大型网站技术架构

核心原理与案例分析

李智慧 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

好评袭来

这是我看过的最接地气的一本介绍互联网架构的书籍，深入阐述了大型网站所面临的各种架构问题及解决方案，内容通俗易懂，而且对架构师的领导艺术进行了介绍，很值得从事互联网的技术人员学习和参考。

IBM 咨询经理　种新华

此书读来亲切，能用不到 300 页的篇幅将网站架构的过去及未来说得如此通俗易懂，与作者多年的亲身实践分不开，并由此想到一个问题：当此书人手一本的时候，阿里、腾讯、京东……的面试官们怎么办呢？

Oracle 资深工程师　付银海

智慧同学，人如其名，在阿里巴巴，人称“教授”，可见其博学多才。《大型网站技术架构：核心原理与案例分析》一书更是其多年积淀厚积薄发之作，涵盖构建大型互联网应用所需的关键技术，兼具实用性和前瞻性，无论是高并发、高性能还是海量数据处理、Web 前端架构，都有针对性的解决之道。尤其难得的是此书还对架构师的内涵及技术管理有比较深刻地阐述，实在是同类书籍中难得一见的。作为互联网应用的开发者、架构师和创业者的你，一定不要错过本书，本书足以解决你的技术之忧。

拓维信息平台研发总监　陈斌

教授（本书作者在阿里巴巴的昵称）曾在知名的大型互联网公司第一线浴血多年，经验不可谓不丰富，然而更难得的是他不仅博闻强记，更用行云流水的幽默文风，将现代大型互联网的内部要害一一庖解。也许各家细节略有不同，但大部分的大型互联网站基本都可以用这样的视角去解读。相信本书不仅对程序员，甚至对很多架构师也有参考价值，尤其值得关注的是教授在书中颇多技术之外的考量思索，我愿意称之为互联网基因。

堆糖网技术合伙人　曹文炯

有幸拜读了这本《大型网站技术架构：核心原理与案例分析》，本书从多个层面说明了如何构建一个高可用、高性能、高可扩展性的网站系统，并结合了阿里巴巴及其他互联网企业先进的架构实践经验进行案例分析，讲述非常全面且具指导意义。本书从网站的架构设计、快速开发、高效部署、业务监控、服务治理、运维管理等多个角度描述了架构设计的相关重点，涉及的核心技术包括前端优化、CDN、反向代理、缓存、消息队列、分布式存储、分布式服务、NoSQL 存储、搜索、监控、安全等一系列保证大型网站安全可靠运行的关键技术点。本书还提供了网站如何从小型网站伴随用户成长，逐步扩展到大型网站的架构演进思路，是互联网架构师们不可多得的一本技术参考书。

中兴通讯总工程师　钱煜明

设计和规划一个网站的总体架构涉及方方面面的东西，备选的方案也很多，如何在五花八门，纷繁复杂的技术中构建最适合用户的网站架构，变成了一件极具争议和挑战性的工作。一个好的架构可以以最低的成本，在满足用户需求的同时，满足整个网站的架构灵活性；同样，一个糟糕的架构可能会让你的客户在花费了大量金钱后，得到一堆笨重、复杂且不切实际的东西，或是由于系统过于复杂，故障不断，或是由于架构不够灵活，阻碍业务的发展等等。

回顾网站架构的发展历程，我们可以发现任何大型网站架构的发展都非一蹴而就的，同自然界生物物竞天择的自然进化规律一样，大型网站的架构发展和演变也基本遵循着类似的规律。我们可能无法想象几年后网站架构的样子，因为在互联网行业快速变化的当下，你甚至很难准确地预测未来一年网站的产品演变方向，甚至网站流量规模。于是，

产品设计师和工程师们提得最多的是迭代和演变，这在一个网站系统架构设计过程中显得尤为重要，因为我们永远无法像传统行业一样，去精确地估算，并按预先精确设计好的图纸去完成我们的产品。那是不是网站的架构设计和规划就毫无规律及章法可循了呢？答案显然不是，在互联网快速发展的今天，随着搜索引擎、电子商务、社交类等互联网产品逐步应用到每个人的身边，大型网站的架构及很多关键技术的发展，在逐步走向成熟。在构建一个大型网站过程中可能面临一些问题，人们正在尝试逐渐总结并积累出一些具有通用性的、经过验证的且成熟的局部解决方案，这也是本书将呈现给大家的内容。本书中，作者以自己多年大型互联网网站的架构经验，尝试总结当下这些互联网行业中相对成熟且经过大量案例检验的技术和方案。

相信通过阅读本书，您可以一窥大型网站架构的全貌。

阿里巴巴技术专家　余俊

循序渐进，娓娓道来，语言生动，举重若轻。

阿里云高级专家　李文兆

推荐序一

传统的企业应用系统主要面对的技术挑战是处理复杂凌乱、千变万化的所谓业务逻辑，而大型网站主要面对的技术挑战是处理超大量的用户访问和海量的数据处理；前者的挑战来自功能性需求，后者的挑战来自非功能性需求；功能性需求也许还有“人月神话”聊以自慰，通过增加人手解决问题，而非功能需求大多是实实在在的技术难题，无论有多少工程师，做不到就是做不到。IT 系统应用于企业管理已有超过半个世纪的历史，人们在这方面积累了大量的知识和经验（架构模式，领域分析，项目管理），而真正意义上大型网站从出现至今不过短短十多年的时间，很多技术挑战还在摸索阶段。市面上关于传统企业应用开发的书籍汗牛充栋，而真正能够深入全面地阐述大型网站技术架构的图书寥寥无几。所以很多人就很困惑：为什么很多看起来不是很复杂的网站，比如 Facebook、淘宝，都需要大量顶尖高手来开发呢？

值得庆幸的是，作者为我们带来了这本《大型网站技术架构：核心原理与案例分析》，比较全面地阐述了大型网站的主要技术挑战和解决方案。宏观层面上，将网站架构的演化发展、架构模式、核心要素一一道来；微观层面上，将网站架构常用的分布式缓存、负载均衡、消息队列、分布式服务、甚至网站如何发布运维都逐一进行了阐述。大型网站的技术之道尽在于此。

作者在阿里巴巴工作期间，一方面参与基础技术平台产品开发，一方面参与网站架构设计，这些经历使作者能够比较全面地从理论和实践两个视角去看待和描述网站架构。书中的技术内容基本都从为什么（Why）要这么做和如何去做（How）两个层面进行表述。

读者可知其然并知其所以然。

阅读本书也许不能使你就此掌握大型网站架构设计的屠龙之术，但至少使你对网站架构的方法和思维方式能有全面了解。

开卷有益，应该指的就是这样的书。

支付宝研究员　潘磊

推荐序二

这些年互联网技术蓬勃发展，各种成熟的组件、工具、框架越来越丰富，各种理论逐渐发展成熟，各大公司公开的理论和实践资料也越来越多，在各个领域都有比较成熟的解决方案，但是研究领先互联网公司的架构，无论是 Google、Facebook、Amazon 还是淘宝、支付宝、腾讯、百度，都各有其独特的地方。

各个环节都有成熟的产品或者方案，为什么这么多互联网公司的架构还有如此明显的差异呢？是不是照着 Google、Facebook、淘宝的架构做，就能做好一个“大型的互联网应用”呢？

正如本书中所言：“好的设计绝对不是模仿、不是生搬硬套某个模式，而是在对问题深刻理解之上的创造与创新，即使是‘微创新’，也是让人耳目一新的似曾相识。山寨与创新的最大区别不在于是否抄袭、是否模仿，而在于对问题和需求是否真正理解与把握。”

这些大型的互联网应用是设计出来的？还是演化出来的？在设计的过程中需要考虑哪些因素？演化过程中都会面临哪些问题，哪些挑战？

本书从性能、可用性、伸缩性、扩展性、安全性几个网站核心架构要素切入，全面地介绍了这些核心要素面临的问题域、理论基础及应对方案；对这几个方面进行系统地分析，结合目前成熟的解决方案，以及作者自己的工作经验，理论联系实际，踏实细致地提出合理的解决方案，非常值得我们学习和借鉴。

作者还通过对淘宝、Wikipedia、分布式存储系统、秒杀系统等案例的分析，仔细探讨了典型互联网架构的演进过程，剖析了分布式系统设计和实现中的挑战和解决方案，并研究了极端情况下，秒杀给网站带来的难以预计的瞬间高并发冲击的应对策略和架构设计。还通过一些实实在在发生过的故障案例分析，从另一个侧面来说明，我们在做技术架构时，需要考量的一些关键点，这些分享都是不可多得的血泪经验。

本书观点明确，涉及的问题域有针对性和全面性，对问题的分析过程清晰，提出的解决方案切实可行，充分结合了目前成功的互联网公司的架构经验，结合了作者丰富的工作经验，是一本值得行业内人士学习和关注的好书。

作者李智慧在互联网行业具有丰富的经验，在阿里巴巴工作的几年中担任架构师，参与过多个重要的项目和产品的架构设计，遇到和处理了很多复杂的问题，在这方面积累了大量的经验。本书是作者多年的架构师经历，以及时刻的思考和积累的结晶，一词一句都是经验之谈，都是智慧的闪亮。

感谢作者耗费精力给我们带来如此精炼而又内容丰富的一本好书。

支付宝资深架构师　王定乾

序

我为什么要写这本书

我想写一本关于网站架构方面的书源起于 2011 年年末至 2012 年年初发生的两件事。

2011 年末，京东网图书促销，在打 5 折的基础上再满一百送一百，作为一个爱买书胜过爱读书的人，我对这种促销活动根本没有免疫力，于是兴致勃勃地在活动当天登录 www.360buy.com，准备将收藏夹里的图书一网打尽。

往购物车里尽情地塞了一堆书后，点击“购买”按钮，但是浏览器迟迟没有响应，预感到京东的服务器可能因为并发访问量过高，超过了系统的最大负载能力，果然过了一会，浏览器页面显示“Service is too busy”。我不甘心，返回购物车页面继续点击“购买”按钮，浏览器继续显示“Service is too busy”。

于是我猜测：能够正常访问购物车，却不能成功购买，问题应该是出在订单系统，B2C 网站生成一个订单需要经历扣减库存、扣减促销资源、更新用户账户等一系列操作，这些操作大多是数据库事务操作，没有办法通过缓存等手段来减轻数据库服务器负载压力，如果事前没有设计好数据库伸缩性架构，那么京东的技术团队将遇到一个大麻烦。

当天晚上，我登录新浪微博，看到京东的大老板刘强东发布了一条微博：“我已经紧急采购了 10 台服务器，增强网站后台，明天继续促销一天，一定让大家买到书”。即使在有成熟数据库伸缩性架构设计的前提下，进行一次数据库扩容也是件棘手的事，而京东只需要一个晚上就能搞定，让我对京东的技术实力刮目相看。

第二天一上班，我的第一件事就是登录 www.360buy.com，点击“购买”按钮后悲剧地发现页面还是“Service is too busy”。当天晚上，刘强东又发布了一条微博：“请信息部的同事喝茶”。还配了一张照片：一张大桌子，只有一杯茶，旁边放了一把刀……

我想京东信息部的同事绝对不是有意要捉弄他们的老板和客户，很可能是他们错误地判断了系统的瓶颈及伸缩性架构的困难，对老板做出了过度承诺，而这背后折射出的是他们对网站架构的本质缺乏了解。

另一件事发生在 2012 年年初，当时的中国铁道部官方售票网站 www.12306.cn 在春运期间因为大量用户访问而崩溃，无法有效访问。12306 作为一个运营不久的网站，缺乏大规模并发访问处理的经验，遇到一些问题其实不奇怪，不管花多少钱，经验教训都需要经历时间和挫折才能得到。奇怪的是，12306 的架构师似乎对这种可能发生的大规模并发访问产生的问题完全没有一点概念，系统好像根本没有经过任何高并发场景下的性能评估和性能测试，就那么干脆利落地崩溃了，趴在那里长时间起不来。

这两件事情促使我想写一本关于网站架构的书，阐述网站技术架构最基本的驱动力，基础的架构设计原理，以及架构方案选择的价值观。希望软件工程师们在解决问题之前，能够认真思考自己面对的真正问题究竟是什么，有哪些技术方案可以选择，其基本原理是什么。所以这本书里没有高深的算法和聱牙诘屈的公式，也很少有程序代码。读者可以把本书当作网站架构设计的科普书，即使对网站架构没有什么了解，也能够比较轻松地阅读。

在本书的写作过程中（2012 年下半年），没有再看到京东促销宕机的新闻，12306 也逐渐稳定成熟。我们虽然无法猜测京东“信息部的同事”和 12306 网站的工程师们付出了多少努力，但能在相对比较短的时间里解决这些技术问题，也说明了网站架构其实并不难，真正能解决问题的技术一定是简单的。

本书致力于把这些简单的技术和道理呈现给读者。

如何阅读本书

我自己读书不求甚解，遇到看不懂的地方就跳过去，但是希望作者对难点和重点能换个角度和方式在后面章节再叙述，以帮助我重新思考和认识前面不能理解的重要知识。

机械制图的时候，通常使用三视图描述一个机械零件，从正视、侧视、俯视三个角度对一个零件绘图，从而全面描述一个零件的结构。软件架构设计中常用的 4+1 视图模型，也是一种多角度描述软件系统设计的手段。

本书中，重要的架构原理和技术方案都采用多角度描述的方法。

第 1 篇，从演化、模式、要素三个维度描述网站整体架构。

第 2 篇，从性能、可用性、伸缩性、扩展性、安全这五个要素方面详细描述网站架构核心原理，其中重要的负载均衡、异步处理、分布式缓存等技术方案又在不同章节从多角度进行描述。

第 3 篇，通过几个具体案例再一次从整体和局部描述网站架构方法。

第 4 篇，从架构师做事的角度回顾网站技术架构，读者在阅读前面技术章节感到枯燥的时候，也可以跳到本篇休闲放松下。

阅读本书过程中有任何问题和建议，请联系作者。新浪微博：**@大型网站技术架构**。

致 谢

2012 年五一节前夕，当我拜访博文视点的编辑胡辛征，商谈出版一本关于大型网站技术架构的图书时，没有想到自己面临的挑战是如此巨大。

整个图书写作过程就像喝醉了酒：头痛如裂，有很多话想说，但又不知该从何说起。

我很庆幸，这个过程有你们陪伴、支持、鼓励和帮助，是你们给了我继续前行的勇气。

感谢易普际的培训顾问周腾飞，策划并鼓励我去写这本书。

感谢阿里巴巴的技术专家余俊和何坤，这本书的大纲和结构就是和你们在钱塘江畔散步时聊出来的，但很遗憾最后没能成功蛊惑你们和我一起创作本书。

感谢阿里巴巴高级开发工程师熊红亮、丁夏珍；IBM 咨询经理种新华、架构师吴业勇；百度产品经理王晟；Intel 运维工程师 Liu Gongmin 给予的建议和鼓励。

感谢博文视点的编辑刘皎、郑柳洁，以及许多我不知道名字的编辑为本书最终出版付出的努力。

感谢阿里巴巴资深架构师潘磊、王定乾、钱霄、王齐，指引我进入网站架构的知识殿堂。

本书很多内容源自阿里同学们的知识库，原谅我无法一一致谢。

感谢我的妻子方芬香，你给予我一个新的世界，让我如此热爱生活。

目 录

第 1 篇 概述

1 大型网站架构演化 2

1.1 大型网站软件系统的特点 …… 3

1.2 大型网站架构演化发展历程 …… 4

1.2.1 初始阶段的网站架构 …… 4

1.2.2 应用服务和数据服务分离 …… 4

1.2.3 使用缓存改善网站性能 …… 5

1.2.4 使用应用服务器集群改善网站的并发处理能力 …… 6

1.2.5 数据库读写分离 …… 7

1.2.6 使用反向代理和 CDN 加速网站响应 …… 8

1.2.7 使用分布式文件系统和分布式数据库系统 …… 9

1.2.8 使用 NoSQL 和搜索引擎 …… 10

1.2.9 业务拆分 …… 11

1.2.10 分布式服务 …… 11

1.3 大型网站架构演化的价值观 …… 13

1.3.1 大型网站架构技术的核心价值是随网站所需灵活应对 …… 13

1.3.2 驱动大型网站技术发展的主要力量是网站的业务发展 …… 13
1.4 网站架构设计误区 …… 14
1.4.1 一味追随大公司的解决方案 …… 14
1.4.2 为了技术而技术 …… 14
1.4.3 企图用技术解决所有问题 …… 14
1.5 小结 …… 15

2 大型网站架构模式 16

2.1 网站架构模式 …… 16
2.1.1 分层 …… 17
2.1.2 分割 …… 18
2.1.3 分布式 …… 18
2.1.4 集群 …… 19
2.1.5 缓存 …… 20
2.1.6 异步 …… 20
2.1.7 冗余 …… 21
2.1.8 自动化 …… 22
2.1.9 安全 …… 23
2.2 架构模式在新浪微博的应用 …… 23
2.3 小结 …… 25

3 大型网站核心架构要素 26

3.1 性能 …… 27
3.2 可用性 …… 28
3.3 伸缩性 …… 29
3.4 扩展性 …… 30
3.5 安全性 …… 30

3.6 小结 …… 31

第 2 篇 架构

4 瞬时响应：网站的高性能架构 34

4.1 网站性能测试 …… 35
4.1.1 不同视角下的网站性能 …… 35
4.1.2 性能测试指标 …… 36
4.1.3 性能测试方法 …… 39
4.1.4 性能测试报告 …… 41
4.1.5 性能优化策略 …… 41
4.2 Web 前端性能优化 …… 42
4.2.1 浏览器访问优化 …… 42
4.2.2 CDN 加速 …… 43
4.2.3 反向代理 …… 44
4.3 应用服务器性能优化 …… 45
4.3.1 分布式缓存 …… 45
4.3.2 异步操作 …… 52
4.3.3 使用集群 …… 53
4.3.4 代码优化 …… 54
4.4 存储性能优化 …… 58
4.4.1 机械硬盘 vs. 固态硬盘 …… 58
4.4.2 B+树 vs. LSM 树 …… 59
4.4.3 RAID vs. HDFS …… 61
4.5 小结 …… 64

5 万无一失：网站的高可用架构 66

5.1 网站可用性的度量与考核 67
5.1.1 网站可用性度量 67
5.1.2 网站可用性考核 67
5.2 高可用的网站架构 69
5.3 高可用的应用 71
5.3.1 通过负载均衡进行无状态服务的失效转移 72
5.3.2 应用服务器集群的 Session 管理 73
5.4 高可用的服务 76
5.5 高可用的数据 78
5.5.1 CAP 原理 79
5.5.2 数据备份 82
5.5.3 失效转移 84
5.6 高可用网站的软件质量保证 85
5.6.1 网站发布 85
5.6.2 自动化测试 86
5.6.3 预发布验证 87
5.6.4 代码控制 88
5.6.5 自动化发布 90
5.6.6 灰度发布 91
5.7 网站运行监控 91
5.7.1 监控数据采集 92
5.7.2 监控管理 93
5.8 小结 94

6 永无止境：网站的伸缩性架构 95

6.1 网站架构的伸缩性设计 97
6.1.1 不同功能进行物理分离实现伸缩 97
6.1.2 单一功能通过集群规模实现伸缩 98
6.2 应用服务器集群的伸缩性设计 99
6.2.1 HTTP 重定向负载均衡 100
6.2.2 DNS 域名解析负载均衡 101
6.2.3 反向代理负载均衡 102
6.2.4 IP 负载均衡 103
6.2.5 数据链路层负载均衡 104
6.2.6 负载均衡算法 105
6.3 分布式缓存集群的伸缩性设计 106
6.3.1 Memcached 分布式缓存集群的访问模型 107
6.3.2 Memcached 分布式缓存集群的伸缩性挑战 107
6.3.3 分布式缓存的一致性 Hash 算法 109
6.4 数据存储服务器集群的伸缩性设计 112
6.4.1 关系数据库集群的伸缩性设计 113
6.4.2 NoSQL 数据库的伸缩性设计 117
6.5 小结 119

7 随需应变：网站的可扩展架构 121

7.1 构建可扩展的网站架构 122
7.2 利用分布式消息队列降低系统耦合性 123
7.2.1 事件驱动架构 123
7.2.2 分布式消息队列 124
7.3 利用分布式服务打造可复用的业务平台 126
7.3.1 Web Service 与企业级分布式服务 128

7.3.2 大型网站分布式服务的需求与特点 …… 129
7.3.3 分布式服务框架设计 …… 130
7.4 可扩展的数据结构 …… 131
7.5 利用开放平台建设网站生态圈 …… 132
7.6 小结 …… 134

8 固若金汤：网站的安全架构 135

8.1 道高一尺魔高一丈的网站应用攻击与防御 …… 136
8.1.1 XSS 攻击 …… 136
8.1.2 注入攻击 …… 138
8.1.3 CSRF 攻击 …… 139
8.1.4 其他攻击和漏洞 …… 140
8.1.5 Web 应用防火墙 …… 141
8.1.6 网站安全漏洞扫描 …… 142
8.2 信息加密技术及密钥安全管理 …… 142
8.2.1 单向散列加密 …… 143
8.2.2 对称加密 …… 144
8.2.3 非对称加密 …… 144
8.2.4 密钥安全管理 …… 145
8.3 信息过滤与反垃圾 …… 146
8.3.1 文本匹配 …… 147
8.3.2 分类算法 …… 148
8.3.3 黑名单 …… 149
8.4 电子商务风险控制 …… 150
8.4.1 风险 …… 151
8.4.2 风控 …… 151
8.5 小结 …… 153

第 3 篇　案例

9　淘宝网的架构演化案例分析　156

9.1　淘宝网的业务发展历程 …… 157
9.2　淘宝网技术架构演化 …… 158
9.3　小结 …… 162

10　维基百科的高性能架构设计分析　163

10.1　Wikipedia 网站整体架构 …… 163
10.2　Wikipedia 性能优化策略 …… 165
10.2.1　Wikipedia 前端性能优化 …… 165
10.2.2　Wikipedia 服务端性能优化 …… 166
10.2.3　Wikipedia 后端性能优化 …… 167

11　海量分布式存储系统 Doris 的高可用架构设计分析　169

11.1　分布式存储系统的高可用架构 …… 170
11.2　不同故障情况下的高可用解决方案 …… 171
11.2.1　分布式存储系统的故障分类 …… 172
11.2.2　正常情况下系统访问结构 …… 172
11.2.3　瞬时故障的高可用解决方案 …… 173
11.2.4　临时故障的高可用解决方案 …… 174
11.2.5　永久故障的高可用解决方案 …… 175

12　网购秒杀系统架构设计案例分析　176

12.1　秒杀活动的技术挑战 …… 177
12.2　秒杀系统的应对策略 …… 177
12.3　秒杀系统架构设计 …… 178

12.4 小结 …… 182

13 大型网站典型故障案例分析 183

13.1 写日志也会引发故障 …… 184
13.2 高并发访问数据库引发的故障 …… 184
13.3 高并发情况下锁引发的故障 …… 185
13.4 缓存引发的故障 …… 185
13.5 应用启动不同步引发的故障 …… 186
13.6 大文件读写独占磁盘引发的故障 …… 186
13.7 滥用生产环境引发的故障 …… 187
13.8 不规范的流程引发的故障 …… 187
13.9 不好的编程习惯引发的故障 …… 188
13.10 小结 …… 188

第 4 篇 架构师

14 架构师领导艺术 190

14.1 关注人而不是产品 …… 191
14.2 发掘人的优秀 …… 191
14.3 共享美好蓝图 …… 192
14.4 共同参与架构 …… 193
14.5 学会妥协 …… 194
14.6 成就他人 …… 194

15 网站架构师职场攻略 196

15.1 发现问题，寻找突破 …… 197
15.2 提出问题，寻求支持 …… 199

15.3 解决问题，达成绩效 …… 201

16 漫话网站架构师 203

16.1 按作用划分架构师 …… 203
16.2 按效果划分架构师 …… 204
16.3 按职责角色划分架构师 …… 205
16.4 按关注层次划分架构师 …… 205
16.5 按口碑划分架构师 …… 206
16.6 非主流方式划分架构师 …… 207

附录 A 大型网站架构技术一览 208

附录 B Web 开发技术发展历程 215

后记 218

第 1 篇

概　述

1 大型网站架构演化

如果把上世纪 90 年代初 CERN 正式发布 Web 标准和第一个 Web 服务的出现当做互联网站的开始，那么互联网站的发展只经历了短短 20 多年的时间。在 20 多年的时间里，互联网的世界发生了巨大变化，今天，全球有近一半的人口使用互联网，人们的生活因为互联网而产生了巨大改变。从信息检索到即时通信，从电子购物到文化娱乐，互联网渗透到生活的每个角落，而且这种趋势还在加速。因为互联网，我们的世界正变得越来越小。

同时我们也看到，在互联网跨越式发展的进程中，在电子商务火热的市场背后却是不堪重负的网站架构，某些 B2C 网站逢促销必宕机几乎成为一种规律，而铁道部电子客票官方购票网站的频繁故障和操作延迟更将这一现象演绎得淋漓尽致。

一边是企业在网站技术上的大量投入，一边却是网站在关键时刻的频繁宕机；一边是工程师夜以继日地加班工作，一边却是网站故障频发新功能上线缓慢；一边是互联网业务快速发展多领域挑战传统行业，一边却是网站安全漏洞频发让网民胆战心惊怨声载道。

如何打造一个高可用、高性能、易扩展、可伸缩且安全的网站？如何让网站随应用所需灵活变动，即使是山寨他人的产品，也可以山寨的更高、更快、更强，一年时间用户数从零过亿呢？

1.1 大型网站软件系统的特点

与传统企业应用系统相比，大型互联网应用系统有以下特点。

高并发，大流量：需要面对高并发用户，大流量访问。Google 日均 PV 数 35 亿，日均 IP 访问数 3 亿；腾讯 QQ 的最大在线用户数 1.4 亿（2011 年数据）；淘宝 2012 年“双十一”活动一天交易额超过 191 亿，活动开始第一分钟独立访问用户达 1000 万。

高可用：系统 7×24 小时不间断服务。大型互联网站的宕机事件通常会成为新闻焦点，例如 2010 年百度域名被黑客劫持导致不能访问，成为重大新闻热点。

海量数据：需要存储、管理海量数据，需要使用大量服务器。Facebook 每周上传的照片数目接近 10 亿，百度收录的网页数目有数百亿，Google 有近百万台服务器为全球用户提供服务。

用户分布广泛，网络情况复杂：许多大型互联网都是为全球用户提供服务的，用户分布范围广，各地网络情况千差万别。在国内，还有各个运营商网络互通难的问题。而中美光缆的数次故障，也让一些对国外用户依赖较大的网站不得不考虑在海外建立数据中心。

安全环境恶劣：由于互联网的开放性，使得互联网站更容易受到攻击，大型网站几乎每天都会被黑客攻击。2011 年国内多个重要网站泄露用户密码，让普通用户也直面一次互联网安全问题。

需求快速变更，发布频繁：和传统软件的版本发布频率不同，互联网产品为快速适应市场，满足用户需求，其产品发布频率是极高的。Office 的产品版本以年为单位发布，而一般大型网站的产品每周都有新版本发布上线，至于中小型网站的发布就更频繁了，有时候一天会发布几十次。

渐进式发展：与传统软件产品或企业应用系统一开始就规划好全部的功能和非功能需求不同，几乎所有的大型互联网站都是从一个小网站开始，渐进地发展起来的。Facebook 是伯克扎克同学在哈佛大学的宿舍里开发的；Google 的第一台服务器部署在斯坦福大学的实验室里；阿里巴巴则是在马云家的客厅里诞生的。好的互联网产品都是慢慢运营出

来的，不是一开始就开发好的，这也正好与网站架构的发展演化过程对应。

1.2 大型网站架构演化发展历程

大型网站的技术挑战主要来自于庞大的用户，高并发的访问和海量的数据，任何简单的业务一旦需要处理数以 P 计的数据和面对数以亿计的用户，问题就会变得很棘手。大型网站架构主要就是解决这类问题。

1.2.1 初始阶段的网站架构

大型网站都是从小型网站发展而来，网站架构也是一样，是从小型网站架构逐步演化而来。小型网站最开始时没有太多人访问，只需要一台服务器就绰绰有余，这时的网站架构如图 1.1 所示。

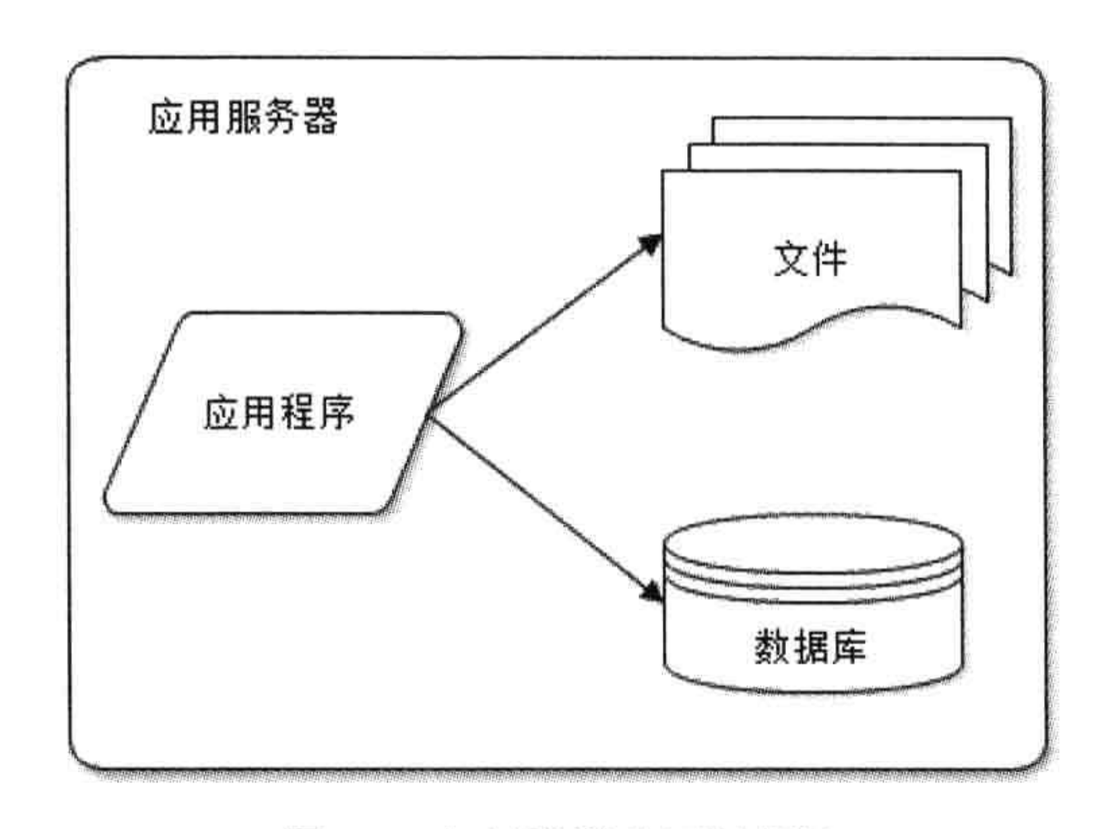

图 1.1 初始阶段的网站架构

应用程序、数据库、文件等所有的资源都在一台服务器上。通常服务器操作系统使用 Linux，应用程序使用 PHP 开发，然后部署在 Apache 上，数据库使用 MySQL，汇集各种免费开源软件及一台廉价服务器就可以开始网站的发展之路了。

1.2.2 应用服务和数据服务分离

随着网站业务的发展，一台服务器逐渐不能满足需求：越来越多的用户访问导致性能越来越差，越来越多的数据导致存储空间不足。这时就需要将应用和数据分离。应用和数据分离后整个网站使用三台服务器：应用服务器、文件服务器和数据库服务器，如

图 1.2 所示。这三台服务器对硬件资源的要求各不相同，应用服务器需要处理大量的业务逻辑，因此需要更快更强大的 CPU；数据库服务器需要快速磁盘检索和数据缓存，因此需要更快的硬盘和更大的内存；文件服务器需要存储大量用户上传的文件，因此需要更大的硬盘。

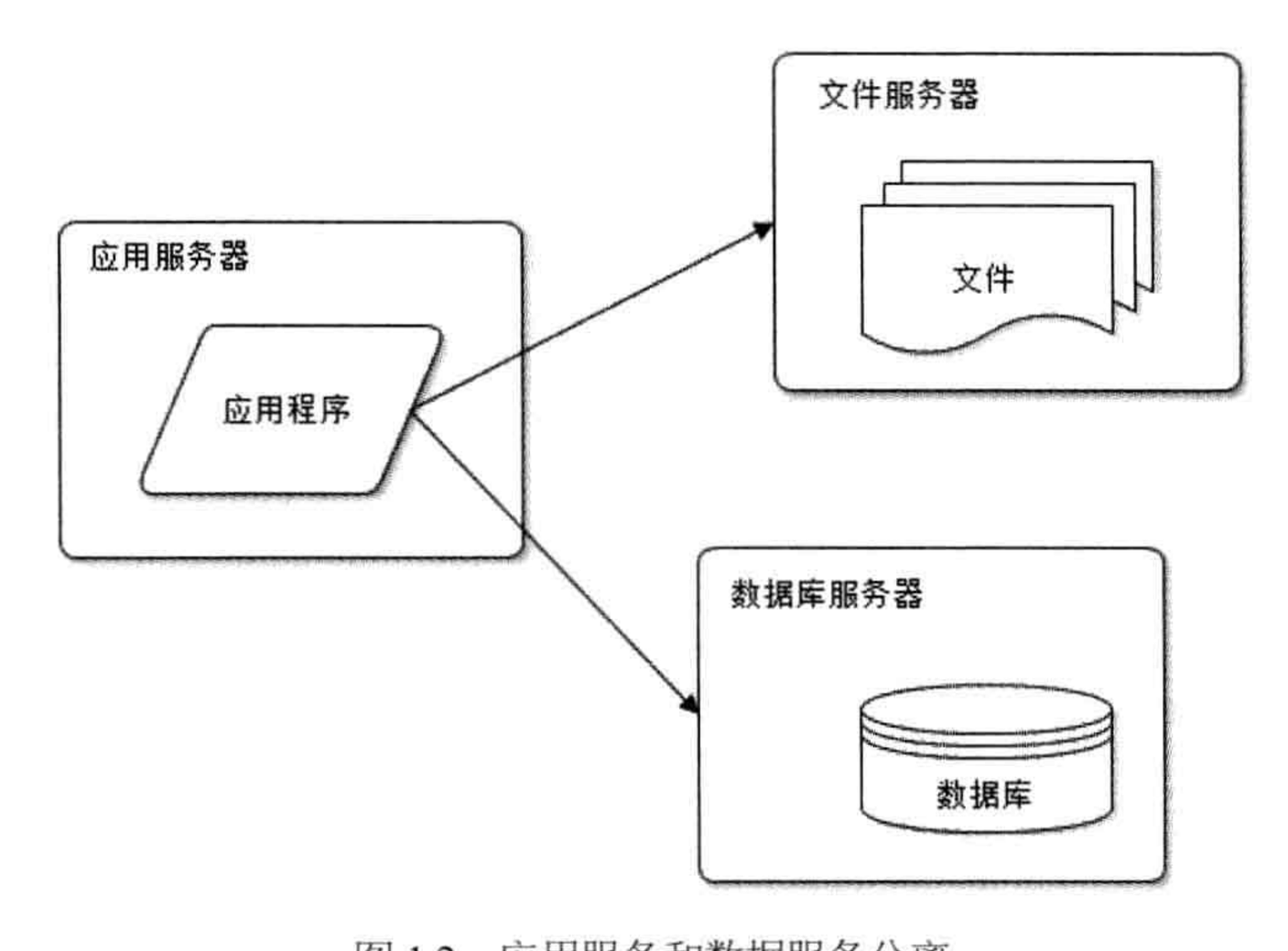

图 1.2 应用服务和数据服务分离

应用和数据分离后，不同特性的服务器承担不同的服务角色，网站的并发处理能力和数据存储空间得到了很大改善，支持网站业务进一步发展。但是随着用户逐渐增多，网站又一次面临挑战：数据库压力太大导致访问延迟，进而影响整个网站的性能，用户体验受到影响。这时需要对网站架构进一步优化。

1.2.3 使用缓存改善网站性能

网站访问特点和现实世界的财富分配一样遵循二八定律：80%的业务访问集中在 20%的数据上。淘宝买家浏览的商品集中在少部分成交数多、评价良好的商品上；百度搜索关键词集中在少部分热门词汇上；只有经常登录的用户才会发微博、看微博，而这部分用户也只占总用户数目的一小部分。

既然大部分的业务访问集中在一小部分数据上，那么如果把这一小部分数据缓存在内存中，是不是就可以减少数据库的访问压力，提高整个网站的数据访问速度，改善数据库的写入性能了呢？

网站使用的缓存可以分为两种：缓存在应用服务器上的本地缓存和缓存在专门的分布式缓存服务器上的远程缓存。本地缓存的访问速度更快一些，但是受应用服务器内存限制，其缓存数据量有限，而且会出现和应用程序争用内存的情况。远程分布式缓存可以使用集群的方式，部署大内存的服务器作为专门的缓存服务器，可以在理论上做到不受内存容量限制的缓存服务，如图 1.3 所示。

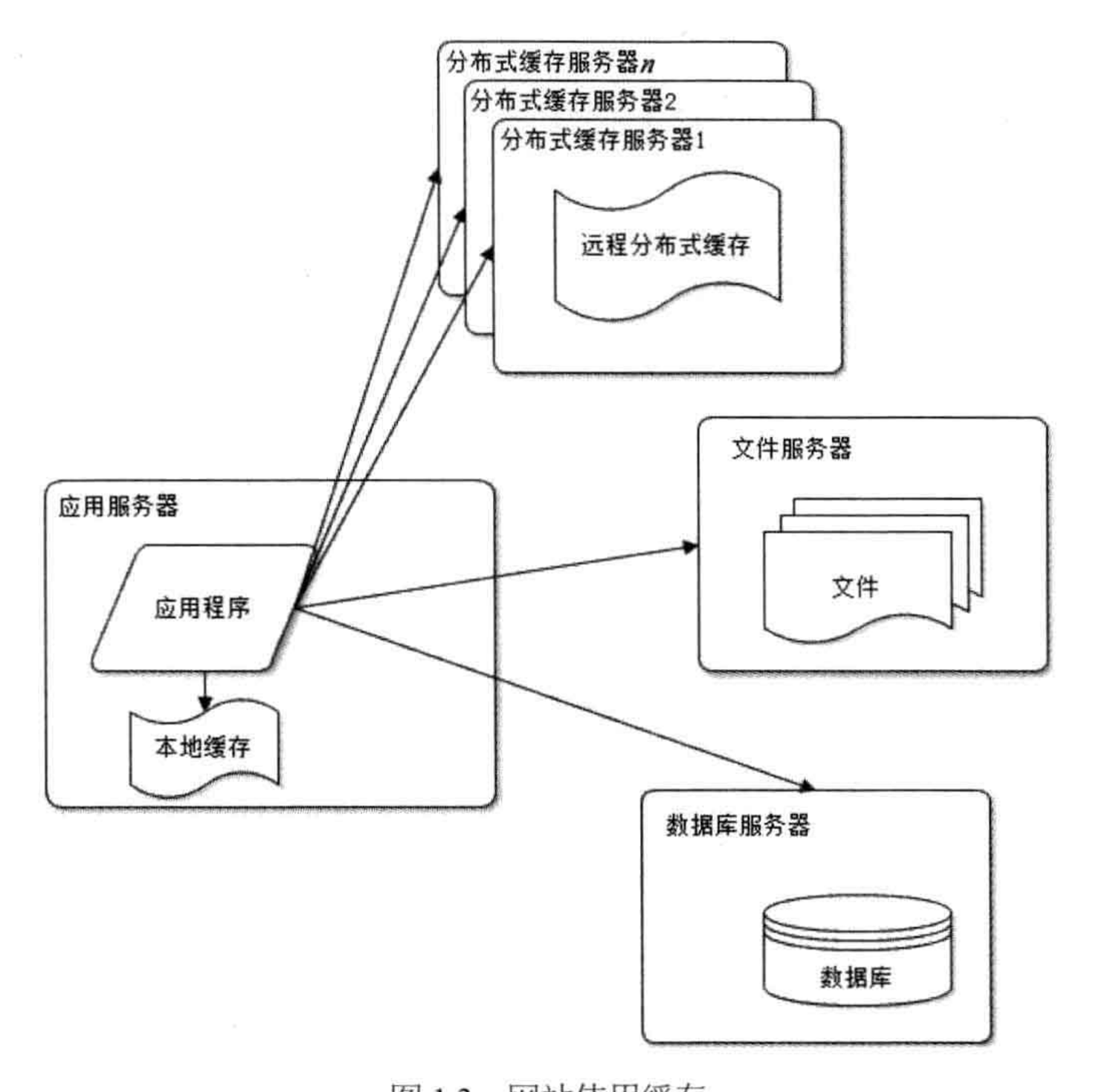

图 1.3　网站使用缓存

使用缓存后，数据访问压力得到有效缓解，但是单一应用服务器能够处理的请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈。

1.2.4　使用应用服务器集群改善网站的并发处理能力

使用集群是网站解决高并发、海量数据问题的常用手段。当一台服务器的处理能力、存储空间不足时，不要企图去换更强大的服务器，对大型网站而言，不管多么强大的服务器，都满足不了网站持续增长的业务需求。这种情况下，更恰当的做法是增加一台服务器分担原有服务器的访问及存储压力。

对网站架构而言，只要能通过增加一台服务器的方式改善负载压力，就可以以同样的方式持续增加服务器不断改善系统性能，从而实现系统的可伸缩性。应用服务器实现集群是网站可伸缩集群架构设计中较为简单成熟的一种，如图 1.4 所示。

