# 作者简介

王佩华

毕业于天津理工大学，目前就职于京东公司，任架构师，从事多年金融及电信行业分布式系统架构的设计及研发智理工作。

多项微服务领域专利的发明人，热衷于参与开源社区并主导了公司微服务项目的开源和推广，擅长通信协议设计、分布式中间件设计、高并发系统架构设计及云原生架构等。

## 内容简介

在当今的数字化经济时代，微服务架构已经成为公司业务构建的主流架构模式，代表了未来的技术发展趋势，同时微服务也成为开发者的必备技能。

本书从微服务架构的设计理念和方法论切入，从不同角度全面介绍微服务特性、使用场景、组织流程、构建交互、部署交付等软件工程各个关键环节和核心要素，既包含了具体微服务技术的源码解读、原理分析，也加入了作者在电信、金融领域积累的真实案例和实践经验。

全书分为原理篇、实践篇、进阶篇。原理篇涵盖微服务的概念、采用前提、领域驱动设计、DevOps；实践篇对Spring Boot、Spring Cloud 治理框架、系统集成、微服务数据架构、微服务交付、微服务的监控等重要技术话题展开深入讲解；进阶篇主要介绍函数式编程及响应式微服务架构、Kubernetes、云原生架构生态。

本书不仅适合初学者深入理解微服务架构，也可以作为团队管理者或者架构师进阶微服务架构的技术参考手册。

## 专家力荐

微服务化是近年来系统架构领域的一场重要变革，本书作者不仅从理论上带大家理解微服务，还难能可贵地结合了自己在实际业务中的微服务落地实践。

相信这些凝结了作者智慧的宝贵经验，会让有志于此领域的读者受益良多。

——**祁宁** SegmentFault社区CTO，Typecho作者

微服务经过了长足的发展，在每个阶段所产生的信息都很多。在信息爆炸的当今，找到一本将信息梳理得井井有条的好书，是提升学习效率的最佳途径。

本书层次分明，分为原理篇、实践篇和进阶篇，适用于广泛的人群。理论篇对新手入门非常友好，实践篇非常适合在工作中解决实际问题的开发者，进阶篇则面向响应式编程和云原生架构，是高手的必备技能。

——**张亮** Apache Member，ShardingSphere PMC Chair

本书讲解了微服务架构落地过程中的领域驱动设计、服务注册与发现、负载均衡、限流熔断、网关和微服务监控等实战技巧，并从Service Mesh、Serverless、云原生等视角讲解了未来微服务架构的走向。本书值得一读，读者必定能从中取长补短，构建或补充自己的微服务架构知识体系。

——**许进** 《重新定义Spring Cloud实战》作者，Spring Cloud中国社区（spring cloud.cn）创始人

本书结构清晰，从原理、实践、进阶三个方面对微服务架构进行深度解析。涵盖内容丰富，从领域驱动设计到微服务治理，从Spring Cloud生态到响应式微服务架构体系建设。如果你正在进行Java微服务架构设计，或者正打算快速学习基于Spring Cloud的微服务架构，本书将为你节省许多宝贵时间。

——**黄勇** 《架构探险》作者，阿里巴巴前高级架构师

佩华的新书内容覆盖面非常广。从分布式系统到微服务架构，从Spring Boot到Spring Cloud的各种组件，从Docker到Kubernetes，从领域驱动设计到响应式编程和云原生开发，都有作者很多独到的见解。致力于微服务开发的工程师从中汲取并归纳为自己的知识，真是再好不过了。

——**梁鑫** SIA开源项目创始人，互联网金融公司高级架构师

本书深入浅出地讲解了微服务的前生今世，包含了丰富的实践经验，对微服务架构进行了全面的解构，无论是广度还是深度都颇为经典，个人觉得踩坑复盘经验最为珍贵，也是国内微服务方面少有的能够写这么详细的图书。

——**耿航** 中国电子技术标准化研究院木兰开源社区运营负责人

本书对微服务的理论和实践，都进行了由浅入深的剖析，同时从技术体系的纵深出发，对微服务架构的技术栈进行逐一拆解，指导实践落地。对于初识微服务的同学，可以较快地理解微服务，包括理念、概念、架构思想、边界等，在结合实践的同时快速上手微服务架构；对于资深的微服务玩家，这是一个完整的、体系化的复盘和总结，对于微服务架构的利与弊、架构处理的平衡都进行了详细的分解与剖析，有助于推动读者对架构演进的思考。

——**张真** 百信银行首席架构师

在云原生时代，微服务架构是企业PaaS化演进及中台建设的必备技能。本书系统性地解析了微服务架构领域相关的理论知识、实践落地经验及技术发展趋势，可以帮助微服务相关人员构建知行合一的能力，对设计开发分布式应用系统有很好的指导作用，也是技术管理者技术选型的参考书。

——**向江旭** 美的集团IoT技术副总裁兼首席技术官

微服务是当前软件开发的发展趋势，越来越多的系统采用微服务方式构建。作者基于十余年的经验积累，深入浅出地介绍了微服务系统的原理、模式、最佳实践，为我们构建微服务系统提供了非常有价值的指引和参考。我向大家推荐这本书。

——**高崇** 伦敦证券交易所集团高级技术专家

本书中的内容来源于作者多年的工作积累和实践总结，从理论到实践再到进阶，以全方位递进的方式对微服务的设计和应用进行了解读，能够让大家在日常开发工作中少走弯路，有很强的指导意义。

——**王新栋** 极客时间《OAuth 2.0实战》专栏作者，京东技术专家

# 前言

当前，微服务架构在国内正处于蓬勃发展的阶段，无论是大型互联网公司还是传统的IT企业，纷纷采用微服务架构构建系统。在过去几年里，DevOps、云原生、面向演进式架构等理念已经深入人心，围绕微服务生态也出现了大量的组件、框架、工具，这很好地支撑了海量的数据增长和用户业务需求的快速变化。本书将从微服务理论开始介绍，结合作者多年的工作经验，深入讲解分布式系统和微服务架构，从而帮助技术人员切实掌握微服务架构技术。

## 缘起

“物之所至者，志亦至焉；志之所至者，礼亦至焉；礼之所至者，乐亦至焉；乐之所至者，哀亦至焉。”

——楚简《民之父母》

2008年，在一次软件架构设计高端课程中，有人将中国传统的“五至”与软件工程做了形象的类比，提到从事软件开发工作的工程师要“以物为本”。这次课程学习让我印象深刻，也让刚刚从事软件开发工作的我第一次对“架构”这个词有了直观而深刻的认识，这次课程学习成了我后来的架构师之路的灯塔。

在“五至”中，对“物、志、礼”进行了明确的排序：先有物，而后有志，最后有礼，所以重点是“物”。我们把这种思想对应到软件工程的项目实践中，可以这样理解：物就是最终的交付物，即软件产品；志对应项目的利益相关人；而礼则对应项目的过程管理。这样的排序不无道理，在项目实践过程中，应该以最终的交付物为中心，所以我们必须重视架构设计，只有经过完善的架构设计，才能呈现最好的软件产品。在项目的初期就要做好架构设计，达到“物之所至”；协调项目的利益相关方，达成共识，达到“志亦至焉”；在项目的进行过程中，做好项目管理和流程监控，达到“礼亦至焉”。

由上可知，软件架构是决定软件工程成败的首要因素，架构之道就是面对不同的问题域找到最佳的技术实现方案。在软件工程中，技术架构同时受到组织、流程管理的影响，然而软件产品最终能否成功仍然需要“以物为本”。

## 与时俱进

纵观计算机产业，过去的半个世纪，遵从摩尔定律的规律，计算力一直保持着大跨度的提高。1956年，英特尔创始人戈登·摩尔提出，集成电路的集成度每两年会翻一番；而后这个周期缩短到18个月，微处理器的性能每隔18个月提高1倍。然而，硅芯片已逼近物理和经济成本上的极限，许多专业人士纷纷预测，摩尔定律在不久的将来会失效。界时，我们会面临CPU性能提升放缓，计算力增长势微的局面。同时云计算、大数据、物联网、边缘计算、人工智能等技术的进一步成熟所产生的海量数据，却加大了对后端数据中心计算力的需求。

数字化经济的快速发展和云计算给底层IT系统带来的巨大变革正是当下微服务架构快速发展的时代背景。Gartner预计，从2018年到2022年，PaaS将成为未来的主流平台交付模式，而PaaS平

台需要更加灵活的云原生应用架构做技术支撑，微服务架构正是云原生架构落地的关键技术。

正所谓“大道至简”，微服务本身是一个化繁为简的过程，它采用细粒度的分布式架构模式，通过系统化的思考方式，将纷繁复杂的业务逻辑映射到底层技术。在软件构建方面，微服务倾向于使用面向服务和领域驱动设计（Domain Driven Design）的方法论，将现实中的问题投影到对象的世界。“抽象、分解、扩展、复用”是常见的微服务构建系统的内功心法。然而软件开发没有“银弹”，架构设计还要从使用、性能、成本、效率、团队、收益等多方面权衡（Trade-off），进行综合考虑。

微服务架构的目标是，将业务与技术的复杂度进行分离，使业务更专注于实现对客户的价值交付，而将非功能需求封装在平台或者底层SDK中。目前在企业的应用开发中，Spring Boot和Spring Cloud平台作为微服务的技术开发框架，依然占据主流地位；而伴随容器和Kubernetes平台的崛起，结合自动化和DevOps持续交付流程，微服务可以显著提升应用交付的效率和产品的质量，此外。微服务还有如下优势：

- 更加快速地响应业务需求。
- 可提升应用的开发效率。
- 可满足对云原生的支持。
- 可满足系统的弹性、扩展性的需求。
- 容错性及生产就绪特性可保证服务的高可用。
- 有标准化的服务实现和交付方式。
- 基于“不可变基础设施”模式，可以减少环境因素的影响。
- 可提升系统性能及资源利用率。

传统粗放式的开发和运维方式将逐渐被取代，而更加高效、智能化、自动化的开发和运维方式将使每个人从中获益。当前，微服务和云原生应用架构还在快速演进之中，其间充满了机遇和挑战。作为软件从业人员，面对技术的更新迭代，我们唯有整装待发，才能与时俱进。

## 关于本书

本书的原理篇将深入讲解当前微服务架构的理念和方法论；实践篇主要讲解Spring Boot和Spring Cloud微服务框架体系；进阶篇关注微服务发展的技术趋势。本书既涵盖了我多年在传统电信企业中积累的分布式架构设计经验，也涵盖了互联网金融行业的前沿技术实践；本书既有我在大型项目中的经历分享和痛点复盘，也有开源软件的案例介绍，以及当下微服务技术趋势的深度剖析和预测。在本书的代码案例中，使用的是Spring Boot 1.5.11，在进阶篇中使用了Spring Boot 2.x，这两种版本在使用和配置方面有诸多不同，并且其中一些功能仅支持JDK1.8及以上版本，读者在阅读中需要注意。

## 内容结构

本书在结构上从3个层次深入解读微服务架构，希望读者能够循序渐进地深入了解微服务架构的理论及技术实践；当然，具备一定微服务架构经验的工程师也可以根据需要选择性地阅读。

- **原理篇**

原理篇主要讲解微服务的概念、微服务与云原生的关系、微服务的主要特性及设计哲学、采用微服务的前提、微服务构建的理论基础及基本原则，以及领域驱动设计和微服务构建方法论、DevOps方法论等相关知识。

- **实践篇**

实践篇主要对微服务架构落地的脚手架、关键技术、系统集成、数据架构、持续交付、服务治理监控等重要技术话题展开讲解。另外，针对初学者对Spring Boot和Spring Cloud体系“入门容易、精通难”的问题，实践篇中结合框架源码详细解构了Spring Boot的框架底层运行机制，并具体介绍了Spring Cloud核心治理组件及其工作原理，还会讲解如何基于开源软件做扩展性开发。

- **进阶篇**

进阶篇主要介绍函数式编程、响应式微服务架构设计原理、常用的响应式编程框架、Spring Boot 2新特性，讲解Kubernetes工作原理、微服务与云原生生态的技术融合演进、微服务未来的技术发展趋势。

由于我在编写本书的同时需要兼顾工作，时间和精力有限，书中难免有不足之处，恳请广大读者批评指正。

## 致谢

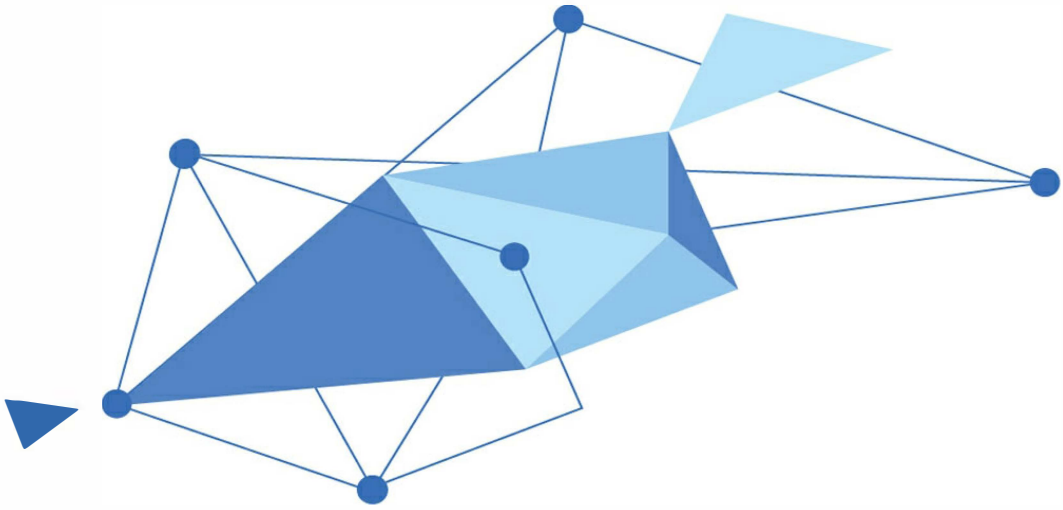
首先感谢工作中的各位同事，感谢生活中的朋友，感谢宜信公司的高蕾涵同学，感谢电子工业出版社董英老师的鼓励和帮助，因为你们我才有动力完成这本书。最后，谨以此书献给我的妻子和家人，感谢你们对我工作的大力支持！

王佩华

## 读者服务

微信扫码回复：41238

- 获取本书配套资源
- 获取作者提供的各种共享文档、线上直播、技术分享等免费资源
- 加入本书读者交流群，与作者互动
- 获取博文视点学院在线课程、电子书20元代金券



# 原理篇

## 本篇内容

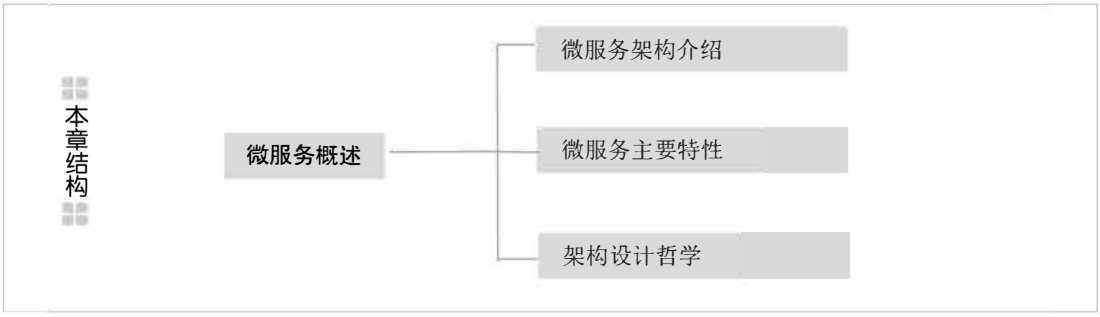
本篇我们会介绍微服务架构迅速发展的时代背景、微服务的定义和主要特性，以及其背后的设计哲学。

我们还将从实际业务场景出发介绍采用微服务架构的前提、如何对单体架构进行微服务化改造、巨石型应用的拆分迁移策略。

同时，针对微服务架构的构建过程，将围绕技术、组织、流程管理等软件工程要素展开详细讨论，深入讲解康威定律、DevOps、领域驱动设计、云原生12要素及相关的概念、理论、架构原则。

第1章

# 微服务概述




微服务的概念来源于Martin Fowler的一篇知名博文： MicroServices 。在博文中，“微服务架构”这个术语用来描述一种将软件应用程序设计为可独立部署的服务套件的特定方式。

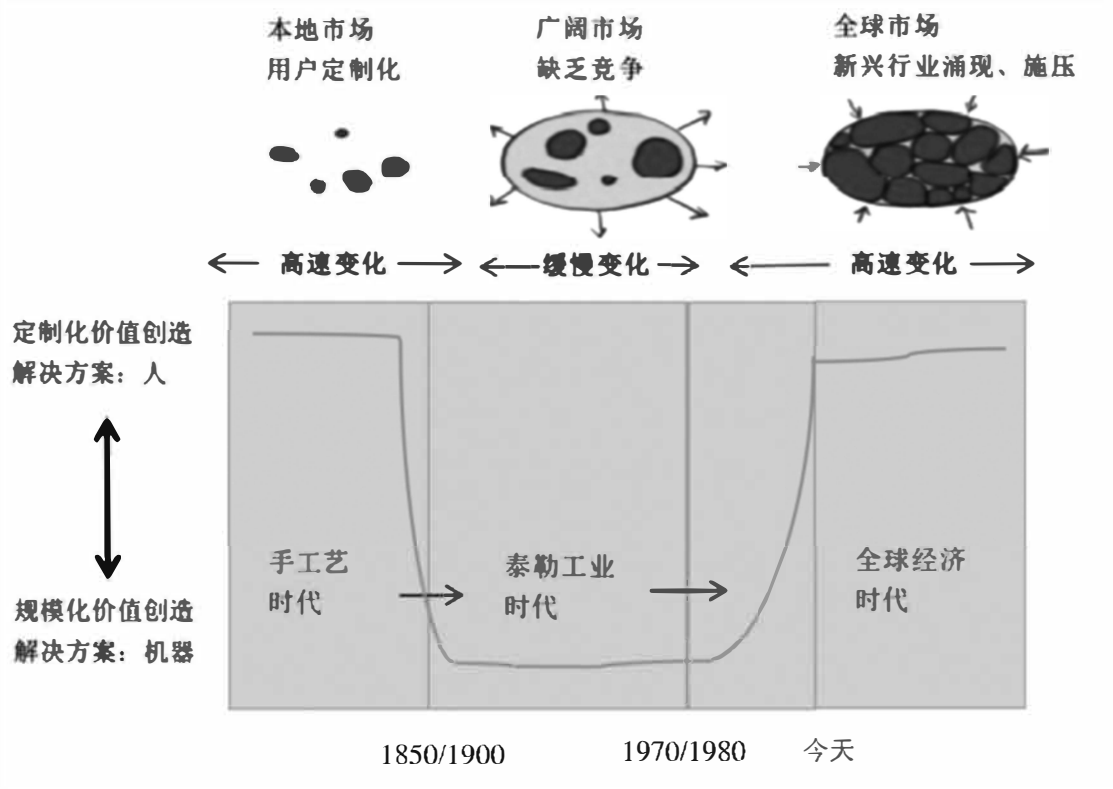
“细粒度自治服务”“自动化部署”“围绕业务能力”“端点智能”“语言和数据的分散控制”，从这些描述微服务架构特征的术语中，我们发现了一种越来越吸引人的软件系统风格。

## 1.1 微服务架构介绍

### 1.1.1 背景介绍

目前不仅各大互联网公司已经在大规模地应用微服务架构，而且传统行业也逐渐接受了这种架构模式，纷纷开始采用微服务架构构建业务系统。为什么微服务架构会如此受欢迎？微服务架构是设计而来还是演变而来的呢？要了解这些问题，我们需要从现代经济模式和企业组织架构入手来了解微服务架构崛起的时代背景。

Niels Pflaeging在 Organize for Complexity 一书中通过“浴缸曲线”（见下图）将西方从20世纪到现在的经济模式划分为三个时代：本地市场和用户定制化的“手工艺时代”；通过机械规模化提升效率和比拼成本，市场广阔而缺乏竞争的“泰勒工业时代”；以知识工人为主体，新兴行业涌现、施压，从而带来市场需求快速变化的“全球经济时代”。




在“手工艺时代”，产品的价值创造完全取决于掌握技艺的手工艺者，局部市场、高度动态化、定制化是这个时代的特点，但这种模式很难做到规模化地生产和持续地输出价值。在“泰勒工业时

代”，主流的组织是上传下达的“命令控制型”组织，更适合简单、重复的规模化生产，但这种组织架构的不足是对市场响应慢，在应对复杂变化方面十分脆弱。而在“全球经济时代”，由新兴行业带领，逐渐兴起更多扁平或分散的复杂的自适应组织，这种架构模式更加倾向于跨职能混搭和协作，和市场直接对接，可以快速灵活地响应市场变化。

可以说，组织、系统架构和技术之间隐含着映射关系。从组织所采用的技术栈和架构特征可以快速推断出组织的业务模式和组织架构方式。单体架构、垂直架构、集中式数据库常见于泰勒型组织；而云计算、微服务、DevOps等技术更加适合于复杂的自适应组织。同时，技术架构反过来也受限于强势的组织架构和企业管理文化约束。

回顾早期的软件系统，企业采用单体架构可以快速满足业务的简单诉求，然而随着项目规模的扩大、业务模块的耦合、组织人员的膨胀，使得单体架构冗余而数量庞大的代码越来越无法适应企业灵活应对变化的需求，大量紧耦合的代码导致应用的模块界限日益模糊，业务发展急需匹配高可用、可扩展、隔离性好、复用性强的应用系统架构，而传统的单体架构显然无法满足企业的需求。

在 Microservice Pattens 一书中，作者使用餐饮应用FTGO（Food to Go）举例，说明了系统是如何一步步走向单体地狱的。系统的过度复杂使得业务逻辑耦合、开发速度缓慢、交付周期长、难以扩展，这给开发人员带来了极大的挫折感，生产效率也随之大幅下降。单体架构在运行状态下，出现故障难以隔离，局部异常问题往往会影响整个系统的正常运行，最终导致整体服务的不可用，给业务人员造成的收入损失、给客户造成的糟糕体验都让人无法忍受。

正是在这样的时代背景和业务诉求下，微服务架构成为了解决复杂问题的灵丹妙药。微服务架构在应对需求的变化、容错处理、服务复用及扩展、提升开发效率、简化交互等方面都有明显的优势。同时，敏捷、DevOps、持续集成/持续交付、容器技术、Spring Cloud框架、轻量级服务、领域驱动设计等的涌现也为微服务架构的发展奠定了基础。

综上所述，持续快速响应市场、高度动态化、应对复杂场景的能力已经成为企业的核心竞争力，企业越来越需要一个能够面对变化、并且能够主动拥抱变化的软件架构，而微服务架构正是在这样的时代大背景下逐渐发展壮大起来的。

### 1.1.2 微服务的定义

微服务并没有一个明确的官方定义，它可以解释为一种架构编程思维，更多地被描述为一种架构风格。微服务架构的概念可以说来源于技术专家多年的工作积累和最佳实践总结，是通过不断发展、演进逐渐形成的。

**架构演进论**

在“技术雷达”里，微服务最早以“Micro-service”，而非“MicroService”出现，从架构演进的角度来说，微服务是从SOA（Services Oriented Architecture，面向服务架构）发展演进而来的，是更先进的细粒度的SOA实现方式。

SOA和微服务本质上都是分布式的架构模式，但是传统的基于ESB（Enterprise Service Bus，企业服务总线）的SOA架构和微服务架构要解决的问题其实并不相同。在J2EE（Java 2 Platform Enterprise Edition，Java企业级应用平台）的企业架构中，ESB在异构系统的集成方面发挥功能。而微服务架构更多地采用轻量级的通信协议，围绕业务进行服务拆分、解耦、复用、隔离，实现细粒度的分布式服务构建和管理。

微服务架构对传统SOA架构最大的改变是强化端点（Endpoints）和弱化通道（Dumb Pipe），抛弃了ESB过度复杂的业务规则、编排、消息路由等功能。微服务强调服务的自治性，服务应用从一开始就可以独立地进行开发和演进。另外，微服务间的通信应尽可能地轻量化，微服务在SOA的基础上更加关注细粒度服务的独立性、可扩展性、伸缩性和容错性等。

**最佳实践论**

微服务的另外一种定义来源于架构大师们多年的最佳实践总结，Martin Fowler于2014年在他自己的博文中首次提出微服务的定义，概括总结如下：

“微服务架构”是一种将单个应用程序作为一套小型服务开发的方法，服务之间相互协调、互相配合，每个服务运行在其独立的进程中，并以轻量级机制（通常采用HTTP协议）进行交互通信。这些服务是围绕业务功能构建的，它们可以通过全自动部署机制进行独立部署。微服务采用去中心化的管理理念、可以用不同的编程语言编写，并使用不同的数据存储技术。总结起来微服务有以下几大特征：

- 通过服务组件化。
- 围绕业务能力组织。
- 是产品不是项目。
- 智能端点和哑管道。
- 去中心化治理。
- 去中心化数据管理。
- 基础设施自动化。
- 为失效设计。
- 演进式设计。

可以看出，Martin Fowler试图将微服务定义为一个一般化的架构“最佳实践”集合。微服务的这些特征也很好地融合了领域驱动设计、自动化、DevOps、容器等先进的技术实践、架构理念和方法论。

与此同时，微服务的架构风格被描述为一种可以实现业务功能松散耦合（Loosely Coupled）的、具备一定服务边界（Bounded Context）的服务集合。这种架构风格使得大型、复杂的应用实现CI/CD（Continuous Integration/Continuous Delivery，持续集成/持续交付）成为可能，并且技术栈可以独立发展演进。

归根结底，微服务本质上还是一种分布式系统架构。它强调系统应该按照业务领域边界做细粒度的拆分和部署；它剔除了SOA中的企业服务总线，使用轻量级、标准化的HTTP（REST API）协议进行交互集成。微服务架构有利于应用应对业务的快速变化和规模化发展，通过对软件复杂性的有机治理，使系统易于有序化重构及扩展。

### 1.1.3 微服务与云原生

云原生（Cloud Native）可以理解为一系列技术及思想的集合，既包含微服务、容器等技术载体，也包含DevOps（开发与运维的合体）的组织形式和沟通文化。企业采用基于云原生的架构进行构建、运行、管理现代应用的技术模式能够平滑而快速地将业务迁移到云上，享受云的高效性和按需伸缩的能力。

**云原生的提出**

“效率”：天下武功、唯快不破，面对激烈的市场竞争，企业把服务产品快速交付的能力作为制胜的法宝。云原生架构的提出与应用的快速开发、快速交付的能力密不可分。作为对比，与传统企业为应用提供和部署软件按周、按月来计算，互联网公司经常在一天内可以进行上百次发布。这种快速迭代和部署是建立在云基础设施和自动化持续交付能力之上的。

“弹性”：随着用户规模和需求的增长，我们的应用需要能够快速扩展，提高服务能力。传统企业依靠购买硬件的方式来提升和扩展服务能力，而云原生架构可以通过虚拟化的技术实现按需扩展，动态地扩展服务实例以满足计算、存储、服务资源的弹性需求。

“可靠”：服务仅仅做到快速交付和弹性扩展是远远不够的，还需要兼顾系统的稳定性、可用性、持久性，而这种特征与服务的容错能力、故障隔离能力、可视化能力、系统快速恢复能力紧密相关，也就是系统所谓的“反脆弱”能力。现代应用如果想在这些“非功能需求”上得到保障，就需要采用云原生技术和云原生基础设施保障服务的高可用性。

**云原生与微服务**

“云原生”的本质和目标是一种应用模式，它能够帮助企业快速、持续、可靠、规模化地交付软件，其中关键的支撑组件总结如下。

- 容器化的抽象封装：标准化代码和服务，每个部分（应用程序、进程等）都封装在自己的容器中，有助于复用和资源隔离。实现方式代表有Docker、rkt。
- 动态管理：通过集中式的编排和调度系统来动态管理及优化资源。实现方式代表有Kubernetes、Swarm和Mesos。
- 面向微服务：应用程序基于微服务架构，显著提升开发效率、提高架构演进的灵活性和可维护性。实现方式代表有Spring Boot、Spring Cloud。

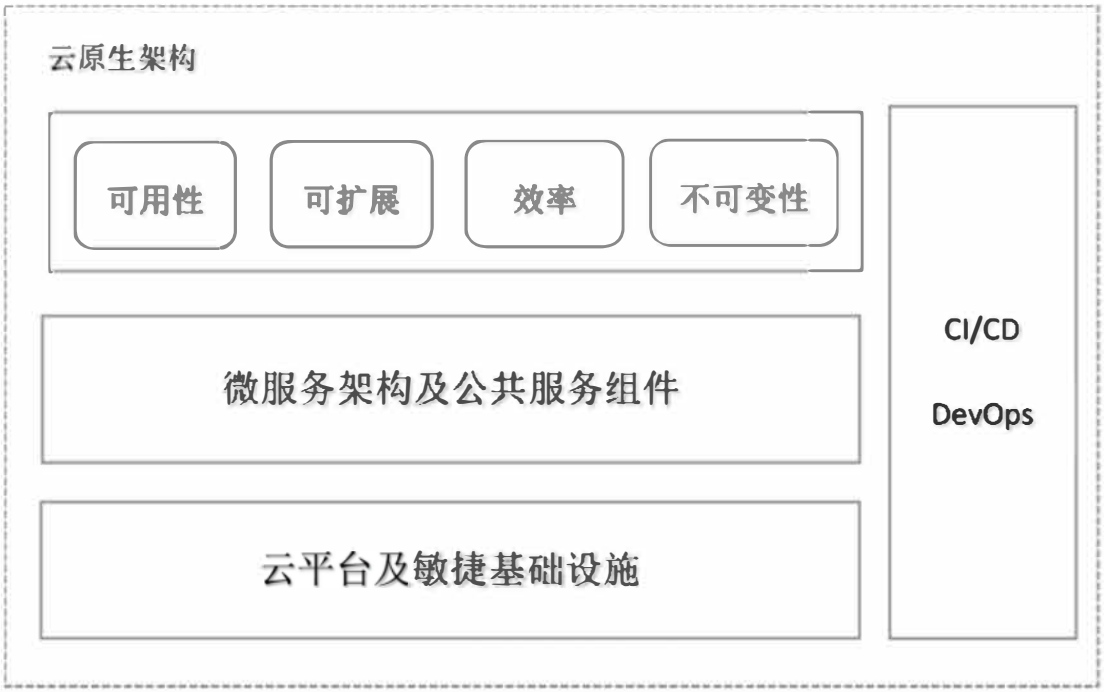
从云原生的代表组件可以看出，云原生的主要组成技术包括容器、服务编排管理和微服务技术。云原生可以说是从概念上统一了构建、交付、运行现代应用的最佳实践集合。在运行环境上，强调应用程序的运行环境是以容器和Kubernetes为主的云基础设施；在流程管理上，主要配合使用持续集成、持续发布以及DevOps能力；在软件开发上，基于微服务架构构建现代应用程序和软件。虽然微服务架构也可以运行在传统虚拟机或物理机上，但是微服务架构的最佳运行载体是以容器为代表的云原生环境。

**云原生架构**

云原生架构的思想和特征可以通过云原生12要素表达，我们在后面的章节中会详细讲解12要素的主要内容，从实践的角度，作为最早践行Cloud Native的Netflix，在云原生架构的目标、原则和措施等方面进行了详尽描述：

- 不可变性。使用易失效的基础设施来构建高敏捷（Highly Agile）、高可用（Highly Available）服务。其目的是为了提高伸缩性（Scalability）、可用性（Availability）、敏捷性（Agility）、效率（Efficiency）。
- 关注点分离。通过微服务架构实现关注点分离，避免出现“决策瓶颈”。实际上，实现关注点分离有助于提升系统的可扩展性和可用性。
- 反脆弱性。默认所有的依赖都可能失效，在设计阶段就要考虑如何处理这些失效问题。为了让系统更强壮，Netflix会不断地攻击自己、主动破坏，以提醒技术人员系统要进行反脆弱性设计。
- 高度信任的组织。Netflix基于信任的管理风格，相信自己的员工可以做出正确的决策，倡导给基层员工自主决策权。
- 共享。在Netflix，管理是比较透明的，共享能够促进技术人员的成长。

结合云原生的特征和云原生的架构实践，我们将云原生架构图描述如下（见下图），从架构层面来说，Cloud Native是构建在不可变基础设施（以容器技术为代表）和以微服务架构技术为基础的分布式系统之上的，这里的云包括私有云、公有云和混合云。微服务架构作为云原生概念的核心组成部分，本质上就是保证我们更好地适应云环境下的高效开发和运维。




### 云原生成熟度

对于系统采用云原生架构的程度，我们可以用云原生成熟度模型进行精确的等级划分和判断，成熟度模型一共包含4个等级（从Level 0到Level 3成熟度逐级递增），从这些等级划分可以看到开发和运行应用程序的原则、模式和具体技术实践。如果你所在公司正在进行微服务的改造或者云原生架构的迁移，可以参考如下表所示的成熟度模型正确地评估你所在组织的云原生架构等级。

| 级别 | 描述 | 关键技术 |
| --- | --- | --- |
| Level 3：适应性 | 应用程序能够以全自动的方式检测或预测变化并对其做出反应。<br>应用程序管理和控制功能从应用程序中抽离出来，或者使用外部应用程序控制服务 | 人工智能分析决策<br>智能化运维<br>高度自动化<br>成熟的公共服务和基础设施 |
| Level 2：抽象 | 应用程序必须与基础架构完全分离。<br>抽象出应用程序蓝图、部署策略、扩展策略、关联和布局规则等。<br>应用程序服务必须是弹性的和可适应的。应用程序也应该被设计，以便一个服务的失败不会级联到其他服务。<br>应用程序由多个服务组成，每个服务的设计都是弹性的、可适应的、可组合的、最小的和完整的 | 微服务治理平台<br>容器资源调度平台<br>CI/CD发布平台<br>分布式存储、消息中间件<br>分布式数据库 |
| Level 1：松耦合 | 应用服务满足隔离、自治特性并与运行环境分离。<br>应用服务代码与配置文件存储和数据管理层解耦分离。<br>应用服务与网络依赖分离，可以实现动态服务注册与发现 | 微服务框架<br>初步的持续交付能力<br>配置中心、注册中心 |

续表

| 级别 | 描述 | 关键技术 |
| --- | --- | --- |
| Level 0：虚拟化 | 应用程序运行在虚拟机或云实例上。<br>应用程序创建不可变的应用程序映像 | 虚拟化隔离<br>模块化<br>负载能力 |

## 1.2 微服务主要特性

### 1.2.1 粒度更细的服务

微服务架构相比SOA分布式架构强调按业务边界做细粒度的服务拆分。SOA架构使用粗粒度的服务模式来封装业务和技术能力，减少服务交互，但同时带来了业务耦合的复杂性。而微服务架构本质上是一个做减法的架构，将规模庞大的单体系统进行服务拆分，每个细粒度服务的功能和职责单一。当然，服务的粒度并不是拆得越细越好，如果拆分不当，还会造成服务频繁地跨网络操作，增加系统的整体复杂性。

首先，微服务粒度的划分要求工程师充分理解和洞察业务领域的边界，保证你所拆分的服务是自包含的。所谓“自包含”就是说你的服务是可以独立部署、独立演进的，你的服务可以自主地完成某个特定的、单一的功能。

其次，细粒度服务应该同时具备高内聚和低耦合两个特征。高内聚要求将系统中相关的元素和行为聚集在一起，把不相关的元素和行为放在别处；低耦合是指降低微服务之间的相互依赖程度和相互作用关系，如果服务之间存在紧密联系，说明它们的耦合度比较高，最好不要做拆分操作，而应该做聚合操作，这样可以使信息的传递和协作比拆分成独立的服务更加简单可控。

另外，细粒度服务应该尽量做到独立。这一特性也适用于单一职责原则：SRP（Single Responsibility Principle），该原则由Robert C.Martin提出。从面向对象设计的角度看，所谓职责是指一个类（Class）变化的原因。如果一个类有多个改变动机，那么这个类就具有多个职责，而单一职责原则就是指一个类或者模块应该有且只有一个改变原因。

下面总结一下粒度更细的服务带来的好处：

- 粒度更细的服务使每一个服务专注做好一件事情。每个服务完成一个单一任务，在功能不变的情况下，应用被拆分为多个可管理的服务，很好地解决了系统的复杂性问题。
- 粒度更细的服务有助于新人对工程的学习。对于一个大型的、生命周期比较长的项目，人员的流动和组织变化是经常发生的事情，而庞大的单体架构容易使模块之间相互耦合，功能界限模糊，同时增加了新人的学习成本。

- 粒度更细的服务有利于部署。对于大型单体项目，模块之间往往存在紧密的代码耦合，一个子模块的编译错误往往会导致整个应用无法构建成功，而细粒度的服务可以通过独立工程解决“牵一发而动全身”的问题。
- 粒度更细的服务具备更好的复用性。在软件领域，我们一直提倡使用复用的方式构建系统，粒度更细的服务通过独立的部署，通过声明语言无关、平台无关的标准接口（REST API、gRPC）对外暴露服务，实现了积木式的架构搭建模式，提高了软件整体的开发效率。

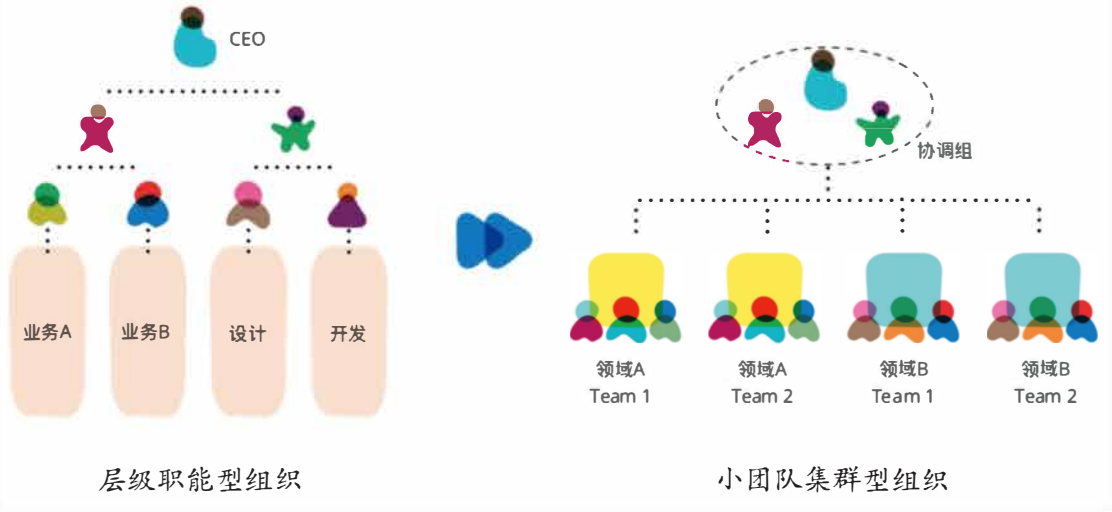
### 1.2.2 围绕业务划分团队

传统的IT企业习惯根据人员掌握的技能来划分组织。例如，熟悉前端的同事，都集中在一个前端开发团队；熟悉数据库的同事，一般都会集中在DBA（Database Administrator，数据库管理员）团队；熟悉测试的同事，专门成立一个测试团队专职做测试工作。我们习惯于将这样的团队称为“职能型组织”，它的优势是资源集中，有利于同一职能内部的专业人士交流和经验积累。

然而，职能型组织最大的问题是团队之间不容易协调利益冲突，容易形成部门墙或者叫部门壁垒。当职能部门有多个项目同时进行时，就会产生资源失衡问题，不利于各职能部门之间的沟通交流和团结协作。业务的需求变化如果牵涉多个职能型组织所负责的模块协作联调，往往会出现项目排期问题、优先级问题，对于跨地域、跨国家的组织，还会出现时差问题、沟通及文化差异问题，这个时候反而增加了团队之间的沟通和协调成本，降低了开发效率。

微服务架构更加提倡以业务为中心，强调围绕业务领域来划分团队。团队由具备不同能力象限的人员组成，而这样的全功能型团队相比职能型团队可以防止人员之间的互相扯皮、互相指责的问题。同一个团队围绕业务领域沟通效率更高，团队合作更加积极主动，有更强的主人翁意识（Ownership）。从技术的维度看，微服务架构倾向于在指定范围的“业务界限上下文”中定义标准规范的交互方式，这样能够保证业务接口（API）更加稳定，在后续服务的迭代升级过程中具备更好的业务兼容性和可演进性。

综上所述，在围绕业务构建微服务架构的时候，解决的一个本质问题就是人员分工的问题，正如康威定律所说，任何组织所设计的系统、所交付的软件产品方案在结构上都应该与该组织的沟通结构和组织方式保持一致。下面是组织结构演进示意图。



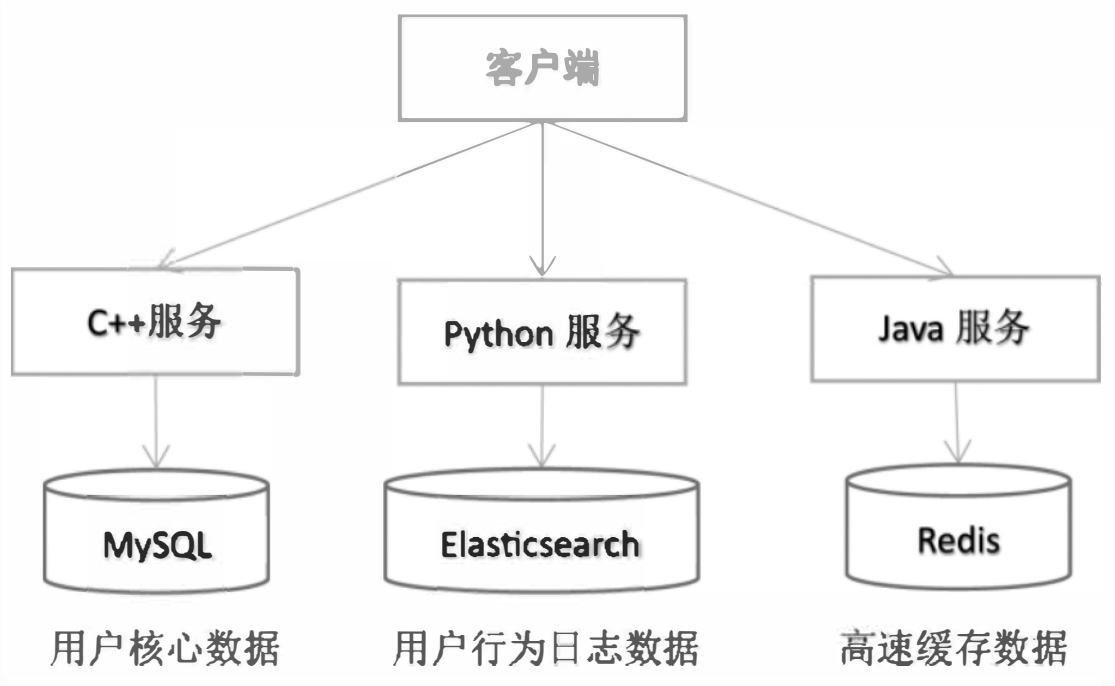

层级职能型组织　　小团队集群型组织

### 1.2.3 技术多样性

微服务架构不限定提供服务方所使用的技术栈和技术选型。微服务架构倾向于服务之间使用标准的轻量级的通信协议（HTTP）完成服务的集成和通信。例如，对于性能要求比较高、对网络通信效率比较关注的服务，可以使用C++语言构建；对于文本分析性的业务，可以采用Python脚本语言；而对于企业应用级的Web项目，使用Java语言开发比较合适。可见，每一种语言和技术都有其“擅长”的场景和适合解决的问题。

微服务架构提倡数据存储的多样性和独立性。不同的数据存储引擎有各自擅长处理的业务类型数据。对于公司的核心业务即OLTP（On-Line Transaction Processing，联机事务处理）业务，可能会采用MySQL这样的关系型数据库。关系型数据库的特点是遵循ACID原则 注 ，对事务的一致性有更好的支持，通过标准的SQL语言就可以方便地实现结构化数据的查询和更新。

在NoSQL数据库阵营中，对于日志数据，可以存放在Elasticsearch这样的LSM树数据结构存储引擎中，适合日志搜索、查询操作；对于分布式系统之间的共享数据，采用Redis这样的内存引擎，在读写效率、高并发性能上有更大的优势；如果是文档型数据，使用MongoDB这样的文档存储引擎更加高效便利。下面是采用不同编程语言和技术栈配合不同的数据存储类型的技术多样式示意图。




微服务架构提倡在技术多样性的场景中，选择最适合的技术栈。微服务通过使用标准的API接口对外暴露服务，给尝试新技术提供了更加友好的架构支持。

然而，很多公司也推崇使用统一的编程语言和标准化的技术栈。统一技术栈的优势也是明显的，首先它会带来开发效率的提升；单一技术栈的维护成本相对较低；新加入的开发人员也能够尽快适应统一的编程语言和架构风格；项目的风险相对比多技术栈有更好的可控性。

即便如此，我们说微服务架构还是向着异构化、技术多样性的趋势在发展，因为只有保持技术的多样性，才能保证技术生态的生命力。对于技术栈和技术选型来说，架构师需要一个Trade-off（权衡利弊）的过程。

### 1.2.4 去中心化

大型企业在集成异构系统和完成进程之间的通信时，一种传统的架构模式就是使用ESB消息总线技术，它可以完成信息路由、业务规则编排、协议转化等功能。虽然，ESB架构改变了传统软件的架构模式，消除了不同应用之间的技术差异，协调了不同应用服务的协作运行方式，实现了服务之间的集成和整合，但是，ESB架构倾向于使用集中式的架构管理模式，它本质上是一种中心化的架构。我们将这种企业服务总线或服务编配系统的方案称为“智能管道和哑终端”模式，它会导致业务逻辑的中心化和哑服务问题。

“哑终端”（Dumb Endpoint）会导致ESB消息总线过度复杂，这种中央式的架构模式存在天然的技术与业务耦合问题。业务编排和业务消息转化能力与业务功能全部集中在单一逻辑控制单元中，它并没有做很好的业务封装，而是将业务逻辑的复杂性全部传递到了消息总线中。同时，随着服务规模的扩大，中心化架构的可扩展性会成为一个极大的障碍。业务中的职责边界不清和ESB中心化的问题还会暴露性能问题，成为系统的瓶颈。

微服务架构摒弃了ESB的设计理念，在微服务架构中，服务使用智能端点（Smart Endpoint）模式。智能端点强调所有的业务逻辑应该自包含在业务内部的处理逻辑单元中，它可以确保在服务限界内服务的内聚性，而服务之间的通信应该尽量轻量化和简单化。同时，微服务使用哑管道（Dumb Pipe）通信机制，将业务无侵入的公共组件抽象出来，封装在通用的消息基础设施中（API网关、消息中间件等）。

我们把微服务架构这样的设计理念称为“去中心化”。微服务架构倾向于服务之间订立标准化的服务契约，目标是通过明确清晰的服务边界和服务契约机制让服务可以各自独立迭代和演进。

为了最大化微服务能带来的自治性，我们需要给拥有服务的团队委派决策和控制权。去中心化管治的最高境界就是亚马逊所宣扬的“构建并运行它”的理念，团队对构建的软件的方方面面负责。

### 1.2.5 自动化运维

微服务架构的采用也引入了很多复杂性，关键问题是我们不得不管理大量的服务。微服务增大了运维负担；有更多的东西需要部署，有更多的地方需要监控，错误自然也成倍增加。而解决这些问题的一个关键方法就是拥抱“自动化文化”。前期花费一定的成本，构建支持微服务的工具是很有意义的，比如，自动化测试保证开发迭代中的代码质量，使用自动化发布工具将微服务部署到各个环境，使用配置文件来明确不同环境间的差异，创建自定义镜像来加快部署，创建全自动化的不可变服务器。

自动化一直是软件系统运维的最佳实践，也是微服务架构强调的重要特性。云技术使得底层基础设施及运行在之上的组件自动化变得非常简单。尽管前期投入通常会更高，但从中长期来看，无论是人力运维成本，还是在系统的弹性和性能方面，几乎总能获得更多的回报。自动化可以比人更快地修复、扩展和部署系统。

自动化贯穿软件生命周期的整个过程，在持续集成领域，我们经常使用Jenkins等工具自动构建、测试和部署微服务软件包。微服务不仅应该自动化部署，还应该努力实现金丝雀测试和回滚等过程的自动化。

除非系统负载几乎从未发生变化，否则应该根据负载的增加对微服务进行自动扩展，并根据负载的持续下降进行自动收缩。通过扩展可以确保服务仍然可用，通过按比例收缩可以降低成本。

随着微服务及云原生架构的大规模推广和使用，部署和运维的复杂度会逐渐从业务端下沉到以Kubernetes为代表的基础设施PaaS平台，利用云和微服务架构，我们可以更加快速地部署和交付我们的服务，围绕快速交付的基础设施建设是微服务架构规模化发展的首要任务。

### 1.2.6 快速演进

软件的固有特性随着时间的推移会变得越来越难以改变，软件的组成部分会因为各种各样的问题变得脆弱，难以操作。软件和现实世界一样，当人类的需求和环境供给达到平衡时，世界是美好

的，然而当这种平衡因为虫害或者气候变化被打破时，人类需要向生态中引入变化，重新建立平衡。对于软件系统，同样存在这种动态的平衡，我们需要提早对系统进行规划和设计。

尽管很多人喜欢在一个理想的环境下来讨论架构，然而，对于庞大复杂的单体架构，很多因素可能促使我们将混乱引入系统工程：业务需求的快速变化、工作任务的优先级冲突、有限的人力资源和预算、软件工程师水平的参差不齐、缺乏规范的开发流程和部署方式。另外，如果是遗留系统，还会存在代码版本混乱、冗余的代码逻辑等技术债务。这种技术包袱总会带来灾难性的后果。通常，业务人员往往不想放弃还在工作的系统，而开发人员，面对单体系统的腐化，只能通过不断地堆叠功能完成任务。不停地做加法，架构成为塞满各种功能和修复逻辑的庞然大物，最终产生破窗效应。而这种架构上的缺陷也将持续加重我们的技术债务，业务人员要么忍受这样糟糕的设计、不断地妥协，要么丢弃已有系统，推倒重来，这样的做法对于资源有限的团队和公司来说，显然是难以承受的。

微服务架构强调在项目早期将软件分成若干个阶段及不同的模块，从时间、业务维度及架构维度上做水平和垂直化的分解。微服务构建的首要任务就是理解业务的问题域，好的架构师会充分考虑业务领域的内聚性，降低业务之间的耦合，寻求两者的平衡，并将架构的可扩展性作为重要的设计考量因素。微服务架构的一个特征就是面向架构演进，微服务架构的目标正是通过业务领域的边界划分、通过服务的隔离来分解问题，逐个击破，因此微服务架构天然具备了可演进性。

## 1.3 架构设计哲学

如果说软件开发的本质是不断挖掘问题领域中隐藏的错综复杂性，那么架构解决的问题就是如何管理这些复杂性。而在软件领域，最为复杂的软件实体莫过于软件操作系统。从数以千计的工程师参与开发的UNIX操作系统到Linux开源系统的成功，越来越多的人开始关注和思考UNIX技术背后隐藏的设计哲学。

UNIX设计哲学概括为一句话就是“小而专注”。可以说，微服务架构理念和UNIX设计哲学一脉相承，微服务将UNIX设计哲学中的核心准则通过概念的抽象，描述成了更加通用的架构风格和设

计原则。下面，让我们跟随经典重新认识在AT&T公司诞生的UNIX操作系统和它背后的设计哲学，所谓“温故而知新”，这些经典思想能加深我们对微服务架构的认识和理解。

### 1.3.1 小即是美

在软件的设计开发过程中，软件系统的规模很容易膨胀，工程师喜欢将纷繁复杂的功能全部堆积在一个程序中，这样的好处是代码唾手可得。然而，根据二八法则，实际运行中的代码其实往往只占到我们代码量的20%。所以，切忌将大而全的工程作为我们的目标，相反，我们应该将功能设计成为实用且简洁的小程序 注 。

在我的职业生涯中，经历过很多设计复杂、规模庞大的单体系统。有的是基于C++编程语言的后端大型系统，有的是使用J2EE–XML编程风格、交织着庞大功能模块的巨石型应用，这些项目通常由10人以上的研发团队负责，动辄百人/月的开发计划。这种大型单体系统在上线后，时常发生系统内存溢出、系统宕机等问题，让开发人员心惊胆战。往往一个小问题会影响整个系统的正常运行，排查解决过程需要检查整个工程的代码逻辑，开发人员疲于解决生产上线后的各种问题和Bug修复。

事实证明，我们需要将庞大、复杂的系统分解成小程序。正如UNIX系统中强调的KISS（Keep it Simple and Stupid）原则，可以说整个UNIX系统都是由数目众多的小程序组合完成各种复杂的操作系统功能的。每个小程序只完成其中一小部分功能。表面上，这些小程序都很低效，但是正是通过像搭积木一样将不同功能的小程序通过变换顺序和组合的方式，完成了各种意想不到的丰富而强大的功能。

小程序之美体现在响应变化上。通过小程序可以将这种改变控制在足够小的范围，保证不会给整个系统带来巨大的影响。

小程序之美体现在工程结构的简化和完备上。当你的程序充满了个性化的反射调用和程序员独特的编程思维痕迹，那么应该反省你的代码是不是已经脱离了小程序应有的代码简洁和易于理解。

小程序之美体现在性能优势上。小程序对外部的依赖少，从而可以快速启动，基于资源收敛式的反应式编程模型可以提升性能并减少资源浪费。

小程序之美体现在团队合作和独立演进上。清晰的边界划分是团队协作的有效手段，而体现在微服务架构上就是服务提供者和消费者可以预先订立契约，可以根据契约独立、并行开发各个服务，这样就实现了服务之间的解耦，使其功能能够独立进化。

### 1.3.2 做好一件事

小程序应该只做好一件事，应该保持对一件事的专注力。在UNIX设计哲学中，解决一个问题并将问题解决到完美，比同时解决多个问题更为重要。然而在我们的工程中，随着项目的进展，我

们很难将庞大的代码库进行清晰的模块划分，更糟糕的是我们很多时候并不知道这些模块之间的服务界限。

在UNIX操作系统中，我们可以发现很多命令的强大之处正是只有单一的功能，并将这件事干好，也就是所谓的“Do one thing，do it well”。而且UNIX首创的管道可以把这些命令任意地组合，以完成一个更为强大的功能。这些哲学到今天都在深深地影响着整个计算机产业，下面我们罗列一些经典的UNIX命令程序，如下表所示。

| 命令 | 功能描述 |
| --- | --- |
| awk ‘{print $1}’ <file> | 显示文件的第一列 |
| cmp <file1> <file2> \| <command> | 比较两个文件 |
| find / -type f -name <file> -print | 在整个文件系统中查找一个文件 |
| grep ‘[a-z][0-9]’ <file> | 以某个正则表达式查找文件行 |
| mount -p | 查看挂载的文件卷 |
| sed -e ‘s/hello/john/g’ <file> | 把某文件中的hello替换成john |

微服务的目标是独立地完成一件事并做到最好。微服务可以根据业务的边界来确定应用的行为，这样它不仅可以很容易地确定某个功能逻辑对应的代码，而且由于该服务专注在某个边界之内，因此可以更好地避免由于代码库过大衍生出的复杂性问题。

### 1.3.3 快速建立原型

只做好一件事的小程序可以让我们的项目轻装前行，我们不必再担心系统会成为一个庞然大物而无法控制。对于单体巨石应用，需要制定周密的计划去编写设计文档，以及为聚合不同模块之间功能而准备脚本和代码，相比之下，自治的小程序更加清楚自己的业务职责和业务范围。开发人员可以对小程序快速建立原型，缩短服务的交付周期，迅速构建可以供用户使用的程序和服务，并从结果中得到反馈，向最终的目标前进。

在UNIX设计哲学中，软件发生变化是不可避免的，这种变化来源于沟通的失败、需求理解的差异、知识经验的限制等，所以软件工程相比任何其他工程都更加容易返工，需要软件从业者不断试错、总结、验证，并根据期望重新建立共识。尽快建立原型是一个重要的步骤，有了具体的原型，可以降低项目的风险，可以给客户展示和进行可视化跟踪。往往这个过程是伴随着系统的迭代和演进的。

回到微服务，我们说微服务架构在建立原型上有天然的优势，微服务架构的很多特性能让我们快速落地和逐步地独立完善和迭代项目。

- 微服务架构强调细粒度的服务，在服务规模上尽量精简和务实。
- 微服务架构只做好一件事，没有过多的其他因素干扰，我们可以将注意力集中在完成这件事情上，尽量排除干扰因素。
- 微服务架构提供轻量级的通信集成方式，有利于集成测试和验证结果。

● 微服务架构建立在已有的技术基础之上，能够更加快速地构建、发布和体验应用。

### 1.3.4 软件的复利效应

在软件工程实践中，在项目的启动或者早期阶段，经常会面临技术方案的选择问题。一些软件工程师会陷入自我保护的状态，认为别人的方案存在缺陷，自己如果重新做会比现有方案做得更好，但其实可能是因为他不了解“每个软件可能都是在某种约束条件下工作的，而且适用于某些特定的场景”。我们把这种现象称为“NIH（Not Invented Here）综合征 [注] ”。NIH综合征的结果就是重新发明轮子，患有NIH综合征的人相信内部开发更安全、更高效、速度更快、维护成本更低，然而，其实他们并不一定能够使用创新的方案来解决实际的问题。

目前，软件正朝着规模化和标准化的方向发展，标准驱动下的软件开发和集成方式，要求你的工作能够集成到标准中，而不是另起炉灶。NIH综合征的危险之处就是，你的软件、你的服务无法与标准对接，你的系统可能成为一个孤岛系统。例如，在你提供了私有协议的RPC方式暴露服务的情况下，你的服务只能生存在自己的闭环体系中，而且基于标准的HTTP接口API的调用方式很难与你的服务集成。

项目的生命周期受很多因素的影响，预算、人员成本、推广等都会给项目的持续发展带来风险，NIH综合征会给项目的持续发展带来额外的成本问题，因为你缺少相关人员、资料、生态的支持，而采用标准化的技术可以从互联网中得更多的技术、社区和人员的支持。

下面让我们看看UNIX的实践，所谓“前人种树、后人乘凉”。查阅UNIX坎坷的发展历程，我们就会发现，UNIX操作系统是在数千名工程师的辛勤努力的基础上发展起来的。UNIX中复杂的逻辑都是由小的程序累积而成的，聪明的程序员总是可以借用前人写的优秀代码实现自己的功能，这样才能更快、更好地扩大软件的影响力和威力，放大自己的工作成果。

在微服务领域，我们已经看到非常多优秀的微服务技术框架，例如本书后续会详细介绍的Spring Boot微服务脚手架。它是在Spring生态的基础上发展起来（Spring Boot也是软件复利的成果）的开发框架。在Java世界，Spring Boot带给广大开发者的是更加简化的工作流程，以及更高的开发效率。这也是Spring出色的地方，弥补了Java在简化工作上的空白，让开发者开发者避免了大量的重复工作。Spring Cloud则降低了建立分布式系统的复杂度，Spring的忠实热爱者可以充分享受软件复利效应带来的福利。

### 1.3.5 可移植性优先

可移植性在软件工程中的重要性无论怎么强调都不过分，因为这个哲学正是UNIX操作系统能够成为“常青树”的秘诀。在UNIX环境下，Shell脚本具备更好的可移植性，Shell脚本通常由多个UNIX命令组成，Shell可执行文件间接由UNIX命令解释器解释执行。如果你对效率要求不高，可以尽量使用Shell脚本执行，Shell脚本语法简单、使用方便、运行之前不需要编译、具备很强的文本处理能力，开发效率也比较高。

但是Shell归根结底还是一种弱类型语言，没有严格的数据类型检查。Shell的缺点是I/O性能不高，同时因为是解释性语言，对于值计算类型的问题也没有很好的支持。如果你想让程序具有更好的可移植性，可以选择具备跨平台能力的编程语言。Java便是一种跨平台语言，基于JVM虚拟机平台，Java制定了字节码执行引擎规范，可以满足程序的可移植性要求。这也是Java语言发展这么多年依然具有强大生命力的重要原因。

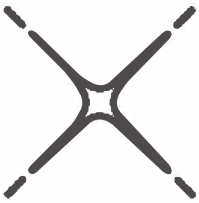
在微服务时代，容器的隔离性和可移植性可以说为软件开发带来了革命性的颠覆，Docker技术通过采用LXC 注 虚拟化手段，利用Linux系统的Namespace和Cgroup技术确保了应用程序与资源的隔离。Docker通过和各大厂商联合发起OCI（开发容器标准），规范了应用运行时的容器镜像标准。镜像的打包、构建、部署、命名过程都按照统一规范进行，进而标准化了底层运行时支撑环境，这样你就可以在统一的容器环境下灵活地交付、部署和移植代码。通过采用Docker容器技术，我们将不需要再关注操作系统的特殊性和差异性，可以更关注应用程序本身，底层多余的环境因素可以通过容器提供的虚拟环境来屏蔽。也就是说，我们的微服务具备了更好的可移植性，真正做到了“一次构建，随处运行”。

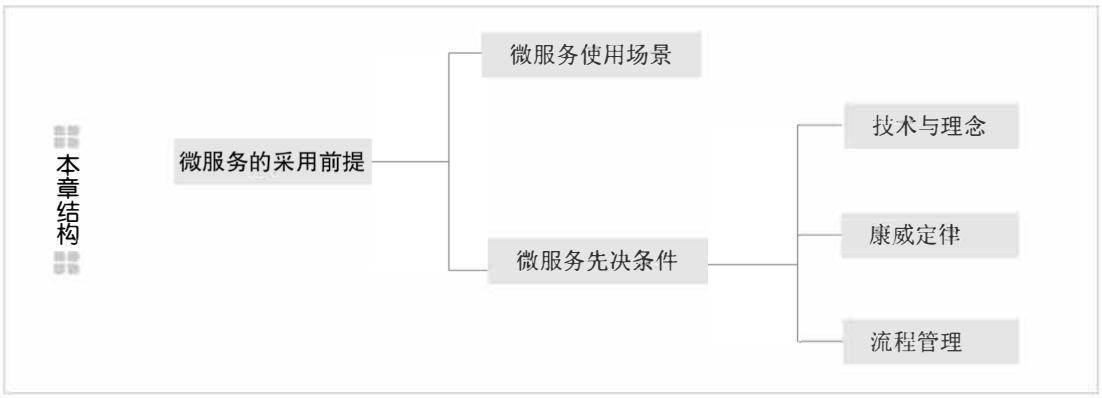
## 1.4 小结

本章我们了解什么是微服务、微服务架构产生和发展的背景、云计算时代下微服务与云原生架构的关系、微服务架构的主要特性，及微服务架构与UNIX设计哲学的内在联系。然而微服务架构并不适用于所有场景，下一章将带大家了解微服务的采用前提。

第2章

# 微服务的采用前提






有些公司对微服务架构存在着“盲目崇拜”，大型企业在微服务架构上的成功更加增加了人们转型微服务架构的信心，但是大公司的技术实践未必适用于你所在的公司，我们有必要重新审视微服务架构，了解微服务的采用前提。

微服务会带来分布式下应用的开发、测试、运维等多方面的问题。拆分后的细粒度服务从原来进程内部方法调用转变为分布式跨网络调用，由此带来了更多的服务治理难题。从技术的角度看，需要根据项目的实际情况和业务场景等工程约束条件，决定是否采用微服务。

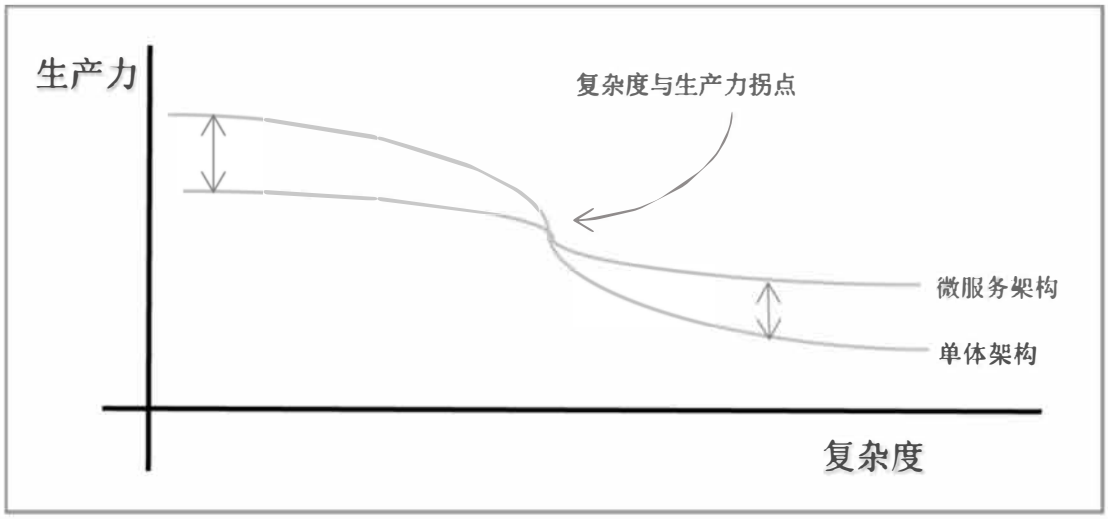
通过康威定律，我们知道组织结构与一个公司的技术架构存在着紧密的关联性。公司的数字化转型，不仅在技术上需要与时俱进，还要和企业的组织方式、管理流程的调整同步进行。公司能否围绕技术、组织、流程这些工程要素进行迭代升级，是能否转型到微服务架构的前提。

## 2.1 微服务使用场景

### 2.1.1 项目复杂度

微服务架构主要解决的问题是通过对庞大的单体架构进行服务拆分，使得服务更加容易理解和控制。当你的业务应用逻辑本身复杂度不是很高时，微服务架构的威力是很难发挥的，所谓“杀鸡焉用宰牛刀”。

下图揭示了微服务架构与单体架构的复杂度与生产力的关系。




在项目复杂度较小时，采用单体架构的生产力更高；复杂度到了一定规模时，单体架构的生产力开始急剧下降，这时对其进行微服务化的拆分才是合算的。复杂度和生产力虽然存在拐点，但并没有量化复杂度的拐点，或者说没有明确系统或代码库的规模达到具体多大时才更加适合开始进行微服务化的拆分。

团队决定构建一个微服务项目时，需要根据项目的复杂度判断是否有必要采用微服务架构，因为微服务本身也会给系统带来非业务功能的复杂性，所以在转型微服务时要慎重考虑。大多数采用微服务架构的场景还是单体架构的改造，这是业务开发人员在巨石型应用带给团队极大困扰之后的一个自然选择。这个项目阶段往往就是所谓的项目复杂度与生产力的“拐点”时期，对单体架构进行合理的服务拆分是实施微服务架构的一大前提，在后续的章节中，我们会进一步详细讲解服务拆分的依据和策略。

Martin Fowler曾经说过：“……除非你的系统复杂到难以管理的程度，否则不要考虑采用微服务……”微服务架构需要额外的开销，比如服务设计、服务通信、服务管理和系统资源。采用微服务是有代价的，如果一个应用程序无法充分利用微服务的优势，那么采用微服务反而得不偿失。所以，在我们采用微服务之前，首先需要做一个很好的权衡，需要明白使用微服务的驱动力是否充足；业务是否复杂到需要借助微服务拆分来解决问题，以快速响应变化。

### 2.1.2 团队规模

微服务架构非常适合大型项目团队。对于大型项目，《人月神话》中的“人月互换理论”已经证明是失败的，这种方式往往忽略了沟通的成本。正如著名的“两个比萨原则”：如果两个比萨不足以喂饱一个项目团队，那么这个团队可能就显得太大了。

然而，对于小型的项目团队或者只有少数开发人员维护的系统，其实是没有必要使用微服务架构的。单体架构的简单性有助于简化团队成员的工作，并且可以将系统的复杂度控制在一定范围内。使用微服务架构会增加组织的沟通成本，模块之间的跨网络交互也会给开发和运维带来额外的成本，将本来简单的事情复杂化。

所以，在实施微服务架构前，首先请关注你的团队规模。在人力资源有限的情况下，其实并不推荐使用微服务架构，因为如果你的公司没有像亚马逊、Netflix这样的技术储备和平台储备，微服务架构反而会增加系统的复杂度，进而带来一系列问题，让你怀念单体架构带给你的简单性。

### 2.1.3 变更频率

2016年，Gartner发布了关于应用变化速度的报告“Pace-Layered Application Strategy”，该报告以变化速度为标准将业务应用分为三层。

- SOI（Systems of Innovation，敏态业务）：比如互联网业务，需求变更快，要求快速迭代、快速交付。
- SOR（Systems of Record，稳态业务）：比如传统业务，变更周期长、变化频率低、变化成本高、变化风险高。
- SOD（Systems of Differentiation，中台业务）：解决前后端的开发速度匹配失衡问题，中台为前台与后台之间添加的一组“变速齿轮”。

对于新兴行业中的敏态业务，它们需要有更加动态化和更快的响应，服务往往需要更快的交付速度和更加频繁的版本发布。微服务架构的一个显著优势，就是对变化的快速响应。微服务架构相比单体架构更加独立，所以在快速开发、持续集成、持续交付上有更明显的优势。另外，微服务架构强调使用标准、轻量级的通信协议进行服务交互，契合中台作为集成前、后台资源的业务形态。所以从业务形态和服务的交付频率上说，微服务比较适合在SOI和SOD的场景中应用。

传统的稳态业务，比如大型的电信项目、银行项目，本身周期比较长，变化频率相对较低，我们不需要迁移到微服务架构，而是最好保持它目前的运行状态。

我们可以借助工具来检查哪些项目的变更频率比较快，可以利用GitLab这样的工具统计项目代码的提交次数和构建频率，以便决定哪些项目需要进行微服化改造。还可以使用源代码管理系统来查看代码的活跃度。以Git存储库为例，可以使用常用的Linux工具，通过几个命令行选项来运行Git日志。例如，我们可以使用命令生成提交次数最多的“前十个代码文件列表”。此外，也可以利用全新的“代码鉴定”工具（比如CodeScene）深入了解项目，CodeScene可以识别代码中的热点，帮助我们找出代码活跃区域。

### 2.1.4 项目类型

从来没有一种架构模式适用于所有业务形态。目前企业内部还有很多对性能有严苛要求的系统运行在单体架构之上。虽然单体架构存在诸多缺点，但是单体架构内的各个组件之间的交互更加简单，内部的方法调用更加高效。对I/O性能要求比较高的实时计算系统或者嵌入式系统，往往关注服务的延迟和服务的吞吐性能。

相比单体架构，在完成相同功能的情况下，微服务架构可能需要经过更多的网络交互调用，而远程调用势必增加系统在网络上的I/O延迟和花费在网络上的数据传输处理时间。所以，对性能敏感的项目，微服务架构对系统造成的性能影响显然是无法被接受的。

另外，微服务架构更适合处理OLTP（On-Line Transaction Processing，联机事务处理）类型的项目，而不擅长处理OLAP（On-Line Analytical Processing，联机数据处理）类型的项目。大数据处理中使用比较广泛的架构是Lambda和Kappa等，以及Hadoop技术。

### 2.1.5 遗留系统迁移

在生产环境中，我们有大量的遗留系统。我们可以通过逐渐改进和演变的方式实现遗留系统向微服务架构的迁移。

遗留系统是否要迁移到微服务架构是一个战略问题，你需要甄别遗留系统是否适合采用微服务架构。

在战术层面，你需要制定详细的改造迁移策略，如功能剥离、灰度替换、数据解耦、数据同步、滚动发布等。如果你正在面对一个遗留系统的微服务改造项目，那么无论它的原始设计多么随意，无论它现在变得多么糟糕，在把它重构成微服务之前，都要认真仔细地思考一下，它正处在软件生命周期的什么阶段？它是一个任务关键型系统吗（比如包含了一个不可替代的遗留数据库）？你需要多长时间来替换整个系统？更新或者替换过程需要一个长期详尽的计划吗？

微服务架构在更新或替换遗留系统方面扮演着重要的角色，没有策略指引的迁移很可能会造成灾难性的后果。在后续的微服务构建相关章节中，我们会讲解常见的微服务改造模式。

## 2.2 技术与理念

微服务的概念还在快速发展的过程中，它不仅给我们提供了分布式下细粒度服务设计、构建、交付、运维的方法，同时整合了过去几年行业的先进技术和最佳实践。

### 2.2.1 面向服务

大部分企业选择微服务架构是业务驱动的。对于基于传统J2EE技术栈的Web项目而言，早期单体架构就是所谓的“一个War包打天下”，将应用程序的所有功能都打包成一个独立的War包，部署在Tomcat的指定目录下就可以顺利运行。然而，软件项目是一个不断迭代和变化的过程，业务模块的增加、功能的扩展、人员的更迭、需求的变动最终都需要修改代码来实现。于是代码跟随版本的不断升级而逐渐膨胀变得难以维护。单体架构的灵活性、可扩展性、可运维性都明显下降，开发人员效率降低、系统稳定性变差、局部小问题导致“牵一发而动全身”。

在这种情况下，单体架构为了保证程序内部的高内聚、低耦合，引入了分层的架构模式。分层架构在某种程度上解决了不同类型代码的逻辑耦合问题，模块之间有了更加清晰的职责划分，降低了单体架构的整体复杂度。而分层架构的问题是没有聚焦当前业务逻辑，以技术为导向的架构形态很难做到服务的复用，业务模块无法独立部署和演进。

微服务的理念与SOA服务架构是一脉相承的，微服务架构同样强调面向服务，将一个大的“问题空间”通过领域建模拆解分为实体之间的关系和行为，使用限界上下文（Bounded Context）将实现细节封装起来，让服务可以独立伸缩，每个服务都有明确的边界。

在面向业务构建微服务时，我们不应该把主要的关注点放在是采用Tomcat还是Jetty应用容器上，也不应该放在Spring Cloud、Docker、RPC这些技术概念或框架上。微服务架构首先要考虑的是解决业务的问题。在开始微服务架构转型之前，请先理解业务，洞察业务边界、职责划分，让团队专注于实现某个特定的应用服务。微服务需要根据一定的软件设计原则来实现面向服务的架构模式，本书在微服务构建章节会深入讲解领域驱动设计如何帮助我们对系统进行合理的服务划分，如何拆解、聚合以实现服务的开发和复用。另外，面向服务的系统的服务边界划分需要我们格外注意，因为错误的服务领域划分将会使服务陷入大量的远程调用和分布式事务中，在这种情况下，微服务给整个系统带来的不是便利而是麻烦。

### 2.2.2 底座技术

从效率的角度出发，微服务架构需要配套的技术栈和技术底座 注 支撑。目前，很多一线互联网公司已经成功基于底座技术实现了微服务架构的落地和实践。

技术选型是落地微服务架构的关键环节，公司在落地微服务架构的过程中，不仅仅要关注技术本身，更重要的是结合自己公司的技术现状和人员技术背景，根据已有的技术栈来进行微服务架构的落地和实践。

从技术选型的角度出发，我们需要根据一定的优先级来考虑：需求满足度、社区活跃度、技术掌控能力。

- “需求满足度”是最重要的因素，因为技术和架构最根本的动机还是满足业务的需求，如果一项先进的技术没有满足用户的需求，则这项技术将失去发展的动力。
- “社区的活跃度”是非常重要的参考，因为社区的活跃度代表了这项技术被广大开发者接受的程度和这项技术的广泛度和生命力。对于技术薄弱的公司，最好采用在一线互联网公司落地并且在社区内拥有良好口碑的开源产品。因为这些技术已经在大企业中经过了生产环境的验证，并且有良好的社区生态，可以得到更多的技术支持。当然，有实力的公司会选择通过自研的方式落地微服务架构，而对于小规模团队，还是建议采用社区的技术框架落地微服务架构，一方面可以不用从头开始，另一方面，也会降低公司的整体学习成本。
- 同时，需要考虑公司人员的技术掌控能力。例如，如果一个框架使用C语言实现了微服务架构，虽然在性能上有优势，但整个团队没有懂C语言的开发者，那么就需要重新考虑是否有其他的替代技术方案。

基于Spring社区的影响力，目前可以认为Spring Boot是构建Java微服务架构的事实标准；另外，Dubbo是阿里多年的生产级分布式微服务实践的技术结晶，Dubbo本质上是一套基于Java的RPC框架，服务治理能力非常强，在国内技术社区中具有很大的影响力。还有Apache ServiceComb等国内外知名微服务框架，这里就不再赘述。

### 2.2.3 架构技术

单体架构被拆分为微服务后，需要解决众多服务的治理及复杂度管控问题。微服务存在领域模型建设、服务边界划分、服务与服务之间的依赖、服务交互集成、独立数据管理等问题，针对这些，我们需要优化我们的架构设计理念和设计方法。

**领域驱动设计**

Eric Evans在《领域驱动设计》一书中对不同公司的业务应用程序中遇到的复杂问题进行了总结，帮助我们在现实世界中进行建模。他为领域驱动设计提出了大量的最佳实践和经验技巧：

- 领域驱动设计方法有利于软件开发团队与业务部门或领域专家密切合作，使开发人员与业务人员达成共识。

- 技术人员和业务专家应该首先进行领域建模，找到有界的上下文和相关的核心域，以及普遍存在的语言、子域、上下文映射，用于简化软件项目的复杂度，使得设计思路能够更加清晰、设计过程更加规范。

对于微服务架构而言，它和领域驱动设计同样关注业务。领域驱动设计理念聚焦于领域建模、实体、边界划分、界限上下文。领域驱动设计非常适合从业务上去划分微服务的边界，定义服务对外暴露的接口。每一个微服务都应该是一个可以独立开发、部署、运行的自治主体。可以说，领域驱动设计是指导微服务架构设计、解耦业务、服务拆分、服务构建的关键原则。

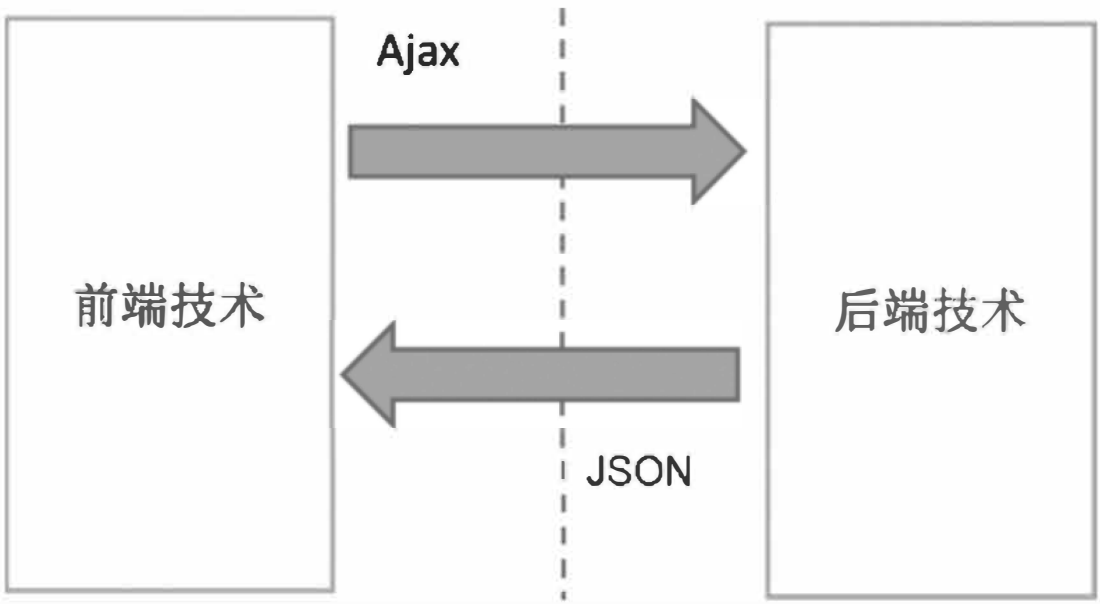
**前后端分离技术**

微服务倡导专业分工，每个组件都专注于各自的业务领域。而大部分软件，尤其是面向企业领域的系统基本都是由前端和后端服务组成的。将前后端分离作为切入点，我们可以轻松地开启微服务化改造之旅。下面是微服务架构进行前后端职责划分的主要规则：

- 技能分离，前后端可以使用不同的特定语言或框架来实现最佳的微服务实践。
- 职责分离，前端主要负责和用户的交互逻辑，后端主要负责业务逻辑和资源的管理。
- 部署分离，前端和后端可以做到独立发布，不存在发布过程的耦合，前端和后端可以根据约定的API进行版本迭代和独立演进。

前后端分离架构有利于将微服务在技术层面上解耦，后端微服务可以专注于实现对后端服务和资源的管理，而不用再关注与用户交互的逻辑验证等问题。

前后端分离有利于微服务中的各组件的演进和组合，微服务架构细粒度的服务可以让不同前端共享不同后端，通过“搭积木”的方式实现服务的复用和独立的演进，如下图所示。




**事件驱动技术**

在单体架构下，系统的复杂性在于如何做好模块之间的解耦，因为模块依然存在于同一个进程中，实现进程内的并发处理和多线程模型的管理是单体架构的主要工作。所以单体架构的瓶颈往往

是CPU，不适合网络I/O密集型的计算处理场景。而在微服务架构下，因为细粒度的服务之间的交互主要通过分布式网络进行，而事件驱动架构为微服务提供了更多的跨网络集成优势。

- 基于事件的体系结构是异步的，不存在网络阻塞。在提供服务之前，我们必须考虑网络的局限性。选择REST同步调用方式存在服务调用依赖问题，会产生级联消息雪崩效应，而选择事件机制我们不用担心同步调用的问题。
- 基于消息队列的异步消息处理机制。相比请求/响应模式的服务集成方式，采用Broker代理的服务集成方式可实现服务之间的解耦，同时可以提高服务的性能、可靠性、可扩展性。然而，事件驱动框架也会在消息一致性、消息的监控和传输上给系统带来额外的技术复杂性。
- “协同”优先“编排”原则。事件编排机制会由“中心大脑”来领导并驱动整个流程，这个大脑就像交响乐队中的指挥；事件协同机制会预先说明清楚系统中各个部件的职责，而把具体怎么实现留给各个部件。总的来说，事件编排机制的缺点是明显的，中心控制点承担了太多职责，它会成为网状结构的中心枢纽。而事件协同机制不仅可以降低系统的耦合性，还可以让我们以更加灵活的方式修改现有系统。
- CQRS（Command and Query Responsibly Separate，命令查询职责分离），是介于脚本驱动和领取驱动之间的一种服务建模与数据交互模式，是事件驱动领域中被广泛使用的一个概念，通过CQRS可以解决数据读写交叉问题，并能有效降低业务逻辑的复杂性。

### 2.2.4 服务监控与治理

在单体架构时代，因为所有服务都集中在少数几个系统中，系统之间的相互调用关系相对简单，在出现故障时，可以将问题根源锁定在有限的系统范围内并定位问题。而在微服务架构下，众多微服务实例之间有频繁的分布式跨网络协作、相互远程调用，这时如果没有一整套服务治理方法，帮助我们保证SLA（Service-Level Agreement，服务等级协议）、增强服务治理水平、提升微服务的治理与运维效率，那么微服务的转型之路将举步维艰。

技术团队关注的焦点往往是架构的实现和业务建模，容易忽略微服务架构带来的一系列负面影响，通过微服务监控与治理可以全方位地掌控当前服务的运行状态和资源利用情况，可以说，服务治理与监控既是微服务架构在平台层面的核心工作，也是微服务应用“长治久安”的前提。通过微服务治理能力的提升，可以提供对业务应用的快速响应能力、保证业务的健康稳定及持续演进；在技术上，可以帮助微服务的开发和运维人员实时地掌握微服务的运行状态，以及进行问题定位和故障恢复。

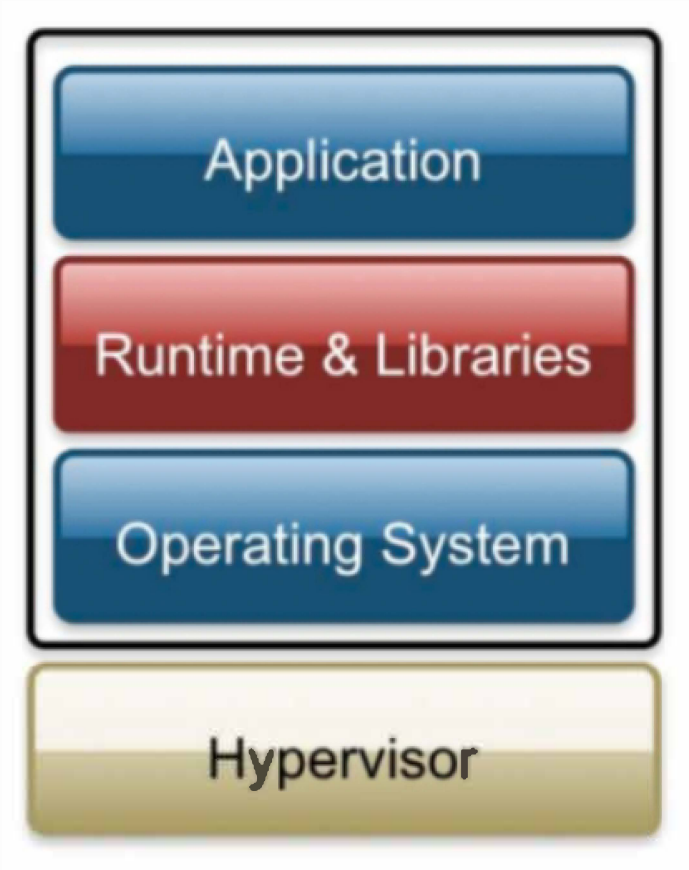
在服务治理的技术选型上，Spring Cloud提供了服务治理的一站式解决方案。同时，微服务框架结合众多技术组件可以提供Metrics指标监控、日志监控、调用链等信息监控、健康检查和告警通知等功能。Metrics监控主要依赖于时间序列数据库，目前较成熟的产品有Prometheus和OpenTSDB；我们可以采用ELK（Elasticsearch、Logstash、Kibana三个技术框架缩写）技术栈实现日志的归集、存储、搜索和可视化报表查看等；可以采用Spring Cloud Sleuth日志收集工具包，结合Zipkin和HTrace，作为Spring Cloud的一种分布式追踪解决方案。

### 2.2.5 容器和自动化技术

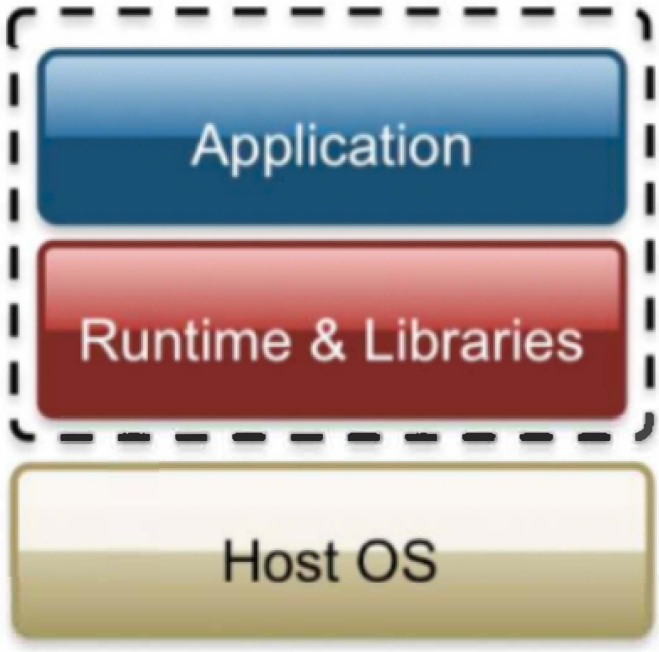
容器和自动化技术可以说是微服务规模化发展需要解决的首要问题。基于容器的部署和交付无论在软件开发领域，还是运维相关领域，都带来了巨大的技术颠覆。通过自动化交付部署可以将开发与运维环节打通。交付物的标准化使得我们可以“标准化”地交付应用及它所依赖的运行环境。Docker容器一次构建、多次交付的特性可以使微服务具备了更好的可移植性、可复用性。

很多人把容器化和微服务架构混为一谈，认为自己的服务部署到了容器后，系统就是一个微服务架构了。其实二者存在本质上的差异。可以说，一个巨石型单体架构即使部署到了容器上，依然无法改变它架构上的“拙劣”。

目前容器技术仍然是微服务架构最佳的运行时环境和平台，也很好地支撑了微服务架构去中心化、细粒度、容错、伸缩的特性。如下图所示，是虚拟机与容器不同的隔离机制。




虚拟机




容器

使用传统虚拟机的应用存在如下运维弊端：

- 部署非常慢、成本非常高、资源浪费。
- 难以迁移和扩展。
- 服务与硬件厂商提供的特性绑定。

使用容器对微服务架构进行自动化运维的优势：

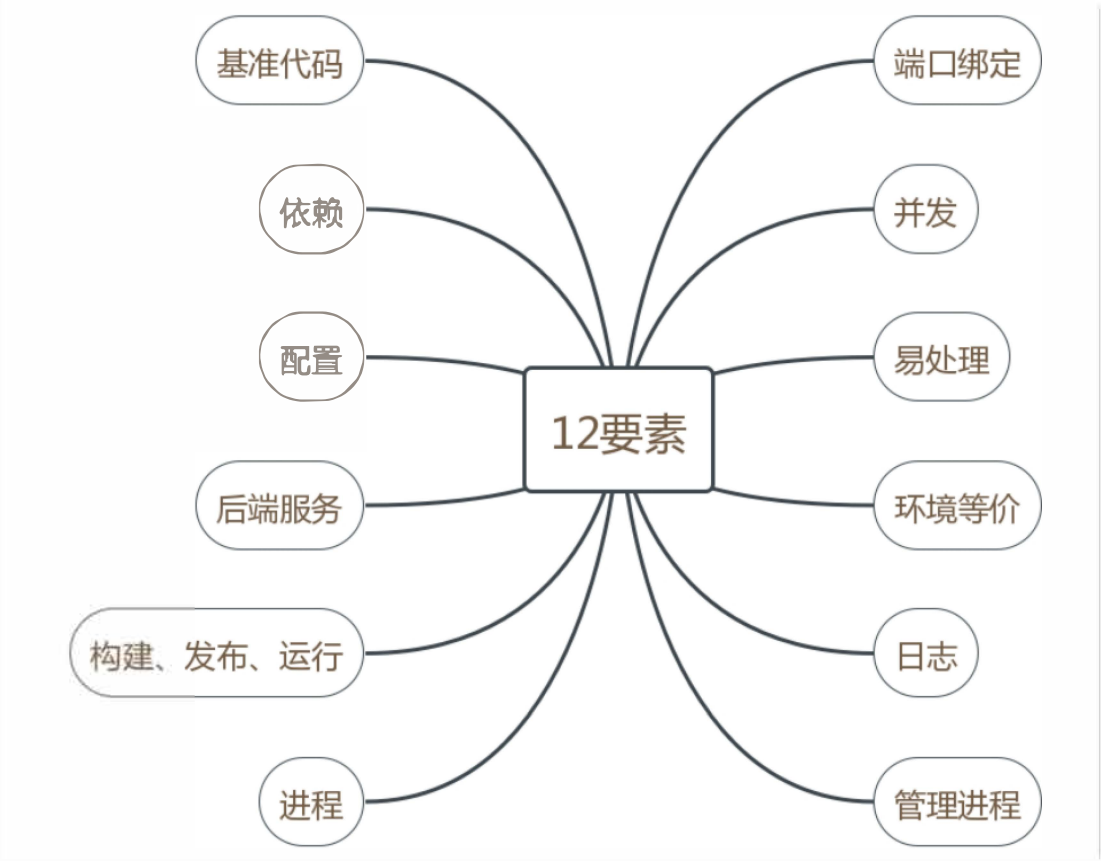
- 容器的资源开销更小，Docker本身共享操作系统内核，容器可以节省更多物理机资源。
- 在开发和运维之间搭建了一座桥梁，是实现DevOps的最佳方案。
- 容器可以对软件和其依赖的标准镜像格式打包，保证应用在不同运行环境下的统一交付，解决环境差异问题和大规模的交付部署问题。
- 容器本质上相当于一个进程，每个容器都可以看作一个不同的微服务进程，因此可以独立升级，与底层共享操作系统，性能更加优良，系统负载更低，在同等硬件资源条件下可以运行更多的应用实例，更充分地利用系统资源。
- 在没有适合的工具和自动化运维的情况下，使用微服务架构会导致灾难。目前基于容器技术和Kubernetes技术的平台已经成为微服务交付和管理的最佳平台。

### 2.2.6 云原生12要素

Gartner定期发布的技术成熟度曲线可以帮助我们方便地评估技术的成熟度，同时帮助我们制定战略，即采用怎样的技术工具。Gartner最新发布的技术影响力和普及模式与成熟度分析说明，微服务架构已经在中国全面进入成熟落地阶段，从全球来讲，很多知名IT企业和公司，像亚马逊、Netflix、阿里巴巴，已经采用微服务架构重塑了自己的核心业务系统。当前，很多公司为了节省成

本、减少对基础硬件设施的投入，纷纷选择采用公有云或者私有云的方式管理公司的应用服务和基础设施。

云原生12要素（如下图所示）是基于云的架构设计和开发模式需要具备的一套全新的理念，也是云原生应用开发的最佳实践原则。




- 要素1：基准代码

  基准代码和应用之间总是保持一一对应的关系：一旦有多个基准代码，就不能称为一个应用，而是一个分布式系统。分布式系统中的每一个组件都是一个应用，每一个应用都可以分别使用云原生12要素进行开发。

- 要素2：依赖

  显式声明依赖关系，应用程序不会隐式依赖系统级的类库。要通过依赖清单，确切地声明所有依赖项。此外，在运行过程中通过依赖隔离工具来确保程序不会调用系统中存在但清单中未声明的依赖项。这一做法会统一应用到生产和开发环境。像Java语言使用的Maven打包软件或者Cradle都可以显式声明依赖。

- 要素3：配置

  通常应用的配置在不同部署环境（预发布、生产环境、开发环境等）中会有很大差异。Factor推荐将应用的配置存储于环境变量中。我们可以非常方便地在不同的部署环境中修改环境变量，却不动一行代码。

- 要素4：后端服务

这里的后端服务指的是应用运行所依赖的各种服务，例如数据库、消息代理、缓存系统等，对于云原生应用来说，往往还会有日志收集服务、对象存储服务及各种通过API访问的服务，可以把这些服务作为外部的、通过网络调用的资源。

- 要素5：构建、发布和运行

  构建、发布和运行三个阶段需要有严格的界限，部署工具应该明确每个发布版本的唯一发布ID对应的构建版本。

- 要素6：进程

  应用进程的内部不要保存状态信息，以一个或多个无状态的进程运行应用，任何状态信息都应该被保存在数据库、缓存系统等外部服务中。应用实例之间的数据共享也要通过数据库和缓存系统等外部服务进行，直接的数据共享不但违反无状态原则，还引入了串行化的单点，这会给应用的横向扩展带来障碍。

- 要素7：端口绑定

  通过端口绑定提供服务，应用完全自我加载而不依赖于任何网络服务器就可以创建一个面向网络的服务。互联网应用通过端口绑定来提供服务，并监听发送至该端口的请求。

- 要素8：并发

  通过进程模型进行扩展，任何计算机程序一旦启动，就会生成一个或多个进程。互联网应用采用多种进程运行方式，而不是线程方式，进程是“一等公民”。

- 要素9：易处理

  要求应用可以瞬间（理想情况下是数秒或者更短时间）启动或停止，因为快速启动和优雅终止可最大化程序的健壮性，这将有利于应用快速进行横向扩展和变更，或者进行故障后的重新部署，这两者都是程序健壮性的体现。

- 要素10：环境等价

  环境等价的浅层次含义是要求在开发环境和线上环境中使用相同的软件栈，并尽可能为这些软件栈使用相同的配置，以避免程序对不同环境的依赖。

- 要素11：日志

  应用程序应该将其产生的事件以每个事件一行的格式按时间顺序输出，这点毋庸置疑，但是本要素想说的其实是：应用程序不要自行管理日志文件。

- 要素12：管理进程

  一次性管理进程应该和正常的常驻进程使用同样的环境。这些管理进程和任何其他的进程一样使用相同的代码和配置，基于某个发布版本运行。后台管理代码应该随其他应用程序代码一起发布，从而避免同步问题。屏蔽环境差异和集群下的多实例问题。

总之，云原生12要素很适合快速部署应用程序，因为它们不需要对将要部署的环境做任何假定。它允许底层云平台使用简单而一致的机制，轻松实现自动化，快速配置新环境并部署应用，同时实现规模优化、应用快速迭代演进。

## 2.3 康威定律

在设计系统时，组织所交付的方案结构将不可避免地与其沟通结构一致。

——梅尔文·康威

### 2.3.1 协作问题

根据康威定律，技术架构与组织的职责划分相关，而职责划分从根本上确立了组织的沟通协作方式，这种协作方式最终决定了技术架构的形态。如果你的组织本身是比较松散的协作方式，往往你的架构会变得离散；而如果你的组织是紧耦合的，架构往往也会慢慢向紧耦合的方式发展。

当技术人员将单体应用拆分成多个细粒度服务的时候，就产生了服务之间的协调沟通问题，而这种问题都是建立在组织结构之上的。组织为了解决协作的问题，就会设置沟通管理方式，当我们对组织划分不合理的时候，解决问题需要跨部门沟通时，就会产生巨大的沟通成本。而康威定律提醒我们，在实施微服务架构时，不仅仅需要关注软件的架构和设计，更要关注团队之间的委派、分配和协作的方式。如果组织划分不合理将会使我们的工作失焦，导致系统架构与业务沟通协作模式不匹配。

微服务架构倾向于组织架构围绕业务领域边界进行划分，这种协作方式是比较理想的。通过构建与业务架构一致的团队组织，实现每个独立的团队都可以为自己负责的业务和技术负责。这种组织结构的好处在于：需要升级或者变更服务时，各角色成员可以在团队内部进行高效沟通，有利于达成共识和高效协作。只有外部服务之间的接口需要变更时才需要跨部门沟通，如果前期在服务之间的交互上定义了良好的接口，则接口变更的概率并不大，即使接口模式有变更，也可以通过一定的设计模式和规范来协作解决。

组织划分的最终目标是实现每个团队组织都有清晰的边界和职责。通过组织的划分，形成高内聚、可复用的业务形态，聚焦的技术架构有利于功能模块的沉淀积累和迭代演进。把不属于自己职责的业务或者技术，尽量从自己的服务中剔除，通过组织协作的方式实现低耦合，避免职责重复带来的工作重复和效能损失。

康威定律从本质上说明了组织结构对系统架构的影响，强调系统设计与组织结构的不一致导致的危险和协作问题。当我们的系统规模开始扩大时，这种协作问题将会对微服务系统的业务响应、需求变更、质量保证等方面产生更加深刻的影响。

### 2.3.2 沟通效率问题

《人月神话》中说：一项工作在估算和安排中使用工作单位“人月”来计算成本。这里需要说明的是“成本”虽然可以根据开发产品的人数和时间确定，但是工作的进度却不一定，因为使用“人月”来评估一项工作的规模是存在欺骗性的，人数和时间在一定程度上是无法互换的，一个原因就是忽略了最重要的因素：人员之间的沟通成本。

软件开发本质上是一项系统工作，需要团队成员密切配合才能完成，是错综复杂关系下的一种密集脑力活动实践。沟通、交流的工作量非常大，它很快会消耗任务分解所节省下来的个人时间。而添加更多的人手，实际上增加了沟通的成本，降低了个人的工作效率。

沟通效率往往会受到组织结构的影响。按照康威定律，我们的组织结构应该与我们的应用架构保持一致，最早践行微服务架构的亚马逊和Netflix可以说是执行这条原则的典范，通过小团队的构建提升了沟通的效率。

例如，在Netflix，存在很多小而独立的团队，这些独立的团队创建的服务也是彼此独立的，这最小化了沟通成本，带来了效率的提升。这样，软件架构也可以得到快速迭代和演进。

人与人的沟通是非常复杂的，一个人的沟通精力是有限的，所以当问题太复杂，需要很多人协调解决的时候，我们需要拆分组织来提高沟通效率。团队的组织方式从某种程度上决定了他们设计的系统架构，而高效的沟通不仅有利于为业务人员提供即时的反馈，也能够以最小的代价达成共识，实现降本增效。

### 2.3.3 组织的演进

在实施微服务初期，管理者一般将大部分精力集中在满足业务和服务的功能性诉求上。很多小型团队在不需要太多精力投入的情况下，通过开源软件搭建注册中心、API网关、容器等微服务基础设施，就可以完成基本的微服务管理功能。

然而，随着业务的快速增长和微服务规模的扩大，以及系统复杂性的上升，维护这样一套基础设施会给业务团队带来额外的压力，在关注业务自身功能的同时，管理者还要将精力消耗在非功能需求和以技术为导向的微服务支撑平台上。这个时候，管理者往往低估了微服务架构的复杂性，以及伴随微服务发展的高可用、高并发、可扩展性等技术诉求。管理者需要重新审视组织结构是否适应公司整体技术架构的发展。微服务架构从某种程度上说是一个CTO工程。

我们需要业务团队更加聚焦于自己的核心业务能力，正如Supercell公司，它是一家典型的以小团队模式进行游戏开发的公司，一般来说两个员工或者5个员工，最多不超过7个员工组成独立的

开发团队，称为Cell（细胞），这也是公司名字Supercell（超级细胞）的由来。该公司可以通过这种小团队，快速试错，来检验游戏是否被用户接受和受欢迎程度，实现了小步快跑。

然而Supercell的小团队背后是有一个大的平台组织来做支撑的。我们说微服务架构同样需要具备平台化的可复用和支撑能力，平台化的思维和支撑能力不仅仅意味着需要建设以技术为导向的微服务架构、自动化发布平台、容器基础设施，还需要组织结构能够根据业务形态进行持续的演进。微服务架构如果想规模化、成体系化发展，甚至像亚马逊和Neftlix这样的公司一样对外输出技术能力，还需要公司做统一的战略规划，需要有以技术为主的平台团队，以及以业务为导向的中台团队。微服务架构的体系化仅从技术维度进行是难以奏效的，最终技术架构一定会受到组织结构的影响，组织结构的演进对企业的竞争力和非线性的增长至关重要。

## 2.4 流程管理

微服务从技术架构出发，使应用系统具备快速响应、灵活部署、敏捷交付、持续演进的特性成为可能，而规模化的微服务交付如果没有完整的软件工程和流程管理体系、自动化的流程交互运维工具，很难持续发展。

### 2.4.1 敏捷方法论

最早的软件开发是没有标准的，软件质量好坏完全取决于软件工程师的能力和素养。伴随着软件行业的发展，软件的整个开发流程需要一套方法论来规范，CMMI（Capability Maturity Model Integration，即能力成熟度模型集成）这种基于瀑布模式的标准软件开发流程解决了软件开发的标准化问题。在CMMI体系下，软件开发逐步向工程化迈进，但是也缺少了动态性和灵活性，人们开始反思传统方法的弊端，进而提出了敏捷方法。

2001年2月，由Martin Fowler等17位软件开发专家起草的敏捷宣言发表，敏捷联盟成立。我们可以看到一个熟悉的名字：Martin Fowler，微服务架构的概念最早也是他提出来的。

下面我们了解一下敏捷宣言中敏捷软件开发的四个核心价值观。

- 个体和互动高于流程和工具：这里强调了人与人直接沟通的重要性和高效性，所以有了站立会议。但不要理解为完全不需要流程和工具，只是侧重人与人的沟通。
- 工作的软件高于详尽的文档：这里强调把更多的时间放在开发软件上。但并不是说文档就一点也不要写了，关键性的文档（项目整体流程描述及架构图等）还是要有的。
- 客户合作高于合同谈判：这里要区分公司属性，外包公司团队和企业自有项目团队肯定会采取不同的策略。这句话应该是讲给企业自有项目团队的，为了企业合作双方的利益，这个观点是正确的。但并不是说毫无底线地向客户妥协，而是提倡密切合作，以便提前发现问题，双方都可以减少损失，实现共赢。
- 响应变化高于遵循计划：这里道出了软件开发不可回避的问题，就是需求变更。站在企业整体利益的角度，需求变更是很正常的。随着市场变化、时间推移，之前的需求如果不做修改可能真的就影响了产品的落地效果。但这也不是说需求可以任意地变来变去，需要有既懂业务又懂技术的人来负责评估。

Scrum（迭代式增量软件开发过程）是实现敏捷开发的具体方式之一，是一种具体实施方案和流程，也称为过程管理框架。Scrum的主要原则是缩短软件的交付周期，通过将过程分解为短期的工作循环，每一个循环为一个Sprint（冲刺），在每个Sprint中，由项目团队确定目标，Scrum团队由不同类型的人员组成，包括开发、测试、业务人员、美工等，每个团队的目标由项目管理者在计划会议上确定，团队可以自行确定实现Sprint目标的方法和途径，可以自行管理日常工作和计划的执行。

微服务架构与敏捷研发流程一脉相承。微服务是将一个完整的系统分割成若干微小的、具备独立性的功能单元，每个功能单元是可以具备一个实际意义的小功能集。各个功能单元之间尽量是解耦或松耦合的，可以实现独立开发而不依赖其他功能单元。而敏捷保证微服务架构能够更好地适应需求的变化，保持团队的高效沟通，敏捷利用小型工作增量、频繁迭代与原型设计等手段，可以使我们摆脱大规模单体软件开发的风险。

微服务架构更多地从技术的角度提升开发和运维的效率，而敏捷方法论贯穿了软件工程的整个流程，它重视流程、沟通、协作。可以说，敏捷在管理流程上是对微服务架构落地的有益补充和保障。

### 2.4.2 DevOps转型

随着软件开发敏捷化的推进，DevOps正在成为软件交付的最佳模式。DevOps是一种研发模式和一种方法论，从2009年开始逐渐流行起来。它不是框架或技术的东西，是更好的优化开发（DEV）、测试（QA）、运维（OPS）的流程，使开发运维一体化，通过高度自动化的工具与流程来使得软件构建、测试、发布更加快捷、频繁和可靠。DevOps的主要目的在于提高产品的交付能力与效率，要践行DevOps首先要改变的就是管理理念。

DevOps其实包含了三个部分：开发、测试和运维。从单体架构转型到微服务架构后，我们可能面临开发、测试和运维诸多挑战，微服务架构需要借助DevOps方法论进行持续开发和持续集成

的原因如下：

- 服务拆分成为细粒度的服务后，服务之间需要持续集成，才能完成整体的功能迭代，需要自动化测试、自动化交付。相比单体架构，微服务架构有更高频率的软件部署集成、发布、交付诉求。
- 大多数微服务架构使用容器和云平台，需要在流程上支持快速的发布流程，支持规模化的微服务交付场景。
- 要理解微服务之间的复杂交互，需要优秀的诊断与监控工具，而软件系统诊断与监控不仅仅是运维人员的职责，开发人员也需要深入参与到软件的交付和后期的系统运维和运营中，提升微服务之间交互的可观测性。

软件团队DevOps转型需要从下面几个方面展开，这其中不仅包含技术的转型，也包含组织、流程等关键要素的转型。

- 技能方面：团队需要不断引入优秀的技术和好的设计来增强敏捷能力；保持良好的软件架构风格；集成工具链，搭建自动化软件集成、交付、发布平台。
- 组织与流程方面：DevOps需要组建一个自治的团队，使团队向全栈开发方向成长。另外需要融合不同的职能人员，比如开发人员和运维人员工作在同一个部门，向同一个领导汇报。要求开发人员和业务人员在整个项目开发期间必须天天在一起工作。需要建立激励体制和反省体制，团队每隔一定的时间在如何才能更有效地工作方面进行反省，然后对自己的行为进行调整。开发人员需要遵守相应的原则来保证进度。软件进度的唯一标准就是“能使用的软件”，优先要做的事是通过持续地交付有价值的软件来使客户满意。为了达到这个目的，在开发过程中要最小化可执行产品，持续集成/持续交付、持续部署（CI/CD），测试驱动开发（TDD 注 ）。交付的时间间隔越短越好，使用户经常看到软件效果。其中测试驱动开发可以优化代码设计，提高代码的可测试性，建立和代码同步增长的自动化测试用例，根据迭代积累的经验和需求变化情况对计划进行不断的调整和细化。
- 文化建设方面：实行激励机制，通过建立激励制度，强化和奖励那些引导组织朝着目标前进的行为。举行每日站立会议，进行高效的会议、记录问题并跟踪问题的解决过程；进行可视化管理，可视化管理可以让所有团队成员直观地获得当前项目的进度信息，及时暴露问题；保证团队理解一致并且适应变化，团队需要一定的敏捷理念，认清客户是逐步发现真正的需求的。

总之，DevOps贯穿软件工程的整个生命周期，DevOps不只包含CI/CD方法论，除了技术和流程，它还包含企业管理文化。

### 2.4.3 自动化管理工具

在DevOps实践中，自动化管理工具的使用是非常重要的，下面让我们看看一些常用的代表性工具。

● IT基础设施自动化

云服务：使用公有云（如阿里云、AWS等）服务，不需要买硬件服务器、租用机柜。私有云服务可以基于Kubernetes进行扩展构建。

● 代码管理

Git：存储代码，管理代码的版本。

● 配置管理

Chef：它是一个非常有用的DevOps工具，用于管理配置文件。使用此工具，DevOps团队可以避免跨10000台服务器进行配置文件的更改，相反，只需要在一个地方进行更改，然后自动反映在其他服务器上。

● 自动部署

Jenkins：这个工具可以实行自动部署，有助于持续集成和测试。

● 交付镜像

Docker：不仅可以交付软件代码，还能将代码依赖的工程运行环境一同打包进行交付。

● 日志管理

ELK：这个工具可以收集、存储和分析所有日志。

● 性能管理

App Dynamic：提供实时性能监控功能。此工具收集的数据有助于开发人员在出现问题时进行调试。

● 监控

Nagios：当基础设施和相关服务宕机时，确保人们得到通知很重要，Nagios就是这样一个工具，它可以帮助DevOps团队发现和纠正问题。

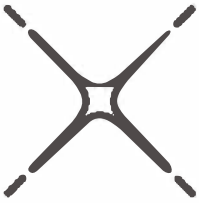
## 2.5 小结

软件世界没有“银弹”，不存在理想的软件模型提供全面的解决方案。每一个公司或者企业都需要结合自身的情况和场景来选择是否采用微服务架构。如果你正在基于微服务架构构建或者改造你

的系统，那么请注意你使用的技术理念和软件方法论与微服务架构是否存在冲突。总之，在软件工程中，除了技术因素，组织结构、研发流程等都会对微服务架构能否成功落地产生重要影响。

第3章

# 微服务构建



微服务构建本质上是软件构建过程中长期演进积累的一系列理念、架构原则、工具和最佳实践。

领域驱动设计的软件思想体系和方法论可以用于指导微服务建模、微服务划分、微服务架构设计等相关工作，它可以促使技术人员与领域专家达成共识，构建领域边界合理、具备明确界限上下文、关注点分离、独立自治的微服务。

## 3.1 领域驱动设计

### 3.1.1 领域驱动设计概述

领域驱动设计（Domain Driven Design）概念的兴起可以追溯到1986年，《人月神话》的作者Brooks提出软件的本质复杂性（Essential Complexity）存在于复杂的业务领域中，技术仅仅是辅助工具，它解决的问题是帮助业务领域从现实问题映射转换成软件实现。

领域驱动设计在战略设计层面，从业务视角出发使技术人员专注于问题域，从领域专家那里获得领域见解，通过模块划分建立领域服务边界，通过界限上下文明确服务的职责。

领域驱动设计在战术设计层面，从技术的视角出发，提炼有效的业务模型，实施领域建模、架构设计完成软件的落地。领域驱动设计通过隔离业务与技术的复杂性，成为程式化、标准化的软件架构设计范式。

**软件复杂度的来源**

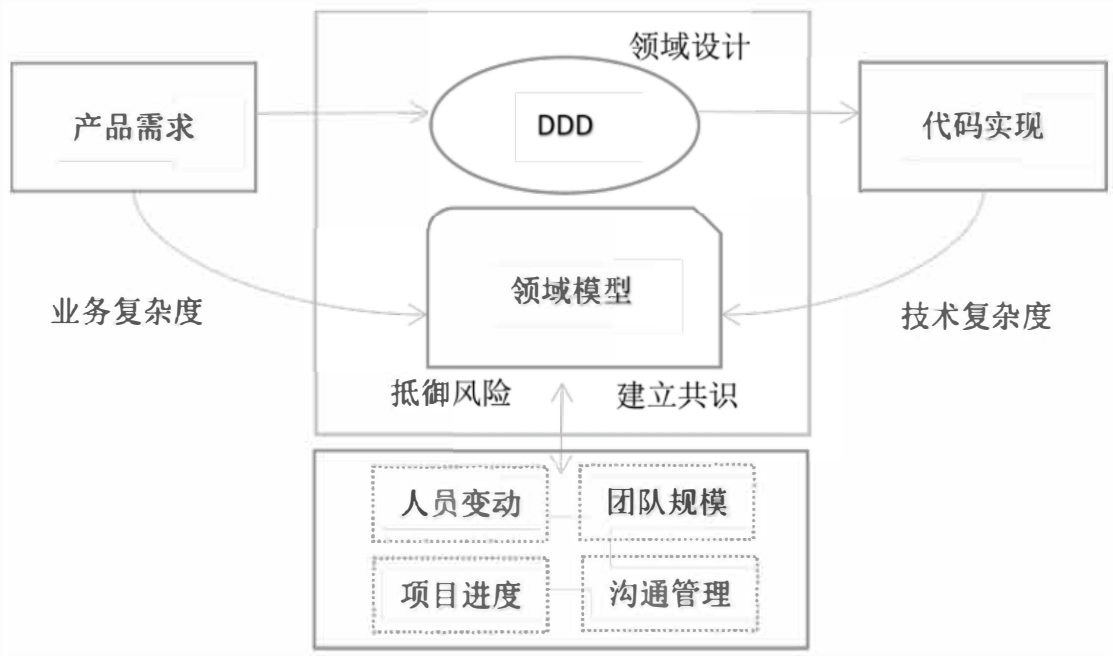
- 业务的复杂性：业务的复杂性体现在业务流程不清晰、业务参与人员多、业务与技术耦合等方面。在业务的早期阶段，为了快速满足功能需求容易形成面条式的代码风格，这样的代码风格会导致软件模块膨胀、开发效率降低、功能扩展步伐放缓、业务模型与代码脱节等。
- 技术的复杂性：技术的复杂性来源于对项目的质量属性需求 注 ，诸如系统的性能、客户体验、服务高可用性等。为解决服务的响应延迟、吞吐、安全等问题，我们会引入缓存、消息队列、第三方模块组件，而这些技术的整合给系统引入了额外的复杂性和技术挑战。

**领域驱动解决之道**

解决这种软件构建中面临的复杂性问题，我们需要从领域开始着手，与业务专家一起获得领域见解，促使软件利益干系方在领域内建立通用语言。技术人员通过建模的手段提炼出事物的本质，以便更好地指导应用系统的构建和规划。

领域驱动设计中包含了大量成熟的理论、概念、模式和架构，它包含一套解决复杂领域模型的软件架构方法，思想是围绕业务模型来连接和实现核心业务概念。

领域驱动设计可以让业务和技术的变化产生的不可预知因素互相分离，将人员变动、团队规模、协作沟通等外界因素变化对产品和项目的影响封装在一个可控的容器和框架下，从而解决软件面临的复杂性问题，如下图所示。




**事务脚本模式与领域建模模式**

- 事务脚本模式：事物脚本模式常见于单体应用中，它将所有逻辑全部组织在一个单一过程方法中，从数据库的调用到不同业务逻辑、策略的执行全部集成在一个大的方法块中。它的好处是简单、容易实现，它的缺点是没有自己的状态，也无法扩展，容易将服务组件与数据存储模型之间的刚性依赖引入业务逻辑中。
- 领域建模模式：领域建模模式将业务逻辑转移到了领域对象（Domain Object）中，每个领域对象完成属于自己的业务行为。同时数据存储层的逻辑也变得相对简单，数据库不再参与领域模型的业务逻辑，而是回归数据“持久化”的本质。

使用领域模式可以提升系统的内聚性和可重用性，通过不同类之间的协同完成所有功能。另外，多态的模式也让扩展新的策略更加方便，业务语义更加通用、显性化。领域建模过程遵循“SOLID”原则并实现业务域的逻辑解决方案。

**说明：SOLID原则**

1.Single Responsibility Principle：单一职责原则

2.Open Closed Principle：开闭原则

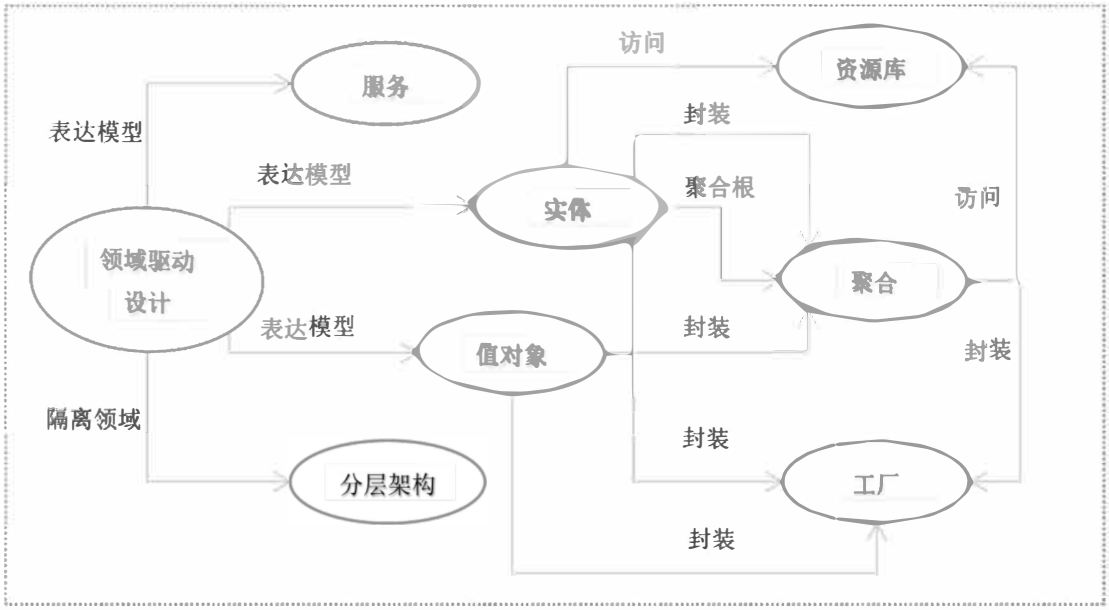
3.Liskov Substitution Principle：里氏替换原则

4.Interface Segregation Principle：接口隔离原则

5.Dependence Inversion Principle：依赖倒置原则

**领域驱动设计核心要素**

如下图所示是领域驱动设计的核心要素，包含领域驱动设计中的通用模型术语和重要的战术模式。这些模式不仅可以捕获和传递领域中的概念、关系及逻辑，也能帮助我们管理业务的复杂性并确保领域模型的行为清晰明确。




- 领域：相对于软件系统来说就是系统要解决的现实问题。
- 子域：对于领域进行不同维度切分的相对内聚的子系统单元。
- 分层架构：通过分层架构将业务域和技术逻辑域隔离。
- 服务：服务通常是领域对象的调用方，用来协调领域对象完成指定业务逻辑职责。
- 实体：实体与面向对象中的概念类似，它是领域模型的基本元素，在领域模型中，实体应该具有唯一的标识符。
- 值对象：值对象是没有唯一标识符的实体。值对象在领域模型中是可以被共享的，它们应该是不可变的，当有其他地方需要用到值对象时，可以将它的副本作为参数传递。
- 聚合：聚合使用边界将内部和外部的对象划分开来。每个聚合有一个根，这个根是一个实体作为外部可以访问的唯一对象。
- 资源库：是封装的所有获取对象引用所需的逻辑单元。
- 工厂：工厂用来封装对象创建所必需的信息，当聚合根建立时，所有聚合包含的对象将随之建立。

## 3.1.2 专注问题域

解决一个业务场景中的复杂问题从理解问题域开始，通过专注于问题域并理解复杂问题背后的实质，你才能设计有效的模型来应对业务的挑战。在项目初期，尽量避免沉溺于技术实现，而要把

焦点集中在问题领域，不要忘记技术服务业务的原则。

**理解问题域**

我们以一个金融场景下的“业务运营监控系统”为例进行分析。经过与运营管理专家和相关业务方的多轮需求探论，我们初步了解了用户的业务诉求和痛点。需要强调的是对于问题域的充分理解是我们的首要任务。

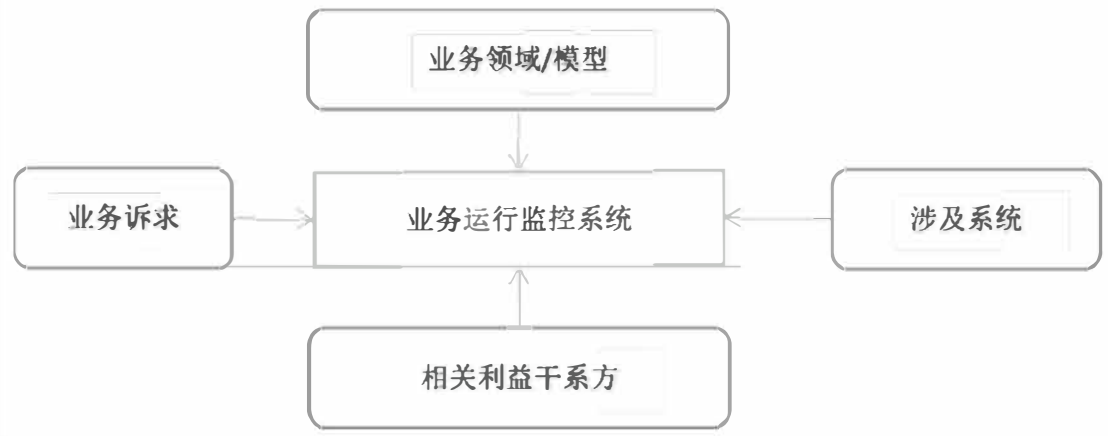
这里整理了一份需求文档，它详细地记录了问题域的具体范围和详细需求。这份文档不仅是业务与技术团队之间的一份沟通文档，也可以作为软件生命周期在需求分析阶段的一个清晰的、规范化的知识协作产物。

| 业务运营监控系统需求文档 | |
|---|---|
| 项目需求方 | 普惠金融·城市信贷运营管理部 |
| 需求背景：<br>城市信贷业务长期以来使用人肉运维的业务管理方式，运营团队无法通过数字化的方式管理线下团队的运营情况和系统运行状态。业务运营人员反馈进件存在积压，业务被阻断，无法快速地发现和定位问题系统，造成客户长时间等待甚至流失。<br>目前的解决方法是联合多个技术团队排查、定位问题，这种方式效率低下、沟通成本高。运营团队急需可视化的业务监控系统，监控当前业务的运营情况，定位业务积压的系统，方便运营部门的后续复盘、追责 | |
| 项目范围 | 项目针对普惠金融业务板块的城市信贷业务，目标是对公司贷前流程所涉及的子业务系统和线下门店实现透明的运营状态监控和管理 |
| 详细需求 | 1. 当进件出现积压时，系统可快速定位业务问题并自动通知相关系统的运维人员。<br>2. 系统可通过预先设置的预警邮箱，将具有业务含义的报警内容通知到指定运营人员。<br>3. 运营人员可统计当前进件主要被阻塞在什么业务流程阶段。<br>4. 运营人员可以统计进件在不同阶段的流量状态。<br>5. 运营人员可以统计当前不同门店的进件状态。<br>6. 运营人员可以对历史进件状态进行回溯，实现业务追责和系统故障定级。<br>7. 根据过滤条件查询门店运营状况 |
| 涉及系统 | 综合信贷系统、申请系统、合同系统、信审系统等 |

**提炼问题域**

理解复杂问题并从中识别、提炼出关键的业务模型，即提炼问题域是领域驱动设计的关键环节。团队可以通过头脑风暴的形式罗列出领域中的所有事件，整合之后形成最终的领域事件集合。

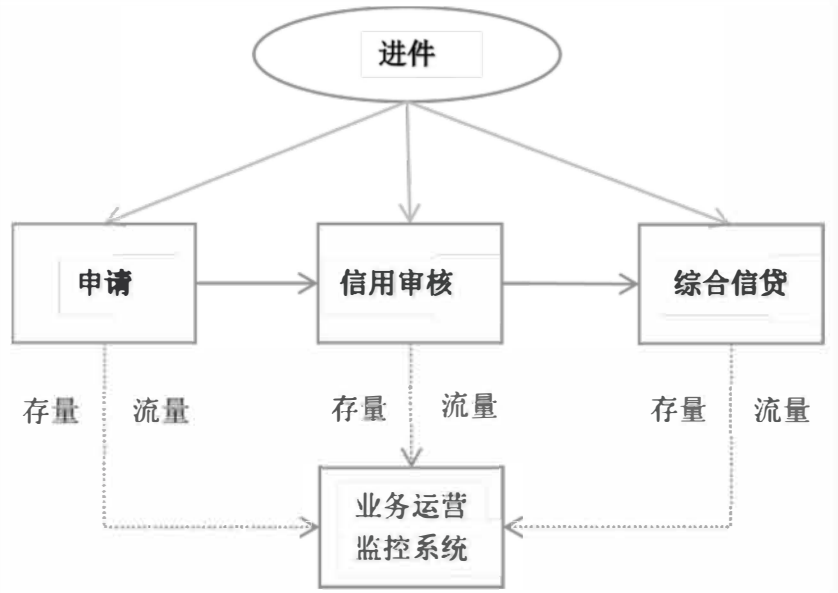
你需要在关键事件标记的范围里，参照不同利益干系方的业务诉求，组织领域事件和模型，同时，你需要整理出与项目关联的上下游系统，如下图所示。



通过挖掘隐藏在领域事件中的核心领域模型，我们可以找到从问题空间到方案空间的对应映射关系。针对上述业务监控系统案例，“进件存量 注 ”和“进件流量 注 ”的概念成为我们发现的重要领域模型。

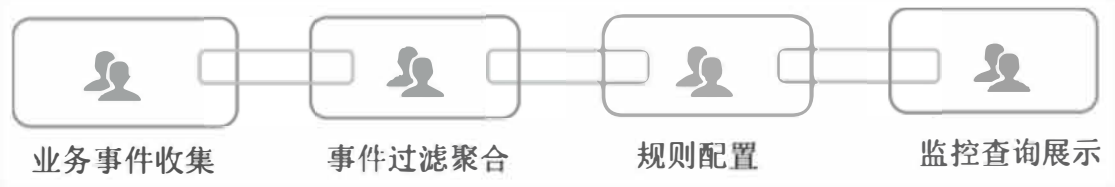
作为衡量业务系统运转状态的重要指标，业务的“存量”状态可以表示业务的积压情况，而业务的“流量”状态可以表示业务流转的变化情况。

如下图所示是我们总结的监控系统概要视图，其中实线表示的是城市信贷业务工作流中进件在不同系统的流向，而虚线表示的则是业务的存量、流量在业务监控系统的事件记录。

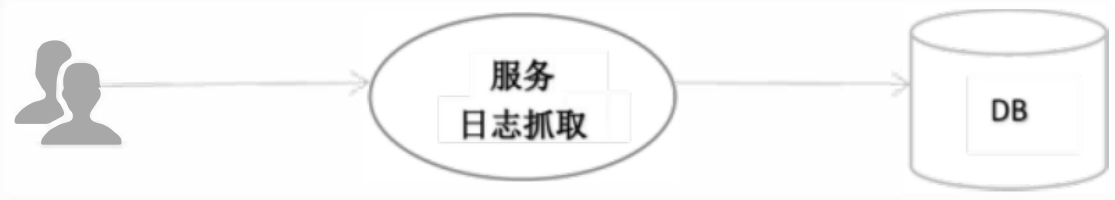



### 3.1.3 服务的拆分

完成问题域的理解和提炼后，我们需要对整体系统做进一步的服务拆分。下图是我们根据业务领域能力对“业务运营监控系统”进行拆分后的子领域服务及模块划分说明。




- 业务事件收集（如下图和表所示）

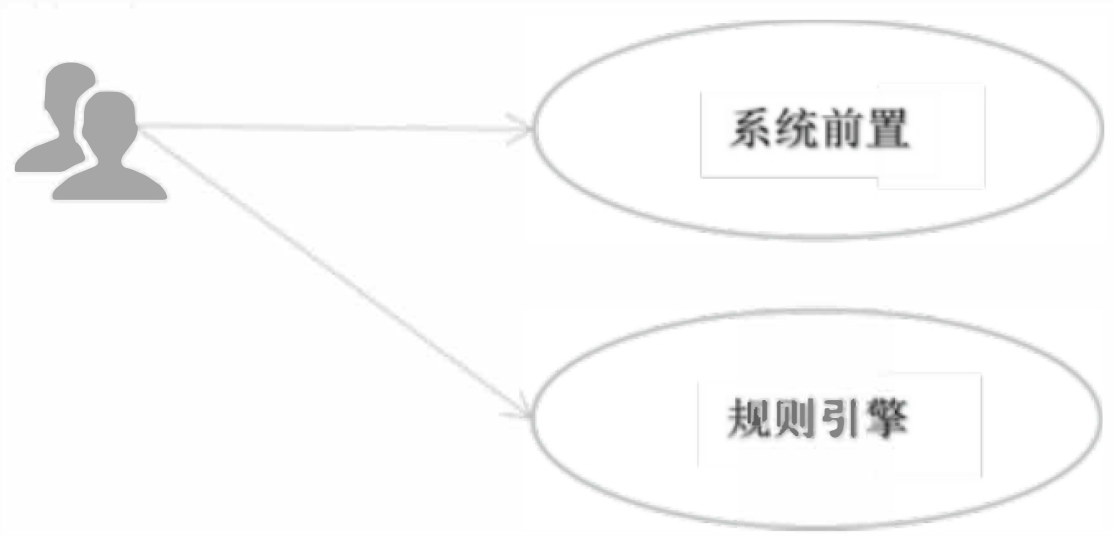



| 模块名称 | 简述 |
| --- | --- |
| 服务日志抓取 | 采集与贷前运营环节相关系统的业务事件日志，作为监控数据来源 |

- 事件过滤聚合（如下图和表所示）

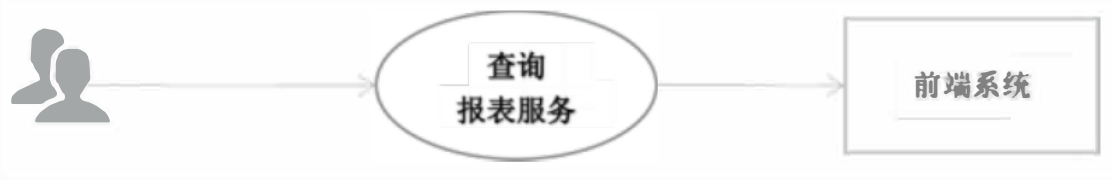
| 模块名称 | 简述 |
| --- | --- |
| 实时计算服务 | 实时计算服务利用外部分析系统对抓取日志进行过滤和分析，并生成明细监控数据 |
| 定时调度服务 | 调度服务主要做明细监控数据备份留存及数据的二次加工服务 |

- 规则配置（如下图和表所示）




| 模块名称 | 简述 |
| --- | --- |
| 系统前置 | 系统前置作为和客户端的接口，提供预警查询、规则配置等功能 |
| 规则引擎 | 规则引擎需要配置预警、过滤条件等配置信息 |

- 监控查询展示（如下图和表所示）




| 模块名称 | 简述 |
| --- | --- |
| 查询报表服务 | 根据业务方的需求可以查看当前业务存量、流量实时信息、报表统计信息 |

**为什么要做服务拆分**

- 降低系统的整体复杂性：根据业务领域进行合理的服务拆分是一个有效控制系统复杂性的方法。

- 提高效率：服务拆分后，代码模块相互隔离，并发的开发模式可以提升开发人员的效率。
- 团队人员各司其职：拆分的项目可分派给擅长相关方面技术的人员，让团队成员各司其职，降低工作的耦合度。
- 共享和自治：可以通过定义好的服务接口进行服务共享，同时拆分后的服务也更加自治。
- 解决依赖问题：通过服务拆分，可以清晰地了解哪些服务依赖会对业务造成影响，从而准备预案。

**服务拆分的依据**

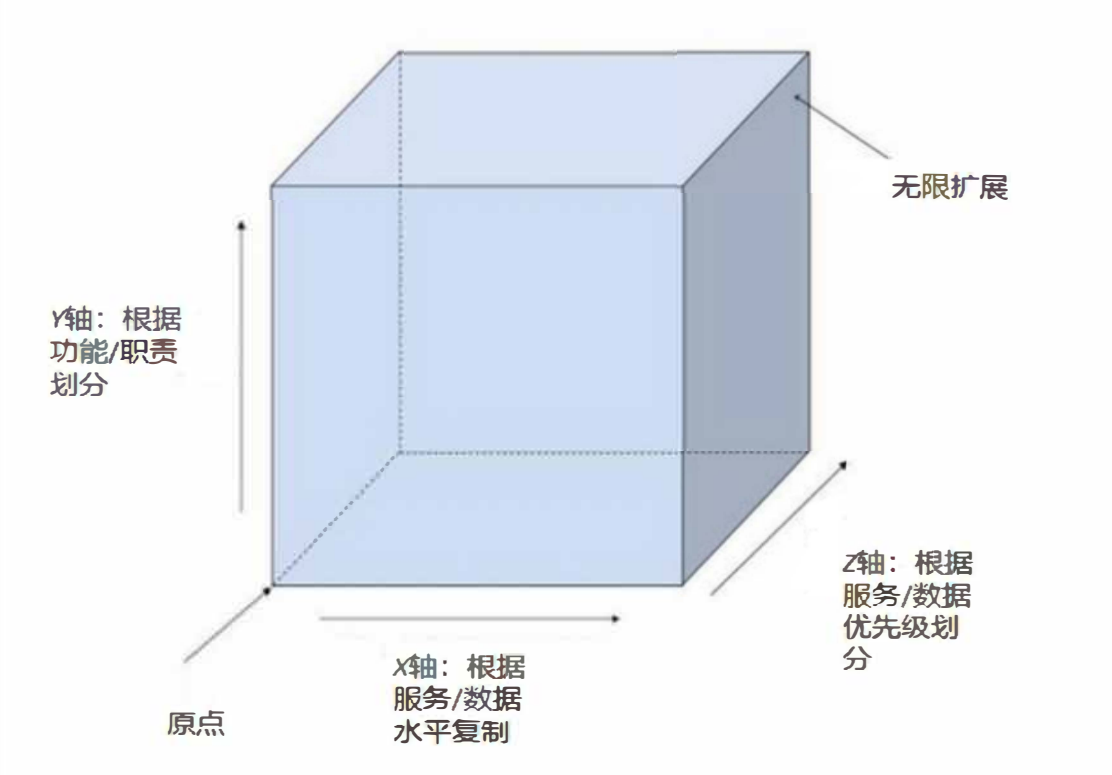
高内聚、低耦合是服务拆分的主要依据，下面我们列举一些常用的服务拆分策略，了解如何对单体架构进行拆分。

- 区分服务类型：工具服务区别于业务服务，它的特点是与业务领域无关，根据其用途可以进一步细分，一般包括的形式有公共工具服务、资源工具服务、包装器服务等。
- 根据功能定义划分服务：领域驱动设计通过分析问题空间和业务逻辑，将应用程序定义为域，域由多个子域组成，每个子域对应于业务的不同功能部分。
- 根据技术边界划分服务：对于产品类型的服务使用技术能力划分服务边界，前后端分离架构就是通过技术栈划分服务边界的典型架构模式。

**服务拆分范式**

通过增加服务实例或者机器来解决服务的容量和可用性问题是常用的可扩展架构解决方案。在《可扩展艺术》一书中提出了系统的可扩展性模型：AKF可扩展立方 注 ，可以作为服务拆分的范式。

如下图所示是使用Scale Cube的3D模型实现的一个微服务架构模型，在 X 轴上通过API网关进行水平扩展，在 Y 轴上进行单体拆分后的微服务构建，服务之间可以通过REST API进行简单交互， Z 轴是数据维度的拆分。



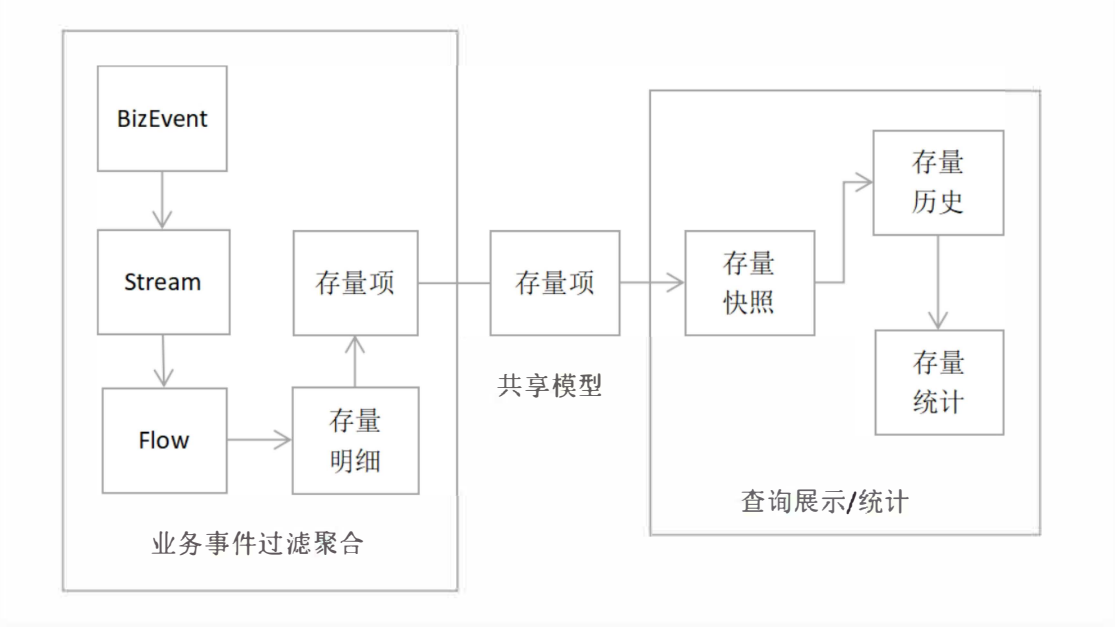

- X 轴：服务扩展，通过克隆的方式水平扩展。一般是负载均衡后运行多个应用副本，达到某个服务的高吞吐量和高可用性。
- Y 轴：功能拆分，通过拆分不同的事务进行扩展。微服务对应着 Y 轴，即将单体应用拆分为微服务应用。
- Z 轴：数据分区，通过分隔相同的事务进行扩展，例如数据库分库分表。

总之，服务支持水平扩展以提升容量；对功能的拆分体现在对业务模型的切入和深入理解上；应用数据的划分是微服务的重要原则，如果数据的耦合问题无法解决，那么应用服务的划分还会有代码耦合和级联影响。

### 3.1.4 界限上下文

在找到服务边界并把系统拆分后，我们需要使用“界限上下文”的概念明确服务之间的交互共享模型和行为接口，它不仅可以有效地限定领域的职责边界和特性范围，也可以控制问题域的规模，进而以化整为零的方式控制整个系统的复杂性。

在业务运营监控项目中，存量项模型作为业务过滤聚合服务和存量查询统计服务的共享模型，关系如下图所示。




为了实现捕获和统计监控业务运营过程中的不同阶段存量的业务状态，我们将存量项作为上述两个服务上下文的共享模型，但我们不会暴露“过滤聚合服务”中的存量明细、Flow、Stream等模块的实现细节。

作为两个独立的服务主体，它们应该在边界上有明确的界线划分和通信机制。如果服务边界与领域的界限上下文能够保持一致，那么我们已经为高内聚、低耦合的微服务架构实现了关键的一步。

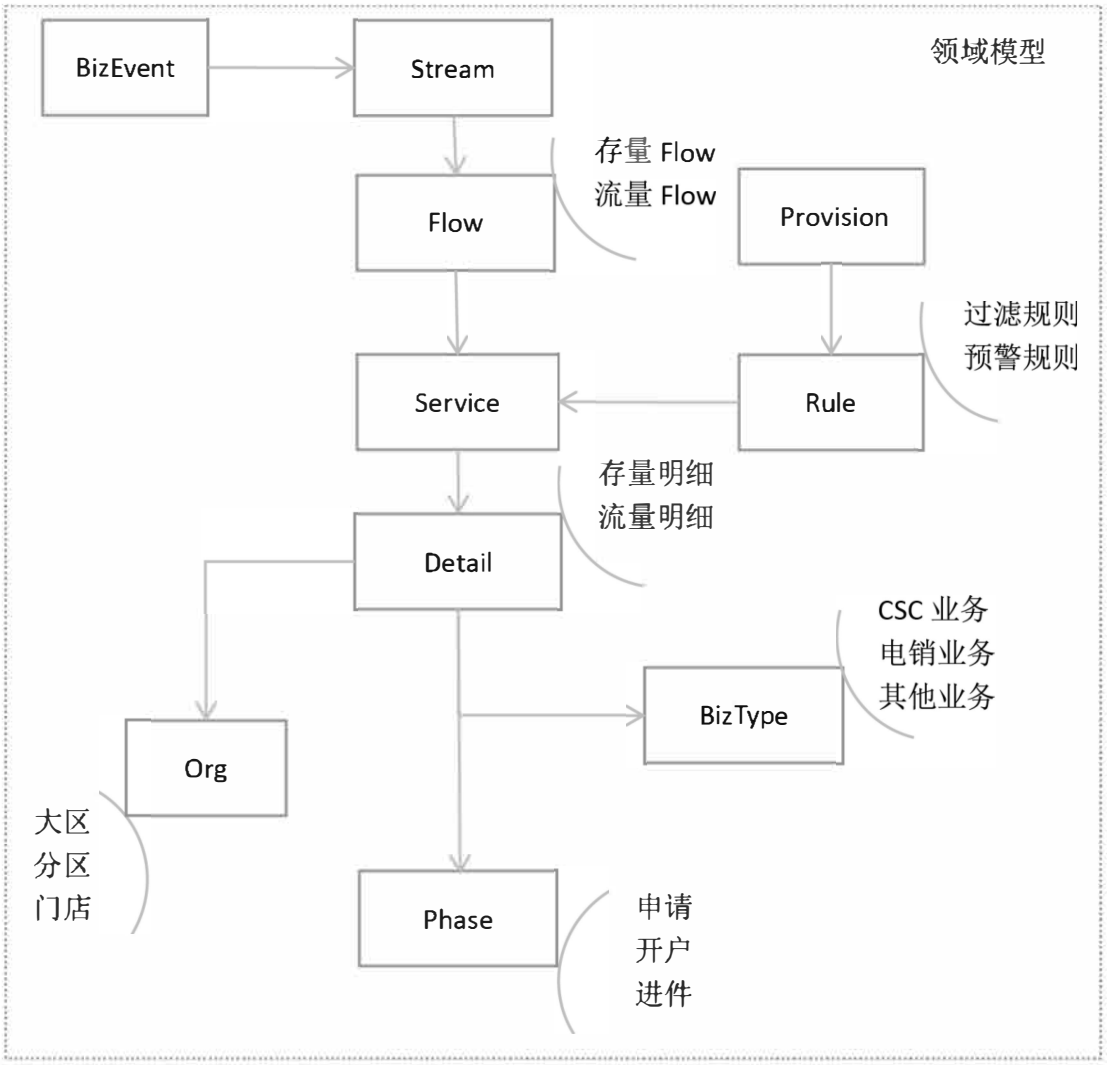
### 3.1.5 领域建模

领域建模是领域驱动设计的核心，通过领域模型可以封装对业务的抽象，建立业务概念与领域规则的关系。领域模型更关注的是业务语义的显性表达，而不是具体的数据存储及代码逻辑实现细节，它可以有效地降低业务人员和技术人员之间的沟通成本。

**案例分析**

回到“业务运营监控系统”中，我们把业务监控的核心诉求聚焦在“业务事件”，以及业务的存量和流量领域模型。

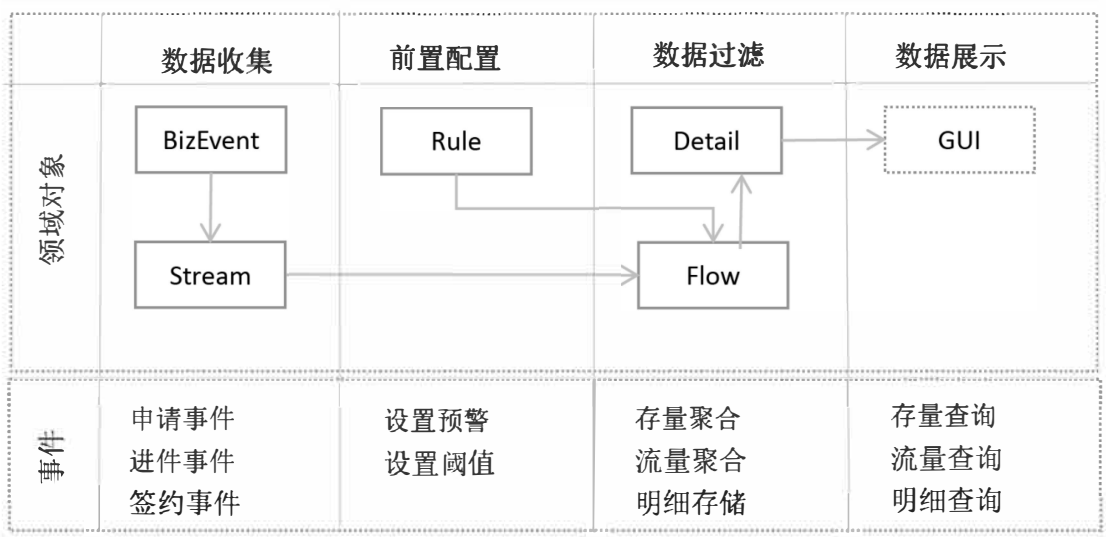
在整理了领域服务的核心模块后，我们可以把业务方关注的组织信息、业务类型信息、业务阶段信息进行进一步领域模型细化，如下图所示。




- BizEvent：业务事件是业务监控的数据源，使用统一的JSON格式记录消息事件，以日志方式封装当前业务系统发生的事件详情。
- Stream：对应一个端到端的数据流转概念，通常我们会将BizEvent事件发送到Kafka的一个Topic上，通过建立Stream可以在消费端处理指定Topic上的数据流。
- Flow：Flow对应一个监控业务计算逻辑，存量Flow可以统计对应的存量状态，流量Flow统计当前业务的流量状态。
- Service：它并非领域对象，表示一个通用的服务层，Service提供业务存量和流量的查询、备份、预警等业务方法。
- Provision：用户配置前置通用服务，不对应领域对象，主要接收用户的配置请求，并保存为业务规则。
- Rule：即规则模型，属于核心领域模型，业务方可以通过它灵活地定制关心的业务状态并进行预警、过滤等。
- Detail：属于业务的中间监控过程详情，属于领域对象，同时包含组织、阶段、业务类型等明细对象属性（Org、Phase、BizType）。

使用领域建模的设计方法可以进一步将“业务监控系统”内部的领域服务与领域模型对象关联，显性地表达每个领域模型的具体工作职责及业务行为事件与领域对象之间的上下文映射关系，如下

图所示。

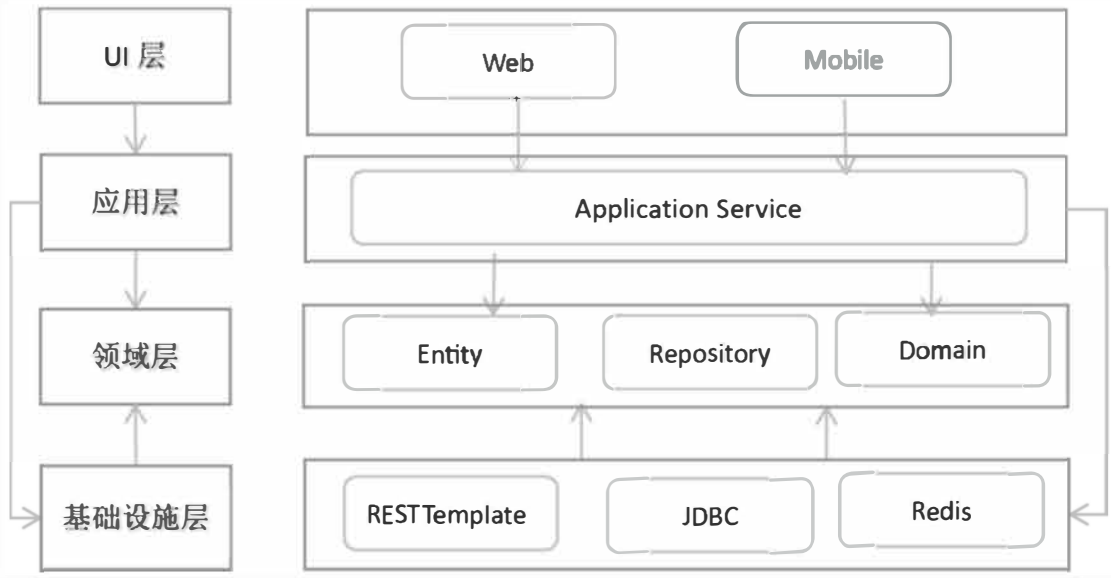



## 3.1.6 架构设计

架构设计的本质是管理业务和技术复杂性，使系统易于有序化重构及扩展。高质量的架构一定是高度抽象的、围绕业务的、易于理解的、面向演进的。

**分层架构设计**

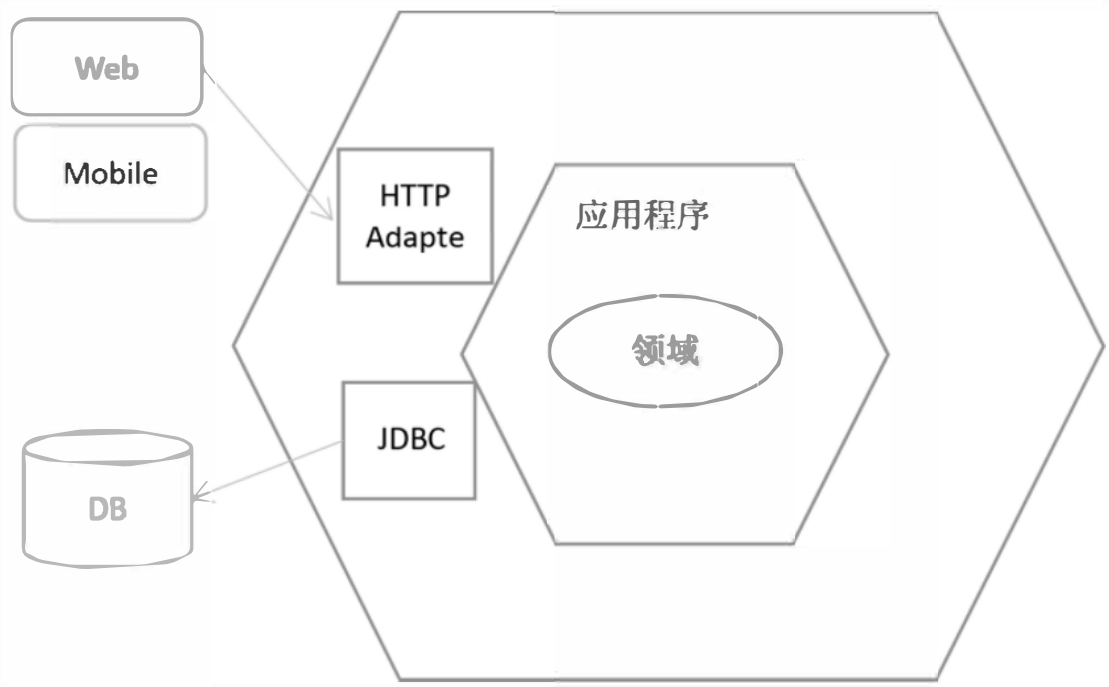
领域驱动设计遵循“关注点分离”原则，将技术实现逻辑封装在基础设施层；将业务逻辑封装在领域层，尽量使领域层代码与其他层技术细节分割开来；将应用层作为黏合剂，实现前两者的协作；同时UI层可以基于Swagger技术暴露REST API。分层架构如下图所示。




**六边形（Hexagonal）架构模式**

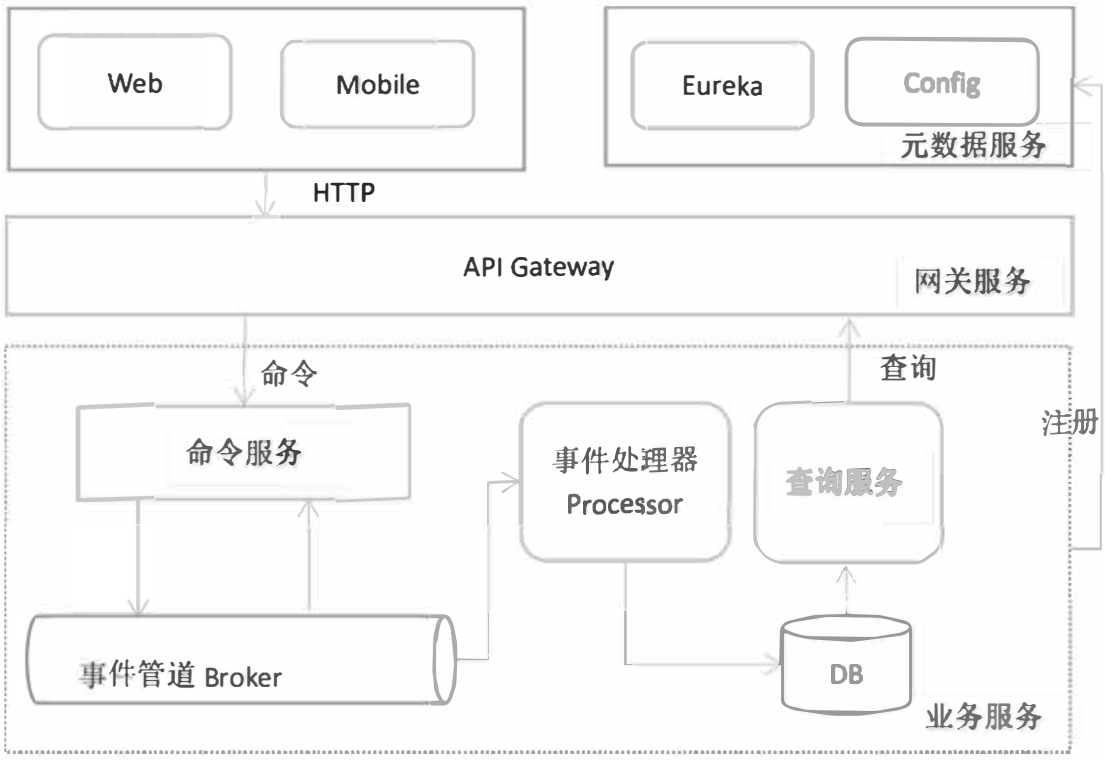
六边形架构模式又称为“端口–适配器”模式，它将系统分为内部和外部。内部代表应用的业务逻辑，外部代表应用的驱动逻辑、基础设施或其他应用。内部以API接口呈现，通过端口和外部系

统通信。外部系统需要使用不同的适配器，适配器负责对协议进行转换。应用程序能够以一致的方式与实际运行的设备和数据库相隔离，方便开发和测试，六边形架构模式如下图所示。




**微服务架构模式**

微服务架构是强调细粒度、单一职责的架构模式。微服务架构更关注的是系统的非功能需求：质量属性、演进能力、扩展性、观测性、软件交付效率等。微服务使用CQRS（命令/查询职责分离）中的事务脚本模式应对查询场景，而对于复杂的业务逻辑场景，使用领域驱动设计模式。微服务架构模式如下图所示。



## 3.2 微服务化改造

对单体架构现状的不满和难以控制是推动微服务化改造的重要因素，企业在向微服务架构转型的过程中面临诸多挑战，需要采用相应的策略模式进行微服务化改造。

### 3.2.1 技术债务

单体架构下技术债务的产生原因多种多样，总结下来这些技术债务大体可以分为业务复杂、交付质量低、非功能需求不达标等三大类。

- 业务复杂：开发人员依靠模块的叠加加速软件交付，后期形成规模庞大的单体架构，导致业务代码臃肿、业务逻辑耦合、无法复用等问题。
- 交付质量低：单体架构缺少自动化测试能力，存在局部代码质量问题，容易引发整个系统的可用性问题。
- 非功能需求不达标：代码的腐化和缺少维护、重构、改进导致性能逐渐下降等问题，在极端情况下，甚至出现不同资源竞争的短板效应，造成整个系统崩溃。

