭建了专用 DNS 服务器，那么你可以通过修改配置来快速完成任务，如果你是在使用第三方 DNS 服务，也没有关系，通过域名管理平台同样可以完成 DNS 修改工作。但是，这些都得依赖人力，的确，它们显得不够快速和自动化，关键的时候时间就是一切，特别是当需要和监控系统集成实现自动故障转移时，这些方法都显得力不从心。

也许你听过动态 DNS，这其实是 DNS 协议的一个特性（Standard Dynamic update DNS，DDNS，RFC2126），它允许 DNS 服务器开放特定的服务，为我们自动化远程修改 DNS 记录提供了可能。

这让我想起了现在几乎所有宽带路由器都支持的一个功能，那就是动态域名解析，你还有印象吗？当你的主机使用动态 IP 地址接入互联网，并且你希望将某个域名指向这台主机时，所谓的动态域名解析便发挥了作用，它做的事情很简单，就是在每次 IP 地址变更时及时地更新 DNS 服务器，当然，一定的延迟仍然是在所难免的，同样是因为 DNS 记录的 TTL。

利用同样的思路，当我们监测到某台实际服务器发生故障后，便可以通过动态 DNS 协议来迅速修改 DNS 记录。

一些不足

刚才我们已经提到了由于 DNS 记录的缓存带来的更新延迟，这导致我们对于调度器的控制总是跟不上节奏，这或多或少是一件令人遗憾的事情。

相比于 HTTP 重定向方式的调度器，DNS 服务器更像魔术师，它可以在用户面前很好地隐藏实际服务器，没有用户能直接看到 DNS 解析到了哪一台实际服务器，也没有人关心这个，但是与此同时，它也给服务器运维人员的调试带来了一些不便，人们需要通过修改/etc/hosts 来为域名指定某个实际服务器的 IP 地址，以跳过 DNS 服务器变幻莫测的调度。

除此之外，在这种基于 DNS 的负载均衡框架之下，负载均衡调度器工作在 DNS 层面，这导致它的调度灵活性被或多或少地削弱，策略的开发存在一定的局限性，比如你无法将 HTTP 请求的上下文引入到调度策略中，而在前面介绍的基于 HTTP 重定向的负载均衡系统中，调度器工作在 HTTP 层面，它可以在充分理解 HTTP 请求之后根据站点的应用逻辑来设计调度策略，比如根据请求的不同 URL 来进行合理的过滤和转移。

另一方面，根据实际服务器的实时负载差异来调整调度策略，这需要 DNS 服务器在每次解析操作时分析各服务器的健康状态，对于 DNS 服务器来说，这种自定义开发存在较高的门槛，更何况大多数站点只是使用第三方 DNS 服务，根本没有自主开发的可能，所以大多数人只能享受到单一的 RR 调度算法。

事实上，就算是 DNS 服务器能够做到最完美的调度，也不要忘了，DNS 记录缓存的出现将会再次成为毁灭者，各级节点的 DNS 服务器不同程度的缓存会让你晕头转向，如此庞大的体系简直是太复杂了，完全超出我们的分析能力，你得研究地理、人口、城市、交换节点等，从来没有见过如此复杂的事情，还是算了吧。

的确，DNS 服务器充当了一个粗放型的请求调度器，这给我们带来了一些遗憾，在这种情况下，如何让多台实际服务器最大程度地保持比较均衡的负载，是我们需要持续考虑的问题。

这里顺便说一下，所谓的“均衡”，不能狭义地理解为分配给所有实际服务器的工作量一样多，因为有时候，多台服务器的承载能力各不相同，这可能体现在硬件配置、网络带宽的差异，也可能因为某台服务器身兼多职，我们所说的“均衡”，也就是

希望所有服务器都不要过载，并且能够最大程度地发挥作用。

这里我们拿点真实的数据来看看，在我过去参与开发的一个站点中，同样是基于 DNS 的负载均衡，采用 RR 调度方式，有两台同样配置的实际服务器 web127 和 web129，作为站点的动态内容服务器，你可以认为这两台服务器的承载能力完全相同，而且我们希望它们能够完成同样的工作量。

我们分别来看看这两台服务器在最近 24 小时内的 WAN 网卡流量和系统负载，首先看看它们的 WAN 网卡流量图，如图 2-3 和图 2-4 所示。

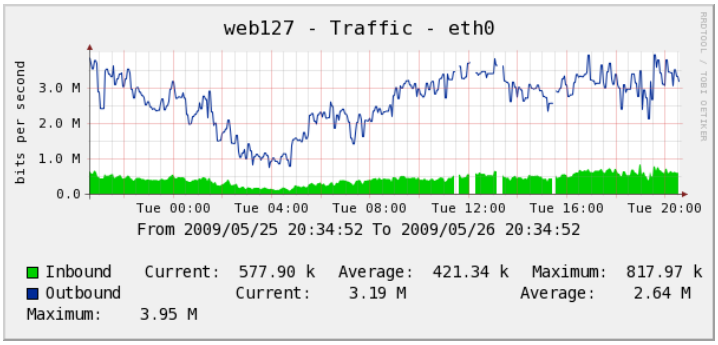


图 2-3　服务器 web127 的 WAN 网卡流量图

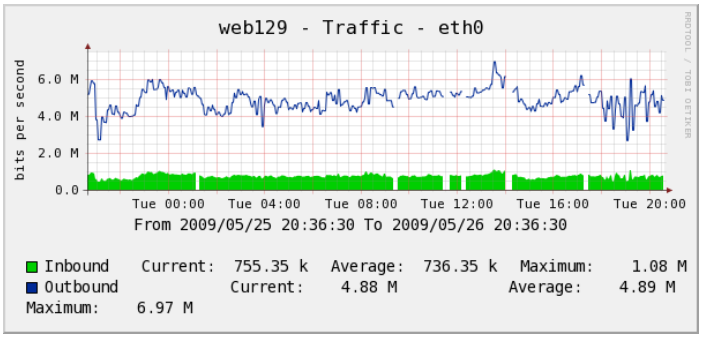


图 2-4　服务器 web129 的 WAN 网卡流量图

注意以上这两幅图的纵坐标比例是不同的，从总体上看，不论是从网卡流入还是流出的数据量，web129 都要明显大于 web127，而且两者的变化趋势几乎没有太多相似的规律。

接下来我们看两台服务器的系统负载图，如图 2-5 和图 2-6 所示。

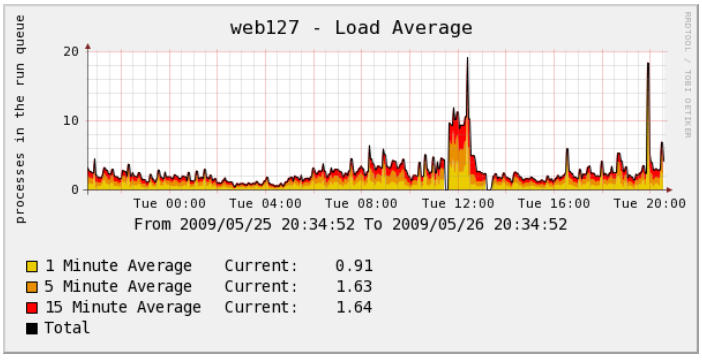


图 2-5　服务器 web127 的系统负载曲线图

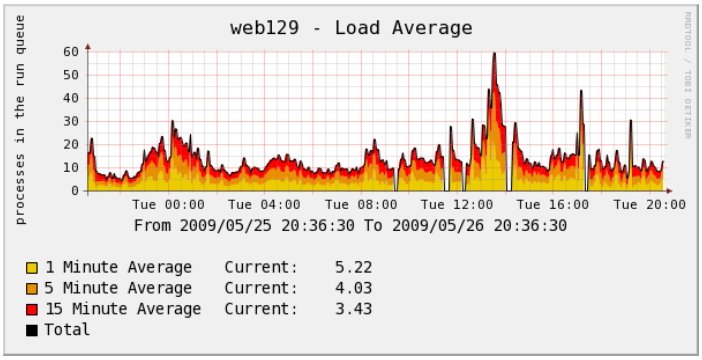


图 2-6　服务器 web129 的系统负载曲线图

显然，作为调度器的 DNS 服务器并没有很好地完成工作量均衡分配，而且差异比较大，这也在我们的意料之中，当然，这种差异的程度随着应用场景的不同会发生变化，总之，当你了解了这些内容后，是否选择基于 DNS 的负载均衡方式完全取决于你的需要。

2.4　反向代理负载均衡

我们前面曾经介绍过 Web 反向代理，的确，反向代理缓存让我为之振奋，然而，反向代理的作用不止如此，它同样可以作为调度器来实现负载均衡系统。

显然，反向代理服务器的核心工作便是转发 HTTP 请求，因此它工作在 HTTP 层面，也就是 TCP 七层结构中的应用层（第七层），所以基于反向代理的负载均衡也称为七层负载均衡，实现它并不困难，目前几乎所有主流的 Web 服务器都热衷于支持基于反向代理的负载均衡，随后我们的介绍中便会不同程度地涉及这些内容。

转移和转发

相比于前面介绍的 HTTP 重定向和 DNS 解析，作为负载均衡调度器的反向代理，对于 HTTP 请求的调度体现在“转发”上，而前者则是“转移”。

不论是转移还是转发，它们的根本目的都是相同的，那就是希望扩展系统规模，来提高承载能力，这是毋庸置疑的。

单纯地区分这些概念并没有什么意义，你需要明白的是，这种机制的改变，使得调度器完全扮演用户和实际服务器的中间人，这意味着：

- 任何对于实际服务器的 HTTP 请求都必须经过调度器；
- 调度器必须等待实际服务器的 HTTP 响应，并将它反馈给用户。

看到这里，你是否想起前面故事中的外包接口人呢？没错，接口人的工作职能也正是“转发”，因为他不希望让客户直接和外包公司接触，原因就不用我多说了吧。

正因为这样，基于反向代理的负载均衡需要我们用另一种思考方式来评判它，我们接着来看下面的内容。

按照权重分配任务

刚才我们提到，在这种全新的调度模式下，任何对于实际服务器的 HTTP 请求都必须经过调度器，这使得我们一直以来苦恼的问题终于有望解决了，那就是可以将调度策略落实到每一个 HTTP 请求，从而实现更加可控的负载均衡策略。

顺便说明一下，在基于反向代理的负载均衡系统中，我们也常把实际服务器称为后端服务器（Back-end Server）。

当不同能力的后端服务器并存的时候，调度器并不希望平均分配任务给它们，这很容易理解，的确，大锅饭时代的结束已经宣告了平均分配的愚昧。

本着能者多劳的原则，有些反向代理服务器可以非常精确地控制分配权重，这里我们用 Nginx 作为反向代理服务器，来看看权重设置对于整体吞吐率的影响。

我们准备了两台服务器作为后端，分别为 10.0.1.200 和 10.0.1.201，它们的 80 端口都运行着 Apache。同时，我们用 PHP 编写了一个动态程序，它可以模拟不同程度的计算任务，为了尽可能不依赖其他资源，我们将它设计成 CPU 密集型的程序，代码如下所示：

```
<?php
$num = $_GET['num'];
$sum = 0;
for ($i = 0; $i < $num; ++$i)
{
       $sum += $i;
}
echo $sum;
?>
```

可以看出，这个 PHP 程序可以根据 URL 参数的传递进行不同开销的计算。那么，接下来我们分别来对这两台后端服务器进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.9
Server Hostname:        10.0.1.200
Server Port:            80
Document Path:          /test.php?num=100000
Document Length:        13 bytes
Concurrency Level:      100
Time taken for tests:   5.076 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      203000 bytes
HTML transferred:       13000 bytes
Requests per second:    197.00 [#/sec] (mean)
Time per request:       507.614 [ms] (mean)
Time per request:       5.076 [ms] (mean, across all concurrent requests)
Transfer rate:          39.05 [Kbytes/sec] received

Server Software:        Apache/2.2.11
Server Hostname:        10.0.1.201
Server Port:            80
Document Path:          /test.php?num=100000
Document Length:        13 bytes
Concurrency Level:      100
Time taken for tests:   13.728 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      219564 bytes
HTML transferred:       13338 bytes
Requests per second:    72.84 [#/sec] (mean)
Time per request:       1372.829 [ms] (mean)
Time per request:       13.728 [ms] (mean, across all concurrent requests)
Transfer rate:          15.62 [Kbytes/sec] received
```

可以看到，同样是 10 万次的循环计算，两台后端服务器的吞吐率存在很大的差异，这是怎么回事呢？忘了告诉你了，这里我们故意挑选了两台配置不同的服务器，它们的配置如表 2-1 所示。

表 2-1　两台后端服务器的具体配置

后端服务器 IP 地址	CPU	内存
10.0.1.200	Intel 1.60GHz × 2	8GB
10.0.1.201	Intel 1.60GHz	4GB

所以，10.0.1.201 这台服务器的吞吐率显得逊色一些，不过没有关系，权重分配正好能够派上用场。

接下来我们在另一台 IP 地址为 10.0.1.50 的服务器上运行 Nginx，作为反向代理服务器，也就是负载均衡调度器，并为它配置两个后端服务器，如下所示：

```
upstream backend {
    server 10.0.1.200:80;
    server 10.0.1.201:80;
}
```

以上的配置只是这里提到的关键部分，更多的配置你可以查阅 Nginx 的官方文档，里面有很详细的介绍。

经过这样的配置后，Nginx 将任务平均分配给两个后端，虽然我们知道这样做不太明智，但还是希望先来看看这样做的结果。我们对调度器进行同样条件的压力测试，结果如下所示：

```
Requests per second:    144.26 [#/sec] (mean)
```

吞吐率只有 144.26reqs/s，还不如那个 197reqs/s 的后端服务器，的确，罪魁祸首就是另一台后端服务器，它拖了后腿，帮了倒忙，但是也不能怪它，调度器应该对此负责。

接下来，我们来改变权重分配，如下所示：

```
upstream backend {
    server 10.0.1.200:80 weight=2;
    server 10.0.1.201:80 weight=1;
}
```

这意味着让更强的那台服务器比另一台多干一倍的任务，它们的分配权重为 2:1，再来看看测试结果：

```
Requests per second:    224.37 [#/sec] (mean)
```

现在的吞吐率已经超过了任意一台后端服务器的单独成绩，但只是微微超出，这显得另一台后端服务器没帮上什么忙，似乎可有可无，不行，它必须证明自己的存在是有价值的。

接下来，我们将权重分配调整为 3:1，如下所示：

```
upstream backend {
    server 10.0.1.200:80 weight=3;
    server 10.0.1.201:80 weight=1;
}
```

再次进行同样的压力测试，结果如下所示：

```
Requests per second:    266.28 [#/sec] (mean)
```

可以看到，吞吐率又有明显的提升，现在已经比较接近两台后端服务器的独立成绩之和，那么，如果我们继续调整分配权重为 4:1，结果又会如何呢？你一定很想知道，下面我们进行一下调整：

```
upstream backend {
    server 10.0.1.200:80 weight=4;
    server 10.0.1.201:80 weight=1;
}
```

再来看看测试结果：

```
Requests per second:    249.18 [#/sec] (mean)
```

好，我们将这四种权重比例下的整体吞吐率绘制成曲线图，如图 2-7 所示。

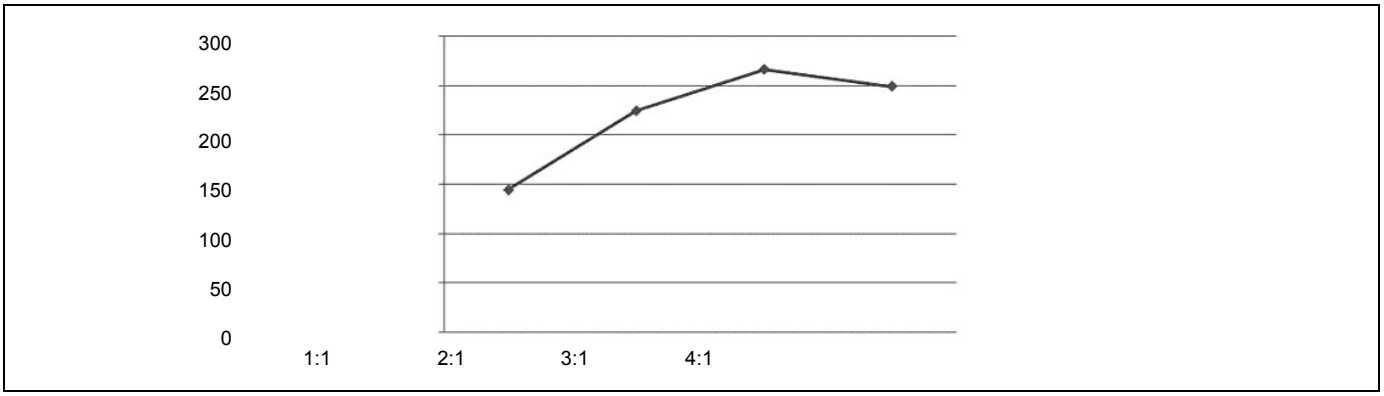


图 2-7 两台后端服务器不同权重分配比例下的吞吐率

事实告诉我们，吞吐率开始下降了，也就是说，对于以上两台后端服务器，当它们的分配权重为 3:1 的时候，整个系统可以获得最佳的性能表现。你也许已经想到了，之所以是 3:1，原因就在于两台后端服务器的独立成绩刚好近似等于 3:1，所以按照它们的能力来分配权重比例，显然可以最大程度地物尽其用。

当然，支持带有权重分配机制的 RR 调度策略，不只出现在 Nginx 中，很多其他的反向代理服务器软件也都将此视为一个必备的内置功能，但是需要强调的是，不论是哪个具体的实现，它们的权重分配机制本身都是相同的，由于这种机制和工作方式的局限，不同实现所表现出的性能几乎不会有太大差异。

这里我们不妨使用 HAProxy 来代替 Nginx，完成同样的工作，并且依次进行各种权重比例下的压力测试。HAProxy 也是一款主流的反向代理服务器，可以作为负载均衡调度器，我们对 HAProxy 进行后端配置，其中关键部分示例如下：

