微服务架构模式

设计可信安全系统的一种建议架构模式

CSA GCR cloud security alliance
GREATER CHINA REGION ®

应用容器和微服务工作组的网址：

https://cloudsecirtiyalliance.org/research/working-groups/containerization/

序 言

随着数字化时代的到来，微服务应用进入飞速发展时代。微服务是一种新兴的分布式系统开发范式，在架构层面，安全性是必需认真考虑的重要工作。我国随着《数据安全法》和《个人信息保护法》的颁布，对安全和数据保护的重视程度日益提高，架构层的安全问题必将上升到组织安全治理层面。

如何保证微服务架构的安全？本文档给出了 CSA 的最佳实践与总结，通过 CSA 微服务安全参考架构以及安全控制措施叠加的新思路，保证了微服务在架构层面的安全性，CSA 微服务安全工作组也在陆续推出微服务安全相关的指南与白皮书，文章深入浅出，值得大家参考。

李雨航 Yale Li

CSA 大中华区主席兼研究院院长

致 谢

本文档《微服务架构模式》(Microservices Architecture Pattern)由 CSA 应用容器和微服务工作组专家编写，CSA 大中华区秘书处组织翻译并审校。

中文版翻译专家组（排名不分先后）：

组　长：高　卓

翻译组：高　卓　贺　进　李　岩　廖武锋　马琳琳

审校组：郭鹏程　姚　凯

感谢以下单位对本文档的支持与贡献：

北京江南天安科技有限公司　　浪潮云信息技术有限公司

北京北森云计算股份有限公司

英文版本

主编/工作组联合主席：Anil Karmel　Andrew Wild

主要供稿者：Gustavo Nieves Arreaza　Marina Bregkou　Craig Ellrod
Michael Holden　John Jiang　Kevin Keane
Numrata Kulkarni　Vani Murthy　Pradeep Nampiar
Vinod Bbu Vanjarapu　Mark Yanalitis

审稿者：Ankur Gargi　Alex Rebo　Michael Roza　Ankit Sharma

CSA 分析师：Hillary Baron　Marina Bregkou

CSA 全球人员：Claire Lehnert　AnnMarie Ulskey

在此感谢以上专家。如译文有不妥当之处，敬请读者联系 CSA GCR 秘书处给与雅正！联系邮箱：research@c-csa.cn； 云安全联盟 CSA 公众号。

目录

1. 引言

以较弱方式构建微服务的影响始终存在[1]，表现为不够安全和把应用编程接口（API）过度暴露，构成了微服务应用程序风险的核心部分。一些业务和技术负责人跳过搭建架构的方法[2]，仅凭几条粗略的要求寻找软件解决方案。然而在开放市场上寻求解决方案的人们最终还是要用搭建架构的方式把采购的解决方案融入到现有控制环境中。即便是新建的微服务应用程序也要与企业旧有的其他部分集成——没有哪家公司会年年淘汰现有架构。不采用架构方法而单纯购买解决方案将不可避免地在日后引入各种制约和附加组件，修修补补的资金成本会随着时间的推移而不断增加。

无论企业领导者倾向于购买现成解决方案还是支持“内部构建”，API 消费和微服务集成都会是一种常见的系统集成预期。最好能有一种办法指导将架构的使用、架构模式和安全控制措施叠加集成为一个整体，确保信息安全成为既定要求的集合。微服务架构模式和相应的安全控制措施的叠加为微服务的开发奠定了基础，形成一种完整的思路。模式和叠加可确保信息安全根植于产品之内。如果做得好，安全控制措施叠加会变得与用于创建微服务应用程序的架构和设计方法难以区分。有人称这种现象为“DevSecOps”（开发-安全-运维）[3]。安全控制措施叠加的概念起源于美国《联邦信息系统管理法案（FISMA）》。[4]根据 FISMA 的说法，“安全叠加是运用裁剪指南对控制基线裁剪后得出的一个全套特定控制措施、控制措施强化和补充指南集。”美国国家标准和技术研究所（NIST）特别出版物 SP 800-53《联邦信息系统和机构安全和隐私控制措施》第 4 版第 3.3 节[5]对安全控制措施叠加作了进一步阐述。尽管叠加是 NIST 引入的概念，但是 ISACA COBIT 5、PCI-DSS 3.2.1 或 CSA CCM v3.0.1 等其他控制框架也可使用。

软件开发的展开离不开以软件设计模式为引导[6]。安全控制措施叠加（overlay）是指由全

---

[1] Hinkley, C. (2019, November 6). Dissecting the Risks and Benefits of Microservice Architecture. TechZone360.https://www.techzone360.com/topics/techzone/articles/2019/11/06/443660-dissecting-risks-benefits-microservice-architecture.htm.

[2] The Open Group. The TOGAF Standard, Version 9.2 Overview, Phase A, B, C, D, E, F, and G. Retrieved August 11, 2021, from https://www.opengroup.org/togaf.

[3] Cloud Security Alliance. (2019, August 1). Information Security Management through Reflexive Security. https://cloudsecurityalliance.org/artifacts/information-security-management-throughreflexive-security/(13, 14, 16).

[4] NIST. Security and Private Control Overlay Overview. Retrieved August 11, 2021, from https://csrc.nist.gov/projects/risk-management/sp800-53-controls/overlay-repository/overlay-overview.

[5] NIST. (2020, September). NIST Special Publication 800-53, Revision 5: Security and Privacy Controls for Information Systems and Organizations.https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf.

[6] Bass, L., Clements, P. C., & Kazman, R. (2012, September). Software Architecture in Practice, Third Edition.https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264.

套特定控制措施、控制措施强化和补充指南组成的离散集，可集成到架构设计流程中，充当嵌入和既定的管理、技术或物理要求。软件设计模式与安全控制措施叠加结合到一起会告诉我们，软件开发工作要把安全作为一个设计元素“内置”到软件产品之中，而不能只把安全当作最后才涂抹到软件产品上的一层外衣，而这一层外衣到了日后成为必须付出极高代价才能做出改变的地方。本文后面的章节将为使软件架构形成一个完整思路打下基础。架构意义上的完整思路是指表现得像一个完美数学方程式的架构表达方式；把这个思路向前推进，可以得到它的预期产品，向后推演，则可以看到它的非功能和功能要求。

开发人员一旦掌握软件设计模式和安全控制措施叠加，就可以在架构成熟度上更上一层楼，从特定的微服务视角揭示底层服务交付框架具体方面（如数据、集成和部署架构）所需要的控制状态。虽然这些还不是“微服务架构”，但可以支持对于那些要求保证系统行为可重复性、可靠性、准确性和完整性的架构和设计。

微服务架构风格体现在分布式应用程序的处理足迹里，其中静态配置、动态适应和抽象化设想组合在一起所形成的能力，产生人们所说的“应用功能”。在微服务和容器化转型出现之前，许多配置和抽象化设想存在于单个整体式应用程序边界之内。随着整体代码库变得越来越大，内部应用程序的状态和相互依赖变得越来越难以分辨，从而带来许多架构、开发和运维上的挑战。然而，让一名业务代表负责业务流程功能，同时让另一名产品负责人履行应用托管义务并在一个组织结构下管理联合开发人员的情况依然十分常见。微服务架构风格改变了组织结构，就像改变软件的构建和组装一样。架构的每个部分，无论是在平台、软件定义、应用编程接口（API）层面，还是在软件开发层面，都是在微服务应用程序交付的整体功能中执行具体和特定功能。机构把多个开发团队组织到一起应对技术变化，响应计算、内存、存储和网络虚拟化成一种能力的趋势。存储、网络和服务器计算机团队的混合体由分散的团队组合而成。

随着微服务软件开发深入人心，业界越来越重视 DevSecOps（开发-安全-运维）[7]、软件组装和应用程序安全。例如，一个业务负责人可能拥有涵盖整个工作流程的应用程序，但是只负责处理涉及改进现有能力或建立新能力的前瞻性请求。业务负责人关心的是已交付或可交付成品的价值最大化；这个角色游离于软件构建团队之外。在微服务架构风格中，多个产品负责人服从于某一个业务负责人，与业务负责人保持一致的情况是可能存在的。产品负责人和相关软件开发团队（也就是敏捷用语中所说的“机组”[crew]）可能拥有特定功能的所有权，如搜索

[7] Cloud Security Alliance. (2019, August 7). Six Pillars of DevSecOps.https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/. 5-6.

功能（API 使用、排序、算法、元数据编目），而第二个产品负责人则专注于前端客户的用户体验（风格、演示、流程和整体体验）。数据的集成可能会由服务于多个产品负责人数据需求的一个联合“机组”负责。整体应用程序把所有这些角色和行动捆绑进一个垂直的支持体系，这便是微服务架构风格的一个关键特点——微服务软件的开发和生产部署行动使机构的传统垂直体系扁平化。不仅应用程序的功能是分散布局的，而且它的支撑结构也是分布式的，从而迫使我们更多地依靠自动化提供以前在垂直体系中相互隔离的各种基础设施、策略、安全、身份和网络功能。运维部门通常拥有一套部署和管理 IT 服务的流程，但是部署和测试的责任有时会左移给构建软件的开发人员。架构师往往需要在实施软件工程的过程中掌握新的技能，才能更好地把习惯和传统的架构工作转换成微服务架构风格。

附录 B 进一步定义业务负责人、产品负责人、开发人员、运维人员和架构师的角色。有关这五种角色的更多详细信息和具体定义，请参见词汇表。

2. 目的

CSA 于 2020 年 2 月出版《实现安全微服务架构的最佳实践》[8]，为读者提供了可信安全系统设计指南，其中最后一章着重从开发人员、运维人员和架构师的视角阐述。

本文旨在提出一种可重复的方法，用于按“MAP”（Microservices Architecture Pattern，微服务架构模式）构建、开发和部署微服务。我们提出的这个“MAP”包含微服务独立运行和与其他微服务通信所需要的全部信息——这些微服务聚合到一起，会形成转而又会成为应用程序成分的能力。本文描述了“MAP”的关键元素、应该怎样设计和部署，以及应该怎样通过一种合规即代码方法把安全和合规左移。

本文的主要目的是开发一个厂商中性的参考架构基础，从这个基础分解出软件和平台（企业）平面体现的软件架构模式，以后还可以通过添加安全控制措施叠加重新构建。微服务架构模式的成功分解和重组就证明了这一点；其中的集成操作便是安全控制措施的叠加。

[8] Cloud Security Alliance. (2020, February 24). Best Practices in Implementing a Secure Microservices Architecture.https://cloudsecurityalliance.org/artifacts/best-practices-in-implementing-a-securemicroservices-architecture/m.

3. 读者

本文的目标读者是应用程序开发人员、应用程序架构师、系统和安全管理员、安全项目经理、信息系统安全官以及其他对应用程序容器和微服务的安全负有责任或感兴趣的人员。

我们假定读者具备一定程度的操作系统、网络和安全专业知识，同时还掌握了应用程序容器、微服务和敏捷应用程序方面的专业知识。由于应用程序容器技术具有不断变化的性质，因此我们鼓励读者借助其他资源（包括本文列出的参考文献）获得更新和更详细的信息。

4. 架构与解决方案

架构并不是解决方案。解决方案是指通过架构、模式和设计上的努力满足特定行业需要或解决具体业务问题的办法。解决方案旨在向客户和企业负责人提供持续的价值。以 POS 机系统为例，POS 机系统是人员之间交互、技术支持的业务流程和后端支付平台运行的综合体现，用于创建一种只靠架构无法实现的特定业务能力。POS 机解决方案的设计是概念、系统属性和模式为应对某一特定业务挑战而组合到一起的结果。任何问题都会有化解挑战症结的解决方案。但是你若想搞清问题的来龙去脉，就必须回到源头去了解这个问题之所以会产生的条件和决定。

架构与解决方案的区别在于：解决方案的根本在于设计。设计包含一组模式，而这些模式起源于最早的抽象形式——一个架构。架构构成了运行环境中系统的基本概念或属性，由系统的元素、关系以及系统的设计和进化原则体现出来。商业和技术草图和图纸依然是架构师向工程设计团队表述自己想法的主要手段。为了进一步消除开发过程中的可变性，架构师与工程师一起挑选商定的模式，共同奠定设计和框架设计活动的基础。架构、模式和设计不引用特别命名的技术，保持了厂商中性。设计完成后，业务负责人和技术负责人决定是全部自行构建、全部对外采购还是结合这两种构建方法，从而针对具体问题创建一个符合业务需要的技术解决方案。有些业务和技术负责人选择跳过构建架构的过程并仅凭一组要求寻找现成的解决方案，而其他人则选择根据要求自己构建解决方案。

那些到市场上寻求一站式、垂直集成、低代码或无代码解决方案的公司，最终也还是要用构建架构的方法把买来的解决方案融进现有的控制环境。而这将不可避免地导致权衡，其中有些对初始条件极为敏感，日后倒是能够引入约束或解决方案重构，但是到了那时，改造会带来

经济成本的上升；而这无非是在偿还不断积累的技术负债。

许多人认为微服务也是一种架构，但微服务其实只是一种架构风格。大肆使用水泥预制板的粗野主义和依托精雕细刻的橡木承载使命感的工匠风格代表了能够渲染整栋建筑物的架构风格。每种风格都有自己的架构原则，应用得当时，这些原则会体现在蓝图中，引导产生特定设计，从而在物理上呈现出预期的风格。如果目标是构建高度模块化的分布式应用程序，则应用微服务原理、架构、模式和设计会导致出现一款微服务应用程序。

4.1 模式和控制措施叠加

模式是指以可预测方式发生的一组可重复动作和安排。模式是可以通过物理外观、直接或间接观察或分析看出的。设计应用系统时，软件模式是用于解决一类计算机编程问题的一种已知可重用方法。软件模式显示构建元素之间的关系，但不规定问题或要求的最终解决方案。我们可以把软件模式视为软件代码与支持软件的系统之间的中间结构体现。以往的软件模式缺乏一个关键的宏观架构表述：安全控制措施指南。开发人员通常不会自行开发安全解决方案，而是选择依靠从平台或应用程序继承的功能，只有在迫不得已的情况下才会创建自己的安全控制措施。用加盐的散列函数给保存的口令单向加密是开发人员自己构建安全控制措施的一个例子。历史上软件模式都是指导软件开发，并不使用信息技术（IT）安全控制措施。

IT 安全控制措施提供了调节和管理应用系统行为的手段。IT 安全控制措施是一种抽象说法，表述了与应对感知风险的预防、检测或纠正对策相应的底层技术、管理或物理能力。IT 安全控制措施的目标是把潜在风险降低到可接受、无害或无关紧要的水平。控制目标也是一种抽象说法，但不同之处在于它是对以特定方式实施控制措施所要达到的预期结果或目的的陈述。控制目标通过使用一个或通常的一组控制措施实现；后者便是所谓的深度防御——用不同类型的多个控制措施协同预防、检测和/或纠正次优运行状态。控制目标源自一个控制框架，后者是可用于帮助业务流程负责人履行防止信息丢失职责的一组基本控制措施。多层布局的控制措施可以在应用程序生命周期的不同阶段以及应用程序运行环境的不同层面发挥作用。IT 安全控制措施从新软件创意诞生，到构建、部署，再到平台运行的所有阶段一直存在；这些阶段构成了所谓软件开发生命周期。

安全专业人员确实在推动开发人员左移，但是应用程序安全领域要求的专业水平与 IT 安全平台专业人员技术水平的差异越来越大，距离也越来越远。机构可以通过培养或雇用具有开

拓精神的软件开发人员或者使用由不同领域专业人员组成的角色组合管理技能上的差距。在整体式应用程序下，管理技能差距是可以实现的，但是分布式应用程序设计的实体化会扩大竖井式组织结构部门的控制范围，从而进一步加剧技能差距和所有权之争。

表现不同且论述或探索最少的是企业平台层面——这里有多个竖井式组织结构部门使用 IT 控制措施（网络、安全、服务器、存储、消息传递）——与软件开发层面（安全软件开发生命周期和安全测试自动化）之间的空间。安全控制措施叠加可以把企业平台层面与较低的软件开发层面联系到一起。安全叠加的使用代表了一个适合可扩展安全架构角色的领域，其中保密性、完整性和可用性可扩展到把弹性和隐私问题也包括进来。随着世界转向使用以软件为中心的抽象化方法以及企业接受软件中心主义，依赖不仅要求服务具有弹性，而且还要求可通过整个人机交互过程中自我管理数据访问能力实现完整的隐私。安全架构作为一种完整思路涵盖了人、平台，外加软件。

安全架构代表了企业架构中专门解决信息系统弹性问题并为满足安全要求的功能提供架构信息的部分。微服务语境下的安全架构不仅在现有平台和软件架构上引入了控制措施叠加的概念，而且安全架构师有责任和义务划出一条界线，把体现在平台层面的控制措施叠加与体现在软件本身的控制措施叠加区分开来。作为一种完整的安全架构思路，安全叠加是用于降低风险的多项控制措施的集合体，是管理、技术和物理控制措施的组合。

全面考虑问题的安全架构师往往会先构建威胁模型，然后才把控制措施运用到复杂的解决方案架构之中。威胁建模是把控局面的一种方式。红蓝对抗、STRIDE、攻击树、Trike、VAST、PASTA 和 ISO-31010 Delphi 是风险识别方法的示例。威胁建模在很大程度上属于围绕着人展开的一种活动。最好的分析产生于各有侧重的多样化人群，这样的群体对攻击面各个方面的深入体验，远远超过一个人的单独认知。即便是分析受到的系统性冲击，群体分析也不太容易变得脆弱。强健的威胁模型来自于对当前状态、制度历史和想象力的深入的行业纵向认识，以及选择一种适合问题的分析方法，而不是安全架构师的方便程度。让威胁模型契合战术实施范围并避免空幻、存在主义和黑天鹅场景至关重要。威胁模型得出的结果会使 IT 安全控制措施的落实合法化，从而形成一个可以抑制已知或假定风险的强大叠加。在这一点上，安全架构不同于解决方案架构，因为它可以减轻乃至消除在拟议的解决方案架构中发现的潜在技术、管理或物理风险。

一个微服务应用程序不会把每个设计模式和每个安全控制措施叠加全部囊括其中，但是会

包含那些被认为对于有效设计不可或缺，可以解决客户问题的软件架构模式和安全叠加。而这正是软件层面实现扁平化并融进平台层面的结合部。在微服务架构风格中，任何解决方案除非在每个所用模式都配备了相应的安全控制措施叠加，否则都谈不上概念完整。所有叠加都与模式配套，才算解决方案架构构建完成。模式与叠加组合到一起，构成了微服务应用程序的安全控制措施状态。

4.2 微服务架构模式简介

为了便于指导安全控制措施叠加对微服务的施用，通用微服务架构模式用两个分支形成了两个不同的视角。第一个视角着眼于企业层面。企业层面包含了可协助实现微服务架构治理的信息技术资产。企业期望减少控制措施状态的变数。自定义编码、生产状态变通方案和一次性修改都会增加开发成本。技术负债（如以增添安全设备形式出现的技术负债）、在设计中引入会限制灵活性的紧耦合等，可能会妨碍基础设施作为代码部署的能力，从而形成没有必要的持久存在。企业环境期望软件开发尽可能多地继承安全控制措施，以防开发团队在编制应用程序安全指南的过程中出现可变性和不可靠性。安全叠加同时存在于企业层面和软件层面。API 注册中心处理已完成开发的微服务的清单和版本以及主导服务供应商和第三方集成。服务存储库（存放 API 规范、模板、代码脚手架、用于构建过程的构件等）在微服务开发过程中建立统一性、自动进行静态和动态安全测试，并把微服务引入容器，最终通过公共管道交付合为一体的功能（例如会先触发安全检查，然后触发配置管理资源部署计划的 Jenkins 操作）。理想状态是，按企业层面的要求在执行引导进程的过程中交付平台层面的安全钩、调用和集成，这样可以避免在运行期间不得不进行的修改。

