



Service Mesh 微服务架构设计

刘松超 著

ARCHITECTURE DESIGN OF SERVICE MESH AND MICRO-SERVICES

资深架构师撰写，从设计与工程化视角分析Service Mesh，伴随大量一线
实践案例的呈现。

涵盖微服务实施原则，Kubernetes的架构设计与实践，Service Mesh工程
化设计思想与发展趋势等。



更多IT书籍请关注：www.cmsblogs.cn

see more please visit: <https://homeofbook.com>

架构师书库

Service Mesh微服务架构设计

刘俊海 著

ISBN: 978-7-111-63684-7

本书纸版由机械工业出版社于2019年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

前言

为什么要写这本书

作为新一代微服务架构，Service Mesh技术有效地解决了当前微服务架构和治理过程中的痛点问题，一经推出便引起很大的反响，近两年持续成为架构领域的热点。特别是Google联合Lyft等公司推出的Istio，架构优雅、功能强大，迅速成为Service Mesh领域的明星项目。我非常看好Istio在微服务领域的价值，一直持续关注着这个项目，我发现在Service Mesh或者微服务技术领域，已有的书籍和资料大多关注具体语言栈和具体技术的使用，而真正聚焦架构设计方面的书则偏少，因此想从架构设计方面对Service Mesh进行深入剖析。

本书从微服务架构和治理角度出发，聚焦Service Mesh的架构设计，试图从微服务技术演进的视角，全面揭开Service Mesh技术神秘的面纱。

读者对象

- 业务架构师
- 业务开发和运维人员
- 云计算基础设施开发者、架构师
- 对微服务技术感兴趣的人员
- 对云原生架构感兴趣的人员

如何阅读本书

本书分为两篇，共计10章。

基础篇（第1~3章），本篇着重讲解微服务架构和治理，以及Service Mesh技术当前的现状。

第1章为微服务架构，聚焦微服务实施的时机、准备工作和具体实施等；

第2章为微服务治理，通过服务治理解决引入微服务后带来的一系列挑战；

第3章为Service Mesh概述，讲述为什么Service Mesh能够解决微服务治理中的痛点问题，以及Service Mesh的发展历程和当前现状。

架构篇（第4~10章），本篇深入剖析Istio/Envoy在架构设计层面的原理和实现，以及Service Mesh未来展望。

第4章详细分析Envoy的整体架构，并且就架构设计层面与Nginx进行全方位的对比分析；

第5章分析Istio的整体架构以及各个组件的功能和设计；

第6章和第7章分别从控制流与数据流的角度，分析请求的处理策略与配置以及在整个Service Mesh中的流向和处理；

第8章讨论Istio的服务治理，重点聚焦可观测性和链路治理；

第9章讨论如何将Service Mesh中的一些架构思想和设计运用到平常的工程架构中去。

第10章展望Service Mesh技术在云原生架构下的未来和发展。

其中，第4~7章为本书的重点章节，如果你没有充足的时间完成全书的阅读，可以选择阅读重点章节。如果你是有着一定经验的资深人员，本书会是一本不错的案头书。

勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。可以通过微信号cloudnative_techdev，或者邮箱junhai0909@qq.com联系到我。期待能够得到大家的真挚反馈，在技术之路上互勉共进。

致谢

感谢百度谢广军、孙晓，滴滴出行卢红波、王正克、姜泰旭、段俊伟、奚媛，好未来张国辉、陈雷、韩天峰在工作和技术上无微不至的指导，我的成长离不开各位的大力支持和栽培。

感谢ServiceMesher技术社区的全体同仁，特别是ServiceMesher技术社区的组织者，是他们通过大量技术布道，最早将Service Mesh技术引入国内，对国内Service Mesh技术发展做出了很大贡献。

特别致谢

最后，我要特别感谢太太刘敏和儿子，为写作这本书我牺牲了很多陪伴他们的时间，也正因为有他们的付出与支持，我才能坚持写下去。

同时，感谢我的父母和岳父母，有了他们的帮助和支持，我才有时间和精力去完成写作工作。

最后要重点感谢高婧雅编辑，得益于她的耐心审稿、宝贵的建议以及用心的修改，本书的质量才进一步得到提升。

谨以此书献给我最亲爱的家人，以及众多热爱微服务技术和Service Mesh技术的朋友们！

刘俊海

第一篇 基础篇

第1章 微服务架构

微服务在解决业务复杂性、架构可扩展性等问题的同时，也带来了很多技术上的挑战，本章会围绕微服务展开，讨论微服务的收益，微服务改造的时机，以及实际微服务实施相关的工作。如果读者对微服务本身已经比较了解，可以跳过或有选择地阅读感兴趣的章节。

1.1 为什么需要微服务

微服务架构是过去几年架构设计领域的热点，接下来就先讨论一下微服务是什么，以及微服务架构带来的收益。

1.1.1 传统单体服务的问题

业务发展初期，业务流量很小，对业务迭代效率的要求很高，使用单体服务开发灵活，部署简单。随着业务的快速发展，业务复杂度越来越高，同时业务开发人员也急剧增加，单体服务逐渐暴露出来一些问题，主要问题如下。

（1）开发效率低

业务逻辑都集中在单个服务中，所有的开发者都在一个项目里面进行修改，代码并入时经常会出现相互等待和代码并入冲突的现象，严重影响开发效率。另外，多人同时操作一份代码，出错的概率也会增加。

上线方面，同一时刻有多个特性需要上线观察，为了保证线上服务的稳定性，一般要求特性上线以灰度的方式进行，不同特性之间串行上线。业务迭代比较快时，多个特性同时需要上线，就需要排队顺序等待，这样就会严重影响业务的迭代效率。

（2）可维护性差

业务逻辑都集中在一个服务里面，单个开发者不太可能完全掌控整个业务，因此业务修改和问题追查定位都非常困难。由于很少有人能够对业务非常熟悉，增加新功能或者修改原有功能时更多像是在打补丁，导致代码质量和可维护性越来越差，各种功能耦合在一起，新人接手的时候都不知道从何下手。

（3）架构扩展性差

不同模块可能有不同的架构要求，单体服务很难照顾到大家的差异化需求，如果想针对某个模块进行新架构和新语言的调整与支持，也比较困难。

（4）部署不灵活

任何一处很小的改动，都需要对整个服务进行编译、部署和上线，这可能会影响整个系统的稳定性，同时很难对某个特定的模块进行单独的容量规划和部署设计。

（5）健壮性差

单体服务的所有模块都在一起，某个模块出现严重问题，会导致整个业务不可用。当业务代码规模很庞大时，系统的故障点会很多，严重影响业务的健壮性和可用性。

1.1.2 微服务的定义

微服务就是为了解决单体服务的上述问题而生的，按照微服务权威专家Martin Fowler的定义，微服务架构是将单个服务拆分成一系列小服务，这些小服务都拥有独立的进程，通过HTTP RESTful API之类的轻量级通信方式进行通信，而作为独立的业务服务，则可采用一些自动化部署机制独立部署，每个服务可以使用不同的开发语言和数据存储技术，实现去中心化的服务管理。

微服务架构的核心诉求是支撑业务敏捷开发和部署，因此微服务架构的本质是如何优雅地支持微服务的“拆分”和“组合”，如何进行合理的架构拆分，如何最大限度地减少微服务之间沟通的成本，这是微服务架构的关键所在。

1.1.3 微服务与康威定律

微服务作为一个新技术，虽然在最近几年才受到越来越多的重视和推崇，但很早之前就有充足的理论基础。微服务的很多理念在《康威定律》中被充分阐述过。康威定律指出，设计系统的组织，其产生的设计等同于组织之内、组织之间的沟通结构，康威定律对系统架构设计有重要的指导价值，核心要点如下。

- 1) 架构设计必须解决沟通问题，好的设计应该降低沟通复杂度。这里蕴含高内聚、低耦合、服务隔离等架构设计的一些思想。
- 2) 一个系统没法一次做完美，迭代升级应该是常态。这里蕴含持续集成、敏捷开发、自动部署、容错设计等思想。
- 3) 人员组织结构一般会与系统设计趋同，你想要什么样的系统就需要什么样的组织结构。
- 4) 大系统终究会拆解成小系统，“合久必分，分而治之”；互联网大多会变成大系统，微服务是架构发展的一个非常自然的演变状态。

康威定律完美诠释了微服务架构的必要性和合理性，人与人沟通复杂，当问题复杂时，需要拆分组织来解决沟通问题；组织结构影响设计，通过组织拆分来改善沟通，从而形成拆分的设计（微服务架构）；被拆分的子系统设计是内聚的，对外边界是简单明确的，如果它能降低沟通成本，设计就是合理的；不要指望一个系统是大而完美的，它一定是迭代出来的。

微服务不只是个技术问题，更多的是关于组织和团队的问题，系统架构和组织之间存在映射关系，如果组织结构不支持，则无法建立高效的系统架构，反之也是这样。

1.1.4 微服务的收益

互联网业务的两个显著特点是业务发展快和业务高并发，微服务在这两个方面均有很大的收益，通过有效支撑业务创新和高并发架构，微服务架构成为传统架构演进时的必然选择。

通过将单个服务拆分成多个微服务，多个微服务可以独立开发、独立测试、独立运维，不同团队可以并行开发，互不影响，可以有力地支撑业务的快速迭代，方便业务创新和试错。

互联网业务通过业务模式创新和运营模式创新，用户和流量变化很快，需要从架构层面支撑业务流量的伸缩变化。微服务架构将单个服务拆分为多个子服务，每个微服务均可以独立进行容量评估，非常灵活，可以很好地支撑业务的高并发需求。

1.2 微服务架构的挑战

微服务的开发模式和单体服务差异比较大，对设计、开发、测试、运维等研发流程的各个阶段都提出了新的挑战。

1.2.1 服务拆分

服务拆分是微服务架构过程中最重要的一步，关系到微服务架构的成败，糟糕的服务拆分会影响后续的服务化过程，严重时可能会导致大量的返工，甚至是推倒重来；好的服务化拆分，意味着服务化已经成功了一半。微服务拆分时会遇到很多问题和权衡，是一个开着飞机换引擎的过程，比如拆分粒度和拆分原则如何确定；服务拆分是否需要考虑后向兼容；拆分顺序是怎麼样的；服务拆分和业务当前迭代之间的关系怎么权衡等。

1.2.2 开发挑战

微服务背景下，微服务的定位是实现特定的业务功能，因此会有大量新增微服务的需求。业务写一个在线的服务，除了实现必需的功能外，还有众多非功能需求要考虑，常见的非功能需求有可扩展性（业务流量变大、业务特性复杂度增加时，业务可以方便扩展）、稳定性（可靠性、可用性）、兼容性（上下版本兼容）、健壮性（面对异常流量时，业务需要能够正确处理）、安全性、性能等，如此众多的非功能需求要考虑，给业务的快速迭代带来很多障碍。同时微服务众多，如果大家没有遵循一致的开发模式和通信模式，跨微服务的沟通成本很大，会给开发、测试、运维等环节带来很大的开销，因此微服务架构的第一个挑战是建立一套完善的服务标准化方案，将微服务的通信和服务治理进行标准化。微服务标准化具体包含命名、路由、RPC、IDL、服务治理等，它对于服务开发效率和稳定性提升有重要意义，并且可以有效提升服务的可维护性。

另外一个挑战是微服务分布式通信带来的复杂性，由于分布式通信的特点，需要考虑容错、数据一致性、超时、重试等话题，同时服务注册和发现、集群路由和负载均衡、服务调用链路质量的跟踪等，都是需要考虑的问题。

分布式事务的处理，单体服务下的事务处理，到了分布式环境下就变得更为复杂，如果根据具体的业务场景进行折中和权衡，是个很大的问题。

1.2.3 测试挑战

微服务架构下，整个系统有一系列承载明确功能的微服务，通过一定的微服务通信组成，微服务架构的特点对软件测试也提出了一些挑战。

（1）开箱即用、一键部署的集成环境

微服务架构下，服务和依赖很多，拓扑复杂，服务随时可能会有各种变更，如何快速构建简单易用的测试环境，是一个很大的挑战。

（2）测试场景和测试确定性

微服务架构下，依赖和网络的细微变化，就可能导致测试结果发生变化，给测试场景和测试用例的构造带来很多困难，如何增加测试的确定性，保障测试效率和效果，是摆在大家面前的一个重要问题。

（3）微服务相关的非功能测试

微服务架构对服务治理有很高的要求，微服务治理相关的非功能需求很多，如降级限流、超时熔断、容灾容错设计等，因此需要构建一套针对微服务非功能需求进行测试的基础设施，保证微服务治理相关的非功能需求的有效性，增加微服务的可用性和安全性。

（4）自动化测试

微服务背景下，服务个数变多，服务变更频繁，对自动化测试的要求也会更高，很多线下测试没有问题的服务，线上可能会出问题，如果能够构建一套生产环境可用的自动化测试基础设施，对微服务的测试效率和效果会有很大的帮助。

1.2.4 运维挑战

监控上，单体服务的监控比较简单，微服务背景下，一个服务变为多个服务，需要对多个服务分别进行监控，对多个服务之间的通信也需要监控，监控管理变得非常复杂。

部署上，微服务拓扑复杂，有太多服务需要维护和管理，并且服务变更很频繁，对服务的部署和运维提出很高的要求。

问题追查上，单体服务的请求在一个服务内完成，微服务的请求由多个微服务协作完成，每个微服务也部署多个节点，整个调用链路很长，业务出现问题时，如何在比较短的时间内快速找出具体问题所在，是个很大的挑战。

依赖管理上，微服务依赖众多，如果每个依赖发生故障都会影响系统的整体稳定性，系统就会变得非常脆弱，因此需要对服务的各种依赖进行有效管理，比如哪些是强依赖，哪些是弱依赖，依赖故障对整个系统的影响，这些问题都需要提前预估并制定相应的预案。

容量管理上，如何在细粒度的状态下，更有效地管理数量庞大的微服务，如何预估系统的容量，业务爆炸式增长或者通过运营手段导致业务流量突增时，如何快速扩容？如何在保障业务稳定发展的同时，控制成本支出。微服务的容量管理和容量规划是个系统工程，需要在效率和成本之间有很好平衡，需要有完善的基础设施支持。精细化的容量管理需要做到可视化、可量化、可优化，对运维提出了很高的要求。

微服务架构最大的挑战在运维上，微服务的复杂性给微服务运维带来了许多难题。如果不能很好地解决微服务的运维问题，微服务架构的优势就无法体现，微服务改造也很容易陷入泥潭，因此，解决微服务的运维挑战，构建一体化的自动化运维基础设施，是微服务架构的关键一环。

1.3 微服务化的具体时机

微服务拆分确实会带来很多实实在在的收益，但同时在开发、测试、运维等多个方面也带来了很多挑战。特别是在业务发展初期，团队人员不多，对微服务周边技术和基础设施的积累不够，贸然采取微服务架构，不仅无法带来预期的收益，还可能严重阻碍业务的快速迭代，严重时甚至可能变成一个灾难。特别是互联网领域，业务迭代效率是第一位的，单个服务开发、测试和部署都相对简单，遇到的技术难度相对较少，如果引入微服务架构，面对的技术问题成倍增加，在技术上就需要进行大的投入，这对于追求业务快速迭代的中小团队来说得不偿失的。正常的做法是微服务改造之前，平常多进行一些微服务架构相关的技术储备，当单体服务的痛点达到一定程度，已经影响业务的正常迭代效率，才需要考虑微服务改造。

架构和技术都是为了业务服务的，因为评估微服务拆分时机，也需要从业务角度出发，而不是从技术角度进行评估。如果当前架构出现如下影响业务快速发展的场景时，可以考虑进行微服务改造。

（1）频繁出现因为需求排队上线而延期的现象

单个服务下，为了保证线上服务的稳定性，同时更好地监控上线特性的业务效果，不少研发团队都会制定相应的制度，需要上线的人提前提出申请，串行排队上线，如果经常出现因为排队等待而影响到业务的快速迭代，就需要考虑进行微服务拆分了，微服务拆分后，大家可以独享服务的上线窗口，加快并行开发的效率。

（2）频繁出现代码提交冲突的现象

单体服务下，多人同时在修改同一服务的代码，随着业务越来越复杂，团队的人越来越多，大家平常都在自己的分支上进行开发，分支代码合入主线时，很容易出现merge冲突的现象，或者解决merge冲突时出现代码合入错误。这时就需要考虑是否可以通过微服务拆分提高并行开发的效率，同时减少出错的概率。

（3）频繁出现一个简单功能特性需要同时修改众多文件的场景

之前团队一个真实的例子，上线一个简单的计价需求，修改文件超过50个，结果实际上线的时候发现还有2处修改遗漏。根本原因是服务代码量太庞大，并且代码结构组织不好，各种时期的历史代码，以及一些临时添加的代码遍布各处，随着业务的长期迭代，业务逻辑耦合很深，相似的判断逻辑分散到项目的各个角落，很少有人能够掌控全部逻辑。由于不了解代码的全貌，每次修改都小心翼翼，一不小心就很容易出错。这种情况下可以考虑进行微服务改造了，不然不仅会影响业务的迭代效率，同时也给整个系统的稳定性带来诸多隐患。

微服务的一大收益是解决业务高并发时带来的问题。高并发场景下，随着业务的快速发展，流量变化很快，对业务的容量评估和快速扩容提出了很高的要求，采用微服务架构拆分业务，不同的业务场景可以独自进行容量评估和扩容，比较灵活，同时可以大大节约成本。因此不少同学认为业务高并发也是微服务的一个重要拆分时机，对此我有不同的看法。对于快速成长的业务，业务迭代效率永远是第一位的，其次才是业务稳定性，成本一般在业务发展到一定阶段，业务成熟并且模式比较固定的时候才会提到一定的高度，成本

问题一般情况下不足以成为微服务拆分的一个充分条件。

1.4 微服务化开展前的准备工作

微服务改造是个综合性的系统工程，涉及研发全流程的各个维度，因此在微服务实施前需要进行一些必要的准备工作，比如从团队、技术上进行一系列的储备，确保微服务实施可以稳步进行。下面重点从微服务开发框架、微服务标准化以及持续集成和发布这几个维度分析微服务开展前的一些准备工作。

1.4.1 微服务开发框架

微服务背景下，微服务通信越来越复杂，通信场景和通信方式越来越多。一个运行良好的服务，不仅需要完成必需的功能需求，还要关注性能、稳定性、健壮性、兼容性、扩展性等诸多方面，此外还有日志、监控、统计、安全、容灾、容错等。从上面的需求来看，写一个完善的服务需要考虑的因素太多，特别是在移动互联网这种需求多变、对项目开发迭代速度和开发效率要求特别高的场景下，工程师很难在很短的时间考虑这么全面，写出符合预期的高性能高可用服务。

绝大多数服务对非功能特性和服务治理的需求都有一定的相似性，因此如果能够将功能需求之外的部分从业务中剥离，通用的、业务无关的特性对业务人员透明，那么业务人员只需要将精力聚焦在业务逻辑上，就可以快速完成微服务的开发。

微服务开发框架就是为了实现上述需求而产生的，聚焦和业务逻辑关系不太大的非功能需求，提高微服务开发效率，同时更为重要的是，使用相同的服务框架，业务服务可以更容易地实现服务标准化。

通常情况下，一个成熟完善的服务治理框架，基本功能层面需要包含如下几个部分。

（1）业务服务的脚手架

业务服务的脚手架是指在不考虑和其他服务通信的情况下，如何快速搭建一个业务服务。微服务架构下，需要从头快速创建微服务的场景很多，很多服务，尤其是在线服务，除了业务相关的处理逻辑之外，服务启动、服务退出、服务请求处理等流程几乎类似，至少可以针对绝大多数场景，抽象出一些通用的模式来。同时业务服务都会使用一些通用的基础组件，如日志库、配置库等，这些基础组件每个服务都会需要，因此对组件的性能、稳定性都有很高的要求。

（2）微服务通信

微服务通信包含的东西很多，如网络模型、服务发现、流量路由、负载均衡、连接池管理等。

（3）集成公司内的其他基础设施

很多公司都会有一些自研或者在开源基础上二次开发的基础设施，比如各种中间件产品、配置中心、部署平台、监控平台等，服务开发框架很重要的一个作用是和这些内部基础设施进行集成，对微服务业务人员屏蔽这些基础设施的细节。特别是一些中大型公司基础设施体系在实现上差异很大，从而很难直接使用开源的微服务开发框架，一般采用自研或者在开源基础上进行二次开发。

（4）微服务治理支持

系统经过微服务改造后，在效率、稳定性等环节都会遇到不少新的挑战，需要完善的服务治理支持，这些服务治理一般也是以微服务开发框架为支撑或者抓手的。服务治理是个很大的话题，放在第2章中专门讨论，这里不再展开。

1.微服务框架的选型考虑

选择和评估一个微服务开发框架时，应该从稳定性、易用性、调试与运维友好性这几个维度来考量。

1) 稳定性和可靠性是第一位的，如果代码质量不高，经常出现一些不太好解释的现象，这样肯定会影响线上服务的可用性。

2) 易用性，首先API接口设计要非常简洁明确，没有什么歧义，使用起来非常方便，没有什么大的心智负担。对于一些配置项框架要设置最合理的默认值，如果一个框架有非常多的配置参数，并且都交给用户，美其名曰是给用户灵活性，其实是把复杂和负担留给用户，把简单留给自己，这种框架是极其不负责任的。

3) 调试与运维友好性常常被市面上的大多数微服务开发框架所忽略，很多框架经常号称自己有强大的性能表现，但很少有框架在调试和运维上下工夫，如何让使用框架的人在遇到问题时方便快捷地定位，如何支持业务自定义地添加调试和定位信息，这些都是框架需要考虑的问题。

4) 最后要考虑功能和性能，功能和性能要能够满足当前业务需求。把这个放在最后是因为绝大多数框架在功能和性能上均能够满足需求。

对于性能评估和性能测试，大家平常一般主要关注QPS和耗时这两块。需要强调的是，性能测试前需要确定性能测试的基准，比如99.9%的响应时间必须在10ms之内。后续所有的性能测试需要严格按照这个基准执行。

针对QPS需要关注框架在多核或多线程下的扩展性；针对耗时，不仅要关注平均耗时，更要观察耗时长尾分布，以及耗时长尾是否有放大效应。

2.微服务改造时的语言选型考虑

不少人在进行微服务改造时，经常会有这样的疑问和想法：是否有必要在微服务化的同时选择更合适的语言？比如之前是PHP，微服务改造时想用Golang进行重构，这样是否可行？每种语言背后都有一套完善的语言栈体系，语言栈切换过程中很容易踩到一些坑，同时微服务改造本身也是一个复杂度很高的过程，如果把微服务改造和语言切换这两个风险很高的事情放在一起进行，会导致整个微服务改造过程中的风险增加很多，当过程中出现问题时也很难排查和控制。因此除非有特别的原因，不建议在微服务改造的时候进行语言层面的整体切换和重构，当然个别相对独立且风险可控的子服务，可以考虑语言层面的重构。

3.协议层面的扩展和增强

Thrift是当前使用非常广泛的高性能RPC框架，它功能强大，使用起来很容易上手，我们可以在短时间内迅速搭建一个可用的Thrift在线服务，以很好地解决开放效率、兼容性、性能等问题。Thrift有着完善的RPC和多语言支持，但直接在生产环境中使用开源的Thrift框架还是会遇到不少问题，比如没有完善的集群管理和治理支持、RPC协议无法扩展、没有定位和诊断相关的可视化支持等。国内外不少使用Thrift的团队都会针对开源Thrift进行定制和二次开发，下面就以生产环境中Thrift服务治理的一些实战经验为例，详细讨论对微服务开发框架的整体思考，以及Thrift协议层面的扩展和增强。

Thrift服务治理整体架构如图1-1所示，分为业务层、服务层、协议层和传输层4个部分，其中业务层是服务的业务逻辑实现，客户端和服务端通过IDL接口进行交互，对开源Thrift的扩展与增强通过服务层、协议层和传输层来完成。

服务层对开源Thrift进行两方面的增强：一个是客户端在集群管理方面的扩展，具体包括服务发现、流量路由、节点熔断和健康检查、负载均衡以及连接池管理等；还有一个是链路治理相关的插件实现，客户端和服务端分别提供通用的插件扩展机制，不同场景下进行相应的扩展和定制。

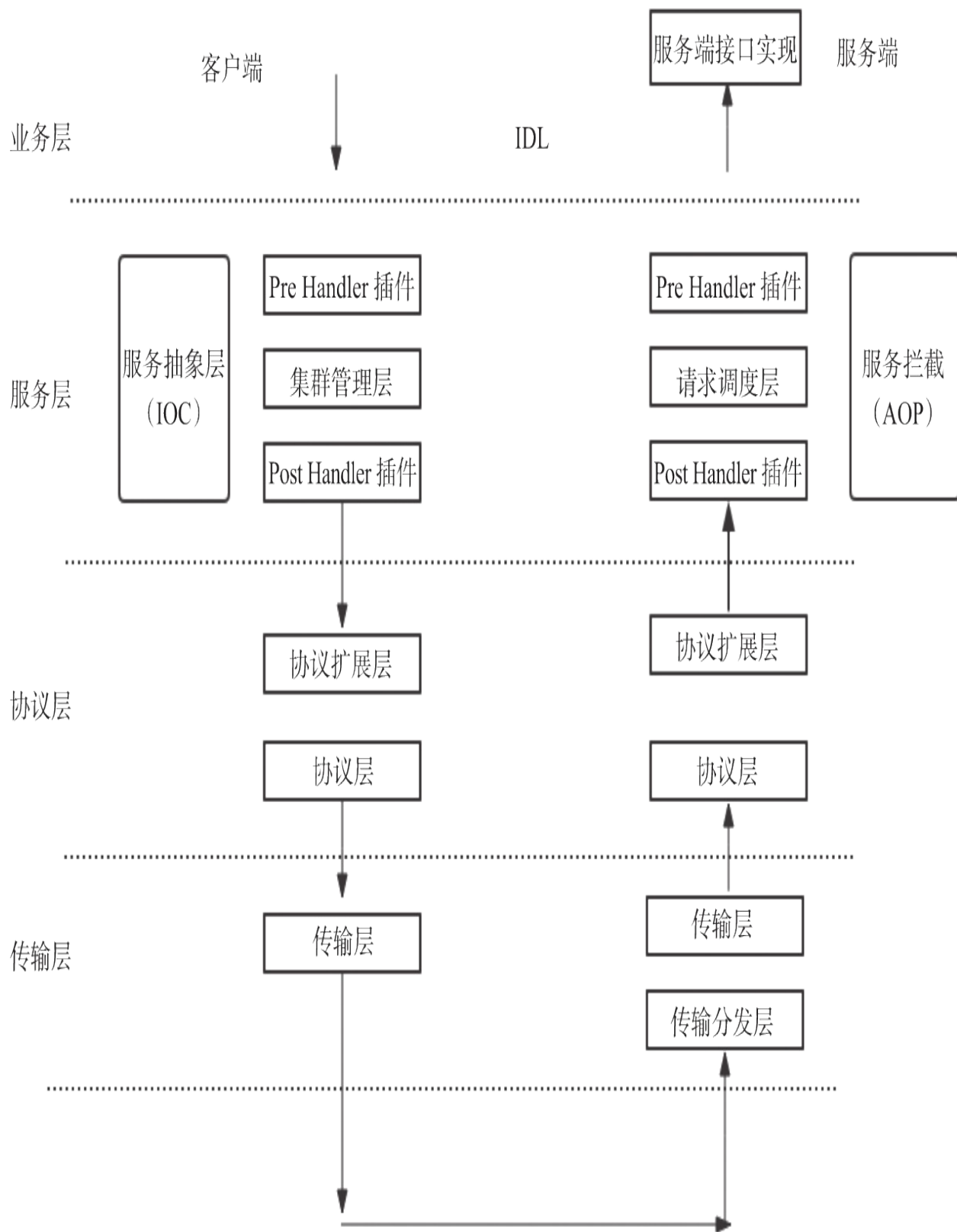


图1-1 Thrift服务治理整体架构

开源Thrift在扩展性方面有着非常出色的设计，协议、传输和调度层面均可以方便地进行扩展，但唯独在RPC协议层面没有提供扩展的入口，协议层定位是提供RPC协议扩展支持，通过协议扩展，可以方便地支持很多服务治理特性。

Thrift作为二进制协议，优点是传输效率和性能比较高，缺点是测试和调试诊断不太方便。而HTTP协议恰恰相反，HTTP有太多现成的工具可以使用，测试和调试非常方便。Thrift在传输层，增加了一个传输分发层的概念，使得Thrift服务同时可以接收HTTP协议消息，从而方便我们对Thrift服务进行测试。

RPC协议扩展是RPC框架非常重要的一个基础能力，基于扩展能力才能灵活支持不同场景下的扩展需求，比如以压缩特性为例，当某个基础Thrift服务当前不支持压缩特性时，如果想支持压缩特性，需要将传输层修改为支持压缩的传输层。Thrift压缩和非压缩的传输层之间不能兼容，需要客户端和服务端同时升级。对于有重点上游业务的基础服务来说，同时升级客户端和服务端是几乎不可能的事情。另外，为了支持分布式跟踪特性，需要在请求的整个链路中传递跟踪相关信息，如果框架协议层面无法支持，就需要每个业务均做出相应的支持，这不仅会影响业务迭代效率，同时因为每个业务实现方式各异，很难保证分布式跟踪方案的完备性和标准化。

开源Thrift在RPC协议支持上实现比较简单，支持name（rpc方法名）、message_type^[1]和seq_id 3个字段，并且没有针对RPC协议扩展进行预留设计，比如预定扩展字段等，导致后续要对RPC协议进行扩展支持特别困难。

实际使用的Thrift过程中，经常会碰到一些通用的服务治理需求，比如分布式跟踪、全链路压测、多环境流量调度支持等，这些通用的服务治理能力最好是通过RPC协议的方式下沉到框架层面，对业务透明，否则每个业务人员都需要感知这些特性，因此在RPC协议层面进行扩展支持和增强是Thrift使用过程中不可回避的问题，大体有如下扩展思路。

（1）协议层扩展

开源Thrift之所以无法原生支持扩展，主要是没有预留消息头长度之类的信息，协议层只要增加一个字段来标识协议层的总长度，消息解析时把不认识的跳过即可。因此可以针对每个协议层协议增加一个后向兼容的新协议，比如通过BinaryProtocolNew兼容BinaryProtocol，通过CompactProtocolNew兼容CompactProtocol。具体来说，BinaryProtocolNew根据version判断新老版本，如果为0x80010000就是老版本，按照官方的BinaryProtocol进行解析；否则是新版本，按照新版本进行解析。

新版本在协议层增加Length字段，表示编码后的协议层数据总长度。在有Length字段后，如果接收到新协议消息，可以把不认识的字段直接跳过，实现前向兼容。通过协议层面扩展RPC协议，新协议可以识别旧的协议，可以实现协议的前向兼容。这里最大的问题是旧协议无法针对新协议进行处理，没有办法实现协议的后向兼容。

（2）传输层扩展

Facebook内部使用的fbthrift采用的就是传输层扩展的方式，RPC消息发送前在业务请求前面加一个Header，Header里面可以添加一些自定义的RPC协议字段，如协议类型、压缩类型、是否支持加密与logid等；接收时先接收Header，根据Header中协议类型初始化相应的协议类和传输类，进行后续的报文解析。这种方式与协议层扩展相比，灵活性和通用性更好，但兼容性的问题依然存在，没有办法和开源的官方协议版本兼容。

（3）基于rpc name的扩展

Thrift RPC协议当前包括rpc_name、seq_id和msg_type 3个字段，这3个字段只有rpc_name是字符串类型，属于变长字段，可以考虑基于这个字段做一些文章。思路是将rpc_name扩展成JSON格式的RPC协议，示例如下：

```
rpc_name?trace_id=welcome&span_id=didi&compress_type=1
```

在开源Thrift的rpc_name字段基础上，加上JSON格式的KV信息，用于服务治理相关的扩展支持，同时为了减少对业务的影响，可以在Thrift编译器层面进行修改，Thrift编译器生成代码时，自动加上相关的支持。

和上面两种方式相比，该方法优雅地屏蔽了使用方式的差异，但仍然没有解决和官方版本的完全兼容问题。

（4）基于协议保留自动的扩展

为了提高RPC通信时的效率，Thrift在协议定义时为每个字段指定相应的字段序号，网络交互时不再传递字段名，而是统一传递字段序号，减少网络传输的数据量。但不清楚Thrift官方具体出于哪些考虑，当前字段序号不能设置为0，0作为保留字段。可以利用保留字段0进行RPC协议的扩展。

协议扩展信息的透传通过各语言SDK的方式进行，SDK需要提供相应的机制，方便业务设置和获取协议扩展信息，因此SDK需要提供业务和框架之间信息传递的方式。对于PHP、Python等语言来说，可以直接通过全局变量来完成；对于C++、Java等多线程语言来说，可以通过线程局部变量TLS的方式进行传递；对于Golang这种基于协程的语言，可以参考TLS，自行实现协程局部变量GLS，基于GLS进行数据透传。

基于协议保留字段的协议扩展见图1-2。

在客户端增加了协议装饰层，协议扩展信息通过TLS的方式统一维护，协议装饰层通过TLS获取扩展信息后，使用保留字段序号0，和业务请求一块打包，隐式传递到对端。

在服务端，流量拦截机制会在请求解析时判定是否有0号字段，如果有就将该字段解析出来，通过TLS机制设置后供业务查询使用。

通过协议扩展机制，可以方便地支持分布式跟踪等特性，服务治理相关的处理直接下沉到框架SDK中，对业务透明。而本节开头提到的压缩支持等特性，只需要在协议扩展中增加压缩标志，这样服务端可以同时兼容压缩和非压缩两种请求消息，实现业务友好的平滑兼容升级。

4.集群通信的“可靠性”

在没有完善的服务治理支持的情况下，当下游节点故障时，所有的上游服务均需要将故障节点摘除，如果不及时摘除，上游在服务访问时仍然会选择故障节点，会导致大量的访问失败。根本原因是上游服务不具备对下游故障的感知能力，没有及时将故障节点从可用节点中摘除，这是很多团队初期都会碰到的一个痛点问题。

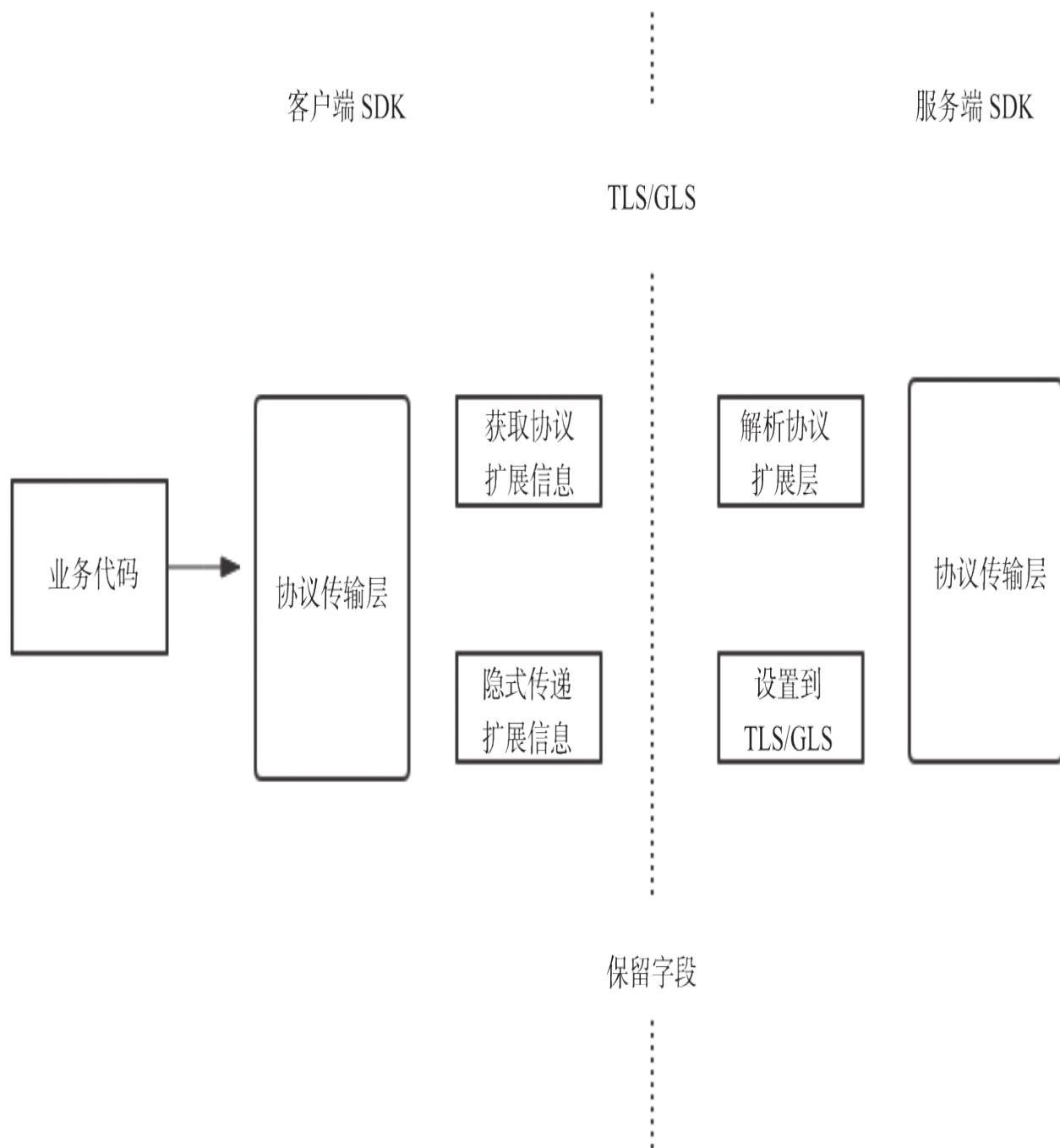


图1-2 Thrift协议优雅扩展方案

为了保证下游变更时上游无感知，调用下游服务时的SDK需要具备故障节点熔断和健康检查能力，上游每次访问下游服务时均记录服务访问的metric指标，最核心的指标就是成功还是失败。为了判断下游服务的健康状况，针对每个下游节点引入了一个健康度的概念。为了简化判断，每次访问失败时健康度加1，访问成功时健康度减1，每次访问下游节点前，判断该节点的健康度，如果超过一定的阈值，可以判断该节点为故障节点，从可用节点列表中摘除，这个过程就是熔断。这里有个细节，当判定节点为故障节点后，还需要判断服务当前故障节点是否已经超过一定的阈值，如果故障节点过多且可用节点过少，服

务的全部流量均会打到少量的可用节点上，这些节点可能会因为承载不了这么大的流量而雪崩。为了避免这种现象，当故障节点超过一定阈值时，不再将新的故障节点加入到故障节点列表，仍然放在可用列表中，负载均衡时可以承载一部分流量。当然这部分流量会访问失败，但通过分流这部分失败流量，可以避免集群雪崩导致的全部流量访问失败。

为了保证微服务集群通信的链路稳定和正常运行，下游节点熔断后还需要有相应的探测和恢复机制，也就是节点健康检查机制，一般有[被动探测](#)和[主动探测](#)两种方式。

被动探测是将故障节点设置为故障状态，冷却一段时间后重新将该节点加入到可用节点列表中使用。如果下游节点已经恢复，访问下游成功时，下游服务的健康度会逐渐恢复正常；如果下游没有恢复，访问下游仍然失败，会将该节点再次从可用节点列表中摘除，同时设置冷却时间，等待下一次探测。

和被动探测直接使用正常业务流量进行请求不同，为了避免对正常业务的影响，主动探测是使用额外的探测请求来判断下游节点的可用性。主动探测模式下，节点熔断后，每隔一段时间定时启动主动探测流程，并且和被动探测时直接将故障下游节点放回可用节点列表不同，主动探测返回成功标识后才会将节点放回到可用节点列表，如果探测失败则不进行任何处理，等待第一次的定时探测流程。和被动探测相比，在下游节点故障时主动探测不会导致额外的业务请求失败，对业务的可用性影响较小。但是使用主动探测有个前提，[健康检查时必须使用正确的协议来构造和下游故障节点的连接](#)，比如通过HTTP协议访问下游时，就需要通过构造HTTP请求对故障下游进行探测；通过Redis协议访问下游时，就需要通过构造Redis请求对故障下游进行探测。

此外，为了减少故障节点反复故障时访问失败，当将故障节点重新放回可用节点列表时，不会直接将该节点的健康度直接清零，而是维持在一定的水位上，比如下游服务的健康度阈值为10（连续超过10次访问失败时，判定下游节点为故障节点），如果将健康度重置为0，在故障节点不稳定性状态时，需要累积10次失败才能重新进入故障状态，因此可以设置为一个相对高的水位，比如为8，保证失败几次后可以快速进入故障状态，避免对业务产生影响。

从上面的分析和讨论来看，节点的熔断和健康检查机制比较复杂，涉及很多技术细节。业务发展初期，在服务访问SDK不太收敛的情况下，各个业务以及不同语言下分别增加熔断和健康检查逻辑是件有挑战的事情，对于一些有众多多语言上游服务的基础服务，为了保证下游故障或者变更时对上游节点透明，可以采用增加一个Proxy代理的思路，将节点熔断和健康检查等集群访问和服务治理均收敛到代理上，以保证下游基础服务的变更对上游服务透明。

[1] 消息类型，如请求消息、应答消息以及oneway消息等，oneway是不需要回复结果的RPC请求。

1.4.2 微服务标准化

1.接口标准化

接口标准化在微服务架构设计过程中发挥着重要的作用。在微服务的交互中，统一规范的接口标准可以大大减少沟通成本，提高开发和维护效率。

很多公司主流协议为HTTP协议和Protobuf/Thrift协议，HTTP协议开发、测试比较方便，而Protobuf/Thrift等二进制协议性能更好些。一般来说，和端交互比较多的业务服务使用HTTP协议较多，对性能有一定要求的后台基础服务偏向使用二进制协议。

以Thrift协议来说，Thrift协议本身有自己的一套IDL，通过IDL生成标准服务代码，包含数据通路和类型定义，相对规范；为了保持微服务接口层面的一致性，实现微服务接口标准化，HTTP协议可以参考Thrift协议的思路，增加IDL描述支持。HTTP服务的IDL化可以规范标准、降低人为操作的出错率、降低下游服务接入成本和维护成本。下游服务不需要再维护代码，只需要维护配置，这点对于尚未使用统一的服务治理框架的团队来说特别有益。下游服务的升级修改收敛于单纯的IDL文件，版本更好管理。

我们可以基于接口标准化，为接口增加相应的描述信息，并通过代码自动生成机制，将IDL文件直接生成接口文档，实现接口文档标准化。

为了减少业务对服务治理支持的开销，可以把服务治理相关的配置放到IDL文件，实现服务治理配置的代码化。

因此，IDL文件成为服务标准化中非常重要的一个载体，通过IDL文件可以实现接口标准化、文档标准化和服务治理标准化。

2.日志标准化

日志标准化是服务标准化的关键一环，各个微服务之间只有遵循相同的日志规范，才能方便后续的日志治理，针对Log、Trace等日志形态，均需要制定统一的规范。

日志输出的目的不仅是发现问题，好的日志可以帮助我们快速排查出问题的原因，进而采取相应的应对措施，因此日志规范需要聚焦如何更加方便快捷地追查问题，提高问题追查的效率。

3.透传通道标准化

为了对微服务进行精细化的治理和管控，很多服务治理特性均需要在同一请求处理的各个环节获取到请求相关的服务治理信息，因此系统需要能够支持服务治理相关的信息在请求处理的整个链路中透传，服务治理聚焦的都是非功能需求，这些需求业务服务其实是不需要关注的，因此需要有一定的机制来减少服务治理的非功能特性对业务的侵入，最大程度上实现业务透明的服务治理接入。

透传通道就是为了实现上述所说的服务治理透明接入，思路是将服务治理相关的非功能需求信息从业务中解耦出来，在所有微服务之间进行传递。

透传通道承载业务无关的服务治理信息传递，是业务微服务和服务治理基础设施的桥梁，关系到整个服务治理的效率和效果，因此需要有标准化的透传通道，实现通道数据传输的可靠、透明和可扩展。

1.4.3 持续集成与发布

微服务改造过程中，需要保证改造过程中的修改及时得到验证，因此需要在持续集成和发布方面进行提前准备，确保微服务改造能够正常迭代进行。

1.研发全流程平台

微服务研发全流程包括很多环节，比如代码分支管理、代码review、测试准入、提测、自动化测试、服务部署、服务变更等，每个环节都有相应的工作流，研发 workflow 平台将研发全流程中的工作流串在一起，形成一个完整的闭环。这不仅可以提供特性研发工作的质量和规范度，还可以基于完整闭环机制方便地对每个特性的线上效果进行复盘和总结，通过微服务数字化运营，提高业务的迭代效率。

2.灰度发布

微服务化在架构层面是一个非常大的改动，为了不影响线上业务的稳定性，需要采用灰度发布的方式，每一次发布均需要先进行小范围灰度，观察完全没有问题之后再放量发布。

灰度发布的维度可以根据业务的特点而定，常见的灰度维度有按照机器、按照流量百分比、按照城市或者人群等。

为了支持精细化灰度发布，微服务改造前需要对部署系统进行适配升级，保证微服务化过程的每一步都精确可控，将对线上服务的影响降到最低。

1.5 微服务实施

微服务改造过程中会面临很多挑战，接下来从服务拆分、服务通信以及服务稳定性设计这几个维度出发，讨论微服务实施过程中需要着重注意的问题。

1.5.1 微服务拆分

服务拆分是微服务改造的第一步，也是最为关键的一步，拆分是否合理，决定了整个微服务化过程的成败，因此，需要有科学的拆分原则和拆分方法，保证拆分过程有序进行。

1. 拆分原则

微服务拆分前，需要确定拆分的基本原则，微服务拆分的一个非常重要原则是无状态，无状态服务方便扩展和运维，所以无状态设计需要贯穿到微服务架构设计的全局层面进行思考。

服务拆分粒度并不是越细越好，需要结合当前团队的人员情况而定，比如当前团队只有5个人，如果拆分后的微服务个数超过50，和人员的数量严重不匹配，不仅不会带来效率的提升，反而会导致效率的下降，建议拆分后每个人维护的微服务不要超过3个，并且每个微服务都有相应的备用负责人，规避可能的突发风险；针对多团队异地开发的情况，最好拆分后每个子团队负责的微服务相对独立，尽量减少异地团队的直接沟通成本。

微服务拆分时经常会遇到一些问题，比如是否需要拆分、拆分粒度的问题等，一般原则是遇到痛点问题时才进行拆分，非拆分不可时才启动微服务拆分，不要为了拆分而拆分。对于拆分方案拿捏不准的场景，尽量先不进行拆分，等想清楚了再定，避免拆分不合理，带来大量返工。

拆分后的微服务组织上层次不要过深，一个请求的处理过程中经过的微服务个数不要超过5个，链路太深会导致定位和追查问题特别麻烦，同时也会带来一定的性能损耗。

对于拆分后的子服务来说，尽量避免相互依赖的场景，子服务调用时相互依赖会导致升级和维护时特别麻烦，增加很多不必要的运维成本。

2. 拆分方法

微服务拆分的总体思想是根据高耦合、低内聚的原则识别出各个微服务的边界。具体拆分思路和业务形态紧密相关，没有绝对的标准，下面介绍微服务架构拆分中使用比较多的两种拆分方式。

（1）以数据为维度进行拆分

按照数据拆分，是微服务拆分中最常见的一种方式，没有特殊考虑时一般根据领域实体数据进行拆分，拆分出来的服务负责处理给定的数据/资源的所有操作。

以电商系统为例，大体可分为订单系统、用户系统、库存系统等。以订单系统为例，订单系统就负责订单核心数据的维护、订单数据的增删查改，确立了主要实体数据后可以根据数据的查询、修改等确定服务的接口，如订单的各种查询接口以及订单状态更新接口。

根据数据维度把系统可以拆分出若干个子服务，但这些数据之间会有不少关联关系。

以百度贴吧为例，贴吧中的人、吧、帖是三个最重要的实体数据，这些实体数据可以分别放到不同的服务中。但是一些常见的实体关系，比如人发过的帖子、人收藏的吧，以及吧的会员、吧的帖子列表等，这些实体关系数据的维护，原则上放在哪个实体上维护没有标准的答案，某个人发过的帖子放在人服务和帖服务都可以，反复在这些地方纠结也不会带来太多的收益，可以人为地指定一个约束，如某个人发过的帖子就放在帖服务里面。

（2）按照使用场景拆分服务

按照使用场景进行服务拆分，这种拆分方式也比较常见，如顺风车的所有后端查询操作之前都在一个单体服务里面，有固定路线/临时路线查询顺路订单，有根据路线查询附近订单，还有查询跨城订单，以及后续的需求乘客查询附近司机，乘客查询顺路司机等。之前这些在线查询接口都在一个单体服务中，多个RD会同时负责和修改这个服务的代码，开发效率低下，测试、上线都会相互影响，维护起来比较困难，后来把这些查询按照功能都拆分为不同的子服务，每个RD单独负责，大大提高了开发、测试以及运维效率。

（3）重要和非重要的拆分

将核心逻辑和非核心逻辑拆分为不同的微服务，然后采用不同的高可用处理方案，比如核心服务尽量做到机房、集群等多维度的冗余和隔离；非核心服务则可以适当降低可用性标准，出现问题时只要能够及时降级和熔断即可。

（4）变和不变的拆分

按照变更频次对服务进行拆分，尽量将变化聚集到少部分微服务上面，系统的绝大多数服务变更很少，可以减少变更对整个系统的冲击和影响。

1.5.2 微服务通信

为了保证拆分之后的微服务能够通力合作，共同对外提供服务能力，需要通过一定的机制将拆分之后的微服务合起来，也就是微服务通信，接下来讨论微服务通信过程中常见的一些问题。

1.微服务通信方式选择

微服务通信时，首先需要考虑的是通信方式的选择，对一些不需要返回业务数据的微服务来说，究竟是采用同步方式还是异步方式呢？同步方式架构层面要求比较高，异步方式架构层面比较优雅，但运维成本比较高，出现问题时排查起来不太方便。之前公司中有过一次不成功的架构升级实例，业务形态完全围绕订单状态流转进行，当前的架构初衷是构建一套基于事件驱动的基础设施，所有微服务均以事件驱动的方式进行触发，最后不仅对业务人员的挑战比较大，同时运维、高可用和问题追查的成本比较高。因此微服务化时，核心服务推荐均采用同步通信的方式，一些非核心服务可以采用异步通信的方式。

2.微服务编排

微服务编排上，一些可以并行调用的场景推荐采用并行调用的方式，可以减少请求的整体耗时，提高用户体验。

当然对于有些场景来说，直接采用并行调用不一定是最好的方式，需要综合考虑和分析。比如，对于搜索和广告引擎来说，整个系统会有大量的过滤子服务组成，如果完全采用串行的方式，并且有着良好的漏斗模型设计，这时请求的整体耗时可能会上升，但整体的调用量会减少不少，成本上有很大的节省；如果采用完全并行的方式，整体耗时会有很大的改善，但无法充分利用漏斗模型，成本上会有一定的浪费。面对这种场景，需要有一套完善的微服务编排引擎，能够同时兼顾性能和成本上的需求，做到微服务编排的最优化。

3.API接口设计

API接口设计上，需要尽可能简单易用，一个接口只实现一个确定的功能，不要设计复杂或者含义不明确的接口，避免滥用；同时接口设计时做好前后向兼容，以及幂等性设计。

4.合理设置超时和重试

微服务通信上，需要合理设置超时和重试，超时和重试设置不要只考虑当前链路，而要从请求全链路的角度，对超时和重试进行统一梳理。

除非有特殊情况，建议在调用的入口统一进行重试，请求过程中不再进行重试，避免故障时的多级重试导致的整体雪崩。

5.数据一致性设计

同时操作多个微服务的场景，需要保证不同微服务之间的数据一致性，数据一致性设

计尽量遵循简单的原则，除非特别场景，不要使用分布式事务。

多个微服务操作时，成功或失败需要以核心数据为准，当同时对多个核心数据进行操作时，可以以其中一个为准，遇到个别数据不一致的场景，可以采用线下对账的方式对不一致的数据进行校对和修正。

1.5.3 微服务稳定性保障

微服务改造中，挑战最大的就是拆分之后的稳定性保障，拆分之后链路复杂、故障点众多，需要一套体系化的稳定性保障机制。

1.稳定性保障的目标

微服务稳定性保障需要从事前、事中和事后全方位进行考虑。微服务架构下，应用程序、依赖服务、网络、硬件等都有可能出现故障，稳定性设计和保障的具体目标如下。

故障预防，尽可能减少故障的产生，绝大多数稳定性问题和稳定性故障发生都有一定的诱因，并且一般是在多种拦截手段均失灵的情况下故障才会发生，如果我们在故障发生前制定完备的稳定性保障措施，可以最大限度地减少稳定性故障的发生。

故障快速定位，完全不出故障的业务是不存在的，关键是出故障时能够快速发现故障，只有及时发现，才能在最短时间内采取相应的解决措施。

故障快速止损，发生故障后第一时间要进行业务止损，恢复业务的正常运行，故障深层次的具体原因可以事后再分析和复盘解决。

2.稳定性保障的6个维度

系统故障点很多，稳定性保障就是对故障点进行管理的过程。可以从故障点管理的角度将整个稳定性设计和保障分为如下**隔离、冗余、容灾容错、变更管理、时间相关故障管理**与**运维友好**6个维度。

(1) 隔离

影响系统稳定性的不稳定性因素和故障点很多，从稳定性保障的角度看，很自然的想法就是，如何尽量减少如此多的故障点给系统稳定性带来的隐患，比如某电商业务有200个微服务组成，如果这100个微服务中的任何一个出问题，导致业务不可用，那么系统的可用性就会很低。业务层面如果能够梳理和抽象出保障业务核心运行的最小系统（比如上面提到的电商业务最小系统包含的服务可能会小于50个，其他都是增值和支撑性的服务），同时将最小系统之外的不稳定性因素从最小系统中隔离出来，就可以大大增强系统的稳定性和健壮性。因此，稳定性设计的第一个原则就是“隔离”，通过各种隔离机制，将核心服务之前的故障点隔离出去，保证核心服务的可用性。

(2) 冗余

通过隔离，可以减少绝大多数不稳定性因素对系统稳定性的影响，但如果核心服务或核心服务所在的环境出问题，就无法从隔离中受益。我们需要有相应的机制保证核心业务的部分单元出问题时，业务整体运行不受大的影响，这种机制就是冗余。通过服务级别、机器级别、集群级别、机房级别等多种维度的冗余，我们可以保证：即便核心服务出问题，也可以通过相应的流量切换策略，将流量切到冗余节点上，保证业务不受影响。

(3) 容灾容错

通过冗余可以保证核心服务及其部署环境变化时的系统稳定性，但如果系统外部输入有变化，比如遇到突然大流量、异常流量或者突发的安全攻击，而系统事先没有相应的应对机制，则业务很可能瞬间被击垮。因此，稳定性设计的第三个原则是“容灾容错”，通过构建多维度的容灾容错体系，保证系统面对异常输入时，仍然能够提高稳定的输出能力。



提示

隔离、冗余及容灾是解决外部环境与输入导致的各种故障点和风险。

①**隔离**强调的是将微服务架构体系中非核心服务导致的故障隔离出去，减少非核心因素对业务核心的稳定性影响，隔离工作做好之后只需要考虑核心服务的稳定性。

②**冗余**解决的是核心服务面对各种环境变化时的稳定性应对，比如服务故障、交换机故障、网络故障、机房故障等，通过各种层次的冗余和流量调度机制，保证业务面对各种硬件和环境变化时仍然可以通过冗余切换提供稳定的服务。

③**容灾**解决的是面对各种输入变化时的稳定性应对，比如突发大流量、异常请求、安全攻击等，容灾容错保证系统面对各种输入突变问题时，稳定性不受大的影响。通过隔离、冗余和容灾，解决了非核心故障点风险、环境和输入风险，保障了业务最小子系统面对各自外部变化时均能够提供稳定输出。

(4) 变更管理

通过隔离、冗余与容灾可以排除和应对业务最小子系统的各种外部稳定性风险，变更管理、时间相关故障管理是为了解决核心系统自身的稳定性问题。绝大部分稳定性故障都是由变更引起，系统如果长时间没有任何变更，很少会有稳定性问题，因此服务稳定性保障的关键一环是严把变更这一关，保证变更质量。

(5) 时间相关故障管理

服务没有变更时，有一类故障很少发生并且很难发现，就是随时间变化而产生的ID越界和溢出，这类故障平常测试时很难发现，并且发生时会对整个系统产生很大的影响，需要从设计层面开始把控，比如随时间变化的ID字段尽量使用int64。

(6) 运维友好

隔离、冗余和容灾减少了核心服务的外部稳定性风险，变更管理和时间相关故障管理保证了核心服务自身的稳定性。上述5种措施构成了业务稳定性预防的整个闭环，但即使设计得再好，也不能完全杜绝稳定性故障，稳定性风险只能最大限度地减少，因此服务的研发生命周期中，还需要加上运维友好的考虑。设计时需要针对稳定性问题的快速发现进行特别考虑，如日志怎么输出，统计信息如何收集等。通过运维友好设计，比如log、metric和trace等方式的良好设计与运用，建立全方位多维度的故障发现机制，保证出现问题时可以快速发现和快速止损，最大程度上减少稳定性风险的影响。

3.稳定性保障的设计建议

总之，隔离、冗余和容灾、变更管理、时间相关故障管理和运维友好构成了整个的稳定性保障体系，下面会对这几个维度详细展开讨论。

（1）隔离机制

隔离机制的指导原则是将变和不变、重要和非重要区分开来，变更是稳定性故障的最主要的来源，将容易发生变化的部分从核心服务和核心流程中剥离开来，减少核心部分的变更，可以保障核心系统的稳定性。

隔离机制的一大手段就是解耦，通过解耦可以把核心服务和非核心服务隔离开来，同时核心服务访问非核心服务时，通过熔断、超时和重试等机制，最大限度地保障非核心服务故障不会影响整体的稳定性。

异步化是流程隔离的主要方式，为了减少非核心服务故障对核心服务的影响，可以将一些非核心流程异步化处理。

为了减少对核心服务的影响，核心服务建议单独部署，如果特殊情况下需要和其他服务混部，一定要保证有精细的资源隔离机制，保证不会因为其他服务资源使用不当，影响核心服务的稳定性；同时核心服务用到的一些关键基础服务、关键数据服务也建议隔离开来，分开部署，避免其他服务出问题对核心流程的影响。

（2）冗余

为了保证故障时冗余机制的可用性，事先需要进行完善的冗余容量规划，各级故障发生后，剩余容量足够承载峰值流量，一个大体的原则是：机房内尽量满足 $N+2$ 冗余，2用来冗余（应急变更故障和其他故障），机房间 $N+1$ 冗余，1个机房挂掉后，剩余机房能承载所有流量。

为了尽量避免冗余同时失效的情况，冗余副本之间需要相互独立，完全对等，不能相互依赖，机房内副本跨交换机部署（此时一般也能保证跨机柜），如果有多机房冗余的情况，各机房独立，不能有完全相同的依赖。

系统为了构建完善的冗余机制，服务设计时尽量无状态，无状态的服务随时可部署启动，方便水平扩展。

为了用好冗余，常态下，需要支持灵活的流量调度策略，具体包含服务发现、流量路由、负载均衡等；节点故障时，通过健康检查、故障节点剔除、动态路由切换等机制，可以平滑地将流量从故障节点切到冗余节点，保证节点故障不会影响系统整体稳定性。

（3）容灾容错

为了减少异常流量对系统的影响，服务开发和设计时需要采取防御性编程，提前想到可能面临的种种异常情况，并针对性地进行防御和解决。

服务可以通过降级和限流，减少突发大流量对系统的冲击，保证系统稳定输出，为了保证降级和限流操作的即时性，系统需要支持配置的动态修改和生效。

（4）变更管理

为了提高变更的质量，减少变更带来的稳定性风险，需要从测试、上线和规范等多角度进行保障。

测试上，通过建设线上/线下多级测试环境，比如线下的联调环境、集成环境、仿真环境，线上的预发布环境，通过不同用途的环境，提高各种场景的模拟能力，提高故障拦截率。

针对变更，需要制定完善的变更规范，变更时严格按照规范进行，再小的变更都可能会产生稳定性隐患，因此变更时一定要加强稳定性意识，变更的每一步操作都要进行各项监控项检查，如果出现问题立即进行回滚。

对所有变更操作，不管是服务变更、配置变更和数据变更，均要采用灰度机制，灰度观察没有问题后再放量，关键特性需要设置相应的特性开关，方便根据需要灵活地对特性进行开启和关闭。

最后，针对变更过程中比较容易遇到的问题，可以梳理出一套变更反模式出来，供其他人作为反面例子参考，下面是梳理的之前线上服务遇到的典型变更反模式。

1) 代码搭车上线。

比如由于缺乏有效的代码修改管理机制，某产品线由于代码搭车上线，出现过多次线上故障，并且由于变更时涉及的修改比较多，导致问题定位和追查时非常困难。

2) 服务回滚时遗漏回滚代码。

某业务隐私页面改动，首页浮层上线有问题，导致呼叫量下降，上线回滚后恢复；回滚后没有第一时间把代码回滚掉，其他人上线时将未回滚的问题代码再次带上线，导致连续两天出现系统故障。

因此，服务回滚的时候，必须第一时间回滚代码，保证主线代码任何时候都是干净没有问题的。

3) 服务启动或者回滚时间过长。

某产品计价服务上线异常，回滚时单个服务回滚时间太长，导致未能短时间内快速止损。经排查，回滚过程中部署系统和服务都存在耗时过长的现象，由于服务回滚速度比较慢，产生了较大的线上服务故障。定期检查和优化服务的启动和回滚时间，保证出现故障时可以第一时间完成回滚操作。

4) 配置文件缺少有效的校验机制。

配置文件由模型产出，数据配送系统实时配送到线上服务，模型产生的配置文件有问题，引发线上故障。因此，针对配置文件，尤其是策略模型产出的配置文件，需要建立严格的检查和校验机制。

5) 小流量后的修改没有经过严格的测试和灰度验证。

某服务经过小流量灰度后，代码又有少量修改，再次上线时未灰度，导致线上故障。

再小的变更，都要进行测试、灰度和双重检查（double check）。修改一行代码，也可能导致线上的稳定性故障。

（5）运维友好

为了实现运维友好的系统设计，系统需要将故障分析和定位时涉及的所有相关信息监控起来，构建完善的监控闭环，对系统层、服务层、接口层、业务层等维度进行监控收集和告警。

为了减少系统的稳定性隐患，微服务架构设计中尽量遵循简单的设计原则，从业务的真实需求出发，避免纯粹从技术角度出发的高大上技术方案，如果不是业务的核心功能，必要时可以进行一定的折中和裁剪，尽量保证系统的简单和简洁性。

另外，尽量使用一些验证过的技术，对于没有充分验证过的新技术的引入需要特别谨慎，如果一定要引入，需要控制好节奏，循序渐进。

1.6 本章小结

通过学习本章，读者可以对微服务架构有个整体的了解，微服务可以带来很多收益，同时在开发、测试以及运维上都会带来很大的挑战，可以从是否影响业务迭代速度、是否出现并行开发而导致代码冲突等角度出发，判断当前是否合适进行微服务改造；确定微服务改造后，需要提前从服务框架、服务标准化、服务持续集成和发布等方面提前准备，保证微服务的顺利开展；微服务具体实施时，首先要确定好微服务拆分的原则，良好的微服务拆分是保证微服务成功的一半，然后从微服务通信、微服务稳定性保障等维度进行架构设计，保证微服务改造的高质量落地。