### 3.2.2 微服务化改造时机

对于存在技术债务的单体架构，在实施微服务化改造工作前，需要从客观和主观两个方面来判断当前时间点是否是进行微服务化改造的最佳时机。

所谓客观因素包括上一节所说的技术债务因素。此外，代码冲突频率、组织人员规模、产品迭代速速、用户规模量级等量化指标也可以作为微服务化改造的重要客观依据。

微服务化改造时机同样受到技术团队主体愿望和人员技术能力的制约，概括如下：

- 团队的技术选型需要达成一致，并在组织层面上有一致的指导和规范。
- 团队需要根据业务所处阶段、当前系统项目使用的技术栈和团队人员能力决定是否适宜转型微服务。

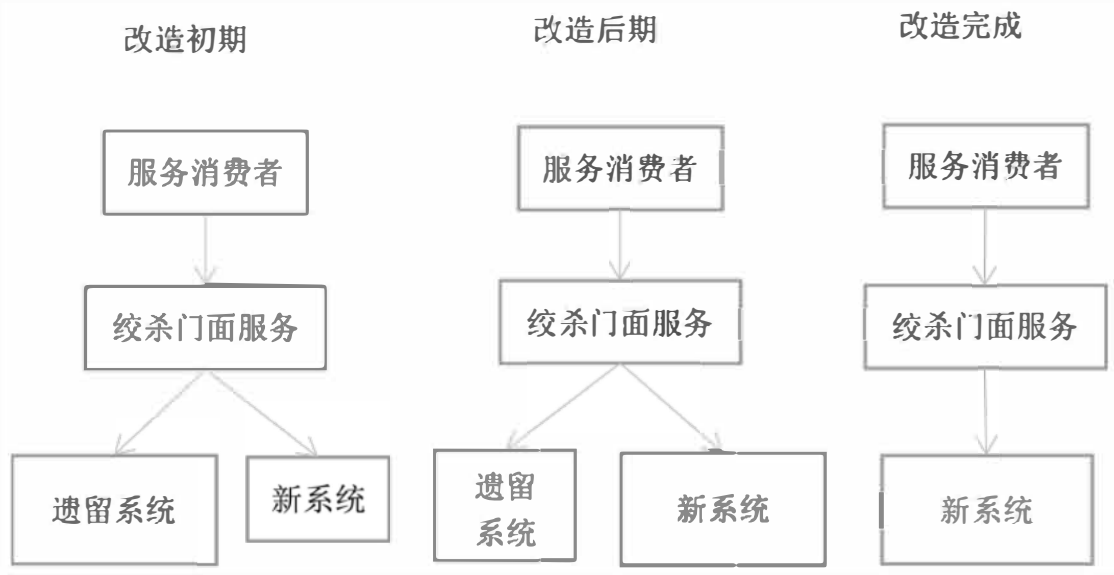
● 团队是否具备自动化交付和微服务治理平台等技术支撑能力。

### 3.2.3 单体架构的改造模式

单体架构进行微服务化改造需求遵循一定的改造模式。在不影响业务正常运行的前提下实现业务的平滑过渡，下面我们列举一些经常使用的微服务化改造模式。

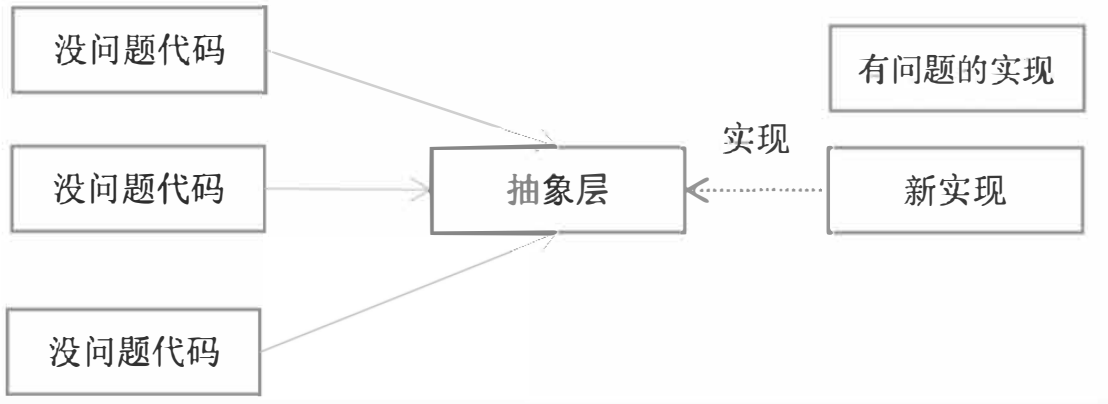
**绞杀者模式（Strangler Pattern）**

绞杀者模式通过逐步替换而非一次性替换的方式来保证新旧系统的平滑过渡。运用一系列易于理解的小规模替换定期交付新的微服务，逐步淘汰遗留系统的功能模块，最终实现全部替换，流程如下图所示。




**修缮者模式**

修缮者模式源于古老的软件工程格言“任何问题都可以通过增加一个中间层解决”，就如修房或修路一样，将老旧待修缮的部分进行隔离，用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍然能够协同完成工作。修缮者模式的基本原理来自Martin Fowler的重构方法，如下图所示。

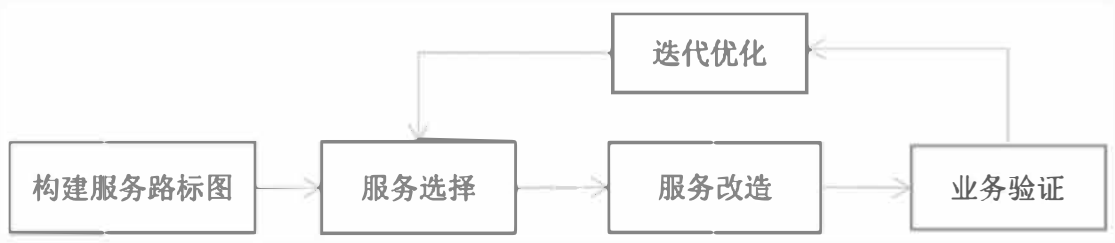



这种模式的实现方式可以分成三个主要步骤。

- 抽象层提取：首先通过识别内部的待拆分功能，对其增加抽象接口层，同时对原有代码进行改造，确保其同样实现该抽象层，这样在依赖关系上就添加了一个中间层。
- 抽象层实现：为抽象层提供新的实现，新的实现采用微服务方式。
- 抽象层替换：采用新的实现对原有的各个抽象层实现进行逐步替换，直至原有实现被完全废弃，从而完成新老实现方式的替换。

**演进式改造流程**

演进式改造流程是一种以逐步演进的方式对遗留系统进行改造的流程，通过构建服务路标图、服务选择、服务改造、业务验证、迭代优化完成微服务化改造，如下图所示。




- 构建服务路标图：由架构师、业务分析师及技术负责人共同参与构造出一个服务路标图，并接受来自各个方面人员的反馈。
- 服务选择：有了服务路标图之后，遵循价值最大化的原则，从多种角度去制定优先拆分策略，优先拆分相对独立、容易实施的业务部分。
- 服务改造和业务验证：在改造过程中需要验证新的服务是否满足业务需求。在新服务上线投入使用并稳定后，可以从遗留系统中移除原有的代码模块。

**边车（SideCar）模式**

传统企业中存在大量的遗留系统，对这些遗留系进行微服务化改造的成本很高。对于这些系统，我们的选择并不一定是将其进行微服务化改造，而是将其接入微服务环境中，与其他服务共同协作来实现业务需求。边车模式可为不同语言的遗留系统提供一个同构的接入接口。对于原遗留系统应用程序的每个实例都部署和托管了一个边车实例，实现非侵入式接入。

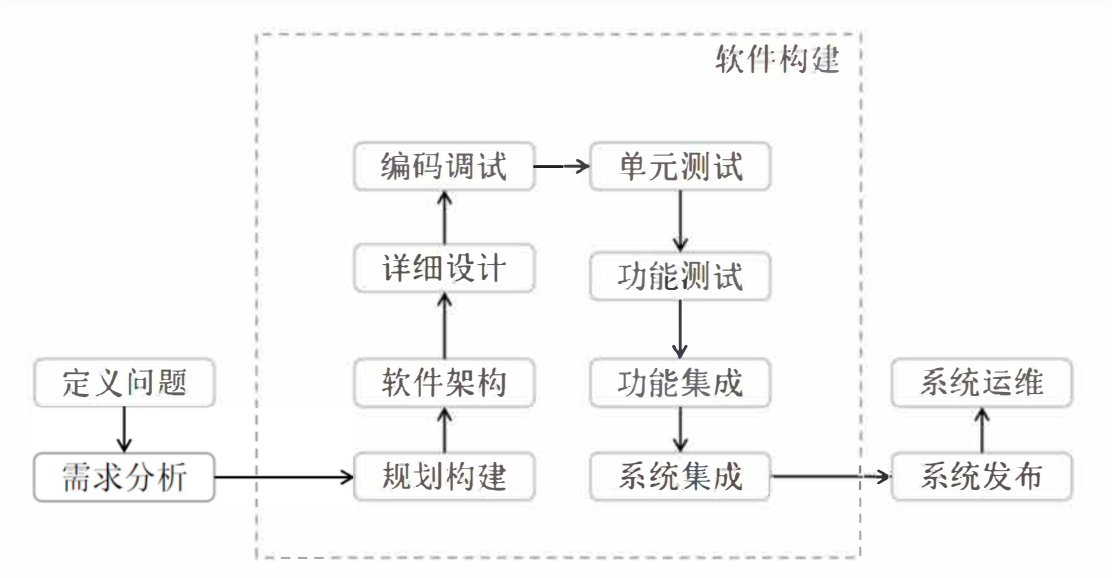
## 3.3 微服务构建进阶

本节我们将从更宏观的软件构建视角切入来总结微服务构建的最佳实践，宗旨是指导开发者合理地设计和构建可演进式的系统架构。

## 3.3.1 软件构建

软件构建通常是指软件的详细架构设计、编码、调试、测试和集成等方面的工作。作为软件开发的主要组成部分，软件构建依然是软件开发工程的核心活动，它在整个软件开发交付周期中大概占到30%～80%的时间。

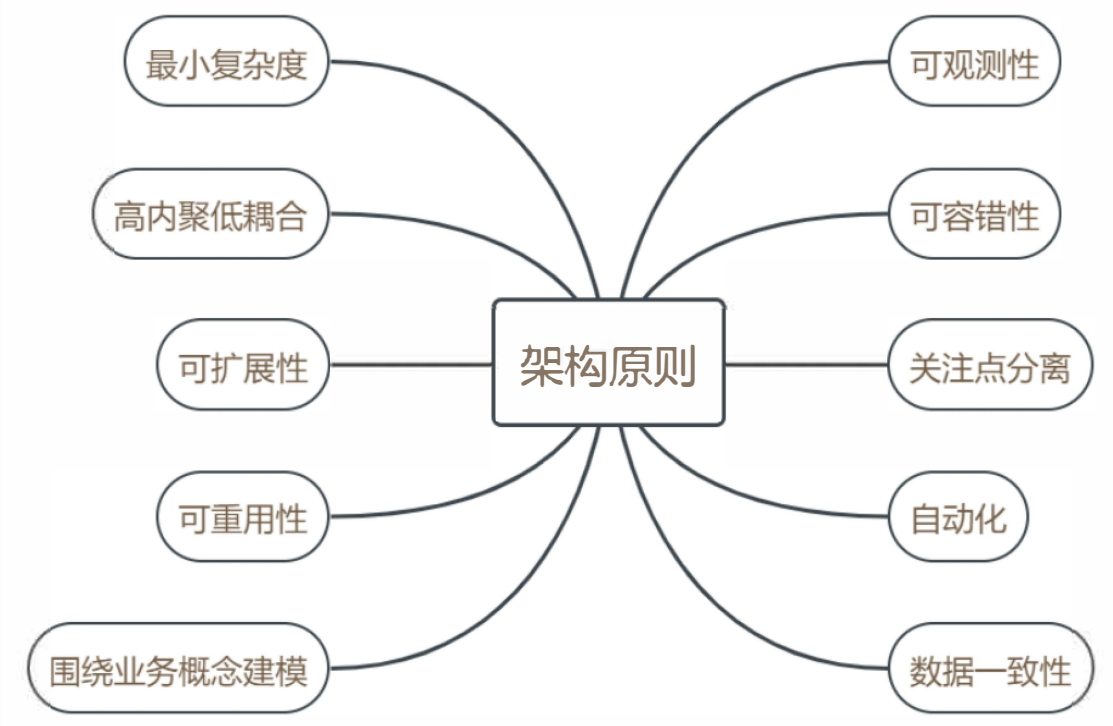
下图是经过几十年的实践积累，研究人员给出的软件开发过程中各种不同关键活动的时间顺序。




上述活动独立拿出来都可以作为软件工程中的主题加以讨论，而“软件构建”作为软件开发中的核心活动显然具有更重要的地位。

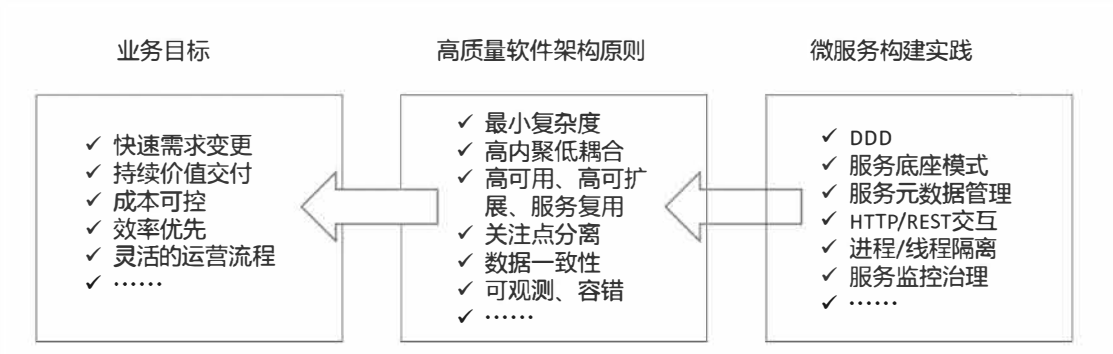
软件构建要重视代码质量，因为源码是软件唯一的精准描述和最终产物，敏捷的开发流程也强调可阅读的代码大于规格文档的理念。源码可以增强软件工程的可管理性，源码的质量和可读性是决定软件能否持续演进的重要因素。

软件构建使用高层架构设计指导系统的编程开发约束。高质量的架构可以使构建活动更加容易，使复杂的系统分解成为可管理、独立、层次化的软件集合；而糟糕的架构设计将使你的构建活动举步维艰、困难重重，严重影响系统的后期维护和扩展。下图是我们总结的高质量架构原则。




## 3.3.2 微服务构建实践

微服务构建倾向于使用领域驱动设计模式，从技术实现的层面遵循并实践高质量的软件架构原则，目标是持续快速地满足业务需求，支撑灵活的软件工程流程，实现成本可控及高效的价值交付。我们可以将业务目标、高质量软件架构原则、微服务构建实践三者的关系表述如下图所示。




如果对微服务构建实践从时间维度做进一步细化，我们可以将其划分为微服务架构定义、架构落地、规模化发展三个阶段设计。

在微服务架构定义阶段，我们使用领域驱动设计的方法论来完成对业务的建模及服务边界的接口定义。与传统的基于技术实现的分层架构模式不同，领域驱动设计更加关注业务，它在实际业务场景中通过业务边界的识别对领域服务进行界限划分；微服务架构强调功能职责单一、围绕业务组织团队、轻量级的集成交互机制等特性，都与领域驱动设计模式概念高度一致、不谋而合。

在微服务架构落地阶段，需要一个具备承载业务、简单生产就绪的软件底座技术来启动和运行我们的业务服务，在技术选型上，Spring Boot基于约定优于配置的编程范式，可以极大地提升软件开发人员的工作效率。同时成熟的社区支持、广大的开发群体、全面完整的技术框架支撑都是Spring Boot成为微服务架构落地的首选技术栈。

在微服务架构规模化发展阶段，服务治理成为解决系统复杂性、可用性、可观测性、可靠性、容错性、关注点分离的关键技术。在技术选型上，我们使用Spring Cloud技术生态作为微服务的治理体系。此外，服务集成交互机制、后端数据一致性、容器技术、持续集成、持续交付、监控治理都是微服务构建和规模化管理需要重点关注的技术领域。

微服务架构的发展趋势是将更多的公共组件、模块能力、横切关注点下沉到基础设施。软件构建的复杂性从业务层转移到基础设施层，通过将业务代码与技术架构解耦，实现业务价值的快速交付。

### 3.3.3 微服务架构反模式

前面介绍了微服务的一些好的设计模式和实践，下面列举一些微服务架构的反模式，这样我们可以在微服务构建过程中更好地进行取舍。

**微服务划分粒度越小越好**

微服务粒度的大小并没有统一的业界标准，并非越小越好。服务划分的主要依据是业务属性，组织层级、团队规模、代码规模、业务复杂度都对服务粒度划分产生影响。

**内聚不相关功能服务**

服务必须清楚地与业务能力保持一致，不应该试图做一些超出范围的事。功能隔离问题对架构治理而言至关重要，否则它会破坏敏捷性、性能和可扩展性，导致创建一个紧耦合、无法扩展演进的架构。

**不重视自动化**

随着微服务规模的扩大，自动化集成和交付成为微服务交付的关键。微服务的目标是驱动敏捷，为我们提供所需的自动化工具。

**服务架构分层**

团队过度关注技术层面的内聚，而不是功能相关的重用。这样会形成一个由横向团队管理的人造物理层，导致交付依赖。

**手动配置环境**

当我们创建的服务越来越多时，服务的配置管理会面临失控的风险。大部分生产部署不顺利的情况都是由于配置错误造成的，手动管理配置信息在微服务架构下将变得越来越困难。

**未使用版本控制**

在微服务的世界里，复杂度会随着服务数量的增加而增加。制定一个版本控制策略可以使服务的消费者轻松迁移，并且服务提供者可以透明地部署变更而不产生级联影响。

**缺少API网关**

每个服务都要单独实现鉴权、过滤等功能，正确的做法是集中管理和监控部分非功能性问题。API网关可以编排跨功能的微服务，同时实现共享服务的复用。

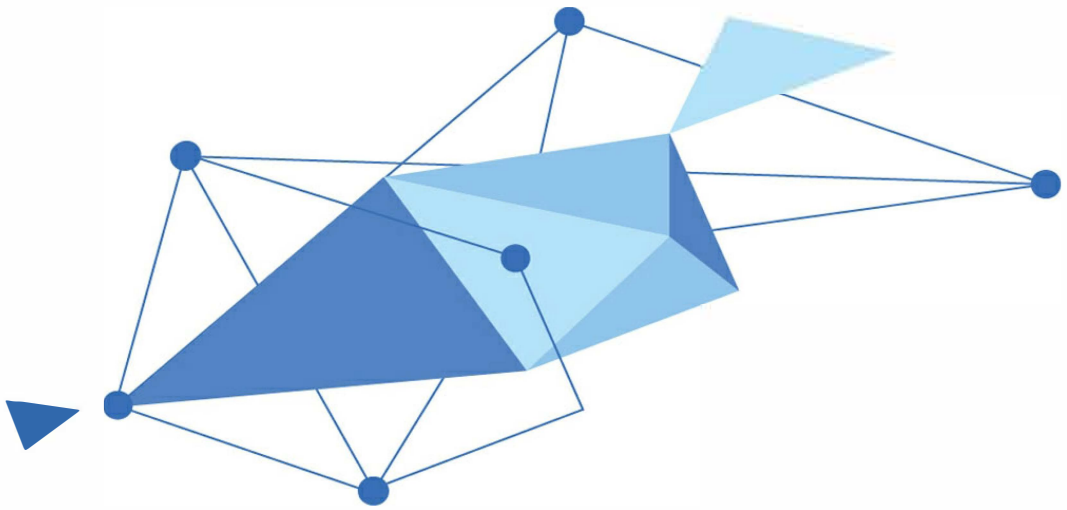
**服务依赖第三方系统**

当服务依赖第三方系统时，服务就不是松耦合的了。要让服务有独立性，交付的每个服务都必须具备独立的测试套件。

## 3.4 小结

领域驱动设计可以保证业务模型和代码模型的一致性，把业务与技术复杂性分离，通过边界划分来控制业务的复杂性，目前微服务架构的兴起带来了实现领域驱动设计的最佳实践环境。

软件构建过程本质上是一个复杂的过程，这种复杂性伴随在软件工程的整个生命周期。使用微服务架构、领域驱动的软件建模模式可以让我们找到这种复杂性问题的解决之道。



# 实践篇

## 本篇内容

本篇是微服务架构的实践篇，我们将从技术实现层面讨论如何实践和落地微服务架构。

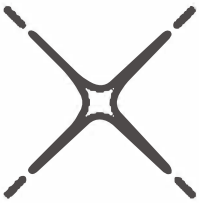
在微服务架构模式下，使用一种称为“基底”（chassis）模式的服务开发方式可以快速高效地实现服务启动、服务注册、配置管理、容错、负载均衡、安全等基础功能。本篇重点介绍Pivotal公司开源的Spring Boot项目，以及在这种模式下的技术实践。

微服务架构的难点是分布式架构下细粒度服务交互的复杂性和服务治理。本篇通过介绍Spring Cloud框架，讲解微服务治理体系的关键技术，以及如何保证服务的SLA 注 。

同时，在细粒度服务的交互集成、数据一致性管理、服务交付部署、服务监控跟踪等方面，我们都将介绍当前主流的技术实践和解决方案。

第4章

# 脚手架



微服务本身是一种架构风格，也是指导组织构建软件的一系列最佳实践集合。然而，业务团队在拆分应用后，会产生更多细粒度服务，并面临这些服务在分布式网络环境中的复杂性。如何专心实现业务逻辑而不陷入微服务架构的技术细节，对开发者来说是一大难题。

本章将介绍脚手架的概念，并介绍JVM环境下在技术成熟度、架构完整性、生态活跃度等方面都占据优势的微服务脚手架项目：Spring Boot。

## 4.1 脚手架介绍

### 4.1.1 什么是脚手架

脚手架是一种用在建筑领域的辅助工具，或者说是为了保证各施工过程顺利进行而搭设的工作平台，有兴趣的读者可自行查看维基百科上的定义。

对应到软件工程领域，脚手架可以解释为帮助开发人员在开发过程中使用的开发工具、开发框架，使用脚手架你无须从头开始搭建或者编写底层软件。下面的“脚手架”定义来自Stack Overflow[注]，更加偏向于应用服务框架使用的一种编程思想或者说编程范式（供参考）。

> 脚手架：是一种元编程的方法，程序员编写一份规格说明书（Specification），用来描述怎样去使用数据库，然后由编译器脚手架根据这份规格说明书生成相应的代码，进行增、删、改、查等数据库的操作，在脚手架上更高效地建造出强大的应用。

### 4.1.2 为什么需要脚手架

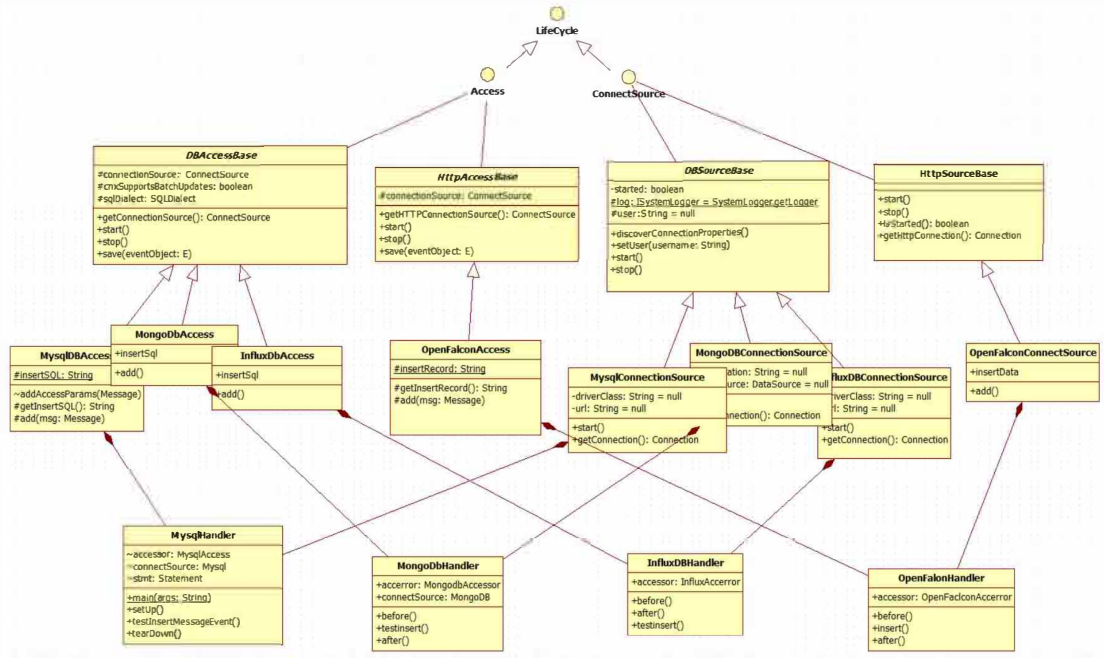
为什么软件开发需要脚手架呢？我们通过软件开发的一些基本原则看一下脚手架对软件工程的重要作用。

- 复用原则（Reuse Principle）：现在我们推崇的是极致化的编程体验，缩短的开发时间、大量的开发任务、支持需求的变更、高频率的应用服务交付，这些都给软件开发人员带来了前所未有的压力。其中，软件复用技术被公认为解决这些问题的行之有效的方法。从计算机软件编程的发展历史来看，从面向过程的编程语言到面向对象的高级编程语言的广泛使用，是软件复用技术进步的体现。从对象的复用到更大的组件复用，再到如今的框架的复用、服务的复用，都是在利用他人的优秀成果来放大你的工作价值。当一个新手使用脚手架时，对于一个具体问题，可以套用现成的解决方案加以扩展。使用脚手架的应用，仅需通过简单的注解和配置就可以具备健康状态检查、生产环境就绪、可观测等基本服务能力。对于一个业务逻辑问题，可以复用已有的逻辑，一步步迭代，敏捷开发。
- DRY原则（Don't Repeat Yourself）：DRY原则直译过来就是“不要重复你自己”。这一原则和复用原则类似，强调尽量在项目中减少重复的代码行、重复的方法、重复的模块。其实，软件设计原则和模式最本质的思想都是“消除重复”。我们经常提到的重用性和可维护性其实是基于减少重复这一简单而重要的思想的。DRY原则意味着系统内的每一个部件都应该是唯一的，并且是具有明确含义的（不模糊的）。我们可以通过应用职责单一、接口隔离等原则尽量拆分系统、模块、类和方法，使每一个部件都是职责明确并且可重复使用的。
- 开闭原则（Open Close Principle）：开闭原则中的“开”就是指对功能的扩展是开放的，“闭”是指对于原有代码的修改是封闭的。通俗一点讲，软件系统通常是由各种模块组成的，软件系统在增加一项新的功能时，应该在不修改现有代码的基础上操作。实现开闭原则的关键就是“抽象”，从微观的角度讲，开闭原则适用于一个业务模型的类的设计，把系统内的所有可能行为抽象为一个抽象底层，在这个抽象底层中规定需要提供的方法接口，具体实现类通过集成、代理、委托的方式，扩展实现新的行为或者新的功能。从宏观的角度讲，我们说开闭原则就是将公共模块、开发约定、最佳技术实践经过共享、提炼沉淀到封闭的底层技术基座；而将变化频繁的业务模块、独特的功能逻辑通过继承、组合和集成的方式实现对扩展的开放。

### 4.1.3 不要重新造轮子

不要重新造轮子（Stop Reinventing The Wheel），这个原则可以说是软件开发里的“金科玉律”。在实际的软件工程场景中，脚手架的使用正是为了我们避免重新造轮子。如果你不借助已有框架或者工具，不仅不会提升开发效率，还会将自己陷入重新造轮子的风险中。这里举一个发生在本人参与的实际项目中的血淋淋的案例。

我们在一个服务治理项目中需要在数据持久层实现一个通用的数据存储接入组件，初衷是降低应用接入不同数据存储引擎的复杂性，屏蔽使用者对不同持久层的感知差异，使开发者通过简单的配置就能适配不同的数据类型。另外，因为期望完全掌握对持久层的控制，以及实现深度定制化的数据转换功能，所以我们并没有使用Spring框架，而是走上了自研持久层的道路。如下图所示是自研持久层的UML框架图。




我们分别对接了MySQL、MongoDB、OpenTSDB、HBase等数据源。系统从架构设计到落地花费了大概两个月的时间，还不包括后期的测试、对接、调试、修复Bug至少一个月的磨合期。然而，由于人员、经验、项目周期等各种因素，最终自研持久层框架的效果并没有达到预期，后期在与业务对接的过程中还出现了各种技术和业务适配问题。

在经过一番技术调研后，我们决定逐渐使用Spring Data替代原有的自研持久层框架。经过项目的实践后，我们发现在业务的需求满足性、易用性、开发效率、业务稳定性等各个方面，Spring Data都具备压倒性的优势，它不仅可以完全满足业务和技术上的需求，而且可以简化我们的开发工作、显著提升工作效率。二者的复盘对比如下表所示。

| 对比项 | 自研持久层 | Spring Data | Comment（说明） |
| --- | --- | --- | --- |
| 代码量 | 框架代码量+业务适配代码量 | 业务适配代码量 | 总体代码量，粗略统计自研代码是Spring Boot代码的10倍 |
| API使用难度 | 使用JSON作为API接口 | 使用Template规范 | Spring具有面向对象的API接入方式，使用更加方便和友好 |
| 技术支持 | 只有自研团队做技术支持 | Spring框架在互联网上有大量的资源可参考 | Spring具有更好的技术支持 |

续表

| 对比项 | 自研发持久层 | Spring Data | Comment（说明） |
| --- | --- | --- | --- |
| 可靠性 | 由于技术框架开发时间有限，可靠性差 | 可靠性高 | Spring依赖成熟的社区力量，拥有更高的可靠性 |
| 成熟度 | 技术成熟度低 | 有较高的技术成熟度 | Spring技术更加成熟可靠 |
| 运维难度 | 技术框架运维复杂，需要大量配置文件等 | 基于注解的声明式架构 | Spring框架由于成熟度高，主要运维难度在业务实现上 |
| 业务适配性 | 自定义的JSON语法，业务适配需要额外的代码 | Spring提供更完善的适配API和注解支持 | Spring框架优势明显 |
| 人员成本 | 需要额外的框架开发人员 | 只需要业务开发人员 | 自研持久层的运维成本高 |

Spring Data项目通过使用对象的语义可以让我们更方便地操作不同类型的数据。它将应用的骨架部分通过“抽象”提取出来，形成了一套系统的开发范式和行为模式。Spring Boot脚手架也为自定义的复杂查询、修改操作提供了扩展的Repository类和自动化配置，使添加定制化的扩展方法更加轻松方便。我们只需要理解Spring为我们提供的操作API接口，就可以实现复杂的查询等业务逻辑。

Spring Data将我们的数据持久层框架进行了进一步的封装，开发者通过简单的注解，就可以实现将不同类型数据放到不同持久层集合的存储映射操作。可以说，脚手架工程可以为开发人员屏蔽繁杂的数据存储引擎底层差异和具体工作细节，提升了开发效率，降低了开发难度。

除非你是这个领域的专家，或者没有现成的软件脚手架能够满足你的需求，否则请停止“愚蠢地重复造轮子”的行为。

### 4.1.4 常用脚手架

下面列举一些软件开发中经常使用的脚手架，看一下如何通过脚手架提高我们的开发效率。

**Vue框架**

对于前端开发人员来说，Vue无疑是一套简单的、易于使用的构建用户界面的前端脚手架。根据Vue的官网说法，Vue是一套构建用户界面的渐进式的JavaScript框架。与其他重量级框架不同的是，Vue采用自底向上的增量开发的设计，Vue的目标是通过尽可能简单的API实现响应的数据绑定和组合的视图组件。

vue-cli脚手架构建工具，可用于快速搭建大型单页应用。该工具提供“开箱即用”的构建工具配置，带来了现代化的前端开发流程。只需几分钟即可创建并启动一个带热重载、保存时静态检查

及可用于生产环境的构建配置的项目：

```
#全局安装 vue-cli
$ cnpm install --global vue-cli
```

要创建基于Webpack模板的项目，首先我们选定目录，然后在命令行中把目录转到选定的目录即可，可以使用下面的命令：

```
#my-project 为自定义项目名
$ vueinit Webpack my-project
初始化一个项目，可以使用：
$ vueinit Webpack-simple my-project
```

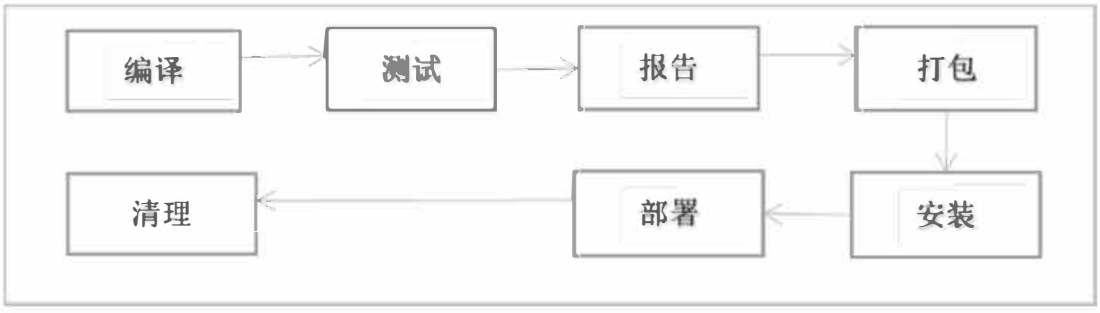
**Maven**

Maven是一个跨平台的项目管理工具，是服务于Java平台的项目构建、依赖管理、项目信息管理工具。同时使用Maven可以规范项目骨架及包层次结构、命名配置文件、生成代码原型等。

Maven提供了archetype插件来帮助开发人员快速勾勒出项目的骨架，要使用本地Maven仓库中的脚手架创建新项目，直接执行如下Maven命令，根据提示依次输入groupId、version、package信息即可：

```
mvn archetype:generate \
-DarchetypeGroupId=com.xxx \
-DarchetypeArtifactId=archetype-spring-boot \
-DarchetypeVersion=1.0.0 \
-DgroupId=com.xxx \
-DartifactId=demo-archetype-generate \
-Dversion=1.0.0 \
-DarchetypeCatalog=internal \
-X
```

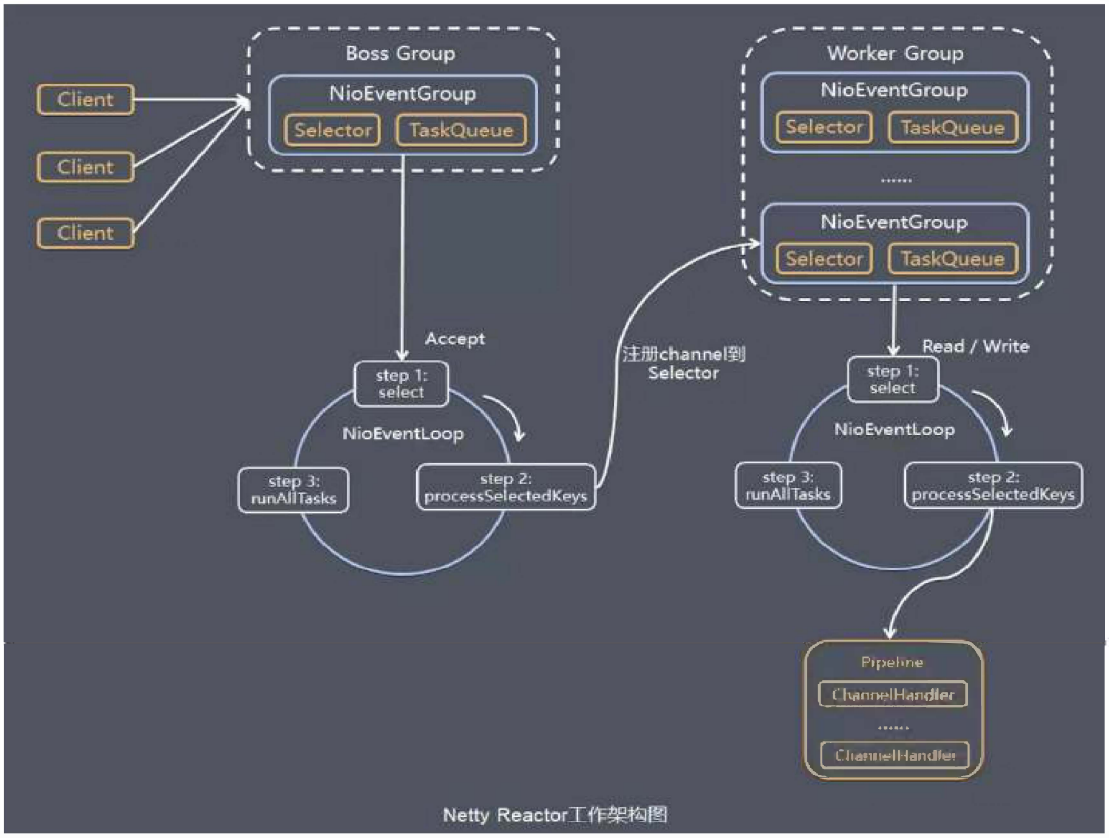
Maven自动化构建简化了开发人员手动构建工程的过程，规范了项目的构建过程。Maven自动化构建流程如下图所示。




**Netty**

Netty是JBoss提供的一个Java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可用性的网络服务器和客户端程序。在Java世界中还没有Netty框架的时候，Java自带的NIO非常复杂，并且还会出现Epoll Bug（代码缺陷），这个Bug会触发Selector空轮询，导致CPU的使用率达到100%。Netty的解决方式是，在 N 次空轮询后自动关闭

Selector，避免了原生NIO的空轮询问题。而且Netty有很好的线程模型和内存管理框架，如下图所示是Netty Reactor工作架构图。




**Java EE**

Java EE的全称是Java 2 Platform Enterprise Edition，它是在SUN公司领导下，多家公司参与共同制定的企业级分布式应用程序开发规范。

Java EE技术由一系列技术规范和技术组件组成，包括RESTful Web Server（JAX-RS）、Jersey Rest框架、Java Servelt、JMS、EJB等。通常可以把满足这些标准的业务应用部署在Tomcat、JBoss等Web服务器上运行。

**Dropwizard**

Dropwizard只需通过简单配置就能让你的类提供RESTful服务。Dropwizard是一个微服务框架，是各项技术的一个集成封装，它包含以下组件：

- 嵌入式Jetty：一个应用程序被打包成一个jar文件，并使用自己嵌入的Jetty容器。除此之外，无任何其他war文件和外部Servlet容器。
- JAX-RS：Jersey，用来写基于REST的Web服务。
- JSON：REST服务数据传递处理全部用JSON，使用Jackson库。
- 日志：使用Logback和SLF4。
- 数据库：使用Hibernate集成ORM框架。

- 指标：使用Metrics作为指标度量工具，在Java代码中嵌入Metrics代码，可以方便地对业务代码的各个指标进行监控，同时Metrics能够很好地跟Ganlia、Graphite结合，方便地提供图形化接口。

在微服务架构领域，Dropwizard可以说是早期脚手架的一个代表，然而Spring Boot青出于蓝而胜于蓝，在Spring强大和成熟的技术生态下，Spring Boot展现出来的特性更加优雅，也更加契合当前微服务架构的理念，下一节我们将正式开始Spring Boot之旅。

## 4.2 Spring Boot启动

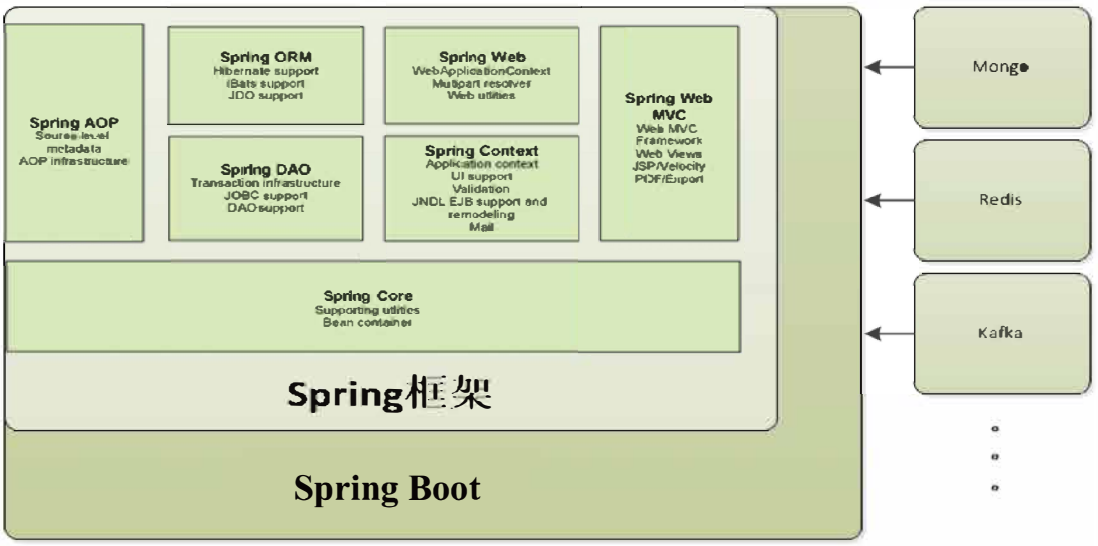
Spring Boot是Spring旗下的一个子项目，其设计目的是简化Spring应用的初始搭建及开发过程，Spring Boot可以快速启动和运行你的Spring应用服务。

### 4.2.1 Spring Boot概述

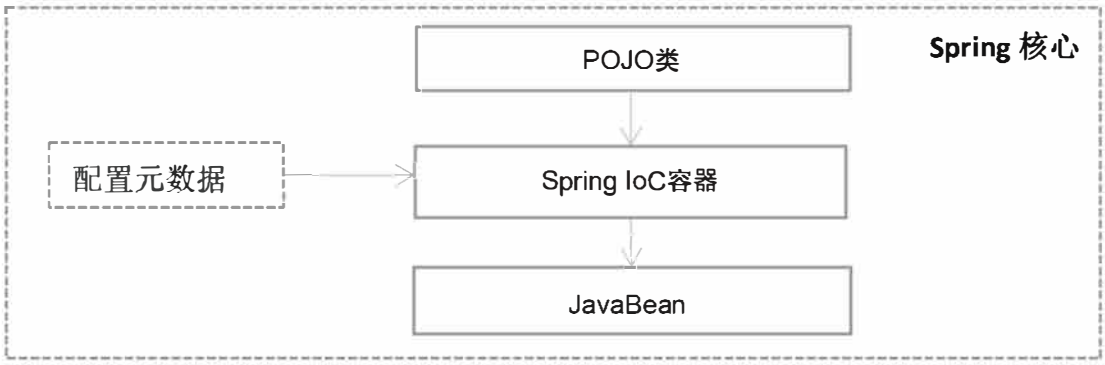
Spring Boot本质上是基于Spring内核的一个快速开发框架，是“约定优先于配置”理念下的最佳实践，通过解析Spring Boot的启动过程，可以帮助我们逐渐了解它的工作机制和其背后整合Spring快速开发的实现原理。

**磨刀不误砍柴工**

在开始讲解Spring Boot之前，首先让我们从整体架构上认识Spring家族，正所谓“不知全局者不足以谋一域”，如下图所示是Spring Boot与Spring生态的关系。




● Spring Core：Spring Core是Spring框架的核心模块，集成在Spring框架中，提供了我们熟知的控制反转（IoC 注 ）机制。Spring的核心是管理轻量级的JavaBean组件，提供对组件的统一生命周期和配置组装服务管理，如下图所示。



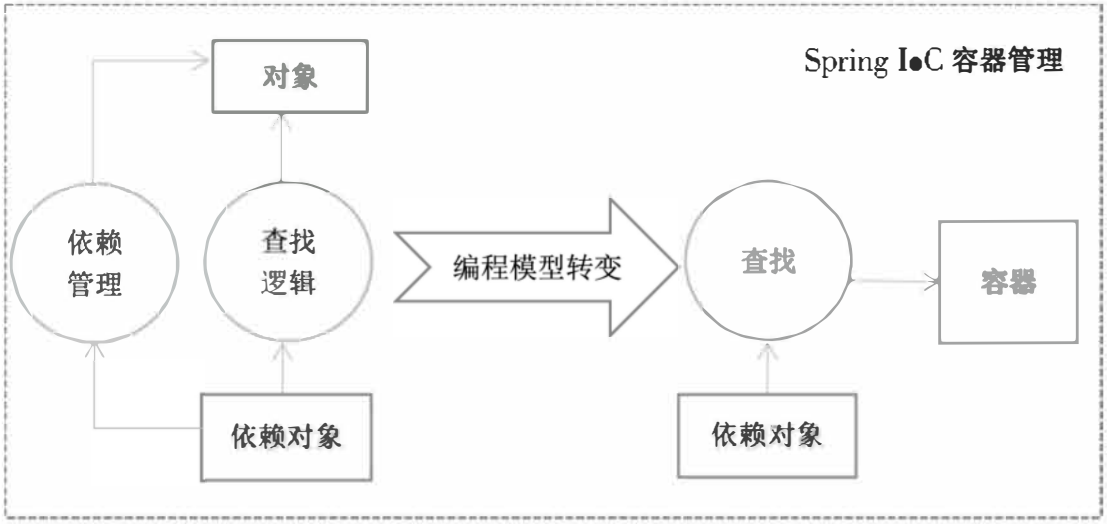

● Spring框架：Spring框架的核心就是，控制反转和面向切面（AOP）机制，同时它为开发者了提供众多组件，包括Web容器组件（Spring Web MVC）、数据接入组件（Spring DAO）、数据对象映射组件（Spring ORM）等。这些组件基于Spring Core的IoC容器开发，同时Spring框架可以配置管理所有轻量级的JavaBean组件和维护众多JavaBean组件之间的关系。简单地说，Spring为开发者提供了一个一站式的轻量级开发框架平台。

● Spring Boot：Spring Boot是一个微服务框架，以“Boot”命名，很好地说明这个框架的初衷——快速启动。Spring Boot从代码结构上来说包含了Spring框架，或者说是在Spring框架基础上做的一个扩展。它在延续Spring框架的核心思想和设计理念的基础上，简化了应用的开发和组件的集成难度。Spring Boot是为了简化Spring应用的创建、运行、调试、部署等特性而出现的，使用Spring Boot脚手架可以让微服务开发者做到专注于业务领域的开发，无须过多地关注底层技术实现细节。

● Spring中的IoC机制与JavaConfig的关系：我们知道，Spring IoC机制是Spring框架的核心，通过控制反转机制实现JavaBean组件和JavaBean组件依赖关系的管理。如果不使用IoC技术，开发者需要手动创建、查找、管理业务逻辑对象和依赖。

如果说“程序=算法+数据”，那么这里我们可以把这些JavaBean组件看作我们需要维护的数据，当数据（对象）规模膨胀时，将给我们的应用带来极大的耦合度和复杂度。而通过Spring IoC容器可以方便地管理我们的对象。

下图是Spring IoC容器给开发人员带来的编程模型的转变，它可以降低程序代码之间的耦合度，将耦合的对象依赖关系从代码中移除，通过将对象和依赖关系放在注解（或者XML配置文件）中，将程序对组件的控制权转交给IoC容器，进行统一管理。开发者只需要专注于业务的JavaBean组件的实现，查找逻辑和依赖逻辑全部由Spring IoC容器帮助打理。




在Spring 3.0之前，JavaBean组件一直是通过XML配置文件来配置管理的，Spring 3.0之后为我们提供了Java代码形式（JavaConfig）的配置功能。JavaConfig功能从Spring 3.0以后已经包含在了Spring的核心模块中（JavaConfig并非Spring Boot新特性），可以说JaveConfig就是Spring IoC容器的纯Java实现版本。在Spring Boot中，JavaConfig已经完全代替applicationContext.xml，实现了XML的零配置，如下所示是两种不同配置模式示例。

- 基于XML配置文件方式

```
<?XML version="1.0" encoding="UTF-8"?>
//省略
    <bean id="button" class="javax.swing.JButton">
        <constructor-arg value="Hello World" />
    </bean>
</XML>
```

- 基于JavaConfig方式

JavaConfig可以被看成一个XML文件，只不过是用Java代码编写的。

```
@Configuration
public class DemoConfiguration {
    @Bean
    public JButton button() {
        return new JButton("Hello World");
    }
}
```

**JavaConfig的优势**

上面我们介绍了Spring实现的JavaBean管理模式，主要有XML配置文件和JavaConfig两种方式，Spring Boot采用的JavaConfig主要有下面几个优点。

- 面向对象配置：由于配置被定义在JavaConfig中的类中，可以充分使用Java面向对象的功能，用户可以实现配置继承、配置重写等面向对象特性。
- 减少大量滥用XML：由于Spring把所有逻辑业务类都以XML配置文件的形式来表达Bean，造成XML文件充斥整个项目，带来了开发、维护的复杂性，开发人员需要频繁地在XML和Java语言之间来回切换。
- 类型安全和重构支持：因为注释在类源代码中，所以JavaConfig为应用提供了类型安全的方法来配置管理Spring容器，由于Java对泛型的支持，我们可以按照类型而不是名称检索JavaBean，这带来了更大的灵活性和重构支持。

本节我们带大家简单地回顾了Spring框架的整体架构，这些都是“开胃菜”，下面让我们看看Spring Boot是如何创建一个独立运行、生产级别的Spring应用的。

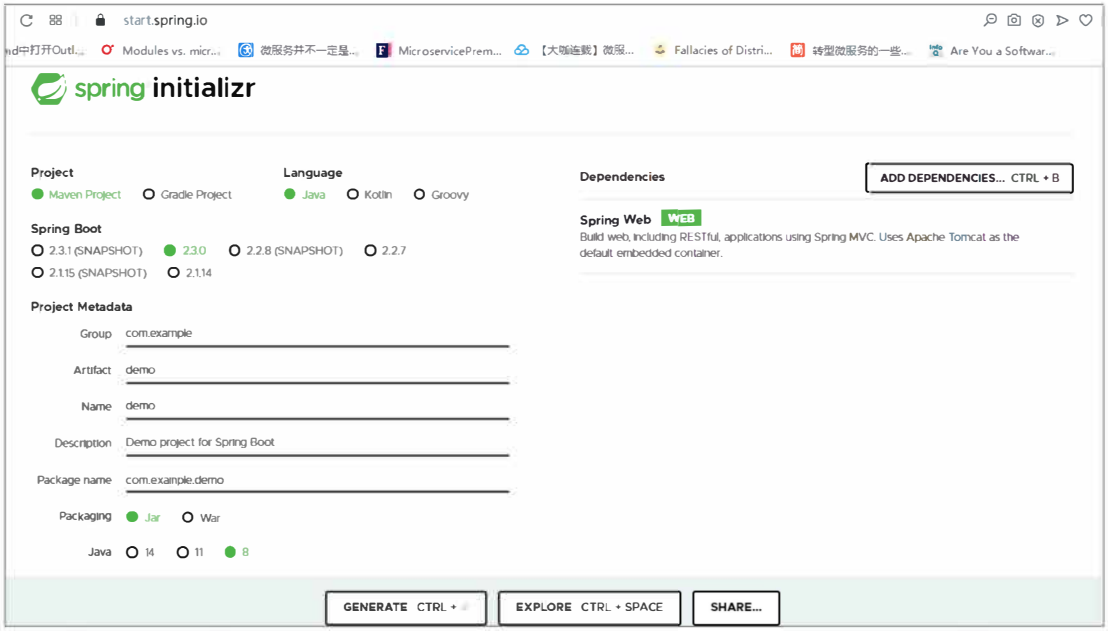
## 4.2.2 Spring Boot快速搭建

1.开发环境准备工作

在开始之前，我们需要搭建IDE（集成开发环境），目前流行的IDE有IntelliJ IDEA、Spring Tools、Visual Studio Code和Eclipse等。Java Development Kit（JDK）我们推荐使用OpenJDK 8或者OpenJDK 11。

2.使用Spring-Initializr快速构建工程

我们可以通过Spring官方提供的Spring Initializr [注] 来构建Spring Boot项目，它不仅完美支持IDEA和Eclipse，而且能自动生成启动类和单元测试代码，给开发人员带来了极大的便利。如下图所示是Spring官方的Spring Boot构建工程模板，你可以使用Maven或者Gradle进行初始化项目的构建。你需要填写Group、Artifact等工程元数据信息，最后单击GENERATE按钮生成Spring Boot模板工程。




## 3.静态工程模板

Spring Boot静态工程目录模板示例如下：

```
|+-mvnw
|+-mvnw.cmd
|+-pom.xml
|+-src
 |+-main
    |+-java
    |+-com
    +company
       +-project
        +-Application.java          ---- Spring Boot 启动类
        +-config                    ---- Spring Boot 启动配置类
        |  +- DeomConfig.java
        +-controller                ---- 前端控制 Web 接口层
        |  +- DemoController.java
        +-domain                    ---- 数据服务实体层
        |  +- Entity.java
        +-service                   ---- 数据服务接口层
        |  +- Service.java
            +-ServiceImplement.java ---- 数据服务接口实现类
        +-dao                       ---- 数据访问接口层
        |  +- DemoRepository.java
        |  +- DemoMapper.java
        +-Rest                      ---- HTTP 远程访问 Rest 层
        |  +- DemoRest.java
        +-api                       ---- HTTP 远程访问接口层
        |  +- Entity.java
```

## 4.增加启动代码，开始Spring Boot的启动流程

```
package com.xxxxx;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        LOGGER.info("DemoApplication started...");
        SpringApplication.run(DemoApplication.class， args);
    }
}
```

5.运行Spring Boot应用

```
./mvnw spring-boot:run
```

## 4.2.3 @SpringBootApplication注解详解

下面我们通过源码了解Spring Boot是如何工作的。首先我们看一下@SpringBootApplication注解，它是用来标注主程序的，表明这是一个Spring Boot应用，也是一个组合注解，主要由下面三个注解组成：

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

@SpringBootApplication注解定义如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
@Filter(type = FilterType.CUSTOM， classes = TypeExcludeFilter.class)，
@Filter(type = FilterType.CUSTOM， classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
    @AliasFor(annotation = EnableAutoConfiguration.class， attribute = "exclude")
    Class<?>[] exclude() default {};
    @AliasFor(annotation = EnableAutoConfiguration.class， attribute =
    "excludeName")
    String[] excludeName() default {};
    @AliasFor(annotation = ComponentScan.class， attribute = "basePackages")
    String[] scanBasePackages() default {};
    @AliasFor(annotation = ComponentScan.class， attribute =
    "basePackageClasses")
    Class<?>[] scanBasePackageClasses() default {};
}
```

**@SpringBootConfiguration注解解析**

@SpringBootConfiguration注解代码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}
```

@SpringBootConfiguration来源于@Configuration，二者的功能都是将当前类标注为配置类，并将类中以@Bean注解标记的方法的实例注入Spring容器。@Configuration是JavaConfig形式的Spring容器的配置类所使用的。

**@EnableAutoConfiguration注解解析**

@EnableAutoConfiguration注解代码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}
```

@EnableAutoConfiguration可以说是Spring Boot启动注解的主角，其中最关键的注解是@Import（AutoConfigurationImportSelector.class）。

借助AutoConfigurationImportSelector类，注解@EnableAutoConfiguration可以帮助Spring Boot应用将所有符合条件的@Configuration配置加载到当前Spring Boot创建并使用的Spring容器中。

> **注意：** 在Spring Boot 1.5以后，EnableAutoConfigurationImportSelector类已经被AutoConfigurationImportSelector类所取代。

下面我们看一下AutoConfigurationImportSelector类中的selectImports方法是如何实现自动加载配置类的，源码如下：

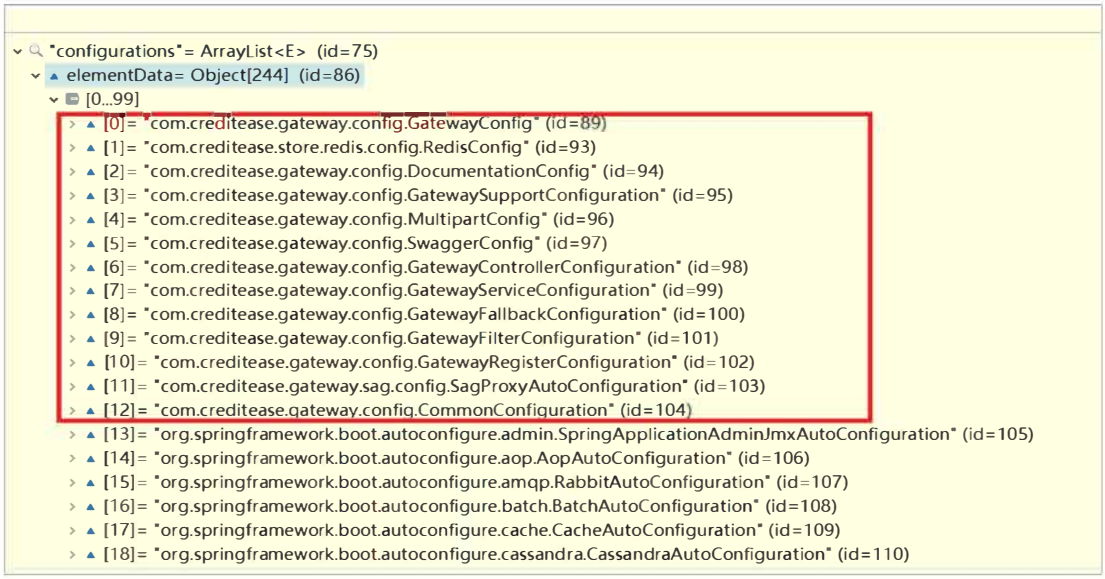
```
public class AutoConfigurationImportSelector
    implements DeferredImportSelector, BeanClassLoaderAware,
ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered
{
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata =
            AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoade;
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations =
            getCandidateConfigurations(annotationMetadata, attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = filter(configurations, autoConfigurationMetadata);
        fireAutoConfigurationImportEvents(configurations, exclusions);
        return configurations.toArray(new String[configurations.size()]);
    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}
```

getCandidateConfigurations方法如下：

```
protected List<String> getCandidateConfigurations(AnnotationMetadatametadata,
        AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories.
        If you "+ "are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

getCandidateConfigurations方法的主要逻辑是调用Spring框架提供的一个工具类SpringFactoriesLoader。SpringFactoriesLoader是Spring内部提供的一种约定俗成的加载配置类的方式，使用SpringFactoriesLoader可以从指定classpath下读取META-INF/spring.factories文件的配置，并返回一个字符串数组。通过这个方法，所有自动配置类都会自动加载到Spring容器中。

如下图所示是一个Spring Boot应用启动过程的内存快照，可以看到，在“配置列表对象”中除了Spring自带的配置类，还有第三方的自动配置类。我们可以根据SpringFactoriesLoader规定的协议自定义配置类。




上面框线标注的配置类对应下面的META-INF/spring.factories配置文件，这个Properties格式的文件中主键（Key）可以是接口、注解、抽象类的全名，值（Value）是以“，”分割的实现类，如下所示：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.creditease.gateway.config.GatewayConfig，\
com.creditease.store.redis.config.RedisConfig，\
com.creditease.gateway.config.DocumentationConfig，\
com.creditease.gateway.config.GatewaySupportConfiguration，\
com.creditease.gateway.config.MultipartConfig，\
com.creditease.gateway.config.SwaggerConfig，\
com.creditease.gateway.sag.config.SagProxyAutoConfiguration，\
com.creditease.gateway.config.CommonConfiguration
```

总结一下，@EnableAutoConfiguration的作用及SpringFactoriesLoader启动加载配置类流程如下：

（1）从classpath中搜索所有META-INF/spring.factories配置文件，然后将其中org.springframework.boot.autoconfigure.EnableAutoConfiguration的Key对应的配置项加载到Spring容器。

（2）@EnableAutoConfiguration可以排除配置选项，排除方式有两种，一种方式是在使用@SpringBootApplication注解时，使用exclude属性排除指定的类，代码如下：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
@SpringBootApplication(exclude={DataSourceAutoConfiguration.class,})
public class Application {
//省略
}
```

另外一种方式是：单独使用@EnableAutoConfiguration注解，其内部的关键代码实现如下：

```
@EnableAutoConfiguratio(exclude = {DataSourceAutoConfiguration.class})
public class Application {
    //省略
}
```

**@ComponentScan注解解析**

@ComponentScan注解代码如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
}
```

@ComponentScan注解本身是Spring框架加载Bean的主要组件，它并不是Spring Boot的新功能，这里不对@ComponentScan扫描和解析Bean的过程进行详细说明，感兴趣的读者可以自行查阅资料进行了解。

@ComponentScan注解的作用总结一句话就是：定义扫描路径，默认会扫描该类所在的包下所有符合条件的组件和Bean定义，最终将这些Bean加载到Spring容器中。下面是我们总结的@ComponentScan的主要使用方式：

- @ComponentScan注解默认会装配标识了@Component注解的类到Spring容器中。
- 通过basepackage可以指定扫描包的路径。
- 通过includeFilters将扫描路径下没有以上注解的类加入Spring容器。
- 通过excludeFilters过滤出不用加入Spring容器的类。

## 4.2.4 Spring Boot启动流程进阶

每一个Spring Boot程序都有一个主入口，这个主入口就是main方法，而main方法中都会调用SpringBootApplication.run方法，一个快速了解SpringBootApplication启动过程的好方法就是在run方法中打一个断点，然后通过Debug的模式启动工程，逐步跟踪了解Spring Boot源码是如何完成环境准备和启动加载Bean的。

查看SpringBootApplication.run方法的源码就可以发现Spring Boot的启动流程主要分为两个大的阶段：初始化SpringApplication和运行SpringApplication。而运行SringApplication的过程又可以细化为下面几个部分，后面我们会对启动的主要模块加以详解。

```
(步骤 1)初始化 SpringApplication
(步骤 2)运行 SpringApplication:
    (步骤 2.1)SpringApplicationRunListeners 应用启动监控模块
    (步骤 2.2)ConfigurableEnvironment 配置环境模块和监听
        (步骤 2.2.1)创建配置环境
        (步骤 2.2.2)加载属性配置文件
        (步骤 2.2.3)配置监听
    (步骤 2.3)ConfigurableApplicationContext 配置应用上下文
        (步骤 2.3.1)配置应用上下文对象
        (步骤 2.3.2)配置基本属性
        (步骤 2.3.3)刷新应用上下文
```

**初始化SpringApplication**

步骤1进行SpringApplication的初始化，配置基本的环境变量、资源、构造器、监听器。初始化阶段的主要作用是为运行SpringApplication对象实例启动做环境变量准备以及进行必要资源构造

器的初始化动作，代码如下：

```
public SpringApplication(ResourceLoader resourceLoader, Object... sources) {
    this.resourceLoader = resourceLoader;
    initialize(sources);
}
@SuppressWarnings({ "unchecked", "rawtypes" })
private void initialize(Object[] sources) {
    if (sources != null && sources.length > 0) {
    this.sources.addAll(Arrays.asList(sources));
    }
    this.WebEnvironment = deduceWebEnvironment();
    setInitializers((Collection)
    getSpringFactoriesInstances(ApplicationContextInitializer.class));
    setListeners((Collection)
    getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

SpringApplication构造方法的核心是this.initialize（sources）初始化方法，SpringApplication通过调用该方法完成初始化工作。deduceWebEnvironment方法用来判断当前应用的环境，该方法通过获取两个类来判断当前环境是否是Web环境。而getSpringFactoriesInstances方法主要用来从spring.factories文件中找出Key为ApplicationContextInitializer的类并实例化，然后调用setInitializers方法设置到SpringApplication的initializers属性中，找到它所有应用的初始化器。接着调用setListeners方法设置应用监听器，这个过程可以找到所有应用程序的监听器，最后找到应用启动主类名称。

**运行SpringApplication**

步骤2 Spring Boot正式地启动加载过程，包括启动流程监控模块、配置环境加载模块、ApplicationContext容器上下文环境加载模块。refreshContext方法刷新应用上下文并进行自动化配置模块加载，也就是上文提到的SpringFactoriesLoader根据指定classpath加载META-INF/spring.factories文件的配置，实现自动配置核心功能。运行SpringApplication的主要代码如下：

```
public ConfigurableApplicationContext run(String... args) {
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
            DefaultApplicationArguments(args);
        ConfigurableEnvironment environment = prepareEnvironment(
            listeners, applicationArguments);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        prepareContext(context, environment, listeners,
            applicationArguments, printedBanner);
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        listeners.finished(context, null);
        //省略
        return context;
    }
}
```

步骤 2.1

步骤 2.2

步骤 2.3

1.SpringApplicationRunListeners应用启动监控模块

应用启动监控模块对应上述步骤2.1，它创建了应用的监听器SpringApplicationRunListeners并开始监听，监听模块通过调用getSpringFactoriesInstances私有协议从META-INF/spring.factories文件中取得SpringApplicationRunListeners监听器实例。

当前的事件监听器SpringApplicationRunListeners中只有一个EventPublishingRunListener广播事件监听器，它的Starting方法会封装成SpringApplicationEvent事件广播出去，被SpringApplication中配置的listeners所监听。这一步骤执行完成后也会同时通知Spring Boot其他模块目前监听初始化已经完成，可以开始执行启动方案了。

2.ConfigurableEnvironment配置环境模块和监听

对应上述步骤2.2，下面是分解步骤说明。

（1）创建配置环境，对应上述步骤2.2.1，创建应用程序的环境信息。如果是Web程序，创建StandardServletEnvironment，否则创建StandardEnvironment。

（2）加载属性配置文件，对应上述步骤2.2.2，将配置环境加入监听器对象中（SpringApplicationRunListeners）。通过configurePropertySources方法设置properties配置文件，通过执行configureProfiles方法设置profiles。

（3）配置监听，对应上述步骤2.2.3，发布environmentPrepared事件，即调用ApplicationListener的onApplicationEvent事件，通知Spring Boot应用的environment已经准备完成。

3.ConfigurableApplicationContext配置应用上下文

对应上述步骤2.3，下面是分解步骤说明。

（1）配置Spring应用容器上下文对象，对应上述步骤2.3.1，它的作用是创建run方法的返回对象ConfigurableApplicationContext（应用配置上下文），此类主要继承了ApplicationContext、Lifecycle、Closeable接口，而ApplicationContex是Spring框架中负责Bean注入容器的主要载体，负责Bean加载、配置管理、维护Bean之间的依赖关系及Bean的生命周期管理。

（2）配置基本属性，对应上述步骤2.3.2，prepareContext方法将listeners、environment、banner、applicationArguments等重要组件与Spring容器上下文对象相关联。借助SpringFactoriesLoader查找可用的ApplicationContextInitializer，它的initialize方法会对创建好的ApplicationContext进行初始化，然后它会调用SpringApplicationRunListener的contextPrepared方法，此时Spring Boot应用的AppIcaionContext已经准备就绪，为刷新应用上下文准备好容器。

（3）刷新应用上下文，对应上述的步骤2.3.3，refreshContext（context）方法将通过工厂模式产生应用上下文环境中所需要的Bean。实现spring-boot-starter-*（mybatis、redis等）自动化配置的关键，包括spring.factories的加载、Bean的实例化等核心工作。最后SpringApplicationRunListener调用finished方法告诉Spring Boot应用程序容器已经完成ApplicationContext装载。

## 4.2.5 Spring Boot自动装配机制

Spring Boot的快速发展壮大，得益于“约定优于配置”的理念。Spring Boot自动装配流程中最核心的注解是@EnableAutoConfiguration，在上一节的启动流程中我们已经讲过，它可以借助SpringFactoriesLoader“私有协议特性”将标注了@Configuration的JavaConfig全部加载到Spring容器中，而如果是基于条件的装配及调整顺序的Bean装配，需要Spring Boot有额外的自动化装配机制。下面从@EnableAutoConfiguration开始进阶讲解，加深我们对Spring Boot自动装配机制的认识。

**基于条件的自动装配**

下面是@EnableAutoConfiguration注解，它同样是一个组合注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}
```

从源码可见，最关键的就是@Import（AutoConfigurationImportSelector.class）注解的实现。借助EnableAutoConfigurationImportSelector模块，@EnableAutoConfiguration可以帮助Spring Boot应用将所有符合条件的@Configuration配置都加载到当前的容器中。同时借助Spring框架原有的底层工具SpringFactoriesLoader（服务发现机制）和根据特定条件装备Bean的Condition xxx 条件注解实现智能的自动化配置工作。Bean的加载过滤过程主要是通过下面的方法实现的。

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
        AutoConfigurationMetadata autoConfigurationMetadata =
    AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations =
        getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
        //省略
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return StringUtils.toStringArray(configurations);
}
```

源码解析如下：

（1）执行AutoConfigurationMetadataLoader.loadMetadata（this.beanClassLoader）会加载META-INF/spring-autoconfigure-metadata.properties下的所有配置信息。

（2）执行getCandidateConfigurations（annotationMetadata，attributes）会加载所有包下 META-INF/spring.factories 的信息并组装成 Map，然后读取 Key 为org.springframework.boot.autoconfigure.EnableAutoConfiguration的数组，并将这个数组返回。

（3）执行getExclusions（annotationMetadata，attributes）会获取限制候选配置的所有排除项（找到不希望被自动装配的配置类）。

（4）执行checkExcludedClasses（configurations，exclusions）会对参数exclusions进行验证并去除多余的类，它对应@EnableAutoConfiguration注解中的exclusions属性。

（5）执行filter（configurations，autoConfigurationMetadata）会根据项目中配置的AutoConfiguration-ImportFilter类进行配置过滤。

通过查看源码，我们可以发现AutoConfigurationImportFilter是一个接口，OnClassCondition才是它的实现类，而OnClassCondition就是Spring Boot的Condition实现类。@ConditionalOnClass代码如下：

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(OnClassCondition.class)
public @interface ConditionalOnClass {
    String[] name() default {};
}
```

@ConditionalOnClass是基于@Conditional的组合注解，在上述的第（5）步中，Spring Boot可以通过这个注解实现按需加载，只有在@Configuration中符合条件的Class才会被加载进来。@Conditional注解本身是一个元注解，用来标注其他注解，如下所示：

```
@Conditional({myCondition.class})
```

通过利用@Conditional元注解，可以构造满足自己条件的组合条件注解，Spring Boot正是通过这样的方式实现了众多条件注解，实现了基于条件的Bean构造，还有Bean相互依赖情况下的顺序加载，它不需要再通过显性的基于XML文件的依赖文件进行构造。从上述讲解我们可以知道，Spring Boot结合Java元注解概念、Spring底层容器配置机制，以及使用类似Java SPI（Service Provider Interface）机制实现的私有配置加载协议，最终实现了“约定优于配置”。

在Spring Boot的Autoconfigure模块中，还包含了一批这样的组合注解，这些条件的限制在Spring Boot中以注解的形式体现，通常这些条件注解使用@Conditional来配合@Configuration和@Bean等注解来干预Bean的生成，常见的条件注解如下。

- @ConditionalOnBean：Spring容器中存在指定Bean时，实例化当前Bean。
- @ConditionalOnClass：Spring容器中存在指定Class时，实例化当前Bean。
- @ConditionalOnExpression：使用SpEL表达式作为判断条件，满足条件时，实例化当前Bean。
- @ConditionalOnJava：使用JVM版本作为判断条件来实例化当前Bean。
- @ConditionalOnJndi：在JNDI存在时查找指定的位置，满足条件时，实例化当前Bean。
- @ConditionalOnMissingBean：Spring容器中不存在指定Bean时，实例化当前Bean。
- @ConditionalOnMissingClass：Spring容器中不存在指定Class时，实例化当前Bean。
- @ConditionalOnNotWebApplication：当前应用不是Web项目时，实例化当前Bean。
- @ConditionalOnProperty：指定的属性是否有指定的值。
- @ConditionalOnResource：类路径是否有指定的值。
- @ConditionalOnSingleCandidate：指定Bean在Spring容器中只有一个。
- @ConditionalOnWebApplication：当前应用是Web项目时，则实例化当前Bean。

有了组合注解，开发人员从大量的XML和Properties中得到了解放，可以抛弃Spring传统的外部配置，使用Spring自动配置，spring-boot-autoconfigure依赖默认配置项，根据添加的依赖自动加载相关的配置属性并启动依赖。应用者只需要引入对应的jar包，Spring Boot就可以自动扫描和加载依赖信息。

**调整自动配置顺序**

在Spring Boot的Autoconfigure模块中还可以通过注解对配置和组件的加载顺序做出调整，从而可以让这些存在依赖关系的配置和组件顺利地在Spring容器中被构造出来。

- @AutoConfigureAfter是spring-boot-autoconfigure包下的注解，其作用是将一个配置类在另一个配置类之后加载。
- @AutoConfigureBefore是spring-boot-autoconfigure包下的注解，其作用是将一个配置类在另一个配置类之前加载。

例如，在加载ConfigurationB之后加载ConfigurationA：

- 实现ConfigurationA.class

```
@Configuration
@AutoConfigurationAfter(ConfigurationB.class)
public class ConfigurationA{
    Public ConfigurationA(){System.out.println(“Config A test”);}
}
```

- 实现ConfigurationB.class

```
@Configuration
public class ConfigurationB{
    Public ConfigurationB(){System.out.println(“Config B test”);}
}
```

- 创建配置META-INF/spring.factories文件

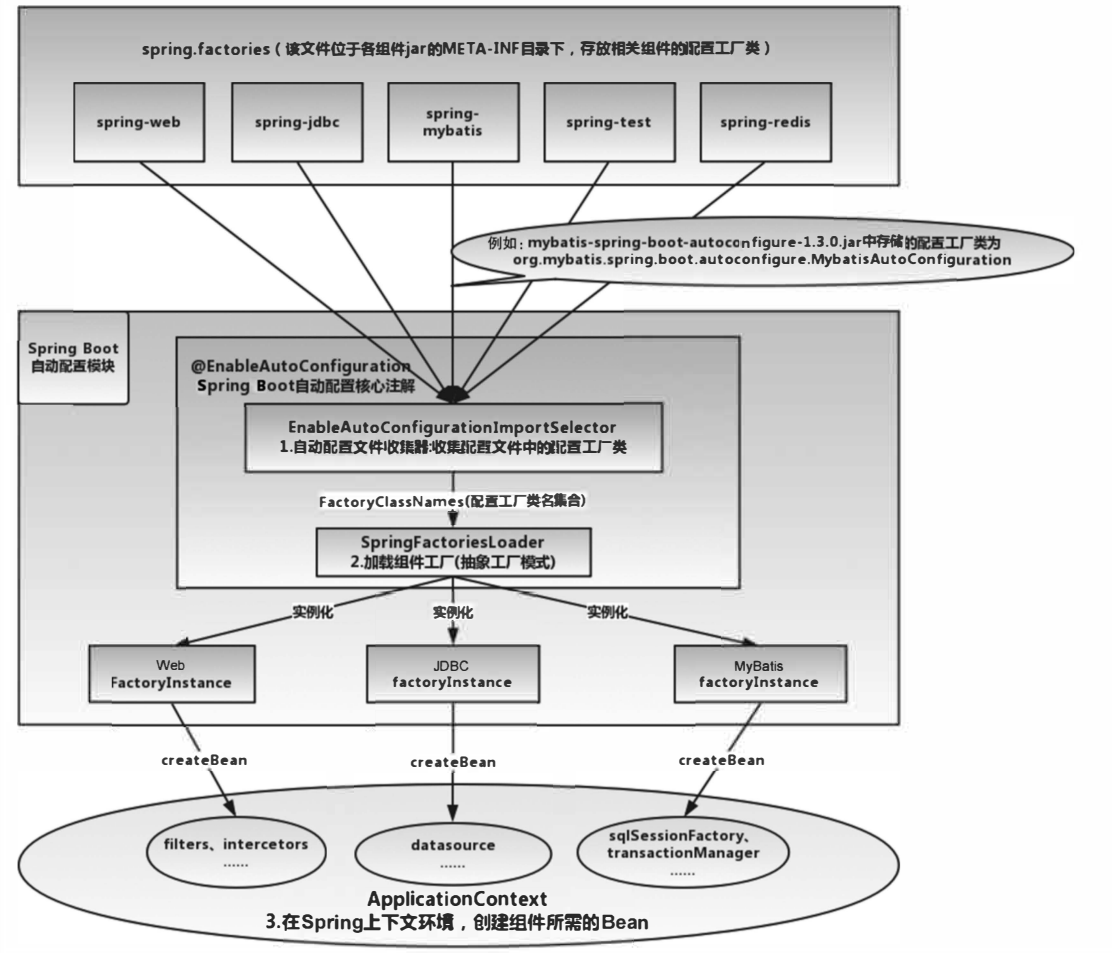
```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.test.configuration.ConfigurationA, \
com.test.configuration.ConfigurationB
```

通过上面的步骤，就可以实现自动调整Bean的加载顺序。另外，Spring为我们提供了@AutoConfigureOrder注解，也可以修改配置文件的加载顺序，示例代码如下：

```
@Configuration
@AutoConfigureOrder(2)
public class ConfigurationA{
    Public ConfigurationA(){System.out.println(“Config A test”);}
}
```

**自动化配置流程**

无论是应用初始化还是具体的执行过程，都要调用Spring Boot自动配置模块，下图有助于我们形象地理解自动配置流程。

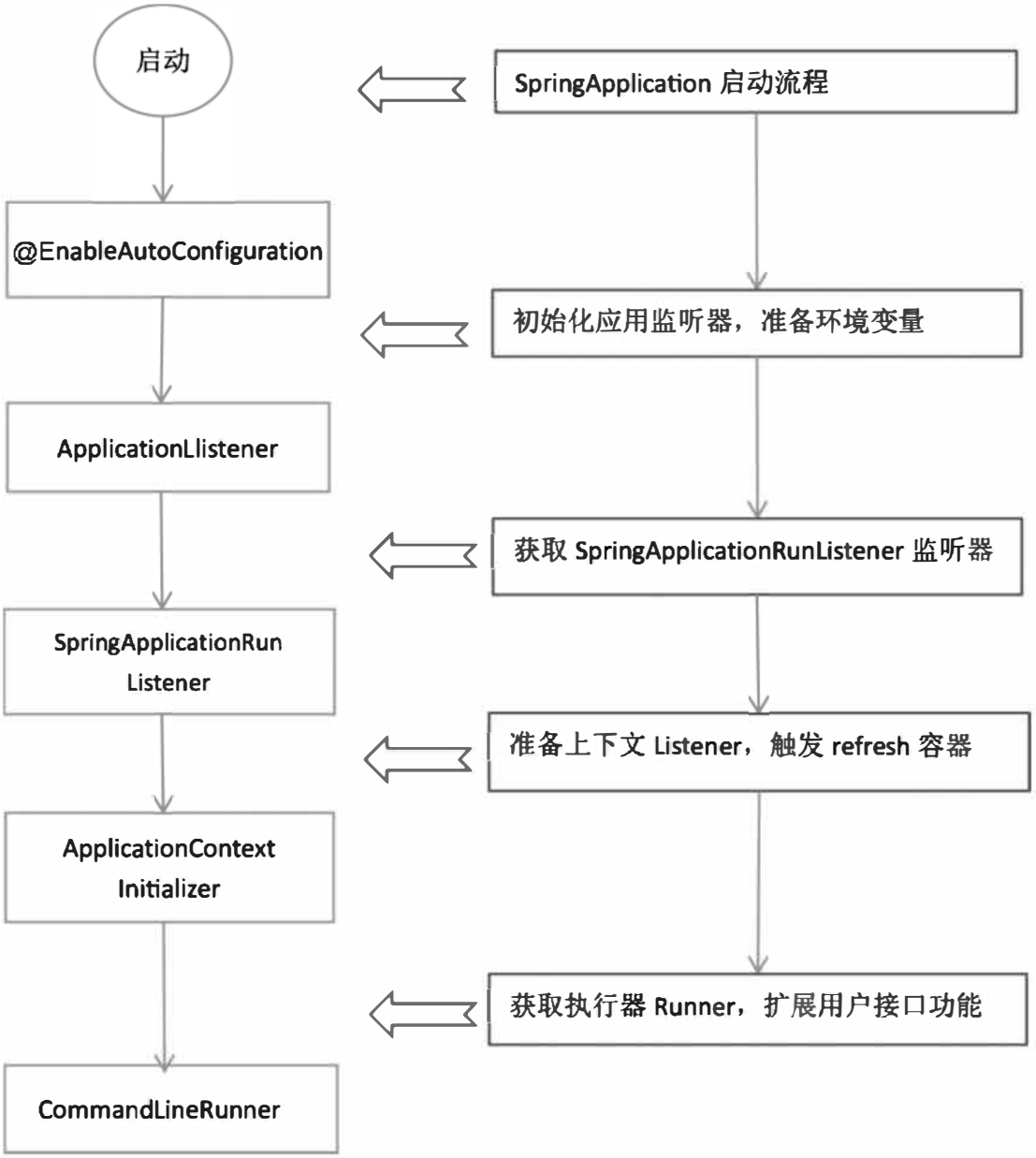



例如，mybatis-spring-boot-starter、spring-boot-starter-web等组件的META-INF下均含有spring.factories文件，在自动配置模块中，SpringFactoriesLoader收集到文件中的类全名并返回一个类全名的数组，返回的类全名通过反射被实例化，就形成了具体的工厂实例，最后工厂实例来生成组件所需要的Bean。

## 4.2.6 Spring Boot功能扩展点详解

在深入分析了Spring Boot的启动过程及其自动装配原理后，我们发现，Spring Boot的启动过程中使用了“模板”模式和“策略”模式，并且利用SpringFactoreisLoader的“私有协议”可以完成很多功能的扩展，满足启动的定制化。

我们将这些主要的扩展点结合源码加以总结，如下图所示。




### @EnableAutoConfiguration

从功能扩展点的角度，@EnableAutoConfiguration借助SpringFactoriesLoader可以将标注了@Configuration注解的JavaConfig类汇总并加载到最终的ApplicationContext中，使用条件注解可以在自动化配置过程中定制化Bean的加载过程：

```
@ConditionOnClass
@ConditionOnBean
@CondtionOnMissingClass
@CondtionOnMissingBean
@CondtionOnProperty
```

### ApplicationListener

ApplicationListener属于Spring框架，它是对Java中监听模式的一种实现方式，如果需要为Spring Boot应用添加我们自定义的ApplicationListener，那么有两种方式：

- 通过SpringApplication.addListeners（...）或者SpringApplication.setListener（...）方法添加一个或者多个自定义的ApplicationListener。

- 借助SpringFactoriesLoader机制，在Spring Boot项目自定义的META-INF/spring.factories文件中添加配置，以下是Spring Boot默认的ApplicationListener配置：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener, \
org.springframework.boot.builder.ParentContextCloserApplicationListener, \
org.springframework.boot.context.FileEncodingApplicationListener, \
org.springframework.boot.context.config.AnsiOutputApplicationListener, \
```

**SpringApplicationRunListener**

SpringApplicationRunListener的作用是，在整个启动流程中，作为监听者接收不同执行点的事件通知。没有特殊情况一般不需自定义的SpringApplicationRunListener。SpringApplicationRun-Listener源码如下：

```
public interface SpringApplicationRunListener {
    void starting();
    void environmentPrepared(ConfigurableEnvironment environment);
    void contextPrepared(ConfigurableApplicationContext context);
    void contextLoaded(ConfigurableApplicationContext context);
    void finished(ConfigurableApplicationContext context,  Throwable exception);
}
```

**ApplicationContextInitializer**

ApplicationContextInitializer也属于Spring框架，它的主要作用是，在ConfigurableApplicationContext类型（或者子类型）的ApplicationContext做刷新（refreshContext）之前，允许我们对ConfiurableApplicationContext的实例做进一步的设置和处理。

不过一般情况下我们不需要自定义一个ApplicationContextInitializer，Spring Boot框架默认也只有以下四个实现而已：

```
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer, \
org.springframework.boot.context.ContextIdApplicationContextInitializer, \
org.springframework.boot.context.config.DelegatingApplicationContextInitializer, \
org.springframework.boot.context.embedded.ServerPortInfoApplicationContextIniti
```

**CommandLineRunner**

CommandLineRunner并不是Spring框架原有的概念，它属于Spring Boot应用特定的回调扩展接口，源码如下：

```
public interface CommandLineRunner {
    /**
    * Callback used to run the bean.
    * @param args incoming main method arguments
    * @throws Exception on error
    */
    void run(String... args) throws Exception;
}
```

所有CommandLineRunner的执行时间点是在Spring Boot应用完全初始化之后（这里我们可以认为是Spring Boot应用启动类main方法执行完成之前的最后一步）。当前Spring Boot应用的ApplicationContext中的所有CommandLineRunner都会被加载并执行。

## 4.3 Spring Boot Starter技术

### 4.3.1 Spring Boot Starter概述

Spring Boot能够迅速地在微服务开发领域流行起来，并影响众多Spring和Java开发社区开发人员，可以说主要原因有两个。

- 一是Spring的约定优于配置的特性（Convention Over Configuration），这个特性的关键实现机制就是自动装配机制。同时这一特性很好地遵循了简约开发原则，它不仅减少了软件开发人员的开发工作量和需要做的决定数量，获得简单易用的收益，而且方便扩展又不失灵活性。从Spring到Spring Boot，从Ant到Maven，本质上都践行了约定优于配置的原则。
- 二是Spring Boot基于“Spring Boot Starter技术”，即开箱即用的自动配置模块。在传统Spring应用系统中，我们需要完成众多的烦琐配置和多个jar包的手动引入及代码的初始化工作，才能将所需要的模块引入工程中。而Spring Boot Starter的出现，简化了我们的配置，更重要的是将我们带入一种“可插拔”的编程模式。我们只需要在Maven中引入对应的现成Starter依赖，在代码中添加必要的注解，就可以获得开箱即用的对应功能。同时，我们可以结合Spring Boot的自动配置机制，实现自定义Starter组件，从而成为一个自包含的组件和模块，供第三方使用。

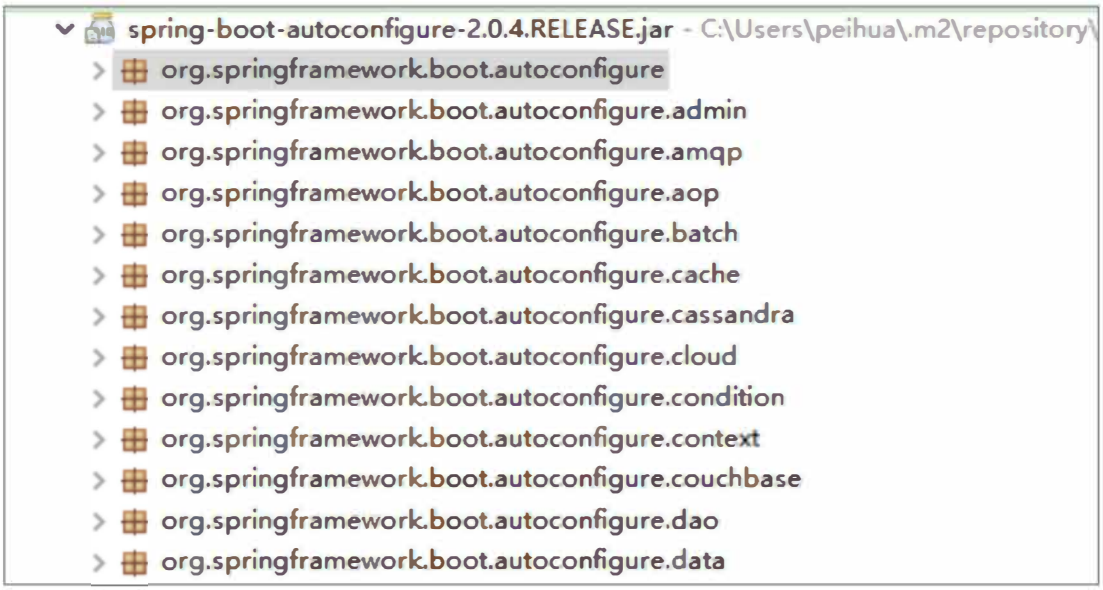
从Starter的命名方式我们可以区分出两类Starter。

- Spring官方Starter：命名应遵循spring-boot-starter-{name}的格式，如spring-boot-starter-web作为Spring Boot Web模块的官方artifactId。
- Spring非官方Starter：命名应遵循{name}-spring-boot-starter的格式，如mybatis-spring-boot-starter。本章中介绍的自定义Starter属于后者。

**Spring官方Starter**

对于Spring官方Starter，只需在pom.xml配置文件中增加对于Starter的依赖，这个Starter就能够通过代码配置上下文发现并将所需要jar包进行关联，在自动配置类中可以通过@ConditionalOnClass来决定是否实例化（ConditionalOnClass是指在classpath发现需要的依赖的类时实例化）。

所有的Starter其实都是要通过代码配置被上下文发现的，可以在spring-boot-autoconfigure-xxx.jar源码包中查看，例如下图所示，我们可以看到Spring Boot自带的Starter实现。




对于Spring Boot内置Web容器来说，只要通过@ConditionalOnClass发现了Tomcat这个类（配置了spring-boot-starter-web的Maven依赖），Spring Boot就会自动检查项目依赖并启动Tomcat服务，如下代码所示：

```
@Configuration
@EnableConfigurationProperties(ServerProperties.class)
public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {
    @ConditionalOnClass({ Tomcat.class,  UpgradeProtocol.class })
    public static class TomcatWebServerFactoryCustomizerConfiguration {
        @Bean
        public TomcatWebServerFactoryCustomizertomcatWebServerFactoryCustomizer(
         Environment environment,  ServerProperties serverProperties) {
          return new TomcatWebServerFactoryCustomizer(environment, serverProperties);
        }
    }
}
```

### Spring非官方Starter

原理上基本与官方Starter一致，需要在已经实现的artifactId上再封装一层，这一层只负责包含具体的实现类和配置类，而这个Starter的pom.xml文件相当于一个Facade门面，代码如下：

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.5</version>
</dependency>
```

进入pom.xml文件，可以发现自包含的依赖关系，代码如下：

```
<project >
<modelVersion>4.0.0</modelVersion>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot</artifactId>
    <version>1.3.5</version>
    <packaging>pom</packaging>
    ...
    <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>${mybatis.version}</version>
      </dependency>
      <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>${mybatis-spring.version}</version>
      </dependency>
      <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-autoconfigure</artifactId>
        <version>1.3.5</version>
      </dependency>
      <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.5</version>
      </dependency>
    //省略
</project>
```

在这个pom.xml文件中，我们发现了mybatis–spring–boot–autoconfigure自动配置类，通过它我们可以完成MyBatis配置JavaConfig的自动配置工作和MyBatis实例化配置。它的配置类代码如下：

```
@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class })
@ConditionalOnBean(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
public class MybatisAutoConfiguration {
   //省略
   public MybatisAutoConfiguration(MybatisProperties properties,
                            ObjectProvider<Interceptor[]> interceptorsProvider,
                            ResourceLoader resourceLoader,
                            ObjectProvider<DatabaseIdProvider>
                            databaseIdProvider,
                            ObjectProvider<List<ConfigurationCustomizer>>
                            configurationCustomizersProvider){
      this.properties = properties;
      this.interceptors = interceptorsProvider.getIfAvailable();
      this.resourceLoader = resourceLoader;
      this.databaseIdProvider = databaseIdProvider.getIfAvailable();
      this.configurationCustomizers = configurationCustomizersProvider.getIfAvailable();
    }
}
```

通过MybatisAutoConfiguration自动化配置类，就实现了MyBatis的配置在启动时被Spring Boot程序加载到Spring Boot的Factory工厂并实例化为Bean。

## 4.3.2 Spring Boot常用开箱即用Starter

开箱即用的Starter组件也遵循Spring约定优于配置的理念，针对企业日常应用开发场景，通过引入这些现成的Starter可以简化开发流程。

**spring-boot-starter-web快速开发**

除了开发少数的独立应用，大部分情况下，我们都使用Spring MVC组件开发企业Web应用，为了帮助我们快速搭建并开发一个Web项目，Spring Boot提供了spring-boot-starter-web自动配置模块，只要将spring-boot-starter-web加入项目的Maven依赖即可：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

在我们的工程中加入上面的Starter依赖后，就得到了一个可直接执行的Web应用环境，在当前项目下运行mvn spring-boot：run，可以直接启动一个使用了嵌入式Tomcat服务请求的Web应用服务。目前我们还没有提供任何Web请求的Controller，所以访问任何路径都会返回一个Spring Boot默认提供的错误页面，我们可以在当前项目下新建一个服务根路径作为Web请求的Controller实现，如下代码所示：

```
@RestController
public class startController {
    @RequestMapping("/")
    public String start() {
        return "demo，start";
    }
}
```

**spring-boot-starter-jdbc与数据访问**

为了使Spring Boot成为我们自动配置数据访问的基础设施，我们需要直接或者间接地依赖spring-jdbc，当spring-jdbc位于Spring Boot应用的classpath路径时，会触发数据访问相关的自动配置行为。最简单的做法就是把spring-boot-starter-jdbc添到应用的依赖文件中。默认情况下，如果我们没有配置任何DataSource，那么Spring Boot会为我们自动配置一个基于嵌入式数据库的DataSource，这种自动配置行为其实很适合于测试场景，但对实际的开发帮助不大，基本上我们会自己配置一个DataSource实例。下面我们将spring-boot-starter-jdbc加入项目的Maven依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!--引入 mysql 驱动包-->
<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <scope>runtime</scope>
</dependency>
```

如果我们的工程只依赖一个数据库，那么使用DataSource自动配置模块提供的参数是最方便的。在本书的7.1.4节中，我们会对“使用spring-boot-starter-jdbc访问MySQL”进行详细讲解。

**spring-boot-starter-aop及其使用场景**

在Java语言中，AOP（Aspect Oriented Programming，面向切面编程）通过提供另一种思考程序结构的方式来补充OOP（Object Oriented Programming，面向对象编程）。OOP中模块化的关键单元是类。而在AOP中，模块化单元由“切面”组成。通过预编译方式和运行期动态代理实现程序功能的统一维护。AOP是Spring框架的一个重要特色，它可对既有程序定义一个切入点（Pointcut），然后在切入点前后切入不同的执行任务。常见使用场景有：打开/关闭数据库连接、打开/关闭事务、记录日志等。基于AOP不会破坏原来的程序逻辑，因此它可以很好地对业务逻辑的各个部分进行抽离，从而使得业务逻辑各个部分的耦合度降低，提高程序的复用性，同时提高开发效率。Spring Boot对AOP提供了Starter组件支持，Maven依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

spring-boot-starter-aop主要由两部分组成：

- org.springframework.boot.autoconfigure.aop.AopAutoConfiguration：提供配置类。
- 通知：配置类中涉及的通知类型有前置通知、后置最终通知、后置返回通知、后置异常通知和环绕通知。

下面是一个实现了Web Controller调用的日志切面打印Web请求参数及相应结果的实例。

```
@Aspect
@Component
public class WebLogAspect {
    private Logger logger = Logger.getLogger(getClass());
    @Pointcut("execution(* com.test.demo.Web.*.*(..))")
    public void excudeService(){}
    @Around(value = "excudeService()")
    public Object doAround(ProceedingJoinPoint pjp) throws Throwable {
     Object resultObj = null;
     try {
        RequestAttributes requestAttributes = RequestContextHolder.getRequestAttributes();
        ServletRequestAttributes sra = (ServletRequestAttributes)requestAttributes;
        HttpServletRequest request = sra.getRequest();
        resultObj = pjp.proceed();
        int status = getProcessResult(resultObj);
        logger.info("RESPONCE : "+ status );
    }
}
```

注解解释如下：

- 使用@Aspect注解将一个Java类定义为切面类。
- 使用@Pointcut定义一个切入点完成切面功能，根据对应参数执行不同的逻辑：
  - 任意公共方法的执行：

```
execution(public * *(..))
```

○ 任何一个以“set”开始的方法的执行：

```
execution(* set*(..))
```

○ AccountService接口的任意方法的执行：

```
execution(* com.xyz.service.AccountService.*(..))
```

○ 定义在service包里的任意方法的执行：

```
execution(* com.xyz.service.*.*(..))
```

○ 定义在service包和所有子包里的任意类的任意方法的执行：

```
execution(* com.xyz.service..*.*(..))
```

○ 定义在pointcutexp包和所有子包里的JoinPointObjP2类的任意方法的执行：

```
execution(* com.test.spring.aop.pointcutexp..JoinPointObjP2.*(..))")
```

**注意：** 根据需要，可在切入点的不同位置切入内容。

- 使用@Before在切入点开始处切入内容。
- 使用@After在切入点结尾处切入内容。
- 使用@AfterReturning在切入点return内容之后切入内容。
- 使用@Around在切入点前后切入内容，并自己控制何时执行切入点自身的内容。
- 使用@AfterThrowing处理当切入内容部分抛出异常之后的逻辑。

**spring-boot-starter-logging日志功能**

常见的日志系统大致有Java.util.logging、Log4J、Commons-logging等，spring-boot-starter-logging是Spring Boot为日志功能提供的一种默认实现。Maven依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
 </dependency>
```

Spring Boot能够使用Logback、Log4J2、java util logging作为日志记录工具，默认使用Logback。日志默认输出到控制台，也能输出到文件中。如果想改变Spring Boot提供的应用日志设定，可以：

- 遵循Logback的约定，在classpath中使用自己定制的logback.XML配置文件。
- 在文件系统的任意一个位置提供自己的logback.xml配置文件，然后通过logging.config配置项指向这个配置文件，再引用它。

**spring-boot-starter-security与应用安全**

spring-boot-starter-security主要面向的是Web应用开发，对应的Maven依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

spring-boot-starter-security默认会提供一个基于HTTP Basic认证的安全防护策略，默认用户为user，访问密码则在当前Web应用启动的时候打印到控制台。要想定制，则在配置文件中进行配置。

## 4.3.3 Spring Boot生产就绪与环境配置

Spring Boot自带的spring-boot-actuator模块提供的生产就绪（production-ready）特性与运行状况指标检查功能，可以帮助你深入掌握运行中的Spring Boot应用程序，一探Spring Boot程序的内部信息。

**Actuator**

Actuator指用于移动或控制某物的机械装置的制造业术语，Actuator可以从一个小的变化产生大量的运动。

要将Actuator添加到基于Maven的项目中开启Spring Boot的生产就绪特性，请加载以下依赖项：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

spring-boot-actuator自动配置模块默认为我们提供了很多Endpoint，根据Spring的官方定义，Endpoint的解释如下。

**Endpoint**

Endpoint是执行器端点，可用于监控应用及与应用进行交互，Spring Boot包含很多内置的端点，你也可以自己添加。例如，health端点提供了应用的基本健康信息。

Endpoint分成两类：原生端点和自定义端点。自定义端点主要是指扩展性端点，用户可以根据自己的实际应用，定义一些自己比较关心的指标，在运行期进行监控。如下表所示是Endpoint使用ID标识的方式定义的可用监控对象（原生端点）。

| ID | 描述 | 默认是否启用 |
| --- | --- | --- |
| auditevents | 暴露当前应用程序的审计事件信息 | 是 |
| beans | 暴露一个应用中所有Spring Bean的完整列表 | 是 |
| conditions | 显示配置类和自动配置类（configuration和auto-configuration） | 显示classes的状态及它们被应用或未被应用的原因 |
| configprops | 显示所有@ConfigurationProperties的集合列表 | 是 |
| env | 暴露来自Spring的ConfigurableEnvironment的属性 | 是 |
| flyway | 如果有，则显示数据库迁移路径 | 是 |
| health | 显示应用的健康信息（当使用一个未认证连接访问时，显示简单信息） | 如果使用认证连接访问，则显示全部信息详情 |
| info | 显示任意的应用信息 | 是 |
| liquibase | 如果有，则显示任何Liquibase数据库迁移路径 | 是 |
| metrics | 显示当前应用的metrics信息 | 是 |
| mappings | 显示所有@RequestMapping路径的集合列表 | 是 |
| scheduledtasks | 显示应用程序中的计划任务 | 是 |
| sessions | 允许从Spring会话支持的会话存储中检索和删除 | 用户会话。使用Spring Session支持反应性Web应用程序时不可用 |
| shutdown | 应用可以以优雅的方式关闭 | 否 |
| threaddump | 执行一个线程dump | 是 |

如果你的应用是一个Web应用（Spring MVC、Spring WebFlux或Jersey），你还可以使用如下表所示的Endpoint。

| ID | 描述 | 默认是否启用 |
| --- | --- | --- |
| heapdump | 返回一个GZip压缩的hprof堆dump文件 | 是 |
| jolokia | 通过HTTP暴露JMX Bean（当Jolokia在类路径上时，WebFlux不可用） | 是 |
| logfile | 返回日志文件内容（如果设置了logging.file或logging.path属性），支持使用HTTP Range头接收日志文件内容的部分信息 | 是 |
| Prometheus | 以可以被Prometheus服务器抓取的格式显示metrics信息 | 是 |

Actuator的原生API有很多，大体上可以细分为以下三类。

- 应用配置类：应用配置、环境变量、自动化配置等。

```
http://localhost:8080/actuator/conditions
http://localhost:8080/actuator/beans
http://localhost:8080/actuator/env
http://localhost:8080/actuator/info
http://localhost:8080/actuator/mappings
```

- 度量指标类：运行时监控到的指标，如内存、线程池、HTTP统计信息等。

```
http://localhost:8080/actuator/health
http://localhost:8080/actuator/auditevents
http://localhost:8080/actuator/loggers
http://localhost:8080/actuator/metrics/jvm.memory.max
http://localhost:8080/actuator/metrics
http://localhost:8080/actuator/heapdump
```

- 操作控制类：如关闭应用等操作类。

```
http://localhost:8080/actuator/shutdown
```

**启用/禁止端点规则**

- 默认情况下，除shutdown外的所有端点均已启用。要启用单个端点，可使用management.endpoint.<id>.enabled属性。以下示例启用shutdown端点：

```
management.endpoint.shutdown.enabled=true
```

- 可以通过management.endpoints.enabled-by-default来修改端点的默认配置，以下示例启用info端点并禁用所有其他端点：

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

- 禁用的端点将从应用程序上下文中完全被删除。如果只想更改端点公开（对外暴露）特性，可使用include和exclude属性，详情见下表。

| 属性值 | 默认 |
| --- | --- |
| management.endpoints.jmx.exposure.exclude | / |
| management.endpoints.jmx.exposure.include | * |
| management.endpoints.Web.exposure.exclude | / |
| management.endpoints.Web.exposure.include | info， health |

**说明：** include属性列出了公开的端点的ID，exclude属性列出了不应该公开的端点的ID。exclude属性优先于include属性，意思是指同一端点ID同时出现在include属性表和exclude属性表时，exclude属性优先于include属性，即此端点不被暴露。

**Endpoint的两种主要访问方式**

要实现端点的访问，Spring Boot为我们提供了两种方式。

- **基于JMX的监控**

Java管理扩展（JMX）提供了一种监视和管理应用程序的标准机制，默认情况下，Spring Boot将管理端点公开为org.springframework.boot域中的JMX Mbean。Actuator通过EndpointMBean-ExportAutoConfiguration将所有Endpoint实例以JMX Mbean形式对外监控暴露，并默认注册在org.springframework.boot域下，可以使用Jconsole命令启动Java管理及监控控制台查看端点信息。

也可以使用使用Jolokia（基于JSR-160规范实现的MBean服务）通过HTTP实现JMX远程管理。Jolokia是一个JMX-HTTP桥，它提供了一种访问JMX Beans的替代方法。想要使用Jolokia，只需添加org.jolokia：jolokia-core的依赖。例如，使用Maven添加以下配置，然后在HTTP管理服务器上可以通过/jolokia访问Jolokia。

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
```

另外，如果想要禁用JMX端点，可以使用下面的配置方式：

```
endpoints.jmx.enabled=false
```

● **基于HTTP的监控**

如果你正在开发一个Web应用程序，Actuator会自动配置通过HTTP公开的所有已启用的端点，并通过以“management.”为前缀的配置项对端点进行开放。因为HTTP是标准的协议，对于跨语言、跨平台访问有天然的优势，使用HTTP的方式暴露端点信息有利于与其他监控平台和系统进行对接。

Spring Boot执行器自动将所有启用的端点通过HTTP暴露出去。默认约定使用端点的ID作为URL路径，例如，health暴露为/health。如果不想通过HTTP暴露端点，可以将管理端口设置如下：

```
management.port=-1
```

**保护HTTP端点**

在配置文件中设置management.security.enabled： false，这样所有的用户都可以用Actuator。但是这样的方式可能会暴露服务的敏感信息，并且在默认情况下，Actuator端点暴露在服务于常规HTTP的同一个端口上。如果你的应用程序是公开部署的，你可能希望添加Spring Security来进行用户身份验证。当添加Spring Security时，默认的“basic”身份验证将被启用。

使用HTTP暴露端点的方式与使用任何敏感网址一样，如果你希望为HTTP端点配置自定义安全性，比方说只允许具有特定角色的用户访问它们，Spring Boot提供了一些方便的RequestMatcher对象，可以与Spring Security结合使用，具体方式如下。

1.引入Security的Maven依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2.在配置文件中修改application.yml

```
management
    context-path: /manageActuator
    security
        enabled: false
        roles: SUPERUSER
        user.name: admin
        user.password: admin
```

3.定制Spring Security

我们可以在Security中赋予用户不同的权限，Actuator使用这种权限管理来控制用户的行为。Security可以拦截特定的请求：

```
@Configuration
@EnableWebSecurity
public class ActuatorSecurityConfig extends WebSecurityConfigurerAdapter{
    @Autowired
    Environment env;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        String contextPath = env.getProperty("management.context-path");
        if(StringUtils.isEmpty(contextPath)) {
            contextPath = "";
        }
        http.csrf().disable();
        http.authorizeRequests()
                .antMatchers("/**"+contextPath+"/**").authenticated()
                .anyRequest().permitAll()
                .and().httpBasic();
    }
}
```

### Actuator的版本差异

**注意：** Spring Boot 2.0以后对Actuator做了很大的改动，主要变化如下：

- 2.X比1.X多了一个根路径：/actuator。
- 2.X的属性发生了变化，如下表所示。

| 1.X的属性 | 2.X的属性 |
| --- | --- |
| endpoints.<id >.* | management.endpoint.<id>.* |
| endpoints.cors.* | management.endpoints.Web.cors.* |
| endpoints.jmx.* | management.endpoints.jmx.* |
| management.address | management.server.address |
| management.context-path | management.server.servlet.context-path |
| management.ssl.* | management.server.ssl.* |
| management.port | management.server.port |

- 部分端点路径发生了变更：

```
/autoconfig -> 更名为 /conditions
/docs -> 被废弃
/trace -> 更名为 /httptrace
/dump -> 更名为 /threaddump
```

### 自定义健康检查器

在介绍自定义健康检查器前，我们先看一下Spring Boot定义的一套健康检查框架，后面我们根据整个框架定制一个健康检查器。对于健康检查器来说，最重要的接口就是HealthIndicator，代码如下：

```
public interface HealthIndicator {
```

```
    /**
    * Return an indication of health.
    * @return the health for
    */
    Health health();
}
```

抽象类AbstractHealthIndicator是健康检查类的骨架，实现了health方法，同时声明了doHealthCheck方法。Spring Boot的健康信息都从ApplicationContext中加载各种HealthIndicator实现，主要实现流程如下：

- 实例化Health$Builder。
- 调用抽象方法doHealthCheck进行状态的检查，出现异常则状态变为Status.DOWN。
- 下面是HealthCheck类的具体业务实现类示例：

```
@Controller
public class HealthCheck{
    public static boolean flag = false;
    @Autowired
    private DBRepository dataRepository;
    @RequestMapping(value="/checkHealthStatus", method =
        RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public String checkHealthStatus(){
        try{
            int result = dataRepository.checkDbStatus();
            this.flag = true;
        }catch (Exception e){
            this.flag = false;
        }
        return "当前数据库的状态:"+flag;
    }
}
```

你可以通过management.health.defaults.enabled配置项将健康检查项全部禁用，也可以通过management.health.xxxx.enabled将其中任意一个禁用。如果我们需要提供自定义的健康检查信息状态，可以通过HealthIndicator的接口来实现，并将该实现类注册为JavaBean。你需要实现其中的health方法，并返回自定义的健康状态响应信息，该响应信息应该包括一个状态码和要展示的详细信息。例如，下面就是一个自定义的HealthIndicator的实现类：

```
@Configuration
public class MyHealthIndicator implements HealthIndicator{
    @Override
    public Health health() {
        if(HealthCheck.flag){  //自定义逻辑，判断如果应用是健康的，则为 UP，否则为 DOWN
            return new Health.Builder(Status.UP).build();
        }else{
            return new Health.Builder(Status.DOWN).build();
        }
    }
}
```

## 4.3.4 Spring Boot安全管理

Spring Security是一款基于Spring的安全框架，主要包含认证和鉴权两大安全模块。Spring Security能够为基于Spring的企业应用系统提供声明式的安全访问控制解决方案。它提供了一组可

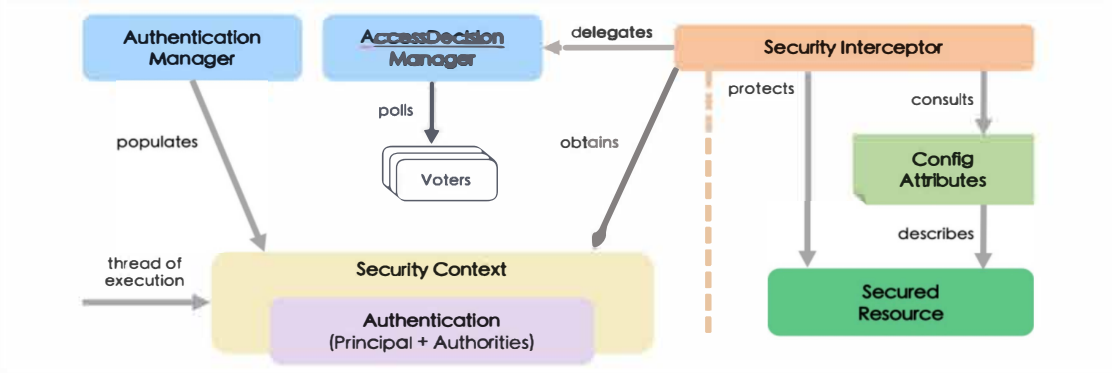
以在Spring应用上下文中配置的Bean，充分利用了Spring IoC（控制反转）、DI（依赖注入）和AOP（面向切面编程）。Spring Security本身比较复杂，其中包含众多子项目，如Spring Security OAuth、Spring Security JWT、Spring Security CAS等，本节将对Spring Security的核心模块及架构原理进行介绍。

### Spring Security核心模块

权限系统一般包含两大核心模块：Authentication（认证）和Authorization（鉴权）。

- Authentication模块负责验证用户身份的合法性，生成认证令牌，并保存到服务端会话中（如TLS）。
- Authorization模块负责从服务端会话中获取用户身份信息，与访问的资源进行权限比对。

官方给出的Spring Security的核心架构如下图所示。




核心架构解读如下。

- Authentication Manager：负责认证管理，解析用户登录信息（封装在Authentication模块中），读取用户、角色、权限信息并进行认证，认证结果被回填到Authentication，保存在Security Context中。
- AccessDecision Manager：负责投票表决，通过投票器的结果实现一票通过（默认）、多票通过、一票否决策略。
- Security Interceptor：负责权限拦截，包括Web URL拦截和方法调用拦截。通过Config Attributes获取资源的描述信息，借助AccessDecision Manager进行鉴权拦截。
- Security Context：安全上下文，保存认证结果。提供了全局上下文、线程继承上下文、线程独立上下文（默认）三种策略。
- Authentication：认证信息，保存用户的身份标识、权限列表、证书、认证通过标记等信息。
- Secured Resource：被安全管控的资源，如Web URL、用户、角色、自定义领域对象等。
- Config Attributes：资源属性配置，描述安全管控资源的信息，为Security Interceptor提供拦截逻辑的输入。

Spring Boot进行集成，需要引入Maven依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

在Spring Security的框架设计中，关键是AbstractSecurityInterceptor类，它是一个抽象类，安全认证流程的具体实现逻辑都贯穿在AbstractSecurityInterceptor类中，可以把AbstractSecurityInterceptor当成为一个模板类，它的实际执行流程都由具体子类所实现，每种受保护资源或者对象都由AbstractSecurityInterceptor的实现类拦截实现。Spring Security提供了两个具体实现类，Method-SecurityInterceptor用于受保护的方法，FilterSecurityInterceptor用于受保护的Web请求：

- public class FilterSecurityInterceptor extends AbstractSecurityInterceptor implements Filter{...}
- public class MethodSecurityInterceptor extends AbstractSecurityInterceptor implements Method-Interceptor{...}

上述两个继承AbstractSecurityInterceptor的实现类具有一致的执行逻辑：

（1）将正在请求调用的受保护对象传递给beforeInvocation方法进行权限鉴定。

（2）如果权限鉴定失败，则直接抛出异常。

（3）如果鉴定成功，则将尝试调用受保护对象，调用完成后，不管是成功调用，还是抛出异常，都将执行finallyInvocation方法。

（4）如果在调用受保护对象后没有抛出异常，则调用afterInvocation方法。

**Spring Security核心源码**

Spring Security默认使用的过滤器是FilterSecurityInterceptor，它的认证及鉴权流程和主要源码如下：

```
public class FilterSecurityInterceptor extends AbstractSecurityInterceptor implements
  Filter {
    private static final String FILTER_APPLIED =
        "__spring_security_filterSecurityInterceptor_filterApplied";
    private FilterInvocationSecurityMetadataSource securityMetadataSource;
    private boolean observeOncePerRequest = true;
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException
    {
        FilterInvocation fi = new FilterInvocation(request, response, chain);
        invoke(fi);
    }
    public void invoke(FilterInvocation fi) throws IOException, ServletException {
    if ((fi.getRequest() != null)&& (fi.getRequest().getAttribute(FILTER_APPLIED) != null)
            && observeOncePerRequest) {
        fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
    }
    else {
        if (fi.getRequest() != null && observeOncePerRequest) {
            fi.getRequest().setAttribute(FILTER_APPLIED, Boolean.TRUE);
        }
        InterceptorStatusToken token = super.beforeInvocation(fi);
        try {
            fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
        }
        finally {
            super.finallyInvocation(token);
        }
        super.afterInvocation(token, null);
    }
  }
}
```

从源码来看，AuthenticationManager负责制定规则，AbstractSecurityInterceptor负责执行。并且，Spring Security的Web安全方案基于Java的Servlet规范进行构建，所以如果你的开发框架是脱离Servlet规范实现的Web框架，则无法使用Spring Security提供的默认Web安全方案。在Servlet规范中，实现关卡功能的特性就是Filter组件，Spring框架使用将GenericFilterBean注入Spring容器的方式来让Filter可以享受依赖注入的好处。

WebSecurity会初始化FilterChainProxy，它通过扩展GenericFilterBean间接实现了Filter接口，同时持有一组SecurityFilterChain，真正执行防护任务的SecurityFilterChain中定义的一系列Filter的代码如下：

```
public interface SecurityFilterChain {
    boolean matches(HttpServletRequest request);
    List<Filter> getFilters();
}
```

Spring Security为SecurityFilterChain中的Filter设定了优先级顺序。下面的代码是Security的Filter实现逻辑。

```
final class FilterComparator implements Comparator<Filter>, Serializable {
    private static final int INITIAL_ORDER = 100;
    private static final int ORDER_STEP = 100;
    private final Map<String, Integer> filterToOrder = new HashMap<>();
    FilterComparator() {
    Step order = new Step(INITIAL_ORDER, ORDER_STEP);
    put(ChannelProcessingFilter.class, order.next());
    put(ConcurrentSessionFilter.class, order.next());
    put(WebAsyncManagerIntegrationFilter.class, order.next());
    put(SecurityContextPersistenceFilter.class, order.next());
    put(HeaderWriterFilter.class, order.next());
    put(CorsFilter.class, order.next());
    put(CsrfFilter.class, order.next());
    put(LogoutFilter.class, order.next());
    filterToOrder.put(
      "org.springframework.security.oauth2.client.Web
        .OAuth2AuthorizationRequestRedirectFilter", order.next());
    filterToOrder.put(
        "org.springframework.security.saml2.provider.service.servlet
        .filter.Saml2WebSsoAuthenticationRequestFilter", order.next());
    put(X509AuthenticationFilter.class, order.next());
    put(AbstractPreAuthenticatedProcessingFilter.class, order.next());
    filterToOrder.put("org.springframework.security.cas.Web.CasAuthenticationFilter",
        rder.next());
    filterToOrder.put(
      "org.springframework.security.oauth2.client.Web.OAuth2LoginAuthenticationFilter",
      order.next());
    //省略
    filterToOrder.put(
        "org.springframework.security.oauth2.client.Web
        .OAuth2AuthorizationCodeGrantFilter", order.next());
    put(SessionManagementFilter.class, order.next());
    put(ExceptionTranslationFilter.class, order.next());
    put(FilterSecurityInterceptor.class, order.next());
    put(SwitchUserFilter.class, order.next());
    }
    //省略
}
```

虽然Filter很多，但可以简单划分为几类，除个别Filter在每个SecurityFilterChain中都有，其他可以根据需要选用：

- 信道与状态管理类，比如ChannelProcessingFilter用于处理HTTP或者HTTPS之间的切换，而SecurityContextPersistenceFilter用于重建或者销毁必要的SecurityContext状态。
- 常见Web安全防护类，比如CsrfFilter。
- 认证和授权类，比如BasicAuthenticationFilter、CasAuthenticationFilter等。

**HttpSecurity**

Spring通过HttpSecurity完成定制化Filter的加载，拦截顺序及Filter的初始化存在于WebSecurityConfigurerAdapter的初始化方法init中，通过getHttp方法可以获得HttpSecurity的对象。下面是WebSecurityConfigurerAdapter的初始化方法：

```
public void init(final WebSecurity Web) throws Exception {
    final HttpSecurity http = getHttp();
    Web.addSecurityFilterChainBuilder(http).postBuildAction(new Runnable() {
        public void run() {
            FilterSecurityInterceptor securityInterceptor = http
                    .getSharedObject(FilterSecurityInterceptor.class);
            Web.securityInterceptor(securityInterceptor);
        }
    });
}
```

这个方法先构建HttpSecurity对象，然后通过WebSecurity对象的addSecurityFilterChainBuilder方法添加到securityFilterChainBuilders的List中，最后完成组件过滤器链。我们可以通过返回的HttpSecurity对象重写addFilter方法，加入定制化的Filter对象。

**定制实现spring-boot-starter-security**

我们可以通过给出一个继承了WebSecurityConfigurerAdapter的JavaConfig配置类的行为，进行更深一级的对spring-boot-starter-security组件的定制化改造。通过前面的Spring Security的源码分析，我们知道主要的方式就是继承WebSecurityConfigurerAdapter，这样做的好处在于，我们依然可以使用spring-boot-starter-security默认的一些行为，只需要对必要的行为进行调整，比如：

- 配置多个AuthenticationManager实例。
- 对HttpSecurity定义的默认资源访问规则进行重新定义。
- 对提供的默认WebSecurity行为进行调整。

为了能够让这些调整生效，我们定义的WebSecurityConfigurerAdapter实现类一般在顺序上需要先于spring-boot-starter-security默认提供的配置，所以一般需要配合@Order注解进行标注，代码如下：

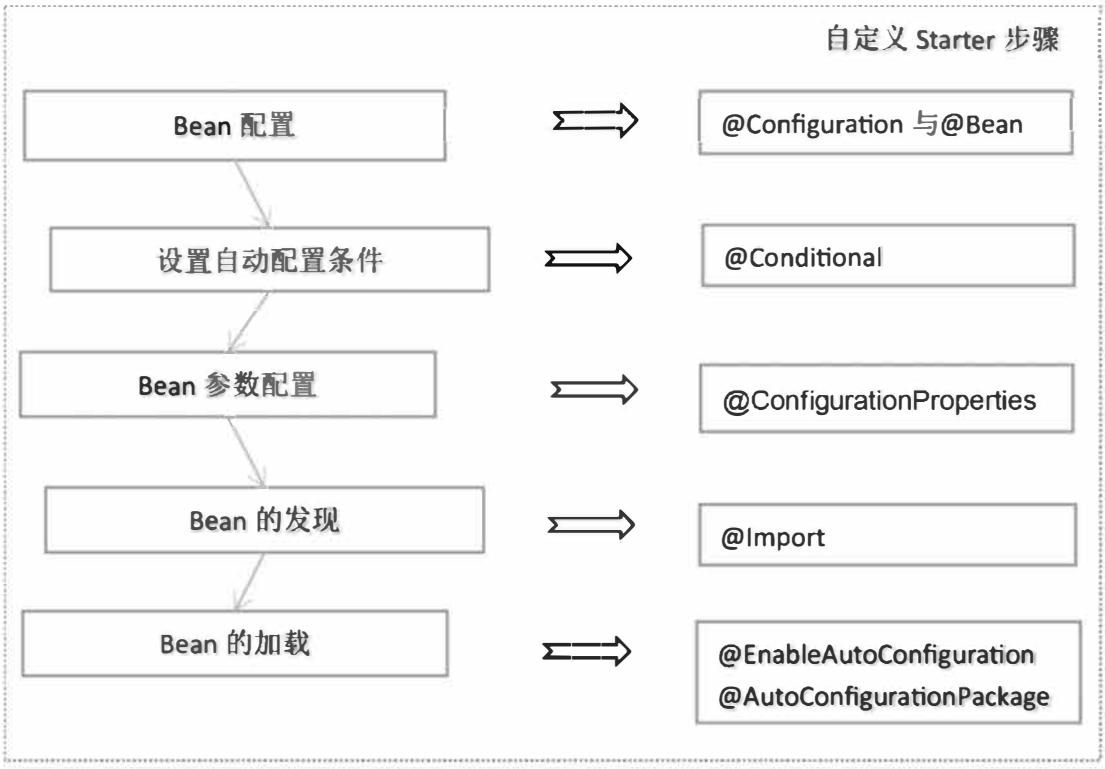
```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class DemoSecurityConfiguration extends WebSecurityConfigurerAdapter {
    protected DemoSecurityConfiguration() {
        super(true); //取消默认提供的安全相关 Filter 配置
    }
    @Override
    public void configure(WebSecurity Web) throws Exception {
        //省略
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //省略
    }
    //通过覆盖其他方法实现对 Web 安全的定制
}
```

总之，WebSecurityConfigurerAdapter是Spring Security框架为我们预先设定的一个安全框架，它允许我们对Web安全相关的功能进行定制化开发，但是在某些场景下我们还是会感觉不方便，此时我们可以直接实现并注册一个标注了@EnableWebSecurity的JavaConfig配置类到IoC容器，从而实现一种“颠覆性”的定制，示例代码如下：

```
@Configuration
@EnableWebSecurity
public class OverhaulSecurityConfiguration {
    @Bean
    public AuthenticationManager authenticationManager() {
        //省略
    }
    @Bean
    public AccessDecisionManager accessDecisionManager() {
        //省略
    }
    @Bean
    public SecurityFilterChain mySecurityFilterChain() {
        //省略
    }
}
```

## 4.3.5 Spring Boot实现自定义Starter

下面我们通过介绍在一个微服务网关项目（Sia-Gateway已在GitHub开源）中自定义Starter，了解自定义Starter的关键步骤，以及实现一个自定义注解的完整步骤，如下图所示。




使用到的关键注解和功能的对应关系如下。

- @Configuration与@Bean的作用：基于Java代码的Bean配置。
- @Conditional的作用：设置自动配置条件依赖。
- @EnableConfigurationProperties与@ConfigurationProperties的作用：读取配置文件并将其转换为Bean。
- @EnableAutoConfiguration、@AutoConfigurationPackage与@Import的作用：Bean的发现与加载。

**准备工作**

在开始实现自定义Starter之前，我们需要手动创建两个工程。前文中已有介绍，第一个工程更像一个门面（Facade）工程，也可以通过引入第一个自定义的Starter工程验证自定义Starter的功能；第二个工程则是实现自定义Starter的实际业务功能部分，完成自动化配置定义、发现配置及实现特定的业务功能。

- sag-spring-boot-starter(pom.xml)

```
<?XML...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.msa</groupId>
    <artifactId>sag-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>sag-spring-boot-starter</name>
    <!-- 启动器 -->
    <dependencies>
        <!-- 引入自动配置模块 -->
        <dependency>
            <groupId>com.gf</groupId>
            <artifactId>sag-spring-boot-starter-autoconfigurer</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

- sag-spring-boot-starter-autoconfigurer(pom.xml)

```
<?XML...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.msa</groupId>
    <artifactId>sag-spring-boot-starter-autoconfigurer</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>sag-spring-boot-starter-autoconfigurer</name>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!-- 引入 spring-boot-starter，所有 Starter 的基本配合 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
    </dependencies>
</project>
```

**基于Java代码的Bean配置**

下面是自定义Starter的详细配置加载过程，首先使用自动配置AutoConfiguration初始化Bean对象实例，代码如下：

```
@Configuration
@EnableConfigurationProperties({ SagProxyProperties.class })
@ConditionalOnBean(SagProxyMarkerConfiguration.Marker.class)
public class SagProxyAutoConfiguration {
    @Autowired
    private SagProxyProperties properties;
    @Bean
    @ConditionalOnMissingBean(ThirdPartyComponent.class)
    public ThirdPartyComponent getThirdcomp() {
        ThirdPartyComponent tpcomp = new
            ThirdPartyComponent(properties.getFilterdynamic(),
        properties.getJavapackageName(),  properties.getFilterinterval());
            return tpcomp;
    }
}
```

从上面的代码中可以看到@Configuration、@Bean，这两个注解一起使用可以创建一个基于Java代码的配置类，它可以用来替代加载相应XML配置文件的过程。@Configuration注解的类可以看作让Spring容器管理的Bean实例的工厂。@Bean注解代表准备注册到Spring容器的对象实例，也就是一个带有@Bean的注解方法将返回的对象，该对象应该被注册到Spring容器中。自动配置类SagProxyAutoConfiguration的作用是自动生成微服务网关需要的Bean实例ThirdPartyComponent，并将这个Bean实例交给Spring容器管理，从而完成Bean的自动注册。

**自动配置条件依赖**

从SagProxyAutoConfiguration类中使用的注解可以看出，要完成自动配置是有依赖条件的，即@ConditionalOnBean（SagProxyMarkerConfiguration.Marker.class）。

根据条件（ConditionalOnBean），要完成SagProxy的自动化配置，我们需要在类路径中判断声明类SagProxyMarkerConfiguration.mark.class，当这个Bean存在时才会完成Bean的自动注册。

**Bean参数的获取**

至此我们已经知道了Bean的配置过程，但是还没有看到Spring Boot是如何读取YAML或者Properites配置文件的属性来创建数据源的，在SagProxyAutoConfiguration类里使用了@EnableConfigurationProperties注解：

```
@EnableConfigurationProperties({ SagProxyProperties.class })
```

SagProxyProperties中封装了Proxy配置的各个属性，并且使用Spring自带的注解ConfigurationProperties指定了配置文件的前缀为“sag 注 ”，相关配置代码如下：

```
@ConfigurationProperties(prefix = "sag")
public class SagProxyProperties {
    private String version = "sag_1.0";
    private String desc = "SIA GATEWAY Provide ";
    private String port = "8080";
    private String superenabled = "false";
    private String debugenabled = "false";
    private String ribbonenabled = "true";
    private String logenabled = "true";
    private String ratelimitenabled = "true";
    private String statisticenabled = "true";
    private String filterdynamic = "true";
    private String javapackageName = "com.creditease.gateway.filter
}
```

通过以上分析我们可以得知：@ConfigurationProperties注解的作用是把YAML或者Properties配置文件转化为Bean对象，@EnableConfigurationProperties注解的作用是使@ConfigurationProperties注解生效，如果只配置@ConfigurationProperties注解，在Spring容器中获取不到YAML或者Properties配置文件转化的Bean。

**Bean的发现**

Spring Boot默认扫描启动类所在的包下的主类与子类的所有组件，但并没有包括依赖包中的类，那么依赖包中的Bean是如何被发现和加载的？这还要从启动类中的@SpringBootApplication注解说起，源码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM,  classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM,  classes = AutoConfigurationExcludeFilter.class) })
    public @interface SpringBootApplication {
    /**
    * Exclude specific auto-configuration classes such that they will never be applied.
    * @return the classes to exclude
    */
    @AliasFor(annotation = EnableAutoConfiguration.class,  attribute = "exclude")
    Class<?>[] exclude() default {};
}
```

- @Configuration：其作用前面我们已经讲过了，被注解的类将成为一个Bean配置类。
- @EnableAutoConfiguration：其功能很重要，可以借助@Import的支持，收集和注册依赖包中相关的Bean定义。
- @ComponentScan：其作用就是自动扫描并加载符合条件的组件，比如@Component和@Repository等，最终将这些Bean定义加载到Spring容器中。

@EnableAutoConfiguration注解引入了@AutoConfigurationPackage和@Import这两个注解。@AutoConfigurationPackage的作用就是对自动配置package进行自动管理，而注解@Import主要完成需要自动配置的包的导入，源码如下：

```
@SuppressWarnings("deprecation")
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}
```

要搜集并注册到Spring容器的那些Bean来自哪里呢？如下代码所示：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {
}
```

Registrar类的作用是扫描主配置类的同级目录及子包，并将相应的组件导入Spring Boot创建管理容器中，源码如下：

```
static class Registrar implements ImportBeanDefinitionRegistrar， DeterminableImports {
@Override
public void registerBeanDefinitions(AnnotationMetadata metadata，
    BeanDefinitionRegistry registry) {
    register(registry， new PackageImport(metadata).getPackageName());
    }
    @Override
    public Set<Object> determineImports(AnnotationMetadata metadata) {
        return Collections.<Object>singleton(new PackageImport(metadata));
    }
}
```

如果进入AutoConfigurationImportSelector类，可以发现SpringFactoriesLoader的loadFactoryNames方法会调用loadSpringFactories从所有的jar包中读取META-INF/spring.factories文件信息。

下面是spring-boot-autoconfigure这个jar包中的spring.factories文件的部分内容，其中有一个Key为org.springframework.boot.autoconfigure.EnableAutoConfiguration的值定义了需要自动配置的Bean，通过读取这个配置可以获取一组@Configuration类。

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.creditease.gateway.config.GatewayConfig，\
com.creditease.store.redis.config.RedisConfig，\
com.creditease.gateway.config.MultipartConfig，\
com.creditease.gateway.config.SwaggerConfig，\
com.creditease.gateway.config.GatewayControllerConfiguration，\
com.creditease.gateway.config.GatewayServiceConfiguration，\
com.creditease.gateway.config.GatewayFallbackConfiguration，\
com.creditease.gateway.config.GatewayFilterConfiguration，\
com.creditease.gateway.config.GatewayRegisterConfiguration，\
com.creditease.gateway.sag.config.SagProxyAutoConfiguration，\
com.creditease.gateway.config.CommonConfiguration
```

每个xxxAutoConfiguration都是一个基于Java的Bean配置类。不是所有的xxxAutoConfiguration都会被加载，会根据xxxAutoConfiguration上@ConditionalOnClass等条件判断是否加载；最后，通过反射机制将spring.factories中@Configuration类实例化为对应的Java实例。至此，我们已经知道Spring Boot是通过怎样的机制发现准备自动配置的Bean的，接下来就要考虑怎样将这些Bean加载到Spring容器。

**Bean的加载**

如果要将一个普通类交给Spring容器管理，Spring Boot通常使用下面两种方式实现Bean的加载。大多数情况下我们会使用第一种方式，而对于自定义Starter，显然我们使用了第二种方式。

- 方式一：在配置类（@Configuration）中增加方法级别注解（@Bean）或者使用类级别注解，使用@Controller、@Service、@Repository、@Component注解标注该类的方式注入Bean，然后确保@ComponentScan自动扫描路径包含上述注解类。
- 方式二：从Spring Boot的自动化配置过程可以发现，@EnableAutoConfiguratio注解中使用了@Import（{AutoConfigurationImportSelector.class}）注解。而@Import注解中的类AutoConfigurationImportSelector实现了DeferredImportSelector等一系列接口，该接口是ImportSelector的子接口，通常我们可以直接实现ImportSelector接口并实现selectImports方法。当我们通过@Import注解向实现了ImportSelector接口的选择器添加相应的自动化配置注解，并在启动类中使用该注解时，selectImports方法将会交给容器调用，从而实现Bean的自动化加载，Spring Boot正是通过这种机制来将Bean注入容器的。此外，还可以通过BeanDefinitionRegistry动态注入Bean，详细逻辑这里就不赘述了。

## 4.4 Spring Boot Web容器

Web应用开发是企业开发的重要领域，Spring Boot 1.X的Web容器管理方式基于Servlet容器技术栈。Servlet容器主要基于同步阻塞I/O架构，HTTP请求和线程是一对一的关系，主要是TPR模型，即一个请求对应一个线程。主要的业务逻辑也是基于命令式的编程模式。以Spring

MVC框架为主，Web容器方面以Tomcat为主，也可以通过自动配置功能改为Jetty/UnderTow容器。

Spring Boot 2.X主要基于异步非阻塞I/O架构，HTTP请求基于收敛的线程模型，网络层使用基于Reactor的I/O多路复用模式，业务逻辑基于函数式编程模式，以Spring WebFlux为主要框架。在Web容器方面可以基于Servlet 3.0的异步模式，默认情况下使用Netty作为容器。

本节我们主要以Spring Boot 1.X讲解嵌入式Web容器的启动和加载原理，在进阶篇的响应式编程中将介绍Spring 5及Spring Boot 2.X的响应式框架WebFlux对Web应用服务的支持。

## 4.4.1 Spring Boot Web容器配置

Spring Boot对Web项目的支持主要是Spring Boot对Spring MVC框架的继承。Spring MVC框架是一个基于Servlet容器标准的Web容器框架实现，Spring Boot向Spring MVC提供开箱即用的Starter：spring-boot-starter-web。

Spring Boot应用中利用自动配置功能，只需要在pom.xml文件中加入下面的Web依赖，就可以直接启动一个Web服务：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Web MVC框架使用特定的@Controller或者@RestController注解的Bean作为处理HTTP请求的端点，通过@RequestMapping注解将控制器中的方法与HTTP请求进行映射，示例如下：

```
@RestController
public class DemoController {
    @RequestMapping (value="/{user}",  method = RequestMethod.GET)
    public User getUser(){
    Return “zhangsan is return”;
    }
}
```

Spring Boot为Spring MVC提供了自动配置功能，包含如下主要配置特性。

● 自动配置ViewResolver引入ContentNegotiatingViewResolver组件功能。

示例：在应用中添加ViewResolver组件用来匹配HTML静态页面，如果没有匹配成功，则返回false，由其他ViewResolver继续尝试匹配。ContentNegotiatingViewResolver会组合所有的视图解析器，代码如下。

```
@Bean
public ViewResolver htmlResolver()
{
   Return new HtmlResourceView ();
}
public class HtmlResourceView extends InternalResourceView {
    @Override
    public boolean checkResource(Locale locale) {
        File file = new File(this.getServletContext().getRealPath("/") + getUrl());
        return file.exists();//判断该页面是否存在
    }
}
```

● 自动注册Converter、GenericConverter、Formatter Bean。

示例：将页面提交数据转化为后台数据，实现格式化，代码如下。

```
@Configuration
public class MyConfiguration {
@Bean
@CondtionOnProperty(prefix=" spring.mvc" , name=" data-format" )
public Formatter<Date> dateFormatter() {
    return new DateFormatter(this.mvcProperties.getDateFormat());
    }
}
```

● 对HttpMessageConverters的支持。

示例：Spring Boot可以为HttpMessageConverters类添加自定义转换类，通过这种方式可以将所有的HttpMessageConverters的Bean添加到Converter列表，覆盖默认的转换器列表，代码如下。

```
@Configuration
public class MyConfiguration {
    @Bean
    public HttpMessageConverters customConverters() {
    HttpMessageConverter<?> additional = ...
    HttpMessageConverter<?> another = ...
    return new HttpMessageConverters(additional,  another);
    }
}
```

● 自动注册MessageCodeResolver。

● 自动使用ConfigurableWebBindingInitializer Bean。

● 使用WebMvcConfigurerAdapter类型的Bean来定制化配置。

默认情况下，Spring Boot会以/src/main/resources/static作为查找静态资源的文件路径，如果想自定义静态资源映射目录，需要重写addResourceHandlers来添加指定路径，重写addResourceLocations来指定静态资源路径。

```
@Configuration
public class MyWebMvcConfigurerAdapter extends WebMvcConfigurerAdapter {
/**
 * 配置静态访问资源
 */
```

```
@Override
 public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/my/**")
    .addResourceLocations("classpath:/my/");
    super.addResourceHandlers(registry);
  }
}
```

总之，我们可以根据自己的意愿，对默认的Spring MVC的组件配置加以修改，方法也很简单，通过在IoC容器中注册新的同类型Bean来替换即可。如果你希望完全接管Spring MVC的所有相关配置，可以添加自己的@Configuration，并使用@EnableWebMvc注解实现定制化配置。

**JAX-RS和Jersey框架**

如果你喜欢JAX-RS和REST风格的编程模型，可以使用下面的Starter替代Spring MVC框架，Spring支持Jersey 1.X和Jersey 2.X等技术框架。这里我们只介绍Spring Boot对Jersey 2.X的支持，在pom.xml文件中加入下面的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
</dependency>
```

Spring Boot对Jersey的配置有三种主要方式。在开始不同的配置方式前，我们注册一个端点对象资源，示例代码如下：

```
@Component
@Path("/hello")
public class Endpoint {
    @GET
    public String message() {
    return "Hello";
    }
}
```

- 第一种方式，创建一个自定义的ResourceConfig：

```
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
    register(Endpoint.class);
    }
}
```

- 第二种方式，返回一个ResourceConfig类型的@Bean：

```
@Bean
public ResourceConfig resourceConfig() {
    ResourceConfig config = new ResourceConfig();
    config.register(Endpoint.class);
    return config;
}
```

- 第三种方式，配置一组ResourceConfigCustomizer对象。

Spring Boot提供了ResourceConfigCustomizer接口，让我们更灵活地对ResourceConfig对象进行配置。要使用该接口，我们需要先注释掉前面两节中提到的相关代码，然后创建一个类：

```
@Component
public class MyResourceConfigCustomizer implements ResourceConfigCustomizer {
    @Override
    public void customize(ResourceConfig config) {
        config.register(SpringbootResource.class);
    }
}
```

默认情况下，Jersey将以Servlet的形式注册一个ServletRegistrationBean类型的@Bean。它的名字为jerseyServletRegistration，该Servlet默认会延迟初始化。

你可以通过spring.jersey.servlet.load-on-startup自定义配置Jersey组件。通过创建相同名字的Bean，可以禁用或覆盖框架默认的Bean。设置spring.jersey.type=filter可以使用Filter的形式代替Servlet，相应的@Bean类型变为jerseyFilter-Registration，该Filter有一个@Order属性，你可以通过spring.jersey.filter.order设置该属性。Servlet和Filter在注册时都可以使用spring.jersey.init.*定义一个属性集合并将其传递给init参数进行初始化。

**内嵌容器的配置**

Spring Boot的另一大特性就是支持内嵌的Web容器，包括Tomcat、Jetty和UnderTow服务器，大多数开发者只需要使用合适的Starter来获取一个完全配置好的实例即可，内嵌服务器默认监听8080端口的HTTP请求。spring-boot-starter-web默认使用Tomcat作为Web容器，你可以在pom.xml中去除spring-boot-starter-tomcat依赖，然后引入spring-boot-starter-jetty或者spring-boot-starter-undertow模块作为替代Web容器方案。Starter还提供了以“server.”为前缀的配置项对嵌入式容器配置进行修改。配置项的加载和定制化钩子加载过程如下。

1.自动化配置嵌入式容器

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(BeanPostProcessorsRegistrar.class)
public class EmbeddedServletContainerAutoConfiguration {
    /**
    * Nested configuration if Tomcat is being used.
    */
    @Configuration
    @ConditionalOnClass({ Servlet.class,  Tomcat.class })
    @ConditionalOnMissingBean(value=EmbeddedServletContainerFactory.class, search=
        SearchStrategy.CURRENT)
    public static class EmbeddedTomcat {
        @Bean
        public TomcatEmbeddedServletContainerFactory
            tomcatEmbeddedServletContainerFactory() {
        return new TomcatEmbeddedServletContainerFactory();
        }
    }
}
```

2.初始化TomcatEmbeddedServletContainerFactory的Bean对象

```
public class TomcatEmbeddedServletContainerFactory
    extends AbstractEmbeddedServletContainerFactory implements ResourceLoaderAware {
        public TomcatEmbeddedServletContainerFactory() {
            super();
        }
}
```

3.定制化Bean扩展逻辑

EmbeddedServletContainerCustomizerBeanPostProcessor在加载Bean后开始初始化配置项PostProcessor的处理逻辑：

```
public class EmbeddedServletContainerCustomizerBeanPostProcessor
        implements BeanPostProcessor， ApplicationContextAware
{
 private void postProcessBeforeInitialization(ConfigurableEmbeddedServletContainer bean)
 {
      for (EmbeddedServletContainerCustomizer customizer : getCustomizers()) {
           customizer.customize(bean);
      }
 }
}
```

4.配置文件加载

从配置文件中，你可以加载配置文件对象的配置值。如果配置文件中没有相关配置项，将使用默认代码设定配置。

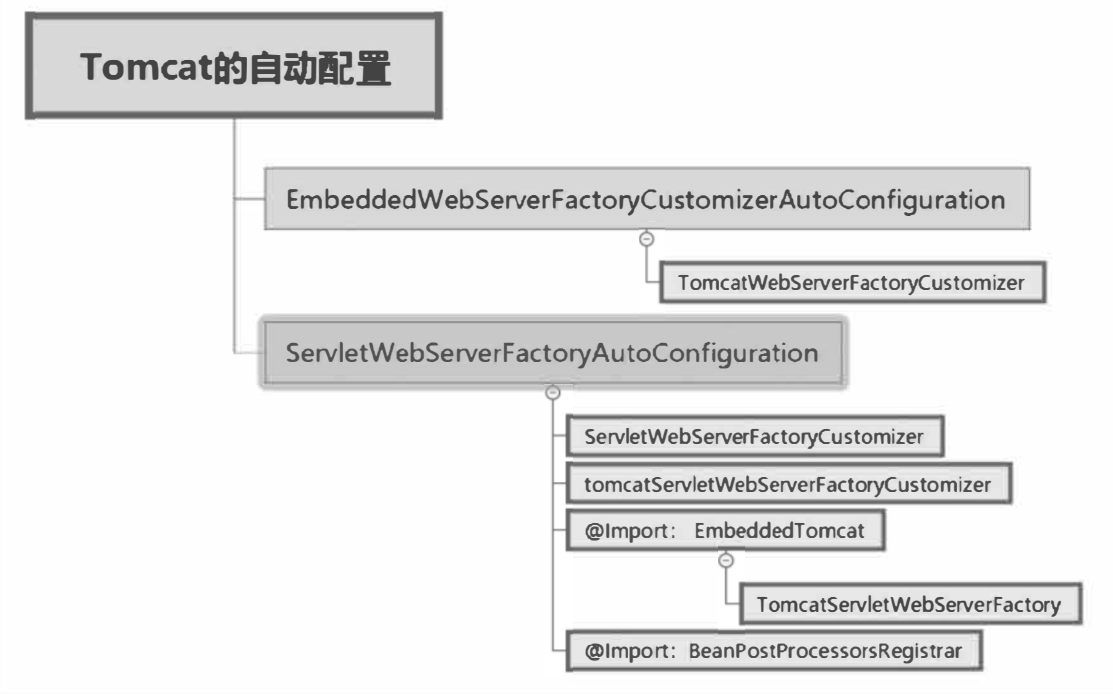
```
@ConfigurationProperties(prefix = "server"， ignoreUnknownFields = true)
public class ServerProperties
    implements EmbeddedServletContainerCustomizer， EnvironmentAware， Ordered {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        if (getPort() != null) {
            container.setPort(getPort());
        }
    }
    //省略
}
```

5.Web容器定制化

如果你需要对Web容器进行更深入的定制，可以使用对应的Factory自动化配置Tomcat容器，它是初始化的关键流程和步骤，代码示例如下：

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new TomcatEm
        beddedServletContainerFactory();
    factory.setPort(9000);
    factory.setSessionTimeout(10， TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND， "/
    notfound.html");
    return factory;
}
```

下图是Spring Boot启动过程中Tomcat容器完成自动配置的类图结构。我们在最新的Spring Boot下查看Tomcat的相关配置，发现有两个自动装配类，分别包含了三个定制器，还有一个工厂类。




## 4.4.2 Spring Boot嵌入式Web容器原理

Spring Boot的目标是构建“非常容易创建、独立、产品级别的基于Spring的应用”。这些应用是“立即可运行的”。在这个过程中，完全没有代码生成，不需要配置任何特殊的XML配置，为了这个目标，Spring Boot在Spring 4.0框架之上提供了很多特性，帮助应用以“约定优于配置”“开箱即用”的方式来启动应用并运行上下文。

Spring Boot同样改变了一个传统的Web应用服务的启动流程和部署方式。通过自动配置机制，Spring Boot提供了一个嵌入式的运行时容器环境，并使用代码注解的方式在代码中将URL服务地址映射到Controller的方法完成服务映射。开发者不再需要关心传统容器（如Tomcat）中web.xml的配置，同时实现容器的具体技术都是可替换及可更改的，这些技术以插件化的Starter组件方式在运行时加载到Spring容器中。

**ContainerLess理念**

微服务把应用和它所依赖的组件包、配置文件及附带的运行脚本打包成一个单一、独立、可执行的jar包文件。

在实现Web服务器时，几乎不需要任何配置就可以启动Tomcat。你只需要使用java-jar命令就可以让Tomcat成为Spring Boot的一个自包含的可运行组件和单元。同时，这种自带容器的运行部署方式对云开发环境天然友好。在Spring Boot启动流程和容器配置中，其实我们已经介绍了一点内嵌Tomcat容器的原理，下面具体看一下Spring Boot加载Tomcat的具体执行步骤。

1.Spring Boot引入Tomcat依赖

当我们在项目中加入org.springframework.boot、spring-boot-starter-web等依赖后，Maven会把Tomcat依赖的一些jar包组件也加载进来。

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>8.5.31</version>
</dependency>
```

2.创建Web容器的Context

在Spring Boot的run方法中，我们发现其中Web容器加载很重要的一步就是下面的代码：创建、加载、刷新、运行Spring容器的ConfigurableApplicationContext模块。

```
protected ConfigurableApplicationContext createApplicationContext() {
  Class<?> contextClass = this.applicationContextClass;
  if (contextClass == null) {
    try {
     contextClass = Class.forName(this.WebEnvironment
        ? DEFAULT_Web_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
    }
    catch (ClassNotFoundException ex) {
     throw new IllegalStateException(
        "Unable create a default ApplicationContext,  "
          + "please specify an ApplicationContextClass", ex);
   }
  }
  return (ConfigurableApplicationContext)
    BeanUtils.instantiate(contextClass);
}
```

在经历过Context的创建及Context的一系列初始化步骤之后，调用Context的refresh方法。它最终会调用AnnotationConfigEmbeddedWebApplicationContext类的refresh方法，并由其父类EmbeddedWebApplicationContext执行刷新，源码如下：

```
@Override protected void onRefresh()
{
    super.onRefresh();
    try
    {
        createEmbeddedServletContainer();
    }
    catch (Throwable ex) {
        throw new ApplicationContextException("Unable to start Web server",  ex);
    }
}
```

3.创建Tomcat实例

这里我们继续跟进createEmbeddedServletContainer方法，获得一个嵌入式的容器工厂类：EmbeddedServletContainerFactory。它是一个抽象工厂类，你可以根据不同的容器类型选择不同的容器加载实现。Spring Boot的默认容器是Tomcat，其工厂类实现了Tomcat实例的加载，代码如下：

```
@Override
public EmbeddedServletContainer getEmbeddedServletContainer(
    ServletContextInitializer... initializers) {
```

```
    Tomcat tomcat = new Tomcat();
    File baseDir = (this.baseDirectory != null ? this.baseDirectory: createTempDir("tomcat"));
    tomcat.setBaseDir(baseDir.getAbsolutePath());
    Connector connector = new Connector(this.protocol);
    tomcat.getService().addConnector(connector);
    customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    tomcat.getEngine().setBackgroundProcessorDelay(-);
    for (Connector additionalConnector : this.additionalTomcatConnectors) {
       tomcat.getService().addConnector(additionalConnector);
    }
    prepareContext(tomcat.getHost(), initializers);
    return getTomcatEmbeddedServletContainer(tomcat);
}
```

在prepareContext方法中，可以将默认的JSP和Servlet Bean组件加载到Spring容器，并对所有ServletContextInitializer进行合并，然后利用合并后的初始化类对Context进行配置，代码如下：

```
if (isRegisterDefaultServlet()) {
    addDefaultServlet(context);
}
if (isRegisterJspServlet() && ClassUtils.isPresent(
    getJspServletClassName(), getClass().getClassLoader())) {
      addJspServlet(context);
      addJasperInitializer(context);
      context.addLifecycleListener(new StoreMergedWebXMLListener());
}
ServletContextInitializer[] initializersToUse = mergeInitializers(initializers);
configureContext(context, initializersToUse);
//省略
//初始化 Tomcat 容器服务
protected TomcatEmbeddedServletContainer
getTomcatEmbeddedServletContainer(Tomcat tomcat) {
    return new TomcatEmbeddedServletContainer(tomcat, getPort() >= 0);
}
```

4.Tomcat初始化

下面是Tomcat的正式初始化过程，从tomcat.start方法开始，Tomcat实例开始运行。

```
private synchronized void initialize() throws EmbeddedServletContainerException {
    TomcatEmbeddedServletContainer.logger.info("Tomcat initialized with
        port(s): " + getPortsDescription(false));
    try {
        addInstanceIdToEngineName();
        removeServiceConnectors();
        this.tomcat.start();
        rethrowDeferredStartupExceptions();
        startDaemonAwaitThread();
      }
      catch (Exception ex) {
         throw new EmbeddedServletContainerException("Unable to start embedded
        Tomcat", ex);
      }
}
```

5.Tomcat组件加载

下面的代码是在Tomcat容器启动后，允许用户存储自定义scope，用来将Web专用的scope注册到BeanFactory中，同时配置servlet、filter、listener、context-param等。beans.onStartup（servletContext）方法中将实现从Servlet到URLMapping的映射，至此Tomcat的初始化工作完成。

```
private void selfInitialize(ServletContext servletContext) throws ServletException {
  prepareEmbeddedWebApplicationContext(servletContext);
  ConfigurableListableBeanFactory beanFactory = getBeanFactory();
  ExistingWebApplicationScopes existingScopes = new
    ExistingWebApplicationScopes(beanFactory);
  WebApplicationContextUtils.registerWebApplicationScopes(beanFactory,
     getServletContext());
  existingScopes.Restore();
  WebApplicationContextUtils.registerEnvironmentBeans(beanFactory,
     getServletContext());
  for (ServletContextInitializer beans : getServletContextInitializerBeans()) {
    beans.onStartup(servletContext);
  }
}
```

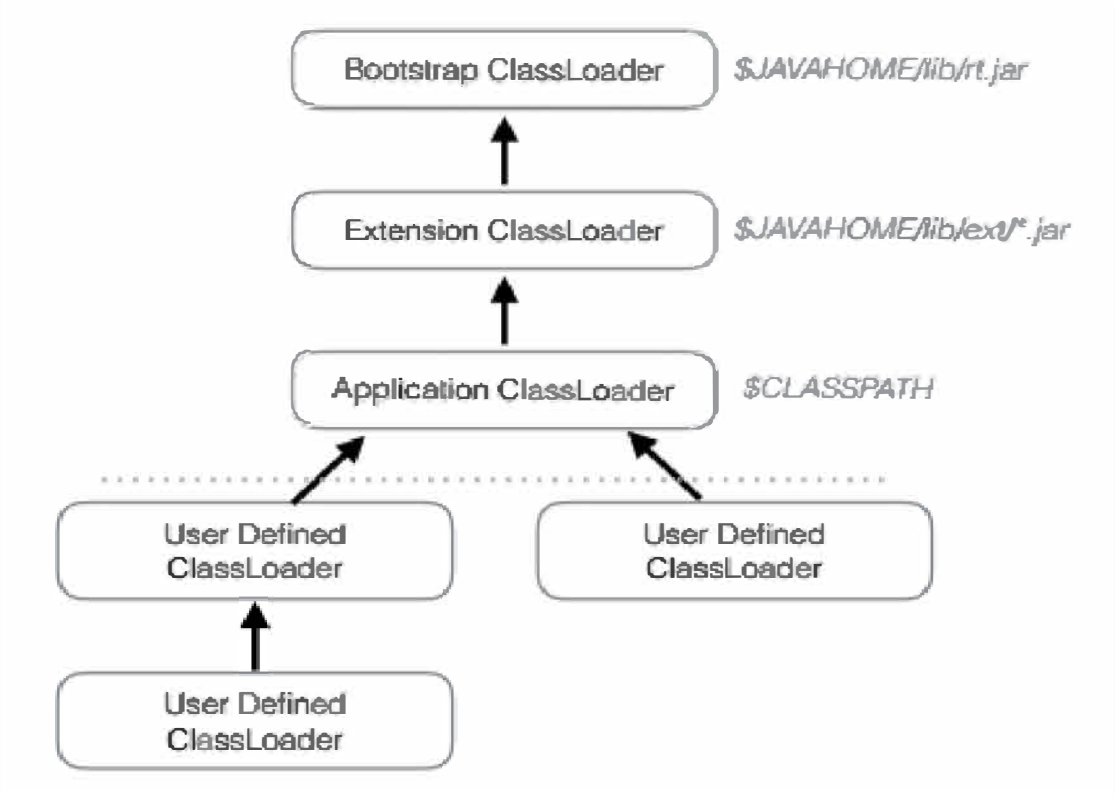
## 4.4.3 Spring Boot的ClassLoader加载机制

在Spring Boot的嵌入式Web容器原理一节中，我们已经介绍了Spring Boot对Tomcat容器的加载过程，本节我们进一步讲解Spring Boot的ClassLoader加载机制。

熟悉Tomcat工作原理的人应该知道，Tomcat内部实现了自定义的类加载器，打破了Java的双亲委派机制，下面我们先看看什么是双亲委派机制。

**双亲委派机制**

双亲委派机制是指Java的类加载器收到一个类加载请求时，该类加载器首先会把请求委派给父类加载器。每个类加载器都是如此，只有当父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。Java类加载机制如下图所示。



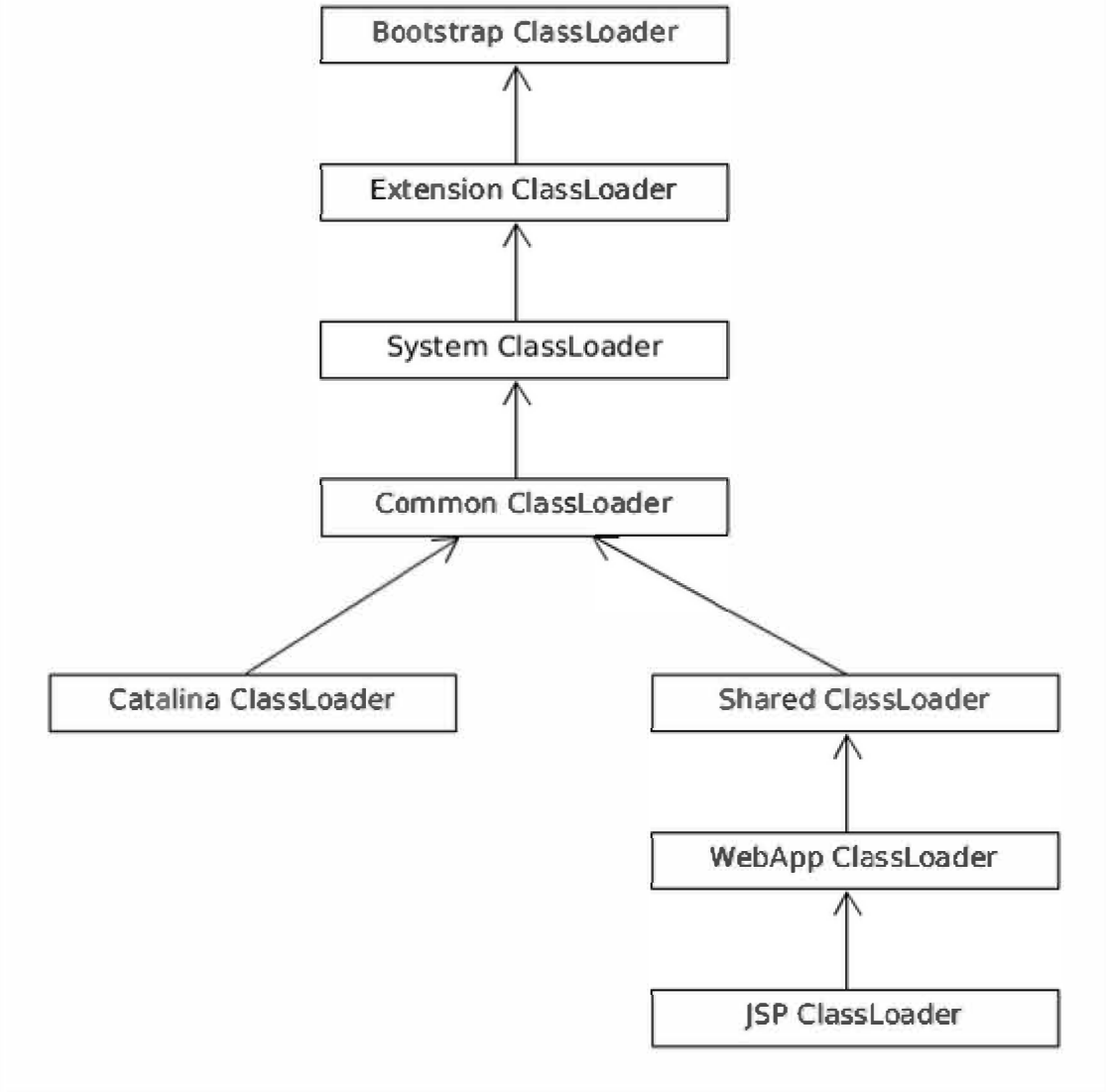

我们通常将类加载器分为下面的三种类型。

- 启动类加载器（Bootstrap ClassLoader）：加载jre/lib/rt.jar。
- 扩展类加载器（Extension ClassLoader）：加载jre/lib/ext/*.jar。
- 应用程序类加载器（Application ClassLoader）：加载classpath上指定的类库。

如果使用JDK默认的双亲委派模式，Tomcat的类加载器可以加载吗？我们思考一下Tomcat作为一个Web容器的使用场景。

在Web容器中，可能同时需要部署两个以上的应用程序。一个典型的场景是不同的应用程序会依赖同一个第三方类库的不同版本，不能要求同一个类库在同一个服务器中只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。

Tomcat如果使用默认类加载器，是无法加载两个相同类库的不同版本的。所以Tomcat团队设计了自己独特的类加载机制，解决上面的应用jar包冲突等问题，通过自定义的类加载机制可以完美地解决Tomcat容器中不同应用的隔离问题。下面我们看看Tomcat的类加载机制图和JDK默认的加载机制图的区别，如下图所示。




