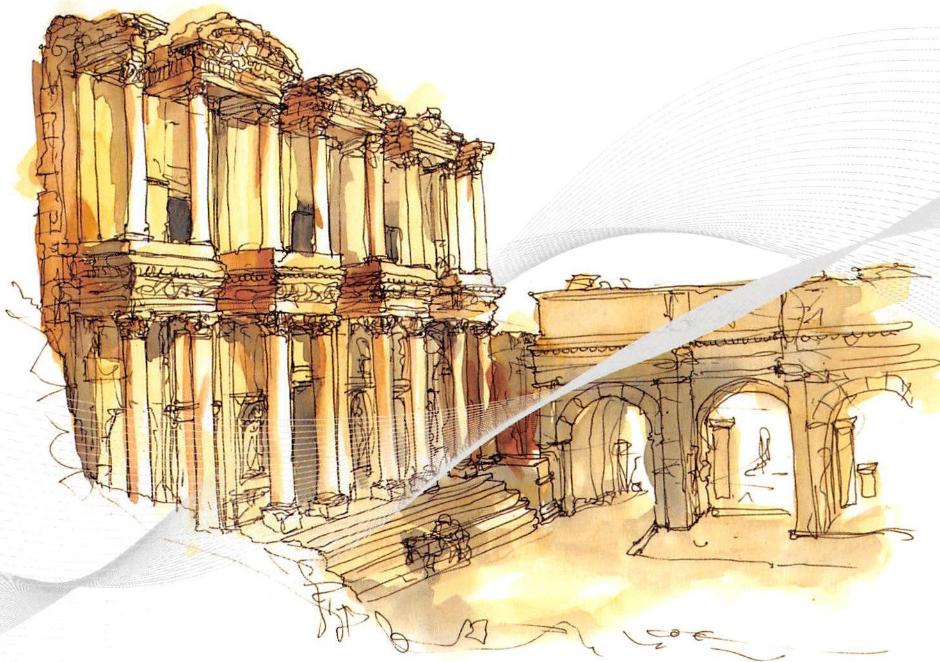
仅供非商业用途或交流学习使用

Broadview®
www.broadview.com.cn



大型网站技术架构演进与性能优化

许令波◎著

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅供非商业用途或交流学习使用

大型网站
技术架构演进与性能优化

许令波◎著

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

仅供非商业用途或交流学习使用

内 容 简 介

本书从一名亲历者的角度，阐述了一个网站在业务量飞速发展的过程中所遇到的技术转型等各种问题及解决思路。从技术发展上看，网站经历了 Web 应用系统从分布式、无线多端、中台到国际化的改造；在解决大流量问题的方向上，涉及了从端的优化，到管道、服务端，甚至基础环境优化的各个层面。

书中总结的宝贵经验教训可以帮助读者了解当网站遇到类似问题时，应如何思考不同的解决思路、为什么要这样做、并最终选择合适的方案。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

大型网站技术架构演进与性能优化 / 许令波著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34135-9

Ⅰ. ①大… Ⅱ. ①许… Ⅲ. ①网站－开发 Ⅳ.①TP393.092

中国版本图书馆 CIP 数据核字(2018)第 087106 号

责任编辑：刘　皎
印　　刷：北京画中画印刷有限公司
装　　订：北京画中画印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：13.5　字数：284 千字
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

仅供非商业用途或交流学习使用

好评袭来

君山经历了淘宝网发展速度和架构变化最快的时代，这是一个机会和挑战并存的时代，许许多多无法用常理理解的需求不断涌现，许许多多从未遇见过的问题横在面前，许许多多创新的解法横空出世！君山把传统的软件工程开发理念和新机遇下的技术创新相结合，在性能优化领域不断创新：小到字节码层面的优化、大到架构上的重建——他的探索工作在淘宝网的技术发展史上留下了痕迹。

君山做事情有几个特点：一是能把技术和业务相结合，在处理业务需求和问题时轻车熟路，在处理完业务需求的同时还会带来技术上的创新；二是善于推动技术创新落地，用自己的实践诠释了“创新只有被人使用、在业界形成潮流才算是真正的创新”这句话；三是善于总结思考，他每次都把技术和业务上遇到的问题和解法总结下来，并乐于分享，让团队共同成长！

——阿里巴巴研究员　小邪

做技术做到后期才会发现写代码并不是全部。随着业务的快速迭代，对系统的架构演进和相关技术的权衡会变得越来越重要，在不同的阶段会有不同的取舍。特别是大型系统，除了要考虑技术，还要考虑相匹配的组织架构、工程文化等因素——这些挑战是很难通过亲历来获取的，毕竟成功的大型系统不算太多。

仅供非商业用途或交流学习使用

作者曾是淘宝网一线的技术专家，亲身经历了淘宝网业务飞速增长的过程，并将其中的经验和学习的过程记录下来，完整地为我们展现了一个初级系统在演化成一个全球分布式的系统的过程中，从语言选择、分布式框架改造、平台化演进、系统优化到稳定性建设等关键过程的思考，内容翔实可信。在这些最佳实践中，技术点也许并不是最重要的，读者可收获多维度的启发和共鸣，推荐阅读！

——阿里云研究员　褚霸

一家伟大的互联网企业一般都离不开高超技术的支撑，而高超技术的养成又离不开每天迎面而来的各种挑战。本书作者有幸经历了淘宝网这些年的技术巨变，碰到了无数的问题，积攒了很多并发架构设计和性能优化的经验。好的架构是一个系统的根本，好的性能是一个系统稳定运行的保证，本书应该可以给大家带来不一样的收获。

——PerfMa CEO　你假笨（寒泉子）

针对 C 端用户的互联网业务是爆发式的、井喷式的，其带来的用户流量压力和对计算能力的要求也是非常惊人的，如何利用廉价的架构设计来部署分布式服务以应对亿级流量的场景是个非常严峻的问题。《大型网站技术架构演进与性能优化》一书讲解了高可用架构演化的进程，并提供了互联网架构性能优化的方法。正所谓互联网技术唯“快”不破——解决了性能问题，其他问题也就迎刃而解。如果你的业务正处于流量并发暴增与系统架构变革的十字路口，那么本书恰好就是你的“菜”。这是一本关于互联网高并发架构设计的优秀书籍，它从各角度剖析系统设计的演化与优化，循序渐进地将一系列复杂问题阐述得清晰、简单、易懂，是一本理论与实践相结合的实用书籍。

——《分布式服务架构：原理、设计与实战》、《可伸缩服务架构：框架与中间件》
作者　李艳鹏

对于一个高并发大流量网站的架构师而言，你的系统到底能够承受多高的并发、多大的流量，只有在你的系统经历了更高的并发、更大的流量以后才能知道。事前再多的设计、评审、测试、预演也只能让你相信，而不能让你知道。淘宝网作为全球最大的电子商务网站，每年的双 11 都会承受这个星球上可能是最大的并发访问压力，

仅供非商业用途或交流学习使用

那么淘宝的技术人员遇到了哪些挑战？做了哪些工作？感谢这本《大型网站技术架构演进与性能优化》，让我们一窥究竟。

——《大型网站技术架构：核心原理与案例分析》作者　李智慧

君山老师曾多次出席技术大会 SDCC 并担任讲师及出品人，为技术总监、架构师等参会者带去了很多干货的分享。实践出真知，任何脱离实际工作的讨论无疑在浪费宝贵的时间。作者在淘宝网经历了 Web 应用系统从分布式、无线多端、中台到国际化的改造；在解决大流量问题的方向上，积累了很多从端的优化，到管道、服务端，甚至基础环境优化的经验，这些助力他真正成为我们技术社区的明星专家，相信此书肯定会给广大的技术开发者带来一线的知识。

——CSDN 主编　钱曙光

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前言

从 1 亿到 50 亿的技术之路

从 2009 年到 2016 年，笔者非常幸运地经历了网站 PV 从 1 亿到 50 亿的飞速发展历程，在此过程中积累了一些大流量高并发网站架构设计和优化的经验。从技术发展来看，笔者经历了 Web 应用系统从分布式、无线多端、中台到国际化的改造；在解决大流量问题的方向上，积累了很多从端的优化到管道到服务端甚至到基础环境优化的经验。现在您手头这本书所介绍的内容，大部分是笔者看到、学到的，是亲身参与和实践的经验。

本书要表达的内容并不是简单罗列所做过的事情，而主要是帮助读者了解当网站遇到类似问题时，应如何思考不同的解决思路、为什么要这样做、如何做出最终的方案选择……其实每种架构的选择必然有它专属的现实场景，因此本书涉及的这些话题也不一定就是最完美的解决方案。但，我希望本书的分享能启发大家在解决类似问题时的思考和判断。

一、架构演进之路

本书分成两个部分。第一部分主要介绍整个网站由于业务发展所经历的几次主要的架构演进，包括：从 PHP 到 Java 的改造、分布式改造、无线化改造、中台的改造、国际化改造。第二部分主要介绍如何从不同的层次解决整个网站在大流量情况下遇到

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的性能瓶颈，包括端和管道的优化、应用层代码级优化、应用架构的优化、端到端的全链路优化。最后介绍在做架构和性能优化的过程中必须面对的稳定性问题——如何体系化地解决网站的稳定性，这是非常关键的。

阶段一 从 PHP 到 Java

很多网站早期都是基于 Linux+Apache+MySQL+PHP 架构的网站，从当时来看，这种非常流行的个人网站架构的确也非常匹配当时的发展状态。PHP 语言的特性是快速发布，从页面渲染到数据库访问，均可以在一个页面里全部搞定。

即使放到今天，这种架构仍然还有很多人在用，它的优点就是非常简单高效，但缺点也非常明显：扩展性和分布式不好，不适合企业级的、复杂业务逻辑的大规模协同开发。

随着网站的发展，大家觉得应该将 PHP 切换到 Java。为什么要切换到 Java 语言呢？一般来说，企业选择开发语言会有如下考虑。

（1）语言本身的特性。每种语言开发出来都有它的特性和所适合的场景，像 Python、PHP 这类脚本语言非常适合快速简单的开发方式，而 Java 则比较适合构建复杂业务逻辑的企业级开发，但是开发效率会比 PHP 要差一点。

（2）程序员队伍。企业选择何种开发语言，还要看市场上的人才队伍是不是足够大，是不是有很高层次的人才。是否有高层次的人才，取决于当前的行业老大是不是也在用这种语言，比如当前的顶级互联网公司如果在用 Java，那么自然这些公司的 Java 人才比较多，这样，他们的经验可以被快速复制到其他公司中。

（3）语言所对应的工具生态是否完善。一个语言是否有生命力，要看这个语言对应的生态工具是否完善，它的社区是否活跃。我们要用到各种工具，而我们也不可能自己去写每种工具，因此，是否能方便地利用开源工具，快速提升开发效率也是非常关键的。像现在很多大公司开源了很多 Java 的中间件产品，这些中间件可以直接拿来使用，就不需要再重新开发了。

综合以上因素，电商网站选择 Java 语言作为主要的系统开发语言是非常合适的。

从 PHP 切换到 Java 后，整个网站采用 WebX+EJB+iBatis+JBoss+Oracle（后面又将 EJB 改成 Spring）的架构，但是随着业务量的不断增大，存储层的瓶颈暴露出来。为了解决存储问题，就逐渐用上了非常昂贵的 IBM 小型机、Oracle 的数据库以及 EMC 的高端存储（IOE）；并对数据库做了分库的拆分，分布式缓存（Tair）也随之诞生，分布式文件系统 TFS 开始出现，CDN 也慢慢建立了。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

阶段二 分布式改造

所谓分布式改造，就是尽量让系统无状态化，或者让有状态的信息封装在一定范围内，以免限制应用的横向扩展。简单来说，就是即便某个应用的少数服务器"宕"掉，也不会影响整体业务的稳定性。

要实现应用的分布式改造必须先解决以下几个问题。

（1）把应用微服务化：即将大量粗粒度的应用逻辑拆小，做服务化改造。

（2）必须先建立分布式服务框架。必须具备分布式 RPC 框架、异步消息系统、分布式数据层、分布式文件系统和服务的发现、注册和管理。

（3）必须要解决分布式 Session 问题。

为了做业务的服务化改造，我们大量拆分了当时的业务系统，形成了商品中心、交易中心、用户中心、店铺中心。这些服务作为底层服务供上层的前台系统调用，此时的系统架构变成了如图 0.1 所示的形式。

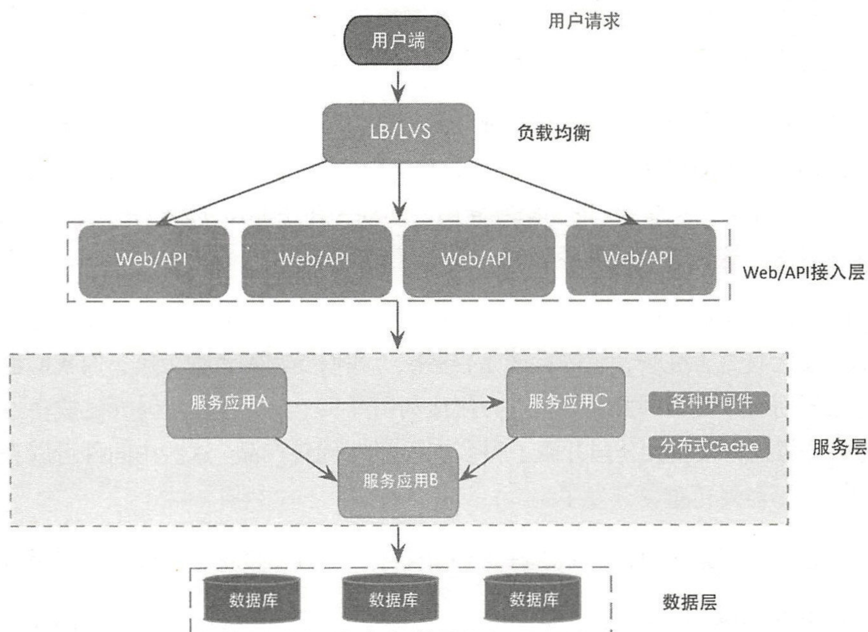


图 0.1　分布式改造后的架构

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

现在来看，系统的分布式改造为网站接下来 5 年的发展奠定了很好的基础，整个网站的扩展性非常好。几乎每个初创企业都必须经历一次分布式化的改造，才能为企业的长期发展奠定基础。笔者前面提及的几个重要的中间件产品和关键的分布式 Session 等技术在《深入分析 Java Web 技术内幕（修订版）》一书中有详细的介绍，感兴趣的读者可以自行去学习了解。

阶段三 无线化改造

到了 2013 年，无线技术已经非常火爆了。在此之前，无线的业务总是跟着 PC 走，基本上是 PC 做好后无线再复制一份，而且无线和 PC 还不是同一个前台系统，导致一个功能要做两遍，并且无线部门的开发人员本来就不多。这些给业务的发展带来很多问题。因此 2013 年底，启动了 All in 无线项目，目标就是用一套系统、一套架构快速支撑多端的个性化，并在服务端做了很多模块化和组件化的改造。

随着无线技术的发展，我们也开始尝试一些新技术，最具代表性的就是 Node。Node 在业界很火，前端同学非常推崇，而且它也大幅提升了前端的开发效率和体验。目前大家也多方尝试使用 Node 技术，并且用在一些业务线上。但是，从今天来看，Node 并没有达到我们想象中的发展前景。这其中有多种原因，本书的后续章节会专门介绍网站的无线化改造实践，也将分享 Node 在实际使用中的一些思考。

经过无线化改造的应用系统架构如图 0.2 所示。

阶段四 中台改造

中台这个概念早期是由美军的作战体系演化而来的，技术上的“中台”主要是指学习这种高效、灵活和强大的指挥作战体系。电商经过十几年的发展，组织已经庞大而复杂，业务不断细化拆分，也导致野蛮发展的系统越来越不可维护，开发和改造效率极低，也有很多新业务不得不重复造轮子，所以中台的目标是解决效率问题，同时降低创新成本。书中会有单独的一章介绍笔者看到、学到的中台的实践和思考。

阶段五 国际化

国际化一般会有两种思路：一种是一套原始代码部署到多个地方，各地的系统基本没有什么关联、保持相互独立，每个地方再根据本地实际情况做一些个性化的定制。一般来说，会精简原始代码，减少不必要的依赖。这种思路在一些跨国公司用得比较多，但是这个对技术要求比较低，不是我们介绍的重点。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

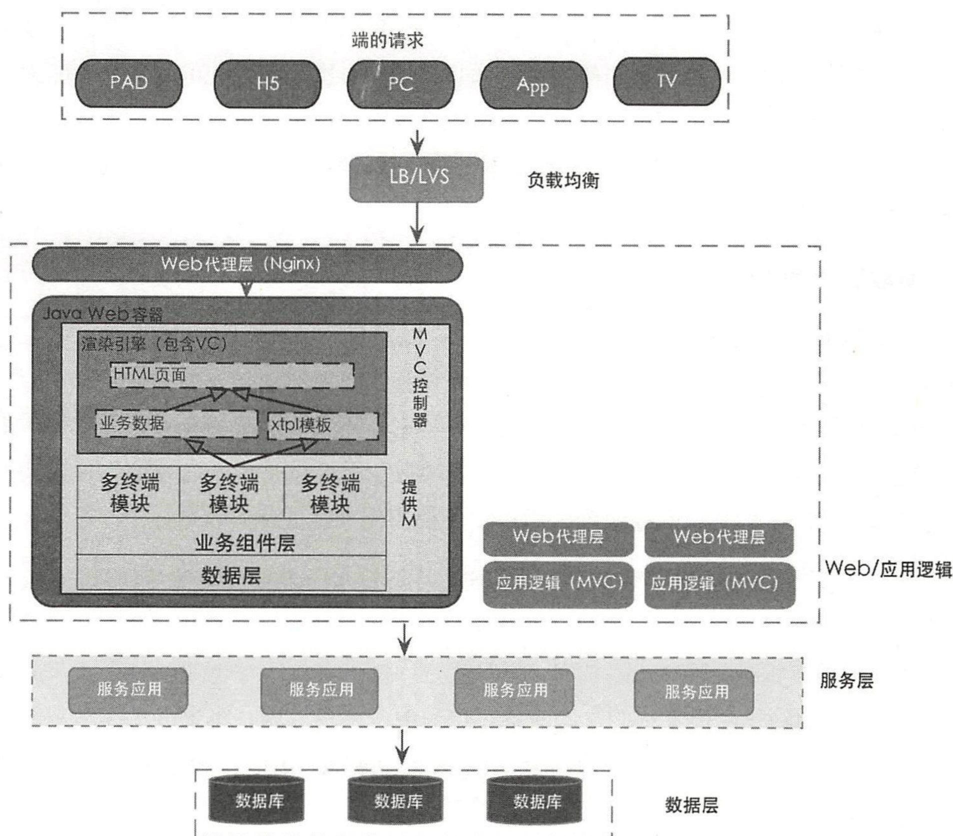


图 0.2　无线化改造后的应用系统架构

另一种思路就是我们要介绍的国际化，它主要解决如何将一套系统部署到多地的问题。一般国际化有两个发展阶段：第一个阶段是在国内实现了交易的单元化；第二个阶段是实现了中美的跨国部署。

国际化的本质仍然是要解决以下的通用问题：多语言问题、多时区问题、数据路由问题、全球数据的同步与复制问题。这些内容我们将在第 5 章介绍，提供一些可参考的通用思路。

二、挑战性能瓶颈

从第 5 章开始将介绍网站 PV 量从 1 亿到 50 亿的发展过程中应用系统遇到的各种性能瓶颈，以及我们的解决方案。这个过程可大致分为 3 个时间段，如图 0.3 所示。

仅供非商业用途或交流学习使用

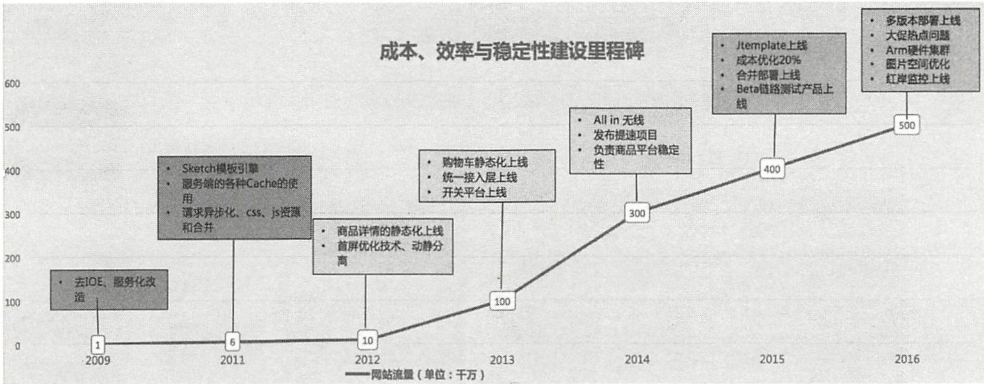


图 0.3　网站 PV 量从 1 亿发展到 50 亿的过程

第 1 个时间段：主要解决网站的易用性和扩展性问题。例如当时提的“去 IOE”就是将数据库从 Oracle 迁到 MySQL 上，通过分库分表来解决扩展性问题，同时做了很多网页端的优化工作，如 JavaScript（以下简称 JS）的异步加载、首屏优先渲染等。

第 2 个时间段：这段时间网站的流量增长非常迅速，每年双 11 的流量更是异常疯狂，系统出现了性能瓶颈。记得某次评估双 11 的流量，光详情页系统就需要增加上万台服务器，简直无法想象！所以必须考虑用非常规手段来优化性能。这个阶段我们主要通过静态化技术来解决读系统的性能瓶颈，大概用了 1~2 年的时间来持续迭代，直到实现了静态化系统改造才算彻底解决了性能问题，使系统能够支持上百万的 QPS。

第 3 个时间段：由于业务系统的复杂性上升非常快、业务的耦合度比较高，给系统的稳定性带来很大挑战，所以这个阶段进行了系统的稳定性建设，产出了很多稳定性工具和产品，另外性能优化也更朝着架构优化的方向发展。

我们还做了很多提升应用性能和开发效率的工作，从成本的角度来看，可以把这些工作归结为如图 0.4 所示的内容。

仅供非商业用途或交流学习使用

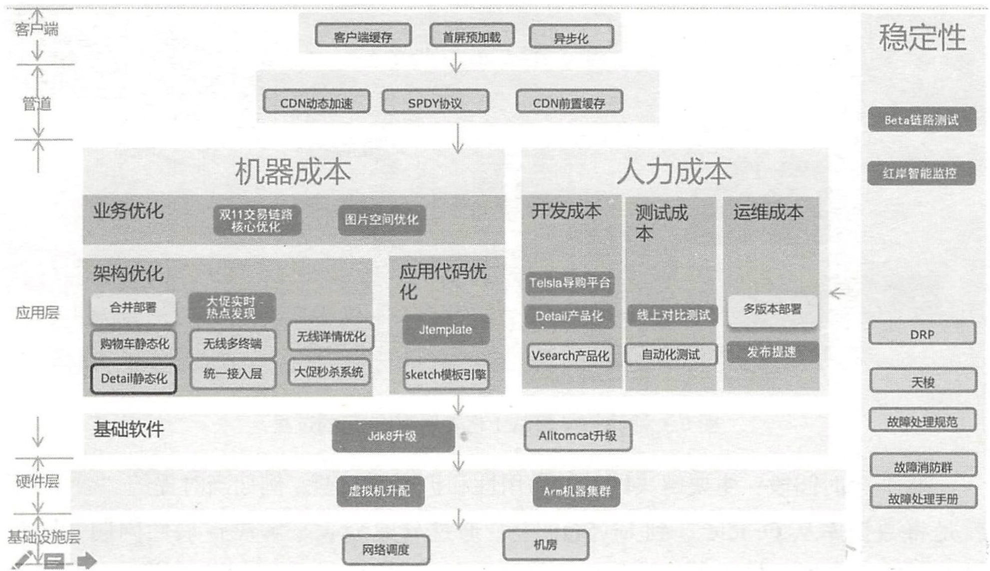


图 0.4　主要工作的归类

从端到中间的管道、再到应用层以及基础设施的优化，本书会分成 4 项内容来介绍：应用层代码优化、应用架构的优化、全链路的优化和基础设施的优化。

最后一章我们会介绍在整个网站飞速发展的同时，如何在进行架构的升级、应对突发的大流量这些极端的情况下，仍能保持整体网站的高可用和稳定性。至于如何做好双 11 的稳定性，这些内容已有多次的分享，本章会做一些系统的介绍。

三、组织架构的变迁

如果说技术是生产力，那么组织就决定了生产关系，而生产关系要适应生产力的发展。笔者当时所在的团队是业务平台，可以理解成技术大团队中的架构师团队，本身没有具体的业务产品，专注于解决一些横向的、架构上的问题。一般每个重要项目都会有一名架构师参与，业务平台和业务的关联比较直接、紧密。

中间件团队和业务平台相比，则和业务的距离稍微远一点，专注于业务开发过程中偏公共和通用的技术组件。这个团队和业务也相对比较紧密，业务开发中一些通用的技术组件，例如同步远程调用 RPC 框架、异步通信消息框架、业务动态配置框架、Web 开发框架、分布式数据层、分布式 Session 框架等，都是在业务开发中会用到且需要统一和规范使用的通用组件。

由于业务平台、中间件团队和业务的关系相对紧密，因此团队的组织关系一定要和业务开发捆绑在一起。在早期，这两个团队的 Leader 最好也是业务开发团队的 Leader，而且这个 Leader 要有强力的话语权。否则这两个团队的业务很难落地。架构师和中间件团队可以理解为业务开发中偏精英一点的团队，从长期发展来看，业务开发也有强烈的意愿要往偏技术的方向发展，所以他们的关系不一定是长期稳定的。

当技术团队人员扩张到接近 1000 人时，业务平台就慢慢解散了：因为随着技术团队分工越来越细，它的专业度也会越来越深，统一的架构师团队显然已经无法支撑这么多的业务团队了。在这种情况下，业务平台就被拆解分散到各个技术产品团队中，业务平台的 Leader 也成为新的技术团队的 Leader。该演进过程如图 0.5 所示。

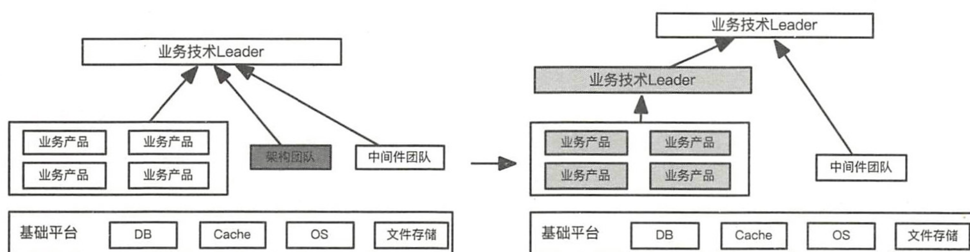


图 0.5　组织架构的可能演进

在图 0.5 中，中间件团队在建立伊始，一定要和业务团队保持紧密关联，同时，一些中间件产品最好是发展得比较完善后，再独立成图 0.5 右侧的结构，这会比较好。如果从一开始，中间件团队就保持了图 0.5 右侧的图的结构，在落地时会比较困难。

不过凡事都不是绝对的，人与人之间最大的问题其实就是信任问题。公司内部同事之间的协作最大的问题来自于利益的分配，团队之间合作最和谐的方式就是既有合作又有制约。这其中很多都是出于业务上的考虑。除了上面所介绍的，还有一个在组织结构上对技术影响比较大的事件——中台的提出——我们后面会介绍。

四、工程师文化的形成

在我看来，所谓的工程师文化是指就算没有 KPI 的考核，大家仍然还会认为某些事情是应该做的，而某些事情是不应该做的。例如，遇到问题必须追根溯源、敬畏线上的每次变更并力保稳定性、在技术沟通上对事不对人、简单坦诚、做事不给自己也不给别人挖坑等。

笔者感受比较深的是如果一家公司有一个类似双 11 这样的集中活动，那么对公司就是一次技术水平的大考，在这种压力下，在技术上要追求极致，工程师文化就自然而然地、有默契地形成了。因此，像双 11 这种能取得大家共识的公共事件可以在很大程度上推动技术的发展。

工程师文化和每个公司目前所处的环境和阶段也是息息相关的。如果一个创业公司每天面临着如何生存的问题，而员工却在考虑技术的设计是否完美，显然不是太现实。在这种情况下，业务开发的效率和稳定性会更重要。不过，长久来看，该还的债终究是要还的，一个好的工程师文化的建立对公司未来的良好发展至关重要。

目 录

1 构建大型网站：分布式改造 1

1.1 为什么要做分布式化 1
1.2 典型的分布式架构 2
1.3 分布式配置框架 4
1.4 分布式 RPC 框架 6
1.5 分布式消息框架 8
1.6 分布式数据层 11
1.7 分布式文件系统 12
1.8 应用的服务化改造 15
1.9 分布式化遇到的典型问题 16
1.10 分布式消息通道服务的设计 19
1.11 典型的分布式集群设计思路 21
1.12 总结 24

2 无线化：无线时代下的架构演进 26

2.1 无线环境下的新挑战 26
2.2 端的演进 28
2.3 无线链路的优化 32
2.4 服务端的演进 36
2.5 思考：开发语言选择的思考 44

2.6 总结 46

3 大型网站平台化演进：大中台小前台 49

3.1 为什么需要中台 49
3.2 什么是中台 53
3.3 提升中台的效率 55
3.4 中台是否能解决一切问题 64
3.5 总结 65

4 全球化下的网站演进：全球部署方案 66

4.1 国际化的背景 67
4.2 面临的技术挑战 68
4.3 全球部署的目标架构 69
4.4 何为单元化 69
4.5 单元化解决什么问题 70
4.6 单元化数据分片方案 70
4.7 数据路由方案 74
4.8 接入层路由 78
4.9 服务层路由 79
4.10 数据层路由 81
4.11 Sequence ID 的冲突问题 83
4.12 异地多活 84
4.13 多语言问题 85
4.14 多时区问题 86
4.15 全球数据同步与数据路由 89
4.16 通用版与定制版的选择 90
4.17 全球化部署中遇到的坑 91
4.18 总结 92

5 应用程序优化：代码级优化 93

5.1 优化思路 93
5.2 影响性能的关键因素 97
5.3 Java 特性的优化 102
5.4 减少并发冲突 104

5.5 减少序列化 105
5.6 减少字符到字节的转换 105
5.7 使用长连接 106
5.8 总结 106

6 应用架构探索：合并部署 108

6.1 什么是架构 108
6.2 什么是合并部署 110
6.3 能解决什么问题 112
6.4 如何解决 114
6.5 取得的效果 118
6.6 更进一步地做多版本部署 118
6.7 关于高密度部署的思考 121
6.8 总结 122

7 链路优化：大秒系统的极致优化思路 123

7.1 一些数据 123
7.2 热点隔离 124
7.3 动静分离 125
7.4 基于时间分片削峰 133
7.5 数据分层校验 134
7.6 实时热点发现 136
7.7 关键技术优化点 137
7.8 大促热点问题思考 140
7.9 总结 141

8 全局基础设施优化：资源调度优化 142

8.1 什么是资源调度 142
8.2 资源抽象层 144
8.3 物理资源调度 149
8.4 应用层调度 152
8.5 遇到的问题 155
8.6 总结 164

9 网站高可用建设：大型网站的稳定性建设 165

9.1 故障带来的影响 165
9.2 网站的可用性指标 166
9.3 稳定性建设思路 167
9.4 高可用体系化建设 171
9.5 研发人员的转变 180
9.6 稳定性组织保障 182
9.7 疑难问题排查思路 183
9.8 总结 190

附录 给新人成长的几点建议 191

参考资料 197

1 构建大型网站：分布式改造

一个创业公司起步时很可能就两台机器，一台 Web 服务器、一台数据库服务器，在一个应用系统中集成了所有功能模块，所以可能并不会考虑拆分系统、更不需要考虑分布式改造。但是，随着业务的发展、流量的增长，单应用已经远远不能满足业务需求，分布式化成为必由之路。

1.1 为什么要做分布式化

一个网站的技术架构早期很多都是一个 Linux+Apache+MySQL+PHP 的系统，随着业务的扩展和流量的爆发式增长，该系统很快达到了瓶颈，是不是一定要对它做分布式改造呢？其实我们在早期也尝试用过一些高端的服务器（IOE），但一方面价格昂贵，另一方面即便这样也阻挡不了瓶颈的到来，分布式改造成为必由之路。

所谓分布式改造，就是尽量让系统无状态化，或者让有状态的信息封装在一定范围内，以免限制应用的横向扩展。简单来说，就是当一个应用的少数服务器宕掉后，不会影响整体业务的稳定性。

要实现应用的分布式改造必须先解决好以下几个问题。

第一，应用需要微服务化。即将大量粗粒度的应用逻辑拆小做服务化改造。

第二，必须先建立分布式服务框架。必须具备分布式配置系统、分布式 RPC 框架、异步消息系统、分布式数据层、分布式文件系统、服务的发现、注册和管理。

第三，必须解决状态一致性问题。

下面我们将围绕以上问题展开描述。

1.2 典型的分布式架构

分布式架构与传统的单机架构最大的区别在于分布式架构能解决两个方向的扩展问题：一是横向扩展，二是纵向扩展。

横向扩展，主要用来解决应用架构上的容量问题。由于单台服务器能支撑的服务能力始终是有限的，所以我们在架构上就必须做到能够支持横向服务能力的扩展。最典型的横向扩展是图 1.1 所示的 Web/API 接入层，它在支持 1 亿 PV 和 10 亿 PV 时所需要的服务器数量必然是完全不一样的，因此要考虑当服务器不够用时，它也能支撑 PV 的无限增长。因此这两层一般都属于无状态的服务。

纵向扩展，主要解决业务的扩展问题。当业务不断扩展时，业务逻辑的复杂度也会不断上升，所以在架构上要能根据功能的划分进行纵向层次的划分。例如，Web/API 层只做页面逻辑或者展示数据的封装，服务层做业务逻辑的封装等。业务逻辑层还可以划分成更多的层次，以支持更细的业务的组合。

一个典型的分布式网站架构如图 1.1 所示。它将用户的请求通过负载均衡随机分配给一台 Web 机器，Web 机器再通过远程调用请求服务层。但是数据层一般都是有状态的，而数据要做到分布式化，就必须保证数据的一致性。要保证数据的一致性，一般都需要对最细粒度的数据做单写控制，因此要记录数据的状态、做好数据的访问控制等。

一个有状态的分布式架构如图 1.2 所示。分布式集群中一般都有一个 Master 负责管理集群中所有机器的状态和数据访问的规制等，为了保证高可用 Master 也有备份，Master 通常会把访问的路由规则推给实际的请求发起端，这样 Client 就可以直接和实际要访问的节点通信了，避免中间再经过一层代理。

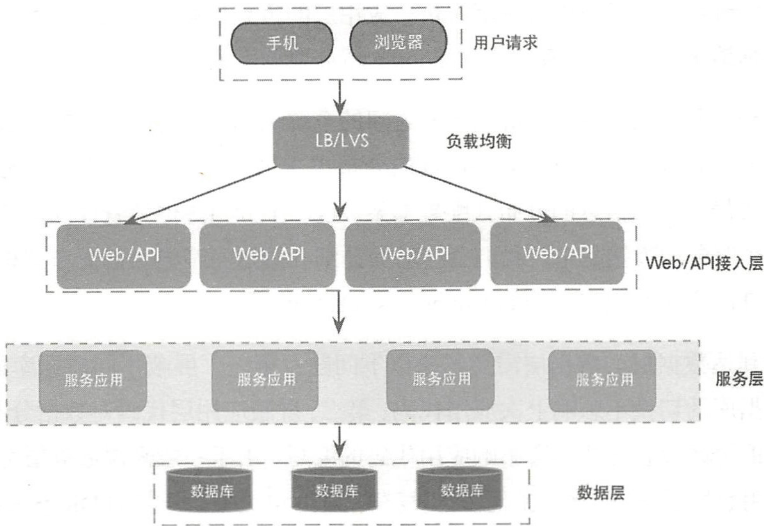


图 1.1　典型的分布式架构

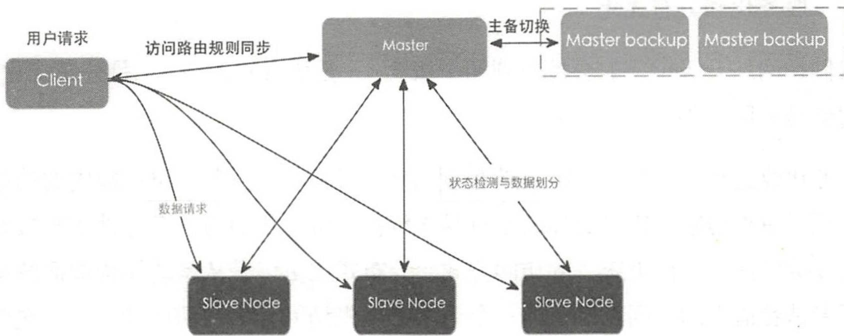


图 1.2　有状态的分布式架构

还有一种分布式架构是非 Master-Slave 模式而是 Leader 选举机制，即分布式集群中没有单独的 Master 角色，每个节点功能都是一样的，但是在集群的初始化时会选取一个 Leader 承担 Master 的功能。一旦该 Leader 失效，集群会重新选择一个 Leader。这种方式的好处是不用单独考虑 Master 的节点的可用性，但是也会增加集群维护的复杂度。

（1）需要分布式中间件

从前面典型的分布式架构上可以看出，要搭建一个分布式应用系统必须要有支持分布式架构的框架。例如首先要有一个统一的负载均衡系统（LB/LVS）帮助平均分配外部请求的流量，将这些流量分配到后端的多台机器上，这类设备一般都是工作在第

四层，只做链路选择而不做应用层解析；应用层的负载均衡可以通过 HA 来实现，例如可以根据请求的 URL 或者用户的 Cookie 精准地调度流量。

请求到达服务层，就需要解决服务之间的系统调用了。这时，需要在服务层构建一个典型的分布式系统，包括同步调度的分布式 RPC 框架、异步调度的分布式消息框架和解决静态配置信息的分布式配置框架。这三个分布式框架就像人体的骨骼和经络，把整个服务层连接起来。我们会在后面详细介绍这三个典型的分布式框架（分布式框架的开源产品有很多，例如 Dubbo、RocketMQ 等）。

请求到达数据层。数据层需要解决以下问题：第一，屏蔽不同数据库的差异性，使底层数据库的切换不影响上次应用代码；第二，屏蔽应用层代码对数据分布的感知，使对数据的分区或者分片不会影响应用代码的编写。由于一般来说数据层都是有状态的，所以用数据层解决分布式问题会更复杂、难度也更大。开源的 DRDS 等都是用于解决这类问题的。

（2）服务化和分布式化

我们在网站升级中一般会接触到两个概念：一是服务化改造；二是分布式化改造。那么它们是一回事吗？

服务化改造更多是从业务架构的角度出发，目的是将业务做更细粒度的功能拆分，使业务逻辑更加清晰、边界更加清楚且易于维护；服务化的另一个好处是收敛业务逻辑，通过接口标准化提供统一的访问方式。分布式化更多是从系统架构层面的角度出发，更多是看请求的访问路径，即一个请求必须先访问什么再访问什么、一次访问要经过哪些步骤才能最终有结果等……因此，这是两个不同层面的工作。

1.3 分布式配置框架

分布式配置框架可以说是其他分布式框架的基础，因为在分布式系统中要做到所有机器节点都完全对等几乎是不可能的，必然有某些机器或者集群存在某些差异，但同时又要保证程序代码是一份，所以解决这些差异的唯一办法就是差异化配置——配置框架就承担着这些个性化的定制功能：它把差异性封装到配置框架的后台中，使集群中的每台机器节点的代码看起来都是一致的，只是某些配置数据有差异。

配置框架的原理非常简单，即向一个控制台服务端同步最新的一个 K/V 集合、JSON/XML 或者任意一个文件，配置信息可以是在内存中的也可以是持久化的文件。

它的最大难点在于集群管理机器数量的能力和配置下发的延时率。

如图 1.3 所示，分布式配置框架一般有以下两种管理方式。

其一，拉取模式。拉取模式就是 Client 集群主动向 ConfigServer 机器询问配置信息是否有更新，如果有则拉取最新的信息更新自己。这种模式对配置集群来说比较简单，不需要知道 Client 的状态也不需要管理它们，只需处理 Client 的请求就行了，是典型的 C/S 模式。缺点是 ConfigServer 没法主动及时更新配置信息，必须等 Client 请求时再更新。一般 Client 都会设置一个定时更新周期，通常是几秒钟。

其二，推送模式。这种模式是当 ConfigServer 感知到配置信息变化时主动把信息推送给每个 Client。它需要 ConfigServer 感知到每个 Client 的存在，需要保持和它们之间的心跳，这使得 ConfigServer 的管理难度增加了，当 Client 的数据很大时，ConfigServer 有可能会成为瓶颈。

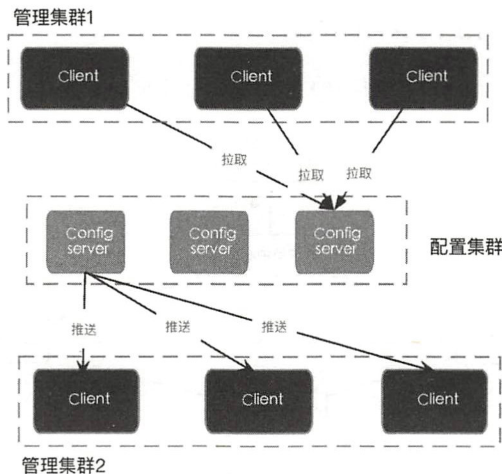


图 1.3 分布式配置框架

在实际的应用场景中，一般而言，如果对配置下发延时率比较敏感而且 Client 数据不是太大（千级别）时，推荐使用推送模式，像 ZooKeeper 就是这种框架（我们在后面的小节会介绍几种分布式管理的场景）；而当 Client 数据在万级以上时推荐使用拉取模式。不过，不管运用哪种模式我们都要考虑以下两个问题。

（1）由于是典型的 Master/Slaver 模式，要求在数据下发时要控制节奏，不要让 ConfigServer 的网卡成为瓶颈，尤其是当下发的数据量较大时。

（2）要保证配置下发的到达率，即 ConfigServer 需要精确地知道每个 Client 是否

已经是最新的配置数据。一般解决这个问题的办法是给每次配置信息的更新增加一个版本号，然后Client在每次要更新时都用这个版本号和ConfigServer的最新数据比较，如果小于ConfigServer再自我更新。

分布式配置框架是最简单的管理 Client 机器及相应配置下发功能的框架，因此，它也很容易发展成带有这两种属性的其他的工具平台，如名字服务、开关系统等。

1.4 分布式 RPC 框架

应用系统要做分布式改造，必须先要有分布式 RPC 框架，否则将事倍功半，为什么呢？这就像盖楼一样，如果没有先搭好骨架的话，很可能就是给自己“埋坑”。要做好分布式 RPC 框架需要实现服务的注册、服务发现、服务调度和负载均衡、统一的 SDK 封装。当前 Java 环境中分布式 RPC 框架如 Dubbo、HSF 都是比较成熟的框架，但是其他语言像 PHP、C++还不多。

一个通常意义上的 RPC 框架一般包含如图 1.4 所示的结构。

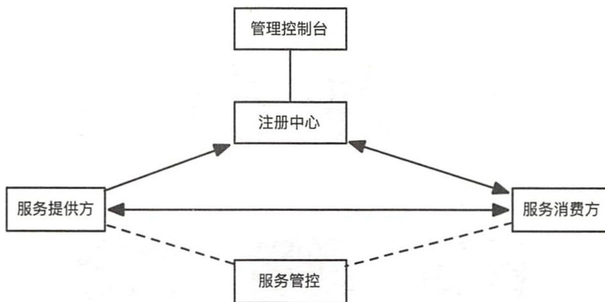


图 1.4 RPC 框架

（1）服务注册

服务要能被发现，必须先注册。服务的注册对 Java 程序来说非常简单，只需要在应用启动时调用 RPC 框架直接向服务端注册就行，一般需要传递类名、方法名、参数类型以及版本号。

由于语言上的一些限制，用 PHP 来做服务的注册和发现相比之下要困难很多。由于 PHP 不像 Java 那样很容易地支持长连接，所以在 PHP 文件启动时很难像 Java 程序那样在初始化函数里完成注册。对 PHP 来说目前有两个办法：第一个办法是写一个 FPM 的扩展，在第一次 FPM 初始化时完成服务的注册。但是如果一个 FPM 部署了多

仅供非商业用途或交流学习使用

个服务模块，那么如何区分也是一个难题；第二个办法是在 PHP 模块里手动配置一个要注册的服务名录列表，在 PHP 打包发布时进行注册。

服务注册后，服务发布者所在的机器需要和注册中心保持心跳以维持自己处于一直可以提供服务的状态，否则注册中心就会踢掉机器。对 PHP 来说，维持 RPC 连接是个麻烦事，所以一般会在本机另外再起一个 proxy agent 或者直接发送 HTTP 请求，但是这样做比较消耗性能。

（2）服务发现

服务发布方注册服务后，服务调用方就要能够发现服务。服务发现（如图 1.5 所示）最重要的一个目标就是要把服务和提供服务的对应机器解耦，而不是通过机器的 IP 寻找服务。

服务调用方只需要关心服务名，不用关心该服务由谁提供、在哪儿提供、是否可用……服务发现的组件会把这些信息封装好。

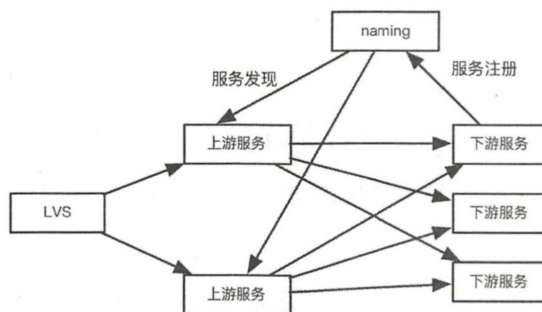


图 1.5 服务发现

对 Java 语言来说。服务发现就是把对应的服务提供者当前能够存活的机器列表推给服务调用者的机器的内存，真正发起调用时随机选取一个 IP 就可以发起 RPC 调用。

对 PHP 来说，如何把服务发布者的机器列表推给调用方也很麻烦，目前的解决方案更倾向于在本机的 Agent 中完成地址列表的更新，然后真正调用时再查询 Agent 更新的本地文件，并从本地文件中查找相应的服务。

（3）服务调度和负载均衡

服务注册和发现完成后，就要处理服务的调度和负载均衡了（如图 1.6 所示）。服务调用有两个关键点：一是要摘除故障节点，二是负载均衡。

仅供非商业用途或交流学习使用

- 摘除故障节点。这对 Java 来说比较容易处理：一旦有机器下线后，很容易更新地址列表。但对 PHP 来说就只能定期从 Agent 中拉取最新的地址列表，做不到像 Java 那样实时。
- 负载均衡。负载均衡需要将调用方的请求平均分布到不同的服务提供者的机器上，一个最简单的算法就是随机选取，做得复杂一点可以给每个提供者 IP 设置一个权重，然后根据权重选取。

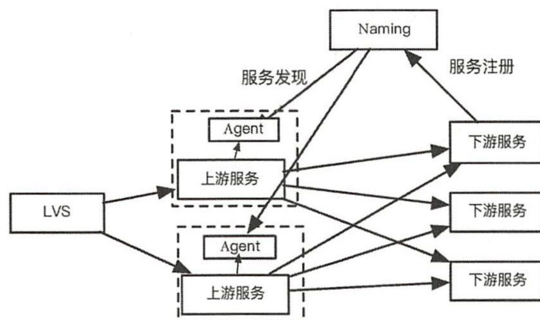


图 1.6　服务调度与负载均衡

（4）统一的 SDK 封装

服务框架需要提供一个统一的客户端和服务端的标准接口规范，这样可以减少业务开发的重复工作量，例如 SDK 会统一封装通信协议、失败重试以及封装一些隐式参数传递（trace 信息）。

Java 通过提供一个统一的 jar 包，封装了服务发布和调用的接口，业务层只要做些简单的配置就能方便地调用服务，例如 Java 一般都会配置一个 Spring 的 Bean。

对其他语言来说，运用 IDL 规范是个好选择。在 thrift 的基础上，修改 code-gen，生成 struct（class）的 read 和 write，生成 Client 和 server 插件框架，并基于 thrift 的 lib 提供 Binary Protocol、TCP Transport。

1.5　分布式消息框架

一个分布式应用系统中，除了 RPC 调用之外，还需要在应用之间传递一些消息数据，这时分布式消息中间件就成为必需品。消息中间件主要用于异步和一个 Provider 多个 Consumer 的场景中。消息可分为实时消息和延时消息。

仅供非商业用途或交流学习使用

1. 实时消息

实时消息就是当消息发送后，接受者能实时消费的消息（如图 1.7 所示），很多开源的消息中间件如开源的 RocketMQ，Apache Kafka 等都是比较成熟的消息中间件。

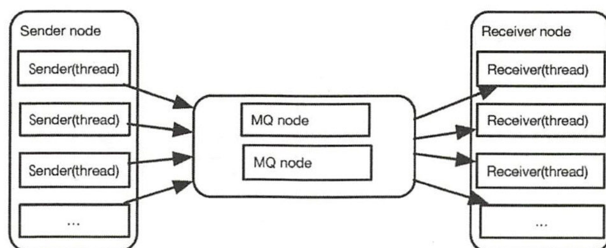


图 1.7 实时消息

（1）异步解耦

异步解耦的好处体现在多个方面：可以分开调用者和被调用者的处理逻辑，降低系统耦合，解决处理语言之间的差异、数据结构之间的差异以及生产消息和消费者的速度差异（削峰填谷）。

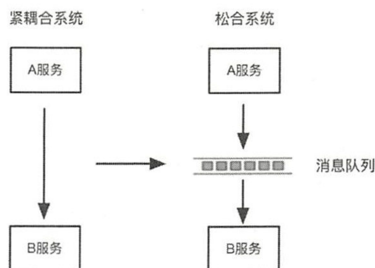


图 1.8 异步消息

通过中间的消息队列服务，可以做很多事情，但是要保证以下两点。

- 最终的一致性。一致性要求不能丢消息，保证消息最终可达，如果最终不可达要反馈给发送方。
- 消息的有序性。有序性是指消息的先后顺序，先后的顺序是按消息的发送时间排序。

（2）多消费端

一个消息被多个订阅者消费是典型的一种应用场景（如图 1.9 所示），多消费端非常适合用在单一事情触发的场景中。例如当一个订单产生时，下游会对这个订单做很

仅供非商业用途或交流学习使用

多额外的处理，而消息的生产者对这些消息的消费者根本不会关心，非常适合用在一个大型的异构系统中。

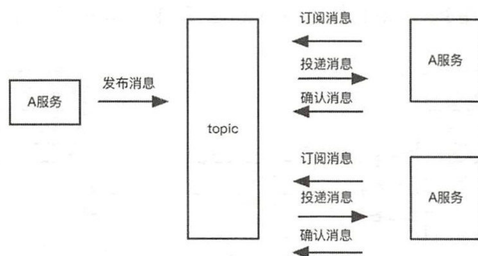


图 1.9　多端订阅消费

在此场景中我们会遇到下面这些典型问题。

- 确认消息是否被消费。当出现多个消费者时，需要确定哪些消息发送成功，哪些消息发送失败，是否要重新发送……所以消息队列中需要增加消费确认标识，以确定哪些消费端已经成功消费了消息，而投递失败的需要重新投递。
- 消息队列需要有容错能力。当某个消息失败后，需要消息队列有容错能力，保证消费端恢复后能重新投递消息，所以消息队列要有能力保存一定的消息量。

2. 延时消息

除了实时消息外，延时消息用得也比较广泛。典型的例子比如一张电影票的订单产生后，用户在 15 分钟后仍然没有付款，那么系统会要求在 15 分钟后取消该订单，释放座位（如图 1.10 所示）。这种场景非常适合用延时消息队列来处理。

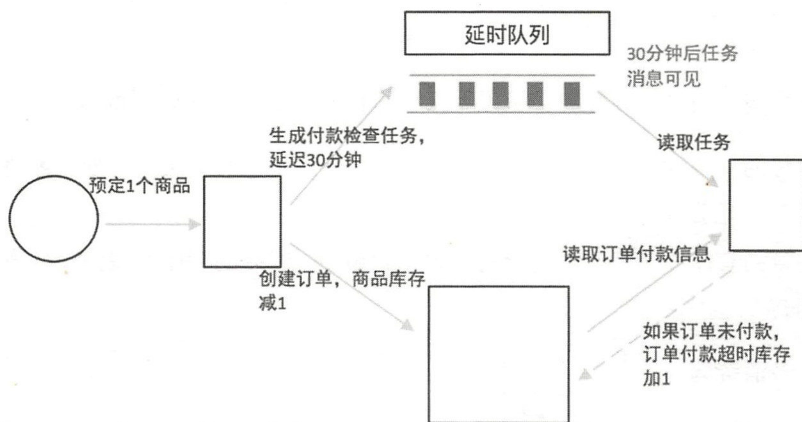


图 1.10　延时消息队列

仅供非商业用途或交流学习使用

延时消息在技术实现上比实时消息队列要更难，因为它需要增加一个触发事件，而这个触发事情有时候不一定是时间触发事件，还有可能是其他消息事件触发，这样导致它所承担的业务逻辑会更重，架构也会更复杂。

延时消息的核心是需要有一个延时事情的触发器，此外还必须解决消息的持久化存储问题，其他方面和实时消息队列差不多。总体来说，所有的消息队列都必须要解决最终一致性、高性能和高可靠性问题。

3．几种常见的消息中间件

在 http://queues.io/上几乎列出了当前大部分开源的消息队列，每个产品各有特点，适用于不同应用场景，适合的才是最好的。

1.6　分布式数据层

分布式数据层主要解决数据的分库分表、主备切换以及读写分离等问题，统一封装数据库的访问细节，如建立连接中的用户名和密码、连接数、数据类型的转换等信息。

（1）分库分表

分布式数据层最重要的功能是对数据做分库分表处理（如图 1.11 所示），尤其对互联网公司来说，数据量的不断增长要求切分数据是相当平常的任务。

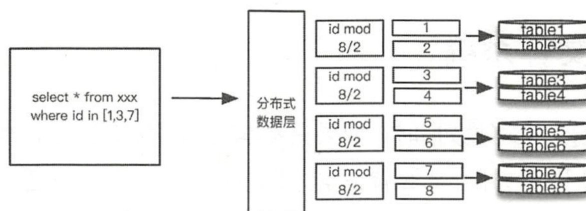


图 1.11　分库分表

当一条或者一批 SQL 提交给分布式数据层，并根据某些标识进行规制运算后，应该将这些 SQL 分发给规则被命中的机器去执行并返回结果。这里最重要的是数据分片的规制要对开发透明，即写 SQL 的同学不用关心他要请求的数据到底在哪台机器上，当我们改变数据分片规制时，只需要修改路由规则而无须修改 SQL。

所以很显然该分布式数据层需要解析用户的 SQL，并且有可能会重写 SQL（例如

仅供非商业用途或交流学习使用

修改表名或者增加一些 HINT 等信息）。

（2）主备读写分离

数据库的读写分离是常见的操作，如图 1.12 所示。由于数据库资源非常宝贵，为了保证数据库的高可用一般都会设置一主多备的架构；为了充分利用数据库资源，都会进行读写分离，即写主库读从库。在同机房的场景下，数据库主从复制的延迟非常低，对应用层没有什么影响。

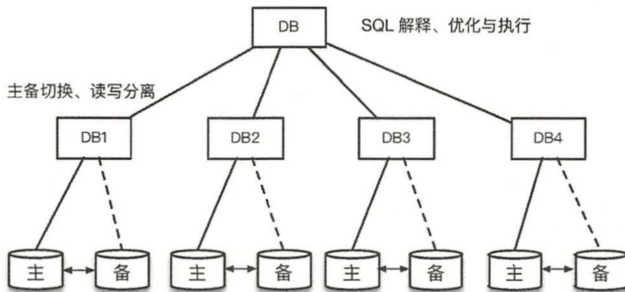


图 1.12　数据库读写分离

在原理上主从的读写分离比较简单，就是拆开用户的写请求和读请求，并分别路由到不同的 DataSource 上。在这种场景下要注意读写一致性的问题。在某些场景如双 11 抢单时，用户下完单立即查询下单是否成功，如果查询从库延时会比较大，用户很可能看不到下单成功界面从而重复下单，要有保障机制避免此类问题发生。

1.7　分布式文件系统

有些应用需要读写文件数据时，如果只写本机的话那么就会和本机绑定，这样这个应用就成为有状态的应用，那么就很难方便地对这个应用进行迁移，水平扩展也变得困难。

不仅是文件数据，一些缓存数据也存在类似问题。现在很多的分布式缓存系统 Redis、Memcache 等就是用于解决数据的分布式存储问题的。

当前开源的分布式文件系统很多，像开源的 TFS、FastDFS、GFS 等，它们主要解决的是数据的高可用性和高性能问题，下面我们介绍一个分布式文件系统 Seaweedfs 的设计，它的设计比较巧妙，很有启发性（如图 1.13 所示）。

仅供非商业用途或交流学习使用

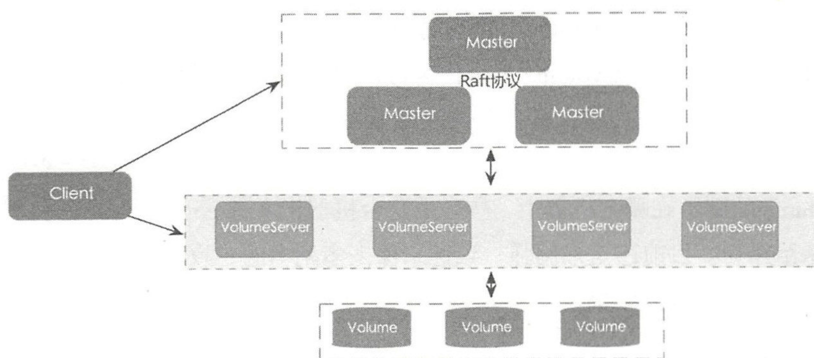


图 1.13 一种分布式文件系统架构

Seaweedfs 文件系统有 3 个非常重要的组成部分，分别是 Master、VolumeServer 和 Volume。

- Master 管理多个 VolumeServer，它的工作是分配某文件到 VolumeServer 中，并维护文件的副本，它通过 Raft 协议保证一致性。
- VolumeServer 管理多个 Volume。每个 Volume 包含固定大小的存储空间，每个 Volume 可存储多个文件直到 Volume 的整个存储空间被写满为止。Master、VolumeServer 和 Volume 只是逻辑上的划分，可以运行在同一台机器上。
- 每个 Volume 表示一个固定大小的存储空间。

假设一个 Volume 有 30GB 的容量，被分配一个唯一的 32bit 的 VolumeID；每个 VolumeServer 维护多个 Volume 和每个 Volume 剩余可写的存储空间，并上报给 Master；Master 维护整个集群中当前可写的 Volume 列表，一旦 Volume 写满就标识为只读。如果整个集群中没有可写的 Volume，那么整个集群都将不可写，只能通过扩容增加新的 VolumeServer。

Client 要上传一个文件首先需要向 Master 申请一个 fid，Master 会从当前可写的 Volume 列表中随机选择一个。和大多数分布式文件系统一样，这个 fid 就是表示文件在集群中的存储地址，最重要的是 fid 的前 32bit，代表的是 VolumeID，表示该文件具体存储在哪个 Volume 中并返回此 fid 应该存储的 VolumeServer 的机器地址。

典型的一个上传文件请求如 curl -F "file=@/tmp/test.jpg" "192.168.0.1:9333/submit"，返回{"fid":"3,01f83e45ff","fileName":"test.jpg","fileUrl":"192.168.0.1:8081/3,01f83e45ff","size":12315}。Client 拿到 fileUrl 再将文件实际上传到 192.168.0.1:8081 的 VolumeServer 中。

仅供非商业用途或交流学习使用

文件的多副本也是在 Master 上管理的，巧妙的地方在于文件的多副本不是以单个用户的文件为单位而是以 Volume 为单位进行管理。例如上面的“3,01f83e45ff”这个 fid，它是存在 3 的 Volume 中，那么这个 3 Volume 会有一个副本，即在其他的 VolumeServer 上也有一个 3 的 Volume，那么当 test.jpg 上传到 192.168.0.1:8081 时，它会查询 Master 这个 Volume 的副本在哪台机器上，然后由这台机器把文件 copy 到对应的机器上，再返回结果给用户。目前还是采用强一致性来保证多副本的一致性，如果某个副本上传失败则返回用户失败的结果。