第二个视角着眼于软件层面。图 1 是分布式微服务应用的分解图，这种状况存在于软件层面，是最接近软件代码的表现方式。该图描述了合成的分布式微服务应用程序的一般性图景。它像是一张 X 光片，在上半部分显示主要的可继承的企业层面的系统集成，在下半部分显示微服务组件。各类机器客户端（浏览器、物联网、移动、API 集成）和物理消费者通过企业层面访问分布式微服务应用程序提供的功能。API 控制着客户端使用的表示逻辑，并提供一项且只有一项业务功能，同时包含用来控制客户端可以从暴露的 API 获取哪些内容的其他业务规则。API 可以整理来自多个来源的数据，并且可以访问安放在托管微服务 API 的容器外面数据卷上的数据。微服务利用企业层面的服务和软件层面的 API 网关服务在容器之间平衡负载，允许多个微服务实例在生产中同时运行。在软件层面的微服务环境下，由 sidecar 服务代表公用程序

服务处理与微服务中存在业务逻辑无关的任务。每个微服务 API 都有一个 sidecar 与之配对，而在集群容器环境中，sidecar 根据服务通信策略和安全策略交付服务的手段。通信流的主要组成机制是网络访问控制逻辑。控制会以虚拟局域网、基于策略的路由选择、基于上下文的 ACL、特定网关逻辑和软件定义的网络的形式出现。这些企业层面执行方案中的每一个都自带安全叠加权衡。构成通信流和控制流的主要手段，是企业层面严格管理的授权，以及携带被加密客户端权限（即 OAuth）、外部管理的托管凭证（如平台层面机对机凭证托管和管理）和 mTLS（企业层面相互传输层安全证书管理）的访问令牌。安全控制措施叠加可以在一个层面内以及跨企业和软件层面发挥作用——从而实现深度防御。安全控制措施叠加同时存在于两个层面。

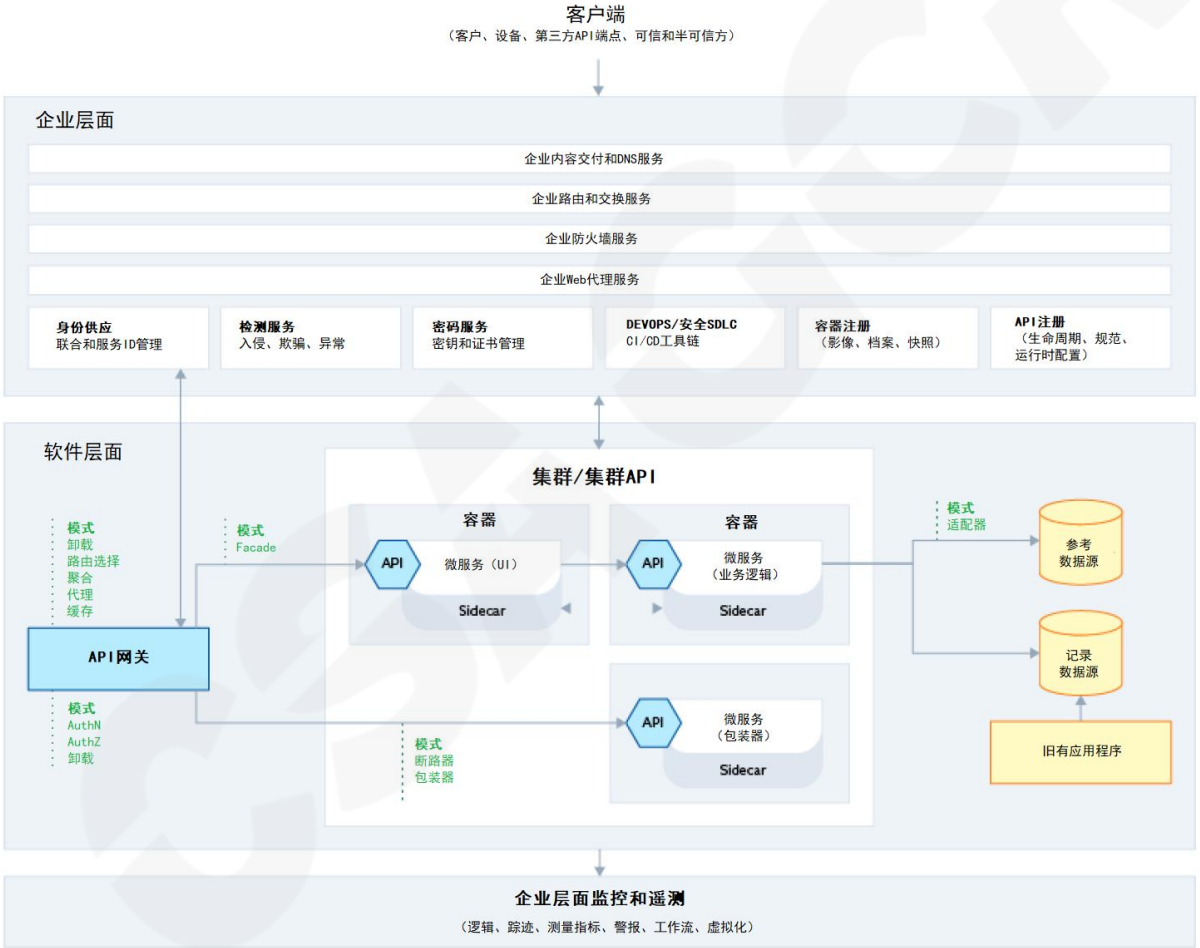


图4-1 微服务参考架构——企业层面和软件层面

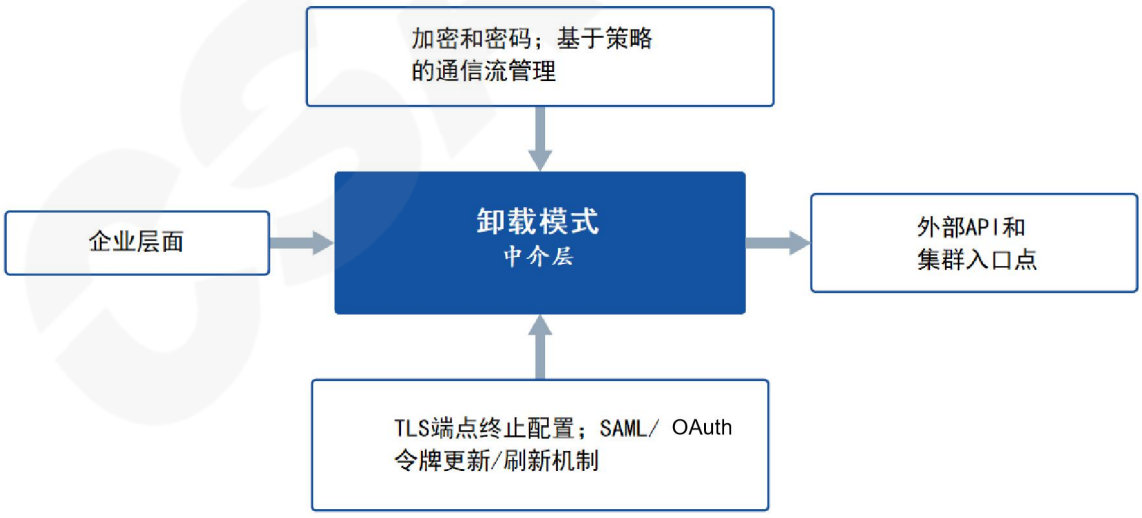
5. 微服务架构模式

5.1 模式

以下架构模式适用于将微服务开发成业务应用。研究这些模式时一定会注意到，正是许多模式的共同作用，构成了一个安全的系统。尽管最初可能要从某一个模式（如身份验证）入手，但必须搞清，这些模式是怎样通过彼此交互支撑起一个安全而富有弹性的业务解决方案的。

5.1.1 卸载(Offload)模式

卸载是针对具体环境的一个通用数据流动作。卸载可以与 API 网关功能紧密耦合，例如通过内核软件或加速器硬件提供 TLS 密码终止功能，从而使后端设备不必管理 TLS 连接。卸载也可以与数据访问层紧密耦合，在这一层把数据写入分散到多个同时进行的数据提交动作中，从而提高数据写入或读取的速度。卸载还可以应用于身份验证和授权。一般来说，被卸载的功能是常被许多其他服务作为共享服务使用的功能。如果选择卸载一项服务，就表明一个资源不必再被微服务 API 纳入它的代码库。



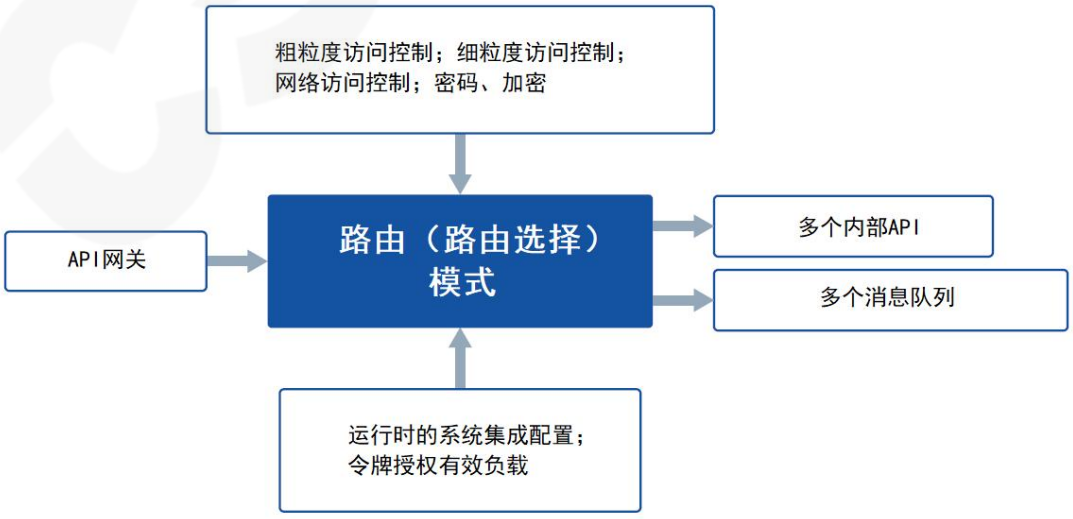
卸载模式	
版本	1.0

模式目的	展现巩固技术能力的机会
层面位置	企业（平台）层面
结构描述	API网关是外部流量进入微服务应用的入口。
行为描述	卸载TLS证书托管；TLS终止；AuthN和AuthX请求中介服务
数据特性	传输中的数据
主要依赖性	平台层面与IAM平台、证书管理平台相互连接;上游入口堡垒防火墙;上游负载平衡平台
次要依赖性	容器集合体（container fleet）；相邻的安全日志相互连接
内部事件/消息传递需要	HTTPS v2/3；AS2
外部事件/消息传递链接	API层面日志；网关层面系统日志；AuthN和AuthX消息交换
事件响应行为	发送，不接收。没有转换；原始机器输出
共同的上游链接	防火墙；全局和/或本地通信流负载平衡；网络间ACL
共同的下游链接	发证机构；配置管理；API注册中心；集群管理API安全环境；机器ID/服务ID凭证托管
Ops安全回接	API性能监控；syslog监控；机器健康监控；异常检测
DevSecOps回接	安全SDLC；API注册中心；Swagger/YAML API定义；AuthN和AuthX控制

评估方法	API性能趋势分析；SIEM日志分析并与上游事件关联；跨API访问端口和协议的异常关联
控制措施叠加的特性	可从平台继承，而不是内置于API规范或容器基础设施。
复合状态（独有/通用）	通用；其他模式也使用这一叠加。

5.1.2 路由（路由选择）模式

当一个端点需要暴露背后的多项服务时，可使用路由模式，根据入站请求路由请求和消息。路由选择策略和路由通信流管理被嵌入服务网格和/或使用sidecar服务。如果服务网格没有嵌入路由策略和通信流管理，则需要把API规范与有关消息队列设计的路由逻辑结合到一起实现API拓扑（即“谁可以与谁交谈，谁可以从谁那儿获得消息”）。以往，大型分布式整体式传统应用程序依靠消息队列传输交易信息，以便应用保持状态。如果没有服务网格可供使用，微服务将可能不得不拥有路由选择逻辑，以此执行原本该由sidecar服务执行的公用程序功能。另外，路由选择模式（或功能）可以设置在API网关上。路由模式可以存在于硬件或软件中。

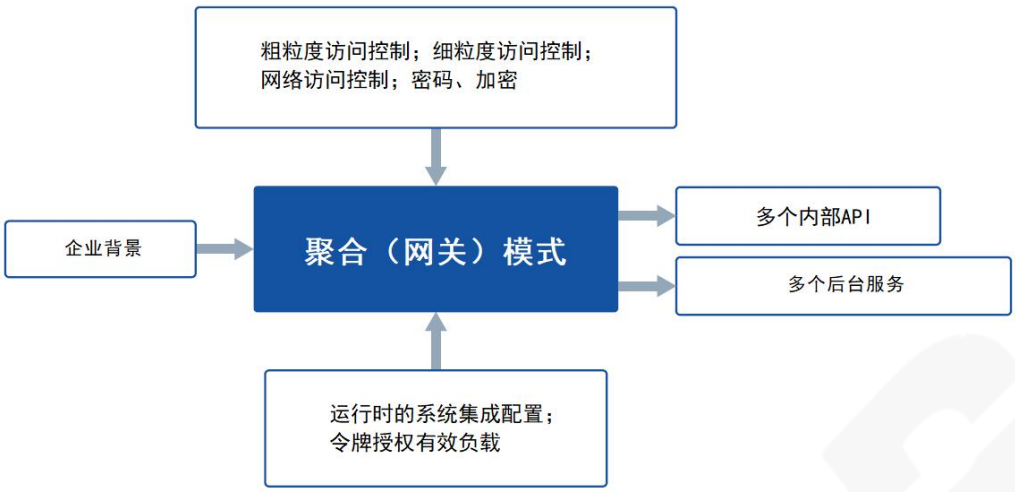


路由模式	
版本	1.0
模式目的	根据请求中的数据，如识别身份、来源、客户端类型、请求主机值、请求URI路径元素等的请求标头，将请求通信流路由给同一服务的不同版本或实例。根据通过源IP地址确定的请求源在有限时间内将请求路由给服务的一个新版本，便是路由选择模式的一个用例。 路由目的地也可以是消息队列而不是同步服务端点。
层面位置	企业（平台）层面
结构描述	根据路由策略配置所定义的请求或消息数据，将请求路由到服务的特定版本或实例或消息队列。
行为描述	根据请求数据和路由策略配置对入站请求进行评估，确定目标服务实例的目的地或消息队列。
数据特性	使用中的数据/传输中的数据
主要依赖性	API网关、经过配置的服务网格功能、sidecar等路由组件必须可用。
次要依赖性	路由目的地或消息队列必须可用。
内部事件/消息传递需要	目标服务和消息队列目的地的健康和可用
外部事件/消息传递链接	服务网格日志和遥测
事件响应行为	响应目标服务不可用的自适应或后备路由

共同的上游链接	IAM平台、网关平台（API，负载平衡，基于策略的路由）
共同的下游链接	队列、数据存储库、其他内部API
Ops安全回接	API网关可用性 DDOS预防 扩展或节制通信流以确保可用性
DevSecOps回接	SAST——服务网格配置审计 DAST——服务端点的AuthN和AuthZ测试 IAST——剔除假阳性结果
评估方法	路由功能的可观察性、遥测和日志记录。日志记录可提供实时可见性和可观察性。
控制措施叠加的特性	预防性——DevSecOps控制、DDoS预防 检测性和纠正性——扩容/收缩以确保可用性
复合状态（独有/通用）	通用

5.1.3 聚合模式

聚合模式接收并向多个微服务发出请求，然后把对后端服务发出的多个请求合并成一个请求，用以响应初始请求。当许多设备向一个中心点发送交易消息和活动日志时，往往会在云边缘发生软件定义的聚合。其他模式可能会在网关聚合背后发挥作用，如 Facade 模式、代理模式和断路器模式，具体由应用目标和通信特点决定。这些设计模式有类似的结构，但是使用它们的意图或目的各不相同。

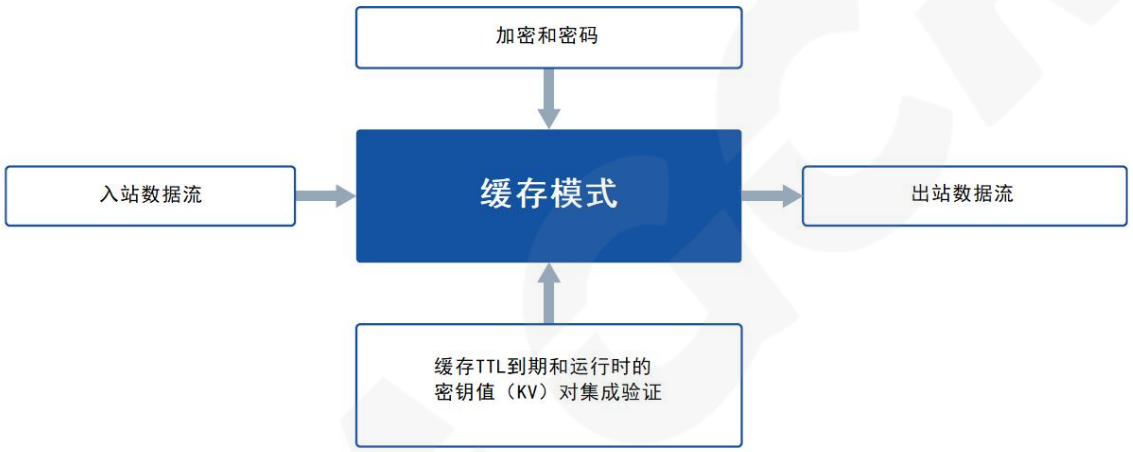


聚合模式	
版本	1.0
模式目的	聚合（网关）模式的目的是减少应用程序对后端服务提出请求的数量，并提高应用程序在高时延网络上的性能。
层面位置	企业（平台）层面
结构描述	聚合（网关）模式用于将多个单独的请求聚合成一个请求或响应。
行为描述	当一个客户端需要向各种后端系统发出多个调用来执行一项操作时，聚合（网关）模式会很有用。
数据特性	当聚合器合并来自多个终端的数据集时，使用中的数据可能需要得到安全保护。否则，聚合器可能只在请求者与响应者之间传输数据。
主要依赖性	网络（网络可能会带来严重时延）
次要依赖性	可用性和接近性（将与这一模式通信的各种系统的可用性）

内部事件/消息传递需要	聚合器模式可安全验证请求者的身份并传递一个访问令牌。服务可验证请求者是否得到授权可以执行所涉操作。
外部事件/消息传递链接	网络安全控制措施（网络访问控制——AuthN、AuthZ、通过加密保护静止数据等）
事件响应行为	确保聚合网关具有可满足应用程序可用性要求的弹性设计。 聚合网关可能是一个单点故障（SPOF）点，因此应该具备处理负载平衡的良好能力。 聚合网关应该配备有适当的控制，可确保来自后端系统的任何响应延迟都不会造成性能问题。
共同的上游链接	配置管理；API安全；策略管理和执行
共同的下游链接	维护、运行时间/停机时间、集成、测试、漏洞扫描和打补丁
Ops安全回接Tie-back	API性能监控；系统日志监控；健康监控；异常检测、事件管理、反恶意软件/病毒、漏洞抑制、补丁
DevSecOps回接	安全SDLC；敏捷、更简单/更灵活的开发和测试周期、持续集成/持续交付（CI/CD）管道
评估方法	日志记录/监控；警报、基于授权失败条件的计量警报、SIEM事故和事件管理
控制措施叠加的特性	网关可改善并直接影响应用程序的性能和规模。 预防性：漏洞抑制、打补丁 纠正性：事件响应 检测性：性能监控、健康监控
复合状态（独有/通用）	通用

