

分布式服务框架

原理与实践

李林锋 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

构建企业互联网架构的关键在于系统分布式和服务化,尤其对于大型网站和大型企业系统,系统的灵活性、超大容量、弹性和自治能力是非常大的挑战。在《分布式服务框架原理与实践》一书中,作者基于深厚的软件技术积累和电信领域成功应用实践,对如何构建分布式服务化系统,提供了原理分析、关键技术、开发案例以及业界技术对比,非常系统化,不论是学习分布式服务技术还是深入大型互联网架构都非常实用。

——华为云集成平台首席架构师 苗彩霞

认识林锋已有多,从《Netty权威指南》到本书的诞生,再次见证了作者在该领域深厚的沉淀。浏览该书的目录以及相关章节,我惊讶于作者在这些领域深入的洞察和实践。该书几乎覆盖了分布式系统开发的每一个关键技术点,包括最为重要的通信框架设计、时下流行的微服务、服务路由关联的技术和策略,以及饱受争议的OSGi。强烈推荐相关从业人员阅读此书。

——苏宁云商云计算中心技术总监 汤泳

在大型网站架构设计方面摸爬滚打多年后,看到《分布式服务框架原理与实践》如获至宝,作者条理清晰、由浅入深地解析了分布式服务架构所涉及方方面面的关键技术和原理,既有纵向演进介绍,又有横向竞品对比。尤其针对各种场景所提出的设计原则或最佳实践,都是作者的实战总结,有些经验的获取成本高昂,非常宝贵。本书完全可以直接用于指导分布式服务系统的构建。

——中国移动手机阅读基地平台首席架构师 胡穗

分布式的应用在设计、开发以及部署的各个方面都比较复杂,国内外也没有权威的图书进行系统介绍,于是在这方面,我们不得不一遍遍地踩坑。林锋有着深厚的技术基础和丰富的架构经验,这本集他经验和心血而成的图书,包含了分布式系统的方方面面,既有宏观的理论介绍,也有来自一线的经验分享,相信它必将成为架构师和开发人员的必备图书!

——东软集团资深软件工程师、InfoQ编辑 张卫滨

“微服务”无疑是本年度最热的技术关键词之一!那如何落地微服务呢?我认为首先要实现服务化,而本书恰好提供了一个很好的服务化操作指导。作者首先分析了作为一个分布式服务框架所需具备的能力,包括服务注册中心、服务调用、服务路由、服务发布/灰度发布等;其次作者分析了服务底层如何有效地进行通信,包括通信框架、序列化/反序列化及协议栈等;再次作者分析了服务如何做到高可靠性及高安全性等重要特性;最后作者也阐述了从服务化如何向微服务演进。

——麻袋理财首席架构师 王天青

以OpenStack为规范建设的IaaS、以Docker为代表的容器技术、以分布式微服务框架构建的业务平台即将颠覆业务系统整体建设方案,新的系统建设方案将极大提升业务系统的可用性、扩展性和应变能力。微服务架构对于运营商内容型业务的互联网化转型意义非凡,系统架构微服务化才能真正支撑好业务转型的需要。本书将成为帮助大家更好地理解微服务框架关键技术的原理和实现的必备书籍。

——咪咕动漫系统支撑部技术总监 李鹏

锋兄在华为一直从事核心代码的架构设计和开发,属于实战型架构师,而且乐于分享。《分布式服务框架原理与实践》源于他在多年架构设计工作中的实战经验,阅读价值极高!在面向大规模、分布式系统架构中,服务框架是其中的核心和必经之路。祝贺锋兄新书造福广大程序员!

——青蛙CEEWA运动无人机合伙人、

前华为开放平台总架构师 冯毅

近些年来,越来越多网站需要同时提供Web、移动App、OpenAPI多种访问方式,基于分布式服务的业务分治与复用需求越来越强烈,使用分布式服务构建系统已经成为互联网开发的常用手段。但是分布式服务的关键技术有哪些?核心原理是什么?最佳实践是什么?本书作者作为分布式框架的开发者根据自己的实践经验编写的这本《分布式服务框架原理与实践》或可为您解惑。分布式服务框架用到的各种技术也是整个互联网分布式技术的一个缩影,您也可窥一斑而知全豹,通过本书学习掌握各种分布式开发技巧。

——宅米网CTO、

《大型网站技术架构:核心原理与案例分析》作者 李智慧
整书由构建分布式服务为基础讲起,逐步深入到分布式服务的保障机制,最后也讲解了时下新兴分布式设计方案微服务架构。书中内容组织清晰,图例详实,非常便于理解与吸收,是一本不错的提升分布式服务架构能力的书籍。

——链家网架构师 吕毅

本书深度阐述了应用和系统架构方面的设计和原理,真实体现了李林锋丰富的技术架构经验以及乐于分享的精神。在业务系统越来越讲究高可用、高性能、可伸缩扩展、高安全性、自动运维的今天,本书集合了大型企业多年的架构思路,为技术以及产品人员提供了重要的参考依据,从理念上提升了每位读者的技术水平,非常值得深入阅读和理解。

——阿里云PaaS平台产品架构师 杨林



博文视点Broadview



@博文视点Broadview



责任编辑:董英

封面设计:李玲

上架建议:计算机>架构设计

ISBN 978-7-121-27919-5

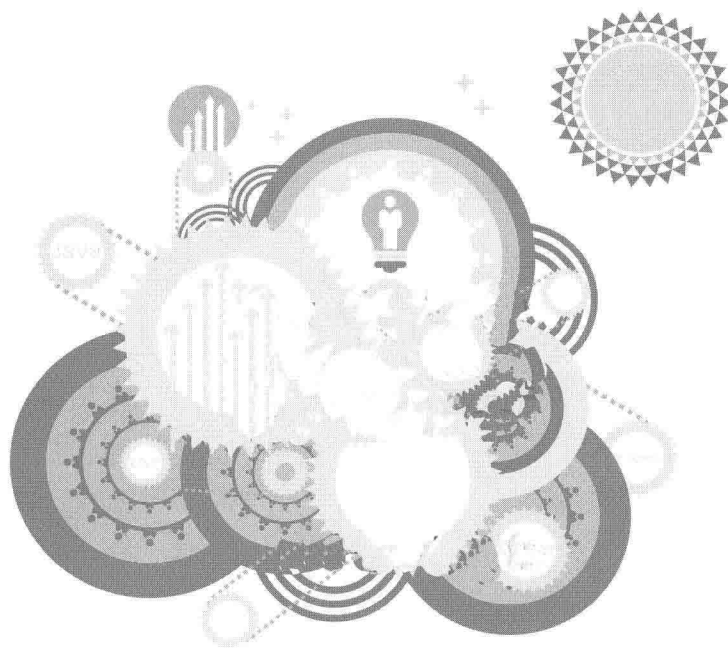


定价:65.00元

分布式服务框架

原理与实践

李林锋 / 著



電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书作者具有丰富的分布式服务框架、平台中间件的架构设计和实践经验，其主导设计的华为分布式服务框架已经在全球数十个国家成功商用。书中依托工作实践，从分布式服务框架的架构设计原理到实践经验总结，涵盖了服务化架构演进、订阅发布、路由策略、集群容错和服务治理等多个专题，全方位剖析服务框架的设计原则和原理，结合大量实践案例与读者分享作者对分布式服务框架设计和运维的体会；同时，对基于 Docker 部署微服务以及基于微服务架构开发、部署和运维业务系统进行了详细介绍。

本书适合架构师、设计师、软件开发工程师、测试工程师，以及其他对互联网分布式架构感兴趣的相关人士阅读。

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

分布式服务框架原理与实践 / 李林锋著. —北京：电子工业出版社，2016.1
ISBN 978-7-121-27919-5

I. ①分… II. ①李… III. ①分布式计算机—研究 IV. ①TP338.8

中国版本图书馆 CIP 数据核字（2015）第 308395 号

责任编辑：董 英

印 刷：北京京师印务有限公司

装 订：北京京师印务有限公司

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：787×980 1/16 印张：19.5

字数：415 千字

版 次：2016 年 1 月第 1 版

印 次：2016 年 3 月第 2 次印刷

印 数：4001~7000 册 定价：65.00 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。

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

服务热线：（010）88258888。

序一

IT 的体系架构在历史上经历了几次大的变化。从主机瘦客户机时代，到 Client Server 兴起，然后过渡到 Browser Server 的架构，再到移动+云计算+大数据的大热。

总结起来，IT 的核心变迁轨迹是在客户端不断提升体验，易联易用，而在服务器端则是不断追求性能和成本优化改进。近几年，还有一个非常明显的趋势是技术的成熟度和融合度不断提高，移动、云计算领域平台型的公司（Android、iOS、AWS）使得整个 IT 能力的使用成本很低，进入速度非常快，现在的高中生也可以利用手头的工具非常快速方便地参与到软件构建中来，这在以前是不可想象的。移动互联网兴起以后，大概在短短 5 年内，世界上绝大部分原来在 PC 端可以满足的需求都由移动端的应用实现了一遍。IT 已经变成了一个快速消费品，而不是一个奢侈品。

技术的进步使得 IT 的敏捷性大大提升，但是对于一个大型系统来说，如何能够降低系统的复杂度，提升敏捷性是关键而又头疼的问题。我们也看到一些通用的标准和最佳实践已经建立起来了，降低模块之间的耦合度，提升组件的内聚性，规范对外的接口，实现分布式的系统架构，把一个大型系统通过服务化的方式规划治理起来，已经成为一个共识。一个现代的大型 IT 系统，服务可以多至十万、百万级，如此众多的服务，从设计、开发、运行、编排、维护到治理，每一个环节都需要大量深入仔细的考虑，才能够运转起来。我们可以把这样一个系统比喻成一个城市，城市里面有成千上万的公司，每一个公司都有自己的业务来往，同时又需要现代化的交通、电力、通信、金融等体系的支持。无论是小公司还是大公司，都依赖于整个城市的运作和治理体系。公司和城市是相辅相成的关系。IT 系统里面的业务模块和服务化框架也是相辅相成的关系。服务化框架对于一个大型 IT 系

是不可或缺的。

业界有很多介绍服务化理念和技术点的文章和书籍。但真正能够在理论、实践、技术要点、眼界多方面全面覆盖的资料,还是比较缺乏的。我很高兴看到林锋能够总结自己在理论、产品和客户实践多方面的认知和经验,为读者奉献《分布式服务框架原理与实践》一书,深入浅出地介绍分布式服务的概念、体系和关键技术点。希望这本书能够帮助你了解分布式服务框架,掌握分布式服务体系和技术要点,同时也能实践服务化给你的 IT 系统带来的敏捷。

黄省江

华为软件 PaaS 平台&云中间件技术总监

序二

容器 SDN 技术与微服务架构实践从 20 世纪末期的第一波互联网浪潮开始，软件架构的主流就逐步从 Office 这类纯客户端软件逐步过渡到服务端的架构设计。与传统的客户端设计相比，服务端的架构设计更关注伸缩性、可用性和可维护性。很可惜的是，现在市面上讲语言、讲算法、讲设计模式或者讲某一门独立的技术的书都不少，但服务端架构设计的书却寥寥无几。

从私下沟通的结果来看，大部分互联网企业都没有解决好上面的几个问题。正如建筑设计的现代化是从结构工程师开始的，软件设计的现代化会从架构师开始，希望李林锋这本书能够帮助广大的架构师或者有志于成为架构师的人掌握好这些知识，让架构师能够带领开发团队构建出稳定、安全、可维护、可伸缩的合格产品，让架构师在软件开发现代化这条路上起到领路人的作用。

本书最末一章讲述了最近很火的微服务架构，微服务架构的思想包含了对以前种种架构模式的反思，也通过 Docker、Mesos 等技术变成第一个可以轻松产品化的架构思想。我相信再过几年对于开发人员，特别是服务端的开发人员，他们所面临的开发模式将与现在的开发模式有很大的差异，这种差异我觉得甚至会大过程序可以有服务端这个概念的引入。

在微服务架构下，开发的门槛将进一步降低，分布式将更加自然而不是依赖于艰难的设计，运维的负担也将降低至一个极低的水平。我们可以期望通过微服务架构，从想法到产品的距离将更短，也能期待涌现出更多让人叹为观止的产品，还能期待能出现些我们现在完全无法预测的技术和产品。

李道兵
七牛云首席架构师

前言

2008 年 9 月份，我有幸参与了一个华为软件公司的国内 Top3 项目，作为一名有经验的开发，项目经理安排我负责整个系统的实时交易和后台服务端设计。

尽管有 ERP 软件开发经验，但是第一次参与这么大项目的架构设计和开发，压力还是非常大，经常夜不能寐。

好在公司当时已经研发了 Java Web 框架，它基于 Spring + Struts + iBatis 构建，利用公司成熟的 MVC 框架我们很顺利地完成了项目的开发和交付，并最终成功上线运行。

随着业务的发展，用户数和需求逐步增多，团队规模越来越大，我们遇到了很多棘手的问题：

- 1) 代码重复率高：一些业务层的公共功能，被多个模块重复开发，导致研发成本上升，代码质量下降，架构腐化，为后续系统的运维和新功能的开发带来巨大的挑战。
- 2) 需求变更困难：由于长流程无法有效拆分、代码重复率高等因素，导致每次需求变更就影响一大片，需要做大量的回归测试来保证质量，需求的交付周期被拉长。
- 3) 部署效率低：业务没有拆分，很多功能模块都打到同一个 war 包中，一旦有一个功能发生变更，就需要重新打包和部署；巨无霸应用由于包含功能模块过多，编译、打包时间比较长，一旦编译过程出错，需要根据错误重新修改代码再编译，耗时较长。

- 4) 学习成本高：业务流程是由一长串本地接口或者方法调用串联起来的，臃肿而冗长，而且往往由一个人负责开发和维护。随着业务的发展和需求变化，本地代码在不断的迭代和变更，最后形成了一个个垂直的功能孤岛，只有原来的开发者才理解接口调用关系和功能需求，一旦原有的开发者离职或者调到其他项目组，这些功能模块的运维就会变得非常困难。
- 5) 缺乏统一的 RPC 框架：由于 Web 框架只提供了 HTTP/HTTPS 协议，例如 SOAP、SMTP 等协议栈需要业务自行集成第三方的开源框架，超时重发、网络断连等底层故障需要在应用上层统一封装和处理，工作繁琐而且容易出错，对业务开发人员的技能要求也非常高。

随着公司业务不断发展，传统 MVC 架构已经无法再满足业务对平台的诉求，因此在 2010 年下半年我们开始研发新的 SOA 中间件，它包括企业集成总线 ESB、流程编排引擎 BPM、RPC 通信框架等。新的 RPC 通信框架底层封装了 Java NIO 通信框架 Netty、常用的序列化/反序列化框架，以及为应用层提供线程池和消息调度器，基于 RPC 通信框架，业务可以快速的实现跨进程的远程通信，而不需要关心底层的通信细节，例如链路的闪断、失败重试等，极大的提升了应用的开发效率。

在很长一段时间，自研的 RPC 框架成为业务首选的 Java 服务端框架。

随着 RPC 框架的推广和使用日益深入，一些新的公共需求被反馈过来：

- 1) 依赖管理：当服务越来越多时，服务 URL 配置管理变得非常困难，希望有一个统一的服务注册中心管理服务的依赖关系。
- 2) 透明路由：通过订阅发布机制，消费者只需要关心服务本身，并不需要配置具体的服务提供者地址，实现服务的自动发现。
- 3) 服务治理：业务失败之后的放通处理，超时时间控制、流控等常用运维功能，希望能够独立出一个服务治理中心，统一对集群各节点的服务做在线治理，提升治理效率，保障服务 SLA。
- 4) 其他……

为了解决这些问题，以 RPC 框架为核心，我们构建了全新的分布式服务框架，相比于传统 RPC 框架，它提供了如下新特性：

- 1) 基于注册中心的服务订阅/发布机制，支持服务自动发现和健康状态检测。
- 2) 集群容错。
- 3) 依赖解耦，全配置化开发，对应用零侵入。
- 4) 服务治理，包括服务降级、服务调用链跟踪、服务上线审批和下线通知等。
- 5) 服务化最佳实践等。

分布式服务框架不仅仅包含核心的运行时类库，还包括服务划分原则、服务化最佳实践、服务治理、服务监控、服务开发框架等，它是一套完整的解决方案，用来协助应用做服务化改造，以及指导用户如何构建适合自己业务场景的服务化体系，将服务化的价值发挥到极致。

基于分布式服务框架，业务终于可以把全部精力都放到应用层的逻辑开发，研发效率、系统可靠性都得到了极大的提升。目前，华为电信软件主要解决方案几乎所有的 Java 系统都基于分布式服务框架构建，底层的基础框架实现了统一。

最近一年多来，随着 DevOps 和以 Docker 为首的容器技术的发展，微服务架构逐渐流行起来，微服务架构的流行有其必然的历史原因，它是敏捷开发、基础设施服务化、DevOps 和互联网行业快速发展的综合产物。亚马逊 AWS、Netflix 等都是微服务的成功实践者，相信未来国内越来越多的大型应用也会演进到微服务架构。

华为软件公司的 Java 架构经历了传统的 MVC 垂直架构-RPC 框架-分布式服务框架，目前正在向 Docker + 微服务方向演进，整个服务化架构的演进历程也是业界技术变迁的一个缩影。

在这 7 年的演进历程中，我有幸全程参与了相关框架的架构设计和核心模块的开发，深有感触。在此希望将自己设计、开发和运维的相关经验分享出来，为初学者和相关经验人士提供一些启发，汲取相关经验，少走些弯路。

能够完成本书需要感谢很多人，首先感谢华为公司给我提供了足够大的舞台，感谢华为 PaaS 平台&中间件团队领导和同事莫晓军、望岳和王世军等，以及与我在分布式服务框架团队一起战斗过的开发、测试和资料。

其次要感谢我的家人，你们一直在背后默默的支持我。感谢参与本书编辑的英姐、美工以及其他人员，你们的辛苦换来了本书的如期上市。

最后要感谢所有的读者，你们的支持和鼓励是我写作本书的动力源泉。

李林锋

2015 年 12 月于南京

目 录

第 1 章 应用架构演进	1
1.1 传统垂直应用架构.....	2
1.1.1 垂直应用架构介绍	2
1.1.2 垂直应用架构面临的挑战.....	4
1.2 RPC 架构	6
1.2.1 RPC 框架原理	6
1.2.2 最简单的 RPC 框架实现	8
1.2.3 业界主流 RPC 框架.....	14
1.2.4 RPC 框架面临的挑战	17
1.3 SOA 服务化架构	18
1.3.1 面向服务设计的原则.....	18
1.3.2 服务治理	19
1.4 微服务架构	21
1.4.1 什么是微服务	21
1.4.2 微服务架构对比 SOA.....	22
1.5 总结	23
第 2 章 分布式服务框架入门	25
2.1 分布式服务框架诞生背景	26
2.1.1 应用从集中式走向分布式.....	26

2.1.2	亟需服务治理.....	28
2.2	业界分布式服务框架介绍.....	29
2.2.1	阿里 Dubbo.....	30
2.2.2	淘宝 HSF.....	33
2.2.3	亚马逊 Coral Service.....	35
2.3	分布式服务框架设计.....	36
2.3.1	架构原理.....	36
2.3.2	功能特性.....	37
2.3.3	性能特性.....	39
2.3.4	可靠性.....	39
2.3.5	服务治理.....	40
2.4	总结.....	41
第 3 章	通信框架.....	42
3.1	关键技术点分析.....	43
3.1.1	长连接还是短连接.....	43
3.1.2	BIO 还是 NIO.....	43
3.1.3	自研还是选择开源 NIO 框架.....	46
3.2	功能设计.....	47
3.2.1	服务端设计.....	48
3.2.2	客户端设计.....	50
3.3	可靠性设计.....	53
3.3.1	链路有效性检测.....	54
3.3.2	断连重连机制.....	56
3.3.3	消息缓存重发.....	57
3.3.4	资源优雅释放.....	58
3.4	性能设计.....	59
3.4.1	性能差的三宗罪.....	59
3.4.2	通信性能三原则.....	60
3.4.3	高性能之道.....	61
3.5	最佳实践.....	61
3.6	总结.....	64

第 4 章	序列化与反序列化	65
4.1	几个关键概念澄清	66
4.1.1	序列化与通信框架的关系	66
4.1.2	序列化与通信协议的关系	66
4.1.3	是否需要支持多种序列化方式	67
4.2	功能设计	67
4.2.1	功能丰富度	67
4.2.2	跨语言支持	68
4.2.3	兼容性	69
4.2.4	性能	70
4.3	扩展性设计	71
4.3.1	内置的序列化/反序列化功能类	71
4.3.2	反序列化扩展	72
4.3.3	序列化扩展	75
4.4	最佳实践	77
4.4.1	接口的前向兼容性规范	77
4.4.2	高并发下的稳定性	78
4.5	总结	78
第 5 章	协议栈	79
5.1	关键技术点分析	80
5.1.1	是否必须支持多协议	80
5.1.2	公有协议还是私有协议	80
5.1.3	集成开源还是自研	81
5.2	功能设计	82
5.2.1	功能描述	82
5.2.2	通信模型	82
5.2.3	协议消息定义	84
5.2.4	协议栈消息序列化支持的字段类型	85
5.2.5	协议消息的序列化和反序列化	86
5.2.6	链路创建	89

5.2.7	链路关闭	90
5.3	可靠性设计	90
5.3.1	客户端连接超时	90
5.3.2	客户端重连机制	91
5.3.3	客户端重复握手保护	91
5.3.4	消息缓存重发	92
5.3.5	心跳机制	92
5.4	安全性设计	92
5.5	最佳实践——协议的前向兼容性	94
5.6	总结	95
第 6 章	服务路由	96
6.1	透明化路由	97
6.1.1	基于服务注册中心的订阅发布	97
6.1.2	消费者缓存服务提供者地址	98
6.2	负载均衡	98
6.2.1	随机	98
6.2.2	轮循	99
6.2.3	服务调用时延	99
6.2.4	一致性哈希	100
6.2.5	粘滞连接	101
6.3	本地路由优先策略	102
6.3.1	injvm 模式	102
6.3.2	innative 模式	102
6.4	路由规则	103
6.4.1	条件路由规则	103
6.4.2	脚本路由规则	104
6.5	路由策略定制	105
6.6	配置化路由	106
6.7	最佳实践——多机房路由	107
6.8	总结	108

第 7 章 集群容错	109
7.1 集群容错场景	110
7.1.1 通信链路故障	110
7.1.2 服务端超时	111
7.1.3 服务端调用失败	111
7.2 容错策略	112
7.2.1 失败自动切换 (Failover)	112
7.2.2 失败通知 (Failback)	113
7.2.3 失败缓存 (Failcache)	113
7.2.4 快速失败 (Failfast)	114
7.2.5 容错策略扩展	114
7.3 总结	115
第 8 章 服务调用	116
8.1 几个误区	117
8.1.1 NIO 就是异步服务	117
8.1.2 服务调用天生就是同步的	118
8.1.3 异步服务调用性能更高	120
8.2 服务调用方式	120
8.2.1 同步服务调用	120
8.2.2 异步服务调用	121
8.2.3 并行服务调用	125
8.2.4 泛化调用	129
8.3 最佳实践	130
8.4 总结	131
第 9 章 服务注册中心	132
9.1 几个概念	133
9.1.1 服务提供者	133
9.1.2 服务消费者	133
9.1.3 服务注册中心	133

9.2	关键功能特性设计	134
9.2.1	支持对等集群	135
9.2.2	提供 CRUD 接口	136
9.2.3	安全加固	136
9.2.4	订阅发布机制	137
9.2.5	可靠性	138
9.3	基于 ZooKeeper 的服务注册中心设计	139
9.3.1	服务订阅发布流程设计	139
9.3.2	服务健康状态检测	141
9.3.3	对等集群防止单点故障	142
9.3.4	变更通知机制	144
9.4	总结	144
第 10 章	服务发布和引用	145
10.1	服务发布设计	146
10.1.1	服务发布的几种方式	146
10.1.2	本地实现类封装成代理	148
10.1.3	服务发布成指定协议	148
10.1.4	服务提供者信息注册	149
10.2	服务引用设计	150
10.2.1	本地接口调用转换成远程服务调用	150
10.2.2	服务地址本地缓存	151
10.2.3	远程服务调用	151
10.3	最佳实践	152
10.3.1	对等设计原则	152
10.3.2	启动顺序问题	153
10.3.3	同步还是异步发布服务	153
10.3.4	警惕网络风暴	154
10.3.5	配置扩展	154
10.4	总结	156

第 11 章 服务灰度发布	157
11.1 服务灰度发布流程设计	158
11.1.1 灰度环境准备	158
11.1.2 灰度规则设置	159
11.1.3 灰度规则下发	160
11.1.4 灰度路由	161
11.1.5 失败回滚	162
11.1.6 灰度发布总结	163
11.2 总结	163
第 12 章 参数传递	164
12.1 内部传参	165
12.1.1 业务内部参数传递	165
12.1.2 服务框架内部参数传递	168
12.2 外部传参	169
12.2.1 通信协议支持	169
12.2.2 传参接口定义	170
12.3 最佳实践	171
12.3.1 防止参数互相覆盖	171
12.3.2 参数生命周期管理	171
12.4 总结	172
第 13 章 服务多版本	173
13.1 服务多版本管理设计	174
13.1.1 服务版本号管理	174
13.1.2 服务提供者	175
13.1.3 服务消费者	175
13.1.4 基于版本号的服务路由	176
13.1.5 服务热升级	177
13.2 与 OSGi 的对比	178
13.2.1 模块化开发	179

13.2.2	插件热部署和热升级	184
13.2.3	不使用 OSGi 的其他理由	185
13.3	总结	185
第 14 章	流量控制	186
14.1	静态流控	187
14.1.1	传统静态流控设计方案	187
14.1.2	传统方案的缺点	188
14.1.3	动态配额分配制	188
14.1.4	动态配额申请制	190
14.2	动态流控	191
14.2.1	动态流控因子	192
14.2.2	分级流控	192
14.3	并发控制	193
14.3.1	服务端全局控制	193
14.3.2	服务消费者流控	194
14.4	连接控制	195
14.4.1	服务端连接数流控	195
14.4.2	服务消费者连接数流控	195
14.5	并发和连接控制算法	195
14.6	总结	197
第 15 章	服务降级	198
15.1	屏蔽降级	199
15.1.1	屏蔽降级的流程	199
15.1.2	屏蔽降级的设计实现	200
15.2	容错降级	202
15.2.1	容错降级的工作原理	202
15.2.2	运行时容错降级	204
15.3	业务层降级	205
15.4	总结	205

第 16 章 服务优先级调度	207
16.1 设置服务优先级	208
16.2 线程调度器方案	209
16.3 Java 优先级队列	210
16.4 加权优先级队列	211
16.5 服务迁入迁出	212
16.6 总结	213
第 17 章 服务治理	214
17.1 服务治理技术的历史变迁	215
17.1.1 SOA Governance	215
17.1.2 分布式服务框架服务治理	217
17.1.3 AWS 云端微服务治理	217
17.2 应用服务化后面临的挑战	218
17.2.1 跨团队协作问题	219
17.2.2 服务的上下线管控	220
17.2.3 服务安全	220
17.2.4 服务 SLA 保障	221
17.2.5 故障快速定界定位	221
17.3 服务治理	222
17.3.1 服务治理架构设计	223
17.3.2 运行态服务治理功能设计	225
17.3.3 线下服务治理	232
17.3.4 安全和权限管理	234
17.4 总结	237
第 18 章 分布式消息跟踪	239
18.1 业务场景分析	240
18.1.1 故障的快速定界定位	240
18.1.2 调用路径分析	241
18.1.3 调用来源和去向分析	242

18.2 分布式消息跟踪系统设计	242
18.2.1 系统架构	243
18.2.2 埋点日志	244
18.2.3 采样率	247
18.2.4 采集和存储埋点日志	248
18.2.5 计算和展示	249
18.2.6 调用链扩展	251
18.3 总结	251
第 19 章 可靠性设计	253
19.1 服务状态检测	254
19.1.1 基于服务注册中心状态检测	254
19.1.2 链路有效性状态检测机制	255
19.2 服务健康度检测	256
19.3 服务故障隔离	257
19.3.1 进程级故障隔离	257
19.3.2 VM 级故障隔离	259
19.3.3 物理机故障隔离	260
19.3.4 机房故障隔离	261
19.4 其他可靠性特性	262
19.4.1 服务注册中心	262
19.4.2 监控中心	262
19.4.3 服务提供者	262
19.5 总结	263
第 20 章 微服务架构	264
20.1 微服务架构产生的历史背景	265
20.1.1 研发成本挑战	265
20.1.2 运维成本高	267
20.1.3 新需求上线周期长	268
20.2 微服务架构带来的改变	268
20.2.1 应用解耦	268

20.2.2	分而治之	270
20.2.3	敏捷交付	271
20.3	微服务架构解析	271
20.3.1	微服务划分原则	272
20.3.2	开发微服务	272
20.3.3	基于 Docker 容器部署微服务	274
20.3.4	治理和运维微服务	277
20.3.5	特点总结	278
20.4	总结	279
第 21 章 服务化最佳实践		280
21.1	性能和时延问题	281
21.1.1	RPC 框架高性能设计	281
21.1.2	业务最佳实践	285
21.2	事务一致性问题	286
21.2.1	分布式事务设计方案	287
21.2.2	分布式事务优化	288
21.3	研发团队协作问题	289
21.3.1	共用服务注册中心	290
21.3.2	直连提供者	290
21.3.3	多团队进度协同	291
21.3.4	服务降级和 Mock 测试	291
21.3.5	协同调试问题	292
21.3.6	接口前向兼容性	292
21.4	总结	292

第 1 章

应用架构演进

随着业务的发展，应用规模不断扩大，系统内部的巨无霸应用越来越多，常规的垂直应用架构已经无法应对复杂业务带来的各种挑战。通过将业务公共能力抽象成原子服务，对复杂应用进行水平拆分和服务化，实现服务消费者和提供者的解耦。公共能力抽取和复用，可以有效降低公共模块重复开发建设的成本。

传统垂直架构改造的核心就是要对应用做服务化改造，服务化改造使用到的核心技术架构就是分布式服务框架。

本章对应用架构的演进历史进行剖析，使读者能够更清晰和全面地了解应用架构的历史演进过程以及未来架构的发展方向。

1.1 传统垂直应用架构

在 2006 年之前，业界比较流行的有：LAMP 架构，即 Linux + Apache + PHP（前后台界面和业务逻辑）+ MySQL 数据库（读写分离）；MVC 架构，即 Spring + Struts + iBatis/Hibernate + Tomcat；厚重的 EJB 企业架构也流行过很长一段时间。

尽管上述三种架构的技术实现细节存在较大差异，但它们有一个共性：即垂直应用架构。垂直应用架构技术比较单一，学习成本低，开发上手较快，测试、部署和运维也比较简单，因此在很长一段时期里都占据着统治地位。

1.1.1 垂直应用架构介绍

下面以经典的 MVC 垂直架构为例，对垂直应用架构的特点进行分析总结。

MVC 垂直架构逻辑架构图示例如图 1-1 所示。

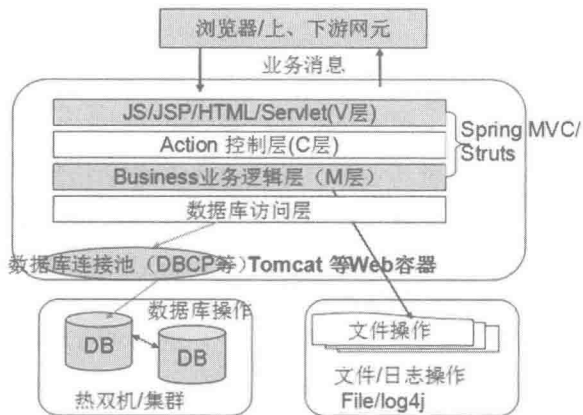


图 1-1 MVC 垂直架构

MVC 架构通常分为三层：

- 1) 最前端是视图展示层(View)，主要用于前端 Portal 展示，使用的开发工具为 JSP、

JS、HTML+CSS 等。视图是用户看到并与之交互的界面，视图向用户显示相关的业务数据，并能接收用户的输入。视图层并不执行实际的业务逻辑，也不改变数据模型。它能接收模型发出的数据更新请求，从而对用户界面进行同步更新。

- 2) 中间为调度控制层 (Control)，主要用于前端 Web 请求的分发，调度后台的业务逻辑执行，可以通过继承 Struts 的 Action 实现。当 Web 用户单击 Web 页面中的提交按钮来发送 HTML 表单时，控制器接收请求并调用相应的模型组件去处理请求，然后调用相应的视图来显示模型返回的数据。
- 3) 第三层为应用模型层 (Model)，模型是应用程序的主体部分。模型代表了业务数据和业务执行逻辑。当数据发生变化时，它要负责通知视图部分。一个模型可以同时为多个视图提供数据，由于同一个模型可以被多个视图重用，所以提高了应用的可重用性。

标准的 MVC 模式并不包含数据访问层，但是在实际开发过程中，通常需要专门的数据库连接池和统一的数据库访问接口对接数据库。于是 ORM 框架逐渐流行起来，iBatis、Hibernate 等 ORM 框架屏蔽了底层的数据库连接池和数据源实现，同时提供了诸如 Naming-SQL、面向对象的数据查询接口等 JDBC 上层封装，极大地降低了使用原生 JDBC 驱动开发的成本，提升了开发效率。

通常基于 MVC 架构开发的应用代码会统一打成一个 war 包，部署在 Tomcat 等 Web 容器中。不同的应用功能之间通过本地 API 进行调用，基本不存在跨进程的远程服务调用。

业务的组网也非常简单，在小规模应用场景下，通常只需要做热双机即可，服务端监听浮动 IP，通过 Watch Dog 监测应用进程，判断应用进程宕机或者僵死之后，将应用切换到备机中，然后尝试重新拉起主机。双机部署方案逻辑组网示意图如图 1-2 所示。

在高并发、大流量的应用场景中，需要做集群，通常的组网方案是前端通过 F5 等负载均衡器做七层负载均衡（或者使用 SLB 等软负载），后端做对等集群部署，它的逻辑组网如下图 1-3 所示。

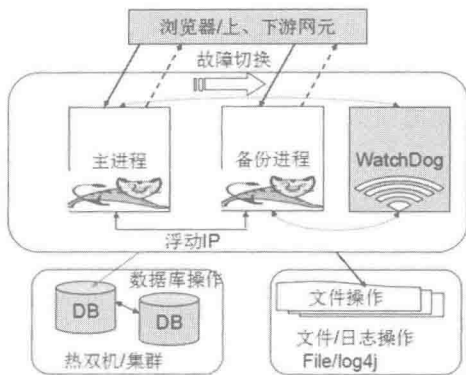


图 1-2 双机逻辑组网图

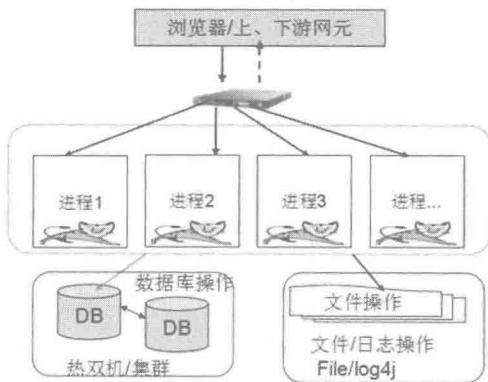


图 1-3 带负载均衡的集群组网

1.1.2 垂直应用架构面临的挑战

在业务发展初期，应用规模比较小，基于 JEE 构建的垂直应用架构还能够有效支撑业务的发展。随着业务的不断发展，应用规模日趋庞大，传统垂直架构开发模式的弊端变得越来越突出，具体如下：

- 1) 复杂应用的开发维护成本变高，部署效率逐渐降低。以实际项目为例，2007 年我参与一个基于传统 MVC 垂直架构开发的企业 ERP 系统，由于业务功能不断膨胀，

最后代码全量编译和部署一次需要 15 分钟。更为严重的是只要某个功能编译出错或者功能测试出问题，就需要重新打包编译，软件的部署效率极低。

- 2) 团队协作效率差，部分公共功能重复开发，代码重复率居高不下。随着业务的发展，团队规模不断扩大，研发被打散到不同的开发小组中。不同的功能模块可能会依赖一些公共能力组件，由于没有类似服务化这种技术契约约束，也很难在不同团队间实现无缝沟通和共享，加之公共能力都是些本地 API 实现，这就导致公共 API 被重复开发，不能在团队间有效重用，这会导致编写大量的重复代码。
- 3) 系统可靠性变差。随着业务的发展，访问量逐渐攀升，网络流量、负载均衡、数据库连接等都面临巨大压力。某个节点的故障会导致分摊到其他节点的流量陡增，引起“雪崩效应”，高并发、大流量对系统的可靠性要求非常高。垂直架构将所有的应用模块都部署到同一个进程中，如果某个应用接口发生故障，例如内存泄漏，会导致整个节点宕机。由于是对等集群部署，这就意味着其他节点也有类似问题，宕机可能会此起彼伏，严重影响业务的正常运行。
- 4) 维护和定制困难。由于业务代码不断膨胀，功能越来越复杂，已有垂直架构模式下无法对复杂的业务进行拆分，代码修改牵一发而动全身，维护和定制都非常困难。
- 5) 新功能上线周期变长。主要有两个原因导致交付效率下降。(1) 公共 API 变更导致测试工作量激增。前面已经介绍过垂直架构存在大量的重复代码，当某个被重复开发的公共 API 发生变更之后，会导致多处代码需要重复修改，重复测试，这会引入大量的回归测试工作量；(2) 新特性无法独立部署和交付。新功能需要跟其他老功能一起编译、打包和测试，如果测试出 Bug，整个系统需要重新打包和部署，这种强耦合会导致整个交付效率下降。究其根本是传统的垂直架构无法做到按照服务或者特性独立交付和运维。

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速地响应多变的市场需求。同时将公共能力 API 抽取出来，作为独立的公共服务供其他调用者消费，以实现服务的共享和重用，

降低开发和运维成本。应用拆分之后会按照模块独立部署，接口调用由本地 API 演进成跨进程的远程方法调用，此时 RPC 框架应运而生。

1.2 RPC 架构

RPC 的全称是 Remote Procedure Call，它是一种进程间通信方式。允许像调用本地服务一样调用远程服务，它的具体实现方式可以不同，例如 Spring 的 HTTP Invoker，Facebook 的 Thrift 二进制私有协议通信。

RPC 概念术语在上世纪 80 年代由 Bruce Jay Nelson 提出，在他的论文中对 RPC 进行了如下总结。

- 1) 简单：RPC 概念的语义十分清晰和简单，这样建立分布式计算就更容易。
- 2) 高效：过程调用看起来十分简单而且高效。
- 3) 通用：在单机计算中过程往往是不同算法和 API，跨进程调用最重要的是通用的通信机制。

2006 年之后，随着移动互联网的发展，各种智能终端的普及，远程分布式调用已经成为主流，RPC 框架也如雨后春笋般诞生，开源和自研的 RPC 框架的普及标志着传统垂直应用架构时代的终结。

1.2.1 RPC 框架原理

RPC 框架的目标就是让远程过程（服务）调用更加简单、透明，RPC 框架负责屏蔽底层的传输方式（TCP 或者 UDP）、序列化方式（XML/JSON/二进制）和通信细节。框架使用者只需要了解谁在什么位置提供了什么样的远程服务接口即可，开发者不需要关心底层通信细节和调用过程。

RPC 框架的调用原理图如图 1-4 所示。

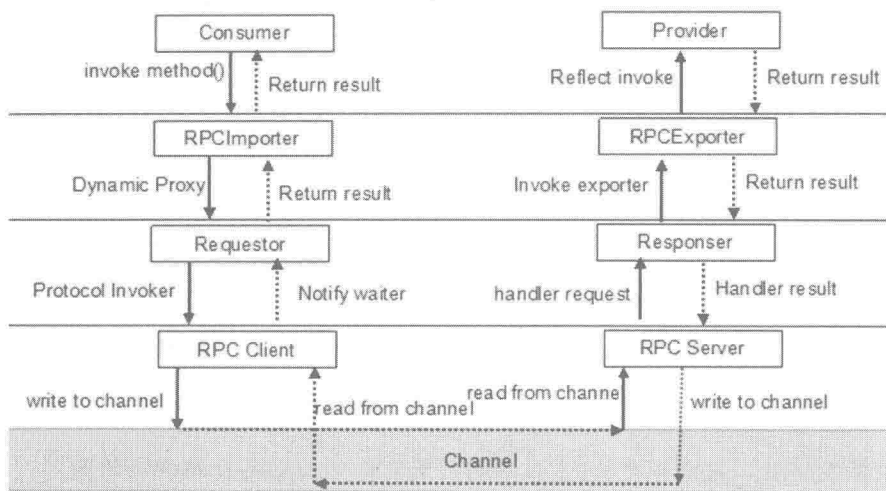


图 1-4 RPC 框架原理图

RPC 框架实现的几个核心技术点总结如下。

- 1) 远程服务提供者需要以某种形式提供服务调用相关的信息，包括但不限于服务接口定义、数据结构，或者中间态的服务定义文件，例如 Thrift 的 IDL 文件，WS-RPC 的 WSDL 文件定义，甚至也可以是服务端的接口说明文档；服务调用者需要通过一定的途径获取远程服务调用相关信息，例如服务端接口定义 Jar 包导入，获取服务端 IDL 文件等。
- 2) 远程代理对象：服务调用者调用的服务实际是远程服务的本地代理，对于 Java 语言，它的实现就是 JDK 的动态代理，通过动态代理的拦截机制，将本地调用封装成远程服务调用。
- 3) 通信：RPC 框架与具体的协议无关，例如 Spring 的远程调用支持 HTTP Invoke、RMI Invoke，MessagePack 使用的是私有的二进制压缩协议。
- 4) 序列化：远程通信，需要将对象转换成二进制码流进行网络传输，不同的序列化框架，支持的数据类型、数据包大小、异常类型及性能等都不同。不同的 RPC 框

架应用场景不同,因此技术选择也会存在很大差异。一些做得比较好的 RPC 框架,可以支持多种序列化方式,有的甚至支持用户自定义序列化框架(Hadoop Avro)。

分析完 RPC 框架的原理之后,下节我们将不依赖任何第三方类库,自己实现一个 Java 版的最简单 RPC 框架,总代码量不超过 200 行。

1.2.2 最简单的 RPC 框架实现

下面通过 Java 原生的序列化、Socket 通信、动态代理和反射机制,实现最简单的 RPC 框架。它由三部分组成:

- 1) 服务提供者,它运行在服务端,负责提供服务接口定义和服务实现类。
- 2) 服务发布者,它运行在 RPC 服务端,负责将本地服务发布成远程服务,供其他消费者调用。
- 3) 本地服务代理,它运行在 RPC 客户端,通过代理调用远程服务提供者,然后将结果进行封装返回给本地消费者。

下面看具体实现,首先是服务端接口定义和实现,代码如下:

代码清单 1-1 EchoService 接口定义

```
public interface EchoService {  
    String echo(String ping);  
}
```

EchoService 接口的实现类代码如下:

代码清单 1-2 EchoServiceImpl 代码

```
public class EchoServiceImpl implements EchoService {  
    @Override  
    public String echo(String ping) {  
        return ping != null ? ping + " --> I am ok." : " I am ok.";  
    }  
}
```

RPC 服务端服务发布者代码实现如下：

代码清单 1-3 RpcExporter 代码

```
public class RpcExporter {
    static Executor executor
= Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    public static void exporter(String hostName, int port) throws Exception
    {
        ServerSocket server = new ServerSocket();
        server.bind(new InetSocketAddress(hostName, port));
        try
        {
            while(true)
            {
                executor.execute(new ExporterTask(server.accept()));
            }
        }
        finally
        {
            server.close();
        }
    }

    private static class ExporterTask implements Runnable
    {
        Socket client = null;
        public ExporterTask(Socket client)
        {
            this.client = client;
        }

        @Override
        public void run() {
```

```

ObjectInputStream input = null;
ObjectOutputStream output = null;
try
{
    input = new ObjectInputStream(client.getInputStream());
    String interfaceName = input.readUTF();
    Class<?> service = Class.forName(interfaceName);
    String methodName = input.readUTF();
    Class<?>[] parameterTypes = (Class<?>[])input.readObject();
    Object[] arguments = (Object[])input.readObject();
    Method method = service.getMethod(methodName, parameterTypes);
    Object result = method.invoke(service.newInstance(), arguments);
    output = new ObjectOutputStream(client.getOutputStream());
    output.writeObject(result);
}
catch(Exception e)
{
    e.printStackTrace();
}
finally
{
    if (output != null)
        try {
            output.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    if (input != null)
        try {
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
}

```

```

        if (client != null)
            try {
                client.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
}
}}}

```

服务发布者的主要职责如下：

- 1) 作为服务端，监听客户端的 TCP 连接，接收到新的客户端连接之后，将其封装成 Task，由线程池执行。
- 2) 将客户端发送的码流反序列化成对象，反射调用服务实现者，获取执行结果。
- 3) 将执行结果对象反序列化，通过 Socket 发送给客户端。
- 4) 远程服务调用完成之后，释放 Socket 等连接资源，防止句柄泄漏。

RPC 客户端本地服务代理源码如下：

代码清单 1-4 RpcImporter 代码

```

public class RpcImporter<S> {
    public S importer(final Class<?> serviceClass, final InetSocketAddress addr)
    {
        return (S) Proxy.newProxyInstance(serviceClass.getClassLoader(),
new Class<?>[] {serviceClass.getInterfaces()[0]},
        new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable
            {
                Socket socket = null;
                ObjectOutputStream output = null;

```

```

ObjectInputStream input = null;
try
{
    socket = new Socket();
    socket.connect(addr);
    output = new ObjectOutputStream(socket.getOutputStream());
    output.writeUTF(serviceClass.getName());
    output.writeUTF(method.getName());
    output.writeObject(method.getParameterTypes());
    output.writeObject(args);
    input = new ObjectInputStream(socket.getInputStream());
    return input.readObject();
}
finally
{
    if (socket != null)
        socket.close();
    if (output != null)
        output.close();
    if (input != null)
        input.close();
}
}
});
}
}

```

本地服务代理的主要功能如下：

- 1) 将本地的接口调用转换成 JDK 的动态代理，在动态代理中实现接口的远程调用。
- 2) 创建 Socket 客户端，根据指定地址连接远程服务提供者。
- 3) 将远程服务调用所需的接口类、方法名、参数列表等编码后发送给服务提供者。

4) 同步阻塞等待服务端返回应答, 获取应答之后返回。

完成 RPC 框架代码开发之后, 我们接着编写测试代码:

代码清单 1-5 RpcTest 代码

```
public static void main(String[] args) throws Exception {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                RpcExporter.exporter("localhost", 8088);  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }).start();  
    RpcImporter<EchoService> importer = new RpcImporter<EchoService> ();  
    EchoService echo = importer.importer(EchoServiceImpl.class, new  
InetSocketAddress("localhost", 8088));  
    System.out.println(echo.echo("Are you ok ?"));  
}
```

首先, 创建一个异步发布服务端的线程并启动, 用于接收 RPC 客户端的请求, 根据请求参数调用服务实现类, 返回结果给客户端。

随后, 创建客户端服务代理类, 构造 RPC 请求参数, 发起 RPC 调用, 将调用结果输出到控制台上, 执行结果如图 1-5 所示。

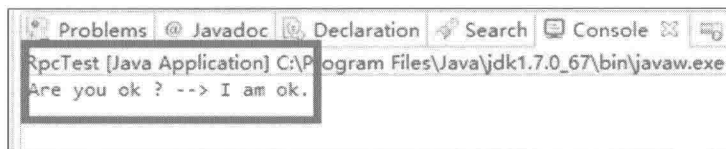


图 1-5 RPC 框架测试结果

1.2.3 业界主流 RPC 框架

业界开源的 RPC 框架非常多，比较主流的 RPC 框架举例如下：

- 1) 由 Facebook 开发的远程服务调用框架 Apache Thrift。
- 2) Hadoop 的子项目 Avro-RPC。
- 3) caucho 提供的基于 binary-RPC 实现的远程通信框架 Hessian。
- 4) Google 开源的基于 HTTP/2 和 ProtoBuf 的通用 RPC 框架 gRPC。

下面我们分别对上述四种 RPC 框架进行简单介绍，首先了解下 Apache Thrift。Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用框架。它采用接口描述语言（IDL）定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中创建高效无缝的服务，如 C++、Java、Python、PHP、Ruby、Erlang、Perl、C#、Cocoa、Smalltalk 等。其传输数据采用二进制格式，相对 XML 和 JSON 等序列化方式体积更小，对于高并发、大数据量和多语言的环境更有优势。

Thrift 服务包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式运行在服务器上，可以和现有的 J2EE 服务器 /Web 容器无缝对接。Thrift 支持的基本类型：bool、byte、i16、i32、i64、double、string；结构体类型：struct（Java POJO 对象）；容器类型：list、set、map；异常类型：exception。它可以满足各种语言复杂数据结构的传输。

Thrift 允许开发者选择客户端与服务端之间通信协议的类别，在传输协议上总体可以划分为文本和二进制传输协议。为节约带宽，提高传输效率，一般情况下使用二进制传输协议，在性能要求不高的场合，为了提高可读性有时也会使用文本类型的协议，常用的协议有以下几种。

- 1) TBinaryProtocol：二进制编码格式数据传输协议。
- 2) TCompactProtocol：高效率的压缩二进制编码格式数据传输协议。
- 3) TJSONProtocol：使用 JSON 编码的数据传输协议。

Thrift 支持的通信方式有如下几种。

- 1) TSocket: 使用阻塞式 I/O 进行传输, 最简单常用的模式。
- 2) TFramedTransport: 使用非阻塞方式, 按块的大小进行传输, 类似于 Java 中的 NIO 非阻塞通信。
- 3) TNonblockingTransport: 使用非阻塞方式, 用于构建异步客户端。

Apache Avro 是 Hadoop 下的一个子项目, 它本身既是一个序列化框架, 同时也实现了 RPC 的功能, 它的特性如下:

- 1) 丰富的数据结构类型。
- 2) 快速可压缩的二进制数据形式。
- 3) 存储持久数据的文件容器。
- 4) 提供远程过程调用 RPC。
- 5) 简单的动态语言结合功能。

相比于 Apache Thrift 和 Google 的 Protocol Buffer, Apache Avro 具有以下特点:

- 1) 支持动态模式。Avro 不需要生成代码, 这有利于搭建通用的数据处理系统, 同时避免了对业务代码的侵入。
- 2) 数据无须加标签。读取数据前, Avro 能够获取模式定义, 这使得 Avro 在数据编码时只需要保留更少的类型信息, 有利于减少序列化后的数据大小。
- 3) 无须手工分配的域标识。Thrift 和 Protocol Buffers 使用一个用户添加的整型域唯一性定义一个字段, 而 Avro 则直接使用域名, 该方法更加直观、更加易扩展。

Apache Avro 的可定制性非常好, 主要体现在如下几个方面: 传输层和业务逻辑层分离, 用户可以专注于业务逻辑开发; 服务端有一个协议注册工厂和序列化注册工厂, 针对不同的应用场景用户可以定制私有协议和不同的序列化方式, 满足业务领域服务的需求。

客户端支持同步和异步调用，用户可以根据实际业务场景做不同的选择。

Hessian 是一个由 Caucho Technology 开发的轻量级二进制 RPC 框架，Hessian 通过 Servlet 提供远程服务，可以将某个请求映射到 Hessian 服务。Spring 的 DispatcherServlet 支持该功能，DispatcherServlet 可将匹配模式的请求转发到 Hessian 服务。Hessian 的 Server 端提供一个 Servlet 基类，用来处理发送的请求，而 Hessian 的远程过程调用则使用动态代理来实现，采用面向接口编程。因此，Hessian 服务通常通过 Java 接口对外暴露。

Hessian 的优点如下：简单易用，面向接口编程，通过接口暴露服务，轻量级，可以穿透防火墙；采用二进制传输，序列化效率高；支持多语言，包括 Java、Python、C++、.NET、C#、PHP、Ruby 等；可与 spring 集成，配置比较简单。

gRPC 是一个高性能、通用的开源 RPC 框架，其由 Google 主要面向移动应用开发并基于 HTTP/2 协议标准而设计，基于 ProtoBuf（Protocol Buffers）序列化协议开发，支持众多开发语言。gRPC 基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

gRPC 目前提供 C、Java 和 Go 语言版本，这三个版本的源码全都托管在 Github 上，分别是 grpc、grpc-java、grpc-go。其中 C 版本支持 C、C++、Node.js、Ruby、Objective-C、PHP 和 C#等。

gRPC 框架的主要特性如下。

- 1) 支持 Protobuf: gRPC 使用 ProtoBuf 的 IDL(.proto)来定义数据结构和 service。Protobuf 是一个灵活、高效、结构化的数据序列化框架，相比于 XML 等传统的序列化工具，它更小、更快、更简单。Protobuf 支持数据结构化一次可以到处使用，甚至跨语言使用，通过代码生成工具可以自动生成不同语言版本的源代码，甚至可以在使用不同版本的数据结构进程间进行数据传递，实现数据结构的前向兼容。当前 gRPC 仅支持 Protobuf，且不支持在浏览器中使用。由于 gRPC 的设计能够支持多种数据格式，所以读者能够很容易实现对其他数据格式（如 XML、JSON 等）的支持。

- 2) 支持多种语言: gRPC 支持多种语言, 并能够根据语言类型自动生成客户端和服务端代码, grpc-java 已经支持 Android 开发。
- 3) 基于 HTTP/2 设计: 由于 gRPC 基于 HTTP/2 标准设计, 所以相对于其他 RPC 框架, gRPC 带来了更多强大功能, 如双向流、头部压缩、多复用请求等。这些功能给移动设备带来重大益处, 如节省带宽、降低 TCP 链接次数、节省 CPU 使用和延长电池寿命等。同时, gRPC 还能够提高了云端服务和 Web 应用的性能, gRPC 既能够在客户端应用, 也能够服务器端应用, 从而以透明的方式实现客户端和服务端通信和简化通信系统的构建。

1.2.4 RPC 框架面临的挑战

在大规模服务化之前, 应用可能只是通过 RPC 框架, 简单的暴露和引用远程服务, 通过配置服务的 URL 地址进行远程服务调用, 路由则通过 F5 硬件负载均衡器或者 SLB 进行简单的负载均衡。

当服务越来越多时, 服务 URL 配置管理变得非常困难, F5 等硬件负载均衡器的单点压力也越来越大。此时需要一个服务注册中心, 动态地注册和发现服务, 使服务的位置透明。消费者在本地缓存服务提供者列表, 实现软负载均衡, 这可以降低对 F5 等硬件负载均衡器的依赖, 也能降低硬件成本。

随着业务的发展, 服务间依赖关系变得错综复杂, 甚至分不清哪个应用要在哪个应用之前启动, 架构师都不能完整地描述应用之间的调用关系。需要一个分布式消息跟踪系统可视化展示服务调用链, 用于依赖分析、业务调用路径梳理等, 帮助架构师清理不合理的服务依赖, 防止业务服务架构腐化。

服务的调用量越来越大, 服务的容量问题就暴露出来了, 某个服务需要多少机器支撑、什么时候该加机器? 为了解决容量规划问题, 需要采集服务调用 KPI 数据, 进行汇总和分析, 通过计算得出服务部署实例数和服务器的配置规格。

服务上线容易下线难, 上线前的审批, 下线通知, 需要统一的服务生命周期管理流程

进行管控。不同的服务安全权限不同，如何保证敏感服务不被误调用？服务的访问安全策略又如何制定？

服务化之后，随之而来的就是服务治理问题。目前业界开源纯粹的 RPC 框架服务治理能力都不健全，当应用大规模服务化之后会面临许多服务治理方面的挑战，要解决这些问题，必须通过服务框架 + 服务治理来完成，单凭 RPC 框架无法解决服务治理问题。

1.3 SOA 服务化架构

SOA 是一种粗粒度、松耦合的以服务为中心的架构，接口之间通过定义明确的协议和接口进行通信。SOA 帮助工程师们站在一个新的高度理解企业级架构中各种组件的开发和部署形式，它可以帮助企业系统架构师以更迅速、可靠和可重用的形式规划整个业务系统。相比于传统的非服务化架构，SOA 能够更加从容地应对复杂企业系统集成和需求的快速变化。

1.3.1 面向服务设计的原则

SOA 面向服务的一般原则总结如下。

- 1) 服务可复用：不管是否存在即时复用的机会，服务均被设计为支持潜在的可复用。
- 2) 服务共享一个标准契约：为了与服务提供者交互，消费者需要导入服务提供者的服务契约，这个契约可以是一个 IDL 文件、Java 接口定义、WSDL 文件，甚至是个接口说明文档。
- 3) 服务是松耦合的：服务被设计为功能相对独立、尽量不依赖其他服务的独立功能提供者。
- 4) 服务是底层逻辑的抽象：只有经服务契约所暴露的服务对外部世界可见，契约之外底层的实现逻辑是不可见的。

- 5) 服务是可组合、可编排的：多个服务可能被编排组合成一个新的服务，这允许不同逻辑抽象地自由组合，促进服务的复用。
- 6) 服务是自治的：逻辑由服务所控制，并位于一个清晰的边界内，服务已经在边界内被控制，不依赖于其他服务。
- 7) 服务是无状态的：服务应当不需要管理状态信息，因此能够维持松耦合性。服务应当被尽可能设计成无状态，即便这意味着要将状态管理移至他处。
- 8) 服务是可被自动发现的：服务发布上线后，允许被其他消费者自动发现；当服务提供者下线后，允许消费者接收服务下线通知。

1.3.2 服务治理

SOA 服务化之后，随着业务的不断发展，服务数量越来越多，应用服务化之后给系统运维带来很大的挑战：

- 1) 分布式框架下的服务调用性能。
- 2) 服务化架构如何支持线性扩展。
- 3) 如何实现高效、实时的服务多维度监控。
- 4) 大规模分布式环境下的故障快速定界和定位。
- 5) 分布式环境下海量日志在线检索、模糊查询。
- 6) 服务的流控、超时控制、服务升降机等管控手段。
- 7) 服务的划分原则，如何实现最大程度复用。
- 8)

此时，SOA 服务治理是关键。SOA 服务治理主要包括如下几个方面。

- 1) 服务定义：SOA 治理最基础的方面就是监视服务的创建过程。必须对服务进行标

识，描述其功能，确定其行为范围并设计其接口。创建服务时需要与使用这些服务的团队进行协调，以确保服务能够满足消费者需求，避免重复工作。

- 2) 服务生命周期管理：服务需要进行规划、设计、实现、部署、维护，并最后下线。服务开发生命周期通常都具有五个主要的阶段。计划阶段——已标识了新服务并正在设计中，不过尚未实现或正在实现中；测试阶段——服务完成开发后，上线前必须对服务进行测试，有些测试可能需要在灰度环境中执行，此环境会作为服务上线前的测试床来模拟生产环境进行最后验证；运行阶段——生产环境中正在运行的服务；弃用阶段——尽管该服务仍然在生产环境中运行，但是不会长时间运行，过一段时间将会下线，警告服务消费者尽快停止使用此服务；废弃阶段——这是服务的最后一个阶段，表示一个生命周期结束的服务。注册中心可以将该服务物理删除，也可以逻辑删除，标识该服务已经被废弃。
- 3) 服务版本治理：服务发布后不久，服务提供者就开始根据需要对服务进行修改。包括问题修复，添加新的功能，有时还需要删除非必需的功能。服务修改之后，需要对服务进行重新编译和打包，发布新版本的服务。新版本的前向兼容性，灰度发布等需要按照统一的策略进行管理。
- 4) 服务注册中心：服务提供者如何发布服务？服务消费者如何订阅服务？已经发布的服务如何重新注册和下线，新注册的服务如何被消费者动态发现，需要统一的服务注册中心支持服务的订阅发布和动态发现机制。
- 5) 服务监控：服务监控中心需要对服务的调用时延、成功率、吞吐率等数据进行实时采样和汇总，通过图形化报表的形式展示，以方便运维人员对服务的运行质量进行实时分析和掌控。
- 6) 运行期服务质量保障：包括服务限流、服务迁入迁出、服务升降机、服务权重调整和服务超时控制等，通过运行期的动态治理，可以在不重启服务的前提下达到快速提升服务运行质量的目标。
- 7) 快速的故障定界定位手段：故障定界定位主要包括两方面的内容。(1) 大规模分布式环境下海量业务/平台日志的采集、汇总和实时在线检索，支持多维度的条件

检索、模糊查询，可以快速地在线查看各种系统运行日志，方便问题定位；（2）分布式消息跟踪，通过调用链打通业务、服务调用和异常，发现线上系统故障源；通过在线和离线调用链大数据分析，得到链路各个依赖的稳定性指标，梳理依赖链路风险表，识别系统核心功能的服务调用依赖关系，评估可能的最大风险点，针对性改进以预防风险，同时为容量规划和扩容提供数据决策依据。

- 8) 服务安全：是否允许任何人调用任何服务，数据敏感型服务是否允许所有用户访问所有数据，服务使用者和提供者之间交换的数据是否需要保护，服务是否需要做安全认证，对内对外安全策略差异等。服务调用必须能够提供安全功能，对服务的访问进行权限控制，将服务的访问权仅限于授权使用者。用户安全标识需要在服务调用中携带，用于授权敏感数据的访问。服务安全访问策略有多种，例如可以通过动态生成令牌（Token）的方式做安全访问授权，服务提供者动态生成令牌（Token）并告知服务注册中心，由注册中心决定是否告之消费方，这样就能在注册中心页面上做复杂的授权模型。

1.4 微服务架构

微服务架构（MSA）是一种服务化架构风格，通过将功能分散到各个离散的服务中以实现解决方案的解耦。

1.4.1 什么是微服务

为了最大限度复用已有的资产，保护 IT 资产投资，大多数异构系统并没有使用统一的服务化框架进行深入改造。通过将业务接口发布成标准的 Web 服务（例如 SOAP），利用 XML 序列化格式进行数据交换。部分遗留系统由于技术等原因无法改造，直接通过企业服务总线（ESB）进行异构协议的转换、Data Mapper 和消息路由，实现与其他服务的互通。不同组件服务化使用的技术和划分原则不同，SOA 服务化之后的质量也良莠不齐，

部分设计不合理的服务并没有体现服务化之后的价值。

SOA 架构解决了应用服务化问题，随着服务化实践的不断深入，服务规模越来越大，服务治理面临的挑战也越来越多，微服务架构风格应运而生。实际上它的诞生并非偶然：敏捷开发强调迭代开发，快速交付可用的功能；持续交付促使我们构建更快、更可靠、更频繁的软件部署和交付能力；CI 自动化构建帮助我们简化环境的创建、编译、打包和部署；DevOps 的流行促进小团队独立运作和交付。受这些因素的综合影响，以及 SOA 服务化实施多年的经验积累，综合因素共同作用促使微服务架构诞生。

微服务架构的主要特征如下。

- 1) 原子服务，专注于做一件事：与面向对象原则中的“单一职责原则”类似，功能越单一，对其他功能的依赖就越少，内聚性就越强，“高内聚、松耦合”。
- 2) 高密度部署：重要的服务可以独立进程部署，非核心服务可以独立打包，合设到同一个进程中，服务被高密度部署。物理机部署，可以在一台服务器上部署多个服务实例进程；如果是云端部署，则可以利用 LXC（例如 Docker）实现容器级部署，以降低部署成本，提升资源利用率。
- 3) 敏捷交付：服务由小研发团队负责设计、开发、测试、部署、线上治理、灰度发布和下线，运维整个生命周期支撑，实现真正的 DevOps。
- 4) 微自治：服务足够小，功能单一，可以独立打包、部署、升级、回滚和弹性伸缩，不依赖其他服务，实现局部自治。