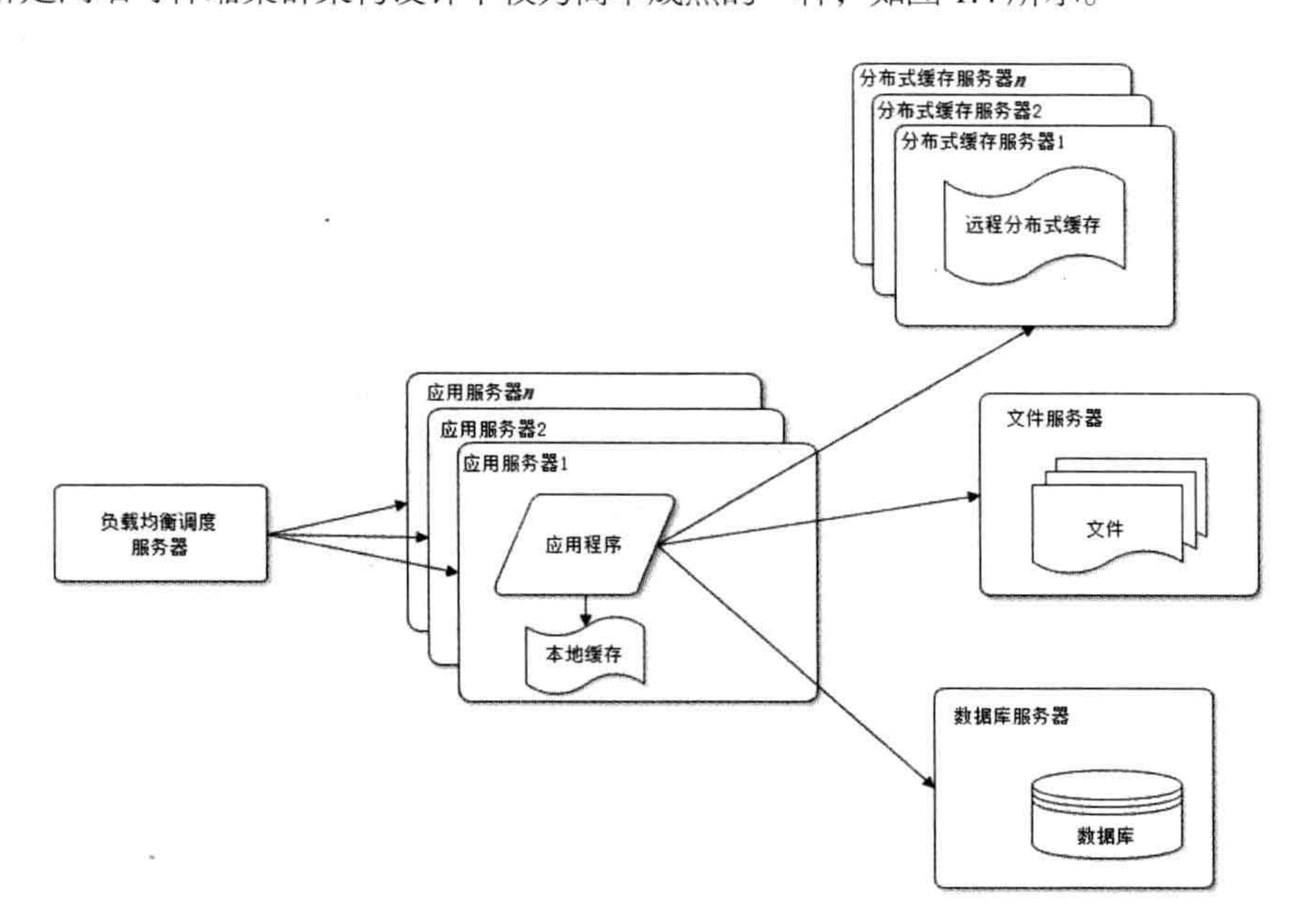

图 1.4 应用服务器集群部署

通过负载均衡调度服务器，可将来自用户浏览器的访问请求分发到应用服务器集群中的任何一台服务器上，如果有更多的用户，就在集群中加入更多的应用服务器，使应用服务器的负载压力不再成为整个网站的瓶颈。

1.2.5 数据库读写分离

网站在使用缓存后，使绝大部分数据读操作访问都可以不通过数据库就能完成，但是仍有一部分读操作（缓存访问不命中、缓存过期）和全部的写操作需要访问数据库，在网站的用户达到一定规模后，数据库因为负载压力过高而成为网站的瓶颈。

目前大部分的主流数据库都提供主从热备功能，通过配置两台数据库主从关系，可以将一台数据库服务器的数据更新同步到另一台服务器上。网站利用数据库的这一功能，实现数据库读写分离，从而改善数据库负载压力，如图 1.5 所示。

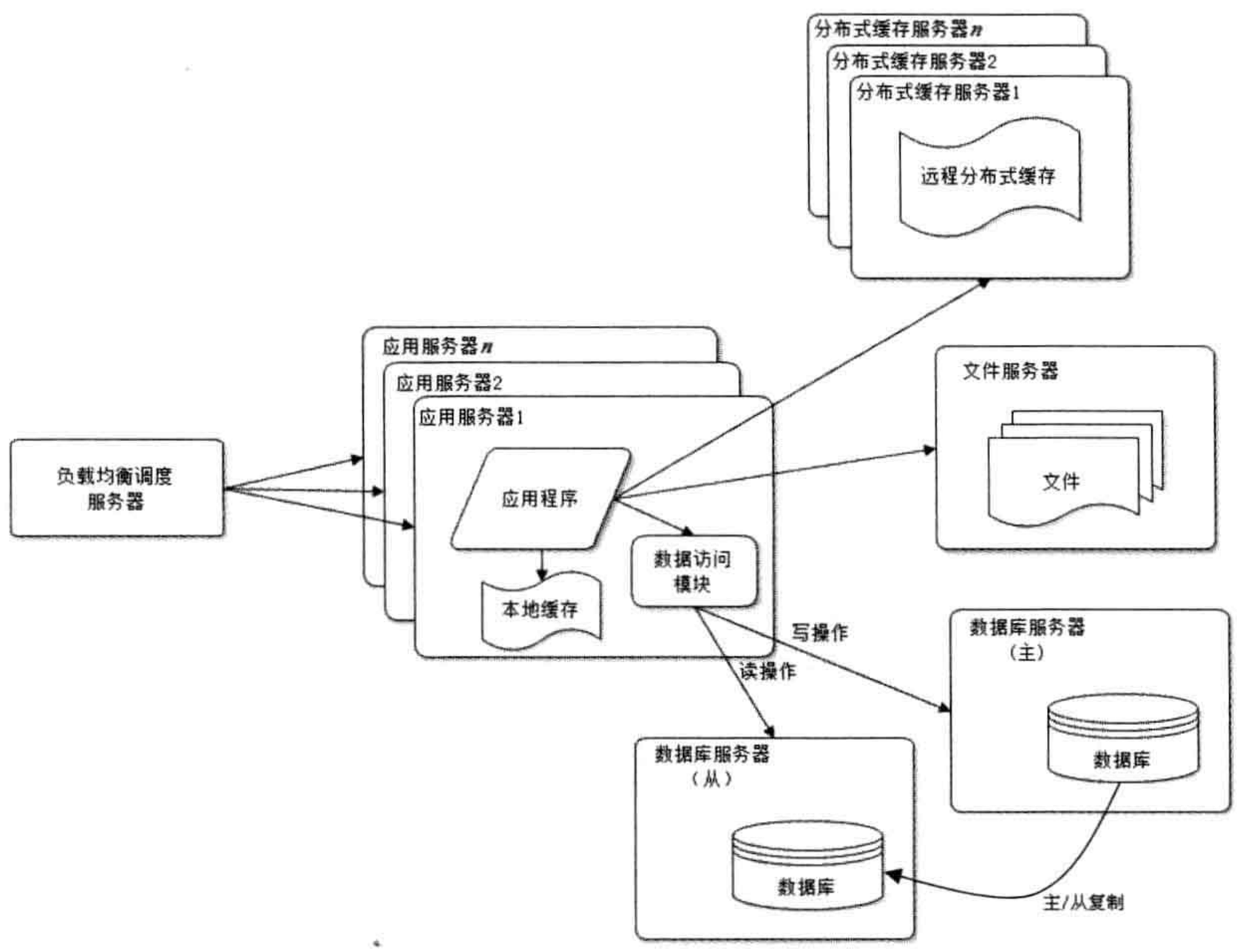

图 1.5　数据库读写分离

应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制机制将数据更新同步到从数据库，这样当应用服务器读数据的时候，就可以通过从数据库获得数据。为了便于应用程序访问读写分离后的数据库，通常在应用服务器端使用专门的数据访问模块，使数据库读写分离对应用透明。

1.2.6　使用反向代理和 CDN 加速网站响应

随着网站业务不断发展，用户规模越来越大，由于中国复杂的网络环境，不同地区的用户访问网站时，速度差别也极大。有研究表明，网站访问延迟和用户流失率正相关，网站访问越慢，用户越容易失去耐心而离开。为了提供更好的用户体验，留住用户，网站需要加速网站访问速度。主要手段有使用 CDN 和反向代理，如图 1.6 所示。

CDN 和反向代理的基本原理都是缓存，区别在于 CDN 部署在网络提供商的机房，使用户在请求网站服务时，可以从距离自己最近的网络提供商机房获取数据；而反向代理则部署在网站的中心机房，当用户请求到达中心机房后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户。

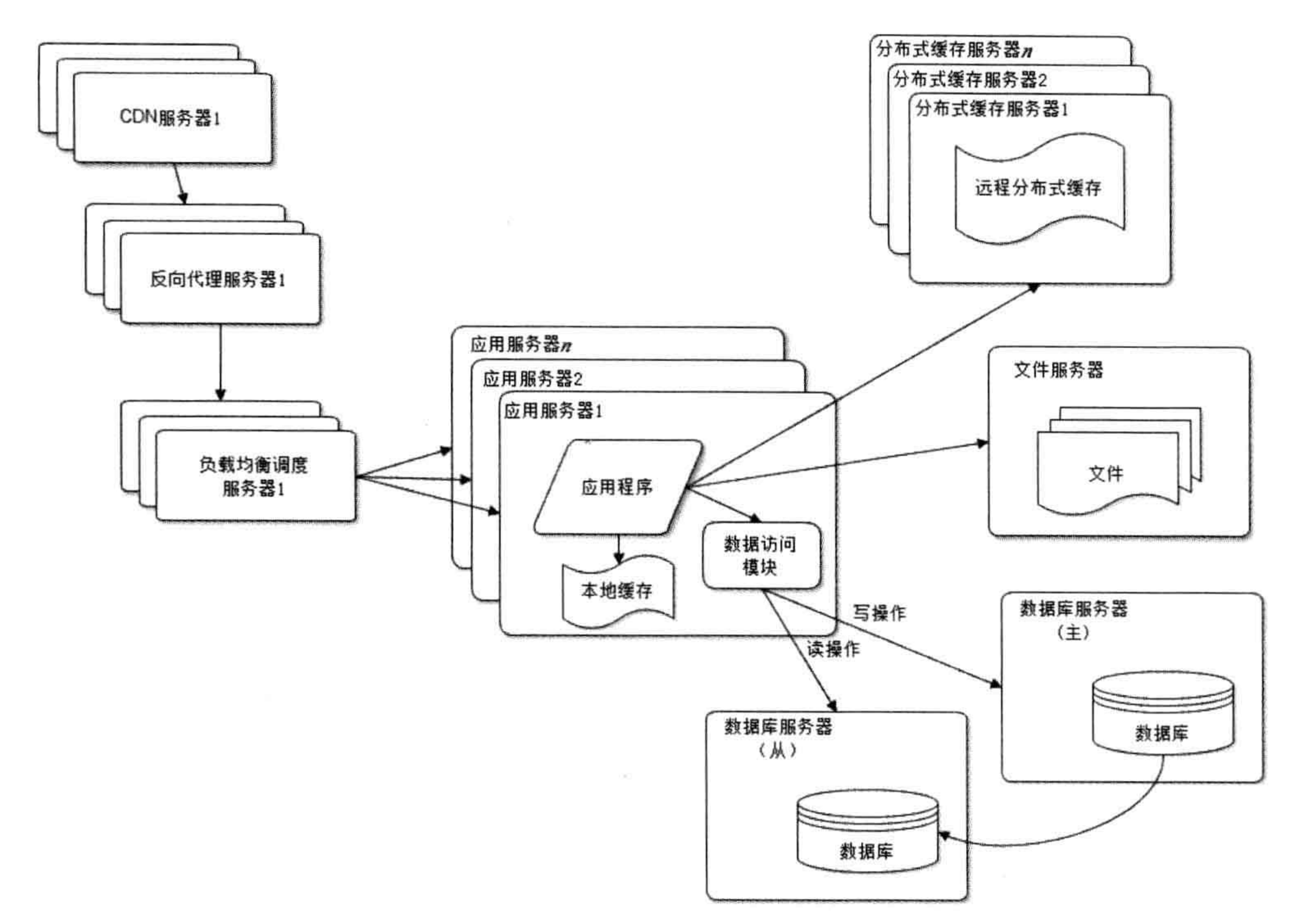

图 1.6　网站使用反向代理和 CDN 加速访问

使用 CDN 和反向代理的目的都是尽早返回数据给用户，一方面加快用户访问速度，另一方面也减轻后端服务器的负载压力。

1.2.7　使用分布式文件系统和分布式数据库系统

任何强大的单一服务器都满足不了大型网站持续增长的业务需求。数据库经过读写分离后，从一台服务器拆分成两台服务器，但是随着网站业务的发展依然不能满足需求，这时需要使用分布式数据库。文件系统也是一样，需要使用分布式文件系统，如图 1.7 所示。

分布式数据库是网站数据库拆分的最后手段，只有在单表数据规模非常庞大的时候才使用。不到不得已时，网站更常用的数据库拆分手段是业务分库，将不同业务的数据库部署在不同的物理服务器上。

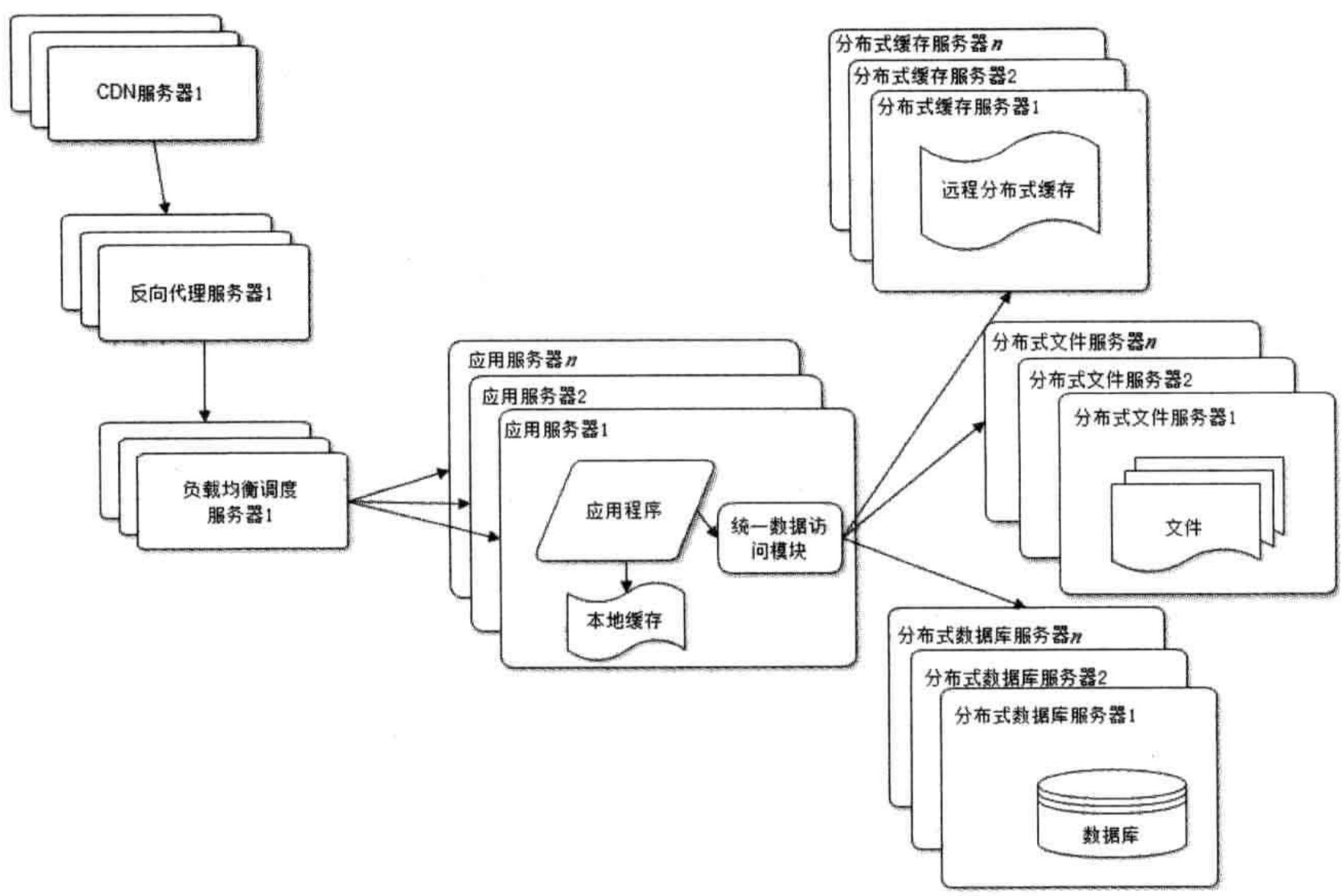

图 1.7　使用分布式文件和分布式数据库系统

1.2.8　使用 NoSQL 和搜索引擎

随着网站业务越来越复杂，对数据存储和检索的需求也越来越复杂，网站需要采用一些非关系数据库技术如 NoSQL 和非数据库查询技术如搜索引擎，如图 1.8 所示。

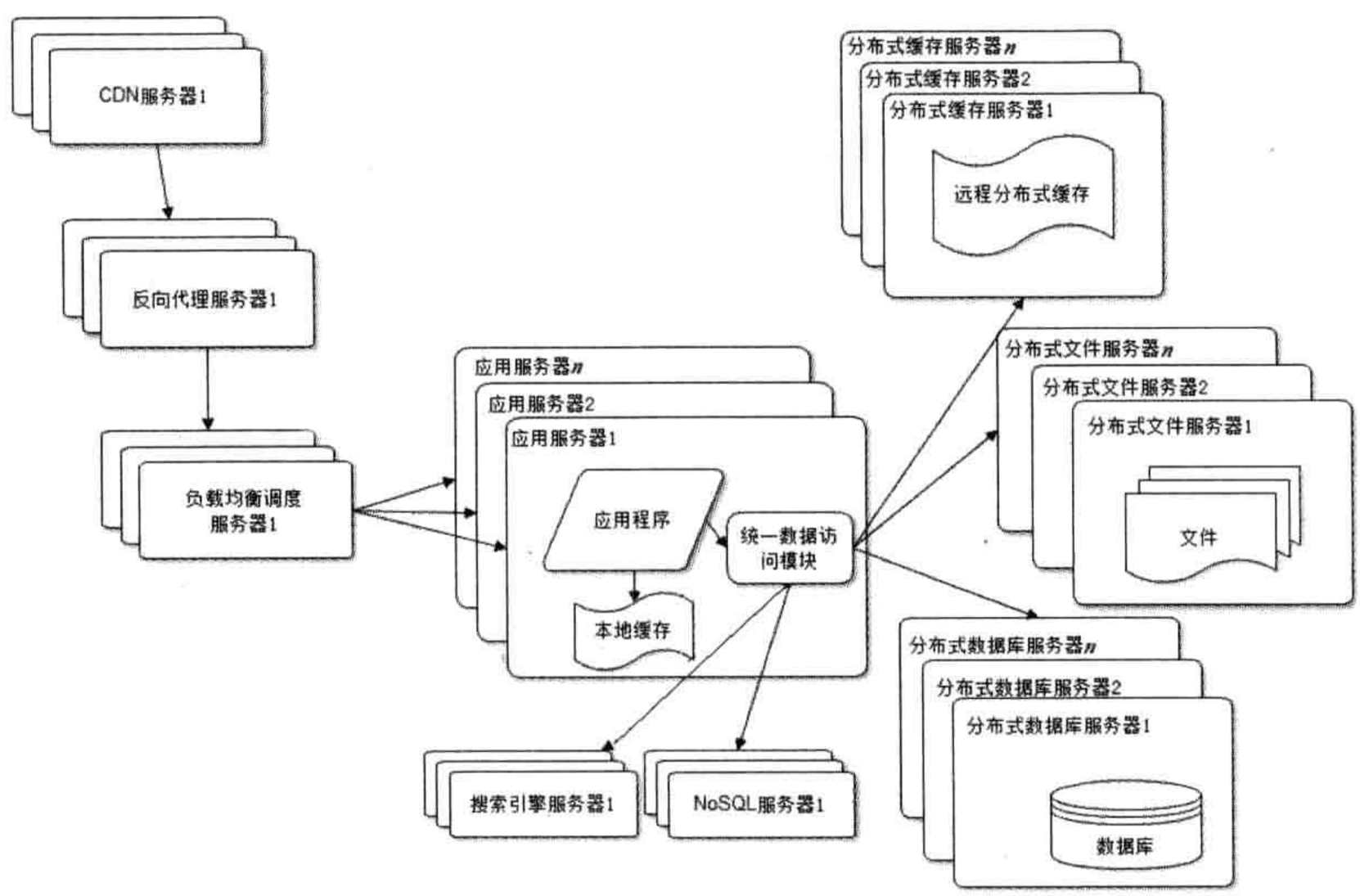

图 1.8　使用 NoSQL 系统和搜索引擎

NoSQL 和搜索引擎都是源自互联网的技术手段，对可伸缩的分布式特性具有更好的支持。应用服务器则通过一个统一数据访问模块访问各种数据，减轻应用程序管理诸多数据源的麻烦。

1.2.9 业务拆分

大型网站为了应对日益复杂的业务场景，通过使用分而治之的手段将整个网站业务分成不同的产品线，如大型购物交易网站就会将首页、商铺、订单、买家、卖家等拆分成不同的产品线，分归不同的业务团队负责。

具体到技术上，也会根据产品线划分，将一个网站拆分成许多不同的应用，每个应用独立部署维护。应用之间可以通过一个超链接建立关系（在首页上的导航链接每个都指向不同的应用地址），也可以通过消息队列进行数据分发，当然最多的还是通过访问同一个数据存储系统来构成一个关联的完整系统，如图 1.9 所示。

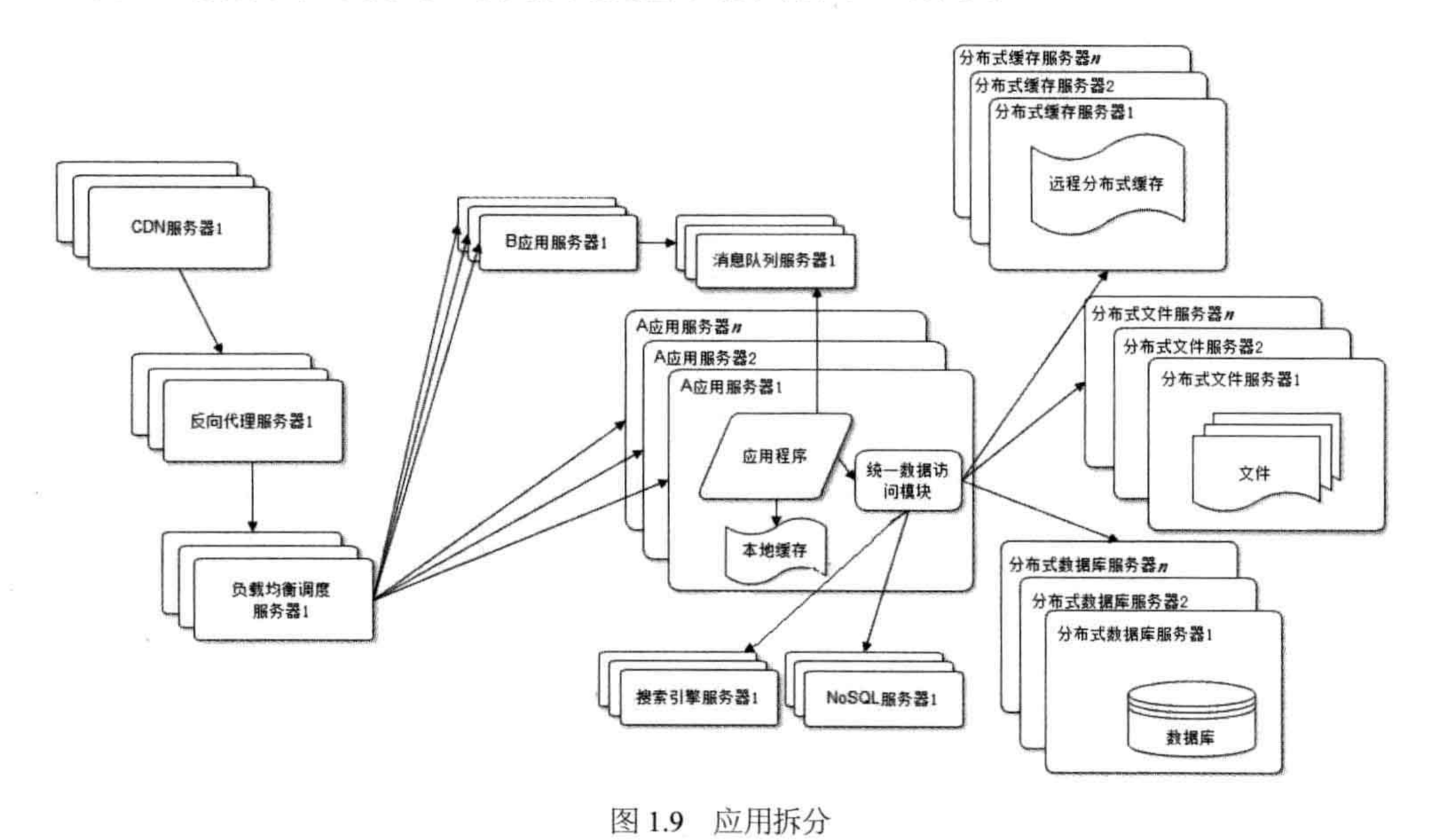

图 1.9 应用拆分

1.2.10 分布式服务

随着业务拆分越来越小，存储系统越来越庞大，应用系统的整体复杂度呈指数级增加，部署维护越来越困难。由于所有应用要和所有数据库系统连接，在数万台服务器规

模的网站中，这些连接的数目是服务器规模的平方，导致存数据库接资源不足，拒绝服务。

既然每一个应用系统都需要执行许多相同的业务操作，比如用户管理、商品管理等，那么可以将这些共用的业务提取出来，独立部署。由这些可复用的业务连接数据库，提供共用业务服务，而应用系统只需要管理用户界面，通过分布式服务调用共用业务服务完成具体业务操作，如图 1.10 所示。

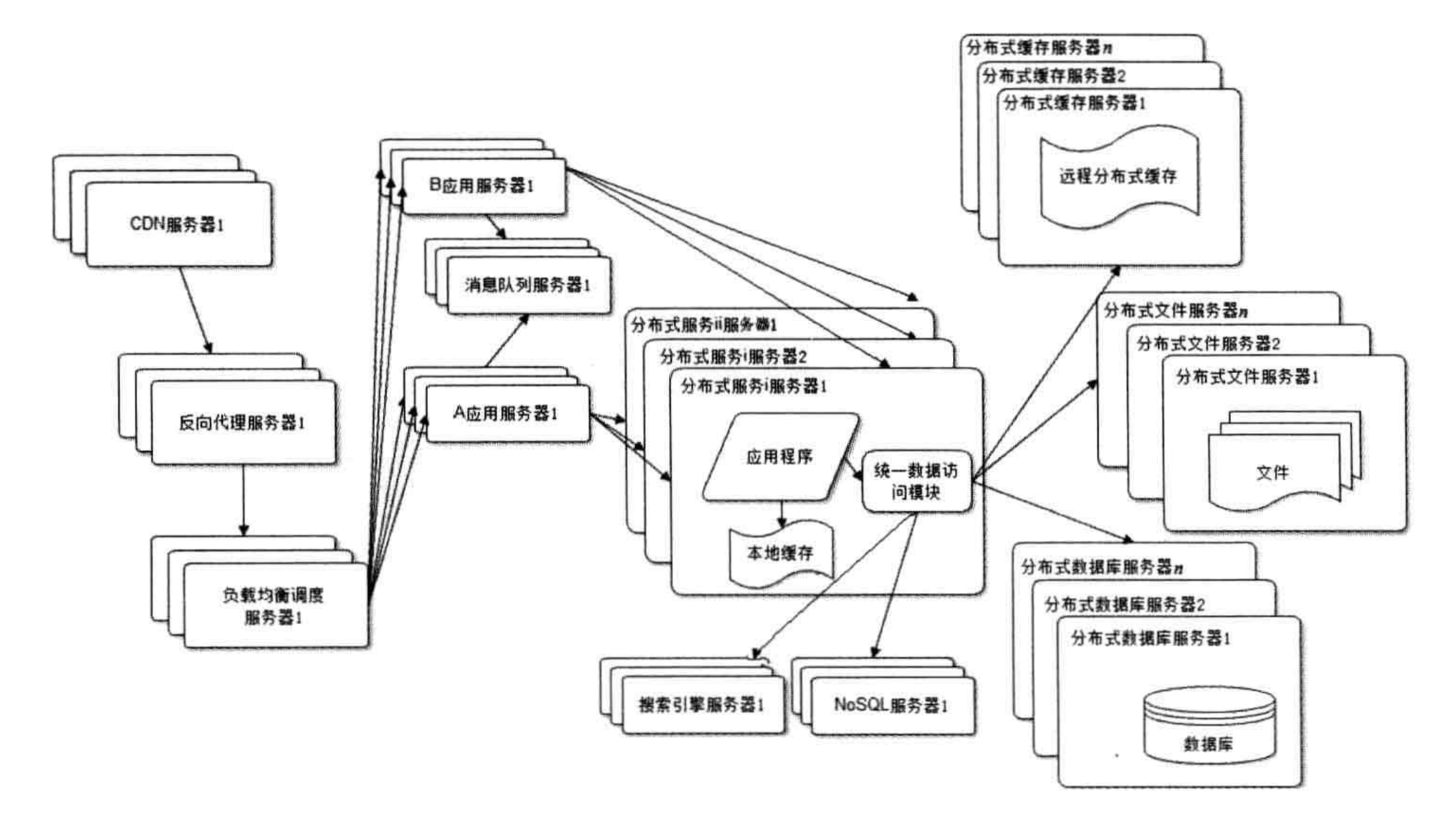

图 1.10　分布式服务

大型网站的架构演化到这里，基本上大多数的技术问题都得以解决，诸如跨数据中心的实时数据同步和具体网站业务相关的问题也都可以通过组合改进现有技术架构来解决。

但事物发展到一定阶段，就会拥有自身的发展冲动，摆脱其初衷，向着使自己更强大的方向发展。既然大型网站架构解决了海量数据的管理和高并发事务的处理，那么就可以把这些解决方案应用到网站自身以外的业务上去。我们看到目前许多大型网站都开始建设云计算平台，将计算作为一种基础资源出售，中小网站不需要再关心技术架构问题，只需要按需付费，就可以使网站随着业务的增长逐渐获得更大的存储空间和更多的计算资源。

1.3 大型网站架构演化的价值观

这个世界没有哪个网站从诞生起就是大型网站；也没有哪个网站第一次发布就拥有庞大的用户，高并发的访问，海量的数据；大型网站都是从小型网站发展而来。网站的价值在于它能为用户提供什么价值，在于网站能做什么，而不在于它是怎么做的，所以在网站还很小的时候就去追求网站的架构是舍本逐末，得不偿失的。小型网站最需要做的就是为用户提供好的服务来创造价值，得到用户的认可，活下去，野蛮生长。

所以我们看到，一方面是随着互联网的高速发展，越来越多新的软件技术和产品从互联网公司诞生，挑战传统软件巨头的江湖地位。另一方面却是中小网站十几年如一日地使用 LAMP 技术（Linux＋Apache＋MySQL＋PHP）开发自己的网站，因为 LAMP 既便宜又简单，而且对付一个中小型网站绰绰有余。

1.3.1 大型网站架构技术的核心价值是随网站所需灵活应对

大型网站架构技术的核心价值不是从无到有搭建一个大型网站，而是能够伴随小型网站业务的逐步发展，慢慢地演化成一个大型网站。在这个漫长的技术演化过程中，不需要放弃什么，不需要推翻什么，不需要剧烈的革命，就那么润物细无声地把一个只有一台服务器，几百个用户的小网站演化成一个几十万台服务器，数十亿用户的大网站。今天我们看到的大型网站，Google，Facebook，Taobao，Baidu 莫不遵循这样的技术演化路线。

1.3.2 驱动大型网站技术发展的主要力量是网站的业务发展

创新的业务发展模式对网站架构逐步提出更高要求，才使得创新的网站架构得以发展成熟。是业务成就了技术，是事业成就了人，而不是相反。所以网站架构师应该对成就自己技术成绩的网站事业心存感恩，并努力提高技术回馈业务，才能在快速发展的互联网领域保持持续进步。

不过我们也看到有些传统企业投身互联网，在业务问题还没有理清楚的时候就从外面挖来许多技术高手，仿照成功的互联网公司打造技术平台，这无疑是南辕北辙，缘木求鱼。而这些技术高手离开了它们熟悉的环境和工作模式，也是张飞拿着绣花针使不上劲来。

1.4 网站架构设计误区

在大型网站架构发展过程中有如下几个容易出现的误区。

1.4.1 一味追随大公司的解决方案

由于大公司巨大成功的光环效应，再加上从大公司挖来的技术高手的影响，网站在讨论架构决策时，最有说服力的一句话就成了“淘宝就是这么搞的”或者“Facebook 就是这么搞的”。

大公司的经验和成功模式固然重要，值得学习借鉴，但如果因此而变得盲从，就失去了坚持自我的勇气，在架构演化的道路上迟早会迷路。

1.4.2 为了技术而技术

网站技术是为业务而存在的，除此毫无意义。在技术选型和架构设计中，脱离网站业务发展的实际，一味追求时髦的新技术，可能会将网站技术发展引入崎岖小道，架构之路越走越难。

1.4.3 企图用技术解决所有问题

最典型的例子就是 2012 年年初 12306 故障事件后，软件开发技术界的反应。

各路专业和非专业人士众说纷纭地帮 12306 的技术架构出谋划策，甚至有人提议帮 12306 写一个开源的网站，解决其大规模并发访问的问题。

12306 真正的问题其实不在于它的技术架构，而在于它的业务架构：12306 根本就不应该在几亿中国人一票难求的情况下以窗口售票的模式在网上售票（零点开始出售若干天后的车票）。12306 需要重构的不仅是它的技术架构，更重要的是它的业务架构：调整业务需求，换一种方式卖票，而不要去搞促销秒杀这种噱头式的游戏。

后来证明 12306 确实是朝这个方向发展的：在售票方式上引入了排队机制、整点售票调整为分时段售票。其实如果能控制住并发访问的量，很多棘手的技术问题也就不是什么问题了。

技术是用来解决业务问题的，而业务的问题，也可以通过业务的手段去解决。

1.5 小结

时至今日，大型网站的架构演化方案已经非常成熟，各种技术方案也逐渐产品化。许多小型网站已经慢慢不需要再经历大型网站经历过的架构演化之路就可以逐步发展壮大，因为现在越来越多的网站从建立之初就是搭建在大型网站提供的云计算服务基础之上，所需要的一切技术资源：计算、存储、网络都可以按需购买，线性伸缩，不需要自己一点一点地拼凑各种资源，综合使用各种技术方案逐步去完善自己的网站架构了。

所以能亲身经历一个网站从小到大的架构演化过程的网站架构师越来越少，虽然过去有这种经历的架构师也很少（从小型网站发展成大型网站的机会本来就极少），但是将来可能真就没有了。

但也正因为网站架构技术演化过程难以重现，所以网站架构师更应该对这个过程深刻了解，理解已成熟的网站架构技术方案的来龙去脉和历史渊源，在技术选型和架构决策时才能有的放矢，直击要害。

2 大型网站架构模式

关于什么是模式，这个来自建筑学的词汇是这样定义的：“**每一个模式描述了一个在我们周围不断重复发生的问题及该问题解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复工作**”。模式的关键在于模式的可重复性，问题与场景的可重复性带来解决方案的可重复使用。

我们的现实生活中充斥着几乎千篇一律的人生架构模式：读重点学校，选热门专业，进稳定高收入的政府部门和企业，找门当户对的配偶，生一个听话的孩子继续这个模式……但是人生不同于软件，精彩的人生绝不会来自于复制。

也许互联网产品不是随便复制就能成功的，创新的产品更能为用户创造价值。但是网站架构却有一些共同的模式，这些模式已经被许多大型网站一再验证，通过对这些模式的学习，我们可以掌握大型网站架构的一般思路和解决方案，以指导我们的架构设计。

2.1 网站架构模式

为了解决大型网站面临的高并发访问、海量数据处理、高可靠运行等一系列问题与挑战，大型互联网公司在实践中提出了许多解决方案，以实现网站高性能、高可用、

易伸缩、可扩展、安全等各种技术架构目标。这些解决方案又被更多网站重复使用，从而逐渐形成大型网站架构模式。

2.1.1 分层

分层是企业应用系统中最常见的一种架构模式，将系统在横向维度上切分成几个部分，每个部分负责一部分相对比较单一的职责，然后通过上层对下层的依赖和调用组成一个完整的系统。

分层结构在计算机世界中无处不在，网络的 7 层通信协议是一种分层结构；计算机硬件、操作系统、应用软件也可以看作是一种分层结构。在大型网站架构中也采用分层结构，将网站软件系统分为应用层、服务层、数据层，如表 2.1 所示。

表 2.1 网站分层架构

应用层	负责具体业务和视图展示，如网站首页及搜索输入和结果展示
服务层	为应用层提供服务支持，如用户管理服务，购物车服务等
数据层	提供数据存储访问服务，如数据库、缓存、文件、搜索引擎等

通过分层，可以更好地将一个庞大的软件系统切分成不同的部分，便于分工合作开发和维护；各层之间具有一定的独立性，只要维持调用接口不变，各层可以根据具体问题独立演化发展而不需要其他层必须做出相应调整。

但是分层架构也有一些挑战，就是必须合理规划层次边界和接口，在开发过程中，严格遵循分层架构的约束，禁止跨层次的调用（应用层直接调用数据层）及逆向调用（数据层调用服务层，或者服务层调用应用层）。

在实践中，大的分层结构内部还可以继续分层，如应用层可以再细分为视图层（美工负责）和业务逻辑层（工程师负责）；服务层也可以细分为数据接口层（适配各种输入和输出的数据格式）和逻辑处理层。

分层架构是逻辑上的，在物理部署上，三层结构可以部署在同一个物理机器上，但是随着网站业务的发展，必然需要对已经分层的模块分离部署，即三层结构分别部署在不同的服务器上，使网站拥有更多的计算资源以应对越来越多的用户访问。

所以虽然分层架构模式最初的目的是规划软件清晰的逻辑结构便于开发维护，但在

网站的发展过程中，分层结构对网站支持高并发向分布式方向发展至关重要。因此在网站规模还很小的时候就应该采用分层的架构，这样将来网站做大时才能有更好地应对。

2.1.2 分割

如果说分层是将软件在横向方面进行切分，那么分割就是在纵向方面对软件进行切分。

网站越大，功能越复杂，服务和数据处理的种类也越多，将这些不同的功能和服务分割开来，包装成高内聚低耦合的模块单元，一方面有助于软件的开发和维护；另一方面，便于不同模块的分布式部署，提高网站的并发处理能力和功能扩展能力。

大型网站分割的粒度可能会很小。比如在应用层，将不同业务进行分割，例如将购物、论坛、搜索、广告分割成不同的应用，由独立的团队负责，部署在不同的服务器上；在同一个应用内部，如果规模庞大业务复杂，会继续进行分割，比如购物业务，可以进一步分割成机票酒店业务、3C 业务，小商品业务等更细小的粒度。而即使在这个粒度上，还是可以继续分割成首页、搜索列表、商品详情等模块，这些模块不管在逻辑上还是物理部署上，都可以是独立的。同样在服务层也可以根据需要将服务分割成合适的模块。

2.1.3 分布式

对于大型网站，分层和分割的一个主要目的是为了切分后的模块便于分布式部署，即将不同模块部署在不同的服务器上，通过远程调用协同工作。分布式意味着可以使用更多的计算机完成同样的功能，计算机越多，CPU、内存、存储资源也就越多，能够处理的并发访问和数据量就越大，进而能够为更多的用户提供服务。

但分布式在解决网站高并发问题的同时也带来了其他问题。首先，分布式意味着服务调用必须通过网络，这可能会对性能造成比较严重的影响；其次，服务器越多，服务器宕机的概率也就越大，一台服务器宕机造成的服务不可用可能会导致很多应用不可访问，使网站可用性降低；另外，数据在分布式的环境中保持数据一致性也非常困难，分布式事务也难以保证，这对网站业务正确性和业务流程有可能造成很大影响；分布式还导致网站依赖错综复杂，开发管理维护困难。因此分布式设计要根据具体情况量力而行，切莫为了分布式而分布式。

在网站应用中，常用的分布式方案有以下几种。

分布式应用和服务：将分层和分割后的应用和服务模块分布式部署，除了可以改善网站性能和并发性、加快开发和发布速度、减少数据库连接资源消耗外；还可以使不同应用复用共同的服务，便于业务功能扩展。