5.1.4 缓存模式

应用设计中的缓存通常是用来满足提高可用性、改善应用性能和/或减少后端数据读写的要求的。缓存可以是嵌入式的，也可以是分布式的。主要缓存模式有许多衍生。如果一项业务操作对于后续使用具有持续的价值，可以考虑把结果缓存起来。例子包括按交易定价的数据元素，如可计量的外部接口等。第二个例子是消费者与现代分布式微服务应用中的许多外部 API 交互时的会话授权。把数据缓存在同一地点的同一层中可以简化调试，从而避免出现多个缓存实例彼此不同步的情况。



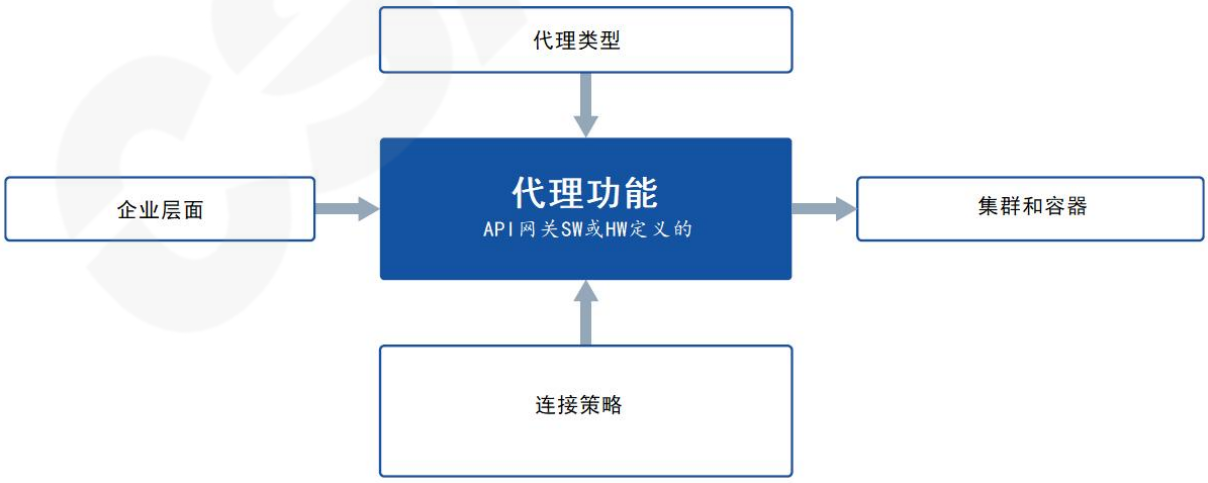
缓存模式	
版本	1.0
模式目的	提高性能、可扩展性和可用性
层面位置	软件层面
结构描述	把被频繁访问的数据临时复制到离应用更近的快速存储中。

行为描述	通过快速数据交付模式提高性能，以防I/O数据连接出现严重问题；利用缓存的凭证提高可用性，以防应用会话中断。缓存可以避免出现因快速检索数据而产生的时延情况，从而消除了频繁扩展的需要。
数据特性	使用中的数据（内存）
主要依赖性	最终一致性、缓存大小、基于缓存点击率或近来被用最少策略的回收策略、冷启动、缓存集合体（caching fleet）停运、通信流模式的变化、扩散的下游停运、网络时延、请求合并
次要依赖性	通信协议的类型、缓存服务管理
内部事件/消息传递需要	内部API、SOAP、RPC、ICP、HTTP/S
外部事件/消息传递链接	AppDev调试（基于合规要求的更严格调试）
事件响应行为	发送/接收/排队/消息传递、事件驱动、异步/同步
共同的上游链接	全局和/或局部通信流负载平衡
共同的下游链接	内部API；与记录的数据源或参考的数据源紧密耦合的端口和协议
Ops安全回接	API监控；异常检测；安全监控流程
DevSecOps回接	用于缓存大小/内存超限问题的SAST、DAST、缓存客户端库的漏洞打补丁
评估方法	API加固指南（如果使用了容器，则策略即代码形式的加固指南）、网络时延和性能、负载测试、测试缓存失效和欺骗的方法

控制措施叠加的特性	API跟踪/调试、密码叠加（传输、处理和使用中的数据）、加速新缓存实例（更短的TTL、动态扩展内存配置）以避免缓存丢失、缓存失效和负载平衡
复合状态（独有/通用）	通用；其他模式也会使用这个叠加

5.1.5 代理

代理模式是一种靠硬件启用或靠软件定义的结构体，在机器之间充当可读数据流中介。代理的能力可体现在不同的设计取向上。代理代表另一个人“发声”或“行事”，要么隐藏请求者，要么承担请求者授予的责任。在机器对机器的数据传输中，代理可以是透明的，代理功能在请求者和内容提供者之间起中介作用，或者在发起的 API 端点和响应的 API 端点之间起中介作用。透明代理功能拦截数据并执行缓存、卸载或重定向等操作，但是不改动数据流（修改数据流是适配器模式的责任）。反向代理功能代表提供者做出响应，从而隐蔽提供者的身份。此外还有其他几种类型的代理。分布式微服务应用程序通常使用透明和反向代理。



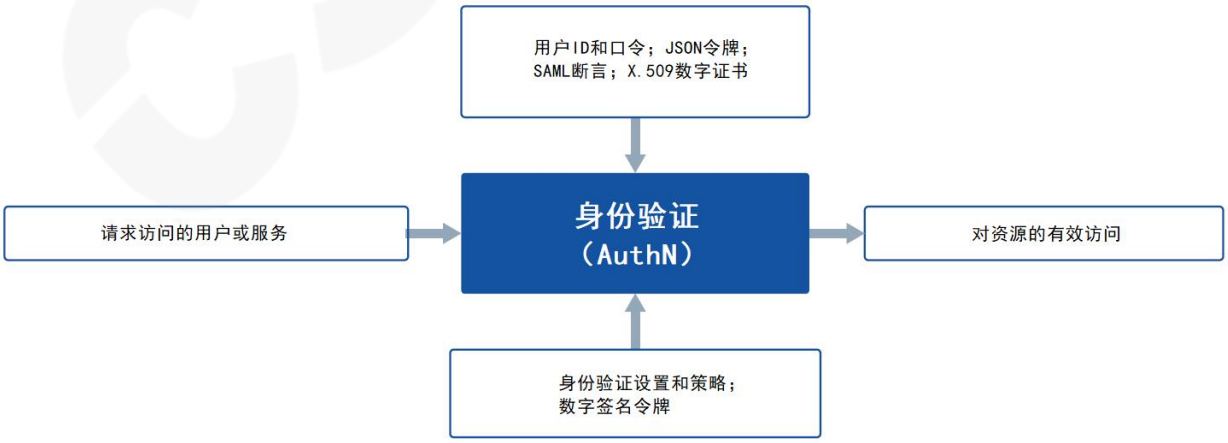
代理模式	
版本	1.0
模式目的	在服务消费者和服务提供者之间建立中介联系
层面位置	软件层面
结构描述	代理接收来自各种服务消费者的输入，充当API或容器的安全检查点。然后根据安全策略处理来自服务消费者的输入，决定是接受、拒绝还是终止指向服务提供者的连接。
行为描述	代理模式可以扩展或变成专用，以纳入网络或访问控制代理（即专注于网络第三或第四层的源、目标、端口和协议属性的代理）、身份代理（即执行身份验证和授权的代理），等等。代理也可以与其他模式协作（例如给入站请求添加元数据，以帮助促进适当路由）。收到的数据流输入会依照安全策略接受评估。假若没有违反策略，服务消费者提出的请求将被传递给服务提供者。
数据特性	代理应该把通信从服务消费者传递给服务提供者。它不应存储数据，评估暂时排队的请求的可能情况除外。如果需要，代理可以（但不是必须）用元数据充实数据流。
主要依赖性	代理要求网络连接、运行环境和应用程序端点能够正常运行并做出响应。
次要依赖性	代理应该持续执行安全策略，即便与策略定义组件断开连接也是如此。请注意，当无法获得最新版安全策略时，所执行的策略可能会是一个过时或针对特定时间点的版本。

内部事件/消息传递需要	名义上运转正常的代理可以独立运行，但可能会从与以下模式的交互中受益： AuthN——检查身份验证凭证是否仍然有效并且没有被撤销。 卸载——提取被解密的TLS流，检查策略合规。 路由/路由选择——确保服务消费者的元数据（如原始源IP地址）会被传达给路由/路由选择模式。
外部事件/消息传递链接	代理应该能够向安全分析解决方案（例如SIEM）提供事件的细节和决定，以便进行企业安全事件关联和事故调查。 理想状态是，代理能够获取威胁情报或有关可疑或受损用户、设备和端点的高可信度清单，从而使安全策略合规决定做到有理有据。
事件响应行为	代理的响应有以下三个选项： 接受：允许从服务消费者到服务提供者的数据流或连接，并作为可选项对消费者做出回应。 终止：阻止连接，不回应消费者。 拒绝：阻止连接，同时必须回应消费者。
共同的上游链接	AuthN、卸载
共同的下游链接	路由/路由选择
Ops安全回接	基础设施即代码（IaC）、配置管理、安全信息事件管理（SIEM）、负载平衡器、防火墙
DevSecOps回接	用于一致和可重复执行的持续集成/持续交付（CI/CD）管道、用于安全执行的静态应用安全测试（SAST）、用于策略执行的动态应用安全测试（DAST）

评估方法	分析被允许和被拒绝的通信，与企业数据关联，确认代理是按要求决定允许/拦截通信的，并主动尝试破坏代理（例如通过红茶测试/渗透测试）。
控制措施叠加的特性	预防性/纠正性——访问控制 检测性——给企业关联增加视角。可能有助于事件调查 监管性——可能必须满足审计员或企业客户的要求
复合状态（独有/通用）	通用。具有独有的控制措施叠加。 代理控制措施叠加可能是通用的，与其他模式类似。

5.1.6 AuthN（身份验证）模式

AuthN(身份验证)模式可确保访问微服务的服务和用户身份与其声称的一致。例如,OpenID Connect 扩展了 OAuth 2.0 的身份验证规程。OAuth 2.0 是一个纳入了身份验证规程的授权框架。身份验证服务器把身份令牌提供给依赖方，内含有关身份验证和身份信息的声明（通常在通过门户登录之后）。



身份验证模式	
版本	1.0
模式目的	建立信任并验证被声称的实体身份的过程
层面位置	软件层面和企业（平台）层面
结构描述	客户端——无论是用户还是其他微服务——在使用服务之前都需要识别身份
行为描述	在授权模式决定允许访问资源之前对用户或服务进行身份验证。身份验证由两个相关概念组成，“识别”用户或服务身份，以及“验证”用户或服务确实是他们（它们）自称的人或物。
数据特性	身份验证模式可以生成有关用户或服务是否已被核实身份的布尔回应。其他回应可能还包括一个令牌或断言，表明实体就是他们自称的身份。身份令牌或断言可供API端点、API网关、MESH和ISTIO使用。
主要依赖性	身份验证依靠存储凭证的可信身份存储库（如Active Directory、LDAP等），也可能依靠公钥基础设施（PKI）给身份断言或令牌签名以及对这些令牌或断言验证。
次要依赖性	根据所用身份验证协议，可能要有一个客户端（依赖方）、用户（资源拥有者）和授权服务器签发访问令牌（IDP）。
内部事件/消息传递需要	OAUTH的令牌和客户端事件消息、触发警报（调查和控制）的OAUTH策略、来自ISTIO组件的日志、AuthN和AuthZ消息交换
外部事件/消息传递链接	AppDev调试、SIEM

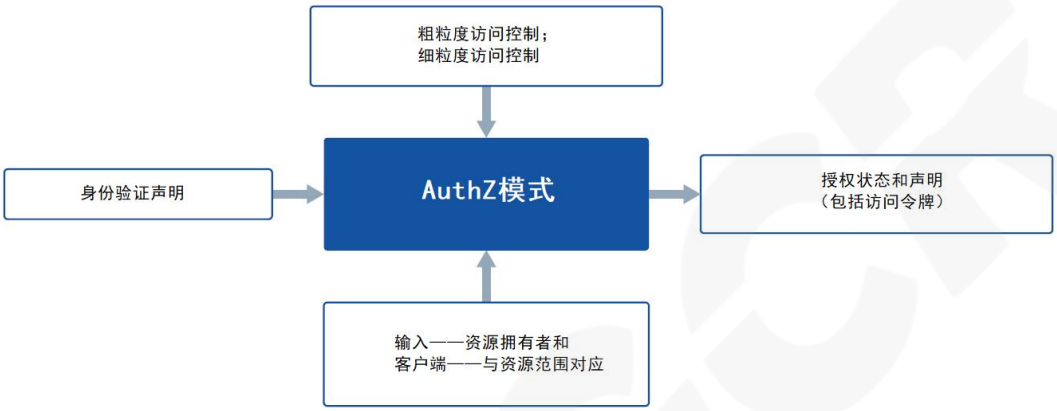
事件响应行为	在执行策略过程中做出“允许”或“拒绝”响应。准备用日志记录身份验证事件，供企业事件关联之用。如果某一特定用户的过多身份验证事件超过某个阈值（例如60秒内超过三次身份验证失败），则可能必须生成警报。
共同的上游链接	防火墙；全局和/或本地通信流负载平衡；网络间ACL
共同的下游链接	发证机构；配置管理；API注册中心；集群管理API安全背景；机器ID/服务ID凭证托管、代理、授权（AuthZ）
Ops安全回接	API性能监控；系统日志监控；机器健康监控；异常检测
DevSecOps回接	安全SDLC；API注册中心；Swagger/YAML API定义；AuthN和AuthX控制
评估方法	API性能趋势分析；用于异常检测和事故响应的SIEM日志分析和关联
控制措施叠加的特性	预防性——访问控制
复合状态（独有/通用）	通用；其他模式也使用这一叠加。

5.1.7 AuthZ（授权）模式

AuthZ（授权）确定经过身份验证的用户可以做什么或授予他们什么权限。把这些权限从授权服务器传递到资源服务器的标准方法之一，是将范围编码到 JWT（JSON Web 令牌）中。AuthZ 服务根据用户的角色/属性决定授予什么权限。所涉步骤归纳如下：

- AuthN 规程执行完毕后，客户端连同 AuthN 服务授权令牌一起，向 AuthZ 服务发出授权请求，这一请求将被传递给 AuthZ 服务。
- AuthZ 服务验证授权令牌后，将根据用户/角色的请求回复一个访问代码；这个代码通常采用 JWT 格式，内含“被准许的权限”。
- 执行完 AuthZ 规程后发送给 API 网关的请求在“授权标头”中包含了一个 JWT。

AuthZ 用于客户端请求访问资源的授权或许可权。客户端以授权代码的形式向 AuthZ 服务发送用户身份验证的结果，如果验证成功，AuthZ 服务将回复一个访问代码，其中通常嵌入了范围信息。这个范围应该与被资源服务器准许对资源进行操作的细节对应。访问令牌通常以 JWT 格式编码。



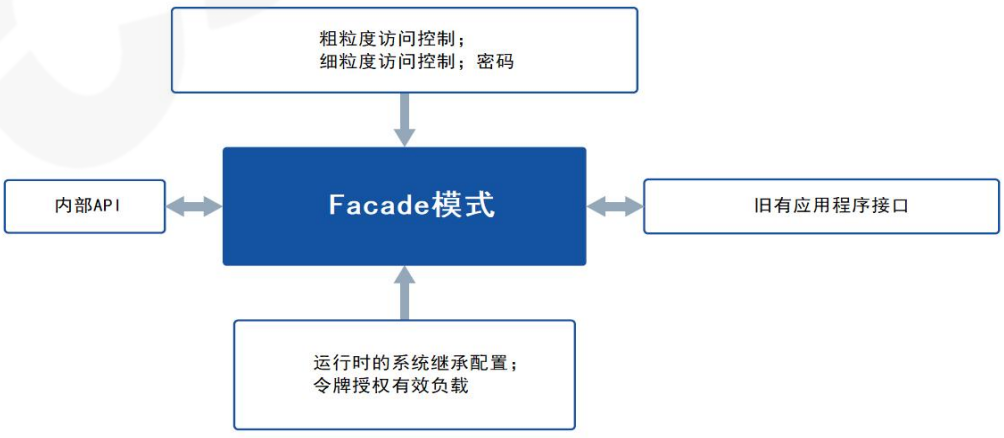
授权模式	
版本	1.0
模式目的	根据AuthZ服务中配置的授权策略，确定授权客户端可以代表用户执行的操作。
层面位置	软件层面和企业（平台）层面
结构描述	AuthZ机制返回信息，用于确定用户是否可以按自己提出的请求进行操作。
行为描述	AuthZ服务通过客户端接收资源拥有者的身份验证声明（通常以授权代码的形式），然后在访问令牌中以规定范围的方式把资源拥有者和客户端与得到授权可以访问的资源对应。

	注：如果授权失败，则应终止请求。请求者需要重新发出授权请求。
数据特性	用户名、客户端ID、授权令牌和可能的范围
主要依赖性	AuthN服务、访问/JWT令牌
次要依赖性	HTTP POST/GET、TLS、证书、访问控制
内部事件/消息传递需要	验证输入，其中包括客户端凭证和授权代码。查询授权策略，以从范围的角度将资源拥有者和客户端与得到授权可以访问的资源匹配。
外部事件/消息传递链接	将输入验证和授权策略查询的结果记录到日志中，供安全监控和审计之用。
事件响应行为	如果输入验证和授权查询成功，则返回一个访问令牌和得到授权的范围，HTTP状态码为200。 如果失败，理想情况下将返回以下两个HTTP状态码中的一个： HTTP 401：未得到授权。出现了与身份验证相关的问题。 HTTP 403：禁止。客户端被拒绝访问资源。 其他状态码可作为可选项提供给客户端。
共同的上游链接	身份验证服务、上游微服务（服务到服务）、边缘服务（路由、安全）、卸载
共同的下游链接	下游微服务（服务到服务）、API网关、内部服务、集成、Facade模式、聚合模式
Ops安全回接	API性能监控；系统日志监控；健康监控；异常检测、事故管理、反恶意软件/病毒、漏洞抑制、补丁、测试、维护、正常运行时间/停机时间

DevSecOps回接	安全SDLC；敏捷、更简单/更灵活的开发和测试周期、更小的更改、快速开发（持续集成/持续交付）
评估方法	日志记录/监控；根据授权失败的情况发出警报、SIEM事故和事件管理
控制措施叠加的特性	访问控制是预防性控制。
复合状态（独有/通用）	通用；其他模式也会使用这一叠加。

5.1.8 Facade 模式

在经典软件架构和设计范式中，Facade 是一种结构化模式，用作底层更复杂软件编码的前端接口。Facade 可以遮蔽复杂组件之间的交互，并且作为一种简化手段为客户端应用程序和接口提供方便，用以对更复杂的软件交互提出委托调用。Facade 通过增添层级加大深度，把软件子系统构建到各个层级（即可以访问其他较低层级 Facade 的 Facade）之中。对单体分解时，Facade 可能并不是展示新研发微服务的最佳选择。从长远看，使用代理模式恐怕才更合适，因为代理模式可以在不再被需要时撤销，而且与 Facade 模式相比，代理模式也更易于执行。



