API 设计模式

奥拉夫·齐默尔曼
米尔科·斯托克
[美] 丹尼尔·吕布克 著
乌韦·兹敦
切萨雷·保塔索

蒋 楠 译

清華大學出版社
北 京

北京市版权局著作权合同登记号 图字：01-2023-1359

图书在版编目（CIP）数据

API设计模式 / (美) 奥拉夫·齐默尔曼等著 ; 蒋楠译.
北京 : 清华大学出版社, 2025. 3. -- ISBN 978-7-302-68172-4
Ⅰ. TP393.092.2
中国国家版本馆CIP数据核字第2025G0B171号

责任编辑：王　军
封面设计：高娟妮
版式设计：恒复文化
责任校对：成凤进
责任印制：沈　露

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：天津鑫丰华印务有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张：**24.75　　**字　　数：**617 千字
版　　次：2025 年 3 月第 1 版　　**印　　次：**2025 年 3 月第 1 次印刷
定　　价：128.00 元

产品编号：098655-01

各方赞誉

“API 正在席卷全球。软件开发过程中的组织和协作越来越依赖于 API。设计 API 时会遇到各种挑战，而使用模式是应对这些挑战的有效手段。本书有助于从业者提高 API 设计的效率：他们可以集中精力设计自己的应用程序，而标准的设计问题则可以通过使用模式解决。对于从事 API 相关工作的人士来说，本书会改变他们设计和理解 API 的方式。”

——Erik Wilde

Axway 公司 Catalyst 团队成员

“五位作者通过浅显易懂的方式探讨了 API 生命周期中涉及的各种设计模式，本书涵盖从定义到设计的各个阶段。无论是已经写过几十个 Web API 的老手，还是刚刚接触这一领域的新手，都能从阅读本书中受益。书中介绍的模式有助于保持 API 设计具有一致性，并帮助开发人员应对设计过程中可能遇到的各种挑战。本书非常值得一读。”

——James Higginbotham

《Web API 设计原则》作者

LaunchAny 公司创始人、首席 API 顾问

“如今，API 在软件开发领域无处不在。API 设计看似简单，但是所有饱受糟糕设计之苦的开发人员都有体会，掌握 API 设计的技巧并不容易，它远比表面看到的现象要精细和复杂。本书的五位作者运用丰富的经验和多年研究工作的成果，创建出一套关于 API 设计的结构化知识体系。本书不仅可以帮助读者理解用于构建高质量 API 所需掌握的基本概念，而且会提供一套实用的模式，可供开发人员在构建自己的 API 时使用。本书适合所有从事现代软件系统设计、开发或测试的人士阅读。”

——Eoin Woods

Endava 公司首席技术官

“在系统设计(尤其是在日益主导软件生态系统的分布式系统设计)中，需要进行大量权衡取舍，而 API 是帮助管理这些权衡取舍的关键要素之一。在我看来，本书通过使用易于理解的概念来消除 API 理解和设计过程中呈现出的复杂性，无论是已有一定经验的工程师，还是刚刚开始从事软件工程和架构设计的新人，都不难理解书中讨论的内容。如果你希望在系统设计中发挥关键作用，那么掌握书中介绍的 API 设计概念和模式至关重要。”

——Ipek Ozkaya

美国卡内基梅隆大学软件工程研究所软件解决方案部智能软件系统工程技术总监

2019—2023 年《IEEE 软件》期刊主编

“我相信，在当今这个时代中，‘API 优先’设计将成为大型复杂系统设计的主旋律。正因为如此，本书的出版恰逢其时，值得所有架构师一读。”

——Rick Kazman
美国夏威夷大学马诺阿分校教授

“终于看到系统探讨 API 设计这一重要话题的著作了！真希望能早几年读到本书，书中介绍的模式非常有用。”

——Gernot Starke 博士
INNOQ 研究员

“在我看来，某些软件项目之所以失败，是因为中间件技术掩盖了系统的分布式特性：程序员设计出适用于集中式系统的 API，但是在分布式环境中远程调用这种 API 时就会出现问题。本书探讨了在相互依存的世界中软件所需具有的分散性，并给出了设计各部分功能模块之间的接口时长期适用的建议。书中介绍的模式并不局限于某种特定的中间件技术，它们既能帮助开发人员创建和理解互联软件系统，也有助于管理这些系统目前和今后在功能上的必要演进。这些系统不仅在全球范围内的业务中发挥着作用，而且广泛应用于汽车、房屋以及日常生活所依赖的几乎每一项技术中。”

——Peter Sommerlad
独立咨询顾问
《面向模式的软件架构(卷 1)：模式系统》作者
Security Patterns 作者

“本书堪称软件工程师和架构师在 API 设计、演进、文档化方面的‘瑞士军刀’。我特别欣赏本书的一点是，五位作者并没有简单地将模式抛给读者，而是围绕现实示例进行解释，并提供实践性的架构决策支持，还通过案例研究来剖析模式和决策，因此理解书中介绍的模式语言可谓易如反掌。读者可以直接查找某个问题对应的解决方案，也可以通读各章以全面了解与 API 设计相关的问题和解决方案。所有模式经过精心设计且命名得体，并通过了从业者社区的同行评审。阅读本书是一种享受。”

——Uwe van Heesch 博士
执业软件架构师
Hillside Europe 前副总裁

“本书讨论的 44 种 API 模式涵盖各个方面，有助于软件工程师和架构师设计出具备互操作性的软件系统。五位作者不仅介绍了 API 的基础知识，还提供了大量研究示例。对于今后将要从事软件工程行业的人士来说，本书是一本优秀的教程。书中讨论的许多模式在实践中极为有用，已被应用于设计集成化、任务关键型铁路运营中心系统的 API。”

——Andrei Furda
日立轨道 STS 澳大利亚分公司高级软件工程师

序 一

我参与策划和编辑的“Addison-Wesley Signature Series 丛书”侧重于介绍软件开发过程中的有机增长和改进，下文会对此详细论述。在我看来，首先有必要介绍一下我与本书第一作者 Olaf Zimmermann 教授之间那种自然而然的交流。

我经常提到系统设计遵循康威定律。该定律指出，沟通是软件开发过程中的关键要素。系统设计不仅反映出设计者具有的沟通思维，此外个人在沟通过程中采用的语言组织和方式同样重要。良好的沟通可以促进有意义的讨论并激发思想火花，从而持续推动创新产品的开发。2019 年 11 月，我在瑞士伯尔尼举行的 Java 用户组(Java User Group，JUG)会议上遇到了 Zimmermann 教授。我当时做了关于反应式架构和编程的发言，并介绍了如何结合领域驱动设计来使用反应式架构。在我的发言之后，Zimmermann 教授进行了自我介绍。我还见到了他指导的研究生、后来成为同事的 Stefan Kapferer。他们两人相互合作，以“有机”的方式设计和开发了名为 Context Mapper 的开源产品(一种用于领域驱动设计的领域特定语言和工具)。与 Zimmermann 教授的偶遇最终促成了本书的出版。在介绍“Addison-Wesley Signature Series 丛书”出版的宗旨之后，我会进一步分享这次邂逅的故事。

“Addison-Wesley Signature Series 丛书”致力于帮助读者提高软件开发成熟度，并在以业务为中心的实践中取得更大成功。这套丛书强调采用各种方法(包括反应式架构、面向对象编程、函数式编程、领域建模、大小合适的服务、模式、API)对软件进行有机改进，并探讨相关基础技术的最佳用法。

接下来，我会重点谈谈什么是*有机改进*(organic refinement)。

一位朋友和同事最近使用“有机”一词来描述软件架构，从而引起了我的注意。虽然我在软件开发领域听说过也使用过这个词，但是在看到*有机架构*(organic architecture)这样的表达方式后，我开始深入思考“organic”的含义。

请读者想一想“organic”这个词，甚至“organism”这个词。二者在大多数情况下指代生物体，但有时也用来描述某些表现出类似生命特征的非生物体。“organic”一词源自希腊语，其词源与身体的功能器官有关。查一查“organ”的词源就会知道，这个词的含义很广泛。实际上，“organic”一词的含义也很广泛，它包括：身体器官、实施演进过程、某种功能性工具、乐器。

“organism”一词的第一种含义是“生物体”，涵盖从宏观的超大型生物到微观的单细胞生命形式，这方面的例子俯拾皆是。“organism”一词的第二种含义是“有机体”，相关的例子可能就不那么容易想到了。一个例子是组织(organization)，这个词由“有机”(organic)和“有机体”(organism)两部分构成，旨在描述一种具有双向依赖关系的结构化事物。之所以将组织比作有机体，是因为组织内部的各个部分相互依赖：组织需要各个部分来维持运作，各个部分也需要组

织来发挥作用。

从这个角度出发，可以将上述思维方式扩展到非生物体，因为它们表现出类似于生物体的特征。以原子为例，每个原子都是一个独立的系统，所有生物体都由原子组成。但是原子本身不属于生物体，而且不会繁殖。尽管如此，考虑到原子能够持续运动和相互作用，仍然可以将原子视为某种生物体。原子之间甚至会相互结合。一旦发生这种情况，那么每个原子就不仅仅是一个独立的系统，它们还与其他原子共同形成子系统，各个原子结合后所表现出的组合行为会催生出一个更大的整体系统。

因此，从某种程度上说，任何与软件相关的概念都是“有机”的，因为即使是非生物体也会表现出生物体具有的某些特征。当我们通过具体场景讨论软件模型概念、绘制架构图、编写单元测试及其相应的领域模型单元时，软件就开始变得鲜活起来。软件并非一成不变，因为我们在不断讨论如何改进和优化软件，其中一个场景的变化会引发另一个场景出现变化，从而对架构和领域模型产生影响。在持续迭代的过程中，改进软件带来的价值可以促使有机体逐步增长。随着时间的推移，软件也在不断发展。我们通过使用有用的抽象来处理和解决复杂性问题，软件在发展，其功能也在发生变化。所有这些工作都有一个明确的目的，那就是在全球范围内更好地为人类服务。

遗憾的是，与有机体一样，软件往往难以“茁壮成长”，而且经常出现“发育不良”的情况：即使初期一切正常，但随着时间的推移也容易生病(出现故障)、变形(功能异常)、长出不自然的附属物(增加不必要的功能)、萎缩(功能失效)或恶化(质量下降)。更糟糕的是，尽管我们投入时间和精力去改进存在问题的软件，但是这些努力不仅没能提高软件质量，反而带来更多问题。最糟糕的是，即使改进失败导致问题恶化，但软件依然不会完全崩溃，从而令对这些问题的处理变得更加棘手——其实，软件要是真的崩溃就好了。我们不得不选择把软件“置于死地”。要做到这一点，需要具备屠龙者一样的勇气、技巧和坚韧不拔的意志——不是一位屠龙者，而是几十位精力充沛的屠龙者。实际上，解决复杂的软件问题通常需要几十位具有极高智慧和专业技巧的“屠龙者”。