```
listen  proxy_1 10.0.1.50:8003
        mode http
        option httplog
        option dontlognull
        balance roundrobin
        stats uri /hastat
        server backend_1 10.0.1.200:80 weight 3
        server backend_2 10.0.1.201:80 weight 1
```

可以看到，我们将 HAProxy 监听在 8003 端口，然后为它设置了前面用到的两台后端服务器，HAProxy 也同样支持权重分配机制，接下来我们进行同样的压力测试，这里只列出各种权重分配比例下的整体吞吐率，如表 2-2 所示。

表 2-2 HAProxy 在不同权重分配比例下的吞吐率

调度权重分配	1:1	2:1	3:1	4:1
压力测试吞吐率	148.96 reqs/s	207.97 reqs/s	249.00 reqs/s	239.79 reqs/s

的确，在以上四种权重分配比例下，测试结果几乎与使用 Nginx 时不相上下。

所以，这里我们不会将太多的笔墨放在比较或者推荐具体的某个反向代理服务器软件，因为那些都是不停变化的商业产品，其中一些可能经不起时间的考验，而唯有真理和本质是相对持久的，一旦你了解了这些内容之后，至于如何选择，就像去逛超市一样，各家产品都说自己很强大，而你需要的是分析和测量。

调度器的并发处理能力

对于作为负载均衡调度器的反向代理服务器来说，它首先必须是一台经得起考验的 Web 服务器，没错，因为它工作在 HTTP 层面，它的一切工作都得从处理用户的 HTTP 请求开始。

所以，反向代理服务器本身的并发处理能力显得尤为重要，幸运的是，早在第 3 章中，我们就已经对影响服务器并发处理能力的各种因素进行了深入的探讨，你可以再次重温那些内容。

为此，请不要将反向代理服务器放到一个神秘的位置上，一旦你将它视为某种意义上的 Web 服务器，那些你曾经熟悉的概念

将再次上演，比如 I/O 模型和并发策略。

这也或多或少地影响了反向代理服务器软件的市场格局，主流的 Web 服务器都争先恐后地支持反向代理机制，比如 Apache、Lighttpd、Nginx 等，因为它们作为 Web 服务器所取得的辉煌成就让它们不费吹灰之力就可以转型成为一台马力强劲的负载均衡调度器。

扩展的制约

到现在为止，作为负载均衡调度器的反向代理服务器似乎出尽了风头，我们接下来将目光转移到另一个重要的方面，那就是这种负载均衡系统的扩展能力。理所当然，作为调度器的反向代理服务器成为我们关注的重点，因为它扮演着接口人的重要角色。

我们知道反向代理服务器工作在 HTTP 层面，对于所有 HTTP 请求都要亲自转发，可谓是大事小事亲历亲为，这也让我们为它捏了一把冷汗，你也许在怀疑它究竟有多大能耐，能支撑多少后端服务器，的确，这直接关系到整个系统的扩展能力。

接下来，我们选择了两台承载能力基本相当的后端服务器，仍然通过反向代理服务器实现负载均衡，它们的网络结构如图 2-8 所示，其中各服务器的用途和 IP 地址如表 2-3 所示。

表 2-3　反向代理负载均衡系统网络结构示意图中的服务器说明

服务器用途	内部网络 IP	外部网络 IP
反向代理服务器	10.0.1.50	125.12.12.12
后端服务器	10.0.1.210	-
后端服务器	10.0.1.201	-

然后，我们对后端服务器和调度器分别进行了一系列的压力测试，值得一提的是，这一次我们并不是在被测试的服务器上执行 ab，而是在与被测试服务器同一网段的其他服务器上执行 ab 进行压力测试，这样可以减少 ab 本身的开销对测试结果的影响，同时也是为了和随后的 IP 负载均衡测试结果保持可比性。

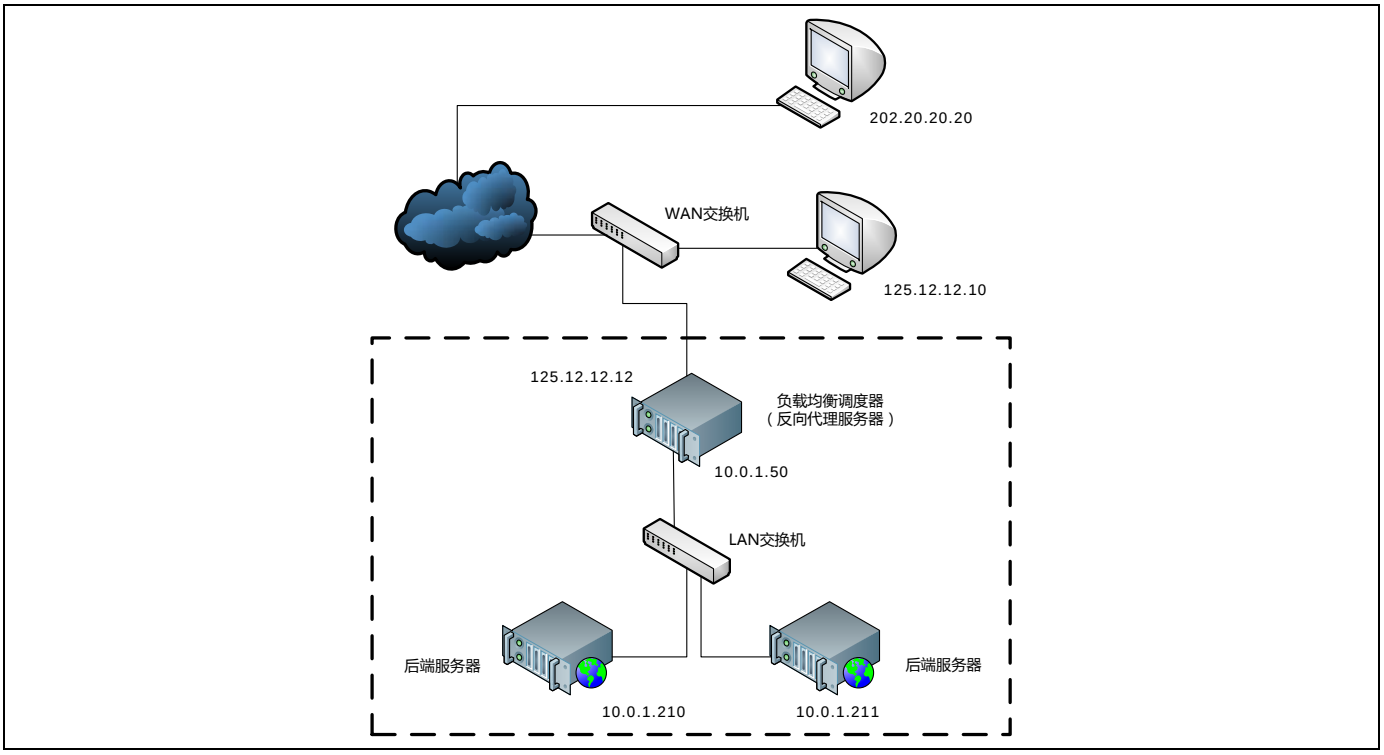


图 2-8　反向代理负载均衡系统网络结构示意图

我们来看看测试结果，如表 2-4 所示。

表 2-4　后端服务器和反向代理服务器分别对于不同内容的压力测试结果

内容类型	10.0.1.210	10.0.1.211	125.12.12.12
静态（10Bytes）	13022.53	13328.71	7778.5
动态（num=100）	10529.97	10642.43	7589.22
动态（num=500）	8244.66	8177.55	7137.96
动态（num=1000）	6950.16	6634.42	6458.53
动态（num=5000）	2771.98	2654.07	4454.4
动态（num=10000）	1444	1478.96	2468.48
动态（num=50000）	322.26	331.79	635.39
动态（num=100000）	151.9	157.48	296.6

在分析这些数据之前，我先来解释一下，在表 2-4 中，左面第一列给出了 8 种 Web 内容，它们对应着不同程度的 CPU 计算和 I/O 操作，对 Web 服务器来说这意味着处理这些内容将花费不同的开销，其中的动态内容正是前面那个可以根据指定循环次数进行计算的 PHP 程序，在这里它又派上了用场。右面的三列分别列出了两个后端服务器的独立测试结果和整个负载均衡系统的测试结果，当然，它们的单位都是 reqs/s。

为了更好地分析表格中的数据，我们绘制了一幅柱状对比图，如图 2-9 所示。

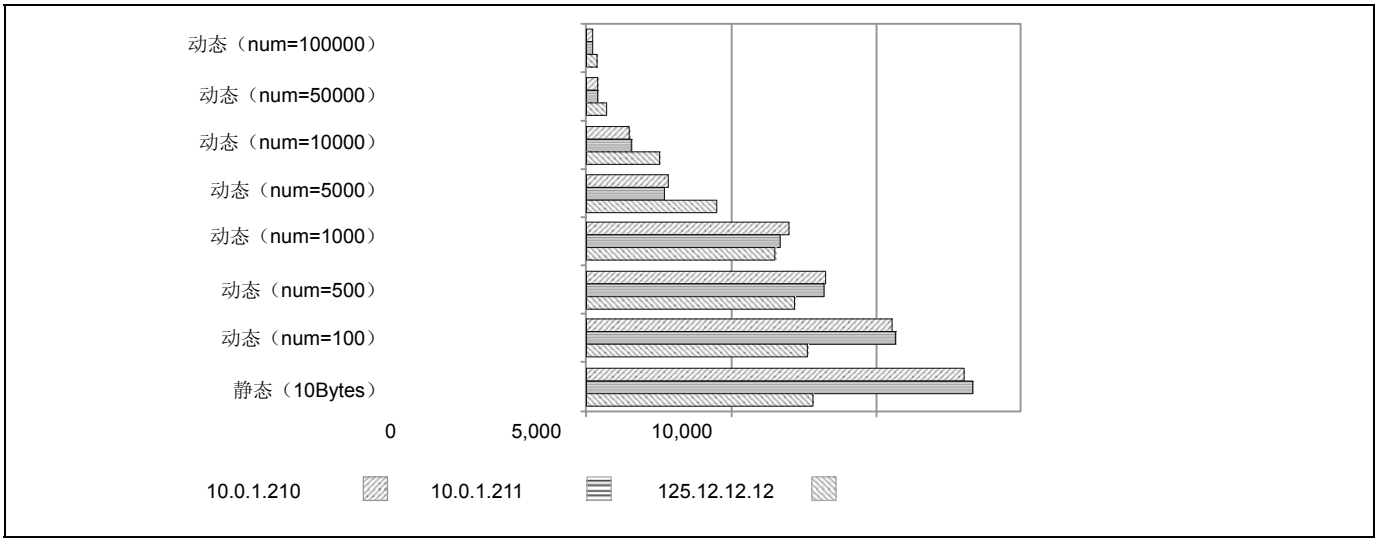


图 2-9　后端服务器和反向代理服务器分别对于不同内容的压力测试结果柱状对比图

从上往下看，内容处理的开销逐渐减少，两台后端服务器的吞吐率逐渐增加，反向代理服务器的吞吐率也随之增加，但是，我们需要注意的是，对于以上各种类型的内容，反向代理服务器的吞吐率是否为两台后端服务器的吞吐率叠加之和呢？当然，这是我们希望看到的。

从图上可以清晰地看出，当 num=100000 时，内容处理的开销最大，负载均衡系统的整体吞吐率几乎等于两台后端服务器的吞吐率之和，随后当 num 逐渐减少，整体吞吐率开始渐渐地落后于两台后端服务器的吞吐率之和，当 num=1000 的时候，整体吞吐率甚至还不如任意一个后端服务器的吞吐率高，如图 2-10 所示的对比图更加直观地反映了整体吞吐率的变化趋势。

这里我们虽然只用了两台后端服务器，但是已经充分暴露了调度器的“接口人瓶颈”，这种瓶颈效应随着后端服务器内容处理时间的减少而逐渐明显，这不难解释，反向代理服务器进行转发操作本身是需要一定开销的，比如创建线程、与后端服务器建立 TCP 连接、接收后端服务器返回的处理结果、分析 HTTP 头信息、用户空间和内核空间的频繁切换等，通常这部分时间并不长，但是当后端服务器处理请求的时间非常短时，转发的开销就显得尤为突出。

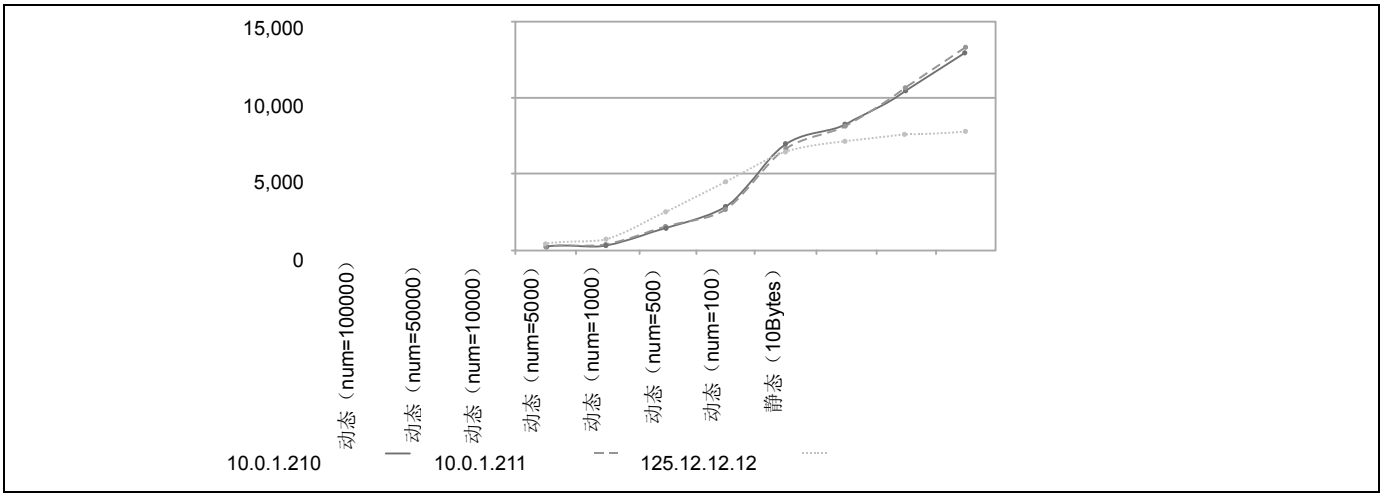


图 2-10　后端服务器和反向代理服务器分别对于不同内容的压力测试结果曲线对比图

另一方面，我们前面也提到了调度器本身的并发处理能力，这也使得当反向代理服务器的吞吐率逐渐接近极限时，无论添加多少后端服务器都将无济于事，因为调度器已经忙不过来了。

所以，你已经意识到，工作在 HTTP 层面的反向代理服务器扩展能力的制约，不仅来自于自身的对外服务能力，也归咎于其转发开销是否上升为主要时间。

为此，我们再来如图 2-9 所示的对比图，从上到下算是两个极端，从图上可以看出，最适合通过反向代理服务器来实现负载均衡的，正是位于图表上方的几种内容，它们就像从事劳动密集型工作一样，人多力量大；而位于图表底部的几种内容，反向代理服务器逐渐变得吃力，尤其是最后的静态内容，整体不但没有提升吞吐率，还浪费了多台服务器资源，对于这种情况，更适合使用前面介绍的基于 DNS 的负载均衡方式。

健康探测

在前面介绍基于 DNS 的负载均衡时，我们曾经提到用于了解实际服务器状态的监控系统，通过它，我们可以第一时间发现存在故障的服务器，然后快速动态更新 DNS 服务器。

当然，这里提到的监控系统并不是 DNS 服务器的组成部分，你可以选择一些开源项目或者商业产品，也可以根据需要自己开发，但是无论如何，你都需要付出不少的人力或者金钱。

幸运的是，一些反向代理服务器软件希望出人头地，成为更加出色的负载均衡调度器，它们将监控后端服务器的职责视为自己的神圣使命，得益于反向代理服务器的工作机制，它们可以轻松有效地监视后端服务器的任何举动，而你只需要简单配置即可。

通常来说，我们希望能够监控后端服务器的很多方面，比如系统负载、响应时间、是否可用、TCP 连接数、流量等，它们都是负载均衡调度策略需要考虑的因素。在这里，我们使用 Varnish 作为调度器，来监控后端服务器的可用性。

你一定还记得 Varnish 吧，前面我们介绍反向代理缓存的时候曾经提到过它，没错，它同时也支持负载均衡，我们来修改 Varnish 的配置文件，增加后端服务器和负载均衡策略等配置，如下所示：

```
backend b1 {
  .host = "10.0.1.201";
  .port = "80";
  .probe = {
        .url = "/probe.htm";
        .interval = 5s;
        .timeout = 1s;
        .window = 5;
        .threshold = 3;
  }
}
backend b2 {
  .host = "10.0.1.202";
```

```
    .port = "80";
    .probe = {
          .url = "/probe.htm";
          .interval = 5s;
          .timeout = 1s;
          .window = 5;
          .threshold = 3;
    }
}
director lb round-robin {
    {
        .backend = b1;
    }
    {
        .backend = b2;
    }
}
```

可以看到，我们通过 backend 关键字定义了两个后端服务器，并且用 director 关键字定义了一个名为 lb 的负载均衡调度器，同时采用了 RR 调度策略。这里需要说明一下，Varnish 目前对于 RR 调度似乎不支持分配权重的设置，所以这里我们选用了两台承载能力基本相同的后端服务器。

在 backend 的定义部分中，我们看到了.probe 设置，这代表了 Varnish 的探测器，根据这些配置，调度器将会每隔 5 秒钟请求后端服务器的/probe.htm，只有当 HTTP 响应头代码为 200 时，调度器才认为该后端服务器是可用的。

作为后端服务器上的被探测内容，这里我们使用了一个静态文件 probe.htm，你也可以使用 PHP 等动态程序，甚至在程序中包含数据库查询等操作，这样可以更加贴近真实场景，反应实际状态，但是有一点需要注意，那就是当你要报告一个不可用故障时，只需要返回一个不是 200 的状态码即可。

同时，别忘了还要在 Varnish 的 vcl_recv 回调过程中调用刚才定义的调度器，如下所示：

```
sub vcl_recv {
    if (req.request != "GET" &&
      req.request != "HEAD" &&
      req.request != "PUT" &&
      req.request != "POST" &&
      req.request != "TRACE" &&
      req.request != "OPTIONS" &&
      req.request != "DELETE") {
        return (pipe);
    }
    if (req.request != "GET" && req.request != "HEAD") {
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        return (pass);
    }
    set req.backend = lb;
    return (pass);
}
```

以上的粗体部分正是我们添加的调用语句，你一定还记得 vcl_recv 这个过程的触发条件，它是在反向代理服务器接收到用户的 HTTP 请求后被调用，所以这时候需要通过调度器来选择适当的后端服务器并转发请求。同时，最后一行的 return (pass)也是经过了我们的修改，原来的 return (lookup)意味着反向代理缓存将可能发挥作用，而这里我们并不需要反向代理缓存。

在验证探测器之前，我们先对两台后端服务器分别进行压力测试，它们的吞吐率表现如表 2-5 所示。

表 2-5　两台后端服务器的独立吞吐率

后端服务器 IP 地址	吞吐率（reqs/s）
10.0.1.201	596.29
10.0.1.202	580.41

现在我们使用 Varnish 作为调度器，它会将我们的请求轮流转发给两台后端服务器。但是，我们暂时不开启 Varnish 的探测器，来看看会发生什么结果。

接下来，关键的时刻到了，我们将其中一台后端服务器的 Web 服务关闭，这导致它将无法处理任何 HTTP 请求。然后，我们对调度器进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        10.0.1.50
Server Port:            8010
Document Path:          /test.php?num=10000
Document Length:        206 bytes
Concurrency Level:      100
Time taken for tests:   0.908 seconds
Complete requests:      1000
Failed requests:        500
   (Connect: 0, Receive: 0, Length: 500, Exceptions: 0)
Write errors:           0
Non-2xx responses:      500
Total transferred:      343500 bytes
HTML transferred:       108500 bytes
Requests per second:    1101.85 [#/sec] (mean)
Time per request:       90.756 [ms] (mean)
Time per request:       0.908 [ms] (mean, across all concurrent requests)
Transfer rate:          369.62 [Kbytes/sec] received
```

很糟糕，有一半数量的请求都失败了，虽然整体吞吐率基本接近于两个后端服务器的独立吞吐率之和，但是这又有什么意义呢？

现在我们开启刚才配置好的探测器，重新启动 Varnish，为了让探测器工作，我们等待了 5 秒钟，再次进行压力测试，结果如下所示：

```
Server Software:      lighttpd/1.4.20
Server Hostname:      10.0.1.50
Server Port:          8010
Document Path:        /test.php?num=10000
Document Length:      11 bytes
Concurrency Level:    100
Time taken for tests: 1.825 seconds
Complete requests:    1000
Failed requests:      0
Write errors:         0
Total transferred:    229000 bytes
HTML transferred:     11000 bytes
Requests per second:  547.99 [#/sec] (mean)
Time per request:     182.484 [ms] (mean)
Time per request:     1.825 [ms] (mean, across all concurrent requests)
Transfer rate:        122.55 [Kbytes/sec] received
```

可以看到，调度器已经放弃了关闭 Web 服务的那台后端服务器，即便整体吞吐率大幅度下降，但至少不会给用户返回一个错误页面。

在实际应用中，为了提高整个负载均衡系统的可用性而不影响性能，我们可以部署一定数量的备用后端服务器，这样即便是一些后端服务器出现故障后被调度器放弃，备用后端服务器也可以接替它们的工作，保证整体的性能。

粘滞会话

负载均衡调度器最大程度地让用户不必关心后端服务器，我们知道，当采用 RR 调度策略时，即便是同一用户对同一内容的多次请求，也可能被转发到了不同的后端服务器，这听起来似乎没什么大碍，但有时候，或许会带来一些问题。

- 当某台后端服务器启用了 Session 来本地化保存用户的一些数据后，下次用户的请求如果转发给了其他后端服务器，将导致之前的 Session 数据无法访问；
- 后端服务器实现了一定的动态内容缓存，而毫无规律的转发使得这些缓存的利用率下降。

如何解决这些问题呢？从表面上看，我们需要做的就是调整调度策略，让用户在一次会话周期内的所有请求始终转发到一台特定的后端服务器上，这种机制也称为粘滞会话（Sticky Sessions），要实现它的关键在于如何设计持续性调度算法。

既然要让调度器可以识别用户，那么将用户的 IP 地址作为识别标志最为合适，一些反向代理服务器对此都有支持，比如 Nginx 和 HAProxy，它们可以将用户的 IP 地址进行 Hash 计算并散列到不同的后端服务器上。

对于 Nginx，只需要在 upstream 中声明 ip_hash 即可，如下所示：

```
upstream backend {
    ip_hash;
    server 10.0.1.200:80;
    server 10.0.1.201:80;
}
```

而对于 HAProxy 的配置也非常简单，你需要在 balance 关键字后面添加 source 策略名称，同时要使用 TCP 模式，如下所示：

```
listen  proxy_1 10.0.1.50:8003
        mode tcp
        option httplog
        option dontlognull
        balance source
        stats uri /hastat
        server backend_1 10.0.1.200:80
        server backend_2 10.0.1.201:80
```

除此之外，你还可以利用 Cookies 机制来设计持久性算法，比如调度器将某个后端服务器的编号追加到写给用户的 Cookies 中，这样调度器便可以在该用户随后的请求中知道应该转发给哪台后端服务器。这样做可以更加细粒度地追踪到每一个用户，试想一下，当有很多用户隐藏在一个公开 IP 地址后面时，利用 Cookies 的持久性算法将显得更加有效。

另一方面，回到我们刚才提到的第二个问题，我们希望将对同一个 URL 的请求始终转发到同一台特定的后端服务器，以充分利用后端服务器针对该 URL 进行的本地化缓存，要实现这一点，HAProxy 也提供了支持，我们使用 uri 策略名称配置如下：

```
listen  proxy_1 10.0.1.50:8003
        mode http
        option httplog
        option dontlognull
        balance uri
        stats uri /hastat
        server backend_1 10.0.1.200:80
        server backend_2 10.0.1.201:80
```

这使得作为调度器的 HAProxy 将对请求的 URL 进行 Hash 计算，然后散列到多台后端服务器上。

好，我们已经实现了粘滞会话，但是如此一来，粘滞会话可能或多或少地破坏了均衡策略，至少像权重分配这样的动态策略已经无法工作，我们对此不能视而不见，否则前面的努力即将付诸东流。

当然，问题的关键在于，我们究竟是否要通过实现粘滞会话来迁就系统的特殊需要呢？在权衡代价之后你认为是否值得呢？最为关键的问题是前面提到的两个问题是否能从根本上避免呢？如果可以，这很值得去考虑。

事实上，在后端服务器上保存 Session 数据和本地化缓存，的确是一件不明智的事情，它使得后端服务器显得过于个性化，以至于和整个系统格格不入，如果允许的话，我们应该尽量避免这样的设计，比如采用分布式 Session 或者分布式缓存等，让后端服务器的应用尽量与本地无关，也可更好地适应环境。

2.5 IP 负载均衡

我们已经充分了解了反向代理服务器作为负载均衡调度器的工作机制，其本身的开销已经严重制约了这种框架的可扩展性，从而也限制了它的性能极限。

那么，能否在 HTTP 层面以下实现负载均衡呢？答案是肯定的，还记得本书开头那个星际铁路系统的故事吗？回忆一下网络分层模型，事实上，在数据链路层（第二层）、网络层（第三层）以及传输层（四层）都可以实现不同机制的负载均衡，但有所不同的是，这些负载均衡调度器的工作必须由 Linux 内核来完成，因为我们希望网络数据包在从内核缓冲区进入进程用户地址空间之前，尽早地被转发到其他实际服务器上，没错，Linux 内核当然可以办得到，随后我们会介绍位于内核的 Netfilter 和 IPVS，而用户空间的应用程序对此却束手无策。

另一方面，也正是因为可以将调度器工作在应用层以下，这些负载均衡系统可以支持更多的网络服务协议，比如 FTP、SMTP、DNS，以及流媒体和 VoIP 等应用。

这里我们先来介绍基于 NAT 技术的负载均衡，因为它可以工作在传输层，对数据包中的 IP 地址和端口信息进行修改，所以

也称为四层负载均衡。

DNAT

前面曾经提到过网络地址转换（Network Address Translation，NAT），它可以让用户身处内部网络却与互联网建立通信，而在这里，为了突出应用场景的差异，我想创造一个更加适合的名称，那就是“反向 NAT”，有点类似于反向代理的命名，是的，我们将实际服务器放置在内部网络，而作为网关的 NAT 服务器将来自用户端的数据包转发给内部网络的实际服务器，这为进一步实现负载均衡提供了可能。

这里说的“反向 NAT”，其实就是 DNAT，不同于 SNAT 的是，它需要修改的是数据包的目标地址和端口，这种技术在很多普通的家用宽带路由器上都有支持，如果你曾经配置过任意一款宽带路由器，或许会看到 NAT 设置，或者也称为端口映射设置，因为我们知道通过 NAT 可以修改数据包的目的地端口，很好地隐藏内网服务器的实际端口，提高安全性。

如图 2-11 所示，我们利用家用宽带路由器进行 NAT 设置，再加上前面提到的动态 DNS，完全可以在家里搭建一个公开的小型 Web 站点。

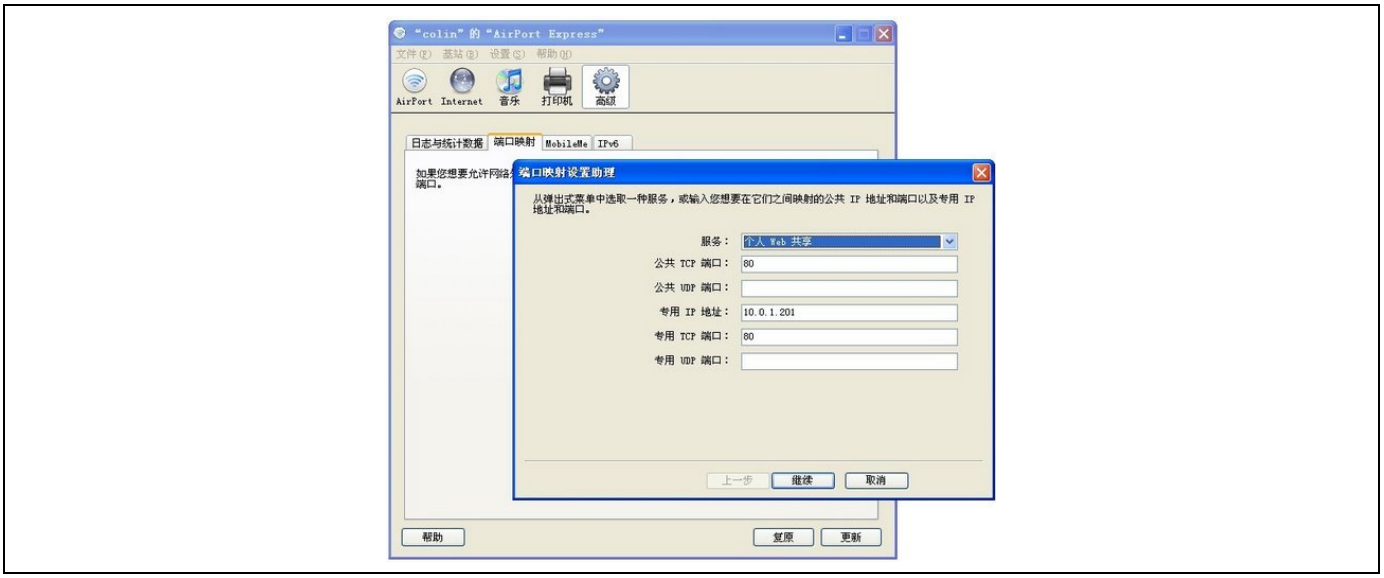


图 2-11　家用宽带路由器的端口映射设置

NAT 服务器做了什么

那么，基于 NAT 的负载均衡系统是如何工作的呢？这里我们举个简单的例子，你将会了解 NAT 服务器在整个过程中做了什么。

如图 2-12 所示，NAT 服务器拥有两块网卡，分别连接外部网络和内部网络，IP 地址分别为 125.12.12.12 和 10.0.1.50。与 NAT 服务器同在一个内部网络的是两台实际服务器，IP 地址分别为 10.0.1.210 和 10.0.1.211，它们的默认网关都是 10.0.1.50，并且都在 8000 端口上运行着 Web 服务。另外，我们假想用户端的 IP 地址为 202.20.20.20。

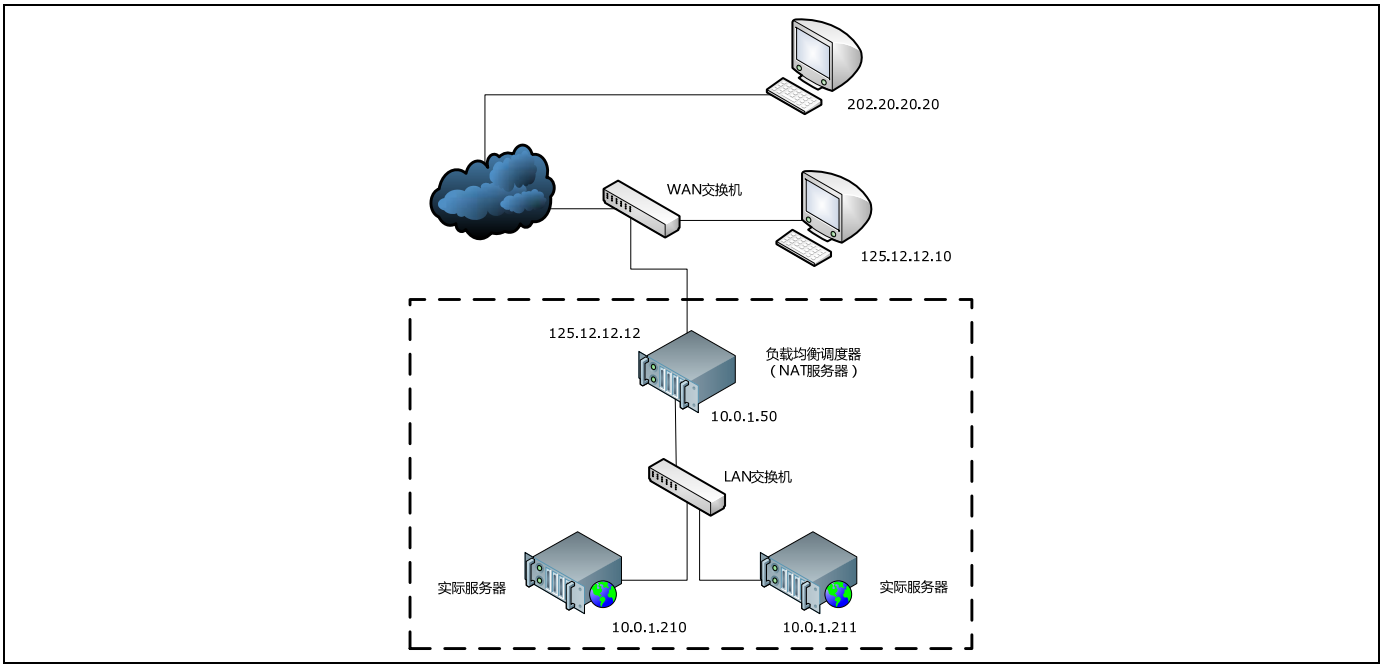


图 2-12　基于 NAT 的负载均衡系统网络结构图

表 2-6　基于 NAT 的负载均衡系统网络结构图中的服务器说明

服务器说明	内部网络 IP	外部网络 IP	默认网关
NAT 服务器	10.0.1.50	125.12.12.12	125.12.12.1
实际服务器	10.0.1.200	-	10.0.1.50
实际服务器	10.0.1.201	-	10.0.1.50