1.4.2 微服务架构对比 SOA

两者的主要差异如下。

- 1) 服务拆分粒度：SOA 首先要解决的是异构应用的服务化；微服务强调的是服务拆分尽可能小，最好是独立的原子服务。
- 2) 服务依赖：传统的 SOA 服务，由于需要重用已有的资产，存在大量的服务间依赖；

微服务的设计理念是服务自治、功能单一独立，避免依赖其他服务产生耦合，耦合会带来更高的复杂度。

- 3) 服务规模：传统 SOA 服务粒度比较大，多数会采用将多个服务合并打成 war 包的方案，因此服务实例数比较有限；微服务强调尽可能拆分，同时很多服务会独立部署，这将导致服务规模急剧膨胀，对服务治理和运维带来新的挑战。
- 4) 架构差异：微服务化之后，服务数量的激增会引起架构质量属性的变化，例如企业集成总线 ESB（实总线）逐渐被 P2P 的虚拟总线替换；为了保证高性能、低时延，需要高性能的分布式服务框架保证微服务架构的实施。
- 5) 服务治理：传统基于 SOA Governance 的静态治理转型为服务运行态微治理、实时生效。
- 6) 敏捷交付：服务由小研发团队负责微服务设计、开发、测试、部署、线上治理、灰度发布和下线，运维整个生命周期支撑，实现真正的 DevOps。

总结：量变引起质变，这就是微服务架构和 SOA 服务化架构的最大差异。

1.5 总结

到这里，应用架构的演进介绍就结束了。通过对 4 代架构的功能和特性分析，我们可以清晰地了解服务化的历史演进过程，以及未来服务化架构的演进方向。

下面通过服务化架构演进图，总结一下这 4 代架构的演进历史，如图 1-6 所示。

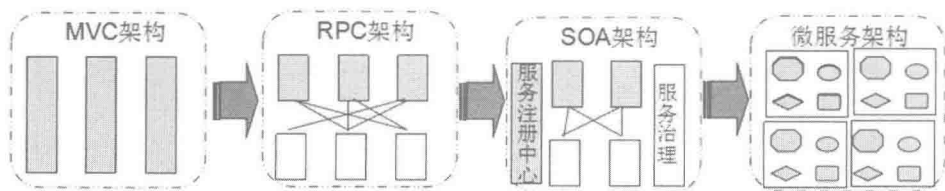


图 1-6 应用架构演进历史

- ◎ MVC 架构：当业务规模很小时，将所有功能都部署在同一个进程中，通过双机或者前置负载均衡器实现负载分流；此时，用于分离前后台逻辑的 MVC 架构是关键。
- ◎ RPC 架构：当垂直应用越来越多，应用之间交互不可避免，将核心和公共业务抽取出来，作为独立的服务，实现前后台逻辑分离。此时，用于提高业务复用及拆分的 RPC 框架是关键。
- ◎ SOA 架构：随着业务发展，服务数量越来越多，服务生命周期管控和运行态的治理成为瓶颈，此时用于提升服务质量的 SOA 服务治理是关键。
- ◎ 微服务架构：随着敏捷开发、持续交付、DevOps 理论的发展和实践，以及基于 Docker 等轻量级容器（LXC）部署应用和服务的成熟，微服务架构开始流行，逐渐成为应用架构的未来演进方向。通过服务的原子化拆分，以及微服务的独立打包、部署和升级，小团队敏捷交付，应用的交付周期将缩短，运维成本也将大幅下降。

分布式服务框架入门

在一个不断发展的大型应用中，新的业务需求和功能不断增加，技术也在不断演进，不同团队构建的功能子系统采用的技术架构五花八门，子系统之间的开发、部署和运维模式也存在较大差异。如果企业内部没有统一的服务框架进行技术层面的拉通，开发和运维效率都将受到很大制约。

传统垂直架构改造的核心就是要对应用进行服务化，服务化改造使用到的核心技术就是分布式服务框架。

2.1 分布式服务框架诞生背景

分布式服务框架的诞生并非偶然，当业务发展到一定阶段之后，会面临很多新的挑战，包括新的商业模式、已有技术架构无法支撑业务快速发展等。为了保证业务的快速、健康发展，往往会产生新的技术架构来应对挑战，分布式服务框架的产生也符合这种技术发展规律。

2.1.1 应用从集中式走向分布式

随着业务的发展，应用的功能越来越多，打包、部署和新特性上线周期也变得越来越长，大流量、高并发的用户访问对后台服务的压力越来越大，我们只能通过不断增加硬件的方式来满足应用的低时延和高吞吐量，如图 2-1 所示。

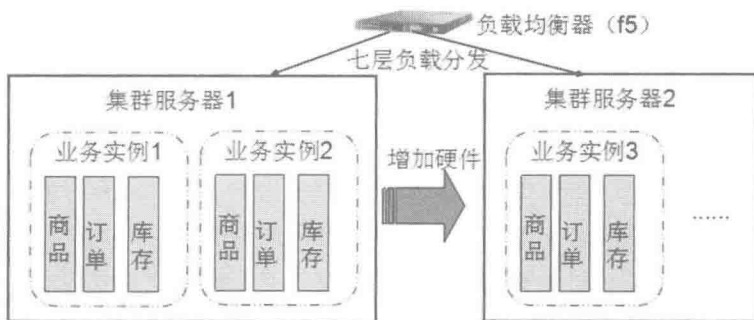


图 2-1 集中式架构通过新增硬件方式扩容

通过新增硬件的方式对应用进行扩容，可以暂时顶住高峰期大并发对系统的冲击，但是仍有一些棘手问题无法通过扩容的方式解决。

- 1) 业务不断发展，应用规模日趋庞大，大型复杂应用的开发维护成本高，部署效率降低；应用数量膨胀，数据库连接数持续变高。
- 2) 代码复用是难题：由于公共模块都是进程内部的本地 API 调用，开发者经常会按需开发，这会导致大量功能类似甚至相同的 API 被重复开发。一旦涉及到公共模

块的功能变更，所有重复实现都需要重新修改、编译和测试，影响范围广，重复测试工作量大。即便一些团队有专门的公共模块开发小组，但是复用的代码往往会由多个开发小组共同出人维护，代码 merge 和分支管理很难做好。

- 3) 敏捷持续交付面临巨大挑战：想要在一个架构师都无法理顺的巨无霸业务中新增或者修改一个功能，难度是非常大的。业务模块之间的循环依赖、重复 API 定义和开发、不合理的调用、冗长复杂的业务流程对新特性的上线简直是梦魇。新需求功能开发完成之后，测试需要对新功能的影响点、配置修改等做综合分析，决定是否做回归测试以及圈定测试范围。等这一系列工作完成之后，发现 2~3 周已经过去，尽管从研发到测试大家都拼命加班，但新需求的上线周期依然达不到老板的预期。

大规模系统架构的设计一般原则就是尽可能地拆分，以达到更好的独立扩展与伸缩、更灵活的部署、更好的隔离和容错、更高的开发效率。

具体的拆分策略大体上可以分为横向拆分和纵向拆分。

- ◎ 纵向拆分：通过对业务进行梳理，根据业务的特性把应用拆开，不同的业务模块独立部署，例如一个 CRM 系统就可以按照客户域、产品域、资源域、营销管理域等拆分，将复杂的业务线拆分成相对独立、灵活的具体能力域，由大变小、分而治之。纵向拆分示意图如图 2-2 所示。

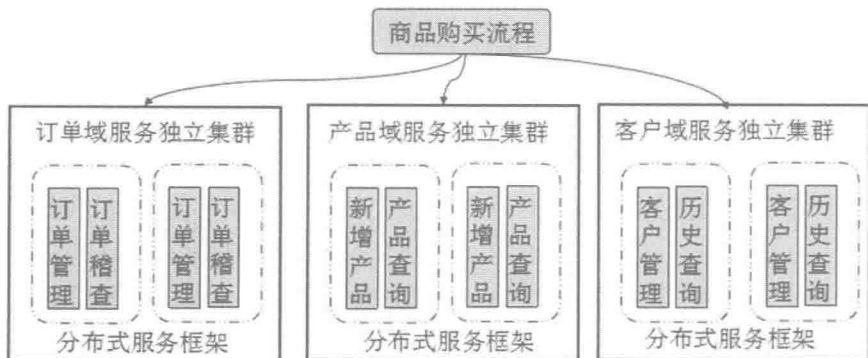


图 2-2 业务纵向拆分

- ◎ 横向拆分：将核心的、公共的业务拆分出来，通过分布式服务框架对业务进行服务化，消费者通过标准的契约来消费这些服务。服务提供者独立打包、部署和演进，与消费者解耦，业务横向拆分之后示意图如图 2-3 所示。

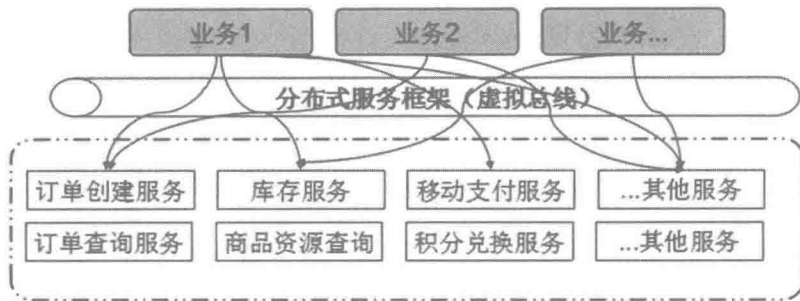


图 2-3 业务横向拆分

2.1.2 亟需服务治理

使用 Web 服务或者 RPC 框架对业务进行拆分之后，随着服务数的增多，亟需一个服务治理框架，有效管控服务，提升服务运行期质量，防止业务服务代码架构腐化。

服务治理的主要诉求如下。

- 1) 生命周期管理：服务上线随意，线上服务鱼龙混杂，上线容易下线难。服务上线前的审批流程、测试发布流程和服务下线通知机制需要规范化。
- 2) 服务容量规划：随着业务的发展，服务调用量越来越大，服务的容量问题逐渐暴露出来，某个服务需要多少机器支撑、什么时候该加机器、加多少台、需要什么样的配置？需要通过采集服务调用性能、时延、成功率、系统资源占用等综合指标，通过对历史数据比对分析，识别服务容量瓶颈。对采集的服务性能数据进行综合计算，得出单核 CPU 能够支持的服务调用量，为容量规划提供精确的数据支撑。
- 3) 运行期治理：大促期间流量陡增，系统资源成为瓶颈，对非核心服务采取降级、限流的措施，保证核心业务的运行；缓存失效时，系统压力转移到数据库，服务调用

时延突然增大，业务失败率升高，需要在线调大服务调用超时时间，保证业务成功率；非核心服务发生故障时，希望对业务放通，不调用远程服务，取而代之的是执行本地的降级逻辑，例如记录放通账单用于事后对账，平台需要支持动态服务降级功能；为了保证服务运行期质量，需要具备强大的在线动态服务治理能力。

- 4) 服务安全：敏感数据服务化之后，如何对消费者鉴权，防止非法的数据访问？相同的服务，对内部消费者和外部第三方消费者提供的能力存在差异，例如屏蔽部分方法，如何对不同的消费者做不同的权限控制？服务化之后，服务的安全控制是服务治理的核心功能之一。

典型的 SOA 服务治理生命周期如图 2-4 所示。

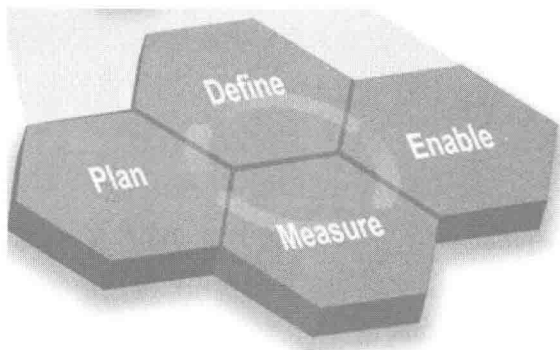


图 2-4 SOA 服务治理生命周期

- ◎ 计划：确定服务治理的重点。
- ◎ 定义：定义服务治理模型。
- ◎ 启用：实现并实施服务治理。
- ◎ 度量：根据实施效果，改进服务治理模型。

2.2 业界分布式服务框架介绍

无论是大型网站的电商平台，还是电信、金融、游戏等行业，都需要一个高性能、低

时延的分布式服务框架来解决业务服务化和服务化之后的治理问题。

目前开源的分布式服务框架有阿里巴巴的分布式服务框架 Dubbo，以及当当网基于 Dubbo 增强而来的 DubboX。

非开源的分布式服务框架，在移动互联网领域主要有淘宝的 HSF、亚马逊内部使用的分布式服务框架 Coral Service。电信行业主要有华为软件开发的分布式服务框架 DSF，它已经在全球数十个商用局点成功商用多年，每天支撑超过 100 亿次服务调用。

需要指出的是，Facebook 开源的 Thrift、Twitter 开源的 Finagle 都是非常优秀的 RPC 框架，在这些互联网巨头内部使用时，一定会有其他辅助的基础设施来支持分布式服务管理、服务自动发现和服务治理。遗憾的是，这部分核心功能目前并没有开源，单纯的 RPC 并不是完整的分布式服务框架，因此在本章中不对它们做介绍，感兴趣的同学可以自行学习。

2.2.1 阿里 Dubbo

Dubbo 是阿里巴巴内部的 SOA 服务化治理方案的核心框架，每天为 2000+ 个服务提供 3 000 000 000+ 次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点。Dubbo 自 2011 年开源后，已被许多非阿里系公司使用。

Dubbo 的功能架构图如图 2-5 所示。

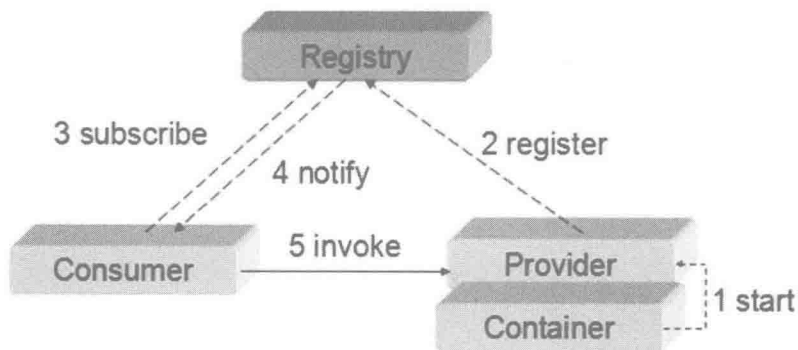


图 2-5 Dubbo 架构图

Dubbo 的工作原理如下：

- 1) 轻量级 Java 容器通过 main 函数初始化 Spring 上下文, 根据服务提供者配置的 XML 文件将服务按照指定协议发布, 完成服务的初始化工作。
- 2) 服务提供者根据配置的服务注册中心地址连接服务注册中心, 将服务提供者信息发布到服务注册中心。
- 3) 消费者根据服务消费者 XML 配置文件的服务引用信息, 连接注册中心, 获取指定服务的地址等路由信息。
- 4) 服务注册中心根据服务订阅关系, 动态地向指定消费者推送服务地址信息。
- 5) 消费者调用远程服务时, 根据路由策略, 从本地缓存的服务提供者地址列表中选择—个服务提供者, 然后根据协议类型建立链路, 跨进程调用服务提供者 (非本地路由优先策略时)。

Dubbo 架构的主要质量属性如下。

1. 连通性

- ◎ 注册中心负责服务地址的注册与查找, 相当于目录服务, 服务提供者和消费者只在启动时与注册中心交互, 注册中心不转发请求, 压力较小。
- ◎ 监控中心负责统计各服务调用次数, 调用时间等, 统计先在内存汇总后每分钟一次发送到监控中心服务器, 并以报表展示。
- ◎ 服务消费者向注册中心获取服务提供者地址列表, 并根据负载算法直接调用提供者, 同时汇报调用时间到监控中心。
- ◎ 注册中心、服务提供者、服务消费者三者之间均为长连接, 监控中心除外。
- ◎ 注册中心通过长连接感知服务提供者的存在, 服务提供者宕机, 注册中心将立即推送事件通知消费者。
- ◎ 注册中心和监控中心全部宕机, 不影响已运行的提供者和消费者, 消费者在本地

缓存了提供者列表。

- ◎ 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

2. 健壮性

- ◎ 监控中心宕掉不影响使用，只是丢失部分采样数据。
- ◎ 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务。
- ◎ 注册中心对等集群，任意一台宕掉后，将自动切换到另一台。
- ◎ 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通信。
- ◎ 服务提供者无状态，任意一台宕掉后，不影响使用。
- ◎ 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复。

3. 伸缩性

- ◎ 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心。
- ◎ 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者。

4. 扩展性

- ◎ 微内核+插件式设计，平等对待第三方：由插件管理容器构成微内核，其他外围功能都通过插件的方式实现，平等对待第三方，使用者可以替换平台的默认实现，也可以通过插件扩展新的功能。
- ◎ 管道式设计，服务调用前后提供拦截面（类 AOP 切面）：通过服务调用前拦截，事后通知的方式，开放服务调用拦截面给框架使用者，用于功能扩展，Dubbo 自身的大多数功能，也基于 Filter 拦截实现。

- ◎ 原子化扩展点，最大化复用和扩展：扩展点是单个功能的抽象，只负责完成一件事，例如序列化框架扩展点，用于扩展 Dubbo 默认提供的序列化方式。

除了 RPC 框架功能，Dubbo 还提供了丰富的服务治理功能，原理如图 2-6 所示。

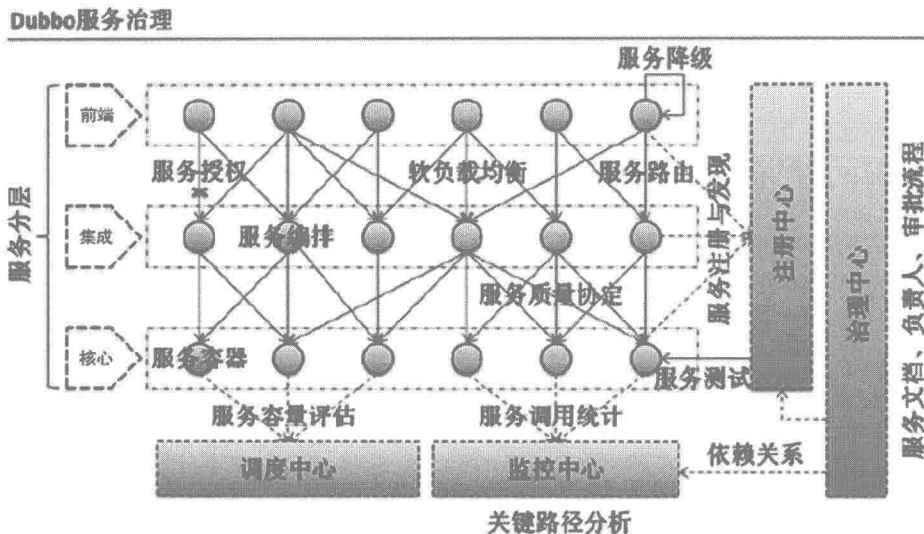


图 2-6 Dubbo 服务治理

2.2.2 淘宝 HSF

HSF 是淘宝的分布式服务框架，它的使用者遍布淘系 1000 多个应用，基于 HSF 发布的服务种类超过 3000 个，集群规模达到 8000+ 台，每天的调用量超过 300 亿次（2012 年的统计数据）。

HSF 的架构图如图 2-7 所示。



图 2-7 HSF 架构图

HSF 框架的功能总结如下。

- 1) 配置化开发,对业务代码低侵入:支持通过 Spring XML 配置方式发布和消费服务,降低对业务代码的侵入。
- 2) 插件管理体系:平台与应用分开部署,运行期依赖,外部采用与应用独立的 classloader 隔离,内部采用 OSGi 隔离。
- 3) 异步 NIO 通信,多种序列化方式:通信框架使用基于 Mina 封装的 TB-Remoting,序列化支持 Java 序列化、Hession 等,服务提供者和消费者之间采用长连接通信。
- 4) 灵活的路由能力:客户端软负载,随机、轮循等多种路由策略,支持容灾和失效恢复等。
- 5) 多协议支持:WebService、PB (Protocol buffer) 和 Hession (HTTP) 等。
- 6) 服务治理:HSF 支持多个纬度的服务治理策略,包括服务监控、服务分组、限流降级、服务授权等。

HSF 服务治理整体结构图如图 2-8 所示。

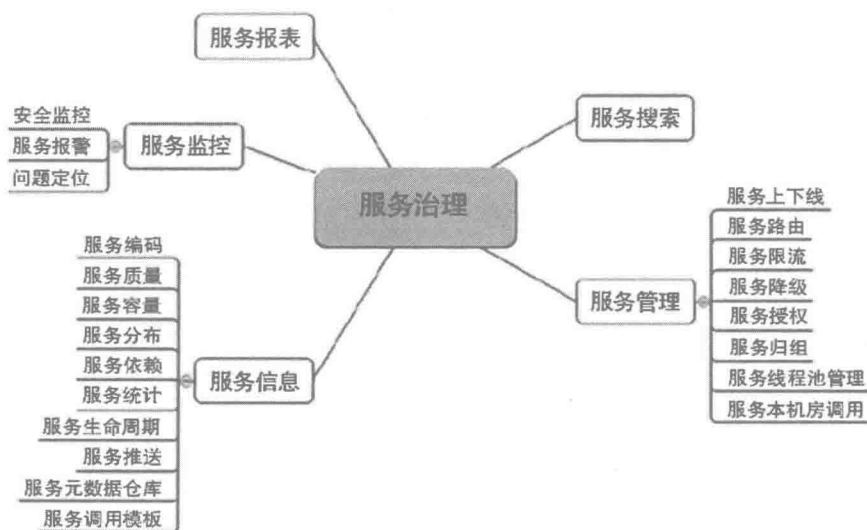


图 2-8 HSF 服务治理结构图

2.2.3 亚马逊 Coral Service

Coral Service（如图 2-9 所示）是亚马逊内部使用的基于 Java 的分布式服务框架，它的功能特点总结如下：

- 1) 支持多协议。
- 2) 轻量级架构，非常容易与已有的系统集成。
- 3) 配置化开发，对业务代码侵入低，开发效率高。
- 4) 与亚马逊的其他基础设施集成，实现 DevOps。

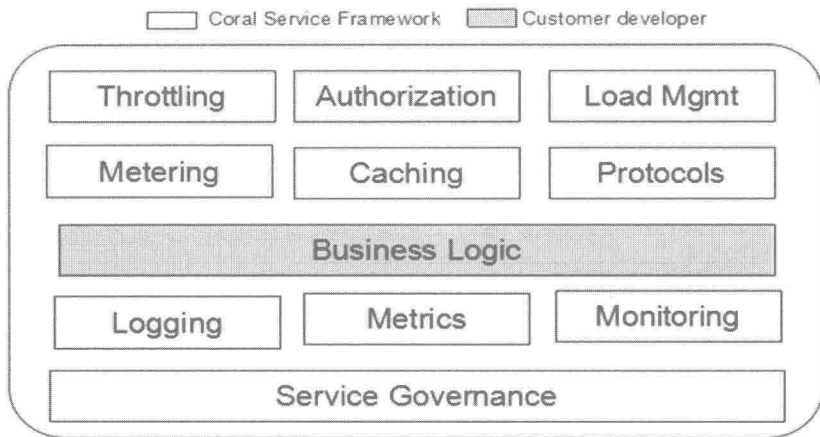


图 2-9 Coral Service Framework

亚马逊内部服务化概况：

- ◎ 全公司统一服务化开发框架，统一运行平台，快速高效服务开发。
- ◎ 所有应用后端完全服务化，由多个微服务组件构成，服务共享、重用。
- ◎ 服务由小团队（2 Pizza Team）负责服务开发、测试、部署、治理，运维整个生命周期支撑。
- ◎ 高度自动化和 DevOps 支持，一键式部署和回滚。

- ◎ 超大规模支持：几十万个服务，成千上万开发者同时使用，平均每秒钟有 1~2 个服务部署。

2.3 分布式服务框架设计

本节我们介绍分布式服务框架的架构原理和概要设计，详细的设计方案及其实现将在后续章节中分专题详细讲解。

2.3.1 架构原理

尽管不同的分布式服务框架实现细节存在差异，但是核心功能差异不大，下面通过图 2-10 所示的逻辑架构图了解分布式服务框架的整体架构。

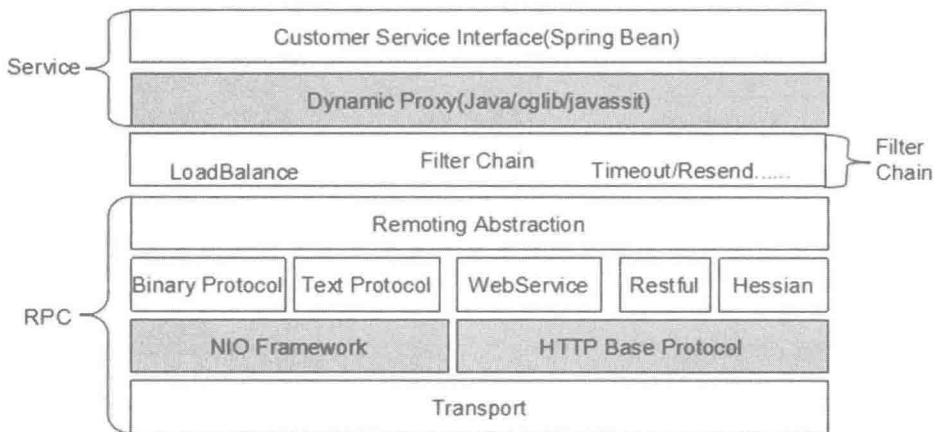


图 2-10 分布式服务框架的逻辑架构图

通常，分布式服务框架的架构可以抽象为三层。

- 1) RPC 层：包括底层通信框架（例如 NIO 框架的封装、公有协议的封装等）、序列化和反序列化框架、用于屏蔽底层通信协议细节和序列化方式差异的 Remoting 框架。

- 2) Filter Chain 层：服务调用职责链，提供多种服务调用切面供框架自身和使用者扩展，例如负载均衡、服务调用性能统计、服务调用完成通知机制、失败重发等。
- 3) Service 层：主要包括 Java 动态代理，消费者使用，主要用于将服务提供者的接口封装成远程服务调用；Java 反射，服务提供者使用，根据消费者请求消息中的接口名、方法名、参数列表反射调用服务提供者的接口本地实现类。再向上就是业务的服务接口定义和实现类，对于使用 Spring 配置化开发的就是 Spring Bean，服务由业务来实现，平台负责将业务接口发布成远程服务。

从功能角度看，分布式服务框架通常会包含另外两个重要功能：服务治理中心和服务注册中心，业务需求不同，具体实现细节也存在很大差异。以服务注册中心为例，HSF 使用的是基于数据库的 ConfigServer，Dubbo 默认使用的是 ZooKeeper。

服务注册中心负责服务的发布和通知，通常支持对等集群部署，某一个服务注册中心宕机并不会导致整个服务注册中心集群不可用。即便整个服务注册中心全部宕机，也只影响新服务的注册和发布，不影响已经发布的服务的访问。

服务治理中心通常包含服务治理接口和服务治理 Portal，架构师、测试人员和系统运维人员通过服务治理 Portal 对服务的运行状态、历史数据、健康度和调用关系等进行可视化的分析和维护，目标就是要持续优化服务，防止服务架构腐化，保证服务高质量运行。

2.3.2 功能特性

尽管不同的分布式服务框架实现细节存在差异，功能特性也不尽相同，但是一些公共能力是必须要具备的，下面我们对分布式服务框架的功能特性进行总结，如表 2-1 所示。

表 2-1 分布式服务框架的功能特性

特 性 名	功 能 名	说 明
服务订阅发布	配置化发布和引用服务	支持通过 XML 配置的方式发布和导入服务，降低对业务代码的侵入
	服务自动发现机制	支持服务实时自动发现，由注册中心推送服务地址，消费者不需要配置服务提供者地址，服务地址透明化
	服务在线注册和去注册	支持运行态注册新服务，也支持运行态取消某个服务的注册

续表

特 性 名	功 能 名	说 明
服务路由	默认提供随机路由、轮循、基于权重的路由策略等	默认提供常用的路由策略，避免每个框架使用者都重复开发
	粘滞连接	总是向同一个提供方发起请求，除非此提供方挂掉，再切换到另一台
	路由定制	支持用户自定义路由策略，扩展平台的功能
集群容错	Failover	失败自动切换，当出现失败，重试其他服务器，通常用于读操作；也可用于幂等性写操作
	Failback	失败自动恢复，后台记录失败请求，定时重发，通常用于消息通知操作
	Failfast	快速失败，只发起一次调用，失败立即报错，通常用于非幂等性的写操作
服务调用	同步调用	消费者发起服务调用之后，同步阻塞等待服务端响应
	异步调用	消费者发起服务调用之后，不阻塞立即返回，由服务端返回应答后异步通知消费者
	并行调用	消费者同时对多个服务提供者批量发起服务调用请求，批量发起请求后，集中等待应答
多协议	私有协议	支持二进制等私有协议，支持私有协议定制和扩展
	公有协议	提供 Web Service 等公有协议，用于外部服务对接
序列化方式	二进制类序列化	支持 Thrift、Protocol buffer 等二进制协议，提升序列化性能
	文本类序列化	支持 JSON、XML 等文本类型的序列化方式，提升通用性和可读性
统一配置	本地静态配置	安装部署修改一次，运行态不修改的配置，可以存放到本地配置文件中
	基于配置中心的动态配置	运行态需要调整的参数，统一放到配置中心（服务注册中心），修改之后统一下发，实时生效

需要指出的是，分布式服务框架并非需要 100%支持表 2-1 中的功能特性，具体问题具体分析。分布式服务框架是为企业业务服务的，每个公司都需要根据自己业务的实际需求来选择优先支持哪些功能，对于不常用或者根本使用不到的，可以裁剪掉。

2.3.3 性能特性

应用服务化之后，由原来的本地 API 调用变成跨进程远程服务调用，网络通信、消息序列化和反序列化、反射调用和动态代理等都会导致性能损耗、时延增加。对于复杂的应用，例如购买商品，后台会涉及 100 多个服务调用。如果服务框架的性能比较差，时延将会被放大数十倍，因此，分布式服务框架的性能指标非常重要，性能特性总结如表 2-2 所示。

表 2-2 分布式服务框架的性能特性

线性特性	说 明
高性能	在同等资源占用情况下，单服务提供者的 TPS 要尽量高
低时延	在同等资源占用情况下，服务调用时延要尽量低
性能线性增长	扩展服务提供者，性能要能够线性增长

2.3.4 可靠性

应用由单机调用演进到分布式部署之后，由于网络故障、服务提供者故障等因素，会导致业务失败率增加，分布式服务框架需要具备很强的可靠性，来保证业务的成功率，可靠性总结如表 2-3 所示。

表 2-3 可靠性功能列表

特 性 名	功 能 名	说 明
服务注册中心	服务健康状态检测	注册中心通过心跳检测服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
	故障切换	注册中心对等集群，任意一台宕掉后，将自动切换到另一台
	高 HA	注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通信
消除单点故障	服务无状态	服务提供者无状态，任意一台宕掉后，不影响使用
	服务集群容错	只要集群中有一台机器的服务提供者可用，业务就不会中断
链路健壮性	心跳检测	链路空闲时没有业务消息，通过定时心跳检测链路是否可用
	断连重连机制	链路断连之后，根据客户端配置的重连策略定时重连，重连成功之前消息不会路由到断连的服务提供者上

2.3.5 服务治理

服务治理是分布式服务框架重要的特性，如表 2-4 所示。

表 2-4 服务治理功能列表

特 性 名	功 能 名	说 明
服务运行态管控	服务路由	业务高峰期，通过动态修改路由策略实现导流
	服务限流	资源成为瓶颈时，服务端和消费者的动态流控
	服务迁入迁出	实现资源的动态分配
	服务降级	服务提供者故障时或者业务高峰期，进行服务强制或者容错降级，执行本地降级逻辑，保证系统平稳运行
	服务超时控制	动态调整超时时间，在业务高峰期保障业务调用成功率
服务监控	性能统计	统计项包括服务调用时延、成功率、调用次数等
	统计报表	提供多维度、实时和历史数据报表，同比、环比等性能对比数据，供运维、运营等使用
	告警	指标异常，根据告警策略发送告警，包括但不限于短信、E-mail、记录日志等
服务生命周期管理	上线审批	服务提供者不能随意线上发布服务，需要通过正规的审批流程批准之后才能上线
	下线通知	服务提供者在下线某个服务之前一段时间，需要根据 SLA 策略，通知消费者
	服务灰度发布	灰度环境划分原则、接口前向兼容性策略，以及消费者如何路由，都需要灰度发布引擎负责管理
故障快速定界定位	分布式日志采集	支持在大规模分布式环境中实时采集容器、中间件和应用的各种日志
	海量日志在线检索	支持分布式环境海量日志的在线检索，支持多维度索引和模糊查询
	调用链可视化展示	通过分布式消息跟踪系统输出调用链，可视化、快速地进行故障定界
	运行日志故障定位	通过调用链的故障关键字，在日志检索界面快速检索故障日志，用于故障的精确定位
服务安全	敏感服务的授权策略	敏感服务如何授权，防止恶意调用
	链路的安全防护	消费者和服务提供者之间的长连接，需要增加安全防护，例如基于 Token 的安全认证机制

2.4 总结

本章我们介绍了分布式服务框架的前世今生，技术是为商业目标服务的，业务发展到什么规模，就有什么样的技术与之匹配。所以，分布式服务框架实际并没有标准，不同的业务场景会有不同的特性需求。

尽管行业、业务规模不同会导致架构存在一定差异，但是差异是相对的，共性是绝对的。通过对阿里的 Dubbo、淘宝的 HSF 和亚马逊的 Coral Service 进行分析，我们惊讶地发现三者存在很多共性，例如配置化发布服务、基于服务注册中心的订阅发布机制、内部私有二进制协议通信、灵活的路由策略、服务治理等。正因为如此，设计与开发通用的分布式服务框架是完全可行的。

第 3 章

通信框架

单机版的本地方法调用变成远程服务调用之后，一个高性能的通用通信框架成为分布式服务框架必不可少的有机组成部分。

通信框架涉及到 Socket 通信、多线程编程、协议栈等相关知识，这部分在 Java 技术堆栈中属于偏难掌握的部分。本章将对通信框架的原理和设计重点进行详细讲解，以期大家可以尽快熟悉通信框架的设计要点并在实际工作中灵活使用。

3.1 关键技术点分析

在设计通信框架之前，我们需要对一些重要的技术点进行选型分析，这样可以帮助我们明确目标和范围，针对目标架构做设计，而不是到处发散无法汇聚。

3.1.1 长连接还是短连接

绝大多数的分布式服务框架（RPC 框架）都推荐使用长连接进行内部通信，为什么选择长连接而不是短连接呢？具体原因总结如下。

- 1) 相比于短连接，长连接更节省资源。如果每发送一条消息就要创建链路、发起握手认证、关闭链路释放资源，会损耗大量的系统资源。长连接只在首次创建时或者链路断连重连才创建链路，链路创建成功之后服务提供者和消费者会通过业务消息和心跳维系链路，实现多消息复用同一个链路节省资源。
- 2) 远程通信是常态，调用时延是关键指标：服务化之后，本地 API 调用变成了远程服务调用，大量本地方法演化成了跨进程通信，网络时延成为关键指标之一。相比于一次简单的服务调用，链路的重建通常耗时更多，这就会导致链路层的时延消耗远远大于服务调用本身的损耗，这对于大型的业务系统而言是无法接受的。

正是因为上述原因，分布式服务框架服务提供者和消费者之间通常采用长连接进行通信。

3.1.2 BIO 还是 NIO

在 JDK 1.4 推出 Java NIO 之前，基于 Java 的所有 Socket 通信都采用了同步阻塞模式（BIO），这种一请求一应答的通信模型简化了上层的应用开发，但是在性能和可靠性方面却存在着巨大的瓶颈。因此，在很长一段时间里，大型的应用服务器都采用 C 或者 C++ 语言开发，因为它们可以直接使用操作系统提供的异步 I/O 或者 AIO 能力。当并发访问量

增大、响应时间延迟增大之后，采用 Java BIO 开发的服务端软件只有通过硬件的不断扩容来满足高并发和低时延，它极大地增加了企业的成本，并且随着集群规模的不断膨胀，系统的可维护性也面临巨大的挑战，只能通过采购性能更高的硬件服务器来解决问题，这会导致恶性循环。

正是由于 Java 传统 BIO 的拙劣表现，才使得 Java 支持非阻塞 I/O 的呼声日渐高涨，最终，JDK1.4 版本提供了新的 NIO 类库，Java 终于也可以支持非阻塞 I/O 了。对于通信框架，采用 BIO 还是 NIO，还是两者都支持，需要在设计之初就做出选择。

在选型之前，我们首先对经典的 BIO 通信模型进行分析，如图 3-1 所示。

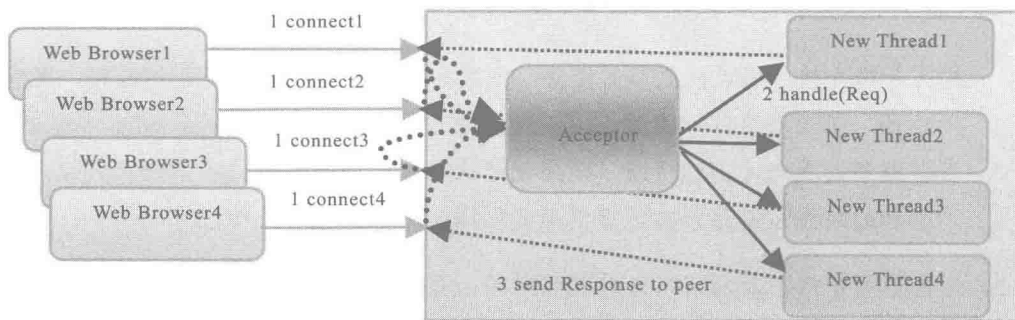


图 3-1 同步阻塞 I/O 服务端通信模型

采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回回答给客户端，线程销毁，这就是典型的一请求一应答通信模型。

该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈 1:1 的正比关系，由于线程是 Java 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

显而易见，如果我们要构建高性能、低时延、支持大并发的应用系统，使用同步阻塞 I/O 模型是无法满足性能线性增长和可靠性的。

在 I/O 编程过程中,当需要同时处理多个客户端接入请求时,可以利用多线程或者 I/O 多路复用技术进行处理。I/O 多路复用技术通过把多个 I/O 的阻塞复用到同一个 select 的阻塞上,从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型相比, I/O 多路复用的最大优势是系统开销小,系统不需要创建新的额外进程或者线程,也不需要维护这些进程和线程的运行,降低了系统的维护工作量,节省了系统资源。

JDK 1.4 提供了对非阻塞 I/O (NIO) 的支持, JDK1.5_update10 版本使用 epoll 替代了传统的 select/poll,极大地提升了 NIO 通信的性能。

与 Socket 类和 ServerSocket 类相对应, NIO 也提供了 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单,但是性能和可靠性都不好,非阻塞模式则正好相反。开发人员一般可以根据自己的需要来选择合适的模式,一般来说,低负载、低并发的应用程序可以选择同步阻塞 I/O 以降低编程复杂度,但是对于高负载、高并发的网络应用,需要使用 NIO 的非阻塞模式进行开发。

NIO 采用多路复用技术,一个多路复用器 Selector 可以同时轮询多个 Channel,由于 JDK 使用了 epoll()代替传统的 select 实现,所以它并没有最大连接句柄 1024/2048 的限制。这也就意味着只需要一个线程负责 Selector 的轮询,就可以接入成千上万的客户端,这确实是个非常巨大的进步。

采用多路复用器 Selector 实现的 Reactor 通信模型如图 3-2 所示。

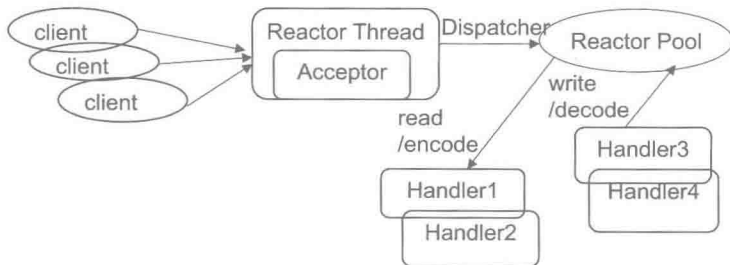


图 3-2 Reactor 线程模型

通过对两种 I/O 模型的对比，显然在通信框架中应该使用更高效的 NIO 非阻塞 I/O。

3.1.3 自研还是选择开源 NIO 框架

尽管 JDK 提供了丰富的 NIO 类库，网上也有很多 NIO 学习例程，但是直接使用 Java NIO 类库想要开发出稳定可靠的通信框架却并非易事，原因如下。

- 1) NIO 的类库和 API 繁杂，使用麻烦，你需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等。
- 2) 需要具备其他的额外技能做铺垫，例如熟悉 Java 多线程编程。这是因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的 NIO 程序。
- 3) 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。
- 4) JDK NIO 的 BUG，例如臭名昭著的 epoll bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK1.6 版本的 update18 修复了该问题，但是直到 JDK1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有被根本解决。该 BUG 以及与该 BUG 相关的问题单可以参见以下链接内容：

- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933
- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

随着开源 NIO 框架的发展，目前越来越多的商用系统采取直接集成开源 NIO 框架的方式代替之前的自研方案。以最成熟的 NIO 框架 Netty 为例，它已经得到成百上千的商用项目验证。例如 Hadoop 的 RPC 框架 avro 使用 Netty 作为底层通信框架、实时流式计算框架 Storm 底层通信框架也采用的是 Netty，还有 Twitter 内部使用的 RPC 框架 Finagle，其底层通信框架也基于 Netty 构建。

Netty 的优势总结如下：

- ◎ API 使用简单，开发门槛低。
- ◎ 功能强大，预置了多种编解码功能，支持多种主流协议。
- ◎ 定制能力强，可以通过 `ChannelHandler` 对通信框架进行灵活地扩展。
- ◎ 性能高，通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优。
- ◎ 成熟、稳定，Netty 修复了已经发现的所有 JDK NIO BUG，业务开发人员不需要再为 NIO 的 BUG 而烦恼。
- ◎ 社区活跃，版本迭代周期短，发现的 BUG 可以被及时修复。同时，更多的新功能会加入。
- ◎ 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它已经完全能够满足不同行业的商业应用了。

Netty 的表现如此优秀，因此我们的通信框架基于 Netty 进行设计和开发。

3.2 功能设计

分布式服务框架的底层通信框架首先是一个通用的通信框架，它不应该与具体的协议绑定。基于通信框架之上，可以构建私有协议栈和公有协议栈。

它的架构原理如图 3-3 所示。

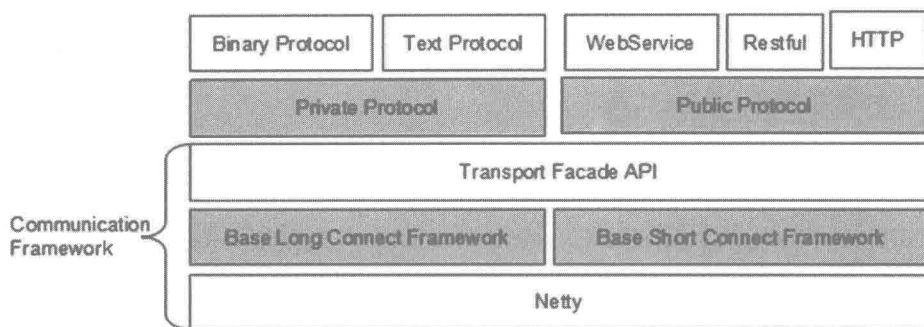


图 3-3 通信框架架构图

3.2.1 服务端设计

通信框架服务端的职责如下：

- 1) 提供上层 API（屏蔽底层 NIO 框架），用于初始化服务端实例，设置服务端通信相关参数，包括服务端的 I/O 线程池（线程组参数）、监听地址、TCP 相关参数、接收和发送缓冲区大小等。
- 2) 提供可扩展的编解码插件，用户可以通过扩展的方式实现自定义协议的编码和解码。
- 3) 提供拦截面，用于私有协议栈开发。例如通过新增鉴权插件实现服务端对客户端的安全认证。

服务端最重要的设计原则有三条：

- 1) 服务端只提供上层的 API，不与任何具体协议绑定。
- 2) 服务端提供给用户的 API 要尽量屏蔽底层的通信细节，防止底层的变更引起上层级联变更。例如 Netty3 升级到 Netty4，整个包路径都发生了改变，如果在底层屏蔽了 Netty 接口，上层不受影响。
- 3) 服务端功能上不要求全，重点在可扩展性上。

利用 Netty 提供的 NIO 服务端类库，可以非常轻松地实现通信框架服务端的开发。下面我们对 Netty 服务端的使用进行讲解：Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和用户交互的边界做了封装，目的就是减少用户开发工作量，降低开发难度。ServerBootstrap 是 Socket 服务端的启动辅助类，用户通过 ServerBootstrap 可以方便地创建 Netty 的服务端，创建代码示例如下：

代码清单 3-1 基于 Netty 的通信服务端创建

```
// 配置服务端的 NIO 线程组
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
    .option(ChannelOption.SO_BACKLOG, 100)
    .handler(new LoggingHandler(LogLevel.INFO))
    .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch)
            throws IOException {
            ch.pipeline().addLast(
                new NettyMessageDecoder(1024 * 1024, 4, 4));
            ch.pipeline().addLast(new NettyMessageEncoder());

            ch.pipeline().addLast("readTimeoutHandler",
                new ReadTimeoutHandler(50));
            ch.pipeline().addLast(new LoginAuthRespHandler());
            ch.pipeline().addLast("HeartBeatHandler",
                new HeartBeatRespHandler());
        }
    });
// 绑定端口，同步等待成功
b.bind(NettyConstant.REMOTEIP, NettyConstant.PORT).sync();
}
```

基于 Netty 开发通信框架服务端，有以下几个关键点需要掌握。

- 1) 用于接收客户端连接的线程池：通常被称为 bossGroup，它的构造方法与处理 I/O 读写的线程池相同（workerGroup），都是通过 new NioEventLoopGroup 创建。bossGroup 的线程数建议设置为 1，因为它仅负责接收客户端的连接，不做复杂的逻辑处理，为了尽可能少地占用资源，它的取值越小越好。
- 2) TCP 参数设置：建议在 API 层面开放 TCP 参数设置，为一些特殊的私有协议预留扩展点，对于大多数的开发者，使用默认值即可。Netty 的 ChannelOption 类提供了对 TCP 参数的封装，由于是工具常量类，未来发生变更的可能性很小，因此可以直接开放给上层使用。
- 3) 编解码框架的定制：通信框架封装 Netty 的 MessageToByteEncoder 和 LengthFieldBasedFrameDecoder，提供统一、通用的编解码接口或者抽象类，由用户实现编解码接口，实现编解码的灵活定制。
- 4) 通信层业务逻辑的定制：利用 Netty 的 ChannelPipeline，将 ChannelHandler 的链式编排能力以参数或者配置化的方式开放出来，由用户灵活扩展，实现协议层逻辑定制。例如示例代码中的超时检测、握手认证等。需要指出的是，如果担心底层通信框架 API 发生变更直接影响用户的代码，可以在 Netty 的 ChannelHandler 上面再抽象包装一层，通过 Facade 模式屏蔽底层的实现细节。

3.2.2 客户端设计

相比于服务端，客户端的创建更复杂一些。需要考虑网络连接超时、连接失败等异常场景，下面我们首先对基于 Netty Client 的客户端创建流程进行分析，如图 3-4 所示。

下面对客户端的创建流程进行详细介绍。

第一步，创建 Bootstrap 实例，Bootstrap 是 Socket 客户端创建工具类，用户通过 Bootstrap 可以方便地创建 Netty 的客户端并发起异步 TCP 连接操作。

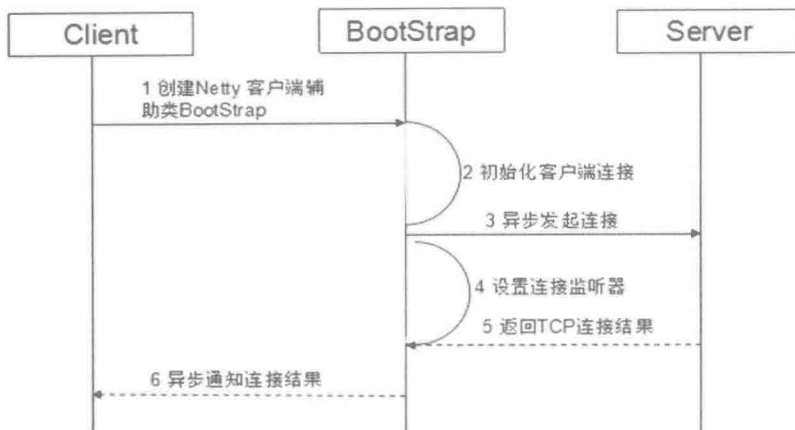


图 3-4 客户端创建流程图

第二步，初始化 TCP 连接参数，设置编解码 Handler 和其他业务 Handler，代码示例如下：

```

b.group(group).channel(NioSocketChannel.class)
    .option(ChannelOption.TCP_NODELAY, true)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch)
            throws Exception {
            ch.pipeline().addLast(
                new NettyMessageDecoder(1024 * 1024, 4, 4));
            ch.pipeline().addLast("MessageEncoder",
                new NettyMessageEncoder());
            ch.pipeline().addLast("readTimeoutHandler",
                new ReadTimeoutHandler(50));
            ch.pipeline().addLast("LoginAuthHandler",
                new LoginAuthReqHandler());
            ch.pipeline().addLast("HeartBeatHandler",
                new HeartBeatReqHandler());
        }
    })
  
```

```

    });
}

```

第三步，调用 Bootstrap 的 connect 方法异步发起连接，代码如下所示：

```

b.group(group).channel(NioSocketChannel.class)
    .connect(
        new InetSocketAddress(host, port),
        new InetSocketAddress(NettyConstant.LOCALIP,
            NettyConstant.LOCAL));

```

需要指出的是，connect 是异步连接，程序并不会在此等待。底层 TCP 是否连接成功，不得而知。为了让用户后续可以判断连接操作结果，异步连接返回了 ChannelFuture 对象，用于通知连接结果。连接结果有以下两种判断方法。

- ◎ 同步等待连接操作结果：用户线程将在此 wait()，直到连接操作完成之后，线程被 notify()，用户代码继续执行，相关的接口说明如图 3-5 所示。

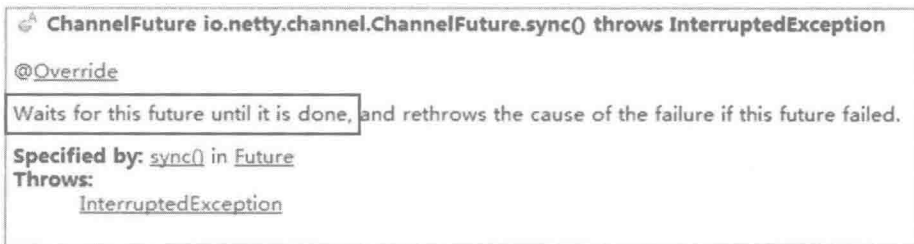


图 3-5 同步等待异步连接操作结果

- ◎ 注册监听器，等待操作完成之后的异步通知：可以新增一个或者多个监听器，用于监听异步连接操作结果，操作完成之后系统会回调监听器的相关方法，接口说明如图 3-6 所示。

第四步，采用设置连接监听器的方式，用于连接结果异步通知，通知接口如图 3-7 所示。

第五步，服务端返回 TCP 握手应答，系统回调监听器操作完成接口。

第六步，在操作完成接口中实现相关业务逻辑，通知客户端连接操作完成。



图 3-6 注册监听器用于异步通知

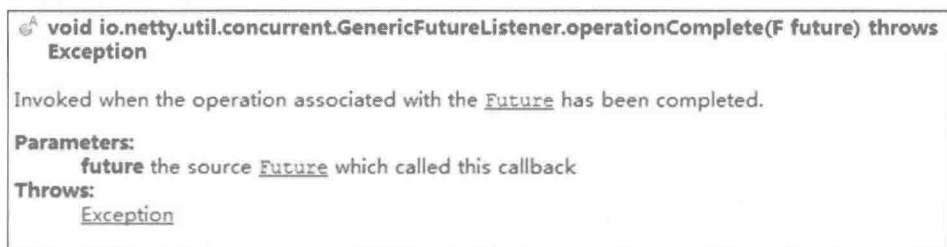


图 3-7 异步连接通知接口

上述步骤只描述了连接正常的场景，对于连接失败、连接超时、断连重连等并没有详细描述，在实际的商业项目开发中，必须要对上述异常场景进行处理。

3.3 可靠性设计

在客户端和服务端进行网络通信时，网络闪断、网络超时、通信对端宕机等故障时有发生，为了保证异常场景下系统的可用性，通信框架必须具备很高的可靠性，对于大多数电信软件，需要能够支持 5 个 9 的高可靠性。

本节将对通信框架常用的可靠性设计要点进行总结，使读者能够掌握通信框架可靠性设计的关键技术。

3.3.1 链路有效性检测

当网络发生单通、连接被防火墙 Hang 住、长时间 GC 或者通信线程发生非预期异常时，会导致链路不可用且不易被及时发现。特别是异常发生在凌晨业务低谷期间，当早晨业务高峰期到来时，由于链路不可用会导致瞬间的大批量业务失败或者超时，这将对系统的可靠性产生重大的威胁。

从技术层面看，要解决链路的可靠性问题，必须周期性地对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。

心跳检测机制分为三个层面：

- 1) TCP 层面的心跳检测，即 TCP 的 Keep-Alive 机制，它的作用域是整个 TCP 协议栈。
- 2) 协议层的心跳检测，主要存在于长连接协议中，例如 SMPP 协议。
- 3) 应用层的心跳检测，它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路可用，对方活着并且能够正常接收和发送消息。作为高可靠的 NIO 框架，Netty 也提供了心跳检测机制，下面我们一起熟悉下心跳的检测原理，如图 3-8 所示。

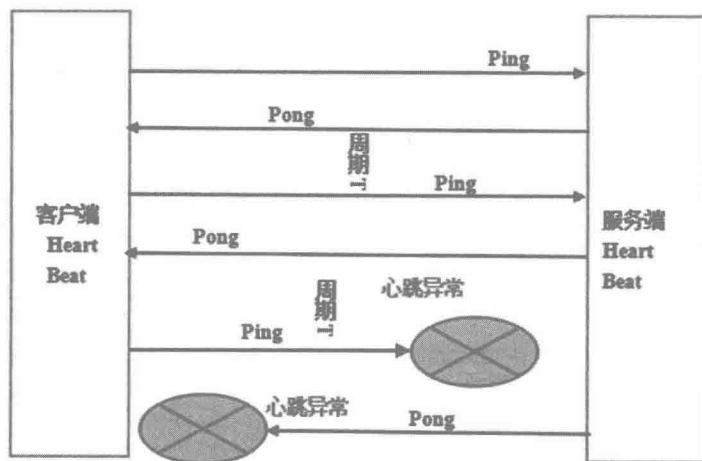


图 3-8 链路心跳检测

不同的协议，心跳检测机制也存在差异，归纳起来主要分为两类。

- ◎ Ping-Pong 型心跳：由通信一方定时发送 Ping 消息，对方接收到 Ping 消息之后，立即返回 Pong 应答消息给对方，属于请求-响应型心跳。
- ◎ Ping-Ping 型心跳：不区分心跳请求和应答，由通信双方按照约定定时向对方发送心跳 Ping 消息，它属于双向心跳。

心跳检测策略如下：

- ◎ 连续 N 次心跳检测都没有收到对方的 Pong 应答消息或者 Ping 请求消息，则认为链路已经发生逻辑失效，这被称作心跳超时。
- ◎ 读取和发送心跳消息的时候如何直接发生了 I/O 异常，说明链路已经失效，这被称为心跳失败。

无论发生心跳超时还是心跳失败，都需要关闭链路，由客户端发起重连操作，保证链路能够恢复正常。

Netty 的心跳检测实际上是利用了链路空闲检测机制实现的，它的空闲检测机制分为三种：

- ◎ 读空闲，链路持续时间 t 没有读取到任何消息。
- ◎ 写空闲，链路持续时间 t 没有发送任何消息。
- ◎ 读写空闲，链路持续时间 t 没有接收或者发送任何消息。

Netty 的默认读写空闲机制是发生超时异常，关闭连接，但是，我们可以定制它的超时实现机制，以便支持不同的用户场景，链路空闲接口定义如下：

```
protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt)
throws Exception {
    ctx.fireUserEventTriggered(evt);
}
```

链路空闲的时候并没有关闭链路，而是触发 `IdleStateEvent` 事件，用户订阅 `IdleStateEvent` 事件，用于自定义逻辑处理，例如关闭链路、客户端发起重新连接、告警和打印日志等。利用 Netty 提供的链路空闲检测机制，可以非常灵活地实现链路空闲时的有效性检测。

3.3.2 断连重连机制

当发生如下异常时，客户端需要释放资源，重新发起连接：

- 1) 服务端因为某种原因，主动关闭连接，客户端检测到链路被正常关闭。
- 2) 服务端因为宕机等故障，强制关闭连接，客户端检测到链路被 Rest 掉。
- 3) 心跳检测超时，客户端主动关闭连接。
- 4) 客户端因为其他原因（例如解码失败），强制关闭连接。
- 5) 网络类故障，例如网络丢包、超时、单通等，导致链路中断。

客户端检测到链路中断后，等待 `INTERVAL` 时间，由客户端发起重连操作，如果重连失败，间隔周期 `INTERVAL` 后再次发起重连，直到重连成功。

为了保证服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待 `INTERVAL` 时间之后再发起重连，而不是失败后就立即重连。

为了保证句柄资源能够及时释放，无论什么场景下的重连失败，客户端都必须保证自身的资源被及时释放，包括但不限于 `SocketChannel`、`Socket` 等。重连失败后，需要打印异常堆栈信息，方便后续的问题定位。

利用 Netty Channel 提供的 `CloseFuture`，可以非常方便地检测链路状态，一旦链路关闭，相关事件即被触发，可以重新发起连接操作，代码示例如下：

```
future.channel().closeFuture().sync();  
    } finally {
```



```

// 所有资源释放完成之后，清空资源，再次发起重连操作
executor.execute(new Runnable() {
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(3); // 3秒之后发起重连，等待句柄释放
        } catch (Exception e) {
            // 发起重连操作
            connect(NettyConstant.PORT, NettyConstant.REMOTEIP);
            // 异常处理相关代码省略
        }
    }
});

```

3.3.3 消息缓存重发

当我们调用消息发送接口的时候，消息并没有真正被写入到 Socket 中，而是先放入 NIO 通信框架的消息发送队列中，由 Reactor 线程扫描待发送的消息队列，异步地发送给通信对端。假如很不幸，消息队列中积压了部分消息，此时链路中断，这会导致部分消息并没有真正发送给通信对端，示例如图 3-9 所示。

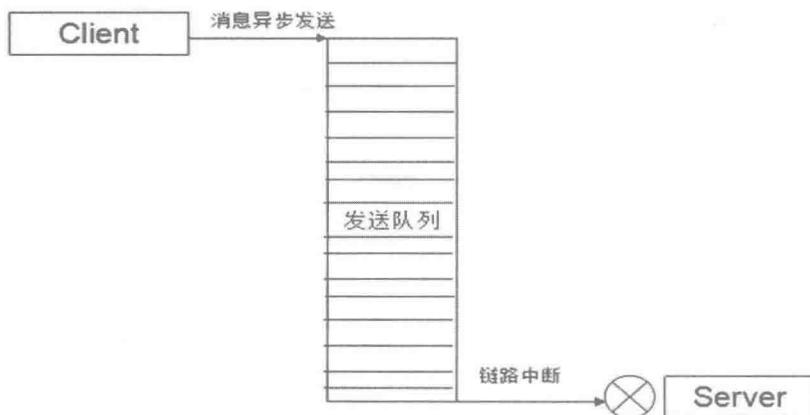


图 3-9 链路中断导致积压消息没有发送

发生此故障时，我们希望 NIO 框架能够自动实现消息缓存和重新发送，遗憾的是作为基础的 NIO 通信框架，无论是 Mina 还是 Netty，都没有提供该功能，需要通信框架自己封装实现，基于 Netty 的实现策略如下：

- 1) 调用 Netty ChannelHandlerContext 的 write()方法时，返回 ChannelFuture 对象，我们在 ChannelFuture 中注册发送结果监听 Listener。
- 2) 在 Listener 的 operationComplete 方法中判断操作结果，如果操作不成功，将之前发送的消息对象添加到重发队列中。
- 3) 链路重连成功之后，根据策略，将缓存队列中的消息重新发送给通信对端。

需要指出的是，并非所有场景都需要通信框架做重发，例如服务框架的客户端，如果某个服务提供者不可用，会自动切换到下一个可用的服务提供者之上。假定是链路中断导致的服务提供者不可用，即便链路重新恢复，也没有必要将之前积压的消息重新发送，因为消息已经通过 FailOver 机制切换到另一个服务提供者处理。所以，消息缓存重发只是一种策略，通信框架应该支持链路级重发策略。

3.3.4 资源优雅释放

Java 的优雅停机通常通过注册 JDK 的 ShutdownHook 来实现，当系统接收到退出指令后，首先标记系统处于退出状态，不再接收新的消息，然后将积压的消息处理完，最后调用资源回收接口将资源销毁，最后各线程退出执行。

通常优雅退出有个时间限制，例如 30s，如果到达执行时间仍然没有完成退出前的操作，则由监控脚本直接 kill -9 pid，强制退出。

通信框架是分布式服务框架的有机组成部分，也需要支持优雅停机。Netty 提供了完善的优雅停机接口，通过调用相关接口，可以实现线程池、消息队列、Socket 句柄、多路复用器等的资源释放。

Netty 优雅退出的相关接口定义如图 3-10 所示。

⚡ **Future<?> io.netty.util.concurrent.EventExecutorGroup.shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit)**

Signals this executor that the caller wants the executor to be shut down. Once this method is called, isShuttingDown() starts to return true, and the executor prepares to shut itself down. Unlike shutdown(), graceful shutdown ensures that no tasks are submitted for *'the quiet period'* (usually a couple seconds) before it shuts itself down. If a task is submitted during the quiet period, it is guaranteed to be accepted and the quiet period will start over.