分布式静态资源：网站的静态资源如 JS，CSS，Logo 图片等资源独立分布式部署，并采用独立的域名，即人们常说的动静分离。静态资源分布式部署可以减轻应用服务器的负载压力；通过使用独立域名加快浏览器并发加载的速度；由负责用户体验的团队进行开发维护有利于网站分工合作，使不同技术工种术业有专攻。

分布式数据和存储：大型网站需要处理以 P 为单位的海量数据，单台计算机无法提供如此大的存储空间，这些数据需要分布式存储。除了对传统的关系数据库进行分布式部署外，为网站应用而生的各种 NoSQL 产品几乎都是分布式的。

分布式计算：严格说来，应用、服务、实时数据处理都是计算，网站除了要处理这些在线业务，还有很大一部分用户没有直观感受的后台业务要处理，包括搜索引擎的索引构建、数据仓库的数据分析统计等。这些业务的计算规模非常庞大，目前网站普遍使用 Hadoop 及其 MapReduce 分布式计算框架进行此类批处理计算，其特点是移动计算而不是移动数据，将计算程序分发到数据所在的位置以加速计算和分布式计算。

此外，还有可以支持网站线上服务器配置实时更新的**分布式配置**；分布式环境下实现并发和协同的**分布式锁**；支持云存储的**分布式文件**系统等。

2.1.4 集群

使用分布式虽然已经将分层和分割后的模块独立部署，但是对于用户访问集中的模块（比如网站的首页），还需要将独立部署的服务器集群化，即多台服务器部署相同应用构成一个集群，通过负载均衡设备共同对外提供服务。

因为服务器集群有更多服务器提供相同服务，因此可以提供更好的并发特性，当有更多用户访问的时候，只需要向集群中加入新的机器即可。同时因为一个应用由多台服务器提供，当某台服务器发生故障时，负载均衡设备或者系统的失效转移机制会将请求转发到集群中其他服务器上，使服务器故障不影响用户使用。所以在网站应用中，即使是访问量很小的分布式应用和服务，也至少要部署两台服务器构成一个小的集群，目的就是提高系统的可用性。

2.1.5 缓存

缓存就是将数据存放在距离计算最近的位置以加快处理速度。缓存是改善软件性能的第一手段，现代 CPU 越来越快的一个重要因素就是使用了更多的缓存，在复杂的软件设计中，缓存几乎无处不在。大型网站架构设计在很多方面都使用了缓存设计。

CDN：即内容分发网络，部署在距离终端用户最近的网络服务商，用户的网络请求总是先到达他的网络服务商那里，在这里缓存网站的一些静态资源（较少变化的数据），可以就近以最快速度返回给用户，如视频网站和门户网站会将用户访问量大的热点内容缓存在 CDN。

反向代理：反向代理属于网站前端架构的一部分，部署在网站的前端，当用户请求到达网站的数据中心时，最先访问到的就是反向代理服务器，这里缓存网站的静态资源，无需将请求继续转发给应用服务器就能返回给用户。

本地缓存：在应用服务器本地缓存着热点数据，应用程序可以在本机内存中直接访问数据，而无需访问数据库。

分布式缓存：大型网站的数据量非常庞大，即使只缓存一小部分，需要的内存空间也不是单机能承受的，所以除了本地缓存，还需要分布式缓存，将数据缓存在一个专门的分布式缓存集群中，应用程序通过网络通信访问缓存数据。

使用缓存有两个前提条件，一是数据访问热点不均衡，某些数据会被更频繁的访问，这些数据应该放在缓存中；二是数据在某个时间段内有效，不会很快过期，否则缓存的数据就会因已经失效而产生脏读，影响结果的正确性。网站应用中，缓存除了可以加快数据访问速度，还可以减轻后端应用和数据存储的负载压力，这一点对网站数据库架构至关重要，网站数据库几乎都是按照有缓存的前提进行负载能力设计的。

2.1.6 异步

计算机软件发展的一个重要目标和驱动力是降低软件耦合性。事物之间直接关系越少，就越少被彼此影响，越可以独立发展。大型网站架构中，系统解耦合的手段除了前面提到的分层、分割、分布等，还有一个重要手段是异步，业务之间的消息传递不是同步调用，而是将一个业务操作分成多个阶段，每个阶段之间通过共享数据的方式异步执

行进行协作。

在单一服务器内部可通过多线程共享内存队列的方式实现异步，处在业务操作前面的线程将输出写入到队列，后面的线程从队列中读取数据进行处理；在分布式系统中，多个服务器集群通过分布式消息队列实现异步，分布式消息队列可以看作内存队列的分布式部署。

异步架构是典型的生产者消费者模式，两者不存在直接调用，只要保持数据结构不变，彼此功能实现可以随意变化而不互相影响，这对网站扩展新功能非常便利。除此之外，使用异步消息队列还有如下特性。

提高系统可用性。消费者服务器发生故障，数据会在消息队列服务器中存储堆积，生产者服务器可以继续处理业务请求，系统整体表现无故障。消费者服务器恢复正常后，继续处理消息队列中的数据。

加快网站响应速度。处在业务处理前端的生产者服务器在处理完业务请求后，将数据写入消息队列，不需要等待消费者服务器处理就可以返回，响应延迟减少。

消除并发访问高峰。用户访问网站是随机的，存在访问高峰和低谷，即使网站按照一般访问高峰进行规划和部署，也依然会出现突发事件，比如购物网站的促销活动，微博上的热点事件，都会造成网站并发访问突然增大，这可能会造成整个网站负载过重，响应延迟，严重时甚至会出现服务宕机的情况。使用消息队列将突然增加的访问请求数据放入消息队列中，等待消费者服务器依次处理，就不会对整个网站负载造成太大压力。

但需要注意的是，使用异步方式处理业务可能会对用户体验、业务流程造成影响，需要网站产品设计方面的支持。

2.1.7 冗余

网站需要 7×24 小时连续运行，但是服务器随时可能出现故障，特别是服务器规模比较大时，出现某台服务器宕机是必然事件。要想保证在服务器宕机的情况下网站依然可以继续服务，不丢失数据，就需要一定程度的服务器冗余运行，数据冗余备份，这样当某台服务器宕机时，可以将其上的服务和数据访问转移到其他机器上。

访问和负载很小的服务也必须部署至少两台服务器构成一个集群，其目的就是通过

冗余实现服务高可用。数据库除了定期备份，存档保存，实现**冷备份**外，为了保证在线业务高可用，还需要对数据库进行主从分离，实时同步实现**热备份**。

为了抵御地震、海啸等不可抗力导致的网站完全瘫痪，某些大型网站会对整个数据中心进行备份，全球范围内部署**灾备数据中心**。网站程序和数据实时同步到多个灾备数据中心。

2.1.8 自动化

在无人值守的情况下网站可以正常运行，一切都可以自动化是网站的理想状态。目前大型网站的自动化架构设计主要集中在发布运维方面。

发布对网站都是头等大事，许多网站故障出在发布环节，网站工程师经常加班也是因为发布不顺利。通过减少人为干预，使**发布过程自动化**可有效减少故障。发布过程包括诸多环节。**自动化代码管理**，代码版本控制、代码分支创建合并等过程自动化，开发工程师只要提交自己参与开发的产品代号，系统就会自动为其创建开发分支，后期会自动进行代码合并；**自动化测试**，代码开发完成，提交测试后，系统自动将代码部署到测试环境，启动自动化测试用例进行测试，向相关人员发送测试报告，向系统反馈测试结果；**自动化安全检测**，安全检测工具通过对代码进行静态安全扫描及部署到安全测试环境进行安全攻击测试，评估其安全性；最后进行**自动化部署**，将工程代码自动部署到线上生产环境。

此外，网站在运行过程中可能会遇到各种问题：服务器宕机、程序 Bug、存储空间不足、突然爆发的访问高峰。网站需要对线上生产环境进行**自动化监控**，对服务器进行心跳检测，并监控其各项性能指标和应用程序的关键数据指标。如果发现异常、超出预设的阈值，就进行**自动化报警**，向相关人员发送报警信息，警告故障可能会发生。在检测到故障发生后，系统会进行**自动化失效转移**，将失效的服务器从集群中隔离出去，不再处理系统中的应用请求。待故障消除后，系统进行**自动化失效恢复**，重新启动服务，同步数据保证数据的一致性。在网站遇到访问高峰，超出网站最大处理能力时，为了保证整个网站的安全可用，还会进行**自动化降级**，通过拒绝部分请求及关闭部分不重要的服务将系统负载降至一个安全的水平，必要时，还需要**自动化分配资源**，将空闲资源分配给重要的服务，扩大其部署规模。

2.1.9 安全

互联网的开放特性使得其从诞生起就面对巨大的安全挑战，网站在安全架构方面也积累了许多模式：通过**密码**和**手机校验码**进行身份认证；登录、交易等操作需要对网络通信进行**加密**，网站服务器上存储的敏感数据如用户信息等也进行加密处理；为了防止机器人程序滥用网络资源攻击网站，网站使用**验证码**进行识别；对于常见的用于**攻击**网站的 XSS 攻击、SQL 注入、进行编码转换等相应处理；对于垃圾信息、敏感信息进行**过滤**；对交易转账等重要操作根据交易模式和交易信息进行**风险控制**。

2.2 架构模式在新浪微博的应用

短短几年时间新浪微博的用户数就从零增长到数亿，明星用户的粉丝数达数千万，围绕着新浪微博正在发展一个集社交、媒体、游戏、电商等多位一体的生态系统。

同大多数网站一样，新浪微博也是从一个小网站发展起来的。简单的 LAMP（Linux+Apache+MySQL+PHP）架构，支撑起最初的新浪微博，应用程序用 PHP 开发，所有的数据，包括微博、用户、关系都存储在 MySQL 数据库中。

这样简单的架构无法支撑新浪微博快速发展的业务需求，随着访问用户的逐渐增加，系统不堪重负。新浪微博的架构在较短时间内几经重构，最后形成现在的架构，如图 2.1 所示。

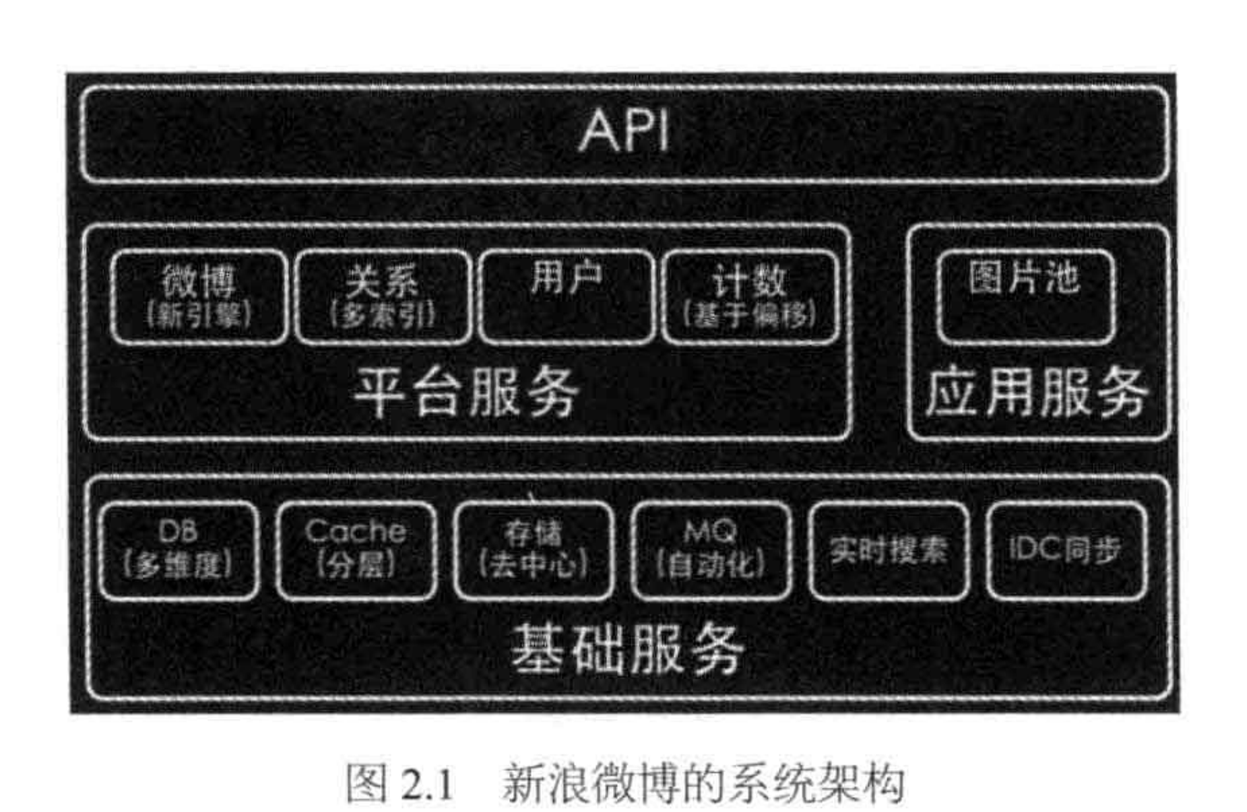

图 2.1 新浪微博的系统架构

（图片来源：http://timyang.net/architecture/weibo/）

系统**分为三个层次**，最下层是基础服务层，提供数据库、缓存、存储、搜索等数据服务，以及其他一些基础技术服务，这些服务支撑了新浪微博的海量数据和高并发访问，是整个系统的技术基础。

中间层是平台服务和应用服务层，新浪微博的核心服务是微博、关系和用户，它们是新浪微博业务大厦的支柱。这些服务被分割为独立的服务模块，通过依赖调用和共享基础数据构成新浪微博的业务基础。

最上层是 API 和新浪微博的业务层，各种客户端（包括 Web 网站）和第三方应用，通过调用 API 集成到新浪微博的系统中，共同组成一个生态系统。

这些被分层和分割后的业务模块与基础技术模块**分布式**部署，每个模块都部署在一组独立的服务器**集群**上，通过远程调用的方式进行依赖访问。新浪微博在早期还使用过一种叫作 MPSS（MultiPort Single Server，单服务器多端口）的分布式集群部署方案，在集群中的多台服务器上，每台都部署多个服务，每个服务使用不同的端口对外提供服务，通过这种方式使得有限的服务器可以部署更多的服务实例，改善服务的负载均衡和可用性。现在网站应用中常见的将物理机虚拟化成多个虚拟机后，在虚拟机上部署应用的方案跟新浪微博的 MPSS 方案异曲同工，只是更加简单，还能在不同虚拟机上使用相同的端口号。

在新浪微博的早期架构中，微博发布使用同步推模式，用户发表微博后系统会立即将这条微博插入到数据库所有粉丝的订阅列表中，当用户量比较大时，特别是明星用户发布微博时，会引起大量的数据库写操作，超出数据库负载，系统性能急剧下降，用户响应延迟加剧。后来新浪微博改用**异步**推拉结合的模式，用户发表微博后系统将微博写入消息队列后立即返回，用户响应迅速，消息队列消费者任务将微博推送给所有当前在线粉丝的订阅列表中，非在线用户登录后再根据关注列表拉取微博订阅列表。

由于微博频繁刷新，新浪微博使用多级**缓存**策略，热门微博和明星用户的微博缓存在所有的微博服务器上，在线用户的微博和近期微博缓存在分布式缓存集群中，对于微博操作中最常见的“刷微博”操作，几乎全部都是缓存访问操作，可以获得很好的系统性能。

为了提高系统的整体可用性和性能，新浪微博启用了多个数据中心。这些数据中心

既是地区用户访问中心，用户可以就近访问最近的数据中心以加快访问速度，改善系统性能；同时也是数据**冗余**复制的灾备中心，所有的用户和微博数据通过远程消息系统在不同的数据中心之间同步，提高系统可用性。

同时，新浪微博还开发了一系列**自动化**工具，包括自动化监控，自动化发布，自动化故障修复等，这些自动化工具还在持续开发中，以改善运维水平提高系统可用性。

由于微博的开放特性，新浪微博也遇到了一系列的**安全**挑战，垃圾内容、僵尸粉、微博攻击从未停止，除了使用一般网站常见的安全策略，新浪微博在开放平台上使用多级安全审核的策略以保护系统和用户。

2.3 小结

在程序设计与架构设计领域，模式正变得越来越受人关注，许多人寄希望通过模式一劳永逸地解决自己的问题。正确使用模式可以更好地利用业界和前人的思想与实践，用更少的时间开发出更好的系统，使设计者的水平也达到更高的境界。但是模式受其适用场景限制，对系统的要求和约束也很多，不恰当地使用模式只会画虎不成反类犬，不但没有解决原来的老问题，反而带来了更棘手的新问题。

好的设计绝对不是模仿，不是生搬硬套某个模式，而是对问题深刻理解之上的创造与创新，即使是“微创新”，也是让人耳目一新的似曾相识。山寨与创新的最大区别不在于是否抄袭，是否模仿，而在于对问题和需求是否真正理解与把握。

3 大型网站核心架构要素

关于什么是**架构**，一种比较通俗的说法是**“最高层次的规划，难以改变的决定”**，这些规划和决定奠定了事物未来发展的方向和最终的蓝图。

从这个意义上说，人生规划也是一种架构。选什么学校、学什么专业、进什么公司、找什么对象，过什么样的生活，都是自己人生的架构。

具体到**软件架构**，维基百科是这样定义的：**“有关软件整体结构与组件的抽象描述，用于指导大型软件系统各个方面的设计”**。系统的各个重要组成部分及其关系构成了系统的架构，这些组成部分可以是具体的功能模块，也可以是非功能的设计与决策，他们相互关系组成一个整体，共同构成了软件系统的架构。

一般说来，除了当前的系统功能需求外，软件架构还需要关注性能、可用性、伸缩性、扩展性和安全性这 5 个架构要素，架构设计过程中需要平衡这 5 个要素之间的关系以实现需求和架构目标，也可以通过考察这些架构要素来衡量一个软件架构设计的优劣，判断其是否满足期望。

3.1 性能

性能是网站的一个重要指标，除非是没得选择（比如只能到 www.12306.cn 这一个网站上买火车票），否则用户无法忍受一个响应缓慢的网站。一个打开缓慢的网站会导致严重的用户流失，很多时候网站性能问题是网站架构升级优化的触发器。可以说性能是网站架构设计的一个重要方面，任何软件架构设计方案都必须考虑可能会带来的性能问题。

也正是因为性能问题几乎无处不在，所以优化网站性能的手段也非常多，从用户浏览器到数据库，影响用户请求的所有环节都可以进行性能优化。

在浏览器端，可以通过浏览器缓存、使用页面压缩、合理布局页面、减少 Cookie 传输等手段改善性能。

还可以使用 CDN，将网站静态内容分发至离用户最近的网络服务商机房，使用户通过最短访问路径获取数据。可以在网站机房部署反向代理服务器，缓存热点文件，加快请求响应速度，减轻应用服务器负载压力。

在应用服务器端，可以使用服务器本地缓存和分布式缓存，通过缓存在内存中的热点数据处理用户请求，加快请求处理过程，减轻数据库负载压力。

也可以通过异步操作将用户请求发送至消息队列等待后续任务处理，而当前请求直接返回响应给用户。

在网站有很多用户高并发请求的情况下，可以将多台应用服务器组成一个集群共同对外服务，提高整体处理能力，改善性能。

在代码层面，也可以通过使用多线程、改善内存管理等手段优化性能。

在数据库服务器端，索引、缓存、SQL 优化等性能优化手段都已经比较成熟。而方兴未艾的 NoSQL 数据库通过优化数据模型、存储结构、伸缩特性等手段在性能方面的优势也日趋明显。

衡量网站性能有一系列指标，重要的有响应时间、TPS、系统性能计数器等，通过测试这些指标以确定系统设计是否达到目标。这些指标也是网站监控的重要参数，通过监控这些指标可以分析系统瓶颈，预测网站容量，并对异常指标进行报警，保障系统可用

性。

对于网站而言，性能符合预期仅仅是必要条件，因为无法预知网站可能会面临的访问压力，所以必须要考察系统在高并发访问情况下，超出负载设计能力的情况下可能会出现的性能问题。网站需要长时间持续运行，还必须保证系统在持续运行且访问压力不均匀的情况下保持稳定的性能特性。

3.2 可用性

对于大型网站而言，特别是知名网站，网站宕掉、服务不可用是一个重大的事故，轻则影响网站声誉，重则可能会摊上官司。对于电子商务类网站，网站不可用还意味着损失金钱和用户。因此几乎所有网站都承诺 7×24 可用，但事实上任何网站都不可能达到完全的 7×24 可用，总会有一些故障时间，扣除这些故障时间，就是网站的总可用时间，这个时间可以换算成网站的可用性指标，以此衡量网站的可用性，一些知名大型网站可以做到 4 个 9 以上的可用性，也就是可用性超过 99.99%。

因为网站使用的服务器硬件通常是普通的商用服务器，这些服务器的设计目标本身并不保证高可用，也就是说，很有可能会出现服务器硬件故障，也就是俗称的服务器宕机。大型网站通常都会有上万台服务器，每天都必定会有一些服务器宕机，因此网站高可用架构设计的前提是必然会出现服务器宕机，而高可用设计的目标就是当服务器宕机的时候，服务或者应用依然可用。

网站高可用的主要手段是冗余，应用部署在多台服务器上同时提供访问，数据存储在多台服务器上互相备份，任何一台服务器宕机都不会影响应用的整体可用，也不会导致数据丢失。

对于应用服务器而言，多台应用服务器通过负载均衡设备组成一个集群共同对外提供服务，任何一台服务器宕机，只需把请求切换到其他服务器就可实现应用的高可用，但是一个前提条件是应用服务器上不能保存请求的会话信息。否则服务器宕机，会话丢失，即使将用户请求转发到其他服务器上也无法完成业务处理。

对于存储服务器，由于其上存储着数据，需要对数据进行实时备份，当服务器宕机时需要将数据访问转移到可用的服务器上，并进行数据恢复以保证继续有服务器宕机的

时候数据依然可用。

除了运行环境，网站的高可用还需要软件开发过程的质量保证。通过预发布验证、自动化测试、自动化发布、灰度发布等手段，减少将故障引入线上环境的可能，避免故障范围扩大。

衡量一个系统架构设计是否满足高可用的目标，就是假设系统中任何一台或者多台服务器宕机时，以及出现各种不可预期的问题时，系统整体是否依然可用。

3.3 伸缩性

大型网站需要面对大量用户的高并发访问和存储海量数据，不可能只用一台服务器就处理全部用户请求，存储全部数据。网站通过集群的方式将多台服务器组成一个整体共同提供服务。所谓伸缩性是指通过不断向集群中加入服务器的手段来缓解不断上升的用户并发访问压力和不断增长的数据存储需求。

衡量架构伸缩性的主要标准就是是否可以用多台服务器构建集群，是否容易向集群中添加新的服务器。加入新的服务器后是否可以提供和原来的服务器无差别的服务。集群中可容纳的总的服务器数量是否有限制。

对于应用服务器集群，只要服务器上不保存数据，所有服务器都是对等的，通过使用合适的负载均衡设备就可以向集群中不断加入服务器。

对于缓存服务器集群，加入新的服务器可能会导致缓存路由失效，进而导致集群中大部分缓存数据都无法访问。虽然缓存的数据可以通过数据库重新加载，但是如果应用已经严重依赖缓存，可能会导致整个网站崩溃。需要改进缓存路由算法保证缓存数据的可访问性。

关系数据库虽然支持数据复制，主从热备等机制，但是很难做到大规模集群的可伸缩性，因此关系数据库的集群伸缩性方案必须在数据库之外实现，通过路由分区等手段将部署有多个数据库的服务器组成一个集群。

至于大部分 NoSQL 数据库产品，由于其先天就是为海量数据而生，因此其对伸缩性的支持通常都非常好，可以做到在较少运维参与的情况下实现集群规模的线性伸缩。

3.4 扩展性

不同于其他架构要素主要关注非功能性需求，网站的扩展性架构直接关注网站的功能需求。网站快速发展，功能不断扩展，如何设计网站的架构使其能够快速响应需求变化，是网站可扩展架构主要的目的。

衡量网站架构扩展性好坏的主要标准就是在网站增加新的业务产品时，是否可以实现对现有产品透明无影响，不需要任何改动或者很少改动既有业务功能就可以上线新产品。不同产品之间是否很少耦合，一个产品改动对其他产品无影响，其他产品和功能不需要受牵连进行改动。

网站可伸缩架构的主要手段是事件驱动架构和分布式服务。

事件驱动架构在网站通常利用消息队列实现，将用户请求和其他业务事件构造成消息发布到消息队列，消息的处理者作为消费者从消息队列中获取消息进行处理。通过这种方式将消息产生和消息处理分离开来，可以透明地增加新的消息生产者任务或者新的消息消费者任务。

分布式服务则是将业务和可复用服务分离开来，通过分布式服务框架调用。新增产品可以通过调用可复用的服务实现自身的业务逻辑，而对现有产品没有任何影响。可复用服务升级变更的时候，也可以通过提供多版本服务对应用实现透明升级，不需要强制应用同步变更。

大型网站为了保持市场地位，还会吸引第三方开发者，调用网站服务，使用网站数据开发周边产品，扩展网站业务。第三方开发者使用网站服务的主要途径是大型网站提供的开放平台接口。

3.5 安全性

互联网是开放的，任何人在任何地方都可以访问网站。网站的安全架构就是保护网站不受恶意访问和攻击，保护网站的重要数据不被窃取。

衡量网站安全架构的标准就是针对现存和潜在的各种攻击与窃密手段，是否有可靠

的应对策略。

3.6 小结

性能、可用性、伸缩性、扩展性和安全性是网站架构最核心的几个要素，这几个问题解决了，大型网站架构设计的大部分挑战也就克服了。因此本书第二篇即按这五个架构要素进行组织。

本章既可以看作本书第二篇的前情提要，同时也可以当做第二篇的总结和归纳，阅读本章过程中如果有任何困惑都不必纠结，请直接跳过，等读完全书后可以再回头重新回顾。

第 2 篇

架　构

4

瞬时响应：网站的高性能架构

什么叫高性能的网站？

两个网站性能架构设计方案：A 方案和 B 方案，A 方案在小于 100 个并发用户访问时，每个请求的响应时间是 1 秒，当并发请求达到 200 的时候，请求的响应时间将骤增到 10 秒。B 方案不管是 100 个并发用户访问还是 200 个并发用户访问，每个请求的响应时间都差不多是 1.5 秒。哪个方案的性能好？如果老板说“我们要改善网站的性能”，他指的是什么？

同类型的两个网站，X 网站服务器平均每个请求的处理时间是 500 毫秒，Y 网站服务器平均每个请求的处理时间是 1000 毫秒，为什么用户却反映 Y 网站的速度快呢？

网站性能是客观的指标，可以具体体现到响应时间、吞吐量等技术指标，同时也是主观的感受，而感受则是一种与具体参与者相关的微妙的东西，用户的感受和工程师的感受不同，不同的用户感受也不同。

4.1 网站性能测试

性能测试是性能优化的前提和基础，也是性能优化结果的检查和度量标准。不同视角下的网站性能有不同的标准，也有不同的优化手段。

4.1.1 不同视角下的网站性能

软件工程师说到网站性能的时候，通常和用户说的不一样。

1. 用户视角的网站性能

从用户角度，网站性能就是用户在浏览器上直观感受到的网站响应速度快还是慢。用户感受到的时间，包括用户计算机和网站服务器通信的时间、网站服务器处理的时间、用户计算机浏览器构造请求解析响应数据的时间，如图 4.1 所示。

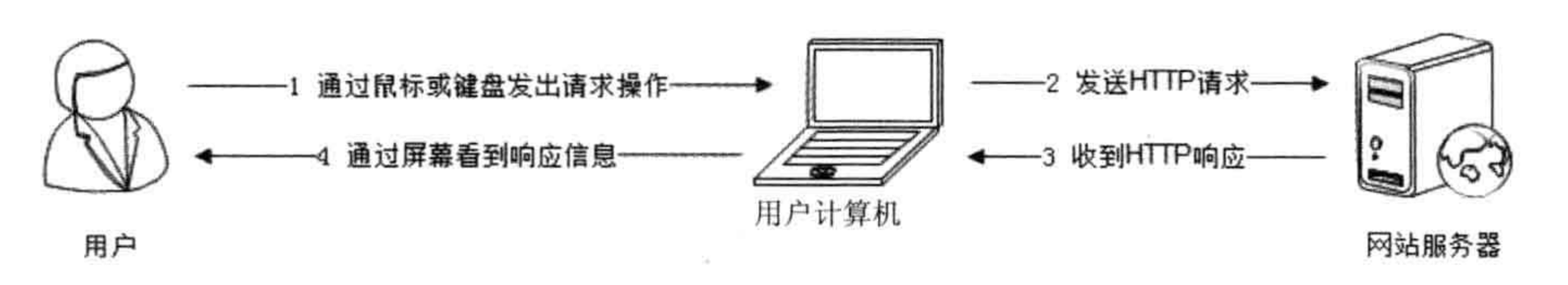

图 4.1 用户视角的网站性能

不同计算机的性能差异，不同浏览器解析 HTML 速度的差异，不同网络运营商提供的互联网宽带服务的差异，这些差异最终导致用户感受到的响应延迟可能会远远大于网站服务器处理请求需要的时间。

在实践中，使用一些前端架构优化手段，通过优化页面 HTML 式样、利用浏览器端的并发和异步特性、调整浏览器缓存策略、使用 CDN 服务、反向代理等手段，使浏览器尽快地显示用户感兴趣的内容、尽可能近地获取页面内容，即使不优化应用程序和架构，也可以很大程度地改善用户视角下的网站性能。

2. 开发人员视角的网站性能

开发人员关注的主要是应用程序本身及其相关子系统的性能，包括响应延迟、系统吞吐量、并发处理能力、系统稳定性等技术指标。主要的优化手段有使用缓存加速数据读取，使用集群提高吞吐能力，使用异步消息加快请求响应及实现削峰，使用代码优化

手段改善程序性能。

3. 运维人员视角的网站性能

运维人员更关注基础设施性能和资源利用率，如网络运营商的带宽能力、服务器硬件的配置、数据中心网络架构、服务器和网络带宽的资源利用率等。主要优化手段有建设优化骨干网、使用高性价比定制服务器、利用虚拟化技术优化资源利用等。

4.1.2 性能测试指标

不同视角下有不同的性能标准，不同的标准有不同的性能测试指标，从开发和测试人员的视角，网站性能测试的主要指标有响应时间、并发数、吞吐量、性能计数器等。

1. 响应时间

指应用执行一个操作需要的时间，包括从发出请求开始到收到最后响应数据所需要的时间。响应时间是系统最重要的性能指标，直观地反映了系统的“快慢”。表 4.1 列出了一些常用的系统操作需要的响应时间。

表 4.1 常用系统操作响应时间表

操　作	响应时间
打开一个网站	几秒
在数据库中查询一条记录（有索引）	十几毫秒
机械磁盘一次寻址定位	4 毫秒
从机械磁盘顺序读取 1MB 数据	2 毫秒
从 SSD 磁盘顺序读取 1MB 数据	0.3 毫秒
从远程分布式缓存 Redis 读取一个数据	0.5 毫秒
从内存中读取 1MB 数据	十几微秒
Java 程序本地方法调用	几微秒
网络传输 2KB 数据	1 微秒

（部分数据来源：http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html）

测试程序通过模拟应用程序，记录收到响应和发出请求之间的时间差来计算系统响应时间。但是记录及获取系统时间这个操作也需要花费一定的时间，如果测试目标操作本身需要花费的时间极少，比如几微秒，那么测试程序就无法测试得到系统的响应时间。

实践中通常采用的办法是重复请求，比如一个请求操作重复执行一万次，测试一万次执行需要的总响应时间之和，然后除以一万，得到单次请求的响应时间。

2. 并发数

指系统能够同时处理请求的数目，这个数字也反映了系统的负载特性。对于网站而言，并发数即网站并发用户数，指同时提交请求的用户数目。

与网站并发用户数相对应的还有网站在线用户数（当前登录网站的用户总数）和网站系统用户数（可能访问系统的总用户数，对多数网站而言就是注册用户数）。其数量比较关系为：

网站系统用户数>>网站在线用户数>>网站并发用户数

在网站产品设计初期，产品经理和运营人员就需要规划不同发展阶段的网站系统用户数，并以此为基础，根据产品特性和运营手段，推算在线用户数和并发用户数。这些指标将成为系统非功能设计的重要依据。

现实中，经常看到某些网站，特别是电商类网站，市场推广人员兴致勃勃地打广告打折促销，用户兴致勃勃地去抢购，结果活动刚一开始，就因为并发用户数超过网站最大负载而响应缓慢，急性子的用户不停刷新浏览器，导致系统并发数更高，最后以服务器系统崩溃，用户浏览器显示“Service is too busy”而告终。出现这种情况，有可能是网站技术准备不充分导致，也有可能是运营人员错误地评估并发用户数导致。

测试程序通过多线程模拟并发用户的办法来测试系统的并发处理能力，为了真实模拟用户行为，测试程序并不是启动多线程然后不停地发送请求，而是在两次请求之间加入一个随机等待时间，这个时间被称作思考时间。

3. 吞吐量

指单位时间内系统处理的请求数量，体现系统的整体处理能力。对于网站，可以用“请求数/秒”或是“页面数/秒”来衡量，也可以用“访问人数/天”或是“处理的业务数/小时”等来衡量。TPS（每秒事务数）是吞吐量的一个常用量化指标，此外还有 HPS（每秒 HTTP 请求数）、QPS（每秒查询数）等。

在系统并发数由小逐渐增大的过程中（这个过程也伴随着服务器系统资源消耗逐渐

增大），系统吞吐量先是逐渐增加，达到一个极限后，随着并发数的增加反而下降，达到系统崩溃点后，系统资源耗尽，吞吐量为零。

而这个过程中，响应时间则是先保持小幅上升，到达吞吐量极限后，快速上升，到达系统崩溃点后，系统失去响应。系统吞吐量、系统并发数及响应时间之间的关系将在本章后面内容中介绍。

系统吞吐量和系统并发数，以及响应时间的关系可以形象地理解为高速公路的通行状况：吞吐量是每天通过收费站的车辆数目（可以换算成收费站收取的高速费），并发数是高速公路上的正在行驶的车辆数目，响应时间是车速。车辆很少时，车速很快，但是收到的高速费也相应较少；随着高速公路上车辆数目的增多，车速略受影响，但是收到的高速费增加很快；随着车辆的继续增加，车速变得越来越慢，高速公路越来越堵，收费不增反降；如果车流量继续增加，超过某个极限后，任何偶然因素都会导致高速全部瘫痪，车走不动，费当然也收不着，而高速公路成了停车场（资源耗尽）。

网站性能优化的目的，除了改善用户体验的响应时间，还要尽量提高系统吞吐量，最大限度利用服务器资源。

4．性能计数器

它是描述服务器或操作系统性能的一些数据指标。包括 System Load、对象与线程数、内存使用、CPU 使用、磁盘与网络 I/O 等指标。这些指标也是系统监控的重要参数，对这些指标设置报警阈值，当监控系统发现性能计数器超过阈值时，就向运维和开发人员报警，及时发现处理系统异常。

System Load 即系统负载，指当前正在被 CPU 执行和等待被 CPU 执行的进程数目总和，是反映系统忙闲程度的重要指标。多核 CPU 的情况下，完美情况是所有 CPU 都在使用，没有进程在等待处理，所以 Load 的理想值是 CPU 的数目。当 Load 值低于 CPU 数目的时候，表示 CPU 有空闲，资源存在浪费；当 Load 值高于 CPU 数目的时候，表示进程在排队等待 CPU 调度，表示系统资源不足，影响应用程序的执行性能。在 Linux 系统中使用 top 命令查看，该值是三个浮点数，表示最近 1 分钟，10 分钟，15 分钟的运行队列平均进程数。如图 4.2 所示。

```
top - 16:36:49 up 1 day,  5:53,  7 users,  load average: 0.14, 0.20, 0.16
```

图 4.2 在 Linux 命令行查看系统负载

4.1.3 性能测试方法

性能测试是一个总称，具体可细分为性能测试、负载测试、压力测试、稳定性测试。

性能测试

以系统设计初期规划的性能指标为预期目标，对系统不断施加压力，验证系统在资源可接受范围内，是否能达到性能预期。

负载测试

对系统不断地增加并发请求以增加系统压力，直到系统的某项或多项性能指标达到安全临界值，如某种资源已经呈饱和状态，这时继续对系统施加压力，系统的处理能力不但不能提高，反而会下降。

压力测试

超过安全负载的情况下，对系统继续施加压力，直到系统崩溃或不能再处理任何请求，以此获得系统最大压力承受能力。

稳定性测试

被测试系统在特定硬件、软件、网络环境条件下，给系统加载一定业务压力，使系统运行一段较长时间，以此检测系统是否稳定。在不同生产环境、不同时间点的请求压力是不均匀的，呈波浪特性，因此为了更好地模拟生产环境，稳定性测试也应不均匀地对系统施加压力。

性能测试是一个不断对系统增加访问压力，以获得系统性能指标、最大负载能力、最大压力承受能力的过程。所谓的增加访问压力，在系统测试环境中，就是不断增加测试程序的并发请求数，一般说来，性能测试遵循如图 4.3 所示的抛物线规律。

图 4.3 中的横坐标表示消耗的系统资源，纵坐标表示系统处理能力（吞吐量）。在开始阶段，随着并发请求数目的增加，系统使用较少的资源就达到较好的处理能力（a～b 段），这一段是**网站的日常运行区间**，网站的绝大部分访问负载压力都集中在这一段区间，

被称作性能测试，测试目标是评估系统性能是否符合需求及设计目标；随着压力的持续增加，系统处理能力增加变缓，直到达到一个最大值（c 点），这是**系统的最大负载点**，这一段被称作负载测试。测试目标是评估当系统因为突发事件超出日常访问压力的情况下，保证系统正常运行情况下能够承受的最大访问负载压力；超过这个点后，再增加压力，系统的处理能力反而下降，而资源消耗却更多，直到资源消耗达到极限（d 点），这个点可以看作是**系统的崩溃点**，超过这个点继续加大并发请求数目，系统不能再处理任何请求，这一段被称作压力测试，测试目标是评估可能导致系统崩溃的最大访问负载压力。

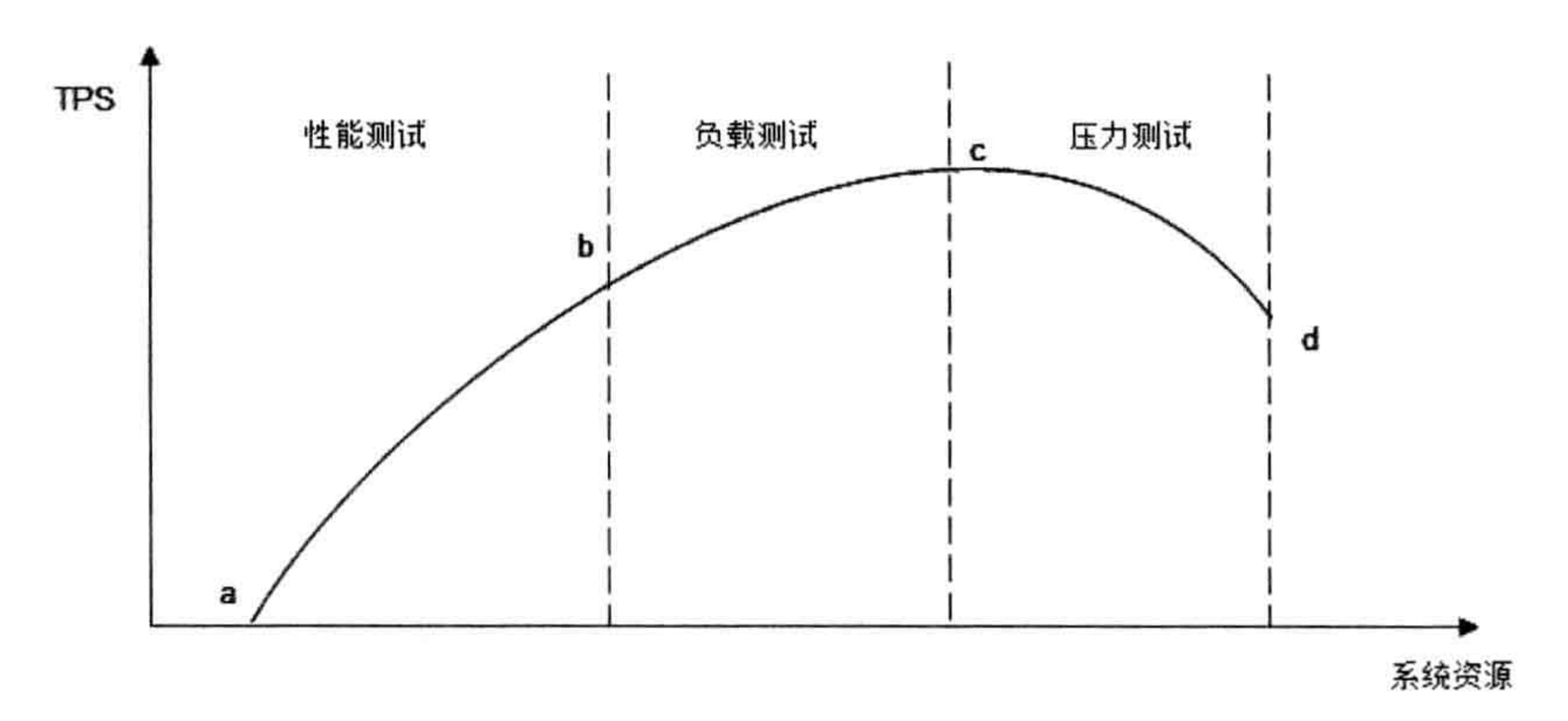

图 4.3　性能测试曲线

性能测试反应的是系统在实际生产环境中使用时，随着用户并发访问数量的增加，系统的处理能力。与性能曲线相对应的是用户访问的等待时间（系统响应时间），如图 4.4 所示。