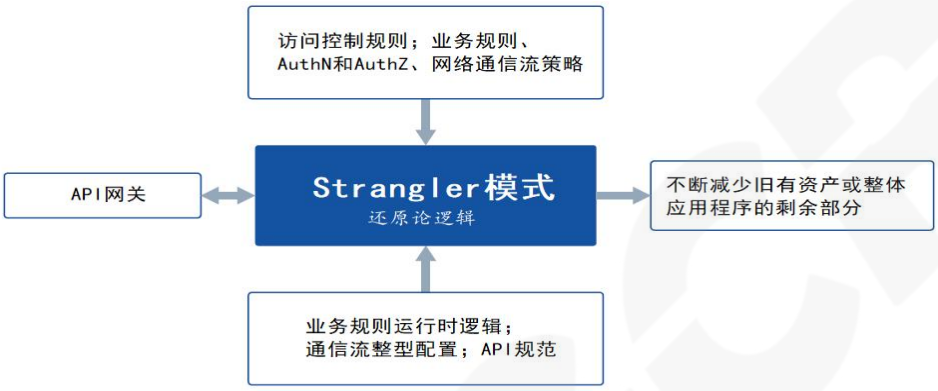
Facade 模式	
版本	1.0
模式目的	Facade模式的目的是允许在客户端与微服务之间有一个协议中介，以提供更好的用户体验。客户不必了解微服务，只需要通过Facade模式知道API和安全配置就可以了。使用Facade提供的粗粒度API可以简化客户端访问。
层面位置	企业（平台）层面
结构描述	安全视角下的Facade模式是支持API的底层微服务的API入口/出口点。从软件层面的安全角度来说，Facade是指是API从单个Facade接口（内部API）向许多不同微服务API（外部API）的转换。
行为描述	Facade模式是客户端与子系统交互并进行系统级别更改且不影响客户端代码的一种简单手段。客户端当前与子系统类之间属于松耦合，而不是紧耦合。
数据特性	考虑了服务的安全性、服务之间的通信、传输中和使用中数据的保护。
主要依赖性	特定的REST API（Facade与外部API和内部API之间存在相互依赖关系） 协议：（Facade需要熟知客户端和微服务的协议，因为它提供协议中介） 各种子系统的可用性：Facade可能是单点故障点，否则客户端无法与微服务通信。
次要依赖性	子系统的设计变更 各种子系统的可连接性/AuthN AuthZ（XACML、RBAC、ABAC）将为各种子系统建立客户端需要的权限。

内部事件/消息传递需要	传递给内部事件处理或消息传递的遥测、日志记录和可观察性信息
外部事件/消息传递链接	Facade通过向客户端适当公开API来对客户端提供接口，而不是将客户端与子系统紧密耦合。因此，客户端委托Facade代它们执行操作。
事件响应行为	性能降级时最好调用断路器模式。
共同的上游链接	微服务（外部API）配置管理；API安全；策略管理和执行、断路器
共同的下游链接	API客户端和API网关（内部API）配置管理；API安全；策略管理和执行。
Ops安全回接	IT安全控制措施、与基础设施即代码集成。API性能监控；系统日志监控；健康监控；异常检测、事故管理、反恶意软件/病毒、漏洞抑制、补丁
DevSecOps安全回接	CI/CD管道与代码安全工具集成，如snyk、stackhawk等。安全测试工具，如SAST、DAST、第三方漏洞扫描工具（其他例子还包括Twistlock和PureSec [PANW]）等
评估方法	控制验证。日志记录/监控；警报、基于授权失败情况的计量警报、SIEM事故和事件管理
控制措施叠加的特性	预防性：SAST、DAST、漏洞扫描 检测性：性能监控、健康监控、异常检测
复合状态（独有/通用）	通用，使用与聚合模式相同的控制措施叠加

5.1.9 Strangler Fig 模式

Strangler 模式以增量方式“扼杀”旧有应用程序的功能。采用 Strangler 模式时，较旧的应用程序会把特定功能移交给新的现代化代码库。最终，新代码库将替换旧系统的功能，而

旧有系统将退出服务。这种模式以“从旧到新”的过渡分解整体式应用程序，中介层软件架构状态用其他模式（例如代理和包装器模式）使新的微服务与剩余的旧有代码库交互。最终，整体式旧有应用程序以一种可控的方式分解和收缩，包装器和代理模式也可以就此退出。我们必须注意这一模式的代价因子。



Strangler 模式	
版本	1.0
模式目的	这一模式的目的是以渐进/增量方式将旧有后端系统迁移到新架构。整体式应用程序可以批量替代（绿场部署），也可以通过把某些功能从整个应用程序中分割出来并加以更新，使它们得以脱离整体的处理边界独立运行，从而保持功能“常青”。建议把这种模式与蓝绿部署配套执行，以此保证业务连续性计划。
层面位置	软件层面
结构描述	以增量方式用新应用程序或服务替换特定功能。这会形成两个独立的应用在同一URI空间中并存的局面。随着时间的推移，重构的新应用程序逐阶段“扼杀”或替换原始应用程序，直到最终可以关闭整个旧有应用程序。每个迁移阶段都会有一个路由器或逻辑来决定把特定功能路由给哪项服务（旧有的或新的）。

行为描述	旧有应用程序与微服务模式并行，直到旧有功能被完全替换。每个阶段都有一个路由器或逻辑决定特定旧有或新功能采用哪条路径或路由路线。Strangler模式还依靠事件驱动的架构模式[9]管理横跨旧有整体式组件和微服务的状态和数据。新的微服务域模型可能需要包含旧有域属性。如果需要，这些状态和数据还要通过双向同步加转换维护。客户端必须为最终一致性做好规划。 https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro/
数据特性	传输中的数据/使用中的数据
主要依赖性	发送给后端系统的请求有可能被人拦截。Facade的单点故障和性能瓶颈。最终一致性是REST JSON的一个折衷方案，特别是在横跨多个安全区或应用层的时候。如果这个要求是设计提出的硬指标，则必须靠缓存、把延迟写入与用户会话捆绑等功能来关闭或降低一致性。
次要依赖性	访问新的微服务和旧有整体式应用程序的后端数据源
内部事件/消息传递需要	交互过程和交互模式。根据应用程序或系统上下文，把注释[10]描述的消息传递模式用于Strangler应用程序，以提供数据和应用程序状态同步。
外部事件/消息传递链接	AppDev和数据库交易日志记录
事件响应行为	发送/接收/排队/消息传递
共同的上游链接	Facade模式、API网关、API代理

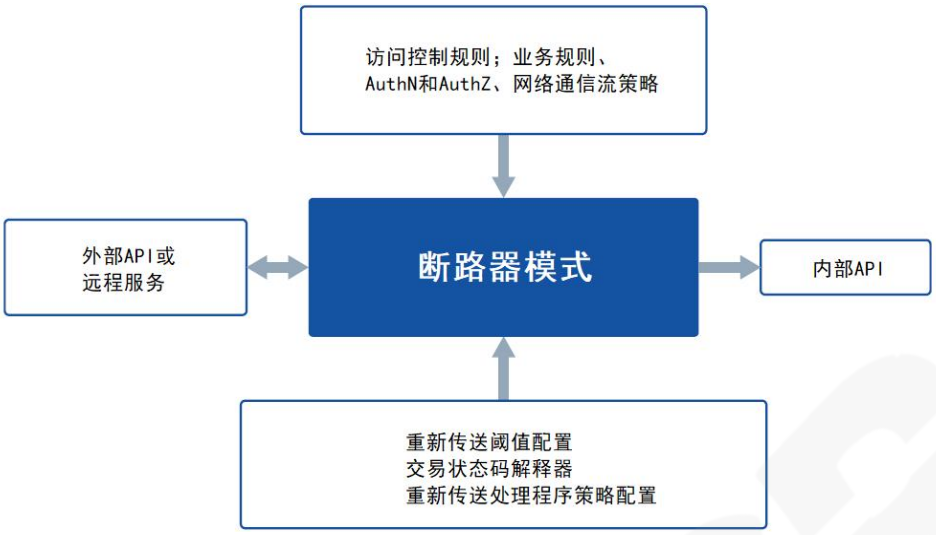
[9] IBM. Patterns in Event-Driven Architectures – Introduction. IBM Garage Event-Driven Reference Architecture. Retrieved August 11, 2021, from https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro.

[10] Enterprise Integration Patterns. Enterprise Integration Patterns - Messaging Patterns Overview. Retrieved August 11, 2021, from https://www.enterpriseintegrationpatterns.com/patterns/messaging/.

共同的下游链接	服务总线、API代理、事件适配器、服务总线队列/主题、适配器模式、Sagas模式
Ops安全回接	IaC（基础设施即代码）；策略即代码，由DevOps管道驱动的策略即代码（通常是Jenkins，但也可以是其他）
DevSecOps回接	预处理单元代码质量评审测试；95%测试覆盖率；在存储库层面（请求签收）单元测试过程中进行的管道驱动代码质量SAST测试；按构建要求执行系统集成测试过程中进行的管道化DAST测试；根据API（swagger或YAML）规范，依照清晰API拓扑结构管道化自动部署到容器环境中
评估方法	根据通过/失败基础设施策略即代码进行的管道化自动测试；API性能、负载测试
控制措施叠加的特性	检测性
复合状态（独有/通用）	通用，具有独有的控制措施叠加——通用

5.1.10 断路器(Circuit Breaker)模式

断路器模式是一种停止或限制请求-响应交互的方法，目的是防止当服务进入威胁状态或停止运行时发生更大故障。这一模式是预防故障扩大或造成系统完全停机（即关闭状态[closed_state]）并在软件层面驻留的一种手段。威胁信号可能是超过既定阈值的数据交换，也可能是超过既定阈值的连续故障。结果可能是连接“跳闸”，并在超时期间（即打开状态[open_state]）静默不语，同时向请求者回复一个错误；而超时到时后，模式可能会再次打开连接；或者，结果也可能是允许少数几个交易成功通过（即半开[half_open]）。如果失败第二次出现，会有第二次超时随之而来，直到微服务返回关闭状态（closed_state）。

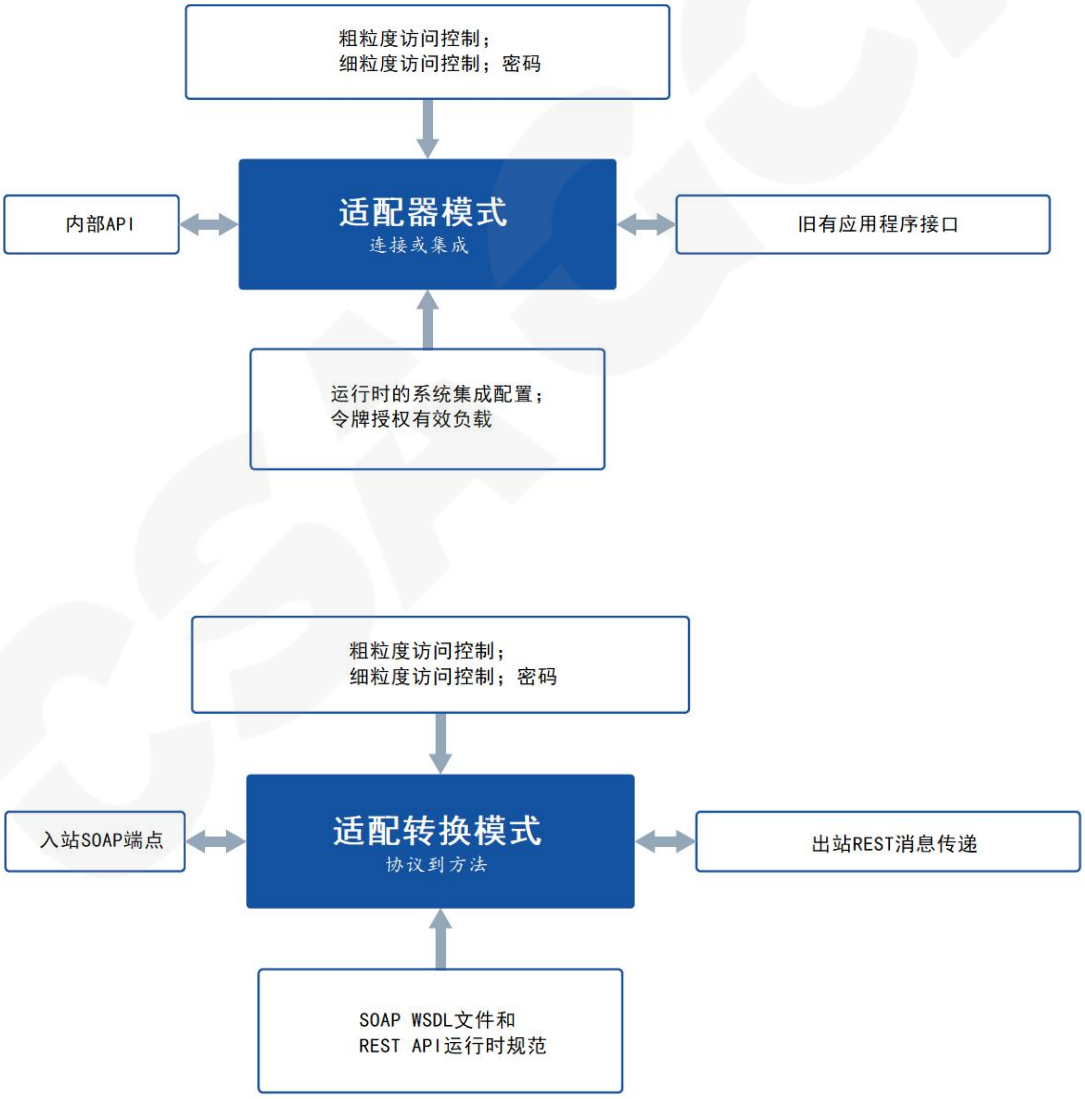


断路器模式	
版本	1.0
模式目的	构建一个容错和弹性系统，当下游服务由于网络或服务故障等资源问题而无响应时，系统依然能够正常生存。
层面位置	软件层面
结构描述	断路器模式监控下游服务请求-响应交互并检测故障。这一模式保护上游服务不受级联故障的影响，并以默认响应/错误或缓存的最后结果做出回复。它还检查下游服务的状态是否已从故障状态恢复。
行为描述	断路器模式允许上下游服务之间在系统处于关闭状态时进行请求-响应交互。当断路器观察到下游呼叫失败次数超过阈值时，断路器将断开连接，并在超时期间将状态更改为打开状态。断路器允许交互在规定的时间段后检查下游服务的连接或资源的可用性，即处于半开状态。如果下游服务成功允许交互，断路器将关闭连接并允许上下游服务之间进行请求-响应交互。 1. 监控下游服务请求和响应并检测故障。 2. 阻止部分或所有请求级联到下游服务。 3. 根据标准响应或错误条件响应请求。

数据特性	传输中的数据/使用中的数据
主要依赖性	最佳阈值配置、网络连接、服务可用性
次要依赖性	API网关，阈值配置
内部事件/信息传递需求	这一模式独立工作，但是可以利用重试模式恢复与下游服务的交互，并在断路器处于打开状态时利用缓存模式将最后的结果保存并提供给上游服务。
外部事件/消息传递链接	活动需要记入日志用于调试，并且还将触发事件通知，上报运维团队，以便对下游服务或网络故障采取适当措施。
事件响应行为	断路器打开连接时，这一模式以标准错误或最后结果形式响应上游服务。下游服务不可用时，回复HTTP状态码503。
共同的上游链接	代理、路由
共同的下游链接	重试、缓存
Ops安全回接	通信流监控、性能监控、服务可用性、配置管理
DevSecOps回接	持续集成/持续交付（CI/CD）管道、状态应用安全测试（SAST）、动态应用安全测试（DAST）
评估方法	对照各种阈值配置分析行为并模拟下游服务故障。日志和遥测分析。
控制措施叠加的特性	预防性、纠正性、检测性、抑制
复合状态（独有/通用）	通用

5.1.11 适配器（包装器/转化/转换）模式

适配器模式把不兼容接口无法理解的数据流转换为其他的表现形式。提出请求的客户端只能看到目标接口，而不看到位于接口后面的适配器。适配器这种结构化模式在两个独立的不可连接接口之间搭起一座桥，进而“包装”入站请求，以便于接收接口理解。适配器模式包装不兼容的协议以便于相互通信，或者包装数据格式以帮助数据帧从一种格式转换为另一种格式（即供应商第三方数据连接器或适配器）。如果适配器模式“修饰”了代码，则还会在运行时添加功能，但不修改底层结构（即用一个包装器类来避免在软件中创建额外的子类）。



适配器模式	
版本	1.0
模式目的	提供数据转化/转换功能接口（例如：在同一应用程序中将数据从一种格式转换为另一种格式，或者将数据从一个系统迁移到另一个系统）。提供迁移功能接口，但将数据传输到另一个系统可能会导致存储库更改（在服务或方案映射层面）。
层面位置	软件层面
结构描述	转换可以是显式或隐式（自动）数据转换。转换模式可能是系统的组成部分，由系统的应用层来转换数据类型，使它们可以放进数据库表，或者将一种API通信协议转换成另一种API通信协议。
行为描述	每种编程语言都有自己的转换规则。适配器模式的目的是转换数据类型、通信协议或方法。在将方法转换成Restful方法时，目的是期望交易和消息达到最终一致性。
数据特性	传输中的数据和静止的数据
主要依赖性	平台层面与IAM平台、证书管理平台的相互连接；内部核心网络；消息队列管理平台、上游入口堡垒防火墙；上游负载平衡平台。其他内部API
次要依赖性	需要与连接数据库的各种服务断开，不强制执行ACID（原子、一致、隔离、持久）
内部事件/消息传递需要	HTTPS v2/3；AS2。消息传递可以采用传统消息传递格式（SOAP）、专用API、CRUD RPC调用、使用URI和HTTP的客户端-服务器端连接器——REST、发布/订阅消息传递、gRPC
外部事件/信息传递链接	API层面日志记录；网关层面系统日志；消息交换的身份验证和授权，如RPC、gRPC、发布/订阅消息和公共API

事件响应行为	事件驱动的消息传递、向订阅者发布事件信息、异步、松耦合
共同的上游链接	堡垒防火墙；全局和/或本地流量负载平衡；网络间ACL
共同的下游链接	发证机构；配置管理；API注册中心；API安全上下文下的集群管理；机器ID/服务ID凭证托管。
Ops安全回接	API性能监控、系统日志监控、机器健康监控、异常检测
DevSecOps回接	安全SDLC；API注册中心；Swagger/YAML API定义；身份验证和授权控制、控制和限制API调用以及静态代码分析
评估方法	API性能趋势分析；SIEM日志分析和与上游事件的相关；API访问端口和协议间的异常关联。发布/订阅消息传递系统、支持流和事件队列以及内部和外部审计
控制措施叠加的特性	重新构建和重新设计不跨微服务共享表格的边界。用于表格上显性或隐性转换数据的转换模式。跨基础设施传递事件信息。执行可通过切换转换模式的API网关。迁移过程中可从平台继承控制。
复合状态（唯一/通用）	通用

6. 安全控制措施叠加

如前所述，软件开发的展开离不开以软件设计模式作为引导。安全控制措施叠加（overlay）是指由全套特定控制措施、控制措施强化和补充指南组成的离散集，可集成到架构设计进程之中，充当嵌入式既定管理、技术或物理要求。软件设计模式与安全控制措施叠加结合到一起会告诉我们，软件开发工作要把安全作为一个设计元素“内置”到软件产品之中，而不能只把安全当作到了最后才涂抹到软件产品上的一层外衣，使这一点到了日后成为必须付出极高代价才能做出改变的地方。就软件（具体来说是我们这里所讲的微服务）的设计而言，在任何软件开始开发之前就考虑采用安全控制措施叠加，会使我们有机会在架构意义上形成视软件安全为重