“Addison-Wesley Signature Series 丛书”的目的就在于此。如前所述，我参与策划和编辑的这套丛书致力于帮助读者通过使用各种方法(包括反应式架构、面向对象编程、函数式编程、领域建模、大小合适的服务、模式、API)来提高软件开发成熟度并取得更大成功。除此之外，这套丛书还会探讨相关基础技术的最佳用法。实现这些目标不是一朝一夕之功，而是需要有计划、有技巧地进行逐步改进。我和这套丛书的其他作者随时准备提供帮助。为此，我们尽最大努力来实现既定目标。

现在接着聊一聊我的故事。与 Zimmermann 教授初次见面时，我邀请他和 Kapferer 在几周后前往德国慕尼黑，参加我组织的“实现领域驱动设计”工作坊(IDDD Workshop)。虽然他们两人无法抽出时间全程参与，但是愿意参加第三天也就是最后一天的活动。我的第二个提议是请 Zimmermann 教授和 Kapferer 在工作坊结束后抽空演示 Context Mapper 工具。与会者和我对这款工具印象深刻，这也促成我们在 2020 年有了进一步合作。尽管如此，我和 Zimmermann 教授还是设法经常见面，并继续讨论 Context Mapper 的设计。记得有次见面时，Zimmermann 教授提到他在公开提供 API 模式方面所做的努力，并向我展示了部分 API 模式以及他和其他人围绕这些模式所开发的附加工具。我邀请 Zimmermann 教授参与“Addison-Wesley Signature Series

丛书”的编写工作，本书就是取得的成果。

后来，我与 Zimmermann 教授和 Daniel Lübke 进行视频通话，启动了本书的编写工作。虽然我没有与另外三位作者 Mirko Stocker、Uwe Zdun 和 Cesare Pautasso 进行过交流，但是考虑到他们的资历，我对作者团队的专业水平充满信心。值得一提的是，Zimmermann 教授和 James Higginbotham 通力合作，确保本书与同属“Addison-Wesley Signature Series 丛书”的《Web API 设计原则》(*Principles of Web API Design*)在内容上互为补充。总体而言，本书五位作者对行业文献所做的贡献给我留下了深刻印象。API 设计的重要性毋庸置疑。本书出版后反响热烈，表明它恰好切中 API 设计的“痛点”。相信读者也会认同本书的价值。

——Vaughn Vernon
“Addison-Wesley Signature Series 丛书”的编辑
《实现领域驱动设计》一书的作者

序 二

API 无处不在。API 经济推动了云计算、物联网等技术领域的创新，也是众多公司实现数字化转型的关键推手。几乎所有企业应用程序都有用于集成客户、供应商和其他业务合作伙伴的外部接口，而解决方案内部接口将应用程序拆分为更容易管理的模块(例如，松散耦合的微服务)。基于 Web 的 API 在分布式环境中至关重要，但是这些 API 并非集成远程方的唯一手段：基于队列的消息传递通道和基于发布/订阅的通道广泛用于后端集成，并通过公开 API 来实现消息生产者与消息使用者之间的交互。gRPC 和 GraphQL 的发展势头也很强劲，且受到越来越多的关注。因此，进行 API 设计时有必要遵循业内认可的最佳实践。理想情况下，API 设计应该在各种技术环境中保持一致，并在技术发生变化时仍然能够正常运行。

模式相当于一套词汇表，用于描述特定问题及其解决方案。模式在抽象与具体之间取得平衡，从而既有永恒性，又有现实意义。以 Gregor Hohpe 和 Bobby Woolf 合著、同属“Addison-Wesley Signature Series 丛书”的《企业集成模式》(*Enterprise Integration Patterns*)为例，自从我担任 IBM MQ 系列产品的首席架构师以来，就一直在企业培训和行业项目中使用这本书。各种消息传递技术“你方唱罢我登台”，但是服务激励器(Service Activator)、幂等接收者(Idempotent Receiver)等消息传递的概念始终存在。我自己写过云计算模式、物联网模式和量子计算模式，甚至还写过用于数字人文领域的模式。由 Martin Fowler 撰写、同属“Addison-Wesley Signature Series 丛书”的《企业应用架构模式》(*Patterns of Enterprise Application Architecture*)介绍了远程外观(Remote Facade)、服务层(Service Layer)等模式。这些图书深入探讨了设计分布式应用程序时需要考虑的各种因素，但是并没有涵盖所有内容。因此，我很高兴看到 API 设计空间如今也得到了模式的支持——对于 API 客户端与 API 提供者之间传输的请求消息和响应消息来说，模式尤其重要。

本书的五位作者均为资深的架构师和开发人员，既有经验丰富的行业专家和设计模式领域的“大 V”，也有学术研究者和讲师。我与其中三位作者共事多年，自 2016 年他们启动微服务 API 模式(Microservice API Pattern，MAP)项目以来，我就一直在关注项目的进展情况。他们忠实地运用了模式概念：每种模式文本基于一套通用模板构建而成，首先描述问题背景(包括设计要素)，然后提出概念性的解决方案，并附有具体的示例(通常是 RESTful HTTP)，同时针对优缺点进行批判性讨论以解决设计初期面临的问题，最后给出使用相关模式的建议。许多模式经过模式会议的指导和作者工作坊的讨论，获得的反馈能够帮助这些模式在几年时间里反复改进、逐步完善，从而在此过程中形成集体知识。

本书致力于从多方面、多角度探讨 API 设计，包括范围界定、架构、消息表示结构、质量属性驱动的设计以及 API 演进。可以通过不同的方式来浏览本书介绍的模式语言，这些方式包

括项目阶段或结构元素(例如，API 端点和操作)。本书使用图形图标来表述每种模式包含的核心内容，这一点与 *Cloud Computing Patterns* 一书的做法类似。这些图标不仅可以作为记忆辅助工具，还能用于描绘 API 及其元素。本书提供独特而新颖的决策模型，汇集了有关模式应用的常见问题、可选方案和评判标准。这些模型提供循序渐进、简单易懂的设计指导方针，同时也保留了 API 设计中固有的复杂性。通过在一个示例案例中逐步运用书中讨论的决策模型，读者能够更容易理解这些模型及其建议。

在阅读本书第II部分(模式参考部分)时，应用和集成架构师会发现，端点角色(如PROCESSING RESOURCE)和操作职责(如 STATE TRANSITION OPERATION)的相关内容十分有用，可以帮助他们合理确定 API 的规模并做出部署决策(尤其是云环境中的部署决策)。毕竟，状态是 API 设计中的重要因素，而书中介绍的某些模式专门用于处理 API 背后需要进行的状态管理。此外，本书深入探讨了标识符的应用(如 API KEY、ID ELEMENT 等模式)和多种响应塑造方案(例如，使用从 GraphQL 抽象而来的 WISH LIST 和 WISH TEMPLATE)，并针对如何公开不同类型的元数据给出了实用建议，这些内容都能使 API 开发人员受益。

到目前为止，我还没有看到其他图书采用模式的形式来记录生命周期管理和版本控制策略。本书介绍的 LIMITED LIFETIME GUARANTEE 和 TWO IN PRODUCTION 模式在企业应用程序中十分常见，这些演进模式会受到 API 产品负责人和维护人员的青睐。

总而言之，本书既有理论基础，也有实际应用，在包含大量深刻建议的同时又不失大局观。书中介绍的 44 种模式分为五大类，它们从实际的项目提炼而来，不仅经过了严谨的学术论证，还吸收了业界的意见和建议。我相信，无论现在还是将来，这些模式都会对模式社区有所裨益。无论是业内的 API 设计人员，还是从事与 API 设计和演进工作相关的研究者、开发者和教育者，都能从阅读本书中受益。

——Frank Leymann 教授
欧洲科学院院士
德国斯图加特大学应用系统架构研究所所长

前　言

前言包括以下内容。

- 本书的背景和宗旨：动机、目标和范围。
- 适合阅读本书的人群：目标读者及其使用场景和信息需求。
- 本书的组织方式：以模式作为知识载体。

P.1　动机

人类可以用多种不同的语言进行交流，软件也是如此。软件不仅可以采用各种编程语言编写，还可以通过多种协议(如 HTTP)和消息交换格式(如 JSON)来传输数据。每当用户更新自己的社交网络个人资料、在线下单、刷卡购物时，HTTP、JSON 以及其他技术就会发挥作用。

- 应用程序前端(例如，智能手机上运行的移动应用程序)向应用程序后端(例如，网上商城的订单系统)发出交易处理请求。
- 应用程序的各个模块会交换长时间存在的数据(例如，客户配置文件或产品目录)，也会与业务合作伙伴、客户和供应商的系统进行数据交换。
- 应用程序后端提供外部服务(例如，支付网关或云存储)所需的数据和元数据。

这些场景中涉及的软件组件(无论是大型、中型还是小型组件)彼此通信，以实现各自的目标并共同服务于最终用户。为了应对这种分布式挑战，软件工程师通过*应用程序编程接口*(Application Programming Interface，API)进行应用程序集成。每个集成场景至少涉及两类通信参与者：API 客户端和 API 提供者。API 客户端使用 API 提供者对外公开的服务，API 文档则规定了客户端与提供者之间的交互方式。

就像人类在交流时可能产生误会一样，软件组件在通信过程中也常常难以理解对方传递的信息。设计人员很难确定消息中应该包含多少信息以及如何组织这些信息，也很难就最适合的对话风格达成一致。在表达需求或响应请求时，通信参与者既不希望信息过少，也不希望信息过多。有些应用程序集成和 API 设计非常成功，相关方能够理解对方传递的信息并达成各自的目标，彼此之间的数据交换既有效又高效。有些应用程序集成和 API 设计则缺乏清晰度，从而令参与者感到困惑或存在压力。冗长的消息和碎片化的对话不仅可能使通信通道不堪重负，而且会带来无谓的技术风险，还可能增加开发和运营过程的工作量。

那么，如何评判集成 API 设计的优劣？API 设计人员应该采取哪些措施以促进积极的客户

端开发者体验？理想情况下，有关如何设计出高质量集成架构和 API 的指导方针不应依赖于任何特定的技术或产品。技术和产品“你方唱罢我登台”，但是相关的设计建议应该在很长一段时间内保持适用性。不妨用现实世界中的事物进行类比：无论是古罗马政治家西塞罗采用的修辞和雄辩原则，还是美国心理学家 Marshall Rosenberg 在《非暴力沟通》(*Nonviolent Communication: A Language of Life*)[Rosenberg 2002]一书中提出的原则，都没有局限于使用英语或其他任何自然语言。这些原则具有普适性，不会随着自然语言的发展而过时。我们撰写的这本书旨在为集成专家和 API 设计人员提供一套类似的工具和术语。本书围绕 API 设计和演进所采用的模式来探讨各个知识点，这些模式适用于各种通信范式和技术(我们主要以基于 HTTP 和基于 JSON 的 Web API 为例进行讨论)。

目标和范围

我们致力于通过采用行之有效、可以重复使用的解决方案元素来帮助克服 API 设计和演进过程中存在的复杂性。

> 如何从利益相关者的目标、重要的架构需求和已经得到验证的设计元素入手，设计出既具备可理解性又具备可持续性的 API?