扩容比较简单，当集群中所有的 Volume 都写满时，再增加一些 VolumeServer 并增加若干 Volume，Master 会收集新增的 VolumeServer 中空闲的 Volume 并加入到可写的 Volume 列表中。

如果有机器挂掉的话，由于 Volume 是有备份 Volume 的，所以只要存在一份 Volume，Master 都会保证能够返给用户正确的请求。这里需要注意的是 Volume 的多备份管理并没有主次的概念，每次 Master 都会在可用的多备份中随机选择一个返回。假设 3 这个 Volume 的副本分别在 192.168.0.2:8081 和 192.168.0.3:8081 上，那么如果 192.168.0.3:8081 机器“宕”掉，那么这个 3 对应的 Volume 就会被设置为只读，实际上 192.168.0.3:8081 上所有对应的 Volume 都会被设置为只读——只要某个 Volume 的副本数减少，都会禁止再写。

综上，这个文件系统的设计思路可以总结成以下两点。

第一，通过 Volume 作为基本的存储管理单元，类似作为一个“集装箱”，一旦“集装箱”装满了，就不再写入而转为只读。每个存入的文件都被放在一个 Volume 中，并且返回一个文件名；存储系统不再维护原始文件名与生成的文件名的对应关系。文件系统的一致性是以 Volume 的单位进行保证的，并且每个 Volume 可以独立存储和移动。

第二，分布式文件的路由信息是由 Master 来管理和分配的，而且每次请求都需要强依赖 Master，Master 的高可用则由 Raft 来协调。

到目前为止，该文件系统 0.7 的版本还不是太完善，表现在没有一个很完善的 Client 程序来处理 Master 到 VolumeServer 之间的跳转，一般需要自己写。但是它最吸引人的地方就是 Volume（集装箱式的设计），这个设计比较简单和独立，尤其是管理比较方便。当然，大部分分布式系统的设计也都有相似之处。

仅供非商业用途或交流学习使用

1.8 应用的服务化改造

解决好跨应用的连接和数据访问后，我们的应用也要做好相应的改造，如应用分层的设计、接口服务化拆分等。

（1）应用分层设计

应用分层设计很有必要。例如最起码要把对数据库的访问统一抽象出来形成数据层，而不是直接在代码里写 SQL——这会使重构应用和水平拆分数据库非常困难。

我们通常从垂直方向划分应用，分成服务层、业务逻辑层和数据层，每一层尽量做到解耦：上层依赖下层，而下层不要反向依赖上层。

应用分层最核心的目的是每个层都会封装一些信息、完成一些特定的功能需求，层与层之间通过接口交互，而且交互的数据是清晰和固定的，做到隔离和交互。可以从以下两个方向判断分层是否合理。

第一，如果我要增加一些新需求或者修改某些需求时，是否能清楚地知道要到哪个层去完成，换句话说，这些分层的职责是否清晰。

第二，如果每个层对我的接口不变，那么每个层内部的修改是否会导致其他层也发生修改，即每个层是否做到了收敛。

分层设计中最怕的就是在接口中设计一些超级数据结构，如传递一个对象，然后把这个对象一直传递下去，而且每个层都可能修改这个对象。这种做法导致两个问题：一是一旦该对象更改，所有层都要随之更改；二是无法知道该对象的数据在哪个层被修改，在排查问题时会比较复杂。因此，在设计层接口时要尽量使用原生数据类型如 String、Integer 和 Long 等。

（2）微服务化

微服务化，是从水平划分的角度尽量把服务分得更细，每个业务只负责一个功能单元，这样可以把这些微服务组合成更大的功能模块。也就是有目的地拆小应用，形成单一职责从而提升系统可维护性、扩展性和开发效率。

图 1.14 所示是基于 Spring Boot 构建的一个典型的微服务架构，它按照不同功能将大的会员服务和商品服务拆成更小原子的服务，将重要稳定的服务独立出来，以免经常更新的服务发布影响这些重要稳定的服务。

仅供非商业用途或交流学习使用

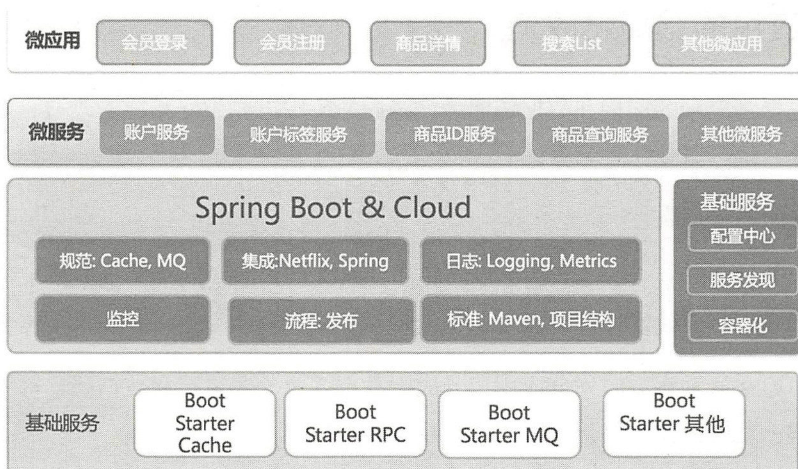


图 1.14　微服务架构

1.9　分布式化遇到的典型问题

在大型分布式互联网系统中，Session 问题是典型的分布式化过程中会遇到的难题。因为 Session 数据必须在服务端的机器中共享，并要保证状态的一致性。该问题在《深入分析 Java Web 技术内幕（修订版）》的第 10 章中有详细的介绍。

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，它是一个为分布式应用提供一致性服务的软件，所提供的功能包括：配置维护、域名服务、分布式同步、组服务等。下面我们介绍一下典型的分布式环境下遇到的一些典型问题的解决办法。

1. 集群管理（Group Membership）

ZooKeeper 能够很容易地实现集群管理的功能，如图 1.15 所示。如果多台 Server 组成一个服务集群，那么必须有一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，就必须知会集群中的其他集群，并重新分配服务策略。同样，当集群的服务能力增加时，就会增加一台或多台 Server，这些也必须让“总管”知道。

仅供非商业用途或交流学习使用

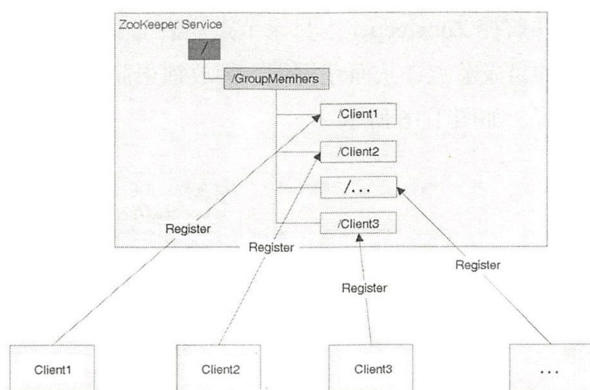


图 1.15 集群管理结构图

ZooKeeper 不仅能够维护当前集群中机器的服务状态，而且能够选出一个“总管”，让“总管”来管理集群——这就是 ZooKeeper 的另一个功能 Leader Election。

它的实现方式是在 ZooKeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, Boolean watch) 方法并设置 watch 为 true。由于是 EPHEMERAL 目录节点，当创建它的 Server 死去时，这个目录节点也随之被删除，所以 Children 将会变化；这时 getChildren 上的 Watch 将会被调用，通知其他 Server 某台 Server 已死了。新增 Server 也是同样的原理。

那么，ZooKeeper 如何实现 Leader Election，也就是选出一个 Master Server 呢？和前面的一样，每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号——我们可以选择当前最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于它是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以在当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免传统上单 Master 容易出现的单点故障问题。

2. 共享锁（Locks）

在同一个进程中，共享锁很容易实现，但是在跨进程或者不同 Server 的情况下就不好实现了。然而 ZooKeeper 能很容易地实现这个功能，它的实现方式也是通过获得锁的 Server 创建一个 EPHEMERAL_SEQUENTIAL 目录节点，再通过调用 getChildren 方法，查询当前的目录节点列表中最小的目录节点是否是自己创建的目录节点，如果是自己创建的，那么它就获得了这个锁；如果不是，那么它就调用 exists(String path,

Boolean watch)方法，并监控 ZooKeeper 上目录节点列表的变化，直到使自己创建的节点是列表中最小编号的目录节点，从而获得锁。释放锁很简单，只要删除前面它自己所创建的目录节点即可，如图 1.16 所示。

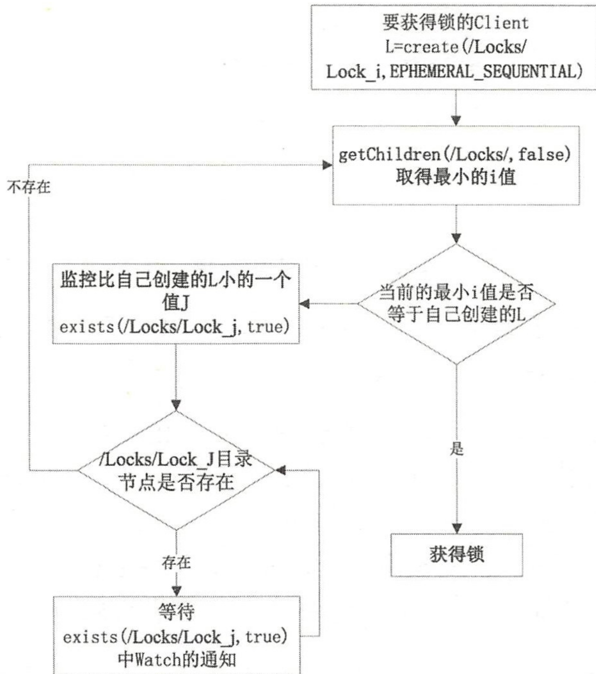


图 1.16　ZooKeeper 实现共享锁的流程图

3. 队列管理

ZooKeeper 可以处理以下两种类型的队列。

其一，同步队列。即当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

其二，队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

用 ZooKeeper 实现同步队列的实现思路如下。

- 创建一个父目录/synchronizing，每个成员都监控标志（Set Watch）位目录/synchronizing/start 是否存在，然后每个成员都加入这个队列；
- 加入队列的方式就是创建/synchronizing/member_i 的临时目录节点，之后每个成员获取/ synchronizing 目录的所有目录节点，也就是 member_i；

- 判断 i 的值是否已经是成员的个数，如果小于成员个数等待/synchronizing/start 的出现，如果已经相等就创建/synchronizing/start。

我们用图 1.17 的流程图来直观地展示该过程。

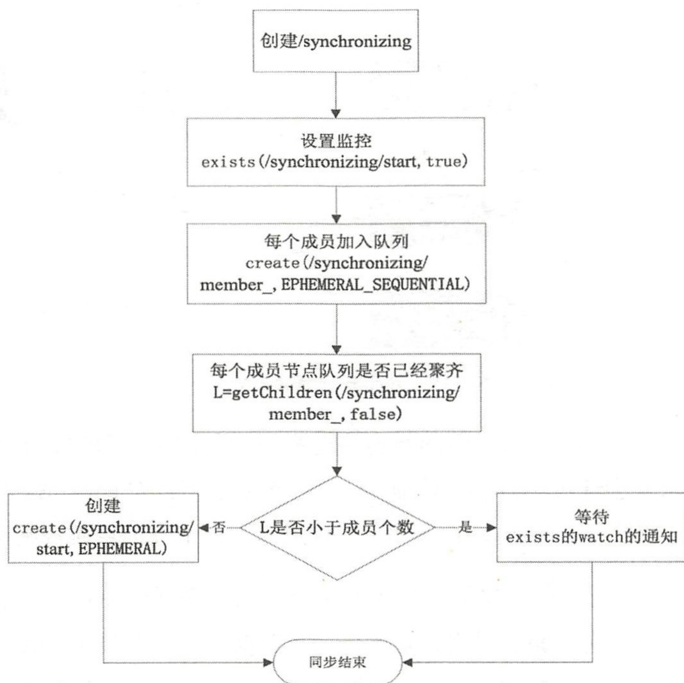


图 1.17 同步队列流程图

用 ZooKeeper 实现 FIFO 队列的思路如下。

在特定的目录下创建 SEQUENTIAL 类型的子目录/queue_i，这样就能保证所有成员加入队列时都是有编号的；出队列时通过 getChildren( )方法返回当前所有队列中的元素，再消费其中最小的一个，这样就能保证 FIFO。

1.10 分布式消息通道服务的设计

分布式消息通道广泛应用在很多公司，尤其是在移动 App 和服务端需要上传、推送大量的数据和消息时。比如打车 App 每天要上传大量的位置信息，服务端也有很多订单要及时推送给司机；此外，由于司机是在高速移动过程中，所以网络连接的稳定性也不是很好——这类场景给消息通道的高可用设计带来很大的挑战。

如图 1.18 所示是一个典型的移动 App 的消息通道的设计架构图，这种设计比较适合上传数据量大，并且高速移动导致网络不太稳定的链路。

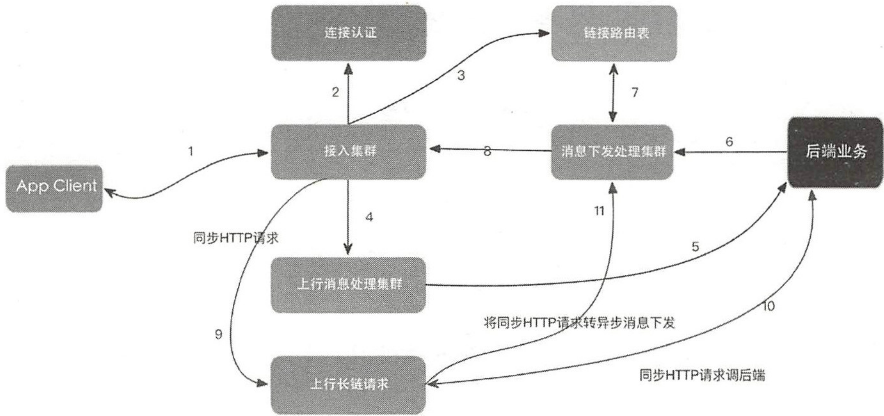


图 1.18　分布式消息通道设计

链路 1 是 Client 和整个服务端的长连接链路，一般采用私有协议的 TCP 请求。如果是第一次请求还会通过 2 做链接认证，认证通过后会把该 Client 和接入集群的某个服务器做个 K/V 对，并记录到路由表里——这可以方便下发消息时找到该链接。

经过链路 4，上行消息处理集群会将 TCP 请求转成普通的 HTTP 请求，再调用后端业务执行具体的业务逻辑，或者只是上传一个数据而已，不做任何响应。如果业务有数据需要下发，会经过链路 6，把消息推送到消息下发处理集群，由它把消息推送给 Client。

消息下发集群会查询链接路由表，确定当前 Client 的链接在哪台机器上，再通过该服务器把消息推送下去。这里常见的问题是当前 Client 的网络不可达，导致消息无法推送。在这种情况下，消息下发处理集群会保持该消息，并定时尝试再推送；如果 Client 重新建立连接，连接的服务器也会随之变化，那么消息下发集群会去查询链接路由表再重新连接新的 K/V 对。

链路 9 是为了处理 Client 端的一些同步请求而设计的。例如 Client 需要发送一个 HTTP 请求并且期望能返回结果，这时 Client 中的业务层可能直接请求 HTTP，再经过 Client 中的网络模块转成私有 TCP 协议，在上行长链请求集群转成 HTTP 请求，调用后端业务并将 HTTP 的 response 转成消息发送到消息下发处理集群，异步下发给 Client，到达 Client 再转成业务的 HTTP response。这种设计的主要考虑是当 HTTP 响应返回时，如果长链已经断掉，该响应就没法再推送回去。因此，这种上行同步请求

而下行异步推送是一种更高可用的设计。

从整体架构上看，只有接入集群是有状态的，其他集群都是无状态的，这也保证了集群的扩展性。如果接入点在全国有多个点，并且这些点与服务端有专线网络服务，接入集群还可以做到就近接入。

1.11 典型的分布式集群设计思路

当前的分布式集群管理中通常有两种设计思路：一种是 Maser/Slaver 模式（如图 1.19 所示）；一种是无对等的设计思路（如图 1.20 所示）。两种思路各有优缺点。

如图 1.19 所示是一个典型的 Maser/Slaver 模式的架构，它的优点是 Master 节点是固定集中式的，管理着所有其他节点，统一指挥、统一调度，所有信息的一致性都由它控制，不容易出错，是一种典型的集权式管理。它的缺点也很明显：一旦 Master 挂了，整个集群就容易崩溃；另外，由于它控制了所有的信息，所以也容易成为性能瓶颈。

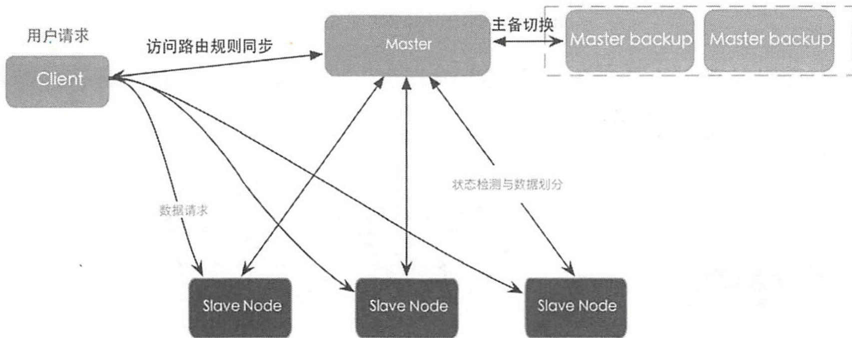


图 1.19 一种分布式集群设计

对等的集群管理模式中，最典型就是 Cassandra 的集群管理。它的特点是每个节点的功能都是一样的，所以每个节点都有能力成为 Master 节点。在一个集群中要实现这一点，要求整个集群中所有机器的状态都要保持一致，那么 Cassandra 是如何做到的呢？Cassandra 利用了 Gossip 协议（谣言协议）：一个节点状态发生变化很快被传播到机器中的所有节点，于是每个节点发生相应的变更。这种管理方式是通过节点之间充分的信息交换来实现管理的，因此信息的及时、充分交互是必要前提。但这会导致机器之间交互控制信息过多，并且集群越大信息越多，管理越复杂，在出现 bug 时不太容易排查。

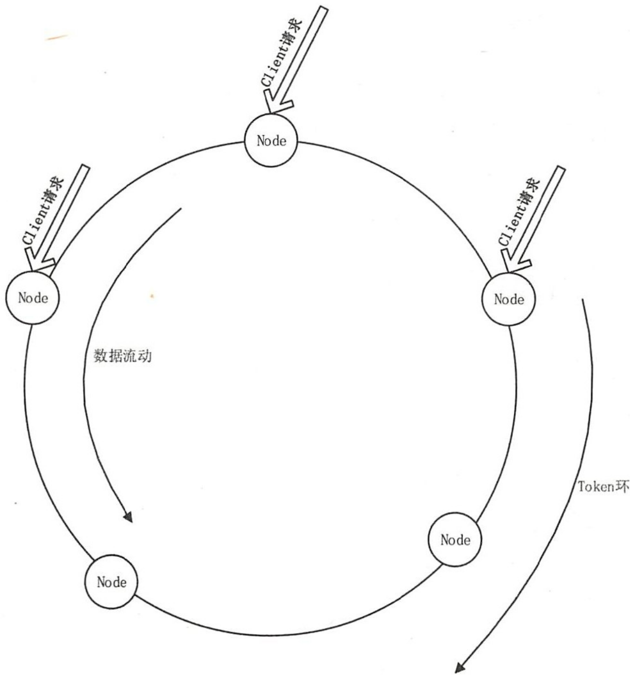


图 1.20　无主 Master 集群设计

下面我们以开源的 Tair 集群管理为例着重介绍 Maser/Slaver 的一种管理模式（如图 1.21 所示），它的设计比较巧妙。

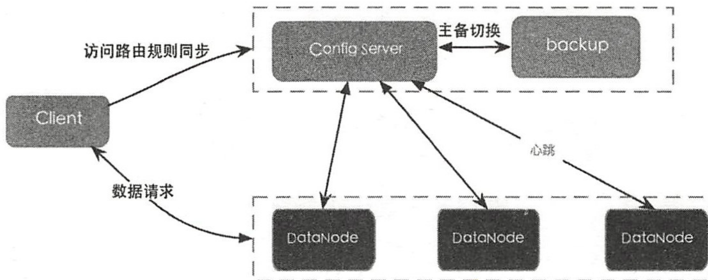


图 1.21　一种 Master/Slaver 集群设计

从集群的架构上可以看出通常有 3 个角色：Client、ConfigServer 和 DataNode，整个集群通过一个路由信息对照表来管理，如表 1-1 所示。

表 1-1　路由对照表

Bucket	Node
0	192.168.0.1

续表

Bucket	Node
1	192.168.0.2
2	192.168.0.1
3	192.168.0.2
4	192.168.0.1
5	192.168.0.2

Bucket 是 DataNode 上数据管理的基本单位，通过 Bucket 可以将用户的数据划分成若干个集合。表 1-1 中分成 6 个 Bucket，那么所有用户的数据可以对 6 取模，这样每条数据都会存储在其中的一个 Bucket 中，而每个 Bucket 也会对应一台 DataNode。只要控制这张列表就可以控制用户数据的分布。

ConfigServer 与 DataNode 保持心跳，并根据 DataNode 的状态生成对照表，Client 主动向 ConfigServer 请求最新的对照表并缓存。ConfigServer 最重要的责任就是维护对照表，但从实际的数据交互角度看，它并不是强依赖——正常的数据请求不需要和 ConfigServer 交互，即使 ConfigServer 挂掉也不会立即影响整个集群的工作。原因在于对照表在 Client 或者 DataNode 上都有备份，因此 ConfigServer 不是传统意义上的 Master 节点，也就不会成为集群的瓶颈。

下面介绍一下它们如何解决集群中的状态变更：扩容和容灾。

（1）扩容

假如要扩容一台机器 192.168.0.3，那么整个集群的对照表需要重新分配，而重新分配对照表必然也会伴随着数据在 DataNode 之间的移动。数据的重新分配必须基于两个原则：尽可能地保持现有的对照关系，均衡地分布到所有节点上。新的对照表如表 1-2 所示。

表 1-2　扩容的新对照表

Bucket	Node
0	192.168.0.1
1	192.168.0.2
2	192.168.0.1
3	192.168.0.2
4	192.168.0.3
5	192.168.0.3

此时只需将 4 和 5 Bucket 数据移动到新机器上。

（2）容灾

容灾模式比扩容更复杂一些，除了上面两个原则以外，还需要考虑数据的备份情况。假如保存了3份数据，则对照表如表1-3所示。

表1-3　容灾的新对照表

Bucket	Node	Node	Node
0	192.168.0.1	192.168.0.2	192.168.0.3
1	192.168.0.2	192.168.0.3	192.168.0.1
2	192.168.0.1	192.168.0.2	192.168.0.3
3	192.168.0.2	192.168.0.1	192.168.0.3
4	192.168.0.3	192.168.0.1	192.168.0.2
5	192.168.0.3	192.168.0.1	192.168.0.2

第一列的Node作为主节点，如果主节点挂掉，那么第二列的备份节点就会升级为主节点；如果备份节点挂掉则不会受影响，而是再重新分配一个备份节点以保证数据的备份数。

当DataNode节点发生故障时，ConfigServer要重新生成对照表，并把新的对照表同步给所有的DataNode。Client是如何获取最新对照表的呢？每份对照表都有一个版本号，每次Client向DataNode请求数据时，DataNode都会把自己对照表的版本号返回给Client，如果Client发现自己的版本低，则会从ConfigServer拉取最新的对照表。这种方式会产生一个问题：当对照表发生变更时，Client有可能会更新不及时导致请求失败。为何ConfigServer不把对照表主动推送给Client呢？当然可以，但这会导致ConfigServer保持对每个Client的心跳，加重ConfigServer的负担，尤其当Client数量非常大的时候，容易给ConfigServer造成管理瓶颈。因此，上面的设计其实是取中考虑，即在Client的数量和DataNode发生故障的概率之间选择一个。

综上，集群管理中最大的困难就是当DataNode数量发生变化、涉及的数据发生迁移时，既要保证数据的一致性，又要保证高可用。

1.12　总结

网站的分布式化改造是非常重要的一步，从刚开始的单应用单层级的结构转换成多应用多层级的分布式应用需要解决前面我们提到的众多问题，而最核心的是要解决以下问题。

第一，纵向业务逻辑的分层拆分，要方便不同工种的程序开发人员的协作，如前端和后端开发人员的协作效率，业务开发人员与偏技术的中间件开发人员的分工等。

第二，横向不同业务系统的拆分，如商品和会员系统独立，交易与支付系统的独立等。这种拆分更多是业务领域知识的专业化，同时也为系统的稳定性和扩展性提供更好的支持。一般而言，业务系统的拆分也会引起组织结构的变化。

第三，系统的横向和纵向的拆分必须首先解决好系统之间的连接问题，而这些系统之间如何连接就是分布式系统必须解决的问题。本章主要介绍了在这些环节的一些通用的解决办法。

最后我们用图 1.22、图 1.23 来总结单应用系统向分布式系统演进的过程。

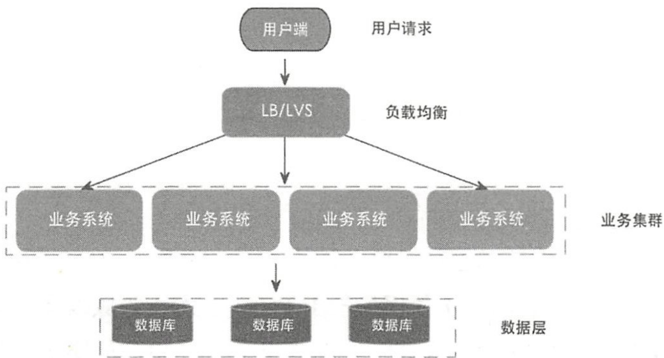


图 1.22　单应用集群架构

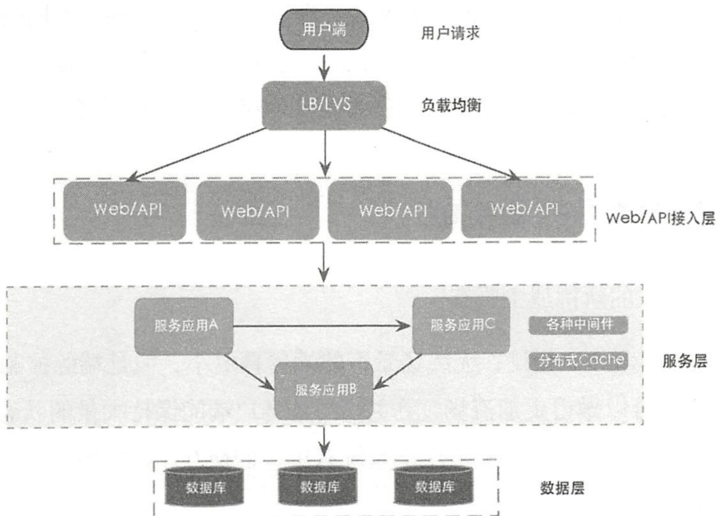


图 1.23　典型的分布式集群架构

2 无线化：无线时代下的架构演进

从 2012 年开始，无线技术蓬勃发展，各种终端（像 Pad、TV、各种 Console）兴起，人们接触互联网的途径已经不仅仅局限于 PC 了。无线领域以客户端体验为主，比 Web 拥有更好的交互和体验，这对传统的 Web 提出了更高的要求。在新形势下网站也需要适配各种不同的终端，这不仅仅是界面上的差异，而且在交互上也要根据不同终端的特点来优化体验。但是对后端来说，有可能 90%的功能在不同的终端均是一样的，那么应该如何解决服务端的架构：是用一套代码一个系统支撑多个终端，还是使每个终端对应独立的服务端系统？这些都是当时必须回答的问题。

2.1 无线环境下的新挑战

无线环境带来的新挑战主要有：

- 端的问题。和 PC 相比，无线场景下的端屏幕更小，但是端的控制力更强，端上的交互可以做得更加流畅，更主要的是客户端能保持大量的状态数据，减少与后台的交互；客户端能对底层系统有更多的交互，如本地缓存；客户端的网络不稳定，但是网络调用会更加可控。
- 服务端的问题。由于同一个业务需要暴露给多个终端，那么我们要考虑如何做

到一个业务逻辑能够多端复用，前端的交互逻辑如何复用，前端是否需要 MVC 化？

- 多端登录和多屏互动的问题。

早期无线化时，都是针对无线终端单独搭建一套系统：从网关一直到后端的系统，如图 2.1 所示。

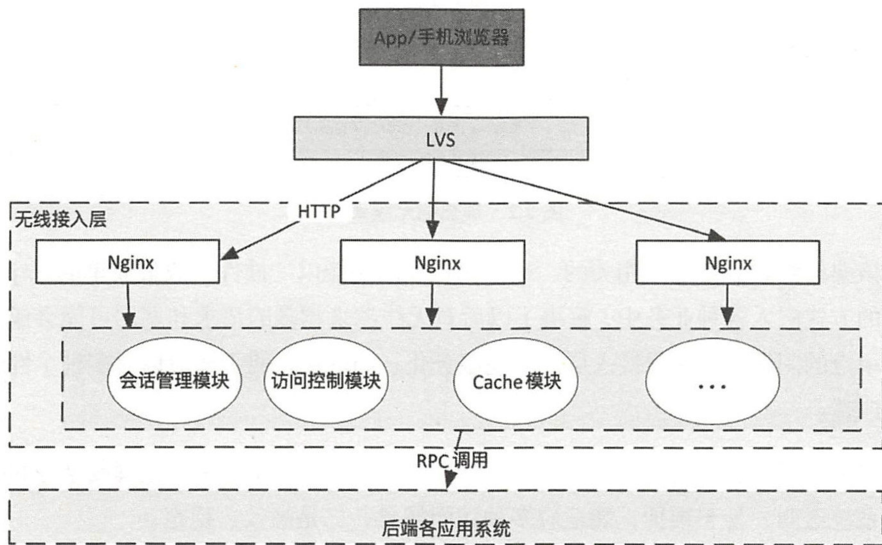


图 2.1　早期的无线架构

这种系统最明显的特征就是会有一个统一的接入网关层，这一层会解决一些通用问题，像下面这些典型场景会在该统一接入层上解决：统一 HTTP 接入、登录会话管理、安全访问控制、Sign 安全验证、参数的合法性检查、热数据的 Cache、协议转化、API 的注册及路由、API 的生命周期管理和监控平台等。

不过，虽然在发展初期这个方式是合理高效的，到了后期，我们越来越发现其承载的功能较多，当每个系统都必须做数据输出后，统一接入层作为单点的问题就会比较突出，因此这时有必要考虑弱化它的功能，使它仅作为统一接入的 URL 转发，并保留一些稳定的、和业务无关的功能；与业务相关的功能则全部迁移到各自的系统中去，由各业务系统单独维护。

所以，这些管理功能可以通过“旁路依赖”的方式进行，理想的调用链路如图 2.2 所示。

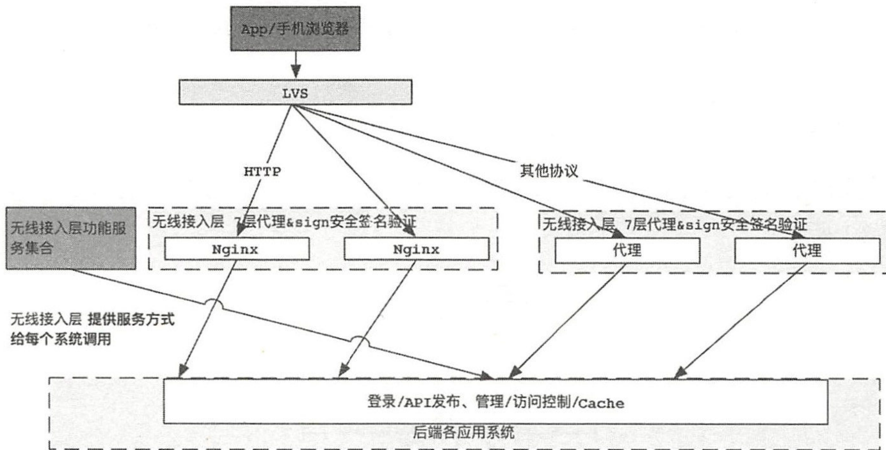


图 2.2　理想的无线架构

该架构将和业务比较相关的、或者比较重的功能以“插件”或最简单的以 jar 包依赖的方式嵌入各种业务中，解决了以后对无线越来越多的请求和调用可能会成为瓶颈和单点的问题，使无线接入层的功能服务化、插件化，便于适应以后多种个性化的架构改造。

综上，理想的无线接入层要有以下特性：一是尽量简单，不要处理复杂逻辑；对业务系统透明、易于维护，能适应多种访问场景；二是高效、稳定。

2.2　端的演进

客户端主要有两种选择：一种是基于浏览器 HTML5 页面的，一种是 Native 模式的。到底是选择 HTML5 还是 Native，Native 如何解决快速迭代问题？

1．是 Native 还是 HTML5

当前移动端主要还是以 Native 实现为主，从用户体验角度来考虑，Native 的实现要比 HTML5 更流畅，同时 Native 还可以基于本地做很多在浏览器里不能做的优化，如大数据的存储、可以定制的通信协议、更方便地保持长连接以及更容易实现的实时消息推送。

当然 HTML5 也有无法比拟的优势，比如客户端更轻量级、服务端发布更迅速、不需要用户升级版本等。长期来看，移动端是否会像早期 PC 那样从富客户端转向浏览器呢？笔者觉得未必，理由如下。

首先，相比 HTML5，Native 实现性能优势更好。当前移动端都在追求极致体验，App 无疑会比 HTML5 有更多的优势；其次，移动端屏幕较小，基于网页的交互和 App 相比还有很多限制。最重要的是，不同的商家会主推带有品牌标识的 App 还是会向统一的浏览器靠拢？从目前的趋势看，App 会是手机端上争夺的重点，所以笔者推测直接基于手机端的浏览器的应用不会成为主流的前端。

2. HTML5 的页面优化

HTML5 页面优化一般可以从以下几个地方入手。

第一，CSS 内联异步加载。如果页面中有内容要依赖 CSS 的加载，很多时候就会出现白屏——这其实就是 CSS 阻塞了加载，CSS 出不来就导致看不到首屏。CSS 内联加载可以节省异步 HTTP 请求，CSS 内联异步加载后可以大大缓解白屏问题。不过，就算内联以后也要观察异步 CSS 文件的大小，并且异步之后要观察 domReady 的时间变化。当然 CSS 内联也有可能会导致 repaint 和 reflow 的问题，并且由于异步内容增大，服务端的性能开销也会增加。

第二，其他的优化。端上的优化已经有一整套的优化方法列表了，这里介绍一些我们在实践中发现并验证过的一些特别的优化点，如 assets 合并、整合页面中 inline 的 JS\CSS 到外部文件、将 iframe 改为 JSONP 调用、背景图合并和将非首屏内容加载改为异步等。

第三，bigpipe 首屏加载。2012 年的时候，Facebook 有一个比较火的技术叫 bigpipe，可以提升页面的首屏加载效果，于是我们尝试过采用类似的技术测试首屏的加载效果，点击链接 http://www.webpagetest.org/video/compare.php?tests=140318_M5_7GV%2C140318_Z2_7CJ&thumbSize=200&ival=100&end=full，可以通过 webpagetest 看到页面的优化效果。

3. Cookie 压缩

在无线场景下要额外注意 Cookie，如果没有留意，它可能会占用你一次无线请求下的大部分内容，而且有可能并不会让你察觉，所以有必要对 Cookie 进行压缩测试。

Cookie 是在 HTTP 的头部，通常的 gzip 和 deflate 都是针对 HTTP body 的压缩但并不能压缩 Cookie，要想对 Cookie 做压缩测试必须单独处理，压缩方式是将 Cookie 的多个 K/V 对看成普通的文本，进行文本压缩。

（1）Cookie 的全部压缩

将所有 Cookie 看成一个文本字符串，对其进行压缩编码。举例如下。

- 原 Cookie：a=123;b=234;c=356;i=124
- 压缩后 Cookie：tzip=xxxx

（2）Cookie 的部分压缩

仅仅对部分 Cookie 项进行压缩，仅压缩 b、c、i 选项，举例如下。

- 原 Cookie：a=123;b=234;c=356;i=124
- 压缩后 Cookie：a=123;tzip=xxxx

压缩算法同样可以使用 gzip 和 deflate 算法，但是需要注意的一点是根据 Cookie 的规范，Cookie 中不能包含控制字符，仅仅只能包含 ASCII 码为 34-126 的可见字符，所以要将压缩后的结果再进行转码，可以用 Base32 或者 Base64 编码，如表 2-1 所示。

表 2-1　几种压缩对比

处理方式	最终大小(byte)	压缩比率
Defalter	1189	42%
Defalter + Base32	1904	7.2%
Defalter + Base64	1588	22.65 %
Defalter + Ascii85	1489	27.5%

可以配置一个 Filter，在页面输出时对 Cookie 进行全部或者部分压缩，代码如下。

```
    private void compressCookie(Cookie c, HttpServletResponse res) {
            try {
                ByteArrayOutputStream bos = null;
                bos = new ByteArrayOutputStream();
                DeflaterOutputStream dos = new DeflaterOutputStream
(bos);
                dos.write(c.getValue().getBytes());
                dos.close();
                System.out.println("before compress length:" +
c.getValue().getBytes().length);
                String compress = new sun.misc.BASE64Encoder().
encode(bos.toByteArray());
                res.addCookie(new Cookie("compress", compress));
                System.out.println("after compress length:" + compress.
getBytes().length);
            } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
```