要元素的完整思路。美国国家标准和技术研究所特别出版物NIST SP 800-53第5修订版在SA控制措施系列（“开发人员安全和隐私架构和设计”）和PL-8控制措施系列（“安全和隐私架构”）的控制目标中提供了有关安全架构控制的具体指南。本文的远大目标是避免出现不得不把安全架构和设计强行加进已经开发完成的代码中的情况，而是提供一种方法，让有关软件安全控制措施的研讨在控制能力还是一张白纸的时候就与开发工作同速向前推进。

2001年，马里兰大学实验软件工程组（ESEG）的Boehm和Basili用一篇论文发布了他们的调查结果，介绍了会增加软件代码开发成品缺陷的行为以及由此产生的后果。二人得出结论认为，到了产品投产后再去查找和修复软件存在的问题，所需付出的成本是在设计过程中就把问题查找出来并加以修复所付成本的100倍[11]。对于那些不待软件开发完备或者不把漏洞（安全漏洞也是软件缺陷的一部分）弥补干净就匆匆把软件向外发布的“敏捷开发”（Agile）团队来说，这些缺陷必将重返开发团队堆积起来，并且还要与当前积压的下一版本的性能开发工作一起重排优先级——好不容易赢得的加分才过几周就毁之于一旦。缺陷会产生真正的“敏捷开发”风险后果，其中包括冲刺膨胀风险（冲刺中的点太多）、降低性能编码速度，以及可能没有在当初全面完成好开发工作的DEV（开发）部门带来不得不召回新软件进行修复的额外心理负担。软件质量不是一种状态；它是一个逐渐成熟的过程。通过把安全控制措施叠加嵌入软件要求，团队可以从安全要求中剔除被发现的部分自然波动（例如，随着时间的推移，由于添加、删除和修改一项要求而产生的百分比变化），从而朝着不断成熟的DevOps（开发-运维）迈出一步[12]，并逐渐转向形成一种从设计的早期阶段便就安全要求展开沟通和协作的文化。到了质量保障阶段才开始研讨控制措施，未免为时过晚。

尽管本文的内容可以对各种角色起到帮助作用，但它是针对安全架构的作用和架构师的理念量身定制的。本节微服务架构模式（MAP）附带一个作业练习指导（见附录D），旨在通过具体实例帮助读者为架构模式形成控制措施叠加。我们编制这个作业练习指导的目的，是为安全架构创建一种经济实用、轻量级、不会在设计过程中牺牲控制挑选基本原则的非统计分解/重组方法。下一节，第5章介绍的每种软件架构模式都会有一个与之对应的安全控制措施叠加（对应的形式包括：一对一、多对一、一对多等）。从设计的角度说，最初的概念经过扩展，已经

[11] Boehm, B., & Basili, V. R. (2001). Software Defect Reduction Top 10 List. Computer, 34(1), 135–137.

[12] V. Suma, B. R. Shubhamangala and L. M. Rao, “Impact analysis of volatility and security on requirement defects during software development process,” International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), Chennai, 2012, pp. 1-5, doi: 10.1049/ic.2012.0145.

演变成具有已知控制状态的软件架构构件（相关软件模式的集合）概念。而从逻辑上说，架构构件可以组装成解决方案构件[13]。后期分解软件要求时，技术要求和技术安全配置会同时自己显现出来。本文的一位使用者用软件模式和与之匹配的叠加创建微服务软件应用程序，并在任何“敏捷开发”故事（agile story）编写、故事指向和软件开发发生之前，粗略地处于近似控制状态。

6.1 叠加介绍

安全控制措施叠加可通过执行相关业务和安全策略把风险降至一个可接受水平。控制挑选是在业务需要、投放市场时间和风险容忍度之间取得平衡的结果。安全叠加包装了一种软件模式，尽管它可能会带来更大的安全控制覆盖范围，但是，适合于一种模式的，只会有一个安全叠加。在微服务架构内的不同位置和层级发挥作用的控制措施，会产生形成软件深度防御的累加效应。

有些叠加会给我们带来在微服务架构中策略执行的机会。然而，并非每种模式及其相关叠加都会帮助我们执行策略（PEP）。策略的落实决定了是允许还是拒绝具体计算或交易操作。我们在把软件模式组装到拟议的应用程序中，然后对控制措施叠加编目时，需要评价叠加内或跨叠加的其他控制措施是否能够在叠加失效时起到补偿作用。其他软件模式叠加中的控制措施可能是支持主控制措施的，表现为一种双保险，但实际上或许并无必要。我们可以把下文将要介绍的叠加看作是软件模式构件的组成部分。我们用这些模式组装高层软件架构，然后对整体

[13] The Open Group. The Open Group TOGAF Standard version 9.2. The Open Group: Document Number: C182 Found at https://pubs.opengroup.org/architecture/togaf9-doc/arch/index.html. Access on 22 FEB 2021.

控制措施进行编目，以了解控制措施的覆盖范围或控制措施缺失的区域。安全控制措施叠加把一个由特定控制措施组成的离散集施用于相应软件模式。用一个关联表来表示就是，表的左列是软件模式，而表的右列模式的控制措施集。

6.1.1 服务叠加

服务通常执行某一特定功能并通过API把功能提供给服务的消费者；而API可通过REST/JSON、gRPC/Protobuf和SOAP/XML协议和格式访问。旧有Web服务还在通过使用量级更重的SOAP/XML协议的API提供功能。API在微服务架构中扮演着非常关键的角色，可以构建复杂应用系统并简化应用集成。目前，大多数数据泄露都是由于API缺乏安全设计导致的。API是常被攻击者盯上的一个目标，因为API会暴露服务功能以及个人可识别信息（PII）等敏感数据。

服务叠加可以为保护服务实例、方法和协议等方面的服务提供安全控制措施指导。API服务应该有一个机制，通过它对每个用OpenID Connect、OAuth 2.0、SAML等协议访问API的请求进行身份验证和授权。一般来说，我们可以在API网关上验证最终用户的上下文，并通过mTLS上的服务账户借助服务层面的信任访问下游服务，不过这种做法可能过于宽松，但是这一问题可以通过保持用户层面安全上下文以及服务层面信任缓解。把对API的访问记录到日志中并实施监控，可以用作发现安全问题的一项检测性控制措施。速率限制控制措施可用来预防针对API服务的DDoS攻击。为了防止网络内部的威胁，服务到服务的通信可依靠HTTPS等安全通信协议以及内部服务之间的适当身份验证/授权或使用内部pod localhost IP通信进行。

服务叠加只侧重服务层面模式，如Facade、Strangler、断路器、适配器模式等。我们在这里讨论服务叠加，是为了确保基于REST、gRPC和SOAP的API服务可发挥预期效力。

REST用结合TLS的HTTP在服务端点上安全执行操作。REST以JSON标准格式消费API的负载。对于输入，应该从长度、范围、格式和类型的角度验证。请求和响应，应该进行内容类型验证。输入请求应该限制为固定大小，当请求的负载超过这一大小时，应拒绝请求并显示413错误。应该启用日志记录和监控，以掌握输入验证和令牌验证的所有失败情况。敏感数据应该通过请求正文或标头传输，而不可直接在URL中传输。应避免在CORS标头中使用常规原点设置。

gRPC是一个高性能轻量级RPC框架。gRPC借助HTTP/2在服务之间执行操作。gRPC把协议缓冲区用作消息交换格式。由于gRPC中的消息采用二进制格式，应该进行彻底的负载检查和内容

验证，分离出恶意负载和避免敏感数据意外泄露。gRPC支持多种语言，因此选择更好的内存管理语言对于避免内存管理漏洞至为关键。开发人员应该避免在代码中使用默认的不安全通信选项，如“InsecureChannelCredentials”等。

SOAP是一种基于XML的消息传递协议，用于在服务之间交换信息。SOAP的内置标准（例如“Web服务安全”）用XML身份验证、XML签名和SAML令牌保护消息传递。对SOAP/XML的负载应该对照相关XML方案定义进行验证。SOAP应该为每个输入参数定义最大字符长度限制，此外还应该为固定参数模式定义严格的强验证模式。

模式	安全功能	控制措施系列	主要控制措施
Facade	在各种内部子系统执行身份验证机制。	IA	IA-3(1)，IA-5(2)，IA-9(1)
	对可通过用户输入访问数据来源的各种内部子系统实施对象层面授权。	AC	AC-3(1,7)，AC-6(1)
	当防火墙只允许经Facade和Facade IP发来的请求通过时，内部子系统受到防火墙保护。	SC	SC-7(11，15)
	确保内部子系统的可用性。	SC	SC-6
	避免从内部子系统暴露过多数据。	AC	AC-4(5，6，19，22)
	内部子系统失效管理，如无效审计、多个拥有者、旧有代码、过度简化等。	PL	PL-2(3)
	确保交互流中每项服务或功能的安全。	PL	PL-2(2)
	验证输入的长度、格式和类型，定义并只允许合适的请求大小。CORS、安全标头。	AC	AC-4(12，14，19)

	避免在没有根据允许列表进行适当属性过滤的情况下把客户端提供的数据与数据模型绑定。	AC	AC-4(15, 19, 20)
	在客户端一侧使用TLS 1.2+，确保服务器得到身份验证以及数据得到加密。	SC	SC-8(1, 3)
	使用gRPC内置身份验证机制，例如SSL/TLS、ALTS和基于身份验证的Google令牌	IA	IA-3(1), IA-5(2)
	强制使用gRPC凭证对象，用以创建整个信道（信道凭证）或单个API调用（调用凭证）。	IA	IA-9(1)
	gRPC支持多种语言并尝试使用内存安全语言，以避免出现高影响级内存管理漏洞，例如因缓冲区溢出而产生的远程代码执行。	AT	AT-2(1)
	在Facade接口进行SOAP/XML负载解析和方案验证，以避免常见漏洞（SQL注入和跨站点脚本）、DDT和XML方案漏洞。对照畸形XML实体验证XML炸弹攻击。	AC	AC-4(5, 6, 12, 15, 19, 20)
Strangler	对可通过用户输入访问数据来源的各种旧有子系统和新微服务实施对象层面授权。	AC	AC-3(1), AC-3(7), AC-6(1)
	在各种内部子系统执行身份验证。	IA	IA-3(1), IA-5(2), IA-9(1)
	在原有应用强制使用HTTPS端点；确保被传输数据的安全，允许客户端验证服务器身份并保持被传输数据的完整性。	SC	SC-8(1, 3)

	执行并强制使用API密钥，以避免受DDoS攻击影响。	SC	SC-5(1)
	旧有应用和新应用失效管理，例如无效审计、多个拥有者、旧有代码和过度简化。	PL	PL-2(3)
	在旧有后端服务进行SOAP/XML负载解析和方案验证。	SC	AC-4(5, 6, 12, 15, 19, 20)
断路器	应对因策略配置不当而导致的系统级联故障风险。	CM	CM-2(1), CM-3(2, 3), CM-6(1)
	下游服务监控失败的风险。	CA	CA-7(3)
适配器	验证输入的长度、格式和类型，定义并只允许合适的请求大小。	AC	AC-4(12, 14, 19)
	在各项后端服务执行身份验证机制。	IA	IA-3(1), IA-5(2), IA-9(1)
	避免在没有根据允许列表进行属性过滤的情况下把客户端提供的数据与数据模型绑定。	AC	AC-4 (15, 19, 20)
	确保内部子系统的可用性。	SC	SC-6
	在旧有后端服务进行SOAP/XML负载解析和方案验证。	SC	AC-4(5, 6, 12, 15, 19, 20)

服务叠加还提供安全控制措施指导，将安全控制措施应用于软件开发过程，为各种模式构建、测试和部署安全的API服务。

我们应该在软件开发进程的构建阶段定义编码标准并将其投入实际应用（尽可能自动执行）。

- 选择正确的引导框架。
- 依照指定的测量指标确保代码质量。
- 保持全单元测试覆盖。
- 依照指定的测量指标进行缺陷跟踪。
- 选择多次部署一个代码库而不选用多个代码库。
- 利用服务存储库（如Swagger定义）。

持续改进和持续集成（CI/CD）管道可确保低要求环境下的运行时间与实际生产中的相同。

- 安全测试基线：漏洞扫描、跟踪和修复。
- 在集成开发环境（IDE）中（以DEV[开发]部门执行的左移静态测试作为部门工作合并的前提）。
- 跨存储库（针对软件漏洞的静态企业层面测试）。
- 在质量保证（QA）环境中（功能性微服务应用及其容器的动态企业层面测试）。
- 在生产中：对照基线评价容器安全态势运行时。

模式	安全功能	控制措施系列	主要控制措施
所有模式	按指定的测量指标执行和跟踪编码标准。	SA	SA-15(1, 2, 4, 7)
	确保所有DEV和QA环境都接近生产环境。	SA	SA-3
	确保对微服务的所有安全评价活动（构建和运行时间）全都自动进行。	SA	SA-11(1, 2, 5, 8)
	确保在软件开发周期内进行漏洞扫描。	RA	RA-5(3, 9, 10)

6.1.2 IAM 叠加

IAM（身份和访问管理）在第一次验证后保留用户和服务器的身份，以避免由于没有正确处理会话持续时间而导致衍生攻击（如点击劫持和跨站点伪造）、识别用户以及允许/注册活动和阻止会话一段时间——若对方真是攻击者，则永久禁用。

在用于建立IAM功能的代码库中，我们有Auth、OAuth和OpenID可供使用。我们可以把部分用户信息添加到会话中，然后把这个密钥作为身份验证的一部分发送给Kerberos等票证处理系统或JWT等会话令牌，以此启用一次性口令和多因子身份验证功能。

模式	安全功能	控制措施系列	主要控制措施
卸载	SAML/Oauth令牌的升级/更新机制。这一控制措施允许通过用户友好界面集中管理用户访问，并且保存整个会话的用户信息。	SC	SC-23(3)
	与相邻安全日志互连的容器集合体（container fleet）。这一控制措施像双因子身份验证一样，可抵御跨站点伪造请求等攻击。	AU	AU-15
路由（路由选择）	IAM策略像一些路由器和交换机网络供应商所做的那样定义了路由选择。负载均衡路由可在OSI模型第3层抵御DDOS或DOS攻击，但可以把IAM用到第7层。	AC	AC-4
	API的预验证引用尝试用CORS源和内容安全策略等标头消费其他API，可避免通过无效路由发送数据以节约资源（API和用户角色是不允许尝试消费	AC	AC-4(30, 31)

	API的）。		
聚合	通过编排对API进行分组并通过IAM访问这些API组。这种身份验证系统允许在Docker容器部署完毕后将管理会话的资源负载委托出去，可避免Docker容器溢出并改善API的高可用。	AC	AC-29
缓存	IAM凭证像缓存那样在Docker凭证保密服务中保留一段时间，避免了向IAM持续发送验证用户的请求。	IA	IA-4(9)
AuthN	用自签名客户端数字证书请求调用离线服务和使用RPA的触发器，可帮助验证登录和验证码，而无需用户进行消费API的交互，就像三因子身份验证中的“盐”块（用户的唯一标识符块）可以避免用户和机器伪造那样。	AC IA SC	AC-4(17), AC-4(21), AC-4(3), IA-5(2), SC-23(5)
AuthZ	在访问JWT或Kerberos等令牌时添加IAM用户信息，可帮助验证本类功能的用户并阻止跨站伪造等攻击。	SC	SC-23(3)
	为各种子系统执行XACML、RBAC、ABAC。允许按子组细分授权控制，而不是用一个正式的联合系统帮助实现安全冗余。	AC IA	AC-10, IA-5(8)
Facade	在IAM失败、过载并在前端和HTTP/HTTPS/TLS中成功发送消息时发出消息，告知用户会话的实际状态，以避免混淆、多次失败尝试或请求不完整。	IA	IA-4(4), IA-4(6), IA-3(1)

断路器	将DDOS保护代码库与IAM集成并在它们之间建立消息链，以跟踪可能由攻击引发的任何事件。	SA	SA-4(2)，SA-8(8)

6.1.3 网络叠加

网络功能通常在执行卸载、聚合、路由选择、代理和缓存模式的网关中实现。网络叠加确保信息和信息系统得到这些模式的保护。从以操作为中心的角度说，网络叠加可以通过源IP地址（谁[Who]）、目的地URL（什么[What]）以及通信协议和加密（如何[How]）控制访问。从以数据为中心的角度说，网络叠加可以控制供访问的数据是加密还是不加密（如何）。而从以人为中心的角度说，网络叠加可以通过阻止公司内部网以外的源IP地址或被分配给商业禁运国的源IP地址来控制访问。

微服务聚合到一起卸载许多基本服务（如以单个服务形式出现并依靠服务路由选择[通常使用ISTIO]到达单个服务的IAM服务），这种情况已变得越来越普遍。聚合通过在众多服务间共享URL端点实现。而路由选择通过目标服务的不同URI路径实现。结果往往是应用层形成一个网格或网络。网格的设计要求周密考虑安全因素以避免出现不良后果。例如，一个典型的错误是把内部服务请求路由到了外部端点。网格设计的一般原则是：

- 用于不同安全保密级别数据的端点必须相互隔离。
- 用于不同安全保密级别用户的端点必须相互隔离。
- 服务于不同信任区域（例如内部与外部）的服务必须部署足够的边界控制。

模式	安全功能	控制措施系列	主要控制措施
卸载	这种模式为所有微服务采用了标准第4层通信保护（TLS）。它通过密码功能提供保密性和完整性；拒绝缺乏充分密码保护的入站通信流。如果没有充分的密码保护，数据的保密性和完整性会面临风险。	SC，AC	SC-7(11)，SC-8(1)，SC-12(2，3)，AC-17(2)
	这一模式通常还向通信的一方或两方提供身份验证。这种较低通信层的身份验证不能替代应用层的身份验证。	IA	IA-3(1)，IA-5(2)
代理	这一模式用来只暴露被选定的区域，而不把一个信任区域里的所有微服务全部暴露给另一个信任区域。它架起了跨越信任边界的通信桥梁。	SC	SC-7(5，11，17)
	限制特定服务流的通信。如果没有这些控制，数据会过度暴露。	AC	AC-4(2，6)
聚合	这一模式将多个不同微服务聚合成一个整体。它有过度暴露数据的风险。对一项服务合适的保护对另一项服务可能不合适。因此，用于不同安全保密级别数据的服务不得聚合。	AC	AC-4(2，6，8，11，12，14)
	为了保持功能层面的授权，用于不同安全保密级别用户的服务不得聚合。	AC	AC-3(3，4，13，14，15)
	必须制定并落实细粒度授权策略。聚合服务不得打破对象层面的授权。	AC	AC-6(1，2，3，4，9，10)
	对服务必须定期评价数据暴露的影响。数据安全类别或安全保密级别的变化必须体现到服务聚合的变化中。	RA	RA-2(1)