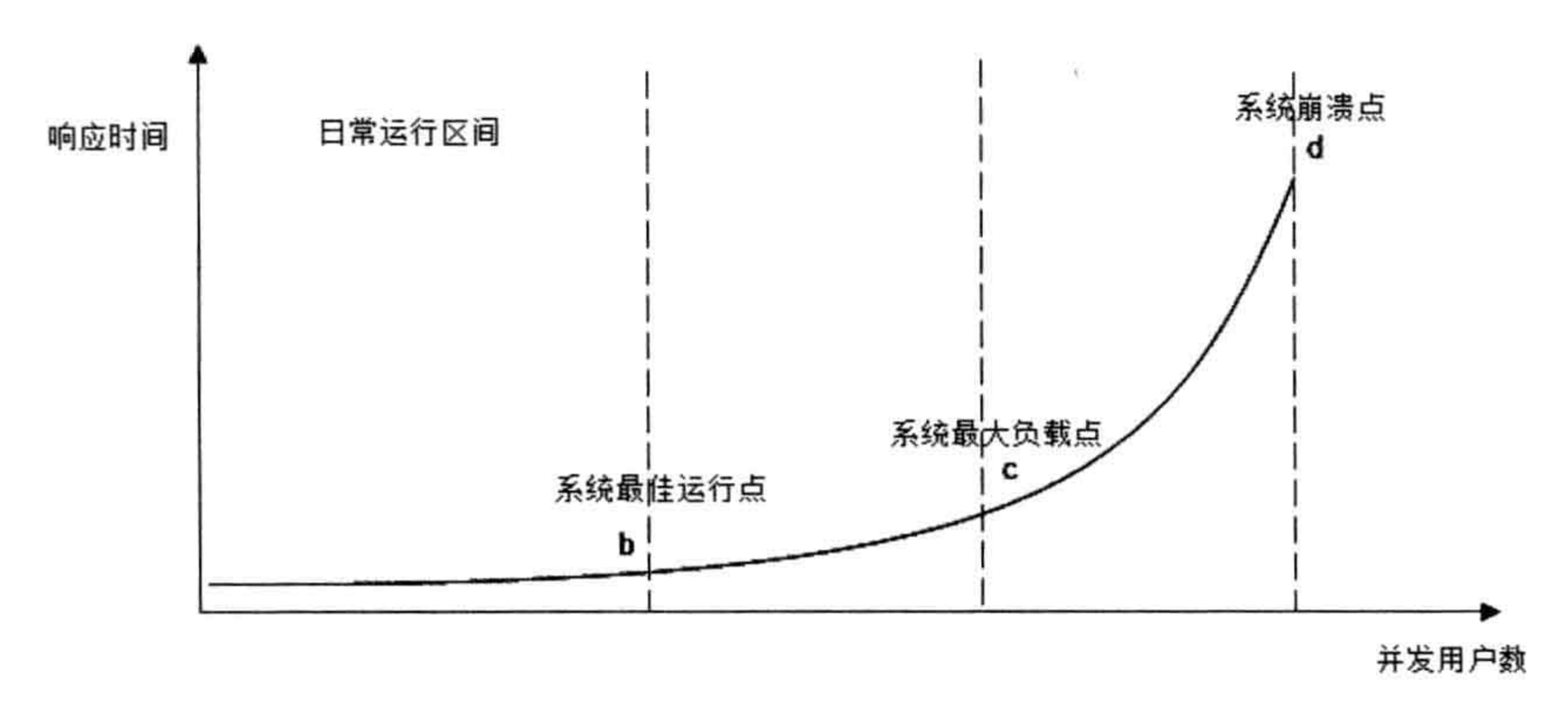

图 4.4　并发用户访问响应时间曲线

在日常运行区间，可以获得最好的用户响应时间，随着并发用户数的增加，响应延迟越来越大，直到系统崩溃，用户失去响应。

4.1.4 性能测试报告

测试结果报告应能够反映上述性能测试曲线的规律，阅读者可以得到系统性能是否满足设计目标和业务要求、系统最大负载能力、系统最大压力承受能力等重要信息，表 4.2 是一个简单示例。

表 4.2 性能测试结果报告

并 发 数	响应时间（ms）	TPS	错误率（%）	Load	内 存（GB）	备 注
10	500	20	0	5	8	性能测试
20	800	30	0	10	10	性能测试
30	1000	40	2	15	14	性能测试
40	1200	45	20	30	16	负载测试
60	2000	30	40	50	16	压力测试
80	超时	0	100	不详	不详	压力测试

4.1.5 性能优化策略

如果性能测试结果不能满足设计或业务需求，那么就需要寻找系统瓶颈，分而治之，逐步优化。

1. 性能分析

大型网站结构复杂，用户从浏览器发出请求直到数据库完成操作事务，中间需要经过很多环节，如果测试或者用户报告网站响应缓慢，存在性能问题，必须对请求经历的各个环节进行分析，排查可能出现性能瓶颈的地方，定位问题。

排查一个网站的性能瓶颈和排查一个程序的性能瓶颈的手法基本相同：检查请求处理的各个环节的日志，分析哪个环节响应时间不合理、超过预期；然后检查监控数据，分析影响性能的主要因素是内存、磁盘、网络、还是 CPU，是代码问题还是架构设计不合理，或者系统资源确实不足。

2. 性能优化

定位产生性能问题的具体原因后，就需要进行性能优化，根据网站分层架构，可分为 Web 前端性能优化、应用服务器性能优化、存储服务器性能优化 3 大类。

4.2 Web 前端性能优化

一般说来 Web 前端指网站业务逻辑之前的部分，包括浏览器加载、网站视图模型、图片服务、CDN 服务等，主要优化手段有优化浏览器访问、使用反向代理、CDN 等。

4.2.1 浏览器访问优化

1. 减少 http 请求

HTTP 协议是无状态的应用层协议，意味着每次 HTTP 请求都需要建立通信链路、进行数据传输，而在服务器端，每个 HTTP 都需要启动独立的线程去处理。这些通信和服务的开销都很昂贵，减少 HTTP 请求的数目可有效提高访问性能。

减少 HTTP 的主要手段是合并 CSS、合并 JavaScript、合并图片。将浏览器一次访问需要的 JavaScript、CSS 合并成一个文件，这样浏览器就只需要一次请求。图片也可以合并，多张图片合并成一张，如果每张图片都有不同的超链接，可通过 CSS 偏移响应鼠标点击操作，构造不同的 URL。

2. 使用浏览器缓存

对一个网站而言，CSS、JavaScript、Logo、图标这些静态资源文件更新的频率都比较低，而这些文件又几乎是每次 HTTP 请求都需要的，如果将这些文件缓存在浏览器中，可以极好地改善性能。通过设置 HTTP 头中 Cache-Control 和 Expires 的属性，可设定浏览器缓存，缓存时间可以是数天，甚至是几个月。

在某些时候，静态资源文件变化需要及时应用到客户端浏览器，这种情况，可通过改变文件名实现，即更新 JavaScript 文件并不是更新 JavaScript 文件内容，而是生成一个新的 JS 文件并更新 HTML 文件中的引用。

使用浏览器缓存策略的网站在更新静态资源时，应采用批量更新的方法，比如需要

更新 10 个图标文件，不宜把 10 个文件一次全部更新，而是应一个文件一个文件逐步更新，并有一定的间隔时间，以免用户浏览器突然大量缓存失效，集中更新缓存，造成服务器负载骤增、网络堵塞的情况。

3. 启用压缩

在服务器端对文件进行压缩，在浏览器端对文件解压缩，可有效减少通信传输的数据量。文本文件的压缩效率可达 80%以上，因此 HTML、CSS、JavaScript 文件启用 GZip 压缩可达到较好的效果。但是压缩对服务器和浏览器产生一定的压力，在通信带宽良好，而服务器资源不足的情况下要权衡考虑。

4. CSS 放在页面最上面、JavaScript 放在页面最下面

浏览器会在下载完全部 CSS 之后才对整个页面进行渲染，因此最好的做法是将 CSS 放在页面最上面，让浏览器尽快下载 CSS。JavaScript 则相反，浏览器在加载 JavaScript 后立即执行，有可能会阻塞整个页面，造成页面显示缓慢，因此 JavaScript 最好放在页面最下面。但如果页面解析时就需要用到 JavaScript，这时放在底部就不合适了。

5. 减少 Cookie 传输

一方面，Cookie 包含在每次请求和响应中，太大的 Cookie 会严重影响数据传输，因此哪些数据需要写入 Cookie 需要慎重考虑，尽量减少 Cookie 中传输的数据量。另一方面，对于某些静态资源的访问，如 CSS、Script 等，发送 Cookie 没有意义，可以考虑静态资源使用独立域名访问，避免请求静态资源时发送 Cookie，减少 Cookie 传输的次数。

4.2.2 CDN 加速

CDN（Content Distribute Network，内容分发网络）的本质仍然是一个缓存，而且将数据缓存在离用户最近的地方，使用户以最快速度获取数据，即所谓网络访问第一跳，如图 4.5 所示。

由于 CDN 部署在网络运营商的机房，这些运营商又是终端用户的网络服务提供商，因此用户请求路由的第一跳就到达了 CDN 服务器，当 CDN 中存在浏览器请求的资源时，从 CDN 直接返回给浏览器，最短路径返回响应，加快用户访问速度，减少数据中心负载压力。

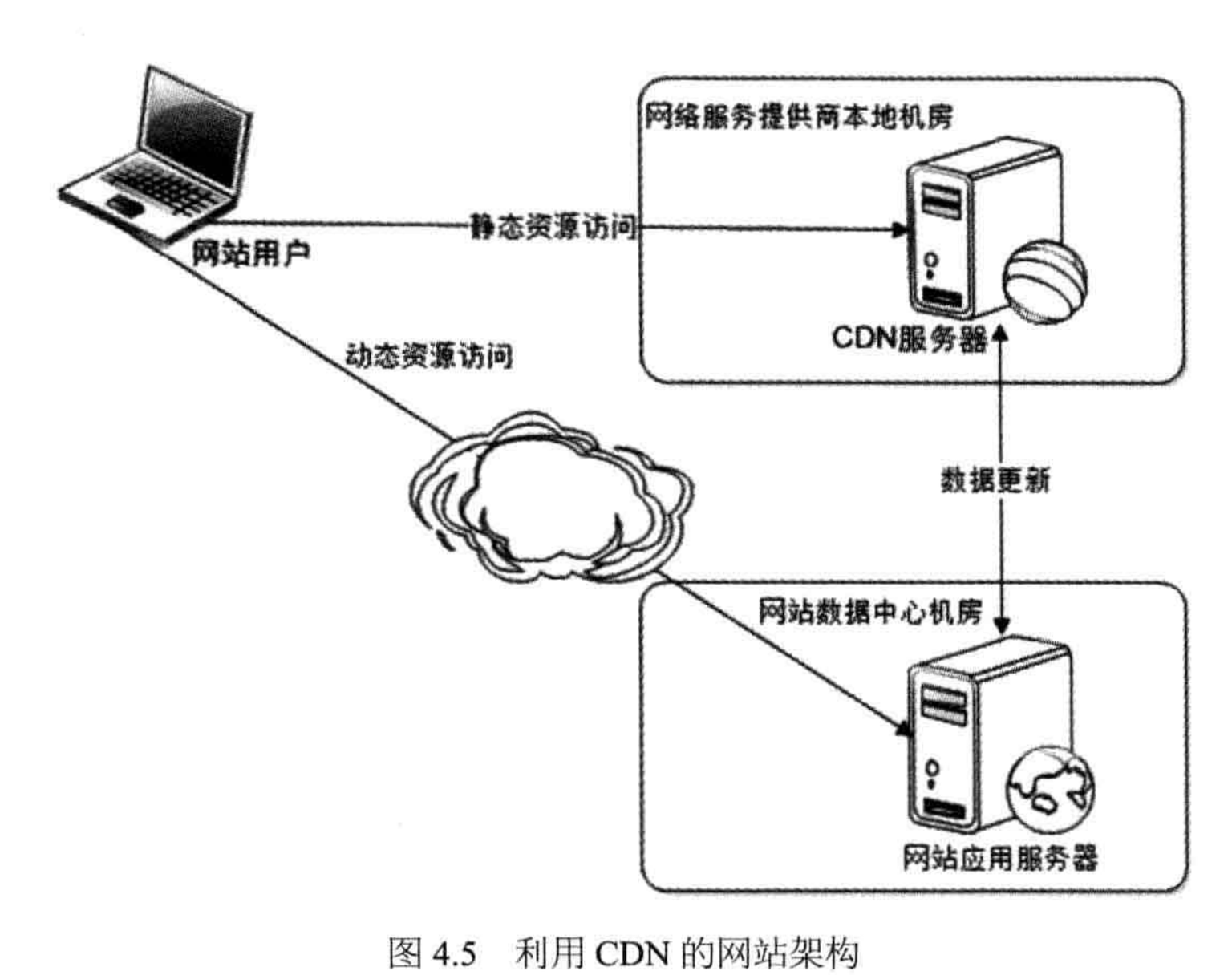

图 4.5　利用 CDN 的网站架构

CDN 能够缓存的一般是静态资源，如图片、文件、CSS、Script 脚本、静态网页等，但是这些文件访问频度很高，将其缓存在 CDN 可极大改善网页的打开速度。

4.2.3　反向代理

传统代理服务器位于浏览器一侧，代理浏览器将 HTTP 请求发送到互联网上，而反向代理服务器位于网站机房一侧，代理网站 Web 服务器接收 HTTP 请求。如图 4.6 所示。

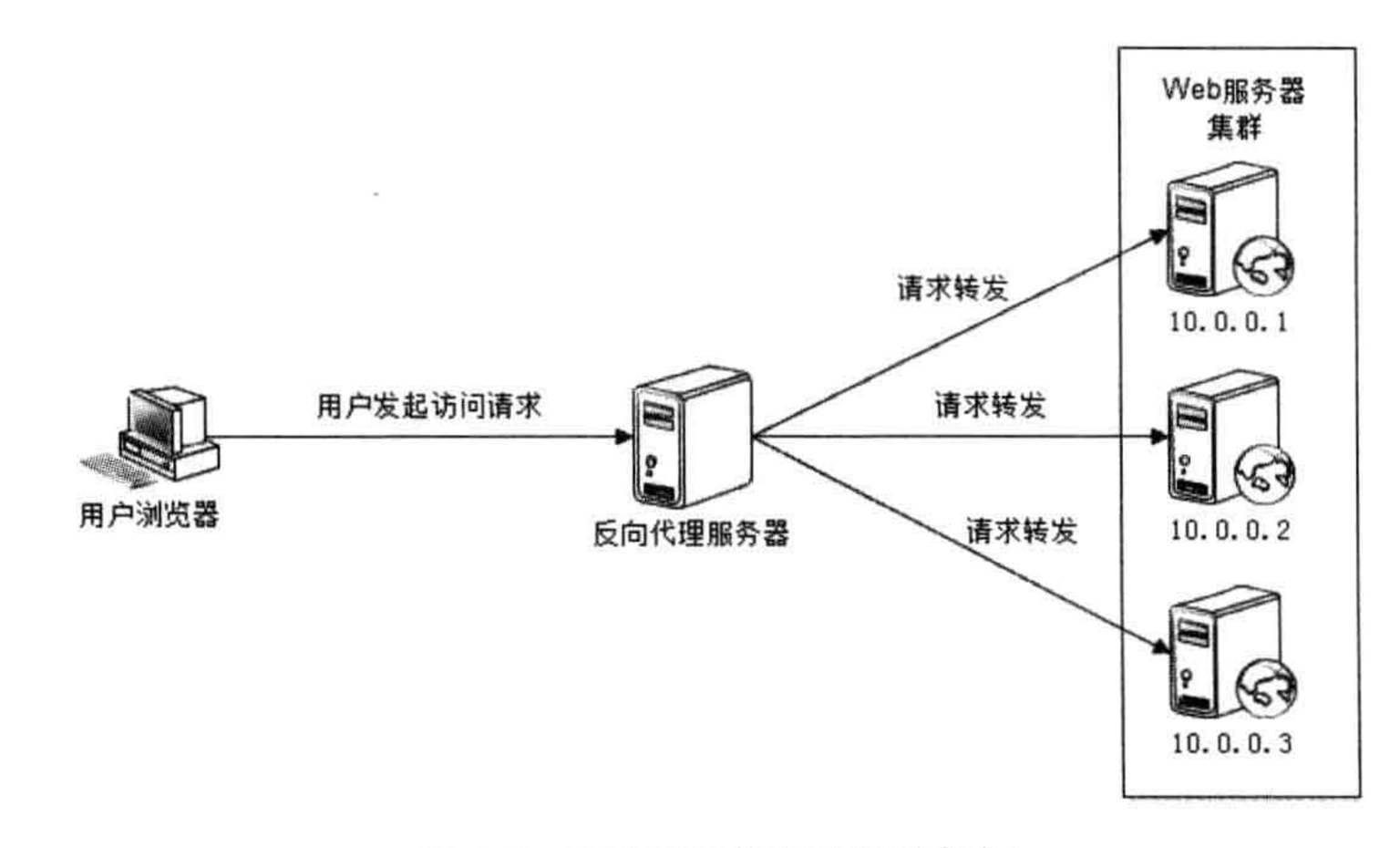

图 4.6　利用反向代理的网站架构

和传统代理服务器可以保护浏览器安全一样，反向代理服务器也具有保护网站安全

的作用，来自互联网的访问请求必须经过代理服务器，相当于在 Web 服务器和可能的网络攻击之间建立了一个屏障。

除了安全功能，代理服务器也可以通过配置缓存功能加速 Web 请求。当用户第一次访问静态内容的时候，静态内容就被缓存在反向代理服务器上，这样当其他用户访问该静态内容的时候，就可以直接从反向代理服务器返回，加速 Web 请求响应速度，减轻 Web 服务器负载压力。事实上，有些网站会把动态内容也缓存在代理服务器上，比如维基百科及某些博客论坛网站，把热门词条、帖子、博客缓存在反向代理服务器上加速用户访问速度，当这些动态内容有变化时，通过内部通知机制通知反向代理缓存失效，反向代理会重新加载最新的动态内容再次缓存起来。

此外，反向代理也可以实现负载均衡的功能，而通过负载均衡构建的应用集群可以提高系统总体处理能力，进而改善网站高并发情况下的性能。

4.3 应用服务器性能优化

应用服务器就是处理网站业务的服务器，网站的业务代码都部署在这里，是网站开发最复杂，变化最多的地方，优化手段主要有缓存、集群、异步等。

4.3.1 分布式缓存

回顾网站架构演化历程，当网站遇到性能瓶颈时，第一个想到的解决方案就是使用缓存。在整个网站应用中，缓存几乎无所不在，既存在于浏览器，也存在于应用服务器和数据库服务器；既可以对数据缓存，也可以对文件缓存，还可以对页面片段缓存。合理使用缓存，对网站性能优化意义重大。

网站性能优化第一定律：优先考虑使用缓存优化性能。

---

1. 缓存的基本原理

缓存指将数据存储在相对较高访问速度的存储介质中，以供系统处理。一方面缓存访问速度快，可以减少数据访问的时间，另一方面如果缓存的数据是经过计算处理得到的，那么被缓存的数据无需重复计算即可直接使用，因此缓存还起到减少计算时间的作用。

缓存的本质是一个内存 Hash 表，网站应用中，数据缓存以一对 Key、Value 的形式存储在内存 Hash 表中。Hash 表数据读写的时间复杂度为 O（1），图 4.7 为一对 KV 在 Hash 表中的存储。

计算 KV 对中 Key 的 HashCode 对应的 Hash 表索引，可快速访问 Hash 表中的数据。许多语言支持获得任意对象的HashCode，可以把HashCode理解为对象的唯一标示符，Java 语言中 Hashcode 方法包含在根对象 Object 中，其返回值是一个 Int。然后通过 Hashcode 计算 Hash 表的索引下标，最简单的是余数法，使用 Hash 表数组长度对 Hashcode 求余，余数即为 Hash 表索引，使用该索引可直接访问得到 Hash 表中存储的 KV 对。Hash 表是软件开发中常用到的一种数据结构，其设计思想在很多场景下都可以应用。

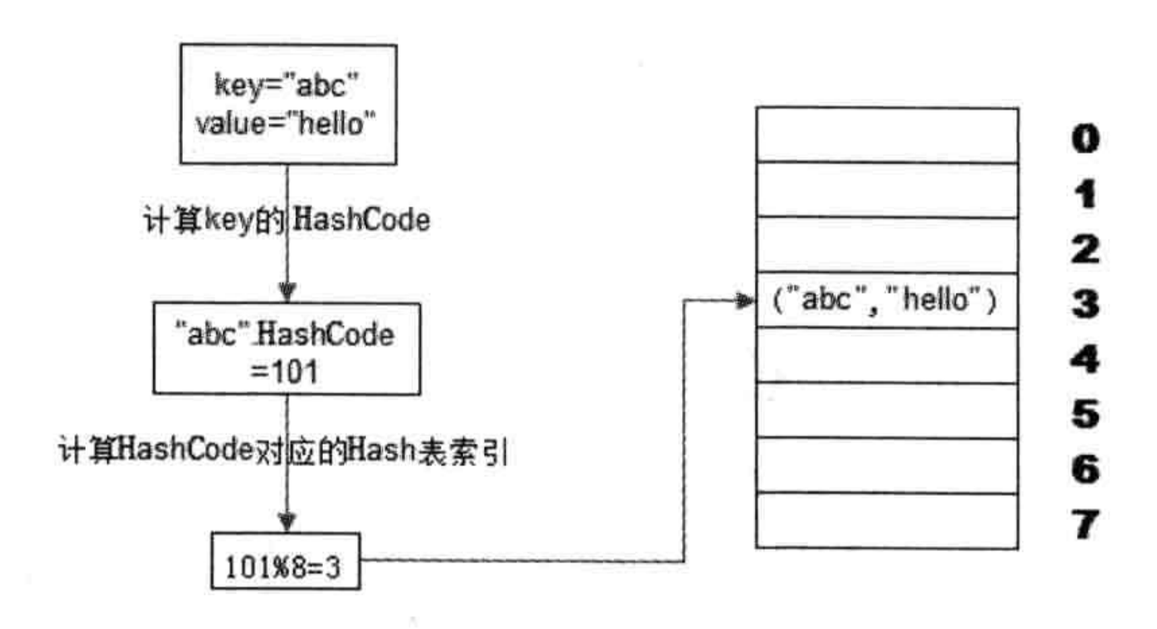

图 4.7　Hash 表存储例子

缓存主要用来存放那些读写比很高、很少变化的数据，如商品的类目信息，热门词的搜索列表信息，热门商品信息等。应用程序读取数据时，先到缓存中读取，如果读取不到或数据已失效，再访问数据库，并将数据写入缓存，如图 4.8 所示。

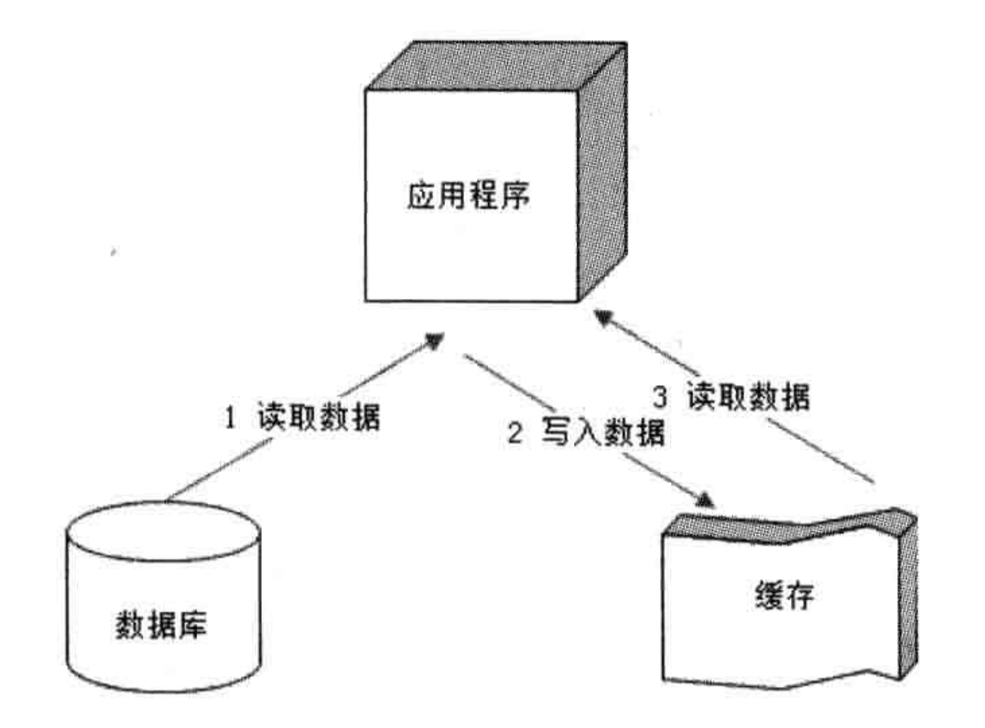

图 4.8　使用缓存存取数据

网站数据访问通常遵循二八定律，即 80%的访问落在 20%的数据上，因此利用 Hash 表和内存的高速访问特性，将这 20%的数据缓存起来，可很好地改善系统性能，提高数据读取速度，降低存储访问压力。

2. 合理使用缓存

使用缓存对提高系统性能有很多好处，但是不合理使用缓存非但不能提高系统的性能，还会成为系统的累赘，甚至风险。实践中，缓存滥用的情景屡见不鲜——过分依赖低可用的缓存系统、不恰当地使用缓存的数据访问特性等。

频繁修改的数据

如果缓存中保存的是频繁修改的数据，就会出现数据写入缓存后，应用还来不及读取缓存，数据就已失效的情形，徒增系统负担。一般说来，数据的读写比在 2:1 以上，即写入一次缓存，在数据更新前至少读取两次，缓存才有意义。实践中，这个读写比通常非常高，比如新浪微博的热门微博，缓存以后可能会被读取数百万次。

没有热点的访问

缓存使用内存作为存储，内存资源宝贵而有限，不可能将所有数据都缓存起来，只能将最新访问的数据缓存起来，而将历史数据清理出缓存。如果应用系统访问数据没有热点，不遵循二八定律，即大部分数据访问并没有集中在小部分数据上，那么缓存就没有意义，因为大部分数据还没有被再次访问就已经被挤出缓存了。

数据不一致与脏读

一般会对缓存的数据设置失效时间，一旦超过失效时间，就要从数据库中重新加载。因此应用要容忍一定时间的数据不一致，如卖家已经编辑了商品属性，但是需要过一段时间才能被买家看到。在互联网应用中，这种延迟通常是可以接受的，但是具体应用仍需慎重对待。还有一种策略是数据更新时立即更新缓存，不过这也会带来更多系统开销和事务一致性的问题。

缓存可用性

缓存是为提高数据读取性能的，缓存数据丢失或者缓存不可用不会影响到应用程序的处理——它可以从数据库直接获取数据。但是随着业务的发展，缓存会承担大部分数

据访问的压力，数据库已经习惯了有缓存的日子，所以当缓存服务崩溃时，数据库会因为完全不能承受如此大的压力而宕机，进而导致整个网站不可用。这种情况被称作缓存雪崩，发生这种故障，甚至不能简单地重启缓存服务器和数据库服务器来恢复网站访问。

实践中，有的网站通过缓存热备等手段提高缓存可用性：当某台缓存服务器宕机时，将缓存访问切换到热备服务器上。但是这种设计显然有违缓存的初衷，缓存根本就不应该被当做一个可靠的数据源来使用。

通过分布式缓存服务器集群，将缓存数据分布到集群多台服务器上可在一定程度上改善缓存的可用性。当一台缓存服务器宕机的时候，只有部分缓存数据丢失，重新从数据库加载这部分数据不会对数据库产生很大影响。

产品在设计之初就需要一个明确的定位：什么是产品要实现的功能，什么不是产品提供的特性。在产品漫长的生命周期中，会有形形色色的困难和诱惑来改变产品的发展方向，左右摇摆、什么都想做的产品，最后有可能成为一个失去生命力的四不像。

缓存预热

缓存中存放的是热点数据，热点数据又是缓存系统利用 LRU（最近最久未用算法）对不断访问的数据筛选淘汰出来的，这个过程需要花费较长的时间。新启动的缓存系统如果没有任何数据，在重建缓存数据的过程中，系统的性能和数据库负载都不太好，那么最好在缓存系统启动时就把热点数据加载好，这个缓存预加载手段叫作缓存预热（warm up）。对于一些元数据如城市地名列表、类目信息，可以在启动时加载数据库中全部数据到缓存进行预热。

缓存穿透

如果因为不恰当的业务、或者恶意攻击持续高并发地请求某个不存在的数据，由于缓存没有保存该数据，所有的请求都会落到数据库上，会对数据库造成很大压力，甚至崩溃。一个简单的对策是将不存在的数据也缓存起来（其 value 值为 null）。

3. 分布式缓存架构

分布式缓存指缓存部署在多个服务器组成的集群中，以集群方式提供缓存服务，其架构方式有两种，一种是以 JBoss Cache 为代表的需要更新同步的分布式缓存，一种是以 Memcached 为代表的不互相通信的分布式缓存。

JBoss Cache 的分布式缓存在集群中所有服务器中保存相同的缓存数据，当某台服务器有缓存数据更新的时候，会通知集群中其他机器更新缓存数据或清除缓存数据，如图 4.9 所示。JBoss Cache 通常将应用程序和缓存部署在同一台服务器上，应用程序可从本地快速获取缓存数据，但是这种方式带来的问题是缓存数据的数量受限于单一服务器的内存空间，而且当集群规模较大的时候，缓存更新信息需要同步到集群所有机器，其代价惊人。因而这种方案更多见于企业应用系统中，而很少在大型网站使用。

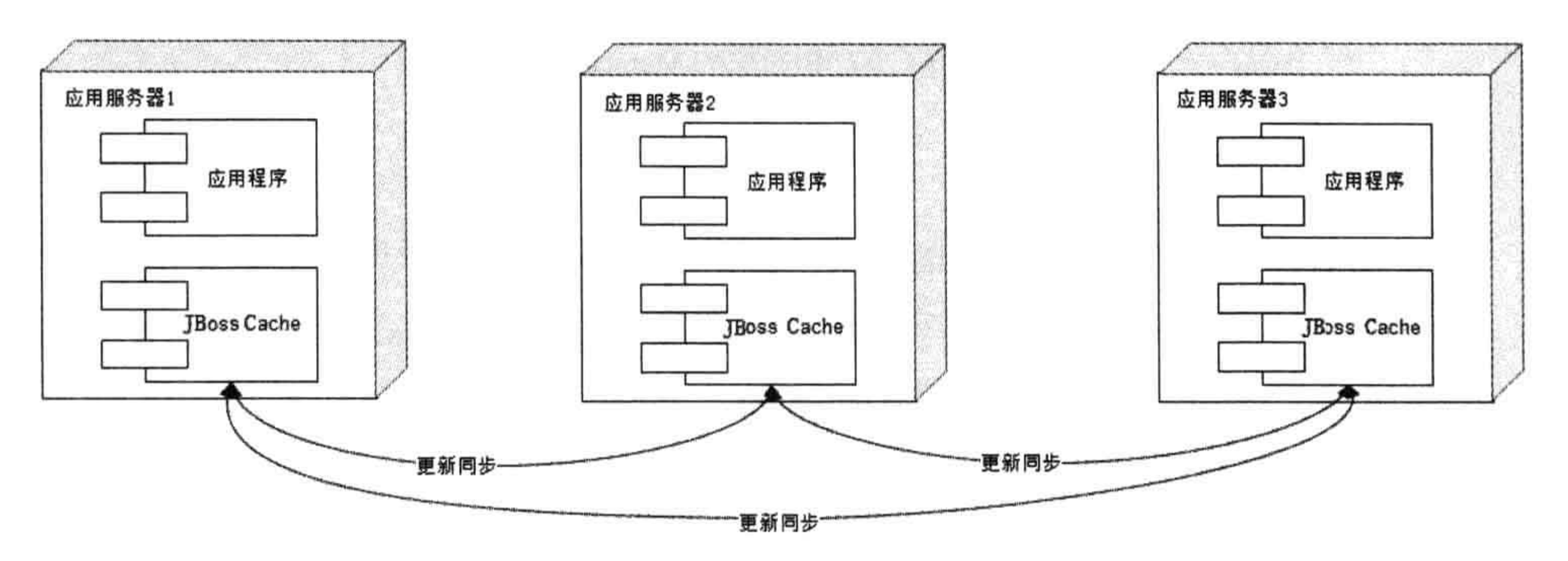

图 4.9 需要更新同步的 JBoss Cache

大型网站需要缓存的数据量一般都很庞大，可能会需要数 TB 的内存做缓存，这时候就需要另一种分布式缓存，如图 4.10 所示。Memcached 采用一种集中式的缓存集群管理，也被称作互不通信的分布式架构方式。缓存与应用分离部署，缓存系统部署在一组专门的服务器上，应用程序通过一致性 Hash 等路由算法选择缓存服务器远程访问缓存数据，缓存服务器之间不通信，缓存集群的规模可以很容易地实现扩容，具有良好的可伸缩性。

Memcached 的伸缩性设计参考本书第 6 章内容。

4. Memcached

Memcached 曾一度是网站分布式缓存的代名词，被大量网站使用。其简单的设计、优异的性能、互不通信的服务器集群、海量数据可伸缩的架构令网站架构师们趋之若鹜。

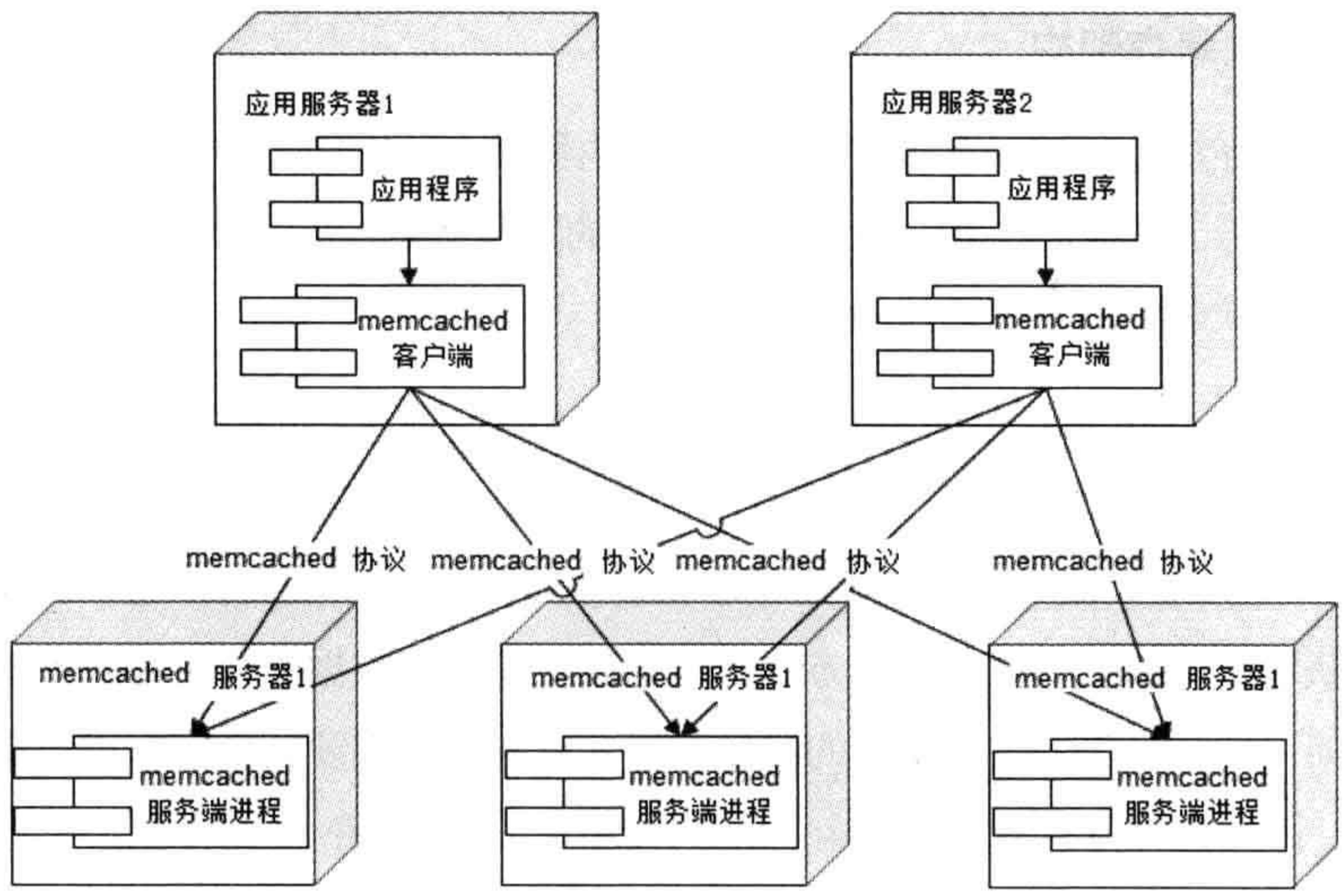

图 4.10 不互相通信的 Memcached

简单的通信协议

远程通信设计需要考虑两方面的要素，一是通信协议，即选择 TCP 协议还是 UDP 协议，抑或 HTTP 协议；一是通信序列化协议，数据传输的两端，必须使用彼此可识别的数据序列化方式才能使通信得以完成，如 XML、JSON 等文本序列化协议，或者 Google Protobuffer 等二进制序列化协议。Memcached 使用 TCP 协议（UDP 也支持）通信，其序列化协议则是一套基于文本的自定义协议，非常简单，以一个命令关键字开头，后面是一组命令操作数。例如读取一个数据的命令协议是 get <key>。Memcached 以后，许多 NoSQL 产品都借鉴了或直接支持这套协议。

丰富的客户端程序

Memcached 通信协议非常简单，只要支持该协议的客户端都可以和 Memcached 服务器通信，因此 Memcached 发展出非常丰富的客户端程序，几乎支持所有主流的网站编程语言，Java、C/C++/C#、Perl、Python、PHP、Ruby 等，因此在混合使用多种编程语言的网站，Memcached 更是如鱼得水。

高性能的网络通信

Memcached 服务端通信模块基于 Libevent，一个支持事件触发的网络通信程序库。

Libevent 的设计和实现有许多值得改善的地方，但它在稳定的长连接方面的表现却正是 Memcached 需要的。

高效的内存管理

内存管理中一个令人头痛的问题就是内存碎片管理。操作系统、虚拟机垃圾回收在这方面想了许多办法：压缩、复制等。Memcached 使用了一个非常简单的办法——固定空间分配。Memcached 将内存空间分为一组 slab，每个 slab 里又包含一组 chunk，同一个 slab 里的每个 chunk 的大小是固定的，拥有相同大小 chunk 的 slab 被组织在一起，叫作 slab_class，如图 4.11 所示。存储数据时根据数据的 Size 大小，寻找一个大于 Size 的最小 chunk 将数据写入。这种内存管理方式避免了内存碎片管理的问题，内存的分配和释放都是以 chunk 为单位的。和其他缓存一样，Memcached 采用 LRU 算法释放最近最久未被访问的数据占用的空间，释放的 chunk 被标记为未用，等待下一个合适大小数据的写入。

当然这种方式也会带来内存浪费的问题。数据只能存入一个比它大的 chunk 里，而一个 chunk 只能存一个数据，其他空间被浪费了。如果启动参数配置不合理，浪费会更加惊人，发现没有缓存多少数据，内存空间就用尽了。

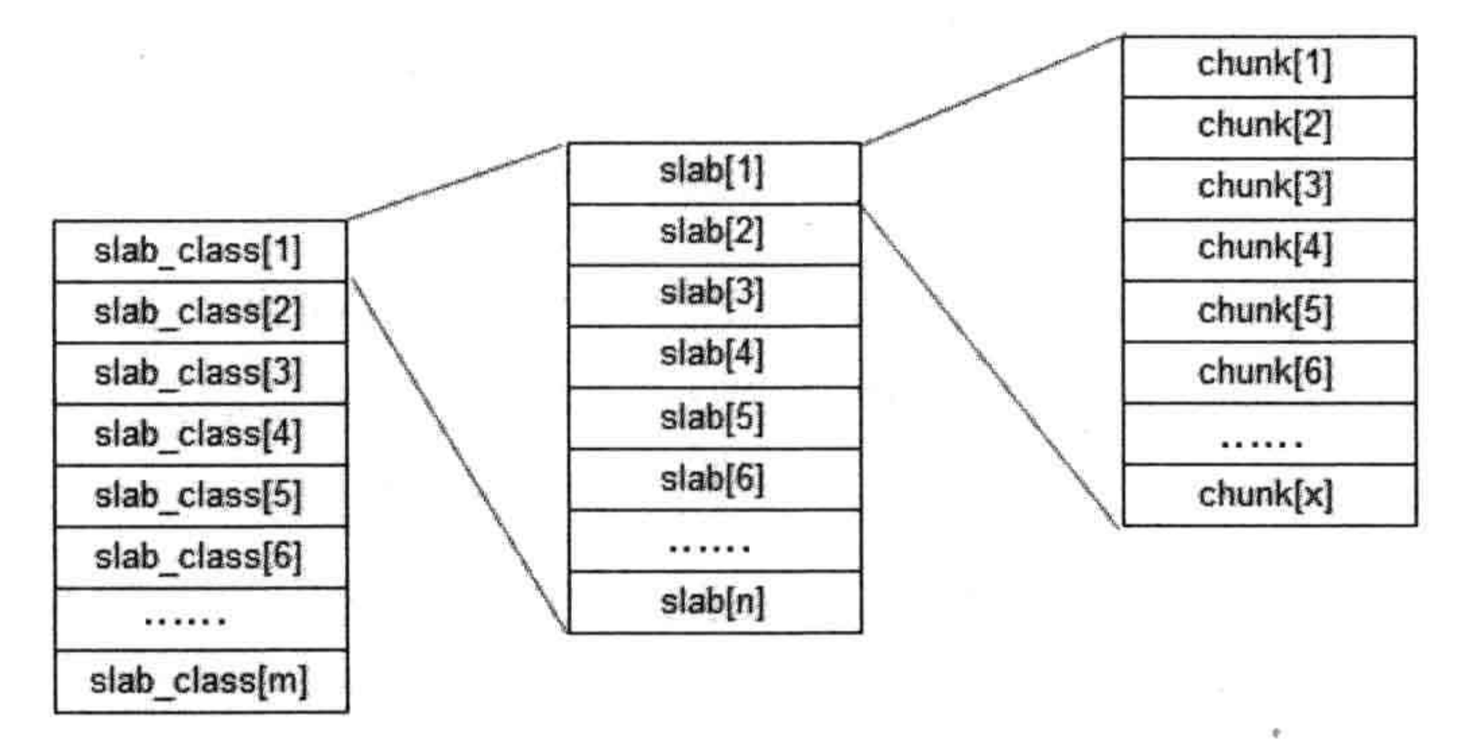

图 4.11　Memcached 内存管理

互不通信的服务器集群架构

如上所述，正是这个特性使得 Memcached 从 JBoss Cache、OSCache 等众多分布式缓存产品中脱颖而出，满足网站对海量缓存数据的需求。而其客户端路由算法一致性 Hash 更成为数据存储伸缩性架构设计的经典范式（参考本书第 6 章）。事实上，正是集群内服务器互不通信使得集群可以做到几乎无限制的线性伸缩，这也正是目前流行的许多大数