上面这段代码是用 DeflaterOutputStream 对 Cookie 进行压缩，等 Deflater 压缩后再进行 Base64 编码，相应的可以用 InflaterInputStream 解压，代码如下。

```
    private void unCompressCookie(Cookie c) {
        try {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            byte[]  compress  =  new  sun.misc.  BASE64Decoder().
decodeBuffer(new String(c.getValue().getBytes()));
            ByteArrayInputStream  bis  =  new  ByteArrayInputStream
(compress);
            InflaterInputStream inflater = new InflaterInputStream
(bis);
            byte[] b = new byte[1024];
            int count;
            while ((count = inflater.read(b)) >= 0) {
                out.write(b, 0, count);
            }
            inflater.close();
            System.out.println(out.toByteArray());;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

2KB 大小的 Cookie 压缩前与压缩后的字节数相差 20%左右，如果网站的 Cookie 在 2~3KB 左右，而某天有 1 亿的 PV 的话，那么一天就能够产生 4TB 的带宽流量，因此，从节省带宽成本来看，压缩 Cookie 还是很有必要的。

4．URL 短域名

URL 短域名也很好理解，如果无线数据传输中有大量的域名，而域名又比较长，就会产生很多无谓的数据传输，最典型的应用像微博的 http://t.cn，可以节省很多字节。但是像这种直接使用真实的 t.cn 的短域名是比较奢华的办法，比较简单的是使用约定的标签替换，在解析时再替换回去。

5．CDN 前置缓存

在有大量静态数据请求的页面中使用 CDN 前置缓存对网站的加速访问非常有效。图 2.3 对比分析了杭州主站和 CDN 上的两张图片，一张是空图片，一张是 50KB 大小的图片。空图片用于测试 RTT，50KB 的图片用于测试网速。分析结果如图 2.3 所示。

图 2.3　CDN 前置缓存对比

总体上看 CDN 的加载速度比杭州主站快 50%。

6. 如何实现端的快速迭代

前面介绍了无线场景下端的优化措施，那么当我们使用 Native 来实现时，遇到的一个问题是基于 App 的 Native 如何解决客户端更新和服务端的快速迭代问题，一般有两种思路：一种是客户端用同一种技术开发，然后通过工具编译技术把它编译成不同平台上能够执行的代码，如当前的 React Native；另一种思路是将客户端中经常需要更新的模块做成动态推送的，用模板+数据的方式，在不同的客户端平台上实现一个小的解析引擎来实现快速个性化的定制。

那么再说回来，基于前面的这些推断，多终端和服务端交互主要是以数据+模板的方式为主，那么服务端提供格式化的数据将成为必然选项。所以涉及的问题就是服务端既要提供格式化的数据（HTTP JOSN 数据），又要支持传统的 PC 的方式：基于 JOSN 数据渲染出 HTML 页面。我们在后面会进一步介绍如何解决无线和传统 PC 之间的这种差异。

2.3　无线链路的优化

服务端响应时间只占整个请求路径上很小的一部分，PC 上更重要的是优化首屏的加载，无线端更多则是优化中间的管道。

1. 无线端请求合并

无线环境下做请求合并的收益是比较大的，所以会将当前的两次请求在服务端做

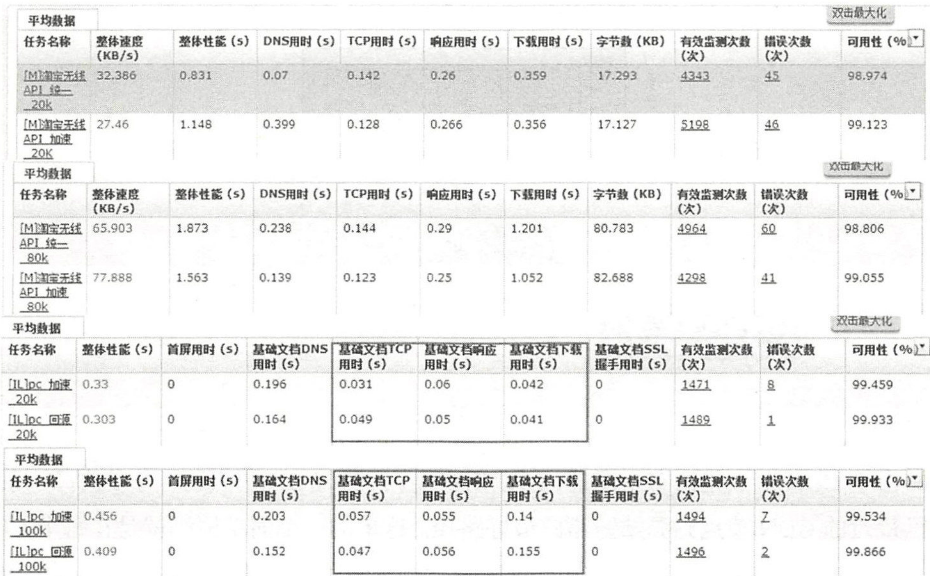
ESI 合并为一个请求。从图 2.4 可以看出，在无线环境下减少请求的数量可以明显减少总耗时。

任务名称	整体性能（s）	投影DNS用时（s）	投影TCP用时（s）	投影响应用时（s）	投影下载用时（s）	有效监测次数（次）	错误次数（次）	可用性（%）	元素个数（个）
[M]淘宝无线API_两次加速_全部	1.89	0.382	0.302	0.63	0.077	4521	15	99.669	2
[M]淘宝无线API_加速_一次源站	1.486	0.459	0.244	0.349	0.036	5670	18	99.684	1
[M]动态加速_全部节点	1.492	0.476	0.215	0.333	0.057	5669	23	99.596	1

图 2.4　合并无线请求

2．数据量大小的影响

无线环境下数据大小对性能的影响比在 PC 端的更明显，PC 端数据从 20KB 到 80KB 增加了 100 毫秒，而无线端数据从 20KB 到 80KB 增加了 700 毫秒，如图 2.5 所示。因此是否能控制页面大小对无线端的性能影响很大。



平均数据　双击最大化

任务名称	整体速度（KB/s）	整体性能（s）	DNS用时（s）	TCP用时（s）	响应用时（s）	下载用时（s）	字节数（KB）	有效监测次数（次）	错误次数（次）	可用性（%）
[M]淘宝无线API_统一_20k	32.386	0.831	0.07	0.142	0.26	0.359	17.293	4343	45	98.974
[M]淘宝无线API_加速_20K	27.46	1.148	0.399	0.128	0.266	0.356	17.127	5198	46	99.123

平均数据　双击最大化

任务名称	整体速度（KB/s）	整体性能（s）	DNS用时（s）	TCP用时（s）	响应用时（s）	下载用时（s）	字节数（KB）	有效监测次数（次）	错误次数（次）	可用性（%）
[M]淘宝无线API_统一_80k	65.903	1.873	0.238	0.144	0.29	1.201	80.783	4964	60	98.806
[M]淘宝无线API_加速_80k	77.888	1.563	0.139	0.123	0.25	1.052	82.688	4298	41	99.055

平均数据　双击最大化

任务名称	整体性能（s）	首屏用时（s）	基础文档DNS用时（s）	基础文档TCP用时（s）	基础文档响应用时（s）	基础文档下载用时（s）	基础文档SSL握手用时（s）	有效监测次数（次）	错误次数（次）	可用性（%）
[IL]pc_加速_20k	0.33	0	0.196	0.031	0.06	0.042	0	1471	8	99.459
[IL]pc_回源_20k	0.303	0	0.164	0.049	0.05	0.041	0	1489	1	99.933

平均数据

任务名称	整体性能（s）	首屏用时（s）	基础文档DNS用时（s）	基础文档TCP用时（s）	基础文档响应用时（s）	基础文档下载用时（s）	基础文档SSL握手用时（s）	有效监测次数（次）	错误次数（次）	可用性（%）
[IL]pc_加速_100k	0.456	0	0.203	0.057	0.055	0.14	0	1494	7	99.534
[IL]pc_回源_100k	0.409	0	0.152	0.047	0.056	0.155	0	1496	2	99.866

图 2.5　无线环境下数据量大小的影响

综上，我们得出如下结论。

- 在无线环境下，减少网络请求次数对首屏加载性能有比较明显的影响；
- 无线环境下的文件大小与 PC 环境下的文件大小对性能的影响效果不同：无线环境下的数据大小对性能影响比在 PC 环境下的更明显，所以是否能控制页面大小对无线环境下的性能影响很大；
- CDN 直接 Cache 可以大幅提升性能，所以尽量将数据 Cache 到 CDN，这对无线端同样是有效的；
- 小数据情况下，动态加速和直接回主站没有明显优势，再加上当前动态加速链路还在调优中，所以当前无线数据直接回统一 Cache 比较理想，待动态加速更加成熟后再走 CDN。

3. CDN 动态加速

CDN 的动态加速技术也是比较流行的一种优化技术，它的技术原理就是在 CDN 的 DNS 解析中通过动态的链路探测来寻找回源最好的一条路径，然后通过 DNS 的调度将所有请求调度到选定的这条路径上回源，从而提高用户访问的效率，如图 2.6 所示。

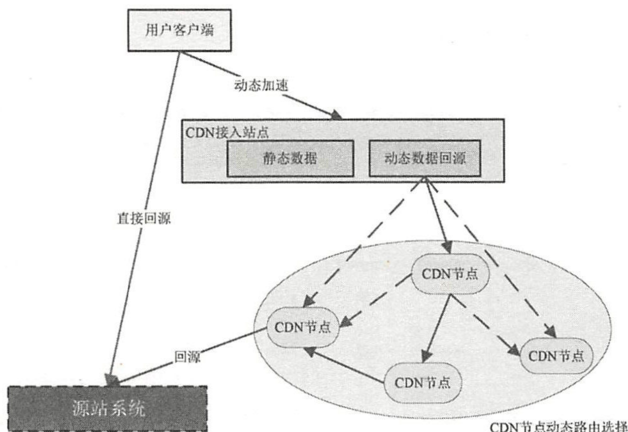


图 2.6　CDN 动态加速

由于 CDN 节点是遍布全国的，所以用户接入一个 CDN 节点后，可以选择一条从自己最近的 CDN 节点到源站链路的最好路径。这里的一个简单原则就是在每个 CDN 节点上，从源站下载一个一定大小的文件，看哪条链路的总耗时最短。这样可以构成

一个链路列表，然后绑定到DNS解析上、更新到CDN的Local DNS。当然，是否走这条链路并不一定只依据“耗时”这个唯一条件，有时候也要考虑网络成本，例如走某个节点虽然可以节省 10 毫秒时间，但是网络带宽成本却增加很多，还要综合考虑网络链路安全等其他因素。

4．WebP 图片优化

WebP是一种同时提供了有损压缩与无损压缩的图片文件格式，派生自图像编码格式VP8。WebP最初在2010年发布，目标是缩小文件并达到和JPEG格式相同的图片质量，以便节省图片文件的网络传送时间。

Chrome 浏览器以及 Chromium 内核的浏览器都支持 WebP 格式的图片，Android 4.0以上也都原生支持该类型图片。根据caniuse.com的统计，目前约有70%的用户可以享受到WebP格式带来的好处。

（1）WebP带来的收益

节约的带宽=CDN流量×可以使用WebP格式图片的流量占比×支持WebP的浏览器占比×WebP格式压缩率（实测为38%~41%）。从公式可知，带宽收益与CDN流量、支持WebP的浏览器（主要是Chrome内核浏览器）占比、页面中可以使用WebP格式的图片占比成正比。

我们以商品详情系统的放大镜图片（40×40 / 310×310）和所有装修中的图片使用WebP 为例，计算出详情系统使用 WebP 后，一个月大约会节约数十万元左右的带宽费用，非常可观。

（2）最佳实践

WebP有m0~m6等7种压缩级别，一般来说级别越高画质越好。但级别越高转换为WebP所要消耗的CPU资源也更多。在用WebP转换图片时要注意以下问题。

① 转换为WebP图片的时候，使用有损压缩（与m0~m6压缩级别无关）会导致：

- 红色字体被压缩之后普遍偏暗；
- 部分蓝色字体被压缩之后偏模糊；
- 当背景为黑色的时候，红色小字体偏模糊；
- 当背景为红色的时候，黑色小字体偏模糊。

② 低版本的Chrome浏览器不支持透明通道的WebP，所以对PNG图片不能使用

WebP 格式。

③ gif 转换为 WebP 动画非常耗性能，所以只处理第一帧图片。

④ 原图转换成 WebP 非常耗性能，图片的所有缩略图（如 310×310）都是先转换为 JPEG 格式，再转换为 WebP 格式。

通过分析近万张图片我们得出以下结论：压缩级别在 m0~m2 时，不少图片丢失色块（出现马赛克），压缩级别为 m3 及以上时则极少出现此问题，同时 m3 级别所节约的带宽和 m4 级别是非常接近的，但 m3 的转化性能明显高于 m4。因此，我们决定使用 m3 级别。

2.4 服务端的演进

在无线场景下服务端首先要解决多端适配的问题，虽然在无线业务发展的初期，无线端对应的后端系统大部分都是和 PC 独立的，但是这种场景不可能长久：一方面无线端和 PC 端的逻辑大部分是重合的，不断重复开发不合理；另外一方面，端的种类越来越多，不可能每个终端都单独搞一套系统，所以最后必然会用一套代码一套系统同时支持多端业务。

1. JSON 化接口

在 PC 时代服务端的架构通常如图 2.7 所示。

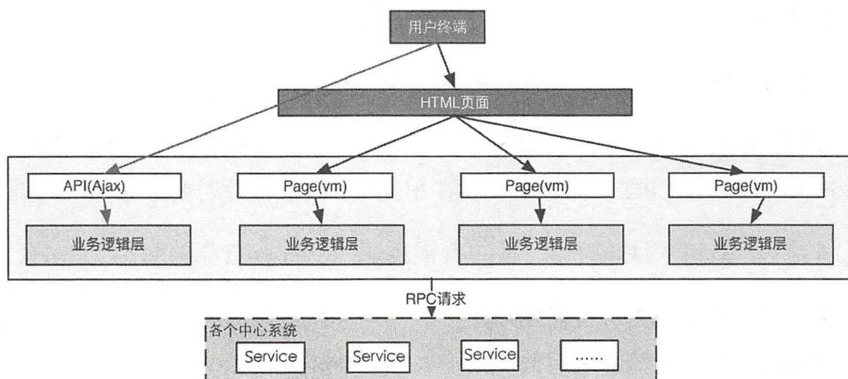


图 2.7 PC 时代服务端的架构

可以看出，PC 时代的架构基本是以页面为单位输出业务，辅以 Ajax 或 JSONP

来异步请求细粒度业务数据，由于大部分的渲染由后端模板直接渲染输出 HTML，会有很多页面逻辑直接以模板语言的方式被固化在模板中，也有很多页面逻辑被写在 MVC 中的 V 层中，导致前端的 HTML 是很薄的一层展现层，也很难做一些复杂的交互，并且很难做复用。特别是在页面适配多终端的情况下，会出现为多个终端编写各自的页面逻辑甚至业务逻辑的问题，导致系统整体不可控。

针对前端的多样性展示特点，我们把展示逻辑和数据对象解耦，把后端的数据 JSON 化，重新整理我们的架构，如图 2.8 所示。

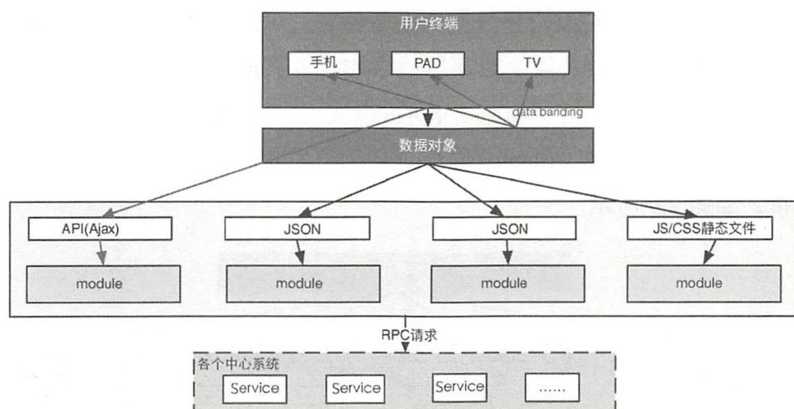


图 2.8 JSON 化数据接口

现在的架构和之前的相比，最大的区别就是弱化了后端的 MVC 架构，前端和后端尽量只做数据交换，做到前后端分离。一旦前后端针对接口做了分离之后，那么像页面改版等工作只需要前端就可以完成，而后端也能专心完成核心业务逻辑的处理。

此时需要解决的问题就是前端逻辑要做到前端业务逻辑和前端展示逻辑的分离，原因在于针对不同的多终端需求，要输出不同的 DOM 结构，有时候没有办法做到全平台的响应式设计，那么这时就需要为不同的终端输出不同的展示和交互逻辑。

2. 业务层组件化

前面的图 2.8 介绍了服务层需要做 module 化，其实业务层组件化在无线化改造过程中起的作用非常大，我们尝试过两种方案，分述如下。

第一，基于适配器的模式。这是偏浏览型的系统，即后面的业务逻辑并不复杂，主要是页面的展示比较多样，更多的是页面中展示模块的多种组合，最典型的就是商品详情页面。

仅供非商业用途或交流学习使用

第二，基于 SPI 的模式。这种更多是偏流程型的系统，业务逻辑比较复杂，流程编排也较多，需要用流程引擎做定制，这种系统的典型例子是交易系统。

基于 SPI 的组件化模式我们将在中台一章介绍，这里主要介绍页面的组件化设计思路（基于适配器的模式）。

以商品详情页为例，详情数据一般包括商品本身的信息（标题、类目、描述信息）、优惠信息、卖家信息、评价、商品服务、交易信息等内容，这些信息一般都通过 Spring Bean 的方式注入，这一层可以称为领域数据 Provider 层；业务逻辑层需要对这些不同的模块数据进行一定的规制校验和数据的重新封装，可以称为领域模块 Module 层；再到最上层，每个页面终端还需要对数据的格式或者数据类型做些处理，例如过滤一些字段等，这一层主要是针对页面需要展示的数据做些适配，所以这一层叫领域数据 Adaptor 层，它的最大特点是 Adaptor 能够支持动态发布和加载，所以这一层经常会有修改和发布，如图 2.9 所示。

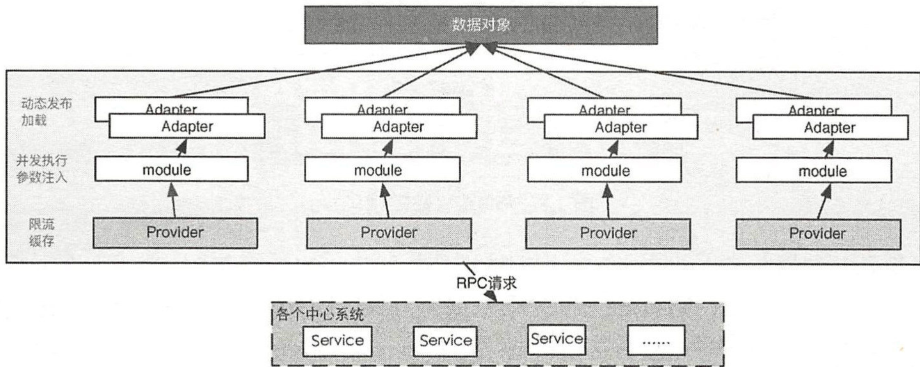


图 2.9　组件化服务端

3. MC 与 V 的分离演进

随着无线端的快速普及，前后端分离技术走上前台。Node 由于它的一些特性被工程师（尤其是前端工程师）快速接受。Node 火起来的另一个原因是无线技术的流行——很多应用要从传统的 PC 端开发转移到无线多端的场景——在这种情况下，渲染层和逻辑层的分离变得更重要，而 Node 刚好可以很好地渲染前端页面，所以它越来越受重视，也的确给我们的开发带来了很多好处。

Node 的主要特性有二：一是基于事件驱动，二是无阻塞 I/O，还有其他一些优点如单线程等。总体来说，Node 是为轻量级、分布式的实时数据服务这类应用（像微

仅供非商业用途或交流学习使用

博、Facebook 等典型场景，需要非常实时化、个性化、高并发的数据服务）提供运行容器而设计的。

（1）Node 带来新的思路

如前分析，由于无线终端的兴起，后端提供基于 JOSN API 的数据接口变得非常普遍，那么从无线和 PC 端的业务合并，一个系统提供多终端、多语言适配的角度来看，Node 能否在其中扮演传统服务端 MVC 中 V 的角色？

下面看看在实际开发环境中如何使用 Node，在引入 Node 之前有必要先介绍下当前传统的 Web 服务端架构，如图 2.10 所示。

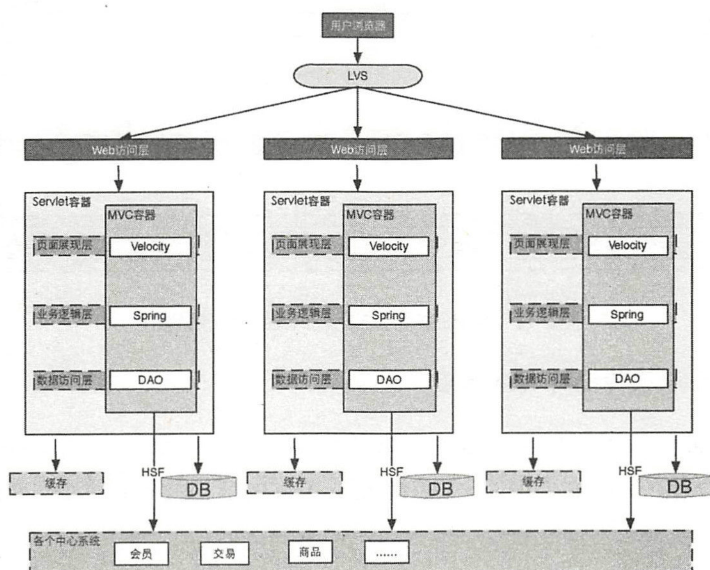


图 2.10　传统的 Web 服务端架构

（2）Node 作为单独的一层

在当前这种架构下如何融入 Node 呢？最保守的一种方案是将当前 Java Web 中的 View 层从 MVC 中独立出来，交给 Node 完成，Java Web 只提供基于 JSON 的数据接口供 Node 调用，图 2.10 中的架构变成了图 2.11 中的形式。

这种方式是在我们当前的访问路径上增加了一个 Web 代理层，所以也带来了一些问题，即这一层和当前的 Web 服务器层有点重复。那么 Node 能否取代前端的 Web 系统，成为主流的 MVC 框架呢？

仅供非商业用途或交流学习使用

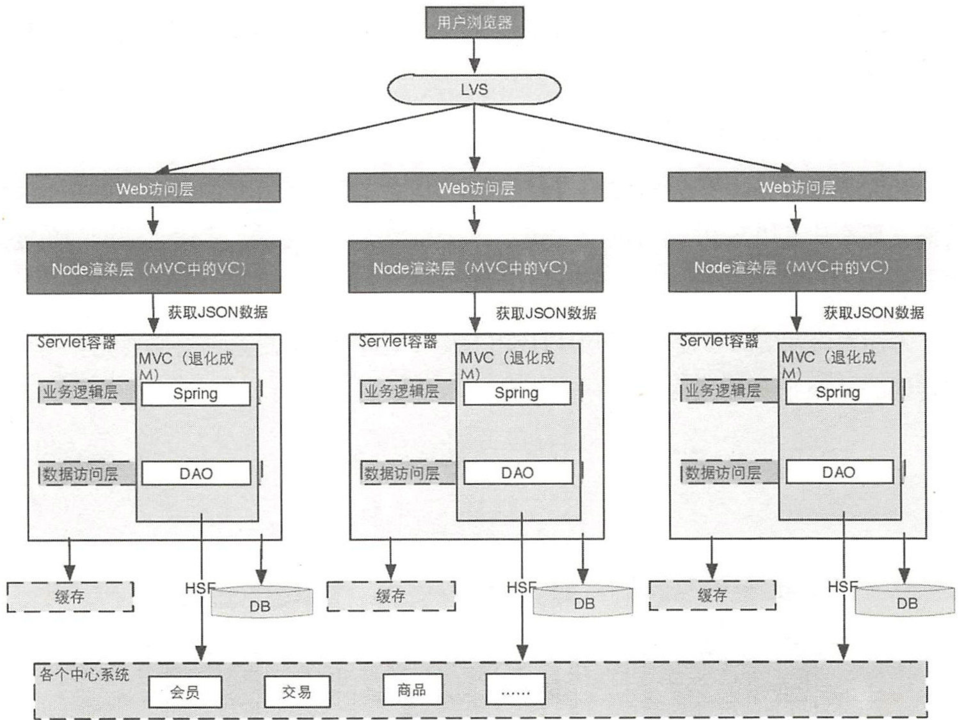


图 2.11　Node 作为渲染层加入到传统架构中

（3）Node 替换传统的 Web 框架

Node 和 Java Web 一样可以提供 MVC 管理功能，一个系统中同时存在两套 MVC 框架显然不合理，那么如果用 Node 替换 Java Web 的话，服务端的架构变成如图 2.12 所示的形式。

此时，我们用 Node 上的 MVC 框架如 express 替代 Java Web 中 Webx，也就是用 JavaScript 替换 Java。

（4）在 Java 中运行 Node

基于前面的分析，不管是在现有基础上单独增加一层 Node 还是整个替换 Java Web 层都不太合理，那是否意味着这种前后端分离的思路没有合理之处？是否会有更好的实现呢？

仅供非商业用途或交流学习使用

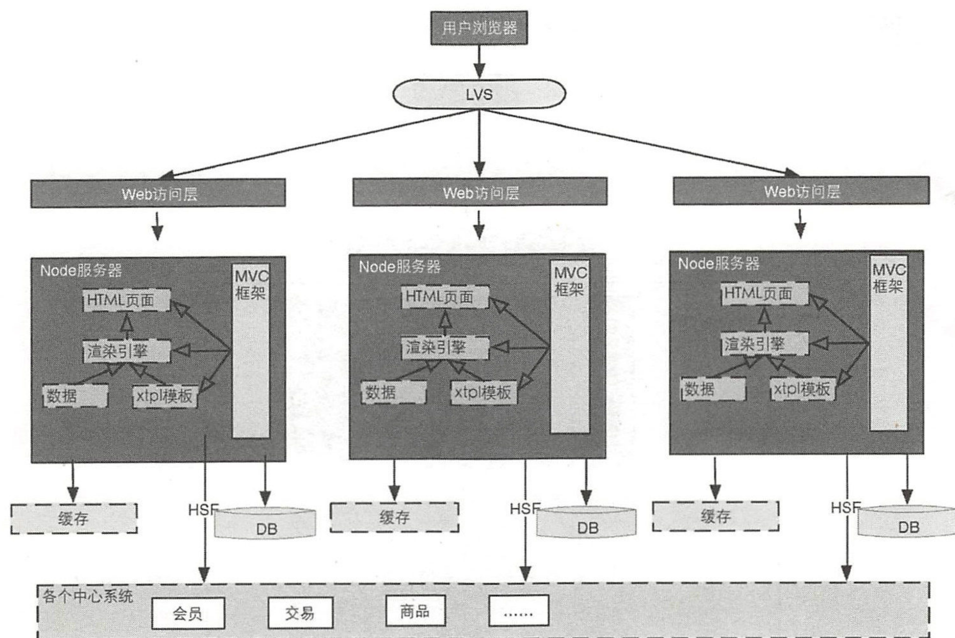


图 2.12 Node 替代 Java Web

传统的 MVC Web 软件架构将渲染层独立出来交由前端控制有其合理性，而在多终端的开发模式下，将业务逻辑层和前端渲染层分离有利于提升后端的开发效率，即后端只需要关注自己的业务逻辑和数据的输出（在 Native 开发环境下，服务端只需要输出 JSON 数据，客户端的页面渲染由客户端完成），HTML5 和 PC 端所需要的 HTML 渲染统一交给前端完成，这样操作有利于前端更好地开发模板。

按旧的思路，先要画好模板（HTML），交给后端转化成相应的模板（如 Velocity 或 JSP），再在复杂的 Java Web 工程下调试页面，而前端要独立运行整个 Java Web 工程还是相当困难的。如果把渲染层全部交给前端，那么前端只要和后端约定好数据，就可完成全部页面，减少了双方的交流成本（但这也有可能会把原本是后端的开发工作量转嫁给前端了）。在这种模式下，前端掌握了渲染层的控制权，开发体验会有大幅提升。

再说回来，我们一直讨论的基于 Node 实现的前端分离方案，可以把它分解成 Node 技术和前后端分离技术。很明显前后端分离在当前多终端背景下有其合理性，但是是否一定要用 Node 来实现呢？答案是不一定。当前还有两种方案可选。

仅供非商业用途或交流学习使用

一种将 Node 层代码抛到 Java 体系上，如图 2.13 所示。

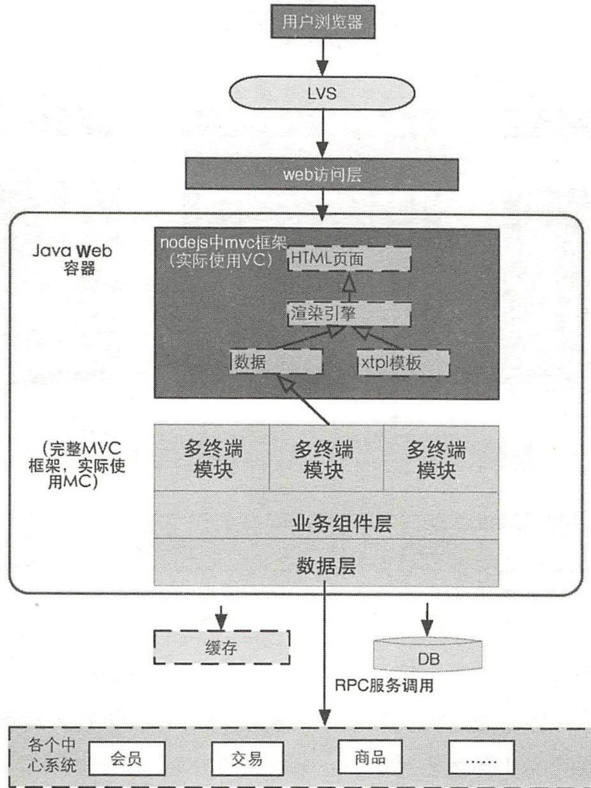


图 2.13　将 Node 嵌入到 Java 中

Java 8（Nashorn）中可以支持 JavaScript 的解析，而 Avatar.JS 可以将 Node 无缝迁移过来，但是经过测试，Nashorn+Avatar.JS 的执行效率太差，会导致 4~10 倍的性能下降，就算是最好的情况下，也会导致 50%的性能下降，这就很难达到工程级别应用的要求。

另外一种方案就是在 Java Web 体系下，将渲染层独立出来，渲染层和业务逻辑层仍然通过 JSON（或者大对象）交互数据，使得渲染层既可以在 Java 上渲染也可以在 Node 上执行，如图 2.14 所示。

这种方式与前一种的区别是只做渲染引擎的适配，即模板在 Node 和 Java 上都可以解析，而不是把 Node 的整个 MVC 都搬过来。由于 Node 和 Java Web 都有控制逻辑（即 MVC 中的 C），如果 Node 和 Java 中的逻辑不一致会导致两边渲染出来的 HTML 不一致，因此需要把 URL 改造成满足 RESTFULL 的格式，尽量把 C 的逻辑简单化。

仅供非商业用途或交流学习使用

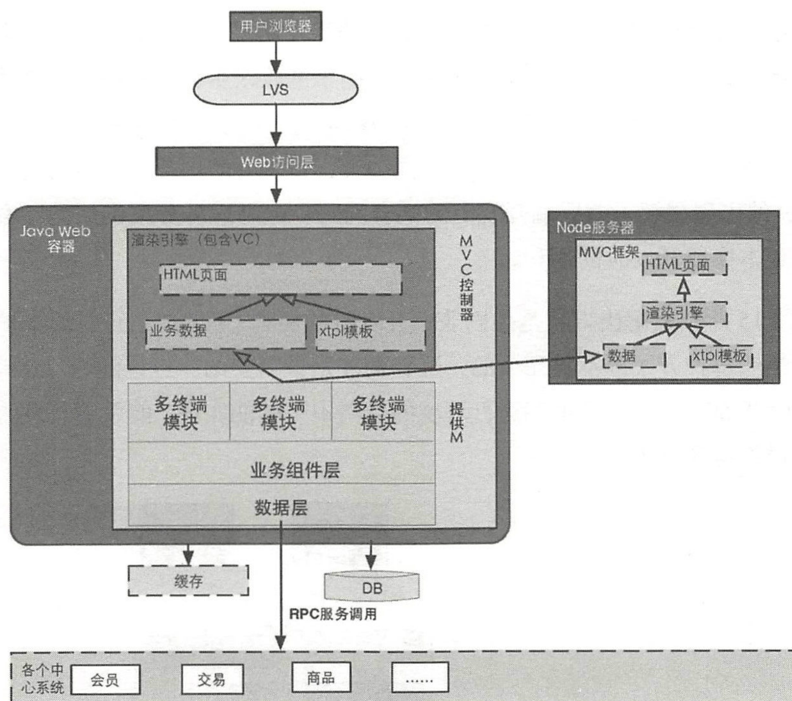


图 2.14　Java Web 兼容 Node 的模板层

图 2.14 的架构正是我们在详情系统上的实践，它成功的关键在于将 XTPL 的模板从 Node 上无缝迁移到了 Java 上，另外就是保持了页面路由的尽量简单，这样前端在 Node 上开发的重点只是 XTPL 模板。

采用这种解决方案可以达到以下几个目的。

- 后端系统完成组件化改造，PC 和无线逻辑统一起来了；
- 将渲染层独立，渲染层与业务辑层通过 JSON 数据对象交互；
- 前端开发同学完全掌握了 XTPL+JS 逻辑，有更多的掌控力；
- 前端开发页面不需要依赖后端的 Java 系统，调试页面可以在 Node 中完成；
- 并没有在系统的架构再增加一层，运维上也没有引入新的 Node 系统。

（5）最佳实践

随着技术的不断进步，开发模式也一直在发生变化。早期的页面渲染和业务逻辑全部集中在一起，如 ASP、PHP、JSP 技术，后来由于业务逻辑不断变得复杂，出现了 MVC 的开发框架，前后端工程师的分工也越发清楚。中间也曾有前端工程师负责

仅供非商业用途或交流学习使用

整个渲染层和控制层的实现的，如 Extjs+Ajax 的开发模式，但是由于整个渲染是在浏览器端完成的，受制于客户端渲染性能和搜索引擎的收录页面的硬缺陷，这种方式很难成为主流。一直到今天，前后端开发模式仍然是由后端工程师管理 M 和 C，由前端工程师来实现 V，开发环境和运行环境是同一套，开发上的耦合增加了沟通和调试成本。Node 的出现缓解了前后端开发上的耦合问题，但是这种分离仍然是以增加运行时的维护成本换取开发时的便利。

如图 2.15 所示的解决方案是想以兼顾开发时的便利、不增加运行时的维护成本为出发点，当然每种方案都不是完美的，适合的才是最重要的。也许随着 Java 中执行 JS 技术的不断成熟，开发环境和运行环境的分离不久就能实现，前后端开发的耦合度问题也就最终得到解决。

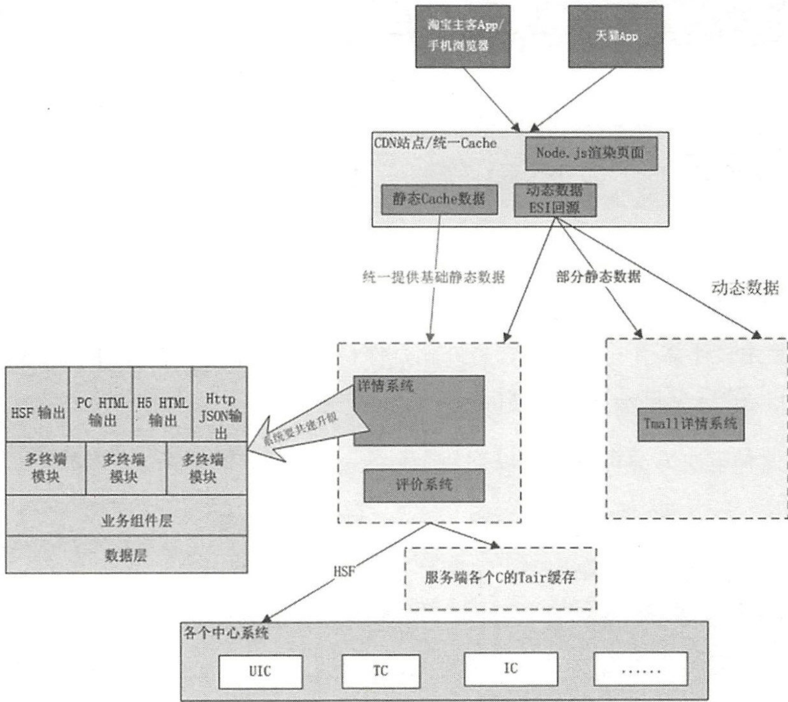


图 2.15　一个最佳实践示例

2.5 思考：开发语言选择的思考

前面讨论了如何将 Node 与现有的架构融合以面对新的无线场景，并更好地提升开发效率，我们主要阐述了在当前成熟的体系下向 Node 演进的一个思路，当然这并

仅供非商业用途或交流学习使用

不是唯一路径。也有很多程序员讨论是否要将后端的 Java 体系全部替换为 Node 体系，这就涉及 Java 技术栈和 Node 技术栈的比较，以下是笔者关于该问题的思考。

我们从语言特性、开发效率和成本因素三个方面比较 Java 与作为后来者的 Node。

1. 语言特性

JavaScript 作为 Node 上运行的语言，和 Java 相比，优缺点很明显。JavaScript 语法简单，很容易编写基于事件的驱动的实现，但是 JavaScript 基于面向对象的描述能力偏弱，不像 Java 是真正的面向对象语言，同时 JavaScript 对数据类型的定义也比较单一，要么是数值类型要么是字符类型。很明显，Java 更擅长构建复杂逻辑的大型应用程序。在语言运行效率上，JavaScript 原本是解释执行，Java 是编译执行，但由于 Node 做了优化，所以两者运行效率差别不大。

2. 开发效率

开发效率可以从语言的复杂度、程序员培养、开发工具包的丰富性以及编码效率几个方面比较。

- 语言的复杂度。从开发角度来看，Java 和 JavaScript 都不需要关心内存的管理，都是基于虚拟机来管理内存；从并发角度来看，JavaScript 是基于事件触发的，而 Java 是基于线程的，因此 JavaScript 更占优势；此外，JavaScript 是无阻塞 I/O 的，在 I/O 效率上比 Java 有优势（尽管 Java 8 也将更好地支持异步 I/O）。
- 程序员培养。目前 Java 语言仍然是仅次于 C 语言的第二大编程语言，而 JavaScript 排在第 10 位，Java 程序员队伍要比 JavaScript 大很多，很显然招聘 Java 程序员要比招聘 JavaScript 程序员更容易。
- 开发工具包。很多时候一个语言的开发效率要看这个语言的支持工具包和组件的丰富性，Java 经过这么多年的发展，工具类库已经非常丰富，几乎任何你想要的工具类库都能在网上找到。JavaScript 虽然也发展了很长时间，但是基于 JavaScript 的工具类库主要集中在前端，能够直接用于 Node 的仍然很少。当然 Node 的社区非常活跃，可以预见 Node 的工具类库增长也会非常迅速。但是要达到 Java 的规模尚需时日。
- 编码效率。Java 语言的运行基于 JVM，但是 Java 的部署效率稍差；JavaScript 使测试更加简单，但是 debug 机制仍然不完善。

3．成本因素

前面主要是从技术角度考虑，但是如果要从成熟的 Java 体系迁移到 Node，成本也是一个重要的考虑因素。

首先是学习成本。如果公司大部分是 Java 程序员，现在要迁往 Node，很明显这个学习成本会非常巨大，即使这个迁移是渐进式的，长期来看仍然是要将一部分 Java 程序员替换成 JavaScript 程序员。先不管程序员是公司内部培养的还是从外部招聘的，我们都可以算一下公司招聘一名程序员的成本有多大：一名普通工程师的年薪假定为 10 万元，猎头费一般是年薪的 20%以上，也就是 2 万元，再加上一个月的实习成本 1 万元，加在一起约 3 万元。这对于有 1 万名以上开发人员的大公司而言，人力成本可想而知。如果招聘应届生，由于应届生的培养周期更长，学习成本会更高。

其次是环境成本。公司的基础服务产品如中间件是基于 Java 开发的，如果要替换成 JavaScript，必然要再另外开发，还得开发配套的运维工具等，这个成本也可想而知。

最后是维护成本。Java 和 JavaScript 都是基于容器运行的，和 V8 引擎相比，程序员显然对 JVM 更熟悉。另外，从排查问题的难易程度来看，针对 JVM 的工具显然更完善。

4．人的因素

对于一家成熟的公司而言，假如现有的 Web 系统都改用 Node 实现，必然会有很多 Java 工程师要从事 Node 的开发，因为已有的前端工程师人数肯定支撑不了现有业务的发展。我们假定一部分 Java 工程师愿意学习 JavaScript 并成为全栈工程师，那么他们是否也愿意用两种不同的语言完成同一个任务呢？正常来说，如果能用同一个方式完成全部工作，那么把一个任务分成两种不同的方式来完成的必要性就会大打折扣。所以从这些角度来看，要让一家很成熟的公司切换语言栈是非常困难的。

尽管替换技术栈很困难，但是无论如何都应该统一技术栈，尤其是主流的业务开发，更应该使用统一的技术栈——这就像秦始皇统一语言一样——所带来的好处显而易见。

2.6 总结

总结一下，在传统的 PC 端互联网时代，标准的系统架构一般都如图 2.16 所示。

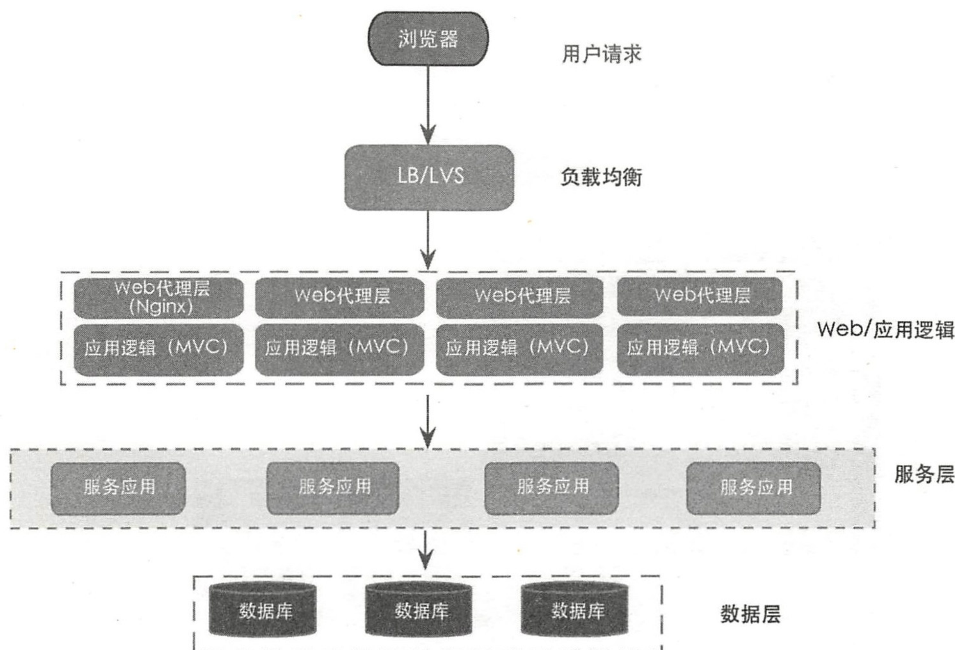


图 2.16　PC 端下的标准架构

在 Java 技术栈中，典型的架构是在整个业务逻辑层前面有一个 Nginx 代理层来接受和转发 Web 请求，将请求转发给 Tomcat 等 Java 应用服务器来处理。读者朋友要问了，像 Tomcat 本身也能直接处理 Web 请求为何还要用 Nginx？原因有二：一是 Nginx 更擅长处理大量连接请求，因为 Nginx 持有链接的成本要比 Java 低，这和语言执行效率也有一定关系；二是 Nginx 可以增加更多的功能模块例如限流保护、安全防护、gzip 压缩以及灵活的代理服务等，由于 Nginx 的执行效率更高，这些模块放在 Nginx 上要比放在 Tomcat 上更合适。

应用层中一般都是标准的 MVC 架构，MC 和 V 基本上逻辑分开但物理上是耦合的，然后应用层会通过 RPC 框架调用其他服务，从而构建一个完整的分布式系统。

而无线化时代最重要的变化是终端的变化，这导致 MVC 中的 V 发生明显变化：应用层不仅要输出 PC 下的 HTML 页面、还要输出纯结构化的数据，这会导致应用层出现分裂，在无线时代下，需要一套逻辑代码支持多终端的系统架构，如图 2.17 所示。

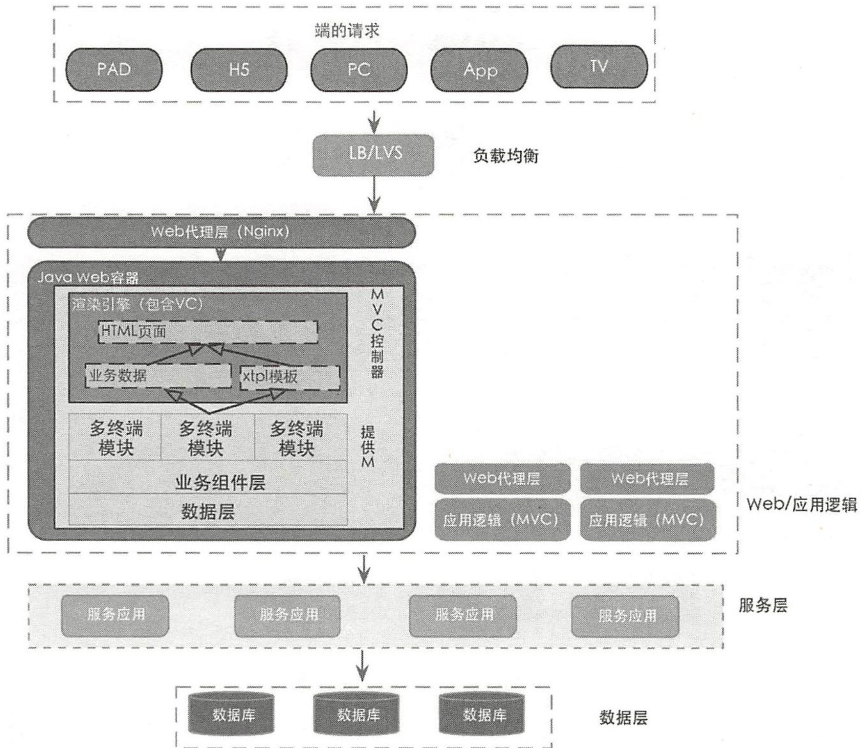


图 2.17　无线环境下一套代码支持多端架构示例

整体架构上变化并不大，最重要的是将 MVC 做更多的解耦，尤其是 M 和 V 分开，便于 V 实现更多个性化的渲染，既提供 HTML 的输出也支持纯数据（JSON 格式）的输出。

3 大型网站平台化演进：大中台小前台

中台这个概念早期是由美军的作战体系演化而来的，技术上所说的“中台”主要是指学习这种高效、灵活和强大的指挥作战体系。电商经过十几年的发展，组织已经庞大而复杂，业务不断细化拆分，也导致野蛮发展的系统越来越不可维护，开发和改造效率极低，也有很多新业务不得不重复造轮子，所以中台的目标是为了解决效率问题，同时降低创新成本。

本章内容参考了参考资料1的内容,同时结合笔者的一些观察和思考,仅供参考。

3.1 为什么需要中台

前面提及中台的出现,本质上是为了解决业务实现的效率问题,降低创新的成本,但是这类问题是一直存在的，为什么要在这个时点提出来呢，以前就没有效率问题吗?

1. 一些数据

电商的业务复杂度，可以通过一些数据来说明。举一个很明显的例子，2009年左右，当时商品详情系统的PV才1亿左右，到了2016年已经近50亿了，翻了50倍。

而当时的系统架构和现在的架构完全不可同日而语。

2. 系统规模的复杂度

一个复杂网站的系统架构一般会经历如下的演变过程。

（1）单系统。早期业务简单，几台机器就支撑了一个业务系统。刚开始系统不稳定，每天晚上都需要把系统重新启动一下。单系统阶段也可分成两段：最早是花 50 万元买的一个 PHP 系统；随着业务的发展，系统逐步改造成了 Java 技术体系，名字叫 Denali。

（2）分布式业务系统。到了 2007 年，团队已经有了上千人，项目支撑面临巨大的挑战，系统架构必须升级进化。这就开启了第二个阶段：分布式业务系统阶段。我们开始做分布式战略，把原来的单一系统拆分成多个高内聚、低耦合的中心化系统。现在大家耳熟能详的用户中心、商品中心、交易中心、店铺中心都是这个阶段出现的。这同时也意味着把上千人的团队拆分成了业务相对比较集中的小团队。每个独立的系统可以独立设计、独立接需求、独立发布，整个研发效率和系统稳定性都上了一个台阶。

（3）业务平台。电商发展的速度实在太快了，到了 2011 年，随着各种 B2C 网站、各种导购网站的出现，可能会把一个大型网站从组织架构上拆成几个独立的事业部。多个独立事业部的业务决策链路更短、业务发展更快，技术人员也快速增长。事业部的定位不一样、业务发展方向不一样、业务的管控规则不一样，甚至在一些业务规则上还可能相互冲突。

我们都知道，在做业务系统的时候，为了快速应对每天的业务需求变更，很多时候都是通过代码来写业务逻辑的，而在业务抽象建模，系统架构的开放性方面都很不成熟——这会导致业务逻辑之间的耦合和相互影响，大幅降低研发效率。系统架构必须升级，这就开启了第三个阶段：业务中心平台化阶段。什么是平台？就是要把基础能力和每个业务方的特性业务拆分，隔离业务和业务之间的逻辑。比如说两个相似的业务方业务有可能是冲突的，但他们需要在同一个平台上执行，这时我们必须把业务的逻辑分开。这个阶段开始升级会员平台、商品平台、交易平台等等。平台化最核心要点的是业务抽象建模和系统架构的开放性：业务抽象解决 80%的共性问题，系统架构开放性解决 20%的个性化问题。

（4）业务中台。随着生态的复杂度、业务的复杂度、系统复杂度的升级，又遇到了新的问题。领域的平台化虽然解决了领域内部的问题，但是每一个业务的执行都是

跨领域的，涉及会员、商品、交易、营销、店铺、评价、支付、物流、售后等等……一套业务逻辑横跨几十个系统。比如一件衣服的商品发布规则、交易规则和营销规则均分散在不同的系统中，而且相互关联，时间一长就没有人能说清全局了。如果程序员通过翻查代码还原出所有的逻辑，代价极大。事态发展到后来，我们会发现评估需求的成本可能会大于实际开发的成本，而真正有效的工作占比很少，导致整个研发效率和业务响应速度都比较差。这已经不是单纯的技术问题了，而是复杂生态的协作问题。这时，我们开启了第四个阶段：业务中台化阶段。此阶段主要解决信息获取成本高、互联互通成本高、服务具有不确定性和低水平重复建设这四个问题。

那么，如何来解决这些问题呢，我们可以了解一下传统的建筑行业和互联网本身的基础设施建设，基本上都要靠三样东西来共同解决复杂生态的协作问题：

- 协议标准、运行机制。
- 满足标准的分布式执行单元。
- 中心化的控制单元。

比如移动互联网，我们现在之所以上网能用手机，它的根本是什么呢？网络的协议，也就是我们对网络的理解，它是基石。在这个前提下，我们的各种设备，不管是什么品牌的手机，只要满足 3G 协议、4G 协议，就可以插卡上网。也就是这张 SIM 卡确定了我们的身份，从运营商控制网络上获取了控制信息，它知道我们是谁，能不能上网，网络速率等等信息。再回头来看我们的电商生态，就是要根据我们对商业的理解，把一些基础协议梳理出来。例如什么是业务？什么是业务身份？各个业务领域的边界是什么？每个领域提供的基础服务是什么？领域服务和领域服务之间的流程链接标准是什么？之后，再在这些思想的指导下建立业务平台化的实施标准和业务管控标准。因此，电商业务中台是一套由业务能力标准、运行机制、业务分析方法论，配置管理和执行系统以及运营服务团队构成的体系，能够给各业务方提供快速，低成本创新的能力。

（5）构建基础平台。从业务开发角度来看，中台主要是为了提升业务的开发效率，此处的开发效率主要是指个人协作的效率。如果换成从机器维护的角度来看分工协作，那么技术要解决的无非就是数据、算法和计算三方面的问题。

- 数据。哪些数据有效，哪些数据重复，数据该放在哪个数据中心；
- 计算资源如何利用。平时大部分机器的负载都比较低，如何有效利用这个计算资源，在高峰和低谷都能充分发挥它的作用；

- 计算和数据的协作。计算和数据是否放在一起？如果不放在一起，迁移数据就要考虑带宽问题，会增加新的成本变量；
- 计算性价比的评估，依赖于算法。

数据、计算资源和好算法才能构建出优秀的基础设施，在此基础上才能给上层业务提供更好、更稳定的基础，所以搭建高效的基础平台是非常重要的。

3. 组织管理的复杂度

其实复杂的问题都不是技术上的，往往是人和组织上的，所以如何提升人和组织的效能就比较关键了。

（1）呼唤全能工程师。在几十个人维护一个单系统的情况下，决定效率的就是这几十名工程师的技能水平，每个人的效率往往就决定了整体的效率。这种情况下Superman 能发挥很大的作用。

（2）呼唤系统架构师。当一个团队达到上千人时，单系统肯定搞不定了，必须要构建分布式系统了，工程师必须要分工了。这个阶段最容易从业务开发团队中诞生中间件团队，他们专门解决系统之间的连接问题。这个时期也会诞生一批能力比较强的系统架构师，他们决定系统该如何设计以保持高可用、高性能和高扩展性。从组织建设上，整个团队会按照技术分工的维度进行细化拆分，如：

- 产生架构团队即系统架构师，对整体的系统架构进行规划，保障总体设计的高可用、高性能和高扩展性；
- 产生业务开发团队即业务开发工程师，专注实现业务逻辑的开发；
- 产生中间件团队，专注开发和维护系统中一些通用技术组件，为业务开发提供支持，提升开发效率；
- 产生 UED 团队，解决界面交互问题；
- 产生测试团队，保障开发的可用性问题。

（3）业务平台团队诞生。当团队达到几千人时，光靠技术角色分工已经无法解决问题时，就必须开始平台化建设，也就是业务架构师要发挥作用的时候了。公司的每个业务领域必须进行平台化建设，如电商业务中进行商品平台、交易平台、营销平台、会员平台的拆分。这些拆分后的平台再为上层业务提供基础的服务，便于上层业务进行更多元化的组合。在这个阶段组建业务平台团队是最合适不过的了，这样可以解决公共基础业务的集中管控问题，避免基础服务的重复和无序建设。

（4）业务中台组织诞生。当公司规模达到几万人时，一般公司都会采取多个垂直化事业部的组织形式，每个事业部一般都是全编制的技术团队，这其实也是重复造轮子最严重的时期。但是，一些基础的业务能基于业务平台中的 service 构建业务吗？其实也很难！因为人员一旦增多，再靠人与人之间的信息传递已经不可能有效运转了。例如你甚至很难知道公司当前到底提供哪些服务了，因为如果没有机制保障服务的注册和发现的话，它的获取成本会非常之高。此阶段影响效率的主要就是信息获取成本、互联互通成本和违约成本。

当公司达到上万人甚至几万人规模时，必然存在以下两种情况。

第一，没有一个平台型的业务部门，例如公共业务平台、中间件、基础技术平台。在这种情况下，各垂直业务部必然会建设各自的平台，产生大量重复建设，导致某些技术基础设施和业务基础服务不统一，甚至公司的技术栈都不一致，严重影响公司的效率和长远发展。

第二，有一个平台型的业务部门，但是也会出现各种问题。

- 不知道谁有什么样的服务能力、由谁提供支持、服务质量如何，团队信用度如何；
- 找到了有能力的团队，但 BU 目标不一致，不一定会获得支持；
- 沟通不畅、对同一个名称的理解各异（“多国语言”），需要“翻译”；
- 支持的质量与个体能力有差异，具有不确定性；
- 系统间协同难：同一个需求需要在多个系统中实现，相互连接需要定制，导致成本高；
- 后续支持不可控：开始支持，后续不支持了，没有显性违约成本；
- 支持方即便做得好也没有可度量的标准，缺乏长远的动力。

显然第一种情况我们是不提倡的，无法持续发展；但是第二种情况也会存在各种问题，这些问题也正是构建业务中台需要解决的问题。

3.2 什么是中台

所谓的业务中台就是：通过制定标准和机制，把不确定的业务规则和流程通过工业化和市场化的手段确定下来，以减少人与人之间的沟通成本，同时还能最大程度地提升协作效率。

- 中台的目标：减少沟通成本，提升协作效率。
- 中台的实现手段：制定标准和规范。
- 原则：集中管控，分布式执行。

那么应该如何建设中台，我们先看一下中台的定位。

1. 中台的定位

所谓定位就是清楚地告诉别人我有什么、我要什么和我不要什么，对电商业务来说，业务中台内容可参见图 3.1。

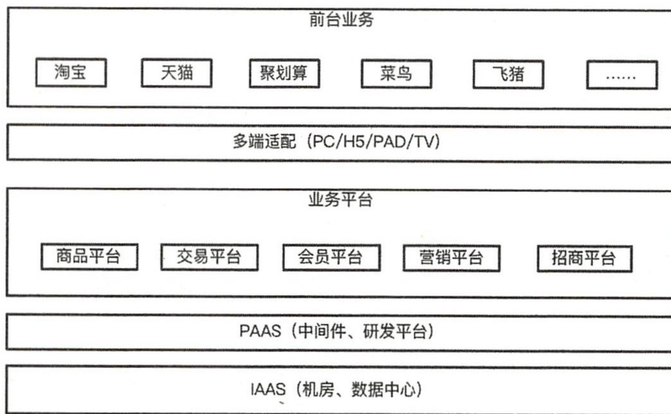


图 3.1 中台的整体架构

业务中台需要收敛一些基础的业务服务，如会员、商品、交易、营销和结算等。这些基础的服务会被整个电商业务使用，所以统一管理是很有必要的。

那么业务中台还需要什么？因为中台的目标是要向上层业务提供这些基础的服务，那自然必须能够清楚地描述自己到底有哪些服务、数据和功能，我们可以把它统称为能力。所以还需要能够定义能力（标准和规范）、能力的发现、能力的注册、能力的列表以及能力的评价和更新机制等。

业务中台也不是什么都做，除了有基本的基础服务和服务能力外，还要定义中台的边界。图 3.2 描述了业务中台一些基本的工作范围，它需要能够对接能力，同时又服务好能力使用方，而自己并不负责实现具体的业务。

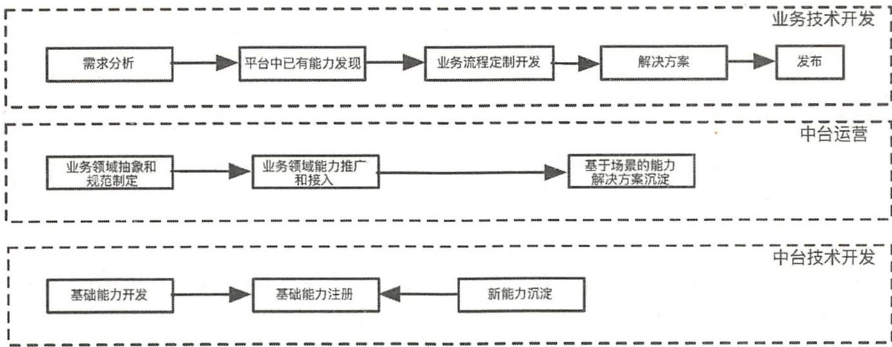


图 3.2 中台的职责范围

2. 中台的生命周期

如图 3.3 所示，整个中台包含如下的软件生命周期过程：需求域、实现域和执行域。需要域主要通过需求的分析将需求结构化，形成业务清单和能力地图（本质上就是确定这个业务所需要的数据和功能组件接口）。实现域主要就是将数据和功能组件接口整合成服务该业务的实现（并给他标识一个身份），如果当前的功能集不满足的话，还需要有部分的开发工作。最后的执行域就是把这些功能代码或者配置部署到线上，让用户去实际执行。

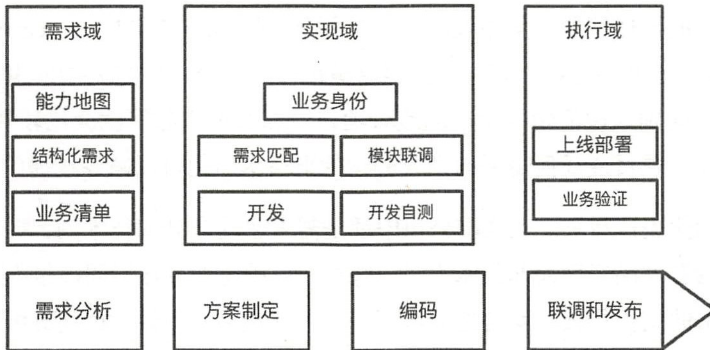


图 3.3 中台的生命周期

3.3 提升中台的效率

我们再回到业务层上来具体看一下如何提升研发效率。说到研发效率还真不仅仅就是如何写代码这么简单，读过《人月神话》一书的人都知道，要开发一个产品，并不是人越多越好，而是涉及人与人之间的信息传递成本，有时候人与人的沟通成本甚

仅供非商业用途或交流学习使用

至大于写代码的成本。另外，当系统变复杂了以后，代码应该写在哪里，也就是人要去理解这个原有的系统也变得很困难。有一个冷笑话：评估一个业务需求，先要找各个系统的产品经理花上 2 周评估时间，然后开发同学再用 2 周时间评估怎么实现，最后真正写代码只需要 1 天时间。由此可见，大部分时间并没有真正花在实现业务需求上，这样的研发效率肯定是不合理的。

所以要考虑如何解决人写代码的效率问题、人与人沟通的问题，这是解决业务研发效率的关键所在。

1. 沟通效率问题

有合作必然就有沟通，提升沟通效率最好的办法就是形成默契，要形成默契就要通过规范和约定等手段把大家圈在同一个语言频道上。

需求阶段的沟通比较多，如统一术语、需求结构化表达、统一业务身份。

（1）统一术语。这在一个公司非常重要，比如在某些公司，“PM”这个词是指产品经理，而在另外一些公司是指项目经理；还有诸如应用、系统、模块这些基本的称谓，如果大家理解不一样的话会很糟糕。

在公司中形成术语也有一些技巧。比如给产品取一个好名字，名字既可成为术语也可以是很好的传播符号，像滴滴的一个服务框架叫 DiSF（didi service framework，滴师傅），这个名字有内容、有含义很容易被记住。在公司中与人沟通也是一样，如果很难一句话向别人说清楚某个产品或者项目，那么推广起来就会比较费劲，因此，把自己的产品、项目或任何需要向别人表达的事情术语化可以减少沟通的成本。

（2）结构化表达需求。互联网公司中都是需求驱动产品和技术的，提出需求的人大多都是运营和产品经理而非技术人员，这就不可避免地存在不同岗位人员的沟通理解问题。需求的结构化表达就是把需求用一系列的术语、图标、页面等可以更好理解和呈现的方式（一般产出就是 PRD）在同一个语境中表达出来，让对方更好的理解。

沟通需求的过程就是把不确定性确定下来的过程，需求越具体沟通越容易。把需求结构化，和把系统中需要经常改动的逻辑配置化要达到的效果是一样的，最终产生的结果就是一个变更记录。

（3）统一业务身份。业务身份是管理一个业务在各个业务域中定义的业务规则索引，是一串平台可识别的编码，该编码由构成业务的要素经过一定组合关系运算生成。在平台的运行域中，各个业务域的系统根据输入的参数条件，进行业务身份判断运算，

仅供非商业用途或交流学习使用

最终根据识别出的业务身份结果，执行该业务在本系统定义的业务规则。

要实现统一业务身份必须要解决：

- 系统之间同一项业务的联通性问题，让系统自动呈现业务整体视图；
- 业务条件没有生命周期管理、系统长期维护困难的问题，要统一业务条件识别；
- 业务上下线相互影响、回归工作量大和效率较低的问题。这就要对业务逻辑进行能力抽象，建立封闭性，从而隔离业务。

要对业务身份进行统一管理，需要实现：

- 对业务身份标识的统一注册和管理，一般需要构建一个运营平台；
- 有业务规则的配置界面；
- 规则的执行引擎。

所以统一业务身份需要入口的注册管理、规则的配置与变更，以及规则的运行域，缺一不可。

2. 开发效率

如何高效地写代码是程序员永恒的话题。这涉及很多因素，如程序员对代码语言本身的掌握程度（比如 JDK8 中引入闭包代码可以使代码更简洁），写代码所用的 IDE 以及快捷键的使用程度，等等。笔者在这里先抛开这些问题，阐述下从开发到测试再到运维的整个效率问题。

开发人员都不希望别人乱碰自己写的代码，对代码有绝对的控制权，所以一般都喜欢掌控（Owner）系统，即这个系统我说了算。在这种情况下，曾有一段时间我们把系统拆得很小，结果诞生了很多同质的系统，越来越多地在做重复的事情；此后又经历了系统合并的阶段。但是，系统合并也会带来新问题，即开放过程中冲突比较厉害，包括打包部署的效率都很低，在这种情况下会有两种解决方案：一是开发态和运行态分离；二是对系统进行分层和抽象建模。

所谓开发态和运行态分离，就是大家线下的开发都是独立进行的，包括打包和部署，接口的调用分开，走远程调用。但是在线上部署时，都是部署在同一个容器中，把远程调用变成本地调用。这种思路我们在“合并部署”一章中有介绍，本质上可以做到开发态和运行态的分离，同时兼顾开发效率和运行效率，如图 3.4 所示。

仅供非商业用途或交流学习使用

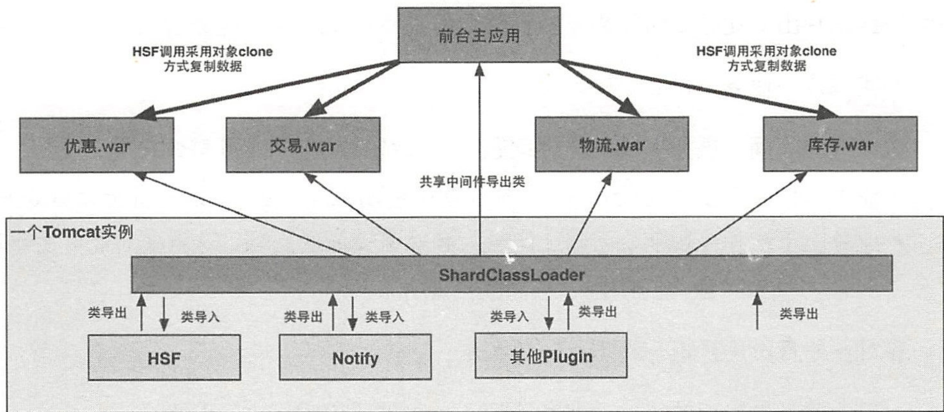


图 3.4　开发态与运行态分离 1

将开发态和运行态分离的另一个方法就是采用 Node 技术。Node 被很多前端同学推崇的重要原因也正是它把前、后端的开发解耦了，解决了开发和调试效率低下的问题。但是这里也会出现一些问题（我们在“无线化”一章中已有分析）：解决办法也是将开发态和运行态分离，通过 jtemplate 模块引擎使前端可以在 Node 中开发程序，但是线上仍然可在 Java 中运行，兼顾前端和后端的开发效率和线上的维护成本，如图 3.5 所示。

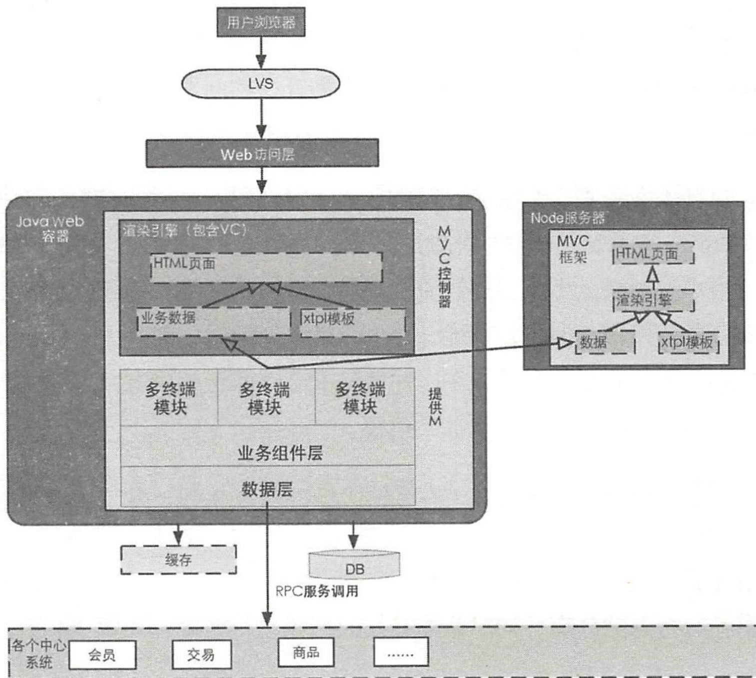


图 3.5　开发态与运行态分离 2

仅供非商业用途或交流学习使用

第二种对系统进行分层和抽象的解决方案是非常常用的。最典型的是对交易环节进行一次重构，对典型的交易场景进行抽象建模设计，对核心流程进行流程引擎改造，使得交易流程可以增加很多扩展点，不同的业务场景可以根据需要扩展自己的个性逻辑，整个交易环节抽象成一个流程框架和一系列的业务扩展，每个不同的业务之间互不干扰。

3．测试效率

整个软件生命周期涉及很多环节：需求、开发、测试、上线、运维……涉及很多协作。这些环节都会对效果有影响。其中，测试效率非常重要，因为测试花费的时间几乎和开发所花的时间是一样的。关于如何提升测试效率，我们总结了一些实践经验，分述如下。

（1）全链路 Beta 测试

继续保持 Beta 环境与线上环境的一致性，将核心链路上的应用，Beta 环境 HSF 打通，减少 90%由于环境问题导致的 P1、P2 故障，如图 3.6、图 3.7 所示。

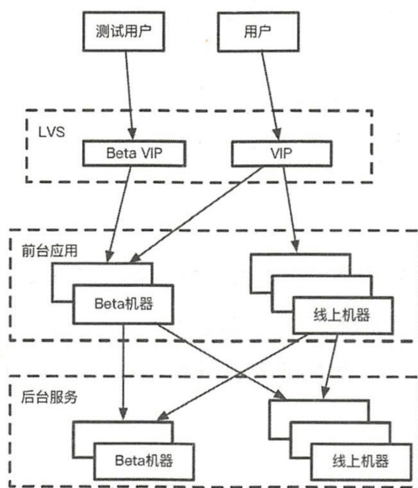


图 3.6　Beta 发布环境改造之前

仅供非商业用途或交流学习使用

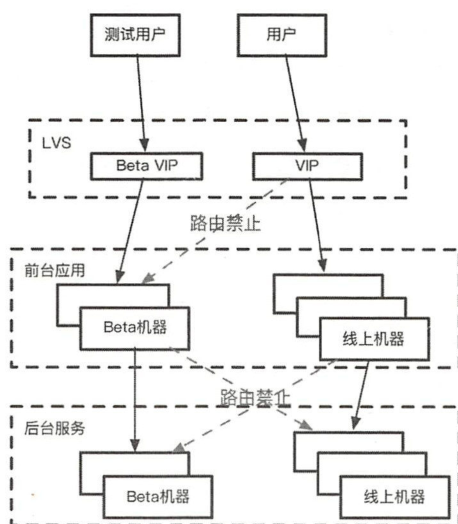


图 3.7 Beta 发布环境改造之后

打通之后，可以实现以下效果：

- 测试环境可以做到召之即来，挥之即去；
- 在分批发布前，可以在极短的时间内有针对性地验证核心功能；
- 也可以选择性地屏蔽 Cache 的访问；
- 数据轨迹可以实时透出。

4. 运维效率

运维包括线上和线下两部分，运维效率会在两个环节表现得最明显，一个是线下的打包编译步骤、代码分发步骤；一个是线上的下线→重启→上线步骤、发布检查步骤和回滚步骤。下面我们分别看看在这些环节有哪些地方可以优化。

（1）打包编译环节

- 优化流程。环境分配，可以预先分配好代码 copy，要主动准备而不用每次编译代码时再做环境方面的准备工作；
- 预处理。监控代码版本修改，当代码被修改后，自动触发代码合并冲突检查做代码编译和打包操作，不要等到用户点击再触发；每个分支代码更新主动和主干做 Merge，发现有冲突要主动通知相应开发人员修改，不要等到打包部署时再临时修改；

仅供非商业用途或交流学习使用

- 代码编译优化。规则检查，业务依赖的包要做依赖规范化管理，通过工具识别依赖；减少应用依赖 SNAPSHOT 版本 Jar，可以节省 Maven 编译时间；Maven 打包优化，优化 Maven 配置减少不必要的消耗；
- 增量编译。减少编译时间的办法之一就是只编译变化的部分；比较代码修改时间和编译的代码更新时间可以区分那些修改的类，并针对它们做增量编译，大大减少编译时间。
- 打包机器硬件升级。提升编译速度的另一个办法是升级机器硬件，使用更多的 CPU 或者更多的内存可以明显提升编译速度；多组机器 standby 以处理并发修改情况，并始终保持应用处于可用状态，减少开发上厕所的次数。

（2）代码分发步骤

代码分发主要考虑两个问题，一个是代码的下载，最好是支持 P2P 下载，这样的下载效率最高（虽然大部分情况是 HTTP 下载较多，但真心不建议采用）；二是如果代码包比较大且同时下载的机器比较多时，要考虑下载机器的网卡流量是否满足，这一点必须特别留意。

（3）下线、重启、上线步骤

- 下线环节。下线被动等待 15 秒健康检查失败，能否主动通知 LVS 下线，而不是被动等待 3 次 3 秒的检查失败后再下线；
- 重启。初始化各种服务，去掉不必要的服务初始化，将一些服务改成慢加载，部分服务可以并行初始化。