虽然关于 HTTP、Web API 和集成架构(包括面向服务的架构)的讨论和资料比比皆是，但是到目前为止，有关单个 API 端点和消息交换的设计尚未得到足够的关注：

- 远程公开的 API 操作数量以多少为宜？请求消息和响应消息中应该交换哪些数据？
- 如何确保 API 操作和客户端-提供者交互具有松耦合？
- 消息表示应该采用扁平结构还是层次嵌套结构？如何就表示元素的含义达成一致，以便正确理解并有效处理这些元素？
- API 提供者应该负责处理 API 客户端发送的数据(可能需要更改提供者端状态并连接到后端系统)，还是只负责为客户端提供共享数据存储？
- 如何以一种受控的方式对 API 进行更改，以便在支持扩展性的同时不会破坏兼容性？

针对在某些需求背景中反复出现的特定设计问题，本书介绍的模式概括出了一些行之有效的解决方案，能够在一定程度上回答上述问题。这些模式侧重于处理远程 API(而不是程序内部 API)，旨在改善客户端和提供者端的开发者体验。

P.2 目标读者

本书针对具备一定基础、希望进一步提高技能和设计水平的软件专业人员，书中介绍的模式主要面向对平台无关的架构知识感兴趣的集成架构师、API 设计人员和 Web 开发人员。无论

是专注于后端集成的专家，还是负责为前端应用程序提供支持的 API 开发人员，都可以从这些模式提供的知识中获益。本书侧重于探讨 API 端点粒度和通过消息交换产生的数据，因此对 API 产品负责人、API 审核员、云租户和云提供商也有一定的参考价值。

> 本书适合具有一定经验的软件工程师(如开发人员、架构师或产品负责人)阅读，他们已经了解 API 的基础知识，并且希望提高自己的 API 设计水平(包括如何设计消息数据契约和如何实现 API 演进)。

对学生、讲师和从事软件工程研究的人员来说，本书介绍的模式及其展示方式可能也很有用。我们会介绍 API 的基础知识和 API 设计对应的领域模型，以确保即使读者不了解 API 的基础知识，也能理解本书讨论的内容及其模式。

如果读者了解现有的模式及其优缺点，则可以提高 API 设计和演进方面的熟练程度。通过应用书中推荐的适合特定需求背景的模式，开发、使用和演进 API 及其提供的服务将变得更容易。

P.2.1 使用场景

我们希望 API 的设计和使用过程能带来令人愉悦的体验。为此，本书及其讨论的模式有如下三种主要的使用场景：

1. 构建统一的词汇表、明确需要做出的设计决策并分享可选方案和相关的权衡取舍，以促进开展 API 设计方面的讨论和工作坊。掌握这些知识后，API 提供者就可以设计出高质量且风格独特的 API，从而既能满足客户端的短期需求，又能满足客户端的长期需求。
2. 简化 API 设计审查的流程并加快对 API 进行客观比较的速度，以保证 API 的质量，并以向后兼容和可扩展的方式实现 API 的演进。
3. 提高 API 文档的质量，加入不依赖于平台的设计信息，以便 API 客户端开发人员更容易理解所提供的 API 具有的功能和限制。这些模式可以直接嵌入 API 契约，而且不仅能应用于新的设计，也能应用于现有的设计。

为了展示这些模式的用法并帮助读者开始使用模式，本书将讨论一个虚构的案例，并给出两个在实际项目中运用模式的案例。

读者不需要掌握任何特定的建模方法、设计技术或架构风格就能阅读本书，但是了解对齐-定义-设计-完善(Align-Define-Design-Refine，ADDR)过程、领域驱动设计(Domain Driven Design，DDD)等概念会有一定帮助。附录 A 将简要回顾这些概念。

P.2.2 现有的设计启发法(和知识空白)

关于特定的 API 技术和概念，有不少图书值得一读，它们提供了深入的建议。举例来说，《RESTful Web Services Cookbook 中文版》(*RESTful Web Services Cookbook*)[Allamaraju 2010]围绕如何构建 HTTP 资源 API 展开讨论，例如，选择哪种 HTTP 方法(POST 还是 PUT)。其他图

书从路由、转换、保证交付等方面解释了异步消息传递机制的工作原理，有兴趣的读者不妨读一读《企业集成模式》(*Enterprise Integration Patterns*)[Hohpe 2003]。《领域驱动设计》(*Domain-Driven Design*)[Evans 2003]和《实现领域驱动设计》(*Implementing Domain-Driven Design*)[Vernon 2013]致力于探讨战略性领域驱动设计，对于初步了解 API 端点和服务识别有一定帮助。目前，市面上已有介绍面向服务的体系结构、云计算和微服务基础设施模式的图书。关于数据存储(包括关系数据库和 NoSQL 数据库)的结构也有详细的资料，还有一整套用于分布式系统设计的模式语言，相关讨论可参见《面向模式的软件架构：分布式计算的模式语言》(*Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*)[Buschmann 2007]。此外，《发布！软件的设计与部署》(*Release It!: Design and Deploy Production-Ready Software*)[Nygard 2018a]详细讨论了操作设计和生产环境部署。

对于 API 设计流程方面的内容(包括如何根据目标进行端点识别以及如何设计操作)，已经出版的图书也做过深入探讨。举例来说，《Web API 设计原则》(*Principles of Web API Design*)[Higginbotham 2021]一书建议采用包括七个步骤的四个流程阶段。《Web API 设计》[Lauret 2019]一书提出 API 目标画布的概念，《设计并构建 Web API》[Amundsen 2020]一书的讨论则涉及 API 故事。

尽管上面提到的这些图书提供了有价值的设计建议，但是远程 API 设计领域的文献资料仍然不够全面。具体来说，API 客户端与提供者之间传输的请求消息和响应消息应该采用何种结构？《企业集成模式》[Hohpe 2003]一书虽然介绍了三种用于表示消息类型(事件消息、命令消息、文档消息)的模式，但是并没有详细解释这些模式的内部机制。然而，系统之间交换的“外部数据”不同于程序内部处理的“内部数据”[Helland 2005]，这两类数据在可变性、生命周期、准确性、一致性、保护需求等方面存在显著差异。举例来说，制造商与物流公司通过远程 API 和消息传递通道共同管理供应链时交换产品定价和运输信息需要的架构设计比较复杂，而在库存系统内部增加本地库存项计数器需要的架构设计可能相对简单。

本书主要探讨消息表示设计(也就是外部数据[Helland 2005]或 API 的 PUBLISHED LANGUAGE [Evans 2003])，致力于填补有关 API 端点、操作和消息设计的知识空白。

P.3 作为知识共享载体的模式

软件模式是复杂的知识共享工具，至今已有超过 25 年的历史。我们之所以选择采用模式的形式来分享 API 设计的建议，是因为模式名称旨在构建一个领域专用的术语体系，也就是所谓的“通用语言”(ubiquitous language)[Evans 2003]。例如，基于队列的消息传递机制广泛使用各种企业集成模式，消息传递框架和工具甚至也实现了这些模式。

模式并非闭门造车的产物，而是根据实践经验提炼而来的，然后根据同行反馈不断完善。模式社区已经发展出一套用于组织反馈流程的实践方法，指导(shepherding)和作者工作坊(writers' workshop)是其中两种特别重要的实践方法[Coplien 1997]。

每种模式的核心内容围绕一个问题以及相应的解决方案展开。通过探讨影响决策的因素和由此产生的后果，有助于开发人员在评估期望达到和实际达到的质量特征以及某些设计的缺点时做出正确的决策。我们还会讨论替代解决方案，并提供相关模式和技术实现的建议，以便读

者全面理解并应用这些模式。

需要注意的是，模式的意义不在于提供完整的解决方案，而在于提供概念性的指导方针，使用者可以根据具体的 API 设计背景进行调整。换句话说，模式很灵活，旨在描述可行的解决方案，但是不提供“无脑复制”的蓝图。如何采用和实现模式以满足项目要求或产品需求，仍然要由 API 设计人员和负责人决定。

长期以来，本书的五位作者一直在工业界和学术界应用和教授模式。有的作者已经写过用于程序设计、架构设计以及分布式应用程序系统与其部件集成的模式[Voelter 2004; Zimmermann 2009; Pautasso 2016]。

在我们看来，模式概念非常契合 P.1 节和 P.2 节讨论的使用场景。

P.3.1 微服务 API 模式

我们提出的模式语言称为*微服务 API 模式*(Microservice API Patterns)，这种缩写为“MAP”的语言从公开和使用 API 时所交换的消息的角度帮助使用者全面理解 API 设计和演进。这些消息及其有效载荷按照表示元素的形式进行组织。由于 API 端点及其操作的架构职责不同，因此表示元素的结构和含义也不同。消息结构会显著影响 API 及其底层实现的设计时质量和运行时质量。以网络和端点负荷(例如 CPU 消耗和网络带宽使用)为例，少量较长的消息与大量较短的消息并不一样。此外，成功的 API 会随着时间的推移而演进，因此需要妥善管理这些变化。

“MAP”还有“地图”之意。之所以选择这一隐喻和缩写词来表示我们提出的模式语言，是因为地图和模式语言都能提供方向和指导，可以帮助使用者了解在抽象的解决方案空间中有哪些可用选项。API 将来自客户端的请求路由到底层服务实现，因此 API 本身也具有映射性质。

我们承认，将模式语言命名为“微服务 API 模式”可能是一种“吸引眼球”的做法。如果微服务在本书出版后不再流行，那么我们保留更改模式语言名称和重新使用缩写词的权利。例如，缩写同样为“MAP”的“消息 API 模式”(Message API Patterns)也能很好地概述模式语言的适用范围。本书通常将“MAP”称为“模式语言”或“我们提出的模式”。

P.3.2 本书模式的适用范围

本书是一个志愿者项目的最终成果。该项目于 2016 年秋季启动，致力于处理 Web API 和其他远程 API 的设计和演进，以解决端点和消息职责、结构、质量问题以及服务 API 演进。本书介绍的模式语言有助于回答以下问题：

- 每个 API 端点的架构角色是什么？端点角色和操作职责如何影响服务规模和服务粒度？
- 在请求消息和响应消息中，表示元素的数量以多少为宜？这些元素的结构如何？怎样对表示元素进行分组，并为这些元素添加补充信息？
- API 提供者如何确保 API 的质量既能达到一定水平，又能以经济有效的方式利用现有资源？如何向相关方传达和解释质量方面的权衡？
- API 专业人员如何处理支持期、版本控制等生命周期管理方面的问题？如何在更新 API 时保持向后兼容性，并将不可避免的破坏性变更告知相关方？