	对服务必须定期评价用户安全保密级别。用户安全保密级别的变化必须体现到服务聚合的变化中。	RA	RA-5(5)
路由	这一模式在路由选择之前给不同的微服务分配一条信道。它有把一项服务暴露给另一项服务并造成数据过度暴露的风险。针对不同安全保密级别数据的服务必须用不同的信道路由。	AC	AC-4(2, 6, 8, 11, 12, 14)
	为了保持功能层面的授权，针对不同安全保密级别用户的服务必须用不同的信道路由。	AC	AC-3(3, 4, 13, 14, 15)
	必须制定并落实细粒度授权策略。为多项服务分配信道不得打破对象层面的授权。	AC	AC-6(1, 2, 3, 4, 9, 10)
	对服务必须定期评价数据暴露的影响。数据安全类别或安全保密级别的变化必须体现到服务聚合的变化中。	RA	RA-2(1)
	对服务必须定期评价用户安全保密级别。用户安全保密级别的变化必须体现到服务聚合的变化中。	RA	RA-5(5)
缓存	这一模式为不同微服务缓存数据。它有暴露不同敏感度数据并造成数据过度暴露的风险。不同安全保密级别的数据不得由同一个缓存模式提供服务。	AC	AC-6(1, 2, 3, 4, 9, 10, 12)
	这一模式降低了微服务对数据可用性的高要求，并保护它们免受数据交付资源的限制。	AC	AC-21(2)

除了各个模式各有特定的安全功能外，网络模式还有利于实现基于网络的测试和监控，例如分布式 API 跟踪等。基于网络的测试和监控可以帮助：

- 发现并映射资源之间的API关系。
- 根据警报生成API拓扑图，从而简化故障排除工作。
- 根据拓扑关系使可视化和导航资源变得更成熟。
- 找到故障点和存在服务中断漏洞的区域。

6.1.4 监控叠加

监控控制措施叠加旨在提供事件日志、测量指标和警报，以确保微服务、容器以及底层平台或运行时环境的安全和可用。

控制措施叠加侧重于应用程序和微服务，以及容器和底层平台。在这种意图下会出现以下场景：

- 当微服务生成安全事件时，需要捕捉事件并记录到日志中。作为可选项，可能还需要发出警报。
- 实例化容器时，可能需要捕捉和记录该事件。
- 当微服务性能的测量指标超过预定义的阈值时，底层容器或平台可能需要纵向/横向扩展以便于微服务增加负载。
- 当解决方案生态系统在低效状态下运行时，需要通知运维团队。
- 业务利益相关人可能希望在出现纵向/横向扩展的情况时，特别是当成本成为一个必须考虑的因素时得到通知。
- 事件响应团队可能需要访问详细日志数据以进行调查。

以上内容虽然没有详尽列出所有场景，但是我们希望它能为这个特定叠加所要求采用的控制提供某种上下文背景。主要控制措施功能可以部署在控制或企业层面。但是，若想启用控制，往往还需要在微服务层面有特定配置。例如，如果容器平台或运行时环境尽管设有详细日志记录设施，但微服务没有被配置成在日志中记录事件（即日志层面设置不当），则控制措施将失败。微服务的开发人员理应把注意力集中在业务功能上，而把事件的日志记录简化成写进一个输出流，然后由系统捕捉、整理或聚合这个事件流，并转发或路由到平台的日志记录设施、系统日志接收器、仪表板和其他分析和监控工具。

我们在探索审计、监控和警报控制时，要基于以下指导原则为自己挑选一套控制：

- 安全和关键操作事件必须记录到日志中。
- 日志必须值得信赖（例如，不可变、不可否认），可支持取证和根因分析。
- 日志记录机制必须能够从故障中恢复。
- 应用程序开发人员不必关注捕捉或路由日志的具体方式。

我们在设计和执行监控控制CS时，可以从NIST SP 800-53中得到有关这一控制措施的进一步指导。下表可帮助为应用程序或微服务监控叠加识别拟议的控制：

模式	安全功能	控制措施系列	主要控制措施
所有模式	日志记录支持取证、调查、性能、运行弹性和根因分析。注重点是信息的质量而非遥测数据的数量。	AU	AU-2, AU-3, AU-12
	监控应用程序和容器配置的变更对于系统运行中断的取证和根因分析至关重要。当关键组件配置元素的变更影响了解决方案的安全态势时，会生成警报。	CM	CM-2, CM-3(5)

下表可帮助识别容器、平台或企业层面控叠加要求使用的控制措施：

模式	安全功能	控制措施系列	主要控制措施
所有模式	日志记录支持取证、调查、性能、运行弹性和根因分析。注重点是信息的质量而非遥测数据的数量。	AU	AU-2, AU-3, AU-9, AU-12, AU-16

	监控代理配置的变更对于系统运行中断的取证和根因分析至关重要。当关键代理配置元素的变更影响了解决方案的安全态势时，会生成警报。	CM	CM-2，CM-3，CM-12
	持续检查、分析数据并将其关联为可操作信息。响应关键事件。通过警报、报告和仪表板向利益相关人展示调查结果、行动和趋势。	CA	CA-7
	确定发生关键安全事件后，机构必须努力了解所发生的具体情况、事件发生的原因以及应该如何防止同类关键事件再次发生。	IR	IR-4，IR-5

这些单个控制措施并不是孤立工作的，而是汇合在一起共同体现架构师的设想或理念，确保建立起适当的监控。

例如，保留事件的不可变日志要求对监视器有一个基线预期（CM-2“基线配置”），确保基线功能始终可行且不被破坏（CM-3(5)“配置变更控制”），以及遥测和日志数据在充分保护下不会被人损坏（AU-9“审计信息保护”）。

监控控制措施叠加适用于第5章各节描述的所有模式。

最后，应用程序的开发和运维团队必须认识到，监控控制措施本身可能会要求配备监视器或警报机制。产品负责人和管理者应该把握机会研究和分析相关信息，例如关键绩效指标（KPI）、安全测量指标和这些指标的发展趋势等，以确保业务运营需要可以得到满足。此外，我们还要充分分离不同贡献者的职责，并对影响功能和安全态势的问题排出逐个解决的优先顺序。

6.1.5 密码叠加

加密在微服务中用于保护数据。加密的类型由数据、环境和存在的任何威胁决定。我们可以通过详细的威胁建模识别攻击场景和潜在攻击向量、数据的敏感性、适用法规合规或法律要

求以及潜在数据泄露对公司声誉的影响。就微服务而言我们知道，三种状态下的数据需要得到保护，这三种状态贯穿了微服务的各种模式，如卸载、路由选择、聚合、缓存、AuthN、AuthZ、Facade和适配器模式。

- 传输中（网络）；
- 使用中/处理中（CPU、内存、缓存）；
- 静止（存储）。

1）传输中

最终用户通过面向外部的端点验证身份。外部服务到微服务的访问、用户到微服务的访问以及各项微服务本身之间的访问等各种通信路径也需要得到保护。由于微服务之间存在着如此之多的通信，因此通过传输层安全（mTLS）上的相互身份验证来对传输中的数据实施保护至关重要。HTTPS（TLS）在设计上通过具有两项功能的X.509证书确保数据的隐私性和完整性：

- 授予通过公钥基础设施（PKI）使用加密通信的许可权；
- 验证证书持有者的身份。

微服务会与授权服务器和其他微服务通信。这些通信过程可能使用了秘密。而这些秘密可能是API密钥、客户端秘密或用于基本身份验证的凭证。这些秘密应该受到加密保护，不应检入源控制系统。

2）使用中/处理中

与静止数据和移动数据相比，使用中/缓存中数据的保护更为复杂，因为“使用中的数据”必须保持需要时可供访问的状态。缓存中数据的保护涉及——缓存中数据的保密性以及在缓存与使用缓存的应用程序之间移动的数据的保密性。缓存中的数据可以分区或隔离存储，每个分区都应该通过强身份验证和严格（及细粒度）访问控制施以保护。每个分区都可以通过只供受权实体使用的加密密钥进一步加密。在缓存与应用程序之间移动的缓存数据则可通过安全协议（如mTLS、TCP、HTTP）予以保护。机构还应具备记录、跟踪、报告、监控和发出警报的能力，以检测和预防潜在威胁。对于缓存，还可以通过以下方式进一步保护：

- 规定可用最大内存量限制；
- 给缓存中的密钥配置一个有效期限，到期后自动从缓存中删除；
- 缓存失效（确保删除陈旧数据，根据策略自动移除密钥值）。

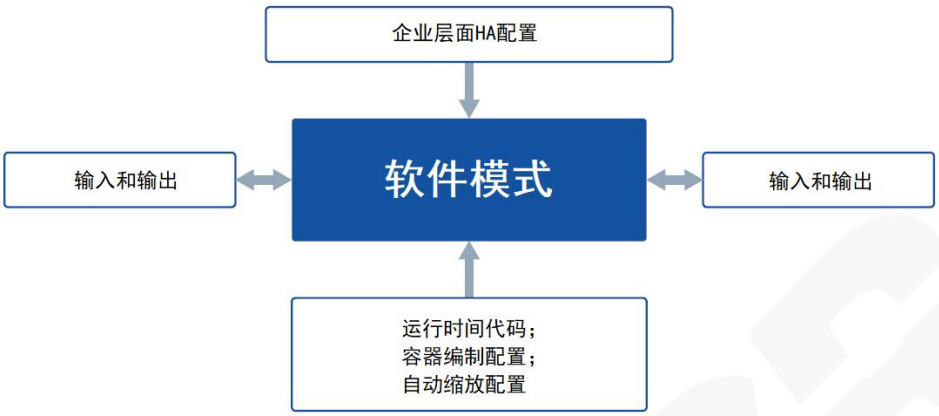
3）静止

对访问的身份验证完成后，在允许通过访问令牌访问由服务提供的资源/数据之前，建立经过身份验证的用户安全上下文至关重要。访问令牌必须有数字签名，以证明签发者的真实性和令牌的完整性。令牌还可以加密，以确保数据（例如敏感的财务数据）的保密性。令牌化用一种叫作令牌的非敏感等价物替代敏感数据元素。

模式	安全功能	控制措施系列	主要控制措施
卸载	这一模式通过密码功能提供传输层保护（mTLS）、保密性和完整性；令牌验证、加密、SSL证书管理等安全功能需要得到适当管理。证书需要适当配置和部署，并需要定期更新直至到期。证书在部署之前需要接受测试。	AC IA SC CM	AC-17, IA-3 (1) SC-23(3) CM-3(6)
路由（路由选择）	路由选择模式在网络的上面提供了一个层，这意味着你的应用程序代码可以连接到服务而不是单个容器。 这一模式使用了密码、加密、粗粒度/细粒度访问控制、令牌授权有效负载等安全控制措施。	AC SC	AC-10 SC-23(3)
聚合	聚合器模式通过服务接收请求，随后向多项服务发出请求，把得到的结果组合到一起后对最初的请求做出响应。聚合器微服务可以从多个微服务中收集数据并将结果返回给消费者。这一模式为实现安全通信使用了密码、加密、粗粒度/细粒度访问控制、令牌授权有效负载等安全控制措施。	AC SC	AC-10 SC-23(3)

缓存	缓存模式把数据保存在设备内存中以供反复访问，以此增加可用性、提高性能、减轻后端负载或减少停机时间。缓存模式使用了密码、加密、粗粒度/细粒度访问控制、运行时的密钥对值完整性验证（JSON Web令牌）等安全控制措施。	AC SC	AC-10 SC-23(3)
AuthN	AuthN模式提供验证实体身份的能力。AuthN模式借助密钥标志（数字签名）、JSON Web令牌、加密策略保护这一模式的数据。	IA AU SC	IA-9 AU-10 SC-23(3)
AuthZ	AuthZ模式可用来验证实体是否得到访问特定信息的授权，或者是否得到允许可以执行某些操作。AuthZ模式借助密钥标志（数字签名）、JSON访问令牌、加密策略保护这一模式的数据。	SC	SC-23(3)
Facade	Facade微服务模式可隐藏向外部服务发出请求或提供基于某些业务逻辑的结果集时所必须做到的复杂性。这一模式使用了密码、加密、粗粒度/细粒度访问控制、令牌授权有效负载等安全控制措施。	AC SC	AC-10 SC-23(3)
适配器（包装器/转化/转换）	适配器模式提供了一种方法，可用来识别如何以简单方式在类别和对象之间建立关系。当不兼容的模块需要与现有模块集成而不需作任何源代码修改时，可使用这一模式。该模式使用了密码、加密、粗粒度/细粒度访问控制、令牌授权有效负载等安全控制措施。	AC SC	AC-10 SC-23(3)

6.1.6 微服务的弹性和可用性叠加



1）微服务的弹性和可用性

本节微服务架构模式选择以论证治理、风险和合规（GRC）的重要性为主线贯穿全文，而不是单把GRC拿出来详细论述它的各个主题。GRC元素在有关安全叠加的各个小节里到处可见。微服务的弹性和可用性紧跟向软件左移的发展趋势并脱离GRC的具体计划管理，是由从企业层面继承来的架构属性（例如备份、恢复、高可用性和水平扩展，外加可进一步提升负载下性能的软件弹性设计策略）所产生的效应的综合体现。当弹性体现了由三条具体架构原则构成的元能力时，往往会整体视为一种能力。

- 吸收能力——可以吸收超出应用程序正常活动范畴的冲击或压力。
- 自适应能力——应用程序可以在负载下以预计的方式做出改变，通过降低服务水平保证继续运行。
- 恢复能力——当应用程序负载的压力和适应性超出监控显示的正常状态时，应用程序可以自我恢复到以前的状态。

可用性是指应用程序在多用户多模式状态下正常运行并能在既定时间内执行其约定功能的可计量持续时间。应用程序可能会在“启动”或“打开”状态下不可用，如按计划进行维护保养期间，这种情况叫计划内停机时间。可用性[14]方程式把计划内停机时间也作为一个因素考虑在内，将可用性用时间百分比表示（ITIL方程[15]）。在微服务环境中，可用性涵盖为提供某

[14] Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html.

[15] Rance, S. (2017, June 22). How to Define, Measure, and Report IT Service Availability. ITSM Tools. https://itsm.tools/how-to-define-measure-and-report-service-availability/.

项特定业务功能而安排的一系列微服务（微服务链）。下面是一个简单的例子：

（服务1×(.99可用)×服务2(.99可用)×服务3(.99可用)＝.97）

单独评价时，所有3项微服务的可用性都为99%。但是随着微服务交易链的延长，端到端交易的可用性会有所降低。我们应该用平台弹性策略应对这种链式效应，要在设计微服务应用程序时减少交易链中接口跳转的数量。我们还可以利用企业层面的平台功能抵消跨多个微服务应用程序交易的可计算服务可用性的预期下降。追求高可用性平台功能，如冗余（N+1或更高），自然要按业务要求防止出现级联微服务故障。我们要在企业层面和软件层面前端实施负载平衡控制，并继续依靠传统的备份和恢复能力——但要谨慎小心。容器平台遵循传统策略，但是从为微服务提供备份和恢复的角度看，与其说这是托管微服务的平台，倒不如说它们与大容量数据备份和恢复共同点更多。

2）微服务软件的弹性和可用性

随着微服务架构风格逐渐成熟，Heroku的开发人员于2012年创建的“十二因素应用程序”[16]框架一次又一次证明了自己。微服务架构风格的核心原则是每项应用服务只承担一个责任。为每个容器操作系统只构建和部署一种业务能力，使那些会在类似情况下出现变化的成品被组装到一起后，可以与因为不相干目的而发生变化的其他服务分离开来。一个应用组件与其他应用组件之间保持松散耦合关系乃至完全独立可以确保当需要比较复杂的业务操作时，通信只通过API发生。我们不妨以要求具有订单更新能力的一个业务工作流程为例。一项微服务被托管在一个docker容器中，经编码可执行记录更新操作，体现了一种业务能力，即更新订单，而这种业务能力正是由这项被调用来以新数据更新订单记录的微服务展示出来的。在这种原子级别服务情况下，微服务应用程序显然代表了一组微服务，它们之间相互通信以执行更复杂的业务任务。在微服务架构风格中，将业务流程隔离为单项微服务，是横向扩展处理以适应负载不断增加情况的主要手段——这也是通过可继承的容器和集群平台可用性实现技术弹性的关键驱动因素。为大规模部署而设的容器编排功能允许在镜像存储库中维护、保护和管制主容器镜像和文件定义。这种编排功能管理着容器负载，可根据应用程序的性能而非容器的性能提供调整内存和CPU大小的能力。最后，编排功能还可以根据应用程序负载阈值提供自动扩展（水平扩展）新容器实例的能力。

---

[16] Wiggins, A. (2017). The Twelve-Factor App. https://12factor.net/.

首席开发工程师和软件架构师可以在软件的开发设计过程中专门设计软件的弹性能力，以防出现平台的能力无法独自满足业务要求的情况。具体策略包括主动或机会性 TCP 超时和连接重置、回退编码以在较低功能状态下提供服务、用断路器或包装器软件架构模式来约束和控制流量激增，以及专门用于线程隔离和/或多线程的代码。这里的重点是需要最终达成一致的无状态交易。提高应用程序和数据可用性的其他平台策略还包括，在同一微服务的多个实例之间实施软件负载平衡控制，把本地和远程缓存软件模式用于数据冗余，执行数据排队，以及必要时将 Facade 或缓存软件模式重构成一个隔板，把服务隔离到数据隔间中，以打破关键微服务之间发生级联故障的潜在可能性。我们要根据业务流程的弹性和可用性要求采用多种策略建立独立于实际微服务软件编码的环境变量、配置和支持服务。企业层面和软件层面的能力可推动微服务的弹性和可用性不断提升。**微服务备份和恢复的策略和权衡取舍**

微服务应用程序无法从单个和独立的备份中以一致的状态恢复。[17]微服务应用程序的状态变化把备份置于风险之下。由不同微服务管理着的不同实体之间的连接需要保持有效和完好无损。微服务的备份和恢复与数据存储库的交易恢复是紧密耦合的。即便快照备份和交易日志无法恢复抓拍快照那一刻的当时状态，微服务数据货币也依旧向前迈进了未来。即便在最好的高可用性交易日志关系数据库系统下，15 分钟的延迟也还是常态。为了保证数据的一致性，我们要在备份过程中锁定数据记录并同步抓拍微服务状态快照。锁定应用程序和数据会限制读写服务的可用性，只允许进行读操作。系统将根据备份的持续时间（由性能最低的数据存储和企业层面网络设计决定）拒绝或推延客户端有关改变状态的请求，直到整个应用程序完成备份。

3）完全可用性和具有有限可用性的一致备份最终会不一致

我们在独立备份微服务每项服务的状态时，可以在把事件添加到备份交易日志中之前先备份一项服务，然后在把相应事件添加到备份交易日志之后，再备份其他服务。而在从备份恢复服务时，两项服务中只会有一项恢复自己的完整日志。因此，从备份恢复后，微服务架构的全局状态将变得不一致。备份和恢复后互联网超媒体中出现链接失效便属于这种情况。失效的链接是指已不再能循着它从A页到达B页，或者从目标A到达目标B的相对参引。当相关参引不存在时，就会产生URI失效的情况。在微服务背景下，微服务B与微服务A始终一致，直到它变得不可用。B从过时的备份中恢复后，它将不具有与打破交易链的A最新记录的事件相对应的状态。如果业务要求规定在备份操作的过程中不可给微服务日志添加任何事件，此项要求实际上降低

[17] Pardon, G., Pautasso, C., & Zimmerman, O. (2019). Consistent Disaster Recovery for Microservices: The BAC Theorem. IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bactheorem.pdf.