据技术的基本架构特点。

虽然近些年许多 NoSQL 产品层出不穷，在数据持久化、支持复杂数据结构、甚至性能方面有许多产品优于 Memcached，但 Memcached 由于其简单、稳定、专注的特点，仍然在分布式缓存领域占据着重要地位。

4.3.2 异步操作

使用消息队列将调用异步化，可改善网站的扩展性（参考本书第 7 章内容）。事实上，使用消息队列还可改善网站系统的性能，如图 4.12 和图 4.13 所示。

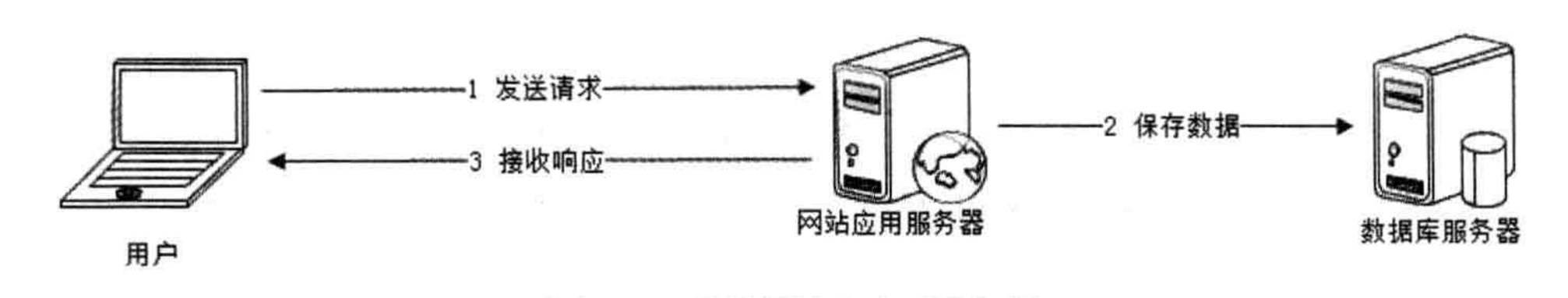

图 4.12 不使用消息队列服务器

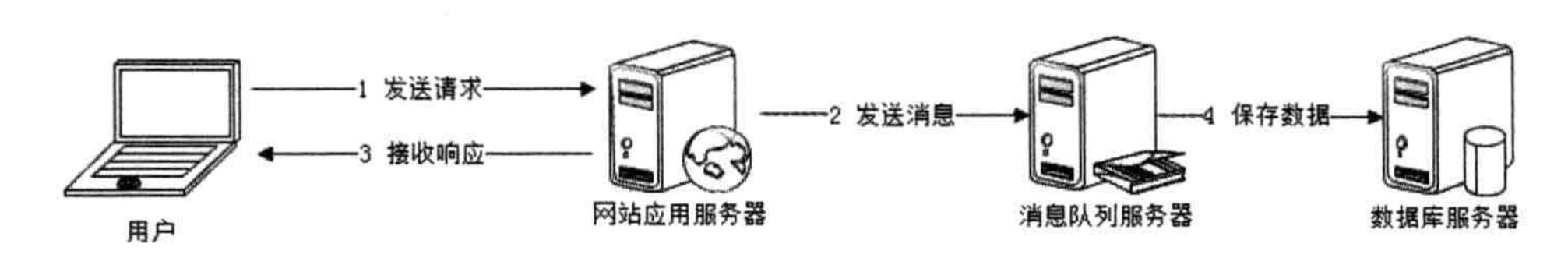

图 4.13 使用消息队列服务器

在不使用消息队列的情况下，用户的请求数据直接写入数据库，在高并发的情况下，会对数据库造成巨大的压力，同时也使得响应延迟加剧。在使用消息队列后，用户请求的数据发送给消息队列后立即返回，再由消息队列的消费者进程（通常情况下，该进程通常独立部署在专门的服务器集群上）从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度远快于数据库（消息队列服务器也比数据库具有更好的伸缩性），因此用户的响应延迟可得到有效改善。

消息队列具有很好的削峰作用——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。在电子商务网站促销活动中，合理使用消息队列，可有效抵御促销活动刚开始大量涌入的订单对系统造成的冲击。如图 4.14 所示。

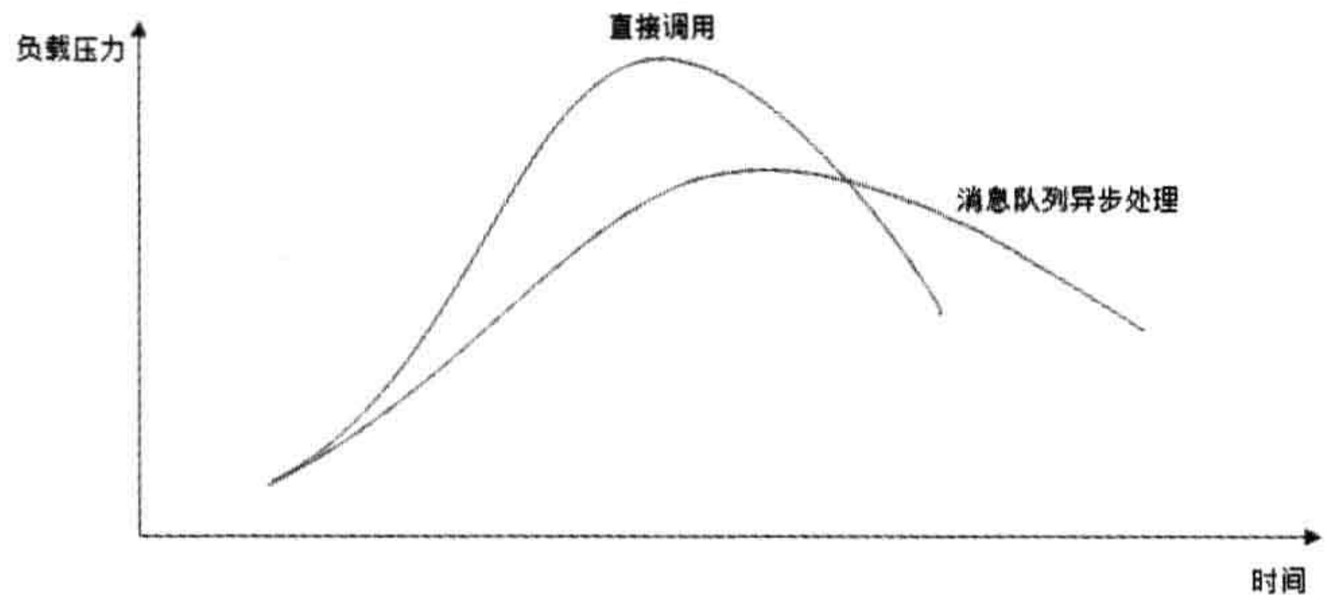

图 4.14　使用消息队列消除并发访问高峰

需要注意的是，由于数据写入消息队列后立即返回给用户，数据在后续的业务校验、写数据库等操作可能失败，因此在使用消息队列进行业务异步处理后，需要适当修改业务流程进行配合，如订单提交后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单，甚至商品出库后，再通过电子邮件或 SMS 消息通知用户订单成功，以免交易纠纷。

任何可以晚点做的事情都应该晚点再做。

4.3.3　使用集群

在网站高并发访问的场景下，使用负载均衡技术为一个应用构建一个由多台服务器组成的服务器集群，将并发访问请求分发到多台服务器上处理，避免单一服务器因负载压力过大而响应缓慢，使用户请求具有更好的响应延迟特性，如图 4.15 所示。

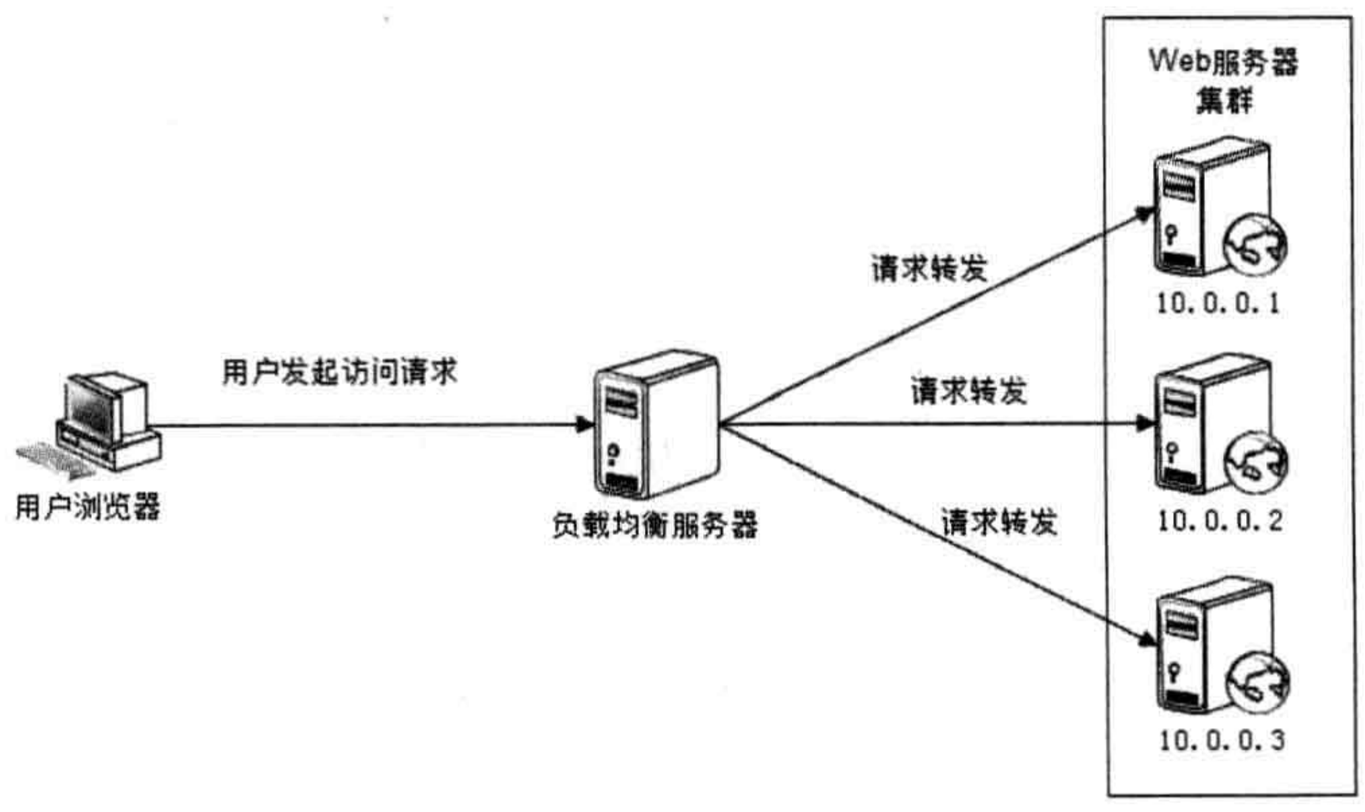

图 4.15　利用负载均衡技术改善性能

三台 Web 服务器共同处理来自用户浏览器的访问请求，这样每台 Web 服务器需要处理的 http 请求只有总并发请求数的三分之一，根据性能测试曲线，使服务器的并发请求数目控制在最佳运行区间，获得最佳的访问请求延迟。

4.3.4 代码优化

网站的业务逻辑实现代码主要部署在应用服务器上，需要处理复杂的并发事务。合理优化业务代码，可以很好地改善网站性能。不同编程语言的代码优化手段有很多，这里我们概要地关注比较重要的几个方面。

1. 多线程

多用户并发访问是网站的基本需求，大型网站的并发用户数会达到数万，单台服务器的并发用户也会达到数百。CGI 编程时代，每个用户请求都会创建一个独立的系统进程去处理。由于线程比进程更轻量，更少占有系统资源，切换代价更小，所以目前主要的 Web 应用服务器都采用多线程的方式响应并发用户请求，因此网站开发天然就是多线程编程。

从资源利用的角度看，使用多线程的原因主要有两个：IO 阻塞与多 CPU。当前线程进行 IO 处理的时候，会被阻塞释放 CPU 以等待 IO 操作完成，由于 IO 操作（不管是磁盘 IO 还是网络 IO）通常都需要较长的时间，这时 CPU 可以调度其他的线程进行处理。前面我们提到，理想的系统 Load 是既没有进程（线程）等待也没有 CPU 空闲，利用多线程 IO 阻塞与执行交替进行，可最大限度地利用 CPU 资源。使用多线程的另一个原因是服务器有多个 CPU，在这个连手机都有四核 CPU 的时代，除了最低配置的虚拟机，一般数据中心的服务器至少 16 核 CPU，要想最大限度地使用这些 CPU，必须启动多线程。

网站的应用程序一般都被 Web 服务器容器管理，用户请求的多线程也通常被 Web 服务器容器管理，但不管是 Web 容器管理的线程，还是应用程序自己创建的线程，一台服务器上启动多少线程合适呢？假设服务器上执行的都是相同类型任务，针对该类任务启动的线程数有个简化的估算公式可供参考：

启动线程数=［任务执行时间/（任务执行时间–IO 等待时间）］×CPU 内核数

最佳启动线程数和 CPU 内核数量成正比，和 IO 阻塞时间成反比。如果任务都是 CPU

计算型任务，那么线程数最多不超过 CPU 内核数，因为启动再多线程，CPU 也来不及调度；相反如果是任务需要等待磁盘操作，网络响应，那么多启动线程有助于提高任务并发度，提高系统吞吐能力，改善系统性能。

多线程编程一个需要注意的问题是线程安全问题，即多线程并发对某个资源进行修改，导致数据混乱。这也是缺乏经验的网站工程师最容易犯错的地方，而线程安全 Bug 又难以测试和重现，网站故障中，许多所谓偶然发生的“灵异事件”都和多线程并发问题有关。对网站而言，不管有没有进行多线程编程，工程师写的每一行代码都会被多线程执行，因为用户请求是并发提交的，也就是说，所有的资源——对象、内存、文件、数据库，乃至另一个线程都可能被多线程并发访问。

编程上，解决线程安全的主要手段有如下几点。

将对象设计为无状态对象：所谓无状态对象是指对象本身不存储状态信息（对象无成员变量，或者成员变量也是无状态对象），这样多线程并发访问的时候就不会出现状态不一致，Java Web 开发中常用的 Servlet 对象就设计为无状态对象，可以被应用服务器多线程并发调用处理用户请求。而 Web 开发中常用的贫血模型对象都是些无状态对象。不过从面向对象设计的角度看，无状态对象是一种不良设计。

使用局部对象：即在方法内部创建对象，这些对象会被每个进入该方法的线程创建，除非程序有意识地将这些对象传递给其他线程，否则不会出现对象被多线程并发访问的情形。

并发访问资源时使用锁：即多线程访问资源的时候，通过锁的方式使多线程并发操作转化为顺序操作，从而避免资源被并发修改。随着操作系统和编程语言的进步，出现各种轻量级锁，使得运行期线程获取锁和释放锁的代价都变得更小，但是锁导致线程同步顺序执行，可能会对系统性能产生严重影响。

2. 资源复用

系统运行时，要尽量减少那些开销很大的系统资源的创建和销毁，比如数据库连接、网络通信连接、线程、复杂对象等。从编程角度，资源复用主要有两种模式：单例（Singleton）和对象池（Object Pool）。

单例虽然是 GoF 经典设计模式中较多被诟病的一个模式，但由于目前 Web 开发中主要使用贫血模式，从 Service 到 Dao 都是些无状态对象，无需重复创建，使用单例模式也就自然而然了。事实上，Java 开发常用的对象容器 Spring 默认构造的对象都是单例（需要注意的是 Spring 的单例是 Spring 容器管理的单例，而不是用单例模式构造的单例）。

对象池模式通过复用对象实例，减少对象创建和资源消耗。对于数据库连接对象，每次创建连接，数据库服务端都需要创建专门的资源以应对，因此频繁创建关闭数据库连接，对数据库服务器而言是灾难性的，同时频繁创建关闭连接也需要花费较长的时间。因此在实践中，应用程序的数据库连接基本都使用连接池（Connection Pool）的方式。数据库连接对象创建好以后，将连接对象放入对象池容器中，应用程序要连接的时候，就从对象池中获取一个空闲的连接使用，使用完毕再将该对象归还到对象池中即可，不需要创建新的连接。

前面说过，对于每个 Web 请求（HTTP Request），Web 应用服务器都需要创建一个独立的线程去处理，这方面，应用服务器也采用线程池（Thread Pool）的方式。这些所谓的连接池、线程池，本质上都是对象池，即连接、线程都是对象，池管理方式也基本相同。

3. 数据结构

早期关于程序的一个定义是，程序就是数据结构+算法，数据结构对于编程的重要性不言而喻。在不同场景中合理使用恰当的数据结构，灵活组合各种数据结构改善数据读写和计算特性可极大优化程序的性能。

前面缓存部分已经描述过 Hash 表的基本原理，Hash 表的读写性能在很大程度上依赖 HashCode 的随机性，即 HashCode 越随机散列，Hash 表的冲突就越少，读写性能也就越高，目前比较好的字符串 Hash 散列算法有 Time33 算法，即对字符串逐字符迭代乘以 33，求得 Hash 值，算法原型为：

```
hash(i) = hash(i-1) * 33 + str[i]
```

Time33 虽然可以较好地解决冲突，但是有可能相似字符串的 HashCode 也比较接近，如字符串“AA”的 HashCode 是 2210，字符串“AB”的 HashCode 是 2211。这在某些应用场景是不能接受的，这种情况下，一个可行的方案是对字符串取信息指纹，再对信息指纹求 HashCode，由于字符串微小的变化就可以引起信息指纹的巨大不同，因此可以获

得较好的随机散列，如图 4.16 所示。

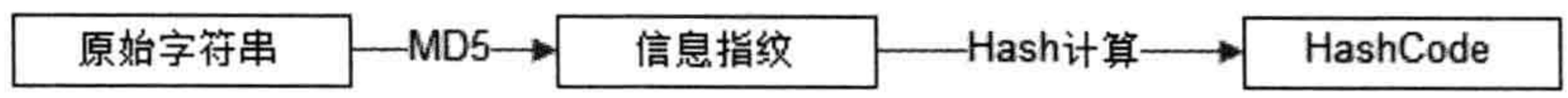

图 4.16　通过 MD5 计算 HashCode

4．垃圾回收

如果 Web 应用运行在 JVM 等具有垃圾回收功能的环境中，那么垃圾回收可能会对系统的性能特性产生巨大影响。理解垃圾回收机制有助于程序优化和参数调优，以及编写内存安全的代码。

以 JVM 为例，其内存主要可划分为堆（heap）和堆栈（stack）。堆栈用于存储线程上下文信息，如方法参数、局部变量等。堆则是存储对象的内存空间，对象的创建和释放、垃圾回收就在这里进行。通过对对象生命周期的观察，发现大部分对象的生命周期都极其短暂，这部分对象产生的垃圾应该被更快地收集，以释放内存，这就是 JVM 分代垃圾回收，其基本原理如图 4.17 所示。

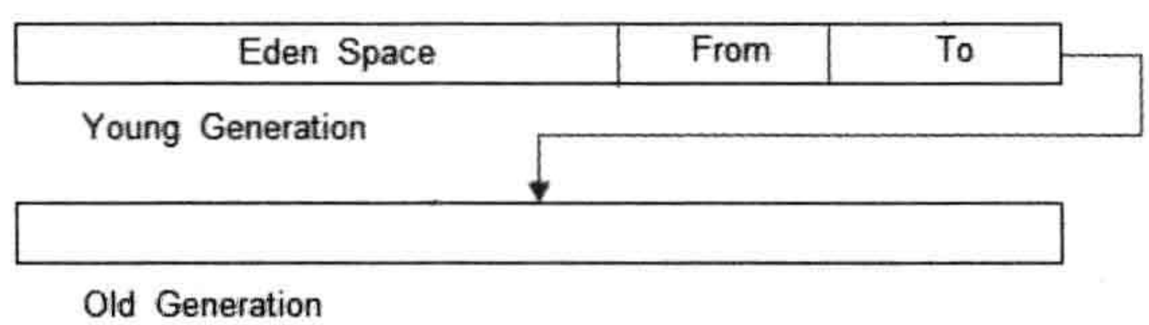

图 4.17　JVM 分代垃圾回收机制

在 JVM 分代垃圾回收机制中，将应用程序可用的堆空间分为年轻代（Young Generation）和年老代（Old Generation），又将年轻代分为 Eden 区（Eden Space）、From 区和 To 区，新建对象总是在 Eden 区中被创建，当 Eden 区空间已满，就触发一次 Young GC（Garbage Collection，垃圾回收），将还被使用的对象复制到 From 区，这样整个 Eden 区都是未被使用的空间，可供继续创建对象，当 Eden 区再次用完，再触发一次 Young GC，将 Eden 区和 From 区还在被使用的对象复制到 To 区，下一次 Young GC 则是将 Eden 区和 To 区还被使用的对象复制到 From 区。因此，经过多次 Young GC，某些对象会在 From 区和 To 区多次复制，如果超过某个阈值对象还未被释放，则将该对象复制到 Old Generation。如果 Old Generation 空间也已用完，那么就会触发 Full GC，即所谓的全量回收，全量回收会对系统性能产生较大影响，因此应根据系统业务特点和对象生命周期，

合理设置 Young Generation 和 Old Generation 大小，尽量减少 Full GC。事实上，某些 Web 应用在整个运行期间可以做到从不进行 Full GC。

4.4 存储性能优化

在网站应用中，海量的数据读写对磁盘访问造成巨大压力，虽然可以通过 Cache 解决一部分数据读压力，但是很多时候，磁盘仍然是系统最严重的瓶颈。而且磁盘中存储的数据是网站最重要的资产，磁盘的可用性和容错性也至关重要。

4.4.1 机械硬盘 vs. 固态硬盘

机械硬盘是目前最常用的一种硬盘，通过马达驱动磁头臂，带动磁头到指定的磁盘位置访问数据，由于每次访问数据都需要移动磁头臂，因此机械硬盘在数据连续访问（要访问的数据存储在连续的磁盘空间上）和随机访问（要访问的数据存储在不连续的磁盘空间）时，由于移动磁头臂的次数相差巨大，性能表现差别也非常大。机械硬盘结构如图 4.18 所示。

固态硬盘又称作 SSD 或 Flash 硬盘，这种硬盘没有机械装置，数据存储在可持久记忆的硅晶体上，因此可以像内存一样快速随机访问。而且 SSD 具有更小的功耗和更少的磁盘震动与噪声。SSD 硬盘如图 4.19 所示。

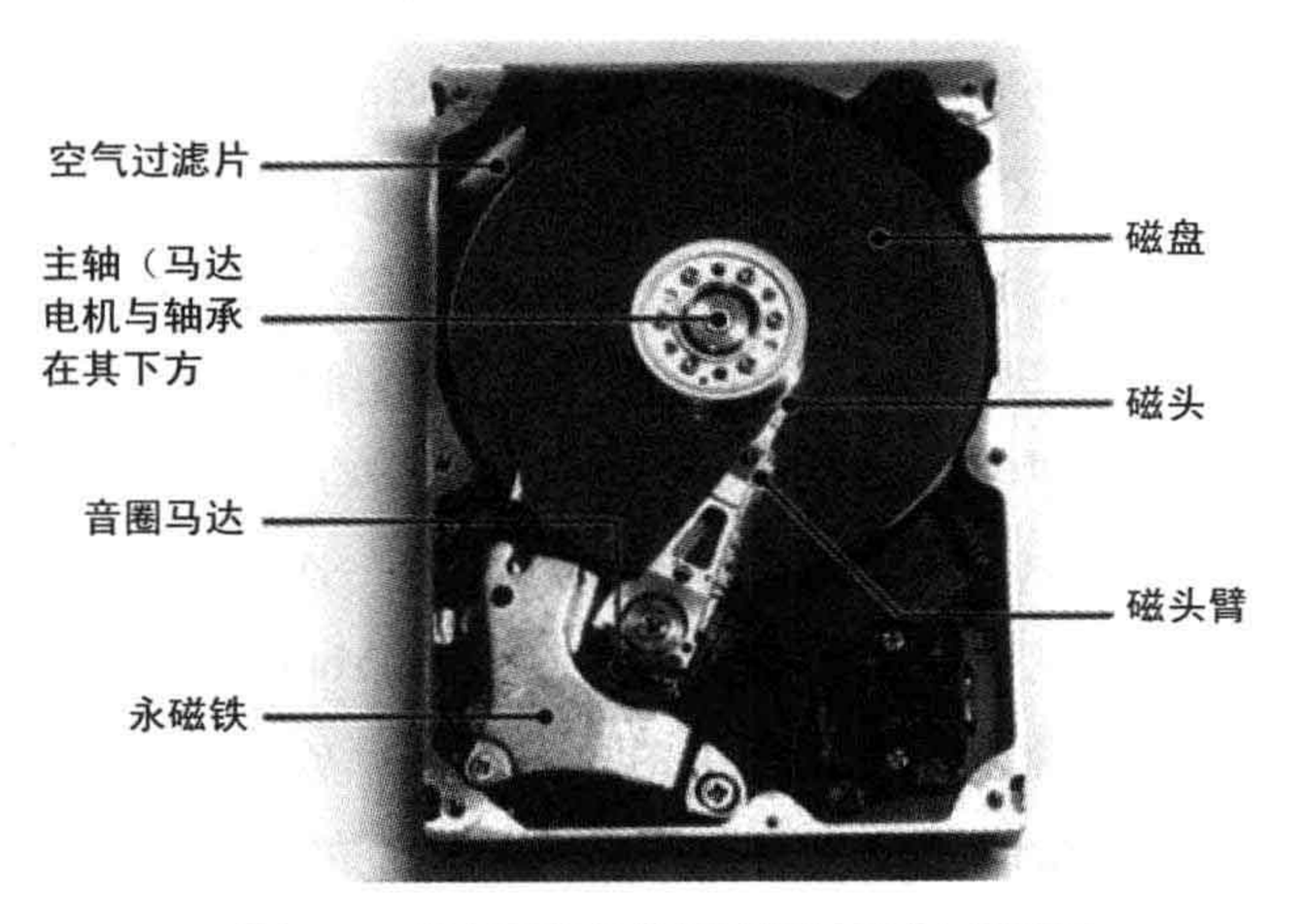

图 4.18 机械硬盘结构图（图片来自互联网）

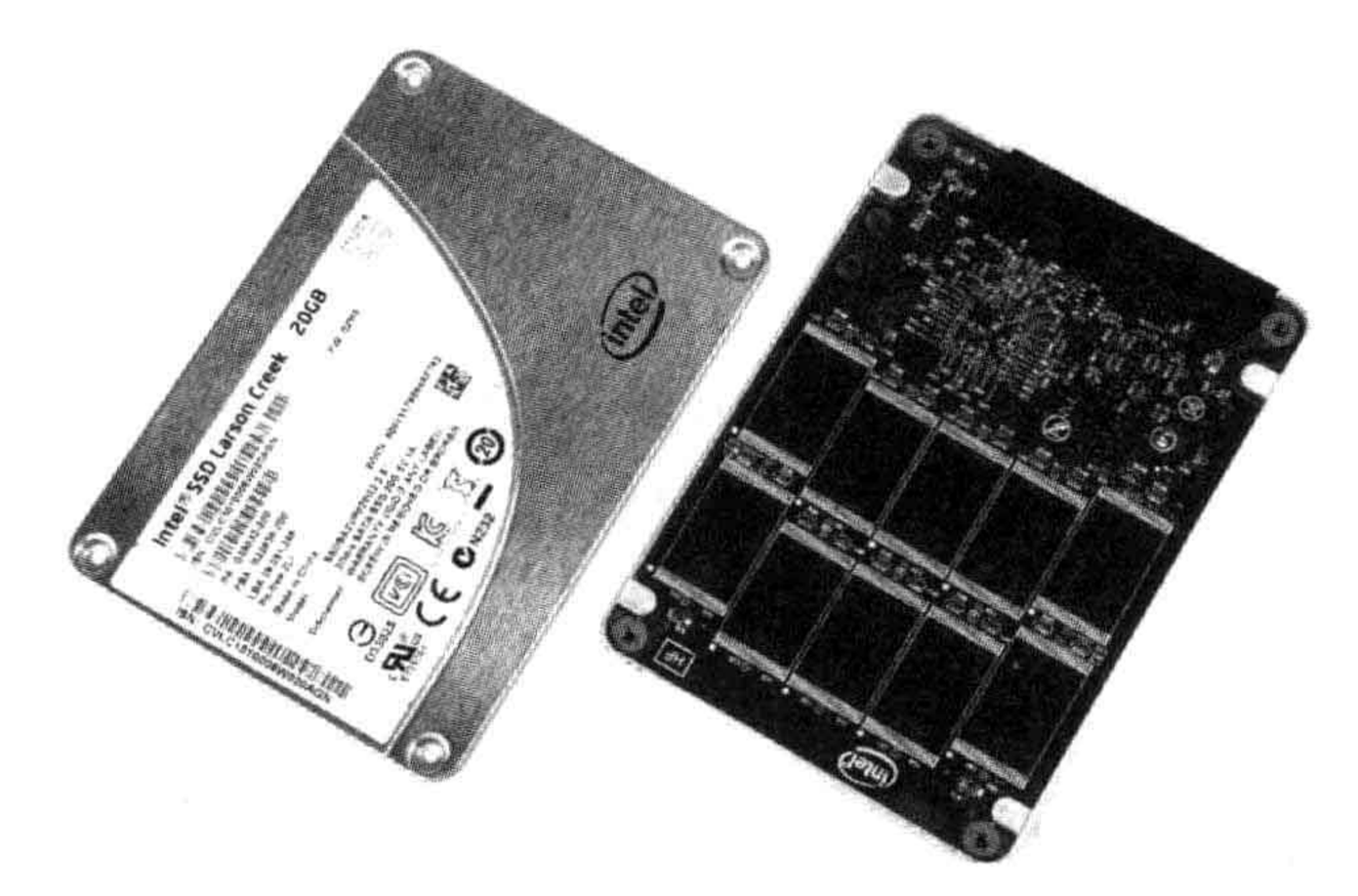

图 4.19　SSD 硬盘（图片来自互联网）

在网站应用中，大部分应用访问数据都是随机的，这种情况下 SSD 具有更好的性能表现。但是目前 SSD 硬盘还不太成熟，可靠性、性价比有待提升，因此 SSD 的使用还在摸索阶段。但是相信随着 SSD 工艺水平的提高，逐步替代传统机械硬盘是迟早的事。

4.4.2　B+树 vs. LSM 树

本书前面提到，由于传统的机械磁盘具有快速顺序读写、慢速随机读写的访问特性，这个特性对磁盘存储结构和算法的选择影响甚大。

为了改善数据访问特性，文件系统或数据库系统通常会对数据排序后存储，加快数据检索速度，这就需要保证数据在不断更新、插入、删除后依然有序，传统关系数据库的做法是使用 B+树，如图 4.20 所示。

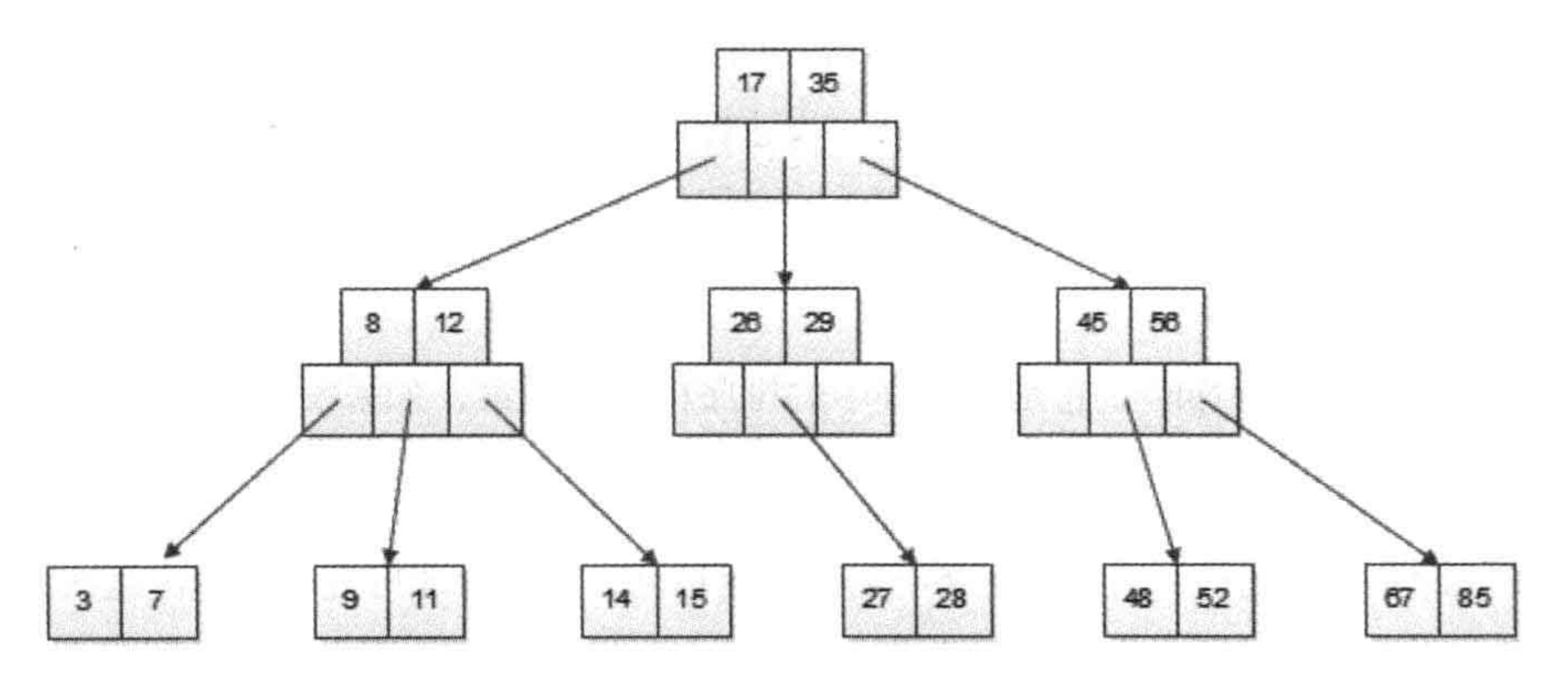

图 4.20　B+树原理示意图

B+树是一种专门针对磁盘存储而优化的 N 叉排序树，以树节点为单位存储在磁盘中，从根开始查找所需数据所在的节点编号和磁盘位置，将其加载到内存中然后继续查找，直到找到所需的数据。

目前数据库多采用两级索引的 B+树，树的层次最多三层。因此可能需要 5 次磁盘访问才能更新一条记录（三次磁盘访问获得数据索引及行 ID，然后再进行一次数据文件读操作及一次数据文件写操作）。

但是由于每次磁盘访问都是随机的，而传统机械硬盘在数据随机访问时性能较差，每次数据访问都需要多次访问磁盘影响数据访问性能。

目前许多 NoSQL 产品采用 LSM 树作为主要数据结构，如图 4.21 所示。

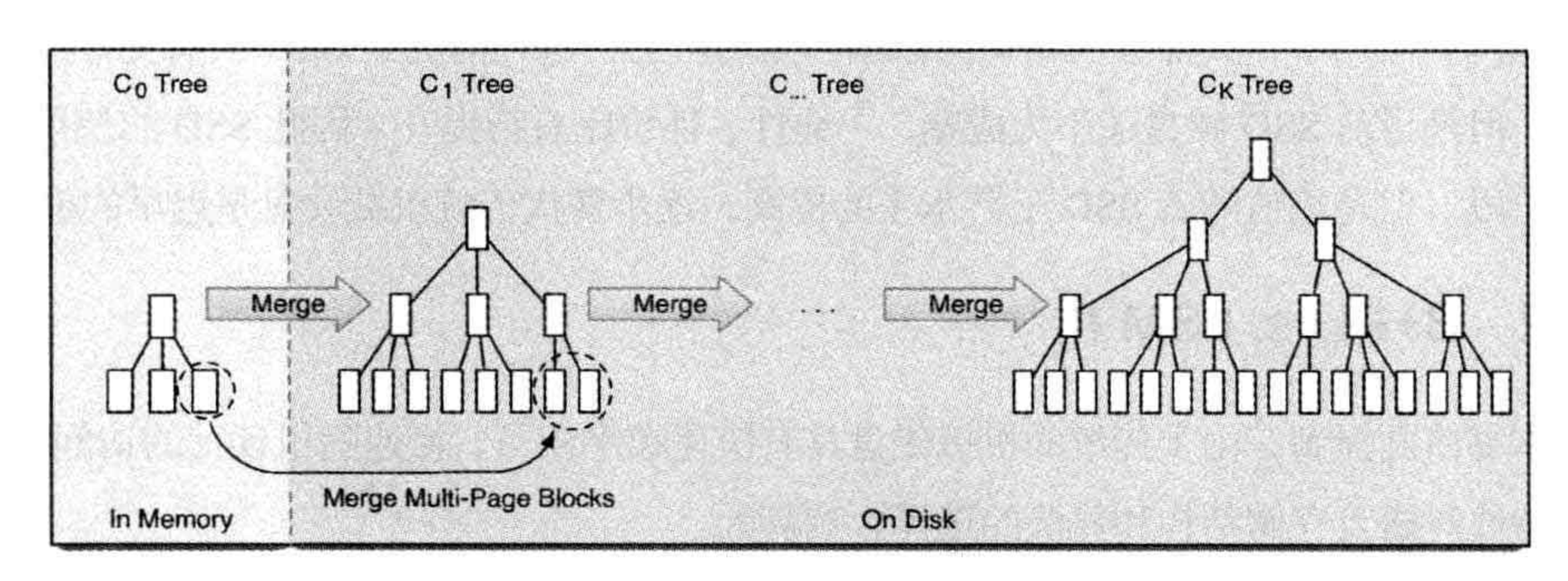

图 4.21　LSM 树原理示意图（图片来源互联网）

LSM 树可以看作是一个 N 阶合并树。数据写操作（包括插入、修改、删除）都在内存中进行，并且都会创建一个新记录（修改会记录新的数据值，而删除会记录一个删除标志），这些数据在内存中仍然还是一棵排序树，当数据量超过设定的内存阈值后，会将这棵排序树和磁盘上最新的排序树合并。当这棵排序树的数据量也超过设定阈值后，和磁盘上下一级的排序树合并。合并过程中，会用最新更新的数据覆盖旧的数据（或者记录为不同版本）。

在需要进行读操作时，总是从内存中的排序树开始搜索，如果没有找到，就从磁盘上的排序树顺序查找。

在 LSM 树上进行一次数据更新不需要磁盘访问，在内存即可完成，速度远快于 B+树。当数据访问以写操作为主，而读操作则集中在最近写入的数据上时，使用 LSM 树可

以极大程度地减少磁盘的访问次数，加快访问速度。

作为存储结构，B+树不是关系数据库所独有的，NoSQL 数据库也可以使用 B+树。同理，关系数据库也可以使用 LSM，而且随着 SSD 硬盘的日趋成熟及大容量持久存储的内存技术的出现，相信 B+树这一“古老”的存储结构会再次焕发青春。

4.4.3 RAID vs. HDFS

RAID（廉价磁盘冗余阵列）技术主要是为了改善磁盘的访问延迟，增强磁盘的可用性和容错能力。目前服务器级别的计算机都支持插入多块磁盘（8 块或者更多），通过使用 RAID 技术，实现数据在多块磁盘上的并发读写和数据备份。