在动手编写模式之前，我们研究了大量 Web API 和 API 相关规范，并结合自身的专业经验进行思考。我们发现，无论是在公共 Web API 中，还是在行业应用程序开发和软件集成项目中，都能看到这些模式得到了实际应用(已知用途)。从 2017—2020 年，许多模式的过渡版本经过欧洲程序模式语言会议(EuroPLoP)[1]的指导和作者工作坊的讨论，后来被收录在相关的会议论文集中[2]。

P.3.3　切入点、阅读顺序和内容组织

在复杂的设计过程中处理棘手问题(API 设计有时无疑也属于棘手问题，这个词的解释可参考维基百科[Wikipedia 2022a])时，往往难以把握全局，经常出现“只见树木，不见森林”的情况。问题解决活动很复杂，不可能也不适合将其分解为一系列预定义的步骤或标准化流程。因此，我们提出的模式语言提供了多个切入点。读者可以根据自己的情况，从本书任何一个部分开始阅读。更多建议请参见附录 A。

本书分为**基础知识概述**、**模式**、**实践应用**等三个部分，图 P-1 显示了各个部分包含的章节和逻辑依赖关系。

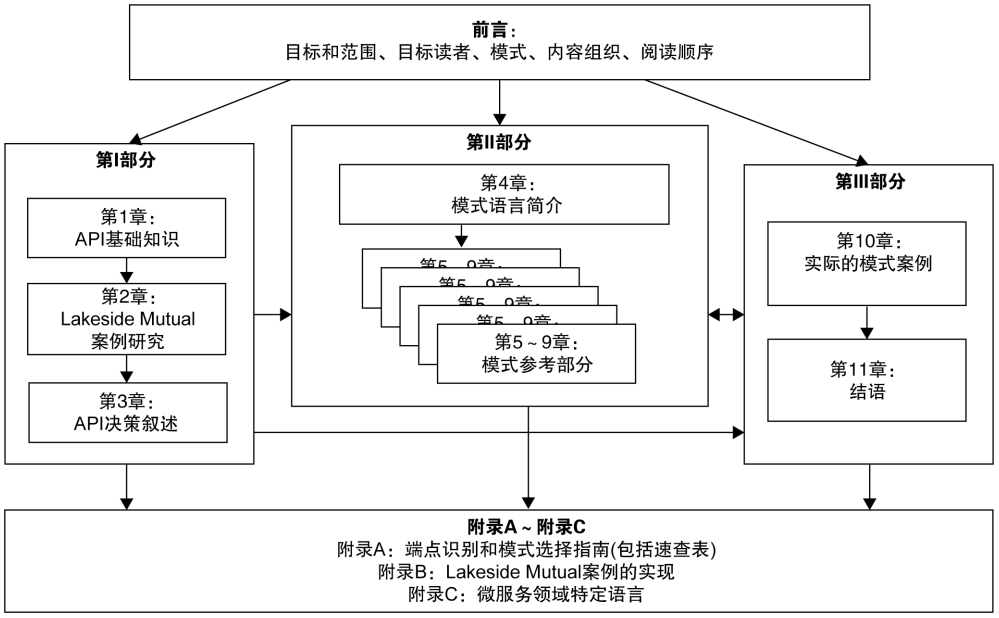

图 P-1　本书各个部分及其依赖关系

第 I 部分包括第 1～3 章。第 1 章主要从概念层面介绍 API 设计。第 2 章将首次介绍 Lakeside Mutual，我们以这家虚构的保险公司作为案例研究，书中许多示例也来自 Lakeside Mutual。这

1　https://europlop.net/content/conference.

2　我们决定不在本书中包含大量已知的用例，此类信息可在网上和 2016 年至 2020 年的 EuroPlop 会议记录中找到。在一些补充资料中，你还可以找到额外的实现提示。

一章的内容包括 Lakeside Mutual 的业务背景、需求、现有系统以及初步的 API 设计。第 3 章将探讨决策模型，通过这些模型可以了解模式语言中各种模式之间的关系。我们还会讨论模式选择标准，并通过 Lakeside Mutual 案例解释如何做出关键的决策。这些决策模型既能帮助读者理解本书的内容，也能帮助开发人员在实际应用模式时做出明智的决策。

第 II 部分是模式参考部分，包括第 4～9 章。第 4 章将概括介绍模式语言，第 5～9 章将深入讨论所有 44 种模式。第 II 部分的章节结构和可能的阅读顺序如图 P-2 所示。一种方案是先阅读第 4 章以了解 ATOMIC PARAMETER、PARAMETER TREE 等基本结构模式，再阅读第 6 章以了解 ID ELEMENT、METADATA ELEMENT 等元素构造型。

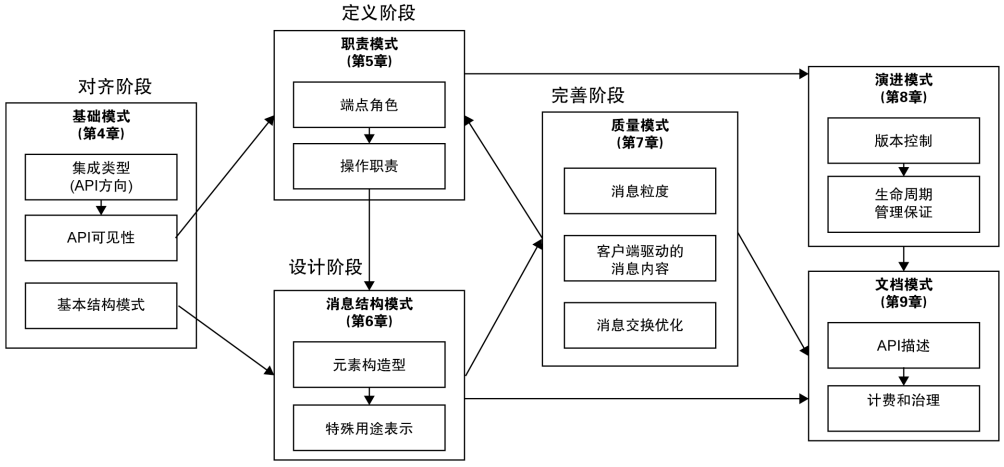

图 P-2 超级模式图：第 II 部分的章节结构和各章之间的关系

每种模式的描述相当于一篇小型专业文章，长度大多在几页。这些描述的组织结构完全相同：首先介绍模式的应用时机和理由，其次解释模式的运行机制并给出至少一个具体示例，然后分析应用某种模式会产生哪些后果，并在读者了解该模式的用法后介绍其他可能适用的模式。我们的模式名称采用**小型大写字母**格式，即所有字母均为大写字母，但是高度比标准的大写字母稍低(如 PROCESSING RESOURCE)。第 4 章将详细介绍模式模板，该模板从 EuroPLoP 会议提出的模板中衍生而来[Harrison 2003]。根据评审过程中得到的意见和建议，我们对模板进行了小幅度重构(感谢 Gregor 和 Peter 的帮助)。我们介绍的模式旨在处理对架构有重大影响的需求，所以特别强调质量属性及其冲突。正因为如此，在制定 API 设计和演进的决策时需要进行权衡，以便找出最合适的解决方案。

第 III 部分包括第 10～11 章。第 10 章以两个不同领域的项目为例探讨模式的实际应用，这两个项目分别涉及电子政务和建筑行业的报价/订单管理。对于 API 今后的发展，第 11 章将给出我们的反思和展望。

附录 A 会提供一份以问题为导向的速查表，也可以由此入手阅读本书。我们还会讨论 44 种模式与职责驱动设计(Responsibility Driven Design，RDD)、领域驱动设计(Domain Driven Design，DDD)、对齐-定义-设计-完善(Align-Define-Design-Refine，ADDR)过程之间的关系。对于 Lakeside Mutual 案例研究，附录 B 会提供 API 设计的更多细节。附录 C 将概括介绍微服务

领域特定语言(Microservice Domain Specific Language，MDSL)。MDSL 是一种用于微服务契约的语言，通过<<Pagination>>等装饰器提供内置模式支持。MDSL 能够与 OpenAPI、gRPC 协议缓冲区、GraphQL 以及其他接口描述语言和服务编程语言进行绑定，并为这些语言提供生成器支持。

本书部分示例采用 Java 编写(但是 Java 代码的数量不会很多)，还会涉及大量 JSON 和 HTTP 方面的内容(例如，curl 命令及其响应)。有极少数内容可能与 gRPC、GraphQL 和 SOAP/WSDL 有关，不过无须担心，这些内容会以简单易懂的方式呈现出来，即使读者不了解相关技术的专业知识也能理解。此外，部分示例采用 MDSL 进行描述(我们之所以要开发一种新的接口描述语言，是因为当示例的复杂程度超出“Hello World”的范畴时，就不适合采用 OpenAPI 的 YAML 或 JSON 格式进行描述，否则会导致内容过长，无法在一页篇幅中展示)。

更多信息请浏览本书的配套网站 https://api-patterns.org。

希望读者能够受益于我们的工作成果，从而使我们提出的模式有机会被全球的集成架构师和 API 开发人员采纳，并融入他们的知识体系中。我们很乐意听取你的反馈和建设性批评。

Olaf Zimmermann
Mirko Stocker
Daniel Lübke
Uwe Zdun
Cesare Pautasso

致 谢

感谢 Vaughn Vernon 在我们撰写本书期间提供的所有反馈和鼓励，我们很荣幸参与了他主持编辑的“Addison-Wesley Signature Series 丛书”项目。特别感谢培生集团的 Haze Humbert、Menka Mehta、Mary Roth、Karthik Orukaimani 和 Sandra Schroeder 给予我们的大力支持，也感谢 Frank Leymann 教授为本书作序并提供宝贵意见。在文字编辑 Carol Lallier 的帮助下，本书的后期处理十分顺利，对我们来说是一次愉快的经历。

本书第 10 章以两个实际项目为例探讨模式的应用，相关信息由 Terravis 平台和 INNOQ 咨询公司提供，所以我们很感谢 Walter Berli 和 Werner Möckli(Terravis)以及 Ghadir 和 Willem van Kerkhof(INNOQ)为此付出的努力。Nicolas Dipner 和 Sebnem Kaslack 在他们的学期论文和学士学位项目中设计了最初的模式图标。Toni Suter 编写了 Lakeside Mutual 示例应用程序的大部分代码。作为 Context Mapper 项目的创始人和开发者，Stefan Kapferer 也对 MDSL 工具有所贡献。