了微服务对需要执行某些状态转换/写操作的客户端的可用性。这样的权衡取舍通过协调微服务的服务备份确保快照抓拍于交互发生之前或双方把交互的影响记录到日志中之后，从而避免出现不一致。但是这种做法侵害了两项微服务的独立性，因为它给操作生命周期引入了紧耦合。正是出于这一原因，微服务应用程序的备份和恢复应该在pod（由许多相关容器组成的集合体）或集群（由许多相关pod组成）层面进行而不是针对单项微服务进行。我们最好把微服务应用程序部署成一个负载平衡的“A”端和一个镜像“B”端，其中一端保持可读写状态，另一端则保持只读状态，用于维护、打补丁或恢复（反之亦然）。支持性数据服务将需要利用缓存和排队减轻最终的不一致。RESTful服务总是处于数据不一致的状态，这便是在基于协议的数据传输上选择用REST法进行交易通信的一种权衡取舍。微服务的状态会放大数据的不一致。从备份恢复后，自主持久性（或多语言持久性）和一致性不可能同时存在。而当前实现的“最终一致性”也未必会有很好的效果。如果两个系统是独立的并且都受变化的影响，则可能会出现各项微服务的备份不以同一速度变化的情况。微服务备份必须同时进行才能保持一致性，这在实践中意味着所有服务共享一个数据存储，或者跨不同数据存储形成一致的备份。[18]

生产性能分析要求规定，检测出的所有损坏和孤立的微服务接口连接都应该记录到日志之中，以便运维人员在应用程序运行中断期间或发生灾害之后决定采用什么策略尝试恢复应用程序的一致性。常用的DevSecOps策略包括：

1）不采取行动

系统和用户接受系统的某些部分存在不一致。

2）依靠缓存中的数据

系统依靠数据的多层缓存。如果一层缓存恢复数据失败，系统可求助次级缓存，或者调用记录源。否则的话，系统只能靠向最终用户发出只读警告的方式处理事件。

3）人工干预——孤立的进程状态

孤立进程可能会导致验证错误，进而拒绝入站交易事件和消息。一项性能分析要求规定要定期运行验证服务和/或数据库一致性检查器。

---

[18] Github. (2019.) Guide on Microservices: Backups and Consistency | Consistent Disaster Recovery for Microservices - the BAC Theorem. dhana-git/Guide on Microservices: Backups and Consistency.md. Retrieved August 11, 2021, from https://gist.github.com/dhanagit/3dda5326b3bd15a93d3389a6c30d3000.

4）人工干预——错失的进程状态

对于容易从源头复制的事件，应该通过重放找到错失的事件。对于不容易从源头复制的事件，则可能需要通过人工干预以一条或多条用户命令的形式施用（或重新施用）错失的状态，或者接受事件丢失的事实。

5）服务连续性方面的考虑

我们或许应该考虑单独投资建设专用于备份和恢复的企业层面高带宽网络，以确保备份和恢复通信流不必与公司的日常工作通信流争夺带宽。高可用性（HA）数据层：有了HA备份网络，备份进程便可不间断进行，在理论上可瞬间完成恢复。不过，对每项微服务都使用高可用性数据层将是一个昂贵的解决方案，这样做会增加系统操作的复杂性，并会在许多方面给公司带来不便。因此，我们还要对处理应用程序恢复后所形成的最终不一致性需要花费多大成本进行比较。如前所述，备份应该在pod和集群层面进行。其次，我们应该考虑采用分布式快照（也叫“抓拍”）。快照是整个微服务架构的全局状态检测和逐点检查所不可缺少的。我们可以在各种微服务出于备份目的将自己的状态存储进快照的过程中对它们进行协调。采用快照数据存储策略时，每个本地数据库都包含其微服务的当前状态。但是这会限制独立部署、运行和发展微服务的灵活性。[19]我们应该在软件的设计阶段，就把恢复组件状态的功能直接编程进软件，从组件的已知初始状态开始，按每条消息在组件变得不可用之前先后到达组件/实体的顺序重放组件收到的每条消息，最终实现恢复组件的状态。我们应该采用缓存模式复制缓存中的数据，同时依靠从文件或存储库对环境和配置数据的重新读取，而不是依赖当前的运行时状态。

模式	安全功能	控制措施系列	主要控制措施
Facade、Strangler、断路器、缓存、包装器	自动验证和保留过去和当前基线配置、配置设定、环境设置，用于库存、保留、偏差检测和准确性，以便系统可以恢复到运维人员选择的状态。	CM	CM-2(2)，CM-2(3)，CM-6(1)，CM-8(3)，CM-8(6)

[19] Pardon, G., Pautasso, C., & Zimmerman, O. (2019). Consistent Disaster Recovery for Microservices: The BAC Theorem. IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bactheorem.pdf.

企业层面	力争实现可继承的高可用性（N+1）冗余，以确保手边有足够的容量，或在需要时可供调用；与运行中的资产分开单独存储和维护关键映像、文件定义、环境及配置信息。	CP	CP-2(2), CP-9(3), CP-9(6)
企业层面	为常规网络通信流活动以外的通信流专门提供用于备份和恢复的网络、子网、DNS和带宽。	SC	SC-37(1)

总结/结论

云计算、DevOps文化的出现，再加上各大供应商纷纷推出自己的软件构建模式和表现形式，使软件开发这个拥挤的领域充斥着许多强大的声音。供应商一方面要展示自家平台的精细差异，另一方面又要让自持语言和概念的客户也参与进来。软件模式的使用有着悠久而辉煌的历史，我们听到的或许未必是新鲜事物，但它们却是被在新介质上用新方法验证过的真实概念。

“左移”是传统上的一种软件开发概念，是指测试和工具要与软件工程师离得更近，以便功能可以在软件构建过程中更早得到测试。DevSecOps运动开启了对将安全控制措施叠加应用于传统软件设计模式的探索。有关控制的讨论往往会追溯工作计划或项目层面的评价工作。DevSecOps激励机构招募、培训优秀人才并开展防御性软件开发继续教育。把安全设计概念直接引入软件设计，往往有赖于具有前卫设计理念的首席软件工程师和安全架构师的存在，由他们一方面指导软件开发团队的工作，另一方面解决软件构建过程中遇到的难题。在信息安全领域，控制框架的成熟和进化已超出了软件的范畴，业界目前在尝试通过自动化和“即代码”的能力推动将控制策略融入软件部署进程，而不是软件设计进程。

微服务架构模式（MAP）工作组尝试在代码退出开发环境之前就把控制措施叠加施用于软件，以这种方式迈过“左移”的最后阶段。我们的远大目标是创建一个框架，基于这个框架在软件模式层面探索以原子方式看待控制的思路，从而使团队得以在讨论软件设计的同时把安全设计的问题也探讨清楚。完美的状态是，当MAP软件模式和叠加作为软件架构的构件加入进来时，微服务架构风格所特有的深度防御也随之展现出来。理想的结果是在与定义了软件要求的“敏捷开发”故事相同的冲刺计划中出现针对安全控制能力的“敏捷开发”故事。我们应该不再依赖更高级别的管道测试或软件开发人员集成开发环境（IDE）内的传统“左移”测试，而是让软件模式外加叠加从编写代码阶段提前跳到设计阶段，为新软件的开发提供所需要的要求和测试覆盖。

在设计进程的早期阶段就引入安全控制绝非易事。有些安全叠加存在于瀑布式组织流程中而且完全依赖强制建立的企业层面平台。依靠策略执行点能力才能实现适当控制功能的叠加要到稍后的应用程序测试阶段才能接受测试。团队可能要有一个“11天”全力冲刺，但是在这期间往往需要依照多周服务水平协议的规定如期交付安全性能。尽管存在许多明显制约，但是带安全叠加的软件模式毕竟为安全架构与软件架构的相互交融开辟了一条途径。微服务架构模式

外加叠加框架还不能说完善，谈不上没有缺陷和可以抵御持续的挑战。与任何新生事物一样——迭代是诞生于使用的必然产物。

附录 A：缩略语

ACL——Access Control List，访问控制列表

API——Application Programming Interface，应用编程接口

GRC——Governance, Risk and Compliance,治理、风险和合规

JSON——JavaScript Object Notation, JavaScript，对象表示法

JWT——JavaScript Web，令牌

LAN——Local Area Network，局域网

MAP——Microservices Architectural Pattern，微服务架构模式

SDLC——Software Defined Life Cycle，软件开发生命周期

附录 B：词汇表 Glossary

架构师 Architect

负责设置部署流程和管理 IT 服务的个人或团队。他们负责确保基础设施和操作环境的稳定运行，用来支持内部和外部客户的应用程序部署，其中包括网络基础设施、服务器和设备管理、计算机操作、IT 基础设施库（ITIL）管理和服务台服务。[20]。

架构 Architecture

系统在其环境中的基本概念或特性，体现在它的要素、关系以及设计和演化原则中[21]。

架构描述 Architecture Description

作为一种概念模型，架构描述涉及：

- 表述一种架构；
- 确定所涉及的系统；
- 确定一个或多个利益相关人；
- 确定一个或多个（有关所涉系统的）关注点；
- 包含一种或多种架构观点和一个或多个架构视角；
- 可能包含回应；
- 可能包含回应规则；
- 包含一个或多个架构原理[22]。

架构模式 Architectural Pattern

针对特定环境下软件架构常见问题的可重复使用的通用解决方案。架构模式与软件设计模式近似，但是涉及范围更广。架构模式涵盖软件工程中的各种问题，例如计算机硬件性能限制、

---

[20] CSA. Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems (Cloud Security Alliance: 2019) 42

[21] ISO/IEC/IEEE 42010. (2011). Systems and Software Engineering—Architecture: A Conceptual Model of Architecture Description. Retrieved August 11, 2021, from http://www.iso-architecture.org/ieee-1471/cm/

[22] ISO/IEC/IEEE 42010. (2011). Systems and Software Engineering—Architecture: A Conceptual Model of Architecture Description. Retrieved August 11, 2021, from http://www.iso-architecture.org/ieee-1471/cm/.

高可用性、业务风险最小化等。[23]

可用性 Availability

配置项或 IT 服务在被要求时执行约定功能的能力。可用性通常以百分比计算。这种计算通常基于约定的服务时间和故障停机时间。通过度量 IT 服务的业务输出来计算可用性是最佳做法。[24]

业务负责人 Business Owner

产品负责人的一种角色，负责在业务上实现可交付结果整体价值最大化；对外代表团队管理者的角色。在实践中，业务负责人要么是“主要”利益相关者，要么是团队的引领人，要么是产品负责人的主管。[25]

控制（动词）Control (v)

（监控）执行限制或施加直接影响；（减少）降低发生率或严重程度，尤其是降低到无害水平；（治理）通过检验或实验进行检查、测试或验证。

《韦氏大词典》：管理风险的手段，其中包括策略、规程、指南、实践规范或组织结构，可能属于行政、技术、管理或法律性质。范围说明：也用作防护措施或对策的同义词。另见内部控制。[26]

缓存 Cache

缓存是构建可伸缩微服务应用程序的一项要求。数据可以缓存在内存中或本地高速硬盘上。

控制目标 Control Objective

对通过在特定进程中执行控制规程取得预期结果或达到预期目的的陈述。[27]

控制框架 Control Framework

---

[23] Wikipedia contributors. (2020, March 20). Architectural pattern. Wikipedia. https://en.wikipedia. org/wiki/Architectural_pattern.

[24] Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html.

[25] Scrum Dictionary. Business Owner. ScrumDictionary.Com. Retrieved April 17, 2021, from https://scrumdictionary.com/term/business-owner/.

[26] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[27] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

可帮助业务流程负责人履行职责，防止企业出现财务损失或信息丢失的一套基本控制措施。[28]

开发人员 Developer

构建软件程序的商业或技术专业人士；计算机程序员（同义词），可以指计算机某一领域的专家，也可以指用一种或多种计算机编程语言为多种软件编写代码的全面型人才。[29]

企业经营者 Enterprise Operator

负责提出战略设计建议的个人或团队。他们通过将有关云、容器和微服务组件的知识应用于企业问题来打造可满足企业战略需要的最佳架构。此外，他们还与开发人员和经营人员合作，制定和维护解决方案路线图并监督路线图的落实情况，以确保解决方案得到有效和高效实施。

中介层 Inter-Mediation

API Facade 是隔在微服务与暴露于外部服务的 API 之间的一个层或网关。Facade 模式在应用程序接口、应用程序开发人员与复杂服务之间建立了一个缓冲区或一个层。你可以把多个 API 集成到不同的微服务中，而 Facade 模式把复杂性抽象出来，通过一个简单的界面展示。

微服务架构风格 Microservice Architectural Style

微服务架构通常是指在结构设计上可以使用被称作微服务的基本元素的应用程序，其中的每个元素都在自己的进程中运行并通过轻量级机制（通常是基于 HTTP 资源的 API）进行通信。这些服务围绕着业务能力构建，可通过全自动部署系统独立部署。这些服务只接受最低限度集中管理，可以用不同的编程语言编写并可采用不同的数据存储技术。[30]

运维人员 Operator

负责设置部署流程和管理 IT 服务的个人或团队。他们负责确保基础实施和操作环境的稳定运行，用来支持内部和外部客户部署应用程序，其中包括网络基础设施、服务器和设备管理、

---

[28] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[29] Wikipedia contributors. (2021b, August 7). Programmer. Wikipedia. https://en.wikipedia.org/wiki/Programmer.

[30] Cloud Security Alliance. Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems (Cloud Security Alliance: 2019) 42

计算机操作、IT 基础设施库（ITIL）管理和服务台服务。[31]

产品负责人 Product Owner

确定客户需要和更大业务目标（即将要实现的产品或功能）、阐明产品成功的样子并推动团队将产品愿景变成现实的人。[32] [33]

传播 Propagation

这里的传播是指安全上下文通过不同服务的传播。

软件 Software

告诉计算机如何完成工作的数据或计算机指令的集合。物理硬件就是在由软件构建而成的系统下执行工作的。[34]

软件架构 Software Architecture

系统结构，其中包括软件元素、这些元素的外部可见属性以及它们之间的关系。[35]

软件设计模式 Software Design Pattern

针对软件设计中常见问题的可重复使用的通用解决方案。它并不是可以直接转化成源代码或机器代码的成品设计，而只是说明如何解决问题的一种描述或模板，可供在许多不同情况下使用。[36]

解决方案 Solution

解决方案是架构、模式和设计工作对解决特定行业需要或业务问题的实际应用。解决方案旨在向持续客户和业务负责人提供价值。

---

[31] Cloud Security Alliance. Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems (Cloud Security Alliance: 2019) 42

[32] Mansour, S. (2020). Product Manager. Atlassian Software.https://www.atlassian.com/agile/product-management/product-manager.

[33] Agile Alliance. Product Owner. Accessed August 10, 2021 athttps://www.agilealliance.org/glossary/product-owner/.

[34] Cambridge Dictionary. (2021, August 11). Software. https://dictionary.cambridge.org/dictionary/english/software.

[35] Bass, L., Clements, P. C., & Kazman, R. (2012, September). Software Architecture in Practice, Third Edition. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264.

[36] Wikipedia contributors. (2021a, June 14). Software design pattern. Wikipedia. https://en.wikipedia.org/wiki/Software_design_pattern.

安全策略 Security Policy

表述企业信息安全理念和承诺的高阶文件。[37]

安全规程 Security Procedure

有关操作步骤和流程的正式文件，详细说明应该怎样实现安全策略和标准提出的安全目的和目标。[38]

安全标准 Security Standard

阐明需要采取的措施和必须关注的重点的实践规范、指示、指南、原则或基线；它们是对安全策略所述问题的解释说明。[39]

安全测试 Security Testing

确保修改后的或新的系统包含适当控制措施并且不会引入可能会危害其他系统或者误用系统或其信息的安全漏洞。[40]

安全架构 Security Architecture

是企业架构中专门涉及信息系统弹性问题并且为并执行满足安全要求的功能提供架构信息的部分。[41]

安全控制措施叠加 Security Controls Overlay

叠加是根据裁剪指南针对控制基线进行裁剪后得出的一个全套特定控制、控制强化和补充指南集。[42]

有关控制措施叠加的更多信息，请参阅美国国家标准与技术研究所（NIST）特别出版物 NIST SP 800-53 第 4 修订版第 3.3 节“创建叠加”及附录 I“叠加模板”。

---

[37] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[38] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[39] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[40] ISACA. Interactive Glossary & Term Translations. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

[41] Gantz, S. D., & Philpott, D. R. (2013). FISMA and the Risk Management Framework. ScienceDirect.

[42] NIST Information Technology Laboratory: Computer Security Resource Center (CRSC). (2009, June 12). FISMA Implementation Project. https://www.nist.gov/programs-projects/federal-informationsecurity-management-act-fisma-implementation-project.

Strangle

“Strangler”是一种参考模型，用于描述随着时间的推移不断给应用程序添加新的微服务，同时又随着时间的推移逐步停用整体式应用程序的某些性能，从而将整体式应用程序一点点现代化成微服务架构的过程，体现了动态模型开发过程中的剖析和过渡。

技术负债 Technical Debt

一种设计或构建方法，是短期内效果不错的权宜之计，但是创造了一个技术背景，在这种背景下，同一项工作以后完成的成本要高于现在就完成的成本（包括随着时间的推移而增加的成本）。[43]

转化 Transform

数据转化是指从源中提取数据，使其改变或转换成一种或另外一种格式，并将其加载到目标系统中。

转换 Translate

适配器微服务将（通常基于功能的）服务包装并转变成基于实体的 REST 接口。这允许将现有类别的接口用作另一类别的接口。

公用程序 Utility

（此处特指Sidecar。）Sidecar模式网格通过应用程序下的服务代理抽象底层基础设施。代理处理通信流、微服务之间的通信、连接、管理、负载平衡、可用性和遥测数据。Sidecar模式网格范式可跨多个云提供编排服务和不受底层云架构约束的架构独立性。

[43] McConnell, S. (2013). “Managing Technical Debt (slides),” in Workshop on Managing Technical Debt (part of ICSE 2013): IEEE, 2013.