其中：

- Common ClassLoader：Tomcat最基本的类加载器，加载路径中的Class可以被Tomcat容器本身及各个WebApp访问。
- Catalina ClassLoader：Tomcat容器私有的类加载器，加载路径中的Class对于WebApp不可见。
- Shared ClassLoader：各个WebApp共享的类加载器，加载路径中的Class对所有WebApp可见，但是对于Tomcat容器不可见。
- WebApp ClassLoader：各个WebApp私有的类加载器，加载路径中的Class只对当前WebApp可见，各个项目就是通过各自的WebApp ClassLoader加载进入Tomcat容器的。

**探索Spring Boot的ClassLoader**

Spring Boot的内置Tomcat是如何加载到我们的项目中的呢？我们还是从SpringApplication的run方法开始追溯Tomcat启动Web Server的过程，ApplicationContext执行刷新操作并创建嵌入式容器，源码如下：

```
public class EmbeddedWebApplicationContext extends GenericWebApplicationContext {
    @Override
```

```
    protected void onRefresh() {
        super.onRefresh();
        try {
            createEmbeddedServletContainer();
        }
        catch (Throwable ex) {
            throw new ApplicationContextException("Unable to start embedded
            container", ex);
        }
    }
}
private void createEmbeddedServletContainer() {
    EmbeddedServletContainer localContainer = this.embeddedServletContainer;
    ServletContext localServletContext = getServletContext();
    if (localContainer == null && localServletContext == null) {
        EmbeddedServletContainerFactory containerFactory =
            getEmbeddedServletContainerFactory();
        this.embeddedServletContainer = containerFactory
            .getEmbeddedServletContainer(getSelfInitializer());
    }
    else if (localServletContext != null) {
        try {
            getSelfInitializer().onStartup(localServletContext);
        }
        catch (ServletException ex) {
            throw new ApplicationContextException("Cannot initialize servlet context", ex);
        }
    }
    initPropertySources();
}
```

然后，进入EmbeddedServletContainer的getEmbeddedServletContainer方法，它会初始化Tomcat实例并准备Context。

```
@Override
public EmbeddedServletContainer getEmbeddedServletContainer(
    ServletContextInitializer... initializers) {
//省略
//准备 Context
prepareContext(tomcat.getHost(), initializers);
//创建 Container
return getTomcatEmbeddedServletContainer(tomcat);
}
```

最后，跟进prepareContext方法，我们就可以看到嵌入式Tomcat的类加载方式，源码如下：

```
protected void prepareContext(Host host, ServletContextInitializer[] initializers) {
    File docBase = getValidDocumentRoot();
```

```
        docBase = (docBase != null ? docBase : createTempDir("tomcat-docbase"));
        TomcatEmbeddedContext context = new TomcatEmbeddedContext();
        context.setName(getContextPath());
        context.setDisplayName(getDisplayName());
        context.setPath(getContextPath());
        context.setDocBase(docBase.getAbsolutePath());
        context.addLifecycleListener(new FixContextListener());
        context.setParentClassLoader(
         this.resourceLoader != null ? this.resourceLoader.getClassLoader()
              : ClassUtils.getDefaultClassLoader());
        resetDefaultLocaleMapping(context);
        addLocaleMappings(context);
        try {
         context.setUseRelativeRedirects(false);
        }
        catch (NoSuchMethodError ex) {
         // Tomcat is < 8.0.30. Continue
        }
        SkipPatternJarScanner.apply(context, this.tldSkip);
        WebappLoader loader = new WebappLoader(context.getParentClassLoader());
        loader.setLoaderClass(TomcatEmbeddedWebappClassLoader.class.getName());
        loader.setDelegate(true);
        context.setLoader(loader);
        if (isRegisterDefaultServlet()) {
         addDefaultServlet(context);
        }
        if (shouldRegisterJspServlet()) {
         addJspServlet(context);
         addJasperInitializer(context);
         context.addLifecycleListener(new StoreMergedWebXmlListener());
        }
        ServletContextInitializer[] initializersToUse = mergeInitializers(initializers);
        configureContext(context, initializersToUse);
        host.addChild(context);
        postProcessContext(context);
    }
```

可见，Spring Boot以启动线程的Context ClassLoader作为Tomcat的WebApp ClassLoader的父类加载器，而Tomcat的WebApp类加载器使用TomcatEmbeddedWebAppClassLoader。所以整个项目的jar包的加载都是由Spring Boot的主线程Context ClassLoader完成的，于是Context ClassLoader就可以访问我们的Web容器下的所有资源了。

需要说明的是，Spring Boot使用了FatJar技术将所有依赖放在一个最终的jar包文件BOOT-INF/lib中，它可以把当前项目的Class全部放在BOOT-INF/classes目录中。你可以在Spring Boot的工程项目中看到，在pom.xml文件中引入了如下依赖：

```
<plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
      <includeSystemScope>true</includeSystemScope>
   </configuration>
</plugin>
```

jar包目录结构如下：

```
spring-boot-demo-1.0.0.jar
├── META-INF
│   └── MANIFEST.MF
├── BOOT-INF
│   ├── classes
│   │   └── 应用程序
│   └── lib
│   └── 第三方依赖 jar
└── org
└── springframework
└── boot
└── loader
└── springboot 启动程序
```

从这个目录结构中，你可以看到Tomcat的启动包（tomcat-embed-core-8.5.29.jar）就在Lib目录下。而FatJar的启动Main函数就是JarLauncher，它负责创建LaunchedURLClassLoader来加载/lib下面的所有jar包。下面是Spring Boot应用的Manifest文件内容。

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: peihua
Start-Class: com.test.demo.SpringbootDemoApplication
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Version: 2.2.5.RELEASE
Created-By: Apache Maven 3.6.3
Build-Jdk: 1.8.0_121
Main-Class: org.springframework.boot.loader.JarLauncher
```

这里的Main-Class是org.springframework.boot.loader.JarLauncher，它是这个jar包启动的Main函数。还有一个Start-Class：com.test.demo.SpringbootDemoApplication，它是应用自己的Main函数。Spring Boot将jar包中的Main-Class进行了替换，换成了JarLauncher，并增加了一个Start-Class参数，这个参数对应的类才是真正的业务Main函数入口。我们再看看这个JarLaucher具体干了什么，源码如下：

```
public class JarLauncher{
    static void main(String[] args) {
        new JarLauncher.launch(args);
    }
    protected void launch(String[] args) {
    try {
        JarFile.registerUrlProtocolHandler;
        ClassLoader cl = createClassLoader(getClassPathArchives);
        launch(args， getMainClass， cl);
    }
    catch (Exception ex) {
    System.exit(1);
    }
}
```

launch方法分为三步：

（1）注册URL协议并清除应用缓存。

（2）设置类加载路径。

（3）执行main方法。

这里面，Spring Boot自定义的ClassLoader能够识别FatJar中的资源，包括：在指定目录下的项目编译Class、在指定目录下的项目依赖jar包。Spring Boot支持多个！/分隔符，通过自行实现的ZipFile解析器实现了对URL插入的定制化Handler，将获取的URL数据作为参数传递给自定义的URLClassLoader，最终实现资源的获取和解析。

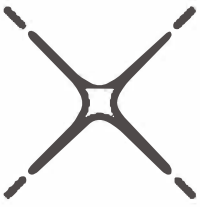
综上，在传统的以Tomcat容器部署War包项目中，我们的Web项目其实是一个被加载对象。Tomcat容器作为主线程的父类加载器来加载不同的应用，Tomcat独特的WebApp ClassLoader各自加载不同目录下的War包应用，应用之间使用ClassLoader实现了很好的隔离。

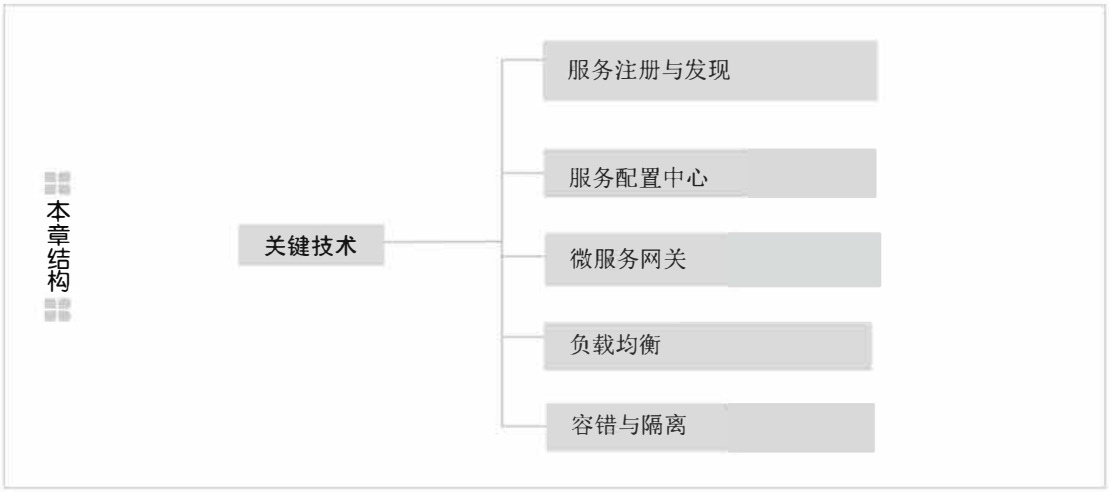
Spring Boot主要通过实例化SpringApplication来启动应用，内置的Tomcat容器实现相关Web环境及初始化资源准备，并将Tomcat内嵌的WebApp ClassLoader作为子ClassLoader挂载到Spring Boot的主线程Context ClassLoader。同时，Spring Boot中的@Controller、@RequestMapping等Web服务资源通过自动装配机制，在SpringApplication启动过程中通过扫描将资源对象加载到Spring IoC容器中。最后Spring Boot使用FatJar自定义的jar包压缩和加载机制，规范了Spring Boot项目的包及目录结构。

## 4.5 小结

目前，基于脚手架（基底）模式进行软件构建已经成为微服务架构落地的主流开发方式，可以显著提升开发人员的工作效率。Spring Boot本身基于Spring框架，继承了Spring强大的技术特性。本章我们对Spring Boot框架的核心模块和机制进行了剖析，详细讲解了Spring Boot的自动化配置原理、Starter机制和自定义Starter的工作原理，固化了“约定优于配置”和“开箱即用”等简洁的开发理念和高效开发方式。同时，本章也是后续Spring Cloud微服务治理的基础，在开始技术进阶之前，务必掌握Spring Boot基础原理，这样才能做到事半功倍。

关键技术






在微服务架构中，帮助开发者快速构建应用的脚手架技术无疑是非常重要的。以Spring Boot为代表的基底技术在继承了Spring框架思想的同时将简洁便利、约定优于配置、开箱即用等特性进一步发扬光大。然而仅仅依靠Spring Boot还不足以支撑微服务架构应对服务高可用、服务动态配置、服务高可扩展、服务负载均衡、服务容错与隔离等非功能需求，我们还需要相关基础设施提供服务治理及管控能力。

Pivotal公司的Spring Cloud可以说是JVM平台上微服务治理框架的集大成者。本章我们将详细讲解服务注册中心、服务配置中心、微服务网关三个微服务运行时的关键支撑系统。另外，将介绍Spring Cloud组件依赖Ribbon及Hystrix模块如何实现负载均衡和熔断管理等。

## 5.1 服务注册与发现

在云原生架构下，微服务需要具备极强的动态性及可扩展性，而服务注册与发现机制正是微服务可扩展性的基础。在微服务体系中，服务注册中心是微服务的核心模块，它是微服务架构中对服务的位置信息、心跳信息、元数据信息进行管理的重要基础设施。服务注册中心通过中心化、动态化的方式管理众多微服务实例。

### 5.1.1 服务注册与发现原理

在传统的应用系统中，基于硬编码方式对外提供服务地址的方式存在诸多问题，在服务扩展、服务高可用、服务升级方面，如果使用硬编码方式将使服务调用方与服务提供方产生紧耦合问题。

在早期的分布式系统架构中，使用DNS（Domain Name Server，域名服务器）协议作为服务发现机制，它是实现服务调用方与服务提供方解耦的一种简单有效的方式。DNS协议是一个“古老”的协议，也是最基本、最通用的协议之一。几乎所有操作系统底层都支持DNS协议，基于DNS协议的服务发现具备非常好的通用性，几乎使用所有编程语言都可以无缝接入。DNS可以让我们的一个服务名称与一组IP地址关联，或者与一个负载均衡器关联，实现多实例的分发。但是基于DNS协议的服务发现机制还是存在灵活性差、无法定制、端口及语言框架等问题。

在云原生范式的微服务架构中，服务的地址信息、服务的描述信息、服务的健康状态、服务的心跳信息、服务的版本、服务的元数据都是服务注册中心关心的初始数据。所以越来越多的基于“私有”协议的服务注册中心涌现出来。后面我们会详细介绍一些目前比较流行的微服务架构的服务注册中心。

下面我们首先来了解一下服务注册与服务发现两者之间的关系。服务注册是生产者将自己的服务元信息上传到服务注册表中的过程，而服务发现是一个消费者通过服务注册表实时获取可用生产者服务信息的过程。

**服务注册方式**

- 自注册：顾名思义就是服务提供方在启动服务时自己把提供服务的IP和端口发送到注册中心，并通过心跳的方式维持健康状态；服务下线时，自己把相应的数据删除。典型的案例是

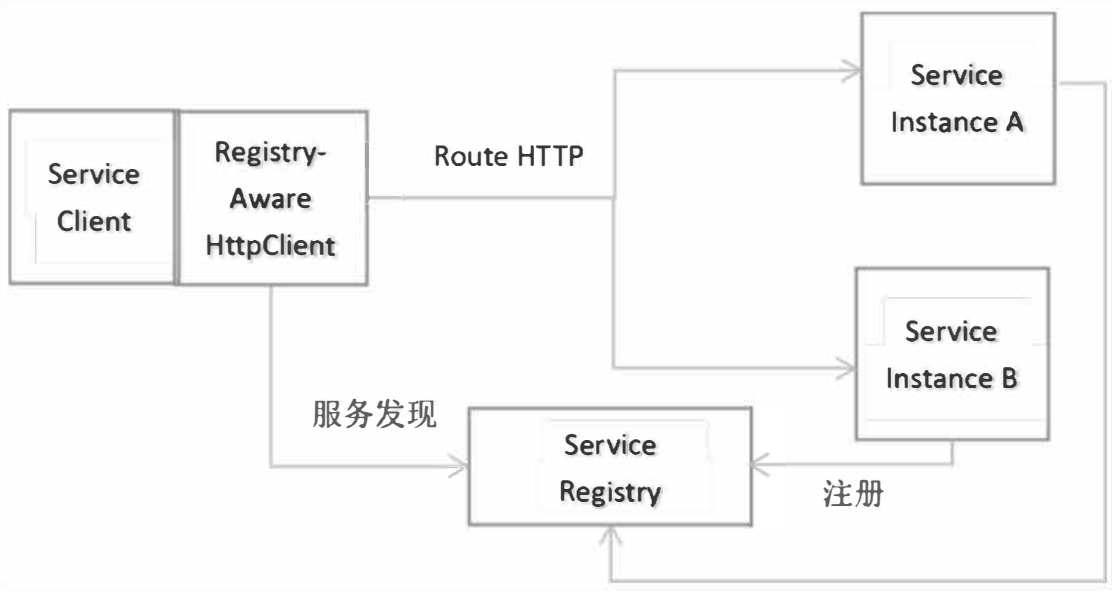
使用Eureka客户端发布微服务。

- 第三方注册：存在一个第三方的系统负责在服务启动或停止时向注册中心增加或删除服务数据。Netflix Prana就是一款第三方注册系统，可与非JVM应用共同运行，并利用Eureka为该应用注册。
- 注册中心主动同步：是指将注册中心和调度或发布系统打通，注册中心主动同步最新的服务IP列表。在Kubernetes体系中，Core DNS订阅API Server数据就是采用这种方式实现的。

**服务发现方式**

- **客户端服务发现**

在向某一服务发送请求时，客户端会通过查询服务注册表（Service Registry）获取该服务实例的位置，该注册表中包含所有服务的位置。下图展现了这种模式的结构。




客户端服务发现拥有以下优势：

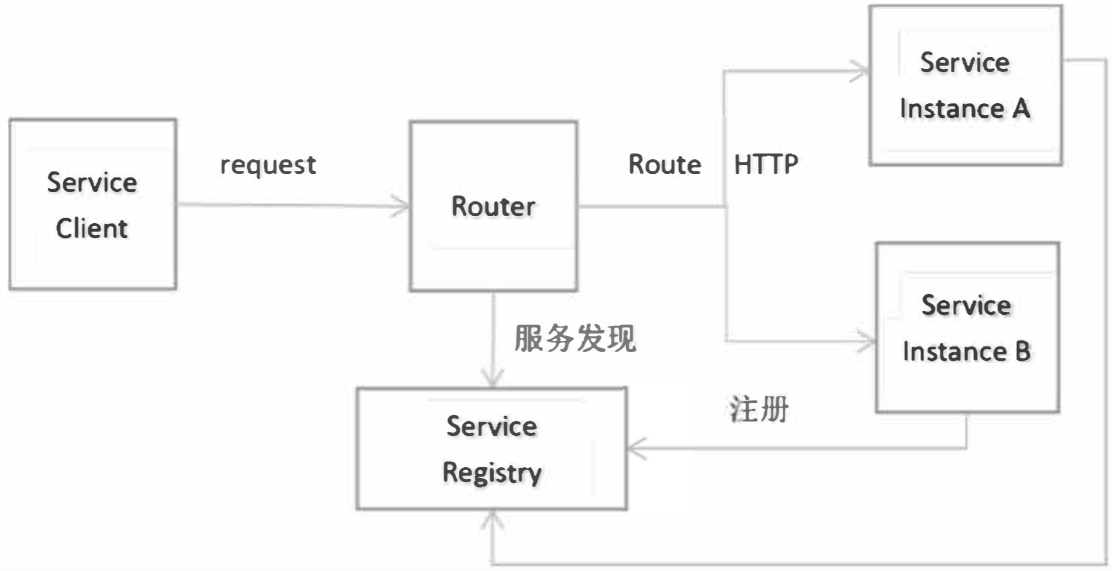
○ 相较于服务端服务发现，活动部件与网络中转数量更少。
○ 客户端可以根据自己的策略实现负载均衡，有掌控优势。

客户端服务发现存在以下弊端：

○ 客户端与服务注册表耦合。
○ 需要为应用程序中使用的每种编程语言或框架建立客户端服务发现逻辑。举例来说，Netflix Prana就为非JVM客户端提供了一套基于HTTP代理的服务发现方案。

- **服务端服务发现**

在向某一服务发送请求时，客户端会通过在已知位置运行的路由器（或者负载均衡器）发送请求。路由器会查询服务注册表，并向可用的服务实例转发该请求。服务注册表也可能内建于路由器中。下图展现了这种模式的结构。




服务端服务发现拥有以下优势：

○ 相较于客户端服务发现，其客户端由于无须实现发现功能，对代码没有侵入，而且客户端只需要向路由器发送请求即可。

○ 可以解耦客户端与服务端的依赖关系。

服务端服务发现存在以下弊端：

○ 路由机制作为另一系统组件进行安装与配置，不仅需要具备一定的接入能力，还需要为其配置一定数量的副本。

○ 相较于客户端服务发现，服务端服务发现需要更多的网络跳转。

## 5.1.2 微服务注册中心技术选型

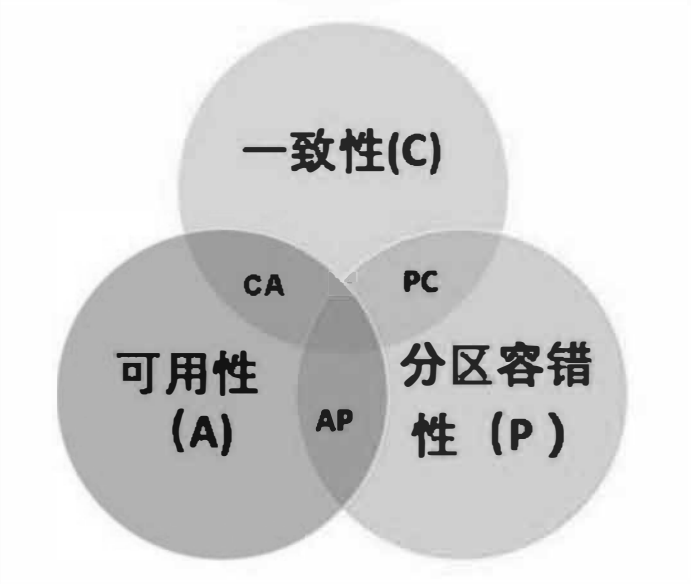
技术选型可以说是软件开发人员经常面临的问题，技术选型的首要原则是满足业务场景的需要，《人月神话》中提到软件的本质复杂性在于复杂的业务领域，而技术仅仅是辅助工具，技术选型解决的问题是将业务领域问题转换为软件实现。除此之外，从技术的层面上来说，我们还需要考虑如下关键因素。

- 软件的成熟度及软件架构的完整性。不要盲目追求新技术，要采用成熟稳定的技术，在生产环境中经过长时间运行检验的技术可减小我们的使用风险。
- 软件工程性方面。需要考虑公司人员擅长的技术，如果公司目前的业务系统大部分都使用Java相关技术，迁移到以Java为核心的服务框架或者技术工具会更加容易，且可以降低学习成本。
- 社区活跃度。软件在开源之后，社区越活跃说明有越多的人关注和使用该软件，你也可以在遇到技术问题后得到更多的帮助。

随着目前微服务架构越来越流行和成熟，作为微服务架构的核心模块，各种第三方注册中心如雨后春笋般涌现，很多开源产品都实现了服务注册与发现的功能，如何选择一个适合你的服务注册中心呢？下面我们以三款主流的注册中心（ZooKeeper、Consul、Eureka）为例，开始我们的详细说明和分析。

**注册中心与CAP**

CAP理论告诉我们，一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项，如下图所示。




- 一致性：指所有节点在同一时刻的数据完全一致。
- 可用性：指服务一直可用，而且响应时间正常。例如，不管什么时候访问X节点和Y节点都可以正常获取数据值，而不会出现问题。
- 分区容错性：指在遇到某节点或网络分区故障时，仍然能够对外提供满足一致性和可用性的服务。例如X节点和Y节点出现故障，但是依然可以很好地对外提供服务。

**CAP的取舍**

- 满足CA舍弃P，也就是满足一致性和可用性，舍弃分区容错性。这也就意味着你的系统不是分布式的了，因为分布式就是把功能分开部署到不同的机器上。
- 满足CP舍弃A，也就是满足一致性和分区容错性，舍弃可用性。这也就意味着你的系统允许有一段时间访问失效等，不会出现数据不一致的情况。
- 满足AP舍弃C，也就是满足可用性和分区容错性，舍弃一致性。这也就意味着你的系统在并发访问的时候可能会出现数据不一致的情况。

事实证明，在分布式系统中，为了避免单点故障，分区容错是不可避免的，所以对于注册中心来说只能从CP（优先保证数据一致性）、AP（优先保证数据可用性）中根据你的业务场景选择一种。

对于服务注册与发现场景来说，针对同一个服务，即使注册中心的不同节点保存的服务提供者信息不尽相同，也不会造成灾难性的后果。对于服务的消费者来说，能消费才是最重要的，拿到可能不正确的服务实例信息可以通过重试的方法再次获取，总比因为无法获取实例信息而无法消费好。所以，对于服务发现而言，可用性比数据一致性更加重要，即AP胜过CP。Spring Cloud Netflix在设计Eureka时遵守的就是AP原则。

- Eureka：使用AP原则、无Master/Slave（主/备）节点之分，一个节点挂了，自动切换到其他节点，实现了去中心化，它优先保证了服务的可用性。Eureka使用P2P（Pear to Pear，点对点）的对等通信原则，每一个Peer都是对等的。在这种架构风格中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的URL指向其他节点。每个节点都可被视为其他节点的副本。
- Consul：使用CP原则，采用分布式一致性协议实现健康检查、链值对存储。为了维护数据的一致性，通常需要选举出一个Leader（领导者）来进行协调，Consul的Raft协议要求必须过半数的节点都写入成功才认为注册成功。在Leader“挂掉”之后、重新选举出Leader之前Consul服务不可用。
- ZooKeeper：使用CP原则，它可以保证服务的一致性。搭建集群的时候，如果某个节点失效，则会进行Leader选举，或者半数以上节点不可用则无法提供服务，因此可用性没法满足。ZooKeeper使用Paxos算法保证数据的一致性。

**服务侵入性差别**

Eureka属于应用内的注册方式，对应用的侵入性比较强，而且目前主要支持基于Java语言方式的接入。不过Eureka使用HTTP方式接入，降低了异构系统接入Eureka的难度，如果需要使用Eureka注册中心，可以使用SideCar（边车）模式完成。

Consul属于应用外注册方式，本质上把应用当成一个黑盒，通过consul-template和envconsul等工具集成需要应用程序进行的更改，避免了对语言的侵入性。

ZooKeeper属于应用内的注册方式，对跨语言支持较弱，如果需要接入ZooKeeper，则需要使用对应语言的SDK方式支持，对语言的侵入性比较大。

**服务状态监听差别**

Eureka使用客户端心跳的方式实现服务状态的监控。同时Eureka为了在网络不稳定情况下保护服务地址，可以通过配置开启自我保护功能。使用“/health”端点服务可以获得更多本地服务健康状态数据，同时Eureka也支持自身监控。

Consul提供了系统级和应用级健康检查。一种方式是服务主动向Consul告知健康状态，还有一种是服务被动方式，由Consul主动来检查服务端状态，可以查看服务状态、内存、硬盘等健康指标数据，Consul也支持自身监控。

ZooKeeper通过长连接和KeepAlive的方式检查服务进程和连接情况。同时ZooKeeper提供Watch机制监听对象变化事件，实现分布式的通知功能。当触发服务端的一些指定事件时，它就会向客户端发送通知，但是ZooKeeper本身不支持自身监控。

上述三服务注册中心都支持TTL（Time To Live，生存时间）机制。也就是说如果客户端在一定时间内没有向注册中心发送心跳，则服务注册中心会将这个客户端摘除。

**易用性差别**

Eureka和Consul都是专门设计作为注册中心使用的，所以在服务模型和API封装方面比较友好，同时在应用协议方面，Eureka的定位就是注册中心，支持HTTP协议，第三方接入也相对容易。Consul同时支持HTTP和DNS，在协议层面Consul更通用，Consul扩展了很多功能，除了注册中心，它也支持作为配置中心、键值对存储、丰富形式的健康检查、多数据中心、ACL控制、日志快照生成等功能。

ZooKeeper基于TCP上的私有协议完成服务注册与发现，ZooKeeper的客户端使用起来比较复杂，ZooKeeper自己的定位是分布式协调器，也扩展了很多的功能：分布式锁、Watcher通知机制、分布式下的队列管理、键值对存储，没有服务发现模型及想用的API封装，所以ZooKeeper作为单纯的服务注册中心的易用性比较差。

**编程语言实现差别**

Eureka与ZooKeeper都是基于Java实现的，定制开发需要使用Java，与Spring Cloud的生态圈有先天的良好的适配性。Consul使用GO开发，开发定制需要使用GO，Spring Cloud封装了GO的客户端实现。

多数**据中心支持**

ZooKeeper和Eureka都不支持在多数据中心场景下的服务注册与发现，而Consul支持多数据中心。Consul通过WAN的Gossip协议完成跨数据中心的同步，而且其他的系统需要额外的开发工作来实现。

总结，对于技术选型而言，使用者选择一款开源软件产品，最重要的是从外部特征或者主要的功能特性考虑是否满足业务场景的需求。对于服务注册中心而言，上面总结的是ZooKeeper、Eureka、Consul的一些根本性差别，当然，这些注册中心在编程语言技术栈、数据存储模型、使用的一致性算法、访问协议等内部原理机制方面也都存在很大的不同。

## 5.1.3 Spring Cloud Eureka

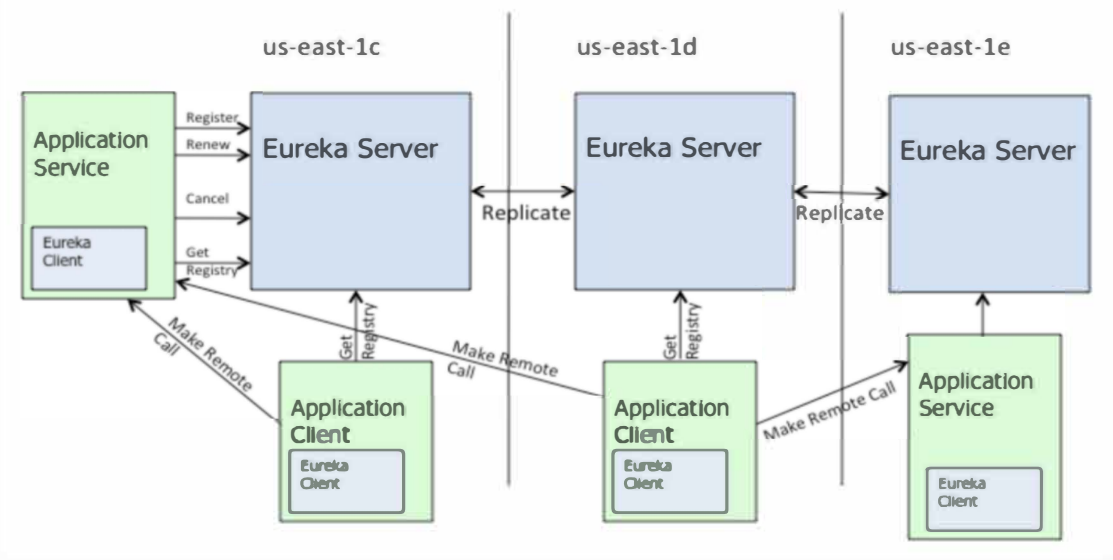
Eureka来自古希腊词语，含义是“我找到了！我发现了！”相传阿基米德发现浮力原理后说出了这个词。Eureka的官方介绍如下：

Eureka项目最早是由Netflix OSS开源的一个基于REST的服务发现组件，用来定位运行在AWS 注 中的后端服务。Eureka客户端是一个Java客户端，它可以用来简化与服务器的交互，作为轮询负载均衡器提供服务的故障切换支持。

Spring Cloud Eureka基于Eureka进行二次封装，增加了更人性化的UI，使用更方便。Eureka包括Eureka Server和Eureka Client两个模块。一般Eureka Server由多个Eureka Server实例组成，可避免单点问题，而Eureka Client又分为服务的生产者和服务的消费者。Eureka Server采用

点对点对等通信实现去中心化架构，每个Server节点需要添加有效的URL注册给其他Server节点，实现注册信息互相备份，达到高可用的效果。

如下图所示是Eureka的整体高可用架构图，从CAP理论看，Eureka是一个AP系统，优先保证可用性（A）和分区容错性（P），不保证强一致性（C），只保证最终一致性。




**Eureka Server接入步骤**

1.添加Eureka Server的依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

2.添加配置文件

```
server:
    port: 8881
spring:
    application:
    name: eureka-server
eureka:
    instance:
      preferIpAddress: true
      instance-id: ${spring.cloud.client.ipAddress}:${server.port}
    hostname: eureka-server
    client:
        registerWithEureka: true #把自身当作客户端注册到其他 Eureka 服务器
        fetchRegistry: true #在本地缓存所有实例注册信息
    serviceUrl:
        #设置与 Eureka Server 交互的地址，查询服务和注册服务都需要依赖这个地址
        defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

3.配置启动类

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class， args);
    }
}
```

**Eureka Client的接入步骤**

1.添加Eureka Client的依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

2.配置application.yml文件

```
server:
    port: 8086
spring:
    application:
    name: eureka-client
eureka:
    client:
    serviceUrl:
        #Eureka Server 的地址
        defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

3.配置Eureka Client启动类

```
@SpringBootApplication
@EnableEurekaClient
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class， args);
    }
}
```

**Eureka的主要配置参数**

Eureka配置参数众多，它的很多功能都是通过配置参数来实现的，了解这些参数的含义有助于我们更好地应用Eureka的各种功能。

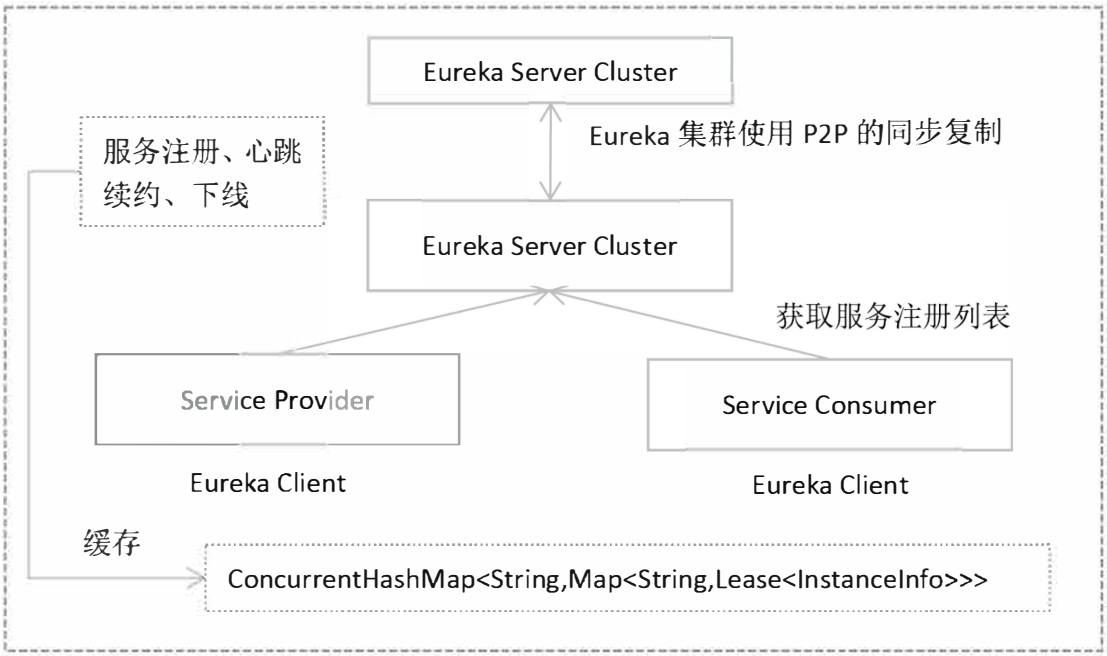
- Eureka Instance配置
  - instance-id：表示实例在注册中心注册的唯一ID。
  - prefer-ip-address：“true”实例以IP地址的形式注册，“false”实例以机器HostName的形式注册。
  - lease-expiration-duration-in-seconds：表示Eureka Server在接收到上一个心跳之后等待下一个心跳的秒数（默认为90s）。
  - lease-renewal-interval-in-seconds：表示Eureka Client向Eureka Server发送心跳的频率（默认为30s）。
- Eureka Server配置
  - enable-self-preservation：表示注册中心是否开启服务的自我保护能力，默认为true。
  - renewal-percent-threshold：表示Eureka Server开启自我保护的系数，默认为0.85。
  - eviction-interval-timer-in-ms：表示Eureka Server清理无效节点的频率，默认为60s。

- Eureka Client配置
  - register-with-eureka：表示此实例是否注册到Eureka Server以供其他实例发现。
  - fetch-registry：表示客户端是否从Eureka Server获取实例注册信息。
  - serviceUrl.defaultZone：表示客户端需要注册的Eureka Server的地址。

## 5.1.4 Eureka架构与设计原理

**Eureka的架构设计**

Eureka的架构主要包含两大部分：Eureka Server Cluster和Eureka Client Cluster，架构图如下。




Eureka使用P2P（点对点）的心跳机制保证Eureka Client启动后向Eureka Server发送心跳，默认周期是30s，如果Eureka Server在多个心跳周期内没有接收到心跳，将会从服务注册中心剔除相应节点。其中LeaseManager是Eureka的核心模块，代码如下。

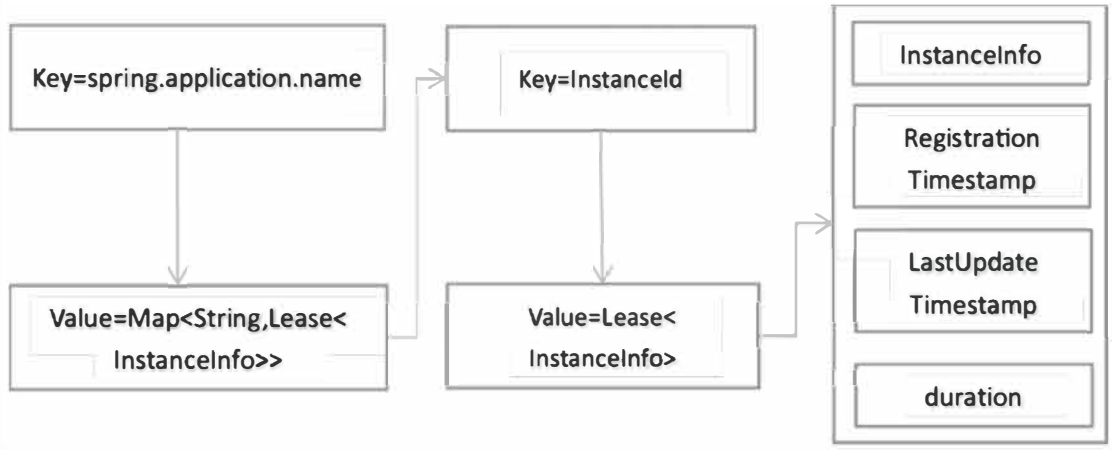
```
public abstract interface LeaseManager<T>
{
    public abstract void register(T paramT， int paramInt， boolean paramBoolean);
    public abstract boolean cancel(String paramString1， String paramString2， boolean
        paramBoolean);
    public abstract boolean renew(String paramString1， String paramString2， boolean
        paramBoolean);
    public abstract void evict();
}
```

Eureka心跳机制主要有以下几个操作方法。

- Register：用于实现服务注册实例信息接口。
- Cancel：用于删除服务实例信息接口。
- Renew：用于服务实例与Eureka Sever完成远程心跳，维持续约接口。
- Evict：是服务端的一个方法，用来剔除过期的服务实例的接口。

### Eureka的数据模型

如下图所示是Eureka Server保存注册实例信息的数据结构，可以看到它是一个双层Map结构。

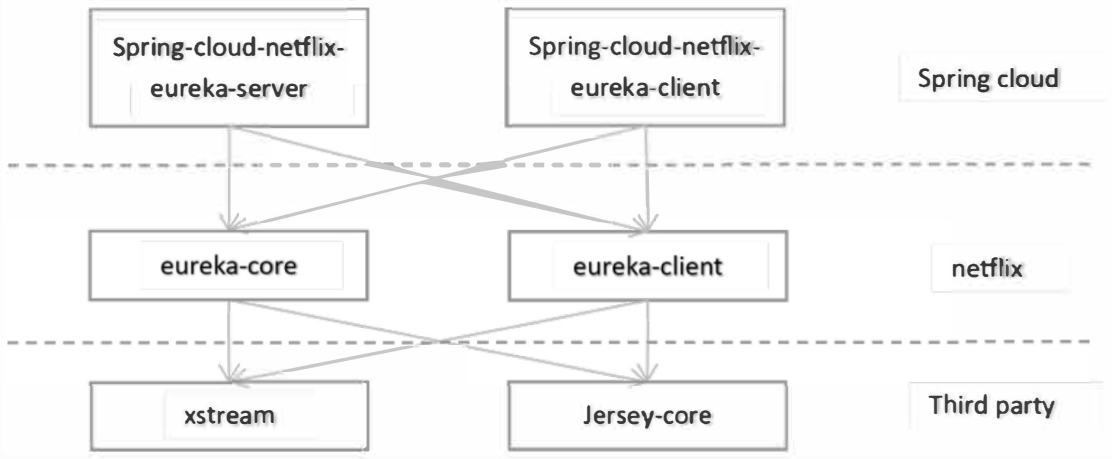



- 第一层ConcurrentHashMap，Key值是spring.application.name，Value值是一个Map数据结构。
- 第二层ConcurrentHashMap，Key值是InstanceId，Value值是Lease对象，Lease的实现代码如下：

```
public Lease(T r,  int durationInSecs)
{
    this.holder = r;
    this.registrationTimestamp = System.currentTimeMillis();
    this.lastUpdateTimestamp = this.registrationTimestamp;
    this.duration = (durationInSecs * 1000);
}
```

### Eureka工程中的包层级关系

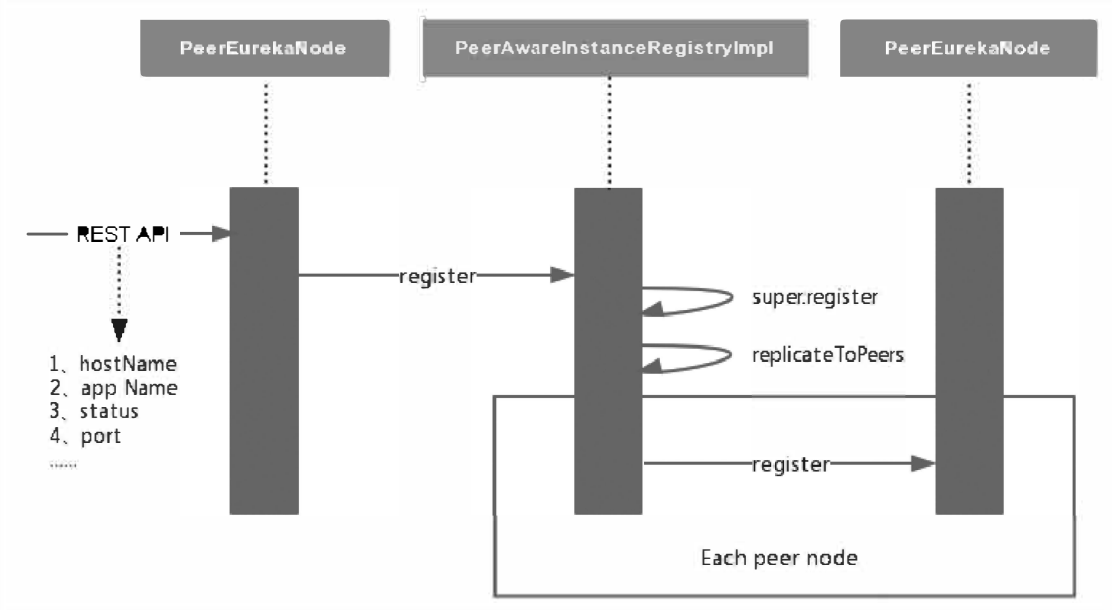
如下图所示，eureka-core和eureka-client是Netflix用来实现服务注册与发现功能的核心包，所有服务的注册与发现都是在包中实现的。



Eureka使用Jersey和XStream第三方库组件配合JSON作为Server和Client之间的通信协议。spring-cloud-netflix-eureka-server和spring-cloud-netflix-eureka-client是Spring Cloud借助自动配置管理机制，将Eureka资源加载到Spring IoC容器，实现的对Eureka功能的封装和管理。可见，Spring Cloud就像一个“八爪鱼”一样，可以轻松集成其他组件。

**Eureka的心跳流程图**

Eureka使用心跳机制维持与Server的连接，主要包含注册、续约、退出、剔除等服务生命周期相关操作。Server节点复制的流程图如下所示。

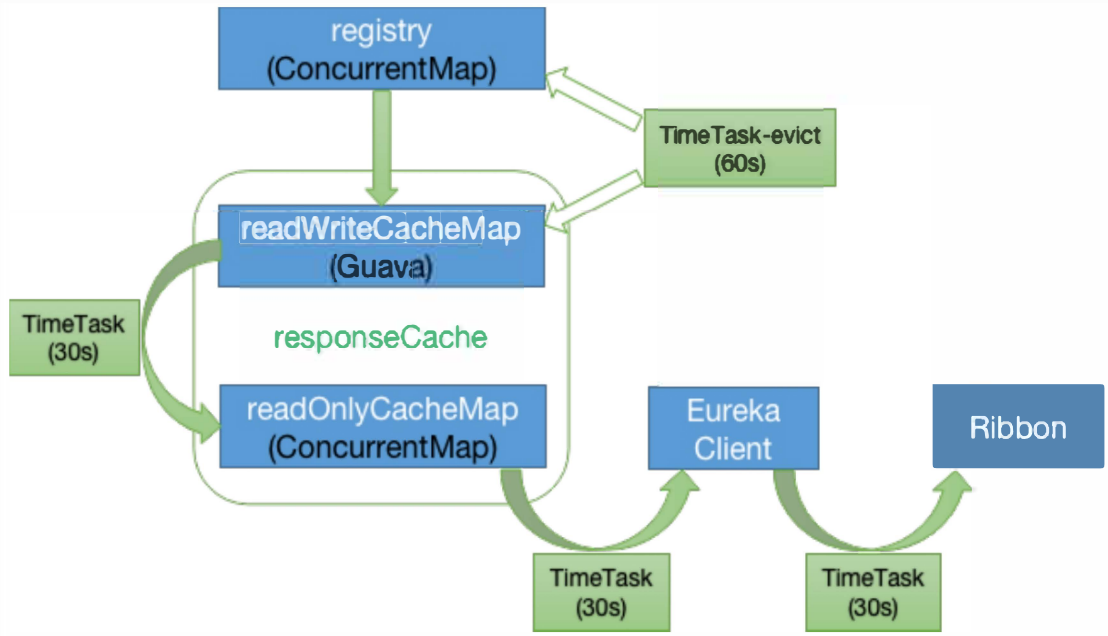



Eureka通过REST接口实现对外服务暴露，所有接口在ApplicationResource中，ApplicationResource会调用PeerAwareInstanceRegistry的接口。Eureka注册中心的核心生命周期管理入口正是InstanceRegistry，它继承了LookupService、LeaseManager接口，提供应用实例的注册与发现服务。AbstractInstanceRegistry是应用对象注册表的抽象实现，其中定义了注册、续约、退出、剔除等方法，贯穿于服务实例在注册中心Server中的整个生命周期管理，而PeerAwareInstanceRegistry提供Eureka Server集群内注册信息的同步服务接口。PeerAwareInstance-RegistryImpl继承了抽象注册表类AbstractInstanceRegistry，新增了peer相关处理和RenewalThreshold的更新。下面是核心代码示例：

```
    public abstract class AbstractInstanceRegistry implements InstanceRegistry
    {
        public void register(InstanceInfo registrant， int leaseDuration， boolean isReplication)
        {
        //省略
        }
    }
    @Singleton
    public class PeerAwareInstanceRegistryImpl extends AbstractInstanceRegistry
        implements PeerAwareInstanceRegistry
    {
        public void register(InstanceInfo info， boolean isReplication)
        {
            int leaseDuration = 90;
            if ((info.getLeaseInfo() != null) && (info.getLeaseInfo().getDurationInSecs() > 0)) {
            leaseDuration = info.getLeaseInfo().getDurationInSecs();
            }
            super.register(info， leaseDuration， isReplication);
          replicateToPeers(Action.Register， info.getAppName()， info.getId()， info， null，
isReplication);
        }
        //省略
    }
```

## 5.1.5 Eureka缓存机制

Eureka Server存在三个缓存变量：registry、readWriteCacheMap和readOnlyCacheMap，用于保存服务注册信息，默认情况下定时任务每30s将readWriteCacheMap同步至readOnlyCacheMap一次，每60s清理一次超过90s未续约的节点，Eureka Client每30s从readOnlyCacheMap更新服务注册信息一次，前端则从registry更新服务注册信息。如下图所示是和Eureka缓存机制相关的服务和数据结构的关系。




Eureka Server三级缓存组件如下表所示。

| 缓存 | 类型 | 说明 |
| --- | --- | --- |
| registry | ConcurrentHashMap | 实时更新，AbstractInstanceRegistry类的成员变量，前端请求的是这里的服务注册信息 |
| readWriteCacheMap | Guava Cache/LoadingCache | 实时更新，ResponseCacheImpl类的成员变量，缓存时间是180s |
| readOnlyCacheMap | ConcurrentHashMap | 周期更新，ResponseCacheImpl类的成员变量，默认为30s，Eureka Client默认从这里更新服务注册信息，可以配置直接从readWriteCacheMap更新 |

Eureka Server缓存相关配置如下表所示。

| 配置 | 默认 | 说明 |
| --- | --- | --- |
| eureka.server.useReadOnlyResponseCache | true | Client从readOnlyCacheMap更新数据，如果为false则跳过。readOnlyCacheMap直接从readWriteCacheMap更新 |
| eureka.server.responsecCacheUpdateIntervalMs | 30000 | readWriteCacheMap更新至readOnlyCacheMap，默认为30s |
| eureka.server.evictionIntervalTimerInMs | 60000 | 清理未续约节点（evict）周期，默认为60s |
| eureka.instance.leaseExpirationDurationInSeconds | 90 | 清理未续约节点超时时间，默认为90s |

对于Eureka Client来说，它有两种角色：服务生产者和服务消费者，一般会配合Ribbon或Feign使用。在启动后，Eureka Client作为服务生产者会立即向Server注册，暴露服务地址供消费者使用，默认情况下每30s续约（renew）一次；作为服务消费者它立即向Server全量更新服务注册信息，默认情况下每30s增量更新服务注册信息一次；Ribbon延时1s向Client获取使用的服务注册信息，默认每30s更新一次使用的服务注册信息，只保存状态为UP的服务。

Eureka Client二级缓存组件如下表所示。

| 缓存 | 类型 | 说明 |
| --- | --- | --- |
| localRegionApps | AtomicReference | 周期更新，DiscoveryClient类的成员变量，Eureka Client保存服务注册信息，启动后立即向Server全量更新，默认每30s增量更新一次 |
| upServerListZoneMap | ConcurrentHashMap | 周期更新，LoadBalancerStats类的成员变量，Ribbon保存使用且状态为UP的服务注册信息，启动后延时1s向Client更新，默认每30s更新一次 |

Eureka Client缓存相关配置如下表所示。

| 配置 | 默认 | 说明 |
| --- | --- | --- |
| eureka.instance.leaseRenewalIntervalInSeconds | 30 | Eureka Client续约周期，默认为30s |
| eureka.client.registryFetchIntervalSeconds | 30 | Eureka Client增量更新周期，默认为30s（正常情况下为增量更新，如果出现超时或与Server端不一致等情况则全量更新） |
| ribbon.ServerListRefreshInterval | 30000 | Ribbon更新周期，默认为30s |

总结，因为Eureka基于AP原则，服务端和客户端的实例状态信息可能会有不一致的情况，我们需要了解Eureka的缓存机制，以便于了解服务状态同步异常问题的原因，也可以通过修改相应配置参数缓解缓存不一致问题。对于需要保证强一致性的服务场景，建议使用ZooKeeper。

**配置优化**

为了避免Eureka Server和Client缓存问题，一般会配置多节点集群，这样单个节点服务上线的状态更新滞后并没有什么影响。

下面总结一些配置优化，用来改善数据一致性和状态感知滞后引起的状态不一致问题。

- 服务端：缩短readOnlyCacheMap更新周期，可减少滞后时间。

```
eureka.server.responsecCacheUpdateIntervalMs: 10000
```

- 服务端：关闭readOnlyCacheMap缓存，让Eureka Client直接使用readWriteCacheMap更新服务注册信息。

```
eureka.server.useReadOnlyResponseCache: false
```

- 客户端：缩短服务消费者更新周期。Eureka Client和Ribbon二级缓存影响状态更新，缩短这两个定时任务的周期可减少滞后时间，例如配置定制化开发：

```
eureka.client.registryFetchIntervalSeconds: 5
ribbon.ServerListRefreshInterval: 2000
```

### 5.1.6 Eureka定制化开发

Eureka Server作为服务注册中心可能有不一样的定制化需求，这个时候需要在Eureka Server开源工程的基础上完成定制化开发。

在默认情况下，Eureka会展示Eureka Server集群中所有的注册服务实例，没有对用户做权限控制，我们可以根据不同用户的权限在Admin管理端做定制化开发，展示该用户所属权限下的服务实例状态数据，示例代码如下：

```
/**
 * @param 前端传给 Eureka 的用户组权限
 * @return 返回用户组权限的服务实例
 */
public List<Application> queryNodesByGroupName(List<String> groupNames) {
    Applications apps =
            EurekaServerContextHolder.getInstance().getServerContext().
                getRegistry().getApplications();
    List<com.netflix.discovery.shared.Application> aps = apps.getRegisteredApplications();
    if (groupNames == null) {
        LOGGER.info("groupNames is null, return none apps");
        return null;
    }
    //可以根据用户的角色信息返回对应的应用服务信息
    if (groupNames.contains(SystemConstants.ADMIN_ROLE)) {
        LOGGER.info("groupNames contains admin, return all apps");
        Collections.sort(aps, APP_COMPARATOR);
        return aps;
    }
    List<Application> applications = new ArrayList<Application>();
    for (String groupName : groupNames) {
        for (com.netflix.discovery.shared.Application ap : aps) {
            if (ap.getName().startsWith(groupName.toUpperCase())) {
                applications.add(ap);
            }
        }
    }
    Collections.sort(applications, APP_COMPARATOR);
    return applications;
}
```

另外，在具备定制化权限管理的基础上，我们可以通过覆盖Eureka自带的上下线操作方法，实现对服务实例的下线或者上线操作，它可以完成服务实例的灰度发布，示例代码如下：

```
/**
 * @param appName
 * @param id
 */
public boolean setOutOfService(String appName, String id) {
    PeerAwareInstanceRegistry peerAwareInstanceRegistry =
            EurekaServerContextHolder.getInstance().getServerContext()
            .getRegistry();
    return peerAwareInstanceRegistry.statusUpdate(appName, id,
        InstanceInfo.InstanceStatus.OUT_OF_SERVICE, null, false);
}
/**
 * @param appName
 * @param id
 */
public boolean setUp(String appName, String id) {
    PeerAwareInstanceRegistry peerAwareInstanceRegistry =
        EurekaServerContextHolder.getInstance().getServerContext().getRegistry();
    return peerAwareInstanceRegistry.statusUpdate(appName, id,
        InstanceInfo.InstanceStatus.UP, null, false);
}
```

## 5.2 服务配置中心

服务配置中心是对微服务进行集中式配置管理的重要机制。集中配置管理可以分离应用代码与不同环境下的配置信息，实现应用“一次打包、随处运行”，通过这种外部化的配置管理还可以实现配置修改实时生效、灵活的权限及安全管理等特性。

### 5.2.1 服务配置中心管理

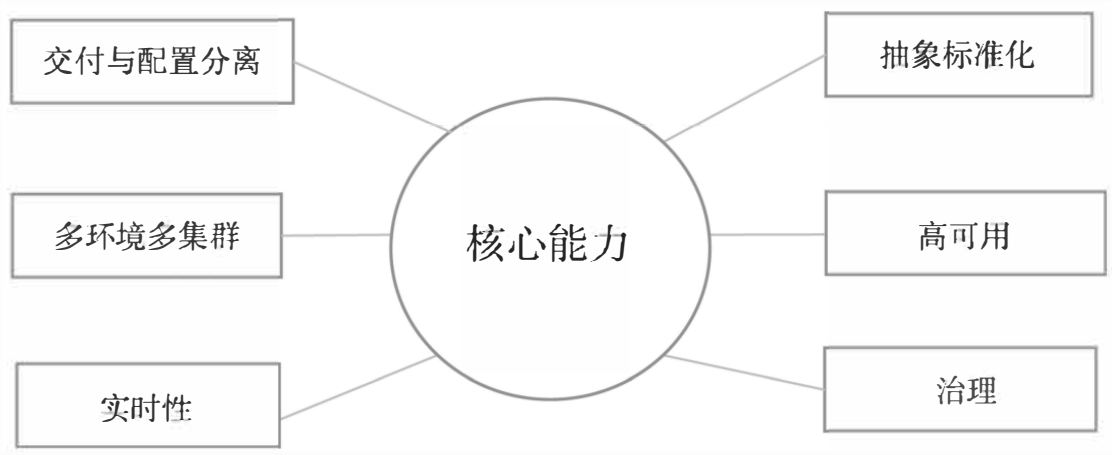
在传统的中心化单体架构中，所有的配置项都是通过本地的静态配置文件进行管理的，对于不同的环境（开发、测试、生产），我们需要手动维护和切换调整不同的配置。在分布式微服务环境下，一个系统往往又由多个微服务组成（ X 个），每个微服务都需要独立的配置文件（ Y 个），而分布式部署又会有多个机器或者容器（ Z 个），那么在这种情况下，如果使用静态配置管理，我们需要同时管理 X × Y × Z 个配置文件，这样无疑是非常低效并且容易出错的。

所以我们需要将各种配置参数全部放到一个集中的地方（服务配置中心，简称配置中心）进行统一管理，并提供一套标准的接口规范。当不同服务需要获取参数时，可以从配置中心拉取和配置，当修改配置时，可以由配置中心统一下发给集群中的所有实例。配置中心可以解决传统的配置文件的如下问题：

- 服务修改不灵活。配置中心具备“实时更新”的功能，它可以用来解决传统的服务修改不灵活问题。当线上系统需要调整参数的时候，只需要在配置中心动态修改即可。
- 配置文件无法区分环境。通过“配置与应用分离”可以解决传统的配置文件无法区分环境问题，配置并不跟着环境走，当不同环境有不同需求的时候，就到配置中心获取，可降低运维成本。
- 配置文件过于分散。采用“配置集中管理”可解决传统的配置文件过于分散的问题。所有的配置都集中在配置中心管理，不需要每个项目都自带一个配置文件，降低了开发成本。
- 配置修改无法追溯。采用静态配置文件，当配置进行修改后，不容易形成记录，更无法追溯是谁修改的、什么时间修改的、修改前的内容是什么。当配置出错时，更没办法回滚。配置中心可以统一记录所有更改记录，用于后续审计管理。

**配置中心的核心能力**

如下图所示是配置中心的核心能力。




- 交付与配置分离：在打包部署时，传统应用系统会为不同环境打出不同配置包，例如为开发、测试、生产环境分别制作发布包，每个包里包含特定配置。现代微服务提倡云原生（Cloud Native）和不可变基础设施（Immutable Infrastructure）的理念，推荐采用容器镜像这种方式打包和交付微服务，应用镜像一般只打一个包，可以部署到不同环境。这就要求交付物（比如容器镜像）和配置分离，交付件只制作一份，并且是不可变的，可以部署到任意环境，而配置由配置中心集中管理，所有环境的配置都可以在配置中心集中配置，运行期应用根据自身环境到配置中心动态拉取相应的配置。
- 抽象标准化：企业应该由框架或者中间件团队提供标准化的配置中心服务，封装配置管理的细节和配置的不同格式，方便用户进行自助式的配置管理。一般用户只需要关注两个抽象和标准化的接口：
  - 配置管理界面UI，方便应用开发人员管理和发布配置。
  - 封装好的客户端API，方便应用集成和获取配置。
- 多环境多集群：现代微服务应用大都采用多环境部署，一般标准化的环境有开发、测试、生产等，有些应用还需要多集群部署，例如支持跨机房或者多版本部署。配置中心需要支持对多环境和多集群应用配置的集中式管理。
- 高可用：配置中心必须保证高可用，否则可能影响大量微服务的正常启动或者配置更新。在极端的情况下，如果配置中心不可用，客户端也需要有降级策略，保证应用不受影响。
- 实时性：配置更新需要尽快通知到客户端，这个周期不能太长，理想状态下应该是实时的。有些配置的实时性要求很高，例如主备切换配置或者蓝绿部署配置，需要具有秒级切换配置的能力。
- 治理：配置中心需要具有治理能力，具体包括：可进行人员登录操作记录追溯；提供配置版本控制，出现问题时能够及时回滚到上一个版本；提供配置权限控制，发布配置变更需要认证授权，不是所有人都能修改和发布配置；支持灰度发布，发布配置时可以先让少数实例生效，确保没有问题再逐步放量。

**配置中心的其他优势**

- 实现实时的配置读取、更新、取消。

- 权限控制，能够根据公司的SSO或者LDAP基础设施进行配置权限管理。
- 环境管理，开发、测试、生产环境下的配置可以做隔离处理。
- 配置回滚，当发现配置错误或者在该配置下程序发生异常时可以立即回滚到之前的版本。
- 灰度发布，有时候我们新上线一个功能，想先通过少部分流量测试一下，我们可以随机只修改部分应用的配置，测试正常后再将功能推送到所有的应用。

### 5.2.2 Spring Cloud Config

Spring Cloud Config为分布式系统配置提供了服务端和客户端的支持，包括Config Server和Config Client两部分。Spring Cloud Config通过配置服务（Config Server）来为所有的环境和应用提供外部配置的集中管理，它适用于各类Spring应用，也能对应用的开发、测试、生产环境的配置做切换、迁移。

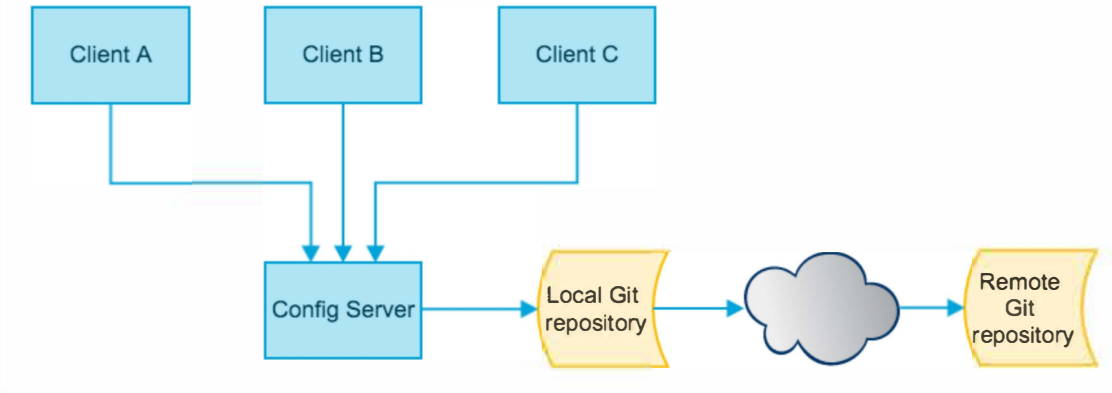
使用Spring Cloud Config Server，你可以在所有环境中管理应用程序的外部属性，还可以分离应用与配置文件，并且根据应用当前所处环境，动态地加载对应的配置文件，它符合“应用配置与代码隔离”的原则。

**Spring Cloud Config的主要特性**

- 提供服务端和客户端支持。
- 集中式、中心化管理分布式环境下的应用配置。
- 使用Git管理方式，天然具备版本控制能力。
- 基于Spring环境，实现了与Spring应用的无缝集成。
- 支持动态更新配置文件。
- 语言独立，可用于任何语言开发的程序。
- 默认基于Git仓库实现（也支持SVN、数据库、MongoDB），可进行配置的版本管理。

**Spring Cloud Config基本原理**

Config Server是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用Git存储配置内容（也可使用SVN、本地文件系统或Vault存储配置），因此可以实现对配置的版本控制与内容审计。如下图所示是Config Server的架构图。




- Config Client（Client A、Client B、Client C）：提供了基于Spring的客户端，应用只要在代码中引入Config Client的jar包即可工作。
- Config Server：需要独立部署的一个Web应用，它负责把Git上的配置返回客户端。
- Remote Git repository：即远程Git仓库，我们会把配置信息存储在一个远程仓库中，通过现成的Git客户端来管理配置。
- Local Git repository：即Config Server的本地Git仓库，Config Server接到来自客户端的配置获取请求后，先把远程仓库的配置克隆到本地的临时目录，然后从临时目录读取配置并返回。

**Spring Cloud Config的使用**

Spring Cloud Config基于HTTP，通过统一的配置服务中心进行集中化的远程配置文件管理。Config Client可以通过指定配置中心服务地址，主动从配置中心拉取服务的配置信息，完成配置获取，从而达到配置与代码分离的效果。

Spring Cloud Config默认使用Git的存储和管理方式，在采用Config作为生产和测试环境配置中心管理配置文件时，首选的存储方案也是使用Git。使用Git的主要优势如下：

- 它天然支持对配置文件的版本管理。
- Git默认提供Web界面的管理方式，方便用户从GitLab前端查看配置和管理配置，同时可以利用Git的权限管理给不同用户赋予不同的查看配置文件的权限。
- Git引擎还具备WebHook功能，它可以实现配置文件的动态更新等。
- 支持使用SVN、文件服务器、JDBC等方式获取配置文件。

**Spring Cloud Config客户端加载流程**

客户端从配置管理中获取配置的执行流程如下：

（1）应用启动时根据bootstrap.yml中配置的应用名{application}、环境名{profile}、分支名{label}，向Config Server请求获取配置信息。

（2）Config Server根据Config Client的请求及配置从Git仓库中查找并定位符合的配置文件。

（3）Config Server通过“git clone”命令将配置下载到Git本地文件系统，并建立缓存。

（4）Config Server创建Spring的ApplicationContext实例从Git本地仓库中加载配置文件，然后将读取的配置信息返回给Config Client。

（5）Config Client在获取到Config Server返回的配置数据后，将配置内容加载到客户端自己的ApplicationContext实例，该配置内容的优先级高于客户端内部的配置内容。

下面我们结合实例讲解一下基本的Config Server和Config Client的接入步骤。

**Config Server接入步骤**

在启动配置Config Server前，如果使用Git仓库存储配置文件，则需要先搭建Git仓库，篇幅所限，Git仓库搭建步骤省略。

1.引入Config Server Maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

2.启动配置中心服务，让客户端可以发现服务端，在启动类上加注解@EnableConfigServer

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class， args);
    }
}
```

3.添加配置中心配置文件

```
server:
  port: 6001 #服务端口
spring:
  application:
    name: spring-cloud-config-server #应用名
  cloud:
    config:
      server:
       git:
          uri: https://localhost/springcloudtest/springcloud-config-file.git
          searchPaths: test1-config-repo，test2-config-repo
          basedir: /tmp/spring-cloud-repo
          username:xxx
          password: *****
          force-pull: true
```

- spring.cloud.config.server.git.uri：配置的Git仓库地址。
- spring.cloud.config.server.git.searchPaths：与URI配合使用，定位Git库的子目录，指定搜索路径，如果有多个路径则使用“，”分隔。

- spring.cloud.config.server.git.basedir：使用Git作为后端配置，需要从远程库获取配置文件，存储到本地文件。默认存储在系统临时目录下，目录名的前缀为config-repo-，如在Linux下可能是/tmp/config-repo-。因为/tmp下的内容有可能被误删，为了保险，最好修改存储目录。如果要修改存储目录，可以修改spring.cloud.config.server.git.basedir参数。
- spring.cloud.config.server.git.force-pull：配置中心从远程Git仓库读取数据时，可能会出现本地的文件拷贝被污染的情况，这时配置中心无法从远程库更新本地配置。设置force-pull=true，可强制从远程库中更新本地库。
- spring.cloud.config.server.git.username：访问Git仓库的用户名。
- spring.cloud.config.server.git.password：访问Git仓库的用户密码。

4.引入Security配置

Config Server如果希望客户端能够授权访问配置中心，需要引入Maven依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

加入如下配置文件：

```
security:
  basic:
    enabled: true
  user:
    name: config-repo
    password: 123456
```

客户端需要增加如下配置来授权访问配置中心服务器：

```
spring.cloud.config.username=config-repo ， password=123456
```

下面看一下Config定义好的配置文件资源与URI的映射规则：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

需要注意的是：

- URI中的application对应是Config Client的应用名称，如果在Git中是用应用名来定义目录的，这个应用也对应你在Git中的目录名称。
- URI中的profile对应的是应用激活使用的环境名称，如果在Maven中指定Profile，可以在Maven中指定配置中通过标签<activation>指定Profile，也可以从配置文件中指定，一般Profile有test、dev、prod等。
- URI中label指的是Git的分支，默认是master分支。

5.测试验证

上述配置设置好后，可以启动Config Server，通过HTTP网络工具测试是否可以取得Config Server的配置信息。使用下面的地址访问配置中心：

```
http://localhost:6001 /spring-cloud-config-client/test/master
```

配置中心会返回如下响应：

```
{
    "name": "spring-cloud-config-client",
    "profiles": ["test"],
    "label": "master",
    "version": "92ed69d2975290af780852608bce35c71044234e",
    "state": null,
    "propertySources": [{
        "name":
           "springcloudtest/springcloud-config-file.git/file:/tmp/spring-cloud-repo
           /default/spring-cloud-config-client-test.yml",
        "source": {
           "spring.test.configKey": "this is git config info from test envirioment!"
      }
    }]
}
```

**Config Client接入步骤**

1.引入客户端依赖

```
<dependency>
     <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-config-client</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

**注意：** Maven依赖中需要加入对Actuator的支持，以实现Config Client中用一个refresh端点来实现配置文件的刷新功能。另外，在Config Client的配置文件中也需要打开相关Actuator端点的配置生效功能。

2.添加Config Client配置文件

```
server:
  port: 8008
spring:
  application:
    name: spring-cloud-config-client
  cloud:
    config:
      uri: http://localhost:6001  #配置中心地址
      profile: test    #配置环境，对应 Config Server URL 中的{profile}
       label: master   #配置分支
```

```
        failFast: true
      #激活定时任务，当 Git 版本发生变更时加载最新配置上下文
       watcher:
         enabled: true
     management:
        endpoints:
           Web:
             exposure:
                 include:refresh，health，info
```

- 上述配置文件中的uri对应配置中心的Server地址，label代表请求的是哪个Git分支，profile对应分支下的配置文件，如test、dev、prod。考虑到Spring Boot对配置文件的加载优先级，最好在bootstrap.yml文件里命名配置文件，因为加载远程配置文件的优先级会比加载本地配置文件高。
- spring.profile.active可以指定Spring Boot运行的环境，而spring.cloud.config.profile是客户端指定拉取资源库的profile，如果有多个profile，一般最后一个起作用。
- 如果我们希望在启动一个服务时若无法连接到服务端，能够快速返回失败信息，则可以通过failFast来设置开启Config Client快速失效功能。
- management：endpoints：web：exposure：include：“*”表示打开全部请求端点，要让Config Client支持动态刷新和查看配置端点信息，只需要打开部分端点。

3.启动Config Client类

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
       SpringApplication.run(Application.class， args);
    }
}
```

Config Client获取分布式配置信息主要有下面两种方式。

- 使用@Value方式获取配置文件

  这种方式是获取配置文件的常用方式，最简单，代码如下：

```
@Copmponent
public class RemoteConfig {
    @Value("${spring.test.configKey}")
    public String configKey;
}
```

- 通过创建一个远程配置信息Bean的方式获取配置信息

  首先定义一个映射远程配置信息的Bean类RemoteConfigProperties，该类中所有的属性名均需与配置中心的配置文件的内容相同，例如demo-config-profile-env，字段名为demoConfigProfileEnv，如果属性中有点，则应视点后面的部分为下级，应再定义相关的配置映射类，该类的完整代码如下：

```
@Component
@ConfigurationProperties(prefix=“spring.test”)//如果有前缀，则可以设置 prefix=XXX
public class RemoteConfigProperties {
   private String configKey
   public String getConfigKey() {
       return configKey
   }
   public void setConfigKey(String Key{
       this.configKey = configKey;
   }
}
```

**注意：** 属性字段名与配置项的名称应保持一致，若有下级则定义下级的配置类，需要根据配置类采用Java的内部类进行映射匹配。例如，配置文件如果是Map或List接口，则需要使用Java的对应数据接口存储映射配置文件。

## 5.2.3 Config Server配置详解

上一节我们介绍了基本的配置中心接入步骤。对于公司内使用的配置中心（Config Server），如果考虑同时支持不同部门的不同项目，需要考虑更多的Config Server管理Git的配置使用方式。

**Spring Cloud Config中的占位符**

Spring Cloud Config服务器支持一个Git仓库URI，其中包含{application}、{profile}及{label}的占位符，使用Git URI的占位符可以轻松支持“每个应用程序一个repo”的策略。当使用Git作为配置中心来存储各个微服务应用的配置文件时，URI中的占位符的使用可以帮助我们规划和实现通用的仓库配置，代码示例如下：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://localhost/myorg/{application}
```

**说明：** 这里的{application}代表了应用名称，当客户端向Config Server发起获取配置请求时，Config Server会根据客户端的spring.application.name信息来填充{application}占位符以定位配置资源的存储位置。基于一个环境一个仓库的使用原则，可以使用{profile}代替{application}。另外在{application}中使用特殊字符"（_）"可以支持多组织。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://localhost/{application}
```

**说明：** 这里的{application}的格式为“organization（_）application”。

**注意：** {label}很特别，如果Git分支和标签包含“/”，那么{label}在HTTP的URL中应用使用“（_）”替代，以避免改变URI的含义，指向其他URI资源。例如，如果标签是foo/bar，则替换“/”将导致出现类似于foo（_）bar的标签。如果使用像curl这样的命令行客户端（例如使用引号将其从Shell中转出来），请小心URL中的方括号。

**多模式匹配及多存储仓库**

我们也可以使用{application}/{profile}进行模式匹配，以便获取相应的配置文件。它的格式是用一组逗号分隔的{application}/{profile}，其中的参数可以使用通配符。目前，我们加入了一个应用客户端，它的配置如下：

```
spring:
    application:
      name: customize-config-client
    cloud:
      config:
        uri: http://locahost:6001  #假设配置中心地址为本地 6001 端口
        profile: dev
        label: master
```

Config Client按照上述配置请求服务器仓库地址，下面我们通过模式配置规则对Config Server服务端进行HTTP请求，地址为uri：http://locahost：6001/test-config-client/dev/master。

由URI可知，我们使用了/{application}/{profile}[/{label}的匹配模式。下面我们看一下加入了多模式匹配的配置中心（Config Server）的Config配置：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://localhost/spring-cloud-samples/config-repo
          repos:
            simple: https://localhost/simple/config-repo
            special:
              pattern: special-*/dev*
              uri: https://localhost/special/config-repo
            customize:
              pattern: customize-*
              uri: https://localhost/customize/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

上面的配置分别对应三组不同配置匹配规则，默认情况下，Git访问的是https://localhost/spring-cloud-samples/config-repo仓库地址。如果你的URI中的

{application}/{profile}没有匹配到Git默认的仓库地址，Config Server将根据模式匹配规则尝试能否匹配其他仓库地址：

- “simple”仓库匹配的是“simple/*”（在所有环境下它仅仅匹配一个仓库simple），目前从我们的客户端请求URI中发现，这项repo资源不匹配。
- “local”仓库将匹配所有名字以“local”开头的{application}，也是在所有的环境下。“/*”前缀会自动添加到所有没有设置{profile}的模式中。
- “customize”和“special”使用模式匹配规则，pattern为“customize-*”标识的应用，表示{application}是以customize-开头的应用repo资源，例如，customize-config-client的应用名称与customize-*的模式相互匹配，所以返回该资源下的repo资源。

**注意：**

- 如果模式中需要配置多个值，那么可以使用逗号分隔。
- 如果{application}/{profile}没有匹配到任何资源，则使用配置的默认URI：spring.cloud.config.server.git.uri。
- 当我们使用YAML类型的文件进行配置时，如果模式属性是一个YAML数组，也可以使用YAML数组格式来定义。这样可以设置成多个配置文件，如下代码所示：

```
        spring:
          cloud:
            config:
              server:
                git:
                  uri: https://localhost/spring-cloud-samples/config-repo
                  repos:
                    development:
                      pattern:
                        - */development
                        - */staging
                      uri: https://localhost/development/config-repo
                    staging:
                      pattern:
                        - */qa
                        - */production
                      uri: https://localhost/staging/config-repo
```

**路径搜索占位符等配置**

当我们把配置文件存放在Git仓库的子目录中时，可以通过设置searchPaths来指定该目录。同样，searchPaths也支持上面的占位符，示例如下：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://localhost/spring-cloud-samples/config-repo
          searchPaths: demo-config-repo，  springCloud-config*
            cloneOnStart: true
```

**说明：** 在上面的例子中，将在demo-config-repo和以springCloud-config开头的目录中搜索配置文件。cloneOnStart设置为True时，服务器在启动的时候克隆仓库，而如果没有该项配置，表示服务器可以在第一次请求配置文件时克隆远程仓库。

## 5.2.4 Config Server定制化开发

Config Server配置中心默认提供Git的方式及Git文件管理GUI作为配置中心的前端可视化管理工具，但是默认的Git文件存储方式存在配置文件的配置项格式校验、不同项目的版本回滚及配置实时更新等问题，所以我们可以通过对Config Server进行二次开发实现定制化的功能。

**Config Server源码解析**

● Config Server自动化配置

查看@EnableConfigServer注解，通常都是引入某种Configuration类来达到自动化装配某些Bean的目的，如下代码所示：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(ConfigServerConfiguration.class)
public @interface EnableConfigServer {
}
```

● ConfigServerConfiguration实现了一个标志类

```
@Configuration
public class ConfigServerConfiguration {
   class Marker {}
   @Bean
   public Marker enableConfigServerMarker() {
      return new Marker();
   }
}
```

ConfigServerConfiguration类里面并没有实现太多Bean的装配，这里利用一种折中方式引入需要的自动配置。请看下面的类，Marker唯一被引用的地方在类中。

```
@Configuration
@ConditionalOnBean(ConfigServerConfiguration.Marker.class)
@EnableConfigurationProperties(ConfigServerProperties.class)
@Import({ EnvironmentRepositoryConfiguration.class,
    CompositeConfiguration.class, ResourceRepositoryConfiguration.class,
    ConfigServerEncryptionConfiguration.class, ConfigServerMvcConfiguration.class })
public class ConfigServerAutoConfiguration {
}
```

**说明：** @ConditionalOnBean（ConfigServerConfiguration.Marker.class）表示当装配了ConfigServerConfiguration.Marker的实例时才会执行ConfigServerAutoConfiguration。

从源码我们可以发现，Config配置类中另外引入了5个辅助配置类，篇幅所限这里不对每个配置类进行详细说明，仅对对Config Server最为关键的EnvironmentRepositoryConfiguration加以说明。

● 查看EnvironmentRepositoryConfiguration类

```
@Configuration
@EnableConfigurationProperties({
    SvnKitEnvironmentProperties.class, JdbcEnvironmentProperties.class,
       NativeEnvironmentProperties.class,  VaultEnvironmentProperties.class })
@Import({ CompositeRepositoryConfiguration.class,
    JdbcRepositoryConfiguration.class,  VaultRepositoryConfiguration.class,
      SvnRepositoryConfiguration.class,  NativeRepositoryConfiguration.class,
      GitRepositoryConfiguration.class,
      DefaultRepositoryConfiguration.class })
public class EnvironmentRepositoryConfiguration {
}
```

这里的@Import又引入了7种配置类，查看文档会发现其实刚好对应Config Server的几种实现方式，其中Git使用的配置类就是GitRepositoryConfiguration。

以GitRepositoryConfiguration为例，GitRepositoryConfiguration其实是默认的实现方式，查看DefaultRepositoryConfiguration的代码：

```
@Configuration
@ConditionalOnMissingBean(value = EnvironmentRepository.class,  search =
SearchStrategy.CURRENT)
class DefaultRepositoryConfiguration {
   @Autowired
   private ConfigurableEnvironment environment;
   @Autowired
   private ConfigServerProperties server;
   @Autowired(required = false)
   private TransportConfigCallback customTransportConfigCallback;
   @Bean
   public MultipleJGitEnvironmentRepository defaultEnvironmentRepository(
         MultipleJGitEnvironmentRepositoryFactory gitEnvironmentRepositoryFactory,
         MultipleJGitEnvironmentProperties environmentProperties) throws
           Exception {
         return gitEnvironmentRepositoryFactory.build(environmentProperties);
   }
}
```

它首先装配一个MultipleJGitEnvironmentRepository的Bean对象，然而实际每种配置类的最终实现都会装配EnvironmentRepository配置类。最终通过调用EnvironmentRepository的findOne方法来查询配置。而接口EnvironmentRepository（Config Server实现定制化加载配置的关键接口）只提供了一个方法findOne，通过传入application、profile和label来获得配置项：

```
public interface EnvironmentRepository{
     Environment findOne(String application, String profile, String label);
}
```

**配置中心定制化实现**

在默认情况下，Config Server可以通过在配置文件中进行简单配置实现数据库访问和存储。但是，Config Server自带的SQL实现方式存在一定的缺陷，因为它的表结构的功能定义相对简

单，没有版本管理和权限管理功能。下面我们基于Config Server开源实现，根据“读写分离”的原则实现自定义的配置中心定制化改造。

- 配置文件写入部分

  在开始实现自定义的EnvironmentRepository之前，我们需要进行数据库表设计，因为篇幅所限，我们仅列出几个核心表的设计。在设计时主要使用表config_namespace存储配置文件主表，包含所有应用及对应配置文件信息，`application_name`、`label`和`profile`三个字段是联合主键，对应一个应用配置文件。表结构如下：

```
DROP TABLE IF EXISTS `config_namespace`;
CREATE TABLE `config_namespace`
(
   `id`                BIGINT(8) UNSIGNED    NOT NULL AUTO_INCREMENT,
   `application_name`  VARCHAR(100)          NOT NULL COMMENT '应用名',
   `label`             VARCHAR(100)          NOT NULL COMMENT 'label',
   `profile`           VARCHAR(100)          NOT NULL COMMENT 'profile',
   `format`            VARCHAR(50)           NOT NULL COMMENT '配置文件格式',
   `creator`           VARCHAR(100)          NOT NULL DEFAULT '' COMMENT '创建者',
   `modifier`          VARCHAR(100)          NOT NULL DEFAULT '' COMMENT '修改者',
   `create_time`       DATETIME              NOT NULL COMMENT '创建时间',
   `modified_time`     DATETIME              NOT NULL COMMENT '修改时间',
   PRIMARY KEY (`id`),      UNIQUE KEY (`application_name`, `label`, `profile`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4 COMMENT ='配置文件 namespace 表';
```

对于配置文件的更改历史和版本相关信息，我们使用表config_release表示，其中namespace_id对应config_namespace中`application_name`、`label`和`profile`联合主键的信息。下面是表的结构：

```
DROP TABLE IF EXISTS `config_release`;
CREATE TABLE `config_release`
(
  `id`           BIGINT(8) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '自增 id',
  `namespace_id` BIGINT(8) UNSIGNED NOT NULL COMMENT 'namespace id',
  `version`      VARCHAR(100)    NOT NULL COMMENT '版本号',
  `content`      mediumtext      NOT NULL COMMENT '配置文件内容',
  `format`       VARCHAR(50)     NOT NULL COMMENT '配置文件格式',
  `remark`       VARCHAR(200)    NOT NULL DEFAULT '' COMMENT '备注',
  `state`        TINYINT         NOT NULL DEFAULT 0 COMMENT '标示发布状态',
  `creator`      VARCHAR(100)    NOT NULL DEFAULT '' COMMENT '发布者',
  `create_time`  DATETIME  NOT   NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时',
  PRIMARY KEY (`id`),
  UNIQUE KEY (`version`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4 COMMENT ='配置发布表';
```

在自定义的Config Server中，我们需要使用注解@EnableConfigServer初始化并加载配置中心的环境变量以启动配置中心，代码如下：

```
@SpringBootApplication
@EnableConfigServer
public class SpringCloudDefineConfigServer {
    public static void main(String[] args) {
        SpringApplication.run(SpringCloudDefineConfigServer.class, args);
    }
}
```

根据上述数据库表结构设计，配置文件写入部分我们主要提供一个Admin管理端，允许前端通过GUI界面实现对配置文件在数据库中的插入、修改、删除，这里我们仅介绍核心表的添加配置功能，主要通过Controller类、Service类与DAO类实现数据库插入配置项，配置服务中心的Controller类负责配置文件的添加，实现代码实例如下：

```
@RestController
@RequestMapping(value = "/item")
@Api(description = "item 管理")
public class ConfigItemController {
  @PostMapping(value = "/insert")
  @ApiOperation(value = "插入 item")
  public ResultBody<ConfigItem> insert(@Valid @RequestBody ConfigItem item) {
        if (!PropertiesUtil.formatCheck(item.getContent(), item.getFormat())) {
            return ResultBody.buildInValidParamResult("content check failed!");
        }
        int sqlResult = service.insert(item);
        if (sqlResult == 0) {
            return ResultBody.buildInValidParamResult("insert failed.");
        }
        return ResultBody.buildSuccessResult(item);
    }
}
```

配置服务中心的Service类的配置文件插入实现代码如下：

```
@Service
@Transactional(rollbackFor = Exception.class)
@Slf4j
public class ConfigItemServiceImpl {
    @Override
    public int insert(ConfigItem item) {
        //权限管理部分代码省略，这里可以根据操作用户判断是否对配置文件有操作权限
        item.setCreator(username);
        item.setModifier(username);
        item.setIsReleased(false);
        if (item.getRemark() == null) {
            item.setRemark("");
        }
        ConfigItemModifiedRecord record = new ConfigItemModifiedRecord();
        record.setNamespaceId(item.getNamespaceId());
        record.setContent(JsonHelper.toJSONString(PropertiesUtil.compareRTL("",
            item.getContent(), item.getFormat())));
        record.setVersion("");
        record.setCreator(username);
        modifiedRecordMapper.insert(record);
        return itemMapper.insert(item);
    }
}
```

配置服务中心DAO类的配置文件写入使用Spring Boot的@Repository注解及MyBatis实现，代码如下：

```
@Repository
public interface ConfigItemMapper {
  int insert(ConfigItem item);
}
```

对应的MyBatis配置如下：

```
<?XML version="1.0" encoding="UTF-8"?>
<mapper namespace="com.sia.config.dao.mapper.ConfigItemMapper">
<insert id="insert"
    parameterType="com.spring.config.common.model.ConfigItem" useGeneratedKeys="true"
    KeyProperty="id"
    KeyColumn="id">
      insert into config_item (namespace_id, format, remark, is_released, creator, modifier,
        content)
   values (#{namespaceId, jdbcType=BIGINT}, #{format, jdbcType=VARCHAR},
   #{remark,jdbcType=VARCHAR}, #{isReleased,jdbcType=BIT}, #{creator,jdbcType=VARCHAR},
         #{modifier, jdbcType=VARCHAR}, #{content, jdbcType=LONGVARCHAR})
    </insert>
</mapper>
```

- 配置文件读取部分

  如上述分析，可以使用EnvironmentRepository作为我们实现自定义的配置文件存储查询的接口类，只需要覆盖实现自定义的EnvironmentRepository的Bean，就可以替换读取逻辑，从默认的基于Git的存储读取方式转换为我们想要的基于数据库的配置读取方式。定制化EnvironmentRepository的主要代码如下：

```
@Component
```

```
public class CustomizeJdbcEnvironmentRepository implements EnvironmentRepository, Ordered
{
    private int order;
    private final JdbcTemplate jdbc;
    @Autowired
    private ConfigClientRecordMapper clientRecordMapper;
    public CustomizeJdbcEnvironmentRepository (JdbcTemplate jdbc,
        JdbcEnvironmentProperties properties) {
        this.jdbc = jdbc;
        this.order = properties.getOrder();
    }
    @Override
    public Environment findOne(String application, String profile, String label) {
        String config = application;
        if (StringUtils.isEmpty(label)) {
            label = ConfigConstant.MASTER;
        }
        if (StringUtils.isEmpty(profile)) {
            profile = ConfigConstant.DEFAULT;
        } else if (!profile.startsWith(ConfigConstant.DEFAULT)) {
            profile = ConfigConstant.DEFAULT + ", " + profile;
        }
        String[] profiles =
           StringUtils.commaDelimitedListToStringArray(profile);
           Environment environment = new Environment(application, profiles, label, null,
                        null);
        if (!config.startsWith(ConfigConstant.APPLICATION)) {
            config = ConfigConstant.APPLICATION + ", " + config;
        }
        List<String> applications = new ArrayList<>(new LinkedHashSet<>(
            Arrays.asList(StringUtils.commaDelimitedListToStringArray(config))));
        List<String> envs = new ArrayList<>(new LinkedHashSet<>(Arrays.asList(profiles)));
        Collections.reverse(applications);
        Collections.reverse(envs);
        for (String app : applications) {
            //统一转为小写，查询不区分大小写
            app = app.toLowerCase();
            for (String env : envs) {
                List<Long> namespaceIdList = jdbc.queryForList(
                    "select id from config_namespace where application_name=? and
                    profile=? and label=?",
                    new Object[]{app, env, label}, Long.class);
                if (namespaceIdList == null || namespaceIdList.isEmpty()) {
                    continue;
                }
                List<ConfigRelease> releaseList = jdbc.query(
                    "select * from config_release where namespace_id=? and state=0",
```

说明 1#

说明 2#

```
                    new Object[]{namespaceIdList.get(0)}, new
                BeanPropertyRowMapper<>(ConfigRelease.class));
            if (releaseList == null || releaseList.isEmpty()) {
                continue;
            }
            ConfigRelease release = releaseList.get(0);
            Properties properties;
            if (ConfigConstant.YML.equals(release.getFormat())) {
                properties = PropertiesUtil.ymlStringConvertToProperties
                (release.getContent());
            } else {
                properties = PropertiesUtil.propertiesStringConvertToProperties
                (release.getContent());
            }
            if (properties.isEmpty()) {
                continue;
            }
            log.info("get config success, app:{}, profile:{}, label:{}", app, env,
                    label);
            recordClientVersion(release);
            environment.add(new PropertySource(app + ConfigConstant.HYPHEN +
                env, properties));
        }
    }
    return environment;
}
public void recordClientVersion(ConfigRelease release) {
    try {
        ConfigClientRecord record = new ConfigClientRecord();
        record.setNamespaceId(release.getNamespaceId());
        record.setVersion(release.getVersion());
        record.setIp(IpUtil.getIpAddr());
        clientRecordMapper.insert(record);
    } catch (Exception e) {
        log.error("record client version error!", e);
    }
}
@Override
public int getOrder() {
    return order;
}
public void setOrder(int order) {
    this.order = order;
}
}
```

说明 3#

○ 说明1＃：使用JDBC的方式将初始化定制化。
○ 说明2＃：findOne是EnvironmentRepository的接口，源码如下：

```
public interface EnvironmentRepository {
    Environment findOne(String var1, String var2, String var3);
}
```

该方法中最核心的机制就是通过application、profile、label三个参数找到对应的配置文件信息（基本单元）。在默认的Config Server实现方式中，它使用Git Client完成Git文件读取的方式加载对应Git Server中的配置文件，而在我们的定制化实现方式中，使用JDBC的方式获取配置文件信息。说明2＃根据上述三个参数查找出三个联合主键对应的唯一配置文件ID。

○ 说明3＃：这部分代码通过上一步取得的唯一ID信息从配置表（config_release）中取得配置文件信息，该信息对应ConfigRelease类模型信息。下面是ConfigRelease的代码实现：

```
@AllArgsConstructor
@NoArgsConstructor
@ApiModel(value = "已发布配置 bean")
@Data
public class ConfigRelease implements Serializable {
    @ApiModelProperty(value = "配置发布 ID")
    private Long id;
    @ApiModelProperty(value = "配置文件 namespace ID")
    private Long namespaceId;
    @ApiModelProperty(value = "版本号")
    private String version;
    @ApiModelProperty(value = "内容")
    private String content;
    @ApiModelProperty(value = "配置格式")
    private String format;
    @ApiModelProperty(value = "备注")
    private String remark;
    @ApiModelProperty(value = "标示当前状态是历史、当前或回滚")
    private Byte state;
    @ApiModelProperty(value = "发布人")
    private String creator;
    @ApiModelProperty(value = "发布时间")
    private Date createTime;
    private static final long serialVersionUID = 1L;
}
```

EnvironmentController实现配置中心Admin前端管理及权限验证入口，Config Server使用HTTP的方式作为Config Client配置文件的接口调用方式，所以我们通过EnvironmentController来暴露给客户端的配置入口，代码如下：

```
@RequestMapping("/{name}/{profiles}/{label:.*}")
public Environment labelled(@PathVariable String name， @PathVariable String profiles，
  @PathVariable String label) {
  if (name != null && name.contains("(_)")) {
  // "(_)" is uncommon in a git repo name， but "/" cannot be matched
  // by Spring MVC
  name = name.replace("(_)"， "/");
  }
 if (label != null && label.contains("(_)")) {
  // "(_)" is uncommon in a git branch name， but "/" cannot be matched
  // by Spring MVC
  label = label.replace("(_)"， "/");
 }
 Environment environment = this.repository.findOne(name， profiles， label);
 if(!acceptEmpty && (environment == null ||
       environment.getPropertySources().isEmpty())){
   throw new EnvironmentNotFoundException("Profile Not found");
   }
   return environment;
}
```

可以看出，/{name}/{profiles}/{label：.*}路径参数正好与我们的请求方式相对应，因此Config Server是通过建立一个RestController来接收读取配置请求的，然后使用EnvironmentRepository来进行配置查询，返回Environment对象的JSON结果值，客户端接收时也应该将其反序列化为Environment的一个实例。

## 5.3 微服务网关

微服务网关在微服务架构中作为HTTP请求的统一调用入口，用来屏蔽和隔离内部服务实现细节，保护、增强和控制对微服务的访问，实现了服务之间调用关系的松散耦合，增强了服务的可重用性。此外还可以将安全、灰度、熔断等公共组件和切面功能放置到网关节点，实现系统的关注点分离。

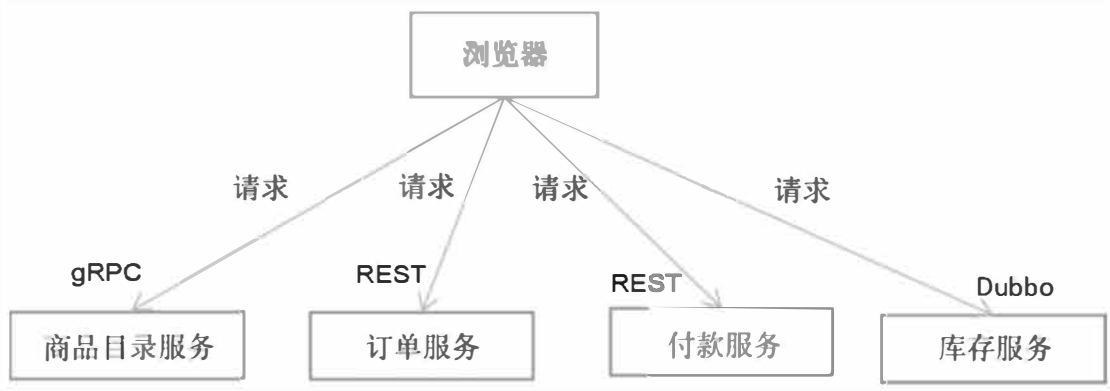
### 5.3.1 微服务网关模式

在微服务架构中，一个大应用被拆分为多个小的服务系统，这些小的系统根据领域划分独立完成某项功能，也就是说这些小系统可以拥有自己的数据库、框架甚至语言等。这些小系统通常通过REST API风格的接口供浏览器、H5、Android、iOS及第三方应用程序调用。

例如，我们把商品应用拆分为四个独立的子服务：商品目录服务、订单服务、付款服务、库存服务。商品目录服务使用gRPC方式对外暴露接口，订单和付款服务采用REST方式，而库存服务使用Dubbo方式，客户进入商品页面后需要同时使用多个微服务查询当前订单的详情。

**客户端直连服务端模式**

下图是客户端直连服务端模式。如果客户端想查看商品目录、订单、付款、库存这些服务，需要分别向这些服务的URL地址发送请求，然后客户端需要根据各服务端返回的响应完成聚合。



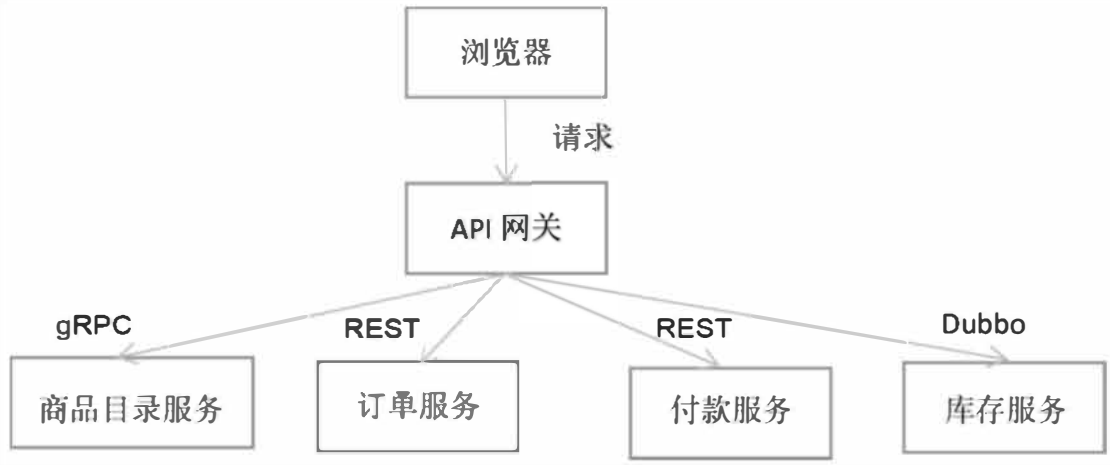

这种模式的缺点是明显的：

- 效率低下。客户端如果需要一个一键下单的功能，可能会涉及与多个微服务的接口交互，需要先查看商品库存、然后完成订单、最后付款。当客户端API接口粒度与后端API接口粒度不匹配时，浏览器需要来回多次访问请求，才能完成操作。而浏览器与后端的微服务是通过WAN公网通信的，这样带来的一个问题就是带宽的浪费。
- 协议适配问题。在这个实例中我们发现，商品目录服务采用了gRPC通信协议，订单、付款服务采用了REST方式的HTTP。库存服务可能是一个遗留系统，采用了Dubbo框架，对外暴露的是Dubbo的RPC接口。在这种情况下，客户端如果想实现对不同接口的访问和对接，需要分别适配不同的技术栈，这本身也给前端工作带来了复杂性和非Web协议的不友好性。应用对外暴露的接口最好使用对Web友好、对防火墙友好的HTTP或者SOAP。应用内部之间的调用可以采用RPC这样的远程方法。
- 耦合性强。这种模式的另一个重大缺陷就是客户端与服务端存在接口层面的依赖，服务端的技术栈和接口变化都会对调用方产生极大的影响。随着时间的推移，这种依赖将束缚服务的调用方与提供方的迭代和演进，这给微服务架构下的服务划分及重构都带来了难度。
- 可替代性下降。例如，当前端调用方需要从浏览器迁移到移动端时，在浏览器上使用的对接技术（JavaScript对接gRPC、JavaScript对接Dubbo）将无法迁移到移动端。

正因为客户端直连服务端模式存在如此多的问题，所以在微服务架构中我们很少直接通过客户端与服务端通信。我们知道，在软件开发领域有一句名言：任何软件工程遇到的复杂问题都可以通过增加一个中间层来解决。下面我们看看微服务网关是如何解决这个问题的。

**API网关模式**

API网关模式通过在客户端和服务端之间增加一个中间层，使得客户端可以在一次请求中向多个服务获取数据，请求数的减少也会直接改善用户体验。如下图所示是API网关的使用图示。




首先，在使用API网关模式后，API网关对外屏蔽了服务的内部细节，通过一个粗粒度的URL可以实现一键下单服务，如/product/order：xxx。客户端对于网关的请求没有来回多轮请求，你只需要对网关下发一个请求，网关对外屏蔽调用多个内部微服务请求的操作细节。当完成所有服务请求后，网关将各个响应进行聚合再返回给客户端，减少了外网请求和响应的交互次数，提高了交互

的效率。由于能够对返回数据进行灵活处理，API网关减少了请求往返次数，从而简化了客户端的调用，也提高了访问服务的性能。

其次，假设每个系统使用的协议不同，那么系统之间的调用或者数据传输就存在协议转换，API网关可以通过内置的协议转换引擎实现多种协议的适配和转换，将对外的请求统一在HTTP-REST的协议规范下。常用的处理机制通过泛化调用的方式实现协议之间的转化。实际上就是将不同的协议转换成通用协议，然后将通用协议转化成本地系统能够识别的协议。这一转化工作通常由API网关完成。

第三，API网关实现客户端和服务端调用关系和部署环境的解耦，它向客户端隐藏了应用划分的微服务的部署版本。尽管微服务架构支持客户端直接与微服务交互，但当需要交互的微服务数量较多时，解耦就成为单体迈向微服务的必要工作。对于演进式架构，一个服务可能同时存在多个版本，对于A/B测试的场景，通过在URL中增加版本号（例如/path/version1/xxxx），或者在HTTP-Head中增加版本参数的方式，网关可以根据负载策略进行请求流量调节。客户端可以灵活地选择在不同场景使用不同版本的后端服务。

在微服务架构的网络拓扑中，微服务网关是一个处于应用程序和服务（提供REST API接口服务）之间的中间件系统，它可以用来完成管理授权、访问控制和流量限制等。REST API接口服务对调用者透明，隐藏在API网关后面的业务系统可以专注于创建和构建业务逻辑，服务调用者和服务提供者通过网关实现了解耦。

当然，API网关作为一个高可用组件，也增加了系统的复杂性和瓶颈点。我们很容易将微服务架构的API网关与SOA分布式架构的ESB系统联系起来。ESB作为服务总线也可以实现协议转换、解耦服务提供方和服务调用方，ESB还有重量级的服务编排等和业务逻辑关联比较强的特性，所以我们有必要说明一下API网关和ESB的区别。

**API网关和ESB的比较**

ESB是在应用服务化的早期、伴随着EAI（Enterprise Application Integration，企业应用集成）和SOA的理念而产生的。它产生之初，是一个解决服务集成问题的服务中间件。同时，ESB所处理的服务是企业级的服务，服务粒度比较粗，它践行的是“共享”的架构模式。它的主要功能包括服务的调解和路由、消息增强、消息转换和协议转换，这是它必须具备的也是具有强大优势的地方，消息队列的处理、大数据的传输更是它的强项。总之，ESB是一种集中式的服务总线方式，可以以最小的代价把竖井式架构的应用改造成面向服务的架构。但由于通过ESB暴露出来的服务以紧耦合的方式固化在竖井应用中，其服务的柔性比较差，服务更改的代价比较大，导致ESB暴露的服务可能无法快速适应需求的变化。

API网关是伴随着微服务的广泛使用而产生和发展的，API网关是为了协调单体应用拆分后的众多微服务而产生的。

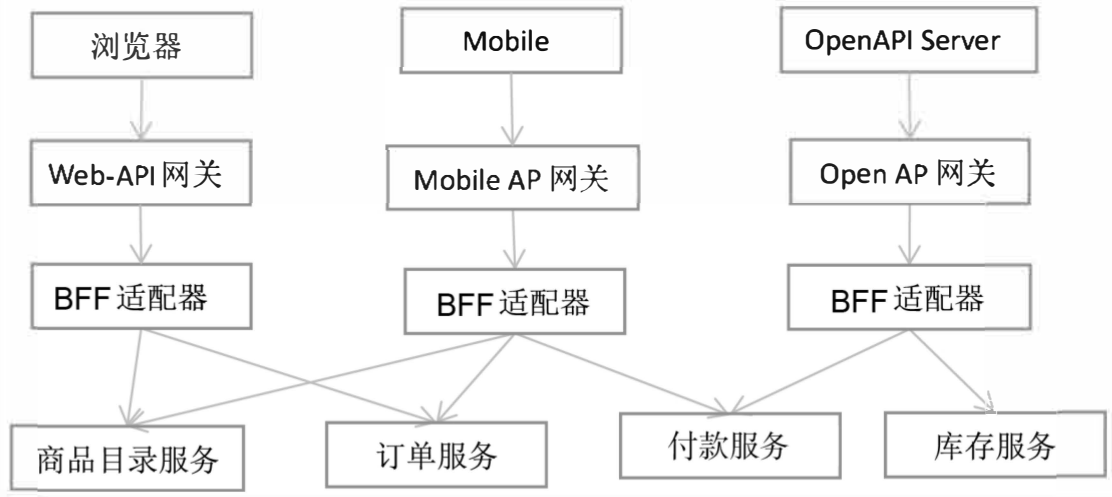
API网关主要完成微服务集成、服务路由、灰度发布、流量控制等非业务属性公共功能，对于业务属性比较强的服务调配编排、消息增强转换、业务规则管理等功能就不是微服务网关要考虑的事情，而是由微服务自行处理。而微服务应用下各个微服务之间的异步数据传输依旧需要单独的消息处理软件用发布订阅机制来进行处理，比如使用Kafka、RabbitMQ等消息中间件。

微服务的最初目的不是功能的重用，而是适合小团队自主开发，达到敏捷开发、敏捷发布的目的，以缩短软件上市周期，所以它强调独立。而API网关作为单一入口，通过请求适配整合后台微服务体系，面向各种客户端提供统一服务请求。

API网关的一个问题就是中心化的架构问题，我们需要考虑不同的前端、不同的业务团队，可能对网关存在隔离性的需求，在发布、部署、可用性方面，如何设计一个去中心化的网关系统也是实施微服务架构需要考虑的。

**BFF网关模式**

BFF全称是Backends For Frontends（服务于前端的后端）。BFF是指在设计API时为不同的设备提供不同的API接口，虽然它们可能实现相同的功能，但因为不同设备的特殊性，需要区别处理。如下图所示是BFF网关模式的示例。




客户端都不直接访问服务端的公共接口，而是调用BFF层提供的接口，BFF层再调用基层的公共服务。不同的客户端拥有不同的BFF层，它们为客户端提供定制化的API接口。我们可以认为BFF是一种适配服务，将后端的微服务进行适配（主要包括聚合裁剪和格式适配等），向无线设备暴露友好和统一的API，方便无线设备访问后端服务。

采用BFF网关模式能够满足因不同客户端特殊的交互引起的对新接口的要求，所以一开始就会针对相应的设备设计好对应的接口。从职责分配上来说，BFF网关模式每一层的功能更加清晰，网关（一般由独立框架团队负责运维）专注跨横切面（Cross-Cutting Concerns）的功能，使用相同的技术栈和公共类库；而BFF层主要是适配层，BFF层的开发人员可以更加专注于业务逻辑。

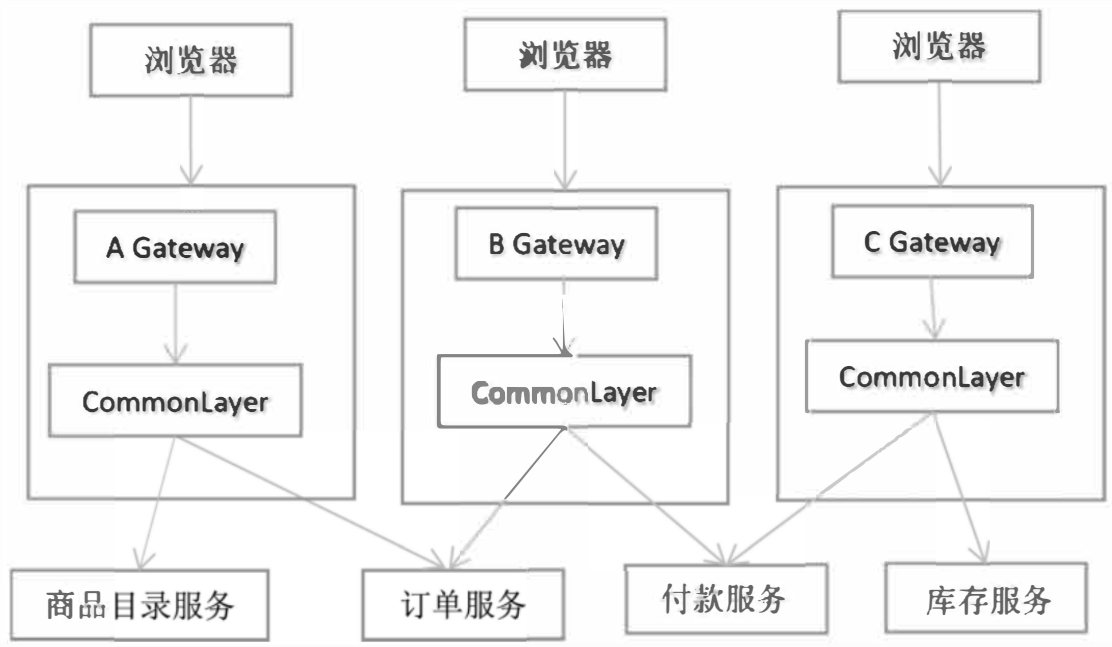
通过客户端→网关层→BFF层→微服务层的划分，整个微服务架构层次清晰、职责分明，它是一种灵活的能够支持业务不断创新的演进式架构。

**微网关模式**

虽然BFF网关模式在某种程度上实现了不同组件功能之间的解耦，但是它主要是前端类别的去中心化和适配。BFF网关模式适用于对不同客户端（前端技术类别）类型的解耦。然而，对于不同业务类型或者不同团队网关服务提供者来说，BFF网关模式还是存在相互影响、相互耦合的情况。

微网关（MicroGateway）模式可以理解为去中心化的网关模式，微服务架构的一个重要特性就是去中心化。我们可以从业务的维度进一步划分网关系统，在软件设计层面增加一个“网关组”的抽象概念，一个网关组对应一个独立的业务领域。网关组的概念也契合了微服务架构中的一些理念：业务系统依赖微服务网关提供清晰明确的服务边界，业务系统通过微服务网关对外暴露业务的标准服务接口。

在部署方式上，充分利用并结合了容器自动化的技术，在解决最后一公里的问题上，将网关以云端容器资源的方式交付给不同的业务方，通过共享网关SDK部署包的方式将网关的服务下沉到容器中实现和执行，从而在时间和空间上做到了弹性和灵活交付。使用网关的具有不同权限的用户可以同时维护各自所属网关组下的网关节点。下图展示的是去中心化的微网关模式。




目前Service Mesh使用的也是去中心化的网关模式，将数据平面和控制平面解耦，是区别于应用层和通信层的一种云原生上下文。边车模式（SideCar Pattern）本质上也是一种去中心化的微服务的管理方式。在本书的进阶篇中，我们会更详尽地介绍微网关、边车模式、Service Mesh的演进过程和微服务的发展趋势。

## 5.3.2 网关的主要功能

微服务网关作为微服务后端服务的统一入口（Entry Point），它可以统筹管理后端服务，主要分为数据平面（Data Plane）和控制平面（Control Plane）。

- 数据平面的主要功能是接入用户的HTTP请求和微服务被拆分后的聚合。使用微服务网关统一对外暴露后端服务的API和契约，路由和过滤功能正是网关的核心能力模块。另外，微服务网关可以实现拦截机制和专注跨横切面（Cross-Cutting Concerns）的功能，包括协议转换、安全认证、熔断限流、灰度发布、日志管理、流量监控等。
- 控制平面的主要功能是对后端服务做统一的管控和配置管理。例如，可以控制网关的弹性伸缩；可以统一下发配置；可以对网关服务添加标签；可以在微服务网关上通过配置Swagger功能统一将后端服务的API契约暴露给使用方，完成文档服务，提高工作效率和降低沟通成本。

**路由功能**

路由是微服务网关的核心能力。通过路由功能微服务网关可以将请求转发到目标微服务。在微服务架构中，网关可以结合注册中心的动态服务发现，实现对后端服务的发现，调用方只需要知道网关对外暴露的服务API就可以透明地访问后端微服务。

**负载均衡**

API网关结合负载均衡技术，利用Eureka或者Consul等服务发现工具，通过轮询、指定权重、IP地址哈希等机制实现下游服务的负载均衡。

**协议转换**

API网关的一大作用在于构建异构系统，API网关作为单一入口，通过协议转换整合后台基于REST、AMQP、Dubbo等不同风格和实现技术的微服务，面向Web Mobile、开放平台等特定客户端提供统一服务。

**安全认证**

一般而言，无论对内网还是外网的接口都需要做用户身份认证，而用户认证在一些规模较大的系统中都会采用统一的单点登录（Single Sign On）系统，如果每个微服务都要对接单点登录系统，那么显然比较浪费资源且开发效率低。API网关是统一管理安全性的绝佳场所，可以将认证的部分抽取到网关层，微服务系统无须关注认证的逻辑，只关注自身业务即可。常见的安全性技术（如密钥交换、客户端认证与报文加解密等）都可以在API网关中实现。

**黑白名单**

微服务网关可以使用系统黑名单，过滤HTTP请求特征，拦截异常客户端的请求，例如DDoS攻击等侵蚀带宽或资源迫使服务中断等行为，可以在网关层面进行拦截过滤。比较常见的拦截策略是

根据IP地址增加黑名单。在存在鉴权管理的路由服务中可以通过设置白名单跳过鉴权管理而直接访问后端服务资源。

**灰度发布**

微服务网关可以根据HTTP请求中的特殊标记和后端服务列表元数据标识进行流量控制，实现在用户无感知的情况下完成灰度发布。

**流量染色**

和灰度发布的原理相似，网关可以根据HTTP请求的Host、Head、Agent等标识对请求进行染色，有了网关的流量染色功能，我们可以对服务后续的调用链路进行跟踪，对服务延迟及服务运行状况进行进一步的链路分析。

**限流熔断**

在某些场景下需要控制客户端的访问次数和访问频率，一些高并发系统有时还会有限流的需求。在网关上可以配置一个阈值，当请求数超过阈值时就直接返回错误而不继续访问后台服务。当出现流量洪峰或者后端服务出现延迟或故障时，网关能够主动进行熔断，保护后端服务，并保持前端用户体验良好。

**服务管控**

网关可以统计后端服务的请求次数，并且可以实时地更新当前的流量健康状态，可以对URL粒度的服务进行延迟统计，也可以使用Hystrix Dashboard查看后端服务的流量状态及是否有熔断发生。

**文档中心**

网关结合Swagger，可以将后端的微服务暴露给网关，网关作为统一的入口给接口的使用方提供查看后端服务的API规范，不需要知道每一个后端微服务的Swagger地址，这样网关起到了对后端API聚合的效果。

**日志审计**

微服务网关可以作为统一的日志记录和收集器，对服务URL粒度的日志请求信息和响应信息进行拦截。

## 5.3.3 网关的技术选型

- Nginx+Lua

  Nginx是一个高性能的HTTP和反向代理服务器。Nginx一方面可以做反向代理，另外一方面可以做静态资源服务器，接口使用Lua动态语言可以完成灵活的定制功能。

- Kong

  Kong是一款API管理软件，它本身是基于Nginx+Lua的，但比Nginx提供了更简单的配置方式，数据采用Apache Cassandra或PostgreSQL存储。Kong的一个非常诱人的地方就是提供了大量的插件来扩展应用，通过不同的插件可以为服务提供各种增强功能。

- Spring Cloud Zuul

  Zuul是Netflix公司开源的一个API网关组件。目前，结合Spring Cloud提供的服务治理体系，可以完成请求转发、路由规则配置、负载均衡，以及集成Hystrix实现熔断功能。

- Spring Cloud Gateway

  Spring Cloud Gateway是基于Spring 5.0和Spring Boot 2.0构建的，提供路由等功能。

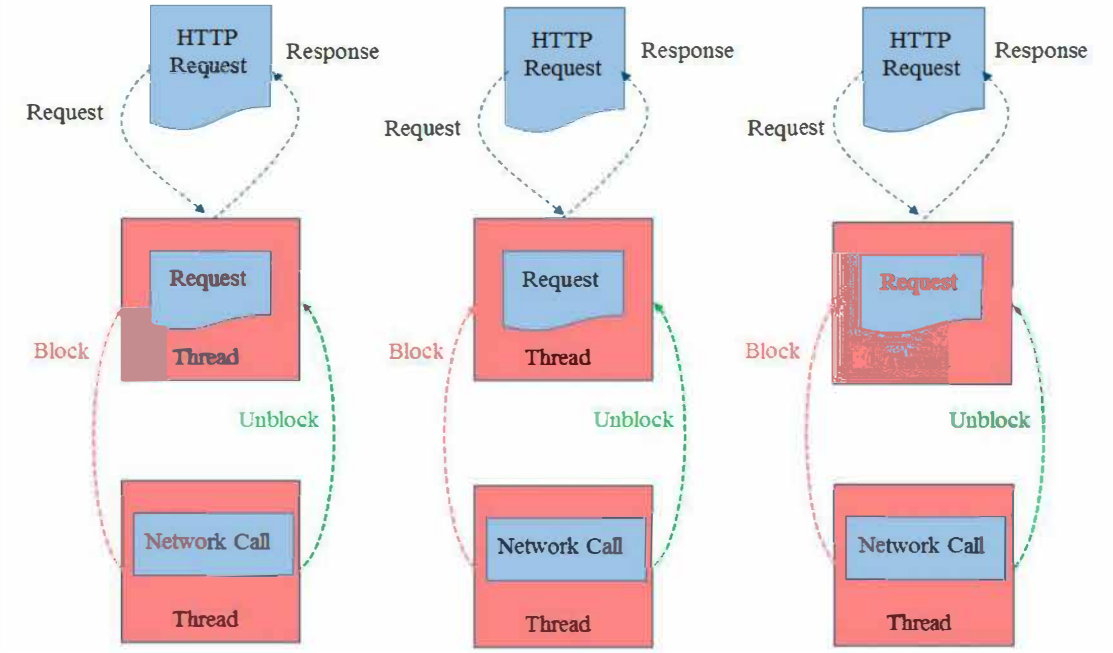
### 5.3.4 Spring Cloud Zuul网关

Spring Cloud Zuul是Spring Cloud在Netflix开源的Zuul网关的基础上，经过整合与增强实现的生产级别的微服务网关系统。Netflix架构总监Adrian Cockcroft曾表示：在Netflix的开源项目中，Zuul网关是一个容易被忽略但是最强大的基础服务之一。

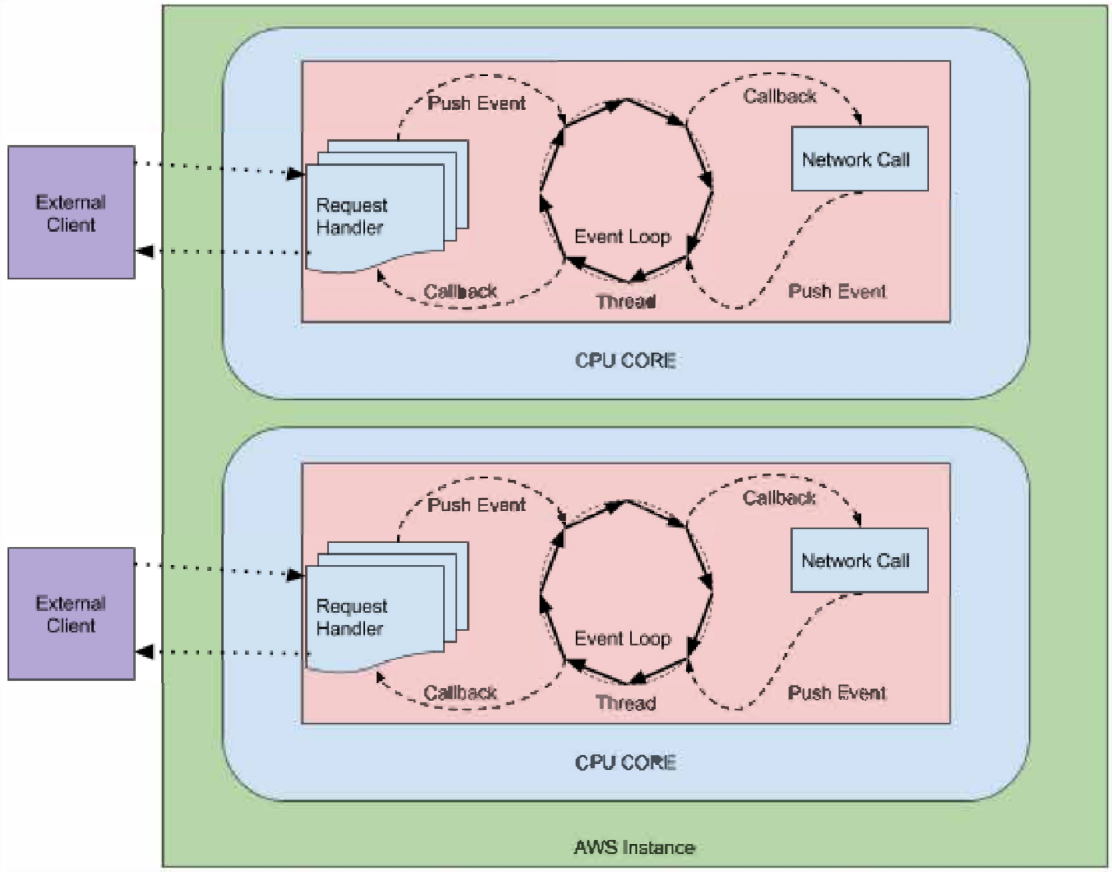
Zuul网关主要用于支持智能路由、安全认证、区域和内容感知，将多个底层服务聚合成统一对外暴露的API。Zuul网关的一大亮点是动态可编程，配置可以秒级生效。Netflix对Zuul的官方介绍：Zuul是从设备和网站到后端应用程序所有请求的门面（Facade），它为内部服务提供可配置的对外URL到服务的映射，及基于JVM的后端路由器。它具备以下功能。

- 动态路由：Zuul可以无缝对接Eureka服务发现中心，通过配置路由匹配规则，将符合匹配条件的请求转发到对应的后端服务上。
- 负载分配：为每一种负载类型分配对应的容量，并弃用超出限定值的请求。默认情况下，Zuul的负载均衡使用的是Ribbon的ZoneAwareLoadBalancer。该类的算法就是对服务发现中可用实例的轮询，成功追踪到响应的可用区域。
- 压力测试：逐渐增加指向集群的负载流量，从而计算性能水平。
- 静态响应处理：在边缘位置直接建立部分响应，从而避免其流入内部集群。
- 多区域弹性：跨越AWS（亚马逊云服务）区域进行请求路由，旨在实现负载均衡使用多样化并保证边缘位置与使用者尽可能接近。
- 验证与安全保障：识别面向各类资源的验证要求并拒绝那些与要求不符的请求。
- 审查与监控：在边缘位置追踪数据及统计结果，从而为我们提供准确的生产状态。

Zuul1的功能相对比较简单，它本质上是基于Spring MVC框架开发的一个Web Servlet应用。Zuul1的核心模块是一系列Filter过滤器，使用阻塞式的I/O，通过线程池技术实现请求的并发处理。每个请求都对应独立的线程，处理后端的业务逻辑。如下图所示是Zuul1的主要编程模型。




Zuul1的线程模型决定了Zuul1在高并发场景下存在性能瓶颈，所以Netflix为了解决网关的I/O瓶颈，开发了基于NIO（非阻塞I/O）模式的Zuul2。下图是Zuul2基于Reactor模式的架构实现。




Zuul2的设计相对复杂，采用了Netty框架实现异步非阻塞的编程模型，Zuul2虽然在性能上比Zuul1有明显的优势，然而Zuul2的问题是在编程模型和代码调试、排查问题上复杂性比较高。另外Zuul2的线程异步特性也给以本地线程方式进行请求跟踪和监控工作带来了麻烦。

在生态上，Spring Cloud Finchley继续沿用了Zuul1，没有采用Zuul2，一方面是因为Zuul2的改动比较大，从生态稳定性和兼容性上来讲，Zuul1有明显的优势，另一方面是因为Spring Cloud

已经基于Spring Boot 2.0和Reactor实现了Spring Cloud Gateway异步网关。

Zuul1目前与Eureka、Ribbon、Hystrix都实现了无缝的对接融合，具备更多额外能力，Zuul1已经部署在很多在生产环境中，经过大规模验证，同时Zuul1可以使用Servlet 3.0规范支持的AsyncServlet进行优化，可以实现前端异步，支持更多的连接数，达到和Zuul2一样的效果，但是不用引入太多异步复杂性。

### Spring Cloud Zuul1的接入与配置

1.添加Maven依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-Zuul</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
     </dependency>
</dependencies>
```

2.配置文件

```
spring:
  application:
    name: Zuul-service
eureka:
  client:
    service-url:
     defaultZone: http://localhost:8761/eureka
  instance:
    instance-id:
       ${spring.application.name}:${spring.cloud.client.ipAddress}:
       ${spring.application.instance_id:${server.port}}
    prefer-ip-address: true
server:
  port: 8080
```

3.定义Zuul网关启动类

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.Zuul.EnableZuulProxy;
//使用@EnableZuulProxy 注解开启 Zuul 的 API 网关服务
@SpringBootApplication
@EnableZuulProxy
public class ZuulApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulApplication.class, args);
    }
}
```

### Zuul的路由配置方式

默认情况下，在application.yml中配置Zuul的路由，主要有三种主要的路由配置方式。

● 单实例serviceId映射

配置文件如下：

```
Zuul:
    routes:
    user-service:
      path: /users/**
      serviceId: user-service
```

上面的路由配置是一个从/users/**到user-service服务的映射规则，我们可以把它简化为一个较简单的配置，映射规则与serviceId都不用写，可以通过http://localhost：8080/user-service/user/test？id=1来调用。Zuul会给user-service添加一个默认的映射规则/user-service/**，相当于：

```
Zuul:
  routes:
    client-a:
      path: /user-service/**
      serviceId: user-service
```

● 单实例URL映射

除了路由到服务，Zuul还能路由到物理地址，将serviceId替换成URL即可：

```
Zuul:
  routes:
    client-a:
      path: /users/**
       url: http://localhost:8080/users 的地址
```

● 多实例路由

默认情况下，Zuul会使用Eureka中集成的基本负载均衡功能，如果想使用Ribbon的负载均衡功能，就需要指定一个serviceId，此操作需要禁止Ribbon使用Eureka，在E版之后新增了负载均衡的配置。

```
Zuul:
  routes:
    client-a:
      path: /client/**
      serviceId: client-aribbon:
  eureka:
    enabled: false #禁止 Ribbon 使用 Eurekaclient-a:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
    listOfServers: 10.1.12.5:8080，10.1.25.6:8081
```

● forward本地跳转

Zuul支持在网关内部跳转，即本地跳转，在网关中写好一个接口，例如：

```
@RestController
public class TestController {
    @RequestMapping(value = "/testMethod", method = RequestMethod.GET)
    public String forwardMethod(Integer a){
        return "本地跳转:"+a;
    }
}
```

配置以下信息即可实现本地跳转：

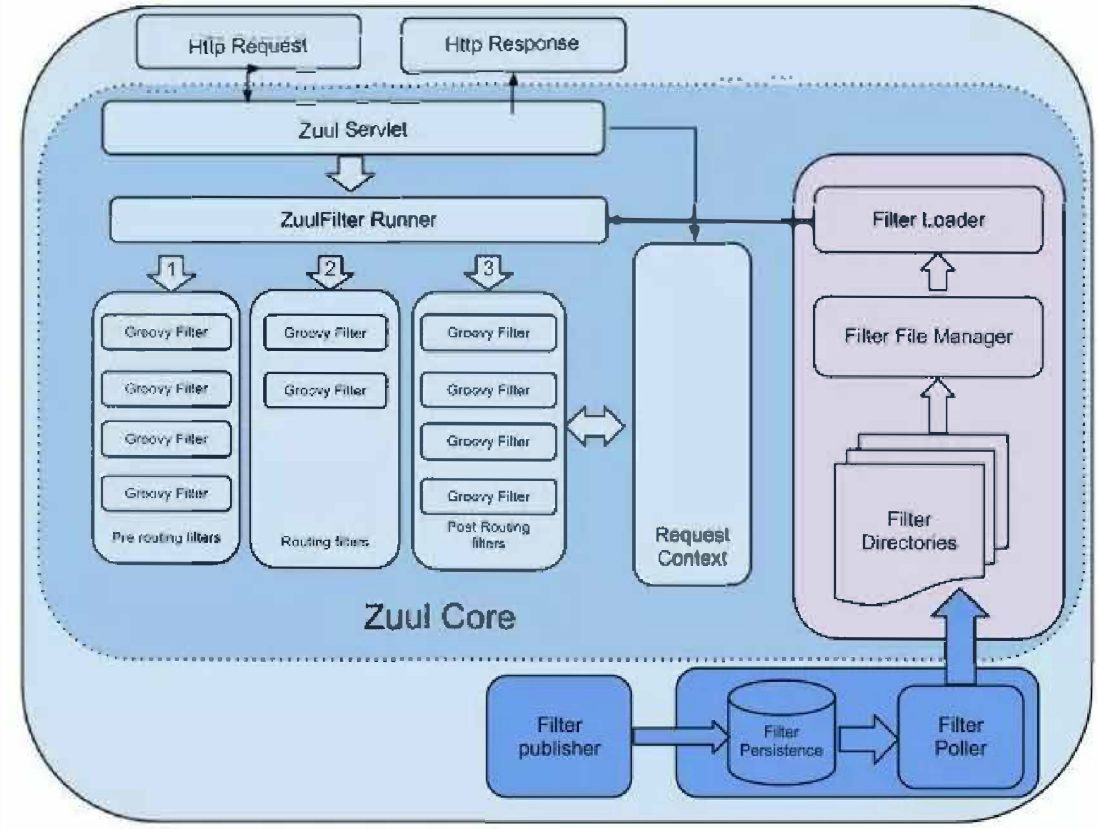
```
zuul:
  routes:
    client-a:
      path: /test/**
      url: forward:/forwardMethod
```

## 5.3.5 Zuul的主要工作原理

Zuul对HTTP请求的处理核心就是ZuulServlet类，而ZuulServlet本质上是符合Java EE规范的Servlet实现类。Zuul的核心处理逻辑由一系列Filter组成，而这些Filter全部封装在ZuulServlet类的Runner中，Runner在接收到HTTP请求后会依次调用ZuulFilter，而ZuulFilter则是Zuul在初始化时加载到FilterProcessor的Filter实例。

**ZuulFilter机制**

Zuul最主要的工作机制是基于ZuulFilter的链式调用请求机制，ZuulFilter之间没有直接的通信，它们之间通过一个RequestContext静态类来进行数据传递。RequestContext类中通过ThreadLocal变量来记录每个Request所需要传递的数据。ZuulFilter可以使用Java或者Groovy动态代码实现，同时使用Filter Loader的动态文件夹轮询扫描功能，可以将ZuulFilter动态加载到FilterProcessor中，实现动态Filter功能。下图是Zuul的一个整体架构图。




ZuulServlet是处理HTTP请求的核心类，它被嵌入Spring Dispatch机制中，从请求调用栈中可以发现它由Spring DispatchServlet处理，代码如下：

```
public class ZuulServlet extends HttpServlet {
    private ZuulRunner ZuulRunner;
    public void service(javax.servlet.ServletRequest servletRequest,
        javax.servlet.ServletResponse servletResponse) throws ServletException,
            IOException {
        try {
            init((HttpServletRequest) servletRequest,  (HttpServletResponse) servletResponse);
            RequestContext context = RequestContext.getCurrentContext();
            context.setZuulEngineRan();
            try {
                preRoute();
            } catch (ZuulException e) {
                error(e);
                postRoute();
```

```
                return;
            }
            try {
                route();
            } catch (ZuulException e) {
                error(e);
                postRoute();
                return;
            }
            try {
                postRoute();
            } catch (ZuulException e) {
                error(e);
                return;
            }
        } catch (Throwable e) {
            error(new ZuulException(e, 500, "EXCEPTION_" + e.getClass().getName()));
        } finally {
            RequestContext.getCurrentContext().unset();
    }
```

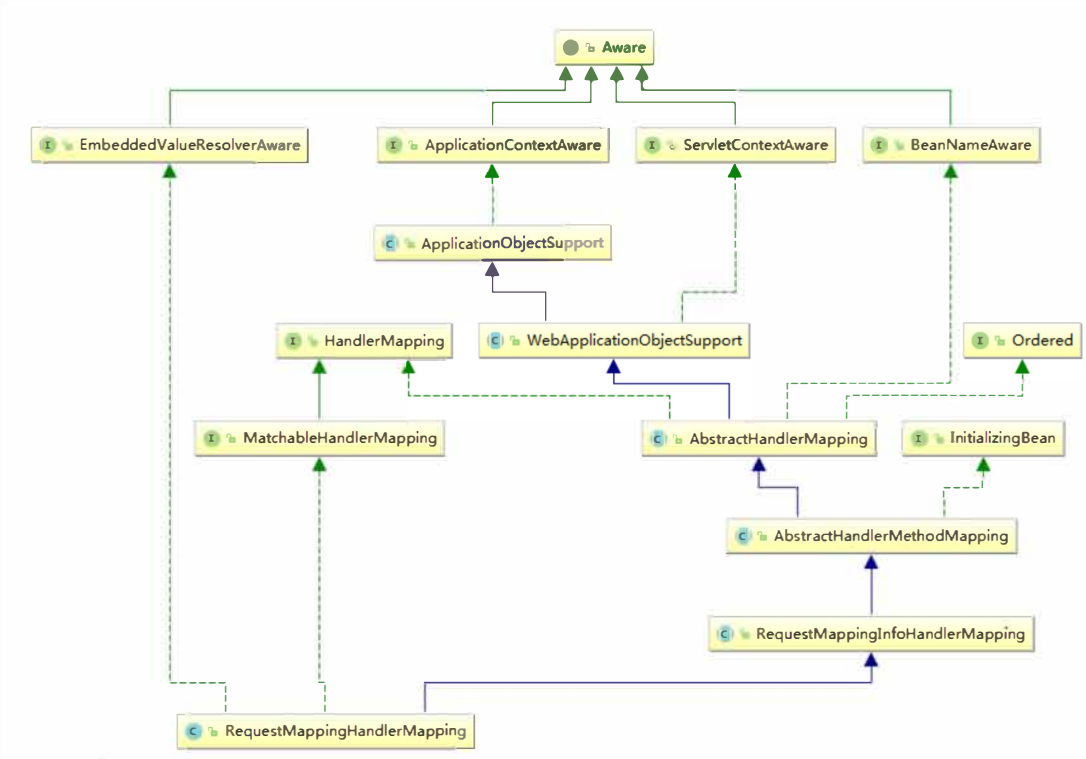
### Zuul路由机制

Zuul的核心工作原理主要有两部分：一部分就是上面讲的根据Zuul的Filter链式调用机制完成HTTP请求的Pre、Route、Post阶段的请求拦截和处理；另一部分是Zuul提供的路由机制，它可以实现根据HTTP请求到ZuulServlet的路由匹配映射，而路由匹配映射的工作主要是通过ZuulHandlerMapping模块完成的，在构造ZuulHandlerMapping时传入的RouteLocator是CompositeRouteLocator。ZuulHandlerMapping代码如下：

```
public class ZuulHandlerMapping extends AbstractUrlHandlerMapping {
    public ZuulHandlerMapping(RouteLocator routeLocator, ZuulController zuul) {
        this.routeLocator = routeLocator;
        this.zuul = zuul;
        setOrder(-200);
    }
    @Override
    protected Object lookupHandler(String urlPath, HttpServletRequest request)
        throws Exception {
        if (this.errorController != null && urlPath.equals(this.errorController.getErrorPath())) {
```

```
            return null;
        }
        if (isIgnoredPath(urlPath, this.routeLocator.getIgnoredPaths()))
            return null;
        RequestContext ctx = RequestContext.getCurrentContext();
        if (ctx.containsKey("forward.to")) {
            return null;
        }
        if (this.dirty) {
            synchronized (this) {
                if (this.dirty) {
                    registerHandlers();
                    this.dirty = false;
                }
            }
        }
        return super.lookupHandler(urlPath, request);
    }
    private void registerHandlers() {
        Collection<Route> routes = this.routeLocator.getRoutes();
        if (routes.isEmpty()) {
            this.logger.warn("No routes found from RouteLocator");
        }
        else {
            for (Route route : routes) {
                registerHandler(route.getFullPath(), this.zuul);
            }
        }
    }
}
```

从代码中我们可以看到，ZuulHandlerMapping继承了AbstractUrlHandlerMapping，Zuul的路由匹配使用了Spring MVC的映射功能。ZuulHandlerMapping复写了父类的lookupHandler方法。它的目的是将HTTP URL请求映射到对应的Controller，并将这个映射关系注册到Spring MVC中。如下图所示是ZuulHandlerMapping的类结构。



在ZuulHandlerMapping类的registerHandlers方法中，它将调用routeLocator.getRoutes方法注册所有路由对象。其中routeLocator的实现有三种，分别是SimpleRouteLocator、DiscoveryClientRouteLocator、DiscoveryClientRouteLocator。这三种RouteLocator最终都会由CompositeRouteLocator聚合统一管理，同时这些Route的处理类会设置为ZuulController。下面看一下ZuulController的源码实现。

```
public class ZuulController extends ServletWrappingController {
    public ZuulController() {
        setServletClass(ZuulServlet.class);
        setServletName("zuul");
        setSupportedMethods((String[]) null);
    }
    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        try {
            return super.handleRequestInternal(request,  response);
        }
        finally {
        RequestContext.getCurrentContext().unset();
        }
    }
}
```

查看ZuulController的父类ServletWrappingController的源码，它将请求都交给ZuulServlet来处理。由此可知，Zuul是如何将Route信息中配置的路由信息映射到ZuulController，而后由ZuulController委托给ZuulServlet来处理的。

最后，让我们梳理一下Zuul对HTTP请求的核心路由逻辑和Filter处理步骤：

（1）DispatcherServlet的doDispatch是HTTP请求的入口，通过AbstractHandlerMapping的getHandlerInternal方法找到对应的处理类，然后getHandlerInternal会调用lookupHandler方法。

（2）ZuulHandlerMapping覆写了抽象类AbstractHandlerMapping的lookupHandler方法，而registerHandlers方法遍历CompositeRouteLocator完成URL与对应的ZuulRoute映射规则的加载。

（3）在lookupHandler方法中首先判断urlPath是否被忽略，如果被忽略则返回null。

（4）判断路由规则有没有加载过或者更新过，如果没有加载或者更新，则重新加载。

（5）ZuulHandlerMapping注册处理器的时候使用的是ZuulController，也就说Zuul的所有请求对应的路由规则公共处理器都是ZuulController。最终这个处理器会将请求委托给ZuulServlet来处理，然后经过Zuul定义的和自定义的拦截器执行具体请求处理逻辑。

## 5.3.6 Zuul的插件机制及定制化开发

Zuul的另外一个重要机制就是Filter动态加载机制。Zuul支持的动态Filter由Groovy代码编写，动态管理Groovy的File目录变更并动态编译和加载。

● Filter类文件动态管理

Zuul通过FilterFileManager组件监控存放Filter文件的目录，定期扫描这些目录，如果发现有新Filter源码文件或者Filter源码文件有改动，则对文件进行编译和加载。FilterFileManager管理目录轮询的变化和新的Groovy过滤器。轮询间隔和目录在类的初始化中指定，并且轮询器将进行检查、更改和添加操作。下面的代码开启轮询线程。

```
poller = new Thread("GroovyFilterFileManagerPoller") {
    public void run() {
        while (bRunning) {
           try {
             //定时休眠 sleep(pollingIntervalSeconds * 1000);
             //扫描目录，监听目录变化
             manageFiles();
           } catch (Exception e) {
             e.printStackTrace();
           }
        }
     }
    };
    poller.setDaemon(true);
    poller.start(); //开启轮询线程
}
```

startPoller开启轮询线程以定时调用manageFiles方法扫描目录，监听目录变化的startPoller方法在FilterFileManager初始化时调用一次下面的代码完成目录扫描及检测。

```
void manageFiles() throws Exception， IllegalAccessException， InstantiationException {
    //1. 通过 getFiles 方法获取目录中的所有 Filter 文件
    List<File> aFiles = getFiles();
    //2. 通过 processGroovyFiles 方法处理所有 Filter 文件
    processGroovyFiles(aFiles);
}
List<File> getFiles() {
    List<File> list = new ArrayList<File>();
    //遍历配置的动态 Filter 文件存放目录，加载目录文件
    for (String sDirectory : aDirectories) {
        if (sDirectory != null) {
            File directory = getDirectory(sDirectory);
            File[] aFiles = directory.listFiles(FILENAME_FILTER);
            if (aFiles != null) {
                list.addAll(Arrays.asList(aFiles));
            }
        }
    }
    return list;
}
//处理所有动态 Filter 文件
void processGroovyFiles(List<File> aFiles) throws Exception，
    InstantiationException， IllegalAccessException {
     for (File file : aFiles) {
        FilterLoader.getInstance().putFilter(file);
     }
}
```

● Filter类文件动态编译

Zuul动态加载Filter文件，并通过编译器将文件编译成Class，目前Zuul通过定义DynamicCodeCompiler接口及Groovy编译的实现类GroovyCompiler来完成Groovy编写的Filter的

动态编译。DynamicCodeCompiler接口定义如下：

```
public interface DynamicCodeCompiler {
    /**
    * 通过源码字符串及 Filter 名称编译 Class
    */
    Class compile(String sCode， String sName) throws Exception;
    /**
    * 通过源码文件编译 Class
    */
    Class compile(File file) throws Exception;
}
```

GroovyCompiler类型加载的源码如下：

```
public class GroovyCompiler implements DynamicCodeCompiler {
    private static final Logger LOG = LoggerFactory.getLogger(GroovyCompiler.class);
    @Override
    public Class compile(String sCode， String sName) {
        GroovyClassLoader loader = getGroovyClassLoader();
        LOG.warn("Compiling filter: " + sName);
        Class groovyClass = loader.parseClass(sCode， sName);
        return groovyClass;
    }
    GroovyClassLoader getGroovyClassLoader() {
        return new GroovyClassLoader();
    }
    @Override
    public Class compile(File file) throws IOException {
        GroovyClassLoader loader = getGroovyClassLoader();
        Class groovyClass = loader.parseClass(file);
        return groovyClass;
    }
}
```

● Java类型动态Zuul Filter加载

基于Zuul的字节码加载机制和File文件扫描与类文件动态加载机制，也支持基于Java语言的动态Zuul Filter加载（因为篇幅所限，代码部分省略），具体实现代码已经在GitHub上开源，读者可以通过查找开源项目Sia-Gateway获取。

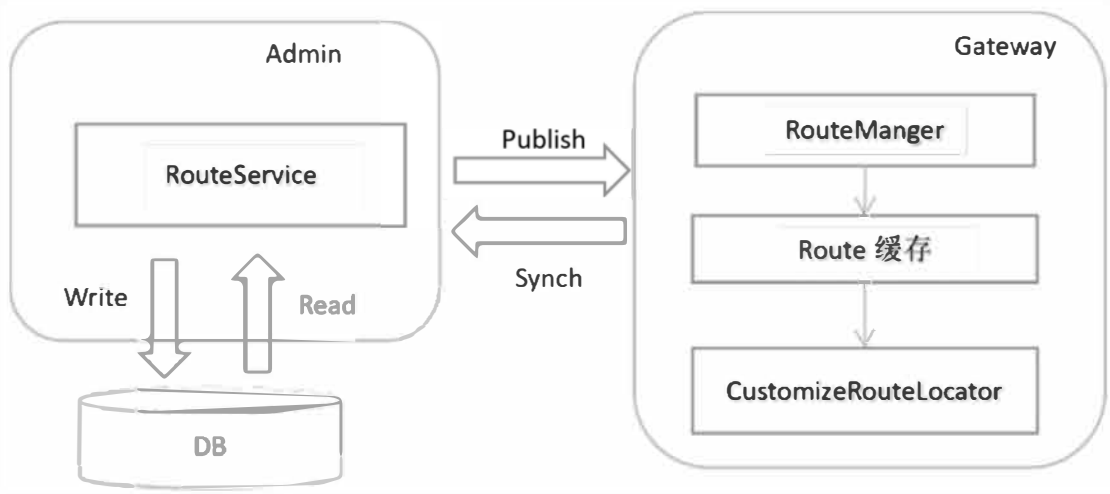
## 5.3.7 Zuul的动态路由

启动时Zuul会读取静态配置文件加载路由信息，将URL Path与路由映射关系建立好，提前加载到内存。在很多场景下，我们需要在不停止Zuul进程的前提下，完成路由映射规则的重新建立，这时候我们就需要动态路由（Dynamic Routing）功能，有两种实现动态路由的方式。

方式一：通过Spring Boot Acturator开启Zuul的Endpoint功能，它支持Refresh动态刷新配置文件，这种方式的好处是Zuul无须做任何修改，也不需要维护路由映射规则，缺点是没有可视化界面，维护起来比较烦琐。

方式二：覆写RouteLocator的List<Route>getRoutes（）方法，通过事件刷新机制，从数据库中读取路由配置规则。这是常用的Zuul动态路由解决方案，它可以轻松地实现可视化管理，减少

引入新的Spring Cloud组件的依赖绑定。Sia-Gateway（GitHub已开源项目）使用了基于MySQL DB的动态路由机制。如下图所示是Zuul动态路由架构图。




Zuul的动态路由思路及解决方案如下。

首先，Admin作为前端管理界面将用户对路由的添加、修改、删除等操作通过RouteService存储到DB中。DB中的存储结构如下图所示。

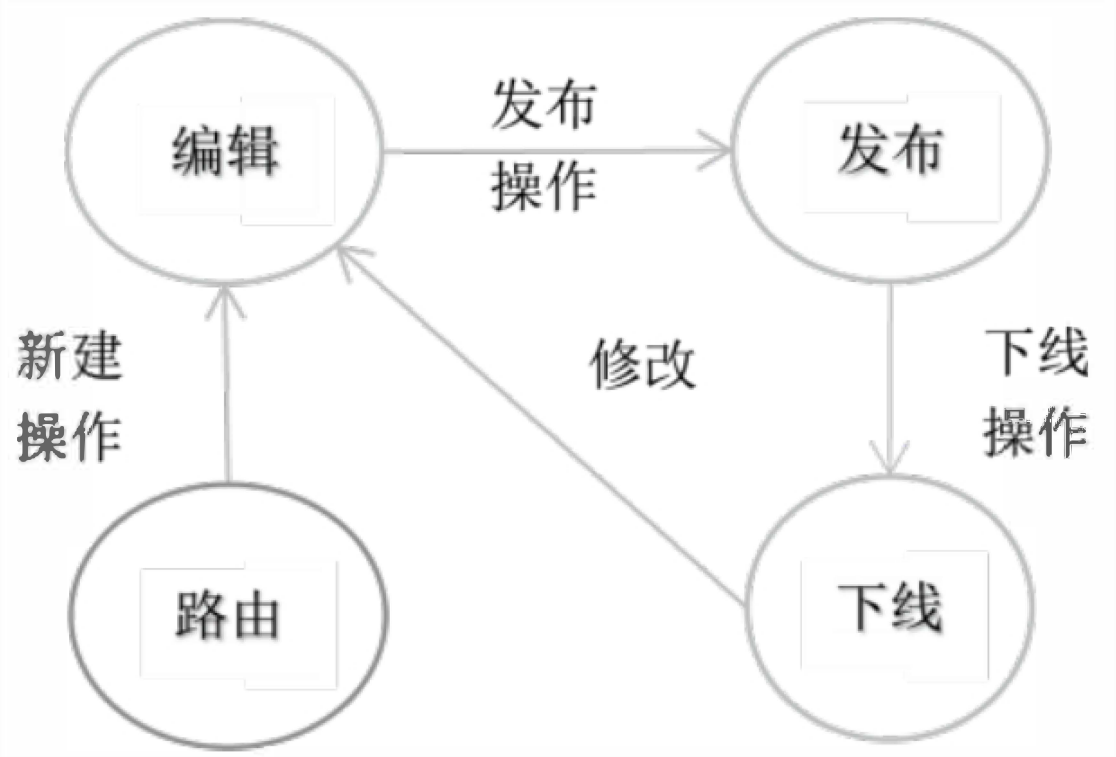
| id | path | serviceId | url | retryable | enabled | stripPrefix | apiName | zuulGroupName | routeStatus | strategy |
|---|---|---|---|---|---|---|---|---|---|---|
| [illegible]-book-test2 | /book2/** | | http://10.143.131.12:8091 | 1 | 1 | 1 | caantrvow-book-... | API-GATEWAY-CORE | ONLINE | SERVICEURL |
| hsibai-test | /aaaa/** | | 111 | 0 | 1 | 0 | hsibai-test | API-GATEWAY-CORE | EDIT | SERVICEURL |
| caanhuana-test1 | /caanhuana1/** | caantrvow-book-test | | 1 | 1 | 1 | caanhuana-test1 | API-GATEWAY-CORE | DOWNLINE | SERVICEID |
| caanhuana-test2 | /caanhuana2/** | | http://10.143.131.12:8091 | 1 | 1 | 1 | caanhuana-test2 | API-GATEWAY-CORE | ONLINE | SERVICEURL |
| cii | /a/** | aaa | [illegible] | 1 | 1 | 1 | cii | API-GATEWAY-CORE | DOWNLINE | SERVICEID |
| test | /test/** | test | | 1 | 1 | 1 | testApplication | API-GATEWAY-CORE | ONLINE | SERVICEID |
| test0910 | /test/* | test44 | | 0 | 1 | 1 | 测试0910 | API-GATEWAY-CORE | DOWNLINE | SERVICEID |
| toumira-atoservice-c... | /chinaRa/** | toumi-atoservice-chinara | [illegible] | 1 | 1 | 0 | 服务 | API-GATEWAY-CORE | DOWNLINE | SERVICEID |
| vttest | /acu/** | vtt-book-test | | 0 | 1 | 0 | | API-GATEWAY-CORE | ONLINE | SERVICEID |
| vttest4 | /book3/* | vtt-book-test | | 1 | 1 | 1 | | API-GATEWAY-CORE | ONLINE | SERVICEID |
| vttest5 | /acu/* | vtt-book-test | | 0 | 1 | 1 | | API-GATEWAY-CORE | DOWNLINE | SERVICEID |
| sydusadvasd | /o/** | downstream | [illegible] | [illegible] | 1 | 1 | APP-GATEWAY-... | APP-GATEWAY-CORE | ONLINE | SERVICEID |
| hello-book-dev | /test2/** | downstream | [illegible] | [illegible] | 1 | 1 | [illegible] | APPLICATION-GATEWAY-C... | ONLINE | SERVICEID |
| ceshi | /ceshi/? | d | d | 0 | 1 | 0 | ceshi | ceshi | EDIT | LISTOFSER... |

字段映射关系如下。

- id：标识路由的唯一ID，唯一主键，可以根据路由ID查找路由。
- ZuulGroupName：网关集群组名，标识这个新建的路由归属在哪个网关集群下面。
- apName：应用名称，标识路由的别名。
- path：匹配路径，新建路由的路径匹配Patten（例如/foo/**），所有发到/foo/**路径下的请求都会转发到这个路由下面。
- strategy：后端服务策略，后端路由有以下三种策略（只能选其中一种策略）。
  - SERVICEID策略：针对连接到Eureka上的应用，根据配置的ServiceID，网关会动态匹配一个后端服务。
  - SERVICEURL策略：针对非Eureka上的应用根据配置的URL映射到匹配的URL后端服务上。
  - LISTOFSERVICE策略：针对非Eureka上的应用，可以选择多个IP：PORT（通过逗号分隔）实现路由负载，默认使用RoundRobin方式实现负载均衡。例如“192.168.1.3：8070，192.168.1.2：8090”。
- SERVICEID：匹配一个新建路由的后端服务唯一标识。
- url：后端服务URL，匹配一个新建路由的后端服务URL物理地址。
- stripPrefix：前缀是否生效，标识这个路由在转发时是否需要删除前缀设置。

- 当stripPrefix=true时，主要的路由映射关系如下：
  http://127.0.0.1:8181/api/user/list->http://192.168.1.100:8080/user/list
- 当stripPrefix=false时，主要的路由映射关系如下：
  http://127.0.0.1:8181/api/user/list->http://192.168.1.100:8080/api/user/list).

其次，Admin对Route的状态管理类似状态机，网关节点的路由状态变更通过事件触发机制实现，以达到路由状态的一致性。如下图所示是路由（Route）状态在Admin上的状态流转图。




当Admin在修改Route状态时，它需要首先进行Route下线，当Route处于发布上线状态时，执行发布路由操作会调用publishRoute操作，publishRoute会调用Gateway的对外刷新接口，Gateway会从Admin同步最新的路由信息，并将Route设置为发布状态。

网关节点的路由管理机制主要由两部分组成，一部分通过自定义RouteLocator从Admin同步最新的路由状态，Admin会访问数据库，并返回给网关节点最新的路由状态信息，另一部分就是路由缓存状态管理，即同步更新。下面是代码实现。

```
public class CustomizeRouteLocator extends SimpleRouteLocator implements
    RefreshableRouteLocator {
    private ZuulProperties properties;
    @Autowired
    RouteOptService ropt;
    private StringBuffer routeBuffer;
    public StringBuffer getRouteBuffer() {
        return routeBuffer;
    }
    private AtomicBoolean refreshCalled = new AtomicBoolean(false);
    private AtomicBoolean initDone = new AtomicBoolean(false);
    public SagRouteLocator(String servletPath, ZuulProperties properties) {
        super(servletPath, properties);
        this.properties = properties;
    }
    @Override
    public void refresh() {
        doRefresh();
    }
    @SuppressWarnings("unchecked")
    @Override
    public Map<String, ZuulRoute> locateRoutes() {
        try {
            LinkedHashMap<String, ZuulRoute> routesMap;
            /**
             * Step1: 从 Cache 加载 Load 路由信息*/
            LinkedHashMap<String, ZuulRoute> routesMapfromCache =
             (LinkedHashMap<String, ZuulRoute>) LocalCacheManager.cacheMgr
                    .get(GatewayConstant.routeL1CacheKey);
            if ((routesMapfromCache != null) && (refreshCalled.get() == false)) {
                logger.debug("> Load RouteMap from local cache..");
                routesMap.putAll(routesMapfromCache);
            }
        else {
                //首先：将 refreshCalled 标志位重置为 flase
                if (refreshCalled.get()) {
                    refreshCalled.compareAndSet(true, false);
                }
                Map<String, ZuulRoute> routeMapFromDB =
            ropt.locateRoutesFromDB();
                //激活 Cache
                routeCacheEnabled();
                //将 DB-Route 信息同步到 Cache 中
                LocalCacheManager.cacheMgr.put(GatewayConstant.routeL1CacheKey,
                    routeMapFromDB);
                routesMap.putAll(routeMapFromDB);
            }
            return values;
        }
        catch (Exception e) {
            new GatewayException(ExceptionType.CoreException, e);
        }
        return null;
    }
    public void routeCacheEnabled() {
          if (!LocalCacheManager.cacheMgr.exists(GatewayConstant.routeL1CacheKey)) {
            LocalCacheManager.cacheMgr
                .enableL1Cache(GatewayConstant.routeL1CacheKey);
        }
        else {
            LocalCacheManager.cacheMgr.del(GatewayConstant.routeL1CacheKey);
        }
    }
    public AtomicBoolean getRefreshCalled() {
        return refreshCalled;
    }
    public void setRefreshCalled(AtomicBoolean refreshCalled) {
        this.refreshCalled = refreshCalled;
    }
}
```

说明 1#

说明 2#

说明 3#

说明 4#

- 说明1#：CustomizeRouteLocator是自定义路由加载的核心处理类，该类继承了SimpleRouteLocator，并实现了RefreshableRouteLocator接口。该类的主要功能是覆盖简单路由定位器的具体实现类，完成具体路由的加载策略及Zuul的内部事件刷新机制。
- 说明2#：refresh方法是RefreshableRouteLocator刷新事件的回调方法，该回调方法在ZuulHandlerMapping执行setDirty方法时被触发。ZuulHandlerMapping的代码如下：

```
public class ZuulHandlerMapping extends AbstractUrlHandlerMapping {
    public void setDirty(boolean dirty) {
        this.dirty = dirty;
        if (this.routeLocator instanceof RefreshableRouteLocator) {
         ((RefreshableRouteLocator) this.routeLocator).refresh();
    }
}
```

ZuulRefreshListener在@ZuulServerAutoConfiguration自动化配置类中被初始化，setDirty（true）将触发配置信息的重新加载并触发refresh方法，代码如下：

```
private static class ZuulRefreshListener implements ApplicationListener<ApplicationEvent> {
    @Autowired
    private ZuulHandlerMapping ZuulHandlerMapping;
    private HeartbeatMonitor heartbeatMonitor = new HeartbeatMonitor();
    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof ContextRefreshedEvent
            || event instanceof RefreshScopeRefreshedEvent
              || event instanceof RoutesRefreshedEvent) {
               this.ZuulHandlerMapping.setDirty(true);
        }
        else if (event instanceof HeartbeatEvent) {
            if (this.heartbeatMonitor.update(((HeartbeatEvent)event).getValue())) {
                this.zuulHandlerMapping.setDirty(true);
            }
        }
    }
}
```

- 说明3#：locateRoutes方法是SimpleRouteLocator的回调方法，下面是SimpleRouteLocator的具体实现，可以看到SimpleRouteLocator在doRefresh事件中回调了locateRoutes方法。

```
protected void doRefresh() {
    this.routes.set(locateRoutes());
}
protected Map<String, ZuulRoute> locateRoutes() {
     LinkedHashMap<String, ZuulRoute> routesMap = new LinkedHashMap<String,
        ZuulRoute>();
    for (ZuulRoute route : this.properties.getRoutes().values()) {
        routesMap.put(route.getPath(), route);
    }
    return routesMap;
}
```

- 说明4#：这部分代码是自定义路由加载的核心策略，我们设置了一个布尔型的原子变量：refreshCalled，当这个变量设置为true时，表示需要自定义Locator，强制从数据库中加载最新路由信息。当从数据库同步路由信息并将其存储到本地缓存中时，将refreshCalled设置为false，这样下次加载路由信息时，从缓存中加载就可以了，不需要从远端数据库中加载。这样做的好处是，可以明显提升维护本地路由信息的效率。RouteLocatorUpdater的作用就

是当Admin调用refreshRoute方法时，将refreshCalled原子变量设置为true，强制从数据库同步加载路由信息。

```
public class RouteLocatorUpdater {
    private static final Logger LOGGER = LoggerFactory.getLogger(RouteLocatorUpdater.class);
    @Autowired
    ApplicationEventPublisher publisher;
    @Autowired
    RouteLocator routeLocator;
    @Autowired
    SagRouteLocator routlocator;
    public void refreshRoute() {
        LOGGER.info("> RouteLocatorUpdater 路由刷新調用...");
        /**
         * Step1:将 refreshCalled 设置为 ture，强制 locateRoutes 从数据库获取路由信息
         */
        routlocator.getRefreshCalled().compareAndSet(false， true);
        /**
         * Step2: 发送通知事件，数据库同步路由信息，取得最新的路由表
         */
        RoutesRefreshedEvent routesRefreshedEvent = new
            RoutesRefreshedEvent(routeLocator);
        publisher.publishEvent(routesRefreshedEvent);
    }
}
```

最后一步，就是Admin从数据库获取路由数据信息，即从数据库中加载网关的路由信息，并返回给网关节点，作为最新的路由信息。