感谢所有对本书内容提出意见和建议的人士。Andrei Furda 对书中的介绍性材料提出了自己的看法，他还审查了许多模式的内容；Oliver Kopp 和 Hans-Peter Hoidn 在自己的项目中运用了我们提出的模式并给出了反馈，还与同行组织了几次非正式工作坊；James Higginbotham 和 Hans- Peter Hoidn 审阅了本书手稿。在此向他们深表谢意。

此外，许多同行的反馈令我们受益匪浅，尤其是参加了 EuroPLoP 2017、EuroPLoP 2018、EuroPLoP 2019、EuroPLoP 2020 的指导者和作者研讨会的参与者给出的反馈。感谢以下人士提出的宝贵意见：Linus Basig、Luc Bläser、Thomas Brand、Joseph Corneli、Filipe Correia、Dominic Gabriel、Antonio Gámez Díaz、Reto Fankhauser、Hugo Sereno Ferreira、Silvan Gehrig、Alex Gfeller、Gregor Hohpe、Stefan Holtel、Ana Ivanchikj、Stefan Keller、Michael Krisper、Jochen Küster、Fabrizio Lazzaretti、Giacomo De Liberali、Fabrizio Montesi、Frank Müller、Padmalata Nistala、Philipp Oser、Ipek Ozkaya、Boris Pokorny、Stefan Richter、Thomas Ronzon、Andreas Sahlbach、Niels Seidel、Souhaila Serbout、Apitchaka Singjai、Stefan Sobernig、Peter Sommerlad、Markus Stolze、Davide Taibi、Dominic Ullmann、Martin (Uto869)、Uwe van Heesch、Timo Verhoeven、Stijn Vermeeren、Tammo van Lessen、Robert Weiser、Erik Wilde、Erik Wittern、Eoin Woods、Rebecca Wirfs-Brock 以及 Veith Zäch。还要感谢选修瑞士东部应用科技大学拉珀斯维尔技术学院的《高级模式和框架》和《应用程序架构》课程、选修瑞士意大利语区大学的《软件架构》课程的几届学生。他们在课上讨论了我们提出的模式，并给出了意见和建议。

作 者 简 介

Olaf Zimmermann 是一位长期从事服务导向的专家，拥有架构决策建模领域的博士学位。他是瑞士东部应用科学大学软件学院软件架构学的顾问和教授，致力于研究敏捷架构设计、应用集成、云原生、领域驱动设计以及面向服务的系统。Zimmermann 曾在 ABB 公司和 IBM 公司担任软件架构师，为世界各地的电子商务客户和企业应用程序开发客户提供服务，早年编写过用于系统和网络管理的中间件。Zimmermann 是国际开放标准组织(The Open Group，TOG)授予的杰出(首席)IT 架构师，也是《IEEE 软件》期刊 Insights 专栏的联合编辑。他著有 *Perspectives on Web Services* 一书，也是 IBM 红皮书 *Eclipse Development* 第一版的作者。Zimmermann 在个人网站和博客平台 Medium 发表文章。

Mirko Stocker 是一位热爱编程的开发者，他对选择前端开发还是后端开发犹豫不决，所以决定从事介于二者之间的工作——API 开发，并发现 API 也蕴藏着许多有趣的挑战。Stocker 与他人共同创办了两家法律技术领域的初创公司，目前仍然担任其中一家公司的首席执行官。Stocker 的职业经历助力他成为瑞士东部应用科学大学软件工程学教授，负责编程语言、软件架构、Web 工程等领域的研究和教学工作。

Daniel Lübke 是一位独立的编程和咨询软件架构师，专门从事业务流程自动化和数字化项目。他的研究方向包括软件架构、业务流程设计和系统集成，这些领域本质上需要使用 API 来开发解决方案。Lübke 于 2007 年获得德国汉诺威大学博士学位，此后参与过不同领域的多个行业项目，积累了丰富的经验。他撰写过几种图书，也是许多文章和研究论文的作者和编辑。Lübke 为客户提供培训，并定期在 API 和软件架构的相关会议上发表演讲。

Uwe Zdun 是奥地利维也纳大学计算机科学学院软件架构研究组的全职教授，研究方向为软件设计和架构、经验软件工程、分布式系统工程(微服务、基于服务的系统、云计算、API、区块链系统)、DevOps 和持续交付、软件模式、软件建模以及模型驱动开发。Zdun 参与过相关领域的众多研究项目和行业项目，并发表了 300 多篇学术文章，还与他人合著有 *Remoting Patterns* 和 *Process-Driven SOA* 两本专业图书。

Cesare Pautasso 是瑞士意大利语区大学信息学院软件研究所的全职教授，也是该校架构、设计和 Web 信息系统工程研究小组的负责人。他曾担任第 25 届欧洲程序模式语言会议(EuroPLoP 2022)主席。Pautasso 于 2004 年获得瑞士苏黎世联邦理工学院博士学位，2007 年在 IBM 苏黎世研究实验室短暂工作期间有幸结识 Zimmermann。Pautasso 是《SOA 与 REST：用 REST 构建企业级 SOA 解决方案》(*SOA with REST*)一书的合著者，并通过自出版平台 Leanpub 出版了 *Beautiful APIs*、*Beautiful API Evolution*、*RESTful Dictionary*、*Just Send an Email: Anti-patterns for Email-centric Organizations* 等多种图书。

目　录

第Ⅰ部分　基础知识概述

第 I 部分

基础知识概述

第 I 部分介绍的三章知识旨在为开发人员充分理解本书内容奠定基础。第 1 章将介绍 API 的基本概念，并解释远程 API 的重要性和设计 API 时面临哪些挑战，以便为开发人员阅读后续章节铺平道路。

第 2 章将介绍 Lakeside Mutual 案例研究，并以此作为贯穿全书的示例。Lakeside Mutual 是一家虚构的保险公司，其所使用的系统体现出 API 模式的实际运用情况。

第 3 章围绕需要做出的具体决策来概述各种模式，第 II 部分将详细讨论这些模式。每种决策致力于解答 API 设计存在的某个问题，这些模式为解决方案提供了备选项。我们还会介绍 Lakeside Mutual 的决策结果。第 3 章讨论的决策模型有助于开发人员组织 API 设计工作，也可作为 API 设计审查的核对清单。

第 1 章

API 基础知识

本章首先介绍远程 API 的背景知识，然后解释 API 的极端重要性，并分析 API 设计过程中遇到的主要挑战(包括耦合和粒度问题)，最后介绍 API 领域模型，以帮助开发人员熟悉全书使用的术语和概念。

1.1 从本地接口到远程 API

如今，几乎不存在完全断开连接的应用程序，哪怕是独立应用程序往往也会提供某种外部接口。以通常基于文本的文件为例，文件导入/导出就是一种简单的接口。从某种意义上讲，也可以将使用了操作系统剪贴板的复制和粘贴功能看作接口。在应用程序内部，每种软件组件同样会提供接口[Szyperski 2002]。这些接口描述了组件对外公开的操作、属性和事件，但不会透露组件内部采用的数据结构或实现逻辑。要使用组件，开发人员必须学习并理解组件提供的接口。某个选定的组件可能使用其他组件提供的服务，倘若如此，则说明该组件对一个或多个所需的接口存在出站依赖。

有些接口比其他接口更加公开。例如，中间件平台和框架通常会提供 API。具有平台特征的 API 最早见于操作系统，目的是将用户应用程序软件与操作系统的实现区分开来，可移植操作系统接口(POSIX)和 Win32 API 就属于这种平台 API。平台 API 既要具备足够的通用性和表现力，以便开发人员构建不同类型的应用程序；又要在多个操作系统版本中保持稳定，以便旧版应用程序在操作系统升级后仍然能够继续运行。将操作系统组件的内部接口纳入公开发布的 API 对文档质量提出了很高的要求，且对接口随着时间推移而可能出现的变化类型也施加了严格的限制。

API 既可以跨越操作系统进程的边界，也可以暴露在网络中，以支持在不同物理或虚拟硬件结点上运行的应用程序之间相互通信。长期以来，企业一直借助这类远程 API 来实现应用程序的集成[Hohpe 2003]。如今，远程 API 一般位于移动应用程序或 Web 应用程序的前端与这些应用程序的服务器后端之间的边界，并且往往将其部署在云数据中心。

通常情况下，应用程序前端使用由应用程序后端管理的共享数据。因此，同一个 API 既可以支持不同类型的 API 客户端(例如移动应用程序和富桌面客户端)，也可以支持并发运行的多个客户端实例。有些 API 甚至向由其他组织开发和操作的外部客户端开放系统。这种开放性不

仅会引发安全方面的担忧，例如涉及有权访问 API 的应用程序客户端或最终用户；而且会带来战略层面的影响，例如各方必须就数据所有权和服务级别协商一致、达成共识。

假设本地组件接口和连接应用程序的远程 API 都存在*共享知识*(shared knowledge)，各方需要利用这些知识来开发具备互操作性的软件。就像我们可以毫不费力地将电源线插入匹配的电源插座一样，使用 API 的目的是实现兼容系统的集成。

共享知识包括以下内容：

- 对外公开的操作以及它们提供的计算服务或数据操纵服务；
- 调用操作时所交换数据的表示和含义；
- 可观察的属性(例如组件状态和有效状态转换的相关信息)；
- 事件通知和错误条件(例如组件故障)的处理。

远程 API 还会定义以下内容：

- 在网络之间传输消息所用的通信协议；
- 网络端点，包括位置和其他访问信息(例如地址和安全凭据)；
- 与分发有关的故障处理策略，包括由底层通信基础设施引起的故障(例如超时、传输错误、网络和服务器瘫痪)。

API 契约体现出各方参与者的期望。API 遵循基本的信息隐藏原则，其实现秘而不宣，只披露最低限度的信息(例如怎样访问 API 并使用 API 提供的服务)。举例来说，在设计与 GitHub 集成的软件工程工具时，开发人员可以通过 GitHub API 了解如何创建和检索问题、问题包括哪些属性(或字段)等内容，但无法得知为公共 API 服务的问题管理应用程序采用哪种编程语言、数据库技术、组件结构或数据库模式。

需要注意的是，并非所有系统和服务从一开始就使用 API，而且 API 可能随着时间的推移而消失。举例来说，社交平台 X(原 Twitter)向第三方客户端开发人员开放其 Web API 以提高知名度，从而很快催生出一个完整的客户端生态系统，吸引了众多用户。为了从用户生成的内容中获利，X 后来选择关闭 API，并把部分客户端应用程序收归旗下，继续在内部进行维护。由此可见，随着时间的推移，API *演进*必须受到管理。