Parameters:

- quietPeriod** the quiet period as described in the documentation
- timeout** the maximum amount of time to wait until the executor is shutdown() regardless if a task was submitted during the quiet period
- unit** the unit of quietPeriod and timeout

Returns:

the terminationFuture()

图 3-10 优雅退出接口定义

目前 Netty 向用户提供的主要接口和类库都提供了资源销毁和优雅退出的接口，用户的自定义实现类可以继承这些接口，完成用户资源的释放和优雅退出。

3.4 性能设计

分布式服务框架被广泛应用于大数据处理、互联网消息中间件、游戏和金融行业等。大多数应用场景对底层的通信框架都有很高的性能要求，通信框架在设计之初就必须要考虑如何实现高性能。

3.4.1 性能差的三宗罪

“罪一”：网络传输方式问题。传统的 RPC 框架或者基于 RMI 等方式的远程服务（过程）调用采用了同步阻塞 I/O，当客户端的并发压力或者网络时延增大之后，同步阻塞 I/O 会由于频繁地 wait 导致 I/O 线程经常性的阻塞，由于线程无法高效工作，I/O 处理能力自然下降。

采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，接收到客户端连接之后，为其创建一个新的线程处理请求消息，处理完成之后，返回应答消息给客户端，线程销毁，这就是典型的一请求一应答模型。该架构最大的问题就是不具备弹性伸缩能力，当并发访问量增加后，服务端的线程个数和并发访问数成线性正比，由于线程是 Java 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能急剧下降，随着并发量的继续增加，可能会发生句柄溢出、线程堆栈溢出等问题，并导致服务器最终宕机。

“罪二”：序列化性能差。Java 序列化存在如下几个典型问题。

- ◎ Java 序列化机制是 Java 内部的一种对象编解码技术，无法跨语言使用。例如对于异构系统之间的对接，Java 序列化后的码流需要能够通过其他语言反序列化成原始对象（副本），目前很难支持。
- ◎ 相比于其他开源的序列化框架，Java 序列化后的码流太大，无论是网络传输还是持久化到磁盘，都会导致额外的资源占用。
- ◎ 序列化性能差，资源占用率高（主要是 CPU 资源占用高）。

“罪三”：线程模型问题。由于采用同步阻塞 I/O，这会导致每个 TCP 连接都占用 1 个线程，由于线程资源是 JVM 虚拟机非常宝贵的资源，当 I/O 读写阻塞导致线程无法及时释放时，会导致系统性能急剧下降，严重的甚至会导致虚拟机无法创建新的线程。

3.4.2 通信性能三原则

尽管影响 I/O 通信性能的因素非常多，但是从架构层面看主要有三个要素。

- 1) 传输：用什么样的通道将数据发送给对方。可以选择 BIO、NIO 或者 AIO，I/O 模型在很大程度上决定了通信的性能。
- 2) 协议：采用什么样的通信协议，HTTP 等公有协议或者内部私有协议。协议的选择不同，性能也不同。相比于公有协议，内部私有协议的性能通常可以被设计得更优。

- 3) 线程：数据报如何读取？读取之后的编解码在哪个线程进行？编解码后的消息如何派发？Reactor 线程模型的不同，对性能的影响也非常大。

3.4.3 高性能之道

利用 Netty 可以非常便捷地构建高性能的通信框架，下面我们对 Netty 支持高性能通信的架构特性进行总结。

- 1) 异步非阻塞通信：Netty 的 I/O 线程 `NioEventLoop` 由于聚合了多路复用器 `Selector`，可以同时并发处理成百上千个客户端 `SocketChannel`。由于读写操作都是非阻塞的，这就可以充分提升 I/O 线程的运行效率，避免由频繁的 I/O 阻塞导致的线程挂起。另外，由于 Netty 采用了异步通信模式，一个 I/O 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 I/O 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。
- 2) 高效的 I/O 线程模型：Netty 支持 Reactor 单线程模型、Reactor 多线程模型和主从 Reactor 多线程模型，可以满足不同的容量和性能需求。
- 3) 高性能的序列化框架：不同的应用场景对序列化框架的需求也不同，对于高性能应用场景，Netty 默认提供了 Google 的 `Protobuf` 二进制序列化框架，如果用户对其他二进制序列化框架有需求，也可以基于 Netty 提供的编解码框架扩展实现。

除了上述总结的三点之外，Netty 还支持零拷贝、内存池等其他性能相关的特性，读者可以根据实际项目需要决定是否在工作中使用它们。

3.5 最佳实践

大规模分布式系统下性能的线性增长和可靠性对架构是个很大的挑战，通信链路由单条膨胀到成千上万，内存增长、线程争用等问题变得越来越突出，稍有不慎就会发生内存

溢出、线程创建失败、句柄耗尽等问题，严重影响系统的稳定性。

最重要的是需要解决通信线程合理利用问题，在功能设计阶段我们演示了如何通过 Netty 的 `ServerBootstrap` 和 `Bootstrap` 创建通信框架的服务端和客户端，这块儿的 NIO 线程池使用很有讲究，下面我们详细讲解下。

误区 1：不指定线程池线程大小。

代码如下：

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
    .....后续代码省略
```

如果不指定线程池大小，Netty 会优先使用 `-Dio.netty.eventLoopThreads` 指定的值，如果用户没有设置，默认采用 `CPU Core * 2 (Runtime.getRuntime().availableProcessors() * 2)`。通信框架的服务端只负责客户端的接入，不处理 I/O 读写操作，负载非常轻，因此只需要单线程即可满足客户端接入，因此代码可以优化为：

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup(N);
try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
    .....后续代码省略
```

对于 I/O 读写线程池 `workerGroup`，线程个数依旧需要根据实际情况进行设计，如果并发接入的客户端不是很多，通信框架的负载不重，建议配置值为： $1 \leq N \leq \text{CPU Core}$ 。如果并发接入数非常多，I/O 处理逻辑又比较复杂，则建议根据实际性能测试结果逐步调大线程数。总而言之，尽量不要使用系统默认值。

误区 2: I/O 线程池使用不当, 导致通信线程膨胀。

在大规模分布式系统中, 客户端和服务端的长连接链路成百上千, 这就意味着某个消费者需要连接海量的服务提供者, 如果设计不当, 会导致 I/O 线程急剧膨胀, 严重影响系统性能和稳定性, 示例代码如下:

```
for(InetSocketAddress addr : serverAddrList)
{
    Bootstrap b = new Bootstrap();
    EventLoopGroup group = new NioEventLoopGroup();
    b.group(group).channel(NioSocketChannel.class)
    .option(ChannelOption.TCP_NODELAY, true)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch)
            throws Exception {
            .....此处代码省略
        }
    });
    // 发起异步连接操作
    ChannelFuture future = b.connect(
        new InetSocketAddress(host, port),
        new InetSocketAddress(NettyConstant.LOCALIP,
NettyConstant.LOCAL_PORT)).sync();
    .....后续代码省略
}
```

如果有 1000 个服务提供者, 意味着 `NioEventLoopGroup` 被创建了 1000 个, 假设服务器配置是 8C, 则单个 `NioEventLoopGroup` 会创建 16 个 I/O 线程, 1000 个连接, I/O 线程数为 16000 个, 这个太可怕了。更加麻烦的是线下 UT 只测试功能, 不会发现这个问题。测试部的集成测试组网规模往往比线上少, 另外测试人员也不会关注到进程线程数这个级别, 所以问题很有可能被遗漏到线上, 当某一天开发者被从睡眠中喊醒的时候, 方知晚矣。

解决上述问题通常有两种策略:

- 1) 根据客户端连接数, 评估 I/O 线程数, 创建一个大的线程池 `NioEventLoopGroup`, 所有客户端连接共用。
- 2) 创建一个包含 `NioEventLoopGroup` 的数组, 将客户端连接按照 Hash 算法分组, 将所有连接均匀的打散在 `NioEventLoopGroup` 中, 该方案的优点是降低锁竞争, 提升处理效率。

3.6 总结

本章我们对通信框架的设计要点进行了总结和分析, 除了基础功能, 通信框架的可靠性和性能也是非常重要的指标。

高性能通信框架的构建成功, 为分布式服务框架后续的设计打下了坚实的基础, 也为上层的应用层协议栈开发提供了良好的基础设施。

序列化与反序列化

服务提供者和消费者通过网络进行通信，对象需要进行序列化和反序列化，常见的序列化和反序列化方式很多，如何选择是重点也是难点。

4.1 几个关键概念澄清

通常我们也习惯将序列化（Serialization）称为编码（Encode），它将对象序列化为字节数组，用于网络传输、数据持久化或者其他用途。

反之，反序列化（Deserialization）/解码（Decode）把从网络、磁盘等读取的字节数组还原成原始对象（通常是原始对象的副本），以方便后续的业务逻辑操作。

进行远程跨进程服务调用时（例如 RPC 调用），需要使用特定的序列化技术，对需要进行网络传输的对象做编码或者解码，以便完成远程调用。

4.1.1 序列化与通信框架的关系

序列化与通信框架不是强耦合的关系，通信框架提供的编解码框架可以非常方便地支持用户通过扩展实现自定义的序列化格式。用户也可以在应用程序及其他位置实现对象的序列化和反序列化，通信框架的编解码接口作为可选插件，并不强制用户一定要在通信框架内部实现消息的序列化和反序列化。

4.1.2 序列化与通信协议的关系

序列化与通信协议是解耦的，同一种通信协议可能由多种序列化方式承载，同一种序列化方式也可以用在不同协议里。

以 HTTP 协议为例，承载消息体的可以是 XML、JSON 等文本类的协议，也可以是图片附件等二进制流媒体协议。

在设计分布式服务框架时，序列化和反序列化是一个独立的接口和插件，它可以被多种协议重用、替换和扩展，以实现服务框架序列化方式的多样化，满足不同业务领域的用户需求。

4.1.3 是否需要支持多种序列化方式

整体而言，序列化可以分为文本类和二进制类两种，不同的业务场景需求也不同，分布式服务框架面向的领域是多样化的，因此它的序列化/反序列化框架需要具备如下特性：

- ◎ 默认支持多种常用的序列化/反序列化方式，文本类例如 XML/JSON 等，二进制的如 PB（Protocol Buffer）/Thrift 等。
- ◎ 序列化框架可扩展，用户可以非常灵活、方便地扩展其他序列化方式。

4.2 功能设计

在设计序列化/反序列化框架的时候，我们需要从功能、跨语言支持、兼容性、性能等多个角度进行综合考量。

4.2.1 功能丰富度

考察序列化框架功能的一个重要指标就是它支持的数据结构种类，有些序列化框架对数据结构的支持不完善，某些类型不支持。例如 fastjson 对泛型的支持不友好，MessagePack 的早期版本不支持 Map、List 等。

原则上，支持的数据结构种类越丰富越好，毕竟分布式服务框架面向的是不同业务领域，需要兼顾各种业务场景。

另外一个问题就是序列化/反序列化接口是否友好、简洁，例如 MessagePack 的序列化接口就非常简洁好用：

```
public class MsgpackEncoder extends MessageToByteEncoder<Object> {
    @Override
    protected void encode(ChannelHandlerContext arg0, Object arg1, ByteBuf
arg2) throws Exception {
```

```
MessagePack msgpack = new MessagePack();  
byte[] raw = msgpack.write(arg1);  
arg2.writeBytes(raw);  
}  
}
```

通过序列化代码可以看出，MessagePack 只需要两行代码即可完成对象的序列化，而且不需要定义中间态的文件做静态编译，使用起来非常便捷。

反观 PB (Protocol Buffer)，需要定义中间态的 .proto 文件，根据该文件生成需要序列化的对象数据结构定义，然后编写 writer 和 reader 方法，完成序列化和反序列化。如果没有 IDE 工具辅助开发，使用起来还是不太方便。

4.2.2 跨语言支持

相信大多数 Java 程序员接触到的第一种序列化或者编解码技术就是 Java 默认提供的序列化机制，需要序列化的 Java 对象只需要实现 java.io.Serializable 接口并生成序列化 ID，这个类就能够通过 java.io.ObjectInput 和 java.io.ObjectOutput 序列化和反序列化。

由于使用简单，开发门槛低，Java 序列化得到了广泛的应用，但是由于它自身存在很多缺点，我们的分布式服务框架不会选择它作为基础特性提供给用户。它的最大缺点就是无法支持跨语言。

对于跨进程的服务调用，服务提供者可能会使用 C++ 或者其他语言开发，当我们需要和异构语言进程交互时，Java 序列化就难以胜任。由于 Java 序列化技术是 Java 语言内部的私有协议，其他语言并不支持，对于用户来说它完全是黑盒。Java 序列化后的字节数组，别的语言无法进行反序列化，双方无法互通。

反观 MessagePack、PB、Thrift、Avro 等都支持多种语言，以 MessagePack 为例，它支持 Java、Python、Ruby、Haskell、C#、OCaml、Lua、Go、C、C++ 等。

衡量序列化框架通用性的一个重要指标就是对多语言的支持，因为数据交换的双方很难保证一定采用相同的语言开发，如果序列化框架和某种语言绑定，它就很难跨语言。分布式服务框架根据业务特点不同，可能会采用不同的语言实现。使用同一种服务框架的不同语言开发的服务之间要能够互通，序列化和反序列化首先要能够支持互通，这就是支持跨语言的重要所在。

4.2.3 兼容性

服务上线之后，由于业务变更、Bug 修复等很快发现需要进行版本升级，版本升级就会涉及到前向兼容问题。尽管通过服务的灰度发布可以实现多个服务版本共存，但是它并不能解决所有的问题。

服务的前向兼容通常涉及以三个层面。

- 1) 接口的兼容：包括接口定义、接口的输入参数和返回值。
- 2) 业务的逻辑兼容：接口能够兼容是前提，内部实现的业务逻辑也需要能够前向兼容；例如根据新增的字段判断新老流程，老的业务消息没有携带新扩展的字段就走老的流程，新的业务则走新流程。
- 3) 数据兼容性：包括但不限于关系型数据库、非关系型数据库、表、索引等。

本节我们主要讨论接口兼容性，一个好的序列化框架应该支持数据结构的前向兼容性，例如新增字段、删除字段、调整字段顺序等。

服务化之后的业务特点就是服务独立打包、独立部署和升级，为了保证业务能够快速敏捷交付，服务框架需要保证服务接口的前向兼容性。我们在做序列化框架技术选型时，需要将此指标纳入到考虑范围中。

以 PB 为例，它有一个非常棒的特性，即“前向”兼容性好，不必破坏已部署的、依靠“老”数据格式的程序就可以对数据结构进行升级。这样服务升级时就可以不必担心因为消息结构的改变而造成的大规模的代码重构或者迁移的问题，添加新的消息中的字段并

不会引起已经发布的程序的任何改变。

4.2.4 性能

序列化/反序列化的性能主要有三个指标：

- 1) 序列化之后的码流大小。
- 2) 序列化/反序列化的速度。
- 3) 资源占用，主要是 CPU 和堆内存。

下面我们看下 Protobuf 等序列化框架的性能对比数据，可以作为技术选型依据如图 4-1 和图 4-2 所示。

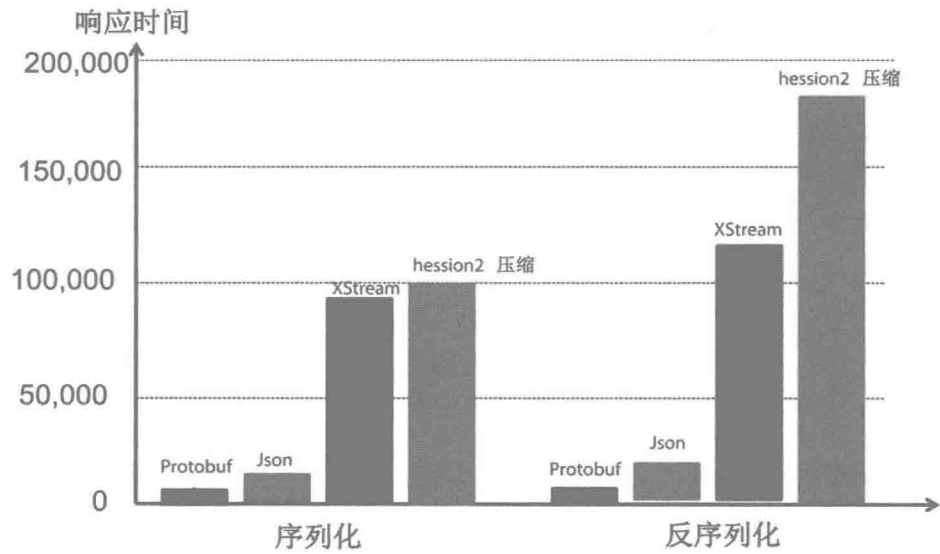


图 4-1 序列化和反序列化速度对比

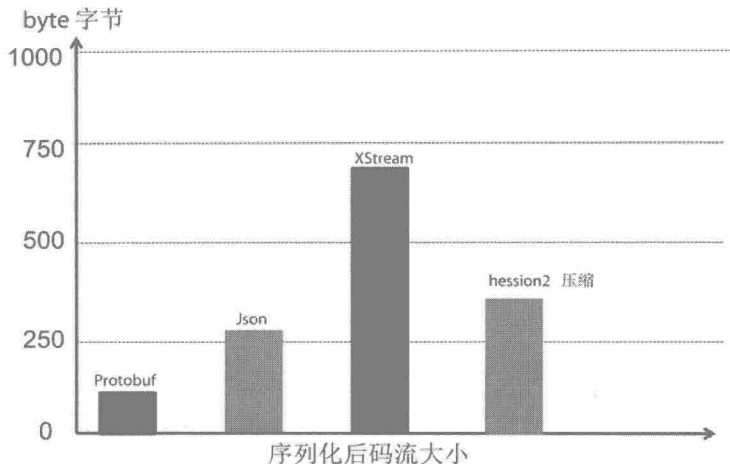


图 4-2 序列化之后码流大小对比

通过测试对比，我们发现 Protobuf 的性能全面占优。需要指出的是，分布式服务框架在面向不同领域应用时，需求也不同。例如大型网站内部的分布式服务框架的特点是组网规模大、高并发、海量的小消息通信、以内部服务调用为主，此类场景就比较适合使用高性能的 Protobuf 作为序列化框架。对于企业内部 IT 系统，服务框架序列化可以选择 Json/XML 等文本类数据格式，它的可读性更好。

4.3 扩展性设计

利用 Netty 提供的编解码框架，可以非常快速的实现序列化/反序列化框架的扩展，Netty 内置了丰富的序列化/反序列化功能类库，用户可以直接使用，避免二次开发。

4.3.1 内置的序列化/反序列化功能类

为了降低用户的开发难度，Netty 对常用的功能和 API 做了装饰，以屏蔽底层的实现细节。编解码功能的定制，对于熟悉 Netty 底层实现的开发者而言，直接基于 ChannelHandler 扩展开发，难度并不是很大。但是对于大多数初学者或者不愿意去了解底层实现细节的用

户，需要提供给他们更简单的类库和 API，而不是 ChannelHandler。

Netty 在这方面做得非常出色，针对编解码功能，它既提供了通用的编解码框架供用户扩展，又提供了常用的编解码类库供用户直接使用。在保证定制扩展性的基础之上，尽量降低用户的开发工作量和开发门槛，提升开发效率。

Netty 预置的编解码功能列表如图 4-3 所示，包括 base64、Protobuf、JBoss Marshalling、spdy 等。



图 4-3 Netty 预置的编解码功能列表

4.3.2 反序列化扩展

在分布式服务框架中，反序列化的扩展包括两部分：

- 1) 业务发布服务的时候，可以指定协议类型和承载数据的序列化方式，例如将购买商品服务发布成 HTTP 服务，序列化格式采用 XML；同时允许用户指定新增的序列化格式发布服务。
- 2) 序列化类库能够以插件的格式插入到通信调用链中，实现序列化格式的扩展。在这个过程中，需要考虑 TCP 的粘包和拆包等底层相关的技术细节。

我们首先看序列化功能的扩展，以 MessagePack 序列化框架为例，我们首先定义 MessagePack 解码器，它继承自 Netty 的 MessageToMessageDecoder，代码示例如下：

```
public class MsgpackDecoder extends MessageToMessageDecoder<ByteBuf>{
    @Override
    protected void decode(ChannelHandlerContext arg0, ByteBuf arg1,
        List<Object> arg2) throws Exception {
        final byte[] array;
        final int length = arg1.readableBytes();
        array = new byte[length];
        arg1.getBytes(arg1.readerIndex(), array, 0, length);
        MessagePack msgpack = new MessagePack();
        arg2.add(msgpack.read(array));
    }
}
```

首先从数据报 arg1 中获取需要解码的 byte 数组，然后调用 MessagePack 的 read 方法将其反序列化为 Object 对象，将解码后的对象加入到解码列表 arg2 中，这样就完成了 MessagePack 的解码操作。

接着我们看半包的处理，如果不处理半包，Netty 调用 decode 方法传递的 ByteBuf 对象可能就是个半包，我们拿半包做反序列化就会失败，因此在反序列化之前，我们需要保证调用解码方法时传递的是个完整的数据包。

了解 TCP 通信机制的读者应该都知道 TCP 底层的粘包和拆包，当我们在接收消息的时候，不能认为读取到的报文就是个整包消息，特别是对于采用非阻塞 I/O 和长连接通信的程序。

如何区分一个整包消息，通常有如下 4 种做法：

- 1) 固定长度，例如每 120 个字节代表一个整包消息，不足的前面补位。解码器在处理这类定长消息的时候比较简单，每次读到指定长度的字节后再进行解码。
- 2) 通过回车换行符区分消息，例如 HTTP 协议。这类区分消息的方式多用于文本协议。

3) 通过特定的分隔符区分整包消息。

4) 通过在协议头/消息头中设置长度字段来标识整包消息。

针对不同的协议, Netty 提供了 4 种反序列化(解码)工具类, 用户只需要指定参数即可实现半包的处理, 分别如下。

- 1) **LineBasedFrameDecoder**: 回车换行解码器, 如果用户发送的消息以回车换行符作为消息结束的标识, 则可以直接使用 Netty 的 **LineBasedFrameDecoder** 对消息进行解码, 只需要在初始化 Netty 服务端或者客户端时将 **LineBasedFrameDecoder** 正确地添加到 **ChannelPipeline** 中即可, 不需要自己重新实现一套换行解码器。
- 2) **DelimiterBasedFrameDecoder**: 分隔符解码器, 用户可以指定消息结束的分隔符, 它可以自动完成以分隔符作为码流结束标识的消息的解码。回车换行解码器实际上是一种特殊的 **DelimiterBasedFrameDecoder** 解码器。
- 3) **FixedLengthFrameDecoder**: 固定长度解码器, 它能够按照指定的长度对消息进行自动解码, 开发者不需要考虑 TCP 的粘包/拆包等问题, 非常实用。
- 4) **LengthFieldBasedFrameDecoder**: 通用半包解码器。大多数的协议(私有或者公有), 协议头中会携带长度字段, 用于标识消息体或者整包消息的长度, 例如 SMPP、HTTP 协议等。由于基于长度解码需求的通用性, 以及为了降低用户的协议开发难度, Netty 提供了 **LengthFieldBasedFrameDecoder**, 自动屏蔽 TCP 底层的拆包和粘包问题, 只需要传入正确的参数, 即可轻松解决“读半包”问题。

通过代码示例, 我们看下如何解决 **MessagePack** 的读半包问题:

```
public void initChannel(SocketChannel ch)
    throws Exception {
    ch.pipeline().addLast("frameDecoder", new
LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));
    ch.pipeline().addLast("msgpack decoder", new
MsgpackDecoder());
    ch.pipeline().addLast("frameEncoder", new
```

```

LengthFieldPrepender(2));
        ch.pipeline().addLast("msgpack encoder", new
MsgpackEncoder());
        ch.pipeline().addLast(
            new EchoClientHandler(sendNumber));
    }
});
}

```

在 MessagePack 解码器之前增加 LengthFieldBasedFrameDecoder, 用于处理半包消息, 这样后面的 MsgpackDecoder 接收到的永远是整包消息。它的工作原理如图 4-4 所示。



图 4-4 LengthFieldBasedFrameDecoder 工作原理图

随后再新增 MsgpackDecoder, 用于将整包消息使用 Msgpack 反序列化为普通的对象, 实现反序列化。

4.3.3 序列化扩展

序列化扩展比较简单, 我们只需要继承 Netty 的 MessageToByteEncoder 类, 实现 encode 方法即可, 代码示例如下:

```

public class MsgpackEncoder extends MessageToByteEncoder<Object> {
    @Override
    protected void encode(ChannelHandlerContext arg0, Object arg1,
ByteBuf arg2) throws Exception {
        MessagePack msgpack = new MessagePack();
        byte[] raw = msgpack.write(arg1);
    }
}

```

```

        arg2.writeBytes(raw);
    }
}

```

MsgpackEncoder 继承 MessageToByteEncoder，它负责将 Object 类型的 POJO 对象编码为 byte 数组，然后写入到 ByteBuf 中。

然后，将 MsgpackEncoder 编码类添加到 ChannelPipeline 中，当用户发送消息时，Netty 会自动将消息按照 Msgpack 的格式序列化为二进制 byte 数组，代码如下：

```

public void initChannel(SocketChannel ch)
    throws Exception {
    ch.pipeline().addLast("frameDecoder", new
LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));
    ch.pipeline().addLast("msgpack decoder", new
MsgpackDecoder());
    ch.pipeline().addLast("frameEncoder", new
LengthFieldPrepender(2));
    ch.pipeline().addLast("msgpack encoder", new
MsgpackEncoder());
    ch.pipeline().addLast(
        new EchoClientHandler(sendNumber));
}
}

```

需要指出的是，在 MsgpackEncoder 之前添加了 LengthFieldPrepender，它的职责是根据用户发送的二进制数组长度，将消息长度写入到消息头中，占用 2 个字节，它的工作原理示意图如图 4-5 所示。



图 4-5 LengthFieldPrepender 编码器原理

通过 `LengthFieldPrepender` 可以将待发送消息的长度写入到 `ByteBuf` 的前 2 个字节，编码后的消息组成为“长度字段+原消息”的格式。

4.4 最佳实践

分布式服务框架在实际应用中，会遇到很多挑战，例如服务前向兼容性、服务调用时延等，本节针对序列化相关的最佳实践进行总结，帮助设计和开发人员尽量少走弯路。

4.4.1 接口的前向兼容性规范

业务在最初使用分布式服务框架时，通常不会提服务前向兼容性需求。当服务线上运行一段时间之后，由于种种原因，服务需要升级，此时，服务兼容性问题就凸现出来。

服务的前向兼容性仅仅依靠分布式服务框架或者业务自身，都无法实现。需要双方形成合力，才能比较好地解决这个问题。

就分布式服务框架而言，需要做的事情有两个：

- 1) 制定“分布式服务框架接口兼容性规范”，在规范中要明确服务框架支持哪些兼容性，例如新增字段、删除字段还是修改字段；如果这些规范需要用户在开发时感知，例如代码注解等，就要明确规范落地的具体措施，它可能依靠项目组的开发规范或者 IDE 工具。在业务使用分布式服务框架时，作为约束强制要求业务必须遵循该规范。
- 2) 引导客户，在使用初期按照最佳实践设计服务化接口，防止业务上线之后为兼容性做大的重构。

4.4.2 高并发下的稳定性

一些序列化框架功能测试各项指标都很好，但是在高并发压力下，可能会出现时延变大、性能毛刺等问题。这些问题会引起 E2E 时延变大、服务调用超时等问题，严重影响系统的稳定性。

在实际项目开发中，需要模拟现网高并发场景对序列化框架做压测和稳定性测试，如果序列化框架存在全局锁、较激烈的线程竞争等问题，多线程、高并发压力测试就会出现。究其原因是一些序列化框架为了实现线程安全，使用了全局锁等，这从使用角度看确实简单，但是在高并发场景下就会出现性能下降、耗时不稳定等问题。

解决此类问题的方案很多，例如每个线程聚合一个序列化/反序列化类库，避免多线程竞争。在实际工作中，可以具体问题具体分析。

4.5 总结

序列化和反序列化是 RPC 框架的基础组成部分，设计的好坏对服务化框架的性能、可扩展性和可靠性影响都很大。尽管业界存在多款序列化框架，但是我们在设计时需要从功能丰富度、跨语言支持、兼容性、性能，甚至社区活跃度等多个角度去综合考量，从中择优。

第 5 章

协议栈

不同服务在性能上适用不同协议进行传输。比如对接异构第三方服务时，通常会选择 HTTP/Restful 等公有协议；对于内部不同模块之间的服务调用，往往会选择性能较高的二进制私有协议。

5.1 关键技术点分析

大部分服务框架都支持多协议，但是多协议却不是必须的。如果公司内部使用，则可以根据业务需要决定支持的具体协议，如果是开源的通用分布式服务框架，考虑到通用性往往默认会提供一些常用的协议，例如 Web Service 和 HTTP。

5.1.1 是否必须支持多协议

答案是否定的，尽管从功能上看分布式服务框架默认支持的协议种类越多，功能越强大。但是，分布式服务框架与 ESB 最大的差异就是它不负责异构系统的对接，这意味着所有使用分布式服务框架服务化的业务都需要遵循服务化规范，包括通信协议、服务配置等。

是否支持多协议，需要根据业务的实际需要、公司对分布式服务框架的定位而定。如果确实有实际需要，支持多协议也未尝不可。

需要指出的是，分布式服务框架需要具备通过扩展的方式支持多协议的能力，协议栈应该作为一个架构扩展点开放出来。

5.1.2 公有协议还是私有协议

大部分分布式服务框架/RPC 框架内部都采用私有协议进行通信，但是，SOA 理论又告诉我们应该使用标准的 Web 协议（例如 Web Service）提供服务。同样是服务化，为什么有两种截然不同的做法呢？很多读者对此不解。

其实，理解这个也不难。对于经历过企业 IT 系统 SOA 化和业务模块服务化的人而言，很容易理解这两者的差异。

SOA 服务化的目标就是重用 IT 系统已有资产、实现异构系统之间的灵活互通。重用

已有的异构系统，实现不同系统之间的服务调用，最佳的做法就是使用标准的公有协议，其中 Web Service 最合适。

服务提供者通过 WSDL 向注册中心提供服务接口描述，注册中心通过 UDDI 发布服务提供者的服务，服务消费者检索到服务提供者服务信息之后，通过标准的 SOAP 协议调用服务提供者，实现服务远程调用。相比于 Web Service，HTTP 等公有协议缺少服务描述文件、服务注册中心和服务订阅发布机制，不太适合作为 SOA 服务化的标准协议。

分布式服务框架主要是为了解决内部服务化之后业务接口跨进程通信问题，吞吐率、时延等性能指标是关键。在性能方面，私有协议往往可以根据业务的具体需求进行针对性优化，因此性能更优。

以 Web Service 公有协议为例，它的性能存在如下缺陷：

- 1) SOAP 消息使用 XML 进行序列化，相比于 PB 等二进制序列化框架，性能低一大截。
- 2) SOAP 通常由 HTTP 协议承载，HTTP1.0/1.1 不支持双向全双工通信，而且一般使用短连接通信，性能比较差。

如果没有特殊需求，分布式服务框架默认使用性能更高、扩展性更好的私有协议（二进制）进行通信。对于部分特殊需要与外部对接的服务，可以考虑引入 HTTP/Restful 等公有协议。

5.1.3 集成开源还是自研

分两种场景，对于私有协议，通常是自研的，因为开源的 RPC 框架等协议栈与序列化框架，甚至是功能都耦合在一起，想剥离难度很大。另外就是开源方案的功能满足度、灵活性等很难满足业务需求，往往要做二次优化。

对于公有协议，建议如下：

- 1) 如果业界已经有比较成熟的开源框架，在性能、功能、可靠性等方面能够满足需

求，则优先选择采用集成开源框架的方案。

- 2) 如果使用到的功能不多，或者集成的开源框架在关键技术指标上不能完全满足需求，改造工作量太大，也可以考虑基于通信框架（基于 Netty）自研。

5.2 功能设计

公有协议栈往往采用集成开源的方案，集成起来比较简单，只需要做简单 Facade 即可，本节不做详细介绍。对于需要自研的私有协议栈，本节将展开重点讲解。

5.2.1 功能描述

私有协议栈承载了业务内部各模块之间的消息交互和服务调用，它的主要功能如下：

- 1) 定义了私有协议的通信模型和消息定义。
- 2) 支持服务提供者和消费者之间采用点对点长连接通信。
- 3) 基于 Java NIO 通信框架，提供高性能的异步通信能力。
- 4) 提供可扩展的编解码框架，支持多种序列化格式。
- 5) 握手和安全认证机制。
- 6) 链路的高可靠性。

5.2.2 通信模型

私有协议栈通信模型如图 5-1 所示。



图 5-1 私有协议栈通信模型

服务提供者和消费者之间采用单链路、长连接的方式进行网络通信，链路创建流程如下：

- 1) 客户端发送握手请求消息，携带节点 ID 等有效身份认证信息。
- 2) 服务端对握手请求消息进行合法性校验，包括节点 ID 有效性校验、节点重复登录校验和 IP 地址合法性校验，校验通过后，返回登录成功的握手应答消息。
- 3) 链路建立成功之后，客户端发送业务消息。
- 4) 链路成功之后，服务端发送心跳消息。
- 5) 链路建立成功之后，客户端发送心跳消息。
- 6) 链路建立成功之后，服务端发送业务消息。
- 7) 服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

需要指出的是，服务消费者和服务提供者双方链路建立成功之后，就可以进行全双工通信。无论客户端还是服务端，都可以主动发送请求消息给对方，通信方式可以是 TWO WAY 或者 ONE WAY。

双方之间的心跳采用 Ping-Pong 机制，当链路处于空闲状态时，客户端主动发送 Ping 消息给服务端，服务端接收到 Ping 消息后发送应答消息 Pong 给客户端，如果客户端连续

发送 N 条 Ping 消息都没有接收到服务端返回的 Pong 消息,说明链路已经挂死或者对方处于异常状态,客户端主动关闭连接,间隔周期 T 后发起重连操作,直到重连成功。

5.2.3 协议消息定义

通常协议栈的消息模型分为两部分,消息头和消息体。消息头存放协议公共字段和用户扩展字段,消息体则用于承载消息内容。以 HTTP 协议为例,请求消息头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息,常用的消息头关键字有 Accept、Authorization、Host 等。

私有协议的消息模型与公有协议类似,它也包含消息头和消息体两部分。它的参考定义如表 5-1 所示。

表 5-1 消息定义模型

名 称	类 型	长 度	描 述
header	Header	变长	消息头定义
body	byte []	变长	字节数组:对于请求消息,它是方法的参数;对于响应消息,它是返回值

消息头通常包含服务调用相关的公共参数,参考定义如表 5-2 所示。

表 5-2 消息头定义

名 称	类 型	长 度	描 述
crcCode	整型 int	32	协议栈校验码,它由三部分组成。 1) 0xAFBA: 固定值,表明该消息是私有协议消息,2 个字节 2) 主版本号: 1~255, 1 个字节 3) 次版本号: 1~255, 1 个字节 $crcCode = 0xAFBA + \text{主版本号} + \text{次版本号}$
length	整型 int	32	消息长度,整个消息,包括消息头和消息体

续表

名 称	类 型	长 度	描 述
type	byte	8	0: 业务请求消息 1: 业务响应消息 2: 业务 ONE WAY 消息（既是请求又是响应消息） 3: 握手请求消息 4: 握手应答消息 5: 心跳请求消息 6: 心跳应答消息
priority	byte	8	消息优先级: 0~255
interfaceName	string	变长	接口名
methodName	string	变长	方法名
attachment	Map<String, String>	变长	可选字段, 用于扩展消息头

消息头中最后一个字段 **attachment** 是个 **Map**, 用于协议栈自身、用户协议扩展使用, 例如可以设置 **TraceID** 表示全流程业务跟踪的消息 ID。消息体是个 **byte** 数组, 它的设置存在如下四种场景。

- 1) 请求消息: 服务提供者接口请求参数类型编码 + 请求参数值编码。
- 2) 应答消息: 服务提供者返回给消费者的应答消息编码。
- 3) 心跳消息: 不需要设置。
- 4) 握手应答消息: 应答结果编码。

5.2.4 协议栈消息序列化支持的字段类型

协议栈可以由不同的序列化框架承载, 不管使用哪种序列化方式, 承载在它之上的都是业务消息, 业务消息与协议和序列化框架无关。因此, 需要在协议栈层面约束支持的数据结构类型, 防止用户使用不支持的数据结构导致编码或者解码失败。

考虑到跨语言通用性，协议栈支持的数据结构类型尽量大众化，如表 5-3 所示。

表 5-3 协议栈支持的数据类型

字段类型	备注说明
boolean	包括它的包装类型 Integer
byte	包括它的包装类型 Byte
int	对应于 C/C++的 int32
char	包括它的包装类型 Character
short	对应 C/C++的 int16
long	对应 C/C++的 int64
float	包括它的包装类型 Float
double	包括它的包装类型 Double
string	对应 C/C++的 String
list	支持各种 List 的实现
array	支持各种数组的实现
map	支持 Map 的嵌套和泛型
set	支持 Set 的嵌套和泛型

5.2.5 协议消息的序列化和反序列化

消息的序列化分为两部分，消息头的序列化和消息体的序列化，两者采用的机制不一样。原因是协议栈可以由不同的序列化框架承载，标识序列化格式的字段在消息头中定义，因此我们必须首先对消息头做通用解码，获取序列化格式，然后根据类型再调用对应的解码器对消息体做解码。如果消息头的序列化不是通用的，我们就无法对其做反序列化。

消息头的编码规范如下。

- ◎ `crcCode`: `java.nio.ByteBuffer.putInt(int value)`，如果采用其他缓冲区实现，必须与其等价。
- ◎ `length`: `java.nio.ByteBuffer.putInt(int value)`，如果采用其他缓冲区实现，必须与其等价。

- ◎ **type**: `java.nio.ByteBuffer.put(byte b)`, 如果采用其他缓冲区实现, 必须与其等价。
- ◎ **priority**: `java.nio.ByteBuffer.put(byte b)`, 如果采用其他缓冲区实现, 必须与其等价。
- ◎ **interfaceName**: 将接口名编码为 `byte` 数组, 然后先编码长度, 再编码内容。
- ◎ **methodName**: 将方法名编码为 `byte` 数组, 然后先编码长度, 再编码内容。
- ◎ **attachment**: 它的编码规则为——如果 **attachment** 长度为 0, 表示没有附件, 则将长度编码设为 0, `java.nio.ByteBuffer.putInt(0)`; 如果大于 0, 说明有附件需要编码, 具体的编码规则如下。
 - 首先对附件的个数进行编码, `java.nio.ByteBuffer.putInt(attachment.size())`。
 - 然后对 **Key** 进行编码, 先编码长度, 再将它转换成 `byte` 数组之后编码内容, 具体代码如下。

```
String key = null;
String value = null;
for (Map.Entry<String, Object> param : attachment.entrySet()) {
    key = param.getKey();
    buffer.writeString(key);
    value = param.getValue();
    buffer.writeString(value);
}
key = null;
value = null;
```

- ◎ **body** 的编码: 通过服务发布时指定的序列化格式将其序列化为 `byte` 数组, 然后调用 `java.nio.ByteBuffer.put(byte [] src)` 将其写入 `ByteBuffer` 缓冲区中。

由于整个消息的长度必须等全部字段都编码完成之后才能确认, 所以最后需要更新消息头中的 **length** 字段, 将其重新写入 `ByteBuffer` 中。

协议消息的反序列化（解码）规范如下。

- ◎ **crcCode**: 通过 `java.nio.ByteBuffer.getInt()` 获取校验码字段，其他缓冲区需要与其等价。
- ◎ **length**: 通过 `java.nio.ByteBuffer.getInt()` 获取协议消息的长度，其他缓冲区需要与其等价。
- ◎ **type**: 通过 `java.nio.ByteBuffer.get()` 获取消息类型，其他缓冲区需要与其等价。
- ◎ **priority**: 通过 `java.nio.ByteBuffer.get()` 获取消息优先级，其他缓冲区需要与其等价。
- ◎ **interfaceName**: 通过 `java.nio.ByteBuffer.getInt()` 获取接口名称长度，然后再根据长度获取内容 `byte[]`，最后根据字节数组构造 `String` 类型的接口名。
- ◎ **methodName**: 通过 `java.nio.ByteBuffer.getInt()` 获取方法名称长度，然后再根据长度获取内容 `byte[]`，最后根据字节数组构造 `String` 类型的方法名。
- ◎ **attachment**: 它的解码规则为——首先创建一个新的 `attachment` 对象，调用 `java.nio.ByteBuffer.getInt()` 获取附件的长度，如果为 0，说明附件为空，解码结束，继续解消息体；如果非空，则根据长度通过 `for` 循环进行解码。

```
String key = null;
String value = null;
for (int i = 0; i < size; i++) {
    key = buffer.readString();
    value = buffer.readString();
    this.attachment.put(key, value);
}

key = null;
value = null;
```

- ◎ **body**: 根据消息类型，按照指定的编码规则，对消息体进行解码。

5.2.6 链路创建

协议栈包括服务端和客户端，对于上层应用程序而言，不需要刻意区分到底是客户端还是服务端，在分布式组网环境中，一个节点可能既是服务端也是客户端，这个依据具体的用户场景而定。

客户端主动发起连接流程：如果 A 节点需要调用 B 节点的服务，但是 A 和 B 之间还没有建立物理链路，则由调用方主动发起连接，此时，调用方为客户端，被调用方为服务端。

考虑到安全，链路建立需要通过基于 IP 地址或者号段的黑白名单安全认证机制，作为样例，本协议使用基于 IP 地址的安全认证，如果有多个 IP，通过逗号进行分割。在实际商用项目中，安全认证机制会更加严格，例如通过密钥对用户名和密码进行安全认证。

客户端与服务端链路建立成功之后，由客户端发送握手请求消息，握手请求消息的定义如下。

- ◎ 消息头的 `type` 字段值为 3。
- ◎ 可选附件个数为 0。
- ◎ 消息体为空。

服务端接收到客户端的握手请求消息之后，如果 IP 校验通过，返回握手成功应答消息给客户端，应用层链路建立成功。握手应答消息定义如下。

- 1) 消息头的 `type` 字段值为 4。
- 2) 可选附件个数为 0。
- 3) 消息体为 `byte` 类型的结果，0：认证成功；-1：认证失败。

链路建立成功之后，客户端和服务端就可以互相发送业务消息了。

5.2.7 链路关闭

由于采用长连接通信，在正常的业务运行期间，双方通过心跳和业务消息维持链路，任何一方都不需要主动关闭连接。

但是，在以下情况下，客户端和服务端需要关闭连接：

- 1) 当对方宕机或者重启时，会主动关闭链路，另一方读取到操作系统的通知信号，得知对方 REST 链路，需要关闭连接，释放自身的句柄等资源。由于采用 TCP 全双工通信，通信双方都需要关闭连接，释放资源。
- 2) 消息读写过程中，发生了 I/O 异常，需要主动关闭连接。
- 3) 心跳消息读写过程中发生了 I/O 异常，需要主动关闭连接。
- 4) 心跳超时，需要主动关闭连接。
- 5) 发生编码异常等不可恢复错误时，需要主动关闭连接。

5.3 可靠性设计

分布式服务框架可能会运行在非常恶劣的网络环境中，网络超时、闪断、对方进程僵死或者处理缓慢等情况都有可能发生。为了保证在这些极端异常场景下服务仍能够正常调用或者自动恢复，需要底层的协议栈支持高 HA。

5.3.1 客户端连接超时

在传统的同步阻塞编程模式下，客户端 Socket 发起网络连接，往往需要指定连接超时时间，这样做的目的主要有两个：

- 1) 在同步阻塞 I/O 模型中，连接操作是同步阻塞的，如果不设置超时时间，客户端 I/O 线程可能会被长时间阻塞，这会导致系统可用 I/O 线程数的减少。

- 2) 业务层需要：大多数系统都会对业务流程执行时间有限制，例如 Web 交互类的响应时间要小于 3s。客户端设置连接超时时间是为了实现业务层的超时。

对于 NIO 的 `SocketChannel`，在非阻塞模式下，它会直接返回连接结果，如果没有连接成功，也没有发生 IO 异常，则需要将 `SocketChannel` 注册到 `Selector` 上监听连接结果。所以，异步连接的超时无法在 API 层面直接设置，而是需要通过用户自定义定时器来主动监测。

Netty 在创建 NIO 客户端时，支持设置连接超时参数。Netty 的客户端连接超时参数与其他常用的 TCP 参数一起配置，使用起来非常方便，上层用户不用关心底层的超时实现机制。这既满足了用户的个性化需求，又实现了故障的分层隔离。

5.3.2 客户端重连机制

客户端通过链路关闭监听器监听链路状态，如果链路中断，等待 INTERVAL 时间后，由客户端发起重连操作，如果重连失败，间隔周期 INTERVAL 后再次发起重连，直到重连成功。

为了保证服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待 INTERVAL 时间之后再发起重连，而不是失败后就立即重连。

为了保证句柄资源能够及时释放，无论什么场景下的重连失败，客户端都必须保证自身的资源被及时释放，包括但不限于 `SocketChannel`、`Socket` 等。

重连失败后，需要打印异常堆栈信息，方便后续的问题定位。为了防止服务端正常下线长期不再上线，客户端通常会对重连次数做限制，防止无限重连下去无谓的损耗资源。

5.3.3 客户端重复握手保护

当客户端握手成功之后，在链路处于正常状态下，不允许客户端重复握手，以防止客户端在异常状态下反复重连导致句柄资源被耗尽。

服务端接收到客户端的握手请求消息之后，首先对 IP 地址进行合法性检验，如果校验成功，在缓存的地址表中查看客户端是否已经登录，如果已经登录，则拒绝重复登录，返回错误码-1，同时关闭 TCP 链路，并在服务端的日志中打印握手失败的原因。

客户端接收到握手失败的应答消息之后，关闭客户端的 TCP 连接，等待 INTERVAL 时间之后，再次发起 TCP 连接，直到认证成功。

为了防止由服务端和客户端对链路状态理解不一致导致的客户端无法握手成功的问题，当服务端连续 N 次心跳超时之后需要主动关闭链路，清空该客户端的地址缓存信息，以保证后续该客户端可以重连成功，防止被重复登录保护机制拒绝掉。

5.3.4 消息缓存重发

无论客户端还是服务端，当发生链路中断之后，在链路恢复之前，缓存在消息队列中待发送的消息不能丢失，等链路恢复之后，重新发送这些消息，保证链路中断期间消息不丢失。

考虑到内存溢出的风险，建议消息缓存队列设置上限，当达到上限之后，应该拒绝继续向该队列添加新的消息。

5.3.5 心跳机制

在凌晨等业务低谷期时段，如果发生网络闪断、连接被 Hang 住等网络问题时，由于没有业务消息，应用进程很难发现。到了白天业务高峰期时，会发生大量的网络通信失败，严重的会导致一段时间进程内无法处理业务消息。为了解决这个问题，在网络空闲时采用心跳机制来检测链路的互通性，一旦发现网络故障，立即关闭链路，主动重连。

5.4 安全性设计

为了保证整个集群环境的安全，内部长连接采用基于 IP 地址的安全认证机制，服务

端对握手请求消息的 IP 地址进行合法性校验：如果在白名单之内，则校验通过；否则，拒绝对方连接。

如果需要将服务开放给第三方非信任域的消费者，需要采用更加严格的安全认证机制，例如基于密钥和 AES 加密的用户名+密码认证机制，也可以采用 SSL/TLS 安全传输。

协议栈的安全原理如图 5-2 所示。

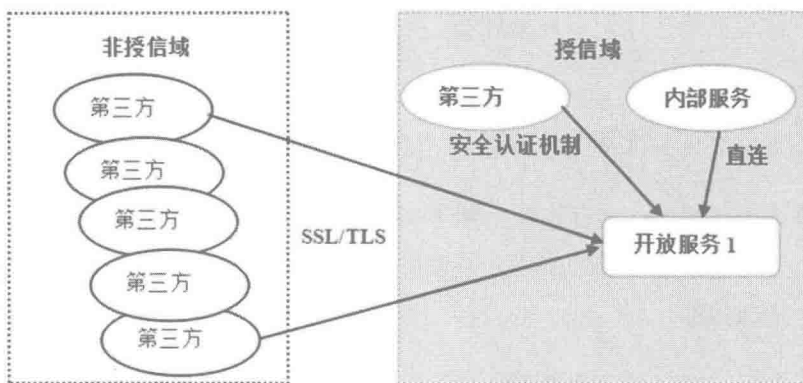


图 5-2 对第三方开放的服务框架

对第三方开放的分布式服务框架的服务调用存在三种场景：

- 1) 在企业内网，开放给内部其他模块调用的服务，通常不需要进行安全认证和 SSL/TLS 传输。
- 2) 在企业内网，被外部其他模块调用的服务，往往需要利用 IP 黑白名单、握手登录等方式进行安全认证，认证通过之后双方使用普通的 Socket 进行通信，如果认证失败，则拒绝客户端连接。
- 3) 开放给企业外部第三方应用访问的服务，往往需要监听公网 IP（通常是防火墙的 IP 地址），由于对第三方服务调用者的监管存在诸多困难，或者无法有效监管，这些第三方应用实际是非授信的。为了有效应对安全风险，对于敏感的服务往往需要通过 SSL/TLS 进行安全传输。

5.5 最佳实践——协议的前向兼容性

细节决定成败，协议栈设计亦如此。由于 Java 存在大量开源的协议栈，我们通常奉行“拿来主义”，大多数同学都没有完整设计并实现过协议栈，因此协议栈的设计开发是个难点。

下面我们就对实际工作中可能会踩到的坑总结下，通过最佳实践的方式分享给大家。

随着业务的不断发展，功能不断增强，对分布式服务框架的需求也越来越多，例如通过隐式传参让协议消息头携带业务消息上下文发送给服务端。

对于分布式服务框架，功能也在不断增加，例如支持不同的序列化方式，需要在消息头中新增一个标识序列化方式的字段。如果协议栈没有良好的可扩展性和兼容性设计，就会遇到很多棘手的问题。

服务上线之后，来自不同业务团队的消费者会依赖某类服务提供者，假如服务提供者升级了协议栈，协议栈不支持前向兼容，消费者发送的老版本的消息服务端将不能识别，站在服务提供者角度，希望所有依赖此服务的消费者都升级。但是消费者可能来自公司的不同业务线、公司外的第三方合作伙伴、手机终端用户等，让所有消费者同时全量升级，显然不现实。

因此，协议栈的前向兼容性和可扩展性非常重要，在 5.2.3 节我们设计消息模型的时候，已经考虑到了协议的前向兼容性，核心的设计原则有 2 个：

- 1) 消息头第一个字段中携带协议的版本号，用于标识消息协议版本。
- 2) 消息头最后一个字段是 Map 类型的扩展字段，用于服务框架自身或者用户扩展消息头。

如果涉及到的不兼容的修改，建议升级协议版本号；如果是新增部分特性，建议通过 Map 扩展的方式进行，不要升级协议版本。

5.6 总结

最后总结一下通信框架、序列化/反序列化和协议栈的关系：协议描述了分布式服务框架的通信契约，序列化和反序列化框架用于协议消息对象和二进制数组之间的相互转换，通信框架在技术上承载协议，协议要落地，需要依赖通信框架提供的基础通信能力。

第 6 章

服务路由

分布式服务框架上线运行时都是集群组网，这意味着集群中存在某个服务的多实例部署，消费者如何从服务列表中选择合适的服务提供者进行调用，这就涉及到服务路由。分布式服务框架要能够满足用户灵活的路由需求。

6.1 透明化路由

很多开源的 RPC 框架调用者需要配置服务提供者的地址信息，尽管可以通过读取数据库的服务地址列表等方式避免硬编码地址信息，但是消费者依然要感知服务提供者的地址信息，这违反了透明化路由原则。

6.1.1 基于服务注册中心的订阅发布

在分布式服务框架中，服务注册中心用于存储服务提供者地址信息、服务发布相关的属性信息，消费者通过主动查询和被动通知的方式获取服务提供者的地址信息，而不需要像之前那样在代码中硬编码服务提供者地址信息。消费者只需要知道当前系统发布了哪些服务，而不需要知道服务具体存在于什么位置，这就是透明化路由。它的工作原理就是基于服务注册中心（例如 ZooKeeper）的订阅发布机制。

服务注册中心的工作原理如图 6-1 所示。

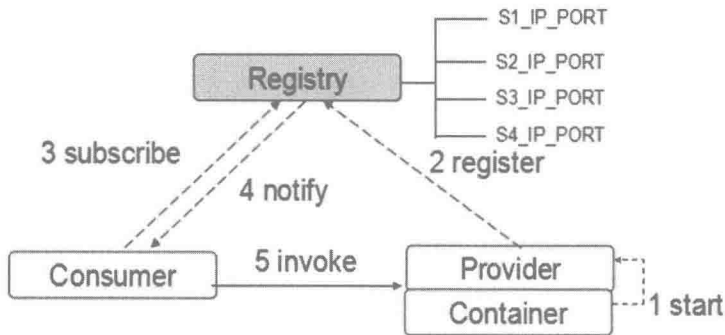


图 6-1 服务注册中心工作原理

服务消费者和服务提供者通过注册中心提供的 SDK 与注册中心建立链路（例如 ZooKeeper 采用长连接），服务提供者将需要发布的服务地址信息和属性列表写入注册中心。服务消费者根据本地引用的接口名等信息，从服务注册中心获取服务提供者列表，缓存到本地。

由于消费者可能先于服务提供者启动，或者系统运行过程中新增服务提供者，或者某个服务提供者宕机退出，就会导致注册中心发生服务提供者地址变更。注册中心检测到服务提供者列表变更之后，将变更内容主动推送给服务消费者，消费者根据变更列表，动态刷新本地缓存的服务提供者地址。

6.1.2 消费者缓存服务提供者地址

消费者调用服务提供者时，不需要每次调用都去服务注册中心查询服务提供者地址列表，消费者客户端直接从本地缓存的服务提供者路由表中查询地址信息，根据路由策略进行服务选路。

当服务提供者发生变更时，注册中心主动将变更内容推送给消费者，由后者动态刷新本地缓存的服务路由表，保证服务路由信息的实时准确性。

采用客户端缓存服务提供者地址的方案不仅仅能提升服务调用性能，还能保证系统的可靠性。当注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存的地址信息进行通信，只是影响新服务的注册和老服务的下线，不影响已经发布和运行的服务。

6.2 负载均衡

负载均衡策略是服务重要的属性，分布式服务框架通常会提供多种负载均衡策略，同时支持用户扩展负载均衡策略。

6.2.1 随机

采用随机算法进行负载均衡，通常在对等集群组网中，随机路由算法消息分发还是比较均匀的。它的主要缺点有两个：

1) 在一个截面上碰撞的概率较高。

2) 非对等集群组网，或者硬件配置差异较大，会导致各节点负载不均匀。

通常在实现上会采用 JDK 提供的 `java.util.Random` 或者 `java.security.SecureRandom` 在指定服务提供者地址列表中生成随机地址，消费者基于随机生成的服务提供者地址进行远程调用。

6.2.2 轮循

轮循，按公约后的权重设置轮循比率，到达边界之后，继续绕接。它的主要缺点是存在慢的提供者累积请求问题。比如第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

轮循策略的实现非常简单，它的原理就是按照权重，顺序循环遍历服务提供者列表，到达上限之后重新归零，继续顺序循环。

6.2.3 服务调用时延

消费者缓存所有服务提供者的服务调用时延，周期性的计算服务调用平均时延，然后计算每个服务提供者服务调用时延与平均时延的差值，根据差值大小动态调整权重，保证服务时延大的服务提供者接收更少的消息，防止消息堆积。

该策略的特点就是要保证处理能力强的服务提供者接收到更多的消息，通过动态自动权重调整消除服务调用时延的振荡范围，使所有服务提供者服务调用时延接近平均值，实现负载均衡。

基于服务调用时延负载均衡调整之前效果如图 6-2 所示，振荡范围非常大。

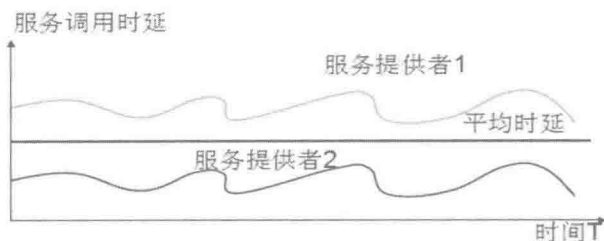


图 6-2 基于服务调用时延负载均衡调整前

调整之后效果如图 6-3 所示，服务调用时延基本相同，围绕着平均时延做小范围振荡。

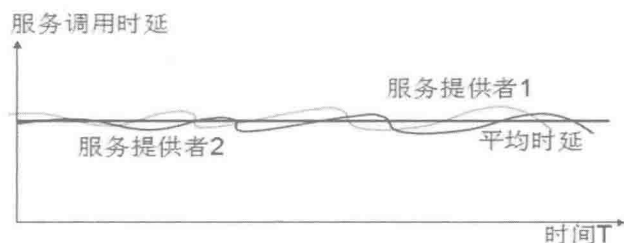


图 6-3 基于服务调用时延负载均衡调整后

6.2.4 一致性哈希

相同参数的请求总是发到同一个服务提供者，当某一台提供者宕机时，原本发往该提供者的请求，基于虚拟节点，平摊到其他提供者，不会引起剧烈变动。平台提供默认的虚拟节点数，可以通过配置参数进行修改。

一致性 Hash 环工作原理如图 6-4 所示。

一致性 Hash 算法已经非常成熟，各语言版本的代码都是现成的，在实现时可以直接引用到项目中。

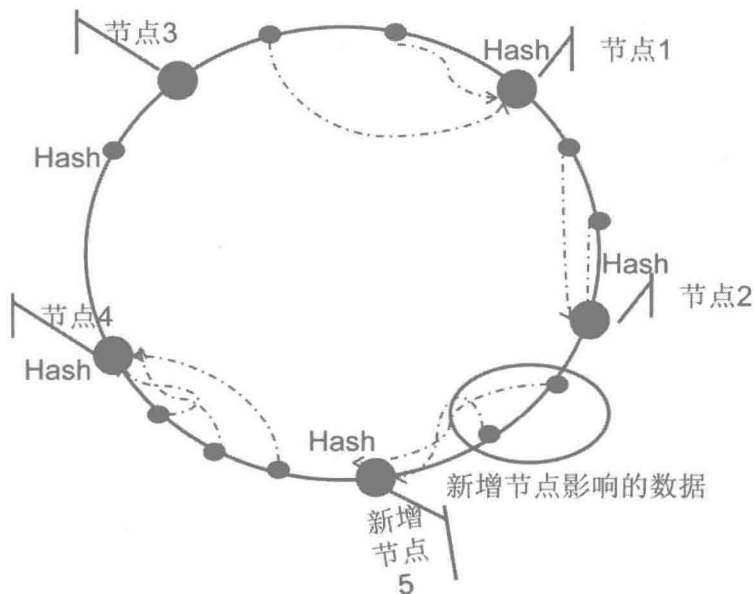


图 6-4 一致性 Hash 环原理

6.2.5 粘滞连接

粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起服务调用，除非该提供者宕机，再连接另一台。由于服务通常被强烈建议设计成无状态的，因此，粘滞连接在实际项目中很少使用。

粘滞连接的实现比较简单，客户端首次跟服务端创建链路时，将该链路标记为粘滞连接，每次路由时直接选择粘滞连接，不执行负载均衡路由接口。当链路中断时，更新粘滞连接为不可用，重新寻找下一个可用的连接，将其标记为粘滞连接。

6.3 本地路由优先策略

6.3.1 injvm 模式

在一些业务场景中，本地 JVM 内部也发布了需要消费的服务。该场景下，从性能、可靠性等角度考虑，需要优先调用本 JVM 内部的服务提供者，这种本地优先的路由模式被称为 injvm 模式。

injvm 协议是一个伪协议，它不开启端口，也不发起远程服务调用，优先调用 JVM 内的服务提供者，相比于其他路由策略，它的执行流程相同，只不过将远程服务调用替换成了内部调用。

如果用户配置了 injvm 路由模式，则优先寻找本 JVM 内的服务提供者，如果没有找到相关服务，则发起远程服务调用。对于上层用户而言，不感知底层具体的调用方式。

6.3.2 innative 模式

如果物理机或者 VM 配置比较好，多个应用进程往往会选择合设。服务消费者和服务提供者可能会被部署到同一台机器上（VM）。服务路由时优先选择本机的服务提供者，如果找不到再重新发起远程服务调用，该模式被称为 innative 模式。

innative 模式的处理流程如下：

- 1) 首先看本进程 JVM 内部是否有符合要求的服务提供者，如果有直接发起本地 API 调用。
- 2) JVM 内部没有，选择服务提供者 IP 地址与本机 IP 地址相同的本地合设的服务提供者进程，通过本地网卡回环调用服务提供者。
- 3) 如果服务器（VM）内部找不到符合条件的服务提供者，则发起远程服务调用。

innative 模式的优点在于通信流量走本地网卡回环，不需要占用网络带宽。另外，可靠性更强、服务调用时延也更低，更节省带宽等资源。

6.4 路由规则

负载均衡、本地路由优先等路由策略通常可以满足大部分业务的线上需求，但是在一些场景中需要对路由策略设置一些过滤条件，比较常用的是基于表达式的条件路由和脚本路由。

6.4.1 条件路由规则

条件路由的应用场景如下。

- 1) 通过 IP 条件表达式进行黑白名单访问控制，例如 `consumerIP != 192.168.1.1`。
- 2) 流量引导，只暴露部分服务提供者，防止整个集群服务都被冲垮，导致其他服务也不可用，例如 `providerIP = 192.168.3*`。
- 3) 读写分离： `method = find*,list*,get*,query* => providerIP = 192.168.1.*`。
- 4) 前后台分离： `app = web* => providerIP = 192.168.1.*`, `app = java* => providerIP = 192.168.2.*`。
- 5) 灰度升级，将 Web 前台应用路由到新的服务版本上： `app = web* => providerIP = 192.168.1.*`。

基于条件表达式的路由规则主要用于地址二次过滤，它由两部分组成：规则和表达式。下面我们分别对它们进行说明。

条件规则由两部分组成：消费者规则和服务提供者规则，可以用分隔符将它们隔离开，

例如 =>、: 等。分隔符之前的为消费者匹配条件，所有参数和消费者的 URL 进行对比，当消费者满足匹配条件时，对该消费者执行后面的过滤规则。分隔符之后为提供者地址列表的过滤条件，所有参数和提供者的 URL 进行对比，消费者最终只拿到过滤后的地址列表。

对空条件规则的支持：如果匹配条件为空，表示对所有消费方都适用；如果过滤条件为空，表示禁止访问所有服务提供者。

表达式由以下三部分组成。

- ◎ 参数：主要包括服务提供者的属性信息、消费者的属性信息。例如 method、IP 等。
- ◎ 条件：就是 Java 表达式，例如等于 (=)、不等于 (!=)、或 (|) 与 (&) 等。
- ◎ 值：包括常量，例如 192.168.1.1，模糊匹配*，变量 \$+变量名，例如\$parameter。

6.4.2 脚本路由规则

脚本路由规则的特点是通常都支持动态编译，修改后可以实时生效，不需要重启系统，这在线上动态调整路由规则时非常有效。

经常使用的路由脚本语言有 JavaScript、Groovy、MVEL 等。它的配置示例如下：

```
"script:/**/com.netty.EchoService?category=routers&dynamic=true&rule=" +
URL.encode("function route(serviceAddress) {
    ... } (serviceAddress)"
)
```

其中，route (serviceAddress) 是通过脚本实现的路由规则，它的代码示例如下：

```
function route(serviceAddress)
{
    InetAddress addr = InetAddress.getLocalHost();
```

```
String ip= addr.getHostAddress().toString();//获得本机 IP
for(Address addr : serviceAddress)
    if (ip.equals(addr.getIP()))
        return addr;
return serviceAddress.get(0);
}
```

6.5 路由策略定制

平台除了提供默认的路由策略之外，在架构上还需要支持业务扩展路由算法，实现业务自定义路由。

自定义路由的主要应用场景如下。

- 1) 灰度升级，用户需要按照业务规则进行灰度路由：例如按照用户省份路由、按照请求来源终端类型（IOS、安卓等）、按照手机号段等；不同的用户按照规则路由到不同的集群环境中，例如没有同步升级的用户路由到升级前的环境，同步配套升级的消费者请求路由到灰度升级后的新版本中。
- 2) 服务故障、业务高峰期的导流：通过自定义路由，将异常的峰值流量导流到几台或者 1 台服务器上，防止整个集群负载过重导致整个生产系统雪崩。
- 3) 与业务领域强相关的非通用路由定制需求。

路由扩展策略如下：

- ◎ 平台提供路由接口供用户实现。
- ◎ 平台提供路由配置 XML Schema 定义，支持用户扩展路由规则。
- ◎ 业务发布自定义路由策略，对于通过 Spring Bean 方式的服务发布，用户定义扩展路由 bean，然后在服务提供者配置路由规则的时候，引用相关扩展的 bean 即