```
public List<RouteObj> locateRoutesFromDB() {
    LOGGER.info(">>> 从数据库加载路由信息到网关...");
    List<RouteObj> results = null;
    try {
        results = gatewayJdbcTemplate.query(SELECTROUTE，
                new BeanPropertyRowMapper<>(RouteObj.class));
    }
    catch (DataAccessException e1) {
        LOGGER.error("> locateRoutesFromDB exception:{}"， e1);
    }
    catch (Exception e1) {
        LOGGER.error("> locateRoutesFromDB exception:{}"， e1);
    }
    return results;
}
```

**注意：** 在网关获取动态路由信息的过程中，使用REST方式通过Admin代理获取路由信息，没有使用网关节点直接去数据库查询路由信息，主要有两个原因：

- 网关如果直接连接数据库，就会产生网关与数据库的强耦合关系，对于所有网关服务来说，都需要引入对MySQL数据库的依赖。
- 网关节点服务如果使用连接数据库的方式，那么就需要数据库的相关配置（用户名、密码）等信息，从数据安全的角度考虑，网关作为云原生的服务资源，应该尽量少暴露给后端用户，我们应该通过网关Admin服务统一管理数据库资源。

### 5.3.8 Zuul Filter扩展功能实现

在Zuul的工作原理中，我们已经看到Zuul主要采用基于Filter链的工作调用模式，通过自定义Filter机制可以动态扩展网关服务功能。网关的一个重要作用就是提供公共服务组件，而这些组件的功能也大都与Filter的拦截机制有关。下面我们以网关中经常会使用的灰度发布、服务限流为例，来说明如何通过Filter机制结合Ribbon和其他模块来实现。

**实现灰度发布**

目前常见的发布策略有蓝绿发布和灰度发布（金丝雀发布）。服务发布策略本质上是一种流量切换和流量导流策略。

- 蓝绿发布：在发布的过程中用户对服务的重启无感知，通常情况下通过新旧版本并存的方式实现，也就是说在发布的流程中，新的版本和旧的版本是相互热备的，通过切换路由权重的方式（非0即100）实现不同应用的上线或者下线。
- 灰度发布：在线上运行的服务中加入少量的新版本服务，然后从这少量的新版本服务中快速获得反馈，根据反馈决定服务最后的交付形态。

灰度发布要做的就是修改Ribbon的负载策略，基本的思路是，根据Eureka的Metadata配置设置自定义元数据（服务版本信息）与网关中设置的路由负载策略（Ribbon的Rule规则）进行匹配，选择符合路由条件的后端服务进行导流操作。

具体而言，因为Zuul使用的是基于同步线程的请求处理模式，所以我们可以通过ThreadLocal变量记录当前HTTP请求的特征数据，将特征数据与Ribbon中维护的后端服务实例的元数据进行模式匹配，再根据当前路由设置的负载策略就可以将HTTP请求映射到对应的后端服务实例。下面是主要的实现过程。

在实现灰度发布策略前，需要保证后端服务实例注册在Eureka中，并设定元数据的服务实例的版本信息。

```
spring.application.name = provide-test
server.port = 7770
eureka.client.service-url.defaultZone = http://localhost:1111/eureka/
#启动后直接将该元数据信息注册到 Eureka
#eureka.instance.metadata-map.version = v1
```

Zuul主要依靠Filter实现HTTP拦截，执行灰度发布过滤功能，其中ribbonHolder存储的是后端路由ID对应的路由策略信息。ribbonHolder在初始化时加载Admin配置的路由策略，篇幅所限，此处省略加载过程。routeribbonholder通过设置的路由策略可以动态执行蓝绿发布策略或者灰度发布策略，代码实现如下：

```
@Component
public class RibbonBRRouteFilter extends ZuulFilter{
    public Map<String, RouteRibbonHolder> ribbonHolder = new HashMap<>();
    //省略
    @Override
    public void run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        routeid = (String) ctx.get(PROXY_KEY);
        RouteRibbonHolder routeribbonholder;
        synchronized (mutex) {
            routeid = routeid.toUpperCase();
            routeribbonholder = ribbonHolder.get(routeid);
        }
        String strategy = routeribbonholder.getStrategy();
            switch (strategy) {
            /***
             * 蓝绿发布
             */
            case "Greenblue":
                String crrentVersion = routeribbonholder.getCurrentVersion();
                LOGGER.info("> RouteRibbonServiceImpl
                    routeid:[{}], crrentVersion:[{}]", routeid, crrentVersion);
                if (crrentVersion != null && !StringHelper.isEmpty(crrentVersion)) {
                    RibbonFilterContextHolder.getCurrentContext().add("version",
                        crrentVersion);
                }
                break;
            /***
             * 灰度发布
             */
            case "Canary":
                HttpServletRequest request = ctx.getRequest();
                String versionValue = null;
                /**
                 * step1:取 HTTP 头信息参数
                 */
                if (StringHelper.isEmpty(versionValue)) {
                    String context = routeribbonholder.getContext();
                    versionValue = request.getHeader(context);
                }
                /**
                 * step2:取 HTTP 头信息参数
                 */
                if (StringHelper.isEmpty(versionValue)) {
                    versionValue = request.getHeader(VERSION);
                }
                if (!StringHelper.isEmpty(versionValue)) {
                    LOGGER.info("runRibbon, Version Value is " + versionValue);
                    if (!StringHelper.isEmpty(versionValue)) {
                        RibbonFilterContextHolder.getCurrentContext()
                            .add("version", crrentVersion);
                    }
                }
        }
    }
}
```

说明#

**说明#：** 灰度发布Filter中的核心方法就是给RibbonFilterContextHolder设置当前线程上下文的后端服务版本信息。RibbonFilterContextHolder主要利用ThreadLocal变量来解决如何将灰度发布的信息传递给本地线程变量，然后当HTTP请求经过灰度发布Filter时，它可以通过Ribbon的元数据路由规则匹配对应的服务ID。下面是RibbonFilterContextHolder的实现代码：

```
public class RibbonFilterContextHolder {
    private static final ThreadLocal<RibbonFilterContext> CONTEXTHOLDER = new
        InheritableThreadLocal<RibbonFilterContext>() {
        @Override
        protected RibbonFilterContext initialValue() {
            return new DefaultRibbonFilterContext();
        }
    };
    public static RibbonFilterContext getCurrentContext() {
        return CONTEXTHOLDER.get();
    }
    /**
     * Clears the current context.
     */
    public static void clearCurrentContext() {
        CONTEXTHOLDER.remove();
    }
}
```

根据Ribbon的路由规则设置，Zuul基于Spring Boot的自动化配置机制，加载spring.factories实现自定义的路由规则配置加载，主要步骤如下。

- 第一步，在/META-INF/spring.factories下加载自动配置：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.XXX.gateway.ribbon.RibbonDiscoveryRuleAutoConfiguration
```

- 第二步，实现自动配置类：

```
@Configuration
@ConditionalOnClass(DiscoveryEnabledNIWSServerList.class)
@AutoConfigureBefore(RibbonClientConfiguration.class)
@ConditionalOnProperty(value = "ribbon.filter.metadata.enabled", matchIfMissing = true)
public class RibbonDiscoveryRuleAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public BaseDiscoveryEnabledRule metadataAwareRule() {
        return new MetadataMatchRule();
    }
}
```

然后将元数据与后端服务（Original Server）建立映射匹配规则：

```
public class MetadataMatchRule extends BaseDiscoveryEnabledRule {
    public MetadataMatchRule() {
        this(new MetadataMatchPredicate());
    }
    public MetadataMatchRule(BaseDiscoveryEnabledPredicate predicate) {
        super(predicate);
    }
}
```

- 第三步，根据元数据匹配后端服务（Original Server）规则：

```
public class MetadataMatchRule extends BaseDiscoveryEnabledRule {
    public MetadataMatchRule() {
        this(new MetadataMatchPredicate());
    }
    public MetadataMatchRule(BaseDiscoveryEnabledPredicate predicate) {
        super(predicate);
    }
}
```

● 第四步，基于元数据匹配的断言：

```
public class MetadataMatchPredicate extends BaseDiscoveryEnabledPredicate {
    @Override
    protected boolean apply(DiscoveryEnabledServer server) {
        final RibbonFilterContext context = RibbonFilterContextHolder.getCurrentContext();
        final Set<Map.Entry<String， String>> attributes = Collections
                .unmodifiableSet(context.getAttributes().entrySet());
        final Map<String， String> metadata = server.getInstanceInfo().getMetadata();
        return metadata.entrySet().containsAll(attributes);
    }
}
```

说明#

**说明 #：** 上述代码是元数据匹配的断言逻辑，可以看到它从Filter中设置的线程上下文中获得对应的后端服务版本信息，然后与HTTP请求中后端服务版本信息进行比对，当HTTP请求经过这个Filter时就会根据apply中设计的断言逻辑来选择符合条件的后端server，实现灰度发布功能。

**实现服务限流**

服务限流是微服务网关对API流量进行保护的一种常用手段，可以防止网络攻击，限制客户端的请求速度，在一定程度上可以保证后端服务不因为流量过载而宕机。后面的容错和隔离相关章节也会进一步说明限流的策略有哪些，下面说明如何通过Zuul+Guava RateLimiter实现服务限流功能。RateLimiter是Google开源的实现了令牌桶算法的限流工具。

首先引入spring-cloud-zuul-ratelimit组件的Maven依赖：

```
<dependency>
    <groupId>com.marcosbarbero.cloud</groupId>
    <artifactId>spring-cloud-zuul-ratelimit</artifactId>
    <version>2.0.4.RELEASE</version>
</dependency>
```

然后自定义实现Filter类：

```
@Component
public class RateLimitFilter extends ZuulFilter{
    //省略
    @Override
    public void run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        routeid = (String) ctx.get(PROXY_KEY);
        if (routeid == null) {
            URL routeHost = ctx.getRouteHost();
            if (routeHost != null) {
                String url = routeHost.toString();
                routlimitMap.putIfAbsent(url， RateLimiter.create(1000));
            }
        }
        RateLimiter rateLimiter = routlimitMap.get(routeid);
        LOGGER.info("routeid:" + routeid + "  rate:" + rateLimiter.getRate());
        if (!rateLimiter.tryAcquire()) {
```

```
            HttpStatus httpStatus = HttpStatus.TOO_MANY_REQUESTS;
            try {
                ctx.getResponse().setContentType(MediaType.TEXT_PLAIN_VALUE);
                ctx.getResponse().setStatus(httpStatus.value());
                ctx.getResponse().getWriter().append(httpStatus.getReasonPhrase());
                ctx.setSendzuulResponse(false);
            }
            catch (IOException e) {
                LOGGER.error("> RateLimitServiceImpl runRateLimit 报错:", e);
            }
        }
    }
}
```

RateLimiter基于Guava提供的令牌桶算法的实现类，可以依据系统的实际情况来调整生成token的速度。RateLimiter.create（1000）可以理解为：每秒往桶里放入1000个令牌。RATE_LIMITER.tryAcquire方法尝试获取令牌桶里的令牌，如果有令牌表示目前流量未达上限，则返回true，同时总的令牌数减1。如果没有令牌，说明令牌桶已达到阈值，则返回false，并设置HTTP状态码返回的信息。

## 5.3.9 Zuul源码解析

从源码分层的角度，Spring Cloud Zuul可以分为两个部分：Zuul–Core模块（ZuulServlet、FilterProcessor、ZuulFilter等）提供Zuul核心工作运行机制及对应的HTTP处理逻辑；spring–cloud–netflix–core（ZuulHandlerMapping、ZuulController等）是Spring部分的处理模块，负责把Zuul接入Spring Cloud体系。

**Zuul的源码分析**

Zuul应用开始于@EnableZuulProxy或@EnableZuulServer注解，所以一个不可缺少的步骤就是在启动程序中加入@EnableZuulProxy，代码如下：

```
@EnableCircuitBreaker
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Import(ZuulProxyMarkerConfiguration.class)
public @interface EnableZuulProxy {
}
```

其中，Zuul的自动化加载配置主要自动化加载如下两个配置类：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.cloud.netflix.zuul.ZuulServerAutoConfiguration，\
```

```
org.springframework.cloud.netflix.zuul.ZuulProxyAutoConfiguration
```

ZuulProxyMarkerConfiguration是一个标签类，通过自动配置类条件判断@ConditionalOnBean（ZuulServerMarkerConfiguration.Marker.class）使Zuul的自动化配置加载类生效，代码如下：

```
@Configuration
@EnableConfigurationProperties({ ZuulProperties.class })
@ConditionalOnClass(ZuulServlet.class)
@ConditionalOnBean(ZuulServerMarkerConfiguration.Marker.class)
@Import(ServerPropertiesAutoConfiguration.class)
public class ZuulServerAutoConfiguration {
//省略
}
```

ZuulProxyAutoConfiguration类完成Zuul自动装配功能，代码如下：

```
@Configuration
@Import({ RibbonCommandFactoryConfiguration.RestClientRibbonConfiguration.class,
    RibbonCommandFactoryConfiguration.OkHttpRibbonConfiguration.class,
    RibbonCommandFactoryConfiguration.HttpClientRibbonConfiguration.class,
    HttpClientConfiguration.class })
@ConditionalOnBean(ZuulProxyMarkerConfiguration.Marker.class)
public class ZuulProxyAutoConfiguration extends ZuulServerAutoConfiguration {
  //省略
}
```

- Zuul中属性相关源码

```
ZuulProperties ZuulProperties; //YAML 里的配置，Zuul 开头相关配置在这注入
ServerProperties server; //YAML 里的配置，Zuul 开头相关配置在这注入
ErrorController errorController; //处理错误的 Controller，自定义实现
```

- 重要Bean说明
  - ZuulController：接管所有的HTTP请求，将所有请求转交给ZuulServlet去处理。
  - ZuulRefreshListener：通过监听Spring Context发布机制事件，监听心跳消息。
  - CompositeRouteLocator：复合路由定位器，主要集成所有的路由定位器（如配置文件路由定位器、服务发现定位器、自定义路由定位器等）来定位路由。
  - SimpleRouteLocator：主要加载配置文件的路由规则。
  - RefreshableRouteLocator：路由刷新接口，实现此接口的路由定位器才能被刷新。
  - ZuulHandlerMapping：它将路由规则绑定到ZuulController上面。
- Zuul中默认Filter实现加载类
  - ServletDetectionFilter：它的作用是判断请求的来源，可以判断请求来自dispatcherServlet还是zuulServlet，并将判断结果存放到RequestContext中。
  - Servlet30WrapperFilter：它将原始请求进行包装，将原始的HttpServletRequest请求包装成Servlet30RequestWrapper类型。
  - FormBodyWrapperFilter：作用同Servlet30WrapperFilter一样，也是对请求的一个包装。
  - DebugFilter：用于动态开启Debug日志。
  - PreDecorationFilter：为当前请求做一些预处理，比如，进行路由规则的匹配、在请求上下文中设置该请求的基本信息及路由匹配结果的一些设置信息等，这些信息将是后续过滤器进行处理的重要依据。
  - RibbonRoutingFilter：通过Ribbon和Hystrix来向服务实例发起请求。
  - SimpleHostRoutingFilter：主要用来转发ServiceId（即非Eureka后端服务）为空的路由消息，它直接使用HttpClient来转发请求。
- RoutesEndpoint

  在引入Spring Boot Actuator时会新增一个routes端点，可以通过/routes查询具体的路由信息。
- 底层Web容器替换

spring-boot-starter-web模块默认使用Tomcat作为内嵌容器，如果我们想要切换为Jetty或者UnderTow，只需要添加相应容器的依赖即可。我们以Jetty举例，添加如下Maven依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

内嵌容器由EmbeddedServletContainerAutoConfiguration配置类决定：当类路径下有Jetty等类时，它将会注入JettyEmbeddedServletContainerFactory。内嵌容器的自动配置源码如下：

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(BeanPostProcessorsRegistrar.class)
public class EmbeddedServletContainerAutoConfiguration {
    /**
     * Nested configuration if Tomcat is being used.
     */
    @Configuration
    @ConditionalOnClass({ Servlet.class,  Tomcat.class })
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class,
        search = SearchStrategy.CURRENT)
    public static class EmbeddedTomcat {
        @Bean
        public TomcatEmbeddedServletContainerFactory
            tomcatEmbeddedServletContainerFactory() {
                return new TomcatEmbeddedServletContainerFactory();
        }
    }
    @Configuration
    @ConditionalOnClass({ Servlet.class,  Server.class,  Loader.class, WebAppContext.class })
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class,
        search = SearchStrategy.CURRENT)
    public static class EmbeddedJetty {
      @Bean
      public JettyEmbeddedServletContainerFactory jettyEmbeddedServletContainerFactory()
      {
        return new JettyEmbeddedServletContainerFactory();
      }
    }
}
```

## 5.4 负载均衡

负载均衡（Load Balance）是分布式网络环境中的重要机制，在微服务架构中，通过负载均衡可以实现系统高可用性、集群扩容等。负载均衡机制也决定着整个系统的性能和稳定性。

## 5.4.1 负载均衡机制

负载均衡可以说是高可用网络基础架构的关键组件，也是一种集群技术。它通过将网络流量分担到不同的网络服务器实现了业务容量的水平横向扩展，负载均衡不仅可以提升集群中不同服务器实例的负载性能，还可以保证集群整体的稳定性，也可以根据一定的负载策略实现灰度发布、蓝绿部署、A/B测试等。

通过硬件或者软件的方式负载均衡会维护一个服务列表清单，当用户发送请求时，它会将请求发给负载均衡器，后者根据（轮训、随机、加权）算法从可用服务列表中取出一台服务器的地址，进行请求转发，完成负载功能。

**负载均衡技术优势**

- 高性能：负载均衡技术将业务请求均衡地分配到多台设备或多条链路上，提高了整个系统的性能。
- 可扩展性：负载均衡技术可以方便地增加集群中设备或链路的数量，在不降低业务质量的前提下满足不断增长的业务需求。
- 高可靠性：单个甚至多个设备或链路发生故障不会导致业务中断，提高了整个系统的可靠性。
- 可管理性：大量的管理工作都集中在应用负载均衡技术的设备上，设备群或链路只需要通常的配置和维护即可。
- 透明性：对用户而言，集群等同于可靠性高、性能好的设备或链路，用户感知不到也不必关心具体的网络结构，增加和减少设备或链路均不会影响正常的业务。

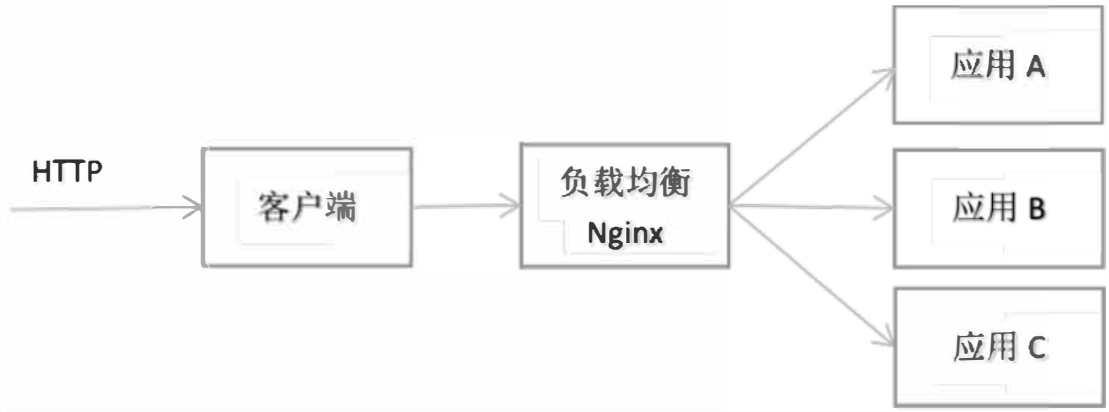
负载均衡有很多不同的分类方法，比如硬件负载均衡和软件负载均衡。F5就是比较常见的硬件负载均衡产品。F5以性能稳定著称，很多功能是软件负载均衡不具备的（应用交换、会话交换、状态监控、智能网络地址转换、通用持续性等）。然而缺点也比较明显，除了设备价格昂贵、配置冗余，更重要的是没有软件负载均衡灵活和不能满足定制化的需求。

软件负载均衡中，基于DNS的负载均衡是常用的解决方案。此外，Nginx可以说是Web应用服务领域使用最为广泛的前置软件负载系统，Nginx可以根据系统与应用的状况来分配负载，对于复杂的应用来说，有很好的性价比。LVS（Linux Virtual Server）是一个虚拟服务器集群系统，采用IP地址均衡技术和内容请求分发技术实现负载均衡。

负载均衡的另外一种分类方式是根据负载服务器清单列表存放的位置划分为服务端负载均衡和客户端负载均衡。

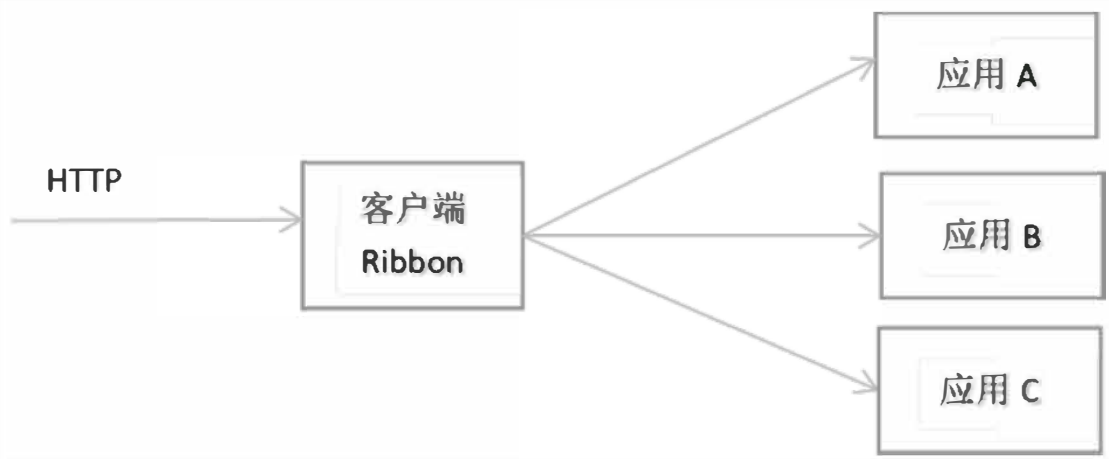
**服务端负载均衡**

类似于Nginx或者F5，它们都可以归类为服务端负载均衡，所有后端服务器地址清单通过注册中心或者集中配置的方式存储在服务器中。负载均衡器也可以通过心跳的方式动态调整服务列表清单的内容，从而使客户端总能访问到一个在线、可用的后端服务。服务端负载均衡如下图所示。




**客户端负载**

在客户端负载均衡中，客户端自己维护一份服务器地址清单。Ribbon就是客户端负载均衡工具，Ribbon缓存维护服务列表，如果服务实例注销或宕机，Ribbon能够自行将其剔除。Ribbon利用从Eureka Server中读取的服务信息列表（存储在本地即客户端中），在调用服务实例时合理地进行负载，直接请求到具体的微服务。Ribbon客户端负载均衡如下图所示。




另外，需要说明的是，我们也可以把Ribbon作为服务端负载均衡使用，当我们把Ribbon放入Zuul网关时，对于请求客户端而言，Zuul网关+Ribbon的组合就可以完成服务端的负载均衡。

## 5.4.2 四层与七层负载均衡

服务端负载均衡分为四层负载均衡和七层负载均衡。

**四层负载均衡**

四层负载均衡支持IPv4协议和IPv6协议，是基于流的服务端负载均衡，对报文进行逐流分发，将同一条流的报文分发给同一个服务器。四层负载均衡对基于HTTP的七层业务无法做到按内容进

行分发，限制了负载均衡的适用范围。四层负载均衡有NAT（Network Address Translation，网络地址转换）和直接路由（Direct Routing，以下简称DR）两种应用方式。

**七层负载均衡**

七层负载均衡只支持IPv4协议，是基于内容的服务端负载均衡，对报文的承载内容进行深度解析，包括HTTP、RTSP等，根据其中的内容进行逐包分发，按既定策略将连接导向指定的服务器，实现了使用范围更广泛的负载均衡。七层负载均衡仅支持NAT方式。

## 5.4.3 负载均衡算法

负载均衡算法决定了后端的哪些健康服务器会被选中。使用算法的前提条件是定义一个服务器列表，每个负载均衡的算法会从中挑出一个服务器作为算法的结果。常用的算法如下。

- 轮循均衡（Round Robin）：每一次来自网络的请求轮流分配给内部的服务器。该算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。
- 权重轮循均衡（Weighted Round Robin）：根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如，服务器A的权值被设计成1，服务器B的权值是3，服务器C的权值是6，则服务器A、B、C将分别接收到10%、30%、60%的服务请求。该算法能确保高性能的服务器的使用率更高，避免低性能的服务器负载过重。
- 随机均衡（Random）：把来自网络的请求随机分配给内部的多个服务器。
- 权重随机均衡（Weighted Random）：类似于权重轮循均衡算法，只是在处理请求时是一个随机选择的过程。
- 响应速度均衡（Response Time）：负载均衡设备对内部各服务器发出一个探测请求（例如Ping），然后根据内部各服务器对探测请求的最快响应时间来决定让哪个服务器来响应客户端的服务请求。该算法能较好地反映服务器的当前运行状态，但这里的最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务器间的最快响应时间。
- 最少连接数均衡（Least Connection）：客户端的每一次请求服务在服务器停留的时间可能会有较大的差异。随着工作时间的增加，如果采用简单的轮循或随机均衡算法，每个服务器上的连接进程可能会有极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部的每个服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使负载更加符合实际情况，更加均衡。
- 处理能力均衡：该算法将把服务请求分配给内部处理负荷（根据服务器CPU型号、CPU数量、内存大小及当前连接数等换算而成）最轻的服务器，由于考虑到了内部服务器的处理能力及当前网络运行状况，所以该算法相对来说更加精确，尤其适合运用到第七层（应用层）负载均衡中。

- DNS响应均衡：在Internet上，无论是HTTP、FTP还是其他的服务请求，客户端一般都是通过域名解析来找到服务器确切的IP地址的。在此均衡算法下，分处在不同地理位置的负载均衡设备收到同一个客户端的域名解析请求，在同一时间内把此域名解析成各自相对应服务器的IP地址（即与此负载均衡设备在同一位地理位置的服务器的IP地址）并返回给客户端，客户端将以最先收到的域名解析IP地址来继续请求服务，而忽略其他的IP地址响应。这种均衡策略适合应用在全局负载均衡下，对本地负载均衡是没有意义的。

### 5.4.4 Spring Cloud Ribbon

Ribbon的中文名称是“丝带”或者“蝴蝶结”，寓意Ribbon可以向丝带一样和其他组件配套使用。Ribbon可以和Eureka对接实现Eureka Client的客户端软件负载均衡，Eureka在发现后端服务数据后，Ribbon可以根据后端服务的元数据信息进行灵活的动态路由和负载均衡；也可以直接使用Ribbon提供的注解实现客户端软件负载均衡；当Ribbon应用在微服务网关Zuul中时，可以实现服务端的定制化路由转发和负载均衡。

Ribbon虽然不像服务注册中心、服务配置中心、微服务网关一样是微服务架构的运行时服务单元，但是Ribbon作为一个工具类框架，几乎出现在Spring Cloud微服务体系的每一个微服务和基础设施中，实现基于HTTP、TCP的客户端负载均衡。

**Ribbon的接入与配置**

1.添加Maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

spring-cloud-starter-ribbon是对spring-cloud-starter-netflix-ribbon包的封装，其包含的组件有：

- ribbon
- ribbon-httpclient
- spring-cloud-strater
- spring-cloud-starter-netflix-archaius
- spring-cloud-netflix-core

2.定义配置类

```
@Configuration
public class RibbonConfig {
    @LoadBalanced
    @Bean
    public RestTemplate RestTemplate() {
        return new RestTemplate();
    }
}
```

在该配置类中创建RestTemplate，并且使用@LoadBalanced注解。该注解使得RestTemplate具有了客户端负载均衡的能力。

3.实现Ribbon服务端服务

在pom.xml文件中需要加入Eureka-Client的依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

在application.yml文件中配置端口号和应用名称等信息：

```
spring.application.name: ribbon-provider
server.port: 2222
eureka.client.service-url.defaultZone: http://localhost:19002/eureka/
```

提供一个Ribbon服务端Controller接口，示例如下：

```
@RestController
public class DemoController {
    @GetMapping("/hello")
    public String hello(String name){
        System.out.println("invoke my service");
        return "hello，  " + name;
}
```

启动两个服务提供者实例，端口号分别为2222和2223。

4.定义Ribbon客户端消费服务

场景一：Ribbon与Eureka联合使用，Ribbon的服务端列表从Eureka注册中心动态发现获得，首先在pom.xml文件中引入如下主要依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

在application.yml文件中配置端口号和应用名称等信息：

```
spring.application.name: ribbon-consumer
server.port: 3333
eureka.client.service-url.defaultZone: http://localhost:19002/eureka/
```

Ribbon消费服务使用的接口示例如下：

```
/**
 * 开启客户端负载均衡
 */
@LoadBalanced
@Bean
RestTemplate RestTemplate() {
    return new RestTemplate();
}
@Autowired
RestTemplate RestTemplate;
@GetMapping("/test")
public String test() {
    System.out.println("enter test");
    String username = "zhangsan";
    String result =
    //ribbon-provider 为 Eureka 中注册节点名称
RestTemplate.getForEntity("http://ribbon-provider/hello?name=" + username,
    String.class).getBody();
    return result;
}
```

启动上述配置的各环境实例，访问消费服务接口，我们会发现在两个服务提供者的实例控制台轮流输出“invoke my service”信息。因为Ribbon默认的负载均衡策略为轮询，所以会交替访问两个实例。

场景二：在禁用Eureka的情况下，Ribbon的服务端列表需要在配置文件中指定。修改客户端的application.yml配置文件，添加配置Ribbon客户端的信息：

```
server:
   port: 8888
spring:
   application:
     name: my-client
   my-server:
 ribbon:
    eureka:
       enabled: false
 listOfServers: localhost:8090，localhost:9092，localhost:9999
 ServerListRefreshInterval: 15000
```

在上面的配置中，我们设置了eureka.enabled、listOfServers和ServerListRefreshInterval三个属性。Ribbon中的负载均衡器默认通过Eureka来发现服务端列表，在我们的例子中，因为没有使用Eureka，所以将ribbon.eureka.enabled属性设置为false。取而代之，我们通过listOfServers为Ribbon提供了静态的服务端列表。ServerListRefreshInterval属性表示服务端列表的刷新时间间隔，单位为毫秒。

然后，在ClientApplication类中，我们将RestTemplate切换到使用Ribbon客户端来获取服务端地址。

```
    @SpringBootApplication
    @RestController
    @RibbonClient(name = "my-server",  configuration = ServerConfiguration.class)
        public class ClientApplication {
        @LoadBalanced
        @Bean
        RestTemplate RestTemplate(){
            return new RestTemplate();
        }
        @Autowired
        RestTemplate RestTemplate;
        @RequestMapping("/hi")
        public String hi(@RequestParam(value="name",  defaultValue="Artaban") String name) {
         String greeting = this.RestTemplate.getForObject("http://my-server/greeting",
String.class);
         return String.format("%s,  %s!",  greeting,  name);
        }
        public static void main(String[] args) {
            SpringApplication.run(ClientApplication.class);
        }
    }
```

在RestTemplate Bean上加上@LoadBalanced注解，就能让RestTemplate在请求时具备负载均衡的能力。同时，对这个类添加@RibbonClient注解，并指明客户端的名字和额外的配置类。额外的配置类可以设置负载均衡规则等。

```
public class ServerConfiguration {
        @Autowired
        IClientConfig ribbonClientConfig;
        @Bean
        public IPing ribbonPing(IClientConfig config) {
            return new PingUrl();
        }
        @Bean
        public IRule ribbonRule(IClientConfig config) {
            return new AvailabilityFilteringRule();
        }
    }
```

我们可以通过创建具有相同名称的Bean来覆盖Spring Cloud Netflix的任何与Ribbon相关的Bean。这里我们覆盖了默认负载均衡使用的IPing和IRule。默认的IPing是一个NoOpPing（并没有真正Ping服务端实例），默认的IRule是一个ZoneAvoidanceRule。

IPing会发送Ping命令给一个URL来检测服务端是否正常，所以我们的服务端中有一个匹配路径的方法，提供对Ping命令的检查功能。

IRule使用的是AvailabilityFilteringRule实现，将会使用Ribbon内置的断路由功能来过滤掉“断开”的服务器。如果Ping一个服务器连接失败或者读取失败，Ribbon会认为这个服务器已经失效，直到它能够正常返回才会对其进行负载均衡访问。

**Ribbon测试流程**

（1）分别使用以下命令启动三个服务端实例。

```
$ mvn spring-boot:run
$ SERVER_PORT=9092 mvn spring-boot:run
$ SERVER_PORT=9999 mvn spring-boot:run
```

（2）启动客户端实例$mvn spring-boot：run。

（3）访问http://localhost：8888/hi，然后观察服务端实例，你会发现三个服务端实例都收到了Ribbon的Ping信息。

```
2017-05-03 16:49:05.290  INFO 6126 --- [nio-9092-exec-4]
com.rason.server.ServerApplication: Access /
2017-05-03 16:49:28.963  INFO 6126 --- [io-9092-exec-10]
com.rason.server.ServerApplication : Access /
```

（4）刷新浏览器，多次访问http://localhost：8888/hi，你会发现请求将会轮流被三个服务端实例的中一个处理。

```
2017-05-03 16:51:32.572  INFO 5998 --- [nio-8090-exec-6]
com.rason.server.ServerApplication    : Access /greeting
```

（5）关闭其中一个服务端实例，一旦Ribbon已经Ping到它没有响应，你会发现请求将在剩下的实例中进行负载均衡处理。

## 5.4.5 Ribbon的核心工作原理

下面我们从Ribbon的负载均衡核心组件、负载均衡策略、Ribbon的工作流程等方面讲解Ribbon的核心工作原理。

**Ribbon的负载均衡核心组件**

- Rule：用于从服务列表中选取服务的逻辑组件。
- Ping：在后台运行的确保服务可用性的组件。
- ServerList：服务列表，它可以是静态的也可以是动态的，如果是动态的（DynamicServer-ListLoadBalancer），将会启动一个后台线程定期刷新和过滤服务列表。

这些组件可以使用程序进行配置，也可以使用客户端配置属性进行配置，以下是配置文件中相关的属性名（注意带上前缀<clientName>.<nameSpace>.）：

- NFLoadBalancerClassName：配置ILoadBalancer的实现类。
- NFLoadBalancerPingClassName：配置IPing的实现类。
- NFLoadBalancerRuleClassName：配置IRule的实现类。
- NIWSServerListClassName：配置ServerList的实现类。
- NIWSServerListFilterClassName：配置ServerListFilter的实现类。

例如，将backend-service后端服务配置使用自带的RetryRule进行路由转发，下面是配置详情：

```
backend-service:
    ribbon:
        NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RetryRule
```

在我们的应用程序中Netflix为每个Ribbon客户端名称创建一个ApplicationContext，用于客户端自定义一些组件实现，包括：

- IClientConfig：为客户端或者负载均衡器保存客户端配置信息。
- ILoadBalancer：代表一个软件的负载均衡接口。
- ServerList：定义了如何获取可供选择的服务列表。
- IRule：描述负载均衡策略。
- IPing：说明如何执行服务器的周期性Ping。

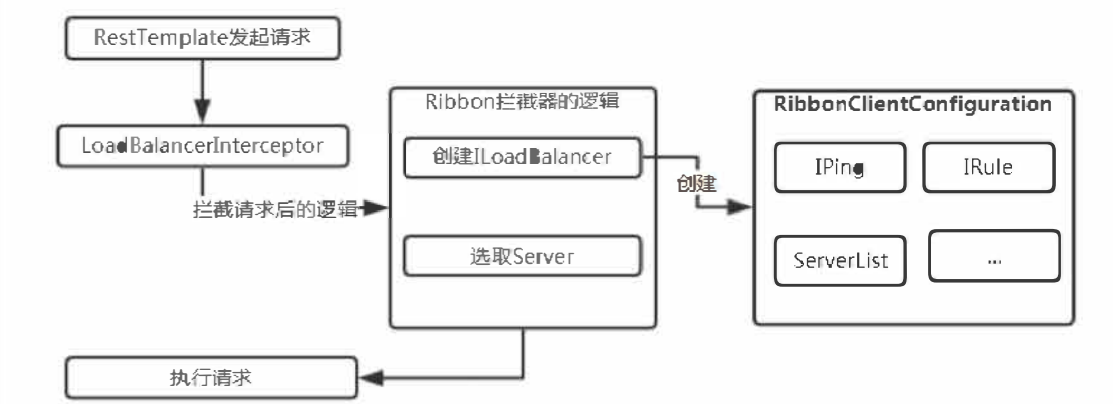
**Ribbon的负载均衡策略**

Ribbon客户端组件提供一系列完善的配置选项，比如连接超时、重试、重试算法等。Ribbon内置可插拔、可定制的负载均衡组件，下面是用到的一些负载均衡策略。

- 简单轮询负载均衡（RoundRobinRule）：这是最简单的通过循环法选择服务的负载均衡策略。它通常用作默认策略或更高级策略的后备策略。
- 可用服务过滤负载均衡（AvailabilityFilteringRule）：这个策略会跳过断路中的服务，或者高并发连接数的服务。默认情况下，如果RestClient最近三次连接均失败，则认为该服务实例断路，该实例会保持断路状态30s后进入回路关闭状态，如果此时仍然连接失败，那么等待进入关闭状态的时间会随失败次数的增加指数级增长。
- 加权响应时间负载均衡（WeightedResponseTimeRule）：这个策略会为每一个服务按响应时长分配权重，响应时间越长权重越低，权重代表了该服务会被选中的概率。
- 区域感知负载均衡：这个策略基于区域同源关系（Zone Affinity，也就是更倾向于选择发出调用的服务所在的托管区域内的服务，这样可以降低延迟，节省成本）选择目标服务实例。Ribbon使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个后端服务，前者使用一个区域为单位考察可用性，不可用的区域会被丢弃，从剩下的区域中选择可用的服务，AvailabilityPredicate用于过滤掉连接数过多的服务。
- 重试负载均衡（RetryRule）：当请求分发到集群中的一个服务后，请求连接失败或者响应超时，会重新请求以获取可用服务，可以选择重试当前服务节点，也可以改换其他节点。
- 随机负载均衡（RandomRule）：在现有服务之间随机分配流量的一种负载均衡策略。

**Ribbon的工作流程**

下图是Ribbon的工作流程简图，我们在源码分析中会进一步分析，了解Ribbon是如何拦截HTTP请求进行负载均衡的。




Ribbon首先会拦截所有标注@LoadBalance注解的RestTemplate，然后将Ribbon默认的拦截器LoadBalancerInterceptor添加到RestTemplate的执行逻辑中，这样每次执行RestTemplate时都会拦截HTTP请求。当有请求发生时，Ribbon默认会创建一个lloadBalancer实例。lLoadBalance在RibbonClientConfiguration中完成自动配置，它可以实现定制化的lRule、lPing、ServerList。当发生HTTP请求时，lloadBalancer在服务集群中选择一个服务，然后发送请求到后端服务。

## 5.4.6 Ribbon源码解析

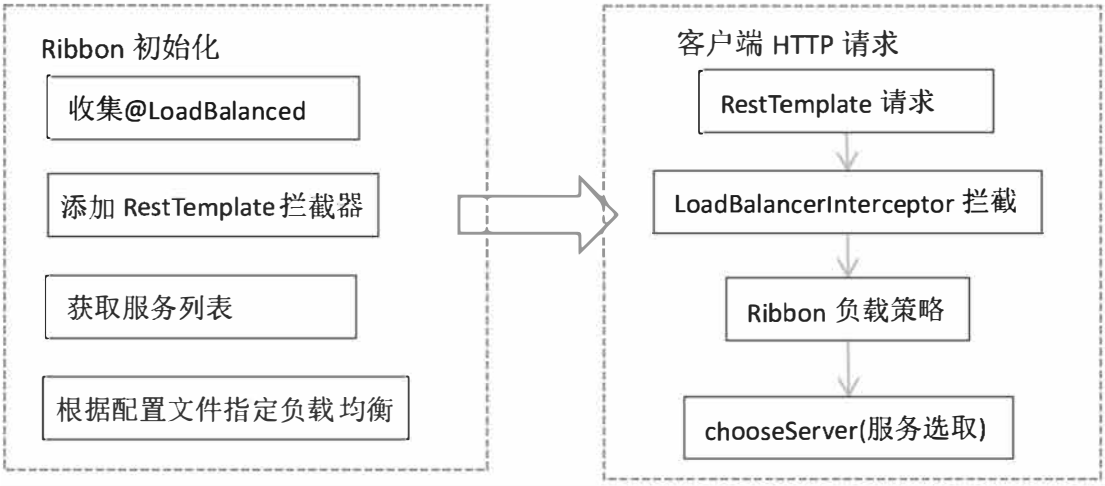
Ribbon的源码解析我们从@LoadBalanced开始讲起，添加@LoadBalanced注解后AsyncRestTemplate就具备了负载均衡的能力，代码如下：

```
@Bean(name = "loadbalanceAsyncRestTemplate")
@LoadBalanced
@ConditionalOnMissingBean(name = "loadbalanceAsyncRestTemplate")
public AsyncRestTemplate loadbalanceAsyncRestTemplate() {
    return new AsyncRestTemplate();
}
```

在初始化HTTP客户端时会加载Ribbon的拦截代码，同时根据配置文件中设置的负载均衡策略或者代码实现定制好的负载均衡策略，实现HTTP请求过程中的后端服务分发。所以源码解读可以分为两个部分。

- 初始化构造过程：获取@LoadBalanced注解标记RestTemplate或者AsyncRestTemplate，然后添加拦截器。
- 负载均衡服务选择过程：在Ribbon设定的负载均衡策略下，从服务集群中根据预定的负载均衡策略实现后端服务的选取及请求转发。

如下图所示是一个简要的Ribbon初始化及调用拦截HTTP请求实现负载均衡的流程。




**Ribbon的初始化过程**

@LoadBalanced注解对RestTemplate做了标记，使用Ribbon的自动化配置加载类实现对负载均衡客户端的加载，在生成的RestTemplate的Bean上添加注解后，它会配置LoadBalancerClient。首先我们看一下LoadBalancerClient的源码实现：

```
public interface LoadBalancerClient {
  ServiceInstance choose(String serviceId);
  <T> T execute(String serviceId,  LoadBalancerRequest<T> request) throws IOException;
  URI reconstructURI(ServiceInstance instance,  URI original);
}
```

**说明：** LoadBalancerClient是一个接口，里面有三个方法。

- ServiceInstance choose（String serviceId）方法，根据传入的serviceId（服务名），从负载均衡器中选择一个服务实例，服务实例通过ServiceInstance类来表示。
- execute方法，使用从负载均衡器中选择的服务实例来执行请求内容。
- URI reconstructURI（ServiceInstance instance，URI original）方法：用来重新构建URI。我们通过RestTemplate请求后端服务时会使用serviceId（服务名），这个方法会把请求的URI进行转换，返回host+port，再通过host+port的形式去请求服务。

从META-INF/spring.factories文件来看Ribbon的自动化加载机制，主要是Spring Common的LoadBalancerClient的初始化和加载过程：

```
# AutoConfiguration
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.cloud.client.CommonsClientAutoConfiguration, \
org.springframework.cloud.client.discovery.noop.NoopDiscoveryClientAutoConfiguration, \
org.springframework.cloud.client.hypermedia.CloudHypermediaAutoConfiguration, \
org.springframework.cloud.client.loadbalancer.AsyncLoadBalancerAutoConfiguration, \
org.springframework.cloud.client.loadbalancer.LoadBalancerAutoConfiguration, \
org.springframework.cloud.client.serviceregistry.ServiceRegistryAutoConfiguration, \
org.springframework.cloud.commons.util.UtilAutoConfiguration, \
org.springframework.cloud.client.discovery.simple.SimpleDiscoveryClientAutoConfiguration
# Environment Post Processors
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.cloud.client.HostInfoEnvironmentPostProcessor
```

### Ribbon的自动配置实现

下面是Ribbon的自动化配置实现：

```
@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
@EnableConfigurationProperties(LoadBalancerRetryProperties.class)
public class LoadBalancerAutoConfiguration {
    @LoadBalanced
    @Autowired(required = false)
    private List<RestTemplate> RestTemplates = Collections.emptyList();
    @Bean
    public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
            final List<RestTemplateCustomizer> customizers) {
            return new SmartInitializingSingleton() {
            @Override
            public void afterSingletonsInstantiated() {
            for (RestTemplate RestTemplate :
                LoadBalancerAutoConfiguration.this.RestTemplates) {
                    for (RestTemplateCustomizer customizer : customizers) {
                        customizer.customize(RestTemplate);
                    }
                }
            }
        };
    }
    @Autowired(required = false)
    private List<LoadBalancerRequestTransformer> transformers = Collections.emptyList();
    @Bean
    @ConditionalOnMissingBean
    public LoadBalancerRequestFactory loadBalancerRequestFactory(
        LoadBalancerClient loadBalancerClient) {
            return new LoadBalancerRequestFactory(loadBalancerClient,  transformers);
    }
    @Configuration
    @ConditionalOnMissingClass("org.springframework.retry.support.RetryTemplate")
        static class LoadBalancerInterceptorConfig {
    @Bean
    public LoadBalancerInterceptor ribbonInterceptor(
        LoadBalancerClient loadBalancerClient,
        LoadBalancerRequestFactory requestFactory) {
            return new LoadBalancerInterceptor(loadBalancerClient,  requestFactory);
        }
    @Bean
    @ConditionalOnMissingBean
    public RestTemplateCustomizer RestTemplateCustomizer(
        final LoadBalancerInterceptor loadBalancerInterceptor) {
        return new RestTemplateCustomizer() {
        @Override
        public void customize(RestTemplate RestTemplate) {
            List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                RestTemplate.getInterceptors());
            list.add(loadBalancerInterceptor);
            RestTemplate.setInterceptors(list);
            }};
        }
    }
}
```

说明 1#

说明 2#

说明 3#

说明 4#

- 说明1 #： Ribbon将所有标记@LoadBalanced注解的RestTemplate保存到一个List集合中。
- 说明2 #： Ribbon借助了Spring IoC容器的SmartInitializingSingleton机制。实现该接口后，当所有单例Bean都被初始化完成后，容器会调用afterSingletonsInstantiated实现RestTemplateCustomizer的customize定制化方法。

- 说明3＃：获取RestTemplate的interceptors，在构造LoadBalancerInterceptor时需要传入LoadBalanceClient实例参数，LoadBalanceClient是一个接口，具体实现类将实现choose（服务实例选择）和execute（请求转发执行）方法，这一步完成Ribbon负载均衡策略Bean的构造。
- 说明4：将说明3＃中构造的loadBalancerInterceptor Bean实例注入RestTemplate的定制化Bean中，这一步骤也会在说明2＃的afterSingletonsInstantiated方法中被调用，完成RestTemplate的定制化及与LoadBalancerInterceptor实例关联。

**Ribbon的重试策略**

对于Ribbon的重试策略，可以参考RetryTemplate类的实现，它可以实现Ribbon的重试策略对RestTemplate的拦截控制，代码如下：

```
@Configuration
@ConditionalOnClass(RetryTemplate.class)
public static class RetryInterceptorAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public RetryLoadBalancerInterceptor ribbonInterceptor(
        LoadBalancerClient loadBalancerClient， LoadBalancerRetryProperties properties，
        LoadBalancedRetryPolicyFactory lbRetryPolicyFactory，
        LoadBalancerRequestFactory requestFactory) {
        return new RetryLoadBalancerInterceptor(loadBalancerClient， properties，
            lbRetryPolicyFactory， requestFactory);
    }
    @Bean
    @ConditionalOnMissingBean
    public RestTemplateCustomizer RestTemplateCustomizer(
        final RetryLoadBalancerInterceptor loadBalancerInterceptor) {
            return new RestTemplateCustomizer() {
            @Override
            public void customize(RestTemplate RestTemplate) {
                List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                    RestTemplate.getInterceptors());
                list.add(loadBalancerInterceptor);
                RestTemplate.setInterceptors(list);
            }
        };
    }
}
```

**Ribbon的负载均衡行为逻辑**

Ribbon主要在Spring Cloud Netflix中完成负载均衡行为的初始化过程，这部分初始化主要依赖spring-cloud-netflix-core模块。

下面我们看一下Ribbon在Spring Cloud中是如何实现初始化的，首先看Ribbon的自动加载机制META-INF/spring.factories：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.cloud.netflix.archaius.ArchaiusAutoConfiguration，\
org.springframework.cloud.netflix.feign.ribbon.FeignRibbonClientAutoConfiguration，\
org.springframework.cloud.netflix.feign.FeignAutoConfiguration，\
org.springframework.cloud.netflix.feign.encoding.FeignAcceptGzipEncodingAutoConfiguration，\
org.springframework.cloud.netflix.feign.encoding.FeignContentGzipEncodingAutoConfiguration，\
org.springframework.cloud.netflix.hystrix.HystrixAutoConfiguration，\
org.springframework.cloud.netflix.hystrix.security.HystrixSecurityAutoConfiguration，\
org.springframework.cloud.netflix.ribbon.RibbonAutoConfiguration，\
org.springframework.cloud.netflix.rx.RxJavaAutoConfiguration，\
org.springframework.cloud.netflix.metrics.servo.ServoMetricsAutoConfiguration，\
org.springframework.cloud.netflix.Zuul.ZuulServerAutoConfiguration，\
org.springframework.cloud.netflix.Zuul.ZuulProxyAutoConfiguration
org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker=\
org.springframework.cloud.netflix.hystrix.HystrixCircuitBreakerConfiguration
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.cloud.netflix.metrics.ServoEnvironmentPostProcessor
```

下面是RibbonAutoConfiguration的实现，从源码中可以看到它会构造SpringClientFactory，LoadBalancerClient初始化构造需要SpringClientFactory作为参数。

```
@Configuration
@ConditionalOnClass({ IClient.class， RestTemplate.class， AsyncRestTemplate.class，
Ribbon.class})
@RibbonClients
@AutoConfigureAfter(name =
"org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration")
@AutoConfigureBefore({LoadBalancerAutoConfiguration.class，
AsyncLoadBalancerAutoConfiguration.class})
@EnableConfigurationProperties(RibbonEagerLoadProperties.class)
public class RibbonAutoConfiguration {
    //省略
@Bean
public SpringClientFactory springClientFactory() {
    SpringClientFactory factory = new SpringClientFactory();
    factory.setConfigurations(this.configurations);
    return factory;
    }
    @Bean
    @ConditionalOnMissingBean(LoadBalancerClient.class)
    public LoadBalancerClient loadBalancerClient() {
        return new RibbonLoadBalancerClient(springClientFactory());
    }
}
```

在Ribbon的自动化配置类中会通过RibbonClientConfiguration配置类获取YAML配置中的负载均衡配置，构造SpringClientFactory并生成LoadBalancer，代码如下：

```
@SuppressWarnings("deprecation")
@Configuration
@EnableConfigurationProperties
@Import({HttpClientConfiguration.class，
OkHttpRibbonConfiguration.class，
RestClientRibbonConfiguration.class，
HttpClientRibbonConfiguration.class})
public class RibbonClientConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public ILoadBalancer ribbonLoadBalancer(IClientConfig config，
        ServerList<Server> serverList， ServerListFilter<Server> serverListFilter，
        IRule rule， IPing ping， ServerListUpdater serverListUpdater) {
            if (this.propertiesFactory.isSet(ILoadBalancer.class， name)) {
                return this.propertiesFactory.get(ILoadBalancer.class， config， name);
            }
            return new ZoneAwareLoadBalancer<>(config， rule， ping， serverList，
                serverListFilter， serverListUpdater);
        }
    }
```

可以看到，在RestTemplate执行HTTP请求时是如何通过Ribbon设置的拦截机制构造HTTP客户端请求的。RestTemplate继承了InterceptingHttpAccessor，而父类InterceptingHttpAccessor提供了获取及添加拦截器的方法，代码如下：

```
public class RestTemplate extends InterceptingHttpAccessor implements RestOperations {
   //省略
}
```

InterceptingHttpAccessor抽象类的作用正是为我们添加自定义的拦截器：

```
public abstract class InterceptingHttpAccessor extends HttpAccessor {
   private List<ClientHttpRequestInterceptor> interceptors =
        new ArrayList<ClientHttpRequestInterceptor>();
   public void setInterceptors(List<ClientHttpRequestInterceptor> interceptors) {
      this.interceptors = interceptors;
   }
   public List<ClientHttpRequestInterceptor> getInterceptors() {
      return interceptors;
   }
   //省略
}
```

这段代码的实际拦截器的实例注入（依赖注入）过程其实来自上面的Ribbon自动化配置类LoadBalancerAutoConfiguration，在配置类中它已经完成了拦截器的注册。

下面我们看一下当拦截HTTP请求后，RestTemplate将会执行哪些操作。首先，RestTemplate执行HTTP请求，从RestTemplate的实现源码中，不难发现请求最终都会执行到doExecute方法中。查看doExecute的调用链路，我们发现它都会执行到LoadBalancerInterceptor的intercept拦截方法中，代码如下：

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {
   private LoadBalancerClient loadBalancer;
   private LoadBalancerRequestFactory requestFactory;
    //省略代码
   @Override
   public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
           final ClientHttpRequestExecution execution) throws IOException {
      final URI originalUri = request.getURI();
      String serviceName = originalUri.getHost();
      /**
       *拦截请求，并调用 loadBalancer.execute 方法
       *在该方法内部完成 Server 的选取。向选取的 Server
       *发起请求，并获得返回结果。
       */
      return this.loadBalancer.execute(serviceName,
        requestFactory.createRequest(request, body, execution));
   }
}
```

我们跟进到Ribbon的执行拦截逻辑，LoadBalancerClient.execute方法的具体代码如下：

```
public class RibbonLoadBalancerClient implements LoadBalancerClient {
    //省略
    @Override
    public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
        IOException {
          ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
          Server server = getServer(loadBalancer);
          if (server == null) {
              throw new IllegalStateException("No instances available for " + serviceId);
          }
          RibbonServer ribbonServer = new RibbonServer(serviceId, server,
            isSecure(server, serviceId),
            serverIntrospector(serviceId).getMetadata(server));
            return execute(serviceId, ribbonServer, request);
        }
        protected Server getServer(ILoadBalancer loadBalancer) {
            if (loadBalancer == null) {
                return null;
            }
            return loadBalancer.chooseServer("default");
        }
    }
}
```

从上述代码可知，Ribbon首先根据服务的serviceId调用getLoadBalancer方法得到ILoadBalancer。创建loadBalancer的过程可以理解为组装选取服务的规则、服务集群的列表、检验服务是否存活等特性的过程（加载RibbonClientConfiguration配置类）。这里的核心是getServer方法，根据ILoadBalancer来选取一个具体的Server，选取的过程会按照服务的负载均衡策略、服务列表、服务存活情况进行筛选判断，对我们自定义的负载均衡策略将执行chooseServer操作，最终根据这些约束选择一个后端的服务实例。

## 5.5 容错与隔离

微服务需要具备应对分布式环境的可容错性。在构建软件的开始阶段，就应该认识到网络和信息传递的不可靠性。我们需要对可能发生的故障设计出相应的软件隔离机制和措施，制定相应的容错策略，这个基本原则就是“Design for Failure”：为失败而设计。微服务架构可以在发生故障时通过合理的行为快速做出错误隔离和恢复机制，提供高可用性的服务。

### 5.5.1 隔离机制

在构建可容错软件系统的过程中要解决的本质问题就是故障隔离。在传统的单体架构下一旦应用出现故障，整个服务的可用性都会受到影响，因为所有模块都耦合在一个大的单体进程中，所以发生故障的位置很难确定，应用也很难从故障中恢复过来，这也是单体架构广遭诟病的一大原因。所以，我们需要将单体应用拆分成功能独立、相互隔离的微服务应用。

对于微服务架构而言，最关键的一个原则就是将系统划分成一个个相互隔离、自治的微服务，这些微服务通过定义良好的协议进行通信。我们希望存在隐患或者缺陷的服务不会对正常运行的模块或者服务产生不利的影响。

造船行业有一个专业术语叫作“舱壁隔离”，利用舱壁将不同的船舱隔离开来，如果某一个船舱进了水，那么就可以立即封闭舱门，形成舱壁隔离，其他船舱不受影响，船还可以正常航行。Docker通过“舱壁隔离”实现了进程的隔离，使得容器与容器之间不会互相影响。Spring Cloud Hystrix则实现了进程内部线程池的隔离，通过给每个依赖服务分配独立的线程池进行资源隔离，从而避免服务雪崩。

**进程隔离机制**

进程是传统操作系统中的重要隔离机制，每一个进程拥有独立的地址空间，提供操作系统级别的保护区。一个进程出现问题不会影响其他进程的正常运行，一个应用出错也不会对其他应用产生副作用。

随着微服务和容器技术的发展，我们说微服务的最佳载体正是容器。容器的本质就是一个进程，Docker鼓励一个容器只运行一个进程。这种方式非常适合以进程为粒度的微服务架构。也有人利用Docker容器作为轻量级的虚拟化方案，在单个容器中同时运行多个进程，这种使用方式往往会给应用带来隔离性问题和运行隐患。

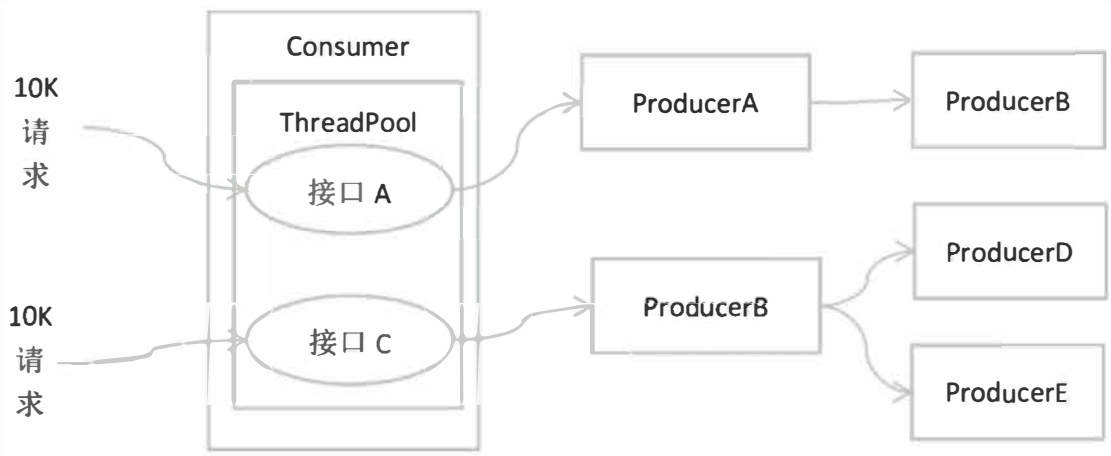
进程与进程之间的互相隔离实现了容器之间互不影响的特性。在启动一个容器时，本质上就是启动了一个进程，Linux通过Namespace技术实现容器之间的隔离，通过Cgroups来实现容器的资源控制。用户的应用进程实际上就是容器里PID=1的进程，也是其他后续创建的所有进程的父进程，这意味着没有办法同时运行两个不同的应用，除非你能事先找到一个公共的PID=1的程序来充当两个不同应用的父进程。

容器作为一种沙箱技术，提供了类似集装箱的功能，把应用封装起来，这样被装进集装箱的应用就可以方便地被复制和移动。容器具备了天然的“不共享任何资源”的特性，所以可以认为容器是独立的服务主体。容器在与其他容器交互时，需要使用基于网络的消息通信机制，摆脱了模块之间的强依赖耦合。我们在本书后面的内容中会进一步介绍容器隔离机制的细节。

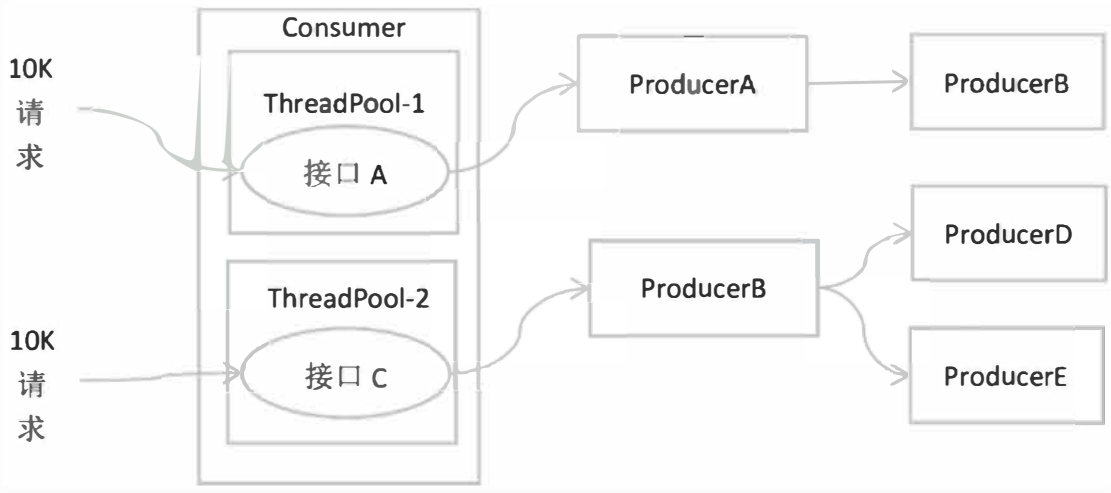
**线程隔离机制**

进程虽然具有较好的隔离性，但是进程之间交互需要跨进程边界，进程在数据共享方面存在数据传输的开销。而多线程并发编程模式可以最大限度地提高程序的并行度，线程作为操作系统最小的调度单元可以更好地利用多处理器的优势，恰当地使用线程可以降低开发和维护的开销，并且可以提高复杂应用的性能，当前在软件系统中大量使用了多线程编程模型。

线程隔离主要指线程池的隔离，在应用系统内部，将不同请求分类发送给不同的线程池，当某个服务出现故障时，可以根据预先设定的熔断策略阻断线程的继续执行。来看如下图所示的案例。




在Consumer模块中，接口A和接口C共用相同的线程池，当接口A的访问量激增时，接口A因为与接口C共用相同的线程池，所以势必影响接口C的效率，进而可能产生雪崩效应。如果我们使用线程隔离机制，那么可以将接口A和接口C做一个很好的隔离，如下图所示。




使用线程隔离机制将使线程池内可能出现问题的线程和其他线程隔离运行在一个独立的线程池中，一旦此线程出现问题，不会影响其他线程的运行，防止雪崩效应的产生。

**信号量隔离机制**

信号量semaphore是一个并发工具类，用来控制可同时并发的线程数。其内部维护了一组虚拟许可，通过构造器指定许可的数量，每次线程执行操作时先通过acquire方法获得许可，执行完毕再通过release方法释放许可。如果无可用许可，那么acquire方法将一直阻塞，直到其他线程释放许可。

在信号量隔离机制下，接收请求和执行下游依赖在同一个线程内完成，不存在线程上下文切换所带来的性能开销。

信号量的资源隔离只是起到开关的作用，比如，服务A的信号量大小为10，那么就是说它同时只允许有10个Tomcat线程来访问服务A，其他请求都会被拒绝，从而达到资源隔离和限流保护的作用。

## 5.5.2 微服务的风险

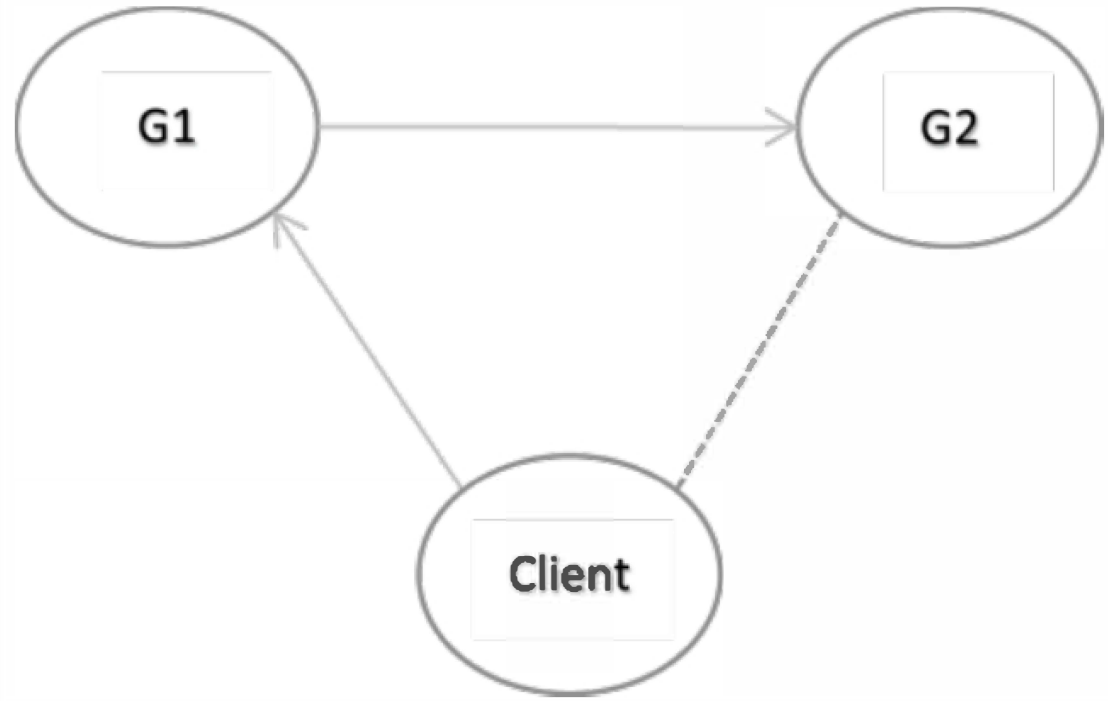
在微服务架构下，传统的单体应用被拆分为多个服务后，服务的数量变多了，同时之前单体架构下进程内部的方法调用转变为分布式网络环境下的远程调用，因此构建分布式微服务系统带来了额外的开销。

- 性能：分布式系统是跨进程、跨网络的调用，受网络延迟和带宽的影响。
- 可靠性：由于高度依赖于网络状况，任何一次的远程调用都有可能失败，随着服务的增多还会出现更多的潜在故障点。因此，如何提高系统的可靠性、降低因网络导致的故障率是系统构建的一大挑战。
- 异步：异步通信大大增加了功能实现的复杂度，并且有定位难、调试难等问题。
- 数据一致性：要保证分布式系统的数据强一致性成本是非常高的，需要在C（一致性）、A（可用性）、P（分区容错性）三者之间做出权衡。

下面是微服务场景下我们会面临的常见风险。

**分布式固有的复杂性**

分布式系统的CAP理论告诉我们分区容错（Partition Tolerance）是不可避免的。如下图所示，G1和G2是两台分布式跨网络的服务器，G1向G2发送一条信息，G2可能无法收到。所以，对于分布式系统，只要具有大于零的概率，根据墨菲定律你就不能避免它发生。




分布式系统中多个计算机在进行通信过程中，由于网络的不可靠性，每次网络通信都会伴随网络不可用的风险。即便网络通信正常，服务的延迟也会远远大于单机下的调用延迟。所以消息丢失或者延迟是非常普遍的情况。

基于上述分析，我们在进行微服务系统架构设计的时候，就必须考虑当网络分区出现时，分布式系统就会出现局部服务失效问题，我们需要做出相应的服务容错处理。

**服务的依赖性**

在微服务架构场景下，除了微服务自身缺陷造成的服务不可用，对基础设施的依赖、对上下游微服务的依赖都可能造成依赖错误的发生。相比服务自身失败而言，服务对外部平台的依赖往往更加难以发现和处理，服务依赖失败也是在设计微服务时需要重点考虑的失败因素。同时，细粒度的服务也增加了不同服务之间的依赖和级联影响，因为服务依赖失败而造成的失败扩散，或者核心服务对非核心业务的依赖，都会造成依赖风险。

**雪崩效应**

我们常把“基础服务故障”导致“级联故障”的现象称为雪崩效应。雪崩效应描述的是服务生产者不可用导致消费者不可用，并将不可用逐渐放大的过程。

软件系统会发生各种错误，如硬件设施的损坏、软件系统内存溢出和资源的OOM等问题，在微服务架构下，我们可能还会遇到的问题就是雪崩效应。微服务架构的应用系统通常包含多个服务层。微服务之间通过网络进行通信，从而支撑起整个应用系统，因此微服务之间难免存在依赖关系。在公司内部的网络拓扑中，我们会发现一个HTTP服务的请求往往会经历很多个服务节点，例如，一个电商平台的下单场景往往会经过如下链路：App客户端→API网关→账单服务→支付服务

→库存服务。但是，在错综复杂的网络中各式各样的问题（硬件原因、软件原因）都会造成系统异常，难免有些请求会失败，雪崩效应就此产生。

- 硬件故障：比如宕机、机房断电、光纤被挖断等。
- 流量激增：比如异常流量或者用户重试导致的系统负载升高。
- 程序Bug：比如代码循环调用的逻辑问题、资源未释放引起的内存泄漏问题等。
- 线程同步等待：系统间经常采用同步服务调用模式，核心服务和非核心服务共用一个线程池和消息队列，如果一个核心业务线程调用非核心线程，这个非核心线程交由第三方系统完成，当第三方系统本身出现问题时，导致核心线程阻塞，一直处于等待状态，而进程间的调用是有超时限制的，最终这条线程将断掉，引发雪崩效应。

总之，面对微服务架构场景下的风险，我们需要一定的应对措施和容错策略，下面是我们总结的容错管理的主要原则：

- 按照进程或者线程进行软件的划分和隔离。
- 将错误限制在可以快速失败（fail–fast）的软件模块中，避免错误模块对整体系统造成影响。
- 使用备份机制或者集群方式应对硬件或者软件的故障。

### 5.5.3 降级保护

服务降级通常是针对非核心业务在业务流量激增情况下一种服务策略，通过服务降级可以保证核心业务的顺利进行。如果是主动降级，通常会返回一个默认值，被动降级是指当发现异常时，为了控制异常的影响范围而触发的自动服务降级。

**降级分类**

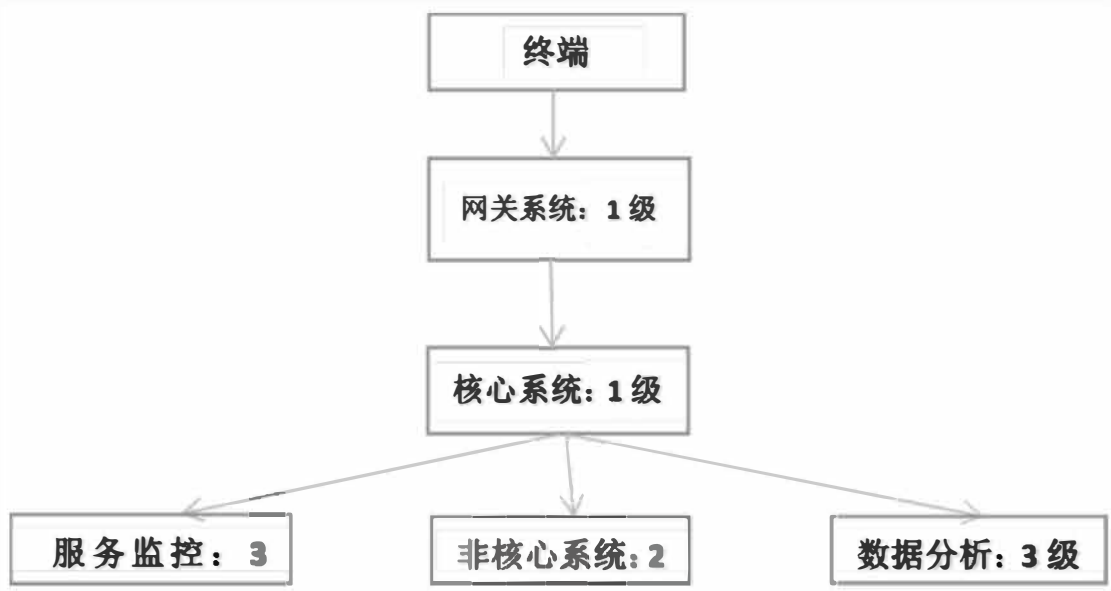
- 超时降级：配置好超时时间和超时重试次数，并使用异步机制探测恢复情况。
- 失败次数降级：主要针对一些不稳定的API，当失败调用次数达到一定阈值时自动降级，同样要使用异步机制探测恢复情况。
- 故障降级：比如要调用的远程服务“挂”了（网络故障、DNS故障、HTTP服务返回错误的状态码、RPC服务抛出异常），系统可以直接降级。降级后的处理方案有：采用默认值（比如库存服务“挂”了，返回默认现货）、兜底数据（比如广告服务“挂”了，返回提前准备好的一些静态页面）、缓存（使用之前暂存的一些数据）。
- 限流降级：在高并发或者秒杀场景中，系统可能会因为访问量太大而导致崩溃，此时开发者会使用限流机制来限制访问量，当达到限流阈值后，后续请求会被降级。

**降级与熔断的区别**

降级和熔断这两个概念很容易被等同，它们最终达到的效果都是保护系统，都是为防止系统整体崩溃采用的技术手段。从用户体验的角度看，二者也有相似之处，都是某些功能暂时丧失可用

性。但是，二者之间还是存在明显的差别的，具体如下：

- 触发条件的差别。熔断通常是自动触发的，而降级除了异常情况下的熔断，也可以根据程序中预先设置的代码逻辑进行手动降级。
- 分级的差别。熔断通常依赖一个系统中的整体的框架处理逻辑，每个微服务都需要无差别地具备熔断的特性，而降级则需要针对业务的优先级和重要性进行分级。对于核心业务，一般设置的级别较高；而对于核心系统下游的非核心业务，如果是弱依赖关系，那么级别可以相对降低，但是如果它们具有强依赖关系，那么非核心业务此时也会升级为核心业务。在下图中，网关系统处于调用链路的入口，优先级高；数据分析、服务监控都是旁路系统，属于辅助性功能，相对优先级较低。




下面总结一下Spring Cloud中两种简单的实现服务降级的方法。

- 在FeignClient中实现服务降级：

```
@FeignClient(value = "order", configuration = OrderClientConfiguration.class, fallback
= OrderClientFallback.class)
public interface OrderClient {
    //查询购物订单扣费状态(内)
    @RequestMapping(value = "/order/queryOrderCost", method = RequestMethod.GET)
    QeuryOrderCostResVo queryOrderCost(@RequestParam(value = "orderId") String
        orderId) throws InternalApiException;
}
```

- 在Hystrix中一般使用fallbackMethod实现服务降级：

```
//@HystrixCommand 表示当调用服务失败时，会调用 fallbackMethod 方法
@Servicepublic class ItemService {
    @Autowired
    private RestTemplate RestTemplate;
    @HystrixCommand(fallbackMethod = "fallbackMethod")
    public List<Item> getItemList(){
        return RestTemplate.getForObject("http://feign-provider/item/list", List.class);
    }
    public List<Item> fallbackMethod(){
        List<Item> list = new ArrayList<>();
```

```
            list.add(new Item(1, "test"));
            return list;
        }
    }
```

## 5.5.4 限流保护

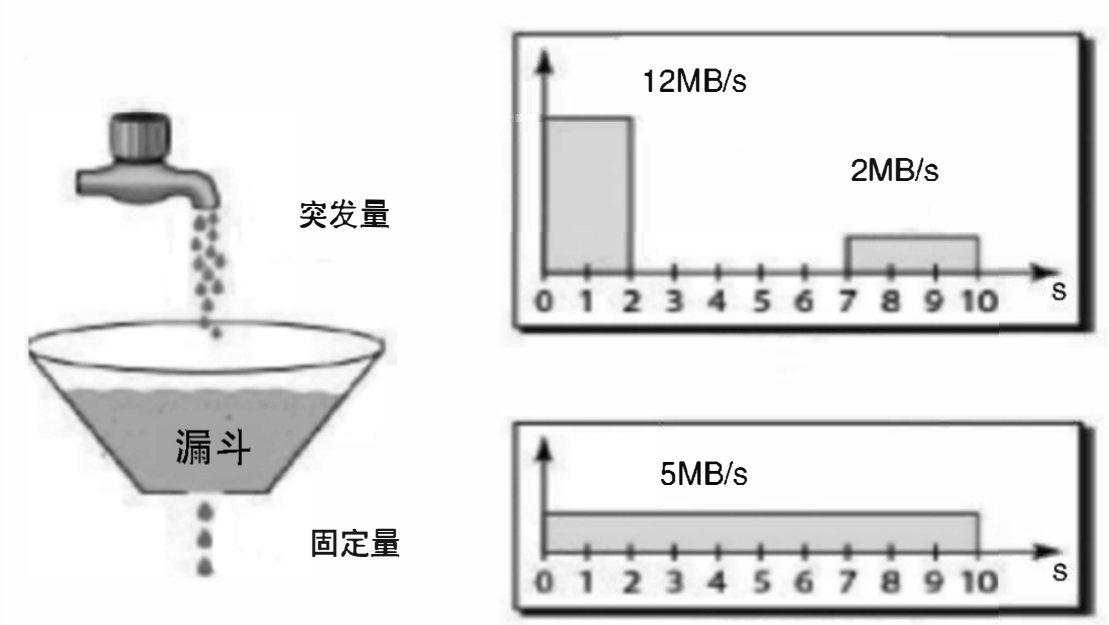
限流的目的是保护系统不被大量请求冲垮，通过限制请求的速度来保护系统。在电商的秒杀活动中，限流是必不可少的一个环节。

**计数器**

比较简单的限流做法是维护一个单位时间内的计数器，每次允许请求计数器都加1，当单位时间内计数器累加到设定的阈值后，之后的请求都被拒绝，直到超过单位时间，再将计数器重置为零。此方式有一个弊端：如果在单位时间1s内允许100个请求，10ms已经通过了100个请求，那后面的990ms只能拒绝请求，我们把这种现象称为“突刺现象”。常用的更平滑的限流算法有两种：漏桶算法和令牌桶算法。

**漏桶算法**

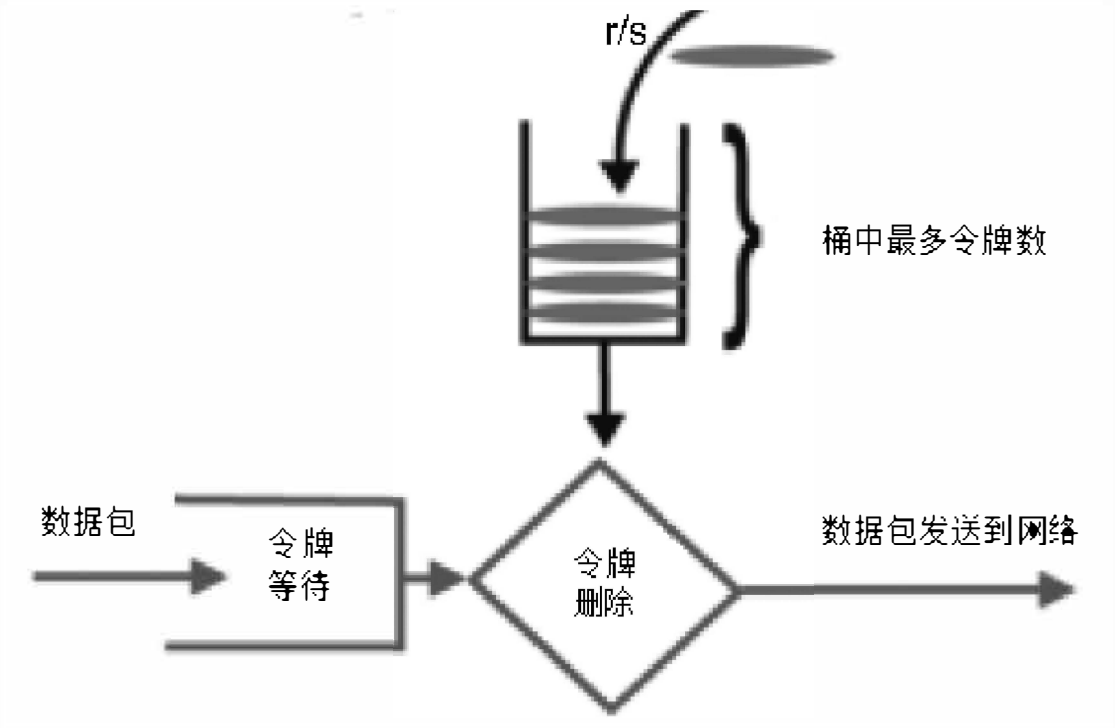
漏桶算法的思路很简单，水（请求）先进入漏桶里，漏桶以一定的速度出水（接口有响应速度），当水流入的速度过大时（访问频率超过接口响应速度）会直接溢出，然后就拒绝请求。如下图所示，可以看出漏桶算法能强行限制数据的传输速度。因为漏桶的漏出速度是固定的，所以，即使网络中不存在资源冲突（没有发生拥塞），漏桶算法也不能增大流量。因此，漏桶算法对于存在突发特性的流量来说缺乏效率。




**令牌桶算法**

令牌桶算法和漏桶算法效果相似，令牌桶算法更加容易理解。随着时间的流逝，系统会按恒定的1/QPS时间间隔（如果QPS=100，则间隔是10ms）往桶里加入令牌（就像有个水龙头在不断地

加水），如果桶已经满了就不再加了。新请求来临时，会拿走一个令牌，如果没有令牌可拿了，就阻塞或者拒绝请求，如下图所示。




令牌桶算法的另外一个好处是可以方便地改变速度。一旦需要提高速度，则按需提高放入桶中的令牌的速度即可。一般会定时（比如100ms）往桶中增加一定数量的令牌，有些变种算法则实时地计算应该增加的令牌的数量。

**限流实现**

Guava限流器RateLimiter是基于令牌桶算法实现的一个多线程限流器，它可以均匀地处理请求，当然它并不是一个分布式限流器，只是对单机进行限流。它可以定时拉取接口数据，下面是一个简单的限流功能的实例。

首选引入Maven依赖：

```
<!-- guava  -->
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>18.0</version>
</dependency>
```

然后使用Guava限流，Java代码实现如下：

```
/**
 * 以 1r/s 往桶中放入令牌
```

```
 */
private RateLimiter limiter = RateLimiter.create(1.0);
private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
@GetMapping("/indexLimiter")
public String indexLimiter() {
    //如果用户在 500ms 内没有获取到令牌，就直接放弃获取并进行服务降级处理
    boolean tryAcquire = limiter.tryAcquire(500， TimeUnit.MILLISECONDS);
    if (!tryAcquire) {
        log.info("Error ---时间:{}，获取令牌失败.", sdf.format(new Date()));
        return "系统繁忙，请稍后再试.";
    }
    log.info("Success ---时间:{}，获取令牌成功.", sdf.format(new Date()));
    return "success";
}
```

### 5.5.5 熔断保护

断路器（Circuit Breaker）就像保险丝，在电路系统中，一般在所有的家电系统连接外部供电的线路中间都会加一个保险丝，当外部电压过高，达到保险丝的熔点时，保险丝就会被熔断，从而可以切断家电系统与外部电路的联通，进而保障家电系统不会因为电压过高而损坏。

Hystrix提供的熔断器就有类似功能，在一定时间内调用的服务次数达到设定的阈值，并且出错的次数也达到设置的出错阈值时，就会进行服务熔断，让服务调用方执行本地设置的降级策略。但是Hystrix提供的熔断器具有自我反馈、自我恢复的功能，Hystrix会根据调用接口的情况，让熔断器在关闭（closed）、打开（open）、半打开（half-open）三种状态之间自动切换。

- closed→open：正常情况下熔断器为closed状态，当访问同一个接口的次数超过设定阈值并且错误比例超过设置的错误阈值时，就会打开熔断机制，这时候熔断器的状态从closed变为open。
- open→half-open：当服务接口对应的熔断器状态为open时，所有服务调用方调用该服务接口时都执行本地降级方法，Hystrix提供了一种测试策略，也就是设置了一个时间窗口，从熔断器的状态变为open开始的一个时间窗口内，调用该服务接口时都委托服务降级方法执行。如果时间超过了时间窗口，则熔断器的状态从open变为half-open，这个时候熔断器允许定量服务请求。
- half-open→closed：当熔断器的状态为half-open时，如果调用成功达到一定比例，则关闭熔断器，否则熔断器的状态再次变为closed。

### 5.5.6 超时与重试

在服务容错模式中，超时模式是最常见的容错模式。在实际开发中，有太多的故障是没有设置超时时间导致的服务“Hang住”或者OOM异常，或者是超时时间设置不合理导致的资源无法回收问题，最终导致系统崩溃。这些故障都是因为没有意识到超时设置的重要性而造成的。

**超时场景**

- 代理层超时与重试：Haproxy、Nginx、Twemproxy组件可实现代理功能，如Haproxy和Nginx可以实现请求的负载均衡，Twemproxy可以实现Redis的分片代理。我们需要设置代理与后端真实服务器之间的网络连接和读写超时时间。
- Web容器超时：Tomcat、Jetty等提供HTTP服务运行环境，我们需要设置客户端与容器之间的网络连接和读写超时时间，以及在此容器中默认的Socket网络连接和读写超时时间。
- 中间件客户端超时与重试：如消息中间件、CXF、Httpclient等，我们需要设置客户的网络连接和读写超时时间，以及失败重试机制。
- 数据库客户端超时：如MySQL、Oracle，需要分别设置JDBC Connection、Statement的网络连接和读写超时时间、事务超时时间、获取连接池连接等待时间。
- 业务超时：如超时订单取消任务、超时活动关闭，以及通过Future＃get（timeout，unit）限制某个接口的超时时间。
- 前端Ajax超时：浏览器通过Ajax访问网络时的网络连接和读写超时时间。

**重试机制**

重试是伴随着超时的，常见于因网络不稳定导致的服务调用超时场景。重试策略的参数设定一般需要与超时时间设置结合，要考虑接口的响应时间。立刻重试可能不是太好的策略，因为这样会导致在网络抖动的情况下对依赖服务的大量重试请求风暴；太长时间后再重试，会占用资源，失去重试机制的容错价值。在集群下，需要考虑对下游服务集群的同一个服务实例的重试次数与切换其他服务实例进行重试次数的比例，通常建议原有机器负载过高而响应延迟时，可以切换到集群中的其他服务实例，这样更快返回响应的概率会更大一点。

**幂等**

所谓幂等就是多次执行操作所产生的影响与一次执行的影响相同。在允许重试的场景中，我们需要保证服务提供方能够实现业务逻辑的幂等，因为重试机制可能导致服务提供方被多次调用。幂等设计需要解决的是“写重试”的问题。

## 5.5.7 Spring Cloud Hystrix容错框架

Hystrix中文名称为“豪猪”，平时性情温顺，在感受到危险时，用浑身长满的刺来保护自己。Hystrix的整体设计原则是防止单个服务的故障（网络、资源耗尽）等原因产生的分布式下的级联失败，通过快速失败代替队列实现优雅的服务降级，当依赖服务恢复正常后，可快速恢复服务正常运行状态，同时Hystrix提供实时的监控、报警及DashBoard等功能。

Hystrix同样是Netflix开源的一个分布式系统容错框架。在分布式网络环境下，不可避免地出现服务之间因为网络超时、代码异常等原因产生各种各样的调用失效问题，Hystrix通过延迟容忍和错误容忍逻辑可以控制分布式系统之间的交互，在失败调用超过预先设置的阈值时，会自动隔离服务访问、抑制级联错误、支持fallback降级处理等，保证系统的可用性和弹性。

Spring Cloud将Hystrix的容错组件进行了自动化配置，在Spring Cloud微服务架构中可以通过注解机制实现Hystrix与不同组件模块的联合使用，实现请求调用的容错处理。

**Hystrix的主要特性**

- 断路器机制：Hystrix的断路器工作机制非常简单。当Hystrix Command请求后端服务失败数量超过一定比例时（默认为50%），断路器会切换到开路状态（Open），这时所有请求会直接失败而不会发送到后端服务，同时断路器有自我检测并恢复的功能。
- 资源隔离：在Hystrix中，主要通过线程池来实现资源隔离。通常在使用的时候，我们会根据调用的远程服务划分出多个线程池。如果对性能有严格要求而且确信自己调用服务的客户端代码不会出问题，可以使用Hystrix的信号模式（Semaphores）来隔离资源。
- 优雅降级：fallback相当于降级操作。对于查询操作，我们可以实现一个fallback方法，当请求后端服务出现异常时，可以使用fallback方法提供返回值。fallback方法的返回值一般是默认值或者来自缓存的值。
- 请求缓存：对于无差异的后端服务请求，我们通常会把第一次请求缓存，对于后面的请求，我们会直接返回第一次请求的缓存作为响应。

**Hystrix通常的使用方法**

1.使用Hystrix-core的HystrixCommand源码方式

首先，增加Maven配置依赖：

```
<dependencies>
   <dependency>
       <groupId>com.netflix.hystrix</groupId>
       <artifactId>hystrix-core</artifactId>
       <version>1.5.13</version>
   </dependency>
</dependencies>
```

其次，编写业务Command逻辑实现（返回“success”），它继承了HystrixCommand：

```
public class HelloCommand extends HystrixCommand<String> {
   protected HelloCommand() {
       super(HystrixCommandGroupKey.Factory.asKey("test"));
    }
   @Override
   protected String run() throws Exception {
       //模拟请求外部接口需要的时间
       Thread.sleep(500);
       return "success";
    }
    @Override
    protected String getFallback() {
    //当外部请求超时后，会执行 fallback 方法中的逻辑
    System.out.println("执行了回退方法");
    return "error";
    }
}
```

最后，调用Hystrix命令类：

```
package com.ivan.client.hystrix;
public class App {
      public static void main(String[] args) {
        HelloCommand command = new HelloCommand();
        String result = command.execute();
        System.out.println(result);
    }
}
```

2.使用注解@HystrixCommand方式实现熔断及降级管理

首先，加入Maven依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

其次，开启Hystrix断路器，使用@EnableHystrix开启Spring Boot对Hystrix的支持：

```
@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
@EnableHystrix
public class CloudServiceRibbonApplication {
    public static void main(String[] args) {
        SpringApplication.run(CloudServiceRibbonApplication.class,  args);
    }
    @Bean
    @LoadBalanced
    RestTemplate RestTemplate() {
        return new RestTemplate();
    }
}
```

然后，增加@HystrixCommand注解和fallback方法降级实现。该注解对该方法创建了熔断器的功能，并指定了fallbackMethod熔断方法，熔断方法直接返回一个字符串，字符串为"hi，"+name+"，sorry，error！"。

```
@Service
public class TestService {
    @Autowired
    RestTemplate RestTemplate;
    @HystrixCommand(fallbackMethod = "hiError")
    public String hiService(String name) {
        return RestTemplate.getForObject("http://CLOUD-EUREKA-CLIENT/
            hi?name="+name，String.class);
    }
    public String hiError(String name) {
        return "hi，"+name+"，sorry，error!";
    }
}
```

3.在Feign中整合Hystrix

首先，开启Hystrix配置：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/server:
  port: 8765spring:
  application:
    name: cloud-service-feign
    feign.hystrix.enabled: true
```

其次，使用@EnableFeignClients启动并加载应用：

```
@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
@EnableFeignClients
public class CloudServiceFeginApplication {
    public static void main(String[] args) {
        SpringApplication.run(CloudServiceFeginApplication.class， args);
    }
}
```

然后，通过在FeignClient中增加fallback属性，配置连接失败错误的处理类：

```
@FeignClient(value = "cloud-eureka-client"，fallback = TestServiceHystric.class)
public interface TestService {
    @RequestMapping(value = "/hi"，method = RequestMethod.GET)
    String sayHiFromClientOne(@RequestParam(value = "name") String name);
}
```

最后，创建fallback方法，当访问接口有问题或发生错误时，直接调用此接口：

```
@Component
public class TestServiceHystric implements  TestService{
    @Override
    public String sayHiFromClientOne(String name) {
        return "sorry "+name;
    }
}
```

4.在Zuul网关中使用fallback功能实现熔断降级

Zuul默认提供了对Hystrix的支持，在@EnableZuulProxy注解中，实现代码如下：

```
@EnableCircuitBreaker
@EnableDiscoveryClient
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Import(ZuulProxyMarkerConfiguration.class)
public @interface EnableZuulProxy {
}
```

在Zuul的HTTP请求源码中，Zuul请求转发会被Hystrix包装成一个Command发送给后端服务，Zuul默认提供了如下请求转发方式：

```
@Configuration
@ConditionalOnRibbonHttpClient
protected static class HttpClientRibbonConfiguration {
    @Autowired(required = false)
    private Set<ZuulFallbackProvider> ZuulFallbackProviders = Collections.emptySet();
```

```
    @Bean
    @ConditionalOnMissingBean
    public RibbonCommandFactory<?> ribbonCommandFactory(
    SpringClientFactory clientFactory， ZuulProperties ZuulProperties)
    {
        return new HttpClientRibbonCommandFactory(
            clientFactory，ZuulProperties， ZuulFallbackProviders
        );
    }
}
```

通过源码可以知道，Zuul将FallbackProvider实例保存在一个Set集合中，并作为HttpClient-RibbonCommandFactory构造器的参数。Zuul在转发请求时最终会利用AbstractRibbonCommand进行处理。AbstractRibbonCommand继承了HystrixCommand，而转发请求的业务逻辑在重写的HystrixCommand类的run方法中，源码如下：

```
@Override
protected ClientHttpResponse run() throws Exception {
    final RequestContext context = RequestContext.getCurrentContext();
    RQ request = createRequest();
    RS response = this.client.executeWithLoadBalancer(request， config);
    context.set("ribbonResponse"， response);
    if (this.isResponseTimedOut()) {
        if (response != null) {
            response.close();
        }
    }
    return new RibbonHttpResponse(response);
}
```

我们知道HystrixCommand提供了getFallback方法，这个方法的作用是当run方法出现异常时自动调用getFallback方法，从而完成降级。由于AbstractRibbonCommand继承了HystrixCommand，它不仅重写了run方法，而且重写了getFallback方法，源码如下：

```
@Override
protected ClientHttpResponse getFallback() {
    if(ZuulFallbackProvider != null) {
        return ZuulFallbackProvider.fallbackResponse();
    }
    return super.getFallback();
}
```

下面是自定义的ZuulFallbackProvider实现，如果有相应的实现，那么它会直接回调自定义实现类的fallbackResponse方法。

```
@Component
public class CustomizeFallbackProvider implements FallbackProvider {
    private static Logger logger = LoggerFactory.getLogger(CustomizeFallbackProvider .class);
    private String defultRout = "*";
    @Override
    public String getRoute() {
        return defultRout;//所有路由
    }
    @Override
    public ClientHttpResponse fallbackResponse(Throwable cause) {
        StackTraceElement[] elm = cause.getStackTrace();
        StringBuffer stackTrace = new StringBuffer();
        for (StackTraceElement e : elm) {
            stackTrace.append(e.toString());
            stackTrace.append("\n");
        }
        return null;
    }
    @Override
    public ClientHttpResponse fallbackResponse() {
        return null;
    }
}
```

Zuul配置Hystrix时需要注意以下配置信息：

```
hystrix:
  threadpool:
    default:
      ##并发执行的最大线程数，默认为 10
      coreSize: 100
      ##BlockingQueue 的最大队列数
      maxQueueSize: 150
      queueSizeRejectionThreshold: 150
  command:
    default:
      execution:
        timeout:
          enabled: true
        isolation:
          strategy: THREAD
          semaphore:
            maxConcurrentRequests: 3000
          thread:
            timeoutInMilliseconds: 130000
      fallback.isolation.semaphore.maxConcurrentRequests: 300
        #enabled: false
        #forceOpen: false
```

使用Zuul集成Hystrix需要注意：

- Zuul的超时时间要大于等于Hystrix的超时时间，避免Hystrix设置无法生效。
- 在无重试场景下，通过Feign+Ribbon方式进行服务调用时，Hystrix的超时时间要小于Ribbon的超时时间，否则在Ribbon调用其他服务时就已经超时了，Hystrix无法进行熔断及降级。
- 在有重试场景下，如果有组件跟Hystrix配合使用，一般来讲，建议Hystrix的超时时间大于其他组件的超时时间，否则将可能导致重试失效。例如，如果Ribbon的超时时间为1s，重试3次，Hystrix的超时时间应略大于3s。

**Hystrix的服务降级模式**

Hystrix通过注解@HystrixCommand可以降低代码的侵入性，下面我们把Hystrix中Command与fallback的常用模式总结一下。

1.同步Command，同步fallback

```
@HystrixCommand(fallbackMethod = "getDefaultValue")
@PostMapping("/toCart/{productId}")
public ResponseEntity addCart(@PathVariable("productId") Long productId) throws
    InterruptedException {
    Long aLong = cartFeignClient.addCart(productId);
    System.out.println(aLong);
    return ResponseEntity.ok(productId);
}
private ResponseEntity getDefaultValue(Long productId) {
    return ResponseEntity.ok(0);
}
```

2.异步Command，同步fallback

```
@HystrixCommand(fallbackMethod = "getDefaultAsyncAddCart")
public Future<ResponseEntity<Long>> asyncAddCart(@PathVariable("productId") Long
    productId) throws InterruptedException {
    log.info("异步 command：run。。。");
    Thread.sleep(5000);    //触发降级逻辑
    return new AsyncResult<ResponseEntity<Long>>() {
        @Override
        public ResponseEntity<Long> invoke() {
            return ResponseEntity.ok(cartFeignClient.addCart(productId));
        }
    };
}
private ResponseEntity<Long> getDefaultAsyncAddCart(Long productId， Throwable throwable) {
    log.info("异步 command，同步 fallback：run。。。");
    return ResponseEntity.ok(0L);
}
```

3.异步Command，异步fallback

```
@HystrixCommand(fallbackMethod = "getDefaultAsyncAddCart2")
public Future<ResponseEntity<Long>> asyncAddCart2(@PathVariable("productId") Long
    productId) throws InterruptedException {
    log.info("异步 command：run。。。");
    Thread.sleep(5000);
    return new AsyncResult<ResponseEntity<Long>>() {
        @Override
        public ResponseEntity<Long> invoke() {
            return ResponseEntity.ok(cartFeignClient.addCart(productId));
        }
    };
}
```

**注意：** 异步fallbackMethod这里必须加@HystrixCommand注解，否则运行时会报错：

```
@HystrixCommand
private Future<ResponseEntity<Long>> getDefaultAsyncAddCart2(Long productId,
    Throwable throwable) {
    log.info("异步 command，同步 fallback：run。。。");
    log.warn("", throwable);
    return new AsyncResult<ResponseEntity<Long>>() {
        @Override
        public ResponseEntity<Long> invoke() {
            return ResponseEntity.ok(0L);
        }
    };
}
```

## 5.5.8 Hystrix的核心工作原理

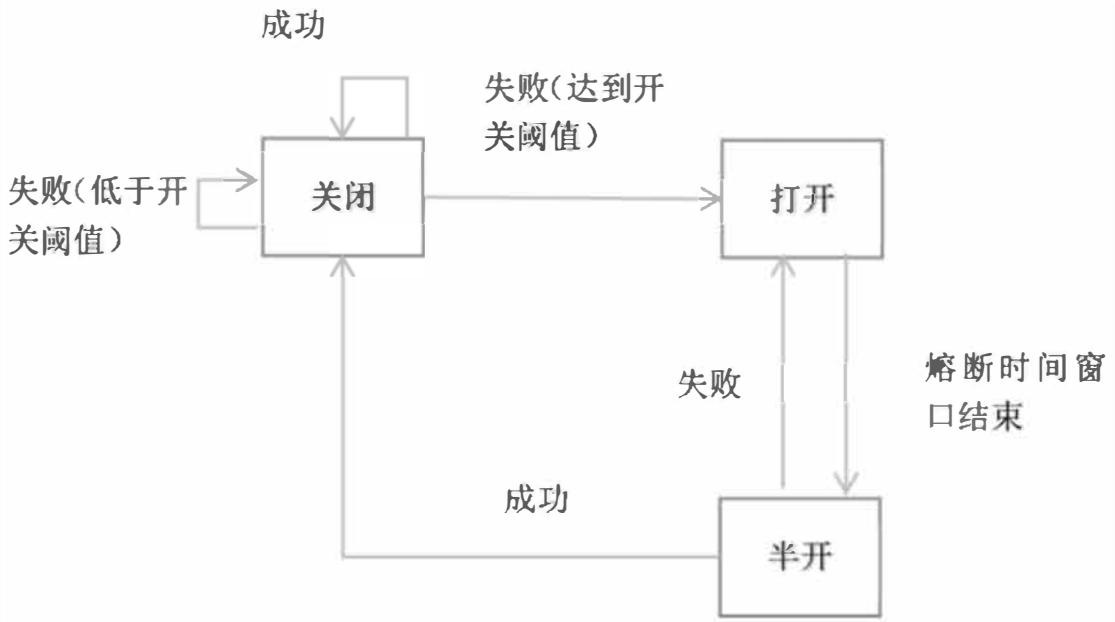
Hystrix的本质作用是当系统资源过载（Over Load Control）时提供服务状态保护机制，包括下面四个方面。

- 熔断：当失败率达到阈值时自动触发降级（如因网络故障或超时造成的失败率高），熔断器触发的快速失败会进行快速恢复。
- 隔离（线程池隔离和信号量隔离）：限制调用分布式服务的资源使用，某一个调用的服务出现问题不会影响其他调用。
- 降级：超时降级、资源不足时（线程或信号量）降级，降级后可以配合降级接口返回托底数据，做到优雅降级。
- 缓存：提供了请求缓存、请求合并的实现方法。

**Hystrix——熔断**

熔断器的原理很简单，可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定允许操作继续或者立即返回错误。熔断器开关相互转换的逻辑如下图所示。



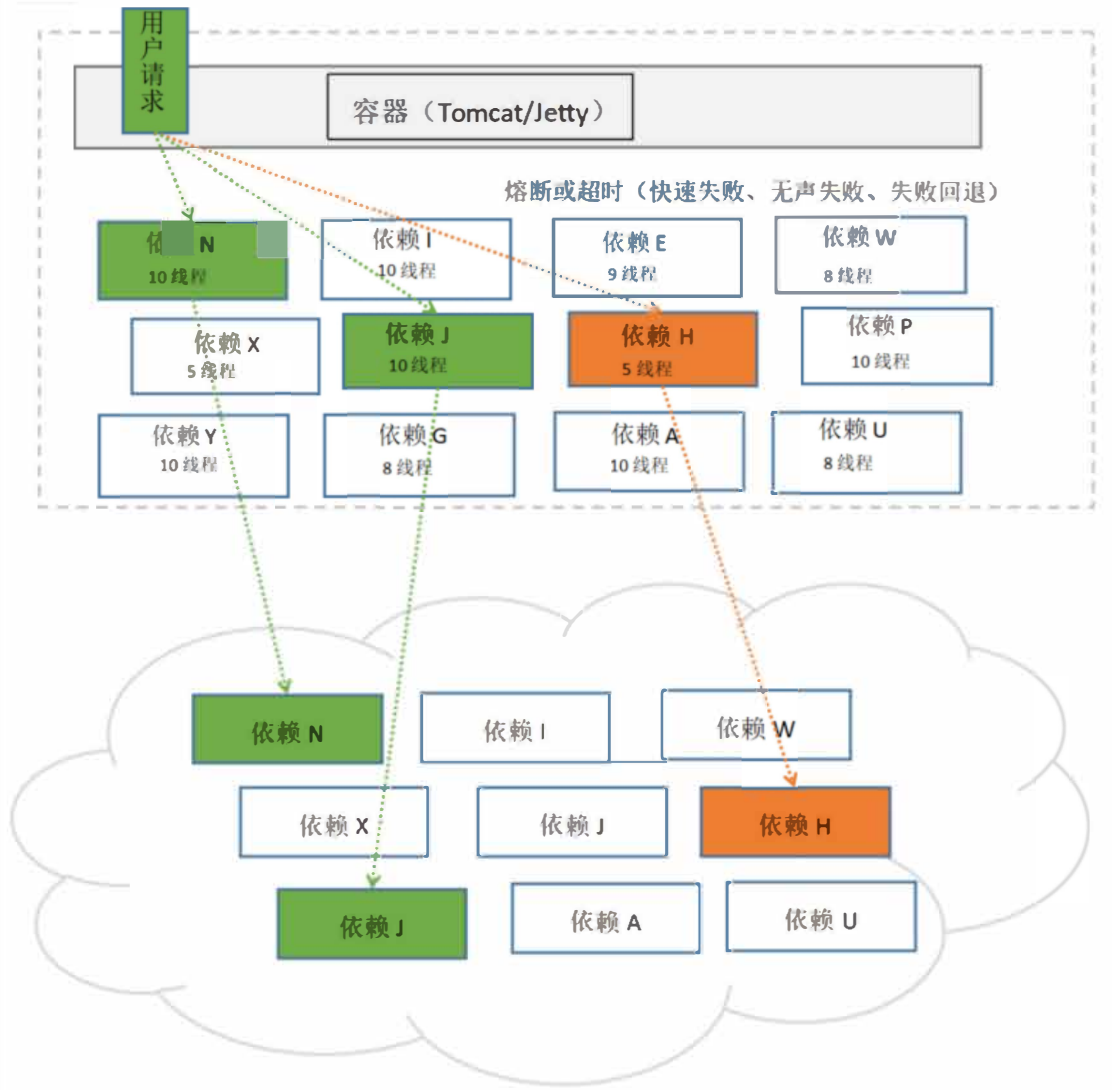

Hystrix中的熔断器（Circuit Breaker）也起到这样的作用，Hystrix在运行过程中会向每个CommandKey对应的熔断器报告成功、失败、超时和拒绝的状态，熔断器维护并计算统计的数据，根据这些统计信息来确定是否打开。如果打开，后续的请求都会被截断（不再执行run方法里的内容，直接执行fallback方法里的内容）。然后隔一段时间（默认是5s），尝试半开，放一部分请求进来，相当于对依赖服务进行一次健康检查，如果服务没问题，熔断器关闭，随后完全恢复调用。

**Hystrix——隔离**

对于微服务系统来说，一个从客户端发来的HTTP请求往往途径众多微服务，在处于高流量状态下，如果其中一个服务器资源出现饱和状态，就会影响上游系统。如下图所示，Hystrix可以将服务调用包裹在HystrixCommand中，每一个HystrixCommand都维护着一个线程池，从而隔离服务，当一个服务产生延迟时，其“吞噬”的资源也只会限定在该HystrixCommand内（比如至多只会占用 N 个线程资源），而不会对全局造成影响。

Hystrix的隔离主要是为每个依赖组件提供一个隔离的线程环境，有两种隔离模式。

- 线程池隔离模式：使用一个线程池来存储当前的请求。线程池对请求做处理，设置任务返回处理超时时间，堆积的请求堆积入线程池队列。这种方式需要为每个依赖的服务申请线程池，有一定的资源消耗，好处是可以应对突发流量（流量洪峰来临时，处理不完可将数据存储到线程池里慢慢处理）。
- 信号量隔离模式：使用一个原子计数器（或信号量）来记录当前有多少个线程在运行，请求到来时先判断计数器的数值，若超过设置的最大线程个数，则丢弃该类型的新请求，若不超过，则执行计数操作，请求到来计数器+1，请求返回计数器–1。这种方式是严格的控制线程且立即返回模式，无法应对突发流量（流量洪峰来临时，处理的线程超过数量，其他的请求会直接返回，不继续去请求依赖的服务）。




两种隔离模式的主要区别如下表所示。

| 比较项 | 线程池隔离模式 | 信号量隔离模式 |
| --- | --- | --- |
| 线程 | 与调用线程不相同的线程 | 与调用线程相同 |
| 开销 | 排队、调度、上下文开销等 | 无线程切换，开销低 |
| 异步 | 支持 | 不支持 |
| 并发支持 | 支持（最大线程池大小） | 支持（最大信号量上限） |

Hystrix隔离策略相关的参数如下。

- execution.isolation.strategy=THREAD|SEMAPHORE：设置线程或信号量隔离模式。
- hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：设置隔离模式的超时时间，默认值是1000ms。
- execution.isolation.semaphore.maxConcurrentRequests：设置在使用时允许到HystrixCommand.run方法的最大请求数，默认值是10。
- execution.timeout.enabled：设置HystrixCommand.run方法执行时是否开启超时设置，默认开启。
- execution.isolation.thread.interruptOnTimeout：发生超时时是否中断HystrixCommand.run方法，默认是true。

- execution.isolation.thread.interruptOnCancel：取消时是否中断HystrixCommand.run方法，默认是false。

**Hystrix——降级**

所谓降级，就是指在Hystrix执行非核心链路功能失败的情况下，我们如何处理，比如我们返回默认值等。如果我们要回退或者降级处理，代码上需要实现HystrixCommand.getFallback方法或者HystrixObservableCommand方法。

Netflix的Hystrix对微服务降级处理实现提供两种方式：

- 通过添加注解@HystrixCommand方式来实现。
- 通过继承HystrixCommand类来实现。

1.使用@HystrixCommand注解实现服务降级

使用注解可以最小限度地侵入代码，可以快速让原来的功能支持服务降级，使用时仅需在要进行服务降级处理的方法上增加@HystrixCommand注解即可，并通过fallbackMethod属性设置在降级处理时所使用的方法，然后在降级方法中实现服务降级处理。需要注意：通过fallbackMethod属性所指定的方法要与原方法具有相同的方法签名，否则降级会失败。下面是@HystrixCommand的参数说明。

- groupKey：设置HystrixCommand分组的名称。
- commandKey：设置HystrixCommand的名称。
- threadPollKey：设置HystrixCommand执行线程池的名称。
- fallbackMethod：设置HystrixCommand服务降级所使用的方法名称，注意该方法需要与原方法定义在同一个类中，并且方法签名也要一致。
- commandProperties：设置HystrixCommand属性，例如断路器失败百分比、断路器时间容器大小等。
- ignoreException：设置HystrixCommand执行服务降级处理时需要忽略的异常，当出现异常时不会执行服务降级处理。
- observableExecutionMode：设置HystrixCommand执行的方式。
- defaultFallback：设置HystrixCommand默认的服务降级处理方法，如果同时设定fallbackMethod属性，会优先使用fallbackMethod属性所指定的方法。该属性所指定的方法没有参数，需要注意返回值与原方法返回值的兼容性。

2.继承HystrixCommand类实现服务降级

除使用注解方式来完成服务降级实现外，Hystrix还提供了两个对象来支持服务降级实现处理：HystrixCommand 和 HystrixObserableCommand 。 如 果 继 承 HystrixCommand 则 需 要 实 现 getFallback方法，代码如下：

```
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
public class HystrixFallback extends HystrixCommand<String> {
    private String name;
    public HystrixFallback() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("test")));
    }
    public HystrixFallback(String name) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("test")));
               this.name = name;
    }
    @Override
    protected String run() throws Exception {
        //实现具体的业务处理逻辑
        return null;
    }
    @Override
    protected String getFallback() {
        //实现服务降级处理逻辑
        return super.getFallback();
  }
}
```

HystrixObserableCommand用于所依赖服务返回多个操作结果的时候，在实现服务降级时，如果是继承HystrixObserableCommand，则需要实现resumeWithFallback方法，代码如下。

```
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixObservableCommand;
import rx.Observable;
public class HystrixObervableFallback extends HystrixObservableCommand<String> {
    private String name;
    public HystrixObervableFallback() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("test")));
    }
    public HystrixObervableFallback(String name) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("test")));
        this.name = name;
    }
    @Override
    protected Observable<String> construct() {
        //实现具体的业务处理逻辑
        return null;
    }
    @Override
    protected Observable<String> resumeWithFallback() {
        //实现服务降级处理逻辑
        return super.resumeWithFallback();
    }
}
```

### Hystrix——缓存

Hystrix有两种方式来应对高并发场景，分别是请求缓存与请求合并缓存。请求缓存是在同一请求多次访问中保证只调用一次这个服务提供者的接口，同一请求第一次的结果会被缓存，保证同一请求多次访问返回结果相同。

Hystrix的缓存实现方式主要有两种：继承方式和注解方式，用得比较多的方式是注解方式，因为注解方式开发快而且相对简单，如下表所示。

| 注解 | 描述 | 属性 |
| --- | --- | --- |
| @CacheResult | 该注解用来标记请求命令返回的结果应该被缓存，它必须与@HystrixCommand注解结合使用 | cacheKeyMethod |

续表

| 注解 | 描述 | 属性 |
| --- | --- | --- |
| @CacheRemove | 该注解用来让请求命令的缓存失效，失效的缓存根据commandKey进行查找 | commandKey，cacheKeyMethod |
| @CacheKey | 该注解用来在请求命令的参数上做标记，使其作为cacheKey，如果没有使用此注解，则会使用所有参数列表中的参数作为cacheKey | value |

## 1.使用@CacheResult开启请求缓存功能

```
@Service
public class CacheResultDemo {
    @Autowired
    private RestTemplate RestTemplate;
    @CacheResult(cacheKeyMethod = "getUserId")
    @HystrixCommand(fallbackMethod = "hiConsumerFallBack")
    public User hiConsumer(String id) {
        //SERVICE_HI 是服务端的，hi 为服务端提供的接口
        return RestTemplate.getForEntity("http://SERVICE_HI/hi", ser.class).getBody();
    }
    public String hiConsumerFallBack(String id, Throwable e) {
        return "This is a error";
    }
    public String getUserId(String id) {
        return id;
    }
}
```

## 2.使用CacheKey开启缓存

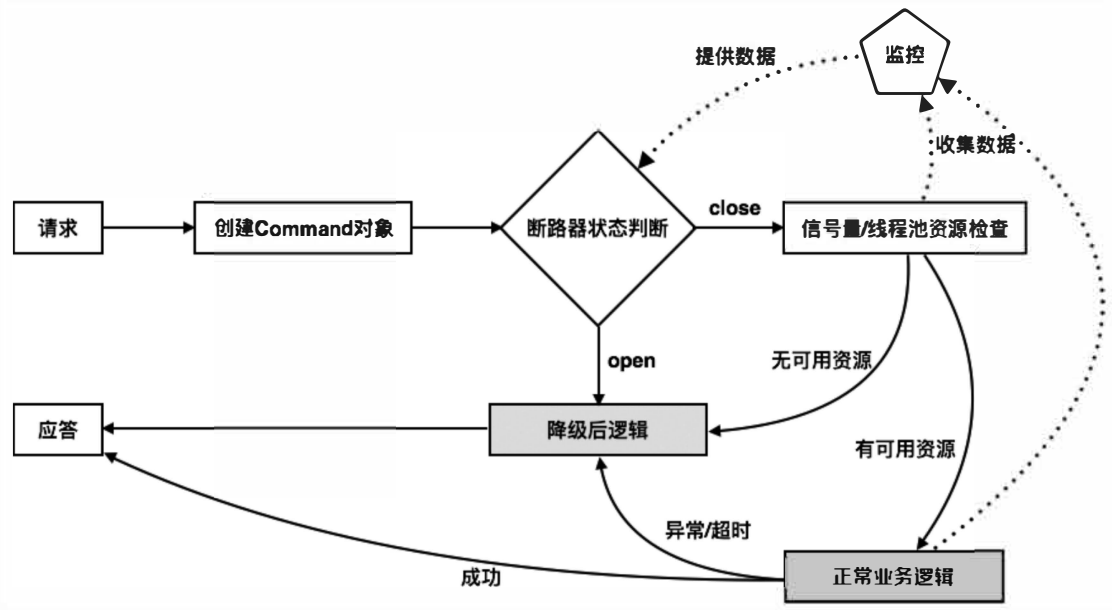
```
@Service
public class CacheKeyDemo {
    @Autowired
    private RestTemplate RestTemplate;
    @HystrixCommand(fallbackMethod = "hiConsumerFallBack")
    public User hiConsumer(@CacheKey("id") String id)
            return RestTemplate.getForEntity("http://SERVICE_HI/hi", User.class).getBody();
      }
      public String hiConsumerFallBack(String id, Throwable e) {
            return "This is a error";
     }
}
```

## 3.通过@CacheRemove注解来实现失效缓存清理功能

```
@Service
public class CacheRemoveDemo {
      @Autowired
      private RestTemplate RestTemplate;
      @CacheRemove(commandKey = "getUserId")
      @HystrixCommand(fallbackMethod = "hiConsumerFallBack")
      public void update(@CacheKey("id") User user) {
            RestTemplate.postForObject("http://SERVICE_HI/hi", user, User.class);
      return;
      }
      public String hiConsumerFallBack(String id, Throwable e) {
            return "This is a error";
      }
      public String getUserId(String id) {
            return id;
      }
}
```

**Hystrix的工作流程**

Hystrix使用RxJava作为响应式的编程框架。这里我们简单介绍一下Hystrix的工作流程，一个简化版本的Hystrix执行流程如下图所示。




首先，构造一个HystrixCommand或HystrixObservableCommand对象。

- 如果期望依赖项返回单个响应，则构造一个HystrixCommand对象，代码如下：

```
HystrixCommand command =new HystrixCommand(arg1， arg2);
```

- 如果期望依赖项返回发出响应的可观察对象，则构造一个HystrixObservableComman对象，代码如下：

```
HystrixObservableCommand command = new HystrixObservableCommand(arg1， arg2);
```

有4种方法可以执行Hystrix命令（前两种方法只适用于简单的HystrixCommand对象，不适用于HystrixObservableCommand对象）。

- execute：该方法与queue方法以相同的方式获取一个Future对象，然后在这个Future上调用get方法来获取可观察对象发出的单个值。
- queue：该方法将可观察对象转换为BlockingObservable对象，以便将其转换为Future对象，然后返回此Future对象。
- observe：该方法可以立即订阅可观察对象，并开始执行命令的流。返回一个可观察对象，当订阅该对象时，它将重新产生结果并通知订阅者。
- toObservable：该方法返回的可观察值不变，需要订阅后才能真正开始执行命令流程。

下面是Hystrix的具体执行逻辑。

1.构造Hystrix命令

构造一个HystrixCommand或HystrixObservableCommand对象，用于封装请求并在构造方法中配置请求被执行需要的参数。

2.执行Hystrix命令

根据上文中提供的4种方式执行命令。

3.判断是否缓存了响应

如果你为命令启用了请求缓存，并且在缓存中命中了可用请求的响应，则缓存的响应将立即以可观察到的形式返回。

4.判断熔断电路是否打开

当执行命令时，Hystrix将与断路器一起检查熔断电路是否打开。如果熔断电路打开，那么Hystrix将不执行命令并回退。如果熔断电路关闭，则继续执行，检查是否有可用的容量来运行命令。

5.线程池、队列、信号量是否已满

如果与命令关联的线程池和队列（或信号量，如果不在线程中运行）已满，那么Hystrix将不执行命令，执行逻辑跳转到第7步。

6.计算电路健康状态

执行HystrixObservableCommand.construct或HystrixCommand.run方法，Hystrix向断路器报告成功、失败、拒绝或超时，如果执行逻辑失败或者超，则执行逻辑跳转第7步；否则执行逻辑跳转到第8步；

7.回退

Hystrix试图恢复你的回滚命令，并执行回退逻辑或者fallback备用逻辑。

8.返回成功的响应

如果Hystrix命令成功，它将以可观察到的形式返回响应给调用者。

## 5.5.9 Hystrix源码解析

@HystrixCommand注解是由名为javanica的Hystrix contrib库提供的。javanica是一个Hystrix的子项目，用于简化Hystrix的使用。我们还需要添加一个依赖：

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-javanica</artifactId>
    <version>RELEASE</version>
</dependency>
```

查看hystrix-javanica的readme及注解@HystrixCommand的引用可以发现，HystrixCommand-Aspect实现了对@HystrixCommand注解的拦截和处理。下面是Aspect拦截部分的实现源码：

```
@Aspect
public class HystrixCommandAspect {
    private static final Map<HystrixPointcutType, MetaHolderFactory>
        META_HOLDER_FACTORY_MAP;
    static {
        META_HOLDER_FACTORY_MAP = ImmutableMap.<HystrixPointcutType,
        MetaHolderFactory>builder().put(HystrixPointcutType.COMMAND,
        new CommandMetaHolderFactory())
                .put(HystrixPointcutType.COLLAPSER,
        new CollapserMetaHolderFactory()) .build();
    }
   @Pointcut("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand)")
    public void hystrixCommandAnnotationPointcut() {
    }
    @Pointcut("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCollapser)")
    public void hystrixCollapserAnnotationPointcut() {
    }
    @Around("hystrixCommandAnnotationPointcut() || hystrixCollapserAnnotationPointcut()")
    public Object methodsAnnotatedWithHystrixCommand(
        final ProceedingJoinPoint joinPoint) throws Throwable {
//省略
}
```

HystrixCommand继承了AbstractCommand类，它提供了更多的构造函数和构造参数建造者，最终调用run方法实现用户业务逻辑，而AbstractCommand类就要实现隔离、熔断等核心功能。如果查看AbstractCommand类的代码，你会发现这个类主要使用了RxJava响应式框架。下面我们对AbstractCommand类的熔断过程加以讲解。

AbstractCommand类的initCircuitBreaker方法是熔断器的构造方法入口，它判断是否打开了熔断器，只有在打开了熔断器后才会通过HystrixCircuitBreaker.Factory工厂新建一个熔断器，源码如下：

```
private static HystrixCircuitBreaker initCircuitBreaker(boolean enabled, HystrixCircuitBreaker
       fromConstructor, HystrixCommandGroupKey groupKey,
       HystrixCommandKey commandKey, HystrixCommandProperties properties,
       HystrixCommandMetrics metrics) {
       if (enabled) {
           if (fromConstructor == null) {
               return HystrixCircuitBreaker.Factory.getInstance(commandKey, groupKey,
                   properties, metrics);
           } else {
               return fromConstructor;
           }
       } else {
           return new NoOpCircuitBreaker();
       }
   }
```

HystrixCircuitBreaker.Factory类根据CommandKey对熔断器进行了缓存，如果缓存里存在Key，则直接取缓存里的Key，如果不存在，则新建HystrixCircuitBreakerImpl对象，用于熔断操

作，源码如下：

```
class Factory {
//circuitBreakersByCommand 是一个 ConcurrentHashMap，这里缓存了系统的所有熔断器
  private static ConcurrentHashMap<String， HystrixCircuitBreaker>
  circuitBreakersByCommand = new ConcurrentHashMap<String， HystrixCircuitBreaker>();
  public static HystrixCircuitBreaker getInstance(
           HystrixCommandKey Key，
           HystrixCommandGroupKey group，
           HystrixCommandProperties properties，
           HystrixCommandMetrics metrics) {
       HystrixCircuitBreaker previouslyCached = circuitBreakersByCommand.get(Key.name());
       if (previouslyCached != null) {
           return previouslyCached;
       }
       HystrixCircuitBreaker cbForCommand =
           circuitBreakersByCommand.putIfAbsent(Key.name()，
           new HystrixCircuitBreakerImpl(Key， group， properties， metrics));
       if (cbForCommand == null) {
           return circuitBreakersByCommand.get(Key.name());
       } else {
           return cbForCommand;
       }
    }
    public static HystrixCircuitBreaker getInstance(HystrixCommandKey Key) {
       return circuitBreakersByCommand.get(Key.name());
    }
    /**
     * Clears all circuit breakers. If new requests come in instances will be recreated.
     */
    /* package */static void reset() {
       circuitBreakersByCommand.clear();
    }
}
```

HystrixCircuitBreakerImpl类里定义了一个状态变量，分别为关闭、打开、半开状态。我们重点关注allowRequest方法，在allowRequest方法里首先判断forceOpen属性是否打开，如果打开则不允许有请求进入。然后判断forceClosed属性，如果这个属性为true，刚对所有的请求放行，相当于熔断器不起作用。isAfterSleepWindow方法用于放行超过了指定时间后的流量。下面是主要的代码实现：

```
class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {
    private final HystrixCommandProperties properties;
    private final HystrixCommandMetrics metrics;
    enum Status {
        CLOSED， OPEN， HALF_OPEN;//三种状态通过枚举来定义
    }//默认是关闭的状态
    private final AtomicReference<Status> status = new
        AtomicReference<Status>(Status.CLOSED);
    private final AtomicLong circuitOpened = new AtomicLong(-1);
    private final AtomicReference<Subscription> activeSubscription = new
        AtomicReference<Subscription>(null);
    protected HystrixCircuitBreakerImpl(HystrixCommandKey Key，
        HystrixCommandGroupKey commandGroup，
        final HystrixCommandProperties properties，
```

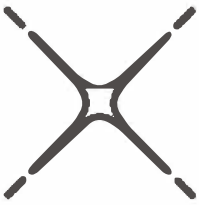
```
            HystrixCommandMetrics metrics) {
            this.properties = properties;
            this.metrics = metrics;
            Subscription s = subscribeToStream();
            activeSubscription.set(s);
        }
        //用于判断熔断器是否打开
        @Override
        public boolean isOpen() {
            if (properties.circuitBreakerForceOpen().get()) {
                return true;
            }
            if (properties.circuitBreakerForceClosed().get()) {
                return false;
            }
            return circuitOpened.get() >= 0;
        }
        @Override
        public boolean allowRequest() {//用于判断是否放行流量
            if (properties.circuitBreakerForceOpen().get()) {
                return false;
            }
            if (properties.circuitBreakerForceClosed().get()) {
                return true;
            }//第一次请求肯定就放行了
            if (circuitOpened.get() == -1) {
                return true;
            } else {//半开状态将不放行
                if (status.get().equals(Status.HALF_OPEN)) {
                    return false;
                } else {
                    return isAfterSleepWindow();
                }
            }
        }
    }
```

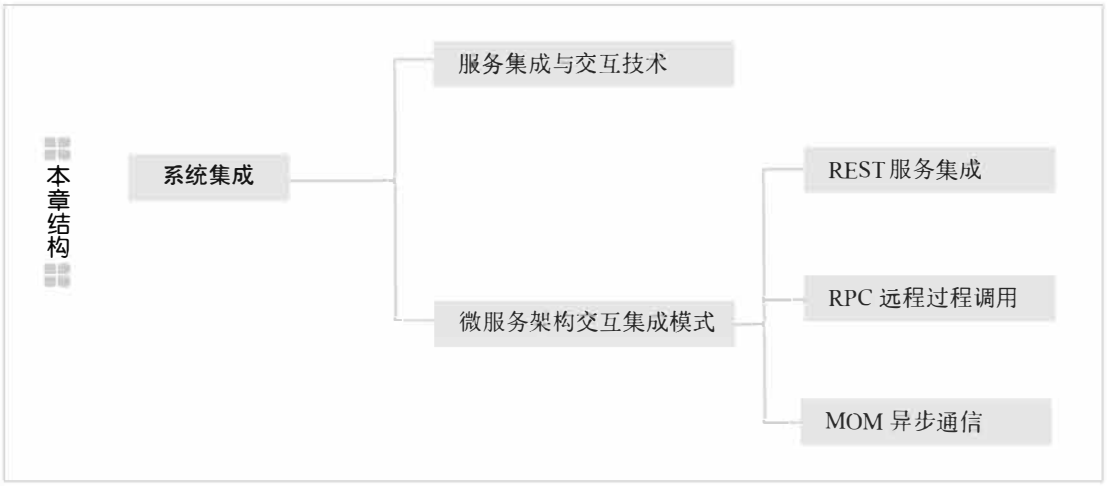
## 5.6 小结

微服务架构的关键技术是为微服务提供高可用、可扩展性、可容错性的平台基础设施。Spring Cloud作为微服务的治理平台为我们提供了相关的技术和服务组件，而开源的解决方案往往无法做到满足个性需求，我们需要根据实际场景和业务特性做定制化开发。定制化开发的前提是我们能够充分理解这些关键技术的底层运行原理和工作机制。

第6章

# 系统集成






系统集成是相对拆分而言的，当巨石型应用拆分为细粒度的微服务后，错综复杂的代码可以分解为独立的模块加以治理。然而，传统应用内部原本基于方法的调用方式可能会转变为跨进程的分布式网络调用方式，网络的不可靠性给服务模块之间的交互带来了复杂性。所以，微服务系统的集成对微服务架构能否成功落地至关重要。

微服务架构强调基于HTTP的轻量级的服务交互模式，这一章我们将这种基于请求/响应模式的交互模式与RESTful架构结合，介绍微服务“声明式API”和契约优先的开发原则。同时我们会深入讲解主流RPC架构的实现原理和RPC通信方式的优势和缺点。

微服务架构的另外一种集成模式基于消息中间件的异步交互方式。这种交互模式无疑带给了微服务更多的灵活性和自治性，但也带来了复杂性，我们需要在使用场景中做出权衡，选择适合自己的消息中间件。

## 6.1 服务集成交互技术

我们知道软件系统的集成主要分为服务接口集成和数据集成。数据集成一般通过ETL（全称为Extract–Transform–Load，用来描述将数据从来源端经过抽取、转换、加载至目的端的过程）方式实现数据的传递、聚合等操作。ETL、实时数据流处理是数据领域与数据处理相关的技术话题，这里不赘述，本章我们只关心应用之间的交互技术和服务之间通过接口集成的技术。

微服务通常使用分布式跨网络的交互调用。我们通过网络协议、Linux I/O模式、序列化方式三个关键要素来讲解服务集成交互技术，它们是决定服务交互效果、影响交互效率最关键的因素。

### 6.1.1 网络协议

分布式系统采用跨网络的协议进行通信。网络协议是计算机网络中为进行数据交换而建立的规则、标准或约定的集合。无论是前端与后端之间，还是后端微服务之间，交互都会涉及网络协议。为了使不同计算机厂家生产的计算机能够相互通信，以便在更大的范围内建立计算机网络，国际标准化组织（ISO）提出了“开放系统互联参考模型”，即著名的OSI/RM模型（Open System Interconnection/Reference Model），它将计算机网络体系结构的通信协议划分为七层，自下而上依次为：

- 物理层（Physics Layer）
- 数据链路层（Data Link Layer）
- 网络层（Network Layer）
- 传输层（Transport Layer）
- 会话层（Session Layer）
- 表示层（Presentation Layer）
- 应用层（Application Layer）

TCP/IP无疑是当今互联网使用最广泛的网络互联协议。TCP/IP与OSI体系一样也采用了分层结构，它们之间的分层架构映射如下表所示。

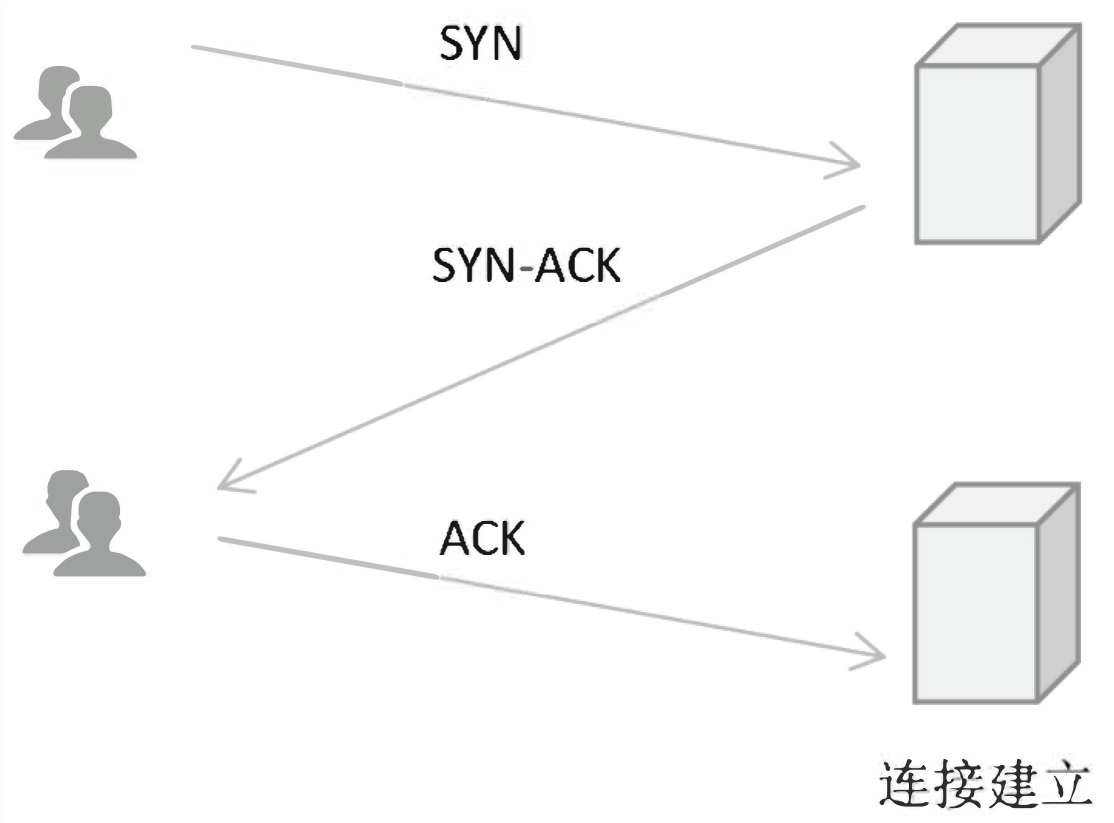
| OSI七层网络模型 | TCP/IP四层网络模型 | 对应网络协议 |
| --- | --- | --- |
| 应用层 | 应用层 | HTTP、FTP、SMTP |
| 表示层 | | Telnet、SNMP、Rlogin |
| 会话层 | | DNS、SMTP |
| 传输层 | 传输层 | TCP、DUP |
| 网络层 | 网络层 | IP、ICMP、ARP |
| 数据链路层 | 数据链路层 | Ethernet、Arpanet |
| 物理层 | | IEEE 802.1A、IEEE 802.2～IEEE 802.11 |

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF（The Internet Engineering Task Force，国际互联网工程任务组）的RFC 793定义。TCP旨在适应支持多网络应用的分层协议层次结构。连接到不同但互联的计算机通信网络的主计算机中的成对进程之间依靠TCP提供可靠的通信服务。

TCP主要包括两大事务：连接管理（建立、关闭连接），以及面向字节流的数据传输及控制。

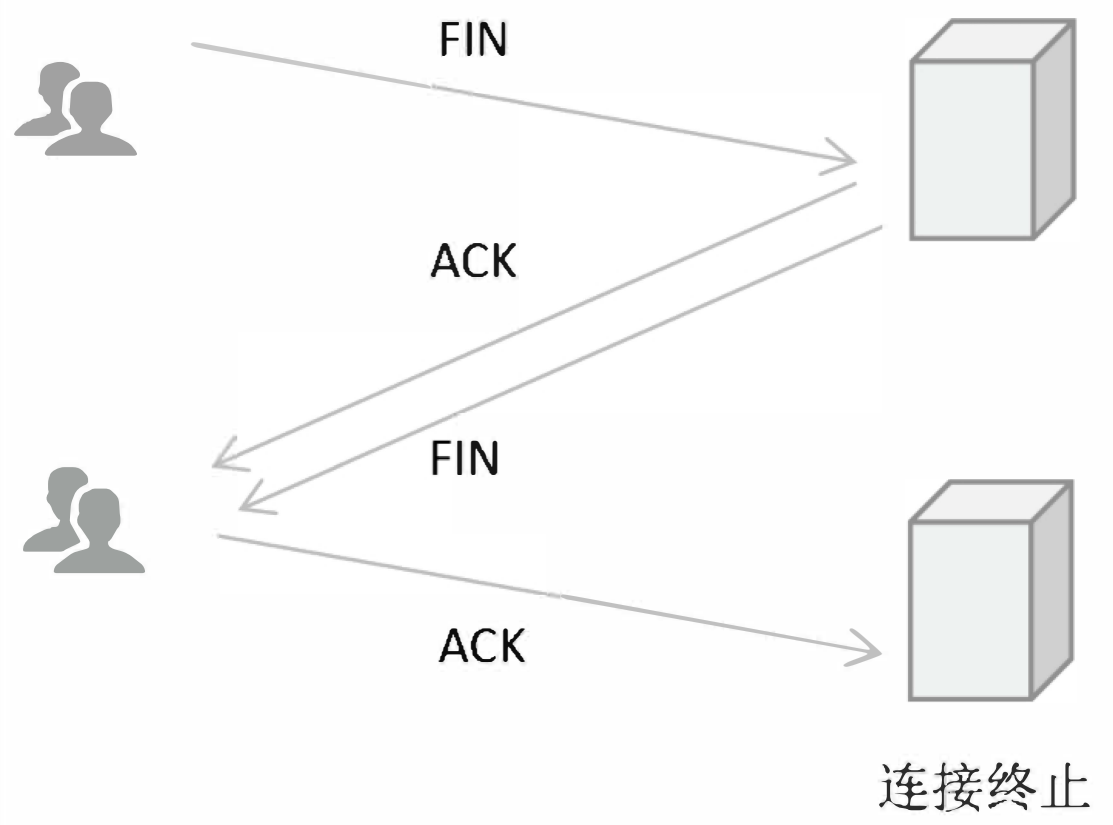
**建立连接**

TCP是互联网中的传输层协议，使用三次握手协议建立连接。当主动方发出SYN连接请求后，等待对方回答SYN-ACK，并最终对对方的SYN执行ACK确认。TCP三次握手如下图所示。




**终止连接**

建立一个连接需要三次握手，而终止一个连接要经过四次握手，这是由TCP的半关闭（Half-Close）造成的。具体TCP连接终止过程如下图所示。




**长短连接**

短连接是指当客户端与服务端建立连接后，双方完成一次读写过程，然后关闭连接，双方不需要额外的控制手段维持连接。优点是管理简单，缺点是延时高、效率低，每次传输数据需要重新建立连接。

长连接是指当客户端与服务端建立连接后，它们之间的连接不会主动关闭，后续的读写操作都会继续使用这个连接，优点是可以连续发送多个数据包，减少资源消耗、降低延时。

**数据传输**

下面是TCP在数据传输过程中的一些主要功能特性：

- TCP把数据流分割成适当长度的报文段，最大传输段大小（MSS）通常受该计算机连接的网络的数据链路层的最大传送单元（MTU）限制。
- 在可靠性上，TCP为了保证报文传输的可靠性，给每个包一个序号，序号也保证了传送到接收端实体的包按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认（ACK）。如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据（假设丢失了）将会被重传。
- 在正确性上，TCP用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算校验和来保证数据的正确性与合法性。
- 在流量控制上，采用滑动窗口协议，协议中规定对于窗口内未经确认的分组需要重传机制。

**微服务常用的应用协议**

- HTTP（Hyper Text Transfer Protocol，超文本传输协议）：HTTP可以说是目前互联网上使用的公共语言，Web浏览器、服务器和相关的Web应用程序都是通过HTTP完成通信的。HTTP基于TCP，属于应用层的面向资源存取和修改的协议，由于具有简捷、快速的特性，适用于分布式超媒体信息系统。HTTP对API技术无关性的支持是其最大的优点，浏览器的支持及不需要中间代理都简化了协议的架构。同时HTTP对RESTful架构的天然友好的支持，使其成为开发接口和系统集成的标准。
- gRPC：由Google开发，是一款语言中立、平台中立、开源的远程过程调用协议。基于HTTP2标准设计，具有诸如双向流、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。gRPC客户端应用可以像调用本地对象一样直接调用另一台机器上服务端应用的方法，使得我们能够更容易地创建分布式应用和服务。与许多RPC系统类似，gRPC也基于以下理念：定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）；在服务端实现这个方法，并运行一个gRPC服务器来处理客户端调用；在客户端拥有一个像服务端一样的方法。gRPC最大的优势就是语言无关性，对于微服务架构技术栈多样性的特性有非常好的支持，也是很多公司采用gRPC作为自己后端系统集成的协议标准的原因。
- AMQP：全称Advanced Message Queuing Protocol，是一个进程间传递异步消息的协议。AMQP使用长连接，是一个使用TCP提供可靠投递的应用层协议。RabbitMQ就是遵从AMQP协议开发的一个RPC远程调用框架。RabbitMQ中的交换器、交换器类型、队列、绑定、路由键等都遵循AMQP中相应的概念。可以将AMQP看作一系列结构化命令的集合，这里的命令代表一种操作，类似HTTP中的方法（GET、POST、PUT、DELETE等）。
- RSocket：是一种用于反应式应用程序的新网络协议，是第七层语言无关的应用网络协议，是一种基于Reactive Streams背压的双向、多路复用、消息的二进制协议。除了TCP，同时支持的通信协议还有UDP、WebSocket等。RSocket与HTTP的不同之处在于它定义了四种交互模型。
  - ○ fire-and-forget：异步触发，不需要响应。
  - ○ request/response：请求/响应，发出一个请求，获取一个响应，就像HTTP一样。
  - ○ request/stream：请求/流式响应，一个请求对应多个或无数个流式响应。
  - ○ channel：双向异步通信，也就是支持Channel。

**选择适合你的协议**

上述应用层协议是我们精心选择的微服务可能会使用的网络协议，不同的网络协议适合不同的应用场景。

- 如果你的微服务之间是“点对点”的请求/响应交互方式，可以采用基于HTTP的REST方式或者RPC方式的调用。一般来说，HTTP具备更好的通用性，RPC（如gPRC）交互的性能优势更加明显，使用何种方式作为你的微服务集成标准你需要做利弊权衡。至于使用同步I/O

还是异步I/O，这个基础性的选择会不可避免地影响后续的代码实现和技术架构，下一节我们会详细讲解I/O中的同步和异步、阻塞和非阻塞的相关技术和其对服务集成的影响。

- 如果你的微服务系统属于“异步执行”，或者属于一对多的发布/订阅场景，那么可以考虑使用消息中间件作为交互平台。服务生产者与服务消费者可以使用一个第三方消息代理平台通过事件驱动机制完成服务集成。
- 对于Reactive风格的应用，通常需要非阻塞并且与异步行为匹配。很多场景是服务端从客户端请求数据，支持单个连接上的多路复用，允许任意交互模式的双向消息流。这个时候HTTP可能已经无法满足你的需求，RSocket网络协议给微服务提供了很好的端到端的网络体验，我们将会在本书的进阶篇中进一步讲解。

### 6.1.2 Linux I/O模式

下图是根据同步、异步、阻塞、非阻塞四个指标总结的Linux下四个象限的I/O通信模式。

| | 阻塞 | 非阻塞 |
|---|---|---|
| 同步 | Read/Write | Read/Write nonblock |
| 异步 | I/O Multiplexing<br>Select/poll | AIO |

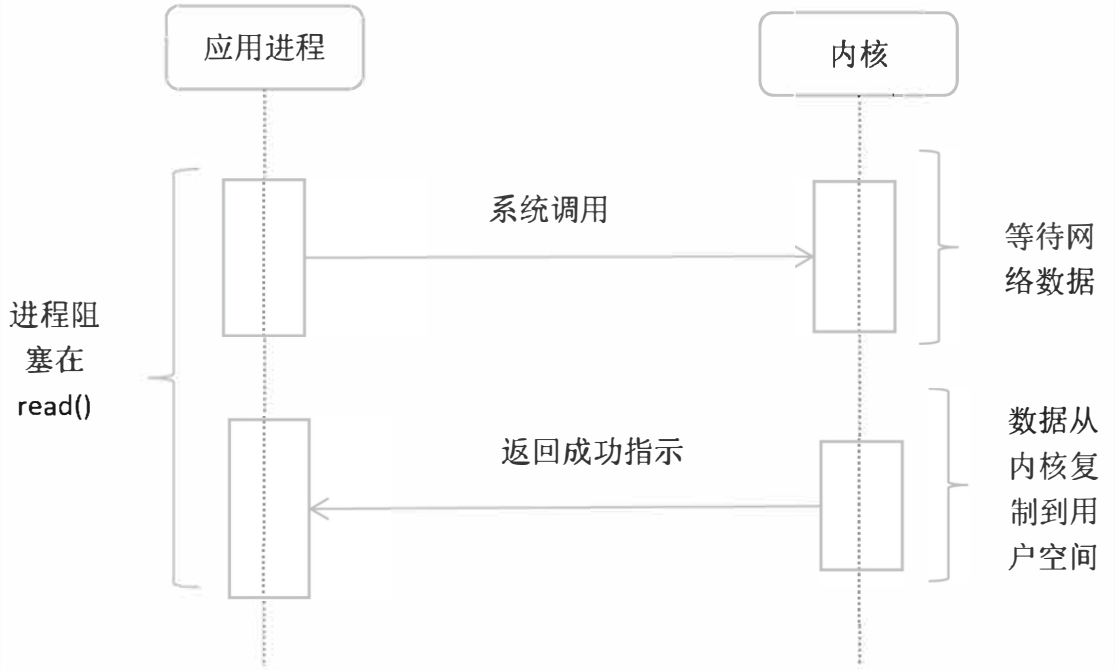
一般将网络消息是否有返回结果作为同步与异步的区分标准。

- 同步：应用程序要直接参与I/O读写操作，并等待消息响应结果。
- 异步：所有I/O读写交给操作系统去处理，不等待消息响应结果，程序只需要等待通知。

阻塞与非阻塞是指应用I/O读写操作是否阻塞。

- 阻塞：往往需要等待缓冲区中的数据准备好后才处理其他的事情，否则一直等待。
- 非阻塞：当进程访问数据缓冲区时，如果数据没有准备好则直接返回，不会等待。如果数据已经准备好，也直接返回。

同步阻塞流程图如下。




同步阻塞对应的Linux API为recvfrom（Linux下的Socket接收数据函数）。下面我们以读操作为例看一下同步阻塞模式工作流程：

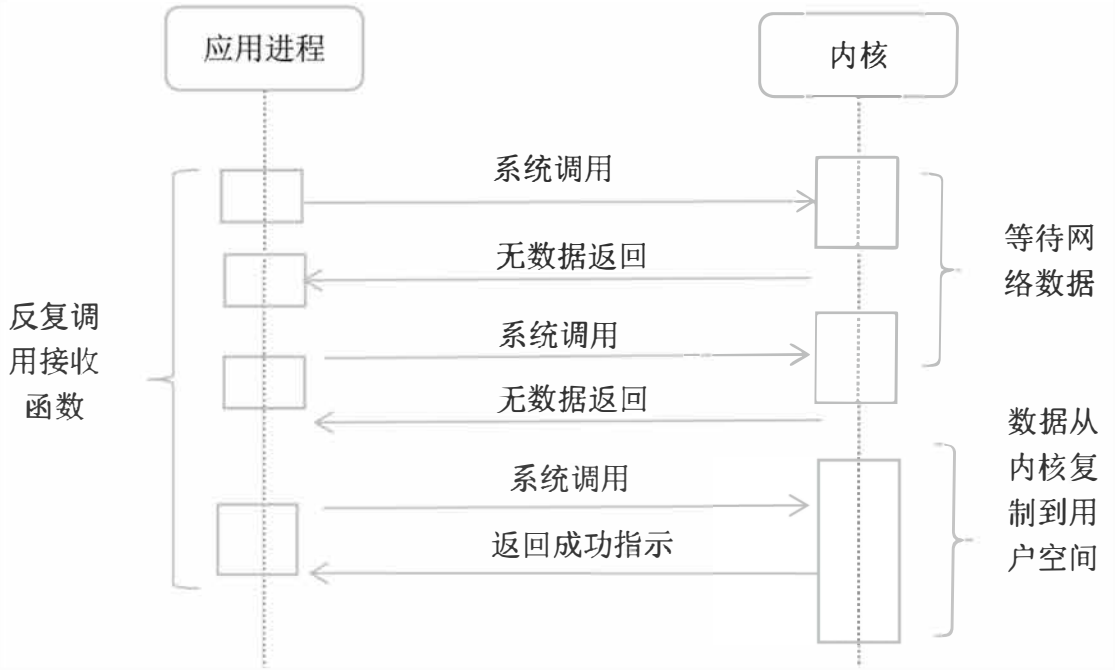
（1）进程发起读操作，进行recvfrom系统调用。

（2）内核开始进入准备数据（从磁盘复制到缓冲区）阶段，准备数据是要消耗一定时间的。

（3）与此同时，应用用户态进程阻塞，进入等待数据状态。

（4）数据从内核复制到用户空间，内核返回结果，进程解除阻塞。也就是说，内核准备数据和数据从内核复制到进程这两个过程都是阻塞的。

同步非阻塞流程图如下。



同步非阻塞对应的Linux API为recvfrom-noblocking。你可以通过设置Socket的初始化参数使其变为非阻塞状态。当对一个非阻塞状态的Socket执行读操作时，流程如下：

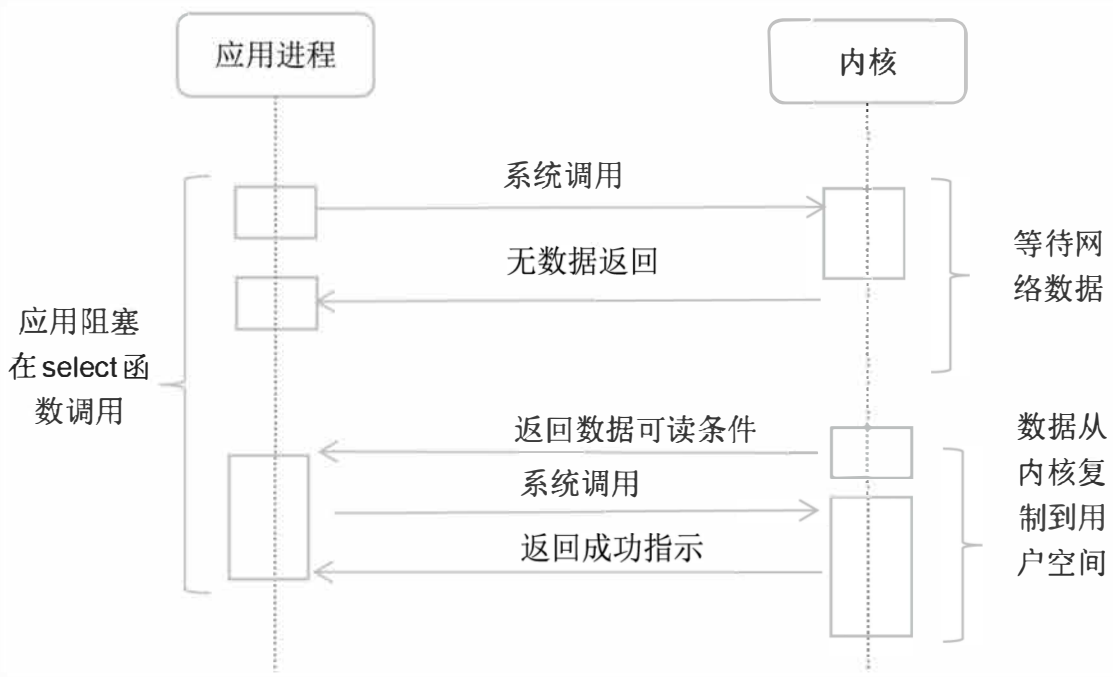
（1）用户进程发出读操作。

（2）如果内核中的数据还没有准备好，那么读操作并不会阻塞用户进程，而是立刻返回一个错误。从用户进程的角度来看，它发起一个读操作后不需要等待，马上就能得到返回结果。

（3）当用户进程再次发起读操作时，一旦内核中的数据准备好了，那么它马上就将数据复制到用户空间，然后返回。

非阻塞I/O的特点是用户进程在内核准备数据的阶段需要不断地主动询问数据好了没有。

异步阻塞流程图如下。




异步阻塞对应的Linux API为select、poll、epoll。异步阻塞其实就是我们经常提的I/O多路复用模式。epoll监听不同网络事件，当有事件通知时就通知用户进程。好处是单个进程可以处理多个Socket。现在看一下I/O多路复用的流程：

（1）当用户进程调用了select函数，那么整个进程会被阻塞。

（2）与此同时，内核会“监视”所有select函数负责的Socket（网络文件句柄）。

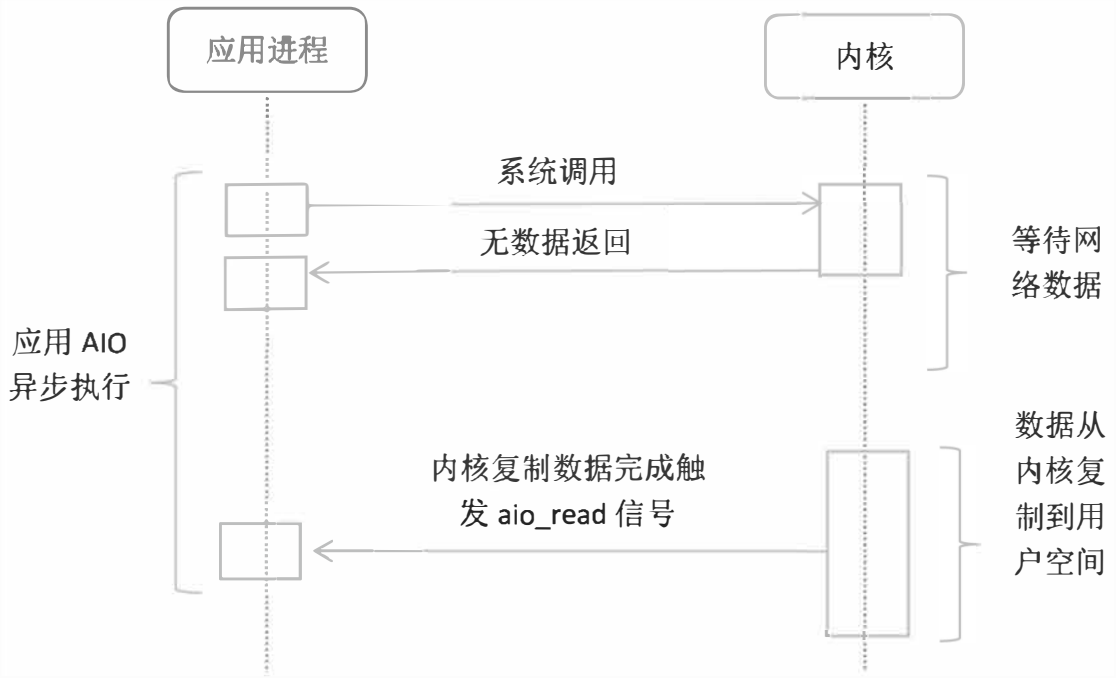
（3）任何一个Socket中的数据准备好了，select函数就会返回。

（4）这个时候用户进程再调用读操作，将数据从内核复制到用户空间。

所以，I/O多路复用的特点是通过一种机制，使用一个单独的线程同时等待多个文件描述符，而这些文件描述符其中的任意一个Socket（套接字描述符）进入读/写就绪状态，select函数就可

以返回。

异步非阻塞流程图如下。



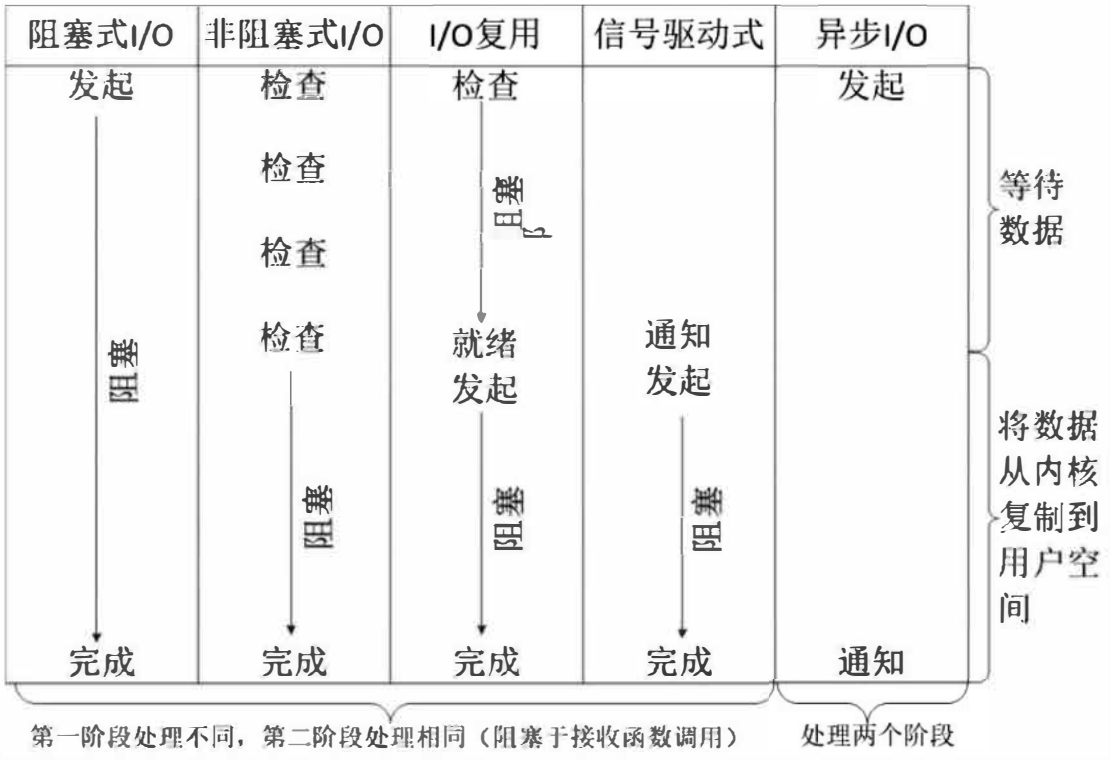

异步非阻塞对应的Linux API为aio_read/aio_write，流程如下。

（1）用户进程发起读操作之后，立刻开始去做其他事情。

（2）从内核的角度看，当它收到一个异步读操作之后，首先它会立刻返回，不会对用户进程产生任何阻塞。

（3）内核会等待数据准备完成，然后将数据复制到用户空间，当这一切都完成之后，内核会给用户进程发送一个信号，通知用户读操作已完成。

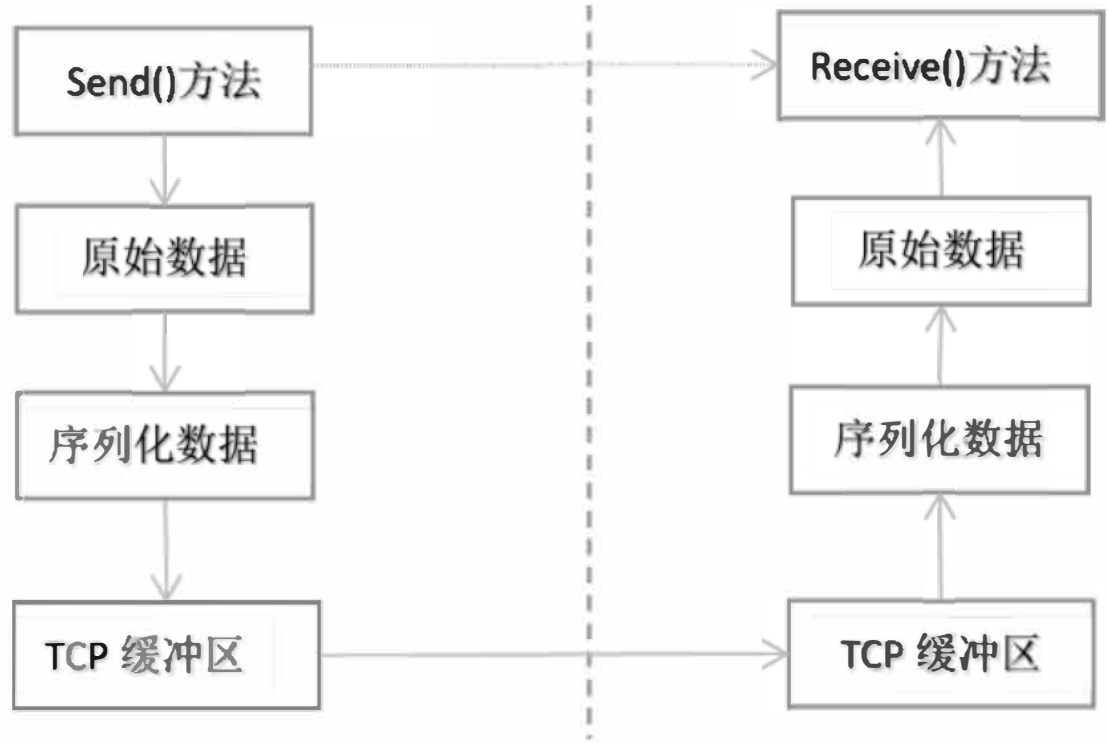
总结一下，上述四种I/O模式都可以分为两个阶段：一个是数据准备阶段，另一个是内核与用户空间的数据复制阶段，如下图所示。