（4）回滚

回滚等于重新发布，直接利用本机的老 war 包快速重启，不需要再走包分发步骤，要有手动回滚脚本。如果回滚时间长则减少回滚批次，采用发布一批机器就下线一批机器的方式：下线的机器保持 standby，老代码不提供服务，出现问题后再立即下线新发布的机器，将 standby 的机器立即上线。这样可以快速达到回滚的目的，在 30 秒内就能完成回滚。

5. 中台的典型实践

我们以交易为例来说明中台的实践。之所以选择交易场景为例，是因为交易是多领域数据的汇聚点，涉及的数据比较多也比较复杂，例如涉及商品特征及库存、会员权益及资产、营销活动、物流订单和支付等。它不仅有垂直的业务场景如 3C 以及所

仅供非商业用途或交流学习使用

有针对 3C 产品的特殊配送安装的交易规则，还有很多横向的业务场景（如聚划算业务，3C 产品也可以参加聚划算），而所有参加聚划算的商品又是另外一种交易规则。这些交易规则最后都要在下单环节实现，可以想象这里面肯定有很多冲突的规则，怎么写这些 if...else...是令程序员头疼的事。

所以在实践中，就会产生如下问题。

- 业务间相互耦合、没有隔离，任何改动都会影响其他业务，导致分析影响和测试的成本高；
- 业务逻辑没有分层、没有抽象，不能区分公共逻辑与个性逻辑；
- 需求实现成本高，每次发布周期太长；
- 组织问题：业务边界不清楚，业务实现人员很难产生主人翁（Owner）感。

如何解决这些问题？业务耦合和分层都是典型的技术问题，这是一个典型的由复杂业务逻辑所导致的软件架构混乱的问题，可以通过软件设计上的抽象和建模来解决。分层设计的思路如图 3.8 所示。

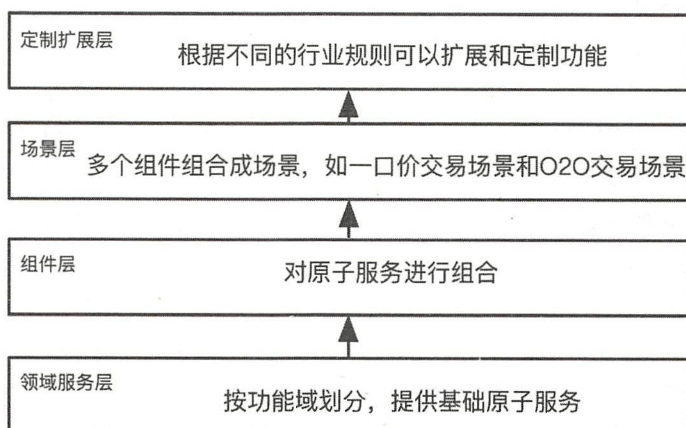


图 3.8　中台架构分层设计

架构分层的目标是要降低系统间的依赖、可以做到某一层的无损替换和增加复用度，让开发人员有更好的分工，还可以将功能进行组件化设计增加复用性，如图 3.9 所示。

将分层设计和功能的组件化进程组合，就会形成一些基于业务含义的场景，也就是说场景就是：针对特定的业务类型，通过流程将一个或多个功能组件进行编排，并通过数据和规则的配置来实现特定业务逻辑。如图 3.10 所示。

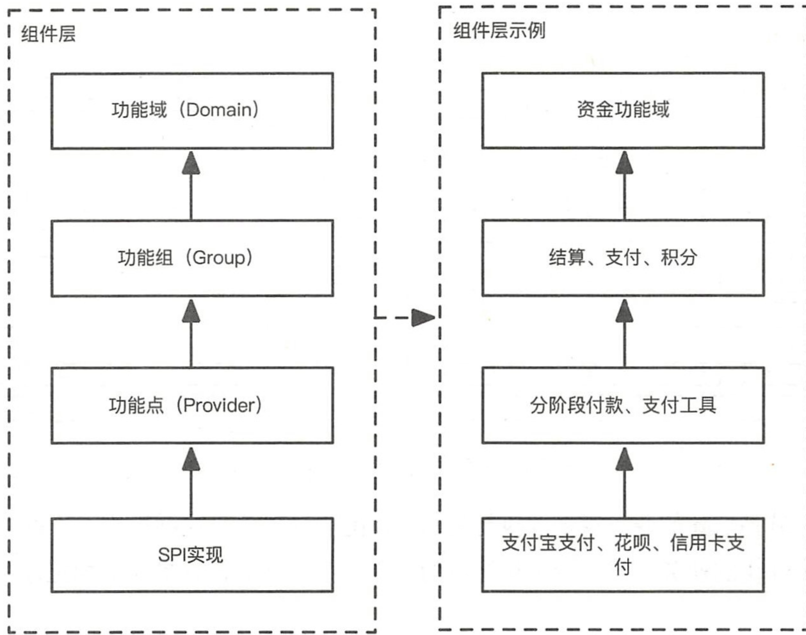


图 3.9　中台功能组件实例

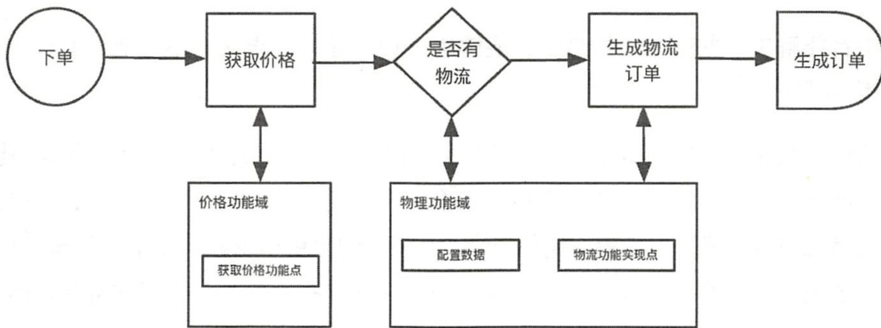


图 3.10　中台的场景示例

在实际的系统构建时，可以把整个系统分成两部分来看：运行域和配置域。分离运行域和配置域，把更多固定不变的流程通过配置的方式实现，以减少开发的工作量，同时也减少了服务变更的次数。如图 3.11 所示。

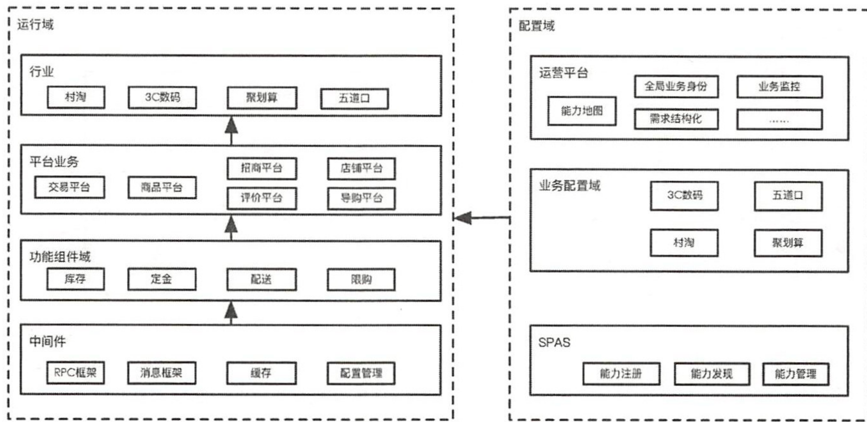


图 3.11　中台的运行域、配置域分离

运行域是实际的业务逻辑的执行机构，而配置域更多是控制机构、负责定制和控制部分个性化的业务逻辑，减少运行域的变更次数，从而保持平台的稳定性。

3.4　中台是否能解决一切问题

中台能解决一些问题，但是中台能解决一切问题吗？很显然不可能，中台也只在小范围内适用。

前面一直在说中台是为了解决效率问题，但是效率提升还离不开一个因素：成本。对互联网业务来说，仅从开发效率角度来看，当规模还没大到一定程度时，可以简单地通过增加投入提升开发效率。比如滴滴，从最早的出租车业务到专车、快车、代驾、租车和顺风车，每个业务线系统基本都独立，尽管这些业务的重合度非常高，但是为了能快速开发，把它们分开反而效率会更高。

不过，随着业务越来越庞大，复杂度越来越高，再这样发展下去，不论从成本（浪费会比较大）还是技术（不利于技术沉淀）来看，都是不利的，因此在这样的场景下非常适合使用中台。那么中台适合什么场景呢？最典型的场景就是公司多个业务线的业务场景相似、在技术实现上非常类似，像电商和出行这类业务。

那么，即使是在非常适合使用中台的场景下，是否一用上中台就一切都万事大吉了呢？很显然也不是。就拿前面提到的交易场景来说，我们对交易场景做了很多的抽象，进而根据抽象的结果建模，试图在一定的确定场景下灵活化处理，使建模后的结果更灵活，但是建模的前提仍然是针对特定场景的，所以这种场景的使用仍然会受限。

例如当前在第一版交易系统重构后，O2O 模式出现了，此前重构的模式就很难在原有的交易模型上良好运转了。

3.5 总结

一般一个业务系统会经历单系统、分布式系统、产品化、平台化以及最终中台化的发展历程。不同阶段的区别如下。

（1）单系统，就是单个系统，业务形态比较单一，所有业务逻辑在一个系统中实现，对应的开发协作一般在 10 个人左右。这种结构一般是在业务发展初期为了应对快速开发产生的，不用太多考虑稳定性和扩展性，唯一的刚性要求就是快速实现需求。

（2）分布式系统，当开发人员达到 100 人左右时，就必须拆分系统了，按照业务单元进行角色划分，要考虑好稳定性和扩展性，因为此时别人可能会依赖你的服务。

（3）产品化，就是更多地把系统当成一个产品来提供。当客户使用产品时要考虑他的学习成本、要考虑是否能够定制客户的需求、对用户的问题反馈是否能及时响应（售后服务）以及产品是否稳定可靠……这些都需要由产品的提供者来保障，也就是要尽量保证产品的标准化、规范化和可靠性。

（4）平台化，就是在产品化基础上，你不仅希望更多的人使用你的产品，而且还愿意邀请客户、合作伙伴一起建设和完善系统，给他们提供一整套的服务；你也不仅仅满足固定的需求，还会主动替客户着想，挖掘他的潜在需求。平台化比较适合团队规模千人左右的情况。

（5）中台。其实我们大部分的业务场景中只要做到业务的平台化就很好了，在业务边界比较清晰的情况下，只要把基础的业务平台建设好，就可以非常快速地组装新业务系统。但是当团队达到上万人规模时，信息获取成本高、互联互通成本高、服务能力不确定……这会带来非常高的协调成本，当协调成本达到一定程度时就不会再有协同了——每个系统都会倾向于自己实现需求而不是依赖别人——这就会导致每个业务要形成自己的闭环并产生很多的重复建设，成为恶性循环。中台就是用来打破恶性循环，建立便于协同的业务标准和机制的。

4 全球化下的网站演进：全球部署方案

随着业务的快速发展，全球化部署成为必然要求，在技术上实现业务系统的全球部署至关重要。全球化部署需要解决以下几个关键问题。

第一，业务核心单元的梳理。这些核心单元必须可以裁剪或添加，毕竟我们不太可能把一整套系统全部照搬到国外；

第二，核心单元必须可以快速部署到国外的机房，最好能够一键部署，即首先要实现单元化部署；

第三，实现全球数据连通。全球部署的最终目的是要能实现数据的连通，例如中国的卖家可以很方便地将商品发布到国外——一地发布、全球买卖；

第四，出于研发效率的考虑，部署在全球的业务系统要有良好的定制性和扩展性。

本章将围绕以上问题展开讨论：先介绍单元化和异地多活的实现，再介绍现实中国际化部署需要解决的技术难题。

4.1 国际化的背景

国际化一般有两种类型：一种是进口业务，像天猫国际和全球购；一种是出口业务，像速卖通（AE）和淘海外。如何实现全球运营？一种是走平台输出模式，即由总部公司负责输出技术和运营平台，由总部公司自己来负责每个国家的运营；另外一种是走本地化的模式，即总部公司只输出技术平台，运营由各国当地人负责。除此之外，第二种模式还有一个变种，就是收购本地的电商然后总部公司只做技术支持，运营仍然由本地人负责。

目前国际化主要有如图 4.1 所示的几种场景。

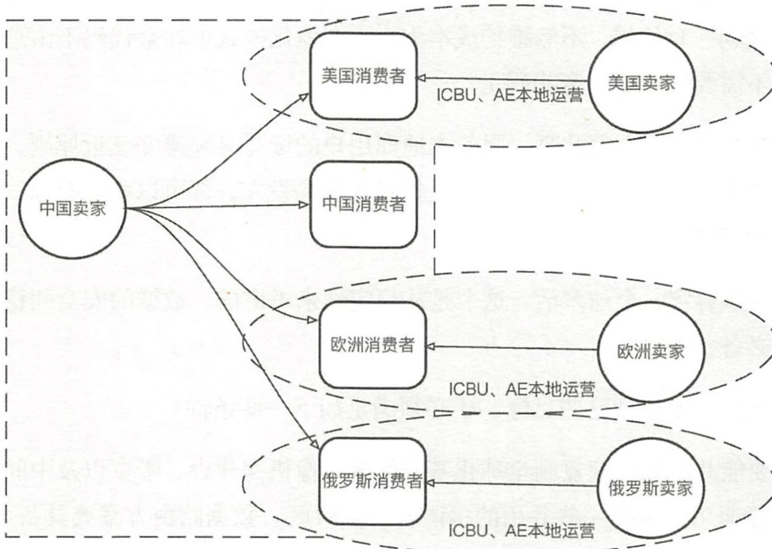


图 4.1 国际化的场景

与以上几个场景对应的系统建设存在以下两种思路。

（1）将国内的系统完整的复制过去在本地重新搭建一套，实现本地化运营，两边的系统相互独立，数据不通。

（2）只在当地建设个性化的系统，当地的系统和国内数据是打通的，整体还是一套系统。

很显然我们要介绍的国际化建设是针对后面一种来介绍。

4.2 面临的技术挑战

全球部署和后面将介绍的国内单元部署有些不同的地方：一是它必须采用就近访问原则，必须要保证用户体验；二是它不能跨区域进行大流量的并发读写，因为延时比较大，对系统的吞吐量会有致命的影响，所以这种情况下要避免中心节点。

除了前面介绍的业务上的挑战，在国际化中技术所面临的挑战也很大。前面提及两种业务场景下，都会遇到国际化部署的问题，国际化部署必须要达到以下目标。

（1）单元化。业务系统首先要实现单元化部署，根据核心业务流程梳理出核心单元，单元内尽量做到单元封闭，保证数据按照单一维度进行单写。

（2）一套代码、全球部署。即不管是国内还是国外，如果业务都由总部公司维护，那肯定要做到一套代码，不然维护成本太高（本地化模式中如果代码不由总部公司维护，可能不需要统一成一套代码）。

（3）服务本地、数据共享。服务本地即用户的读写必须遵守就近原则，否则跨国访问的延时太高体验太差，数据共享是指一次商品发布，即可以卖到全球，因此数据要共享到所有的站点。

（4）区域容灾、全球多活。这个是从稳定性来考虑的，数据的安全和稳定性是网站运营的必备条件。

技术上除了要达到这些目标，还必须满足如下一些条件。

（1）要能共享和快速复制全球化基础设施。像机房建设、带宽以及中间件这些软件环境要能通用，包括一些共用的多语言、多时区、数据路由方案要具备可复制性，这都是非常重要的。

（2）基础业务数据要能互通。比如要能打通商品、商家、店铺、营销数据，整个数据要能共享，能被每个业务使用。

（3）业务系统要做抽象、提升可扩展性，能快速支撑业务发展。系统的每个功能点如果能做到可组装、可裁剪，就对国际化部署非常有利——系统的灵活性越高，业务支撑效率就会越好。

4.3　全球部署的目标架构

图 4.2 展示了全球化架构的一个设想。

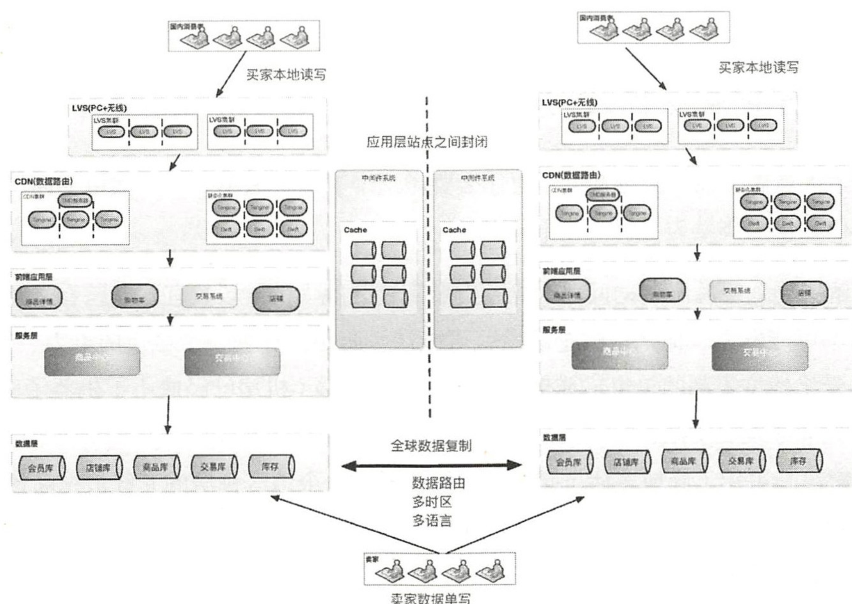


图 4.2　全球架构设想

我们首先来看看如何实现单元化。

4.4　何为单元化

所谓单元化就是按照某种维度对数据进行水平拆分，拆分后数据分布在不同地域并且对数据的更新是单写的，它要做到：

- 业务单元内闭环。单元内所有业务操作要尽量做到自封闭，各单元内包含业务所需的所有服务和所依赖的全部数据；
- 单元化部署，即单元作为服务扩容部署的基本单位。

所以单元化其实是一种相对完美的妥协，把数据进行单元化拆分的核心目的是为了解决一致性问题：因为不可能在数据层做分布式事务，但是为了部署的便利性，就要使单元尽量封闭从而方便做单元的部署。

单元化同样受到很多约束：

- 数据正确性。首先要保证数据不会错，其次保证数据不会丢；
- 单元封闭。减少跨单元访问，尽量做到对业务透明；
- 就近访问。实时性、高性能。

4.5 单元化解决什么问题

技术人员做事情一般都会问：为什么要做这个事情？做单元化当然也要回答这个问题。单元化主要是为了解决以下问题。

- 解决物理资源限制的问题。电商发展到这么庞大的阶段，有些问题看来几乎是不可能发生，但是在这个阶段就可能产生瓶颈，比如一个城市的电力、带宽、受机架位限制的单机房或地域会出现容量瓶颈（机房已经放不下机器了），导致业务发展受限；
- 解决高可用（异地多活）问题。比如要考虑一个地区机房的光纤被挖断该怎么办？如果一个城市发生地震该怎么办？像这些机房发生物理硬件损坏等各种情况都要有所考虑；
- 解决国际化、全球化业务问题。除了考虑资源瓶颈和安全因素外，还要考虑成本和体验因素：哪个地方的机房建设成本更低，国际化业务的发展需要使网站的部署更接近用户，这样才有更好的体验，等等。

以上种种因素都要求从一地多机房变成多地多机房，更重要的是做到多地能同时提供服务、流量可以在多地之间切换，而不仅是简单地备份站点。

4.6 单元化数据分片方案

多地部署而且多地同时提供服务，例如部署多地多写，必然会设计分布式事务问题，最典型的是商品减库存：多地的买家下单同时都减库存，必然导致商品被超卖，一旦发生超卖就会导致非常严重的损失。但是，如果依靠分布式事务来保证数据一致性，性能又会成为瓶颈。试想，如果在双 11 的高峰时段，那么大的请求量如果都有事务的话，DB 基本无法工作，那么这该如何破解呢？下面介绍几种数据单元化的划分方式。

1．中心-多单元模式

考虑到多单元建设的成本问题，不可能也没必要把所有系统都单元化，只需要把最小的核心系统单元化就能达到目的，即大部分业务系统放在中心单元，再对中心单元的系统做冷备，如图 4.3 所示。

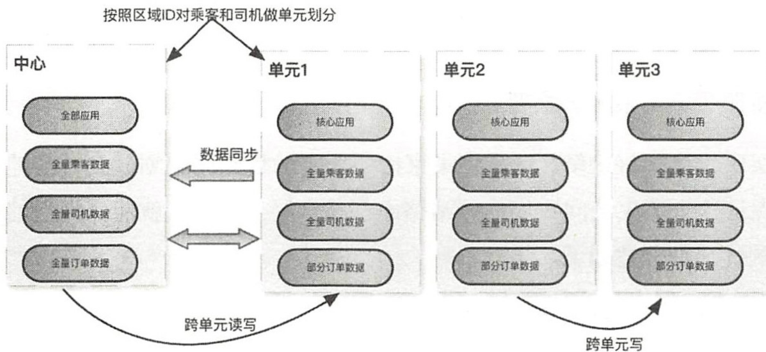


图 4.3　中心-单元模式

单元化的核心系统可能会对中心-单元的系统存在一些跨单元调用,这些调用可以尽量做到异步调用。

在数据层面，由中心保留全量数据，单元机房可以根据需要决定是否需要存储部分业务的全量数据。

2．多机房 A-A 模式

单元化要解决单业务的单机房的资源限制，就必须让同一业务系统在多机房同时提供服务，即业务层面的 Active-Active 模式（数据库级的 A-A，但是表行级数据仍然是 A-S 模式），但是由于我们不会在多机房之间做分布式事务，所以必须对数据进行分片，保证同一数据的更新始终发生在同一个机房，所以这是一个更细粒度的 Active-Standby 的模式。

要做到这一点，必须要做两件事：

第一，按照某个维度对数据进行划分：比如是司机还是乘客；

第二，解决数据层的异地复制和一致性问题：数据库双向复制和一致性校验。

3．数据按照什么维度划分

典型的电商数据一般分为买家数据、卖家数据（针对出行领域就是乘客和司机模

型），商品理论上也属于卖家数据。按照不同维度来看数据主要是想保证数据的单写，也就是给不同维度的数据做路由。例如，如果按照买家数据维度来划分的话，就会把这个用户的写操作一直路由到一个地方去实现，这样可以避免多地同时写带来的数据冲突。

按照不同的维度划分就意味着其他数据都要绑定上去，我们来分析下按照不同的维度划分都有哪些利弊。

（1）按照买家（乘客）维度

按照买家维度很好理解，就是将买家按照一定的规则固定划分到每个地区的机房中，只要保证这个买家访问我们的网站路由都是全局一致的，就能保证他的数据的一致性。它的优缺点分述如下。

按买家维度划分数据的优点如下。

- 按照买家维度，可以更容易把与买家相关的数据聚集在一起，聚合买家需要操作的数据可以保证买家读写数据的体验。例如买家需要查询订单列表数据，那么由于买家的数据都在一个机房，所以不需要到多个地方去查询数据。
- 由于买家的操作是高频操作，所以对数据的一致性要求更高，对数据的实时性要求也同样如此。例如买家下了单立即就能看到信息，不能存在因数据异地同步而产生的延迟感，所以最好不要分离买家的读写操作，而按照买家维度来划分刚好可以满足这个要求。
- 按照买家维度来划分，买家的数据天然就是分片的，业务上没有跨机房的读写需求，那么就不需要对买家数据做过多的多地复制，可以减少数据中心的复制成本。

当然这种方式也有以下缺点。

- 按照买家维度划分数据，那么商品就必须多地复制，假如深圳和上海的买家都要买同一个商品，这个商品就必须在两地做同步复制。
- 卖家的操作会很困难，例如不同买家的订单在不同的机房，对同一个卖家来说，必须做多机房的读取，在体验上会有损失。

（2）按照卖家（司机）维度

卖家维度主要是把卖家操作的相关数据聚合在一个机房，例如卖家的商品数据、

营销数据、订单数据、店铺和库存等。所有对这些数据的操作都要保证单写。卖家维度最大的好处就是卖家操作的体验会大大提升，缺点自然就是牺牲了买家的体验，如果一个订单中涉及多个卖家的话，会涉及跨机房的调用，延时会比较大。

不管是按哪个维度，我们都需要考虑按照什么规则进行划分，一般来说，须遵守以下规则。

第一，就近访问。天然地根据买家的地域对买家进行划分，可以最大程度地保障买家的用户体验，也能充分利用我们的 CDN 优势，看起来是个不错的规则。但是根据地域划分也带来一个不可控因素，即需要依赖 DNS 的解析，但由于 DNS 的解析通常有很多不确定因素（例如用户可以随意切换 DNS 的域名解析服务器地址），所以必须在服务端做额外保障，这样会增加复杂度。

第二，Hash 取模。即将用户的数字 ID 根据 Hash 值取模，随机分配到某个机房。这个划分方式比较简单，也比较容易控制，缺点就是没有考虑用户的地域因素，比如有可能深圳的用户会访问上海的机房。

第三，对用户建路由表。即在用户与访问机房之间建立一个映射关系，通过映射关系表来控制用户访问。这个方式看起来比较灵活，但这个用户路由表将会非常庞大，也容易成为中心节点，稳定性和性能瓶颈会是一个隐患。

综上，按不同维度划分的影响也不一样，按照买家维度划分对买家有利，按照卖家维度划分对卖家有利。是更多地保障买家利益还是卖家利益，自然不必多说了。

4．数据漫游问题

数据漫游就是当我们把某个买家已划分到某个单元，但是买家物理位置发生漂移。例如正常情况下我们把美国的用户默认划分到美国的机房，但是如果他来中国出差，想要访问我们的网站时，我们是否还要把他的请求路由回美国机房再访问呢？

为了保持数据一致性，我们要么把他的所有写请求再路由回去，要么把他的路由规则改到中国来，让他默认就访问中国的网站，为实现这一点，我们需要把他的所有其他数据都漫游到中国，比如账号信息、积分优惠券等。不然中国的系统可能就无法获取这些信息了。

4.7 数据路由方案

数据路由方案是根据前面讨论的数据划分的原则来保证数据的读写的正确性，最终的目标就是要保障数据都是单写的、不会存在数据错乱问题。例如一个卖家的商品同时被中美两地修改，那么在做数据复制时就会存在冲突，就会出现数据的一致性问题——这是要避免的，由此可知保证数据写在一个地方是非常重要的。

数据路由方案可以通过以下几种方法来实现。

1. 主键 ID 上做标识

这种方法最简单，即主键 ID 自身携带路由信息，如 UserID 后四位表示用户所在区域，那么我们根据 UserID 做路由只要直接取出后四位做判断，例如打车的订单 ID 中就带有 cityid，所以用订单 ID 做路由就比较简单。

但是用主键 ID 直接做路由的场景比较少，大部分情况下很难有远见在主键 ID 上做标识，而且以后想增加也很困难，涉及的改动会比较大，所以如果有当然最好，没有的话此路也就不通了。

2. 设置路由表

这种路由方式是将所有路由信息放在一个映射表中，最简单的就是一个 k/v 对，例如 UserId⇔cityid，接口调用时根据 UserId 查询 cityid 再根据 cityid 来路由，但是也有如下限制。

第一，路由表存放的信息要在内存中，因为访问量会很大，否则会产生性能瓶颈；

第二，路由表需要集中控制，所以必须是单点，一旦路由表错误数据会错乱；

第三，路由表存放的数据有限，不能太大，否则内存是个负担。按照经验，路由表可以设计成 bit 来做索引，一般存储 1 亿用户的数据没有问题，超过 1 亿就很困难了。

路由表的关键在于可以针对每一个用户映射一个目的地，相当于一个 Mapping 规则。这个方式显然比较灵活，而且可以随时修改这个 Mapping。例如一个美国人到中国出差时，我们可以把他的路由规则从美国切换到中国，那么他在中国也都是就近访问了。

但是做路由表也有非常大的问题，就是这个 Mapping 会非常大。假设一个 long 类型的用户 ID，映射到一个 4 位 int 类型的目的地区域码时，所占的空间是 64+4bit，

如果是 20 亿用户的话就有 1.5GB，全部放在内存就要求每个系统的每台机器上都分配这么大的内存，这几乎是不可能的。如果放在磁盘上的话，性能会受影响，因为针对磁盘的访问会产生非常大的瓶颈，所以必须要采用另外的办法，比如采用 Bloom filter 算法（即用一个很大的数组的索引来表示这个用户 ID）。

即使是用 bit 来表示用户 ID，也始终会存在瓶颈，因为用户数一直都在增长，那么还有没有更好的办法？答案是肯定的：我们可以直接在用户 ID 上做文章，给每个国家或区域的用户 ID 分配不同的段，如美国的用户 ID 都是从 2 的 32 次方开始，俄罗斯是从 2 的 33 次方开始，规则如下。

（1）用户 ID 根据区域分段

- 使用分段规则计算用户所在区域
- 按区域划分用户群

（2）默认中国区域服务

- 不满足区域分段规则的海外用户，取中国区域用户差集
- 额外配置路由表数据

（3）不满足区域分段规则的海外用户（存量海外用户）

- 满足区域分段规则的海外用户（区域切换用户）
- 中国区域用户切换区域（区域切换用户）

路由表是最高优先级，路由判断的流程如图 4.4 所示。

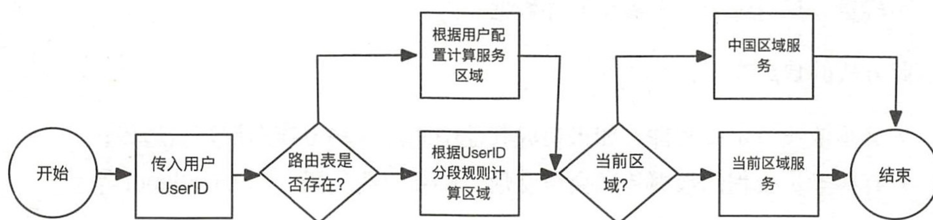


图 4.4　路由判断的流程

该方法的优点在于常驻内存高效、可靠、准确，可容灾；缺点在于需要单独维护，会占用约 100MB 左右的内存，数据必须异步推送。

路由表的方式比较灵活，可以做精细化的控制，但是路由表本身的数据一致性和可维护性会是一个挑战，另外也可能会成为一个单点，所以路由表本身的稳定性也要

非常小心的设计。

3. trace 透传

这个方式是将需要路由的信息通过 trace 透传下去，通过 HTTP 的 Header、RPC 的 Header、以及数据层等中间件透传下去，然后在接口调用的地方取出 trace 来做路由。

这种方式改动量最小，但是也有需要注意的地方：第一，trace 信息要能透传下去，最怕在有些地方丢掉 trace；第二，trace 的透传需要依赖中间件的配合，如果中间件不完善，透传将会很困难。

这种方式是给每个用户打上标记，在 Web 层查询这些标记（比如是 us（美国）还是 cn（中国）)，然后通过中间件的协议头把这些标记传递下去，该过程如图 4.5 所示。

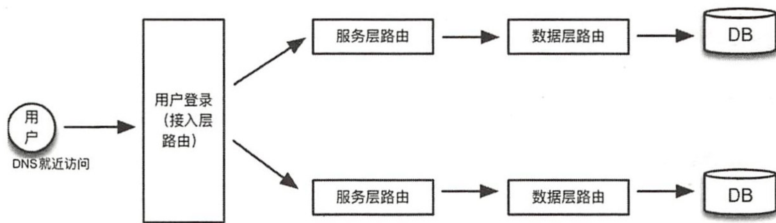


图 4.5　预计 TraceID 模式

此方式的优点在于：

- 不用改 API，不用必须带有 userID 等参数；
- 轻量，只有计算、不需要额外数据。

此方法的缺点在于：

- 必须依赖特定中间件，如果换成其他中间件，路由数据没办法透传；
- 有些异步线程的数据传递会很麻烦，必须改造代码，如 threadPool 等。

不管是哪种方式做数据路由，必须要在多个层次上来保证，如图 4.6 所示。

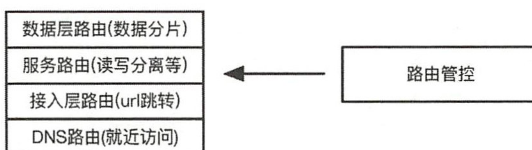


图 4.6　路由信息的多层保障

DNS 解析时保证用户就近访问本地的站点，在 Web 层做接入层的路由判断；然后是服务层保证服务调用的正确性，最后是数据层做最后的校验，防止数据写到错误的地方。

4．路由模式

我们会设置三种路由模式：中心模式（center）、单元模式（unit）、混合模式（mix）。

（1）中心模式

如果设置成中心模式，那么所有接口的路由都回中心单元，这是对那些没有进行单元化的系统设置的模式，例如我们把对乘客基本信息的写设置成中心模式，那么调乘客的写接口都由路由中心单元完成。

（2）单元模式

单元模式是指按照单元化原则进行路由的接口，都需要被路由到正确的单元完成，例如对司机写的接口都被强制路由到司机所归属的单元是。

路由是根据路由表、ID 或者 trace 信息由服务框架来完成的。

（3）混合模式

混合模式是表示路由原则可以是本单元优先，如果本单元不存在回中心单元调用（例如在一些读接口中可以配置成混合模式），则优先从本单元读取；如果本单元没有部署系统，则到中心单元读取。

- 读写分离

乘客和司机数据接口需要做读写分离的改造，方便单元部署，例如乘客个人信息的数据在单元机房只需要部署读接口，写全部在中心机房完成。中心机房的数据由 MySQL 同步到单元机房。

- 路由切换

当某个单元发生故障或者人为演练的时候，需要切换路由。

在本方案中就是要切换单元 ID<=>cityid 的映射关系（Map 结构），具体的对应如下。

- 乘客的路由：由于都在中心单元，所以路由信息不需要切换；
- 司机的路由：司机归属 area 重新映射到目标单元 ID；
- 订单的路由：订单中 area 重新映射到目标单元 ID。

切换顺序如下。

- 先在服务层、消息层、数据层禁写要切换的 city，抛错误异常；
- 先在消息层推送新的路由规则；
- 再推送数据层、服务层以及接入层的新的路由规则；
- 恢复禁写的 city。

4.8 接入层路由

搞清楚了数据按照什么维度来划分，接下去就要考虑怎么给这些数据做路由了，数据的路由可以分为多个层次，如图 4.7 所示。

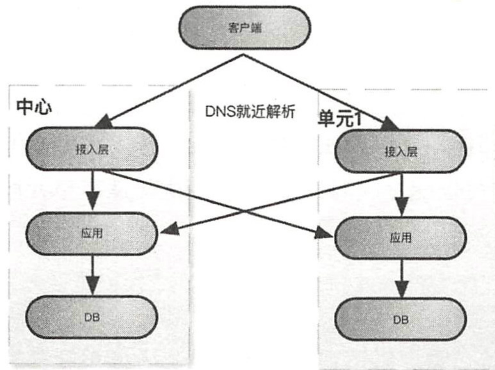


图 4.7 接入层路由

下面先看访问层的路由问题，假如君山被划分在上海机房，怎样才能让君山固定到上海机房去下单呢？首先，要判断君山该去哪，就要知道君山的 userID，如何知道？——可以让用户登录或者种 cookie；知道了 userID 之后怎么路由呢？我们也有以下两种办法。

（1）基于多域名跳转

在前端浏览型的系统中跳转到写请求时，会通过生成不同的域名来实现，例如在商品详情页面需要下单时，下单的 URL 域名变成 sh.buy.taobao.com。DNS 解析会自

动把这个域名解析到上海的下单系统。如果没有前端系统帮助生成不同的域名又该怎么办呢？读请求可以做 302 跳转，写请求则只能做代理转发了，如图 4.8 所示。

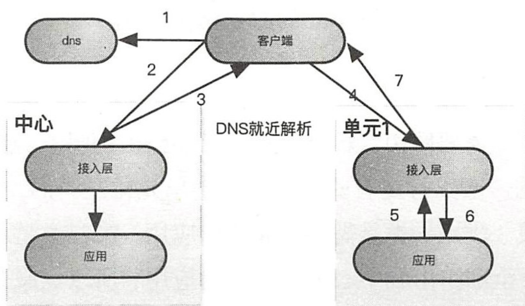


图 4.8 基于多域名跳转的方法

（2）基于 CDN 代理

另外一种就是所有需要单元化的请求都走一层代理层来转发，这个代理自然是放在 CDN 上最好，也就是所有的用户请求先就近接入到本地 CDN 上，然后在 CDN 上做代理转发。这个不需要做域名上的变化，而是在接入层上做 VIP 的路由控制。此方法如图 4.9 所示。

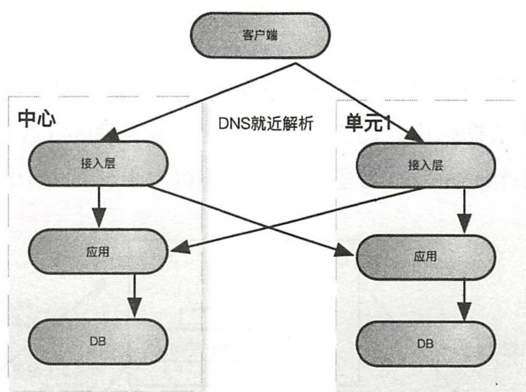


图 4.9 基于 CDN 代理的方法

4.9 服务层路由

前面的访问层更多地是前台的 Web 系统，而服务层是系统之间调用的关键环节。服务层是分布式的，包括服务的注册、发现和路由调度。而注册、发现和调度必然需

要一个“管理中心”，把不同的服务串联起来（姑且称为CS吧）。

CS需要解决三个问题：一是哪些服务需要路由；二是单元内服务如何路由；三是单元之间的服务如何路由，即跨单元的服务路由问题。

（1）哪些服务需要路由

为什么说哪些服务需要路由，难道还有不需要路由的吗？当然，要做应用的单元化必然是分步实施的，所以肯定有些应用是不走单元化逻辑的，例如那些长尾应用。所以要把应用划分成3类：一类是需要单元化的应用；第二类是不需要做单元化的应用；还有一类是需要全部路由到中心节点访问的应用，所以在服务注册时就要做好服务划分。

（2）单元内服务的路由

单元内服务的路由比较简单，可以直接根据买家ID进行路由选择：CS根据买家ID计算出目标服务的应用，再随机选择一个IP给调用方发起调用。

（3）跨单元的服务路由

跨单元的服务调用一般存在两种情况：一种是同步的RPC调用，这是点对点的调用；另外一种是异步的消息投递，一般是一对多的调用。

同步的RPC调用关键是两个机房的CS需要全量地知道两个机房对应应用的全量IP列表，否则没法做数据路由，所以这里需要做机房之间的CS的数据同步，还要根据路由模式来决定这个服务是否需要路由到对应机房上，图4.10是同步的服务路由。

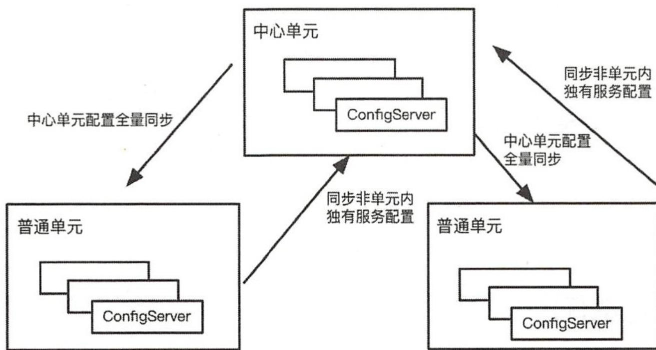


图4.10　跨单元同步服务路由

异步的消息投递，需要解决的是怎样把一个机房的消息投递到另一个机房，本质

上也是两个机房的数据复制问题，通过同步一个同步组件分别订阅需要跨单元的发送的消息 Topic。该过程如图 4.11 所示。

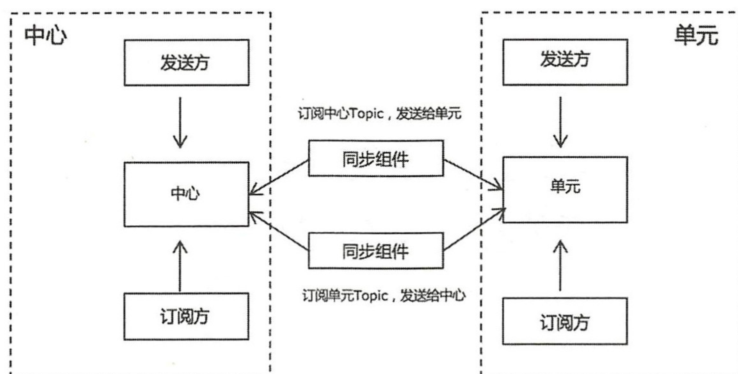


图 4.11 跨单元异步消息路由

4.10 数据层路由

数据层路由主要包括三个部分：一是对数据写入 DB 做最后一层的正确性校验；二是做 DB 之间的数据复制；三是不同机房之间 Cache 中的数据的一致性问题。下面我们分别阐述。

1. 正确性校验

正常情况下服务层如果路由正确的话，在数据层应该不会出现数据错乱的问题，但是在数据写的 DB 之前再做一次正确性校验可以多加一层保险，实现思路就是在执行 SQL 之前检查一下这个用户是否是单元用户或者中心用户，如果不匹配的话则抛异常。

2. 数据复制

DB 之间的数据复制主要根据是单元写的数据还是中心写的数据，如果是单元写的数据就需要在单元和中心之间做双向复制（目的是为了保证在做切换的时候，其他单元也有该用户的数据）；如果是中心写的数据，一般只要做中心向单元的数据复制就可以了，如图 4.12 所示。

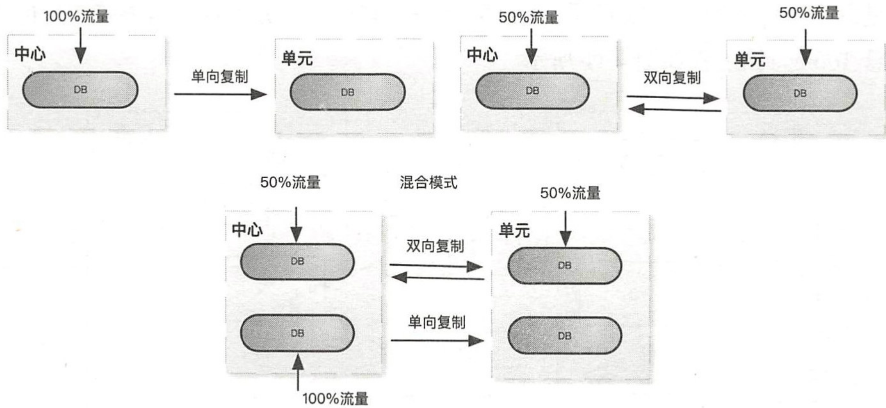


图 4.12 单元间的数据复制

数据的多地复制一般存在如图 4.12 中所示的几种情况。(1)中心同步到单元。数据单向复制，只有在中心机房部署的业务数据需要同步到各个单元中；(2)中心和单元相互同步。需要多个备份的数据如订单数据可以做中心和单元的相互同步；(3)混合模式。不同库可以做混合的同步。

如图 4.13 所示，中心跨单元读写，例如公司运营或系统修改单元机房的用户订单数据；单元跨单元写，例如跨区域接单有可能出现乘客和司机不在一个单元，这时会出现跨单元数据写问题。

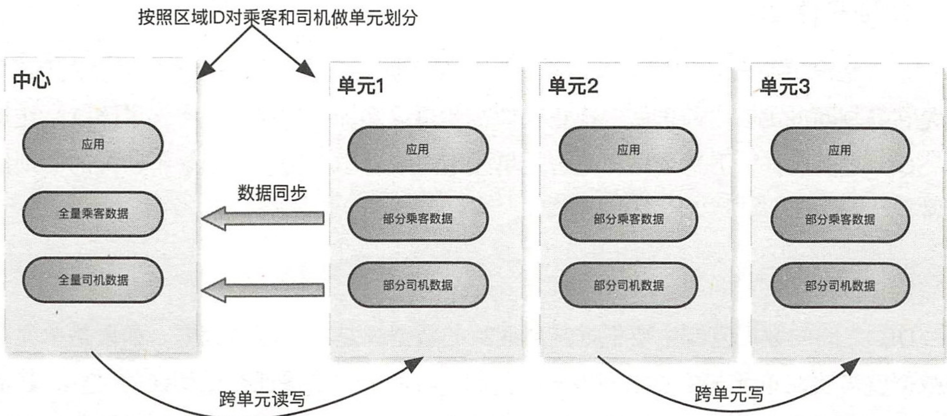


图 4.13 跨单元数据架构

3. Cache 数据一致性

Cache 中数据一致性问题是个比较普遍的问题，如图 4.14 所示。

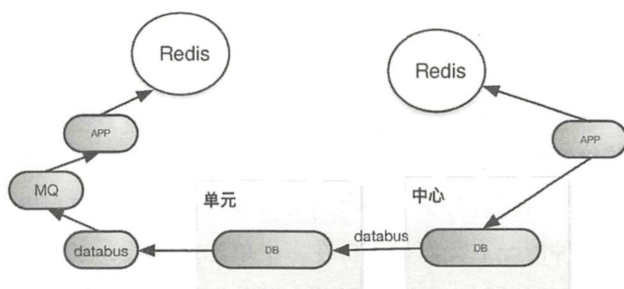


图 4.14 跨单元 Cache 数据一致性

当中心的 DB 数据变更后会失效缓存，理论上单元的缓存也应该失效，这里采用异步消息的方式失效单元的缓存，同样为了保证缓存的命中率，单元也可以通过监听 DB 的数据变更来填充数据，APP 异步双 put 也是一个可以增加命中率的备选选项。

4.11 Sequence ID 的冲突问题

要做全球数据的复制与同步就绕不开 Sequence ID 的问题。比如美国卖家发布的商品所生成的商品 ID 与中国卖家发布的商品 ID 在数据库表的 Sequence 不能冲突，否则最后在做数据复制时就会出问题，但是如何保证这两个地方的 Sequence 不冲突呢？目前这种跨区域 Sequence 的问题一般有以下几种解决思路。

第一，基于一个全局统一的 Sequence 生成器，由这个统一的生成器来分配每个库的 Sequence 分段，至于这个统一的生成器具体是怎么设计成高性能和高可用的，则有很多方法。但是这种思路的最大缺点就是“统一”，这给维护带来很大的问题。

第二，提前预设分段，比如 A 库用奇数、B 库用偶数；或者基于每个数字做 Hash；还比如美国用 0~10 亿的字段，中国用 10~20 亿的字段等，总之就是提前给每个库设置一个规则。这种思路的最大缺点规则是比较死，更改起来不方便而且无法一直执行。

第三，设置起始值加步长的方式，并且基于数据库表做更新。比如有中美欧三地的三个数据库 A、B、C，每个数据库设置的起始值分别是 0、1000、2000，步长都是 1000，把起始值写入数据库的 Sequence 表中，每次重新生成 Sequence 时就加 1000，即每个数据库 ID 可以表示成 ID=（ID 起始值+1000）×数据库数，假如 B 库用尽后，三个库的起始值分别是 0、4000、2000……，并以此类推，那么每个库用的 Sequence ID 都不会冲突。这种方式灵活性比较好，就算增加数据库也不需要调整规则，不需要统一的控制中心，所以不存在维护问题，是比较理想的方式。

4.12 异地多活

目前我们说的这套单元化思路是按照“一个中心、多个单元”的思路来建设的，单元是按照买家维度来划分的，也就是把全网的买家划分在不同的单元，由于这种思路没有解决事务问题，所以涉及多个买家的交叉数据就只能在中心机房操作，例如库存等卖家的数据，这种情况下我们很容易在多个单元之间做买家数据的容灾切换。那么，如果中心机房出现问题怎么办呢？目前有两种解决方法，一种是对中心机房做一个冷备机房；第二种是在现有基础上再按照卖家维度做一次切分，把卖家数据也划分到不同的单元机房，但是这样的后果是会使网站架构更加复杂。

当某个单元机房发生故障时，需要把这个单元的用户数据切换到中心或者其他单元，切换过程中最重要的是要保证数据的正确性，而保证数据正确性最关键的一步是数据写的一致性，也就是用户原先在 A 单元写改到 B 单元写。由于单元之间的数据是需要同步的，所以切换的时候要保证该用户需要同步的数据已经完成，否则就会出现脏数据。此外，用户的路由规则也需要调整，由于路由规则的生效也需要时间，因此切换必须按照一定的步骤来实现，我们分述如下。

（1）禁写要切换的用户请求。在服务层、数据层和消息的层面都会丢弃该请求并报异常。

（2）送消息的路由规则。消息的路由规则需要判断流量是从中心切往单元还是从单元切往中心。如果是从中心切往单元则先推送单元的消息服务端，反之则先推送中心的消息端，以保证消息不丢失。

（3）推送默认的路由规则。将新的路由规则推送上去，加载完成服务层中间件以及接入层如 Nginx 等依赖系统。

（4）关闭之前设定的用户禁写规则。

- 用户按照新的路由规则进行路由；
- 命中用户（需要且流量的用户，比如 buyerId%10）停写中心库停写；
- 等待数据完全同步到单元；
- 开启单元到中心的回流；
- 开启单元写。

4.13 多语言问题

多语言问题主要包含三方面内容：一是系统中的文案多语言问题，即我们程序模板中的文案在不同的时候显示不同的语言，如商品的价格提升文案；二是用户提交的数据的翻译问题，如针对用户上传的商品描述信息，我们需要做翻译，这时就要用到翻译引擎；三是要在数据模型层解决多货币、多价格、尺码等需求，还会涉及业务逻辑的差异问题。

1. 多语言文案的解决方案

文案中多语言问题的解决方案目前已经比较成熟，即系统中涉及语言相关的文案统一用标签来代替，这些标签所对应的语言的解析设置在另外一个系统中，在部署或者系统启动时通过推送或者拉取的方案把标签替换成相应的语言即可，如图 4.16 所示。这里比较麻烦的是 JS 中的文案，因为它是一个静态文件，解决方法一是把 JS 中的文案用标签代替，在真正渲染时发送一个异步请求，请求所有标签所对应的语言并替换；二是把整个 JS 文件替换成对应的语言，不同的语言用不同的文件。

2. 多语言的存储

针对多语言的存储问题一般有以下两种解决思路。

一是将多语言版本仅存储在 Cache 中。把经过翻译的多语言版本存储在一个 Cache 系统中，当用户访问时，再从 Cache 中拉取对应的语言数据并展现。如商品数据（item 对象）会有多种语言，把它翻译成多种语言存储在 Cache 中，如果商品数据有更新，再令此 Cache 失效。

二是在模型层解决，即通过设计合理的数据结构解决数据的多语言存储问题，如商品的数据结构可以设计成图 4.15 的样式。

把商品数据中有多语言属性的字段独立出来并进行扩展存储，再通过产品 ID 做统一标识，在不同的语言区域选择不同的商品 ID。请注意，这个转换工作可能是在原始商品发布阶段实现的，用转换工具将原始商品数据发布成多种不同语言的商品数据并存储。

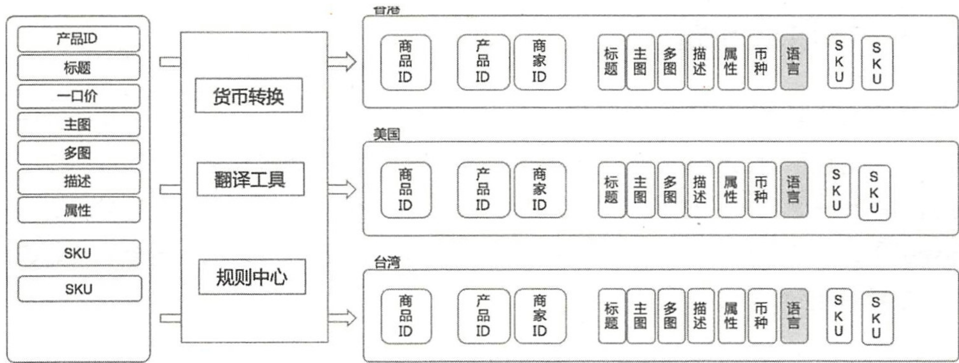


图 4.15　多语言的商品存储结构

3．实时翻译引擎

实时翻译引擎一般也有两种使用方式：一种是在浏览器端加载页面时，直接调用开源的在线翻译工具如 Google 翻译，将页面直接翻译成对应的语言；另一种是在实时搜索服务中，当用户搜索某种关键词时，需要把用户输入的语言实时翻译成后端检索服务中对应的索引的语言才能匹配。在多语言环境中，我们不可能将所有的数据均按语种分别建立一份全量索引。

4.14　多时区问题

多时区问题是全球部署中最基本的问题，涉及很多环节。有很多需要依赖时间才能执行的任务，如发货周期、定时优惠活动等。另外，在多机房的运维中，时区不统一会带来很多麻烦，如跨区域的数据复制就需要检查数据的写入时间顺序。

1．基本概念

多时区问题涉及如下概念。

UTC 时间：世界标准时间，与时区无关，以格林尼治时间 1970-01-01 00：00：00 开始计算的绝对时间；

时区：零时区格林尼治时间 24 个时区，两个时区之间递增或递减一个小时。如中国属于东 8 区（GMT+8），美国属于西 5 区至西 10 区。分别是东部时间、中部时间、山地时间、太平洋时间、阿拉斯加时间和夏威夷时间；

美国的夏令时：夏季把时间提前一小时，秋季再调回去；判断标准是当前地区是

否执行夏令时并且当前本地时间是否处于夏令时执行期间；美国的夏令时实行得比较乱，由各州各县自行决定。各地如加拿大、墨西哥、新西兰、欧洲执行夏令时的时间都不一样。

以上不同时间的换算关系如下。

- 标准时间=UTC 时间或者可直接换算成 UTC 时间；
- UTC 时间=本地时间–时区偏移量–是否夏令时；
- 本地时间=UTC 时间+时区偏移量+是否夏令时；
- 时区偏移量=地区所在的时区和零时区的时间差；
- 夏令时=地区+本地当前时间+夏令时执行期间。

2. 解决思路

时区问题的解决思路，要从数据的维护和数据的复制角度出发。首先，对运维来说，最好全球部署的系统都基于同一个时区，时间的基准都是统一的，对机器时间的设置也简单，运维很方便；如果系统部署在不同的国家不同的时区，那么一旦某处出错，基于机器时间的程序都会出错；针对数据的全球复制，最好也要统一时区。假如把数据从美国复制回中国，如果 DB 层的数据时间不一致，在复制过程中就需要做数据转换，这对数据复制工具而言是一个比较复杂的操作，毕竟数据复制工具是不应该关注业务层数据逻辑的。综上，解决时区问题的整体思路如下。

- 服务器时间、数据存储以及服务层在全球部署时统一时区，不做时区相关的转换；
- 时区相关的转换在表现层和控制层做，都依赖于统一的时区转换工具；
- 系统需要感知访问者的时区，可以通过用户 IP 或者用户自己选择和设置的时区实现。

3. 多时区方案选择

在方案选择上，可以分成多种场景来分析。

- 针对没有历史包袱、全新构建的系统，数据存储层可以使用与时区无关的 UTC 时间；
- 在有历史包袱的情况下，要考虑与存量的数据中的时区做兼容，可以使用以存量数据为主的统一时区；
- 只有本地部署，但需要做全球的数据复制。在这种情况下，每个地区的系统也

是独立维护的，可以使用本地时区。

下面分别介绍这三种场景。

方案一，使用 UTC 时间

（1）数据库中使用 long 型存储 UTC 时间；程序中使用 long 型 UTC 时间进行计算；

（2）服务层、存储层直接使用 long 型计算；展示层根据业务要求对时间做本地化转换；控制层对时间参数进行处理，将其转换成 UTC 时间；

（3）为控制层提供本地时间计算服务，对于下游系统，直接传递 long 型表示时间；对于上游系统，将标准时间转换成 long 型处理。

方案一的优点在于服务层、存储层均不需要时区转换，只有展示层需要进行时区转换。它的缺点在于需要修改服务，将原来的 date 型转成 long 型；需要修改数据库，将原来的 date 型数据转成 long 型；需要修改服务的上下游服务，将原来的 date 型转成 long 型；需要修改展示层，将 long 型转成用户需要的本地时间。

方案二，使用本地时间

（1）各服务器、数据库、应用程序使用统一的本地时间；

（2）服务层、存储层继续保持原有的 date 型；展示层不对时间做本地化处理，展示服务器的本地时间；

（3）尽量不做全球化数据同步，同步时尽量使用 long 型。

方案二的优点在于表示层、服务层、存储层均不需要做时区转换。它的缺点在于表示层展示的时间为统一服务器时间，在跨时区时需要用户自行计算本地时间；在数据同步时需要对 date 型数据做数据转换。

方案三，使用同一时区

（1）全平台使用统一时区的时间；

（2）服务层、存储层不对时间进行换算；展示层根据业务要求对时间进行本地化换算；控制层处理时间参数，将它转换成统一时区时间；为控制层提供本地时间计算服务。

方案三的优点在于服务层、存储层不需要修改，数据同步时不需要关心时区问题。它的缺点在于需要修改表示层，本地时间计算服务会增加；需要感知用户所在地区。

这三种方案各有千秋，在不同的场景下可以选择不同的时区方案，表 4-1 是三种方案的对比。

表 4-1　三种方案的对比

应用范围	UTC 时间	本地时间	统一时区时间
系统时间	本地时间	本地时间	东 8 区时间
DB 时间类型	long	Date	Date
表示层	换算	不变	换算
控制层	换算	不变	换算
服务层	不变	不变	不变
存储层	不变	不变	不变
数据同步	不变	换算	不变
应用通信	转换	不变	不变
地区计算服务	需要	不需要	需要
时间显示	用户时间	服务器时间	用户时间
现有系统修改量	大(各层、上下游)	小	中(表示、控制)
适合的业务	全球业务	本地业务	全球业务

4.15　全球数据同步与数据路由

如果把全球的站点看成一个统一的整体，理想状态就是每个站点在不同的国家和地区服务本地的用户。然而，要实现全球的买和卖，有几个关键数据必须要打通：会员数据、商品数据、商家数据和、交易数据。要打通这些数据就必须要做全球数据的同步和复制；为了保证数据的一致性，就需要对数据做数据路由，以保证数据在一个地方单写，要遵循的原则是就近读写。

1．数据复制

数据复制的策略有两种：一种是异步通过消息发送；一种是同步通过数据管道复制。两种方式各有优缺点，有各自的应用场景。

（1）异步通过消息发送数据的方式。有些中间件产品可以解析数据库的 binlog，监控数据库表的数据变化并将变化的数据通过消息流发送出去，由应用层决定如何处理；应用层可以完整地拿到整行的数据变更信息如变化前的数据、变化后的数据，属

于哪一种变更操作等，也可以通过程序修改这些数据，灵活性非常高。但是在性能上会有一些损失，比较适合数据量不是太大但是对数据需要做定制的场景。

（2）通过数据管道复制的方式。另外一个开源产品 DRC 工具提供了直接在数据库层复制数据的方式，直接将一个库的数据远程对等复制到另外一个库，可以实现简单数据过滤，如可以决定某一行数据是否复制到远端。

为了保证数据的安全性，数据的复制需要通过专网进行，这也可以保证网路带宽和数据的时效性，中美的数据同步一般会有两根专线，数据延时一般在200毫秒左右。

2. 各国对数据的保护政策

系统部署到国外后，数据并不是可以随意复制回本地的：有些国家的法规政策对此有严格的限制。像俄罗斯就对本国公民的隐私数据有严格的保护，限制本国的数据外流，因此，如果在俄罗斯建站，就无法再把数据复制回本地，欧盟对数据外流也有严格限制。

除此以外，软件版权也存在政策风险，像已有的各种各样的开源协议如 BSD、Apache Licence、GPL、LGPL、MIT 等；而每个国家对软件版权的重视程度也不一样，如果不了解就会有法律风险。

4.16 通用版与定制版的选择

通用版和定制版的区别在于是否通过一套代码来解决全球部署中遇到的所有差异性问题，这些差异包罗万象，具体包括以下内容。

（1）由国家的法律法规限制导致的问题。例如在有些国家不能售卖某些商品，有些国家的用户私人数据不能外流，有些国家对软件版权的管理非常严格，等等；

（2）每个国家的业务规则存在差异。例如剃须刀是属于 3C 类目还是属于生活用品类目；度量衡单位的差异会导致鞋子、衣服的尺码不一样；

（3）系统的分层和模块化做得不够会导致系统的依赖非常复杂，很难把全部的系统都完整地部署到所有国家，因此，能否无须修改大量代码即可方便地去掉一些不需要的依赖，成为一项挑战；

（4）部署和维护成本的问题，对于一个业务量很小的网站，是否需要部署那么庞大的产品集群？这不仅对部署需要的机器成本、而且对软件的运维成本也是一种浪费，

那么是否需要把分得很细的系统再合并成一个系统？显然这对节省成本会更有帮助。

（5）依赖的中间件是开源的还是自己开发的？因为二者应用的环境不一样，所以维护成本也不一样。

到底使用通用版还是定制版，要根据具体的环境来选择，但是笔者认为有一些基本的原则可供参考，首先就是要分清楚每个部署的版本是否最终都由同一个团队来维护，即目标是否都是一个人维护一套代码？这一点非常关键。

- 如果是，肯定最终要选择通用版，但是可以分步骤来实现。为了快速上线，先期可以做个定制版，逐渐把需要定制的逻辑点抽离出来封装成独立的二方库，或者做好逻辑分层，把差异化的业务逻辑做成服务便于维护，再慢慢实现代码的兼容，最终实现一套代码全球部署。
- 如果不是，那么是否完全是同一套代码并不那么重要了，但是在数据层打通这点比较重要，也就是如果有必要的话，底限是数据能够互通，即每个国家的存储模型要统一，这是很有必要的。

4.17 全球化部署中遇到的坑

全球化部署是非常复杂的一项工程，有很多环节都可能出现差错，所以要谨慎地设计很多预案，下面就是需要注意的、可能会遇到的一些坑，避免掉坑最重要的原则就是要保持数据的准确性和一致性。

（1）脏数据

脏数据主要是由于在单元内写了不应该在本单元写的数据引起的。原因有可能是路由规则错误或者一致性出现了问题。

（2）路由规则不一致

在 Nginx、服务框架、消息框架、DBProxy 中，路由规则都是关键的路由决策点，如果几款产品中的路由规则不一致，就很容易出现脏数据。

（3）路由规则生效延迟

由于路由规则分布在多个地方，因此路由规则的变更肯定会产生生效的延迟问题，这就需要从机制上避免路由规则延迟带来的风险。

（4）服务接口改造遗漏

需要修改服务提供端以确保兼容性，否则，就有可能直接在本单元写了不应该写的用户数据。

单元内部署的应用也会提供相关的写服务接口，目前的方案是在发布服务时做一个标识来避免生效，但这样做仍然有可能会有遗漏，导致脏数据产生。

（5）应用层绕过路由规则直接写数据库

这个情况容易出现在一些后台的 MIS 系统中。由于是内部系统，所以在实际的操作中没有按照规范实现，这是比较容易忽略的，需要警惕。

（6）MySQL 同步数据错误

跨机房数据同步时如出现错误，很容易就会产生脏数据。

（7）数据同步故障

数据同步出现故障的主要影响是读数据的不一致，可以将回滚单元中的读写数据源回主库来解决。

（8）中心-单元网络故障

当单元机房-中心机房网络故障时，会导致单元机房和中心机房数据不一致，也会影响功能。目前来看，如网络故障时间较长，只能通过切换取消单元机房，相当于切换为同城模式。

4.18 总结

全球部署越来越成为现在互联网公司发展的一项必备的要求，本章介绍的全球部署不仅仅是把系统简单地部署到海外去，而且还要求部署在海外的各系统之间也要成为一个整体。它们之间是相互联系的，数据是打通的、服务是共用的、用户是共享的。所以，本章介绍的全球部署是建立在系统进行单元化的基础上的，也就是将整个系统的数据按照某种维度进行单元的划分，并将这些单元数据分布在全球多个数据中心，各系统之间的数据是共享和互通的，基于这些数据所沟通的服务也是互通的——这种部署才是真正的全球部署。基于此背景本章介绍了很多实践中遇到的问题，并提供了一些实际的经验供大家参考。

5 应用程序优化：代码级优化

从本章开始，介绍流量从 1 亿到 50 亿的增长过程中遇到的各种性能优化问题，首先从最基本的代码级优化开始介绍（本章的部分思想借鉴了本书的推荐人之一小邪的成果）。

代码级优化也就是应用服务端的优化，主要是提升系统的单机 QPS（Query Per Second），或者减少 RT（Response Time）提升用户体验。如何提升应用的 QPS 呢？由于不同的开发语言有各自的特性（如 Java 字符编码很耗时、序列化、并发锁冲突等），我们只有非常熟悉它们在某些场景下的使用规则，才能在合适的场景下使用合适的技术，提升应用的 QPS。

5.1 优化思路

做优化首先要知道从哪里入手，也就是要知道系统的瓶颈在哪里。一个请求会消耗很多资源：CPU、内存、网络、磁盘等，这些资源中总会有一个先到达瓶颈，只有优化最先到达瓶颈的资源才会产生实际效果。如图 5.1 所示，最短的那块板决定了木桶（系统）的容量。

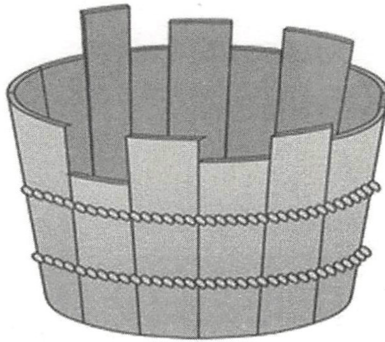


图 5.1　发现资源瓶颈

如何发现瓶颈？从代码执行部分来看，就是看哪里消耗了最多的 CPU 时间，从消耗 CPU 最多的地方做优化效果更明显。所以首先要掌握一些工具，例如压测工具、发现代码热点的工具。

1．压测工具

针对 Java 有两个经典的代码热点分析工具：JProfiler 和 Yourkit，如图 5.2 所示。

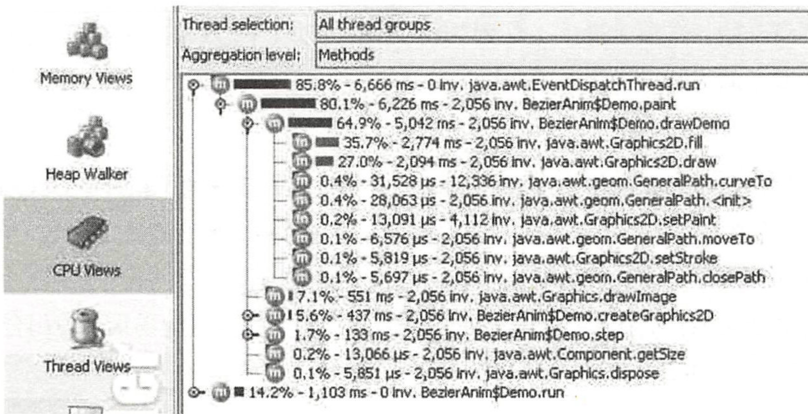


图 5.2　分析代码热点的工具

从中可以看出每种方法下的 CPU 消耗，优化占 CPU 消耗大头的方法，效果肯定最好，当然这并不等于消耗 CPU 的代码就一定会有优化空间，还需要观察 CPU 具体在做什么，例如在做正则运算的话能不能把运算结果缓存起来。

JProfiler 不仅可以观察到 CPU 消耗，还能看到内存、线程等使用情况，如图 5.3 所示。

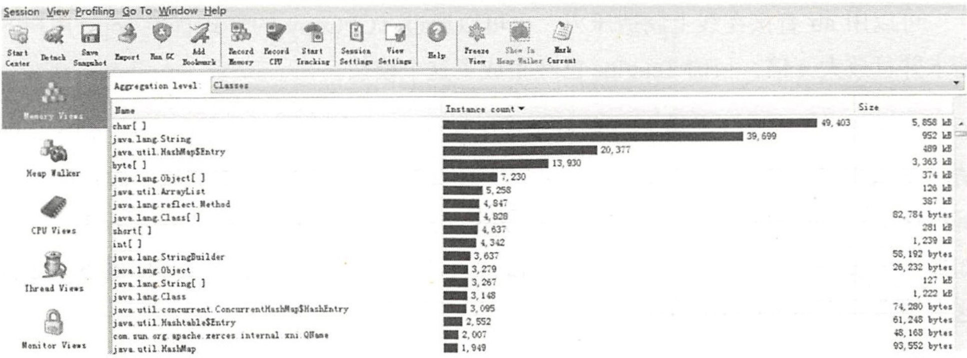


图 5.3　JProfiler 分析内存的占用

关于这些工具的具体使用方法可以参考相应的技术文档，这里不再赘述，要注意的是目前这些压测工具都是以线下为主，直接能够在线上生产环境中操作的很少。在线上环境要能产生压测流量得用线上代码分析工具，这样才能更真实和方便。

线上真实环节的压力测试有很多种，对大部分读系统来说可以通过 Apache 的 ab 压测工具来完成，使用非常方便，如命令$ab -c $c -n $n $url，如图 5.4 所示。