第2章 微服务治理

业务微服务化之后，在开发、测试、运维等各个环节都会带来很多挑战，服务治理聚焦解决业务微服务化之后带来的突出问题。

本章着重讨论服务治理的问题域和挑战，以及服务治理的演进，最后讨论微服务治理的理想方案。

2.1 微服务治理基础

2.1.1 服务治理由来

在单体服务时代并没有服务治理的概念，随着单个服务慢慢演变和拆分为众多微服务组成的系统，服务提供者与服务使用者之间如果没有明确的契约和规范，开发、测试和运维过程中会带来很大的沟通成本。之前线上服务遇到过的一个真实的例子，服务代码中调用地理围栏服务，这块代码平常没什么变更，大家关注也不多，线上服务调用地理围栏突然发生故障，服务使用方联系之前负责对接的人，发现早已离职，并且这个服务多年没有修改，当前处于无人维护状态，导致出问题时无能及时响应的悲剧。

服务治理就是解决服务提供者和服务使用者之间的链路通信与治理，主要包括如下几个维度的目标。

（1）服务定义和SLA

服务定义是向外界说明当前服务提供者具体提供了哪些功能和能力，具体通过什么方式来使用服务，以及服务对外承诺的服务SLA^[1]。

（2）服务注册中心

服务使用者使用服务之前需要先知道服务提供方具体在什么地方，服务注册中心负责管理各个服务的地址信息，用于服务使用者查询。

（3）服务生命周期管理

服务生命周期管理用于对服务部署的整个生命周期进行管理，包含服务的上线和下线，服务的扩容和缩容等。

（4）服务通信和链路治理

服务通信和链路治理是服务提供方和使用方约定，为了达成SLA而采用的一些治理手段，比如超时重试、限流降级等。

（5）服务授权和通信安全

通信安全严格来说也是属于链路通信和治理的范畴，只不过由于安全特别重要，可以将这块单独提出来进行讲述。

[1] 服务等级协议 (Service-Level Agreement), 是服务提供方和服务使用方之间的协议，定义了服务品质的承诺 以及达不到承诺的具体惩罚措施，服务品质主要包括QPS、各分位耗时等。

2.1.2 服务治理的目标与愿景

服务治理整体愿景是打造可见、可控、可追溯、可预测的全系统视图，立足全局，业务优先。可见即所得，可见皆可控。虽然服务治理的立场是系统全局，但是服务治理必须做到深入业务需求，解决业务实际问题。服务治理的开展和推广是讲究策略的，需要和业务方明确服务治理的长期收益。除此以外，服务治理要能溯源系统问题，提升定位效率。服务治理还要有能力提前发现全系统中的异常，如场景单点故障、全系统容量瓶颈和不合理的容量冗余等。

服务治理的目标是提供标准化、平台化、智能化的一站式服务治理设施，支持业务的稳定运行和高效迭代。服务治理的范围贯穿服务的全生命周期。

1) 标准化是服务治理一切工作的基石，服务治理在规范、工具、框架和平台各个层次上都要有统一的标准。

2) 平台化是擎天柱，标准如果完全依靠人来执行，不仅效率非常低，也很难保证不会出现各种遗漏以及落地和标准不一致的现象，将标准进行平台化落地，可以保证服务治理标准可以高效、稳定执行。

3) 智能化是服务治理的理想状态，属于远景目标和仰望的星空，期望所有的服务治理工作能逐步通过自动化和智能化方式推进。

4) 一站式，强调服务治理的闭环体系，所有通用的可感、可控、可追溯治理均能通过闭环高效完成。

服务治理基础设施的呈现形式多样，具体包括规范、工具、框架和平台等。

稳定是服务治理的基础和底线，易用才能提高用户效率，服务治理一般只关注稳定性和效率，对性能关注比较少，具体来说稳定运行，提高业务的稳定性，是服务治理的第一目标；高效迭代，提高业务的迭代效率，是服务治理的另一目标。

从方法论的角度看，服务治理是把复杂的、重复的事情简单化，把简单的事情标准化，标准的事情流程化，流程的事情自动化。

2.1.3 服务治理的工作范畴

基于上述服务治理目标，服务治理的实际工作包括两部分：**一站式的服务治理平台、普适性的服务开发框架**。其中一站式的服务治理平台作为服务治理输出的根据地，最重要的作用是服务治理规范的固化。服务治理规范从经验到wiki、从wiki到平台的发展趋势也表明了服务治理规范是业务方和服务治理团队之间一个非常重要的桥梁。普适性的服务开发框架，是服务治理团队非常重要的一个抓手，平台能做的事情是有限的，而广泛使用的开发框架则可以完成更多的事情。普适性笔者认为服务开发框架最重要的优点，如何和业务方结合、提升服务开发框架的普适性非常重要。

从内容上看，服务治理具体工作范围可以概括如下。

(1) 系统可见

主要是服务治理平台的工作，具体包含服务静态描述，比如服务基本信息、服务等级、全局依赖拓扑、路由关系、IDL（SDK）、权限、SLA、流量控制策略等，服务治理平台需要有能力和帮助我们梳理依赖层级、判定服务等级。

服务实时拓扑，具体包括全局流量拓扑、报表审计等，服务治理平台需要有能力和展示实时流量状态，基于场景定位线上问题。

(2) 系统可控

打通服务治理平台和服务开发框架，为其他平台输出能力，具体包括路由切换、降级预案等。

(3) 系统可追溯

将问题现场进行录制，如流量录制、日志录制等方式，然后使用相应的跟踪和追溯系统，对典型场景下的常见问题进行回放和溯源，做到“一次录制，多次回放”。基于追溯和回放机制，可以解决根因定位、场景分析等问题。

(4) 系统可量化、可预测

系统可预测，是指预测业务接下来一段时间的行为，比如流量、容量、安全等，同时通过各种工具和平台检测系统的潜在风险。以容量规划为例，需要能够预见和判断业务接下来一段时间的增加趋势和运营活动，判断可能的流量峰值以及对系统容量的影响；接下来通过全链路压测、单系统压测等方式探测系统可能的容量风险。

(5) 立足全局、业务优先

服务治理的目的是解决业务的稳定性和效率问题，必须完全聚焦业务，从业务出发，重点解决业务当前的痛点问题。

2.1.4 服务治理闭环体系

服务治理本质上来说是解决微服务化后产生的一系列挑战，比如效率治理、稳定性治理、效果治理、性能成本治理等。这些治理举措一般作用于设计开发阶段，我们可以称之为“正向治理”，在服务上线后，可以基于运行时的各种数据，进行综合治理和反馈，从而构成一个治理闭环，如图2-1所示。

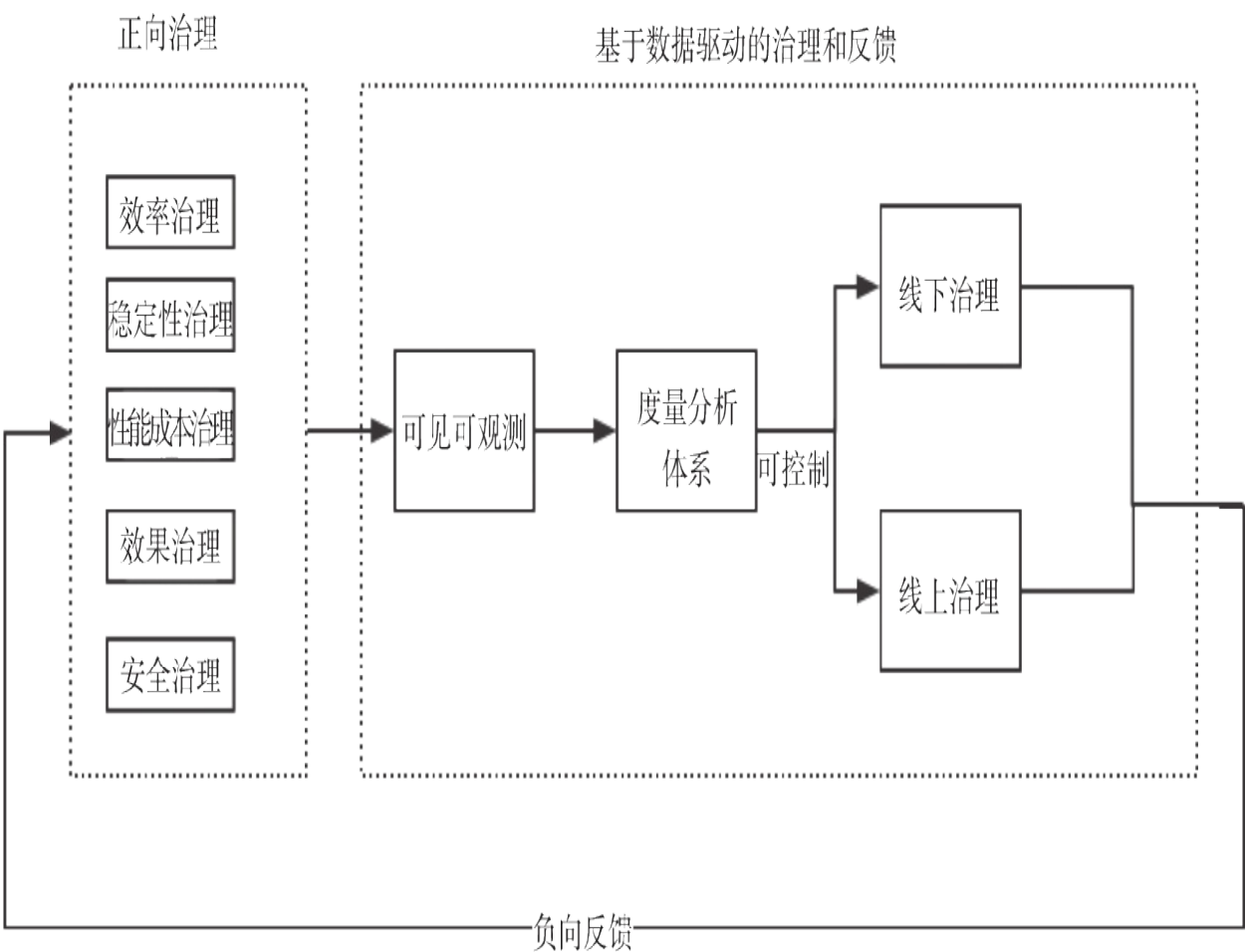


图2-1 服务治理闭环体系

在上述服务治理闭环中，通过多维度的度量数据收集，结合静态的服务元数据信息，构成一个完整的服务治理数据集，然后基于这些数据进行综合度量和分析。根据分析结果会产生相应的控制和管控措施，分别从线上和线下两个维度进行综合治理，同时将治理结果负向反馈回来，作为进一步治理的输入。

本章接下来几节均围绕这个治理闭环展开，对服务治理进行深入剖析。

2.2 正向服务治理

正向治理是通过相应的治理手段，解决微服务改造后带来的一系列问题，接下来围绕效率、稳定性和效果方面的治理工作展开讨论。

2.2.1 效率治理

效率包括的范围很广，具体包括开发效率、测试效率、运维效率等。开发效率一般通过采用微服务框架的方式解决，运维效率放到稳定性治理里面讨论，接下来主要讨论测试效率相关的治理，包括测试环境构造，以及测试数据的获取。

1. 基于流量录制和回放的测试

微服务化架构下，微服务自身依赖的第三方服务、数据越来越多，给传统的测试方式带来很多困难，如被依赖的线下服务不稳定；服务无法提供期望的响应数据；缺少测试场景构造标准等。针对这些问题，一般能想到的思路是提高测试环境稳定性、自己构造测试数据和测试场景等方式，提高测试的确定性和有效性，但这些方式不能从根本上解决问题。比如，服务不稳定的情况仍然无法避免，通过更改代码注入对象等方式不仅繁杂而且非常容易出错，并且没有一定的依据，构造的测试场景不充分，心里没底。

上述问题导致了代码质量下降、自测/测试困难，在业务越来越复杂，依赖服务越来越多的大环境下，这些问题变得越加严重，到了必须要升级解决方案的地步。

针对上述微服务测试的不足，尝试推出了流量录制和回放解决方案，通过一定的方式将线上的真实流量录制下来，在线下进行回放。线下测试时，基于线上的回放流量可以灵活地测试服务的各自功能，还可以基于回放流量进行编辑修改，对服务依赖的第三方请求进行mock和定制，使代码开发变得简单方便，且可随改随测，解决RD开发不便、自测流程过长的的问题。

录制的流量要包含服务的请求/返回流量，以及能关联上的与其对应的第三方交互流量。如果仅有服务的请求/返回流量，则只能被应用在查询类只读系统中，应用范围受限；如果第三方交互不能和请求正确关联，则无法正确应用这些流量。

要解决流量的关联难题，一个解决方案是将流量整流，使其串行化。将线上原本服务并行流量的一台机器，控制其接收到的流量，将其服务的流量降低至串行，且流量之间间隔一定的时间（以便给异步第三方请求预留时间），那么这些流量的请求/响应和其第三方交互流量就被自动地关联了起来。

还有一个难题是如何将请求和第三方交互流量关联起来，这块其实没有一个特别通用的解决方案，可以基于分布式跟踪技术将请求和对第三方系统调用的流量关联起来。但这种方式难点是并非所有第三方交互流量都带有请求跟踪标识。还有一种思路是通过拦截语言层面的系统调用，将请求和对第三方系统的调用进行关联。

2. 基于仿真环境的测试

作为在线服务，为了满足灰度发布、测试等多维度的需求，一般需要支持灵活的分流策略，将流量调度到不同的环境中。

为了支持精细的灰度发布策略，需要根据一定的规则将线上流量分流到小流量集群，为了支持使用真实流量进行高仿真测试，还需要根据策略规则将一定流量引到仿真环境中。小流量分流的特点是流量发往小流量集群后，不再进行任何处理，直接等待小流量集

群的返回结果，然后将该结果转发给调用侧。仿真环境分流的特点是发往仿真环境的流量只是一份“影子”流量，相当于Oneway请求（不需要应答消息），对业务处理过程没有任何影响，还是和没有仿真环境的情况下一样处理。

仿真环境支持可以在服务框架层面进行，为了实现基于请求内容的路由和转发，框架充当了一个Proxy的角色，将请求解析后再发包发送给下游环境。

基于仿真流量的特点，为了不对业务产生影响，发送到下游仿真环境的超时时间可以设置得非常小，即使有一些流量超时也没有关系，同时为了在自动生成的代码框架下实现仿真环境处理后的Oneway效果，可以自定义一个传输层，这个传输层的具体实现是什么都不进行处理，直接丢弃即可。

2.2.2 稳定性治理

稳定性治理是服务治理的重中之重，下面重点讨论稳定性治理的体系和模型。

1.稳定性治理模型

稳定性治理是一个对故障进行管理的过程。从故障管理的视角看，可以分为故障预防、故障发现、故障定位、故障止损以及故障恢复5个阶段，稳定性建设的各项工作融入故障模型的各个阶段。如果将稳定性故障和火灾进行类比，稳定性工作模型实际上是一个防火-放火-灭火模型。所谓防火，就是通过各种机制和措施，提前排查出系统中各种可能的隐患，防止灾难的发生；灭火是指实际发生了问题，就要最大限度地进行止损，减少灾难的影响面，尽快恢复业务的正常运行；放火类似于消防演习，定期模拟灾难的发生，并制定相应的疏散通道，通过演习可以排查出当前仍然有哪些待改进的地方。比如，消防意识不够，疏散通道设计不太合理等，通过不断的周期性演习和针对性改进，可以提高大家在面对真实火灾的应对速度和处理能力，可以对实际的灾难的控制有很好的参考价值，演习尽可能有一定的逼真度，如果只是例行的走个过场，不会收到太大的效果。

从各阶段的抓手上看，故障预防包括稳定性设计、风险度量分析体系以及变更管理这几个环节，分别从服务设计、风险检测和分析及变更拦截这几个维度，在研发生命周期的不同环节对故障进行多级拦截。

服务上线后需要有相应的机制能够检测当前系统是否正常工作，当系统不正常时有相应的控制措施，因此从故障发现、定位和止损上，需要有一连串的基础设施支持，保证系统可见、故障可发现可定位、可控制。

2.基于容器化的稳定性治理

微服务在开发、测试、运维、容量成本等方面带来了诸多难题，容器技术的使用可以很大程度上缓解微服务架构所带来的问题。将容器技术和微服务架构结合，从开发、测试到上线，实现了“一次编写，到处运行”。

容器的最有革命性的创新是**镜像技术**，它将应用程序、基础库和环境等封装在一起，作为微服务封装和运行的基石。轻量级的镜像技术作为微服务的交付方式，从如下几个方面极大地影响和改变了微服务生态体系。

（1）测试

物理机环境下，如果直接基于裸机，那么环境搭建的开销很大，并且很难保证不同环境的一致性。虚拟机环境下，虽然也引入了镜像，但镜像特别大，一般都是几十GB，甚至上百GB，很难快速创建和迁移。容器环境下，由于镜像比较轻量，每次变更后，可以快速创建本次变更对应的镜像，同时可以基于本次镜像快速创建多个完全相同的测试环境，容器镜像封装了所有运行应用程序所必需的相关细节，比如应用依赖以及操作系统。这就使得镜像从一个环境移植到另外一个环境更加灵活，有力地支撑了微服务快速迭代场景下的测试。

（2）部署

微服务架构下微服务个数比较多，并且每个服务的变更非常频繁，运维的工作量很大，借助容器镜像，可以把环境交付提前。每个研发多付出5%的工作量，换取运维200%的工作量，可以加速微服务变更的快速部署和落地。

微服务架构下流量变化很快，遇到突发大流量时，如果系统具备快速便捷的扩容/缩容能力，可以极大地提高系统的稳定性和灵活性。我们可以借助容器技术，快速构建完善的扩缩容基础设施，流量峰值时快速扩容，流量低峰时缩容。

微服务的云化架构，在运维层面和之前会有很大的差异，基础设施层面，比如部署系统、配置系统、监控系统等都需要针对上云进行相应的适配调整，服务云化过程中也会遇到很多特有的问题，下面会梳理下之前微服务云化过程中实际过程中遇到的一些典型问题，以便给后续的微服务云化落地一些帮助。

（1）容器网络和物理机网络打通

之前的微服务云化迁移是直接从物理机迁移到Kubernetes容器集群，迁移一般是一个服务一个服务逐渐迁移的，如果Kubernetes集群和原来物理机集群通信上完全不互通，会使服务迁移前后在部署上有很多的改动。比如以服务间调用来说，之前是集群内部的调用，迁移期间需要修改为以Kubernetes Ingress的方式进行调用，但整个物理机集群都被迁移到Kubernetes集群后，在Kubernetes网络中，又需要改回到集群内部调用的方式。从终态看，整个迁移过程中做了很多无用的工作。

为了减少Kubernetes容器化改造过程中频繁的调用方式改动，我们在迁移过程中遵循一个重要的原则，迁移前后网络互通，这样迁移过程中调用方式不需要有任何变化，迁移过程中业务完全不需要感知。

（2）认清物理机和容器环境上的差异

云化架构下，虽然使用上和之前没有明显的差异，但毕竟是两个完全不同的环境，在物理机上验证完全没有问题，不代表容器环境下也可以正常工作。之前遇到过一个問題，服务之前运行完全正常，迁移到云化架构下一段时间后，由于日志被错误地输出到了内存型文件系统中，内存被一点点耗尽，由于没有完善的线上监控，引发线上故障；导致问题的具体原因是物理机日志组件使用了字符设备，容器环境下虽然在宿主机下也配置了对应的字符设备，但并未为宿主机下的容器单独创建，因此云化迁移过程中，应该对容器和物理机环境上的一些差异有着清醒的认识。

（3）容器资源隔离

容器环境下，各种服务混部，当某个容器消耗的资源超过一定限度时，如果没有完善的资源隔离机制，就会导致同一宿主机上的其他服务异常。精细化的资源隔离技术是容器技术大规模推广的一个必要前提。

（4）故障容灾

服务上云毕竟是个全新的环境，可能会遇到各种类型的问题，特别是网络层面，从硬件网络过渡到软件定义网络（SDN），虽然灵活性比之前大大加强，但SDN网络在成熟度和稳定性上比之前的硬件网络还有不小的差异。之前遇到过一次SDN网络异常，导致整个机房的容器服务不可用，幸好当时有多机房流量切换预案，才没有导致更大的问题。因此

服务上云前，需要提前想到有哪些可能的故障类型，针对每种故障设置相应的容灾预案。

2.3 效果治理

微服务架构的初衷主要是建立一个方便快速迭代和演进的架构，帮助业务实现方案快速落地和试错，因此在微服务研发流程上，需要重点关注两个事情：如何支撑业务特性快速落地，如何建立业务效果的闭环反馈机制。

1.配置化

工程技术同学从接触编程开始就同时认识了配置，一个工程有线上/线下环境、服务地址、各种功能参数等配置。我认为，配置本质上是一个软件对外提供的开放的能力，能够在不改动软件逻辑的情况下按照用户的设置执行特定的流程和功能。业务系统虽然从功能特性上看迭代很快，但一旦业务模式确定后主流程是基本不变的，变的是针对各种业务场景的一些定制。可以把业务系统比作一个软件，如何能够将一些业务通用功能抽离出来，把差异做成配置，以配置化的方式支持可能的扩展和定制，就可以大大加快业务迭代速度。

为了实现配置化，首先需要有一个通用的配置平台，在配置平台UI上进行配置信息的修改和查询；同时需要有多语言的配置SDK库，支持获取最新的配置信息。在配置发生变更时，配置平台能够把变更后的配置信息推送到业务服务所在的集群上，供多语言的配置SDK使用。配置平台的实现还是很复杂的，如果对具体的实现原理和实现细节感兴趣，可以研究下携程开源的分布式配置中心项目，地址：<https://github.com/ctripcorp/apollo>，适用于微服务配置管理场景，完成度和成熟度比较高。

配置化的本质是对业务进行建模，把每个对象模型化，对系统进行抽象和建模，是代码的另外一种写法，需要业务架构层面对业务逻辑进行合理的抽象，识别出常见的业务变化点，针对变化进行建模。

2.基于A/B测试的业务效果评估体系

在互联网企业中，在线实验特别是A/B测试在产品创新和企业发展的过程中扮演着非常重要的角色。我们通常需要利用A/B测试来验证一些新特性或者调整，并以此来支撑我们在产品开发过程的决策，这样才能真正做到数据驱动的业务决策，因此基于A/B测试建立一套完善的业务效果评估体系，可以对每个变更的效果都可以度量。

效果评估需要先建立完善的业务指标体系，比如业务的一级和二级指标有哪些，有了具体的业务指标，效果评估才有基础和依据。其次，需要有一个针对指标的效果评估平台，真正做到业务效果可视化、可量化、可优化。

2.4 可见可观测

可见可观测是服务治理反馈机制的第一步，只有获取到足够多有价值的数据，才能对服务的运行状态进行分析和控制。

2.4.1 服务可见性

服务元数据平台负责维护服务相关的元数据信息，主要包括服务层面、接口层面、拓扑层面这几个维度的元数据信息，通过服务元数据平台，可以对系统提供全方位的可见性。

服务层面，服务元数据平台汇聚了当前所有的服务列表，默认按照服务功能进行分类，业务需要某种功能的服务时就可以直接到服务商城浏览和查询，做到服务信息的透明化，减少重复建设。为了增强服务查询的灵活性，可以支持多种服务查询方式，比如按照服务所属部门查询、按照服务属性查询等，服务基本信息平台化等。服务基本信息如下。

- 1) **服务描述**：简要描述服务提供的基本能力，服务的适用场景等，供潜在服务使用者参考，必要时可以加上服务的详细描述信息wiki，以及服务对应的邮件组和沟通群。
- 2) **服务所有权**：服务当前所属部门、服务当前的owner等。
- 3) **服务对外接口**：服务接口定义，使用说明和注意事项等。
- 4) **服务SLA**：服务对外的SLA承诺。
- 5) **服务的上下游拓扑**：服务商城中维护了每个服务的上下游依赖，基于上下游依赖，不仅可以查询上下游服务的使用方式和使用情况，同时也可以进行服务重大变更时的上下游服务通知。
- 6) **服务变更**：服务商城中维护服务每个重要变更的变更日志，重要变更时会通过相应机制知会上下游依赖，上下游会评估是否需要适配升级等，这样服务使用者可以从变更历史中了解到服务的整个发展脉络。
- 7) **服务接入和资源配额管理**：服务如何接入，资源配额如何申请等。
- 8) **服务线上部署和线下测试环境信息**：描述了服务线上和线下的部署信息，使用者直接基于平台给定的环境使用。

服务元数据平台的作用是将服务基本信息平台化，同时可以将服务接入、服务客服等日常工作从线下转移到线上，减少线下的人力成本开销。因此服务元数据平台的评价标准是微服务接入和运营时线下沟通的比重。

2.4.2 变更可见性

变更是引发系统故障的主要因素，通过对各个维度的变更进行系统的梳理和记录，不仅方便出现故障时的追溯和定位，后续还可以基于完善的变更事件库，对这些变更的原因、质量以及影响进行全面的审计和分析，从中找到规律性的东西，建立相应的改进反馈闭环。

服务变更是变更的最重要来源，常见的服务变更方式有应用变更、配置变更、数据变更以及预案变更等。

对于微服务架构来说，除了关注当前服务的变更之外，还需要关注上下游依赖服务的变更，以及部署层面关联服务的变更，比如和当前服务混部在同一台物理机上的其他服务变更。

除了服务变更，还需要对服务周边的各个环境变更进行记录，比如网络变更、机器变更、机房变更、交换机变更，这些变更都可能对服务的正常运行产生影响。

2.4.3 可观测性

微服务架构下，各个微服务采用分布式部署，并且通过网络进行分布式通信，随着微服务个数和集群规模的扩大，系统中会产生各种形式的故障，并且很多故障是无法事先预测的，因此需要有完善的平台和工具体系，能够对微服务的运行状态进行全方位多维度的监控，使得我们可以随时掌握微服务当前的运行状态，并且针对运行状态不正常的场景进行及时处理。

可观测性就是用于上述微服务状态监控的工具集合，具体分为Logging（日志系统）、Metrics（度量系统）和Tracing（分布式跟踪系统）3个层次，三者的关系图如图2-2所示。

（1）Logging系统

日志用于对系统中离散的事件进行记录，比如服务的调试信息和错误信息，日志是系统监控的基石，也是服务状态监控和问题诊断的第一个抓手，通过日志可以大体判断系统的运行状态是否正常。

日志是最常见、最通用的监控手段，但服务的日志监控和告警一般都需要人工添加，不仅效率低，也很容易遗漏。同时不同服务的日志格式和日志信息可能都会有差异，不太方便进行标准化，不仅日志收集、处理和展示比较麻烦，有太多个性化的需求，而且每个服务的日志都不统一，基于日志的全系统问题定位也非常麻烦。

（2）Metric系统

为了提高日志和监控的标准化，引入了Metric的概念，Metric就是将日志中可以聚合的部分通过标准化的协议进行处理，Metric定义一套完善的日志收集、传输和处理标准，通过Metric可以实现日志和监控的标准化，同时基于Metric的日志聚合特性，聚合后的日志会小很多，减少日志系统的成本开销。

（3）Tracing系统

Tracing用于记录请求级别的信息，它会跟踪请求整个链路的执行过程和各阶段耗时信息，基于Tracing，可以定位请求性能问题和跨服务交互相关的问题。

Logging、Metric和Tracing这3个系统关注的侧重点有所不同。Logging系统对服务的运行状态和事件进行记录，比如调试日志、错误日志等，日志记录一般会比较全（有时出于性能考虑可能需要在不同场景下打开相应的日志开关），这些日志会作为问题追查和定位的依据。Metric系统侧重事件聚合，相比日志来说，Metric关注的对象相对小一些，比如服务QPS、成功率、耗时等，聚焦核心指标，基于Metric可以方便实现服务核心指标的监控告警自动化。Tracing关注请求级别的处理。使用过程中一般先关注日志监控是否有问题，如果有问题通过Metric查看具体的服务和指标异常，最后通过Tracing进行全链路排查。

Metric系统和Tracing系统在实际中是交叉使用的，可类比公安破案的过程。

1) 公安会在犯罪高发地段部署监控，以便及时发现各类蛛丝马迹，比如一段时间内，每个过客的神色、停留时长，类似Metric系统，以在大面上发现隐患或嫌疑。

2) 通过Metric系统锁定高嫌疑犯后，会进一步分析嫌疑犯的衣食住行轨迹，得到证据链，从而给嫌犯定罪。衣食住行轨迹系统，就类似Tracing系统。这里通过Tracing获得证据链，确认之前的异常指标确实导致了问题。

3) 最终从客观角度分析罪犯的作案根因时，可能又会将一段时间内该罪犯的行为指标进行时序分析，最终分析出该罪犯是因强制拆迁投诉无门而报复社会。这里通过复盘Metric，在微观面（个人层次）找到犯罪根因。

综上所述，一般通过无处不在的Metric监控，发现大面上的异常或隐患，如机器cpu idle过低、服务崩溃退出等；再通过Tracing系统，查看异常机器或实例上的轨迹，确认异常由哪个环节引入，造成了多大损失（例如个别机器CPU idle低处理慢，因为有上游重试，未必会造成损失）；如果问题由业务逻辑导致，此时基本能找到根因；否则再通过Metric系统，确定运行环境类根因，如查看异常机器或实例上的QPS、CPU idle率、进程CPU使用率的秒级曲线，最终判定cpu idle低是由QPS增加导致，还是其他进程CPU使用率增加导致。

一般来说，对于大故障，直接关联分析各种指标就能定位出问题；对于零星故障，业务逻辑故障，上下游耦合类故障（如调度），要通过跟踪才能分析清楚。

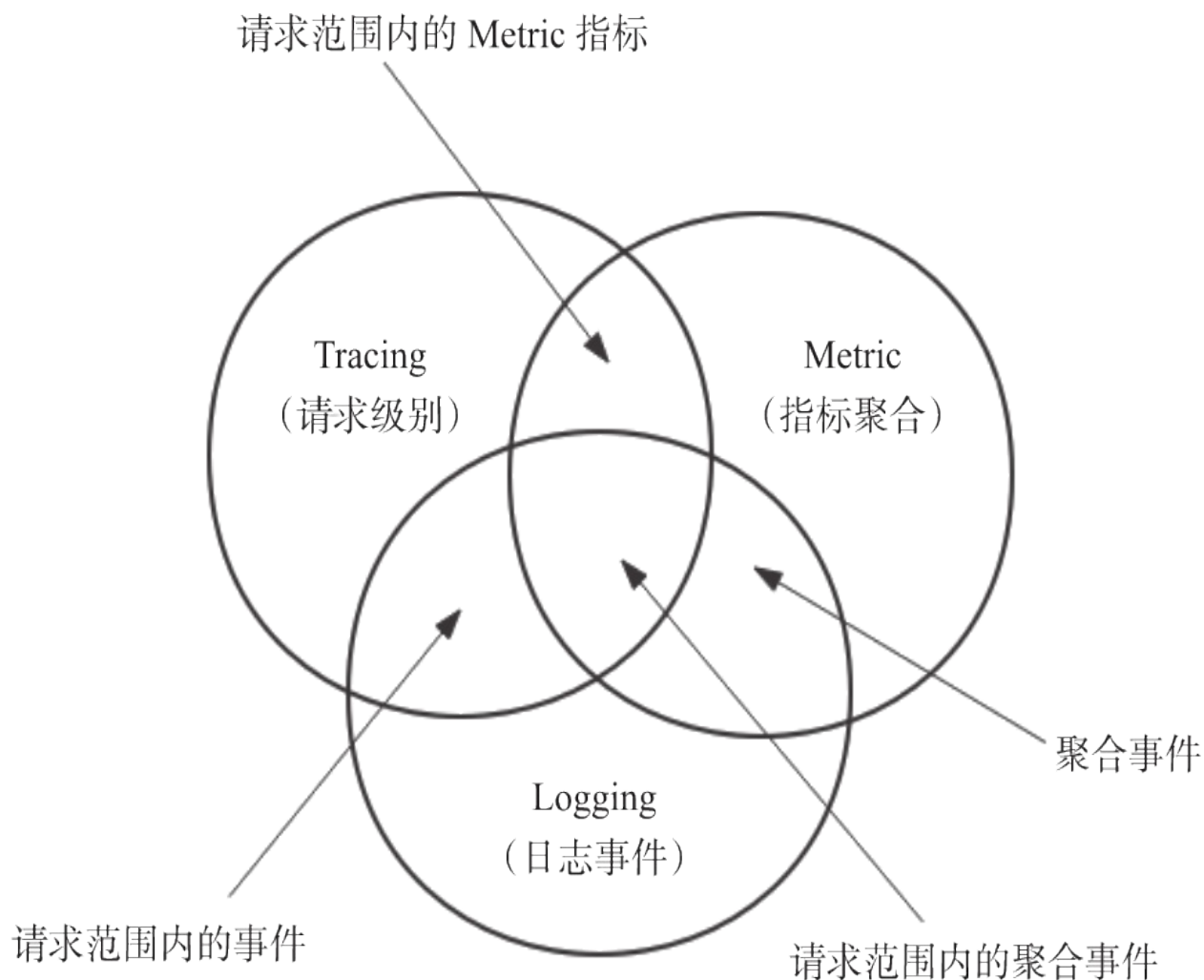


图2-2 可观测机制关系图

1. 标准化监控Metric

监控标准化是在业务中埋点主动上报标准业务指标来满足业务监控需求的解决方案。期望在业务完成标准上报之后，自动给业务提供调用关系拓扑、自动错误率监控、自动服务可用性计算、服务性能评估等高级功能。业务服务一般习惯于使用日志监控的方式，监控系统对日志的监控也容易实现，但日志监控有如下几个缺点。

(1) 规则简单

日志监控采集只能适用于简单规则，比如流量采集、接口平均耗时等，在耗时上采用平均耗时，在分析业务性能时缺少精细粒度的分位耗时信息，比较难科学反应模块性能。

(2) 业务量大时影响性能

日志监控分析在业务流量大的情况下，在线消耗性能比较多。

(3) 可维护性差

日志监控采用配置分析的模式，让监控系统和日志格式有依赖，如果修改代码的时候不小心调整了日志格式，忘记调整对应监控正则，会导致日志监控出错。

相比之下，标准化监控，主动上报监控指标，可以自动绘出调用链，为服务可用性提供衡量指标，支持复杂的统计功能，可以定制业务相关的复杂策略。

为了实现监控标准化，针对监控项的收集和上报，需要制定一个规范，如果各个业务或者一个业务的各个接口都按照上述的规范，那么可以在监控自动化上做到更高程度，避免人工配置。

举个例子：如果一个业务的各个接口都有错误率，且各个接口的错误率不能超过30%，那么只需要配置错误率 $\geq 30\%$ 就报警，则新增一个接口自动添加报警，可以实现报警自动化。监控项规范可以规定，所有上报的监控项均需要关注QPS、耗时和错误率这几个服务基础监控指标。

标准化项采集实现起来也比较简单，可以借鉴开源系统statsd的设计思想，在业务代码中引入监控标准化SDK。SDK负责实现监控项埋点和统计功能，以及服务间RPC调用质量等业务指标，SDK通过UDP协议上报业务监控指标项到本机的Agent，本机Agent以一定的周期（如10s）为周期聚合业务指标，并将聚合结果发送到监控平台。

2. 分布式跟踪Tracing

大型互联网公司的微服务架构下，一个业务可能会有上百个微服务组成，一次请求由几十次服务调用共同完成。这种架构下就会出现很多跨服务的问题，比如请求处理为什么突然变慢，问题具体出在那个环节；业务故障的排查，是哪个流程中出错等，通过分布式跟踪系统，可以建立一个以请求为中心的全局视图，查看请求处理过程中各个阶段的具体状态和处理情况。

2010年，Google发表了一篇名为《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》的论文，介绍了Google生产环境中大规模使用的分布式跟踪系统Dapper的设计原理和使用经验，Dapper掀开了分布式跟踪系统的序幕。之后Twitter根据这篇论文开发了自己的分布式跟踪系统Zipkin，并将其开源。现在不少国内外的大型互联网公司也都有自己的分布式跟踪系统，比如Uber的Jaeger、阿里的鹰眼、京东的Hydra、微博的Watchman等，这些分布式跟踪系统的原理基本都差不多，一般分为数据采集、数据传输、数据处理和数据展示4个方面，下面以Dapper为例，讲述分布式跟踪系统的大体实现。

首先看下分布式跟踪相关的几个重要概念，TraceID用于对请求进行唯一标识，一般是请求进入链路处理入口时生成（比如在接入层生成），此后一直在请求处理的整个链路中一直携带和传递，不再修改。TraceID是分布式跟踪系统最重要的概念，通过TraceID串联请求的各个阶段，同时也通过TraceID查询请求的整个处理链。

Span ID用于标识请求在一个模块内的处理过程，理论上讲，Span ID只需要做到同一个TraceID下唯一即可，不需要全局唯一。为了通过服务调用关系实现请求调用链信息的组织，调用下游服务时，需要将自身的Span ID和TraceID传递给下游服务，下游服务会将上游服务传递过来的Span ID作为自己的父Span ID，并通过一定的方式生成自身的Span ID，然后使用TraceID、父Span ID和Span ID生成标准化的Trace日志，分布式跟踪系统收集各个服务的Trace日志，然后通过TraceID完成一个请求对应Trace日志的收集，通过父

Span ID和Span ID完成请求跟踪树的建立。

Annotation用于给跟踪信息添加相应的注释，这个注释信息是带时间戳的，Annotation可以添加任意请求相关的信息，通过Annotation，方便进行一些业务相关的问题定位。

对于分布式跟踪系统来说，TraceID和Span ID的正确生成是一个非常重要的前提，不同分布式跟踪系统的ID生成规则不一样，对于TraceID来说，必要条件是全局唯一，同时为了方便根据TraceID进行问题定位和追查，TraceID生成时可以加入一些请求相关的基本信息。

对于Span ID的生成，必要的约束条件是保证同一个请求下的不同节点对应的Span ID唯一即可。不同的分布式跟踪系统，生成Span ID的方式也不尽相同，比如阿里的鹰眼系统巧妙用Span ID来体现调用层次关系（例如0，0.1，0.2，0.1.1等），对于请求的根节点来说，使用一个特殊的标识来标识当前节点是根节点，没有上游信息。

对于分布式跟踪来说，设计和使用过程中，有一些需要遵守的要点。

（1）低侵入接入

低侵入接入不仅是分布式跟踪，也是所有服务治理相关特性的共同需求。服务治理特性接入最好能做到对业务透明，这样接入的过程中不会有特别大的阻力，如果业务需要花费比较大的改造成本才能完成接入，业务接入的意愿就不会那么强烈。

为了实现透明低侵入接入，一般可以将分布式跟踪特性下沉到服务框架或者Service Mesh层面，尤其是Service Mesh可以做到让分布式跟踪特性的修改对业务服务完全透明，业务服务零改动。

（2）低开销

为了减少对业务服务正常运行的影响，分布式跟踪需要做到尽可能轻量，尤其是一些性能特别敏感的业务，需要随时调整Trace日志采集频率，甚至需要Trace日志采集开关。

（3）高稳定

分布式跟踪作为基础服务治理组件，如果没有足够的稳定性保障，会给业务服务带来非常严重的后果，因此分布式跟踪系统设计和实现过程中的各个环节都要考虑稳定性因素，尽可能简化设置。对于Trace日志输出来说，不要有太复杂的处理逻辑，同时日志收集代理尽可能轻量部署，减少不必要的依赖。

当前国内外主流互联网公司基本上都有自己的分布式跟踪系统，但这些系统实现各异，并且相互不兼容，无法实现跨业务、跨公司的全链路跟踪，制约了全链路跟踪的进一步发展，为了解决不同分布式跟踪系统API不兼容的问题，诞生了OpenTracing规范，OpenTracing是一个轻量级的标准化层，希望通过OpenTracing实现分布式跟踪的兼容互通和标准化。当前OpenTracing已进入CNCF，成为云原生基础设施的一部分，OpenTracing通过提供平台无关、语言无关的API，使得开发人员能够方便地替换追踪系统，目标是为全球分布式追踪提供统一的概念和分布式跟踪标准。

OpenTracing数据模型中，重点定义了两个相关的概念Span Tag和Log Field，其中Span

Tag定义一系列作用于整个Span过程的Tag；每个Span的log操作，都具有一个特定的时间戳（这个时间戳必须在Span的开始时间和结束时间之间），以及一个或多个field，Log Field定义了一系列标准化的Log Field字段，通过Span Tag和Log Field实现了分布式跟踪输出的标准化。

OpenTracing规范还在不断完善过程中，当前Jaeger和Zipkin均实现了对OpenTracing的支持。

2.5 量化分析体系

量化分析体系是基于数据对系统进行量化、定位和分析，然后产生相应的治理措施，指导接下来的线上治理和线下治理，接下来从度量、定位和风险分析这几个维度对量化分析体系进行展开讨论。

2.5.1 稳定性风险度量

稳定性工作包含的内容很多，在稳定性建设过程中，经常会遇到这样一个问题——服务当前的稳定性现状，不太好衡量和判断。比如，服务变更一直以来是影响服务稳定性的重要因素，由于没有明确的衡量标准和规范，很难说清楚服务变更做到什么程度才算是符合稳定性要求的变更，这样会导致执行服务变更的同学很难发现自己平常工作的不足，同时团队的管理者层面看不到自己团队变更风险的全貌和严重程度。服务变更只是稳定性建设的一个缩影，稳定性建设的很多维度和服务变更类似。

为了掌握稳定性建设的真实情况，同时引导和规范业务同学在稳定性建设时的做法，针对稳定性的一些重要环节，需要制订一定的度量标准，对业务同学日常的稳定性建设进行度量，明确告诉大家当前的稳定性工作处于什么水准，具体哪些地方需要改进。

以服务变更为例，流程规范规定高峰期不能上线，平台可以强制规定不能在这个时间上线，但一定会有紧急的需求需要在高峰期操作，所以平台还必须放开这个口子。这类操作带来的风险也是确实存在的，它不应该被经常执行；比如灰度检查通常暂停的时间越长越有可能发现问题，但平台只能约束至少有暂停或暂停一个固定时间，更长的暂停时间则由执行者把握。诸如这些问题，可以结合变更规范制定一个量化标准——更信用分标准。比如**变更信用分**标准可以规定，暂停时间越长分数越高，高峰期如有操作则需要扣分，同时不同优先级的模块采用不同的标准，最高优先级的模块有强制双重校验的要求，需要两个人校验后才能在平台上线，如果没有在平台执行这个要求也需要扣分。按照变更信用分标准，对每周的上线单进行系统分析，并按团队汇总、量化和排名，让大家能从全局的角度看到问题的总体情况和各自的严重程度，并能够从上往下索引到各个团队具体的变更单、变更人和变更参数，甚至直到具体的变更模块配置界面，以此促进各个团队有针对性地发现变更风险并清晰地知道如何进行完善。定期将每个团队本周的变更评分情况，以及存在变更风险的变更操作单，发送到对应团队的负责人与变更风险单的实际操作者，方便业务人员了解当前的变更风险，并及时针对性地进行改进。

针对稳定性建设的其他方面，我们也可以按照上述变更信用分的思路，建立相应的度量标准。比如**监控告警**，针对基础监控是否有遗漏配置、上下游依赖健康是否完备、告警策略是否符合要求等情况，可以推出监控健康分，量化服务监控告警的完备情况，引导用户进行监控告警的完善工作。比如预案建设方面，可以推出预案健康分，量化一个服务的预案建设情况，涉及降级限流等预案是否完备，预案执行是否符合灰度要求，预案是否可以回滚等。

对于稳定性度量来说，主要是确定具体的度量指标，以及每个度量指标的标准评分。其实具体评分并不是那么重要，关键是大家对度量标准能够达成一致，能够对稳定性日常工作有实际的量化指导效果。

2.5.2 基于多维度监控的故障定位

线上服务出现问题时，当有足够多的监控信息时，才能直观地定位问题。但随着业务规模变大，微服务的个数越来越多，链路、拓扑、网络越来越复杂，相应的监控事件越来越多。当出现故障，可能瞬间出现大量的报警信息，从众多告警中快速找到故障原因，确定相应的止损预案，是一个非常重要且有挑战性的事情。

出现故障时，首先需要确定故障的影响面，可以基于场景和分布式跟踪拓扑将业务组织成一个全局的“灭火图”，灭火图中包括所有核心服务的可用性指标，比如错误率、QPS、耗时等，出现故障时，先从灭火图中看出故障影响的业务和服务，接下来确定故障定位的范围。

为了从纷繁复杂的众多事件中定位具体的原因，可以将各维度的监控报警、各种变更事件以事件的方式，按照时间轴整合成一个时间线，有了事件时间线，我们就可以将关注焦点放到故障时间前一段时间内的监控告警事件以及变更事件上，从而根据具体的事件类型确定相应的预案和止损措施。

2.5.3 风险分析

基于线上实时的可观测数据，以及研发全流程的变更和操作数据，我们可以得出很多维度的报表和趋势数据。这些维度可以涵盖服务治理的各个环节，比如链路SLA、超时重试、容量管理、强弱依赖关系等，这些数据可以作为接下来分析的基础。

同时，根据之前的风险分析以及一些静态的服务元数据信息，会形成一个和当前实时治理数据对应的历史基准库，将当前数据和历史基准库进行比较，从中找到趋势和规律，进而发现潜在的风险。

实时治理数据分析后一些有价值的东西可以沉淀到历史基准库，作为后续风险分析的基础，进而形成一个风险分析的闭环机制。在基于风险分析的在线上治理和线下治理中，会结合具体的场景和实例进行全面的剖析。

2.6 线上治理

线上治理是根据量化分析的结果，通过相应的预案对线上服务的运行状况进行调整，保证线上服务正常运行，接下来讨论线上服务常见的预案，以及如何保证预案的自动触发和自动调整。

2.6.1 线上预案体系

故障快速定位和止损的理想处理方式是将故障定位和预案执行打通，当出现故障时，能够判断出故障的大体类型以及对应的预案，并触发预案的自动执行。

当然实际的故障处理过程中有很多地方需要考虑，虽然并不是所有故障都能提前建立相应的预案，但我们可以根据历史故障和一些先验知识将故障进行归类，建立相应的预案。另外，建立预案时应尽量方便执行和触发，如果不方便执行，很难短时间内处理故障，最关键的问题是判断预案触发的时机，以及当前是否应该执行预案。

线上服务稳定性故障大体可以归类为如下原因。

- 1) **变更引起的故障**。变更是稳定性故障的最主要来源，系统广义的变更源很多，最常见的服务变更一般包含应用变更、配置变更和数据变更。除了服务变更之外，环境和硬件的变化，比如网络带宽变化、机房链路变化等，也可以归为广义的变更范畴。
- 2) **流量和容量变化引起的故障**。这类故障对应于之前稳定性保障中分析的输入流量突变，如果服务提前没有足够的应对机制，会导致一定的稳定性隐患。
- 3) **依赖故障**。依赖服务故障会影响调用依赖服务的上游服务，依赖服务故障又分为强依赖服务故障和弱依赖服务故障，这两者会有相应的处理方式。
- 4) **机房、网络等硬件和环境故障**。硬件和环境故障的特点是没有办法预测，随机性和偶然性因素很大，并且一旦发生往往是系统级别的问题，会产生很严重的后果。
- 5) **其他**。比如ID生成器溢出导致的故障。

故障的场景化是指根据上述稳定性故障的大类，再细化出一些方便识别判断的场景，比如入口流量突增、接入层故障、强依赖服务故障、弱依赖服务故障等细分的场景。划分这些场景的目的是发生故障时，进一步识别故障的根因（不一定是最根本原因，而是从止损的角度进行归类）。因此可以对那些比较容易通过可观测性指标判断出故障的场景类型，并且方便制定相应的场景化预案的故障，进行场景化归类。

2.6.2 基于Metric的预案自动触发

对于降级、限流和冗余切流这几类比较明确的场景，可以基于Metric进行故障判断，并且和预案自动打通，以依赖服务故障为例，可以根据Metric成功率指标同环比变化，判断依赖服务是否有异常，如果通过Metric发现当前确实有异常，先查询变更管理平台，依赖服务当前是否有相关的变更操作。如果有变更，建议依赖服务接口人立即进行变更回滚操作；如果没有变更操作，再判断当前对依赖服务的调用是强依赖还是弱依赖，如果是弱依赖，可以启动自动降级预案对依赖服务进行降级，如果是强依赖，降级肯定不能解决问题，可以通过预先制定好的冗余切换预案，启动服务级、集群级或者机房级别的流量切换。

基于Metric的场景和预案打通，目标是朝着故障定位自动化和智能化的方向演进，但需要根据实际情况逐步推进，对于一些不太容易判断的场景，建议谨慎操作，避免可能的误判，同时要定期对预案进行演练，保障预案触发的有效性。

2.6.3 治理参数动态调整

限流降级和切流是服务治理的3个利器，分别解决上游、下游和服务自身故障问题。这些服务治理特性和实现本身并不复杂，使用时的关键和难点在于触发的具体时机，比如什么时候启动限流、什么情况下启动降级等，下面以限流为例详细讨论如何设置治理参数以及限流的触发时机。

以登录服务为例，登录过程中需要使用加密算法对密码进行校验，一次登录过程CPU耗时1s，登录服务线上使用的服务器是40核，因此一台机器上登录服务的QPS大概是40左右。当某一秒有200个请求进到登录服务，我们希望服务可以在接下来的5s内，每秒完成40个请求，这很好理解，也符合常理，因为这样才能达到稳定输出的服务目标。但事实符合我们上面的预期吗，我们以一个简单的golang程序为例来看看。

```
package main

import "fmt"
import "golang.org/x/crypto/bcrypt"
import "time"
import "runtime"

func test(cost int, id int){
    fmt.Println(time.Now(), "BEGIN test ", id)
    code, _ := bcrypt.GenerateFromPassword([]byte("password"), cost)
    fmt.Println(time.Now(), "END test ", id, code[0])
}

func main() {
    runtime.GOMAXPROCS(1)
    for i:=0; i<5; i++ {
        go test(17, i)
    }
    time.Sleep(1e16)
}
```

程序运行完，结果如下：

```
2019-01-15 00:24:46.451649842 +0800 CST BEGIN test 1
2019-01-15 00:24:46.451685368 +0800 CST BEGIN test 2
2019-01-15 00:24:46.451693975 +0800 CST BEGIN test 3
2019-01-15 00:24:46.451699783 +0800 CST BEGIN test 4

2019-01-15 00:25:36.89918498 +0800 CST END test 0 36
2019-01-15 00:25:37.090741796 +0800 CST END test 2 36
2019-01-15 00:25:37.098754534 +0800 CST END test 3 36
2019-01-15 00:25:37.183491458 +0800 CST END test 1 36
2019-01-15 00:25:37.18871652 +0800 CST END test 4 36
```

通过观察上面的输出日志，我们可以看出结果并不是每秒稳定输出，而是在一段时间后，这一批请求都在很接近的时间被处理完。和我们上述的预期不太相符。这涉及Golang的协程调度，并不是等到协程运行到阻塞点才交还CPU，这在逻辑上也是正确且应该的，目的是防止协程实现存在问题而将CPU全部消耗掉。但这个逻辑对于CPU密集型的服务而言，在遇到尖峰流量时，就会产生我们不希望的结果。

回到上面讨论过的具体例子，200个请求进入服务后，如果上游设置的超时时间比5s短，这200个请求对外就全部超时了。也就是说，只要有超出CPU处理能力范围的大流量

到达登录服务时，如果不做特殊处理，这些请求无一例外全部超时。这个问题会影响CPU密集型的Golang服务，采用类似调度策略的Java/C++服务也有类似问题。

具体如何解决呢？很容易想到用单接口限流来解决。通过处理能力计算出每秒最多能接收的请求量，超出就抛弃请求，防止大流量对服务质量的冲击。这就是尖峰限流。

对于登录服务来说，可以把限流阈值设置为40QPS，超出部分就丢弃，通过单接口限流解决这个问题，但这个方案的问题也很明显：如果高CPU占用接口在梳理时被遗漏了，该接口消耗了绝大多数CPU，导致限流失效；同时新增加的接口，靠人和经验来评估限流值，也很容易遗漏设置。

继续思考，我们会发现这个方案最重要的问题：接口不止有一个，而它们使用相同的CPU资源。那每个接口允许的限流值怎么设置，到底要给其他接口留多少余量呢？

以登录服务为例，如果设置成40QPS就是不给其他接口留余量，但由于前述的Golang CPU调度特性，极端情况下只要有其他接口流量，所有登录请求都会超时，整体有效吞吐变成零。那应该设置成多少，20、10？我们会发现无论多少都不适合，因为不知道极端情况下流量特点到底是什么样的。也就是说，单接口限流可以满足突发流量的需求，但不足以完成极端情况下的流量调度需求。如果一个服务同时有I/O消耗型接口与CPU消耗型的接口，通过简单的接口限流方案（即手工设置所有接口的最大QPS）无法兼顾效率与吞吐，且存在严重的维护问题（说不定哪天某个新接口就忘记设置限流了）。

对于同时有I/O消耗型接口与CPU消耗型接口的服务，我们可以通过部署拆分将CPU消耗型接口和I/O消耗型接口独立部署，让它们互不影响。但这样并不能从根本上解决问题。因为计算型的接口不止一个，总不可能每一个计算型接口都单独部署集群。只要设置过的人都会遇到这个疑惑：到底要设置成多少合适？每个接口的限流设置稍微高一点，请求压力一大系统就可能总体过载。因此考虑到安全原则，实际情况中接口限流都是设置成非常安全但低效的阈值，但这样会导致大量的机器资源浪费，因此，我们必须抛弃这种安全但低效的阈值管理方式，寻找一种更加科学合理的流控方式。

再细化一下流控的需求，我们需要的流控方案应该是：足够安全、高效，能保证持续吞吐，能充分利用CPU的。其中的核心是持续吞吐和高效。最后实际产出的技术方案是多重级联的自动流控，具体设计思路大体如下。

（1）带反馈机制的动态限流

设计思路：任何接口在极限流量下，只要超出处理能力，就一定会出现接口超时，但因为突发流量的控制，对外吞吐能力是稳定的。只要超时率超过一定比例，就启动阈值的上升和下降，控制进入系统的请求包数量。

（2）CPU限制

为了控制程序对CPU的使用，留给其他系统进程（如各种Agent），同时为了防止通过流量阈值一次进入太多的请求，我们控制最多起40个Golang协程用于处理CPU消耗型的接口（这也是我们的突发限流）。其他请求排队等待。这里面需要注意的是，出队时要注意时间，如果请求已经超时需要直接抛弃，但这个时间不需要很长，同时需要结合SLA，考虑在秒级平缓瞬时请求流量。

(3) 过载保护

过载保护，即实际采用方案的连接数超过一定阈值就主动拒绝请求。这很容易理解，但里面有个小细节是拒绝的方式：不能简单地忽略请求，而应该快速断开连接或者回复服务器当前太忙消息给调用方。

2.7 线下治理

线下治理是根据量化分析的结果，对稳定性治理、架构治理等进行调整，下面从链路治理和架构治理相关的实例出发，看看如何通过量化分析指导线下治理工作。

2.7.1 链路稳定性治理

1.链路风险分析的重要性

链路风险分析就是从链路通信的历史数据出发，分析出链路当前存在的风险，减少链路通信的隐患，提高系统的整体稳定性。链路风险分析可以解决的问题域很多，如超时时间设置是否合理、重试次数设置是否合理、服务SLA指标设置是否合理、服务的强弱依赖关系是否符合预期等，服务通信相关的一大部分故障是由于链路风险导致，可以通过链路风险分析的方式提前发现和解决，避免故障的发生。

链路风险分析的基础是链路实时拓扑图和Metric指标，下面会针对几种典型的链路风险，进行详细的分析。

2.超时和SLA风险

客户端访问服务端的超时时间配置，和实际访问情况不符，是非常常见的一种链路风险。上游服务的超时配置过小，会导致一些本来可以正常返回的请求超时，影响服务的SLA和正常的服务体验；超时配置过大，会导致下游服务故障时上游服务超时等待的时间太长，严重时会把整个系统拖垮。因此，超时设置直接关系到系统的稳定性，需要有相应的机制指导服务的超时设置，并及时发现线上系统中的超时配置隐患。

超时配置风险主要有两种：一种是超时时间和实际不相符；还有一种是上下游的超时设置不匹配，比如有A、B、C 3个服务，服务A访问服务B，服务B访问服务C，但实际业务中经常会遇到服务A访问服务B的超时时间比服务B访问服务C的超时时间小的情况。

上述两种超时配置风险均可以通过链路风险分析的方式解决。思路其实很简单，Service Mesh收集服务间通信的Metric指标数据，通过Metric数据可以分析出服务访问当前的各分位耗时，比如99.99分位耗时是50ms，即99.99%的请求都能够在50ms范围内正确返回，我们只需要根据业务实际情况，把要求的分位耗时（如业务要求4个9的成功率，也就是99.99分位）和超时配置进行比较，如果差异过大，则存在超时配置风险。可以定期将当前存在相应风险的配置通过一定的方式知会业务人员，如果业务人员判断确实有问题，就可以尽快对这个风险进行修复。

实际超时配置修复时，可以使用上述分位耗时作为基数，加上一定的偏移作为Buffer，按照之前的经验值，基数较小，可以加上固定时间的偏移（如5~50ms，具体情况设置），如基数较大时，可以使用固定倍数的偏移（如1.1~1.5倍）。

SLA风险也可以采用同样的思路进行分析，这里不再详细展开。

3.强弱依赖或重试风险

微服务之间通信，如果该链路通信失败会导致整个请求处理失败，一般将这两个微服务之间的关系称之为强依赖，反之称之为弱依赖。我们可以基于服务的强弱依赖关系，进行降级、熔断等处理。

强弱依赖服务风险，是指链路通信之间的关系和预期不符。比如，服务A调用服务B

的链路是弱依赖，但随着需求迭代业务逻辑发生变化，可能不经意间服务A调用服务B的链路实际上变成了强依赖，但大家仍然按照之前的先验知识，把它当成弱依赖，这就是一个很大的风险点。特别是链路故障时，基于弱依赖的前提进行降级等操作时，就可能会酿成悲剧，导致整个系统不可用。因此需要有相应的机制，定期检测出当前链路关系的风险点。

强弱依赖风险分析可以采用故障注入的思路，但与直接在线上系统进行故障注入和演练不同，为了不影响线上服务的稳定性，可以通过线下环境进行强弱依赖风险检测。具体思路是采用故障注入配合自动化测试，通过Service Mesh获取到系统的拓扑关系图，自然就可以得到所有的调用链路以及具体的链路关系，这个可以称之为基准链路关系表。针对每个调用链路，通过故障注入的方式让链路调用返回失败，构造一个测试用例，在该测试用例基础上执行正常的自动化测试。如果自动化测试成功，证明该链路失败对整个流程没有影响，说明该链路是强依赖，反之是弱依赖。通过这种方式，可以判断出每个链路最新的链路关系，然后和基准链路关系进行比较，如果有差异，说明链路关系发生了变化，产生了新的链路依赖风险。

重试风险也可以采用同样的思路进行分析，这里不再详细展开。

4. 集群或拓扑风险

集群或拓扑风险是风险分析的一个主要来源，之前的工作中遇到过不少集群或拓扑风险的真实案例。比如，某线上集群的一些机器因为保修临时下线一段时间，但机器修复好之后遗漏挂载，导致部分机器白白闲置；服务A调用服务B本来是同机房调用，由于故障或者流量切换演练等原因临时将调用关系切换到调用其他机房的的服务B服务，但事后没有切回来，导致服务A调用服务B一直是跨机房访问，影响用户体验和系统稳定性；某服务S线上部署时没有考虑到地理位置因素，将过多的服务节点部署到同一个交换机下，交换机故障时导致服务同时有多个节点不可用，可用节点数不足导致服务雪崩。

集群部署风险或者拓扑风险没有通用的解决思路，需要针对具体的风险增加相应的风险应对措施，以上述列举的服务下的多个节点部署在同一个交换机下为例，一般在同一个交换机下的机器，IP网段信息差不多，可以根据类似的信息，通过程序定期扫描所有的线上集群，根据集群节点的地址信息，判断是否存在拓扑部署风险。

5. 链路调用风险

链路实时拓扑数据是一个宝藏，可以从中慢慢挖掘出链路调用层面的很多风险。比如，当前服务调用超过20个下游服务，扇出过大，不太符合微服务的设计准则，可以考虑是否有必要进行进一步的拆分。

微服务架构中，单个请求的链路特别长时，会带来一定的性能问题，因此可以从全局链路拓扑中将TOP10长的链路，或者链路深度超过6的链路列出来，反馈给业务人员，看看是否有必要进行架构层面的调整。

微服务拆分和设计过程中，建议不要出现两个微服务相互依赖的场景，可以通过链路拓扑图查找当前是否存在成环的链路，如果成环，说明服务之间产生了相互依赖，可以将类似风险反馈给业务人员进行整改。

6. 链路风险分析的方法论思考

链路风险分析是发现风险并对风险进行抽象和建立自动化检测机制的过程，本质上是对稳定性风险进行精细化管理的系统性工程，需要长期持续地建设下去。

发现风险是链路风险分析的第一步，为了能够持续发现系统新的风险，建议将风险分析和稳定性反模式结合起来：①根据系统中出现的重大故障，以及之前积累的容易出现故障的一些典型问题，梳理出稳定性反模式，也就是稳定性实践中容易犯的，不应该出现的一些模式；②判断这些反模式是否能够通过自动化的方式检测出来。

同时为了方便新增风险的检测，可以建立一个完善的风险分析框架，具体包含风险现状、风险改进闭环、风险报告、风险自动通知机制等，新增的风险分析就直接基于框架开发，相当于增加个插件，可以大大提高风险分析的效率。

2.7.2 架构与资源治理

业务迭代过程中，可能会出现很多临时性的调用关系，调用关系使用一段时间后不再使用，长时间积累下去，可能会导致服务存在不少冗余代码，影响服务的可维护性。我们可以通过实时拓扑图和静态的基准拓扑图进行比较，如果出现基准拓扑图中存在，而实时拓扑不存在的服务调用链路，说明当前链路不再使用，基于实时拓扑和基准拓扑的差异比较，可以考虑将这种冗余调用代码去除。

为了保证变更的质量，服务在新增特性时通常会给它配置相应的特性开关，通过开关来区别不同版本的代码，随着架构的迭代升级，可能会导致不少版本的代码和分支不再使用，我们可以采用和上述冗余链路检测类似的方式。当检测到特性开关判断总是返回特定值时，说明其他分支当前已经不再使用，可以提示相关人员，这类无用代码是否可以移除。

我们也可以从资源调用和使用报表中发现当前服务资源是否有使用不当的地方，比如缓存资源遗漏设置过期时间，缓存资源设置的过期时间过大等；如果存储资源出现过大的对象，或者长期不再使用的数据，也可以反馈给业务方相关人员，以确定使用上是否需要进行调整。

2.8 服务治理演进

服务治理当前有远程Proxy方式、智能客户端方式以及本地Proxy这3种主要承载方式，下面分别讨论这几种方式的优缺点。

2.8.1 远程Proxy方式

微服务下的服务框架基本解决了微服务开发和测试的效率问题，业务人员可以把精力聚焦投入到业务需求上面。多语言服务化背景下，服务治理的基础设施的每一次改动和升级，需要多语言框架的同时升级，这里多语言服务框架的开发、测试和维护的开销有很大的工作量。

由于服务框架以Lib的形式存在，和业务代码存在于一个服务里面，会导致做业务需求的业务人员和负责服务治理的基础设施人员之间仍然会有很多沟通和交互，随着组织和团队的扩大，受康威定律的影响，这个沟通成本会变得不可控。

如何减少多语言接入场景下服务治理基础设施的维护开销，同时减少服务治理基础设施升级对业务的影响，尽量将服务治理和业务解耦，这是微服务架构面前的一个课题。

客户端和服务端通信，客户端和服务端都需要有不少服务治理相关的考虑，比如客户端访问服务端时需要考虑服务发现、流量路由、负载均衡、调用熔断、请求降级、mock支持、请求跟踪等诸多特性，服务端也需要考虑限流、降级、安全、统计等。随着业务迭代发展，客户端和服务端需要支持的服务治理特性越来越多，这会影响业务迭代效率。架构设计上有个不成文的说法，任何软件工程遇到的问题都可以通过增加一个中间层来解决。按照这个说法，可以很自然地想到，微服务架构下，是否可以在客户端和服务端中间增加一个中间层，避免两者直接通信，同时把服务治理相关功能从客户端和服务端代码解耦，移到中间层来，这样服务治理特性的变化收敛在中间代理层，对客户端和服务端透明。