四种I/O模式的主要区别就是第一阶段，因为第二阶段都需要阻塞等待数据从内核复制到用户空间。只有“异步非阻塞”的第二阶段与其他三种I/O模式都不相同，它在这两个阶段都是通过操作系统的事件回调通知完成的。

### 6.1.3 序列化方式

序列化是进程间在网络上进行数据交互所使用的消息格式。为了更好地理解序列化的概念，我们先从网络通信讲起。我们知道现在的网络通信技术大部分都是基于TCP/IP来实现的，下图展示的是网络上两台机器（发送端、接收端）之间通信的过程。



可以看出，序列化就是将对象转换为二进制形式（字节数组），一般也将序列化称为编码（Encode），主要用于网络传输、数据持久化等；而反序列化过程正好相反，它将从网络、磁盘等读取的字节数组还原为对象，一般也将反序列化称为解码（Decode），主要用于网络传输对象，以便完成远程调用。

网络传输对象的消息格式的技术选型对微服务架构非常重要，API的可扩展性、适应性、对内部技术的抽象封装性都与消息格式有关。可以说消息格式会给微服务的集成和通信效率带来极大的影响。

**基于文本的方式**

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。它的优点是易于阅读和编写，也易于机器解析和生成，它是ECMAScript的一个子集。JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、C++、C＃、Java、JavaScript、Perl、Python等）。简洁和清晰的层次结构使得JSON成为理想的数据交换格式，所以它不仅易于机器解析和生成，同时可以有效地提升网络传输效率。

XML同HTML一样，这种可扩展标记语言是标准通用标记语言的一个子集，它是描述网络上的数据内容和结构的标准。尽管如此，XML不像HTML，HTML仅仅提供了在页面上显示信息的通用方法（没有上下文相关功能和动态功能），XML则为数据赋予上下文相关功能，它继承了标准通用标记语言的大部分功能，却使用了不太复杂的技术。

JSON与XML的对比如下。

- 可读性：JSON和XML的可读性可谓不相上下，一边是简易的语法，另一边是规范的标记形式，两者各具优势。
- 可扩展性：XML天生有很好的扩展性，JSON的可扩展性也很强，不过JSON在JavaScript主场作战，可以存储JavaScript复合对象，有着XML不可比拟的优势。
- 编码难度：XML有丰富的编码工具，比如Dom4j、Dom、SAX等，JSON也提供相关编码工具。
- 与浏览器的兼容性：JSON作为JavaScript引擎能识别的数据格式，可以解决浏览器不兼容问题，而XML无法满足Ajax对浏览器的兼容性要求。

**基于二进制的方式**

在Java类中启用可序列化特性主要通过java.io.Serializable接口实现。它是一个标记接口，这意味着它不包含任何方法或字段，仅用于标识可序列化的语义。这个对象的所有属性（包括private属性、其引用的对象）都可以被序列化和反序列化来保存、传递。不需要序列化的字段可以使用transient修饰。由于Serializable对象完全以它存储的二进制位为基础来构造，它主要采用JDK自带的Java序列化实现，缺点是性能不理想。

Protobuf是一种和语言平台无关的数据交换格式，由一系列键值对组成。消息的二进制版本仅使用字段当作Key，不同字段的属性和类型只能通过“消息类型的定义”（即.proto文件）在解码端确定。Protobuf的优点就是传输效率高，序列化后体积相比JSON和XML很小，支持跨平台多语言，消息格式升级和兼容性还不错，序列化和反序列化速度很快。

上述两种二进制编码方式的主要区别是Protobuf有更好的平台中立性和编程语言无关性，Protobuf可以进行API版本升级，优于Java序列化方式。

## 6.2 REST服务集成

微服务架构倾向于使用轻量级的通信机制（通常是HTTP提供的API调用方式）实现服务之间的交互，基于API优先的服务契约管理成为微服务架构的重要原则之一。REST在HTTP的基础上提供了一系列架构约束和原则，帮助微服务更好地实现通信和集成。

### 6.2.1 REST API

REST的全称为Representational State Transfer，中文翻译为“表述性状态转移”或“表现层状态变化”。如果一个架构符合REST原则，则称它为RESTful架构。

**REST与HTTP**

首先要说明的是，虽然HTTP（1.0版本和1.1版本）的主要设计者和REST概念的提出者是同一个人，但是REST和HTTP有着本质的区别。HTTP本身是万维网的支撑协议，也是一项通用协议规范，而REST描述的则是客户端与服务端的一种交互形式。下面介绍HTTP和REST的主要区别。

- HTTP的详细内容可以参考RFC2616。HTTP采用了请求/响应模式。客户端向服务端发送一个请求，请求头包含请求的方法、URI、协议版本，以及请求修饰符、客户信息和内容的类似于MIME的消息结构。服务端以一个状态行作为响应，相应的内容包括消息协议的版本、成功或者错误编码加上服务端信息、实体元信息及可能的实体内容。

- REST本身并没有创造新的技术、组件、服务，隐藏在REST背后的理念是使用Web标准的现有特征和能力，强调Web组件交互的可扩展性、接口的独立性、减少交互延迟中间件。它的目标是更好地利用现有Web标准中的准则和规范，关注的是系统之间的通信行为细节，以及如何改进通信机制的表现。

**REST与JSON**

在服务集成交互技术中，我们已经介绍了两种主要的基于文本的序列化方式：JSON和XML。二进制格式的交互更多应用于RPC方式的交互集成，例如Google的Protocol Buffer和Facebook的Thrift。

在REST的序列化方式上，从灵活性的角度说，JSON无论从数据格式还是使用方式上都更加简单。JSON相比XML，无论在结构的紧凑性还是对浏览器的兼容性上，JSON都有得天独厚的优势。

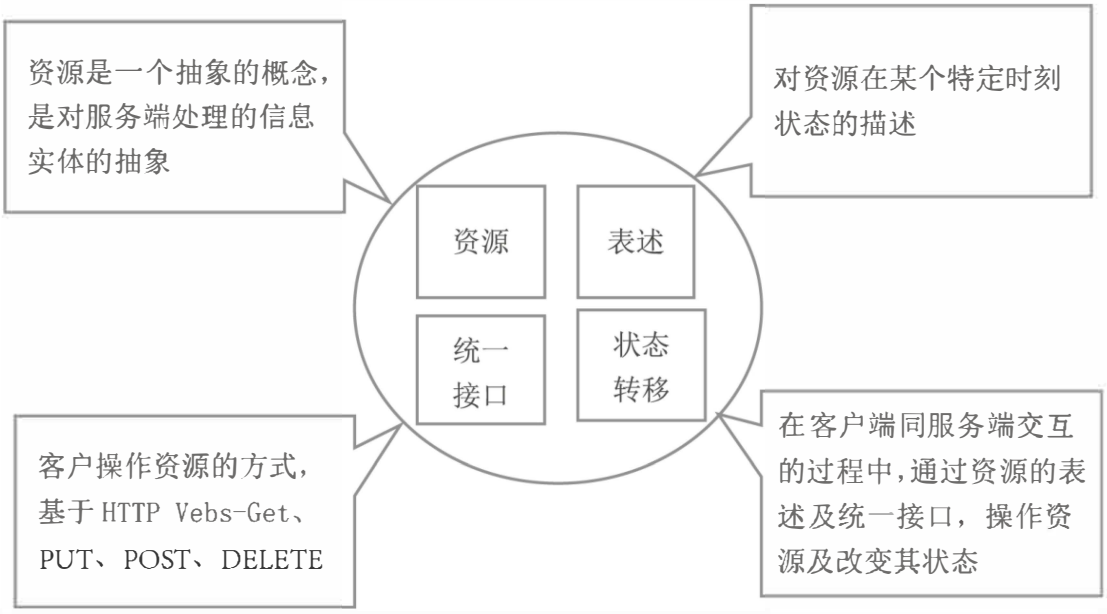
从序列化的性能方面来说，JSON没有过多的标签，JSON主要基于键值对的形式表示数据，所以传输和处理速度都有巨大的优势。

从对象的表述和数据结构与宿主语言的对应方面来看，JSON有更明显的优势，例如哈希表（Hashtable）、键值对（Key/Value）、向量（Vector）、列表（List）及对象组成的数据结构。XML在表达数据结构和对象的转换上都没有JSON方便。

当然XML也有JSON所不具备的优势，像通过标签可以添加属性来存储元数据（Metadata），可以使用连接进行超媒体控制等，当然我个人还是比较倾向于使用JSON。

**REST中的重要概念**

REST从语义层面将响应结果定义为资源，并使用HTTP的标准动词映射作为对资源的操作，形成了一种以资源为核心、以HTTP为操作方式的，与语言无关、平台无关的服务间的通信机制，如下图所示是REST的重要概念。




“资源”是REST中的重要概念，REST中的表现层状态转移的主语就是“资源”。“资源”就是网络中的一个实体，或者说是网络上的一个具体的信息，你可以使用URI（统一资源定位符）指向它，资源总是需要某种格式的载体，可以使用HTML、XML或者JSON表述资源内容。

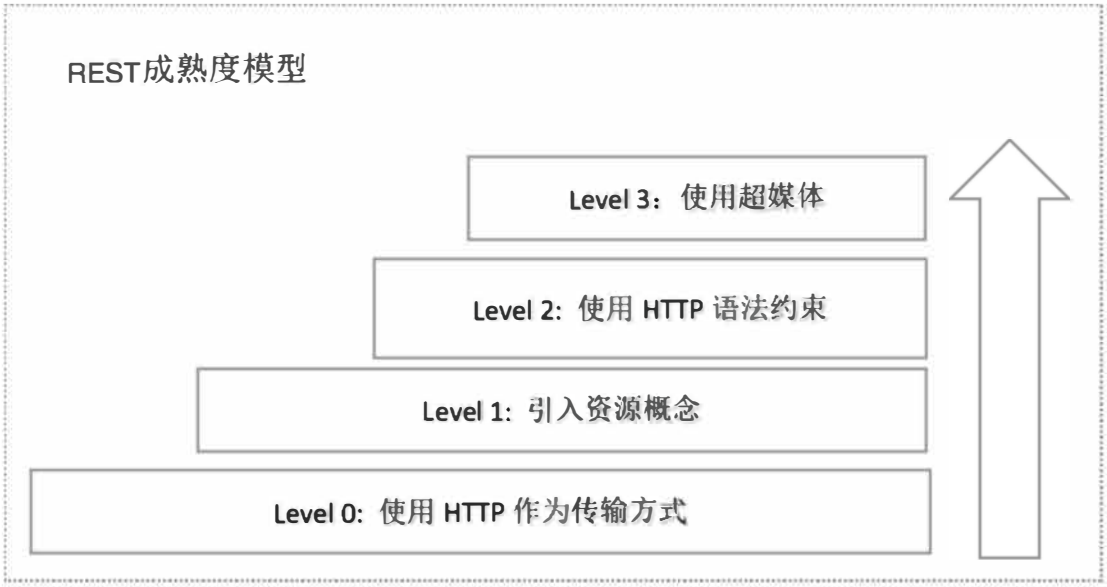
“表述”就是资源在某个特殊时刻具体呈现出来的形式和描述。一种资源可能有多种表述形式，而URI应该只代表资源的位置，它的具体表述形式应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对“表现层”的描述。

“状态转移”是指在客户端与服务端互动的过程中，通过某种手段实现对数据状态的变更。在HTTP中，GET用来获取资源、POST用来创建资源或者更新资源、PUT用来更新资源、DELETE用来删除资源。

“统一接口”包含一组受限的预定义操作，不论什么样的资源，都可以通过相同的接口进行资源的访问。接口应该使用标准的HTTP方法，如GET、PUT和POST，并遵循这些方法的语义。

**REST成熟度模型**

Leonard Richardson发明的REST成熟度模型可以帮助我们更好地使用REST，如下图所示。




Level 0：本层级是REST的最低级别，仅把HTTP作为传输协议来传输数据，还可以把SOAP、JSON-RPC都看成此类，仅仅使用请求/响应模式的通信风格来传递“Plain Old XML”。HTTP可降级为类似TCP的传输层协议。HTTP中的方法不包含业务逻辑语义。

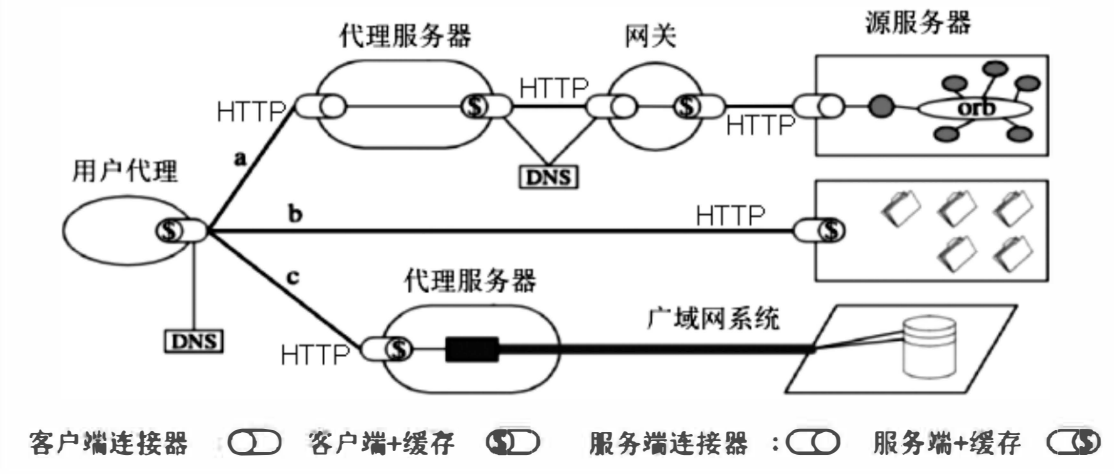
Level 1：本层引入了资源概念，每个资源对应后端的URI资源标识符，HTTP向服务资源端点（Service End-Point）发送POST请求，并向方法中添加参数。

Level 2：使用的API严格根据HTTP的Web语法执行对资源的处理和约束，例如GET用于读取资源、POST用于创建资源、PUT用于更新资源、DELETE用于删除资源。

Level 3：API基于HATEOAS原则设计，简单地说就是响应消息中包含后续操作的URI资源，Level 3拥有协议自描述功能。HATEOAS也是REST的高级形态，一个显而易见的好处是，客户端通过返回结果中的Link资源，可以更好地理解业务、适应变化。同时，这些链接对客户端和服务端也进行了解耦，你不再需要调整客户端来适应服务端的修改，通过双向的语义关联就可以更好地实现前后端分离。

**RESTful架构**

RESTful架构是一种典型的Client-Server架构，但是强调瘦服务端，服务端只应该处理跟数据有关的操作，所有RESTful架构显示相关的工作都应该放在客户端。REST的约束因素有Client-Server、Stateless、Cache、Uniform Interface、Layered System、Code-on-Demand。下图是我们总结的RESTful架构的特征和核心原则。




- 服务器是无状态的，服务端不会保存客户端的会话状态数据，所有状态信息都在双方沟通的消息中。
- 服务器是幂等的，对于相同请求，服务端返回的数据应该相同，所以服务端可以缓存结果，结果可以存储在服务端，也可以存储在客户端。
- 所有操作都基于统一接口（Uniform Interface）的方式进行，每个资源应该都是唯一的。
- 通过客户端来处理资源，也就是说客户端不能直接操作服务端的资源，只能通过响应表达式操作，并发送响应请求，最后由服务端处理资源并返回。
- 客户端和服务端传送的任何一个消息都是自描述的，处理消息需要的上下文都应该被包含在这个消息中。
- 在REST分层结构中，在Client–Server之前也可以加入Proxy层和Gateway层，这些中间层可以加入业务逻辑处理，例如安全控制、负载均衡。
- Code–On–Demand，客户端可以访问服务端的资源，但是并不知道处理服务器返回的结果，而这个处理过程的代码应该是从服务端发送过来的，然后在客户端执行，也就是说客户端的功能是根据需求动态从服务端获得的，这个特性在REST中是可选的。

**REST API的接入**

在传统的电信领域，我们使用CORBA（Common ObjectRequest Broker Architecture，公共对象请求代理体系结构）进行跨平台的交互，通过分布式对象调用来实现分布式架构，CORBA规范规定了ORB（Object Request Broker，对象请求代理）的标准体系。

虽然CORBA有非常严格的API契约机制和规范，然而CORBA的缺陷也是非常明显的，它是制约跨平台的技术发展的重要因素。

- CORBA是面向对象的分布式架构体系，将分布式机制完全绑定为以对象为中心的互操作模式，给分布式系统带来了极大的耦合性，给对象属性的变更带来了复杂性，也带来了不确定性，这个缺陷一直延续到了EJB时代。
- CORBA使用专有的二进制协议，通过IDL（Interface Description Language，接口描述语言）绑定和允许应用程序之间的互操作协议。通过编译IDL文件可以生成桩代码和框架。协

议的复杂性和庞杂的语义规范都增加了开发和运维的难度。SOAP也有CORBA类似的复杂信息交换协议机制，使用XML数据格式，它定义了一整套复杂的标签，WSDL（Web Services Description Language，Web服务描述语言）用来描述服务器地址和接口规范，XML传输效率问题也是SOAP的另一个问题。

REST本身使用HTTP，充分利用了HTTP的平台中立性和网络透传等优势。另外，最重要的是REST基于HTTP抽象资源的分布式调用，将分布式调用绑定在资源的操作上面，而在REST中，资源是一个抽象的概念，资源本身使用URI表示，与具体实现无关，这样就给REST带来了更好的解耦性。下面总结一下使用REST的好处：

- REST非常简单，基于HTTP，没有过多的模式限制。
- REST对浏览器友好，有众多工具和生态支持HTTP Client，例如可以使用curl、postman等工具和插件来测试HTTP-API。
- 支持请求/响应的通信方式。
- HTTP对防火墙友好。
- 通信不需要带中间件，简化了系统架构。

可以说，REST已然成为API开发集成的事实标准。当然，REST没有强制的IDL来定义API，不过目前在社区中也有很多流行的REST IDL规范，使用比较广泛的就是Swagger，它可以作为开发和记录REST API的工具，我们后续章节会加以介绍。下面我们来看一个虚拟的项目：在线商品服务介绍REST API规范接入手册。

**REST API请求示例**

【协议描述】

请求URL结构：https://domain/api/server/class? params，其中各字段含义如下：

- domain，请求地址的HOST&&PORT，假设域名为test.cn。
- api/server，固定值，服务所在的相对路径。
- class，具体调用方法的URL，参考下文的接口列表。
- params，公共请求参数，参考下文的请求参数。

【请求方式】

- 公共请求头参数有Timestamp时间戳，请将其置于HTTP API的请求头中。Timestamp为本请求的UNIX时间戳，用于确认请求的有效期，以秒为单位。
- URL内参数中包含可变字段，如/orders/orderid，orderid为URL内参数，需要对应填值，具体参考下文的接口列表。
- 对于POST请求参数，传递的参数必须使用JSON格式，公共请求参数仍置于URL中，具体方式可参考下文的代码示例。

【返回结果】

API接口使用标准HTTP返回码，只有2XX才是正确返回，下面是可能的返回码汇总：

- 200，请求成功，具体请求结果参考响应内容JSON值。
- 400，多数情况下是指请求参数错误或请求不合法。
- 401，sign值计算错误，或App已被删除。
- 404，设备或对应的App信息不存在，将返回NotFound错误。
- 50X，服务器错误，服务器内部数据或逻辑有误。

**【 REST API示例 】**

1.订单列表

- 描述：获取所有订单ID列表
- 路径：/orders
- 方法：GET
- 参数：page，count

```
curl-X GET https://test.cn/api/server/orders-H"Timestamp:1529051966"
```

2.订单详细信息查询

- 描述：获取指定设备详细信息
- 路径：/orders/orderid
- 方法：GET
- 参数：无

```
curl-X GET https://test.cn/api/server/orders/1234562342-H"Timestamp:1529051966"
```

3.更新订单信息

- 描述：为设备更名，或禁用/恢复设备
- 路径：/orders/orderid
- 方法：PUT
- 参数：无

```
curl-X GET https://test.cn/api/server/orders/1234562342_-H"Timestamp:1529051966"
```

## 6.2.2 Swagger接口文档规范

上一节中我们使用REST标准描述了一个使用订单服务的API文档。然而API文档的维护，给技术人员带来了额外的工作量。另外，技术文档人员对API的理解的偏差，也给API文档的可操作性带来了问题。对于后端人员，Swagger为我们提供了一个在线的REST API文档，可以提供给不同团队，加速开发过程，也提供了更加便携的基于标准文档的交互方式。

**Swagger API标准**

Swagger是一个规范和完整的框架，用于生成、描述、调用和可视化RESTful风格的Web服务。Swagger的目标是定义标准的、和语言无关的接口，让人和计算机无须访问源码、文档或进行网络流量监测就可以发现和理解服务的能力。

Swagger规范定义了一组描述一个API所需的文件格式，类似于描述Web服务的WSDL。通过Swagger进行REST API的正确定义，用户可以理解远程服务并使用最少实现逻辑与远程服务进行交互。与底层编程所实现的接口类似，Swagger消除了调用服务时产生的理解差异。

**集成Swagger的步骤**

1.导入Maven依赖

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

2.实现Swagger配置类

```
@EnableSwagger2  //Swagger2 启动注解
@ComponentScan(basePackages = {"cn.demo.controller"})//扫描 Controller 类
@Configuration
public class SwaggerConfig extends WebMvcConfigurationSupport
{
    @Bean
    public Docket createRestApi() {
       return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
 }
 private ApiInfo apiInfo() {
     return new ApiInfoBuilder()
        .title("《SwaggerDemo》")
        .description("description:摘要")
        .contact(new Contact("Devil",  )
        .version("6.6.6")
        .build();
}
```

注解说明：

● @EnableSwagger2：Swagger2启动注解。

● @Configuration：声明这是一个配置类。

3.配置接口的API参数描述

```
@Controller
@Api(value="aaaaaaa", description="User 的相关信息接口")
public class UserController {
    @Autowired
    private UserService userService;
    @RequestMapping("/getAll")
    @ResponseBody
    @ApiOperation(value="获取所有 user", notes="获取所有 user", httpMethod = "POST")
    public String getAll(){
        //查出的所有部门信息
        List<User> list = userService.selectAll();
        return list.toString();
    }
}
```

**网关集成Swagger**

Swagger是一个API文档生成工具，在微服务架构中，API网关可以起到聚合后端众多微服务的作用，同时可以利用微服务网关集成Swagger生成所有微服务的接口文档。下面是基于网关Zuul集成Swagger的文档示例。

```
@Configuration
@EnableSwagger2
@Primary
//实现 SwaggerResourcesProvider，配置 Swagger 接口文档的数据源
public class SwaggerConfig  implements SwaggerResourcesProvider {
    //RouteLocator 可以根据 Zuul 配置的路由列表获取服务
    private final RouteLocator routeLocator;
    public SwaggerConfig(RouteLocator routeLocator) {
        this.routeLocator = routeLocator;
    }
    //这个方法用来添加 Swagger 的数据源
    @Override
    public List<SwaggerResource> get() {
        List resources = new ArrayList();
        List<Route> routes = routeLocator.getRoutes();
        //通过 RouteLocator 获取路由配置，遍历所配置服务的接口文档，实现动态获取
        for (Route route: routes) {
            resources.add(swaggerResource(route.getId(),
                route.getFullPath().replace("**", "v2/api-docs"), "2.0"));
        }
        return resources;
    }
    private SwaggerResource swaggerResource(String name, String location, String version) {
        SwaggerResource swaggerResource = new SwaggerResource();
        swaggerResource.setName(name);
        swaggerResource.setLocation(location);
        swaggerResource.setSwaggerVersion(version);
        return swaggerResource;
    }
}
```

## 6.2.3 JAX-RS提供REST服务

如果你更喜欢JAX-RS为REST端点提供的编程模型，你可以使用相应的实现代替Spring MVC框架。Spring Boot支持Jersey1.x和Jersey2.x，我们这里只介绍Spring Boot对Jersey2.x的支持。

首先，引入Spring Boot针对Jersey的starter包：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jersey</artifactId>
</dependency>
```

创建一个ResourceConfig类型的@Bean组件，用于注册所有的端点（Endpoint），可以注册任意数量的，然后实现ResourceConfigCustomizer的Bean来进一步自定义端点功能。

```
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
     register(Endpoint.class);
    }
}
```

注册的所有端点都需要注解@Components和HTTP资源Annotations（比如@GET）：

```
@Component
@Path("/hello")
public class Endpoint {
    @GET
    public String message() {
        return "Hello";
    }
}
```

Endpoint是一个Spring组件（@Component），它的生命周期受Spring容器管理，你可以使用@Autowired添加依赖，也可以使用@Value注入外部配置。Jersey的Servlet会被注册，并默认映射到/*，你可以将@ApplicationPath添加到ResourceConfig来改变该映射。默认情况下，Jersey将以Servlet的形式注册为一个ServletRegistrationBean类型的@Bean。通过创建相同Name的Bean组件，可以禁用或覆盖框架默认产生的Bean。设置spring.jersey.type=filter可以使用Filter的形式代替Servlet，相应的@Bean类型变为jerseyFilterRegistration，该Filter有一个@Order属性，你可以通过spring.jersey.filter.order设置。注册Servlet和Filter时都可以使用spring.jersey.init.*定义一个属性集合并传递给init参数。

## 6.2.4 Feign实现REST调用

Feign是一个声明式的Web Service客户端，它使得编写Web Service客户端更为容易。Feign受到Retrofit、JAXRS2.0、WebSocket的影响，采用声明式的API调用模式。

**Feign的特征**

- Feign基于声明式的REST调用方式，相比Rest-Template、HTTPClient等命令性HTTP客户端，Feign通过代理模式屏蔽了调用方与底层HTTPClient技术耦合的调用细节。Feign的调用就像使用本地方法调用完成服务的请求。
- Feign简化了请求的编写，可以动态地选择使用HTTP客户端实现，可以结合Eureka、Ribbon、Hystrix等组件实现服务发现、负载均衡、熔断等。

- Spring Cloud对Feign进行了封装，它支持可插拔的注解，所以支持Spring MVC标准注解和使用HttpMessageConverters模块做消息转换。
- Feign可以对请求进行拦截，提供HTTP模板，使用简单的注解和配置实现定义HTTP请求的参数、格式、地址等信息。它支持可插拔的HTTP编码器和解码器等。

**Feign的使用方法**

首先，需要在pom.xml文件中引入对Feign的依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

其次，创建应用主类Application，并通过@EnableFeignClients注解开启Spring Cloud Feign的支持功能：

```
@EnableEurekaClient
@SpringBootApplication
@EnableFeignClients(basePackages = { "com.demo" })
public class Application {
     public static void main(String[] args) {
        SpringApplication.run(Application.class， args);
    }
}
```

然后，定义DemoServiceFeign接口，接口中使用@FeignClient注解指定服务名来绑定服务，之后使用Spring MVC的注解来绑定具体该服务提供的REST接口功能：

```
@FeignClient(value = "demo-service-provider")
public interface DemoServiceFeign {
    @RequestMapping(value = "/demo/getHost"， method = RequestMethod.GET)
    public String getHost(String name);
    @RequestMapping(value = "/demo/postPerson"， method = RequestMethod.POST，
      produces = "application/json; charset=UTF-8")
    public Person postPerson(String name);
}
```

最后，创建一个RestClientServcie服务来实现对Feign客户端的调用，使用@Autowired直接注入上面定义的HelloServiceFeign实例：

```
@Service
public class RestClientService {
    @Autowired
    private DemoServiceFeign client;
    public Person  gethost(String name) {
       return client.getHost(name);
  }
}
```

**Feign的源码解析**

Feign是一个伪Java HTTP客户端，Feign本身不做任何请求处理。Feign借鉴AOP设计思想，通过注解生成HTTP Request模板，从而简化HTTP API的开发。我们可以使用注解的方式定制Request API模板，分离HTTP请求使用者与具体实现。

下面从注解@EnableFeignClients开始讲解，深入理解Feign声明式的HTTP客户端的调用过程。

@EnableFeignClients的主要注解声明如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(FeignClientsRegistrar.class)
public @interface EnableFeignClients {
    String[] value() default {};
    String[] basePackages() default {};
    Class<?>[] basePackageClasses() default {};
    Class<?>[] defaultConfiguration() default {};
    Class<?>[] clients() default {};
}
```

注解@EnableFeignClients告诉Spring Boot容器扫描所有使用注解@FeignClient定义的Feign客户端。通过注解@Import导入FeignClientsRegistrar类。

FeignClientsRegistrar类实现了接口ImportBeanDefinitionRegistrar，这个接口的设计目的就是被某个实现类实现，配合@Configuration注解的使用者配置类使用，在配置类被处理时，用于额外注册一部分Bean定义，代码如下：

```
@Override
public void registerBeanDefinitions(AnnotationMetadata metadata,
        BeanDefinitionRegistry registry) {
    registerDefaultConfiguration(metadata, registry);//客户端的默认配置的注册
    registerFeignClients(metadata, registry);//
}
```

@EnableFeignClients注解被处理时的调用栈如下：

```
AbstractApplicationContext#invokeBeanFactoryPostProcessors
  PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors
    foreach BeanDefinitionRegistryPostProcessor : #postProcessBeanDefinitionRegistry
      ConfigurationClassPostProcessor#postProcessBeanDefinitionRegistry
        processConfigBeanDefinitions
          ConfigurationClassBeanDefinitionReader#loadBeanDefinitions
            foreach ConfigurationClass : #loadBeanDefinitionsForConfigurationClass
                loadBeanDefinitionsFromRegistrars
                  foreach ImportBeanDefinitionRegistrar : #registerBeanDefinitions
                    FeignClientsRegistrar#registerBeanDefinitions
```

在程序启动后，程序会通过包扫描将由@FeignClient注解修饰的接口连同接口名和注解信息一起取出，赋值给BeanDefinitionBuilder，再根据BeanDefinitionBuilder得到BeanDefinition，最后将BeanDefinition注入Spring IoC容器中，源码如下：

```
public void registerFeignClients(AnnotationMetadata metadata,
    BeanDefinitionRegistry registry) {
    ClassPathScanningCandidateComponentProvider scanner = getScanner();
        scanner.setResourceLoader(this.resourceLoader);
    Set<String> basePackages;
    Map<String, Object> attrs = metadata
    //省略
    Map<String, Object> attributes = annotationMetadata.getAnnotationAttributes(
        FeignClient.class.getCanonicalName());
    String name = getClientName(attributes);
    registerClientConfiguration(registry, name, attributes.get("configuration"));
    registerFeignClient(registry, annotationMetadata, attributes);
    }
}
private void registerFeignClient(BeanDefinitionRegistry registry,
        AnnotationMetadataannotationMetadata,
        Map<String, Object> attributes) {
    String className = annotationMetadata.getClassName();
     //声明代理类名称
    BeanDefinitionBuilder definition = BeanDefinitionBuilder
        .genericBeanDefinition(FeignClientFactoryBean.class);
        //省略
    BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDefinition,
        className, new String[] { alias });
    //将 BeanDefinition 注入 Spring IoC 容器
    BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);
}
```

在注入BeanDefinition之后，通过JDK动态代理机制生成FeignClient代理对象，使用FeignClient接口里面的方法时，方法会被拦截，源码在ReflectiveFeign类中，如下所示：

```
public <T> T newInstance(Target<T> target) {
    //target.type 接口类上的方法和注解解析成 MethodMetadata，并转换成内部的 MethodHandler
    Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);
    Map<Method, MethodHandler> methodToHandler =
            new LinkedHashMap<Method, MethodHandler>();
    List<DefaultMethodHandler> defaultMethodHandlers =
            new LinkedList<DefaultMethodHandler>();
    //省略
    for (Method method : target.type().getMethods()) {
        if (method.getDeclaringClass() == Object.class) {
        continue;
  } else if(Util.isDefault(method)) {
    DefaultMethodHandler handler = new DefaultMethodHandler(method);
    defaultMethodHandlers.add(handler);
    methodToHandler.put(method, handler);
  } else {
    methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(),
        method)));
  }
 }
 InvocationHandler handler = factory.create(target, methodToHandler);
 T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(),
        new Class<?>[]{target.type()}, handler);
  for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
    defaultMethodHandler.bindTo(proxy);
  }
 return proxy;
}
```

从factory.create（target，methodToHandler）可以看出，InvocationHandler实际委托给了methodToHandler，而methodToHandler默认是SynchronousMethodHandler.Factory工厂类创建的，newInstance方法最终生成Feign的动态代理。

在调用过程中，Feign首先会通过代理对象调用FeignInvocationHandler的invoke方法，代码如下：

```
private final Map<Method, MethodHandler> dispatch;
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
    //Method 方法对应一个 MethodHandler
    return dispatch.get(method).invoke(args);
}
```

最终，在methodToHandler匹配成功后，调用SynchronousMethodHandler类的invoke方法进行HTTP拦截。它会根据参数生成RequestTemplate对象，该对象是HTTP请求模板，其中的executeAndDecode方法会执行HTTP请求，并获取Response响应，代码如下：

```
@Override
public Object invoke(Object[] argv) throws Throwable {
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    Retryer retryer = this.retryer.clone();
    while (true) {
      try {
        return executeAndDecode(template);
      } catch (RetryableException e) {
        retryer.continueOrPropagate(e);
        if (logLevel != Logger.Level.NONE) {
          logger.logRetry(metadata.configKey(), logLevel);
        }
        continue;
      }
    }
}
```

## 6.3 RPC远程过程调用

在微服务架构中，使用RPC（Remote Procedure Call）进行服务之间的交互是我们通常采用的一种集成方式，与REST方式的请求调用模式相比，RPC具有更强的契约规范（Schema），同时相比REST方式也有更好的性能优势。

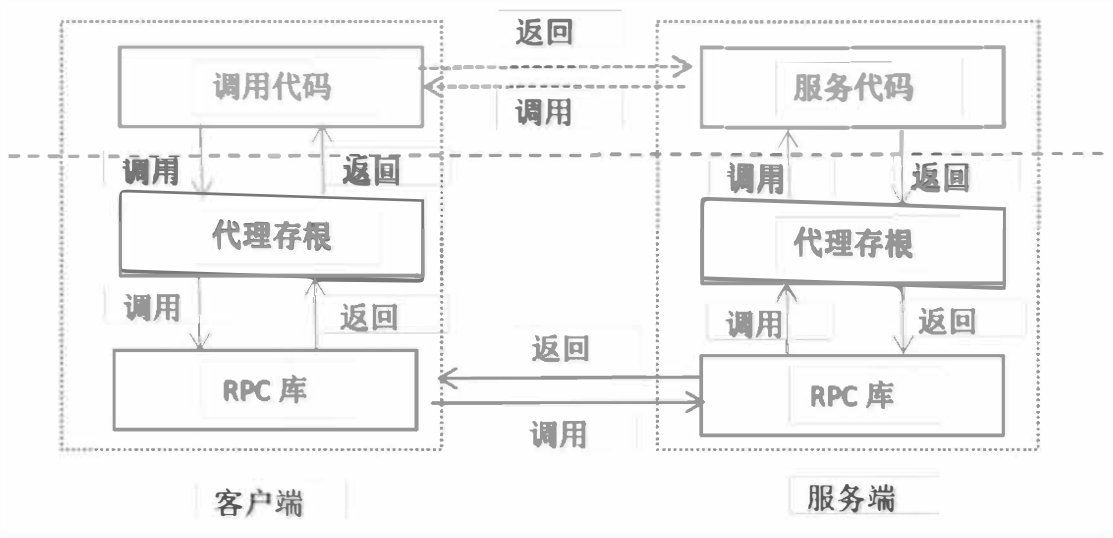
### 6.3.1 RPC框架概述

RPC是一种进程间通信方式，可以像调用本地服务一样调用远程服务。RPC的核心并不在于使用什么协议，RPC的主要目标是让远程服务调用更简单、透明，让远程方法调用如本地调用一样方便，我们并不需要知道这个调用的方法所部署的网络位置。

**RPC通信过程**

RPC主要用到了动态代理模式。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。开发人员在使用的时候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，并不需要关心底层通信细节和调用过程，如下图所示。

- 远程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、中间态的服务定义文件。例如Facebook的Thrift的IDL文件，Web Service的WSDL文件，服务的调用者需要通过一定的途径获取远程服务调用相关的信息。
- 远程代理对象：远程代理对象是为一个对象在不同地址空间提供的局部代表，它可以将一个对象隐藏于不同地址空间。




- 通信：RPC框架与具体的协议无关。
- 序列化：毕竟是远程通信，需要将对象转化成二进制流进行传输。不同的RPC框架应用的场景不同，在序列化上也会采用不同的技术。

**RPC和REST对比**

RPC和REST在请求模式上都属于点对点的请求/响应模式。REST是一种设计风格，它的很多思维方式与RPC是相互冲突的。RPC的思想是把本地函数映射到API，也就是说一个API对应的是一个函数方法（Function）。例如本地有一个getAllUsers方法，能通过某种约定的协议来远程调用这个getAllUsers方法，至于这个协议是Socket、HTTP还是其他协议并不重要，RPC中的主体是动作，强调互操作性。而REST则不然，它的URL链接主体是资源，REST强调远端的资源访问，而且REST大多数基于HTTP完成，规定了使用HTTP Method表达本次要做的动作，类型一般也不超过四五种。这些动作表达了对资源仅有的几种转化方式。

RPC最大的劣势是“紧耦合”，RPC客户端以多种方式与服务实现紧密耦合，它很难在不中断客户端的情况下更改服务实现。所以RPC的集成交互方式更偏向内部调用，而REST更偏向外部调用。Web服务应该算RPC的一个子集，理论上RPC能实现的功能用Web服务也能实现，甚至很多RPC框架选用HTTP作为传输层。

我们讨论RPC和Web的区别，主要关注两个因素：序列化协议和传输协议。序列化协议比如常见的XML，以及JSON和比较现代的Protocol Buffers、Thrift。传输协议比如TCP、UDP，以及更高层的HTTP 1.1、HTTP 2.0。

一般我们用RPC而不是HTTP构建自己的服务，通常考虑下面的因素：

- 接口是否需要Schema约束。
- 是否需要更高效的传输协议（TCP、HTTP 2.0）。
- 是否对性能、数据包的大小非常敏感。

HTTP是基于文本的协议，头部需要提供多冗余（对于RPC服务而言）的协议头信息。HTTP中我们用得最多就是RESTful框架，而RESTful框架本质上基于弱Schema约束协议。对于RPC方式的调用来说，Thrift这种序列化协议的优势在于有非常严格的IDL（交互规范约束Schema）的存在，可以保证服务端接收的参数和Schema保持一致。

**RPC与Web Service对比**

Web Service是在RPC的基础上发展而来的。它可以使用开放的XML来描述、发布、发现、协调和配置这些应用程序，它可以用于开发分布式交互操作的应用程序。而RPC使用C/S方式发送请求到服务器，等待服务器返回结果。

Web Service提供的服务是基于Web容器的，底层使用HTTP，类似于一个远程的服务提供者。比如天气预报服务，对各地客户端提供天气预报，是一种请求应答机制，是跨系统、跨平台的。

**RPC与RMI的对比**

RMI只用于Java中以客户端的存根对象（Stub）作为远程接口进行远程方法的调用。每个远程方法都具有方法签名。如果一个方法在服务器上执行，但是没有匹配的签名被添加到这个远程接口（Stub）上，那么这个新添加的方法就不能被RMI客户端所调用。

RPC是网络服务协议，与操作系统和语言无关，RPC通过网络服务协议向远程主机发送请求，请求包含一个参数集和一个文本值，通常形成“classname.methodname（参数集）”的形式。RPC远程主机会去搜索与之相匹配的类和方法，找到后就执行方法并把结果编码，再通过网络协议发回。

## 6.3.2 主流RPC通信框架

目前流行的开源RPC框架还是比较多的，使用比较多的有阿里巴巴的Dubbo、Facebook的Thrift、Google的gRPC等。

- Dubbo：阿里巴巴开源的一个RPC框架，在很多互联网公司和企业应用中广泛使用。协议和序列化框架都具备可插拔等特性，应用可通过高性能的RPC实现服务的输出和输入功能，它可以和Spring框架无缝集成。
- gRPC：Google开发的高性能、通用的开源RPC框架。gRPC由Google面向移动应用开发，并基于HTTP2协议标准设计。它基于ProtoBuf（Protocol Buffers）序列化协议开发，且支持众多开发语言。
- Thrift：Facebook开源的RPC框架，它是一个跨语言的服务开发框架。应用对于底层的RPC通信是透明的。不过对于用户来说需要学习特定领域语言，还是有一定成本的。
- Motan：新浪微博开源的一个Java框架。Motan在微博平台中已经广泛应用，每天为数百个服务完成近千亿次的调用。与Dubbo相比，Motan在功能方面并没有那么全面，也没有实现特别多的扩展，对跨语言调用支持较差，主要支持Java。
- Hessian：Hessian是一个Web Service框架，支持RPC调用，功能简单，使用起来也方便。它采用二进制RPC协议，基于HTTP进行传输，通过Servlet提供远程服务，可以通过Hessain提供的API来发起请求和接收请求。
- Spring Cloud：Pivotal公司在2014年对外开源的RPC框架，它仅支持Java，最近几年生态发展得比较好，是Java领域目前事实上的微服务框架标准。

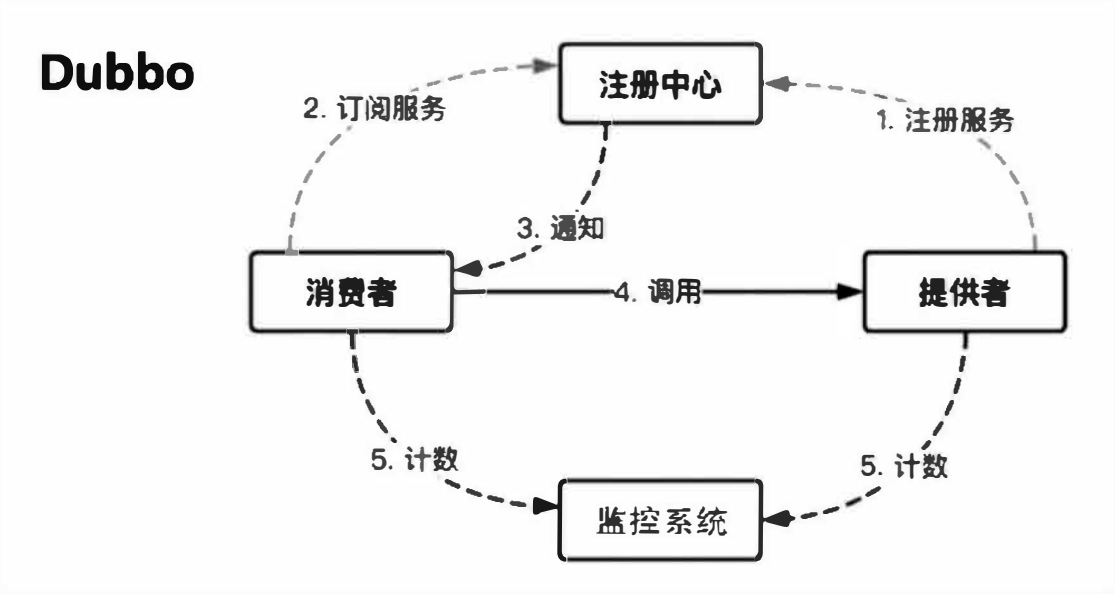
RPC框架的功能比较如下表所示。

| 功能 | Hessian | Montan | gRPC | Thrift | Dubbo | Spring Cloud |
|---|---|---|---|---|---|---|
| 开发语言 | 跨语言 | Java | 跨语言 | 跨语言 | Java | Java |
| 服务治理 | × | √ | × | × | √ | √ |
| 多序列化框架支持 | Hessian | √（支持Hessian2、JSON，可扩展） | ×（只支持ProtoBuf） | ×（Thrift格式） | √ | √ |
| 多种注册中心 | × | √ | × | × | √ | √ |
| 管理中心 | × | √ | × | × | √ | √ |
| 跨编程语言 | √ | ×（支持PHP和C） | √ | √ | × | × |
| 支持REST | × | × | × | × | × | √ |
| 关注度 | 低 | 中 | 中 | 中 | 中 | 中 |
| 上手难度 | 低 | 低 | 中 | 中 | 低 | 中 |
| 运维成本 | 低 | 中 | 中 | 低 | 中 | 中 |
| 开源机构 | Caucho | Weibo | Google | Apache | 阿里巴巴 | Apache |

通过上面不同功能的对比，我们可以看到，不同的RPC框架有不同的适用场景。如果你更加关注服务治理，这类框架能够提供包括服务注册、管理中心在内的整套的微服务技术架构支持，典型代表有Spring Cloud、Dubbo、Montan。如果你更关注跨语言服务调用，典型代表有Hessian、gRPC、Thrift。

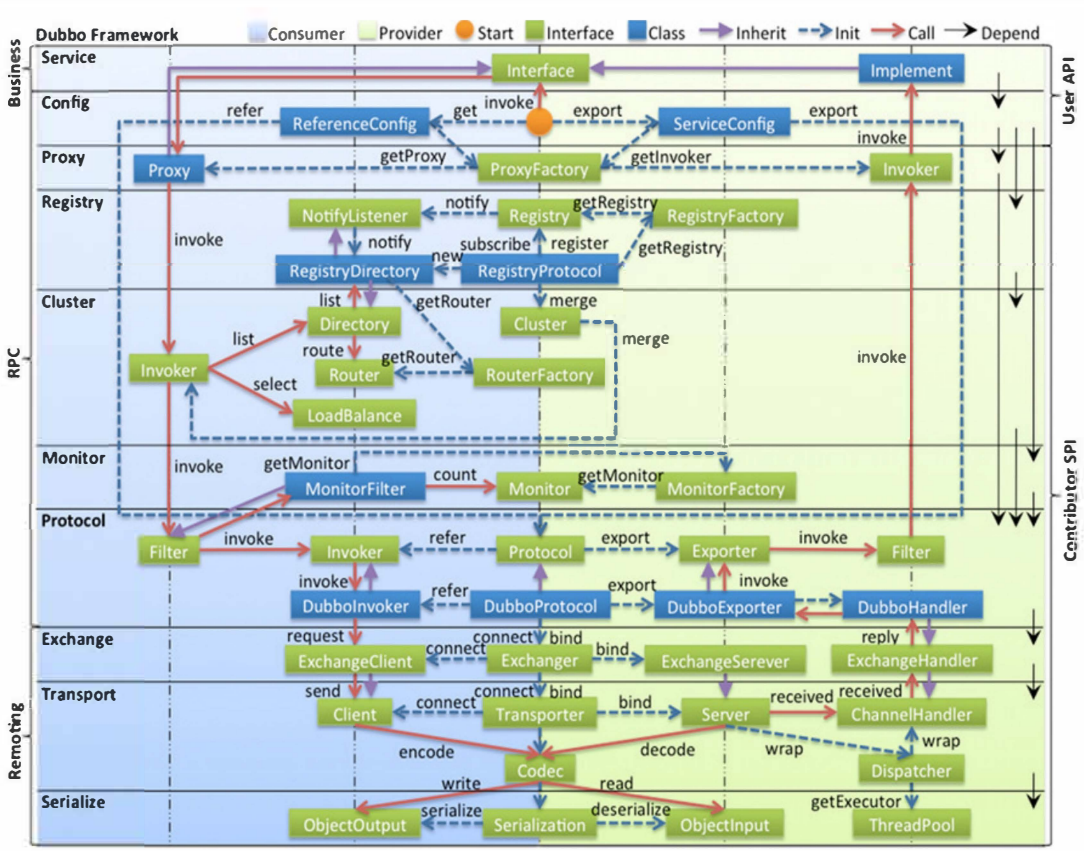
### 6.3.3 Dubbo架构进阶

Dubbo架构主要包含四个角色：消费者、提供者、注册中心和监控系统，如下图所示。




具体的交互流程是：消费者（Consumer）通过注册中心获取提供者（Provider）节点后，通过Dubbo的客户端SDK与Provider建立连接，并发起调用。Provider通过Dubbo的服务端SDK接收Consumer的请求，处理后再把结果返回给Consumer。

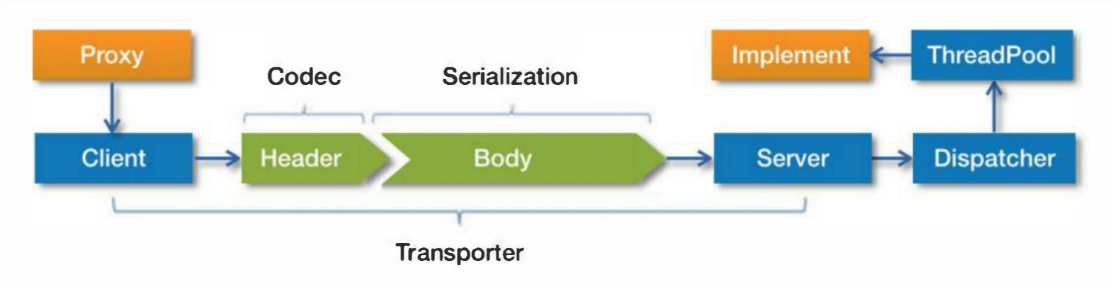
对于采用Dubbo进行RPC调用的解决方案，消费者和提供者都需要引入Dubbo的SDK来完成远程调用。因为Dubbo本身是采用Java实现的，所以要求服务消费者和服务提供者也都必须采用Java实现。不过开源社区已经开始使用对核心扩展点进行TCK（Technology Compatibility Kit）提升框架的兼容性。它为用户增加一种扩展实现，只需通过TCK，即可确保与框架的其他部分兼容运行，可以有效提高健壮性，也方便第三方接入。下面是Dubbo的官方详细架构。



- 左边部分是服务消费者使用的接口，右边部分是服务提供者使用的接口，位于中轴线上的为双方都用到的接口。
- 从下至上分为十层，各层均为单向依赖，右边的黑色箭头代表层之间的依赖关系，每一层都可以剥离上层被复用，其中Service和Config层为API，其他各层均为SPI（Service Provider Interface）。
- 浅色小块为扩展接口，深色小块为实现类，图中只显示用于关联各层的实现类。
- 深色虚线为初始化过程，即启动时组装链，红色实线为方法调用过程，即运行时调时链，紫色三角箭头为继承（读者可到官网查看彩色图片），可以把子类看作父类的同一个节点，线上的文字为调用的方法。

**Dubbo服务调用过程**

Dubbo服务调用过程比较复杂，包含众多步骤，比如发送请求、编解码、服务降级、过滤器链处理、序列化、线程派发及响应请求等。下面我们重点分析请求的发送与接收、编解码、线程派发及响应的发送与接收等过程。Dubbo的服务调用过程如下图所示。




首先服务消费者通过代理对象Proxy发起远程调用，接着通过网络客户端Client将编码后的请求发送给服务提供者的网络层，也就是Server。Server在收到请求后，首先要做的事情是对数据包进行解码。然后将解码后的请求发送至分发器Dispatcher，再由分发器将请求派发到指定的线程池上，最后由线程池调用具体的服务。这就是一个远程调用请求的发送与接收过程。

**服务消费者发送请求**

Dubbo支持同步和异步两种调用方式，其中异步调用还可细分为“有返回值”的异步调用和“无返回值”的异步调用。所谓“无返回值”的异步调用是指服务消费者只管调用，但不关心调用结果，此时Dubbo会直接返回一个空的RpcResult。若要使用异步特性，需要服务消费者手动进行配置。默认情况下，Dubbo使用同步调用方式。服务调用的线程栈快照如下图所示。

```
proxy0#sayHello(String)
  -> InvokerInvocationHandler#invoke(Object, Method, Object[])
    -> MockClusterInvoker#invoke(Invocation)
      -> AbstractClusterInvoker#invoke(Invocation)
        -> FailoverClusterInvoker#doInvoke(Invocation, List<Invoker<T>>, LoadBalance)
          -> Filter#invoke(Invoker, Invocation)  // 包含多个 Filter 调用
            -> ListenerInvokerWrapper#invoke(Invocation)
              -> AbstractInvoker#invoke(Invocation)
                -> DubboInvoker#doInvoke(Invocation)
                  -> ReferenceCountExchangeClient#request(Object, int)
                    -> HeaderExchangeClient#request(Object, int)
                      -> HeaderExchangeChannel#request(Object, int)
                        -> AbstractPeer#send(Object)
                          -> AbstractClient#send(Object, boolean)
                            -> NettyChannel#send(Object, boolean)
                              -> NioClientSocketChannel#write(Object)
```

**服务提供者接收请求**

默认情况下，Dubbo使用Netty作为底层的通信框架。Netty首先会通过解码器对数据进行解码，并将解码后的数据传递给下一个处理器的指定方法。

解码器将数据包解析成Request对象后，NettyHandler的messageReceived方法紧接着会收到这个对象，并将这个对象继续向下传递。其间该对象会被依次传递给NettyServer、MultiMessageHandler、HeartbeatHandler以及AllChannelHandler处理。最后由AllChannelHandler将该对象封装到Runnable实现类对象中，并将Runnable放入线程池中执行后续的调用逻辑，调用栈如下图所示。

```
NettyHandler#messageReceived(ChannelHandlerContext, MessageEvent)
  -> AbstractPeer#received(Channel, Object)
    -> MultiMessageHandler#received(Channel, Object)
      -> HeartbeatHandler#received(Channel, Object)
        -> AllChannelHandler#received(Channel, Object)
          -> ExecutorService#execute(Runnable)     // 由线程池执行后续的调用逻辑
```

Dispatcher就是线程派发器。需要说明的是，Dispatcher真实的职责是创建具有线程派发能力的ChannelHandler，比如AllChannelHandler、MessageOnlyChannelHandler和ExecutionChannelHandler等，其本身并不具备线程派发能力。Dubbo的5种不同的线程派发策略如下表所示。

| 策略 | 用途 |
| --- | --- |
| all | 所有消息都派发到线程池，包括请求、响应、连接事件、断开事件等 |
| direct | 所有消息都不派发到线程池，全部在I/O线程上直接执行 |
| message | 只有请求和响应消息派发到线程池，其他消息均在I/O线程上执行 |
| execution | 只有请求消息派发到线程池，不含响应，其他消息均在I/O线程上执行 |
| connection | 在I/O线程上，将连接断开事件放入队列，有序地逐个执行，其他消息派发到线程池 |

默认配置下，Dubbo使用all派发策略，即将所有的消息都派发到线程池。请求对象会被封装在ChannelEventRunnable中，ChannelEventRunnable将会是服务调用过程的新起点。所以接下来我们看一下以ChannelEventRunnable为起点的服务提供者的线程调用栈，如下图所示。

```
ChannelEventRunnable#run()
  -> DecodeHandler#received(Channel, Object)
    -> HeaderExchangeHandler#received(Channel, Object)
      -> HeaderExchangeHandler#handleRequest(ExchangeChannel, Request)
        -> DubboProtocol.requestHandler#reply(ExchangeChannel, Object)
          -> Filter#invoke(Invoker, Invocation)
            -> AbstractProxyInvoker#invoke(Invocation)
              -> Wrapper0#invokeMethod(Object, String, Class[], Object[])
                -> DemoServiceImpl#sayHello(String)
```

**向用户线程传递调用结果**

响应数据解码完成后，Dubbo会将响应对象派发到线程池。要注意的是，线程池中的线程并非用户的调用线程，所以要想办法将响应对象从线程池传递到用户线程上。

用户线程在发送完请求后，调用DefaultFuture的get方法等待响应对象的到来。当响应对象到来后，用户线程会被唤醒，并通过调用编号获取属于自己的响应对象。

**Dubbo设计原理**

Dubbo在架构上通过SPI机制（SPI的全称为Service Provider Interface，SPI机制是一种服务发现机制）的设计，使得整体架构具备了极高的可扩展性。下面是Dubbo的核心设计原理：

- 采用Microkernel+Plugin模式，Microkernel负责组装Plugin，Dubbo自身的功能也是通过扩展点实现的，也就是Dubbo的所有功能点都可被用户自定义扩展所替换。
- 采用URL作为配置信息的统一格式，所有扩展点都通过传递URL携带配置信息。

SPI机制的本质是将接口实现类的全限定名配置在文件中，并由服务加载器读取配置文件，加载实现类，这样它可以在运行时动态为接口替换实现类。正因为此特性，我们可以通过SPI机制为程序提供拓展功能，这样可以在运行时动态为接口替换实现类。Dubbo就是通过SPI机制加载所有组件的，不过Dubbo并未使用Java原生的SPI机制，而是对其进行了增强，使其能够更好地满足需求。

**Dubbo SPI示例**

首先，我们定义一个接口，名称为Hello：

```
@SPI
public interface Hello{
```

```
    void sayHello();
}
//接下来定义两个实现类，分别为 TestA 和 TestB
public class TestA implements Hello{
    @Override
    public void sayHello() {
        System.out.println("Hello，  I am TestA .");
    }
}
public class TestB implements Hello{
    @Override
    public void sayHello() {
        System.out.println("Hello，  I am TestB .");
    }
}
```

Dubbo SPI的相关逻辑被封装在了ExtensionLoader类中，通过ExtensionLoader类我们可以加载指定的实现类。Dubbo SPI所需的配置文件需放置在META-INF/dubbo路径下，配置内容如下：

```
testA= com.spi.TestA
testB= com.spi.TestB
```

与Java SPI实现类配置不同，Dubbo SPI通过键值对的方式进行配置，我们可以按需加载指定的实现类。另外，在测试Dubbo SPI时，需要在Robot接口上标注@SPI注解。

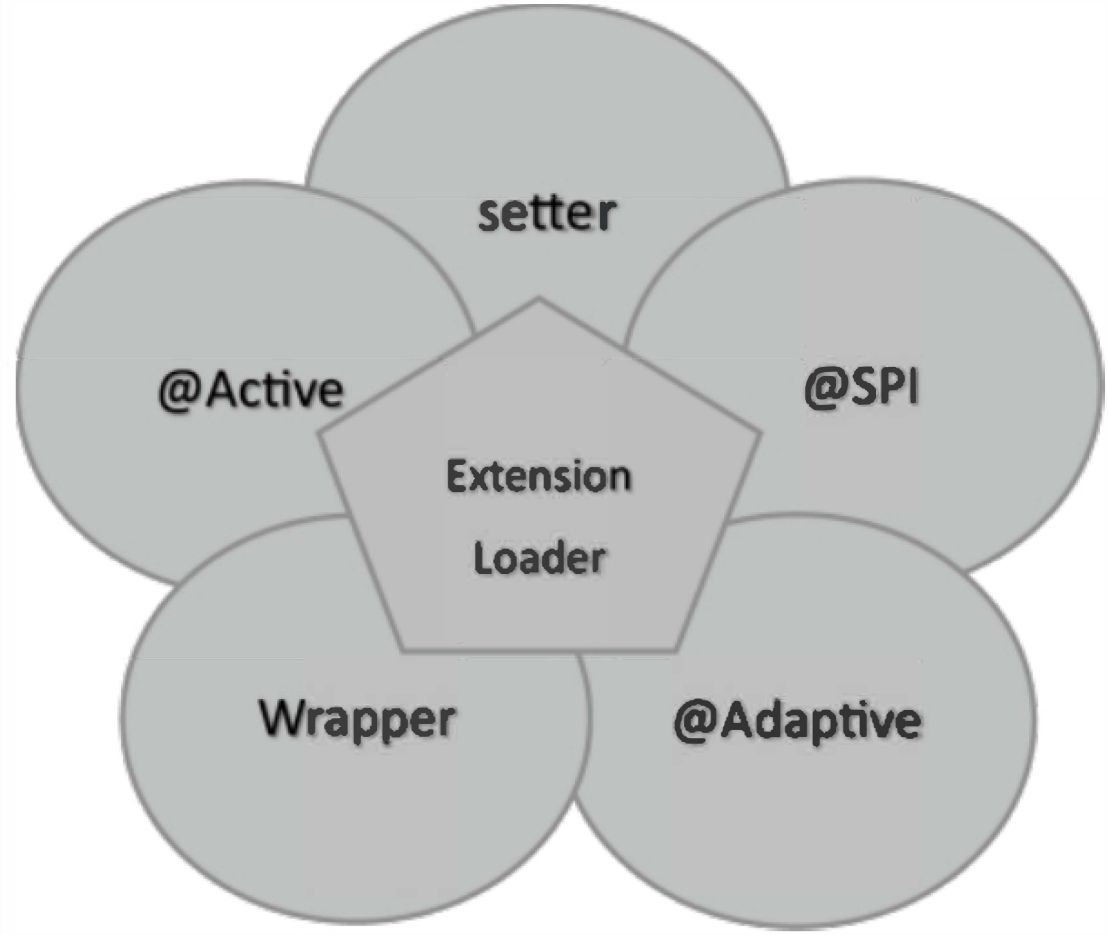
```
public class DubboSPITest
{
    @Test
    public void sayHello() throws Exception
    {
        ExtensionLoader<Hello> extensionLoader =
            ExtensionLoader.getExtensionLoader(Hello.class);
        Hello testa= extensionLoader.getExtension("testA");
        testa.sayHello();
        Hello testb= extensionLoader.getExtension("testB");
        testb.sayHello();
    }
}
```

上述代码的输出结果如下：

```
Hello，  I am TestA
Hello，  I am TestB
```

**SPI机制**

下面我们结合源码来理解Dubbo的SPI机制和整体架构特性，需要明确几个核心概念，如下图所示。




● ExtensionLoader

ExtensionLoader作为整个SPI机制的核心起着无可替代的作用，扩展点并不会强制所有用户都使用Dubbo提供的某些架构。例如Dubbo提供了ZooKeeper注册中心，但是如果我们更倾向于其他的注册中心，我们可以替换掉Dubbo提供的注册中心。我们称这种可被替换的技术实现点为扩展点，类似的扩展点有很多，例如Protocol、Filter、Loadbalance等。

鉴于ExtensionLoader的用法比较多，下面我们以ExtensionLoader类作为入口进行讲解。首先，我们通过ExtensionLoader的getExtensionLoader方法获取一个单例实例，然后通过ExtensionLoader的getExtension方法获取拓展类对象。其中，getExtensionLoader方法用于从缓存中获取与拓展类对应的ExtensionLoader实例，若缓存未命中，则创建一个新的实例。下面我们以ExtensionLoader的getExtension方法作为入口，代码如下：

```
public T getExtension(String name) {
    if (name == null || name.length() == 0)
        throw new IllegalArgumentException("Extension name == null");
    if ("true".equals(name)) {
        //获取默认的拓展实现类
        return getDefaultExtension();
    }
    //Holder，顾名思义，用于持有目标对象
    Holder<Object> holder = cachedInstances.get(name);
    if (holder == null) {
        cachedInstances.putIfAbsent(name,  new Holder<Object>());
        holder = cachedInstances.get(name);
    }
    Object instance = holder.get();
```

```
        //双重检查
        if (instance == null) {
            synchronized (holder) {
                instance = holder.get();
                if (instance == null) {
                    //创建拓展实例
                    instance = createExtension(name);
                    //将实例设置到 holder 中
                    holder.set(instance);
                }
            }
        }
        return (T) instance;
    }
```

上面代码的逻辑比较简单，首先检查缓存，缓存未命中则创建拓展对象。下面我们来看一下创建实例化对象的代码实现：

```
private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, (T) clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && wrapperClasses.size() > 0) {
            for (Class<?> wrapperClass : wrapperClasses) {
                instance = injectExtension((T)
                wrapperClass.getConstructor(type).newInstance(instance));
                }
            }
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance(
                name: " + name + ", class: " +type + ")
                 could not be instantiated: " + t.getMessage(), t);
        }
    }
```

代码 1#

createExtension模块中包含了如下步骤：

（1）通过getExtensionClasses获取所有的拓展类。

（2）通过反射创建拓展对象。

（3）向拓展对象中注入依赖。

（4）将拓展对象包裹在相应的Wrapper对象中。

我们在通过名称获取拓展类之前，需要根据配置文件解析出拓展项名称到拓展类的映射关系表（Map<名称，拓展类>），之后再根据拓展项名称从映射关系表中取出相应的拓展类即可。相关过程的代码如下：

```
private Map<String, Class<?>> getExtensionClasses() {
     Map<String, Class<?>> classes = cachedClasses.get();
     if (classes == null) {
         synchronized (cachedClasses) {
             classes = cachedClasses.get();
             if (classes == null) {
                 classes = loadExtensionClasses();
                 cachedClasses.set(classes);
             }
         }
     }
     return classes;
}
```

这里也是先检查缓存，若缓存未命中则通过synchronized加锁，加锁后再次检查缓存，并判空。此时如果classes仍空，则通过loadExtensionClasses加载拓展类。

下面分析loadExtensionClasses方法的逻辑：

```
//从配置文件中加载扩展点
private Map<String, Class<?>> loadExtensionClasses() {
    final SPI defaultAnnotation = type.getAnnotation(SPI.class);
    if(defaultAnnotation != null) {
        String value = defaultAnnotation.value();
        if(value != null && (value = value.trim()).length() > 0) {
            String[] names = NAME_SEPARATOR.split(value);
            if(names.length > 1) {
                throw new IllegalStateException("more than 1 default extension
                    name on extension " + type.getName()
                        + ": " + Arrays.toString(names));
            }
            if(names.length == 1) cachedDefaultName = names[0];
        }
    }
    Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
    loadFile(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
    loadFile(extensionClasses, DUBBO_DIRECTORY);
    loadFile(extensionClasses, SERVICES_DIRECTORY);
    return extensionClasses;
}
```

代码 2#

loadExtensionClasses方法总共做了两件事情，一是对SPI注解进行解析，二是调用loadDirectory方法加载指定目录中的配置文件。

SPI注解解析过程比较简单，loadDirectory方法先通过类加载器获取所有资源链接，然后通过loadResource方法加载资源。loadResource方法用于读取和解析配置文件，并通过反射加载类，最后调用loadClass方法进行其他操作。Dubbo会从以下三个路径读取并加载扩展点配置文件：

```
META-INF/services/
META-INF/dubbo/
META-INF/dubbo/internal/
```

● Wrapper

在实例化扩展点的代码中可以看到，在加载某个接口的扩展类时，如果某个实现中有一个拷贝类构造函数，那么该接口实现就是该接口的包装类，此时Dubbo会在真正的实现类上层包装上Wrapper。即这个时候从ExtensionLoader中返回的实际扩展类是被Wrapper包装的接口实现类。在上文代码的createExtension（String name）实例化扩展点中（代码1＃）可以看到相关代码实

现：将反射创建的instance实例作为参数传给Wrapper的构造方法，并通过反射创建Wrapper实例，而后在Wapper实例中注入依赖，最后将Wapper实例赋值给instance实例。

● Setter

Dubbo IoC通过setter方法注入依赖。Dubbo首先会通过反射获取实例的所有方法，然后遍历方法列表，检测方法名是否具有setter方法特征。若有这个特征则通过ObjectFactory获取依赖对象，最后通过反射调用setter方法将依赖设置到目标对象中。整个过程对应的注入扩展点代码如下：

```
private T injectExtension(T instance) {
    try {
       if (objectFactory != null) {
       for (Method method : instance.getClass().getMethods()) {
        if (method.getName().startsWith("set")
         && method.getParameterTypes().length == 1
         && Modifier.isPublic(method.getModifiers())) {
           Class<?> pt = method.getParameterTypes()[0];
           try {
             String property = method.getName().length() > 3 ?
                method.getName().substring(3, 4).toLowerCase()
                    + method.getName().substring(4) : "";
                    Object object = objectFactory.getExtension(pt, property);
                    if (object != null) {
                        method.invoke(instance, object);
                    }
                } catch (Exception e) {
                //省略
        }}}}
    }
    return instance;
}
```

扩展点实现类的成员如果为其他扩展点类型，ExtensionLoader会自动注入依赖的扩展点。ExtensionLoader通过扫描扩展点实现类的所有set方法来判定其成员。

● @SPI

在SPI代码实例中，Dubbo只有接口类使用了@SPI注解才会去加载扩展点实现，Dubbo本身重新实现了一套SPI机制，支持AOP与依赖注入，并且可以利用缓存提升加载实现类的性能，也支持实现类的灵活获取。下面是@SPI的定义：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface SPI {
    /**
     * default extension name
     * 设置默认扩展类
     */
    String value() default "";
}
```

在上文的loadExtensionClasses中（代码2＃）中，我们可以看到getExtensionLoader会对传入的接口进行校验，其中就会检验接口是否被@SPI注解，通过获取并缓存接口的@SPI注解上的默认实现类cacheDefaultExtensionName，再调用loadDirectory方法记载指定目录中的配置文件。源码实现如下：

```
private void cacheDefaultExtensionName() {
    final SPI defaultAnnotation = type.getAnnotation(SPI.class);
    if (defaultAnnotation == null) {
        return;
    }
    String value = defaultAnnotation.value();
    if ((value = value.trim()).length() > 0) {
    //检测@SPI 注解内容是否合法，如果不合法则抛出异常
    String[] names = NAME_SEPARATOR.split(value);
    if (names.length > 1) {
        throw new IllegalStateException("...");
      }
    //设置默认扩展类名称
    if (names.length == 1) {
          cachedDefaultName = names[0];
     }
     }
}
```

● @Adaptive

在Dubbo中，很多扩展都是通过SPI机制进行加载的，比如Protocol、Cluster、LoadBalance等。然而有些扩展并不想在框架启动阶段被加载，而是希望在扩展方法被调用时根据运行时参数进行加载。

在对自适应扩展生成过程进行深入分析之前，我们来看一下与自适应扩展息息相关的一个注解，即@Adaptive注解，该注解的定义如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE， ElementType.METHOD})
public @interface Adaptive {
    String[] value() default {};
}
```

@Adaptive可注解在类或方法上。当@Adaptive注解在类上时，Dubbo不会为该类生成代理类。当@Adaptive注解在方法（接口方法）上时，Dubbo则会为该方法生成代理逻辑。@Adaptive注解在类上的情况很少，在Dubbo中仅有两个类被@Adaptive注解了，分别是AdaptiveCompiler和AdaptiveExtensionFactory。

getAdaptiveExtension方法是获取自适应扩展的入口方法，相关代码如下：

```
public T getAdaptiveExtension() {
    //从缓存中获取自适应扩展
    Object instance = cachedAdaptiveInstance.get();
    if (instance == null) {
        if (createAdaptiveInstanceError == null) {
            synchronized (cachedAdaptiveInstance) {
                instance = cachedAdaptiveInstance.get();
                if (instance == null) {
```

```
                    try {
                        //创建自适应扩展
                        instance = createAdaptiveExtension();
                        //将自适应扩展设置到缓存中
                        cachedAdaptiveInstance.set(instance);
                    } catch (Throwable t) {
                        createAdaptiveInstanceError = t;
                        throw new IllegalStateException("fail to create
                            adaptive instance: ...");
                    }
                }
            }
        } else {
            throw new IllegalStateException("fail to create adaptive instance:  ...");
        }
    }
    return (T) instance;
}
```

getAdaptiveExtension方法首先会检查缓存，如果缓存未命中，则调用方法创建自适应扩展。下面我们看一下createAdaptiveExtension方法的代码：

```
private T createAdaptiveExtension() {
    try {
        //获取自适应扩展类，并通过反射实例化
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
        } catch (Exception e) {
        throw new IllegalStateException("
            Can not create adaptive extension ...");
    }
}
```

createAdaptiveExtension方法的代码包含了三个逻辑，分别如下：

- ○ 调用getAdaptiveExtensionClass方法获取自适应扩展Class对象。
- ○ 通过反射进行实例化。
- ○ 调用injectExtension方法向扩展实例中注入依赖。

- ● @Activate

@Activate注解表示一个扩展是否被激活，可以放在类定义和方法上，Dubbo将它用在SPI扩展类定义上，表示这个扩展实现的激活条件和时机。下面是代码示例：

```
@Activate(group = Constants.PROVIDER)
public class RpcServerInterceptor implements Filter{
    private final ServerRequestInterceptor serverRequestInterceptor;
    private final ServerResponseInterceptor serverResponseInterceptor;
    public RpcServerInterceptor () {
       //省略
    }
}
```

上述示例表示只有当group参数作为提供者时才会使RpcServerInterceptor拦截逻辑生效，这个注解的作用和Spring Boot中的@Condition注解类似。

## 6.3.4 Spring Cloud集成Dubbo

目前Dubbo在国内还是有较多公司在使用的，一方面是因为Dubbo作为阿里巴巴开源的一个SOA服务治理解决方案，在国内发展较早，有比较好的先发优势；另一方面是因为在国内很多工程

师对Dubbo框架都比较熟悉，有比较完善的文档介绍和实例；还有，Dubbo框架的性能优势和基于SPI的扩展机制也是Dubbo的优势所在。

然而，现在很多人也拿Dubbo与Spring Cloud做比较，其实Dubbo本质上是一个RPC框架，实现了SOA架构下的微服务治理，而Spring Cloud下有众多子项目，分别覆盖了微服务开发的各个方面，所以在一定程度上讲，Dubbo可以算是Spring Cloud的子集。

在Spring Cloud构建的微服务系统中，大多数开发者都使用官方提供的Feign组件来进行内部服务通信，这种声明式的HTTP客户端使用起来非常简洁、方便、优雅。但是在使用Feign消费服务的时候，相比Dubbo这种RPC框架而言，性能较低。所以基于Dubbo RPC方式的服务集成的交互方式也是Spring Cloud体系的一个重要补充。

**提供Dubbo服务**

下面通过一个简单的示例演示如何将Dubbo接入Spring Cloud。我们假设存在一个Dubbo RPC API，由服务提供者为服务消费者暴露接口：

```
public interface DemoService {
  String sayHello(String name);
}
```

首先，添加依赖：

```
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

然后，在application.yml中添加Dubbo的相关配置信息，示例配置如下：

```
spring.application.name: dubbo-spring-boot-starter
spring.dubbo.server: true
spring.dubbo.registry: N/A
```

接下来，在Spring Boot应用上添加@EnableDubboConfiguration，表示要开启Dubbo功能（Dubbo Provider服务可以使用或者不使用Web容器）。

```
@SpringBootApplication
@EnableDubboConfiguration
public class DubboProviderLauncher {
  //省略
}
```

编写你的Dubbo服务，只需要在要发布的服务上添加@Service（import com.alibaba.dubbo.config.annotation.Service）注解，其中interfaceClass属性表示要发布服务的接口声明。

```
@Service(interfaceClass = IHelloService.class)
Component
public class HelloServiceImpl implements IHelloService {
  //省略
}
```

启动你的Spring Boot应用，观察控制台，你可以看到Dubbo启动的相关信息。

**消费Dubbo服务**

首先，添加依赖：

```
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

其次，在application.properties中添加Dubbo的相关配置信息，示例如下：

```
spring.application.name: dubbo-spring-boot-starter
```

然后，开启@EnableDubboConfiguration：

```
@SpringBootApplication
@EnableDubboConfiguration
    public class DubboConsumerLauncher {
  //省略
}
```

最后，通过@Reference注入需要使用的interface：

```
@Componentpublic class HelloConsumer {
  @Reference(url = "dubbo://127.0.0.1:20880")
  private IHelloService iHelloService;
}
```

**Spring Boot与Dubbo集成**

上面的示例适用于新建项目，可以很方便地将Dubbo集成到Spring Boot应用，相比传统的Dubbo基于XML的配置方式，Spring Boot遵循“约定优于配置”理念，只需要加入几行注解就可以完成工作，而对于已经使用传统方式而非Spring Boot方式接入Dubbo框架实现的系统，如何通过增加一些代码就可以将Dubbo服务纳入Spring Cloud的体系是另外一个重要的课题。

● 思路一：将Dubbo服务的对外接口暴露为REST API

对于Dubbo服务提供者来说，可以通过@RestController封装服务端代码，对外暴露REST API。使用时，我们只需要在调用端的Service中注入InvokeRemoteService就可以像调用本地方法一样进行远程调用：

```
@RestController
public class TestRemoteCall {
  @Autowired
private InvokeRemoteService invokeRemoteService;
  public GroupVO findById(Integer id) {
     return invokeRemoteService.findByGroupId(id);
  }
}
```

对于Dubbo服务的消费者，你可以借助Spring Cloud中的Feign作为HTTP REST的调用接口，对于Dubbo服务，你可以向原来对外提供的Service interface类加入@FeignClient注解，支持外部调用，将对外暴露接口加上@RequestMapping或者@RestController注解，并且把接口改成REST风格的，代码如下：

```
@FeignClient(name = "test")
public interface InvokeRemoteService {
    @RequestMapping(value = "/group/{groupId}")
    GroupVO findByGroupId(@PathVariable("groupId") Integer adGroupId);
}
```

上面的代码中我们声明了一个HTTP“模板”，这个“模板”有一个方法声明findByGroupId，可以通过注解定义这个方法需要发起的HTTP请求信息（注解与Spring MVC完全相同）。

● 思路二：将Spring Cloud服务Dubbo化

这一改造的思路是替换Spring Cloud的Feign的底层调用协议，将原本使用HTTP Client的处理请求转交给Dubbo RPC来处理，同时将原本对外提供的REST API转换为Dubbo的服务，可以参考GitHub上的Dubbo开源项目（dubbo-spring-boot-project）。

首先，加入下面的Maven依赖：

```
<properties>
    <spring-boot.version>2.2.6.RELEASE</spring-boot.version>
    <dubbo.version>2.7.6</dubbo.version>
</properties>
<dependencyManagement>
    <dependencies>
        <!-- Spring Boot -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- Apache Dubbo  -->
        <dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-dependencies-bom</artifactId>
            <version>${dubbo.version}</version>
            <type>pom</type>
                <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <!-- Dubbo Spring Boot Starter -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>2.7.6</version>
    </dependency>
</dependencies>
```

然后，实现RPC接口定义：

```
public interface UserService {
    String selectByPrimaryKey( Long id);
}
```

服务端可以支持多协议发布服务：

```
@RestController
@Service(version = "1.0.0",  protocol = {"dubbo", "feign"},  timeout = 10000)
@RequestMapping(value = "/user/")
public class UserServiceImpl implements UserService {
    public User selectByPrimaryKey(@PathVariable("id") Long id) {
         return userDao.selectByPrimaryKey(id);
    }
}
```

接着，我们完成对消费端的实现：

```
@FeignClient(path = "/user")
@DubboClient(protocol = "feign",  value = @Reference(timeout = 10000,  version =
    "1.0.0"))
public interface UserService
 {
    @RequestMapping(value = "/{id}",  method = RequestMethod.GET)
    User selectByPrimaryKey(@PathVariable("id") Long id);
}
```

在application.properties中添加Dubbo的版本信息和客户端超时信息，向启动类添加@Enable-DubboConfiguration注解，这里我们配置的这些参数会在项目启动时被加载到DubboProperties类中。

```
spring:
  application:
    name: dubbo-demo-consumer
  dubbo:
    application:
      name: ${spring.application.name}
    protocol:
      name: feign
    registry:
      protocol: zookeeper
      address: localhost:2181
    scan: com.example
```

最后，实现Dubbo自动化配置：

```
@ConditionalOnProperty(prefix = “dubbo”， name = "enabled"， matchIfMissing = true)
@Configuration
@AutoConfigureAfter(DubboRelaxedBindingAutoConfiguration.class)
@EnableConfigurationProperties(DubboConfigurationProperties.class)
public class DubboAutoConfiguration {
    @ConditionalOnProperty(prefix = DUBBO_SCAN_PREFIX， name =
    BASE_PACKAGES_PROPERTY_NAME)
    @ConditionalOnBean(name = BASE_PACKAGES_PROPERTY_RESOLVER_BEAN_NAME)
    @Bean
    public ServiceAnnotationBeanPostProcessor serviceAnnotationBeanPostProcessor(
          @Qualifier(BASE_PACKAGES_PROPERTY_RESOLVER_BEAN_NAME)
           PropertyResolver propertyResolver) {
              Set<String> packagesToScan =
                  propertyResolver.getProperty(BASE_PACKAGES_PROPERTY_NAME，
                  Set.class， emptySet());
          return new ServiceAnnotationBeanPostProcessor(packagesToScan);
    }
}
```

上面我们实现了提供Dubbo的@Service注解服务。在DubboAutoConfiguration配置类中启动Bean，当配置文件中的前缀以“dubbo”开始时，会注入相关配置并完成初始化，然后获取所有加了@Service注解的类，使用反射生成代理类。当我们使用HTTP请求这些由@Service注解的类的方法时，它会将HTTP请求转换成Dubbo请求，调用这个代理类将调用结果返回。

### 6.3.5 Spring Cloud集成gRPC

gRPC本身的跨平台特性及性能上的优势都促使很多大公司采用gRPC的RPC解决方案作为微服务交互的标准交互集成方式。

到目前为止，Spring Cloud官方并没有支持gRPC，但是在GitHub上有非常多的第三方开源项目支持gRPC与Spring Cloud的集成，start数目最多的开源项目是grpc-spring-boot-starter。该项目也是Spring Cloud社区推荐的gRPC项目。下面是这个项目的主要特性：

- 在Spring Boot应用中，通过@GrpcService自动配置并运行一个嵌入式的gRPC服务。
- 使用@GrpcClient自动创建和管理gRPC通道（Channels）和桩代码（Stub）。
- 支持Spring Sleuth作为分布式链路跟踪解决方案。
- 支持全局和自定义的gRPC服务端/客户端拦截器。

- 支持Spring Security。
- 支持Metric（基于micrometer/actuator）。
- 适用于（non-shaded）grpc-netty。

**Spring Boot中gRPC的接入**

gRPC接入Spring Cloud主要分为三个工程模块，即服务定义模块、服务提供模块和服务消费模块。下面是接入gRPC的主要步骤。

1.服务定义

和其他RPC框架类似，gRPC需要做接口定义规范，默认情况下，会使用Protocal Buffers作为接口定义语言（IDL）。

首先，引入Maven依赖：

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>io.grpc:grpc-netty-shaded</artifactId>
    <version>${grpc.Version}</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>io.grpc:grpc-protobuf</artifactId>
    <version>${grpc.Version}</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>io.grpc:grpc-Stub</artifactId>
    <version>${grpc.Version}</version>
</dependency>
```

然后，编写一个.proto文件，定义好服务端的请求数据和响应数据，执行mvn clean install命令，protobuf-maven-plugin插件会根据.proto文件生成对应的Java代码，Maven的install命令会将接口工程打包上传到代码中央仓库，服务端和客户端可以通过Maven将远程中央仓库加载到本地并打包到各自的工程中。下面是IDL的定义（demo.proto）：

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "com.demo.grpc.helloworld";
option java_outer_classname = "HelloWorldProto";
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloResponse) {}
}
message HelloRequest {
    string jsonData= 1;
}
message HelloResponse {
    string jsonData= 1;
}
```

2.gRPC Server实现

首先，引入Maven依赖：

```
<dependency>
    <groupId>com.grpcDemo</groupId>
    <artifactId>grpc.lib.idl</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-spring-boot-starter</artifactId>
    <version>2.7.0.RELEASE</version>
</dependency>
```

其次，使用注解@GrpcService实现服务暴露：

```
@GrpcService
public class GrpcServerService extends SimpleGrpc.SimpleImplBase {
    @Override
    public void sayHello(HelloRequest req， StreamObserver<HelloReply> responseObserver)
    {
        HelloReply reply = HelloReply
        .newBuilder().setMessage("Hello ==> " + req.getName()).build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

然后，启动gRPC Server。默认情况下，gRPC Server会监听9090端口，也可以使用grpc.server.前缀自定义配置。

```
@EnableEurekaClient
@EnableDiscoveryClient
@SpringBootApplication
public class CloudGrpcServerApplication {
    public static void main(String[] args) {
    SpringApplication.run(CloudGrpcServerApplication.class， args);
    }
}
```

3.gRPC Client实现

首先，引入Maven依赖：

```
<dependency>
    <groupId>com.grpcDemo</groupId>
    <artifactId>grpc.lib.idl</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-client-spring-boot-starter</artifactId>
    <version>2.7.0.RELEASE</version>
</dependency>
```

其次，使用注解@GrpcClient（serverName）作为gRPC的桩代码（Stub）：

```
@Service
public class GrpcClientService {
    @GrpcClient("security-grpc-server")
    private SimpleBlockingStub simpleStub;
    public String sendMessage(final String name) {
        try {
            final HelloReply response =
            this.simpleStub.sayHello(HelloRequest
            .newBuilder().setName(name).build());
            return response.getMessage();
        } catch (final StatusRuntimeException e) {
            return "FAILED with " + e.getStatus().getCode().name();
        }
    }
}
```

然后，实现gRPC Client的RestController远程调用：

```
@RestController
public class GrpcClientController {
    @Value("${auth.username}")
    private String username;
    @Autowired
    private GrpcClientService grpcClientService;
    @RequestMapping(path = "/",  produces = MediaType.TEXT_PLAIN_VALUE)
    public String printMessage(@RequestParam(defaultValue = "test") final String name) {
        final StringBuilder sb = new StringBuilder();
        sb.append("Input:\n")
                .append("- name: " + name + " (Changeable via URL param ?name=X)\n")
                .append("Request-Context:\n")
                .append("- auth user: " + this.username
                + " (Configure via application.yml)\n")
                .append("Response:\n")
                .append(this.grpcClientService.sendMessage(name));
        return sb.toString();
    }
}
```

**gRPC的工作原理**

gRPC的工作原理是先通过IDL文件定义服务接口的参数和返回值类型，然后通过代码生成程序生成服务端和客户端的具体实现代码。

gRPC的主要特性包括三个方面。

（1）通信协议采用了HTTP 2，因为HTTP 2提供了连接复用、双向流、服务器推送、请求优先级、首部压缩等机制，所以在通信过程中可以节省带宽、降低TCP连接次数、节省CPU资源，尤其对于移动端应用来说，可以延长电池寿命。

（2）IDL使用了ProtoBuf，ProtoBuf是由Google开发的一种数据序列化协议，它的压缩和传输效率极高，语法也简单，所以被广泛应用在数据存储和通信协议上。

（3）多语言支持，能够基于多种语言自动生成对应语言的客户端和服务端代码。

**gRPC的核心概念**

- 基于服务定义：ProtoBuffer IDL

基于服务定义的思想，默认情况下gRPC使用ProtoBuffer作为IDL（接口定义语言）进行服务和消息的定义，示例代码如下：

```
Service HelloService{
    Rpc SayHello(HelloRequest) return(HelloResponse);
}
message HelloRequest{
    string greeting = 1;
}
Message HelloResponse{
    String reply = 1;
}
```

gRPC可以定义四种类型的服务方法。

- Unary RPC：客户端向服务端发送请求，并得到响应，类似于方法调用。

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

- Server streaming RPC：客户端可以向服务端发送请求，获取服务端返回的流响应，客户端可从流中读取一组消息，客户端可以持续读取消息直至消息全部读取完成，gRPC保证消息顺序的正确性。

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

- Client streaming RPC：客户端会写入一组消息，然后基于流的方式发送给服务端。当客户端写完全部消息后，就等待服务端进行消息的读取并等待服务端响应，gRPC保证消息顺序的正确性。

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

- Bidirectional streaming RPC：服务端和客户端都可以使用读写流发送一组消息。服务端的流和客户端的流是相互独立的，所以服务端和客户端可以按照自己的方式进行流的写入和读取。例如，服务端可以决定在全部接收完客户端发送的消息后再进行响应，或者它可以读取一条消息，就写入一条消息。同样，在流中的消息的顺序是可以保证的。

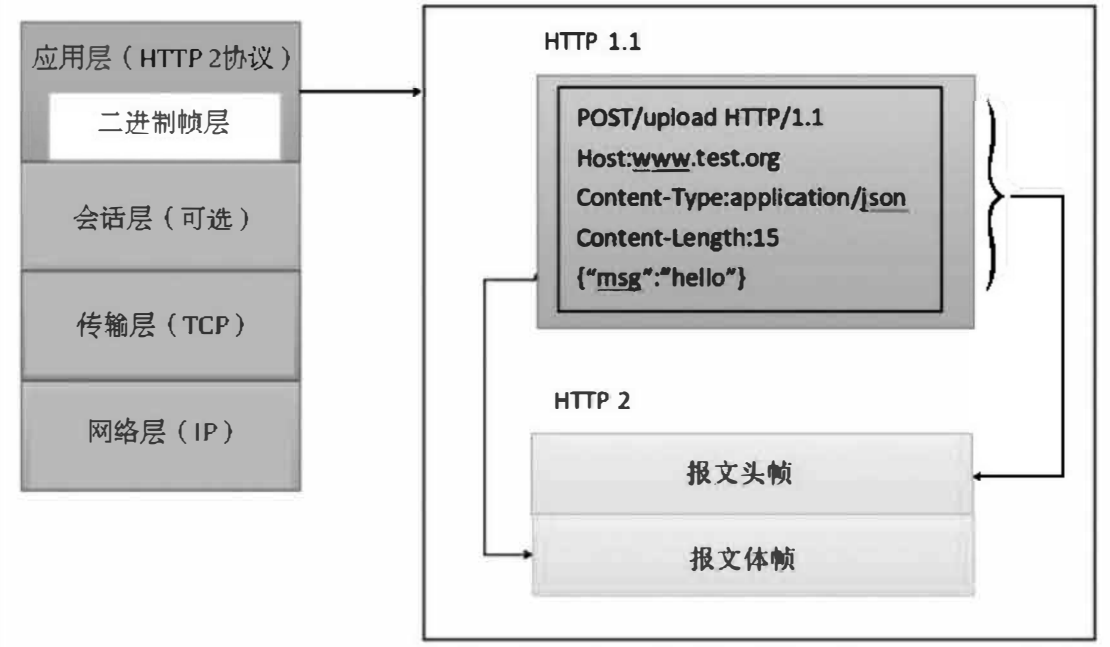
```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

● HTTP 2

HTTP 2通过Stream支持了连接的多路复用，提高了连接的利用率。Stream的重要特性如下：

- 一个连接可以包含多个Stream，多个Stream发送的数据互相不影响。
- Stream可以被客户端和服务端单方面或者共享使用。
- Stream可以被任意一端关闭。
- Stream会确定好发送Frame的顺序，另一端会按照接收到的顺序来处理。
- Stream用一个唯一ID来标识。

虽然看上去协议的格式和HTTP 1完全不同，实际上HTTP 2并没有改变HTTP 1的语义，只是把原来HTTP 1的Header和Body部分用Frame重新封装了一层而已，如下图所示。




HTTP 2的优势如下。

○ 连接共享：HTTP 2要解决的一大难题就是多路复用（MultiPlexing），即连接共享。

○ Header压缩：HTTP 2使用encoder来减少需要传输的Header大小，通信双方各自缓存（Cache）一份Header Fields表，既避免了重复Header的传输，又减小了需要传输Header的大小。高效的压缩算法可以大幅度压缩Header的大小，减少发送包的数量从而降低延迟。

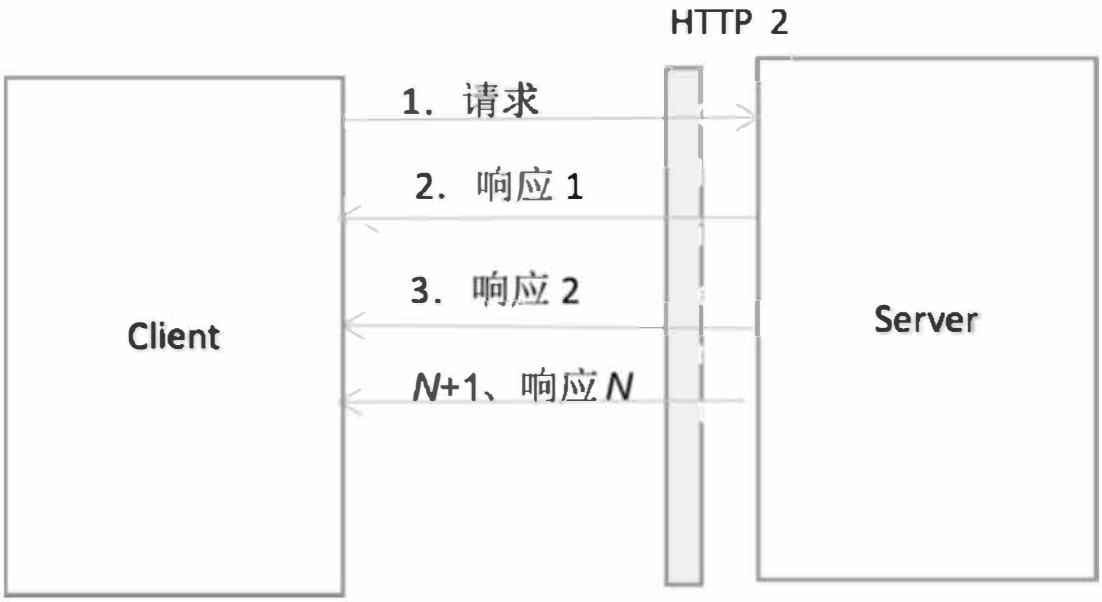
○ 重置连接表现更好：HTTP 2引入RST_STREAM类型的Frame，可以在不断开连接的前提下取消某个请求的Stream，表现更好。

○ 流量控制：每个HTTP 2流都拥有自己的公示的流量窗口，它可以限制另一端发送数据。对于每个Stream来说，两端都必须告诉对方自己还有足够的空间来处理新的数据，而在该窗口被扩大前，另一端只被允许发送那么多数据。

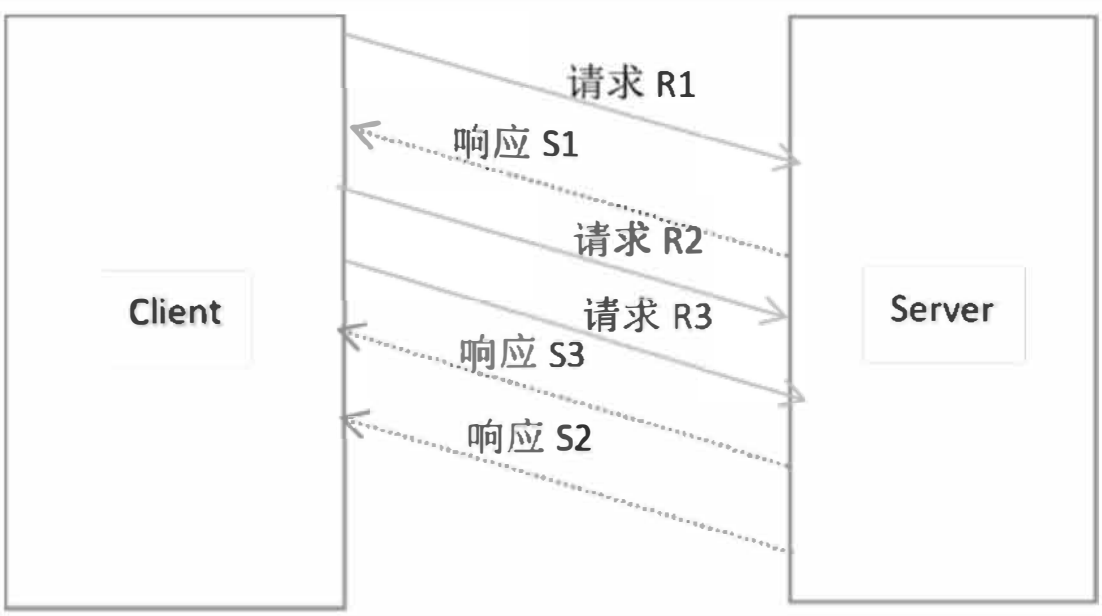
● 支持普通/流式RPC

○ 普通RPC调用：指客户端发送一个请求并获取一个响应。当客户端调用本地的桩方法时，服务端会得到一个RPC被调用的通知，通知中包含了关于此次调用的元数据信息（方法名、指定的合适的超时时间）。服务端可以立即返回一些它自己的初始化元数据，或者等待客户端的请求信息，当然这两种方式是和具体的应用相关的。当服务端接收到客户端的请求信息后，它会执行具体的逻辑以便产生一个响应。响应会和一个描述状态的详细信息及一个可选的附属元数据一起被发送给客户端。如果响应的状态是OK，则客户端就得到了响应，完成了一次RPC调用。

○ 服务端Streaming模式：指客户端发起1个请求，服务端返回 N 个响应，每个响应可以单独返回，它的原理如下图所示。

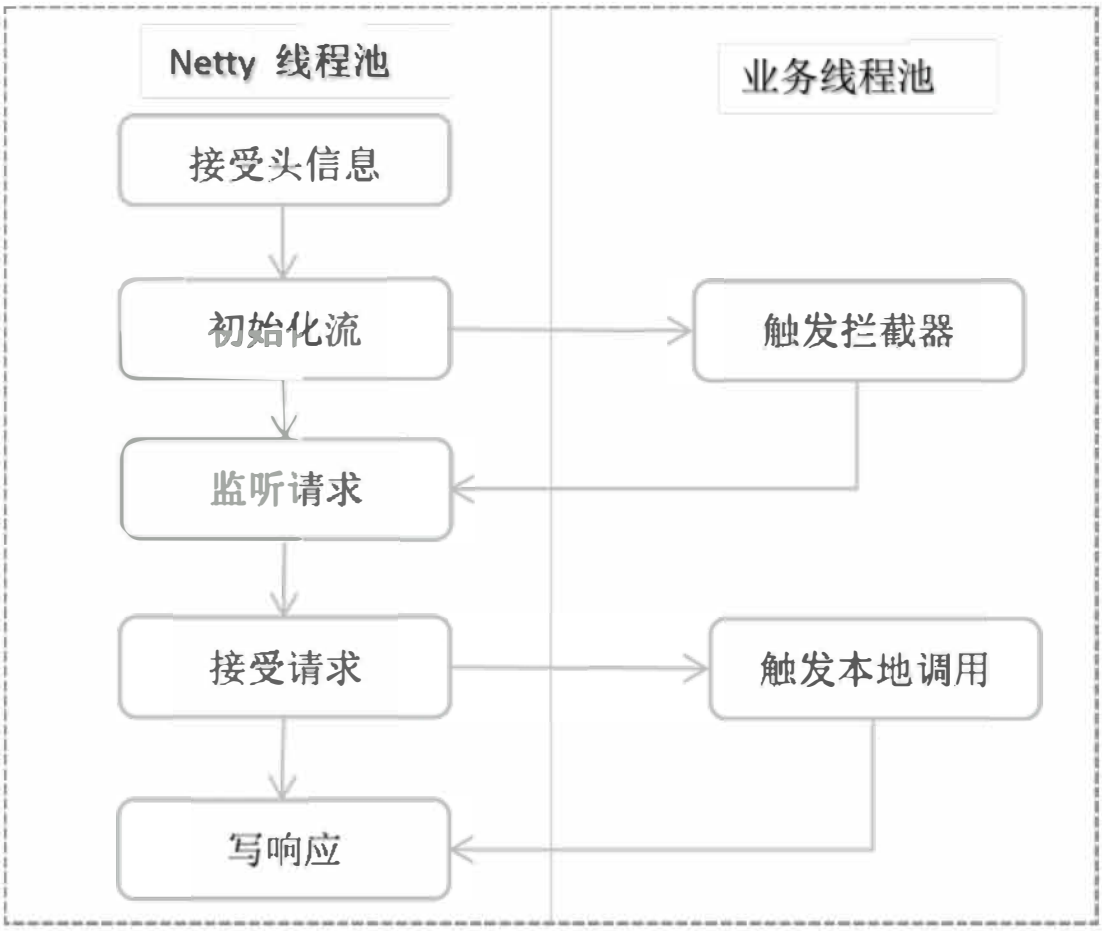


○ 双向流式RPC：客户端发送 N 个请求，服务端返回 N 个或者 M 个响应，利用该特性，可以充分利用HTTP 2的多路复用功能。在某个时刻，HTTP 2链路上可以既有请求也有响应，实现了全双工通信，示例如下图所示。




**gRPC服务调用解析过程**

gRPC的线程模型在Java实现中主要基于Netty底层网络通信框架，它遵循一个基本原则：除了传输过程中的监听及解包相关流程，其他的逻辑处理都会放在业务线程池中。比如序列化与反序列化、拦截器逻辑、本地方法调用。这个设计符合Netty的线程模型实践规范，最大化地保障传输框架的性能，提高服务资源的利用率。gRPC框架向业务层暴露了两个入口，一个是拦截器，在进入本地方法调用前拦截请求，用于处理一些前置逻辑；另一个就是本地服务。为了更清晰地表达业务线程池和Netty I/O线程池的分工，我们用下面的流程图来示意。




（1）NettyServer实例创建：gRPC服务端创建，首先需要初始化NettyServer，它是gRPC基于Netty 4.1和HTTP 2协议栈之上封装的HTTP 2服务端。

（2）NettyServerBuilder的buildTransportServer方法构建：NettyServer构建完成之后，监听指定的Socket地址。

（3）绑定IDL定义的服务接口实现类：gRPC与其他一些RPC框架的差异在于服务接口实现类的调用不是通过动态代理和反射机制，而是通过proto工具生成代码。在服务端启动时，将服务接口实现类实例注册到gRPC内部的服务注册中心上。请求消息接入之后，可以根据服务名和方法名，直接调用启动时注册的服务实例，性能更优。

（4）gRPC服务实例（ServerImpl）构建：ServerImpl负责整个gRPC服务端消息的调度和处理，在创建ServerImpl实例的过程中，会对服务端依赖的对象进行初始化。例如Netty的线程池资源、gRPC的线程池、内部的服务注册类（InternalHandlerRegistry）等。ServerImpl初始化完成之后，就可以调用NettyServer的start方法启动HTTP 2服务端，接收gRPC客户端的服务调用请求。

**grpc-spring-boot-starter源码解析**

grpc-spring-boot-stater的框架设计同样遵循脚手架一章中自定义Starter的方式，以便融合到Spring Boot和Spring Cloud体系。自定义Starter步骤如下。

1.自定义配置

```
# AutoConfiguration
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
net.devh.boot.grpc.server.autoconfigure.GrpcMetadataConsulConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcMetadataEurekaConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcMetadataNacosConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcServerAutoConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcServerFactoryAutoConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcServerSecurityAutoConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcServerMetricAutoConfiguration, \
net.devh.boot.grpc.server.autoconfigure.GrpcServerTraceAutoConfiguration
```

2.在配置文件中加载Bean并初始化

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Service
public @interface GrpcService {
    Class<? extends ServerInterceptor>[] interceptors() default {};
    String[] interceptorNames() default {};
    boolean sortInterceptors() default false;
}
```

3.配置Bean，初始化GrpcService服务

```
@Override
public Collection<GrpcServiceDefinition> findGrpcServices() {
    Collection<String> beanNames = Arrays.asList(this.applicationContext
        .getBeanNamesForAnnotation(GrpcService.class));
    List<GrpcServiceDefinition> definitions =
        Lists.newArrayListWithCapacity(beanNames.size());
    GlobalServerInterceptorRegistry globalServerInterceptorRegistry =
            applicationContext.getBean(GlobalServerInterceptorRegistry.class);
    for (String beanName : beanNames) {
           BindableService bindableService =
        this.applicationContext.getBean(beanName, BindableService.class);
           ServerServiceDefinition serviceDefinition = bindableService.bindService();
           GrpcService grpcServiceAnnotation =
               applicationContext.findAnnotationOnBean(beanName, GrpcService.class);
           serviceDefinition =
                   bindInterceptors(serviceDefinition, grpcServiceAnnotation,
               globalServerInterceptorRegistry);
           definitions.add(new GrpcServiceDefinition(beanName,
               bindableService.getClass(), serviceDefinition));
           log.debug("Found gRPC service: " +
               serviceDefinition.getServiceDescriptor().getName() + ",
               bean: "+ beanName + ", class: " +
               bindableService.getClass().getName());
       }
       return definitions;
}
```

4.配置GrpcServerLifecycle服务

```
@Configuration
@EnableConfigurationProperties
@ConditionalOnClass(Server.class)
@AutoConfigureAfter(GrpcCommonCodecAutoConfiguration.class)
public class GrpcServerAutoConfiguration {
    @ConditionalOnMissingBean
    @ConditionalOnBean(GrpcServerFactory.class)
    @Bean
    public GrpcServerLifecycle grpcServerLifecycle(final GrpcServerFactory factory)
    {
        return new GrpcServerLifecycle(factory);
    }
}
```

5.启动gRPC服务

```
@Slf4j
public class GrpcServerLifecycle implements SmartLifecycle {
    @Override
    public void start() {
        try {
            createAndStartGrpcServer();
        } catch (final IOException e) {
            throw new IllegalStateException("Failed to start the grpc server",  e);
        }
    }
}
```

## 6.4 MOM异步通信

在微服务架构中，使用REST和RPC的方式最大的问题就是请求/响应模式的通信模式可能导致服务之间调用的可用性降低，客户端与服务端需要同时在线，双方都需要知道对方的URL地址，或者服务消费者需要通过某种发现机制来定位服务实例的地址。

MOM（Message Oriented Middleware）是面向消息的中间件，使用消息提供者来协调消息传送操作。这种松耦合的通信机制有助于降低客户端和远程服务之间的依赖性。

### 6.4.1 消息中间件概述

消息中间件通常也被称为消息队列，是系统内部通信的核心机制。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列优势。

**消息中间件的常用概念**

- Broker：消息服务器，作为Server提供消息核心服务。
- Producer：消息生产者，业务的发起方，负责生产消息并传输给Broker。
- Consumer：消息消费者，业务的处理方，负责从Broker获取消息并进行业务逻辑处理。
- Topic：主题，它是发布订阅模式下的消息统一汇集地。不同生产者向Topic发送消息，由MQ服务器分发到不同的订阅者，实现消息的广播。
- Queue：队列，在点对点模式下，特定生产者向特定Queue发送消息，消费者订阅特定的Queue完成指定消息的接收。
- Message：消息体，根据不同通信协议定义的固定格式进行编码的数据包，可以用它来封装业务数据，实现消息的传输。

**消息中间件模式**

- 点对点模式：使用Queue作为通信载体，消息生产者生产消息后发送到Queue中，然后消息消费者从Queue中取出并且消费消息。
- 发布订阅（广播）模式：使用Topic作为通信载体。消息生产者（发布者）将消息发布到Topic中，同时有多个消息消费者（订阅者）消费该消息。和点对点方式不同，发布到Topic的消息会被所有订阅者消费。

**消息中间件流派**

当前，市面上流行的消息中间件有RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ等。它们有各自擅长的领域，分属不同的流派。下面我们通过不同消息队列的流派来了解不同消息中间件的世界观。

- 基于标准规范的消息队列

JMS（Java Message Service）是一套JMS API标准规范。JMS是由Sun公司提出的早期的消息标准，旨在为Java应用提供统一的消息操作，使用Java的世界观通过定制标准来达到统一规范的效果。JMS已经成为Java Enterprise Edition的一部分。从使用的角度看，JMS和JDBC扮演着差不多的角色，用户都可以根据相应的接口与实现了JMS的服务进行通信。

ActiveMQ是Apache的开源项目，是基于JMS规范开发的一个典型的消息队列，完全支持JMS1.1和J2EE1.4规范的JMS Provider实现。在中小型项目中用于解耦和异步操作时，可考虑ActiveMQ，简单易用，对队列数较多的情况支持不好。

- 基于标准协议的消息队列

RabbitMQ是这一流派的典型代表，它是使用Erlang编写的开源的消息队列，本身支持很多协议：AMQP、XMPP、SMTP、STOMP，也正因如此，使得它变得非常重量级，更适合于企业级的

开发。它同时实现了Broker架构，核心思想是生产者不会将消息直接发送给队列，消息在发送给客户端时先在中心队列排队。对路由、负载均衡、数据持久化都有很好的支持。

AMQP的全称为Advanced Message Queuing Protocol，它是一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端和中间件不同产品、不同开发语言等条件的限制。这种模式的消息中间件的主要优点是标准、可靠、通用。

在AMQP中，消息路由和JMS存在一些差别，在AMQP中增加了Exchange（交换机）和Binding（绑定）的角色。生产者将消息发送给Exchange，Binding决定Exchange的消息应该发送到哪个队列，而消费者直接从队列中消费消息。队列和Exchange的绑定关系由消费者来决定。AMQP逐渐成为消息队列的一个标准协议。

- 基于TCP的私有二进制的消息队列

这个流派最典型的代表就是Kafka了，Kafka遵循了一般消息队列的结构。它简化了消息队列功能模型，仅仅提供了一些最基础的消息队列相关功能，但是大幅度优化和提升了吞吐量。这个流派有一个Broker（消息代理）角色，也就是说，Kafka需要部署一套服务器集群，每台机器上都有一个Kafka Broker进程，这个进程负责接收请求、存储数据、发送数据。Kafka的生产消费采用简单的数据流模型。相比RabbitMQ，Kafka强调的不是提升性能和吞吐量，它关注的还是提供非常强大、复杂而且完善的消息路由功能。

RocketMQ也可以归为这个流派，RocketMQ是阿里巴巴参照Kafka设计思想使用Java实现的一套消息队列。

## 6.4.2 消息中间件的使用场景

在微服务架构中，基于消息中间件的交互方式可以解决同步请求/响应模式中服务高度耦合、服务交互灵活性脆弱、交互失败导致服务不可用等问题。

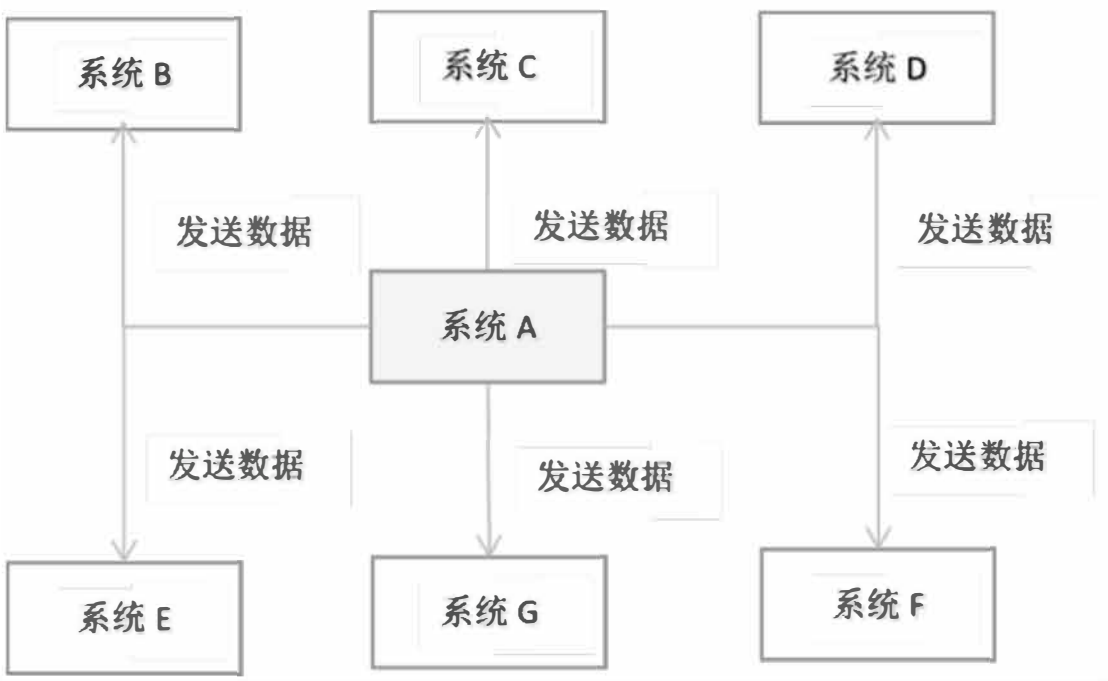
假设我们现在有两个微服务：权限管理服务和人力资源管理服务。下面使用不同集成方式将人力资源服务的数据库变动同步到权限管理服务中。

- 基于请求/响应模式的方案，权限管理服务提供REST方式调用人力资源管理服务提供的API。这种方式的问题是它需要二者必须同时在线，如果一方因为异常宕机，很容易导致数据丢失。另外，当人力资源数据信息很少变化时，权限管理服务每次同步数据都需要全量从人力资源服务中获得数据，效率比较低。所以，请求/响应模式对于服务之间的增量数据同步的场景并不友好，虽然也可以实现，但是这种做法增加了代码的逻辑复杂性。
- 基于共享存储的方案，人力资源管理服务可以将所有人力信息缓存到共享存储（如Redis）中，这样权限管理服务可以不用访问人力资源管理服务，而是访问Redis获得同步信息。这种模式的问题是，两个独立自治的服务会产生系统的紧耦合问题，人力资源管理服务对

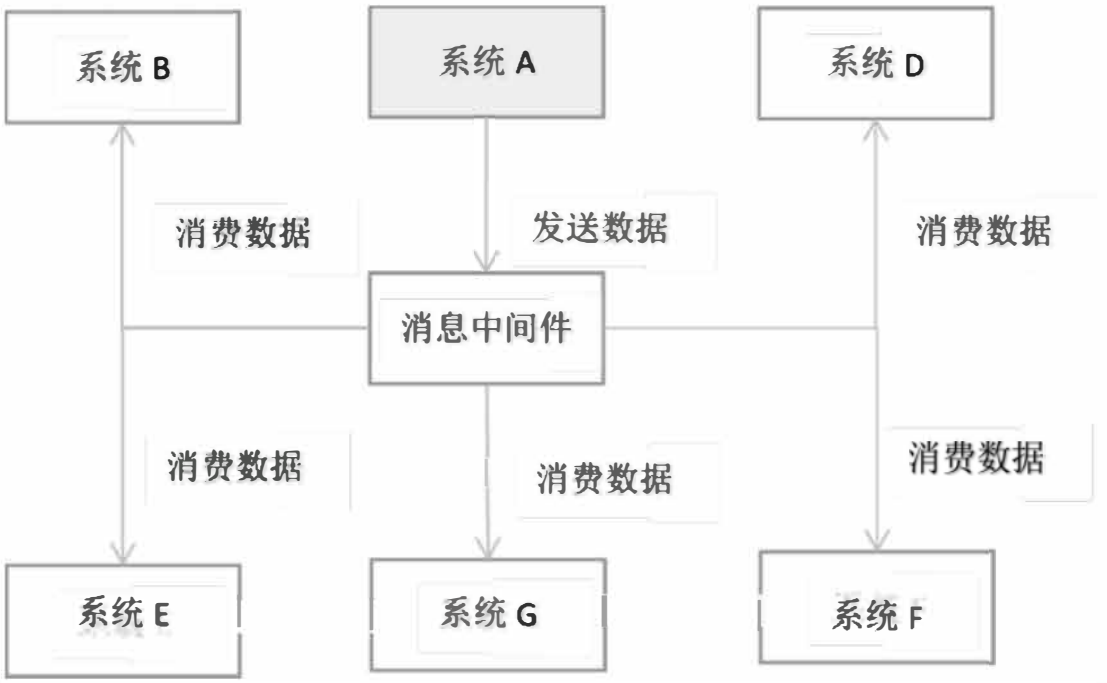
Redis的结构修改会对权限管理服务产生级联影响。另外从微服务数据自治的角度看，共享存储是不独立的，也是不易于扩展的。

- 另外一种方式就是采用消息传递的机制，在人力资源管理服务和权限管理服务之间建立一个消息队列，人力信息的变化通过事件的方式发送出去，人力资源管理服务只需要监听队列的变化，当监听到有新的事件后，就会“消费”这一事件，更新权限管理服务自己的缓存。这一模式可以带来诸多好处，概括为下面几点。
  - 系统解耦

    降低工程间的强依赖程度，针对异构系统进行适配。在项目启动之初预测项目将来会碰到什么需求是极其困难的，通过消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口，当应用发生变化时，可以独立地扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束，提供服务的可用性，如下图所示。



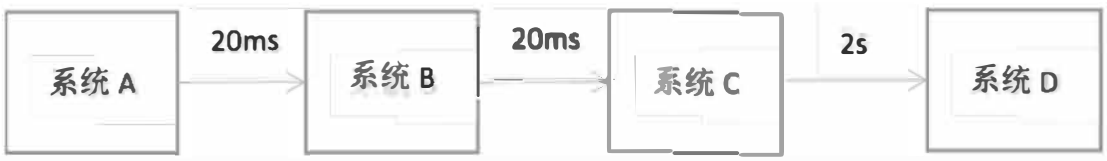

我们可以想象一下，假设有上百个系统都需要系统A的核心数据，一旦有系统加入，系统A就需要修改代码，将数据发送到新加入的系统。反之，如果有系统不再需要系统A发送数据，那么系统A又得修改代码不再向其发送数据。这样的架构设计耦合度太高了，我们可以引入消息中间件来实现系统之间的解耦。即核心系统A生产核心数据，然后将核心数据发送到消息中间件，下游消费系统根据自身的需求从消息中间件里获取消息进行消费，当不再需要数据时不获取消息即可，这样系统之间耦合度就大大降低了。改变后的系统架构如下图所示。



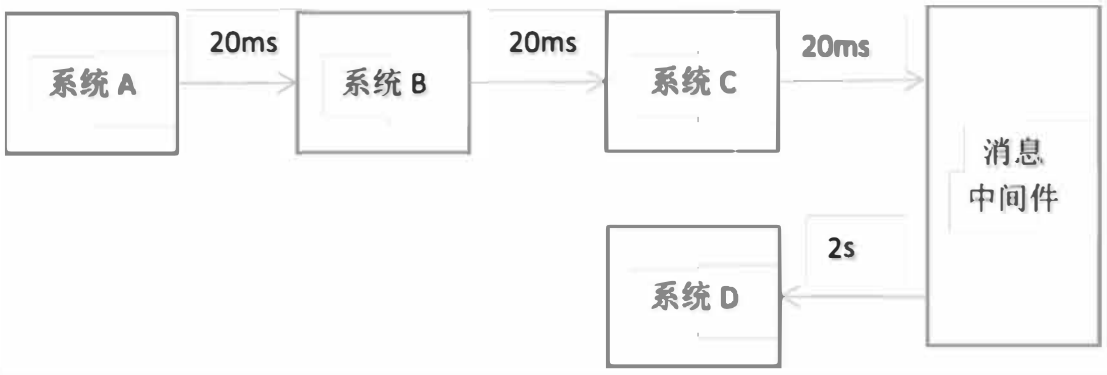

○ 异步调用

有些业务不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。可以将消息队列作为缓存，然后在需要的时候再去处理它们，这样就增加了系统交互的灵活性。

假设有一个系统调用为：系统A调用系统B耗时20ms，系统B调用系统C耗时20ms，而系统C调用系统D需要2s，这样下来整个调用链路需要耗时2040ms。但实际上系统A调用系统B、系统B调用系统C只需要40ms，而系统D的引入直接导致系统性能下降。此时我们应该考虑将系统D的调用抽离出来，做一个异步调用，如下图所示。

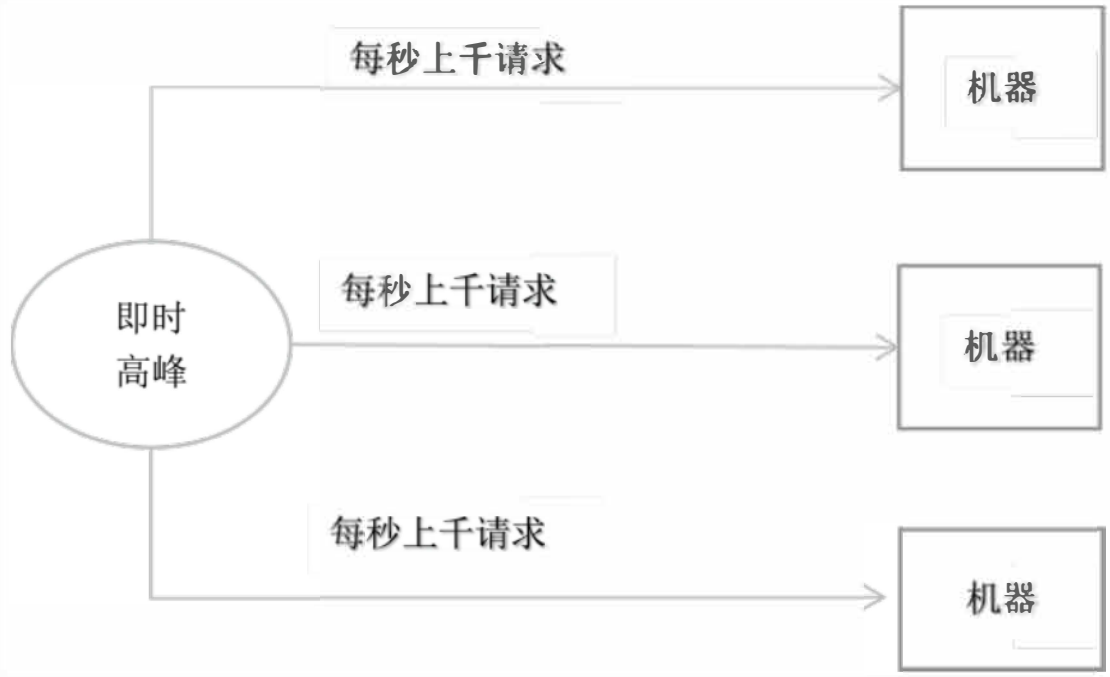



如果使用消息队列进行优化，系统A到系统B再到系统C就直接结束了，然后系统C再将消息发送到消息中间件，系统D从消息中间件里获取消息进行消费，这样系统的性能就提高了接近50倍，如下图所示。




○ 削峰填谷

假设有一个系统，正常时间每秒也就有几百个请求，部署在一个8核16GB的机器上，运行起来非常轻松。然而突然由于一个大促活动，高峰期请求数达到了几千，出现了瞬时流量高峰，此时最容易想到的是加机器，部署超过10台机器，也能扛住此时的高并发，如下图所示。




另外一种方式就是我们可以考虑引入消息中间件，进行流量削峰。一旦流量高峰到来，大量消息会堆积在消息队列里，机器只需要按照自己的最大负荷从消息队列里消费，等流量高峰过了，慢慢地队列里的消息也消费完毕了，这样就达到了削峰填谷的目的，如下图所示。

○ 持久化、冗余

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将丢失。消息队列可以把数据进行持久化，直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的“插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要处理系统明确地指出该消息已经被处理完毕，从而确保数据被安全地保存，直到使用完毕。

## 6.4.3 常用消息中间件

早期使用ActiveMQ作为消息中间件的项目比较多，作为Apache的一个子项目，ActiveMQ支持常用的多种语言：C++、Java、.Net、Python、PHP、Ruby等。

RabbitMQ是一个使用Erlang编写的AMQP（高级消息队列协议）的服务实现。简单来说，它就是一个功能强大的消息队列服务。

Kafka是最早使用Scala实现的一个高性能分布式Publish/Subscribe消息队列系统，具有快速持久化、高吞吐、离线堆积等特性，在实时计算、日志采集等场景和大数据领域已经成为业内的标准。

RocketMQ作为一款纯Java、分布式、队列模型的开源消息中间件，参考了优秀的开源消息中间件Kafka，支持事务消息、顺序消息、批量消息、定时消息、消息回溯等。目前，在金融公司和以消息可靠性、低延迟为主要诉求的场景下RabbitMQ使用得比较多；而Kafka则在日志、数据流转、大数据传输、高吞吐量等方面更有保障，后面我们还会对这两种典型的消息中间件做进一步介绍。

### 6.4.4 RabbitMQ消息中间件

RabbitMQ本身支持很多协议：AMQP、XMPP、SMTP、STOMP，也正因如此，它才变得非常重量级，更适合企业级的开发。它的核心思想是生产者不会将消息直接发送给队列，在发送给客户端时消息先在中心队列排队。它对路由、负载均衡、数据持久化都有很好的支持。

**RabbitMQ核心组件**

RabbitMQ最核心的组件是Exchange和Queue。Exchange和Queue部署在RabbitMQ Broker，而Producer和Consumer部署在应用端。

**Virtual Host**

Virtual Host（虚拟主机）类似权限控制组，一个Virtual Host里可以有多个Exchange和Queue，权限控制的最小粒度是Virtual Host。

**Producer：即数据的发送方（生产者）**

一般一个Message有两个部分：payload（有效载荷）和label（标签），payload顾名思义就是传输的数据，label是exchange的名字或者说是一个tag，它描述payload，而且RabbitMQ也通过label来决定把Message发给哪个Consumer。

**Consumer：即数据的接收方（消费者）**

如果有多个Consumer同时订阅同一个Queue中的消息，Queue中的消息会被分发给多个Consumer。

**Connection与Channel**

两者都是RabbitMQ对外提供的API中最基本的对象。Connection是一个TCP的连接，Producer和Consumer都是通过TCP连接到RabbitMQ Server的。

一个TCP建立后，客户端会建立一个Channel（通信信道），每个Channel都被指派成唯一的ID。Channel是我们与RabbitMQ打交道的最重要的接口，我们的大部分业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

**Queue**

Queue（队列）是RabbitMQ的内部对象，用来存储信息。应用程序在权限范围内可以自由地创建、共享、使用和消费Queue。Queue提供有限制的先进先出保证，服务器会将某一个Producer发出的同等优先级消息按照它们进入队列的顺序传递给某一个Consumer，Queue可以是持久的、临时的或者自动删除的。Queue可以保存在内存、硬盘或者两种介质的组合中，在Virtual Host范围之内，Queue保存消息，并将消息分发给一个或者多个订阅客户端。

**绑定**

所谓绑定就是将一个特定的Exchange与一个特定的Queue绑定起来，绑定的关键字是BindingKey。Exchange和Queue的绑定可以是多对多的关系，每个发送给Exchange的消息都有一个叫作RoutingKey的关键字，Exchange要将该消息转发给特定队列，该队列与交换器的BindingKey必须与消息的RoutingKey相匹配。

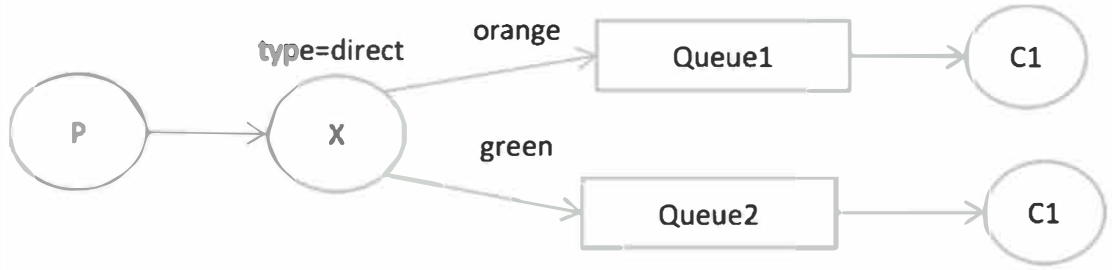
**Exchange（交换器）**

Exchange用于转发消息，但是它不会做存储，如果没有Queue绑定到Exchange，它会直接丢弃Producer发送过来的消息。这里有一个比较重要的概念：路由键（RoutingKey）。当消息发送到Exchange的时候，Exchange会将其转发到对应的队列中，至于转发到哪个队列取决于路由键。

Exchange的功能是接收消息并且转发到绑定的队列，Exchange不存储消息，在启用ACK模式后，Exchange找不到队列会返回错误。Exchange有四种类型：Direct、Fanout、Topic和Headers。

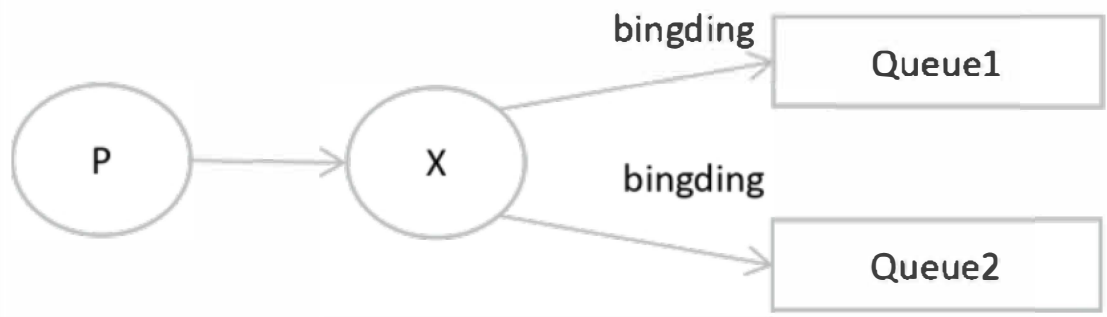
● Direct（默认）：直接交换器

这种方式类似于单播，Exchange会将消息发送给完全匹配RoutingKey的Queue。下图是直接交换器的工作方式：当某个消息到达交换器X时，如果它的RoutingKey是orange，它将直接被交给Queue1，如果是green，直接被交给Queue2。



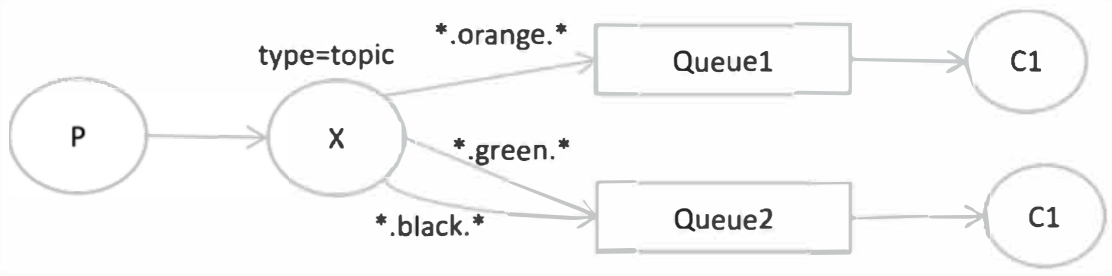

● Fanout：广播式交换器

不管消息的RoutingKey是什么，交换器都会将消息转发给所有绑定的Queue。广播式交换器的工作方式如下图所示：发送到交换器X的所有消息都被无条件地发到所有绑定的Queue上。




● Topic：主题交换器

这种方式类似于组播，交换器会将消息转发给和RoutingKey匹配模式相同的所有队列，比如RoutingKey为orange的Message会转发给绑定匹配模式为*.orange*和 # .orange. # 的队列（*表示是匹配一个任意词组， # 表示匹配0个或多个词组），如下图所示。




● Headers：消息体的Header匹配（ignore）

与Routing不同的地方是，Header模式取消了RoutingKey，使用Header中的Key/value（键/值对）进行匹配。

**订阅模式与检索模式**

RabbitMQ支持两种消息处理模式，一种是订阅模式（Push模式），由Broker主动将消息推送给订阅队列的Consumer；另一种是检索模式（Pull模式），需要Consumer调用channel.basicGet方法主动从队列中拉取消息。

● 订阅模式（Push）

实现一个Consumer，最容易的方式是继承DefaultConsumer类实现订阅模式，重写其中的方法即可，具体使用示例如下：

```
boolean autoAck = false;
channel.basicConsume(queueName， autoAck， "myConsumerTag"，
new DefaultConsumer(channel) {
    public void (String consumerTag，
             Envelope envelope，
             AMQP.BasicProperties properties，
             byte[] body)
        throws IOException
    {
        String routingKey = envelope.getRoutingKey();
        String contentType = properties.getContentType();
        long deliveryTag = envelope.getDeliveryTag();
        // (process the message components here ...)
        // channel.basicAck(deliveryTag， multiple);
        channel.basicAck(deliveryTag， false); //消费端确认成功
    }
});
```

这里因为关闭了消息自动确认机制，所以我们必须手动在handleDelivery方法中确认消息已经消费处理成功。

- 检索模式（Pull）

使用Channel.basicGet拉取消息，返回的数据类型是GetResponse实例。

```
boolean autoAck = false;
GetResponse response = channel.basicGet(queueName， autoAck);
if (response == null) {
} else {
   AMQP.BasicProperties props = response.getProps();
   byte[] body = response.getBody();
   long deliveryTag = response.getEnvelope().getDeliveryTag();
   //省略
   channel.basicAck(method.deliveryTag， false);
}
```

同样，由于这里设置了“autoAck=false；”，我们必须手动确认已经成功接收到了消息。

## 6.4.5 Kafka消息中间件

**Kafka简介**

Kafka是一种分布式的基于发布/订阅的消息系统。

- 以时间复杂度为 O （1）的方式提供消息持久化能力，并保证即使是TB级以上数据也能保证常数时间的访问性能。
- 高吞吐率，即使在非常廉价的商用机器上也能做到单机支持每秒10万条消息的传输。
- 支持Kafka Server间的消息分区及分布式消息消费，同时保证每个分区内的消息顺序传输。
- 支持离线数据处理和实时数据处理。

**Kafka中的关键概念和术语**

● Broker（代理）

Kafka集群包含一个或多个服务器，这种服务器被称为Broker。

● Topic（主题）

每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个Broker上，但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存储于何处）。

● Partition（分区）

Partition是物理上的概念，每个Topic包含一个或多个Partition，创建Topic时可指定Partition的数量。每个Partition对应一个文件夹，该文件夹下存储该Partition的数据和索引文件。

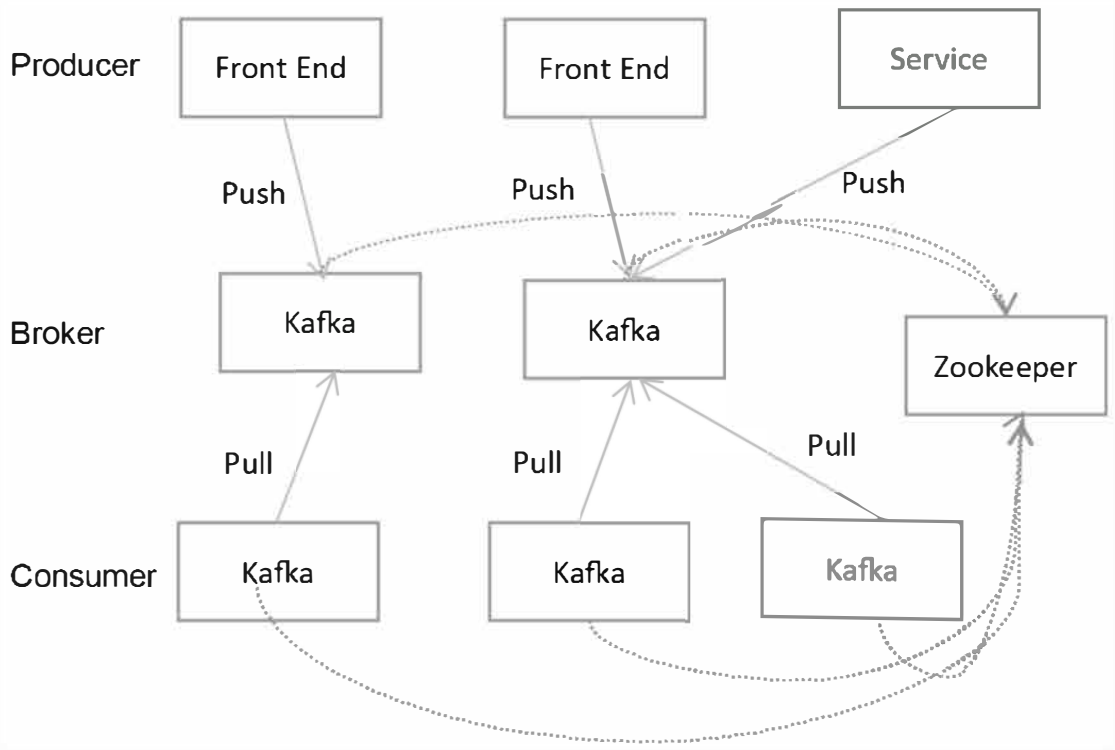
● Producer（生产者）

负责发布消息到Kafka Broker。

● Consumer（消费者）

从Kafka Broker上消费消息。

一个典型的Kafka集群中包含若干Producer（Web前端产生的数据、服务器日志、系统CPU、Memory指标数据等），若干Broker（Kafka支持水平扩展，一般Broker数量越多，集群吞吐率越高），若干ConsumerGroup（消费者集群），以及一个ZooKeeper集群。Kafka通过ZooKeeper管理集群配置，选举Leader，以及在ConsumerGroup发生变化时进行平衡。Producer使用Push模式将消息发布到Broker，Consumer使用Pull模式从Broker检索并消费消息，如下图所示。




**Kafka高性能设计**

Kafka通常的使用场景是高并发和高吞吐场景，当我们需要每秒处理几十万、上百万条消息时，相对其他消息队列实现，Kafka处理得更快。同时当具有百万或千万消费者时，在同等配置的

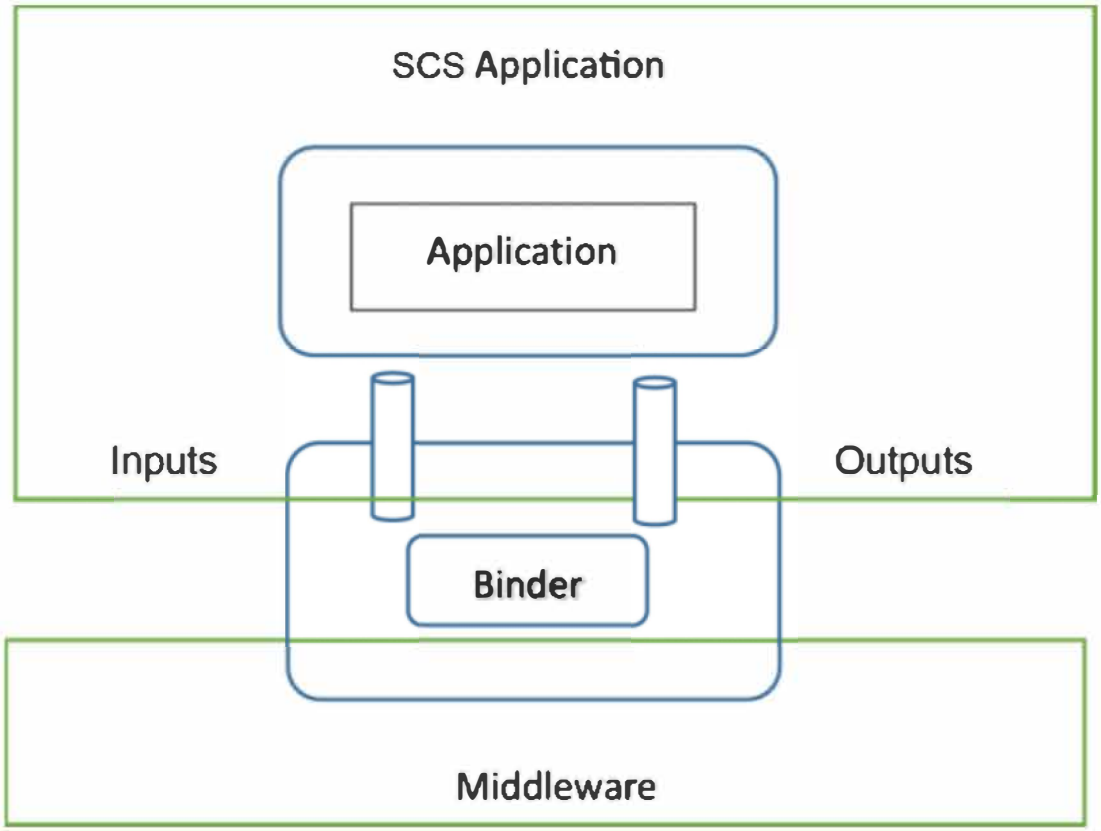
机器下，Kafka所拥有的Producer和Consumer会更多，读者可参考相关资料自学Kafka的高性能设计。

### 6.4.6 Spring Cloud Stream概述

Spring Cloud对Spring Cloud Stream（简称SCS）的定位是用于构建高度可扩展的基于事件驱动的微服务，其目的是简化消息在Spring Cloud应用程序中的开发。同时SCS能够提供一套灵活可扩展的编程模型，在Spring的基础上，支持发布/订阅模型、消费者分组、数据分片等。使用SCS能使微服务基于消息驱动的开发模式更加简单透明。

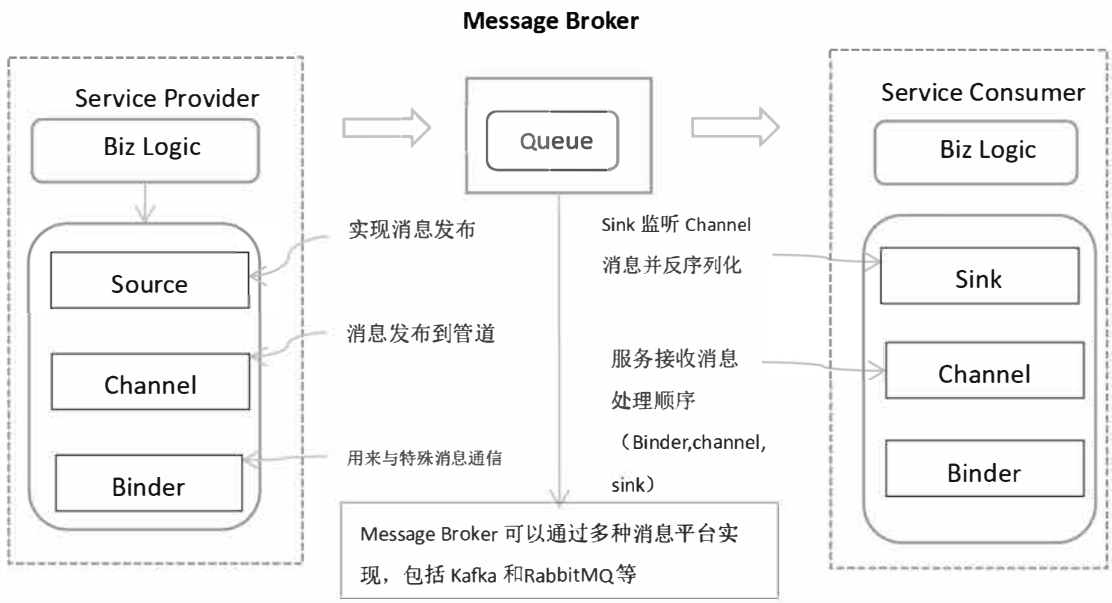
**SCS的架构**

SCS可以简单地理解为是对第三方消息中间件的一个概念封装，开发人员可以将关注点从消息中间件的特性配置转移到对消息的配置。下面是一个简单的架构图。




- Middleware：消息中间件，如RabbitMQ、Kafka、RocketMQ等。
- Binder：可以认为是适配器，用来将Stream与中间件连接起来，不同的Binder对应不同的中间件，需要我们配置。
- Application：由Stream封装的消息机制，很少自定义开发。
- Inputs：输入，可以自定义开发。
- Outputs：输出，可以自定义开发。

如果将SCS架构从消息层面做进一步细化，则可以分为三个模块，如下图所示。




**SCS的核心模块**

● Source

当服务发布消息前的前置业务完成后会通过Source将消息发布出去。Source是一个Spring注解接口，它可以将代表消息主体的POJO对象发布到Channel中，发布之前会把该消息对象序列化（默认使用JSON）。

● Channel

Channel（消息通道）是消息队列的进一步抽象，它会保存Producer发布的或者Consumer接收的消息。Channel名称一般与目标队列名称相关联。然而，消息队列的名称不会直接在代码中暴露，相反Channel名称会被用在代码中，所以只能在配置文件中配置，为Channel选取正确的消息队列进行读和写，而不是在代码中体现。

● Binder

Binder是SCS框架的一部分，它由SCS实现，用来与特殊的消息平台交互。我们可以在不暴露特殊消息平台的类库和API的情况下实现对消息的发布和消费。通过后面的源码介绍，你将会看到它的强大之处。

● Sink

在SCS中，当从消息队列接收到一条消息后，需要Sink。Sink能监听进入Channel中的消息并将消息反序列化成一个POJO对象。之后，消息就能给业务逻辑使用了。

**SCS的接入**

我们以RabbitMQ为例（消息队列的环境搭建这里不做过多的介绍，本章以Stream为主），新建两个Maven工程，分别作为消息消费者（Server-Receiver）和消息生产者（Server-Sender），在两个项目中引入Stream依赖和Stream对RabbitMQ的依赖，再为生产者单独添加Web依赖，以便能够通过HTTP调用发送信息。

1.接入Maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
```

2.消费者启动主类Server-Receiver

```
/**
 * @EnableBinding 表示告诉当前应用，增加消息通道的监听功能
 * 监听 Sink 类中名为 input 的输入通道
 */
@SpringBootApplication
@EnableBinding(Sink.class)
public class ReceiverApplication {
    public static void main(String[] args) {
        SpringApplication.run(ReceiverApplication.class， args);
    }
    /**
    *监听 RabbitMQ 的消息，具体什么队列、什么 Topic，
    *通过配置信息 application 获取
    * @param msg
    */
    @StreamListener(Sink.INPUT)
    public void reader(String msg) {
       Log.info("receiver: {}" ， msg);
    }
}
```

**注意：** 也可以自定义Sink接口，使用@EnableBinding（SinkDemo.class）的形式。下面是实现代码，自定义信道的名称为SinkDemo，Stream框架会创建出名为SinkDemo的Channel：

```
    public interface SinkDemo{
        String INPUT = "custom";
        @Input(SinkDemo.INPUT)
        SubscribableChannel input();
    }
```

3.添加消费者配置文件application.yml

```
spring:
  cloud:
    stream:
      bindings:
         input:
            destination: demotopic
            binder: defaultRabbit
      binders:
         defaultRabbit:
             type: rabbit
             environment:
                spring:
                 rabbitmq:
                     host: localhost
                     port: 5672
server:
  port: 8081
```

具体配置详解说明如下（spring.cloud.stream为前缀）：

- bindings配置
  - input：表示channelName，它是启动类中@EnableBinding（Sink.class）注解中配置的Sink接口，该接口中默认定义了channelName，当然我们也可以自己写Sink接口。
  - destination：消息中间件的Topic。
  - binder：当前bindings绑定的对应的适配器，该实例表示适配RabbitMQ，名称默认为defaultRabbit，可以自定义，接着需要配置该名称对应的类型、环境信息等。
- binders配置
  - defaultRabbit：binder适配器名称，和spring.cloud.stream.bindings.input.binder值一样。
  - environment：表示当前binder对应的配置信息。

4.生产者Server–Sender实现

为SenderApplication启动类添加@EnableBinding注解，实现代码如下：

```
/**
 * @EnableBinding(SenderSource.class) 表示监听 Stream 通道功能
 * SenderSource 为自定义的通道接口
 *
 */
@SpringBootApplication
@EnableBinding(SenderSource.class)
public class SenderApplication {
    public static void main(String[] args) {
        SpringApplication.run(SenderApplication.class, args);
    }
}
```

自定义SenderSource接口，以org.springframework.cloud.stream.messaging.Source源码为参考将Channel的名称改成和消费者的Sink的Channel名称一样。

```
public interface SenderSource {
    /**
     * Name of the output channel.
     */
    String OUTPUT = "input";
    /**
     * @return output channel
     */
    @Output(SenderSource.OUTPUT)
    MessageChannel output();
}
```

5.编写控制器，通过HTTP发送消息

```
@RestController
public class SenderController {
    @Autowired
    SenderSource source;
    @RequestMapping("/send")
    public String sender(String msg) {
        source.output().send(MessageBuilder.withPayload(msg).build());
        return "ok";
    }
}
```

6.添加生产者application.yml配置，配置方式和消费者的配置方式一样

```
spring:
  cloud:
    stream:
      bindings:
         input:
            destination: demotopic
            binder: defaultRabbit
      binders:
         defaultRabbit:
             type: rabbit
             environment:
                spring:
                 rabbitmq:
                     host: localhost
                     port: 5672
server:
  port: 8081
```

7.启动消费者和生产者

首先启动消费者，通过查看日志我们看到程序中声明了一个名称为demotopic.anonymous.88A97a5vQ9Ox07GnNBIKYQ的队列（SCS为我们建的临时队列名称），并且绑定了mytopic主题，创建了一个连上消息队列的连接，下面是部分关键日志输出：

```
Registering MessageChannel input
Registering MessageChannel nullChannel
Registering MessageChannel errorChannel
Registering MessageHandler errorLogger
declaring queue for inbound: demotopic.anonymous.E7egCa5vQ9Ox07, bound to: demotopic
Attempting to connect to: [localhost:5672]
```

然后启动生产者Server-Sender，在启动日志中我们也看到应用创建了到对应的消息队列的连接。接下来我们通过HTTP发送信息：

```
http://localhost:8081/send/?msg=微服务架构进阶
```

在服务消费者的日志中，监听到了对应的消息：

```
receiver {微服务架构进阶}:
```

## 6.4.7 Stream源码解析

Spring Cloud Stream（简称SCS）提供了一系列预先定义的注解来声明输入型和输出型Channel，业务系统基于这些Channel与消息中间件进行通信，而不是直接与具体的消息中间件进行通信。跟踪SCS的源码就会发现，Stream有很多外部依赖，最主要的就是Messaging和Integration两个项目，所以在讲解SCS源码前，有必要先介绍一下Messaging和Integration与SCS体系的关系。

SCS的目标是建立一套统一的基于注解的消息发送机制，屏蔽开发人员直接与底层消息系统进行细节交互，而Messaging模块正是Spring框架中用来做统一消息编程模型的，在Messaging中最关键的数据结构是Message，代码如下：

```
public interface Message<T> {
    T getPayload();
    MessageHeaders getHeaders();
}
```

在Messaging模块中消息通道MessageChannel是一个接口类，用于发送Message消息，可以理解为Messaging模块中的标准接口，类似于J2EE中的Servlet接口，具体实现类可以实现具体消息通道。下面是MessageChannel的代码：

```
public interface MessageChannel {
    /**
    * Constant for sending a message without a prescribed timeout.
    */
    long INDEFINITE_TIMEOUT = -1;
     boolean send(Message<?> message);
     boolean send(Message<?> message， long timeout);
}
```

在Messaging模块中，消息通道的子接口SubscribableChannel继承了MessageHandler消息处理器：

```
pblic interface SubscribableChannel extends MessageChannel{
    boolean subscribe(MessageHandler handler);
    boolean unsubscribe(MessageHandler handler);
 }
```

由MessageHandler真正地消费/处理消息：

```
@FunctionalInterface
public interface MessageHandler {
    void handleMessage(Message<?> message) throws MessagingException;
}
```

Integration基于Spring框架可以实现轻量级的消息传递，也是对Messaging的扩展实现，支持通过声明适配器与SCS集成。它实现了消息过滤、消息转换、消息聚合和消息分割等功能，提供了对 MessageChannel 和 MessageHandler 的实现，包括 DirectChannel 、ExecutorChannel 、PublishSubscribeChannel ，以及 MessageFilter 、ServiceActivatingHandler 、MethodInvokingSplitter 等。下面介绍 Integration 中的两种消息分发器： DirectChannel 和 PublishSubscribeChannel。

```
public class DirectChannel extends AbstractSubscribableChannel {
    private final UnicastingDispatcher dispatcher = new UnicastingDispatcher();
    //省略
}
public class PublishSubscribeChannel extends AbstractExecutorChannel {
    public PublishSubscribeChannel(Executor executor) {
        super(executor);
        this.dispatcher = new BroadcastingDispatcher(executor);
    }
    //省略
}
```

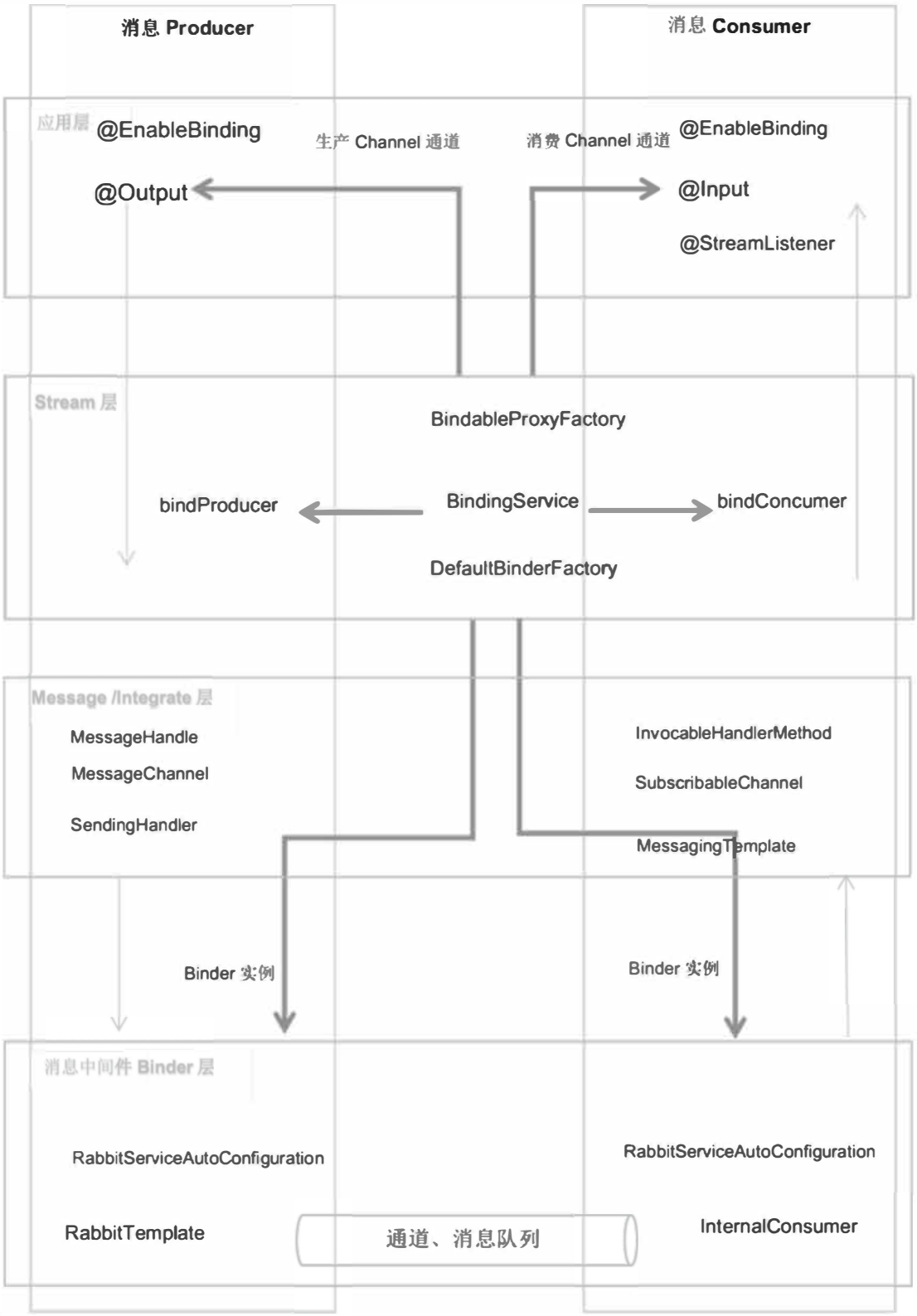
从代码可知，DirectChannel内部的UnicastingDispatcher类型分发器会发到对应消息通道的MessageChannel中，从名字也可以看出来，UnicastingDispatcher是一个单播的分发器，只能选

择一个消息通道。而PublishSubscribeChannel使用BroadcastingDispatcher作为广播消息分发器，会把消息分发给所有的MessageHandler。

SCS在Integration的集成上进行了封装，通过注解的方式和统一的API进行消息的发送和消费，底层消息中间件的实现细节由各个消息中间件的Binder完成，同时，通过与Spring Boot的Externalized Configuration整合，SCS提供了BindingProperties等外部化配置类，这些具体的配置信息将绑定到具体的消息中间件的配置类中。

**SCS的架构流程图**

下面是SCS的架构流程图，我们会从几个层次分别讲解其中相关联的源码和它们之间的交互关系。




## 应用层

SCS为用户提供了三个绑定消息通道的默认实现。

- Sink：通过指定消费消息的目标来标识消息消费者。
- Source：与Sink相反，用于标识消息生产者。
- Processor：集成了Sink和Source的功能，用于标识消息生产者和消费者。

```
public interface Source {
    String OUTPUT = "output";
    @Output(Source.OUTPUT)
    MessageChannel output();
}
public interface Sink {
    String INPUT = "input";
    @Input(Sink.INPUT)
    SubscribableChannel input();
}
public interface Processor extends Source， Sink {
}
```

对应用而言，想要启动SCS的功能，需要先启动注解。@EnableBinding注解是Stream框架运转的起点，通过这个注解可以实现动态注册BeanDefinition，它会将消息通道绑定到自己修饰的目标实例上，从而让这些实例具备与消息队列进行交互的能力。下面我们看源码：

```
@Target({ ElementType.TYPE， ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@Import({BindingServiceConfiguration.class，BindingBeansRegistrar.class，
    BinderFactoryConfiguration.class，SpelExpressionConverterConfiguration.class })
@EnableIntegration
public @interface EnableBinding {
    /**
     * A list of interfaces having methods annotated with {@link Input} and/or
     * {@link Output} to indicate binding targets.
     */
    Class<?>[] value() default {};
}
```