常用 RAID 技术有以下几种，如图 4.22 所示。

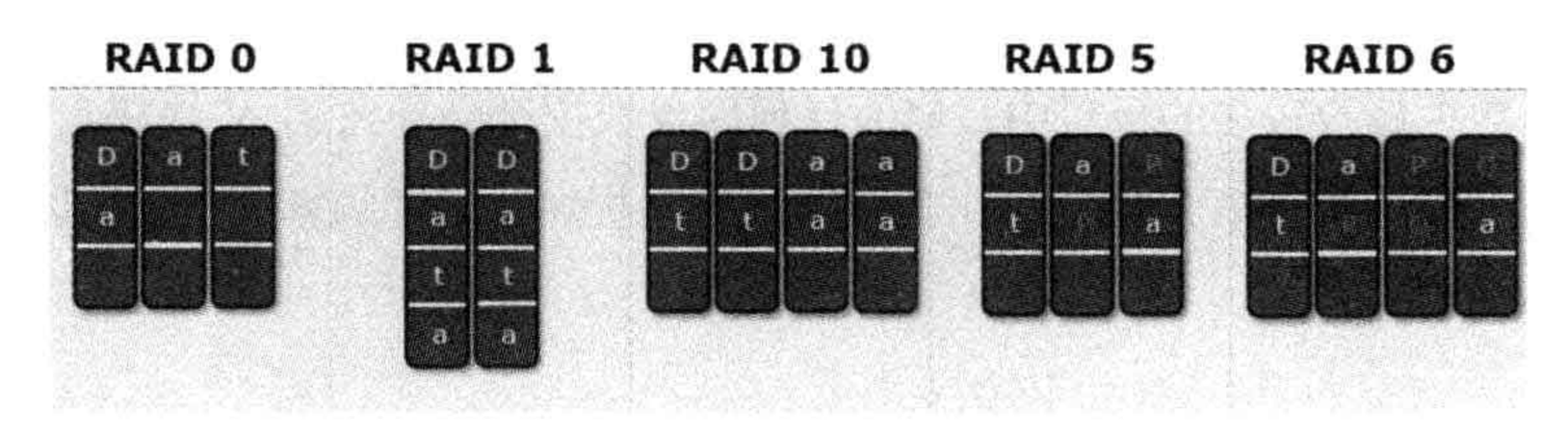

图 4.22 常用 RAID 技术原理图

假设服务器有 N 块磁盘。

RAID0

数据在从内存缓冲区写入磁盘时，根据磁盘数量将数据分成 N 份，这些数据同时并发写入 N 块磁盘，使得数据整体写入速度是一块磁盘的 N 倍。读取时也一样，因此 RAID0 具有极快的数据读写速度，但是 RAID0 不做数据备份，N 块磁盘中只要有一块损坏，数据完整性就被破坏，所有磁盘的数据都会损坏。

RAID1

数据在写入磁盘时，将一份数据同时写入两块磁盘，这样任何一块磁盘损坏都不会导致数据丢失，插入一块新磁盘就可以通过复制数据的方式自动修复，具有极高的可靠性。

RAID10

结合 RAID0 和 RAID1 两种方案，将所有磁盘平均分成两份，数据同时在两份磁盘写

入，相当于 RAID1，但是在每一份磁盘里面的 *N*/2 块磁盘上，利用 RAID0 技术并发读写，既提高可靠性又改善性能，不过 RAID10 的磁盘利用率较低，有一半的磁盘用来写备份数据。

RAID3

一般情况下，一台服务器上不会出现同时损坏两块磁盘的情况，在只损坏一块磁盘的情况下，如果能利用其他磁盘的数据恢复损坏磁盘的数据，这样在保证可靠性和性能的同时，磁盘利用率也得到大幅提升。

在数据写入磁盘的时候，将数据分成 *N*–1 份，并发写入 *N*–1 块磁盘，并在第 *N* 块磁盘记录校验数据，任何一块磁盘损坏（包括校验数据磁盘），都可以利用其他 *N*–1 块磁盘的数据修复。

但是在数据修改较多的场景中，修改任何磁盘数据都会导致第 *N* 块磁盘重写校验数据，频繁写入的后果是第 *N* 块磁盘比其他磁盘容易损坏，需要频繁更换，所以 RAID3 很少在实践中使用。

RAID5

相比 RAID3，方案 RAID5 被更多地使用。

RAID5 和 RAID3 很相似，但是校验数据不是写入第 *N* 块磁盘，而是螺旋式地写入所有磁盘中。这样校验数据的修改也被平均到所有磁盘上，避免 RAID3 频繁写坏一块磁盘的情况。

RAID6

如果数据需要很高的可靠性，在出现同时损坏两块磁盘的情况下（或者运维管理水平比较落后，坏了一块磁盘但是迟迟没有更换，导致又坏了一块磁盘），仍然需要修复数据，这时候可以使用 RAID6。

RAID6 和 RAID5 类似，但是数据只写入 *N*–2 块磁盘，并螺旋式地在两块磁盘中写入校验信息（使用不同算法生成）。

在相同磁盘数目（*N*）的情况下，各种 RAID 技术的比较如表 4.3 所示。

表 4.3 几种 RAID 技术比较

RAID 类型	访问速度	数据可靠性	磁盘利用率
RAID0	很快	很低	100%
RAID1	很慢	很高	50%
RAID10	中等	很高	50%
RAID5	较快	较高	（N–1）/N
RAID6	较快	较（RAID5）高	（N–2）/N

RAID 技术可以通过硬件实现，比如专用的 RAID 卡或者主板直接支持，也可以通过软件实现。RAID 技术在传统关系数据库及文件系统中应用比较广泛，但是在大型网站比较喜欢使用的 NoSQL，以及分布式文件系统中，RAID 技术却遭到冷落。

例如在 HDFS（Hadoop 分布式文件系统）中，系统在整个存储集群的多台服务器上进行数据并发读写和备份，可以看作在服务器集群规模上实现了类似 RAID 的功能，因此不需要磁盘 RAID。

HDFS 以块（Block）为单位管理文件内容，一个文件被分割成若干个 Block，当应用程序写文件时，每写完一个 Block，HDFS 就将其自动复制到另外两台机器上，保证每个 Block 有三个副本，即使有两台服务器宕机，数据依然可以访问，相当于实现了 RAID1 的数据复制功能。

当对文件进行处理计算时，通过 MapReduce 并发计算任务框架，可以启动多个计算子任务（MapReduce Task），同时读取文件的多个 Block，并发处理，相当于实现了 RAID0 的并发访问功能。

HDFS 架构如图 4.23 所示。

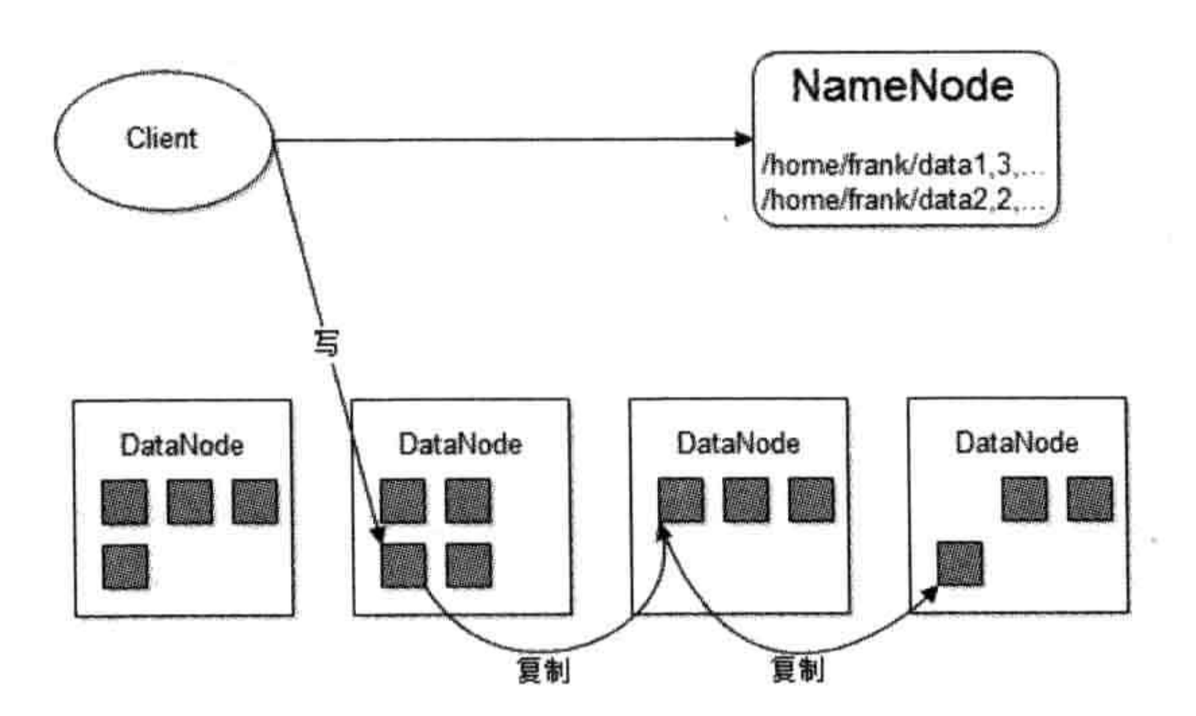

图 4.23 HDFS 架构原理图

在 HDFS 中有两种重要的服务器角色：NameNode（名字服务节点）和 DataNode（数据存储节点）。NameNode 在整个 HDFS 中只部署一个实例，提供元数据服务，相当于操作系统中的文件分配表（FAT），管理文件名 Block 的分配，维护整个文件系统的目录树结构。DataNode 则部署在 HDFS 集群中其他所有服务器上，提供真正的数据存储服务。

和操作系统一样，HDFS 对数据存储空间的管理以数据块（Block）为单位，只是比操作系统中的数据块（512 字节）要大得多，默认为 64MB。HDFS 将 DataNode 上的磁盘空间分成 *N* 个这样的块，供应用程序使用。

应用程序（Client）需要写文件时，首先访问 NameNode，请求分配数据块，NameNode 根据管理的 DataNode 服务器的磁盘空间，按照一定的负载均衡策略，分配若干数据块供 Client 使用。

当 Client 写完一个数据块时，HDFS 会将这个数据块再复制两份存储在其他 DataNode 服务器上，HDFS 默认同一份数据有三个副本，保证数据可靠性。因此在 HDFS 中，即使 DataNode 服务器有多块磁盘，也不需要使用 RAID 进行数据备份，而是在整个集群上进行数据复制，而且系统一旦发现某台服务器宕机，会自动利用其他机器上的数据将这台服务器上存储的数据块自动再备份一份，从而获得更高的数据可靠性。

HDFS 配合 MapReduce 等并行计算框架进行大数据处理时，可以在整个集群上并发读写访问所有的磁盘，无需 RAID 支持。

4.5 小结

网站性能优化技术是在网站性能遇到问题时的解决方案。而网站的性能问题很多是在用户高并发访问时产生的，所以网站性能优化的主要工作是改善高并发用户访问情况下的网站响应速度。本章开篇所举的例子，当老板说“我们要改善网站性能”的时候，他期望的是在 A 方案的基础上，不管是 100 个并发访问还是 200 个并发访问，响应时间都能达到 1 秒。而架构师能做到的，则是利用分布式的方案改善网站并发特性，由于分布式不可避免地会带来架构复杂、网络通信延迟等问题，所以最终设计出来的可能是 B 方案：缩短高并发访问响应延迟的同时，却延长了原来低并发访问时的响应延迟。架构师对这种可能性要心中有数，合理调整相关各方对性能优化的心理预期。

网站性能对最终用户而言是一种主观感受，性能优化的最终目的就是改善用户的体验，使他们感觉网站很快。离开这个目的，追求技术上的所谓高性能，是舍本逐末，没有多大意义。而用户体验的快或是慢，可以通过技术手段改善，也可以通过优化交互体验改善。

即使在技术层面，性能优化也需要全面考虑，综合权衡：性能提升一倍，但服务器数量也需要增加一倍；或者响应时间缩短，同时数据一致性也下降，这样的优化是否可以接受？这类问题的答案不是技术团队能回答的。归根结底，技术是为业务服务的，技术选型和架构决策依赖业务规划乃至企业战略规划，离开业务发展的支撑和驱动，技术走不远，甚至还会迷路。

前沿技术总是出现在前沿业务领域。近几年，以 Google 为首的互联网企业领跑 IT 前沿技术潮流，是因为互联网企业的业务发展远超传统 IT 企业领域，面临更多挑战，对 IT 系统提出了更高的要求。

新技术的出现又会驱动企业开展新的业务。亚马逊等互联网公司利用自己的技术优势进军企业级市场，以技术驱动业务，开展云计算、SaaS 等新兴 IT 业务，逐步蚕食 IBM、HP、Oracle、微软等传统软件巨头的市场。

---

5 万无一失：网站的高可用架构

2011 年 4 月 12 日，亚马逊云计算服务 EC2（Elastic Computer Cloud）发生故障，其 ESB（Elastic Block Storage）服务不可用，故障持续了数天，最终还是有部分数据未能恢复。这一故障导致美国许多使用亚马逊云服务的知名网站（如：Foursquare，Quora）受到影响，并引发了人们对使用云计算安全性、可靠性的大规模讨论。

2010 年 1 月 12 日，百度被黑客攻击，其 DNS 域名被劫持，导致百度全站长达数小时不可访问。该事件一时成为新闻焦点，各种媒体争相报道。

网站的可用性（Availability）描述网站可有效访问的特性（不同于另一个网站运营指标：Usability，通常也被译作可用性，但是后者强调的是网站的有用性，即对最终用户的使用价值），相比于网站的其他非功能特性，网站的可用性更牵动人们的神经，大型网站的不可用事故直接影响公司形象和利益，许多互联网公司都将网站可用性列入工程师的绩效考核，与奖金升迁等利益挂钩。

5.1 网站可用性的度量与考核

网站的页面能完整呈现在最终用户面前，需要经过很多个环节，任何一个环节出了问题，都可能导致网站页面不可访问。DNS 会被劫持、CDN 服务可能会挂掉、网站服务器可能会宕机、网络交换机可能会失效、硬盘会损坏、网卡会松掉、甚至机房会停电、空调会失灵、程序会有 Bug、黑客会攻击、促销会引来大量访问、第三方合作伙伴的服务会不可用……要保证一个网站永远完全可用几乎是一件不可能完成的使命。

5.1.1 网站可用性度量

网站不可用也被称作网站故障，业界通常用多少个 9 来衡量网站的可用性，如 QQ 的可用性是 4 个 9，即 QQ 服务 99.99%可用，这意味着 QQ 服务要保证其在所有运行时间中，只有 0.01%的时间不可用，也就是一年中大约最多 53 分钟不可用。

网站不可用时间（故障时间）＝故障修复时间点–故障发现（报告）时间点

网站年度可用性指标＝（1–网站不可用时间/年度总时间）×100%

对于大多数网站而言，2 个 9 是基本可用，网站年度不可用时间小于 88 小时；3 个 9 是较高可用，网站年度不可用时间小于 9 小时；4 个 9 是具有自动恢复能力的高可用，网站年度不可用时间小于 53 分钟；5 个 9 是极高可用性，网站年度不可用时间小于 5 分钟。

由于可用性影响因素很多，对于网站整体而言，达到 4 个 9，乃至 5 个 9 的可用性，除了过硬的技术、大量的设备资金投入和工程师的责任心，还要有个好运气。

常使用 Twitter 的用户或多或少遇到过那个著名的服务不可用的鲸鱼页面，事实上，Twitter 网站的可用性不足 2 个 9。

5.1.2 网站可用性考核

可用性指标是网站架构设计的重要指标，对外是服务承诺，对内是考核指标。从管理层面，可用性指标是网站或者产品的整体考核指标，具体到每个工程师的考核，更多的是使用故障分。

所谓故障分是指对网站故障进行分类加权计算故障责任的方法。表 5.1 为某网站故障

分类权重表。

表 5.1 网站故障分类权重表示例

分　类	描　述	权　重
事故级故障	严重故障，网站整体不可用	100
A 类故障	网站访问不顺畅或核心功能不可用	20
B 类故障	非核心功能不可用，或核心功能少数用户不可用	5
C 类故障	以上故障以外的其他故障	1

故障分的计算公式为：

故障分＝故障时间（分钟）× 故障权重

在年初或者考核季度的开始，会根据网站产品的可用性指标计算总的故障分，然后根据团队和个人的职责角色分摊故障分，这个可用性指标和故障分是管理预期。在实际发生故障的时候，根据故障分类和责任划分将故障产生的故障分分配给责任者承担。等年末或者考核季度末的时候，个人及团队实际承担的故障分如果超过了年初分摊的故障分，绩效考核就会受到影响。

一个简化的故障处理流程如图 5.1 所示。

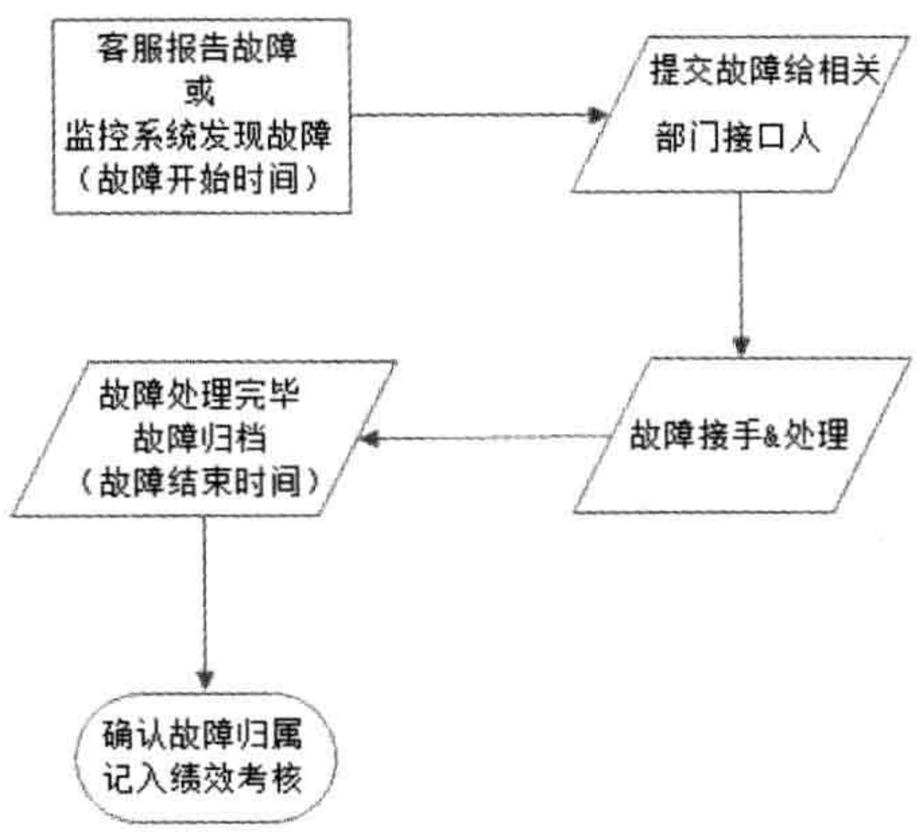

图 5.1 网站故障处理流程示例

有时候一个故障责任可能由多个部门或团队来承担，故障分也会相应按责任分摊到不同的团队和个人。

不同于其他架构指标，网站可用性更加看得见摸得着，跟技术、运营、相关各方的绩效考核息息相关，因此在架构设计与评审会议上，关于系统可用性的讨论与争执总是最花费时间与精力的部分。

当然，不同的公司有不同的企业文化和市场策略，这些因素也会影响到系统可用性的架构决策，崇尚创新和风险的企业会对可用性要求稍低一些；业务快速增长的网站忙于应对指数级增长的用户，也会降低可用性的标准；服务于收费用户的网站则会比服务于免费用户的网站对可用性更加敏感，服务不可用或关键用户数据丢失可能会导致收费用户的投诉甚至引来官司。

5.2 高可用的网站架构

通常企业级应用系统为提高系统可用性，会采用较昂贵的软硬件设备，如 IBM 的小型机乃至中型机大型机及专有操作系统、Oracle 数据库、EMC 存储设备等。互联网公司更多地采用 PC 级服务器、开源的数据库和操作系统，这些廉价的设备在节约成本的同时也降低了可用性，特别是服务器硬件设备，低价的商业级服务器一年宕机一次是一个大概率事件，而那些高强度频繁读写的普通硬盘，损坏的概率则要更高一些。

既然硬件故障是常态，网站的高可用架构设计的主要目的就是保证服务器硬件故障时服务依然可用、数据依然保存并能够被访问。

实现上述高可用架构的主要手段是数据和服务的冗余备份及失效转移，一旦某些服务器宕机，就将服务切换到其他可用的服务器上，如果磁盘损坏，则从备份的磁盘读取数据。

一个典型的网站设计通常遵循如图 5.2 所示的基本分层架构模型。

图 5.2 网站架构基本分层模型

典型的分层模型是三层，即应用层、服务层、数据层；各层之间具有相对独立性，应用层主要负责具体业务逻辑处理；服务层负责提供可复用的服务；数据层负责数据的存储与访问。中小型网站在具体部署时，通常将应用层和服务层部署在一起，而数据层则另外部署，如图 5.3 所示（事实上，这也是网站架构演化的第一步）。

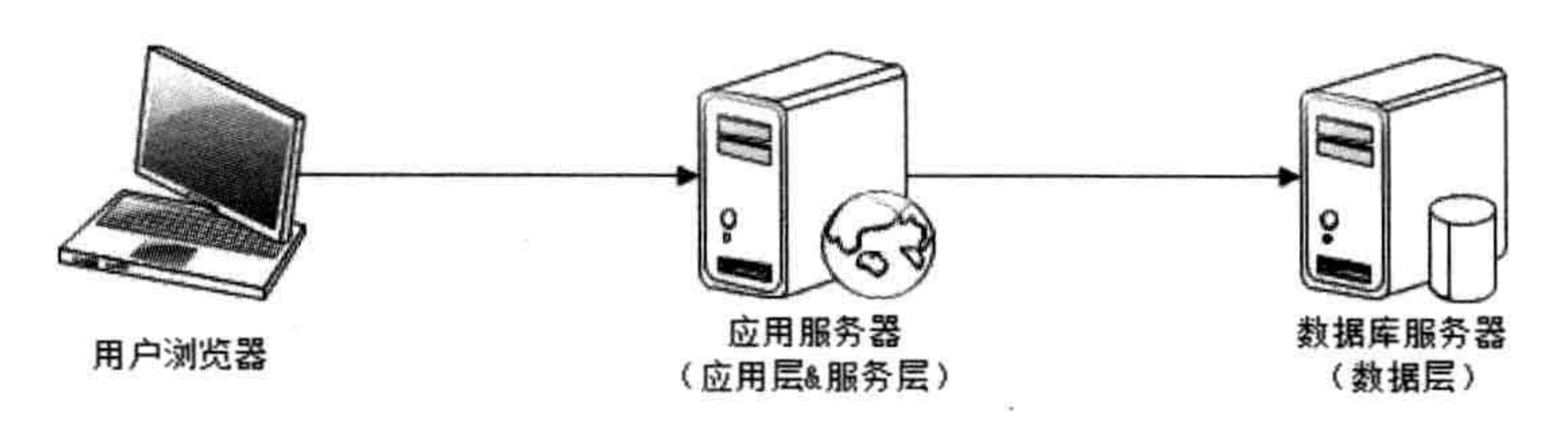

图 5.3　应用和数据分离部署的网站架构

在复杂的大型网站架构中，划分的粒度会更小、更详细，结构更加复杂，服务器规模更加庞大，但通常还是能够把这些服务器划分到这三层中。如图 5.4 所示。

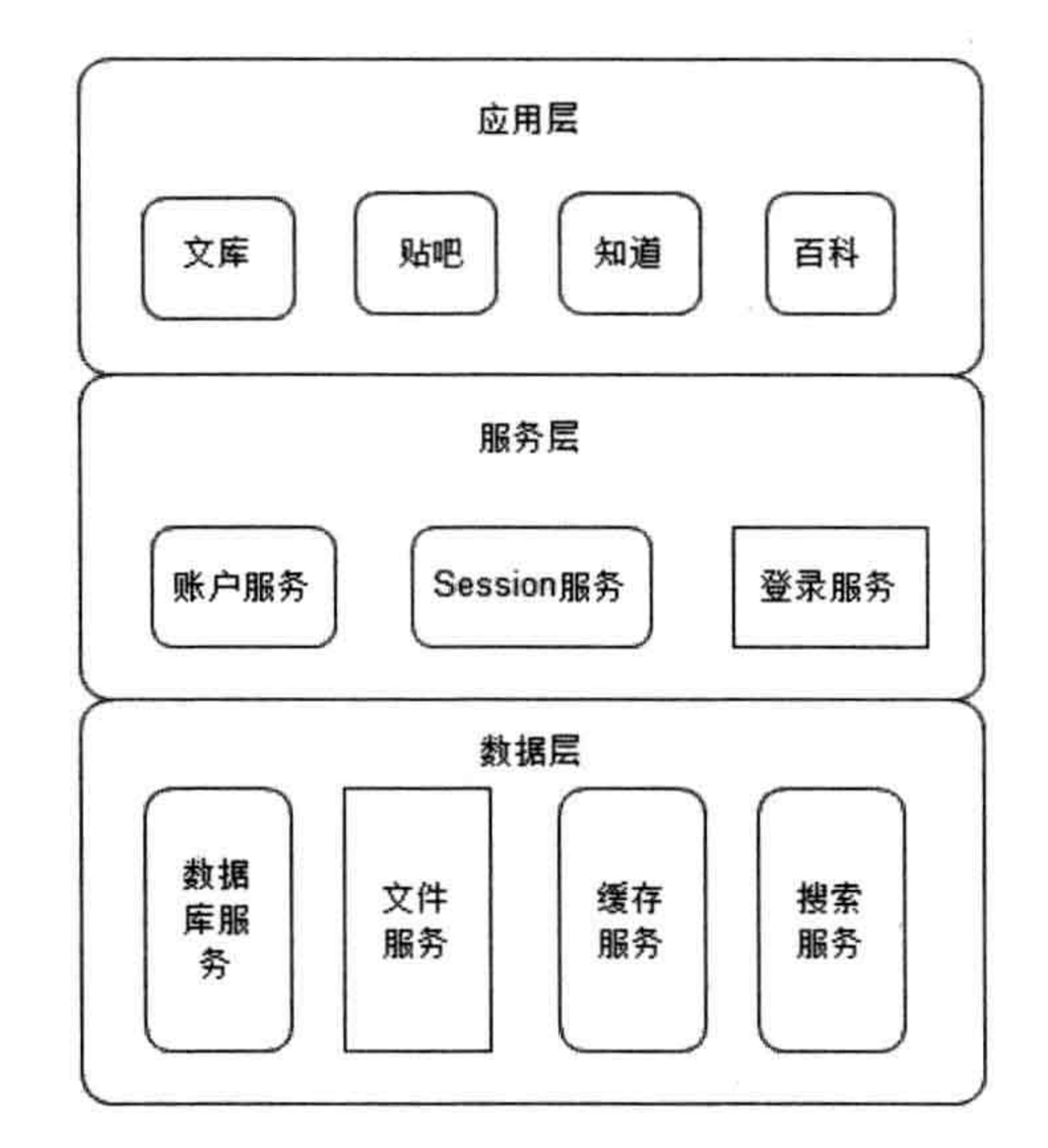

图 5.4　分层后按模块分割的网站架构模型

不同的业务产品会部署在不同的服务器集群上，如某网站的文库、贴吧、百科等属于不同的产品，部署在各自独立的服务器集群上，互不相干。这些产品又会依赖一些共同的复用业务，如注册登录服务、Session 管理服务、账户管理服务等，这些可复用的业务服务也各自部署在独立的服务器集群上。至于数据层，数据库服务、文件服务、缓存服务、搜索服务等数据存储与访问服务都部署在各自独立的服务器集群上。

大型网站的分层架构及物理服务器的分布式部署使得位于不同层次的服务器具有不同的可用性特点。关闭服务或者服务器宕机时产生的影响也不相同，高可用的解决方案也差异甚大。

位于应用层的服务器通常为了应对高并发的访问请求，会通过负载均衡设备将一组服务器组成一个集群共同对外提供服务，当负载均衡设备通过心跳检测等手段监控到某台应用服务器不可用时，就将其从集群列表中剔除，并将请求分发到集群中其他可用的服务器上，使整个集群保持可用，从而实现应用高可用。

位于服务层的服务器情况和应用层的服务器类似，也是通过集群方式实现高可用，只是这些服务器被应用层通过分布式服务调用框架访问，分布式服务调用框架会在应用层客户端程序中实现软件负载均衡，并通过服务注册中心对提供服务的服务器进行心跳检测，发现有服务不可用，立即通知客户端程序修改服务访问列表，剔除不可用的服务器。

位于数据层的服务器情况比较特殊，数据服务器上存储着数据，为了保证服务器宕机时数据不丢失，数据访问服务不中断，需要在数据写入时进行数据同步复制，将数据写入多台服务器上，实现数据冗余备份。当数据服务器宕机时，应用程序将访问切换到有备份数据的服务器上。

网站升级的频率一般都非常高，大型网站一周发布一次，中小型网站一天发布几次。每次网站发布都需要关闭服务，重新部署系统，整个过程相当于服务器宕机。因此网站的可用性架构设计不但要考虑实际的硬件故障引起的宕机，还要考虑网站升级发布引起的宕机，而后者更加频繁，不能因为系统可以接受偶尔的停机故障就降低可用性设计的标准。

5.3 高可用的应用

应用层主要处理网站应用的业务逻辑，因此有时也称作业务逻辑层，应用的一个显著特点是应用的无状态性。

所谓无状态的应用是指应用服务器不保存业务的上下文信息，而仅根据每次请求提交的数据进行相应的业务逻辑处理，多个服务实例（服务器）之间完全对等，请求提交

到任意服务器，处理结果都是完全一样的。

5.3.1 通过负载均衡进行无状态服务的失效转移

不保存状态的应用给高可用的架构设计带来了巨大便利，既然服务器不保存请求的状态，那么所有的服务器完全对等，当任意一台或多台服务器宕机，请求提交给集群中其他任意一台可用机器处理，这样对终端用户而言，请求总是能够成功的，整个系统依然可用。对于应用服务器集群，实现这种服务器可用状态实时监测、自动转移失败任务的机制是负载均衡。

负载均衡，顾名思义，主要使用在业务量和数据量较高的情况下，当单台服务器不足以承担所有的负载压力时，通过负载均衡手段，将流量和数据分摊到一个集群组成的多台服务器上，以提高整体的负载处理能力。目前，不管是开源免费的负载均衡软件还是昂贵的负载均衡硬件，都提供失效转移功能。在网站应用中，当集群中的服务是无状态对等时，负载均衡可以起到事实上高可用的作用，如图 5.5 所示。

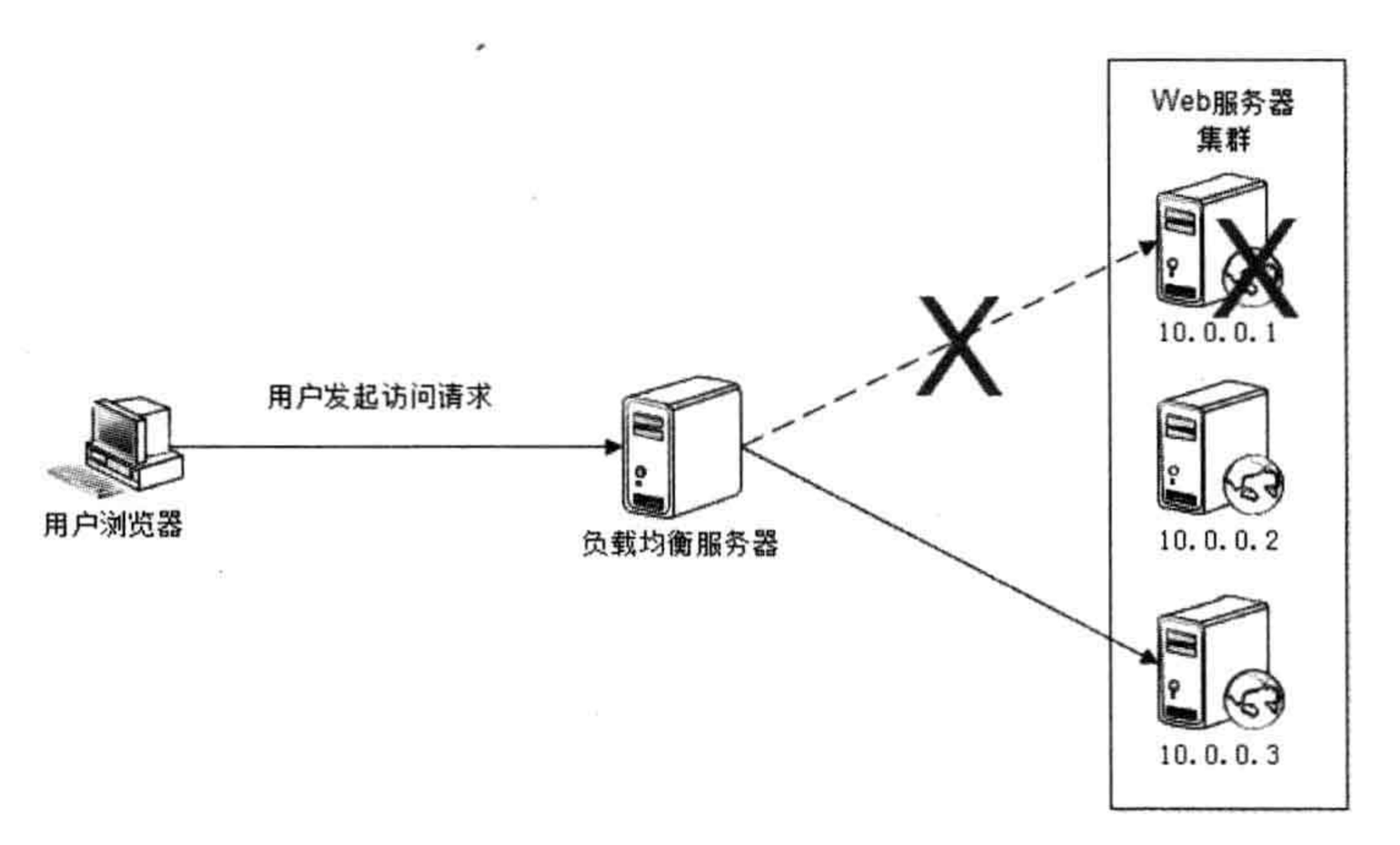

图 5.5 利用负载均衡服务器实现高可用的应用服务

当 Web 服务器集群中的服务器都可用时，负载均衡服务器会把用户发送的访问请求分发到任意一台服务器上进行处理，而当服务器 10.0.0.1 宕机时，负载均衡服务器通过心跳检测机制发现该服务器失去响应，就会把它从服务器列表中删除，而将请求发送到其他服务器上，这些服务器是完全一样的，请求在任何一台服务器中处理都不会影响最终的结果。

由于负载均衡在应用层实际上起到了系统高可用的作用，因此即使某个应用访问量非常少，只用一台服务器提供服务就绰绰有余，但如果需要保证该服务高可用，也必须至少部署两台服务器，使用负载均衡技术构建一个小型的集群。

5.3.2 应用服务器集群的 Session 管理

应用服务器的高可用架构设计主要基于服务无状态这一特性，但是事实上，业务总是有状态的，在交易类的电子商务网站，需要有购物车记录用户的购买信息，用户每次购买请求都是向购物车中增加商品；在社交类的网站中，需要记录用户的当前登录状态、最新发布的消息及好友状态等，用户每次刷新页面都需要更新这些信息。

Web 应用中将这些多次请求修改使用的上下文对象称作会话（Session），单机情况下，Session 可由部署在服务器上的 Web 容器（如 JBoss）管理。在使用负载均衡的集群环境中，由于负载均衡服务器可能会将请求分发到集群任何一台应用服务器上，所以保证每次请求依然能够获得正确的 Session 比单机时要复杂很多。

集群环境下，Session 管理主要有以下几种手段。

1. Session 复制

Session 复制是早期企业应用系统使用较多的一种服务器集群 Session 管理机制。应用服务器开启 Web 容器的 Session 复制功能，在集群中的几台服务器之间同步 Session 对象，使得每台服务器上都保存所有用户的 Session 信息，这样任何一台机器宕机都不会导致 Session 数据的丢失，而服务器使用 Session 时，也只需要在本机获取即可。如图 5.6 所示。

这种方案虽然简单，从本机读取 Session 信息也很快速，但只能使用在集群规模比较小的情况下。当集群规模较大时，集群服务器间需要大量的通信进行 Session 复制，占用服务器和网络的大量资源，系统不堪负担。而且由于所有用户的 Session 信息在每台服务器上都有备份，在大量用户访问的情况下，甚至会出现服务器内存不够 Session 使用的情况。

而大型网站的核心应用集群就是数千台服务器，同时在线用户可达千万，因此并不适用这种方案。

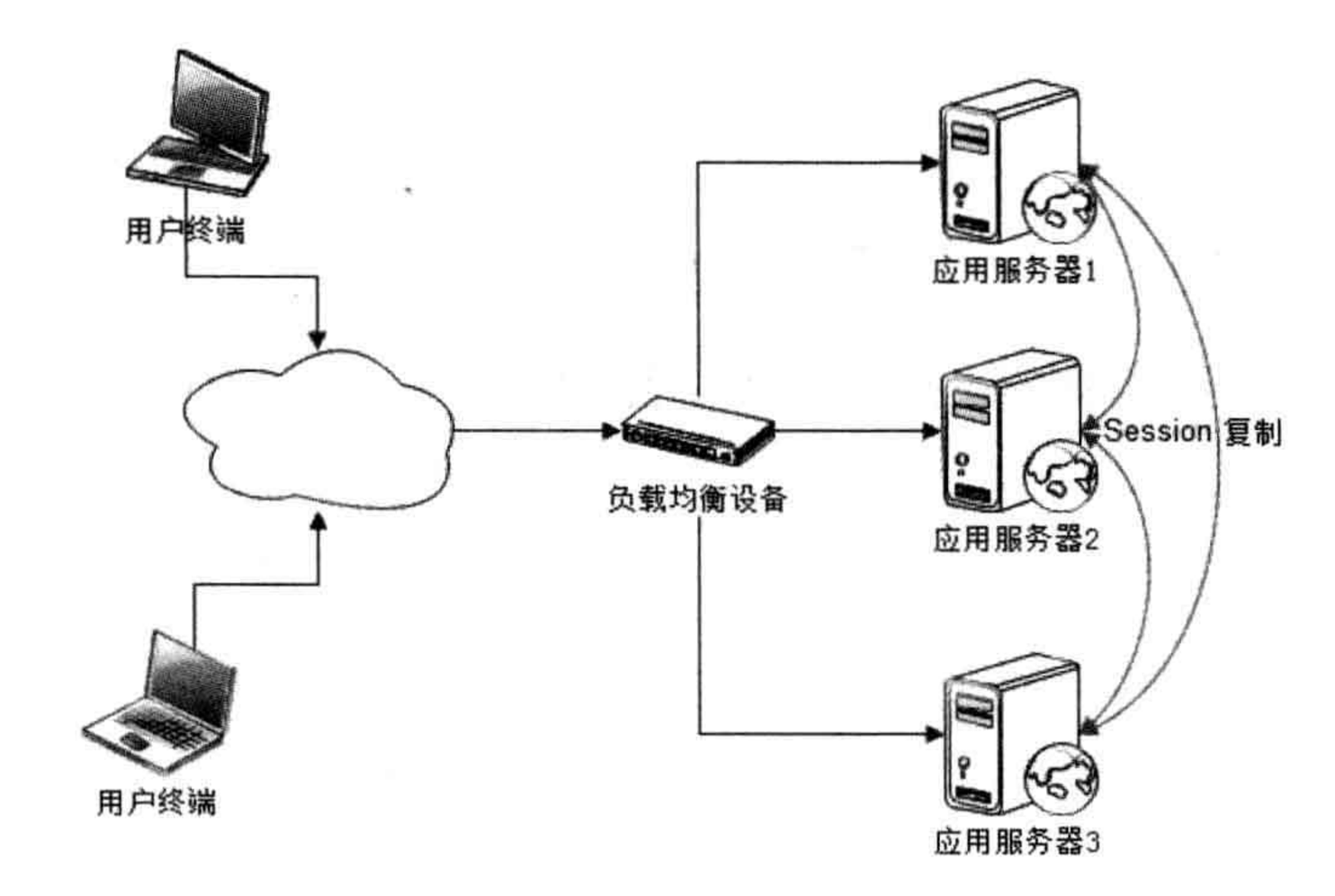

图 5.6 使用 Session 复制实现应用服务器共享 Session

2. Session 绑定

Session 绑定可以利用负载均衡的源地址 Hash 算法实现，负载均衡服务器总是将来源于同一 IP 的请求分发到同一台服务器上（也可以根据 Cookie 信息将同一个用户的请求总是分发到同一台服务器上，当然这时负载均衡服务器必须工作在 HTTP 协议层上，关于负载均衡算法的更多信息请参考本书第 6 章内容。这样在整个会话期间，用户所有的请求都在同一台服务器上处理，即 Session 绑定在某台特定服务器上，保证 Session 总能在这台服务器上获取。这种方法又被称作会话黏滞，如图 5.7 所示。