1.1.1 分布式系统和远程 API 概述

远程 API 包括多种不同的形式。为支持系统部件之间进行通信，过去半个世纪以来涌现出大量将应用程序分解为*分布式系统*的概念和技术：

- 传输和网络协议 TCP/IP 及其*套接字* API 诞生于 20 世纪 70 年代，是互联网采用的基本通信协议。文件传输协议同样出现在这一时期，例如 FTP 和基本文件输入/输出(如访问共享驱动器或挂载的网络文件系统)，过去和如今的编程语言广泛采用这些协议。
- 二十世纪八九十年代，分布式计算环境(Distributed Computing Environment，DCE)等*远程过程调用*(Remote Procedure Call，RPC)和通用对象请求代理体系结构(Common Object Request Broker Architecture，CORBA)、Java 远程方法调用(Java Remote Method Invocation，Java RMI)等面向对象的请求代理引入了抽象层和便利层。近年来，gRPC 等更为新颖的 RPC 框架开始受到青睐。

- IBM MQ(前身为 IBM MQSeries)、Apache ActiveMQ 等基于队列、面向消息的应用程序集成在时间维度中有助于解耦通信参与者。它们与远程过程调用的历史一样悠久，自 2000 年以来出现了新的实现和风格。例如，各大云服务提供商目前都提供自己的消息传递服务，云租户也可以将其他消息中间件部署到云基础设施(实践中往往采用 RabbitMQ)。
- 得益于万维网的普及，HTTP 等面向超媒体的协议在过去 20 年里逐渐兴起。只有遵循描述性状态迁移(Representational State Transfer，REST)风格规定的所有架构约束，才属于 RESTful 架构。虽然并非所有 HTTP API 都能做到这一点，但 HTTP 目前似乎在公共应用程序集成领域居于主导地位。
- 基于连续数据流构建的数据处理管道(例如采用 Apache Kafka 构建的管道)源于经典的 UNIX 管道-过滤器架构，它在 Web 流量和在线购物行为分析等数据分析场景中颇受青睐。

TCP/IP、HTTP 和基于异步队列的消息传递如今依然占有重要地位，而分布式对象再次成为明日黄花，只有某些遗留系统还在使用它们。通过协议或共享驱动器进行文件传输仍然相当普遍。现有的方案能否继续使用尚待观察，新的方案很可能登台亮相。

所有远程和集成技术的目标都是连接分布式应用程序(或其部件)，使它们能够触发远程处理或检索和操作远程数据。如果没有 API 和 API 描述，那么这些应用程序要么不知道如何与远程伙伴系统建立连接并交换数据，要么不清楚如何接收并处理来自这些系统的回复。

1.1.2 远程 API：通过集成协议访问服务

1.3 节将介绍全书使用的 API 术语，本节先把前文的讨论概括为一个定义。

“API”是应用程序接口一词的缩写，因为这个术语源于(通过本地 API 进行的)程序内部分解。API 具有双重性，它们在提供连接服务的同时也提供分离服务。因此，远程 API 也可以指通过使用应用程序集成的通信协议来访问数据、软件服务等服务器端资源(“访问”一词的首字母为“a”，“协议”一词的首字母为“p”，“集成”一词的首字母为“i”，合起来是“API”)。

到目前为止，我们讨论的远程消息传递概念如图 1-1 所示。

远程 API 为集成的应用程序模块提供虚拟和抽象的连接。每个远程 API 至少由另外三个 API 实现：客户端和提供者各有一个本地接口，通信栈的下一层还有一个远程接口。两个本地接口由操作系统、中间件或编程语言库和软件开发工具包(Software Development Kit，SDK)提供，并由 API 客户端和 API 提供者端的应用程序使用。本地接口向需要集成的应用程序组件、子系统或整个应用程序公开网络/传输协议服务(例如基于 TCP/IP 套接字的 HTTP)。

为了实现可互操作通信这一共同目标，各个通信参与者必须就 API 契约达成共识。定义 API 契约时，既要考虑协议和支持协议的端点，也要考虑对外公开的数据。请求和响应消息表示必须以某种方式进行结构化[1]。即使是关于文件导入/导出或文件传输的消息也需要经过仔细设计，因为在这种情况下，文件中包含这些消息。基于剪贴板的集成具有类似的属性。API 契约致力于描述有关消息语法、结构和语义的共享知识，在连接双方的同时也把双方分离开来。

1 响应消息是否存在取决于所用的消息交换模式(本章稍后将讨论我们构建的 API 领域模型)。

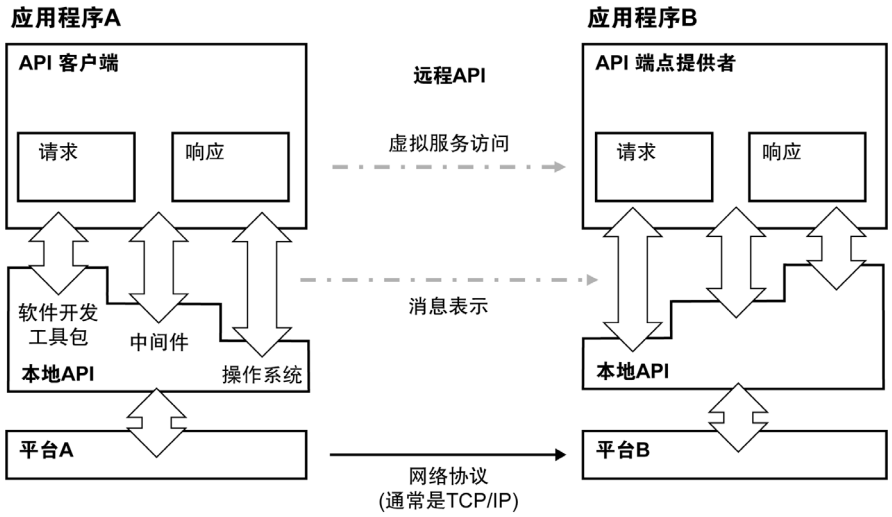

图 1-1 远程 API：基于消息的集成和概念

我们把远程 API 定义如下：

远程 API 是一系列有据可查的网络端点，支持内部和外部应用程序组件相互提供服务。这些服务有助于实现领域特定目标，例如实现业务流程的完全自动化或部分自动化。客户端可以激活提供者端的处理逻辑，或支持数据交换和事件通知。

上述定义确立了本书的设计空间。请注意：本书致力于讨论远程 API，因此从现在开始，除非明确说明，否则“API”一词均指远程 API 而不是本地 API。

API 设计极具挑战性，许多影响决策的因素(又称驱动力或质量属性)在设计中起到至关重要的作用。相关讨论参见 1.2 节。

1.1.3 API 的重要性

我们接下来讨论部分业务领域和技术领域，从中可以找到许多 API 的身影。

1. 现实生活中的各种 API

如今，API 涉及广告、银行、云计算、网站目录、娱乐、金融、政府机构、医疗健康、保险、求职、物流、消息传递、新闻、开放数据、支付、二维码、房地产、社交媒体、旅游、短网址、可视化、天气预报、邮政编码等诸多领域。网络上存在数以千计的 API，通过这些 API 可以访问以服务形式交付的可复用组件。下面给出了上述领域的一些示例：

- 创建并管理广告活动。获取关键字和广告的状态。生成关键字估计。生成广告活动效果的相关报告。

- 在核实客户身份后开设银行账户。
- 在虚拟机上管理和部署应用程序，并跟踪资源消耗情况。
- 确定某个人的身份。查找其电话号码、电子邮件地址、所在位置和人口统计数据。
- 收集、发现并分享自己青睐的报价。
- 检索外汇、股票和商品交易的相关信息。获取市场的实时价格数据。
- 访问空气质量监测、停车设施、电力和水资源消耗、每日新增病例数、紧急服务请求等公共数据集。
- 在保护用户隐私和控制权的前提下，实现健康和健身数据的共享。
- 返回旅游保险、房屋保险和汽车保险的报价，为客户提供即时保险业务。
- 利用基本职位搜索、检索特定职位数据和申请职位的方法，将职位数据库集成到用户软件或网站中。
- 汇总多家货运公司的信息，提供货运等级、运输成本报价以及货物预订和跟踪功能，并能够安排提货和送货。
- 在全球范围内发送短信。
- 充分利用新闻、视频、图片、多媒体文章等已发布的内容。
- 访问在线支付解决方案，包括发票管理、交易处理和账户管理。
- 访问房屋估价、房产详情(包括历史销售价格、城市和社区市场统计数据)、房贷利率、月供估算等服务。
- 研究信息通过社交媒体的传播方式。这些信息可以是假新闻、骗局、谣言、阴谋论、讽刺作品乃至准确的报道。
- 根据类别、国家、地区或用户所在位置获取网络摄像头。获取每个网络摄像头捕捉的一系列图像。添加用户自己的网络摄像头。
- 采用数字天气标记语言(Digital Weather Markup Language，DWML)，以编程方式获取当前观测、预报、天气监视/警告和热带气旋警报。

在上述所有示例中，API 契约定义了 API 的调用位置和调用方式、需要发送的数据以及所接收响应的格式。实际上，部分领域和服务与 API 呈现出共生共荣、不可分割的关系。下面将深入分析其中一些领域和服务。

2. 移动应用程序和云原生应用程序使用并提供大量 API

自智能手机(如 iPhone)和公有云(如亚马逊云服务)于 2006 年前后出现以来，软件的开发方式和向最终用户交付的方式发生了翻天覆地的变化。JavaScript 在 Web 浏览器中的普及和 XMLHttpRequest 规范[1]的引入，共同推动了软件向单页应用程序、智能手机应用程序等富客户端的范式转移。

如今，为移动应用程序或其他最终用户前端提供服务的应用程序后端通常部署到公有云或私有云中。目前，采用“一切皆服务”(XaaS)模式的云服务不计其数，它们具备独立部署、租

1 称为异步 JavaScript 和 XML 技术(AJAX)，详见：https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX.。请注意，JSON 如今比 XML 更受青睐，而且 Fetch API 比 XMLHttpRequest 对象更强大、更灵活。

用、扩展和计费的能力。这种大规模的模块化和(可能的)区域分布需要使用 API，包括云内部的 API 和云租户使用的 API。截至 2021 年，亚马逊云服务提供了 200 多项服务，紧随其后的是微软云和谷歌云[1]。

当云服务提供商向云租户提供 API 时，部署到云端的应用程序开始依赖这些云 API，但自身也会公开并使用应用程序级 API。这类 API 既可以连接云外部应用程序前端与云托管应用程序后端，也可以实现应用程序后端的组件化，从而使这些后端能够充分利用按使用付费、弹性伸缩等云属性，成为名副其实的*云原生应用程序*(Cloud Native Application，CNA)。典型的云原生应用程序架构如图 1-2 所示。