```
Server Software:        Apache/2.2.16
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        20 bytes

Concurrency Level:      200
Time taken for tests:   20.409 seconds
Complete requests:      20000
Failed requests:        0
Write errors:           0
Total transferred:      5900000 bytes
HTML transferred:       400000 bytes
Requests per second:    979.98 [#/sec] (mean)
Time per request:       204.087 [ms] (mean)
Time per request:       1.020 [ms] (mean, across all concurrent requests)
Transfer rate:          282.32 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   69 115.0     69    3058
Processing:    16  134 264.2    100    7935
Waiting:       11  121 260.6     93    7921
Total:         85  203 294.3    157    7936

Percentage of the requests served within a certain time (ms)
  50%    157
  66%    220
  75%    236
  80%    250
  90%    296
  95%    354
  98%    484
  99%    582
 100%   7936 (longest request)
```

图 5.4　Apache ab 压测工具

可以用 ab 直接在线上制造压力，也可以通过 TCPCopy 的方式直接 copy 用户的请求并且放大 *n* 倍，目前 Tengine 已经集成了这个模块。

上面介绍的这些工具都是在服务端产生压测流量，还可以模拟用户发出请求（即请求是从用户的浏览器中发出的），具体实现就是在请求的页面中嵌入一段代码，当解析这个页面时额外再发一个请求到服务端，这样相当于用户帮我们压测系统了，比如加入这段 JS 代码 new Image().src = 'http://xxx/item.htm?id=${itemId}', 你也可以放大这个请求，比如 for 循环 10 次。

但是要注意以上这些请求都是针对读请求的系统，因为不会产生额外的脏数据，如果要进行线上真实的写系统的压测怎么办呢？这就要运用全链路压测技术了。全链路压测技术会模拟线上真实的用户请求，但是把产生的压测数据写到影子表里，关于全链路压测技术我们会在稳定性一章中详细介绍。

2. 发现瓶颈

前面介绍了很多产生压测流量的方法，主要还是想找出系统的能力极限，发现系统的瓶颈。一般来说，系统的瓶颈大多是由 CPU 资源造成的，但实际上还有很多因素导致系统没有发挥出最佳性能。如何判断 CPU 是否达到瓶颈？指标之一是观察压测的时候 CPU 的使用率是否达到了 95%以上，如果 CPU 并没有压满 QPS 就上不去了，那么 CPU 肯定还没到达瓶颈，此时需要仔细分析到底是哪些问题导致 CPU 没有跑满。以下介绍几种排查思路。

- 针对 Java，可以 Jstack 一下，看看当前的 Java 线程都在干啥，是否都在执行代码，抑或是在等待其他远程调用的返回，或者有线程锁的等待。
- 如果你的压测请求不是在本机，要注意一下当前请求的 TCP 连接数，可以通过命令：netstat -n | awk '/^tcp/ {++state[$NF]} END {for(key in state) print key,"\t",state[key]}'来判断，如果有很多 TCP 链路在 waiting（等待），那么可以设置 tcp_tw_reuse 来复用请求。
- 还要检查网卡是否达到了瓶颈，千兆网卡只有 128MB/s 的数据量。可以通过多种工具查看网卡的请求和出现的数据量，如 vmstat 等。
- I/O 也可能成为瓶颈，如果系统有大量的读写磁盘的操作，如写日志、读文件等，可以通过 iostat 命令查看磁盘的使用率，而 iotop 可以检查出哪个进程正在大量进行 I/O 操作。

总之，只有先找到系统的瓶颈才能制定更好的优化方案。除去这些因素之外，在后面的小节中，我们还会详细介绍线程和响应时间的关系模型、并发数设置等和性能优化相关的内容。

5.2 影响性能的关键因素

所谓提升性能，通常意义上就是提升系统的 QPS，即提升系统的吞吐量。要提升系统的 QPS，首先要了解 QPS 与 RT 的关系。

- 对于单线程：QPS=1000 / RT
- 对于多线程：QPS=（1000 / RT）× 线程数量

对单线程而言，QPS 与 RT 是反比关系，可以简单把 RT 理解为 CPU 消耗时间，但是这里也包括了 CPU 的等待时间如 I/O，所以出现了多线程的情况。但是，实际情况远比这个复杂，例如线程之间锁的竞争，所以这仅仅是最理想的情况。

1. QPS 与线程的关系

根据单线程的 QPS 公式，支持的线程数越多 QPS 越高。对线程而言似乎是线程数越多 QPS 越高，但这只在一定范围内适用，当线程数到了一定数量之后，QPS 会持平不再上升，并且随着线程数量的增加，QPS 开始略有下降，同时 RT 开始持续上升。

影响线程数量的两个主要因素是 CPU 数量和线程等待时间。CPU 数量越多线程数量可以越多，但是线程本身也会消耗系统资源，会增加线程的切换成本，从而导致线程得不到 CPU 的执行时间。

对于大部分的 Web 系统，RT 一般由 CPU 执行时间和线程等待时间（远程 RPC 调用、I/O 等待、sleep、wait 等）组成。

在做实际的优化时，要检查系统中最耗时的那部分代码，试图减少 CPU 的消耗，尤其当系统中存在大量的远程调用时，在远程等待环节会消耗很多时间，那么减少总的 CPU 消耗时间对提升 QPS 会有多大的效果？经过实际测试发现，CPU 实际执行时间和线程等待时间这两个因素对 QPS 的影响并不一样：减少 CPU 的执行时间对 QPS 有实质的提升，减少线程的等待时间对 QPS 提升不明显。所以在高 I/O 的系统中，QPS 和 RT 的关系并不是线性的。

对一个高 I/O 的系统而言，假设 CPU 时间 10 毫秒 + I/O 等待时间 40 毫秒 =总时间 50 毫秒，如果 CPU 时间被优化到了 5 毫秒，实际的总时间是 45 毫秒，RT 从 50 毫秒减少到了 45 毫秒，从总体时间上看变化不大，单线程的 QPS 从 20 提升到 22 也并不明显，但是实际上由于 CPU 时间减少了一半，QPS 也几乎可以提升一倍。

2. 设置最佳线程数

所谓最佳线程数是指消耗完服务器的瓶颈资源的一个临界线程数量。到达最佳线程数之前，增加线程会提升 QPS，但是超过这个数之后，线程数量继续递增，则 QPS 不变而 RT 变长，持续递增线程数量则 QPS 开始下降。

这里说明一下，并发用户数和线程数不是一一对应的关系，对于 Java Web 系统而言，在并发请求没有达到 Java Web 服务器设置的并发线程数之前，一个用户请求对应一个服务端的处理线程，但是超过设置的并发线程数继续增加并发用户请求，则线程数将不会改变，用户的请求将被等待。

最佳线程数的计算公式如下。

最佳线程数=[(线程等待时间+线程 CPU 时间)/线程 CPU 时间]×CPU 数量

如何才能获取最佳线程数？可以通过以下两个方法。

第一，单用户压测，查看 CPU 的消耗的百分比，然后直接乘以该百分比，再进行压测；

第二，通过慢慢增加并发请求来进行性能压测，通过观察压测结果判断是否达到服务器的资源瓶颈，以获得最佳线程数量。

并发请求同样要受 RT 的影响，线程多时间消耗长，由于资源的竞争，导致线程等待的时间也上升了。超过最佳线程数之后，线程数量翻倍，RT 翻倍 QPS 不变。例如，假设最佳线程数为 10，并发 100 个请求，那么第一个用例有 90 个线程在瞬间是等待的，如图 5.5 所示。

图 5.5 并发执行

因为并发数量是 10，所以第 1~10 个请求，分别只需要 2 毫秒（实际是 1.91 毫秒，

为了计算方便用 2 毫秒），同时有 90 个线程在等待，第 11~20 个请求，因为已经等待了 2 毫秒，所以等到完成需要 4 毫秒……依此类推，第 90~100 个线程需要 100 毫秒，这个结果和实际结果的 89 毫秒还是比较接近的。

3. 最佳内存设置

处理并发请求时要考虑内存的影响，即每个请求需要消耗多少内存，并发请求数乘以每个请求消耗的内存就是总内存的大小，如果设置的内存小于这个数，就会导致频繁的 Full GC。

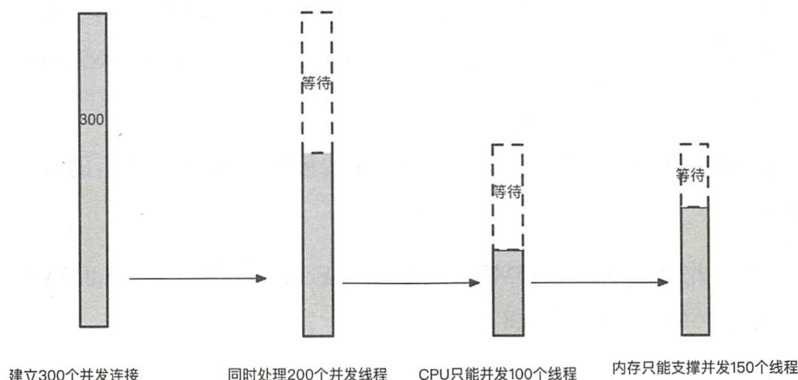


图 5.6　资源限制

如图 5.6 所示，假设 Tomcat 线程数为 300 个，CPU 能支持的最佳线程数只有 100 个左右，内存限制的线程数为 150 个(否则 Full GC 频繁)，因为 CPU 满负荷时只能处理 100 个线程，势必会导致另外 100 个线程在等待状态，线程等待内存始终被占用，导致内存 Full GC 频繁。

如何判断是否达到内存瓶颈？压测时要观察 Old 区内存增长是否正常，如图 5.7 所示。

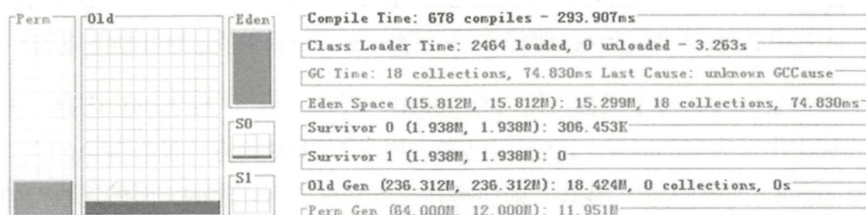


图 5.7　内存资源使用示例

所有对象的创建都是在 Eden 区完成的，当 Eden 区满了，再创建对象，会因为申

请不到空间而触发 minorGC，进行 young(Eden+1 Survivor)区的垃圾回收。在 minorGC 时，Eden 区不能被回收的对象会被放到空的 Survivor（Eden 肯定会被清空），另一个 Survivor 里不能被 GC 回收的对象也会被放入这个 Survivor，始终保证一个 Survivor 是空的。如果 Survivor 满了，则这些对象会被 copy 到 Old 区，或者 Survivor 并没有满，但是有些对象已经足够 Old，也被放入 Old 区。Old 区被放满后再进行完整的垃圾回收。

下面通过例子了解如何设置大小合适的内存。

假设 JVM 配置参数：2 个并发线程，每个线程需要 10MB 的内存，每个线程执行 10 毫秒释放内存（具体配置：JAVA_OPTS=-Xms256m -Xmx256m -Xss128k -XX:MaxTenuringThreshold=15）

（1）A 线程在 Eden 申请了 15MB 内存，A 线程需要 10 毫秒左右的时间才能释放 15MB 内存；

（2）B 线程此时也申请了 15MB 内存，因为 Eden 区总共只有 15.812MB 内存，除去被 A 申请的 15MB，只剩下 0.812MB；

（3）由于 Eden 区内存不够，触发 minorGC，由于此时 A 线程并没有执行完成，那个 15MB 内存不能被回收。所以将 A 线程的 15MB 内存 copy 到其中一个 Survivor 区；

（4）由于 Survivor 区只有 1.938MB 空间，放不下 A 线程的 15MB 内存，则又将 A 线程的 15MB 内存直接拷贝到 Old 区；

（5）拷贝完成之后，Eden 区又有了 15.812MB 内存空间；

（6）B 线程的 15MB 内存在 Eden 区完成了分配。

这种情况下会导致频繁的 Full GC。假设 Eden 区的空间有 30MB，是否内存分配正常？或者说满足(并发线程数×线程占用内存) < Eden 区内存分配就是健康的？

10 个并发线程，每个线程占用内存 10MB，假设 Eden 区有 10MB 内存，由于第 11 个线程来申请内存的时候，Eden 区已经占用了 100MB 内存，没有多余的空间，则发生 MinorGC；但是先前的线程，即便是并发也是有先后顺序的，而最近执行的线程很可能还没有结束，则 100MB 内存并不能被全部释放，可能有 10~*N*MB 是需要被放到 Survivor 区的。如果这个时候 Survivor 区的空间太小，则会直接被放到 Old 区，同

样也会发生频繁 Full GC，所以要保证 Survivor 的合适大小就尽量让对象不要进入 Old 区。这个结论可通过设置 A、B 线程一次申请 1MB 内存空间和 survivor 空间是 1.5MB 的大小以及 A、B 线程一次申请 2MB 的内存来验证，申请 1MB 的内存没有发生频繁 GC，而一次申请 2MB 时则发生频繁 Full GC。

通过前面的测试，可以发现 Eden 区太小，则导致 minorGC 频繁；Survivor 区太小，则非常容易导致对象被直接 copy 到 Old 区（Survivor 区只存放 Eden 区无法被回收的对象，并不能直接说明这些对象相对较老，很多刚刚创建的对象也可能被直接 copy 进来）；Young 区太大，则容易导致一次 minorGC 耗时，一般建议 Young 区为整个堆的 1/4。

Sun 推荐的配置为 Survivor 区一般设置为 Young 区的 1/8，如果 Young 区为 500MB，则 Survivor 区可以设置为 60MB。例如一个线程占用的内存为 2MB，则 50MB 的 Survivor 区支持 25 个并发线程是比较理想的。

所以从上面的结论来看，每个请求需要申请的内存大小非常关键，那么如何计算出每个请求平均占用的内存大小呢？理论上可以通过 Eden /（QPS × minorGC 的平均间隔时间(s)）来计算。

如何减少每次请求中占用的内存大小呢？

- 尽量减少线程请求生命周期里的对象数量；
- 对象创建到可回收区的时间要尽可能短，例如不要在非常耗时的操作前面创建一个大对象，而尽量在真正使用这个对象时再创建，对象使用完成后也要尽量置空。

4．如何提升性能

我们设计压测案例来验证几个因素是如何影响性能的，通常情况下 Web 系统的执行过程如图 5.8 所示。

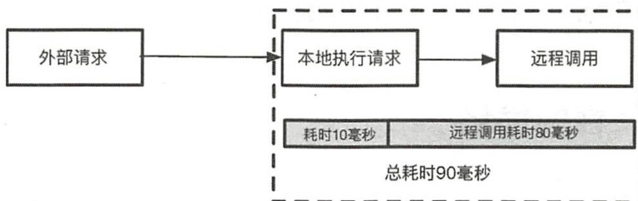


图 5.8　本地执行与远程调用的耗时占比

在压测一个请求时，请求中有一部分是本地处理，会消耗 CPU；另一部分是远程请求，会需要一个等待时间，这是我们 Web 服务端最典型的一个场景。

假设基准性能如下。

CPU 时间为 10 毫秒、I/O 时间为 80 毫秒，一共是 90 毫秒。ab 压测结果为 QPS=100，RT = 90 毫秒，服务器 CPU 85%，最佳线程数 20。

现在想提升 QPS，我们分别从下面两个部分的优化来看看是否能提升性能。

第一，将 I/O 时间从 80 毫秒改成 40 毫秒。如图 5.9 所示，实际压测结果：QPS=176，RT = 50 毫秒，服务器 CPU 85%，提升不明显。

图 5.9　提升远程调用的耗时

第二，CPU 时间从 10 毫秒改成 5 毫秒，如图 5.10 所示。实际压测结果：QPS=200，RT = 85 毫秒，服务器 CPU 85%，提升明显。

图 5.10　提升本地执行的 CPU 时间

所以单纯减少响应时间，并不一定能有效地提升 QPS，结论如下。

- 如果要提升服务器端的响应时间 RT，采用减少 I/O 的时间能达到最佳效果，比如合并多个 I/O 请求；
- 如果要提升 QPS，采用优化 CPU 的时间能达到最佳效果。

5.3　Java 特性的优化

Java 语言本身也有很多优化技巧可以使用，比较常用的如使用 StringBuilder，优化自定义 hasCode() equals() toString()方法，优先使用原始数据类型，不在循环中使用

try…catch，copy 时使用 System.arraycopy()命令，除了这些通用技巧外，我们再介绍一些优化方法，分述如下。

（1）减少编码

Java 的编码运行比较慢是 Java 的一大硬伤，在很多场景下只要涉及字符串的操作（如输入输出操作、I/O 操作）都比较耗 CPU 资源，不管它是磁盘 I/O 还是网络 I/O，因为都需要将字符转换成字节，而这个转换必须编码。因此，减少编码就可以大大提升性能。那么如何才能减少编码呢？例如，在网页输出时可以直接进行流输出，即用 resp.getOutputStream()写数据，把一些静态的数据提前转化成 byte，等到真正往外写的时候再直接用 OutputStream()写，就可以减少静态数据的编码转换。此外，很多存储系统直接存储成字节也是为了减少字符的编码。

（2）使用局部变量

在很多情况下，程序员为了更方便地一次处理一个请求，一般会创建一堆对象，再把这些对象一直传递下去，直到请求执行完成，才会销毁对象。这种操作会导致对象长时间不能回收，降低内存的使用率，所以要鼓励更多地使用局部变量。例如，调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中，这样速度较快；其他变量像静态变量、实例变量等，都在堆中创建速度较慢。栈中创建的变量，随着方法的运行结束，这些内容就没有了也不需要额外的垃圾回收。

（3）减少方法调用

经常看到一些代码如 ob.getXXX()，它的作用只不过是获取某个变量对应的值，但是在一段代码中却需要多次调用——这种操作完全没有必要，可以把这个方法调用的返回结果用一个局部变量保存下来，然后直接用这个局部变量就可以了，这样能减少方法调用的次数（因为每一次方法调用 JVM 都要创建方法栈）。

其他一些经验分列如下。

- 把对象作为 HashMap 的 key；
- web.xml 配置版本信息可以减少启动时 annotation 的扫描时间；
- Logger 创建没有使用 static 修饰符导致线程阻塞；
- 少用 Thread.getStackTrace()；
- 正则运算尽量 Cache。

5.4　减少并发冲突

并发冲突往往会导致程序的性能上不去，成为性能瓶颈。容易出现并发的地方一般都会用锁，判断是否出现锁冲突要看 CPU 没用满的情况下 QPS 是否还能上去，可以通过 jstack 检查线程是否都在 block 状态。

图 5.11 展示了一个并发冲突锁的样例。

图 5.11　一个并发冲突锁示例

Java 中 Class.forName 容易导致线程 Block，所以要给 Class.forName 加缓存；另外创建 Logger 没有使用 static 修饰符也容易导致线程阻塞。修改后如图 5.12 所示。

```java
@Override
protected Class<?> resolveClass(ObjectStreamClass desc)
        throws IOException, ClassNotFoundException {
    if (classLoader == null) {
        classLoader = TairObjectInputStream.class.getClassLoader();
    }

    try {
        String name = desc.getName();
        ConcurrentHashMap<String, Class> cache = Loader2Cache.get(classLoader);
        if (cache == null) {
            cache = new ConcurrentHashMap<String, Class>();
            ConcurrentHashMap<String, Class> old = Loader2Cache.putIfAbsent(classLoader, cache);
            if (old != null) {
                cache = old;
            }
        }
        Class clazz = cache.get(name);
        if (clazz == null) {
            clazz = Class.forName(name, false, classLoader);
            cache.putIfAbsent(name, clazz);
        }
        return clazz;
    } catch (ClassNotFoundException e) {
        return super.resolveClass(desc);
    }
}
```

图 5.12　Class.forName 加 Cache 示例

仅供非商业用途或交流学习使用

5.5 减少序列化

序列化也是 Java 性能的一大天敌，减少 Java 中的序列化操作也能大大提升性能，序列化往往是和编码同时发生的，所以减少序列化也就减少了编码。

序列化大部分是在 RPC 调用中发生，因此避免或者减少 RPC 的调用就可以减少序列化，当然当前的序列化协议也已经做了很多优化提升性能。笔者曾经尝试一种新的技术“合并部署”，可以减少不同应用之间的 RPC 调用从而减少序列化的消耗。所谓“合并部署”就是把原本在不同机器上的两个不同应用合并部署到一台机器上，当然不仅仅是部署在一台机器上，还要能在同一个 Tomcat 容器中、不能走本机的 Socket，这样才能避免序列化的产生。我们会在架构优化一章详细介绍本技术。

5.6 减少字符到字节的转换

我们先看下面这段代码。

```
private static String content = "…100k…";
protected doGet(…){
     response.getWrite().print(content);
}
private static String content = "…100k…";
Private static byte[]  bytes = content.getBytes("GBK");
protected doGet(…){
     response.getOutputStream().write(bytes);}
```

压测这段代码后，发现性能几乎相差一倍左右？为什么会有这么大的差异，原因在于这种转换需要字符编码。编码的过程如下。

（1）通过 java.lang StringCoding 进行 encode；

（2）找到指定的编码 Charset，默认 ISO8859-1；

（3）利用 CharsetEncoder 的实现类，对每个 Char 转成 byte。

每个字符的编码都需要查表，而这种操作非常耗 CPU 资源，所以减少字符到字节的转换或者相反、减少字符编码会非常有效。

另外一个典型的场景就是读取静态文件。一般情况下我们会把静态文件读取到内存，以减少每次从磁盘读文件的消耗，但是还可以进一步做的就是把这个静态文件直

仅供非商业用途或交流学习使用

接转化成字节缓存，也就是把 String 经过编码转换成 byte[]数组，当然我们输出的时候要用 stream 流输出而不是用 out.print()。

在 Java 中涉及字符到字节转换的场景有很多，但是大多伴随着 I/O 同时发生，不管是磁盘 I/O 还是网络 I/O,一般有 I/O 的场景同时也伴随着序列化，序列化时又会有很多的编码转换，所以要尽量减少这种编码转换和序列化。

在 Web 系统中将解释执行转变成编译执行对性能提升也非常有益，在《深入分析 Java Web 技术内幕（修订版）》关于 Velocity 优化的一章中对此有详细介绍，有兴趣的读者可以参阅。

减少页面输出的大小也可以减少转换。因为输出页面时一般要做 Gzip 压缩，这个过程非常耗 CPU。删掉多余的空格、换行以及循环中合并相同的数据也都可以减少转换。

5.7 使用长连接

在内部调用中，会有一些 HTTP 请求，大部分情况以短连接为主，例如一个典型的失效场景，一般如 Varnish、Squid 等会发 PURGE 请求，将这个 HTTP 的短连接改成长连接会显著提升性能。此外，如果是 Java 客户端，建议不要使用开源的 httpclient 来组装 HTTP 请求,而是直接封装一个符合 HTTP 协议的 TCP 包来发送 PURGE 消息，如下所示。

```
    String line = "PURGE/health.htm HTTP/1.1\r\nHost:item.xxxx.com\
r\n\r\n\r\n";
    c.getChannelFuture().getChannel().write(line);
```

5.8 总结

性能优化的过程总结如下。

- 发现短板；
- 减少数据大小；
- 数据分级；
- 减少中间环节、增加预处理。

仅供非商业用途或交流学习使用

下面我们分别阐述。

（1）发现短板主要是考虑在以下场景中会受到一些限制：光速（光速：C = 30 万千米/秒；光纤中 V = C/1.5=20 万千米/秒，即数据传输是有物理距离的限制的）、网速（全国平均上网带宽达到 4.19Mbit/秒，相当于 536KB/秒的网速）、网络结构（交换机/网卡）、TCP/IP、虚拟机（内存/CPU/IO…）和应用本身的一些瓶颈等。

（2）减少数据的大小。有两个地方特别影响性能，一是服务端在处理数据时不可避免地存在字符到字节的相互转化，二是 HTTP 请求时要做 Gzip 压缩，网络传输的耗时，这些都和数据大小密切相关。我们可以从以下方面着手减少数据大小。

- HTML：减少 HTML 文件的大小可以明显提升 QPS，减少空白行、换行、tab，做 Gzip 压缩；
- 图片：图片分辨率、质量、锐化、编码格式；
- JSON：减少 key 的命名长度；
- JSON 结构：减少重复的 key；
- Java 对象：避免生成大对象；减少对象生存时间；
- 请求数：Combo JS\CSS。

（3）数据分级就是要保证首屏为先、重要信息为先，次要信息采用异步加载的方式，提升用户获取数据的体验。

（4）减少中间环节，减少字符到字节的转换，将变的转换为不变的；增加预处理就是去掉不需要的操作。

要做好优化还需要做好应用基线，大概包括以下内容。

- 性能基线：何时性能突然下降；
- 成本基线：去年双 11 用了多少台机器；
- 链路基线：系统发生了哪些变化；
- 持续关注系统的性能：代码级（提升编码质量）；业务（改掉不合理的调用）；架构和链路级（改进架构）；
- 用更通用和批量的方式解决问题：整合系统之间的调研链路（合并部署）；提升整体机器使用率（弹性部署）。

仅供非商业用途或交流学习使用

6 应用架构探索：合并部署

第 5 章介绍了应用本身的代码优化，包括如何写出高质量的代码、减少 CPU 的执行时间和提升单机的执行效率等。本章从更大的范围来探讨系统的优化。在分布式系统中，全部请求不可能都在同一个系统中完成，因此减少系统之间的调用消耗同样可以提升整体性能。

一个大型网站可能有成千上万个系统，一般应用之间的依赖度非常高，每次用户请求会涉及非常多的应用之间的调用，而 Java 中 RPC（Remote Procedure Call，远程过程调用）的调用必须要做序列化与反序列化，大量的 RPC 调用会有大量序列化的消耗，而序列化与反序列化会非常耗 CPU，所以如果能减少 RPC 中的消耗，就可以减少 CPU 消耗、减少调用延时。

6.1 什么是架构

架构这个词被用得有点多，架构本身也有多种解读，很难有一个统一的标准，图 6.1 是笔者对架构的理解。

仅供非商业用途或交流学习使用

图 6.1　一种架构

（1）业务架构抽象。业务架构是面向用户的，主要考虑给用户带来什么价值、什么体验，如帮助用户完成一系列需求的集合。

（2）应用架构。应用架构是面向技术人员的，如何用更低的成本更快更好地实现用户的需求，是应用架构要解决的问题。它需要划分用户界面的功能集、具体业务逻辑的分层、数据的交互逻辑、服务调用的逻辑，还要考虑各种性能和扩展性等问题。

（3）平台（服务）架构。当应用架构复杂到一定程度后，需要单独抽出一层平台架构治理层（如中台），解决业务和应用的扩展性问题，以实现业务模型抽象、灵活扩展服务能力水平、灵活定制和组装应用等功能。

（4）技术架构。技术架构是面向微观问题的，目标是用简单技术解决复杂问题，在稳定性、扩展性和性能之间取得平衡。比如采用什么开发语言、使用哪种开发框架、如何定义标准化接口、选择合理的中间件产品和各种监控工具……这些都属于技术架构的内容。

（5）数据架构。数据架构涉及数据模型的构建和数据库选型，数据结构的设计、数据的读写比例以及性能评估，数据的一致性、安全性以及原子性等内容。

（6）网络（部署）架构。网络（部署）架构和应用层的关系比较紧密，因为应用的容量及高可用性与网络（部署）架构息息相关。它要解决的是网络路由、数据通信协议的优化、带宽网卡的分配、机房容灾以及网络设备的高可用等方面的问题。

（7）组织架构。组织架构是角色的分工协作以及 KPI 的设置，主要关注人员的执行力和效率。

本章重点关注应用层的架构，包括在应用架构层面如何用更低的成本更快更好地

仅供非商业用途或交流学习使用

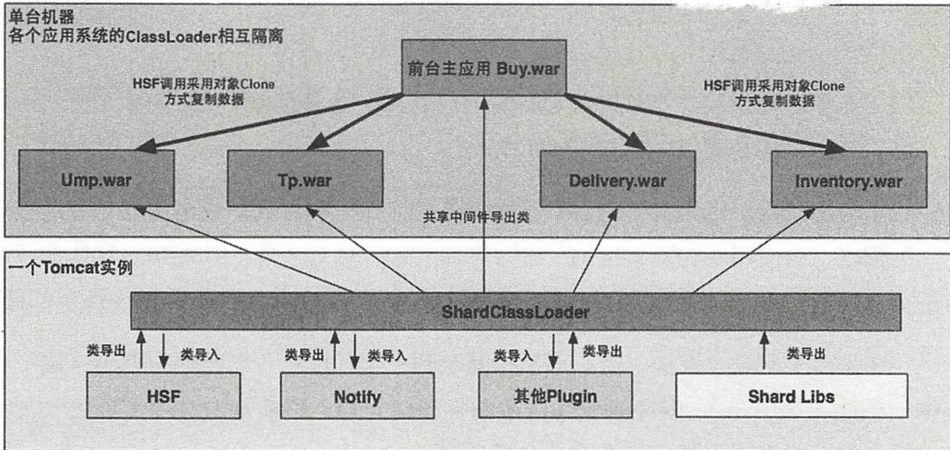


图 6.2 合并部署

为什么要做合并部署？它和硬件层面的多内核架构、操作系统层面的虚拟机技术、容器层面的虚拟化技术 Docker、JVM 的多租户技术以及 Tomcat 本身的多 war 包的部署有何区别呢?

其实它们都可以被称为虚拟化技术，只是在不同的层次上实现而已。比如硬件 CPU 上的多核是把我们的指令执行周期虚拟化了，看起来好像是由多个 CPU 来执行同一个应用程序；操作系统层面的虚拟化有很多，像 KVM、Linux 的 LXC 容器虚拟化等，可以把一台性能强大的物理机拆成多台，每个应用系统好像各自跑在一台独立的机器上；JVM 层的多租户技术，在统一 JVM 进程中可以跑多个应用系统，每个应用系统的生命周期可以独立，包括它们的 CPU、内存等都可以相互隔离。我们这里介绍的合并部署可以归结为应用容器层的虚拟化技术，不同层面虚拟化技术的资源利用率如图 6.3 所示。

仅供非商业用途或交流学习使用

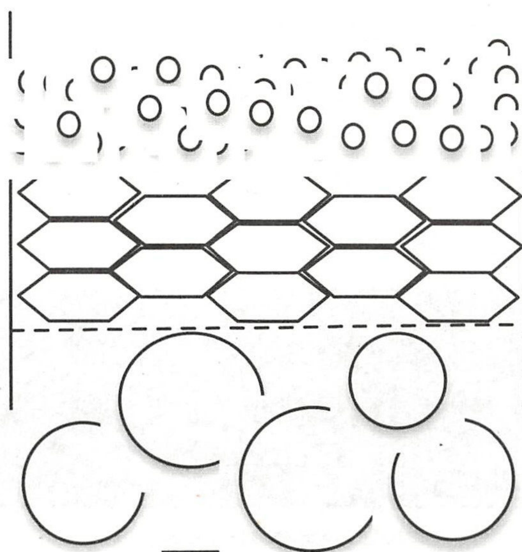


图 6.3 不同虚拟化技术的资源利用率

所有的虚拟化技术都是解决哪些内容独享、哪些内容共享的问题，需要尽可能地细分有限的资源，尽可能发挥出最大的价值，用大白话来说就是尽可能物尽其用、避免浪费。

KVM 可以把一台物理机虚拟成多台虚拟机，这样带来的好处是：

- 充分共享 CPU、内存和磁盘等硬件资源。我们目前的 Java 应用系统一般只需要 4GB 左右的内存，但是一台物理机一般都有 32GB 或者 48GB 的内存，所以一个应用独享一台物理机本身就是一种浪费，磁盘等其他资源也类似。
- 可以让一个应用分布在多台物理机器上。假定 2 台物理机就可以支撑一个应用系统的请求量，那么其中一台物理机发生故障就会损失 50%的容量，风险较高；如果采用虚拟机的部署，可以把 2 台物理机的规模拆成（1 虚 8 的模式）16 台机器，这样一个应用系统就可以部署在 16 台不同的机器上，大大降低了风险。

当然虚拟机太多也会带来负面影响，如增加了机器的维护成本，成倍增加了运维工程师的工作量；使应用到数据库之间产生了更多的连接；给带有心跳健康检查的系统和群发消息的应用带来负担，等等。

再比如 JVM 层的虚拟化技术——多租户技术，可以让多个应用跑在同一个 JVM 进程中，这比 T4 的虚拟化程度更加激进，有可能让上百个应用（长尾应用）跑在同一台机器上。应用之间的共享甚至可以是相同的 jar 包或者 Class。所以虚拟化技术越

仅供非商业用途或交流学习使用

往上层越可以精细地控制，也更有优化空间。

合并部署正是利用了应用系统之间的关联性。如果两个应用系统之间可能会加载同一份缓存数据，那么只要共享一份就可以了；如果两个应用系统之间会相互调用，那么可以直接走 Java 本地调用，不必再通过网络调用。

合并部署的目录结构如图 6.4 所示。