API网关就是用来解决上述微服务治理的效率问题。**API网关**封装了服务访问和服务治理相关的具体逻辑，客户端只需要使用简单的访问方式，统一访问**API网关**，由**API网关**来代理对后端服务的访问，同时由于服务治理特性统一放到**API网关**上面，服务治理特性的变更可以做到对客户端透明，一定程度上实现了服务治理等基础特性和业务服务的解耦，服务治理特性的升级也比较容易实现。

为了简化客户端的访问方式，对调用方屏蔽集群访问的复杂度，**API网关**一般会提供一个**VIP**，调用方直接使用**VIP**进行访问，由负载均衡设备负责**VIP**到**API网关**地址的映射。

API网关和直接使用服务框架相比，优点是业务使用起来很简单，没有什么入门成本，非常容易上手，对业务来说，简单和效率往往是第一位的；同时**API网关**也可以屏蔽多语言调用方式使用方式的差异，避免了微服务框架语言级别的限制，多语言调用方均可以使用简单一致的调用方式访问后端服务，基本解决了多语言的服务治理问题。

当然**API网关**在提供便捷的服务访问和服务治理能力的同时，相应地也会有一些问题。

首先，引入**API网关**，通信层面会增加一跳；如果是采用简单易用的**VIP**访问方式，还需要加上用于**VIP**解析的负载均衡服务这一跳，所以通信层面上比原来直接访问后端服务增加了二跳，性能上肯定会有一定的损耗。

其次，通信链路上多一跳就会多一个故障点，会对系统的整体稳定性和可用性有一定的影响。

最后，API网关简单易用的同时，灵活性和定制化支持不太好，很难支持复杂的服务治理定制化需求，比如服务如果想支持动态路由切换，API网关支持起来就有点力不从心。

因此，在业务相对简单，或业务线上部署结构不太复杂时，使用API网关是一个很好的选择，可以简化微服务的开发和运维，提高业务迭代的效率。但业务如果发展到比较复杂时，比如生产环境有多个机房，或者同一个机房内部有全流量环境、小流量环境等多套环境，就会有比较多的路由策略需求和动态路由切换需求，这种情况下就需要探索服务层面是否可以有一种对效率和扩展性更好的平衡方式。

2.8.2 基于智能客户端的服务框架

和远程Proxy不同，基于智能客户端的服务框架采用和服务端直连的方式，客户端和服务端直接通信，中间不经过任何节点，不仅性能提升，也增强了稳定性。

服务框架需要支持完善的流量调度和容错设计，同时需要支持常见的服务治理策略，对技术的要求相对较高，对于中小公司来说，开发或维护一款完善的服务框架的开销都是不小的。

除了开发维护成本高之外，服务框架作为Lib和客户端服务绑定在一起，因此是语言相关的，所以每个语言均需要相应的服务框架，这会带来很高的多语言服务治理成本。

另外，由于和客户端服务绑定在一起部署，服务框架的迭代升级相对麻烦，当服务框架进行升级时，使用框架的服务均需要进行升级，对于广泛使用的服务框架来说，这是笔不小的开销，框架和微服务的绑定和耦合，会影响框架自身的迭代和创新。

2.8.3 本地Proxy

API网关把服务治理相关特性从服务框架中解耦出来，可以提高效率，但灵活性和扩展性稍差；服务框架可以自如地支持各种服务治理需求，但多语言服务治理支持方面会有很大的开销，同时服务治理特性的升级和维护比较困难。因此，是否有一种机制或方式，能兼顾API网关和服务框架各自的优点，在提高业务开发效率的同时，也能很好地支持微服务治理的扩展性和定制化需求，本地网关应运而生。

本地网关又称为Sidecar，通过在每个服务实例对应的本地机器上部署实现服务治理功能的代理，来实现服务治理能力的快速落地。最有代表性的是Netfilx的微服务中间件Prana，初衷是将Netfilx OSS微服务治理套件的能力，通过HTTP API接口的方式，赋能给非Java语言使用，以很小的开销获得Netfilx OSS体系强大的服务治理能力。受到Netfilx Prana的启发，Spring Cloud也推出了Spring Cloud Netflix Sidecar。

和API网关相比，本地网关的访问方式更为简单。①事先给本地网关绑定固定的端口号，业务服务通过“localhost: 本地端口号”的方式即可访问，虽然和直接访问服务端相比多了一跳，由于是本地访问，比API网关的性能损耗要小。②由于本地网关每个机器上都会有部署，因此API网关单个节点故障对系统稳定性的影响相对较小。③本地网关和服务部署在一起，具有感知本地部署环境的能力，方便一些业务或者环境相关的服务治理特性的落地，比如机房级别的动态路由调整等。

从效率和扩展性上看，本地网关和API网关相比，都有着不少优势。那么是否意味着本地网关就是一个没有问题的完美方案呢，其实本地网关最大的问题是运维上的复杂度比较高，本地网关节点很多，并且业务服务对本地网关也是强依赖，因此需要对本地网关的部署、监控、升级、高可用保障等，都有一套完善的机制来保障。由于本地网关对运维的要求比较高，需要有一套完善的运维工具体系支撑，之前很长一段时间，这种模式在一些互联网公司中得到采用，如Netfilx、Airbnb等，并没有大规模应用起来。

2.9 理想的服务治理架构

为了支撑微服务架构，提高微服务迭代效率，理想的微服务基础设施需要具备如下几个特点。

（1）标准化

标准化的微服务基础设施，保证了微服务架构的通用性、可维护性和可移植性，同时基于标准化的微服务基础设施，可以更方便地实现微服务自身的标准化。

（2）配置化

配置化是保障微服务治理和微服务稳定性、敏捷性的关键，理想的微服务基础设施，需要通过统一的治理平台，对服务治理的效果进行可视化度量，同时可以通过可视化界面方便地调整服务治理的策略，即时生效。通过配置化，可以建立服务治理策略调整和反馈的闭环机制，支撑服务治理特性的快速迭代。

（3）服务化

如果可以将微服务基础设施服务化，通过标准化接口获取服务访问和服务治理能力，可以真正做到服务治理对业务的解耦合与透明，实现服务治理和微服务的并行开发和升级。

（4）平台化、产品化

平台化、产品化的微服务基础设施，可以将服务治理能力通过开放平台，赋能给更多的微服务用户，通过微服务基础设施的产品化，可以加快微服务基础设施的创新。

2.10 本章小结

本章概要地介绍了服务治理的工作范畴、典型痛点以及解决方式，同时着重讨论多语言服务治理的一些调整。通过本章的学习，既全面了解服务治理的复杂度，又可以更好地理解Service Mesh的产生背景。

第3章 下一代微服务框架Service Mesh概要

第3章总体介绍Service Mesh相关的一些重要知识，为什么Service Mesh是下一代微服务框架，Service Mesh解决了微服务架构的哪些痛点需求，Service Mesh当前还有哪些局限和挑战，本章将围绕这些话题展开，逐个进行剖析和阐述。

3.1 Service Mesh基础

3.1.1 什么是Service Mesh

作为Service Mesh技术探索和实践的先行者，全球第一个真正的Service Mesh项目Linkerd负责人、Buoyant公司创始人兼CEO William Morgan第一次完整地阐述了Service Mesh。按照William Morgan的定义，Service Mesh是一个致力于解决服务间通信的基础设施层，其负责在现代云原生应用的复杂服务拓扑下实现请求的可靠传递，在实践中Service Mesh通常实现为一组轻量级网络代理，这些代理与应用程序部署在一起，并且对应用程序透明。

从上述Service Mesh的定义看，**基础设施层是Service Mesh的定位**，致力于解决本书第1章提出的微服务基础设施标准化、配置化、服务化和产品化问题；**服务间通信是Service Mesh技术面对的问题域**，对微服务屏蔽通信的复杂度，解决微服务的通信治理问题；请求的可靠传递是Service Mesh的目标；轻量级网络代理是Service Mesh的部署方式；对应用程序透明是Service Mesh的亮点和特色，Service Mesh接入对业务无侵入，可以非常方便地获取Service Mesh带来的便捷性，算是Service Mesh的一大优势。

综合来看，Service Mesh主要解决用户如下3个维度的痛点需求。

1.完善的微服务基础设施

Service Mesh通过将微服务通信下沉到基础设施层，屏蔽了微服务处理各种通信问题的复杂度，可以看成是微服务之间的抽象协议层，抽象层面可以看成是TCP/IP协议栈的一部分。对于微服务的开发者来说，比如当前使用HTTP或者Thrift进行RPC通信时，你不需要关注TCP/IP这一层的具体实现；有了Service Mesh之后，微服务也不再需要关注RPC通信（包含服务发现、负载均衡、流量调度、限流降级、监控统计等）的一切细节，真正像本地调用一样使用微服务，通信相关的一切工作直接交给Service Mesh。

因此，对于一些需要通过微服务改造提升业务敏捷性，但没有相应技术能力的中小团队来说，可以借助Service Mesh提供的完善微服务基础设施，加速微服务的落地。

2.语言无关的通信和链路治理

功能上，Service Mesh并没有提供任何新的特性和能力，Service Mesh提供的所有通信和服务治理能力在Service Mesh之前的技术中均能找到，比如Spring Cloud就实现完善的微服务RPC通信和服务治理支持。**Service Mesh改变的是通信和服务治理能力提供的方式，通过将这些能力实现从各语言业务实现中解耦，下沉到基础设施层面，以一种更加通用和标准化的方式提供，屏蔽不同语言、不同平台的差异性**，这样不仅有利于通信和服务治理能力的迭代和创新，业务使用的时候也会更加方便。

Service Mesh避免了多语言服务治理上的重复建设，通过Service Mesh语言无关的通信和服务治理能力，助力多语言技术栈的效率提升。

3.通信和服务治理的标准化

1) **微服务治理层面**，Service Mesh是标准化、体系化、无侵入的分布式服务治理平台。

2) **标准化方面**，Sidecar^[1]成为所有微服务流量通信的约束标准，同时Service Mesh的数据平面和控制平面也通过标准协议进行交互。

3) **体系化方面**，从全局考虑，提供多维度立体的微服务可观测能力（Metric、Trace、Logging），并且提供体系化的服务治理能力，比如限流、熔断、安全、灰度等；最为重要的是，Service Mesh通过透明无侵入的方式提供全面的服务治理能力，对微服务本身不会带来直接影响。

通过标准化，带来一致的服务治理体验，减少多业务之间由于服务治理标准不一致带来的沟通和转换成本，提升全局服务治理的效率。

[1] Sidecar,中文上一般被译为“挎斗”或“边车”，类似于三轮摩托车上的挎斗。

3.1.2 Service Mesh的基本模式

根据Service Mesh的发展历程和使用方式，我们可以把Service Mesh划分为两个模式。

1.Sidecar模式

在Service Mesh发展早期，Service Mesh以Sidecar的形态存在。Sidecar模式可以看作是第一代Service Mesh，代表有早期的Linkerd和Envoy。Sidecar模式下，网络代理服务在微服务旁边，为微服务提供通信和链路治理功能。因此，数据平面代理服务也经常被简称为Sidecar。

此时，只有数据平面的网络代理服务没有控制平面，和外部基础设施服务的交互直接在网络代理服务中进行。

第一代Service Mesh通过采用Sidecar模式，将通信和通信链路治理功能从微服务中剥离出来，实现了通信基础设施的下沉和服务化，这里也体现了架构解耦的思想，通过解耦减少了微服务的负担。

2.第二代Service Mesh模式

Sidecar模式的Service Mesh有一个突出的问题，将通信和通信链路治理的所有功能都放到这个代理服务中，导致数据平面代理很重，并且由于承载了太多的特性和功能，使得数据平面代理的更新和修改特别频繁，频繁的更新和升级会导致代理服务出问题的概率增大，影响代理服务的稳定性。同时，Service Mesh模式下，数据平面代理承载了微服务通信的全部流量，对稳定性要求极高，这个服务的任何故障都会对整个系统的稳定性产生很大的影响。为了解决上述频繁升级和稳定性之间的矛盾，将策略和配置决策逻辑从代理服务中脱离出来，形成了独立的控制平面，这就是第二代Service Mesh。

第二代Service Mesh最重要的标志就是控制平面和数据平面分离。数据平面和控制平面并不是新的概念，路由器/交换机等数据通信产品架构上，就有运行于专门处理器上的控制平面和多个独立运行、用于路由或交换功能的数据平面。SDN（Software Defined Network，软件定义网络）将数据平面和控制平面分离，控制平面具有可编程性，使得网络更加智能、灵活和易扩展，激发了网络技术的又一次革命。

第二代Service Mesh借鉴了SDN的思路，基于控制平面和数据平面分离思想，有了完善的控制平面：①所有的代理服务都由控制平面掌控，因为控制平面可以控制整个系统，所以提供了强大的控制能力和策略能力；②将具体的控制逻辑从数据平面移除，简化了数据平面的设计，数据平面不需要和外部系统进行交互，数据平面完全聚焦在变更频率很低的流量路由和转发逻辑上，提升了数据平面的稳定性。

3.Service Mesh架构

第二代Service Mesh的基本架构大致如图3-1所示。

架构上分为数据平面和控制平面两个部分。

（1）数据平面

数据平面负责代理微服务之间的通信，具体包含RPC通信、服务发现、负载均衡、降级熔断、限流容错等，数据平面可以认为是将Spring Cloud、Dubbo等语言相关的微服务框架中通信和服务治理能力独立出来的一个语言无关的进程，并且更注重通用性和扩展性。在Service Mesh中，不再将数据平面代理视为一个个孤立的组件，而是将这些代理连接在一起形成一个全局的分布式网络。

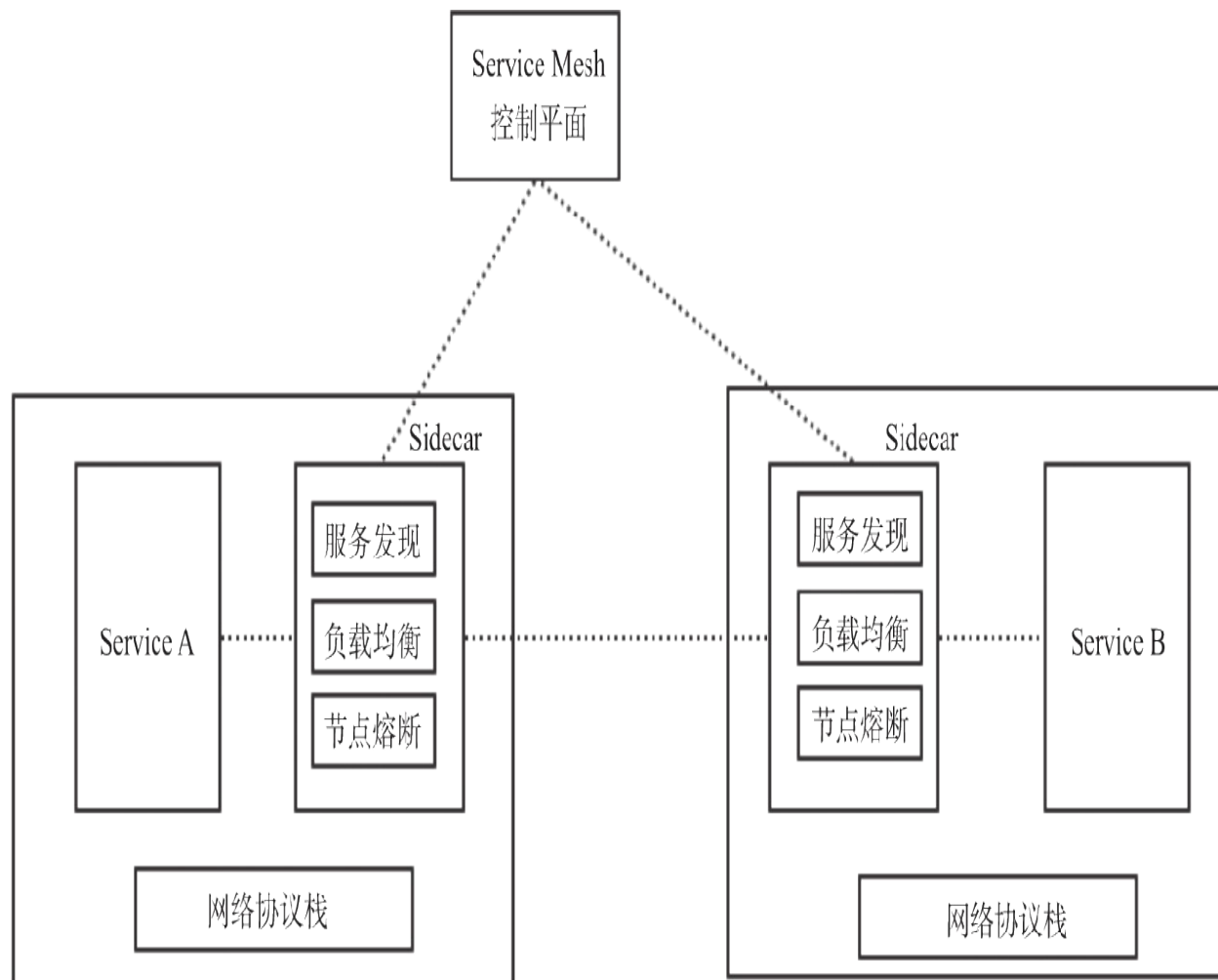


图3-1 经典Service Mesh的基本架构

（2）控制平面

控制平面负责对数据平面进行管理，定义服务发现、路由、流量控制、遥测统计等策略，这些策略可以是全局的，也可以通过配置某个数据平面节点单独指定。控制平面通过一定的机制将策略下发到各个数据平面节点，数据平面节点在通信时会使用这些策略。

3.2 Service Mesh的发展历程

下面大体梳理下Service Mesh从诞生到现在的发展历程，以对Service Mesh的发展脉络和当前现状有大致地了解。

2016年1月，Twitter前基础设施工程师William Morgan和Oliver Gould组建了一个名为Buoyant的公司，同时在GitHub上提交了第一个Linkerd版本Linkerd 0.0.7。这是业界第一个Service Mesh项目，标志着Service Mesh的诞生。

2016年9月，开发Linkerd的Buoyant公司第一次提出了Service Mesh的概念。

2016年9月，Envoy 1.0版本发布，Envoy诞生的时间要比Linkerd还要早些，只是在Lyft内部使用，一直没有对外发布，这次直接发布1.0的可用版本。

2017年1月，Linkerd加入CNCF，成为CNCF的第一个Service Mesh项目。

2017年4月，Linkerd 1.0版本发布，标志着Service Mesh正式进入可用阶段。

2017年5月，Istio 0.1版本发布，Istio一发布，就对Service Mesh界产生了巨大影响，直接导致Service Mesh之后的发展路径发生了一系列的变化。

2017年7月，Linkerd迫于Istio的压力，宣布和Istio集成。

2017年9月，Envoy加入CNCF，成为继Linkerd之后又一个Service Mesh项目。

2017年9月，在Nginx大会上，Nginx发布了0.1.6版本的nginMesh，nginMesh的定位是Istio的数据平面，nginMesh可以说是Nginx在Service Mesh上的试水之作，从发布后一直没有大的进展和动作，当前仍然无法应用于生产环境。

2017年9月，Envoy加入CNCF，成为继Linkerd之后又一个Service Mesh项目。

2017年12月，Buoyant在KubeConf上发布了Conduit 0.1.0版本，由于Linkerd受到了Istio的压制，Buoyant希望Conduit可以和Istio进行竞争，Conduit的架构和Istio差不多，但相比之下更为聚焦，只支持Kubernetes平台。

2018年7月，Istio 1.0版本发布，进入生产可用阶段，同时Istio作为Service Mesh领域的平台和霸主地位已经不可动摇。

2018年8月，Buoyant发布最后一个Conduit版本Conduit 0.5，Conduit至此走进历史，后续会并入Linkerd 2.0。

2018年9月，Linkerd 2.0 GA版本发布，该版本提出了Service Ops的概念，标志着Linkerd正式退出了Service Mesh平台的争夺，成为Service Mesh解决方案提供商。

2018年11月，Envoy成为继Kubernetes、Prometheus之后第三个从CNCF毕业的项目。

2018年12月，亚马逊AWS推出了Service Mesh解决方案App Mesh，标志着云计算领域

的绝对王者AWS开始进入Service Mesh领域。

2018年12月，Envoy和API Gateway提供商Kong同时发布新版本，Kong在新发布的Kong 1.0 GA中，正式增加了对Service Mesh的支持。

2018年的大幕已落，2019年，Service Mesh领域的竞争会更加精彩。

2019年3月20日，Istio 1.1版本在正式发布，这是Istio 1.0版本推出8个月后第一个正式版本，包括Sidecar资源、路由区域感知能力等众多特性。

下面几节会对Service Mesh领域当前有代表性的项目进行详细分析。

3.3 Service Mesh项目Linkerd

3.3.1 Linkerd演进

Linkerd是云原生软件公司Buoyant推出的开源项目，是全球范围内第一个完善的Service Mesh项目，也是第一个将Service Mesh技术广泛用于生产环境，是当之无愧的Service Mesh技术开山鼻祖。Buoyant和Linkerd最早开始不断地宣扬Service Mesh技术理念，构建Service Mesh技术体系并成功推向市场，对Service Mesh技术的发展和壮大起到至关重要的作用，加速了Service Mesh技术的落地。

介绍Linkerd之前先介绍一下和Linkerd渊源颇深的Fingle。Fingle是Twitter开源的支持高并发、高可用的服务治理框架，Finagle除了支持广泛应用的基于请求/答复的RPC协议之外，还集成了如Thrift、Redis、Memcached、MySQL等多种协议，Finagle聚焦统一的、协议无关的服务集成和服务治理功能集成。Fingle经过长期的生产环境验证，已经非常成熟稳定，在JVM系得到广泛应用，基于Finagle可以方便快捷地搭建微服务。

Linkerd建立在Finagle之上，由前Twitter工程师William Morgan（Buoyant首席执行官）和Oliver Gould（Buoyant首席技术官）创建，初衷是将Finagle强大的RPC集群通信与服务治理能力标准化和产品化，摆脱JVM系的限制，提供给更多的用户使用。

Linkerd作为下一代云应用的基础网络层，通过自动化负载均衡、服务发现和运行时恢复能力等诸多特性，使得企业能够在不牺牲可靠性的情况下将其计算架构平滑地从单体服务转移到微服务架构，Linkerd提供如下主要特性。

1) **高性能、高扩展性的集群通信能力**。每秒以最小的时延及负载处理万级数量的请求，Linkerd 2.0的数据平面当前大小仅为10MB左右，并且99.9%的请求耗时小于1ms；部署方便，业务代码和配置不需要任何修改，特别方便水平扩展。

2) **支持多种服务发现方式，并且扩展非常方便**。支持各种服务发现机制，如基于文件（File-based）、ZooKeeper、Consul及Kubernetes；增加自定义的服务发现机制比较方便，扩展性强。

3) **强大的负载均衡算法支持**。推荐使用基于感知时延的负载均衡算法，通过实时统计数据RPC延迟、要处理请求队列大小决定如何分发请求，反馈实时性强，能够根据当前的实时压力情况调整负载；内置多种负载均衡算法，可以根据实际场景灵活选用。

4) **完善的多协议支持**。支持HTTP 2.0、HTTP 1.1、gRPC、Thrift多种协议。

5) **动态路由**。针对HTTP协议，支持多种路由策略；支持动态路由机制，可以通过动态修改路由规则实现蓝绿部署、金丝雀部署、流量迁移等；

6) **内置完善的log/Metric/Trace**。Linkerd通过收集服务间通信的各种统计数据，构建起强大的仪表盘，具体包含每个服务当前的运行状态，请求的成功率，系统的实时拓扑等。

7) 多平台支持。Linkerd通过收集服务间通信的各种统计数据，构建起强大的仪表盘，具体包含每个服务当前的运行状态，请求的成功率，系统的实时拓扑等。

Linkerd诞生后，以其简单易用的使用方式和强大的服务治理能力，一段时间内成为Service Mesh技术领域的旗帜和标杆，当然Linkerd自身也有不少缺点，其中最突出的问题就是：[Linkerd的部署模型过重](#)，设计过程中很少考虑有限资源情况下基于Sidecar模式的Kubernetes部署方式，制约了Linkerd的进一步发展；加上面对后起之秀Istio/Envoy强大的竞争压力，Buoyant给出的答案是利用Linkerd已经超过18个月的生产级Service Mesh经验，快速推出功能完备又可以在低资源环境下运行的Service Mesh，为了不给Linkerd生产环境下的众多用户带来困扰，Buoyant决定启动一个和Linkerd并列的全新项目，这就是Conduit。

Conduit的目标是成为最快、最轻、最简单并且最安全的Service Mesh。Conduit使用Rust构建了快速、安全的数据平面，用Go开发了简单强大的控制平面，总体设计围绕着性能、安全性和可用性进行。同时与Linkerd支持Kubernetes、Mesos等多个调度平台不同，Conduit更为聚焦，只提供对Kubernetes的支持。

在分别运维了两个项目仅仅7个月之后，Buoyant团队觉得Conduit“配得上Linkerd这个名字”，因此Conduit 0.5版本成为Conduit的最后一个主要版本，Conduit后续逐步整合进Linkerd项目，成为Linkerd 2.0的基础继续存在。

2018年9月，Linkerd 2.0发布，Conduit正式并入Linkerd项目，Conduit作为一个存在不到一年的短暂项目，变成了历史。

当前，Linkerd项目以Linkerd 1.x和Linkerd 2.x的方式存在，并行开发，Linkerd 1.x相当于Conduit诞生之前的Linkerd；Linkerd 2.x继续承接类似Conduit的角色，只不过换了个名字。Buoyant正在和社区制定相应的迁移路线图，逐步将Linkerd 1.x的存量用户迁移到Linkerd 2.x。

Conduit是为了对抗Istio而诞生，为了构建和Istio相抗衡的Service Mesh生态体系，Conduit退出舞台也标志着Buoyant退出了Service Mesh生态的想法，后续Service Mesh生态只有Istio一家独大了。后续Linkerd的定位会朝Service Ops方向发展，提供Service Mesh生态体系中一些具体的解决方案。

Linkerd 2.0推出了一个重要的功能，就是Linkerd以“Service Sidecar”的方式独立部署，无须集群范围的安装，即可增强单个服务的集群管理和服务治理功能，并且整个过程对服务没有任何侵入。

从当前看，Linkerd后续可能会两条路线并存。一条路线是提供包含数据平面和控制平面在内的全套轻量级Service Mesh解决方案，这个过程中不做平台，不再强调可扩展性，直接提供用户需要的功能，解决用户的痛点问题，并且特别轻量，不需要用户过多地选择，卖点就是轻量；另一条路线是融入Istio生态体系中，仅提供数据平面。

3.3.2 Linkerd路由机制

路由机制（见图3-2）是Linkerd的核心，Linkerd的主要工作就是收到一个请求消息，然后将该请求消息转发到合适的目的节点，这个过程中主要由服务识别（Identification）、绑定（Binding）、解析（Resolution）和负载均衡4个主要步骤组成，其中识别、绑定、解析属于路由管理的范畴。

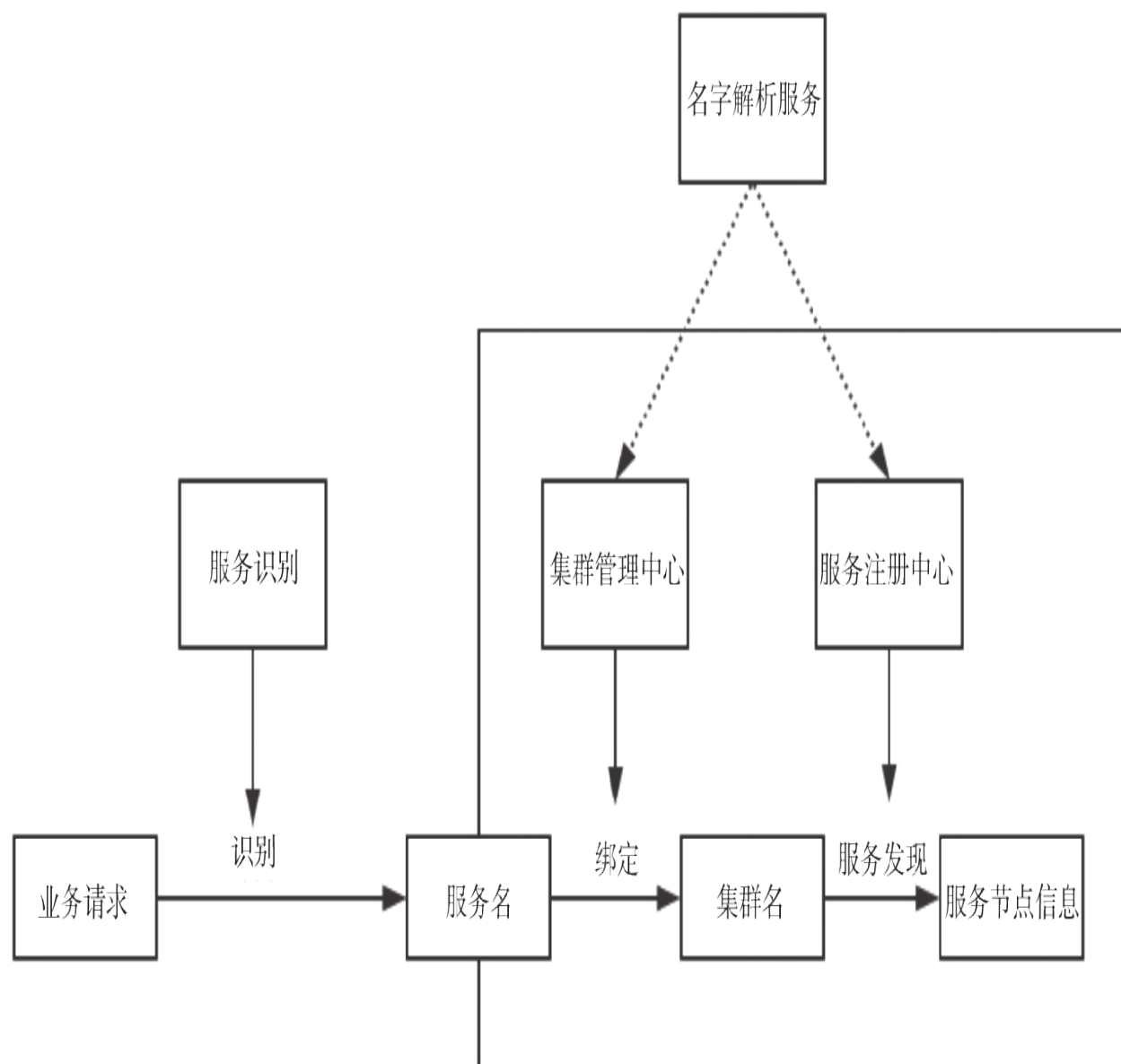


图3-2 Linkerd路由工作机制示意图

识别是指根据请求确定请求调用服务的过程，识别阶段的输出是service name，识别过程是根据识别类型，从请求消息生成一个特定的字符串，这个字符串与随后和这个请求关联，Linkerd路由会有一个默认前缀/svc，Linkerd会将默认前缀加上这个字符串作为本次

请求的路由标识往后传递，对于HTTP请求来说，Linkerd默认情况下使用一个io.l5d.header.token类型的标识，它使用HTTP请求头中的Host字段作为标识，比如curl-H"Host: data"http://example/hello，路由标识时被转换为/svr/data。

同时，识别是一个可扩展的插件，用户可以使用自定义的识别逻辑，将请求按照自己的意愿转换到相应的识别。

通过识别阶段确定了service name之后，由dtab组件根据service name绑定，确定目标服务的集群名称。绑定阶段输出的是client name，client name是目标集群的名称，类似于DNS的域名，和service name只包含目标服务信息不同，client name中还包含集群、区域以及环境等信息。

dtab负责对绑定规则进行管理，比如dtab规则配置如下：/svc=>/#/io.l5d.fs，我们的/svc/test被转换为：/#/io.l5d.fs/test，client name一般以“/\$”或者“/#”开头。

绑定是Linkerd中变化非常多的一个阶段，Linkerd中的很多场景需求，比如蓝绿测试、小流量灰度等均通过绑定来实现。

解析是服务发现的过程，通过named组件将目标集群名称client name转换成一组可用的节点，Linkerd会通过配置文件配置当前使用的服务发现方式，确定服务发现方式后，就可以根据client name，从服务发现中获取节点信息。

回顾上述Linkerd路由查找的整个过程，就是一个根据请求获取被调用服务名，然后根据路由规则从服务名获取对应的服务集群名，最后根据服务发现获取可用节点的过程。其中dtab和named组件分别负责路由规则和服务发现的管理。上面的讨论中，dtab与named和linkerd集成在一个进程中，默认情况下，配置在Linkerd中的dtab路由规则是Linkerd启动时解析，运行时无法动态修改。

为了解决dtab无法动态修改的问题，更好地进行路由规则管理，Linkerd将dtab从linkerd代理中剥离出来，形成一个独立的服务named，named服务独立部署，灵活支持各种路由规则的动态修改和下发。

3.3.3 Linkerd 2.0核心架构

从上面的Linkerd演进可以看出，Linkerd从诞生到现在虽然时间不算长，战略、定位和架构均发生了不少的变化，下面以Linkerd 2.X版本为例，介绍Linkerd当前的架构。

架构层面，Linkerd也是分为控制平面和数据平面，其中控制平面由下面几个部分组成。

- 1) Controller控制器：控制器由负责不同控制功能的多个容器组成，主要包括public-api、proxy-api、destination和tap，其中public-api负责Linkerd对外的API交互；
- 2) Web：Web是Linkerd提供的仪表板；
- 3) Prometheus：Prometheus负责Linkerd Metric信息的收集和存储；
- 4) Grafana：Grafana负责Linkerd的可视化。

Linkerd的数据平面采用的是Rust语言实现的轻量级高性能Proxy代理，通过配置Iptables，它可以透明地管理Kubernetes Pod的入口和出口流量，甚至可以对一个正在运行中的服务透明无感知地添加Linkerd代理。Linkerd支持HTTP、HTTP/2和TCP协议，通过按需诊断tap API，Linkerd拥有强大的可视化监控和诊断能力。

从Linkerd的整体架构看，Linkerd已经完全舍弃了打造Service Mesh平台的想法，内置具体的统计和可视化组件，聚焦为用户提供开箱即可的Service Mesh能力，当前Linkerd确实已经非常轻量，对于没有特别要求的通用场景，Linkerd还是一个不错的选择。

3.4 Service Mesh项目Istio

3.4.1 Envoy

Envoy是一个用C++编写的云原生高性能边缘代理、中间代理和服务代理，作为专门为微服务架构设计的通信总线，定位是作为Service Mesh的数据平面，接管微服务通信的全部流量，对应用程序屏蔽网络和通信的复杂性。

Envoy作为一个独立进程，设计与应用程序一起运行，所有的Envoy形成一个透明的通信网格，每个应用程序发送消息到本地Envoy代理或从本地Envoy代理接收消息，但不需要知道具体的网络拓扑。

Envoy的核心设计理念是网络应该对应用服务透明，职责划分上比较明确，当网络和应用程序出现故障时，可以很容易定位问题的具体根源。

架构上Envoy内部可以分为数据平面、控制平面和管理平面3个部分。其中，控制平面用于对流量路由和转发相关的策略与配置进行管理；控制平面通过标准API获取最新的流量配置信息；数据平面的流量转发就是基于控制平面下发的配置规则进行。

为了方便对Envoy运行状态进行监控和管理，Envoy内置一个HTTP Server，作为Envoy的管理平面，HTTP Server会注册一系列的Handler，对外暴露管理平面的API，用于外界查询当前Envoy各个维度的状态，比如外界可以通过管理平面API查询Envoy当前的集群和路由配置、当前的统计信息等。

Envoy接收到Downstream客户端发过来的请求时，一般先经过类似Iptables的透明拦截机制，然后将流量交给Envoy。Envoy获取请求对应的原目的地址，根据目的地址信息获取对应的网络过滤链配置信息，根据网络过滤链配置建立新的连接，并通过网络过滤链对应的协议对请求进行解码和路由，选择合适的Upstream节点，将请求发送出去，同时建立Downstream连接和Upstream连接的对应关系，将Upstream返回的响应消息发送给Downstream客户端。Envoy的数据流和控制流示意图如图3-3所示。

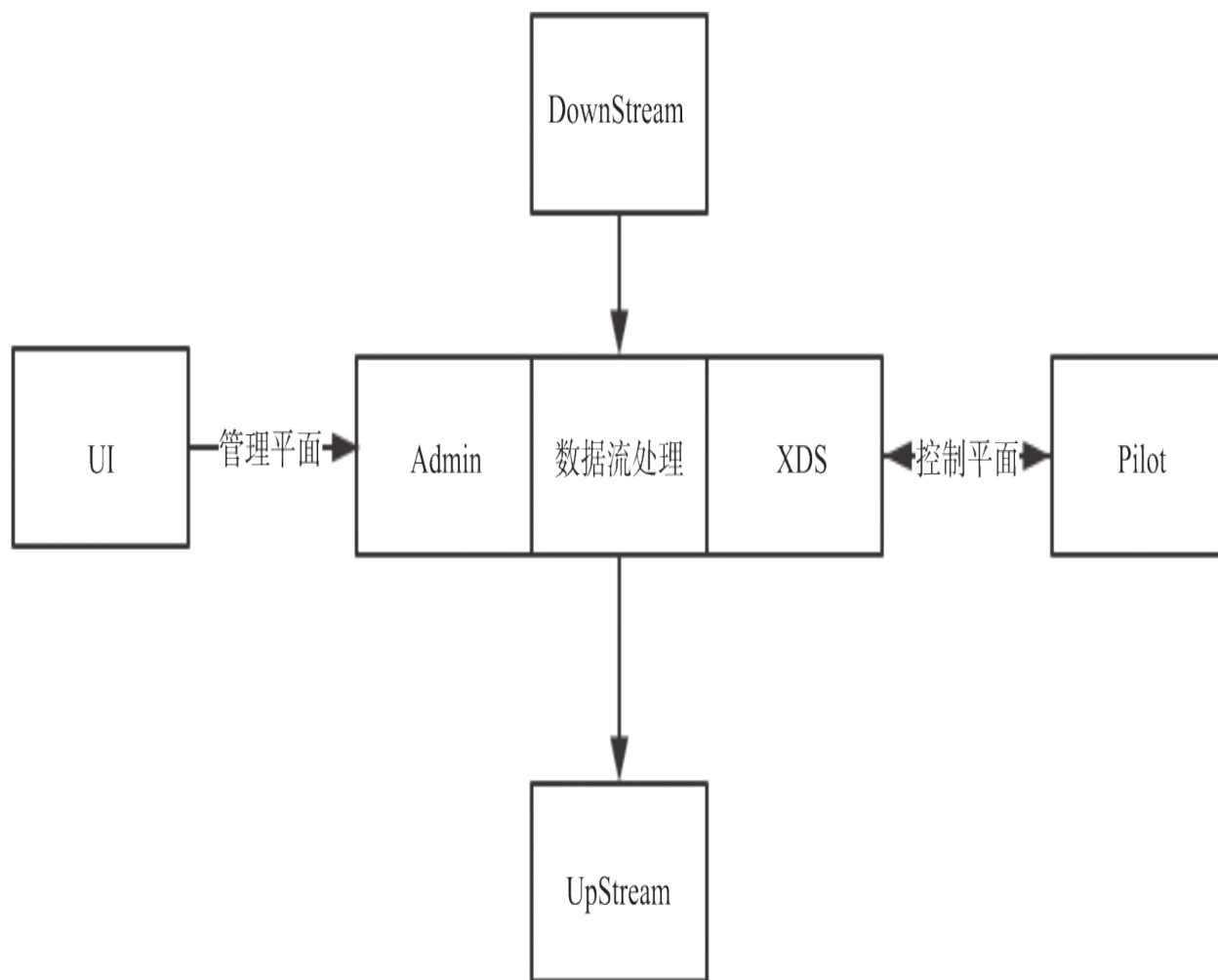


图3-3 Envoy的数据流和控制流

Envoy之所以在如此短的时间内，在Service Mesh数据平面中脱颖而出，是和它优雅的架构设计密切相关，主要体现在如下方面。

（1）扩展性

Envoy扩展性通过插件机制实现，Envoy内部当前已经支持众多扩展点，通过网络协议插件和HTTP处理扩展插件，分别从通信协议和链路治理两个层面进行扩展。Envoy插件机制是Service Mesh数据转发和数据处理的基础，基于插件扩展性机制进行扩展和定制开发非常方便，可以建立起强大的Service Mesh生态。

（2）配置动态化

配置动态化是Envoy架构层面最优雅的设计。基于配置动态化设计，Envoy集群管理与流量转发相关的所有配置均可以通过XDS协议动态下发和生效，不仅减少了运维复杂度，同时借助动态配置化能力，方便通过一定机制发现Envoy运行时的风险并快速调整，提高了Envoy的整体稳定性。

（3）性能

Envoy支持如此众多特性的同时，仍然可以提供优秀的性能指标，得益于良好的架构设计，Envoy架构设计对性能的考虑随处可见。比如数据转发层面，Envoy采用异步事件驱动的方式，并且保证一个请求只会在一个线程内处理，减少了请求在线程间切换的开销；此外通过数据平面和控制平面分离，配置变更时，通过无锁机制保证数据转发的性能不受任何影响。

此外Envoy对Service Mesh的一大贡献是第一个提出了通用数据平面API的概念，通过通用数据平面API，建立了数据平面和控制平面之间交互的标准，实现了数据平面和控制平面通信的标准化。只要基于数据平面API实现，可以根据需要方便地对数据平面或控制平面进行替换，有利于Service Mesh生态体系的建设。

3.4.2 Istio

Istio是Google、IBM和Lyft联合推出的开源微服务Service Mesh框架，旨在解决大量服务的发现、连接、管理、监控以及安全等问题，Istio综合了Google、IBM和Lyft在微服务领域的强大积累和经验，以Envoy为基础，将Envoy作为默认的数据平面，同时提供强大的控制平面能力。一经推出在Service Mesh领域就引起巨大的反响，当前已成为Service Mesh事实上的平台和标准。

Envoy已经可以很出色地完成流量路由和转发工作，Istio在Envoy的基础上，提供如下几方面的核心能力。

（1）动态配置管理和下发

Istio通过提供通用的配置管理能力，保证不同平台、不同环境下的配置均能够通过一致的方式对Envoy进行配置和下发，对Envoy屏蔽了配置管理的复杂性。这个工作具体由Istio的Pilot组件负责完成。

（2）Envoy生命周期管理和运行状态监控

Istio负责Envoy的生命周期管理，启动Envoy，然后监控Envoy的运行状态，如果运行有异常，Istio会尝试对Envoy进行重启，重启仍然不能成功时，Istio会将当前Envoy实例调度到其他节点上运行。这个工作具体由Istio的Pilot组件负责完成。

（3）通信安全

通信安全对于微服务来说是一个非常重要的基础能力，Istio通过托管身份验证、授权和服务之间通信的双向TLS加密，保障通信的安全性，这个工作具体由Istio的security组件负责完成。

（4）策略控制

对于一些核心资源，需要通过一定的策略，在不同消费服务以及服务的不同实例中进行分配，保证资源能够按照预期进行管理，这个工作具体由Istio的Mixer组件负责完成。

（5）遥测统计

Envoy自身也能进行一些遥测统计相关的工作，但如果需要支持不同的遥测统计后端基础设施，可以将这部分工作移到控制平面上，保证Envoy自身架构的简洁性和稳定性，通过收集遥测统计相关的数据和指标，可以对服务进行全方位的监控。这个工作具体由Istio的Mixer组件完成。

3.5 Service Mesh其他解决方案

3.5.1 国外其他Service Mesh项目

Service Mesh技术的迅速发展，撼动了微服务通信和流量管理相关的整个生态体系，相关厂商也针对性地推出了自己的Service Mesh支持方案，比较有影响力的有Aspen Mesh、Consul、nginMesh和Kong，下面分别讨论下各自的诉求和侧重点。

Aspen Mesh是由负载均衡领域的领导者F5推出的Service Mesh解决方案，Aspen Mesh基于Istio 1.0开发，相对Istio 1.0来说有如下优势：

- 作为托管的SaaS平台；
- 丰富的UI仪表板；
- 更多的实验特性；
- 可获得Aspen Mesh工程师团队的专业支持。

HashiCorp Consul通过新增组件Connect来实现对Service Mesh的支持，Connect可自动将任何现有的Consul集群转换为Service Mesh解决方案。Connect通过自动TLS加密和基于身份的认证实现安全的服务到服务通信。Connect建立在易于使用的基础上，可以使用一个配置选项启用它，只需向服务注册添加一行代码，就会自动使任何现有应用程序接收基于Connect的连接。证书轮换是自动的，不会导致停机。

nginMesh是Nginx推出的Service Mesh开源项目，通过增加一定的管理手段，将Nginx作为Istio服务网格平台中的数据平面。它提供负载均衡、服务发现、请求转发、路由规则、性能指标收集等功能，并将以“标准，可靠和安全的方式”使得服务间通信更容易。从GitHub上看，nginMesh最近几个月的更新频率并不算高，特性相比之前也没有大的变化，只能算是Nginx公司的试水之作。

API网关解决方案提供商Kong很早就有支持Service Mesh的想法，从GitHub上很早就加上了Service Mesh的标签，但一直没有具体的动作。2018年9月，Kong 1.0正式发布，通过在Kong实例之间添加MTLS（Mutual Transport Layer Security）和修改插件运行循环来支持Service Mesh部署模式。通过这些更改可以将Kong与服务的每个实例部署在一起，代理服务之间的信息流随服务的伸缩而自动伸缩。

从Aspen Mesh、Consul、nginMesh以及Kong这几个代表性产品对Service Mesh支持来看，当前并没有特别的惊艳之处，这些厂商之前处于其他领域，是担心自己擅长的传统领域被Service Mesh颠覆而采取的防御之作，后续还需要持续进一步观察。

3.5.2 Service Mesh在中国的发展

Service Mesh技术，国内动作最大的当属阿里系，阿里系总体上采用两头出击的方式。其中Dubbo当前的思路是，迎合Istio已成Service Mesh事实上大标准的大形势，将阿里当前成熟的服务治理基础设施融入Istio/Envoy体系，与领域主流开源项目形成合力发展，源于开源、反哺开源。数据平面部分，和Envoy一起，实现Envoy对Dubbo协议的解析和统计信息收集，进入Envoy主线；此外服务路由支持当前还在规划中。控制平面，将阿里之前已经开源的服务治理基础设施和Istio对接，当前已完成与VIPServer、Diamond的对接，正规划与ZooKeeper、Nacos的对接。

蚂蚁金服的Sofa Mesh控制平台基于Istio进行改进和扩展定制，数据平面采用完全自研的方式。据Sofa的技术人员说，他们有个非常宏大的蓝图，将来会把原有的网络和中间件方面的各种能力在Mesh层重新沉淀，打造成为未来新一代架构的底层平台，承载各种服务通信的职责，因此采用了基于Golang语言、完全自研的数据平面。蚂蚁金服的Service Mesh解决方案当前均已经开源，控制平面SOFAMesh在GitHub上的地址是：<https://github.com/alipay/sofa-mesh>，数据平面SOFAMosn在GitHub上的地址是：<https://github.com/alipay/sofa-mosn>，感兴趣的读者可以针对性的研究下。

国内其他公司，华为公有云和腾讯云的PaaS微服务治理平台当前均支持Service Mesh技术，其中华为公有云采用自研的方式，腾讯云的TSF微服务平台采用基于开源的Istio/Envoy技术栈，进行扩展和定制。

微博是国内一线互联网公司中比较早在生产环境下实践Service Mesh技术的公司，积累了不少实战经验，微博的Service Mesh实践是结合自身业务特点，完全从解决业务痛点问题出发而产生的，并且实践比较早，和当前热门的Istio/Envoy以及Kubernetes之间并没有多少关联，算是一种平台无关的Service Mesh实现，微博的Service Mesh技术实践也已经开源，在GitHub上的地址为：<https://github.com/weibocom/motan-go>。

唯品会是国内最早进行Service Mesh相关探索和实践的公司，唯品会的主流技术栈是Java，并且有着成熟完善的Java服务治理体系OSP（Open Service Platform），早在2015年，唯品会就在生产环境中采用Sidecar方式，用于非Java系语言的多语言接入和服务治理支持，很好地解决了业务快速发展过程中的多语言技术栈痛点问题。当前，唯品会正在计划基于Service Mesh进行智能参数治理、智能故障分析和告警以及全链路服务综合治理等工作，同时在筹备2019年将唯品会生产环境中使用的Service Mesh解决方案进行开源。

Service Mesh周边基础设施方面，国内当前也有了长足发展，尤其是分布式跟踪项目Skywalking发展迅速，后续值得持续关注。

3.6 Service Mesh云上产品

3.6.1 AWS App Mesh

2018年12月5日，亚马逊AWS首次推出了自己的Service Mesh产品，App Mesh公开预览版。AWS App Mesh可以轻松监控和控制AWS上运行的微服务，通过标准化的微服务通信方式，为用户提供端到端的可见性，并帮助确保应用程序的高可用性。App Mesh的推出标志着全球云计算领域的绝对领导者AWS正式加入到Service Mesh的竞争，有着非常重要的指标意义。

架构层面，AWS App Mesh基于开源的数据平面代理Envoy构建，使用Envoy管理AWS微服务的所有进出流量，控制平面是AWS自研，用于通过XDS API对Envoy进行配置和管理。AWS App Mesh支持Kubernetes、Amazon ECS、Amazon EKS等多个平台。

AWS App Mesh当前还处于发展早期，处于试用阶段，后续的趋势与进展还需要持续观察。

3.6.2 Azure Service Fabric Mesh

Service Fabric是微软公司出品的开源跨平台的分布式应用平台，通过它可轻松开发、打包、部署和管理微服务。2018年7月，微软发布Service Fabric对应的云托管版本Service Fabric Mesh，Service Fabric Mesh的理念是让微服务开发者关注应用程序本身，而不是集群。Service Fabric Mesh通过提供智能路由、可伸缩性、蓝绿部署、安全等众多服务治理特性，大大提升了业务开发的效率。

架构层面，和AWS App Mesh类似，Service Fabric Mesh也基于Envoy构建，通过Envoy提供强大的流量调度和路由管理能力。

平台支持方面，Service Fabric Mesh支持包括Kubernetes、Cloud Foundry、Docker Compose等众多编排平台。

3.6.3 Google Cloud Service Mesh

Google Cloud在Service Mesh支持上围绕Istio展开。2018年底，Google推出一键集成的Istio“Istio On GKE”，提供流量管理、策略控制、遥测和安全能力。随后Google推出了Istio的完全托管版本Google Cloud Service Mesh，当前提供和开源Istio完全相同的能力，Istio的体系当前已经非常庞大，通过提供托管方式，可以减少开发者在Istio部署、管理上的开销。

除了Google Cloud Service Mesh之外，Google Cloud还发布了一个重量级的流量管理产品Traffic Director。和Istio当前主要聚集Kubernetes平台不同，Traffic Director提供对容器、VM等多种运行环境的支持，可以在多平台、多环境之间进行全局流量调度。

由于Istio当前版本迭代的进度和预期相比较慢，并且Istio主要支持Kubernetes，而当前只运行在Kubernetes上的业务不算多，导致一段时间内，Istio在生产环境中很难直接大规模使用。因此，Google Cloud在Service Mesh上采用两头押宝的方式：Traffic Director定位当下，解决业务目前的痛点问题；Google Cloud Service Mesh定位未来，引领社区步伐，确保下Istio生态下始终保持领先地位。

3.6.4 SuperGloo

SuperGloo是Solo.io公司推出的Service Mesh编排引擎，SuperGloo提供两个主要收益。

1) 和托管的思路类似，降低用户使用Service Mesh的复杂度。

2) 市面上当前主流的Service Mesh项目虽然已经不少，包括Linkerd、Istio、Consul等，但是这些Service Mesh解决方案采用自己独有的方案，没有标准化，不同Service Mesh之间无法互通。SuperGloo构建在这些Service Mesh之上，不仅可以支持不同Service Mesh之间的互通，也支持多个Service Mesh平台之间的全局流量调度。

鉴于当前Service Mesh自身市场前景还不太明朗，SuperGloo这种Service Mesh编排确实是一个大胆而有意思的尝试。

除了SuperGloo之外，Solo.io围绕Service Mesh还发布了不少产品，比如网关产品，目的是押注Service Mesh生态的基础设施和工具链体系，前景值得持续观察。

3.7 Service Mesh标准化

当前市场已经涌现了不少Service Mesh，各自为政，不同Service Mesh之间迁移的成本非常高，为了实现Service Mesh生态的健康持续发展，需要建立一套标准，大家共同遵照相同的标准进行扩展实现。

数据平面层面，CNCF以Envoy Xds v2 API为基础在筹建通用数据平面API，不过尚在讨论和筹备阶段。2019年5月，微软牵头Linkerd、Hashicorp、Solo.io等公司，推出了Service Mesh Interface（简称SMI），目的是构建一套不同Service Mesh互通的标准。

Service Mesh标准化，是Service Mesh发展过程中重要的一步，当前只是迈出了很小的一步。只有解决了标准化的问题，才能构建起Service Mesh生态的基石，促进后续的健康发展。

3.8 本章小结

本章重点介绍Service Mesh当前的发展状况，以及Service Mesh可以解决微服务通信、服务治理标准化等痛点问题。从Service Mesh的发展历程可以看出，Linkerd和Istio是Service Mesh开源领域当前的两大主要势力，并且各主流云服务厂商如AWS、微软Azure、Google云均在Service Mesh领域开始布局。

第二篇 架构篇

第4章 Envoy架构剖析

本章详细介绍Envoy，将从Envoy的核心概念、核心组件以及核心数据结构等开始展开，进而重点分析XDS API设计、Filter插件机制、Envoy的控制流和数据流，从架构设计到实现细节，逐个进行剖析和阐述。

4.1 Envoy整体架构

4.1.1 关键设计约束

Envoy的定位是“云原生时代的高性能边缘/中间/服务代理”^[1]，作为云原生时代的通用代理服务器，需要能够适应和支持云原生用户的高速增长以及由此带来的多样化需求。功能是一切的基础，除了满足基本的功能需求之外，架构设计层面还需要考虑下面这些重要的非功能约束。

(1) 易用性

除了完成自身的功能需求之外，对Service Mesh最重要的一个诉求就是易用性。简单易用，方便运维，系统只有做到简单易用，学习曲线不陡峭，容易上手，才能赢得用户的青睐。

具体到使用上，对于没有特殊要求的普通用户来说，需要能够真正做到Service Mesh接入过程完全透明，业务完全无感知，不需要修改任何代码和配置。

(2) 可用性

可用性是系统竞争力的关键一环，一个频繁出故障的系统会给用户带来很大的业务损失，这样的系统对用户来说是没有价值的。

(3) 可观测性

可观测性和可管理性使得我们可以查看系统的内部运行状态。基于可观测性，我们可以使用数据驱动的思维，对系统的故障进行分析和定位，系统地精细化管控和运营。

(4) 扩展性

对于很多平台性的产品来说，只有具备了良好的扩展性，才能随着不断变化的需求，构建起强大的生态，因此很多系统会把扩展性作为最重要的目标来对待，其实这是有一定误区的。一个系统的基础目标用户是比较确定的，扩展性主要用来提升系统的增量目标用户。易用性、可用性和可观测性是提供基础目标用户体验的基石，也是系统的核心竞争力所在。在没有夯实基础的前提下过分注重扩展，会很容易丢掉赖以生存的根基。虽然基于良好的扩展性，系统的边界范围可以不断扩大和延伸，但是应该在保证基础用户的体验的前提下进行，尤其不能以牺牲95%用户的体验，而过分注重对5%增量用户的扩展和支持，得不偿失。

此外，对于扩展性来说，需要尽可能做到平台无关。虽然Mesh技术在某些平台下可以最大程度地发挥它的优势，但Mesh技术本身不应该和平台绑定，如果绑定就会失去一定的灵活性。

(5) 性能

随着硬件性能的不断提升，加上分布式技术的广泛应用，性能问题一般都可以通过横向与纵向扩展来解决，性能问题貌似不像以前那么重要了。对系统性能的要求，不能成为制约业务发展的短板，很多时候性能往往是一张入场券，性能可能不是系统竞争力的主要组成部分，但如果性能不能满足业务要求，可能拿不到竞争的入场券。

（6）其他

其他方面还有很多，这里主要提一下兼容性。良好的兼容性会使Service Mesh的迁移和推广路径更加清晰。比如，如果能够做到Service Mesh体系和非Service Mesh体系之间兼容与互通，会使存量服务的迁移更顺畅。Mesh服务和非Mesh服务可以自如地进行通信，比如单个服务也可以自如地接入Mesh，享受Mesh基础设施带来的优点。

上面分析的易用性、可用性、可观测性等，是按照重要度以从高到低的顺序排列的，功能需求之外，易用性是第一位的。易用性和扩展性本身是一对矛盾体，扩展支持越多，可能会导致系统越复杂，如果设计不当，很容易影响系统的易用性。

[1] Envoy项目在Github上的描述见Cloud-native high-performance edge/middle/service proxy。

4.1.2 设计原则

Envoy最重要的设计原则是网络透明性，Envoy的设计者认为，网络对于应用程序来说应该是透明的，应用程序只需要按照标准化的方式使用网络，不需要感知网络的具体实现细节，这样当发生应用程序或者网络故障时，很容易区分问题的边界，从而很容易定位出问题的根源。

网络透明是Envoy架构层面总的指导原则，架构设计的各个维度都围绕这个指导原则展开，比如对出入Envoy流量的透明拦截、对通信安全的透明支持以及完善的链路治理支持等，Envoy架构设计和实现层面处处可见网络透明的影子，目的是把用户体验放在第一位，最大程度地提高业务使用Envoy时的易用性。

Envoy架构设计的另外一个重要指导原则是最终一致性。Envoy在配置处理、统计子系统等多个场景都会采用基于最终一致性的设计。通过最终一致性，可以简化Envoy的架构设计，通过牺牲一些不太重要的指标，可以换来架构易用性和性能上的巨大提升。

最后一个设计原则是对性能的折中考虑，Envoy语言选型、架构层面的优秀设计已经可以保证Envoy有非常优异的性能表现。性能对于通用的代理服务来说无疑是非常重要的，Envoy重视性能的同时，不会执着于极限性能，不会为了性能考虑，牺牲架构上的简单和易用性设计，不会为了极限性能牺牲代码的可维护性。

4.1.3 整体架构

Envoy线程模型由负责控制流的主线程和负责数据流的工作线程组成，其中工作线程池的线程个数和CPU核数相当即可。所有工作线程同时监听Iptables透明拦截的Downstream客户端请求，监听器信息由监听发现服务LDS负责管理，监听到客户端请求后在当前线程中创建连接，后续该连接上的请求消息处理均在这个线程内完成，以减少请求处理过程中的线程切换开销。请求处理采用的是异步事件驱动模型，通过各自线程的事件调度器进行事件调度。Envoy整体架构设计如图4-1所示。

Envoy的扩展性由Filter插件机制负责，每个监听器下对应两类插件：一类是监听器相关的过滤插件，用于对监听行为进行扩展；另一类是网络过滤插件，负责建立请求连接、协议解析和路由工作。

当前Envoy实现最完整的是HTTP网络过滤插件，通过路由发现服务RDS管理HTTP协议的路由，HTTP路由是个多层的树状结构，对应的是每个集群以及相应的匹配条件，其中集群信息由集群发现服务CDS管理。Envoy集群有多种类型，其中服务发现模式的集群，其集群节点信息由节点发现服务EDS管理。通过集群的负载均衡算法获取到相应的目标节点后，将Downstream请求信息转发给Upstream。

Envoy请求处理需要的LDS、RDS、CDS和EDS配置信息由控制平面通过XDS协议动态下发给Envoy。

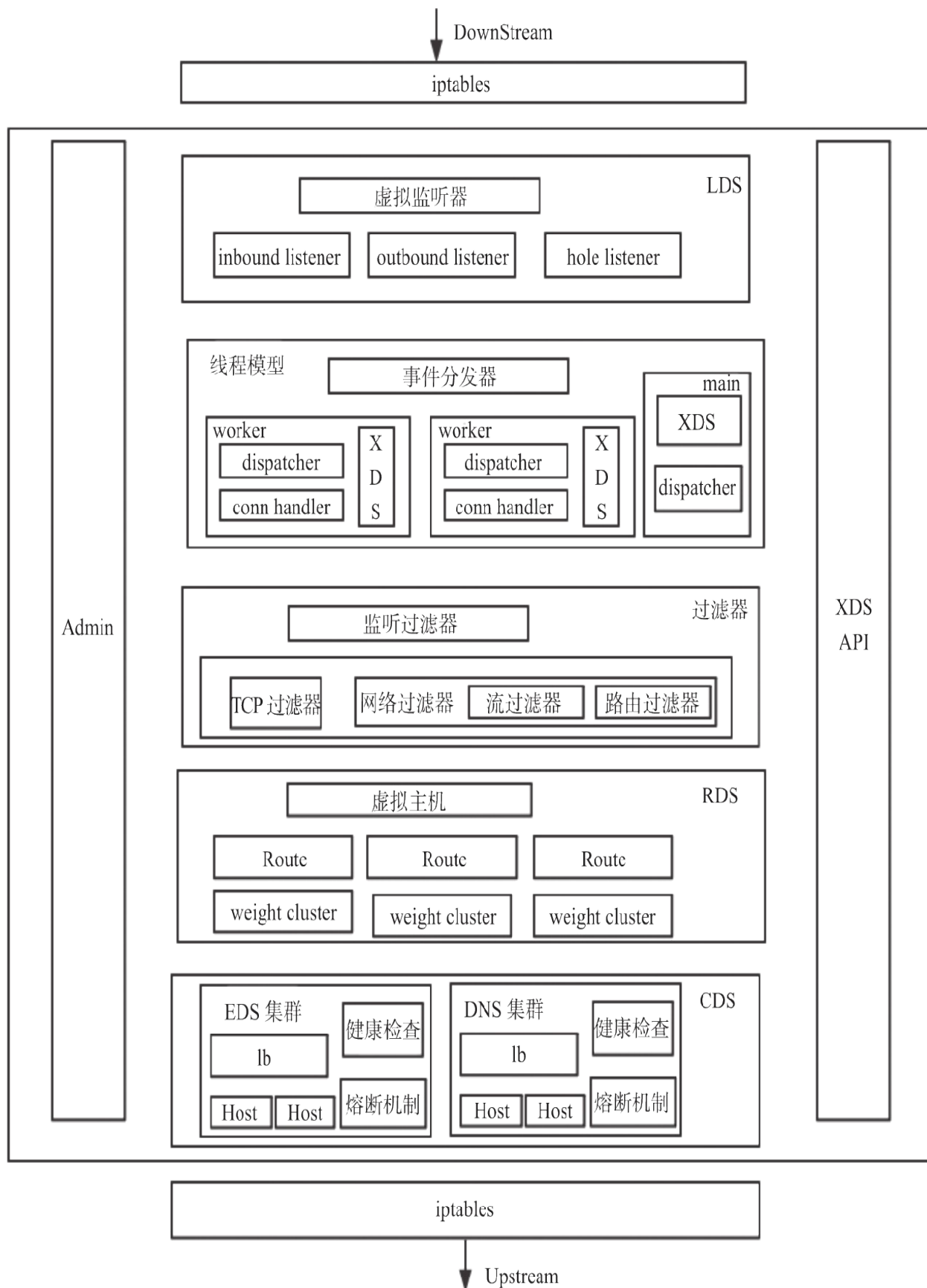


图4-1 Envoy整体架构图

4.2 Envoy网络模型

4.2.1 Envoy事件调度模型

事件调度是一种基于异步事件触发来驱动业务实现和流转的机制，事件调度是Envoy的基石。Envoy的请求处理和状态流转均依赖事件调度机制来实现，Envoy事件调度相关实现在/source/common/event目录下。