可；如果是通过 JDK 的 SPI 方式扩展，则需要将 jar 包的 META-INF/services 目录下的路由策略文件中新增扩展的路由实现类和策略名。

通过扩展的方式定制分布式服务框架路由策略，可以非常容易地实现平台和基线、基线、业务定制的分层，通过扩展的方式增加新的功能，可以保证微内核架构的相对稳定性，防止平台架构腐化。新增路由策略不需要平台发布新的版本，业务可以实现更敏捷的项目交付。

6.6 配置化路由

路由配置策略设计如下。

- 1) 本地配置：包括服务提供者和服务消费者、默认全局配置三种。
- 2) 统一注册管理：无论是服务提供者还是消费者，本地配置的路由策略统一注册到服务注册中心，进行集中化配置管理。
- 3) 动态下发：运维人员通过服务治理 Portal 修改路由规则，更新后的路由规则被持久化到服务注册中心。服务注册中心将变更后的服务路由策略或者规则下发给服务提供者和消费者，消费者更新路由规则库，按照新的路由规则执行，实现路由的动态刷新。

路由配置优先级：客户端配置>服务端配置>全局配置。路由策略支持在服务端和客户端同时配置，为什么要这样设计呢？下面我们对路由配置优先级的设计原理进行详细讲解。

服务提供者更了解服务内部的运行状态，同时，服务提供者通常会和消费者签订《服务质量等级协定（SLA）》，服务提供者需要对服务的调用 Caps、服务调用时延、能够支撑的最大并发数、流控策略等向消费者做出承诺。基于此，服务相关的属性应该由服务提供者配置最合适。

另外一个原因是消费者众多，不是每个消费者都愿意配置服务的路由策略，如果在客

户端配置一个全局的路由策略，配置虽然简单，但是实用性很差，因为不是每个服务路由策略都一样。此时消费者希望由熟悉服务质量属性的服务提供者统一配置路由策略，客户端直接使用，降低开发和维护工作量。

消费者配置路由策略的原因：站在消费者角度，更清楚自己想要达到的路由效果。例如同一个服务提供者，本机、同一个机框和同一个机房的消费者获得的服务调用时延差异都很大，这种差异服务提供者并不清楚，需要消费者根据自己的实际使用效果动态调整，这时客户端配置就需要覆盖服务端配置。

6.7 最佳实践——多机房路由

随着业务的不断发展，单个机房容量已经不足以支撑未来业务的发展，业务开始跨机房部署。

跨机房服务调用会带来时延增加、网络故障概率变大等问题，如何避免跨机房服务调用需要从路由策略上进行考虑。

为了能够相互发现对方的服务，不同机房会共用同一个服务注册中心集群（异地容灾机房除外）。假如机房1发布了服务A，机房2同样也发布了服务A，此时服务注册中心就会将2个不同机房的服务A地址信息推送给消费者，无论是机房1还是机房2的消费者，都将看到两个不同机房的服务。

如果仅仅依靠随机、轮循等负载均衡策略，消息将会被路由到两个机房，达不到不跨机房调用的目标。如何解决跨机房服务调用问题？利用6.4章节的条件路由规则，可以非常便捷地解决问题。

在原有的负载均衡策略基础上，配置条件路由策略，由于不同机房的网段通常不同，可以根据网段条件匹配来实现地址过滤，具体配置策略如下：`app = web*`，`consumerIP = 192.168.1.* => providerIP = 192.168.1.*`。对于机房A中所有的Web前台应用，只访问本机房内部的服务提供者，不跨机房调用。

还有一种解决策略，就是虚拟分组：将整个集群系统的服务提供者（跨机房）逻辑分成若干个组，某个消费者只访问一个虚拟分组的服务提供者，防止跨组服务调用。如果是多机房部署，虚拟分组可以对应 1 个机房；如果是单个机房，则虚拟分组可以对应 1 个机架。通过拆分和分组，防止某个消费者看到集群中所有的服务提供者，既可以减少竞争提升性能，也可以增强故障隔离能力。

需要指出的是，当某个机房发生大面积宕机或者服务提供者无法正常工作时，需要跨机房访问其他健康的服务提供者，防止某个机房故障导致业务中断。

6.8 总结

服务路由功能是分布式服务框架的重要特性，作为基础平台，既要内置丰富的路由策略，同时还要具备灵活的扩展能力，方便用户做二次定制。

第7章

集群容错

集群服务调用失败后，服务框架需要能够在底层自动容错，容错策略很多，分别适用于不同场景。本章将对集群容错的功能和设计进行详细讲解。

7.1 集群容错场景

在分布式服务框架中，业务消费者不需要了解服务提供者的具体位置，它发起的服务调用请求也不包含服务提供者具体地址信息。因此，某个服务提供者是否可用对消费者而言无关紧要，最终的服务调用成功才是最重要的。

经过服务路由之后，选定某个服务提供者进行远程服务调用，但是服务调用可能会出错，下面我们就对可能的故障场景进行分析。

7.1.1 通信链路故障

这里的链路指的是消费者和服务提供者之间的链路（通常为长连接），可能导致链路中断的原因有：

- 1) 通信过程中，对方突然宕机导致链路中断。
- 2) 通信过程中，对方因为解码失败等原因 Rest 掉连接，导致链路中断。
- 3) 通信过程中，消费者 write SocketChannel 发生 IOException 导致链路中断。
- 4) 通信过程中，消费者 read SocketChannel 发生 IOException 导致链路中断。
- 5) 通信双方因为心跳超时，主动 close SocketChannel 导致链路中断。
- 6) 通信过程中，网络发生闪断故障。
- 7) 通信过程中，交换机异常导致链路中断。
- 8) 通信过程中，消费者或者服务提供者因为长时间 Full GC 导致链路中断。

无论何种原因导致的链路中断，最终都会导致本次服务调用失败。

7.1.2 服务端超时

当服务端无法在指定的时间内返回应答给客户端，就会发生超时，导致超时的原因主要有：

- 1) 服务端的 I/O 线程没有及时从网络中读取客户端请求消息，导致该问题的原因通常是 I/O 线程被意外阻塞或者执行长周期操作。
- 2) 服务端业务处理缓慢，或者被长时间阻塞，例如查询数据库，由于没有索引导致全表查询，耗时较长。
- 3) 服务端发生长时间 Full GC，导致所有业务线程暂停运行，无法及时返回应答给客户端。

7.1.3 服务端调用失败

有时会发生服务端调用失败，导致服务端调用失败的原因主要有如下几种：

- 1) 服务端解码失败，会返回消息解码失败异常。
- 2) 服务端发生动态流控，返回流控异常。
- 3) 服务端消息队列积压率超过最大阈值，返回系统拥塞异常。
- 4) 访问权限校验失败，返回权限相关异常。
- 5) 违反 SLA 策略，返回 SLA 控制相关异常。
- 6) 其他系统异常。

需要指出的是，服务调用异常不包括业务层面的处理异常，例如数据库操作异常、用户记录不存在异常等。

7.2 容错策略

服务不同，容错策略往往也不同。下面我们看下集群容错和服务路由的关系，如图 7-1 所示。

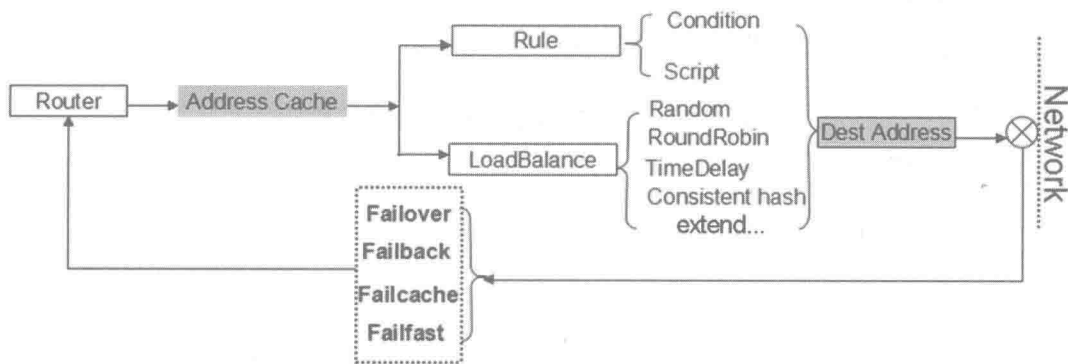


图 7-1 集群容错和服务路由的关系

消费者根据配置的路由策略选择某个目标地址之后，发起远程服务调用，在此期间如果发生了远程服务调用异常，则需要服务框架进行集群容错，重新进行选路和调用。集群容错是系统自动执行的，上层用户并不需要关心底层的服务调用过程。

7.2.1 失败自动切换（Failover）

服务调用失败自动切换策略指的是当发生 RPC 调用异常时，重新选路，查找下一个可用的服务提供者。

服务发布的时候，可以指定服务的集群容错策略。消费者可以覆盖服务提供者的通用配置，实现个性化的容错策略。

Failover 策略的设计思路如下：消费者路由操作完成之后，获得目标地址，调用通信框架的消息发送接口发送请求，监听服务端应答。如果返回的结果是 RPC 调用异常（超时、流控、解码失败等系统异常），根据消费者集群容错的策略进行容错路由，如果是

Failover, 则重新返回到路由 Handler 的入口, 从路由节点继续执行。选路完成之后, 对目标地址进行比对, 防止重新路由到故障服务节点, 过滤掉上次的故障服务提供者之后, 调用通信框架的消息发送接口发送请求消息。

分布式服务框架提供 Failover 容错策略, 但是用户在使用时需要自己保证用对地方, 下面对 Failover 策略的应用场景进行总结:

- ◎ 读操作, 因为通常它是幂等的。
- ◎ 幂等性服务, 保证调用 1 次与 N 次效果相同。

需要特别指出的是, 失败重试会增加服务调用时延, 因此框架必须对失败重试的最大次数做限制, 通常默认为 3, 防止无限制重试导致服务调用时延不可控。

7.2.2 失败通知 (Failback)

在很多业务场景中, 消费者需要能够获取到服务调用失败的具体信息, 通过对失败错误码等异常信息的判断, 决定后续的执行策略, 例如非幂等性的服务调用。

Failback 的设计方案如下: 服务框架获取到服务提供者返回的 RPC 异常响应之后, 根据策略进行容错。如果是 Failback 模式, 则不再重试其他服务提供者, 而是将 RPC 异常通知给消费者, 由消费者捕获异常进行后续处理。

7.2.3 失败缓存 (Failcache)

Failcache 策略是失败自动恢复的一种, 在实际项目中它的应用场景如下:

- ◎ 服务有状态路由, 必须定点发送到指定的服务提供者。当发生链路中断、流控等服务暂时不可用时, 服务框架将消息临时缓存起来, 等待周期 T , 重新发送, 直到服务提供者能够正常处理该消息。
- ◎ 对时延要求不敏感的服务。系统服务调用失败, 通常是链路暂时不可用、服务流

控、GC 挂住服务提供者进程等，这种失败不是永久性的失败，它的恢复是可预期的。如果消费者对服务调用时延不敏感，可以考虑采用自动恢复模式，即先缓存，再等待，最后重试。

- ◎ 通知类服务。例如通知粉丝积分增长、记录接口日志等，对服务调用的实时性要求不高，可以容忍自动恢复带来的时延增加。

为了保证可靠性，Failcache 策略在设计的时候需要考虑如下几个要素：

- ◎ 缓存时间、缓存对象上限数等需要做出限制，防止内存溢出。
- ◎ 缓存淘汰算法的选择，是否支持用户配置。
- ◎ 定时重试的周期 T、重试的最大次数等需要做出限制并支持用户指定。

重试达到最大上限仍失败，需要丢弃消息，记录异常日志。

7.2.4 快速失败（Failfast）

在业务高峰期，对于一些非核心的服务，希望只调用一次，失败也不再重试，为重要的核心服务节约宝贵的运行资源。此时，快速失败是个不错的选择。

快速失败策略的设计比较简单，获取到服务调用异常之后，直接忽略异常，记录异常日志。

7.2.5 容错策略扩展

无论服务框架默认支持多少种容错策略，业务在实际使用过程中一定会有不适应的地方。通过开放容错策略接口的方式，可以支持用户自定义扩展容错策略。

在集群容错设计的时候，需要考虑扩展性，主要从以下几个方面进行设计：

- 1) 容错接口的开放。

- 2) 屏蔽底层细节，用户定制简单。
- 3) 配置应该天生支持扩展，不要让用户扩展服务框架 Schema。

7.3 总结

集群容错从功能上看很简单，设计也并不复杂，但是该特性却非常重要，相比于传统的 RPC 框架，分布式服务框架让用户开发变得更简单，体验也更好。从功能上看，服务框架需要提供更丰富、更细粒度的功能和扩展点，这就是它相比于传统 RPC 框架最大的优势。

第 8 章

服务调用

除了常用的同步服务调用之外，分布式服务框架还需要支持其他几种形式的服务调用，本章将对这些调用方式进行详细讲解。

8.1 几个误区

由于惯性思维，很多人会将传统 MVC 架构或者 RPC 框架的做法带入到分布式服务框架的架构设计中，其中有些思想存在误区，或者已经不合时宜，它们会破坏分布式服务框架的架构品质，本节将对这些误区进行纠正。

8.1.1 NIO 就是异步服务

实际上，通信框架基于 NIO 实现，并不意味着服务框架就支持异步服务调用了，两者本质上不是同一个层面的事情。

在分布式服务框架中，引入 NIO 带来的好处是显而易见的，各种 I/O 对比如表 8-1 所示。

表 8-1 几种 I/O 模型的功能和特性对比

	同步阻塞 I/O (BIO)	伪异步 I/O	非阻塞 I/O (NIO)	异步 I/O (AIO)
客户端个数：I/O 线程	1:1	$M:N$ (其中 M 可以大于 N)	$M:1$ (1 个 I/O 线程处理多个客户端连接)	$M:0$ (不需要启动额外的 I/O 线程，被动回调)
I/O 类型 (阻塞)	阻塞 I/O	阻塞 I/O	非阻塞 I/O	非阻塞 I/O
I/O 类型 (同步)	同步 I/O	同步 I/O	同步 I/O (I/O 多路复用)	异步 I/O
API 使用难度	简单	简单	非常复杂	复杂
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高

引入 NIO 的优点归纳如下。

- ◎ 所有的 I/O 操作都是非阻塞的，避免有限的 I/O 线程因为网络、对方处理慢等原因被阻塞。
- ◎ 多路复用的 Reactor 线程模型：基于 Linux 的 `epoll` 和 `Selector`，一个 I/O 线程可以并行处理成百上千条链路，解决了传统同步 I/O 通信线程膨胀的问题。

NIO 只解决了通信层面的异步问题，跟服务调用的异步没有必然关系，也就是说，即便采用传统的 BIO 通信，依然可以实现异步服务调用，只不过通信效率和可靠性比较差而已。

下面我们对异步服务调用和通信框架的关系进行说明，如图 8-1 所示。

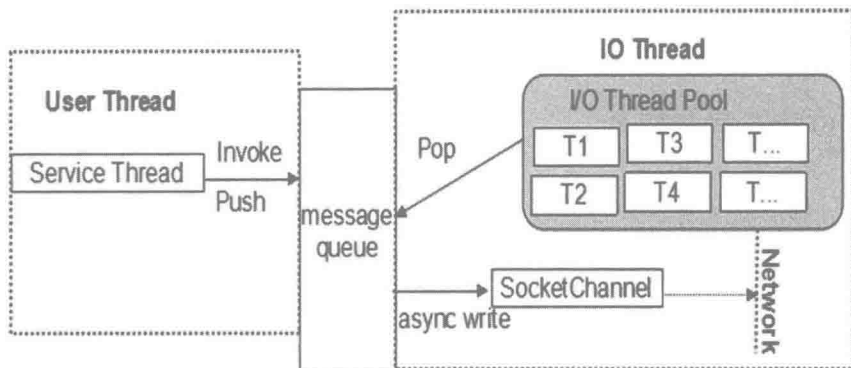


图 8-1 服务调用和通信框架关系

用户发起远程服务调用之后，经历层层业务逻辑处理、消息编码，最终序列化后的消息会被放入到通信框架的消息队列中。业务线程可以选择同步等待、也可以选择直接返回，通过消息队列的方式实现业务层和通信层的分离是比较成熟、典型的做法，现代的 RPC 框架或者 Web 服务器很少直接使用业务线程进行网络读写。

通过图 8-1 我们可以看出，采用 NIO 还是 BIO 对上层的业务是不可见的，双方的汇聚点就是消息队列，在 Java 实现中它通常就是个 Queue。业务线程将消息放入到发送队列中，可以选择主动等待或者立即返回，跟通信框架是否是 NIO 没有任何关系。

8.1.2 服务调用天生就是同步的

有一种比较流行的观点认为 RPC 调用天生就是同步的，请求-应答模式同样是服务调用的主流模式，因此服务调用天生也是同步的。

要理解上面的观点，首先我们需要对服务调用自身的模式进行分析，服务调用抛开技

术实现不谈，它主要有两种模式。

- ◎ OneWay 模式：只有请求，没有应答，例如通知消息。
- ◎ 请求-应答模式：一请求，一应答的模式，这种模式最常用。

OneWay 模式的调用示意图如图 8-2 所示。



图 8-2 OneWay 服务调用

请求-应答模式最常用，例如 HTTP 协议，就是典型的请求-应答模式，如图 8-3 所示。

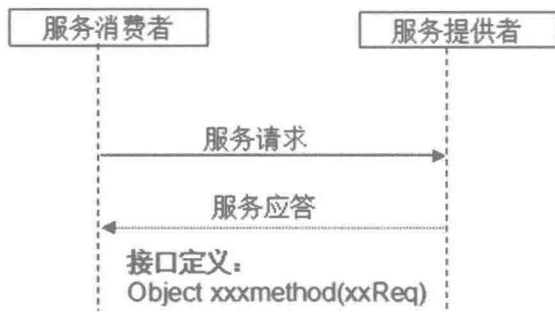


图 8-3 请求-应答模式服务调用

OneWay 模式的服务调用由于不需要返回应答，因此很容易被设计成异步的：消费者发起远程服务调用之后，立即返回，不需要同步阻塞等待应答。

对于请求-应答模式，一般的观点都认为消费者必需要等待服务端响应，拿到结果之后才能返回，否则结果从哪里取？即便业务线程不阻塞，没有获取到结果流程还是无法继

续执行下去。

从逻辑上看，上述观点没有问题。但实际上，同步阻塞等待应答并非是唯一的技术选择，我们也可以利用 Java 的 Future-Listener 机制来实现异步服务调用。从业务角度看，它的效果与同步等待等价，但是从技术层面看，却是个很大的进步，它可以保证业务线程在不同步阻塞的情况下实现同步等待的效果，服务执行效率更高。

8.1.3 异步服务调用性能更高

通常在实验室环境中测试，由于网络时延小、模拟业务又通常比较简单，所以异步服务调用并不一定性能更高，但在生产环境中，异步服务调用往往性能更高、可靠性也更好。主要原因是网络环境相对恶劣、真实的服务调用耗时更多等，这种恶劣的运行环境正好可以发挥异步服务调用的优势。

8.2 服务调用方式

服务框架支持多种形式的服务调用，本节将对这几种服务调用的原理和设计进行讲解。

8.2.1 同步服务调用

同步服务调用是最常用的一种服务调用方式，它的工作原理和使用都非常简单，分布式服务框架默认都需要支持这种调用形式。

它的工作原理如下：客户端发起远程服务调用请求，用户线程完成消息序列化之后，将消息投递到通信框架，然后同步阻塞，等待通信线程发送请求并接收到应答之后，唤醒同步等待的用户线程，用户线程获取到应答之后返回。

它的工作原理图如图 8-4 所示。

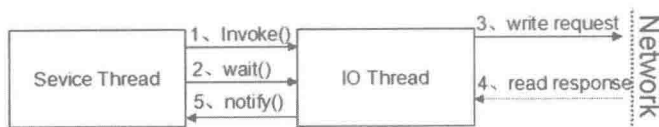


图 8-4 同步服务调用

- 1) 消费者调用服务端发布的接口，接口调用由分布式服务框架包装成动态代理，发起远程服务调用。
- 2) 消费者线程调用通信框架的消息发送接口之后，直接或者间接调用 `wait()` 方法，同步阻塞等待应答。
- 3) 通信框架的 I/O 线程通过网络将请求消息发送给服务端。
- 4) 服务端返回应答消息给消费者，由通信框架负责应答消息的反序列化。
- 5) I/O 线程获取到应答消息之后，根据消息上下文找到之前同步阻塞的业务线程，`notify()` 阻塞的业务线程，返回应答给消费者，完成服务调用。

为了防止服务端长时间不返回应答消息导致客户端用户线程被挂死，用户线程等待的时候需要设置超时时间，这个超时时间与服务端或者客户端配置的超时时间对应。

8.2.2 异步服务调用

基于 JDK 的 `Future` 机制，可以非常方便地实现异步服务调用，JDK 的 `Future` 接口定义如图 8-5 所示。

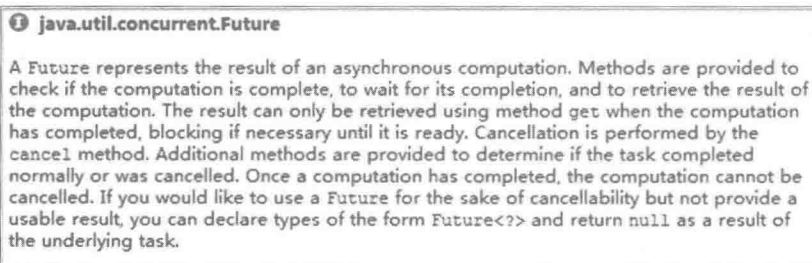


图 8-5 JDK Future Doc

JDK 原生的 Future 主要用于异步操作，它代表了异步操作的执行结果，用户可以通过调用它的 get 方法获取结果。如果当前操作没有执行完，get 操作将阻塞调用线程。

在实际项目中，往往会扩展 JDK 的 Future，提供 Future-Listener 机制，它支持主动获取和被动异步回调通知两种模式，适用于不同的业务场景。

以 Netty 的 Future 接口定义为例，新增了监听器管理接口，监听器主要用于异步通知回调，它的接口定义如图 8-6 所示。

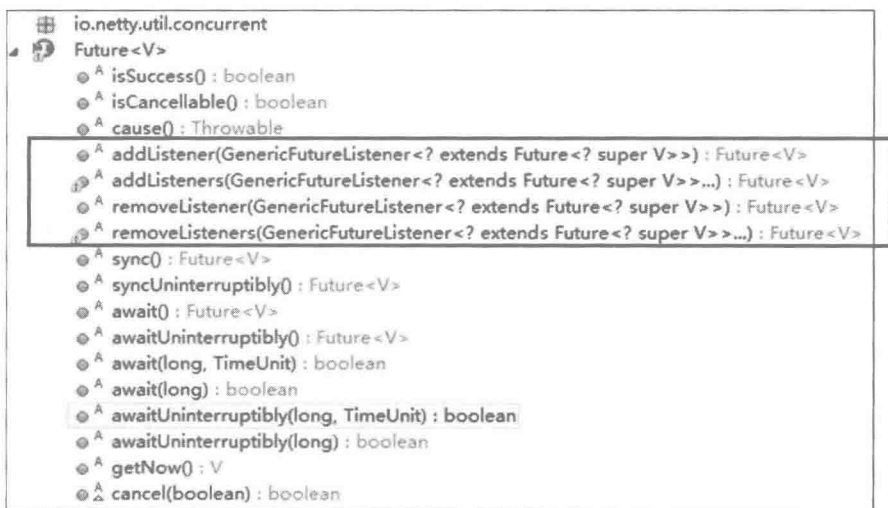


图 8-6 Future-Listener 机制

异步服务调用的工作原理如图 8-7 所示。

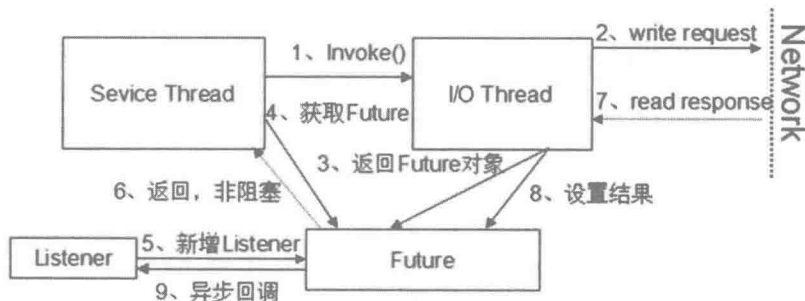


图 8-7 异步服务调用原理图

异步服务调用的工作流程如下：

- 1) 消费者调用服务端发布的接口，接口调用由分布式服务框架包装成动态代理，发起远程服务调用。
- 2) 通信框架异步发送请求消息，如果没有发生 I/O 异常，返回。
- 3) 请求消息发送成功后，I/O 线程构造 Future 对象，设置到 RPC 上下文中。
- 4) 用户线程通过 RPC 上下文获取 Future 对象。
- 5) 构造 Listener 对象，将其添加到 Future 中，用于服务端应答异步回调通知。
- 6) 用户线程返回，不阻塞等待应答。
- 7) 服务端返回应答消息，通信框架负责反序列化等。
- 8) I/O 线程将应答设置到 Future 对象的操作结果中。
- 9) Future 对象扫描注册的监听器列表，循环调用监听器的 operationComplete 方法，将结果通知给监听器，监听器获取到结果之后，继续后续业务逻辑的执行，异步服务调用结束。

需要指出的是，还有另外一种异步服务调用形式，就是不添加 Listener，用户连续发起 N 次服务调用，然后依次从 RPC 上下文中获取 Future 对象，最终再主动 get 结果，业务线程阻塞，相比于老的同步服务调用，它的阻塞时间更短，其工作原理如图 8-8 所示。

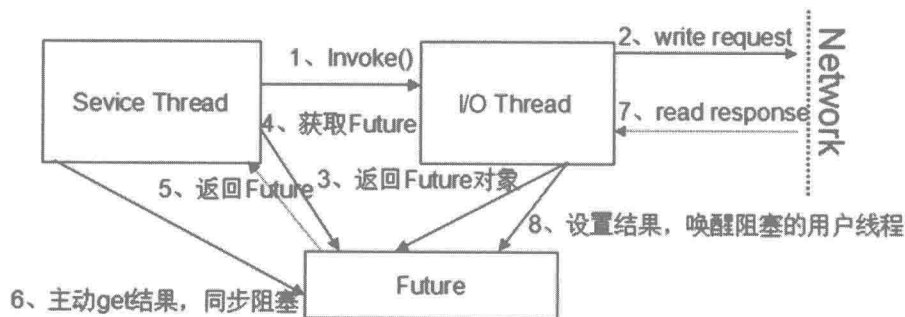


图 8-8 异步服务调用主动 get 结果原理图

异步服务调用的代码示例如下：

```
xxxService1.xxxMethod(Req);  
Future f1 = RpcContext.getContext().getFuture();  
xxxService2.xxxMethod(Req);  
Future f2 = RpcContext.getContext().getFuture();  
Object xxResult1 = f1.get(3000);  
Object xxResult2 = f2.get(3000); }
```

假如 xxxService1 和 xxxService2 发布成异步服务，则调用 xxxMethod 之后当前业务线程不阻塞，立即返回 null。用户不能直接使用它的返回值，而是通过当前线程上下文 RPCContext 获取异步操作结果 Future。获取到 Future 之后继续发起其他异步服务调用，然后获取另一个 Future……最后，通过 Future 的 get 方法集中获取结果。无论有多少个 Future，采用此种方式用户线程最长阻塞时间为耗时最长的 Future，即 $T = \text{Max } t(\text{future1} \dots \text{N})$ 。如果采用同步服务调用，用户线程的阻塞时间 $T = t(\text{future1}) + t(\text{future2}) + \dots + t(\text{futureN})$ 。

异步服务调用相比于同步服务调用有两个优点：

- ◎ 化串行为并行，提升服务调用效率，减少业务线程阻塞时间。
- ◎ 化同步为异步，避免业务线程阻塞。

串行到并行的优化原理如图 8-9 所示。

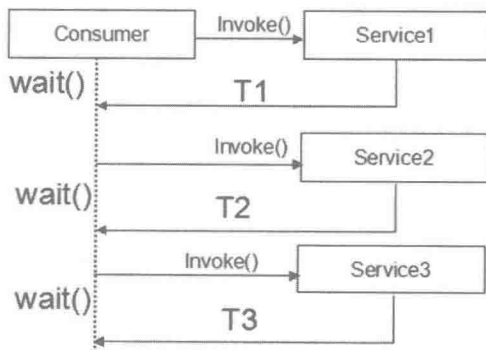


图 8-9 同步调用多个服务场景

由于每次服务调用都是同步阻塞，三个服务调用总耗时为 $T = T1 + T2 + T3$ 。下面我们看下采用异步服务调用之后的优化效果，如图 8-10 所示。

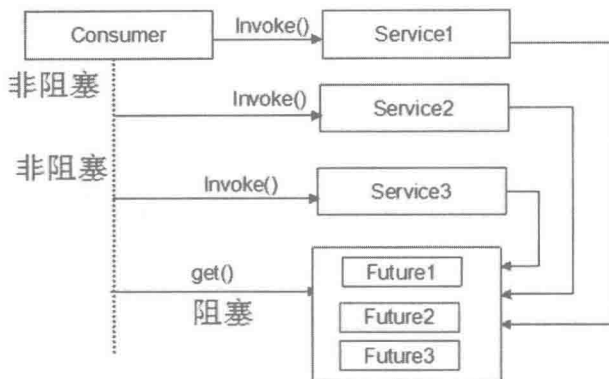


图 8-10 异步多服务调用场景

采用异步服务调用模式，最后调用三个服务异步操作结果 Future 的 get 方法同步等待应答，它的总执行时间 $T = \text{Max}(T1, T2, T3)$ ，相比于同步服务调用，性能提升效果非常明显。

第二种基于 Future-Listener 的纯异步服务调用，它的代码示例如下：

```
xxxService1.xxxMethod(Req);
Future f1 = RpcContext.getContext().getFuture();
Listener l = new xxxListener();
f1.addListener(l);
.....后续代码省略 }
```

基于 Future-Listener 的异步服务调用相比于 Future-get 模式更好，但是在实际使用中有一定的局限性，具体的使用限制留给读者自己思考。

8.2.3 并行服务调用

在大多数业务应用中，服务总是被串行地调用和执行，例如 A 调用 B 服务，B 服务调

用 C 服务，最后形成一个串行的服务调用链：A→B 服务→C 服务→……

串行服务调用比较简单，但在一些业务场景中，需要采用并行服务调用来降低 E2E 的时延：

- ◎ 多个服务之间逻辑上不存在互相依赖关系，执行先后顺序没有严格的要求，逻辑上可以被并行执行。
- ◎ 长流程业务，调用多个服务，对时延比较敏感，其中有部分服务逻辑上无上下文关联，可以被并行调用。

并行服务调用的目标主要有两个：

- 1) 降低业务 E2E 时延。
- 2) 提升整个系统的吞吐量。

我们以手游购买道具流程为例，对并行服务调用进行说明，如图 8-11 所示。

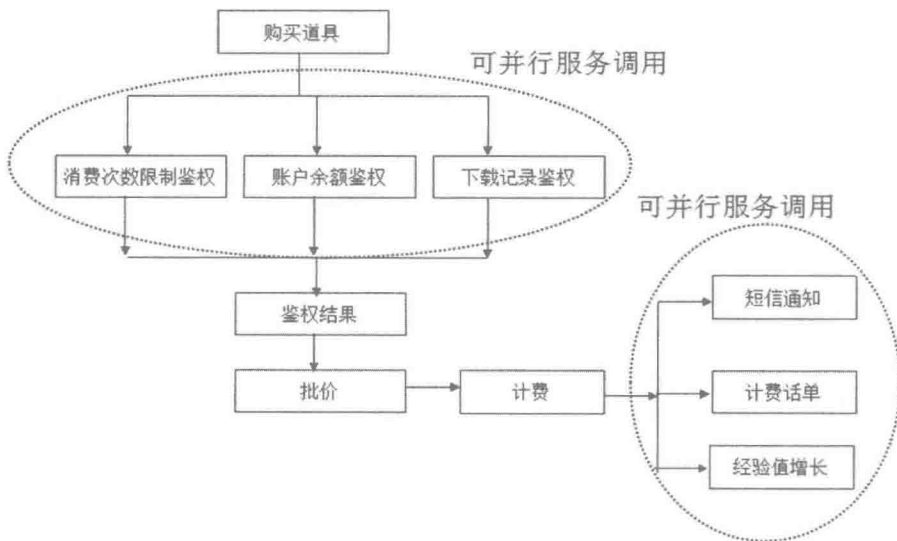


图 8-11 购买道具并行服务调用流程图

在购买道具时，三个鉴权流程实际可以并行执行，最终执行结果做个 Join 即可。如果

采用传统的串行服务调用，耗时将是三个鉴权服务时延之和，显然是没有必要的。计费之后的通知类服务亦如此（注意：通知服务也可以使用 MQ 做订阅/发布），单个服务的串行调用会导致购买道具时延比较长，影响游戏体验。

要解决串行调用效率低的问题，有两个解决对策：

◎ 异步服务调用。

◎ 并行服务调用。

在上一节中已经对异步服务调用进行了讲解，下面我们对并行服务调用进行详细介绍。

并行服务调用的原理：一次同时发起多个服务调用，先做流程的 Fork，再利用 Future 等主动等待获取结果，进行结果汇聚（Join）。实现并行服务调用的几种技术方案：

◎ JDK 7 的 Fork/Join，可以实现子任务的并行执行和结果汇聚。

◎ BPM 的 Parallel Gateway。

◎ 批量串行服务调用。

JDK7 的 Fork/Join 底层会开启多个线程来分解任务，在服务框架中使用会导致依赖线程上下文传递的变量丢失、线程膨胀不可控等问题，因此在并行服务调用时不适合使用 JDK 的 Fork/Join 并行执行框架。

BPM 流程引擎支持并行流程（子流程）调用，它的执行示意图如图 8-12 所示。

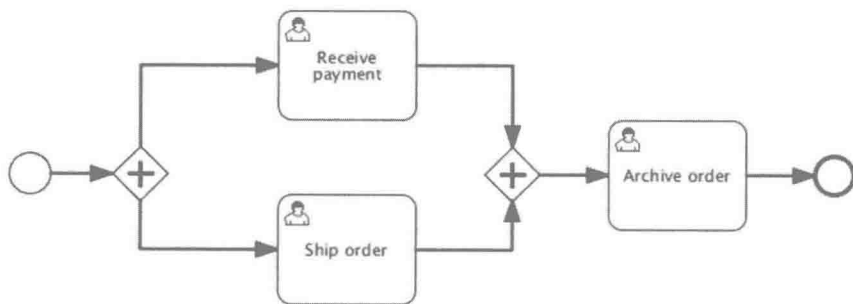


图 8-12 BPM Parallel Gateway 工作流程

Parallel Gateway（并行网关）能在一个流程里用来对并发建模。在一个流程模型里引入并发最直接的网关就是并行网关（Parallel Gateway），它允许 Fork 执行多个路径，或者 Join 多个执行的到达路径。

并行网关的功能基于即将到达的和即将离开的流程顺序流。

◎ Fork: 所有即将离开的顺序流将以并行方式，为每个顺序流程建立一个并发执行器。

◎ Join: 所有的并发执行到达并行网关，在网关里面等待直到每个来到的顺序流的执行到达，条件满足后流程继续通过合并网关。

从技术上看，不同的 BPM 流程引擎具体实现细节也不同，但大多数都支持：通过创建子线程的方式实现并行调用、通过批量调用的方式实现伪异步并行调用。对于服务框架而言，BPM Parallel Gateway 的功能可以满足需求，但是为了并行服务调用引入 BPM 流程引擎显然是得不偿失，我们可以参考 Parallel Gateway 的伪异步并行调用来实现服务框架的并行服务调用。

下面我们对批量串行服务调用实现并行服务调用的原理进行讲解，如图 8-13 所示。

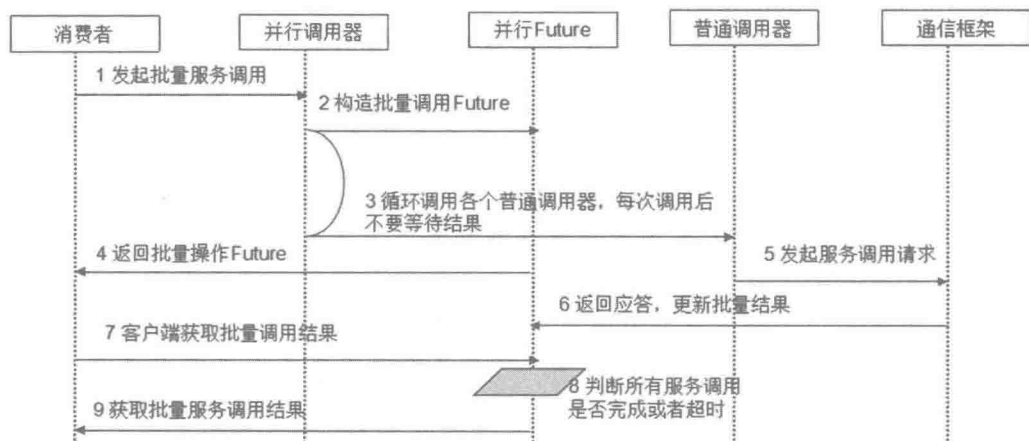


图 8-13 批量服务调用原理图

1) 服务框架提供批量服务调用接口供消费者使用，它的定义样例如下：

```
ParallelService.invoke(serviceName [], methodName[], args [])
```


- 2) 平台的并行服务调用器创建并行 Future, 缓存批量服务调用上下文信息。
- 3) 并行服务调用器循环调用普通的 Invoker, 通过循环的方式执行单个服务调用, 获取到单个服务的 Future 之后设置到 Parallel Future 中。
- 4) 返回 Parallel Future 给消费者。
- 5) 普通 Invoker 调用通信框架的消息发送接口, 发起远程服务调用。
- 6) 服务端返回应答, 通信框架对报文做反序列化, 转换成业务对象更新 Parallel Future 的结果列表。
- 7) 消费者调用 Parallel Future 的 get(timeout)方法, 同步阻塞, 等待所有结果全部返回。
- 8) Parallel Future 通过对结果集进行判断, 看所有服务调用是否都已经完成 (包括成功、失败和异常)。
- 9) 所有批量服务调用结果都已经返回, Notify 消费者线程, 消费者获取到结果列表, 完成批量服务调用, 流程继续执行。

通过批量服务调用+ Future 机制, 我们实现了并行服务调用, 而且没有创建新的线程, 用户不用担心依赖线程上下文的功能出异常。该方案唯一的缺点就是用户需要调用平台提供的并行服务调用接口, 这个会导致 API 层面的依赖, 对于努力构建零依赖的服务框架而言不是最优的选择。但事实上完全的零依赖是不存在的, 即便 100% XML 配置也是一种配置依赖, 所以在设计过程中要能够识别并抓主要矛盾点, 做到有所舍, 否则设计工作将步履维艰。

8.2.4 泛化调用

泛化调用通常包含两种模式: 泛化引用和泛化实现。泛化引用主要用于客户端没有 API 接口及数据模型的场景, 参数及返回值中的所有 POJO 均用 Map 表示, 通常用于框架集成, 比如实现一个通用的服务测试框架。泛化实现主要用于服务器端没有 API 接口及数据模型的场景, 参数及返回值中的所有 POJO 均用 Map 表示, 通常用于框架集成, 比如实

现一个通用的远程服务 Mock 框架。

泛化调用的设计要点如下。

- 1) 分布式服务框架提供泛化接口，供服务提供者实现和消费者引用，它的参考定义如下：

```
public interface GenService{  
    Object invoke(String methodName, String[] paramTypes, Object [] args);  
}
```

- 2) 消费者如果引用泛化接口，则直接将请求参数转换成 Map，应答消息也自动转换成 Map。
- 3) 服务提供者如果使用泛化实现发布服务，则自动将请求参数转换成 Map，调用 GenService 的泛化实现类，应答消息自动包装成 Map 返回。

泛化调用由于比较灵活，没有服务契约，因此在实际项目中慎用，它通常用于测试集成、系统上线之后的回声测试等。

8.3 最佳实践

服务框架支持多种服务调用方式，在实现项目中如何选择呢？建议从以下几个角度进行考虑。

- ◎ 降低业务 E2E 时延：业务调用链是否太长、某些服务是否不太可靠，需要对服务调用流程进行梳理，看是否可以通过并行服务调用来提升调用效率，降低服务调用时延。
- ◎ 可靠性角度：某些业务调用链上的关键服务不太可靠，一旦出故障会导致大量线程资源被挂住，可以考虑使用异步服务调用防止故障扩散。

- ◎ 业务场景：对于测试，不想为每个测试用例都开发一个服务接口，能否做一个通用的测试框架，客户端通过 Map 等泛容器实现通用服务调用。这种场景则可以使用服务泛化调用。
- ◎ 传统的 RPC 调用：服务调用比较简单，对时延要求不高的场景，则可以考虑同步服务调用。

8.4 总结

服务框架往往支持多种形式的调用，我们在设计服务调用时，需要充分考虑用户的使用习惯以及业务面临的主要挑战，在矛盾中做出平衡和取舍，这是一个优秀架构师的基本功。

第 9 章

服务注册中心

对于服务提供者，它需要发布服务，由于应用系统的复杂性，服务的数量、类型不断膨胀；对于服务消费者，它最关心的是如何获取到它所需要的服务。对于服务提供方和服务消费方来说，它们还有可能兼具这两种角色：既需要提供服务，又需要消费服务。

如何有效地管理服务订阅/发布，避免硬编码地址信息是分布式服务框架需要解决的一个问题。通过将服务统一管理起来，可以有效地优化内部应用对服务发布/使用的流程和管理，服务注册中心就是专门用来管理服务订阅/发布的配置管理节点。

9.1 几个概念

在设计服务注册中心之前，我们首先熟悉一下和服务注册中心相关的几个角色。

9.1.1 服务提供者

服务提供者是发布服务的提供方，它通常就是一个普通的 Java 实现类。它的配置示例如下所示：

```
<bean id = "xxxService" class="edu.neu.xxxServiceImpl" />
<xxx:service interface="edu.neu.xxxService" ref="xxxService" />
```

9.1.2 服务消费者

服务消费者是调用远程服务的消费方，它可能是个简单的客户端、Web 前台，也可能嵌套在某个服务的内部。消费者的配置示例如下：

```
<xxx:reference id="xxxService" interface="edu.neu.xxxService" />
<bean class="edu.neu.xxxAction" init-method="start">
    <property name="xxxService" ref="xxxService" />
</bean>
```

9.1.3 服务注册中心

服务注册中心是分布式服务框架的目录服务器，相比于传统的目录服务器，它有如下几个特点。

- 1) 高 HA：支持数据持久化、支持集群。
- 2) 数据一致性问题：集群中所有的客户端应该看到同一份数据，不能出现读或者写

数据不一致。

- 3) 数据变更主动推送：当注册中心的数据发生变更时（增加、删除、修改）需要能够及时将变化的数据通知给客户端。

9.2 关键功能特性设计

当服务越来越多时，服务 URL 配置管理变得非常困难，F5 等硬件负载均衡器的单点压力也越来越大。此时需要一个服务注册中心，动态地注册和发现服务，使服务的位置透明。

服务注册中心的工作原理如图 9-1 所示。

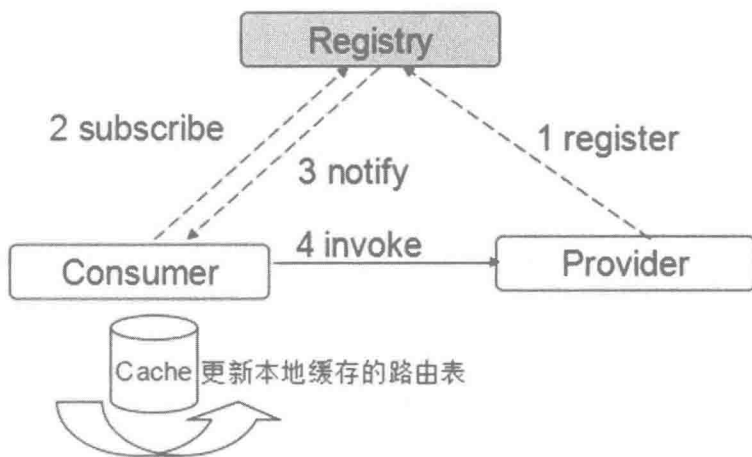


图 9-1 服务注册中心工作原理图

- 1) 服务提供者在启动时，根据服务发布文件中配置的服务发布信息向注册中心注册自己提供的服务。
- 2) 服务消费者在启动时，根据消费者配置文件中配置的服务消费信息向注册中心订阅自己所需的服务，消费者刷新本地缓存的路由表。

- 3) 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心主动推送变更数据给消费者，消费者刷本地缓存的路由表。
- 4) 服务消费者从本地缓存的服务提供者地址列表中，基于负载均衡算法选择一台服务提供者进行调用。

9.2.1 支持对等集群

服务注册中心需要支持对等集群，如图 9-2 所示，其中某一个或者多个服务注册中心进程宕机，不会导致服务注册中心集群功能不可用。

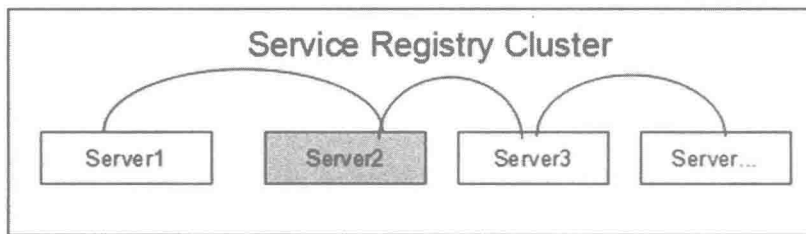


图 9-2 服务注册中心集群

对于客户端，无论服务注册中心集群配置多少个进程，客户端只需要连接其中某一个即可（服务端之间自己进行数据同步），它的组网如图 9-3 所示。

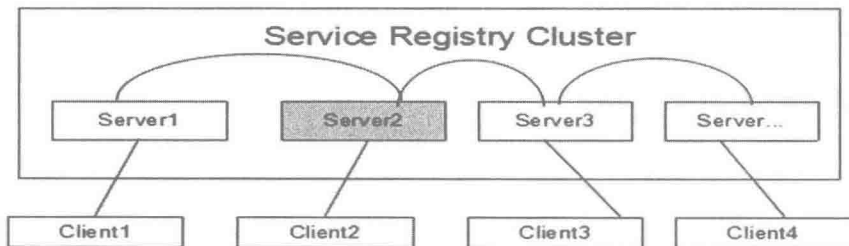


图 9-3 服务注册中心集群组网

9.2.2 提供 CRUD 接口

客户端连接服务注册中心之后，需要能够对服务注册中心的数据进行操作，通常需要用如下几种操作接口。

- ◎ 查询接口：查询系统当前发布的服务信息和订阅的消费者信息。
- ◎ 修改接口：修改已经发布的服务属性或者消费者属性信息，通常用于运行态的服务治理。
- ◎ 新增接口：发布或者订阅新的服务。
- ◎ 删除接口：去注册已经发布的服务，或者消费者取消订阅关系。

9.2.3 安全加固

服务注册中心需要进行安全加固，安全加固主要涉及两部分：

- ◎ 链路的安全性。
- ◎ 数据的安全性。

链路的安全性指的是服务注册中心对客户端连接进行安全认证，认证策略非常多，最简单的就是基于 IP 地址的黑名单校验，更加复杂的有基于用户名+密码的认证，或者基于密钥+数字证书的认证。

认证失败，则关闭链路，拒绝客户端连接，它的工作原理示意图如图 9-4 所示。

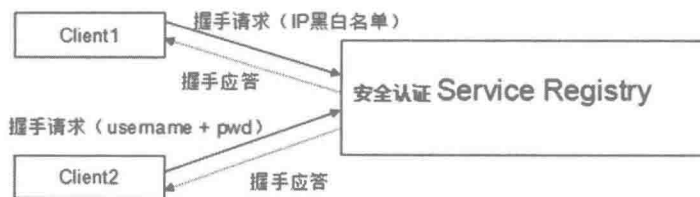


图 9-4 服务注册中心链路安全性检测工作原理

数据的安全性主要针对服务注册中心的数据进行权限控制：

- 1) 非授权客户端既不能读取也不能写入数据。
- 2) 普通运维人员只能读取数据，不能修改数据。
- 3) 管理员既可以读取也可以修改数据。
- 4) 不同的服务目录可以设置不同的访问权限，例如消费者只能查看它所在机房的的服务。

数据安全性的工作原理如图 9-5 所示。

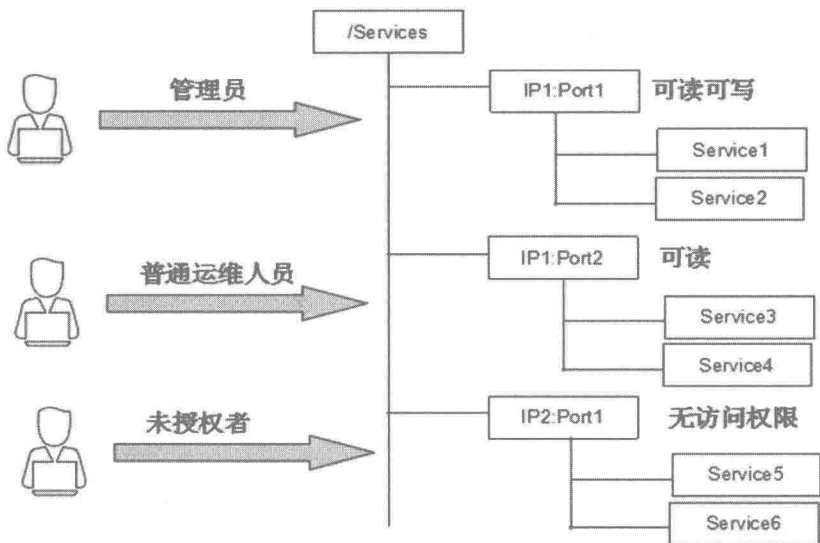


图 9-5 服务注册中心数据安全性的工作原理示意图

9.2.4 订阅发布机制

服务注册中心需要支持服务的订阅发布，对于服务提供者，可以根据服务名等信息动态发布服务；对于消费者，可以根据订阅关系主动获得服务发布者的地址信息等。

订阅发布机制还有一个比较重要的机制就是对变化的监听和主动推送能力：

- ◎ 消费者可以监听一个或者多个服务目录，当目录名称、内容发生变更时，消费者可以实时地获得变更的数据或者变更后的结果信息。
- ◎ 服务提供者可以发布一个或者多个服务，动态修改服务名称、服务内容等，可以主动将修改后的数据或者修改后的结果推送给所有监听此服务目录的消费者。

订阅发布机制具有如下几个优点。

- ◎ 透明化路由：服务提供者和消费者互相解耦，服务提供者位置透明，消费者不需要再硬编码服务提供者地址。
- ◎ 服务健康状态检测：服务注册中心可以实时检测发布服务的质量，如果服务提供者宕机，由服务注册中心实时通知消费者。
- ◎ 弹性伸缩能力（动态发现）：应用在云端部署之后，由于 VM 资源占用率过高，动态伸展出一个新的服务提供者，服务注册中心会将新增的服务提供者地址信息推送给消费者，消费者刷新本地路由表之后可以访问新的服务提供者，实现服务的弹性伸缩。

9.2.5 可靠性

服务注册中心需要支持对等集群，任意一台宕机后，服务都能自动切换到其他正常的服务注册中心。如果服务注册中心全部宕机，只影响新的服务注册、已发布服务的下线，不影响服务的正常运行和调用，消费者可以依靠本地缓存的服务路由表进行路由。

除了服务注册中心自身的可靠性，服务提供者的健康状态检测也由服务注册中心负责检测。服务注册中心通过长连接心跳检测服务提供者的存在。服务提供者宕机，注册中心将立即推送服务下线事件通知消费者，消费者将下线的服务提供者地址从缓存的路由表中删除，新接入的消息将不再路由到故障节点，实现实时故障隔离。

9.3 基于 ZooKeeper 的服务注册中心设计

ZooKeeper 是 Apache Hadoop 的一个子项目，它主要用来解决分布式应用中经常遇到的一些数据管理问题，如统一命名服务、状态同步服务、集群管理、分布式应用统一配置等。

下面我们讲解如何基于 ZooKeeper 来设计服务注册中心。

9.3.1 服务订阅发布流程设计

基于 ZooKeeper 的服务订阅发布流程设计如图 9-6 所示。

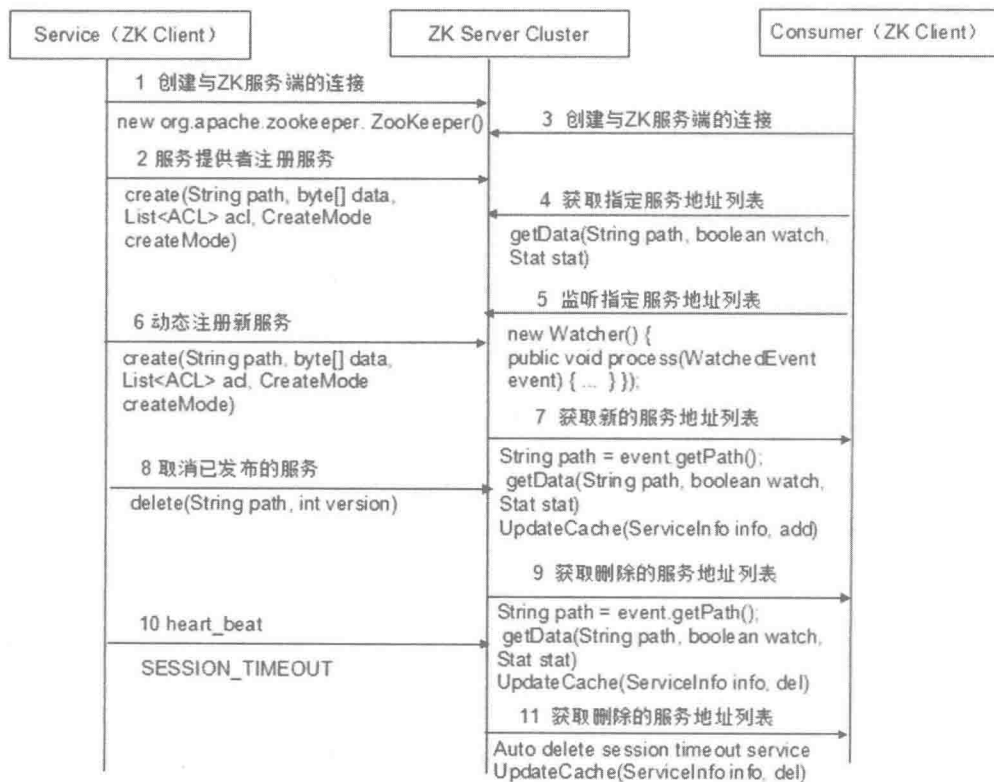


图 9-6 基于 ZooKeeper 的服务注册中心流程设计

第一步, ZooKeeper 客户端通过创建 `org.apache.zookeeper.ZooKeeper` 的一个实例对象连接 ZooKeeper 服务器, 然后调用这个类提供的接口来和服务器交互。

第二步, 根据服务提供者发布的服务列表, 循环调用 `create(String path, byte[] data, List<ACL> acl, createMode)` 接口, 创建目录节点, 同时将服务属性写入目录节点的内容中。`create` 方法用于创建一个给定路径的目录节点, 并给它设置数据。`CreateMode` 标识有四种形式的目录节点, 分别如下。

- ◎ `PERSISTENT`: 持久化目录节点, 这个目录节点存储的数据不会丢失。
- ◎ `PERSISTENT_SEQUENTIAL`: 顺序自动编号的目录节点, 这种目录节点会根据当前已经存在的节点数自动加 1, 然后向客户端返回已经成功创建的目录节点名。
- ◎ `EPHEMERAL`: 临时目录节点, 一旦创建这个节点的客户端与服务器端口的会话超时, 这种节点会被自动删除。
- ◎ `EPHEMERAL_SEQUENTIAL`: 临时自动编号节点。

第三步, 与第一步相同, 不再赘述。

第四步, 消费者根据消费的服务名, 从 ZooKeeper Server 的服务发布目录中查询已经发布的服务地址和属性信息。`getData(String path, boolean watch, Stat stat)` 方法用于获取这个 `path` 对应的目录节点存储的数据, 数据的版本等信息可以通过 `stat` 来指定, 同时还可以设置是否监控这个目录节点数据的状态。获得服务提供者信息之后, 将其更新到本地缓存中, 避免每次服务调用都要实时去 ZooKeeper Server 查询。

第五步, 消费者监听服务提供者目录列表, 当有服务上下线时, 可以获取 ZooKeeper Server 的通知消息。

第六步, 服务提供者调用 `create(String path, byte[] data, List<ACL> acl, CreateMode createMode)` 方法动态注册新的服务。

第七步, 消费者通过监听器 `Watcher` 的 `process(WatchedEvent event)` 方法获取变更的路径信息, 然后调用 `getData(String path, boolean watch, Stat stat)` 接口获取变更的数据, 最后

更新本地缓存的服务路由表。

第八步，服务提供者取消已发布的服务，调用 `delete(String path, int version)` 方法删除 `path` 对应的目录节点，`version` 为 -1 可以匹配任何版本，也就删除了这个目录节点的所有数据。

第九步，消费者接收到变更通知推送消息后，根据删除的路径获取删除的具体服务信息；更新本地缓存的服务路由表，将已经下线的服务从路由表中删除。

第十步，ZK 客户端与服务器的连接会话终止，存储在 ZK 上的所有临时数据与注册的订阅者都被自动移除。

第十一步，消费者获取服务列表被删除的通知消息之后，更新本地缓存的路由表，将发生会话超时的服务提供者从路由表中清除掉，直到对方恢复与 ZooKeeper Server 的连接。

需要指出的是，在实际项目中，消费者和服务提供者的启动顺序是无法控制的，所以步骤一、二和步骤三、四可以互换，对功能无影响。

9.3.2 服务健康状态检测

基于 ZooKeeper 客户端和服务端的长连接和会话超时控制机制，可以非常方便地实现服务健康状态检测。

在 ZooKeeper 中，客户端和服务端建立连接后，会话随之建立，生成一个全局唯一的 Session ID。服务器和客户端之间维持的是一个长连接，在 `SESSION_TIMEOUT` 周期内，服务器会检测与客户端的链路是否正常（客户端定时向服务器发送心跳消息，服务器重置下次 `SESSION_TIMEOUT` 时间）。因此，在正常情况下，Session 会一直有效，并且 ZK 集群所有机器上都保存这个 Session 信息。在出现网络或其他问题的情况下（客户端宕机、网络闪断），如果客户端与之前连接的 ZooKeeper Server 断连了，此时客户端会主动在地址列表中选择新的地址进行连接。

如果 ZK 客户端宕机，或者网络出现故障，超过了 `SESSION_TIMEOUT` 后服务端仍然没有收到客户端的心跳消息，则服务器认为这个 Session 已经结束了。在 ZK 中，很多数

据和状态都是和会话绑定的，一旦会话失效，那么 ZK 就开始清除和这个会话有关的信息，包括这个会话创建的临时节点和注册的所有 Watcher。消费者获取到相关通知消息之后，根据事件类型进行判断就可以得知哪些服务提供者已经宕机或者下线，通过服务名等地址信息更新路由表，即可防止新接入的消息路由到故障节点。

9.3.3 对等集群防止单点故障

ZooKeeper 集群通常由 $2n+1$ 台 Server 组成，每台 Server 都知道彼此的存在。每台 Server 都维护内存状态镜像以及持久化存储的事务日志和快照，对于 $2n+1$ 台 Server，只要有 $n+1$ 台（大多数）Server 可用，整个集群就保持可用。

系统启动时，集群中的 Server 会选举出一台 Server 为 Leader，其他的就作为 Follower。接着由 Follower 来服务 Client 的请求，对于不改变系统一致性状态的读操作，由 Follower 的本地内存数据库直接给 Client 返回结果；对于会改变系统状态的更新操作，则交由 Leader 进行提议投票，超过半数通过后返回结果给 Client。

ZooKeeper 集群管理的核心是原子广播，这个机制保证了各个 Server 之间的数据同步。实现这个机制的协议叫作 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。当服务启动或者在 Leader 崩溃后，Zab 就进入了恢复模式，当 Leader 被选举出来，且大多数 Server 完成了和 Leader 的状态同步之后，恢复模式就结束了。状态同步保证了 Leader 和 Server 具有相同的系统状态。

一旦 Leader 已经和多数的 Follower 进行了状态同步后，它就可以开始广播消息了，即进入广播状态。这时候当一台 Server 加入 ZooKeeper 服务中时，它会在恢复模式下启动，发现 Leader 并和 Leader 进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在广播状态，直到 Leader 崩溃了或者 Leader 失去了大部分的 Follower 支持。

Broadcast 模式类似于分布式事务中的两阶段提交：即 Leader 提起一个决议，由 Follower 进行投票，leader 对投票结果进行计算决定是否通过该决议；如果通过则执行该决议（事务），否则什么也不做。

广播模式需要保证 Proposal 被按顺序处理，因此 ZK 采用了递增的事务 id 号（zxid）来保证。所有的提议（Proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，高 32 位 epoch 用来标识 Leader 关系是否改变，每次一个 Leader 被选出来，它都会有一个新的 epoch；低 32 位用于递增计数。

当 Leader 崩溃或者 Leader 失去大多数的 Follower 时，这时候 ZK 进入恢复模式。恢复模式需要重新选举出一个新的 Leader，让所有的 Server 都恢复到一个正确的状态。

恢复模式下，重新刚从崩溃状态恢复的或者刚启动的 Server 还会从磁盘快照中恢复数据和会话信息。选完 Leader 以后，ZK 就进入状态同步过程：

- 1) Leader 开始等待 Server 连接。
- 2) Follower 连接 Leader，将最大的 zxid 发送给 Leader。
- 3) Leader 根据 Follower 的 zxid 确定同步点。
- 4) 完成同步后通知 Follower 已经成为 uptodate 状态。
- 5) Follower 收到 uptodate 消息后，又可以重新接受 Client 的请求进行服务。

基于 ZooKeeper 服务注册中心的工作原理如图 9-7 所示。

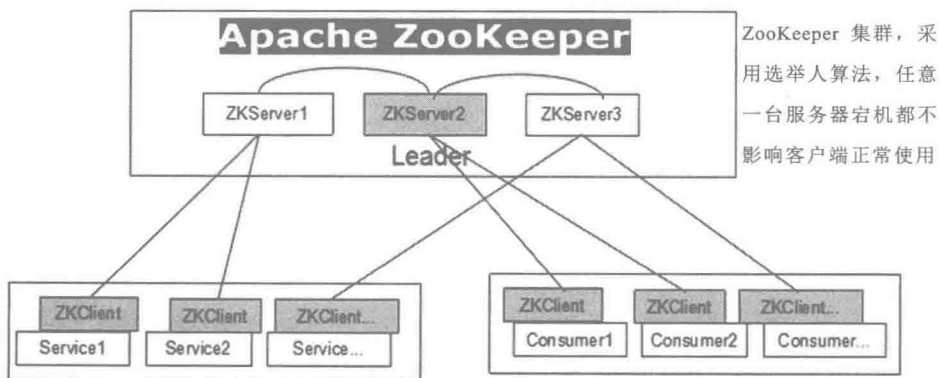


图 9-7 基于 ZooKeeper 集群的服务注册中心工作原理

9.3.4 变更通知机制

服务提供者将服务注册信息保存在 ZooKeeper 的某个服务目录节点中，消费者监控服务配置信息的状态，一旦配置信息发生变化，集群中的每个消费者实例就会收到 ZooKeeper 的通知，然后从 ZooKeeper Server 获取新的服务注册信息应用到系统中。