```
|-- target/            webapp, 容器 部署目录
|    |-- ${APP_Union}.tgz     合并部署 应用 tgz 包, 由 appctl.sh 脚本自动解压
|    |-- ${APP_Union}    解压后的 合并部署应用 ".war" 目录
|        |-- META-INF/
|            `-- application.xml 合并部署描述符, 具体定义请参阅1.1
|        |-- APP_A.war/            合并部署应用A(目录)
|        |-- APP_B.war//            合并部署应用B(目录)
|        `-- APP_C.war//            合并部署应用C(目录)
|    |-- 容器.tgz         打包好的 容器 tgz包
|    `-- 容器.sar/        解压后的容器
|        `-- sharedlib/        存放中间件依赖的jar包, 以及所有合并部署应用共享的jar包
```

图 6.4　合并部署目录结构示例

合并部署需要受到以下约束。

- 必须做成通用 Web 系统都能使用的方式，不做过多的绑定；
- 被合并的系统应该少而精，不吃大锅饭；
- 对开发完全透明；
- 不破坏现有系统架构；
- 在运维层面，要充分利用现有基础设施。

6.3　能解决什么问题

合并部署到底能带来些什么好处呢？理论上它可以提升性能并协助提升效率。

1. 性能的提升

（1）减少网络 I/O 开销。将多个相关联的应用部署在同一个 Tomcat 容器里，很显然可以省去它们之间的网络调用步骤，相应的，也就不存在内部之间的网络调用延时了。根据测算，减少一次远程调用请求至少可以节省 1 毫秒的网络延时，会极大提升整个请求的 RT。在笔者负责的商品详情系统中，RT 可以平均提升 30%左右。

（2）减少应用调用之间的序列化与反序列化。Java 要进行网络 I/O 就必须把 Java 对象序列化成 Java 的原生数据类型（String、byte、int、long 等），所以必须有一个序列化的过程。虽然目前已经有很多序列化方案明显好于 Java 默认的序列化方式，但是

仅供非商业用途或交流学习使用

和 Java 的本地调用相比，它们仍然非常耗时。可以说，序列化与反序列化是 Java 天生的弱点，也是 Java 本身的瓶颈。合并部署恰好可以避免序列化与反序列化的开销。在实际的测试数据中，商品详情系统中的两个系统合并部署可以提升 40%左右的性能，效果还是非常明显的。测试数据如图 6.5 所示。

	PC24K	PC9K	无线 8K	无线 4K
基准环境	135	256	156	303
合并部署环境	800	1540	1356	1882
折算	200	385	339	470.5
比例	48.14%	50.39%	117.31%	55.28%

图 6.5　合并部署测试数据对比

2. 协作效率的提升

这里所说的协作效率的提升是通过分离应用的开发态和应用的运行态来实现的。应用的开发态是指程序员在同一个代码版本上开发和维护的状态，多个程序员开发和维护同一份代码会有协作上的效率问题，人越多效率越低。网站的早期一般就是由上百人维护同一个大系统，导致开发越来越难，合并代码冲突非常厉害，打包发布也存在瓶颈，所以我们后来把它拆成多个子系统，变成一个分布式的应用集群。这本质上还是解决协作效率的问题，副作用是增加了系统调用的复杂度。

为什么合并部署可以提升协作效率？原因如下。

（1）允许系统进行更多的拆分，可以按照功能模块划分出更多子系统，由不同的团队维护不同的子系统，避免开发态下的冲突问题，代码的打包和部署也更加方便和独立。

（2）解决了不同团队之间的 Owner 问题。研发人员一般都有很强的 Owner 心态，希望自己写的代码能够自己掌控。大公司中经常发生抢业务抢系统的事情，此模式一定程度上解决了这类问题。

（3）虽然对系统进行了更多的细化和拆分，但是在运行态避免了传统分布式系统的麻烦，即在线上运行时系统之间的调用仍然走本地，不会增加系统调用的复杂度，也不会影响性能。线上的系统运维和独立的系统运维基本没有太多差别。

目前这种模式还不能解决每个应用运行生命周期的独立性，也就是合并部署的每个应用仍然不能独立部署和重启，后面我们会结合 JVM 层的多租户特性来优化，实现应用的独立部署以及应用的 CPU 之间、内存之间的独立使用。

仅供非商业用途或交流学习使用

6.4 如何解决

合并部署是新一代、创新型的系统架构方案，这一节我们重点介绍它的实现。

1. ClassLoader 的加载体系

合并部署的关键点分述如下。

（1）运用应用和 Tomcat 中间的容器（P 容器）实现了 ClassLoader 的隔离，每个 Web App 对应一个 ClassLoader；

（2）每个应用系统共享的部分抽出来放到 SharedClassLoader 中（像应用中用到的 Spring、Log4j 以及中间件等）；

（3）应用之间的分布式调用框架（HSF）也必须导出到共享的 SharedClassLoader 中，因为只有共享的 SharedClassLoader 才能同时看到每个应用的 Class，如图 6.6 所示。

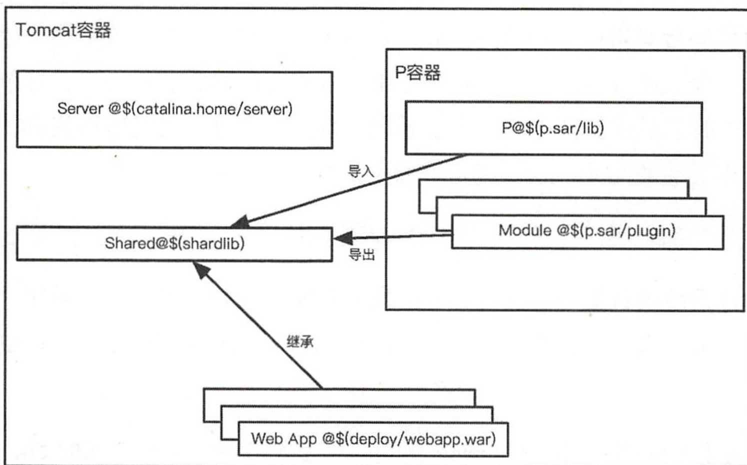


图 6.6 ClassLoader 加载体系

各 ClassLoader 的作用如下。

- Server 类加载器，在 Server 类加载器的 Classpath 下放置了 PClient，该 Client 负责启动 P 容器，类型导出以及 Web App 生命周期的通知等，放在这里是为了保证双亲委派模型中 parent 类加载足够干净（应用加载的顺序是 Common →Shared→Web App）；
- Shared 类加载器，是 P 容器进行类型导出的依据，也是 P 容器放置导出的类型

的地点，由于它的层级高于 Web App，因此可以提供给所有的 Web App 共享；

- P 类加载器，用于隔离 P 和应用之间的关系，运行 p.sar/plugins 中的所有插件；
- Module 类加载器，可以理解为中间件的服务，它们被多个应用所共享，每个 Module 类加载器隔离了相互之间的依赖，也隔离了与应用之间的依赖；
- Web App 类加载器，每个 Web App 之间是隔离的，它们共享 Shared 类加载器中的内容。

2. 对象在不同 ClassLoader 间复制

把原本应用之间的远程调用改成本地调用是如何实现的呢？前面提到过，不同应用虽然在同一个 Tomcat 容器中，但是它们之间的 ClassLoader 是隔离的，理论上是不能转成 Java 方法中的本地调用的，因为两个对象的 ClassLoader 不同，所以必须对对象进行 ClassLoader 之间的复制才能完成两个应用之间的数据交换。该过程如图 6.7 所示。

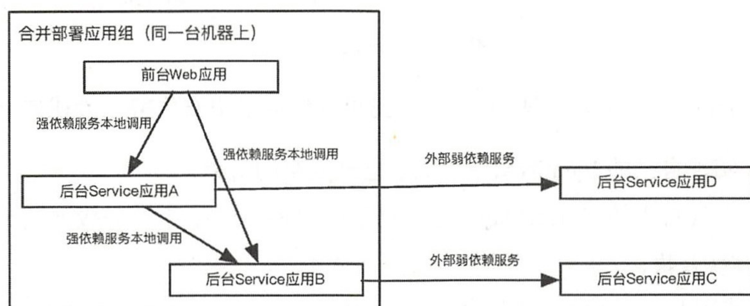


图 6.7　合并部署的应用调用关系

关于对象的深拷贝，对比了开源的几个工具 Cloning 和 Kryo，并对 Cloning 做了多处优化：

- 基于目标 ClassLoader 替换深拷贝时的 context ClassLoader；
- 除了容器类型以外，忽略 20 多种无须复制的类型；
- IdentityHashMap 存储已复制的对象，class 的查询，缓存反射字段查询；
- 用 Unsafe 创建对象，基于字段复制，目标字段如果不存在就不复制，兼容 class 的版本变化；
- 优化后的克隆工具是在支付宝克隆工具上消除了复制过程 Static 字段的判断。在字段缓存时提前判断，对于静态字段比较多的类，克隆时可以减少不必要的遍历和判断；优化数组复制，不可变对象通过 java.lang.System#arraycopy 复制，

这种方法在数组的对象复制上，提升比较明显；

- Unsafe 代替反射调用，在优化后的克隆工具基础上，通过 Unsafe 代替对 Field 的反射调用。

优化后的效果如图 6.8 所示。

图 6.8　Cloning 优化

除了 Cloning 深拷贝外，使用 Kryo 深拷贝在性能上更有优势，分述如下。

- Kryo 实现自己的 IdentityMap，记录已经复制的对象；
- Kryo 实现自己的 ObjectMap，记录对象的 class 类型，带有一级 class cache，优化 class 查找；
- 优先使用 ReflectASM 创建对象；
- transient 默认拷贝；
- 深度优先递归；
- 字段使用 unsafe().putObject()拷贝，集合类用集合类的 add()方法添加元素。

读者可能会问，这种对象的深拷贝比对象的序列化到底好多少？图 6.9 是基于 Cloning 的对象拷贝与 Hessian 序列化的对比测试。

从图 6.9 可以看出，对象拷贝比序列化快 3 倍左右，所以理论上通过对象的深拷贝走合并部署的本地调用，比通过序列化的要快很多。

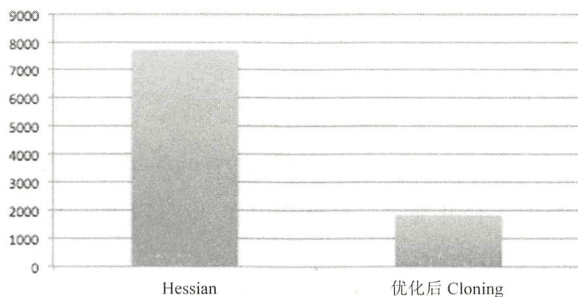


图 6.9　Hessian 与 Cloning 的对比测试

3. 日志问题

日志问题是合并部署项目中的难题，为什么呢？大家在做 Java 开发时经常会遇到 Log 类冲突，原因在于 Log 几乎是每个代码模块都必须要用到的工具：应用系统要依赖 Log、容器要依赖 Log、其他开源框架都要用到 Log，但它们所依赖的 Log 的版本不一定一致，这就会引发各种各样的问题。

我们举个例子来说明。项目中有 A 和 B 两个应用，A 和 B 依赖的 Log 的版本不一样，连接 A、B 应用的容器需要的 Log 也不一样（如果仅仅是 jar 版本不一致问题倒还好），关键在于多应用部署中 A、B 的 Log 要尽量分开，不能只打在一个 Log 文件里。例如，每个应用都要依赖分布式数据层（开源的 TDDL）这个中间件，如果将 TDDL 放到 SharedClassLoader 中，就必须要解决以下问题。

（1）每个应用初始化 TDDL 时必须要创建自己的 Logger。理论上容器启动时，可以通过切换 TCCL（Thread Context ClassLoader）了解当前在初始化哪个应用，可以根据应用设置不同的 Logger，但是 Logger 的初始化一般都在第一次输出日志中完成，所以这一点很难做到。

（2）有很多应用都是直接使用 rootLogger 的，不同应用设置的 rootLogger 会冲突，因此只能有一个 rootLogger，这就会导致 TDDL 的日志只能打在某一个应用的 Log 里。

（3）将 Log 的部分接口类提到 SharedClassLoader 中也是一个办法，例如把 Append 接口提出来，但是通过分析接口类的依赖同样比较多，所以很难做到隔离。

从上面的分析来看，日志问题的确很难处理，如果要做到完全的隔离，必须对中间件做大量 TCCL 的切换，改动量比较大。最终的妥协方案是将部分中间件放到

SharedClassLoader 中，让它们与应用一起放在不同的 ClassLoader 中。

6.5 取得的效果

从上线的几个应用来看，合并部署对性能提升的效果比较明显。我们把商品详情系统和一个优惠系统做了合并部署，性能提升了 40%以上，不仅如此，系统调用的 RT 也下降明显，如图 6.10 所示。

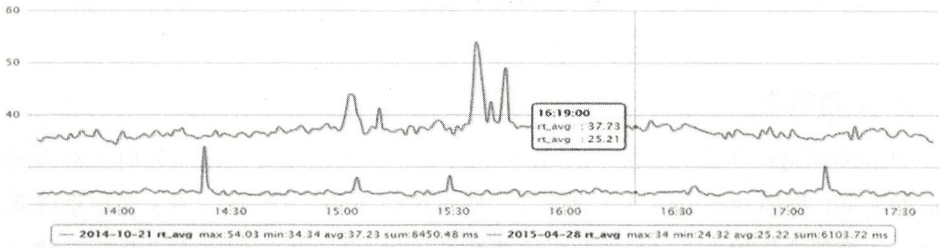


图 6.10　合并部署的效果

RT 平均下降 3~5 毫秒左右，当然不同的系统合并部署的效果会不一样，另外一个合并部署的应用的 RT 下降效果更加明显，差不多有 10 毫秒左右。除了 RT 下降外，波动也会减少，为什么呢？因为被合并部署的应用的主要依赖也被部署在一起了，这样它们受其他应用影响的可能性就变小了。

6.6 更进一步地做多版本部署

前面介绍的合并部署已经解决了如下几个关键问题。

（1）在同一个 Tomcat 里可以同时运行多个应用，多个应用之间不相关的业务代码是相互隔离的，但是又相互调用（像中间件是共享的）；

（2）中间件从面向单机单应用的模式转变成单机多应用的模式，也就是中间件自身能区分当前是哪个应用在执行代码；

（3）解决了基础的运维工具：单机的多应用部署、多应用代码基线的合并等，从以前的单机单应用运维改成单机多应用运维；

综上，我们很自然会想到另外一个问题：能否把单机的多应用改成单机单应用的多个版本，即把不同应用改成相同应用的多个版本。这样会有什么好处？很明显，如

果在同一个 Tomcat 里同时能跑一个应用的多个版本，那么应用就可以做到热启动了，外部流量也可以在两个版本之间随意切换了，对应用的发布和回滚、以及两个版本的 A/B Test 也非常方便了。但是要实现多版本部署也会遇到一些难题。

1．什么是多版本部署

多版本部署是在同一台虚拟机里、同一个 JVM 实例里、同一个 Tomcat 实例里同时跑一个应用的多个版本，从此应用的发布不再需要重启 JVM，回滚只需要切换流量而不需要替换代码，如图 6.11 所示。

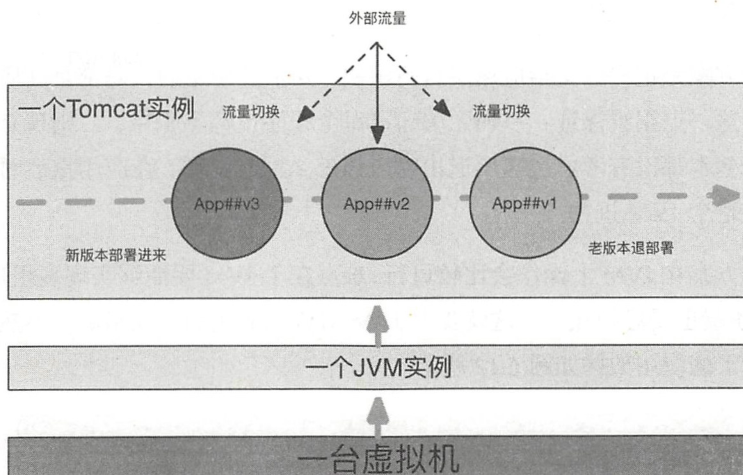


图 6.11　多版本部署示例

线上可以始终保持两个版本，当然也可以是一个版本。当新版本部署进来时，更老的一个版本便退出部署，这样线上始终可以提供对外服务。

2．解决应用的热启动问题

多版本部署可以大大提升发布效率，具体操作时可以分离应用的代码替换和应用接受的外部请求流量，即让外部流量先跑在老版本的代码中，等新版本的代码起来后，再把请求流量切换到新版本中，最后再把更老的代码退出去。JVM 中可以始终保持两个版本的代码，但前提是整个 JVM 不能重启，必须在同一个 JVM 进程里做应用的热启动和退出。

如何实现代码的热替换？需要解决两个关键问题：一是执行完成的线程能正常退出；二是对象和 Class 类以及向操作系统申请的资源能被回收。

解决这两个问题有以下两种方案。

第一，由 Tomcat 容器托管应用层、中间件和各种框架的生命周期的管理。这是一种最优雅的退部署方案，即退出应用时相当于调一个 Destroy，每个线程释放控制的资源，最后再退出自己，一层一层退出最终释放整个应用。采用这种方式的前提是每个应用的代码都写得非常优雅，如果某个地方忘记实现 Destroy 方法，那就比较悲剧了。

第二，这是比较“暴力”的方式，即采用多租户的方式，应用启动时只申请某个租户的资源，当应用要退部署时，在 JVM 层直接杀掉该租户的线程，强制回收租户包括内存在内的所有资源。

以上两种方案各有优缺点。

第一种方案是把责任交给应用层自行解决，要求应用层的代码都是高质量的代码，然而实践中这一点很难保证：一则无法确定每个程序员都不犯错，二则现有的系统当初在构建时基本都没有考虑过应用退出时应该怎么回收资源，在应用重启时也基本都是直接杀掉整个 JVM 进程。

第二种方案在 JVM 上操作会比较可行，难点在于 JVM 要能够实现多租户的特性，也就是 JVM 层也要虚拟化——这就要求 JVM 对资源的控制必须精确，不然会有很多坑。多租户下的应用架构如图 6.12 所示。

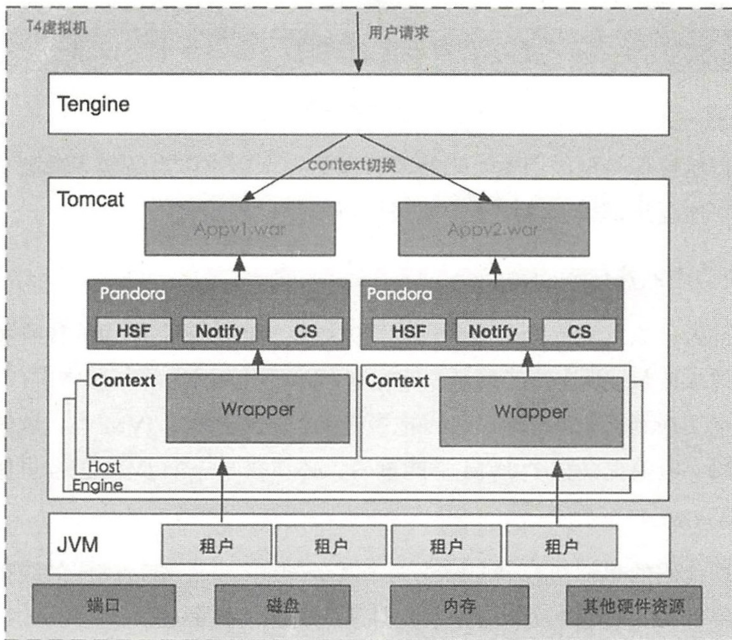


图 6.12　多版本部署应用架构

每个应用对应的 Context 对应一个租户，由租户控制整个应用的生命周期，销毁租户就回收了整个应用。

3. 多租户资源的隔离

多个应用或者一个应用的多个版本跑在同一个 JVM 里，或多或少都会令人担心它们之间会不会相互影响。例如，某个应用发生内存泄漏会不会搞挂整个 JVM，一个应用跑满了 CPU 会不会影响另外一个应用的运行……这些是最典型的资源隔离问题。

多租户技术将 Java 的堆内存划分成多个租户，在创建每个租户时可以指定使用内存的上限，当租户的内存超过上限时就会抛 OOM，但是不影响其他租户的使用。

CPU 资源可以通过 Linux 提供的 cgroup 来控制。一个是控制下限，即保证某个程序最基本的 CPU 需求；另一个是控制上限，保证某个程序不会无限制地占用 CPU。但是，这会产生另外一个问题，即 JVM FullGC 并没有做隔离，也就是当一个租户出现 GC 时无疑也会影响另外一个租户的 CPU 消耗。

4. 多版本部署带来的好处

多版本部署给应用带来很多好处，最明显的就是大大提升了应用的发布效率和回滚效率。根据笔者做过的测试，上千台机器规模的应用的完整发布能在 10 分钟完成，应用的回滚更是达到了秒级。能达到这种效果的原因在于：

第一，应用的新版本的部署是在一个新的租户中完成的，并不影响正在运行的租户，等新版本的租户初始化完成后，再将流量切换过来，而整个 JVM 一直在不间断地提供服务；

第二，应用的回滚更简单一些，如果新发布的版本有问题，只需要直接把流量切换回原来的版本，这里只有流量的切换时间，所以能在秒级完成。

这种发布模式从以前的重启 JVM 来达到替换代码，转变成启动和退出 JVM 的 Tenant 这种更轻量的方式完成应用的部署，而流量的切换也可以更方便，不需要和 JVM 的重启绑定。

6.7 关于高密度部署的思考

不管是不同应用的合并部署还是单应用的多版本部署，本质都是以在单台机器上部署更多的应用为目的的，理论上虚拟化越在上层实现，所能共享的资源就越多，但

是要同时考虑应用的维护成本和机器成本，找到平衡点。虽然我们可以利用 JVM 多租户的特性在一台物理机上部署上百个应用（当然，这些应用是长尾应用，本身的流量不大，性能不会成为瓶颈），能够节省非常可观的机器，但衍生的问题是应用耦合度变高、相互影响的因素增多、稳定性下降、维护成本增加。所以凡事都有两面性，需要权衡。如何找到平衡点呢，笔者根据自身的经验，总结出如下参考因素。

1. 是否是高内聚

和写程序一样，一个高内聚的程序健壮性会更好。能否实现高密度的部署必须考虑程序是否是高内聚的，这里的高内聚是指：

- 边界是否清晰。如应用与应用之间是否隔离（比如都是 war 包）；应用与 Tomcat 的直接交互是否标准、Tomcat 与 JVM 的直接交互是否低耦合等；
- 参与人之间的信息交流是否简单明了，沟通成本是否足够低，否则很难推广和执行。

2. 是否容易工程化

一个新技术最终能否产生价值关键在于它最终能否被工程化，不管采用哪种虚拟化方案，只有能被更好工程化的虚拟化方案才有竞争力。技术工程化需要考虑以下因素：

- 操作流程简单易学，实施门槛低；
- 可以大规模被复制，依赖少兼容性强；
- 传播方便，可以被弹性调度。

综合以上几点要素，当前比较流行的 Docker 刚好符合这些特征，它是否会成为新一代的虚拟机方案，我们拭目以待。目前也有使用 Docker 来实现多版本部署的尝试。

6.8 总结

本章介绍了一种应用架构的优化方案，它的本质是通过合并部署原本联系比较紧密的应用系统，减少它们之间的 RPC 调用，并最终提升系统的性能。所谓的合并部署并不是真正的系统合并，系统的开发态仍然是独立的，只是线上运行时部署在同一台机器上，所以不会影响开发效率，但是对运维会有一些影响。是否做合并部署需要进一步考虑和评估系统的运行时系统资源的隔离问题。

7 链路优化：大秒系统的极致优化思路

上一章介绍了服务端上系统之间的架构优化，本章介绍从用户请求开始的全链路优化案例，即用户端→中间的链路→服务端→数据库的整条链路的极致优化思路。

目前大家所熟悉的秒杀已成为淘系的重要营销品牌，其实它最早是为了解决系统压力而想出来的技术方案。最初秒杀系统是为了满足卖家在某段时间内压低价格吸引眼球的需求所设计的，原型是早期详情页的定时上架功能。但是，这种做法给详情系统带来很大压力，因此，为了隔离这种突发流量设计了秒杀系统。目前秒杀系统经过多年的发展已经比较成熟了。

本章主要介绍大秒系统以及这种典型的读数据的热点问题的解决思路和实践经验。大秒系统是在详情系统的静态化方案基础上发展而来的，关于静态化方案更完整的介绍可以参考《深入分析 Java Web 技术内幕（修订版）》中第 18 章“大浏览量系统的静态化架构设计”。

7.1 一些数据

大家还记得 2013 年的小米秒杀吗？三款小米手机各 11 万台开卖，使用的就是大秒系统。开卖仅 3 分钟，小米旗舰店就成为双 11 第一家销售额最早破亿元的旗舰店。

日志统计显示，前端系统在双 11 峰值约有 60 万次以上的 QPS，而后端 Cache 的集群峰值近 2000 万次/秒、单机也近 30 万次/秒，但是真正的写流量要小很多了，当时最高下单减库存的 TPS（Transaction Per Second，每秒事务处理量）是红米创造的，达到 1 秒 1500 个请求。

下面我们介绍秒杀系统的设计原则，让大家对此有整体的认识。

7.2 热点隔离

秒杀系统的第一个设计原则就是隔离热点数据，即禁止 1%的请求影响剩余 99%的请求。隔离之后也可以更方便地针对 1%的请求做有针对性的优化。在实践中，我们对秒杀系统做了多层次的隔离。

- 业务隔离。把秒杀做成一种营销活动，卖家要参加秒杀活动需要单独报名。从技术上来说，通过卖家报名，我们就可以把他们设置为已知热点，提前做好预热。
- 系统隔离。系统隔离更多是指运行时的隔离，即通过分组部署的方式把 1%的请求与剩下 99%的请求分开。我们针对秒杀活动还申请了单独的域名，目的也是让不同的请求落入不同的集群中。
- 数据隔离。秒杀所调用的数据大部分都是热的数据，比如会启用单独 Cache 集群或者 MySQL 数据库来释放热点数据，目的是避免 0.01%的数据影响其余 99.99%的数据。

当然，实现隔离有很多办法。

- 按照用户来区分，可以给不同的用户分配不同的 Cookie，在接入层再路由到不同的服务接口中，还可以在接入层针对 URL 的不同路径设置限流策略等；
- 在服务层调用不同的服务接口；
- 在数据层可以通过给数据打上特殊的标签来区分。

以上操作的目的都是把已经设定的热点和普通的请求区分开来。

7.3 动静分离

热点隔离以后，接下来的工作就是要对热点数据做动静分离。这也是大流量系统设计的重要原则。下面我们先介绍如何对动态系统进行静态化的架构设计。

1. 什么是静态化系统

首先我们要明白静态化系统是什么，有哪些属性？只有先了解基本属性才谈得上有目标地改造。

静态系统通常有如下特征。

- 一个页面对应的 URL 通常固定。不同的 URL 表示不同的内容，也就是通过 URL 能唯一标识一个页面。
- 页面中不包含浏览者的相关因素。页面中不能包含与浏览者相关的因素，这里所说的“不能包含”不包括 JS 动态生成的部分，也就是页面中 HTML 代码不能显式地含有浏览器相关的 DOM，如不能含有用户的姓名、身份标识以及 Cookie 相关的因素等。
- 页面中不包含时间相关的因素。页面同样不能含有时间（这里的时间不是指客户端浏览器中获取的时间，而是服务器端输出的时间）相关的因素，不能随着时间的变化导致页面中的 DOM 结构发生变化。比如在秒杀活动中，一到某个时间点，页面中的“立即购买”按钮就可以使用——这个时间点就是从服务器端获取的时点。
- 页面中不包含地域因素。这个很好理解，即从北京访问的页面要和从上海访问的页面相同。商品详情页面上的宝贝运费就是典型例子：不同地区的运费不一样。如果要做成静态化的，这个运费就不能直接反映在 HTML 代码中。
- 不能包含 Cookie 等私有数据。Cookie 实际上主要是用来标识访问者信息的工具，如果页面中包含这些私有数据，也就不可能不包含上面这些信息了。所以要满足静态化，就不能包含 Cookie 信息。

再强调一下，静态化页面不仅是传统意义上完全存储在磁盘上的 HTML 页面，它也可能是经过 Java 系统产生的页面，但是它输出的页面本身不包含上面这几类信息；页面中“不包含”是指页面的 HTML 源码中不包含，这一点务必要清楚。

2. 静态化系统能解决什么问题

静态化之前我们围绕 Java 层面做了很多优化，改进的思路也大多是尽量让应用本身更快地获取数据，更快地计算出结果，然后把结果返给用户。我们做了一个极端的测试：将系统全部的数据缓存，再直接返回所有的请求结果，在这种情况下压测 Java 系统，结果性能未能满足期望——即达到 2000 甚至上万次的 QPS——因此在 Java 系统上不可能达成目标。

据此，我们判断 Java 系统本身已经达到瓶颈，它天生就存在不擅长处理大量连接请求、每个连接消耗的内存较多和 Servlet 容器解析 HTTP 协议较慢等弱点。在这种情况下，我们必须跳出 Java 系统，也就是使请求尽量不经过 Java 系统，而在前面的 Web 服务器层就直接返回。于是，我们自然就想到了静态化的架构，静态化系统成为必然的选择。

- 系统静态化为何能达到 Java 系统无法达到的高性能呢？系统静态化的优势在于：改变了缓存方式。直接缓存 HTTP 连接而不是仅仅缓存数据，Web 代理服务器根据请求 URL 直接取出对应的 HTTP 响应头和响应体并直接返回，这个响应连 HTTP 协议都不用重新组装，同样也不一定需要解析 HTTP 请求头，所以能最快地获取数据。
- 改变了缓存的位置。不是在 Java 层面而是直接在 Web 服务器层上做缓存，屏蔽了 Java 层面的一些弱点，Web 服务器（如 Nginx、Apache、Varnish）都擅长处理大并发的静态文件请求。

3. 如何改造动态系统

如何把动态页面改造成适合缓存的静态页面呢？就是通过前面提及的去除影响因素的方法，即通过动静分离把这些因素独立出来。下面以详情系统（Detail）为例介绍如何做动静分离。

- URL 唯一化。详情系统天然地就可以做到 URL 统一化，如每个商品都用 ID 标识，那么 http://item.taobao.com/item.htm?id=xxxx 就可以作为唯一的 URL 标识。
- 分离浏览者相关的因素。浏览者相关的因素包括是否登录以及登录身份等信息，我们可以把它们单独拆分出来，通过动态请求来获取。
- 分离时间因素。通过动态请求获取服务端输出的时间。
- 异步化地域因素。以异步方式获取详情系统上与地域相关的信息。

- 去掉 Cookie。可以通过代码软件删除服务端输出页面中包含的 Cookie，如可以通过 unset req.http.cookie 命令去掉 Varnish 中的 Cookie。

分离出动态内容以后，如何组织这些内容也是非常关键的，因为页面中其他模块会用到这些动态内容（比如判断该用户是否登录等）。通过把这些信息 JSON 化，前端可以很方便地获取它们。

知道了分离哪些内容，又知道怎么组织它们，现在的问题就是如何获取它们并把它们和静态文件组装在一起。我们可以通过两种方式获取动态内容：ESI（Edge Side Includes）和 CSI（Client Side Includes）。

- ESI。即在 Web 代理服务器上做动态内容请求，并将请求插入静态页面中，当用户拿到页面时已经是一个完整的页面。如现在的详情系统就是采用这种方式。这种方式对服务端性能有些影响，但是用户体验较好。
- CSI。即发起一个异步 JS 请求，单独向服务端获取动态内容。这种方式下的服务端性能更佳，但是用户端页面略有延时，体验稍差。

4. 静态化方案的选择

在做静态化架构设计的方案时要遵循一些原则，想清楚以下问题。

- 是否一致性 Hash 分组？做缓存一定是和命中率紧密相关的，命中率又和数据的集中度相关，而数据集中就必然要求一致性 Hash。但是一致性 Hash 的天然缺陷就是会产生热点，热点特别集中时可能会造成网络瓶颈。
- 是否使用 ESI？前面已经分析过 ESI 和 CSI 的利弊，ESI 对性能有影响但是对客户端友好，对前端编程也方便。
- 是否使用物理机？物理机可以提供更大的内存、更好的 CPU 资源。但是，使用物理机的缺点是会导致应用集群的相对集中，进而增加网络风险；对 Java 系统而言，内存增加并不能带来巨大的好处。
- 谁来压缩？在哪里压缩也是一个比较纠结的问题：增加一层 Cache 必然增加数据的传输。由谁来压缩就会影响 Cache 的容量和网络数据的传输量。
- 网卡选择？网卡选择其实是成本问题，要想避免网络瓶颈可以选择万兆网卡和交换机，但是必然会增加成本。

针对这些问题，我们设计了不同的技术方案，方案之间也是一个迭代演进的过程。

（1）方案 1：实体机单机部署

该方案部署结构如图 7.1 所示。

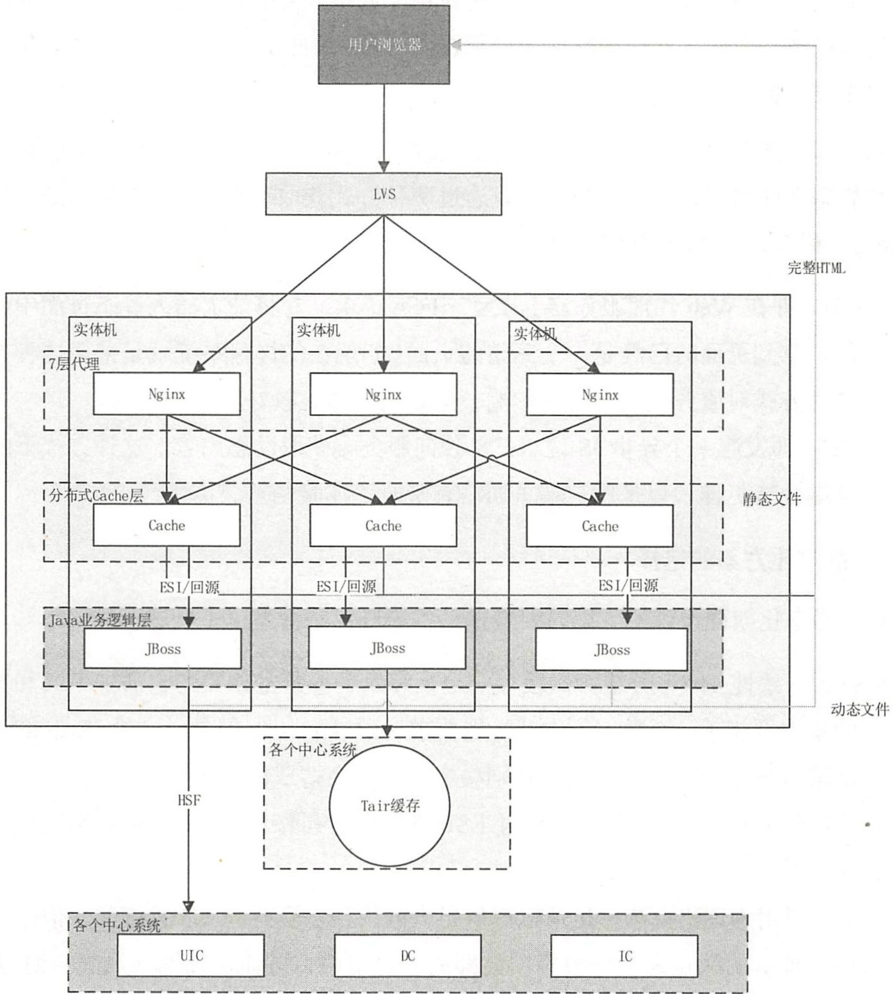


图 7.1　Nginx+Cache+Java 结构实体机单机部署

这种方案增大了 Cache 的内存，并且采用了一致性 Hash 分组的方式来提升命中率。它将 Cache 分成若干组，取得命中率和访问热点的平衡，优点如下：

- 既没有网络瓶颈也能使用大内存；
- 减少 Varnish 机器，提升命中率；
- 提升命中率，减少 Gzip 压缩；

- 减轻 Cache 失效压力。

该方案是一个比较理想的方案，在正常请求下也能达到 50%左右的命中率，对一些基数数据比较小的系统的命中率能达到 80%左右，也是较为理想的。

（2）方案 2：统一 Cache 层

统一 Cache 层是更理想的推广方案，该方案的结构如图 7.2 所示。

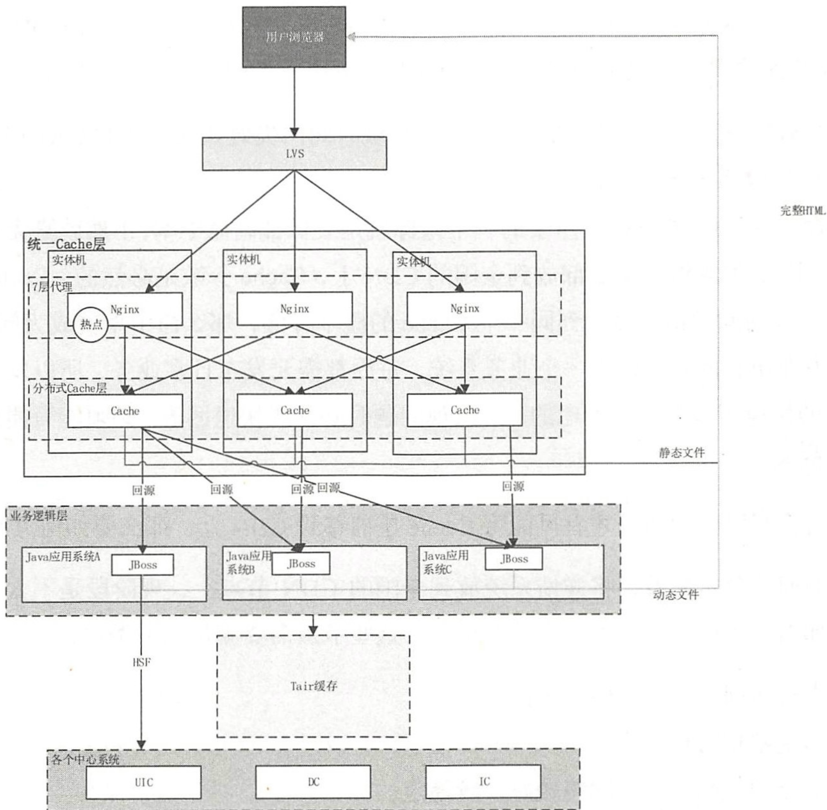


图 7.2 统一 Cache

将 Cache 层单独拿出来统一管理可以减少运维成本，同时也方便其他静态化系统接入。除此之外，它还有如下优点：

- 可以减少多个应用接入使用 Cache 的成本，接入的应用只需维护自己的 Java 系统，不用单独维护 Cache，只要关心如何使用，让更多流量型系统更好地接入使用即可；

- 统一 Cache 易于维护，后续可以加强监控、配置的自动化，维护升级也比较方便；
- 可以共享内存、最大化利用内存，不同系统之间的内存可以动态切换，有效应对攻击等情况；
- 更有助于安全防护。

（3）方案 3：上 CDN

在将动态系统静态化后，很自然会想更进一步——把 Cache 前移到 CDN 上，因为 CDN 离用户最近，效果会更好。但前提是要解决以下问题。

- 失效问题。分布在全国的 CDN 要在秒级时间内失效如此广泛的 Cache 对 CDN 的失效系统要求很高。
- 命中率问题。Cache 最重要的一个指标就是要保证高命中率，不然就毫无意义。同样，如果将数据全部放到全国的 CDN 上，Cache 分散是必然的，Cache 分散导致访问的请求命中到同一个 Cache 的概率降低，那么命中率就成为问题。
- 发布更新问题。作为一个业务系统，每周都需要发布日常业务，所以发布系统的快速简单是必须考虑的——出现问题时必须能快速回滚、必须能简便地排查问题。

只有克服以上问题，才有可能将 Cache 层前移到 CDN 上，那么要如何克服呢？

从前面的分析来看，将详情系统放到全国的 CDN 节点上，现阶段是不太可能实现的。那么是否可以选择若干个节点试行？这些节点需要满足以下条件：

- 靠近访问量比较集中的地区；
- 离主站相对较远；
- 节点到主站之间的网络较好、较稳定；
- 节点容量比较大、不会占用其他 CDN 太多资源；
- 节点不用太多。

基于以上几个条件，选择 CDN 的二级 Cache 会比较合适，它的部署方式如图 7.3 所示。

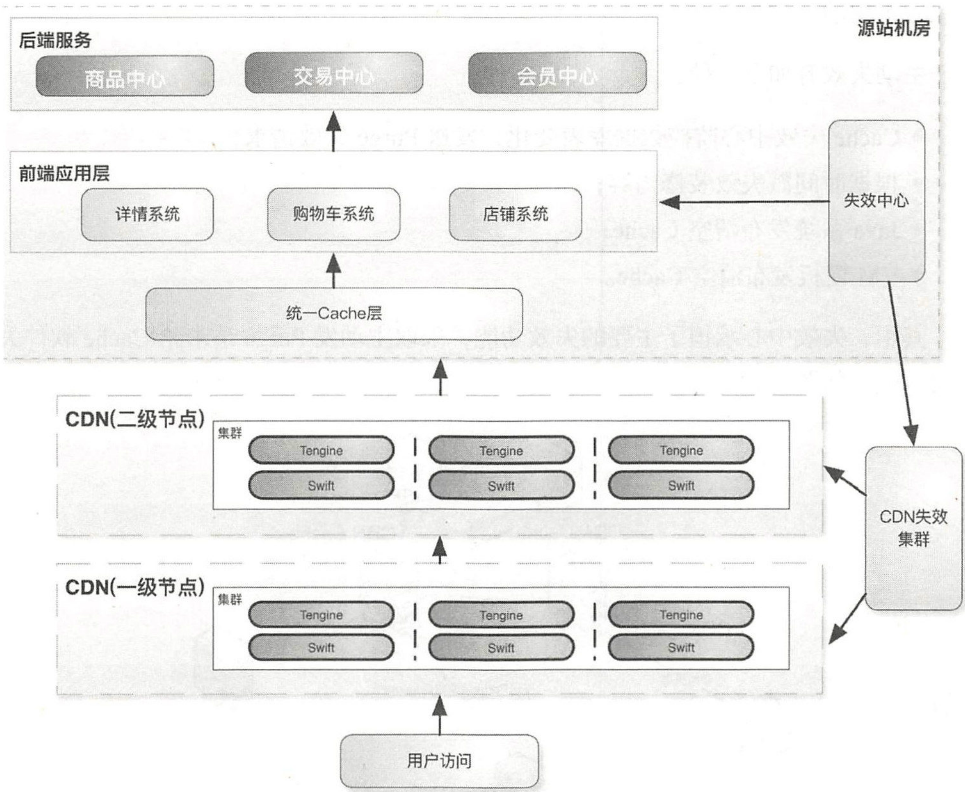


图 7.3　CDN 化部署方案

使用 CDN 的二级 Cache 作为缓存，可以达到和当前服务端静态化 Cache 类似的命中率，因为节点数不多，Cache 不是很分散，访问量也比较集中，这样也解决了命中率问题，同时给用户提供了最好的访问体验，是在当前环境下比较理想的 CDN 化方案。

4．如何失效

搞清楚了怎么缓存、缓存什么之后，接下来就是该考虑如何失效了。失效采用主动失效与被动失效相结合的方式。

（1）被动失效

被动失效主要处理模板变更和一些对时效性不太敏感的数据的失效，采用设置 Cache 时间长度这种自动失效的方式，同时也要开发一个后台管理界面用于手工失效某些 Cache。

（2）主动失效

主动失效有如下几种：

- Cache 失效中心监控数据库表变化，发送 Purge 失效请求；
- 根据时间戳失效装修内容；
- Java 系统发布清空 Cache；
- VM 模板发布清空 Cache。

其中，失效中心承担了主要的失效功能，采取主动发 Purge 请求给 Cache 软件失效的方式，如图 7.4 所示。

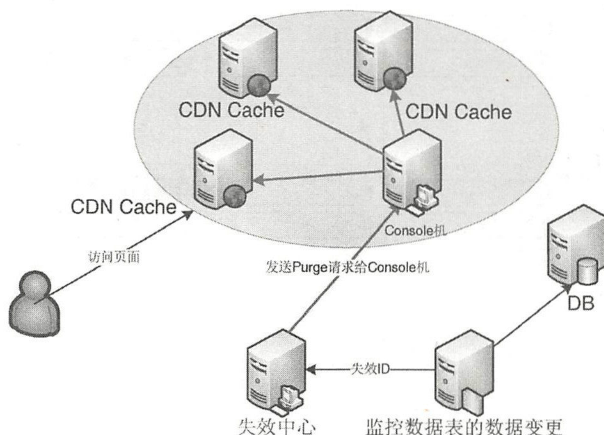


图 7.4　级联失效方式

这种方式由失效中心把失效请求发送给每个 CDN 节点上的 Console 机，再由 Console 机发送 Purge 请求给每台 Cache 机器。

我们的大秒系统是在商品详情系统的基础上发展而来，所以大秒系统本身已经实现了动静分离，除了静态化之外大秒系统还做了很多极致优化，它最新的架构如图 7.5 所示。

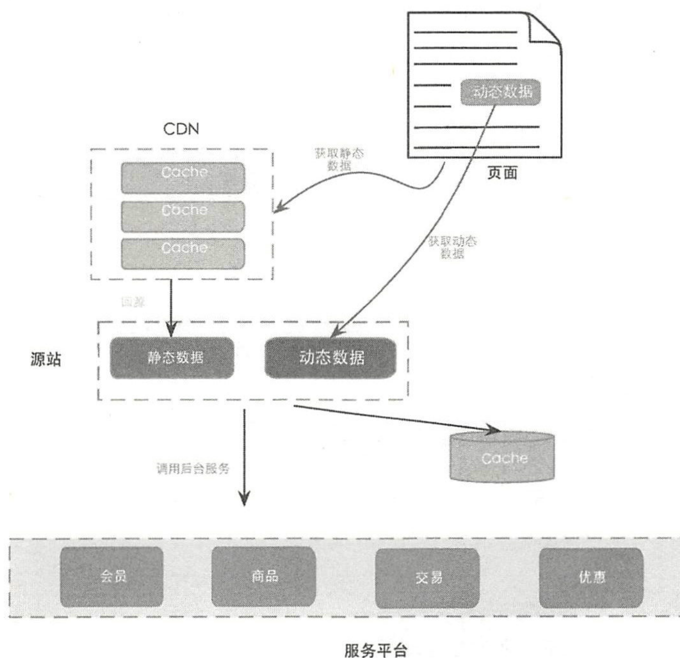


图 7.5　大秒系统架构

大秒系统还有如下特点：

- 把整个页面 Cache 在用户浏览器上；
- 如果强制刷新整个页面，也会请求到 CDN；
- 实际有效请求只是“刷新抢宝”按钮。

通过这样的设计，可以把 90%的静态数据缓存在用户端或者 CDN 上，当秒杀真正开始时，用户不需要刷新整个页面，只需要点击特殊的按钮“刷新抢宝”即可。这种操作只会向服务端请求很少的有效数据，而不需要重复请求大量的静态数据。秒杀的动态数据比普通详情页面的动态数据少，系统的性能也比普通的详情系统提升 3 倍以上。“刷新抢宝”的设计思路能很好地实现不刷新页面就能请求到服务端的最新动态数据。

7.4　基于时间分片削峰

熟悉秒杀的人都知道第一版的秒杀系统本身并没有答题功能，后面才增加了秒杀答题。秒杀答题很重要的作用是防止秒杀器。2011 年秒杀非常火的时候，秒杀器也比

较猖獗。增加答题后，下单的时间基本控制在 2 秒之后，秒杀器的下单比例也下降到 5%以下。新的答题页面如图 7.6 所示。

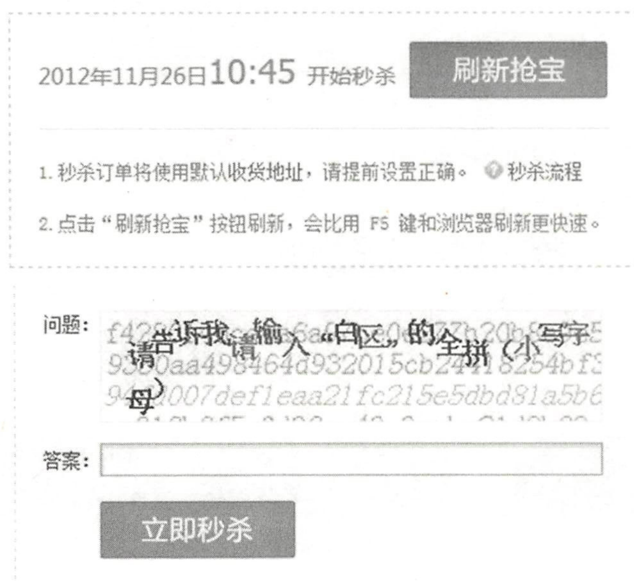


图 7.6　大秒系统的答题页面

增加答题的另一个重要功能就是拉长了峰值的下单请求时间，从以前的 1 秒之内延长到 2~10 秒左右，请求峰值开始基于时间分片。时间的分片对于在服务端处理并发非常重要，它减轻了服务端的压力；秒杀活动建立在请求的时间先后顺序上，靠后的商品请求自然就无法得到满足（没有库存），也就走不到最后下单这一步——这样，真正的并发写的量就非常有限了。这种设计思路目前运用得非常普遍，像支付宝的“咻一咻”、微信的“摇一摇”都是类似的思路。

除了在前端设置答题限制用户端的流量外，在服务端我们一般通过锁或者队列控制瞬间的请求量。

7.5　数据分层校验

对大流量系统的数据做分层校验也是一项重要的设计原则。分层校验就是用“漏斗”式的设计来处理请求，如图 7.7 所示。

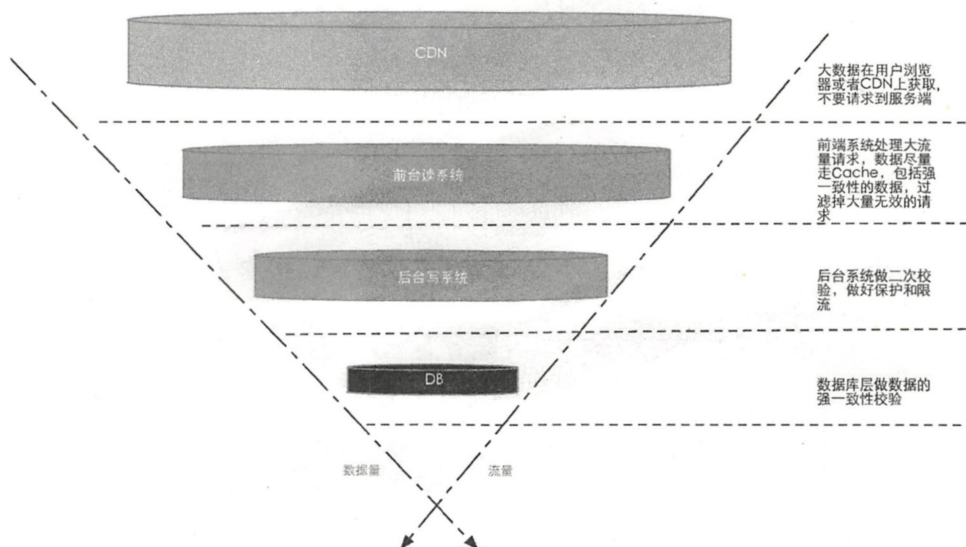


图 7.7　数据分层校验的漏斗设计

它的核心思想是在不同的层次、不断尽可能地过滤掉无效请求，只有“漏斗”最末端的才是有效请求。要达到此效果就必须对数据做分层的校验，以下是分层校验的基本原则：

- 先做数据的动静分离；
- 将 90%的数据缓存在客户端浏览器；
- 将动态请求的读数据 Cache 在 Web 端；
- 对读数据不做强一致性校验；
- 对写数据进行基于时间的合理分片；
- 对写请求做限流保护；
- 对写数据进行强一致性校验。

秒杀系统正是按照这个原则设计的，它的系统架构如图 7.8 所示。

把大量静态、不需要检验的数据放在离用户最近的地方；在前端读系统中检验一些基本信息如用户是否具有秒杀资格、商品状态是否正常、用户答题是否正确、秒杀是否已经结束等；在写数据系统中再校验一些信息：是否非法请求、营销等价物（淘金币等）是否充足、写的数据一致性（检查库存）如何……最后在数据库层保证数据最终准确性（如库存不能减为负数）。

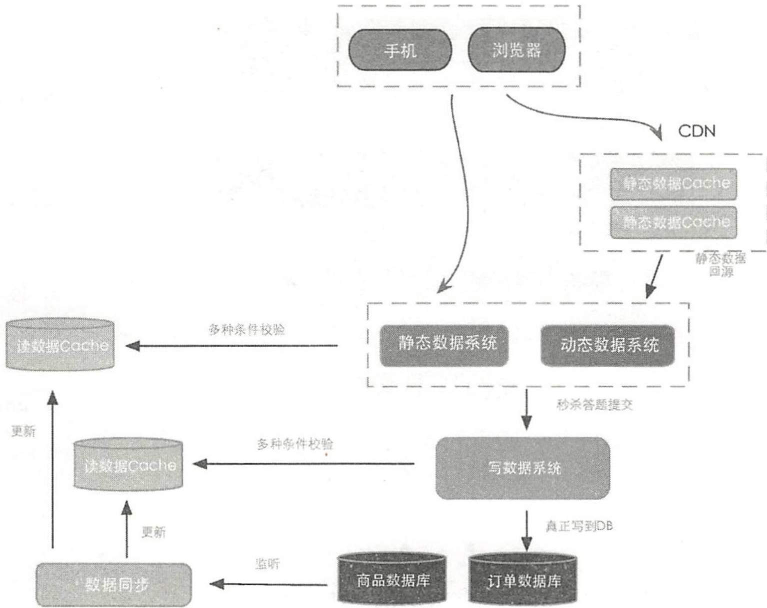


图 7.8　秒杀系统的架构

7.6　实时热点发现

秒杀系统本质上仍然是读数据的热点问题，而且是最简单的一种。前面已经提及通过业务隔离能够提前识别热点数据，就好比通过分析历史成交记录可以发现热门商品、通过分析用户的购物车记录也可以发现比较好卖的商品一样。难点在于未能提前发现的商品突然成为热点——这就要通过实时热点数据分析来判断了。我们做了可以在 3 秒内发现交易链路上的实时热点数据的设计，并根据实时发现的热点数据实时保护每个系统，它的具体实现步骤简述如下。

第一步，构建一个异步的、可以收集交易链路上各个中间件产品（如 Tengine、Tair 缓存、HSF 等）本身统计的热点 key（Tengine 和 Tair 缓存等中间件产品本身已经有热点统计模块）；

第二步，建立热点上报和可以按照需求订阅的热点服务的下发规范，主要目的是通过交易链路上各个系统（详情、购物车、交易、优惠、库存、物流）访问的时间差，把上游已经发现的热点透传给下游系统，提前做好保护（比如在大促高峰期，详情系统是最早知道哪些是热点商品的），在统计接入层上用 Tengine 模块统计出热点 URL；

第三步，将上游系统收集到的热点数据发送到热点服务台上，这样下游系统如交易系统就会知道哪些商品被频繁调用，再对此做热点保护。

整个过程如图 7.9 所示。

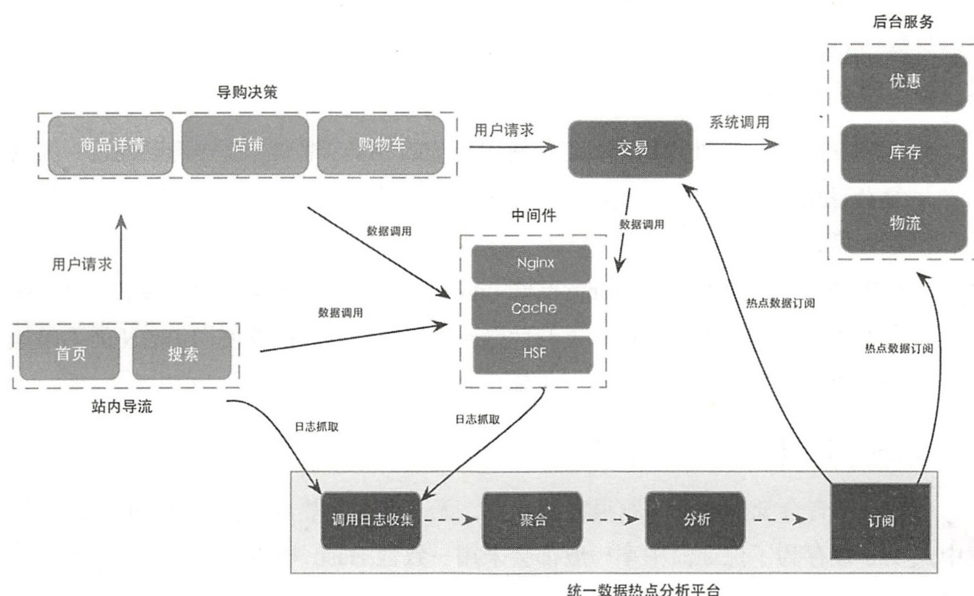


图 7.9　实时热点分析过程

几项重要注意事项如下。

- 热点服务后台抓取热点数据日志最好是异步的，一方面便于做到通用性，另一方面不影响业务系统和中间件产品的主流程；
- 热点服务后台、现有各个中间件和应用本身的保护措施没有取代关系，每个中间件和应用还需要保护自己。热点服务台提供收集热点数据、热点订阅服务的统一规范和工具，便于各个系统热点数据透明化。
- 热点发现要做到实时（3 秒以内）。

7.7　关键技术优化点

前面介绍了大流量读系统的设计手段，当这些手段全部穷尽以后，仍然产生大流量又该如何处理呢？所以秒杀系统还要解决以下关键问题。

1. Java 处理大并发动态请求优化的问题

Java 和通用的 Web 服务器（Nginx 或 Apache）相比，在处理大并发的 HTTP 请求时要弱一点，所以一般我们都会对大流量的 Web 系统做静态化改造，让大部分请求和数据直接在 Nginx 服务器或者 Web 代理服务器（Varnish、squid 等）上直接返回（可以减少数据的序列化与反序列化），Java 层只处理少量数据的动态请求。针对这些请求可以使用以下优化手段：

- 直接使用 Servlet 处理请求。避免使用传统的 MVC 框架，这样可以绕过一大堆复杂且用处不大的处理逻辑，节省 1 毫秒的时间——取决于对 MVC 框架的依赖程度；
- 直接输出流数据。使用 resp.getOutputStream()而不是 resp.getWriter()可以省掉一些不变字符数据的编码，提升性能；数据输出时，推荐使用 JSON 而不是模板引擎（一般都是解释执行）来输出页面。

2. 同一商品被大并发读的问题

也许有读者会觉得这个问题很容易解决，无非就是将热点数据放到 Tair 缓存里。集中式 Tair 缓存为了保证命中率一般都会采用一致性 Hash，所以同一个 key 会落到同一台机器上。虽然单台 Tair 缓存机器也能支撑 1 秒 30 万次的请求，但还是远不足以应付大秒级别的热点商品，该如何彻底解决单点的瓶颈呢？答案是采用应用层的 Localcache，即在秒杀系统的单机上缓存商品相关的数据。那么如何 Cache 数据？答案是划分成动态数据和静态数据分别处理。

- 像商品的标题和描述这些本身不变的数据会在秒杀开始之前全量推送到秒杀机器上、并一直缓存到秒杀结束；
- 像库存这类动态数据会采用被动失效的方式缓存一定时间（一般是数秒），失效后再去 Tair 缓存拉取最新的数据。

读者可能还会有疑问，像库存这种频繁更新的数据一旦数据不一致会不会导致超卖？这就要用到前面介绍的读数据的分层校验原则了，读的场景可以允许一定的脏数据，因为这里的误判只会导致少量原本无库存的下单请求被误认为有库存，可以等到真正写数据时再保证最终的一致性，通过在数据的高可用性和一致性之间的平衡来解决高并发的数据读取问题。

3. 同一数据大并发更新问题

采用 Localcache 和数据的分层校验可以一定程度上解决大并发读问题，但是无论如何还是避免不了减库存这类的大并发写问题，这也是秒杀场景中最核心的技术难题。

同一数据在数据库里肯定是一行存储（MySQL），所以会有大量的线程来竞争 InnoDB 行锁，并发度越高时等待的线程也会越多，TPS 会下降而 RT 会上升，数据库的吞吐量会严重受到影响。这里会出现一个问题，即单个热点商品会影响整个数据库的性能，出现我们不愿意看到的 0.01%商品影响 99.99%的商品的情况。此处的解决思路也是要遵循前面介绍的第一个原则“隔离”——把热点商品放到单独的热点库中，尽管这会带来维护的麻烦（要做热点数据的动态迁移以及单独的数据库等）。

把热点商品分离到单独的数据库并没有解决并发锁的问题，要解决并发锁问题有以下两种办法。

第一种是在应用层做排队。按照商品维度设置队列顺序执行，这样能减少同一台机器对数据库同一行记录操作的并发度，也能控制单个商品占用数据库连接的数量，防止热点商品占用太多的数据库连接。

第二种是在数据库层做排队。应用层只能做到单机的排队，但是应用层机器数量很多，用这种排队方式控制并发仍然是很有限的，如果能在数据库层做全局排队是最理想的。数据库团队开发了 MySQL 的 InnoDB 层上的 patch，可以做到在数据库层上对单行记录并发排队。

该过程如图 7.10 所示。

你可能会有疑问：排队和锁竞争不都是要等待吗，有何区别？如果熟悉 MySQL 的话，应该知道 InnoDB 内部的死锁检测以及 MySQL Server 和 InnoDB 的切换会比较耗性能，MySQL 核心团队还做了很多其他方面的优化，如 COMMIT_ON_ SUCCESS 和 ROLLBACK_ON_FAIL 的 patch，配合在 SQL 里面加 hint，在事务里不需要等待应用层提交 COMMIT 而在数据执行完最后一条 SQL 后，根据 TARGET_AFFECT_ROW 结果就直接提交或回滚，这样可以减少网络的等待时间（平均约 0.7 毫秒）。

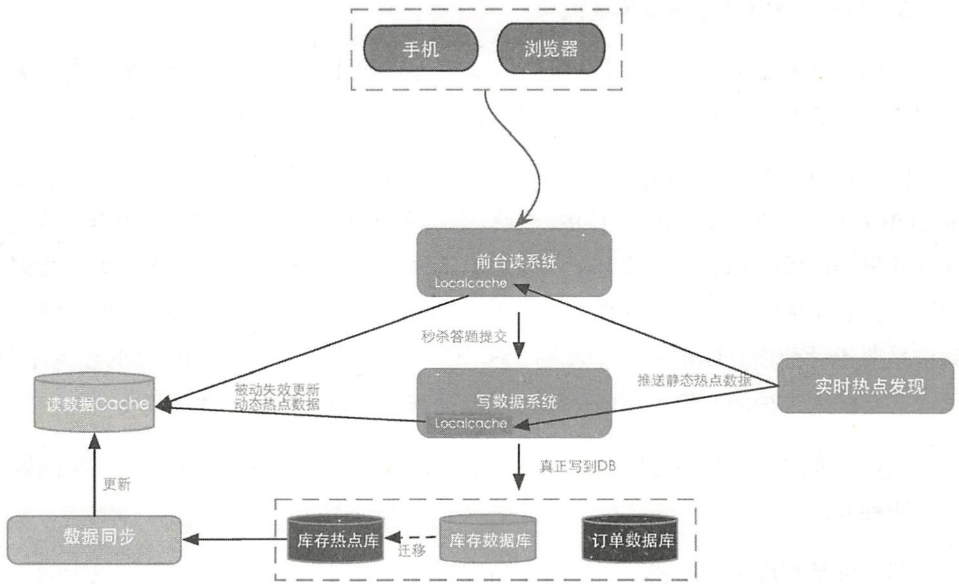


图 7.10　热点数据隔离

7.8 大促热点问题思考

针对秒杀系统所代表的热点问题，我们根据多年的经验总结出一些通用的原则：隔离、动态分离、分层校验，必须从全链路来考虑和优化每个环节……除了优化系统提升性能以外，做好限流和保护也是必备的功课。

除了前面介绍的热点问题，淘系还面临以下其他数据热点问题。

- 数据访问热点，比如详情系统中某些热点商品的访问度非常高，即使是 Tair 缓存这种 Cache 本身也有瓶颈问题，一旦请求量达到单机的极限也会存在热点保护问题。有时候看起来好像很容易解决，比如只需要做好限流，但是一旦某个热点触发了一台机器的限流阀值，那么整台机器 Cache 的数据都将无效，进而间接导致 Cache 被击穿，请求都落到应用的数据库中，出现雪崩现象。所以这类问题需要与具体的 Cache 产品结合才能有比较好的解决方案。

一个通用的解决思路是：在 Cache 的 client 端做本地的 Localcache，当发现热点数据时直接 Cache 在 client 里，而不要请求到 Cache 的 Server。

- 数据更新热点。数据更新问题除了前面介绍的热点隔离和排队处理之外，还有

些场景对商品的 lastmodifytime 字段更新会非常频繁，在某些场景下这些多条 SQL 是可以合并的，一定时间内只执行最后一条 SQL 就行了，这样可以减少对数据库的 update 操作。另外，热点商品的自动迁移理论上也可以在数据路由层完成，利用前面介绍的热点实时发现功能，自动从普通库里把热点数据迁移出来放到单独的热点库中。

- 按照某种维度建立的索引产生的热点数据，比如实时搜索中按照商品维度关联的评价数据。有些热点商品的评价非常多，导致搜索系统在按照商品 ID 建立评价数据的索引时，内存已经存不了了。交易维度关联订单信息也同样有这些问题。这类热点数据需要做数据的散列，需要再增加一个维度，重新组织数据。

7.9 总结

本章重点介绍了一个系统全链路优化的案例。大秒系统是对性能有极致优化需求的系统，所以我们针对它做了很多看起来非常极致的事情，如前端的大部分页面缓存、只刷新极少的数据，服务端去掉 MVC 框架而直接使用 Servlet 处理请求，以及热点数据的隔离等。从用户的请求到服务端的整个请求链路上，每个环节都可以检查是否有能够优化的地方，以达到极致优化的效果。

8 全局基础设施优化：资源调度优化

本章介绍全局基础设施的优化。我们做应用层的优化一般都比较关注软件本身的优化，但是支撑应用运行的基础环境，往往有更大的优化空间。基础设施包括基础应用容器如 JDK、Tomcat、VM，操作系统和文件系统甚至硬件设备，它们其实都有优化的空间，而且由于基础设施的优化是事关全局的，所以通用性会更广、收益会更大。本章我们重点阐述资源调度的优化，因为它最具普遍性、价值也更大。

8.1 什么是资源调度

资源调度一般分为两个阶段：一是实现物理资源的虚拟化（即资源的抽象）。由于当前机器的性能越来越好，硬件配置越来越高，直接用物理机跑业务比较浪费，所以将物理机分割成更小单位的虚拟机，这样可以显著提升机器的利用效率，在公司内部一般采用容器技术来隔离资源。二是将资源虚拟化后进一步在时间和空间上实现更细粒度的编排、优化资源的使用。

1. 一些数据

如果公司的几万台机器都是物理机，那么资源的使用率稍低： CPU、内存和硬盘使用率都较低，例如大部分 proxy 代理机器对内存和 CPU 的要求都比较低，我们完

全可以用一个 4 核 8GB 内存的容器替代一台物理机。

经过简单的计算，我们将物理机（48 核 120GB）进行容器化（4 核 8GB），一台物理机可以当成 12 台机器使用，使用率可以提升 12 倍。由此可见，将物理资源统一抽象成统一的虚拟资源对提升效率非常有用。

2．资源调度提升稳定性和运维效率

对物理机资源进行虚拟化可以提升资源的利用率，而对资源的良好调度可以提升业务的稳定性和运维效率，Docker 的火爆也验证了这一点，很多公司也在进行 Docker 化改造，原因如下。

（1）提升运维效率。Docker 的火爆很大程度要归因于它解决了应用的标准化运维问题，使得应用的部署和运维变得非常简单，只需要一个镜像就可以部署，使服务的应用依赖和部署自动化，减少了人为的干预。

（2）提升稳定性。既然可以做到标准化的部署，那么就可以把应用和运行的机器解耦，解耦后，硬件的差异和故障不会影响上层的应用，就可以做弹性伸缩和调度了。

3．统一资源抽象

将物理资源统一抽象成可以定制化的集合，对上层应用屏蔽时间和空间上的差异，即应用不用关心跑在哪台物理机上、哪个机房甚至哪个数据中心，不用担心宕机的影响，在资源不够用时还可以自动扩容。如图 8.1 所示。

图 8.1　统一资源抽象

8.2 资源抽象层

资源抽象层主要是将下层的物理硬件资源统一进行抽象，抽象成和单个物理硬件无关的资源集合，上层无须关心物理机器的型号，只需专注于具体的资源即可。

资源抽象层需要重点做好以下三件事。

第一，收集和管理具体物理资源；

第二，重新封装抽象的硬件资源属性，使之成为上层可以使用的一个实体，既可以是容器也可以是虚拟机或者资源集合；

第三，数据存储问题。做业务少不了要在本机存储数据，这样机器就成为“有状态”的，不利于全局调度资源。为了能够全局调度，需要解决三个场景下的问题：一是数据不需要永久本地存储但是会实时写到本地的，如应用的日志；二是需要永久存储的如 DB 数据；三是分布式存储场景中，要做到存储与计算分离。

1. 资源的收集和管理

资源的收集就是收集物理机的资源，例如当前型号的机器有多少可用的 CPU、内存、磁盘等信息，它可以分为四个方面的内容。

第一，资源的信息管理。有多少，用了多少，还有多少；

第二，大量物理机器的集群管理。除了通常几十万台的机器管理功能外，还有一部分的任务管理，如负责接收 Master 创建容器的任务等。

第三，资源的合理分配策略和算法。上层的资源请求最终会在每台物理机上进行分配，那么如何分配呢？这里有很多优化空间，保证每台物理机资源的合理利用需要合理的分配策略和算法支撑。

第四，资源的信息管理就是要实现一个 CMDB，能管理物理机和 vhost 的关联关系，必须能管理上万台甚至十几万台规模的机器集群。这样的机器集群管理框架目前可选的比较少，我们选择的是 Mesos，主要基于以下两方面的考虑。一是 Mesos 目前相对比较成熟，主流的大公司使用较多，在实际场景中的使用规模已达 5 万台左右；二是 Mesos 扩展性比较好，本身是轻量级的，可以灵活定制各种 Framework 满足业务需要。

我们分析一下为什么 Mesos 能管理这么大的集群，它的资源分配策略以及它是如

何灵活创建各种容器和配置网络的。Mesos 的集群架构如图 8.2 所示。

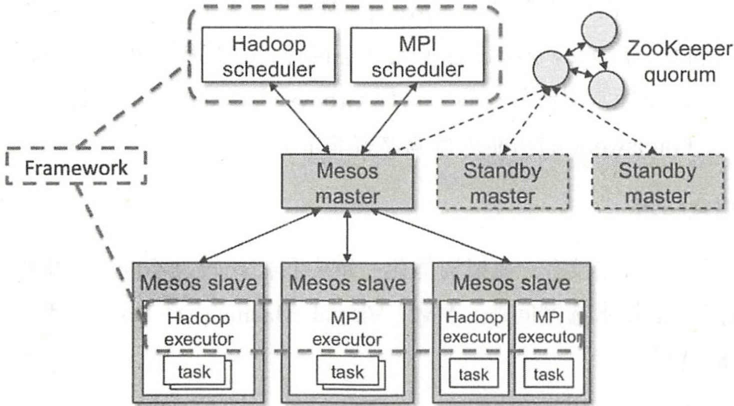


图 8.2　Mesos 的典型架构（http://mesos.apache.org/assets/img/documentation/architecture3.jpg）

Mesos 的模块化设计使得它的集群管理本身可做的事情并不多：Master 仅仅把从 Slave 收集的资源数据汇报给 Framework；Master 和 Slave 通过消息交互消息，不需要一直保持长连接。随着 Slave 规模的扩大，Master 的压力并不会显著增长。Master 本身的高可用是通过 ZK（ZooKeeper）来保证的，整个集群的架构设计非常清晰。

当集群规模很大时，资源的管理和分配策略就会非常重要。分配策略对于最大化充分利用物理资源非常关键，所以要自己定制 Framework 以便更精细化地分配资源。目前我们设计了 4 个分配策略。

（1）最大内存剩余优先分配策略。即集群中内存剩余最多的优先分配，目的是充分使用集群中剩余的空闲内存，把内存资源分配给内存使用量大的系统；

（2）最大 CPU 剩余优先分配策略。类似于内存分配，根据剩余的 CPU 数优先分配给对 CPU 资源需求大的任务；

（3）最大最小资源公平分配策略。这种分配是根据当前任务申请的资源，要查看当前集群中的每台机器、每种资源的使用量是否饱和，优先把任务分配给当前最空闲的机器；

（4）根据资源分配指定分配策略。这种方式比较灵活，就是可以根据用户的需要把任务分配到指定的机器上执行，例如可以给一些机器打上标签，让某类任务在这些带有标签的机器上执行。

从上面的介绍可以知道 Framework 的修改需要比较灵活的支持，而当前 Mesos

的 Framework 的更新还比较麻烦。如果要更新 Framework 的代码，就需要重启每个 Slave 的 execute，进而可能要停止 Slave 上的任务，这在生产环境中是很难接受的。有鉴于此，我们对 Framework 进行了无状态设计，在代码实现上，改用动态语言如 Groovy 来编写需要经常修改的逻辑，这样 Groovy 实现的代码就可以动态加载而不需重启任务，对 Framework 的功能进行调整就非常方便了。

2. 虚拟化技术

当前虚拟化技术比较多，在架构优化一章中我们比较过不同层面虚拟化技术的选择，本节重点讨论技术选型问题：VM（Virtual Machines）、Docker 和 LXC（Linux Container）的选择。

VM 虚拟机确实在原理上比容器技术有更多的消耗，但这些消耗有缺点也有优点，虚拟机可以做到更安全的隔离。容器技术也在不断发展中，有理由相信在不久的将来，容器的安全性也能和虚拟机一样不分伯仲，如图 8.3 所示。

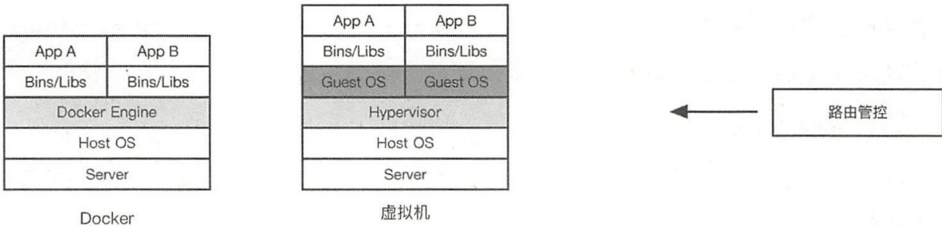


图 8.3　VM 和 Docker 比较（图片来源于 dockone.io）

下面重点讨论 Docker 和 LXC 的选择问题。早期的 Docker 是在 LXC 技术上发展起来的，它从 0.9 版本后才引入了基于 Go 构建的 Libcontainer 的 Execution Driver，它们之间的比较如图 8.4 所示。

其实到目前为止，Docker 和 LXC 在核心技术上的差异并不是那么明显，对于我们来说，最重要的一点是如何解决持久化问题。当前 Docker 是通过 volume 挂载的方式实现数据持久化的，但这对数据存储的目录限制比较死。如果不采用挂载的方式，Docker 本身的存储驱动的基于 copy on write 的技术又会产生约 20%左右的性能损失，这对那些对 I/O 比较敏感的业务不利。

LXC 在存储上虽然也支持 copy on write 技术，但是它可以直接写物理机上的磁盘，性能上基本没什么损失，图 8.5 是 LXC 的磁盘性能测试结果。

仅供非商业用途或交流学习使用

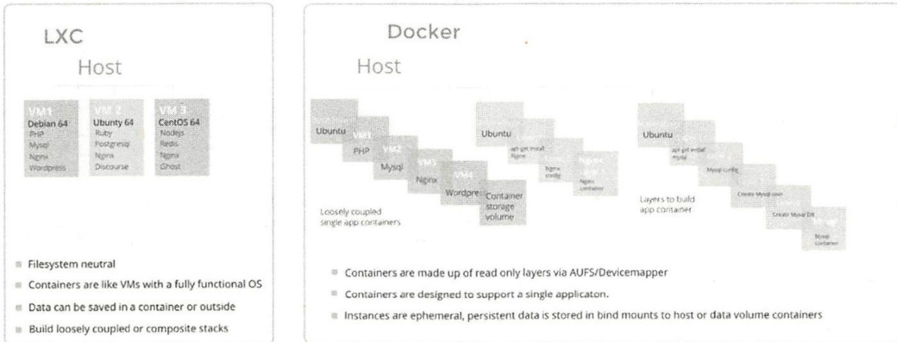
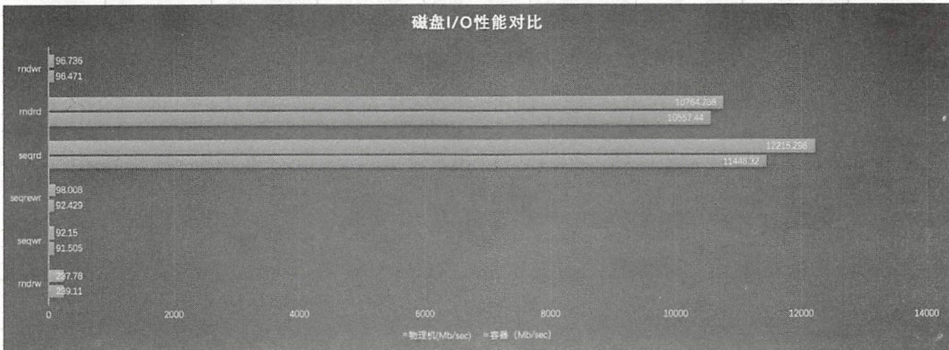


图 8.4 LXC 和 Docker 差异（图片来源 dockone.io）



序号	磁盘读写方式	容器（Mb/sec）	物理机(Mb/sec)	容器(Requests/sec)	物理机(Requests/sec)	并发	块大小（字节）	文件大小（G）
1	rndrw	239.11	237.78	15292.89	15217.78	16	16384	3
2	seqwr	91.505	92.15	5856.34	5897.61	16	16384	3
3	seqrewr	92.429	98.008	5915.48	6272.49	16	16384	3
4	seqrd	11448.32	12215.296	732709.18	781810.57	16	16384	3
5	rndrd	10557.44	10764.288	675648.2	688908.61	16	16384	3
6	rndwr	96.471	96.736	6174.13	6191.1	16	16384	3

图 8.5 LXC 的磁盘性能测试

如果 LXC 选择非 copy on write 技术，会带来另外一个麻烦——即一台物理机上多个容器本身的存储比较费空间（因为多个容器之间的非可写的 libs 不能共享），必须要做取舍。

我们最后选择了折中的方案，在计算型的业务中选择 Docker 作为容器；在对 I/O 要求比较高的场景下选择 LXC 作为容器。

3. 存储技术

在存储技术上，Docker 本身有很多存储驱动可供选择，如 overlay 和 devicemapper，我们做些对比测试。图 8.6 对 devicemapper 和物理机二者的 I/O 做了比较。

仅供非商业用途或交流学习使用

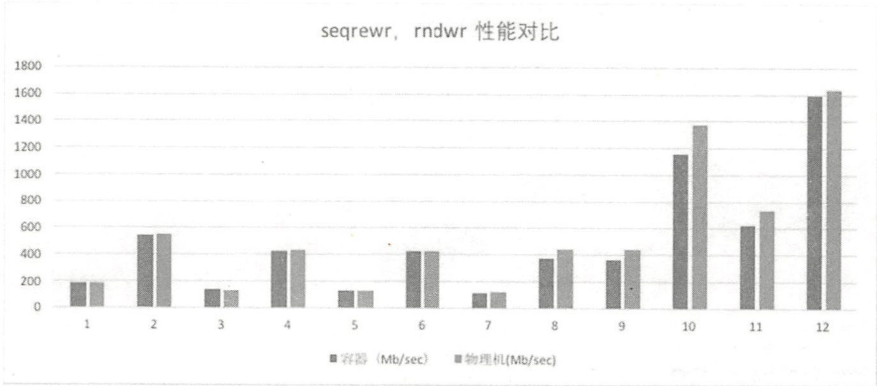


图 8.6　devicemapper 和物理机下的 I/O 测试结果比较

图 8.7 是 overlay 与物理机 I/O 的性能对比。

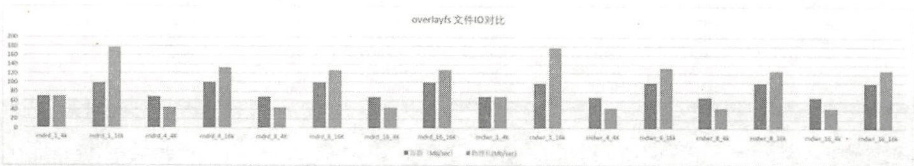
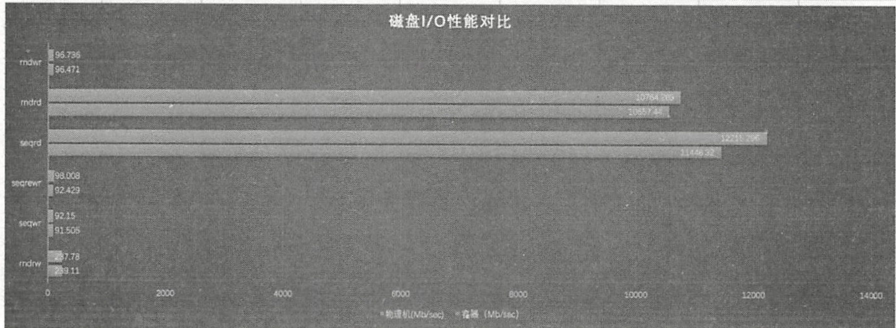


图 8.7　overlay 与物理机下的 I/O 测试结果比较

从上面两个图来看，不管是 devicemapper 还是 overlay，它们与物理机相比，磁盘性能都有 10%~20%的损耗。

那么再来看看 LXC 的情况，图 8.8 是 LXC+LVM 在不同大小的文件下的测试数据。



序号	磁盘读写方式	容器（Mb/sec）	物理机(Mb/sec)	容器(Requests/sec)	物理机(Requests/sec)	并发	块大小（字节）	文件大小（G）
1	rndrw	239.11	237.78	15292.89	15217.78	16	16384	3
2	seqwr	91.505	92.15	5856.34	5897.61	16	16384	3
3	seqrewr	92.429	98.008	5915.48	6272.49	16	16384	3
4	seqrd	11448.32	12215.296	732709.18	781810.57	16	16384	3
5	rndrd	10557.44	10764.288	675648.2	688908.61	16	16384	3
6	rndwr	96.471	96.736	6174.13	6191.1	16	16384	3

图 8.8　LXC+LVM的磁盘性能测试

仅供非商业用途或交流学习使用

从图中可以看出，LXC 下的磁盘性能与物理机的磁盘 I/O 性能基本接近。

存储与计算的分离应该是存储的未来发展方向，每个容器写的数据都能写到一个集中的存储集群中，这样计算节点就能真正做到无状态化，而且计算资源也比较容易调度。

8.3 物理资源调度

资源抽象后就该调度了。要调度资源我们必须要选择一个比较好的调度框架，并选择好的调度和分配策略，尽量最大化地利用资源；还有一些比较高级的调度场景，如先进行计算存储分离，进而进一步实现在线、离线混合部署。

1. 调度框架选型

当前市场上开源的调度框架非常多，图 8.9 是网上流传的各种调度框架的对比图，比较主流的调度框架有 Kubernetes、Swarn、YARN 和 Mesos，它们各有优缺点和各自的应用场景。

	Framework	Architecture	Resource granularity	Multi-scheduler	Pluggable logic	Priority preemption	Re-scheduling	Oversubscription	Resource estimation	Avoid interference
OPEN	Kubernetes	monolithic	multi-dimensional	N[v1.2, DD, Issue]	Y[DD]	N[Issue]	N[Issue]	Y[DD]	N	N
	Swarm	monolithic	multi-dimensional	N	N	N[Issue]	N	N	N	N
	YARN	monolithic/ two-level	RAM/CPU slots	Y	N[app-lvl. only]	N[JIRA]	N	N[JIRA]	N	N
	Mesos	two-level	multi-dimensional	Y	Y[framework-lvl.]	N[JIRA]	N	Y[v0.23, Doc]	N	N
	Nomad	shared-state	multi-dimensional	Y	Y	N[Issue]	N[Issue]	N[Issue]	N	N
	Sparrow	fully-distributed	fixed slots	Y	N	N	N	N	N	N
CLOSED	Borg	monolithic[2]	multi-dimensional	N[2]	N[2]	Y	Y	Y	Y	N
	Omega	shared-state	multi-dimensional	Y	Y	Y	Y	Y	Y	N
	Apollo	shared-state	multi-dimensional	Y	Y	Y	Y	N	N	N

图 8.9　不同调度框架的比较（图片来源于网络）

我们的选择主要考虑两个因素：

- 支持的规模、稳定性。Mesos 当前已经能够支撑大于 3 万个节点、25 万个容器，在大公司如 Twitter、苹果、Uber、ebay 和 Airbnb 大量应用；
- 兼容性、接入成本。需要支持大数据集群的接入；系统支持各种中间件自定义的运维部署系统的接入。

仅供非商业用途或交流学习使用

综上，我们最后选择 Mesos 作为调度框架，并且实现了自定义的 Framework。

2. 统一调度框架的架构

图 8.10 是基于 Mesos 构建的资源调度平台的整体架构，支持的容器有计算型的 Docker 和存储型的 LXC。

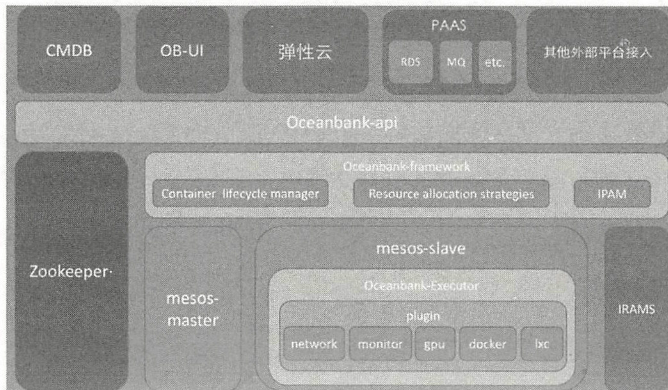


图 8.10 一种资源调度平台架构示例

网络方面同时支持 SDN 和 SRIOV 网络类型，容器里的环境和权限与基础的运维体系都已经打通，并且支持 SSH 直接登录到容器中，登录进去看到的 CPU、内存等资源都是经过隔离的信息。例如分配的 4 核 8GB 的大小，top 命令看到的就是 4 核 8GB。我们使用 LXCFS 隔离，还实现了官方没有实现的 load 的隔离。

LXCFS 中的 load 隔离，实现步骤如下。

（1）计算公式：load(t) = load(t-1) exp(-5/60R) + n(t) (1-exp(5/60R));

（2）如果想在容器中获取正确的 Loadavg 信息，就需要具备以下信息：

- 获取运行在容器中的所有进程（包括：线程）；
- 获取运行在容器中的进程总数；
- 获取运行在容器中的所有进程运行状态；
- 利用 Loadavg 计算公式计算。

（3）难点在于如何控制性能消耗：控制对进程信息获取的系统调用。

在测试 load 隔离的消耗中，我们在容器里模拟了 1 万个进程，每 5 秒进行一次计算，测试发现 load 隔离大约占用 2%的 CPU 资源，目前还是可以接受的。

仅供非商业用途或交流学习使用

图 8.11 是调度平台的界面。

图 8.11　一种资源调度的管理示例

3．存储、计算分离

资源调度的核心是把机器上运行的应用系统与物理机器解耦从而实现更动态的迁移，这样以后不论是动态扩容还是容灾都会很方便。计算节点的动态迁移比较容易实现，有状态的存储数据就比较麻烦了，而将应用的存储数据与存储机器进行解耦就需要更进一步地实现存储计算分析技术。

实现方法就是分离存储和计算，将数据储存在单独的存储节点中。需要数据时再把数据节点挂载到计算节点上，这样计算节点和存储数据的节点就是两个相对独立的逻辑集群，可以动态匹配。这个方法在技术实现上并不复杂。我们可以利用开源的分布式存储系统如 CEPH 等实现数据的分布式部署，再通过接口或者直接挂载的方式访问数据。这里真正复杂的是数据和技术的精确匹配，要考虑到数据细粒度的划分和数据在技术节点和存储节点之前的数据量的传输，即要考虑内网的网络带宽和访问数据的网络延时。

整体来看，计算与存储分离可以解决应用本地 I/O 的瓶颈，将本地 I/O 转换成网络 I/O——显然网络 I/O 的问题更容易解决。随着网卡和内网带宽的不断升级，存储与计算分离越来越有可能成为趋势。

仅供非商业用途或交流学习使用

4. 在线、离线混合部署

在线、离线混合部署（以下简称"在离线混部"）是另外一个提升资源利用效率的有效手段。通常而言，在线的系统都是白天的负载高一点，但晚上一般都会非常低，即使在白天也会有比较明显的波峰和波谷，所以机器利用效率存在比较大的差异。离线任务的真正执行时间可以由我们控制，可以安排在晚上执行，这样就可以利用机器的空闲时间段，如果控制得好，理论上可以提升一倍的机器利用效率。

实现在离线混部有以下前提。

（1）要实现在离线任务的灵活调度。即当机器资源有空闲时，能够通过调度系统方便地调度离线的任务并执行；

（2）要实现在离线任务的资源隔离。即离线任务不能影响在线任务的执行，不能抢占原本属于在线任务的资源。

（3）要能监控在线任务的 QoS，以便实时向调度系统反馈并进行合理的分配。

在实际的应用场景中真正要运用在离线混部还会遇到很多挑战。如大数据集群中的瓶颈到底是计算还是存储；再比如，计算的弹性扩展的实现必然伴随着数据的频繁移动，这会导致 I/O 增大，此时就需要考虑内网的带宽。当前很多大公司已经将离线大数据的计算集群和在线业务的集群分别部署在两个机房中，例如将大数据集群迁到比较僻远的机房，节省机房、电力以及网络带宽成本；将大数据集群集中部署也会降低成本。所以在考虑在离线混部时，要从整体上精确地计算是否划算。

8.4 应用层调度

自从有了 Docker 后，应用的弹性调度更加方便了，应用层调度有两个最重要的应用场景：

（1）突发流量下的弹性伸缩。弹性伸缩不仅是指扩容，还包括在业务低峰时的缩容，以释放多余资源；

（2）通过调度实现故障节点的自动摘除，再重新拉起一个容器来替换故障节点。

1. 弹性伸缩

分享笔者经历的一个例子。我们曾经上线 A 集群，导致对另外一个集群 B 的调

仅供非商业用途或交流学习使用

用量增加很多倍，由于没有及时发现，结果集群 B 被压垮。集群 B 的扩容很麻烦，很长时间都无法恢复正常。从中我们总结出两点经验：

第一，新应用的上线会导致线上流量比例发生变化，需要有能感知到这种变化的机制，提前做好集群的容量规划；

第二，集群的扩容要做到自动化：一键部署、弹性伸缩。

在没有很好的检测手段的情况下，线上的容量规划只能凭感觉判断（是否要增加机器），因此，在能达到弹性伸缩之前，先得对现有集群做自动化的容量规划，我们可以通过以下方法来操作。

（1）线下单机压测

任何系统上线之前都要做基本的性能评估，也就是在单位机器上能支撑的最大 QPS 以及 RT 是多少，它的瓶颈点在哪里？哪个业务场景会触发性能瓶颈点？掌握基本的性能数据后就可以部署上线了。

（2）线上真实流量的压测

由于线上的真实环境复杂多变，线下环境的单点压测数据未必十分准确。线上真实数据可能和线下压测的结果有出入，而数据量大小又会非常影响性能，因此稳妥的做法是在线上的真实环境下用真实数据做若干次压测。如何做线上真实环境的压测呢？我们一般采取以下手段。

- 针对读系统，可以直接在线上制造流量。例如，可以在页面中埋一些请求，当用户打开浏览器时自动发送一个请求回主站，请求都是由用户发起的，只不过多发了一个请求，所以是真实的；或者在线上针对 URL 用 ab、TCPCopy 等压力测试工具进行压测。
- 针对写系统的操作（不能产生脏数据）稍微复杂些。方法是通过引流的方式把一个集群中的流量引到一台或多台压测的机器上，由于都是真实的用户请求，只不过人为把它们引导到压测机器上执行，所以这种操作不会产生脏数据。但操作时必须要控制好，防止压垮压测机器导致线上故障，建议选择夜间操作。
- 线上全链路压测。即便有了线上真实环境下的单机性能测试数据，仍然会有变数。因为一个系统真正在线上运行时，会涉及丰富多样的场景。像订单系统除了下单请求还有查询、删除和修改订单请求，这些请求类型都不一样，请求的数量也不一样，所以并没有一个公式可以精确地计算出线上真实场景下的系统

仅供非商业用途或交流学习使用

容量，唯一的手段就是根据不同的场景做全链路的压测。全链路压测已在前面介绍，此处不再赘述。

以上手段的最终目标都是构建一个应用的容量基线，以此评估当前应用的容量水位，当水位达到一定水平时就需要扩容了。

除了设定性能基线，还需要获取实时的线上负载数据，当集群的整体负载达到60%时就需要扩容。一般公司都已经具备实时监控能力，比较容易获取线上的负载数据，剩下的工作就是把这些数据与弹性伸缩系统打通。

2. 执行弹性调度

获取了负载数据就可以根据需要做弹性伸缩。执行弹性伸缩的手段目前也比较多，此前的调度产品都非常成熟，这里需要强调两点：一是服务的自动路由比较重要，如果基于IP做路由调用需要注意及时清除缓存并更新IP路由表；如果是基于域名做路由，那么naming服务也要足够可靠，像主动探活和被动探活的准确性和灵敏性就非常重要，否则在切换时会有抖动。二是外层的流量负载均衡应随后端机器的变化及时更新，把流量调度到新机器上，一些比较高端的流量调度还能根据流量比例切换来实现系统预热，防止大流量到新机器上产生请求超时问题。

3. 故障自愈

根据经验，每年大约有2%的机器会发生故障，包括硬件故障、软件故障和线上发布出现的故障。

硬件故障的解决办法是让系统与机器解耦，前提有二：一是应用系统能够程序化管理自己的环境依赖，不管是操作系统还是某些lib库，都能被程序化地管理；同时能够自动化运维和发布这些依赖。二是必须让应用系统的存储和计算有一定程度的分离，至少要能够重建计算所依赖的存储数据。

软件故障解决起来相对麻烦一些，因为大部分软件故障都是人为的。解决软件故障的前提是建立软件的基线。基线就是上一次正确运行时，所有软件系统依赖的版本快照。每修改一处代码都需要更新一个新的版本，一旦出现故障就可以回滚到上一个版本。当然并不是每一次更新都能够回滚，所以对待这种不能回滚的变更就要格外小心了。

除了软件的版本化控制，我们还可以在变更时提供一些保障措施，例如多版本部署或者蓝绿发布等，目的就是在发布变更时可以快速切换到正常的版本。此点前面已

经有过介绍，此处不再赘述。

8.5　遇到的问题

在资源调度中经常会遇到一些疑难问题，这些问题可能会对大家有些帮助，在此分享给大家。

1. 一个 Java 线程夯住问题的排查

Es 迁入 ob 后出现一个奇怪的问题，偶尔会有一个监控项无法获取值，这个监控项的目标是取物理机的系统 load 值，此问题隔段时间又再次出现，于是我们立即用 jstack dump 线程栈进行检查，如图 8.12 所示。