现在我们以一个数据包的真实旅程为例，来跟踪它是如何从用户端到实际服务器，又如何从实际服务器返回用户端，在这一系列过程中，数据包的命运发生了多次改变。

首先，用户端通过 DNS 服务器得知站点的 IP 地址为 125.12.12.12，当然，如果站点不使用域名的话，这一步也可能省略。

接下来，用户端向 125.12.12.12 发送了 IP 数据包，我们将目光放在其中的一个数据包上，它的来源地址和目标地址如下所示：

```
来源地址：  202.20.20.20:6584
目标地址：  125.12.12.12:80
```

当数据包到达 125.12.12.12 的内核缓冲区后，NAT 服务器并没有把它交给用户空间的进程去处理，而是挑选了一台实际服务器，这里恰好为 10.0.1.210，接下来 NAT 服务器将刚刚收到的数据包进行了修改，修改后的来源地址和目标地址如下所示：

```
来源地址：  202.20.20.20:6584
目标地址：  10.0.1.210:8000
```

可以看到，NAT 服务器修改了数据包的目标地址和端口号，紧接着，NAT 服务器指定内部网卡将这个数据包原封不动地投递到内部网络中，根据 IP 层的寻址机制，数据包自然会流入 10.0.1.210 这台实际服务器。

接下来，运行在 10.0.1.210 上的 Web 服务处理了这个数据包，然后将要包含结果数据的响应数据包投递到内部网络，它的来源地址和目标地址如下所示：

```
来源地址：  10.0.1.210:8000
目标地址：  202.20.20.20:6584
```

显然，这个数据包必须先到达默认网关，于是它再次回到了 NAT 服务器，这时候 NAT 服务器再次修改数据包，修改后的来源地址和目标地址如下所示：

```
来源地址：  125.12.12.12:80
```

目标地址： 202.20.20.20:6584

最后，数据包终于回到了用户端。

在整个过程中，NAT 服务器的动作可谓天衣无缝，它欺骗了实际服务器，而在实际服务器上运行的进程总是天真地以为数据包是用户端直接发给自己的，不过，这也许是善意的欺骗。

同时，NAT 服务器也扮演了负载均衡调度器的角色，那么，在讨论调度策略之前，你也许更关心的问题是如何实现 NAT 服务器。

Netfilter/iptables

首先，我们必须得知道 Linux 如何修改 IP 数据包，可以肯定的是，Linux 内核已经具备这样的能力，从 Linux 2.4 内核开始，其内置的 Netfilter 模块便肩负起这样的使命，它在内核中维护着一些数据包过滤表，这些表包含了用于控制数据包过滤的规则。

我们知道，当网络数据包到达服务器的网卡并且进入某个进程的地址空间之前，先要通过内核缓冲区，这时候内核中的 Netfilter 便对数据包有着绝对控制权，它可以修改数据包，改变路由规则。

既然 Netfilter 工作在内核中，我们看不见摸不着，那么如何让它按照我们的需要来工作呢？Linux 提供了 iptables，它是工作在用户空间的一个命令行工具，我们可以通过它来对 Netfilter 的过滤表进行插入、修改或删除等操作，也就是建立了与 Netfilter 沟通的桥梁，告诉它我们的意图。

那么，我们要做的就是让 Linux 服务器成为连接外部网络和私有网络的路由器，但这不是普通的路由器，我们知道路由器的工作是存储转发，除了修改数据包的 MAC 地址以外，通常它不会对数据包做其他手脚，而我们要实现的路由器恰恰是要对数据包进行必要的修改，包括来源地址和端口，或者目标地址和端口。

总之，Linux 内核改变数据包命运的惊人能力，决定了我们可以构建强大的负载均衡调度器，将请求分散到其他实际服务器上。

用 iptables 来实现调度器

可以用 iptables 来实现负载均衡调度器吗？我们来试试吧。

如果你对 iptables 的使用并不熟悉，可以通过丰富的在线文档进行系统的学习，遗憾的是，我们这里不会介绍 iptables 的详细使用规则，事实上要想在有限的篇幅中把它说清楚并不容易，而且还可能会给你留下阴影。

说到 iptables，最多的应用场景就是防火墙了，我几乎为每台 Linux 服务器都毫不犹豫地进行 iptables 防火墙配置，比如以下这段简单的 iptables 规则：

```
iptables -F INPUT
iptables -A INPUT -i eth0 -p tcp --dport 80 -j ACCEPT
iptables -P INPUT DROP
```

它完成了重要的任务，那就是告诉内核只允许外部网络通过 TCP 与这台服务器的 80 端口建立连接，这项规则可以很好地用在 Web 服务器上。

另外，我们还会用 iptables 来实现本机端口重定向，比如以下的规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8000
```

它将所有从外部网络进入服务器 80 端口的请求转移到了 8000 端口，这有什么意义呢？当然是隐藏某些服务的实际端口，同时也便于将一个端口快速切换到其他端口的服务上，提高端口管理的灵活性。

关键的时刻到了，我们将用 iptables 来实现 NAT，在此之前，我们需要执行以下的命令行操作：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

这意味着该服务器将被允许转发数据包，这也为它成为 NAT 服务器提供了可能。

接下来，我们在作为调度器的服务器上执行以下的 iptables 规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8001 -j DNAT --to-destination 10.0.1.210:8000
```

这条规则几乎完成了所有 DNAT 的实现，它将调度器外网网卡上 8001 端口接收的所有请求转发给 10.0.1.210 这台服务器的 8000 端口，至于转发的具体过程，前面我们已经详细介绍过，在此不再赘述。

同样，我们在调度器上执行以下的 iptables 规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8002 -j DNAT --to-destination 10.0.1.211:8000
```

不用多说，这条规则你一定明白了。

这两条规则已经进入了 Netfilter 的过滤表，我们可以通过 iptables 命令来查看它们，如下所示：

```
s-director:~ # iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
Target  prot opt source   destination
DNAT    tcp  --  0.0.0.0/0  0.0.0.0/0 tcp dpt:8001 to:10.0.1.210:8000
DNAT    tcp  --  0.0.0.0/0  0.0.0.0/0 tcp dpt:8002 to:10.0.1.211:8000
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

可以看到，转发规则已经存在，但是还缺少一个环节，这至关重要，我们必须将实际服务器的默认网关设置为 NAT 服务器，也就是说，NAT 服务器必须为实际服务器的网关，否则，数据包被转发后将一去不返。

添加默认网关非常容易，在实际服务器上执行以下命令行操作：

```
route add default gw 10.0.1.50
```

现在，查看路由表，刚才的默认网关已经出现了：

```
s-rs:~ # route
Kernel IP routing table
Destination  Gateway    Genmask       Flags Metric  Ref   Use Iface
10.0.1.0     *          255.255.255.0 U     0       0     0   eth1
125.12.12.0  *          255.255.255.0 U     0       0     0   eth0
link-local   *          255.255.0.0   U     0       0     0   eth0
loopback     *          255.0.0.0     U     0       0     0   lo
default      10.0.1.50  0.0.0.0       UG    0       0     0   eth0
```

现在，我们终于可以通过调度器的 8001 和 8002 端口分别将请求转发到两个实际服务器上，可是，回想前面的问题，用 iptables 来实现负载均衡调度器，看起来有点困难，iptables 似乎只能按照我们的规则来干活，没有调度器应该具备的调度能力和调度策略。

接下来，IPVS 上场的时刻到了。

IPVS/ipvsadm

熟悉了 Netfilter/iptables 的机制后，理解 IPVS（IP Virtual Server）就一点也不难了，它的工作性质类似于 Netfilter 模块，也工作在 Linux 内核中，但是它更专注于实现 IP 负载均衡。

IPVS 不仅可以实现基于 NAT 的负载均衡，同时还包括后面要介绍的直接路由和 IP 隧道等负载均衡。令人振奋的是，IPVS 模块已经内置到 Linux 2.6.x 内核中，这意味着使用 Linux 2.6.x 内核的服务器将无须重新编译内核就可以直接使用它。

要想知道内核中是否已经安装 IPVS 模块，可以进行以下查看：

```
s-mat:~ # modprobe -l | grep ipvs
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_wrr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_wlc.ko
```

```
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_sh.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_sed.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_rr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_nq.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lc.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lblcr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lblc.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_dh.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_ftp.ko
```

这样的结果就意味着 IPVS 已经安装在内核中。

当然，IPVS 也需要管理工具，那就是 ipvsadm，它为我们提供了基于命令行的配置界面，你可以通过它快速实现负载均衡系统，这里也称为 LVS（Linux Virtual Server，Linux 虚拟服务器）或者集群。

你可以从 LVS 的官方站点下载与内核匹配的 ipvsadm 源代码，然后编译它，需要注意的是，在编译过程中，你还需要在 /usr/src/linux 中存有内核源代码（Kernel-source）。

接下来，我们将使用 ipvsadm 来实现基于 NAT 的负载均衡系统，也就是 LVS-NAT。

LVS-NAT

当一切都准备好后，你会发现使用 ipvsadm 组建基于 NAT 的负载均衡系统是一件富有乐趣的事情。

现在我们可以暂时完全忘记 iptables，因为 ipvsadm 在这里可以完全取代它。首先不要忘了打开调度器的数据包转发选项，如下所示：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

然后，你还需要检查实际服务器是否已经将 NAT 服务器作为自己的默认网关，如果不是，赶快添加它，比如：

```
route add default gw 10.0.1.50
```

接下来，就是见证 IPVS 的时刻了，我们执行以下命令行操作：

```
ipvsadm -A -t 125.12.12.12:80 -s rr
ipvsadm -a -t 125.12.12.12:80 -r 10.0.1.210:8000 -m
ipvsadm -a -t 125.12.12.12:80 -r 10.0.1.211:8000 -m
```

第一行规则用来添加一台虚拟服务器，也就是负载均衡调度器，这里的-s rr 是指采用简单轮询的 RR 调度策略。后面两行用来为调度器添加实际服务器，其中-m 表示采用 NAT 方式来转发数据包，这正是我们所希望的。

接下来，我们还可以通过 ipvsadm 来查看所有实际服务器的状态，如下所示：

```
s-director:~ # ipvsadm -L -n
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  125.12.12.12:80 rr
  -> 10.0.1.210:80                Masq    1      0          2
  -> 10.0.1.211:80                Masq    1      0          2
```

大功告成，我们对调度器进行连续的 HTTP 请求，没错，它正是采用 RR 方式将请求均衡地分散到了两台实际服务器上。

关键的问题来了，我们费了很大的力气，终于搭建起基于 NAT 的负载均衡系统，它的性能如何呢？

性能

还记得前面对反向代理负载均衡系统的一系列不同内容的压力测试吗？在使用了 LVS-NAT 后，我们同样针对这些内容，再次对调度器进行压力测试，同时和之前的数据进行比较，如表 2-7 所示。

表 2-7　后端服务器、反向代理服务器、NAT 服务器分别对于不同内容的压力测试结果

内容类型	10.0.1.210	10.0.1.211	反向代理负载均衡	LVS-NAT
静态（10Bytes）	13022.53	13328.71	7778.5	15953.03
动态（num=100）	10529.97	10642.43	7589.22	13850.88
动态（num=500）	8244.66	8177.55	7137.96	13513.79
动态（num=1000）	6950.16	6634.42	6458.53	12941.28
动态（num=5000）	2771.98	2654.07	4454.4	5310.58
动态（num=10000）	1444	1478.96	2468.48	2870.92
动态（num=50000）	322.26	331.79	635.39	646.03
动态（num=100000）	151.9	157.48	296.6	303.68

对于以上的数据，我们仍然绘制了柱状图和曲线图，如图 2-13 和图 2-14 所示。通过前面对基于反向代理的负载均衡系统扩展能力的分析，我们知道，当实际服务器的吞吐率达到一定高度时，反向代理服务器的吞吐率将很快达到极限，而从以下对比图中可以看出，在基于 NAT 的负载均衡系统中，作为调度器的 NAT 服务器可以将吞吐率继续提升到一个新的高度，几乎是反向代理服务器的两倍以上，这当然归功于在内核中进行请求转发的较低开销。

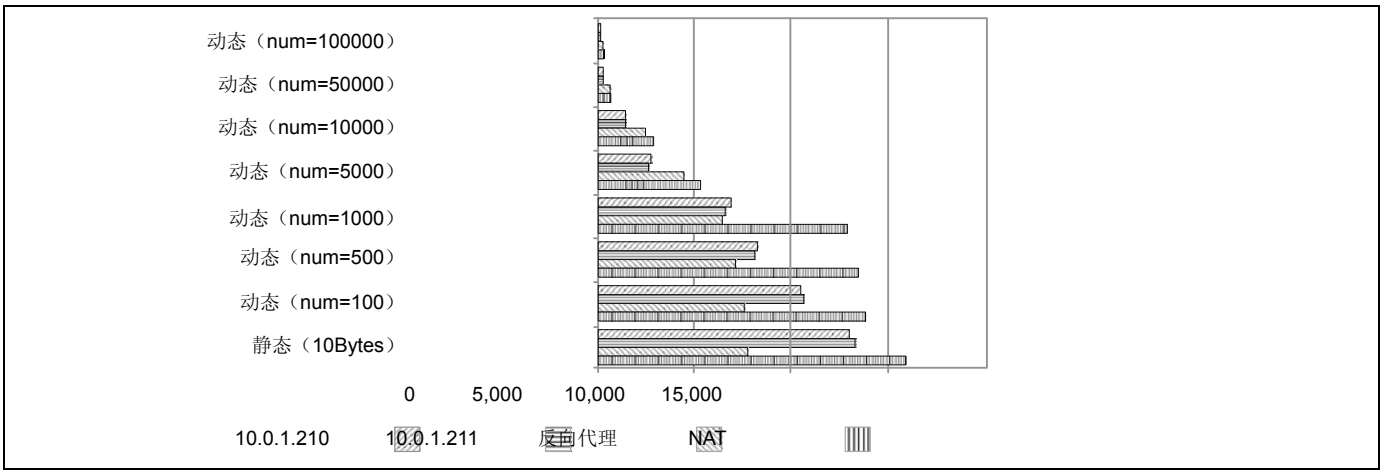


图 2-13　反向代理负载均衡和 LVS-NAT 对比柱状图

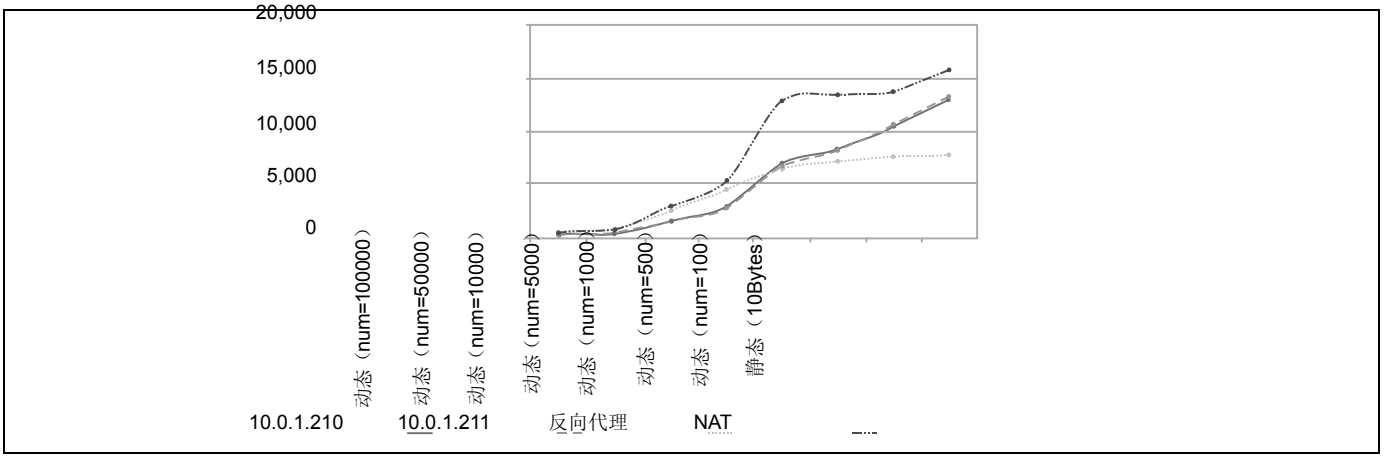


图 2-14　反向代理负载均衡和 LVS-NAT 对比曲线图

另外，从图 2-14 中可以直观地看出，对于 num 大于 5000 的动态内容，也就是实际服务器的吞吐率小于 3000reqs/s 时，不论是基于反向代理，还是基于 NAT，负载均衡的整体吞吐率都差距不大，这意味着对于一些开销较大的内容，使用简单的反向代理来搭建负载均衡系统是非常值得考虑的，至少在初期是一个快速有效的方案，而且它可以非常容易地迁移到 NAT 方式。

动态调度策略

在刚才的 LVS-NAT 中，我们使用了 RR 调度策略，它是一种静态调度策略，当在集群中实际服务器承载能力相当的环境下，它可以很好地实现均衡调度。同时，LVS 也支持带权重的 RR 调度，只需要配置实际服务器的 weight 值即可，当然，这也属于静态调度策略。

除此之外，LVS 提供了一系列的动态调度策略，比如最小连接（LC）、带权重的最小连接（WLC）、最短期望时间延迟（SED）等，它们都可以根据实际服务器的各种实时状态做出调度决策，可谓是察言观色，全面兼顾。

由于篇幅限制，我们不打算对这些动态调度策略进行详细的介绍和测试，另一方面，不同的动态调度策略适合于不同的场景，由于测试环境的局限，我们也无法全都模拟这些场景。但是，你可以通过 LVS 官方文档详细了解这些动态调度策略的机制，一旦了解后，你可以根据站点的需要和部署环境，选择合适的动态调度策略并充分测试性能。

网关瓶颈

尽管如此，作为 NAT 服务器的网关也成为制约集群扩展的瓶颈，我们知道，NAT 服务器不仅要将用户的请求转发给实际服务器，同时还要将来自实际服务器的响应转发给用户，所以，当实际服务器数量较多，并且响应数据流量较大时，来自多个实际服务器的响应数据包将有可能在 NAT 服务器发生拥挤。

显然，考验 NAT 服务器转发能力的时刻到了，由于转发数据包工作在内核中，我们几乎可以不考虑额外的开销，所以，转发能力主要取决于 NAT 服务器的网络带宽，包括内部网络和外部网络。

举个例子，假如 NAT 服务器通过 100Mbps 的交换机与多台实际服务器组成内部网络，通过前面介绍带宽的章节，我们知道这些实际服务器到 NAT 服务器的带宽为共享 100Mbps，这样一来，尽管实际服务器本身可以很容易达到 100Mbps 的响应流量，比如提供下载服务等，但是 NAT 服务器的 100Mbps 出口带宽成了制约条件，使得无论添加多少台实际服务器，整个集群最多只能提供 100Mbps 的服务。

要解决网关带宽的瓶颈也并不困难，我们可以为 NAT 服务器使用千兆网卡，并且为内部网络使用千兆交换机，图 2-15 展示了理想的带宽配置，其中提及的网卡和交换设备都是工作在全双工模式下的，也就是两个方向的数据传输都具备同样的带宽。

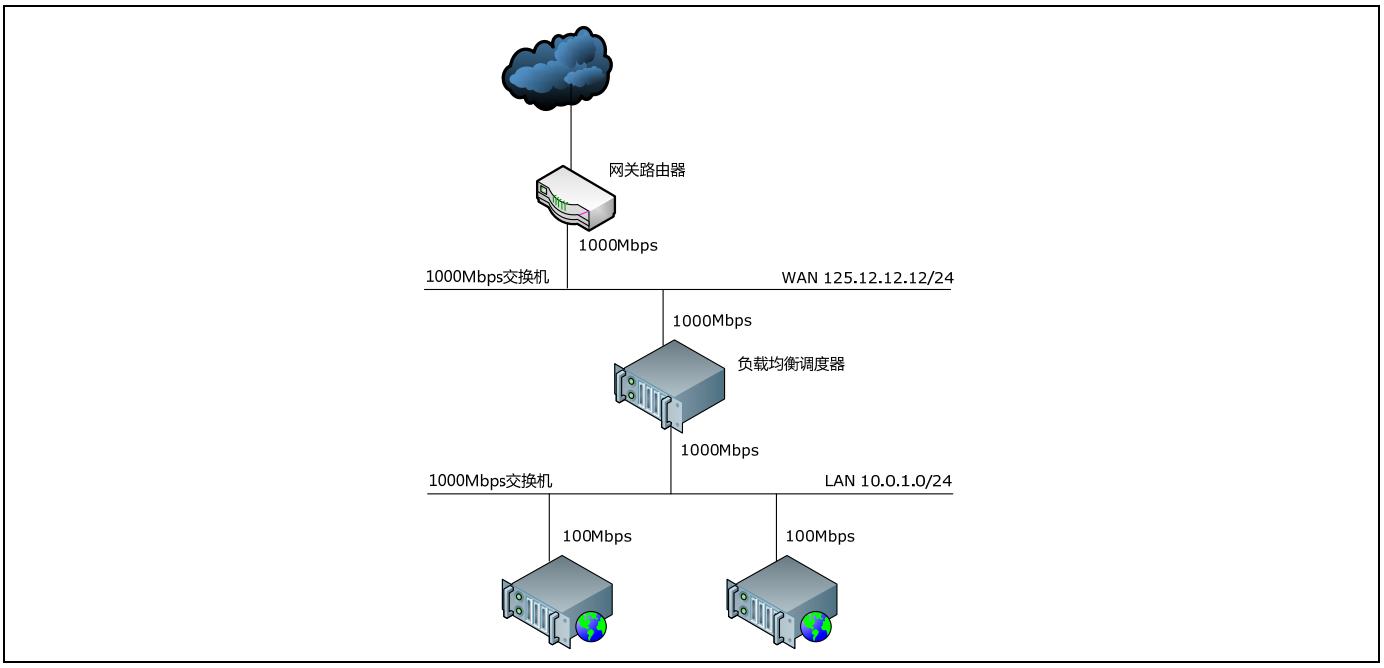


图 2-15　LVS-NAT 的理想带宽配置

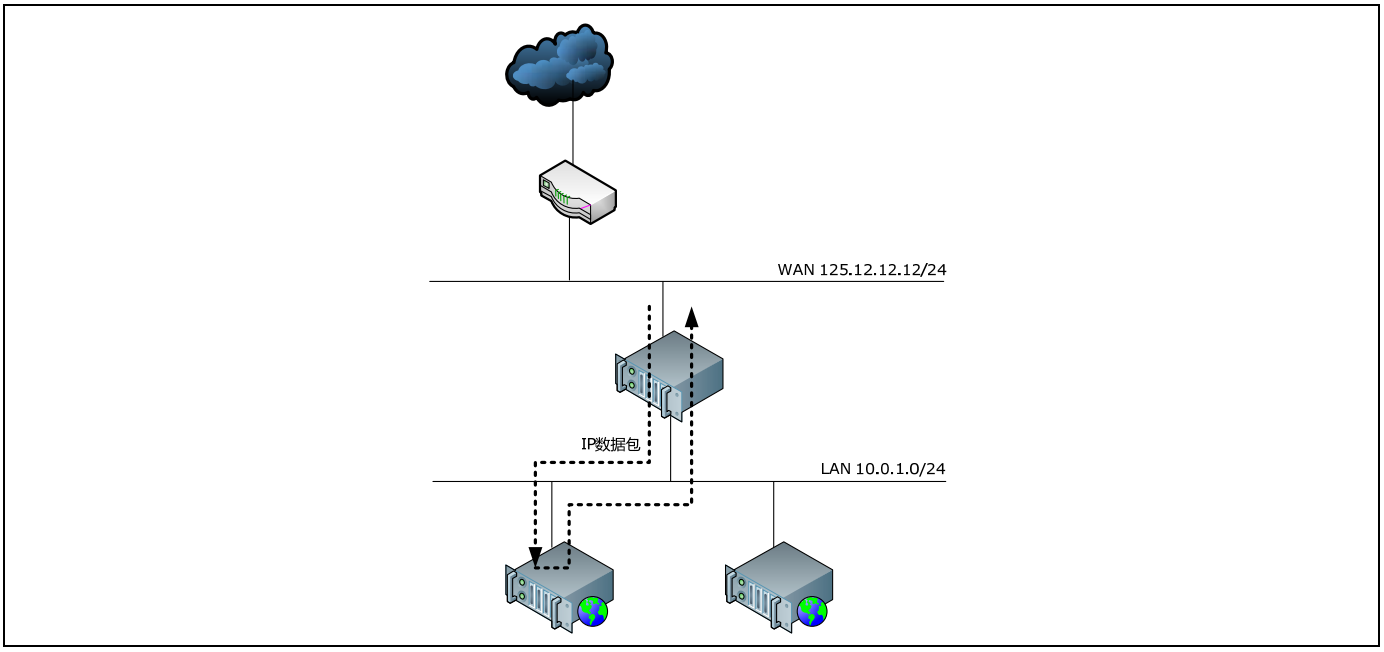


图 2-16　LVS-NAT 的流量走向图

这样做的出发点没错，但是，如果我们希望继续扩展带宽，比如实际服务器使用千兆网卡，那么 NAT 服务器将使用万兆（10Gb）网卡以及万兆交换机，可是，这些设备的昂贵价格足可以让你再购买几台服务器，至少目前是这样。