变更通知的工作原理如图 9-8 所示。

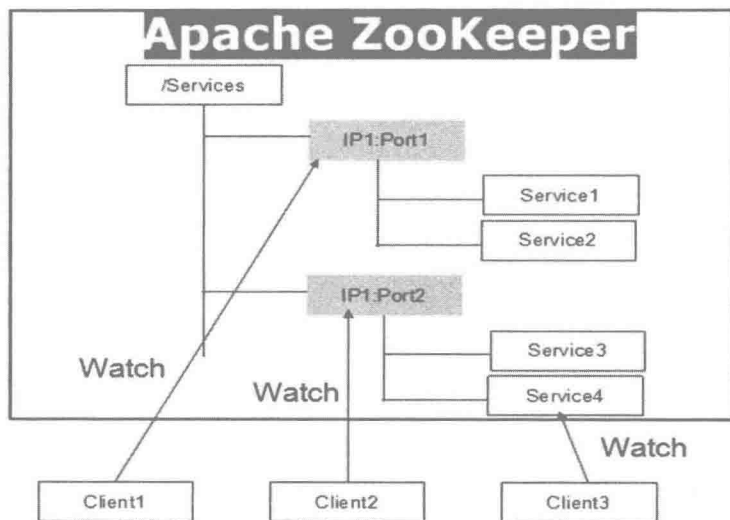


图 9-8 变更通知工作原理图

9.4 总结

分布式服务框架需要一个服务注册中心实现服务的订阅和发布、服务动态发现以及透明化路由。实现分布式服务框架的技术方案有很多，例如基于 ZooKeeper、etcd 或者数据库的技术方案。无论选择哪种技术框架来构建服务注册中心，可靠性、安全性、可扩展性等都是必须要重点考虑的架构特性。

服务发布和引用

服务提供者需要支持通过配置、注解、API 调用等方式，把本地接口发布成远程服务；对于消费者，可以通过对等的方式引用远程服务提供者，实现服务的发布和引用。

10.1 服务发布设计

对于如何将本地 Java 接口发布成远程服务，下面通过服务发布流程来进行讲解，如图 10-1 所示。

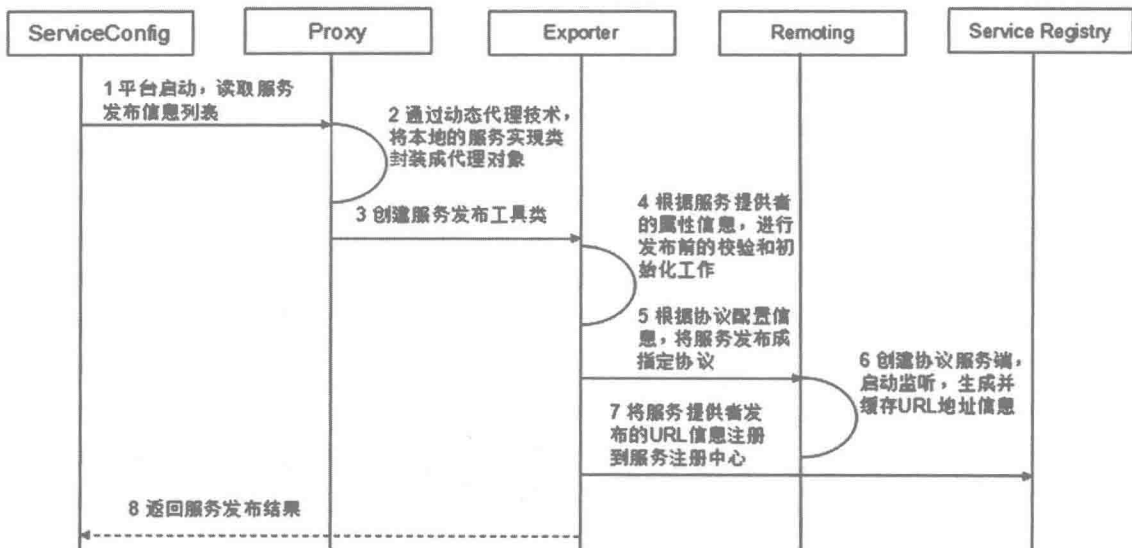


图 10-1 服务发布流程

在下面的小节中，我们将对服务发布过程中的关键技术点进行剖析和讲解。

10.1.1 服务发布的几种方式

通常存在三种服务发布形式：

- 1) XML 配置化方式。
- 2) 注解方式。
- 3) API 调用方式。

三种服务发布方式的优缺点对比如表 10-1 所示。

表 10-1 不同的服务发布方式对比

服务发布方式	优 点	缺 点
XML 配置化	1) 服务框架对业务代码零侵入 2) 扩展和修改方便 3) 修改配置不需要重新编译代码	无
注解	1) 对业务代码低侵入 2) 扩展和修改方便	修改配置需要重新编译代码
API 调用	无	1) 对业务代码侵入较强 2) 容易与某种具体的服务框架绑定 3) 修改之后需要重新编译

通过优缺点对比我们发现，采用 XML 配置化的方式发布服务是最佳选择，这也是大多数服务框架推荐采用配置化方式发布服务的主要原因。

采用 XML 配置化方式发布服务的示例如下：

```
<bean id = "echoService" class="edu.neu.EchoServiceImpl" />
<xxx:service interface="edu.neu.EchoService" ref="echoService" />
```

采用注解方式发布服务的代码示例如下：

```
@Service(name="echoService", version="1.0.1", group="Domain")
public interface EchoService {
    @Method(timeout=3000,retry=1, executeLimit = 10, mock = true)
    String echo(String pingMsg);
}
```

采用 API 调用方式发布服务的代码示例如下：

```
EchoService echoService = new EchoServiceImpl();
ServiceConfig service = new ServiceConfig();
service.setRegistry(registry);
service.setProtocol(protocol);
```

```
service.setRef(echoService);  
service.setName("echoService");  
service.setVersion("1.0.0");  
service.setGroup("Domain");  
service.export();
```

10.1.2 本地实现类封装成代理

对于服务提供者，将本地实现类封装成代理对象不是必需的；也可以利用一系列工具类解析服务提供者信息，然后将服务提供者的地址信息注册到服务注册中心。采用动态代理的好处如下：

- 1) 不管是什么服务，它们的发布流程都是相似的，通过抽象代理层，可以对服务发布行为本身进行封装和抽象。
- 2) 通过动态代理对象，可以对服务发布进行动态拦截，方便平台和业务对服务发布进行个性化定制。
- 3) 便于扩展和替换。

动态代理具体的技术实现可以有多种方式，例如 JDK 自带的 Proxy、Spring 的 InvocationHandler，以及开源的字节码处理框架 javassist，服务框架设计者可以根据具体场景需求选择相应技术。

10.1.3 服务发布成指定协议

根据服务配置的属性信息，将服务发布成指定协议。需要指出的是，同一个服务允许发布成多种协议，多协议发布的配置示例如下：

```
<bean id = "echoService" class="edu.neu.EchoServiceImpl" />  
  
<xxx:service interface= "edu.neu.EchoService" ref="echoService", protocol=  
"HTTP,thrift"/>
```

服务发布工具类 `Exporter` 根据协议信息，创建协议服务端；如果协议服务端已经创建，则不需要重新创建，只需要根据服务提供者信息生成协议 URL，缓存 URL 和服务代理实例的映射关系即可。

相关协议服务端的创建由服务框架的 `Remoting` 模块负责，`Remoting` 负责各种协议的服务端创建、客户端创建、消息的收发、心跳检测、断连重连和端口释放等。

10.1.4 服务提供者信息注册

将服务按照指定协议发布之后，需要将服务发布信息注册到注册中心，服务注册的目的有两个：

- ◎ 供消费者订阅服务地址信息进行服务路由。
- ◎ 基于服务注册中心的统一服务治理。

初始化服务注册中心客户端 SDK，根据配置的注册中心地址连接注册中心，将服务提供者信息进行注册。

服务注册的结构有多种方式，例如按照主机地址、按照服务名或者 URL，目录结构没有固定的模式，设计者需要根据自己产品的实际需要进行判断。以主机地址信息为根目录的服务注册示例如图 10-2 所示。

需要指出的是，服务注册要放在服务发布的最后一个环节进行。因为一旦服务地址信息注册到服务注册中心，消费者就能够获得地址信息并发起请求调用；

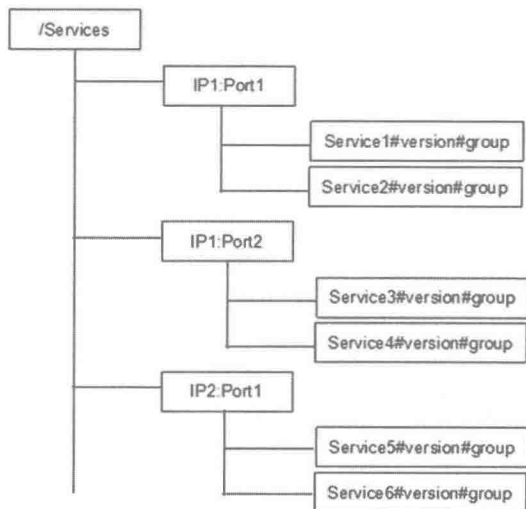


图 10-2 服务信息注册目录划分

而如果此时服务提供者尚未完成初始化，就会导致服务调用失败。

10.2 服务引用设计

消费者导入服务提供者的接口 API 定义，配置服务引用信息，即可在代码中直接调用远程服务，其流程如图 10-3 所示。

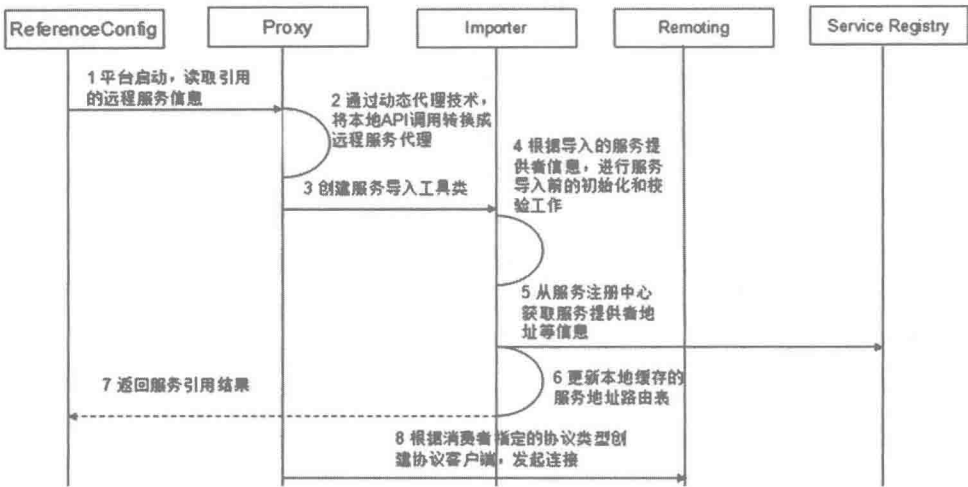


图 10-3 消费者服务引用流程

10.2.1 本地接口调用转换成远程服务调用

消费者导入服务提供者的本地 API 之后，在配置文件中引用远程服务提供者，即可实现远程服务调用，它的配置示例如下：

```
<xxx:reference id="echoService" interface="edu.neu.EchoService" />
<bean class="edu.neu.xxxAction" init-method="start">
    <property name="echoService" ref="echoService" />
</bean>
```

它的原理是根据导入的服务提供者接口 API 和服务引用信息，生成远程服务的本地动态代理对象；它负责将本地的 API 调用转换成远程服务调用，然后将结果返回给调用者。实现本地动态代理的方式很多，上节已经进行过相关介绍，此处不再展开讲解。

10.2.2 服务地址本地缓存

消费者根据引用的服务名等信息，连接服务注册中心，获取已经发布的服务提供者地址等信息，缓存到本地的服务路由表中。服务路由的时候，直接从本地缓存的地址表中获取服务地址信息，按照指定的策略进行服务路由。

服务消费者和提供者的启动顺序无法控制，因此消费者需要检测指定服务目录，监听新的服务提供者注册和已发布服务的下线，它的工作原理如图 10-4 所示。

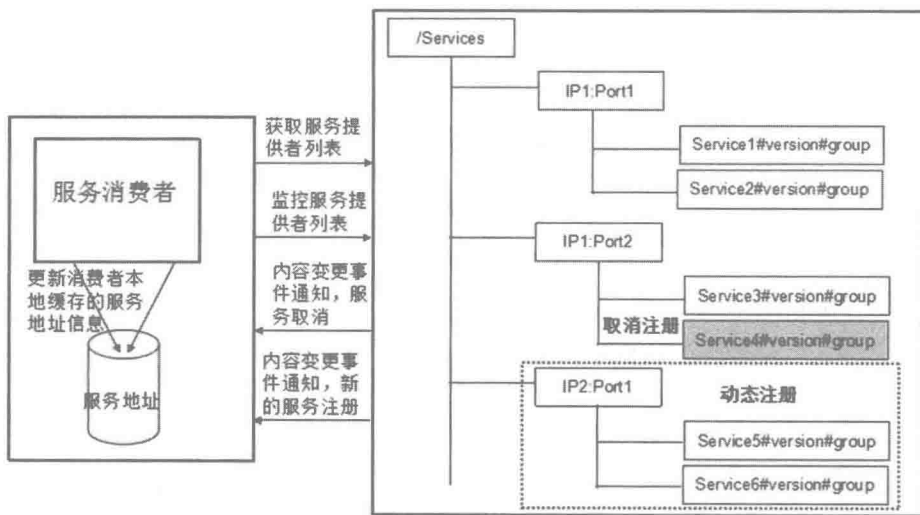


图 10-4 服务地址本地缓存工作原理

10.2.3 远程服务调用

消费者从本地缓存的服务列表中按照指定策略路由，将请求消息封装成协议消息；调

用相关协议的客户端将请求发送给服务提供者，业务线程按照服务调用方式选择同步等待或者注册监听器回调。

协议客户端读取到应答消息，将数据报解码成协议应答消息；再根据序列化格式将消息体反序列化成业务 POJO 对象，唤醒业务等待的线程（或者回调注册的监听器）；业务线程获得服务调用结果返回。

如果消费者调用超时，则按照集群容错策略执行 Failover、Failcache 等重试策略，保证服务调用成功，消费者远程服务调用流程如图 10-5 所示。

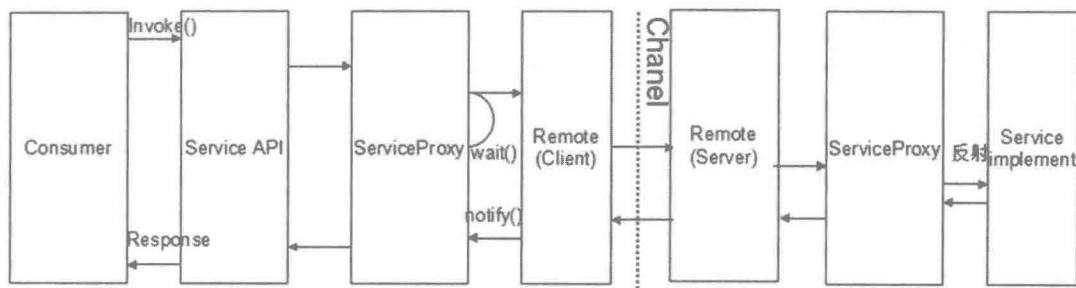


图 10-5 消费者远程服务调用流程

10.3 最佳实践

服务的发布和引用看似简单，但如果设计不当，还是会给用户带来很多困惑和问题，下面针对典型问题进行总结。

10.3.1 对等设计原则

如果服务框架打算支持多种服务发布形式，各种服务发布形式提供的功能应该是对等的。例如，通过 XML 的 Method 元素可以配置方法级参数，那么 API 或者注解也应该支持方法级设置，以防止能力不对等。

要么不提供多种发布形式，要么就提供完全对等的形式，这就是服务发布的准则。

10.3.2 启动顺序问题

在实际项目中，服务提供者、消费者、服务注册中心的启动顺序无法控制（运行态存在故障重启等），因此服务框架需要能够支持这种乱序：

- ◎ 服务注册中心先启动，服务提供者后启动，服务消费者最后启动。
- ◎ 服务注册中心先启动，服务消费者后启动，服务提供者最后启动。
- ◎ 服务提供者先启动，服务注册中心后启动，服务消费者最后启动。
- ◎ 服务提供者先启动，服务消费者后启动，服务注册中心最后启动。
- ◎ 服务消费者先启动，服务注册中心后启动，服务提供者最后启动。
- ◎ 服务消费者先启动，服务提供者后启动，服务注册中心最后启动。

需要指出的是，通常服务注册中心会先于服务提供者和消费者启动；但是如果发生服务注册中心宕机重启等，服务提供者和消费者需要能够自动重连服务注册中心。一些服务注册中心客户端 SDK 自身就支持断连重连，例如 ZooKeeper Client，如果注册中心客户端不支持，需要服务框架提供重连机制。

10.3.3 同步还是异步发布服务

对于大型应用系统，发布的服务比较多，系统启动时间会比较长，一个优化思路就是采用异步发布服务加速系统启动。但是这会带来一个问题，可能系统已经启动成功，但是还有部分服务没有发布，也就是说系统已经启动成功，但是部分服务还没完成初始化。这在集群系统中问题不是很大，因为只要集群中有一个节点完成了该服务的发布，从外部看，服务就是正常的；除非所有节点都没完成某个服务的发布，这种场景下会出现集群系统启动成功，但是延迟一段时间后所有服务才能正常工作。

通常情况下，服务全部准备就绪的时间比较短，而且系统启动之后也并不意味着所有服务都会被立即消费，因此，采用异步的方式发布服务也是可行的。

当然还有另外一些办法可减少系统启动时间。例如对于不经常访问的服务采用延迟发布的策略；还有就是服务的懒加载，只发布服务但是不初始化，等到消费者真正调用的时候才进行初始化服务。

10.3.4 警惕网络风暴

在大规模集群系统中，服务注册中心可能管理数十万条的服务注册信息以及上万个服务提供者和消费者节点。如果服务注册中心管理了大量经常变更的信息，就会发生频繁的变更通知；而这种海量的变更通知可能会挤占服务注册中心的网络带宽，严重时还会导致网络风暴。

因此，在设计时需要考虑如下几个要素：

- ◎ 哪些信息需要注册到服务注册中心，需要甄别。
- ◎ 服务注册中心能够管理的服务上限。
- ◎ 服务注册中心的网络带宽规划。
- ◎ 服务注册中心的磁盘空间规划。
- ◎ 服务注册中心的性能。

10.3.5 配置扩展

无论是服务提供者还是消费者，功能均会不断增加；而通过配置扩展的方式增加新功能，可以保证业务的前向兼容，同时保持架构的稳定性。

一种策略就是服务框架的 XML Schema 提供足够的扩展能力，例如利用 `anyAttribute`

实现未被 schema 规定的属性来扩展 XML 文档，示例如下：

```
<xs:element name="method">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="timeout" type="xs:integer"/>
      <xs:element name="executeLimit" type="xs:integer"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
```

通过 anyAttribute 对 method 元素进行扩展，新增是否支持服务降级的 mock 属性，扩展示例如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.edu.cn"
xmlns="http://www.edu.cn"
elementFormDefault="qualified">

  <xs:attribute name="mock">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="force|exception"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:schema>
```

在代码中，新增解析扩展元素的代码，即可实现框架功能的扩展。

10.4 总结

一个好的分布式服务框架对业务代码的侵入要足够低，例如通过 XML 配置化的方式将普通的 Java 接口发布成远程服务；消费者通过配置化的方式引用服务提供者的接口。基于接口编程，服务框架底层无论如何变更都不会影响上层应用，这就真正实现了业务和平台解耦。

第 11 章

服务灰度发布

灰度发布是指在黑与白之间，能够平滑过渡的一种发布方式。
AB test 就是一种灰度发布方式：让一部分用户继续用 A，一部分用户开始用 B；如果用户对 B 没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到 B 上面来。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题，以保证其影响度。

11.1 服务灰度发布流程设计

服务灰度发布的主要作用如下：

- 1) 解决服务升级不兼容问题。
- 2) 及早获得用户的意见反馈，完善产品功能，提升服务质量。
- 3) 缩小服务升级所影响的用户范围，降低升级风险。
- 4) 让用户及早参与产品测试，加强用户互动。

11.1.1 灰度环境准备

在正式进行服务灰度发布之前，需要对灰度环境进行划分、隔离和准备，其流程设计如图 11-1 所示。

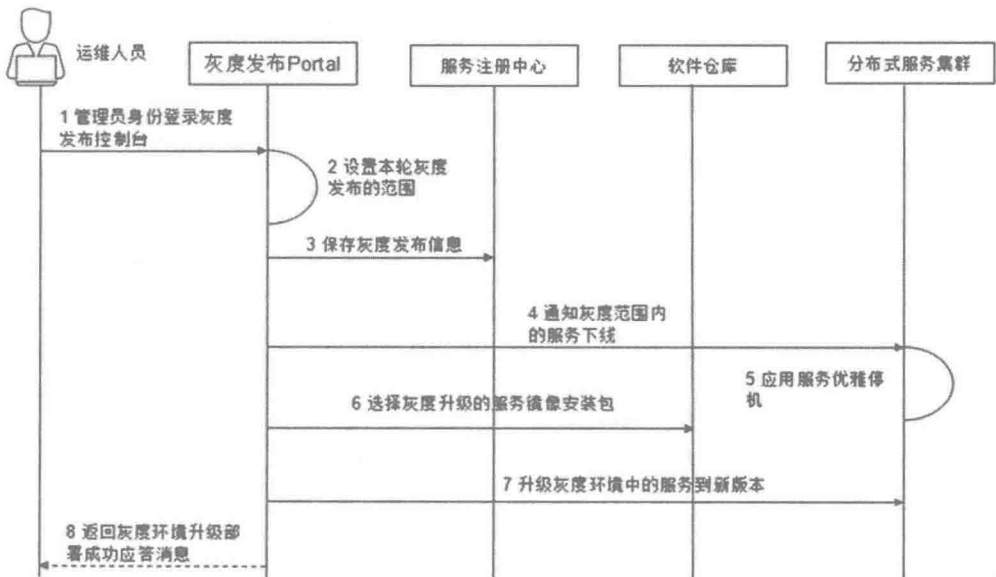


图 11-1 灰度发布环境准备流程

第一步，系统运维人员通过管理员账号登录灰度发布 Portal 或者进入服务治理的灰度发布界面。

第二步，在生产环境中圈定本轮灰度发布的范围，通常它是一个应用集群，包括前后台服务，当然也可能是单个服务。

第三步，将选择的服务灰度发布范围信息保存到服务注册中心，用于后续的规则下发和灰度升级历史记录查询等。

第四步，通知灰度升级范围内的服务实例下线，通常会采用优雅停机的方式让待升级的服务下线，保证升级不中断业务。

第五步，应用进程接收到优雅停机指令之后，将本进程内缓存的消息处理完，然后优雅退出。

第六步，从软件仓库选择需要升级的服务安装镜像包，用于灰度环境的版本升级。

第七步，将升级包批量上传到灰度环境中，把原来的业务软件包做本地备份之后，升级服务版本。

第八步，灰度环境升级部署成功之后，返回灰度环境部署成功消息给灰度发布管理控制台，然后进行后续的灰度发布操作。

11.1.2 灰度规则设置

灰度环境准备完成之后，运维人员对灰度规则进行配置，灰度规则主要用于服务路由。按照规则的不同，部分用户将调用老的服务，另一部分用户则会调用灰度环境中新发布的服务，常用的灰度规则分类如下：

- 1) 按照接入门户类型分类，例如网上营业厅、手机客户端、营业厅实体店、自助业务办理终端等。
- 2) 按照终端类型分类，例如安卓、iOS、Windows Phone 等。

3) 按照区域进行划分, 例如东北、华北、华中等。

4) 其他策略……

通常服务框架会提供几种默认的灰度规则供用户使用, 用户也可以自定义灰度规则。灰度规则的技术实现有多种, 如果要支持复杂规则可以使用规则引擎, 简单的则可以使用表达式, 或者 JSON 文本; 具体使用什么技术, 主要看服务框架的应用场景。一种比较好的策略是, 服务框架不限制灰度规则的具体实现技术, 支持用户自定义灰度规则。

灰度规则设置的流程如图 11-2 所示。

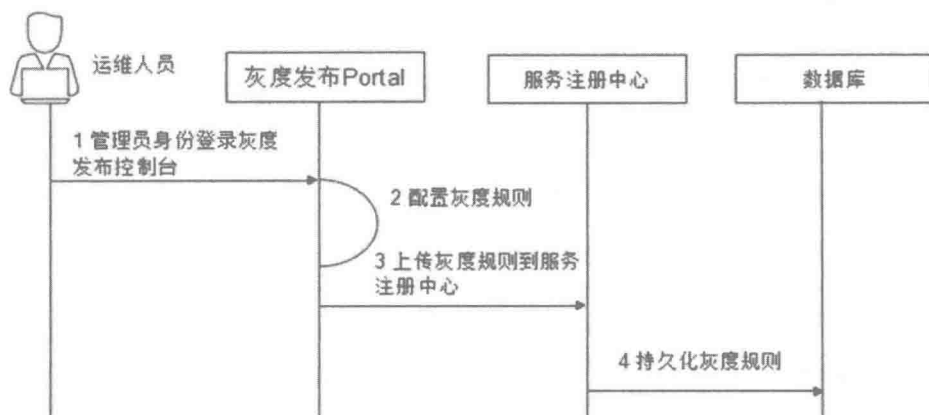


图 11-2 灰度规则设置流程

11.1.3 灰度规则下发

灰度规则设置完成之后, 需要将规则下发给参与消息路由的软负载均衡器 SLB、Web 前台和后台服务, 它的处理流程如图 11-3 所示。

灰度规则下发, 主要由服务注册中心负责推送, 推送的目标包括前端的 SLB 负载均衡器、Web 前台集群和 App 后台服务集群, 各节点将灰度规则缓存到本地内存中; 消息或者服务路由时, 通过路由插件解析灰度规则, 将消息路由到指定的服务版本中。需要指出的是, 灰度规则的通知范围是整个生产环境集群, 包括灰度发布环境和非灰度生产环境。

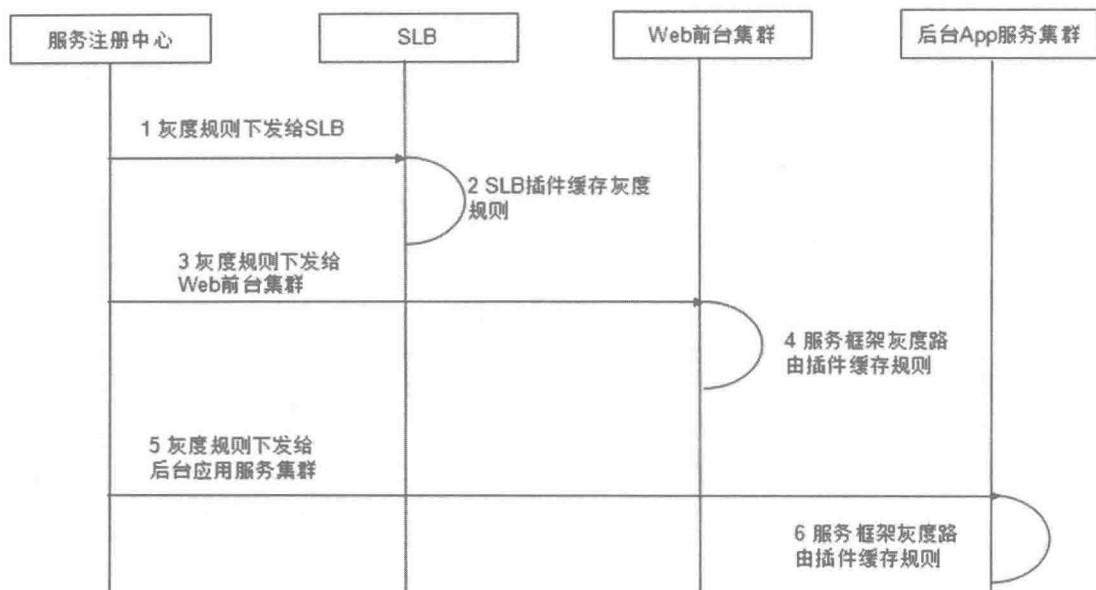


图 11-3 灰度规则下发流程

11.1.4 灰度路由

通过 SLB 定制的灰度发布插件,可以将 HTTP 消息按照规则分发到不同的 Web 前台; Web 前台根据内置的服务框架 SDK,通过客户端灰度路由插件,解析灰度规则,将服务路由到灰度或者非灰度环境,实现服务的灰度路由。

灰度路由的流程如图 11-4 所示。

需要指出的是,如果灰度规则解析失败,实际上就无法区分哪些服务应该路由到灰度环境,这种场景下比较合适的做法就是将服务路由到非灰度环境。如果服务提供者无法处理或者处理失败,则需要对灰度发布做回退处理,并通知所有受影响的服务消费者。

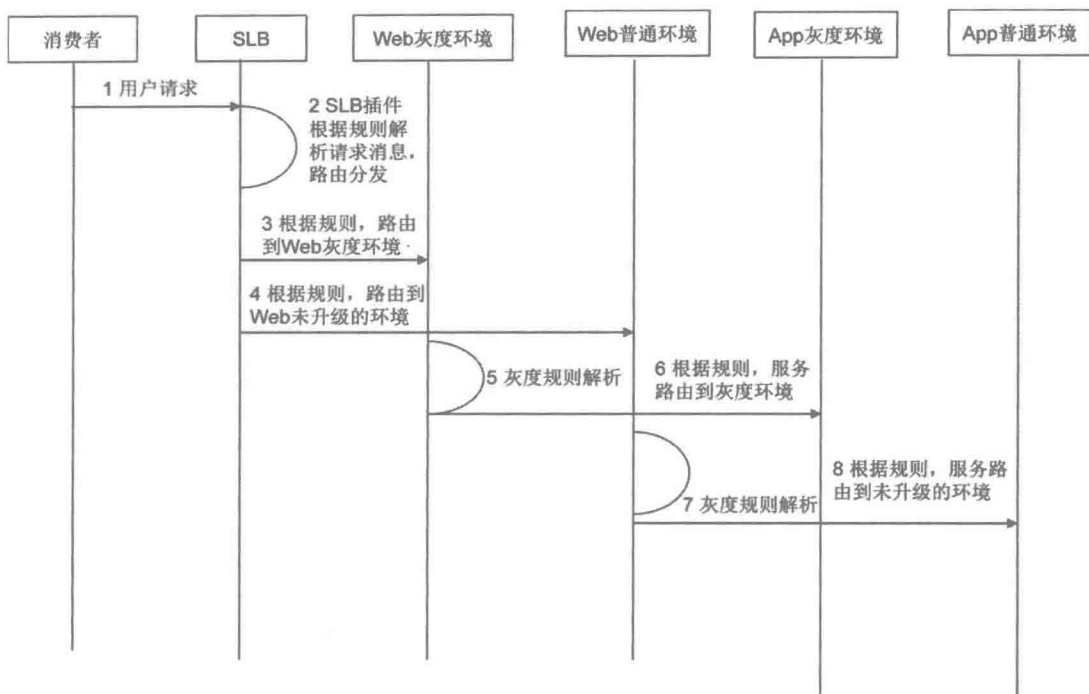


图 11-4 灰度路由流程

11.1.5 失败回滚

如果灰度升级失败，需要支持失败回滚。失败回滚可以是全自动回滚，也可以是人工操作回滚，这取决于灰度发布功能的成熟度和整个业务发布、测试流程的自动化程度。失败回滚的流程如图 11-5 所示。

11.1.6 灰度发布总结

灰度发布之后，需要对灰度发布之后的服务运行和运营情况进行分析，包括服务调用来源分析、服务性能 KPI 数据、用户行为分析报告、用户问卷调查等，通过对这些数据进行分析来改进服务功能，完善产品，为新一轮灰度发布做铺垫。

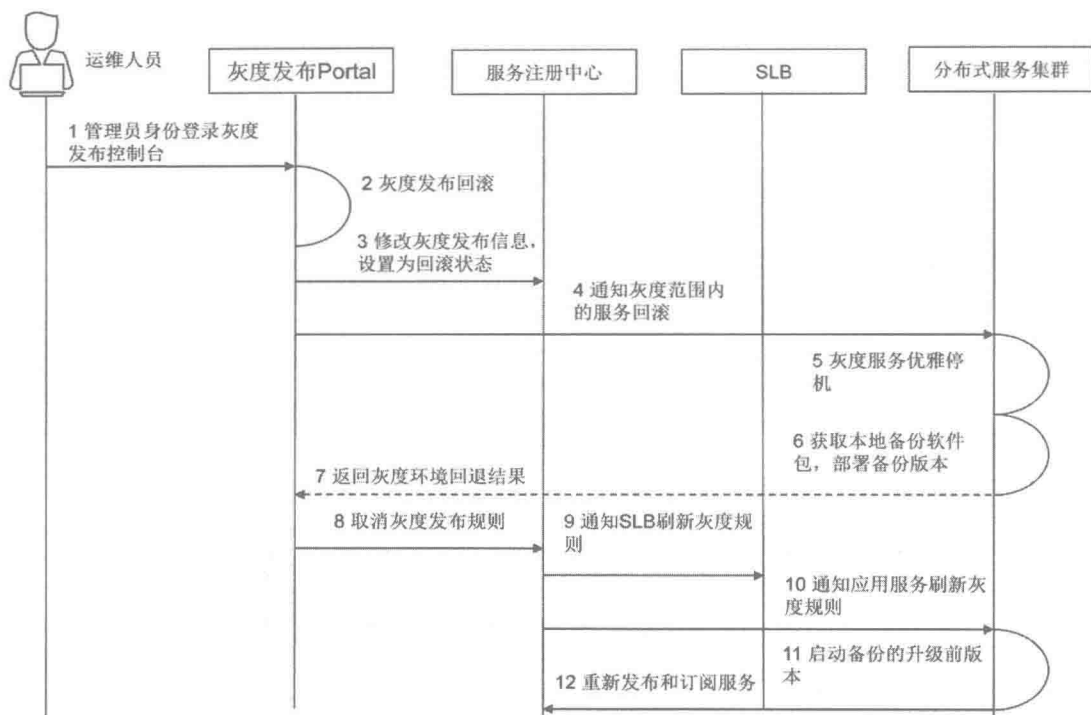


图 11-5 灰度发布失败回滚流程

11.2 总结

互联网产品有一个特点，就是不停的升级，升级，再升级。很多产品基本上保持每周一次的发布频率。系统升级总是伴随着风险，新旧版本兼容的风险，用户使用习惯突然改变而造成用户流失的风险，系统宕机的风险等。为了避免这些风险，很多产品都采用了灰度发布的策略，其主要思想就是把影响集中到一个点，然后再发散到一个面，出现意外情况后很容易就能回退。

分布式服务框架支持服务的灰度发布，可以实现业务的快速试错和敏捷交付，缩短新业务和产品的上线周期，非常重要。

第 12 章

参数传递

服务消费者和提供者之间进行通信时，除了接口定义的请求参数，往往还需要携带一些额外参数，例如消息提供者的 IP 地址、消息调用链的跟踪 ID 等；这些参数不能通过业务接口来进行传递，需要底层的分布式服务框架支持这种参数传递方式。

12.1 内部传参

内部传参主要指服务提供者或者消费者接口内部业务上下文信息的传递，它主要包括如下几种场景：

- 1) 业务内部参数传递。
- 2) 服务框架和业务之间的参数传递。

12.1.1 业务内部参数传递

无论是服务提供者还是消费者，它们都需要调用本地的 API 对业务逻辑进行编排。API 调用将涉及参数传递，比较常用的业务流程编排有三种：

- 1) 硬编码，在业务逻辑中进行 API 调用，参数通过 API 接口进行引用传参。
- 2) 业务编排引擎对业务流程进行编排，参数往往通过抽象的编排上下文进行传递。
- 3) 通过专业的 BPM 流程引擎进行业务逻辑编排，参数通过流程上下文进行传递。

硬编码通常会直接通过方法参数进行参数传递，因为对于本地方法调用而言，不需要有严格的契约对参数进行约束，内部修改和代码重构也非常方便和随意，如图 12-1 所示。

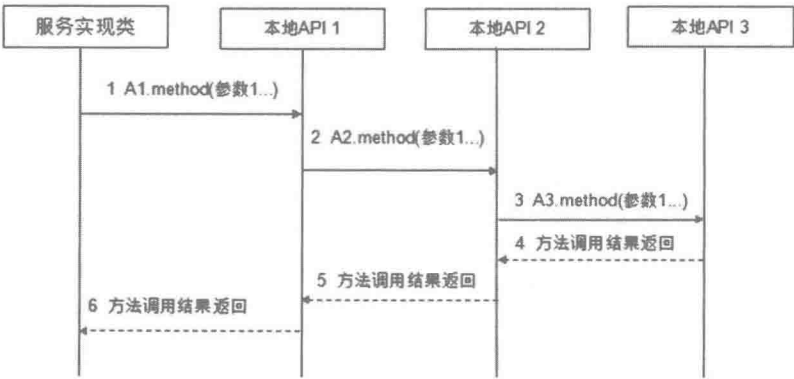


图 12-1 本地方法调用参数传递

还有一种比较常用的方法就是通过线程上下文进行参数传递。通常情况下业务逻辑处理过程中很少发生线程切换，因此通过线程上下文进行隐式传参可以不与某个具体方法接口耦合，对业务接口没有侵入，使用起来也非常方便。例如 Spring 的资源 and 事务线程绑定机制，利用的就是 JDK 提供的线程上下文。使用线程上下文传参是一种隐式传参，上面的方法调用可以简化成如图 12-2 所示的方式。

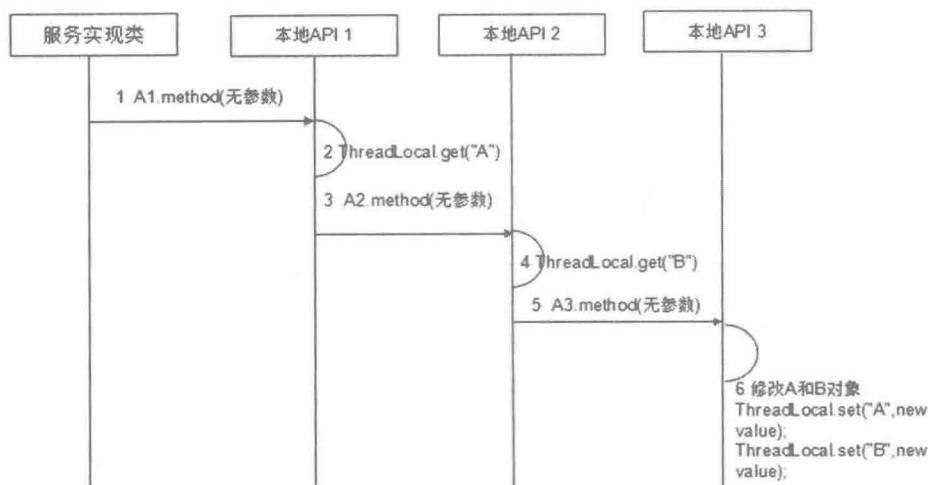


图 12-2 通过线程上下文传递参数

使用线程上下文传参的好处就是不需要再通过参数引用的方式进行参数传递，只要线程不发生切换，可以在任何地方设置、更新和获取上下文信息，就像使用本地变量一样方便，这大大提升了业务的开发效率。

业务编排引擎实际上就是轻量级的 BPM 流程引擎，它往往基于职责链模式实现，业务流程被抽象成 Handler，由职责链编排引擎负责对业务流程进行编排。它的代码示例如下：

```

public interface IHandler{
    void execute(MessageContext context) throws Exception;}
  
```

业务的各个处理流程被抽象成各种 Handler，上下文统一包装成 MessageContext 对象（通常会是一个接口，有不同的实现类），业务参数通过 MessageContext 在不同的 Handler

中被传递处理。采用链式编排的好处是业务修改起来非常方便，编程模型也比较统一，代码更容易管理和维护。假如业务初始处理流程如下：

```
public void initHandlerChain(  
    executeEngine.add("authHandler"); //业务鉴权  
    executeEngine.add("logHandler"); // 记录接口日志  
    executeEngine.add("billHandler"); // 生成账单  
)
```

后来业务流程发生变化，需要在生成账单之后将账单信息写入数据库，我们通过新增数据库处理 Handler 就可以实现流程定制：

```
public void initHandlerChain(  
    executeEngine.add("authHandler"); //业务鉴权  
    executeEngine.add("logHandler"); // 记录接口日志  
    executeEngine.add("billHandler"); // 生成账单  
    executeEngine.add("dbHandler"); // 将账单信息写入数据库  
)
```

当然在业务流程编排引擎中也可以使用线程变量进行参数传递，但是通常 MessageContext 会有多种实现，也支持用户自定义参数，所以线程变量在这种模式下并不常用。

最后一种方式就是 BPM 流程引擎。根据 BPMN 的规范，流程引擎通过流程上下文进行参数传递，用户可以在流程编排界面声明流程级参数和全局参数，流程引擎通过流程上下文进行参数传递，它的使用方式与线程上下文类似，是一种更高级的接口封装形式，如图 12-3 所示。

无论使用哪家供应商提供的 BPM 流程引擎，它都有一套自己的参数定义和传递机制，用户按照指导直接使用即可，非常方便。

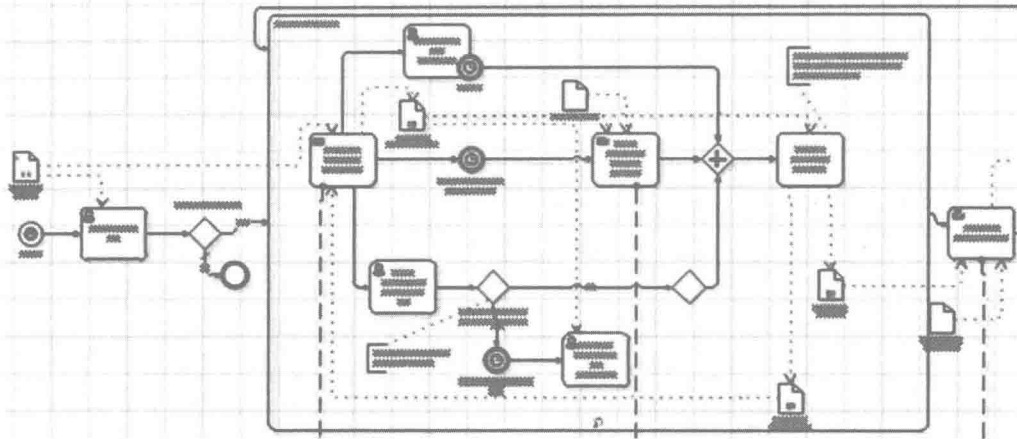


图 12-3 BPM 流程引擎示例图

12.1.2 服务框架内部参数传递

服务框架内部由多个模块组成，模块之间的调用通常会发生线程切换；另外，当服务框架通过反射调用服务接口实现类时，也需要向业务代码传递一些额外的参数，这些参数如何传递？下面我们分别对这两类场景进行分析。

通常框架将数据报反序列化成业务请求对象之后，需要将消息封装成 Task，丢到后面的业务线程池中执行，此时会发生线程切换，示例如图 12-4 所示。

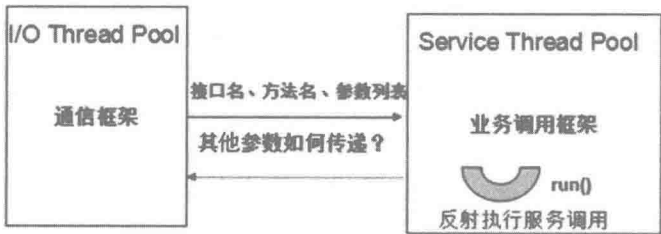


图 12-4 通信模块和服务调用模块的参数传递

由于发生了线程切换，如果通过线程变量的方式传递参数，需要遍历线程上下文，将线程变量复制到业务线程池的线程变量中，非常麻烦。如果后续新增系统参数，往往会忘

记复制，导致参数不一致。

在这种场景中，一般会选择通过消息上下文进行参数传递，例如在业务请求参数中定义 `Map<String, Object>` 扩展参数，用于跨线程的参数传递。

当服务框架回调业务接口实现类时，由于是通过反射调用，业务接口的参数已经定义好了，无法传递其他参数（例如消费者的 IP 地址、调用链 ID 等参数）。为了解决这个问题，需要利用线程变量，因为平台调用服务实现类时不会发生线程切换，所以通过线程变量传参是安全的，它的示例代码如下：