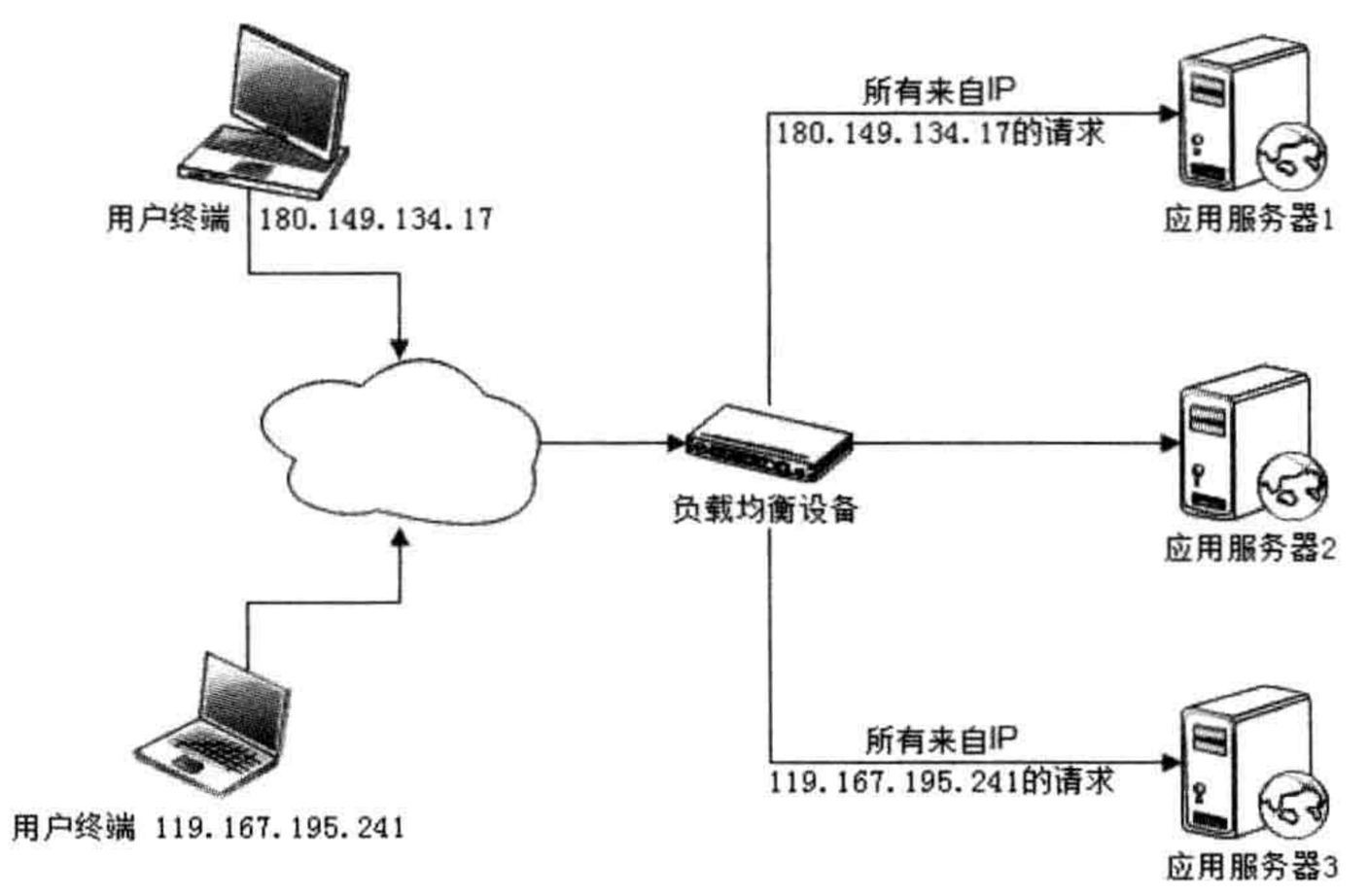

图 5.7 利用负载均衡的会话黏滞机制将请求绑定到特定服务器

但是 Session 绑定的方案显然不符合我们对系统高可用的需求，因为一旦某台服务器宕机，那么该机器上的 Session 也就不复存在了，用户请求切换到其他机器后因为没有 Session 而无法完成业务处理。因此虽然大部分负载均衡服务器都提供源地址负载均衡算法，但很少有网站利用这个算法进行 Session 管理。

3. 利用 Cookie 记录 Session

早期的企业应用系统使用 C/S（客户端/服务器）架构，一种管理 Session 的方式是将 Session 记录在客户端，每次请求服务器的时候，将 Session 放在请求中发送给服务器，服务器处理完请求后再将修改过的 Session 响应给客户端。

网站没有客户端，但是可以利用浏览器支持的 Cookie 记录 Session，如图 5.8 所示。

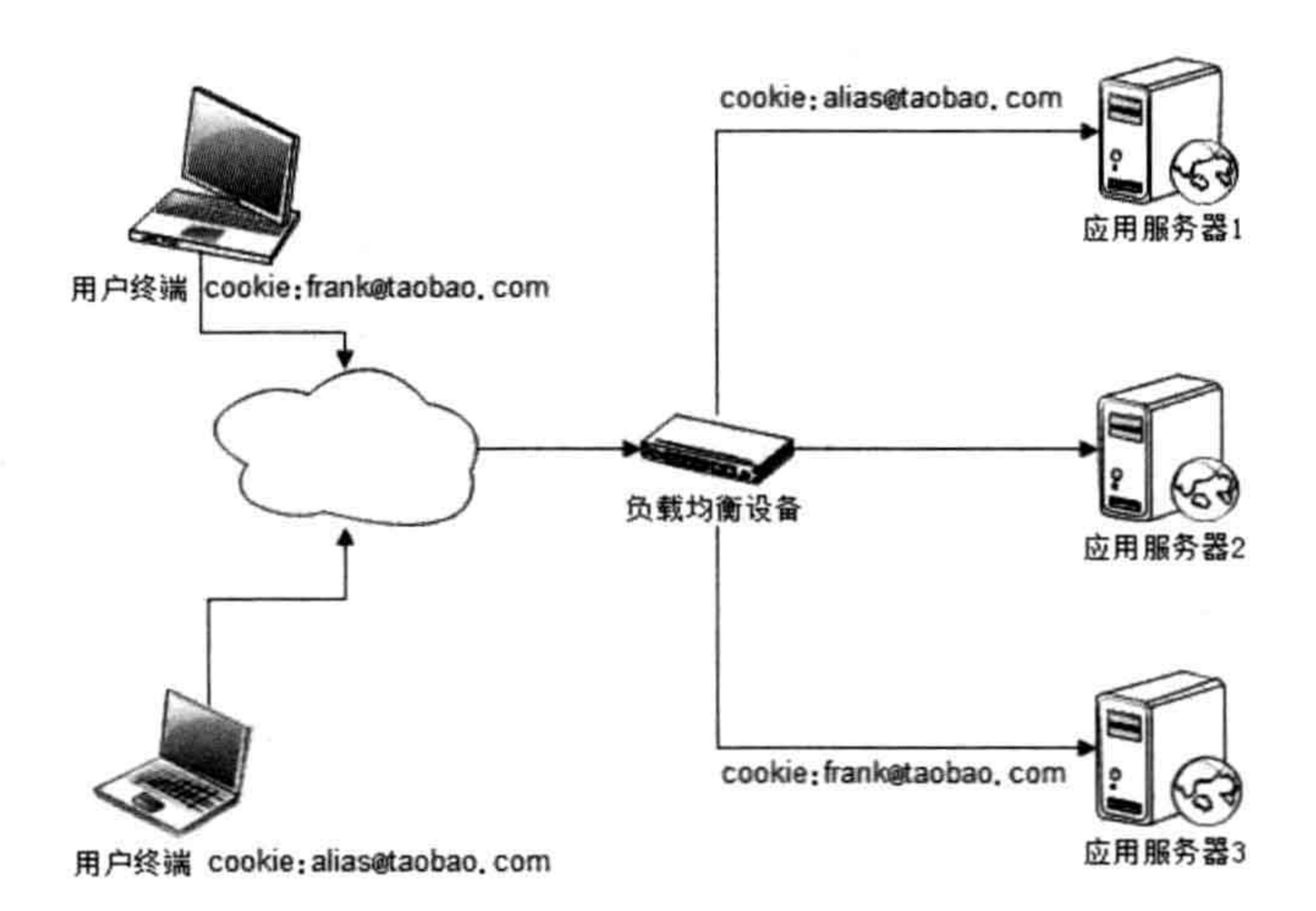

图 5.8 利用 Cookie 记录 Session 信息

利用 Cookie 记录 Session 也有一些缺点，比如受 Cookie 大小限制，能记录的信息有限；每次请求响应都需要传输 Cookie，影响性能；如果用户关闭 Cookie，访问就会不正常。但是由于 Cookie 的简单易用，可用性高，支持应用服务器的线性伸缩，而大部分应用需要记录的 Session 信息又比较小。因此事实上，许多网站都或多或少地使用 Cookie 记录 Session。

4. Session 服务器

那么有没有可用性高、伸缩性好、性能也不错，对信息大小又没有限制的服务器集

群 Session 管理方案呢？

答案就是 Session 服务器。利用独立部署的 Session 服务器（集群）统一管理 Session，应用服务器每次读写 Session 时，都访问 Session 服务器，如图 5.9 所示。

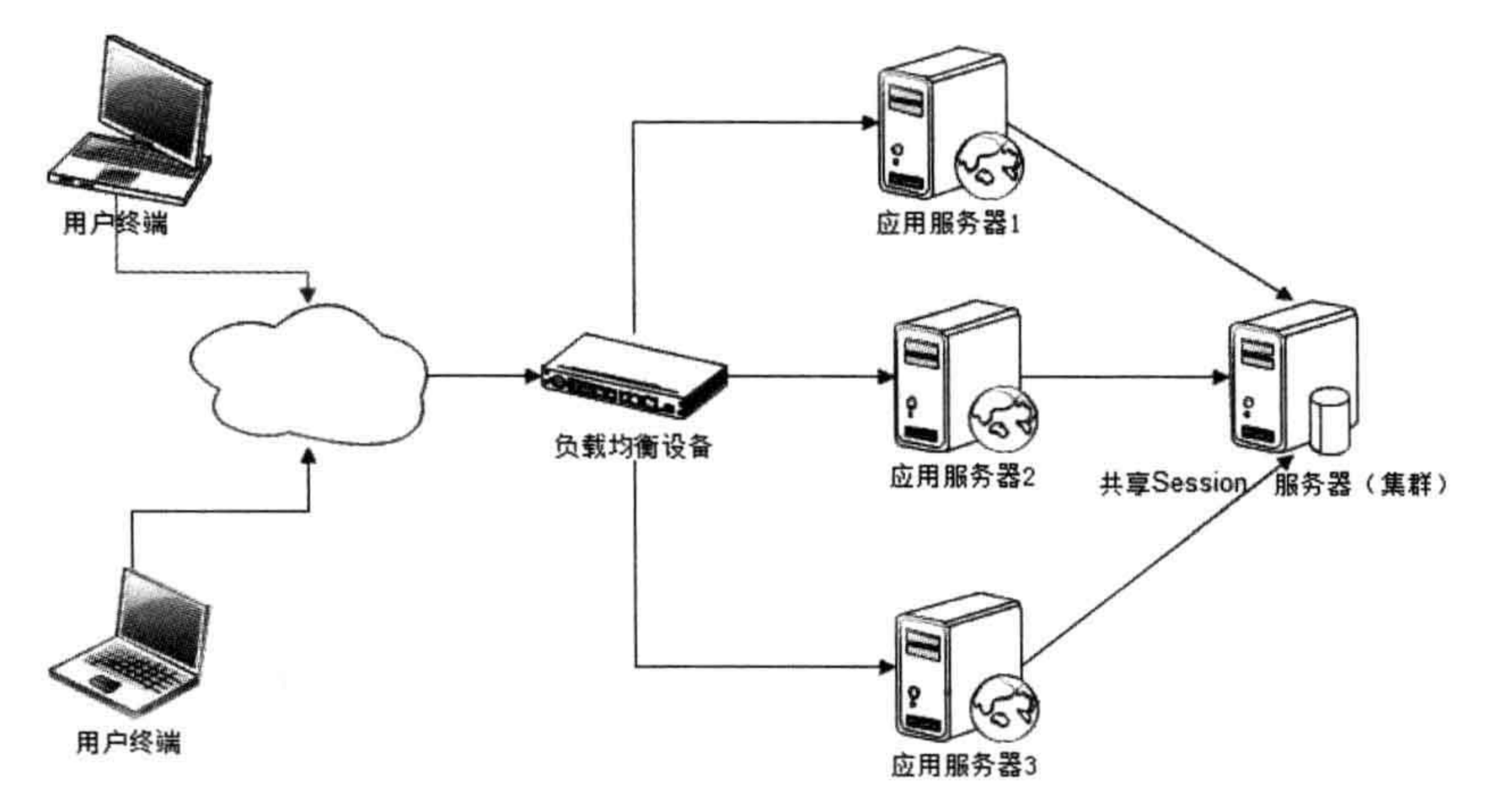

图 5.9　利用 Session 服务器共享 Session

这种解决方案事实上是将应用服务器的状态分离，分为无状态的应用服务器和有状态的 Session 服务器，然后针对这两种服务器的不同特性分别设计其架构。

对于有状态的 Session 服务器，一种比较简单的方法是利用分布式缓存、数据库等，在这些产品的基础上进行包装，使其符合 Session 的存储和访问要求。如果业务场景对 Session 管理有比较高的要求，比如利用 Session 服务集成单点登录（SSO）、用户服务等功能，则需要开发专门的 Session 服务管理平台。

5.4　高可用的服务

可复用的服务模块为业务产品提供基础公共服务，大型网站中这些服务通常都独立分布式部署，被具体应用远程调用。可复用的服务和应用一样，也是无状态的服务，因此可以使用类似负载均衡的失效转移策略实现高可用的服务。

除此之外，具体实践中，还有以下几点高可用的服务策略。

1. 分级管理

运维上将服务器进行分级管理，核心应用和服务优先使用更好的硬件，在运维响应速度上也格外迅速。显然，用户及时付款购物比能不能评价商品更重要，所以订单、支付服务比评价服务有更高优先级。

同时在服务部署上也进行必要的隔离，避免故障的连锁反应。低优先级的服务通过启动不同的线程或者部署在不同的虚拟机上进行隔离，而高优先级的服务则需要部署在不同的物理机上，核心服务和数据甚至需要部署在不同地域的数据中心。

2. 超时设置

由于服务端宕机、线程死锁等原因，可能导致应用程序对服务端的调用失去响应，进而导致用户请求长时间得不到响应，同时还占用应用程序的资源，不利于及时将访问请求转移到正常的服务器上。

在应用程序中设置服务调用的超时时间，一旦超时，通信框架就抛出异常，应用程序根据服务调度策略，可选择继续重试或将请求转移到提供相同服务的其他服务器上。

3. 异步调用

应用对服务的调用通过消息队列等异步方式完成，避免一个服务失败导致整个应用请求失败的情况。如提交一个新用户注册请求，应用需要调用三个服务：将用户信息写入数据库，发送账户注册成功邮件，开通对应权限。如果采用同步服务调用，当邮件队列阻塞不能发送邮件时，会导致其他两个服务也无法执行，最终导致用户注册失败。

如果采用异步调用的方式，应用程序将用户注册信息发送给消息队列服务器后立即返回用户注册成功响应。而记录用户注册信息到数据库、发送用户注册成功邮件、调用用户服务开通权限这三个服务作为消息的消费者任务，分别从消息队列获取用户注册信息异步执行。即使邮件服务队列阻塞，邮件不能成功发送，也不会影响其他服务的执行，用户注册操作可顺利完成，只是晚一点收到注册成功的邮件而已。

当然不是所有服务调用都可以异步调用，对于获取用户信息这类调用，采用异步方式会延长响应时间，得不偿失。对于那些必须确认服务调用成功才能继续下一步操作的应用也不合适使用异步调用。

4. 服务降级

在网站访问高峰期，服务可能因为大量的并发调用而性能下降，严重时可能会导致服务宕机。为了保证核心应用和功能的正常运行，需要对服务进行降级。降级有两种手段：拒绝服务及关闭服务。

拒绝服务：拒绝低优先级应用的调用，减少服务调用并发数，确保核心应用正常使用；或者随机拒绝部分请求调用，节约资源，让另一部分请求得以成功，避免要死大家一起死的惨剧。貌似 Twitter 比较喜欢使用随机拒绝请求的策略，经常有用户看到请求失败的故障页面，但是问下身边的人，其他人都正常使用，自己再刷新页面，也好了。

关闭功能：关闭部分不重要的服务，或者服务内部关闭部分不重要的功能，以节约系统开销，为重要的服务和功能让出资源。淘宝在每年的“双十一”促销中就使用这种方法，在系统最繁忙的时段关闭“评价”、“确认收货”等非核心服务，以保证核心交易服务的顺利完成。

5. 幂等性设计

应用调用服务失败后，会将调用请求重新发送到其他服务器，但是这个失败可能是虚假的失败。比如服务已经处理成功，但因为网络故障应用没有收到响应，这时应用重新提交请求就导致服务重复调用，如果这个服务是一个转账操作，就会产生严重后果。

服务重复调用是无法避免的，应用层也不需要关心服务是否真的失败，只要没有收到调用成功的响应，就可以认为调用失败，并重试服务调用。因此必须在服务层保证服务重复调用和调用一次产生的结果相同，即服务具有幂等性。

有些服务天然具有幂等性，比如将用户性别设置为男性，不管设置多少次，结果都一样。但是对于转账交易等操作，问题就会比较复杂，需要通过交易编号等信息进行服务调用有效性校验，只有有效的操作才能继续执行。

5.5 高可用的数据

对许多网站而言，数据是其最宝贵的物质资产，硬件可以购买，软件可以重写，但是多年运营积淀下来的各种数据（用户数据、交易数据、商品数据……），代表着历史，

已经成为过往，不能再重来，一旦失去，对网站的打击可以说是毁灭性的，因此可以说，保护网站的数据就是保护企业的命脉。

不同于高可用的应用和服务，由于数据存储服务器上保存的数据不同，当某台服务器宕机的时候，数据访问请求不能任意切换到集群中其他的机器上。

保证数据存储高可用的手段主要是数据备份和失效转移机制。数据备份是保证数据有多个副本，任意副本的失效都不会导致数据的永久丢失，从而实现数据完全的持久化。而失效转移机制则保证当一个数据副本不可访问时，可以快速切换访问数据的其他副本，保证系统可用。

关于缓存服务的高可用，在实践中争议很大，一种观点认为缓存已经成为网站数据服务的重要组成部分，事实上承担了业务中绝大多数的数据读取访问服务，缓存服务失效可能会导致数据库负载过高而宕机，进而影响整个网站的可用性，因此缓存服务需要实现和数据存储服务同样的高可用。

另一种观点认为，缓存服务不是数据存储服务，缓存服务器宕机引起缓存数据丢失导致服务器负载压力过高应该通过其他手段解决，而不是提高缓存服务本身的高可用。

笔者持后一种观点，对于缓存服务器集群中的单机宕机，如果缓存服务器集群规模较大，那么单机宕机引起的缓存数据丢失比例和数据库负载压力变化都较小，对整个系统影响也较小。扩大缓存服务器集群规模的一个简单手段就是整个网站共享同一个分布式缓存集群，单独的应用和产品不需要部署自己的缓存服务器，只需要向共享缓存集群申请缓存资源即可。并且通过逻辑或物理分区的方式将每个应用的缓存部署在多台服务器上，任何一台服务器宕机引起的缓存失效都只影响应用缓存数据的一小部分，不会对应用性能和数据库负载造成太大影响。

5.5.1 CAP 原理

在讨论高可用数据服务架构之前，必须先讨论的一个话题是，为了保证数据的高可用，网站通常会牺牲另一个也很重要的指标：数据一致性。

高可用的数据有如下几个层面的含义。

数据持久性

保证数据可持久存储，在各种情况下都不会出现数据丢失的问题。为了实现数据的持久性，不但在写入数据时需要写入持久性存储，还需要将数据备份一个或多个副本，存放在不同的物理存储设备上，在某个存储故障或灾害发生时，数据不会丢失。

数据可访问性

在多份数据副本分别存放在不同存储设备的情况下，如果一个数据存储设备损坏，就需要将数据访问切换到另一个数据存储设备上，如果这个过程不能很快完成（终端用户几乎没有感知），或者在完成过程中需要停止终端用户访问数据，那么这段时间数据是不可访问的。

数据一致性

在数据有多份副本的情况下，如果网络、服务器或者软件出现故障，会导致部分副本写入成功，部分副本写入失败。这就会造成各个副本之间的数据不一致，数据内容冲突。实践中，导致数据不一致的情形有很多种，表现形式也多种多样，比如数据更新返回操作失败，事实上数据在存储服务器已经更新成功。

CAP 原理认为，一个提供数据服务的存储系统无法同时满足数据一致性（Consistency）、数据可用性（Availibility）、分区耐受性（Patition Tolerance，系统具有跨网络分区的伸缩性）这三个条件，如图 5.10 所示。

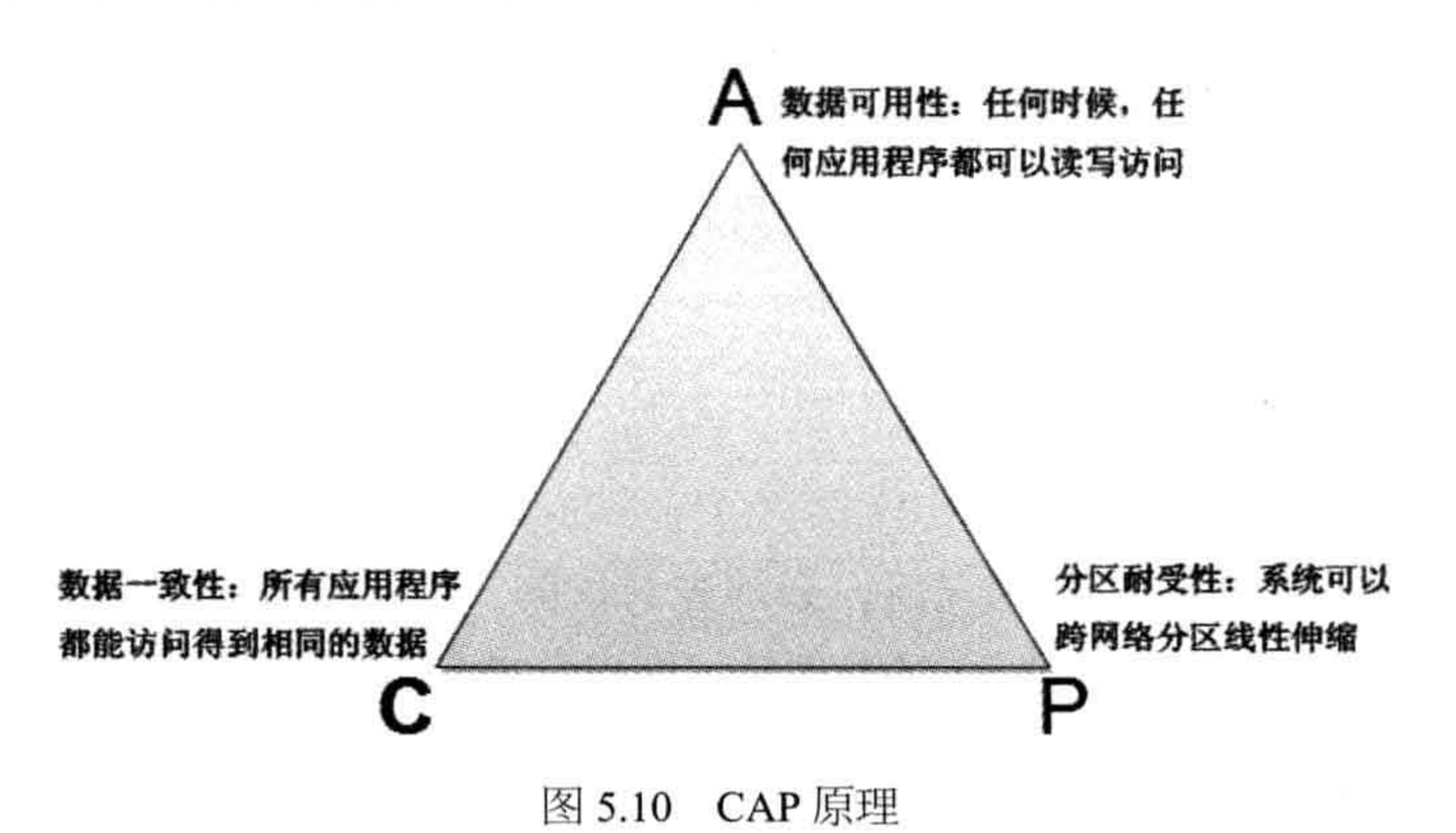

图 5.10　CAP 原理

在大型网站应用中，数据规模总是快速扩张的，因此可伸缩性即分区耐受性必不可

少，规模变大以后，机器数量也会变得庞大，这时网络和服务器故障会频繁出现，要想保证应用可用，就必须保证分布式处理系统的高可用性。所以在大型网站中，通常会选择强化分布式存储系统的可用性（A）和伸缩性（P），而在某种程度上放弃一致性（C）。一般说来，数据不一致通常出现在系统高并发写操作或者集群状态不稳（故障恢复、集群扩容……）的情况下，应用系统需要对分布式数据处理系统的数据不一致性有所了解并进行某种意义上的补偿和纠错，以避免出现应用系统数据不正确。

2012 年淘宝“双十一”活动期间，在活动第一分钟就涌入了 1000 万独立用户访问，这种极端的高并发场景对数据处理系统造成了巨大压力，存储系统较弱的数据一致性导致出现部分商品超卖现象（交易成功的商品数超过了商品库存数）。

CAP 原理对于可伸缩的分布式系统设计具有重要意义，在系统设计开发过程中，不恰当地迎合各种需求，企图打造一个完美的产品，可能会使设计进入两难境地，难以为继。

具体说来，数据一致性又可分为如下几点。

数据强一致

各个副本的数据在物理存储中总是一致的；数据更新操作结果和操作响应总是一致的，即操作响应通知更新失败，那么数据一定没有被更新，而不是处于不确定状态。

数据用户一致

即数据在物理存储中的各个副本的数据可能是不一致的，但是终端用户访问时，通过纠错和校验机制，可以确定一个一致的且正确的数据返回给用户。

数据最终一致

这是数据一致性中较弱的一种，即物理存储的数据可能是不一致的，终端用户访问到的数据可能也是不一致的（同一用户连续访问，结果不同；或者不同用户同时访问，结果不同），但系统经过一段时间（通常是一个比较短的时间段）的自我恢复和修正，数据最终会达到一致。

因为难以满足数据强一致性，网站通常会综合成本、技术、业务场景等条件，结合应用服务和其他的数据监控与纠错功能，使存储系统达到用户一致，保证最终用户访问数据的正确性。

5.5.2 数据备份

数据备份是一种古老而有效的数据保护手段，早期的数据备份手段主要是数据冷备，即定期将数据复制到某种存储介质（磁带，光盘……）上并物理存档保管，如果系统存储损坏，那么就从冷备的存储设备中恢复数据。

冷备的优点是简单和廉价，成本和技术难度都较低。缺点是不能保证**数据最终一致**，由于数据是定期复制，因此备份设备中的数据比系统中的数据陈旧，如果系统数据丢失，那么从上个备份点开始后更新的数据就会永久丢失，不能从备份中恢复。同时也不能保证**数据可用性**，从冷备存储中恢复数据需要较长的时间，而这段时间无法访问数据，系统也不可用。

因此，数据冷备作为一种传统的数据保护手段，依然在网站日常运维中使用，同时在网站实时在线业务中，还需要进行数据热备，以提供更好的数据可用性。

数据热备可分为两种：异步热备方式和同步热备方式。

异步方式是指多份数据副本的写入操作异步完成，应用程序收到数据服务系统的写操作成功响应时，只写成功了一份，存储系统将会异步地写其他副本（这个过程有可能会失败）。如图 5.11 所示。

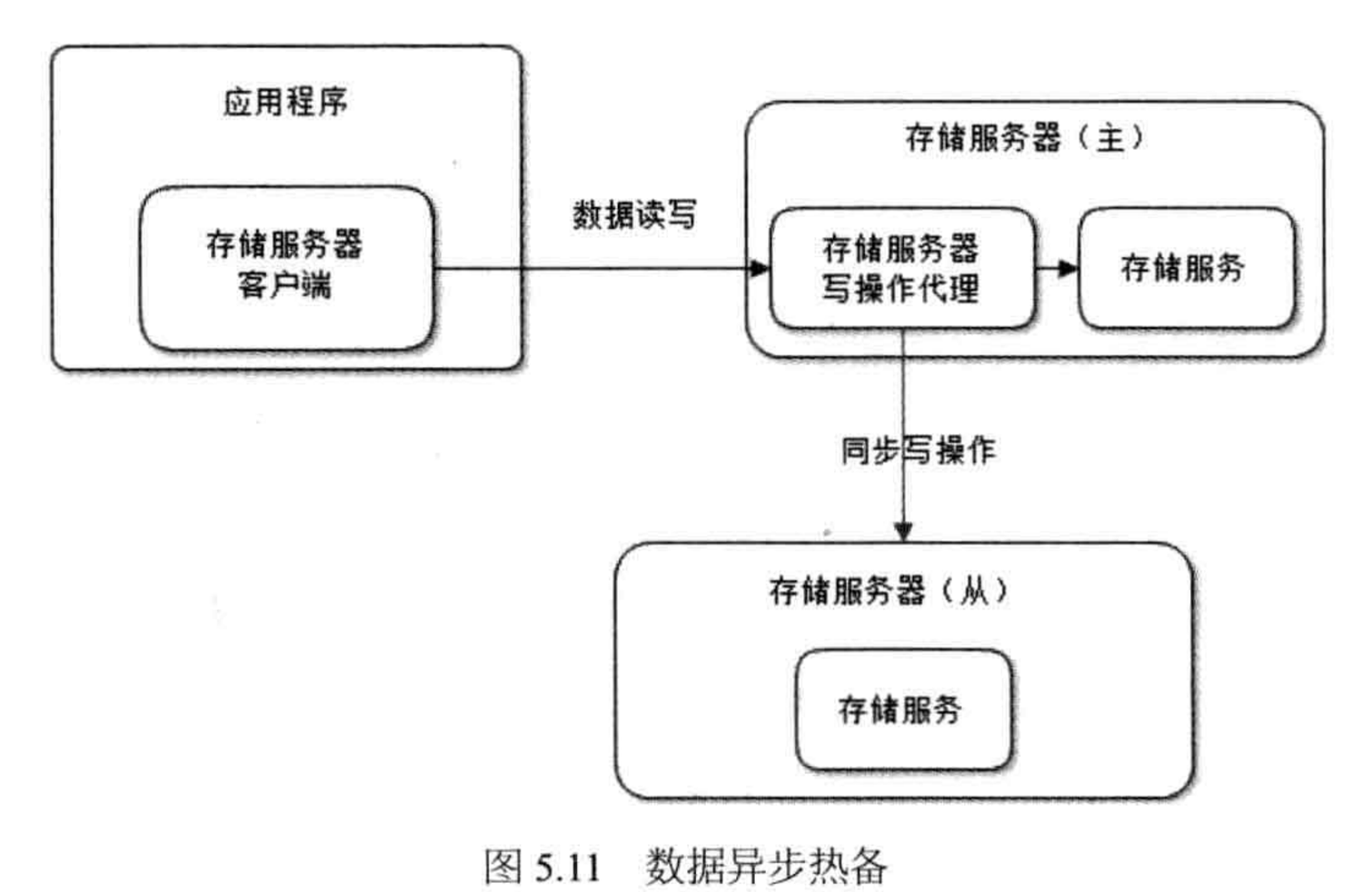

图 5.11 数据异步热备

在异步写入方式下，存储服务器分为主存储服务器（Master）和从存储服务器（Slave），应用程序正常情况下只连接主存储服务器，数据写入时，由主存储服务器的写操作代理

模块将数据写入本机存储系统后立即返回写操作成功响应，然后通过异步线程将写操作数据同步到从存储服务器。

同步方式是指多份数据副本的写入操作同步完成，即应用程序收到数据服务系统的写成功响应时，多份数据都已经写操作成功。但是当应用程序收到数据写操作失败的响应时，可能有部分副本或者全部副本都已经写成功了（因为网络或者系统故障，无法返回操作成功的响应），如图 5.12 所示。

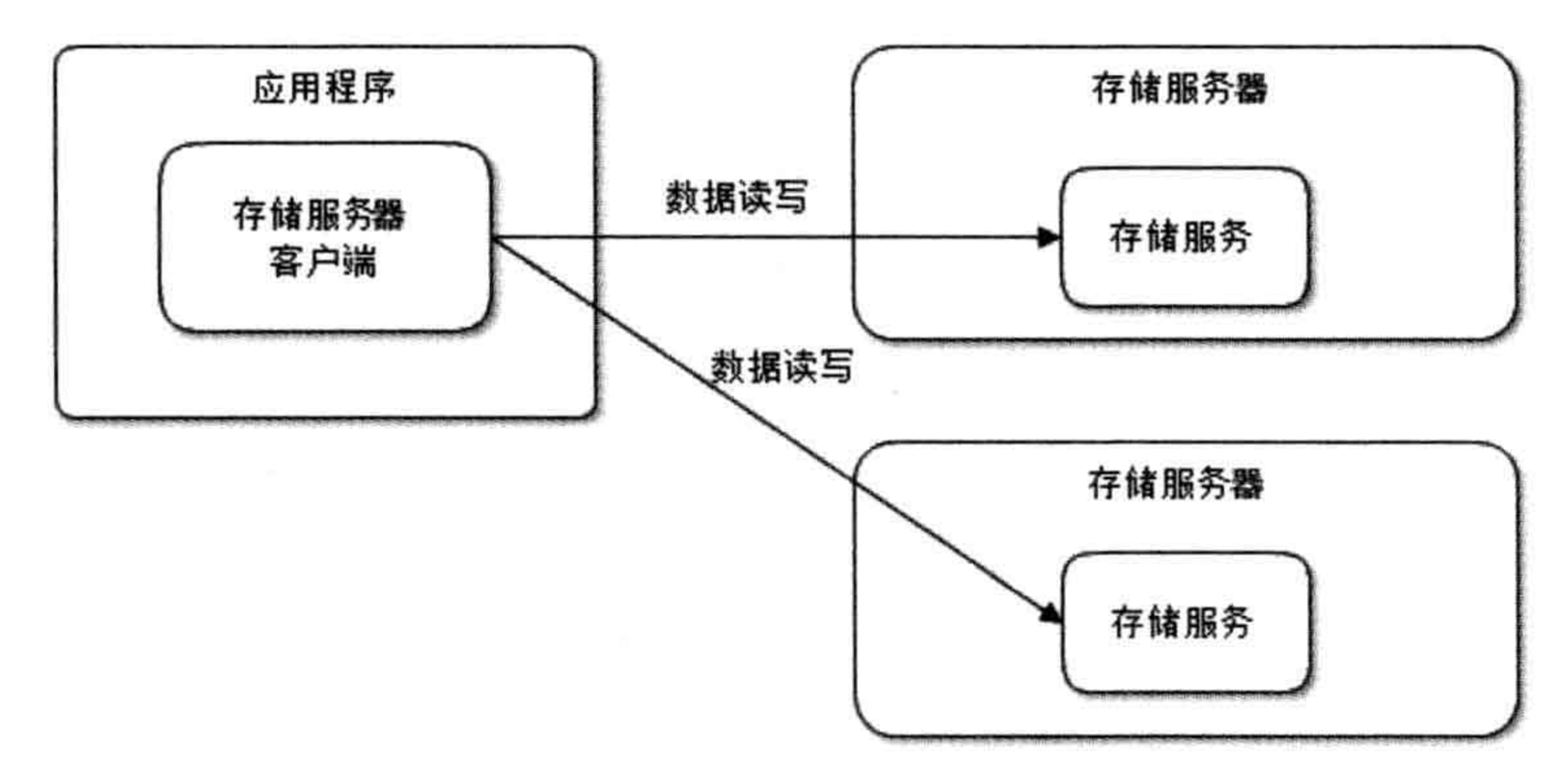

图 5.12 数据同步热备

同步热备具体实现的时候，为了提高性能，在应用程序客户端并发向多个存储服务器同时写入数据，然后等待所有存储服务器都返回操作成功的响应后，再通知应用程序写操作成功。

这种情况下，存储服务器没有主从之分，完全对等，更便于管理和维护。存储服务客户端在写多份数据的时候，并发操作，这意味着多份数据的总写操作延迟是响应最慢的那台存储服务器的响应延迟，而不是多台存储服务器响应延迟之和。其性能和异步热备方式差不多。

传统的企业级关系数据库系统几乎都提供了数据实时同步备份的机制。而一开始就为大型网站而设计的各种 NoSQL 数据库（如 HBase）更是将数据备份机制作为产品最主要的功能点之一。

关系数据库热备机制就是通常所说的 Master-Slave 同步机制。Master-Slave 机制不但解决了数据备份问题，还改善了数据库系统的性能，实践中，通常使用读写分离的方法

访问 Slave 和 Master 数据库，写操作只访问 Master 数据库，读操作只访问 Slave 数据库。

5.5.3 失效转移

若数据服务器集群中任何一台服务器宕机，那么应用程序针对这台服务器的所有读写操作都需要重新路由到其他服务器，保证数据访问不会失败，这个过程叫作失效转移。

失效转移操作由三部分组成：失效确认、访问转移、数据恢复。

1. 失效确认

判断服务器宕机是系统进行失效转移的第一步，系统确认一台服务器是否宕机的手段有两种：心跳检测和应用程序访问失败报告，如图 5.13 所示。

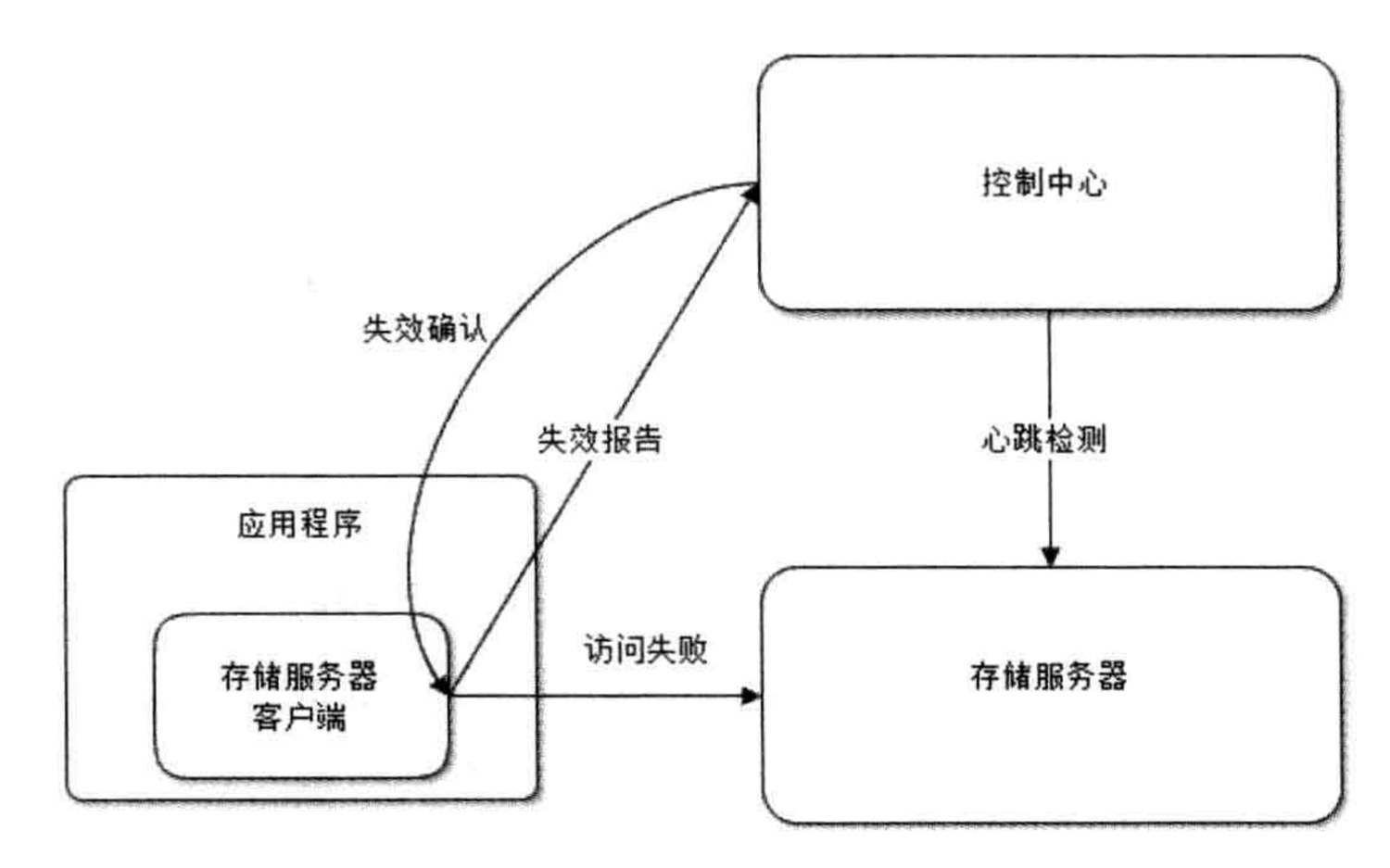

图 5.13 存储服务器失效确认

对于应用程序的访问失败报告，控制中心还需要再一次发送心跳检测进行确认，以免错误判断服务器宕机，因为一旦进行数据访问的失效转移，就意味着数据存储多份副本不一致，需要进行后续一系列复杂的操作。

2. 访问转移

确认某台数据存储服务器宕机后，就需要将数据读写访问重新路由到其他服务器上。对于完全对等存储的服务器（几台存储服务器存储的数据完全一样，我们称几台服务器为对等服务器，比如主从结构的存储服务器，其存储的数据完全一样），当其中一台宕机后，应用程序根据配置直接切换到对等服务器上。如果存储是不对等的， 那么就需要重

新计算路由，选择存储服务器。

3. 数据恢复

因为某台服务器宕机，所以数据存储的副本数目会减少，必须将副本的数目恢复到系统设定的值，否则，再有服务器宕机时，就可能出现无法访问转移（所有副本的服务器都宕机了），数据永久丢失的情况。因此系统需要从健康的服务器复制数据，将数据副本数目恢复到设定值。具体设计可参考本书第 11 章。