即便是为 NAT 服务器使用万兆网卡，充分满足网络带宽，但是服务器本身能够以这样的速度转发数据包吗？我们得看服务器的总线带宽，这取决于总线频率和总线位宽，比如对于支持 400MHz 前端总线频率的 32 位处理器，理论上它的总线带宽为：

```
32bit × 400MHz = 12.8Gbps = 1.6GB/s
```

这意味着 CPU 和内存直接交换数据的速度，当网络数据包通过 10Gbps 的带宽进入内核缓冲区后，理论上可以快速地执行转发，也就是将缓冲区中的数据包进行简单修改后复制到另一块负责发送数据的缓冲区。当然，实际的转发速度肯定是达不到 12.8Gbps 的，还要考虑处理时间等其他因素。

当然，如果条件允许，你也可以购买专业级的负载均衡硬件产品，比如 Cisco 的 LocalDirector 或 F5 的 BIG-IP Local Traffic Manager，它们都可以实现基于 NAT 的负载均衡，并且支持丰富的调度策略，它们可以达到较高级别的带宽，而且拥有强大的管理功能，但价格不菲。

我们不打算详细介绍这些硬件产品，如果有兴趣，你可以查阅它们的官方介绍。但可以肯定的是，它们实现数据转发和调度策略的方式和我们这里的介绍没有太大的差异。

假如你不想花钱去购买千兆交换机或万兆交换机，甚至负载均衡硬件设备，又不想让 NAT 服务器成为制约扩展的瓶颈，怎么办呢？

一个简单有效的办法是，将基于 NAT 的集群和前面的 DNS-RR 混合使用，你可以组建多个条件允许的 NAT 集群，比如 5 个 100Mbps 出口带宽的集群，然后通过 DNS-RR 方式来将用户请求均衡地指向这些集群，同时，你还可以利用 DNS 智能解析实现地域就近访问。

事实上，对于大多数中等规模的站点，拥有 5 个 100Mbps 的 NAT 集群，再加上 DNS-RR，通常足够应付全部的业务。

但是，对于提供下载或视频等服务的大规模站点，100Mbps 的集群带宽就显得微不足道了，其中一台实际服务器甚至就要吞没上百兆的带宽，现在，NAT 服务器的瓶颈出现了，你会选择提高 NAT 服务器的带宽，还是选择其他方案呢？幸运的是，LVS 提供了另一种实现负载均衡的方式，那就是直接路由（Direct Route，DR），接下来我们会详细介绍它。

2.6　直接路由

不同于 NAT 机制，直接路由方式下的负载均衡调度器工作在数据链路层（第二层），简单地说，它通过修改数据包的目标 MAC

地址，将数据包转发到实际服务器上，并且最重要的是，实际服务器的响应数据包将直接发送给用户端，而不经过调度器。

这听起来似乎不可思议，响应数据包可以不经过调度器，这意味着什么呢？

可以肯定的是，实际服务器必须直接接入外部网络，它可以不使用 RFC1918 规定的私有地址，也不能将调度器作为默认网关。

提示：

RFC1918 规定的私有 IP 地址范围是：

```
10.0.0.0      -   10.255.255.255  (10/8 prefix)
172.16.0.0    -   172.31.255.255  (172.16/12 prefix)
192.168.0.0   -   192.168.255.255 (192.168/16 prefix)
```

我们还是先来看看这种方式的实现机制，在此之前，你需要了解一个也许对你来说比较陌生的名词，那就是 IP 别名，它对于直接路由负载均衡的实现至关重要。

使用 IP 别名

我们知道，一个网络接口理所当然地拥有一个 IP 地址，但是除此之外，我们还可以为它配置更多个 IP 地址，它们称为 IP 别名。这里的网络接口可以是物理网卡（如 eth0、eth1），也可以是虚拟接口（如回环网络接口 lo）。根据规定，一个网络接口最多可以设置 256 个 IP 别名，没错，你可以把一个 C 类网段的所有 IP 地址都设置到一个网卡上，理论上没有任何问题。

你也许已经张大了嘴巴，一个网卡竟然可以设置多个 IP 地址，并且拥有同样的 MAC 地址，没错，它们可以很好地工作。在 Linux 中配置 IP 别名非常简单，比如我们在 125.12.12.12 这台服务器上执行以下命令行操作：

```
ifconfig eth0:0 125.12.12.77
```

这时候，我们通过 ifconfig 命令可以查看到刚才配置的 IP 别名，如下所示：

```
s-director:~ # ifconfig
eth0      Link encap:Ethernet   HWaddr 00:19:B9:DF:B0:52
          inet addr:125.12.12.12  Bcast:125.12.12.255  Mask:255.255. 255.0
          inet6 addr: fe80::219:b9ff:fedf:b052/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6428227 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8242159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:847758687 (808.4 Mb)  TX bytes:4007763688 (3822.1 Mb)
          Interrupt:6 Memory:f4000000-f4011100
eth0:0     Link encap:Ethernet  HWaddr 00:19:B9:DF:B0:52
          inet addr:125.12.12.77  Bcast:125.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:6 Memory:f4000000-f4011100
```

这样一来，我们从同网段的另一台服务器尝试访问它，如下所示：

```
s-db:~ # ping 125.12.12.77
PING 125.12.12.77 (125.12.12.77) 56(84) bytes of data.
64 bytes from 125.12.12.77: icmp_seq=1 ttl=64 time=6.35 ms
64 bytes from 125.12.12.77: icmp_seq=2 ttl=64 time=0.123 ms
64 bytes from 125.12.12.77: icmp_seq=3 ttl=64 time=0.119 ms
```

可见，IP 别名对外界来说，和普通 IP 地址没什么区别，当网络中有 ARP 广播询问谁拥有 125.12.12.77 这个 IP 地址时，这台服务器将会积极应答。我们在另一台服务器上查看 ARP 表，其中包含以下两条：

```
s-db:~ # arp
Address         HWtype  HWaddress           Flags Mask      Iface
125.12.12.12    ether   00:19:B9:DF:B0:52   C               eth0
125.12.12.77    ether   00:19:B9:DF:B0:52   C               eth0
```

的确，两个 IP 地址对应着相同的 MAC 地址。

将实际服务器接入外部网络

现在你应该已经知道 IP 别名是怎么回事了，那么，它有什么用呢？

刚才我们说到，调度器通过修改数据包的目标 MAC 地址，将它转发给实际服务器，注意，它并没有修改目标 IP 地址，那么一旦数据包到了实际服务器后，发现实际服务器的 IP 地址并不是数据包的目标 IP 时，你也许无法想象会发生什么事，我想这大概就跟梦游的人醒来时的感觉一样。

没错，我们要做的，就是给实际服务器添加和调度器 IP 地址相同的 IP 别名，这样才可以让转发到实际服务器的数据包找到归属感。

在此之前，我们先将前面的网络结构进行一番调整，如图 2-17 所示，其中各台服务器的 IP 地址和网关如表 2-8 所示。

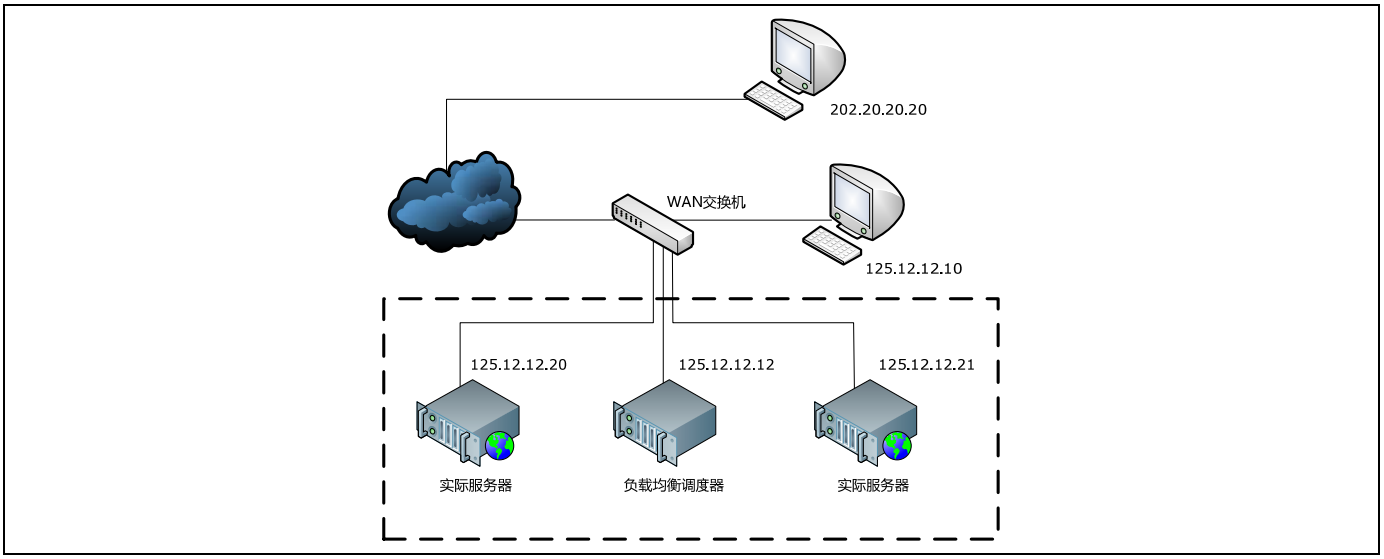


图 2-17 _LVS-DR 负载均衡系统网络结构示意图

表 2-8 _LVS-DR 负载均衡系统网络结构示意图中的服务器说明

服务器说明	外部网络 IP	默认网关	IP 别名
负载均衡调度器	125.12.12.12	125.12.12.1	125.12.12.77
实际服务器	125.12.12.20	125.12.12.1	125.12.12.77
实际服务器	125.12.12.21	125.12.12.1	125.12.12.77

可以看到，我们为两台实际服务器配置了外部网络 IP 地址，这些 IP 地址通常需要你从 IDC 那里购买。同时，我们为调度器也配置了 IP 别名，这将为调度器的故障转移提供便利，后面我们会介绍调度器的可用性。

这样一来，我们将通过 125.12.12.77 这个 IP 别名来访问调度器，你可以将站点的域名指向这个 IP 别名。

接下来，我们为实际服务器配置与调度器相同的 IP 别名，这里我们将 IP 别名添加到回环接口 lo 上，并且设置路由规则，让实际服务器不要去寻找其他拥有这个 IP 别名的服务器，命令行操作如下：

```
ifconfig lo:0 125.12.12.77 broadcast 125.12.12.77 netmask 255.255.255. 255 up
route add -host 125.12.12.77 dev lo:0
```

现在，我们通过 ifconfig 命令来查看网络接口，可以看到以下结果：

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:111299279 errors:0 dropped:0 overruns:0 frame:0
          TX packets:111299279 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

```
          RX bytes:3748942531 (3575.2 Mb)  TX bytes:3748942531 (3575.2Mb)
lo:0      Link encap:Local Loopback
          inet addr:125.12.12.77  Mask:255.255.255.255
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
```

另外，还要防止实际服务器响应来自网络中针对 IP 别名的 ARP 广播，为此，执行以下操作即可：

```
echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore
echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce
echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore
echo "2" > /proc/sys/net/ipv4/conf/all/arp_announce
```

好，准备工作一切就绪后，我们就可以使用 ipvsadm 命令来配置 LVS-DR 集群了。

LVS-DR

接下来，我们在作为调度器的服务器上通过 ipvsadm 命令进行以下配置：

```
ipvsadm -A -t 125.12.12.77:80 -s rr
ipvsadm -a -t 125.12.12.77:80 -r 125.12.12.20:80 -g
ipvsadm -a -t 125.12.12.77:80 -r 125.12.12.21:80 -g
```

有了配置 LVS-NAT 的经验后，你应该已经非常熟悉上述配置规则，这里我们仍然使用了 RR 调度策略，有所不同的是，在添加实际服务器的时候，我们使用了-g 选项，这意味着告诉调度器使用直接路由的方式转发数据包。

这时候我们通过 ipvsadm 命令可以看到以下的状态：

```
s-director:~ # ipvsadm -L -n
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  125.12.12.77:80 rr
  -> 125.12.12.20:80              Route   1      0          0
  -> 125.12.12.21:80              Route   1      0          0
```

需要注意的是，在使用直接路由进行转发的情况下，我们无法修改数据包的目的端口，原因很简单，我们知道这种转发机制工作在数据链路层，所以对于上层的端口信息，它无能为力。

也许你已经感觉到，一旦熟悉了 LVS-NAT 的配置方式后，就可以很容易地将其改进为直接路由方式。

与 LVS-NAT 的性能比较

你最关心的时刻到了，我们来对基于直接路由的调度器进行压力测试，同样是针对之前的一系列不同开销的内容。我们照例将测试结果补充到前面的表 2-7 中，但是这次我们去掉了反向代理的那一列，因为我们这里主要关注的是 LVS-DR 和 LVS-NAT 的比较，压力测试结果如表 2-9 所示。

表 2-9　实际服务器、LVS-NAT、LVS-DR 分别对于不同内容的压力测试结果

内容类型	125.12.12.20	125.12.12.21	LVS-NAT	LVS-DR
静态（10Bytes）	13022.53	13328.71	15953.03	15617.39
动态（num=100）	10529.97	10642.43	13850.88	13941.09
动态（num=500）	8244.66	8177.55	13513.79	13813.6
动态（num=1000）	6950.16	6634.42	12941.28	13239.25
动态（num=5000）	2771.98	2654.07	5310.58	5210.18
动态（num=10000）	1444	1478.96	2870.92	2893.08
动态（num=50000）	322.26	331.79	646.03	649.07
动态（num=100000）	151.9	157.48	303.68	303.88

同样，我们将这些数据绘制成对比曲线图，更加直观地观察它们的变化趋势，如图 2-18 所示。

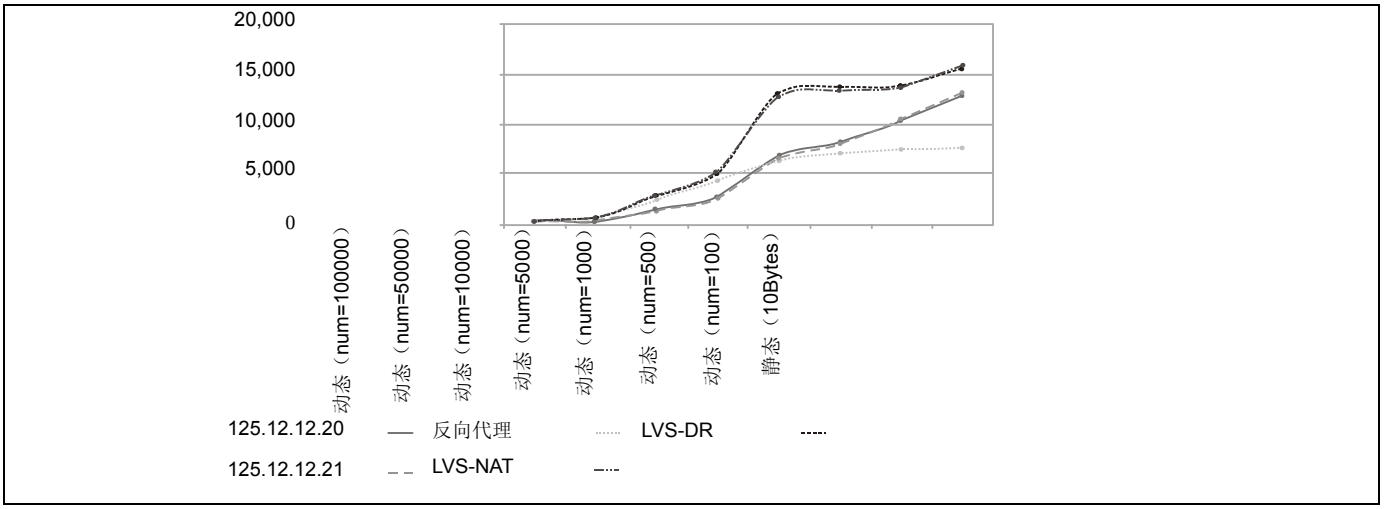


图 2-18　实际服务器、反向代理、LVS-NAT、LVS-DR 分别对于不同内容的压力测试结果对比曲线图

可以看到，LVS-DR 的表现几乎和 LVS-NAT 旗鼓相当，同时，它们都远远超过了基于反向代理的负载均衡系统。那么，相比于 LVS-NAT，LVS-DR 的优势在哪里呢？还记得这种方式的最大特点吗？那就是实际服务器的响应数据包可以不经过调度器而直接发往用户端，如图 2-19 所示，这也是它最大的优势，显然，要让它发挥这种优势，我们希望响应数据包的数量和长度远远大于请求数据包，事实上，大多数 Web 服务正是具备这样的特点，响应和请求并不对称，即便是几十 KB 的网页下载，响应数据包也是请求数据包的很多倍，更不要说大文件的下载。

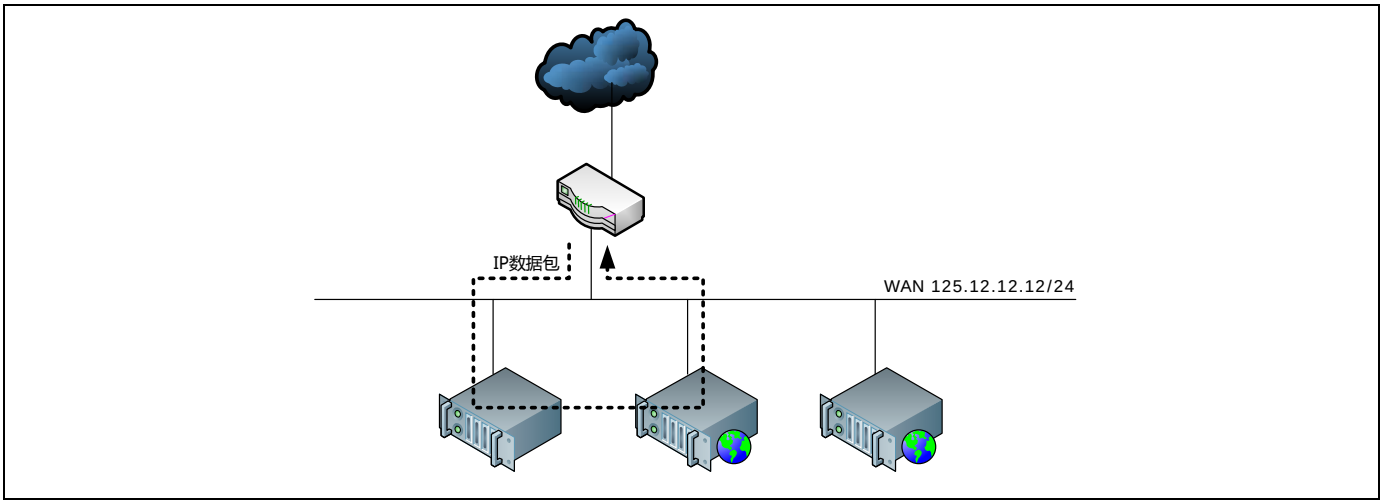


图 2-19　LVS-DR 的流量走向图

这样一来，大量的响应数据包便可以绕过调度器，避免了 LVS-NAT 中调度器的带宽瓶颈，为此，我们来做一个实验，同样基于刚才的网络结构，但是我们在 WAN 交换机上将调度器（10.12.12.12）和两台实际服务器（125.12.12.20、125.12.12.21）的带宽限制为 10Mbps，但不限制 125.12.12.10 这台服务器，它的带宽保证在 100Mbps，因为我们需要通过它来发起测试，事实上也可以假想它就是网关出口。

接下来，我们对实际服务器、LVS-NAT 和 LVS-DR 分别进行了压力测试，这次被测试的内容是一个大小为 22KB 的静态内容，你可以把它当作一个站点的首页 HTML。

测试结果如表 2-10 所示。

表 2-10　针对 22KB 静态内容的压力测试结果对比

被测试服务器	吞吐率（reqs/s）	数据流量（KB/s）	带宽使用量（Kbps）
实际服务器 1	51.75	1134.44	9075.52

实际服务器 2	51.76	1134.71	9077.68
LVS-NAT	51.69	1133.18	9065.44
LVS-DR	102.30	2242.51	17940.08

可以看到，对于前两台实际服务器，吞吐率基本相当，但是我们关心的是数据流量，因为这时候显然带宽成为制约吞吐率的瓶颈，从测试结果上看，带宽使用量接近 10Mbps，但实际上不可能达到理论上的 10Mbps，因为还存在交换机和服务器用户进程对数据包的处理时间。

第三行的 LVS-NAT 意味着调度器通过 NAT 方式将请求分散到前两台实际服务器，这里我们对于连接调度器和两台实际服务器的 LAN 交换机并没有限速，仍然是 100Mbps 的全双工模式，但是我们知道，调度器在 WAN 交换机上的出口带宽已经被限制为了 10Mbps。从测试结果上看，LVS-NAT 并没有提高整体吞吐率，它的瓶颈仍然是调度器的带宽。

最后的 LVS-DR 让我们眼前一亮，数据流量几乎翻了一番，吞吐率自然也随之翻倍，现在你知道 LVS-DR 的优势所在了吧。

虽然这个实验中我们只是将带宽限制到了 10Mbps，但是对于 100Mbps 甚至更高的带宽，其本质都是一样的，我们假设一个理想的带宽配置，如图 2-20 所示。

可以看到，即使调度器只有 100Mbps 的带宽，整个集群也可以通过增加大量的实际服务器来达到 1Gbps 的流量，也就是 WAN 交换机的出口带宽，这充分体现了 LVS-DR 的强大扩展能力。

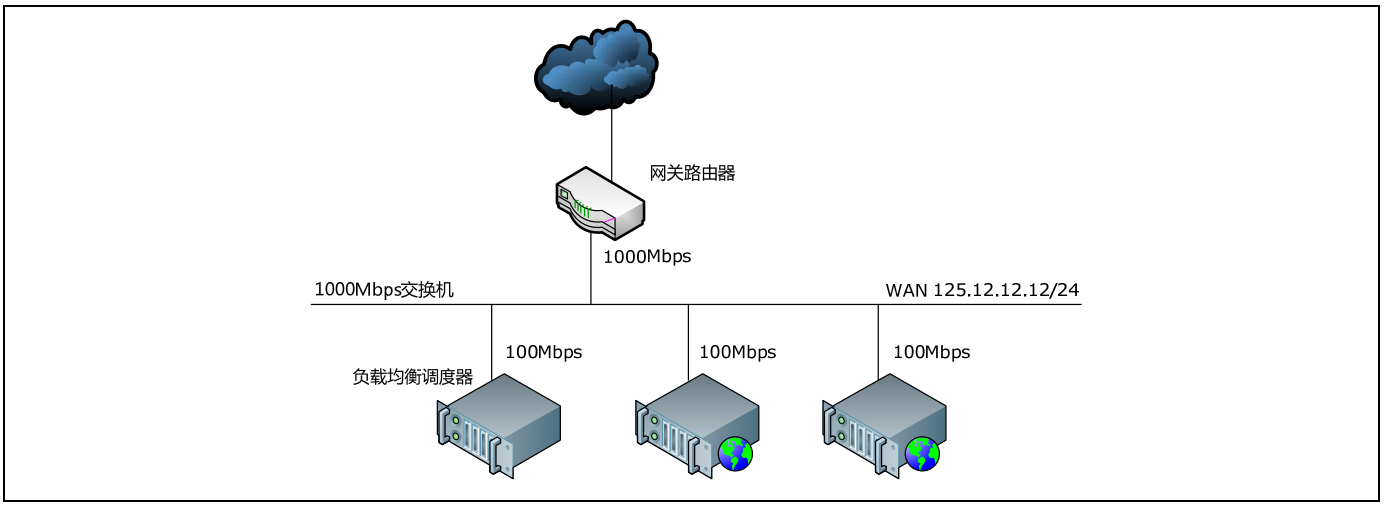


图 2-20 _LVS-DR 的理想带宽配置

由于测试环境的局限，这里我们无法测试 LVS-DR 究竟可以扩展多少台实际服务器，但可以肯定的是，这时候的瓶颈更多地转移到了 WAN 的出口带宽，相比之下，其他在单机上的问题将显得微不足道，因为你可以增加更多的实际服务器来分散诸如 CPU 和磁盘 I/O 等开销。

另一个关键的因素在于，响应数据包和请求数据包的比例，因为从调度器发出请求数据包的速度也是有上限的，是否能够让这些有限的请求数据包引发最大程度的响应风暴，也影响到整个系统的扩展能力。所以，越是响应数据包远远超过请求数据包的服务（如视频），就越应该降低调度器转移请求的开销，也就越能够提高整体扩展能力，最终也就越依赖于 WAN 出口带宽。来自 LVS 官方站点的早期测试结果也告诉我们，LVS-DR 可以容纳 100 台以上的实际服务器，我想对于某些特定的场景，这样的表现没有任何问题。

转型到 DNS-RR

除了性能和扩展性的优势之外，LVS-DR 也非常便于管理，这得益于所有的实际服务器都直接接入 WAN，所以你完全可以直接通过安全 SSH 来登录它们进行各种操作。

而对于 LVS-NAT，你必须登录调度器才可以访问位于 LAN 的实际服务器，这给管理大量的机器带来不便，但这还不算什么，

它也许会给你带来一些安全感。关键问题在于，一旦调度器有朝一日出现故障无法登录，那实际服务器便和你完全隔离了。当然，办法也是有的，你可以让其中某台实际服务器也接入 WAN，作为冗余“跳板机”，或者设置备用调度器，通过 Heartbeat 来完成自动故障转移，随后我们会介绍这方面的内容。