```
"elasticsearch[a          ][management][T#3]" #253 daemon prio=5 os_prio=0 tid=0x00007f122c08c000 nid=0x238 runnable [0x00007f12947c8000]
   java.lang.Thread.State: RUNNABLE
        at sun.management.OperatingSystemImpl.getSystemCpuLoad(Native Method)
        at sun.reflect.GeneratedMethodAccessor7.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at org.elasticsearch.monitor.Probes.getLoadAndScaleToPercent(Probes.java:29)
        at org.elasticsearch.monitor.os.OsProbe.getSystemCpuPercent(OsProbe.java:120)
        at org.elasticsearch.monitor.os.OsProbe.osStats(OsProbe.java:146)
        at org.elasticsearch.monitor.os.OsService$OsStatsCache.refresh(OsService.java:70)
        at org.elasticsearch.monitor.os.OsService$OsStatsCache.refresh(OsService.java:63)
        at org.elasticsearch.common.util.SingleObjectCache.getOrRefresh(SingleObjectCache.java:55)
        at org.elasticsearch.monitor.os.OsService.stats(OsService.java:60)
        - locked <0x000000046c3e47c0> (a org.elasticsearch.monitor.os.OsService)
        at org.elasticsearch.node.service.NodeService.stats(NodeService.java:159)
        at org.elasticsearch.action.admin.cluster.stats.TransportClusterStatsAction.nodeOperation(TransportClusterStatsAction.java:102)
        at org.elasticsearch.action.admin.cluster.stats.TransportClusterStatsAction.nodeOperation(TransportClusterStatsAction.java:54)
        at org.elasticsearch.action.support.nodes.TransportNodesAction.nodeOperation(TransportNodesAction.java:92)
        at org.elasticsearch.action.support.nodes.TransportNodesAction$NodeTransportHandler.messageReceived(TransportNodesAction.java:230)
        at org.elasticsearch.action.support.nodes.TransportNodesAction$NodeTransportHandler.messageReceived(TransportNodesAction.java:226)
        at org.elasticsearch.transport.RequestHandlerRegistry.processMessageReceived(RequestHandlerRegistry.java:75)
        at org.elasticsearch.transport.netty.MessageChannelHandler$RequestHandler.doRun(MessageChannelHandler.java:300)
        at org.elasticsearch.common.util.concurrent.AbstractRunnable.run(AbstractRunnable.java:37)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)

   Locked ownable synchronizers:
        - <0x000000046bac7a68> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
        - <0x00000004707753c8> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

图 8.12　jstack dump 线程栈

从图中可以看出 dump 多次线程栈一直挂在原地没有返回，可以确定线程被夯住了。由于这是个系统调用，所以要看一下这个系统调用执行到了哪个地方？可以通过 pstack 来查看系统的调用栈情况，找到这个线程的 ID 如图 8.13 所示。

```
[ar                  r002 ~]$ pstack 568
Thread 1 (process 568):
#0  0x00007f15272399ed in read () from /lib64/libc.so.6
#1  0x00007f15271c99b0 in __GI__IO_file_underflow () from /lib64/libc.so.6
#2  0x00007f15271ca93e in __GI__IO_default_uflow () from /lib64/libc.so.6
#3  0x00007f15271c5bce in getc () from /lib64/libc.so.6
#4  0x00007f1445389cb8 in get_totalticks () from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.65-3.b17.el7.x86_64/jre/lib/amd64/libmanagement.so
#5  0x00007f144538a12e in get_cpuload_internal () from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.65-3.b17.el7.x86_64/jre/lib/amd64/libmanagement.so
#6  0x00007f144538a2e5 in get_cpu_load () from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.65-3.b17.el7.x86_64/jre/lib/amd64/libmanagement.so
#7  0x00007f15117584ae in ?? ()
#8  0x000000046c3df050 in ?? ()
#9  0x00007f11bc000a30 in ?? ()
#10 0x00007f11c4002ac8 in ?? ()
#11 0x0000000000000000 in ?? ()
```

图 8.13　pstack 调用栈

通过分析这个 libmanagement.so 的 get_totalticks()函数(http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/14a4b0a9a0b7/src/solaris/native/sun/management/LinuxOperatingSystem.c)发现，计算 load 值是要读取/proc/stat 文件，如图 8.14 所示。

图 8.14　/proc/stat 文件

由于最后是在系统调用的地方没有返回，用 strace –p pid 可以看到当前正在执行的系统调用，如图 8.15 所示。

```
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, "", 4096)                    = 0
read(7392, ^CProcess 568 detached
```

图 8.15　strace –p pid 示例

发现一直在系统调用 read()，可以很可能是读文件死循环了，那么用 gdb 调试一下，到底在执行什么代码？如图 8.16 所示。

```
0x00007f1445389d53 <+291>:   mov    -0x50(%rbp),%rax
0x00007f1445389d57 <+295>:   add    -0x58(%rbp),%rax
0x00007f1445389d5b <+299>:   mov    %rdx,0x8(%rcx)
0x00007f1445389d5f <+303>:   add    -0x40(%rbp),%rdx
0x00007f1445389d63 <+307>:   mov    %rax,(%rcx)
0x00007f1445389d66 <+310>:   add    %rdx,%rax
0x00007f1445389d69 <+313>:   mov    %rax,0x10(%rcx)
0x00007f1445389d6d <+317>:   xor    %eax,%eax
0x00007f1445389d6f <+319>:   mov    -0x38(%rbp),%rsi
0x00007f1445389d73 <+323>:   xor    %fs:0x28,%rsi
0x00007f1445389d7c <+332>:   jne    0x7f1445389da3 <get_totalticks+371>
0x00007f1445389d7e <+334>:   add    $0x58,%rsp
0x00007f1445389d82 <+338>:   pop    %rbx
0x00007f1445389d83 <+339>:   pop    %r12
0x00007f1445389d85 <+341>:   pop    %r13
0x00007f1445389d87 <+343>:   pop    %r14
0x00007f1445389d89 <+345>:   pop    %r15
0x00007f1445389d8b <+347>:   pop    %rbp
0x00007f1445389d8c <+348>:   retq
0x00007f1445389d8d <+349>:   mov    %rbx,%rdi
---Type <return> to continue, or q <return> to quit---
0x00007f1445389d90 <+352>:   callq  0x7f14453887a0 <fclose@plt>
0x00007f1445389d95 <+357>:   mov    $0xfffffffe,%eax
0x00007f1445389d9a <+362>:   jmp    0x7f1445389d6f <get_totalticks+319>
0x00007f1445389d9c <+364>:   mov    $0xffffffff,%eax
0x00007f1445389da1 <+369>:   jmp    0x7f1445389d6f <get_totalticks+319>
0x00007f1445389da3 <+371>:   callq  0x7f14453887e0 <__stack_chk_fail@plt>
End of assembler dump.
```

图 8.16　gdb 调试示例

从执行逻辑来看，一直在 get_totalticks 函数中死循环，那么重点看一下到底是哪里可能会导致死循环。

这个函数中，有两处可能导致死循环，一处是：

```
//find the line for requested cpu faster to just iterate linefeeds?
if (which != -1) {
    int i;
    for (i = 0; i < which; i++) {
        if (fscanf(fh, "cpu%*d " DEC_64 " " DEC_64 " " DEC_64 " " DEC_64, &userTicks, &niceTicks, &syste
            fclose(fh);
            return -2;
        }
        next_line(fh);
    }
    n = fscanf(fh, "cpu%*d " DEC_64 " " DEC_64 " " DEC_64 " " DEC_64 "\n",
       &userTicks, &niceTicks, &systemTicks, &idleTicks);
}
```

另一处是 next_line()函数：

```
38: static void next_line(FILE *f) {
39:     while (fgetc(f) != '\n');
40: }
```

如果文件没有换行符，第二处就会一直死循环，容器中/proc/stat 的内容是由主机的 LXCFS 从相应的/proc 文件中获取的，由于 LXCFS 在读取主机/proc 文件内容时可能存在打开文件异常等问题，因此返回内容为空，导致 next_line 函数在 while()中的死循环。在 LXCFS 官方代码的实现中，如果发现打开文件异常等问题则会直接返回，返回内容为空。我们修改 LXCFS，如果发现 LXCFS 文件内容长度为 0，则直接返回一个'\n'字符，这样可以避免 next_line()的死循环。修改代码如下：

仅供非商业用途或交流学习使用


```
static int proc_stat_read(char *buf, size_t size, off_t offset,
        struct fuse_file_info *fi)
{
    struct fuse_context *fc = fuse_get_context();
    struct file_info *d = (struct file_info *)fi->fh;
```

File Name
Bindings.c
Bindings.h
Config.h
Cpuset.c
Cpusetrange.c

在 Bindings.c 的 proc_stat_read()函数中增加一段容错代码：

```
err:
    if(rv == 0) {
        buf[0] = '\n';
        rv = 1;
        lxcfs_error("%s\n", "Internal error: /proc/stat");
    }

    if (f)
        fclose(f);
    free(line);
    free(cpuset);
    free(cg);
    return rv;
} ? end proc_stat_read ?
```

当/proc/stat 文件发送错误或者为空时，始终返回"\n"，保证

```
static void next_line(FILE *f) {
    while (fgetc(f) != '\n');
}
```

修改后，返回了正确的值。

2．一次 OOM 问题的排查

线上容器 Redis 压测完毕后，运行其他程序，出现内存分配失败，使用内存的时候报 OOM，如图 8.17 所示。

```
[7180785.639634] odin-metrics invoked oom-killer: gfp_mask=0xd0, order=0, oom_score_adj=0
[7180785.640772] odin-metrics cpuset=e9605ef9cf58d248872a2f9c2992d26e0f8b299b923bb4705f5f4c7c7ef9b02c mems_allowed=0-1
[7180785.641817] CPU: 21 PID: 19149 Comm: odin-metrics Tainted: G        W  OE K------------   3.10.0-514.16.1.el7.x86_64
[7180785.642859] Hardware name: INSPUR SA5212M4/Shuyu, BIOS 4.0.7 03/29/2016
[7180785.642864]  ffff880bb2e89f60 000000005cb60e37 ffff880100aefcc0 ffffffff81686ac3
[7180785.642865]  ffff880100aefd50 ffffffff81681a6e ffff8804d5ff9d80 0000000000000001
[7180785.642866]  0000000000000000 0000000000000000 0000000000000046 ffffffff811847c6
[7180785.642867] Call Trace:
[7180785.642874]  [<ffffffff81686ac3>] dump_stack+0x19/0x1b
[7180785.642880]  [<ffffffff81681a6e>] dump_header+0x8e/0x225
[7180785.642886]  [<ffffffff811847c6>] ? find_lock_task_mm+0x56/0xc0
[7180785.642889]  [<ffffffff81184c7e>] oom_kill_process+0x24e/0x3c0
[7180785.642891]  [<ffffffff8118471d>] ? oom_unkillable_task+0xcd/0x120
[7180785.642896]  [<ffffffff81093c0e>] ? has_capability_noaudit+0x1e/0x30
[7180785.642899]  [<ffffffff811f3651>] mem_cgroup_oom_synchronize+0x551/0x580
[7180785.642901]  [<ffffffff811f2aa0>] ? mem_cgroup_charge_common+0xc0/0xc0
[7180785.642904]  [<ffffffff81185504>] pagefault_out_of_memory+0x14/0x90
[7180785.642905]  [<ffffffff8167f8da>] mm_fault_error+0x68/0x12b
[7180785.642909]  [<ffffffff81692885>] __do_page_fault+0x395/0x450
[7180785.642944]  [<ffffffffa0626c99>] ? xfs_blkdev_issue_flush+0x19/0x20 [xfs]
[7180785.642946]  [<ffffffff81692975>] do_page_fault+0x35/0x90
[7180785.642948]  [<ffffffff8168eb88>] page_fault+0x28/0x30
[7180785.642951] Task in /docker/e9605ef9cf58d248872a2f9c2992d26e0f8b299b923bb4705f5f4c7c7ef9b02c killed as a result of li
b4705f5f4c7c7ef9b02c
[7180785.642952] memory: usage 4194304kB, limit 4194304kB, failcnt 9519462
[7180785.642952] memory+swap: usage 4194304kB, limit 4194304kB, failcnt 30
[7180785.642953] kmem: usage 0kB, limit 9007199254740988kB, failcnt 0
[7180785.642969] Memory cgroup stats for /docker/e9605ef9cf58d248872a2f9c2992d26e0f8b299b923bb4705f5f4c7c7ef9b02c: cache:4
B inactive_anon:1487408KB active_anon:2706824KB inactive_file:0KB active_file:0KB unevictable:0KB
[7180785.642969] [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_score_adj name
[7180785.643138] [ 5121]     0  5121    16792     7395      36        0             0 systemd
[7180785.643141] [19654]    81 19654     9739      369      20        0          -900 dbus-daemon
[7180785.643143] [19761]     0 19761     4853      315      13        0             0 irqbalance
[7180785.643145] [20065]     0 20065    32558      445      20        0             0 smartd
[7180785.643146] [22312]     0 22312     6923      119      17        0             0 xinetd
```

图 8.17　OOM 示例

仅供非商业用途或交流学习使用

但是执行 echo　3>proc/sys/vm/*drop_caches* 后，系统 Cache 内存依旧没有释放出来。

使用 Redis 压测后，我们分析系统日志，发现有大量的日志写入了 / run/log / journal，经过分析，可以看出这些日志是在 Redis 压测的时候产生的，全是 Redis 的慢日志信息，如图 8.18 所示。

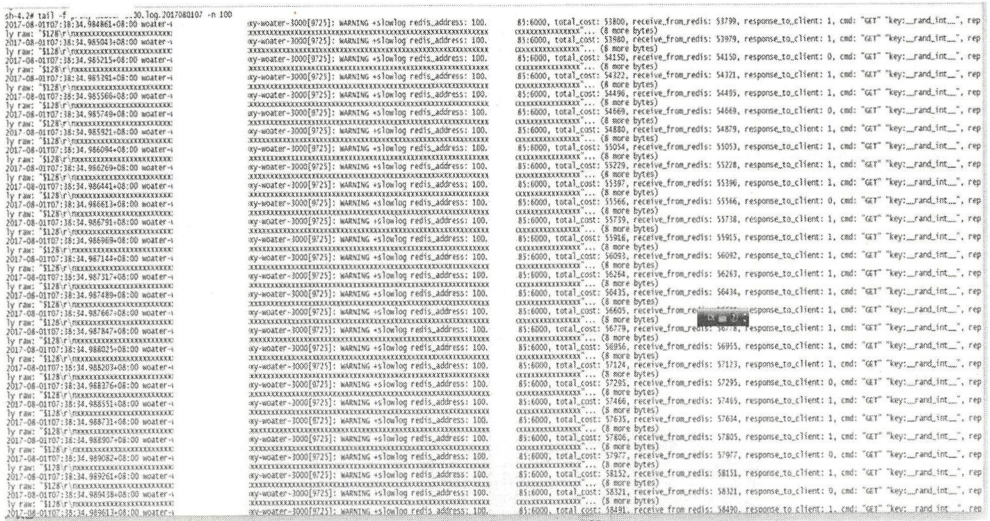


图 8.18　Redis 的慢日志示例

用 free 命令查看系统内存，如图 8.19 所示。

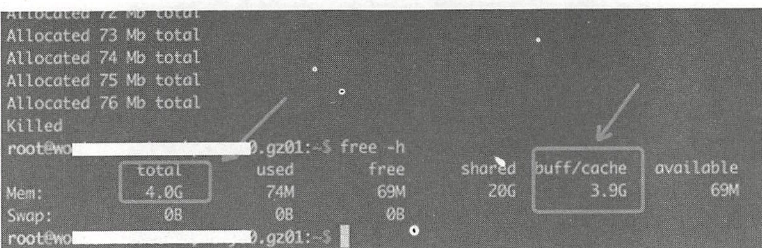


图 8.19　free 命令示例

从图 8.19 可知，大部分内存都被 Cache 占用，如果能分析出为什么 Cache 内存占用过高、由谁占用即可缩小问题范围。我们从压测现象了解到它会产生大量的 Redis 慢日志，因此怀疑 Cache 是由日志文件缓存引起的。

借助 Google Cache 分析工具 linux-fincore（该工具可以快速通过脚步扫描分析占用 Cache 的文件），得出如图 8.20 所示的分析结果。

仅供非商业用途或交流学习使用

filename	size	total_pages	min_cached page	cached_pages	cached_size	cached_perc
system.journal	240	1	0	1	4,096	100.00
system@000555a54c4a84db-bbfee6cc48d586fb.journal~	67,108,864	16,384	0	14,579	59,715,584	88.98
system@000555a568e7ec9a-5d3332180b6192fd.journal~	50,331,648	12,288	0	9,131	37,400,576	74.31
system@000555a57e6a45ae-af32585fb9da12f3.journal~	33,554,432	8,192	0	6,567	26,898,432	80.16
system@000555a5a5b8e743-576cc325e4e0a08c.journal~	25,165,824	6,144	0	4,391	17,985,536	71.47
system@000555ab11589ab2-78c257f546752ea5.journal~	25,165,824	6,144	0	4,894	20,045,824	79.65
system@000555bc556b66bc-38ab27392a6a03a0.journal~	50,331,648	12,288	0	9,859	40,382,464	80.23
system@000555bf5da3517a-0fb5086ea22ac6a7.journal~	16,777,216	4,096	0	2,851	11,677,696	69.60
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000000001-000554959a2b6a9a.journal	75,497,472	18,432	0	16,442	67,346,432	89.20
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000013e27-000554b81b767e5a.journal	83,886,080	20,480	0	17,035	69,775,360	83.18
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000027ad9-000554d85ffa6f97.journal	75,497,472	18,432	0	16,520	67,665,920	89.63
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000003b803-000554fc69ced546.journal	75,497,472	18,432	0	16,528	67,698,688	89.67
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000004f56f-0005552058daac89.journal	75,497,472	18,432	0	16,479	67,497,984	89.40
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000063210-000555449933fd42.journal	75,497,472	18,432	0	16,241	66,523,136	88.11
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000767fc-0005556a4daef4d3.journal	75,497,472	18,432	0	16,020	65,617,920	86.91
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000897f7-00055591429ce345.journal	92,274,688	22,528	0	20,977	85,921,792	93.12
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000009e097-0005559bc8a7a1e1.journal	125,829,120	30,720	0	23,221	95,113,216	75.59
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000b409a-0005559c09ef1a7e.journal	125,829,120	30,720	0	23,207	95,055,872	75.54
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000ca0fb-0005559c4ebd0065.journal	125,829,120	30,720	0	23,271	95,318,016	75.75
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000e0297-0005559c96acd8d2.journal	125,829,120	30,720	0	23,209	95,064,064	75.55
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000000f622d-0005559ce0bc79f8.journal	125,829,120	30,720	0	23,219	95,105,024	75.58
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000010c2dc-0005559d24b823c1.journal	125,829,120	30,720	0	23,218	95,100,928	75.58
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000012233b-0005559d69b32972.journal	125,829,120	30,720	0	23,260	95,272,960	75.72
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001384d7-0005559dafdfe3f8.journal	125,829,120	30,720	0	23,266	95,297,536	75.74
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000014e6f0-0005559dfd4fa694.journal	125,829,120	30,720	0	23,237	95,178,752	75.64
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000164836-0005559e4d8f736a.journal	125,829,120	30,720	0	23,274	95,330,304	75.76
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000017aae2-0005559e9cb0bd10.journal	125,829,120	30,720	0	23,199	95,023,104	75.52
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000190b21-0005559eeb5b3f31.journal	125,829,120	30,720	0	23,258	95,264,768	75.71
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001a6d79-0005559f3ead46de.journal	125,829,120	30,720	0	23,173	94,916,608	75.43
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001bcd4a-0005559f99845e47.journal	125,829,120	30,720	0	23,202	95,035,392	75.53
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001d2e60-000555a000e36503.journal	125,829,120	30,720	0	23,203	95,039,488	75.53
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001e8f5d-000555a06173116b.journal	125,829,120	30,720	0	23,190	94,986,240	75.49
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000001fefd6-000555a0c510cbe4.journal	125,829,120	30,720	0	23,209	95,064,064	75.55
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000002150f7-000555a12f2444b5.journal	125,829,120	30,720	0	23,151	94,826,496	75.36
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000022b0dd-000555a19e7f05e3.journal	125,829,120	30,720	0	23,189	94,982,144	75.49
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000024114b-000555a20e075413.journal	125,829,120	30,720	0	23,220	95,109,120	75.59
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000257256-000555a26ecdd448.journal	125,829,120	30,720	0	23,161	94,867,456	75.39
system@aaee2d0cc3634b1ab3df1b79c81ff554-000000000026d22d-000555a2d447521e.journal	125,829,120	30,720	0	23,217	95,096,832	75.58
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000283376-000555a3480246cc.journal	125,829,120	30,720	0	23,177	94,932,992	75.45
system@aaee2d0cc3634b1ab3df1b79c81ff554-0000000000299456-000555a3d5fceda5.journal	125,829,120	30,720	0	23,198	95,019,008	75.51
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000002af510-000555a43b836fd3.journal	125,829,120	30,720	0	23,194	95,002,624	75.50
system@aaee2d0cc3634b1ab3df1b79c81ff554-00000000002c5548-000555a4a9985542.journal	125,829,120	30,720	0	23,216	95,092,736	75.57

total cached size: 3,244,253,184

图 8.20　linux-fincore 分析结果示例

从中可以看出，journal 日志占用了几乎全部的 Cache 内存。journal 是一个功能强大的日志分析工具，具有历史日志时间端分析、过滤分析、尾部日志分析等多种功能。它的部分功能如图 8.21 所示。

```
# 查看系统本次启动的日志
$ sudo journalctl -b
$ sudo journalctl -b -0

# 查看上一次启动的日志（需更改设置）
$ sudo journalctl -b -1

# 查看指定时间的日志
$ sudo journalctl --since="2012-10-30 18:17:16"
$ sudo journalctl --since "20 min ago"
$ sudo journalctl --since yesterday
$ sudo journalctl --since "2015-01-10" --until "2015-01-11 0
$ sudo journalctl --since 09:00 --until "1 hour ago"

# 显示尾部的最新10行日志
$ sudo journalctl -n

# 显示尾部指定行数的日志
$ sudo journalctl -n 20

# 实时滚动显示最新日志
$ sudo journalctl -f

# 查看指定服务的日志
$ sudo journalctl /usr/lib/systemd/systemd

# 查看指定进程的日志
$ sudo journalctl _PID=1
```

图 8.21　journal 功能示例

Journald 和 systemd 是相互配合提供日志分析服务的。因此可以肯定，journal 日志占用过多 Cache 内存，并且一直占用，没有释放。删除目录 / run/log / journal 下面

仅供非商业用途或交流学习使用

的 journal 日志后，重新 free 查看 Cache 内存占用，已恢复正常，如图 8.22 所示。

	total	used	free	shared	buff/cache	available
Mem:	4096	7	4088	1481	0	4088
Swap:	4096	0	4096			

图 8.22　free 命令示例

通过内核调试诊断工具 systemtap，编写相应的 Cache 释放脚本，发现没有执行释放内核 Cache 释放流程，同时提示如图 8.23 中的信息。

我们通过三个测试例子，发现Linux系统内存中的cache并不是在所有情况下都能被释放当做空闲空间用的。并且也也明确了，即使可以释放cache，也并不是对系统来说没有成本的。总结一下要点，我们应该记得这样几点：

1. 当cache作为文件缓存被释放的时候会引发IO变高，这是cache加快文件访问速度所要付出的成本。
2. tmpfs中存储的文件会占用cache空间，除非文件删除否则这个cache不会被自动释放。
3. 使用shmget方式申请的共享内存会占用cache空间，除非共享内存被ipcrm或者shmdt，否则相关的cache空间都不会被自动释放。
4. 使用mmap方法申请的MAP_SHARED标志的内存会占用cache空间，除非进程将这段内存munmap，否则相关的cache空间都不会被自动释放。
5. 实际上shmget、mmap的共享内存，在内核层都是通过tmpfs实现的，tmpfs实现的存储用的都是cache。

图 8.23　Cache 使用注意事项

从中可知Cache问题已确定，原因是/run 目录挂载类型为内存型文件系统 tempfs，而 Redis 慢日志通过 syslog 写到该目录(/ run/log/journal 目录)，导致占用的 Cache 无法释放。

删除占用 Cache 的 journal 日志，问题解决。

以下总结了针对本问题的两种终极解决办法，任选一种即可。

第一，避免 syslog 日志写入 / run/log/journal 目录，可以通过修改配置实现，如下。

修改/etc/rsyslog.conf 配置：

```
$ModLoad imuxsock
#$ModLoad imjournal
#$IMJournalStateFile imjournal.state
#$OmitLocalLogging on
```

修改/etc/rsyslog.d/listen.conf 如下：

```
$SystemLogSocketName /dev/log
```

仅供非商业用途或交流学习使用

然后重启 journald 服务：sudo systemctl stop systemd-journald.socket

第二，通过限制/run 占用的 Cache 内存空间杜绝类似问题再次出现，如下。

执行 mount -t tmpfs tmpfs /run -o remount,size=1024M，或者修改/etc/fstab: 并添加 tmpfs /run tmpfs remount,size=1024M。

3．LXC 容器 cgroup 被修改的问题

线上 LXC 容器运行一段时间后，发现内存信息发生变化，例如给容器限定的是 4GB 内存，但是通过 free 查看到容器内存信息为宿主机的内存信息，如图 8.24 所示。

```
root@g                    gz01:/$ free -m
             total       used       free     shared  buff/cache   available
Mem:         63911        262      56117       6305        7531       56117
Swap:         7628          0       7628
root@g                    gz01:/$
```

图 8.24　free 命令示例

查看容器 init 进程对应的 cgroup 文件，截图（图 8.25）如下。

```
root                    .gz01:/hx                  us$ exit
exit
[yar                    slave05 ~]$ cat /proc/27835/cgroup
11:blkio:/user.slice
10:perf_event:/lxc/1505a18d02eb4cdebe9709874901f788-1
9:hugetlb:/lxc/1505a18d02eb4cdebe9709874901f788-1
8:cpuacct,cpu:/user.slice
7:devices:/user.slice
6:cpuset:/lxc/1505a18d02eb4cdebe9709874901f788-1
5:freezer:/lxc/1505a18d02eb4cdebe9709874901f788-1
4:net_prio,net_cls:/lxc/1505a18d02eb4cdebe9709874901f788-1
3:pids:/lxc/1505a18d02eb4cdebe9709874901f788-1
2:memory:/user.slice
1:name=systemd:/user.slice/user-51685.slice/session-176928.scope
```

图 8.25　cgroup 示例

从中可以看出，容器所属的 cgroup 控制组中的 blkio、cpu、device、memory 这几个子系统不是期望的信息，容器所属的这几个子系统发生了变化。

那么谁可能修改该文件？ 是应用层程序还是内核修改了该文件?

首先要确定该文件内容来源。通过走读 cgroup 内核代码发现，该文件内容通过查找/sys/fs/cgroup 控制组中各个子系统中的 tasks 文件内容来确定该进程所属的具体子系统，然后关联到/proc/pid/cgroup 文件，当我们 cat 该文件的时候即可获取该进程对应的子系统。

因此只要修改/sys/fs/cgroup 中相应子系统的 tasks 或者 cgroup.proc 文件，/proc/pid/cgroup 文件内容就会随之变化。

仅供非商业用途或交流学习使用

通过走读内核 tasks 文件接口，可以肯定内核只是简单提供读取操作，不会修改 tasks 内容，tasks 内容是由应用层下发至内核的。我们通过如下测试(如图 8.26 所示)，确定修改 tasks 文件内容后，cgroup 也会相应发生变化。

```
[                    u]# cat /proc/23643/cgroup
145:name=cgroup-yyztest:/
11:cpuacct,cpu:/lxc/yyz-test-3
10:cpuset:/lxc/yyz-test-3
9:perf_event:/lxc/yyz-test-3
8:memory:/lxc/yyz-test-3
7:blkio:/lxc/yyz-test-3
6:freezer:/lxc/yyz-test-3
5:devices:/lxc/yyz-test-3
4:net_prio,net_cls:/lxc/yyz-test-3
3:pids:/lxc/yyz-test-3
2:hugetlb:/lxc/yyz-test-3
1:name=systemd:/lxc/yyz-test-3
[r                    ]#
[r                    ]# echo 23643 > /sys/fs/cgroup/memory/user.slice/tasks
[r                    ]#
[r                    ]# echo 23643 > /sys/fs/cgroup/blkio/user.slice/tasks
[r                    ]# cat /proc/23643/cgroup
145:name=cgroup-yyztest:/
11:cpuacct,cpu:/lxc/yyz-test-3
10:cpuset:/lxc/yyz-test-3
9:perf_event:/lxc/yyz-test-3
8:memory:/user.slice
7:blkio:/user.slice
6:freezer:/lxc/yyz-test-3
5:devices:/lxc/yyz-test-3
4:net_prio,net_cls:/lxc/yyz-test-3
3:pids:/lxc/yyz-test-3
2:hugetlb:/lxc/yyz-test-3
1:name=systemd:/lxc/yyz-test-3
```

图 8.26 当前 cgroup 示例

从中可以证实，修改进程所属的 tasks 文件最终会修改/proc/pid/cgroup 文件内容信息，从而导致容器所属资源信息的错误。

通过上面的分析，可以确定 cgroup 是由于应用程序修改控制组子系统相关的文件引起的，范围进一步缩小，只需要找到哪个应用程序做了修改操作即可解决。

走读应用层 LXC 代码，可以确定 LXC 只有在 start 启动阶段才有写 tasks 文件的可能性，lxc-start 运行过程中不存在修改文件的可能性，因此排除 LXC 应用程序。

仔细观察/sys/fs/cgroup 目录下面的各个子系统，发现所有子系统中都包含 mesos_executors.slice 子系统，同时发现线上部分机器 mesos 相关的 cgroup 文件内容也有类似规律，如图 8.27 所示。

```
cgroup              coredump_filter  exe              io              maps
[                   ve154 ~]$ cat /proc/1441/cgroup
11:blkio:/system.slice/mesos-slave.service
10:pids:/
9:memory:/system.slice/mesos-slave.service
8:cpuset:/
7:devices:/system.slice/mesos-slave.service
6:cpuacct,cpu:/system.slice/mesos-slave.service
5:perf_event:/
4:freezer:/
3:hugetlb:/
2:net_prio,net_cls:/
1:name=systemd:/system.slice/mesos-slave.service
[y                   e154 ~]$
```

图 8.27 1441 进程 cgroup 示例

仅供非商业用途或交流学习使用

于是上 gtihub 搜索 mesos 代码，发现其中确实有 mesos_executors.slice 字符串，如图 8.28 所示。

```
Results in apache/mesos

src/linux/systemd.hpp
Showing the top four matches   Last indexed on 11 Jun

30   // stout, and leaving the mesos specific behavior here.
31   namespace mesos {
32
33   /**
34    * The systemd slice which we use to extend the life of any process

40   // TODO(jmlvanre): We may want to allow this to be configured.
41   static const char MESOS_EXECUTORS_SLICE[] = "mesos_executors.slice";
```

图 8.28　部分 mesos 代码说明

于是怀疑是否和 mesos 有关？再到 Mesos 官网 bug 系统查找 cgroup 相关的 bug，找到答案如下(意思是 systemd 会对通过 systemd 启动的程序的 cgruop 做迁移操作，可能会修改 cgroup 文件)：

https://issues.apache.org/jira/browse/MESOS-3352

https://issues.apache.org/jira/browse/MESOS-3425

接着再到 github 的 LXC 仓库和 Docker 仓库搜索 delegate 字符串，发现其 service 配置文件中都默认包含配置项 delegate=true，而我们线上却不包含该配置项，同时从 Docker 的 service 配置文件中可以看到如下提示信息：

```
# set delegate yes so that systemd does not reset the cgroups of docker containers
Delegate=yes
```

进一步确认 systemd 会修改 cgroups 文件。

走读 systemd 代码，最终确认了问题产生的原因，得出解决办法：在 service 配置文件中添加配置项 delegate=true，避免 systemd 修改 cgroup 信息。

8.6　总结

本章针对应用所依赖的基础设施层介绍了一些优化思路，相比应用层的优化，这一层的优化效果更明显、通用性也更强，但是改造成本相应也会更高。本章涉及的内容其实比较多，要把全部内容讲清楚比较难，主要还是以思路介绍为主，并提供了几个实际问题的分析案例，希望对大家有所帮助。

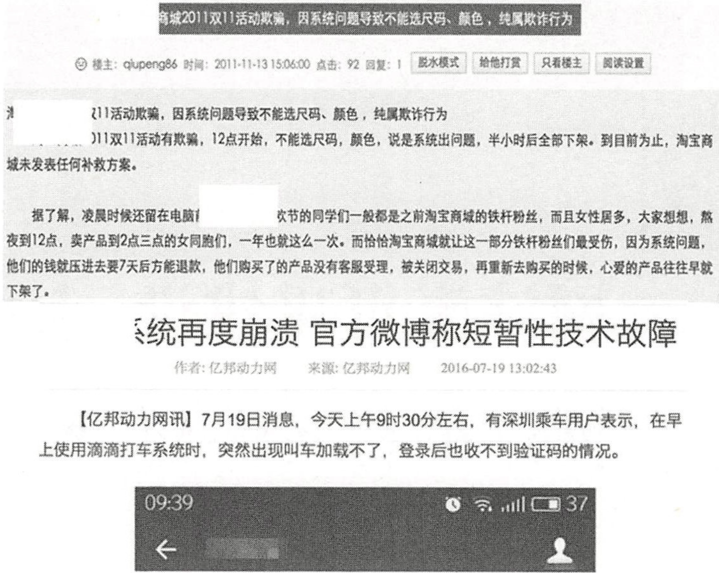
9 网站高可用建设：大型网站的稳定性建设

稳定性是决定网站生死的命脉。网站运行良好时大家可能感受不到稳定性的重要，一旦出问题那可真是够得上“砍头”的，尤其是用户数据出问题，更是一场灾难！稳定性渗透在网站运行的各个环节（包括整个研发过程），甚至有些时候不受我们的控制，比如光纤被挖断、机房断电等。

9.1 故障带来的影响

大家还记得 2011 年双 11 高峰期商品尺码丢失的故障吗？买家的订单中没有内衣的尺寸，卖家为此主动致电买家询问尺寸，被买家臭骂流氓。

和图 9.1 中类似的问题并不少，一个故障影响成千上万的买家和卖家，影响面远远超出想象，导致极差的用户体验、严重影响公司声誉。



商城2011双11活动欺骗，因系统问题导致不能选尺码、颜色，纯属欺诈行为

楼主：qiupeng86 时间：2011-11-13 15:06:00 点击：92 回复：1 脱水模式 给他打赏 只看楼主 阅读设置

双11活动欺骗，因系统问题导致不能选尺码、颜色，纯属欺诈行为

011双11活动有欺骗，12点开始，不能选尺码，颜色，说是系统出问题，半小时后全部下架。到目前为止，淘宝商城未发表任何补救方案。

据了解，凌晨时候还留在电脑 欢节的同学们一般都是之前淘宝商城的铁杆粉丝，而且女性居多，大家想想，熬夜到12点，卖产品到2点三点的女同胞们，一年也就这么一次。而恰恰淘宝商城就让这一部分铁杆粉丝们最受伤，因为系统问题，他们的钱就压进去要7天后方能退款，他们购买了的产品没有客服受理，被关闭交易，再重新去购买的时候，心爱的产品往往早就下架了。

系统再度崩溃 官方微博称短暂性技术故障

作者：亿邦动力网 来源：亿邦动力网 2016-07-19 13:02:43

【亿邦动力网讯】7月19日消息，今天上午9时30分左右，有深圳乘车用户表示，在早上使用滴滴打车系统时，突然出现叫车加载不了，登录后也收不到验证码的情况。

图 9.1 故障带来的影响

9.2 网站的可用性指标

网站可用性即网站正常运行时间的百分比，业界用 *N* 个 9 来量化可用性，最常说的是“4 个 9(99.99%)”的可用性。如图 9.2 所示。

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

图 9.2 网站的可用性指标（图片来源于网络）

网站的可用性如果能达到 4 个 9 基本上就算及格了，即网站一年的不可用时间不超过 52 分钟。为了保障整个网站的全部服务完全不出错，有必要对服务进行分级，以保障核心服务的高可用性。例如，电商网站的交易链路是核心流程，就必须保障交易链接的可用性达到 4 个 9 的目标。

故障时间有两项评估指标，一是平均无故障工作时间（MTBF，Mean time between Failures）；另一个是故障恢复时间（MTTR，Mean time to recover），所以网站的可用性可以用下面的公式表示。

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \times 100 = 99.\text{XXXX}\%$$

MTBF越长表示可靠性越高、正确工作的能力越强；MTTR越小表示恢复性越好。从中可以看出，要提升网站的可用性，就要延长MTBF、缩短MTTR。

我们根据图9.3来分析下哪些因素最容易导致故障。

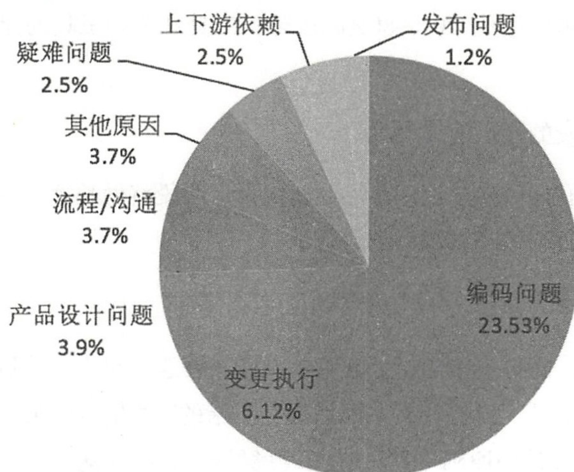


图9.3　导致故障的因素

从中可以看出，程序代码导致的问题占绝大部分，其中编码问题中需求场景考虑不足、流程沟通不到位、异常流处理缺失是产生问题的主要原因。

9.3　稳定性建设思路

网站稳定性的建设是一项综合的系统工程，就像人的健康一样，如果平时不注意健康饮食、不注意锻炼，时间一长身体肯定会出问题，对稳定性的考量也是贯穿整个研发生命周期的，如图9.4所示。

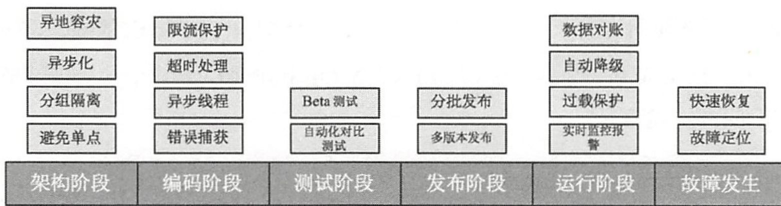


图 9.4 对稳定性的考量深入各个环节

如图 9.4 所示，在网站的架构设计时就要考虑稳定性：考虑到网络光纤有可能被挖断，如果只有一个机房那就悲剧了，100%的流量都没了，甚至要考虑自然灾害的影响而多地建设机房。

稳定性的建设中有两个重要因素：一是思想上重视，开发人员对稳定性的重视（也就是敬畏之心）可以避免 70%~80%的故障；二是规范和工具的建设，用以保障稳定性。

（1）架构阶段的稳定性建设项目

一个网站要有好的稳定性，必须在架构设计阶段就做长远考虑，就像建房子要先打好地基一样，否则楼建得越高越危险。

建设高稳定性的架构必须注意以下几点。

- 避免单点。高可用架构设计的第一条就是要避免单点，从概率出发来看，无论是机器还是人，没有什么东西是不会犯错的，尽量避免某个完整的环节成为单点是架构设计的原则。但是，在某些场景中，有些功能必须放在同一个应用中，例如统一登录、网关、安全过滤等，此时要保证尽量按照人群来划分，不要在逻辑上出现单点；或者尽量把服务拆成组件迁到不同的端上执行，例如，可以把安全服务作为一个模块集成到 Nginx 或者应用机器中，而不是把它做成一个远程服务接口。
- 分组隔离。把应用拆得更细一点，不同的功能模块做成单独的分组，每个分组占用一部分机器集群，这样可以做到服务分组隔离，避免不太重要的分组对重要分组的影响。现在非常流行的微服务就是一种非常好的分组隔离的实践。
- 异步化。在系统设计中要让尽可能多的远程调用异步化，这也是一项重要原则。很多时候会因为一个不太重要的功能的强依赖拖垮了整个应用，所以要尽量把不太重要的依赖改成异步调用，避免影响主调用链路的稳定性。
- 异地容灾。考虑到很多不可抗力因素的影响，我们需要设计网站的异地容灾甚

至全球部署策略，这些不可抗力的事件往往会影响网站的整个架构，必须做长远打算。

（2）编码阶段的稳定性建设

编程阶段的稳定性建设尤其重要，一个好的程序在编码阶段就决定了整个应用系统的质量，所以在编码阶段要注意下面一些规则。

- 错误捕获。一个优雅的系统必然有一套优雅的异常处理机制，在适当的地方如 I/O 处理、远程调用、多线程等关键处捕获异常非常重要，防御性编程可以更好地保证系统的健壮性。
- 异步线程。在一些批处理调用的地方采用异步线程可以保证主请求的正常返回，同样也可以做到部分的隔离，防止部分请求挂起整个应用。
- 超时处理。在远程调用或者调用外围设备时，非常有必要设定超时时间，这样可以保证所有的请求都有一个可预知的返回结果。
- 限流保护。每个应用都有一个承载极限，超过这个极限就会带来很大的不确定性，因此，设置自我保护机制可以保证程序的健壮性。

（3）测试阶段的稳定性建设

测试是程序上线前的最后一道保障，测试也是验证程序是否达到预期功能的手段，在本阶段要注意以下事项。

- 自动化对比测试。对比测试就是用线上真实的环境和数据与预发环境对相同的业务接口做返回值的比较，以此判断新上线的代码是否符合预期。
- Beta 测试。在线上的真实环境中选取若干台机器，通过绑定 VIP 的方式访问此机器上的数据，以此验证程序是否正确。

（4）发布阶段的稳定性建设

发布阶段稳定性建设应留意以下事项。

- 分批发布。分批发布可以降低发布风险，不仅可以减少系统部署重启引起的 RT 抖动，也可以在发现问题时立即终止发布。
- 多版本发布。合并部署是将多个应用系统同时部署在同一个 Web 容器实例中、共享同一个进程，每个应用之间相互隔离，但是应用之间的 RPC 是通过本地调用而不通过网络调用。

（5）运行阶段的稳定性建设项目

运行阶段稳定性建设要做好几件事：实时监控报警、过载保护和自动降级、实时数据对账，核心原则是实时发现问题，提供必要的保护措施。

- 实时监控报警。线上监控必须包含系统监控，主要是监控服务器的 CPU、Load、磁盘、内存等一些系统指标的异常情况；应用监控主要是监控响应时间、QPS、异常错误等；业务监控主要是监控一些业务指标是否有异常（如实时的下单量、司机和乘客的在线数量业务指标）。
- 过载保护和自动降级。线上运行中的系统需要有一些保护措施，如系统的某些指标达到瓶颈时要有必要的保护；当 Load 达到系统的最高瓶颈时需要拒绝一些请求，以防止系统被压垮；涉及一些远程调用时可以设置最大并发数，一旦超过该阀值就自动 fast fail 以保护系统。
- 实时数据对账。这属于业务监控范围。在涉及一些敏感信息时，为了确保正确性，需要有实时的对账校验，最典型的就是资金数据，以及涉及跨单元数据复制时对数据一致性的对账检查。

（6）故障发生时的稳定性建设

一旦发生故障，最重要的就是快速止损、定位故障并快速恢复。按照经验，当故障发生时，第一反应就是快速回滚了解故障现象，根据故障现象判读故障原因，进而找出解决办法。

减少故障的定位时间和快速恢复策略在故障发生时是至关重要的。

- 故障定位。快速定位故障可以缩短故障的恢复时间。如何定位故障？据统计，90%的故障都是由变更所致，所以快速收集变更信息和线上机器的异常数据非常重要，但难点在于这些数据往往散落在各个地方，并且格式多样。
- 快速恢复。影响故障恢复时间的一是故障定位时间，二是所采用的快速恢复的手段。例如在多版本部署出现故障时，可以通过快速切换版本来恢复；在异地多活的情况下可以把流量切换到不同的单元来止损。

9.4 高可用体系化建设

除了从观念上重视系统研发生命周期的各个阶段以外，真正建设高可用的系统还

需要一整套工具体系的支撑，这套体系包括压测体系、管控体系、监控体系、恢复体系和度量体系。

1. 压测体系

压测体系是测试时发现问题的重要手段。压测除了能帮助发现功能异常外，还能发现一些平时不容易发现的问题，如当系统压力大时系统的表现情况、线上系统的容量配比是否合理、系统的容灾保护是否到位等。压测一般分为单系统压测和全链路压测，我们所说的压测是线上真实生产环境的压力测试。

（1）单系统压测

比较容易实现，有多种实现手段。

一种是引流的方式，就是将线上集群中的流量集中到少部分的机器上，当这些机器流量变大时就会达到瓶颈，就能得出单机的极限性能，根据单机的性能就能推算出整个集群的性能。由于是线上的真实用户的访问请求，这种引流的方式不会产生额外的测试数据，所以对读、写系统都合适。

另外一种是放大流量的方式。例如通过 TCPCopy 工具可以把一个请求 copy 出多个重复的请求；还有一个方式是针对页面类型这种系统，可以在页面中注入一些 JavaScript 脚本（new Image().src='http://item.beta.taobao.com/item.htm?id=${itemId}'），在请求这个页面时，会自动向服务端额外发送一个请求，这样可以让用户帮我们制造流量达到压测的目的。当然这种方式会产生压测数据，所以只适合读系统而不适合写系统。

（2）全链路压测

全链路压测是目前比较好的、可以制造出线上大流量的手段。它的优点在于能串联线上全部系统，并让每个系统同时达到流量峰值（尤其是公共系统），所以适用的场景更多，但实施成本相对较高。

全链路压测的技术难度并不大，技术手段主要有流量的制造、流量的标记、测试数据的处理，全链路压测的架构如图 9.5 所示。

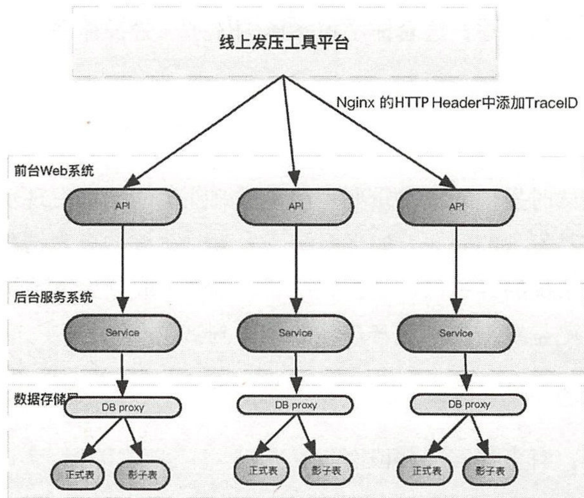


图 9.5　全链路压测架构示例

- 流量的制造

制造流量就是制造真实的用户请求。这对读系统来说比较容易，只要发生 HTTP GET 请求就可以，对写请求而言就需要构造数据了。例如下单操作需要填数量和收货地址等信息，因此需要有一个数据构造平台来构造符合业务流程的数据集合。电商系统的流量制造相对简单，只需要构造一些请求的 URL 集合，再通过流量引擎令这些请求发生即可。流量引擎如图 9.6 所示。

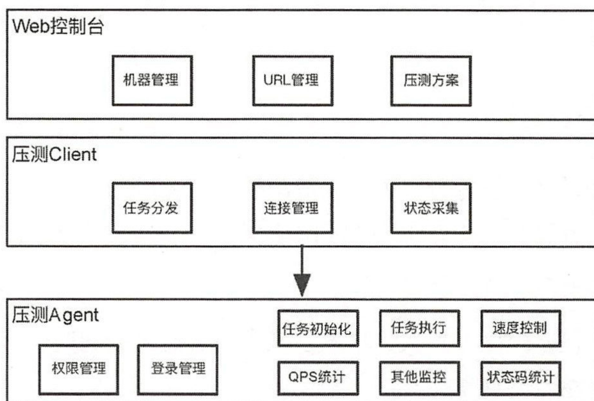


图 9.6　制造流量引擎

电商以外的系统像打车这种类型的系统，需要构建带有状态机的流量引擎来制造流量。

流量的制造除了要能够控制流量大小外，还要注意流量发起所在的物理网络位置，最好能保证流量是从不同的地理位置发起的，流量发起端可以部署在各地的 CDN 节点中，并把不同的网络运营商也考虑进来，这样能更好地模拟真实的用户请求。

- 流量的标记

流量制造出来以后，需要做标记，因为它们属于测试流量，它们的数据是测试数据，不能和线上的真实数据混同。

标记流量有多种方式：一种是设计不同的数据，例如商品数据的商品 ID 都以 9999 开头、把订单 ID 设置在某个特殊的区段内；第二种是给每个请求贴上一个 trace 标签，每个系统调用的地方都能识别出这个标签，一方面可以做一些特殊处理，如取消 token 验证，另外一方面也可以把它路由到特殊的数据表中。

trace 的传递是一个难题，最好的方式是通过中间件来完成：从最外面的 HTTP 协议将 trace 添加在 Header 中，到应用的 RPC 调用中也可以加到协议头中，再到数据层中也一样，最重要的是要保证 trace 标签不能被丢掉否则将出现脏数据。

- 测试数据的处理

对测试数据的处理是最关键的环节，因为是线上真实生产环境的压测，所以对产生的数据的处理不能影响线上的真实数据。对此，我们提出影子表的概念，它和线上的真实表完全一样，甚至和真实表一样用在同一个数据库实例中，通过传递下来的 trace 标记把这个请求需要写的数据路由到影子表中，这样测试流量的读写都在影子表中完成，不会影响线上的真实数据。将测试数据通过影子表来隔离是非常关键的，如果放在一起会引起很多麻烦，也不好清理，即使能够清理也会影响正式表的主键生成规则，影响下游 BI 对数据的分析和一些监控指标的展示等。

2. 管控体系

管控体系主要是在遇到一些异常情况时提供保护系统的措施，包括开关系统、预案系统、限流降级系统等。

（1）开关系统

开关系统主要是管理一些线上常用的操作，尤其是一些带有联动性的操作，通过统一管理可以减少出错的概率。开关系统既要支持基于内存和持久化的操作方式，也要支持单机和集群的灵活操作方式，如图 9.7 所示。