Envoy事件调度模型以Libevent为基础，Libevent是一个开源的基于事件驱动的高性能网络库，当前支持文件、信号和定时器等多种事件类型，Envoy事件调度器由DispatcherImpl负责管理。

```
DispatcherImpl::DispatcherImpl(TimeSystem& time_system, Buffer::WatermarkFactoryPtr&& factory, Api::Api&
: api(api), time_system_(time_system), buffer_factory_(std::move(factory)),
base_(event_base_new()), scheduler_(time_system_.createScheduler(base_)),
deferred_delete_timer_(createTimer([this]() -> void { clearDeferredDeleteList(); })),
post_timer_(createTimer([this]() -> void { runPostCallbacks(); })),
current_to_delete_(&to_delete_1_) {
}
```

Envoy当前使用Libevent的如下接口：

```
//以下几个接口分别用于初始化普通事件、定时器事件和信号事件
event_assign
evtimer_assign
evsignal_assign
//将初始化的event注册到libevent的事件链表上
event_add
//将事件加入到激活队列，event_add的事件只是已注册，加入到激活队列的事件会进入事件触发流程
event_active
//进入事件循环，等待事件触发
event_base_loop
```

Envoy事件调度模型分为事件创建和提交事件给调度器两部分。Envoy通过FileEvent管理使用最频繁的网络I/O事件，DispatcherImpl::createFileEvent负责创建相应的FileEvent。调用时会指定事件类型和事件触发回调函数，FileEventImpl创建时会分别调用event_assign和event_add创建事件，并将事件加入libevent，调用event_assign会传入事件触发回调函数，事件就绪时，会回调注册的事件回调函数。

```
FileEventPtr DispatcherImpl::createFileEvent(int fd, FileReadyCb cb, FileTriggerType trigger, uint32_t ev
return FileEventPtr{new FileEventImpl(*this, fd, cb, trigger, events)};
}

FileEventImpl::FileEventImpl(DispatcherImpl& dispatcher, int fd, FileReadyCb cb,
FileTriggerType trigger, uint32_t events)
: cb_(cb), base_(&dispatcher.base()), fd_(fd), trigger_(trigger) {
assignEvents(events);
event_add(&raw_event_, nullptr);
}

void FileEventImpl::assignEvents(uint32_t events) {
event_assign(&raw_event_, base_, fd_,
EV_PERSIST | (trigger_ == FileTriggerType::Level ? 0 : EV_ET) |
(events & FileReadyType::Read ? EV_READ : 0) |
```

```

        (events & FileReadyType::Write ? EV_WRITE : 0) |
        (events & FileReadyType::Closed ? EV_CLOSED : 0),
        [](evutil_socket_t, short what, void* arg) -> void {
            FileEventImpl* event = static_cast<FileEventImpl*>(arg);
            uint32_t events = 0;
            if (what & EV_READ) {
                events |= FileReadyType::Read;
            }
            if (what & EV_WRITE) {
                events |= FileReadyType::Write;
            }
            if (what & EV_CLOSED) {
                events |= FileReadyType::Closed;
            }
            event->cb_(events);
        },
        this);
    }
}

```

对于定时器事件来说，通过DispatcherImpl::createTimer创建定时器事件，通过TimerImpl::enableTimer完成定时器的提交。

```

TimerPtr DispatcherImpl::createTimer(TimerCb cb) {
    ASSERT(isThreadSafe());
    return scheduler_->createTimer(cb);
}

class RealScheduler : public Scheduler {
public:
    RealScheduler(Libevent::BasePtr& libevent) : libevent_(libevent) {}
    TimerPtr createTimer(const TimerCb& cb) override {
        return std::make_unique<TimerImpl>(libevent_, cb);
    };

private:
    Libevent::BasePtr& libevent_;
};

TimerImpl::TimerImpl(Libevent::BasePtr& libevent, TimerCb cb) : cb_(cb) {
    ASSERT(cb_);
    evtimer_assign(
        &raw_event_, libevent.get(),
        [](evutil_socket_t, short, void* arg) -> void {
            static_cast<TimerImpl*>(arg)->cb_(); }, this);
}

void TimerImpl::disableTimer() { event_del(&raw_event_); }

void TimerImpl::enableTimer(const std::chrono::milliseconds& d) {
    if (d.count() == 0) {
        event_active(&raw_event_, EV_TIMEOUT, 0);
    } else {
        timeval tv;
        tv.tv_sec = us.count() / 1000000;
        tv.tv_usec = us.count() % 1000000;
        event_add(&raw_event_, &tv);
    }
}

```

除了支持常规的事件触发外，Envoy事件调度器也支持任务的异步触发，通过DispatcherImpl::post投递具体任务，DispatcherImpl通过post_timer_定时器触发异步任务的具体触发。

为了方便各个工作线程的事件触发和管理，减少不同线程处理时的锁开销，Envoy线程模型为每个线程创建一个事件调度器，各个模块直接传递指针对象到事件调度器，如果回调时对象已经被销毁，就会出现访问异常，因此必须对传递给事件调度器的指针对象的

生命周期进行有效管理，Envoy使用延迟析构来解决这个问题。

DispatcherImpl在管理析构对象时，先调用DispatcherImpl: : deferredDelete进行被析构对象的注册，并使用deferred_delete_timer_定时器完成被析构对象的实际处理。定时器的处理函数是DispatcherImpl: : clearDeferredDeleteList，clearDeferredDeleteList函数调用每个被析构对象的reset函数进行实际的析构处理。具体处理过程中，DispatcherImpl通过to_delete_1_和to_delete_2_对被析构对象进行交替处理，由于对象析构比异步任务执行的优先级低一些，所以会将被析构对象拆分为两部分交替执行，避免对象析构耗时过长影响其他任务的执行。

4.2.2 Envoy线程模型

Envoy采用控制流和数据流分离的思想，将线程模型主要分为主线程和工作线程两类。

①Envoy的主线程负责控制流逻辑，主要是XDS配置更新、修改XDS相关的配置和数据。

②Envoy的工作线程负责数据流逻辑，主要是请求消息转发处理，转发处理时会查询XDS数据。统计信息也类似，工作线程负责修改统计信息，主线程负责对各个线程的统计信息进行汇总和统一上报。

4.2.3 线程本地存储机制

Envoy中多个线程并发操作同一个数据的场景很多，为了减少多线程并发访问共享数据时的锁开销，提升系统的整体性能，Envoy引入了TLS（线程本地存储）的概念。通过TLS，主线程上运行的代码可以分配到进程范围内的TLS槽，TLS槽是一个允许O（1）访问的向量索引，主线程可以将任意数据放在TLS槽，然后通过事件的方式将数据变更发布到每个工作线程中。每个工作线程都会基于主线程的TLS槽数据，复制一份完全相同的线程本地数据，工作线程只需要访问本线程的线程本地数据，同时每个线程的事件调度器响应主线程发布的变更事件，触发线程本地数据的更新。Envoy TLS的整体处理流程如图4-2所示。

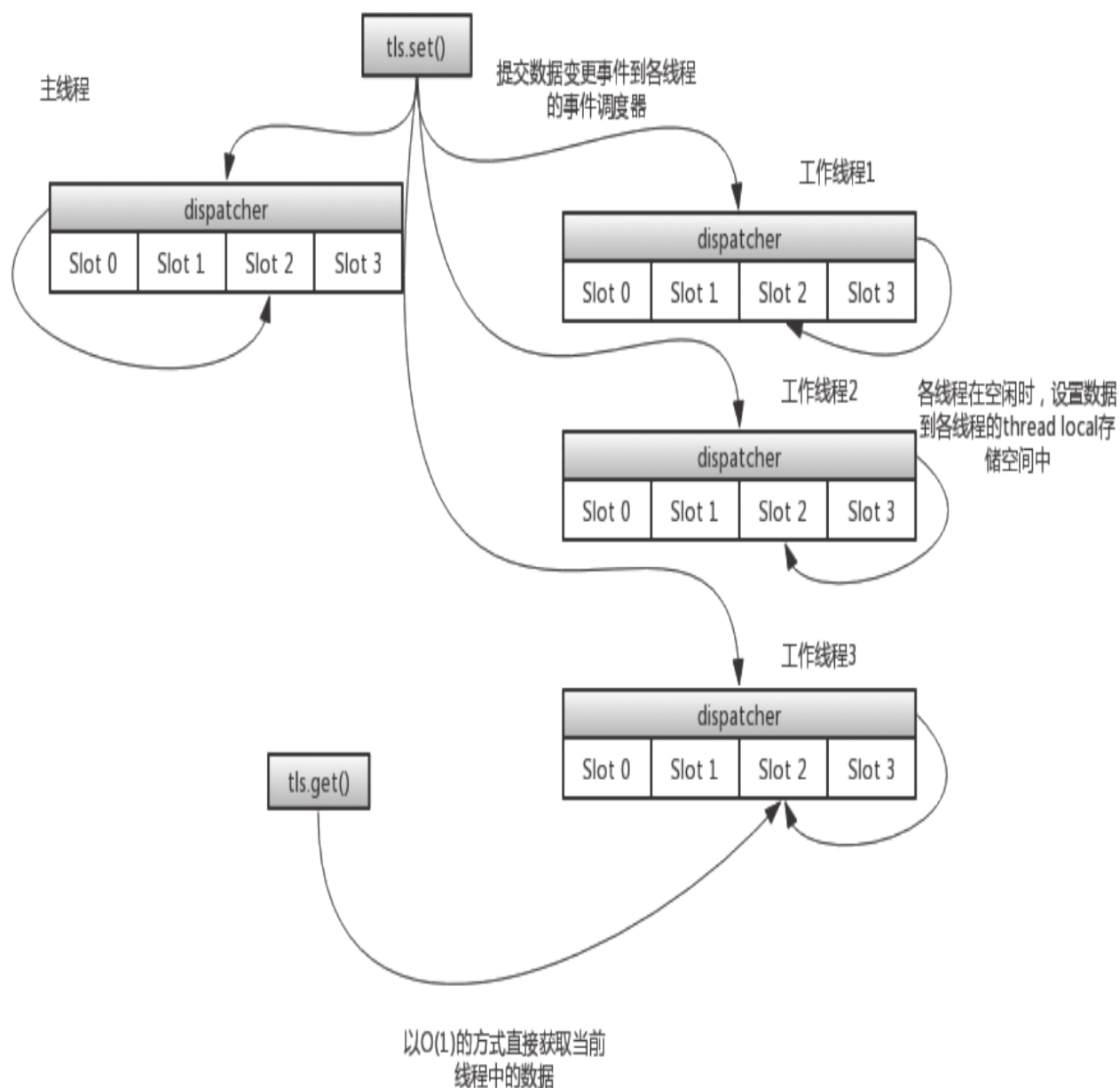


图4-2 Envoy TLS处理流程图

Envoy的TLS数据通过下述的thread_local_data_来进行管理，thread_local_data_是基于C++11 thread_local的线程本地数据，每个线程均拥有各自的线程本地数据，同时通过本地的事件调度器dispatcher_和主线程进行通信。

```
struct ThreadLocalData {
    Event::Dispatcher* dispatcher_{};
    std::vector<ThreadLocalObjectSharedPtr> data_;
};

static thread_local ThreadLocalData thread_local_data_;
```

为了在工作线程和主线程之间实现配置与数据交互，需要在主线程和工作线程之间建立一条信息传递的管道，各个工作线程的角色完全相同，相互之间不需要通信，它们均和主线程进行通信。由于主线程只有一个，各个工作线程通过全局变量main_thread_dispatcher_就可以和主线程进行通信。同时，Envoy工作线程启动后，向主线程注册自己的事件调度器，相当于表明自己的身份，当有事件发生时，主线程就可以通过注册的事件调度器进行事件通知。实现如下：

```
WorkerImpl::WorkerImpl(ThreadLocal::Instance& tls, ListenerHooks& hooks,
Event::DispatcherPtr&& dispatcher,
Network::ConnectionHandlerPtr handler,
OverloadManager& overload_manager, Api::Api& api)
: tls_(tls), hooks_(hooks), dispatcher_(std::move(dispatcher)), handler_(std::move(handler)), api_(api) {
    //注册当前线程的事件调度器，用于当前线程和主线程之间的事件通信
    tls_.registerThread(*dispatcher_, false);
    ...
}
```

在registerThread实现中，如果当前线程为主线程，将当前事件调度器设置到主线程调度器main_thread_dispatcher_；如果当前线程为工作线程，则将当前线程记录到线程列表registered_threads_中，后续主线程会基于registered_threads_对每一个工作线程进行循环处理。

```
void InstanceImpl::registerThread(Event::Dispatcher& dispatcher, bool main_thread) {
    if (main_thread) {
        main_thread_dispatcher_ = &dispatcher;
        thread_local_data_.dispatcher_ = &dispatcher;
    } else {
        registered_threads_.push_back(dispatcher);
        dispatcher.post([&dispatcher] { thread_local_data_.dispatcher_ = &dispatcher; });
    }
}
```

为了对不同的TLS对象进行区分和管理，Envoy为每个TLS对象分配不同的槽位，因此TLS对象使用前需要通过allocateSlot申请当前对象对应的槽位信息。allocateSlot会遍历当前已有槽位信息，查找是否有空闲的槽位，如果有直接拿过来使用；如果没有，分配新的槽位信息。

槽位信息的维护本质是资源池的管理，当前实现采用O(N)的方式从头开始遍历，当已有槽位个数非常多时，这里遍历查找的性能会很低，可以通过记录空闲槽位位置等方式进行优化。

```
SlotPtr InstanceImpl::allocateSlot() {
    for (uint64_t i = 0; i < slots_.size(); i++) {
        if (slots_[i] == nullptr) {
            std::unique_ptr<SlotImpl> slot(new SlotImpl(*this, i));
            slots_[i] = slot.get();
            return slot;
        }
    }
    std::unique_ptr<SlotImpl> slot(new SlotImpl(*this, slots_.size()));
    slots_.push_back(slot.get());
    return slot;
}
```

创建TLS对象的槽位后，会返回当前分配的槽位的地址，作为当前槽位的唯一标识和入口，后续对该TLS对象的所有操作均通过该槽位标识进行。

创建槽位后就可以通过set接口进行TLS数据的设置，registered_threads_存放了各个工作线程的线程本地信息，之前工作线程启动时已经将本线程的事件调度器注册到registered_threads_，这里遍历registered_threads_将数据设置事件推送到各个工作线程的调度器中进行处理，当然对于主线程自身来说是直接处理，具体处理函数是setThreadLocal。

```
void InstanceImpl::SlotImpl::set(InitializeCb cb) {
    for (Event::Dispatcher& dispatcher : parent_.registered_threads_) {
        const uint32_t index = index_;
        dispatcher.post([index, cb, &dispatcher]() -> void { setThreadLocal(index, cb(dispatcher)); });
    }
    //主线程处理
    setThreadLocal(index_, cb(*parent_.main_thread_dispatcher_));
}
```

setThreadLocal的数据处理主要是利用了C++11的Thread Local特性，C++11的Thread Local对象内部为每个线程创建一个独立的副本信息，对Thread Local进行设置操作时，Thread Local内部会对这些线程副本数据进行更新。

```
void InstanceImpl::setThreadLocal(uint32_t index, ThreadLocalObjectSharedPtr object) {
    if (thread_local_data_.data_.size() <= index) {
        thread_local_data_.data_.resize(index + 1);
    }
    thread_local_data_.data_[index] = object;
}
```

对TLS数据的查询和使用通过tls->getTyped的方式，实际调用的是SlotImpl的get接口，get接口的实现非常简单，直接使用SlotImpl保存的槽位编号，返回线程本地数据对应槽位的数据即可。

```
ThreadLocalObjectSharedPtr InstanceImpl::SlotImpl::get() {
    return thread_local_data_.data_[index_];
}
```

4.3 Envoy扩展模型

插件是Envoy扩展性支持的重要方式，通过插件扩展，可以支持不同的网络协议和链路治理特性，也可以对流量进行场景化定制处理。

4.3.1 插件扩展机制

1. 插件注册机制

Envoy插件当前采用静态注册的方式，只需要声明Registry：：RegisterFactory对应的模板类变量，就可以完成插件的注册。以OriginalDst插件为例，声明如下：

```
static Registry::RegisterFactory<OriginalDstConfigFactory,  
                                Server::Configuration::NamedListenerFilterConfigFactory>  
    registered_;
```

插件的具体注册通过RegisterFactory类来完成，在RegisterFactory类中，会通过模板参数声明一个类的成员变量，在类的构造函数中，调用registerFactory完成模板参数的实际注册。

```
template <class T, class Base> class RegisterFactory {  
public:  
    //通过FactoryRegistry构建模板  
    RegisterFactory() { FactoryRegistry<Base>::registerFactory(instance_); }  
private:  
    T instance_{};  
};
```

在FactoryRegistry<Base>：：registerFactory中，直接将插件名和插件实例注册到全局的插件实例map上。

```
static void registerFactory(Base& factory) {  
    factories().emplace(std::make_pair(factory.name(), &factory));  
    ...  
}
```

插件会通过Config：：Utility：：getAndCheckFactory函数获取注册的插件实例，getAndCheckFactory会调用FactoryRegistry：：getFactory函数，查找到相应的插件实例。

```
static Base* getFactory(const std::string& name) {  
    auto it = factories().find(name);  
    if (it == factories().end()) {  
        return nullptr;  
    }  
    return it->second;  
}
```

2. 插件触发机制

Envoy插件的使用方式均比较类似，上面讲述插件注册机制时可以看出，插件静态注册是一个根据插件名称将插件工厂模板注册到全局map的过程。使用插件前需要根据配置文件，使用插件工厂模板获取当前支持的插件工厂列表，示例代码如下：

```

std::vector<XXXFactoryCb> ret;
for (ssize_t i = 0; i < filters.size(); i++) {
    //获取当前配置的插件名称，以及对应的插件配置
    const auto& proto_config = filters[i];
    const auto& string_name = proto_config.name();
    const auto& filter_config = getJsonObjectFromMessage(proto_config.config());
    //获取插件对应的实例工厂
    auto& factory = getAndCheckFactory<XXXFactory>(string_name);
    //根据插件配置，实例化插件工厂
    auto message = Utility::translateToFactoryConfig(proto_config, factory);
    ret.push_back(factory.createFilterFactoryFromProto (*message, context));
}

```

上述实例代码最后通过插件工厂模板的createFilterFactoryFromProto方法创建具体的插件工厂，以下面的OriginalDstConfigFactory插件为例，返回的插件工厂采用的是C++11 Lambda表达式（对C++11 Lambda使用方式不太熟悉的读者，可以自行学习下C++11的语法规则，这里不再详述）。

```

Network::ListenerFilterFactoryCb
OriginalDstConfigFactory::createFilterFactoryFromProto(
    const Protobuf::Message&, ListenerFactoryContext&) override {
    //返回C++11 Lambda表达式形式的插件工厂
    return [](Network::ListenerFilterManager& filter_manager) -> void {
        filter_manager.addAcceptFilter(std::make_unique<OriginalDstFilter>());
    };
}

```

使用插件前，根据上述返回的插件工厂列表构造具体的插件实例。下面createListenerFilterChain函数中的listener_filter_factories_就是根据监听器配置得出的监听器工厂列表，遍历每一个工厂方法进行Lambda表达式展开，实际执行的是filter_manager.addAcceptFilter（std::make_unique<OriginalDstFilter>()）语句，addAcceptFilter完成真正的插件实例构造OriginalDstFilter，然后将插件挂载到插件管理器中。

```

bool createListenerFilterChain(ListenerFilterManager& manager) {
    return buildFilterChain(manager, listener_filter_factories_);
}
bool buildFilterChain(FilterManager& filter_manager,
    const std::vector<Network::FilterFactoryCb>& factories) {
    for (const Network::FilterFactoryCb& factory : factories) {
        //这里会对之前返回的C++11 Lambda表达式形式的插件工厂进行Lambda展开
        factory(filter_manager);
    }
    return filter_manager.initializeReadFilters();
}

```

4.3.2 网络相关插件

Envoy自带一些插件，统一放在/source/extensions目录下，最大的一块在filters^[1]子目录下，负责网络方面的插件实现，当前主要有3部分：分别是监听过滤器、网络过滤器和HTTP过滤器，分别负责连接元数据层面的扩展支持、网络协议方面的扩展支持以及基于协议内容的扩展支持，下面会分别讨论。

1. 监听过滤器

监听过滤器在接收到新连接请求，但还未进行实际的新连接创建工作之前触发，通过监听过滤器插件，可以调整连接元数据信息。Envoy当前支持原目的地、原出发地、Proxy Protocol和TLS Inspector这几种监听过滤器。

Envoy会对Istio集群的所有流量进行透明拦截，通过监听端口号获取拦截层发过来的请求，原目的地监听过滤器用于获取拦截前的目的地址，然后基于该目的地址进行后续的路由和转发操作。

出于问题定位和诊断的考虑，不少场景下下游服务希望上游连接采用的是真实的客户端地址，而不是中间的代理服务器地址。Proxy Protocol监听过滤器就是为了实现这个需求，Proxy Protocol监听过滤器基于Proxy Protocol协议。Proxy Protocol协议是HA Proxy作者Willy Tarreau设计的一个网络协议，通过为TCP添加一个额外头部信息，方便传递一些客户端信息到服务端，比如协议栈、源IP地址等，这对复杂网络场景下需要同步获取客户端真实IP地址非常有用。Proxy Protocol监听过滤器基于Proxy Protocol协议获取客户端真实的IP地址，Envoy和下游建立连接时采用客户端请求的源IP地址，而不是Envoy自身的IP地址，下游感知不到Envoy的存在，以为请求仍然是从真正的客户端发过来的，实现了真正的Envoy透传。需要特别注意的一点，由于Proxy Protocol协议对TCP协议进行了定制扩展，和正常的TCP协议不兼容，因此使用Proxy Protocol协议时需要上下游均支持Proxy Protocol协议才可以，否则会导致消息解析出错。

2. 网络过滤器

网络过滤器主要有两个层面的工作：一是连接相关的，主要是连接相关的控制，比如限流、认证等；二是协议相关的处理，Envoy接收到请求时首先需要知道该请求是什么协议，如何进行解码等，协议处理是网络过滤器的主要工作，也是整个Envoy的工作重心所在，Envoy当前支持HTTP、Thrift、MySQL、Kafka、Dubbo等众多网络协议，下面重点讲一下协议处理中的网络过滤器的原理和大体实现。

网络过滤器的首要工作是协议编解码处理，收到请求消息时对请求消息进行解码处理，获取上游流量路由和转发相关的元数据信息，收到上游的响应消息后，进行编码处理，返回给下游客户端；其次是流量路由和转发，根据当前使用场景和请求元数据信息，决策最合适的Upstream上游，这个也是Envoy的主要价值所在；最后一项工作是Upstream管理，获取与Upstream之间的连接，并建立和Upstream之间的安全通信，同时在Upstream异常时进行完善的容错处理。

3. 协议相关过滤器

协议相关过滤器基于协议内容进行过滤和扩展处理，Envoy当前对HTTP协议、Thrift协议等均有过滤器扩展支持，对HTTP协议的支持比较完善些，当前支持路由、限流、Lua扩展等多种扩展方式。

插件扩展需要有相应的扩展点，对于HTTP协议来说，在编解码的不同阶段，均支持相应的扩展点。当前主要的扩展点有对HTTP消息头、消息体、Trailers这几部分的编码和解码。

[1] 这里的filters名称其实不太准确，因为整个extensions目录下都是插件实现，而filters子目录其实对应的是网络方面插件的扩展实现。

4.3.3 其他扩展插件

除了网络相关插件之外，Envoy还内置了一些插件支持，大体分为流量管理、可观测性和服务管理这几个部分。

流量管理方面，支持的主要是各个网络协议均会用到的协议无关的一些插件，比如健康检查扩展支持、重试和传输层扩展支持等。

可观测性是Envoy一个很大的亮点，Envoy在可观测性的几个核心维度——Metric、Log和Trace均有相应的框架支持。

此外，Envoy还内置一些服务管理方面的插件支持，比如资源限制等。

除了官方源码自带的插件外，Envoy还支持以自定义方式进行插件扩展。Envoy原生代码中没有Mixer相关的实现，Istio数据平面对Mixer的支持是通过单独的proxy项目来实现。

4.4 Envoy数据平面API

流量管理的核心配置主要包含监听器配置、路由配置、集群配置以及集群节点配置，这几种配置一块构成了Envoy流量路由和转发的核心。

流量管理配置当前有静态配置和动态配置两种方式。静态配置是在Envoy服务启动时静态指定，动态配置是指在Envoy运行时，通过XDS协议的方式进行实时更新，下面重点讨论XDS协议的实现和机制。

4.4.1 XDS协议语义

XDS协议分为资源请求和资源响应两个阶段，Envoy会发送DiscoveryRequest资源请求消息，DiscoveryRequest具体包含如下字段。

1) version_info: Envoy目前成功处理的最新响应消息版本号。

2) node: 发送请求的Envoy的唯一标识。

3) resource_names: 请求的资源列表，如果为空，会返回该类型所有的资源。对于LDS/CDS类型来说，由于是全局发现，所以这个字段应该填为空；对于RDS/EDS来说，需要根据LDS/CDS发现的结果填写需要的具体资源。

4) type_url: 请求资源类型，比如lds、cds等。

5) response_nonce: Envoy在收到一个DiscoveryResponse之后会马上再发送一个DiscoveryRequest作为ACK/NACK，从而告诉discovery service消息是否成功处理，发送DiscoveryRequest时通过response_nonce字段指定当前的ACK/NACK对应的DiscoveryResponse编号。

6) error_detail: Envoy处理DiscoveryResponse发过来的配置更新消息失败时，会通过error_detail字段描述详细的错误信息，一般在调试阶段使用。

控制平面收到DiscoveryRequest请求消息，会通过DiscoveryResponse消息返回需要的资源信息，DiscoveryResponse的详细字段描述如下。

1) version_info: 当前响应消息的版本号。

2) resources: 资源列表。

3) canary: 用于通过命令行指定Envoy接收XDS更新时的行为，XDS协议不会对这个字段进行任何操作，这里不再详细讨论。

4) type_url: 资源类型，和DiscoveryRequest的type_url字段含义相同。

5) nonce: 响应消息编号。

在XDS请求和响应消息中，version_info和nonce两个字段含义和使用上很容易混淆，这两个字段均由DiscoveryResponse负责设置，Envoy通过发送ACK/NACK时的DiscoveryRequest消息来确认DiscoveryResponse消息是否已正确处理，其中response_nonce表示对某个编号的DiscoveryResponse消息的回复，version_info表示当前处理成功的DiscoveryResponse消息的版本号。

图4-3描述了数据平面和控制平面之间通过XDS协议进行通信的典型流程。

在图4-3中，对于version_info为X，nonce为A的DiscoveryResponse消息，Envoy处理失败，失败回复的DiscoveryRequest中version_info为空，表示当前还没有成功处理的响应消

息，response_nonce字段填的是A，表示是对编号为A的DiscoveryResponse进行确认；然后控制平面发送version_info为Y，nonce为B的DiscoveryResponse消息，此时Envoy处理成功。确认消息DiscoveryRequest中，version_info为Y，表示当前成功处理的版本为Y，response_nonce字段为B，表示对编号为B的消息进行确认。

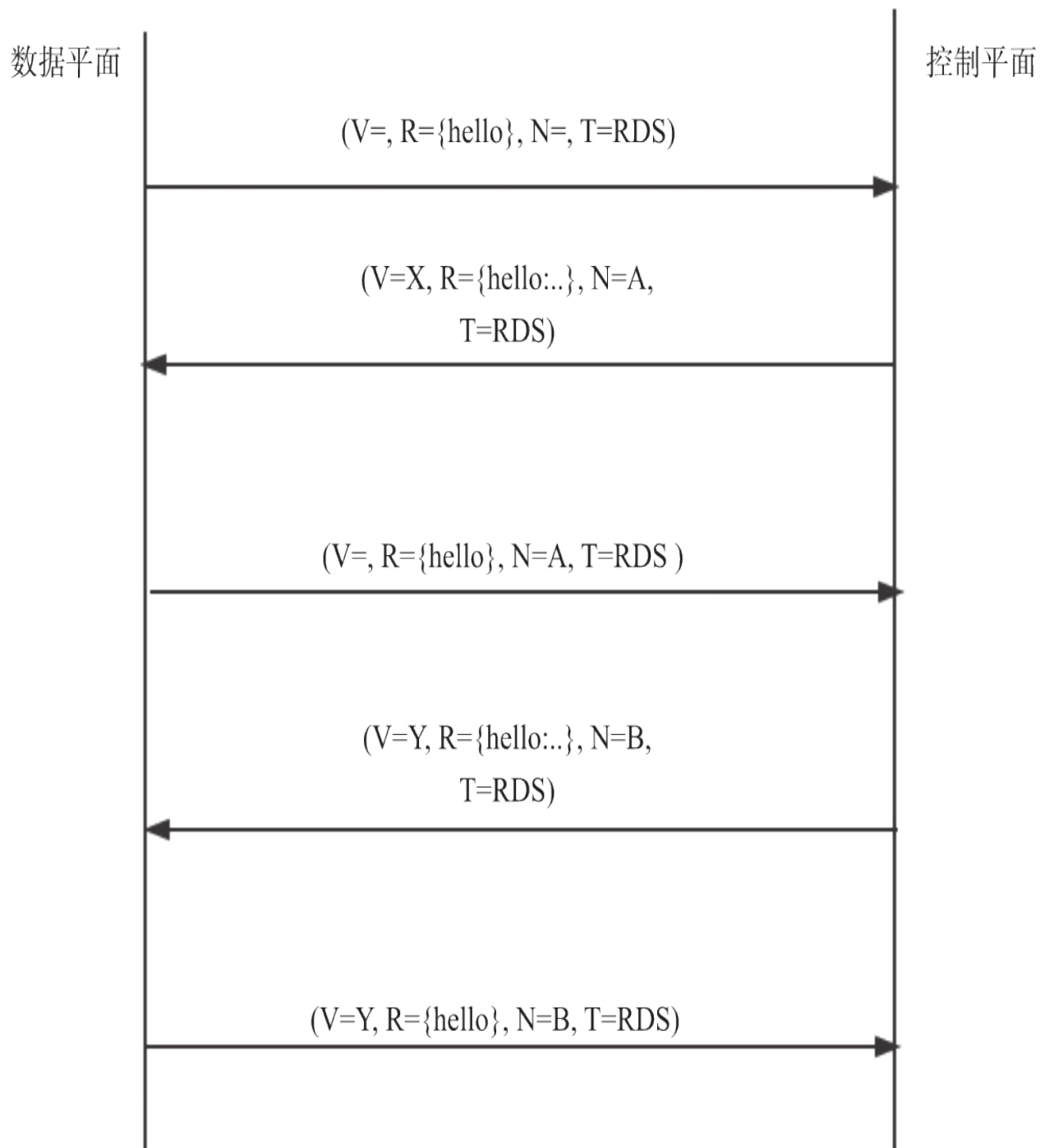


图4-3 Envoy XDS交互图

4.4.2 XDS协议通信

XDS API当前支持两种协议，V1版本的REST协议方式和V2版本的数据平面gRPC协议。V1版本的REST协议针对LDS、CDS、EDS和RDS等。每种XDS资源和Pilot-discovery均会建立一个连接，集群规模比较大时，不仅会有连接资源的浪费，Pilot-discovery服务也会有一定的连接压力。

V1版本中，LDS、CDS、EDS和RDS这些资源使用不同的连接进行管理，独立下发，因此在配置更新过程中，可能会因为资源更新顺序导致更新期间的短暂访问出错，比如先更新RDS信息，路由信息更新后，新路由对应的集群信息还没有触发（由CDS负责更新），这时访问时就会出问题，根本原因是这些资源之间有一定的依赖关系，资源更新顺序也要遵循依赖关系，推荐的更新如下：

- 1) 通过CDS更新集群信息；
- 2) 通过EDS更新集群节点信息；
- 3) 通过LDS更新监听器相关信息；
- 4) 通过RDS更新集群路由信息。

为了解决上述V1版本的REST接口的一些问题和局限性，Istio将数据平面接口升级到了V2版本，V2版本引入了强大的gRPC协议。和REST协议相比，gRPC协议编码效率更高，同时，gRPC支持连接双向通信，带来了通信效率的极大提升。

同时，针对REST协议实现中多个资源更新顺序不当导致的访问出错问题，Istio引入了ADS（Aggregated Discovery Service，聚合发现服务）的概念。ADS将CDS、EDS、LDS、RDS等多个资源均放在一个gRPC连接上进行管理，更新时ADS默认就会按照正确的更新顺序依次更新XDS资源，从根本上解决资源更新顺序不当导致的问题。

REST方式为了兼容旧版本的数据平面，将来会逐步废弃，推荐使用V2版本的gRPC协议方式。

Istio的gRPC接口当前只提供一个全量更新的接口StreamAggregatedResources，每次配置变更时都会返回全量的配置数据，当集群规模比较大时，这个数据量还是挺大的，会给网络带宽带来一定的压力，同时也会给数据平面带来很多不必要的计算。

针对全量更新时遇到的问题，Envoy数据平面当前已经增加了增量更新语义Incremental XDS。在配置发生变更时，Incremental XDS只会对变更部分进行更新，Istio当前还不支持增量更新，将来可能会增加这方面的支持。

4.5 Envoy启动管理

Envoy启动分为通过静态配置启动、通过XDS协议获取动态配置启动以及热重启3种方式，下面分别讨论这几种方式的启动过程。

对于一个健壮的在线服务，大体需要包含如下信息。

（1）网络监听和管理

对于一个网络服务，需要对网络相关的操作进行配置和管理，具体包含监听器信息，网络传输层相关信息以及协议相关信息（比如服务当前具体监听哪些协议，每个协议的监听端口号，监听设置等）。在Envoy中，集中体现为监听器管理`InstanceImpl`、网络`socket`以及传输层具体实现和协议插件支持。

（2）集群管理

集群通信是在线服务，而Proxy转发服务是最重要、最核心的功能。随着业务规模和场景的扩大，对集群管理相关的要求也会越来越高，在Envoy中主要包括集群管理器`ClusterManager`、服务发现、负责均衡、健康检查、节点熔断、流量路由等集群管理最基础的一些功能，也包含比如超时重试、降级限流、Shadow请求等链路治理相关的特性，集群管理是Envoy最核心的价值，也是扩展和变化最多的地方。

（3）可观测性

可观测性是对服务进行全方位、多维度的探测，像医学通过X光对病人的病情进行照射一样，可对服务当前状态进行体检。可观测性当前包括Log、Metric和Trace 3种主要方式。

（4）通信安全

安全性是集群通信过程中非常重要的一个考量。安全如果出问题，会产生灾难性的后果，通信安全包含的维度很多，比如身份认证、安全传输和安全授权等。

下面讨论启动过程中关键流程的初始化和实现。

4.5.1 正常启动

Envoy启动就是一个解析启动配置，初始化Envoy核心数据结构，然后启动工作线程开始工作的过程。

1.配置解析

Envoy启动阶段是个很复杂的过程，具体梳理起来主要包括配置解析、监听管理初始化、工作线程初始化、协议处理插件初始化这样几件事情。启动过程中需要的所有信息通过Envoy启动配置统一进行管理。

Envoy启动配置当前支持V1和V2两个版本，其中V1版本的配置采用JSON格式，并针对JSON格式的配置进行相应的校验。为了方便配置操作，V1版本配置也支持YAML的方式，Envoy内部会在启动的时候将YAML转换为JSON。

JSON格式的配置文件，配置解析比较麻烦，Envoy的启动配置项非常多，光V1版本的启动配置解析的代码量就超过2000行，并且只要对配置项有增加或者修改，都需要修改配置解析代码。为了提高配置解析的效率，Envoy配置文件引入了V2版本，V2版本使用Protobuf 3作为数据平面的API，基于Protobuf 3的V2版本，相比之前的V1版本有很大的改进。从配置文件管理上看，通过引入Protobuf 3作为配置文件的Scheme，利用了Protobuf 3强大的数据描述能力和代码自动生成能力，完全省去了手工配置解析的过程，并且配置文件变化时也不再需要人工修改配置解析过程，完全通过Protobuf 3的代码生成工具一键生成解析代码。同时Protobuf 3强大的数据描述能力，使得对配置文件的正确性验证也非常方便，因此采用Protobuf 3作为配置文件是非常正确的选择，提高了开发效率，并且配置文件的可读性和可维护性也更强。

当前Envoy对V1版本配置的支持是为了兼容旧的版本，V1版本配置将在未来Envoy的版本中被弃用，并最终完全删除。如果你是Envoy的新手，强烈建议从V2配置API开始。

Envoy V2版本的配置文件当前支持JSON、YAML、PB、PB_TEXT等多种格式，配置解析部分的主要工作是根据配置文件的格式，调用Protobuf 3相应的解析函数，将配置文件解析为Bootstrap数据结构的过程，同时会针对该配置进行一定的校验。Bootstrap是个非常重要的数据结构，Bootstrap负责管理Envoy所有的配置参数，它是一个全局的结构，启动过程中的每个阶段都会用到它，Bootstrap的定义如下：

```
message Bootstrap {
  //Envoy节点标识
  envoy.api.v2.core.Node node = 1;
  //静态资源，在启动时通过静态配置指定
  StaticResources static_resources = 2;
  //动态资源，在Envoy启动后，通过XDS协议从控制平面获取
  DynamicResources dynamic_resources = 3;
  //集群管理器
  ClusterManager cluster_manager = 4;
  //管理平面，通过启动一个内置的HTTP服务器，对外界暴露Envoy当前的允许状态
  Admin admin = 12;
  ...
}
```

从Bootstrap的定义可以看出，Bootstrap配置有StaticResources和DynamicResources两种配置。StaticResources是由Envoy启动时静态指定，在这种方式下对Envoy进行配置修改时，必须对Envoy进行重启，操作起来非常不方便；Envoy推荐使用DynamicResources的配置方式，这种方式下配置资源由Istio的控制平面负责管理，以XDS API的方式从Istio的控制平面动态获取，并且在配置发生变更时，控制平面会将最新配置通过XDS API推送给Envoy，实现了配置变更的实时下发和实时生效。

2.核心对象初始化

配置解析完成后，就会进行Envoy核心对象的初始化，先看流量管理相关的几个核心管理类监听管理器ListenerManagerImpl和集群管理器ClusterManagerImpl。ListenerManagerImpl的主要作用是对监听器和工作线程进行管理，这里主要关注监听器管理，工作线程管理放到下面工作线程启动时讨论。Envoy允许同时创建多个监听器，通过ListenerManagerImpl统一进行管理，负责监听器的增、删、查、改，以及生命周期管理；集群管理器ClusterManagerImpl负责管理集群相关信息，如果是静态配置，ClusterManager-Impl构造函数会直接加载集群相关的配置。

其他核心对象，重点关注Envoy管理平面实现AdminImpl。在启动配置中会指定管理平面的监听端口，AdminImpl根据启动配置信息，启动一个简单的HTTP服务器，外界通过访问该HTTP服务器来获取Envoy的当前运行状态，如监听状态、路由配置和集群配置等。

3.工作线程启动

核心对象初始化完成后，Envoy会根据配置的并发度创建一批工作线程，这些工作线程采用异步事件驱动的工作方式，每个线程启动一个独立的事件调度器，当有新的连接到来时，内核会选取一个工作线程进行新连接的处理。

4.5.2 热重启

应用正常重启时，旧进程退出和新进程启动之间很难保证同时进行，可能会导致重启过程中流量损失，为了保证重启过程中业务和流量完全无损，Envoy当前支持优雅的热重启机制。Envoy热重启时，新进程启动时通知父进程热重启消息，同时使用父进程的监听FD创建监听器，启动完成后通知父进程进行退出处理，核心流程大体如图4-4所示。

Envoy热更新的主要逻辑实现在/source/server/hot_restart_impl.cc中。Envoy启动时有个启动参数restartEpoch，标识Envoy服务的当前编号，restartEpoch从0开始编号，每次重启时需要对这个编号加1，因此热重启时新旧Envoy进程的restartEpoch的序号连续，我们就可以基于这个序号关系建立新旧Envoy进程的unix domain socket通信，代码如下：

```
//父进程热重启管理类
HotRestartingParent::HotRestartingParent(int base_id, int restart_epoch)
    : HotRestartingBase(base_id), restart_epoch_(restart_epoch) {
    child_address_ = createDomainSocketAddress(restart_epoch_ + 1, "child");
    bindDomainSocket(restart_epoch_, "parent");
}
//子进程热重启管理类
HotRestartingChild::HotRestartingChild(int base_id, int restart_epoch)
    : HotRestartingBase(base_id), restart_epoch_(restart_epoch) {
    initDomainSocketAddress(&parent_address_);
    if (restart_epoch_ != 0) {
        parent_address_ = createDomainSocketAddress(restart_epoch_ - 1, "parent");
    }
    bindDomainSocket(restart_epoch_, "child");
}
```

Envoy 子进程

Envoy 父进程

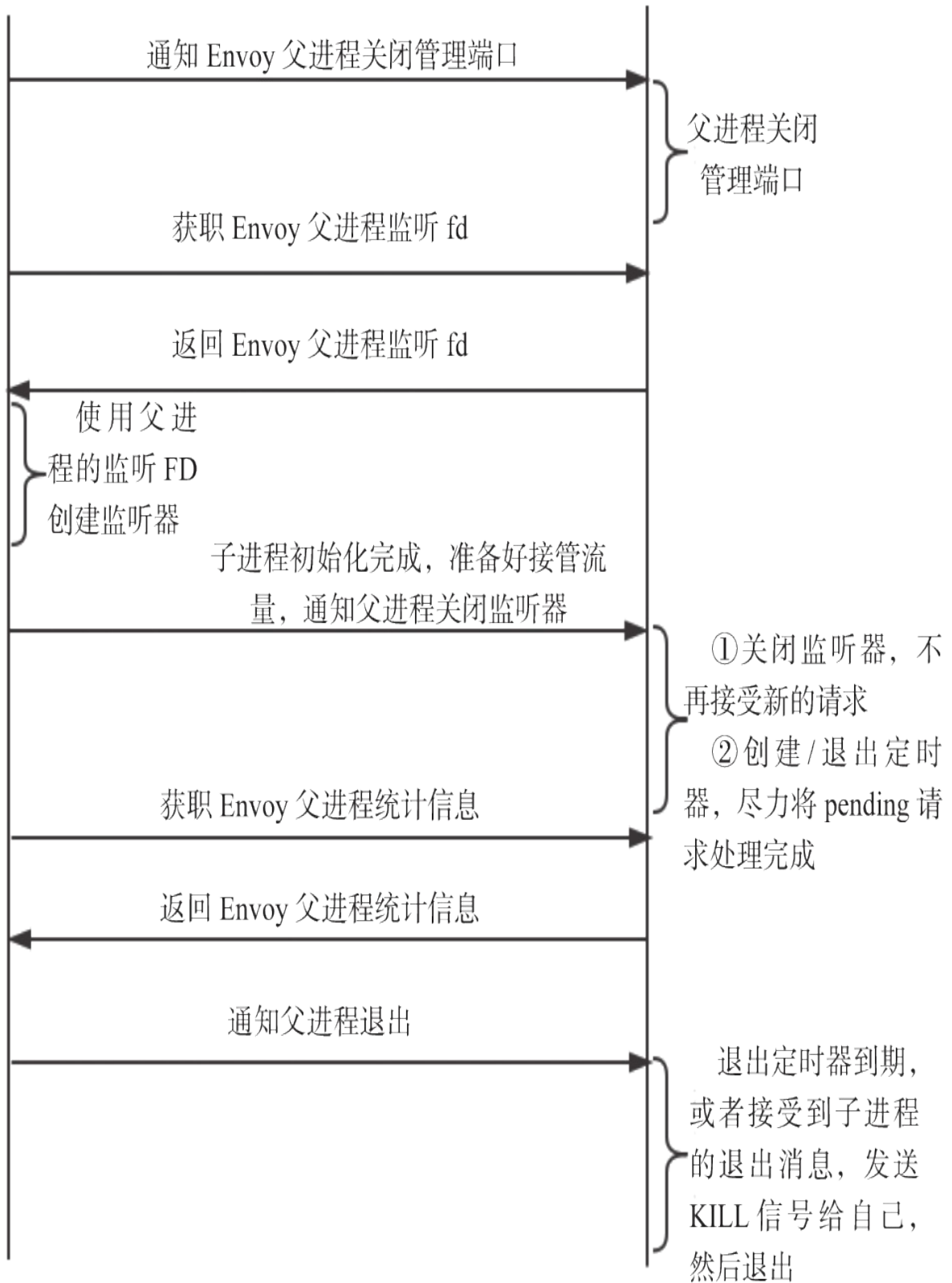


图4-4 Envoy热重启流程图

4.6 Envoy与Nginx架构层面的对比

Nginx是Envoy出现之前网络通信中间件领域非常有代表性的开源系统，功能强大，性能出色，扩展性很强，已经形成了强大的生态，成为HTTP流量管理领域事实上的标杆。Envoy作为后起之秀，虽然定位和目标上与Nginx有不少差异，但架构设计层面，Envoy和Nginx都有很多的可取之处，下面会从功能定位、整体网络模型、连接处理、请求解析、插件机制、配置管理、部署和运维、管理方式等诸多维度，对Envoy和Nginx进行详细剖析和比较，通过与Nginx功能和架构层面的全方位对比，大家也可以对Envoy的架构设计有更立体的认识。

4.6.1 功能与定位

Nginx最核心的功能是Web服务器和反向代理服务器，Web服务器完成对HTTP请求协议的解析和以HTTP协议格式响应请求、缓存、日志处理这些基本Web服务器功能；反向代理服务器完成对请求的转发、负载均衡、鉴权、限流、缓存、日志处理等代理常用功能。

除了对Nginx协议的支持外，Nginx还支持普通的TCP、UDP反向代理功能，同时以stream方式支持通用的基于4层协议的反向代理，比如MySQL代理、Memcached代理等。

Envoy的目标比较远大，定位是透明接管微服务之间的通信流量，将通信和服务治理功能从微服务中解耦，通过Envoy可以方便地增加对自定义协议的支持。

概括起来，Nginx的关键词是Web服务器和反向代理，Envoy是透明接管流量，更加体现对流量的控制和掌控力。

另外，从使用方式上看，微服务对Nginx是显式调用，通过Nginx完成负载均衡等相关功能，对Envoy是隐式调用，业务微服务不需要感知Envoy的存在，和使用Envoy使用相同的方式进行通信，只不过不再需要关注通信和链路治理的细节。

4.6.2 网络模型

网络模型上，Nginx采用的是经典的多进程架构，由master进程和worker进程组成。其中，master进程负责对worker进程进行管理，具体包含监控worker进程的运行状态，根据外部输入的一些管理命令向worker进程发送处理信号以及worker进程退出时启动新的worker进程。worker进程负责处理各种网络事件，各个worker进程之间相互独立，一同竞争来自客户端的新的连接和请求，为了保证请求处理的高效，一个请求处理的全部过程在同一个worker进程中。worker进程的个数推荐配置为与当前环境的CPU核数相同。

自从Nginx诞生以来，一直使用上述经典的多进程架构。这种架构下，请求处理过程中如果遇到特别耗时的操作，比如磁盘访问、第三方服务同步访问等，会导致处理该请求的进程被夯住，不仅CPU资源没有得到充分利用，夯住时间比较长时不仅会影响当前请求，严重时会导致本进程的待处理请求大量超时。为了解决这种问题，Nginx从1.7.11版本开始引入了线程池的概念，如果遇到耗时特别长的逻辑，可以增加线程池配置，放到线程池中进行处理。线程池机制的引入对Nginx架构来说是个很好的补充，通过针对性地解决耗时特别长的一些阻塞场景，使得Nginx的性能达到一个新的高度。

和Nginx不同，Envoy采用了多线程的网络架构，Envoy一般会根据当前CPU核数创建相同个数的worker线程，所有worker线程同时对Envoy配置的监听器进行监听，接受新的连接，为每个新连接实例化相应的过滤器处理链，处理该连接上的所有请求。和Nginx类似，Envoy的每个请求的处理全流程都在同一个线程下进行。

从上面的分析看，Envoy和Nginx的网络处理方式大体类似。这两种方式都是全异步的编程模式，所有的操作都是异步进行，每个执行上下文使用一个单独的事件调度器，对该执行上下文的异步事件进行调度和触发，只是承载网络的执行上下文有差异，Nginx通过多进程的方式承载，Envoy使用的是多线程方式。

Nginx通过线程池的方式，从设计上解决了异步编程中的阻塞问题，但仍然没有从根本上解决这个问题，如果遇到设计或者代码层面没有注意到的问题场景，仍然会出现因为当前请求阻塞导致后续等待的请求得不到处理而超时的现象。由于都是全异步的编程模式，Envoy也会遇到同样的问题，不过Envoy开始尝试着进行解决，具体的解决方式是：为每个worker线程分别设置一个看门狗，并通过定时器定期更新本线程看门狗的更新时间，主线程会监控各个worker线程看门狗一段时间内是否有更新，如果超过一段时间没有更新，可以认为该线程的看门狗定时更新操作得不到执行的机会，从而推断出这个线程当前已经夯住，无法处理请求消息。Envoy通过这种机制可以检测出worker线程是否被长时间阻塞住，在此机制的基础上，后续可以增加相应的处理（比如将待处理请求移到其他线程，然后把该线程杀掉），可以从机制上解决工作线程被阻塞的问题。

4.6.3 连接处理

Nginx通过`worker_connections`参数来控制每个worker能够建立的最大连接数，从Nginx网络模型可以看出，客户端连接到来时，所有空闲的进程都会去竞争这个新连接。这种竞争如果导致某个进程得到的新连接比较多，同时该进程的空闲连接也会很快用完，如果不进行控制，后续该进程获取新连接时会遇到没有空闲连接而丢弃，而有的进程有空闲连接却获取不到新连接。那么直接按照均等的方式将连接分配给各个进程是否可行呢？这种方式其实也是有问题的，不同连接上可能承载的请求QPS差异很大，可能会出现两个进程处理相同连接数，但一个特别忙另外一个特别闲的现象，因此为了保证各个工作进程都能够最大限度地提供自己的计算能力，需要对连接进行精细化管理，Nginx采取的方式是各工作进程根据自身的忙闲程度，动态调整获取新连接的时机，具体实现是：当本进程当前连接数达到最大`worker_connections`的7/8时，本worker进程不会去试图拿`accept`锁，也不会去处理新连接，这样其他worker进程就更有机会去处理监听句柄，建立新连接。而且，由于超时时间的设定，使得没有拿到锁的worker进程去拿锁的频率更高，通过这种方式，Nginx解决了worker进程之间的负载均衡问题。

Envoy也会遇到和Nginx类似的负载不均问题，Envoy当前发展很快，同时需要解决的问题很多，Envoy社区的人觉得这个问题当前的优先级还不够高，后续会根据具体情况对这个问题进行讨论和解决。

4.6.4 插件机制

Nginx拥有强大的插件扩展能力，基于Nginx的插件扩展机制，业务可以非常方便地完成差异化和个性化定制，Nginx插件通过模块的方式提供，具体来说，Nginx主要提供如下几种形式的插件扩展：

- 1) 通过stream机制进行协议扩展，比如增加memcached协议代理和负载均衡等；
- 2) 以Handler方式处理HTTP请求；
- 3) 对HTTP请求和响应消息进行过滤，比如可以修改和定制消息内容等；
- 4) 访问Upstream时的负载均衡，可以提供自定义的负载均衡机制。

对于最成熟的HTTP协议来说，Nginx把整个请求处理过程划分为多个阶段，当前一共有包含读取请求内容、请求地址改写等一共11个处理阶段，业务需要在某个阶段进行扩展和定制处理时，只需要挂载该阶段对应的回调函数，Nginx核心处理HTTP请求到这个阶段时，会回调之前注册的回调函数进行处理。

Nginx对模块的支持总体来说不算灵活，Nginx模块必须和Nginx自身源码一块编译，并且只能在编译期间选择当前支持的模块，不支持运行时进行模块动态选取和加载，大家一直以来吐槽比较多。为了解决这个问题，Nginx在1.9.11版本引入了模块动态加载支持，从此不再需要替换Nginx文件即可增加第三方模块扩展。

Nginx也支持Lua扩展，利用Lua语言的简单易用和强大的协程机制，可以非常方便地实现很多扩展机制，并且性能也能够基本满足需求。

Envoy也提供了强大的插件扩展机制，当前使用最多的地方是监听过滤插件和网络处理过滤插件。和Nginx相比，Envoy网络插件定位在协议层面，以HTTP协议为例，Envoy并没有那么细粒度的插件扩展机制，如果想对Envoy的HTTP协议处理进行扩展，当前并没有提供特别多的扩展点。

Envoy的插件当前采用的是静态注册的方式，插件代码和Envoy代码一块进行编译，和Nginx不同，Envoy从最开始就支持插件的动态加载，Envoy通过独特的XDS API设计，可以随时对Envoy的XDS插件进行定制修改，Istio将修改后的XDS配置通过Grpc的方式推送给Envoy动态加载和生效。

此外，当前Envoy社区和Cilium社区一块探索利用，利用eBPF提供的用户态网络定制能力，对Envoy的流量进行精细化的管理和扩展定制。Cilium从1.3版本开始，引入了Envoy的Go扩展，通过Go扩展实现Filter插件向Envoy注册，主要实现的还是OnData()函数，当Envoy接收到流量时，就会调用插件的OnData函数进行处理。

Envoy在Lua扩展支持方面也进行了一些探索性的工作，当前已经试验性地支持使用Lua脚本对HTTP请求进行过滤和调整。Lua脚本HTTP过滤器当前仍处于实验阶段，不建议直接在生产环境中使用，后续待验证成熟后才能在生产环境使用。成熟后可以在更多的场景下通过Lua脚本机制增强Envoy的扩展性。

4.6.5 配置管理

Nginx的配置格式使用自定义的方式，配置结构和配置解析过程都非常复杂。

配置组织上，Nginx采用层层嵌套的方式，最外层是核心模块的配置，主要包括events事件模块、http模块和stream模块。对于最复杂的http模块来说，下面会嵌套着server子模块，负责监听和管理http请求，server子模块下还会嵌套着一到多个location，用于描述具体的路由规则。

Nginx的配置解析代码可以说是Nginx整个源码体系中最晦涩难懂的部分，配置解析的核心代码在ngx_conf_file.c中，通过层层嵌套解析，形成了一棵树状的配置信息结构。

和Nginx相比，Envoy的配置管理部分设计得就比较优雅，配置格式直接使用了Protobuf 3，复用了Protobuf 3的数据描述能力和自动代码生成机制，完全省去了配置解析的过程，并且也非常方便对配置格式的有效性进行验证。

Nginx的配置都是静态配置，不支持任何形式的动态配置能力。

动态配置能力是Envoy相比Nginx最核心的竞争力，通过动态配置，可以在线修改流量路由和链路治理策略，实现策略配置修改的即时生效。

4.6.6 内存管理

Nginx内存管理通过Nginx内存池实现，通过内存池的良好设计，使用内存池进行内存管理时，不需要关注内存什么时候释放，连接销毁时会回收连接对应的内存资源；同时通过内存池可以减少内存碎片，通过内存对齐、内存分页等机制，可以有效减少缓存未命中（Cache Miss），提高内存访问的效率。

Envoy在内存管理方面的支持还很粗糙，直接基于原生的内存管理库，内存释放时机管理通过智能指针管理内存的生命周期。

和Nginx相比，Envoy在内存管理方面提供的支持不多，完全依赖智能指针机制，对使用者的要求比较高，同时无法满足对性能的极致要求。

4.6.7 部署与运维

从运维上说，Nginx采用多进程的通信架构，主进程负责维护工作进程的状态，主要有以下两方面。

一方面，监控工作进程的状态，状态异常时会对工作进程进行重启；

另一方面，主进程也可以接收外界的管理信号，通知工作进程完成相应的操作，比如通过kill信号通知Nginx优雅重启，主进程在接收到重启信号后，会先重新加载配置，然后启动新进程开始接收新的请求，并向所有工作线程发送信号，通知它们不再接收新的请求，并在处理完所有pending请求后自动退出。

Envoy是通过控制平面组件Pilot Agent和Sidecar-injection进行管理，Pilot Agent负责管理Envoy的运行状态，当Envoy状态有问题时，Pilot Agent会将Envoy重启，如果重启仍然不能解决问题，会尝试将Envoy调度到其他环境下；Sidecar-injection负责Envoy的自动注入。

因此，从架构上看Nginx是自运维的，Envoy是借助控制平面来完成Envoy运行状态的管理，运维上比Nginx方便。

4.6.8 观测与诊断

从服务的可观测以及诊断上说，Envoy投入了大量的精力，已经全面支持Log、Metric、Trace等可观测机制，并且每种观测方式均提供相应的扩展机制，Nginx在这方面相对逊色很多，可观测以及诊断支持上相对弱一些，当前可能是为了和商业版进行区分，Nginx商业版有完善的运维和问题诊断支持。

4.7 本章小结

本章详细讨论了Envoy的架构设计，包括线程模型、事件模型和配置模型等，通过Envoy的架构剖析以及与Nginx的深度比较，可以看出Envoy架构层面的优秀所在。

第5章 Istio架构剖析

本章会从Istio的部署模式、代码结构、流量管理以及Mesh管理、服务管理、通用配置管理等进行展开，从架构设计到实现细节，逐个进行剖析和阐述。

5.1 Istio整体架构

第3章已经简要介绍了Istio的基本功能，Istio在微服务之间建立连接，接管通信功能，对业务微服务屏蔽通信细节，同时通过流量控制、策略控制、遥测统计、Istio安全机制等对微服务进行监控和管理，使得微服务架构更加健壮、安全和易扩展。Istio整体架构如图5-1，整体上分为控制平面（Control Plane）和数据平面（Data Plane）两部分。

控制平面中，istio-telemetry提供可观测性支持，基于istio-telemetry可以掌控整个Istio集群当前的运行状态；istio-pilot和istio-policy分别负责流量控制和策略控制，观测到运行状态不符合预期时，可以通过istio-pilot对流量和路由转发进行控制与调整，通过istio-policy可以灵活动态地引入新的基础设施，对Istio运行进行精细化控制。

下面5.1.1和5.1.2节会分别对数据平面组件和控制平面组件进行详细介绍。

5.1.1 数据平面组件

1.istio/proxy和pilot-agent

istio/proxy是Istio在Envoy官方源码基础上的扩展实现，除了包括Envoy的全部功能外，还包括了一些扩展支持，具体路径为<https://github.com/istio/proxy>。

Istio扩展通过Envoy过滤器插件的方式提供，当前主要扩展是Mixer客户端的支持，基于该扩展可以与Mixer Server进行gRPC交互，完成策略检查、配额控制和遥测统计相关的工作。

pilot-agent是一个本地代理，负责管理istio/proxy服务的整个生命周期。具体包括istio/proxy准备启动参数和配置文件，负责管理istio/proxy的启动过程，istio/proxy启动后的运行状态监控。当监控到istio/proxy运行状态不正常时，pilot-agent负责将istio/proxy重新调整到正常状态，调整策略具体包括重启或者调度到其他节点运行等。

部署上，istio/proxy不会单独产生镜像，而是和pilot-agent一起打包到一个容器镜像中，pilot-agent将会以子进程的方式启动istio/proxy，并监控istio/proxy的运行状态。

pilot-agent和istio/proxy均会启动相应的HTTP管理端口，供外界查询当前的运行状态，如果运行状态不符合预期时，可以采取相应的控制策略。

2.istio-init

istio-init负责业务微服务的流量透明拦截，通过配置相应的流量拦截策略，可以将业务服务的所有出入流量转发到本地的数据平面代理节点处理。

Istio当前使用Iptables作为默认的流量拦截策略，istio-init通过istio-iptables.sh拦截配置，配置相应的Iptables规则，完成具体的流量拦截操作。

3.node-agent

Istio为了实现不同节点之间的安全通信，通信一般是通过TLS的方式进行。为了完成TLS通信，数据平面代理节点需要获取证书和密钥相关的信息。

node-agent就是负责证书和密钥等安全信息的获取，和istio/proxy、pilot-agent以及istio-init的Pod级别的部署方式不同，node-agent当前采用Kubernetes Node级别的部署方式，每个Node上部署一个node-agent。

node-agent作为代理服务，接收数据平面代理节点的证书请求，转换成控制平面需要的格式，请求控制平面的证书管理服务，完成证书和密钥的获取和下发。

5.1.2 控制平面组件

1.istio-pilot

istio-pilot负责Istio数据平面的XDS配置管理，具体包括：①流量管理和服务发现配置等，通过提供通用的流量管理模型和服务发现模型，对接不同的平台适配层；②提供统一的XDS API，供Envoy服务调用。

部署上，通过Kubernetes Service istio-pilot对外提供服务，可以配置为1到多个副本。

Istio当前支持多集群模式，istio-pilot不仅可以负责当前Istio集群的XDS配置管理，同时也可以支持其他Istio集群以远程的方式和istio-pilot进行交互。

2.sidecar-injector

sidecar-injector负责数据平面代理Sidecar的注入，通过Sidecar的透明注入支持，可以减少Istio部署和运维上的复杂度，对Istio的大规模开展有着非常重要的意义。

3.istio-telemetry和istio-policy

istio-telemetry和istio-policy均属于Mixer服务，分别负责遥测统计和策略控制工作。这两个服务的代码和二进制文件完全一样，之所以分开部署，是因为遥测统计和策略控制是两个差异很大的功能特性，分开部署方便采用不同的配置和管控方式，也可通过部署隔离避免相互之间的影响。

4.istio-catedral

istio-catedral属于通信安全，负责安全证书的管理和发放，istio-catedral并不是唯一的证书管理方式，Istio当前支持Catedral、Vault和Google等多种证书管理方式，Catedral是当前默认的证书管理方式。

当采用其他证书管理方式时，不需要部署istio-catedral，比如使用Vault时就不再需要部署istio-catedral，相反则需要使用Vault对应的证书服务器。

5.istio-galley

istio-galley是Istio1.1新引入的配置管理框架，在引入Galley之前，Istio控制平面的各个组件需要分别对Kubernetes资源进行管理，包括资源的配置验证，监控资源配置变化，并针对配置变更采取相应的处理等。

为了减少Istio不同组件配置管理方面的重复工作，Istio将配置管理相关的工作抽象出一个单独的组件Galley，后续其他Istio组件不再需要单独和各种平台上的资源直接打交道，而是使用简明的接口和Galley进行交互，由Galley负责配置验证、配置变更管理、配置源管理、多平台适配等工作。

5.2 Istio的Kubernetes基础

Kubernetes当前已经成为容器编排领域事实上的标准，为了方便Istio的管理，减少重复工作量，Istio很多地方直接复用了Kubernetes的基础能力，最大程度利用Kubernetes强大基础设施的优势，因此讨论Istio控制流需要首先了解Kubernetes相关的一些背景知识。为了让之前对Kubernetes不太了解的读者对kubernetes有整体的认识，下面先介绍下和Istio关系比较大的一些kubernetes基础知识。

Kubernetes是一个管理容器化应用和服务的开源平台，当前已经发展成一个非常庞大的生态系统，已经成为容器调度领域事实上的标准。Kubernetes最主要的工作是负责容器化服务的调度和生命周期管理，而容器化服务之间的通信问题则交给Istio解决。Istio作为致力于解决微服务通信问题的通信平台，有很多地方需要借助Kubernetes提供的平台和基础设施能力。