当然，闲置一台备用调度器，对于一些普通规模的站点来说，可能并不愿意。幸运的是，对于 LVS-DR，一旦调度器失效，你可以马上将 LVS-DR 切换到 DNS-RR 模式，这几乎只需要增加几条 DNS 记录，将域名解析到多台实际服务器的真实 IP 地址即可。一旦调度器恢复后，你便可以再次修改 DNS 记录，将域名仅指向调度器，切换回 LVS-DR。

总的来说，LVS-DR 非常适合搭建可扩展的负载均衡系统，不论是 Web 服务器还是文件服务器，以及视频服务器，它都拥有出色的表现。但前提是，你必须为实际服务器购买一系列的合法 IP 地址，不过，相比于负载均衡硬件设备，它们还是要便宜得多。

2.7 IP 隧道

与 LVS-DR 的原理非常类似，基于 IP 隧道（IP Tunneling）的负载均衡系统同样可以用 LVS 来实现，也称为 LVS-TUN。与 LVS-DR 不同的是，实际服务器可以和调度器不在同一个 WAN 网段，调度器通过 IP 隧道技术来转发请求到实际服务器，所以实际服务器也必须拥有合法的 IP 地址。

基于 IP 隧道的请求转发机制，简单地说，它是将调度器收到的 IP 数据包封装在一个新的 IP 数据包中，转交给实际服务器，然后实际服务器的响应数据包可以直接到达用户端。

当然，要实现 IP 隧道技术还存在一定的前提条件，那就是所有的服务器都必须支持“IP Tunneling”或者“IP Encapsulation”协议。幸运的是，Linux 对此支持良好，同时，“IP Tunneling”正成为各个操作系统的标准协议，所以为实际服务器使用各种操作系统也将成为可能。

对于 LVS-TUN 的配置和性能测试，我们这里就不做详细介绍了，当你了解并实践了 LVS-DR 后，LVS-TUN 对你来说不会陌生。

另外，基于 IP 隧道的独特方式，我们可以将实际服务器根据需要部署在不同的地域，并且根据就近访问的原则来转移请求，比如一些 CDN 服务便是基于 IP 隧道技术来实现的。

总的来说，LVS-DR 和 LVS-TUN 都适合响应和请求不对称的 Web 服务器，可以非常有效地提高集群的扩展能力，但如何选择它们，更多的不是因为性能和扩展性，而是取决于你的网络部署需要，比如刚才提到的 CDN 服务需要将实际服务器部署在不同的 IDC，从而必须使用 IP 隧道技术。

2.8 考虑可用性

对于一些关键的 Web 应用，可用性至关重要，为了实现高可用性的系统，我们不能容忍任何的单点故障，即便只是偶然。所谓的单点故障，是指系统中一旦某个组件发生故障，便会导致整个系统的失败，所以这种故障是致命的。

提示：

没有侥幸这回事，最偶然的意外，似乎也都是有必然性的。

——【美】爱因斯坦

在负载均衡系统中，多台实际服务器在分散开销的同时，本身也提高了实际服务器的可用性，一般来说，为了在个别实际服务器发生故障后整个系统能够继续承载同样的负载，我们必须将实际服务器的数量保证在略多于实际情况下的数目，这也有利于避免由于大量突发请求造成的雪崩效应。

而对于调度器，转移请求的机制注定它存在单点故障，为此，我们必须通过其他办法来有效实现故障平滑转移，以保证调度器的高可用性。

Heartbeat 可以很好地解决这个问题，它的诞生正是为了实现高可用性。简单地说，我们可以准备一台备用调度器，通过运行

Heartbeat 对主调度器进行心跳检测，一旦发现主调度器停止心跳，便立即启动故障转移，接管主调度器，这个接管过程包括 IP 别名变更、相关服务的启动等。随后，一旦主调度器恢复后，备用调度器便自动将相关资源转交回主调度器。

为了避免主调度器和备用调度器之间线路的单点故障，大量的事实证明，最好采用多条独立线路进行连接，这样将不依赖需要电源的交换机。你也可以使用串行电缆，通过服务器的 COM 端口进行连接，虽然线路长度有限，但这样还可以带来一点点额外的安全性，即便是主调度器被恶意破坏者登录，它也无法通过串行电缆登录备用调度器。

至于交换机，由于异常流量造成阻塞的情况时有发生，但大多数情况下发生在 IDC 级别的交换机上，所以不需要我们担心。一般来说，IDC 级别的交换机在物理层发生故障的概率较小，即便是发生了，更换备用交换机也并不困难，灾难总是在所难免的。

另外，作为物理层设备的网线，也有可能发生故障，虽然我们很少遇到，但如果希望让线路不存在单点故障，可以使用 Linux Bonding 技术来将多条线路绑定在一台服务器的多个网卡上，对流量进行 RR 负载均衡，或者将一条线路设置成为备用模式。Bonding 在 Linux 内核 2.4.12 以后已经被默认支持，你可以通过以下方法查看内核是否支持 Bonding，否则便需要将它编译到内核。

```
s-colin:~ # modprobe -l | grep bonding
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/drivers/net/bonding/bonding.ko
```

值得一提的是，通过 Bonding 实现流量负载均衡，还可以帮助我们实现带宽聚合，比如我们可以将 6 根 100Mbps 独享的线路绑定到 bond0 虚拟网卡上，实现 600Mbps 的出口带宽，我们对下载服务器采用了这种策略，可以看到以下结果：

```
s-colin:~ # cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.0.1 (January 9, 2006)
Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0
Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:24:8c:33:f4:b8
Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:24:8c:33:f4:b9
Slave Interface: eth2
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:04:23:89:69:e2
Slave Interface: eth3
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:04:23:89:69:e3
Slave Interface: eth4
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:0e:0c:a2:d9:08
Slave Interface: eth5
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:0e:0c:a2:d9:09
```

内容分发和同步

的确，利用 NFS 等共享文件系统可以帮助我们在多台服务器之间共享文件，但是在这种机制下，不论是性能还是可用性，都无法达到更高的要求，更关键的是，共享文件系统本身就是一个不强调扩展的概念，它更像一个中央集权的统治体系，最终将成为制约发展的罪魁祸首。

提示：

中央集权是相对于地方分权而言，其特点是地方政府在政治、经济、军事等方面没有独立性，必须严格服从中央政府的命令，一切受控于中央。中华人民共和国建立后，实行中国共产党宣称代表了民主集中制的人民代表大会制，中央集权和地方分权相结合，既保证中央统一领导，集中处理国家事务，同时又充分发挥地方的主动性和积极性，使地方享有一定的自主权。

对于上面这段话，道理很容易理解，不过我们要做的可没有这么复杂，对于大规模站点的文件存储和访问，我们如何把“中央集权和地方分权相结合”呢？显然，这里的“地方分权”在某种意义上意味着让 Web 服务器可以拥有独立的文件副本，所以我们需要将文件复制到多台服务器上，同时需要考虑更多的问题，那么具体如何做呢？

3.1 复制

还记得前面我们提到的图片服务器吗？当时我们采用 NFS 的方式将它映射到多台 Web 服务器上，而在这里，我们希望将图片服务器上的照片文件复制到集群中的每一台 Web 服务器上，如图 3-1 所示。

这样一来，Web 服务器将可以直接读取本地磁盘的图片来响应用户的 HTTP 请求，这意味着只要 Web 负载均衡调度器不出意外，那么文件访问本身将不再成为瓶颈，因为共享文件系统的单点性能问题已经彻底不存在了。

但与此同时，一个更重要的问题诞生了，那就是如何实现复制呢？更加严峻的问题是在面对大量的服务器进行复制时，我们需要考虑哪些策略呢？

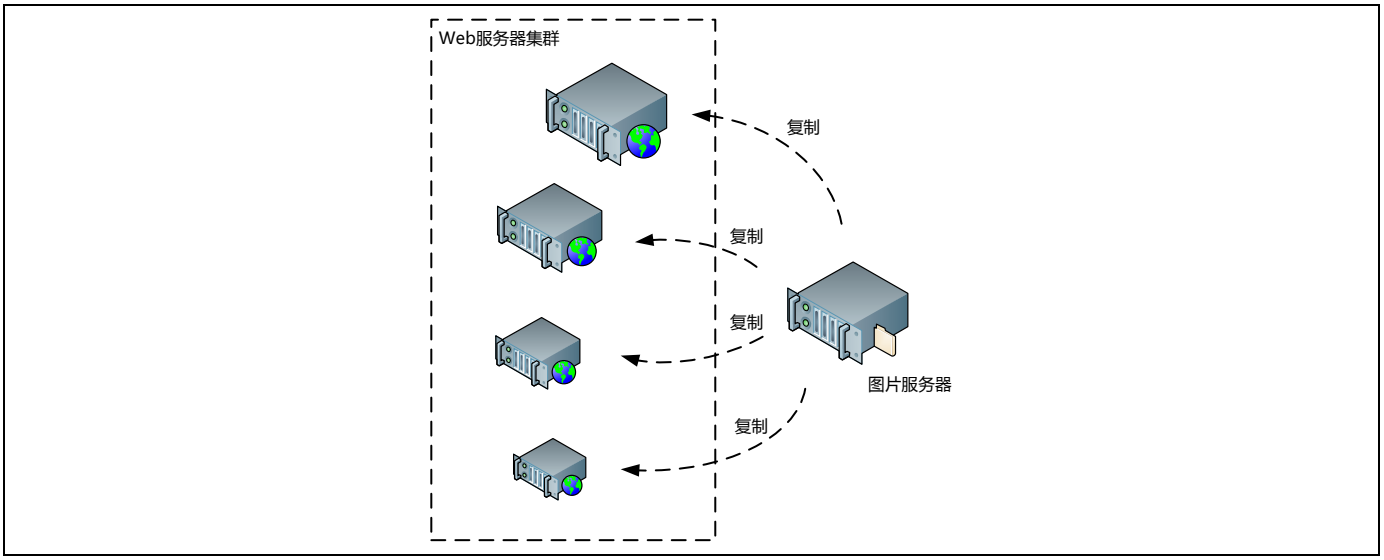


图 3-1 图片服务器到多台 Web 服务器的文件复制

提示：

每一项重大成就都会带来新的问题。任何一个发展随着时间的推移都会出现新的严重的困难。

——【美】爱因斯坦

总的来说，我们可以通过两种方式来实现复制，分别为主动分发和被动同步，主要区别在于复制的发起方和触发方式不同，所以，这里的“被动”实质上是相对于发送文件一端而言的。无论如何，它们都基于 TCP/IP 网络来传输数据。

针对这两种方式，如果条件允许，你完全可以自己开发软件来实现，这样的好处是你可以根据站点的需要来实现一些富有针对性的功能。我们曾经为站点开发了一套专用的页面分发系统，它具备一些特色，比如对于小文件的合并传输，我们知道对于小文件（如几十 KB 以下），每次传输的准备工作和收尾工作都是相当不划算的，合并传输则带来持久连接的优越性。除此之外，我们还支持异步复制，并且由分发进程来保证复制的成功率。此外，我们还可以充分利用合适的 I/O 模型和并发策略，提高分发服务器端的并发处理能力。

当然，对于大多数中等规模的站点来说，完全可以通过一系列的开源软件来实现复制，下面我们就来分别介绍它们。

3.2 SSH

提到 SSH（Secure Shell），大家并不陌生，它是建立在应用层和传输层基础上的安全协议，可以用于传输任何数据，我们希望用它来实现文件复制，当然，这属于主动分发的方式。

SSH 有很多功能，它既可以代替 Telnet，又可以为 FTP、POP 等协议提供安全的传输通道。关于它的安全性，这里我们就不重点介绍了，而它作为传输通道的表现，正是我们所关心的，这里不得不提到它的另一个优点，那就是它对传输的数据进行了压缩，以加快传输速度。

值得一提的是，我们这里需要的并不是基于命令行的 SSH，而是将 SSH 的能力集成到站点的应用程序中，幸运的是，很多用于 Web 开发的主流语言都拥有 SSH 客户端代码库或者相应的扩展。这里我们以 PHP 为例，它可以通过 PECL 扩展来实现 SSH 客户端操作，更重要的是，它还实现了基于 SSH 的 SCP 和 SFTP，这正是我们进行文件复制所需要的。

SCP

在 PHP 程序中通过 SCP 来进行文件分发并不困难，按照函数手册的指引，不需要几行代码就可以完成服务器之间的文件复制。这里有一点需要注意，在 SSH 服务器端，也就是分发文件的目标服务器上，我们需要对 SSH 的服务器端配置选项进行一些修改，通常配置文件为/etc/ssh/sshd_config，我们修改以下内容：

```
UseDNS no
PasswordAuthentication yes
```

这样一来可以减少 SSH 服务器进行 DNS 解析的时间，并且允许我们在 Web 应用程序中通过密码验证的方式来登录 SSH 服务器。

一个用 PHP 编写的用于分发文件的例子如下所示：

```
<?php
$conn = ssh2_connect("10.0.1.201", 22);
ssh2_auth_password($conn, "user", "pwd");
ssh2_scp_send($conn, "/home/user/list.htm", "/home/user/list.htm", 0644);
?>
```

的确非常容易，那么，再次回到 I/O 延迟的问题上来，通过基于 SSH 通道的 SCP 进行文件分发，它的 I/O 延迟与前面的 NFS 相比，结果如何呢？

我们同样对 22KB 和 23MB 的两个文件进行了分发测试，结果如表 3-1 所示。

表 3-1　远程 SCP、本地 SCP 和本地磁盘写操作测试结果

文件大小	远程 SCP	本地 SCP	本地
22KB	0.182s	0.210s	0.002s

23MB	2.452s	0.654s	0.109s

与前面 NFS 的测试结果相比，我们不难看出，对于 22KB 的文件，SCP 花费了较多的时间，这或许归咎于额外的身份验证机制。对于 23MB 的大文件，远程 SCP 传输的时间和 NFS 的测试结果不相上下，这时候身份验证的开销已经成为次要因素，而在本地 SCP 传输中，身份验证的开销再次成为不可忽略的一部分。

显然，在绝对速度的对比中，SCP 并没有什么明显的优势，但我们要知道，关键的问题是，在我们试图通过文件复制来创建更多副本的那一刻起，绝对速度已经不是那么重要了，相比之下，如何将文件复制到更多的服务器上成为延续生命力的关键所在。所以，这又回到应该选择共享文件系统还是文件复制的问题上，对此，我们的目的已经完全不同。

在共享文件系统中，文件都存储在 NFS 服务器上，即便网络带宽可以充分满足需要，但总有一天 NFS 服务器的磁盘会成为性能的杀手，从而制约更大规模的共享访问。而在文件分发机制中，文件被复制到多台服务器上，这本身便分散了磁盘 I/O，相比于 NFS 的集中式访问，这相当于是针对磁盘 I/O 的负载均衡。

SFTP

在实际应用中，除了用 SCP 来传输文件，我们还需要对远程服务器进行必要的文件系统操作，比如创建目录、删除文件等，前面提到了 SFTP，我们可以在 PHP 中轻松地操作它，如下所示：

```
<?php
$conn = ssh2_connect("10.0.1.201", 22);
ssh2_auth_password($conn, "user", "pwd");
$sftp = ssh2_sftp($conn);
ssh2_sftp_mkdir($sftp, "/home/user/newdir", 0666);
?>
```

通过 SFTP，我们可以对远程服务器的文件系统进行任何操作，就像操作本地的文件系统一样，当然，前提是你必须拥有合法的权限。

多级分发

在较大规模的站点中，我们可能需要将文件复制到更多的服务器上，比如我们拥有 200 台静态内容服务器，它们通过 LVS 和 DNS 等混合方式实现负载均衡，并且分布在不同地域的 IDC 中，对于任何静态内容的更新，我们都需要快速地更新到这 200 台服务器上。

显然，直接将文件从文件服务器分发到 200 台 Web 服务器很容易产生以下问题：

- 由于目标服务器较多，并且部署在不同地域，分发过程会持续较长时间，这会造成一部分目标服务器表现出较大的内容更新延迟。
- 大量消耗文件服务器的本地系统资源，当分发任务较多时，本地会产生大量进程阻塞，影响其他任务的正常运行。

为此，我们可以考虑进行多级分发，如图 3-2 所示。

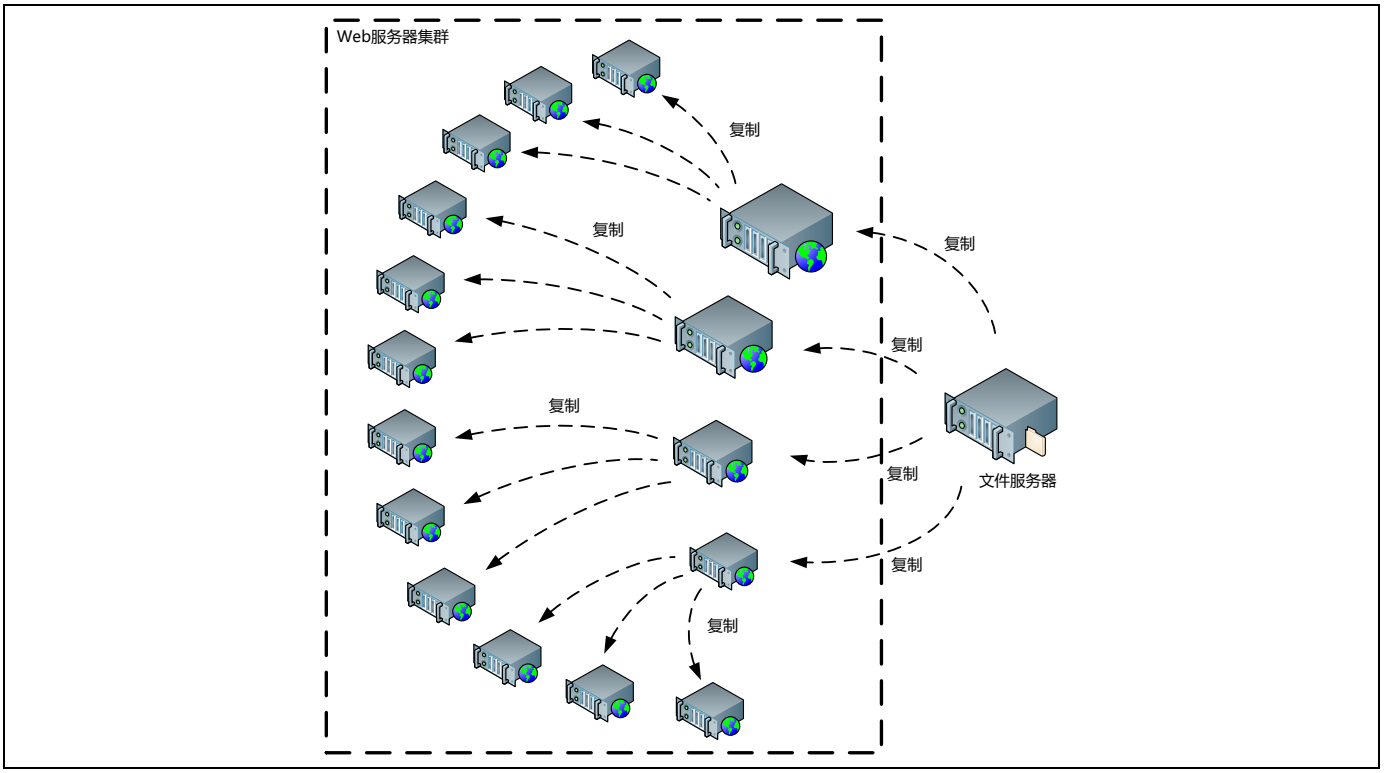


图 3-2　静态内容的多级分发

这样一来，文件服务器将很大一部分工作量巧妙地转移到了更多的 Web 服务器上，并且可以充分利用地域就近分发策略，一些 Web 服务器可以更快地将文件转发给同地域的其他 Web 服务器。

实现这样的多级分发并不困难，即便是无法开发专用的分发软件，你仍然可以基于一系列的开源产品进行快速构建，比如：

- 通过 HTTP 请求来触发某台 Web 服务器启动某文件的分发；
- Web 服务器使用异步的方式进行文件分发，分发采用前面提到的 SSH 扩展，分发结束后根据需要通知下一级进行分发。

对于异步方式，我们会在后面关于分布式计算的章节中进行介绍。另外，对于以上的多级文件分发，我们应该将用于分发的网络进行最大程度的流量隔离，使它不要消耗 Web 服务器的 WAN 带宽。一般而言，对于跨地域或者跨 IDC 的文件分发，除非使用专线，否则必须通过 WAN 来进行，而对于同一 IDC 内的文件分发，我们完全可以根据需要来组建 LAN，实现分发流量隔离。

3.3　WebDAV

另一种实现文件分发的简单方式是利用 HTTP 扩展协议 WebDAV。RFC2518 对 WebDAV 进行了详细的定义，它允许我们基于 HTTP/1.1 协议来对 Web 服务器进行远程文件操作，包括文件和目录的创建、修改和修改等，另外，WebDAV 的设计目的还包括了对于版本控制的支持，还记得 Subversion 的 HTTP 工作方式吗？它便是使用 WebDAV 来实现的。

如今，主流的 Web 服务器软件几乎都支持 WebDAV 扩展，比如 Apache 和 Lighttpd 都有相应的 WebDAV 模块，安装并配置它们非常容易，同时，你可以使用 litmus 来测试它们对于 WebDAV 的支持程度，比如是否支持 COPY、MOVE 等指令，并且能否正常工作等。

WebDAV 完全基于 HTTP，它的定义方式与 RESTful 风格十分接近，或者说 RESTful 是对 WebDAV 风格的回归，无论如何，WebDAV 很容易使用，你几乎可以自己构造客户端 HTTP 请求，比如：

```
MKCOL /files/2009/ HTTP/1.1
Host: www.server.org
```

这意味着我们请求 Web 服务器创建相应的目录，随后的 HTTP 响应如下所示：

```
HTTP/1.1 201 Created
```

这代表目录创建成功。当需要携带更多的描述信息时，WebDAV 使用 XML 格式来组织数据，所以，我们同样可以非常容易地操作它。

后面我们要介绍的分布式文件系统 MogileFS 便采用了 WebDAV 来实现文件分发，当然，WebDAV 只是一个文件分发的具体实现，对于分发的策略和管理，前面介绍的内容同样适用。

3.4 rsync

除了 SCP 或 WebDAV 等主动分发方式，我们还可以采用被动同步的方式来实现文件复制，在这种情况下，接收文件的一端将主动向文件服务器发起同步请求，并根据两端文件列表的差异，有选择性地进行更新，从而保证它和文件服务器的内容一致。

Linux 下的 rsync 工具便可以非常出色地完成这项任务，但值得一提的是，通常情况下 rsync 可以采用 SSH 作为传输通道，但这种方式在性能上受限于 SSH 的传输机制，通过前面的介绍，我们知道 SSH 存在一些额外开销，特别对于小文件的传输，这些开销更显得很不值得。所以，如果条件允许的话，我们尽量使用 rsync 的独立服务器端进程 rsyncd 来负责文件传输，它同时也将使用独立的服务器端口。

启动了 rsyncd 后，便可以在进程树中看到它：

```
 3961 ?        Ss     0:27 /usr/sbin/rsyncd --daemon
```

通过 rsyncd 的服务器端配置，定义多个需要同步的目录，同时还可以指定账号文件，即便不使用 SSH，也可以实现身份验证。比如在以下的配置中，我们定义了 img，并指向站点的图片目录。

```
s-cache:~ # vi /etc/rsyncd.conf
gid = users
read only = true
hosts allow = 10.0.1.0/24
use chroot = no
transfer logging = true
log format = %h %o %f %l %b
log file = /var/log/rsyncd.log
slp refresh = 300
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
lock file = /var/run/rsyncd.lock
[img]
       path = /data/www/img/htdocs
       comment = img.highperfweb.com
       read only = yes
       auth users = user
       secrets file = /etc/rsyncd.secrets
```

这样一来，拥有授权 IP 地址的服务器便可以随时从这台服务器同步文件，一般来说，我们会将同步指令放置在定时任务中，比如通过以下的 crontab 设置，同步任务将每 5 分钟进行一次。

```
*/5 * * * * rsync -vzrtopg --progress --delete img@10.0.1.200::img /data/www/img/htdocs/ > /dev/null
2>&1
```

一切都非常简单，通过定时任务，rsync 对文件的同步可以完全对应用程序透明，但这也恰恰使得应用程序无法细粒度地控制它的工作。当然，你也可以通过应用程序来很好地控制 rsync 的节奏，比如让它不间断地重复运行，即便是一个简单的 Shell 程序也能做得到，这样可以最大程度地减少被动同步机制的复制延迟。

另一方面，从我们关心的性能角度来看，对于少量文件的同步，rsync 的表现基本上无可挑剔，但是对于大量的文件，情况便不那么乐观。注意，这里所说的并不是实际复制的文件数目，而是被同步的目录中所有文件个数，因为 rsync 在同步的时候必须扫描被同步目录中的所有文件，并根据文件最后修改时间，和本地进行对比，以找出需要复制的文件，或者需要删除的文件，这样一来，当目录中存在大量的文件时，扫描的开销便可想而知。

我们来看一个例子，以下的 rsync 同步命令将执行近 2 分钟。

```
s-img:~ # rsync -vzrtopg --progress --delete img@10.0.1.200::img /data /www/img/htdocs/
receiving file list ...
```

```
612600 files to consider

sent 66 bytes  received 15109850 bytes  165135.69 bytes/sec
total size is 11426157579  speedup is 756.20
```