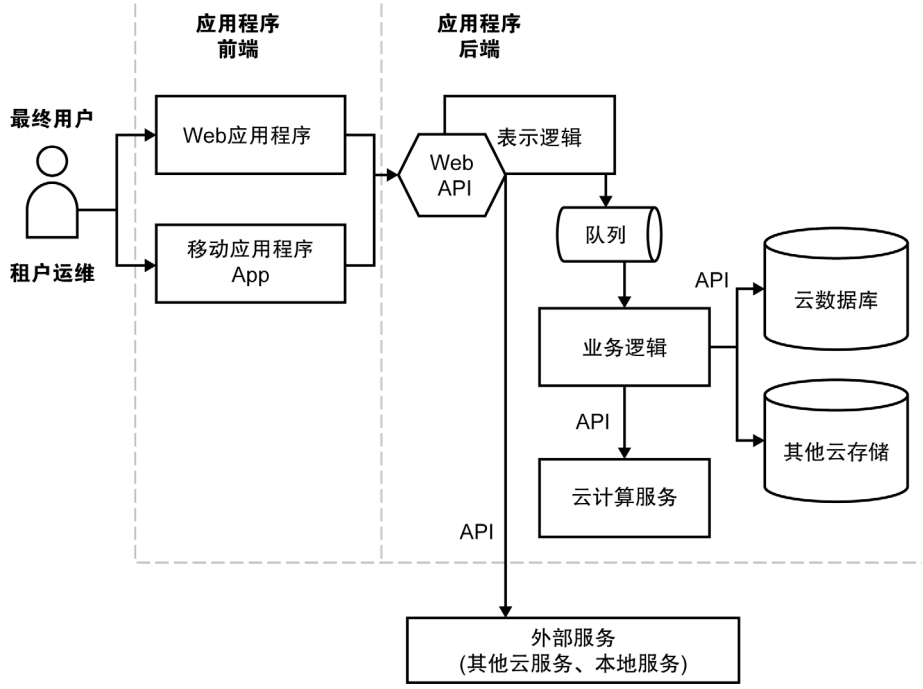

图 1-2 云原生应用程序(Cloud Native Application，CNA)架构

从架构的角度来看，隔离状态、分布、弹性、自动化和松耦合(Isolated State，Distribution，Elasticity，Automation，Loose Coupling，IDEAL)是云原生应用程序所期望具备的理想属性[Fehling 2014]。技术文献采用一系列原则来描述云应用程序的特征，IDEAL 是其中之一。作为 IDEAL 的超集，以下七项特征概括了云原生应用程序能够成功运行并充分利用云计算优势所需具备的要素[Zimmermann 2021a]：

1. 适用性
2. 规模调整(rightsizing)和模块化
3. 主权和容忍
4. 弹性和保护
5. 可控性和适应性

1 很难确定具体的数字，取决于如何区分各项服务。

6. 工作负荷感知和资源高效利用
7. 敏捷性和工具辅助

第 2 项特征(规模调整和模块化)要求必须使用 API。云应用程序管理(第 5 项特征)同样需要用到 API，DevOps 工具链(第 7 项特征)也能从 API 中受益。

例如，Kubernetes 已成为在本地和云端运行应用程序并协调底层计算资源的热门之选，这款集群管理软件能够解决重复部署大量单体应用程序和服务的问题。所有应用程序服务通过 API 相互通信，并与各自的客户端交换数据。Kubernetes 平台本身也会公开管理 API [Kubernetes 2022]和命令行接口。Kubernetes 提出的操作者概念通过 API 和其上的 SDK 对外公开，从而进一步提高了可扩展性。Kubernetes 甚至还具备管理应用程序 API 的能力。

又如，软件即服务(Software-as-a-Service，SaaS)提供商往往不仅提供可定制的多租户最终用户应用程序，而且通过 HTTP 向第三方开放其应用程序功能。举例来说，Salesforce 提供通过 HTTP API 进行数据访问和集成的功能。截至本书完稿时，Salesforce 有 28 个可用的 API，涵盖营销、B2C 商业活动、客户数据管理等多个领域。

3. 微服务之间通过 API 相互通信

近年来，微服务一词几乎随处可见。自从 2014 年 4 月 James Lewis 和 Martin Fowler 在网上发表文章[Lewis 2014]以来，业界对这种相当先进的系统分解方法已进行过深入讨论。随着面向服务的体系结构(Service Oriented Architecture，SOA)进入持续软件交付和云计算的时代，微服务应运而生。抛开炒作的成分不谈，微服务可以定位为 SOA 的一种子风格或实现方法，既强调服务的独立可部署性、可伸缩性和可变性，也强调分散性、自主决策和容器编排[Pautasso 2017a]。

每种微服务仅承担一项职责，该职责应该代表领域特定的业务功能。微服务通常部署在轻量级虚拟化容器(如 Kubernetes 和 Docker)中，封装自身状态，并通过远程 API(一般采用 HTTP，但也可以采用其他协议)相互通信。这些服务 API 有助于确保彼此之间具有松耦合，从而能够在不影响整体架构的情况下进行演进或替换[Zimmermann 2017]。

微服务的范围有限且侧重于实现单一业务功能，因此有利于软件复用。微服务支持能够进行持续交付的敏捷软件开发实践。例如，一支团队往往只负责一种微服务，可以独立进行开发、部署和操作。如前所述，微服务也是实现 IDEAL CNA 的理想之选。独立部署微服务时，可以通过容器虚拟化和弹性负载均衡实现横向的按需可伸缩性。通过保持现有的服务 API 不变，微服务能够实现单体应用程序的增量迁移，从而降低软件现代化改造带来的风险。

微服务同样会带来新的挑战。微服务具有分布式和松耦合的特性，进而对 API 设计和系统管理提出了很高要求。考虑到分布式架构引入的通信开销和糟糕的 API 设计，微服务架构的性能可能受到影响。举例来说，把单体、有状态的应用程序解耦为独立、具有自主性的微服务时，就会带来数据一致性和状态管理方面的挑战[Furda 2018]。为此，必须避免单点故障或级联故障扩散造成的影响。采用传统的备份和灾难恢复策略时，无法同时确保整个微服务架构具有自主性和一致性[Pardon 2018]。如果希望扩展架构以纳入大量微服务，则需要采用严格的方法来管理、监控和调试生命周期。

某些挑战可以通过构造足够完备的基础设施加以克服。例如，负载均衡器会引入(托管)冗余；又如，就算下游微服务实例出现故障，使用断路器[Nygard 2018a]也能降低上游微服务实例崩溃的风险(这种风险最终会导致整个系统瘫痪)。随着时间的推移，服务 API 仍然需要进行适当的规模调整和演进。

本书致力于从端点粒度和操作/数据耦合的角度来分析 API 层面的服务规模调整，微服务基础设施不是讨论重点。但如果 API 服务的规模足够大，那么基础设施设计的难度就会降低，因此本书也会间接讨论基础设施设计。

4. API 属于产品，可能形成生态系统

软件产品是可以购买(或授权使用)的实体资产或虚拟资产。对于所购买的产品，付费客户对其使用寿命、质量和可用性有一定期待。API 既可以作为独立的产品存在，也可以随其他软件产品一起提供(例如，用于加载产品的主数据，或进行配置和定制以满足特定用户群体的需要)。即使没有自己的业务模型或没有直接为业务战略做出贡献，API 仍然应该被“视为产品”[Thoughtworks 2017]。API 应该有专门的业务负责人、治理结构、支持系统和路线图。

举例来说，由深度学习算法提供支持的数据湖需要使用数据，而这些数据必须从某处获取。如果把数据比作数字时代的石油，那么消息信道和事件流就是输油管道，中间件/工具/应用程序就是炼油厂，API 则相当于输油管道、生产者与消费者之间的阀门。数据湖既可以是对外公开 API 的市场化产品，也可以是管理方式类似于市场化产品的企业内部资产。

软件生态系统是指“一系列参与者在共同的技术平台上进行互动，从而孕育出大量软件服务或解决方案”[Manikas 2013]。生态系统由自然增长、相互独立但又彼此相关的部分和参与者组成，要么完全分散，要么以做市商(market maker)为中心。Cloud Foundry 等开源市场属于软件生态系统，苹果公司的 App Store 则属于转售软件生态系统。两种生态系统的成功与 API 息息相关：API 既支持应用程序加入或离开生态系统，也支持成员之间进行通信和协作，还支持分析生态系统的健康状况[Evans 2016]。

我们以旅游管理生态系统为例进行讨论。这种生态系统可能提供两种 API，一种用于引导(加载租客、出行平台等生态系统成员)，另一种用于支持行程规划、报告和分析应用程序的开发(提供目的地排名、住宿评价等功能)。生态系统的各个组成部分通过 API 相互通信，在预订火车票、飞机票或酒店房间时也通过 API 与市场/生态系统制造者交换数据。

生态系统能否取得成功，取决于 API 设计和演进是否正确。软件生态系统越复杂、越动态，其 API 设计就越具有挑战性。多条消息在参与者之间传输，它们的关系由 API 契约描述，这些消息构成了持续时间较长的对话。生态系统的成员必须就格式、协议、对话模式等问题达成一致。

5. 小结

本节讨论的所有示例、场景和领域都与远程 API 及其契约息息相关，类似的示例、场景和领域还有很多。简而言之，API 是近年来几乎所有主要发展趋势的使能技术，既包括前文提到的移动端/Web 端和云服务，也涵盖人工智能和机器学习、物联网、智慧城市、智能电网等领域。就连云端的量子计算也离不开 API，谷歌量子人工智能开发的量子引擎 API(Quantum Engine API)便是一例。

1.2　API 设计中的决策驱动因素

在图 1-1 所示的架构中，API 起到相当独特的“连接-分离”作用，从而催生出大量具有挑战性、有时甚至相互抵触的设计问题。例如，在对外公开数据(以便客户端能够善加利用)与隐藏实现细节(以便能够随着 API 演进而进行调整)之间，务必找到平衡。一方面，API 对外公开的数据表示必须满足客户端的信息和处理需求；另一方面，必须以易于理解、可维护的方式对 API 进行设计和记录。向后兼容性和互操作性在 API 设计中十分重要。

本节将介绍特别重要的驱动因素，我们会反复讨论这些贯穿全书的因素。下面从 API 的成功要素入手讨论。

1.2.1　API 的成功要素

从某种意义上讲，成功是一项见仁见智的衡量标准。衡量 API 成功与否的一种观点认为：

只有多年前设计和发布、每天都能以最小延迟和零停机时间为数十亿付费客户端提供服务的那些 API，才称得上是成功的 API。

另一种相反的观点则认为：

对于完全根据 API 文档构建的外部客户端来说，如果新发布的 API 能够成功接收并响应该客户端的第一个请求，而且不需要原始实现团队的帮助或介入，那么这样的 API 就可以被视为已经取得了成功。