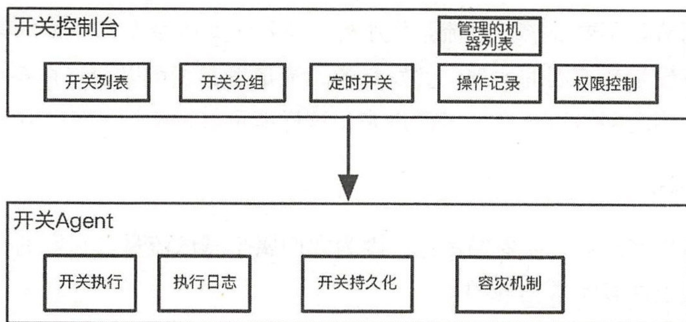


图 9.7　开关系统示例

（2）预案系统

预案系统整合业务降级开关，保证业务降级的一致性和完整性，保证所有应急预案处理统一调度、统一决策，信息互通、消除孤岛。如图 9.8 所示。

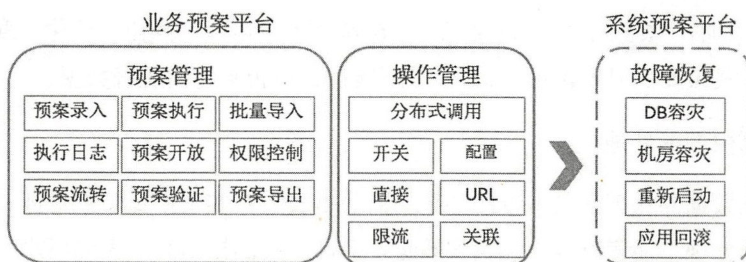


图 9.8　预案系统示例

（3）限流降级系统

当系统的容量达到瓶颈时，需要执行限流降级来保护系统，既要可以人工执行开关，也要支持自动化的保护；既要支持 URL 以及方法级别的细粒度的限流，也要可以基于 QPS、线程、Load 设置阀值。如图 9.9 所示。

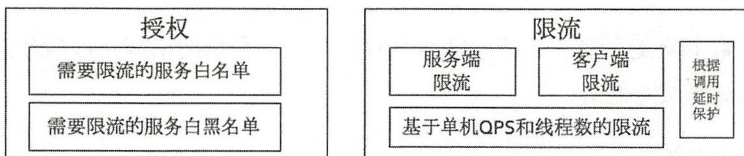


图 9.9　限流降级系统示例

3. 监控体系

监控体系是稳定性建设的必备措施，它可以分为多个子系统：异常的智能监控系统、调用链路跟踪系统、端到端的链路染色系统、业务数据轨迹重现系统、业务数据对账系统等。

（1）异常的智能监控系统

利用监控获取的基础数据，动态计算出实时异常指标范围。智能监控系统将系统内多个异常指标做关联、系统之间做关联、异常和系统变更事件做关联；用户可以给异常填反馈和打标，提供完整的异常特征如机房、事件段、历史上是否出现过以及异常堆栈等，如图 9.10 所示。

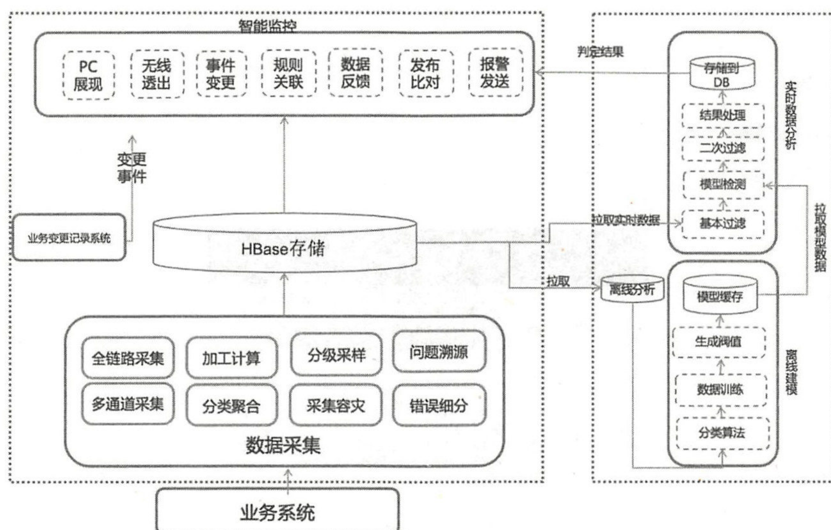


图 9.10 智能监控系统示例

这个智能监控系统可以将数据很好地关联在一起，可以快速发现整个调用链路上的异常点，如图 9.11 所示。

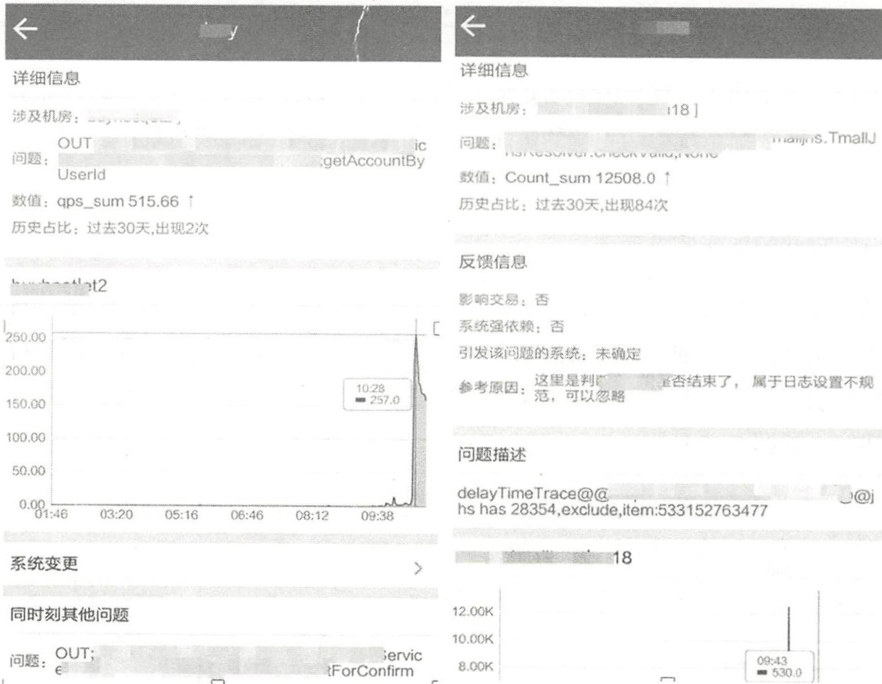


图 9.11　监控展示示例

（2）调用链路跟踪系统

调用链路跟踪系统可以记录一次调用所涉及的所有系统和数据库，以及每个调用节点的 QPS、RT 等，如图 9.12 所示。

http://[illegible]n/cart.htm 备注	carts						1,430 (at 14:59)	319.81
0-L0@queryItemAndSkuWithPVToText~LQ 备注	itemcenter[illegible]	[illegible]sk…	强依赖	P0	…	是	90 (at 15:13)	5.75
[illegible]etBaseUserByUserId~IS 备注	uicfinal	根据用户数字ID…	强依赖	P0	…	是	30 (at 12:20)	0.19
[illegible]ce@getExtraUserByUserId~IS 备注	uicfinal	根据用户数字ID获…	强依赖	P0	…	是	80 (at 12:41)	8.16
[illegible]eryPriceListAndPutIntoCache~LS	itemcente[illegible]	做全量查询，并讲…	强依赖				10 (at 11:02)	0.00
[illegible]queryMemberCartItems~IC	carts						1,300 (at 14:59)	264.47
GET:group_[illegible]	tair[illegible]	[illegible]的数据快…	弱依赖		…		180 (at 12:22)	0.39
GET:group_[illegible] 备注	tair[illegible]	[illegible]务平台…	弱依赖	P2	…	是	1,300 (at 13:00)	135.76
GET:group_[illegible]	tair[illegible]	[illegible]于存放和商品…	弱依赖	P1	无	是	190 (at 04:13)	0.83
GET:group_ses[illegible]00 备注	tair[illegible]	查询用户session…			…		1,330 (at 14:59)	267.26
GET:group_[illegible]	tair[illegible]1)						440 (at 19:21)	1.08
GET:grour[illegible]5 备注	tair[illegible])	[illegible]户基本信息…	强依赖	P0	…	是	3,080 (at 15:59)	564.49
PREFIX_GET_HIDDENS:group_[illegible]2 备注	tair[illegible]m)	[illegible]前置缓存场…	强依赖	P0	…	是	22,160 (at 10:32)	566.95
item[illegible]L0@queryItemById~IQA 备注	itemcente[illegible]	根据id获取商品记录	强依赖	P0		是	190 (at 15:26)	0.12
[illegible]e@getBaseUserListByUserIds~MS 备注	uicfi[illegible]	根据用户数字ID(u…	弱依赖	P0	…	是	30 (at 09:55)	0.74
GET:group_[illegible]	tair[illegible])	[illegible]务对象	弱依赖	P2		是	310 (at 10:30)	9.25
GET:group_[illegible]	tair[illegible])	[illegible]toryp…	弱依赖	P2	…	是	380 (at 04:13)	1.65

图 9.12　链路跟踪示例

它的实现原理比较简单：每个请求中带有一个 trace 信息，把 trace 一直携带到该请求的所有调用链路上，打印到日志中，并通过日志中的 trace 关联信息形成一个调用栈。

链路跟踪系统还可以实现链路染色功能，可以根据用户 ID 或者商品 ID，记录用户端的所有操作，并上传（request、response），和服务端的整个调用链路做关联。

如图 9.13 所示，服务端把染色标记推送给 App，App 将染色标记关联到某个用户，然后 App 会记录下这个用户的所有操作记录日志（包括向服务端发送的请求），调用服务端的请求也会带上特定的标识，在服务端通过链路追踪系统，把这个请求的所有依赖都记录下来，同时 App 也会将端上用户的操作记录上传到服务端，这样染色系统就同时有了 App 用户的操作日志和服务端的链路调用日志，就可以知道被染色的用户的整个操作是否有异常了，便于排查疑难问题。

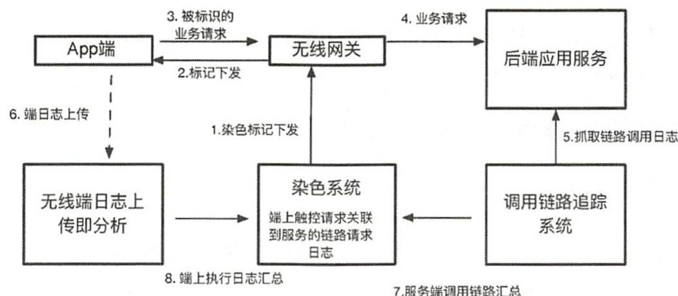


图 9.13　染色系统示例

（3）业务数据轨迹重现系统

所谓业务数据轨迹重现就是可以追踪数据的所有变更记录，了解是谁在哪台机器上调了哪个接口、在哪个时间点从某个数据变更成另一个数据的完整记录。它的逻辑结构如图 9.14 所示。

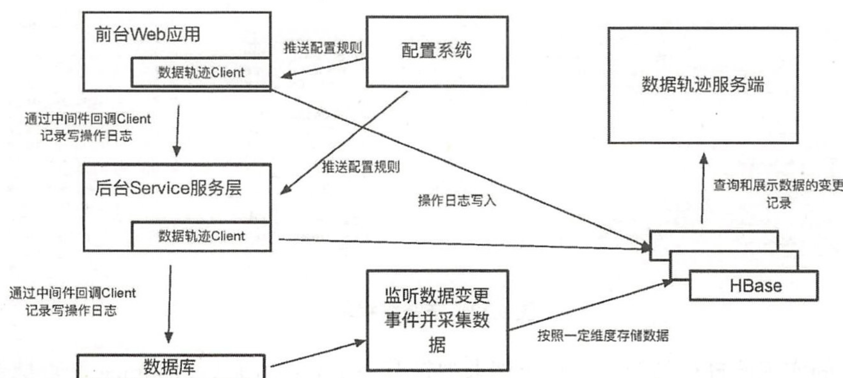


图 9.14　数据轨迹系统示例

数据轨迹系统主要是记录写操作的操作日志，它是如何实现的呢？答案是依赖中间件。我们在中间件里可以添加一些钩子，把钩子封装在 Client 中，通过配置系统推送一些规则，要控制哪些请求需要记录日志，如在给更新商品信息这个接口需要记录下请求接口的参数信息，那么 RPC 中间件就会打印这些请求的日志，然后通过上面介绍的链路追踪系统把这次请求加上标记，一直追踪到数据库上，所经过的每个系统都会记录此数据的变更记录，在数据库上是数据的最终变化。这里需要另外一个中间件，它需要解析数据库的 binglog、感知数据的 update 以及 delete 的操作，并记录下变更之前和之后的数据，最终这些变更记录都会写到 HBase 里，便于查询和检索。例如可以按照商品维度去查询一段时间内对商品都做过哪些修改操作，这些操作都有详细的操作记录，如果发生错误也可以追溯原因。

按照这个思路，可以实现商品的快照功能。假如你购买的商品包含卖家的承诺，在你下单后，卖家删除了该承诺，这会使订单关联一个商品的变更记录，通过它可以知道卖家在什么时间点修改了商品的描述信息。

（4）数据对账系统

数据对账系统在很多场景下都很有用，例如在跨机房数据复制的场景下，可以分别采样一定时间内、两边数据库的数据进行比较，了解是否有超出预期的延迟、是否有丢失数据等问题；在需要跨系统状态同步的场景下也可以对账比较，例如确认打款和退款后，主订单打款金额和退款金额之和要等于总金额。

数据对账系统的原理如图 9.15 所示。首先需要对接各个数据源，采集实时数据（数据采集一般是通过接受业务的异步消息以及监听数据库的变更消息等方式获取的），通过数据转换把原始数据转化成需要的数据格式以适配写好的规则，并匹配执行。执行规则一般就是对数据进行校验，如果发现校验结果错误就会触发报警或者转到数据订正系统去订正。

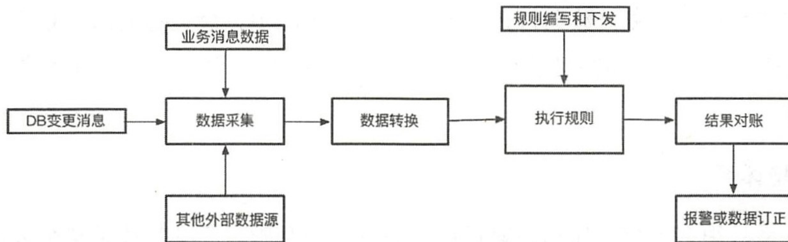


图 9.15　数据对账示例

此系统有两个最关键的部分：一是数据的采集部分。这部分最好与业务系统解耦，而且要便于扩展——便于实时增加新的数据采集源头；二是要能够比较方便地完成规则的编写和下发。规则一般用动态语言编写，用热加载的方式执行。如果需要比较的数据量较大或者较复杂，在执行规则部分还可以引入流式计算来完成。

4. 恢复体系

线上出现问题时，最简单的恢复方式就是回滚以及执行变更操作，如图 9.16 所示。我们前面介绍的多版本部署就是一种有效且快速的回滚方案。

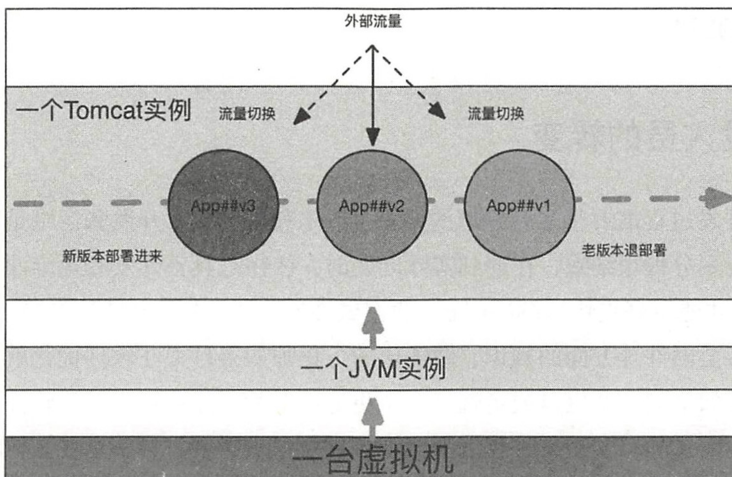


图 9.16　数据恢复示例

应用部署可以一批完成，并不依赖发布批次，所以系统发布可以从目前的 2 个小时降到 10 分钟，完成所有机器的代码替换；应用回滚从目前的 2 个小时降到 1 分钟之内完成，无需重新部署代码，只需要做代码版本切换即可。应用部署时 JVM 不需重启，代码热替换，对外服务不中断。

此外，一个快速的数据订正平台也是非常必要的，尤其在发生数据错乱的时候，平台可以加速数据的恢复。数据订正平台的实现比较简单，最关键的前提是要打通各个数据源，需要动态加载那些可以写数据的 Service 接口，以及一个写程序或者写 SQL 的执行引擎。

5. 度量体系

度量体系是很多系统（如动态容量评估系统）的基础。成本基线是整个公司的成本总和除以关键业务指标的值如订单量。

- 性能基线。需要制定每个业务系统的性能基线，便于做出正确的容量预估，有了这个基线就可以知道什么时候会导致系统的性能下降，配合链路基线可以分析出导致性能下降的是哪个依赖。
- 链路基线。链路基线在系统的依赖度比较高的时候非常有用，例如这个系统到底依赖了哪些外部系统，哪些是强依赖，哪些是弱依赖；依赖了多少外部系统、缓存以及数据库；依赖的深度是多少，分别执行的延时是多少……这对我们分析系统的性能数据非常有帮助。
- 成本基线。成本基线可以反映整个公司的成本情况，从中可以看出成本降低和提高的因素。

9.5 研发人员的转变

实际的开发过程需要很多跨领域的知识。以工程师为例，开发和运维是两个岗位的工作，岗位划分得很细致，在遇到实际问题时，往往是具备开发和运维两个领域知识的工程师，才能更好更快地解决问题。我们做性能优化，也需要具备前端、服务端、网络、运维甚至硬件等方面的知识，做优化的工程师如果只关注软件优化就无法明白硬件的变化（如增加一个磁盘或者选择一个更好的芯片）能达到比单纯软件优化更好的性能；思维模式的转变在某种程度上比知识结构还更重要，具备全栈知识结构的工程师容易有更宽阔的思维模式，更有可能制定出平衡的方案。

（1）向全栈工程师转变

应用开发工程师不仅要能开发前端、无线和后端的代码，还要具备线上运维能力，包括：

- 排查浏览器端的前端问题；
- 排查域名解析和 DNS 劫持等常见问题；
- 解决前端 JS 的错误定位和资源加载限制问题；
- 掌握无线端的网络特性：无线情况下建立 TCP 连接的耗时、数据下载的影响、WiFi 和 4G 以及弱网络下的内容适度匹配等；
- 掌握无线端请求的全链路过程：包括手机端到基站的接入，从省级网关出口到服务端网关再到应用系统等；
- 排查无线端的问题：链路染色、日志上报、舆情收集等；
- 掌握无线端的开发技能；
- 理解服务端 Nginx、Cache、Tomcat 等服务器的配置文件；
- 理解 JDK 的基本配置参数、内存分配方式和 GC 调优；
- 具备线上 Java 运行环境的排查技能。包括一些开源中间件的报错定位、操作系统的端口冲突、JVM 的异常退出等；
- 掌握线上应用系统的性能指标，包括网络、QPS、RT、线程级的 CPU 消耗、Load、内存的 dump；
- 熟悉线上网络部署架构，应用服务器和交换机的连接情况、跨机房和跨单元之间的网络情况；
- 熟悉 CDN 的部署分布；
- 熟悉源站 DNS 的解析步骤，包括 VIP 的管理、LVS 的流量分配、应用服务器之间的健康检查机制等。

以上是工作中经常会遇到的问题和常用的技能知识，掌握这些内容对排查线上问题非常有帮助。

（2）向全链路运维转变

以前开发人员参与线上运维通常只会关注应用系统本身的问题和业务日志错误信息，较少关注整个应用的全链路问题，所以当问题出现时，由于信息掌握不全，很难具体定位问题。工程师的关注点应从单系统转向全链路，必须掌握如下内容。

- 掌握资源依赖的关联系统：页面的资源依赖情况、JS 和 CSS 如何发布到 CDN 节点以及图片的回源方式等；
- 掌握请求链路：掌握无线请求链路上每个关键环节的信息透出，能够根据关系数据追踪请求轨迹；页面的输出、信息是如何聚合的、动态内容和静态内容、页面中的异步加载等；请求的来源和请求的去向。

（3）向工具化和规划化转变

工具化能提升效率和把不规范的流程程序化，减少人为操作的出错概率，它包括如下内容。

- 信息的输出要标准化和规范化，包括端上的 Log、请求跟踪、应用日志输出格式等；
- 数据的采集要工具化、集中化；
- 数据的统计和分析要多维度化；
- 数据的展示可以个性化和可视化；
- 建立长期可以跟踪变化的基线数据，包括性能、成本、链路变更基线。

9.6 稳定性组织保障

稳定性建设是比较难的任务，如果平时运行良好，很难申请到资源支持；但是一旦出问题，压力就非常大，稳定性保障是个苦差事。

然而，稳定性出现问题尤其是数据出现问题，后果会非常严重，在涉及资金时甚至有可能是致命的，因此，有必要从公司层面建立起一套稳定性的保障体系，这涉及组织的建设，如图 9.17 所示，可以成立一个实体或者虚拟团队负责公司的整体稳定性，通过一些抓手落实稳定性的体系化建设。

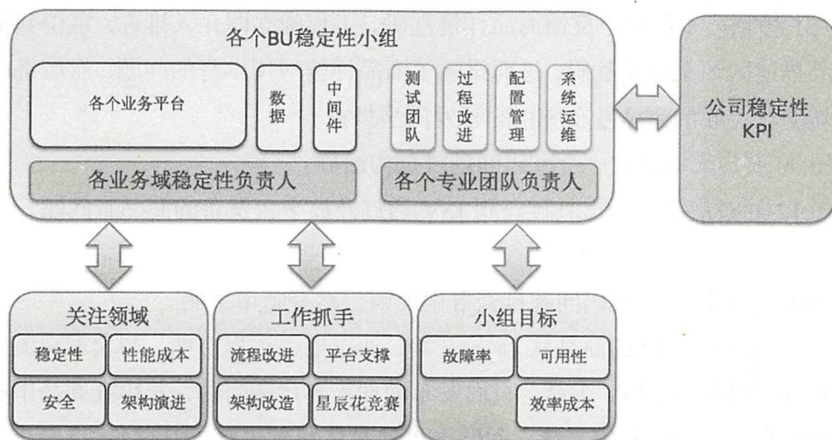


图 9.17　稳定性组织保障示例

9.7　疑难问题排查思路

程序员经常会遇到一些诡异的线上问题，这些问题一般很难得到重现，而且不容易找到原因。本节介绍一些排查疑难问题的思路。

1．商品详情乱码问题

用户反馈详情页面的描述部分有乱码，反复刷新可以重现；只有部分商品是乱码，重新发布该商品后不能重现乱码。如图 9.18 所示。

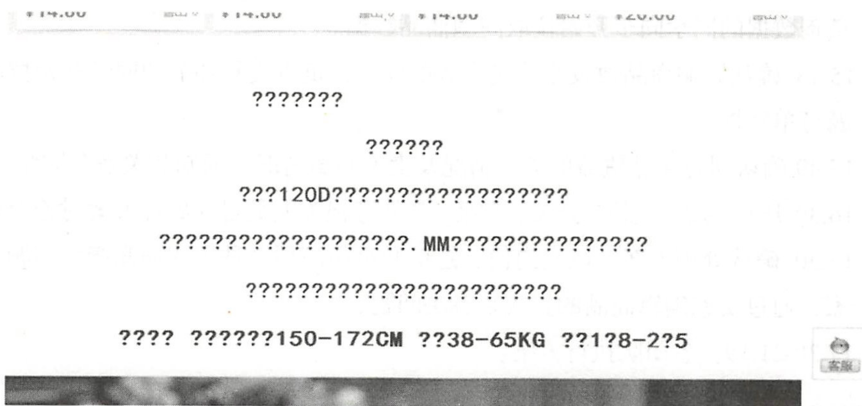


图 9.18　页面乱码示例

故障处理过程如下。

- 9:35 接到第 1 例用户反馈商品详情乱码，工程师立即介入排查，定位到保存商品描述的源文件是乱码，进而排除了商品详情应用本身的问题，然后将此问题转交给关联的商品发布应用的开发同学排查。
- 10:00 召集大家建立一个乱码问题排查沟通群。
- 10:12 通过商品后台工具排查到 ISV 通过开放平台发布的商品有问题，未找到根本原因。
- 10:30 客服同学在乱码问题排查群里反馈仍然有新增案例。
- 11:00 发现有问题的商品从 28 号开始，通过电话远程协助，与卖家交流，分别排除了浏览器、ISV 工具、页面发布助理等客户端问题，并排查商品中心、开放平台、商品发布等系统，问题聚焦到商品发布和商品中心。
- 11:10 发现应用发布之后，若干台机器的编码会出现异常，开始怀疑是发布系统调度切换引起的问题。
- 11:18 将 5%的流量回切，全部采用老的发布方式。先确保新发布的应用系统不会产生乱码。
- 11:36 定位到 2 台商品中心服务器的 boot.log 有英文编码，立刻通知 PE 同学关闭这 2 台服务器。
- 11:36 PE 同学确认商品中心 2 台编码有问题的机器，并下线机器，阻止问题数据的产生（不会产生新的乱码数据）。
- 12:20 确认是由于发布系统的调度切换导致。
- 12:30 确认乱码问题影响的数据范围只出现在商品描述和食品安全的数据中，负责数据的同学同步开始拉取问题商品数据。
- 15:13 确认影响商品数及相关卖家数的同时，负责交易环节的同学开始拉取问题订单数据。
- 15:49 确认问题商品覆盖的类目情况及卖家归属情况，通知相关业务同学。
- 16:30 开发、技术支持、公关、客服、产品等相关人员召开处理方案讨论会议。
- 17:30 确认处理方案：以站内信、短信和页面吊顶三种方式通知产生问题的卖家，通过卖家编辑商品的方式来解决问题。
- 17:30-21:19 各团队执行方案。

问题分析如下。

- 能够重现乱码，就可以排除浏览器端渲染的问题，而有可能是服务器端的发布系统有问题；

- 发布端系统包括商品发布系统、页面发布助理以及第三方发布工具，到底是哪个出问题；
- 以上几个问题排除后，仍然都不能重现；
- 怀疑部分机器有问题，开始排查线上所有机器的配置，确定它们是否都一样；
- 排查发现有两台机器的编码不一致，写了个 JSP 页面调用；
- 立即下线两台机器；
- 为什么有两台机器编码不一致？

排查原因如下。

- 代码和应用环境都一致，到底是谁改了编码；
- 最终发现是发布系统修改的 LC_ALL，而这个优先级最高；
- 发现问题后要分析脏数据产生的影响；
- 排查数据问题可能会产生交易纠纷，如订单数据乱码导致无法发货等。

经验总结如下。

- 不要放过任何很小的错误；
- 涉及数据问题，一定要非常重视；
- 必须熟悉整个商品流量环节；
- 要写健壮性的代码，不给自己挖坑。

2. Mina 内存泄露问题

某系统在运行 20~30 分钟后，系统 swap 区突增，直到 swap 区使用率达到 100%，机器死机，如图 9.19 所示。

```
----total-cpu-usage---- -dsk/total- ---load-avg--- ------memory-usage----- -net/total- -swp/total-
usr sys idl wai hiq siq| read  writ| 1m   5m  15m | used  buff  cach  free| recv  send| used  free
  3  13  85   0   0   0|   0     0 |  0  0.1  0.1|3358M   37M  433M   12M|  32k   42k|  17M 1010M
  4  12  85   0   0   0|   0   448k|  0  0.1  0.1|3385M   29M  414M   12M|  41k   48k|  17M 1010M
  2  13  84   0   0   0|   0  1232k|  0  0.1  0.1|3414M   23M  391M   12M|  41k   48k|  17M 1010M
  3  14  83   0   0   1|   0     0 |  0  0.1  0.1|3444M   22M  362M   12M|  33k   43k|  17M 1010M
  2  21  77   0   0   0|   0    56k|  0  0.1  0.1|3474M   22M  333M   11M|  38k   46k|  17M 1010M
  3  18  79   0   0   0|   0     0 |  0  0.1  0.1|3505M   22M  303M   10M|  36k   49k|  17M 1010M
 16  18  66   0   0   0|   0    72k|  0  0.1  0.1|3542M   22M  266M   11M|  36k   38k|  17M 1010M
 39  44  17   0   0   0|   0  1128k|  0  0.1  0.1|3614M   16M  198M   12M|  19k   41k|  17M 1010M
  3  21  74   1   0   0|   0     0 |  0  0.1  0.1|3643M 9264k  176M   12M|  34k   37k|  17M 1010M
  4  17  79   0   0   0|   0   744k|  0  0.1  0.1|3672M 8036k  148M   12M|  45k   59k|  17M 1010M
  3  12  85   0   0   0|   0     0 |  0  0.1  0.1|3700M 6144k  122M   12M|  37k   39k|  17M 1010M
  3  10  87   0   0   0|8192B   32k|  0  0.1  0.1|3728M 3612k   97M   12M|  43k   58k|  17M 1010M
  5   9  87   0   0   0|   0  1352k|  0  0.1  0.1|3759M 3592k   67M   11M|  39k   49k|  17M 1010M
  3   7  89   0   0   0|   0  8192B|  0  0.1  0.1|3786M  548k   42M   12M|  39k   47k|  18M 1010M
  4  10  87   0   0   0|   0    59M|  0  0.1  0.1|3789M  140k   39M   12M|  46k   54k|  47M  980M
  3  13  81   4   0   0| 464k   62M|  0  0.1  0.1|3790M  152k   39M   12M|  41k   50k|  78M  950M
  2  12  86   0   0   0|   0    60M|  0  0.1  0.1|3790M  160k   39M   11M|  32k   40k| 108M  920M
  4  11  85   0   0   0|   0    63M|  0  0.1  0.1|3790M  148k   39M   12M|  40k   47k| 139M  888M
  7  14  73   6   0   0|4088k   69M|  0  0.1  0.1|3791M  140k   38M  9.9M|  36k   46k| 174M  854M
 33  13   3  52   0   0|   0   137M|  0  0.1  0.1|3789M  144k   39M   12M|  13k   21k| 248M  780M
```

图 9.19　系统指标

系统内存已经达到 3.5GB，已经超过了 Heap 堆设置的上限，但是 GC 却很少，如图 9.20 所示。

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0.18	0.00	84.86	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	85.58	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	86.95	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	86.99	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	88.36	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	89.70	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	90.43	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	91.16	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	92.53	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	92.56	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	93.93	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	94.62	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	95.99	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	97.36	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	97.40	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	98.77	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	99.47	24.86	63.86	16	5.618	1	0.124	5.742

图 9.20　系统 GC 情况

但是 Old 区的空间也几乎没有变化，如图 9.21 所示。

[junshan@v101208.sqa.cm4 admin]$ sudo jstat -gcold 4158 1000

PC	PU	OC	OU	YGC	FGC	FGCT	GCT
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813

图 9.21　系统 Old 区情况

初步判断如下。

首先 Java 进程内存增长非常迅速：在压力测试 20 分钟时就用光 2GB 内存，并且耗光内存后开始使用 swap 区，迅速消耗了 swap 区的空间，导致死机。综上，可以判断肯定发生了内存泄露。

但是通过 jstat 分析 JVM Heap 堆情况和 GC 统计信息，发现 GC 很少尤其是几乎没有 Full GC，如果是 JVM 堆内存被耗光，应该 Full GC 非常频繁，所以初步判断这次内存泄露并不在 JVM 堆中。

为了进一步排查，我们通过 jmap dump 出内存快照，通过 Mat 分析内存数据占用情况，如图 9.22 所示。

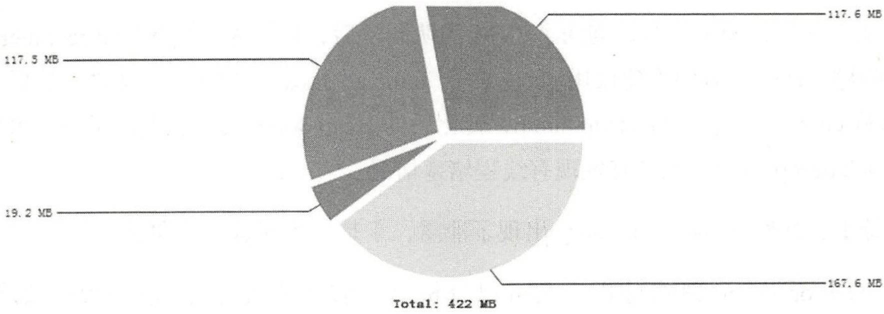


图 9.22 Mat 分析情况

这个堆只使用了近 500MB 内存，和 jstat 得出的堆信息是一致的，而两个最大的对象内容如图 9.23 所示。

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
org.apache.mina.transport.socket.nio.SocketSessionImpl @ 0x75b340588	296	123,340,192
<class> class org.apache.mina.transport.socket.nio.SocketSessionImp	0	0
ch sun.nio.ch.SocketChannelImpl @ 0x75b3404f0	176	448
key sun.nio.ch.SelectionKeyImpl @ 0x75b340560	64	64
writeRequestQueue java.util.concurrent.ConcurrentLinkedQueue @ 0x	32	123,336,840
<class> class java.util.concurrent.ConcurrentLinkedQueue @ 0x7ea	40	40
tail java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x72500f	32	192
head java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x75e24	32	123,336,616
<class> class java.util.concurrent.ConcurrentLinkedQueue$Node	24	24
item org.apache.mina.common.IoFilter$WriteRequest @ 0x75e2	40	224
next java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x75	32	123,336,360
<class> class java.util.concurrent.ConcurrentLinkedQueue$N	24	24
item org.apache.mina.common.IoFilter$WriteRequest @ 0x7	40	280
next java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x	32	123,336,048
Σ **Total: 3 entries**		
Σ **Total: 3 entries**		
Σ **Total: 3 entries**		

图 9.23 Mat 堆信息

都是 org.apache.mina.transport.socket.nio.SocketSessionImpl 对象，由于占用的空间不大（100MB）、持有的对象数也不多，没有引起注意。

于是可以得出结论：要么是 NIO direct memory、要么就是 native memory 内存泄露。我们用查看 NIO direct memory 的工具检查 direct memory 占用的空间大小，如图 9.24 所示。

```
[junshan@v101208.sqa.cm4 java]$ sudo java -classpath .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize 4158
Attaching to process ID 4158, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version 20.1-b02. You may see unexpected results.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NIO direct memory:
  reserved size = 25.984024MB (27246224 bytes)
  max size = 1984.000000MB (2080374784 bytes)
```

图 9.24 检查 direct memory 占用的空间

显示只有 25MB 左右，询问了 JVM 团队同学后，也认为不是 NIO direct memory 出了问题。此时，有同事建议用 gcore 命令 dump 出 Java 进程的 core 文件，然后通过 jmap 将 core dump 转化成 Heap dump，转化后的 Heap dump 文件和前面类似。另外通过 jstack dump 出内存也没有发现有线程堵塞情况。

综上，怀疑是 native memory 出现了泄露，于是开始朝这个方向排查。

使用 oprofiler 热点分析工具可以分析出当前系统执行的热点代码，如果是 native memory 有泄露，那么肯定会出现分配内存的代码是热点的情况，便使用 oprofiler 工具，分析当前 CPU 的消耗情况，如图 9.25 所示。

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples  %         app name                 symbol name
365070   98.4528   no-vmlinux               /no-vmlinux
1688      0.4552   anon (tgid:15394 range:0x2aaaaec01000-0x2aaaaef92000) anon (tgid:15394 range:0x2aaaaec01000·
305       0.0823   libperl.so               /usr/lib64/perl5/5.8.8/x86_64-linux-thread-multi/CORE/libperl.so
240       0.0647   libzip.so                inflate_fast
121       0.0326   libpthread-2.5.so        __write_nocancel
105       0.0283   anon (tgid:15394 range:0x2aaaaef92000-0x2aaab1c02000) anon (tgid:15394 range:0x2aaaaef92000·
102       0.0275   libzip.so                crc32
92        0.0248   libzip.so                huft_build
89        0.0240   libc-2.5.so              _int_malloc
64        0.0173   libjvm.so
        CardTableModRefBS::dirty_card_range_after_reset(MemRegion, bool, int)
63        0.0170   libjvm.so                IndexSetIterator::advance_and_next()
52        0.0140   libjvm.so                PhaseChaitin::Split(unsigned int)
41        0.0111   static-python            /home/tops/bin/static-python
40        0.0108   libjvm.so                PhaseChaitin::build_ifg_physical(ResourceArea*)
40        0.0108   libpthread-2.5.so        pthread_getspecific
37        0.0100   libjvm.so                Copy::fill_to_memory_atomic(void*, unsigned long, unsigned char)
35        0.0094   libjvm.so                jni_GetIntField
34        0.0092   libc-2.5.so              memcpy
34        0.0092   libjvm.so                SymbolTable::lookup(int, char const*, int, unsigned int)
32        0.0086   libjvm.so                PhaseChaitin::gather_lrg_masks(bool)
```

图 9.25　oprofiler 热点分析

从图中可以看出，结果和我们预想的并不吻合，于是采取土办法：通过一部分一部分去掉功能模块来检查到底是哪个模块导致的内存泄露。

通过删除可能会出问题的几个模块后，最后确定是调用 mina 框架给 varnish 发送失效请求时导致的问题，发送的请求数频率越高内存泄露越严重，但是 mina 框架没有使用 native memory 的地方，又陷入僵局。

再次使用 perftools 来分析 JVM 的 native 内存分配情况，通过 perftools 得到的分析结果如图 9.26 所示。

```
[junshan@v024085 ~]$ cat pf1.txt | sort -n -r -k4 | more
  2682.1  99.0%  99.0%   2682.1  99.0% os::malloc
     0.0   0.0% 100.0%   2657.1  98.1% Unsafe_AllocateMemory
     0.0   0.0% 100.0%   2656.9  98.1% 0x00002aaaaeec3266
     0.0   0.0% 100.0%   2656.8  98.1% 0x00002aaaaefdfb77
    18.3   0.7%  99.7%     18.3   0.7% zcalloc
     0.0   0.0% 100.0%     17.8   0.7% JavaMain
     0.0   0.0% 100.0%     17.7   0.7% Threads::create_vm
     0.0   0.0% 100.0%     17.7   0.7% JNI_CreateJavaVM
     0.0   0.0% 100.0%     17.6   0.6% init_globals
     0.0   0.0% 100.0%     17.4   0.6% universe_init
     0.0   0.0% 100.0%     17.2   0.6% Universe::initialize_heap
     0.0   0.0% 100.0%     17.2   0.6% GenCollectedHeap::initialize
     0.0   0.0% 100.0%     11.5   0.4% CMSCollector::CMSCollector
     0.0   0.0% 100.0%      5.3   0.2% GenerationSpec::init
     0.0   0.0% 100.0%      5.0   0.2% ParNewGeneration::ParNewGeneration
     0.0   0.0% 100.0%      4.2   0.2% Hashtable::new_entry
     0.0   0.0% 100.0%      4.2   0.2% BasicHashtable::new_entry
     3.3   0.1%  99.8%      3.3   0.1% apr_palloc
```

图 9.26　perftools 热点分析

图中显示内存的分配和使用都是在操作系统中，没有发现和应用代码相关的，也排除了已知的误用 Inflater/Deflater 的 native memory 的问题。仍然无法找到问题根源！

现在只能换一种思路来考察。我们再次回到 Java 代码。

这次发现代码中使用 org.apache.mina.filter.codec.textline. TextLineEncoder 类来发送和序列化传回来的数据，而且这个类使用的是 direct memory 内存：ByteBuffer buf = ByteBuffer.allocate(value.length()).setAutoExpand(true);

现在，将这个类的这个代码改成使用 JVM Heap 来存放数据：ByteBuffer.allocate (value.length(),false);

顺着这个思路进一步考察：将可能发生的 direct memory 转变成 Heap 堆内存泄露，如果真是这个代码有问题的话，必然会导致 JVM Heap 暴涨，这样就能通过 map 来分析 JVM 堆中的对象情况。

修改代码后再运行，果不其然，当达到 JVM 堆配置的上限时，GC 非常频繁，使用 map 分析 dump 下来的堆，如图 9.27 所示。

cachechecker.bin

Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.mina.transport.socket.nio.SocketSessionImpl @ 0x769483a28	296	3,025,057,464	98.39%
java.util.concurrent.ConcurrentLinkedQueue @ 0x76afffca0	32	3,025,050,072	98.39%
java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x76bdf4340	32	3,025,049,728	98.39%
java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x76bdf437	32	3,025,049,360	98.39%
java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x76bdf4	32	3,025,049,048	98.39%
org.apache.mina.common.IoFilter$WriteRequest @ 0x76bdf438	40	280	0.00%
Σ Total: 2 entries			
org.apache.mina.common.IoFilter$WriteRequest @ 0x76bdf4358	40	336	0.00%
Σ Total: 2 entries			
java.util.concurrent.ConcurrentLinkedQueue$Node @ 0x7191be3f8	32	312	0.00%
Σ Total: 2 entries			

图 9.27　map 分析堆示例

这时显示堆空间都是被 SocketSessionImpl 的 writeRequestQueue 队列持有，这个队列是 mina 的写队列，也就是 mina 不能将发送的数据及时发送出去，导致数据堵在这个队列中，进而导致内存泄露。根据以上分析，结论是使用 mina 导致了 direct memory 泄露。

这里还有一个疑问，为何前面使用的工具没有检查出 direct memory 的使用情况，没有检测出实际使用大小？原因是这个工具目前只记录了 java.nio.Bits.reservedMemory 字段记录的值，也就是所有还没被回收的 DirectByteBuffer 的 capacity 的总和。但是此处我们还需要 Java 堆找出所有 DirectByteBuffer 实例，把收集到的个数乘以 pageSize、再加上 reservedMemory 字段的值才是 direct memory 的总体使用情况。

9.8　总结

高可用建设是网站良好运行的基础，对此要有长期规划、要有体系化的建设思路，更要在日常的具体工作中落实。本章介绍了高可用建设的一些成功经验，以及在实践工作中总结的体系化思路，供读者参考。当然还是那句话，每家公司的环境都不一样、也不是一成不变的，在实践中要结合具体情况找出最有效、最合适自己的方案，这才是最好的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

附录 给新人成长的几点建议

先做事再做人

很多新同学刚进公司时会有很多困惑，到底该做技术还是该做业务？是每天加班加点写代码还是周末要空点时间看书学习？是先追求广度还是先深入钻研一门技术？是两耳不闻窗外事地一心埋头工作还是和同事打成一片参加各种活动？这些问题也曾经困扰了我，在此我愿意以过来人的角度谈谈自己的一些理解。

其实我觉得新人加入团队最关键的是要在前 3 年中建立起个人的品牌，为什么？我觉得其中有些逻辑可以和大家分享一下。在别人对你完全不了解的情况下，你做的第一件事是非常重要的。很多情况下你给别人留下的印象会传递给其他人，所以第一件事情一定要做好。那么，怎样才叫做好了某件事？在现在的职场中，根本还轮不到比拼大家的智商，大部分情况下，做事认真一点、考虑得更深入一点、更积极主动一点，基本就已经超过 70%的人了。对于这些好于 70%的同学，我们都称之为“靠谱”。也就是要把每件事情做好，并且在做事的过程当中构建自己的专业技能。比如做技术的同学，要在某一方面成为团队的专家，在某个领域成为权威，做出令人佩服的成果，逐渐建立自己的影响力。

获得了合作伙伴包括主管的认可，自然获得的机会就会更多，做的事情也会更复杂。你的合作伙伴、和你一起共事的小伙伴也会越来越强，这样就会形成正循环，个人的成长就会比较快。所以初到公司的前 3 年先通过做事形成自己的影响力，积累自

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

己的人脉，锻炼自己对未知事物的判断能力。随着所做的事情越来越多，以前的专业知识所能覆盖的内容就会越来越少，你会遇到越来越多自己不了解的领域。但是，我们不可能把所有不知道的知识都再掌握一遍，此时，培养自己的判断能力就非常重要了——这个判断包括对事物未知发展方向的判断、对事情所产生价值的判断和对人的判断。

所谓后 3 年做人，是指要更多地发挥出人的价值，不光是把事情做好，而是要在合适的时间做合适的事情，这样可以产生更大的价值，每家公司里成为“先烈”的情况都不少。我们要学会寻找多个团队的共同价值点并一起把事情做好，其实这个是最难的。因为越有价值的事情，涉及的利益方也越多，所以平衡好各个利益方的诉求成为事情成败的关键。这个过程中会涉及人的性格、交际能力、向上管理等各种综合能力。

擅于发现兴奋点

新同学刚入职时，一般所做的事情难度都不大，这个时候很容易会产生落差感。像阿里校招的基本全是学校里面的尖子生，但到岗后会发现自己每天做的都是一些琐碎平庸的事情，容易产生失望挫折心理。此时如果不调整好心态，就很难激发自己的工作热情，慢慢就真的变得平庸了。我可以说一下我在阿里的一些经验。当初我和大部分同学一样，一开始也是从最基本的日常任务开始做起，我当时对自己的要求是代码层面不出 bug、注释要写清楚、按时发布。这些都是最基本的要求；但与此同时，我会利用业余时间做好以下几件事情。

（1）主动挖掘工作中可以提升效率的地方，把琐事工具化。记得当时我在做一个搜索需求时，发现传给搜索的参数非常多，而且每次拼 URL 都非常麻烦，由于这个需求是运营的同学提出的，我就做了一个页面，把每个参数做成表单，然后运营同学只要在对应的字段（有说明）表单中填入值，就可以自动生成最终的 URL，这个方法提升了运营同学生成 URL 的效率。当然，做出来以后也得到运营同学的好评。

（2）挖掘潜在效率，不断积累人品。当时我们有个小图书馆，需要专人记录每周每位同学借了哪些书，什么时候归还等信息——这些还都是人肉记录的方式——我就做了个图书管理系统，攒了不少“人品”。

（3）除了以上两点外，最重要的还是要积累自己的技能。我会把在公司用到的中

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

间件都深入学习一遍，并且写成学习文档。和其他同学写的那种简单的使用心得型的文档不同，我写的学习文档都是分析设计原理的：分析为什么这么设计、用到了哪些设计模式、关键的技术点等。那些已经工作了好几年并且熟悉中间件的同学也会有兴趣来看。虽然写这些文章获得不少好评，但是公司内部的读者毕竟有限，而且写文章也花费了我不少精力，所以我要寻找新的动力继续写下去。于时我的写作主题从公司独有的技术框架转移到公司内外均通用的技术框架上，如 Spring、iBATIS、Tomcat 等 JVM 技术，并且寻找机会向外面的杂志和网站投稿。这样一方面可以有一些经济回报，更重要的是可以收到更多的反馈信息。最终，我在在 Developerworks 上投了十几篇文章，并获得网站最受欢迎的作者称号。目前我写的文章估计也是 Developerworks 上单篇访问量最大的。

综上，擅于从工作中寻找源动力非常重要。每做一件事情，都要从不同角度发现其中的价值，并始终保持做事的激情。比如事情做好后，可以获得更多的知识、可以得到别人的感谢、可以获得别人的称赞、可以获得更多人的认可……找准正反馈的来源，激励自己持续地做好事情。

其实写文章和总结一直是我工作中的一个兴奋点，它会带来多种好处。

- 可以系统性地沉淀和总结自己学到的知识，只有经过沉淀和总结的知识才能被更长远地记住；
- 可以增加自己的影响力，因为分享可以给别人带来价值，别人就会关注你；
- 提高做事（尤其是那些琐碎的事情）的动力，把琐碎的事情都连贯起来也会有很好的结果。

所以我们有必要找到工作中的兴奋点，这样可以使你持续地对工作充满激情。

与人协作

在与人协作上，这几年我的感触很深。从最初想一个人完成所有事情，到现在尽量让别人完成所有事情，这种转变真的是很特别的经历。现在回过头来再看新来的同学的表现，促使我把我的转变过程写出来分享给大家。在公司中只有很少的事情是由个人独自完成的，大部分都是由团队协作完成。除了不同的工种差别外，还有相同岗位不同层级之间的差异。以公司中常用的项目运作模式为例，一个项目中一般会有一个项目经理（PM）、一个或多个架构师、若干个不同功能的开发工程师，其他还有 PD、

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

UED、测试、PE 等岗位的工程师。项目经理和架构师这两个角色刚好给两个同质的团队提供了很好的选择，项目经理一般会拿到业务结果，而架构师一般能体现项目中的技术成果。典型的场景是一项很好的技术通过项目经理落地并达到双赢。

在与人协作上一般要经历如下几个阶段。

（1）把别人的想法当成自己的想法，帮别人实现。很简单，初期与人合作的时候，要把别人的事当成自己的事，并且想方设法实现好，该做螺丝钉的时候就要做好。

（2）必须经历自己的想法自己实现的阶段。新人一般会害怕把自己的想法说出来，生怕自己的想法被别人抢走。我也曾经有过这样的担心，而且在实际工作中，的确有些同学会“盗取”别人的想法，稍做改换就到处宣扬是自己的观点这种情况，尤其是当他们抢着向老板汇报时就更令人讨厌了。但现在想想，这种担心也没有必要，因为更重要的不是想法而是实践。对新同学来说，多想少说多实践才是王道，千万不要少想多说少实践。有好的想法不妨说出来，并且用 demo 证明可行性，如果最终证明想法靠谱也不用纠结名义上的结果。就像我前面所说的，先做好事，不用急着去争名，这正是你积累福报的机会。

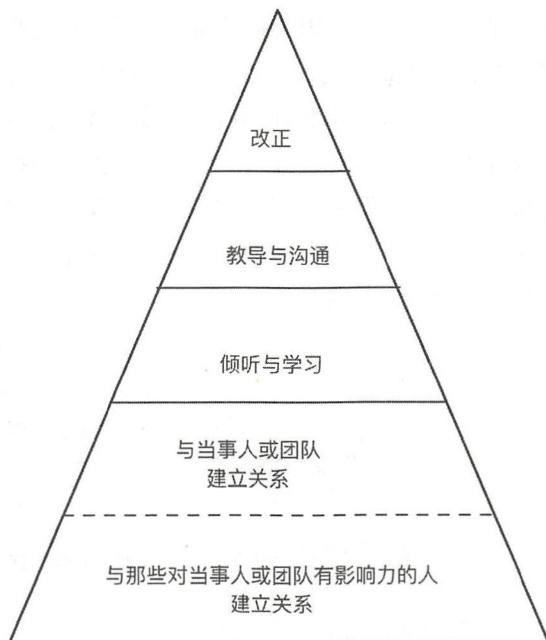
（3）自己的想法找一帮人实现。这个阶段最重要的就是平衡利益，让每位项目成员能获得最大的利益是项目成功的关键，这里不仅仅包括参与项目的成员还包括项目成员的老板们。把事情做好还只是第一步，项目要发挥出最大的价值需要各团队老板们的宣传，只有这样才能有更多的落地场景，最终的成功的确需要天时地利人和，很不容易。

（4）自己的想法，让别人说出来，并且实现。你的想法变成了别人的想法，并且最后实现了，如果你到了这个阶段，那么你肯定已经是一位很有影响力的人了。新人无疑最忌讳出现这种情况，但是如果你前面的几个阶段都做得很好，那么在遇到这种情况时一定会很淡定，因为此时的你应该已经是 Leader 了，这样的任务正是 Leader 的份内之事。

与人协作时我觉得 Choice 课程中介绍的影响力金字塔（阐述如何建立最有效的关系）是非常有参考价值的，把最多的时间花在能对合作伙伴产生影响的人身上，与他们建立良好关系是最有成效的。大家可能会说，这不就是那群天天在公司不干事只会拉关系的人吗？其实建立关系只是手段，真正重要的是建产关系能够产生什么结果，这才是最终的目的。建立影响力是最有效的一种协作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



老板希望你怎么做

下面再根据自己带团队的经历，总结一些与老板相处的经验供大家参考。

（1）第一步要和老板建立相互的信任。请注意我说的是相互信任，这个最重要。你之所以信任老板无非在于你内心是服他的、你知道他是真心愿意培养你的。老板能信任你则主要是交给你的事情，你都能漂亮地完成；认为你会死心塌地地跟随他。很多的管理事故大都是由于缺乏信任所导致的。

（2）积极主动地给老板带来一些惊喜，是你获得更多机会的助燃剂。老板一般喜欢两种人，一种是听话的人，能兢兢业业地完成任务；二是总是有新想法并且爱折腾的人：有想法还不够，最好是有结果。很显然第二种人更容易获得老板的提拔。

（3）适当地给老板找些麻烦也是有好处的，这里的“找麻烦”有两重含义。

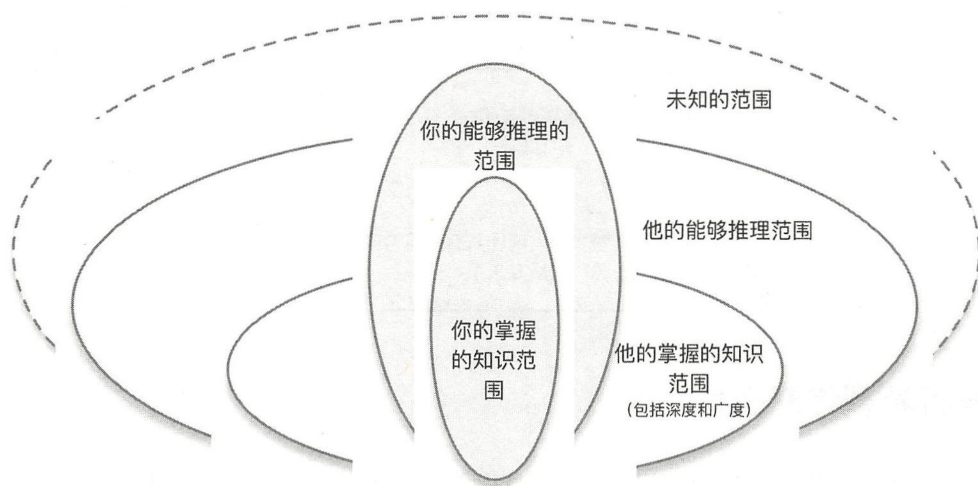
一是要有自己独立的观点。如果发现明显问题时还听之任之，会让老板觉得你没有主见，应该积极地表达自己的观点。当然这样做的前提是双方都是正直的人，如果老板是一个小肚鸡肠的人，那你还是好自为之吧。

二是尽量不要让老板的认知范围完全覆盖你的认知范围，如下图所示。如果这样

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的话，你在老板眼里就是一个透明人，因为你的所有思考和行动都在他可预知的范围内，那样的话就很难对你有很大的期待了。在工作场景中，你要在专业知识的深度、广度或者其他技能上有长项。比如你擅长前端技术、或者线上运维能力非常好、或者思考问题非常全面，或者逻辑推导能力很好……如果能达到下图描述的状态是比较理想的。当然也要避免夜郎自大的情况——当你觉得掌握的知识比老板强，你可能会觉得老板不如你。但是千万不要忘了，老板的推理范围，也就是对事情的判断能力可能还是远远超过你的，所以不能过于自大。



俗话说“跟对一个好老板是你一生的福报”，的确是这样的。以我目前的观察，名老板都是情商很高但内心真正简单透明的人，职场的权术有很多，但是仅靠权术很难真正获得长久的成功。

一言以蔽之，不管你是一个刚毕业的学生，还是牛逼大咖加入了新的环境，前期所有的精力都应该放到具体的小事情上，第一步要以事服人，后面再以德服人——这是我认为的比较好的成长思路。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

参考资料

[1] 玄难.从应用到中台,业务平台的演进[OL].https://yq.aliyun.com/articles/30340.2018-06-06.

[2] 小邪.淘宝前台系统优化实践[OL].http://www.infoq.com/cn/presentations/jjw-taobao-front-optimization-practice/.2018-06-06.

[3] 子柳.淘宝技术这十年[M].北京:电子工业出版社,2013.

[4] Dubbo 服务框架[OL].http://dubbo.apache.org/.2018-06-06.

[5] RocketMQ 消息框架[OL].http://rocketmq.apache.org/.2018-06-06.

[6] TDDL 阿里分布式数据层中间件[OL].https://yq.aliyun.com/articles/25457.2018-06-06.

[7] DRDS 分布式关系型数据库服务[OL].https://help.aliyun.com/document_detail/29659.html.2018-06-06.

[8] HSF 阿里分布式服务框架[OL].https://www.alibabacloud.com/help/zh/doc-detail/63868.htm.2018-06-06.

[9] Kafka 分布式消息框架[OL].http://kafka.apache.org/.2018-06-06.

[10] Redis K/V 存储[OL].https://redis.io/.2018-06-06.

[11] memcached K/V 存储[OL].http://memcached.org/.2018-06-06.

仅供非商业用途或交流学习使用

[12] TFS 阿里分布式对象存储[OL].http://tfs.taobao.org/.2018-06-06.

[13] Seaweedfs 分布式对象存储[OL].https://github.com/chrislusf/seaweedfs.2018-06-06.

[14] Tair 阿里分布式 K/V 缓存[OL].http://tair.taobao.org/.2018-06-06.

[15] Webpagetest 用户测试页面的加载速度[OL].http://www.webpagetest.org/.2018-06-06.

[16] WebP 相关概述[OL].https://baike.baidu.com/item/WebP.2018-06-06.

[17] Caniuse,网站开发浏览器兼容性查询的网站[OL].https://caniuse.com/.2018-06-06.

[18] 关于 Nashorn 的一些介绍[OL].https://www.infoq.com/articles/nashorn.2018-06-06.

[19] 关于滴滴出行中台的建设思路[OL].http://developer.51cto.com/art/201712/559758.htm.2018-06-06.

[20] 毕玄.阿里异地多活的建设思路[OL].http://www.infoq.com/cn/articles/interview-alibaba-bixuan.2018-06-06.

[21] 阿里单元化建设的介绍[OL].http://www.docin.com/p-1514328842.html.2018-06-06.

[22] 李纯.B2B 事业部的技术演进[OL].http://www.ebrun.com/20161122/203052.shtml.2018-06-06.

[23] 李三红.关于 JVM 优化的分享[OL].http://tech.it168.com/a2016/0901/2893/000002893624. shtml.2018-06-06.

[24] 阿里数据库技术架构演进[OL].https://yq.aliyun.com/articles/321080?utm_content=m_38579.2018-06-06.

[25] Mesos 架构介绍[OL].http://mesos.apache.org/documentation/latest/.2018-06-06.

[26] Docker 容器与虚拟机的区别[OL].https://www.zhihu.com/question/48174633.2018-06-06.

[27] Docker 和 LXC 的区别[OL].http://dockone.io/article/368.2018-06-06.

[28] 阿里全链路压测分享[OL].https://yq.aliyun.com/articles/163747.2018-06-06.

[29] 叔同. 双 11 稳定性建设分享[OL].http://www.360doc.com/content/17/0105/20/21332217_620342869.shtml.2018-06-06.

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

好评袭来

作者完整地为我们展现了一个初级系统在演化成一个全球分布式的系统的过程中，从语言选择、分布式框架改造、平台化演进、系统优化到稳定性建设等关键过程的思考……从这些一流的实践中，读者可收获多维度的启发和共鸣，值得一读！

——阿里云研究员
褚霸

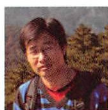
好的架构是一个系统的根本，好的性能是一个系统稳定运行的保证，本书应该可以给大家带来不一样的收获。

——PerfMa CEO
你假笨（寒泉子）

这是一本关于互联网高并发架构设计的优秀书籍，它从各角度剖析系统设计的演化与优化，循序渐进地将一系列复杂问题阐述得清晰、简单、易懂，是一本理论与实践相结合的实用书籍。

——《分布式服务架构：原理、设计与实战》《可伸缩服务架构：框架与中间件》作者
李艳鹏

作者简介



许令波（君山），2009 年毕业后进入阿里巴巴，经历了淘宝网系统架构从早期的单系统到分布式化、平台化、无线化、中台和国际化的改造过程，业务流量从 1 亿次到 50 亿次的爆发式增长过程。在此过程中，他遇到的最大挑战就是大流量和高并发性能瓶颈，也曾多次担任双 11 性能优化负责人，是处理大型网站高并发和稳定性问题的“老司机”。

目前加入滴滴开启新的职业历程，参与滴滴的基础架构研发工作。

注册成为博文视点社区（www.broadview.com.cn）用户，即享受以下服务：

● 提勘误赚积分：可在【提交勘误】处提交对内容的修改意见，若被采纳将获赠博文视点社区积分（可用来抵扣购买电子书的相应金额）。

● 交流学习：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者共同交流。

页面入口：http://www.broadview.com.cn/34135

博文视点Broadview

@博文视点Broadview

上架建议：大型网站

定价：79.00元

责任编辑：刘 皎
封面设计：吴海燕

欢迎投稿
邮箱：Ljiao@phei.com.cn
电话：010-88254395
新浪微博：@皎丫子