可以看到，当扫描结束后，并没有引发实际的复制操作，这说明本地目录中的文件和文件服务器上是保持一致的，但即便是这样，也花费了近 2 分钟的时间。

那么，对于大量文件的同步，有什么更好的办法吗？

3.5 Hash tree

rsync 在同步文件时需要分析目录中每一个文件的更新标记，当有多级目录时，依然如此。如果能让它在减少扫描范围或者次数的同时，仍然可以实现目的，那当然是我们所希望的。

也许你已经发现，事实上在大多数时候，很多文件并没有更新，为什么不把少量文件的更新标记不断地传递到上一级目录呢？这样便可以让扫描过程少走很多冤枉路。

如图 3-3 所示，在文件服务器上，假如底层的灰色文件发生了更新，我们需要让它的上级目录也随之更新，最后修改时间，并且向上以此类推，直到顶层目录，这样一来，从这个文件到根目录的一系列节点都会更新最后修改时间，当同步扫描的时候，只需要遍历那些修改时间更新的目录即可，扫描次数将大大减少。

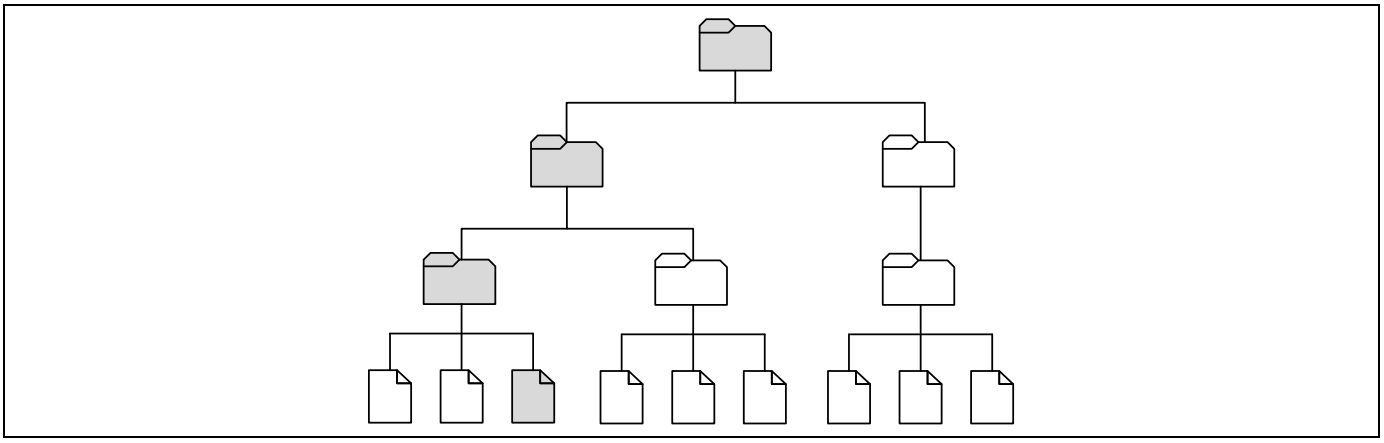


图 3-3　将文件的修改时间传递到上级目录

但是，操作系统本身对于文件的修改并不会自动更新上级目录的修改时间，一些特定的应用程序会这样做，比如通过 VI 编辑某个文件并保存后，你会发现它的所有上级目录都会自动更新修改时间。那么，对于文件同步，我们也必须想办法自己来实现。

幸运的是，Linux 内核中的 inotify 模块可以帮助我们完成这一工作，简单地说，它通过在内核中监视文件系统的一举一动，可以在任何文件的修改时间发生变化后，发出事件通知。这样一来，我们可以通过 inotify 提供的 API 来编写守护进程，持续不断地监视文件修改事件，一旦发现某个文件被修改，便随之更新上级甚至更上一级目录的修改时间，直到被同步目录的根节点。

inotify 本身提供了基于 C 语言的 API，除此之外，你也可以在 PHP 中使用相应的 PECL 扩展，这将大大简化我们的开发工作。有一点需要注意，inotify 已经直接包含在了 Linux2.6.13 的内核中，在此之前，我们必须安装相应的补丁，所以在编译 PHP 扩展的时候，你需要检查 Linux 的内核版本。

```
s-colin:~ # uname -a
Linux s-mat 2.6.16.21-0.8-bigsmp #1 SMP Mon Jul 3 18:25:39 UTC 2006 i686 i686 i386 GNU/Linux
```

这种将文件的修改时间不断向上传递的机制体现了 Hash tree 的动机，但有所不同的是，在 Hash tree 的存储结构中，我们通过特定的 Hash 算法，为树中的每个非叶节点生成一个可以描述它所有子节点唯一性的特征字符串，以便提供对节点信息完整性和正确性的验证，这样一来，一旦某个节点的信息发生变化，那么这种变化便会向上传递，体现在上级节点的特征字符串中。

而这里，我们直接将文件修改时间作为 Hash 值，这非常适合于文件同步这种依赖时间的特殊应用，而且易于实现，几乎不需要额外的计算开销，的确，文件系统本身就是树形结构，Hash tree 的优越性在这里得到了很好的发挥。

到目前为止，我们所做的还只是通过 inotify 实现了文件修改时间的向上传递，除此之外，我们还必须实现基于这种机制的专用服务器端和客户端程序，因为 rsync 并不支持我们所期望的 Hash tree 式扫描，当然，也许你可以尝试修改 rsync 的源代码。另一方面，利用现有的通道也是一个值得考虑的选择，比如你仍然可以借助 SFTP 来获取远程目录列表，并且分析修改时间，然后通过 SCP 来获取需要同步的文件。

利用这种同步方式，我们还需要注意一点，为了最大程度地发挥 Hash tree 的优势，我们需要在目录规划时遵循一些原则，尽量让目录中的文件或者子目录保持在较少的数量，以减少扫描次数，当然，这也必然让扁平的目录结构变得更加冗长，带来更多的目录切换次数，但是，你可以根据实际情况来很好地把握平衡。

3.6 分发还是同步

总的来说，对于分发和同步这两种文件复制方式，你也许在思索究竟应该在站点中选择谁呢，这大可不必发愁，其实它们两者本质上没有太大的差别，都可以达到相对来说比较实时的文件复制，而在选择中起到决定作用的，往往是出于以下一些因素：

- 文件分发需要依赖一定的应用程序逻辑，比如通过 SCP 扩展来编写代码控制文件传输，或者自己开发专用的分发程序，而文件同步可以对应用程序完全透明，部署非常简单；
- 文件分发可以更好地控制触发条件，更加灵活，比如更容易实现多级复制，也就是前面介绍的多级分发；
- 文件同步（比如利用 rsync）是一种天然的异步复制操作，不阻塞 Web 应用程序的运行，而在 Web 应用中要想对文件分发实现异步复制操作，必须得借助其他支持，比如后面要介绍的异步计算。

根据站点的需要，选择合适的复制方式并不困难，相信你的感觉吧！

除此之外，还有一种复制方式，那就是反向代理，的确，它是一个多才多艺的家伙，在这里，我非常乐意将它也归入文件复制的范畴，接下来，我们看它如何发挥作用。

3.7 反向代理

从反向代理缓存到负载均衡调度器，反向代理服务器不止一次出现在书中，而这一次，我们又将目光转向了它在文件复制领域的表现。

事实上，反向代理机制本身就决定了它必须从后端服务器那里不断地复制内容到本地，而这种复制的触发条件，则是用户向反向代理服务器请求内容，我想这种方式也可以称为“动态同步”。

那么，对于静态网页、图片等这些直接暴露给用户的内容来说，通过反向代理服务器实现远程复制，可能是一个更加简单有效的方式。而对于另一些无法由用户通过 HTTP 直接访问的文件，比如 Web 应用程序运行中需要的一些持续更新的配置文件，反向代理服务器对于它们的复制将无能为力，你仍然需要借助于前面介绍的复制方法。

这样一来，我们或许可以将集群中的 Web 服务器直接作为文件服务器的反向代理，或者采用多级代理的方式，类似于多级分发，减少单点压力，同时有利于同地域服务器之间的就近复制。

另一方面，正是因为存在反向代理缓存，所以使用反向代理实现文件复制才有意义，因为这些静态网页或者图片可以长时间保留在反向代理服务器上，快速地为用户提供下载服务，而只有用户第一次请求它们的时候才需要触发复制。

但是，缓存机制也提醒了我们，任何内容总是会有过期的时候，假如这些内容需要频繁的更新，同时你也希望给用户及时的体验，那么让这些内容频繁过期便在所难免。显然，这样会引发频繁的复制，别忘了复制是需要花时间的，就像羊毛出在羊身上，这部分时间仍然需要由用户来承担，你会发现相当一部分用户都在为反向代理服务器到文件服务器的数据传输而买单。

当然，这是比较极端的情况，而在大多数时候，我相信你的站点会有很多可以长期在反向代理服务器上保留的内容。

使用反向代理的另外一个好处是，这种动态同步是基于标准的 HTTP，那么对于同步所需的服务器端程序，相比于 sshd 和 rsyncd，Web 服务器成为最佳选择，你对它已经再熟悉不过了，特别是它出色的并发处理能力，这使得它可以提供非常高效可靠的文件复制。

第 4 章

数据库扩展

当我们对 Web 计算和存储都进行了不同程度的扩展后，站点的规模不断膨胀，这给数据库带来了巨大的查询压力，尽管前面我们介绍了数据库性能优化的一些具体做法，但是，这些显然是不够的，数据库也必须与时俱进，进行扩展。

在解决性能问题的同时，数据库的扩展还带来了一些其他的作用，比如增加存储空间、提高可用性等。接下来我们会介绍数据库扩展的一些思路，它们基本上覆盖了大多数扩展方式。

4.1 复制和分离

正如前面我们将文件复制到多台服务器一样，数据库通过复制来创建冗余副本，同样可以达到分散查询压力的目的。

事实上，当你掌握了前面章节的所有内容后，这一章的内容本质上对你来说并不陌生，比如负载均衡、复制、分离等思路，它们都在前面或多或少地出现过。

主从复制

几乎所有的主流数据库产品都支持复制，这是它们进行简单扩展的基本手段，我们以 MySQL 为例，它支持主从复制，配置并不复杂，简单地说，你只需要做到以下两点：

- 开启主服务器上的二进制日志（log-bin）。
- 在主服务器和从服务器上分别进行简单的配置和授权。

我们知道，MySQL 的主从复制是依据主服务器的二进制日志进行的，也就是说主服务器日志中记录的操作会在从服务器上进行重放，从而实现复制，所以主服务器必须开启二进制日志，它会自动记录所有对数据库产生更新的操作，也包括潜在的更新操作，比如没有删除任何实际记录的 DELETE 操作。

显然，这种复制是异步进行的，从服务器定时向主服务器请求最新日志，而主服务器只需要通过一个 I/O 线程来读取本地二进制日志，并输送给从服务器即可，所以，复制过程对于主服务器的影响非常有限。但是，当存在多个从服务器同时从一个主服务器进行复制的时候，主服务器的磁盘压力会有不同程度的增长，这也并不是无法解决的，你可以采用多级复制策略，就像前面提到的多级分发一样。

在主服务器上对于所有更新操作记录二进制日志，这部分额外开销对性能大概有 1%的影响，与复制的意义相比，这几乎是微不足道的。

读写分离

一旦我们将数据库复制到更多的服务器上后，关键的问题来了，我们应该怎样合理地给它们分配工作量呢？

这里有一点需要注意，对于所有的更新操作，我们必须让它作用于主服务器上，这样才能保证所有数据库服务器上的数据一致。这也是任何使用单向复制机制的系统必须遵循的更新原则，比如在前面提到的文件分发和同步中，我们也只能对文件源进行更新。

为此，我们采用读写分离（R/W Splitting）的方法将应用程序中对数据库的写操作指向主服务器，而将读操作指向从服务器，如图 4-1 所示。

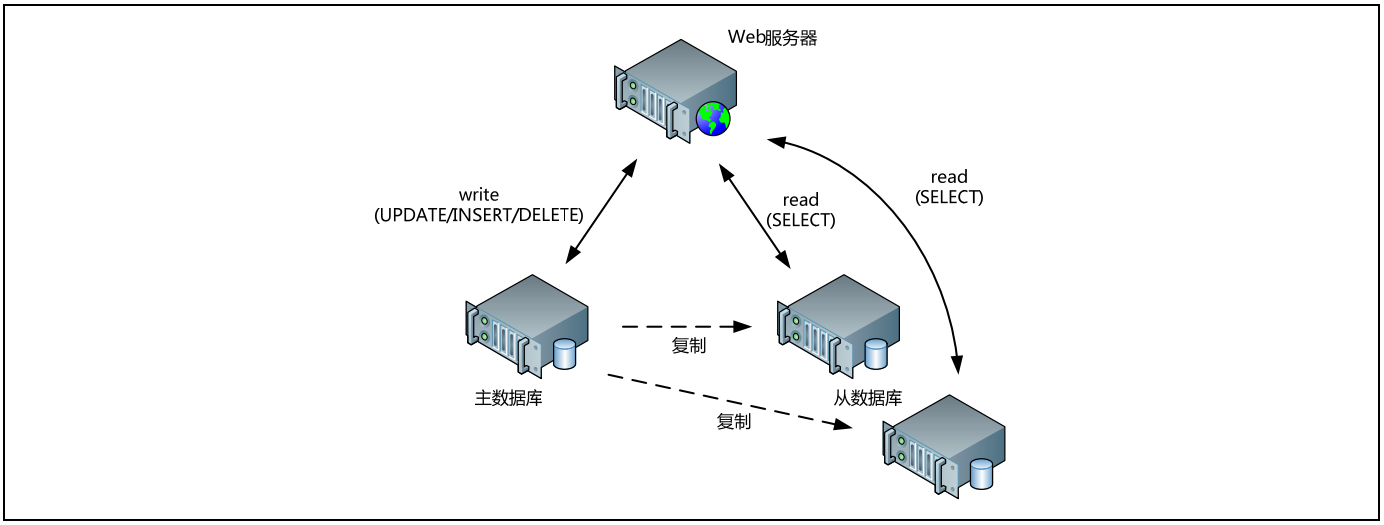


图 4-1　通过数据库主从复制实现读写分离

这样做不仅保证了多台服务器的数据一致性，更重要的是，它符合情理，一般而言，大多数站点的数据库读操作要比写操作更加密集，而且查询条件相对复杂，你可以通过 mysql-report 来查看 SELECT 操作的比例，通过前面对数据库性能优化的介绍，我们知道大部分的开销都消耗在这些查询上，而通过读写分离，我们可以将大量的读操作剥离出来，转移到更多的从服务器上实现水平扩展，这是一个非常简单而有效的策略。

当然，对于以用户创造内容为主的站点来说，数据库写操作依然密集，随后我们会介绍通过分区来实现扩展的思路。

一旦决定采用读写分离，我们必须在应用程序中配置多个数据库连接选项，并且修改数据访问层代码，的确，应用程序知道应该如何区分读写操作，然而，对于将读操作分散到多台从服务器上，应用程序就不那么擅长了，简单的随机分配可能会造成多台从服务器的工作量不均衡，更重要的是，当某台从服务器发生故障后，应用程序并不知道。当然，你也可以不修改应用程序，而将这部分工作交给数据库反向代理，我们来看看它的作用。

使用数据库反向代理

如果你在使用 MySQL，那么可以尝试 MySQL Proxy，它工作在应用程序和 MySQL 服务器之间，负责所有请求和响应数据的转发。

就像 Web 反向代理服务器一样，MySQL Proxy 同样可以在 SQL 语句转发到后端的 MySQL 服务器之前对它进行修改，比如将所有对数据库 A 更新的 SQL 语句修改为更新数据库 B，这也许有些不合时宜，但充分体现了 MySQL Proxy 大权在握，那么对于读写分离，它同样可以实现。

MySQL Proxy 通过 LUA 脚本来描述转发规则，以下的 LUA 代码用于实现读写分离：

```
-- 转移所有非事务的 SELECT 查询语句到从服务器
if is_in_transaction == 0 and
   packet:byte() == proxy.COM_QUERY and
   packet:sub(2, 7) == "SELECT" then
  local max_conns = -1
  local max_conns_ndx = 0
  for i = 1, #proxy.servers do
    local s = proxy.servers[i]
    -- 寻找有空闲连接的从服务器
    if s.type == proxy.BACKEND_TYPE_RO and
       s.idling_connections > 0 then
      if max_conns == -1 or
         s.connected_clients < max_conns then
        max_conns = s.connected_clients
        max_conns_ndx = i
      end
    end
  end
  -- 找到一个有空闲连接的从服务器
  if max_conns_ndx > 0 then
```

```
    proxy.connection.backend_ndx = max_conns_ndx
  end
else
  -- 转发到主服务器
end
return proxy.PROXY_SEND_QUERY
```

MySQL Proxy 默认监听在 4040 端口，我们可以在启动时通过选项参数来指定后端的 MySQL 服务器：

```
mysql-proxy \
--proxy-read-only-backend-addresses=10.0.1.202:3306 \
--proxy-backend-addresses=10.0.1.201:3306 \
--proxy-lua-script=/etc/mysql-proxy/rw-splitting.lua &
```

如图 4-2 所示，我们看到应用程序只需要跟 MySQL Proxy 通信即可，而读写分离的工作都由 MySQL Proxy 来完成，与此同时，MySQL Proxy 还对多个从服务器实现负载均衡以及可用性检测，这些工作由它来做，的确非常适合。

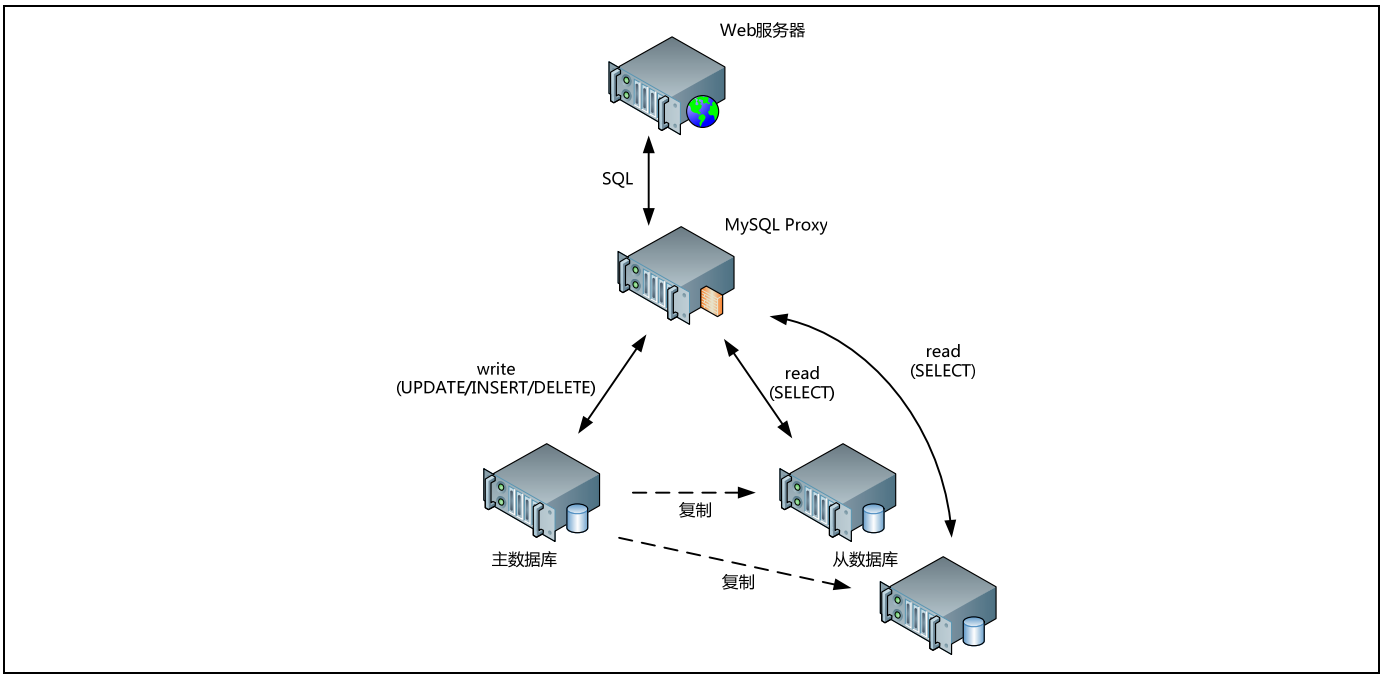


图 4-2　通过 MySQL Proxy 实现读写分离

但是，当存在大量的从服务器时，MySQL Proxy 必然会出现瓶颈效应，不过，即便是到那个时候，主服务器已经无法承受写操作的压力了，图 4-2 中的结构将不可避免地再次调整，接下来怎么办呢？

4.2　垂直分区

对于数据库写操作频繁（write-heavy）的站点来说，仅仅采用主从复制和读写分离可能效果并不明显，假如你的主服务器花费了 80%的时间在写数据，那么所有的从服务器也将花费更多的时间来同步数据，可想而知，所有从服务器只能依靠剩余不到 20%的时间来处理你的 SELECT 查询请求，这时候，增加从服务器所获得的回报将越来越少，呈现边际效益递减。

> 提示：
>
> 边际效益递减是经济学的一个基本概念，它说的是在一个以资源作为投入的企业中，单位资源投入对产品产出的效用是不断递减的，换句话说，就是虽然其产出总量是递增的，但是其二阶导数为负，使得其增长速度不断变慢，使得其最终趋于峰值，并有可能衰退。

另一方面，站点在成长，用户活动越来越频繁，写操作不断增多，主服务器的压力也逐渐接近极限，这时候不论你增加多少台从服务器都无济于事，因为那只是对读操作的分散，并没有对写操作起到任何作用。

这使得我们很自然地想到，应该对写操作也进行分离，这个问题刻不容缓，否则我们的时间将浪费在复制大量毫无意义的垃

级数据上。

最简单的分离方法当然是将不同的数据库分布到不同的服务器上，你会发现有很多数据库之间并不存在关系，或者不需要进行联合（JOIN）查询，那么为什么不把它们放在不同的服务器上呢？比如我们将用户的博客数据库和好友数据库分别转移到独立的数据库服务器上，这种方式称为垂直分区，如图 4-3 所示。

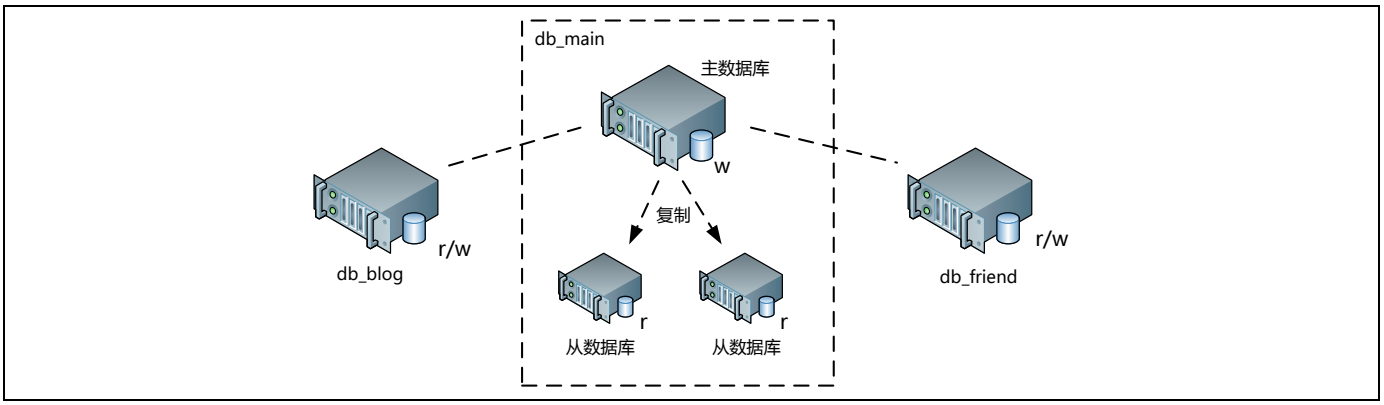


图 4-3　简单的垂直分区

可以看到，我们将 db_blog 和 db_friend 两个数据库分别转移到了独立的服务器上，而 db_main 则代表仍然存放在主服务器上的其他一系列数据库，你仍然可以在这里进行必要的联合查询，但是，不论从扩展的角度，还是从查询性能的角度来看，在进行数据库模型设计以及编写应用程序的时候，都应该尽量减少使用联合查询。