```
//RPCContext 是个线程变量
RPCContext.setParam("callingIP", "10.139.123.11" );
Object value = serviceImpl.invoke(method, args);
//业务通过线程上下文获取调用方 IP 地址
String callingIP = RPCContext.getParam("callingIP");
```

12.2 外部传参

外部传参主要用于服务消费者和提供者之间进行参数传递，它的主要用途包括两个：

- 1) 服务框架自身的参数传递，例如分布式事务中的事务上下文信息传递。
- 2) 业务之间的参数传递，例如业务调用链 ID 的传递，用于唯一标识某个完整业务流程。

12.2.1 通信协议支持

消费者的自定义参数传递到服务端，需要有一个载体，它就是通信协议。一个设计良好的协议，往往支持用户自定义扩展消息头，在协议消息头中，可以预留一个 `Map<String, byte []>` 类型的字段，用于服务框架或者用户自定义参数扩展。

如果不希望传递复杂类型参数，消息头中的扩展字段可以定义为 `Map<String, String>`，

例如 HTTP 协议；如果希望支持复杂类型参数传递，建议定义为 `Map<String, byte[]>`，这种方式最灵活和通用。

12.2.2 传参接口定义

服务框架需要提供参数设置 API 接口，用于业务跨进程的参数传递。考虑到便捷和安全性，建议使用线程变量，例如平台定义一个 `RPCContext` 线程变量供业务传参使用。

服务框架在将业务请求参数传递到通信框架时，需要遍历 `RPCContext`，将框架和业务设置的参数复制到通信线程中，由通信线程在序列化时将请求参数设置到消息头中，传递到通信对端，如图 12-5 所示。

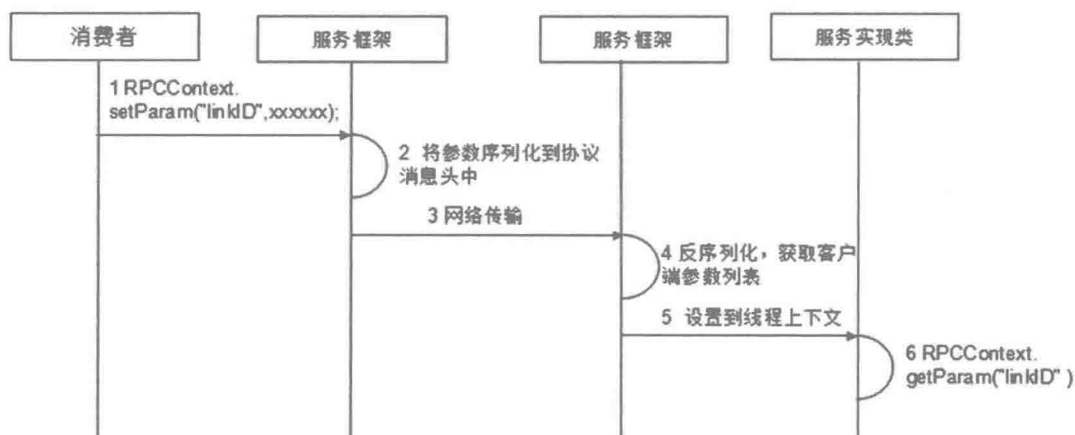


图 12-5 跨进程参数传递

框架提供参数设置接口（例如 `RPCContext`），用户按照规则将需要传递到对端的参数设置到平台上下文中，由服务框架负责参数的传递和编解码。

12.3 最佳实践

12.3.1 防止参数互相覆盖

由于使用的是 Map 或线程变量，因此需要防止参数互相覆盖：

- 1) 服务框架系统参数和业务参数的互相覆盖。
- 2) 业务之间的参数覆盖。

一些系统参数往往默认会被平台占用，例如 IP、timeStamp、Host、ServiceName、Group 等，这些字段太常用了，因此平台需要防止跟业务重名而被覆盖。一个比较好的经验就是平台规划自己可能使用到的系统参数，将这些参数名（Key）预先占用，落实到服务框架使用规范中，不允许业务占用。

对于业务而言，为了防止一个流程内部自身的参数覆盖，需要制定一个业务传参规则，防止参数互相覆盖；这个规则没有标准，只要能够达成目标即可。

12.3.2 参数生命周期管理

无论是使用 Map 还是线程变量，都需要对参数的生命周期进行管理，否则容易导致内存泄漏等问题。通常情况下，服务框架可以对参数生命周期进行自动管理，例如对于服务端，服务调用前设置参数，服务调用后清空参数。但是对于消费者比较复杂，如果在将参数序列化到请求消息头发送之后自动清空参数，后续应答返回之后消费者可能需要继续访问之前的参数，此时参数被服务框架自动删除了；如果服务框架不自动删除，消费者后续不使用该参数或者忘记删除此参数就会导致参数堆积。如果参数名是固定的，这个参数会被覆盖，不会导致太严重的问题；但是如果每次设置的参数名都不同，则会发生内存泄漏。

解决该问题的最好办法就是平台的 API 多提供一个删除模式的参数，允许用户手动删

除，如果用户不指定就使用自动删除模式，通过显式的参数定义将内存管理作为契约强制约束使用者进行设置。

12.4 总结

参数传递在实际项目开发中非常有用，服务框架在设计过程中需要考虑几种不同的参数传递模式，并能给业务提供相关 API 和指导。

服务多版本

服务上线之后，随着业务的发展需要对功能进行变更，或者修复线上的 BUG；服务升级之后，往往需要对服务采用多版本管理。服务多版本管理是分布式服务框架的重要特性，它涉及服务的开发、部署、在线升级和服务治理。

13.1 服务多版本管理设计

服务多版本管理对象包括服务提供者和消费者。

- 1) 服务提供者：发布服务的时候，支持指定服务的版本号。
- 2) 服务消费者：消费服务的时候，支持指定引用的服务版本号或者版本范围。

13.1.1 服务版本号管理

服务的版本号信息并不是仅供人工阅读的，“版本”在分布式服务框架中是一个受系统管理的信息，维护一个服务的不同版本也是分布式服务框架支持服务在线升级的重要特性。当一个消费者依赖某个服务提供者时，通常需要指明它依赖服务的版本号信息。

服务的版本号是有序的，在服务名相同的情况下，两个相同服务名的不同服务版本的版本号可以比较大小。完整的版本号由“主版本号（Major）+ 副版本号（Minor）+ 微版本号（Micro）”构成。

根据经验，上述 3 个版本号的约定俗成划分原则如下。

- 1) 主版本号：表示重大特性或者功能变更，接口或者功能可能会不兼容（在实际开发中应当避免这种不兼容）。
- 2) 副版本号：发生了少部分功能变更，或者新增了一些功能。
- 3) 微版本号：主要用于 BUG 修复，对应于常见的 SP 补丁包。

服务版本比较的原则：从前往后逐项比较，当且仅当服务名、服务主版本号、服务副版本号、服务微版本号全部相同时，两个服务才是同一个服务，否则以第一个出现差异的版本号的大小决定服务版本的大小。

13.1.2 服务提供者

服务提供者的 XML Schema 的 Service Element 需要包含 version 属性，用于服务提供者在发布服务的时候指定服务的版本号。

服务开发完成之后，需要将一个或者多个服务打包成一个 jar 包或者 war 包。为了便于对服务进行物理管理，打包后的名称中会包含服务的版本号信息，例如 com.huawei.orderService_1.0.1.jar。

在微服务架构中，微服务独立开发、打包、部署和升级，因此微服务的版本和软件包的版本可以一一映射。但是在实际开发中，特别是大规模企业应用开发，单独为每个服务打包和部署目前尚未成为主流，它会增加服务软件包的管理和线上治理成本，因此目前主流模式仍然是多个服务提供者合设打成一个大的 jar/war 包。这就会存在一个问题：项目开发初期，各个服务的版本号保持一致；但是运行一段时间之后，有些服务进行了版本升级，有些服务没有，这样当它们被打包成同一个软件包时，这就会导致版本号不一致。

另外，如果为每个服务都指定一个版本号，对开发而言也比较麻烦。有没有更好的服务版本管理方式呢？一种比较好的实践就是微服务 + 全局版本模式。对于经常发生功能变更、需要独立升级的服务，将其独立拆分出来进行微服务化，实现单个微服务级的打包和部署；这样微服务的版本就可以独立演进，而不受制于其他服务。

对于其他服务，服务框架提供全局版本功能，在 Maven 组件工程开发时，只需要为整个工程配置一个版本号，该组件工程包含的所有服务都共用该版本号。如果组件工程包含的某个服务发生了版本变更，就统一升级全局版本号，其他未发生功能变更但是打包在一起的服务做级联升级。这样做的一个原因是服务被打包在一起之后，无论其他服务是否需要升级，只要软件包中的一个服务发生了版本升级，其他合设的服务也必须与其一起打包升级，它们之间存在物理上的耦合，这也是为什么微服务架构提倡微服务独立打包、部署和升级的原因。

13.1.3 服务消费者

与服务提供者不同的是，服务消费者往往不需要指定具体依赖的服务版本，而是一个

版本范围，例如：

version = "[1.0.1, 2.0.8]"; version = "[2.0.3, 3.0.0)"; version = "[3.0.0,)"

在服务框架中引入版本范围的原因有如下：

- 1) 消费者关心的是某个新特性从哪个服务版本中开始提供，它并不关心服务提供者的版本演进以及具体的版本号。
- 2) 消费者想使用当前环境中服务的最新版本，但是并不清楚具体的版本号，希望自动适配最新的服务版本。

如果消费者不指定导入的服务版本，默认将查找“0.0.0”版本的服务提供者，如果找不到则抛出指定的 RPC 异常。这样做的核心目的就是要在开发阶段将服务版本号的设计和管理纳入到研发流程中进行统一管理。

13.1.4 基于版本号的服务路由

服务提供者将服务注册到服务注册中心时，将服务名 + 服务版本号 + 服务分组作为路由关键信息存放到注册中心，服务消费者在发起服务调用时，除了携带服务名、方法名和参数列表之外，还需要携带要消费的服务版本信息，由路由接口负责服务版本过滤，原理如图 13-1 所示。



图 13-1 基于服务版本的地址过滤

如果没有找到满足条件的服务提供者地址列表，则抛出服务路由失败的 RPC 异常，根据集群容错策略做进一步处理。

13.1.5 服务热升级

服务热升级的目标就是在业务不中断的情况下，实现系统的平滑升级，考虑到版本升级的风险，往往需要做多次滚动升级，最终根据升级之后新版本服务的运行状况决定是继续升级还是回退。这就意味着在同一时刻，整个集群环境中会同时存在服务的多个版本在线运行，这就是热升级相比于传统 AB Test 等升级方式的差异。

服务热升级的流程图如图 13-2 所示。

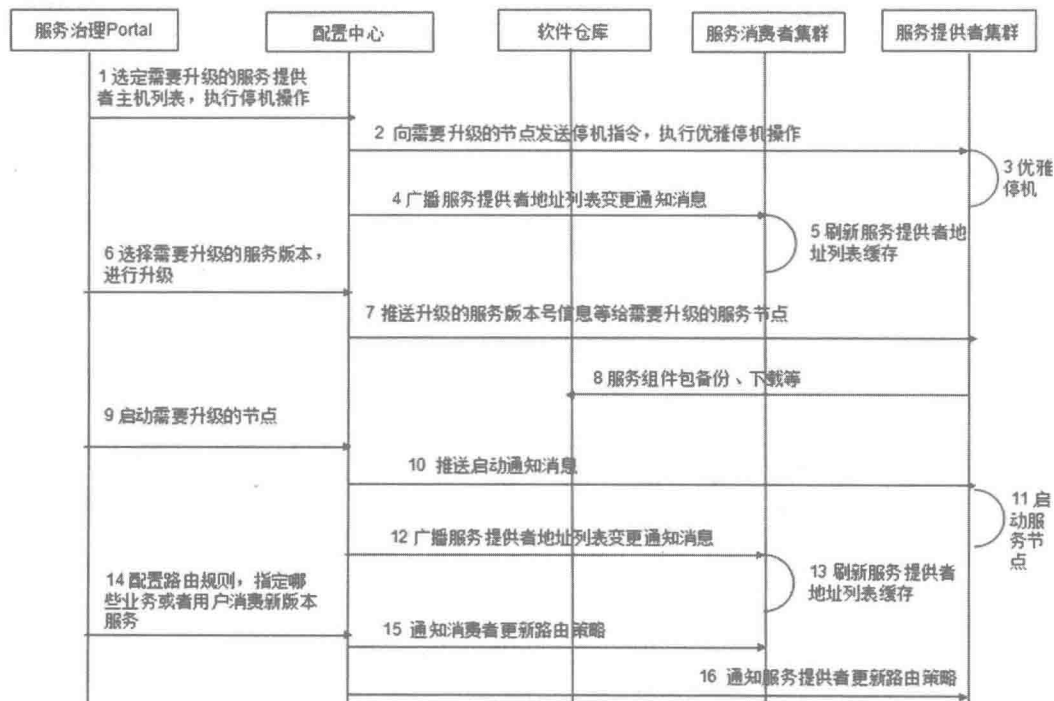


图 13-2 服务热升级流程图

服务热升级的几个核心点如下。

- 1) 升级的节点需要重启，但是由于分布式服务框架具备服务的健康检测和自动发现机制，停机升级的节点自动被隔离，停机并不会中断业务。
- 2) 服务路由规则的定制：如果是滚动式的灰度发布，在相当长的一段时间（例如一周）内线上都会存在服务的多个版本。哪些用户或者业务需要路由到新版本上，需要通过路由策略和规则进行指定，服务框架应该支持用户配置自定义的路由规则来支持灵活的路由策略。
- 3) 滚动升级和回退机制：无论在预发布环境中测试的多么充分，到了真实的生产环境中仍然可能会发生新的问题。此时，需要做服务的热回退，它的步骤跟服务热升级雷同，此处不再赘述。为了降低服务热升级对业务的影响，同时考虑到可靠性，在实际工作中往往采用滚动升级的方式，分批次进行服务的热升级，这样可以及时发现问题，更敏捷地进行特性交付，滚动升级示意图如图 13-3 所示。

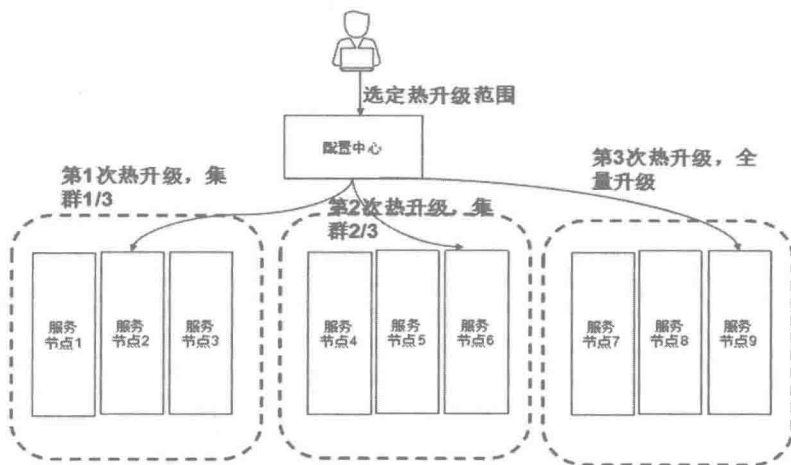


图 13-3 服务滚动热升级示意图

13.2 与 OSGi 的对比

OSGi 联盟成立于 1999 年 3 月，致力于制定管理本地网络设备服务的规范。OSGi 联

盟是为家用设备、汽车、手机、桌面、小型办公环境以及其他环境制定下一代网络服务标准的领导者。OSGi 全名原为 Open Services Gateway initiative, 但现在这个全名已经废弃。

OSGi 联盟基于它成立之初的使命, 推出了 OSGi 服务平台规范, 用于提供开放和通用的架构, 使得服务提供商、开发人员、软件提供商、网关操作者和设备提供商以统一的方式开发、部署和管理服务。

目前最广为认可和应用的是 OSGi 规范 5 (Release 5), 共由核心规范、标准服务 (Standard Services)、框架服务 (Framework Services)、系统服务 (System Services)、协议服务 (Protocol Services)、混合服务 (Miscellaneous Services) 等几部分共同组成。核心规范是 OSGi 规范中的核心部分, 它通过一个分层的框架, 实现了 OSGi 最为成功的动态插件机制, 它主要提供了:

- 1) OSGi Bundle 的运行环境。
- 2) OSGi Bundle 间的依赖管理。
- 3) OSGi Bundle 的生命周期管理。
- 4) OSGi 服务的动态交互模型。

OSGi 两个最核心的特性就是模块化和热插拔机制, 分布式服务框架的服务多版本管理和热升级是否可以基于 OSGi 来实现? 下面带着这个疑问, 围绕着模块化和插件热插拔这两个特性进行详细分析。

13.2.1 模块化开发

在 OSGi 中, 我们以模块化的方式去开发一个系统, 每个模块被称为 Bundle, OSGi 提供了对 Bundle 的整个生命周期管理, 包括打包、部署、运行、升级、停止等。

模块化的核心并不是简单地把系统拆分成不同的模块, 如果仅仅是拆分, 原生的 Jar 包 + Eclipse 工程就能够解决问题。更为重要的是要考虑到模块中接口的导出、隐藏、依赖、版本管理、打包、部署、运行和升级等全生命周期管理, 这些对于原生的 jar 包而言

是不支持的。

传统开发模式的问题：在 Maven 之前，模块的划分通常有两种方式。

- 1) 定义一个大而全的工程包含多个子模块，不同模块之间使用 package 来进行隔离，如图 13-4 所示。

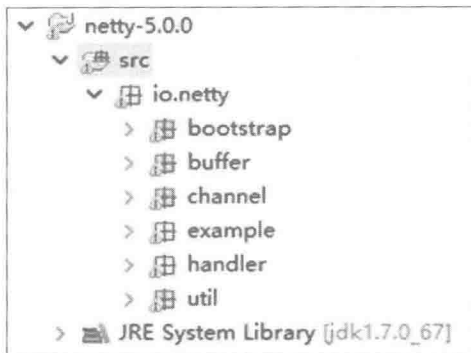


图 13-4 通过 package 来进行模块划分

- 2) 定义多个子工程，工程之间通过工程引用的方式进行依赖管理，如图 13-5 所示。

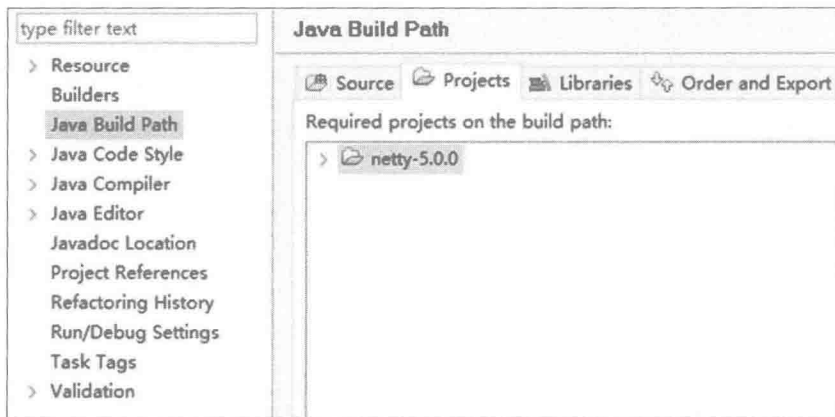


图 13-5 工程引用方式

这两种方式都存在同一个问题：无法实现资源的精细划分和对依赖做统一管理。以 jar 包依赖为例，依赖一个 jar 包就意味着这个 jar 包中所有 public 的资源都可能被引用，但事

实上也许只需要依赖该 jar 包中的某几个 public 接口。我们无法对资源做细粒度、精确的管控，无法控制哪些接口是内部使用、哪些是开放给外部访问的，我们也不知道 public 的接口都被哪些模块依赖和使用，消费者是谁。更为复杂的场景是如果消费者需要依赖不同的接口版本，那该怎么办？

OSGi 很好地解决了这个问题，每个 OSGi 工程是一个标准的插件工程，实际就是一个 Bundle。Eclipse 默认提供了可视化的界面管理插件之间的依赖，我们可以可视化地指定依赖的 package 和需要导出供其他插件使用的 package，OSGi 显式地管理 package 级的接口依赖关系，而不是通常工程级或者 jar 包级的依赖，管控粒度更细。

使用 Eclipse 内置的 plugin project 模板创建的标准 OSGi 工程如图 13-6 所示。

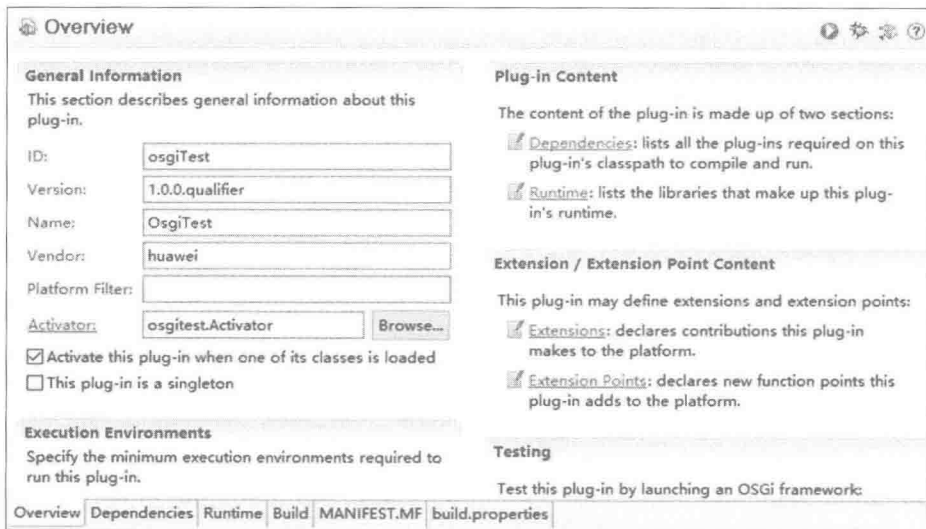


图 13-6 OSGi 插件工程

我们可以选择只导出 osgiTest Bundle 的 api package 下的公共接口和类，隐藏不需要对外开放的 Activator 类，这样就比较容易实现高内聚-低耦合，示例如图 13-7 所示。



图 13-7 选择只导出指定的插件 package

使用 Maven 之后，可以非常方便地对工程进行模块化设计、开发、打包和管理，它解决了开发态的模块化问题，使用 Maven 管理之后的 Netty 工程目录如图 13-8 所示。



图 13-8 使用 Maven 工程管理的 Netty 工程源码

通过 Maven Eclipse 插件，可以可视化的对 Maven 依赖进行管理，包括直接依赖和继承依赖，如图 13-9 所示。

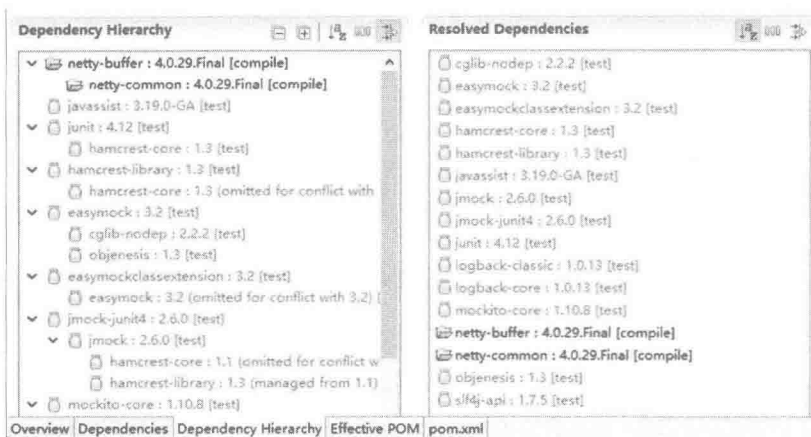


图 13-9 Maven 可视化依赖管理

可能大家还有一个疑问：Maven 实际上管理的是 jar 包层面的依赖，如何实现更细粒度的依赖管理呢，例如 package 级，甚至是接口级？

这个就要靠分布式服务框架的服务导出和导入功能来实现了，我们再温习下服务发布和引用的相关知识：

- 1) 服务提供者通过 `service export` 将某个服务接口发布出去，供消费者使用。
- 2) 服务消费者通过 `service import` 导入某个服务接口，它不关心服务提供者的具体位置，也不关心服务的具体实现细节。

通过服务的导入和导出功能，以及服务注册中心对服务订阅关系的管理，分布式服务框架自身就实现了服务接口级的依赖管理，它比 OSGi 的 package 导入导出功能粒度更细。

利用 Maven 的模块化管理 + 分布式服务框架自身的服务接口导入和导出功能，解决了模块化开发和精细化依赖管理难题，完全可以替代 OSGi 的相关功能。

13.2.2 插件热部署和热升级

OSGi 另外一个非常酷的特性就是动态性，即插件的热部署和热升级，它可以在不重启 JVM 的情况下新安装部署一个插件，或者对已有的插件做热升级，实现了升级不中断业务。

OSGi 的插件热部署和热升级原理就是基于自身的网状类加载机制实现的，在此不详细赘述，感兴趣的读者可以查阅 OSGi 类加载原理相关的资料。下面我们分析下在分布式服务框架中，如何实现服务热部署和热升级。

站在客户的角度，升级不中断业务是根本诉求。至于采用什么样的技术实现，客户并不关心，不同的产品、不同的语言会有不同的实现机制。

分布式服务框架实现服务热部署和热升级的原理如下：

- 1) 服务是分布式集群部署的，通常也是无状态的，停掉其中某一个服务节点，并不会影响系统整体的运行质量。
- 2) 服务自动发现和隔离机制，当有新的服务节点加入时，服务注册中心会向消费者集群推送新的服务地址信息；当有服务节点宕机或者重启时，服务注册中心会发送服务下线通知消息给消费者集群，消费者会将下线服务自动隔离。
- 3) 优雅停机功能，在进程退出之前，处理完消息队列积压的消息，不再接受新的消息，最大限度保障不丢失消息。
- 4) 集群容错功能，如果服务提供者正在等待应答消息时系统退出了，消费者会发生服务调用超时，集群容错功能会根据策略重试其他正常的服务节点，保证流程不会因为某个服务实例宕机而中断。
- 5) 服务多版本管理，支持集群中同一个服务的多个版本同时运行，支持路由规则定制，不同的消费者可以消费不同的服务版本。

基于上述几点，我们发现即便不使用 OSGi，同样可以实现服务的热部署和热升级，而且很多特性本身就是分布式服务框架自身具备的特性，并不需要做额外开发和适配。

相比于 OSGi 在 JVM 内部通过定制类加载机制实现插件的多版本运行和升级，使用分

布式服务框架自身的分布式集群特性实现服务的热部署和热升级，更加简单、灵活和可控。

13.2.3 不使用 OSGi 的其他理由

在分布式服务框架中，不使用 OSGi 的其他理由总结如下：

- 1) 遗留应用的迁移，需要付出很高的代价，这种代价业务通常不愿意买单。
- 2) 为了处理 OSGi/非 OSGi 不同的场景，框架代码变的很复杂。
- 3) 针对 Class Loading 问题的特殊处理，非常不优雅。
- 4) 框架 Bundle 与业务 Bundle 的互相依赖及启动顺序问题很棘手。
- 5) 构建及调试不够完善。
- 6) 用户学习成本高、使用门槛高。
- 7) 为客户带来的真实价值有限，OSGi 的大多数优点是可以被替换的，而且新的技术更简单和实用。

事实上，OSGi 在嵌入式设备领域的价值仍然是不可替代的，但是在企业应用中，特别是分布式服务框架逐步普及之后，OSGi 的价值会非常有限，但是它带来的复杂度却令很多开发者望而生畏，正因为如此，我们在构建分布式服务框架时，不应依赖 OSGi。

13.3 总结

服务多版本在实际项目中非常实用，可以实现服务的热部署和热升级，同时支持按照消费者做差异化路由。未来演进到微服务架构之后，可以实现服务的独立打包、部署、运行和运维，如果没有服务的多版本管理，我们就很难做平滑迁移。

最后，我们通过开发态和运行态两个维度，详细阐述了为什么在分布式服务框架中不建议使用 OSGi。

第 14 章

流量控制

当资源成为瓶颈时，服务框架需要对消费者做限流，启动流控保护机制。流量控制有多种策略，比较常用的有：针对访问速率的静态流控、针对资源占用的动态流控、针对消费者并发连接数的连接控制和针对并行访问数的并发控制。

在实践中，各种流量控制策略需要综合使用才能起到较好的效果，本章针对分布式服务框架的流量控制设计原则和实践进行讲解。

14.1 静态流控

静态流控主要针对客户端访问速率进行控制，它通常根据服务质量等级协定（SLA）中约定的 QPS 做全局流量控制，例如订单服务的静态流控阈值为 100 QPS，则无论集群有多少个订单服务实例，它们总的处理速率之和不能超过 100 QPS。

14.1.1 传统静态流控设计方案

传统的静态流控设计采用安装预分配方案，在软件安装时，根据集群服务节点个数和静态流控阈值，计算每个服务节点分摊的 QPS 阈值，系统运行时，各个服务节点按照自己分配的阈值进行流控，对于超出流控阈值的请求则拒绝访问。

静态预分配方案原理如图 14-1 所示。

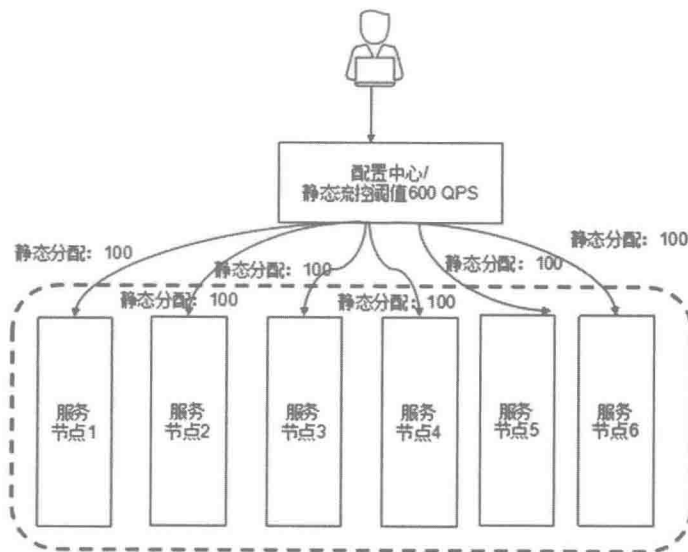


图 14-1 静态预分配方案

服务框架启动时，将本节点的静态流控阈值加载到内存中，服务框架通过 Handler 拦

截器在服务调用前做拦截计数，当计数器在指定周期 T 到达 QPS 上限时，启动流控，拒绝新的请求消息接入。

有两点需要注意：

- 1) 服务实例通常由多线程执行，因此计数时需要考虑线程并发安全，可以使用 Atomic 原子类进行原子操作。
- 2) 到达流控阈值之后拒绝新的请求消息接入，不能拒绝后续的应答消息，否则这会导致客户端超时或者触发 FailOver，增加服务端的负载。

14.1.2 传统方案的缺点

静态分配方案的最大缺点就是忽略了服务实例的动态变化：

- 1) 云端服务的弹性伸缩特性使服务节点数处于动态变化过程中，预分配方案行不通。
- 2) 服务节点宕机，或者有新的服务节点动态加入，导致服务节点数发生变化，静态分配的 QPS 需要实时动态调整，否则会导致流控不准。

分布式服务框架的一个特点就是服务的动态上下线和自动发现机制，这就决定了在运行期服务节点数会随业务量的变化而频繁变化，在这种场景下静态分配方案显然无法满足需求。

当应用和服务迁移到云上之后，PaaS 平台的一个重要功能就是支持应用和服务的弹性伸缩，在云上，资源都是动态分配和调整的，静态分配阈值方案无法适应服务迁移到云上。

14.1.3 动态配额分配制

为了解决静态分配的缺陷，在实践中通常会采用动态配额分配制。它的工作原理：由服务注册中心以流控周期 T 为单位，动态推送每个节点分配的流控阈值 QPS。当服务节点发生变更时，会触发服务注册中心重新计算每个节点的配额，然后进行推送，这样无论是

新增还是减少服务节点数，都能够在下一个流控周期内被识别和处理，这就解决了传统静态分配方案无法适应节点数动态变化的问题。

动态配额分配制的工作流程如图 14-2 所示。

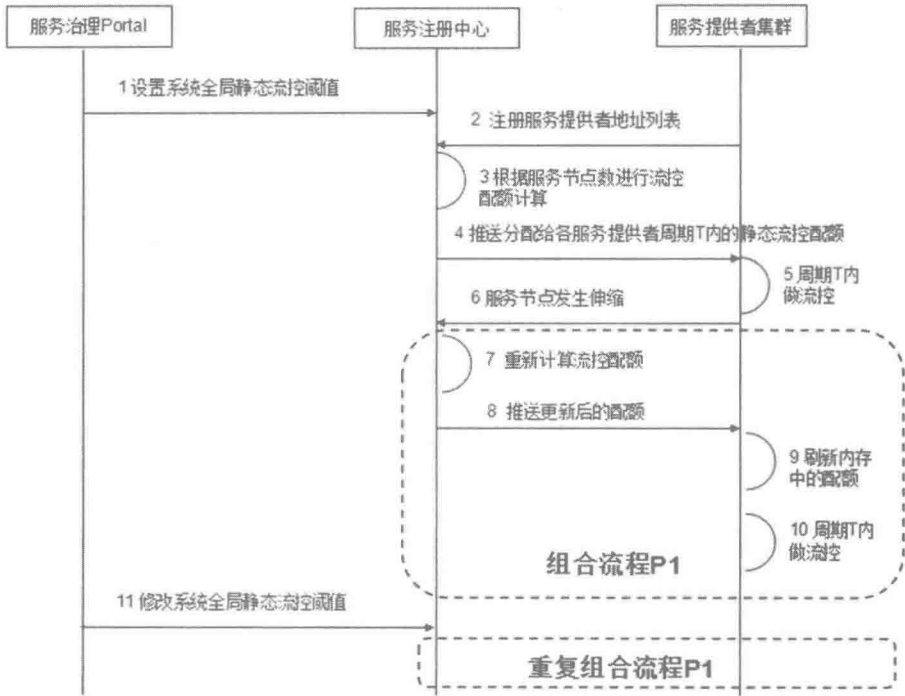


图 14-2 动态配额分配制流程图

在生产环境中，每台机器/VM 的配置可能不同，如果每个服务节点采用流控总阈值/服务节点数这种平均主义，可能会发生性能高、处理快的节点配额很快用完，但是性能差的节点配额有剩余的情况，这会导致总的配额没用完，但是系统却发生了静态流控的问题。一种解决方案就是服务注册中心在做配额计算时，根据各个服务节点的性能 KPI 数据（例如服务调用平均时延）做加权，处理能力差的服务节点分配的指标少些，性能高的分配的指标多些，这样就能够尽可能低地降低流控偏差。

还有另外一种解决方案就是配额指标返还和重新申请，每个服务节点根据自身分配的指标值、处理速率做预测，如果计算结果表明指标会有剩余，则把多余的返还给服务注册

中心；对于配额已经使用完的服务节点，重新主动去服务注册中心申请配额，如果连续 N 次都申请不到新的配额指标，则对于新接入的请求消息做流控。

最后一点就是结合负载均衡进行静态流控，才能够实现更精确的调度和控制。消费者根据各服务节点的负载情况做加权路由，性能差的节点路由到的消息更少，由于配额计算也根据负载做了加权调整，最终分配给性能差的节点配额指标也较少，这样既保证了系统的负载均衡，又实现了配额的更合理分配。

14.1.4 动态配额申请制

尽管动态配额分配制可以解决节点变化引起的流控不准问题，也能够改善平均主义配额分配导致的贫富不均，但是它并不是最优的方案，它的缺点总结如下：

- 1) 如果流控周期 T 比较大，各服务节点的负载情况变化比较快，服务节点的负载反馈到注册中心，由注册中心统一计算之后再做配额均衡，误差会比较大。
- 2) 如果流控周期 T 比较小，服务注册中心需要实时获取各服务节点的性能 KPI 数据并计算负载情况，经过性能数据采集、上报、汇总和计算之后，会有一定的时延，这会导致流控滞后产生误差。
- 3) 如果采用配额返还和重新申请方式，则会增加交互次数，同时也会存在时序误差，效果有限。
- 4) 扩展性差，负载的汇总、计算和配额分配、下发都由服务注册中心完成，如果服务注册中心管理的节点数非常多，则服务注册中心的计算压力就非常大，随着服务节点数的增加服务注册中心的配额分配效率会急速下降，系统不具备平滑扩展能力。

要解决上述问题，可以采用动态配额申请制，它的工作原理如下：

- 1) 系统部署的时候，根据服务节点数和静态流控 QPS 阈值，拿出一定比例的配额做初始分配，剩余的配额放在配额资源池中。

- 2) 哪个服务节点使用完了配额, 就主动向服务注册中心申请配额。配额的申请策略是, 如果流控周期为 T , 则将周期 T 分成更小的周期 T/N (N 为经验值, 默认值为 10), 当前的服务节点数为 M 个, 则申请的配额为 (总 QPS 配额 - 已经分配的 QPS 配额) $/M * T/N$ 。
- 3) 总的配额如果被申请完, 则返回 0 配额给各个申请配额的服务节点, 服务节点对新接入的请求消息进行流控。

动态配额申请制的优点:

- 1) 各个服务节点最清楚自己的负载情况, 性能 KPI 数据在本地内存中计算获得, 实时性高。
- 2) 由各个服务节点根据自身负载情况去申请配额, 保证性能高的节点有更高的额度, 性能差的自然配额就少, 这样能够更合理地调配资源, 实现流控的精确性。

实践经验表明, 采用动态配额申请制的静态流控更精确, 在实战中效果也更好。

14.2 动态流控

动态流控的最终目标是为了保命, 并不是对流量或者访问速度做精确控制。当系统负载压力非常大时, 系统进入过负载状态, 可能是 CPU、内存资源已经过载, 也可能是应用进程内部的资源几乎耗尽, 如果继续全量处理业务, 可能会导致长时间的 Full GC、消息严重积压或者应用进程宕机, 最终将压力转移到集群其他节点, 引起级联故障。

触发动态流控的因子是资源, 资源又分为系统资源和应用资源两大类, 根据不同的资源负载情况, 动态流控又分为多个级别, 每个级别流控系数都不同, 也就是被拒绝掉的消息比例不同。每个级别都有相应的流控阈值, 这个阈值通常支持在线动态调整。

14.2.1 动态流控因子

动态流控因子包括系统资源和应用资源两大类，常用的系统资源包括：

- 1) 应用进程所在主机/VM 的 CPU 使用率。
- 2) 应用进程所在主机/VM 的内存使用率。

主机 CPU、内存使用率采集算法非常多，例如使用 `java.lang.Process` 执行 `top`、`sar` 等外部命令获取系统资源使用情况，然后解析后计算获得资源使用率。也可以直接读取操作系统的系统文件获取相关数据，需要注意的是，无论是执行操作系统的本地命令，还是直接读取操作系统的资源使用率文件，都是操作系统本地相关的，不同的操作系统和服务端，命令和输出格式可能存在很大差异。在计算时需要首先判断操作系统类型，然后调用相关操作系统的资源采集接口实现类，通过这种方式就可以支持跨平台。

常用的应用资源包括：

- 1) JVM 堆内存使用率。
- 2) 消息队列积压率。
- 3) 会话积压率（可选）。

具体实现策略是系统启动时拉起一个管理线程，定时采集应用资源的使用率，并刷新动态流控的应用资源阈值。

14.2.2 分级流控

通常，动态流控是分级别的，不同级别拒掉的消息比例不同，这取决于资源的负载使用情况。例如当发生一级流控时，拒绝掉 1/8 的消息；发生二级流控时，拒绝掉 1/4 消息。

不同的级别有不同的流控阈值，系统上线后会提供默认的流控阈值，不同流控因子的流控阈值不同，业务上线之后通常会根据现场的实际情况做阈值调优，因此流控阈值需要

支持在线修改和动态生效。

需要指出的是为了防止系统波动导致的偶发性流控，无论是进入流控状态还是从流控状态恢复，都需要连续采集 N 次并计算平均值，如果连续 N 次平均值大于流控阈值，则进入流控状态；同理，只有连续 N 次资源使用率平均值低于流控阈值，才能脱离流控状态恢复正常。

根据资源使用率的变化，流控会发生升级或者降级，在同一个流控周期内，不会发生流控级别的跳变。

14.3 并发控制

并发控制针对线程的并发执行数进行控制，它的本质是限制对某个服务或者服务的方法过度消费，耗用过多的资源而影响其他服务的正常运行。

并发控制有两种形式：

- 1) 针对服务提供者的全局控制。
- 2) 针对服务消费者的局部控制。

下面我们分别对这两种场景进行讲解。

14.3.1 服务端全局控制

服务端并行控制的配置示例代码如下：

```
<bean id = "echoService" class="edu.neu.EchoServiceImpl" />
<xxx:service interface="edu.neu.EchoService" ref="echoService" executes="5"/>
```

限制 edu.neu.EchoService 接口，它的并发执行数为 5，如果超过 5，则抛出并行控制异常。

如果要支持服务接口方法级并发控制，它的配置示例如下：

```
<bean id = "echoService" class="edu.neu.EchoServiceImpl" />
<xxx:service interface="edu.neu.EchoService" ref="echoService">
<xxx:method name="echo" executes="5"/>
</xxx:service>
```

分布式服务框架的服务发布 XML Schema 中，**service** 和 **method** 元素需要增加并行控制 **executes** 属性。

14.3.2 服务消费者流控

除了针对服务提供者的全局流控之外，还需要支持针对消费者的流控，不同消费者可以配置不同的流控策略，它的配置如下所示：

```
<bean class="edu.neu.xxxAction" init-method="start">
<xxx:reference id="echoService" interface="edu.neu.EchoService" actives="5"/>
```

限制 edu.neu.EchoService 接口，客户端并发执行数不超过 5。与服务端的流控类似，也支持服务方法级控制，配置示例如下：

```
<bean class="edu.neu.xxxAction" init-method="start">
<xxx:reference id="echoService" interface="edu.neu.EchoService">
<xxx:method name="echo" actives="5"/>
</xxx:reference>
```

分布式服务框架的服务引用 XML Schema 中，**reference** 和 **method** 元素需要增加并行控制 **actives** 属性。

14.4 连接控制

通常分布式服务框架服务提供者 and 消费者之间采用长连接私有协议，为了防止因为消费者连接数过多导致服务端负载压力过大，系统需要支持针对连接数进行流控。

14.4.1 服务端连接数流控

针对分布式服务框架的某个协议，在服务端控制消费者的连接数，示例配置如下：

```
<bean class="edu.neu.xxxAction" init-method="start">  
<xxx:protocol name="xxx" accepts="50"/>
```

限制 XXX 协议客户端连接数不能超过 50。

14.4.2 服务消费者连接数流控

针对某个服务的消费者，限制其连接数，配置示例如下：

```
<bean class="edu.neu.xxxAction" init-method="start">  
<xxx:reference interface="edu.neu.EchoService" connections="50"/>
```

限制接口 edu.neu.EchoService 的消费者，连接数最多不能超过 50 个。

14.5 并发和连接控制算法

并发控制的算法原理如图 14-3 所示。

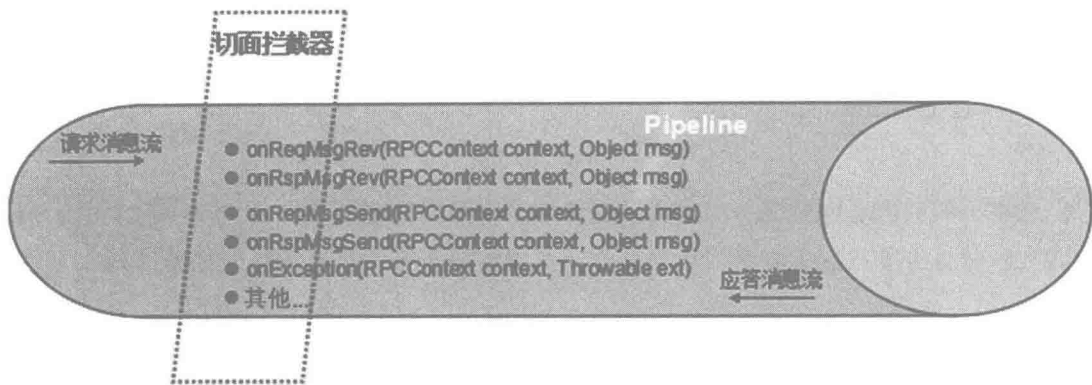


图 14-3 Pipeline 工作原理

基于服务调用 Pipeline 机制，可以对请求消息接收和发送、应答消息接收和发送、异常消息等做切面拦截（类似于 Spring 的 AOP 机制，但是没采用反射机制，性能更高），利用 Pipeline 拦截切面接口，对请求消息做服务调用前的拦截和计数，根据计数器做流控，服务端的算法如下：

- 1) 获取流控阈值。
- 2) 从全局 RPC 上下文中获取当前的并发执行数，与流控阈值对比，如果小于流控阈值，则对当前的计数器做原子自增。
- 3) 如果等于或者大于流控阈值，则抛出 RPC 流控异常给客户端。
- 4) 服务调用执行完成之后，获取 RPC 上下文中的并发执行数，做原子自减。

客户端的流控算法与服务端的不太一样，客户端的流控目的就是要降低对服务端的冲击，因此当客户端达到流控阈值之后，需要将当前的线程挂起，等待其他线程执行完后再执行，或者超时，它的算法如下：

- 1) 获取流控阈值。
- 2) 从全局 RPC 上下文中获取当前的并发执行数，与流控阈值对比，如果小于流控阈值，则对当前的计数器做原子自增。

- 3) 如果等于或者大于流控阈值, 当前线程进入 wait 状态, wait 超时时间为服务调用的超时时间。
- 4) 如果有其他线程服务调用完成, 调用计数器自减, 则并发执行数小于阈值, 线程被 notify, 退出 wait, 继续执行。

示例代码如下:

```
int active = count.getActive();
    if (active >= max) {
        synchronized (count) {
            while ((active = count.getActive()) >= max) { //循环判断
                try {
                    count.wait(remain); //超时等待
                } catch (Exception e) {
                    Log.error(e);
                }
                long elapsed = System.currentTimeMillis() - start;
                remain = timeout - elapsed; //剩余的超时时间
            }
        }
        if (remain <= 0) { //如果服务调用已经超时则没有必要继续等待, 抛出异常
            throw new RpcException ("xxx service control flow timeout --> " + timeout)
        }
        //...后续代码省略
    }
}
```

14.6 总结

流量控制是保障服务 SLA 的重要措施, 也是业务高峰期故障预防和恢复的有效手段, 分布式服务框架需要支持不同的流控策略, 还要支持流控阈值、策略的在线调整, 不需要重启应用即可动态生效, 提升线上服务治理的效率和敏捷性。

第 15 章

服务降级

大促或者业务高峰时，为了保证核心服务的 SLA，往往需要停掉一些不太重要的业务，例如商品评论、论坛或者粉丝积分等。

另外一种场景就是某些服务因为某种原因不可用，但是流程不能直接失败，需要本地 Mock 服务端实现，做流程放通。以图书阅读为例，如果用户登录余额鉴权服务不能正常工作，需要做业务放通，记录消费话单，允许用户继续阅读，而不是返回失败。

上述两种场景，都使用到了分布式服务框架的一个重要服务治理功能：服务降级。服务降级主要包括容错降级和屏蔽降级两种模式，下面我们就两种服务降级的策略和设计进行讲解。

15.1 屏蔽降级

在一个应用实例中，服务往往是合设的，尽管可以通过线程池隔离等方式保证服务之间的资源隔离，但是 100% 的隔离是不现实的。特别是对缓存、网络 I/O、磁盘 I/O、数据库连接资源等公共依赖无法隔离，在业务高峰期或者大促时，服务之间往往存在激烈的竞争，导致订购等核心服务运行质量下降，影响系统的稳定运行和客户体验。

此时需要对非核心服务做强制降级，不发起远程服务调用，直接返回空、异常或者执行特定的本地逻辑，减少自身对公共资源的消费，把资源释放出来供核心服务使用。

15.1.1 屏蔽降级的流程

屏蔽降级的全流程如图 15-1 所示。

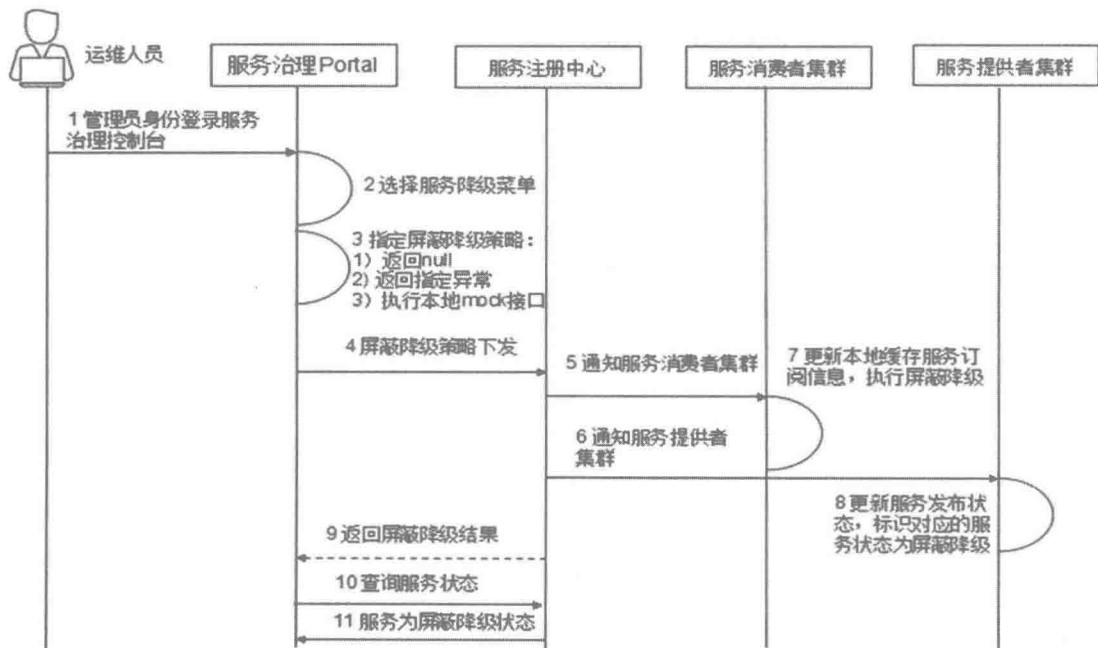


图 15-1 屏蔽降级流程图

第 1 步，运维人员以管理员身份登录服务治理控制台，管理员具备服务治理的全套权限。

第 2 步，运维人员选择服务降级菜单，在服务降级界面中选择屏蔽降级。

第 3 步，通过服务查询界面选择需要降级的服务，注意服务的分组和版本信息，指定具体的降级策略：返回 `null`、返回指定异常还是执行本地 `Mock` 接口实现。

第 4 步，服务治理 `Portal` 通过服务注册中心客户端 `SDK`，将屏蔽降级指令和相关信息发送到服务注册中心。

第 5、6 步，服务注册中心接收到屏蔽降级消息后，以事件的形式分别群发给服务提供者集群和服务消费者集群。

第 7 步，服务消费者接收到屏蔽降级事件通知之后，获取相关内容，更新本地缓存的服务订阅信息。当发起远程服务调用时，需要与屏蔽降级策略做匹配，如果匹配成功，则执行屏蔽降级逻辑，不发起远程服务调用。

第 8 步，服务提供者集群接收到屏蔽降级事件通知之后，获取相关内容，更新本地的服务发布缓存信息，将对应的服务降级属性修改为屏蔽降级。

第 9 步，操作成功之后，服务注册中心返回降级成功的应答消息，由服务治理 `Portal` 界面展示。

第 10 步，运维人员查询服务提供者列表，查看服务状态。

第 11 步，服务注册中心返回服务状态为屏蔽降级状态。

15.1.2 屏蔽降级的设计实现

屏蔽降级通常用于服务运行态治理，开发时不会配置，当外界的触发条件达到某个临界值时，由运维人员/开发人员决策，通过服务治理控制台，进行人工降级操作，它的取值有如下三种：

- 1) mock = force: return null。不发起远程服务调用，直接返回空对象。
- 2) mock = force: throw Exception。不发起远程服务调用，直接抛出指定异常。
- 3) mock = force: execute Bean: <beanName>。不发起远程服务调用，直接执行本地模拟接口实现类。

对于第三种降级策略，被执行的 Spring Bean 需要实现服务提供者的接口，代码示例如下：

```
public interface xxxService
{
    Object xxxMethod(Object req);
}
```

本地 Mock 的 Bean 实现示例如下：

```
public class xxxServiceMock implements xxxService
{
    public Object xxxMethod(Object req)
    { //本地 Mock 逻辑代码实现 }
}
```

服务发布模型中，需要支持服务降级属性，它的配置示例如下：

```
<xxx:service interface="edu.neu.EchoService" ref="echoService" mock =
"force: execute Bean:
echoServiceMock"/>
```

屏蔽降级操作是可逆的，当系统压力恢复正常水平或者不再需要屏蔽降级时，可以对已经屏蔽降级的服务恢复正常。恢复之后，消费者重新调用远程的服务提供者，同时服务状态被修改为正常状态。

15.2 容错降级

当非核心服务不可用时，可以对故障服务做业务逻辑放通，分布式服务框架的业务放通实际属于容错降级的一种。

容错降级不仅仅只用于业务放通，它也常用于服务提供方在客户端执行容错逻辑，容错逻辑主要包括两种。

- 1) RPC 异常：通常指超时异常、消息解码异常、流控异常、系统拥塞保护异常等。
- 2) Service 异常：例如登录校验失败异常、数据库操作失败异常等。

15.2.1 容错降级的工作原理

容错降级的工作原理如图 15-2 所示。

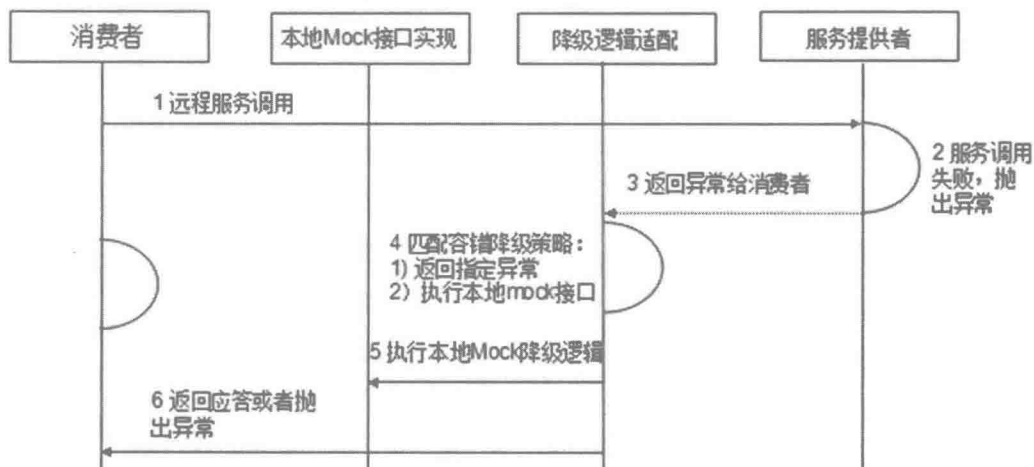


图 15-2 容错降级

容错降级与屏蔽降级的主要差异如下。

- 1) 触发条件不同：容错降级是根据服务调用结果，自动匹配触发的；而屏蔽降级往

往是通过人工根据系统运行情况手工操作触发的。

- 2) 作用不同：容错降级是当服务提供者不可用时，让消费者执行业务放通；屏蔽降级的主要目的是将原属于降级业务的资源调配出来供核心业务使用。
- 3) 调用机制不同：一个发起远程服务调用，一个只做本地调用。

需要指出的是，业务放通的 Mock 接口实现往往放在消费者端，为什么不放在服务提供者一侧，原因如下。

当执行本地业务放通时，可能会依赖消费者本地的资源，包括但不限于消费者依赖的服务、数据结构、数据库资源等，这些资源是消费者私有的，服务提供者可能不可见，如果把这个逻辑搬到服务提供者实现，服务提供者需要依赖这些资源，这会导致系统间的耦合。

另外一个原因就是不同的消费者，消费同一个服务提供者时，对失败之后的处理策略存在差异，如果把这些差异都搬到服务提供者的 Mock 接口实现中，会造成代码臃肿，很难维护。

如果不存在上述两种问题，则业务也可以选择降级逻辑放在服务端实现，对于分布式服务框架而言，也可以支持这两种策略供用户灵活选择。

容错降级的策略如下：

- 1) mock = fail: throw Exception。将异常转义。
- 2) mock = fail: execute Bean: <beanName>。将异常屏蔽掉，直接执行本地模拟接口实现类，返回 Mock 接口的执行结果。

与屏蔽降级不同的是，通常在开发态，就需要指定容错降级的策略，它的消费者配置示例如下：

```
<bean class="edu.neu.xxxAction" init-method="start">
    <xxx:reference id="echoService" interface="edu.neu.EchoService"
actives="5" mock = "fail: execute Bean:
```

echoServiceMock">

需要指出的是，无论是屏蔽降级还是容错降级，都支持从消费者或者服务提供者两个维度去配置，从消费端配置策略更灵活，可以实现差异化降级策略，当然使用起来也更加麻烦。

服务降级策略配置的优先级为：消费者配置策略 > 服务提供者配置策略，屏蔽降级高于容错降级。

15.2.2 运行时容错降级

如果开发态没有指定容错降级策略，系统上线运行后，需要临时增加容错降级策略，服务框架也需要支持在线动态增加容错降级策略，它的工作流程与屏蔽降级类似，如图 15-3 所示。

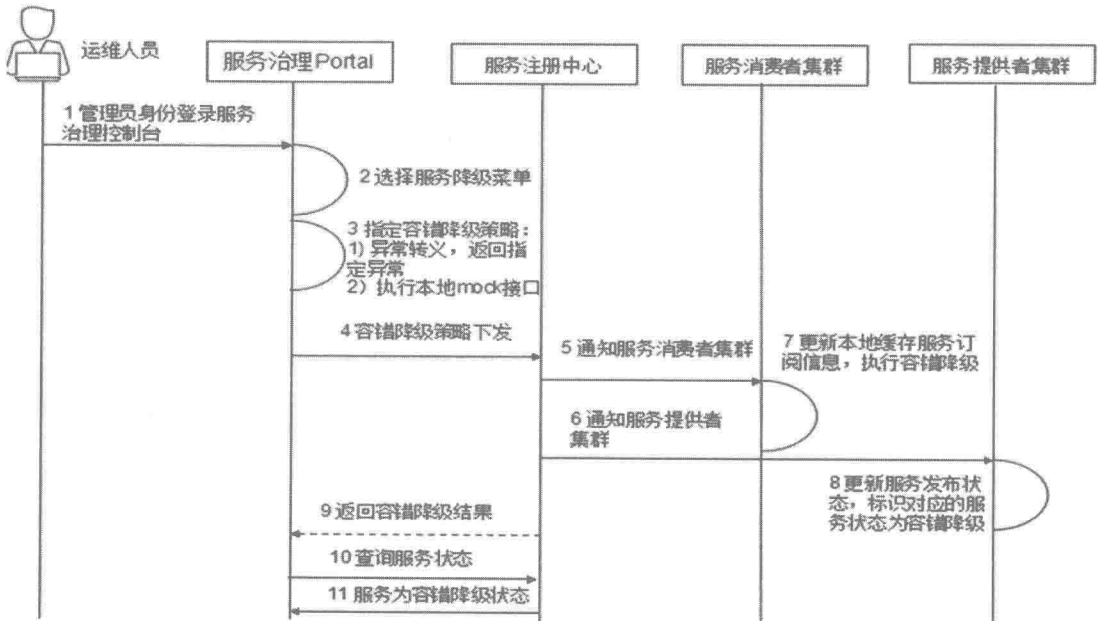


图 15-3 运行时动态添加容错降级策略

在实际项目中，利用容错降级做业务放通是主要的应用场景。

15.3 业务层降级

在实际业务开发过程中，可能会存在比较复杂的业务放通场景，例如“调用 A 服务 + 执行本地方法调用”组合成一个流程，针对这个流程的执行结果做放通，这种场景由于本地方法调用并不经过分布式服务框架，因此需要业务自己做放通处理。

服务降级并不能 100% 满足所有业务放通场景，对于不满足的特殊场景需要业务自己开发业务层的降级框架。

15.4 总结

服务降级功能在实际项目中非常实用，在服务化之前，业务往往需要自己实现放通逻辑或者框架，不同的业务模块、甚至不同的开发者都自己实现了一套私有的放通流程，这对项目的开发和运维都会造成很多麻烦。

更为严重的是由于没有统一的服务降级策略和框架，无法在服务治理 Portal 上进行统一线上降级，在应对业务高峰或者大促海量流量涌入时，运维人员会力不从心，往往需要一大群开发在背后支撑，运维效率非常低下。

基于分布式服务框架的服务降级功能，可以有效提升线上的服务治理效率，保障服务的 SLA。尽管服务降级更多是为了提升服务线上运行质量，但是它反向对服务的设计和开发也有约束。它要求服务在设计之初就要做如下识别：

- 1) 哪些服务是核心服务、哪些是非核心服务？
- 2) 哪些服务支持降级，降级策略是什么？

3) 除了服务降级之外, 是否还存在更为复杂的业务放通场景, 它的策略是什么?

系统的高效、健康运行仅仅依赖线上服务治理和运维是解决不了的, 运维的需求通过分布式服务框架的特性反向映射到设计和开发态, 从设计阶段就开始考虑未来如何高效运维, 才能在根本上提升服务和产品的质量, 这也是矛盾对立和统一的一个具体体现。

服务优先级调度

当系统当前资源非常有限时，为了保证高优先级的服务能够正常运行，保障服务 SLA，需要降低一些非核心服务的调度频次，释放部分资源占用，保障系统的整体运行平稳。

服务优先级的调度策略非常多，对于分布式服务框架而言，需要能够支持服务发布时设置优先级策略，并在资源成为瓶颈时，按照用户配置的优先级策略调度执行服务。

16.1 设置服务优先级

服务在发布的时候，可以指定服务的优先级，如果用户没有指定，采用默认优先级策略，它的配置如下所示：

```
<xxx:service interface="edu.neu.EchoService" ref="echoService" mock =  
"force: execute Bean:  
echoServiceMock" priority = "HIGH"/>
```

服务的优先级可以采用传统的低、中、高三级配置策略，每个级别的执行比例可以灵活配置，如下所示：