第5章重点围绕Kubernetes和Istio的管理展开，首先总体介绍Kubernetes的一些基础知识，然后重点讨论Kubernetes网络模型和Kubernetes下的网络访问模式，剖析Kubernetes网络管理的当前局限性，最后从Kubernetes和Istio的关系入手，介绍Kubernetes中和Istio密切相关的一些特性，了解这些特性对接下来深度剖析Istio的架构会有很大的帮助。

5.2.1 Kubernetes综述

Kubernetes是Google 2014启动的开源项目，它基于Google长达15年生产级环境下的集群管理经验，同时结合来自社区的最佳创意和实践经验，使用Kubernetes，你可以在容器化云环境下快速、高效地满足用户以下几个方面的需求：

- 1) 快速精准的部署应用程序；
- 2) 随时对你的应用程序进行伸缩；
- 3) 支持在数据中心范围内调度资源，避免不必要的资源预留和浪费。

1.Kubernetes设计理念

(1) 声明式API设计

相对于命令式操作来说，声明式操作更容易被用户理解，基于Kubernetes声明式API可以以非常小的代价自定义扩展新的API。

(2) 容错设计

在分布式系统中局部错误是大概率事件，错误可能来自于自身代码问题，也可能来自于系统级别的底层故障，保证系统不出错是不可能实现的，因此系统设计时需要尽可能地对所有可能的错误进行容错处理。

分析和理解Kubernetes的设计理念，可以让我们更深入地理解Kubernetes的原理和架构，更好地使用它进行容器化服务管理，同时Kubernetes先进的设计理念也可以在设计其他系统时给我们一定的启发。

2.Kubernetes核心概念

下面介绍Kubernetes中与服务以及服务通信关系比较大的几个概念。

(1) Pod

Kubernetes并不直接运行容器，而是通过一个抽象层来包装一个或者多个容器，这个抽象被称为Pod。Pod由一个或者多个容器组成，Pod内部的容器共享网络和存储资源。Pod是Kubernetes创建或部署的基本单元，创建的Pod被Kubernetes调度到集群的某个Node上运行。虽然Pod中包含多个容器，为了保证Pod调度时的精细化，还是要保证Pod足够轻量，通常一个Pod中运行一个容器，或者紧密耦合的多个容器。

(2) Replication Controller

Replication Controller（副本管理器）用于Pod的管理，它能够保证Pod持续运行。

①当Pod运行状态有问题时，Replication Controller会杀掉故障Pod，重新创建新的Pod。②Replication Controller会确保在任何时候都有指定数量的Pod副本，如果少于指定

数量的副本，Replication Controller会负责创建新的副本；反之如果副本过多则会关闭多余的Pod副本。③Replication Controller通过监控系统资源利用率变化，实现对Pod进行自动伸缩控制。

还有两个和Replication Controller比较容易混淆的概念——ReplicaSet与Deployment。

（3）ReplicaSet和Deployment

ReplicaSet是Replication Controller的升级版本，和Replication Controller的唯一差别是ReplicaSet增加了选择器支持；Deployment是新一代的Pod管理对象，与Replication Controller相比，它提供了更加完善的功能，使用起来也更加简单方便，只需要在Deployment中描述你需要的目标状态，Deployment Controller就会帮你将Pod和ReplicaSet的实际状态改变为你的目标状态。你可以定义一个全新的Deployment来创建ReplicaSet，或者删除已有的Deployment并创建一个新的来替换。当我们创建了Deployment之后，实际上也创建了ReplicaSet，所以Deployment管理着ReplicaSet（实际上Deployment比ReplicaSet有着更多功能），Deployment使用了Replica Set，是更高一层的抽象概念。除非需要自定义升级功能或者根本不需要升级Pod，否则还是建议使用Deployment而不是直接使用ReplicaSet。

（4）Label

Label是附着在对象上的key-value对，可以在创建对象的时候指定，也可以在对象创建后通过API的方式指定。通过给一个对象定义多个Label，比如部署不同版本的应用到多个环境中，实现对象的精细化管理和控制。Label Selectors用于对Label进行筛选和引用，通常我们通过描述文件中的spec.selector字段来指定Label，从而Kubernetes可寻找到所有包含你指定Label的对象进行管理。

3.Kubernetes整体架构

架构层面，Kubernetes为主从分布式架构，主要由负责调度与控制逻辑的Master节点和执行实际工作的Node节点组成。

Master节点主要包含如下几个部分。

（1）API Server（管理服务器）

API Server负责Kubernetes的对外交互，提供了操作Kubernetes资源的唯一入口，对Kubernetes集群进行的所有修改、查询和管理都需要通过API Server进行。

此外，API Server还扮演了Kubernetes内部消息总线的角色，是Kubernetes集群内部各个模块之间通信的枢纽，其他模块之间不会直接进行调用，而是通过和API Server进行交互来完成自己负责的具体工作。

（2）Scheduler

Scheduler（调度器）负责节点的资源管理和调度，主要工作是接收API Server的Pod管理任务，并分配到具体的节点上，并把调度结果写到etcd中。

(3) Controller Manager

Controller Manager（控制管理器）负责维护集群的状态，Kubernetes当前默认提供Replication Controller、Node Controller、Namespace Controller、Service Controller、Endpoints Controller、DaemonSet Controller等控制器。

(4) 集群资源和状态存储

Kubernetes默认使用etcd作为集群数据存储，当然你也可以替换成其他的存储方案。etcd是一个开源的分布式key-value存储系统，主要用来存储共享配置和服务发现信息，etcd提供了一套完善的REST API接口，使用起来非常方便。etcd提供注册接口，用于监控存储信息的变更，Kubernetes当前所有的资源信息都存储在etcd中，并监控etcd资源的状态变化，当etcd中的信息发生变化时，就可以快速通知集群中相关的组件。

Node节点主要包含如下几个部分。

(1) Kubelet

Kubelet（节点管理器）实现对工作节点的管理，并定期将节点上容器的运行状态汇报给API Server，并通过cAdvisor监控容器和节点资源，节点管理器具体管理的资源包括Pod、容器、镜像、数据卷等。

(2) Kube Proxy（节点代理）

Kube Proxy是Kubernetes在每个节点上运行的网络代理，负责Pod的访问。Kube Proxy负责为Pod创建代理服务，实现服务到Pod的路由和转发。

除了上述核心组件，Kubernetes还有一些比较重要的外围组件，其中一个重要的组件是kubectl。kubectl是用于运行Kubernetes集群命令的管理工具，kubectl内部实现会调用Kubernetes API Server，对Kubernetes状态进行管理。

5.2.2 Kubernetes网络访问模型

1.Kubernetes网络模型

讨论Kubernetes网络模型之前，先看一下Kubernetes的网络设计原则：每个Pod都拥有一个独立的IP地址（IP Per Pod），而且假定所有的Pod都在一个可以直接连通的、扁平的网络空间中。

Kubernetes网络设计原则是Kubernetes网络模型的基础，不管你怎么设计Kubernetes网络，Kubernetes网络设计原则的唯一要求是Kubernetes的所有Pod IP均可以直接被其他所有Pod访问，不管Pod存放在哪个节点。

根据场景的不同，Kubernetes网络访问可以分为如下几种典型情况：

- 1) 同一Pod内的网络通信；
- 2) Pod之间的网络通信；
- 3) Pod到Service之间的通信；
- 4) Kubernetes外部服务访问Pod时的通信；
- 5) Pod访问Kubernetes外部服务时的通信。

下面会针对上面列举的每种情况进行详细讨论。

同一Pod内的多个容器，共享一个Pod IP地址，共用同一个Linux协议栈，可以理解为在同一网络里直接通过localhost+port进行访问，也可以通过共享文件系统来完成Pod内不同容器的通信。

Kubernetes网络模型保证：同一个Kubernetes集群内的任意Pod之间均可以相互访问，因此不同Pod之间通信只要能获取到远端Pod的IP地址，就可以通过这个IP进行访问。

由于Kubernetes集群中容器特别轻量，服务被调度到一组Pod上可能要一段时间后，由于各种原因，服务会被调度到其他Pod上，由于服务对应的Pod地址经常变动，直接通过Pod IP进行访问，在现实上不太可行。

在Kubernetes中，在受到Replication Controller调度的时候，Pod副本是变化的，对应的Pod IP也跟着变化，比如发生迁移或者伸缩的时候，这对于Pod的访问者来说是不可接受的。Kubernetes引入了Service的概念，提供了一套简化的服务代理和发现机制，天然适应微服务架构，用于Kubernetes中不同服务之间的通信。

2.Kubernetes Service

Kubernetes中的Service是一种抽象概念，它定义了一个Pod逻辑集合以及访问它们的策略。Service同Pod的关联关系是通过Label来完成。Service的目标是提供一种桥梁，它会为访问者提供一个固定访问地址，用于在访问时重定向到相应的后端，这使得非

Kubernetes原生应用程序，在无须为Kubernetes编写特定代码的前提下可以轻松访问后端。

Kubernetes Service有3种使用类型。

(1) ClusterIP

ClusterIP是默认的使用方式，自动分配一个仅Kubernetes集群内部可以访问的虚拟IP地址，用于Kubernetes内部服务之间的访问，在不需要考虑外部访问的场景时，ClusterIP是个非常合适的选择。

(2) NodePort

为每个Service在Kubernetes的每台机器上绑定一个端口，这样就可以通过<NodeIP>:NodePort来访问服务，从而实现Kubernetes内部与外部服务均可访问。

(3) Ingress LoadBalancer

在NodePort的基础上创建一个外部的负载均衡器，其他服务访问时先访问负载均衡器，由负载均衡器负责将请求转发到相应的Pod上。

Kubernetes中，Service的具体实现由Kube Proxy负责，实现了内部从Pod到Service及外部的从NodePort向Service的访问。

Kube Proxy目前有userspace、Iptables和IPVS 3种实现方式，userspace是在用户空间，通过Kuber Proxy实现负载均衡的代理服务。这个是Kube Proxy的最初版本，较为稳定，但是效率不高，当前不推荐使用。

Iptables方式因为使用Linux的Iptable NAT来完成转发，也存在不可忽视的性能损耗。在集群规模比较大，Iptables规则特别庞大时，Iptables的性能急剧恶化。

针对Iptables的性能问题，从Kubernetes1.8版本开始，新增了Kube Proxy对IPVS的支持，并且在Kubernetes 1.11 GA版本中被纳入了GA。和Iptables相比，IPVS为大型集群提供了更好的可扩展性和性能，IPVS支持比Iptables更复杂的负载均衡算法（最小负载、最少连接、加权等），同时IPVS支持服务器健康检查和连接重试等功能。

3.Kubernetes服务发现机制

Kubernetes的Service提供了一个通过虚拟IP地址访问Pod的方式，但实际上应用只知道被调用服务的名字，并不知道其对应的虚拟IP地址，因此需要有一个机制，可以根据服务名获取它实际的IP地址，这样用户使用时只需要使用服务名即可，这种机制就是服务发现。

Kubernetes支持两种服务发现的方式。

(1) 环境变量

Pod创建的时候，服务的IP和Port会以环境变量的形式注入到Pod里，比如Pod创建时有一个redis-master服务，服务IP地址是10.0.0.11，Port是6379，则会把下面一系列环境变

量注入到Pod里，再通过这些环境变量访问redis-master服务。

这种方式扩展性比较差，在集群规模变大时运维比较困难。

(2) DNS

Kubernetes集群会内置一个DNS服务器，Service创建成功后，会在DNS服务器里导入一些记录，想要访问某个服务，通过DNS服务器解析出对应的IP和Port，从而实现服务访问。

Kubernetes当前通过kube-dns来完成DNS的工作，具体通过kubedns、dnsmasq等协作完成。kubedns监控Kubernetes节点和服务变化，然后根据Kubernetes DNS规范处理DNS记录；dnsmasq提供缓存和存根域支持。

该架构下，dnsmasq中的安全漏洞导致以前需要发布针对Kubernetes的安全补丁。此外，由于dnsmasq处理存根域，但kubedns处理外部服务，因此你无法在外部服务中使用存根域，会导致一定的扩展性问题。

针对kubedns的扩展性问题，Kubernetes引入了CoreDNS解决方案，CoreDNS实现中，所有DNS功能均在一个独立的服务中完成，通过不同的扩展性插件来对kubedns的功能进行增强。

5.2.3 Kubernetes API管理

1.声明式API接口设计

在Kubernetes中，一切都是对象，Kubernetes对象yaml文件进行定义，通过yaml文件可以定义不同类型的对象，Kubernetes用etcd存储Kubernetes对象进行存储。

Kubernetes对象包含两个属性：**Spec**（对象定义）和**Status**（对象状态）。Spec包含了你期望对象应有的状态，Status是该对象当前实际的状态。对于Kubernetes来说，在任何状况下，它都会尽力确保对象的状态处于你所期望的状态。

创建一个Kubernetes对象通常分为两步：创建对象描述文件，然后通过kubectl命令行接口创建对象。Kubernetes对象描述文件是一个符合yaml语法的文件，在对象描述文件.yaml中必须包含以下字段。

- apiVersion: Kubernetes API的版本。
- kind: Kubernetes对象类型。
- metadata: 唯一标识该对象的元数据，包括name、UID、可选的namespace。
- spec: 标识对象的详细信息，不同对象的spec的格式不同，可以嵌套其他对象的字段。

创建好对象描述文件后，通过kubectl apply -f xxx.yaml发布即可。

下面是一个Kubernetes对象的简单例子，表示一个运行nginx程序的Pod对象：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

Kubernetes 1.7之后，提供了CRD（Custom Resource Definitions，自定义资源定义），通过Kubernetes CRD，可以向Kubernetes API中增加新的资源类型，会比修改Kubernetes的源代码或者创建自定义的API Server来得更加简洁和容易，并且不会随着Kubernetes内核版本的升级而需要代码重新合并，也不会出现兼容性方面的问题。这一功能特性大大提升了Kubernetes的扩展能力。

2.API处理模型

Kubernetes API Server负责对Kubernetes API进行处理，具体处理包括准入控制、资源

处理和存储处理3部分，Kubernetes对象处理流程如图5-2所示。

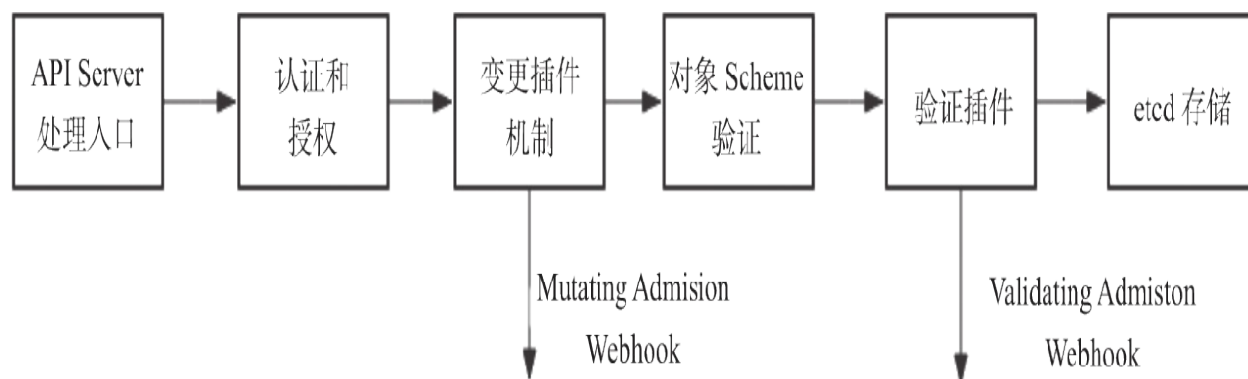


图5-2 Kubernetes对象处理流程

如图5-2所示，认证、授权等机制属于准入控制的范畴，通过各种准入机制保证Kubernetes资源的安全性和隔离性；变更和验证插件是Kubernetes非常有特色的特性，通过Admission Webhook机制对Kubernetes资源进行修改和验证。

Admission Webhook插件机制是Kubernetes 1.9引入的扩展性机制，通过Admission Webhook插件，外界应用可以在不影响Kubernetes核心的情况下，对Kubernetes资源进行变更和验证。外部系统只需要预先注册相应的Admission Webhook处理逻辑，具体包括关注的资源和触发时机，API Server在相应Kubernetes资源处理过程中，会触发之前注册的插件，实现诸如资源变更或者资源验证的工作。

资源处理完成后，Kubernetes将对象资源存储到etcd中，并且通过相应的机制，通知当前对象的关注者，进行对象变更相关的处理。

5.2.4 Istio与Kubernetes的相互关系

Kubernetes当前通过Kube Proxy只能提供基本的通信能力，对于精细粒度的集群管理、流量调度、服务发现、负载均衡等，还需要业务服务自己解决。

Kubernetes的定位是服务调度和服务生命周期管理，微服务通信本身不属于Kubernetes的重点支持范畴。为了减少微服务上云的复杂度，同时为了实现Kubernetes上微服务的标准化，需要有一套完善的基础设施，系统地解决微服务的通信问题，Istio正是解决Kubernetes集群的通信标准化问题，提供业务透明的通信解决方案。

Kubernetes提供的强大的基础设施，为Istio的运维提供了很大的帮助，Service Mesh之前很长时间内没有流行起来，其根本原因是运维复杂度偏高，导致Service Mesh使用起来难度很大。基于Kubernetes提供的云化基础设施，有力地推动了Service Mesh的快速发展和落地。

5.3 Istio流量控制模型

Pilot作为Istio的配置中心，一方面接收istioctl触发的控制面信息，主要是一些Istio核心对象的变更；另一方面接收服务注册中心的服务注册信息。Pilot将上述两部分信息进行一定的处理，发送给相应的Envoy，作为Envoy的动态配置。

Istio有着强大的扩展性支持，不仅支持Kubernetes、Mesos、Cloud Foundry等众多云平台，同时每种云平台内部也支持不同的服务注册中心和对象管理方式。为了对这些扩展进行统一的管理，Pilot定义了一个抽象的模型层，给不同层面的扩展提供一致的视图，当前主要包含对象通用配置和服务通用配置，分别聚焦核心对象抽象管理和Istio服务抽象管理。

5.3.1 流量管理API

Istio流量管理相关的核心资源采用Kubernetes CRD的方式进行管理配置，对于VirtualService、DestinationRule、ServiceEntry、Gateway、EnvoyFilter Kubernetes CRD等资源进行更新操作，直接通过kubectl工具来操作。之前的Istio版本中，Istioctl对kubectl操作Kubernetes CRD资源进行了一些封装，通过kubectl或者Istioctl均可以对Istio的Kubernetes CRD进行操作，当前Istioctl的实现中已经去掉了Kubernetes CRD资源的相关支持，对Kubernetes CRD资源的操作只能通过Kubectl工具来进行。

Istioctl负责Istio中Kubernetes CRD资源之外的修改和查询操作，比如通过Istioctl proxy-config获取Proxy相关的配置，通过Istioctl kube-inject将Envoy Sidecar注入到Kubernetes Pod中。

Istio API负责对Istio数据平面进行控制和管理，自Istio诞生以来，Istio API也发生了很大的变化。当前Istio API使用的是Istio 0.8版本开始引入的networking.istio.io/v1alpha3，当前仍处于v1alpha阶段，可以看出API这块仍然会有不少变化，距离稳定状态还有一段距离。Istio API设计上非常简洁，通过VirtualService、DestinationRule、EnvoyFilter、Gateway、ServiceEntry以及Sidecar 6种Istio资源完成整个Service Mesh的流量管理工作，Istio资源定义在vendor/istio.io/api/networking/v1alpha3/目录下，下面会依次对这几个Istio资源进行详细介绍。

1.VirtualService

VirtualService负责Istio的路由管理，VirtualService定义了一系列路由规则，通过这些路由规则，实现请求在Istio网络中的路由和转发控制。路由就是查找满足最佳匹配条件的目标地址的过程，因此路由规则主要包含路由匹配条件和路由目的地两部分。VirtualService对象具体定义如下：

```
message VirtualService {
  repeated string hosts = 1;
  repeated string gateways = 2;
  repeated HTTPRoute http = 3;
  repeated TCPRoute tcp = 4;
  ConfigScope config_scope = 6;
}
```

其中hosts字段表示请求服务的服务名，一般通过服务注册中心定义，在Istio中注册中心获取。Gateways表示使用当前路由规则的Envoy实例，通常指Istio Gateway或者Istio Sidecar，通过Gateways实现路由和具体Envoy实例的绑定；config_scope表示当前路由规则的可见性，设置成PRIVATE表示路由作用域是Service Mesh范围内，不能用于Service Mesh网络之外，设置成PUBLIC表示该路由是公开的，使用范围没有限制。

http和tcp分别表示HTTP协议与TCP协议对应的路由规则，其中tcp路由实现较为简单，下面先看TCPRoute的具体实现，TCPRoute由路由匹配条件和路由目的地两部分组成，TCPRoute的路由匹配条件由L4MatchAttributes定义：

```
message L4MatchAttributes {
```

```
repeated string destination_subnets = 1;
uint32 port = 2;
string source_subnet = 3;
map<string, string> source_labels = 4;
repeated string gateways = 5;
}
```

L4MatchAttributes中定义了一系列的匹配条件，比如源地址子网掩码、目标地址子网掩码、标签匹配等，**RouteDestination**定义了具体的目标地址以及相应的权重，通过调整各个目标地址的权重，就可以实现不同的路由策略。

```
message RouteDestination {
  Destination destination = 1;
  int32 weight = 2;
}
```

路由目标地址由**Destination**负责管理，**Destination**中**subset**是一个非常重要的概念，通过指定相应的**subset**，我们可以将服务划分为多个抽象的子域，通过**subset**可以根据具体场景灵活定制相应的路由策略，比如AB Test、灰度发布等场景。

```
message Destination {
  string host = 1;
  string subset = 2;
  PortSelector port = 3;
}
```

HTTPRoute负责对HTTP路由进行管理，**HTTPRoute**不仅支持常规的路由匹配和路由查找，还可以支持重定向、重写、重试、故障注入、流量复制等复杂特性，对流量进行精细粒度对控制。

```
message HTTPRoute {
  repeated HTTPMatchRequest match = 1;
  repeated HTTPRouteDestination route = 2;
  HTTPRedirect redirect = 3;
  HTTPRewrite rewrite = 4;
  bool websocket_upgrade = 5;
  google.protobuf.Duration timeout = 6;
  HTTPRetry retries = 7;
  HTTPFaultInjection fault = 8;           //故障注入
  Destination mirror = 9;                 //流量复制
  CorsPolicy cors_policy = 10;
  Headers headers = 16;
}
```

下面是**VirtualService**的一个简单使用实例，默认情况下reviews服务的所有流量均发送到v1版本。对于前缀为“/wpcatalog”的HTTP请求，重写到“/newcatalog”，发送到v2版本。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews-route
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  http:
  - match:
    - uri:
```

```
    prefix: "/wpcatalog"
  rewrite:
    uri: "/newcatalog"
  route:
  - destination:
    host: reviews.prod.svc.cluster.local
    subset: v2
  - route:
  - destination:
    host: reviews.prod.svc.cluster.local
    subset: v1
```

2. DestinationRule

DestinationRule负责Istio路由选取后的流量策略，具体包含负载均衡、连接池管理、节点熔断、subset选取等。DestinationRule和VirtualService一块配合完成Istio流量管理工作，其中VirtualService负责路由选取，DestinationRule负责路由选取后的策略处理，具体包含服务的流量策略以及服务下各个subset的流量策略，DestinationRule对象具体定义如下：

```
message DestinationRule {
  string host = 1;
  TrafficPolicy traffic_policy = 2;
  repeated Subset subsets = 3;
  ConfigScope config_scope = 4;
}
```

其中，TrafficPolicy表示具体的流量策略，具体包含负载均衡策略LoadBalancerSettings、连接池策略ConnectionPoolSettings等；Subset用于对subset进行描述，name和labels一块确定了Subset的范围。

```
message Subset {
  string name = 1;
  map<string, string> labels = 2;
  TrafficPolicy traffic_policy = 3;
}
```

3. Gateway

在Kubernetes环境中，Kubernetes Ingress配置需要在集群外部访问的服务，受限于Kubernetes对Ingress资源的描述，Kubernetes Ingress Controller只能通过各种annotation表达式来支持路由特性，这种方式移植性差，并且对一些复杂的路由场景支持不太友好。

为了解决Kubernetes Ingress带来的问题，Istio引入了Gateway的概念，通过Kubernetes CRD的方式定义了一个新的资源。和Kubernetes Ingress相比，语义层面更加清晰，并且可以方便地支持各种复杂的路由需求。

```
message Gateway {
  repeated Server servers = 1;
  map<string, string> selector = 2;
}
```

Gateway实现中，通过Server定义了网关的行为，通过selector标签选择器来选择和当

前Gateway匹配的数据平面节点。在Server中，port和bind描述了Gateway的部署信息，hosts为Gateway需要暴露的服务列表，前面4.4.1节中已经提到，VirtualService的gateway字段用于将路由信息和gateway进行绑定，通过Gateway和VirtualService，内部服务通过Gateway方式访问，并关联上具体的路由规则，外部服务就可以通过Gateway完成对内部服务的访问。

```
message Server {
  Port port = 1;
  string bind = 4;
  repeated string hosts = 2;
  TLSOptions tls = 3;
  string default_endpoint = 5;
}
```

5.3.2 Istio Mesh模型

为了完成不同场景、不同需求下的Mesh配置和下发，Istio提供了一组API接口，对Sidecar进行配置，Envoy获取XDS配置时，Istio就会根据当前Envoy的配置信息，获取匹配的XDS配置。Istio Mesh主要由MeshConfig、MeshNetwork和ProxyConfig这几个维度，分别负责对Service Mesh整体配置、Service Mesh网络配置以及对某些Sidecar实例单独进行的个性化配置。

1.MeshConfig

MeshConfig负责Istio集群Mesh相关的整体配置，具体分为如下几个方面。

(1) Envoy XDS相关配置

对于Envoy动态配置方式来说，需要设置XDS相关的行为，MeshConfig通过config_sources字段设置XDS Server的地址，Envoy通过该地址就可以获取相应的动态配置信息。

(2) Mixer相关配置

1) MeshConfig中会分别通过mixer_check_server和mixer_report_server指定对应的Mixer Server地址，这里对策略检查和遥测统计分别使用不同的Mixer Server配置，使用的时候比较灵活。

2) disable_policy_checks用于关闭Mixer策略检查特性，对一些性能敏感型的应用服务，可以关闭Mixer策略检查减少Mixer带来的性能开销，默认情况下策略检查特性处于开启状态。

3) policy_check_fail_open用于指定连不上Mixer时策略检查的行为，如果请求策略检查时发现连不上Mixer是否允许当前请求通过，默认情况下把Mixer作为强依赖，连不上时直接拒绝请求，不允许请求通过。个人觉得Istio的这个Mixer检查失败默认行为有待商榷，Mixer本质上不属于Istio请求转发核心链路的一部分，属于可降级的流程，因此Mixer连不上时应该允许请求通过，优先保障业务的稳定性和可用性。

(3) Envoy监听和通信连接相关配置

MeshConfig分别通过proxy_listen_port和proxy_http_port设置整个Istio集群Envoy的监听端口号以及http proxy请求监听端口号，同时通过connect_timeout设置连接超时。

(4) 流量控制相关配置

MeshConfig通过outbound_traffic_policy指定应用访问Istio集群外部服务的行为，一般情况下Istio只允许访问服务注册中心中注册过的服务，推荐通过ServiceEntry配置对Istio集群外部服务的访问。

(5) 通信安全相关配置

Envoy需要和当前节点上的NodeAgent通信，获取双向TLS通信需要的证书信息，MeshConfig通过sds_uds_path字段设置和NodeAgent通信的Unix Domain Socket地址。

(6) Envoy可观测性配置

MeshConfig通过enable_tracing设置是否产生RequestID和开启trace特性，通过access_log_file和access_log_format等设置Envoy访问日志的行为。

此外，MeshConfig会设置默认ProxyConfig，用于实现Sidecar自动注入机制。

2.MeshNetwork

MeshNetworks负责Mesh集群内部的网络访问和路由设置，Mesh集群可以设置若干个Network，每个Network通过唯一的网络名进行标识。Network设置主要有两类：一类是网络中的节点信息，通过NetworkEndpoints设置；另一类是Gateway相关设置，外界访问Istio内部服务时的Gateway地址通过IstioNetworkGateway进行设置。

每个NetworkEndpoints代表可以被Istio集群访问到的一组终端节点集合，Network-Endpoints可以通过显式和隐式两种方式指定。

1) 显式方式可以有两种手段：一种是通过from_registries指定服务注册中心名，另一种是通过from_cidr指定节点的CIDR（无类域间路由，可以通过子网掩码指定一个网络）范围。

2) 隐式方式是通过环境变量ISTIO_META_NETWORK设置对应的节点信息。

3.ProxyConfig

ProxyConfig用于定义Envoy的定制化行为，MeshConfig会指定Istio集群所有Envoy的共同行为，可以通过ProxyConfig进行场景化定制，比如Envoy配置文件地址、管理平面端口号、工作线程并发度等，通过ProxyConfig确定了Envoy启动时的具体参数和行为。

5.4 Mixer模型

Mixer是负责提供策略控制和遥测统计的Istio组件，具体实现如下几个功能。

（1）调用前置检查

Envoy将请求透传给下游服务前，会通过调用Mixer判断该请求是否满足一定的前置条件，具体比如黑名单检查，访问权限校验等。

（2）配额检查

配额检查调用时机和前置检查类似，由Envoy调用下游服务前调用Mixer进行检查，不同的是配额检查聚焦资源的管理，以实现资源的有效管理，具体比如限流、连接数检查等。

（3）统计观测信息上报

服务处理完请求后，Envoy将请求相关的统计信息上报到Mixer，Mixer及其Mixer相关的基础设施，会基于上报数据进行监控、性能分析等，也可以基于对数据的精细化分析，产生相应的策略对链路进行管控，具体的上报信息包含Log、Metric和Trace等。

5.4.1 Mixer基本概念

1.适配器

在对Mixer进行深入剖析前，先介绍下适配器（Adapter）。Mixer处理不同基础设施后端的灵活性是通过使用通用插件模型实现的。每个插件都被称为Adapter，通过通用插件模型可以方便地扩展新的基础设施后端，通过配置可以指定运行时使用的具体适配器。Mixer当前内置了不少Adapter，常见的有Fluentd、Prometheus、Statsd、Redis Quota等。

2.属性

属性是Mixer中一个非常重要的基础概念，用于描述服务请求的特征，具体包含请求内容和请求环境相关的一些参数，比如请求大小、请求头、请求的来源IP地址、目标产生响应的时间戳、响应的状态码等。绝大多数属性数据由Envoy负责生成，但也有一些Mixer适配器会生成属性数据。

Mixer的属性有一个显著特点，属性是固定的，当前均由Mixer指定，不能随意设定。Mixer当前定义了50多个通用属性，具体可以查看Istio官方文档的属性词汇（Attribute Vocabulary）。

Mixer整体架构和功能逻辑都是围绕属性处理展开，Mixer中的每个Adapter会配置自身关注的属性，Mixer接收到Envoy发过来的请求时，会根据请求对应的属性列表，根据一定的匹配规则，调用匹配的Adapter。

Istio通过Mixer提供统一的抽象层，屏蔽基础设施后端的差异，提供强大的插件化扩展能力，基于Mixer可以非常方便地增加新的基础设施后端，同时由于Mixer仅提供机制，而不负责具体的实现策略，使得Mixer架构稳定后的变化比较少，可用性会比较高，通过Mixer提供的缓存机制，不仅可以减少调用后端基础设施的频率，减少延迟，同时也在后端基础设施无响应的时候也能继续工作，一定程度上屏蔽了后端基础设施的故障，保证了系统的整体稳定性。

5.4.2 Mixer通用配置模型

Mixer内置通用的配置模型支持简单的配置方式，提供了强大的扩展和控制能力。Mixer配置模型主要包含3种核心资源的配置，每个资源拥有全局唯一的完全限定名（fully qualified name），具体格式为{metadata.name}.{kind}.{metadata.namespace}，其他资源通过这个完全限定名对该资源进行引用。

1.处理器Handler

Adapter需要参数配置才能工作，处理器Handler就是用于对Adapter进行参数配置和描述，一个典型的Handler配置如下：

```
apiVersion: config.istio.io/v1alpha2
kind: listchecker
metadata:
  name: staticversion
  namespace: istio-system
spec:
  providerUrl: http://white_list_registry/
  blacklist: false
```

上述Handler的完全限定名为staticversion.listchecker.istio-system，每个Adapter有着相应的配置信息定义，具体定义在spec字段中，spec中的数据结构则依赖于对应Adapter的要求，具体可以从Istio官方文档中查看各个Adapter的配置格式。

2.实例Instance

实例Instance定义了属性到Adapter输入的映射关系，下面是一个metric实例的配置，用于生成requestduration指标，配置如下：

```
apiVersion: config.istio.io/v1alpha2
kind: metric
metadata:
  name: requestduration
  namespace: istio-system
spec:
  value: response.duration | "0ms"
  dimensions:
    destination_service: destination.service | "unknown"
    destination_version: destination.labels["version"] | "unknown"
    response_code: response.code | 200
  monitored_resource_type: "UNSPECIFIED"
```

上述Instance的完全限定名为requestduration.metric.istio-system，Handler中定义的所有维度均会在这里定义和属性的映射关系。

3.Rule

Rule（规则）定义了特定Instance调用某一Handler的时机，正是通过规则将输入的性质和Handler完成打通，一个典型的Rule配置如下：

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: promhttp
  namespace: istio-system
spec:
  match: destination.service == "service1.ns.svc.cluster.local" && request.headers["x-user"] == "user1"
  actions:
  - handler: handler.prometheus
    instances:
    - requestduration.metric.istio-system
```

规则中包含有一个`match`元素，用于前置检查，如果检查通过则会执行`action`列表。`action`中包含了一个实例列表，这个列表将会分发给`Handler`。规则必须使用`Handler`和实例的完全限定名。如果规则、`Handler`以及实例全都在同一个命名空间，命名空间后缀就可以在FQDN中省略，例如`handler.prometheus`。

5.4.3 Mixer架构演进以及对性能的影响

通过引入Mixer，解决了Istio策略控制和遥测统计方面的扩展性问题，通过Mixer可以在不影响Envoy核心架构的基础上，灵活方便地支持新的插件和策略，但Mixer的引入也带来了实实在在的性能问题，Mixer采用的通用属性匹配策略，并且几乎每次请求都需要和Mixer Server进行交互，都会带来一定的性能损耗。Mixer虽然在性能改进方面做了大量的工作，截至目前为止，仍然没有交出一份满意的答卷，这也导致不少对性能有很高要求的场景无法使用Mixer。

此外，Istio 1.1架构层面很大的一个变动是增加对远程Adapter的支持。在Istio1.1之前，所有Adapter的实现均在Mixer服务端内部完成，当某个Adapter需要更新时需要更新整个Mixer服务，同时单个Adapter的故障也会影响整个Mixer的可用性。根本原因是架构层面所有Adapter和Mixer核心实现耦合在一起，导致运维上不太灵活，为了解决这个问题，Istio 1.1架构上采取了更加激进的变革，支持Adapter以独立服务的形式存在于Mixer之外，Mixer通过gRPC的方式和进程外Adapter进行通信，通过将Adapter从Mixer中解耦，Mixer可以只保留最核心的逻辑，Adapter开发和部署上可以更加灵活，同时也保证了Mixer自身的稳定性。

远程Adapter带来架构灵活性的同时，导致性能问题雪上加霜。之前Mixer和Adapter代码与部署层面均在一起，属于进程内调用；远程Adapter时，Mixer通过gRPC的方式和远端Adapter服务进行交互，又增加了一层网络调用，进一步带来了性能损耗。

经过上述讨论，Mixer 1.1版本在性能问题上没有给出可行的改进性方案，并且由于支持远程Adapter，可能会导致性能问题更加严重。为了减少Mixer性能对Istio整体的影响，Istio 1.1支持用户对Mixer进行裁剪，通过一定的配置将Mixer特性关闭。

从Istio的早期版本开始，对Mixer性能和策略检查之间权衡的讨论一直没有停止过，Istio 1.1版本在这个问题上达成共识，绝大多数场景中，对性能的需求大于策略检查的需求，因此Istio 1.1版本默认关闭Mixer特性。

5.5 Istio安全

5.5.1 Istio安全基础

1.认证和授权

在展开讨论Istio安全之前，先看一下通信安全方面两个最基本的概念：**认证**（authentication）与**授权**。认证解决的是身份确认的问题，认证过程类似于身份证验证的过程；授权解决的是权限管理的问题，只有拥有相应的权限才能进行权限对应的操作。

以一个典型的知识论坛为例，使用论坛进行各种操作前都必须进行登录，登录就是一个身份确认的过程。用户在注册时，论坛会给每个用户分配一个唯一的标识，用于标识该用户的身份，用户后续在论坛上的所有操作均需要带着这个标识。登录完成后，不同的身份对应不同的权限，比如普通用户有发帖的权限，管理员用户有删帖的权限，因此登录操作是一个认证的过程，在发帖或删帖时确认是否有权限是一个授权的过程，其中认证是授权的基础，授权建立在身份标识的基础上进行。

2.身份

身份（identity）是通信安全的基础，安全认证和授权过程中均需要用到。Kubernetes、GCE和AWS等平台上均已经有自己的身份体系，Istio身份模型架构层面非常开放，直接复用这些业界主流云平台的账号体系，也支持用户自定义服务账户。

3.SPIFFE

SPIFFE（Secure Production Identity Framework For Everyone，通用安全身份框架）是一套用于服务之间进行身份识别的标准，主要包含以下内容。

- 1) SPIFFE ID标准：SPIFFE ID是服务的唯一标识，实现为URI。
- 2) SVID（SPIFFE Verifiable Identity Document）标准：将SPIFFE ID编码到一个加密的可验证文档中。
- 3) SVID API：颁发/撤销SVID的一套API标准。

SPIFFE SVID目前支持的实现方式是X.509数字证书，在X.509 SVID中，采用X.509数字证书的SAN（Subject Alternative Name）扩展字段来保存SPIFFE ID。

当前Istio已支持SPIFFE标准，可以为Istio网络中的服务颁发符合SPIFFE SVID标准的证书，基于这些证书进行身份认证、RBAC授权等相关工作。

4.Istio认证类型

Istio支持基于用户维度和基于服务维度两种认证方式，基于用户的认证是对请求的终端用户进行验证，具体可以通过JWT（JSON Web Token）、Google Auth和自定义身份认证等方式进行；基于服务维度的认证采用双向TLS的方式，为每个服务提供强大的身份认

证能力，通过用户和服务层面的认证，全面保证用户到服务、服务到服务之间通信的安全性。

5.5.2 Istio认证架构

Istio数据平面代理节点之间通过加密机制进行安全通信，为了保障安全通信的有效进行，通信双方需要通过认证机制来确认相互的身份信息，因此需要一套完善的认证架构来保证认证的可靠、高效运行，具体包括证书管理、证书验证、证书传递等，Istio认证整体架构如图5-3所示。

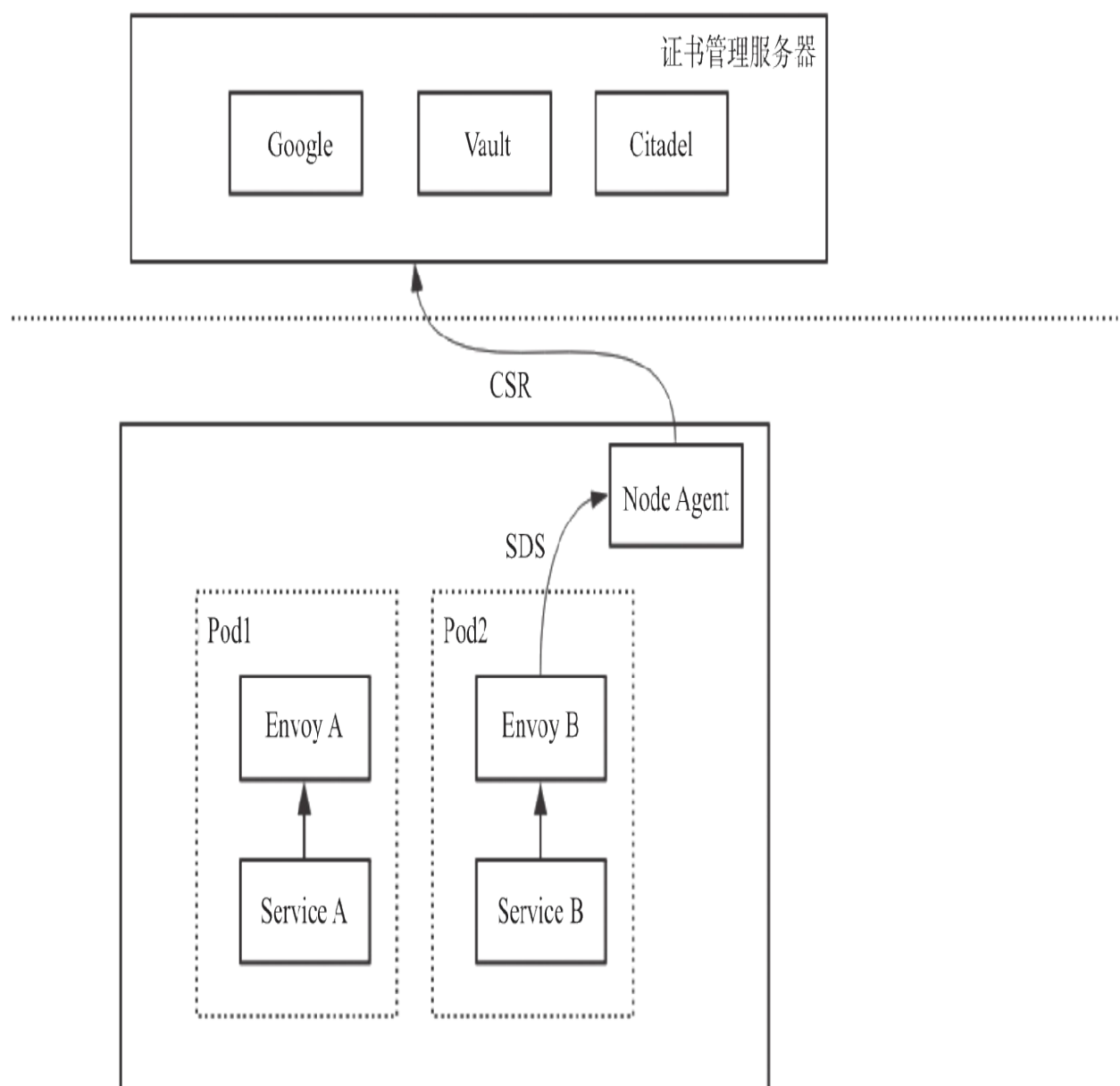


图5-3 Istio认证架构

如图5-3所示，Istio当前针对每个Kubernetes节点部署一个Node Agent代理服务，用于代理本节点上的Envoy和证书管理服务器之间的证书获取，定期进行证书和密钥的获取和

轮换，完整的证书获取流程大体如下。

- 1) Citadel创建一个gRPC服务来接受CSR请求。
- 2) Envoy通过Envoy secret发现服务（SDS）API发送证书和密钥请求。
- 3) Node Agent代理服务收到SDS请求后，节点代理会创建私钥和CSR，并将CSR及其凭据发送给Citadel进行签名。
- 4) Citadel验证CSR中携带的凭证，并签署CSR以生成证书。
- 5) 节点代理通过Envoy SDS API将从Citadel接收的证书和私钥发送给Envoy。

5.6 Istio配置处理框架

为了减少Istio内部各个组件配置处理方面的重复工作，提高配置管理方面的效率，Istio抽象出来一个独立的子项目Galley，专门负责配置管理方面的工作，定位是将其他组件从配置管理的复杂性中解放出来。

Istio 1.1版本之前，Galley仅负责配置验证工作，Istio 1.1版本正式发布后，Galley升级为Istio整个控制平面的配置中心，不仅负责配置资源的验证，还负责配置的获取、处理和分发。Galley整体架构如图5-4所示。

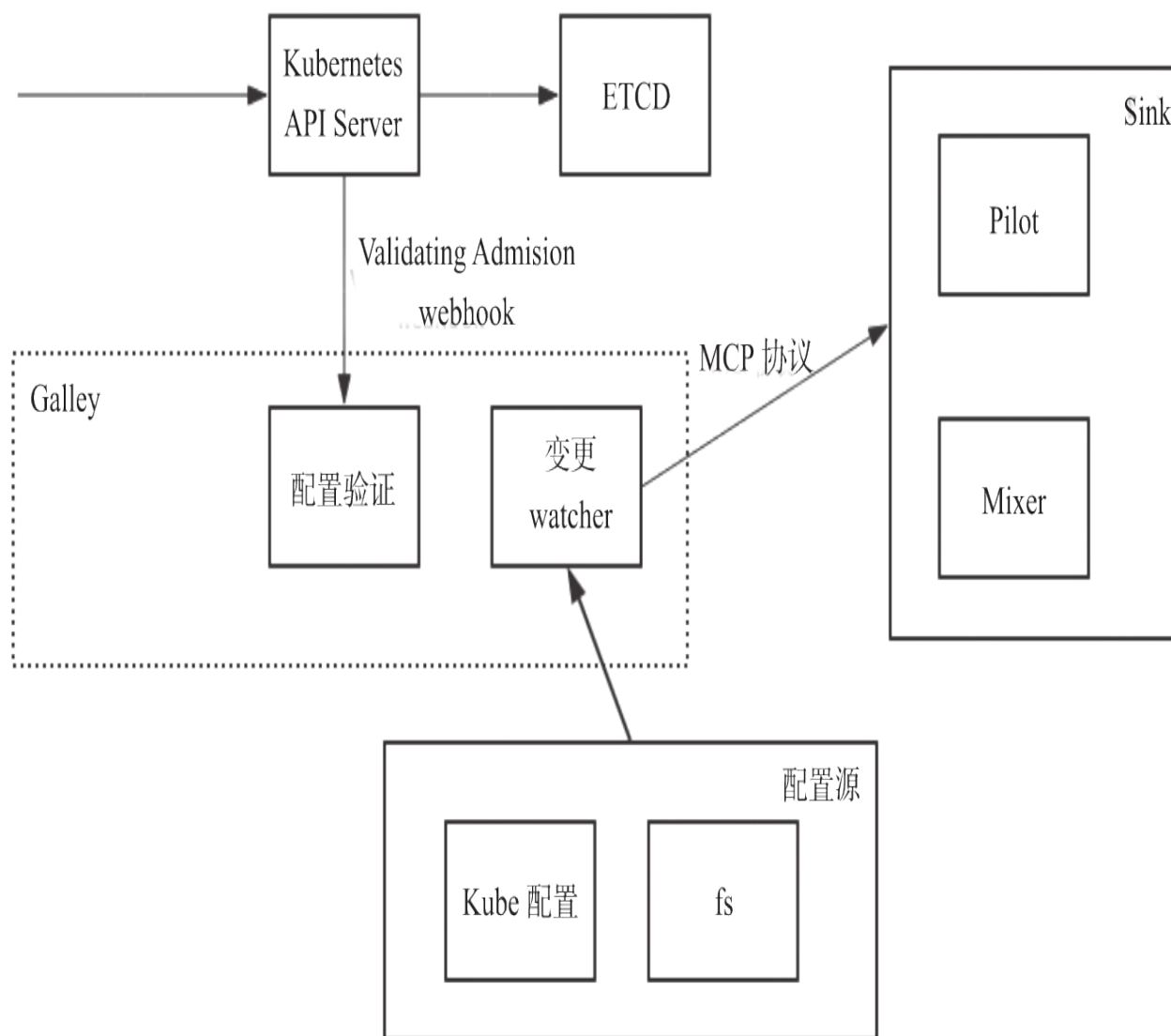


图5-4 Galley整体架构

从图5-4可以看出，Galley充当了一个代理的角色，对配置变更订阅者屏蔽不同平台、不同环境下配置源的差异和使用细节，通过统一的MCP协议和配置变更订阅方进行交互，订阅者只需要指定当前关系的配置源以及相应的配置，配置源的配置发生变更时，

Galley会接管配置处理的所有细节，将最新配置通过MCP协议同步给订阅方。

5.6.1 配置验证

Galley配置验证的目的是验证Istio核心配置的有效性。基于Kubernetes的Validating Admission Webhook机制，Galley向Kubernetes注册Validating Admission Webhook插件，具体包括关注的资源信息以及Galley的地址，Galley启动后会创建HTTP服务器，然后等待Kubernetes API Server发过来的配置验证请求。

Galley当前支持对Pilot资源和Mixer资源的验证，重点对配置资源的Scheme进行验证，当前核心的字段信息如下：

```
WebhookName
DeploymentAndServiceNamespace
DeploymentName
ServiceName
WebhookConfigFile
CertFile
KeyFile
CACertFile
Port
```

5.6.2 配置变更处理和分发

网格配置协议（Mesh Configuration Protocol, MCP）是一个基于订阅的配置分发协议，参考XDS协议的思想构建而成，用于Galley配置订阅方和配置提供方之间的配置交互。

MCP定义了一套完整的配置分发协议，具体包括配置提供方Source、配置订阅方Sink和配置资源本身，当配置提供方和订阅方建立订阅关系后，Galley负责检测配置资源变化，然后按照MCP协议的语义将变更后的配置发送给订阅方。

5.7 本章小结

本章主要介绍了Istio配置体系的全貌，通过本章的学习，可以对Istio的配置模型和交互协议有全面的理解。

第6章 Istio控制流设计

Istio提供的最核心能力是对业务流量进行透明转发和治理。其中，[数据平面聚焦流量转发、链路管控和治理](#)；[控制平面扮演控制流的角色](#)，为数据平面提供强大的管理和配置能力，为数据平面的简洁高效通信保驾护航。

本章接下来会详细剖析Istio控制流体系的架构设计。从功能上看，控制流主要提供两类能力：一类是生命周期管理，具体包含Envoy的自动注入、Envoy启动支持、Envoy运行状态监控以及Envoy异常处理等，目标是利用相应的基础设施和工具，提高Envoy运维的效率，保障Envoy的可用性；另一类是动态配置能力，通过相应机制将用户配置转换为Envoy需要的场景化配置。

6.1 Envoy生命周期管理

Service Mesh架构思想不是最近才产生，之前很长时间内一直没有引起足够的重视，一个很重要的原因是部署和运维的复杂度太高，如果没有相应的基础设施和工具体系支撑，很难保证Service Mesh运维的效率，也很难保障业务Service Mesh故障情况下的业务可用性。Istio在部署运维、生命周期管理方面充分利用Kubernetes基础设施的优点，有力地保障了Service Mesh的效率和可用性。

6.1.1 Envoy注入

Envoy注入是通过相应的手段将Envoy注入到Envoy运行环境中，Istio利用Kubernetes的特点，以优雅的方式实现Envoy注入，大大简化Envoy运维和管理上的开销。

目前，Envoy注入分为**手工注入**和**自动注入**两种方式。手工注入是指使用istioctl工具构建包含Sidecar镜像的模板，然后通过kubectl apply -f xxx.yaml将包含Sidecar镜像的模板文件注入到Pod中；自动注入基于Kubernetes的Admission Webhook机制，在Pod资源创建过程中自动触发。手工注入方式整体效率仍然偏低，而自动注入方式充分利用Kubernetes平台提供的机制和能力，实现了Sidecar透明接入，所以推荐使用自动注入的方式来完成Envoy的注入。下面重点介绍Envoy自动注入的机制和流程。

Istio的sidecar-injector服务负责自动注入的具体实现，sidecar-injector服务中首先创建一个用于Mutating Admission Webhook的Webhook资源实例，同时启动HTTP Server，接收Kubernetes发过来的自动注入请求：

```
// 创建Webhook资源
wh := &Webhook{
    server: &http.Server{
        Addr: fmt.Sprintf(":%v", p.Port),
    },
    sidecarConfig:      sidecarConfig,
    sidecarTemplateVersion: sidecarTemplateVersionHash(sidecarConfig.Template),
    meshConfig:         meshConfig,
    configFile:          p.ConfigFile,
    valuesFile:          p.ValuesFile,
    valuesConfig:        valuesConfig,
    meshFile:            p.MeshFile,
    watcher:             watcher,
    healthCheckInterval: p.HealthCheckInterval,
    healthCheckFile:     p.HealthCheckFile,
    certFile:            p.CertFile,
    keyFile:             p.KeyFile,
    cert:               &pair,
}

// 创建HTTP Server
h := http.NewServeMux()
h.HandleFunc("/inject", wh.serveInject)
wh.server.Handler = h
```

sidecar-injector服务启动后并不会自动触发注入机制，需要通过Mutating Admission Webhook机制创建webhook注册配置，通过上述启动的sidecar-injector服务的/inject接口，监听符合条件的Pod资源。Webhook注册完成后，当监听有符合条件的Pod资源创建时，就会调用/inject接口进行自动注入的实际处理。

下面的inject函数负责Envoy自动注入的具体处理，先判断当前Pod是否满足注入要求，如果满足，根据sidecar-injector的模板配置进行Envoy注入数据的组织和生成，然后打包到Pod中，返回给Kubernetes。

```
func (wh *Webhook) inject(ar *v1beta1.AdmissionReview) *v1beta1.AdmissionResponse {
    ...
    // 请求参数检查，注入条件判断
    if !injectRequired(ignoredNamespaces, wh.sidecarConfig, &pod.Spec, &pod.ObjectMeta) {
        return &v1beta1.AdmissionResponse{
            ...
        }
    }
    ...
}
```

```

        Allowed: true,
    }
}
// Envoy注入数据生成
spec, iStatus, err := InjectionData(wh.sidecarConfig.Template, wh.valuesConfig, wh.sidecarTemplat
    return toAdmissionResponse(err)
}
// 将Envoy注入数据打包到Pod中
patchBytes, err := createPatch(&pod, injectionStatus(&pod), annotations, spec)
if err != nil {
    return toAdmissionResponse(err)
}
// HTTP响应消息组装
reviewResponse := v1beta1.AdmissionResponse{
    Allowed: true,
    Patch:   patchBytes,
    PatchType: func() *v1beta1.PatchType {
        pt := v1beta1.PatchTypeJSONPatch
        return &pt
    }(),
}
return &reviewResponse
}
}

```

Envoy注入信息中包括Envoy服务启动相关的一切信息，Kubernetes后续在启动Pod时，会根据上述的注入信息同时启动Envoy服务。接下来会详细讨论Envoy服务启动的具体逻辑。

6.1.2 Envoy启动管理

Pilot-agent负责Envoy的启动和运行状态监控，Pilot-agent会根据Envoy的类型，拼装相应的启动配置文件和启动参数，进行Envoy的启动或重启；同时通过探测和监控Envoy当前的运行状态，或者接收外界的配置变更等控制指令，对非正常状态的Envoy进行重启或调整。

1.Node ID

Node ID是Envoy的唯一标识，由Istio控制平面负责生成，由节点类型、IP地址、节点ID和DNS域名4部分组成，以"~"分隔，具体如图6-1所示。

类型	~	IP 地址	~	节点 ID	~	DNS 域名

图6-1 Envoy节点组成

节点类型是Envoy具体的运行模式，当前支持sidecar、ingress和router 3种运行模式，Sidecar模式和业务微服务捆绑部署，用于Istio集群内部微服务之间流量的透明代理；Ingress用于Istio集群对外的流量代理，负责将集群内部服务暴露出去，供集群外部访问；Router是独立的L4/L7层代理。

IP地址、节点ID信息和DNS域名具体和当前支持的服务注册中心相关，可以通过Pilot-agent的命令行参数的对应flag指定，如果没有具体指定，Pilot-agent会根据当前连接的服务注册中心采用相应的配置方式。以Kubernetes为例，Kubernetes节点ID就是Pod ID加上当前的命名空间，DNS域名就是DNS域名后缀，具体实例如表6-1所示。

表6-1 Envoy节点ID具体实例

类型	IP 地址	节点 ID	DNS 域名
Sidecar	环境变量 INSTANCE_IP	环境变量 POD_NAME. 环境变量 POD_NAMESPACE	环境变量 POD_NAMESPACE. svc.cluster.local

2.Meta元数据

Envoy元数据负责Envoy自定义属性信息的维护，主要用于数据平面和控制平面之间的XDS交互。元数据信息由控制平面Pilot-agent生成，作为Bootstrap启动信息的一部分，供Envoy服务启动时使用。Envoy启动后和控制平面交互获取当前节点的XDS信息时，会把本节点的Envoy元数据信息透传给控制平面，辅助控制平面生成相应的XDS配置信息。总之，控制平面生成元数据信息用于Envoy启动，Envoy启动后透传元数据信息给控制平面用于XDS配置的生成决策。

Envoy元数据以Key-Value对的形式存在，当前已经支持如ISTIO_PROXY_VERSION、CONFIG_NAMESPACE等超过20个元数据，具体定义在/pilot/pkg/model/context.go。

3.Epoch

Epoch是Envoy进程的启动序号，从0开始编号，第一个Envoy进程对应的Epoch为0，此后Envoy每重启一次，Epoch递增1。引入Epoch是为了管理Envoy的热重启。

4.命令行参数

Pilot-agent根据Envoy配置模板和当前的启动信息，生成相应的Envoy启动文件，用于接下来的Envoy启动过程。

5.配置文件生成

Pilot-agent根据Envoy配置模板和当前的启动信息，生成相应的Envoy启动文件，用于接下来的Envoy启动过程。Envoy配置模板用于定义Envoy Bootstrap启动配置的格式，istio/tools/deb/envoy_bootstrap_v2.json就是Envoy配置模板的一个典型例子。

配置模板定义了一系列需要填充的配置字段，WriteBootstrap函数根据当前启动信息，在配置模板的基础生成相应的配置文件。

```
WriteBootstrap(config *meshconfig.ProxyConfig, node string, epoch int, pilotSAN []string, opts map[string]
    if err := os.MkdirAll(config.ConfigPath, 0700); err != nil {
        return "", err
    }
    //生成配置文件envoy-rev%d.json
    fname := configFile(config.ConfigPath, epoch)
    cfg := config.CustomConfigFile
    if cfg == "" {
        cfg = config.ProxyBootstrapTemplatePath
    }
    cfgTpl, err := ioutil.ReadFile(cfg)
    //获取配置模板
    t, err := template.New("bootstrap").Parse(string(cfgTpl))
    ...
    //拼装配置模板参数
    opts["config"] = config
    ...
    fout, err := os.Create(fname)
    //将配置参数按照模板格式，生成配置文件
    err = t.Execute(fout, opts)
    return fname, err
}
```

6.Envoy的具体启动

Envoy启动后，会记录当前Envoy的配置信息，后续启动相关操作前会先比较新配置是否和当前配置相同，如果配置相同不进行任何处理，否则会继续启动流程。

Envoy启动相关准备工作完成后，即可通过Exec命令启动Envoy进程，启动时将新Envoy进程的标准输出stdout和标准错误stderr进行重定向。启动完成后，通过监控Envoy进程的退出码，对Envoy的运行状态进行监控。

6.1.3 Envoy配置和运行状态监控

1.配置文件监控

Pilot-agent通过watch对象负责配置文件的监控，比如证书文件发生变化时，watch对象感知到变化事件后，通知Pilot-agent进行处理。

配置文件变更感知和处理是通过单独启动一个Go协程的方式进行处理，具体处理函数是watchCerts。watchCerts通过fsnotify的机制^[1]来进行变更感知，感知到配置文件变化时，会调用watchFileEvents进行处理。

```
func watchFileEvents(ctx context.Context, wch <-chan *fsnotify.FileEvent, minDelay time.Duration, notifyFn func() error) {
    for {
        select {
            case ev := <-wch:
                if timer != nil {
                    continue
                }
                //检测到配置变更后，没有立即处理，而是延迟一段时间，
                //延迟的目的是防止处理时配置变更可能还未完成，这样处理时就会产生各种异常情况
                timer = time.NewTimer(minDelay)
                timeChan = timer.C
            case <-timeChan:
                timeChan = nil
                timer.Stop()
                timer = nil
                notifyFn()
            case <-ctx.Done():
                return
        }
    }
}
```

watchFileEvents处理时有个小技巧，配置变更时没有立即进行处理，而是创建一个定时器，定时一段时间后再进行处理，这个是配置热加载场景经常会使用的处理方式。通过延迟一段时间，保证配置变更后的配置文件的完整性，防止配置变更立即处理导致的各种异常问题。

定时触发时会调用notifyFn函数进行处理，这里的notifyFn实际上是SendConfig，SendConfig计算证书文件的Hash值，通过updates Chan通道通知Pilot-agent进行处理。

```
(w *watcher) SendConfig() {
    h := sha256.New()
    generateCertHash(h, w.certs)
    w.updates <- h.Sum(nil)
}
```

2.运行状态监控

Pilot-agent对Envoy运行状态监控和控制有两个维度。

一是，通过监控Envoy的退出码，当Envoy进程退出时，会通过cmd.Wait捕获退出码，然后写到done通道中，Run函数会将退出码返回给Pilot-agent。

```
func (e *envoy) Run(config interface{}, epoch int, abort <-chan error) error {
    ...
    done := make(chan error, 1)
    go func() {
        done <- cmd.Wait()
    }()
    select {
    case err := <-abort:
        cmd.Process.Kill();
        return err
    case err := <-done:
        return err
    }
}
```

二是，当Pilot-agent判断当前Envoy需要退出时，会将退出消息写入abort通道，Envoy进程接收到abort退出消息后，会调用cmd.Process.Kill将当前Envoy进程杀掉。

3.运行状态查看

为了方便外部用户直接查看Envoy的实时运行状态，Pilot-agent启动一个HTTP服务器，外界可以直接通过HTTP的URL接口查看Envoy的运行状态。

Pilot-agent的管理服务器的具体实现在pilot/cmd/pilot-agent/status下，这里会启动一个HTTP服务器，当前支持如下URL查询方式，比如通过“/app-health”可以查看Envoy的健康状态，通过“/healthz/ready”查看Envoy的就绪状态等。

```
mux.HandleFunc("/healthz/ready", s.handleReadyProbe)
mux.HandleFunc("/", s.handleAppProbe)
mux.HandleFunc("/app-health", s.handleAppProbe)
```

这里的具体实现很简单，以handleAppProbe为例，就是通过调用URL的HTTP管理接口，获取Envoy当前具体的运行状态。

4.Envoy重启策略

Pilot-agent会根据Envoy进程的退出状态决定具体的重启策略，Envoy进程的退出状态大体可以分为如下3种情况。

- 1) **退出状态码为空**：说明是Envoy正常退出的，这里只需要记录日志即可。
- 2) **退出状态码为errAbort**：当Envoy进程退出状态为errAbort时，说明当前退出是由Pilot-agent主动触发，Pilot-agent在配置变更等场景时会发送退出控制指令给当前Envoy，然后启动新的Envoy进程，因此这里收到errAbort时只需要记录相应的日志即可。
- 3) **退出状态码非空，但不是errAbort**：该状态说明Envoy热重启过程中出错，这时需要通过errAbort消息通知当前所有Envoy进程进行恢复。

针对Envoy退出或重启出错的场景，Pilot-agent当前采用尽力而为的方式，如果失败会继续进行重试处理，如果连续重试10次均没有成功，说明当前节点可能出问题了，这时Pilot-agent进程会主动退出，由外界进行干预和处理，比如通过Kubernetes调度到其他节点上继续运行。

[1] 如果对fsnotify的具体细节感兴趣，可以找参考资料进行进一步学习，这里不再详述。

6.2 Istio配置变更管理

Istio通过通用配置和服务模型，灵活方便地支持不同平台与环境下的配置、服务变更和管理，下面首先介绍Istio配置管理的通用机制，然后以Kubernetes为例，重点介绍Kubernetes平台下配置管理和服务管理的具体实现。

6.2.1 通用模型和机制

1.对象配置模型

Istio针对对象配置制定通用的Schema描述，具体通过下面的ProtoSchema结构来表示，针对每一个配置对象，Istio均会定义相应的Schema描述，IstioConfigTypes中包括Istio当前支持的核心对象。