- BindingServiceConfiguration的作用是完成BindingService、InputBindingLifecycle、OutputBindingLifecycle等重要Bean的初始化及相关配置文件加载。
- BindingBeansRegistrar的作用是注册声明通道的接口类的BeanDefinition，从而获取这些接口类的实例，并使用这些实例进行消息的发送和接收，具体代码实现如下：

```
public class BindingBeansRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata，
        BeanDefinitionRegistry registry) {
        AnnotationAttributes attrs =
            AnnotatedElementUtils.getMergedAnnotationAttributes(
            ClassUtils.resolveClassName(metadata.getClassName()， null)，
        EnableBinding.class);
        for (Class<?> type : collectClasses(attrs， metadata.getClassName())) {
            if (!registry.containsBeanDefinition(type.getName())) {
                BindingBeanDefinitionRegistryUtils.
                    registerBindingTargetBeanDefinitions(type，type.getName()， registry);
                BindingBeanDefinitionRegistryUtils.
                    registerBindingTargetsQualifiedBeanDefinitions(
                ClassUtils.resolveClassName
                 (metadata.getClassName()， null)， type，registry);
            }
        }
    }
    private Class<?>[] collectClasses(AnnotationAttributes attrs， String className) {
        EnableBinding enableBinding = AnnotationUtils.synthesizeAnnotation(attrs，
        EnableBinding.class， ClassUtils.resolveClassName(className， null));
            return enableBinding.value();
    }
}
```

registerBindingTargetBeanDefinitions方法会调用ReflectionUtils类完成扫描所有被注解@Input和@Output标注了的方法，然后注册BeanDefinition。下面是代码示例：

```
public static void registerBindingTargetBeanDefinitions(Class<?> type,
        final String bindingTargetInterfaceBeanName,
             final BeanDefinitionRegistry registry) {
    ReflectionUtils.doWithMethods(type, new MethodCallback() {
    @Override
    public void doWith(Method method) throws IllegalArgumentException,
           IllegalAccessException {
        Input input = AnnotationUtils.findAnnotation(method, Input.class);
        if (input != null) {
               String name = getBindingTargetName(input, method);
               registerInputBindingTargetBeanDefinition(input.value(),
                   name, bindingTargetInterfaceBeanName,
                    method.getName(), registry);
           }
           Output output = AnnotationUtils.findAnnotation(method, Output.class);
           if (output != null) {
               String name = getBindingTargetName(output, method);
               registerOutputBindingTargetBeanDefinition(output.value(),
                   name, bindingTargetInterfaceBeanName,
               method.getName(), registry);
           }
       }
    });
}
```

registerBindingTargetsQualifiedBeanDefinitions是在注册registerBindingTargetBeanDefinitions时使用的工厂类BeanDefinition，这个工厂类用来生成registerBindingTargetBeanDefinition注册的Bean实例，如下所示：

```
public static void registerBindingTargetsQualifiedBeanDefinitions(Class<?>
       parent, Class<?> type, final BeanDefinitionRegistry registry)
{
    if (type.isInterface()) {
        ootBeanDefinition rootBeanDefinition = new
            RootBeanDefinition(BindableProxyFactory.class);
        rootBeanDefinition.addQualifier(new
            AutowireCandidateQualifier(Bindings.class, parent));
        rootBeanDefinition.getConstructorArgumentValues().
            addGenericArgumentValue(type);
        registry.registerBeanDefinition(type.getName(), rootBeanDefinition);
    }
    else {
        RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(type);
            rootBeanDefinition.addQualifier(new
                AutowireCandidateQualifier(Bindings.class, parent));
        registry.registerBeanDefinition(type.getName(), rootBeanDefinition);
    }
}
```

**Stream层**

Stream层的BindableProxyFactory被初始化为一个rootBeanDefinition，并注册为一个FactoryBean，这样Spring容器就可以获得registerBindingTargetBeanDefinitions方法中所注册的Bean实例（MessageChannel对象实例）。BindableProxyFactory可以说是SCS实现通道接口类声明及相关类型的核心类，代码如下：

```
public class BindableProxyFactory implements
    MethodInterceptor, FactoryBean<Object>, Bindable, InitializingBean {
    @Override
    public synchronized Object invoke(MethodInvocation invocation) throws Throwable {
        Method method = invocation.getMethod();
        Object boundTarget = targetCache.get(method);
        if (boundTarget != null) {
            return boundTarget;
        }
        Input input = AnnotationUtils.findAnnotation(method, Input.class);
        if (input != null) {
            String name = BindingBeanDefinitionRegistryUtils.
                getBindingTargetName(input, method);
            boundTarget = this.inputHolders.get(name).getBoundTarget();
            targetCache.put(method, boundTarget);
            return boundTarget;
        }
        else {
            Output output = AnnotationUtils.findAnnotation(method, Output.class);
            if (output != null) {
                String name = BindingBeanDefinitionRegistryUtils.
                    getBindingTargetName(output, method);
                boundTarget = this.outputHolders.get(name).getBoundTarget();
                targetCache.put(method, boundTarget);
                return boundTarget;
            }
        }
        return null;
    }
@Override
public void afterPropertiesSet() throws Exception {
    Assert.notEmpty(BindableProxyFactory.this.bindingTargetFactories,
            "'bindingTargetFactories' cannot be empty");
    ReflectionUtils.doWithMethods(this.type, new ReflectionUtils.MethodCallback() {
        @Override
        public void doWith(Method method) throws IllegalArgumentException {
            Input input = AnnotationUtils.findAnnotation(method, Input.class);
            if (input != null) {
            String name = BindingBeanDefinitionRegistryUtils
                .getBindingTargetName(input, method);
            Class<?> returnType = method.getReturnType();
            Object sharedBindingTarget = locateSharedBindingTarget(name, returnType);
            if (sharedBindingTarget != null) {
                BindableProxyFactory.this.inputHolders.put(name,
                    new BoundTargetHolder(sharedBindingTarget, false));
            }
            else {
                    BindableProxyFactory.this.inputHolders.put(name,
                        new BoundTargetHolder
```

```
                        (getBindingTargetFactory(returnType).createInput(name), true));
                }
            }
        }
    });
    ReflectionUtils.doWithMethods(this.type, new ReflectionUtils.MethodCallback() {
        @Override
        public void doWith(Method method) throws IllegalArgumentException {
            Output output = AnnotationUtils.findAnnotation(method, Output.class);
            if (output != null) {
                String name = BindingBeanDefinitionRegistryUtils
                .getBindingTargetName(output, method);
                Class<?> returnType = method.getReturnType();
                Object sharedBindingTarget =
                    locateSharedBindingTarget(name, returnType);
                if (sharedBindingTarget != null) {
                    BindableProxyFactory.this.outputHolders.put(name,
                        new BoundTargetHolder(sharedBindingTarget, false));
                }
                else {
                    BindableProxyFactory.this.outputHolders.put(name,
                        new BoundTargetHolder(
                            getBindingTargetFactory(returnType)
                                .createOutput(name), true));
                }
            }
        });
}
```

afterPropertiesSet方法会处理所有被@Input和@Output注解的函数，并将生成函数返回类型实例存储在BoundTargetHolder中，getBindingTargetName方法会返回SubscribableChannelBindingTargetFactory实例，它会在createOutput方法中返回一个DirectChannel实例，该实例会被存储起来供BindableProxyFactory使用。

名称为output的BeanDefinition将BindableProxyFactory设置成其实例工厂类，并将outputMessagefunction方法设置成其实例的工厂函数（BeanFactoryMethod）。当Spring容器创建该实例时，会调用BindableProxyFactory的outputMessagefunction方法，由于BindableProxyFactory实现了MethodInterceptor接口，所以就调用了其invoke方法。invoke方法会从BindableProxyFactory缓存的Channel实例中匹配符合的实例方法，并反射调用。

BindingService是Stream层获取绑定器和执行绑定任务的一个重要类，首先我们看BindingService的bindProducer方法，代码如下：

```
@SuppressWarnings("unchecked")
public <T> Binding<T> bindProducer(T output， String outputName) {
    String bindingTarget = this.bindingServiceProperties
        .getBindingDestination(outputName);
    Binder<T， ?， ProducerProperties> binder = (Binder<T， ?， ProducerProperties>)
        getBinder(outputName， output.getClass());
    ProducerProperties producerProperties = this.bindingServiceProperties
        .getProducerProperties(outputName);
    if (binder instanceof ExtendedPropertiesBinder)
    {
        Object extension = ((ExtendedPropertiesBinder) binder)
           .getExtendedProducerProperties(outputName);
        ExtendedProducerProperties extendedProducerProperties =
            new ExtendedProducerProperties<>(extension);
        BeanUtils.copyProperties(producerProperties， extendedProducerProperties);
        producerProperties = extendedProducerProperties;
    }
    validate(producerProperties);
    Binding<T> binding = binder.bindProducer(bindingTarget， output，producerProperties);
    this.producerBindings.put(outputName， binding);
    return binding;
}
```

在BindingService实现中，getBinder方法最终会调用DefaultBinderFactory中的getBinder方法实现，我们可以看到，DefaultBinderFactory的作用就是获取具体的Binder实现并提供给相应的MessageChannel实例。DefaultBinderFactory的初始化依赖于BinderTypeRegistry获得的BinderType列表。DefaultBinderFactory的getBinder实现中会调用BinderConfiguration获取对应的Binder实例，通过跟踪BinderConfiguration的初始化过程，可以发现BinderConfiguration是在BinderFactoryConfiguration执行getBinderConfiguration方法时将bindingServiceProperties变量中的BinderProperties与BinderTypeRegistry中的BinderType结合，封装成BinderConfiguration对象。BinderProperties封装了Stream从application.yml文件中读取的关于Binder的配置信息，而BinderType则是具体Binder的实现类信息。DefaultBinderFactory的getBinderInstance实现如下：

```
private <T> Binder<T， ?， ?> getBinderInstance(String configurationName) {
    if (!this.binderInstanceCache.containsKey(configurationName)) {
        BinderConfiguration binderConfiguration =
    this.binderConfigurations.get(configurationName);
    if (binderConfiguration == null) {
        throw new IllegalStateException("Unknown binder configuration: "
            + configurationName);
    }
    BinderType binderType =
        this.binderTypeRegistry.get(binderConfiguration.getBinderType());
```

```
        Assert.notNull(binderType, "Binder type " +
            binderConfiguration.getBinderType() + " is not defined");
        Properties binderProperties = binderConfiguration.getProperties();
        ArrayList<String> args = new ArrayList<>();
        for (Map.Entry<Object, Object> property : binderProperties.entrySet()) {
            args.add(String.format("--%s=%s", property.getKey(), property.getValue()));
        }
        //省略
        args.add("--spring.jmx.default-domain=" + defaultDomain + "binder." +
            configurationName);
        args.add("--spring.main.applicationContextClass=" +
            AnnotationConfigApplicationContext.class.getName());
        List<Class<?>> configurationClasses = new ArrayList<Class<?>>(
            Arrays.asList(binderType.getConfigurationClasses()));
        SpringApplicationBuilder springApplicationBuilder = new SpringApplicationBuilder()
            .sources(configurationClasses.toArray(
            new Class<?>[] {})).bannerMode(Mode.OFF).web(false);
        boolean useApplicationContextAsParent = binderProperties.isEmpty() &&
            this.context != null;
        if (useApplicationContextAsParent) {
            springApplicationBuilder.parent(this.context);
        }
        if (useApplicationContextAsParent || (environment != null &&
            binderConfiguration.isInheritEnvironment())) {
            if (environment != null) {
                StandardEnvironment binderEnvironment = new StandardEnvironment();
                binderEnvironment.merge(environment);
                springApplicationBuilder.environment(binderEnvironment);
            }
        }
        ConfigurableApplicationContext binderProducingContext = springApplicationBuilder
            .run(args.toArray(new String[args.size()]));
        @SuppressWarnings("unchecked")
        Binder<T, ?, ?> binder = binderProducingContext.getBean(Binder.class);
        if (this.listeners != null) {
            for (Listener binderFactoryListener : listeners) {
                binderFactoryListener.afterBinderContextInitialized(
                configurationName, binderProducingContext);
            }
        }
        this.binderInstanceCache.put(configurationName, new
            BinderInstanceHolder(binder, binderProducingContext));
        }
        return (Binder<T, ?, ?>)
        this.binderInstanceCache.get(configurationName).getBinderInstance();
    }
```

这里的getBinderInstance方法中会生成一个ConfigurableApplicationContext来创建Binder实例，在创建ConfigurableApplicationContext实例时，它会将BinderConfiguration设置到SpringApplicationBuilder中。

ConfigurableApplicationContext调用getBinder方法时，会使用BinderConfiguration的属性和配置生成BinderConfiguration中设置的具体类型的Binder实现。如果你使用的Binder是RabbitMQ，那么对应的RabbitServiceAutoConfiguration会自动初始化并加载RabbitMessageChannelBinder实例。

在Stream层对Binder实例的初始化工作都完成后，再回到BindingService的bindProducer方法实现，它会调用AbstractMessagChannlBinder的doBindProducer方法，关键代码如下：

```
@Override
public final Binding<MessageChannel> doBindProducer(final String destination, MessageChannel
    outputChannel,
    final producerProperties) throws BinderException {
        Assert.isInstanceOf(SubscribableChannel.class, outputChannel,
            "Binding is supported only for SubscribableChannel instances");
    final MessageHandler producerMessageHandler;
    final ProducerDestination producerDestination;
    try {
        producerDestination =
            this.provisioningProvider.provisionProducerDestination(destination,
                producerProperties);
     }//省略
}
```

从源码可知，ProvisioningProvider是一个接口，不同的Binder实现可以根据接口实现各自不同的ProducerDestination和ConsumerDestination，代码如下：

```
public interface ProvisioningProvider<C extends ConsumerProperties,
    P extends ProducerProperties>
{
    ProducerDestination provisionProducerDestination(String name, P properties)
      throws ProvisioningException;
    ConsumerDestination provisionConsumerDestination(String name, String group, C
        properties)throws ProvisioningException;
}
```

doBindProducer会调用createProducerMessageHandler方法创建MessageHandler实例，MessageChannel会使用SendingHandler封装后的MessageHandler实例，当有output消息时，将消息发送给最终的Binder实例。

通过上面的步骤，基本上在Stream层就完成了对生产者的绑定操作，消费者的绑定就是将SubscribableChannel与具体的消息队列实现连接，doBindConsumer与doBindProducer流程类似。

首先通过ProvisioningProvider的provisionConsumerDestination方法创建ConsumerDestination，然后调用createConsumerEndpoint方法创建MessageProducer实例，最后生成DefaultBinding实例，代码如下：

```
    @Override
    public final Binding<MessageChannel> doBindConsumer(String name, String group,
        MessageChannel inputChannel, final C properties) throws BinderException {
        MessageProducer consumerEndpoint = null;
        try {
            final ConsumerDestination destination =
            this.provisioningProvider.provisionConsumerDestination(name, group, properties);
            final boolean extractEmbeddedHeaders =
                HeaderMode.embeddedHeaders.equals(properties.getHeaderMode())
                && !this.supportsHeadersNatively;
            ReceivingHandler rh = new ReceivingHandler(extractEmbeddedHeaders);
            rh.setOutputChannel(inputChannel);
            final FixedSubscriberChannel bridge = new FixedSubscriberChannel(rh);
            bridge.setBeanName("bridge." + name);
            consumerEndpoint = createConsumerEndpoint(destination, group, properties);
            consumerEndpoint.setOutputChannel(bridge);
            if (consumerEndpoint instanceof InitializingBean) {
                ((InitializingBean) consumerEndpoint).afterPropertiesSet();
            }
            if (consumerEndpoint instanceof Lifecycle) {
                ((Lifecycle) consumerEndpoint).start();
            }
            final Object endpoint = consumerEndpoint;
            EventDrivenConsumer edc = new EventDrivenConsumer(bridge, rh);
            edc.setBeanName("inbound." + groupedName(name, group));
            edc.start();
            return new DefaultBinding<MessageChannel>(name, group, inputChannel,
                endpoint instanceof Lifecycle ? (Lifecycle) endpoint : null) {
            @Override
            protected void afterUnbind() {
                try {
                    if (endpoint instanceof DisposableBean) {
                        ((DisposableBean) endpoint).destroy();
                    }
                }
                catch (Exception e) {
                    AbstractMessageChannelBinder.this.logger
                        .error("Exception thrown while unbinding " + this.toString(), e);
                }
                afterUnbindConsumer(destination, this.group, properties);
                destroyErrorInfrastructure(destination, group, properties);
                }
            };
        }
}
```

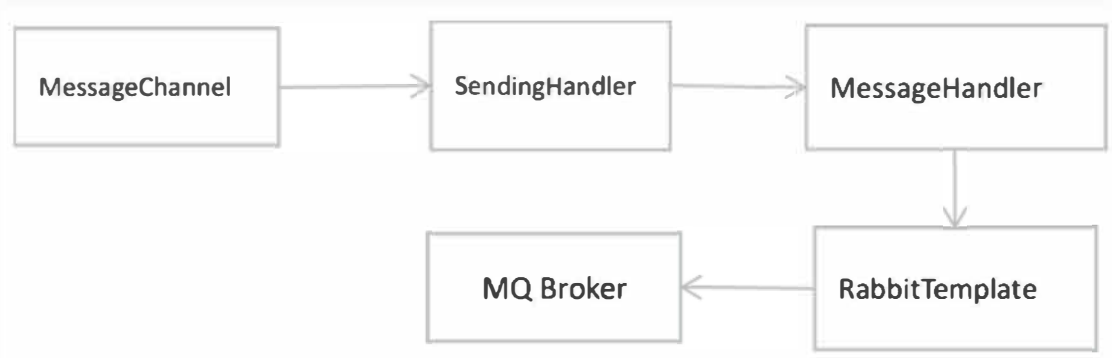
### Message/Integrate/消息中间件Binder层

从@Output注解可以看到，Stream框架会使用MessageChannel发送消息。通过BindingService的doBindProducer方法创建并绑定SendingHandler对象，然后调用handleMessageInternal方法，它会将消息再发送给delegate对象处理。下面是SendingHandler对象的handleMessageInternal方法的代码实现：

```
    @Override
    protected void handleMessageInternal(Message<?> message) throws Exception  {
        Message<?> messageToSend = (this.useNativeEncoding) ? message
            : serializeAndEmbedHeadersIfApplicable(message);
        this.delegate.handleMessage(messageToSend);
    }
```

delegate是之前在BindingServer中抽象类AbstractMessageChannelBinder执行的createProducerMessageHandler方法返回的生产者MessageHandler实例。对于RabbitMQ

Binder来说，就是rmqpOutboundEndpoint对象，该实例将最终调用其handlerMessage方法，该方法进一步调用RabbitTemplate的send方法。消息发送流程如下图所示。




**消息的接收过程**

消息的接收过程可以分为两个阶段：第一个阶段是从RabbitMQ到SubscribableChannel的过程。我们从@Input注解可以看到，Stream框架会使用SubscribableChannel接收消息。第二个阶段是注解@StreamListener告诉SubscribableChannel如何将消息发送给对应的Sink接收端对应的回调方法。

Spring的RabbitMQ使用InternalConsumer作为默认的消息消费方，当接收到对应消息后，会调用handleDelivery方法将RabbitMQ消息发送给BlockingQueueConsumer中的队列。下面是handleDelivery的源码实现：

```
public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body)throws IOException {
    try {
        if (BlockingQueueConsumer.this.abortStarted > 0) {
            if (!BlockingQueueConsumer.this.queue.offer(new
                Delivery(consumerTag, envelope, properties, body),
                    BlockingQueueConsumer.this.shutdownTimeout,
                    TimeUnit.MILLISECONDS)) {
                    RabbitUtils.setPhysicalCloseRequired(getChannel(), true);
                    BlockingQueueConsumer.this.queue.clear();
                    getChannel().basicNack(envelope.getDeliveryTag(), true, true);
                    getChannel().basicCancel(consumerTag);
                    try {
                        getChannel().close();
                    }
                    catch (TimeoutException e) {
                    }
                }
            }
            else {
                BlockingQueueConsumer.this.queue.put(new
                    Delivery(consumerTag, envelope, properties, body));
            }
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
}
```

AsyncMessageProcessingConsumer类是Runnable类型的，它会消费阻塞队列，并将消息传给AmqpInboundChannelAdapter。AmqpInboundChannelAdapter实例是在BindingService构造createConsumerEndpoint时创建的consumerEndpoint，并将它与对应的Channel绑定。下面是

AmqpInboundChannelAdapter 的关键代码，即processMessage方法，它会调用MessagingTemplate对象的send方法将消息发送给SubscribableChannel模块。

```
public class AmqpInboundChannelAdapter extends MessageProducerSupport
        Implements OrderlyShutdownCapable {
    protected class Listener implements ChannelAwareMessageListener, RetryListener {
    @SuppressWarnings("unchecked")
    @Override
    public void onMessage(final Message message, final Channel channel)
        throws Exception {
        try {
            if (AmqpInboundChannelAdapter.this.retryTemplate == null) {
                try {
                    processMessage(message, channel);
                }
                finally {
                    attributesHolder.remove();
                }
            }
            else {
                AmqpInboundChannelAdapter.this.retryTemplate.execute(new
                    RetryCallback<Object, RuntimeException>() {
                    @Override
                    public Void doWithRetry(RetryContext context) throws
                        RuntimeException
                    {
                        processMessage(message, channel);
                        return null;
                    }
                },
                 (RecoveryCallback<Object>)
                AmqpInboundChannelAdapter.this.recoveryCallback);
            }
        }
    }
}
```

下面就是消息处理的第二个阶段，就是将SubscribableChannel中的消息发送给指定的方法，主要靠@StreamListener注解实现。@StreamListener是注释在消费方法上的注解，用来接收输入型通道的消息，Stream定义了StreamListenerAnnotationBeanPostProcessor类，用来处理项目中的@SteamListener注解。

StreamListenerAnnotationBeanPostProcessor实现了BeanPostProcessor接口，用来在Bean初始化之前和之后两个时间点对Bean实例进行处理。

postProcessAfterInitialization是在Bean实例初始化之后被调用的方法，它会遍历Bean实例中的所有函数，处理那些被@StreamListener注解修饰的函数。

afterSingletonsInstantiated方法会遍历mappedListenerMethods对应的所有Entry对象，为每一个StreamListenerHandlerMethodMapping创建一个MessageHandler实例。然后根据条件生成DispatchingStreamListenerMessageHandler并注册给SubscribableChannel。

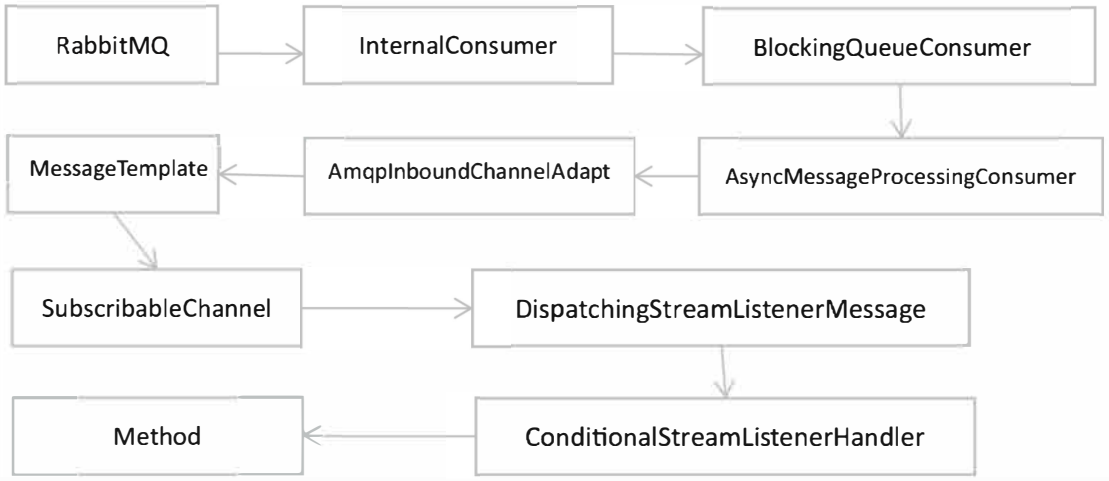
下面是StreamListenerAnnotationBeanPostProcessor的代码实现：

```
    public final void afterSingletonsInstantiated() {
```

```
    for (Map.Entry<String,  List<StreamListenerHandlerMethodMapping>>
        mappedBindingEntry : mappedListenerMethods.entrySet())
    {
    ArrayList<DispatchingStreamListenerMessageHandler.ConditionalStreamListener
        MessageHandlerWrapper> handlers = new ArrayList<>();
    for (StreamListenerHandlerMethodMapping mapping : mappedBindingEntry.getValue()) {
        final InvocableHandlerMethod invocableHandlerMethod =
            this.messageHandlerMethodFactory.createInvocableHandlerMethod(
                mapping.getTargetBean(), checkProxy(mapping.getMethod(),
                mapping.getTargetBean()));
            StreamListenerMessageHandler streamListenerMessageHandler =
                new StreamListenerMessageHandler(
                    invocableHandlerMethod,  resolveExpressionAsBoolean(
                        mapping.getCopyHeaders(),  "copyHeaders"),
        springIntegrationProperties.getMessageHandlerNotPropagatedHeaders()
        streamListenerMessageHandler.setApplicationContext(this.applicationContext);
        streamListenerMessageHandler.setBeanFactory(
            this.applicationContext.getBeanFactory());
        if (StringUtils.hasText(mapping.getDefaultOutputChannel())) {
               streamListenerMessageHandler.
                setOutputChannelName(mapping.getDefaultOutputChannel());
        }
        streamListenerMessageHandler.afterPropertiesSet();
        if (StringUtils.hasText(mapping.getCondition())) {
            String conditionAsString =
                resolveExpressionAsString(mapping.getCondition(),  "condition");
            Expression condition =
                SPEL_EXPRESSION_PARSER.parseExpression(conditionAsString);
            handlers.add(
                New DispatchingStreamListenerMessageHandler.
                ConditionalStreamListenerMessageHandlerWrapper(
                    condition,  streamListenerMessageHandler));
            }
        else {
            handlers.add(
                new DispatchingStreamListenerMessageHandler
                    .ConditionalStreamListenerMessageHandlerWrapper(
                    null,  streamListenerMessageHandler));
            }
        }
       this.mappedListenerMethods.clear();
    }
}
```

当SubscribableChannel接收到消息后，会调用DispatchingStreamListenerMessageHandler类的handleRequestMessage方法，该方法会调用ConditionalStreamListenerHandler的handleMessage方法。

findMatchingHandlers方法根据ConditionalStreamListenerHandler的Expression实例来判断ConditionalStreamListenerHandler是否适合处理当前这个消息，最终消息经过InvocableHandlerMethod传递给对应的函数。SCS消费消息的整体流程如下图所示。




## 6.4.8 Stream应用进阶

在微服务架构下，细粒度服务之间更容易发生频繁的分布式集成与交互。基于消息中间件的服务交互模式，或者建立以事件驱动为主导的架构模型，可以帮助业务建立和实现核心的领域事件驱动交互机制。

领域事件（Domain Event）的通信改变了领域对象的状态，比如订单创建事件、库存添加事件。一个领域事件可以表达正在发生在一个领域对象上的行为。领域事件的发生伴随着修改的数据、时间戳、聚合ID及其他附加信息。在微服务架构下，领域事件以发布/订阅的模式，将事件发布到MQ中间件，允许多个不同微服务订阅事件并消费事件，这个事件可能是一个订单创建事件（OrderCreateEvent）或订单修改事件（CustomerModifyEvent）。以上方式都遵循面向对象的方式，然而这些对象穿梭在生产者、分布式消息队列和消费者中，变成了共享类库，当众多微服务需要依赖共享类库时，就产生了高度的耦合。这种共享分布式对象实现远程调用是通过把共享对象打成jar包被不同微服务共享依赖的，也是分布式系统中的典型反模式，当一个领域事件被修改后，每个依赖的微服务都会受到影响。

使用SCS通过Spring Message的公共消息事件机制，可以支持事件驱动架构，同时避免依赖共享Domain对象类型。下面我们在SCS的一些元注解的基础上结合Spring的一些小特性实现类似CORS的EDA架构。

**Spring Cloud Stream处理事件**

SCS提供了@StreamListener注解来控制序列化的方式，它作为方法的入参并执行方法，例如：

```
    @StreamListener(Sink.INPUT)
    public void handle(Foo foo){
    //省略
}
```

新的事件分发特性在@StreamListener上增加了Condition属性来使消息路由到多个监听器成为可能，Condition的值是用SPEL表达式运算出来的一个boolean值。Condition应用到传入的消息

上，能够计算任何消息载荷、特定的消息头或其组合。这提供了一种极其灵活的路由机制，且不需要不同的事件类型定义类。例如，我们定义一个带String eventType属性的Event类型，SCS将提供开箱即用的功能：

```
@EnableBinding
class MyEventHandler{
 @StreamListener(target=Sink.INPUT，condition="payload.eventType=='CustomerCreatedEvent'")
 public void handleCustomerEvent(@Payload Event event) {
 }
 @StreamListener(target=Sink.INPUT，
      condition="payload.eventType=='AccountCreatedEvent'")
 public void handleAccountEvent(@Payload Event event) {
  }
}
```

其中，Event类型的定义如下：

```
package com.example.domain;
public interface Event {
    String getEventType();
}
```

**定制化事件注解**

虽然可以使用通用的Event类型，但是仍然需要通过@StreamListener注解加入SEL Condition注解来过滤，使用起来比较麻烦。现在我们基于@StreamListener注解实现自定义事件注解，我们通过事件类型直接定位到我们的方法。自定义注解@EventHandler的定义如下：

```
@StreamListener
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
 public @interface EventHandler {
    @AliasFor(annotation=StreamListener.class， attribute="target")
    String value() default "";

    @AliasFor(annotation=StreamListener.class， attribute="target")
    String target() default Sink.INPUT;
    @AliasFor(annotation=StreamListener.class， attribute="condition")
    String condition() default "";
}
```

通过@EventHandler的定义，我们可以把事件监听注解简化为如下形式：

```
@EnableBinding
class MyEventHandler{
    @EventHandler(condition="payload.eventType=='CustomerCreatedEvent'")
    public void handleCustomerEvent(@Payload Event event) {
     }
     @EventHandler(condition="payload.eventType=='AccountCreatedEvent'")
    public void handleAccountEvent(@Payload Event event) {
     }
}
```

这里，我们可以通过将Condition表达式转化为一个模板实现，重载SCS处理@StreamListener注解的BeanPostProcessor，这样通过eventType就可以实现识别事件的方法。重载的函数如下：

```
@Configuration
public class EventHandlerConfiguration {
    private static String eventHandlerSpelPattern = "payload.eventType=='%s'";
    @Bean(name = STREAM_LISTENER_ANNOTATION_BEAN_POST_PROCESSOR_NAME)
    public static BeanPostProcessor streamListenerAnnotationBeanPostProcessor() {
        return new StreamListenerAnnotationBeanPostProcessor() {
        @Override
        protected StreamListener postProcessAnnotation(
            StreamListener originalAnnotation, Method annotatedMethod) {
        Map<String, Object> attributes = new HashMap<>(
            AnnotationUtils.getAnnotationAttributes(originalAnnotation));
        if (StringUtils.hasText(originalAnnotation.condition())) {
            String spelExpression =
                String.format(eventHandlerSpelPattern, originalAnnotation.condition());
                attributes.put("condition", spelExpression);
        }
        return AnnotationUtils.synthesizeAnnotation(attributes,
            StreamListener.class, annotatedMethod);
        }};
    }
}
```

接下来，我们可以用自定义的@EnableEventHandling注解来引入这个configuration：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import({EventHandlerConfiguration .class})
public @interface EnableEventHandling {
}
```

我们修改EventHandler注解，定义一个eventType属性来做Condition的别名：

```
@StreamListener
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface EventHandler {
    @AliasFor(annotation=StreamListener.class, attribute="condition")
    String value() default "";
    @AliasFor(annotation=StreamListener.class, attribute="target")
    String target() default Sink.INPUT;
    @AliasFor(annotation=StreamListener.class, attribute="condition")
    String eventType() default "";
}
```

然后，我们可以使用自定义的事件处理注解，只需要提供事件类型，就可以从消费者发送订阅事件：

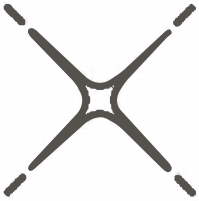
```
@EnableEventHandling
class MyEventHandler{
    @EventHandler(eventType = "CustomerCreatedEvent")
    public void handleCustomerEvent(@Payload Event event) {
     }
     @EventHandler(eventType = "CustomerCreatedEvent")
    public void handleAccountEvent(@Payload Event event) {
     }
}
```

## 6.5 小结

微服务集成架构倾向于使用标准化的HTTP、基于REST API的架构交互模式进行集成。此外，考虑到性能也可以采用RPC的调用方式。对于异步交互过程，使用消息队列可以实现微服务之间的充分解耦和异构集成。

Spring Cloud提供了Spring Cloud Stream框架，它可以屏蔽底层通信技术细节，并且实现了基于消息的轻量级微服务集成解决方案。还可用使用Spring Cloud Stream实现基于事件驱动和CQRS的系统架构。

微服务数据架构

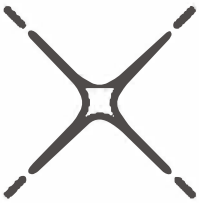


微服务架构强调技术的多样性，选择最合适的技术解决业务的实际问题，这一原则同样适用于微服务数据存储领域。目前随着数据海量的增长、数据类型的多样性、对数据访问性能更快的诉求，关系数据库越来越不能满足用户的需求，于是NoSQL数据库应运而生。本章我们首先介绍数据分类，以及不同数据类型适合采用的不同数据存储技术，同时介绍NoSQL存储与关系数据库的主要区别和特性。

在微服务架构下，对于数据一致性的处理，强一致性的事务管理机制不一定是适合的解决方案，之前单体架构下强一致性的事务模式在微服务架构中可能会带来一系列性能损失和数据一致性问题与挑战。微服务架构有很多不同的设计考量，它强调去中心化的数据治理，更强调每个微服务都拥有自己独立的数据存储，而不同服务在数据共享方面需要采取一定的策略和补偿方式来保证数据的一致性。通过对TCC、Saga等模式的介绍，我们可以了解当前微服务架构数据的最终一致性解决方案。

第7章

# 微服务数据架构



微服务架构强调技术的多样性，选择最合适的技术解决业务的实际问题，这一原则同样适用于微服务数据存储领域。目前随着数据海量的增长、数据类型的多样性、对数据访问性能更快的诉求，关系数据库越来越不能满足用户的需求，于是NoSQL数据库应运而生。本章我们首先介绍数据分类，以及不同数据类型适合采用的不同数据存储技术，同时介绍NoSQL存储与关系数据库的主要区别和特性。

在微服务架构下，对于数据一致性的处理，强一致性的事务管理机制不一定是适合的解决方案，之前单体架构下强一致性的事务模式在微服务架构中可能会带来一系列性能损失和数据一致性问题与挑战。微服务架构有很多不同的设计考量，它强调去中心化的数据治理，更强调每个微服务都拥有自己独立的数据存储，而不同服务在数据共享方面需要采取一定的策略和补偿方式来保证数据的一致性。通过对TCC、Saga等模式的介绍，我们可以了解当前微服务架构数据的最终一致性解决方案。

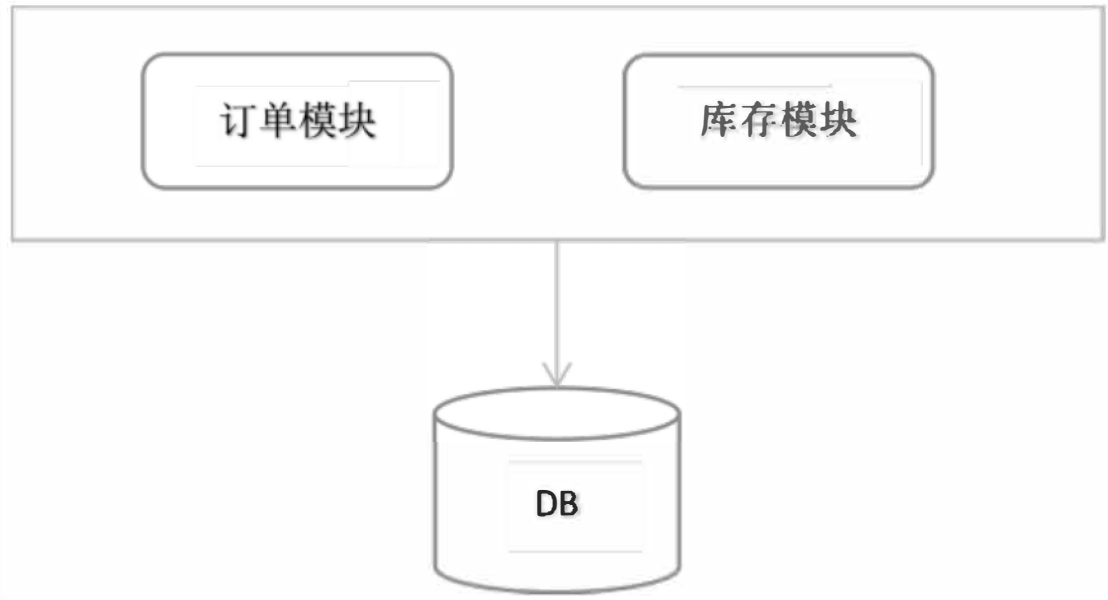
## 7.2 事务管理理论

## 7.2.1 事务管理概述

事务（Transaction）提供一种机制，它将一个执行过程涉及的所有操作都纳入一个不可分割的执行单元。组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，只要其中任一操作执行失败，都将导致整个事务的回滚。事务拥有ACID特性，后面我们会对事务的ACID特性做进一步说明。

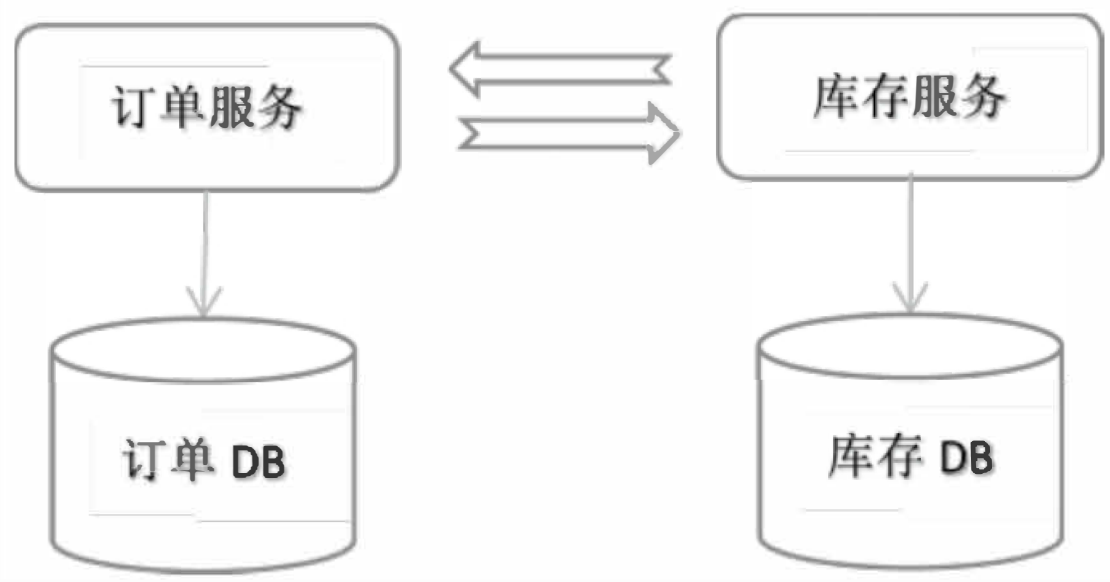
**本地事务和分布式事务**

本地事务是在同一个进程实例中调用不同的资源形成事务，紧密依赖底层资源管理器（例如数据库连接），事务处理局限在当前事务资源内。此种事务处理方式不存在对其他应用进程或实例的依赖。一般来说，本地事务只涉及一个Connection的Commit。本地事务用JDBC事务可以实现，依赖数据库机制就可以保证事务的ACID特性。下图是模拟购物系统的两个模块组成的本地事务的案例。




项目早期，在系统规模及用户体量不大的时候，两个模块打包在同一个系统中，通过本地事务就可以保证订单表（Order Table）和账务表（Account Table）的数据一致性，然而随着系统容量的增加，基于规模、扩展性、性能、共享等因素，需要将两个模块拆分成独立的微服务，这时两个模块如果需要数据的同步，就面临数据不一致的问题。

而使用分布式事务是早期解决非本地事务的一个主要手段。分布式事务指事务的参与者、支持事务的服务器、资源服务器及事务管理器分别位于不同的分布式系统的不同节点上。而分布式事务的实现方式有很多种，最具有代表性的是由Oracle Tuxedo系统提出的XA分布式事务协议。XA协议包括两阶段提交（2PC）和三阶段提交（3PC）两种。下图反映了跨越多个服务的一个分布式事务场景。




**微服务独立数据库模式**

微服务分布式架构倾向于为每一个独立的服务使用各自独立的数据库，因为如果不同微服务采用共享存储的方案，那么这两种服务就在数据层面产生了一种“有状态”的数据依赖关系，两个服务之间就产生耦合关系，在共享存储的情况下很难做到系统的独立性和自治性，对后期业务的扩展、可用性等方面都会产生重大影响，所以合理的方式就是各自使用独立的数据存储。使用消息传递或者分布式事务方式保证服务之间的数据一致性，而2PC就是比较传统和典型的数据一致性方案。

## 7.2.2 ACID理论

关系数据库天生可以解决具有复杂事务场景的问题，关系数据库完全满足ACID特性。ACID的具体含义如下。

- 原子性（Atomicity）：事务作为一个整体被执行，包含在其中的数据库操作要么全部被执行，要么全部都不执行。
- 一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态指数据库中的数据应满足完整性约束。除此之外，一致性还有另外一层语义，就是事务的中间状态不能被观察到（这层语义也有说应该属于原子性）。
- 隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行，如同只有这一个操作在被数据库执行一样。
- 持久性（Durability）：已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时，此操作将不可逆转。

具有ACID特性的数据库支持强一致性。强一致性表示数据库本身不会出现不一致的情况，每个事务都是原子的，或者成功或者失败，事务间是隔离的，互相完全不影响，而且最终状态是持久落盘的。因此，数据库会从一个明确的状态到另一个明确的状态，中间的临时状态是不会出现的，如果出现也会被及时自动修复，因此是强一致的。

我们的本地事务由资源管理器进行管理：事务的ACID特性是通过InnoDB日志和锁来保证的。InnoDB是MySQL的一个存储引擎，大部分人对MySQL都比较熟悉，这里简单介绍一下数据库事务实现的一些基本原理。在本地事务中，服务和资源在事务的包裹下可以看作是一体的。

- InnoDB隔离性实现：通过数据库锁的机制实现。
- InnoDB原子性和一致性实现：通过UndoLog来实现事务的原子性。在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为UndoLog），然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，则系统可以利用UndoLog中的备份将数据恢复到事务开始之前的状态。
- InnoDB持久性实现：通过RedoLog（重做日志）来实现，RedoLog记录的是新数据的备份。在事务提交前，只要将RedoLog持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是RedoLog已经持久化。系统可以根据RedoLog的内容，将所有数据恢复到最新的状态。

严格的ACID事务对隔离性的要求很高，在事务执行中必须将所有的资源锁定。对于长事务来说，整个事务期间对数据的独占，将严重影响系统的并发性能。因此，在高并发场景中，对ACID的部分特性进行放松，从而提高性能，这便产生了CAP理论和BASE理论。

### 7.2.3 一致性理论

一致性就是数据保持一致，在分布式系统中，可以理解为多个节点中数据的值是一致的。而一致性又可以分为强一致性和弱一致性。最终一致性本质上也是弱一致性的一种特殊形式。分布式事务的目的是保障跨数据库的数据一致性，而跨数据库的事务操作是不可控的，网络及个别节点宕机都会造成数据的不一致。单机事务的ACID特性不适用于分布式网络条件下的事务控制，CAP理论告诉我们，数据的一致性和服务可用性、分区容忍无法同时满足；而BASE理论强调，在分布式系统下的数据一致性应该放弃瞬时态的一致性来换取服务的可用性和数据的最终一致性。数据一致性模型可以分为下面三类：

- 强一致性：是程度最高的一致性要求，也是最难实现的。以关系数据库中更新操作为案例，系统中的某个数据被成功更新后，后续任何对该数据的读取操作都是更新后的值。
- 弱一致性：系统中的某个数据被更新后，后续对该数据的读取操作可能得到更新后的值，也可能是更改前的值。但经过“不一致时间窗口”这段时间后，后续对该数据的读取都是更新后的值。
- 最终一致性：在某一时刻用户或者进程查询到的数据可能都不同，但是最终成功更新的数据都会被所有用户或者进程查询到。简单理解为，在一段时间后，数据会最终达到一致状态。这个状态是弱一致性的特殊形式，存储系统保证在没有新的更新的条件下，最终所有访问的都是最后更新的值。

### 7.2.4 CAP理论

分布式事务一直是业界的难题，难在CAP定理，即一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

- C：表示一致性，所有数据变动都是同步的，数据一致更新。
- A：表示可用性，指在任何故障模型下，服务都会在有限的时间内处理响应。
- P：表示分区容忍、分区容错性和可靠性。

在分布式系统中，网络无法100%可靠，分区其实是一个必然，如果我们选择了CA而放弃了P，那么当发生分区现象时，为了保证一致性，必须拒绝请求，但是A又不允许，所以分布式系统理论上不可能选择CA架构，只能选择CP或者AP架构。由于关系数据库是单节点的，因此不具有分区容错性，但是具有一致性和可用性。而分布式的服务化系统都需要满足分区容错性，那么我们必须在一致性和可用性中进行权衡，具体表现在服务化系统处理的异常请求在某一个时间段内可能是不完全的，但是经过自动的或者手工的补偿后，达到了最终一致性。

### 7.2.5 BASE理论

BASE理论解决了CAP理论提出的分布式系统一致性和可用性不能兼得的问题，BASE在英文中有“碱”的意思，而ACID有“酸”的意思，所以基于这两个英文提出了酸碱平衡理论。简单来说，就是需要在不同的场景下，分别使用ACID或者BASE来解决分布式系统中数据的一致性问题。

BASE理论与ACID截然不同，BASE理论满足CAP理论，通过牺牲强一致性，获得可用性，一般应用在服务化系统的应用层或者大数据处理系统，通过达到最终一致性来尽量满足业务的绝大部分需求。BASE理论包含三个元素。

- BA：Basically Available，基本可用。
- S：Soft State，柔性状态，状态可以有一段时间不同步。
- E：Eventually Consistent，最终一致性，最终数据是一致的，而不是时时保持强一致性。

BASE理论的柔性状态是实现BASE理论的方法，基本可用和最终一致性是目标。按照BASE理论实现的系统，由于不保证强一致性，系统在处理请求的过程中，可以存在短暂的不一致性。在短暂的不一致性窗口，请求处理处于临时状态中，系统在进行每步操作时，记录每一个临时状态；当系统出现故障时，可以从这些中间状态继续未完成的请求处理或者退回到原始状态，最后达到一致状态。

BASE理论解决了CAP理论中没有网络延迟的问题，在BASE理论中用柔性状态和最终一致性保证了延迟后的一致性。

柔性事务的理念则是通过业务逻辑将互斥锁操作从资源层面上移至业务层面。通过放宽对强一致性的要求，来换取系统吞吐量的提升。另外提供自动的异常恢复机制，在发生异常后也能确保事务的最终一致性。

## 7.3 微服务架构的数据一致性

微服务架构下，最好的分布式数据一致性解决方案就是尽量避免分布式事务，然而，在很多场景下，分布式事务是难以避免的。在金融、电信领域中，很多业务场景要求数据的强一致性，同时要保证服务的可扩展性和可靠性。如何保证分布式事务下的数据一致性成为微服务架构的一个重要课题和难点。

### 7.3.1 解决方案概览

在工程领域，分布式事务的讨论主要聚焦于强一致性和最终一致性的解决方案。常见的分布式事务有基于XA协议的两阶段（2PC）提交模式，以及改良版本的三阶段（3PC）提交模式。Java事务编程接口（Java Transaction API，JTA）和Java事务服务（Java Transaction Service，JTS）正是基于XA协议的实现。分布式事务包括事务管理器TM（Transaction Manager）和一个或多个支持XA协议的资源管理器RM（Resource Manager）。在微服务架构下，我们倾向使用最终一致性的方案。下面将介绍2PC模式、TCC模式、Saga模式等。

- 2PC模式，分布式事务比较典型的解决方案，但是对于微服务架构而言，可能这一方案并不适用，主要原因是不同微服务可能使用的数据存储类型不同。如果使用的NoSQL不支持事务的数据库，那么事务根本无法实现2PC模式。此外，2PC模式本身也存在同步阻塞、单点故障和性能问题。
- TCC模式，相比2PC模式，具有更强的灵活性和性能优势，TCC模式本质上是基于服务层的2PC编程模式，把事务从数据库层的事务操作逻辑抽象到业务服务层，通过服务层业务逻辑实现服务的补偿模式。
- Saga模式，核心理念是将事务切分成一组依次执行的短事务，也可以理解成基于业务补偿逻辑实现分布式下的高性能分布式事务模式。较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。在不同的分布式应用架构下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播、业务的侵入性、隔离性等，实现机制也是复杂多变的。

- 可靠消息模式的解决方案是使用消息队列进行系统间的解耦。由上游服务发起事件，通过消息队列传递到下游服务，下游服务接收到消息后进行事件消费，最终完成业务，达到数据一致。为了解决消息队列及上下游服务的不可靠性，通常还会借助额外的事件表和定时器辅助完成数据补偿。

### 7.3.2 两阶段提交模式

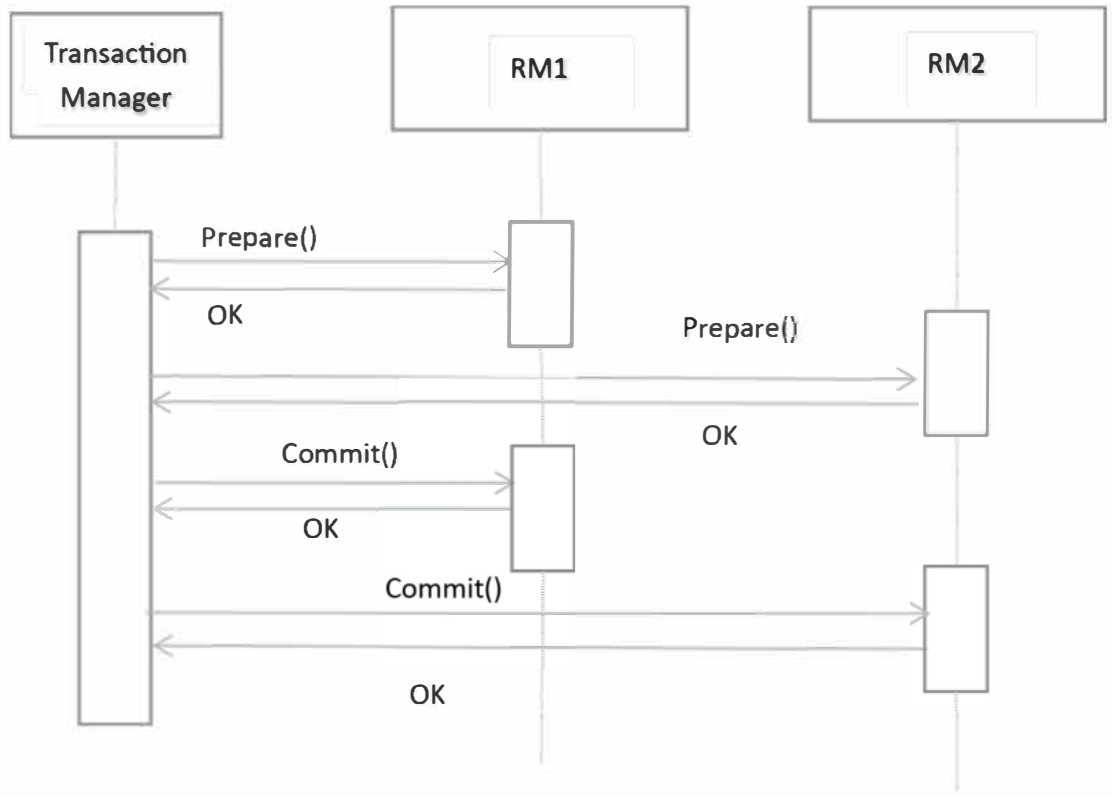
**2PC（两阶段提交）**

2PC是一个非常经典的强一致、中心化的通过原子提交来实现分布式事务一致性管理的协议。这里所说的中心化指协议中有两类节点：中心化协调者节点（Coordinator）和 N 个参与者节点（Participant）。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要协调者统一掌控所有节点（又称作参与者）的操作结果，并最终指示这些节点是否要把操作结果进行真正的提交或回滚。

2PC将整个事务流程分为两个阶段：准备阶段（Prepare Phase）和提交阶段（Commit Phase）。整个事务过程由事务管理器和参与者组成，事务管理器负责决策整个分布式事务的提交和回滚，事务参与者负责自己本地事务的提交和回滚。大部分关系数据库，如Oracle、MySQL，都支持两阶段提交协议。下面是计算机数据库进行两阶段提交的说明。

- 准备阶段：事务管理器为每个参与者准备（Prepare）消息，每个参与者在本地执行事务，并写本地的Undo/Redo日志，此时事务没有被提交。（Undo日志记录修改的数据，用于数据回滚；Redo日志记录修改后的数据，用于提交事务后写入数据文件）。
- 提交阶段：如果事务管理器收到了参与者执行失败或者超时的消息，则直接向每个参与者发送回滚消息；否则，发送提交消息。参与者根据事务管理器的消息执行提交或者回滚操作，并释放事务处理过程中使用的锁资源。

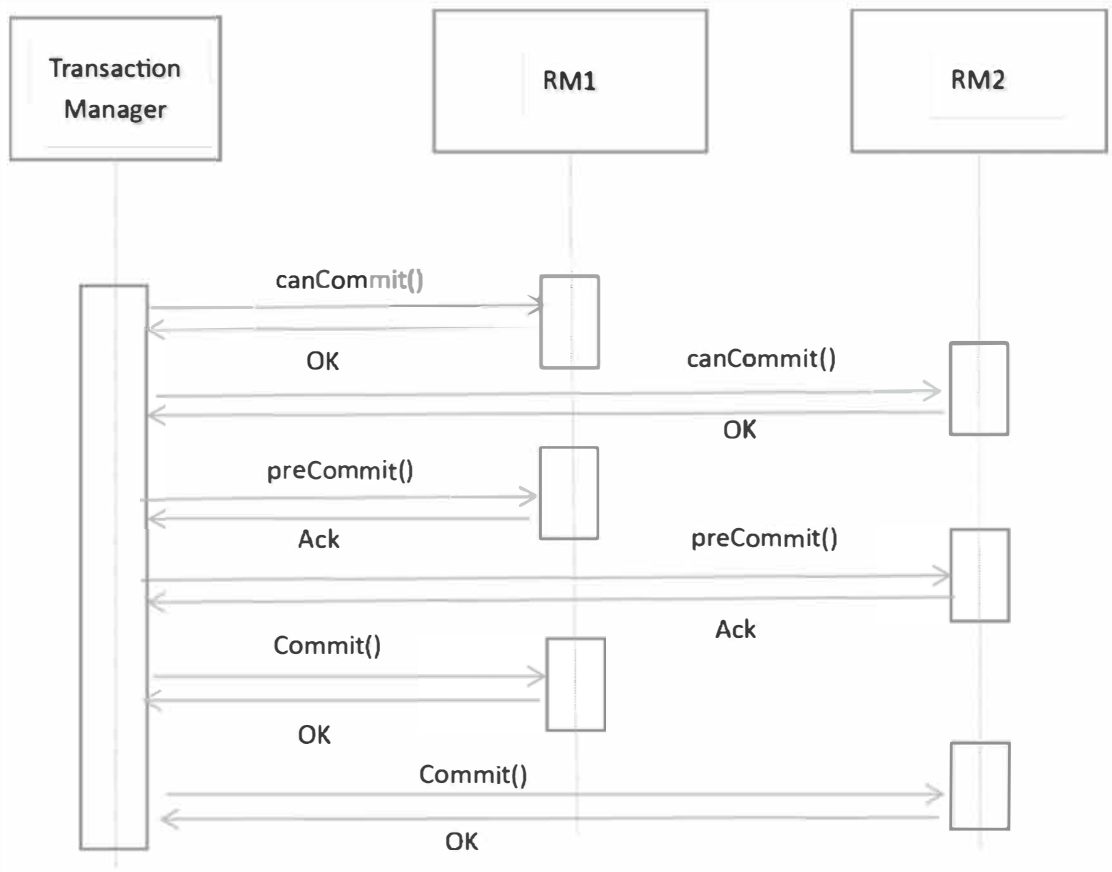
2PC的算法思路可以概括为：参与者将操作成败的结果通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是执行提交操作还是回滚操作。两阶段提交就是分成两个阶段提交，第一阶段询问各个事务数据源是否准备好，第二阶段才真正将数据提交给事务数据源。但因为2PC协议成本比较高，又有全局锁的问题，性能会比较差。现在我们基本上不会采用这种强一致性解决方案。2PC流程如下图所示。




在2PC中，如果两阶段出现协调者和参与者都宕机的情况，则有可能出现数据不一致的问题，同时还存在着诸如同步阻塞、单点问题、脑裂等问题，所以研究者们在2PC的基础上做了改进，提出了3PC。

### 3PC（三阶段提交）

3PC是2PC的改进版本，流程如下图所示。



3PC要解决的最关键问题就是协调者和参与者同时宕机的问题，所以3PC将2PC的第一阶段一分为二，形成了由canCommit、preCommit和Commit 3个阶段组成的事务处理协议。

3PC的核心理念是：在询问时并不锁定资源，除非所有参与者都同意了，才开始锁资源。一旦参与者无法及时收到来自协调者的信息，它就会默认执行Commit，而不会一直持有事务资源并处于阻塞状态，但是这种机制也会导致数据一致性问题。3PC在2PC的基础上做了如下改进：

- 增加了超时机制。
- 在两阶段之间插入了准备阶段。

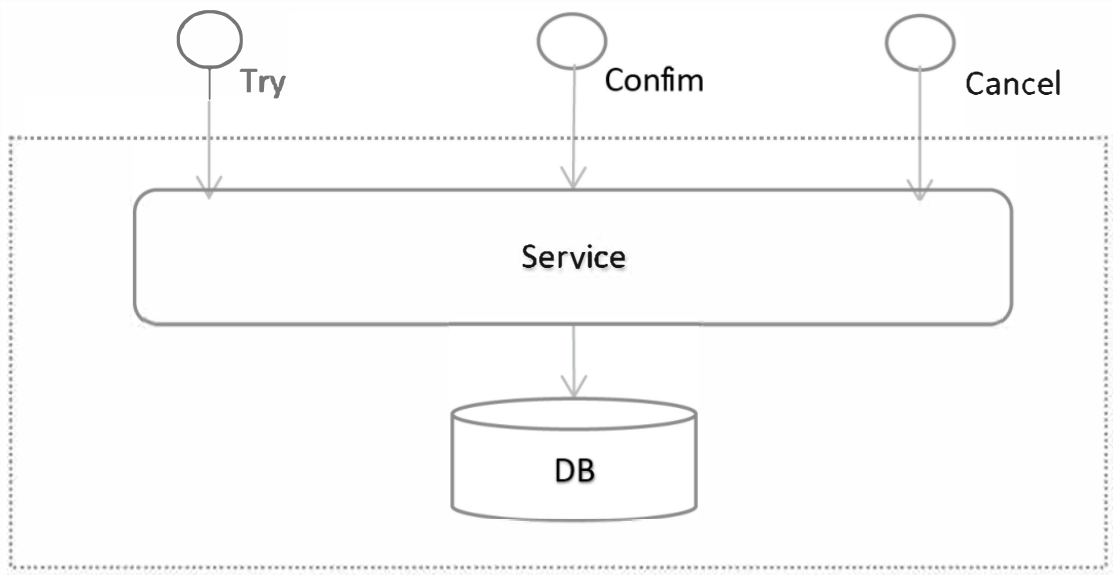
### 7.3.3 TCC补偿模式

关于TCC（Try-Confirm-Cancel）的概念，最早是由Pat Helland在2007年发表的一篇名为Life beyond Distributed Transactions：an Apostate ’s Opinion 的论文中提出的。在该论文中，TCC还是以Tentative-Confirmation-Cancellation命名的。

TCC的核心思想是：针对每个操作都要注册一个与其对应的确认和补偿（撤销）操作。TCC事务处理流程和2PC类似，不过2PC通常都在跨库的DB层面，而TCC本质上就是一个应用层面的2PC，需要通过业务逻辑来实现。

**TCC模式的工作原理**

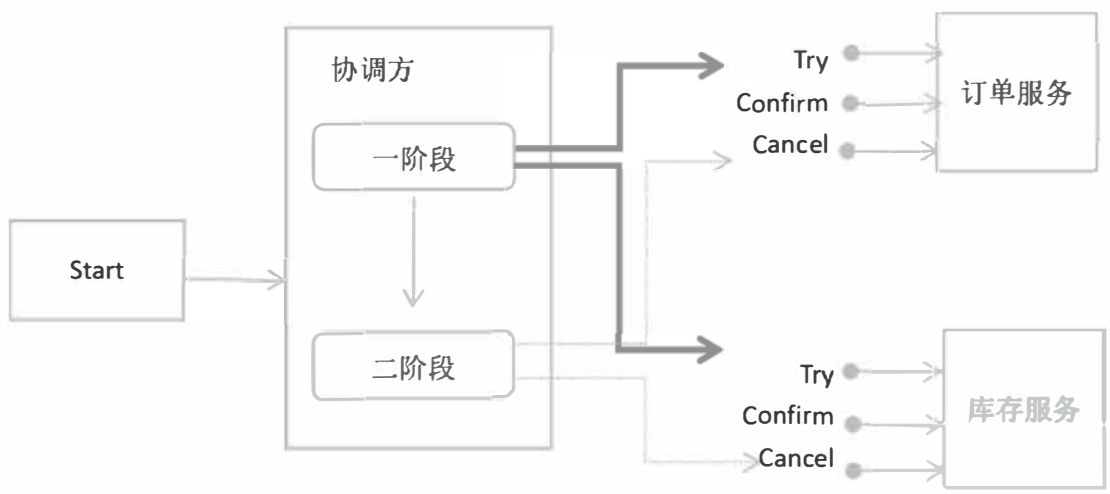
TCC模式的工作原理如下图所示。TCC将一个完整的事务提交分为Try、Confirm、Cancel 3个操作。




- Try：预留业务资源/数据效验。
- Confirm：确认执行真正要执行的业务，如果所有事务参与者的Try操作都执行成功了，就会调用所有事务参与者的Confirm操作，确认资源。Confirm操作满足幂等性，要求具备幂等设计，Confirm失败后需要进行重试。

- Cancel：取消执行，如果有事务参与者在Try阶段执行失败，就调用所有已成功执行Try阶段的参与者的Cancel方法，释放Try阶段占用的资源。Cancel操作满足幂等性，Cancel阶段的异常和Confirm阶段的异常处理方案基本上一致。

下面是实际模拟用户下单的一个业务场景使用TCC模式的主要流程图。TCC第一阶段是图中的粗线条部分，这个阶段需要分别执行订单服务和库存服务的Try事务，预留必需的业务资源；在第一阶段执行成功后，就会进入第二阶段的Confirm操作，如果不成功，则进行Cancel操作。




**TCC的优点和缺点**

- TCC的优点：让应用自己定义数据库操作的粒度，使降低锁冲突、提高吞吐量成为可能。TCC的特点在于业务资源检查与加锁，一阶段进行校验，锁定资源，如果第一阶段都成功，则第二阶段对锁定资源进行交易逻辑，否则对锁定资源进行释放，这样就避免了数据库两阶段提交中的锁冲突和长事务低性能风险。
- TCC的缺点：业务逻辑的每个分支都需要实现Try、Confirm、Cancel 3个操作，应用侵入性较强，改造成本高。另外，实现难度较大，需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。为了满足一致性的要求，Confirm和Cancel接口必须实现幂等。

**TCC的开源解决方案**

目前，国内的蚂蚁金服主要采用TCC模式进行分布式事务管理。下面总结了常用的TCC开源分布式管理框架：

- Seata
- Tcc-transaction
- Hmily
- ByteTCC
- EasyTransaction

## 7.3.4 Saga长事务模式

1987年，普林斯顿大学的Hector Garcia-Molina和Kenneth Salem发表了一篇论文： Sagas[注] 。这篇论文提出了使用Saga机制作为分布式事务的替代品，以解决长时间运行的分布式事务（long-Lived Transaction，LLT）问题，LLT指长时间持有数据库资源的长活事务。该论文认为业务过程经常由很多步骤组成，每一个步骤都涉及一个事务，如果将这些事务组成一个分布式事务，就可以实现总体一致。然而在长时间运行的分布式事务中，使用分布式事务会影响效率和系统的并发处理能力，因为在执行分布式事务时会有锁产生。Saga通过确保每一个业务过程都有修正事务来减少系统对分布式事务的依赖。这种在业务流程中执行修正事务的方式最终保证了系统数据一致性。

每一个LLT的Saga都由一系列sub-transaction Ti组成，每个Ti都有对应的补偿动作Ci，补偿动作用于撤销Ti造成的结果。同时Saga定义了两种恢复策略：

- 向后恢复模式（Backward Recovery），在这种模式下，每一个内部子事务都有一个对应的补偿事务，如果任一子事务失败，则将撤销之前所有成功的sub-transaction，使整个Saga的执行结果都撤销。
- 向前恢复模式（Forward Recovery），这种模式假设每个子事务最终都会成功，适用于必须要成功的场景，执行顺序如下：T1，T2，…，T j （失败），T j （重试），…，T n，其中T j 是发生错误的sub-transaction，此时执行重试逻辑，在该模式下不需要执行补偿事务。

微服务架构大师Chris Richardson在介绍微服务架构与数据一致性技术时，提出了Saga模式，将微服务中的分布式事务划分为一组小的事务，所划分事务或者全部提交，或者全部回滚。强调在微服务中，每个单独的微服务都可以确保ACID，因为每一个微服务都具备自己的数据库，但是，Saga模式作为整体并不保证隔离性，所以需要对异常情况进行补偿操作。

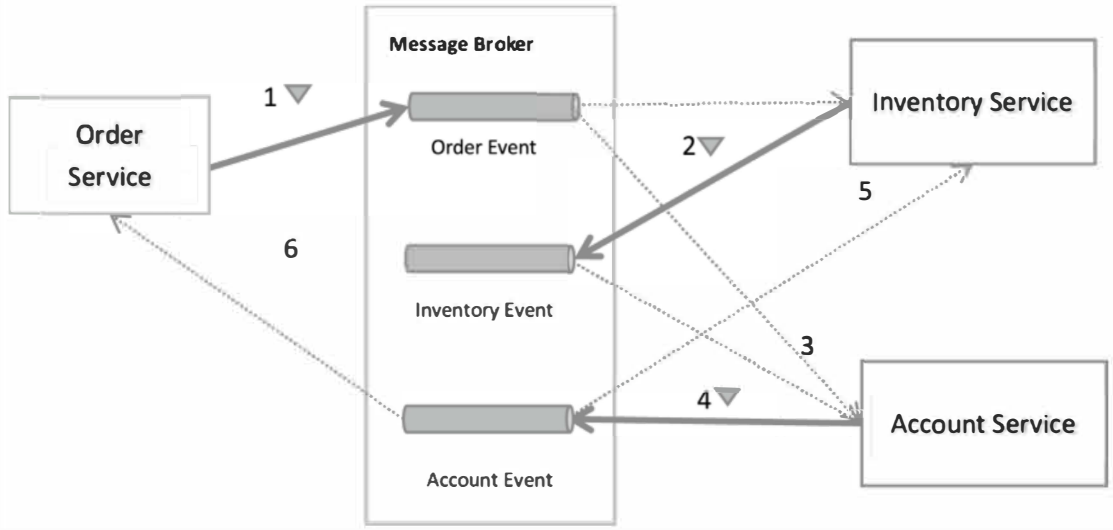
在Saga模式中，为了保障事务提交和回滚，应使用事务日志结尾和消息传递的方式。在发消息之前，将消息写入本地数据库，这就是所谓的事务日志结尾。它的作用是当新的日志到来时，可以直接发布，这样就可以强化ACD特性；同时Saga模式中子模块之间的消息通信建议采用消息传递方式，因为和HTTP通信协议相比，消息传递方式具备持久性。

**Saga模式的工作原理**

在微服务架构下，Saga存在两种协调模式。

- 编排模式（Choreography）

这种模式在微服务之间传递Saga，没有重协调器，每个微服务都监听其他服务，并决定采取行动。这种模式最大的优势就是简单，容易理解，参与者之间松散耦合，对于参与者较少的情况，适合采用这种模式。下面是使用编排模式的分布式事务时序图，实线表示消息发布，虚线表示订阅。




事件执行顺序如下：

（1）订单服务（Order Service）在Approval_pending状态下创建了订单，并发布订单创建事件。

（2）库存服务（Inventory Service）消费订单创建事件，在Inventory_pending状态下验证订单，订单出库，并创建Inventory Create Event。

（3）账户服务（Account Service）消费订单创建事件并进入等待的状态。

（4）账户服务消费Inventory Event，收取费用并发布Account Event。

（5）库存服务订阅Account Event商品出库，更改Inventory_pending状态为Accept状态。

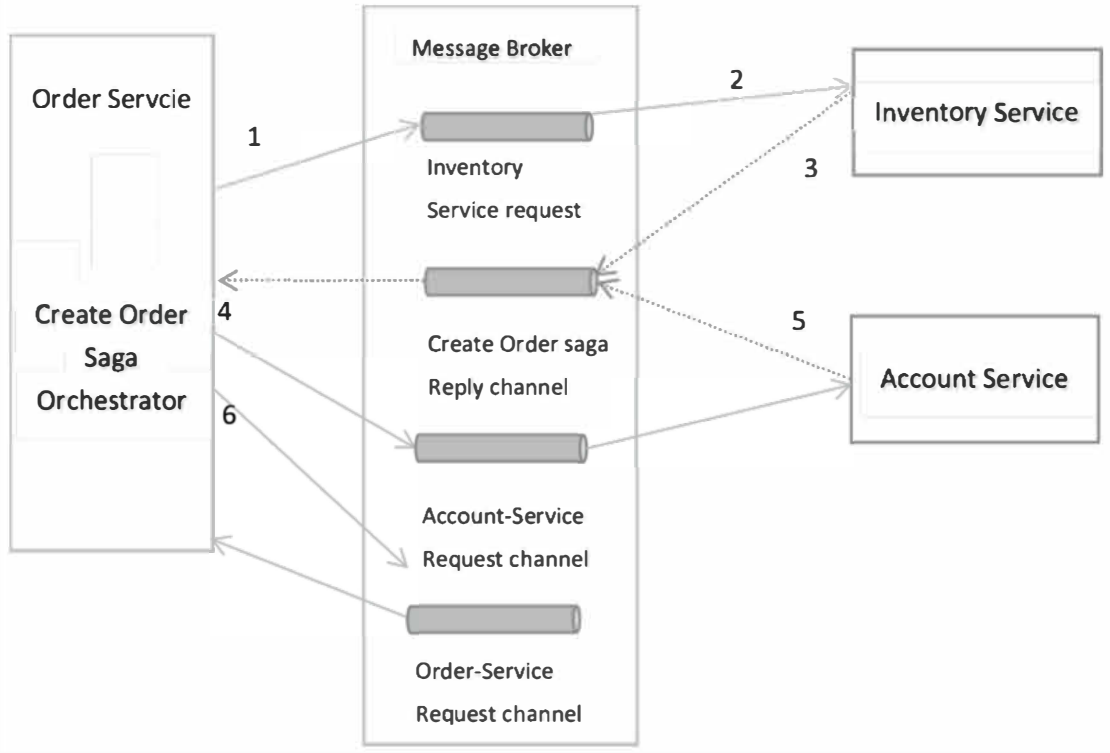
（6）订单服务收到Account Event，订单状态改为Approved状态。

（7）如果上述订单服务失败，那么库存服务消费费用收取失败事件，回滚之前库存事务清单，将状态改为拒绝。

（8）如果上述账户服务失败，那么订单服务消费费用收取失败事件，回滚订单状态改为拒绝。

● 编制模式（Orchestrator）

这种模式需要一个集中的服务触发器，跟踪Saga的所有子任务调用情况，根据调用情况来决定是否采用补偿措施。这种中央协调的方式可以减少每一个微服务之间的循环依赖，集成处理事件决策和逻辑排序，也更容易推理Saga组合，保证长事务数据的最终一致性。使用业务流程时，可以定义一个控制类，其唯一职责是告诉Saga参与者该做什么。Saga控制使用命令或异步回复样式交互与参与者进行通信。下面是使用编制模式的分布式事务时序图，实线表示消息request，虚线表示消息reply。




事件执行顺序如下：

（1）订单服务（Order Service）创建一个订单和一个订单控制器：Saga Orchestrator。

（2）Saga Orchestrator向库存服务（Inventory Service）发送一个创建订单命令。

（3）库存服务回复库存出库命令。

（4）Saga Orchestrator向账号服务（Account Service）发送一个费用收取命令。

（5）账号服务回复订单费用扣除命令。

（6）Saga Orchestrator向订单服务发送一个订单已批准命令。

**说明：** 基于业务流程，Saga编制的每个步骤都包括更新数据库和发布消息的服务，例如订单服务持久化和创建Saga Orchestrator，并向第一个Saga参与者发送消息。第一个Saga参与者是库存服务，通过更新数据库回复消息。然后订单服务通过更新Saga协调器的状态向下一个Saga参与者发送命令，并处理参与者的命令回复响应，服务必须使用事务性的消息传递，以便自动更新数据并发布消息。

**编排模式与编制模式**

微服务中推荐使用编制模式。相比编排模式，编制模式有下面几个优势。

- 编制模式有更简化的依赖关系。编排模式中，服务之间需要掌握相互依赖关系，而且针对不同的异常，可能还需要采用不同的补偿措施，Saga Orchestrator则调用Saga参与者，所

有参与者之间是不需要了解Orchestrator的实现细节的。

- 每个服务只需要暴露各自的API供Orchestrator调用即可，因此每个微服务之间有较少耦合，可以降低每个参与者的复杂度。
- 可以使用Saga Orchestrator更加方便地增加或减少Saga分布式事务逻辑，不需要改变每一个参与者的Saga内部事务。同时可以基于Orchestrator进行关注点分离，并简化业务，操作更加容易、方便。

**Saga与ACID**

Saga不提供对整体隔离性的保证。一个长事务划分为若干本地事务，在本地事务提交后，整个Saga未完成之前，其他服务可以访问“未完成”的中间状态事务数据；Saga协调器实现原子特性，通过日志实现持久性；通过本地日志与Saga日志保证事务一致性。所以Saga模式只支持ACD，不提供隔离性的保证。

**Saga与2PC的区别**

Saga与2PC最主要的区别在于，2PC是一个强一致性的分布式事务模式，Saga和TCC都通过应用服务层牺牲ACID特性来实现事务的最终一致性，Saga和TCC可以理解为分布式环境下通过事务补偿模式的事务控制模型，只是Saga和TCC有各自不同的实现策略。

**Saga与TCC的区别**

Saga和TCC最大的区别在于，Saga没有预留资源，而是直接提交到数据库，Saga比TCC少了一步Try操作，无论最终事务成功或失败，TCC都需要与事务参与方交互两次。而Saga在事务成功的情况下只需要与事务参与方交互一次。如果事务失败，则需要采取补偿事务的方式进行回滚。

**Saga的开源解决方案**

Saga为相互独立、自治的微服务提供了一种分布式网络场景下的数据一致性的解决方案。下面是一些Saga模式的开源解决方案，由于篇幅所限，这里不再赘述方案的实现细节。

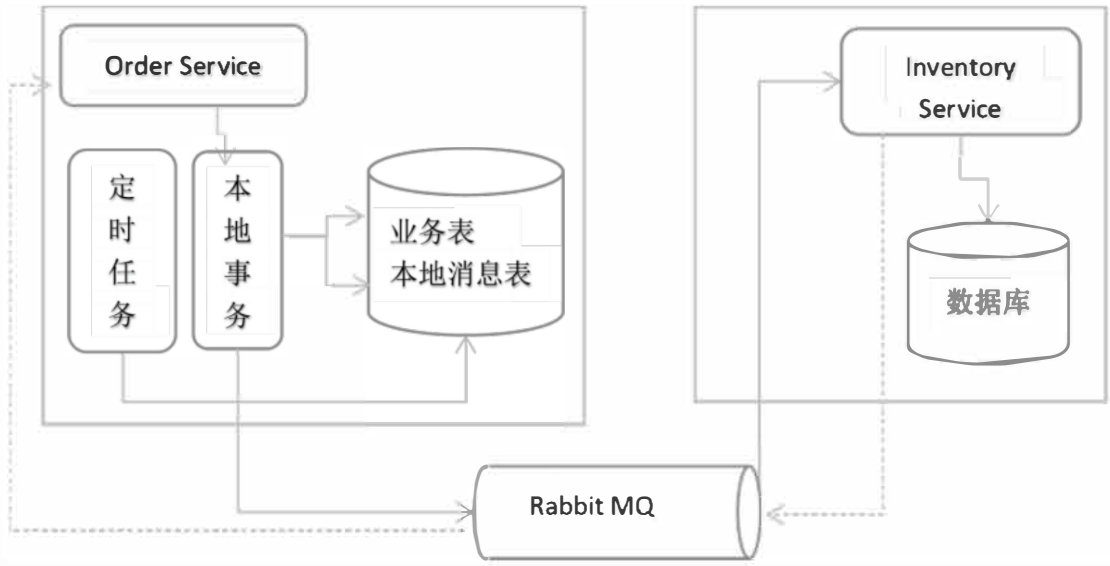
- ServiceComb Saga
- Axon Framework
- Eventuate-tram-sagas

## 7.3.5 可靠消息模式

可靠消息模式主要采用一个可靠的消息中间件作为中介，事务的发起方在完成本地事务后向可靠的消息中间件发起消息，事务消费方在收到消息后处理消息，该方案强调的是双方最终的数据一致性。

如下图所示，订单服务将消息发送给订单服务队列，库存服务监听订阅了订单服务的消息队列，并从消息队列中消费信息。由此可以看到，从事务的发起方到消息中间件，再到事务的消费方，中间都会通过网络，由于网络的不可靠性，导致分布式事务的数据不一致性。

基于这种网络的不可靠性，依靠本地消息表和可靠消息队列的模式可以解决分布式事务的不一致性。而这种模式，对业务的侵入性相对较小，在实现复杂度上也比较可控，国内很多互联网公司都采用了可靠消息模式来解决分布式事务的一致性问题。而这一方案的提出和思路来源于Ebay，之后这种模式在业内被广泛使用。这种分布式事务模式的本质是本地事务+可靠消息，以达到最终事务的一致性。我们来看可靠消息模式的具体实现原理，如下图所示。




可靠消息模式的思想是：事务的发起方需要额外创建一个本地消息表，将本地消息表和业务数据放在同一个事务中提交执行。也就是说，二者要么同时成功，要么同时失败。例如，将订单创建事务和库存验证及库存出库事件消息日志放入同一个事务中，下面是伪代码举例：

```
    begin transaction:
        Create Order(); //订单创建事务
        Inventory accept event(); //库存出库事件消息
    Commit transaction;
```

从上述伪代码可知，本地数据库与库存出库事务处于同一事务中，二者的绑定操作具备了原子性，工作时序如下：

（1）本地事务提交后，可以使用触发的方式对本地消息表进行查询和消息推送，或者使用定时器的方式轮询本地消息表进行消息推送。

（2）消费者的消息可以使用可靠的消息中间件机制，例如RabbitMQ中的ACK（消息确认机制）保证消费者可以一定消费到消息；又如库存服务在接收到消息并且完成消息业务处理后，回复ACK，说明此时库存服务已经正确同步数据；如果库存服务没有回复ACK，则消息中间件在没收到ACK消息时，将保留消息，并重复投递此消息。

（3）当消息中间件反馈消费成功后，库存服务可以回调一个订单服务的确认API，这时订单服务可以从本地事务表中删除对应的消息队列。

（4）在订单服务中，如果定时任务重复把本地事务表中的消息发到库存服务，则需要消息消费方（库存服务）提供消息的幂等性支持。

**幂等性**

简单来说，幂等性的概念就是：除了错误或者过期的请求（换言之就是成功的请求），无论多次调用还是单次调用，最终得到的效果都是一致的。通俗来说，只要有一次调用成功，再采用相同的请求参数，无论调用多少次（重复提交），都应该返回成功。

例如库存服务对外提供服务接口，必须承诺实现接口的幂等性，这一点在分布式系统中极其重要。

- 对于HTTP调用，承诺幂等性可以避免表单或者请求操作重复提交，造成业务数据重复。
- 对于异步消息调用，承诺幂等性通过对消息去重处理也是为了避免重复消费造成业务数据重复。

下面是几种常用的幂等性处理设计方案。

- 数据库表设计对逻辑上唯一的业务键唯一索引，这是在数据库层面做最后的保障。
- 业务逻辑上的防重，例如创建订单的接口，首先通过订单号查询库表中是否已经存在对应的订单，如果存在，则不做处理，直接返回成功。

补**偿方案**

在可靠消息事务方案中，事务发起方需要确保消息发送到消息队列中，而消息队列成为系统的瓶颈。当消息队列异常，或者消息消费失败导致数据不一致时，需要采取补偿措施，而常用的补偿方案由消息消费方负责。之所以使用消费方补偿模式有下面两个主要理由：

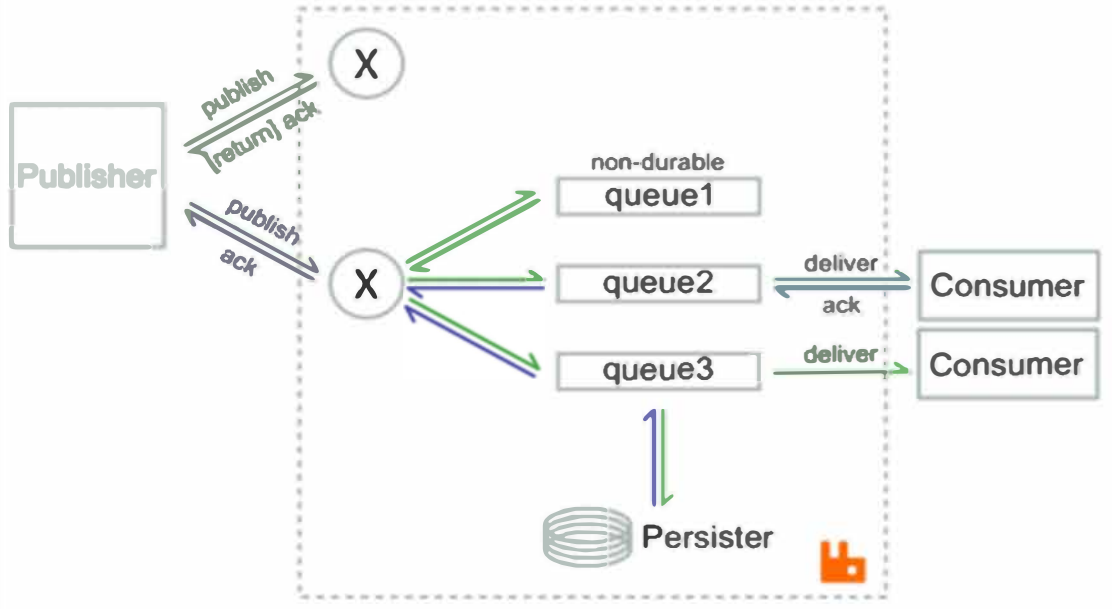
- 一般来说，数据不一致大概率发生在消息消费异常场景，如果由消息发送方补偿错误，往往无法解决消费异常问题；同时一个消息可能存在多个消费方，如果消息补偿模块在所有上游服务中编写，可能无法满足所有消费方的异常补偿场景。

- 当消费方出现问题时，需要定位事务发起方，然后才能通过上游来补偿，这种方式会增加处理生产问题的复杂度。

异步消息交互时，采取的补偿措施通常统一由消息消费方实现，这种方式将类似本地事件表的方式，在消息消费失败后，将失败消息写入本地数据库，然后启动定时任务进行重试，当达到重试上限时，进行预警和人工干预。

**RabbitMQ可靠消息传输实践**

在众多消息队列中，RabbitMQ最重要的特性就是将消息的可靠性作为传输消息考虑的第一要素。目前，RabbitMQ已经成为金融行业中消息队列的标配。我们将通过RabbitMQ的几个关键因素讲解RabbitMQ如何保证消息的可靠传输。RabbitMQ流程如下图所示。




**确认机制**

网络异常、机器异常、程序异常等多种情况都可能导致业务丢失消息。对消息进行确认可以解决消息的丢失问题，确认成功意味着消息已被验证并被正确处理。确认机制能用在两个方向：允许消费者告诉服务器（Broker）已经收到了消息，也允许服务器告诉生产者接收到了消息。前者就是我们常说的消费者ACK，后者就是我们常说的生产者Confirm。

RabbitMQ使用生产者消息确认、消费者消息确认机制来提供可靠交付功能。

- 生产者消息确认：生产者向RabbitMQ发送消息后，等待它回复确认成功；否则生产者向RabbitMQ重发该消息。此过程可以异步进行，生产者持续发送消息，RabbitMQ将消息批量处理后再回复确认；生产者通过识别确认返回中的ID来确定哪些消息被成功处理。开启生产者消息确认机制：

```
#开启发送确认
spring.rabbitmq.publisher-confirms=true
#发送失败退回
spring.rabbitmq.publisher-returns=true
```

- 消费者消息确认：RabbitMQ向消费者投递消息后，等待消费者回复确认成功；否则RabbitMQ重新向消费者投递该消息。该过程同样可以异步处理，RabbitMQ持续投递消息，消费者批量处理完后回复确认。可以看出，RabbitMQ/AMQP提供的是“至少一次交付”（at-least-once delivery）的策略，异常情况下消息会被重复投递或消费。开启消费者消息确认机制：

```
#开启 ACK
spring.rabbitmq.listener.simple.acknowledge-mode=manual
```

**持久化机制**

设置交换机、队列和消息都为持久化，它可以在服务器重启时保证消息不丢失信息，集群节点提供冗余能力，对解决Broker的单点故障至关重要。在RabbitMQ集群中，所有的定义都可以被冗余处理，例如交换器和绑定关系等，而队列只存在于一个节点上。对于队列而言，可以通过配置把队列镜像到多个节点上。

- 交换机的持久化（通过查看源码易知，默认是支持持久化的），代码如下：

```
@Bean
DirectExchange advanceExchange() {
    return new DirectExchange(exchangeName);
}
```

- 队列的持久化（通过查看源码易知，默认是支持持久化的），代码如下：

```
@Bean
public Queue advanceQueue() {
    return new Queue(queueName);
}
```

- 消息的持久化。当我们使用RabbitTemplate调用convertAndSend（String exchange，String routingKey，final Object object）方法时，默认是持久化模式。

**生产者**

当使用确认机制时，生产者从连接或者Channel故障中恢复过来，会重发没有被Broker确认签收的消息。如此一来，消息就可能被重复发送，可能是由于网络故障等原因，Broker发送了确认，但是生产者没有收到而已。或者消息根本就没有发送到Broker。正因为生产者为了可靠性可能会重发消息，所以在消费者消费消息处理业务时，还需要去重，或者对接收的消息做幂等处理（推荐幂等处理）。生产者增加确认机制非常简单，Channel开启Confirm模式，然后增加监听即可。

```
//选择确认机制
channel.confirmSelect();
//确认消息监听
channel.addConfirmListener(new ConfirmListener() {
    @Override
    public void handleAck(long deliveryTag， boolean multiple) throws IOException {
        System.out.println("消息已经 ack，tag: " + deliveryTag);
    }
    @Override
    public void handleNack(long deliveryTag， boolean multiple) throws IOException {
        //对于消费者没有 ACK 的消息，可以做一些特殊处理
        System.out.println("消息被拒签，tag: " + deliveryTag);
    }
});
```

说明：RabbitMQ还有事务机制（txSelect、txCommit、txRollback），也能保障消息的发送。不过事务机制是“同步阻塞”的，所以不推荐使用。而Confirm模式是“异步”机制。通过事务机制与Confirm模式的TPS性能对比，我们可以很明显地看到，事务机制是性能最差的。

RabbitMQ支持的4种交换器类型中，只有fanout模式不存在路由不到队列的情况。因为它会自动路由到所有队列中，跟绑定Key没有任何关系。所以，在满足业务的前提下，笔者建议，尽可能使用fanout模式的类型交换器。

DLX（Dead Letter Exchange，死信邮箱或死信交换机）就是一个普通的交换机，与一般的交换机没有任何区别。当消息在一个队列中变成死信时，通过这个交换机将死信发送到死信队列中。

我们可以通过DLX来解决这个问题，假设一些消息没有被消费，那么它就会被转移到绑定的DLX上。对于这类消息，我们消费并处理死信队列即可。

```
Map<String， Object> argsMap = new HashMap<>();
//死信队列
argsMap.put("x-dead-letter-exchange"， DLX_EXCHANGE_NAME);
//设置队列过期时间
argsMap.put("x-message-ttl"， 60000);
channel.exchangeDeclare(EXCHANGE_NAME， BuiltinExchangeType.DIRECT， true);
//死信的关系一定要在 queue 声明时指定，而不能在 exchange 声明时指定
channel.queueDeclare(Queue_NAME， true， false， false， argsMap);
```

**消费者**

只有消费者确认的消息，RabbitMQ才会删除它，不确认就不会被删除。所以，在消费端，建议关闭自动确认机制。应该在收到消息、处理完业务后，手动确认消息。消费者手动确认消息的实现代码如下：

```
DefaultConsumer consumer = new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag， Envelope envelope，
                               AMQP.BasicProperties props， byte[] body){
        System.out.println("死信队列接收的消息：" + new String(body));
        //手动确认消息接收成功
       channel.basicAck(envelope.getDeliveryTag()， false);
        //channel.basicNack(envelope.getDeliveryTag()， false， false);
    }
};
//推模式，并且关闭自动确认机制，即 autoAck=false
channel.basicConsume(Queue_NAME， false， consumer);
```

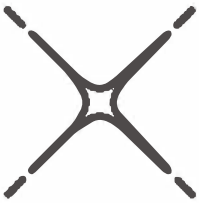
注意上面方法中void basicAck（long deliveryTag，boolean multiple）的第二个参数multiple。要说明这个参数的含义，首先需要清楚“deliveryTag”概念，即投递消息的唯一标识符，它是一个“单调递增”的Long型正整数。假设此次basicAck的tag为123456，如果multiple=false，则表示只确认签收这一条消息。如果multiple=true，则表示确认签收tag小于或等于123456的所有消息。

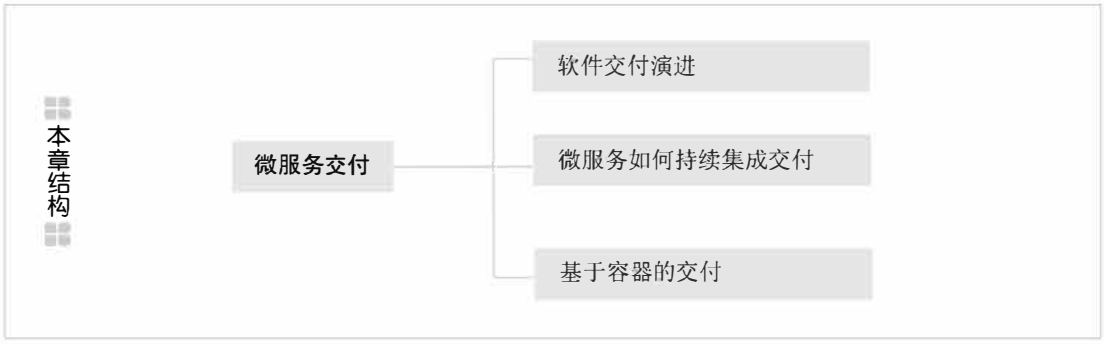
## 7.4 小结

在微服务架构下，我们强调要根据微服务的数据类型和业务场景选择合适的后端数据存储类型。对于微服务架构下分布式应用中的数据一致性管理，不推荐使用分布式事务，微服务数据架构通过放弃分布式网络的强一致性，来提升微服务之间的交互性能。另外，在微服务数据架构中，我们介绍了常见的TCC、Saga、可靠消息模式，可以作为保证数据之间最终一致性的解决方案。

第8章

# 微服务交付






软件交付作为软件工程中的一个重要环节，在过去的软件发展历程中一直不断地演进和优化。从最早的瀑布式软件开发过程到敏捷开发流程，从Scrum到DevOps，针对软件交付的方法论和技术也一直在延续发展。目前，基于容器技术的发展，软件交付的频率从按月交付转变为按天交付，交付平台工具和软件架构的发展使交付的成本越来越低，耗费的时间越来越少，不仅提高了工程师的交付、运维、生产效率，也极大地缩短了用户体验软件价值的周期。

持续集成指软件开发人员在开发完成后进行系统集成的过程，包括编译、发布、自动化测试等步骤。而持续交付建立在持续集成的基础上，它将最终产品发布给用户。

目前，越来越多开发者采用微服务进行软件构建和云原生架构的开发，对自动化运维工具的需求也越来越大。微服务拆分后，细粒度的服务足够小，可以独立部署，能做到对变化及时响应，持续集成和持续部署成为微服务规模化交付的基石。

本章我们将介绍软件交付的演进历史，同时会探讨微服务采用什么工具保证持续集成与持续交付，最后介绍脱胎于微服务架构思想的容器技术如何来帮助微服务持续集成和交付。

## 8.1 软件交付演进

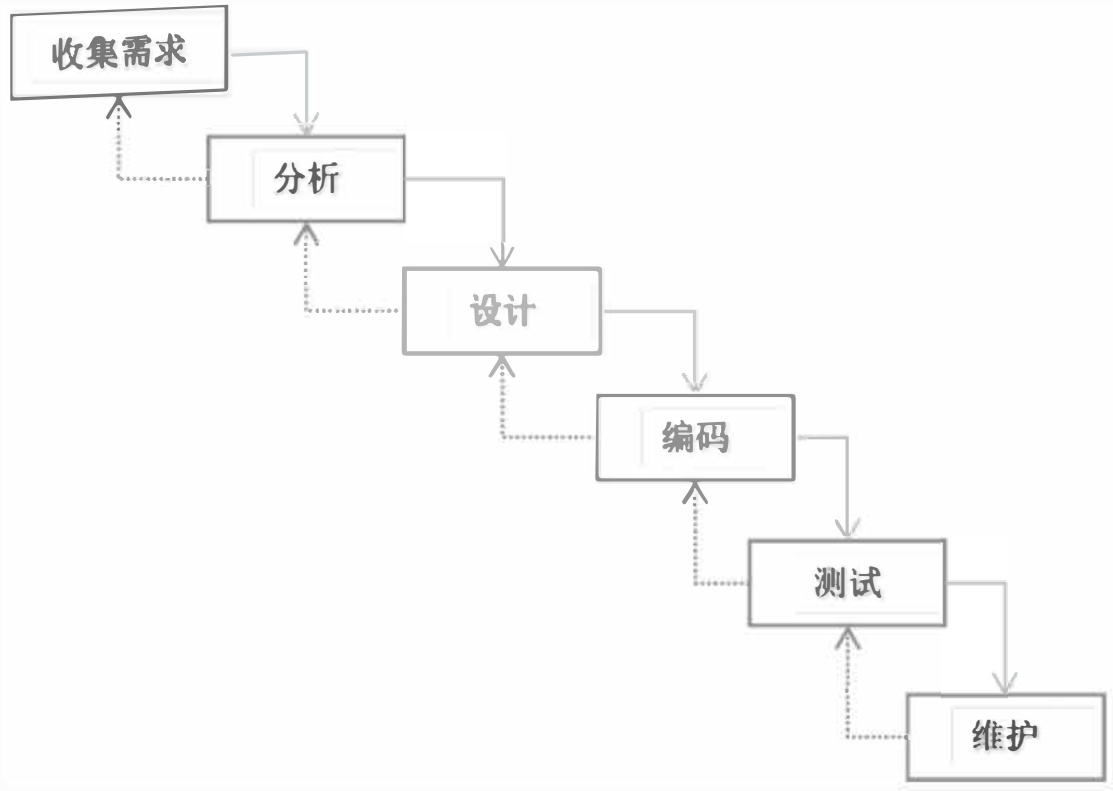
从宏观角度来说，软件开发活动同任何事物一样，都存在诞生、成长、成熟、衰亡的生命过程，一般称为“软件生命周期”。在软件发展演进历程中，软件过程模型和交付方式都发生了革命性的演进和进化。

### 8.1.1 软件过程模型

所谓软件过程模型，就是一种开发策略，这种策略针对软件工程的各个阶段提供了一套范形，使工程的进展达到预期目的。下面我们来看不同时期的软件过程模型。

**瀑布模型（Waterfall Model）**

瀑布模型提出了软件开发的系统化的、顺序的方法。其流程从收集需求开始，通过分析、设计、编码、测试和维护过程，最终提供一个完整的软件，并提供持续的技术支持，如下图所示。



模型特点：

- 阶段间具有顺序性和依赖性。必须等前一阶段的工作完成后，才能开始后一阶段的工作。
- 文档驱动的开发方式，没有合格的文档就是没有完成该阶段的任务。
- 前一阶段的输出文档就是后一阶段的输入文档，因此，只有前一阶段的输出文档正确，后一阶段的工作才能得到正确的结果。
- 每个阶段结束前都要对所完成的文档进行评审，以便及早发现问题，改正错误。但是事实上，越是早期阶段犯下的错误，暴露出来的时间越晚，排除故障改正错误所需付出的代价也就越大。因此，“及时审查”是保证软件质量、降低软件成本的重要措施。

**增量模型（Incremental Model）**

增量模型指把软件产品作为一系列增量构件来设计、编码、集成和测试。每次把新的构件集成到现有软件中时，所得到的软件必须是可测试的。它对软件过程的考虑是：在整体上按照瀑布模型的流程实施项目开发，以方便对项目的管理；但在软件实际开发过程中，却将软件系统按功能分解为许多增量构件，并以构件为单位逐个创建与交付，直到全部增量构件创建完成，并都被集成到系统中，交付用户使用。

模型特点：

- 当使用增量模型时，第一个增量往往是核心的产品。
- 客户对每个增量的使用和评估都作为下一个增量发布的新特性和功能。
- 该模型的每一个线性序列都产生软件的一个可发布的“增量”。

**基于构件的开发模型**

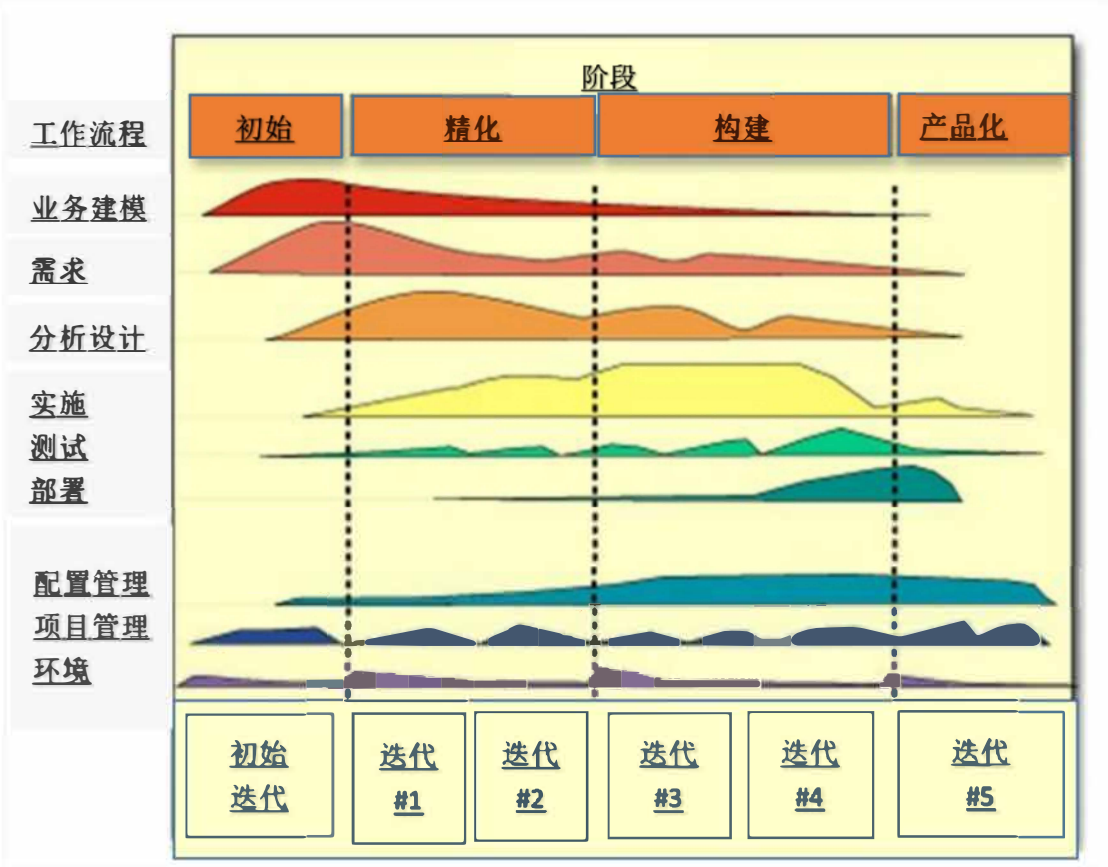
基于构件的开发方法指利用预先包装的构件来构造应用系统。构件可以是组织内部开发的构件，也可以是商品化的成品构件。基于构件的开发模型具有许多螺旋模型的特点，本质上它是演化模型，需要以迭代方式构件软件。其不同之处在于，基于构件的开发模型采用预先打包的软件构件来开发应用领域。

模型特点：

- 应用软件可由预先编好的、功能明确的产品构件定制而成，并用不同版本的构件实现应用的扩展和更新。
- 利用模块化方法，构件可复用。提高了开发效率，将复杂的难以维护的系统分解为互相独立、协同工作的构件，并努力使这些构件可反复重用。
- 突破时间、空间及不同硬件设备的限制，利用客户和软件之间统一的接口实现跨平台的互操作。

**统一过程模型**

统一过程模型（Rational Unified Process，RUP）是一种“用例和风险驱动，以架构为中心，迭代并且增量”的开发过程，由UML方法和工具支持。迭代是将整个软件开发项目划分为许多“袖珍项目”，每个“袖珍项目”都包含正常软件项目的所有元素，如需求、分析设计、实施、测试、部署，以及内部和外部的发布，如下图所示。




模型特点：

- RUP可以用二维坐标来描述。横轴通过时间组织，是过程展开的生命周期特征，体现开发过程的动态结构，用来描述的术语主要包括周期（Cycle）、阶段（Phase）、迭代（Iteration）和里程碑（Milestone）。
- 纵轴以内容来组织，是自然的逻辑活动，体现开发过程的静态结构，用来描述的术语主要包括各种项目活动及工作流程。

**敏捷开发（Agile Development）**

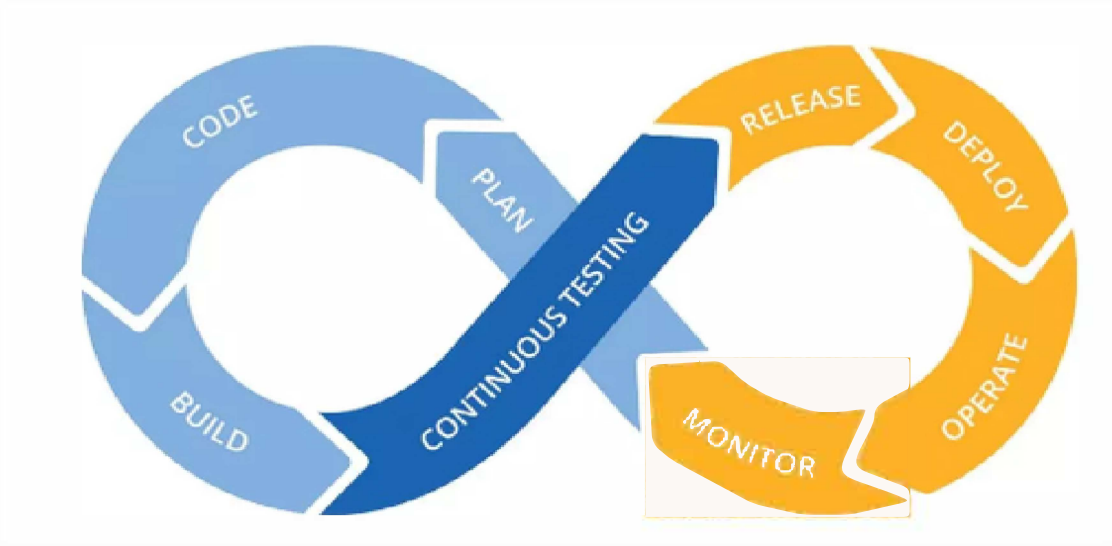
敏捷开发的总体目标是通过“尽可能地、持续地对有价值的软件的交付”使客户满意。通过在软件开发过程中加入灵活性，敏捷开发使用户能够在开发周期的后期增加或改变需求。敏捷开发的典型方法有很多，每一种方法都基于一套原则，这些原则实现了敏捷开发所宣称的理念（敏捷宣言）。

**持续集成、交付、部署（CI/CD）**

- 持续集成（Continuous Integration，CI）：指频繁地将代码集成到主干，目的是使产品能够快速迭代，同时保持高质量。其核心措施是，代码集成到主干之前，必须通过所有的测试

用例，只要有一个测试用例失败，就不允许集成到主干。

- 持续交付（Continuous Delivery，CD）：指在一个短周期内完成软件产品的产出过程，以保证软件可以稳定、持续地保持在随时可以发布的状况。它的目标在于让软件的构建、测试与发布变得更快更频繁。这种方式可以减少软件开发的成本与时间，减少风险。
- 持续部署（Continuous Deployment，CD）：是持续交付的下一步，指代码经过评审后，自动部署到生产环境。可以理解为QA测试完成后，应用准备部署上线生产环境，如下图所示。



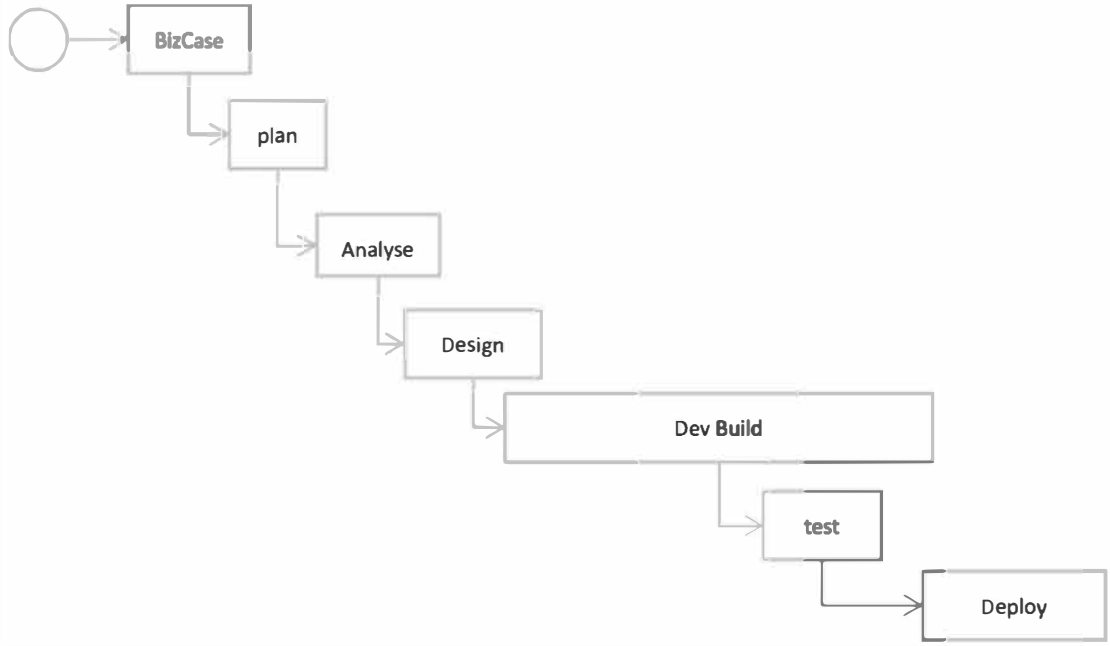

模型特点：

- 快速发现错误，每完成一点更新，就集成到主干，防止分支大幅度偏离主干。
- 持续部署的目标是，代码在任何时刻都是可部署的，可以进入生产阶段。
- 持续部署的前提是，能自动化完成测试、构建、部署等步骤。

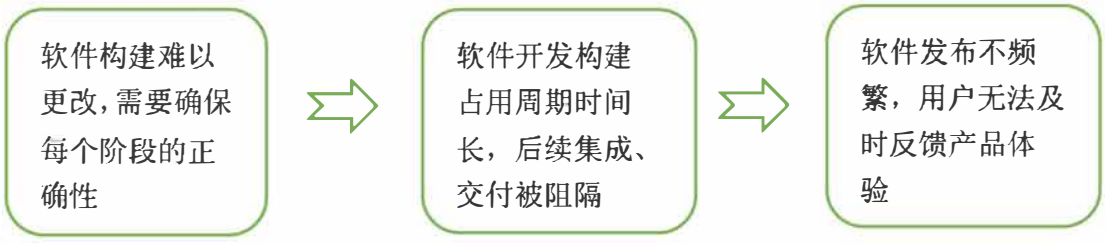
## 8.1.2 交付演进历程进阶

**早期的软件交付模式**

从软件过程模型的变化，我们可以发现，软件工程开发过程最早借鉴了建筑行业。建筑行业的特点是施工周期长，工种众多，涉及的专业分门别类，从开发商到工程设计师，从实施建设的工人到质量监理等。而建筑行业的工作流程是把项目分为不同阶段，每个阶段每个流程都由不同的专业人士负责，而且各阶段之间还要保持严格的顺序性，下图是瀑布模型交付阶段的分解过程。

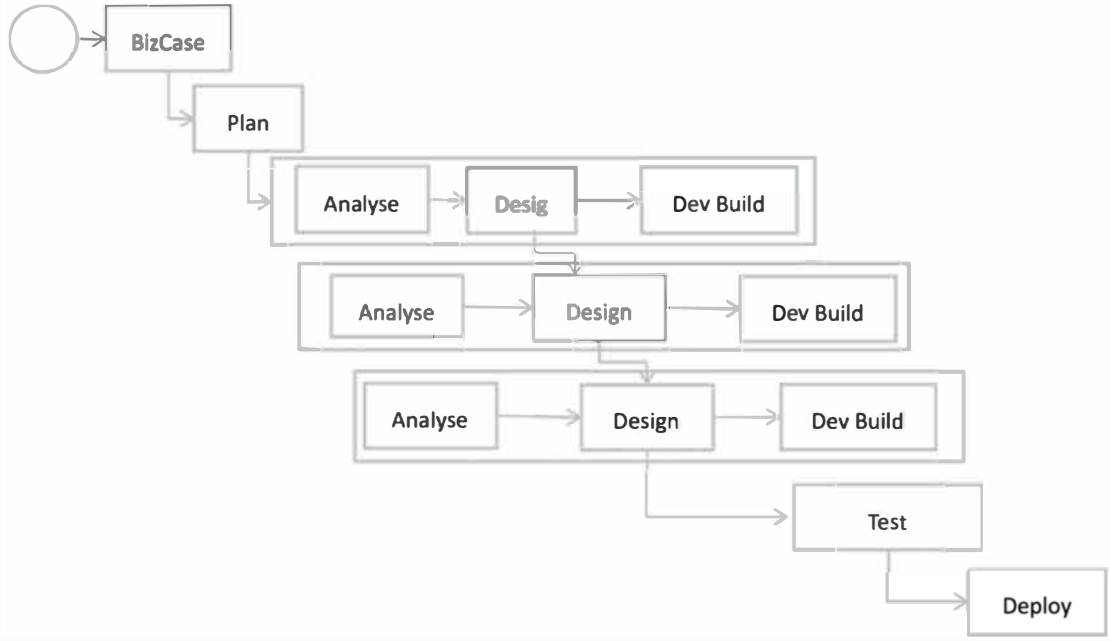



软件最早的瀑布模型正是借鉴的这一流程方式，成为软件开发早期（20世纪80年代）事实上的软件交付标准。而这一流程本身存在诸多问题，例如项目前期投入巨大，导致后期被取消，50%以上项目预算需要翻倍等。增量模型本质上是基于瀑布模型的构建方式，这一软件交付主要模式的特性如下图所示。

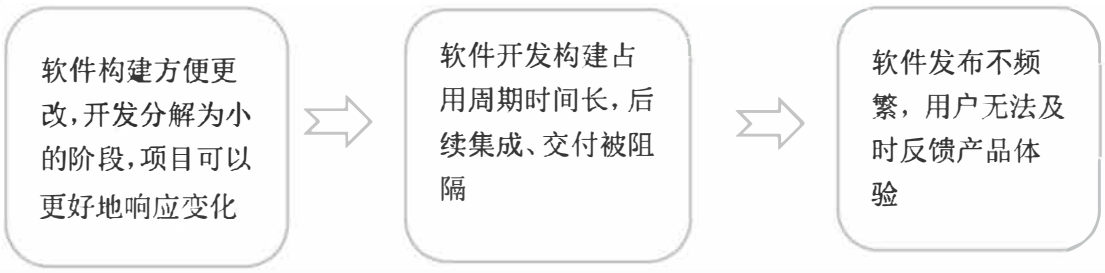



**敏捷开发模式**

在软件最后交付阶段，瀑布模型很难对前期设计或者构建进行根本性的改动，这是导致采用软件失败的根本原因。如果前期无法确切知道要构建怎样的产品，就无法保证每个阶段的正确性。而软件工程与建筑行业有很大不同的地方是，软件的变更是很容易的，而实体建筑的每个阶段都需要缜密的设计，否则返工的成本是巨大的。软件变更后的构建并不困难，所以我们可以把软件开发的迭代过程分割成更小的周期，这样就可以在分解后的每个小周期中检验和改进，有利于减少返工，如下图所示。




敏捷开发的方法论和过程模型强调，软件应该灵活应对变化。敏捷开发的核心价值观是：互动交流高于需求规格文档设计，软件架构师无法在软件开发的初期就设计完备的架构解决方案，而是将分析、设计分解到每个小的迭代开发周期中，给软件开发工程师赋予更高的责任，这样软件架构师不再是软件过程的瓶颈，而一线开发人员在面临挑战时可以提出更好和更切合实际的解决方案。在敏捷开发方法论中，项目的范围也是随着实际情况而变动的，时间和成本估计可以根据交付物通过大致推算得到，前期无法给出实际的价值评估和判断，敏捷开发的主要软件交付模式的特性如下图所示。



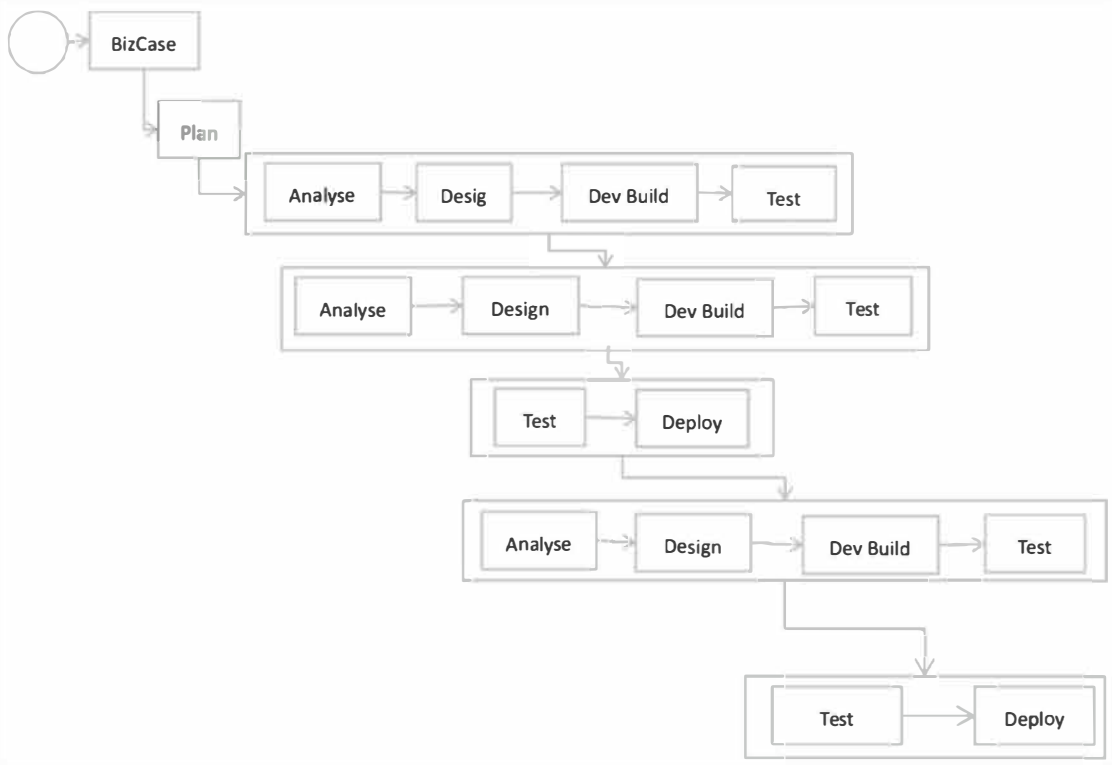

**持续集成/交付（CI/CD）**

我们看到敏捷开发方法论的软件过程模型解决了项目响应变化的问题，但客户还是无法体验产品，并获得实际的反馈，我们还需要面对频繁的测试集成和交付的挑战。

持续集成的好处是我们的代码在任何一个小的阶段中都是可供部署的，所以我们可以在更早的阶段验证软件的局部功能是否满足用户的期望，也可以及早发现软件潜在的问题，解决问题的成本也会更低，而软件集成和软件交付往往是可重复的工作，如果在每一次迭代中，我们都将这些测试、交付的工作自动化构建，将极大地提高生产率，开发工程师也可以将主要精力投入到业务的实际逻辑和业务功能中。

持续部署的好处是可以更快地发现软件的问题，也可以引入功能发布控制机制。我们可以把部分功能通过标识进行状态管理，例如A/B测试，进行量化的用户效果评估，这让我们可以更精准地

验证用户的需求，也可以进一步改善产品，如下图所示。




在持续交付过程中，需求以小批量的形式在团队的各个角色间顺畅流动，并在较短的周期内完成小粒度的频繁发布。实际上，频繁的交付不仅能持续为用户提供价值，还能产生快速的反馈，帮助业务人员制定更好的发布策略。该交付模式的特性如下图所示。

## 8.2 微服务如何持续集成交付

在微服务架构下，每一个微服务都是职责单一的，可独立部署单元，但是众多分散的微服务也带来了代码共享、微服务功能依赖、版本管理等复杂问题，微服务需要解决配置管理、服务持续集成、持续部署、快速交付过程中的一系列软件挑战。下面我们将讲解如何通过配置管理、持续集成、持续交付为微服务保驾护航。

## 8.2.1 配置管理概述

微服务架构下，应用被拆分为众多细粒度服务，对统一的配置管理的需求也随之增加。软件配置管理（Software Configuration Management，SCM）指通过执行版本控制、变更控制的规程，以及使用合适的配置管理软件，来保证所有配置项的完整性和可跟踪性。软件配置管理在整个软件生命周期中建立和维护项目产品的完整性，它的目标如下。

- 目标1：对软件配置的各项工作进行计划。
- 目标2：被选择的项目产品得到识别、控制并且可以被相关人员获取。
- 目标3：已识别出的项目产品的更改得到控制。
- 目标4：使相关组别和个人及时了解软件基准的状态和内容。

**软件配置项**

在配置管理流程中，IT组件及运用这些IT组件所提供的服务被称为软件配置项（SCI）。软件配置项包含了软件代码、代码的版本描述、文档数据（开发人员、用户）、软件所属组织等所有软件生产过程中产生的数据。这些配置项可以作为元数据信息对外暴露数据接口，其他第三方平台可以根据这些配置项的元数据信息调整系统中的相关配置。

**配置管理数据库（CMDB）**

配置管理数据库（Configuration Management Data Base，CMDB）的官方解释是，它用来存储与管理企业IT架构中设备的各种配置信息，包含了配置项全生命周期的信息及配置项之间的关系。CMDB开放的数据服务供各个业务系统调用，从而让业务系统具备统一的视图和配置信息。

其实微服务在自动化运维、标准化运维、DevOps中都离不开CMDB，可以说CMDB是运维体系的基石。有了配置信息数据库，后面各种标准、流程都可以建立在CMDB基础之上，实现真正的标准化、自动化、智能化运维，在节约运维成本的同时，降低运维流程混乱带来的操作风险。

**基线**

在开始软件项目开发前，我们需要一个稳定的代码版本。我们将这个稳定的代码版本称为代码基线，基线是软件进一步开发的基础，同时基线的确立是软件项目中的一个里程碑。当基线形成后，项目负责SCM的人需要通知相关人员，并且告知在什么位置可以找到基线的代码版本，这个过程可被认为是内部发布。至于对外的正式发布，更是应当从基线的版本中发布。下面总结一下基线配置的优势。

- 制定基线可以让所有开发人员的工作在基础代码层面保持一致和同步，为开发工作制定一个基点和快照。
- 制定基线可以让项目之间建立前后继承关系，而不会因为各自代码的基线不一致，导致无法追溯变更。
- 当代码变更有问题或者不可靠时，基线为分支代码提供了一个取消变更或者回退版本的操作。

在微服务架构中，基线模式通常把框架下的某个版本作为代码基线。以Spring Boot代码为例，Spring Boot的快速迭代，出现了众多版本。从Spring Boot 1.x到Spring Boot 2.x，不同的版本配置，让Spring Boot对依赖配套的底层组件和类库都有很大的差异。如果我们在开发过程中，没有统一的代码基线，在后续微服务的开发集成过程中，将出现代码版本冲突的风险及类定义缺失等底层代码不兼容问题。对于Spring Boot的版本兼容问题，我们可以参考官方网站发布的Release说明。

## 8.2.2 持续集成概述

持续集成（Continues Integration，CI）最早在1991年由Grady Booch命名并提出。XP极限编程中采用了CI的概念，并提倡每天不止一次集成。当时持续集成的目的就是能够通过采用自动化的手段和工具，来代替手动处理事务，实现项目的持续集成。而提出微服务概念的软件大师Martin Fowler对持续集成的定义则更加准确。

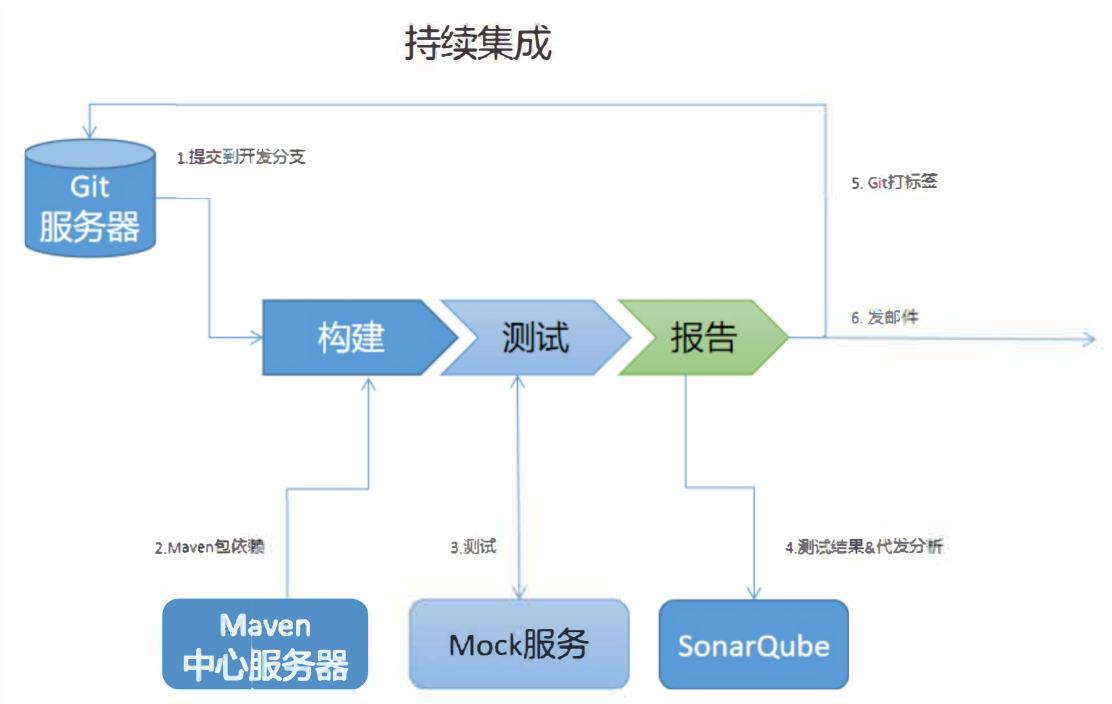
> 持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化构建（包括编译、发布、自动化测试）来验证，从而尽快地发现集成错误。

在微服务架构下，我们关注更多的是如何将一个大的单体拆分成多个微服务。分而治之虽然解决了系统的耦合问题，但是就像汽车的零部件被拆解后，如何再组装成一个完整的汽车一样，微服务同样需要通过相关调用、相互依赖、系统集成，最终完成整体对外功能。而持续集成正是将各个微服务的功能集成在一起来整体对外提供服务的。

同时，持续集成必须考虑针对这些独立部署的微服务，如何建立起代码仓库中的代码版本与CI构建和交付物版本之间的对应关系。在代码被更新后，保证每一次迭代和构建都可以经过质量验证，回归测试用例如何自动化执行，代码的频繁更新需要有一个良好的持续集成方案来高效地验证，并控制变更代码的副作用（Side-Effect）。

## 8.2.3 持续集成Pipeline

下图是持续集成的一个Pipeline（管道）方案，里面包含了我们通常会使用的代码集成工具和最佳实践。下面我们对持续集成使用的工具一一进行介绍。




**Git代码构建**

在微服务架构下，我们通过代码集成到主干分支完成一次CI构建，代码分支管理有多种不同的构建模式，具体如下：

- 单独代码库，一个CI构建：在这种方式下，微服务使用相同的代码库，并且使用一个单独的CI构建，这种模式容易产生代码冲突，一般不建议采用。
- 单独代码库，多个CI构建：这种模式通过创建多个子目录，每个子目录由单独的CI构建，这种模式在实际开发场景中使用较多。这种方式的问题是所有微服务代码都在相同的代码库，存在服务代码耦合隐患。
- 多个代码库，多个CI构建：每个微服务都有自己独立对应的代码库，这使微服务更加独立，代码管理更加独立。但缺点是跨服务的共享代码修改变得复杂，需要将共享代码上传到私服，通过版本工具进行代码升级更新。

对于使用传统SVN或者CVS的代码库构建的项目，使用第二种模式比较常见；而把Git作为代码库，划分为多个代码库则更适合管理，相比传统单一代码库，集成过程中产生代码冲突的可能性会大大降低，也降低了集成代码过程中处理冲突花费的精力和成本。

如果你的微服务使用了单一代码库和多个CI构建的模式，为了降低集成的风险，建议提高本地代码提交到主干分支的频率，减少由于代码变化过大引起的集成风险。

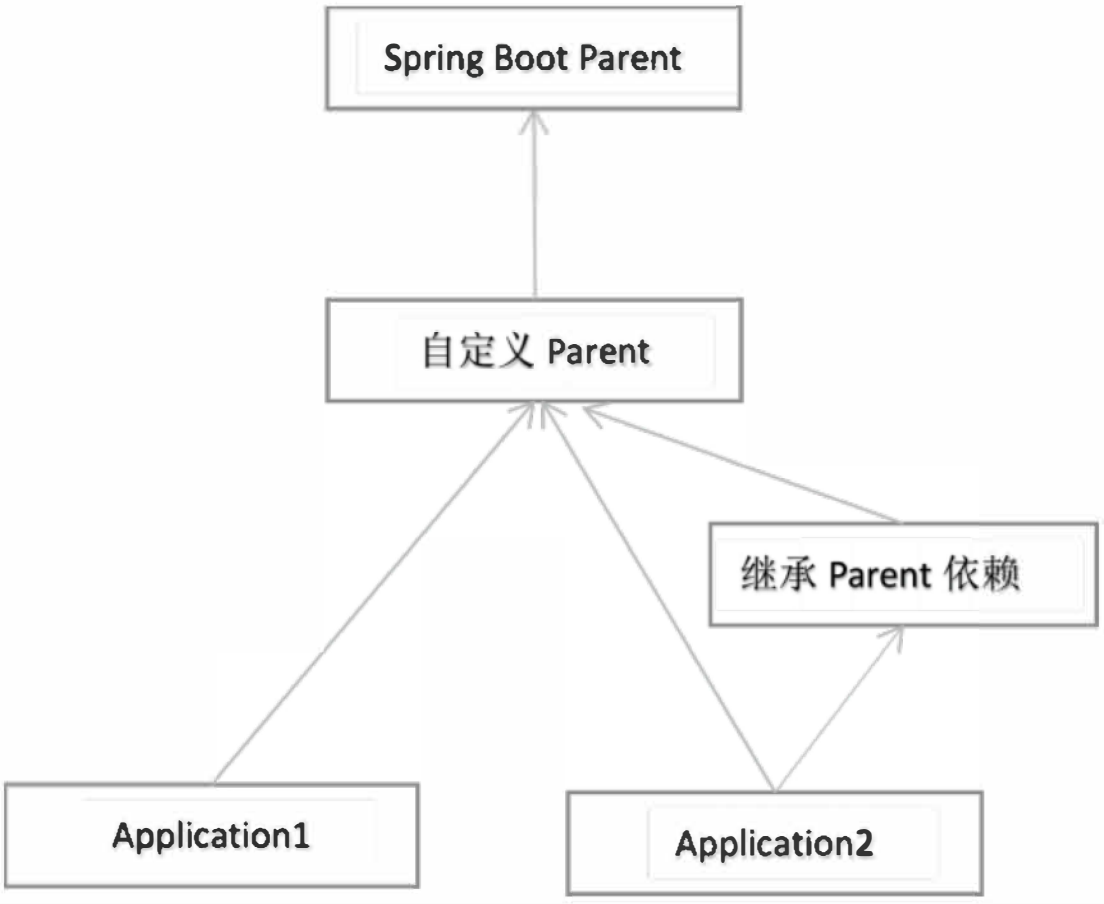
**Maven构建&Nexus私服**

在Spring Boot项目中，大部分公司目前还是以Maven作为主要的代码编译工具。Maven工具的安装十分简单，IDEA和Eclipse工具对Maven都有相应插件的支持。这里我们主要介绍Maven中的pom.xml文件。

最基本的pom.xml文件包含工程信息、Spring Boot父工程、属性配置、依赖包、构建插件。下面是pom.xml文件的一个简单模板。

```
<?XML...">
    <modelVersion>4.0.0</modelVersion>
    <!-- 工程信息 -->
    <groupId>com.xxx</groupId>
    <artifactId>demo</artifactId>
    <version>1.0-SNAPSHOT</version>
     <!-- Spring Boot 父工程 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.2.RELEASE</version>
    </parent>
    <properties>
        <!-- 相关属性、第三方依赖版本号 -->
    </properties>
    <dependencies>
        <!-- Spring Boot 依赖、其他依赖 -->
    </dependencies>
    <!-- 构建 -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

下面是微服务Spring Boot应用依赖关系管理的示意图。




- 最上层Spring Boot Parent：只有pom.xml文件，无代码。
- 中间层继承Parent依赖：包含pom.xml文件和一些具有通用性的代码，如工具类等。
- 最下层应用：Spring Boot应用工程，含有启动类，与单体应用类似。

**Nexus私服管理**

为避免在微服务下滥用包，应该统一管理第三方依赖的版本，同时为了方便mvn deploy操作，可以加上公司内部Maven私服的信息。

首先，执行代码上传私服命令。Maven私服常用的操作是将代码库安装到本地，并推送到Maven私服，代码如下：

```
mvn clean install deploy
```

其次，在项目pom.xml中设置的版本号添加Snapshot标识的都会发布为Snapshot版本（快照版），没有Snapshot标识的都会发布为Release版本。Snapshot版本会自动增加一个时间戳作为标识。如1.0.0-SNAPSHOT发布后变成1.0.0-SNAPSHOT-20180522.123456-1.jar，需要注意的是，Snapshot作为开发过程中使用的快照版是最为方便的，它可以随时更改代码，并重新被上传到Nexus。而Release发行版不能更改代码，如有修改提交失败，或因为版本冲突，则需要为Release指定新的版本号。下面我们对比下Snapshot与Release的区别。

- nexus-releases：用于发布Release版本。
- nexus-snapshots：用于发布Snapshot版本。

Release版本与Snapshot版本的定义如下。

- Release:1.0.0/1.0.0-RELEASE.
- Snapshot:1.0.0-SNAPSHOT.

在分发到远程Maven仓库之后，所有能访问该仓库的用户都能使用你的组件，然后我们需要配置pom.xml的distributionManagement来指定Maven分发构件的位置，在pom.xml中添加如下代码实现远程仓库的配置功能。

```
<distributionManagement>
    <repository>
        <id>nexus-releases</id>
        <name>Nexus Release Repository</name>
        <url>http://127.0.0.1:8081/repository/maven-releases/</url>
    </repository>
    <snapshotRepository>
        <id>nexus-snapshots</id>
        <name>Nexus Snapshot Repository</name>
        <url>http://127.0.0.1:8081/repository/maven-snapshots/</url>
    </snapshotRepository>
</distributionManagement>
```

**注意：** 虽然目前Maven仍然是Java代码构建的事实标准，但是Groovy的Gradle构建工具相比Maven摒弃了冗长的XML语法格式，同时在集成Maven诸多优点的前提下，Gradle在并行编译和编译效率方面都有更好的体验。在Spring Boot 2.3.0中，Spring Boot开始使用Gradle而非Maven进行构建。而Spring Boot团队给出的切换到Gradle的主要原因是减少构建所需要的时间。对于Spring Boot构建的复杂性，Gradle的构建缓存能力，以及增量和并行构建都是模块化CI构建中加快构建速度的重要因素。

**测试验证**

在持续集成中，当代码提交到主干分支后，需要单元测试、集成测试、功能测试。对于微服务而言，频繁的代码发布和快速迭代需要一个自动化回归测试方案，帮助验证增加的代码改动是否有副作用。

在系统集成测试中，对于一些依赖上下游服务的测试验证工作，还要模拟对端服务的API，这种方式就是俗称的“挡板测试”，其实就是Mock程序，用于客观条件限制下，在无法搭建整体测试环境的情况下，使用软件系统模拟依赖的服务功能。常见的使用场景是，当真实的调用可能产生资费时，就需要挡板技术解决系统集成时的代码验证问题。

下面我们使用JUnit实现Spring Boot程序的单元测试功能，从Spring Boot 2.2.0开始引入JUnit 5作为单元测试默认库。

Spring Boot 2.2.0之前，spring-boot-starter-test包含了JUnit 4的依赖，Spring Boot 2.2.0之后替换成了JUnit Jupiter。下面的实例用来演示使用Spring Boot整合JUnit的单元测试。

## 1.创建项目，添加Maven依赖

```
<!--Spring Boot 项目依赖的父项目-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<dependencies>
    <!--注入 Spring Boot 启动器-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--添加 JUnit 环境的 jar 包-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
</dependencies>
```

## 2.Dao层实例代码

```
package com.uniTest.dao;
import org.springframework.stereotype.Repository;
@Repository
public class UserDaoImpl {
    public void saveUser(){
        System.out.print("insert into user...");
    }
}
```

## 3.Service层实例代码

```
@Service
public class UserServiceImpl {
    @Autowired
    private UserDaoImpl userDaoImpl;
    public void saveUser(){
        userDaoImpl.saveUser();
    }
}
```

## 4.编写启动类

```
package com.uniTest;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class App {
    public static void main(String[] args){
        SpringApplication.run(App.class, args);
    }
}
```

## 5.编写测试文件，运行testSaveUser方法即可

```
/**
 * SpringBoot 测试类
 *
 * @RunWith:启动器 SpringJUnit4ClassRunner.class：JUnit 与 Spring 环境进行整合
 * @SpringBootTest(classes={Application.class}) //1.当前类为 Spring Boot 的测试类
 *@SpringBootTest(classes={Application.class}) //2.加载 Spring Boot 启动类,启动 Spring Boot
 * JUnit 与 Spring 整合@Contextconfiguartion("classpath:applicationContext.XML")
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {Application.class})
public class UserServiceTest {
    @Autowired
    private UserServiceImpl userServiceImpl;
    @Test
    public void testSaveUser(){
        userServiceImpl.saveUser();
    }
}
```

6.执行mvn clean install会自动触发JUnit测试工作，测试结果会输出到Console

```
Results :
Tests run: 10,  Failures: 0,  Errors: 0,  Skipped: 0
```

**代码静态检查**

SonarQube是一个用于代码质量检测管理的开放平台，SonarQube可以通过不同的插件对这些结果进行再加工处理，通过量化方式度量代码质量的变化。SonarQube不仅提供了对IDE的支持，可以在Eclipse和IntelliJ IDEA这些工具里联机查看结果；同时对大量的持续集成工具提供了接口支持，可以很方便地在持续集成中使用。另外，SonarQube的插件还可以对Java以外的其他编程语言提供支持。

- 编码规范：是否遵守了编码规范，遵循了最佳实践。
- 潜在的BUG：可能在最坏情况下出现问题的代码，以及存在安全漏洞的代码。
- 文档和注释：过少（缺少必要信息）、过多（没有信息量）、过时的文档或注释。
- 重复代码：违反了Don't Repeat Yourself原则。
- 复杂度：代码结构太复杂（如圈复杂度高），难以理解、测试和维护。
- 测试覆盖率：编写单元测试，特别是针对复杂代码的测试覆盖是否足够。
- 设计与架构：是否高内聚、低耦合，依赖最少。

在Jenkins中下载SonarQube插件，安装完成后，在系统管理->系统设置中，找到SonarQube servers模块，填写需要扫描的工程代码路径，构建成功后即可在SonarQube地址中登录访问，查看代码扫描情况。

**Jenkins实现自动化集成**

Jenkins是一个持续集成工具，也是我们持续集成整个流程的主角，从Git源码库中的代码拉取、单元测试验证到代码扫描报告的一系列流程，都可以通过Jenkins实现自动化执行。它可以根据设定持续定期编译，运行相应代码；运行UT或集成测试；将运行结果发送至邮件或展示结构。

目前，在现有的持续集成工具中，Jenkins已经成为持续集成事实上的标准，Jenkins的Pipeline插件已经定义了这些标准过程。Jenkins用Java编写，可在Tomcat等流行的Servlet容器中运行，也可独立运行。通常与版本管理工具、构建工具结合使用；常用的版本管理工具有SVN、Git，构建工具有Maven、Ant、Gradle。Jenkins的安装和初始化启动过程可以参考Jenkins官网。

下面是使用Docker构建Jenkins的CI Server的过程。

首先，制作Jenkins的DockerFile配置文件。

```
FROM jenkins:2.46.3
MAINTAINER Walter Fan
USER root
RUN mkdir /var/log/jenkinsRUN mkdir /var/cache/jenkinsRUN chown -R jenkins:jenkins
    /var/log/jenkinsRUN chown -R jenkins:jenkins /var/cache/jenkinsUSER jenkins
ENV JAVA_OPTS="-Xmx1596m"
```

其次，构建Jenkins Docker Image。

```
docker build -t jenkins-image .
```

然后，启动Jenkins CI容器服务。

```
docker run --restart always -v /workspace/jenkins:/var/jenkins_home -p 8080:8080 -p
    50000:50000 --name=jenkins-container -d jenkins-image
```

使用Docker部署Jenkins的好处是应用程序可以发布并直接部署Docker镜像，省去大量环境配置。我们在后面的Docker交付中，将对基于Docker的交付模式做进一步讲解。

下面，我们总结一下使用Jenkins实现持续集成的流程。

（1）开发者将自己分支的最新代码推到主干分支。

（2）从Git Master分支收到新的代码提交后，会触发Webhook启动相应的Jenkins job，启动Jenkins的整个CI流程。

（3）Jenkins从对应项目设置的代码仓库拉取相应编译环境的代码，完成代码的编译。

（4）Jenkins根据Pipeline插件功能增加集成测试任务，用于自动化回归测试。

（5）Jenkins根据事先设置的脚本任务和集成的SonarQube插件，完成对指定项目的代码扫描和代码分析报告，并将结果发送到指定邮箱。

下面是利用Jenkins的Pipeline DSL代码描述持续集成的示例流程。

```
Pipeline {
  Agent any
   stages {
      stage('Build') {
        steps {
            sh 'make'
        }
     }
     stage('Test'){
        steps {
            sh 'make check'
            junit 'reports/**/*.XML'
        }
     }
     stage('Deploy') {
        steps {
            sh 'make publish'
        }
     }
}
```

总结一下，上述过程原本需要开发人员手动切换到不同环境进行例行工作，现在通过Jenkins实现了常规化、自动化的持续集成工作，有效避免了开发人员集成代码的重复工作。上述工作完成后，就实现了从代码提交到代码集成编译、单元测试、代码检查相关的集成工作。利用其Pipeline式的管理方式，Jenkins不仅可以进行持续集成，还可以进一步实现后续的代码部署和交付任务。

## 8.2.4 持续交付概述

持续交付（Continuous Delivery，CD）是一种软件工程方法，开发者可以用快速、自动化和可重复的方式从源码库部署交付软件，确保交付过程的标准化和自动化。它更加注重随时随地提供可靠发布的能力。

如果说持续集成是让我们的团队通过快速迭代提交代码到共享代码库中，提前发现模块集成中的问题，那么持续交付就是通过构建自动化的交付平台，允许团队更加频繁地交付可工作的软件产品，从而能够更快地得到用户对产品的反馈和对产品价值的体验。

持续交付最早由Jez Humble和David Farley提出，并在《持续交付：发布可靠软件的系统方法》一书中对其实现进行系统说明。而本书未对持续交付做明确定义，目前对持续交付比较认同的定义来自DevOps Handbook上的定义。

> 持续交付指的是，所有开发人员都在主干上进行小批量工作，或者在短时间存在的特性分支上工作，并且定期向主干合并，同时始终让主干保持可发布状态，并能做到在正常工作时段里按需进行一键式发布。开发人员在引入任何回归错误时（包括缺陷、性能问题、安全问题、可用性问题等），都能快速得到反馈。一旦发现这类问题，就立即加以解决，从而保持主干始终处于可部署状态。

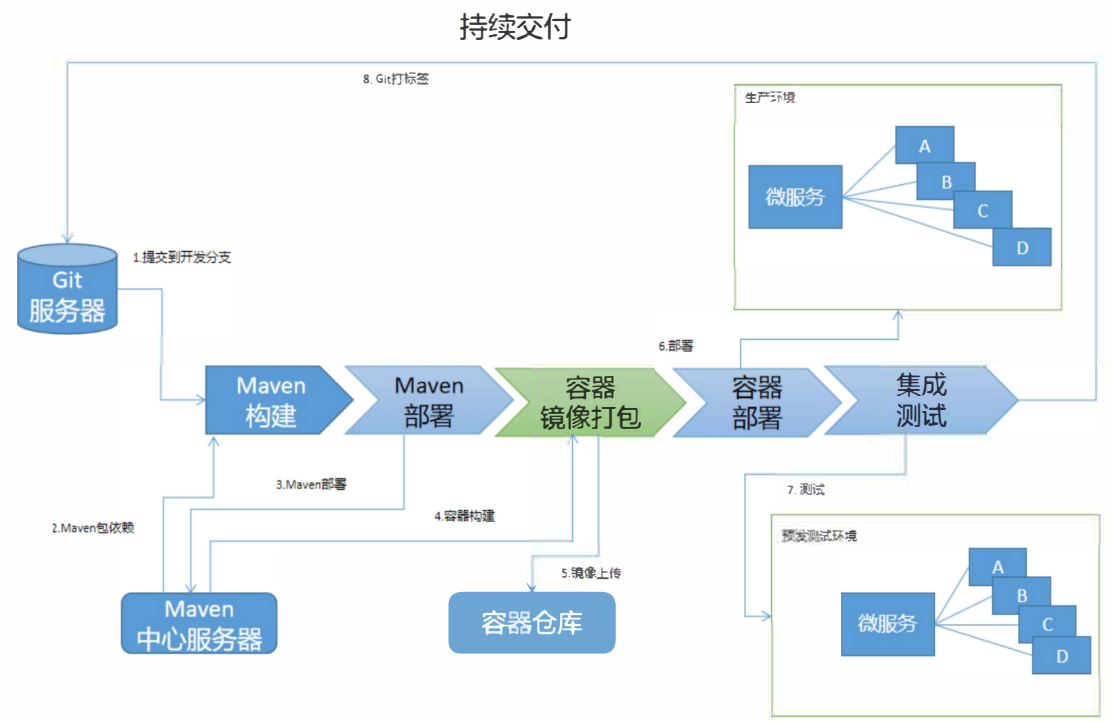
首先，微服务的流行本身就是在持续交付方法论下发展起来的。在单体架构下，所有模块集成在一个代码库下，往往一个小的改动会牵一发而动全局，代码的耦合影响产品对外的发布交付速

度，如果不进行微服务的拆分，则系统的持续交付将面临一系列问题，而将单体分解成独立、自治、可部署的微服务，单独交付部署，就可以减少代码之间的耦合及相互影响。同时，微服务的交付周期短、功能单一、扩展性好、技术多样性等特征，都是持续交付所需要的。

同时，微服务的发展需要持续交付的支撑，微服务在拆分后面临服务众多、版本众多的问题，如果没有一套自动化的代码构建、测试、部署打包工具，则很难应付微服务在规模化部署下的持续交付。另外，持续交付的价值不仅仅在于提升服务交付的效率，它还通过指定统一的标准、规范的流程、自动化平台，影响着软件研发到运维的整个生命周期。而持续交付最终将软件的研发流程进化为DevOps软件开发模式，实现端到端的软件价值交付。

## 8.2.5 持续交付Pipeline

下面是持续交付Pipeline的流程图。可以看出，持续交付的前提是持续集成，也可以说持续交付包含持续集成过程。




持续交付的核心是构建一个自动化的流水线，而这个持续交付工作平台可能贯穿了软件项目的整个生命周期。它不仅要串联起软件工程代码的提交集成、测试验证、打包管理、镜像交付，同时将影响整个公司组织端到端的业务，从开发测试到部署交付的整个流程过程。

下面我们总结一下实现持续交付的核心工作流程。

- 准备工作
  - 集成工具的技术选型，在持续集成一节中，源码的管理、持续集成工具的选择、自动化测试等软件工具的选择都将是后续完成自动化流程的关键步骤。
  - 部署载体的选择，目前容器已经成为微服务部署实施的标准运行时环境，而对于有状态服务或者存储相关服务，我们需要提前规划好交付方式的步骤。
  - 软件工程和组件结构可以帮助更频繁地发布服务，而不会对用户产生影响。

- 软件监控和服务治理是持续交付的重要机制，在构建持续交付平台前，就应该为服务运维中需要的日志管理、应用及容器状态、指标性能等需求做配套设计开发。
- 组织结构、业务流程的变化，需要支持持续交付。

● 自动化一切

- 自动化的代码构建和打包。
- 自动化的测试集成。
- 自动化的发布机制。
- 测试环境下服务的自动化部署。
- 生产环境下服务的自动化部署。

● 构建持续交付流水线

我们在持续集成中，说过利用Jenkins的Pipeline可以实现自动化的流水线流程。如果你所在公司的人力不是很充足，则可以使用开源版本的Jenkins构建流水线平台。对于定制化的流水线，我们可以把软件生命周期中从代码到生产环境的所有过程全部可视化地展示出来，实现端到端的持续交付，实现持续交付过程中的增强功能。

- 实现环境定制化配置管理。
- 实现版本的回滚。
- 实现灰度发布和版本切换。
- 记录操作日志审计。
- 可以实现网络的监控和拓扑管理。

## 8.3 基于容器的交付

基于容器的交付可以说是软件交付方式的一次技术革命。容器技术将服务交付依赖的底层基础设施进行了标准化，屏蔽了多样化的环境差异。对开发人员来说，容器技术可以把对交付物的质量保证及测试验证工作进一步提前到开发集成阶段，缩短代码到实际交付物的距离；对运维人员来说，服务的部署、扩容、回滚更为方便；同时容器技术成为规模化微服务部署、DevOps、不可变基础设施从思想到落地的关键技术环节。

### 8.3.1 Docker概述

本节是对Docker的概述，将从Docker的概念、容器与虚拟机的区别、容器交付的优势等方面进行说明。

**Docker的概念**

首先，容器技术本质上是对计算机资源的隔离与控制，可以理解为一种沙盒技术。沙盒就是它能够像集装箱一样，把应用及应用依赖的基础设施一起定义、封装，打包为镜像，这样应用就可以独立部署、复用；同时应用与应用之间相互隔离。

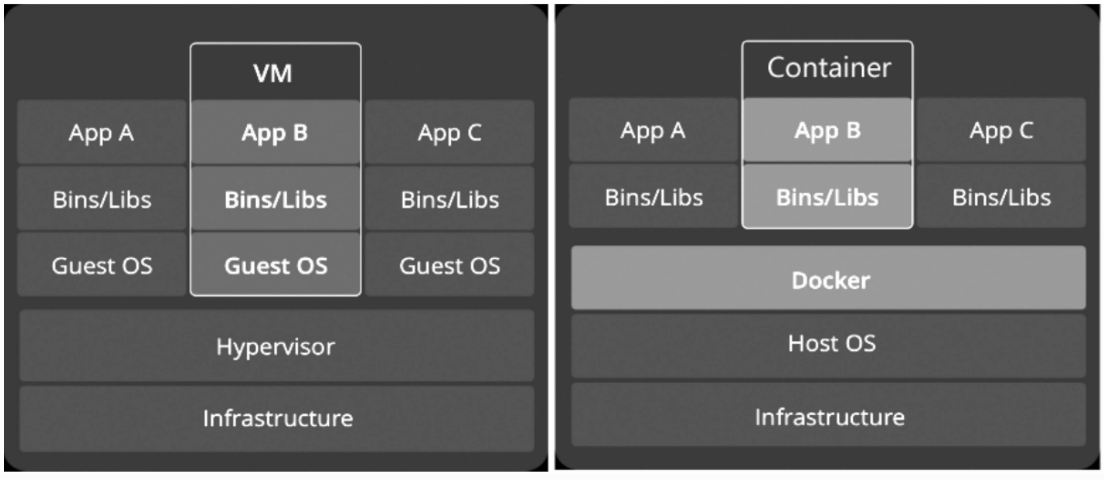
而Docker是一个开源的容器引擎，是容器概念的落地实现，早期容器内核是基于LXC（LinuX Container）实现容器创建和管理的，开发者可以将应用及其依赖的软件打包在一个可移植的镜像中，镜像可以被推送到私有仓库或者远程仓库；运维人员可以从仓库中加载镜像并启动容器。下面是维基百科对Docker的定义。

> Docker是一个开源的软件项目，可以自动化部署应用程序在软件容器下的工作，借此在Linux操作系统上，提供一个额外的软件抽象层，以及操作系统层虚拟化的自动管理机制。Docker利用Linux核心中的资源分离机制，例如Cgroup，以及Linux核心命名空间（NameSpace），来建立独立的软件容器（Container）。这可以在单一Linux实体下运作，避免启动一个虚拟机造成额外负担。

**容器与虚拟机的区别**

容器技术与传统虚拟机技术虽然都是虚拟化技术，但还是有本质差别的。传统的虚拟机是在物理机的层面上增加了一层虚拟化技术，这种虚拟化技术上的每一个虚拟机都拥有一个完整的操作系统，有很好的隔离性。在技术实现上，虚拟机技术由Hypervisor（虚拟机监视器）来负责创建虚拟机，这个虚拟机必须运行一个完整的虚拟操作系统才能执行用户的应用进程，这就不可避免地带来了额外的资源消耗和占用。此外，用户应用运行在虚拟机里，它对宿主机操作系统的调用就不可避免地要经过虚拟化软件的拦截和处理，对计算资源、网络和I/O的损耗非常大。同时，每个虚拟机内部都包含了操作系统，所以启动速度相对比较慢。

如下图所示是容器（Container）（右侧）与虚拟机（VM）的差异比较。容器本质上是一个进程，容器之间共享了操作系统内核。



在Linux操作系统中，Cgroup和NameSpace两种技术是实现Docker的关键。容器没有完整的操作系统层，并且是进程级别的，所以它的启动速度是秒级启动，占用资源相对虚拟机也比较低。

- Cgroup全称Control Group。Linux操作系统通过Cgroup可以设置进程使用CPU、内存和I/O资源的限额。Cgroup可以对容器进行层次化的分组，并可以按组实现资源限制和控制策略。
- NameSpace在Linux中是管理Host的全局唯一的资源，它可以实现容器的工作空间与资源隔离，容器中看到的文件系统、网卡、进程等资源，通过NameSpace技术可以使这些资源看上去都是容器自己的。

**容器交付的优势**

通过上面的介绍，我们可以看到，使用容器进行交付有诸多优势，具体如下。

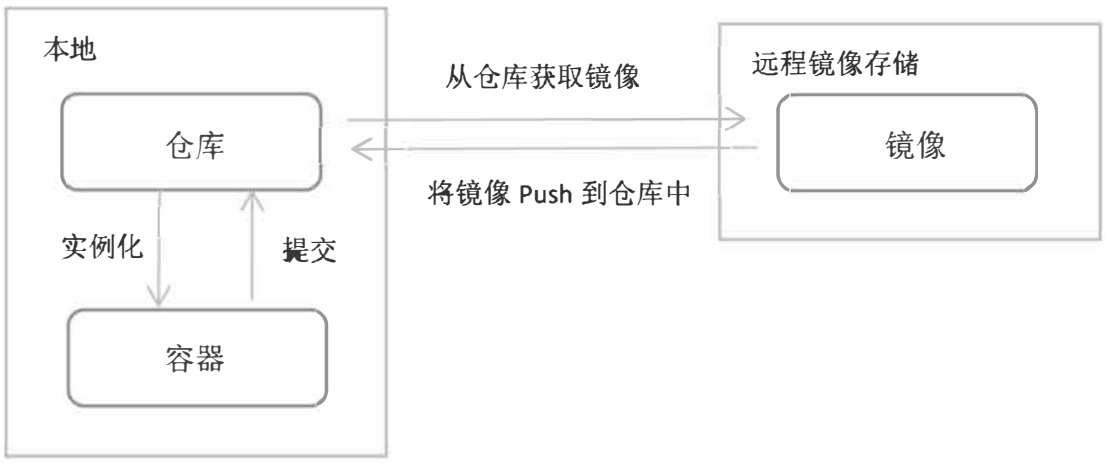
- 屏蔽环境差异：在《SRE Google运维解密》一书中，来自Google的运维人员总结到，过去生产事故中有一半事故产生的原因，都是生产环境配置的变更，或者是由测试、生产环境不一致导致的。由于容器技术基于开放的标准，因此相比基于代码的交付，基于容器的交付能够标准化服务，对整个服务运行环境进行打包交付。
- DevOps一体化：在没有容器之前，软件的交付往往需要通过开发人员手写各种脚本完成服务的启动和部署，而这些脚本在不同环境下的差异很容易在开发人员与运维人员之间产生误解和分歧。容器技术的出现统一了开发人员与运维人员的领域俗语，通过标准界面打通了开发人员与运维人员之间的技术壁垒。同时基于容器镜像的测试和部署也大大简化了整个CI/CD过程。
- 高效的资源利用率和隔离特性：容器之间共享底层操作系统，相比虚拟机技术，性能更加优越，系统负载更低。可以充分利用系统资源，通过NameSpace、Cgroup技术保证容器之间的隔离性。
- 跨平台及云原生特性：容器技术实现了操作系统层面上的“一次构建，随处运行”的理念，通过设定一套标准化的配置方式，提升了容器的跨平台特性和可移植性。同时，云平台通过提供对容器的支持，让容器成为云原生平台的标准化交付技术。
- 规模交付：容器技术是微服务架构规模化部署的最佳载体，容器运行的基于进程的概念、隔离特性及单一职责特性都与微服务架构相辅相成。在微服务规模化部署场景下，我们需要通过容器的编排技术保证微服务之间的依赖特性、部署协调和运行多集群微服务。容器编排技术保证微服务在出现故障和性能瓶颈时，水平扩展可以弹性伸缩，资源调用编排技术保证微服务架构的高可用。

## 8.3.2 Docker的原理

Docker有三个核心组件，掌握这三个组件的概念有助于我们进一步了解Docker的工作机制。

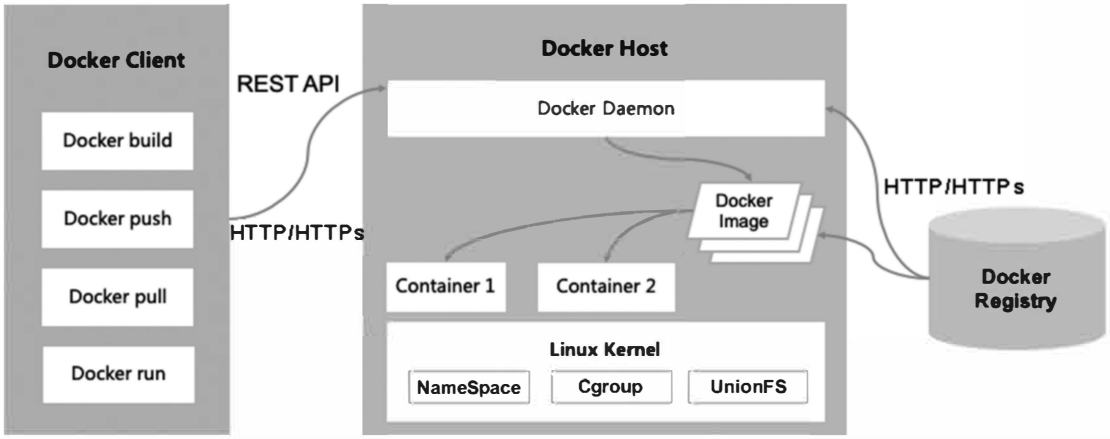
- 镜像：Docker Image，它是容器运行所需要的静态二进制文件和依赖包的集合，可以将它理解为一个面向Docker的只读模板，容器镜像基于分层的联合文件系统（UnionFS）实现。用户可以根据需求，通过DockerFile定制容器镜像，同时Docker提供了对镜像的各种API操作命令实现镜像版本管理的功能。
- 容器：Docker Container，是从镜像创建的应用，是镜像的动态运行实例，Docker利用容器来运行和隔离应用。镜像自身是只读的，容器从镜像启动时在镜像最上层创建一个可写层，镜像本身保持不变，容器启动后以进程的方式运行。另外，Docker支持一个容器对应一个进程的方式，而这种方式也非常适合以单进程运行为主的微服务架构。
- 仓库：Docker Repository，主要用来存放镜像，可以分为公有仓库和私有仓库。当用户创建了自己的镜像之后，就可以使用推送的方式将它上传到指定的公有仓库或私有仓库。这样用户下次在另一台机器上使用该镜像时，只需将其从仓库拉取下来就可以了。