经过这样的垂直分区后，所有对于 db_blog 和 db_friend 的读写操作都将被分散到其他服务器上，如果它们带走了 60%的工作量，那么值得庆贺。

按照这样的思路，一旦我们需要为站点开发新的应用，便可以通过增加新的数据库服务器来实现扩展。

同样，我们还可以再次通过主从复制来对 db_blog 和 db_friend 两个数据库进行读写分离，如图 4-4 所示。

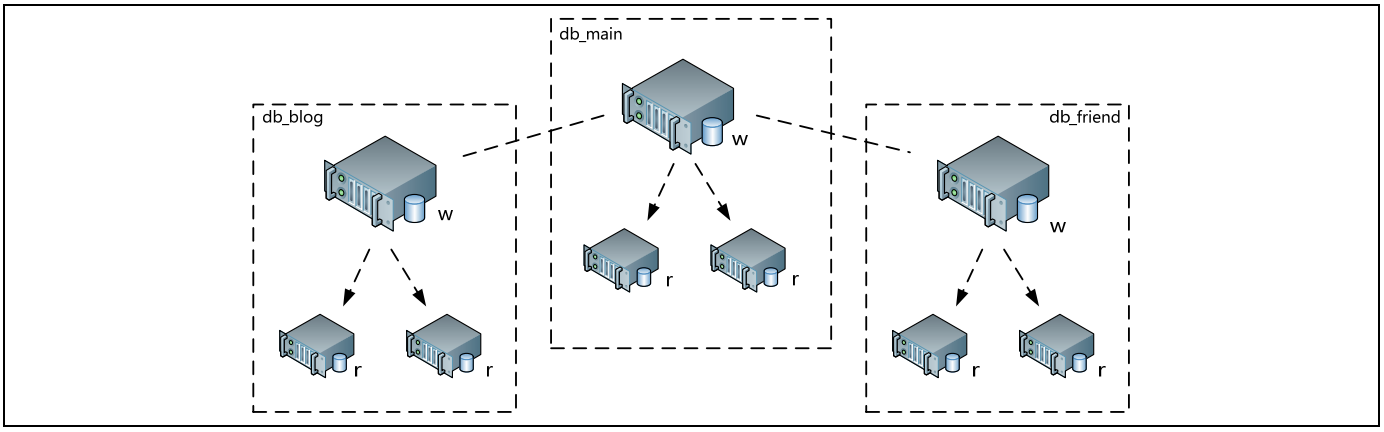


图 4-4　对多个分区分别进行读写分离

这的确是个不错的主意，然而，同样的问题在不久的未来仍然会困扰我们，当 db_blog 数据库的主服务器再次无法承受写操作压力时，我们又该如何呢？对这个数据库再次进行垂直分区吗？也许你可以将不同的数据表转移到不同的服务器上，但是当数据表也达到写操作极限的时候呢？似乎这种扩展方式要山穷水尽了，那么，我们换一种思路，来看看水平分区。

4.3　水平分区

水平分区（Sharding）意味着我们可以将同一数据表中的记录通过特定的算法进行分离，分别保存在不同的数据表中，从而可以部署在不同的数据库服务器上。

事实上，很多大规模的站点基本上都经历了从简单主从复制到垂直分区，再到水平分区的步骤，这是一个必然的成长过程。下面我们来看看如何实现水平分区，值得一提的是，水平分区并不依赖于特定的技术，它更多的是一种逻辑层面的规划，需要一定的经验和不断的分析。

把数据放在不同分区中

继续前面的例子，我们希望将 db_blog 数据库中的数据拆分到不同的服务器上，并且应用程序能够知道如何找到它们。

db_blog 数据库中存储了用户发表的博客内容，主要数据都在 tbl_posts 表中，它的结构如下所示：

```
CREATE TABLE `tbl_posts` (
  `post_id` int(11) NOT NULL auto_increment,
  `post_title` varchar(64) NOT NULL,
  `post_content` text NOT NULL,
  `post_time` int(11) NOT NULL default '0',
  `user_id` int(11) NOT NULL,
  PRIMARY KEY  (`post_id`),
  KEY `user_id` (`user_id`)
)
```

其中，user_id 代表博客用户的 ID，现在我们要将所有用户划分为两部分，为了尽量均衡，我们可以按照 user_id 的奇偶性质来划分，user_id 为奇数和偶数的博客内容分别存储到不同的数据库服务器上，如图 4-5 所示。

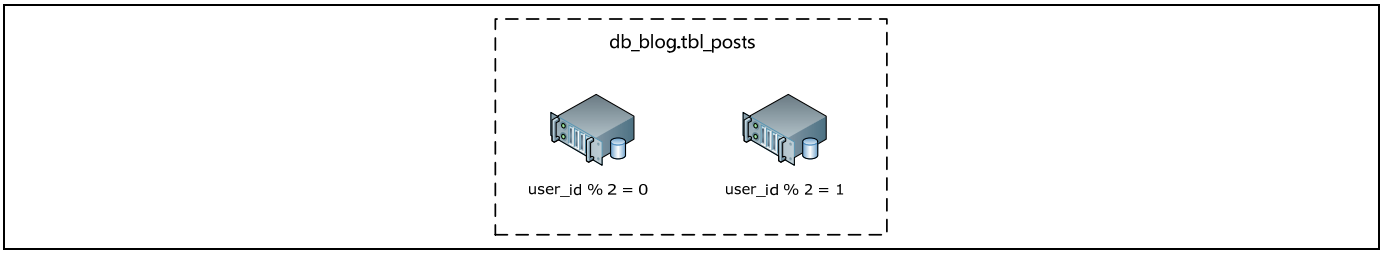


图 4-5　按照 user_id 奇偶性质进行水平分区

如此一来，我们建立了两个水平分区，它们分别位于图 4-5 中的两台服务器上，显然，只要我们知道博客内容的作者 ID，便可以知道应该去哪个分区查询内容。

同时，我们也要修改应用程序的数据访问代码。在分区之前，用户访问某篇博客内容时，可能使用以下的 URL：

```
http://www.highperfweb.com/blog_post.php?post_id=342352
```

通过 URL 中指定的 post_id，我们可以轻松地找到内容，代码如下所示：

```
<?php
$db = new DataAccess();
$db->selectDb("db_blog");
$sql = "select * from tbl_posts where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

而采用水平分区后，刚才的 URL 显然不能正常工作了，因为数据分散在两台数据库服务器上，应用程序并不知道应该连接到哪一台服务器，我们必须告诉它，所以，我们提供了新的 URL：

```
http://www.highperfweb.com/blog_post.php?post_id=342352&user_id=10032
```

我们在 URL 中增加了 user_id，同时，修改后的数据访问代码如下所示：

```
<?php
$db = new DataAccess($user_id);
$db->selectDb("db_blog");
$sql = "select * from tbl_posts where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

可以看到，在创建数据访问实例时，我们传入了 user_id 变量，它将引导应用程序连接到正确的数据库服务器。

在以上这个分区的例子中，我们实际上是对数据按照 user_id%2 的不同结果进行划分，同样，我们也可以通过 user_id%10 将 tbl_posts 的记录划分为 10 个分区，如图 4-6 所示。

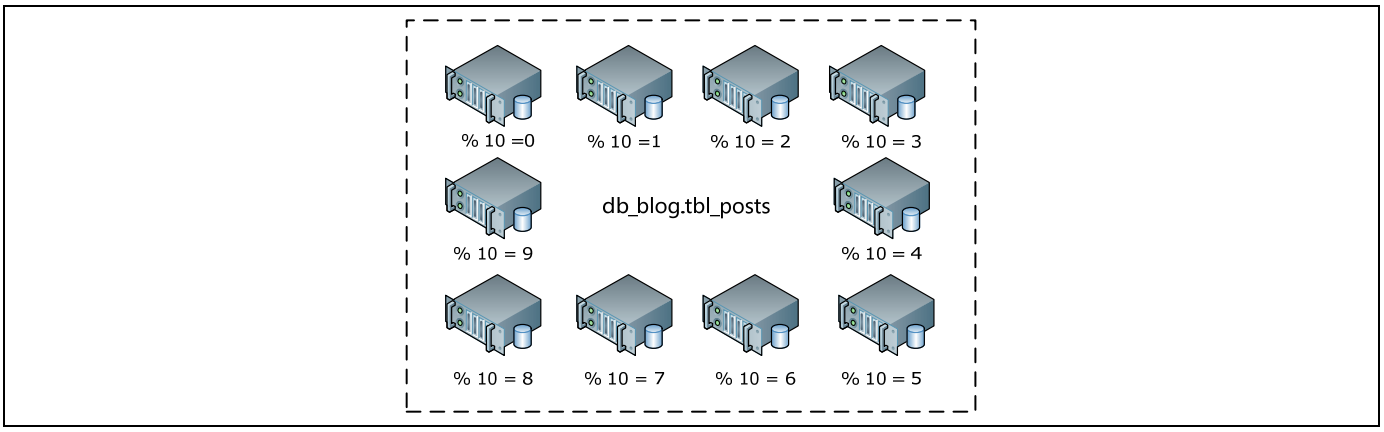


图 4-6　按照 user_id%10 进行水平分区

分区和分表

事实上，在考虑分区之前，我们一般会对数据库进行分表，它们的思路都是相同的，比如对于刚才的 tbl_posts 表，我们仍然按照 user_id%10 将它分为 10 个数据表：

```
tbl_posts_00
tbl_posts_01
tbl_posts_02
tbl_posts_03
tbl_posts_04
tbl_posts_05
tbl_posts_06
tbl_posts_07
tbl_posts_08
tbl_posts_09
```

这种分表的策略甚至会在数据库建立时便采用，因为我们希望数据表的记录数保持在相对较少的数量，这有利于减少查询时间，从而为数据库减少不必要的开销。

当然，分表只是单台数据库的优化策略，一旦到了必须考虑扩展的时候，分区便派上用场，不过，已经实现的分表将使得分区更加容易，因为数据已经是分离的，只需要迁移到其他服务器即可。

这时候，分表算法和分区算法可能不一致，比如我们希望将这 10 个表分布在两台服务器上，那么我们要在应用程序中维护一份映射关系表，比如将前 5 个数据表分配到一台数据库服务器，后 5 个数据表分配到另一台数据库服务器，这时的数据访问代码如下所示：

```
<?php
$db = new DataAccess($user_id);
$db->selectDb("db_blog");
$tbl_name = getTblName($user_id);
$sql = "select * from " . $tbl_name . " where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

如何分区

究竟如何分区呢？我们应该遵循哪些原则呢？

首先，你得考虑，应该对哪些数据进行分区，哪些数据可以放到分区中，比如用户信息、好友关系，它们是否适合进行分区

呢？其实，这个问题很难回答，但在大多数时候，对哪些数据进行分区并不是我们可以选择的，对于那些频繁访问而导致站点接近崩溃的热点数据，我们没有理由不对它们考虑分区，甚至在一开始就要考虑。

当然，在我们不得不对一些数据进行分区的时候，也意味着我们将失去一些操作它们的能力，比如原本你可以通过一条联合查询语句就能轻松搞定的任务，在分区后，你必须先通过用户 ID 找到正确的分区，然后查到对应的好友 ID，再通过好友 ID 找到对应的分区，从而找到好友信息。但是，你也不必感到沮丧，曾经的联合查询将会随着站点规则的增大变得越来越昂贵，而分区后的查询方式将会更加容易保持相对稳定的开销。

一旦我们知道要对哪些数据实施分区后，接下来就得找到一个用于分区的字段，我们称为分区索引字段，比如前面的 user_id，它必须和所有的记录都存在关系，一般我们会用被分区数据的主键或者外键。当使用主键时，你得保证它不能使用 auto_increment 自增类型。这一步非常简单，没有一概而论的方法，你要做的是抛开技术，认真地想一想你要什么，不久你便会得到结果。

接下来，基于这个字段，你得考虑采用什么分区算法，我们希望它可以带来良好的可扩展性，并且让各个分区的工作量相对均衡，通常有以下几种常用的分区算法。

哈希算法

刚才我们通过 user_id%10 来实现分区便是这种算法，它非常容易实现，而且应用程序通过 user_id 找到分区只需要进行简单的计算，几乎不存在额外开销。

相对于其他算法，哈希算法可以为多个分区比较均衡地分配工作量，特别是当记录数量级较多时，各个分区更加趋近于均衡。但是，这种算法对于扩展并不友好，一旦我们需要从 10 个分区扩展到 20 个分区，这便涉及所有数据的重新分区，你不得不暂停站点，等待漫长的计算。

范围

这种算法是指按照分区索引字段的范围进行分区，比如我们可以将 user_id 为 1~10000 的记录存储在一个分区中，而将 10001~20000 的用户存储在另一个分区中，以此类推。这使得应用程序需要维护一个简单的范围映射表，比如根据 user_id 来计算所属分区。

显然，它可以带来很好的扩展性，随着用户数量的不断增长，我们可以创建更多的分区。但是，各个分区的工作量会存在较大的差异，比如老用户所在的分区压力相对较大，或者一部分 ID 比较接近的热点用户导致所在分区压力过大。

映射关系

这种算法将对分区索引字段的每个可能的结果创建一个分区映射关系，这个映射关系将会非常庞大，应用程序已经无法通过简单的逻辑或者配置文件来维护它，而需要将它也写入数据库，比如当应用程序需要知道 user_id 为 10 的用户的博客内容在哪个分区时，它必须查询数据库获得答案，当然，我们会使用缓存来提高性能。

由于这种方式详细保存了每一个记录的分区对应关系，所以各个分区具有较强的可伸缩性，我们可以灵活地控制它们的规模，并且轻松地将数据从一个分区迁移到另一个分区，这也使得各个分区可以通过灵活的动态调节来保持平衡。

分区扩展

对于刚才提到的几种分区算法，可扩展性是我们比较关心的，从整体上看，能否很好地扩展意味着是否可以保持性能或者带来更好的性能。

在这里，分区的可扩展性更多体现在能否快速平滑地实现扩展，并且进行最少单位的数据移动。

通过 Web 负载均衡的体验，我们发现，计算能力是一种内在的、无形的力量，它的扩展本身几乎不需要时间，而数据库水平分区的扩展则意味着必要的数据迁移，以及重建平衡，这种流动带来了时间开销。

的确，不论采用哪种分区策略，增加分区都是一件不小的事情，大多数站点都会选择在夜深人静的时候挂上暂停维护的公告，当然，一旦持续时间较长，用户便会开始抱怨。

不幸的是，没有太多通用的方法帮助你很好地解决分区扩展，因为没有人知道你的站点拥有什么样的数据，并且将会以何种速度成长，但我们可以做的是，根据站点的实际情况，选择适合的分区策略，并且尽早地制定合理的扩展规划。

分区反向代理

还记得前面提到的 MySQL Proxy 吗？它帮助应用程序实现了读写分离，而在这里，另一个开源产品 Spock Proxy 也起到了类似的作用，它可以帮助应用程序实现水平分区的访问调度，这意味着我们不需要在应用程序中维护那些分区对应关系了。

Spock Proxy 本身的大部分代码正是基于 MySQL Proxy，同时它也进行了一些改进，在这里，Spock Proxy 的作用如图 4-7 所示。关于它的详细介绍，你可以查看在线文档。

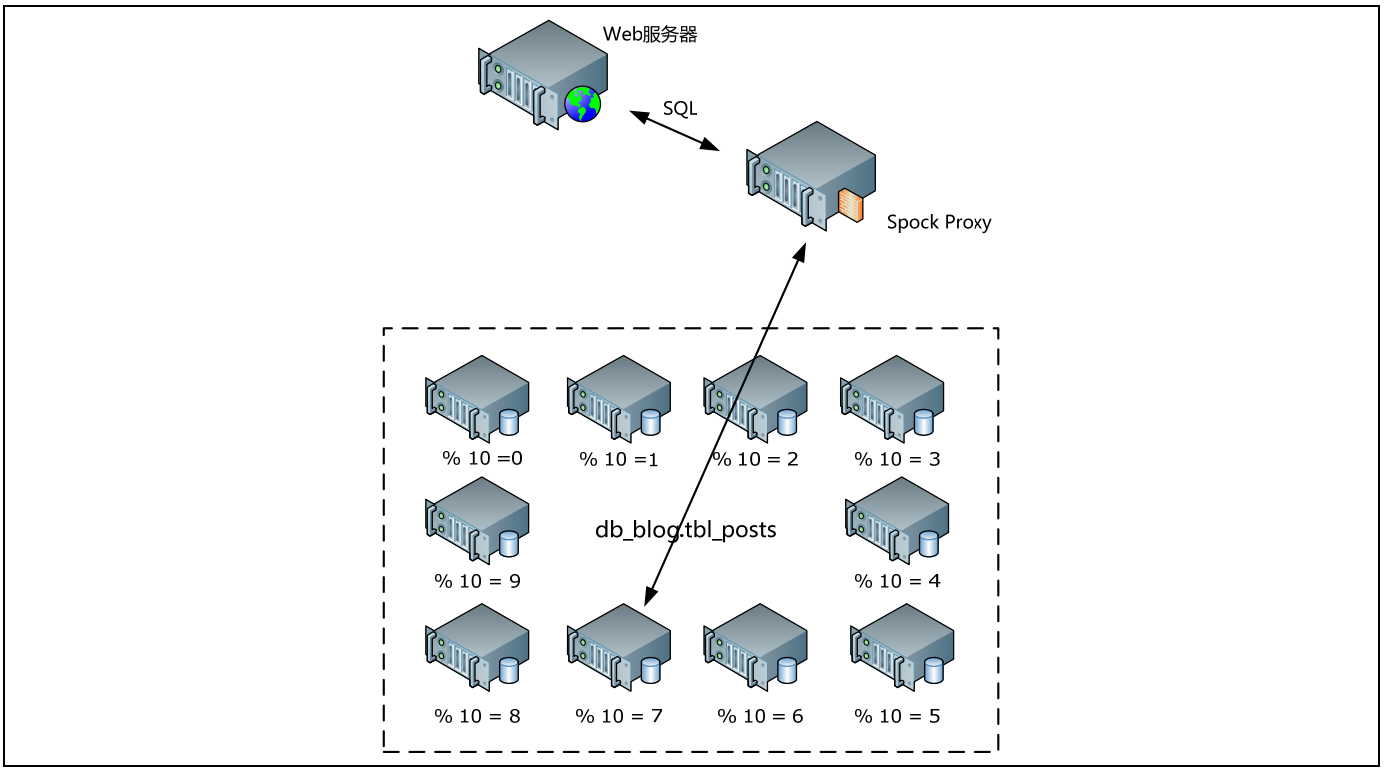


图 4-7　通过 Spock Proxy 访问多个 MySQL 水平分区

第 5 章

性能监控

我们需要通过监控的手段来洞察站点性能的变化，它会带给你优化的理由，也会告诉你瓶颈的真相，更重要的是，它让你拥有敏锐的直觉，没有它，你可能根本无法知道你的站点是否健康，因为你不可能每天都去费力地阅读日志。

没有人会认为站点的性能一成不变，的确，它时时刻刻都在变化，各种各样的原因会导致站点不能保持我们预期的性能，不论我们做了多少努力，你总是不知道下一刻会发生什么，性能监控在某种程度上就像是晴雨表，它能反映一定的性能变化规律和趋势，所以，能够快速从监控数据和图表中找到线索是你必须具备的本领。

事实上，在前面的章节中我们已经或多或少地用到了一些监控数据和图表，可见，这些内容和性能分析是不可分割的，这一章我们将主要介绍一些提供性能监控的工具和系统，你可以通过它们快速搭建监控中心。

提示：

运筹帷幄之中，决胜于千里之外。

——刘邦

5.1 实时监控

Nmon 是一款工作在服务器本地的实时监控软件，它可以提供时间间隔为秒的系统监控，我们来看它的监控界面，如图 5-1 所示。

这还只是 Nmon 所能监控的一部分，它还可以监控内核状态、系统负载、虚拟内存、NFS 等，只要有足够大的显示器，你可以把它们都添加到主界面中，服务器的一切活动尽收眼底，你可以在最短的时间内知道服务器现在在忙什么。

除此之外，你还可以用 Nmon 来录制数据，并通过另一个分析工具 Nmon Analyser 生成监控统计报告，如图 5-2 及图 5-3 所示，我们生成了间隔为 1 秒的监控报告，其中包括 CPU 使用率和磁盘 I/O。

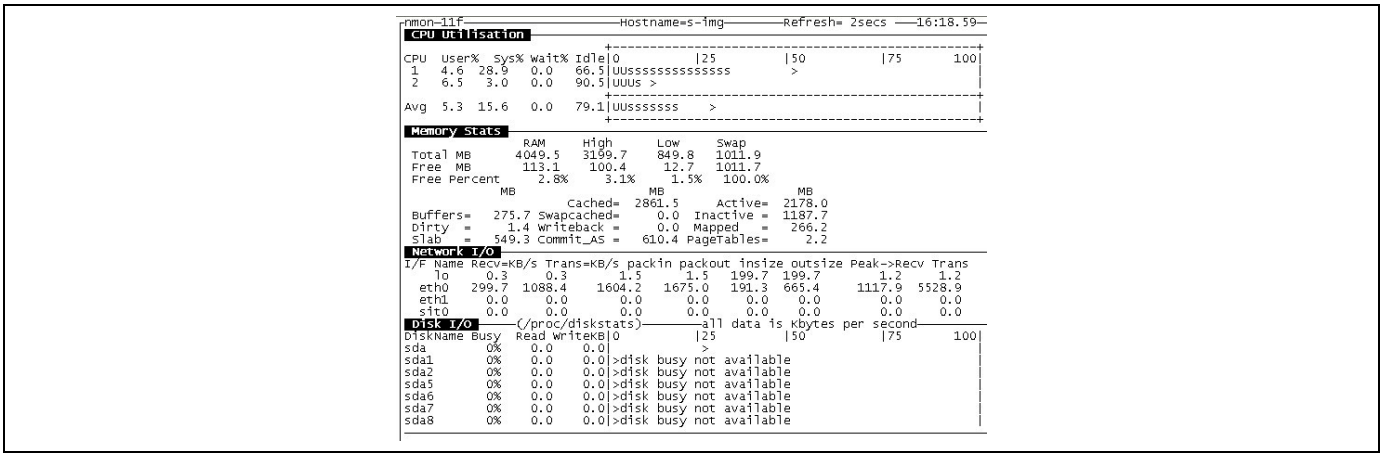
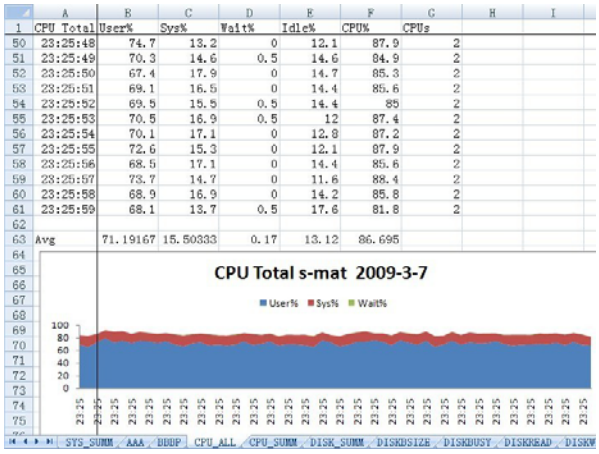


图 5-1 通过 Nmon 来进行系统监控

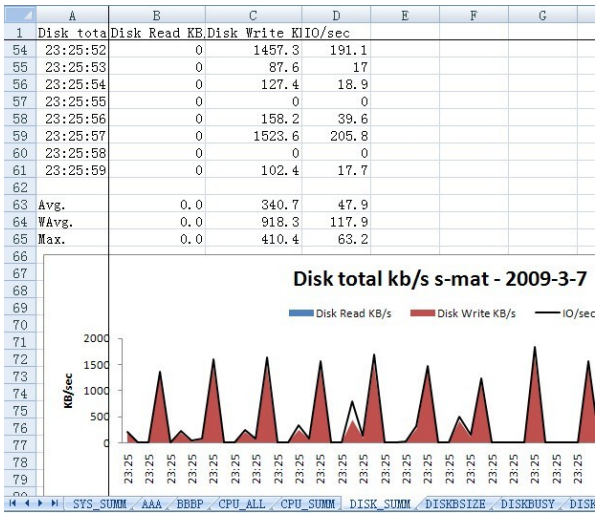


CPU Total	User%	Sys%	Wait%	Idle%	CPU%	CPUs
23:25:48	74.7	13.2	0	12.1	87.9	2
23:25:49	70.3	14.6	0.5	14.6	84.9	2
23:25:50	67.4	17.9	0	14.7	85.3	2
23:25:51	69.1	16.5	0	14.4	85.6	2
23:25:52	69.5	15.5	0.5	14.4	85	2
23:25:53	70.5	16.9	0.5	12	87.4	2
23:25:54	70.1	17.1	0	12.8	87.2	2
23:25:55	72.6	15.3	0	12.1	87.9	2
23:25:56	68.5	17.1	0	14.4	85.6	2
23:25:57	73.7	14.7	0	11.6	88.4	2
23:25:58	68.9	16.9	0	14.2	85.8	2
23:25:59	68.1	13.7	0.5	17.6	81.8	2
Avg	71.19167	15.50333	0.17	13.12	86.695	

图 5-2　通过 Nmon Analyser 获得 CPU 使用率统计报表

通常，我们只是在一些必要的时候进行实时监控，主要包括：

- 快速查看系统某组件的状态，比如进行网络结构调整后需要快速知道当前时刻的网络流量变化，而通过 MRTG 等方法你可能需要等待几分钟后的图表更新。
- 观察一些底层的系统状态，比如内核切换、进程队列、中断次数等，这些状态通常无法通过其他监控系统获得。
- 根据最小时间间隔的状态变化来进行一些诊断，比如通常的监控系统会将 5 分钟内的状态进行平均计算，那么 5 分钟内的变化情况我们并不知道，也许对于一些系统组件来说，这 5 分钟内的变化曲线隐藏着重要的线索。
- 你喜欢快节奏的工作方式，喜欢看到屏幕快速刷新。



Disk tota	Disk Read KB	Disk Write K	IO/sec
23:25:52	0	1457.3	191.1
23:25:53	0	87.6	17
23:25:54	0	127.4	18.9
23:25:55	0	0	0
23:25:56	0	158.2	39.6
23:25:57	0	1523.6	205.8
23:25:58	0	0	0
23:25:59	0	102.4	17.7
Avg.	0.0	340.7	47.9
WAvg.	0.0	918.3	117.9
Max.	0.0	410.4	63.2

图 5-3　通过 Nmon Analyser 获得磁盘 I/O 统计报表

除此之外，你可能并不需要经常进行实时监控，毕竟这需要你通过 SSH 登录到服务器上进行操作，而如果你要兼顾很多台服务器的话，这样做显然很不现实，其实，大多数时候，我们更希望通过远程来对服务器实施监控，并且建立统一入口的监控中心，同时，也可以放慢监控的节奏，因为过于实时的监控对服务器本身来说也存在一定的开销。

5.2　监控代理

一旦我们需要通过远程来监控服务器，那么监控代理程序便必不可少，它负责将服务器本地的各种状态定期上报给监控中心，或者响应来自监控中心的请求，这使得远程监控成为可能。

你可以为站点服务器开发专用的监控代理程序，并从本地文件中获取状态数据，在 Linux 中，一切都是文件，当然也包括各种系统状态，比如我们可以在这里看到详细的内存使用情况：