Istio根据IstioConfigTypes向配置中心注册关心的对象列表，当对象信息发生变化时，配置中心将最新配置推送给Istio。

```
type ProtoSchema struct {
    //对象范围，当ClusterScoped为true时表示当前对象是集群范围
    ClusterScoped bool
    //scheme对象名
    SchemaObjectName string
    //类型
    Type string
    // Plural is the type in plural.
    Plural string
    //对象所在的组
    Group string
    //对象版本
    Version string
    //对象名，可以通过该名称创建protobuf格式的对象实体
    MessageName string
    //配置验证函数
    Validate func(name, namespace string, config proto.Message) error
    //MCP集合
    Collection string
}
```

ProtoSchema规定了每个对象类型的Schema，关键词是“类型的Schema描述”，通过Schema即可实现某一种对象的描述，针对每一个具体的对象实体，Istio通过如下Config对象来进行描述：

```
type Config struct {
    //对象元数据信息
    ConfigMeta
    //对象实际信息，以protobuf消息的方式存放
    Spec proto.Message
}
```

Config对象包括对象元数据和对象实体两部分，Istio针对所有对象制定了统一的元数据描述格式ConfigMeta，具体包含如下信息。

- 1) Type: 对象的配置类型，以Kubernetes为例，对应的是Kubernetes资源对象的类型，比如virtual service。
- 2) Version: 对象的API版本信息。
- 3) Name: 对象资源名，也是对象的唯一标识。

4) **Namespace**: 对象所属的命名空间, Envoy节点也会设置相应的命名空间, 获取XDS配置时会选取和当前节点处于相同命名空间的对象。

5) **Label**: 对象的标签信息, 和Kubernetes的Label含义相同, 用于在对象选取时根据标签进行匹配。

对象实体信息由Config对象的Spec字段来表示, 这是个Protobuf格式的消息, 各配置中心需要提供相应的转换机制, 将自身的数据格式转换成Istio内部的通用配置模型。

除了配置对象描述外, 还需要有统一的对象管理机制, Istio通过下面的ConfigStoreCache接口对Istio对象进行管理, ConfigStoreCache提供了一组抽象接口, 通过这些抽象接口完成对象的增删改查、存储和变更通知。可以基于ConfigStoreCache进行扩展实现, 定义自己的插件实现。

```
type ConfigStoreCache interface {
    ConfigStore
    RegisterEventHandler(typ string, handler func(Config, Event))
    Run(stop <-chan struct{})
    HasSynced() bool
}
type ConfigStore interface {
    ConfigDescriptor() ConfigDescriptor
    Get(typ, name, namespace string) *Config
    List(typ, namespace string) ([]Config, error)
    Create(config Config) (revision string, err error)
    Update(config Config) (newRevision string, err error)
    Delete(typ, name, namespace string) error
}
```

2.服务配置模型

服务通用模型定义了什么是Istio的服务、如何发现Istio服务以及服务变更时的通知机制, 各个平台、各个服务注册中心均可以基于服务通用模型定义自己的插件扩展。

Istio Service对象用于描述一个Istio服务, 一个Istio服务主要包含如下信息。

1) **Hostname**: 服务名, 也是服务的唯一标识。

2) **Address**: 服务对应负载均衡器的IPV4地址, 通过负载均衡器来完成服务访问流量的分发。

3) **ClusterVIPs**: 服务的虚拟IP地址。

4) **PortList**: 服务的端口信息列表, 服务的端口可能不止一个, 同时需要强调的是服务的端口和服务实例的端口信息不一定一致。

5) **Resolution**: 服务实例获取方式, 比如通过DNS方式或者直通模式。

6) **ServiceAttributes**: 服务属性信息, 一般用于Mixer或者RBAC策略使用。

此外, Istio通过ServiceDiscovery定义了服务注册中心的抽象接口, 服务注册中心通过实现这组接口进行服务发现扩展支持。与对象通用配置类似, 服务注册中心通过

Controller接口提供服务变更机制，关注服务变更的模块通过注册变更通知事件就可以对服务和节点变更事件进行相应处理。

```
type ServiceDiscovery interface {
    Services() ([]*Service, error)
    GetService(hostname Hostname) (*Service, error)
    InstancesByPort(hostname Hostname, servicePort int, labels LabelsCollection) ([]*ServiceInstance, error)
    GetProxyServiceInstances(*Proxy) ([]*ServiceInstance, error)
    GetProxyLocality(*Proxy) string
    ManagementPorts(addr string) PortList
    WorkloadHealthCheckInfo(addr string) ProbeList
}
type Controller interface {
    //服务信息变更回调函数
    AppendServiceHandler(f func(*Service, Event)) error
    //节点信息变更回调函数
    AppendInstanceHandler(f func(*ServiceInstance, Event)) error
    //控制器启动入口
    Run(stop <-chan struct{})
}
```

3.配置中心

配置中心负责实现Istio对象配置模型，对Istio对象配置进行管理，Istio当前内置支持文件存储、Kubernetes CRD以及MCP这几种配置中心方式，当然也可以支持自定义的配置中心。

文件存储方式不灵活，管理上非常不方便，Istio当前主要使用Kubernetes CRD的方式对配置信息进行存储。为了减少各个组件对配置的重复开发工作，Istio MCP机制专门抽象出一个组件Galley对配置进行统一管理，Istio配置中心后续会逐渐迁移到基于MCP的Galley上。

4.服务注册中心

Istio服务注册中心负责Istio集群内部服务，以及Istio集群使用的集群外部服务信息的管理，服务注册中心负责管理服务发现信息，具体主要包括服务信息、节点信息、端口信息等。服务发现信息发生变更时通过统一的抽象接口进行变更通知。下面分别从Istio集群内部服务和集群外部服务这两方面展开介绍。

在Istio集群内部服务的注册和管理上，Istio原生支持Kubernetes、Consul和Memory 3种服务注册中心方式，当前只有Kubernetes达到生产环境下可用，其他成熟度偏低，不推荐生产环境下使用。

Istio通过External注册中心对Istio集群外部服务进行管理。External注册中心通过kubernetes CRD资源ServiceEntry定义了Istio集群引入外部服务的方式，其本身并不负责外部服务的具体实现。Istio使用方通过相应的机制将外部服务以ServiceEntry的形式进行存储和变更管理，External注册中心只需要关注ServiceEntry CRD资源的变更事件，有变更时触发Envoy的配置变更。

由于Istio同时存在集群内部服务注册中心和针对集群外部服务的External注册中心，为了方便对这些注册中心进行管理，Istio引入了聚合服务注册中心aggregate的分层概念，其他注册中心向aggregate注册，aggregate自身不负任何逻辑，只负责调用自身管理的注册中心服务。Istio服务注册中心的整体架构如图6-2所示。

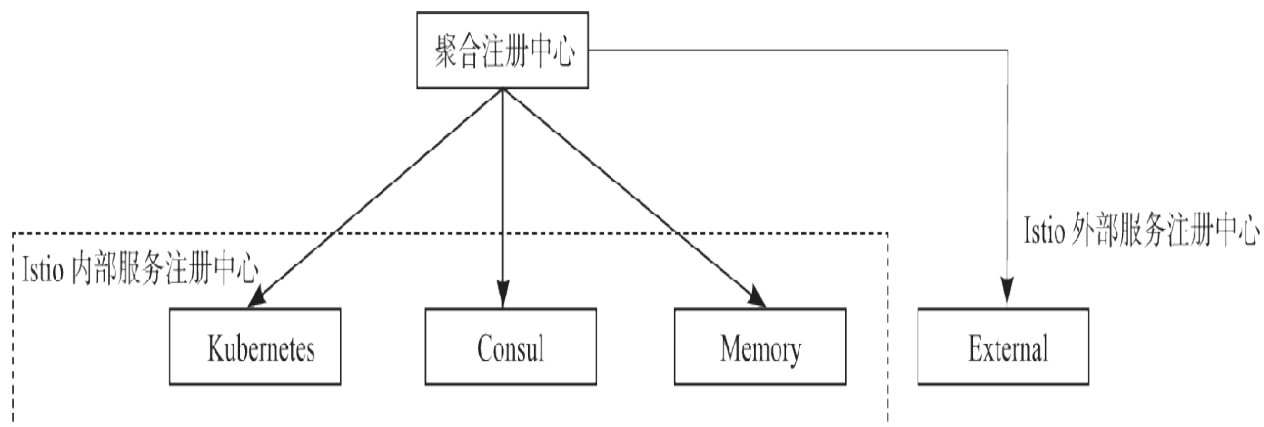


图6-2 服务注册中心架构图

聚合注册中心对Istio外部服务注册中心External和Istio内部服务注册中心Kubernetes、Consul进行统一管理，初始化过程中，会根据配置创建具体的注册中心实例，同时会创建缺省的External注册中心，聚合注册中心通过统一的接口层对它们进行统一管理。

6.2.2 Kubernetes具体实现

1.Kubernetes配置中心

kubernetes CRD方式的初始化在makeKubeConfigController函数中实现，首先会读取配置文件，根据配置文件调用crd.NewClient创建kubernetes CRD Client，创建过程中会完成kubernetes CRD资源的注册，当前支持的kubernetes CRD资源定义在model.IstioConfigTypes中，包括VirtualService、Gateway等。

```
func (s *Server) makeKubeConfigController(args *PilotArgs) (model.ConfigStoreCache, error) {
    kubeCfgFile := s.getKubeCfgFile(args)
    configClient, err := controller.NewClient(kubeCfgFile, "", model.IstioConfigTypes, args.Config.Contro
    ...
    return controller.NewController(configClient, args.Config.ControllerOptions), nil
}
```

Kubernetes CRD资源注册完成后将会调用crd.NewController，构建Kubernetes CRD资源对象变更事件的处理框架，处理框架的思路很简单，框架首先向Kubernetes注册关注的资源类型，资源变更时Kubernetes会回调框架，同时框架提供注册函数让关心资源变更的逻辑注册回调函数。注册信息包括关注的资源类型和变更通知回调函数，通过这种注册机制，每种资源类型都会形成一个变更通知链，资源变更触发时，框架会遍历变更资料对应的通知链，调用事先注册好的回调函数进行通知处理。具体实现时由SharedIndexInformer、queue和Kube Client配合完成，crd.NewController会调用addInformer向Kubernetes注册资源变更回调函数，同时提供RegisterEventHandler函数用于其他模型向crd.NewController注册回调函数。

2.Kubernetes服务注册中心

Kubernetes注册中心负责Kubernetes平台服务和节点信息的获取与变更处理，通过Kubernetes Informer机制关注Kubernetes核心资源的变化，当核心资源发生变化时通知Istio进行配置变更处理。

Kubernetes注册中心当前关注Pod、Node、Service、Endpoint 4种核心资源，当Service和Endpoint资源变更时，直接触发服务和节点信息的变更；当Pod资源变更时，比如标签变化可能导致根据标签进行资源匹配发生变化，应该触发XDS配置变更。

6.3 控制平面和数据平面的XDS交互

Istio的控制平面和数据平面通过XDS协议进行配置资源的请求与下发，通信层面对XDS的支持，有V1版本的HTTP+JSON方式和V2版本的gRPC+Protobuf方式，由于V1版本存在资源消耗过多、不支持双向通信等问题，社区后续不再继续支持V1版本（当前存在是为了兼容之前的存量用户，新用户不建议使用V1版本，后续会逐渐废弃掉），所以接下来基于V2版本对控制平面和数据平面的XDS交互展开讨论。

Istio控制平面服务会专门启动一个gRPC Server，用于和Envoy之间的XDS数据交互，Envoy启动时会通过调用这个接口和Pilot-discovery建立连接。建立连接后，Pilot-discovery会将该Envoy节点相关的全量配置信息通过ADS协议返回给Envoy，同时Pilot-discovery会管理与Envoy交互的每个连接。如果没有任何配置变更，则Pilot-discovery和Envoy之间不会有任何的通信交互；Pilot-discovery会监控配置变更事件，如果配置发生变更，则Pilot-discovery会轮询和Envoy的每个连接，将变更后的最新配置返回给Envoy。控制平面和数据平面的XDS交互图示如图6-3所示。

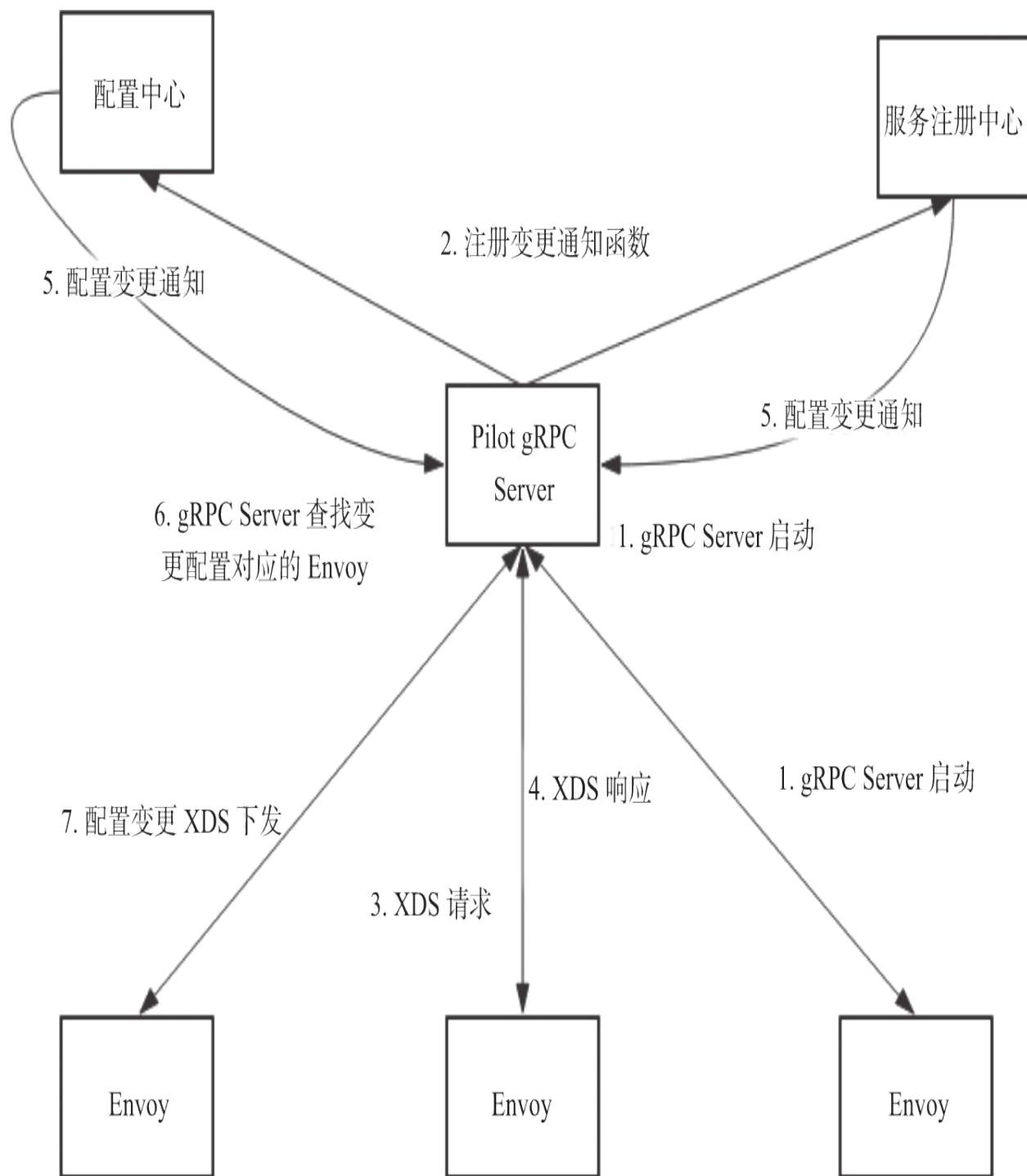


图6-3 控制平面和数据平面XDS交互图

gRPC Server的启动、Envoy的XDS请求、XDS消息下发和Envoy的XDS响应处理一起构成了XDS消息处理的完整闭环，下面逐一展开讨论。

6.3.1 控制平面的gRPC Server启动

控制平面Pilot服务启动时，启动专门的gRPC Server，用于管理和数据平面之间的XDS消息交互。数据平面的新连接到来时，会通过gRPC接口处理函数StreamAggregated-Resources进行新连接处理。

为了对配置中心和服务注册中心的变更进行感知与处理，gRPC Server启动后就注册相应的回调函数，用于处理变更后的XDS，注册逻辑如下：

```
//注册服务变更通知函数
serviceHandler := func(*model.Service, model.Event) { out.clearCache() }
if err := ctl.AppendServiceHandler(serviceHandler); err != nil {
    return nil
}
//注册服务节点变更通知函数
instanceHandler := func(*model.ServiceInstance, model.Event) { out.clearCache() }
if err := ctl.AppendInstanceHandler(instanceHandler); err != nil {
    return nil
}
if configCache != nil {
    //注册配置变更通知函数
    configHandler := func(model.Config, model.Event) { out.clearCache() }
    for _, descriptor := range model.IstioConfigTypes {
        configCache.RegisterEventHandler(descriptor.Type, configHandler)
    }
}
```

6.3.2 Envoy的XDS请求

对于V2版本gRPC协议来说，ADS的触发由GrpcMuxImpl负责管理。GrpcMuxImpl的start函数会调用establishNewStream创建gRPC连接，并发起XDS请求流程。通过GrpcMuxImpl: : onReceiveMessage进行XDS响应消息的解析和处理，onReceiveMessage主要通过watch->callbacks_.onConfigUpdate触发各个具体XDS资源的配置更新，各个XDS资源初始化时会通过SubscriptionFactory: : subscriptionFromConfigSource注册对ADS配置变更的关注。

6.3.3 Istio XDS配置下发

先看新连接场景下的XDS配置下发逻辑，StreamAggregatedResources函数负责新连接的处理，定义在pilot/pkg/proxy/envoy/v2/ads.go文件中。StreamAggregatedResources函数首先调用newXdsConnection创建连接管理结构，然后启动一个新的协程单独负责新连接消息的解析，同时主流程进入等待逻辑，等待新的消息（主要是新请求消息和配置更新消息）。这个新的协程不是长驻协程，处理完请求后就会立即退出。receiveThread函数负责新协程的消息处理，主要是判断消息是否有问题，如果没有问题，将读到的请求消息放到reqChannel通道里面。

StreamAggregatedResources函数的for循环是无限循环流程，这里会监控两个Channel通道的消息，一个是reqChannel的新连接消息，一个是pushChannel的配置变更消息。

```
func (s *DiscoveryServer) StreamAggregatedResources(stream ads.AggregatedDiscoveryService_StreamAggregate
// 新连接到来，初始化XDS响应消息推送对应的上下文
err := s.globalPushContext().InitContext(s.Env)
...
// 创建XDS连接并保存，用于后续配置变更时的XDS响应消息下发
con := newXdsConnection(peerAddr, stream)
// XDS请求消息接收，接收后存放到reqChannel中
reqChannel := make(chan *xdsapi.DiscoveryRequest, 1)
go receiveThread(con, reqChannel, &receiveError)
for {
    select {
    case discReq, ok := <-reqChannel:
        err = s.initConnectionNode(discReq, con)
        ...
        switch discReq.TypeUrl {
        case ClusterType:
            ...
            err := s.pushCds(con, s.globalPushContext(), versionInfo())
        case ListenerType:
            ...
            err := s.pushLds(con, s.globalPushContext(), versionInfo())
        case RouteType:
            ...
            err := s.pushRoute(con, s.globalPushContext(), versionInfo())
        case EndpointType:
            ...
            err := s.pushEds(s.globalPushContext(), con, versionInfo(), nil)
        }
    }
    if !con.added {
        con.added = true
        s.addCon(con.ConID, con)
        defer s.removeCon(con.ConID, con)
    }
    case pushEv := <-con.pushChannel:
        //配置变更时的XDS消息下发
        err := s.pushConnection(con, pushEv)
        ...
    }
}
```

来看reqChannel消息的处理，reqChannel收到新数据时，表明新连接对应的数据已经读取完成，这里会从reqChannel中取出XDS请求消息，然后根据不同类型的XDS请求，调用相应的XDS下发逻辑。比如，TypeUrl为ClusterType时，调用pushCds进行集群信息的推送，TypeUrl为ClusterType或ListenerType时，调用pushLds完成监听器信息的推送。

在V2版本的XDS协议实现中，为了保证多个XDS数据下发的顺序，LDS、RDS、CDS和EDS等所有XDS交互均在一个gRPC连接上完成，因此StreamAggregatedResources接收到第一个请求时会调用s.addCon将连接保存起来，供后续配置变更时使用。

下面再看一下配置更新的时机，之前NewDiscoveryServer函数已经分别注册了针对Kubernetes CRD资源和注册中心的变更事件通知回调函数，当这两者有变更时，会调用注册的回调函数，这几类资源变更的回调处理都是调用out.clearCache函数，当配置中心和注册中心发生变更时，通过之前注册的回调函数clearCache进行变更处理，clearCache的实现很简单，就是将变更消息写到updateChannel通道里。

```
func (s *DiscoveryServer) clearCache() {
    s.ConfigUpdate(true)
}
func (s *DiscoveryServer) ConfigUpdate(full bool) {
    s.updateChannel <- &updateReq{full: full}
}
```

handleUpdates函数负责变更事件的具体处理，Istio收到变更事件并没有立即进行处理，而是创建一个定时器事件，通过定时器事件延迟一段时间。这样做的初衷：一方面可以减少配置变更下发的频率，进而减少控制平面和Envoy交互的通信开销；另一方面，延迟对配置变更消息的处理，可以保证配置变更下发时变更的完整性，如果变更时立即处理，可能会遇到变更不完整的情况，导致一些不可控的结果。配置变更时的延迟生效思路，在很多开源系统都被采用，也算是一个通用的架构思路。

```
func (s *DiscoveryServer) handleUpdates(stopCh <-chan struct{}) {
    fullPush := false
    for {
        select {
        case r := <-s.updateChannel:
            lastConfigUpdateTime = time.Now()
            if debouncedEvents == 0 {
                timeChan = time.After(DebounceAfter)
                startDebounce = lastConfigUpdateTime
            }
            debouncedEvents++
            if r.full {
                fullPush = true
            }
        case now := <-timeChan:
            timeChan = nil
            eventDelay := now.Sub(startDebounce)
            quietTime := now.Sub(lastConfigUpdateTime)
            // 判断当前是否触发push下发
            if eventDelay >= DebounceMax || quietTime >= DebounceAfter {
                pushCounter++
                go s.doPush(fullPush)
                fullPush = false
                debouncedEvents = 0
                continue
            }
            timeChan = time.After(DebounceAfter - quietTime)
        case <-stopCh:
            return
        }
    }
}
```

按时上述配置变更延迟触发策略，当满足触发逻辑时就会创建一个新的协程，调用AdsPushAll进行XDS配置下发的实际处理。AdsPushAll中针对EDS增量更新的场景，调用

edsIncremental进行EDS增量处理，否则通过startPush进行全量更新XDS下发，startPush函数会遍历当前的Envoy连接，将配置更新消息写到每个连接对应的pushChannel通道。

pushChannel通道消息的处理由StreamAggregatedResources完成。该函数通过无限循环关注eqChannel和pushChannel两个通道的消息。收到pushChannel消息时，直接调用pushAll函数完成配置下发，这里会依次调用pushCds、pushEds、pushLds、pushRoute分别进行集群信息、服务发现信息、监听器信息和路由信息的下发，从机制上解决了由于V1版本各XDS资源下发顺序有问题导致的配置变更瞬间请求处理出错问题。

6.3.4 Envoy的XDS消息接收

对于v2版本gRPC协议来说，ADS的触发由GrpcMuxImpl负责管理，GrpcMuxImpl的start函数中调用establishNewStream创建gRPC连接，并发起XDS请求流程。通过GrpcMuxImpl: : onReceiveMessage进行XDS响应消息的解析和处理，onReceiveMessage中重点是通过watch->callbacks_.onConfigUpdate触发各个具体XDS资源的配置更新，各个XDS资源初始化时会通过SubscriptionFactory: : subscriptionFromConfigSource注册对ADS配置变更的关注。

6.4 XDS配置生成

6.4.1 可见性

在Istio 1.1版本之前，在服务发现和流量路由方面，每个Envoy的行为是完全相同的，都需要接收整个Istio集群的服务和路由信息，同时要感知整个Istio集群的XDS信息变更，比如Istio集群有2000个微服务，Envoy节点A关联的服务节点的出入口流量，涉及的服务只有两个，面对这种场景，Istio 1.1版本之前的XDS交互有如下两个突出问题。

（1）不必要的配置数据下发

每次XDS配置变更时，对整个Istio集群的XDS配置进行全量下发，Envoy节点A关注的服务信息只有两个，但每次下发2000个服务的XDS配置信息。海量数据下发，不仅会影响Pilot和Envoy之间网络传输的性能，同时XDS配置的编解码与存储也会消耗Envoy节点大量的计算和存储资源。

（2）不必要的配置下发频率

虽然Envoy节点A关注的服务信息只有两个，但当2000个服务的信息以及路由变化时，均需要向所有的Envoy节点下发全量的XDS配置信息，即使这些服务的信息和变更Envoy节点A根本不需要关注，这样就会产生大量不必要的XDS配置变更交互。

为了解决这个问题，Istio 1.1版本引入了Envoy可见性的概念，具体通过Sidecar资源和ExportTo属性来定义，通过Sidecar资源可以自定义设置一类Sidecar的行为，用于对数据平面代理进行精细化和场景化配置，通过ExportTo可以指定服务或者路由资源的作用域。

此外，Istio还可以通过工作负载定义相应的标签，对关注的资源进行选取和过滤。

1.ExportTo属性

ExportTo属性用于指定核心资源的作用域，Istio作用域以命名空间为边界，当前VirtualService、DestinationRule和ServiceEntry均支持ExportTo属性，以VirtualService为例，ExportTo的定义如下：

```
message VirtualService {  
    ...  
    //由当前对象导出的namespace列表，用于限制Istio核心资源的可见性和作用域  
    //原则上，export_to可以设置为任意的namespace列表，但当前仅支持设置为.或*  
    //.表示当前对象是私有对象，作用域限制在当前namespace  
    //*表示当前对象是公有对象，可以作用于所有的namespace  
    //如果export_to没有设置任何namespace，则使用全局的缺省可见性语义  
    //可以设置全局缺省的可见性语义为.或*  
    repeated string export_to = 6;  
}
```

XDS配置变更时，会根据全局Mesh配置，设置VirtualService当前的缺省可见性，VirtualService资源初始化会根据ExportTo属性，将VirtualService资源划分为公有VirtualService资源和私有VirtualService资源。

当向具体的Envoy节点下发XDS配置时，会根据Envoy节点的命名空间，选取相应的VirtualService资源，具体查找逻辑如下：

```
func (ps *PushContext) VirtualServices(proxy *Proxy, gateways map[string]bool) []Config {
    configs := make([]Config, 0)
    //首先查找私有VirtualService
    if proxy == nil {
        for _, virtualSvcs := range ps.privateVirtualServicesByNamespace {
            configs = append(configs, virtualSvcs...)
        }
    } else {
        configs = append(configs, ps.privateVirtualServicesByNamespace[proxy.ConfigNamespace]...)
    }
    //然后查找公有VirtualService
    configs = append(configs, ps.publicVirtualServices...)
    ...
}
```

2.Sidecar Scope

Sidecar资源定义了一类Sidecar节点的行为，基于Sidecar资源，可以实现Sidecar节点监听行为的精确过滤，减少Sidecar节点对不必要信息的关注。Sidecar资源的定义如下：

```
message Sidecar {
    WorkloadSelector workload_selector = 1;
    repeated IstioIngressListener ingress = 2;
    repeated IstioEgressListener egress = 3;
}
```

其中，WorkloadSelector指定了具体的负载规则，通过WorkloadSelector可以将Envoy部署到某些特定的节点上，IstioIngressListener和IstioEgressListener分别用于对Inbound监听器与Outbound监听器进行定制，比如设置监听端口号，设置流量拦截策略等。

6.4.2 配置生成机制

XDS响应消息生成是根据当前Envoy节点的配置，将当前节点相关的服务配置和流量管理配置转换成XDS消息的过程，ConfigGenerator是Istio的XDS响应消息生成器，负责生成和数据平台Envoy进行交互的各种XDS响应消息，当前包括监听器配置LDS、HTTP路由配置RDS、集群配置CDS（EDS响应相对比较简单，当前没有在ConfigGenerator范围内），ConfigGenerator除了基本的核心实现外，还提供相应的扩展性机制。

1.通用扩展性机制

为了方便实现特性扩展，ConfigGenerator不仅支持基本信息的组织和拼装，还支持多个维度的插件扩展。Istio Pilot初始化时会注册当前使用的插件，XDS信息组织时会调用注册的插件回调函数进行扩展，比如对应上述扩展机制，LDS会调用当前注册插件的OnOutboundListener和OnInboundListener接口，分别进行Outbound和Inbound方向的监听器信息扩展。Istio当前支持authz、authn、health和mixer这几种扩展插件。

此外，Istio可以通过EnvoyFilter对象，对Envoy监听器的过滤链进行定制和扩展，实时控制Envoy的运行行为。具体实现由insertUserFilters负责完成，先通过getUserFiltersForWorkload获取当前Envoy节点对应的EnvoyFilter信息，然后按照EnvoyFilter中指定的具体位置，将自定义Filter对象放到指定位置。

2.监听器动态调整机制

为了灵活地对Istio服务运行状态进行观测和控制，Istio引入了在线动态调整机制，通过实时对Istio数据平面代理的监听器进行修改和调整，进行代理行为的个性化定制，可以实现诸如自定义方式查看流量状态、动态跟踪和调试以及自定义流量整形和路由调整等功能。

Istio的在线动态调整机制具体通过EnvoyFilter对象实现，EnvoyFilter对象的定义如下：

```
message EnvoyFilter {  
  //工作负载  
  map<string, string> workload_labels = 1;  
  //自定义过滤器列表  
  repeated Filter filters = 2;  
}
```

其中工作负载workload_labels用于指定该EnvoyFilter对象的作用范围，EnvoyFilter对象的突出特点是个性化定制，配置下发时，通过下面的getUserFiltersForWorkload函数获取和当前节点匹配的EnvoyFilter，获取成功时才会进行后续的EnvoyFilter处理。

```
func getUserFiltersForWorkload(in *plugin.InputParams) *networking.EnvoyFilter {  
  env := in.Env  
  //获取和当前Envoy节点对应WorkloadLabel匹配的EnvoyFilter  
  f := env.EnvoyFilter(in.Node.WorkloadLabels)  
  if f != nil {  
    return f.Spec.(*networking.EnvoyFilter)  
  }  
}
```

```
    return nil  
}
```

下面Filter对象用于定义自定义过滤器的具体行为，除了通过工作负载进行整个EnvoyFilter对象的匹配条件判断之外，每一个Filter还可以通过ListenerMatch指定精细粒度的匹配行为，当前已经支持通过IP地址或者端口号进行匹配过滤（比如只在特定地址上生效）和利用监听类型进行过滤（比如只在Gateway模式下生效）等多种过滤条件。

为了精细控制自定义过滤器的行为，Filter对象可以通过InsertPosition精确控制每个自定义过滤器在整个过滤器链的具体位置，比如在整个过滤链开头：整个过滤链结尾，甚至某个特定过滤器的前后等，通过InsertPosition，可以实现对流量和转发行为的精确观测与控制。

```
message Filter {  
    //自定义过滤器匹配条件  
    ListenerMatch listener_match = 1;  
    //插入位置  
    InsertPosition insert_position = 2;  
    //过滤器类型，当前主要有HTTP过滤器和NETWORK过滤器两种类型  
    FilterType filter_type = 3;  
    //过滤器名称  
    string filter_name = 4;  
    //过滤器配置  
    google.protobuf.Struct filter_config = 5;  
}
```

EnvoyFilter是Istio一个非常大胆的尝试和创新，通过EnvoyFilter机制，可以把所有的流量展示到用户面前，结合lua脚本等动态语言，可以真正实现可见可感、实时在线调整和生效。

基于EnvoyFilter的在线调整机制，会影响整个Istio网络的行为，稍有不慎，就会导致整个Istio网络不可用，因此一般情况下最好不要使用，特殊场景下使用时需要异常小心，配合相应的灰度机制，通过灰度观察确保没有问题之后再放量生效。

6.4.3 XDS配置生成实现

Envoy按照使用场景可以分为3种。

1) **model.Sidecar**: Sidecar模式，和应用服务一块部署在容器中，对进出应用服务的流量进行拦截。

2) **model.Router**: Router模式，作为独立的代理服务，对应用的L4/L7层流量进行代理。

3) **model.Ingress**: Ingress模式，作为集群入口的Ingress代理，对集群的入口流量进行拦截和代理。

其中，Router模式与Ingress模式均属于和应用服务不在一起部署的纯代理场景，在XDS配置方面没有任何区别，因此从XDS响应消息生成的角度看，可以归为一类，称为Gateway模式。

对于Sidecar模式来说，Envoy负责服务出入方向流量的透明拦截，并且出入方向的流量在监听管理、路由管理等方面有很大的区别，因此Sidecar的XDS配置均按照出入方向分别进行组织和管理。入流量一般称为Inbound，出流量一般称为Outbound。

由于Router模式和Ingress模式均属于单独的代理模式，在XDS配置管理方面没有大的差异，可以统一为Gateway模式，因此在XDS管理配置上可以划分为Sidecar Inbound模式、Sidecar Outbound模式和Gateway模式。

这几个模式下的XDS配置生成的方式也都比较类似，首先获取当前模式下Sidecar对应的服务列表，然后基于服务列表，拼装相应的XDS信息。

1.Inbound Sidecar

Inbound方向代表的是发往本节点的流量，同时Inbound只会将请求转发到和当前Sidecar部署在同一个部署单元上的服务节点上，因此Inbound方向的集群信息和路由信息都比较确定，只需要获取到当前Sidecar节点对应的服务信息即可。

对于HTTP来说，这里会拼装HTTP对应的路由信息，具体包含clusterName集群信息和VirtualHost信息。由于Inbound方向使用单一的集群、单一的VirtualHost，并且集群固定只有一个节点信息（本机节点），因此Inbound方向的路由查找和流量转发比较确定。Inbound方向的VirtualHost称为“Inbound|http|instance.Endpoint.ServicePort.Port”。

对于TCP来说，Inbound方向的路由信息当前未按照具体的协议类型进行区分，直接通过TCP Proxy的方式进行路由，这样只能进行全局的统计和管控，无法对Inbound方向的流量进行协议相关的链路治理和管控。

2.Outbound Sidecar

Outbound方向代表的是从当前节点发往本节点外的流量，因此首先需要获取当前节点

服务对应的上游服务列表，对于有作用域相关设置的Sidecar来说，需要根据作用域设置获取服务列表，对于没有作用域相关设置的Sidecar来说，返回的是注册中心的所有服务列表，因此当集群和服务规模比较大时，为了减少大量无效的XDS交互，需要设置相应的作用域信息。

获取到服务列表之后，需要根据服务当前支持的协议信息进行XDS响应消息的拼装，对于HTTP协议来说，一般通过80端口号对外提供服务，因此多个HTTP Outbound服务可以共用一个监听器“0.0.0.0: 80”，使用统一的HTTPConnManager对不同HTTP Outbound服务进行管理，使用80端口号作为Rds配置的RouteConfigName，Rds对应的具体路由配置由RDS协议负责解析。

由于具体的应用协议不同，TCP协议并且不像HTTP协议那样拥有知名端口号，因此无法直接使用端口号作为监听器标识。对于这类协议，一般会采用虚拟IP地址作为服务的监听标识。

3.Gateway

对于Gateway模式来说，Gateway对应的服务信息和路由信息通过Istio核心资源Gateway指定。Gateway节点配置生成时，首先根据当前Gateway节点的工作负载标签，获取匹配的Gateway核心资源列表，然后将这些核心资源列表进行合并处理，产出一个逻辑的Gateway，定义如下：

```
type MergedGateway struct {
    //物理端口到虚拟Server的映射
    Servers map[uint32][]*networking.Server
    //Server到Gateway名称的映射
    GatewayNameForServer map[*networking.Server]string
    //路由名到Server的映射
    ServersByRouteName map[string][]*networking.Server
    //Server到路由名的映射
    RouteNamesByServer map[*networking.Server]string
}
```

上述逻辑Gateway包括Gateway节点XDS配置生成需要的元数据信息，根据这些信息就可以拼装生成相应的XDS响应消息。

6.5 XDS配置的Envoy处理

6.5.1 XDS配置变更的判断

Envoy接收到控制平面发过来的XDS新配置后，首先需对比新配置和当前配置，判断是否有变更，如果没有任何变更，则不需要进行任何处理。

针对配置变更判断，Envoy对当前配置和新配置分别计算Hash值，Hash值的计算方法如下：

```
static std::size_t hash(const Protobuf::Message& message) {
    ProtobufTypes::String text;
    {
        Protobuf::io::StringOutputStream string_stream(&text);
        Protobuf::io::CodedOutputStream coded_stream(&string_stream);
        coded_stream.SetSerializationDeterministic(true);
        message.SerializeToCodedStream(&coded_stream);
    }
    return HashUtil::xxHash64(text);
}
```

将Protobuf消息序列化成字符串，然后生成相应的Hash值即可，基于Hash值的判断简单高效，可以避免大量无谓的消息解析和处理。如果Hash值相同，不进行任何处理；Hash值不相同，才需要进行XDS新配置的解析和加载。

6.5.2 CDS配置的延迟处理

针对配置频繁变更的场景，为了减少频繁变更对Envoy CPU计算的消耗，Envoy针对这种场景采用变更延迟处理的方式，CDS变更延迟处理特性的定义如下：

```
message Cluster {  
  ...  
  message CommonLbConfig {  
    //变更合并窗口，如果设置，窗口内的本集群的健康检查、权重、元数据等配置变更消息均合并在一起  
    //等到update_merge_window过期时间到达时，统一处理  
    //如果没有设置update_merge_window，则采用默认值1000ms  
    //如果想关闭这个特性，需要将update_merge_window设置为0  
    google.protobuf.Duration update_merge_window = 4;  
  }  
  CommonLbConfig common_lb_config = 27;  
}
```

CDS集群配置更新onClusterInit时，首先通过scheduleUpdate判断当前配置是否需要延迟处理，如果延迟处理，则当前不进行任何处理；否则，通过postThreadLocalClusterUpdate进行配置更新处理。

只有纯粹的更新操作才会进行延迟和合并处理，对于有新增或者删除节点的配置更新消息来说，需要立即处理。

```
void ClusterManagerImpl::onClusterInit(Cluster& cluster) {  
  ...  
  bool scheduled = false;  
  const auto merge_timeout = PROTOBUF_GET_MS_OR_DEFAULT(cluster.info()->lbConfig(), update_merge_window  
  const bool is_mergeable = !hosts_added.size() && !hosts_removed.size();  
  if (merge_timeout > 0) {  
    scheduled = scheduleUpdate(cluster, priority, is_mergeable, merge_timeout);  
  }  
  //如果配置更新没有被延迟处理，则立即通过postThreadLocalClusterUpdate进行投递处理，否则不进行任何处理  
  if (!scheduled) {  
    postThreadLocalClusterUpdate(cluster, priority, hosts_added, hosts_removed);  
  }  
  ...  
}
```

scheduleUpdate负责具体的延迟判断，确认需要延迟处理时，根据merge_timeout创建延迟定时器；如果不需要延迟处理，则删除之前创建的延迟定期器。

```
bool ClusterManagerImpl::scheduleUpdate(const Cluster& cluster, uint32_t priority, bool mergeable, const  
  ...  
  const uint64_t delta_ms = std::chrono::duration_cast<std::chrono::milliseconds>(delta).count();  
  const bool out_of_merge_window = delta_ms > timeout;  
  if (out_of_merge_window || !mergeable) {  
    updates->disableTimer();  
    updates->last_updated_ = time_source_.monotonicTime();  
    return false;  
  }  
  
  if (updates->timer_ == nullptr) {  
    updates->timer_ = dispatcher_.createTimer([this, &cluster, priority, &updates]() -> void {  
      applyUpdates(cluster, priority, *updates);  
    });  
  }  
}
```

```
    if (!updates->timer_enabled_) {  
        updates->enableTimer(timeout);  
    }  
    return true;  
}
```

6.5.3 集群和节点配置处理

1. 集群节点管理的整体结构

CDS和EDS负责集群和集群节点信息的管理，整体结构见图6-4。集群管理器ClusterManagerImpl是Envoy集群管理的入口，负责Envoy集群相关配置和操作的管理，由集群管理器factory ProdClusterManagerFactory具体创建。

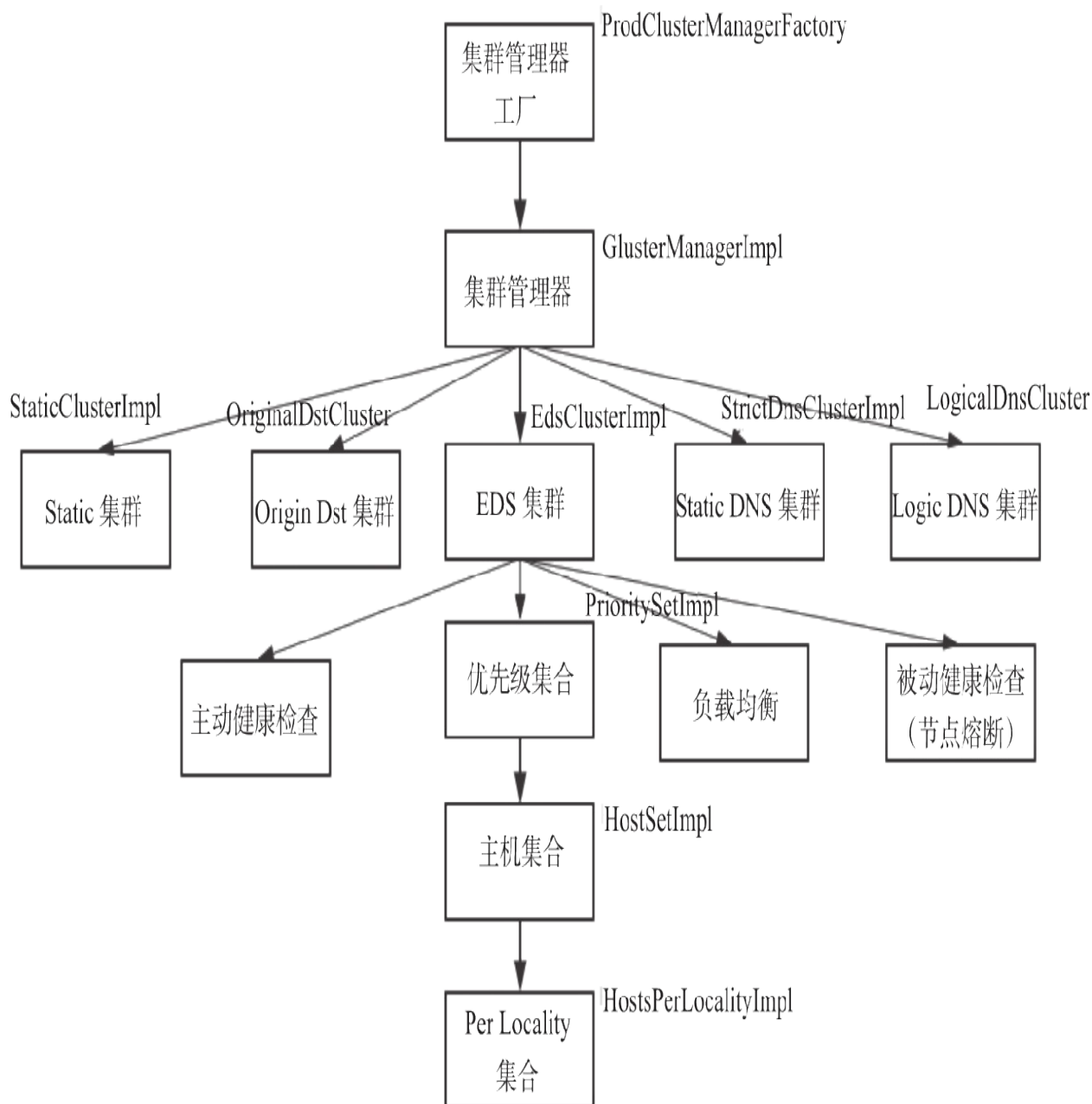


图6-4 Envoy 集群管理结构图

集群Cluster用于对Envoy的集群进行管理，主要包括集群节点信息、负载均衡、OutlierDetection和CircuitBreaker等，Envoy集群节点信息发现当前支持静态方式和动态方式，根据节点发现方式，Envoy集群当前支持如下几种类型：

```
enum DiscoveryType {  
    STATIC = 0;           //静态配置  
    STRICT_DNS = 1;       //通过DNS获取节点配置  
    LOGICAL_DNS = 2;      //和静态DNS相比，不会对upstream节点的连接池进行回收处理  
    EDS = 3;              //节点配置由控制平面服务注册中心管理，通过XDS协议获取  
    ORIGINAL_DST = 4;     //使用被透明拦截前的DST地址作为upstream节点  
}
```

集群节点信息的维护是集群管理的重中之重，Envoy的集群节点信息整体上分为优先级、Locality两个层次。一个集群的所有节点按照优先级不同划分为多个优先级集合（负载均衡时默认只会选取最高优先级集合里面的节点），Envoy的所有负载均衡机制都建立在感知地理位置的负载均衡算法之上，因此针对每个优先级集合，又按照地理位置不同，划分为多个Locality节点集合。

2.集群变更的触发和通知机制

Envoy接收到CDS、EDS分别下发的集群和节点配置变更后，主线程将根据最新配置构建出一颗新的集群配置树，然后通过TLS机制，通知所有的工作线程进行线程本地配置的更新。

对于集群和节点成员、节点状态信息的变化，当前主要有负载均衡、监控检查和节点熔断这几个子系统关注；Envoy的其他组件如果需要感知集群和节点相关的变更，可以通过如下系列接口注册相应的回调函数：

```
//集群成员变更回调  
addMemberUpdateCb  
//集群优先级变更回调  
addPriorityUpdateCb  
//集群健康检查状态变更回调  
addHostCheckCompleteCb  
//集群熔断状态变更回调  
addChangedStateCb  
//工作线程本地集群变更回调  
addThreadLocalClusterUpdateCallbacks
```

6.5.4 路由配置处理

RouteConfiguration用于对Envoy的路由进行管理，通过RouteConfiguration、VirtualHost和Route多级路由结构，构建起一个完整的路由查找树，Envoy接收到请求消息后，通过该路由查找树，查找到最合适的路由节点。Envoy路由分层结构如图6-5所示。

6.5.5 监听器配置处理

监听器Listener用于管理Envoy网络监听相关的配置，Envoy可以同时配置一到多个监听器，监听器通过Listener对象定义：

```
message Listener {
  string name = 1;
  core.Address address = 2;
  repeated listener.FilterChain filter_chains = 3;
  google.protobuf.BoolValue use_original_dst = 4 [deprecated = true];
  google.protobuf.UInt32Value per_connection_buffer_limit_bytes = 5;
  core.Metadata metadata = 6;
  DrainType drain_type = 8;
  repeated listener.ListenerFilter listener_filters = 9;
  google.protobuf.Duration listener_filters_timeout = 15;
  google.protobuf.BoolValue transparent = 10;
  google.protobuf.BoolValue freebind = 11;
  repeated core.SocketOption socket_options = 13;
  google.protobuf.UInt32Value tcp_fast_open_queue_length = 12;
}
```

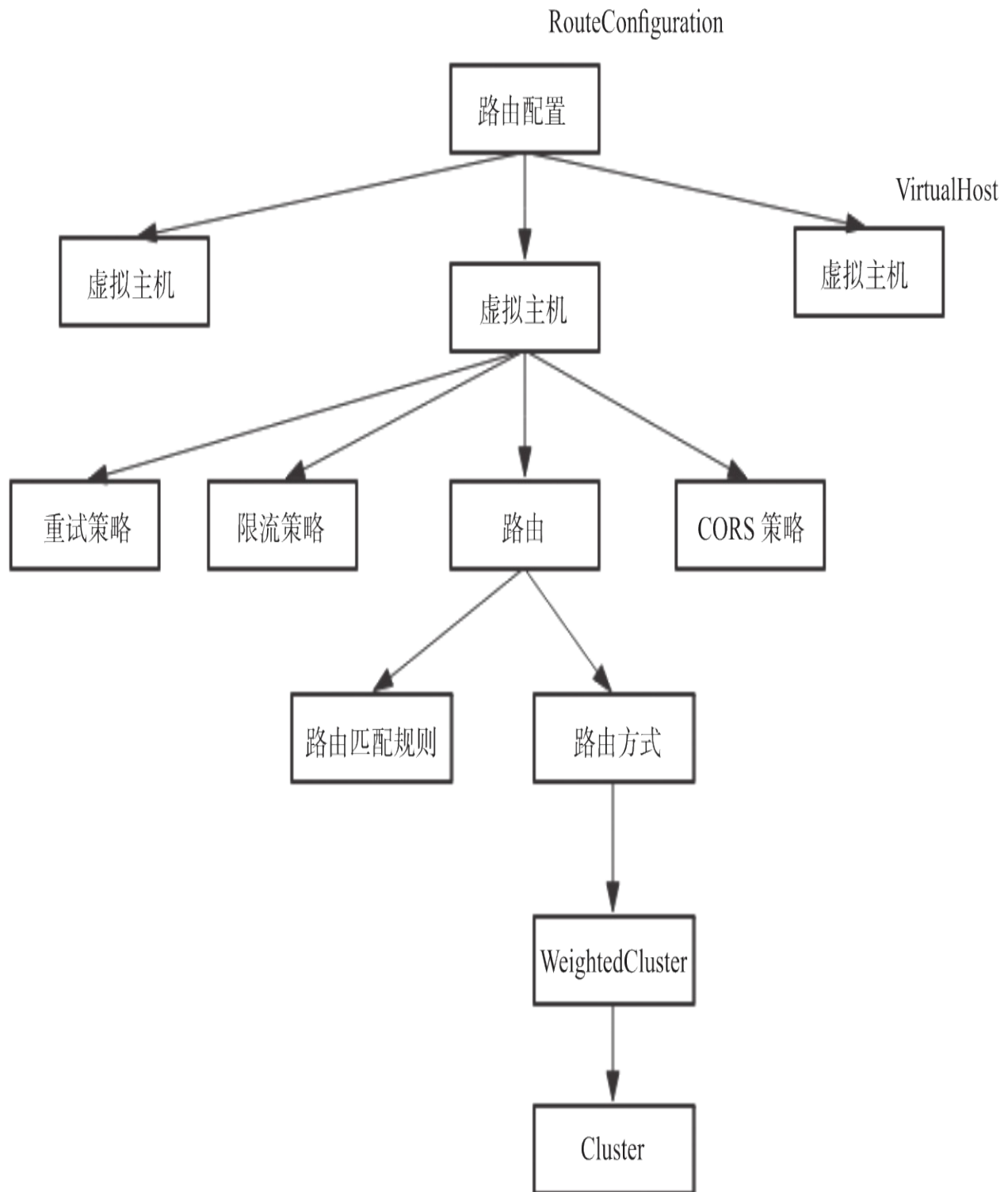


图6-5 Envoy路由分层结构

监听器配置中两个重要的信息是`listener_filters`和`filter_chains`，前者是监听器对应的过滤链，用于对监听行为进行扩展定制。比如`original_dst` filter用于获取被Iptables透明拦截后的请求原目的地址；`filter_chains`用于创建监听后的连接以及后续的请求处理，

`filter_chains`会指定对应的传输层以及协议处理，同时每个`filter_chain`都会有相应的匹配条件，比如按照目的地址匹配、按照源地址匹配等，监听器接收到新的连接后，会判断各个`filter_chain`的匹配条件，选取合适的过滤链处理。

6.6 本章小结

本章从Istio的控制流的入口开始，对Istio的控制流的全部过程进行深入的剖析，从Envoy启动、协议交互到配置解析等。通过本章的学习，可以全面了解到Istio配置处理的原理和细节，对接下来分析Istio的数据流也会有一定的帮助。

第7章 Istio数据流设计

本章从Istio数据流的入口开始，详细梳理Istio数据流的整个路径。

数据流层面对通信流量的透明拦截和调度，是Envoy最主要的价值所在，也是Envoy提供的最本质功能，下面会以最主流的HTTP协议请求处理为例，全面分析Istio数据流端到端的全过程，对于请求拦截部分，Istio当前默认通过Iptables进行，下面针对请求拦截的具体讨论也均会基于Iptables来展开，整个过程的主要处理步骤如下。

1) Client端Iptables的请求拦截：Client端Iptables基于配置的Iptables过滤规则，将请求消息交给client端Envoy。

2) Client端Envoy处理：Client端Envoy接着会进行连接建立、数据解析、流量路由、负载均衡、Upstream访问等一系列处理，将请求转发Server端。

3) Server端Iptables的请求拦截：Server端Iptables基于配置的Iptables过滤规则，将发往本机的请求消息交给Server端Envoy。

4) Server端Envoy处理：Server端Envoy会进行连接建立、数据解析、将请求转发给本机的Server服务处理、将Server服务返回的响应消息返回给Client端Envoy。

5) Client端Envoy对响应消息处理：Client端Envoy负责解析Upstream的响应消息，然后返回给Client端。

下面会结合Istio的具体实现，对上述Istio数据流的整个流程进行详细分析。

7.1 Iptables

7.1.1 Iptables的基本原理

Netfilter是Linux内核提供的一套完善的数据包处理框架，通过一整套的钩子（hook）管理机制，可以实现数据包过滤、网络地址转换以及基于协议类型的连接跟踪等功能，Iptables是基于Netfilter的数据包过滤系统，通过自己设定的规则以及处理动作对数据报文进行检测和处理。

Iptables分为内核模块和用户态两部分，用户空间的Iptables位于/sbin/目录下，是用于设置规则的命令行工具，通过命令行制定相应的规则，指定内核模块进行相应的数据访问控制。

链、表和规则是Iptables内核处理中非常重要的几个概念。

（1）链

链也可以称为钩子，是指Iptables过滤的时机。Iptables当前有PREROUTING、INPUT、FORWARD、OUTPUT和POSTROUTE 5种链，其中PREROUTING用于在路由选择前处理数据包，POSTROUTE用于在路由选择后处理数据包，INPUT用于处理发送到本机的数据包，OUTPUT负责处理本机发送出去的数据包，FORWARD负责对转发数据包进行处理。

针对不同的数据流，有如下3种典型的链处理时机。

1) 发送给本机的数据包：当数据包从物理层和数据链路层传输过来，如果数据包是访问Linux主机本身，则经过PREROUTING和INPUT钩子函数，到达传输层和应用层。

2) 转发数据包：数据包从物理层和数据链路层传输过来，如果数据包需要转发，则经过PREROUTING、FORWARD和POSTROUTE 3个钩子函数。

3) 发送数据包：当数据包从Linux主机本身向外发送数据包，要经过OUTPUT和POSTROUTE两个钩子函数。

（2）表

表是负责完成某个过滤功能的模块，Iptables当前有Filter表、NAT表、mangle表和raw表4个，它们在内核中对应4个不同的模块，这几个模块对应的功能和包含的钩子函数如下。

1) Filter表：Filter表是Iptables默认的过滤功能，确定是否放行数据包，包含INPUT、FORWARD和OUTPUT 3个钩子函数。

2) NAT表：NAT表负责修改数据包的源IPPORT和目的IPPORT，包含PREROUTING、OUTPUT和POSTROUTE这几个钩子函数。

3) **mangle表**: mangle表用于对连接进行跟踪, 包含PREROUTING、INPUT、FORWARD、OUTPUT和POSTROUTE 5个钩子函数。

4) **raw表**: raw表用于关闭NAT表上开启的连接跟踪功能, 包含PREROUTING、OUTPUT 2个钩子函数。

综合来说, Iptables、表、链和规则之间是层层包含的关系, Iptables可以包含多个表, 每个表包含多个链, 每个链包含多个用于过滤的规则。

7.1.2 Iptables在Istio中的使用

Iptables负责对Envoy的入口和出口流量进行拦截。针对入口流量拦截，假设所有的流量均需要定向到Envoy代理，8090是Envoy的监听端口，Iptables规则如下：

```
Iptables -t nat -A PREROUTING -p tcp -j REDIRECT -to-port 8090
```

针对出口流量拦截，也是类似的思路，Iptables规则如下：

```
Iptables -t nat -A OUTPUT -p tcp -j REDIRECT -to-port 8090
```

通过Iptables拦截转发后，目标端口被改写了，可以通过SO_ORIGINAL_DST TCP套接字获取原始的ipport。

Istio通过proxy_init组件进行基于Iptables的流量透明拦截，proxy_init组件实际执行的命令是tools/deb/istio-iptables.sh（这是一个脚本程序，包括很多Iptables命令），通过配置Iptables来劫持Pod中的流量。

7.2 监听管理

监听是处理网络请求的第一步，Envoy使用监听器来具体管理监听请求工作，可以同时支持多个监听器，每个监听器负责一类网络请求的监听行为。

为了减少对业务的影响，Istio对业务流量采取了透明拦截的方式，Iptables等将拦截后的流量转发到Envoy，因此，为了实现正确的流量路由和转发，Envoy的监听器分为两类：

1) **虚拟监听器**：需要绑定相应的端口号，同时配置相应的Iptables规则，将Iptables拦截的流量转发到虚拟监听器对应的端口上；

2) **真实监听器**：用于处理Iptables拦截前的“真实目的地址”，虚拟监听器接收到监听请求时，按照一定的匹配规则查找对应的真实监听器，将新监听请求转交给真实监听器进行处理，真实监听器由于不需要和网络进行交互，因此不需要配置和绑定具体的端口号。

7.2.1 监听器建立

LDS配置首次获取或者更新时，均需要建立新的监听器。新建监听器主要有两个工作：一个是对监听过滤插件和网络过滤插件进行初始化，供后续建立新连接时使用；还有一个是创建监听器对应的套接字。

Envoy当前支持UNIX域套接字、UDP和TCP 3种监听方式，对于需要绑定具体端口号的监听器来说，创建相应的套接字后，绑定到相应的端口。

对于热重启的场景来说，为了保证旧的Envoy进程优雅退出，新启动的Envoy进程直接使用旧进程的监听套接字，保证新旧进程之间完美衔接，从而不会带来流量的损失。

7.2.2 监听器和工作线程绑定

7.2.1节中创建的监听器只是“逻辑”上的监听器，并没有和真实的网络建立起联系，因此还不能提供实际的监听工作。

Envoy下各个工作线程拥有独自的事件调度器，创建“逻辑”监听器后，会遍历当前所有的工作线程，将监听器加入到每个工作线程中，各个工作线程独立建立自己网络层面的监听器。

网络监听器具体由事件调度器DispatcherImpl触发创建，具体如下：

```
Network::ListenerPtr
DispatcherImpl::createListener(Network::Socket& socket, Network::ListenerCallbacks& cb, bool bind_to_po
    return Network::ListenerPtr{new Network::ListenerImpl(
        *this, socket, cb, bind_to_port, flag)};
}
```

每个工作线程创建相应的ListenerImpl实例，ListenerImpl实例会通过evconnlistener_new提交监听事件到libevent调度器，监听回调函数为listenCallback，同时当新连接到来时会触发listenCallback的回调。

```
ListenerImpl::ListenerImpl(Event::DispatcherImpl& dispatcher, Socket& socket, ListenerCallbacks& cb, bool
    : BaseListenerImpl(dispatcher, socket), cb_(cb) {
    if (bind_to_port) {
        listener_.reset(evconnlistener_new(&dispatcher.base(),
            listenCallback, this, 0, -1, socket.ioHandle().fd()));
    }
}
```

ListenerImpl实例创建时会传入之前的逻辑监听器，在listenCallback函数中，直接调用逻辑监听器的onAccept函数进行新连接相关的处理。

```
void ListenerImpl::listenCallback(evconnlistener*, evutil_socket_t fd, sockaddr* remote_addr, int remote_
    ListenerImpl* listener = static_cast<ListenerImpl*>(arg);
    ...
    listener->cb_.onAccept(
        std::make_unique<AcceptedSocketImpl>(std::move(io_handle), local_address, remote_address),
        listener->hand_off_restored_destination_connections_);
}
```

7.3 连接管理

连接管理负责创建客户端连接，在实际创建网络层连接之前，需要根据客户端的元数据信息匹配适合的监听器和协议过滤插件，协议过滤插件确定后，新连接相关的参数和准备工作就均已确定，从而可以进入网络层新连接的实际创建。

7.3.1 监听器匹配

Envoy对通信进行接管的基础是透明拦截，拦截后请求的目的地址被统一替换为Envoy虚拟监听器地址，为了完成后续的请求流量识别和路由转发，需要能够有相应的机制还原出原来的目的地址，并根据该目的地址查找到匹配的监听器。

从7.2.2节可以看出，新连接到来时会触发逻辑监听器的接收新连接逻辑onAccept函数，onAccept函数的主要逻辑是创建并执行监听过滤插件。

```
void ConnectionHandlerImpl::ActiveTcpListener::onAccept(
    Network::ConnectionSocketPtr& socket, bool flag) {
    auto active_socket = std::make_unique<ActiveSocket>(*this,
        std::move(socket), flag);
    //创建监听过滤插件列表
    config_.filterChainFactory().createListenerFilterChain(*active_socket);
    //执行列表中的监听过滤插件
    active_socket->continueFilterChain(true);
    ...
}
```

Envoy当前支持多个监听过滤插件，最重要的是原目的地址过滤插件OriginalDstFilter。OriginalDstFilter的实现很简单，即调用getOriginalDst，通过SO_ORIGINAL_DST套接字选项，获取真实的目的地址，并设置到socket中。

```
Network::FilterStatus OriginalDstFilter::onAccept(Network::ListenerFilterCallbacks& cb) {
    Network::ConnectionSocket& socket = cb.socket();
    const Network::Address::Instance& local_address = *socket.localAddress();
    if (local_address.type() == Network::Address::Type::Ip) {
        Network::Address::InstanceConstSharedPtr original_local_address = getOriginalDst(socket.fd());
        if (original_local_address) {
            //还原原目的地址
            socket.restoreLocalAddress(original_local_address);
        }
    }
    return Network::FilterStatus::Continue;
}
```

continueFilterChain函数中，再执行完监听过滤插件后，查找流量拦截后的监听器，进行相应的新连接处理，否则，启动创建传输层新连接的流程。

```
void ConnectionHandlerImpl::ActiveSocket::continueFilterChain(bool success) {
    ...
    if (hand_off_restored_conn_ && socket->localAddressRestored()) {
        //根据目的地址查找匹配的逻辑监听器
        new_listener = findActiveListenerByAddress(*socket->localAddress());
    }
    if (new_listener != nullptr) {
        tcp_listener->onAccept(std::move(socket_), false);
    } else {
        //创建传输层的新连接
        listener_.newConnection(std::move(socket_));
    }
}
```

7.3.2 协议过滤器匹配

在Envoy中，每个监听器下会对应多个网络过滤插件，每个过滤插件对应一种网络协议类型，处理相应协议的请求解析和路由。在进行监听器配置时会配置每个过滤插件的匹配条件（通过FilterChainMatch字段表示），具体包含目的地址、目的端口号、源地址等，创建新连接时依次根据目的地址、目的端口号等进行匹配判断，查找最合适的网络过滤插件。

7.3.3 创建新连接

确定好新连接的网络过滤插件后，新连接对应的传输层也相应确定了，接下来就可以进行传输层新连接的创建流程。

newConnection负责新连接的创建。首先创建传输层的连接，然后提交读写就绪事件到事件调度器，等待新数据的到来。

```
void ConnectionHandlerImpl::ActiveTcpListener::newConnection(
    Network::ConnectionSocketPtr&& socket) {
    ...
    auto transport_socket = filter_chain->transportSocketFactory().createTransportSocket(nullptr);
    Network::ConnectionPtr new_connection =
        parent_.dispatcher_.createServerConnection(std::move(socket), std::move(transport_socket));
    onNewConnection(std::move(new_connection));
}
```

createServerConnection函数负责网络层面的服务段连接，这里只是简单地创建连接对应的管理实例**ConnectionImpl**。

```
Network::ConnectionPtr
DispatcherImpl::createServerConnection(Network::ConnectionSocketPtr&& socket, Network::TransportSocketPtr&
    return std::make_unique<Network::ConnectionImpl>(*this, std::move(socket), std::move(transport_socket),
}
```