这三个组件组成了Docker的整个生命周期，下图是三者之间的关系。




Docker采用C/S架构，下面是Docker的主要运行过程和原理图，同时涵盖了Docker的核心组件。

- 首先，Client通过相应的Docker命令以及HTTP或REST API等方式与Docker Daemon实现Docker服务的使用与管理。
- 其次，Docker Daemon作为服务端（部署在本地或者远程），负责监听Client的请求并管理Docker对象（容器、镜像、网络等），Docker Image提供容器运行所需的所有文件。
- Docker Daemon通过访问容器镜像仓库，负责Docker镜像的存储管理。镜像仓库可以用DockerHub或者自建私有镜像仓库。可以通过Docker push或pull操作往镜像仓库上传或下载镜像。




## 8.3.3 Docker构建部署过程

Docker构建部署过程主要包括下面三个步骤。

- 构建：通过DockerFile来描述应用依赖的运行环境，包括操作系统、启动端口、执行脚步，通过编译运行DockerFile来生成Docker镜像，并放在本地仓库中。
- 装载：将镜像推送（Push）到远端仓库。
- 部署运行：从仓库拉取（Pull）镜像，并创建容器实例，启动部署容器实例。

下面我们以金融公司开源微服务网关产品——SIA-Gateway（GitHub上的开源项目）为例，演示如何编写DockerFile，以及Docker构建部署过程。

**网关项目源码结构**

```
├── sia-gateway-admin-buildcomponent   网关 Admin 组件集合
│   ├── sia-gateway-admin              网关 Admin 监控系统组件
│   ├── sia-gateway-synchspeed         网关对下游服务实时感知组件
.
├── sia-gateway-admin-display          网关系统前端代码
.
├── sia-gateway-buildcomponent         网关 core-buildcomponent
│   ├── sia-gateway-core               网关 Core 服务
│   │   ├── sia-gateway-base           网关 Core 服务基础依赖
│   │   ├── sia-gateway-reactive
├── Dockerfile  网关 DockerFile
├── docker-build.sh  构建镜像脚本
├── docker-run.sh  启动容器和服务
├── docker-start.sh  网关在 Docker 中的启动脚本
├── pom.xml    Maven 编译
```

**说明：** SIA-Gateway项目采用前后端分离架构，由Admin、前端（Vue）、核心（Core）三个模块组成。在Docker示例代码中，我们将三个模块整体打包到一个SIA-Gateway镜像中。

**DockerFile解析**

```
FROM centos:latest
MAINTAINER peihuabest@163.com
#添加 yum 源
RUN /bin/bash -c 'curl -O /etc/yum.repos.d/CentOS-Base.repo
http://xxxx[1].com/repo/Centos-7.repo;yum clean all;yum makecache'
RUN yum install -y gcc gcc-c++ glibc* automake autoconf libtool make libXML2-devel pcre-devel
openssl openssl-devel libicu-devel file libaio libaio-devel libXext libmcrypt libmcrypt-devel
numactl unzip zip groupinstall chinese-support vixie-cron crontabs  telnet-server
telnet.*
java-1.8.0-openjdk lsof && localedef -c -f UTF-8 -i zh_CN zh_CN.utf8

#添加第三方依赖包
ADD ./third-libary/  /opt/
RUN /bin/bash -c 'rpm -ivh /opt/tcl-8.5.13-8.el7.x86_64.rpm;rpm -ivh
/opt/tcl-devel-8.5.13-8.el7.x86_64.rpm;rm -rf /opt/tcl*;cd /opt/;useradd nginx -s
/sbin/nologin;chown -R nginx.nginx /opt/nginx'
# ENV LANG zh_CN.UTF-8
ENV LC_ALL "zh_CN.UTF-8"

#添加 AdminServer 后端工程
ADD sia-gateway-admin-buildcomponent/target/gateway_admin_1.0.zip  /app/jar/ROOT/
# add gateway core
ADD sia-gateway-buildcomponent/target/gateway_1.0.zip  /app/jar/ROOT/
#添加 Admin 前端工程
ADD  sia-gateway-admin-display/dist   /app/jar/ROOT/dist/
#添加启动脚本
ADD docker-start.sh  /app/jar/ROOT/
#运行
RUN /bin/bash -c 'cd /app/jar/ROOT;chmod +x *.sh;unzip gateway_1.0.zip;unzip
gateway_admin_1.0.zip;rm -rf  gateway_1.0.zip;rm -rf  gateway_admin_1.0.zip'
```

对上述DockerFile说明一下，对Java开发程序员来说，大多数人都对Maven集成工具比较熟悉。其实DockerFile的编写过程与Maven有类似的地方，可以按照打包镜像工作的属性大致分为三部分，我们将这三部分总结如下。

- 首先，选择基础镜像及yum源定义，如果部署机器能够使用阿里yum源，则此步忽略；应用运行加载需要的基础设施和依赖库。
- 其次，对于yum源中不存在的第三方依赖软件库，可以从本地通过ADD命令加载，并为镜像定义全局环境变量。
- 最后，加载已经打包好的应用软件包到镜像，也可以指定应用工作目录，以及需要使用的挂载点（Volume）等；定义镜像运行时需要执行的相关解压包或者授权等相关指令。

**构建镜像**

使用DockerFile构建镜像，需要注意的是，首先要把DockerFile依赖的应用包进行打包。对于SIA-Gateway项目而言，可以在根目录下执行mvn install脚本进行代码打包。打包完成后，执行docker.build.sh构建网关镜像。下面是构建镜像的代码。

```
#!/usr/bin/env bashdocker
build -t reg.caiwu/sia/gateway:v1 .
```

镜像制作完成后，可以使用docker push命令将镜像推送到私有仓库或者远程仓库，Docker官方提供了一个搭建私有镜像的仓库，只需把镜像下载下来，运行容器并暴露5000端口。Docker Hub是Docker官方维护的容器镜像，可以使用docker search命令搜索所有镜像，本节省略Docker Hub的登录访问过程。

● 将SIA-Gateway推送到Docker私有仓库的命令如下。

```
docker Push localhost:5000/reg.caiwu/sia/gateway:v1
```

● 将SIA-Gateway推送到Docker Hub的命令如下。

```
build Push reg.caiwu/sia/gateway:v1
```

● 可以通过docker pull命令验证从私有仓库下载容器镜像。

```
build pull reg.caiwu/sia/gateway:v1
```

**运行和部署容器镜像**

容器镜像构建好后，就可以运行和部署容器镜像，使用docker-run.sh可以从本地加载部署运行SIA-Gateway镜像。docker-run.sh脚本如下。

```
#!/usr/bin/env bash
str=$(cd "$(dirname "$0")";pwd)
echo ${str}/sia-gateway-admin-buildcomponent/config
docker run --name gateway-test -d reg.caiwu/sia/gateway:v1
-v ${str}/sia-gateway-admin-buildcomponent/config:/app/jar/ROOT/gatewayadmin/config
-v ${str}/sia-gateway-buildcomponent/config:/app/jar/ROOT/gateway/config
-v /etc/localtime:/etc/localtime
-p 18086:18086 -p 8080:8080 -p 8040:8040
--restart=on-failure:10
/bin/bash  -c " /app/jar/ROOT/docker-start.sh "
```

对docker run命令说明如下。

● docker run--name gateway-test-d reg.caiwu/sia/gateway:v1

使用docker镜像reg.caiwu/sia/gateway：v1以后台运行模式启动一个容器，并将容器命名为gateway-test。

● -v参数和-P参数

使用-v参数可以将主机的一个目录映射到容器的目录，例如主机目录${str}/sia-gateway-buildcomponent/config到容器目录/app/jar/ROOT/gateway/config。使用-P参数可以将容器的8080端口映射到主机的8080端口。

● /bin/bash-c命令

docker run可以使用/bin/bash-c命令执行一个后台任务，这样容器就不会退出了。目前-c参数后紧跟的参数是/app/jar/ROOT/docker-start.sh（对应执行的后台任务），而docker-start.sh脚本文件是在DockerFile中通过ADD命令打包到镜像中的。

**容器载入后运行脚本**

我们来看下docker-start.sh。这个命令是容器启动后加载docker run命令执行的容器内部网关程序的启动脚本，这里我们启动了Admin和Core中的多个服务。需要注意的是，在这个脚本中，至少需要有一个服务的进程是前台运行的，保证Docker不会销毁退出。至此，通过以上步骤实现了微服务应用基于Docker启动部署的过程。

```
#!/usr/bin/env bash
/opt/nginx/sbin/nginx -c /opt/nginx/conf/nginx.conf
cd /app/jar/ROOT/gatewayadmin/bin
chmod +x *.sh
/app/jar/ROOT/gatewayadmin/bin/start_gateway_admin_test.sh
/app/jar/ROOT/gatewayadmin/bin/start_gateway_service_test.sh
/app/jar/ROOT/gatewayadmin/bin/start_gateway_synchspeed_test.sh
/app/jar/ROOT/gatewayadmin/bin/start_gateway_stream_test.sh
/app/jar/ROOT/gatewayadmin/bin/start_gateway_monitor_test.sh
echo "启动网关核心"
cd /app/jar/ROOT/gateway/bin
chmod +x *.sh
/app/jar/ROOT/gateway/bin/start_gateway_test.sh
echo "启动完毕"
```

**DockerFile的常用指令**

● FROM指令：是整个DockerFile的入口，必须是第一条指令。代表新制作镜像的基础镜像。

【格式】

```
FROM scratch
```

Docker中存在一种特殊的情况，就是不以任何基础镜像为基准。基础镜像可以自己制作，也可以从开源的仓库拉取，例如Docker Hub或者国内阿里云的免费仓库。

● COPY指令：用于将宿主机文件复制到镜像内的指定路径。

【格式】

```
COPY <源路径>... <目标路径>
```

或

```
COPY ["<源路径 1>", ... "<目标路径>"]
```

● ADD高级复制：ADD的本质作用类似COPY，将本地文件添加到容器中。

【格式】

```
ADD <src>… <dest>
```

（1）ADD过来的压缩包可以自动在目标路径下进行解压。

（2）原始路径可以是一个链接，ADD过程会尝试从该链接下载所需的文件到目标路径。

（3）一般情况下，建议使用COPY，而不是ADD。因为COPY过来的文件可以配合使用RUN来进行解压或者其他操作，搭配使用更灵活，而且单条语句所负担的功能唯一。

● ENV设置环境变量：ENV指令用于定义镜像的环境变量。

【格式】

```
ENV <key> <value>
ENV <key1>=<value1> <key2>=<value2>...
```

● EXPOSE暴露端口：EXPOSE指令声明运行时容器提供的服务端口，在运行时并不会因为这个声明，应用就会开启这个端口的服务。

【格式】

```
EXPOSE <端口 1> [<端口 2>...]
```

在DockerFile中写入这样的声明有两个好处，一是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；二是在运行中使用随机端口映射时，也就是docker run-P时，会自动随机映射EXPOSE的端口。

● VOLUME挂载共享卷。

【格式】

```
VOLUME ["<路径 1>", "<路径 2>"...]
VOLUME <路径>
```

Docker的使用原则除了每个容器做尽量少的事情，还要求容器运行时应该尽量保持容器存储层不发生写操作。对于数据库类需要保存动态数据的应用，其数据库文件应该保存在卷（Volume）中，也就是将本地磁盘的某一个目录挂载至容器内。这样的共享目录可以同时被多个不同的容器所使用。

● CMD服务启动指令：Docker不是虚拟机，而是一个进程。作为进程，可以设置启动镜像时的具体参数，其实就是设置一些你想自动启动的服务。

【shell格式】

```
CMD <命令>
```

【exec格式】

```
CMD ["可执行文件", "参数 1", "参数 2"...]
```

【参数列表格式】

```
CMD ["参数 1", "参数 2"...]
```

在指定ENTRYPOINT指令后，用CMD指定具体的参数。

● RUN指令：是DockerFile中最常用的指令之一，用来执行命令行的命令。

【格式】

shell格式：RUN<命令>，类似直接在终端输入命令。例如：

```
RUN echo '<h1>Hello， Docker!</h1>' > /usr/share/tomcat/welcome.html
```

## 8.3.4 Docker Compose编排服务

Docker Compose是Docker官方推出的一个单机容器编排工具。Compose是由用户定义的适合运行的多个容器的应用程序，可以在YMAL配置文件（docker-compose.yml）定义和配置服务，以及服务所需的环境变量、镜像、网络、数据卷等依赖，然后使用Docker Compose命令启动服务。Docker Compose可以自动从配置文件读取程序需要配置启动的单个或多个容器。使用Docker Compose可以一键启动多个容器，这样就解决了多个服务打包在一个镜像或者启动多个镜像的问题，只需要编写一个docker-compose.yml，就可以同时构建并启动多个镜像。使用Compose有如下三个步骤。

（1）在DockerFile中定义你的应用环境，使其可以在任何地方被复制。

（2）在docker-compose.yml中定义组成应用程序的服务，以便它们可以在隔离的环境中一起运行。

（3）运行docker-compose up命令，Compose将启动并运行整个应用程序。

我们将SIA-Gateway中依赖的多个服务，分别存放在不同的镜像，由Compose统一构建并编排启动SIA-Gateway服务。我们将SIA-Gateway中的三个大模块分别拆分为三个镜像，并将镜像上传到私服。

● Nginx镜像。

● SIA-Gateway Admin镜像。

● SIA-Gateway Core镜像。

```
[root@node1~]docker images
REPOSITORY                    TAG          IMAGE ID        CREATED         SIZE
reg.caiwu/centos-nginx        latest       3ec72fd792c     1 week ago      405MB
reg.caiwu/sia-gateway-admin   latest       ff454ba0b45     1 week ago      463MB
reg.caiwu/sia-gateway-core    latest       gf283322ofd     1 week ago      455MB
```

下面使用docker-compose.yml编排定义SIA-Gateway服务集群。

```
version: '1'
services:
  #部署 SIA-Gateway 前端工程
  nginx:
    image: reg.caiwu/centos-nginx
    hostname: nginx
    #映射端口
    ports:
      - "18086:18086"
    #目录挂载 【容器目录:宿主机目录】
    volumes:
      - /var/log/nginx/logs:/var/siagateway/nginx/logs
    #容器名称
    container_name: nginx
    restart: always
    #部署 SIA-Gateway 后端管理端工程
    sia-gateway-admin:
    image: reg.caiwu/sia-gateway-admin
    #映射端口
    ports:
      - "8040:8040"
    #配置环境变量
    environment:
    depends_on:
      - nginx
    #容器名称
    container_name: sia-gateway-admin
    restart: always
    #部署 SIA-Gateway CORE 节点工程
    sia-gateway-core:
    image: reg.caiwu/sia-gateway-core
    #映射端口
    ports:
      - "8080:8080"
    #配置环境变量
    environment:
    depends_on:
      - nginx
      -sia-gateway-admin
    #容器名称
    container_name: sia-gateway-core
    restart: always
```

将docker–compose.yml上传至服务器，然后进入目录，执行docker命令，启动整个Gateway服务集群。下面的命令将根据YML文件按照顺序启动这个网关服务集群，执行docker–compose命令。

```
docker-compose up –d
```

## 8.3.5 Maven插件构建Docker镜像

Maven是一个强大的项目管理与构建工具，可以用来构建Docker镜像。以下几款Maven的Docker插件比较常用，插件名称可以从GitHub开源项目中搜索到。

- spotify
- fabric8io
- bibryam

从各项目的功能性、文档易用性、更新频率、社区活跃度、Stars等几个纬度考虑，我们选用了第一款由Spotify公司开发的Maven插件，作为构建Docker镜像的工具。下面我们来详细探讨如

何使用Maven插件构建Docker镜像。

**准备工作**

使用该插件构建Docker镜像，需要有一个安装好的Docker运行环境，并且需要在运行该插件的机器上定义DOCKER_HOST环境变量，配置访问Docker的URL，如下：

```
export DOCKER_HOST=tcp://localhost:2375
```

上面的例子是在Linux环境下定义的DOCKER_HOST环境变量，因为Docker安装在本机上，所以使用localhost地址。如果你的Docker运行环境不在本机，则使用Docker所在机器的IP地址。

Docker默认开启远程访问API的端口（2375），如果你开启的是其他端口，则使用具体的端口配置。如果你的Docker没有开启远程访问API，则请自行开启。如果你要在Windows上运行该Maven插件，则同样需要在Windows上配置DOCKER_HOST环境变量，具体如下。

变量名：DOCKER_HOST

变量值：tcp：//127.0.0.1：2375

**配置pom.xml**

在pom.xml中引入dockerfile-maven-plugin插件，并配置该插件，示例如下。

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>1.4.13</version>
 <executions>
  <execution>
  <id>default</id>
  <goals>
    <goal>build</goal>
    <goal>Push</goal>
  </goals>
  </execution>
 </executions>
 <configuration>
  <repository>config-server</repository>
  <tag>${project.version}</tag>
  <buildArgs>
    <JAR_FILE>${project.build.finalName}.jar</JAR_FILE>
  </buildArgs>
 </configuration>
</plugin>
```

下面是插件相关指令参数的说明。

- Repository：指定Docker镜像的仓库名字。
- Tag：指定Docker镜像的Tag。
- buildArgs：指定一个或多个变量，传递给DockerFile，并通过ARG指令进行引用。

- execution：指定build和push目标，当运行mvn package时，会自动执行build目标，构建Docker镜像。当运行mvn deploy命令时，会自动执行push目标，将Docker镜像推送到Docker仓库。

需要说明的是，该插件要求必须提供DockerFile，而且要求放在项目根目录下，即与pom.xml同级目录。然后不需要像docker-maven-plugin插件那样指定dockerDirectory（DockerFile存放路径的参数）。代码如下所示。

```
FROM java:8
ARG JAR_FILE
ADD target/${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

**构建Docker镜像**

接下来，就可以运行Maven命令来构建Docker镜像了。

```
mvn package
mvn dockerfile:build
```

命令执行成功后，运行docker命令检查镜像是否存在。

```
docker images
```

**推送和拉取Docker镜像**

Docker镜像构建好后，我们还可以使用该插件将镜像推送到Docker仓库，在运行的mvn命令行上以参数的形式提供认证信息。

```
mvn dockerfile:Push -Ddockerfile.username=test-Ddockerfile.password=xxxx
```

拉取Docker镜像，运行命令如下。

```
docker pull registry 镜像名
```

运行Docker。

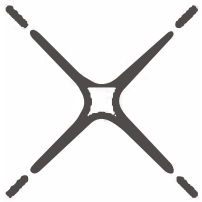
```
docker run -d -v /repositories:/var/lib/registry –e REGISTRY_STORAGE_DELETE_ENABLED=true
-p 5000:5000 --restart=always --privileged=true --name registry registry
```

## 8.4 小结

相比单体架构，微服务架构在部署灵活性上有了很大改善，然而频繁的服务发布也给微服务架构下的软件质量带来了稳定性的挑战。传统的手动部署方式已经不适用于微服务架构，目前基于容器的交付方式已经成为微服务的标准交付方式，通过持续集成交付工具，配合自动化、可持续交付部署的基础设施，来支撑微服务应用快速地迭代交付，成为软件交付的最佳实践。

第9章

# 服务监控治理



俗话说，流水的架构，铁打的监控，任何软件系统能够稳定运行都离不开监控。在微服务架构下，大型的单体架构拆分为众多微服务后，一个请求往往需要经过更多的服务节点。在这种情况下，我们必须知道是在哪个服务环节出现了故障，这就需要针对每一个服务，以及每一个指标都进行全面的监控。

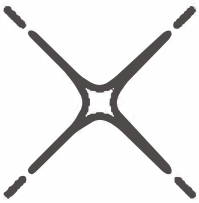
微服务的引入会带来分布式下服务监控和服务治理的技术挑战，之前系统内部的方法调用转变成分布式网络下的RPC，对于服务之间的交互集成和架构设计有更高的约束和要求。面对规模化的容器集群部署、不同种类的监控数据类型、海量的微服务，服务监控和服务治理成为微服务控制系统的关键组成部分。

服务系统监控、错误排查定位、系统运行数据的可视化展示和追踪一直是分布式架构系统的难点。而对于微服务系统的监控，必须具备处理和展示这些数据的能力，并能够通过这些数据快速地定位问题，这样才能帮助业务及时止损。

在本章中，我们将按照服务的指标监控、日志监控、全链路调用追踪三个层次，讲解如何增强系统的可观测性和监控运维能力。Spring Cloud微服务体系在生态上对指标监控、日志及调用追踪等监控技术都有很好的支持，我们会在本章中对相关的Spring Cloud技术组件进行介绍。

第9章

# 服务监控治理



俗话说，流水的架构，铁打的监控，任何软件系统能够稳定运行都离不开监控。在微服务架构下，大型的单体架构拆分为众多微服务后，一个请求往往需要经过更多的服务节点。在这种情况下，我们必须知道是在哪个服务环节出现了故障，这就需要针对每一个服务，以及每一个指标都进行全面的监控。

微服务的引入会带来分布式下服务监控和服务治理的技术挑战，之前系统内部的方法调用转变成分布式网络下的RPC，对于服务之间的交互集成和架构设计有更高的约束和要求。面对规模化的容器集群部署、不同种类的监控数据类型、海量的微服务，服务监控和服务治理成为微服务控制系统的关键组成部分。

服务系统监控、错误排查定位、系统运行数据的可视化展示和追踪一直是分布式架构系统的难点。而对于微服务系统的监控，必须具备处理和展示这些数据的能力，并能够通过这些数据快速地定位问题，这样才能帮助业务及时止损。

在本章中，我们将按照服务的指标监控、日志监控、全链路调用追踪三个层次，讲解如何增强系统的可观测性和监控运维能力。Spring Cloud微服务体系在生态上对指标监控、日志及调用追踪等监控技术都有很好的支持，我们会在本章中对相关的Spring Cloud技术组件进行介绍。

## 9.2 指标型数据监控

### 9.2.1 指标采集概述

在9.1节的监控系统分类中，我们说指标数据是监控系统判断运行状态的一个重要数据来源，这里的指标是在时间维度上捕获的与系统相关的值。这个指标值按照不同的层次，可以进一步分类。

- 基础类型指标：包括CPU、内存、网络、I/O等，基于JVM系统的应用，也可以把JVM的内存回收状态、堆栈等资源占用状态的指标纳入这一类型指标。基础类型指标通常可以从宏观的视角描述当前应用所属容器或者运行环境的基本状态。
- 应用服务类型指标：指服务的运行状态指标。前文我们说的服务延迟、流量吞吐、错误和饱和度即“黄金四指标”，线程个数、队列积压情况等数据都属于应用服务类型指标。因为服务指标最贴近应用服务本身，所以应用服务类型指标可以直观地反映当前服务的运行状态，也是开发运维人员排查异常状态和定位应用错误时的主要判断依据。
- 业务定制化指标：上述两类指标是比较通用的指标类型，然而很多业务需要定制化的指标来衡量某一个业务特性。例如9.1节中提到的通过“存量指标”衡量不同阶段、不同门店的一个业务运营状态。而这个指标的采集需要我们手动在代码的指定位置埋点，在采集指标数据后上报到监控服务器中心。下面介绍在监控指标方面有哪些主要的采集方式。

**系统指标采集方式**

Linux系统自带的命令工具是采集基础类型指标的主要方式，通过Linux系统命令可以发现服务器资源的性能瓶颈和资源占用情况。

- iostat：监控磁盘I/O情况。
- meminfo：查看内核使用内存情况的各种信息。
- mpstat：实时系统监控工具，能查看所有CPU的平均状况信息。
- netstat：显示了大量与网络相关的信息。
- nmon：监控Linux系统的性能、下载及安装。
- pmap：报告每个进程占用内存的详细情况。

- ps pstree：ps告诉你每个进程占用的内存和CPU处理时间，而pstree以树形结构显示进程之间的依赖关系，包括子进程信息。
- sar：显示CPU使用率、内存页数据、网络I/O和传输统计、进程创建活动和磁盘设备的活动详情。
- strace：诊断进程工具。
- tcpdump：网络监控工具，看看哪些进程在使用网络。
- uptime：该命令告诉你这台服务器从开机启动到现在已经运行了多长时间。
- vmstat：监控虚拟内存。
- wireshark：是一个网络协议检测程序，让你可以获取网站的相关资讯。
- dstat：该命令整合了vmstat、iostat和ifstat三种命令作为多类型资源统计工具。
- top：经常用来监控Linux的系统状况，比如CPU、内存的使用。
- ss：用来记录套接字统计信息，可以显示类似netstat一样的信息。
- lsof：列表显示打开的文件。
- iftop：是另一个基于网络信息的类似top的程序，能够显示当前时刻按照带宽使用量或者上传或者下载量排序的网络连接状况。

**应用指标采集方式**

- 手动埋点：手动埋点是侵入式的监控数据采集方式，主要应用在业务定制化的监控场景下。手动埋点的优点是可以更灵活地为我们提供业务内部的监控指标，当然缺点也很明显，需要在代码层面修改代码，具有一定的侵入性。如果项目指标数量有限，并且被埋点代码所在位置集中在个别文件中，可以考虑使用手动埋点的方式。
- 自动埋点：使用手动埋点的方式需要侵入式地修改已有的业务代码，对于很多业务方来说，这样的做法是无法接受的。如果能在程序加载或者运行期间动态地加入监控代码，就可以做到在运行期间动态埋点，无侵入地监控应用系统。在Java技术中，我们可以利用JavaAgent和Javaassist动态字节码改写技术实现自动埋点，增加指标抓取逻辑，这项技术的另一个使用场景就是APM中的调用链技术。在后面两节中，我们会进一步介绍这两种自动埋点捕获数据的“黑科技”。
- 自带监控功能，有以下三种方式。
  - JMX方式：多数Java开发的服务均可由JMX接口输出监控指标。其中不少监控系统都集成了JMX采集插件，除此之外，我们也可通过jmxtrans、jmxcmd等命令工具采集指标信息。
  - HTTP REST方式：Spring Boot提供的Actuator技术可以采集监控信息，并以HTTP REST的方式暴露监控指标。
  - OpenMetrics方式：作为Prometheus的监控数据采集方案，OpenMetrics可能很快会成为未来监控的业界标准。

**指标监控数据存储**

基于时间序列数据库的监控系统是非常适合做监控告警使用的，如果我们要搭建一套新的指标监控系统，就需要使用时序监控作为数据存储引擎，下面我们介绍几款常用的以时间序列数据库为

主的监控数据库。

- Prometheus（普罗米修斯）：2012年开源的一款监控框架，其本质是时间序列数据库，由Google前员工开发。Prometheus采用拉的模式从应用中拉取数据，并支持Alert模块，可以实现监控预警。同时，Prometheus提供了一种推数据的方式，但并不是推送到Prometheus Server中，而是在中间搭建一个PushGateway组件，通过定时任务模块将Metrics信息推送到这个PushGateway中，然后Prometheus Server采用拉的方式从PushGateway中获取数据。其他Prometheus用到的监控组件功能如下。
  - Prometheus Server：需要拉取的数据既可以采用静态方式配置在Prometheus Server中，也可以采用服务发现的方式。
  - PromQL：Prometheus自带的查询语法，通过编写PromQL语句可以查询Prometheus里面的数据。
  - Alertmanager：数据的预警模块，支持通过多种方式发送预警。
  - WebUI：展示数据和图形，通常与Grafana结合，采用Grafana来展示。
- OpenTSDB：2010年开源的一款分布式时序数据库，在这里我们把它主要用在监控方案中。OpenTSDB采用的是HBase的分布式存储，它获取数据的模式与Prometheus不同，它采用的是推模式。在展示层，OpenTSDB自带WebUI视图，可以与Grafana很好地集成，提供丰富的展示界面。但OpenTSDB并没有自带预警模块，需要自己去开发或者与第三方组件结合使用。
- InfluxDB：2013年开源的一款时序数据库，在这里我们主要把它用在监控系统方案中。它也采用推模式收集数据。在展示层，InfluxDB也自带WebUI，可以与Grafana集成。

## 9.2.2 JavaAgent技术

JavaAgent是一种特殊的Java程序，是Instrumentation的客户端。它与普通Java程序通过main方法启动不同，JavaAgent并不是一个可以单独启动的程序，它必须依附在一个Java应用程序（JVM）上，与主程序运行在同一个进程中，通过Instrumentation API与虚拟机交互。

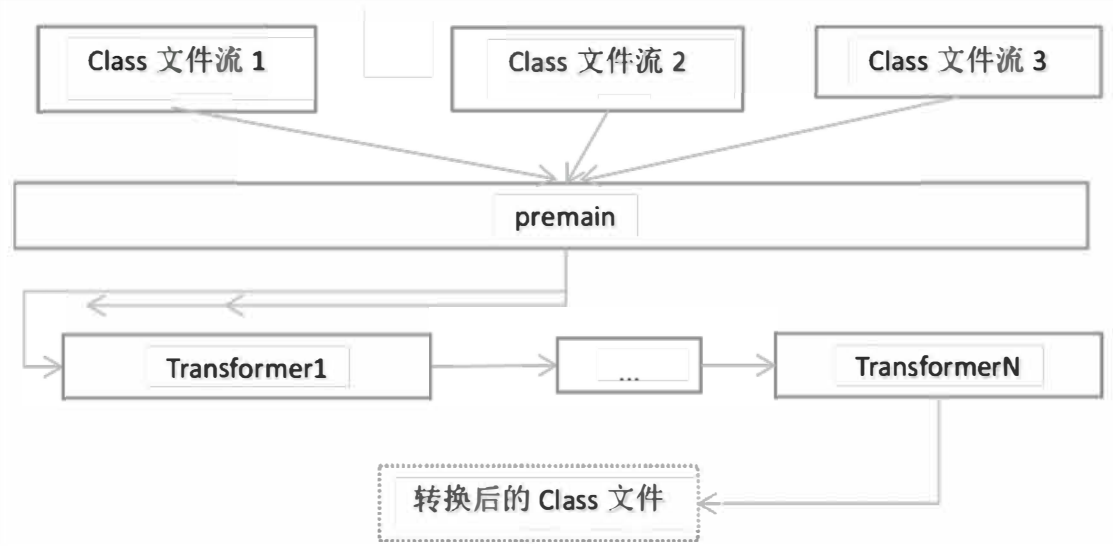
**JVM启动时静态加载**

对于JVM启动时加载的Agent模块代码，Instrumentation会通过premain方法传入代理程序，premain方法会在调用程序main方法之前被调用，同时Instrumentation包含agentmain方法实现字节码改写，二者的区别如下：

- premain方法用于在启动时，在类加载前定义类的TransFormer（转化器），在类加载的时候更新对应的类的字节码。
- agentmain方法用于在运行时进行类的字节码的修改，步骤分为注册类的TransFormer调用和retransformClasses函数进行类的重加载。

premain方法与agentmain方法相比有很大的局限性。premain方法仅限于应用程序的启动时，即main函数执行前。此时还有很多类没有被加载，而这些类使用premain方法是无法实现字节码改写的。

目前，主流的基于探针的监控系统都是基于这种方式实现的对应用的无侵入监控。我们知道程序的入口是main方法，而premain方法代表了在程序正式启动之前执行的动作，它同时具备类似AOP的能力。Transformer提供字节码文件流转化的能力，如下图所示是Class文件转换图。




**字节码改写**

如上图所示，任何Class文件在加载时，都要经过premain这一代码转换环节。通过一系列的TransFormer转换，Class字节码文件流最终转变为我们期望的代码实现，然后被加载到JVM中。修改Class字节码文件流的动作是在Transformer中进行的。我们可以使用Javaassist技术修改字节码文件流（下一节介绍）。下面就是我们实现的一个类，实现了带Instrumentation参数的premain方法。调用addTransformer方法对启动时所有的类进行拦截，示例代码如下：

```
public class PreMainTraceAgent {
    public static void premain(String AgentArgs， Instrumentation inst)
    {
        System.out.println("AgentArgs : " + AgentArgs);
        inst.addTransformer(new DefineTransformer()， true);
    }
    static class DefineTransformer implements ClassFileTransformer
    {
    @Override
    public byte[] transform(ClassLoader loader， String className， Class<?> classBeingRedefined，
        ProtectionDomain protectionDomain， byte[] classfileBuffer)
            throws IllegalClassFormatException {
        if ("java/util/Date".equals(className)){
            try
            {
              //从 ClassPool 获得 CtClass 对象
              final ClassPool classPool = ClassPool.getDefault();
              final CtClass clazz = classPool.get("java.util.Date");
              CtMethod convertToAbbr = clazz.getDeclaredMethod("convertToAbbr");
              String methodBody = "{sb.append(Character.toUpperCase(name.charAt(0)));"
                +"sb.append(name.charAt(1)).append(name.charAt(2));" +
```

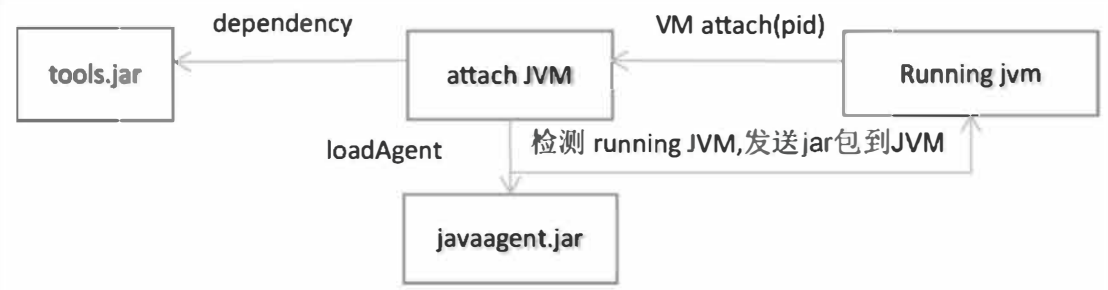
```
                "System.out.println(\"sb.toString()\");" + "return sb;
                 }";
                convertToAbbr.setBody(methodBody);
                byte[] byteCode = clazz.toBytecode();
                //detach 的意思是移除内存中曾经被 Javassist 加载过的数据对象
                clazz.detach();
                return byteCode;
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }
}
```

### JVM启动后动态Instrument机制

关于JVM启动后动态加载Agent的方法，Instrumentation会通过agentmain方法传入程序。agentmain方法在main函数开始运行后才被调用，其最大优势是可以在程序运行期间进行字节码的替换。

Attach API 注 实现动态注入的原理如下。

你的应用程序通过虚拟机提供的attach（pid）方法，可以将代理程序连接（attach）到一个运行中的Java进程上，之后便可以通过loadAgent（AgentJarPath）将Agent的jar包注入对应的进程，然后对应的进程会调用agentmain方法，如下图所示。




工程结构和上面premain的一样，编写AgentMainTest代码示例如下：

```
public class AgentMainTest
{
    public static void Agentmain(String AgentArgs, Instrumentation instrumentation)
    {
        instrumentation.addTransformer(new DefineTransformer(), true);
    }
    static class DefineTransformer implements ClassFileTransformer
    {
        @Override
        public byte[] transform(ClassLoader loader, String className, Class<?>
           classBeingRedefined, ProtectionDomain protectionDomain, byte[] classfileBuffer)
                throws IllegalClassFormatException
        {
            System.out.println("premain load Class:" + className);
            return classfileBuffer;
        }
    }
}
```

JavaAgent运行前启动加载代理程序的方法如下。

JavaAgent有两个启动时机，一个是在程序启动时通过–javaAgent参数启动代理程序；另一个是在程序运行期间通过Java Tool API中的Attach API动态启动代理程序。我们通过–javaAgent来指定我们编写的Agent的jar路径（./{Location}/Agent.jar）。这样在启动时，Agent就可以做定制化的字节码改动了。对于Spring Boot类内置容器的服务，可以使用下面方式：

```
java -javaAgent:{Location}/Agent.jar -jar start.jar
```

在Tomcat启动时，它会读取CATALINA_OPTS环境变量，并将它加入启动命令中。在环境变量中添加如下信息：

```
export CATALINA_OPTS="$CATALINA_OPTS -javaAgent:{Location}/Agent.jar"
```

Java程序运行后加载代理的方法如下。

程序启动之后，我们通过某种特定的手段加载Java Agent。这个特定的手段就是虚拟机的Attach API。这个API其实是JVM进程之间的沟通桥梁，它的底层通过Socket进行通信。JVM A可以发送一些指令给JVM B，JVM B收到指令之后，可以执行对应的逻辑，比如在命令行中经常使用的jstack、jcmd、jps等命令。因为是进程间通信，所以使用Attach API的也是一个独立的Java进程。下面是一个简单的实现，代码示例如下：

```
//13568 表示目标进程的进程 PID
vm = VirtualMachine.attach("13568");
try {
    vm.loadAgent(".../Agent.jar");    } //指定 Java Agent 的 jar 包路径
finally {
    vm.detach();
}
```

## 9.2.3 Javaassist技术

Javaassist是一个开源的分析、编辑和创建Java字节码的类库，在运行时能动态生成类、修改类，并且能直接使用Java编码。

在前面的JavaAgent一节中，我们知道通过Transformer方法可以取得类的字节码文件流，利用Javaassist技术可以解析字节码文件流为类对象，并对其进行修改，非常快速便捷。通过结合Javaassist与JavaAgent技术可以使字节码修改工作事半功倍。

Java字节码以二进制的形式存储在.class文件中，每一个.class文件都包含一个Java类或接口。Javaassist就是一个用来处理Java字节码的类库。它可以在一个已经编译好的类中添加新的方法，或者修改已有的方法，并且不需要对字节码有深入的了解。下面是代码示例，首先需要引入jar包：

```
<dependency>
    <groupId>org.javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.25.0-GA</version>
</dependency>
```

**Javaassist核心模块**

Javaassist中的核心模块是ClassPool、CtClass、CtMethod及CtField这几个类，下面简单介绍这些类的主要作用。

- ClassPool：一个基于HashMap实现的CtClass对象容器，其中键是类名称，值是表示该类的CtClass对象。默认的ClassPool使用与底层JVM相同的类路径，因此在某些情况下，可能需要向ClassPool添加类路径或类字节。
- CtClass：表示一个类，这些CtClass对象可以从ClassPool中获得。
- CtMethods：表示类中的方法。
- CtFields：表示类中的字段。

**Javaassist修改EurekaClient**

下面代码的功能是改写Eureka的onCacheRefreshed（）方法，通过下面字节码的改写可以触发父类的缓存刷新事件（这个功能是原onCacheRefreshed方法中不具备的逻辑），对于所有注册了该监听事件的方法都会执行“父类的缓存刷新”回调事件。

```
public class EurekaClientJavassist {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(EurekaClientJavassist.class);
    private static final String CLASSNAME =
        "org.springframework.cloud.netflix.eureka.CloudEurekaClient";
    private static final String METHODNAME = "onCacheRefreshed";
    public void hookCacheRefresh() {
        try {
            StringBuffer sbf = new StringBuffer();
            sbf.append("super.onCacheRefreshed();");
            String codePre = sbf.toString();
            JavassistProcessor.instance().hookExecuteBefore(this.getClass(),
                CLASSNAME， METHODNAME， codePre);
        }
        catch (Exception ex) {
            LOGGER.error("EurekaClientJavassist EXECUTE FAIL...", ex);
        }
    }
}
```

**Javaassist改写Class示例**

下面的方法是我们对Javaassist API的进一步封装，它可以在任意方法前后对任意Class执行我们添加的自定义代码逻辑，代码如下：

```
public void hookExecuteBefore(Class c， String className， String methodName， String preCode) {
    try {
        ClassPool pool = ClassPool.getDefault();
        ClassClassPath classPath = new ClassClassPath(c);
        pool.insertClassPath(classPath);
        CtClass ctClass = pool.get(className);
        CtMethod cm = ctClass.getDeclaredMethod(methodName);
        cm.insertBefore(preCode);
        ctClass.writeFile();
        ctClass.toClass();
    }
    catch (Exception ex) {
        LOGGER.error("Javassist EXECUTE onCacheRefreshed  FAIL...", ex);
    }
}
@SuppressWarnings("rawtypes")
public void hookExecuteAfter(Class c，String className，String methodName，String proCode)
{
    try {
        ClassPool pool = ClassPool.getDefault();
        ClassClassPath classPath = new ClassClassPath(c);
        pool.insertClassPath(classPath);
        CtClass ctClass = pool.get(className);
        CtMethod cm = ctClass.getDeclaredMethod(methodName);
        cm.insertAfter(proCode);
        ctClass.writeFile();
        ctClass.toClass();
    }
    catch (Exception ex) {
        LOGGER.error("Javassist EXECUTE onCacheRefreshed  FAIL...", ex);
    }
}
```

## 9.2.4 Spring Boot Admin监控详解

Spring Boot Admin是一个开源社区项目，用于管理和监控Spring Boot应用程序。应用程序作为客户端向Spring Boot Admin Server注册（通过HTTP）服务。同时，客户端可以被Spring Cloud的注册中心（例如Eureka、Consul）发现。UI使用AngularJs开发，可以展示Spring Boot Admin Client的Actuator端点上的一些监控数据。常见的监控功能可以分为下面三大类。

**指标类**

- 显示健康状况。
- JVM和内存指标。
- micrometer.io指标。
- 数据源指标。
- 缓存指标。
- 查看JVM系统和环境属性。
- JMX–beans交互。

**配置类**

- 查看Spring Boot的配置属性。

● 支持Spring Cloud的postable/env-和/refresh-endpoint。

**APM类**

● 实时日志级管理。

● 查看线程转储。

● 查看HTTP跟踪。

● 查看auditevents。

● 查看http-endpoints。

● 查看Flyway/Liquibase数据库迁移。

● 下载heapdump。

**接入spring-boot-admin-starter-server**

首先，在工程admin-server中引入Maven依赖。

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.1.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

然后，给工程的启动类AdminServerApplication加上@EnableAdminServer注解，开启AdminServer的功能。

```
@SpringBootApplication
@EnableAdminServer
public class AdminServerApplication {
    public static void main(String[] args) {
        SpringApplication.run( AdminServerApplication.class, args );
    }
}
```

**接入spring-boot-admin-starter-client**

首先，在admin-client工程的pom文件中引入admin-client依赖。

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.1.0</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

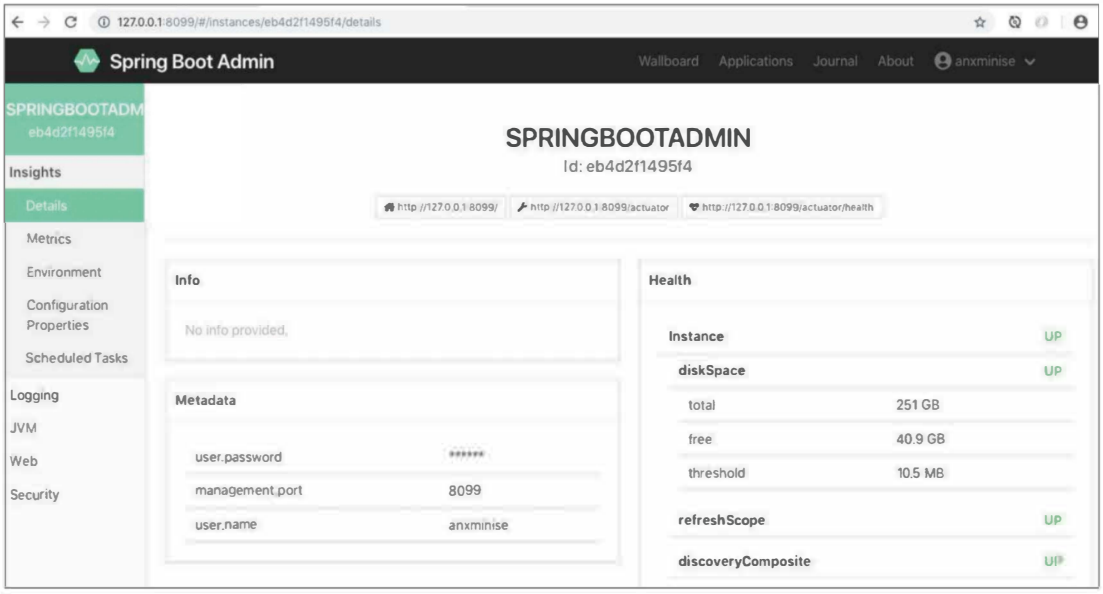
然后，在工程的配置文件application.yml中配置应用名和端口信息，设置admin-server的注册地址为http://localhost：8769，暴露Actuator的所有端口信息，具体配置如下。

```
spring:
  application:
    name: admin-client
  boot:
    admin:
      client:
        url: http://localhost:8099
server:
  port: 8099
management:
  endpoints:
    Web:
      exposure:
        include: '*'
  endpoint:
    health:
      show-details: ALWAYS
```

最后，工程的启动文件如下。

```
@SpringBootApplication
public class AdminClientApplication {
    public static void main(String[] args) {
        SpringApplication.run( AdminClientApplication.class,  args );
}
```

在浏览器上输入http://127.0.0.1：8099，进入Spring Boot Admin界面，下图是访问Spring Boot应用实例的详情界面。




## 9.2.5 Spring Boot集成Prometheus

Spring Boot中的spring-boot-starter-actuator组件已经集成了对Micrometer 注 的支持，其中Metrics端点的很多功能就是通过Micrometer实现的。Prometheus端点默认也是开启支持的，实际上Actuator依赖的spring-boot-actuator-autoconfigure中集成了对很多框架“开箱即用”的接口，其中Prometheus包中集成了对Prometheus的支持，使用Actuator可以轻易地让项目暴露出Prometheus的端点接口。作为Prometheus收集数据的客户端，Prometheus可以通过此端点收集应用中的Micrometer度量数据。

下面以Counter和Timer为例，讲解Spring Boot与Prometheus集成的主要步骤。

首先，引入spring-boot-starter-actuator和spring-boot-starter-web依赖。

```
<dependencies>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-aop</artifactId>
  </dependency>
  <dependency>
      <groupId>io.micrometer</groupId>
      <artifactId>micrometer-registry-Prometheus</artifactId>
      <version>1.1.0</version>
  </dependency>
</dependencies>
```

然后，在配置文件中添加如下配置。

```
spring.application.name：test
server.port :8081
#监控端点配置
#自定义端点路径，将/actuator/{id}设为/manage/{id}
#management.endpoints.Web.base-path: /manage
management.endpoints.Web.exposure.include: *
management.metrics.tags.application: ${spring.application.name}
```

添加启动类。

```
@SpringBootApplication
public class FreemarkerApplication {
    @Value("${spring.application.name}")
    private  String application;
    public static void main(String[] args) {
        SpringApplication.run(FreemarkerApplication.class， args);
    }
    @Bean
    MeterRegistryCustomizer<MeterRegistry> configurer() {
        return (registry) -> registry.config().commonTags("application"， application);
    }
}
```

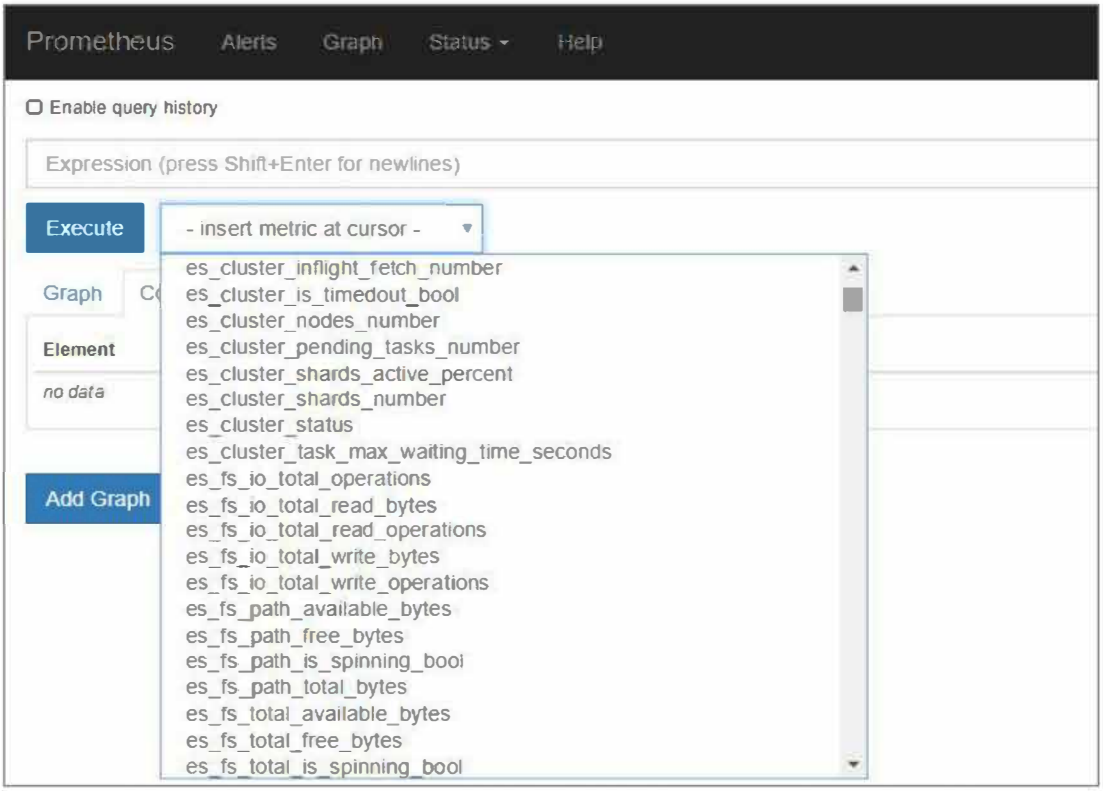
查看度量指标是否集成成功，访问地址为http://localhost：8081/actuator/Prometheus，返回结果如下。

```
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{application="test", area="heap", id="PS Old Gen", } 2.1193976E7
jvm_memory_used_bytes{application="test", area="nonheap", id="Metaspace", } 3.8791688E7
jvm_memory_used_bytes{application="test", area="heap", id="PS Survivor Space", } 0.0
jvm_memory_used_bytes{application="test", area="nonheap", id="Compressed Class Space", }
5303976.0
jvm_memory_used_bytes{application="test", area="heap", id="PS Eden Space", } 8.2574816E7
jvm_memory_used_bytes{application="test", area="nonheap", id="Code Cache", } 8693824.0
# HELP tomcat_global_received_bytes_total
# TYPE tomcat_global_received_bytes_total counter
tomcat_global_received_bytes_total{application="test", name="http-nio-8080", } 0.0
# HELP jvm_threads_daemon_threads The current number of live daemon threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads{application="test", } 20.0
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
tomcat_sessions_alive_max_seconds{application="Prometheus-test", } 0.0
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that the Java virtual
machine is using for this buffer pool
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{application="test", id="mapped", } 0.0
jvm_buffer_memory_used_bytes{application="test", id="direct", } 90112.0
//省略
```

prometheus.yml配置文件包含全局配置、告警插件配置、告警规则、Spring Boot应用配置等。Prometheus的常用配置参数说明如下。

```
#全局配置
global:
  scrape_interval:     15s   #收集一次数据的时长
  evaluation_interval: 15s   #评估一次规则的时长
  scrape_timeout:      10s   #每次收集数据的超时时间
#告警插件配置
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093
#告警规则，可以使用通配符
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
scrape_configs:
  - job_name: 'Prometheus'
    static_configs:
    - targets: ['localhost:9090']
#Spring Boot 应用配置
  - job_name: 'SpringBootPrometheus'
    scrape_interval: 5s
    metrics_path: '/actuator/Prometheus'
    static_configs:
      - targets: ['127.0.0.1:8081']
```

启动Prometheus，使用浏览器访问http://localhost：9090，查看具体的监控指标，如下图所示。




## 9.3 日志监控方案

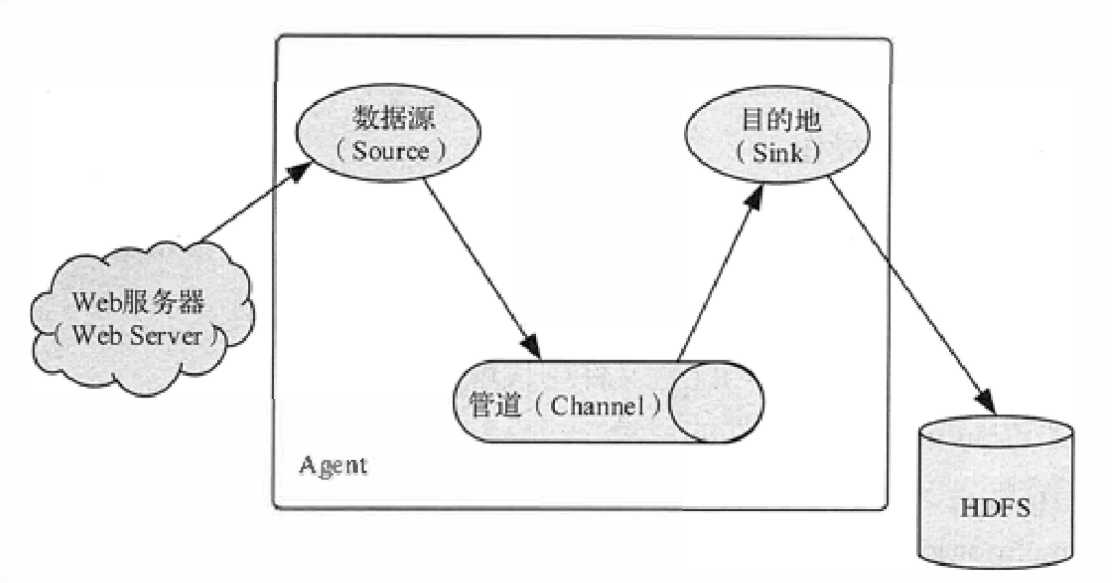
日志对我们进行系统故障定位具有关键的作用。我们的框架代码、系统环境及业务逻辑中一般都会产出一些日志，我们通常会把这些日志记录后统一收集起来，方便在需要的时候进行查询检索。ELK是目前开源领域比较流行且成熟的一站式日志解决方案。

### 9.3.1 日志采集方案

日志采集的代理端（Agent）其实就是一个将数据从源端投递到目的端的程序。我们会使用一个具备数据订阅功能的中间件作为日志采集、分析、存储的中间管道，来实现解耦的功能。目前业界比较流行的日志采集解决方案主要有Flume、Logstash、FileBeat和Fluentd等。

**Flume**

Flume是一个高可用的、高可靠的、分布式的海量日志采集、聚合和传输系统。Flume支持在日志系统中定制各类数据发送方，它可以收集数据。Flume提供对数据进行简单处理并写入各种数据接收方（如文本、HDFS、HBase等）的能力。Flume的核心是把数据从数据源（Source）收集过来，再将收集的数据送到指定的目的地（Sink）。为了保证输送过程一定成功，在送到目的地之前，会先缓存数据到管道（Channel），待数据真正到达目的地后，Flume再删除缓存的数据，整个流程如下图所示。



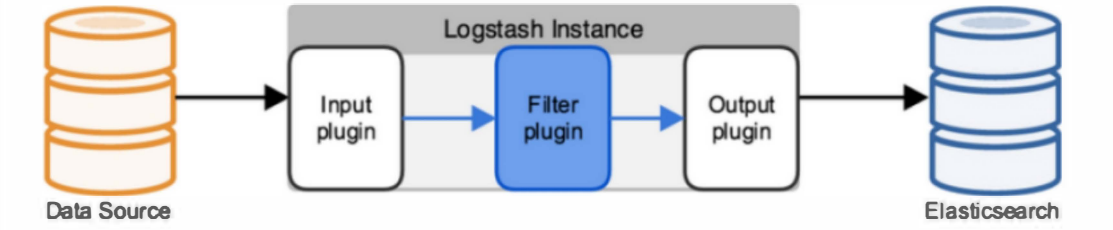

Flume的数据流由事件（Event）贯穿始终，事件是将传输的数据进行封装而得到的，是Flume传输数据的基本单位。如果是文本文件，则事件通常是一行记录。事件携带日志数据，并且携带头信息，这些事件由Agent外部的数据源生成，当Source捕获事件后会进行特定的格式化，然后Source会把事件推入（单个或多个）Channel中。Channel可以看作一个缓冲区，它将保存事件直到Sink处理完该事件。Sink负责持久化日志或者把事件推向另一个Source。

**Logstash**

Logstash是一个分布式日志收集框架，开发语言是JRuby，经常与Elasticsearch、Kibana配合使用，组成著名的ELK技术栈。

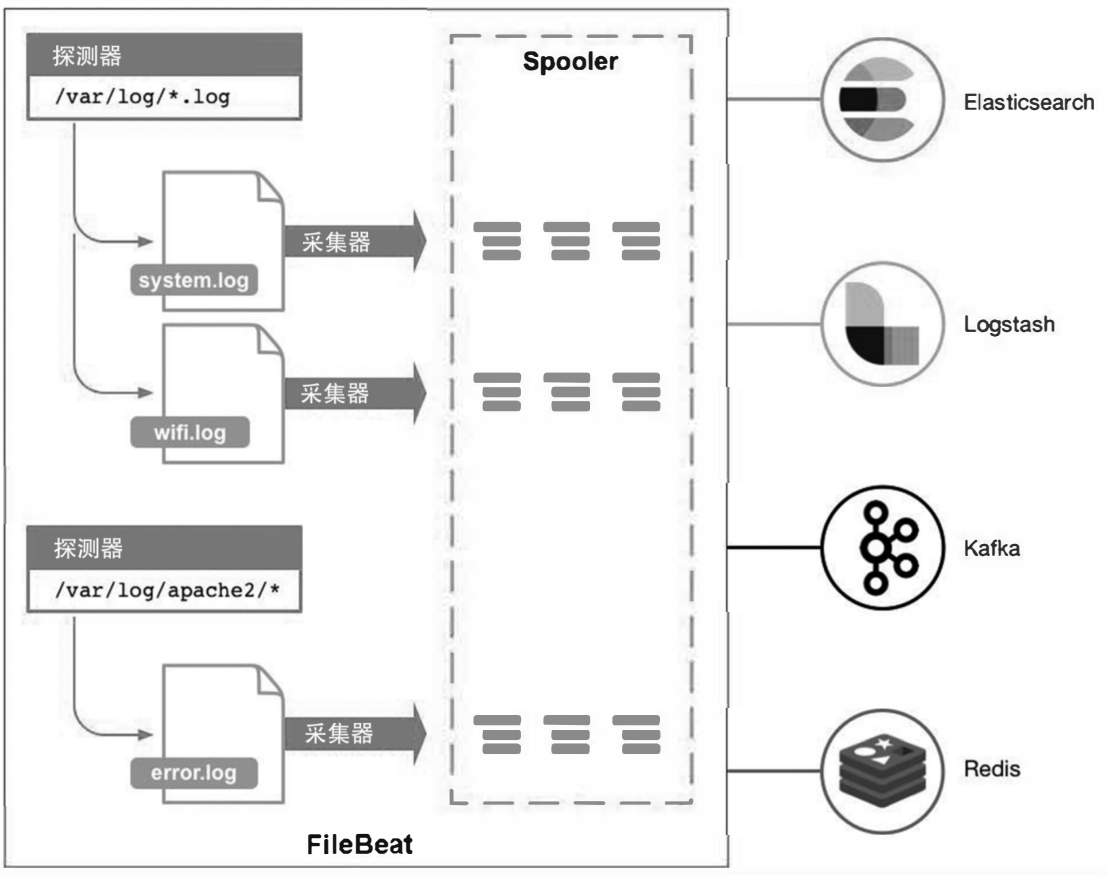
Logstash非常适合做日志数据的采集，既可以采用ELK组合使用，也可以作为日志收集软件单独出现。Logstash单独出现时可将日志存储到多种存储系统或临时中转系统，如MySQL、Redis、Kafka、HDFS、Lucene、Solr等，并不一定是Elasticsearch。

Logstash的设计非常规范，它有三个组件。因为架构比较灵活，所以如果不想用Logstash存储，也可以对接到Elasticsearch，这就是前面所说的ELK了。Logstash的采集过程如下图所示。




**FileBeat**

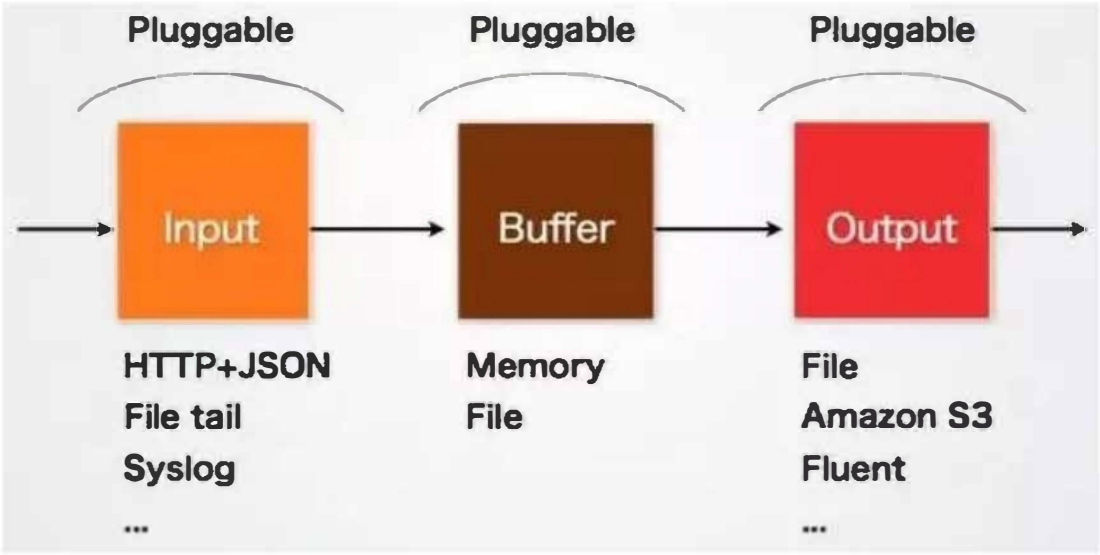
FileBeat和Logstash一样，都属于日志收集处理工具，它是基于原先Logstash的源码改造出来的。与Logstash相比，FileBeat更加轻量，占用资源更少。FileBeat涉及两个组件：探测器（Prospector）和采集器（Harvester）。FileBeat用来读取文件并将事件数据发送到指定的输出。FileBeat的工作流程是这样的：开启FileBeat时，它会启动一个或多个探测器去检测你设置的日志路径或日志文件，在定位到每一个日志文件后，FileBeat启动一个采集器。每个采集器读取一个日志文件的新内容并把数据发送到libbeat，libbeat会集合这些事件并将汇总的数据发送到你设置的外部接收程序中。下面是FileBeat的官方示意图。




**Fluentd**

业界一直采用ELK来管理日志，众所周知，Logstash是一个具有实时渠道能力的数据收集引擎，但和Fluentd相比，它在效能上的表现略逊一筹，故而逐渐被Fluentd取代，ELK也随之变成EFK，同时，Fluentd已经加入CNCF云原生成员。

Fluentd是一个开源的数据收集器，专为处理数据流设计，使用JSON数据格式。它采用插件式的架构（几乎所有源和目标存储都有插件），具有高可扩展性、高可用性，同时实现了高可靠的信息转发。Flueted由三部分组成，如下图所示。

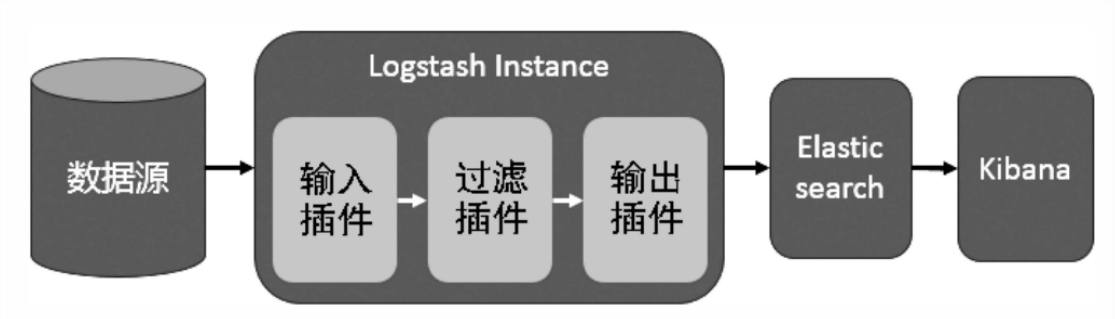



- Input：负责收集数据或者主动抓取数据，支持Syslog、HTTP、File tail等。
- Buffer：负责数据获取的性能和可靠性，也有文件或内存等不同类型的Buffer可以配置。
- Output：负责输出数据到目的地，例如文件。

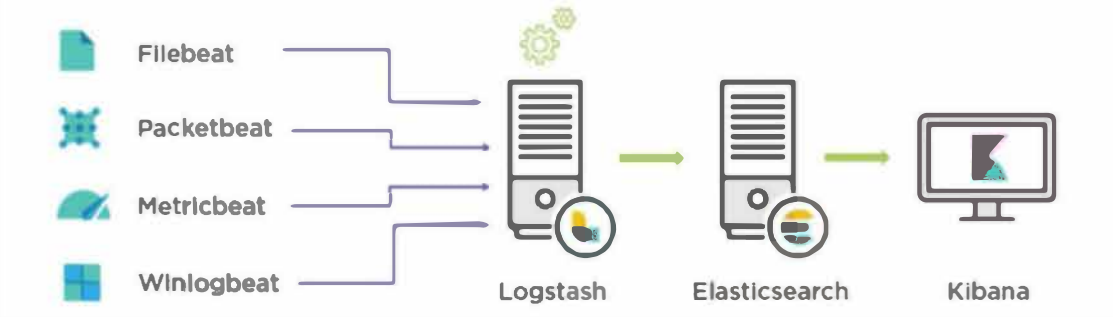
## 9.3.2 ELK日志的解决方案

**ELK简介**

ELK是软件集合Elasticsearch、Logstash、Kibana的简称，由这三个软件及其相关的组件可以打造大规模日志实时处理系统。ELK已经成为目前最为流行的集中式日志解决方案，在最简单的ELK解决方案中只有Logstash通过输入插件从多个数据源获取日志，再经过过滤插件进行数据架构处理，然后数据输出存储到Elasticsearch，在通过Kibana展示，下面是ELK最典型的架构图。




这种架构适用于简单场景，适合初学者搭建使用。而在前文的日志采集方案中，我们知道Logstash的采集存在性能瓶颈，所以在日志采集端通常会使用FileBeat作为日志采集Agent。下面简单介绍另一种ELK的日志改进解决方案——FileBeat+ELK，流程如下图所示。




- FileBeat：获取服务器上指定路径的日志文件，并将这些日志转发到Logstash实例以进行处理。设计FileBeat是为了提高可靠性和降低延迟。在微服务所在服务器上部署FileBeat，主要用来对微服务日志文件进行采集，将采集到的数据输出到指定文件或者队列服务器。
- Logstash：可以作为服务端的数据处理管道，从多个源中提取数据，对其进行转换，然后将其存储到Elasticsearch。简单来说就是日志的收集、分析、过滤工具。从文件系统或者服务器队列进行聚合、过滤，并输出到Elasticsearch存储。
- Elasticsearch：它是一个开源分布式搜索引擎，通常作为日志的存储服务器，提供收集、分析、存储数据三大功能。

Kibana：它是一个基于Web的图形界面，读取Elasticsearch上的集群数据，展示Web查询页面，提供历史日志数据查询，用于搜索、分析和展示存储在Elasticsearch中的日志数据。

**ELK在微服务架构中的局限**

传统的ELK解决方案的最大优势就是对日志全流程支持，从日志采集、存储到展示的所有环节都是“开箱即用”的，但是在实际微服务项目使用过程中依然存在一些问题。

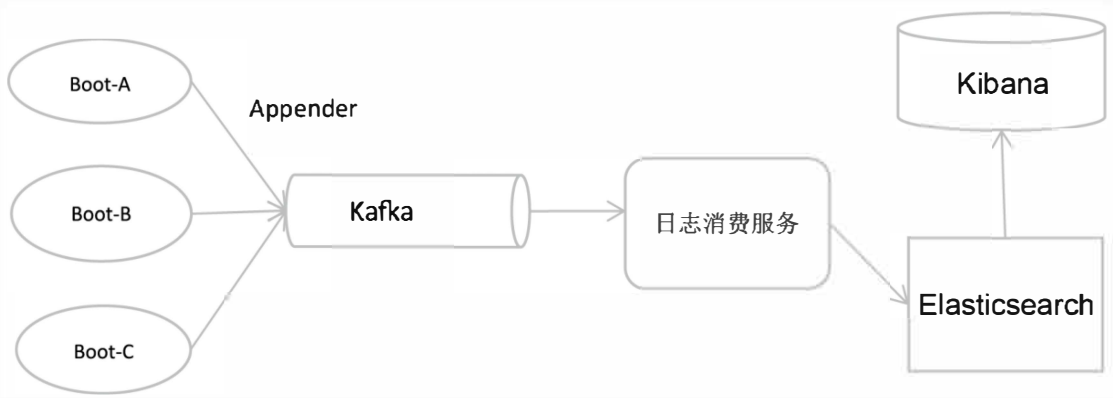
- 每个服务器如果想享受ELK日志解决方案，都需要在宿主机上安装一个代理客户端，而像Logstash这样的采集器，本身比较占用内存，会存在与宿主机应用抢占资源的问题。
- ELK的日志采集机制是使用Agent从磁盘读取增量的日志文件，但是磁盘读取会遇到资源消耗增大、日志读取缓慢等问题。
- 每部署一套新的服务，都需要部署相关的Agent，对Agent后期的升级及配置环境运维都将存在一系列问题。

## 9.3.3 Spring Boot的日志解决方案

针对上述ELK的这些问题，我们有必要在以Spring Boot框架为主的微服务应用系统中，采用更加高效、简易、轻量级的日志解决方案。

我们知道，Spring Boot采用了LogBack作为默认的文件系统，而LogBack有非常强大的Appender机制，可以将日志动态输出到指定的Appender上，这样在日志采集时就不需要为每一个微服务都安装一个Agent了，同时日志不需要再落盘就可以通过第三方消息中间件将日志异步转发出去，避免了为每一台宿主机都安装LogAgent采集进程；同时可以使用定制化的

LoggerAppender，通过Nexus私服发布更新，应用系统可以在编译期完成日志LogAppender的升级，避免了为每一套应用程序修改升级Agent代码。同时对于Logger append的Sink写入端可以选择Kafka或者Redis这样高性能的中间件，作为高并发日志系统的缓存，避免对ELK服务稳定性造成冲击。下面是基于Spring Boot采集日志、归集日志、存储日志的改良的解决方案架构图。




**自定义Appender的配置加载**

下面是在Logback.XML中自定义的Appender实现，使用了异步的Appender。这样，服务在调用LogBack打印日志时，不会阻塞当前应用代码继续执行正常逻辑。

```
<appender name="Kafka" class="com.xxx.mq.message.appender.MqAppender">
</appender>
<appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="Kafka"/>
</appender>
```

**自定义Appender的实现**

根据上面Logback.XML的配置，我们配置了一个Kafka类型的Sink输出Appender：MqAppender，具体的代码实现如下。

```
public class MqAppender extends UnsynchronizedAppenderBase<ILoggingEvent> {
    //省略
    private String Topic;
    private BaseMqProducer producer;
    private boolean enable = true;
    @Override
    protected void append(ILoggingEvent event) {
        if (producer == null || !enable) {
            return;
        }
        Map<String, Object> map = new LinkedHashMap<>(8);
        map.put("logtime", event.getTimeStamp());
        map.put("level", event.getLevel().toString());
        map.put("threadname", event.getThreadName());
        map.put("loggername", event.getLoggerName());
        map.put("msg", event.getFormattedMessage());
        map.putAll(args);
        MqMessage message = new MqMessage(Topic, JsonTransform.toString(map));
        producer.send(message);
    }
    public static void init(String Topic, BaseMqProducer producer, boolean enable,
    Map<String, Object> args) {
        LoggerContext loggerContext = (LoggerContext)
            LoggerFactory.getILoggerFactory();
        for (Logger logger : loggerContext.getLoggerList()) {
            Iterator<Appender<ILoggingEvent>> appenderIter =
            logger.iteratorForAppenders();
            while (appenderIter != null && appenderIter.hasNext()) {
                Appender<ILoggingEvent> appender = appenderIter.next();
                if (appender instanceof SagMqAppender) {
                    setParameter(logger, (SagMqAppender) appender, Topic,
                    producer, enable, args);
                }
                if (appender instanceof AsyncAppender) {
                    Iterator<Appender<ILoggingEvent>> subAppenderIter =
                     ((AsyncAppender) appender).iteratorForAppenders();
                    while (subAppenderIter != null && subAppenderIter.hasNext())
                    {
                        Appender<ILoggingEvent> subAppender =
                        subAppenderIter.next();
                         if (subAppender instanceof SagMqAppender) {
                              setParameter(logger, (SagMqAppender) subAppender,
                             Topic, producer, enable, args);
                        }
                    }
                }
            }
        }
    }
```

在这个自定义Appender类中，有两个核心方法：init方法和append方法。init方法的主要作用是完成当前应用需要的资源初始化工作；append方法则是日志拦截方法，BaseMqProducer类为初始化时构造的Kafka–Producer客户端对象，会调用send方法将构造好的日志消息发送到Kafka中间件。需要注意的是，在append方法中，需要过滤掉Kafka自身的日志输出，以免形成死循环。因为篇幅所限，我们仅就Append思路进行分享，具体Kafka的BaseMqProducer实现方法就不再赘述。

**日志消费服务**

日志消费服务的主要作用是将Kafka收集的日志根据Topic和日志消息负载信息（PayLoad）分发到不同的Elasticsearch的Index中。下面是日志消费服务对Kafka客户端的自动配置代码。

```
@EnableConfigurationProperties(SagMqProperties.class)
@ConditionalOnMissingBean(ConsumerAutoConfiguration.class)
public class ConsumerAutoConfiguration {
    @Autowired
    private SagMqProperties properties;
    @Bean
    @ConditionalOnMissingBean(BaseMqConsumer.class)
    public BaseMqConsumer getMqConsumer() {
        if (properties.getMqType() == SagMqProperties.MqType.Kafka) {
            KafkaConsumer consumer = new KafkaConsumer();
            consumer.setTopicPrefix(properties.getKafka().getTopicPrefix());
            consumer.start();
            return consumer;
        }
        return null;
    }
    @Bean
    @ConditionalOnMissingBean(name = "KafkaListenerContainerFactory")
        public ConcurrentKafkaListenerContainerFactory<?, ?> KafkaListenerContainerFactory(
          ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
           ConsumerFactory<Object, Object> KafkaConsumerFactory) {
           ConcurrentKafkaListenerContainerFactory<Object, Object> factory = new
            ConcurrentKafkaListenerContainerFactory<>();
             configurer.configure(factory, KafkaConsumerFactory);
          //设置批量消费
          factory.setBatchListener(true);
          return factory;
    }
}
```

下面是Kafka的日志消费代码，利用@KafkaListener注解对TopicPattern下的日志做数据消费，MqHandler可以做消息过滤、预警、聚合、数据加工等工作，最终将消息发送给Elasticsearch存储引擎。

```
public class KafkaConsumer extends BaseMqConsumer {
    private String topicPrefix;
    @Autowired
    private MqHandler mqHandler;
    @KafkaListener(topicPattern = "${spring.Kafka.topicPrefix}" + "*")
    public void listen(List<ConsumerRecord<String, String>> records) {
        List<MqMessage> list = new ArrayList<>(records.size());
        for (ConsumerRecord<String, String> record : records) {
            String Topic = record.Topic().substring(topicPrefix.length());
            MqMessage msg = new MqMessage(Topic, record.value());
            msg.addArg("partition", record.partition());
            msg.addArg("timestamp", record.timestamp());
            msg.addArg("key", record.key());
            list.add(msg);
        }
        mqHandler.handle(list);
    }
}
```

**日志存储**

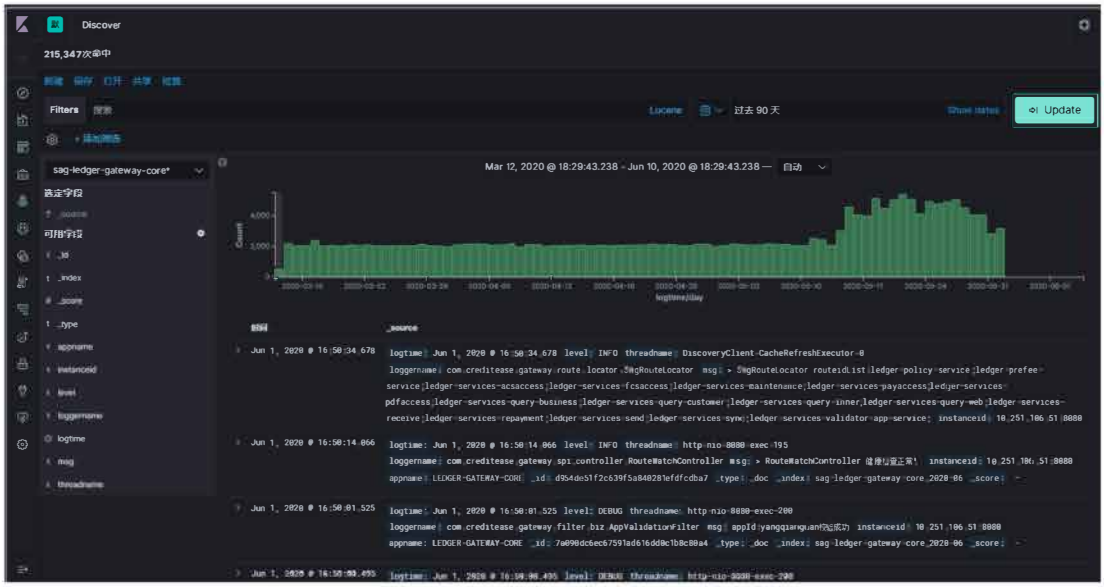
对于日志存储，Spring Boot 2.2.0已经兼容Elasticsearch 7.x，可以直接引入Elasticsearch的Maven依赖。日志的实体类定义代码如下所示。

```
@Data
@NoArgsConstructor
@Accessors(chain = true)
@Document(indexName = "ems", type = "_doc",  shards = 1,  replicas = 0)
public class DocBean {
    @Id
    private Long id;
    @Field(type = FieldType.Keyword)
    private String firstCode;
    @Field(type = FieldType.Keyword)
    private String secordCode;
    @Field(type = FieldType.Text,  analyzer = "ik_max_word")
    private String content;
    @Field(type = FieldType.Integer)
    private Integer type;
    public DocBean(Long id, String firstCode, String secordCode, String content, Integer
            type){
        this.id=id;
        this.firstCode=firstCode;
        this.secordCode=secordCode;
        this.content=content;
        this.type=type;
    }
}
```

下面是持久化实现逻辑。

```
public interface ElasticRepository extends ElasticSearchRepository<DocBean,  Long> {
    //默认注释
    //@Query("{\"bool\" : {\"must\" : {\"field\" : {\"content\" : \"?\"}}}}")
    Page<DocBean> findByContent(String content,  Pageable pageable);
    @Query("{\"bool\" : {\"must\" : {\"field\" : {\"firstCode.keyword\" : \"?\"}}}}")
    Page<DocBean> findByFirstCode(String firstCode,  Pageable pageable);
    @Query("{\"bool\" : {\"must\" : {\"field\" : {\"secordCode.keyword\" : \"?\"}}}}")
    Page<DocBean> findBySecordCode(String secordCode,  Pageable pageable);
}
```

日志展示需要启动Kibana，默认地址是http://127.0.0.1：5601。浏览Kibana界面，Kibana会自动检测Elasticsearch中是否存在该索引名称，通过Filter搜索框可以检索日志，如下图所示。



## 9.4 服务调用链技术

服务调用链技术是微服务架构中对服务进行监控的重要环节，它可以帮助我们清晰地了解当前系统的运行情况，同时帮助我们定位问题，解决分布式网络下服务交互追踪的问题。

### 9.4.1 APM 注 与调用链技术

在单体应用架构拆分为微服务架构后，一个用户请求会跨网络依次调用不同的服务节点进行分布式交互处理，最后将结果汇总处理，再将结果返回给用户。那么在整个处理的链条中，如果有任何一个节点出现了延迟或者超时等问题，都有可能导致最终结果出现异常。在很多场景下，一个功能可能需要多个技术团队、多种技术栈、多个跨地域网络中的不同服务协调完成。我们想要排查出链条中具体哪个服务节点出了问题，其实并不容易。所以微服务架构的一大挑战就是分布式网络交互带来的固有复杂性。

谈到分布式网络环境下的应用交互，就不得不提到分布式计算的八大误区，这些误区无疑都与我们对应用构架所在的基础设施——网络认识不足有关。下面我们将这些误区从应用使用网络的视角，按照正确的理解总结如下：

- 网络是不可靠的，我们的应用可以随时发现网络的异常，并适应网络的不可靠。
- 网络存在延迟，应用系统需要通过网络监控工具了解网络状况，避免系统的异常行为。
- 带宽是有限的，应用的传输数据在逐渐增大，尽管应用系统无法控制生产环境下的网络带宽，但是我们需要模拟网络，监控并控制应用数据的传输上限。
- 网络是不安全的，应用需要在早期就将安全因素考虑到设计方案中。
- 网络的拓扑是变化的，网络设备（例如DNS）、存储资源的迁移都需要在应用架构层面尽量做到对用户无感知。
- 系统存在多个管理员，应用需要向不同管理员提供可视化的应用管理界面，并帮助他们去管理应用。
- 数据传输是需要成本的，应用在网络上传播，无论如何都会耗费额外资源。数据序列化会耗费计算资源、网络传输上的延迟，这些都是数据传输的成本，都要求应用使用更加经济高效的解决方案。

- 网络是异构的，应用之间可能采用不同的网络协议，应用集成交互时最好采用通用协议。

总结来看，这些问题都是分布式网络下的陷阱。在微服务架构下，由于服务之间的交互对网络有如此强的依赖，我们需要时刻了解请求耗时、网络延迟、业务吞吐、系统运行等情况，所以，我们需要一个系统来分析当前系统的瓶颈，解决系统问题。而一套APM系统和调用链技术就可以帮助我们了解当前应用的状态。

**APM与调用链的概念**

APM是一种应用性能管理/监控技术架构，用以将应用运行过程中的函数调用、网络调用等时间和性能进行指标化及可视化展现。

而调用链技术可以说是APM应用性能管理的子集。通过跟踪一次业务调用请求，记录业务在进程内部及进程之间的调用关系（调用信息包括时间、接口、结果）到日志中，然后根据日志信息进行分析处理，掌握分布式网络环境下请求的全链路跟踪还原及展示的技术。调用链技术综合了数据埋点、采集、数据聚合、数据展示等多项技术，可以根据不同层次和维护的分析，标识出服务调用异常，快速定位问题，从微观层面监控系统的运行状态，是一种细粒度的服务监控模式。

**调用链技术的作用**

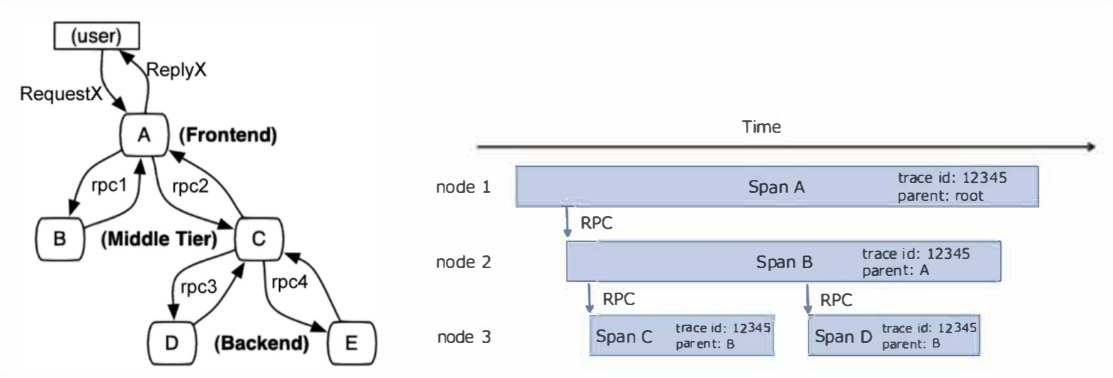
- 快速定位问题：业务全链路监控就是要从业务的视角出发，监控整个业务流程的健康状况，无须多个系统切换，直观看到全局和上下游，方便快速发现、定位问题。这个作用前面一直在讲，在微服务架构下，问题定位就变得非常复杂了，一个请求可能会经过多个服务节点，使用调用链技术能让开发人员快速地定位到问题和问题所在的相应模块。
- 拓扑关系：当微服务拆分后，服务之间的调用关系也随之变得复杂，而调用链技术可以帮助我们准确地掌握服务之间的调用关系，并清晰地表现为网络拓扑图。
- 优化系统：优化系统也是调用链技术很重要的一个功能。因为我们记录了请求在调用链上每一个环节的信息，就可以通过这个来找出系统的瓶颈，做出针对性的优化。还可以分析调用链是否合理，是否调用了不必要的服务节点，是否有更近、响应更快的服务节点。通过对调用链的分析，我们就可以找出最优的调用路径，从而提高系统的性能。

## 9.4.2 Dapper与分布式跟踪原理

追根溯源，最早的分布式调用链跟踪系统——Dapper是由Google设计的。目前业界对调用链技术的实现也都是参考Dapper实现的，可以说Dapper已成为分布式调用链技术的业界标准。Dapper的调用跟踪模型描述了分布式Tracing的基本原理和工作机制。我们来看下核心的概念和术语。

- Trace：一次完整的分布式调用跟踪链路。
- Span：跨服务的一次调用，多个Span组合成一次Trace记录。
- Annotation：用来记录请求特定事件的相关信息（例如时间）。

下图是Dapper在调用链监控系统中描述的从用户发出请求到后端服务返回响应的整个流程。




左图可以理解为调用链经常发生的一个场景，描述了一个完整的调用链发生过程，整个链路是与5台服务器相关的一个服务，包括前端（A）、两个中间层（B和C），以及两个后端（D和E）。当一个用户（这个用例的发起人）发起一个请求时，首先到达前端，然后发送两个RPC到B和C。B会马上做出反应，但是C需要与后端的D和E交互之后再返给A，由A来响应最初的请求。

右图描述的是如何通过TraceID、Span等元素说明一个完整的链路跟踪过程。在Dapper的跟踪树结构中，树节点是这个Trace的基本单元，每个节点都是对Span的引用，节点之间的连线描述的是父子Span之间的直连关系。Dapper记录了Span名称，以及每个Span的ID和父ID，以重建在一次追踪过程中不同Span之间的关系。如果一个Span没有父ID，则被称为Root Span。所有Span都挂在一个特定的跟踪上，也共用一个跟踪ID。所有这些ID用全局唯一的64位整数标识。在一个典型的Dapper跟踪中，我们希望每一个RPC对应一个单一的Span，而且每一个额外的组件层都对应一个跟踪树型结构的层级。

从右图可见，一次请求只有一个唯一的TraceID=12345。在一次请求中，会在网络的开始生成一个全局唯一的用于标识此次请求的TraceID，这个TraceID在这次请求调用过程中无论经过多少个节点都会保持不变，最终可以通过TraceID将这一次用户请求在系统中的路径全部串起来。在这个请求的调用链中，SpanA调用了SpanB，然后SpanB调用了SpanC和SpanD，每一次Span调用都会生成一个自己的Span ID，并且记录自己的上级Span ID是谁。可以通过Span ID来定位当前请求在整个系统调用链中的位置，以及它的上下游节点，通过这些Span ID，整个链路基本上就都能标识出来了。

Dapper允许应用在跟踪过程中添加额外信息，以表征这个Annotation是用来即时记录事件的发生信息的，以下是一系列预定义的用来记录一次请求开始和结束的核心Annotation。

- Client Start（CS）：客户端发起一次请求时的记录。
- Server Receive（SR）：服务器收到请求并开始处理，SR和CS的差值就是网络延时和时钟的误差。
- Server Send（SS）：服务器完成处理并返回客户端，SS和SR的差值就是实际处理时长。

- Client Receive（CR）：客户端收到回复时建立，标志着一个Span的结束。我们通常认为，一旦CR被记录了，一个RPC调用也就完成了。

**常用的分布式调用链技术**

目前市面上很多APM工具都基于Dapper论文，下面简单介绍在业界影响比较广泛的调用链跟踪技术。

- CAT：大众点评开源的基于手动代码埋点和配置的集调用链分析、应用监控、日志采集、监控报警于一体的监控平台工具。
- Zipkin：Twitter开源的调用链分析工具，可以与Spring Cloud Sleuth结合使用，部署简单，使用轻量。
- SkyWalking：国内开源的一款APM工具，基于字节码注入方式，接入端代码无侵入实现，支持多种插件，UI功能比较强大，目前已经进入Apache孵化器。

## 9.4.3 Sleuth与Zipkin技术

Spring Cloud Sleuth为Spring Cloud实现了分布式跟踪解决方案，Sleuth可以结合Zipkin做链路跟踪。Spring Cloud Sleuth的服务链路跟踪功能可以帮助我们快速发现错误根源，以及监控分析每条请求链路上的请求性能。Sleuth的主要工作原理是拦截请求，并在日志中加入额外的Span和Trace的相关信息。从Sleuth 2.0.0开始，Sleuth使用Brave作为调用链工具库。Brave是一个用于捕捉分布式系统之间调用信息的工具库，可以将这些信息以Span的形式发送给Zipkin。

Zipkin是一个分布式跟踪系统，用户可以将Sleuth发送的时序数据存储在Elasticsearch或者MySQL中，Zipkin UI根据数据存储显示服务之间的请求调用详情和依赖调用关系图。

**Spring Cloud Sleuth的主要特性**

- Sleuth的调用链消息日志采用[Appname，TraceID，SpanID，Exportable]条目添加到Slf4J MDCS，该日志格式兼容Zipkin，其条目含义如下。
  - SpanID：发生的特定操作的ID。
  - Appname：记录Span的应用程序的名称。
  - TraceID：包含Span的延迟图的ID。
  - Exportable：是否应将日志导出到Zipkin。
- Sleuth具有采样和延迟分析等特性，减少了Sleuth日志采集输出对应用性能的影响，带内传播调用图相关数据，其他数据基于带外传播，可以向Zipkin系统报告查询和可视化分析。
- Sleuth提供进程之间的上下文传播，可以在Span上设置添加额外的信息，并通过HTTP给其他进程传递消息。
- Sleuth实现了对Spring生态下不同组件（Feign、RestTemplate、Zuul过滤器、Filter等）的拦截策略，并将Span信息植入HTTP Head等载体。

**Spring Cloud Sleuth与Zipkin接入**

创建两个工程：SampleSleuthZipkinApplication和ZipkinServerApplication。

- SampleSleuthZipkinApplication工程：负责模拟应用生产者和应用消费者，在3379端口监听，并生成Sleuth的调用链日志供Zipkin采集分析。
- ZipkinServerApplication工程：负责启动ZipkinServer，默认在9411端口监听，使用HTTP的方式收集Sleuth日志，展现调用链的调用关系。

**SleuthZipkinApplication的代码实例**

1.添加Maven依赖

情况一：如果你只使用Sleuth，在不需要集成Zipkin的情况下，请将如下Maven依赖添加到工程中。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

情况二：如果你想要Sleuth和Zipkin结合使用，请添加Zipkin依赖项。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

2.修改配置文件

```
server:
   port: 3379
spring:
  application:
    name: testsleuthzipkin  //Zipkin 中将展示应用名
    zipkin:
    baseUrl: http://127.0.0.1:9411/  //这是 Zipkin 接受 Sleuth 日志消息的 URL 地址
  sleuth:
    sampler:
      probability: 1.0
        #sample:
    #zipkin:
      #enabled: false  //enabled=false 日志将打印在 Consol，注释此日志将发送到 Zipkin
```

3.创建SampleZipkinApplication工程

```
@SpringBootApplication
@EnableAsync
public class SampleSluethZipkinApplication{
    public static void main(String[] args) {
      SpringApplication.run(SampleZipkinApplication.class,  args);
   }
```

```
    @Bean
    public RestTemplate restTemplate() {
       return new RestTemplate();
    }
    @Bean
    @ConditionalOnProperty(value = "sample.zipkin.enabled",  havingValue = "false")
    public SpanHandler SpanHandler() {
       return new SpanHandler() {
          @Override
          public boolean end(TraceContext context,  MutableSpan Span,  Cause cause) {
             System.out.println(Span.toString());
             return true;
          }
       };
    }
}
```

## 4.创建一个异步的Service测试验证服务

```
@Service
public class SampleBackground {
    @Autowired
    private Tracer tracer;
    private Random random = new Random();
    @Async
    public void background() throws InterruptedException {
      int millis = this.random.nextInt(1000);
      Thread.sleep(millis);
      this.tracer.currentSpan().tag("background-sleep-millis",
      String.valueOf(millis));
    }
}
```

## 5.创建Sleuth的接受用户请求的Controller实现类

```
@RestController
public class SampleController
    implements ApplicationListener<ServletWebServerInitializedEvent> {
    private static final Log log = LogFactory.getLog(SampleController.class);
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private Tracer tracer;
    @Autowired
    private SampleBackground controller;
    private Random random = new Random();
    private int port;
```

```
    @RequestMapping("/hi2")
    public String hi2() throws InterruptedException {
        log.info("hi2");
        int millis = this.random.nextInt(1000);
        Thread.sleep(millis);
        this.tracer.currentSpan().tag("random-sleep-millis", String.valueOf(millis));
        return "hi2";
    }
    @RequestMapping("/call")
    public Callable<String> call() {
        return new Callable<String>() {
        @Override
        public String call() throws Exception {
            int millis = SampleController.this.random.nextInt(1000);
            Thread.sleep(millis);
            Span currentSpan = SampleController.this.tracer.currentSpan();
            currentSpan.tag("callable-sleep-millis", String.valueOf(millis));
            return "async hi: " + currentSpan;
            }
        };
    }
    @RequestMapping("/hi")
    public String hi() throws InterruptedException {
        Thread.sleep(this.random.nextInt(1000));
        log.info("Home page");
        String s = this.restTemplate.getForObject("http://localhost:" + this.port + "/hi2",
                String.class);
        return "hi/" + s;
    }
    @RequestMapping("/async")
    public String async() throws InterruptedException {
        log.info("async");
        this.controller.background();
        return "ho";
    }
    @RequestMapping("/start")
    public String start() throws InterruptedException {
        int millis = this.random.nextInt(1000);
        log.info(String.format("Sleeping for [%d] millis", millis));
        Thread.sleep(millis);
        this.tracer.currentSpan().tag("random-sleep-millis", String.valueOf(millis));
        String s = this.restTemplate.getForObject("http://localhost:" + this.port + "/call",
            String.class);
        return "start/" + s;
    }
}
```

下面是通过模拟不同HTTP调用请求的调用链日志输出。

http://127.0.0.1：3380/hi对应同步请求场景，日志输出内容如下。

```
{"traceId":"8a53757ad20de95f", "parentId":"8a53757ad20de95f", "id":"589f018880591cc3",
"kind":"CLIENT", "name":"GET", "timestamp":1591874786847663, "duration":966816, "localE
ndpoint":{"serviceName":"testsleuthzipkin"},"tags":{"http.method":"GET","http.path":"/hi2"
{"traceId":"8a53757ad20de95f", "parentId":"8a53757ad20de95f", "id":"589f018880591cc3",
"kind":"SERVER", "name":"GET/hi2", "timestamp":1591874786862628, "duration":952958, "lo
calEndpoint":{"serviceName":"testsleuthzipkin"}, "remoteEndpoint":{"ipv4":"127.0.0.1", "por
t":50568}, "tags":{"http.method":"GET", "http.path":"/hi2", "mvc.controller.class":"SampleCo
ntroller", "mvc.controller.method":"hi2", "random-sleep-millis":"916"}, "shared":true}
{"traceId":"8a53757ad20de95f", "id":"8a53757ad20de95f", "kind":"SERVER", "name":"GET
/hi", "timestamp":1591874786361567, "duration":1463194, "localEndpoint":{"serviceName":
"testsleuthzipkin"},"remoteEndpoint":{"ipv4":"127.0.0.1","port":50566},"tags":{"http.metho
d":"GET","http.path":"/hi","mvc.controller.class":"SampleController","mvc.controller.metho
d":"hi"}}
```

http://127.0.0.1：3380/async对应异步请求场景，日志输出内容如下。

```
{"traceId":"f3893e09e6f48a9d", "id":"f3893e09e6f48a9d", "kind":"SERVER", "name":"GET
/async", "timestamp":1591874624865055, "duration":6410, "localEndpoint":{"serviceName":
"testsleuthzipkin"},"remoteEndpoint":{"ipv4":"127.0.0.1","port":50448},"tags":{"http.metho
d":"GET","http.path":"/async","mvc.controller.class":"SampleController","mvc.controller.met
hod":"async"}}
{"traceId":"f3893e09e6f48a9d", "parentId":"f3893e09e6f48a9d", "id":"6ee139b58492a100",
"name":"background", "timestamp":1591874624868865, "duration":559975, "localEndpoint":
{"serviceName":"testsleuthzipkin"}, "tags":{"class":"SampleBackground", "method":"backgro
und", "background-sleep-millis":"559"}}
{"traceId":"f3893e09e6f48a9d", "parentId":"f3893e09e6f48a9d", "id":"16e266e56dcf485f", "
name":"async", "timestamp":1591874624868727, "duration":560300, "localEndpoint":{"servi
ceName":"testsleuthzipkin"}}
```


http://127.0.0.1：3380/start对应同步异步混合请求场景，日志输出内容如下。


```
{"traceId":"be00db6062f19d3f", "parentId":"40e5755b8f8d1b54", "id":"9fe36051dc47a362",
"name":"async", "timestamp":1591875602020796, "duration":639926, "localEndpoint":{"serv
iceName":"testsleuthzipkin"}, "tags":{"callable-sleep-millis":"639"}}
{"traceId":"be00db6062f19d3f", "parentId":"40e5755b8f8d1b54", "id":"688dd48b9d4babe3"
, "name":"async", "timestamp":1591875602020587, "duration":640845, "localEndpoint":
{"serv  iceName":"testsleuthzipkin"}}
{"traceId":"be00db6062f19d3f", "parentId":"be00db6062f19d3f", "id":"40e5755b8f8d1b54",
"kind":"CLIENT", "name":"GET", "timestamp":1591875602007817, "duration":655950,
"localEndpoint":{"serviceName":"testsleuthzipkin"}, "tags":{"http.method":"GET",
"http.path":"/call "}}
{"traceId":"be00db6062f19d3f", "parentId":"be00db6062f19d3f", "id":"40e5755b8f8d1b54"
, "kind":"SERVER", "name":"GET/call", "timestamp":1591875602016042, "duration":647975
"localEndpoint":{"serviceName":"testsleuthzipkin"}, "remoteEndpoint":{"ipv4":"127.0.0.1", "
port":50829},"tags":{"http.method":"GET","http.path":"/call","mvc.controller.class":"Sample
Controller", "mvc.controller.method":"call"}, "shared":true}
{"traceId":"be00db6062f19d3f", "id":"be00db6062f19d3f", "kind":"SERVER", "name":"GET
/start", "timestamp":1591875601814062, "duration":853026, "localEndpoint":
{"serviceName":"testsleuthzipkin"},"remoteEndpoint":{"ipv4":"127.0.0.1","port":50827},
"tags":{"http.method":"GET","http.path":"/start","mvc.controller.class":"SampleController",
"mvc.controller.method":"start", "random-sleep-millis":"191"}}
```


**说明：** 上述代码参考了Spring Cloud Sleuth官方源码实现（Sleuth源码可在GitHub中查找），更多关于Sleuth的有趣实例可参考官网代码。

### ZipkinServerApplication代码实例

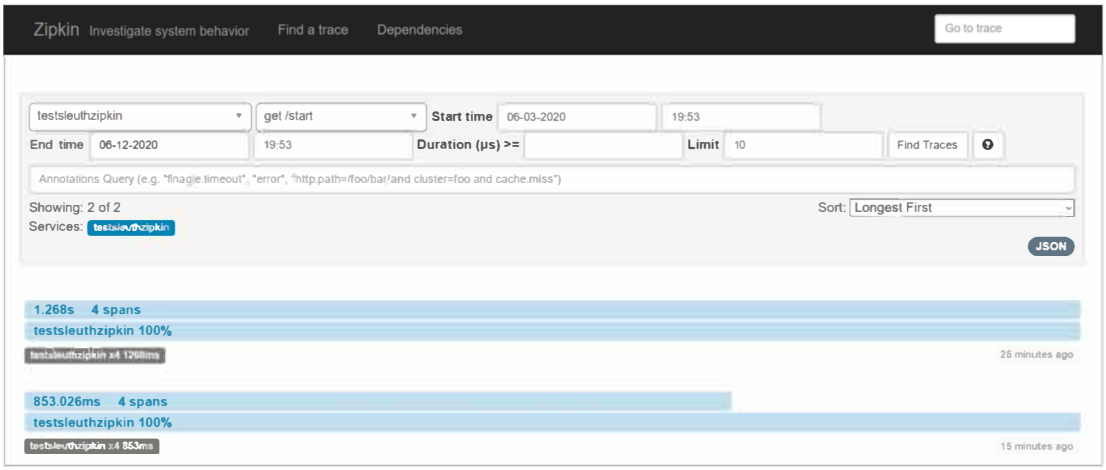
1.引入Maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

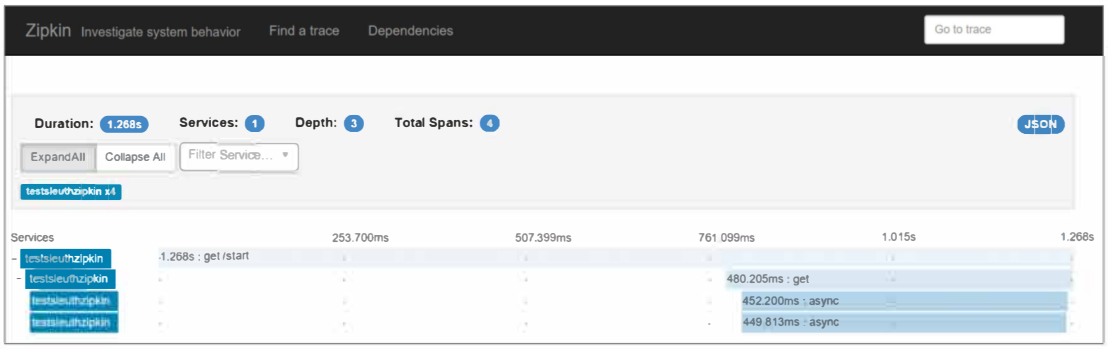
2.使用@EnableZipkinServer启动应用

```
@EnableZipkinServer
@SpringBootApplication
public class ZipkinApplication {
    private static final Logger LOGGER = LoggerFactory.getLogger(ZipkinApplication .class);
    public static void main(String[] args) {
        SpringApplication.run(ZipkinApplication .class,  args);
    }
}
```

访问http://127.0.0.1：9411/zipkin/首页，将展示所有在testsleuthzipkin应用上发生的调用链信息，可以看到一次请求的调用整体延迟和Span数，如下图所示。

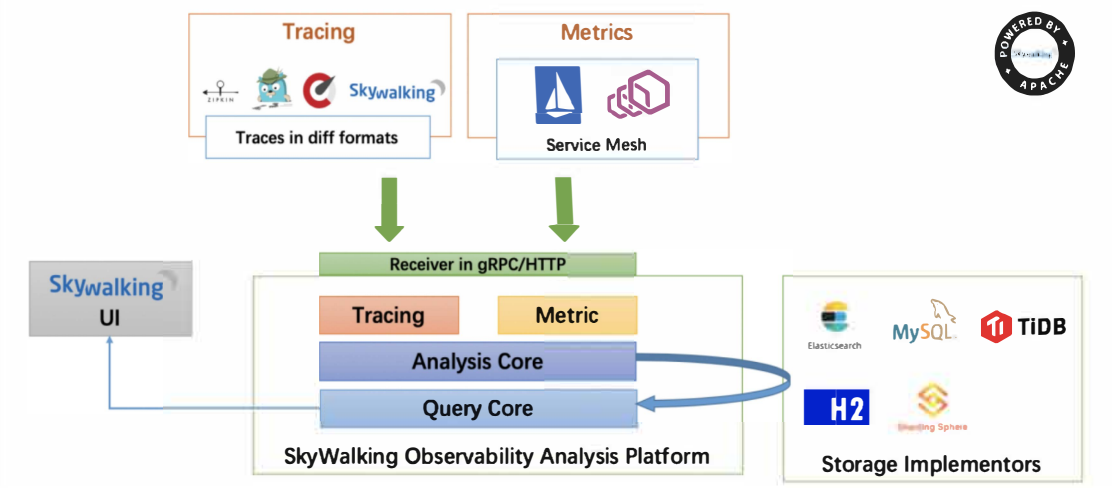



下面一行代表一次完整的请求http://127.0.0.1：3380/start，单击可以查看调用链详情，如下图所示。




## 9.4.4 SkyWalking技术

SkyWalking是国内一个开源并提交到Apache孵化器的产品，是用于收集、分析、聚合、可视化来自不同服务和本地基础服务的数据的可视化的平台。SkyWalking提供了一个可以对分布式系统甚至是跨云服务有清晰了解的简单方法。SkyWalking符合OpenTracing规范，同时提供更加现代化、炫酷的UI，可以更加直观地监控应用。SkyWalking的官方架构如下图所示。




整个系统分为三部分。

- Agent：采集Tracing（调用链数据）和Metrics（指标）信息并上报。
- OAP：收集Tracing和Metrics信息通过Analysis Core模块将数据放入持久化容器中（内存数据库H2、MySQL等），并进行二次统计和监控告警。
- Webapp：前后端分离，前端负责呈现，并将查询请求封装为GraphQL提交给后端；后端通过Ribbon做负载均衡转发给OAP集群，再将查询结果渲染展示。

SkyWalking符合OpenTracing规范，调用链采集数据格式如下。

```
{
    "trace_id": "52.70.15530767312125341",
    "endpoint_name": "Mysql/JDBI/Connection/commit",
    "latency": 0,
    "end_time": 1553076731212,
    "endpoint_id": 96142,
    "service_instance_id": 52,
    "version": 2,
    "start_time": 1553076731212,
    "data_binary": "CgwKCjRGnPvp5eikyxsSXhD//////////8BGMz62NSZLSDM+tjUmS0wju8FQChQAVg
    "service_id": 2,
    "time_bucket": 20190320181211,
    "is_error": 0,
    "segment_id": "52.70.15530767312125340"
}
```

### SkyWalking接入介绍

Idea本地启动：

```
-javaAgent:D:\Workspace\Others\hello-spring-cloud-alibaba\hello-spring-cloud-external-skywalki
ng\Agent\skywalking-Agent.jar
-Dskywalking.Agent.service_name=nacos-provider
-Dskywalking.collector.backend_service=localhost:11800
```

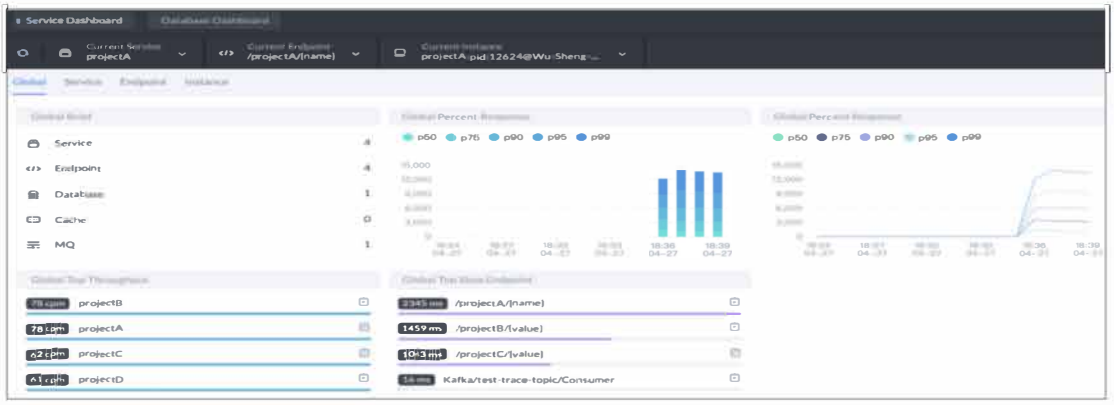
使用jar方式启动：

```
Java-javaAgent:/path/to/skywalking-Agent/skywalking-Agent.jar
-Dskywalking.Agent.service_name=nacos-provider
-Dskywalking.collector.backend_service=localhost:11800 -jar yourApp.jar
```

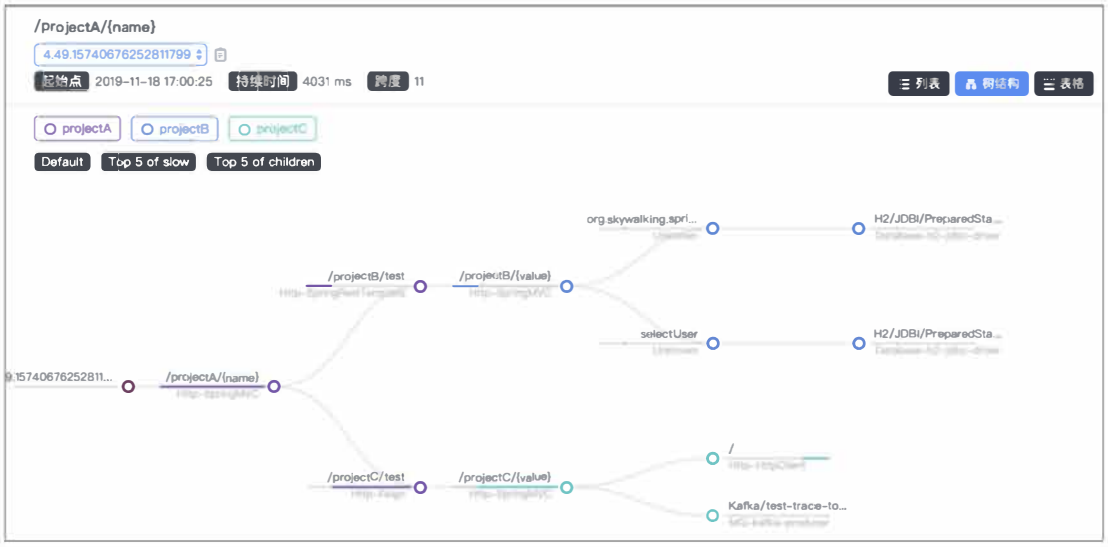
SkyWalking的TraceID与日志组件（Log4j、Logback、ELK等）的集成：

```
<appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <layout
        class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.
          TraceIdPatternLogbackLayout">
            <pattern>
                %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36}
                    - %tid - %msg%n
            </pattern>
        </layout>
</appender>
```

SkyWalking的页面展示，以及整体服务调用情况统计DashBoard，如下图所示。

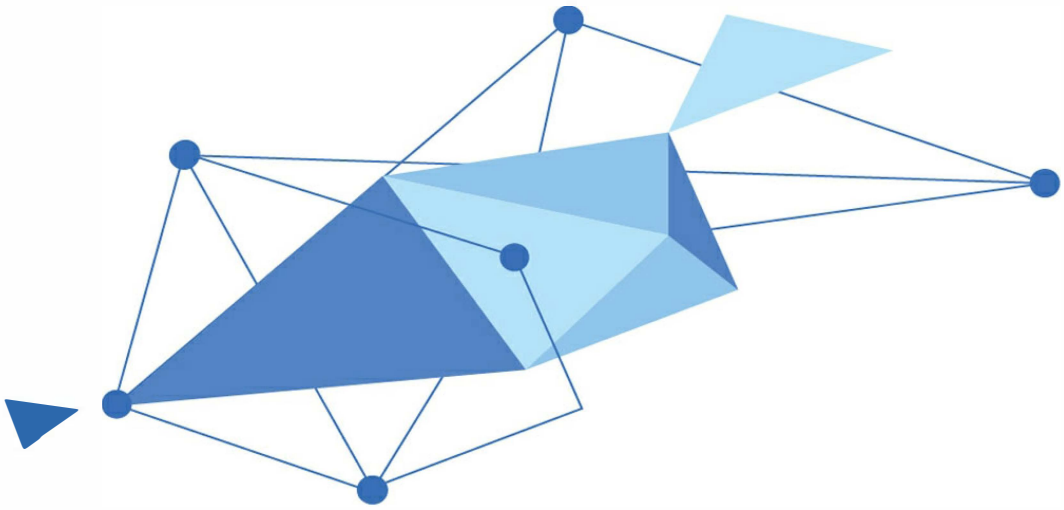



调用链详细信息展示，如下图所示。



## 9.5 小结

监控系统按照原理和作用大致可以分为三类：度量类（Metrics）、日志类（Log）、调用链类（Tracing）。对于不同的监控对象和监控数据类型，需要采用对应的技术栈和监控手段。对于Spring Boot微服务架构，除了开发框架自带的生产就绪组件能够支持服务的健康状态指标查询，我们还需要引入第三方ELK框架作为日志监控组件，与Zipkin和SkyWalking等组件共同实现调用链监控。



# 进阶篇

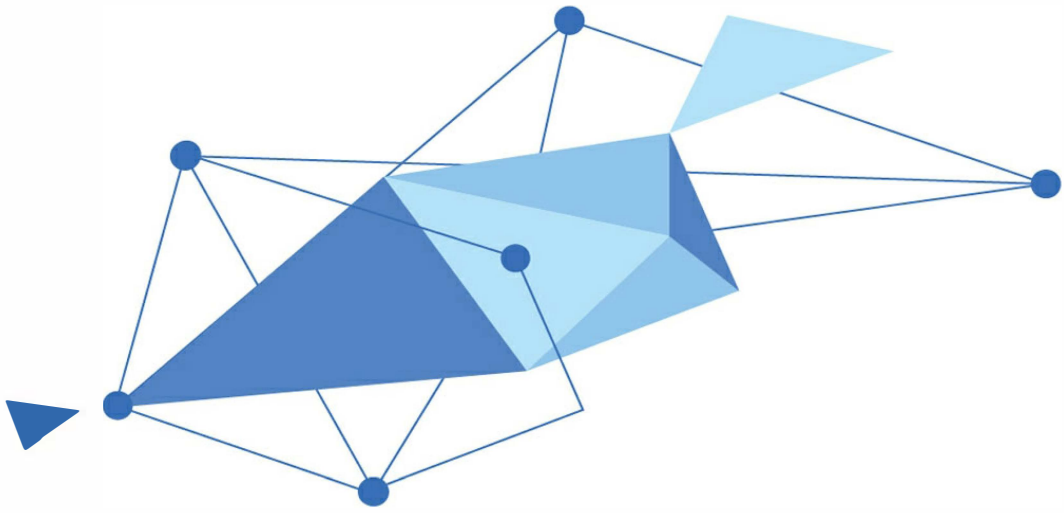
本篇内容

互联网技术的飞速发展、用户规模的扩大、业务需求的快速更新和产品的持续迭代演进都对系统的规模扩展、资源消耗、快速响应能力带来了更大的挑战。

在微服务开发领域，Spring 5集成Reactor响应式框架为Java开发者带来了编程模型和编程范式的革命性的技术。从Spring Boot 2.x到最新的Spring Cloud生态体系，Spring微服务框架在应对业务的快速、响应、扩展性的诉求上一直保持着持续的迭代和演进。

在微服务运行和容器编排领域，Kubernetes已经成为事实上的容器运行编排标准。本篇会介绍Kubernetes的架构理念和关键组件，以及Kubernetes与Spring Cloud生态的融合发展等相关内容。

在微服务架构的发展趋势上，我们将介绍云原生应用架构，以及微服务目前关注的两个技术领域：Service Mesh服务网格及Serverless无服务计算框架。



# 进阶篇

本篇内容

互联网技术的飞速发展、用户规模的扩大、业务需求的快速更新和产品的持续迭代演进都对系统的规模扩展、资源消耗、快速响应能力带来了更大的挑战。

在微服务开发领域，Spring 5集成Reactor响应式框架为Java开发者带来了编程模型和编程范式的革命性的技术。从Spring Boot 2.x到最新的Spring Cloud生态体系，Spring微服务框架在应对业务的快速、响应、扩展性的诉求上一直保持着持续的迭代和演进。

在微服务运行和容器编排领域，Kubernetes已经成为事实上的容器运行编排标准。本篇会介绍Kubernetes的架构理念和关键组件，以及Kubernetes与Spring Cloud生态的融合发展等相关内容。

在微服务架构的发展趋势上，我们将介绍云原生应用架构，以及微服务目前关注的两个技术领域：Service Mesh服务网格及Serverless无服务计算框架。

## 10.1 响应式编程

针对响应式编程，我们会介绍采用响应式编程的动机，并解释响应式编程的含义，然后讲解目前主流的响应式技术框架。

### 10.1.1 响应式编程的动机

当前大多数公司使用Spring或者Spring Boot 1.x技术栈开发后端业务的Web服务器。假设在正常情况下，我们将Tomcat线程池配置为200（最大并发线程数），如果每个人的请求响应时间在200ms以内，并且根据每个请求的超时时间、资源限制和其他因素，粗略估算后，我们得出一个结论，系统在单位时间内最高可以支持1000个用户同时在线访问业务资源。然而，当业务场景是支持一个促销活动时，可能会出现用户规模快速增长或流量（QPS）激增的情况。这时，Tomcat服务器会出现短期资源耗尽及性能急速下降的问题，Tomcat后台爆出大量的服务超时、资源不可用、网页刷新慢等异常，而用户的直观体验就是服务不可用。

下面我们以Tomcat 7.x作为试验对象来模拟上述情景，使用Bench Mark 注 的方法比较客户端随着请求的增加对响应时间（延迟）、系统吞吐、负载均值等指标变化情况的影响。

【测试环境】

OS:Linux Red Hat 4.4.7-11

Memery:16GB

CPU: 4 Processor(Intel(R) Xeon(R))

JDK:1.8

Web服务器：Tomcat 7.x

测试工具：Jmeter–2.9

Monitor Tools: Jconsole, Jvisualvm, Jstat

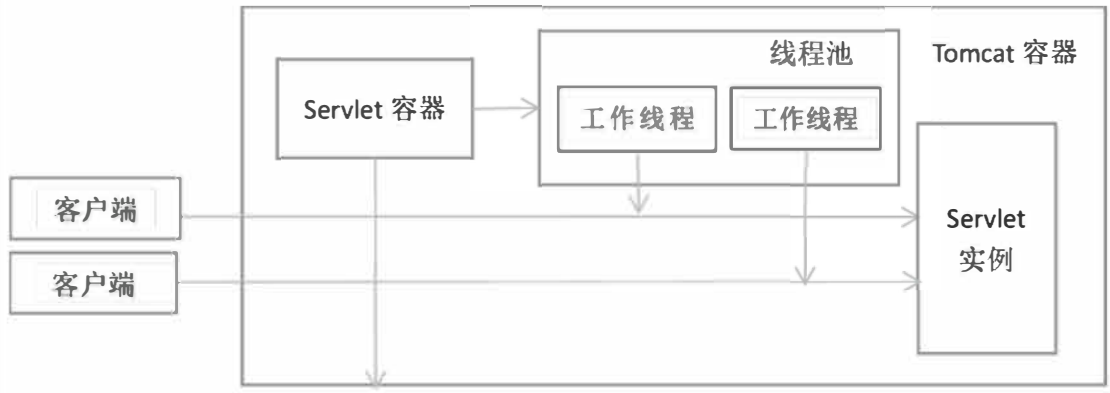
【准备】

在Web容器中只执行一个Servlet服务实例，并且在请求方法中只返回一个“hello”响应。Tomcat模拟200个线程场景，不同线程数对应的服务性能指标结果如下表所示。

| 客户线程数 | 延迟时间（ms） | 吞吐量 | 系统负载均值 |
| --- | --- | --- | --- |
| 100 | 230 | 422 | <1 |
| 500 | 317 | 1545 | <1 |
| 1000 | 382 | 2415 | <1 |
| 1500 | 3269 | 197 | <1 |

从上面Bench Mark的试验结果中发现，当客户请求的线程数增加时，用户的响应延迟时间逐渐增大，同时服务吞吐量也是增大的；但客户并发请求数增长超过1000时，用户的吞吐量下降，并且客户端显示大量请求超时异常：client Non Http response message Read time out。下图是我们从Jconsole中看到的大量的运行态线程。

在默认情况下，在未开启Servlet 3.1异步功能特性时，Tomcat的Servlet处理逻辑是采用同步的编程模型，如下图所示，一个HTTP请求后端对应一个工作线程。




上图是一个典型的基于多线程模型下的Tomcat请求处理模型。由于在Servlet 3以前，Tomcat主要采用同步阻塞等待模式，当请求发生时，就会从线程池中加载对应的工作线程来处理HTTP请求；当客户端请求增加时，对应的工作线程会随之线性增长。当工作线程达到最大时，Tomcat就没有多余的线程供客户端使用，而剩余的请求缓存在Tomcat等待队列中，在请求超时后便会发生Timeout的异常。

可以看到，这种一个请求通过使用一条线程去推动执行控制流的方式是缺少灵活度和响应力的。明显的问题是，当工作线程调用下游服务或者数据库时，如果执行了一个慢操作，那么Tomcat中的工作线程会处于阻塞状态，同步等待在交互边界。在这种情况下，系统的资源被无效占用，当出现大量线程被阻塞占用时，就会发生客户端大量请求重试和重连情况，Web服务可能就会出现工作线程耗尽、内存溢出、雪崩效应、服务不可用等情况。

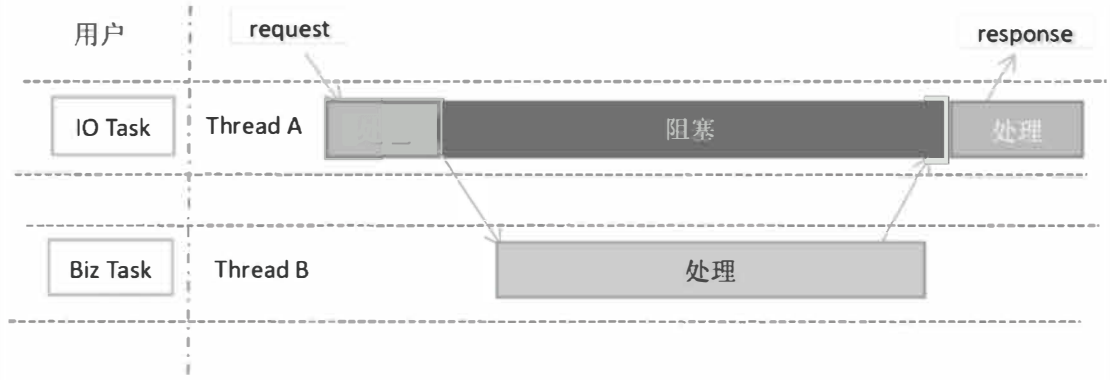
针对用户激增或流量增长的问题，我们常用的解决方案是水平增加硬件资源、扩容服务实例来增加系统的容量。当流量下降时，我们可以缩容服务，降低系统容量。这种依靠增加资源提升性能的方式，在解决系统的可伸缩性问题上会受到系统内部的结构性瓶颈的限制。而这个问题从理论到实践也符合阿姆达尔定律（Amdahl's Law），并且被Gunther的通用可伸缩模型（Gunther's Universal Scalability Model）所证实。所以，我们需要构建一个弹性系统，使系统的资源能够得到更好利用，当流量增加时，我们的处理线程或者资源是收敛的，不会随着请求数量的增加使系统消耗也随之线性增长。

针对同步线程阻塞等待的问题，另外一种解决方案是采用Reactor设计模式。Netty就是一个典型的采用Reactor设计模式的响应式系统。这种系统架构的特征是基于事件触发机制，将服务器端的所有行为抽象成事件并注册监听等待执行，当对应事件到来时会由对应的处理器（Handler）处理，而配合响应式系统最好的方式是将所有任务分解成异步非阻塞的事件，方便异步执行。

总之，针对Java的异步（Asynchronous）编程模型，我们经常有下面两种解决方式。

- 基于多线程的方式，进行Callback。
- 基于Reactor设计模式，使用事件触发机制实现并发处理。

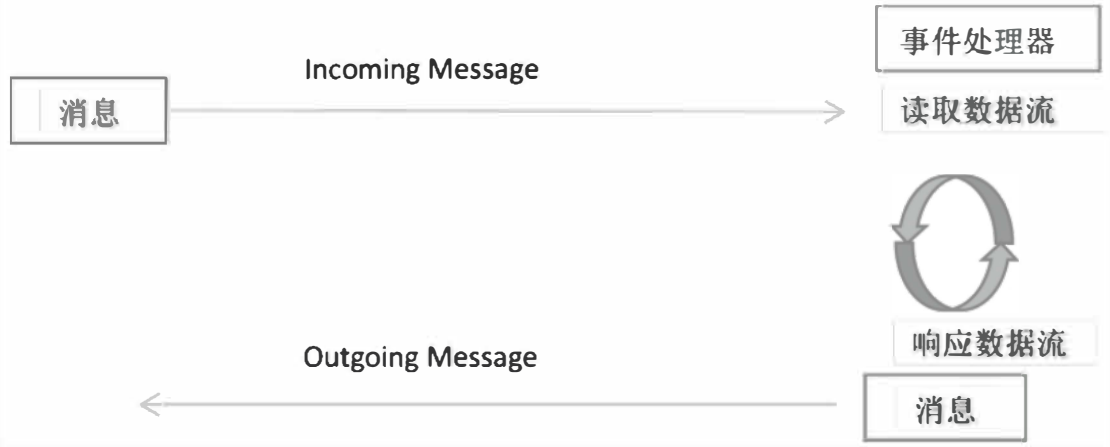
对于传统的Web服务器，我们大多使用第一种方式即异步线程回调的机制，下图展示的是基于线程回调方式的工作流程。




多线程方式的异步编程模型容易引起CPU资源的浪费，真正只有少部分CPU时间片用来处理业务逻辑，而线程的大部分时间都被阻塞在I/O事件等待，这对于稀缺的线程资源来说，是非常不合理的。另外，我们知道Java没有协程的概念，所以不能像有些新型编程语言那样使用“绿色线程”注解决线程资源浪费的问题。虽然，JDK 8引入了CompleteFuture，但是异步回调很难组合，并产生了另外的问题——回调地狱，而且这种方式本质上还是基于线程控制流的方式，采用命令式编程对业务逻辑进行编排和管理。

针对系统的“韧性”或者容错性问题，微服务架构的开拓者Netflix使用Hystrix技术来解决。Hystrix可以解决线程阻塞产生的同步等待，实现线程熔断和降级的目的。虽然可以保护Web服务器在极端情况下的宕机，但是这种做法没有从根本上解决服务器面对用户增长、流量增加带来的服务响应力。当系统熔断发生时，用户的数据可能会丢失，客户端对服务器的消息处理能力和响应能力是无感知的，而使用响应式编程的“背压”特性，客户端可以实时感知并动态调节发送速率和异常响应，服务器也可以更好地保证系统的整体韧性。

异步编程的另外一个模式是基于事件触发机制。将复杂的业务逻辑拆分为异步非阻塞的独立执行单元，这些独立的逻辑执行步骤由数据流触发，通过数据流有效地推动业务逻辑的执行。通过函数的响应、组合达到对数据流的控制、过滤、组合、映射。对这种数据流响应模式的处理流程如下图所示。

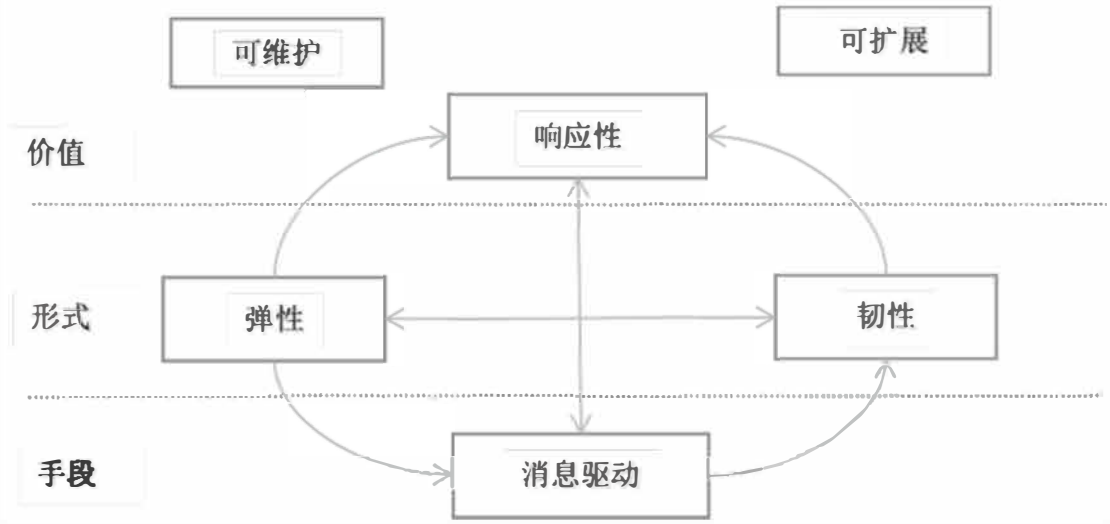



总之，还有很多可以改善当前Java Web服务编程开发模式和系统架构现状的方式，我们就不一一列举了。可以说，传统Spring MVC架构的阻塞式编程模型的性能瓶颈是响应式系统架构和响应式编程模型开发的主要动机，我们需要考虑如何满足系统更加健壮、更具有弹性、更快响应、更

加灵活、更易扩展等特性，从Spring早期基于XML配置进行Web开发模型，到Spring Boot 1.x基于注解方式简化Web的开发方式，再到Spring 5和Spring Boot 2.x全面拥抱响应式编程，软件技术的发展总是围绕成本、效率、性能等生产要素持续迭代演进，通过技术升级逐步实现降低开发成本、快速响应业务、增强用户体验的目标。

## 10.1.2 响应式宣言

上一节我们解释了传统编程模型存在的问题和响应式系统架构及使用响应式编程开发的动机，下面我们介绍一下响应式宣言。响应式宣言是一组架构与设计原则，符合这些原则的系统可以认为是响应式系统。而响应式系统与响应式编程是不同层面的内容。下图基本概括了响应式宣言的主要特性和核心理念。




**响应性**

快速、一致的“响应性”是响应式宣言中最重要的特性。即时响应是可用性和实用性的基石，而更加重要的是，即时响应意味着可以快速检测到问题并且有效地对其进行处理。即时响应的系统专注于提供快速而一致的响应时间，确立可靠的反馈上限，以提供一致的服务质量。这种行为将简化错误处理、建立最终用户的信任，并促使用户与系统形成进一步的互动。

**弹性**

“弹性”使系统在不断变化的工作负载下依然保持即时响应性。反应式系统可以对输入（负载）的速率变化做出反应，比如增加或者减少用于服务这些输入（负载）的资源。这意味着设计上并没有中央瓶颈点，可以对组件进行分片或者复制，并在它们之间分别输入（负载）。通过提供相关的实时性能指标，反应式系统能支持预测及反应式的伸缩算法。这些系统可以在常规的硬件和软件平台上实现成本高效的弹性。

**韧性**

“韧性”指系统在出现失败时依然保持即时响应性。这适用于高可用的任务关键型系统，如果不具备回弹性，系统将在发生失败后丢失即时响应能力。回弹性通过复制、遏制、隔离及委托来实现系统的响应力。失败的扩散被遏制在每个组件内部，与其他组件相互隔离，从而确保系统某部分的失败不会危及整个系统，并能独立恢复。每个组件的恢复都被委托给了另一个组件，此外在必要时可以通过复制来保证高可用。

**消息传递**

反应式系统依赖异步的“消息传递”，它可以确保松耦合、隔离、位置透明的组件之间有着明确边界。这一边界还提供了将失败作为消息委托出去的方式。使用显式的消息传递，可以在系统中塑造并监视消息流队列，并在必要时应用回压，从而实现负载管理、弹性及流量控制。使用位置透明的消息传递作为通信手段，使得跨集群或者在单个主机中使用相同的结构成分和语义来管理失败成为可能。非阻塞式通信使得接收者可以只在活动时消耗资源，从而减少系统开销。

综上所述，响应式宣言也是规范响应式开发的一个行为准则。我们将具备这些特性的系统称为响应式系统。

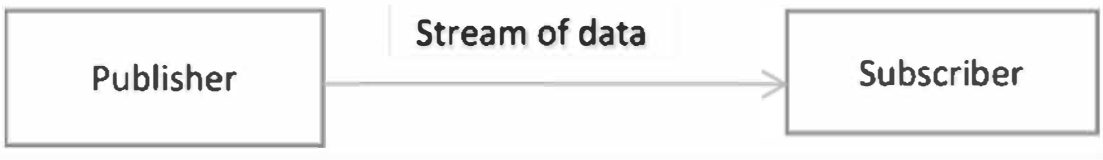
### 10.1.3 响应式编程详解

响应式编程是一种基于异步数据流驱动、响应式、使用声明式范式的编程模型，需要遵循一定的响应式编程开发规范，并且有具体的类库实现。响应式编程基于数据流而不是控制流进行业务逻辑的推进。

**响应式编程与设计模式**

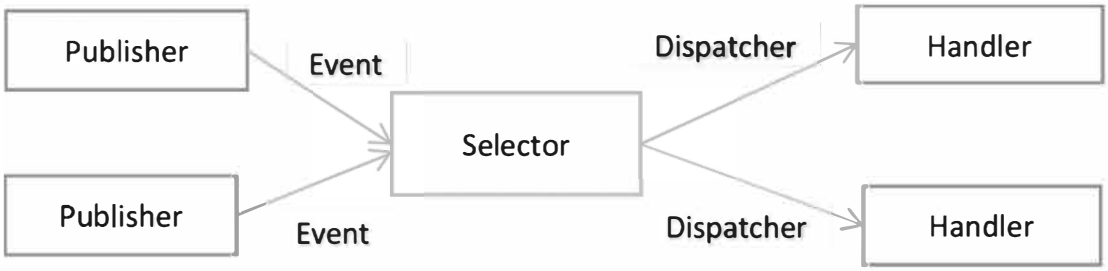
在面向对象编程语言中，响应式编程通常以观察者模式呈现。将响应式流模式和迭代器模式比较，其主要区别是，迭代器基于“拉”模式，而响应式流基于“推”模式。

在命令编程范式中，开发者掌握控制流，使用迭代器遍历“数据”，使用hasNext（）函数判断数据是否遍历完成，使用next（）函数访问下一个元素。在响应式编程模式中，使用观察者模式，数据由消息发布者（Publisher）发布并通知订阅者（Subscriber），而这种观察者模式本身在基于事件监听机制的响应式系统架构中被广泛使用。Java早期的Swing界面设计也是基于视图事件触发业务响应的系统工作模式。所以，从设计模式的角度讲，响应式编程并不是新鲜事物，只是响应式编程将监听的对象扩展到了更大范围：静态或者动态的Stream数据流，如下图所示。




响应式编程还借鉴了Reactor设计模式，我们通常会在高性能NIO网络通信框架中见到Reactor设计模式的身影，用来实现I/O多路复用。其基本思想是将所有要处理的I/O事件注册到一个中心

I/O多路复用器上，同时主线程阻塞在多路复用器上，通过轮询或者边缘触发的方式来处理网络I/O事件。当有新的I/O事件到来或准备就绪时，多路复用器返回并将事件分发到对应的处理器中。Reactor设计模式和响应式编程类似，它们都不主动调用某个请求的API，而是通过注册对应接口，实现事件触发执行，如下图所示。




**响应式编程与响应式架构**

响应式编程很容易和响应式架构混为一谈。前面我们介绍了响应式宣言中的构建软件架构原则，把符合这些原则的系统称为响应式系统。如果说响应式系统与响应式编程之间具有什么关系，那就是响应式系统的架构风格是响应式的，而响应式编程是实现这个架构风格的最佳实践。从宏观角度看，响应式系统由各种不同组件相互操作、调用组成，共同响应用户请求。响应式系统涉及通信协议、I/O模型、网络传输、数据存储等多方面因素，保障系统在响应力、扩展性、容错、灵活性各方面表现出“实时”“低延迟”“轻量”“健壮”的系统特性。而响应式编程可能是这个大的系统架构下的一部分。另外，响应式系统一般是消息驱动的，而响应式编程是事件驱动的。

**消息驱动与事件驱动**

响应式宣言指出了两者的区别：“消息驱动”中消息数据被送往明确的目的地址，有固定导向；“事件驱动”是事件向达到某个给定状态的组件发出的信号，没有固定导向，只有被观察的数据。

- 在一个消息驱动系统中，可寻址的接收者等待消息的到来然后响应消息，否则保持休眠状态，消息驱动系统专注于可寻址的接收者。响应式系统更加关注分布式系统的通信和协作以达到解耦、异步的特性，满足系统的弹性和容错性，所以响应式系统更倾向于使用消息驱动模式。
- 在一个事件驱动系统中，通知的监听者被绑定到消息源上。这样当消息被发出时，它就会被调用，所以，响应式编程更倾向于事件驱动。

**响应式编程与函数式编程**

响应式编程同时容易和函数式编程混淆。函数式编程（Functional Reactive Programming，FRP）在二十年前就被Conal Elliott精确地定义了。在函数式编程中，函数是第一类（first-class）公民，函数式编程由“行为”和“事件”组成。事件是基于时间的离散序列，而行为是不可变的，是随着时间连续变化的数据。函数式编程与响应式编程相比，它更偏重于底层编码的实现细节。

从Java 8开始，Lambda表达式的引入为Java添加了函数式编程的特性，函数式编程提供了闭包的强大功能。Java中的Lambda表达式通常使用（argument）->（body）语法书写，如下所示：

```
(arg1， arg2...) -> { body }
(type1 arg1， type2 arg2...) -> { body }
```

下面是一些典型的Lambda表达式及其函数式接口：

- Consumer<Integer> c=(int x)->{System.out.println(x)};
- BiConsumer<Integer, String>b=(Integer x, String y)->System.out.println(x+":"+y);
- Predicate<String>p=(String s)->{s==null};

在Java 8中新增加了@FunctionalInterface接口，用于指明该接口类型是根据Java语言规范定义的函数式接口。Java 8还声明了一些Lambda表达式可以使用的函数式接口。下面是匿名类和使用函数式编程方式的对比示例。

首先，使用@FunctionalInterface定义一个函数式编程接口。

```
@FunctionalInterface
public interface TaskInterface {
    public void invoke();
}
```

然后，分别使用内部类和Lambda表达式两种方式执行业务逻辑。

```
public class Test {
    public static void execute(TaskInterface task) {
       task.invoke();
}
public static void main(String [] args) {
    //使用内部类进行回调
    execute(new WorkerInterface() {
        @Override
        public void invoke() {
            System.out.println("invoked using Anonymous class");
        }
    });
    //使用 Lambda 表达式
    execute( () -> System.out.println("invoked using Lambda expression") );
    }
}
```

可以看到，在函数式编程中，Lambda表达式允许将一个箭头函数作为参数进行传递，这样的语法表达更加简洁，而本质上由编译器推断并帮助实现转换包装为常规代码。因此，可以用更少的代码来实现相同的功能。而响应式编程的重点是基于“事件流”的异步编程范式，响应式编程通过函数编程方式简化面向对象语言语法的臃肿。响应式编程解决问题的流程是：将一个大的问题拆分为许多独立的小的步骤，而这些小的步骤都可以异步非阻塞地执行；当这些小的子任务执行完，它们会组成一个完整的工作流，并且这个工作流的输入输出都是非绑定的。实现响应式编程的关键就是“非阻塞”，执行线程不会因为竞争一个共享资源而陷入阻塞等待，空耗资源，并且最大化地利用物理资源。

**响应式编程与命令式编程模式**

响应式编程是一种声明式的编程模型，与之相对应的就是命令模式（线程控制流）的编程模型。大家对命令式编程模式比较熟悉，下面是一段常见的基于命令式编程模式的代码：

```
Int a = 1;
Int b =2;
Int sum =  a+b;
System.out.println( “first time ,  a and b sum is :” +sum);
a = 3;
b = 5;
System.out.println( “second time,  a and b sum is :” +sum);
```

上述代码是通过变量的赋值并通过加法计算响应数据之间的对应算数关系结果。但是，这个代码有一个潜在的问题，当我们给这两个变量重新赋值时，第二次的Sum值却没有变化，与我们的期望不符，原因是缺少了执行相加的命令指令。

响应式编程的目的是通过“不可变操作符”固定这种数据，构建数据之间的关系，并正确输出结果，不会因为操作命令的遗忘和缺失导致结果的偏差，造成对应关系和结果错误，下面我们看一下如何使用响应式编程方式来固化这种模式。

下面使用Java 9的Flow API实现两个数的相加功能，按照相同思路，当传入的变量不同时，输出的Sum值也会随着变化，我们把这种对应关系构建为一个声明公式，代码实现如下：

```
SubmissionPublisher<Integer> publisher = new SubmissionPublisher<>();
publisher .subscribe(
    new Flow.Subscriber<Integer>(){
       private Integer sum = 0 ;
       @Override
       public void onNext(Integer i){
          sum += i;
        }
       @Override
       public void onComplete()
       {
          System.out.println( “a and b sum is :” + sum);
        }
        //省略
    }
);
Arrays.asList(1，2 ).stream.forEach(publish::submit);
Arrays.asList(3，5 ).stream.forEach(publish::submit);
publisher.close();
```

从结果看，响应式编程模式的两次Sum值和输入的数值一致，能够达到预期效果。从这个例子中，我们已经初步接触到了响应式编程中数据源也就是事件发布者（Publisher），还有就是事件的监听回调函数集合——消费者（Subscriber）。消费者会根据next、error、complet触发函数对应关系的执行，以及数据的操作符操作，由于消费者的不可变性，可以根据原生的数据结构生成新的数据结构。相比命令式编程，响应式编程使用操作符表述了一个通用业务执行逻辑，一般可以组合达到预期效果，一般的操作符还包含map、filter、reduce等函数，这里就不再赘述了。

## 10.1.4 编程范式

“普通的工程师堆砌代码，优秀的工程师优化代码，卓越的工程师简化代码”。如何写出优雅整洁的代码，不仅是一门学问，也是软件工程的重要一环。在上一节中，我们简单介绍了响应式编程的编程范式，本节我们进一步从开发者的视角、系统的性能、满足用户需求等方面讨论不同编程范式的使用场景和特性优势。

编程范式，又称为编程模型，泛指软件编程过程中使用的编程风格，一般不同的编程范式具有不同的语法特性和差异。目前软件开发技术中常用的典型编程范式有以下几种。

- 命令式编程。
- 面向对象编程。
- 声明式编程。
- 函数式编程。

因为每一个编程范式都有很长的发展历史，在编程语言支持上有不同的标准、组织和语法规范等，本节的目的是希望通过对这些编程范式的介绍，可以帮助我们更好地理解响应式编程范式。

**命令式编程**

命令式编程是非常传统的软件编程方式，命令式编程由不同的逻辑执行步骤组成，通过一步步指令的执行达到业务逻辑的推进，这种方式也称为过程式编程。命令式编程的执行过程非常符合计算机的执行步骤。C语言是命令式编程的典型代表，它更关注的是机器域底层的内存、指令计算、输入输出。在C语言中，我们经常看到大段的过程式指令、各种if/else/for等控制语句、表达式、数据变量的操作、赋值等指令，这种纯指令开发方式要求开发者对计算机的底层工作原理有非常深刻的理解，而且一个指令出现偏差往往会产生不可预知的错误。同时，命令式编程模式的运维也是难度非常高的。

**面向对象编程**

面向对象编程可以说是编程领域的一个分水岭，开启了高级程序语言在软件开发上的统治阶段。面向对象编程从问题域出发，将封装、继承、多态的语言特性映射到我们的现实世界。在面向对象编程里，业务问题被抽象成类、接口模板，数据和行为被统一封装在对象内部，作为程序的基本组成单元。面向对象编程范式在提升软件重用性、灵活性和扩展性上比过程式编程更进一步，C++、Java作为面向对象编程语言的代表，屏蔽了机器底层的内存管理和机器域的管理细节。而面向对象编程虽然有较高的开发效率，但是降低了代码的运行效率，这也限制了面向对象编程在性能要求苛刻场景下的应用。

**声明式编程**

声明式编程受当前“约定优于配置”理念的影响，在软件编程开发领域中被大量应用。声明式编程范式的好处是可以通过声明的方式实现业务逻辑，不需要陷入底层具体的业务逻辑实现细节。声

明式编程范式关注的焦点不是采用什么算法或者逻辑来解决问题，而是描述、声明解决的问题是什么。当你的代码匹配预先设定好规则，业务逻辑就会被自动触发执行。

很多标记性语言，如HTML、XML、XSLT，就遵循声明式编程范式，而Spring Boot基于注解方式的编程模型也是声明式编程的一个代表。Spring框架依赖AOP和IoC编程思想降低了开发者对底层逻辑业务细节的了解程度。例如在Spring Boot中，通过@Transactional注解可以声明一个方法具备事务性的操作，当异常发生时，事务会自动回滚，保证业务逻辑的正常和数据一致性。发生在@Transactional注解背后的实现细节，开发者可以不去关心。

**函数式编程**

在函数式编程范式中，函数无疑是一等公民，函数式编程最具魅力或者最重要的特性就是不可变性。它的不可变性表现在函数式编程表达式的执行结果，只取决于传入函数的参数序列，不受数据状态变化的影响。

函数式编程中的Lambda在Java 8中被引入，可以看成是两个类型之间的关系：一个输入类型和一个输出类型。Lambda演算就是给Lambda表达式一个输入类型的值，它就可以得到一个输出类型的值。这个计算过程也是函数式代码对映射的描述，因为函数式代码的抽象程度非常高，所以也意味着函数式代码有更好的复用性。

函数式编程和命令式编程相比，更加关注消息或者数据的传递，而不像命令式编程，关注的是指令控制流。共享数据的状态在多线程环境下会存在资源竞争的情况，往往我们需要把额外的精力投入到冲突的解决、数据状态的维护中。而函数的不可变性保证了数据在传递处理过程中不会被篡改，也不需要依赖外部的锁资源或者状态来维护并发。所以函数式编程在多核处理器中具有天然的并发性，可以最大化地利用物理资源实现并行处理功能。

目前，在JVM体系中，已经出现了越来越多函数式编程范式的语言，例如Scala、Groovy、Clojure等。在当前计算机多核、数据优先、高性能的诉求下，函数式编程具有更广阔的发展前景和未来。然而有利总会有弊，函数式编程的语法相比面向对象编程更晦涩，在大规模工程化的协调配合中，还是需要我们去权衡利弊。因为无论哪种语言范式，本质上都是工具，最终目的都是为业务服务。

## 10.2 响应式技术框架

目前在后端Web编程和微服务编程领域，存在多种响应式编程技术框架。本节我们从响应式编程规范开始介绍，进一步加深对响应式编程的理解。

### 10.2.1 响应式编程规范

对于响应式编程来说，响应式流是一种非阻塞、响应式、异步流处理、支持背压的技术标准，包括运行时环境（JVM和JavaScript）及网络协议。JDK 9发布的Flow API（java.util.concurrent.Flow）和响应式流规范呼应，成为响应式编程事实上的标准。

响应式流规范提供了一组最小化的接口、方法和协议来描述必要的操作和实体对象。

- Publisher：消息发布者。发布者只有一种方法，用来接受订阅者进行订阅（Subscribe）。T代表发布者和订阅者之间传输的数据类型，接口声明如下：

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

- Subscriber：消息订阅者。当接收到Publisher的数据时，会调用响应的回调方法。注册完成时，首先会调用onSubscribe方法，参数Subscriptions包含了注册信息。订阅者有四种事件方法，分别在开启订阅、接收数据、发生错误和数据传输结束时被调用，接口声明如下：

```
public interface Subscriber<T> {
    //注册完成后，首先被调用
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();}
}
```

- Subscription：连接Publisher和Subscriber的消息交互的操作对象。Subscriber可以请求数据（request），或者取消订阅（cancel）。当请求数据时，参数“long n”表示希望接收的数据量，防止Publisher发送过多的数据。一旦开始请求，数据就会在流中传输。每接收

一个，就会调用onNext（T t）；当发生错误时，onError（Throwable t）被调用；在传输完成后，onComplete（）被调用。接口声明如下：

```
public interface Subscription {
    //请求数据，参数 n 为请求的数据量，不是超时时间
    public void request(long n);
    //取消订阅
    public void cancel();
}
```

- Processor：同时充当Subscriber和Publisher的组件。可以看出，Processor接口继承了Subscriber和Publisher，是流的中间环节，接口声明如下：

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}
```

响应式流中的数据从Publisher开始，经过若干Processor，最终到达Subscriber，即完整的数据管道（Pipeline）。

**背压（Back Pressure）**

在响应式编程规范中，响应式编程采用异步的发布-订阅模式。数据由Publisher推送消息给Subscriber。这种模式容易产生的问题是，当Publisher即生产者产生的数据速度远远大于Subscriber即消费者的消费速度时，消费者会承受巨大的资源压力（Pressure）而可能崩溃。

为了解决以上问题，数据流的速度需要被控制，即流量控制（Flow Control），以防止快速的数据流压垮目标。因此需要反压，即背压（Back Pressure），生产者和消费者之间需要通过一种背压机制来相互操作。这种背压机制要求是异步非阻塞的，如果是同步阻塞的，则消费者在处理数据时，生产者必须等待，会产生性能问题。

## 10.2.2 Java Flow API

从Java 9开始，增加了java.util.concurrent.Flow API，实现了响应式流规范（Reactive Stream Specification），并且把响应式流标准的接口集成到了JDK中。它和响应式流标准接口定义完全一致，之前需要通过Maven引用的API，Java 9之后可以直接使用了。响应式流的标准Maven依赖如下：

```
<dependency>
    <groupId>org.reactivestreams</groupId>
    <artifactId>reactive-streams</artifactId>
</dependency>
```

Java 9通过java.util.concurrent.Flow和java.util.concurrent.SubmissionPublisher实现了响应式流Flow类中定义的四个嵌套的静态接口，用于建立流量控制的组件，Publisher在其中生成一个或多个数据项供Subscriber使用。下面是Java 9 Flow API的核心组件。

- java.util.concurrent.Flow：这是Flow API的主要类，该类封装了Flow API的所有重要接口。需要说明的是，这个类声明为final类型，所以我们无法扩展它。
- java.util.concurrent.Flow.Publisher：每个发布者都需要实现此接口，每个发布者都必须实现它的subscribe方法，并添加相关的订阅者以接收消息。
- java.util.concurrent.Flow.Subscriber：每个订阅者都必须实现此接口，订阅者按照严格的顺序调用方法，此接口有下面四种方法。
  - onSubscribe：这是订阅者订阅了发布者后接收消息时调用的第一个方法。通常我们调用subscription.request就开始从处理器（Processor）接收项目。
  - onNext：当发布者收到项目时调用此方法，这是我们实现业务逻辑来处理流并向发布者请求更多数据的方法。
  - onError：当发生不可恢复的错误时调用此方法，我们可以在此方法中执行清理操作，例如关闭数据库连接。
  - onComplete：这就像finally方法，在发布者没有发布其他项目或者发布者关闭时调用。可以用来发送流成功处理的通知。
- java.util.concurrent.Flow.Subscription：用于在发布者和订阅者之间创建异步非阻塞连接。订阅者调用请求（request）方法来向发布者请求项目。它还有取消订阅（cancel）的方法，即关闭发布者和订阅者之间的连接。
- java.util.concurrent.Flow.Processor：此接口同时扩展了Publisher和Subscriber接口，用于在发布者和订阅者之间转换消息。
- java.util.concurrent.SubmissionPublisher：这个类是对Publisher接口的实现，它将提交的项目异步发送给当前订阅者，直到它关闭。它使用Executor框架，我们将在响应式流示例中使用该类来添加订阅者，然后向其提交项目。

**Java 9 Flow API接入实例**

下面使用Java 9 Flow API实现一个简单的发布消息订阅的例子。

1.创建一个Item类，作为创建从发布者到订阅者之间的流消息的对象

```
public class item{
    private int id;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public item(String s) {
        this.id = s;
    }
    public item() {
    }
    @Override
    public String toString() {
        return "[id="+id+]";
    }
}
```

2.实现一个帮助类，创建一个Item列表

```
public class ItemHelper {
    public static List<Item> getItems() {
        Item e1 = new Item (1);
        Item e2 = new Item (2);
        Item e3 = new Item (3);
        List<Item > items = new ArrayList<>();
        items .add(e1);
        items .add(e2);
        items .add(e3);
        return items ;
    }
}
```

3.实现消息的订阅

```
public class TestSubscriber implements Subscriber<Item> {
    private Subscription subscription;
    private int counter = 0;
    @Override
    public void onSubscribe(Subscription subscription) {
        System.out.println("Subscribed");
        this.subscription = subscription;
        this.subscription.request(1);
        System.out.println("onSubscribe requested 1 item");
    }
    @Override
    public void onNext(Item item) {
        System.out.println("Processing Item "+item);
        counter++;
        this.subscription.request(1);
    }
    @Override
    public void onError(Throwable e) {
        System.out.println("Some error happened");
    }
    @Override
    public void onComplete() {
        System.out.println("All Processing Done");
    }
    public int getCounter() {
        return counter;
    }
}
```

在步骤3中，Subscription变量保持消费者对生产者的引用，通过onNext（）方法进行请求处理；Count变量记录请求个数；在onSubscribe方法中调用订阅请求来开始处理；在onError方法和onComplete方法中调用发生错误和完成时执行的业务逻辑。

4.使用主程序测试完成逻辑

```
public class TestReactiveApp {
    public static void main(String args[]) throws InterruptedException {
        SubmissionPublisher<Item> publisher = new SubmissionPublisher<>();
        TestSubscriber subs = new TestSubscriber ();
        publisher.subscribe(subs);
        List<Item> items = ItemHelper.getItems();
        System.out.println("Publishing Items to Subscriber");
        items .stream().forEach(i -> publisher.submit(i));
        while (items.size() != subs.getCounter()) {
            Thread.sleep(10);
        }
        publisher.close();
        System.out.println("Exiting the TestReactiveApp");
    }
}
```

在步骤4中，首先使用SubmissionPublisher、TestSubscriber创建发布者和订阅者。通过publisher.subscribe（subs）建立发布者与订阅者之间的关联关系；然后发布者通过submit方法发送消息 给订阅者，这个过程是异步执行的；在主线程的while循环中判断Item的size和消费累计的size；当Item全部消费完成时，退出主线程的While循环；最后关闭发布者以免任何内存泄漏。

下面是程序的输出结果：

```
Subscribed
Publishing Items to Subscriber
onSubscribe requested 1 item
Processing Item[id=1]
Processing Item[id=2]
Processing Item[id=3]
Exiting the TestReactiveApp
All Processing Done
```

## 10.2.3 RxJava响应式框架

RxJava基于ReactiveX（Reactive Extensions的缩写）库和框架，使用观察者模式、迭代器模式及函数式编程，提供了异步数据流处理、非阻塞背压等特性。

**Reactive Extensions**

这个概念最早出现在微软的.NET社区中，目前越来越多语言实现了自己的响应式扩展，如Java、Javascript、Ruby等。

Reactive Extensions是响应式编程的一种实现，是解决异步事件流的一种方案。通俗地讲，就是利用它可以很好地控制事件流的异步操作，将事件的发生和对事件的响应解耦，让开发者不再关心复杂的线程处理、锁等并发相关问题。

**RxJava的接入实例**

RxJava 2.x实现了响应式流规范。它是Netflix开发的一个响应式编程框架。下面是RxJava的典型开发代码：