附录 C：参考文献

以下是本文使用过的参考文献。

Agile Alliance. (2021).	Agile Alliance. *Product Owner.* Accessed August 10, 2021 at https://www.agilealliance.org/glossary/product-owner/
Bass, L., Clements, P.C, & Kazman, R. (2012, September).	Bass, L., Clements, P. C., & Kazman, R. (2012, September). *Software Architecture in Practice, Third Edition*. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264.
Boehm, B. & Basili, V. R. (2001).	Boehm, B., & Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34(1), 135–137.
Cambridge Dictionary. (2012, August 11).	Cambridge Dictionary. (2021, August 11). *Software*. https://dictionary.cambridge.org/dictionary/english/software
Cloud Security Alliance. (2019).	Cloud Security Alliance. *Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems* (Cloud Security Alliance: 2019) 42.
Cloud Security Alliance. (2019, July 8).	Cloud Security Alliance. (2019, July 8). *Six Pillars of DevSecOps*. https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/. 5-6
Cloud Security Alliance. (2019, August 1).	Cloud Security Alliance. (2019, August 1). *Information Security Management through Reflexive Security.* https://cloudsecurityalliance.org/artifacts/information-security-management-through-reflexive- security/. (13, 14, 16)

Cloud Security Alliance. (2020, February 24).	Cloud Security Alliance. (2020, February 24). *Best Practices in Implementing a Secure Microservices Architecture.* https://cloudsecurityalliance.org/artifacts/best-practices-in-implementing-a- secure-microservices-architecture/.
Docker. (2021).	Docker (2021). *Docker Hub: Set Up Automated Builds*. Docker. https://docs.docker.com/docker-hub/builds/
Enterprise Integration Patterns. (2021).	Enterprise Integration Patterns. *Enterprise Integration Patterns - Messaging Patterns Overview.* Retrieved August 11, 2021, from https://www.enterpriseintegrationpatterns.com/patterns/messaging/
Gantz, S. D., & Philpott, D. R. (2013).	Gantz, S. D., & Philpott, D. R. (2013). *FISMA and the Risk Management Framework.* ScienceDirect.
Github. (2019).	Github. (2019.) *Guide on Microservices: Backups and Consistency | Consistent Disaster Recovery for Microservices - the BAC Theorem.* dhana-git/Guide on Microservices: Backups and Consistency.md. Retrieved August 11, 2021, from https://gist.github.com/dhana-git/3dda5326b3bd15a93d3389a6c30d3000.
Hinkley, C. (2019).	Hinkley, C. (2019, November 6). *Dissecting the Risks and Benefits of Microservice Architecture.* TechZone360. https://www.techzone360.com/topics/techzone/articles/2019/11/06/443660-dissecting-risks-benefits-microservice-architecture.htm.
IBM. (2021).	IBM. *Patterns in Event-Driven Architectures – Introduction.* IBM Garage Event-Driven Reference Architecture. Retrieved August 11, 2021, from https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro
Information Technology Infrastructure Library. (2021).	Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html
ISO/IEC/IEEE 42010. (2011).	ISO/IEC/IEEE 42010. (2011). *Systems and Software Engineering — Architecture: A Concep ISACA. Interactive Glossary & Term Translations.* Retrieved August 11, 2021, from https://www.isaca.org/resources/ glossary *Model of Architecture Description*. Retrieved August 11, 2021, from

	http://www.iso-architecture.org/ieee-1471/cm/
Mansour, S. (2020).	Mansour, S. (2020). Product Manager. Atlassian Software. https://www.atlassian.com/agile/product-management/product-manager
McConnell, S. (2013).	McConnell, S. "Managing Technical Debt (slides)," in Workshop on Managing Technical Debt (part of ICSE 2013): IEEE, 2013.
Microsoft. (2021).	Microsoft. (2021). Architecture Best Practices: Caching. Microsoft https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching
NIST. (2015, January 22).	NIST. (2015, January 22). *NIST Special Publication 800-53, Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations.* https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final
NIST. (2020, September).	NIST. (2020, September). *NIST Special Publication 800-53, Revision 5: Security and Privacy Controls for Information Systems and Organizations.* https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf
NIST. (2021, August 4).	NIST. (2021, August 4). *Security and Private Control Overlay Overview.* Retrieved August 11, 2021, from https://csrc.nist.gov/projects/risk-management/sp800-53-controls/overlay-repository/overlay-overview.
NIST Information Technology Laboratory. (2009, June 12).	NIST Information Technology Laboratory: Computer Security Resource Center (CRSC). (2009, June 12). *FISMA Implementation Project.* https://www.nist.gov/programs-projects/federal-information-security-management-act-fisma-implementation-project
Pardon, G., Pautasso, C., & Zimmerman, O. (2019).	Pardon, G., Pautasso, C., & Zimmerman, O. (2019). *Consistent Disaster Recovery for Microservices: The BAC Theorem*. IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bac-theorem.pdf

Raible, M. (2020).	Raible, M (2020). *Security Patterns for Microservice Architectures, Encryption and Protection Secrets*. Okta. https://developer.okta.com/blog/2020/03/23/microservice-security-patterns#5-encrypt-and-protect-secrets
Rance, S. (2017, June 22).	Rance, S. (2017, June 22). *How to Define, Measure, and Report IT Service Availability*. ITSM Tools. https://itsm.tools/how-to-define-measure-and-report-service-availability/.
Scrum Dictionary. (2021).	Scrum Dictionary. Business Owner. ScrumDictionary.Com. Retrieved April 17, 2021, from https://scrumdictionary.com/term/business-owner/
The Open Group. (2021).	The Open Group. *The TOGAF Standard, Version 9.2 Overview, Phase A, B, C, D, E, F, and G*. Retrieved August 11, 2021, from https://www.opengroup.org/togaf.
Wiggins, A. (2017).	Wiggins, A. (2017). The Twelve-Factor App. https://12factor.net/
Wikipedia contributors. (2020).	Wikipedia contributors. (2020, March 20). *Architectural Pattern.* Wikipedia. https://en.wikipedia.org/wiki/Architectural_pattern
Wikipedia contributors. (2021, August 7).	Wikipedia contributors. (2021b, August 7). *Programmer*. Wikipedia. https://en.wikipedia.org/wiki/Programmer
Wikipedia contributors. (2021, June 14).	Wikipedia contributors. (2021a, June 14). *Software design pattern.* Wikipedia. https://en.wikipedia.org/wiki/Software_design_pattern

附录 D：作业练习指导

1.0 微服务架构模式模板

1.1 模式作业练习指导

请将你的模式与微服务应用模式参考架构保持一致。请依照以下格式和说明创建一个模式。

1.2 模式模板

软件模式名称<名称>[模板]	
版本	<数字0.0> 模式+叠加的最新迭代
模式目的	<目的> 这个模式要达到什么目的？它应该做什么？
层面位置	<平台层面/软件层面/混合分布式> 哪个或哪些层面？模式存在于平台层面、软件层面还是横跨两个层面？
结构描述	<模式能做什么？> 描述模式的作用——流程分析（SIPOC）——来源、输入、过程、输出、消费者。
行为描述	<模式有什么行为？> 描述模式在正常情况下的行为。

数据特性	<使用中的数据/传输中的数据/静止数据> 是生成新数据、传输现有数据还是将数据静态存储？
主要依赖性	<如果没有它，模式会崩溃> 模式是与名义功能还是别的什么东西紧密捆绑？
次要依赖性	<如果没有它，模式将以次优模式运行> 哪个平台或软件层面元素会因为模式以次优状态运行而不得不处于不良运行状态？
内部事件/消息传递需要	<是否有在进程间或模式间进行通信的需要？> 名义功能是否依赖来自其他模式的内部消息传递？
外部事件/消息传递链接	<日志记录的目的或类型——AppDev调试、SecOps、DBtransact>需要出于什么目的把活动记录到什么地方？
事件响应行为	<出现事件条件时会发生什么情况？> 对DOS或性能降级有什么响应行为？
共同的上游链接	<是否与其他模式有链接？> 是否存在被共同链接的上游模式？
共同的下游链接	<是否与其他模式有链接？> 是否存在被共同链接的下游模式？
Ops安全回接	<是否采用了IT安全控制？><基础设施即代码和配置> 叠加步骤1：采用了什么IT安全控制？

DevSecOps回接	<是否有应用程序安全控制？>＜是否进行过DEV管道安全测试和动态应用程序安全测试?> 叠加步骤2：采用了什么应用程序安全控制？
评估方法	<控制方法验证> 叠加步骤3：采用了什么可接受控制验证方法？
控制措施叠加的特性	<预防性/纠正性/检测性> 叠加步骤4：控制是预防性的、纠正性的还是检测性的？
复合状态（独有/通用）	<通用，与其他模式相同的控制措施叠加；本模式独有的控件叠加> 叠加是也可供其他模式使用的通用叠加还是本模式独有的叠加？

2.0 安全叠加模板

2.1 安全叠加作业练习指导

根据前文第 5 章的描述创建模式后，下一步应该依照以下作业练习指导创建安全叠加。

2.1.1 介绍

这个作业练习指导的主要目的是为本文 CSA MAP（微服务架构模式）所述架构模式开发安全控制措施叠加提供指导和标准化方法。尽管本文的内容可以对各种角色起到帮助作用，但它是针对安全架构的作用和架构师的理念量身定制的。作业练习指导的第二个目的是为安全架构创建一种经济实用、轻量级、不会在设计过程中牺牲控制挑选基本原则的非统计分解/重组方法。这种方法寻求避免在探讨控制措施的过程中出现抽象理念，有时甚至是深奥难懂的风险话题。这种方法包含 8 个步骤，用于充分展示软件架构模式的安全控制措施叠加。

我们期望控制措施叠加也与其他模式匹配。把以这种方法生成的所有其他模式叠加拿来

作一番比较，可以减少控制措施的数量。尽管这种方法容易产生主观性专业判断，但是这样做的局限性完全可以被它带来的好处抵消。这种方法标准化了为微服务应用程序中软件架构模式开发安全控制措施叠加的进程；它针对具体情况量身定制，可以充当本文的作业练习指导。

这种方法可转而用于风险偏好、控制工作集和行业不同的其他情况和环境。该方法可以作为正式风险评价和控制基线确定的先决条件。但是微服务架构模式（MAP）叠加法不能替代 NIST 特别出版物 SP 800-53 第 4 版《联邦信息系统和机构安全和隐私控制措施》第 3 章（挑选安全控制基线的）“流程”（第 28-44 页——英文版，下同）或 NIST SP 800-53 附录 D（从第 D-1 页开始）。

2.1.2 预备知识

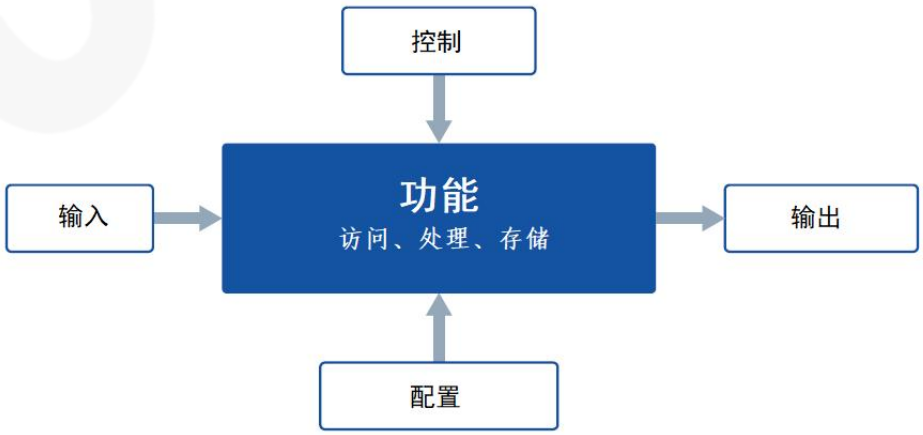
本作业练习指导使用了以下具体词汇和短语：

1）控制措施（特性）：控制措施按行为/功能分类（如防止拒绝服务的控制措施），或者按性质/类型分类（如要求用两个物理签名来防止伪造纸质文件的控制措施）。

a. 预防性、检测性和纠正性控制属于行为/功能控制措施。描述要与行动的时间一致。

b. 物理、管理、技术和监管控制属于性质/类型控制措施。描述要与行动的类型一致。

c. 控制措施的特性可进一步细分为“前瞻性”（预估问题）、“并举性”（在问题发生的当时纠正）和“反馈性”（问题发生后纠正）。



2）特性：某物在特定情况下以某种方式发挥作用时所体现的质量、特点、倾向。

3）有效性（架构的一种质量元属性）：对于控制措施的有效性，可以通过选择质量属性进行评估。实施控制措施是根据定义明确的机构风险偏好进行管理的一种手段。作为一名安全架构师，设计的首要目标是：

- 监控绩效并采取措施确保达到预期结果；
- 提高行动取得预期结果的可能性，正确时间安排下的安全控制架构属性为：

  a. 准确性（粗粒度或细粒度行动）；

  b. 按时间顺序展开（控制行动的时间安排）；

  c. 经济实用（控制的总成本）；

  d. 灵活性（威胁和/或具体情况覆盖）；

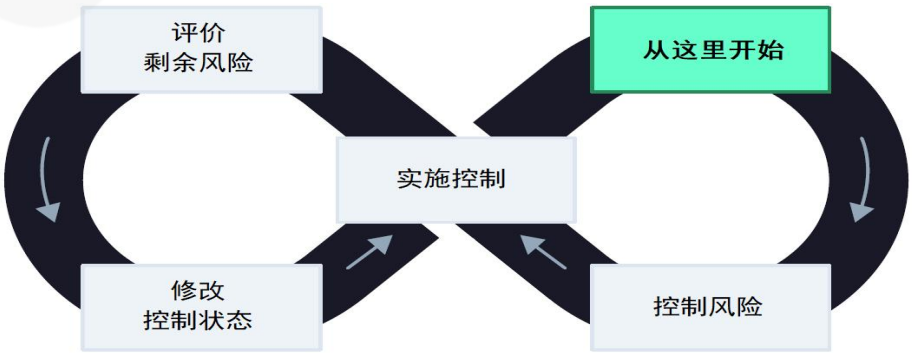
  e. 可理解性（可理解的行动和结果）；

  f. 合理性（绩效和安全之间的平衡）；

  g. 依照准则（控制的参数化调整）；

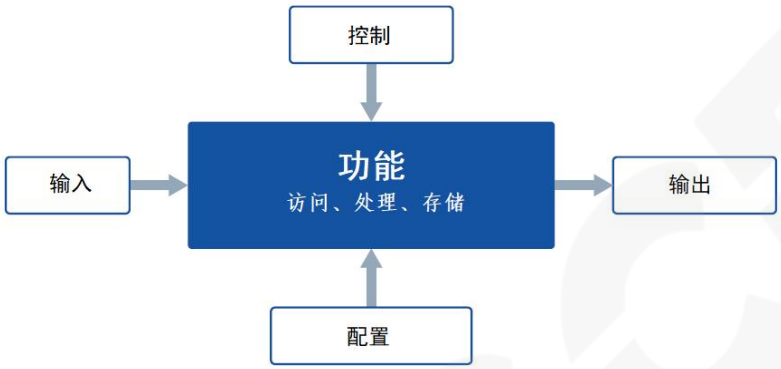
  h. 影响（风险回报权衡或控制应用）。

4）构建控制措施叠加：评价MAP参考架构模式对控制措施叠加的需要时，必须从数据风险入手，讨论保密性、完整性和可用性所受到的威胁。有关风险评价的探讨要从风险评价起步（下图的绿色部分）。而有关端到端风险的探讨，也要从风险评价开始并按以下流程进行：

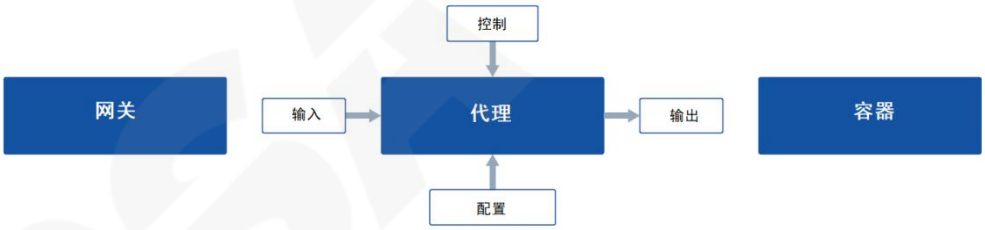


2.1.3 作业练习的步骤

1）返回参照通用控制状态图。把根据 NIST 特别出版物 SP 800-53 第 4 版《联邦信息系统和机构安全和隐私控制措施》附录 D（从 D-1 页开始）分配进“私有内部”类的，以及存在于“低”风险基线中的所有使用中的数据全部考虑在内。示例——



2）使用 MAP 参考架构，返回上一步，查看向模式的输入和从模式的输出，评价所需要的配置。返回通用控制状态图，重新标记 MAP 参考架构的输入、功能（模式名称）和输出。示例——



3）准备一个矩阵表，展示以操作为中心的一个视角。对照更新后的控制措施状态图，提出“谁（可以访问）”、“什么（可以访问）”、“何时（——在什么条件下——可以访问？）”、“怎样（进行访问）”以及“何处（可以进行访问）”的问题并给出答案。示例——谁（可以访问这一功能）？是受 ACL 或 mTLS 证书限制的上游接口。“什么（可以访问这一功能）”？是其他计算接口。“何时（——在什么条件下——可以访问这一功能）”？是在运行时，通过配置，以及在 QA 和测试环境中。“怎样（访问这一功能）”？是通过身份验证（AuthN）和授权（AuthZ）。“何处（可以访问这一功能）”？是在生产、QA 和测试环境中，通过端口和协议，以及通过中介层功能。出于本作业练习目的，应该把配置视为操作理念的组成部分。对控制的挑选应该针对某种安全配置进行。

谁（可以访问这一功能）？	受ACL或mTLS证书限制的上游接口
什么（可以访问这一功能）？	其他计算接口
何时（——在什么条件下——可以访问这一功能)?	在运行时，通过配置，以及在QA和测试环境中
怎样(访问这一功能)?	通过身份验证（AuthN）和授权（AuthZ）
何处（可以访问这一功能）？	在生产、QA和测试环境中，通过端口和协议，以及通过中介层功能

4）再准备一个矩阵表，用来展示以操作为中心的另一个视角。提出与步骤 3 相同的问题，但是要把“可以”改成“不可以”。询问中的默认拒绝是便是“因需可知”的基础，同时也是“深度防御”第一个组成部分。被访问、处理或存储的数据的敏感度与控制措施状态的深度成正比。示例——

谁（不可以访问这一功能）？	开发人员、运维人员、架构师、技术支持人员、经理、非员工
什么（不可以访问这一功能）？	交互式命令壳；特殊脚本命令
何时（——在什么条件下——不可以访问这一功能）？	在运行时，在容器托管的生产环境中，通过对等式脚本
怎样（阻止对这一功能的访问）？	输入过滤器、文件系统层面控制、端口和协议限制
（在）何处（拒绝对这一功能的访问）？	上游、下游、配置、端口和协议限制

5）再准备一个矩阵表，用来展示以操作为中心的第三个视角。提出与步骤 4 相同的问题，但是把人员访问用作查询控制状态的切入点。示例——

谁（不可以访问这一功能）？	拒绝开发和运维人员在生产和运行时进行访问；开发和运维人员被限制对QA的访问
什么（不可以访问这一功能）？	开发人员的命令壳；临时脚本命令
何时（——在什么条件下——不可以访问这一功能）？	工作时间以外，不包括履行行政管理职责的时间
怎样（阻止对这一功能的访问）？	AuthN、AuthZ、文件系统层面控制、端口和协议限制、ACL、xBAC（角色、上下文、属性策略）
（在）何处（阻止对这一功能的访问）？	上游、下游，通过配置限制，访问中介层、端口和协议的使用

6）返回进行风险评价评述。既然控制措施状态图有三个不同视角，那就应该把它们全部拿来形成风险评价。反复进行这种风险评价，直到模式控制状态满足需要为止。示例——

评价风险（属于什么风险类型？）	访问风险、传输泄露风险、拒绝服务风险
控制风险（主要有哪些类别控制？）	访问控制、传输控制、弹性控制
实施控制措施（在何处？）	上游平台，运行时配置，到功能本身
评价剩余风险（是否属于不可接受的数据风险？）	内部人员威胁、新的软件漏洞、故障
修改控制措施状态（改进？）	用新增加的控制抑制剩余风险

7）根据三个矩阵表和风险评价结果，从NIST特别出版物SP 800-53第4版《联邦信息系统和机构安全和隐私控制措施》第2.2节“安全控制措施结构”（第9页）挑选控制措施系列（而不要挑选单个控制措施）。熟悉该出版物的内容是执行步骤7的先决条件。安全架构要求从业者对许多行业标准控制集了如指掌。示例——

控制类别	NIST SP 800-53系列	控制措施特性	层面亲和关系的可继承性
访问控制	AC——访问控制	预防性	企业层面——可继承
传输控制	SC——系统通信和保护	检测性为主，预防性为辅	企业层面——可继承
配置控制	CM——配置管理	检测性为主，预防性为辅	企业层面——可继承
软件保障控制	SA——系统和服务采购	预防性	软件层面——唯一且可继承

8）在步骤8中，从NIST特别出版物SP 800-53第4版《联邦信息系统和机构安全和隐私控制措施》[44]中按所列控制措施系列挑选单个控制措施，分解步骤7的挑选结果。重组步骤7按系列分配的单个控制措施，根据类型和层面亲和关系度把它们关联到一起。这一步完成后，软件架构模式的控制措施叠加才算真正形成。借助NIST SP 800-53附录D（从第D-1页开始）完成这一步骤。切记，这里涉及的信息分类级别属于“低级”。需要注意的是，挑选控制的精确公式是不存在的，对控制措施的挑选会因风险偏好的不同而存在很大差异。

[44] https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final.