在**ConnectionImpl**中，向事件调度器注册socket读写事件回调函数**onFileEvent**，当请求数据到来时，事件调度器就会调用**onFileEvent**进行实际的请求处理。

```
ConnectionImpl::ConnectionImpl(Event::Dispatcher& dispatcher,
    ConnectionSocketPtr&& socket,
    TransportSocketPtr&& transport_socket, bool connected) {
    ...
    file_event_ = dispatcher_.createFileEvent(
        fd(), [this](uint32_t events) -> void { onFileEvent(events); }, Event::FileTriggerType::Edge,
        Event::FileReadyType::Read | Event::FileReadyType::Write);
}
```

7.4 网络I/O和缓冲区管理

建立新连接时已经设置了连接读写就绪事件对应的处理函数`ConnectionImpl: : onFileEvent`。`onFileEvent`是所有网络事件的总入口，当新数据到来时，`libevent`就会触发对`ConnectionImpl: : onFileEvent`的调用，启动传输层的数据读取和解析流程。传输层数据读取完成后，会通过插件机制，进行协议相关的解析和处理。

7.4.1 传输层数据读取

传输层负责实际的I/O读写操作，Envoy在传输层方面支持插件扩展，当前已经支持RawBuffer、TLS以及TAP等多种传输层协议。

1.RawBuffer

RawBuffer是最基础的传输层实现，底层实际使用的是libevent的缓冲区管理机制evbuffer，将就绪的请求数据从内核接收到用户态的缓存区中。

数据读取时，为了在内存占用和读取性能之间达到一定的平衡，读取缓存区当前设置为16KB，这是一个经验值，在具体使用过程中可以根据场景需要进行一定的调整。

2.TLS

TLS是当前默认的传输层实现，也是Envoy安全特性的最集中体现，通过身份认证、证书等机制可以实现安全的微服务通信。

在架构层面，Envoy的TLS实现以open ssl为基础；在证书管理方面，通过SDS协议和控制平面进行证书的协商和交互。

3.TAP传输层

有时出于诊断、调试等目的，需要将符合一定条件的线上流量录制下来，供线下问题定位和流量分析使用，TAP传输层就是为了实现类似的功能。在RawBuffer或TLS等基础传输层上进行一层TAP封装，接收到基础传输层的数据后，TAP按照事先配置好的规则进行流量匹配和筛选，比如根据HTTP消息头进行判断等，对符合要求的流量进行相应的录制处理，比如通过日志进行记录、通过gRPC协议发送到远端服务等。

7.4.2 插件处理

在传输层完成读取数据后，就开始进行核心的处理工作，包括协议解析、请求路由、链路治理和管控等。Envoy实际的处理工作都通过插件的方式封装，当然最重要的是协议处理插件，负责相应协议的请求解析和流量转发，每种协议对应一个协议处理插件。Envoy当前内置支持HTTP、Redis、Thrift、Mongo、Dubbo等众多协议插件，它们由FilterManagerImpl负责统一管理。

新连接建立时已经通过FilterManagerImpl设置了连接对应的协议处理插件。请求数据读取完成时，通过FilterManagerImpl的onRead函数触发整个插件的读过滤过程。

```
void FilterManagerImpl::onRead() {
    onContinueReading(nullptr);
}
void FilterManagerImpl::onContinueReading(ActiveReadFilter* filter) {
    ...
    for (; entry != upstream_filters_.end(); entry++) {
        if (!(*entry)->initialized_) {
            (*entry)->initialized_ = true;
            FilterStatus status = (*entry)->filter_->onNewConnection();
            ...
        }
        BufferSource::StreamBuffer read_buffer = buffer_source_.getReadBuffer();
        if (read_buffer.buffer.length() > 0 || read_buffer.end_stream) {
            FilterStatus status = (*entry)->filter_->onData(read_buffer.buffer, read_buffer.end_stream);
            ...
        }
    }
}
```

FilterManagerImpl的onRead会遍历所有的Upstream插件，每个插件主要完成两个工作：

(1) 连接初始化

在连接刚创建时，触发插件的onNewConnection流程，插件会进行连接相关的初始化工作，比如初始化upstream连接等。

(2) 数据处理

调用插件的onData函数，进行插件的具体协议解析和路由流程。这里需要强调的是，对于I/O连接层面Envoy采用的是异步的方式，一个请求可能会触发多次读取操作，因此单个请求可能会多次调用插件的onData函数，插件在协议解析时需要注意这一点。

7.5 Thrift协议处理

Thrift拥有强大的性能和特性支持，同时内置成熟完善的多语言和RPC体系，在很多公司得到广泛应用。Envoy当前对Thrift的支持已经相对比较完善了，下面就以Thrift协议代理为例，讲述Envoy对以TCP为基础的二进制协议的支持。

7.5.1 Thrift插件的整体架构

下面首先以Thrift插件的核心对象为基础，介绍Thrift插件的整体架构，如图7-1所示。

ConnectionManager是Thrift插件的处理入口，同时也负责请求整个处理过程的流程编排。ConnectionManager从FilterManager接收到Thrift请求消息后，通过Decoder进行协议解码，ActiveRpc负责对下游发过来的请求消息进行处理，处理具体是通过注册到ActiveRpc上的DecodeFilter来完成的。Decoder解码分为多个阶段，解析到每个阶段时均会回调ActiveRpc，ActiveRpc遍历自身的插件列表，进行相应的插件处理。Thrift当前仅支持路由和限流两个插件，其中路由插件负责Upstream节点的选取和连接管理。Decoder解码完成后，路由插件对解析后的请求进行编码，并通过建立好的Upstream发送出去。同时Router注册Upstream连接的读写事件，Upstream响应消息到来时，会通过ResponseDecoder进行解码，然后进行一系列的处理后，通过Downstream连接返回给下游。

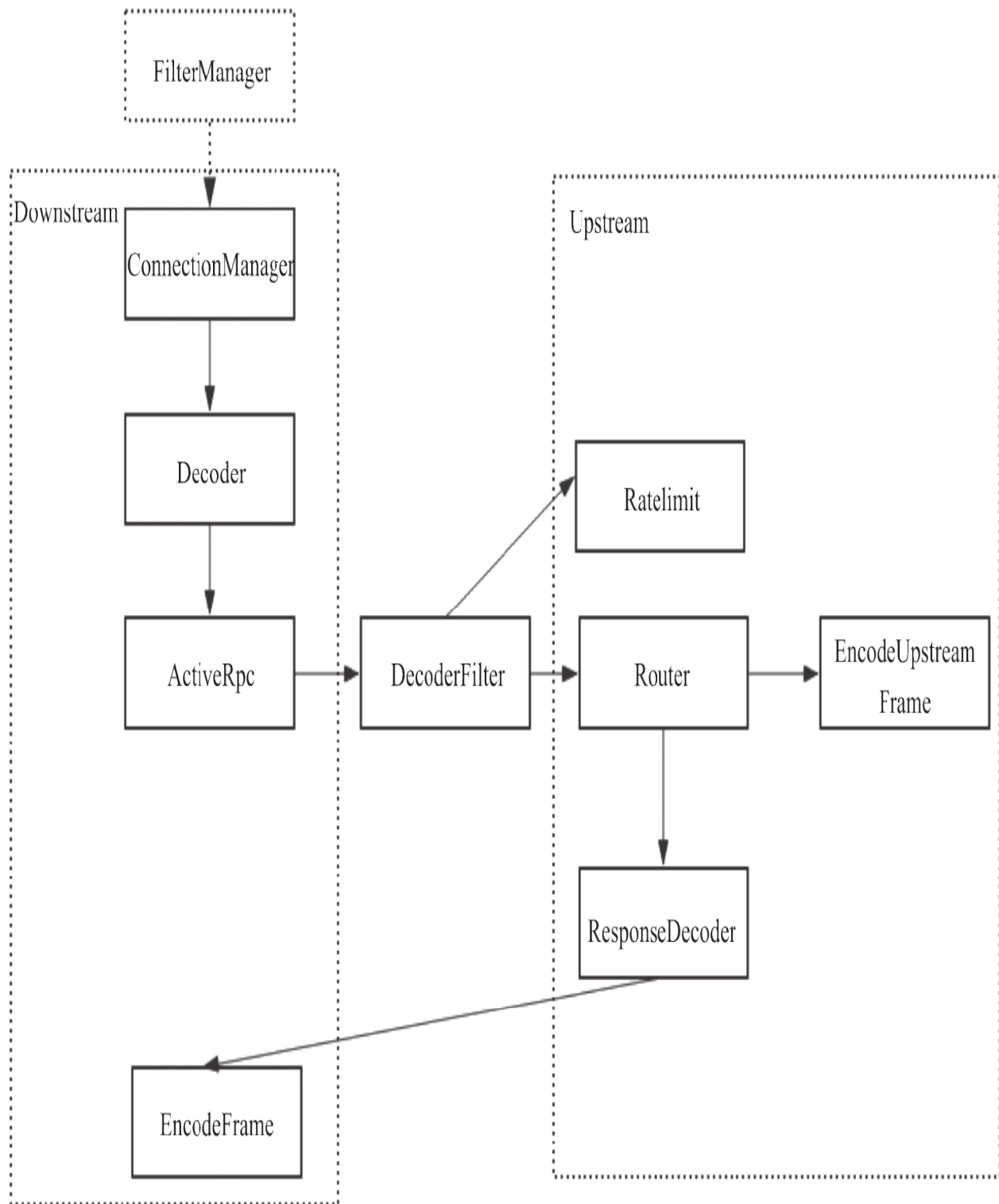


图7-1 Thrift插件的整体架构

7.5.2 协议解析

Envoy的Thrift插件支持多种协议层和传输层实现。协议层当前支持BinaryProtocol、CompactProtocol、TwitterProtocol、AutoProtocol等多种协议层实现，尤其需要提到的是AutoProtocol。它是在BinaryProtocol、CompactProtocol、TwitterProtocol这些协议层基础上的抽象协议层，可以根据RPC请求内容，自动判断出当前请求对应的协议层类型，通过这种方式，Envoy可以透明兼容多种协议层实现，不需要进行硬编码设置，具体实现方式如下：

```
bool AutoProtocolImpl::readMessageBegin(Buffer::Instance& buffer, MessageMetadata& metadata) {
    if (protocol_ == nullptr) {
        if (buffer.length() < 2) {
            return false;
        }

        uint16_t version = buffer.peekBEInt<uint16_t>();
        if (BinaryProtocolImpl::isMagic(version)) {
            // 12字节是BinaryProtocol消息头的最小长度
            if (buffer.length() < BinaryProtocolImpl::MinMessageBeginLength) {
                return false;
            }

            // TwitterProtocol的第一个消息总是一个升级请求，因此事先预留足够的缓冲区
            // 进行TwitterProtocol检测，如果不满足，则退回到二进制协议
            if (TwitterProtocolImpl::isUpgradePrefix(buffer)) {
                setType(ProtocolType::Twitter);
            } else {
                setType(ProtocolType::Binary);
            }
        } else if (CompactProtocolImpl::isMagic(version)) {
            setType(ProtocolType::Compact);
        }
    }
    return protocol_>readMessageBegin(buffer, metadata);
}
```

传输层也采用类似的方式，通过抽象传输层AutoTransport自动识别包括HeaderTransport、FramedTransport和UnframedTransport等多种传输层协议，实现了业务透明的传输支持。

为了方便对协议解析过程进行透明扩展和定制，Envoy支持在解码的每个过程中进行回调处理，DecoderEventHandler定义了解码操作接口，需要重点关注的是messageBegin和messageEnd，分别对应开始解码和解码完成。

7.5.3 协议相关的插件机制

Thrift内部通过插件机制实现链路治理特性的扩展，Thrift插件初始化时会根据配置加载和初始化当前配置的插件列表，实现如下：

```
void ConfigImpl::processFilter(const xxxThriftFilter& proto_config) {
    const ProtobufTypes::String& string_name = proto_config.name();
    //根据插件名获取插件factory
    const Json::ObjectSharedPtr filter_config = MessageUtil::getJsonObjectFromMessage(proto_config.config());
    auto& factory = Envoy::Config::Utility::getAndCheckFactory<ThriftFilters::NamedThriftFilterConfigFactory>();
    //根据插件factory初始化插件实例
    ProtobufTypes::MessagePtr message = Envoy::Config::Utility::translateToFactoryConfig(proto_config, factory);
    ThriftFilters::FilterFactoryCb callback = factory.createFilterFactoryFromProto(*message, stats_prefix_,
    filter_factories_.push_back(callback);
}
```

Thrift请求处理过程中，通过newDecoderEventHandler创建插件过滤链，各个插件通过addDecoderFilter将自身注册到插件过滤链中：

```
void ConfigImpl::createFilterChain(ThriftFilters::FilterChainFactoryCallbacks& callbacks) {
    for (const ThriftFilters::FilterFactoryCb& factory : filter_factories_) {
        factory(callbacks);
    }
}

DecoderEventHandler& ConnectionManager::newDecoderEventHandler() {
    ActiveRpcPtr new_rpc(new ActiveRpc(*this));
    //创建请求对应的过滤插件
    new_rpc->createFilterChain();
    new_rpc->moveIntoList(std::move(new_rpc), rpcs_);
    return **rpcs_.begin();
}
```

addDecoderFilter会为每个插件创建一个ActiveRpcDecoderFilter实例，同时挂接到插件过滤链decoder_filters_中：

```
void addDecoderFilter(ThriftFilters::DecoderFilterSharedPtr filter) override {
    ActiveRpcDecoderFilterPtr wrapper = std::make_unique<ActiveRpcDecoderFilter>(*this, filter);
    filter->setDecoderFilterCallbacks(*wrapper);
    wrapper->moveIntoListBack(std::move(wrapper), decoder_filters_);
}
```

Thrift插件支持在Thrift请求解析的各个阶段增加扩展点，当前支持的阶段有：

messageBegin	//消息解析开始
messageEnd	//消息解析结束
transportBegin	//传输层解析开始
transportEnd	//传输层解析结束
structBegin	//struct解析开始
structEnd	//struct解析结束
fieldBegin	//字段解析开始
fieldEnd	//字段解析结束

以transportBegin为例，只需要设置当前的处理阶段，以及对应的插件回调函数，然后

通过统一的插件触发函数applyDecoderFilters进行插件分发：

```
FilterStatus ConnectionManager::ActiveRpc::transportBegin(MessageMetadataSharedPtr metadata) {
    filter_context_ = metadata;
    filter_action_ = [this](DecoderEventHandler* filter) -> FilterStatus {
        MessageMetadataSharedPtr metadata = absl::any_cast<MessageMetadataSharedPtr>(filter_context_);
        return filter->transportBegin(metadata);
    };

    return applyDecoderFilters(nullptr);
}
```

applyDecoderFilters是Thrift插件的总入口，在这里会遍历当前注册的插件列表，分别调用插件的当前阶段处理函数进行处理。

7.6 HTTP请求处理

HTTP协议是微服务交互中使用最广泛并且也是最成熟的通信协议，可以认为是微服务通信领域的第一等公民，Envoy也是将绝大部分精力用于支持HTTP通信，当前已经有完善的集群通信和链路治理支持，下面会从协议解析、路由管理、负载均衡、连接池实现等维度详细剖析HTTP请求的处理过程。

7.6.1 HTTP请求处理流程

HTTP请求处理的主要逻辑均通过HTTP过滤插件提供，HTTP过滤插件分为请求头处理、请求体处理和Trails 3个阶段。HTTP请求处理的整体流程如图7-2所示。

接收到HTTP请求消息后，首先创建HTTP协议解析器，对请求数据进行解析。

请求解析完成后，就会进行Upstream请求的准备，这里包括完整的集群管理逻辑，具体是路由查找、负载均衡、连接池管理等，成功获取Upstream连接后，就可以基于该连接进行和Upstream之间的交互。

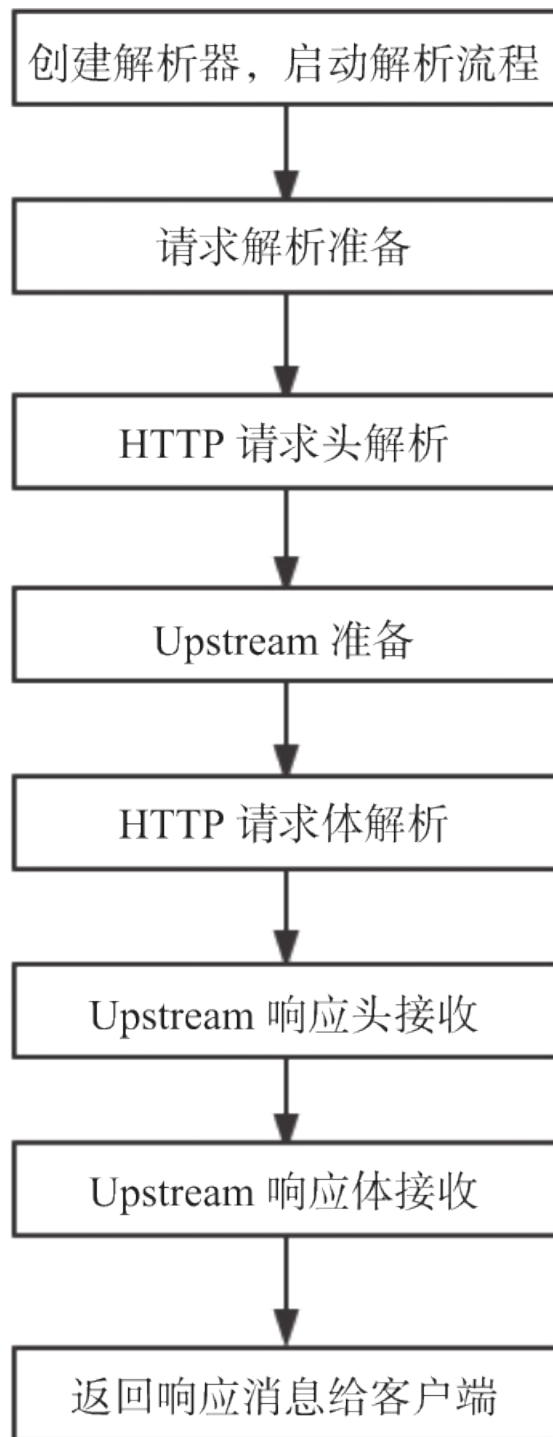


图7-2 HTTP请求处理流程

通过Upstream连接发送数据时，同时指定连接对应的读写函数，当Upstream响应数据到来时，通过已经注册的读写函数对Upstream响应消息进行解析，然后返回最终的响应消息给Downstream。

7.6.2 协议解析

Envoy中每种网络协议对应相应的协议过滤插件。之前的控制流设计中，已经详细讨论过`HttpConnectionManager`的解析流程，在`HttpConnectionManagerFilterConfigFactory::createFilterFactoryFromProtoTyped`中创建HTTP连接管理器`ConnectionManagerImpl`，并将`ConnectionManagerImpl`加入到网络过滤链中。

HTTP请求到达时，触发`ConnectionManagerImpl::onData`函数，正式进入HTTP的数据解析过程，在`ConnectionManagerImpl::onData`中，首先会根据当前配置创建相应的解析器，当前支持HTTP1和HTTP2两种方式，然后调用解析器的`dispatch`流程，开始协议数据的解析。

HTTP1使用开源的HTTP消息解析器`http-parse`（GitHub地址<https://github.com/nodejs/http-parser>，感兴趣的同学可以详细了解）进行协议数据解析。`http-parse`是一个用C语言编写的HTTP消息解析器，可以对HTTP请求和响应消息进行解析，`http-parse`的架构非常轻便，不依赖任何第三方库，解析过程中不会进行任何系统调用及内存分配，不会缓冲数据，可以被随时中断，设计上非常巧妙和高效，使用上也特别简单，被广泛用于高性能HTTP应用程序中。

使用`http-parse`时，首先通过`http_parser_init`初始化解析器，指定当前解析的是请求还是响应消息，然后调用`http_parser_execute`进行实际的解析操作，同时需要传入当前解析使用的回调函数。回调函数包含HTTP解析过程中的所有环节，比如URL解析回调函数、消息头字段名解析回调函数、消息头字段值解析回调函数、消息体解析回调函数等，它们通过`http_parser_settings`进行统一管理。使用`http-parse`解析过程中，会调用`http_parser_settings`中对应的回调函数进行处理。

对于HTTP请求解析来说，首先调用`onMessageBeginBase`触发消息解析开始流程，然后依次进行请求头和请求消息体解析，其中请求头解析完成后调用`onHeadersCompleteBase`，触发之后的一些处理工作，请求全部解析完毕后调用`onMessageCompleteBase`进行本次解析的收尾工作，因此`onMessageBeginBase`、`onHeadersCompleteBase`和`onMessageCompleteBase`是HTTP请求解析过程中的3个关键节点，下面也会围绕这3个节点，展开讨论HTTP解析的整个过程。

```
void ServerConnectionImpl::onMessageBegin() {
    if (!resetStreamCalled()) {
        active_request_ = std::make_unique<ActiveRequest>(*this);
        active_request_>request_decoder_ = &callbacks_.newStream(active_request_>response_encoder_);
    }
}

struct ActiveRequest {
    ActiveRequest(ConnectionImpl& connection) : response_encoder_(connection) {}
    HeaderString request_url_;
    StreamDecoder* request_decoder_{};
    ResponseStreamEncoderImpl response_encoder_;
    bool remote_complete_{};
};

ConnectionManagerImpl::ActiveStream::ActiveStream(ConnectionManagerImpl& connection_manager): connection_
    ...
}
```

onMessageBegin是请求解析过程的起点，负责请求解析过程中相关的准备工作。在ServerConnectionImpl中，ActiveRequest负责管理Envoy和DownStream之间的交互，具体包括DownStream请求的解码以及DownStream响应消息的编码工作，首先调用ConnectionManagerImpl::newStream进行请求解析流的创建。在newStream函数中创建ActiveStream对象，具体管理请求解析的整个过程，ActiveStream有个很重要的字段snapped_route_config_，从初始化信息看对应的就是ConfigImpl。ActiveRequest的request_decoder_负责对请求进行解码，response_encoder_负责对返回给调用方的响应消息进行编码，这里先进行创建，后续在请求处理完成后会用到。

onHeadersCompleteBase和onMessageCompleteBase分别对应HTTP消息头和HTTP消息体的处理。

7.6.3 路由管理

路由发现服务RDS负责HTTP请求的路由查找和管理，需要特别强调的是，RDS仅适用于HTTP协议，Envoy中的其他协议当前还没有相应的XDS支持。

RDS中的HTTP路由采用树状分层结构，根节点为虚拟主机层VirtualHost、然后是路由策略层Route，接着是可选的权重路由节点WeightRouteEntry，最后是叶子节点RouteEntry，路由查找就是从根节点出发，经过层层匹配，查找到叶子节点RouteEntry的过程。

HTTP协议处理主要是由一系列HTTP过滤插件组成。其中Router过滤插件具体负责路由查找、负载均衡以及和上游Upstream之间的交互工作。由于目的路由节点RouteEntry承载了很多服务治理相关的特性，除了Router过滤插件之外，其他HTTP过滤插件也会用到目的路由RouteEntry，比如HTTP限流插件需要使用RouteEntry中的限流策略。因此，Envoy将RDS路由查找的工作提前到插件处理之前，查找到相应的路由结果后，将该路由信息缓存起来，供本次请求接下来的HTTP插件使用。

HTTP插件在处理过程中可能会修改路由相关的元数据信息，比如HTTP消息头，这样就会导致路由结果发生变化。因此，如果出现导致路由结果发生变化的场景，需要先清除路由缓存信息，然后对路由信息进行刷新。

HTTP路由的大体流程如图7-3所示。

如图7-3所示，HTTP路由管理围绕路由缓存展开，HTTP插件处理前通过RDS配置查找路由结果并设置路由缓存；HTTP插件在需要时查找路由缓存获取路由信息；在修改路由元数据时清除路由缓存并刷新最新的路由信息。

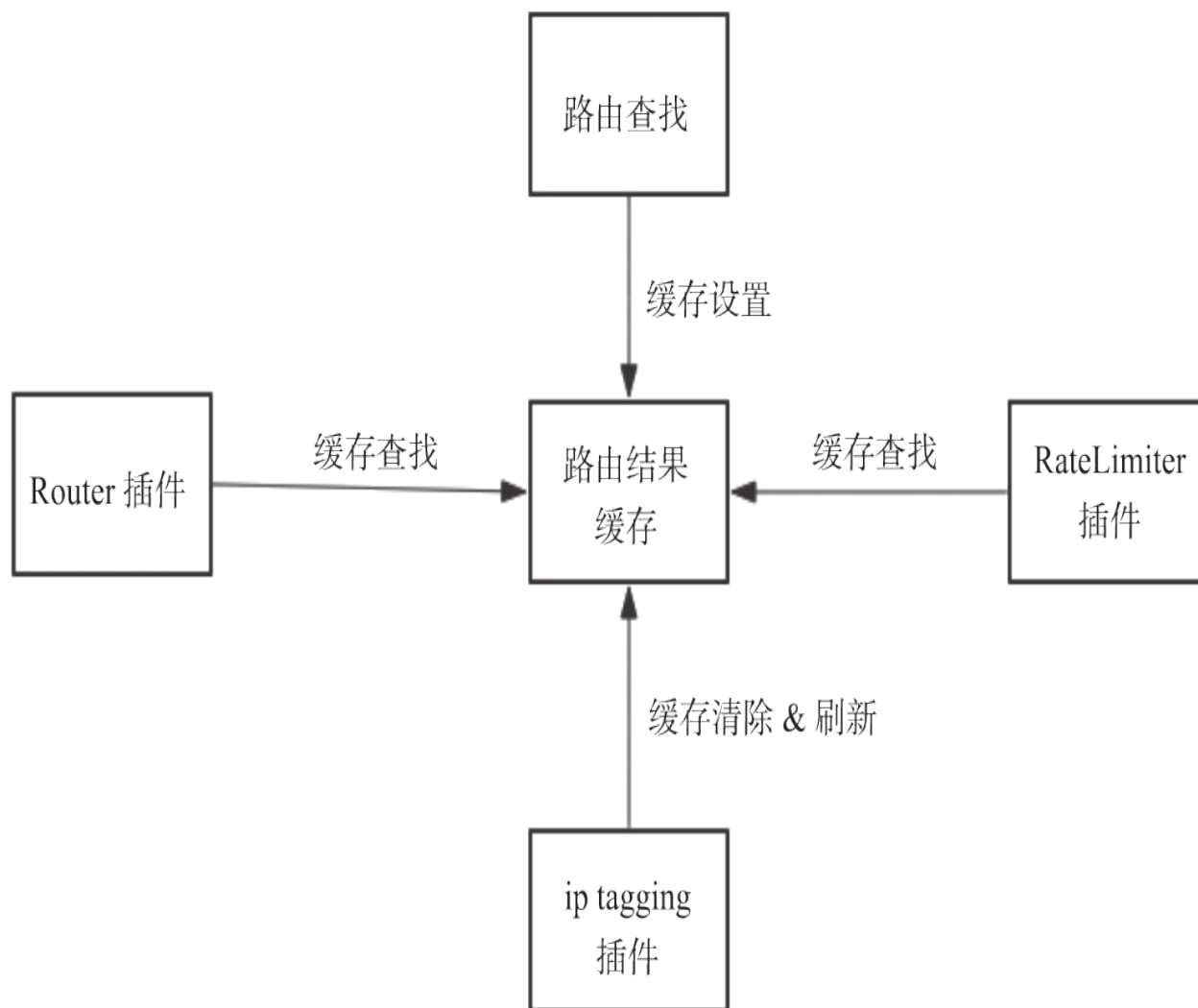


图7-3 HTTP路由管理

HTTP的路由缓存设置在HTTP消息头解析后触发，具体由refreshCachedRoute完成：

```

void ConnectionManagerImpl::ActiveStream::refreshCachedRoute() {
    Router::RouteConstSharedPtr route;
    if (request_headers_ != nullptr) {
        //真正的RDS路由查找过程
        route = snapped_route_config->route(*request_headers_, stream_id_);
    }
    stream_info_.route_entry_ = route ? route->routeEntry() : nullptr;
    //设置目的路由缓存信息
    cached_route_ = std::move(route);
    ...
}
  
```

上面代码中的snapped_route_config_存放的就是路由相关的配置，其初始化信息如下：

```

snapped_route_config_(connection_manager.config_.routeConfigProvider().config())
  
```

routeConfigProvider()返回的是RdsRouteConfigProviderImpl, config函数返回的就是ConfigImpl, ConfigImpl就是上面讨论过的RDS树状分层结构, 维护最新的RDS路由配置。真正的路由查找过程在ConfigImpl的route函数里面。

```
RouteConstSharedPtr route(const Http::HeaderMap& headers, uint64_t random_value) const override {
    //route_matcher_是路由树状分层结构的入口
    return route_matcher_->route(headers, random_value);
}
```

RouteMatcher的route函数根据HTTP消息头信息, 查找匹配的VirtualHost, 进而通过分层查找, 获取相应的RouteEntry。

Envoy当前使用HTTP消息头的Host字段进行VirtualHost查找, 当前支持精确匹配、基于前缀的模糊匹配以及基于后缀的模糊匹配等多种查找方式。

```
RouteConstSharedPtr RouteMatcher::route(const Http::HeaderMap& headers,
                                         uint64_t random_value) const {
    const VirtualHostImpl* virtual_host = findVirtualHost(headers);
    if (virtual_host) {
        return virtual_host->getRouteFromEntries(headers, random_value);
    } else {
        return nullptr;
    }
}
```

VirtualHostImpl: : getRouteFromEntries负责VirtualHost内Route的选取, 对每一个当前支持的Route, 判断是否匹配。

```
RouteConstSharedPtr VirtualHostImpl::getRouteFromEntries(const Http::HeaderMap& headers, uint64_t random_v
...
//查找匹配的Route策略
for (const RouteEntryImplBaseConstSharedPtr& route : routes_) {
    RouteConstSharedPtr route_entry = route->matches(headers, random_value);
    if (nullptr != route_entry) {
        return route_entry;
    }
}
return nullptr;
}
```

Envoy当前支持前缀路由、路径路由以及正则路由等路由策略, 以前缀匹配为例:

```
RouteConstSharedPtr PrefixRouteEntryImpl::matches(const Http::HeaderMap& headers, uint64_t random_value) {
    if (RouteEntryImplBase::matchRoute(headers, random_value) &&
        (case_sensitive_?
StartsWith(headers.Path()->value().getStringView(), prefix_) : StartsWithIgnoreCase(headers.Path()->value(
        return clusterEntry(headers, random_value);
    }
    return nullptr;
}
```

匹配成功后, 调用clusterEntry获取路由目的集群的集群名信息。对于非weighted路由的场景, 直接获取当前信息返回即可; 对于weighted路由, 根据权重获取相应的集群节点。

```
RouteConstSharedPtr RouteEntryImplBase::clusterEntry(const Http::HeaderMap& headers, uint64_t random_value) {
    if (weighted_clusters_.empty()) {
        if (!cluster_name_.empty() || isDirectResponse()) {
            return shared_from_this();
        }
        ...
    }
    return WeightedClusterUtil::pickCluster(weighted_clusters_,
        total_cluster_weight_, random_value, true);
}
```

和上述路由匹配相比，路由获取的流程相对简单，首先查找路由缓存，如果存在，直接返回即可，如果不存在，则通过refreshCachedRoute进行路由刷新操作。

```
Router::RouteConstSharedPtr ConnectionManagerImpl::ActiveStreamFilterBase::route() {
    if (!parent_.cached_route_.has_value()) {
        parent_.refreshCachedRoute();
    }

    return parent_.cached_route_.value();
}
```

路由清除操作通过clearRouteCache进行，这里只是简单地将路由相关缓存清空即可，具体刷新逻辑在下一次路由获取时才会来触发。

```
void ConnectionManagerImpl::ActiveStreamFilterBase::clearRouteCache() {
    parent_.cached_route_ = absl::optional<Router::RouteConstSharedPtr>();
    parent_.cached_cluster_info_ = absl::optional<Upstream::ClusterInfoConstSharedPtr>();
}
```

7.6.4 HTTP过滤链处理

decodeHeaders函数中，首先会调用refreshCachedRoute进行路由查找和路由缓存刷新流程，refreshCachedRoute函数中，最重要的流程是通过snapped_route_config->route查找到的路由信息，并缓存起来，供本次请求中后续的各个HTTP插件使用。snapped_route_config->route实际调用的是ConfigImpl: route触发完整的路由匹配过程。

路由查找成功后通过createFilterChain创建新请求对应的HTTP网络链。Envoy当前已经支持很多HTTP扩展插件，具体都在extensions/filters/http/目录下，HTTP协议管理器http_connection_manager初始化时负责加载这些插件，HTTP扩展插件当前有很多，包含gzip、rbac、ratelimit和route等，这些插件的处理流程都差不多，差异点就是完成不同的具体功能。

```
void addStreamDecoderFilter(StreamDecoderFilterSharedPtr filter) override {
    addStreamDecoderFilterWorker(filter, false);
}
void ConnectionManagerImpl::ActiveStream::addStreamDecoderFilterWorker(
    StreamDecoderFilterSharedPtr filter, bool dual_filter) {
    ActiveStreamDecoderFilterPtr wrapper (new ActiveStreamDecoderFilter(*this, filter, dual_filter));
    filter->setDecoderFilterCallbacks(*wrapper);
    wrapper->moveIntoListBack(std::move(wrapper), decoder_filters_);
}

void Filter::setDecoderFilterCallbacks(Http::StreamDecoderFilterCallbacks& callbacks) {
    callbacks_ = &callbacks;
    buffer_limit_ = callbacks->decoderBufferLimit();
}
```

下面会以流量管理中最重要路由插件为例，调用createFilterChain创建HTTP过滤链时，会调用每个HTTP插件的createFilterFactoryFromProtoTyped函数。对于路由插件来说，对应的是RouterFilterConfig: createFilterFactoryFromProtoTyped，这里会通过addStreamDecoderFilter将ProdFilter加入到ConnectionManagerImpl对应的过滤链中，addStreamDecoderFilter中会创建ActiveStreamDecoderFilter，用于实现通用的HTTP解码逻辑，然后一方面将wrapper后的插件加入到decoder_filters_中，另一方面调用插件的setDecoderFilterCallbacks函数，将ActiveStreamDecoderFilter设置到ProdFilter的callbacks_字段中。callbacks_是ProdFilter中非常重要的一个字段，下面处理过程中会多次用到它。

回到decodeHeaders函数，创建完HTTP网络插件后，会调用ActiveStream中的另外一个decodeHeaders，这里会遍历HTTP网络链的每一个插件，触发相应的decodeHeaders处理流程，对于路由插件来说，就会调用ProdFilter路由插件的decodeHeaders流程。

```
Http::FilterHeadersStatus Filter::decodeHeaders(Http::HeaderMap& headers, bool end_stream) {
    downstream_headers_ = &headers;
    ...
    //获取路由目的Entry
    route_ = callbacks->route();
    ...
    route_entry_ = route->routeEntry();
    Upstream::ThreadLocalCluster* cluster = config_.cm_.get(route_entry->clusterName());
    cluster_ = cluster->info();
    //获取连接池信息
    Http::ConnectionPool::Instance* conn_pool = getConnPool();
    ...
    upstream_request_ = std::make_unique<UpstreamRequest>(*this, *conn_pool);
}
```

```
upstream_request_->encodeHeaders(end_stream);  
if (end_stream) {  
    onRequestComplete();  
}  
return Http::FilterHeadersStatus::StopIteration;  
}
```

在Filter: : decodeHeaders里依次会完成负载均衡、Upstream连接选取以及Upstream请求的编码工作，它在整个请求处理过程中充当中枢的角色。

7.6.5 负载均衡

负载均衡是高性能和高并发的基础，通过负载均衡，可以将流量均匀分布到集群的多个节点上，本节集中讨论Envoy负载均衡的设计和实现。

1. 优先级

Envoy集群管理也是以优先级为基础进行组织的，单个集群按照优先级划分为不同的节点集合，集群成员信息变化时也会按照优先级进行重新组织和整理。

负载均衡选取时，Envoy默认只在最高优先级的候选集中进行选取，出于高可用的考虑，一般都会指定比最高优先级小的可用优先级为备选，这样当最高优先级的候选集中的服务节点均不可用时，可以通过故障转移机制将流量转移到备选优先级对应的候选集中。

2. 恐慌阈值

负载均衡选取过程中，Envoy默认优先选择健康的节点，但由于故障等，一个集群的健康节点比例过低时，按照健康节点的选取策略，集群的全部流量会全部涌向为数不多的健康节点上，容易引发健康节点因为流量过大而雪崩的现象，导致整个集群全部不可用。

为了解决这个问题，Envoy引入了恐慌阈值的概念，默认的恐慌阈值是50%，当健康节点占有所有节点的比例下降到恐慌阈值时，选取节点时不再局限于健康节点范围内，而是在所有节点范围内选取。

3. 区域感知负载均衡

区域感知负载均衡，顾名思义，负载均衡时优先选取和当前区域相同的服务节点，区域感知特性是Envoy中所有负载均衡算法的基石。

hostSourceToUse是区域感知路由的核心实现，首先基于优先级选取初始候选集，默认采用选取健康节点的策略，如果触发恐慌阈值，则退化到选取当前所有节点；然后判断当前是否满足区域感知负载均衡的条件，如果不满足，则直接返回，否则通过区域感知路由算法获取合适的节点列表。综合来看，目标候选节点集合的选取条件包括优先级、区域和是否需要健康节点这几个核心因素。

```
ZoneAwareLoadBalancerBase::HostsSource
ZoneAwareLoadBalancerBase::hostSourceToUse(LoadBalancerContext* context) {
    //基于优先级选取初始候选集
    auto host_set_and_source = chooseHostSet(context);
    const auto host_availability = host_set_and_source.second;
    auto& host_set = host_set_and_source.first;
    HostsSource hosts_source;
    hosts_source.priority_ = host_set.priority();

    //判断是否触发恐慌阈值，如果触发，则将候选集扩大到所有节点
    if (per_priority_panic_[hosts_source.priority_] {
        stats_.lb_healthy_panic_.inc();
        hosts_source.source_type_ = HostsSource::SourceType::AllHosts;
        return hosts_source;
    }

    //区域选取
```

```

absl::optional<uint32_t> locality;
if (host_availability == HostAvailability::Degraded) {
    locality = host_set.chooseDegradedLocality();
} else {
    locality = host_set.chooseHealthyLocality();
}

// 基于区域的本地权重直接进行选择
if (locality.has_value()) {
    hosts_source.source_type_ = localitySourceType(host_availability);
    hosts_source.locality_index_ = locality.value();
    return hosts_source;
}
//无法进行区域路由，退化到选取所有的健康节点
if (per_priority_state_[host_set.priority()]->locality_routing_state_ ==
    LocalityRoutingState::NoLocalityRouting) {
    hosts_source.source_type_ = sourceType(host_availability);
    return hosts_source;
}

//没有打开区域路由特性开关，返回所有的健康节点
if (!runtime_.snapshot().featureEnabled(RuntimeZoneEnabled, routing_enabled_)) {
    hosts_source.source_type_ = sourceType(host_availability);
    return hosts_source;
}

//如果本地区域触发恐慌阈值，则返回所有的健康节点
if (isGlobalPanic(localHostSet())) {
    stats_.lb_local_cluster_not_ok_.inc();
    //如果本地Envoy处于全局恐慌状态，则不启动区域路由机制
    hosts_source.source_type_ = sourceType(host_availability);
    return hosts_source;
}

//进行实际的区域感知路由
hosts_source.source_type_ = localitySourceType(host_availability);
hosts_source.locality_index_ = tryChooseLocalLocalityHosts(host_set);
return hosts_source;
}

```

本地区域信息由PerPriorityState负责维护，具体包括路由策略以及路由到本地区域的百分比等，服务启动或者配置变更时，Envoy会通过regenerateLocalityRoutingStructures重新计算区域路由相关的信息，选取区域路由时，根据这些信息进行选择。

```

enum class LocalityRoutingState {
    //关闭本地区域路由
    NoLocalityRouting,
    //直接区域路由，所有流量均发往本地的服务节点
    LocalityDirect,
    //跨区域路由
    LocalityResidual
};

struct PerPriorityState {
    //路由到本地区域的百分比
    uint64_t local_percent_to_route_{};
    //当前区域路由状态
    LocalityRoutingState locality_routing_state_{LocalityRoutingState::NoLocalityRouting};
    //跨区域路由权重
    std::vector<uint64_t> residual_capacity_;
};

```

4.基于EDF调度的负载均衡

EdfLoadBalancerBase是最早截止时间优先（Earliest Deadline First，EDF）调度算法的基本实现，EDF是非常著名的实时调度算法之一，EDF调度器会优先从已就绪但还没有处理完毕的任务中选择最早截止时间的任务，并将执行该任务所需要的资源分配给它，由于

Envoy和下游服务之前采取的是非阻塞通信的方式，Envoy和每个下游服务节点都会有一个队列，保存当前已发送请求但未完成响应的请求消息，也可以称为Pending队列，对于Envoy EdfLoadBalancerBase负载均衡来说，会优先选择Pending队列最短的节点。

EDF调度算法也是Envoy负载均衡的一个基础算法，与一些在各个下游节点之间静态分配负载的算法相比，EDF根据下游在不同负载情况下的真实表现进行反馈和调整，相比静态分配来说，更加均衡，因此可以作为一些动态负载分配算法的基础。当前的轮询、最小请求算法均是在EDF调度算法的基础上构建的。

5.Envoy常见的负载均衡算法

Envoy当前内置多种负载均衡算法，具体包含轮询、随机、一致性哈希、最小请求等。其中RoundRobinLoadBalancer是轮询负载均衡实现，按照顺序选择健康的下游节点；RandomLoadBalancer为随机负载均衡实现，随机选择一个健康的下游节点，一般情况下，如果没有配置健康检查策略，随机算法会比轮询算法的均衡性好一些，随机选取可以避免在主机发生故障之后，对集群中主机的选择不均衡；RingHashLoadBalancer实现了一个基于ketama哈希算法的一致性哈希负载均衡算法；MaglevTable是参考Google开源的功能强大的负载均衡器Maglev实现的一个简单的Maglev算法。

7.6.6 连接池实现

在Filter: : UpstreamRequest: : encodeHeaders函数里，会调用conn_pool_.newStream创建新的连接，对于HTTP1来说，就是ConnPoolImpl: : newStream。

```
ConnectionPool::Cancellable* ConnPoolImpl::newStream(StreamDecoder& response_decoder, ConnectionPool::Cal
if (!ready_clients_.empty()) {
    ready_clients_.front()->moveBetweenLists(ready_clients_, busy_clients_);
    attachRequestToClient(*busy_clients_.front(), response_decoder, callbacks);
    return nullptr;
}
if (host_->cluster().resourceManager(priority_).pendingRequests().canCreate()) {
    bool can_create_connection =
    host_->cluster().resourceManager(priority_).connections().canCreate();
    //如果没有任何连接，在这里尝试进行创建
    if ((ready_clients_.size() == 0 && busy_clients_.size() == 0) || can_create_connection) {
        createNewConnection();
    }
    return newPendingRequest(response_decoder, callbacks);
} else {
    callbacks.onPoolFailure(ConnectionPool::PoolFailureReason::Overflow, nullptr);
    return nullptr;
}
}
void ConnPoolImpl::attachRequestToClient(ActiveClient& client, StreamDecoder& response_decoder, Connectio
client.stream_wrapper_ = std::make_unique<StreamWrapper>(response_decoder, client);
callbacks.onPoolReady(*client.stream_wrapper_, client.real_host_description_);
}
ConnPoolImpl::StreamWrapper::StreamWrapper(StreamDecoder& response_decoder, ActiveClient& parent)
: StreamEncoderWrapper(parent.codec_client_->newStream(*this)),
  StreamDecoderWrapper(response_decoder), parent_(parent) {
    StreamEncoderWrapper::inner_.getStream().addCallbacks(*this);
    ...
}
```

这里先检查连接资源是否满足创建新连接，如果当前已经有可用连接，可以直接拿来使用，调用attachRequestToClient将请求和连接进行绑定，否则判断是否满足连接创建条件，满足就调用createNewConnection创建新的Upstream连接。

首先看一下attachRequestToClient的逻辑，这里会创建StreamWrapper，负责管理Upstream请求和响应消息的编码、解码，attachRequestToClient的callbacks.onPoolReady实际调用到的是Filter: : UpstreamRequest: : onPoolReady，这里开始Upstream请求的实际编码工作。

```
void ConnPoolImpl::createNewConnection() {
    ActiveClientPtr client(new ActiveClient(*this));
    client->moveIntoList(std::move(client), busy_clients_);
}
ConnPoolImpl::ActiveClient::ActiveClient(ConnPoolImpl& parent)
: parent_(parent),
  connect_timer_(parent_.dispatcher_.createTimer([this]() -> void { onConnectTimeout(); })),
  remaining_requests_(parent_.host_->cluster().maxRequestsPerConnection()) {
    parent_.conn_connect_ms_ = std::make_unique<Stats::Timespan>(
        parent_.host_->cluster().stats().upstream_cx_connect_ms_, parent_.dispatcher_.timeSystem());
    Upstream::Host::CreateConnectionData data =
        parent_.host_->createConnection(parent_.dispatcher_, parent_.socket_options_, nullptr);
    real_host_description_ = data.host_description_;
    codec_client_ = parent_.createCodecClient(data);
    codec_client_->addConnectionCallbacks(*this);
    ...
}
```

然后看一下新连接的建立。通过createCodecClient创建编码客户端CodecClientProd, CodecClient构造函数中会注册对连接事件的关注, 当连接上有新数据时, 就会触发CodecClient的回调处理。

```
CodecClient::CodecClient(Type type, Network::ClientConnectionPtr&& connection,
                        Upstream::HostDescriptionConstSharedPtr host,
                        Event::Dispatcher& dispatcher)
    : type_(type), connection_(std::move(connection)), host_(host),
      idle_timeout_(host->cluster().idleTimeout()) {
    connection_->detectEarlyCloseWhenReadDisabled(false);
    connection_->addConnectionCallbacks(*this);
    connection_->addReadFilter(Network::ReadFilterSharedPtr{new CodecReadFilter(*this)});
    connection_->connect();
    if (idle_timeout_) {
        idle_timer_ = dispatcher.createTimer([this]() -> void { onIdleTimeout(); });
        enableIdleTimer();
    }
    //统一设置NODELAY TCP选项, 理论上该选项需要支持可配置
    connection_->noDelay(true);
}
```

ConnPoolImpl: : ActiveClient的构造函数中, 会调用parent_.host_->createConnection进行ConnectionImpl创建, 同时调用createCodecClient创建codec_client, 并增加该客户端连接的事件监听。

7.7 本章小结

本章从Istio的数据流的入口开始，详细剖析了监听管理、连接管理以及网络缓冲区管理的机制，同时针对HTTP协议和Thrift协议，从多个维度分析了请求解析、插件支持、路由转发等设计和实现机理。

第8章 Istio微服务治理

链路的可观测性是服务治理非常重要的组成部分。基于链路可观测性，可以对系统的状态有全面的把控，也可以对链路进行相应的治理和控制。

本章接下来聚焦Istio链路的观测和治理，详细讨论Istio链路可观测和可治理架构的设计与实现。

8.1 链路稳定性治理

8.1.1 超时机制

超时重试是稳定性保障中非常基础、重要的一环，应用没有设置超时时间或者超时时间设置得不正确，都有可能导致请求长时间得不到响应，长期累积的效应会导致一系列的连锁反应，严重时甚至会导致系统雪崩。

同步阻塞通信场景下，可以直接通过socket选项SO_SNDTIMEO和SO_RCVTIMEO来分别设置读写超时。Envoy属于典型的异步通信场景，I/O访问本来就是无阻塞返回，I/O就绪事件通过Epoll等I/O多路复用机制触发，这种场景下的超时时间一般由专门的定时器来触发。

以HTTP请求为例，当前支持请求级别的全局超时（global timeout）和单次超时（per timeout）两个层次的超时时间，比如单次超时为50ms，全局超时为100ms，在访问第一个节点超时后，还可以重试其他节点。

下面重点看全局超时和单次超时的时机与触发时的具体措施，Envoy在和上游成功建立连接后会回调UpstreamRequest: : onPoolReady，这里通过setupPerTryTimeout设置单次请求超时时间，设置方式就是建立超时时间为per_try_timeout_的定时器，定时器到期后，定时触发函数为UpstreamRequest: : onPerTryTimeout，这里只是调用parent_.onPerTryTimeout。

```
void Filter::UpstreamRequest::setupPerTryTimeout() {
    if (parent_.timeout_.per_try_timeout_.count() > 0) {
        per_try_timeout_ = parent_.callbacks_>dispatcher().createTimer([this]() -> void { onPerTryTimeout();
        per_try_timeout_>enableTimer(parent_.timeout_.per_try_timeout_);
    }
}
```

Filter: : onPerTryTimeout实现具体的请求单次超时逻辑，具体如下：

```
void Filter::onPerTryTimeout() {
    updateOutlierDetection(timeout_response_code_);
    if (maybeRetryReset(Http::StreamResetReason::LocalReset)) {
        return;
    }
    onUpstreamTimeoutAbort(StreamInfo::ResponseFlag::UpstreamRequestTimeout);
}
```

上述Filter: : onPerTryTimeout主要做了如下3件事情。

1) 更新当前Upstream节点的OutlierDetection统计，该Upstream节点的超时或者失败达到一定阈值时，OutlierDetection会将该节点进行熔断，熔断之后短时间内不再选取它，且它只能依赖健康检查机制才能恢复。

2) 重试判断，判断当前是否满足重试条件，如果符合，则进入重试流程并返回。

3) 如果不满足重试逻辑, 则构造Local消息, 返回给Downstream。

至于全局超时, 是在Upstream请求准备就绪的时候设置, 具体见Filter: : onRequestComplete, 定时器超时时间为global_timeout_, 定期触发函数为onResponseTimeout。

```
void Filter::onRequestComplete() {  
    ...  
    if (upstream_request_) {  
        if (timeout_.global_timeout_.count() > 0) {  
            response_timeout_ = dispatcher.createTimer([this]() -> void { onResponseTimeout(); });  
            response_timeout_>enableTimer(timeout_.global_timeout_);  
        }  
    }  
}
```

全局超时触发函数Filter: : onResponseTimeout和单次超时触发函数Filter: : onPerTry-Timeout的处理流程整体差不多, 区别主要是单次超时会判断当前是否需要重试, 全局超时则不需要这个逻辑。

```
void Filter::onResponseTimeout() {  
    ...  
    if (upstream_request_) {  
        upstream_request_>resetStream();  
    }  
    updateOutlierDetection(timeout_.response_code_);  
    onUpstreamTimeoutAbort(UpstreamRequestTimeout);  
}
```

8.1.2 重试机制和重试策略

重试也是保障业务可用性的重要方式，在请求处理过程中，经常会遇到诸如网络抖动、服务瞬时处理慢等现象，超时后采取一些必要的重试策略可以更好地保障服务可用性指标的达成。

1.重试场景

对于Envoy来说，重试有3个重要场景或时机。

1) **Upstream连接失败**：通过负载均衡选取到Upstream节点后，和Upstream节点建立连接失败，这时需要重试其他节点。

2) **Upstream返回结果出错**：Upstream连接没有问题，但请求处理结果不符合预期，对于一些结果场景可以通过重试来解决。

3) **Upstream处理超时**：超时的场景在8.1.1节已经具体讨论过，访问Upstream单次超时后，会通过maybeRetryReset判断当时是否可以重试。

对于Upstream连接失败和超时的场景，需要通过maybeRetryReset判断是否需要重试；对于Upstream返回结果出错的场景，需要通过shouldRetryHeaders进行重试逻辑的判断。

2.重试退避策略

Envoy在判断出当前需要重试时，并不是立即进行重试，而是通过JitteredBackOffStrategy实现一个简单的重试退避策略。首先通过开关变量设置当前的重试基准值base，然后基于基准值创建JitteredBackOffStrategy对象，具体如下：

```
const uint32_t base = runtime_.snapshot().getInteger("upstream.base_retry_backoff_ms", 25);
backoff_strategy_ = std::make_unique<JitteredBackOffStrategy>(base, base * 10, random_);
```

JitteredBackOffStrategy基于指数退避的策略，通过随机值在指数max和max_interval之间选取最小值，保证重试时间的退避性和离散性，具体如下：

```
JitteredBackOffStrategy::JitteredBackOffStrategy(uint64_t base_interval, uint64_t max_interval, Runtime:::
: base_interval_(base_interval), max_interval_(max_interval), random_(random) {
}

uint64_t JitteredBackOffStrategy::nextBackOffMs() {
    const uint64_t multiplier = (1 << current_retry_) - 1;
    const uint64_t base_backoff = multiplier * base_interval_;
    if (base_backoff <= max_interval_) {
        current_retry_++;
    }
    return std::min(random_.random() % base_backoff, max_interval_);
}

void JitteredBackOffStrategy::reset() { current_retry_ = 1; }
```

3.重试节点选取策略

Envoy在重试时，当前支持自定义的节点选取插件策略，可以通过配置指定当前的重试插件。当前支持previous_hosts和previous_priorities两种重试策略，分别用于重试时根据当前节点和优先级情况选取重试节点。

4.重试执行流程

Envoy的重试机制和策略通过RetryStateImpl对象进行管理，因此在Upstream请求开始前，通过createRetryState创建RetryStateImpl对象，这里主要根据路由配置对Retry-StateImpl进行初始化，设置相应的重试策略。

```
Http::FilterHeadersStatus Filter::decodeHeaders(Http::HeaderMap& headers, bool end_stream) {
    ...
    retry_state_ = createRetryState(route_entry->retryPolicy(), headers, *cluster_, config_.runtime_,
    config_.random_, callbacks->dispatcher(), route_entry->priority());
    ...
}
```

下面以Upstream连接失败后的重试为例，介绍重试的触发和执行：连接失败后，通过maybeRetryReset判断当前是否需要重试，这里有两个重要函数，即shouldRetryReset和doRetry，前者进行重试逻辑的判断，后者进行重试的触发执行。

```
bool Filter::maybeRetryReset(Http::StreamResetReason reset_reason) {
    //如果已经返回响应消息给下游节点，则不需要重试
    if (downstream_response_started_ || !retry_state_) {
        return false;
    }
    //重试节点选取策略
    Upstream::HostDescriptionConstSharedPtr upstream_host;
    if (upstream_request_) {
        upstream_host = upstream_request_->upstream_host_;
    }
    if (upstream_host != nullptr) {
        retry_state_->onHostAttempted(upstream_host);
    }
    const RetryStatus retry_status = retry_state_->shouldRetryReset(reset_reason, [this]() -> void { doRetr;
    if (retry_status == RetryStatus::Yes && setupRetry(true)) {
        return true;
    }
    return false;
}
}
```

shouldRetryReset的重试判断分为两个维度。一个是重试策略的判断，Envoy启动时会配置相应的重试策略，比如资源不足相关的失败，不需要进行重试，而HTTP服务端返回5xx错误码的场景，则需要重试等。重试判断的另一个维度是重试次数以及重试相关资源的判断，具体通过RetryStateImpl::shouldRetry进行，重试判断通过后，并不会立即进行重试，而是通过enableBackoffTimer启动一个重试定时器，具体的重试时机和重试触发由enableBackoffTimer负责。

```
RetryStatus RetryStateImpl::shouldRetry(bool would_retry, DoRetryCallback callback) {
    if (retries_remaining_ == 0) {
        return RetryStatus::NoRetryLimitExceeded;
    }

    retries_remaining_--;
    if (!would_retry) {
        return RetryStatus::No;
    }
    ...
}
```

```
callback_ = callback;
enableBackoffTimer();
return RetryStatus::Yes;
}
```

RetryStateImpl::enableBackoffTimer（见下面代码）通过上述讨论的重试退避策略获取重试的具体时间，然后基于这个时间设置定时器，定时器到期后，触发实际的重试操作，具体的重试操作之前一直通过callback进行设置，就是doRetry。

```
void RetryStateImpl::enableBackoffTimer() {
    if (!retry_timer_) {
        retry_timer_ = dispatcher_.createTimer([this]() -> void { callback_(); });
    }
    //这里使用的是完全jittered exponential回退算法
    retry_timer_->enableTimer(std::chrono::milliseconds(backoff_strategy-> nextBackOffMs()));
}
```

在doRetry中，创建相应的连接，连接建立成功后对请求进行编码处理和发送，走一遍和重试前的Upstream发送类似的完整流程。

8.1.3 节点熔断和健康检查机制

当Upstream节点失败率较高时，Envoy支持节点熔断机制：通过一定的判断策略，将故障节点从负载均衡的候选节点集中剔除，保证业务的正常路由和转发不会受到故障节点的影响，Envoy的节点熔断机制由outlier detection负责完成。

对于Envoy通信来说，当通信异常或失败的时候，需要通知outlier detection进行统计，典型的通知机制如下：

```
upstream_host->outlierDetector().putHttpStatusCode(enumToInt(code));
```

outlier detection的putHttpStatusCode会对HTTP错误码进行归类判断，比如当错误码为5××时，会通知主线程进行当前节点OutlierEjectionType: : CONSECUTIVE_5XX错误码的累加。各个工作线程若在进行流量转发时遇到请求失败的场景，均会将相应的节点和失败类型通知主线程进行处理。主线程的DetectorImpl: : ejectHost函数负责具体的熔断判断逻辑。

```
void DetectorImpl::ejectHost(HostSharedPtr host,
                             envoy::data::cluster::v2alpha::OutlierEjectionType type) {
    ...
    if (ejected_percent < max_ejection_percent) {
        //熔断判断
        if (enforceEjection(type)) {
            host_monitors_[host]->eject(time_source_.monotonicTime());
            runCallbacks(host);
            ...
            if (event_logger_) {
                event_logger_->logEject(host, *this, type, true);
            }
        }
    }
    ...
}
```

在DetectorImpl: : ejectHost中判断当前是否达到熔断阈值，如果需要熔断，则设置当前主机的熔断状态，同时通过runCallbacks将熔断时间通知给关心节点熔断的模块。

Envoy提供相应的event logger机制，将所有的熔断事件记录下来，供后续统计分析、性能分析、问题追查等使用。

当节点进入熔断状态后，会从负载均衡的候选节点集中剔除掉一定时间，这段时间可以称为熔断冷却期。需要有相应的机制，定时探测节点的可用情况，如果节点恢复，便将节点从不可用恢复为可用状态。

Envoy当前提供被动健康检查和主动健康检查两种健康检查机制。被动健康检查是等熔断冷却期结束后，直接将节点放回可用节点，供负载均衡使用；主动检查是通过相应的探测流量对故障节点进行探测，当探测到故障节点当前可用时，才会将节点从故障状态调整到可用状态。

```
void ClusterImplBase::setOutlierDetector(const Outlier::DetectorSharedPtr& outlier_detector) {
    if (!outlier_detector) {
        return;
    }
    outlier_detector_ = outlier_detector;
    outlier_detector_>addChangedStateCb([this](HostSharedPtr) -> void { reloadHealthyHosts(); });
}
```

Envoy集群在设置outlier detector的同时，也设置节点熔断状态变更通知函数，当节点熔断时，会触发reloadHealthyHosts函数的调用。在reloadHealthyHosts函数中会重新整理集群里各个优先级的节点信息，然后通过TLS机制通知各个工作线程进行更新，这样工作线程在负载均衡时，就可以使用最新的节点状态信息。

8.1.4 资源限制机制

任何系统的资源都是有一定限度的，如果使用超过一定限度，就会出现各种各样的问题，影响系统的整体可用性，因此为了保障系统的可用性，需要有相应的资源管理机制，保障关键资源的合理使用。

Envoy针对Upstream集群的一些关键资源，比如Upstream集群最大连接数、最大请求数等，通过resource manager进行管理。

resource manager进行资源管理的方式很简单，在资源创建时指定资源的配额限制，同时提供相应的资源更新和配额检查接口。当使用和归还资源时更新相应的资源计数；通过资源配额检查接口判断资源当前是否还有富余。

为了实现资源管理的灵活性，resource manager针对每个资源设置一个开关变量key，从而通过开关特性机制动态修改资源的当前值。

另外，针对每个资源设置一个stats项，当资源超过限额时设置stats状态为超出配额，这样外界通过HTTP管理接口实时查看Envoy运行状态时，也可以查看当前所有的资源状态，比如是否超出资源配额等。

8.1.5 全局限流机制

尽管Envoy提供了完善的资源限制机制，可以很大程度上保证顺利访问Upstream的资源 and 应对压力情况，但对于有大量客户端的服务端程序来说，从任何一个客户端都很难确切地了解服务端程序的真实压力情况，并进行准确的资源限制，因此需要有一个从服务端角度出发的全局视角，对客户端/服务器之间的通信和链路情况进行限制与管控，全局限流就是解决这个问题一个很好的机制。

Envoy当前支持网络层面和HTTP层面两种限流方式。网络层面的限流可以针对需要限流的连接设置网络限流过滤器，这样当新连接创建时，Envoy会通过gRPC远程访问全局限流服务判断是否允许访问，如果不允许，限流插件会在新数据到来时直接返回出错提示。HTTP层面的限流是针对请求级别的，每个HTTP请求会有相应的路由，路由访问前，通过gRPC远程访问全局限流服务判断是否允许访问，如果不允许，则返回失败。

8.2 链路可观测性

8.2.1 Envoy分布式跟踪支持

虽然Istio一直讲服务治理和服务的可观测性对业务代码零侵入，但是要获得一个质量良好的调用链，应用程序还是要配合做些事情，比如需要在类似HTTP头的地方传递这些Span信息，这样这些Span才能被正确地链接成一个trace。因此要求应用程序必须收集和传递这些trace相关的头并传递出去。

下面以HTTP协议的分布式跟踪支持为例，详细讨论Envoy分布式跟踪的支持。

1.跟踪插件初始化

实现层面，Envoy分布式跟踪特性的初始化工作由MainImpl::initializeTracers负责，这里先判断是否为HTTP协议，然后使用配置文件中指定的跟踪插件名，调用对应插件factory的HTTP Tracer创建函数，完成HTTP Tracer的初始化。

```
void MainImpl::initializeTracers(const envoy::config::trace::v2::Tracing& configuration, Instance& server) {
    if (!configuration.has_http()) {
        http_tracer_ = std::make_unique<Tracing::HttpNullTracer>();
        return;
    }
    //初始化跟踪器
    std::string type = configuration.http().name();
    auto& factory = Config::Utility::getAndCheckFactory<TracerFactory>(type);
    ProtobufTypes::MessagePtr message =
        Config::Utility::translateToFactoryConfig(configuration.http(), factory);
    http_tracer_ = factory.createHttpTracer(*message, server);
}
```

Envoy当前支持众多分布式跟踪插件，以最常见的分布式跟踪系统Zipkin为例，createHttpTracerTyped函数中会创建对应的跟踪实现Zipkin::Driver。

```
Tracing::HttpTracerPtr ZipkinTracerFactory::createHttpTracerTyped(
    const envoy::config::trace::v2::ZipkinConfig& proto_config, Server::Instance& server) {
    Tracing::DriverPtr zipkin_driver = std::make_unique<Zipkin::Driver>(
        proto_config, server.clusterManager(), server.stats(), server.threadLocal(), server.runtime(),
        server.localInfo(), server.random(), server.timeSystem());
    //基于跟踪插件实例Driver创建通用的HttpTracer管理对象HttpTracerImpl
    return std::make_unique<Tracing::HttpTracerImpl>(
        std::move(zipkin_driver), server.localInfo());
}
```