```
s-200:~ # cat /proc/meminfo
MemTotal:            4142240 kB
MemFree:             344428 kB
Buffers:             598264 kB
Cached:              2067336 kB
SwapCached:          4 kB
Active:              2598232 kB
Inactive:            1024660 kB
HighTotal:           3272000 kB
HighFree:            333912 kB
LowTotal:            870240 kB
LowFree:             10516 kB
SwapTotal:           1196800 kB
SwapFree:            1196672 kB
Dirty:               500 kB
Writeback:           0 kB
Mapped:              1086088 kB
Slab:                146180 kB
CommitLimit:         3267920 kB
Committed_AS:        6955560 kB
PageTables:          11908 kB
VmallocTotal:        112632 kB
VmallocUsed:         17640 kB
VmallocChunk:        94496 kB
HugePages_Total:     0
HugePages_Free:      0
HugePages_Rsvd:      0
Hugepagesize:        2048 kB
```

也可以看到系统负载和进程队列状态：

```
s-200:~ # cat /proc/loadavg
0.02 0.03 0.00 1/894 21407
```

这样一来，你只需要将这些数据打包，按照你定义的格式上报给监控中心即可。

不过，如果只是监控这些系统状态，你完全可以利用 SNMP 来完成这些工作，SNMP 服务器端本身便是一个出色的监控代理程序，它已经逐渐成为标准，并且支持很多异构平台。

比如，我们通过 SNMP 来获取另一台服务器的所有设备状态，如下所示：

```
s-200:~ # snmpwalk -c public  -v 2c 10.0.1.201
SNMPv2-MIB::sysDescr.0 = STRING: Linux s-mat 2.6.16.21-0.8-bigsmp #1 SMP Mon Jul 3 18:25:39 UTC 2006
i686
SNMPv2-MIB::sysObjectID.0 = OID: NET-SNMP-MIB::netSnmpAgentOIDs.10
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (68135977) 7 days, 21:15:59.77
SNMPv2-MIB::sysContact.0 = STRING: Sysadmin (root@localhost)
SNMPv2-MIB::sysName.0 = STRING: s-mat
SNMPv2-MIB::sysLocation.0 = STRING: Server Room
SNMPv2-MIB::sysORLastChange.0 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORID.1 = OID: SNMPv2-MIB::snmpMIB
SNMPv2-MIB::sysORID.2 = OID: TCP-MIB::tcpMIB
SNMPv2-MIB::sysORID.3 = OID: IP-MIB::ip
SNMPv2-MIB::sysORID.4 = OID: UDP-MIB::udpMIB
SNMPv2-MIB::sysORID.5 = OID: SNMP-VIEW-BASED-ACM-MIB::vacmBasicGroup
SNMPv2-MIB::sysORID.6 = OID: SNMP-FRAMEWORK-MIB::snmpFrameworkMIBComp liance
......
```

这里仅仅列出了其中很少的一部分，事实上整个结果集非常庞大，一般我们要限定需要获取状态的设备名称，可以通过 MIB 来描述它，比如这里我们要获取一台 Windows 服务器的运行时间，如下所示：

```
s-200:~ # snmpwalk -c public  -v 2c 10.0.1.210 hrSystemUptime
HOST-RESOURCES-MIB::hrSystemUptime.0 = Timeticks: (630851406) 73 days, 0:21:54.06
```

包括 MRTG、Cacti 以及 Nagios 在内的很多监控工具都利用 SNMP 来监控远程服务器，而你只需要在被监控的服务器上开启 SNMP 服务，同时对监控来源进行授权配置即可。

当然，SNMP 并不是万能的，有一些我们希望监控的服务并没有提供相应的 SNMP 支持，比如我们要监控 Nginx 服务器当前的 HTTP 并发连接数，该怎么做呢？是否需要自己开发监控代理程序呢？幸运的是，Nginx 提供了必要的 HTTP 监控接口，你可以直接在监控中心请求它即可，比如我们通过请求以下 URL：

```
http://10.0.1.200/status
```

便可以看到 Nginx 当前的运行状态：

```
Active connections: 3020
server accepts handled requests
 5440803 5440803 7336362
Reading: 471 Writing: 2340 Waiting: 209
```

总之，通过监控代理程序，我们可以更加轻松地了解服务器的各种状态，所以，不论你希望监控服务器的何种状态，只需要考虑开发相应的监控代理程序即可，你甚至可以监控 CPU 的温度，前提是你的内核能够支持。

5.3 系统监控

通常情况下，我们通过 SNMP 便可以很容易地对服务器进行一些常规的系统监控，这包括 CPU 使用率、系统负载、内存使用率、网络 I/O、磁盘 I/O、磁盘使用率等，这些是我们比较关心的。当然，我们还需要建立监控中心，对这些状态数据进行统计和呈现。幸运的是，有很多开源产品可以帮助我们，这里我们主要以 Cacti 为例，它完全可以支持刚刚提到的这些系统监控，并且绘制出相应的图表，便于我们浏览。

Cacti 采用 RRDtool 作为监控数据的存储引擎，它是一种专门针对绘制坐标图而设计的存储格式，相对于其他存储结构来说要节省很多存储空间，这为我们长期监控大量服务器提供了可能。相对于本书的主题来说，我们不打算花大量的篇幅从运维工作的角度来介绍监控系统本身的使用，你可以查阅大量相关的在线文档。

对于刚刚提到的 CPU 使用率等状态监控，其数据和图表背后的含义，我们在前面的章节中已经有过详细介绍，所以，接下来我们只会对图表本身进行简单的介绍，当你实际浏览这些监控图表时，应该充分结合前面的内容以及站点的实际情况来具体分析它们的含义。

作为服务器的心脏，CPU 的使用率是我们非常关注的一项系统状态，如图 5-4 所示，这是一台运行着 MySQL 的专用数据库服务器，图上清晰地呈现出 CPU 在用户空间和内核空间的时间开销，可以看得出，用户空间进程花费了大量的 CPU 时间，这正符合数据库应用的特点，如果希望更加深入地了解数据库内部的具体开销，你同样可以对 MySQL 进行监控，随后我们会介绍这方面的内容。

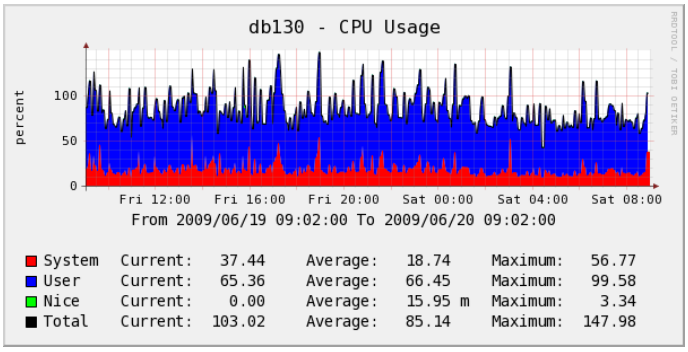


图 5-4 CPU 使用率监控

另外，不难看出，这里的图表采用了常见的累积图方式，它的好处是可以直接呈现出累加结果，比如这里的累加结果便是 CPU 使用率总和。

通过前面的介绍，我们已经了解了系统负载的意义，它的监控图表如图 5-5 所示，我们可以看到 1 分钟、5 分钟、15 分钟的平均负载变化情况，但是，它们的累加结果没有太多实际意义。

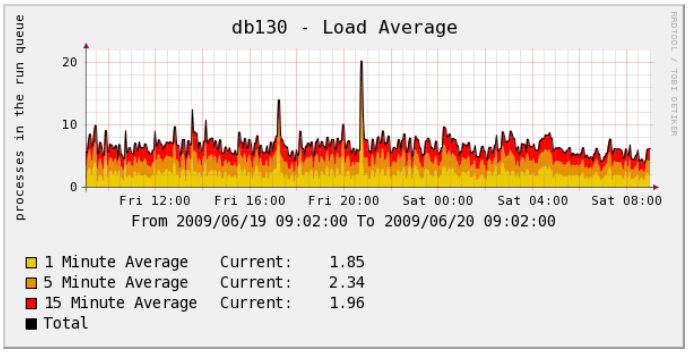


图 5-5　系统负载监控

如图 5-6 所示，这是一台运行着 Apache 的 Web 服务器，可以看到它的内存使用率并不饱和，事实上，它的 httpd 进程并不多，所以并没有消耗太多的物理内存。

而对于以下的 MySQL 服务器，我们为它最大程度地分配了物理内存作为 Innodb 缓存，可以看到内存使用率已经接近饱和，如图 5-7 所示。

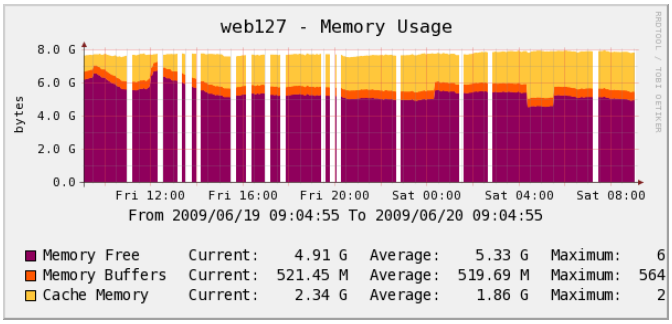


图 5-6　Web 服务器的内存使用率监控

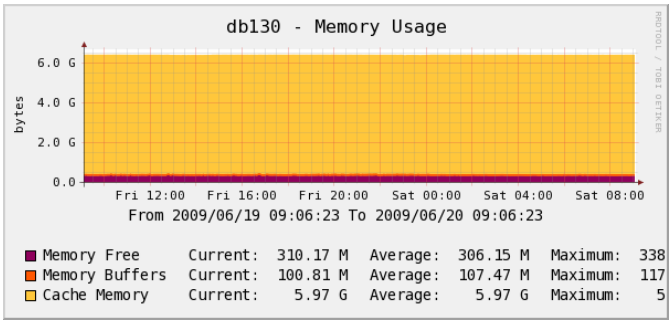


图 5-7　数据库服务器的内存使用率监控

在前面的章节中我们多次提到磁盘 I/O，那么，通过磁盘 I/O 监控，我们可以了解磁盘是否繁忙，包括读操作和写操作，如图 5-8 所示。

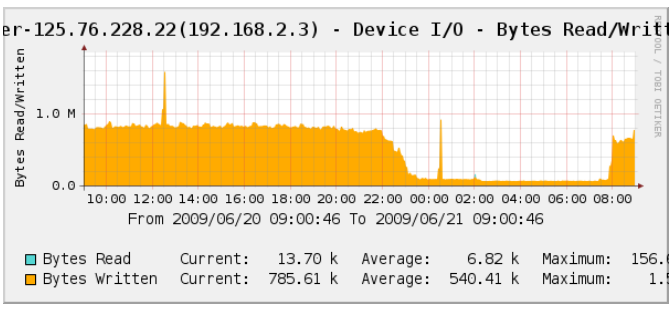


图 5-8　磁盘 I/O 监控

磁盘空间也是我们关注的另一个方面，通过磁盘使用率监控，我们可以更好地了解磁盘空间的使用情况和变化趋势，如图 5-9 所示。

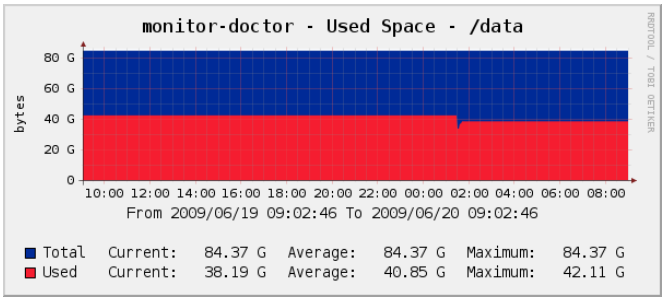


图 5-9　磁盘使用率监控

网络流量监控可以帮助我们了解服务器的通信数据量，包括流入和流出两部分，如图 5-10 所示，对于提供 HTTP 服务的服务器来说，流出的数据量要大于流入的数据量，因为 HTTP 响应数据长度通常大于请求数据长度。另一方面，对于流量情况的了解，可以帮我们更好地把握带宽的使用情况。

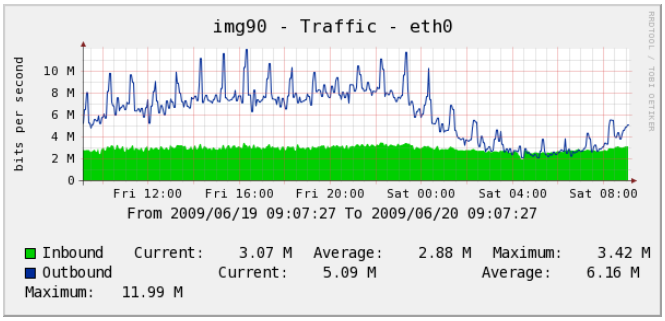


图 5-10　网络 I/O 监控

5.4　服务监控

除了基本的系统监控之外，我们还非常关心应用层服务的状态，它们更加直接地反映了 Web 应用的性能。Cacti 中支持插件机制，我们可以通过一些第三方模板来监控这些服务，同时，我们还需要相应的监控代理，幸运的是，一些常用的服务已经考虑到这一点，它们提供了一定的监控访问接口，比如 Apache 中基于 HTTP 的 mod_status 模块，在 httpd.conf 中配置如下：

```
ExtendedStatus On
<Location /server-status>
    SetHandler server-status
    Order deny,allow
    Deny from all
    Allow from all
</Location>
```

这样一来，我们便可以通过/server-status 的 URL 来查看 Apache 的当前状态，这相当于 Apache 的监控代理，有了它以后，我们在 Cacti 中通过相应的第三方模板，生成了如图 5-11 所示的监控图表。

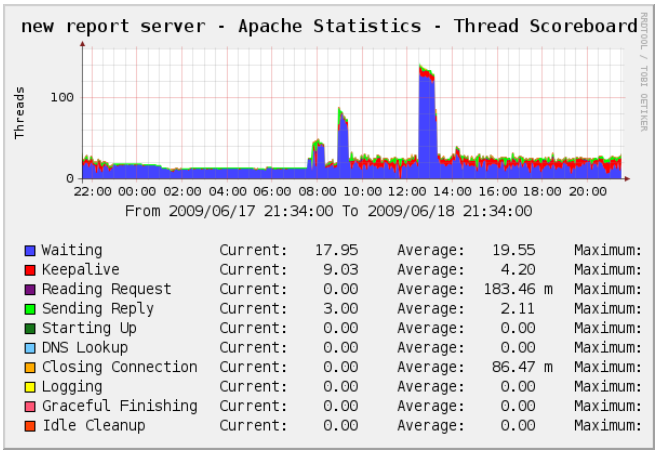


图 5-11　Apache 状态监控

同样，Lighttpd 和 Nginx 也可以按照类似的方式进行状态监控，这里就不再详细介绍。

除了 Web 服务器之外，我们还可能需要对反向代理服务器进行监控，在前面介绍 Varnish 的时候，我们已经介绍了它的监控图表，我们比较关心的是它的缓存命中率等参数。

另外，你或许也需要对 Memcached 进行监控，在介绍分布式缓存的时候我们曾经介绍过这方面的内容，包括 I/O 数据量、缓存命中率、空间使用率等。

同样，在数据库性能优化的章节中，我们了解了数据库的性能对于站点应用来说何等重要，所以，数据库的监控必不可少，我们在 Cacti 中可以对 MySQL 进行各种监控，如图 5-12~图 5-14 所示。

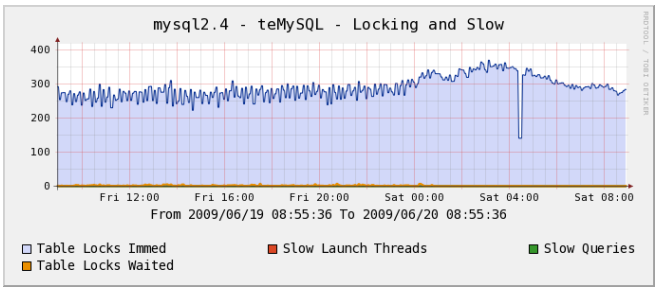


图 5-12　MySQL 的锁定和慢查询监控

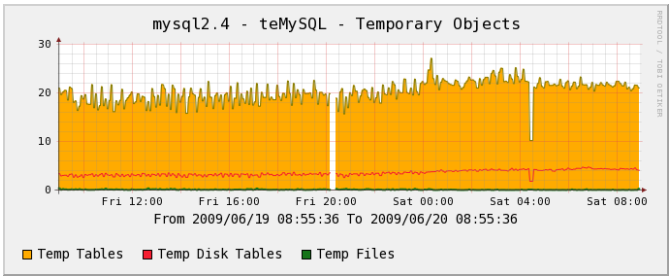


图 5-13　MySQL 的临时表监控

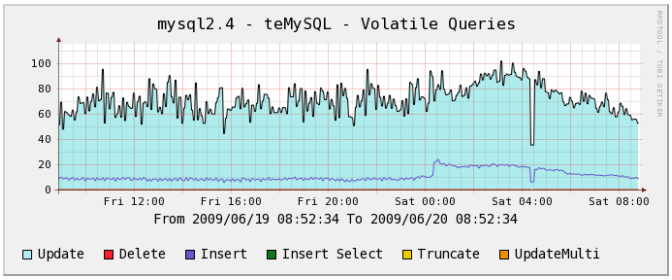


图 5-14　MySQL 的查询统计监控

这几项监控图表所表达的意义，我们同样在前面的章节中有过详细介绍，通过监控，你能够更加自如地运用各种优化策略。

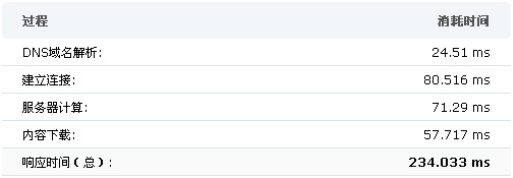
5.5　响应时间监控

的确，以上这些监控措施可以帮助我们直观地了解系统和服务本身的运行状况，但是，从另一个角度来看，位于网络另一端的最终用户只关心响应时间，你知道它们的真实体验吗？我很想知道。同时，用户访问多种不同的 Web 应用，比如站点首页、购物车、查找好友等，它们执行不同的商业逻辑。当然，我们希望了解这些不同 Web 应用的真实服务品质，尽管它们可能都运行在一个 Web 服务器上，但是我们对这些应用的性能要求并不相同，显然，通过以上这些监控手段已经力不从心。

所以，这时候我们需要借助一些第三方工具，这里推荐监控宝（www.jiankongbao.com），它像 Cacti 一样提供基于 Web 的服务界面，但你不用花时间去安装和部署，只需要注册和添加监控任务便可以快速使用。

由于监控宝拥有独立的监控点，所有它可以定时模拟用户来请求你指定的多个 URL，并且记录详细的响应时间，如图 5-15~图 5-17 所示。

图 5-15　HTTP 请求结果快照



过程	消耗时间
DNS域名解析:	24.51 ms
建立连接:	80.516 ms
服务器计算:	71.29 ms
内容下载:	57.717 ms
响应时间（总）:	**234.033 ms**

图 5-16　HTTP 请求过程的详细时间统计表

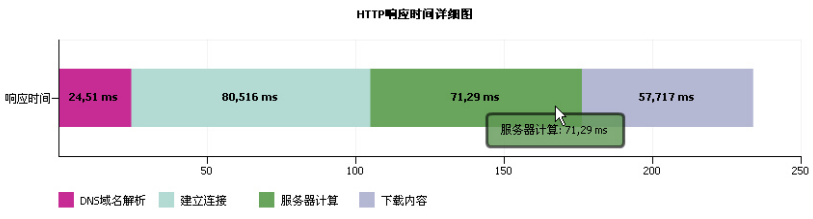


图 5-17　HTTP 请求过程的详细时间展示图

可以看到，在这一系列时间中，DNS 域名解析所消耗的时间实际上取决于 DNS 服务器的性能以及用户到 DNS 服务器的网络状况。除此之外，DNS 记录还可能会缓存在互联网接入服务商的各级 DNS 服务器上，这取决于 DNS 记录的 TTL 值。

建立连接的时间体现了 Web 服务器能否快速地接入用户的请求。在通常情况下，当 Web 服务器的同时连接数达到预设限制时，Web 服务器可能会拒绝新的接入请求，而对于 Apache 这样的多进程模型，当进程数不断增多时，由于上下文切换的时间开销也随之增加，所以建立连接的平均时间也逐渐开始延长。

接下来的服务器计算时间很容易理解，对于静态文件的访问，这部分时间主要用于文件的定位，如果是较小的文件，那么还会包括文件读取时间；而如果是较大的文件，前面章节曾提到过 Web 服务器会使用 sendfile 来直接传送文件内容到网络设备，所以读取文件的时间并没有算入这里的服务器计算时间，而是归入内容下载时间。

对于动态内容的访问，这里的服务器计算时间具有非常重要的参考价值，这些时间意味着什么，你一定比我更加清楚。事实上，这本书的大部分篇幅都在介绍如何减少这部分时间。

最后的内容下载时间从根本上来说取决于用户和服务器两端的带宽。如果你希望减少这部分时间，在前面关于数据网络传输的章节中你会找到部分答案。

另外，监控宝还提供了报告视图，你可以查看一段时间范围的响应时间变化曲线，如图 5-18 所示。

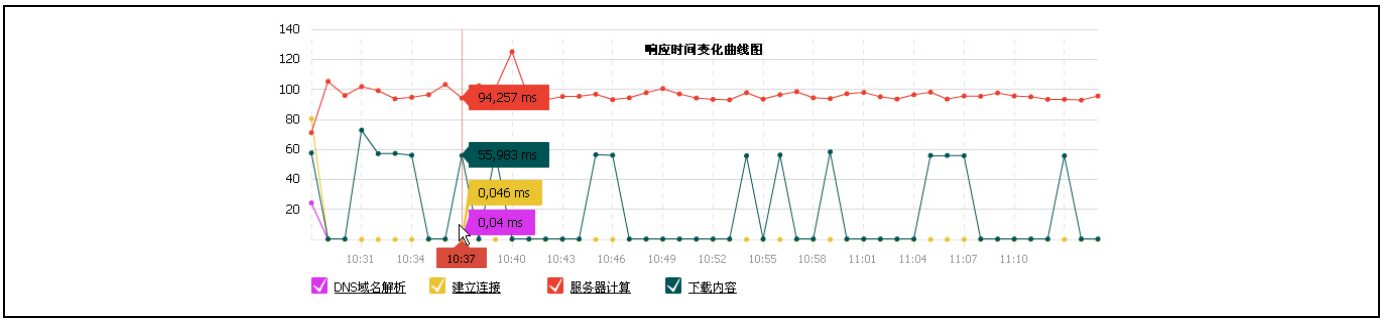
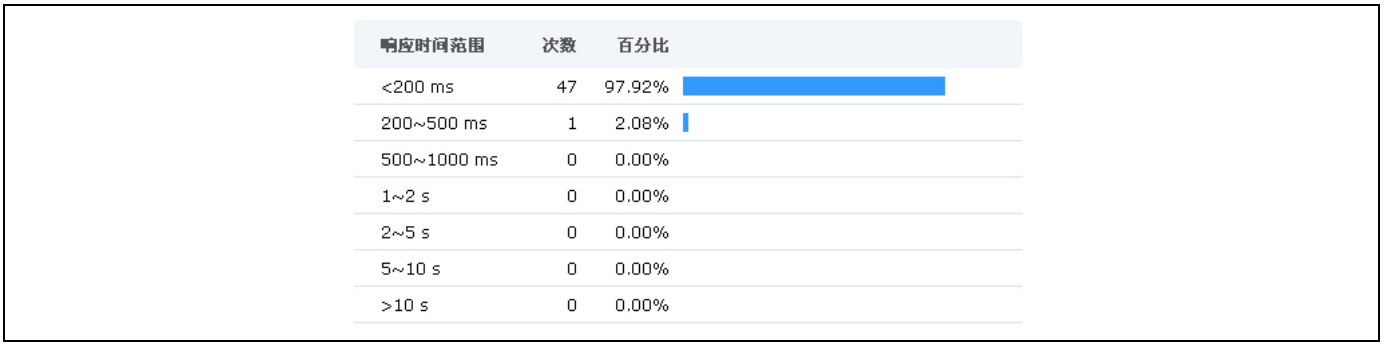


图 5-18　某时间范围的响应时间变化曲线

同时，还可以按照响应时间范围的次数和百分比进行统计，你可以直观地了解服务质量，如图 5-19 所示。



响应时间范围	次数	百分比
<200 ms	47	97.92%
200~500 ms	1	2.08%
500~1000 ms	0	0.00%
1~2 s	0	0.00%
2~5 s	0	0.00%
5~10 s	0	0.00%
>10 s	0	0.00%

图 5-19　响应时间范围的次数和百分比统计

当然，可用性统计也必不可少，如图 5-20 所示。

除此之外，监控宝还提供了一些其他的工具（如故障报警），并且支持 HTTP 以外的其他协议（如 DNS 服务器的监控），这里就不详细介绍了。

图 5-20　可用性统计报告

时刻关注企业软件开发领域的变化与创新

架构师

www.infoq.com/cn/architect

每月8号出版

InfoQ中文站
www.infoq.com/cn

商务合作：sales@cn.infoq.com
读者反馈/投稿：editors@cn.infoq.com