5.6 高可用网站的软件质量保证

在网站运维实践中，除了网络、服务器等硬件故障导致的系统可用性风险外，还有来自软件系统本身的风险。

关于传统的软件测试和软件质量保证管理无需赘言，本节重点讨论网站为了保证线上系统的可用性而采取的一些与传统软件开发不同的质量保证手段。

5.6.1 网站发布

网站需要保证 7×24 高可用运行，同时网站又需要不断地发布新功能吸引用户以保证在激烈的市场竞争中获得成功。许多大型网站每周都需要发布一到两次，而中小型网站则更加频繁，一些处于快速发展期的网站甚至每天发布十几次。

不管发布的新功能是修改了一个按钮的布局还是增加了一个核心业务，都需要在服务器上关闭原有的应用，然后重新部署启动新的应用，整个过程还要求不影响用户的使用。这相当于要求给**飞行中的飞机换个引擎**，既不能让飞机有剧烈晃动（影响用户体验），也不能让飞机降落（系统停机维护），更不能让飞机坠毁（系统故障网站完全不可用）。

网站的发布过程事实上和服务器宕机效果相当，其对系统可用性的影响也和服务器宕机相似。所以设计一个网站的高可用架构时，需要考虑的服务器宕机概率不是物理上的每年一两次，而是事实上的每周一两次。也许你认为这个应用不重要，重启也非常快，用户可以忍受每年一到两次的宕机故障，因而不需要复杂的高可用设计。事实上，由于应用的不断发布，用户需要面对的是每周一到两次的宕机故障。

但是网站发布毕竟是一次提前预知的服务器宕机，所以过程可以更柔和，对用户影响更小。通常使用发布脚本来完成发布，其流程如图 5.14 所示。

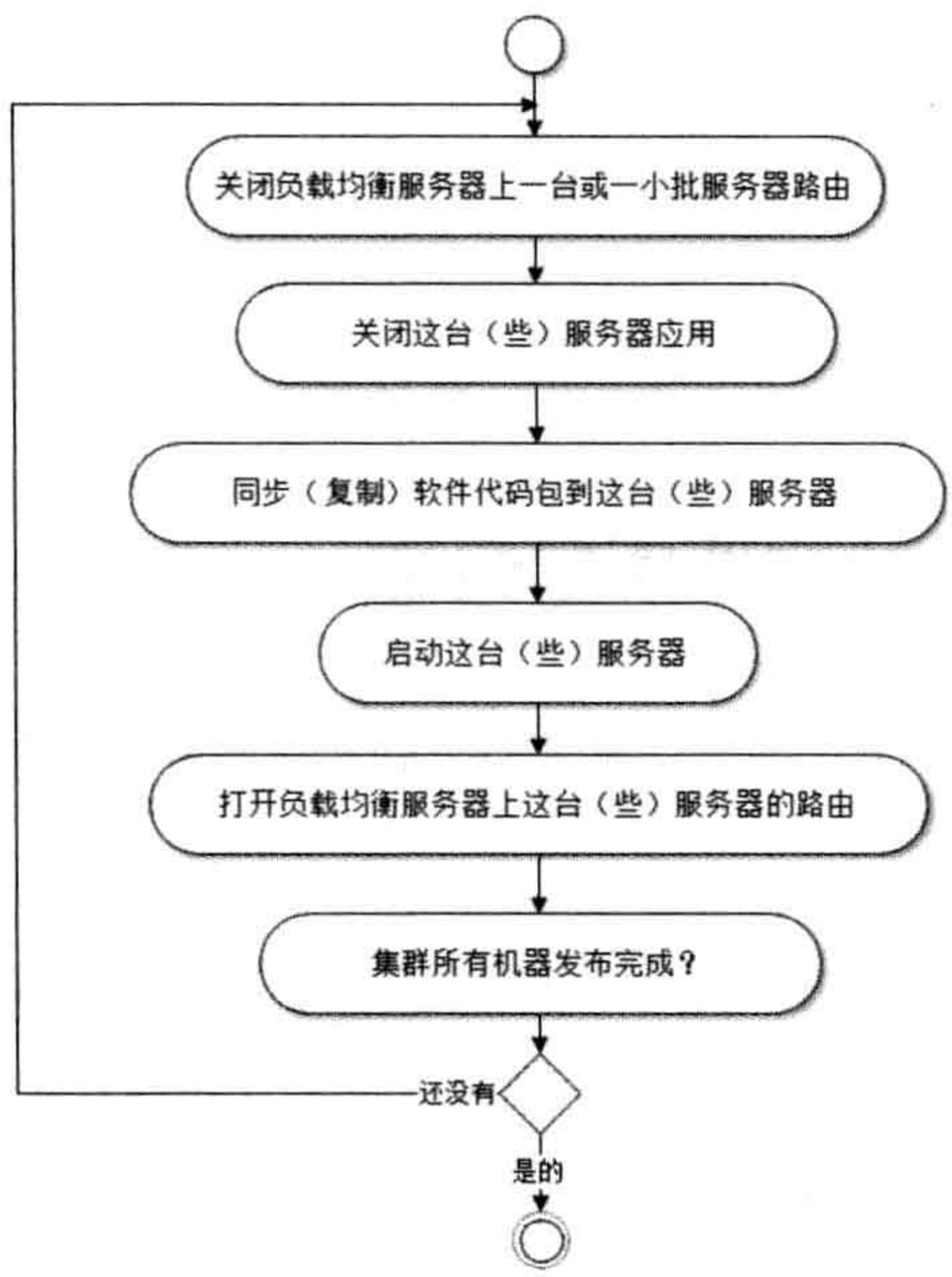

图 5.14 网站应用发布流程

发布过程中，每次关闭的服务器都是集群中的一小部分，并在发布完成后立即可以访问，因此整个发布过程不影响用户使用。

5.6.2 自动化测试

代码在发布到线上服务器之前需要进行严格的测试。即使每次发布的新功能都是在原有系统功能上的小幅增加，但为了保证系统没有引入未预料的 Bug，网站测试还是需要对整个网站功能进行全面的回归测试。此外还需要测试各种浏览器的兼容性。在发布频繁的网站应用中，如果使用人工测试，成本、时间及测试覆盖率都难以接受。

目前大部分网站都采用 Web 自动化测试技术，使用自动测试工具或脚本完成测试。比较流行的 Web 自动化测试工具是 ThoughtWorks 开发的 Selenium。Selenium 运行在浏览器中，模拟用户操作进行测试，因此 Selenium 可以同时完成 Web 功能测试和浏览器兼容测试。

大型网站通常也会开发自己的自动化测试工具，可以一键完成系统部署，测试数据生成、测试执行、测试报告生成等全部测试过程。许多网站测试工程师的编码能力毫不逊于软件工程师。

5.6.3 预发布验证

即使是经过严格的测试，软件部署到线上服务器之后还是经常会出现各种问题，甚至根本无法启动服务器。主要原因是测试环境和线上环境并不相同，特别是应用需要依赖的其他服务，如数据库，缓存、公用业务服务等，以及一些第三方服务，如电信短信网关、银行网银接口等。

也许是数据库表结构不一致；也许是接口变化导致的通信失败；也许是配置错误导致连接失败；也许是依赖的服务线上环境还没有准备好，这些问题都有可能导致应用故障。

因此在网站发布时，并不是把测试通过的代码包直接发布到线上服务器，而是先发布到预发布机器上，开发工程师和测试工程师在预发布服务器上进行预发布验证，执行一些典型的业务流程，确认系统没有问题后才正式发布。

预发布服务器是一种特殊用途的服务器，它和线上的正式服务器唯一的不同就是没有配置在负载均衡服务器上，外部用户无法访问，如图 5.15 所示。

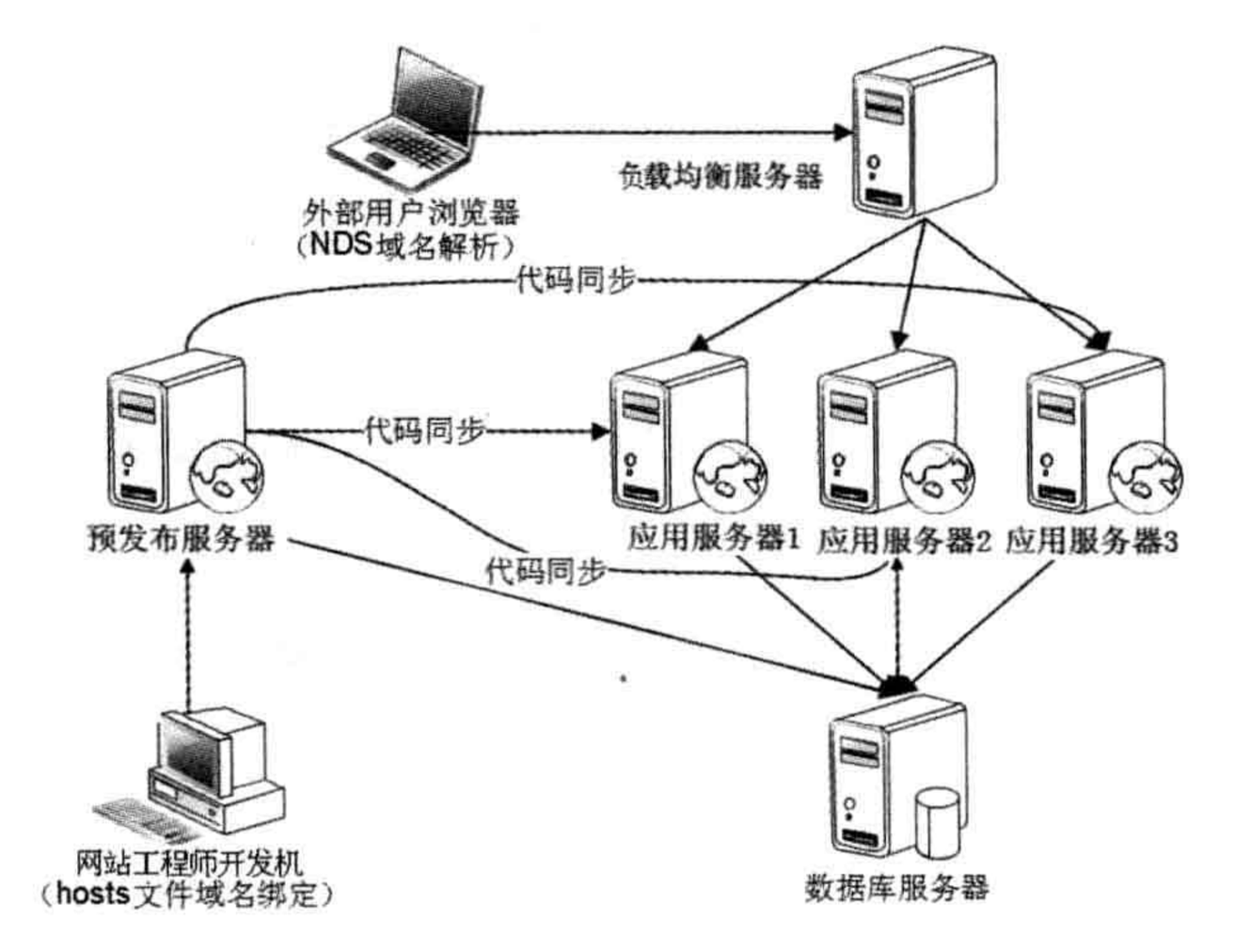

图 5.15 网站应用预发布

预发布服务器和线上正式服务器（应用服务器 1，2，3）都部署在相同的物理环境（同

一个数据中心甚至同一个机架上，如果使用虚拟机，甚至可能在同一个物理服务器上）中，使用相同的线上配置，依赖相同的外部服务。网站工程师通过在自己的开发用计算机上配置 hosts 文件绑定域名 IP 关系直接使用 IP 地址访问预发布服务器。如果在预发布服务器上执行的测试验证是正确的，基本可以确保在线上正式服务器部署时也没有问题。

不过，也有可能会因为预发布验证而引入问题。因为预发布服务器连接的是真实的生产环境，所有的预发布验证操作都是真实有效的数据，这些操作也许会引起不可预期的问题。比如创建一个店铺，上架一个商品，就有可能有真的用户过来购买，如果不能发货，会导致用户投诉。

一个真实的案例是某网站需要验证海外第三方支付功能，每件商品的售价本来是数千美金，工程师不可能花数千美金去验证自己开发的功能，于是将金额改成一美元，验证成功后，幸福地发布上线了，第二天上班后，发现大量商品以一美元的价格成交。

此外，在网站应用中强调的一个处理错误的理念是快速失败（fast failed），即如果系统在启动时发现问题就立刻抛出异常，停止启动让工程师介入排查错误，而不是启动后执行错误的操作。

5.6.4 代码控制

对于大型网站，核心应用系统和公用业务模块涉及许多团队和工程师，需要对相同的代码库进行共同开发和维护。而这些团队对同一个应用的开发维护（开发周期和发布时间点各不相同），如果代码控制环节出了问题，可能将有问题的代码发布上线，将问题带入生产环境，导致系统故障。

网站代码控制的核心问题是如何进行代码管理，既能保证代码发布版本的稳定正确，同时又能保证不同团队的开发互不影响。

目前大部分网站使用的源代码版本控制工具是 SVN，SVN 代码控制和版本发布方式一般有以下两种。

1. 主干开发、分支发布

代码修改都在主干（trunk）上进行，需要发布的时候，从主干上拉一个分支（branch）发布，该分支即成为一个发布版本，如果该版本发现 Bug，继续在该分支上修改发布，并

将修改合并（merge）回主干，直到下次主干发布。

2. 分支开发，主干发布

任何修改都不得在主干上直接进行，需要开发一个新功能或者修复一个 Bug 时，从主干拉一个分支进行开发，开发完成且测试通过后，合并回主干，然后从主干进行发布，主干上的代码永远是最新发布的版本。

这两种方式各有优缺点。主干开发、分支发布方式，主干代码反应目前整个应用的状态，一目了然，便于管理和控制，也利于持续集成。分支开发，主干发布方式，各个分支独立进行，互不干扰，可以使不同发布周期的开发在同一应用中进行。

目前网站应用开发中主要使用的是分支开发、主干发布的方式，如图 5.16 所示。

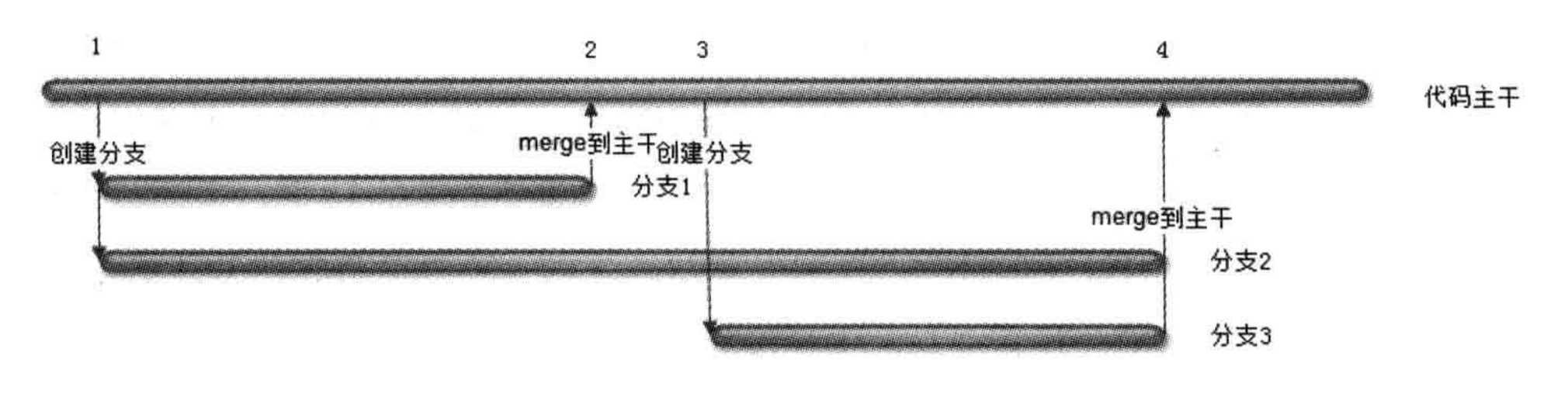

图 5.16 网站分支开发主干发布示意图

可以想象，如果使用主干开发、分支发布，那么在同一个应用上，对于不同开发周期，不同发布时间的项目，有可能 A 项目发布的时候，B 项目只开发了一半，这时候的主干代码是半成品，根本不能发布。而使用分支开发、主干发布的方式，只需要将 A 项目的分支合并回主干即可发布，不受 B 项目发布时间的影响。

目前在开源技术社区，Git 作为版本控制工具，正逐步取代 SVN 的地位。Git 对分布式开发，分支开发等有更好的支持，也更容易在各个开发分支上及时反映主干的最新更新，避免 SVN 在最后提交分支代码时发现和主干代码差别太大难以 merge 成功。但是 Git 的学习成本较高，如何和网站开发流程相结合还缺乏最佳实践和使用规范。不过相信 Git 成为网站的标准版本控制工具是迟早的事。

5.6.5 自动化发布

网站的版本发布频繁，整个发布过程需要许多团队通力合作，发布前，多个代码分支合并回主干可能会发生冲突（conflict），预发布验证也会带来风险，每次发布又相当于一次宕机事故。因此网站发布过程荆棘丛生，一不小心就会踩到雷。

对于有固定发布日期的网站（**很多网站选择周四作为发布日，这样一周前面有三天时间可以准备发布，后面还有一天时间可以挽回错误。如果选择周五发布，发现问题就必须要周末加班了。**），一到发布日，整个技术部门甚至运营部门就如临大敌，电话声此起彼伏，工程师步履匆匆，连空气中的温度都仿佛升高了几度。即便如此，发布过程还是常常出错，发布日工程师加班到凌晨是常有的事。而且容易忙中出错，因发布引发的故障也居高不下。

据说国外某知名互联网公司的 CTO 就因为没有有效手段控制发布故障、减少发布日的加班而引咎辞职。其继任者提出了一个火车发布模型：将每个应用的发布过程看作一次火车旅程，火车定点运行，期间有若干站点，每一站都进行例行检查，不通过的项目下车，剩下的项目继续坐着火车旅行，直到火车到达终点（应用发布成功）。但实际中，有可能所有项目都下车了，开着空车前进是没有意义的，火车不得不回到起点，等待解决了问题再重来一次。还有可能是车上有达官贵人（重点项目，CEO 跟投资人拍胸脯的项目），他不上车，谁也别想走，他出了错，大家都跟着回去重来。简化的火车发布模型如图 5.17 所示。

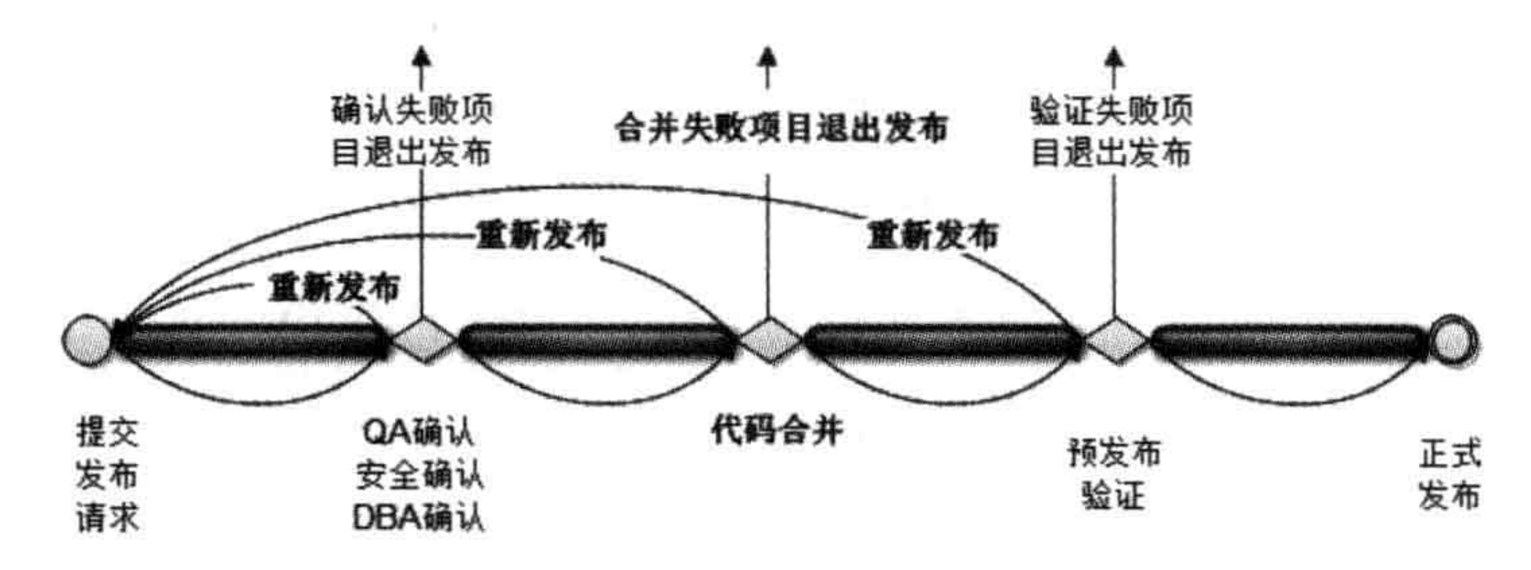

图 5.17 网站火车发布模型

由于火车发布模型是基于规则驱动的流程，所以这个流程可以自动化。采用火车发布模型的网站会开发一个自动化发布的工具实现发布过程的自动化。根据响应驱动流程，自动构造代码分支，进行代码合并，执行发布脚本等。正常流程下，可以做到发布过程

无人值守，无需 SCM（网站配置管理员）参与，每个项目相关人员基于流程执行相应的操作，即可完成应用自动发布。人的干预越少，自动化程度越高，引入故障的可能性就越小，火车准点到达，大家按时下班的可能性就越大。

5.6.6 灰度发布

应用发布成功后，仍然可能发现因为软件问题而引入的故障，这时候就需要做发布回滚，即卸载刚刚发布的软件，将上一个版本的软件包重新发布，使系统复原，消除故障。

大型网站的主要业务服务器集群规模非常庞大，比如某大型应用集群服务器数量超过一万台。一旦发现故障，即使想要发布回滚也需要很长时间才能完成，只能眼睁睁看着故障时间不断增加却干着急。为了应付这种局面，大型网站会使用灰度发布模式，将集群服务器分成若干部分，每天只发布一部分服务器，观察运行稳定没有故障，第二天继续发布一部分服务器，持续几天才把整个集群全部发布完毕，期间如果发现问题，只需要回滚已发布的一部分服务器即可。如图 5.18 所示。

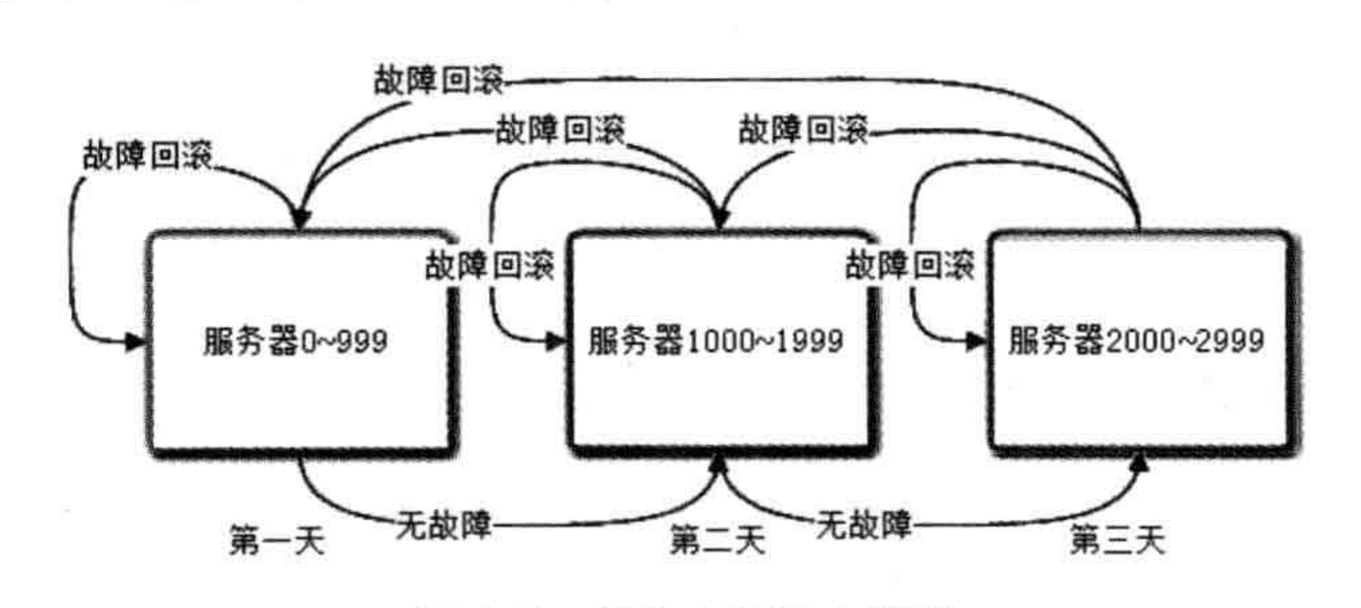

图 5.18 网站灰度发布模型

灰度发布也常用于用户测试，即在部分服务器上发布新版本，其余服务器保持老版本（或者发布另一个版本），然后监控用户操作行为，收集用户体验报告，比较用户对两个版本的满意度，以确定最终的发布版本。这种手段也被称作 AB 测试。

5.7 网站运行监控

“不允许没有监控的系统上线”，这是许多网站架构师在做项目上线评审时常说的一句话。网站运行监控对于网站运维和架构设计优化至关重要，运维没有监控的网站，犹

如驾驶没有仪表的飞机。盲人骑瞎马，夜半临深渊而不知，生死尚且未卜，提高可用性、减少故障率就更无从做起了。

5.7.1 监控数据采集

广义上的网站监控涵盖所有非直接业务行为的数据采集与管理，包括供数据分析师和产品设计师使用的网站用户行为日志、业务运行数据，以及供运维工程师和开发工程师使用的系统性能数据等。

1. 用户行为日志收集

用户行为日志指用户在浏览器上所做的所有操作及其所在的操作环境，包括用户操作系统与浏览器版本信息，IP 地址、页面访问路径、页面停留时间等，这些数据对统计网站 PV/UV 指标、分析用户行为、优化网站设计、个性化营销与推荐等非常重要。

具体用户行为日志收集手段有两种。

服务器端日志收集。这个方案比较简单，Apache 等几乎所有 Web 服务器都具备日志记录功能，可以记录大部分用户行为日志，开启 Web 服务器的日志记录功能即可。其缺点是可能会出现信息失真，如 IP 地址是代理服务器地址而不是用户真实 IP；无法识别访问路径等。

客户端浏览器日志收集。利用页面嵌入专门的 JavaScript 脚本可以收集用户真实的操作行为，因此比服务器日志收集更加精准。其缺点是比较麻烦，需要在页面嵌入特定的 JavaScript 脚本来完成。

此外，大型网站的用户日志数据量惊人，数据存储与计算压力很大，目前许多网站逐步开发基于实时计算框架 Storm 的日志统计与分析工具。

2. 服务器性能监控

收集服务器性能指标，如系统 Load、内存占用、磁盘 IO、网络 IO 等对尽早做出故障预警，及时判断应用状况，防患于未然，将故障扼杀在萌芽时期非常重要。此外根据性能监控数据，运维工程师可以合理安排服务器集群规模，架构师及时改善系统性能及调整系统伸缩性策略。

目前网站使用比较广泛的开源性能监控工具是 Ganglia，它支持大规模服务器集群，并支持以图形的方式在浏览器展示实时性能曲线。

3. 运行数据报告

除了服务器系统性能监控，网站还需要监控一些与具体业务场景相关的技术和业务指标，比如缓冲命中率、平均响应延迟时间、每分钟发送邮件数目、待处理的任务总数等。

对于服务器性能监控，网站运维人员可以在初始化系统时统一部署，应用程序开发完全不关心服务器性能监控。而运行数据需要在具体程序中采集并报告，汇总后统一显示，应用程序需要在代码中处理运行数据采集的逻辑。

5.7.2 监控管理

监控数据采集后，除了用作系统性能评估、集群规模伸缩性预测等，还可以根据实时监控数据进行风险预警，并对服务器进行失效转移，自动负载调整，最大化利用集群所有机器的资源。

系统报警

在服务器运行正常的情况下，其各项监控指标基本稳定在一个特定水平，如果这些指标超过某个阈值，就意味着系统可能将要出现故障，这时就需要对相关人员报警，及时采取措施，在故障还未真正发生时就将其扼杀在萌芽状态。

监控管理系统可以配置报警阈值和值守人员的联系方式，报警方式除了邮件，即时通信工具，还可以配置手机短信，语音报警，系统发生报警时，工程师即使在千里之外、夜里睡觉也能被及时通知，迅速响应。

失效转移

除了应用程序访问失败时进行失效转移，监控系统还可以在发现故障的情况下主动通知应用，进行失效转移。

自动优雅降级

优雅降级是指网站为了应付突然爆发的访问高峰，主动关闭部分功能，释放部分系

统资源，保证网站核心功能正常访问的一个手段。淘宝每年一次的“双十一”促销活动主动关闭“评价”、“确认收货”等非核心功能，以保证交易功能的正常进行，就可以看作是一种优雅降级。

网站在监控管理基础之上实现自动优雅降级，是网站柔性架构的理想状态：监控系统实时监控所有服务器的运行状况，根据监控参数判断应用访问负载情况，如果发现部分应用负载过高，而部分应用负载过低，就会适当卸载低负载应用部分服务器，重新安装启动部分高负载应用，使应用负载总体均衡，如果所有应用负载都很高，而且负载压力还在继续增加，就会自动关闭部分非重要功能，保证核心功能正常运行。

5.8 小结

对公司而言，可用性关系网站的生死存亡。对个人而言，可用性关系到自己的绩效升迁。工程师对架构做了许多优化、对代码做了很多重构，对性能、扩展性、伸缩性做了很多改善，但别人未必能直观地感受到，也许你的直接领导都不知道你做的这些意义何在。但如果你负责的产品出了重大故障，CEO 都会知道你的名字。事物总是先求生存，然后求发展。保证网站可用，万无一失，任重而道远。

6

永无止境：网站的伸缩性架构

所谓网站的伸缩性是指不需要改变网站的软硬件设计，仅仅通过改变部署的服务器数量就可以扩大或者缩小网站的服务处理能力。

京东网（www.360buy.com）在 2011 年年末的图书促销活动中，由于优惠幅度大引得大量买家访问，结果导致网站服务不可用，大部分用户在提交订单后，页面显示“Service is too busy”。当天晚上，京东网老板刘强东在微博发布消息称，已购买多台服务器以增加交易处理能力，第二天继续促销一天。结果第二天，用户在提交订单后，页面继续是“Service is too busy”。显然京东网当时的系统伸缩能力较弱，特别是订单处理子系统几乎没有什么伸缩能力。

与这些缺乏伸缩能力、关键时候掉链子的案例相对应的是淘宝网 2012 年“双十一”的促销活动，在活动开始的第一分钟，即有 1000 万独立用户访问网站，当天成功交易的订单总额达 191 亿，虽然淘宝网及支付宝网站出现了一些问题，但系统总体可用，绝大部分交易顺利完成。

大型网站的“大型”，在用户层面可以理解为大量用户及大量访问，如 Facebook 有超

过 10 亿用户；在功能方面可以理解为功能庞杂、产品众多，如腾讯有超过 1600 种产品；在技术层面可以理解为网站需要部署大量的服务器，如 Google 大约有近 100 万台服务器。

本书开篇曾经讨论过，大型网站不是一开始就是大型网站的，而是从小型网站逐步演化而来的，Google 诞生的时候也才只有一台服务器。设计一个大型网站或者一个大型软件系统，和将一个小网站逐渐演化成一个大型网站，其技术方案是完全不同的。前者如传统的银行系统，在设计之初就决定了系统的规模，如要服务的用户数、要处理的交易数等，然后采购大型计算机等昂贵的设备，将软件系统部署在上面，即成为一个大型系统，有朝一日这个大型系统也不能满足需求了，就花更多的钱打造一个更大型的系统。而网站一开始不可能规划出自己的规模，也不可能有那么多钱去开发一个大型系统，更不可能到了某个阶段再重新打造一个系统，只能摸着石头过河，从一台廉价的 PC 服务器开始自己的大型系统演化之路。

在这个渐进式的演化过程中，最重要的技术手段就是使用服务器集群，通过不断地向集群中添加服务器来增强整个集群的处理能力。这就是网站系统的伸缩性架构，只要技术上能做到向集群中加入服务器的数量和集群的处理能力成线性关系，那么网站就可以以此手段不断提升自己的规模，从一个服务几十人的小网站发展成服务几十亿人的大网站，从只能存储几个 G 图片的小网站发展成存储几百 P 图片的大网站。

这个演化过程总体来说是渐进式的，而且总是在“伸”，也就是说，网站的规模和服务器的规模总是在不断扩大（通常，一个需要“缩”的网站可能已经无法经营下去了）。但是这个过程也可能因为运营上的需要而出现脉冲，比如前面案例中提到的电商网站的促销活动：在某个短时间内，网站的访问量和交易规模突然爆发式增长，然后又回归正常状态。这时就需要网站的技术架构具有极好的伸缩性——活动期间向服务器集群中加入更多服务器（及向网络服务商租借更多的网络带宽）以满足用户访问，活动结束后又将这些服务器下线以节约成本。

国内有许多传统企业“触网”，将传统业务搬上互联网，这是一件值得称道的事，传统行业与互联网结合将会创造出新的经济模式，改善人们的生活。但遗憾的是，有些传统企业将自己的管理模式和经营理念也照搬到互联网领域——在技术方面的表现就是一开始就企图打造一个大型网站。

6.1 网站架构的伸缩性设计

回顾网站架构发展历程， 网站架构发展史就是一部不断向网站添加服务器的历史。只要工程师能向网站的服务器集群中添加新的机器，只要新添加的服务器能线性提高网站的整体服务处理能力，网站就无需为不断增长的用户和访问而焦虑。

一般说来，网站的伸缩性设计可分成两类，一类是根据功能进行物理分离实现伸缩，一类是单一功能通过集群实现伸缩。前者是不同的服务器部署不同的服务，提供不同的功能；后者是集群内的多台服务器部署相同的服务，提供相同的功能。

6.1.1 不同功能进行物理分离实现伸缩

网站发展早期——通过增加服务器提高网站处理能力时，新增服务器总是从现有服务器中分离出部分功能和服务，如图 6.1 所示。

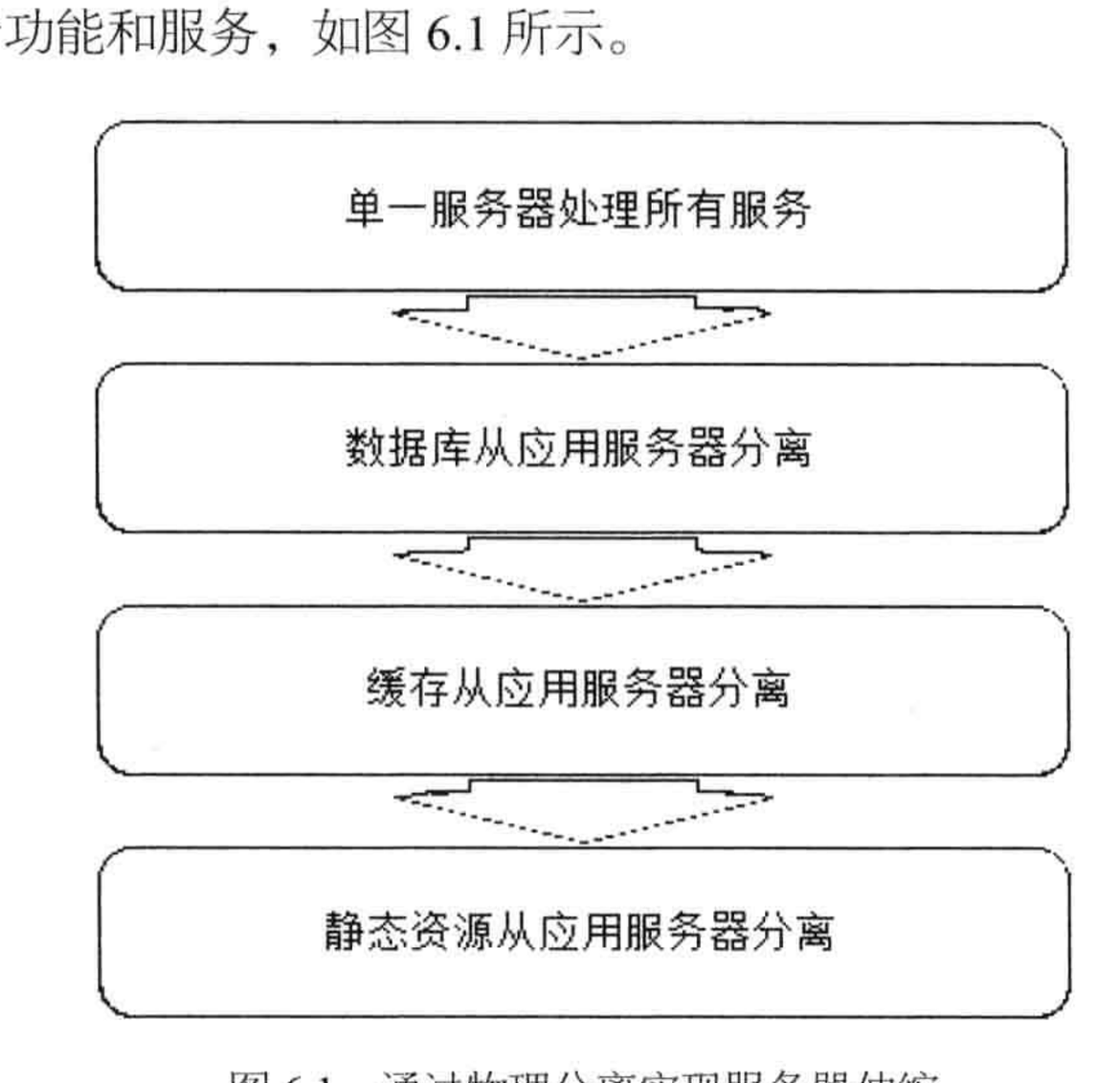

图 6.1 通过物理分离实现服务器伸缩

每次分离都会有更多的服务器加入网站，使用新增的服务器处理某种特定服务。事实上，通过物理上分离不同的网站功能，实现网站伸缩性的手段，不仅可以用在网站发展早期，而且可以在网站发展的任何阶段使用。具体又可分成如下两种情况。

纵向分离（分层后分离）：将业务处理流程上的不同部分分离部署，实现系统伸缩性，如图 6.2 所示。

图 6.2　通过纵向分离部署实现系统伸缩性

横向分离（业务分割后分离）：将不同的业务模块分离部署，实现系统伸缩性，如图 6.3 所示。

图 6.3　通过横向分离部署实现系统伸缩性

横向分离的粒度可以非常小，甚至可以一个关键网页部署一个独立服务，比如对于电商网站非常重要的产品详情页面，商铺页面，搜索列表页面，每个页面都可以独立部署，专门维护。

6.1.2　单一功能通过集群规模实现伸缩

将不同功能分离部署可以实现一定程度的伸缩性，但是随着网站访问量的逐步增加，即使分离到最小粒度的独立部署，单一的服务器也不能满足业务规模的要求。因此必须使用服务器集群，即将相同服务部署在多台服务器上构成一个集群整体对外提供服务。

当一头牛拉不动车的时候，不要去寻找一头更强壮的牛，而是用两头牛来拉车。

以搜索服务器为例，如果一台服务器可以提供每秒 1000 次的请求服务，即 QPS（Query Per Second）为 1000。那么如果网站高峰时每秒搜索访问量为 10000，就需要部署 10 台服务器构成一个集群。若以缓存服务器为例，如果每台服务器可缓存 40GB 数据，那么要缓存 100GB 数据，就需要部署 3 台服务器构成一个集群。当然这些例子的计算都是简化的，事实上，计算一个服务的集群规模，需要同时考虑其对可用性、性能的影响及关联服务集群的影响。

具体来说，集群伸缩性又可分为应用服务器集群伸缩性和数据服务器集群伸缩性。这两种集群由于对数据状态管理的不同，技术实现也有非常大的区别。而数据服务器集群也可分为缓存数据服务器集群和存储数据服务器集群，这两种集群的伸缩性设计也不大相同。

6.2 应用服务器集群的伸缩性设计

我们在本书第 5 章提到，应用服务器应该设计成无状态的，即应用服务器不存储请求上下文信息，如果将部署有相同应用的服务器组成一个集群，每次用户请求都可以发送到集群中任意一台服务器上去处理，任何一台服务器的处理结果都是相同的。这样只要能将用户请求按照某种规则分发到集群的不同服务器上，就可以构成一个应用服务器集群，每个用户的每个请求都可能落在不同的服务器上。如图 6.4 所示。

如果 HTTP 请求分发装置可以感知或者可以配置集群的服务器数量，可以及时发现集群中新上线或下线的服务器，并能向新上线的服务器分发请求，停止向已下线的服务器分发请求，那么就实现了应用服务器集群的伸缩性。

这里，这个 HTTP 请求分发装置被称作负载均衡服务器。

负载均衡是网站必不可少的基础技术手段，不但可以实现网站的伸缩性，同时还改善网站的可用性，可谓网站的杀手锏之一。具体的技术实现也多种多样，从硬件实现到软件实现，从商业产品到开源软件，应有尽有，但是实现负载均衡的基础技术不外以下几种。