```
Observable.create(new ObservableOnSubscribe<String>() {
@Override
public void subscribe(ObservableEmitter<String> emitter) throws Exception {
      emitter.onNext(1);
      emitter.onNext(2);
      emitter.onNext(3);
      emitter.onComplete();
        }
        }).map(value -> value * 10
          }).subscribeOn(Schedulers.io())
      .observeOn(Schedulers.mainThread())
      .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {
                mDisposable=d;
                Log.e(TAG, "onSubscribe");
             }
            @Override
            public void onNext(String value) {
                If(value.equal( “20” )) {
                  mDisposable.disponse();
                }
                Log.e(TAG, "onNext:"+value);
            }
            @Override
            public void onError(Throwable e) {
                Log.e(TAG, "onError="+e.getMessage());
              }
            @Override
            public void onComplete() {
                Log.e(TAG, "onComplete()");
            }
    });
```

**Observable**

Observable可以理解为数据的发射器，对应Java Flow的发布者（Publisher）组件，通过create方法生成Observer对象。它会执行相关业务逻辑并通过emit方法发射数据，传入的参数是ObservableOnSubscribe对象，使用泛型T作为操作对象的类型。你可以重写subscribe方法，里面是具体的数据源计划，前面的例子中是发射三个数字：1、2、3。ObservableEmitter是发射器的意思，有三种发射数据的方法：void onNext（T value）、void onError（Throwable error）、void onComplete（）。onNext方法可以无限调用，观察者（Observer）可以接收到所有发布者（Publisher）发布的数据库，onError和onComplete是互斥的。

**Observer**

Observer是数据的观察者，对应Java Flow的订阅者（Subscriber）组件，通过new方法创建并重写内部方法，onNext、onError、onComplete都是与被观察者发射的方法一一对应的。在本例中，订阅者的onNext方法处理消费数据逻辑，当收到的数据等于20时，将取消订阅，此时数据的发布者就不再向观察者推送数据。通过dispose方法可以取消Observer和Observable之前的订阅关系。

**Scheduler**

RxJava支持异步通信的特性是通过Schedulers组件实现的，Scheduler的中文意思是调度器。在RxJava中，可以通过Scheduler来控制调度线程，从Scheduler的源码可以发现它本质上是操纵

Runnable对象，支持用立即、延时、周期形式来调度工作线程。RxJava 2.x中内置了多种Scheduler实现，适用于不同场景。这些Scheduler可以在代码中直接使用，屏蔽了开发者对线程调用的管理和控制。在前面的例子中我们使用了Schedulers.io（）作为线程调度策略，下表总结的是Schedulers不同的线程调度策略。

| 方法 | 说明 |
| --- | --- |
| Schedulers.computation() | 适用于计算密集型任务 |
| Schedulers.io() | 适用于I/O密集型任务 |
| Schedulers.trampoline() | 在某个调用Schedule的线程执行 |
| Schedulers.newThread() | 每个Worker都对应一个新线程 |
| Schedulers.single() | 所有Worker都使用同一个线程执行任务 |
| Schedulers.from(Executor) | 使用Executor作为任务执行的线程 |

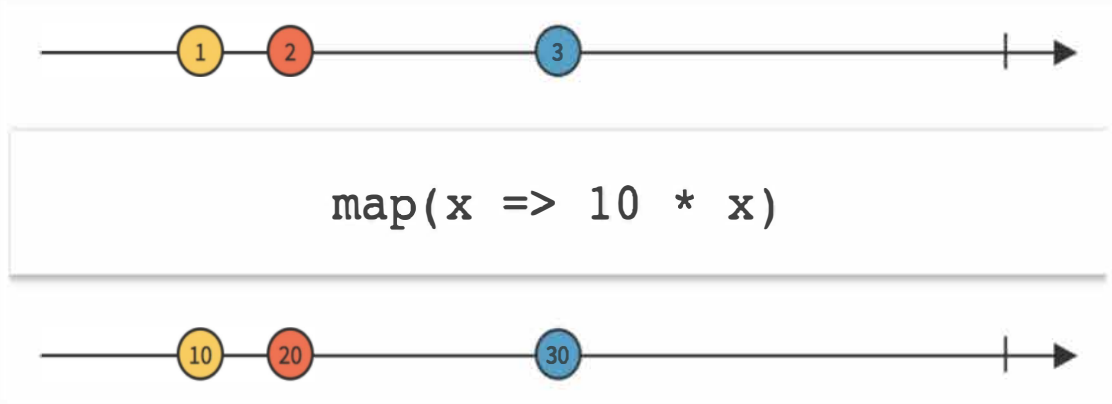
**Operator**

RxJava在处理事件的流转过程中，提供了丰富的操作符，用来改变事件流中的数据。以Map操作符为例，Map的作用是将发射的事件进行Map函数定义的数据转换，再将转换后的事件发射给Observer。转换过程如下图所示。

（1）通过Emitter发射了1、2、3三个数字。

（2）中间通过Map进行转换，转换后事件变成10、20、30。

（3）最后将转换后的事件发射给Observer。




RxJava2-Android-Samples（GitHub开源项目）的Readme.md中总结了RxJava用到的所有操作符，篇幅所限，其他操作符可以从Reactive官方地址获得详解。RxJava的主要操作符如下表所示。

| RxJava的主要操作符 | | | |
| --- | --- | --- | --- |
| Map | Zip | Reduce | Filter |
| FlatMap | Take | Skip | Buffer |
| Concat | Replay | Merge | |

## 10.2.4 Reactor响应式框架

Reactor是Pivotal基于Reactive Streams规范实现的响应式框架。作为Spring的兄弟项目，它进一步扩展了基本的Reactive Streams Publisher及Flux和Mono API等组件，主要使用依赖的组件是Reactor Core模块。

Reactor项目已在GitHub中开源（可使用Reactor关键字搜索），主要包含Reactor Core和Reactor Netty两部分。Reactor Core实现了反应式编程的核心功能，Reactor Netty则是Spring WebFlux等技术的基础。

**Reactor的接入实例**

1.使用Reactor进行响应式编程，加载对应的Maven依赖

```
<dependencies>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
  </dependency>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

2.使用Reactor进行响应式编程的Demo

```
public static void main(String[] args) {
    Logger("运行...");
    Flux.just("1", "2", "3") //发布 1 -> 2-> 3
            .map(value -> "+" + value) //"1" -> "+" 转换
            .subscribe(new Subscriber<String>() {
                private Subscription subscription;
                private int count = 0;
                @Override
                public void onSubscribe(Subscription s) {
                    subscription = s;
                    subscription.request(1);
                }
                @Override
                public void onNext(String s) {
                    if(count==2){
                        throw new RuntimeException("运行时异常！");
                    }
                    println(s);
                    count++;
                    subscription.request(1);
                }
                @Override
                public void onError(Throwable t) {
                    println(t.getMessage());
                }
                @Override
                public void onComplete() {
                    println("完成！");
```

```
                }
            });
    }
    public static void println(String s)
    {
        System.out.println(s);
    }
     private static void  Logger(Object object) {
         String threadName = Thread.currentThread().getName();
        System.out.println("[当前线程名称：" + threadName + "] " + object);
     }
}
```

3.执行上述程序得到如下结果

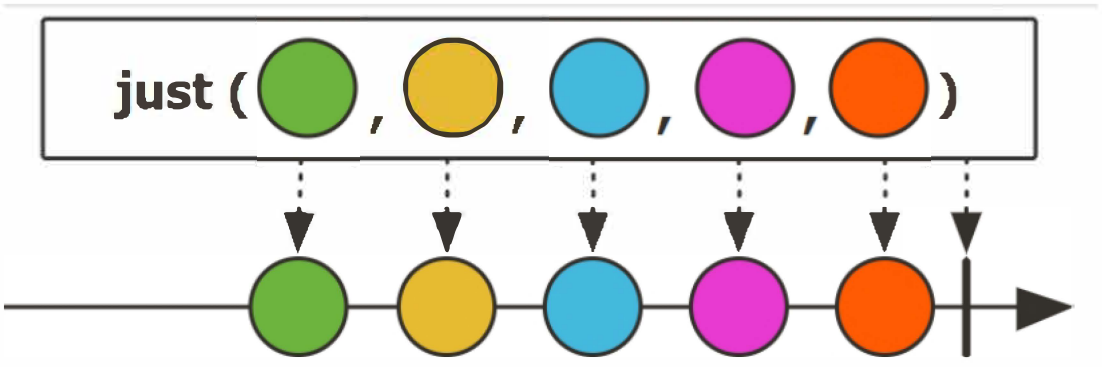
```
[当前线程名称：main] 运行...
[当前线程名称：main] +1
[当前线程名称：main] +2
Exception in thread "main" java.lang.RuntimeException: 运行时异常!
```

在Reactor项目中，主要有与RxJava类似的发布者、订阅者、操作符等关键API和语法概念，下面结合代码实例讲解主要用到的模块。

**Reactor的核心模块**

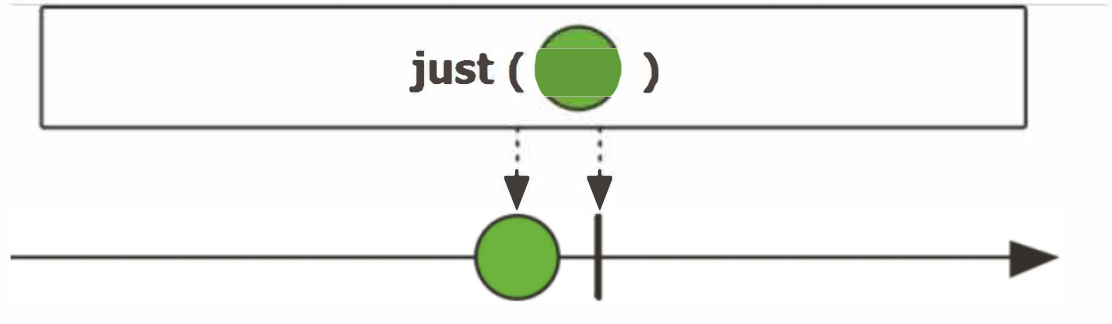
- Flux

Flux是Reactor中数据发布者的重要抽象类。从源码中可以发现，Flux实现了Reactive Streams JVM API Publisher。Flux定义了0～ N 的非阻塞序列，类比非阻塞Stream，在Reactor中充当数据发布者的角色。在上述实例中，Flux通过just方法发布数据流。just方法是Flux常见的创建Stream的方法，此外，还可以通过create、generate、from等方法创建Flux数据流。上面例子中使用最简单的just方法完成了三个数字的构造和声明发布，如下图所示。




- Mono

Mono和Flux类似。从源码中可以发现，Mono同样实现了Reactive Streams JVM API Publisher，实现了0～1的非阻塞结果，如下图所示。

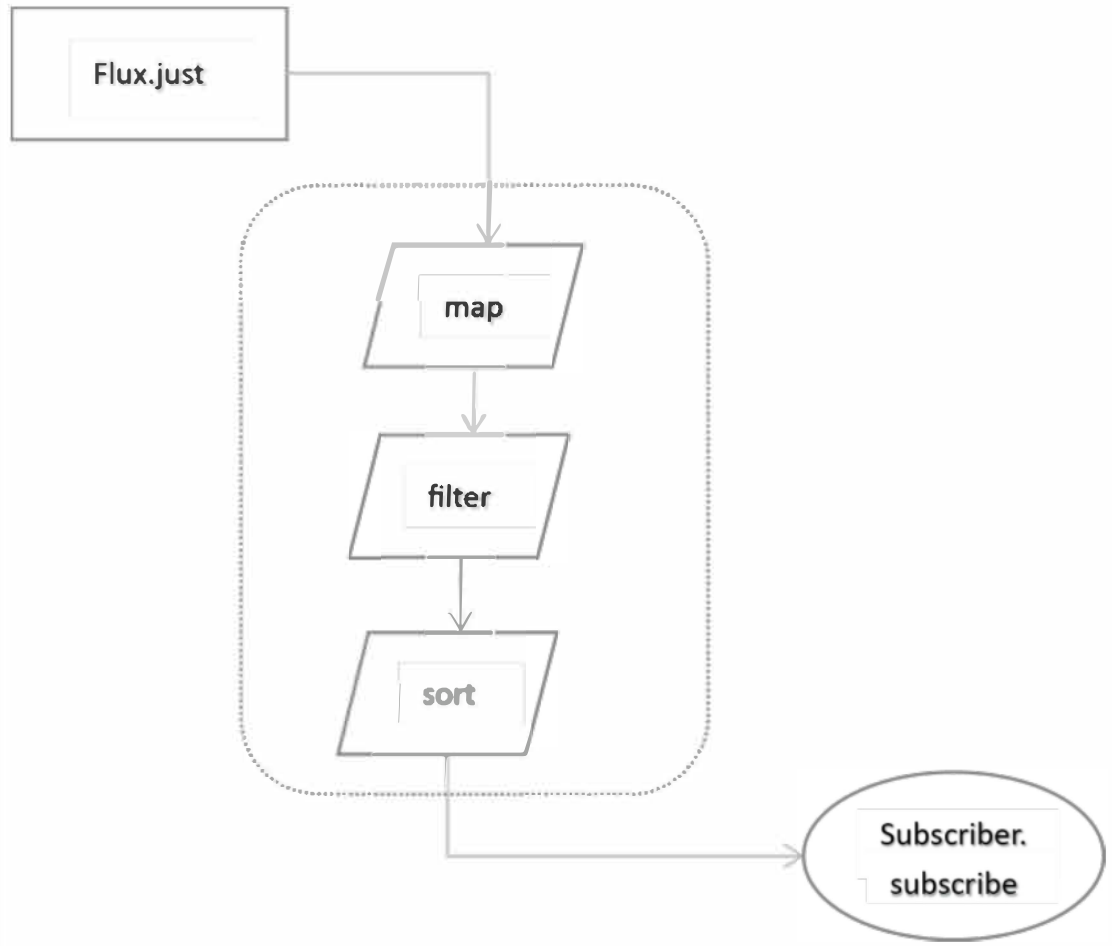



● **Subscriber**

订阅者通过订阅操作，可以处理数据的请求，在订阅方法中需要重写onSubscribe、onNext、onError、onComplete方法来实现数据流的消费。Flux调用subscribe方法后会触发数据的发送，订阅者接收到数据后会触发onSubscribe方法。onSubscribe表示订阅动作的方式，准备发送给真正的消息接收者，然后执行subscription.request方法发送请求数据。代码例子中request（1）表示只发送一条数据，也可以使用subscription.cancel取消上游数据的传输。然后执行onNext方法进行消息的响应处理，在onNext方法中执行request方法可以把数据交给subscription链，循环处理所有数据。

● **Operator**

在Reactor项目中，一个Operator会给一个发布者（Publisher）添加某种行为，并返回一个新的Publisher实例。还可以对返回的Publisher再添加Operator连成一个链条。原始数据沿着链条从第一个Publisher开始向下流动，链条中的每个节点都会以某种方式去转换流入的数据。链条的终点是一个订阅者（Subscriber），Subscriber以某种方式消费这些数据，流程图如下图所示。




下面是对Reactor项目中Operator的总结分类，大致可以分为如下几类。

- 集合Operator：提供集合运算，如map、filter、sort、group、reduce等，和Java 8 Stream的中间操作具有相同的效果。
- 异常处理Operator：提供异常处理机制，如retry、onErrorReturn等。
- 回调Operator：提供Publisher状态转换时的回调，如doOnCancel、doOnRequest等。
- 行为Operator：修改Publisher的默认行为，为其添加更多功能，如buffer、defaultIfEmpty、onBackpressureXXX等。
- 调试Operator：添加调试信息，如log、elapsed等。

## 10.2.5 Vert.X响应式编程

Vert.X是基于JVM构建的一个Reactive工具箱。同时，Vert.X和Spring类似，也有一套微服务开发生态。从开发者的角度来看，Vert.X就是一些库包，提供了HTTP客户端和服务器、消息服务、TCP和UDP底层协议等模块。你可以使用这些模块来构建自己的应用，也可以通过向Vert.X Core（Vert.X的基础组件）中增加任意模块来构建自己的系统。

**Vert.X的主要功能**

- Web开发，Vert.X封装了Web开发常用的组件，支持路由、Session管理、模板等。

- TCP/UDP开发，Vert.X底层基于Netty，提供了丰富的I/O类库，支持多种网络应用开发，不需要处理底层细节（如拆包和粘包），注重业务代码编写。
- 提供对WebSocket的支持，可以做网络聊天室、动态推送等。
- Event Bus（事件总线）是Vert.X的神经系统，通过Event Bus可以实现分布式消息、远程方法调用等。正是因为Event Bus的存在，Vert.X才可以更加便捷地开发微服务应用。
- 支持主流的数据和消息的访问，如Redis、MongoDB、RabbitMQ、Kafka等。
- 支持分布式锁、分布式计数器、分布式Map。

**Vert.X的特性**

- 异步非阻塞：Vert.X就像是跑在JVM上的Node.js（使用事件驱动、非阻塞式I/O模型的JavaScript运行环境），所以Vert.X的第一个优势就是它实现了一个异步的非阻塞框架。
- Vert.X支持多编程语言，在Vert.X上，可以使用JavaScript、Java、Scala、Ruby等语言。
- 不依赖中间件：Vert.X的底层依赖Netty，因此在使用Vert.X构建Web项目时，不依赖中间件。像Node一样，可以直接创建一个HttpServer，相对会更灵活一些，安全性也会更高一些。
- 完善的生态：Vert.X提供数据库操作、Redis操作、Web客户端操作等丰富的组件功能。

**Vert.X的接入实例**

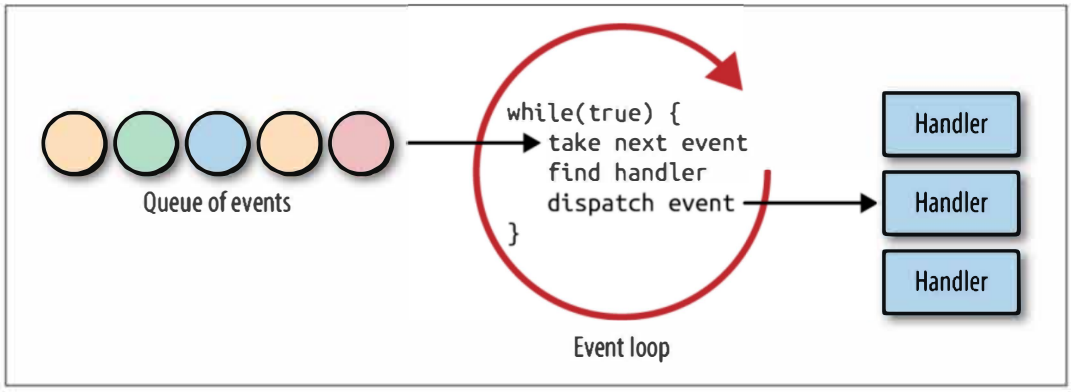
1.加载对应的Maven依赖

```
<dependency>
   <groupId>io.vertx</groupId>
   <artifactId>vertx-web</artifactId>
   <version>3.4.2</version>
</dependency>
```

2.Vert.X提供了一个创建HTTP服务器的简单方法，该服务器会在每次接收到HTTP请求时返回一个“Hello”的response

```
vertx.createHttpServer() .requestHandler(request -> {
    request.response().end("Hello");
}) .listen(8080);
```

在这个例子里，我们创建了一个requestHandler来接收HTTP请求事件，并且返回响应。在Vert.X中，所有API都不会阻塞调用线程，如果不能立即响应结果，Handler会在事件准备好后处理，通过异步操作回调Handler方法触发执行。这种非阻塞的开发模型，可以使用较少的线程处理高并发场景。下面是Vert.X中EventLoop的工作模型图。




Verticle是Vert.X中的重要组件，可以理解成Java中的Servlet、POJO Bean或Akka中的Actor。一个组件可以有多个实例，Verticle实例之间的通信通过Event Bus实现。

ProducerVerticle负责监听8080端口，接收前端请求，它可以通过Event Bus发送一个事件，该事件将被传递给多个该事件的订阅者，代码如下。

```
public class ProducerVerticle extends AbstractVerticle {
    @Override
    public void start() throws Exception {
        HttpServer server = vertx.createHttpServer();
        Router router = Router.router(vertx);
        router.route().handler(BodyHandler.create());
        router.get("/book/:id").handler(this::getBook);
        server.requestHandler(router);
        server.listen(8080);
    }
    private void getBook(RoutingContext ctx) {
        int id = Integer.parseInt(ctx.request().getParam("id"));
        JsonObject json = new JsonObject().put("id", id);
        vertx.eventBus().request("com.msa.vertx.database", json, r -> {
            if (r.succeeded()){
                ctx.response().putHeader("Content-Type", "application/json; charset=utf-8")
                .end(msg);
            } else {
                r.cause().printStackTrace();
                ctx.fail(r.cause());
            }
        });
    }
}
```

ConsumeVerticle负责消费Event Bus的数据并返回响应，代码如下。

```
public class ConsumeVerticle extends AbstractVerticle {
    @Override
    public void start() throws Exception {
        super.start();
        MessageConsumer<JsonObject> consumer =
            vertx.eventBus().consumer("com.msa.vertx.database");
        consumer.handler(msg -> {
            //省略从数据库加载逻辑。。
            System.out.println(msg.body());
            JsonObject json = new JsonObject();
            json.put("id", 1);
            json.put("name", "微服务架构进阶");
            msg.reply(json);
        });
    }
}
```

MainApp是启动类，在main方法中发布两个Verticle，下面代码是启动主流程的方法。

```
public class MainApp{
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        vertx.deployVerticle(new ConsumeVerticle());
        vertx.deployVerticle(new ProducerVerticle());
    }
}
```

浏览器调用接口http://127.0.0.1：8080/book/1，出现下面结果则表示正确。

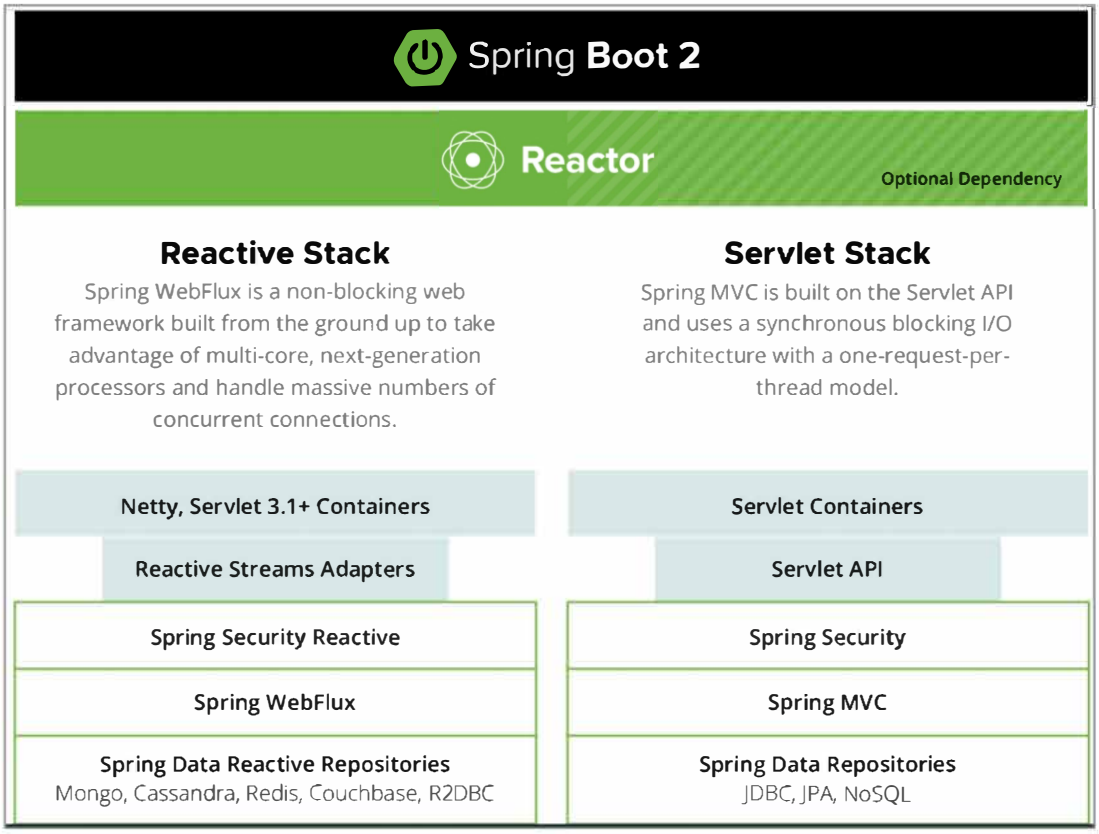
```
{
    “id”:”1”,
    “name”：“微服务架构进阶”
}
```

Verticle具有以下几个特点。

- 每个Verticle都占用一个EventLoop线程，且只对应一个EventLoop。
- 每个Verticle中创建的HttpServer、EventBus等资源都会在回收Verticle时被同步回收。
- 在多个Verticle中创建同样端口的HttpServer，会变成两个EventLoop线程，处理同一个HttpServer的连接，可以利用Verticle的这一特性来提升并发处理性能。

### 10.2.6 Spring Boot 2响应式编程

Spring Boot 2.x在Spring Boot 1.x基础上，基于Spring 5实现了响应式编程框架。从Spring MVC注解驱动的时代开始，Spring官方有意识地去Servlet化。不过在Spring MVC时代，Spring仍然摆脱不了对Servlet容器的依赖，然而借助响应式编程（Reactive Programming）的势头，Spring加速了这一时代的到来。WebFlux将Servlet容器从必须项变为可选项，并且默认采用Netty Web Server作为HTTP容器的处理引擎，形成Spring全新的技术体系，包括数据存储等技术栈。Spring Boot 2官方提供的基于Reactor与Servlet容器生态和技术栈的对比如下图所示。




对比发现，Spring Boot 2.x与Spring Boot 1.x在技术栈上存在巨大差异。Spring Boot 2.x最显著的变化就是采用了响应式的技术体系。底层的Reactive核心组件、响应式WebFlux框架、响应式数据存储、响应式安全、响应式Web服务引擎组成了Spring响应式技术体系。下面列举了Spring Boot 2中支持响应式编程的部分模块。

### Spring Core

Spring Core是Spring的核心模块。Spring Framework 5基于ProjectReactor和RxJava反应式项目及响应式编程规范实现了对响应式编程的支持。在Spring Core中通过引入ReactiveAdapter实现了Object和Publisher<T>的相互转换，代码如下：

```
class ReactiveAdapter {
...省略
    <T> Publisher<T> toPublisher(@Nullable Object source) { ... }
    Object fromPublisher(Publisher<?> publisher) { ... }
}
```

使用者可以通过继承ReactiveAdapter实现定制化的数据类型转换。ReactiveAdapterRegistry可以作为对象池来保持ReactiveAdapter实例并提供相应的数据访问方式。

### 响应式I/O

Spring Core提供了对I/O的响应式编程支持。Spring Core首先引入了一个字节缓存抽象接口DataBuffer，提供了一个DataBufferUtils工具类，可以实现以Reactive方式对I/O进行访问和交互。从下面的示例代码可以看到，DataBufferUtils返回了一个Flux对象，这样就可以使用Reactor相关接口读取test.txt文件，实现背压的响应式特性。

```
Flux<DataBuffer> reactiveHamlet = DataBufferUtils.read(
    new DefaultResourceLoader().getResource("test.txt"),
    new DefaultDataBufferFactory(),
    1024);
```

同时，Spring Core通过下面接口实现了基于响应式流的编解码实现类，这样可以方便DataBuffer实例与对象的相互转化，代码如下：

```
interface Encoder<T> {
    ...
    Flux<DataBuffer> encode(Publisher<? extends T> inputStream,  ...);
}
interface Decoder<T> {
    ...
    Flux<T> decode(Publisher<DataBuffer> inputStream,  ...);
    Mono<T> decodeToMono(Publisher<DataBuffer> inputStream,  ...);
}
```

**Spring WebFlux构建响应式Web服务**

在Web服务方面，Spring 2.x提供了WebFlux框架，基于Flux和Mono对象实现响应式非阻塞Web服务。同时提供了一个响应式的HTTP WebClient，它可以通过函数式的方式异步非阻塞地发起HTTP请求并处理响应。Spring WebFlux也提供了响应式的WebSocketClient。下一节我们会详细讲解Spring的WebFlux框架。

**数据层支持响应式**

开发基于响应式流的应用，就像搭建数据流的管道，使异步数据能够顺畅流过每个环节。大多数系统免不了要与数据库交互，所以我们也需要响应式的持久层API和支持异步的数据库驱动。在消息的处理过程中，如果数据管道在任何一个环节发生阻塞，都有可能造成整体吞吐量的下降。

各个数据库都开始陆续推出异步驱动的技术支持，目前可以支持响应式数据访问的数据库有MongoDB、Redis、Apache Cassandra和CouchDB。

**相关生态的响应式支持**

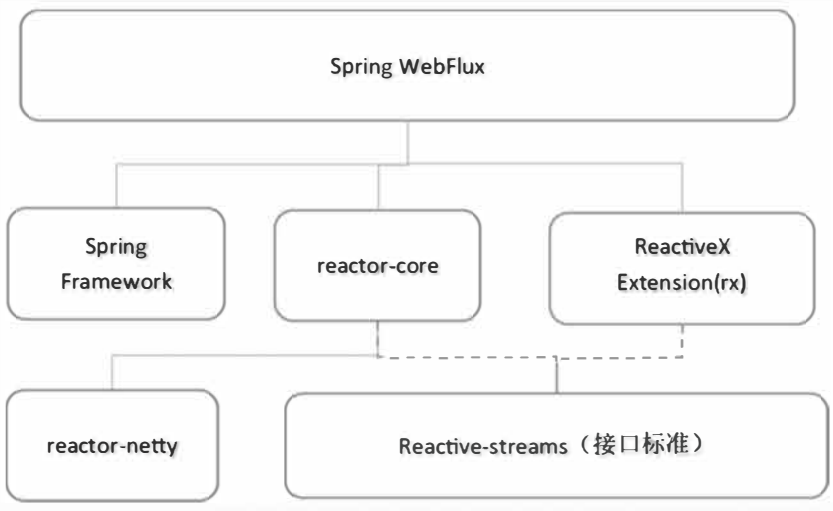
- Spring 5实现了对Spring Security的响应式支持。
- Spring Cloud基于WebFlux框架实现了Spring Cloud Gateway微服务网关。
- Spring Test实现了响应式的支持类WebTestClient。
- 在监控领域，Sleuth也提供对响应式WebFlux的追踪支持。

## 10.3 Spring WebFlux框架

Spring WebFlux是Spring 5发布的响应式Web框架，从Spring Boot 2.x开始，默认采用Netty作为非阻塞I/O的Web服务器。

### 10.3.1 Spring WebFlux概述

Spring WebFlux基于Reactor框架，同时支持RxJava类库，构建响应式编程框架。查看WebFlux的Maven依赖，可以发现它依赖的项目工程包有Reactor、Spring、ReactiveX、RxJava等模块，使用WebFlux需要单独引用它的依赖包，WebFlux主要的包依赖关系如下图所示。




**WebFlux的主要特征**

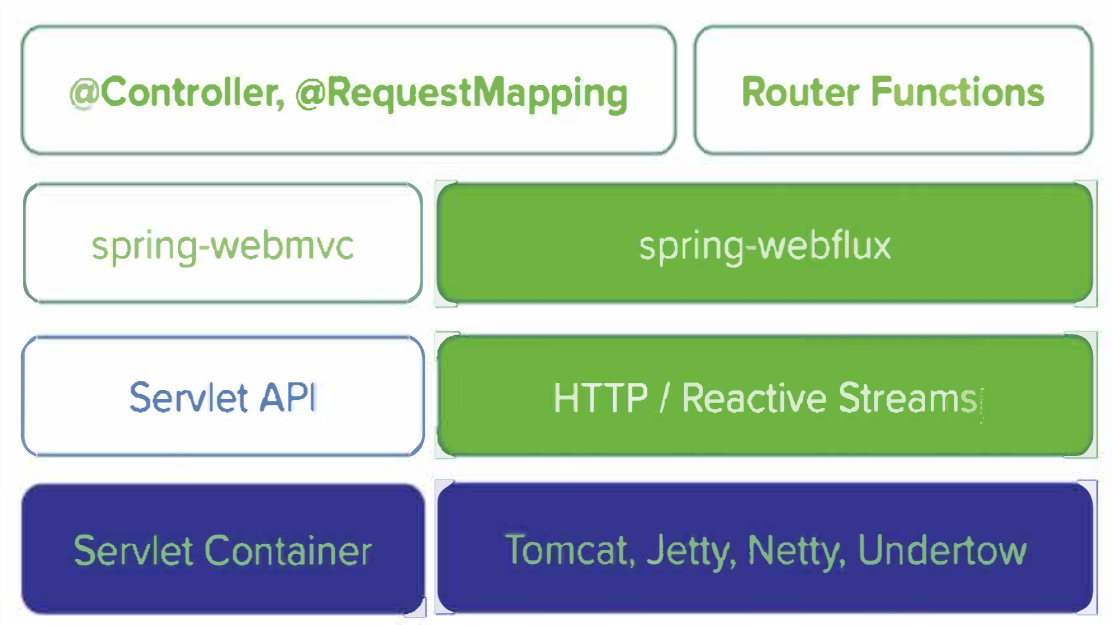
- 采用Reactor响应式编程框架，同时提供对RxJava类库的支持。
- Spring WebFlux基于响应式流，可以建立异步、非阻塞、事件驱动的服务。
- Spring WebFlux和Reactor底层默认使用Netty作为Web服务器，使用线程收敛式方式处理I/O业务逻辑，同时支持异步Servlet 3.1容器（Tomcat、Jetty等）。
- Spring WebFlux同时支持响应式的WebSocket服务开发。
- 支持响应式HTTP客户端，可以用函数式方式异步非阻塞地发送HTTP请求。

**WebFlux的主要模块**

WebFlux的应用方式可以使用基于Spring Boot提供的开发模板，直接访问Spring Initializ网站，创建一个Maven或者Gradle项目，需要添加的依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

在选择Spring Boot版本号时，需要选择2.0.0M2以后的版本才能正确加载WebFlux依赖包，下图是官方提供的Spring WebFlux与Spring MVC的架构对比。




从图中我们可以基本了解WebFlux的主要模块。

- 底层是Web HTTP服务引擎，Spring MVC框架基于传统的Servlet容器，WebFlux实现了Servlet 3.1+规范的容器引擎，Servlet 3.1规范中新增了对异步处理的支持，同时默认采用Netty的非Servlet引擎，采用I/O多路复用的异步非阻塞HTTP引擎。
- Spring MVC使用传统的Servlet API应用方式，而WebFlux基于响应式流框架，支持采用背压（Backpressure）方式的异步数据处理流标准。WebFlux默认继承了Reactor项目实现。
- Spring WebFlux的核心组件完成协调上下文及提供响应式编程支持的工作。
- Spring MVC主要使用注解的方式完成HTTP请求到方法的映射，WebFlux支持注解和函数式两种调用方式，通过函数式风格的API可以创建路由、Handler和Filter等服务组件。

## 10.3.2 WebFlux服务器开发

下面我们分别演示WebFlux提供的注解控制器模式和函数端点模式。介绍使用这两种编程模型实现的代码示例，以及它们不同的语法和差异。

**注解控制器模式**

注解控制器模式与Spring MVC一致，都基于与Spring Web模块相同的注释。Spring MVC和WebFlux控制器都支持反应式（Reactor、RxJava）数据返回类型，因此不容易将它们分开。一个显著的区别是，WebFlux还支持注解@RequestBody来处理事件响应。

首先，我们实现一个获取用户数据的Service，代码如下。

```
@Service
class UserService {
  private final Map<String,  User> data = new ConcurrentHashMap<>();
  Flux<User> list() {
      return Flux.fromIterable(this.data.values());
  }
  Flux<User> getById(final Flux<String> ids) {
    return ids.flatMap(id -> Mono.justOrEmpty(this.data.get(id)));
  }
}
```

然后，定义UserController类，它是具体的Spring MVC控制器，使用UserService获取数据，代码如下。

```
@RestController
@RequestMapping("/user")
public class UserController {
   private final UserService userService;
    @Autowired
    public UserController(final UserService userService) {
      this.userService = userService;
   }
   @GetMapping("")
   public Flux<User> list() {
      return this.userService.list();
   }
  @GetMapping("/{id}")
  public Mono<User>getById(@PathVariable("id") final String id) {
    return this.userService.getById(id);
  }
}
```

**函数端点模式**

函数端点模式是基于Lambda的轻量级功能编程模型。可以将其视为小型库或应用程序，是可用于路由和处理请求的一组实用程序。它与注解控制器模式的巨大差异在于，应用程序负责从开始到结束的请求处理，并通过注解声明完成请求回调处理。

下面代码实现了TaskHandler（实现对UserService的访问）。

```
@Component
public class TaskHandler {
    @Autowired
    private UserService userService;
    public Mono<ServerResponse> getUserById(ServerRequest serverRequest) {
        return ServerResponse.status(HttpStatus.OK)
                .body(Mono.just(UserService.getById().get(Integer.valueOf(serve
```

```
                    rRequest.pathVariable("userId")))),  User.class);
    }
    public Mono<ServerResponse> getAll(ServerRequest serverRequest) {
        Flux<User> userFlux =
                Flux.fromStream(userService.list()
                .entrySet().stream().map(Map.Entry::getValue));
        return ServerResponse.ok().body(userFlux,  User.class);
    }
}
```

将URL路由和Handler函数绑定，代码如下。

```
@Configuration
public class WebFluxRoutingConfiguration {
    @Autowired
    private TastHandler taskHandler;
    @Bean
    public RouterFunction<ServerResponse> routerFunction() {
        return route(GET("/webflux/user/{userId}"),  taskHandler::getUserById)
                .andRoute(GET("/webflux/users"), taskHandler::getAll);
    }
}
```

**说明：** WebFlux通过配置函数路由（RouterFunction）的方式来实现请求的映射，处理TaskHandler的方法的返回类型是Mono<ServerResponse>。

下面我们看一下@FunctionInterface查看route的实现源码：

```
public static <T extends ServerResponse> RouterFunction<T> route(
    RequestPredicate predicate, HandlerFunction<T> handlerFunction) {
        return new DefaultRouterFunction<>(predicate,  handlerFunction);
    }
}
```

从源码中，我们发现RouterFunction返回一个<T extends ServerResponse>对象。在DefaultRouterFunction类中可以看到，在该类的route方法中可以判断请求的参数，如果值为空，则返回Empty，否则返回Mono<HandlerFunction<T>>的一个函数式接口，而这个函数就是Config中配置路由断言时指定的HandlerFunction。源码如下：

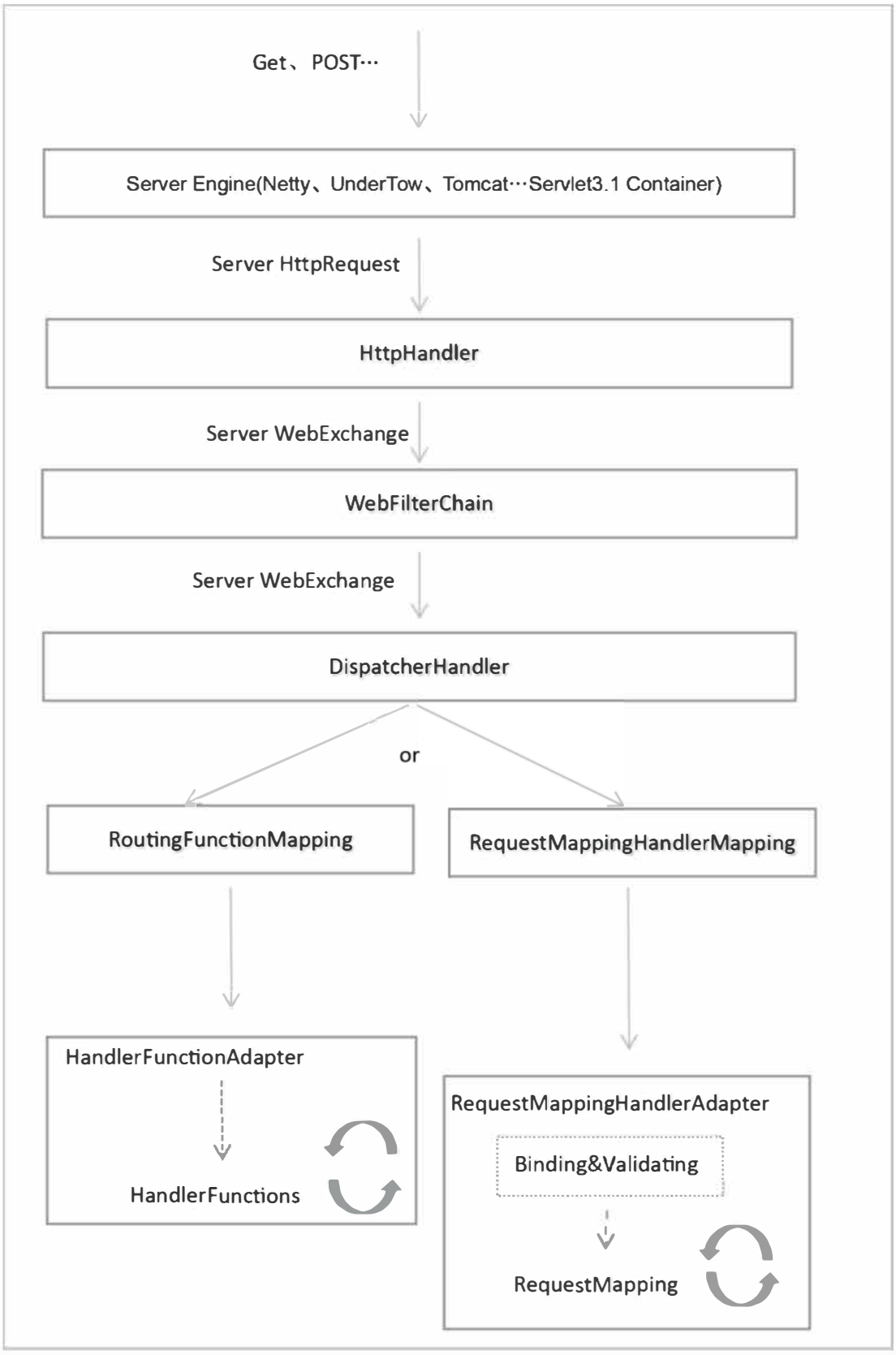
```
private static final class DefaultRouterFunction<T extends ServerResponse>
    extends AbstractRouterFunction<T> {
    private final RequestPredicate predicate;
```

```
    private final HandlerFunction<T> handlerFunction;
    public DefaultRouterFunction(RequestPredicate predicate,
        HandlerFunction<T> handlerFunction) {
      Assert.notNull(predicate, "Predicate must not be null");
      Assert.notNull(handlerFunction, "HandlerFunction must not be null");
      this.predicate = predicate;
      this.handlerFunction = handlerFunction;
    }
    @Override
    public Mono<HandlerFunction<T>> route(ServerRequest request) {
      if (this.predicate.test(request)) {
        if (logger.isDebugEnabled()) {
          logger.debug(String.format("Predicate \"%s\" matches
              against \"%s\"", this.predicate, request));
        }
        return Mono.just(this.handlerFunction);
      }
      else { return Mono.empty(); }
    }
    @Override
    public void accept(Visitor visitor) {
      visitor.route(this.predicate, this.handlerFunction);
    }
  }
```

总之，由上面的源码分析可知，WebFlux底层虽然和传统的Spring Web工作机制完全不同，但是WebFlux依然支持基于注解驱动的编程模型，区别在于WebFlux的并发模型和阻塞特性。函数端点模式是WebFlux通过配置函数路由的方式，实现请求到业务处理函数的映射。对于HTTP请求是如何从Web引擎映射到具体的实现方法的，下一节我们会继续介绍WebFlux的逻辑处理架构和HTTP请求的路由映射过程。

**Spring WebFlux源码架构解析**

与Spring MVC使用DispatcherServlet作为Servlet容器承上启下的重要管理组件类似，在Spring WebFlux框架中，DispatcherHandler有着异曲同工的作用。下面我们根据WebFlux源码讲解它的主要接口和模块的相互关系，WebFlux的工作流程如下图所示。




下面我们按照流程图的执行序列由上至下分别加以解释。

HTTP请求进入WebFlux处理引擎（Server Engine），这里默认使用Netty作为HTTP容器引擎。如果你想修改Servlet容器的服务引擎，则需要在pom.xml文件中添加相应的容器依赖Starter包，这里每个服务引擎都会有自己独立的响应适配器（Adapter）映射HTTP请求或响应到Server HttpRequest和Server HttpResponse组件。

HttpHandler请求处理阶段主要将输入的请求（Server HttpRequest、Server HttpResponse，包括用户Session及相关信息）转换为Server WebExchange。

在WebFilterChain阶段，WebFlux会遍历之前注册到Spring容器的WebFilter对象，WebFilterChain负责执行所有WebFilter中的filter方法。对于WebFlux的Filter功能，一种方式是使用WebFilter在Spring MVC中的Filter接口，以接口的形式放回Mono<Void>；另一种方式是使用HandlerFilterFunction接口实现函数式的过滤调用。

如果所有WebFilter都通过执行并放行请求继续执行，那么WebFilterChain将调用WebHandler接口。

DispatcherHandler实现了WebHandler接口，同时DispatcherHandler是WebFlux实现HTTP消息从框架层映射到对应业务逻辑方法的关键实现类。下面是WebFlux处理消息分发的关键源码：

```
public class DispatcherHandler implements WebHandler, ApplicationContextAware {
    @Override
    public Mono<Void> handle(ServerWebExchange exchange) {
      if (this.handlerMappings == null) {
        return createNotFoundError();
      }
      return Flux.fromIterable(this.handlerMappings)
      .concatMap(mapping -> mapping.getHandler(exchange)).next()
      .switchIfEmpty(createNotFoundError())
      .flatMap(handler -> invokeHandler(exchange, handler))
      .flatMap(result -> handleResult(exchange, result));
    }
}
```

从上面的代码可知，DispatcherHandler的主要流程是遍历HandlerMapping数据结构，并封装成数据流类Flux。它会触发对应的handler方法，执行相应的业务代码逻辑，而HandlerMapping在配置阶段会根据@Controller、@RequestMapping、@GetMapping、@PostMapping注解注册对应的业务方法到HandlerMapping接口，这也是WebFlux兼容注解方式的原因。这些配置路由最终都会通过getHandler（）方法找到对应的处理类。

最后是RequestMappingHandlerAdapter处理阶段，这个映射关系也适用于HandlerAdapter，我们将响应结果转变为数据流返回给handlerResult方法，并将结果转换成数据流序列返回。

## 10.3.3 WebClient开发

WebClient是从Spring WebFlux 5.0开始提供的一个非阻塞的、基于响应式编程范式的HTTP请求客户端工具。WebClient与传统的RestTemplate的主要区别在于基于函数式、响应式和流式的API，使用声明式的代码风格。同时WebClient依赖非阻塞式的编解码器来完成HTTP的请求和响应。

**WebClient的构建**

下面是构建WebClient的一个简单实例。首先通过WebClient.create方法创建一个WebClient实例，然后通过get、post等方法选择适当的客户端调用方式，uri用来指定需要请求的路径，retrieve用来发起请求并获得响应，bodyToMono（String.class）用来指定请求结果需要处理为String、并包装为Reactor的Mono对象，代码如下所示：

```
public class WebClientDemo {
    WebClient webClient = WebClient.create();
    Mono<String> mono = webClient.get().uri("https://localhost:8081").retrieve()
        .bodyToMono(String.class);
        mono.subscribe(System.out::println);
}
```

除了通过create方法构建WebClient，也可以通过WebClient.builder方法创建WebClient.Builder对象。在对Builder对象进行一些配置后调用build方法创建WebClient对象。通过注册一个客户端Filter（ExchangeFilterFunction）实现拦截和修改HttpClient的头信息，同时执行baseURL和默认Cookie的构建，代码示例如下：

```
WebClient webClient = WebClient.builder().baseUrl("http://localhost:8081")
      .filter((request， next) -> {
         ClientRequest filtered = ClientRequest.from(request)
         .header("foo"， "bar")
         .build();
       return next.exchange(filtered);
       }).defaultCookie("cookieName"， "cookieValue").build();
```

Builder自带的API构建WebClient的方法还有下面几种。

- uriBuilderFactory：定制uriBuilderFactory来构建baseURL。
- defaultHeader：每个HTTP请求默认使用的头信息。
- defaultCookie：每个HTTP请求默认使用的Cookie。
- defaultRequest：定制HTTP请求。
- filter：客户端可以构建filterBean实例来对WebClient的参数进行拦截过滤。
- exchangeStrategies：可以定制HTTP消息的发送接收策略。
- clientConnector：设置HTTP客户端。

**WebClient的响应解析**

WebClient使用retrieve（）方法作为获取HTTP响应的最简单方法。它可以接受单个对象（Mono），也可以接受数据流（Flux），同时可以判断返回的响应处理逻辑。

- 解析为Mono对象，代码示例如下。

```
WebClient client = WebClient.create("https://localhost:8090");
Mono<Person> result = client.get()
       .uri("/persons/{id}"， id).accept(MediaType.APPLICATION_JSON)
       .retrieve()
       .bodyToMono(Person.class);
```

- 解析为Flux对象，代码示例如下。

```
Flux<Quote> result = client.get()
            .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
            .retrieve()
            .bodyToFlux(Quote.class);
```

● 通过onStatus（）方法获取4xx/5xx的不同异常响应，代码示例如下。

```
Mono<Person> result = client.get()
       .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
       .retrieve()
       .onStatus(HttpStatus::is4xxClientError, response -> ...)
       .onStatus(HttpStatus::is5xxServerError, response -> ...)
       .bodyToMono(Person.class);
```

**WebClient使用exchange**

相比retrieve方法，WebClient使用exchange方法可以对HTTP响应提供更多控制，获得自己定制的或者想要的结果。可以对clientResponse对象执行flatMap操作，代码如下。

```
Mono<Person> result = client.get()
       .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
       .exchange()
       .flatMap(response -> response.bodyToMono(Person.class));
```

**WebClient提交Body**

假设WebClient需要提交一个JSON对象，如{“name”：“hello”，“id”：“123”}，需要将这个对象传递给远端服务，WebClient会使用ReactiveAdapterRegitry来处理，将Body的异步编解码过程转换为JSON对象，代码示例如下。

```
String baseUrl = "http://localhost:8081";
WebClient webClient = WebClient.create(baseUrl);
Map<String, Object> user = new HashMap<>();
user.put("name", "hello");
user.put("id", "123");
Mono<Void> mono =
webClient.post().uri("/user/add").syncBody(user)
.retrieve().bodyToMono(Void.class);
mono.block();
```

当然也可以将编码后的JSON对象直接传递给WebClient，需要在HTTP头信息中指定ContentType为application/json，也可以加上charset编码。在默认情况下，WebClient将根据请求传递的对象进行解析，处理后自动选择ContentType。代码示例如下。

```
String baseUrl = "http://localhost:8081";
WebClient webClient = WebClient.create(baseUrl);
String userJson =  "{" +
       " \"name\":\"hello\", \r\n" +
       " \"id\":\"123\"\r\n" +
       "}";
Mono<Void> mono = webClient.post().uri("/user/add")
    .contentType(MediaType.APPLICATION_JSON_UTF8)
    .syncBody(userJson).retrieve().bodyToMono(Void.class);
mono.block();
```

## 10.3.4 服务端推送事件

**SSE服务端**

服务端推送事件（Server-Sent Events，SSE）允许服务端不断地推送数据到客户端。相对于WebSocket而言，服务端推送事件只支持服务端到客户端的单向数据传递。SSE也是WebSocket的一个轻量级的替代方案，虽然功能较弱，但优势在于，SSE在已有的HTTP上可以使用简单易懂的文本格式来表示传输的数据。作为W3C的推荐规范，SSE在浏览器端的支持也比较广泛，除了IE，其他浏览器也都提供了支持。在IE上，也可以使用polyfill库来提供支持。对服务端来说，SSE是一个不断产生新数据的流，非常适合用响应式流来表示。在WebFlux中创建SSE的服务端是非常简单的，只需要返回的对象类型是Flux<ServerSentEvent>，就会自动按照SSE规范要求的格式来发送响应。

SseController是一个使用SSE的控制器。其中，randomNumbers方法表示每隔一秒产生一个随机的SSE端点。我们可以使用ServerSentEvent.Builder类来创建ServerSentEvent对象。这里我们指定了事件的名称random，以及每个事件的标识符和数据。事件的标识符是一个递增的整数，而数据则是产生的随机数。下面的代码演示了服务推送事件。

```
@RestController
@RequestMapping("/sse")
public class SseController {
    @GetMapping("/randomNumbers")
    public Flux<ServerSentEvent<Integer>> randomNumbers() {
        return Flux.interval(Duration.ofSeconds(1))
                .map(seq -> Tuples.of(seq, ThreadLocalRandom.current().nextInt()))
                .map(data -> ServerSentEvent.<Integer>builder()
                        .event("random")
                        .id(Long.toString(data.getT1()))
                        .data(data.getT2())
                        .build());
    }
}
```

在测试SSE时，我们只需要使用curl来访问即可。下面的代码给出了调用curl http://localhost：8080/sse/randomNumbers的结果。

```
id:0
event:random
data:751025203
id:1
event:random
data:-1591883873
id:2
event:random
data:-1899224227
```

**SSE客户端**

WebClient还可以用同样的方式来访问SSE服务。这里我们访问的是在之前内容中创建的产生随机数的SSE服务。使用WebClient访问SSE服务在发送请求部分与访问Rest API是相同的，区别在于对HTTP响应的处理。由于SSE服务的响应是一个消息流，我们需要使用flatMapMany把Mono<ServerResponse>转换成Flux<ServerSentEvent>对象，这是通过BodyExtractors.toFlux方法完成的。

参数new ParameterizedTypeReference<ServerSentEvent<String>> （）{}表明了响应式流中的内容是ServerSentEvent对象。由于SSE服务端会不断地发送消息，这里我们只是通过buffer方法来获取前10条消息并输出，代码如下所示。

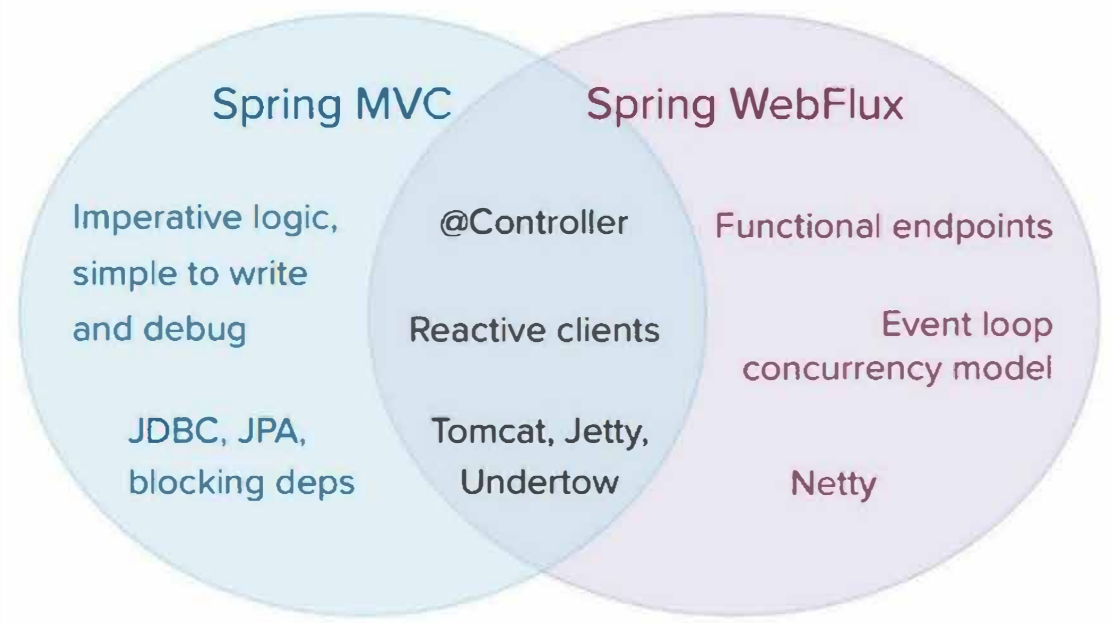
```
public class SSEClient {
public static void main(final String[] args) {
 final WebClient client = WebClient.create();
 client.get()
 .uri("http://localhost:8080/sse/randomNumbers")
  .accept(MediaType.TEXT_EVENT_STREAM)
   .exchange()
   .flatMapMany(response -> response.body(BodyExtractorstoFlux(new
        ParameterizedTypeReference<ServerSentEvent<String>>() {
               })))
    .filter(sse -> Objects.nonNull(sse.data()))
     .map(ServerSentEvent::data)
     .buffer(10)
     .doOnNext(System.out::println)
     .blockFirst();
 }}
```

## 10.3.5 Spring WebFlux的优势与局限

在传统的Java后台服务端开发中，我们使用Spring MVC框架的项目比较多，一个很自然的问题就是，对Spring MVC与Spring WebFlux技术栈的选择问题。开发者需要考虑从Spring MVC转型到Spring WebFlux框架的优势与局限。

**Spring WebFlux与Spring MVC**

下面是官方展示的Spring MVC与Spring WebFlux的框架对比图，可以看出两者在组件功能上的差异。




从上面的图中，我们可以看出两个框架在下面几方面的异同。

- 在编程模型上，Spring MVC偏向于命令式编程，其优点是简单、容易理解，并且对于开发者来说方便调试。而Spring WebFlux倾向于函数式编程模型。在调试和编程难度上相比Spring MVC，Spring WebFlux更大一些。然而如之前所说，函数式编程的优势是代码的可读性更强，更加强调不可变性，比命令式编程有更稳定的表现。
- 在线程模型上，Spring MVC主要受Servlet标准规范（3.x版本之前）的限制，所以主要使用同步式编程模型，通过线程的水平扩展来提升系统的吞吐和响应能力；Spring WebFlux使用事件触发机制的线程模型，在并发处理上可以使用少量的线程支撑高并发场景，收敛式的线程工作机制有利于充分利用物理资源，避免传统模式下线程阻塞等待的问题。
- 在上下游组件生态方面，Spring MVC有更强的优势，因为很多组件都使用了阻塞的交互方式，并且与Spring MVC框架都有很好的兼容性，所以生态方面有更好的适配和更高的成熟度。而Spring WebFlux因为采用异步非阻塞的响应式编程模型，所以目前在存储方面只有少数框架支持，主流的JDBC支持也还在探索当中。
- 共同点：Spring MVC和Spring WebFlux都可以使用注解式的开发方式，同时在Servlet 3.1异步规范下，Spring WebFlux也兼容主流的容器引擎，如Tomcat、Jetty等。同时，Reactive Client作为异步的HttpClient也适用于Spring MVC。

**Spring WebFlux的适用性**

通过上面的特性对比，我们可以发现，虽然Spring WebFlux有诸多性能优势，但是，在业务的适用性和开发者的学习成本上还是有一定限制的。所以我们在架构迁移之前，需要做好准备，才能避免更多问题。下面是Spring官方给出的一些建议。

- 如果你现在使用Spring MVC框架运行，能够支持现有业务对性能的诉求，就尽量保持不变，Spring MVC有大量的类库可供使用，实现简单，易于理解。
- 如果你希望实现轻量级的函数式Web框架，那么可以考虑Spring WebFlux的函数式Web端点。
- 如果你依赖阻塞的持久化API，比如JPA或者JDBC等组件，那么就只能选择Spring MVC框架。目前Spring WebFlux对于非阻塞的JDBC实现，有一些早期的项目在探索，但是还没有成熟的技术方案。
- 在Spring MVC应用程序中进行远程调用，可以使用响应式的WebClient。Spring MVC也可以使用其他响应式组件。
- 对于大型应用程序要考虑到非阻塞方式实现业务功能的学习曲线。最简单的起步方式就是使用WebClient，完全切换到Spring WebFlux框架需要花费精力来熟悉相关的函数式编程API。

**Spring WebFlux的局限**

- 性能的局限

在使用Spring WebFlux过程中，我们很容易犯一个错误，就是误认为只要使用Spring WebFlux，我们的Web服务框架就能在性能上得到极大的提升。而本质上，性能是由很多不同的指标来度量的。

根据Spring官方对Spring WebFlux框架的性能分析，Spring WebFlux并不能使我们的程序跑得更快，在没有WebClient的情况下，请求的延迟时间可能比阻塞式Web框架更长。Spring WebFlux的真正优势是解决Web的吞吐问题，通过非阻塞的编程模型范式可以避免线程的阻塞等待，从而提升系统的整体服务容量。也就是说，Spring WebFlux通过少量的线程就可以处理和应付流量激增的请求，在牺牲小部分请求延迟的情况下，系统的整体资源利用率仍然可以保持稳定，而这要得益于响应式编程模型和非阻塞线程处理模型。

● 开发生态的局限

目前在Java企业开发中，Spring WebFlux是相对成熟的非阻塞式Web开发解决方案。虽然目前Spring生态中有对Redis和MongoDB的非阻塞框架支持，但是上述两种存储方案都基于内存的数据库，而Spring WebFlux访问关系数据库就成为一个绕不开的问题。我们知道JDBC连接池采用阻塞方式，如果使用阻塞的数据库访问方式，那么Spring WebFlux就会退化为传统的阻塞调用方式。

虽然目前有类库宣称已经实现了对JDBC的异步调用，但是并没有成熟的案例应用到生产或者实践中。如果你已经有了一个大型研发团队，还要用Spring WebFlux技术栈，就必须要权衡陡峭的学习曲线和实际的项目收益。如果想要在实际项目中应用异步非阻塞框架，一个切实可行的方法就是使用Spring WebFlux技术组件，如WebClient，通过渐进的技术模块逐步了解相关的技术生态。

● 学习曲线高的局限

Spring WebFlux还有一个局限，就是它的学习曲线相对命令式编程语言还是比较高的，响应式编程模型比函数式编程在语法上更难掌握。习惯于面向对象编程思维的开发者不容易适应响应式编程风格和以数据流驱动的思维模式。这给聚焦业务功能的开发者带来了较高的技术门槛。另外，响应式编程中常用的操作符，也比较难掌握，需要花费额外的工夫和精力才能完全掌握它的具体用法。

总结一下，响应式编程是高负载、高并发、大数据量场景下的应用解决方案，适用于在异步边界作为非阻塞模块交互的技术解决方案。如果你的应用对消息的实时性、高负载、用户量等方面没有太大的诉求，那么使用Spring MVC这样传统的编程框架就足够。所以，在进行技术选型或者编程模型选择时，首先要从业务的性质、用户规模和实际使用场景出发，还要考虑团队技术人员的学习能力和知识储备。选择Spring WebFlux作为Web服务器框架还需要从上述技术、业务、人员等因素来权衡利弊。

## 10.4 Spring Cloud Gateway

Spring Cloud 2.x实现了社区生态下的Spring Cloud Gateway（简称SCG）微服务网关项目。Spring Cloud Gateway基于WebFlux框架开发，目标是替换掉Zuul。

### 10.4.1 Spring Cloud Gateway概述

Spring Cloud Gateway主要有两个特性：①非阻塞，默认使用RxNetty作为响应式Web容器，通过非阻塞方式，利用较少的线程和资源来处理高并发请求，并提升服务资源利用的可伸缩性。②函数式编程端点，通过使用Spring WebFlux的函数式编程模式定义路由端点，处理请求。

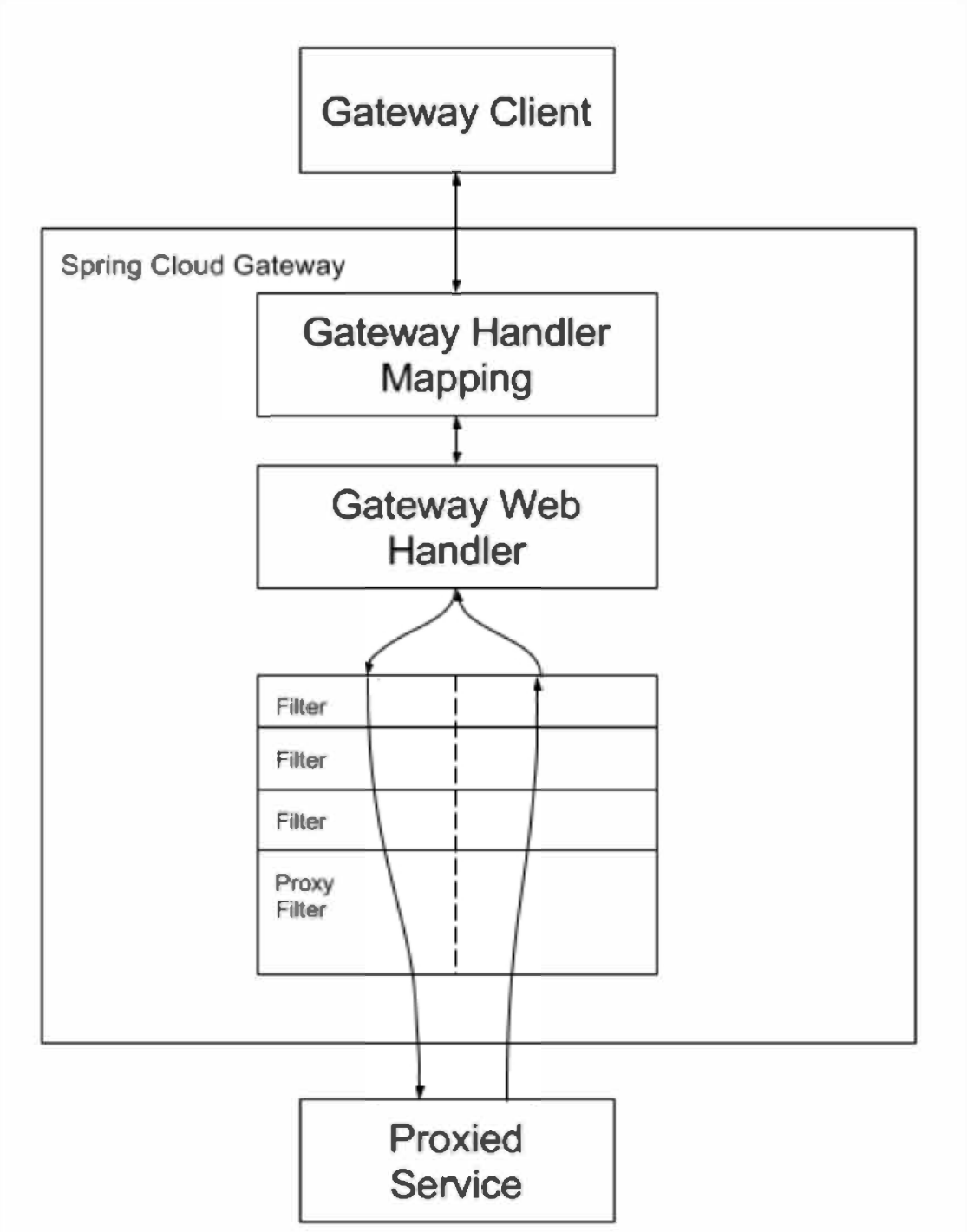
Spring Cloud Gateway可与Eureka、Ribbon、Hystrix等组件配合使用，基于Spring 5的Reactor和Spring Boot 2构建，使用Netty作为底层通信框架，支持异步非阻塞编程模型和响应式编程框架，解决了Zuul框架的I/O阻塞问题和线程收敛问题。使用Spring WebFlux框架可以使Spring Cloud Gateway在高并发场景下具有更好的性能表现，占用更少的资源。

下面是Spring Cloud官方对Spring Cloud Gateway特征的介绍。

- 基于Spring Framework 5、Reactor和Spring Boot 2.0框架。
- 根据请求的属性可以匹配对应的路由。
- 集成Hystrix。
- 集成Spring Cloud DiscoveryClient。
- 把易于编写的Predicates和Filters作用于特定路由。
- 具备一些网关的高级功能，如动态路由、限流、路径重写。

对于微服务网关来说，最核心的特征包括路由和过滤器机制。从功能特性上来看，Spring Cloud Gateway和Zuul具备相似的特性。它们都可以集成Hystrix、Ribbon负载均衡及Spring Cloud的现有组件来实现附加功能。而且Spring Cloud Gateway的本质特性还体现在底层的通信框架上，它可以基于Netty的I/O多路复用和事件响应机制来实现网络通信；它的另外一大特性就是使用Spring Framework 5的响应式编程模型，允许通过Spring WebFlux实现异步非阻塞特性，在性

能和资源利用率上，都有了质的提升。在编程范式上，Spring Cloud Gateway使用函数式编程模式。官方提供的Spring Cloud Gateway的架构图如下所示。




**Spring Cloud Gateway的核心概念**

简单说明一下架构图中的三个术语。

- Filter（过滤器）：和Zuul的过滤器在概念上类似，可以使用Filter拦截和修改请求，实现对上游的响应，进行二次处理，实现横切与应用无关的功能，如安全、访问超时设置、限流等功能。
- Route（路由）：网关配置的基本组成模块，和Zuul的路由配置模块类似。一个Route模块由一个ID、一个目标URI、一组断言和一组过滤器组成。如果断言为真，则路由匹配，目标

URI会被访问。

- Predicate（断言）：Predicate来自Java 8的接口，它可以用来匹配来自HTTP请求的任何内容，例如headers或参数。接口包含多种默认方法，并将Predicate组合成复杂的逻辑（与、或、非），可以用于接口参数校验、路由转发判断等。

**Spring Cloud Gateway的接入和配置**

Spring Cloud Gateway依赖Spring WebFlux提供的Netty运行时环境，所以Spring Boot必须是2.0或者以上版本。基本的Spring Cloud环境配置确认后，主要的接入步骤如下。

1.Maven依赖引入

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

2.路由配置方式一：配置文件方式

```
eureka:
    instance:
        prefer-ip-address: true
    client:
        service-url:
          defaultZone: http://localhost:8888/eureka/
        server:
           port: 8080
spring:
    application:
        name: scg-api-gateway
             cloud:
                 gateway:
                       routes:
                         -id: url-proxy-1
                            uri: https://localhost:8010
                            predicates:
                              -Path=/csdn
                         -id: message-provider-route
                           uri: lb://message-provider
                           predicates:
                              -Path=/message-provider/**
```

各字段含义如下。

- id：自定义的路由ID，保持唯一。
- uri：目标服务地址。
- predicates：路由条件，Predicate接受一个输入参数，返回一个布尔值结果。
  - 第一个Predicate基于URL的方式。配置文件的第一个路由的配置采用URL方式，配置了一个ID为url-proxy-1的URI代理规则。路由的规则为：当访问地址为http://localhost:8080/csdn/1.jsp时，会路由到上游地址https://localhost：8010/1.jsp。
  - 第二个Predicate基于服务ID发现的方式。配置文件的第二个路由的配置采用与注册中心相结合的服务发现方式，与单个URI的路由配置相比，区别其实很小，仅在于URI的schema协议不同。单个URI地址的schema协议，一般为HTTP或者HTTPs协议。

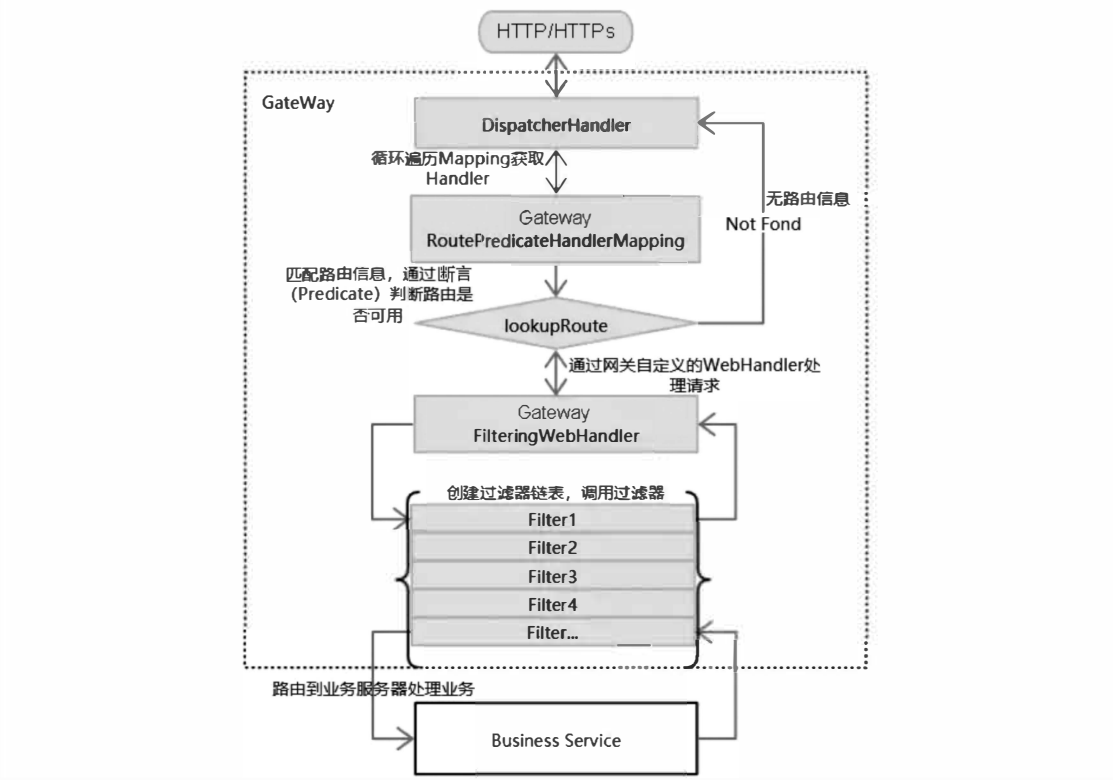
3.基于代码DSL方式的路由配置接入

路由转发功能同样可以通过代码来实现，我们可以在启动类GatewayApplication中添加customRouteLocator方法来定制转发规则，代码如下：

```
@SpringBootApplication
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class， args);
    }
    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
                .route("path_route"， r -> r.path("/csdn").uri("https://localhost:8010"))
                .build();
    }
}
```

## 10.4.2 Spring Cloud Gateway的工作原理

客户端向Spring Cloud Gateway发出HTTP请求后，如果Gateway HandlerMapping确定请求与路由匹配，则将其发送到Gateway WebHandler。WebHandler通过该请求的特定过滤器链处理请求。过滤器可以在发送代理请求之前或之后执行逻辑。在Spring Cloud Gateway的执行流程中，首先执行所有“pre filter”逻辑，然后进行回源请求代理。在请求代理执行完后，执行“post filter”逻辑。在“pre”类型的过滤器中，可以实现参数校验、权限校验、流量监控、日志输出、协议转换等功能；在“post”类型的过滤器中，可以实现响应内容、响应头的修改，日志的输出、流量监控等功能。核心工作流程如下图所示。




**Predicate条件**

在Spring Cloud Gateway中，Spring利用Predicate的特性实现了各种路由匹配规则，通过Header、请求参数等不同条件来匹配对应的路由。

我们来看Spring Cloud Gateway内置的几种Predicate的使用方法。

```
server:
  port: 8080
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: gateway-service
          uri: https://localhost:8010
          order: 0
          predicates:
             - Host=**.foo.org
             - Path=/headers
             - Method=GET
             - Header=X-Request-Id， \d+
             - Query=foo， ba.
             - Query=baz
             - Cookie=chocolate， ch.p
             - After=2020-08-20T17:42:47.789-07:00[America/Denver]
             - Before=2020-08-20T17:42:47.789-07:00[America/Denver]
```

在上述配置文件中，如果多种Predicates同时存在于同一个路由，请求必须同时满足所有条件才能被这个路由匹配。当一个请求满足多个路由的Predicate条件时，请求只会被首个成功匹配的路由转发。下面分别对不同规则的路由匹配进行解释。

● 通过请求路径匹配（Path Route Predicate）

路由断言工厂接收一个参数，根据Path定义好的规则来判断访问的URI是否匹配。配置示例如下：

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: https://localhost:8010
        predicates:
        - Path=/hello/{segment}
```

如果请求路径符合要求，则此路由将匹配，例如/hello/1或者/hello/world。

使用curl测试，命令行输入：

```
curl http://localhost:8080/hello/1
curl http://localhost:8080/hello/xx
curl http://localhost:8080/boo/xx
```

经过测试发现，第一条和第二条命令可以正常获取页面返回值，最后一个命令报404错误，证明路由是通过指定路径来匹配的。

● 通过请求参数匹配（Query Route Predicate）

路由断言工厂接收两个参数：一个必需的参数和一个可选的正则表达式。配置示例如下：

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://localhost:8010
        predicates:
        - Query=helloworld
```

在这样的配置中，只要请求中包含helloworld属性的参数即可匹配路由。使用curl测试，命令行输入：

```
curl localhost:8080?helloworld=x&id=2
```

经过测试发现，只要请求中带有helloworld参数就会匹配路由，不带helloworld参数则不会匹配。还可以将Query的值以键值对的方式进行配置，这样在请求时会对属性值和正则表达式都进行匹配，键值对匹配后才会正确执行路由逻辑。

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: localhost:8010
        predicates:
        - Query=hello, world.
```

在上述路由匹配中，请求中包含hello属性并且参数值是以world开头的、长度为三位的字符串，才会进行匹配和路由。使用curl测试，命令行输入：

```
curl localhost:8080?hello=world
```

测试可以返回正确的页面代码。如果将hello的属性值改为ok，再次访问就会报404错误，证明路由需要匹配正则表达式才会进行路由。

● 通过请求方法匹配

路由断言工厂接收一个参数，即需要匹配HTTP方法。通过POST、GET、PUT、DELETE等不同的请求方式来进行路由。

```
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: http://localhost:8010
        predicates:
        - Method=GET
```

使用curl测试（＃curl默认以GET的方式去请求），命令行输入：

```
curl http://localhost:8080
```

测试返回页面代码，证明匹配到路由。

我们再以POST的方式请求测试。

```
curl -X POST http://localhost:8080
```

返回404错误表示没有找到，证明没有匹配上路由。

● 通过Header属性匹配

路由断言工厂接收两个参数，分别是请求头名称和正则表达式。Header Route Predicate和Cookie Route Predicate一样，也是接收2个参数：一个header的属性值和一个正则表达式。这个属性值和正则表达式匹配则执行。

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: localhost:8010
        predicates:
        - Header=X-Request-Id，  \d+
```

● 通过Host路由匹配

Spring Cloud Gateway可以根据Host名进行匹配转发，Host Route Predicate接收一组参数、一组匹配的域名列表。它通过参数中的主机地址作为匹配规则。

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: http://xxx.com
        predicates:
        - Host=**.xxx.com
```

使用curl测试，命令行输入：

```
curl http://localhost:8080  -H "Host: www.xxx.com"
curl http://localhost:8080  -H "Host: hello.xxx.com"
```

通过测试以上两种Host设置方式，均可匹配到host_route，去掉host参数则会报404错误。

● 时间匹配

Predicate支持设置时间，在请求转发时，先判断这个时间与我们设置的时间，然后进行转发，所以又细分为设置时间后断言、设置时间前断言、设置时间之间断言。

设置时间后断言：从After Route Predicate Factory中获取一个UTC时间格式的参数，当请求的当前时间在配置的UTC时间之后，则成功匹配，否则不能成功匹配。下面是实例配置：

```
spring:
  cloud:
    gateway:
      routes:
       - id: time_route
         uri: localhost:8010
         predicates:
          - After=2018-01-20T06:06:06+08:00[Asia/Shanghai]
```

设置时间前断言：从Before Route Predicate Factory中获取一个UTC时间格式的参数，当请求的当前时间在配置的UTC时间之前，则成功匹配，否则不能成功匹配。下面是实例配置：

```
spring:
  cloud:
    gateway:
      routes:
       - id: after_route
         uri: localhost:8010
         predicates:
          - Before=2018-01-20T06:06:06+08:00[Asia/Shanghai]
```

设置时间之间断言：从Between Route Predicate Factory中获取一个UTC时间格式的参数，当请求的当前时间在配置的UTC时间之间，则成功匹配，否则不能成功匹配。下面是实例配置：

```
spring:
  cloud:
    gateway:
      routes:
       - id: after_route
         uri: localhost:8010
         predicates:
          -Between=2018-01-20T06:06:06+08:00[Asia/Shanghai],
             2019-01-20T06:06:06+08:00[Asia/Shanghai]
```

● 通过Cookie匹配

Cookie路由断言会取两个参数，一个是Cookie name，一个是正则表达式，路由规则是通过获取的对应Cookie name值和正则表达式进行匹配，如果匹配上就会执行路由，如果没有匹配上则不执行。

```
spring:
  cloud:
    gateway:
      routes:
          - id: cookie_route
              uri: http://localhost/8050/test/cookie
              predicates:
              - Cookie=hello,  test
```

● 通过IP地址匹配

RemoteAddr Route Predicate Factory配置一个IPv4或者IPv6网段的字符串或者IP地址。当请求的IP地址在网段之内或者与配置的IP地址相同，匹配成功，则进行转发，否则不进行转发。

```
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: localhost:8010
        predicates:
        - RemoteAddr=192.168.1.1/50
```

可以将curl localhost：8080设置为本机的IP地址进行测试，如果请求的远程地址是192.168.1.30，则此路由将匹配。

**GatewayFilter与GlobalFilter**

Spring Cloud Gateway中有两种Filter，一种是GlobalFilter（全局过滤器），一种是GatewayFilter。GlobalFilter默认对所有路由有效，GatewayFilter需要通过路由分组指定。GlobalFilter接口与GatewayFilter具有相同的签名，是有条件地应用于所有路由的特殊过滤器。

当请求进入路由匹配逻辑时，Web Handler会将GlobalFilter的所有实例和所有GatewayFilter路由特定实例添加到Filter Chain组件。Filter组合执行的顺序由Ordered接口决定，可以通过getOrder方法或使用@Order注释来设置。Spring Cloud Gateway通过执行过滤器将逻辑分为“前置”和“后置”阶段，优先级较高的前置过滤器会优先被执行，而优先级较高的后置过滤器的执行顺序正好相反，最后执行。

**GatewayFilter Factories**

过滤器允许以某种方式修改传入的HTTP请求或返回的HTTP响应。过滤器的作用域是某些特定路由。Spring Cloud Gateway包括许多内置的过滤器工厂。

- 实现前缀修改（增加前缀、去掉前缀）

PrefixPathGatewayFilterFactory及StripPrefixGatewayFilterFactory是一对处理请求URL的前缀的Filter工厂，前者添加前缀，后者去除前缀。

配置文件application.yml如下：

```
spring:
  cloud:
    gateway:
    #配置所有路由的默认过滤器，这里配置的是 gatewayFilter
      default-filters:
      routes:
      - id: server-test  #服务的 id
        uri: lb://server-test #服务的 Application 名称
        order: 0 #路由级别
        predicates:
        - Path=/bus/**  #前缀
        filters:
            - PrefixPath=/mypath
            - id: server-test  #服务的 id
          uri: http;//nameserver #服务的 Application 名称
        order: 0 #路由级别
        predicates:
        - Path=/name/**  #前缀
            - StripPrefix=2 #去掉前缀，去几层
```

  - ○ PrefixPathGatewayFilterFactory允许你在对应的路由请求前增加前缀。例如实例配置中的请求/hello，最后转发到目标服务的路径变为/mypath/hello。
  - ○ StripPrefixGatewayFilterFactory允许你在对应的路由请求前去除前缀，例如实例配置中的请求/name/bar/foo，去除前面两个前缀后，最后转发到目标服务的路径为/foo。
- 实现请求头内容添加和改写

AddRequestHeader GatewayFilter Factory采用一对名称和值作为参数，配置文件application.yml如下：

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: http://localhost:8010
        filters:
        - AddRequestHeader=X-Request-Foo，Bar
```

对于所有匹配的请求，将在向下游请求的头内容中添加x-request-foo：bar header。

● 实现请求体内容添加和改写

AddRequestParameter GatewayFilter Factory采用一对名称和值作为参数，配置参数application.yml如下：

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri:localhost:8010
        filters:
        - AddRequestParameter=foo，bar
```

对于所有匹配的请求，将向下游请求添加foo=bar查询字符串。

● 实现熔断降级

Hystrix GatewayFilter允许向网关路由引入Hystrix，保护服务不受级联故障的影响，并允许在下游故障时提供fallback响应。要在项目中启用Hystrix网关过滤器，需要向Hystrix的依赖Hystrix GatewayFilter Factory添加一个name参数，即HystrixCommand的名称，配置文件application.yml如下：

```
spring:
  cloud:
    gateway:
      routes:
      - id: hystrix_route
        uri: lb://test-service:8080
        filters:
        - name: Hystrix
          args:
            name: fallbackcmd
            fallbackUri: forward:/incaseoffailureusethis
        - RewritePath=/consumingserviceendpoint，/backingserviceendpoint
```

当调用hystrix fallback时，将转发到/incaseoffailureusethis。注意，这个示例还演示了通过目标URI上的“lb”前缀使Spring Cloud Netflix Ribbon客户端实现负载均衡。主要场景是网关应用程序中的内部控制器或处理程序使用fallbackUri，它也可以将请求重新路由到外部应用程序中的控制器或处理程序。

● 分布式限流

Spring Cloud Gateway内置的RequestRateLimiterGatewayFilterFactory提供限流的能力，基于令牌桶算法实现。目前它内置的RedisRateLimiter，依赖Redis来存储限流配置和统计数据。当然你也可以实现自己的RateLimiter，只需实现Spring Cloud Gateway自带的RateLimiter接口或者继承AbstractRateLimiter。

首先，添加Maven依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

其次，添加限流配置。

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: lb://user-center
          predicates:
            - TimeBetween=上午 0:00，下午 11:59
          filters:
            - AddRequestHeader=X-Request-Foo， Bar
            - name: RequestRateLimiter
              args:
                #令牌桶每秒填充的平均速率
                redis-rate-limiter.replenishRate: 1
                #令牌桶的上限
                redis-rate-limiter.burstCapacity: 2
                #使用 SpEL 表达式从 Spring 容器中获取 Bean 对象
                key-resolver: "#{@pathKeyResolver}"
  redis:
    host: 127.0.0.1
    port: 6379
```

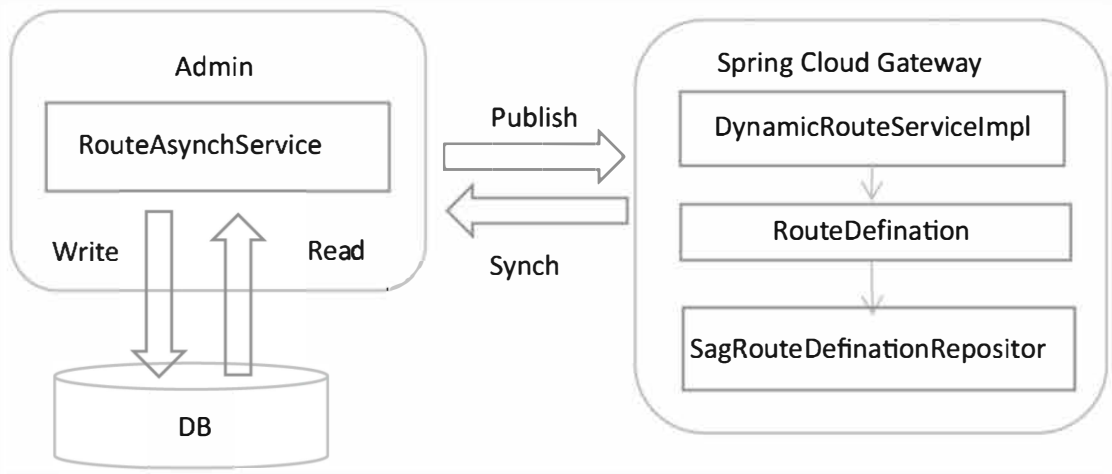
最后，完成对Path的KeyResolver（可以通过KeyResolver来指定限流的Key），实现对特定Path下的限流控制配置。在过滤器中可以配置一个可选的KeyResolver，KeyResolver在配置中根据名称使用SpEL引用Bean。#{@myKeyResolver}是引用名为“pathKeyResolver”的Bean的SpEL表达式。KeyResolver接口允许使用可插拔策略来派生限制请求的Key。代码如下：

```
@Configuration
public class Configuration {
    /**
     * 按照 Path 限流
     *
     * @return key
     */
    @Bean
    public KeyResolver pathKeyResolver() {
        return exchange -> Mono.just(
            exchange.getRequest()
                .getPath()
                .toString()
        );
    }
}
```

## 10.4.3 Spring Cloud Gateway的动态路由

下面介绍基于Spring Cloud Gateway的动态路由实现（相关代码将会随书附带），实现方式与Zuul的动态路由实现方式类似，具有比Zuul更加灵活的路由策略和匹配模式。这两种解决方案如下。

- 通过Spring Cloud Gateway提供的GatewayControllerEndpointduan端点功能，实现路由的增删改查，或者自己实现ApplicationEventPublisherAware接口，实现自定义的路由操作方法。具体可以参考源码：GatewayControllerEndpointduan类。
- 通过实现RouteDefinitionRepository接口，实现自定义的Repository类，实现从数据库或者缓存中动态加载路由信息的功能。架构模式与Zuul的动态路由采用相似的路由加载策略，架构流程图如下。




动态路由思路及解决方案具体如下。

首先，Admin作为前端管理界面，将用户对路由的添加、修改等操作通过RouteAsynchService存储到DB中。DB中的存储结构如下图所示。

| id | routeid | routename | routeorder | routestatus | strategy | predicates | filters | uri | groupName |
|---|---|---|---|---|---|---|---|---|---|
| 2 | testAvnchRoute333 | 异步网关测试 | 0 | ONLINE | serviceid | {"Path":"/dtest88/**"} | {"StripPrefix":"1"} | DOWNSTREAM | SCG-GATEWAY-CORE |
| 3 | testAvnchRoute3366663 | 异步网关测试 | 0 | EDIT | serviceid | {"Path":"/dtest88/**"} | {"StripPrefix":"1"} | DOWNSTREAM | SCG-GATEWAY-CORE |
| 4 | dsdssdf | sdfdfasdasdd | 1 | ONLINE | SERVICEID | {"Path":"/asfasf/?"} | {"StripPrefix":"1"} | asfasf | SCG-GATEWAY-CORE |
| 5 | testAvnchRoute | 异步网关测试222222 | 1 | ONLINE | SERVICEID | {"Path":"/adssd/**"} | {"StripPrefix":"1"} | asdasd | SCG-GATEWAY-CORE |
| 6 | werw | werwr | 1 | ONLINE | SERVICEID | {} | {"StripPrefix":""} | sdsdfdf22222 | SCG-GATEWAY-CORE |
| 7 | werwer | werwer | 0 | ONLINE | SERVICEID | {"Path":"/qweoewf?"} | {"StripPrefix":""} | qweqweoew | SCG-GATEWAY-CORE |
| 8 | qwewe | qeoew33333 | 1 | ONLINE | SERVICEID | {"Path":"/qqwe/?"} | {"StripPrefix":"1"} | qewqwe | SCG-GATEWAY-CORE |

字段映射关系如下。

- routeid：标识路由的唯一ID，可以根据路由ID查找路由，路由ID不能重复。
- routename：应用名称是标识路由的别名，是非必选项。
- routeorder：对应RouteDefinition中的order属性。
- routestatus：路由状态，包括编辑、发布、下线等状态。
- strategy：路由策略，和Zuul的路由策略相似，也支持ServiceID策略和URL策略。
- predicates：对应RouteDefinition中的List<PredicateDefinition> predicates策略集合，以键值对的形式对应断言策略。

- filters：对应RouteDefinition中的List<FilterDefinition> filters集合，以键值对的形式对应过滤器策略。
- uri：对应后端服务，可以是后端服务的ServiceID，也可以是服务的URL地址，与路由策略对应。
- groupname：标识这个新建的路由归属在哪个网关集群下面。

其次，Spring Cloud Gateway的动态路由管理策略都通过Admin接收对网关路由的增删改查命令，然后通过RouteAsynchService将路由更新服务并发布到对应的网关节点，网关节点从数据库动态获得最新的路由状态，更新缓存和当前路由。下面对网关节点的事件监听机制进行讲解。

```
@Service
public class DynamicRouteServiceImpl implements ApplicationEventPublisherAware,
        DynamicRouteService{
    private ApplicationEventPublisher publisher;
    @Autowired
    SagRouteDefinationRepository routedefine;
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher
        applicationEventPublisher) {
        this.publisher = applicationEventPublisher;
    }
    /**
     * 刷新路由表
     */
    @Override
    public boolean refreshRoute() {
        routedefine.setRefreshNeed(true);
        publisher.publishEvent(new RefreshRoutesEvent(this));
        return true;
    }
      /**
     * 获取路由表
     */
    @Override
    public Flux<RouteDefinition> getAllRoute(Map<String,  Object> params) {
        return routedefine.getRouteDefinitions();
    }
}
```

说明 1#

说明1#：在代码段中，refreshRoute方法是事件监听的入口方法，该方法会向Admin管理服务暴露一个REST服务。当Admin对路由进行更改后，会调用refreshRoute方法，触发Spring Cloud Gateway自带的RefreshRoutesEvent事件，同时设置原子布尔变量routedefine为true，在下面的动态路由加载中根据该原子布尔变量决定是从数据库中读取路由还是从缓存中读取路由。

下面是定制化的核心路由动态加载和缓存管理的关键代码，主要通过实现自定义的路由Repository加载类来动态地加载路由，通过继承RouteDefinitionRepository父类来提供路由的配置信息，实现逻辑如下：

```
@Component
public class SagRouteDefinationRepository implements RouteDefinitionRepository {
    /**
     * 路由缓存
     */
    private Set<RouteDefinition> routeDefinitions = new HashSet<>();
    private AtomicBoolean refreshNeed = new AtomicBoolean(false);
    @Autowired
    private SpiService spiservice;
    private static final String SERVICED = "SERVICEID";
    private static final String SERVICEURL = "SERVICEURL";
    @SuppressWarnings("unchecked")
    @Override
    public Flux<RouteDefinition> getRouteDefinitions() {
        Set<RouteDefinition> routeDefineSet = new HashSet<>();
        /**
         * step1: 如果缓存不空&&加载开关关闭，从缓冲中取得路由
         */
        if ((routeDefinitions != null) && (refreshNeed.get() == false)) {
            LOGGER.info("> Load RouteDefinitions from local cache..");
            routeDefineSet.addAll(routeDefinitions);
        }
        else {
            /**
             * step2: 从数据库中加载路由到缓存，并关闭加载开关
             *
             */
            if (refreshNeed.get()) {
                refreshNeed.compareAndSet(true， false);
            }
            List<RouteDefinitionVo> results =
               (List<RouteDefinitionVo>) spiservice.localteRoutefromDB();
            transfer(results， routeDefinitions);
            routeDefineSet.addAll(routeDefinitions);
        }
        return Flux.fromIterable(routeDefineSet);
    }
    public AtomicBoolean getRefreshNeed() {
        return refreshNeed;
    }
    public void setRefreshNeed(boolean needRefresh) {
        refreshNeed.set(needRefresh);
    }
}
```

说明 2#

说明 3#

说明 4#

说明2＃：在代码段中，SagRouteDefinationRepository是自定义的路由加载实现类，这个类实现了RouteDefinitionRepository接口。该接口的源码如下：

```
public interface RouteDefinitionRepository extends RouteDefinitionLocator，
    RouteDefinitionWriter {
}
public interface RouteDefinitionLocator {
    Flux<RouteDefinition> getRouteDefinitions();
}
```

然后，跟进getRouteDefinitions方法，它是RouteDefinitionRouteLocator的回调方法，可以实时更新路由信息，代码如下：

```
public class RouteDefinitionRouteLocator implements
    RouteLocator, BeanFactoryAware, ApplicationEventPublisherAware {
       public Flux<Route> getRoutes() {
         return this.routeDefinitionLocator.getRouteDefinitions().map(this::convertToRoute)
         //TODO: error handling
         .map(route -> {
            if (logger.isDebugEnabled()) {
                logger.debug("RouteDefinition matched: " + route.getId());
            }
            return route;
    });
}
```

从源码中调用链路追溯，可以发现下面的调用链路：

```
DispatcherHandler: handle(ServerWebExchange exchange)
 AbstractHandlerMapping: getHandler(ServerWebExchange exchange)
  RoutePredicateHandlerMapping: getHandlerInternal(ServerWebExchange exchange)
   RoutePredicateHandlerMapping: lookupRoute(ServerWebExchange exchange)
                                RouteDefinitionRouteLocator : getRoutes()
```

说明3＃：在代码段中，refreshNeed（）方法是判断缓存是否失效的标识原子布尔变量，当Admin回调1＃代码段中的刷新接口时，会将该失效接口打开。在路由加载时，如果refreshNeed为false并且routeDefnitions不为空，那么优先加载缓存中的路由信息。如果refreshNeed为true，那么优先执行加载数据库的操作，通过这段代码的逻辑处理就可以保证网关中路由的刷新效率和缓存与数据库中路由信息的同步。

说明4＃：该代码段是从数据库中加载路由的核心实现。localteRoutefromDB（）方法从数据库中加载路由，返回RouteDefnitionVo模型的数据库路由列表信息。下面是该模型类的代码：

```
@EqualsAndHashCode
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class RouteDefinitionVo {
    //路由的 Id
    private String routeid;
    //路由的 Name
    private String routename;
    //路由执行的顺序
    private int routeorder = 0;
    //路由执行的顺序
    private String routestatus ;
    //路由策略
    private String strategy ;
    //路由规则转发的目标 uri
    private String uri ;
    //路由网关组
    private String groupName ;
    //路由断言集合配置
    private String predicates;
 }
```

transfer（）方法实现了从RouteDefnitionVo到RouteDefnition的类型转换，下面是transfer（）方法调用的类型转换的核心代码：

```
private RouteDefinition convertRouteDefinition(RouteDefinitionVo routevo) {
        RouteDefinition definition = new RouteDefinition();
        definition.setId(routevo.getRouteid());
        definition.setOrder(routevo.getRouteorder());
        String strategy = routevo.getStrategy().toUpperCase();
        String uriString = null;
        switch (strategy) {
            case SERVICED:
                uriString = "lb://" + routevo.getUri();
                break;
            case SERVICEURL:
                uriString = routevo.getUri();
                break;
            default:
                uriString = routevo.getUri();
                break;
        }
        URI uri = UriComponentsBuilder.fromUriString(uriString).build().toUri();
        definition.setUri(uri);
        /**
         * Predicates
         */
        List<PredicateDefinition> pdList = new ArrayList<>();
        String predicateJsonString = routevo.getPredicates();
        @SuppressWarnings("rawtypes")
        Map predicat = JsonHelper.toObject(predicateJsonString,  Map.class);
        @SuppressWarnings("unchecked")
        Set<String> keyset = predicat.keySet();
        for (String name : keyset) {
            PredicateDefinition predicate = new PredicateDefinition();
            predicate.setName(name);
            predicate.setArgs(GatewayPredicateDefinitionFactory.getArgs(name,
                              predicat));
            pdList.add(predicate);
        }
        definition.setPredicates(pdList);
        /**
         * Filters
         */
        String filtersJsonString = routevo.getFilters();
        @SuppressWarnings("rawtypes")
        Map filterMap = JsonHelper.toObject(filtersJsonString,  Map.class);
        @SuppressWarnings("unchecked")
        Set<String> keysetFilter = filterMap.keySet();
        List<FilterDefinition> fList = new ArrayList<>();
        for (String name : keysetFilter) {
            FilterDefinition filter = new FilterDefinition();
            filter.setName(name);
            filter.setArgs(GatewayFilterDefinitionFactory.getArgs(name,  filterMap));
            fList.add(filter);
        }
        definition.setFilters(fList);
        return definition;
}
```

说明 5#

说明5＃：在代码段中，GatewayPredicateDefinitionFactory完成断言的模式匹配转换。Predicate-Definition是断言的模型定义，定义name为Key、args为Value。举例如下：

```
predicates[0].name: Path
predicates[0].args[pattern]: "'/'+serviceId+'/**'"
```

GatewayPredicateDefinitionFactory完成过滤器的模式匹配转换。FilterDefinition是过滤器的模型定义，定义name为Key、args为Value。举例如下：

```
filters[1].name: RewritePath
filters[1].args[replacement]: "'/${remaining}'"
```

## 10.4.4 Spring Cloud Gateway源码解析

启动Spring Cloud Gateway，需要依赖官方的Starter组件。下面我们从Maven依赖开始，对Spring Cloud Gateway的源码进行解析。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

**初始化加载**

上述是spring-cloud-starter-gateway启动前需要引用的一个自动配置Starter，可以通过查询该Starter的源码发现Spring Cloud Gateway的实现所依赖的组件，Maven配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>spring-cloud-dependencies-parent</artifactId>
        <groupId>org.springframework.cloud</groupId>
        <version>2.0.4.RELEASE</version>
        <relativePath/>
    </parent>
    <artifactId>spring-cloud-gateway-dependencies</artifactId>
    <version>2.0.2.RELEASE</version>
    <packaging>pom</packaging>
    <name>spring-cloud-gateway-dependencies</name>
    <description>Spring Cloud Gateway Dependencies</description>
    <properties>
    </properties>
     <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gateway-webflux</artifactId>
                <version>${project.version}</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gateway-mvc</artifactId>
                <version>${project.version}</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gateway-core</artifactId>
                <version>${project.version}</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-gateway</artifactId>
                <version>${project.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

可以看到Spring Cloud Gateway的Starter启动类主要依赖spring-cloud-gateway-core组件。使用EnableAutoConfiguration注解完成自动配置初始化信息，我们在Spring Cloud Gateway下的spring.factories（在包spring-cloud-gateway-core）声明文件如下：

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
//依赖包的校验配置
org.springframework.cloud.gateway.config.GatewayClassPathWarningAutoConfiguration，\
//网关的核心配置
org.springframework.cloud.gateway.config.GatewayAutoConfiguration，\
//负载均衡相关的依赖配置信息
org.springframework.cloud.gateway.config.GatewayLoadBalancerClientAutoConfiguration，\
//流控的依赖配置信息
org.springframework.cloud.gateway.config.GatewayRedisAutoConfiguration，\
//注册中心相关的依赖配置
org.springframework.cloud.gateway.discovery.GatewayDiscoveryClientAutoConfiguration
```

## GatewayAutoConfiguration

```
@Configuration
@ConditionalOnProperty(name = "spring.cloud.gateway.enabled",  matchIfMissing = true)
@EnableConfigurationProperties
@AutoConfigureBefore(HttpHandlerAutoConfiguration.class)
@AutoConfigureAfter({GatewayLoadBalancerClientAutoConfiguration.class,
    GatewayClassPathWarningAutoConfiguration.class})
@ConditionalOnClass(DispatcherHandler.class)
public class GatewayAutoConfiguration {
    @Configuration
    //当 classpath 中存在 HttpClient
    @ConditionalOnClass(HttpClient.class)
    protected static class NettyConfiguration {
        ...
    }
    //加载配置 Beans
    //ConfigurationProperty Beans
    @Bean
    public GatewayProperties gatewayProperties() {
        return new GatewayProperties();
    }
    @Bean
    public SecureHeadersProperties secureHeadersProperties() {
        return new SecureHeadersProperties();
    }
    //GlobalFilter Beans
    //加载全局过滤器
    @Bean
    public AdaptCachedBodyGlobalFilter adaptCachedBodyGlobalFilter() {
        return new AdaptCachedBodyGlobalFilter();
    }
    @Bean
    public RouteToRequestUrlFilter routeToRequestUrlFilter() {
        return new RouteToRequestUrlFilter();
    }
    @Bean
    @ConditionalOnBean(DispatcherHandler.class)
    public ForwardRoutingFilter forwardRoutingFilter(DispatcherHandler
        dispatcherHandler) {
         return new ForwardRoutingFilter(dispatcherHandler);
```

```
        }
        @Bean
        public ForwardPathFilter forwardPathFilter() {
            return new ForwardPathFilter();
        }
        @Bean
        public WebSocketService webSocketService() {
            return new HandshakeWebSocketService();
        }
        @Bean
        public WebsocketRoutingFilter websocketRoutingFilter(WebSocketClient
            webSocketClient, ObjectProvider<List<HttpHeadersFilter>>headersFilters) {
                return new WebsocketRoutingFilter(webSocketClient,
            webSocketService,  headersFilters);
        }
        @Bean
        public WeightCalculatorWebFilter weightCalculatorWebFilter(Validator validator) {
            return new WeightCalculatorWebFilter(validator);
        }
}
```

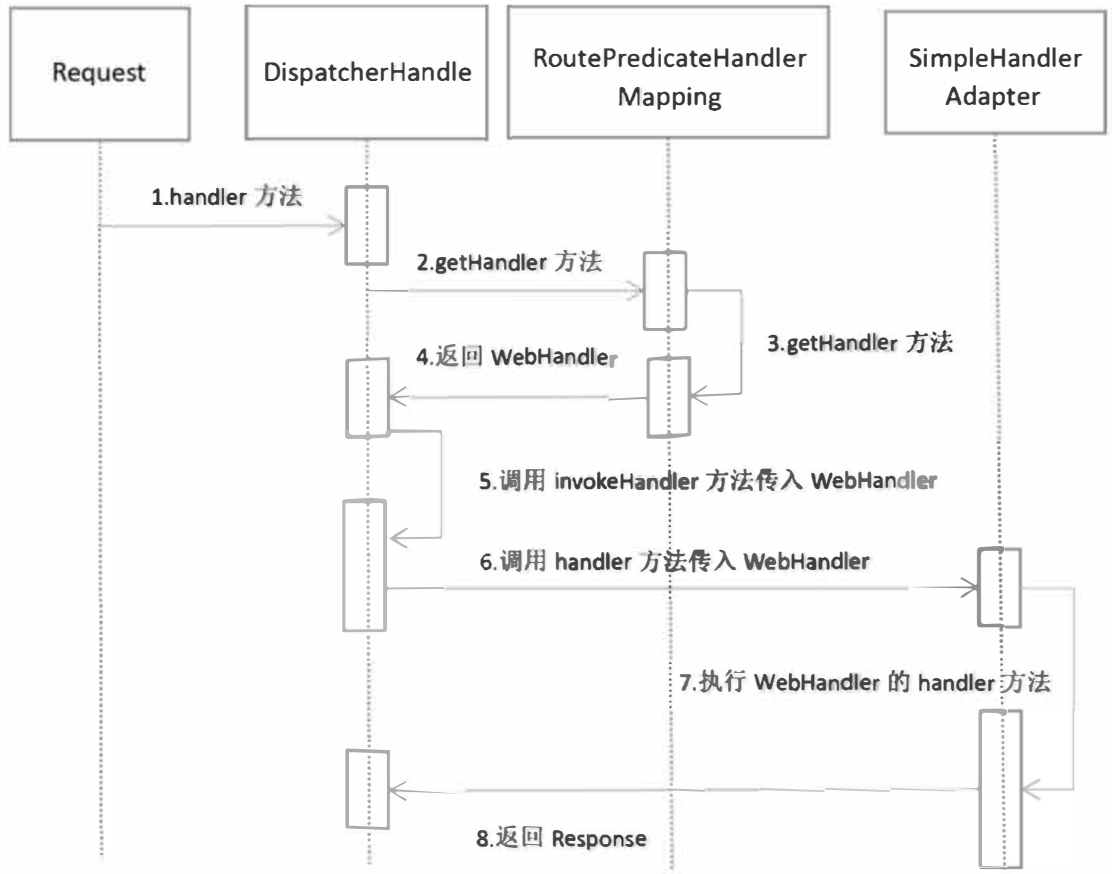
**说明：**

GatewayAutoConfiguration配置是Spring Cloud Gateway的核心配置类，初始化如下组件：

- NettyConfiguration
- GlobalFilter（AdaptCachedBodyGlobalFilter、RouteToRequestUrlFilter、ForwardRoutingFilter、ForwardPathFilter、WebsocketRoutingFilter、WeightCalculatorWebFilter等）
- FilteringWebHandler
- GatewayProperties
- PrefixPathGatewayFilterFactory
- RoutePredicateFactory
- RouteDefinitionLocator
- RouteLocator
- RoutePredicateHandlerMapping（查找匹配到的Route并进行处理）
- GatewayWebfluxEndpoint（管理网关的HTTP API）

**HTTP请求路由源码分析**

Spring Cloud Gateway中使用HandlerMapping对请求的链接进行解析，匹配对应的Route，转发到对应的服务。下图为整个请求的流程，用户请求先通过DispatcherHandler找到对应的GatewayHandlerMapping，再通过GatewayHandlerMapping解析匹配到的Handler；Handler处理完后，经过Filter处理，最终将请求转发到后端服务。




在前面的动态路由加载过程中，其实已经贯穿了整个HTTP请求的调用链路，具体如下：

```
DispatcherHandler: handle(ServerWebExchange exchange)
  AbstractHandlerMapping: getHandler(ServerWebExchange exchange)
    RoutePredicateHandlerMapping: getHandlerInternal(ServerWebExchange exchange)
      RoutePredicateHandlerMapping: lookupRoute(ServerWebExchange exchange)
                                RouteDefinitionRouteLocator : getRoutes()
                                                        FilteringWebHandler: handler
```

请求先由DispatcherHandler进行处理，DispatcherHandler在初始化时会在Spring IoC容器中查找实现HandlerMapping接口的实现类。然后保存到内部变量handlerMappings数据结构中。DispatcherHandler调用handler方法迭代handlerMappings中的HandlerMapping接口，主要源码如下：

```
@Override
public Mono<Void> handle(ServerWebExchange exchange) {
```

```
        if (logger.isDebugEnabled()) {
            ServerHttpRequest request = exchange.getRequest();
            logger.debug("Processing " + request.getMethodValue() + " request for
                 [" + request.getURI() + "]");
        }
        if (this.handlerMappings == null) {
            return Mono.error(HANDLER_NOT_FOUND_EXCEPTION);
        }
        return Flux.fromIterable(this.handlerMappings)
            .concatMap(mapping -> mapping.getHandler(exchange))
                .next()
                .switchIfEmpty(Mono.error(HANDLER_NOT_FOUND_EXCEPTION))
                .flatMap(handler -> invokeHandler(exchange, handler))
                .flatMap(result -> handleResult(exchange, result));
    }
```

AbstractHandlerMapping 在 getHandler 方法中封装了 CORS （Cross-Origin Resource Sharing，跨域资源共享）。因为所有 Handler 都可能涉及 CORS 的处理，所以抽象类 AbstractHandlerMapping提供了getHandlerInternal子类来实现查找Handler的具体方法。

```
    @Override
    public Mono<Object> getHandler(ServerWebExchange exchange) {
        return getHandlerInternal(exchange).map(handler -> {
            if (CorsUtils.isCorsRequest(exchange.getRequest())) {
                CorsConfiguration configA =
                    this.globalCorsConfigSource.getCorsConfiguration(exchange);
                CorsConfiguration configB = getCorsConfiguration(handler, exchange);
                CorsConfiguration config = (configA != null ? configA.combine(configB) : configB);
                if (!getCorsProcessor().process(config, exchange) ||
                    CorsUtils.isPreFlightRequest(exchange.getRequest())) {
                    return REQUEST_HANDLED_HANDLER;
                }
            }
            return handler;
        });
    }
```

RoutePredicateHandlerMapping用于匹配具体的路由，并返回FilteringWebHandler。通过RoutePredicateHandlerMapping中的RouteLocator对象存储启动时加载的路由对象信息。当RoutePredicateHandlerMapping 获取对应的路由时，会将 Route 信息存储到 ServerWebExchanges 属性中，然后返回实现了 WebHandler 接口的 FilteringWebHandler 。FilteringWebHandler是一个存放过滤器的Handler。

调用RoutePredicateHandlerMapping的getHandlerInternal方法从RouteLocator获取路由，并存放在ServerWebExchange中，返回webFilter对象，代码如下：

```
protected Mono<?> getHandlerInternal(ServerWebExchange exchange) {
 exchange.getAttributes().put(GATEWAY_HANDLER_MAPPER_ATTR, getClass().getSimpleName());
    return lookupRoute(exchange)this.flatMap((Function<Route, Mono<?>>) r -> {
           exchange.getAttributes().remove(GATEWAY_PREDICATE_ROUTE_ATTR);
           if (logger.isDebugEnabled()) {
           logger.debug("Mapping [" + getExchangeDesc(exchange) + "] to " + r);
            }
            exchange.getAttributes().put(GATEWAY_ROUTE_ATTR, r);
                return Mono.just(webHandler);
            }).switchIfEmpty(Mono.empty().then(Mono.fromRunnable(() -> {
                exchange.getAttributes().remove(GATEWAY_PREDICATE_ROUTE_ATTR);
                if (logger.isTraceEnabled()) {
                    logger.trace("No RouteDefinition found for [" +
                        getExchangeDesc(exchange) + "]");
                }
            })));
}
protected Mono<Route> lookupRoute(ServerWebExchange exchange) {
    return this.routeLocator
            .getRoutes()
            .concatMap(route -> Mono
                    .just(route)
                    .filterWhen(r -> {exchange.getAttributes().put(
                        GATEWAY_PREDICATE_ROUTE_ATTR, r.getId());
                        return r.getPredicate().apply(exchange);
                       })
                    .doOnError(e -> logger.error("Error applying predicate
                        for route: "+route.getId(), e))
                    .onErrorResume(e -> Mono.empty())
            )
            .next()
            .map(route -> {
                if (logger.isDebugEnabled()) {
                    logger.debug("Route matched: " + route.getId());
                }
                validateRoute(route, exchange);
                return route;
            });
```

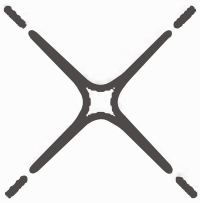
DispatcherHandler通过SimpleHandlerAdapter组件调用FilteringWebHandler模块的handler方法，FilteringWebHandler模块接着调用之前在容器中注册的所有Filter，处理完毕后返回Response，代码如下：

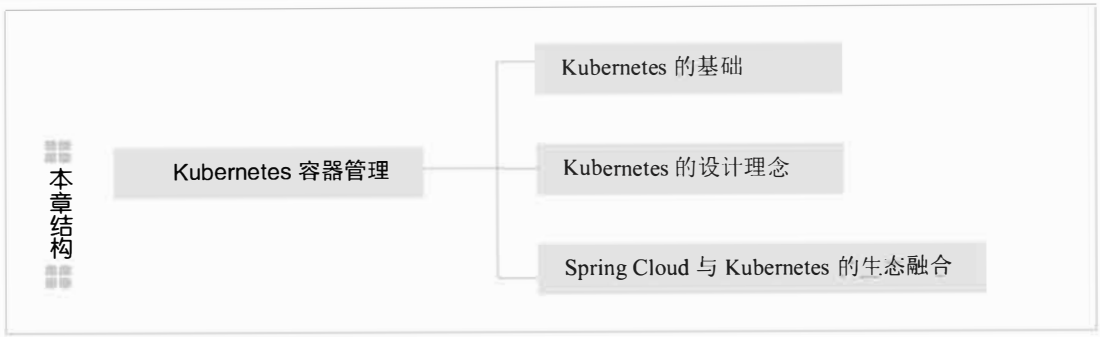
```
public class FilteringWebHandler extends WebHandlerDecorator {
    @Override
    public Mono<Void> handle(ServerWebExchange exchange) {
        return this.filters.length != 0 ?
            new DefaultWebFilterChain(getDelegate(),
            this.filters).filter(exchange) :super.handle(exchange);
    }
}
```

## 10.5 小结

构建响应式微服务可以获得异步、响应性、弹性、快速恢复、背压等系统特性，同时响应式微服务架构在资源占用、高并发、高吞吐、异步处理场景中具有更强的优势。目前响应式框架技术选型众多，如果将响应式编程应用到大规模生产系统中，则需要进行周密的调研，并对实际项目周期、人员经验、技术框架等因素进行综合权衡考虑，避免技术的复杂度问题成为业务发展过程中的瓶颈。

# Kubernetes容器管理

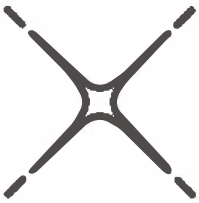


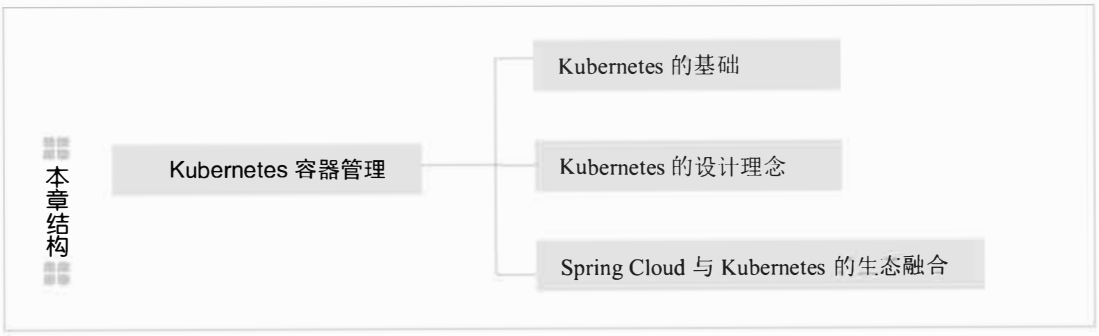



Kubernetes作为分布式容器编排及管理系统，本身采用了微服务的架构设计思想和理念。本章我们会简单介绍Kubernetes的基本概念和关键组件，同时讲解Kubernetes与Spring Cloud的生态融合。

第11章

# Kubernetes容器管理






Kubernetes作为分布式容器编排及管理系统，本身采用了微服务的架构设计思想和理念。本章我们会简单介绍Kubernetes的基本概念和关键组件，同时讲解Kubernetes与Spring Cloud的生态融合。

## 11.2 Kubernetes的设计理念

### 11.2.1 Kubernetes的设计原则

Kubernetes的核心是围绕应用进行大规模的容器扩展和管理，实现自动化部署，同时支持资源调度、集群管理、服务注册与发现、服务编排、服务自愈、安全管理等功能。Kubernetes本身基于微服务架构的设计思想，我们可以从中借鉴其分布式设计的理念和经验。

**分层原则**

Kubernetes作为分布式的云操作系统，其设计理念和功能很多都借鉴了Linux的设计思想，其分层（应用层、管理层、接口层、生态层）的架构设计就是其中的重要理念。

**面向API设计原则**

在Kubernetes平台中，API是管理操作的对象，API Server组件在整个Kubernetes分布式系统中处于统领地位。

**容错性设计原则**

Kubernetes保证每个模块都可以在出错后自动恢复，不会因为连接不到其他模块而自我崩溃。必要时，每个模块都可以优雅地进行服务降级。

**扩展性设计原则**

Kubernetes的控制机制保证了分布式系统下容器运行的稳定性和高可靠性。作为云原生操作系统，Kubernetes开放了容器运行时接口（CRI）、容器网络接口（CNI）和容器存储接口（CSI）。

### 11.2.2 Kubernetes与微服务

我们将Kubernetes的核心组件和微服务架构的设计理念加以对比。

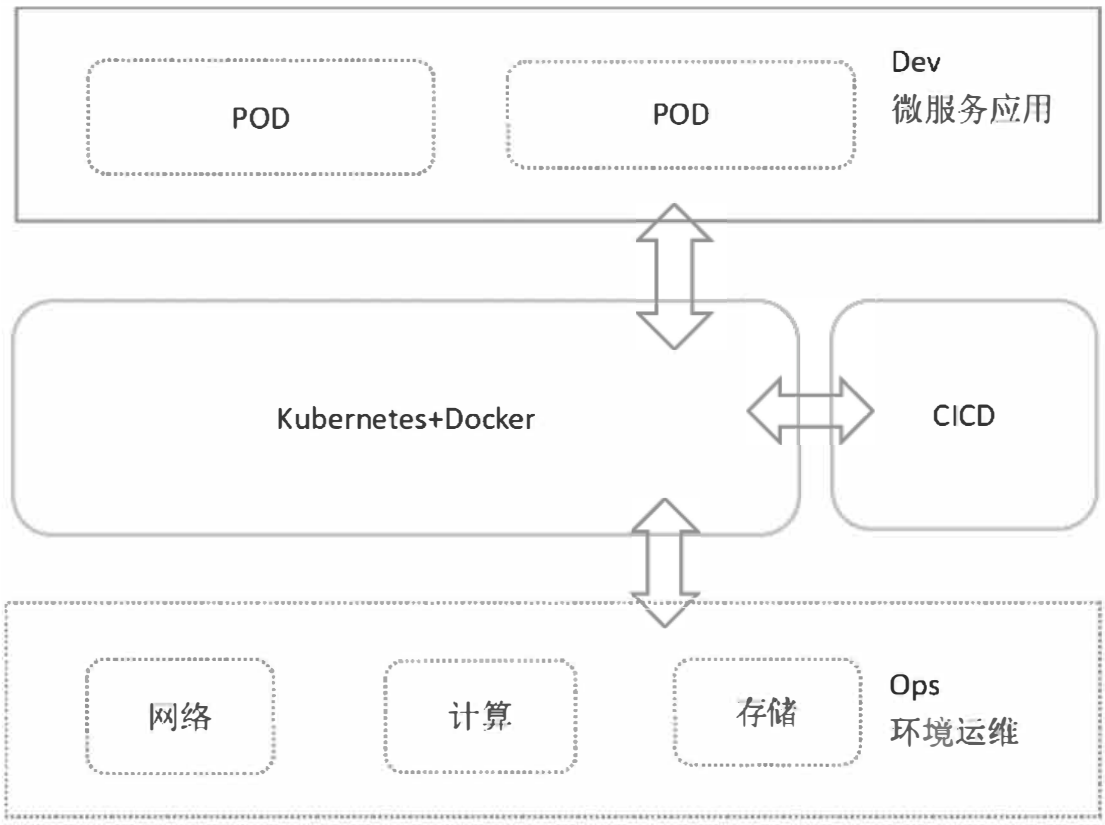
- Kubernetes中的核心概念是Service，表示一个业务系统的“微服务”概念。每个具体的Service都对应一组独立、隔离、业务功能单一的容器进程。
- Kubernetes中的API Server相当于微服务架构中的API网关，实现了前后端模块之间的解耦。
- Kubernetes使用DNS作为服务注册发现机制。相比微服务架构中专有的服务注册与发现组件，DNS具备更好的异构系统的监控性。
- Kubernetes中的众多组件和资源对象都是功能独立的。每个组件都有自己独立的功能，这符合微服务架构独立功能、独立优化、独立演进的思想和理念。
- Kubernetes作为一个容器运行编排管理引擎，始终在技术栈和语言倾向上保持中立，符合微服务架构的技术多样性的特性。
- Kubernetes包含一组可扩展接口，这让它具备了更好的可扩展性。

### 11.2.3 Kubernetes与DevOps

Kubernetes持续交付的优势是让开发者可以将部署工作提前到开发阶段，通过容器技术统一交付代码和环境配置，这样就省去了开发与运维信息同步的时间，避免了信息传递过程中的信息丢失问题，保证了资源与环境的一致性。

另外，在服务的运行过程中，Kubernetes可以结合DevOps来保证服务的高可用、弹性伸缩、负载均衡、灰度发布等平台功能。

下面是Kubernetes、Docker和底层计算资源的关系图。Kubernetes一方面作为连接底层物理资源与应用服务的桥梁，屏蔽应用服务与底层物理环境的直接交互。另一方面，借助DevOps持续开发、持续集成工具，微服务以容器的形态运行在Kubernetes私有云平台上。




## 11.3 Spring Cloud与Kubernetes的生态融合

Spring Cloud和Kubernetes都是微服务运行平台，经常被人们拿来做比较，然而二者所关注的对象和解决的问题还是存在着本质差异的。本节我们从它们各自的功能特性出发，介绍它们与微服务的关系。

### 11.3.1 Spring Cloud与Kubernetes各自的优劣势

微服务综合了软件技术、架构风格、组织、流程管理等软件工程的方方面面，而Spring Cloud和Kubernetes平台都从技术的角度解决微服务架构所关注的问题。

微服务架构不仅提供一套可供评估的软件构建原则及架构风格，还提供一个平台或者工具来显著降低应用转型微服务的难度。下面我们就从微服务关注的这些焦点出发，分别看下Spring Cloud和Kubernetes的优劣势。

**Spring Cloud的优劣势**

优势

- Spring平台提供统一的编程模型，Spring Boot快速创建应用的能力可以显著提高开发者的微服务开发效率。
- Spring生态有成熟的、覆盖大多数运行时软件的库可供选择。它能提供更多的特性、更强的控制，以及更好的语言一致性选项。
- 不同的Spring Cloud库可以很好地整合在一起。例如，Feign客户端可以使用Hystrix作为熔断器，使用Ribbon作为请求负载均衡器。
- 使用Spring Boot和Spring Cloud体系微服务框架，开发者有更强的控制和定制化能力，开发人员可以根据不同业务场景和使用习惯来进行定制化开发，这个决定权掌握在开发者手中。

劣势

- Spring Cloud最主要的缺点是它只针对Java语言。微服务强调技术栈的多样性。Netflix Prana项目实现了SideCar模式，它试图屏蔽开发者接入Netflix基础设施的语言性差异，提供客户端库基于标准的HTTP协议，使那些非JVM语言编写的应用也可以存在于Netflix OSS系统中，但是这种方式显然不够优雅。
- Java开发者需要关注非常多业务功能以外的技术事项。每个微服务都需要运行各种客户端来获得配置恢复、服务发现、负载均衡等功能。除了实现所有的功能性服务，Java开发者还需要投入额外的精力来构建和管理一个通用的微服务平台。
- 在一个完整的微服务项目中，开发者往往需要依赖Spring Cloud平台组件，还需要考虑自动化部署、调度、资源管理、进程隔离、自愈、构建流水线等平台功能。而这些能力除了需要第三方软件的支持，还需要有相应的运行时技术保障。

**Kubernetes的优劣势**

优势

- Kubernetes是语言无关的容器管理平台，能够兼容云原生应用和传统的Web应用。它提供的服务包含配置管理、服务发现、负载均衡、度量收集和日志聚合，而这些平台功能对应用

不存在侵入性。这使得组织可以只提供一个平台，供多个使用不同技术栈的应用项目使用。

- 相比Spring Cloud平台，Kubernetes实现了更广阔的MSA（Micro Service Architect，微服务框架）概念集合。除了提供运行时服务，Kubernetes也提供环境变量、设置资源限制、RBAC、管理应用生命周期、自动伸缩、自愈等特性。

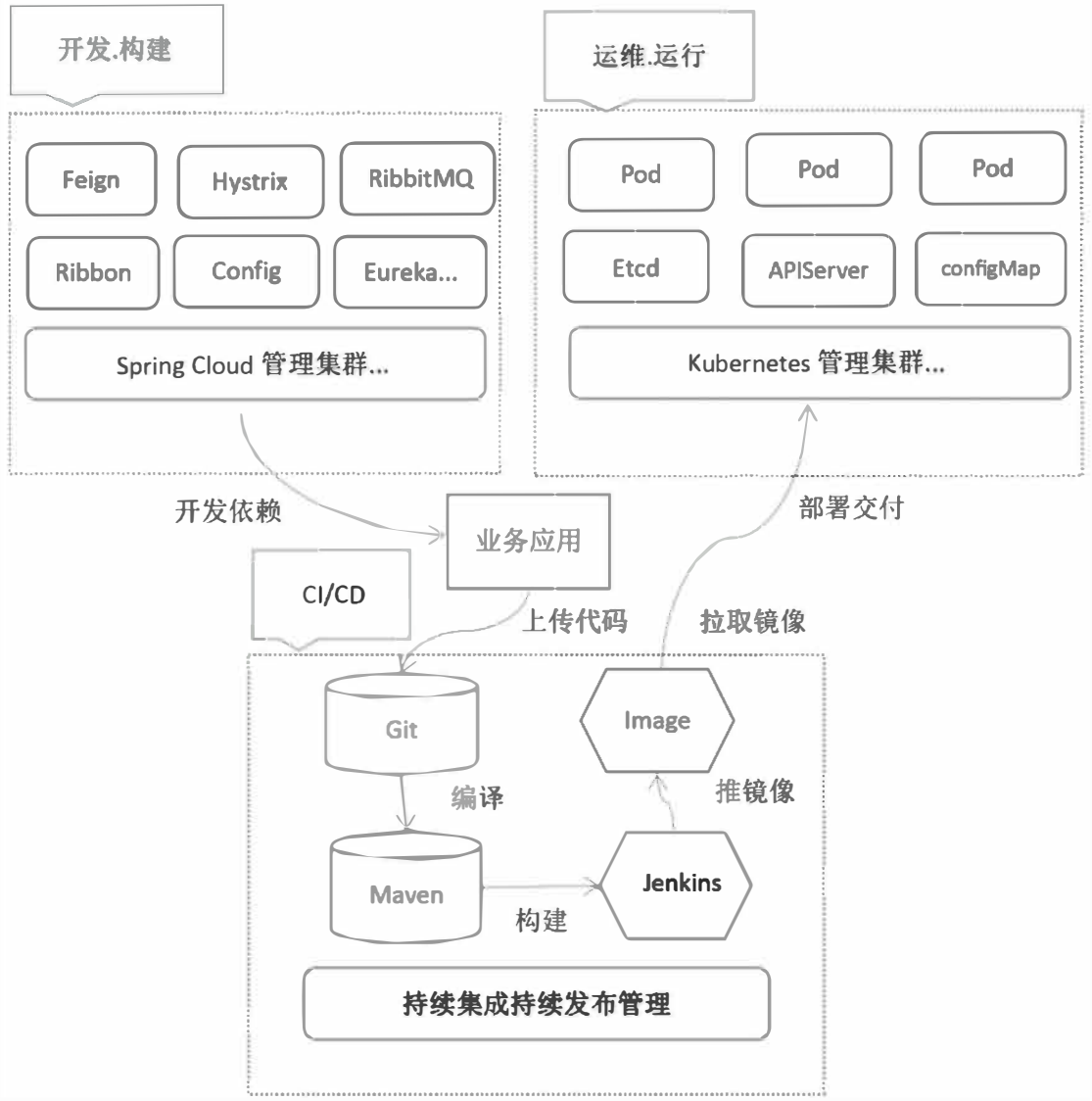
劣势

- Kubernetes是兼容多种语言的，因此它的服务和原语是通用的，没有针对不同的平台做优化，缺少灵活性。例如，配置是通过环境变量或者挂载文件系统传递给应用的。它没有Spring Cloud配置提供的那样精妙的配置更新能力。
- Kubernetes不是一个针对开发者的平台。它的目的是供具有DevOps思想的IT人员或运维人员使用。因此，应用开发者需要学习很多新的概念，以及新的解决问题的方式思路。不管使用MiniKube来部署一个Kubernetes开发实例多么容易，手工安装一个高可用的Kubernetes集群还是有明显的操作成本的。
- Kubernetes在使用过程中，相比Spring Cloud技术平台，从使用体验上来说，更像一个黑盒。当出现技术问题时，调试和跟踪过程都对开发人员不透明，无法做定制化的绑定或者更改，存在一定的技术壁垒。

从上面两者的优劣势对比来看，两个平台都有各自的优势和对微服务不同的关注点和着力点。Spring Cloud相对容易上手，对开发者友好，但是完全掌握平台需要一定的技术积累和实践，才能游刃有余；而Kubernetes是对DevOps友好的，有着陡峭的学习曲线，同时包含了更广泛的微服务概念。

### 11.3.2 Spring Cloud与Kubernetes的融合

结合上述对Spring Cloud和Kubernetes的优劣势分析，我们可以融合它们各自的优势，搭建出适合公司的微服务平台。下图的技术栈和构建流程可以作为参考。




使用Spring Boot，可以帮助业务应用快速开发、简化应用的启动和加载。通过Spring Cloud生态，可以为微服务平台提供服务的注册与发现、配置管理、事件驱动框架、消息队列、安全认证管理、容错管理、负载均衡、健康监测等功能。

Kubernetes平台结合DevOps实践方法论，借助第三方的Jenkins、Maven等工具实现自动打包、构建、上传部署交付物到容器仓库，通过Kubernetes的Yaml文件，可以定义部署交付物在容器集群环境下的集群配置。

### 11.3.3 Spring Cloud Kubernetes项目

Spring Cloud Kubernetes（GitHub开源项目）是Spring Cloud官方提供的一个通用服务接口实现，用来促成Kubernetes原生环境下运行的Spring Cloud或者Spring Boot应用更好地相互集成。

Spring Cloud Kubernetes提供了如下4个核心特性。

- Kubernetes生态意识识别。
- Kubernetes环境下Discovery Client实现服务发现。

● PropertySource使用ConfigMap实现配置加载管理。

● Ribbon在Kubernetes下的发现。

**Kubernetes生态意识识别**

从开发者的使用角度来看，Spring Boot应用程序启动和调试无须在Kubernetes中部署，因为Spring Cloud Kubernetes项目代码依赖Fabric8 Kubernetes Java客户端，它可以使用HTTP协议与Kubernetes Server的Rest API进行通信。

● Kubernetes配置文件自动配置

当应用程序在Kubernetes中作为Pod运行时，名为Kubernetes的Spring配置文件将自动被激活。它可以自定义配置，对Kubernetes平台中部署Spring Boot应用程序按需加载对应的Beans（根据不同的测试、开发、生产配置文件）。

● Istio 注 意识

当应用程序类的路径中包含spring-cloud-kubernetes-istio模块时，相关模块的配置文件将被添加到应用程序中。然后可以在Beans和@Configuration类中使用spring@Profile（"Istio"）注释。这时Spring应用将包含一个客户端模块（Istio-Client），可以通过这个Istio生态意识模块与Istio提供的API进行交互。

**Discovery Client实现服务发现**

该项目提供了Kubernetes的Discovery Client的客户端实现。通过此客户端可以按名称查询Kubernetes端点。Kubernetes API服务器通常将服务公开为代表HTTP和HTTPs地址端点的集合，客户端可以从作为Pod运行的Spring Boot应用程序进行访问。Spring Cloud Kubernetes Ribbon项目可以使用此功能来获取服务端点列表。Maven依赖如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-Kubernetes</artifactId>
</dependency>
```

如果需要启用Discovery Client的加载，请将@EnableDiscoveryClient添加到相应的配置或应用程序类中。

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
      SpringApplication.run(Application.class,  args);
  }
}
```

**PropertySource使用ConfigMap实现配置加载管理**

Kubernetes提供了一个ConfigMap资源，用于以键值对或嵌入式的application.properties或application.yaml文件的形式来外部化要传递给应用程序的参数。在Spring Cloud Kubernetes配置

项目中，Kubernetes ConfigMap实例可以在应用中观察到ConfigMap实例中检测到的变化，并装配Beans或Spring上下文。这个组件的功能与Spring Cloud Config配置中心的功能类似，只不过配置信息源来自Kubernetes的ConfigMap。

找到的所有匹配的ConfigMap都将按以下方式处理。

- 应用单个配置属性。
- 将名为application.yaml或者application.properties的任何属性的内容都用作属性文件。

假设我们有一个名为demo的Spring Boot应用程序，使用以下属性读取其线程池配置。

```
pool.size.core
pool.size.maximum
```

可以将其外部化为Yaml格式的配置映射。

```
kind: ConfigMap
apiVersion: v1
metadata:
    name: demo
data:
    pool.size.core: 1
    pool.size.max: 16
```

**Ribbon在Kubernetes下的服务发现**

Spring Cloud调用微服务的Ribbon组件实现客户端的负载均衡功能，以便自动发现它可以在哪个端点到达给定服务。该机制已在Spring开源项目 注 中实现，Kubernetes客户端可以自动填充Ribbon-ServerList，其中包含有关此类端点的信息。该实现是以下启动器的一部分，可以通过将其依赖项添加到Maven文件来实现该依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-Kubernetes-ribbon</artifactId>
    <version>${latest.version}</version>
</dependency>
```

填充端点列表后，通过匹配Ribbon Client注解中定义的服务名称，Kubernetes客户端搜索位于当前名称空间或项目中的已注册端点。

spring-cloud-starter-kubernetes-ribbon模块可以从spring.factories文件中找到自动配置类。

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.cloud.kubernetes.ribbon.RibbonKubernetesAutoConfiguration
```

自动配置类RibbonKubernetesAutoConfiguration的源码如下。

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties
@ConditionalOnBean(SpringClientFactory.class)
@ConditionalOnProperty(value = "spring.cloud.kubernetes.ribbon.enabled",matchIfMissing =
true)
@AutoConfigureAfter(RibbonAutoConfiguration.class)
@RibbonClients(defaultConfiguration = KubernetesRibbonClientConfiguration.class)
    public class RibbonKubernetesAutoConfiguration {
}
```

KubernetesRibbonClientConfiguration是使用@RibbonClients注解导入的配置类，也就是通过ImportBeanDefinitionRegistrar注册的，源码如下。

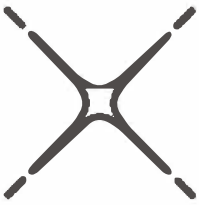
```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(KubernetesRibbonProperties.class)
public class KubernetesRibbonClientConfiguration {
@Bean
@ConditionalOnMissingBean
public ServerList<?> ribbonServerList(KubernetesClient client, IClientConfig config,
    KubernetesRibbonProperties properties) {
    KubernetesServerList serverList;
    if (properties.getMode() == KubernetesRibbonMode.SERVICE) {
        serverList = new KubernetesServicesServerList(client, properties);
    }
     else {
        serverList = new KubernetesEndpointsServerList(client, properties);
    }
    serverList.initWithNiwsConfig(config);
    return serverList;
    }

}
```

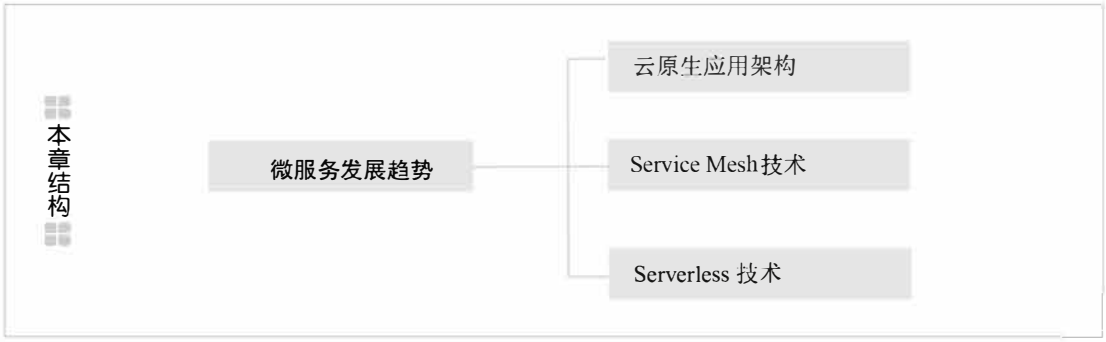
## 11.4 小结

Kubernetes作为容器调度运行平台，保证了微服务的弹性、负载、语言无关、扩缩容等强大的DevOps能力，而Spring Cloud框架提供的开发者经验是面向开发人员友好的微服务平台，两个平台都有各自的强项。在微服务实践过程中，我们需要集成它们各自的优势，最终诉求是满足业务场景、解决复杂问题域、提高开发效能、提升服务的交付效率。

第12章

# 微服务发展趋势






随着Docker技术的普及和Kubernetes在互联网公司的大量部署与使用，微服务架构正在围绕应用如何易于开发交付、减少资源消耗、无侵入治理等方面进行变革和演进。本章我们将讲解云原生架构、Service Mesh技术、无服务器架构（Serverless）技术。

## 12.1 云原生应用架构

云原生应用架构的3个特征包括：容器化、微服务、DevOps。通俗地讲，就是将现代应用基于微服务架构原则（云原生12因子）构建，使用DevOps开发运维一体化的动态管理机制，将微服务自动化部署在私有云或者公有云。

## 12.1.1 云原生应用架构进阶

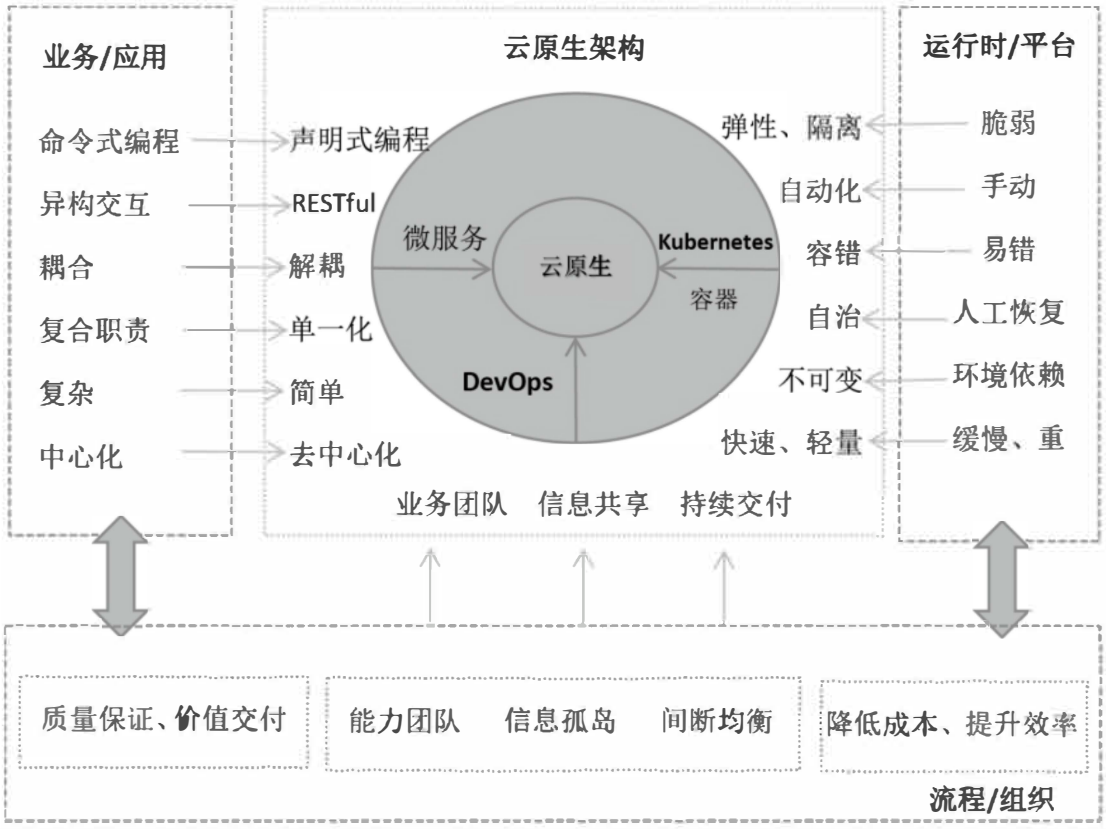
Pivotal是云原生应用的提出者，推出了Pivotal Cloud Foundry云原生应用平台和Spring开源微服务框架。近几年，随着云原生生态的不断发展壮大，Google主导成立的云原生计算基金会对云原生做了重新定义：

> 云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

CNCF对云原生的定义从Cloud和Native两个方面进一步阐述。

- 一方面，强调Cloud代表着应用的运行时，运行环境基于私有云、公有云或者混合云等，有别于传统的数据中心机房环境。
- 另一方面，Native应用是适合云环境的微服务程序，这些微服务程序更加适合运行在云端环境。

下面我们从应用、平台、组织流程等不同视角来看云原生应用架构的演进过程，以及云原生架构相比传统应用软件开发模式的组织特征和架构特性，示意图如下。




**从云平台搭建开始**

云平台相比传统的IT运维方式具备成本低、弹性、可靠性、便捷性等特性，而这些显而易见的优势都成为企业转型使用云平台的重要因素。

**业务微服务化转型**

企业进行云原生架构转型的核心动力是快速响应业务的需求和变化。传统的单体架构成为业务发展的羁绊，微服务给应用开发及部署带来了极大的灵活性和可扩展性。

**持续的价值交付**

持续交付需要基于DevOps方法论，结合持续集成和持续部署（CI/CD）过程完成应用持续的价值交付，从而使业务能从云原生技术架构中得到持续的价值收益。

## 12.1.2 Java的云原生应用优化

云原生应用围绕性能（Performance）、稳定性（Stability）、安全（Security）、扩展性（Scalability）等特性持续迭代演进。

在企业应用开发领域，使用Spring Boot为代表的开发框架作为云原生应用开发框架，依然处于主流地位，但是同样存在诸多劣势。

我们可以从以下几个方面考虑优化应用性能。

**编译速度优化**

Spring Boot在2.3.0.M1版本中对默认的编译工具进行了重大更改，使用Gradle而非Maven作为构建项目的主要代码管理工具。Spring Boot团队考虑切换到Gradle的主要原因是减少构建项目所需的时间，提升编译的效率。

相比Maven，Gradle在依赖管理、多模块构建、插件机制等多方面特性都有更加显著的优势。Gradle使用XML方式进行配置，把“约定大于配置”的设计理念进一步发扬光大，最重要的是并行化的编译速度有显著的提升。

**镜像瘦身**

交付物打包体积的大小直接影响镜像的分发和传输速度，通过对Docker镜像瘦身，可以显著提升构建交付物的效率。

分阶段构建则通过将构建环境和运行环境分离，减少上述构建产生的镜像冗余问题。下面列举了使用两阶段进行打包构建的方式。

```
# First stage: complete build environment
FROM maven:3.5.0-jdk-8-alpine AS builder
# add pom.xml and source code
ADD ./pom.xml pom.xml
ADD ./src src/
# package jar
RUN mvn clean package
# Second stage: minimal Runtime environment
From openjdk:8-jre-alpine
# copy jar from the first stage
COPY --from=builder target/msb-1.0.jar msb.jar
# run jar
CMD ["java", "-jar", "msb.jar"]
```

从上述DockerFile可以看到，我们将镜像构建分成两个阶段。

- 构建（Build）阶段依然采用JDK作为基础镜像，并利用Maven进行应用构建。
- 发布镜像阶段，我们采用JRE作为基础镜像，并从“Build”镜像中直接复制出生成的打包文件。在发布镜像阶段，不包含任何编译时的依赖，减少了镜像体积。

**提升启动速度**

JVM 9引入了AOT编译方式，它会将JVM编译结果保存在SCC中，在后续JVM启动中可以直接重用。与启动时进行的JIT编译相比，从SCC加载预编译的实现要快得多，而且消耗的资源更少，启动时间也得到明显改善。使用方式就是在OpenJDK的启动参数中增加参数配置：–Xshareclasses开启SCC，–Xquickstart开启AOT。从测试结果来看，使用OpenJDK的SCC和AOT特性启动速度提高了50%；而JVM资源占用也减少了400MB左右。

**Java云原生框架探寻**

Java云原生框架在社区中出现了一些技术框架的探索，其中比较典型的代表是Quarkus和Micronaut。二者都可以运行在GraalVM上。GraalVM使AOT编译成为可能，将字节码转换为本地机器代码，从而产生可以本地执行的二进制文件。

## 12.2 Service Mesh技术

服务啮合（Service Mesh）是一种为了保证“从服务到服务”的安全、快速和可靠的通信而产生的基础架构层。在云原生架构下，服务啮合层的提出，可以帮助开发者将服务的交互通信问题与微服务内部的业务问题隔离开来，使之专注于各自的领域。

## 12.2.1 微服务的SideCar模式

SideCar的本意是边车，它在客户端和服务端之间增加了一个服务代理，而这个代理是以应用组件的模式与微服务应用一起打包部署在容器或者单独进程中的。

边车模式类似连接到摩托车的边车，起到辅助和支持的作用。这种模式通过增加了边车（SideCar），实现边车组件完成一些非功能公共服务需求（如服务发现、熔断）。它的最大好处就是可以集成异构系统。

SideCar模式并没有在微服务架构中大面积应用，因为这种模式仅仅作为一种过渡，或者说解决了部分异构系统的服务发现、服务负载等局部问题，但它的局限性在于本质上它还是为特定的基础设施而设计的，无法满足兼容性。

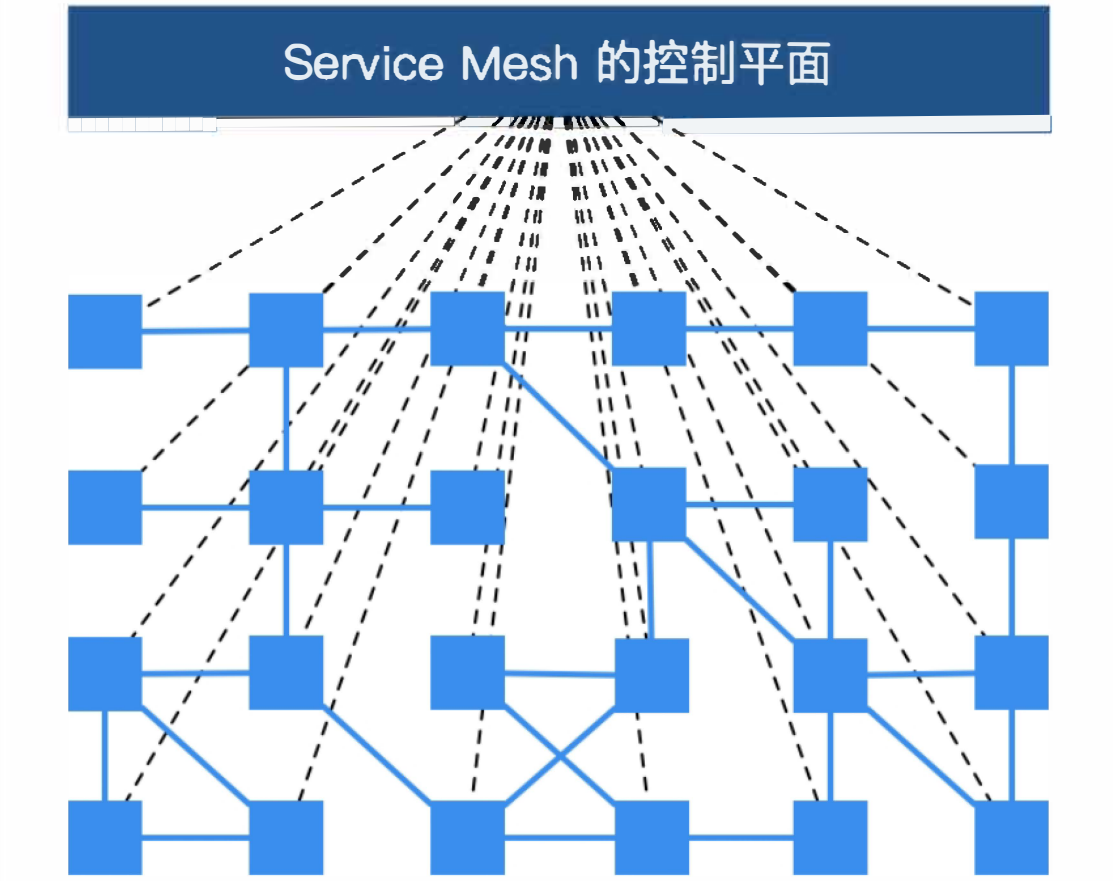
## 12.2.2 Service Mesh的技术前景

Service Mesh解决了SideCar模式的根本问题和局限性。

- Service Mesh是一种网络模型和基础设置，类似TCP/IP通信协议，服务之间通信使用Service Mesh，不再像SideCar被视为一个单独的组件，而是强调网络。
- Service Mesh作为通用协议的控制组件，不仅关注数据面的消息转发，还增加了控制面，实现对流量的控制和管理。

**Service Mesh的概念**

Service Mesh可以总结为一个专门处理服务通信的基础设施层。它的职责是在由云原生应用组成服务的复杂拓扑结构下进行可靠的请求传送。一个典型的Service Mesh架构由服务代理（数据平面）和管理与配置这些代理的“控制平面”组成。Service Mesh的控制平面如下图所示。




**Service Mesh的优势**

相比传统的微服务框架，Service Mesh有以下优势。

- 服务网格将SDK的服务治理能力从业务中剥离出来，拆分为独立进程，通过拦截标准协议和对网络通信层的控制实现服务治理，实现服务治理与业务的分离和解耦。
- 传统的微服务架构存在技术栈、遗留系统改造等约束条件限制，而Service Mesh可以统一管理这些服务，减少了维护和改造的成本。
- 服务网格技术是一个专用的基础设施层，它可以更有效地处理服务的治理需求，如路由的灰度发布、安全认证、加密等特性。

**Service Mesh的局限**

Service Mesh不是“银弹”，它的局限与优势一样明显。Service Mesh本身也带来了更多的复杂性。

- 服务网格技术会极大地增加开发和运维的复杂性。
- 服务网格技术相对不易做定制化开发和扩展，缺少了开发的灵活性。
- 服务网格技术缺少业务特性支持，如分布式事务等特性还没有相应的解决方案。
- 服务网格技术会给业务带来一定的延迟开销，对于性能有极致要求的场景，这样的延迟在大流量场景下，还会带来用户体验的下降。
- 服务网格技术的运行能力将成为整个网络和所有应用的瓶颈。

## 12.3 Serverless技术

### 12.3.1 Serverless的模式

回顾软件技术发展的历史，我们会发现“抽象、分解、集成、复用”的主题贯穿始终，行业的每一次更新换代都伴随着更高程度的软件服务模式的抽象，并随之引起IT行业的商业模式革新。

- On-Premises（本地部署）：硬件裸机时代，机房所有的硬件、操作系统、容器、运行时环境、应用、函数等都需要自己管理。
- IaaS（基础设施即服务）：IaaS处于底层，服务商提供底层或物理层的基础设施资源（如数据中心、电源、服务器机房），客户自己部署应用程序等各种软件。
- CaaS（容器即服务）：CaaS服务的出现使我们不再需要自己维护操作系统层面的东西，只需要维护业务应用系统即可。
- PaaS（平台即服务）：PaaS处于中间层，服务商提供基础设施的底层服务、操作系统，客户自己控制上层的应用程序部署。
- FaaS（函数即服务）：FaaS提供一个平台，用户实现业务函数逻辑，无须构建、维护与开发和启动应用程序相关的基础架构。

**Serverless的提出**

Serverless可以理解为由开发者实现的服务端逻辑运行在无状态的计算容器中，它由事件触发，完全被第三方管理。Serverless架构可以帮助我们减少部署、提高扩展性，并减轻代码后面的基础设施的维护负担。

Serverless从广义上包含FaaS和BaaS（后端即服务）两方面。FaaS解决了应用本身的“无服务器”化，BaaS解决了应用依赖的第三方服务的“无服务器”化。当应用和其依赖的服务都实现了“无服务器”化时，这个应用才算是完整的Serverless应用。

Serverless模型使客户仅需要关心“业务代码”，设计好自己的业务模型，把代码部署到云服务中，就可以完成一系列复杂的部署、运维、监控操作。

### 12.3.2 Serverless的技术前景

**Serverless的使用场景**

- Serverless适用于应用后端事件触发场景，通过调用Serverless云函数，实现文件上传、简单的消息推送、定时器任务等功能。
- Serverless适用于应用的AI智能对话场景，Serverless最核心的竞争优势不仅仅是云函数，更重要的是服务商本身所提供的AI能力。
- Serverless提供云端安全认证服务，可以减少不必要的IT重复建设。
- Serverless提供大数据处理能力，最典型的就是物联网应用，解决实时海量大数据计算和存储问题。

**Knative的Serverless架构方案**

Knative是谷歌开源的一套Serverless架构方案。它提高了构建可在本地、云和第三方数据中心等地方运行的现代化的、以资源为中心且基于容器的应用的能力，以Kubernetes的CRD形式运行存在。

Knative专注于解决以容器为核心的Serverless应用的构建、部署和运行的问题，构建在Kubernetes和Istio的基础上，为构建和部署Serverless及基于事件驱动的应用程序提供了一致的标准模式。

Knative的目标是提供标准的Serverless编排框架，简化烦琐的构建、部署、应用运维、服务治理等步骤，逐渐成为主流的无服务平台和技术的标准。

## 12.4 总结

在前面的章节中，我们已经讨论了很多与微服务有关的话题，最后让我们快速回顾总结一下全书的核心内容。

**技术与价值**

从早期的SOA架构开始，微服务以一种区别于传统单体应用的细粒度服务实现方式出现，使用分布式架构技术解决软件面临的系统性问题，逐渐成为企业开发的主流架构模式。作为一种“基础架构”技术，微服务融合了云原生12因子、领域驱动设计、Docker、Spring Boot、Spring Cloud框架、康威定律、DevOps、CI/CD等众多架构理念和软件技术开发实践。

微服务作为一种优化资源组织的体系结构方法，其价值在于解决实际系统面临的复杂性问题。随着业务规模的扩大，单体应用的程序变得无法适应和管理，微服务架构通过把整体应用划分为小型、自治、独立的服务集群，来有效地提升整体业务系统的可管控性、可维护性、可扩展性。从软件工程的角度来看，微服务通过代码层面的解耦，有利于团队的并行化开发，提高应用系统的交付效率，缩短生产周期；微服务也让开发人员从庞大的单体架构中解脱，不必理解整体系统工程，只需要关注自己负责的业务模块；微服务之间可以通过定义轻量化、标准、规范、用户友好的API进行集成交互。

同时，微服务可以有效地分离业务与技术的复杂度，通过将重复的组件沉淀在微服务架构底座，结合基础设施的隔离运行机制及持续集成、持续交付工具，将与业务无关的共性技术问题抽离，这样开发人员可以专注于业务本身。相比单体架构，微服务在快速响应、弹性、容错、维持速度、可演性方面都有更加显而易见的优势。

然而，微服务不是“银弹”，将整体系统分解为微服务后带来的系统分散性问题，以及微服务对技术、人员、组织、流程管理带来的研发习惯、工作模式的改变，都是考验企业转型微服务架构的重要决策因素。总而言之，任何技术的采用都需要结合业务实际情况进行一番利弊权衡。正所谓“最合适的技术才是最有价值的技术”。如果你的应用只是短期、小众用户的项目，完全可以不考虑系统架构；而随着时间推移，当用户量增长、业务新增需求，而你的系统无法扩展，需要重构扩展时，技术架构对业务的价值影响和冲击可能会造成无法挽回的损失。

**业务与平台**

微服务的实践围绕业务和平台两个方向发展，二者不存在割裂的关系，在开发实践的优先级方面，应该根据公司的发展战略和业务发展阶段有不同的侧重。

大部分企业都基于业务驱动方式选择微服务，原因是随着业务发展到一定程度，积累了大量的单体应用，需要依靠微服务的拆分和领域分解，解决日益增长的业务复杂性问题。微服务在这个阶段主要围绕业务进行微服务化改造。以Spring Boot为代表的基底模式是业务转型微服务的主要构建模式，以DDD（领域驱动设计）思想为指导，结合业务逻辑、开发人员技术能力、组织结构、基础IT架构等多方面因素综合考虑落地实践微服务。

在微服务的治理平台建设方面，随着服务规模的扩大，以Spring Cloud为代表的技术生态为规模化服务提供注册中心、配置中心、微服务网关、容错管理、集成交互、服务链路监控、日志聚集管理等基础设施和能力。

对于微服务的运行时平台，容器目前仍然是微服务应用运行的最佳虚拟环境和载体。容器技术虽然本质上与微服务是独立的，并且有着不同的解决领域和目标，然而结合DevOps实践方法论，它们共同组成了云原生架构的底层技术平台。伴随着微服务规模的增长、发布和交付的频率加快，缺少了容器技术的支持，将使微服务的规模化发展难以维系。自动化CI/CD和以Kubernetes技术为代表的PaaS平台建设成为微服务自动化、规模化、平台化阶段的主要工作。

当然微服务平台化建设也受限于企业的组织结构、流程管理及业务发展等方面，单纯依靠技术投入无法达到预期效果。平台建设的关键需要和业务微服务化改造相辅相成，在开发、测试、交付整个软件流程过程中建立规范和标准、划分职责边界，促进业务领域构建与技术平台融合、达成共识，使微服务架构可以持续迭代演进发展。

**趋势与未来**

目前，微服务技术的发展趋势是对基础设施层的进一步沉淀，实现业务与微服务底层架构运行时的分离。以Service Mesh、Serverless技术为代表的事件驱动形式的微服务架构，将成为下一阶段微服务的发展趋势和重点关注领域。随着云原生平台的重塑，微服务将进一步演进，平台将使开发人员更加专注于业务逻辑实现，抽离的基础服务将极大地提升开发人员的生产力，围绕着微服务、云原生基础设施、演进式架构，这里既充满挑战，也将会迸发出更多的生机与机遇。