```
<xxx:priority  
  <Level type = "HIGH" value = "0.5" />  
  <Level type = "MID" value = "0.3" />  
  <Level type = "LOW" value = "0.2" />  
</xxx:priority>
```

服务发布 XSD 通过扩展 `priority` 属性的方式指定优先级，服务提供者将优先级属性注册到服务注册中心并通知消费者，由消费者缓存服务的优先级，根据不同的优先级策略进行调度。

服务优先级调度有多种策略：

- 1) 基于线程调度器的优先级调度策略。
- 2) 基于优先级队列的优先级调度策略。
- 3) 基于加权配置的优先级调度策略。
- 4) 基于服务迁入迁出的优先级调度策略。

下面我们分别对上述四种策略进行详细讲解。

16.2 线程调度器方案

线程优先级被线程调度器用来判定何时运行哪个线程，理论上，优先级高的线程比优先级低的线程获得更多的 CPU 时间。实际上，线程获得的 CPU 时间通常由包括优先级在内的多个因素决定，一个优先级高的线程自然比优先级低的线程获得更多的 CPU 执行时间，理论上，相同优先级的线程被调度执行的机会是均等的。

如果要设置线程的优先级，可以通过 Thread 的 `setPriority(int newPriority)` 接口，线程优先级的取值介于 `MAX_PRIORITY = 10` 和 `MIN_PRIORITY = 1` 之间，共 10 个级别，默认取值为 `NORM_PRIORITY = 5`。

服务在发布的时候，可以根据用户的优先级配置策略，将服务优先级映射到线程优先级中，然后创建多个不同的优先级线程，分别调度对应的服务，它的工作原理如图 16-1 所示。

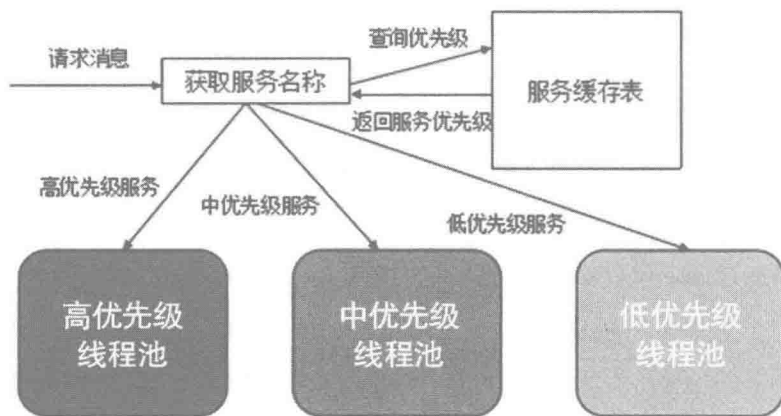


图 16-1 线程优先级调度方案原理

基于线程调度器实现服务优先级调度，算法简单，开发工作量小，因此在实际工作中应用比较广泛。

当有多个线程可以运行时，由线程调度器决定哪些线程将会运行，以及运行多长时间，任何一个合理的操作系统在做出选择时都努力做到公正，但是采用的策略却大相径庭。依

赖于线程调度器来实现服务的优先级调度，很有可能是不可移植的。在不同的操作系统上，相同的优先级配置，执行结果却可能存在很大差异，这对于需要某些精确控制执行比例的服务是不可接受的。

线程优先级可以用来提高一个已经能够正常工作的服务的运行质量，但是却无法保证精确性和跨平台移植性。因此，通常不建议使用线程调度器实现服务的优先级调度。

16.3 Java 优先级队列

Java 的 `PriorityQueue` 是一个基于优先级堆的无界优先级队列。优先级队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 `Comparator` 接口进行排序，具体取决于所使用的构造方法。优先级队列不允许使用 `null` 元素，依靠自然顺序的优先级队列还不允许插入不可比较的对象。

优先级队列的头是按指定排序方式确定的最小元素，如果多个元素都是最小值，则头是其中一个元素（选择方法是任意的）。队列获取操作 `poll`、`remove`、`peek` 和 `element` 访问处于队列头的元素。优先级队列是无界的，但是有一个内部容量，控制着用于存储队列元素的数组大小。它通常至少等于队列的大小，随着不断向优先级队列添加元素，其容量会自动增加，无须指定容量增加策略的细节。

利用 `PriorityQueue` 可以实现服务的优先级调度，其工作原理如图 16-2 所示。

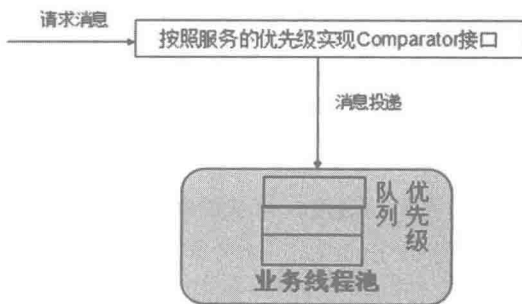


图 16-2 优先级队列实现服务优先级调度

使用 Java 优先级队列，服务的优先级调度由 JDK 负责，由于 `PriorityQueue` 的优先级调度由优先级算法本身决定，与平台无关，因此具备跨平台和可移植性。

`PriorityQueue` 的一个主要缺点就是如果持续有优先级高的消息需要处理，会导致优先级低的消息得不到及时处理而积压。当积压到一定程度之后，低优先级的消息可能已经超时，即便后续得到执行机会，由于已经超时也需要丢弃掉，在此之前，它会一直占用优先级队列的堆内存，同时导致客户端业务线程被挂住等待应答消息直到超时，从资源调度层面看，`PriorityQueue` 的算法并不太适合分布式服务框架。

16.4 加权优先级队列

分布式服务框架的服务优先级调度并不是只处理高优先级的消息，而是按照一定比例优先调度高优先级的服务，采用加权优先级队列可以很好地满足这个需求。

加权优先级队列的工作原理如下：它由一系列的普通队列组成，每个队列与服务优先级 1:1 对应。当服务端接收到客户端请求消息时，根据消息对应的服务优先级取值将消息投递到指定的优先级队列中。非法和没有设置优先级属性的消息，投递到默认的优先级队列中。

工作线程按照服务优先级的加权值，按比例从各个优先级队列中获取消息，然后按照优先级的高低将消息设置到工作线程的待处理消息数组中，由于只有本工作线程会读写消息数组，因此该数组是线程安全的。

工作线程顺序从数组中获取消息进行处理，如果为空，则说明消息已经处理完成，需要从优先级队列中重新按比例采集消息，如果所有队列都为空，没有采集到消息，则工作线程同步阻塞，等待新的消息投递进来，唤醒工作线程，重新采集消息进行处理。

加权优先级队列的工作原理如图 16-3 所示。

不同的优先级可以配置不同的加权值，而且可以将加权值注册到配置服务，由服务治理 Portal 动态修改加权值，这样就可以根据线上业务的实际运行状况，实时动态地调整优

优先级调度策略和比例，有效保障服务的 SLA。

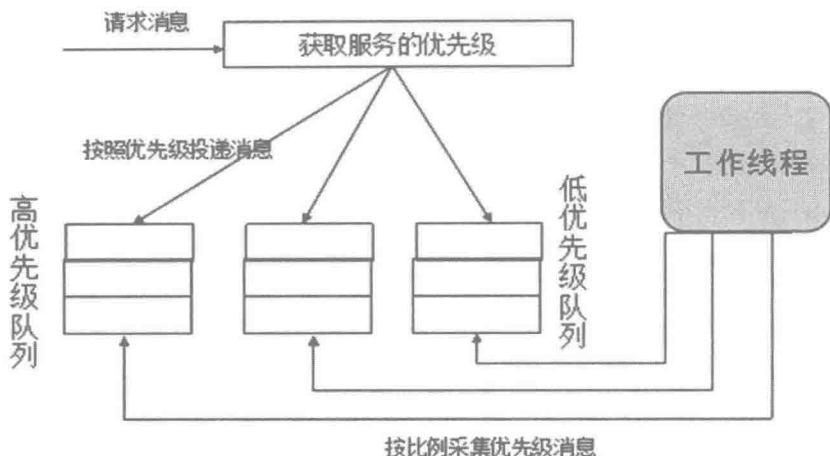


图 16-3 加权优先级队列工作原理

加权优先级队列的主要缺点就是如果优先级等级比较多，对应的优先级队列就会膨胀，如果优先级队列发生积压，这将导致内存占用迅速飙升。为了防止由于优先级等级设置过多带来的优先级队列膨胀，优先级队列与优先级可以按照 $1:N$ 的比例进行设置。单个优先级队列内部再按照优先级的加权值不同做排序，此时就不能采用普通队列来实现，需要自定义实现新的基于加权比重的优先级队列。

16.5 服务迁入迁出

前面介绍的几种优先级调度策略是比较传统的做法，基于服务迁入迁出则是利用分布式服务框架的服务动态发现机制，通过调整服务运行实例数来实现优先级调度。

它的工作原理如下：

- 1) 当系统资源紧张时，通过服务治理 Portal 的服务迁入迁出界面，将低优先级服务的部分运行实例从服务注册中心中迁出，也就是动态去注册。

- 2) 消费者动态发现去注册的服务, 将这部分服务实例的地址信息从路由表中删除, 后续消息将不会路由到已经迁出的服务实例上。
- 3) 由于只迁出了部分服务实例, 被迁出的低优先级服务仍然能够正常处理, 只不过由于部署实例的减少, 得到调度的机会就同比降低了很多, 释放的资源将被高优先级服务使用。通过资源的动态调配, 实现服务的优先级调度。
- 4) 当业务高峰期结束之后, 通过服务治理 Portal 将迁出的服务重新迁入, 低优先级的消息恢复正常执行, 优先级调度结束。

采用服务迁入迁出方案, 是通过运行态的服务治理, 保障服务的优先级调度。相比于前三种方案, 它对性能的影响最小, 灵活度较高。但它的缺点也非常明显, 就是需要人工调整迁入迁出比例来实现资源动态调配, 对于运维人员的经验积累和操作水平都有很高的要求, 自动化程度较低。

16.6 总结

服务的优先级调度与动态流控不同, 流控最终会拒绝消息, 导致部分请求失败。优先级调度是在资源紧张时, 优先执行高优先级的服务, 在保障高优先级服务能够被合理调度的同时, 也兼顾处理部分优先级低的消息, 它们之间存在一定的比例关系。优先级调度本身并不拒绝消息, 但是如果在运行过程中发生了流控, 则由流控负责拒消息。值得注意的是, 通常对于高优先级的管理类消息, 例如心跳消息、指令消息等, 不能被流控掉。

第 17 章

服务治理

随着业务的发展，服务越来越多，如何协调线上运行的各个服务，保障服务的 SLA，对服务架构和运维人员是一个很大的挑战。

随着业务规模的不断扩大，小服务资源浪费等问题逐渐显现，需要能够基于服务调用的性能 KPI 数据进行容量管理，合理分配各个服务的资源占用，提高机器的利用率。

线上业务发生故障时，需要对故障业务做服务降级、流量控制、流量迁移等，快速恢复业务。

随着开发团队的不断扩大，服务的上线越来越随意，甚至发生功能相同、服务名不同的服务同时上线。上线容易下线难，为了规范服务的上线和下线，在服务发布前，需要走服务预发布流程，由架构师或者项目经理对需要上线的服务做发布审核，审核通过的才能够上线。

为了满足服务线下管控、保障线上高效运行，需要有一个统一的服务治理框架对服务进行统一、有效管控，保障服务的高效、健康运行。

17.1 服务治理技术的历史变迁

第一代服务治理 SOA Governance: 以 IBM 为首的 SOA 解决方案提供商推出的针对企业 IT 系统的服务治理框架，它主要聚焦在对企业 IT 系统中异构服务的质量管理、服务发布审批流程管理和服务建模、开发、测试以及运行的全生命周期管理。

第二代以分布式服务框架为中心的服务治理: 随着电商和移动互联网的快速发展，以互联网大厂阿里为首的基于统一分布式服务框架的全新服务治理理念诞生，它聚焦于对内部同构服务的线上治理，保障线上服务的运行质量。相比于传统 IT 架构的服务治理，由于服务的开发模式、部署规模、组网类型、业务特点等差异巨大，因此服务治理的重点也从线下转移到了线上服务质量保障。

微服务架构+云端服务治理: 2013 年至今，随着云计算和微服务架构的发展，以 AWS 为首的基于微服务架构 + 云服务化的云端服务治理体系诞生，它的核心理念是服务微自治，利用云调度的弹性和敏捷，逐渐消除人工治理。

微服务架构可以实现服务一定程度的自治，例如服务独立打包、独立部署、独立升级和独立扩容。通过云计算的弹性伸缩、单点故障迁移、服务健康度管理和自动容量规划等措施，结合微服务治理，逐步实现微服务的自治。

17.1.1 SOA Governance

SOA Governance 的定位：面向企业 IT 系统异构服务的治理和服务生命周期管理，它治理的服务通常是 SOA 服务。

传统的 SOA Governance 包含以下四部分内容。

- 1) 服务建模：验证功能需求与业务需求，发现和评估当前服务，服务建模和性能需求，开发治理规划。
- 2) 服务组装：创建服务更新计划，创建和修改服务以满足所有业务需求，根据治理

策略评估服务，批准组装完成。

- 3) 服务部署：确保服务的质量，措施包括功能测试、性能测试和满足度测试，批准服务部署。
- 4) 服务管理：在整个生命周期内管理和监控服务，跟踪服务注册表中的服务，根据服务质量等级协议（SLA）上报服务的性能 KPI 数据进行服务质量管理。

SOA Governance 工作原理图如图 17-1 所示。

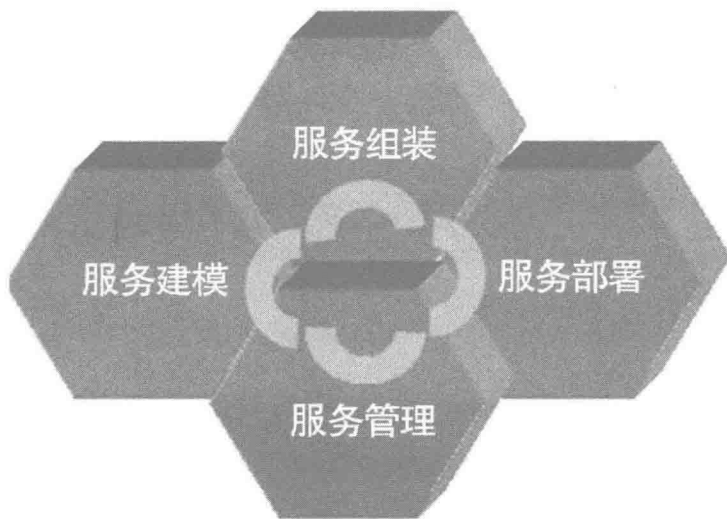


图 17-1 服务治理和最佳实践

传统 SOA Governance 的主要缺点如下：

- 1) 分布式服务框架的发展，内部服务框架需要统一，服务治理也需要适应新的架构，能够由表及里，对服务进行细粒度的管控。
- 2) 微服务架构的发展和业务规模的扩大，导致服务规模量变引起质变，服务治理的特点和难点也随之发生变化。
- 3) 缺少服务运行时动态治理能力，面对突发的流量高峰和业务冲击，传统的服务治理在响应速度、故障快速恢复等方面存在不足，无法更敏捷地应对业务需求。

17.1.2 分布式服务框架服务治理

分布式服务框架的服务治理定位：面向互联网业务的服务治理，聚焦在对内部采用统一服务框架服务化的业务运行态、细粒度的敏捷治理体系。

治理的对象：基于统一分布式服务框架开发的业务服务，与协议本身无关，治理的可以是 SOA 服务，也可以是基于内部服务框架私有协议开发的各种服务。

治理策略：针对互联网业务的特点，例如突发的流量高峰、网络延时、机房故障等，重点针对大规模跨机房的海量服务进行运行态治理，保障线上服务的高 SLA，满足用户的体验。常用的治理策略包括服务的限流降级、服务迁入迁出、服务动态路由和灰度发布等。

以阿里开源的分布式服务框架 Dubbo 为例，它的服务治理体系如图 17-2 所示。

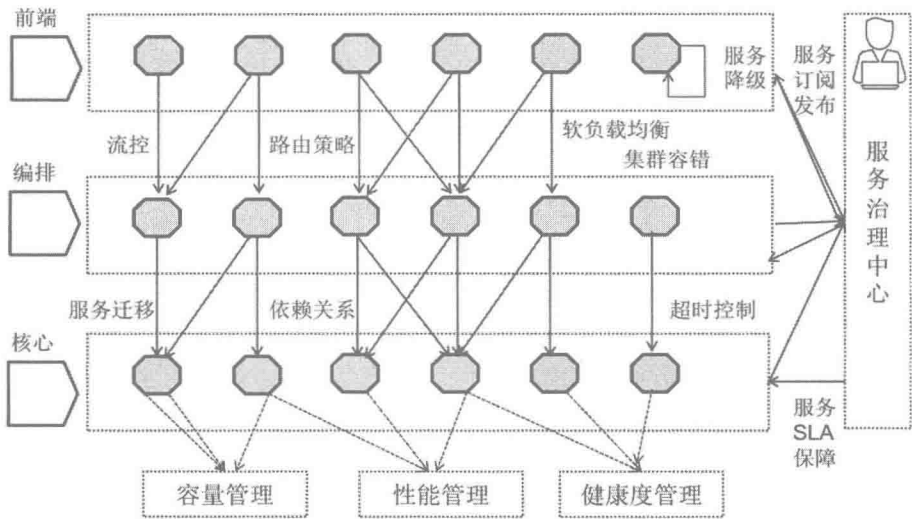


图 17-2 Dubbo 服务治理

17.1.3 AWS 云端微服务治理

随着云计算的发展，Dev&Ops 逐渐流行起来，基础设施服务化（IaaS）为大规模、批

量流水线式软件交付提供了便利，AWS 作为全球最大的云计算解决方案提供商，在微服务云化开发和治理方面积累了非常多的经验，具体总结如下。

- 1) 全公司统一服务化开发环境，统一简单化服务框架（Coral Service），统一运行平台，快速高效服务开发。
- 2) 所有后端应用服务化，系统由多项服务化组件构成。
- 3) 服务共享、原子化、重用。
- 4) 服务由小研发团队（2 Pizza Team）负责服务开发、测试、部署和治理，运维整个生命周期支撑。
- 5) 高度自动化和 Dev&Ops 支持，一键式服务部署和回退。
- 6) 超大规模支持：后台几十万个服务，成千上万开发者同时使用，平均每秒钟有 1~2 个服务部署。
- 7) 尝试基于 Docker 容器部署微服务。
- 8) 服务治理是核心：服务性能 KPI 统计、告警、服务健康度管理、灵活的弹性伸缩策略、故障自动迁移、服务限流和服务降级等多种治理手段，保障服务高质量运行。

17.2 应用服务化后面临的挑战

应用服务化之后，用户在享受服务化带来的分层解耦、服务重用等收益的同时，也面临着巨大的挑战。主要包括：

- 1) 多团队协作问题。
- 2) 服务安全问题。
- 3) 线下服务管控。

4) 线上服务治理，保障服务 SLA。

5) 其他……

本节我们将对应用服务化之后面临的主要挑战进行梳理，推导出服务治理架构。

17.2.1 跨团队协作问题

随着应用规模不断扩大，研发团队通常按照业务领域或者应用分层划分出不同的研发小组，每个小组负责领域内服务开发。

服务多了之后，就会面临沟通成本问题。例如 A 小组和 B 小组同时依赖 C 团队的 S1 和 S2 服务，两个团队的需求优先级都很高，C 团队在进行需求优先级排序的时候，就会面临很大的挑战，究竟该把哪个服务的优先级排在前面？

跨团队服务调用失败，消费方需要找服务提供者进行问题定位和联调，由于服务太多且跨团队，如何快速找到服务提供者，需要到处问人，沟通成本高。

当前系统提供了哪些服务，服务接口定义和参数是什么，服务使用示例，注意事项和约束是什么，消费者很难搞清楚，需要找服务提供者进行咨询，效率低下，服务提供者天天回答类似问题，不胜其烦。

为方便开发测试，经常会在线下共用一个所有服务可用的注册中心，这时，如果一个正在开发中的服务提供者注册，可能会影响消费者，不能正常运行。如果有两个镜像环境，两个注册中心，有一个服务只在其中一个注册中心有部署，另一个注册中心还没来得及部署，而两个注册中心的其他应用都需要依赖此服务，所以需要将服务同时注册到两个注册中心，但却不能让此服务同时依赖两个注册中心的其他服务。

开发完成之后，需要按照真实业务流程进行联调，原子服务、公共服务往往会同时被多个消费者消费，考虑到成本，开发态测试往往共用一套镜像测试环境。消费者 A 和服务提供者 S 进行联调，存在两个问题：

1) 服务提供者 S 分布式部署，存在多个服务实例，如果做断点调试，路由模块会动

态分发消息，随机路由，服务提供者 S 无法确定要连接的 IP 地址。

- 2) 如果打断点，可能其他消费者也正在进行服务调用，调试会被干扰，需要通知所有的开发者不要调用服务 S，显然不可能。

17.2.2 服务的上下线管控

因为发布服务很简单，服务的上线越来越随意，有时候负责服务化的架构师都不知道有人上线了某个服务，使得线上服务鱼龙混杂，甚至出现重复的服务，而服务下线比上线还困难。

因为业务调整等原因，需要结束某些服务的生命周期，服务提供者直接将服务下线，导致依赖该服务的应用不能正常工作。服务下线时，应先标识为过时，然后通知调用方尽快修改调用，通过性能 KPI 接口和调用链分析，确认没有消费者再调用此服务，才能下线。

上线审批、下线通知机制的建立，可以对服务的生命周期做有效治理。

17.2.3 服务安全

数据层服务化之后，对于同组所有的消费者而言都是可见的。一些敏感数据的访问需要授权，不能随意调用。

数据服务开放之后，有授权的消费者可以直接访问，如果消费者数量过多，或者某些消费者频繁调用数据服务，导致数据服务压力过大而不可用，核心的数据服务一挂，影响一大片，人心惶惶。

针对内部应用，服务框架通常采用长链接管理客户端连接，针对非信任的第三方应用，或者恶意消费者，需要具备黑白名单访问控制机制，防止客户端非法链路过多，占用大量的句柄、线程和缓存资源，影响服务提供者的运行质量。

17.2.4 服务 SLA 保障

随着服务的不停升级，总有些意想不到的事发生，比如缓存被穿透导致数据库压力过大而宕机。对于线上服务而言，故障总是不可避免的，每次核心服务一挂，影响一大片，如何控制故障的影响面对用户而言是个巨大的挑战。

大促等业务高峰时，系统资源成为瓶颈。需要对非核心类服务，例如用户评论、粉丝管理、积分管理等服务做限制，保障核心服务的正常运行。由于非核心服务跟系统其他服务打包部署在同一个 Tomcat 等容器进程中，一旦把进程停了，也影响其他合设的服务，如何高效地关停非核心服务，但又不影响其他合设的服务，需要服务治理框架统一考虑。

不同的服务，处理速度差异很大，例如数据库的批量操作和单条操作，系统在上线之初，通常会按照经验值统一设置一个超时时间。但是在线上运行一段时间之后，需要对各服务的超时时间进行个性化调整。如何方便地在线可视化修改服务的超时时间，不需要重启即可动态生效，对提升系统的运维效率有很大帮助。

17.2.5 故障快速定界定位

服务化之前，接口之间的调用通常都是本地方法调用，只需要找到故障节点，所有的故障信息都可以在同一个节点进行采集，故障定界和定位手段尽管比较单一，但是可以满足日常的运维需求。

服务化之后，一个业务流程，底层可能涉及上百个服务调用，任何一个服务发生故障，都有可能导致业务不可用。由于服务分布式部署，部署在成百上千台机器上，如果仍然采用原来的故障定位手段，效率将会非常低下。

分布式服务化之后，故障快速定界和定位面临的挑战如图 17-3 所示。

如果不能有效解决服务化和大规模分布式部署之后给故障定界定位带来的困难，服务的 SLA 将很难有效保障，服务化带来的价值也将大打折扣。

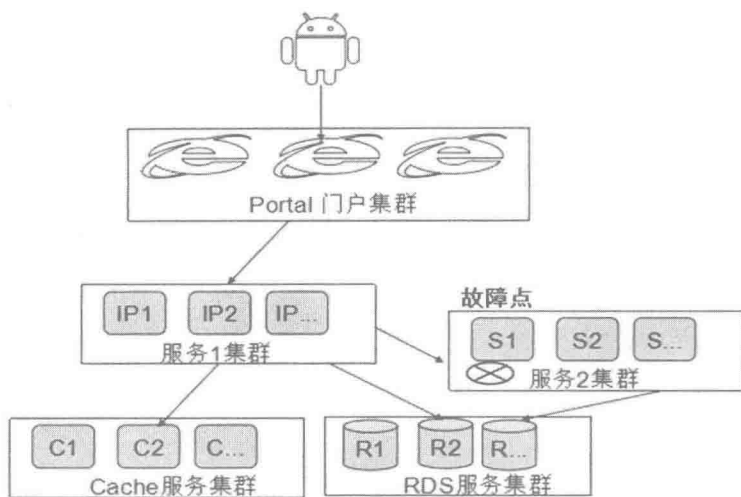


图 17-3 分布式故障定位

17.3 服务治理

基于分布式服务框架的服务治理目标如下。

- 1) 防止业务服务架构腐化：通过服务注册中心对服务强弱依赖进行分析，结合运行时服务调用链关系分析，梳理不合理的依赖和调用路径，优化服务化架构，防止代码腐化。
- 2) 快速故障定界定位：通过 Flume 等分布式日志采集框架，实时收集服务调用链日志、服务性能 KPI 数据、服务接口日志、运行日志等，实时汇总和在线分析，集中存储和展示，实现故障的自动发现、自动分析和在线条件检索，方便运维人员、研发人员进行实时故障诊断。
- 3) 服务微管控：细粒度的运行期服务治理，包括限流降级、服务迁入迁出、服务超时控制、智能路由、统一配置、优先级调度和流量迁移等，提供方法级治理和动态生效功能，通过一系列细粒度的治理策略，在故障发生时可以多管齐下，在线

调整，快速恢复业务。

- 4) 服务生命周期管理：包括服务的上线审批、下线通知，服务的在线升级，以及线上和线下服务文档库的建设。

17.3.1 服务治理架构设计

服务治理是分布式服务框架的一个可选特性，尽管从服务开发和运行角度看它不是必需的，但是如果没有服务治理功能，分布式服务框架的服务 SLA 很难得到保障，服务化也很难真正实施成功。

从架构上看，分布式服务框架的服务治理分为三层，如图 17-4 所示。

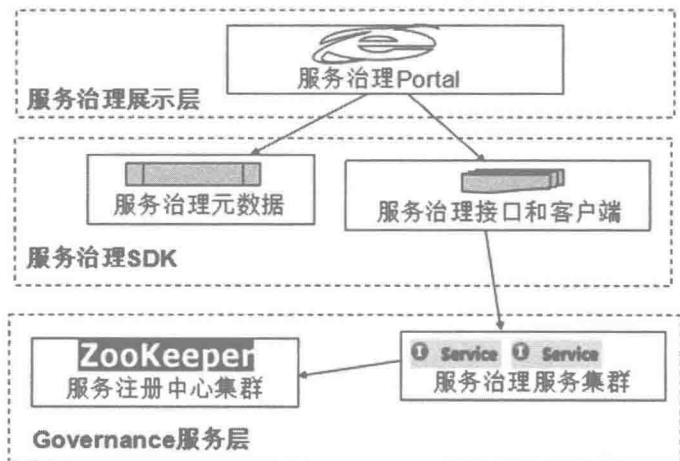


图 17-4 服务治理逻辑架构图

第 1 层为服务治理展示层，它主要由服务治理 Portal 组成，提供可视化的界面，方便服务运维人员进行治理操作。

考虑到服务集群规模和微服务化之后，服务数量非常庞大，服务治理 Portal 通常会提供多维度的检索功能，方便用户操作，例如基于服务名、应用名和主机 IP 地址等的条件检索。

以阿里开源的分布式服务框架 Dubbo 为例，服务治理界面如图 17-5 所示。



图 17-5 Dubbo 服务治理 Portal

第 2 层为服务治理 SDK 层，它主要由如下几部分组成。

- 1) 服务治理元数据：服务治理元数据主要包括服务治理实体对象，包括服务模型、应用模型、治理组织模型、用户权限模型、数据展示模型等。元数据模型通过 Data Mapper 和模型扩展，向上层界面屏蔽底层服务框架的数据模型，实现展示层和服务框架的解耦，元数据也可以用于展示界面的定制扩展。
- 2) 服务治理接口：服务治理 Portal 调用服务治理接口，实现服务治理。例如服务降级接口、服务流控接口、服务路由权重调整接口、服务迁移接口等。服务接口与具体的协议无关，它通常基于分布式服务框架自身实现，可以是 Restful 接口，也可以是内部的私有协议。
- 3) 服务治理客户端类库：由于服务治理服务本身通常也是基于分布式服务框架开发，因此服务治理 Portal 需要集成分布式服务框架的客户端类库，实现服务的自动发现和调用。
- 4) 调用示例：客户端 SDK 需要提供服务治理接口的参数说明、注意事项以及给出常用的调用示例，方便前端开发人员使用。

5) 集成开发指南：服务治理 SDK 需要提供集成开发指南，指导使用者如何在开发环境中搭建、集成和使用服务治理 SDK。

第 3 层为后台服务治理服务层，它通常由一组服务治理服务组成，可以单独部署，也可以与应用合设。考虑到健壮性，通常选择独立集群部署。治理服务的可靠性由分布式服务框架自身来保证，治理服务宕机或者异常，不影响业务的正常使用。服务治理服务通常并不随服务框架发布，治理服务是可选的插件，单独随服务治理框架交付。

服务治理框架可以独立打包、部署和交付，它的交付件通常包括两部分：服务治理 war 包和服务治理 SDK，一个用于开发和测试，一个用于运行态运维。

17.3.2 运行态服务治理功能设计

运行态服务治理首先要能够做到可视：当前系统发布了哪些服务，这些服务部署在哪些机器上，性能 KPI 数据如何，指标是否正常等。

除了告警，对服务健康状态和关键指标的日常巡检也非常重要。服务的发布情况和运行状态是首先需要关注的，如图 17-6 所示。

请输入服务名或接口名		搜索服务	
服务名/接口名	服务版本	服务组	应用
org.netty.demo.cache.CacheService	1.0.0	dsdp	EchoDemo
org.netty.demo.db.SqlClientService	1.0.0	netty	EchoDemo
org.netty.demo.echo.EchoService	1.0.0	netty	EchoDemo

图 17-6 服务列表

通过服务列表查询功能，可以方便地查看当前系统发布了哪些服务，以及服务的版本信息和分组情况。除了能够查看服务的接口信息，通常还需要能够实时查看服务的接口参数，通过 Java DOC 等线下模式，可能会发生因 Java DOC 没有及时更新而与线上服务不一致的情况，因此线上通过服务治理的服务列表查看最准确，如图 17-7 所示。

大数据计算，将性能指标入库。服务治理服务查询性能指标库，获取相关性能数据给前端服务治理 Portal，服务治理 Portal 获取到性能数据之后，按照展示要求进行报表展示，展示界面如图 17-9 所示。

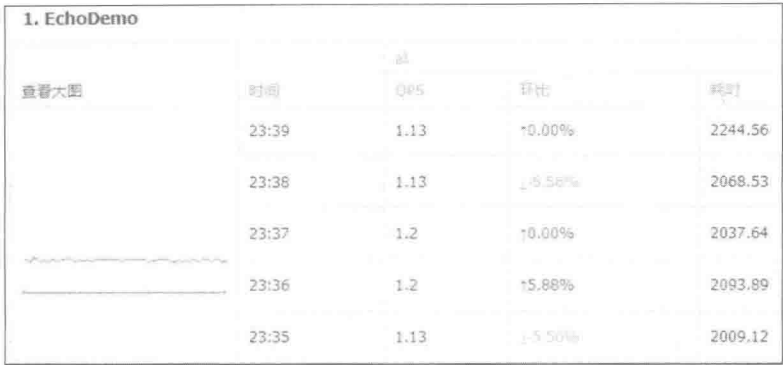


图 17-9 服务调用性能 KPI 数据展示

为了定位问题方便，服务的性能 KPI 数据需要精确到服务接口方法级，如图 17-10 所示。

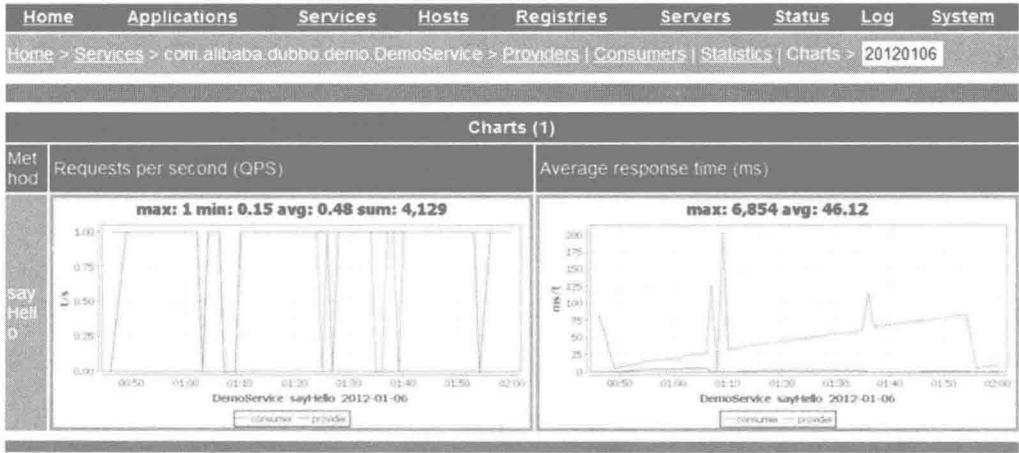


图 17-10 服务方法级性能 KPI 数据（以 Dubbo 为例）

由于性能 KPI 数据的统计周期、统计指标和报表呈现方式差异比较大，因此服务框架很难抽象出一套放之四海而皆准的性能统计功能，因此在设计的时候需要注意以下两点。

- 1) 扩展性：服务性能 KPI 数据采集由插件 Handler 负责，平台和业务均可以通过扩展性能统计插件 Handler 的方式扩展采集指标和采集周期等。
- 2) 原子性：服务提供者和消费者只负责原始数据的采集和上报，不在本节点内做复杂的汇总操作，汇总和计算由性能汇聚节点的 Spark 等大数据流式框架负责。

解决了服务的可视问题之后，我们需要对服务的可管进行设计。在大促等业务高峰期，系统资源往往会成为瓶颈，我们需要对非核心服务做限流来保障核心服务，限流常用的策略就是基于服务调用数 QPS 或者并发执行线程数，针对某个服务或者方法进行流量控制，它的治理界面如图 17-11 所示。

添加限流规则

当前应用: EchoDemo

* 限流类型: HSF限流

* 需要限流的接口: org.netty.demo.cache.CacheService

* 需要限流的方法: query(java.lang.String)

* 被限流的应用: 所有

* 限流粒度: Thread限流

* 限流阈值: 100

确定 取消

图 17-11 添加限流规则

针对不同的服务，可以添加不同的限流规则，如图 17-12 所示。

需要降级的资源	被限流的应用	限流粒度	限流阈值	状态	操作
/service/index.html	所有	QPS限流	800	已启用	编辑 停用 删除
EchoService:echo(java.lang.String)	所有	QPS限流	100	已停用	编辑 启用 删除
SqlClientService:insert(java.lang.String,java.lang. ...	所有	Thread限流	5	已启用	编辑 停用 删除
CacheService:remove(java.lang.String)	所有	QPS限流	500	已启用	编辑 停用 删除
CacheService:query(java.lang.String)	所有	Thread限流	50	已启用	编辑 停用 删除
SqlClientService:query(java.lang.String,java.lang. ...	所有	QPS限流	1000	已启用	编辑 停用 删除
SqlClientService:delete(java.lang.String)	所有	QPS限流	100	已启用	编辑 停用 删除

图 17-12 被限流的服务列表

限流规则添加完成之后，由服务治理服务修改注册中心，注册中心将流控变更信息通知到集群中所有服务提供者和消费者，不需要重启应用即可动态生效。

在实践中，流控的阈值通常是个经验值，设置之后根据性能 KPI 数据决定是否需要进行二次优化，当流量高峰期平稳渡过之后，可以停用流控功能，避免因流控导致业务失败率提升。

当某些非核心服务不可用时，往往需要在业务侧做功能放通，防止业务失败，影响用户体验。例如手机阅读的用户鉴权服务发生故障不可用，如果直接返回鉴权失败，已付费的用户就无法正常阅读，引起客户投诉。此时可以对用户鉴权做放通处理，根据用户的实际消费情况做后计费话单，事后再补扣。

另外一个场景就是大促或者营销期间，希望强制对某些非核心服务做降级，不允许用户访问，将宝贵的资源用在核心服务保障上，此时尽管没有发生故障，我们依旧希望能够根据业务实际运行情况对某些服务做强制降级。

服务降级对于保障核心业务的正常运行，提升用户满意度非常重要，它通常包括容错降级和强制降级两种，服务降级的治理界面如图 17-13 所示。

提供者 消费者 应用 路由规则 动态配置 访问控制 权重调节

返回

服务名: com.alibaba.dubbo.demo.DemoService

消费者应用名:

只推送给指定消费者地址: 不填表示对消费

状态: 禁用

动态配置

参数名: 参数值: 方法级配置如: findP

新增参数

服务降级

所有方法的Mock值: 容错

新增方法

保存

图 17-13 服务降级配置界面（以 Dubbo 为例）

服务降级之后，可以通过服务状态检测界面查看服务的运行状态，如图 17-14 所示。

提供者 消费者 应用 路由规则 动态配置 访问控制 权重调节 负载均衡

☒ 批量屏蔽 ☒ 批量容错 ☒ 批量恢复 ☒ 缺省屏蔽 ☒ 缺省容错 ☒ 缺省恢复

☐ 应用名: 角色: 所有 降级: 所有 操作: 降级: 所有

<input type="checkbox"/>	demo-consumer	消费者	未降级	<input checked="" type="checkbox"/> 屏蔽 <input checked="" type="checkbox"/> 容错	未降级
<input type="checkbox"/>	demo-provider	提供者			

图 17-14 服务降级状态查询（以 Dubbo 为例）

服务降级是可逆的，当系统恢复正常或者故障排除之后，可以对已经降级的服务进行恢复操作，恢复之后消费者将正常调用服务提供者，不再执行降级逻辑。

当系统发生故障时，有时候需要对故障流量做隔离，防止故障扩散到整个集群，此时动态调整路由规则就显得至关重要。

通过服务提供者的 IP 地址进行条件过滤，可以将故障流量引流到特定的几台机器上，

便于故障隔离和故障定位。也可以对一些异常的消费者做黑白名单控制，屏蔽一些消费者对系统稳定性带来的冲击，动态路由规则配置界面如图 17-15 所示。

提供者 消费者 应用 路由规则 动态配置

返回

路由名称: *

优先级: 0

服务名: * com.alibaba.dubbo.demo.DemoService

方法名: 请选择

匹配条件 匹配 不匹配

消费者IP地址: 消费者应用名: 消费者集群:

过滤规则 匹配 不匹配

提供者IP地址: 提供者集群: 提供者协议: 提供者端口:

保存

图 17-15 动态路由规则配置（以 Dubbo 为例）

与流控类似，路由规则也需要支持方法级的配置，在一些场景中，服务的不同方法需要使用不同的治理策略，如果只支持服务级统配，效果会大打折扣。

路由规则动态修改的设计原理是服务治理 Portal 将服务名、方法名、路由规则等信息发送给后台的路由规则变更服务，路由规则变更服务将路由规则变更信息写入服务注册中心，由服务注册中心将变更后的路由规则通知给集群的服务提供者和消费者。服务提供者和消费者更新服务注册缓存信息，消费者在路由的时候执行新的路由规则，选择对应的服务提供者发送消息。

除了以上介绍的常用服务治理功能，线上服务治理还包括了一些其他功能，例如：

- 1) 服务的超时控制。
- 2) 服务的优先级调度。

- 3) 服务的负载均衡策略调整。
- 4) 服务分组调整。
- 5) 其他……

在实际工作中，我们可以根据分布式服务框架的定位和用户的实际需求，提供更多实用的线上治理功能，保障业务的平稳运行。

17.3.3 线下服务治理

随着服务化的推进，服务的开发者越来越多，服务之间的互相调用和依赖也日趋复杂，经常会出现开发者 A 等开发者 B 提供服务接口的情况。另一方面，搭建一套线下的全量环境成本非常高，希望能够直接在 IDE 工具中进行依赖 Mock 和本地仿真测试环境搭建，开发者通过安装 IDE 插件就能够快速构建测试仿真环境，提供 Mock 框架，进行线下本地仿真测试，避免过度依赖服务提供者的开发进度。

以往跨团队协作，往往需要多方的 PM 或者 PL 拿着本团队内的需求迭代计划进行核对，由于各个团队的需求优先级不同，服务间的依赖关系在项目初期也很难梳理清楚，尽管经历多次需求核对，耗时耗力，但是仍然会发生团队之间互相等待的情况。应用的最终交付需要依赖多个团队提供的服务，交付的速度取决于最晚提供服务的团队。这种人工排需求的方式很不精确，也容易拖业务敏捷交付的后腿。业务需要一个需求管理引擎，服务拆分之后，通过流程图拖拽或者配置的方式，将流程 E2E 通过图形化工具展示出来，只需要在应用最前端的首节点设置应用交付日期，引擎自动计算下游各个服务提供者的交付日期，排出迭代计划。利用需求管理工具实现服务接口的开发迭代计划排序，可以避免人为失误导致的需求延期，有效提高跨团队的协作效率。

相对于平台产品，业务服务的升级和修改非常频繁，传统依靠 Java DOC 进行接口说明和传递的方式，往往会因为缺乏文档建设或 API DOC 没有及时刷新，导致消费者拿到的接口定义说明不准确。相比于没有文档，拿到过时、错误的 API DOC 文档对使用者的危害更大。

为了解决这个问题，需要建立服务文档中心，方便线上运维人员查看和多团队之间的协作，它的工作原理如图 17-16 所示。

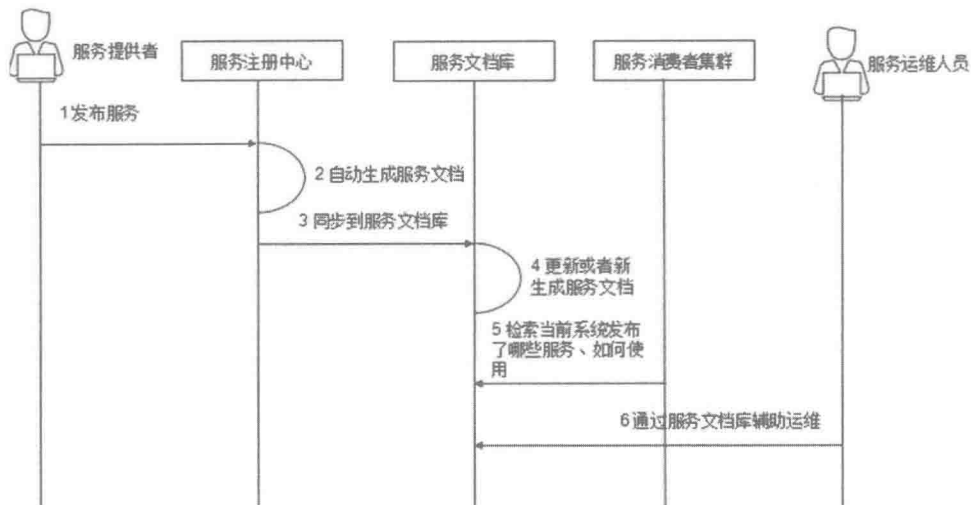


图 17-16 服务文档库工作原理

基于 Java DOC 工具进行扩展，将规则内置到 IDE 开发模板中，并通过 CI 构建工具做编译检测，将不符合要求的服务接口输出到 CI 构建报告中并邮件发送给服务责任人，督促相关责任人进行整改。

当团队规模扩大之后，会划分成平台基线组、业务定制组等不同研发团队，一些团队甚至跨多地协同开发和运维。服务的上线和下线必须要严格管控起来，一旦不合格的服务上线并被消费者消息，再想下线就非常困难了。

对于需要下线的服务管控也很重要，有些服务虽然调用频次不高，业务量也不大。但是如果贸然下线，很有可能导致依赖它的消费者业务调用失败，这会违反服务的 SLA 协定，给服务提供商造成损失。

服务的上线审批、下线通知机制需要建立并完善起来，它的工作原理如图 17-17 所示。

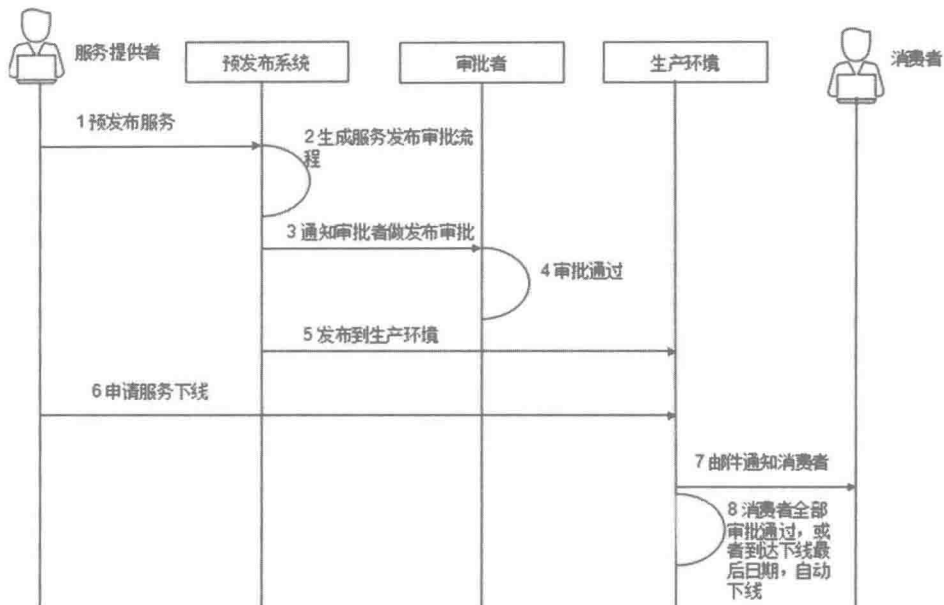


图 17-17 服务上线审批、下线通知流程

除了以上介绍的常用服务治理措施，线下服务治理还包括：

- 1) 业务的梳理、服务划分原则和方法论。
- 2) 服务跨团队协作流程、准则、工具和方法论。
- 3) 服务的接口兼容性原则和规范。
- 4) 其他……

线下服务治理依团队和业务不同，需求也不同，需要业务团队和服务框架团队长期梳理、实践和优化，才能够提升线下服务治理的效率，它的建设是个长期过程，并非一蹴而就。

17.3.4 安全和权限管理

分布式服务框架的安全涉及到两个层面：

- 1) 服务的开放和鉴权机制。
- 2) 服务治理的安全和权限管理。

本节我们针对服务治理的安全和权限管理，进行详细讲解。服务治理的使用者通常分为以下三类。

- 1) 开发或者测试：主要用于问题定位，或者协助运维人员做服务治理。
- 2) 运维人员：主要用于日常运维巡检，查看各种服务性能 KPI 是否正常，是否有告警；当发生问题时，利用服务治理进行故障恢复。
- 3) 管理者：主要关心运营层面的 KPI 数据，只看不管。

不同角色能够看到的界面可以不同，这样就可以屏蔽掉一些复杂的细节，降低服务治理的使用门槛。也可以是同一个视图，但是在视图上加读写权限，实现不同角色间的权限控制。

服务治理框架需要提供角色和权限模型，用于用户定义角色并进行授权，同时要提供账号和角色关联管理功能，方便对账号授权。

角色管理界面示例如图 17-18 所示。

角色管理			添加角色
角色	权限个数	操作	
所有权限	39个	查看权限	管理权限 删除
应用管理者	11个	查看权限	管理权限 删除
服务运维者	22个	查看权限	管理权限 删除
客户CEO	1个	查看权限	管理权限 删除

图 17-18 角色管理界面

用户可以自定义角色，并给角色授权。需要指出的是，角色不能跨级授权，例如 A 账号是应用管理者角色，那他创建的角色权限不能高于应用管理者权限，最多也是 11 个，

与 Oracle 数据库的角色和授权机制类似。

用户可以删除自己创建的角色，非自己创建的角色需要有主账号管理员权限才能删除。授权之后，也可以根据实际情况对权限进行修改。

权限管理首先需要对服务治理框架进行权限建模，划分出具体的权限，例如查看服务列表权限、在线浏览日志权限、服务降级权限等，它的划分示例如图 17-19 所示。



图 17-19 权限管理界面

针对不同的角色，可以灵活进行授权，如图 17-20 所示。

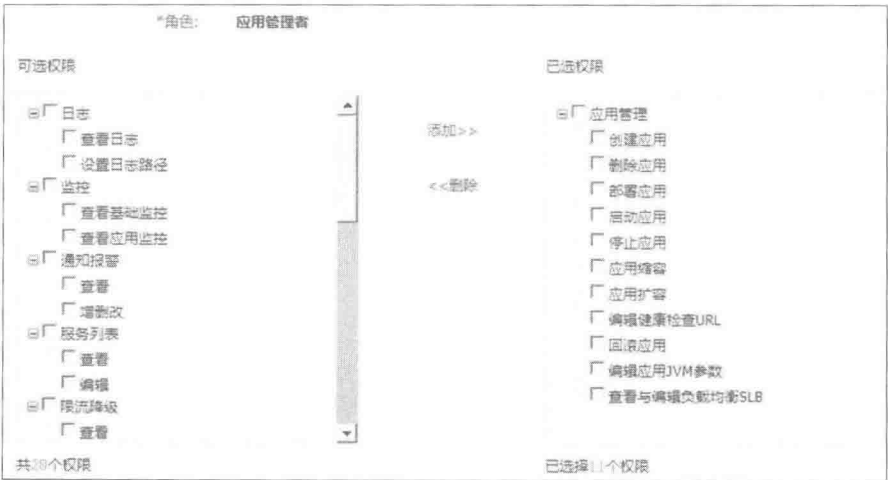


图 17-20 对角色进行授权

角色定义和授权完成之后，就可以给用户账号指定角色，进行权限控制，如图 17-21 所示。



图 17-21 对角色进行授权

完成角色和用户绑定之后，就可以依据权限对服务进行治理，例如配置服务的弹性伸缩策略，如图 17-22 所示。

扩容规则: ☒

触发指标: CPU \geq 60 % RT \geq 100 ms Load \geq 70

触发条件: 任一指标

持续时间超过: 1 分钟

每次扩容的实例数: 1 台

最大实例数: 2 台

图 17-22 配置弹性伸缩策略

对于非授权的操作，可以直接弹出“无操作权限”提示框，阻止客户相关操作。也可以直接把相关界面屏蔽掉，对用户不可见。

17.4 总结

服务治理涵盖的范围非常广，很多治理手段也需要业务在实际开发中积累和沉淀，服

务治理并没有统一的标准，这就是实施服务治理的困难之处。

服务治理总体结构图如图 17-23 所示。

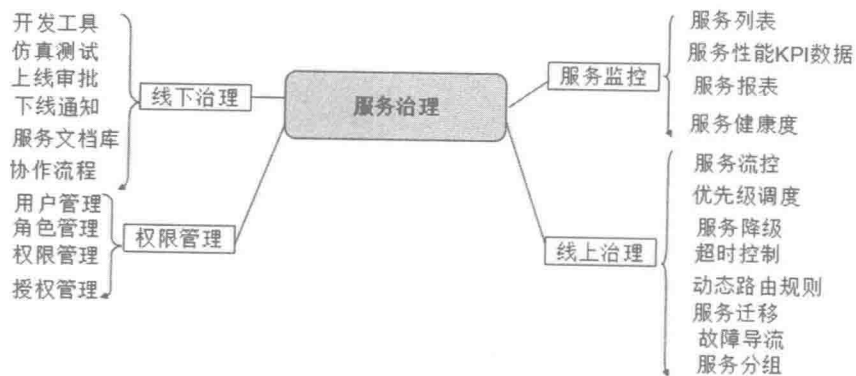


图 17-23 服务治理结构图

分布式消息跟踪

随着业务分布式架构的发展，系统间的系统调用日趋复杂，以电商的商品购买为例，前台界面的购买操作涉及到底层上百次服务调用，涉及到的中间件包括：

- ◎ 分布式服务框架
- ◎ 消息队列
- ◎ 分布式缓存
- ◎ 分布式数据访问中间件
- ◎ 分布式文件存储系统
- ◎ 分布式日志采集
- ◎ 其他……

如果无法有效清理后端的分布式调用和依赖关系，故障定界将会非常困难。利用分布式消息跟踪系统可以有效解决服务化之后系统面临的运维挑战，提高运维效率。

18.1 业务场景分析

随着系统内部的服务增多，一次业务调用可能会通过系统内部多个服务协同调用来完成，这些服务可能由不同的团队开发，并且分布在不同的 VM 节点，甚至可能在多个地域不同的机房内。因此分布式系统需要有一种方式来直观地了解系统的调用及运行状况，测量系统的运行性能，方便准确地指导系统的优化及服务化改进，分布式调用示意图如图 18-1 所示。

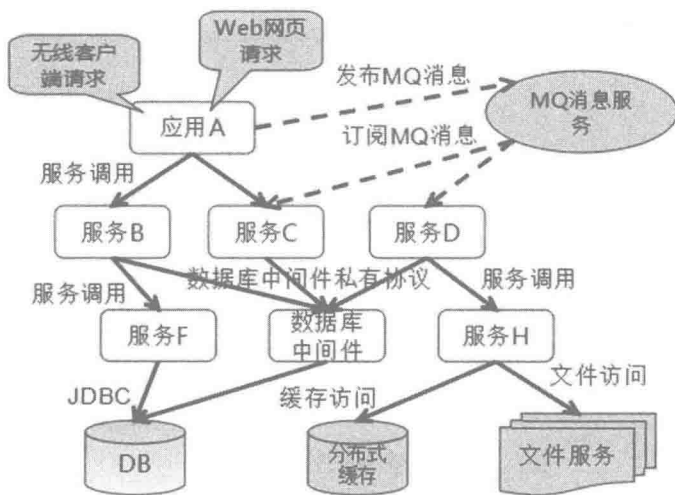


图 18-1 应用分布式调用关系图

18.1.1 故障的快速定界定位

传统应用软件发生故障时，往往通过接口日志手工从故障节点采集日志进行问题分析定位，分布式服务化之后，一次业务调用可能涉及到后台上百次服务调用，每个服务又是集群组网，传统人工到各个服务节点人肉搜索的方式效率很低。

希望能够通过调用链跟踪，将一次业务调用的完整轨迹以调用链的形式展示出来，通过图形化界面查看每次服务调用结果，以及故障信息，提升故障的定位效率。

利用调用链进行快速故障定界示意图如图 18-2 所示。

调用链详情							
调用链TraceId:		781859e714475165380775802d5a2f			搜索		
应用名	IP	类型	状态	大小	服务/方法	时间戳	耗时
EchoDemo	10.169.98.177	UNKNOWN	UNKNOWN	0B	from://8e2ae7d4-767a-4e34-b0dd-af3e7d2554d2		0ms
EchoDemo	10.169.98.177	HSF	OK	0B	org.netty.demo.echo.EchoService@echo~S		502ms
EchoDemo	10.169.98.177	HSF	OK	0B	org.netty.demo.cache.CacheService@save~O		0ms
EchoDemo	10.169.98.177	HSF	OK	0B	org.netty.demo.cache.CacheService@remove~S		0ms
EchoDemo	10.169.98.177	HSF	OK	0B	org.netty.demo.db.SqlClientService@query~SO		501ms

图 18-2 故障快速定界

通过在业务日志中增加调用链 ID，可以实现业务日志和调用链的动态关联。通过调用链进行快速故障定界，然后通过 ID 关联查询，可以快速地定位到业务日志相关信息，方便问题定位。

18.1.2 调用路径分析

通过对调用链调用路径的分析，可以识别应用的关键路径：应用被调用得最多的入口、服务是哪些，找出服务的热点、耗时瓶颈和易故障点，评估最大的风险点，针对性改进以预防风险。同时为性能优化、容量规划等提供数据支撑，示例如图 18-3 所示。

应用	QPS	峰值	调用次数	平均耗时 (MS)	本地耗时 (MS)	依赖度 (%)	标记
bookorder	166.12	588.98	10	320	29	100	易故障
itemcenter	208.39	656.87	16	10	10	83.33	瓶颈点
payment	160.18	580.32	12	18	18	92.28	强依赖
delivery	159.21	578.69	11	9	8	91.66	强依赖
cachenode	153.02	550.52	10	6	5	97.68	强依赖
billcenter	148.89	510.78	8	120	61	60.49	

图 18-3 调用路径分析

18.1.3 调用来源和去向分析

通过调用去向分析，可以对服务的依赖关系进行有效梳理：

- 1) 应用直接和间接依赖了哪些服务。
- 2) 各层次依赖的调用时延、QPS、成功率等性能 KPI 指标。
- 3) 识别不合理的强依赖，或者冗余依赖，反向要求开发进行依赖解耦和优化。

通过对调用来源进行 Top 排序，可以识别当前服务的消费来源，以及获取各消费者的 QPS、平均时延、出错率等，针对特定的消费者，可以做针对性治理，例如针对某个消费者的限流降级、路由策略修改等，保障服务的 SLA。调用来源分析示意图如图 18-4 所示。

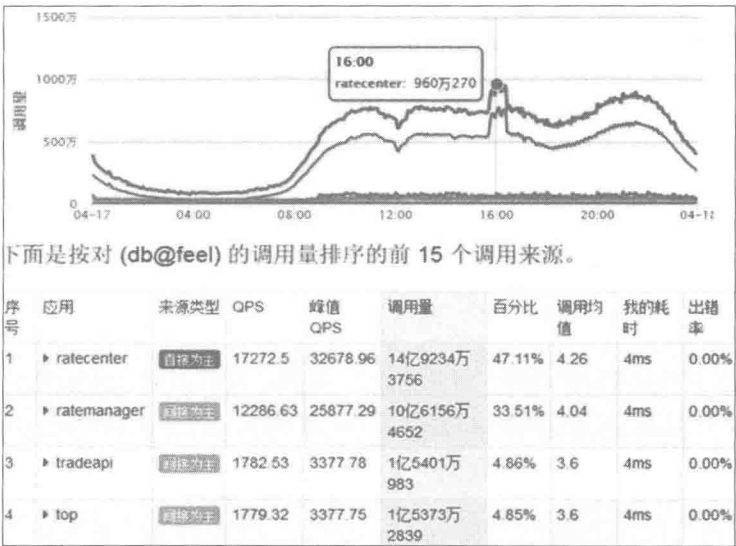


图 18-4 调用来源分析

18.2 分布式消息跟踪系统设计

分布式消息跟踪系统的核心就是调用链，每次业务请求都生成一个全局唯一的 TraceID，

通过跟踪 ID 将不同节点间的日志串接起来，形成一个完整的日志调用链，通过对调用链日志做实时采集、汇总和大数据分析，提取各种维度的价值数据，为系统运维和运营提供大数据支撑。

18.2.1 系统架构

分布式消息跟踪系统的整体架构由四部分组成，如图 18-5 所示。

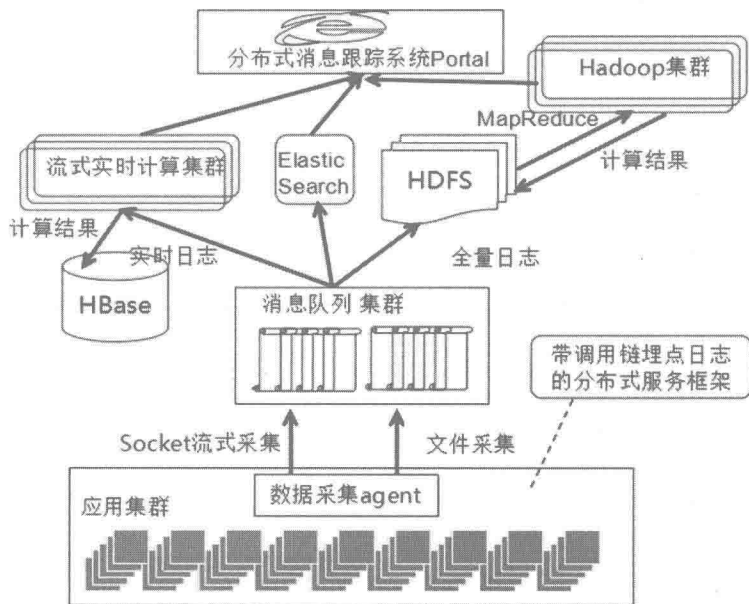


图 18-5 分布式消息跟踪架构图

- 1) 调用链埋点日志生成。
- 2) 分布式采集和存储埋点日志。
- 3) 在线、离线大数据计算，对调用链数据进行分析 and 汇总。
- 4) 调用链的界面展示、排序和检索等。

18.2.2 埋点日志

埋点就是分布式消息跟踪系统在当前节点的上下文信息，埋点可以分为两类：

- 1) 客户端埋点，客户端发送请求消息时生成的调用上下文，通常包括 TraceID、调用方 IP、调用方接口或者业务名称、调用的发起时间、被调用的服务名、方法名、IP 地址和端口等信息。
- 2) 服务端埋点，服务端返回应答消息时在当前节点生成的上下文，包括 TraceID、调用方上下文信息、服务端处理的耗时、处理结果等信息。

埋点日志的实现，通常会包含如下几个功能：

- 1) 埋点规范，主要用于业务二次定制开发和第三方中间件/系统对接。
- 2) 埋点日志类库，服务生成埋点上下文，打印埋点日志等。
- 3) 中间件预置埋点功能，应用不需要开发任何业务代码即可直接使用，也可以通过埋点类库将应用自身的业务字段携带到调用链上下文中，例如终端类型、手机号等。

埋点日志的生成规则如图 18-6 所示。

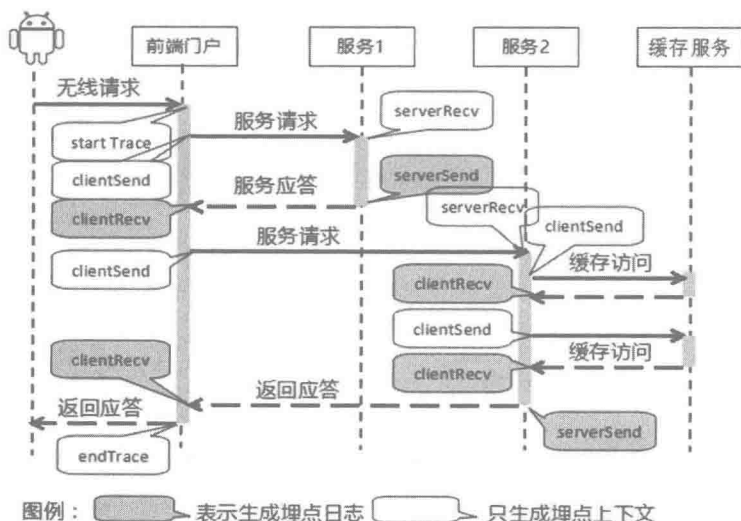


图 18-6 埋点日志生成原理

消息跟踪 ID 通常由调用首节点负责生成（各种门户 Portal），本 JVM 之内通过线程上下文传递 TraceID，跨节点传递时，往往通过分布式服务框架的显式传参传递到下游节点，实现消息跟踪上下文的跨节点传递。

埋点日志上下文通常需要包含如下内容：

- 1) TraceID、RPCID、调用的开始时间、调用类型、协议类型、调用方 IP 和端口、被调用方 IP 和端口、请求方接口名、被调用方服务名等信息。
- 2) 调用耗时、调用结果、异常信息、处理的消息报文大小等。
- 3) 可扩展字段，通常用于应用扩展埋点上下文信息。

消息跟踪 ID（TraceID）是关联一次完整应用调用的唯一标识，需要在整个集群内唯一，它的取值策略有很多，例如 UUID，UUID（Universally Unique Identifier）即全局唯一标识符，是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的。按照开放软件基金会（OSF）制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片 ID 码和许多可能的数字。由以下几部分组合：当前日期和时间（UUID 的第一部分与时间有关，如果你在生成一个 UUID 之后，过几秒又生成一个 UUID，则第一部分不同，其余相同），时钟序列，全局唯一的 IEEE 机器识别号（如果有网卡，从网卡获得，没有网卡以其他方式获得），UUID 的唯一缺陷在于生成的结果串会比较长。

Java 中获取 UUID 的常用方式如下：

```
UUID uuid = UUID.randomUUID();  
String traceID = uuid.toString();
```

除了 UUID 之外，也可以使用具有业务语义的信息组装成 TraceID，包括但不限于如下字段。

- 1) IP 地址和端口：调用发起方和被调用方 IP 地址、端口号。
- 2) 时间戳：埋点上下文的生成时间。
- 3) 顺序号：标识链路传递序列的 RpcID。

4) 进程号：应用的进程 ID。

5) 随机数：例如可以选择 8 位数的随机数。

RpcID 通常用来标识日志埋点顺序和嵌套关系，它也通过显式传参的方式在各个系统间进行传递，通常使用顺序多级编号的方式展示，如图 18-7 所示。

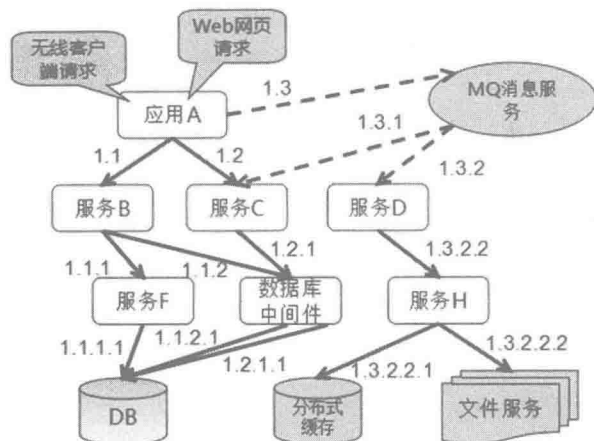


图 18-7 RpcID 生成规则

总结如下：同级调用加 1，例如应用 A 同时调用服务 B 和服务 C，则 RpcID 为 1.1、1.2……跨节点调用加.1，呈现出调用的层级关系，例如服务 B 又调用了服务 F，则服务 B 的 RpcID 为 1.1.1。

从原理上看，埋点日志比较简单，实现起来并不复杂。但是在实际工作中，埋点日志也会面临一些技术挑战，举例如下。

- 1) 异步调用：业务服务中直接调用 MQ 客户端，或者其他中间件的客户端时，可能会发生线程切换，通过线程上下文传递的埋点信息丢失，MQ 客户端会认为自己是首节点，重新生成 TraceID，导致调用链串接不起来。
- 2) 性能影响：由于 Java I/O 操作通常都是同步的，如果磁盘的 WIO 比较高，会导致写埋点日志阻塞应用线程，导致时延增大。频繁地写埋点日志，也会占用大量的 CPU、带宽等系统资源，影响正常业务的运行。

对于线程切换问题，在切换时需要做线程上下文的备份，将埋点上下文复制到切换的线程上下文中，即可解决问题。

频繁写埋点日志影响性能问题，可以通过如下措施改善该问题：

- 1) 支持异步写日志，防止写埋点日志慢阻塞服务线程。具体实现上可以通过采用 log4j 的异步 Appender、独立的日志线程池甚至是 JDK1.7 之后提供的异步文件操作接口。
- 2) 提供可灵活配置的埋点采样率，控制埋点日志量。
- 3) 批量写日志，日志流控机制。

18.2.3 采样率

对于高 QPS 的应用，服务调用埋点本身的性能损耗也不容忽视，为了解决 100%全采样带来的性能损耗，可以通过采样率来实现埋点低损耗的目标。

采样包括静态采样和动态采样两种，静态采样就是系统上线时设置一个采样率，无论负载高低，均按照该采样率执行。动态采样率根据系统的负载可以自动调整，当负载比较低的时候可以实现 100%全采样，在负载非常重时甚至可以降低到 0 采样。

采样率主要包括两个配置参数，分别如下。

- 1) 采样开关：0 表示关闭，不采样；1 表打开，默认为打开。
- 2) 采样周期：采样率使用，表示一个采样周期。
- 3) 采样率：表示在一个采样周期内采样多少次。

是否采样由调用链的首节点进行判断，首节点根据采样率算法，决定某个业务访问是否采样，如果需要采样，则把采样标识、TraceID 等采样上下文发送到下游服务节点，下游服务节点根据采样标识做判断，如果采样则获取调用链上下文并补充完整，反之则不埋点。

18.2.4 采集和存储埋点日志

利用开源的 ELK，可以非常高效地实现分布式日志采集和汇总，它的工作原理图如图 18-8 所示。

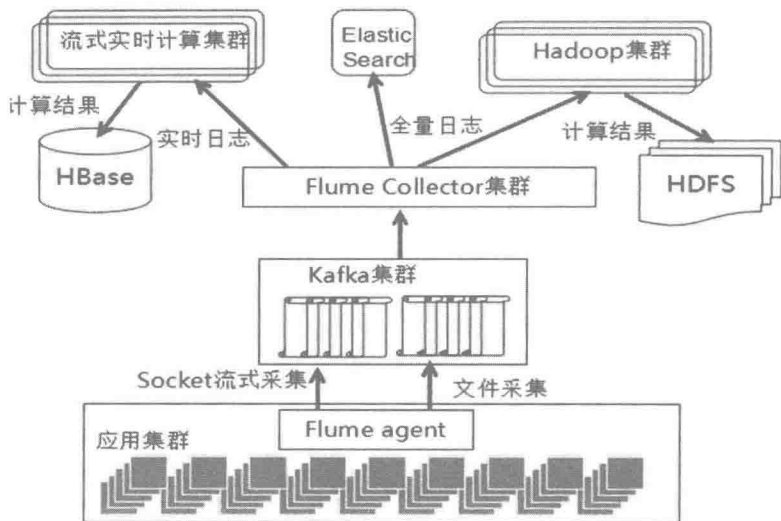


图 18-8 Flume 实现分布式日志采集和汇总

利用 Flume Agent 提供的 Source 插件机制，可以灵活实现埋点日志的采集周期和策略，采集插件实现的时候需要考虑如下几点：

- 1) 采集过程中发生宕机，如何在中断点恢复采集。
- 2) 采集过程中如果埋点日志发生了文件切换（例如到达单个日志文件 100MB 上限之后，自动进行文件切换），如何正确应对。
- 3) 采集 Channel 发生网络故障，导致采集的日志部分发送失败，故障恢复之前，日志如何缓存，故障恢复之后，已采集尚未发送的日志如何发送。
- 4) 考虑到性能，是不是单条采集、批量发送性能更优。

采集节点和汇总节点之间使用 MQ 的原因如下。

- 1) 协调性能处理不均问题：在大规模应用组网条件下，采集节点数远远大于汇总处理节点，采集节点的日志发送速度大于汇总节点的处理能力，导致消息在汇总节点积压，当积压达到一定量时，就会发生 OOM。
- 2) 可靠性：如果通信链路发生故障，或者采集节点宕机，就会反向导致采集节点的消息积压，当采集节点消息积压到一定程度之后，采集节点也会宕机，导致埋点消息丢失。

利用高性能的消息队列 **Kafka**，可以作为采集节点和汇总处理节点的缓冲带，同时利用 **MQ** 集群自身的可靠性和数据持久化机制，保证埋点日志不丢失。

埋点日志的采集周期可以灵活配置，为了做到实时问题定位，通常需要做到秒级采集，例如 5S。全量业务日志和埋点日志的存储策略如下：

- 1) 全量日志存 **HDFS** 和 **Elastic Search**。
- 2) 增量实时日志、计算之后的价值日志存 **HBase**。

18.2.5 计算和展示

汇总得到各应用节点原始的调用链日志之后，可以按照某个服务的消费来源、入口的 URL 做各维度的计算和展示。

例如对同一个入口的调用链做统计分析，需要标准化 URL，针对 URL 中的动态参数和多个域名，可以按照应用、域名归类，使用正则表达式替换动态变化部分。

按照各个维度统计分析之后的价值数据，可以保存在 **HBase** 或者关系型数据库中，方便前端的调用链 **Portal** 报表展示，展示界面示例如图 18-9 所示。

用户可以按照应用名、机器 IP 地址、服务名、协议类型（**HTTP**/服务框架/**MQ**）、调用结果、超时时间等维度进行查询，例如查询应用调用时延超过 500MS 的调用链，查询结果如图 18-10 所示。

调用链查询

应用名称:

机器IP:

服务名称:

调用类型:

调用出错过滤:

调用超时:

ms

时间:

至

搜索

TraceId

开始时间

操作

图 18-9 调用链条件查询

调用链查询

应用名称:

机器IP:

服务名称:

调用类型:

调用出错过滤:

调用超时:

ms

时间:

至

搜索

图 18-10 调用链查询结果

按照条件过滤查询到概要信息之后，可以点击链接查看调用链的详情，详细查看每个服务调用的结果、耗时、消息大小等信息，方便问题定位，如图 18-11 所示。

调用链详情

调用链TraceId:

搜索

应用名	IP	类型	状态	大小	服务/方法	时间轴	耗时
EchoDemo		UNKNOWN	UNKNOWN	0B	from://8e2ae7d4-767a-4e34-b0dd-af3e7d2554d2		0ms
EchoDemo		HSF	OK	0B	org.netty.demo.echo.EchoService@echo~S		1505ms
EchoDemo		HSF	OK	0B	org.netty.demo.cache.CacheService@save~O		0ms
EchoDemo		HSF	OK	0B	org.netty.demo.cache.CacheService@remove~S		702ms
EchoDemo		HSF	OK	0B	org.netty.demo.db.SqlClientService@query~SO		802ms

图 18-11 调用链详细信息

根据调用链的故障定界信息，结合业务的运行日志、接口日志等信息，就可以方便地进行快速问题定位。例如图 18-11，我们可以直接查看对应 IP 上的 `org.netty.demo.echo.EchoService` 接口的 `echo` 方法相关日志。

18.2.6 调用链扩展

除了中间件预埋的字段，应用很多时候需要把应用相关的字段也预埋到埋点上下文中，方便问题定位。

调用链埋点日志需要提供扩展字段和接口，允许业务将应用强相关的信息组装到调用链中，例如：

- 1) 移动终端的型号、IMEI 和 MEID、手机号码、位置信息等。
- 2) 用户访问的浏览器信息（浏览器名称、版本等）、门户信息等。
- 3) 交易 ID、账号信息、订单号码等。

通过业务层的附加信息，可以精确定位到哪个消费者在什么时间调用了哪些服务，调用这些服务的交易相关信息都可以被详细记录下来，当发生用户投诉的时候，利用这些附加信息可以精准地进行问题定位。

当然，在采集用户附加信息时，需要注意个人隐私保护和信息安全，对于用户账号、交易 ID、手机号码等敏感信息，应该加密存储，防止信息泄露。

18.3 总结

利用分布式消息跟踪系统，可以对业务调用的流程进行记录和采集，通过在线和离线的大数据计算，从海量调用日志中抽取对运维和运营有价值的数据，提升业务的运行质量，挖掘新的营销增长点，促进业务由 IT 运营向 DT 智慧运营转型。

对海量调用链日志的采集、存储和计算，实际也给业务提供了日志集中式、后处理能力，由于磁盘存储空间有限，应用的日志往往有个数限制，当超过最大备份数之后老的日志会被绕接掉，如果没有及时采集或者日志产生太快，很容易发生故障日志被覆盖掉，由于大多数问题定位依赖日志，这就给问题定位带来了很大困难。分布式消息跟踪系统可以实时对各类日志进行分布式采集、统一存储管理，为日志的可靠性存储提供了保障。

利用 Elastic Search 等检索服务，为日志指定字段建立索引，可以提供实时的多条件检索功能，方便问题定位。

对分布式消息跟踪的价值总结如图 18-12 所示。

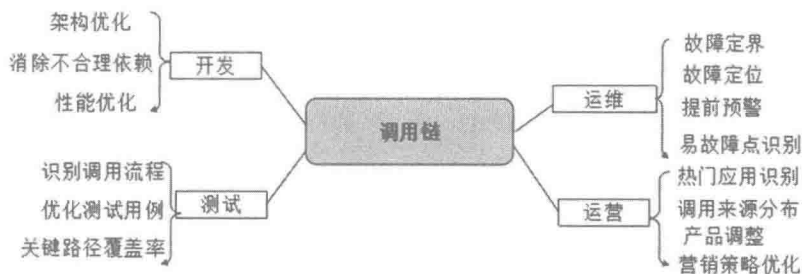


图 18-12 调用链价值汇总

可靠性设计

相对于传统的本地 Java API 调用，跨进程的分布式服务调用面临的故障风险更高。

- 1) 网络类故障：链路闪断、读写超时等。
- 2) 序列化和反序列化失败。
- 3) 畸形码流。
- 4) 服务端流控和拥塞保护导致的服务调用失败。
- 5) 其他异常……

对于应用而言，分布式服务框架需要具备足够的健壮性，在平台底层能够拦截并向上屏蔽故障，业务只需要配置容错策略，即可实现高可靠性。

19.1 服务状态检测

在分布式服务调用时，某个服务提供者可能已经宕机，如果采用随机路由策略，消息会继续发送给已经宕机的服务提供者，导致消息发送失败。为了保证路由的正确性，消费者需要能够实时获取服务提供者的状态，当某个服务提供者不可用时，将它从缓存的路由表中删除掉，不再向其发送消息，直到对方恢复正常。

19.1.1 基于服务注册中心状态检测

在实际项目中，通常会使用服务注册中心对服务提供者进行状态检测，当检测到服务提供者不可用时，会将故障的服务信息广播到集群所有节点，消费者接收到服务故障通知消息之后，根据故障信息中的服务名称、IP 地址等信息，对故障节点进行隔离。

以 ZooKeeper 为例，ZooKeeper 服务端利用与 ZooKeeper 客户端之间的长链接会话做心跳检测，当连续 N 次都没有接收到客户端的心跳应答时，认为对方不可用，将长链接会话删除，同时向其他监听该会话节点的监听者推送超时节点被删除的详细信息，消费者根据超时的节点信息，获取服务名称、版本号、分组信息、IP 地址和端口等，利用这些信息更新服务提供者路由缓存表，将故障服务节点隔离，不再向其发送消息。

基于服务注册中心的状态检测原理图如图 19-1 所示。

服务消费者监听订阅的服务提供者的 Node 列表，当 ZooKeeper 服务端检测到 ZooKeeper 客户端会话超时之后，就会将该会话对应的 Node 删除，并将 Node 删除事件通知到所有监听该 Node 的消费者，消费者根据通知消息中的服务提供者信息更新路由缓存表，不再向故障节点发送消息。

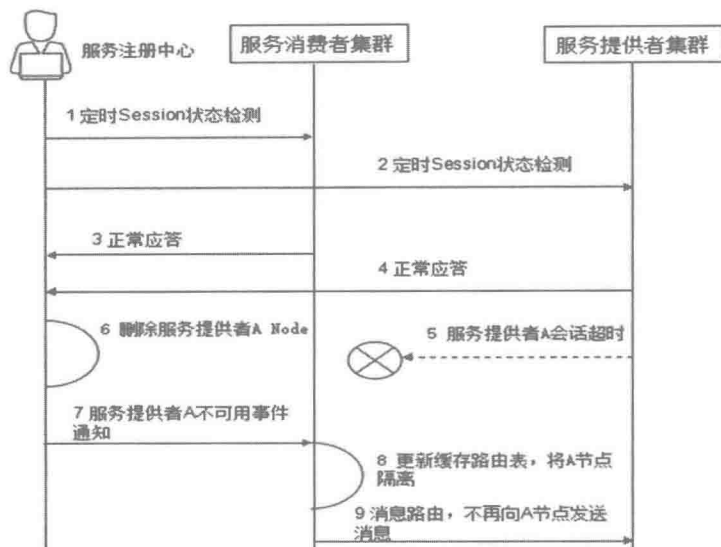


图 19-1 基于服务注册中心的服务状态检测

19.1.2 链路有效性状态检测机制

分布式服务框架的服务消费者和提供者之间默认往往采用长链接，并且通过双向心跳检测保障链路的可靠性。

在一些特殊的场景中，服务提供者和注册中心之间网络可达，服务消费者和注册中心网络也可达，但是服务提供者和消费者之间网络不可达，或者服务提供者和消费者之间链路已经断连。此时，服务注册中心并不能检测到服务提供者异常，但是如果消费者仍旧向链路中断的提供者发送消息，写操作将会失败。

为了解决该问题，通常需要使用服务注册中心检测 + 服务提供者和消费者之间的链路有效性检测双重检测来保障系统的可靠性，它的工作原理如图 19-2 所示。

当消费者通过双向心跳检测发现链路故障之后，会主动释放链接，并将对应的服务提供者从路由缓存表中删除。当链路恢复之后，重新将恢复的故障服务提供者地址信息加入地址缓存表中。

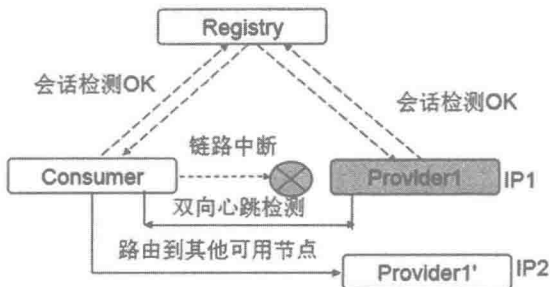


图 19-2 基于链路有效性检测的服务状态检测机制

19.2 服务健康度检测

在集群组网环境下，由于硬件性能差异、各服务提供者的负载不均等原因，如果采用随机路由分发策略，会导致负载较重的服务提供者不堪重负被压垮。

利用服务的健康度检测，可以对集群的所有服务实例进行体检，根据体检结果对健康度做打分，得分较低的亚健康服务节点，路由权重会被自动调低，发送到对应节点的消息相比于其他健康节点会少很多，这样就能够实现“能者多劳、按需分配”，实现更合理的资源分配和路由调度。

服务的健康度检测通常需要采集如下性能 KPI 指标：

- 1) 服务调用时延。
- 2) 服务 QPS。
- 3) 服务调用成功率。
- 4) 基础资源使用情况，例如堆内存、CPU 使用率等。

健康度评分通常由独立的监控节点负责，服务提供者将性能 KPI 数据周期性上报到监控节点，性能汇总和监控节点根据内置的健康度算法对各服务提供者做健康度打分，然后周期性地将服务提供者健康度信息下发给消费者，消费者根据健康度评分动态调整路由权

重系数，实现按需路由-向处理能力强的节点发送更多的消息，保护亚健康服务节点。

服务健康度检测工作原理如图 19-3 所示。

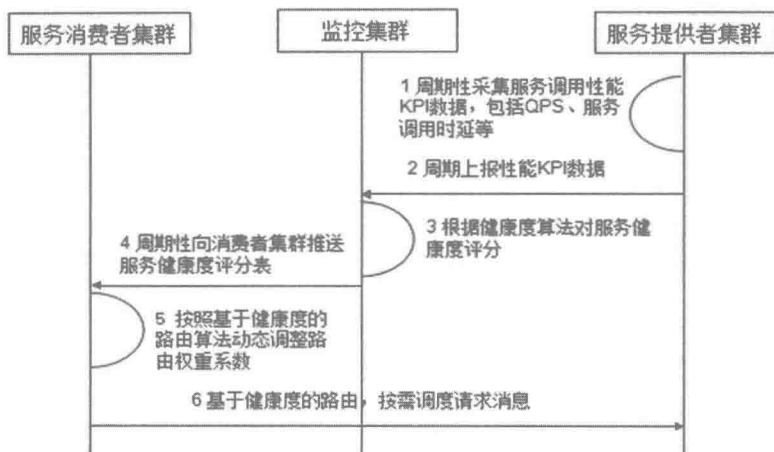


图 19-3 服务健康度检测和路由

19.3 服务故障隔离

服务的故障隔离分为四个层次：

- 1) 进程级故障隔离。
- 2) VM 级故障隔离。
- 3) 物理机故障隔离。
- 4) 机房故障隔离。

19.3.1 进程级故障隔离

进程内部，主要通过将服务部署到不同的线程池实现故障隔离。对于订单、购物车等

核心服务可以独立部署到一个线程池中，与其他服务做线程调度隔离。对于非核心服务，可以合设共享同一个/多个线程池，防止因为服务数过多导致线程数过度膨胀。

它的开发配置示意如下所示：