构造Driver函数过程中，创建两个重要的数据结构Tracer和ReporterImpl，分别负责Span信息构造和Span信息输出（下面会详细讨论）。

2.跟踪数据构建

Envoy接收到HTTP请求后，在对HTTP请求头解码完成后，调用ActiveStream::traceRequest进行Trace处理，实际调用的是HttpTracerImpl的startSpan函数。

```
void ConnectionManagerImpl::ActiveStream::traceRequest() {
    Tracing::Decision tracing_decision = HttpTracerUtility::isTracing(
        stream_info_, *request_headers_);
    active_span_ = connection_manager_.tracer().startSpan(*this,
        *request_headers_, stream_info_, tracing_decision);
    ...
}
```

HttpTracerImpl是HTTP协议相关的分布式跟踪实现，这里主要进行一些HTTP协议相关的适配工作，然后调用实际的插件实现Driver：：startSpan。

```
Tracing::SpanPtr Driver::startSpan(const Tracing::Config& config, Http::HeaderMap& request_headers,
    const std::string&, SystemTime start_time,
    const Tracing::Decision tracing_decision) {
    Tracer& tracer = *tls_->getTyped<TlsTracer>().tracer_;
    SpanPtr new_zipkin_span;
    SpanContextExtractor extractor(request_headers);
    bool sampled{extractor.extractSampled(tracing_decision)};
    auto ret_span_context = extractor.extractSpanContext(sampled);
    if (!ret_span_context.second) {
        //创建Zipkin的根Span
        new_zipkin_span = tracer.startSpan(config,
            request_headers.Host()->value().c_str(), start_time);
        new_zipkin_span->setSampled(sampled);
    } else {
        new_zipkin_span = tracer.startSpan(config,
            request_headers.Host()->value().c_str(),
            start_time, ret_span_context.first);
    }
    ZipkinSpanPtr active_span(new ZipkinSpan(*new_zipkin_span, tracer));
    return std::move(active_span);
}
```

首先会通过SpanContextExtractor从HTTP头信息中获取SpanID信息，如果获取不到，说明请求来自根节点，否则将HTTP头信息中的SpanID作为当前请求的父SpanID，构造当前节点对应的Span信息，具体包含请求的TraceID、当前节点的SpanID、当前节点对应的父SpanID以及当前节点的Annotation信息。

请求转发给下游之前，将当前Span信息通过injectContext注入到请求头中，一方面供Trace日志输出使用，另一方面考虑转发请求到Upstream时也需要当前节点对应的Span信息。

3.跟踪数据输出

HTTP请求处理完成后，在ActiveStream的析构函数中触发trace信息的输出，在finalizeSpan函数中设置HTTP请求头相关字段到trace信息，然后调用插件层的输出函数。

```
ConnectionManagerImpl::ActiveStream::~ActiveStream() {
    ...
    // 请求结束，输出请求对应的trace信息
    if (active_span_) {
        Tracing::HttpTracerUtility::finalizeSpan(*active_span_,
            request_headers_.get(), stream_info_, *this);
    }
    ...
}
```

对于Zipkin来说，trace信息输出前分别针对请求和响应消息，设置相应的Annotation信息，随后通过ReporterImpl：：reportSpan进行实际的输出工作。

```

void ZipkinSpan::finishSpan() { span_.finish(); }

void Span::finish() {
    SpanContext context(*this);
    if (annotations_[0].value() == ZipkinCoreConstants::get().SERVER_RECV) {
        // 构建接收消息对应的Annotation
        Annotation ss;
        ss.setEndpoint(annotations_[0].endpoint());
        ss.setTimestamp(std::chrono::duration_cast<std::chrono::microseconds>(
            time_source_.systemTime().time_since_epoch())
            .count());
        ss.setValue(ZipkinCoreConstants::get().SERVER_SEND);
        annotations_.push_back(std::move(ss));
    } else if (annotations_[0].value() ==
        ZipkinCoreConstants::get().CLIENT_SEND) {
        // 构建发送消息对应的Annotation
        Annotation cr;
        const uint64_t stop_timestamp = std::chrono::duration_cast<std::chrono::microseconds>(
            time_source_.systemTime().time_since_epoch())
            .count();
        cr.setEndpoint(annotations_[0].endpoint());
        cr.setTimestamp(stop_timestamp);
        cr.setValue(ZipkinCoreConstants::get().CLIENT_RECV);
        annotations_.push_back(std::move(cr));
    }
    if (auto t = tracer()) {
        t->reportSpan(std::move(*this));
    }
}

```

ReporterImpl不会对trace信息进行立即输出，而是将其加入到缓存中，满足一定的flush策略时才进行flush输出。

```

void ReporterImpl::reportSpan(const Span& span) {
    span_buffer_.addSpan(span);
    const uint64_t min_flush_spans =
        driver_.runtime().snapshot().getInteger("tracing.zipkin.min_flush_spans", 5U);
    if (span_buffer_.pendingSpans() == min_flush_spans) {
        flushSpans();
    }
}

```

当前有两个条件，一个是缓存的量达到一定限度，一个是缓存的时间达到一定时间，ReporterImpl启动时创建一个flush定时器，专门用于flush操作的定时触发。

```

ReporterImpl::ReporterImpl(Driver& driver, Event::Dispatcher& dispatcher,
    const std::string& collector_endpoint)
    : driver_(driver), collector_endpoint_(collector_endpoint) {
    flush_timer_ = dispatcher.createTimer([this]() -> void {
        driver_.tracerStats().timer_flushed_.inc();
        flushSpans();
        enableTimer();
    });
    enableTimer();
}

```

满足输出条件时，对trace消息进行拼装，然后调用HTTP接口，将trace信息发送到远端的Zipkin存储中心。

```

void ReporterImpl::flushSpans() {
    if (span_buffer_.pendingSpans()) {

```

```
//trace信息拼装
driver_.tracerStats().spans_sent_.add(span_buffer_.pendingSpans());
const std::string request_body = span_buffer_.toStringifiedJsonArray();
Http::MessagePtr message(new Http::RequestMessageImpl());
...
Buffer::InstancePtr body(new Buffer::OwnedImpl());
body->add(request_body);
message->body() = std::move(body);
// HTTP异步发送
driver_.clusterManager()
    .httpClientForCluster(driver_.cluster()->name())
    .send(std::move(message), *this, timeout);
}
}
```

8.2.2 Envoy Metric支持

Metric用于对Envoy的核心指标数据进行聚合，具体由stats子系统实现，Envoy stats包括3类核心统计数据。

(1) Counter（计数器）

计数器用于对单调递增的统计项进行累加统计，比如请求总的QPS。

(2) Gauge

Gauge也是一种很常见的计量，用于记录样本序列。和Counter相比，Gauge可以增加和减少，比如待处理队列中的任务个数、连接池空闲连接个数等，Counter只会增加不会减少，有一个单调变化的特征。

(3) Histogram（直方图）

Histogram用于对一定范围内的样本进行统计分析，最终产生各个分位对应的统计值。分位是Histogram中常见的一个概念，比如50分位耗时A和平均耗时B是两个不同的概念。50分位耗时指50%的请求在A时间范围内返回；平均耗时指所有请求的平均处理时间，很多场景下分位统计比平均统计更能反映数据的分布特性。比如对于请求处理耗时，通过Histogram可以得出50分位的耗时统计、99分位的耗时统计等。

在Envoy初始化流程MainCommonBase函数中，会创建ThreadLocalStoreImpl，作为Envoy stats子系统的stats_store_。

为了简化stats统计项的使用，stats提供了一组宏对Counter、Gauge和Histogram这几类统计项进行操作。下面以Envoy监听器统计为例，介绍stats的使用方式。

```
//stats宏定义
#define GENERATE_COUNTER_STRUCT(NAME) Stats::Counter& NAME##_;
#define GENERATE_GAUGE_STRUCT(NAME) Stats::Gauge& NAME##_;
#define GENERATE_HISTOGRAM_STRUCT(NAME) Stats::Histogram& NAME##_;

#define FINISH_STAT_DECL_(X) + std::string(#X)),

#define POOL_COUNTER_PREFIX(POOL, PREFIX) (POOL).counter(PREFIX FINISH_STAT_DECL_
#define POOL_GAUGE_PREFIX(POOL, PREFIX) (POOL).gauge(PREFIX FINISH_STAT_DECL_
#define POOL_HISTOGRAM_PREFIX(POOL, PREFIX) (POOL).histogram(PREFIX FINISH_STAT_DECL_

#define POOL_COUNTER(POOL) POOL_COUNTER_PREFIX(POOL, "")
#define POOL_GAUGE(POOL) POOL_GAUGE_PREFIX(POOL, "")
#define POOL_HISTOGRAM(POOL) POOL_HISTOGRAM_PREFIX(POOL, "")

#define ALL_LISTENER_MANAGER_STATS(COUNTER, GAUGE) \
    COUNTER(listener_added) \
    COUNTER(listener_modified) \
    GAUGE (total_listeners_warming) \

struct ListenerManagerStats {
    ALL_LISTENER_MANAGER_STATS(GENERATE_COUNTER_STRUCT, GENERATE_GAUGE_STRUCT)
};

ListenerManagerStats ListenerManagerImpl::generateStats(Stats::Scope& scope) {
    const std::string final_prefix = "listener_manager.";
    return {ALL_LISTENER_MANAGER_STATS(POOL_COUNTER_PREFIX(scope, final_prefix), POOL_GAUGE_PREFIX(scope, f
```

```

}
ListenerManagerImpl::ListenerManagerImpl(Instance& server,
    ListenerComponentFactory& listener_factory,
    WorkerFactory& worker_factory, TimeSource& time_source)
    stats_(generateStats(server.stats())) {
    ...
}
bool ListenerManagerImpl::addOrUpdateListener(const envoy::api::v2::Listener& config, const std::string&
    ...
    if (added) {
        stats_.listener_added_.inc();
    } else {
        stats_.listener_modified_.inc();
    }
    ...
}

```

使用stats前会通过ListenerManagerImpl:: generateStats创建stats_数据结构，调用generateStats时传入的scope来表示一组具有相似行为的度量指标，比如可以同时scope范围内的度量指标进行删除。一般按照业务划分scope，通过scope name标识不同的scope，比如Envoy监听器使用一个scope，HTTP请求处理器使用另一个scope。scope通过server_.stats().createScope创建，实际上使用的是ThreadLocalStoreImpl:: createScope，实现如下：

```

ScopePtr ThreadLocalStoreImpl::createScope(const std::string& name) {
    std::unique_ptr<ScopeImpl> new_scope(new ScopeImpl(*this, name));
    Thread::LockGuard lock(lock_);
    scopes_.emplace(new_scope.get());
    return std::move(new_scope);
}

```

generateStats里面使用了两个重要的宏：一是POOL_COUNTER_PREFIX（scope，final_prefix），其展开后为scope.counter（“final_prefix”）；另一个是ALL_LISTENER_MANAGER_STATS。这两个一起展开后完整的ListenerManagerStats信息如下：

```

{
    (scope).counter("listener_manager." + std::string("listener_added")),
    (scope).counter("listener_manager." + std::string("listener_modified")),
    (scope).gauge("listener_manager." + std::string("total_listeners_warming ")),
};

```

stats_对应的是ListenerManagerStats，宏展开后信息如下：

```

struct ListenerManagerStats {
    Stats::Counter& listener_added_;
    Stats::Counter& listener_modified_;
    Stats::Gauge& total_listeners_warming_
};

```

通过stats_（generateStats（server.stats()）将上述generateStats产生的ListenerManagerStats信息赋给stats_使用，业务通过stats_.listener_added_.inc()操作stats信息时，实际调用的是（scope）.counter（"listener_manager."+std::string（"listener_added"）），实现如下：

```

Counter& ThreadLocalStoreImpl::ScopeImpl::counter(const std::string& name) {
    if (parent_.rejects(final_name)) {
        return null_counter_;
    }
    StatMap<CounterSharedPtr>* tls_cache = nullptr;
    if (!parent_.shutting_down_ && parent_.tls_) {
        tls_cache = &parent_.tls_->getTyped<TlsCache>().scope_cache_[this->scope_id_].counters_;
    }

    return safeMakeStat<Counter>(<
        final_name, central_cache_.counters_,
        [](&StatDataAllocator& allocator, absl::string_view name, std::string&& tag_extracted_name,
            std::vector<Tag>&& tags) -> CounterSharedPtr {
            return allocator.makeCounter(name, std::move(tag_extracted_name), std::move(tags));
        },
        tls_cache);
}

```

stats通过TLS Cache减少不同线程同时操作一个度量指标时的锁开销，当第一次对某个度量指标进行操作时，需要通过allocator.makeCounter分配度量指标对应的资源。

```

template <class StatData>
CounterSharedPtr StatDataAllocatorImpl<StatData>::makeCounter(absl::string_view name,
std::string&& tag_extracted_name, std::vector<Tag>&& tags) {
    StatData* data = alloc(name);
    if (data == nullptr) {
        return nullptr;
    }
    return std::make_shared<CounterImpl<StatData>>(*data, *this, std::move(tag_extracted_name), std::move(t
}

```

Envoy stats单个指标数据由StatData负责管理，分配时需要分配StatData对应的资源，对StatData主要字段的修改均采用原子操作，确保在多线程环境下的操作安全。

Envoy stats度量指标的实现比较简单，Count、Gauge和Histogram分别由CounterImpl、GaugeImpl和HistogramImpl实现。

Envoy stats可以配置多个sink，通过sink机制，可以定时将当前统计信息同步到第三方基础设施，Envoy初始化阶段会根据配置，调用stats_store.addSink注册相应的sink实现，同时会启动flush定时器stat_flush_timer_进行定时flush，定时器的回调函数为flushStats。flushStats函数中，需要重点关注的是flushMetricsToSinks，它会循环调用各个sink插件的flush函数。

Envoy当前支持多种sink插件，它们定义在/source/extensions/stat_sinks目录下，以metrics_service为例，MetricsServiceSink：flush会分别拼装Count、Gauge和Histogram flush信息，最后通过gRPC接口发送出去。

8.2.3 Envoy Log支持

Envoy Log当前只有对HTTP协议有着完善的支持，内置对HTTP访问日志的支持。HTTP访问日志特性主要由日志过滤、日志格式化和日志输出3部分组成。HTTP日志输出时，首先通过相应的日志过滤模块过滤获取匹配的日志，然后通过日志格式化模块对日志进行格式化和标准化，最后将格式化日志按照日志输出模块的要求，输出到相应的位置，方便后续的日志分析和查询。

HTTP网络协议处理插件HttpConnectionManager负责HTTP协议的解析和路由转发处理，会解析HTTP启动配置，用于HTTP协议的处理。针对access_log特性，调用access_log的工厂方法，根据access_log配置，创建相应的access_log实例。

access_log实例构建具体由AccessLogFactory::fromProto负责完成。fromProto会先根据配置创建相应的日志过滤器，然后调用factory.createAccessLogInstance创建相应的access_log实例。

```
InstanceSharedPtr
AccessLogFactory::fromProto(const xxx::AccessLog& config,
                           Server::Configuration::FactoryContext& context) {
    FilterPtr filter;
    if (config.has_filter()) {
        filter = FilterFactory::fromProto(config.filter(), context.runtime(), context.random());
    }
    auto& factory =
        Config::Utility::getAndCheckFactory<Server::Configuration::AccessLogInstanceFactory>(config.name())
    ProtobufTypes::MessagePtr message = Config::Utility::translateToFactoryConfig(
        config, context.messageValidationVisitor(), factory);
    return factory.createAccessLogInstance(*message, std::move(filter), context);
}
```

access_loggers当前支持file和http_grpc两种access_log插件实例。file插件是通过异步I/O的方式，按照一定的日志格式，从HTTP请求消息和响应消息中提取需要的信息，完成access_log日志的收集，日志写入过程中不会阻塞工作线程；http_grpc插件方式是将日志通过gRPC协议发送到远端的日志分析和处理平台，Envoy定义了标准的access_log gRPC接口，只需要按照access_log gRPC接口进行定制实现就可以增加新的access_log插件。

请求处理完成后，在ConnectionManagerImpl::ActiveStream::~~ActiveStream里会调用access_log插件的log接口，进行access_log日志的输出，以file插件方式为例，access_log日志的获取和输出处理如下：

```
void FileAccessLog::log(const Http::HeaderMap* request_headers,
                      const Http::HeaderMap* response_headers,
                      const Http::HeaderMap* response_trailers,
                      const StreamInfo::StreamInfo& stream_info) {
    static Http::HeaderMapImpl empty_headers;
    if (!request_headers) {
        request_headers = &empty_headers;
    }
    if (!response_headers) {
        response_headers = &empty_headers;
    }
    if (!response_trailers) {
        response_trailers = &empty_headers;
    }
}
```

```
// 1.日志过滤
if (filter_) {
    if (!filter_>evaluate(stream_info, *request_headers, *response_headers, *response_trailers)) {
        return;
    }
}

// 3.日志输出
log_file_>write(
    // 2. 日志格式化
    formatter_>format(*request_headers, *response_headers, *response_trailers, stream_info));
}
```

从上面的处理可以看出，首先根据日志过滤插件进行日志过滤，如果没有被过滤掉，按照日志格式化标准进行格式化，最后将日志进行异步非阻塞输出。

8.3 本章小结

服务治理是Istio中的关键组成部分，本章围绕稳定性治理和可观测性展开，Istio拥有强大的链路稳定性治理能力，同时基于优秀的架构设计和扩展机制，轻松地支持稳定性和可观测性方面的自定义扩展。

第9章 Service Mesh架构的工程化设计

云计算时代，云计算基础设施的强大支持给容器化服务在架构开发、测试、运维等全流程均带来了革命性的变化。同时为了适应并利用云计算环境和云基础设施的特点，人们对业务服务架构设计提出了很多新的要求。下面会从云计算环境和云基础设施的特点出发，聚焦具体的架构设计思想和原则，详细讨论云原生时代一些新的架构设计挑战。

云计算和容器化环境下的一大改变是，服务运行环境随时可变，和之前的静态架构相比有如下几个显著特点：

- 基础设施即代码；
- 配置化、自动化；
- 可演化架构；
- 数据驱动架构；
- 生产环境驱动的架构。

Istio/Envoy以及所依赖的Kubernetes，都是云计算和云原生时代非常有代表性的开源项目，这些开源项目蕴含众多优秀的架构设计思想，对日常的微服务业务开发会有很多借鉴价值。

本章接下来会重点剖析Istio/Envoy等背后的一些设计原则和架构思想，并结合一些具体实例进行剖析，看看这些架构思想在Istio/Envoy中是如何落地的。

9.1 复用和解耦

数据平面和控制平面分离是Istio最核心的架构思想，整个Istio项目就是在这个准则下建立起来的，可以说没有数据平面和控制平面分离，就没有Istio。

再简单回顾Service Mesh发展的简要历程。第一代Service Mesh采用Sidecar模式，只有数据平面代理，没有控制平面，数据平面代理直接负责和其他外部系统进行交互，通过Sidecar模式，将通信和通信链路治理功能从微服务中剥离出来，实现了通信基础设施的下沉和服务化。这里也体现了架构解耦的思想，通过解耦减少了微服务的负担。

但Sidecar模式下的Service Mesh有一个突出的问题，即将通信和通信链路治理的所有功能都放到这个代理服务中，导致数据平面代理很重，并且由于承载了太多的特性和功能，使得数据平面代理的更新和修改会特别频繁，而频繁的更新和升级会导致代理服务容易出问题，影响代理服务的稳定性。同时，Service Mesh模式下，数据平面代理承载了微服务通信的全部流量，对稳定性要求极高，这个服务的任何故障都会对整个系统的稳定性产生很大的影响。为了解决上述频繁升级和稳定性之间的矛盾，令策略和配置决策逻辑从代理服务中脱离出来，形成了独立的控制平面，这就是第二代Service Mesh。

第二代Service Mesh最重要的标志就是控制平面和数据平面分离，所有的代理服务都由控制平面掌控，通过控制平面可以控制整个系统，从而提供了强大的控制能力和策略能力；同时将具体的控制逻辑从数据平面移除，简化了数据平面的设计，并且数据平面不需要和外部系统进行交互，只聚焦在变更频率很低的流量路由和转发逻辑上，这提升了数据平面的稳定性。

对于数据平面代理Envoy自身来说，设计上也是采取控制流和数据流解耦的方式，主线程处理控制流逻辑，工作线程负责处理请求路由和转发，通过这样的划分和解耦，简化了系统设计，提高了系统的可维护性。

控制平面和数据平面解耦，主要是基于变和不变的考虑，数据平面可以说是Service Mesh的内核，负责提供Service Mesh的最核心价值，因此从架构设计上考虑，应该尽量减少核心内核的变化，而将变化频繁的控制逻辑移到控制平面，可以很好地保证数据平面的稳定性和可维护性。

根据上述分析和讨论，控制平面和数据平面分离似乎是个完美的设计，但是这种设计导致数据平面进行请求转发时，需要调用控制平面Mixer的接口进行前置条件检查和遥测统计上报，致使每个请求转发过程中多了两次RPC调用。根据Istio给出的性能测试数据，这会对Service Mesh的整体性能产生一定的影响，尤其是对一些延迟敏感的微服务的性能的影响还是很大的。对于这个问题，Istio官方没有给出统一的解决方法，当前只是建议，如果是延迟敏感的场景，可以考虑将Mixer功能移到数据代理服务上完成，或者暂时先不开启Mixer特性。

从Mixer访问带来的性能问题可以看出，架构设计没有完美的方案，任何设计都要从具体的业务和场景出发。很多情况下，完美的架构设计在业务现实面前都需要进行一定的折中和取舍。因此，架构设计没有完全通用的方案，只有适合业务的方案才是最好的方案。

9.2 架构扩展机制

任何一个系统，随着使用场景和使用方式的不断变化，随时会面对很多新的挑战。为了应对这些挑战，需要保证在内核基本稳定的前提下建立一套完善的插件机制，插件使用者可以使用自定义的方式进行扩展，并且不影响系统基本架构的稳定性。

Istio作为支撑微服务通信的基础设施，对扩展性有着非常高的要求，提供了多维度、多层次的插件扩展机制，大体汇总如下。

- 1) 服务注册中心：服务管理上，可以提供自定义的服务注册中心支持。
- 2) 通信协议：不仅支持常见的HTTP协议和TCP协议，也对各种用户自定义协议提供支持。
- 3) 链路治理：不仅支持各种自定义的流量调度和流量管理，也可以支持用户以自定义的方式对请求进行加工处理。
- 4) 遥测和统计：支持多种适配器，并且可轻松地增加新的自定义适配器。

Istio通过各种不同形式的插件机制实现了强大的扩展性支持，插件从实现层面看其实很简单，本质上是一个钩子回调函数，插件注册就是将钩子回调函数挂在插件机制上，在事件到来时，触发回调函数的调用。因此研究插件机制重点需要关注两点。

- 1) 插件的抽象。抽象是架构师必备的一项基本能力，抽象就是从变化中提取不变的过程。通过研究经典的抽象过程，可以为后续设计自己的插件机制提供参考。
- 2) 回调和通知的机制。

接下来会从这两个维度剖析Istio当前几个典型插件机制的具体实现。

9.2.1 服务注册中心插件机制

Istio通过服务发现抽象接口进行服务发现基本能力的定义，服务发现抽象接口定义了一个注册中心以提供基本的能力，各注册中心只需要实现这个接口。

插件回调和通知机制方面，服务注册中心提供了两个方向的回调触发。

1) 将注册中心插件注册到Istio中，可以通过插件获得具体的服务发现能力；

2) 插件会提供服务变更抽象接口，Istio反向向插件中注册用于通知的回调函数，当注册中心发生变更时，插件会调用预注册的Istio回调函数。

9.2.2 Envoy Filter插件机制

先看一下Envoy插件注册和查找的具体实现。Envoy插件采用静态注册的方式，通过声明Registry::RegisterFactory实例完成插件的注册。RegisterFactory是一个模板类，定义在Envoy项目的/include/envoy/registry/registry.h文件中，RegisterFactory的大体实现如下：

```
template <class T, class Base> class RegisterFactory {
public:
    RegisterFactory() {
        FactoryRegistry<Base>::registerFactory(instance_);
    }
private:
    T instance_{};
}
```

实现非常简单，定义一个插件实例，在构造函数中调用registerFactory对插件进行注册。在registerFactory函数中，直接将插件名和插件实例放到插件全局map中。

查找和处理插件时，调用Config::Utility::getAndCheckFactory通过插件名，从上述注册的插件全局map中查找需要的插件实例。

9.3 性能设计

9.3.1 基于TLS的无锁设计

Envoy进行Metric信息统计时，主线程通过map结构central_cache_保存每个scope内的Metric信息，具体包含counter、gauge和histograms统计信息，CentralCacheEntry定义如下：

```
struct ScopeImpl : public TlsScope {
    ...
    CentralCacheEntry central_cache_;
    ...
};
struct CentralCacheEntry {
    StatMap<CounterSharedPtr> counters_;
    StatMap<GaugeSharedPtr> gauges_;
    StatMap<ParentHistogramImplSharedPtr> histograms_;
};
template <class Stat> using StatMap = CharStarHashMap<Stat>;
template <class Value>
using CharStarHashMap = absl::flat_hash_map<const char*, Value, CharStarHash, CharStarEqual>;
```

首先讨论没有启用TLS的场景，对于stats统计信息来说，各个工作线程负责更新统计数据，主线程负责对统计数据进行汇总，工作线程更新统计数据时，如果当前统计项不存在，可能需要增加新的统计项，再进行统计结果更新。由于每个工作线程均可能对central_cache_结构进行修改，因此主线程和工作线程对central_cache_操作时均需要加锁。

为了减少锁的粒度，Envoy通过TLS在每个工作线程中均维护一份central_cache_数据，这样各个线程对central_cache_进行并发操作时，绝大多数场景下均不再需要加全局锁（除了统计项不存在的场景，在central_cache_中增加新的统计项时需要加全局锁），只需要对每个统计项（比如Counter、Gauge）进行加锁（Envoy中Counter等统计数据通过C++11的std::atomic进行保护，相当于无锁操作）。

统计数据更新是非常高频的操作，通过TLS减少了统计Metric信息时的锁开销，通过空间换性能的方式（每个统计项均需要增加一个std::atomic进行细粒度的并发控制），提高了系统的整体性能。

Envoy XDS配置信息更新和统计数据的思路类似。不同的是，对于XDS配置更新来说，主线程负责XDS配置的写入，工作线程负责配置的读取，为了减少工作线程和主线程之间的锁开销，通过TLS将XDS配置在各个工作线程中，并均复制一个副本，主线程在更新配置时通知各个工作线程进行配置更新，各个工作线程在处理请求的间隙，对本地的XDS配置进行更新，并在请求转发时直接查询本地的XDS配置，不需要任何加锁操作。

9.3.2 多级缓存机制

对于比较耗时的操作，可以通过缓存机制对处理结果进行缓存，提高系统的整体性能。Istio Mixer检查处理时，在Mixer Client和Mixer Server上均增加相应的缓存模块，通过缓存减少对Mixer以及后端基础设施的调用，提高系统的整体性能。

Kubernetes informer也是一个通过缓存提升系统性能的典型例子，和Mixer缓存对数据的一致性要求比较低不同，Kubernetes informer管理的是Kubernetes核心对象，对数据的一致性要求较高，为了保证本地缓存和远端存储的数据一致，Kubernetes informer增加了对远端存储数据变更的监听，远端存储变化时会通知Kubernetes informer进行当前节点上的数据更新。

9.3.3 批量更新机制

对于不需要同步处理的场景，可以通过同步返回、后续异步批量处理的方式，提高系统性能。Envoy在进行Metric上报时就采用异步批量的方式，当Metric数据积累到一定大小或者积累到一定时间后才进行实际的数据上报。

Envoy进行Mixer配额管理时的预取操作也是一个很好的借鉴思路，Mixer的配额管理采用滑动窗口的机制，为了减少Envoy和Mixer Server之间的交互开销，Envoy每次从Mixer Server获取多个配额，当时间窗口内的配额用尽时，才进行配额获取和更新。和Linux文件系统中有名的预读机制类似，通过预先获取一定的缓冲空间，减少获取与交互的频率。

9.4 架构设计的权衡

架构固然有很多通用的设计思想和原则，但很多架构设计原则本身就是一对矛盾体，比如分布式系统的CAP、性能和效率等，同时满足所有的设计要求是不可能的，需要针对具体的场景，进行一定的折中和取舍。

架构设计就是个权衡的过程，在无法满足所有需求的情况下，不需要同等对待所有的需求，架构设计的本质是取舍，即发现业务的根本需求，权衡哪些需求才是你真正关注的，哪些需求是当前可以舍弃的。同时也要认识到根本需求也不是一成不变的，随着时间和业务的变化，根本需求也可能会有变化。

最终一致性是Envoy中非常重要的一个架构设计原则，在线程模型、服务和配置发现等很多方面都能看到最终一致性的存在。

对于分布式微服务来说，每个微服务节点随时都可能发生故障或者出现连不上的现象；另外，对微服务集群进行任何变更操作时，比如上线升级、回滚等，均不可能同时变更每个服务；在变更微服务拓扑时，也不可能同时重启或关闭每个微服务。因此，对于微服务来说，完全的强一致性是一个几乎不可能实现的事情。

如果微服务的服务发现和配置管理，采用强一致性语义，则网络故障造成网络分区会导致整个集群不可用，这个对于微服务来说是不可接受的，微服务宁可使用5分钟之前的配置数据，也不能容忍因为暂时的网络故障而导致服务不可用。因此，从分布式系统的CAP原则看，服务发现和配置管理本质上是一个强调高可用而对一致性没有太高要求的CP需求，只要能做到最终一致即可。

基于最终一致性的语义，对于Envoy来说，既然服务发现和配置数据不需要保持强一致性，是否可以在服务发现的每个节点内存中缓存一份数据，如果可以在服务发现节点上对数据进行缓存，那么在每个Envoy服务上是否也可以建立类似的缓存机制，如果同时允许在每个Envoy上对服务发现配置进行缓存，那么在Envoy内部的工作线程中是否也可以建立类似的缓存，通过这种基于最终一致性的发散性思考，通过在架构的各个维度全面拥抱最终一致性，可以很大程度上简化设计，提高系统整体的性能和可维护性。

Envoy线程模型的设计就是围绕数据最终一致性这个原则来进行的，Envoy主要由负责管理配置、统计等控制功能的主线程，以及负责流量路由与转发的工作线程组成。主线程和工作线程中均会访问的XDS配置信息可通过TLS的方式进行管理，同时基于最终一致性原则，每个工作线程维护一个本地XDS数据，数据更新操作由主线程负责，主线程会同时通知各个工作线程的事件调度器数据更新事件，各个工作线程在处理请求的间隙，由数据调度器触发本地XDS数据的更新，因此，各个线程更新配置信息的时间肯定是不一致的，但是可以保证最终均会触发相应的配置更新，达到最终一致性。通过将数据一致性模型退化到最终一致性原则，Envoy线程模型简化了设计，同时保证配置数据更新时不需要加锁，不影响各个线程正常的配置转发，提高了Envoy的性能。

9.5 API和SDK设计

9.5.1 声明式API设计

讨论声明式API设计之前，先来了解“命令式”和“声明式”这两个相关的概念。“命令式”和“声明式”是来自于编程语言的两个概念，“命令式”有时也称为“指令式”，是命令机器做具体的事情（what），不管最后的结果是否是你的预期，都会按照你的命令来实现；“声明式”有时也称为“描述式”，是告诉“机器”你想要做什么，由“机器”想出如何做。

声明式API是Kubernetes扩展性的基础，也是Kubernetes建立庞大生态体系的关键一环，通过声明式API，可以对API资源进行在线更新，同时通过使用控制器模式，关注API资源的增删修改事件，完成业务逻辑的编写。

Istio的API对象采用声明式API的方式，通过定义Kubernetes CRD对象，实现Istio资源的扩展。

9.5.2 代码自动生成机制

代码自动生成是一种利用代码生成工具，根据流程图或者代码IDL描述完成代码的自动生成。这种技术的优点是，可以提高开发效率，降低人力成本，本质上是让机器完成有一定规则和模式的重复性工作，把人从枯燥的程序化工作中解放出来，聚焦创新性工作。Istio和Kubernetes当前有不少代码是通过代码自动生成机制自动生成的，这样可提高业务扩展开发的效率。

Kubernetes的代码生成工具当前已经变成独立的项目code-generator，Github地址为<https://github.com/kubernetes/code-generator>，使用code-generator可以非常快捷地开发自定义的资源控制器，从而直接控制Kubernetes API。

Istio中的Mixer模板、Adapter适配器均可通过代码自动生成功能生成，增加新的模板时，只需要增加模板的proto描述，调用mixer_codegen.sh即可。

9.6 配置管理

9.6.1 基于Protobuf 3的配置Scheme描述

Protobuf以其强大的性能和数据描述能力，被广泛地用于微服务之间的RPC通信，但使用Protobuf作为静态配置的场景却并不多见。Istio和Envoy的API接口和静态配置均使用Protobuf 3来进行管理，基于Protobuf 3构建了功能强大的XDS API协议，并且有着很强的扩展性，为业务服务的配置和API管理提供了不少经验参考。

Envoy数据平面启动时向控制平面发送通信请求，控制平面有配置变更时主动推送最新的配置需求，Istio的数据平面和控制平面的交互属于典型的双向通信场景，非常适合采用gRPC作为承载数据平面和控制平面之间通信的载体。使用gRPC决定了数据平面和控制平面之间的通信协议基于Protobuf，也就是XDS API和XDS相关的数据采用Protobuf格式，包括集群信息、路由信息和监听器信息等。Envoy同时支持静态配置和动态配置，这些信息不仅在XDS动态配置中使用，Envoy启动时的静态配置中也需要用到。因此，Envoy启动配置使用Protobuf数据格式，可以保证静态配置和动态配置的统一性，减少数据转化的成本。

Protobuf属于静态描述语言，数据的序列化和反序列化通过自动生成工具一键生成，静态配置使用Protobuf时，可以省去手工编写配置解析代码的工作量，提高业务开发效率，同时Protobuf的静态描述能力使得基于Protobuf进行参数检查和校验非常方便。

当前Protobuf作为静态语言，和JSON等动态数据描述语言相比，不够灵活，给系统的扩展性带来了一些不便。Envoy作为微服务通信的承载平台，对扩展性有着很强的要求，在Protobuf的扩展性支持方面，也摸索出了一些可以参考的经验。

以Envoy的过滤器配置为例。过滤器是Envoy扩展性的基石，基于过滤器机制，可以对请求、连接等多个维度进行定制和修改，比如修改请求头信息，增加一些扩展字段等。通用过滤器使用的配置就比较灵活，不同的过滤器在配置形式和内容上可能差异很大，因此通用过滤器的配置描述本质上属于动态描述的范畴，使用传统的Protobuf静态描述方式可能会遇到问题，针对这种场景，Envoy采用如下的proto定义方式：

```
message Filter {
  string name = 1 [(validate.rules).string.min_bytes = 1];
  oneof config_type {
    google.protobuf.Struct config = 2;
    google.protobuf.Any typed_config = 4;
  }
  reserve 3;
}
```

针对Protobuf在动态结构描述时的弱点，Protobuf 3增加了两个数据结构google.protobuf.Struct和google.protobuf.Any，以增加对JSON的原生支持。其中google.protobuf.Struct用于将任何消息转化成JSON格式，然后将该JSON对象以Struct类型的字段形式存放，通过这种方式，可以解决上面所说的Envoy过滤器配置的扩展性问题，但google.protobuf.Struct自身也有一些问题，比如和Protobuf传统的编码方式相比，JSON格式在编码效率上会差很多，同时Protobuf本身并没有直接转换Protobuf消息和

google.protobuf.Struct的路径，必须通过JSON进行中间转换，这个中间转换也会有一定的性能损耗。为了应对这个问题，Protobuf 3同时引入了google.protobuf.Any数据结构，和google.protobuf.Struct不同，google.protobuf.Any是将消息序列化成二进制流，以一种Byte数组的方式直接存放，编码效率高很多。google.protobuf.Any中存放的是有Scheme描述的消息，google.protobuf.Struct可以存放任意形式的JSON格式数据，并可以没有任何Scheme描述，非常灵活。

在Envoy过滤器配置场景下，会综合google.protobuf.Struct和google.protobuf.Any各自的优缺点，在日志、统计与跟踪等扩展和变化比较多的场景中使用google.protobuf.Struct，因为它灵活，对于其他一些相对固定的扩展需求，使用google.protobuf.Any可以有一定的性能提升。同时可基于Protobuf 3的oneof语义，把这两个需求结合起来。

此外，针对Protobuf 3数据描述灵活性的不足，Envoy在Protobuf 3的基础上增加了Protobuf annotations（标记），比如Protobuf很难直接描述一个大于100小于1000的int32整数（但这个数使用JSON描述起来就非常方便），那么可为Protobuf字段增加相应的标记，同时因为Protobuf在自动生成代码时，允许增加插件机制对这些标记进行自定义处理，所以通过Protobuf annotations，可以增强Protobuf在很多场景的数据描述能力，下面是Envoy使用Protobuf annotations的一个简单示例：

```
message DirectResponseAction{
  int32 age = 1 [(validate.rules).uint32 = {gte: 100, lt: 600}];
  string name = 2 [(validate.rules).string.min_bytes = 1];
}
```

Istio和Envoy通过Protobuf 3的使用，结合具体的使用场景，将静态数据描述语言和动态数据描述语言各自的优点结合起来，也为Protobuf 3提供了数据描述灵活性和扩展性方面的一些经验和思考，这对业务服务的架构设计有一定的参考价值。

9.6.2 配置动态加载机制

基于XDS协议的强大动态配置能力是Istio的一大核心优势，XDS协议作为通用的配置交互协议，已被Istio作为子项目Galley的基础，用于Istio内部各个子系统之间的配置管理。它对业务配置相关的管理也有很好的借鉴价值，可以基于XDS协议在业务微服务和控制中心之间建立一个完善的配置变更通道。

集群配置变更当前使用较多的方式是双缓存切换，这种方式不仅有锁开销，并且实现也比较复杂。Envoy配置变更时数据平面和控制平面之间基于TLS的无锁化设计，给业务服务配置变更的处理提供了一个很好的思路，可以根据实际情况灵活参考。

9.7 本章小结

本章重点剖析Istio背后的一些设计原则和架构思想，对读者自己进行架构设计提供了一些参考和帮助。

第10章 Service Mesh与云原生架构

Istio当前已经可以很好地解决微服务的通信和链路治理问题，本章会深入探讨Istio在其他通信场景下的使用，比如Serverless和南北向通信，最后展望Istio在云原生时代的前景。

10.1 Service Mesh和Serverless

10.1.1 Serverless基础

云计算技术的发展，使得用户只需要关注当前需要的底层资源，通过云基础设施可以自动完成资源的供给和伸缩；通过Kubernetes、Mesos等资源调度和编排平台对业务屏蔽了集群和机器资源管理的复杂度，不仅提高了资源的利用率，也通过为用户提供完善的环境管理和资源管理平台，提高了业务效率。但在这种使用场景下，用户仍然需要预估需要的资源，并在云平台上预先进行分配，而不管这些资源是否会被充分利用。因此对用户的容量预估和资源预估提出了很高的要求，如果预估和实际有很大偏差，就会导致很大的资源浪费，或者出现资源不足而影响业务使用，为了解决这个问题，Serverless应运而生。

Serverless中文可以翻译为“无服务器架构”，是最近几年兴起的一种新型架构理念，Serverless不是指不需要服务器，主要是强调在Serverless下，业务不再需要固定的服务器资源，而是按需分配相应的服务资源。当业务请求比较多时，会多分配相应的服务实例；当业务请求比较少甚至为零时，可以不分配任何服务器资源。Serverless架构和之前的架构相比，最大的差异是：**业务服务不再是固定的常驻进程，而是真正按需启动和关闭的服务实例**。因此Serverless架构可以提供如下两点优势。

（1）按需使用的资源管理

Serverless架构下，容量规划成为历史，对一个业务进行准确的容量规划通常来说是特别有挑战性的事情，需要考虑的因素很多，通过Serverless的即用即弃，不再需要容量规划，减少了业务架构和运维的复杂度。

同时对于按需使用的资源管理，业务只需要为真正使用的部分付费，从而减少了不必要的资源浪费。

（2）简化业务运维复杂度

在微服务架构下，微服务由各个功能模块，以及功能模块之间的通信组成，微服务开发者不仅需要关注业务功能，还要花费很多精力在完成不同微服务的交互和通信管理上。在Serverless架构下，你只需要实现具体的功能函数并提交给Serverless，各功能函数之间的交互由Serverless接管，开发者不再需要关注功能函数交互的细节。

在简化微服务管理复杂度上，Serverless和Service Mesh的目标是一致的，都是将微服务通信和服务治理相关的非功能需求从业务中剥离，对于Serverless是交给Serverless框架负责处理，对于Service Mesh是下沉到底层，成为通信基础设施的一部分。

1.Serverless基础架构

在Serverless中，按需执行的代码片断称为“函数”，它是Serverless资源管理和调度的基本单位，因此Serverless有时也被称为FaaS（Function As A Service，函数即服务）。Serverless架构大体由以下几个部分组成。

（1）函数管理

Serverless需要对用户编写的函数进行管理，通过一定的方式将用户函数变成可调度、可运行的实例。为了支持多个语言的Serverless函数，函数管理需要针对每种语言定义函数规范和标准，提供相应的实现机制。

（2）事件触发器

事件驱动是Serverless非常重要的组成部分，Serverless一般会定义一组关注的事件类型，并提供相应的事件触发和订阅框架，函数需要实现注册好关注的事件类型，事件触发时，Serverless框架查找关注这个事件的函数，触发函数的具体执行。

（3）函数的路由和伸缩管理

在很多Serverless实现中，函数的访问和路由一般由API网关来负责，API网关根据相应的路由规则，找到相应的服务实例进行访问。

在伸缩管理方面，监控函数当前的请求和资源。如果资源过多，则进行相应的资源回收；若资源不够，则增加相应的资源实例。

在Serverless中，函数实例按需创建运行，按需伸缩调整，函数实例必须特别轻量，能够快速启动，快速关停，Serverless也需要提供完善的机制，支持函数实例的快速启动。当前Serverless在快速启动方面还没有很好的解决方案，快速启动的程度决定了Serverless的应用范围和应用场景，在很多场景下，如果没有快速启动的支撑，Serverless很难应用起来。

2.Serverless的发展历程和当前现状

2014年11月，AWS发布了新产品Lambda，开启了全新的Serverless时代。按照当时的描述，Lambda是一种计算服务，它按需运行用户的代码，用户无须关注底层的计算资源。借助AWS Lambda几乎可以在任何类型的应用程序或后端服务中运行代码，并且不必进行任何管理。AWS Lambda在可用性高的计算基础设施上运行代码，执行计算资源的所有管理工作，其中包括服务器和操作系统维护、容量预置和自动扩展、代码监控和记录，用户只需要提供AWS Lambda支持的语言代码即可。

继AWS Lambda之后，很多公有云提供商都推出了自己的Serverless支持。2016年Google、Microsoft Azure相继推出自己的Cloud Function服务，2017年国内公有云提供商阿里云和腾讯云也分别推出了各自的Serverless产品。

从支持的特性和当前的成熟度看，AWS Lambda最为强大，在语言方面，当前支持Node.js、Java、C#、Go和Python；从事件触发来看，AWS Lambda不仅全面支持HTTP、定时器、数据库、对象存储、消息队列等基础事件，还对边缘计算、IoT等新兴技术进行跟踪和支持。AWS Lambda通过Step Function还提供了对事件编排的支持，可以将多个函数编排在一起，以工作流的方式运行。

整体来说，Serverless当前还处于发展初期，正在快速地迭代，并且各个云厂商均推出一套自己的Serverless标准，所以这些Serverless是无法互通的，从而导致一旦选定了某个云厂商，就会被这个云厂商的技术体系绑住，无法自如地迁移。

面对Serverless领域当前的无序和混乱，Google联合IBM、Pivotal、Redhat和SAP，一起推出了Knative，目的是实现Serverless标准化。

10.1.2 Knative

Knative是一个以Kubernetes和Istio为基础的Serverless开源架构方案，目的是解决Kubernetes环境下Serverless应用的构建、部署和运行。Knative和以往Serverless解决方案最大的不同是，Knative不仅管理函数，它的定位是管理所有的工作负载，除了Serverless关注的函数外，还包括普通的微服务，因此Knative强调的是通过相应的机制，用户不用再关注应用的伸缩和运维。

此外，Knative不是实现一个具体的Serverless，而是构建一个能够运行标准化Serverless的平台，Knative提供大量的扩展性支持，允许其他Serverless产品在其上运行。

架构上，Knative由Build、Serving和Eventing 3个核心组件组成。其中Build负责构建工作，把用户定义的函数和应用构建成容器镜像；Serving负责计算工作，具体包含Serverless实例的路由和访问，以及Serverless实例的按需伸缩；Eventing负责事件工作，自动完成事件的绑定和触发执行。

由于本书重点关注微服务的通信和服务治理，下面重点讨论Knative的Serving系统。Serving系统基于Kubernetes和Istio进行开发，利用Kubernetes强大的容器调度和生命周期管理能力以及Istio的通信和通信链路治理能力。另外，为了屏蔽Kubernetes和Istio的复杂度，Knative Serving自身有更高一层的抽象能力，提供一组用来对应用和通信进行管理的抽象概念（可使用Kubernetes CRD对这些抽象概念进行管理）。Knative Serving的主要概念如下。

- 1) Route: 工作负载的路由规则，对应Istio的VirtualService。
- 2) Configuration: 应用的最新配置，对应Kubernetes的Deployment。
- 3) Revision: 每次对工作负载进行修改的快照，对应Istio的Subset。
- 4) Service: 对工作负载的整个生命周期进行管理。

Knative的Serving系统做的事情和Kubernetes、Istio大体相同，但通过提供更高的抽象，屏蔽了Kubernetes、Istio的细节，自动帮应用管理好Deployment、VirtualService以及auto scaling之间的关系。

auto scaling是Knative Serving实现工作负载自动伸缩的关键组件，当前还处于发展早期，有不少性能问题，很不成熟，Knative后续计划将auto scaling下沉，直接复用Kubernetes原生的auto scaling能力。

基于Kubernetes的Serverless产品和解决方案市场上已经有不少，比如Fission、Funktion、Kubeless等，但同时依赖Kubernetes和Istio这两个重量级的项目，Knative还是第一个。Kubernetes和Istio的复杂度都很高，同时基于Kubernetes和Istio，再加上Serverless自身的复杂度，会导致整个Serverless解决方案的复杂度非常高，学习曲线上也非常陡峭，因此很多人都有如下疑问和质疑：Knative中为什么采用Kubernetes和Istio这么重的通信解决方案？

从架构和通信功能实现来看，确实没有必要，为了对上层用户屏蔽底层Kubernetes和Istio的实现细节，Knative又引入了一层抽象概念，对Kubernetes的容器和通信功能进行封装。如果通信直接放在Knative层面上做，只聚焦Kubernetes平台对Serverless的通信支持，不需要平台扩展性和场景扩展性支持，架构设计上会特别简单，同时灵活性也会比现在好。

从短期和静态的角度看，Knative引入Istio，确实没有那么大的必要，因为增加了复杂度，减少了灵活性。但从云计算基础设施的大趋势来看，Kubernetes已经成为容器调度和生命周期管理的事实标准，可以看成云原生时代的Linux操作系统；Istio虽然产生时间不长，但已经显现出强大的发展势头，当前在Service Mesh和Kubernetes生态中的地位已经不可撼动，可以看成云原生时代的TCP/IP协议。

因此，如果从云计算的发展趋势上看，通信功能会被Istio完全接管，并下沉为基础设施的一部分。在通信层面，Istio肯定更聚焦、更专业，Knative从大趋势出发，直接复用Istio的通信层功能，避免后续架构上的修改和返工，可以将自身聚焦到Serverless层面的功能和生态打造上。从架构的角度看，这也是Unix设计哲学的体现——整个系统采用模块化设计，每个模块只聚焦一件事情，并且做好做到极致。

10.2 东西向和南北向通信的统一

在微服务的整体架构中，存在着东西向通信和南北向通信两种方式。所谓的**东西向通信**是解决业务内部各个微服务之间的通信和链路治理；**南北向通信**解决的是集群和外部的交互问题，具体包含集群服务如何提供给外界访问，以及集群内部如何访问外界的服务。这里可以使用传统数据通信上的概念进行类比。可以认为，对外实现一个业务的所有微服务，构成了一个用户驻地网：**东西向通信**负责各个用户终端之间的通信，负责将流量引入到具体的终端用户（对应每个微服务），而具体终端的接入链路则实现接入网的功能；**南北向通信**负责一个独立的子网络和外界的交互，承载了子网络对外交互的全部流量，可以看作骨干链路，实现核心网的功能。

Service Mesh本身主要是聚焦东西向的微服务内部通信。微服务内部服务通信的特点是流量不太大，**QPS**通常来说不太高，对性能总体要求不高，强调的是通信链路的精细化管理能力和稳定性保障能力，如何通过相应的机制减少业务对通信的关注度，给业务提供效率和稳定性支撑。

从通信特性要求上看，南北向通信和东西向通信有着非常大的区别，南北向通信有两个显著的特点。

第一个是流量特别大，需要承载整个业务所有的对外流量，所以**QPS**非常高，对性能有着非常高的要求。

第二个是南北向通信位于业务内网和外部网络之间，需要对业务内网进行保护和拦截，防止非法和异常请求对内网的攻击和影响，因此南北向通信对安全有着非常高的要求，需要不断对安全防控和安全策略进行升级，保护内网业务的安全和稳定性，因此对于南北向通信来说，性能和安全是关键词。

从功能看，东西向通信和南北向通信提供的能力差不多：从通信层面看，都需要关注流量路由和负载均衡；在链路治理方面，都需要关注限流、降级等治理特性。

从上述分析看，在功能层面上，完全可以通过**Istio**实现东西向和南北向通信的统一，基于一套通信基础设施可解决这两个维度的通信问题。当前的主要问题出现在性能上，南北向通信对性能有着非常高的要求，**Istio**在性能方面当前还有很多问题需要解决，不能完全满足南北向通信的需求。

单纯从**Istio**的数据平面看，**Envoy**在设计上和**Nginx**相比各有千秋，同时**Envoy**的性能也足够出色，当前已经有不少基于**Envoy**的API网关解决方案。但综合数据平面和控制平面来看，**Istio**在性能方面还有很多需要改进的空间，如果不在架构上进行一定的调整，目前**Istio**还是很难直接胜任南北向通信的性能需求，东西向通信和南北向通信的完全融合，还有很长的路要走。

10.3 云原生时代的Service Mesh

1.云原生架构

讲云原生前，首先介绍一下CNCF。CNCF（Cloud Native Computing Foundation，云原生计算基金会）于2015年7月成立，隶属于Linux基金会，围绕“云原生”服务云计算，致力于维护和集成开源技术，支持编排容器化微服务架构应用。2016年11月，CNCF开始维护Cloud Native Landscape——汇聚了当前最热门的云原生技术与工具，可为企业构建云原生体系提供参考，在云原生生态领域具有广泛的影响力。

云原生（Cloud Native）是一种方法，用于构建充分利用云计算模型优势的应用。这个概念最早于2013年由Pivotal的Matt Stine提出，得益于它的良好可伸缩性，云原生应用和服务也越来越受到青睐。

云原生包含一组应用模式，用于帮助企业快速、持续、可靠、规模化地交付业务软件。云原生由微服务架构、DevOps和以容器为代表的敏捷基础架构组成。

云原生对应用架构最大的影响是服务不应该仅仅止步于云化，如果仅仅将服务迁移到云上，而对服务本身的架构没有进行云化适配改造，那么服务上云的收益无法最大化。云原生应用架构的诉求是最大程度地利用云基础设施的优势，且应用自身的架构也需要升级。

2.云原生和微服务

严格来说，对于架构、组织和流程，云原生对微服务的影响不大，没有提出太多新的要求，云原生应用架构主要强调业务架构要面向敏捷和变化编程。同时，为了保证微服务在各个云平台之间能够透明迁移，微服务架构和服务治理需要有一定的可移植性，以及采用平台无关的通用方案。对于微服务通信最好不要用私有协议，因为私有协议和外部通用协议交互时需要进行频繁的协议转换；服务治理方面也是如此，如果大家使用的都是私有方案，那么通信和互通上效率就比较低。

为了实现微服务的可移植性，微服务通信和服务治理需要能够标准化和平台无关，因此需要针对微服务通信和服务治理基础设施制定相应的标准。CNCF在这方面做了大量的工作，目的是建立一套完善的微服务基础设施标准，具体包括微服务通信、分布式跟踪、日志、度量、安全等。微服务只有实现了基础设施标准化，才能做到架构层面的平台无关性和可移植性，通过云任何时候均能按需提供和释放计算、网络、存储资源，支撑业务的持续可用和持续创新。

3.云原生和Service Mesh

应用为了实现云原生设计，需要持续轻量化，将业务需求和非业务需求解耦，业务只需要关注业务相关的功能需求，开发、测试和运维的效率才能大大提高。非功能需求解耦并下沉到Service Mesh中，微服务就会变得非常轻量化，方便进行云化部署和维护。因此Service Mesh是云原生落地过程中至关重要的一环，不可或缺。

10.4 Service Mesh现状和展望

1.Istio现状和挑战

Service Mesh技术从诞生到现在还不到4年时间，作为Service Mesh领域的指标性作品，Istio承载着人们对Service Mesh的万千期待，但从出现到现在还不到2年，2019年3月发布了最近一个重要版本Istio 1.1，在性能、API成熟度等方面仍存在诸多问题，架构和性能的博弈从Istio诞生一直延续到现在。Istio过分追求优雅的架构设计，导致性能改进一直没有实质的进展，严重制约着Istio在生产环境下的落地，可以说截至目前Istio仍然处于一种“叫好不叫座”的阶段，基于Istio基本无懈可击的架构设计，以及Google在云计算领域强大的技术影响力，大家对Istio当前还都秉持谨慎看好的态度，仍在持续观望。

Istio性能问题主要是因为Mixer多次远程服务访问开销和流量透明拦截带来的网络与处理开销。针对Mixer访问开销，Istio启动了Mixer v2项目，舍弃了当前Mixer v1架构上的解耦，将Mixer功能下沉到Envoy中，Mixer代码采用Golang编写，而Envoy使用C++语言，为了减少当前已有Mixer代码的迁移工作量，Envoy当前已经尝试通过Web Assembly技术，将当前Golang语言的Mixer代码转换为wasm格式，供Envoy直接调用。

采用基于Iptables的流量透明拦截，客户端与服务器端的服务调用次数和直接调用相比多2次，同时Iptables本身也有很大的性能损耗。针对流量透明拦截带来的网络和访问开销，Envoy已经尝试通过Cilium技术来解决。Cilium是一个为应用程序间的网络通信提供透明保护机制的开源项目，当前主要面向容器使用，Cilium当前已被集成到常见的容器编排系统中，如Kubernetes和Mesos。Cilium的底层实现以BPF技术为基础，工作模式是生成内核级别的BPF程序和容器直接交互，绕开了TCP协议栈，极大地提升了性能。

Istio在Mixer和透明拦截上的性能优化当前都还处于尝试与探索阶段，距离实际使用估计还有不少时间。

2.Service Mesh发展展望

因为Istio当前发展上的一些困境，表现并没有如大家预期的那样，迅速一统Service Mesh领域，这也给了其他Service Mesh解决方案一定的发展窗口期，Service Mesh后续是Istio一枝独秀，还是百家争鸣，当前仍然需要持续观察。

10.5 本章小结

本章主要讨论Istio在Serverless、东西向和南北向通信场景下的现状与问题，以及展望Istio在云原生领域的前景，通过本章的学习，你可以对Istio相关的生态有更清晰的认知。

附录 Service Mesh迁移的要点与原则

本附录主要讨论在实际的生产环境中使用Service Mesh需要注意的一些问题。比如，首先判断是否需要引入Service Mesh解决方案，具体需要综合考虑哪些因素；接着介绍Service Mesh落地路径；然后讨论Service Mesh迁移过程中需要注意的一些原则，以及如何根据实际使用情况，对Istio进行扩展和定制。

A.1 Service Mesh落地路径

在经过A.1节的分析后，确定要进行Service Mesh引入时，就要考虑Service Mesh落地的具体路径。首先是业务当前所处阶段是否已经支持容器化和Kubernetes。如果业务当前已经运行在Kubernetes上，因Istio对Kubernetes有着很好的支持，Service Mesh迁移就会非常顺畅；如果业务当前没有运行在Kubernetes容器上，因Service Mesh领域当前最强大的Istio对Kubernetes有一定的依赖，所以可能就无法直接使用Istio，即使定制修改Istio，解除对Kubernetes的依赖，也需要很大的代价。这时通常有如下两条路径可以选择。

（1）非Kubernetes环境下，先接入Sidecar

如果业务当前没有办法快速容器化，同时又有引入Service Mesh的迫切需求，可以先接入Sidecar，满足业务当前的痛点需求。至于接入Sidecar之后的演进方向，需要看业务的未来规划中是否有Kubernetes容器化的计划，如果有，建议先进行容器化改造后，再由Sidecar的方式演进到Istio。

Istio在非Kubernetes环境下没有很好的支持，对Istio进行定制修改以支持非Kubernetes环境有很大的工作量，除非有非常强烈的需求和非常强大的技术储备，一般不建议这么做，特别是对于一些中小公司来说。

如果一定要在非Kubernetes环境下实施Service Mesh，可以在数据平面使用Envoy，控制平面根据XDS协议自己研发，这样也非常灵活，方便控制。

（2）先进行Kubernetes容器化改造，再接入Istio

对于近期有Kubernetes容器化需求的团队，可以在Kubernetes容器化完成之后，再进行Istio接入，Istio在很多方面利用了Kubernetes强大的基础设施能力，所以只有在Kubernetes下Istio才能最大限度地发挥作用，并且减少运维上的复杂度。

A.2 Service Mesh迁移原则

1.业务透明

为了减少Service Mesh迁移对业务的影响，减少业务的迁移阻力，迁移刚开始一段时间，必须保证迁移过程中，业务完全透明且不需要有任何变更和修改。

方案上，为了保证迁移对业务的完全透明，在数据平面通信上需要采用支持透明拦截的方式，对业务请求流量透明拦截。

2.渐进式迁移

为了减少Service Mesh迁移过程中的风险，必须采用渐进式迁移原则，每次只迁移一个服务，迁移后观察足够长的时间，没有问题后再进行接下来的迁移。

同时为了保证在迁移过程中出现问题时，可以随时回退到迁移之前的状态，需要针对每个迁移动作增加一键回滚开关。如果遇到线上问题，可以立即通过开关关闭迁移特性，回归到迁移之前的状态。

3.迁移前后网络打通

如果Service Mesh迁移前后使用不同的网络方案，迁移过程中就会出现已迁移部分和未迁移部分之间的网络无法直接访问的场景。为了应对这种情况，迁移过程中需要将未迁移服务和已迁移服务之间的通信从内网访问修改为外网访问，外网访问和内网访问方案上差异很大，需要通过修改路由配置的方式来支持，对路由配置的频繁修改，无形中会增加迁移出错的概率。

为了保证迁移过程平稳进行，建议Service Mesh迁移前后采用相同的网络地址空间，保证服务迁移前后均可以正常访问，这样迁移过程中网络访问层面对业务透明，业务不需要感知，访问方式也不需要有任何变化。

4.Istio性能

Service Mesh的实践方面，性能是一个永远不能回避的问题，如果当前性能不能满足业务的要求，就不具备在生产环境下使用的条件。

对于Istio来说，当前性能确实是阻塞Service Mesh大规模实施的关键因素，引入Istio的目的是解决痛点需求，Istio最有价值的部分是Envoy和Pilot，它们一起完成请求路由转发的配置管理，以及实际的转发。

Mixer架构设计上看起来很美好，但业务实际上并不会过多使用Mixer组件，对扩展性和灵活性也并没有那么高的要求，因此出于性能考量，可以把Mixer相关的功能放到Envoy，只实现最精简、最必要的检查和遥测统计逻辑。

Istio security则需要根据业务的具体情况灵活选取。比如我司业务放在自建机房的私有云中，对于Istio面向的微服务内部的东西向通信来说，在内网环境下不需要考虑过多的安全问题，所以可以直接把安全部分裁掉；当然，如果是在公有云中部署Service Mesh服务，则需要对安全部分更多关注。

从解决痛点需求和综合考虑性能，可以先聚焦Istio提供的核心价值，将当前不需要太多关注的部分裁掉。

此外，出于对扩展性的考虑，Istio架构的整体性能会受一定的影响，如果你的业务只运行在特定的平台上，比如Kubernetes，可以根据具体平台对Istio进行相应的定制和优化。

A.3 Istio的扩展和定制

1.Istio和Kubernetes的去耦合

之前已经分析过，对于非Kubernetes环境下，并且相当长一段时间内也没有迁移到Kubernetes的计划时，没有必要使用基于Istio的Service Mesh方案。但如果是多平台支持的场景，比如客户中既有Kubernetes环境用户，也有非Kubernetes环境用户，这种情况下，可以对Istio进行定制修改，去掉对Kubernetes的强耦合，并且增加对其他平台的支持。

Istio对Kubernetes的耦合主要有如下几个方面，因此需要有针对性地进行适配修改。

（1）API资源管理层面对Kubernetes API Server的依赖

资源管理是Istio对Kubernetes依赖最大的地方。Istio对核心资源的管理，是以Kubernetes CRD为基础的，并且使用kubectl作为命令行操作入口，kubectl会调用API Server，将资源存放在etcd中，并通过Kubernetes CRD机制触发资源变更事件通知，通知关心Istio资源变更事件的模块进行相应的处理。

Kubernetes通过kubectl、API Server、etcd和Kubernetes CRD组件，提供一套完善的API资源管理机制，包括资源的增删查改和资源的变更通知，如果需要解除Istio对Kubernetes的绑定，就需要自行实现这一套API管理方式，并且做到平台无关。

（2）通信访问层面对kube DNS的依赖

通信层面，在客户端发送请求前，先通过DNS获取服务的虚拟IP地址，Istio的DNS实现沿用Kubernetes DNS方案（Kubernetes DNS是Kubernetes网络管理中的关键一环），基于DNS，通过服务名就可以直接访问服务。因此通信层面，需要在DNS方案层面解除和Kubernetes的耦合，并使用平台无关的DNS解决方案。

2.平台无关的Istio支持方案

Istio作为控制平面，主要是对数据平面Envoy的服务信息和配置信息进行管理，因此平台无关的Istio支持方案，主要需要提供平台无关的配置管理中心和服务注册中心。

Consul是管理分布式系统的配置和服务发现的开源项目，和ZooKeeper、etcd等其他分布式服务发现方案相比，Consul提供了一站式解决方案，内置了服务发现、健康检查、key/value存储、多数据中心方案、REST API接口等众多特性。推荐使用Consul作为平台无关的配置中心和服务发现解决方案，Istio当前也提供了对Consul的基本支持，但还不太成熟，所以无法直接应用到生产环境中。

3.RPC协议扩展支持

Envoy当前重点提供对HTTP1和HTTP2的支持，而对RPC协议的支持还很弱。RPC协议方面，当前虽已提供对Thrift协议和Dubbo协议的支持，但支持的程度都很弱，截至本书出版时只支持基本的协议编解码，没有精细的路由管理和负载均衡策略，也不支持常见的链路治理特性，因此它们均不具备在生产环境下使用的条件。

如果需要对Envoy增加相应的RPC协议扩展，可以参考Envoy下Thrift协议的具体实现，首先要定义RPC协议对应的Network Filter插件，并通过RegisterFactory进行静态注册。

在插件必须实现的入口函数createFilterFactoryFromProtoTyped中，可以实现插件对应

配置的解析，比如路由相关的配置，同时要通过addFilter向filter manager注册插件协议处理函数。

在插件的OnData函数中实现具体的协议解析逻辑，解析出路由需要的参数，从而可以通过配置的路由策略选取相应的Upstream节点，进而调用Upstream节点进行相应的处理。