```
<xxx:service interface="edu.neu.EchoService" ref="echoService" mock =  
"force: execute Bean:  
    echoServiceMock" priority = "HIGH" executor = "Core-ThreadPool-1"/>  
<bean id="Core-ThreadPool-1" class = "xxx.ThreadExecutor">  
    <property name="corePoolSize" value="10" />  
    <property name="maximumPoolSize" value="15" />  
    <property name="keepAliveTime" value="300" />  
    <property name="queueSize" value="8000" />  
</bean>
```

服务发布的时候，可以指定服务发布到哪个线程池中，分布式服务框架拦截 Spring 容器的启动，解析 executor 标签，生成服务和线程池的映射关系，通信框架将解码后的消息投递到后端时，根据服务名选择对应的线程池，将消息投递到映射线程池的消息队列中。

如果服务发布时没有指定线程池，则映射到默认的线程池中。故障隔离的工作原理如图 19-4 所示。

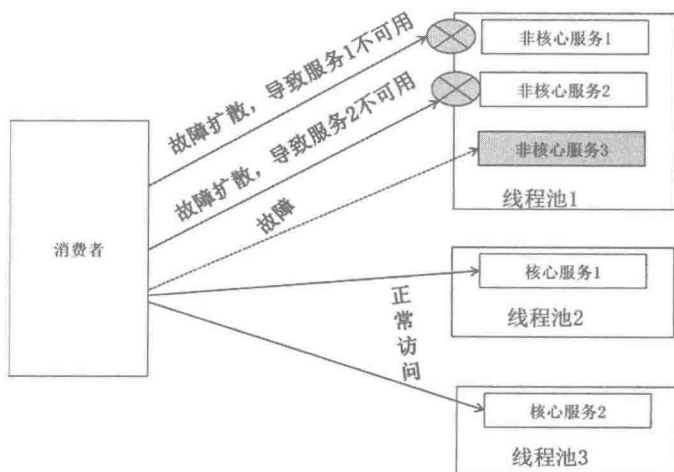


图 19-4 基于线程池的故障隔离

假如非核心服务 3 发生故障，长时间阻塞线程池 1 的工作线程，其他与其共用线程池消息队列的非核心服务 1 和服务 2 只能在队列中排队等待，当服务 3 释放线程之后，排队的服务 1 和服务 2 可能已经超时，只能被丢弃掉，导致处理失败。

采用线程池隔离的核心服务 1 和服务 2，由于各自独占线程池，拥有独立的消息队列，它的执行不受发生故障的非核心服务 1 影响，因此可以继续正常工作。通过独立线程池部署核心服务，可以防止故障扩散，保障核心服务的正常运行。

尽管通过不同的线程池可以实现服务的故障隔离，但是这种隔离并不充分，例如某个故障服务发生了内存泄漏异常，它会导致整个进程不可用，即便实现资源调度层的隔离，仍然无法保证其他服务不受影响。如果要实现物理层面的故障隔离，可以通过 VM 故障隔离来实现。

19.3.2 VM 级故障隔离

通过将基础设施层虚拟化、服务化，将应用部署到不同的 VM 中，利用 VM 对资源层的隔离，可以实现更高层次的服务故障隔离，它的工作原理如图 19-5 所示。

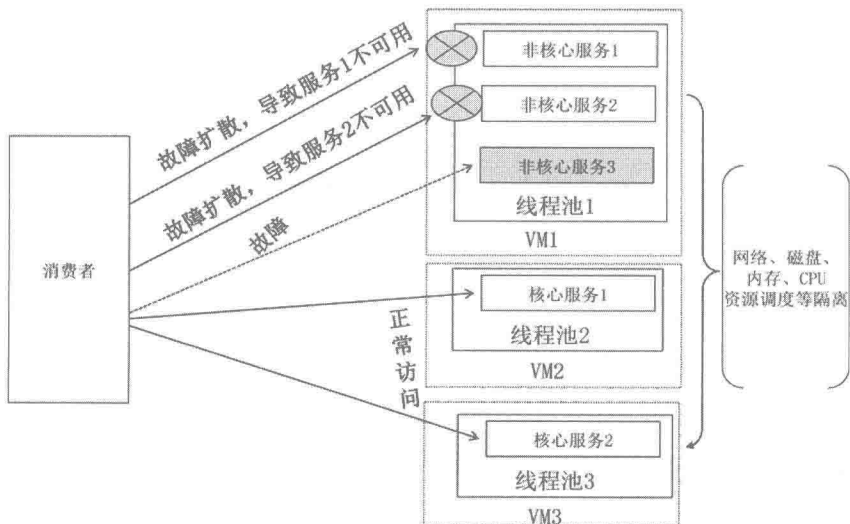


图 19-5 VM 级故障隔离

将核心服务和非核心服务分别部署到不同的 VM 中,利用 VM 的 CPU 调度、网络 I/O、磁盘和内存等资源限制,实现物理资源层的隔离。

当非核心服务 3 发生故障时,无论是线程死循环、内存 OOM,还是句柄耗尽,CPU 资源占用过高,由于 VM 对资源规格的限制,这些故障并不会影响其他 VM 的正常运行,与线程级故障隔离配合使用可以实现逻辑+物理层的故障隔离。

19.3.3 物理机故障隔离

当组网规模足够大、硬件足够多的时候,硬件的故障就由小概率事件转变成为普通事件。如何保证在物理机故障时,应用能够正常工作,是一个不小的挑战。

利用分布式服务框架的集群容错功能,可以实现位置无关的自动容错,它的工作原理如图 19-6 所示。

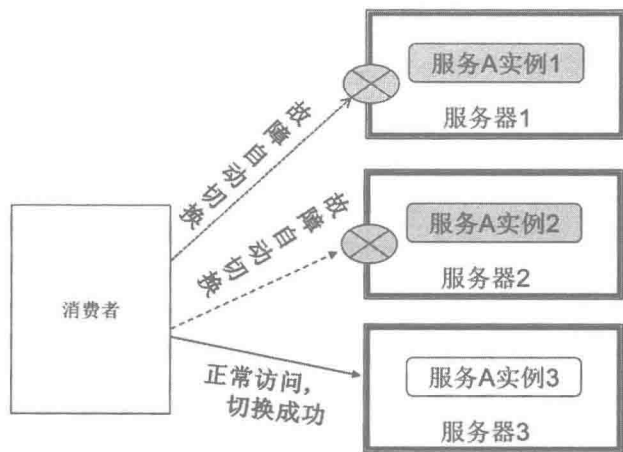


图 19-6 集群容错策略

如果要保证当服务器宕机时不影响部署在上面运行的服务,需要采用分布式集群部署,而且要采用非亲和性安装:即服务实例需要部署到不同的物理机上,通常至少需要 3 台物理机,假如单台物理机的故障发生概率为 0.1%,则 3 台同时发生故障的概率为 0.001%,服务的可靠性将会达到 99.999%,完全可以满足大多数应用场景的可靠性要求。

物理机故障重启之后，通过扩展插件通知 Watch Dog 重新将应用拉起，应用启动时会重新发布服务，服务发布成功之后，故障服务器节点就能重新恢复正常工作。

19.3.4 机房故障隔离

当应用规模非常大时，已经达到单个机房的容量上限，或者做同城容灾时，都需要使用多个机房，下面针对跨机房的容灾和故障隔离方案进行探讨。

以同城双机房为例进行说明，同城机房距离上通常在几十公里，采用光纤专线连接，它的容灾部署策略如图 19-7 所示。

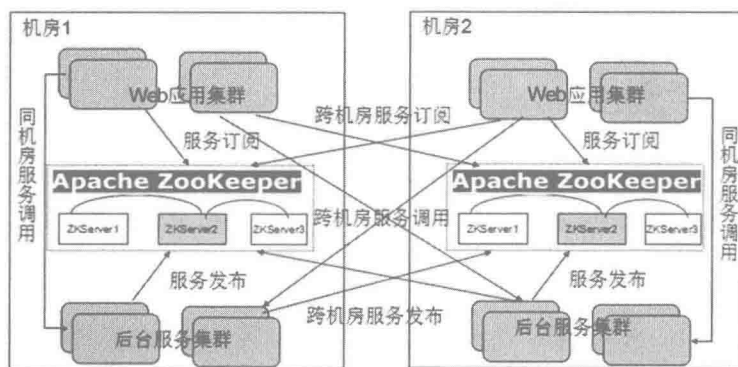


图 19-7 同城机房故障隔离原理图

机房 1 和机房 2 对等部署了 2 套应用集群，每个机房部署一套服务注册中心集群，服务订阅和发布同时针对两个注册中心，对于机房 1 或者机房 2 的 Web 应用，可以同时看到两个机房的服务提供者列表。

路由时，优先访问同一个机房的服务提供者，当本机房的服务提供者大面积不可用或者全部不可用时，根据跨机房路由策略，访问另一个机房的服务提供者，待本机房服务提供者集群恢复到正常状态之后，重新切换到本机房访问模式。

当整个机房宕机之后，由前端的 SLB\F5 负载均衡器自动将流量切换到容灾机房，由于主机房整个瘫掉了，容灾机房的消费者通过服务状态检测将主机房的所有服务提供者从

路由缓存表中删除，服务调用会自动切换到本机房调用模式，实现故障的自动容灾切换。

上面的方案需要分布式服务框架支持多注册中心，同一个服务实例，可以同时注册到多个服务注册中心中，实现跨机房的服务调用。两个机房共用一套服务注册中心也可以，但是如果服务注册中心所在的机房整个宕掉，则分布式服务框架的服务注册中心将不可用。已有的服务调用不受影响，新的依赖服务注册中心的操作将会失败，例如服务治理、运行期参数调整、服务的状态检测等功能将不可用。

19.4 其他可靠性特性

19.4.1 服务注册中心

服务注册中心需要采用对等集群设计，任意一台宕机之后，需要能够自动切换到下一台可用的注册中心。例如 ZooKeeper，如果某个 Leader 节点宕机，通过选举算法会重新选举出一个新的 Leader，只要集群组网实例数不小于 3，整个集群就能够正常工作。

整个服务注册中心集群宕机，消费者仍然能够通过缓存的路由表访问服务提供者，正常业务不受影响。只有依赖 ZooKeeper 的功能暂时不能用，例如服务治理、服务状态检测等。

19.4.2 监控中心

监控中心集群宕机之后，只丢失部分采样数据，依赖性能 KPI 采样数据的服务健康度检测功能不能正常使用，服务提供者和消费者依然能够正常运行，业务不会中断。

19.4.3 服务提供者

某个服务提供者宕机之后，利用分布式服务框架的集群容错策略，会尝试不同的容错

恢复手段，例如使用 FailOver 容错策略，可以自动尝试切换到下一个可用的服务，直到找到可用的服务为止。只要集群组网中还有一个服务提供者可用，应用就不会中断。

如果整个服务提供者集群都宕机，可以利用服务放通、故障引流、容灾切换等手段，快速恢复业务。

19.5 总结

服务化之后，潜在故障点增加了很多，如果能够在分布式服务框架层面解决可靠性问题，上层应用就可以专心进行业务逻辑开发，而不需要到处考虑可靠性和容灾等。高可靠性是分布式架构与生俱来的优势，但是如何落地和实现不仅仅需要理论知识，更多需要从应用实践中获取，理论结合实践，不断提高应用的可靠性。

第 20 章

微服务架构

过去十年，SOA（Service-Oriented Architecture）架构得到了广泛的应用，现在，随着云计算、移动互联网、Docker 容器等技术的快速发展和应用，“微服务”架构（Micro Service Architecture）这一全新的架构风格越来越受到大家的关注，也有越来越多的企业和平台服务提供商在实践中尝试并使用它来解决具体业务问题，微服务架构的流行已经成为未来技术发展趋势之一。

20.1 微服务架构产生的历史背景

微服务架构的产生和流行并不是偶然的，它是多重因素推动下的必然结果。下面我们通过传统 MVC 垂直架构面临的挑战进行剖析，来了解微服务化带来的改变。

20.1.1 研发成本挑战

1. 代码重复率高

通常企业类软件功能复杂、模块比较多，内部会划分成多个研发小组，每个小组负责一部分功能模块的开发、测试和打包，最后由集成小组负责各模块的集成打包。

各小组之间往往会存在交集，例如用户身份证鉴权功能，在手机开户、资费套餐查询等业务中都会被调用，这就造成代码重复率高。一种比较好的做法是将身份证鉴权做成公共类库，供大家调用，此时需要成立一个公共类库小组，负责开发公共功能。但是随着时间的推移，我们会发现公共类库承载的职责越来越多，它需要了解其他模块的业务和功能、甚至依赖其他模块的接口或者类库，这种反向依赖最终会导致公共小组走向两个极端：要么因为职责不清而被解散下放到各个小组、要么臃肿膨胀什么都做，成为整个项目的瓶颈。

在实践中，这种公共小组更多是承载与具体业务模块不强相关的公共类库开发，例如参数合法性校验、数据缓存、SLA 和流控等。

基于 MVC 架构开发的业务特点就是前后台分层，展示和控制层分离，这提升了前后台业务的开发效率。但是代码结构复杂、重复率高一直是个无法很好解决的难题。

在实际项目分工时，开发都是各自负责几个功能，即便开发之间存在功能重叠，往往也会选择自己实现，而不是类库共享，主要原因如下：

- 1) 从技术架构角度看，传统垂直架构的特点是本地 API 接口调用，不存在业务的拆分和互相调用，使用到什么功能就本地开发，非常方便，不需要过度依赖于其他功能模块。

- 2) 从考核角度来看, 共享很难推行。开发只需要对自己开发的模块交付质量负责, 没有义务为他人提供并维护公共类库, 这个非常耗费成本。
- 3) 时间依赖很难把控。对于公共类库的使用者而言, 依赖别人提供此功能, 但是功能提供者可能有更重要的事情在做, 交付时间点无法满足使用者。与其坐等别人提供, 还不如自己开发效率高。
- 4) 跨地域、跨开发小组协调很困难, 业务团队可能跨地域研发, 内部通常也会分成多个开发小组, 各开发小组之间的协调和沟通成本非常高。

功能的重复开发会导致研发成本上升, 代码质量下降, 架构腐化, 为后续系统的运维和新功能的开发带来巨大的挑战。

2. 需求变更困难

代码重复率变高之后, 已有功能变更或者新需求加入都会非常困难, 以充值缴费功能为例, 不同的充值渠道开发了相同的限额保护功能, 当限额保护功能发生变更之后, 所有重复开发的限额保护功能都需要重新修改和测试, 很容易出现修改不一致或者被遗漏, 导致部分渠道充值功能正常, 部分存在 Bug 的问题, 示例如图 20-1 所示。

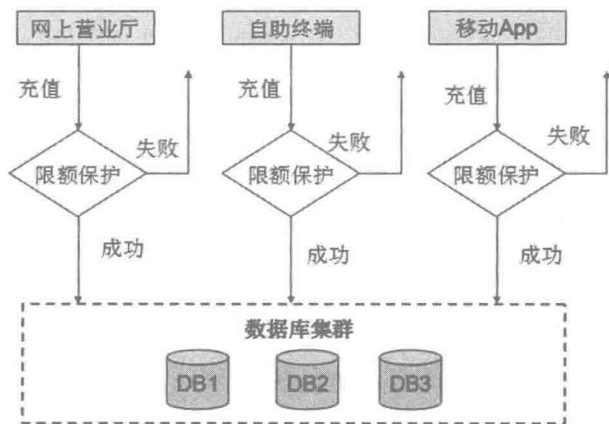


图 20-1 充值限额保护功能需求变更前

修改不一致导致的移动 App 充值失败示例, 如图 20-2 所示。

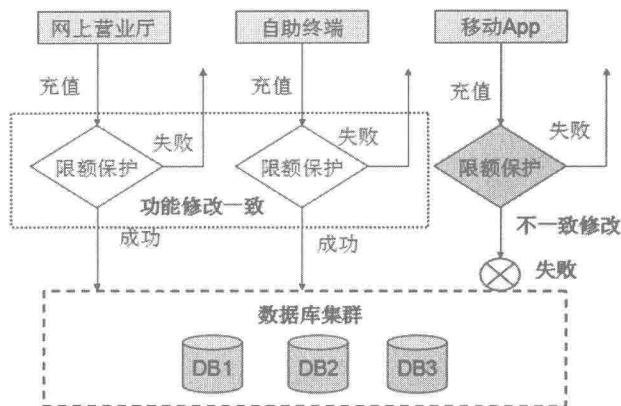


图 20-2 充值限额保护功能需求变更后

20.1.2 运维成本高

1. 代码维护困难

在传统的 MVC 架构中，业务流程是由一长串本地接口或者方法调用串联起来的，臃肿而冗长，而且往往由一个人负责开发和维护。随着业务的发展和需求变化，本地代码在不断地迭代和变更，最后形成了一个垂直的功能孤岛，只有原来的开发者才理解接口调用关系和功能需求，一旦原有的开发者离职或者调到其他项目组，这些功能模块的运维就会变得非常困难，如图 20-3 所示。

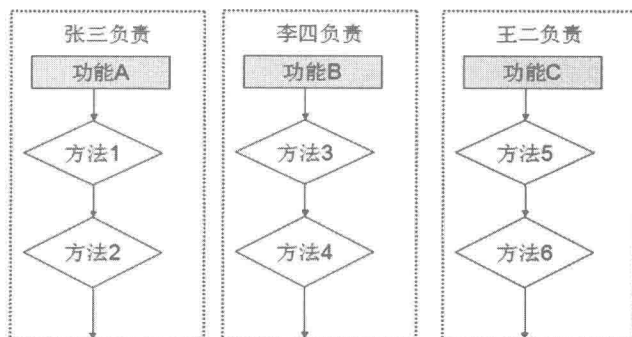


图 20-3 垂直架构导致的功能孤岛

当垂直应用越来越多时，连架构师都无法描述应用间的架构关系，随着业务的发展和功能膨胀，这种架构很容易发生腐化。

2. 部署效率低

部署效率低主要体现在以下三个方面。

- 1) 业务没有拆分，很多功能模块都打到同一个 war 包中，一旦有一个功能发生变更，就需要重新打包和部署。
- 2) 巨无霸应用由于包含功能模块过多，编译、打包时间比较长，一旦编译过程出错，需要根据错误重新修改代码再编译，耗时较长。
- 3) 测试工作量较大，因为存在大量重复的功能类库，需要针对所有调用方进行测试，测试工作量大；另外，由于业务混在一起打包，需要针对集成打包进行专项测试，客观上也增加了测试工作量。

20.1.3 新需求上线周期长

传统垂直架构新需求上线周期较长的原因如下：

- 1) 新功能通常无法独立编译、打包、部署和上线，它可能混杂在老的系统中开发，很难剥离出来，这就无法通过服务灰度发布的形式快速上线。
- 2) 由于业务没有进行水平和垂直拆分，导致代码重复率高，新需求的开发、测试、打包和部署成本都比较高，这制约了新需求的上线速度。

20.2 微服务架构带来的改变

20.2.1 应用解耦

微服务化之前，一个大型的应用系统通常会包含多个子应用，不同应用之间存在很多

重复的公共代码，所有应用共用一套数据库，它的架构如图 20-4 所示。

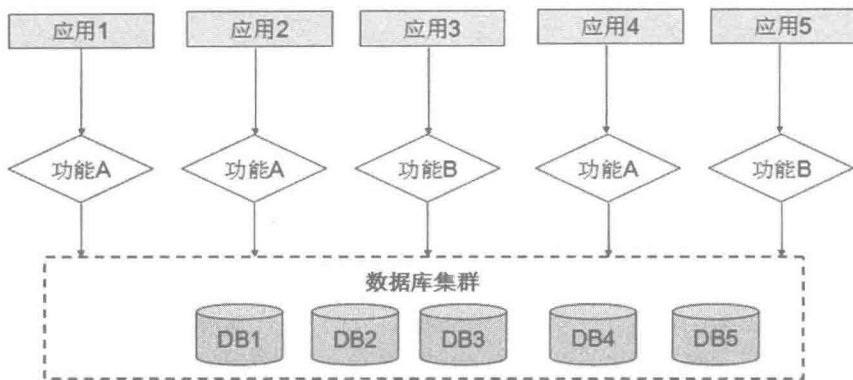


图 20-4 传统应用架构

将功能 A 和 B 服务化之后，应用作为消费者直接调用服务 A 和服务 B，这样就实现了对原有重复代码的收编，同时系统之间的调用关系也更加清晰，示例如图 20-5 所示。

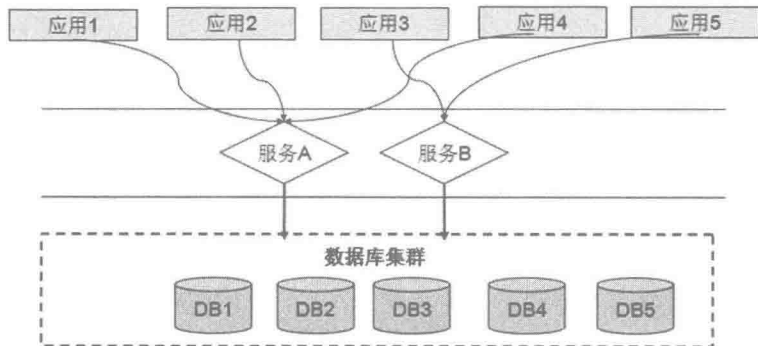


图 20-5 微服务化之后的调用关系

基于服务注册中心的订阅发布机制，可以实现服务消费者和提供者之间的解耦，消费者不需要配置服务提供者的地址信息，即可以实现位置无关的透明化路由，它的开发体验与本地 API 接口调用相似，但是却实现了远程服务调用。通过服务注册中心，可以管理服务的订阅和发布关系，查看服务提供者和服务消费者的详细信息。有了服务订阅关系，业务和服务之间的调用关系变得透明化，不合理的接口依赖、调用关系一目了然。

20.2.2 分而治之

当垂直应用越来越多时，应用之间的交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的底层微服务，使前端应用能更快速地响应多变的市场需求。

应用的拆分分为水平拆分和垂直拆分两种，水平拆分以业务领域为维度，抽象出几个不同的业务域，每个业务域作为一个独立的服务中心对外提供服务。领域服务可以独立地伸缩和升级，快速地响应需求变化，同时与其他业务领域解耦，它的原理如图 20-6 所示。



图 20-6 水平拆分

应用的垂直拆分主要包括前后台逻辑拆分、业务逻辑和数据访问层拆分，拆分之后的效果图如图 20-7 所示。



图 20-7 垂直拆分

需要指出的是，业务垂直拆分之后，Web 前台和后台采用分布式组网，通过分布式服务框架实现前后台业务调用。数据访问层服务可以选择和其他服务合设，也可以独立集群组网，这个需要根据具体的业务场景进行选择。

经过水平和垂直拆分之后，可以将复杂的巨无霸应用拆分成多个独立的微服务，系统由大到小，分而治之。

20.2.3 敏捷交付

软件解决方案的敏捷性，指的是它能够快速进行变更的能力。敏捷性是微服务架构特性中最显著的一点：敏捷性的产生，是将运行中的系统解耦为一系列功能单一服务的结果。微服务架构能够对系统中其他部分的依赖加以限制，这种特性能够让基于微服务架构的应用在应对 Bug 或是对新特性需求时，能够快速地进行变更。这一点与传统的垂直架构恰恰相反，在传统架构中经常发生的一种情况是：“要对应用程序中某个小部分进行变更，就必须对整体架构进行重新编译和构建，并且重新进行全量部署。”

20.3 微服务架构解析

微服务架构（MSA）是一种架构风格，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。

传统架构和微服务架构对比如图 20-8 所示。

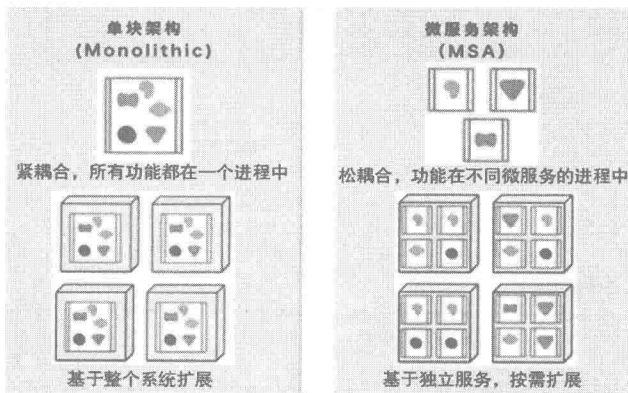


图 20-8 传统架构和微服务架构对比图

20.3.1 微服务划分原则

实际上微服务本身并没有严格的定义，划分原则也有不同的实践，但是比较通用的划分原则是：微服务通常是简单、原子的微型服务，它的功能单一，只负责处理一件事，与代码行数并没有直接关系，与需要处理的业务复杂度有关。有些复杂的功能，尽管功能单一，但是代码量可能成百上千行，因此不能以代码量作为划分微服务的维度。

微服务的“微”并不是一个真正可衡量、看得见、摸得着的“微”。这个“微”所表达的是一种设计思想和指导方针，是需要团队或者组织共同努力找到的一个平衡点，在实践中团队成员可以接受。

微服务总结起来就是小，且专注于做一件事情。

20.3.2 开发微服务

随着团队对业务的理解加深和对微服务实践的尝试，按照团队职责划分，基于微服务架构的开发环境被搭建起来。问题随之也出现了：对于不同的微服务，虽然实现逻辑不同，但是开发方式、持续集成环境、测试策略和部署机制以及后续的上线运维都是类似的，为了满足 DRY 原则并消除浪费，需要搭建统一的开发打包和持续集成环境。

微服务开发套件工具集如图 20-9 所示。

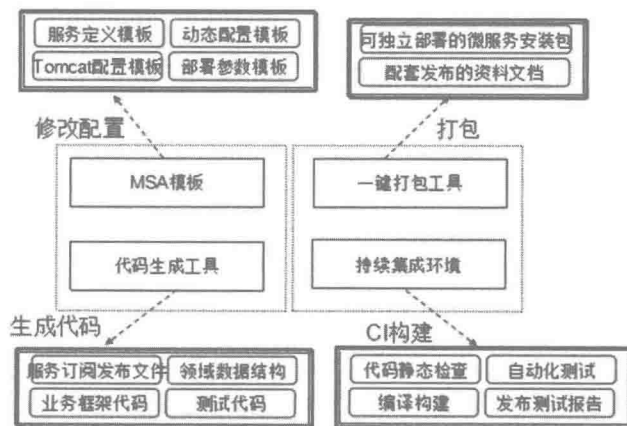


图 20-9 微服务开发套件工具集

MSA 模板是一个独立的代码工程库，主要包括：微服务描述模板文件，以及一系列配置文件模板。

微服务描述模板文件类似于 Thrift 的 IDL 文件，用于描述服务的接口定义、数据结构以及操作集合和参数。微服务可以使用 Tomcat Web 容器部署，将常用的 Tomcat 参数配置提取到模板中，在开发态的时候可以基于界面模板进行修改，打包部署的时候可以替换 Tomcat 的相关配置，省去手工到 Tomcat 目录中修改配置的步骤。动态配置模板将运行时可能修改的参数提取到模板中，在开发态的时候可以设置默认值，例如线程池配置、JVM GC 参数配置、任务队列配置等。部署参数配置将 Web 容器、微服务部署时需要设置的参数提取到模板中，在微服务安装部署的时候运维人员可以根据线上实际环境进行配置修改。

根据 MSA 模板文件，利用微服务 IDE 工具生成微服务代码，主要包括如下几部分：

- 1) 微服务服务提供者发布的服务配置文件、服务消费者订阅的服务配置文件。
- 2) 微服务的接口定义以及默认实现。
- 3) 微服务使用到的数据结构，通常包括请求参数和应答参数数据结构定义。
- 4) 微服务的测试代码，例如远程服务提供者接口的本地 Mock 实现类。

微服务开发完成之后，需要本地打包和单元测试，利用微服务开发 IDE 工具提供的一键打包工具，可以生成完整的微服务部署包，包括：

- 1) 可以独立部署、升级和运维的微服务安装部署包。
- 2) 配套发布的资料文档，包括微服务的 API 接口文档、调用示例、安装指南、帮助指导等，供消费者开发使用，同时也方便测试人员了解微服务的功能特性。

本地测试完成之后，开发人员将代码提交到配置库，由 CI 构建工具定时或者代码提交变更触发自动构建，自动构建流程主要包括：

- 1) 代码静态检查，包括是否符合项目的编码规范、圈复杂度检查、静态编译检查等，检查完成之后，生成代码静态检查报告，并发送给相关责任人，如果指标不达标，则进行整改。

- 2) 代码编译构建, 如果编译或者构建失败, 则生成 CI 构建失败报告, 连同告警邮件一起发送给相关责任人。责任人根据 CI 构建错误原因进行问题排查和修复, 手动或者自动触发 CI 构建流程, 持续构建。
- 3) 构建成功之后, 自动调用自动化测试框架, 进行集成测试, 并生成集成测试报告, 包括微服务个数、测试用例个数、覆盖率、通过率、测试打分等。
- 4) 发布集成测试报告, CI 构建完成之后, 微服务是否能够正式发布上线, 取决于集成测试结果, 根据测试结果, 由测试和开发综合评估, 决定哪些微服务可以发布上线, 哪些需要做 BUG 修复或者改进。

通过使用微服务框架开发套件, 大大缩短了团队开发微服务的周期。同时, 基于套件开发工具和模板, 我们定义了一套团队内部的开发、协作流程, 帮助团队的每一位成员理解并快速构建微服务。

20.3.3 基于 Docker 容器部署微服务

微服务鼓励软件开发者将整个软件解耦为功能单一的服务, 并且这些服务能够独立部署、升级和扩容。如果微服务抽象得足够好, 那么微服务的这一优点将能够提升应用的敏捷性和自治能力。

虽然微服务架构已经逐渐成为一种流行趋势, 但 2013 年 Docker Inc. 公司所发布的 Docker 则给微服务的蓬勃发展注入了更强的活力。Docker 是一套开源工具, 它能够以某种方式对现有的基于容器的虚拟化技术进行封装, 使得它能够在更广阔的工程社区中得到应用。

在 Docker 之前, 容器技术已经出现了十几年, 例如基于轻量级进程的 LXC 容器, 但是一直以来, 容器技术只在部分公司和组织中得到应用, 例如 Google 所有的应用和服务都部署、运行在容器中。Docker 出现之后, 人们对容器的关注和应用产生了爆炸式的增长。

Docker 的吸引力主要来自两个方面: 快速与可移植性。

1. 快速

普通的虚拟机在每次开机时都需要启动一个完整的新操作系统实例，而 Docker 容器能够通过内核共享的方式，共享一套托管操作系统。这意味着，Docker 容器的启动和停止不需要几分钟，只要几百毫秒就足够了。

更快的速度就意味着使用 Docker 容器创建的软件系统比起使用基于虚拟机的解决方案能够实现更高级别的敏捷性，即使将那些基于虚拟机的解决方案通过基于微服务的架构重构之后也一样。此外，容器化的应用比起虚拟机的性能更好，在 2014 年 IBM 发布的一份研究报告中表明：“在几乎所有情况下，容器都能表现出与物理机相等或者是更好的性能。”

Docker 和 VM 的性能对比数据如图 20-10 所示。

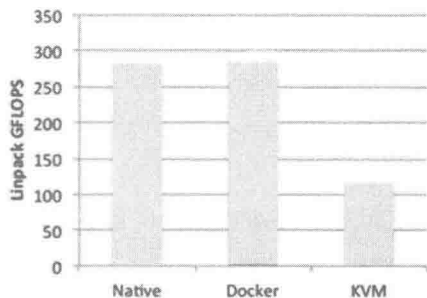


图 20-10 计算效率对比

图 20-10 中从左往右分别是物理机、Docker 和虚拟机的计算能力数据。可见 Docker 相对于物理机其计算能力几乎没有损耗，而虚拟机对比物理机则有着非常明显的损耗。虚拟机的计算能力损耗在 50% 左右。为什么会有这么大的性能损耗呢？一方面是因为虚拟机增加了一层虚拟硬件层，运行在虚拟机上的应用程序在进行数值计算时是运行在 Hypervisor 虚拟的 CPU 上的。另外一方面是由于应用程序本身的特性导致的差异，虚拟机虚拟的 CPU 架构不同于实际 CPU 架构，应用程序一般针对特定的 CPU 架构有一定的优化措施，虚拟化使这些措施作废，甚至起到相反的效果。

2. 可移植性

在基于虚拟机的解决方案中，应用的可移植性通常来说会受到云提供商所提供的虚拟

机格式限制。如果应用程序需要部署到不同类型的虚拟机中，需要针对特定的虚拟机格式做镜像文件，新增很多额外的开发和测试工作量。Docker 容器的设计理念是“一次编写，到处运行”，这可以使开发者避免上面这种限制。工程团队与运维团队可以将他们的基础设施扩展到多个云提供商的服务中，只要 Docker 容器还在运行，就能够保证应用程序的正常运行。这种将应用从云提供商中进行解耦的方式能够给予 IT 团队更大的自由度，也可以在与各个提供商之间的对接中，提高软件解决方案的适应性。

基于 Docker 容器部署微服务至少会带来 4 个方面的优点。

- 1) 线上线下环境等同性：调查报告表明，工程师修复在生产环境上发现的 Bug，比起在代码审查或单元测试阶段修复它，所消耗的精力会多出 40 倍，比起在系统测试阶段修复 Bug 的精力要多出 5 倍。如果代码所测试的环境与实际运行的环境是不同的，那么测试的结果就有可能是无效的，团队或许将为些付出更大的代价，因为他们不得不在生产环境上修复 Bug，因此开发和生产环境的等同性非常重要。与虚拟机相比，运行容器所需的内存要小得多，因此开发者能够在开发机器上运行多个容器。想象一下，如果在生产环境中运行了 10 台虚拟机，那么无论它们的内存占用有多小，想要在一台普通的开发机器上运行 10 个虚拟机也是基本不可能的。开发者为了能够在本地进行测试，不得不采用一些捷径，例如在本地开发环境中使用仿真工具进行测试，这种捷径本质上无法完全模拟生产环境，可能会导致漏测。在开发机器中运行 10 个 Docker 容器却是小菜一碟，这样一来，开发者就能够在开发环境中模拟生产环境，从而增加开发/生产环境的等同性。
- 2) 与特定的云提供商解耦：很多应用在向云端迁移的时候，需要将不同的应用部署到不同的云上，同时也不希望与特定的云提供厂商耦合，过度依赖某个供应商。在容器出现以前，要跨多个云端运行产品是一件相当复杂的任务。各大处于竞争中的云提供商的虚拟机格式都有所不同，从配置管理所需的精力来看，要在多个云端运行产品的代价几乎是不可接受的。由于 Docker 容器包含了运行应用程序所需的运行时环境，以及刚好足够的操作系统资源，因此对于基础设施唯一的要求就是安装 Docker 镜像。这样一来，软件公司只需要在每个云提供商的环境中维护一台能够运行 Docker 的虚拟机就可以了。

- 3) 提升运维效率: Docker 对可移植的容器部署进行标准化, 能够节省时间与精力。如果你正在构建某个应用程序, 你的选择包括物理机、虚拟化的本地基础设施、公有云和私有云, 以及各种可用的 PaaS 选项。那么你在打包与部署软件上最多可以有 6 种不同的方式。而通过 Docker 标准化的容器格式, 这些部署模式中的任意一种的提供商都可以实现一种统一的部署体验。
- 4) 敏捷性: Docker 容器相比于传统的虚拟机, 性能损耗更少, 启动速度也更快, 可以实现应用和服务的秒级部署和启动。在秒杀等业务场景下, 需要能够在几秒的时间内扩容出多个服务实例, 以抵挡流量高峰, 使用 Docker 容器可以实现秒级扩容, 更敏捷、高效地支撑互联网营销业务。

20.3.4 治理和运维微服务

微服务架构对运维和部署流水线要求非常高, 服务拆分的粒度越细, 运维和治理成本就越高, 挑战总结如下。

- 1) 监控度量问题: 海量微服务的各种维度性能 KPI 采集、汇总和分析, 实时和历史数据同比和环比等, 对采集模块的实时性、汇总模块的计算能力、前端运维 Portal 多维度展示能力要求非常高。
- 2) 分布式运维: 服务拆分得越细, 一个完整业务流程的调用链就越长, 需要采集、汇总和计算的数据量就越大, 分布式消息跟踪系统需要能够支撑大规模微服务化后带来的性能挑战。
- 3) 海量微服务对服务注册中心的处理能力、通知的实时性也带来了巨大挑战。
- 4) 微服务治理: 微服务化之后, 服务数相比于传统的 SOA 服务有了指数级增长, 服务治理的展示界面、检索速度等需要能够支撑这种变化。
- 5) 量变引起质变: 当需要运维的服务规模达到一定上限后, 就由量变引起质变, 传统的运维框架架构可能无法支撑, 需要重构。

解决微服务运维的主要措施就是：分布式和自动化。利用分布式系统的性能线性增长和弹性扩容能力，支撑大规模微服务对运维系统带来的性能冲击，涉及的运维功能包括：

- 1) 分布式性能数据采集、日志采集 Agent。
- 2) 分布式汇总和计算框架。
- 3) 分布式文件存储服务。
- 4) 分布式日志检索服务。
- 5) 分布式报表展示框架。

上述几个分布式组件/服务可以根据业务的实际情况选择自研或者使用开源，例如分布式日志采集可以使用 Flume，分布式文件存储可以使用 HDFS，分布式日志检索可以使用 Elastic Search 等。

自动化就是将以以前需要人工操作的运维工作逐步标准化、流程化和规范化，由自动化运维工具实现自动化运维。例如基于容量管理和预测的自动扩容服务，容量管理根据采集的应用性能 KPI 数据和扩容规则，计算出需要扩容的应用实例数，调用 PaaS 平台的弹性伸缩服务，自动创建或者从资源池中获取虚拟机，从软件仓库下载应用镜像并安装，部署并启动应用，实现自动扩容。

利用自动扩容服务，可以代替以前人工扩容，防止误判和误操作，扩容的效率也更高，成本更低。除了自动扩容，包括分布式服务框架的集群容错、分布式消息跟踪系统的故障快速定界和定位等功能，通过运维工具自动化，来应对微服务架构实施之后对运维体系带来的冲击。

20.3.5 特点总结

随着业务需求的快速发展变化，敏捷性、灵活性和可扩展性需求不断增长，迫切需要一种更加快速高效的软件交付方式。微服务就是一种可以满足这种需求的软件架构风格。复杂应用被分解成多个更小的服务，每个服务有自己的归档文件、单独部署，然后共同组成一个复杂应用。总结起来，微服务有如下特点。

- 1) 单一职责原则：每个服务应该负责单独的功能，这是 SOLID 原则之一。
- 2) 独立部署、升级、扩展和替换：每个服务都可以单独部署及重新部署而不影响整个系统。这使得服务很容易升级，每个服务都可以沿着“Art of Scalability”一书定义的 X 轴和 Z 轴进行扩展。
- 3) 支持异构/多种语言：每个服务的实现细节都与其他服务无关，这使得服务之间能够解耦，团队可以针对每个服务选择最合适的开发语言、工具和方法。
- 4) 轻量级：微服务通常由轻量级的分布式服务框架承载，采用 P2P 通信，无中心节点，性能可以线性增长；第三方软件依赖少，减少了类冲突和冗余依赖，集成和升级更方便。

基于上述特点，微服务架构的优点总结如下：

- 1) 开发、测试和运维更加简单。
- 2) 局部修改很容易部署，有利于持续集成和持续交付。
- 3) 技术选择更灵活，不与特定语言和工具绑定。
- 4) 有利于小团队作战，敏捷交付。

20.4 总结

微服务已经成为当下最热门的话题之一。它是一种新的架构风格，涉及组织架构、设计、交付、运维等方面的变革，核心目标是为了解决系统的交付周期，并降低维护成本和研发成本。相比传统的 SOA 架构或者垂直架构，微服务有很多的优势，比如技术的多样性、模块化、独立部署等，但也带来了相应的成本，比如运维成本、服务管理成本等。如何平衡好“微”带来的效益和成本，需要根据公司、团队和应用发展的实际情况，量力而行。

当流水化工厂、持续集成环境（CI）以及服务治理、运维能力跟不上时，强制推行微服务架构，未必是明智之举。技术是为商业服务的，切不可脱离业务实际而使用微服务。

第 21 章

服务化最佳实践

在服务化之前，业务通常都是本地 API 调用，本地方法调用性能损耗较小。服务化之后，服务提供者和消费者之间采用远程网络通信，增加了额外的性能损耗，业务调用的时延将增大，同时由于网络闪断等原因，分布式调用失败的风险也增大。如果服务框架没有足够的容错能力，业务失败率将会大幅提升。

除了性能、可靠性等问题，跨节点的事务一致性问题、分布式调用带来的故障定界困难、海量微服务运维成本增加等也是分布式服务框架必须要解决的难题。本章节我们将对服务化之后面临的挑战进行分析，并给出解决方案和业务最佳实践。

21.1 性能和时延问题

在服务化之前，业务通常都是本地 API 调用，本地方法调用性能损耗较小。服务化之后，服务提供者和消费者之间采用远程网络通信，增加了额外的性能损耗：

- 1) 客户端需要对消息进行序列化，主要占用 CPU 计算资源。
- 2) 序列化时需要创建二进制数组，耗费 JVM 堆内存或者堆外内存。
- 3) 客户端需要将序列化之后的二进制数组发送给服务端，占用网络带宽资源。
- 4) 服务端读取到码流之后，需要将请求数据反序列化成请求对象，占用 CPU 计算资源。
- 5) 服务端通过反射的方式调用服务提供者实现类，反射本身对性能影响就比较大。
- 6) 服务端将响应结果序列化，占用 CPU 计算资源。
- 7) 服务端将应答码流发送给客户端，占用网络带宽资源。
- 8) 客户端读取应答码流，反序列化成响应消息，占用 CPU 资源。

通过分析我们发现，一个简单的本地方法调用，切换到远程服务调用之后，额外增加了很多处理流程，不仅占用大量的系统资源，同时增加了时延。一些复杂的应用会拆分成多个服务，形成服务调用链，如果服务化框架的性能比较差、服务调用时延也比较大，业务服务化之后的性能和时延将无法满足不同业务的性能需求。

21.1.1 RPC 框架高性能设计

影响 RPC 框架性能的主要因素有三个。

- 1) I/O 调度模型：同步阻塞 I/O (BIO) 还是非阻塞 I/O (NIO)。
- 2) 序列化框架的选择：文本协议、二进制协议或压缩二进制协议。

3) 线程调度模型：串行调度还是并行调度，锁竞争还是无锁化算法。

1. I/O 调度模型

在 I/O 编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者 I/O 多路复用技术进行处理。I/O 多路复用技术通过把多个 I/O 的阻塞复用到同一个 `select` 的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O 多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降低了系统的维护工作量，节省了系统资源。

JDK1.5_update10 版本使用 `epoll` 替代了传统的 `select/poll`，极大地提升了 NIO 通信的性能，它的工作原理如图 12-1 所示。

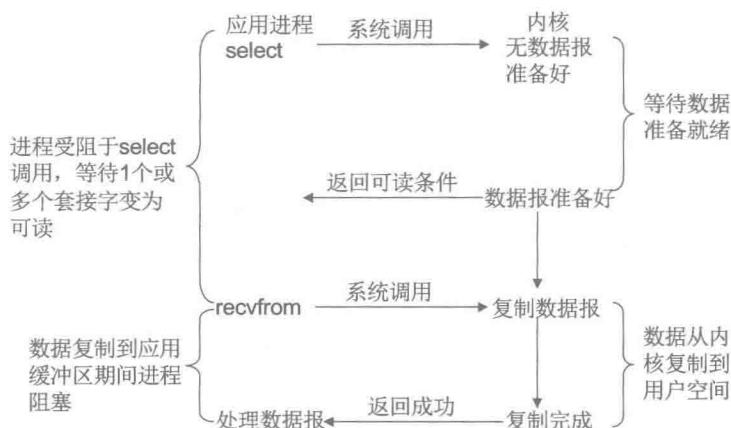


图 21-1 非阻塞 I/O 工作原理

Netty 是一个开源的高性能 NIO 通信框架：它的 I/O 线程 `NioEventLoop` 由于聚合了多路复用器 `Selector`，可以同时并发处理成百上千个客户端 `Channel`。由于读写操作都是非阻塞的，这就可以充分提升 I/O 线程的运行效率，避免由于频繁 I/O 阻塞导致的线程挂起。另外，由于 Netty 采用了异步通信模式，一个 I/O 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 I/O 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

Netty 被精心设计，提供了很多独特的性能提升特性，使它做到了在各种 NIO 框架中性能排名第一，它的性能优化措施总结如下。

- 1) 零拷贝：(1) Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存 (HEAP BUFFERS) 进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。(2) Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便地对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。(3) Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。
- 2) 内存池：随着 JVM 虚拟机和 JIT 即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区 Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽量重用缓冲区，Netty 提供了基于内存池的缓冲区重用机制。性能测试表明，采用内存池的 ByteBuf 相比于朝生夕灭的 ByteBuf，性能高 23 倍左右（性能数据与使用场景强相关）。
- 3) 无锁化的串行设计：在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能地避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。为了尽可能提升性能，Netty 采用了串行无锁化设计，在 I/O 线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。
- 4) 高效的并发编程：volatile 的大量、正确使用；CAS 和原子类的广泛使用；线程安全容器的使用；通过读写锁提升并发性能。

2. 高性能序列化框架

影响序列化性能的关键因素总结如下。

- 1) 序列化后的码流大小（网络带宽的占用）。
- 2) 序列化&反序列化的性能（CPU 资源占用）。
- 3) 是否支持跨语言（异构系统的对接和开发语言切换）。
- 4) 并发调用的性能表现：稳定性、线性增长、偶现的时延毛刺等。

相比于 JSON 等文本协议，二进制序列化框架性能更优异，以 Java 原生序列化和 Protobuf 二进制序列化为例进行性能测试对比，结果如图 21-2 所示。

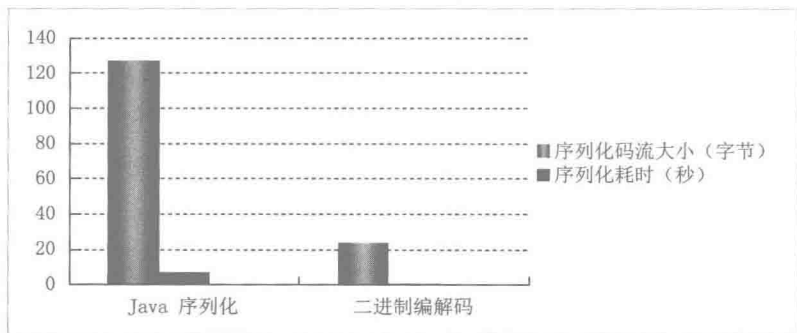


图 21-2 序列化性能测试对比数据

在序列化框架的技术选型中，如无特殊要求，尽量选择性能更优的二进制序列化框架，码流是否压缩，则需要根据通信内容做灵活选择，对于图片、音频、有大量重复内容的文本文件（例如小说）可以采用码流压缩，常用的压缩算法包括 GZip、Zig-Zag 等。

3. 高性能的 Reactor 线程模型

该模型的特点总结如下。

- 1) 有专门一个 NIO 线程：Acceptor 线程用于监听服务端，接收客户端的 TCP 连接请求。
- 2) 网络 I/O 操作：读、写等由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线

程池实现，它包含一个任务队列和 N 个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送。

- 3) 1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，防止产生并发操作。

由于 Reactor 模式使用的是异步非阻塞 I/O，所有的 I/O 操作都不会导致阻塞，理论上一个线程可以独立处理所有 I/O 相关的操作，因此在绝大多数场景下，Reactor 多线程模型都可以完全满足业务性能需求。

Reactor 线程调度模型的工作原理示意如图 21-3 所示。

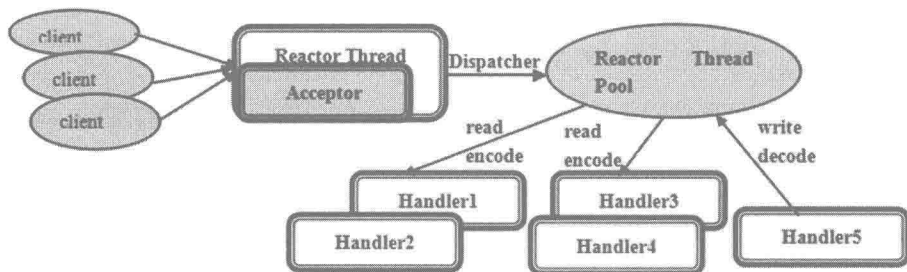


图 21-3 高性能的 Reactor 线程调度模型

21.1.2 业务最佳实践

要保证高性能，单依靠分布式服务框架是不够的，还需要应用的配合，应用服务化高性能实践总结如下：

- 1) 能异步的尽可能使用异步或者并行服务调用，提升服务的吞吐量，有效降低服务调用时延。
- 2) 无论是 NIO 通信框架的线程池还是后端业务线程池，线程参数的配置必须合理。如果采用 JDK 默认的线程池，最大线程数建议不超过 20 个。因为 JDK 的线程池默认采用 N 个线程争用 1 个同步阻塞队列方式，当线程数过大时，会导致激烈的锁竞争，此时性能不仅不会提升，反而会下降。

- 3) 尽量减小要传输的码流大小, 提升性能。本地调用时, 由于在同一块堆内存中访问, 参数大小对性能没有任何影响。跨进程通信时, 往往传递的是个复杂对象, 如果明确对方只使用其中的某几个字段或者某个对象引用, 则不要把整个复杂对象都传递过去。举例, 对象 A 持有 8 个基本类型的字段, 2 个复杂对象 B 和 C。如果明确服务提供者只需要用到 A 聚合的 C 对象, 则请求参数应该是 C, 而不是整个对象 A。
- 4) 设置合适的客户端超时时间, 防止业务高峰期因为服务端响应慢导致业务线程等应答时被阻塞, 进而引起后续其他服务的消息在队列中排队, 造成故障扩散。
- 5) 对于重要的服务, 可以单独部署到独立的服务线程池中, 与其他非核心服务做隔离, 保障核心服务的高效运行。
- 6) 利用 Docker 等轻量级 OS 容器部署服务, 对服务做物理资源层隔离, 避免虚拟化之后导致的超过 20% 的性能损耗。
- 7) 设置合理的服务调度优先级, 并根据线上性能监控数据做实时调整。

21.2 事务一致性问题

服务化之前, 业务采用本地事务, 多个本地 SQL 调用可以用一个大的事务块封装起来, 如果某一个数据库操作发生异常, 就可以将之前的 SQL 操作进行回滚, 只有所有 SQL 操作全部成功, 才最终提交, 这就保证了事务强一致性, 如图 21-4 所示。

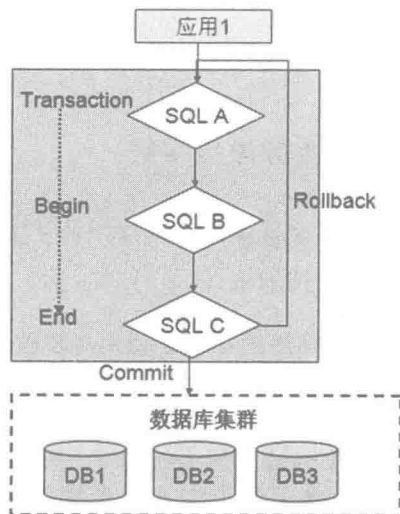


图 21-4 本地事务工作原理

服务化之后，三个数据库操作可能被拆分到独立的三个数据库访问服务中，此时原来的本地 SQL 调用演变成了远程服务调用，事务一致性无法得到保证，如图 21-5 所示。

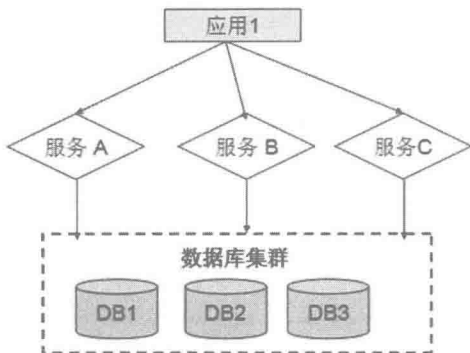


图 21-5 服务化之后引入分布式事务问题

假如服务 A 和服务 B 调用成功，则 A 和 B 的 SQL 将会被提交，最后执行服务 C，它的 SQL 操作失败，对于应用 1 消费者而言，服务 A 和服务 B 的相关 SQL 操作已经提交，服务 C 发生了回滚，这就导致事务不一致。从图 21-5 可以得知，服务化之后事务不一致主要是由服务分布式部署导致的，因此也被称为分布式事务问题。

21.2.1 分布式事务设计方案

通常，分布式事务基于两阶段提交实现，它的工作原理示意图如图 21-6 所示。

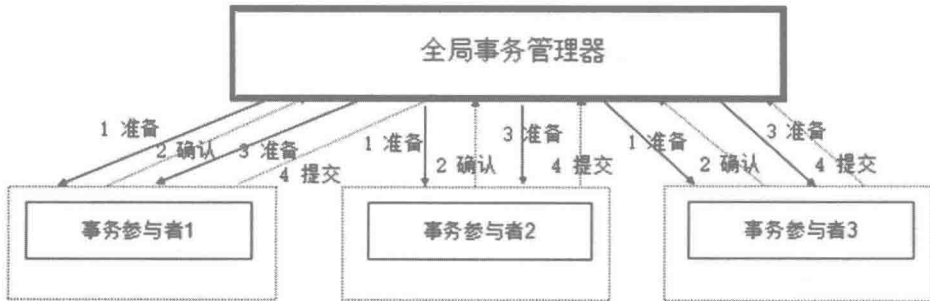


图 21-6 两阶段提交原理图

阶段 1：全局事务管理器向所有事务参与者发送准备请求；事务参与者向全局事务管理器回复自己是否准备就绪。

阶段 2：全局事务管理器接收到所有事务参与者的回复之后做判断，如果所有事务参与者都可以提交，则向所有事务提交者发送提交申请，否则进行回滚。事务参与者根据全局事务管理器的指令进行提交或者回滚操作。

分布式事务回滚原理图如图 21-7 所示。

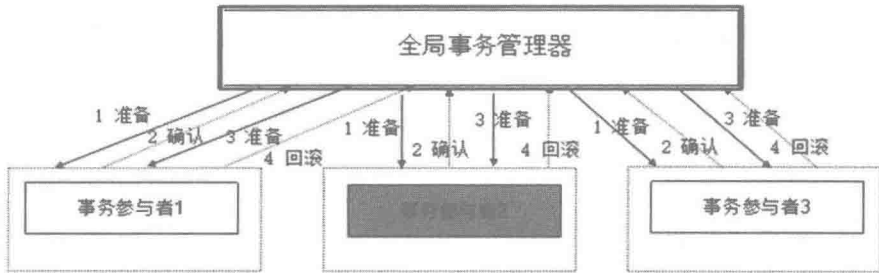


图 21-7 分布式事务回滚原理图

两阶段提交采用的是悲观锁策略，由于各个事务参与者需要等待响应最慢的参与者，因此性能比较差。第一个问题是协议本身的成本：整个协议过程是需要加锁的，比如锁住数据库的某条记录，且需要持久化大量事务状态相关的操作日志。更为麻烦的是，两阶段锁在出现故障时表现出来的脆弱性，比如两阶段锁的致命缺陷：当协调者出现故障，整个事务需要等到协调者恢复后才能继续执行，如果协调者出现类似磁盘故障等错误，该事务将被永久遗弃。

对于分布式服务框架而言，从功能特性上需要支持分布式事务。在实际业务使用过程中，如果能够通过最终一致性解决问题，则不需要做强一致性；如果能够避免分布式事务，则尽量在业务层避免使用分布式事务。

21.2.2 分布式事务优化

既然分布式事务有诸多缺点，那么为什么我们还在使用呢？有没有更好的解决方案来

改进或者替换呢？如果我们只是针对分布式事务去优化的话，发现其实能改进的空间很小，毕竟瓶颈在分布式事务模型本身。

那我们回到问题的根源：为什么我们需要分布式事务？因为我们需要各个资源数据保持一致性，但是对于分布式事务提供的强一致性，所有业务场景真的都需要吗？大多数业务场景都能容忍短暂的不一致，不同的业务对不一致的容忍时间不同。像银行转账业务，中间有几分钟的不一致时间，用户通常都是可以理解和容忍的。

在大多数的业务场景中，我们可以使用最终一致性替代传统的强一致性，尽量避免使用分布式事务。

在实践中常用的最终一致性方案就是使用带有事务功能的 MQ 做中间人角色，它的工作原理如下：在做本地事务之前，先向 MQ 发送一个 `prepare` 消息，然后执行本地事务，本地事务提交成功的话，向 MQ 发送一个 `commit` 消息，否则发送一个 `rollback` 消息，取消之前的消息。MQ 只会在收到 `commit` 确认才会将消息投递出去，所以这样的形式可以保证在一切正常的情况下，本地事务和 MQ 可以达到一致性。但是分布式调用存在很多异常场景，诸如网络超时、VM 宕机等。假如系统执行了 `local_tx()` 成功之后，还没来得及将 `commit` 消息发送给 MQ，或者说发送出去由于网络超时等原因，MQ 没有收到 `commit`，发生了 `commit` 消息丢失，那么 MQ 就不会把 `prepare` 消息投递出去。MQ 会根据策略去尝试询问（回调）发消息的系统（`checkCommit`）进行检查该消息是否应该投递出去或者丢弃，得到系统的确认之后，MQ 会做投递还是丢弃，这样就完全保证了 MQ 和发消息的系统的一致性，从而保证了接收消息系统的一致性。

21.3 研发团队协作问题

服务化之后，特别是采用微服务架构以后。研发团队会被拆分成多个服务化小组，例如 AWS 的 Two Pizza Team，每个团队由 2~3 名研发负责服务的开发、测试、部署上线、运维和运营等。

随着服务数的膨胀，研发团队的增多，跨团队的协同配合将会成为一个制约研发效率提升的因素。

21.3.1 共用服务注册中心

为了方便开发测试，经常会在线下共用一个所有服务共享的服务注册中心，这时，一个正在开发中的服务发布到服务注册中心，可能会导致一些消费者不可用。

解决方案：可以让服务提供者开发方，只订阅服务（开发的服务可能依赖其他服务），而不注册正在开发的服务，通过直连测试正在开发的服务。

它的工作原理如图 21-8 所示。

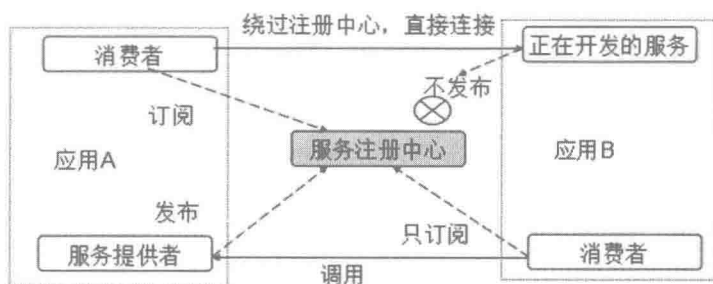


图 21-8 只订阅，不发布

21.3.2 直连提供者

在开发和测试环境下，如果公共的服务注册中心没有搭建，消费者将无法获取服务提供者的地址列表，只能做本地单元测试或使用模拟桩测试。

还有一种场景就是在实际测试中，服务提供者往往多实例部署，如果服务提供者存在 Bug，就需要做远程断点调试，这会带来两个问题：

- 1) 服务提供者多实例部署，远程调试地址无法确定，调试效率低下。
- 2) 多个消费者可能共用一套测试联调环境，断点调试过程中可能被其他消费者意外打断。

解决策略：绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点

对点直联方式将以服务接口为单位，忽略注册中心的提供者列表。

21.3.3 多团队进度协同

假如前端 Web 门户依赖后台 A、B、C 和 D 4 个服务，分别由 4 个不同的研发团队负责，门户要求新特性 2 周内上线。A 和 B 内部需求优先级排序将门户的优先级排的比较高，可以满足交付时间点。但是 C 和 D 服务所在团队由于同时需要开发其他优先级更高的服务，因此把优先级排的相对较低，无法满足 2 周交付。

在 C 和 D 提供版本之前，门户只能先通过打测试桩的方式完成 Mock 测试，但是由于并没有真实的测试过 C 和 D 服务，因此需求无法按期交付。

应用依赖的服务越多，特性交付效率就越低下，交付的速度取决于依赖的最迟交付的那个服务。假如 Web 门户依赖后台的 100 个服务，只要 1 个核心服务没有按期交付，则整个进度就会延迟。

· 解决方案：调用链可以将应用、服务和中间件之间的依赖关系串接并展示出来，基于调用链首入口的交付日期作为输入，利用依赖管理工具，可以自动计算出调用链上各个服务的最迟交付时间点。通过调用链分析和标准化的依赖计算工具，可以避免人为需求排序失误导致的需求延期。

21.3.4 服务降级和 Mock 测试

在实际项目开发中，由于小组之间、个人开发者之间开发节奏不一致，经常会出现消费者等待依赖的服务提供者提供联调版本的情况，相互等待会降低项目的研发进度。

解决方案：服务提供者首先将接口定下来并提供给消费者，消费者可以将服务降级同 Mock 测试结合起来，在 Mock 测试代码中实现容错降级的业务逻辑（业务放通），这样既完成了 Mock 测试，又实现了服务降级的业务逻辑开发，一举两得。

21.3.5 协同调试问题

在实际项目开发过程中，各研发团队进度不一致很正常。如果消费者坐等服务提供者按时提供版本，往往会造成人力资源浪费，影响项目进度。

解决方案：分布式服务框架提供 Mock 桩管理框架，当周边服务提供者尚未完成开发时，将路由切换到模拟测试模式，自动调用 Mock 桩；业务集成测试和上线时，则要能够自动切换到真实的服务提供者上，可以结合服务降级功能实现。

21.3.6 接口前向兼容性

由于线上的 Bug 修复、内部重构和需求变更，服务提供者会经常修改内部实现，包括但不限于：接口参数变化、参数字段变化、业务逻辑变化和数据表结构变化。

在实际项目中经常会发生服务提供者修改了接口或者数据结构，但是并没有及时知会到所有消费者，导致服务调用失败。

解决方案：

- 1) 制定并严格执行《服务前向兼容性规范》，避免发生不兼容修改或者私自修改不通知周边的情况。
- 2) 接口兼容性技术保障：例如 Thrift 的 IDL，支持新增、修改和删除字段，字段定义位置无关性，码流支持乱序等。

21.4 总结

服务化之后，无论是服务化框架，还是业务服务，都面临诸多挑战，本章摘取了其中一些比较重要的问题，并给出解决方案和最佳实践。对于本章节没有列出的问题，则需要服务框架开发者和使用者在实践中探索，找出一条适合自己产品的服务化最佳实践。