如果 API 用于商业环境，则可以根据商业价值来评估 API 成功与否，重点是评估服务运营成本相对于每个 API 客户端直接或间接产生的收入是否具有经济可持续性。API 可能采用不同的商业模式，既包括由广告商资助、可以免费访问的 API，这些广告商对通过 API 构建的应用程序并由其用户提供的数据感兴趣(数据可能由用户自愿提供，也可能由用户非自愿提供)；也包括基于订阅的 API 和按使用付费的 API，这些 API 根据不同的资费计划提供服务。举例来说，谷歌地图曾是独立的 Web 应用程序，而在用户开始通过逆向工程将地图可视化嵌入自己的网站后，谷歌公司才决定开放地图 API。由此可见，最初封闭的架构在用户需求的推动下逐步开放，而一开始可以免费访问的 API 后来发展成为有利可图的按使用付费服务。作为谷歌地图的开源替代方案，开放街道街图(OpenStreetMap)同样提供了一些 API。

第二个成功要素是可见性。如果潜在客户不知道 API 的存在，那么再好的 API 设计也算不上成功。举例来说，既可以通过在公司产品和产品文档中加入 API 链接，也可以通过在程序员圈子里进行宣传来发现公共 API。使用 ProgrammableWeb 和 APIs.guru 之类的 API 目录同样可行。无论采用哪种方式，为宣传 API 而进行的投资最终应该都会得到回报。

API 的实际发布时间既可以根据部署新功能或修复错误所需的时间来衡量，也可以根据为 API 开发功能齐全的客户端所需的时间来衡量。首次调用时间能够很好地反映出 API 文档的质

量和客户端开发人员的引导体验。为缩短首次调用时间，API 的学习曲线也应该尽量平缓。首次创建 n 级工单的时间同样可以作为衡量指标——但愿 API 客户端开发人员要过很久才会遇到需要启动一级、二级或三级支持来解决的错误。

生命周期是衡量 API 成功与否的另一项指标。即使 API 最初的设计者已经离世，API 可能依然存在。成功的 API 通过适应不断变化的客户需求来不断吸引客户，因此往往具有旺盛的生命力。然而，客户(包括那些没有其他选择的客户)仍然青睐功能保持不变和长期稳定的 API。例如，标准化、发展缓慢的电子政务 API 可以满足客户的合规性要求。

总之，API 既要在短时间内实现系统及其部件的快速集成，又要长期支持这些系统的自主性和独立演进。快速集成旨在降低整合两个系统所花费的成本，独立演进则是为了防止系统向高度纠缠和耦合的方向发展，以免无法分离(或替换)。这两个目标在某种程度上是相互矛盾的，相关讨论将贯穿全书。

1.2.2 API 设计有何不同

API 设计会影响所有软件设计和架构。从 1.1 节的讨论可知，独立开发和运营的客户端与服务提供者相互做出的假设是 API 设计的基础。API 能否取得成功，取决于相关各方能否达成一致并长期信守承诺。这些假设和协议涉及以下问题和权衡取舍。

- **一个通用端点与多个特定/专用端点：**应该只设计一种接口供所有客户端使用，还是分别设计 API 供部分或全部客户端使用？哪种方案可以提高 API 的易用性？例如，通用 API 是否具有更好的复用性，但在特定情况下也更难应用？
- **细粒度端点和操作范围与粗粒度端点和操作范围：**如何平衡 API 功能的广度和深度？API 是否应该匹配、聚合或拆分底层系统功能？
- **处理大量数据的少数操作与处理少量数据的大量琐碎操作：**请求和响应消息包含的数据内容应该尽量详尽，还是重点突出？哪种方案的可理解性、性能、可伸缩性和可演进性更好并能减少带宽消耗？
- **数据当前性与数据正确性：**共享陈旧的数据是否胜过完全不共享数据？当可靠的数据一致性(API 提供者内部)与快速响应时间(由 API 客户端感知)之间发生意料之中的冲突时，应该如何处理？应该通过轮询机制报告状态变化，还是通过事件通知或流式传输推送状态变化？命令与查询是否应该分开？
- **稳定的契约与频繁变化的契约：**如何在不牺牲可伸缩性的前提下保持 API 的兼容性？在设计功能丰富、长时间使用的 API 时，如何进行修改以确保不会破坏向后兼容性？

上述问题、方案和标准是 API 设计面临的挑战，开发人员需要根据不同的需求背景做出不同的选择。我们采用的模式会给出可能的答案及其后果。

1.2.3 API 设计难在哪里

最终用户界面设计带来的用户体验要么很愉快，要么很糟心。与之类似，API 设计会影响

开发者体验——受影响的对象既包括学习如何使用 API 来构建分布式应用程序的客户端开发人员，也包括编写提供者 API 实现的开发人员。API 首次发布并在生产环境中投入运行后，其设计会对最终集成系统的性能、可伸缩性、可靠性、安全性和可管理性产生重大影响。如果利益相关方的关注点发生抵触，则必须加以平衡。这种情况下，开发者体验会延伸到操作者体验和维护者体验中。

对 API 提供者和客户端而言，双方的目标和要求既可能有所重叠，也可能相互矛盾，不一定总能实现双赢。API 设计之所以充满挑战，是因为受到了某些非技术性因素的影响。

- **客户端多样性**：API 客户端的需求各不相同，而且会不断变化。API 提供者必须决定是只开发一种统一的 API 来提供足够适用的折中方案，还是根据不同客户的具体需求分别开发 API。
- **市场动态**：API 提供者试图赶上竞争对手的创新脚步，从而可能带来更多变化并催生出不兼容的演进策略，这会超出 API 客户端能够接受或愿意接受的范围。此外，客户端通过寻找标准化的 API 作为保持独立于特定提供者的一种手段，而有些提供者可能通过提供诱人的扩展功能来锁定客户端。假如谷歌地图和开放街道街图采用同一套 API，情况是否会更好？对于这个问题，客户端和提供者端的开发人员或许存在不同的看法。
- **分布谬误**：有时需要通过使用不可靠的网络来访问远程 API。常言道，凡是可能出错的事必定会出错。举例来说，即使某项服务正常运行，客户端也可能暂时无法访问该服务，从而对确保 API 访问的高服务质量(例如，确保 API 可用性和响应时间)构成挑战。
- **控制错觉**：客户端可以使用 API 对外公开的所有数据，使用这些数据的方式有时出乎意料。发布 API 意味着交出一部分控制权，从而使系统面临来自外部(甚至未知)客户端的压力。开发人员必须谨慎决定外界可以通过 API 访问哪些内部系统部件和数据源，因为控制权一旦交出，就很难甚至完全无法收回。
- **演进陷阱**：虽然微服务的初衷是支持频繁进行更改(例如结合 DevOps 实践实现持续交付)，但是设计高质量 API 的机会只有一次。一旦 API 发布并取得成功，就会有越来越多的客户端依靠 API，导致更改和完善它的成本变得越来越高，删减功能时也无法做到完全不影响客户端。尽管如此，API 还是会随着时间的推移而演进。调整 API 时需要采取适当的版本控制实践，以缓和设计稳定性与灵活性之间的紧张关系。有时候，提供者拥有主导演进策略和节奏的市场影响力；有时候，客户端社区在 API 使用关系中更为强势。
- **设计失配**：后端系统在功能范围和质量方面的表现以及端点和数据定义方面的结构，可能与客户端的预期有所不同。为克服这些差异，必须采用某种形式的适配器以转换失配部件。某些情况下，后端系统必须进行重构或重新设计，以满足外部客户端的需求。
- **技术变革和技术漂移**：用户界面技术在不断进步。例如，从键盘和鼠标发展到触摸屏和语音识别，再发展到虚拟现实和增强现实使用的运动传感器(以及其他更先进的技术)。这些技术进步促使开发人员重新思考用户与应用程序的交互方式。API 技术也在不断变化——无论是新的数据表示格式、改进的通信协议还是中间件和工具环境的变化，都需要进行持续性投资，以保证集成逻辑和通信基础设施能够与时俱进[1]。

1 现在还有多少 XML 开发人员和工具？

总之，API 设计可以决定软件项目、产品和生态系统的成败。API 并非单纯地实现工件，而是集成资产。API 具有连接器和分离器的双重属性，而且往往会存在很长时间，因此设计时绝不能草率行事。虽然各种技术层出不穷，但集成设计人员面临的许多基本设计问题及其解决方案保持不变。

下一节将讨论架构方面的重要需求，尽管这些需求会发生一定变化，但是就整体而言，基本的需求在很长一段时间内会保持相关性。

1.2.4 架构方面的重要需求

可以从开发、操作、管理等方面来界定 API 要实现的质量目标。本节进行概述，后续章节将详细介绍这些目标。

- **可理解性**：进行 API 设计时，请求和响应消息中表示元素的结构是一个重要的开发问题。为保证可理解性并避免产生不必要的复杂性，通常建议严格遵照领域模型来编写 API 实现代码并设计 API。请注意，“遵照”并不等同于完全对外公开或完全复制，尽可能隐藏信息也很有必要。
- **信息共享与信息隐藏**：API 指定了客户端期望的内容，同时抽象出提供者端如何满足这些期望。开发人员需要付出时间和精力将规范与软件组件的实现分开。设计 API 时，向接口公开现有的实现细节也许是一种省时省力的解决方案，但这样处理存在严重弊端，那就是今后很难在不影响客户端的情况下修改 API 实现。
- **耦合量**：松耦合是分布式系统及其组件在结构设计方面的内部质量目标，可以认为这种架构原则介于需求(问题)与设计元素(解决方案)之间。通信各方的松耦合包括不同的维度，一是处理命名和寻址约定的引用自主性，二是隐藏技术选择的平台自主性，三是支持同步通信或异步通信的时间自主性，四是涉及数据契约设计的格式自主性[Fehling 2014]。根据定义，API 调用将客户端和提供者端耦合在一起。然而，耦合性越低，客户端和提供者端就越容易独立演进，原因之一在于提供者和使用者必须共享的知识会影响可更改性。举例来说，针对对外公开的数据结构进行规模调整可以带来一定程度的格式自主性。此外，除非确有必要，否则来自同一提供者端的两个 API 不应该进行耦合(例如通过隐藏的依赖关系)。
- **可修改性**：可修改性是可支持性和可维护性的重要子问题。就 API 设计和演进而言，可修改性会纳入向后兼容性，以促进并行开发和部署灵活性。
- **性能和可伸缩性**：从 API 客户端的角度观察，延迟是重要的操作问题，它受到多方面因素的影响，既包括带宽、低级延迟等网络行为，也包括有效载荷的封送(marshalling)和解封送(unmarshalling)等端点处理工作。API 提供者端主要关心吞吐量和可伸缩性。这两项指标意味着即使由于更多客户端使用 API 或现有客户端令负载加重而导致提供者端的负载增加，响应时间也不会延长。
- **数据简约性**：在对性能和安全性要求极高的分布式系统中，数据简约性是一项重要的通用设计原则。但是当通过指定请求和响应消息来迭代并渐进式地定义 API 时，这项原则不一定适用，原因在于添加内容(例如信息项或值对象的属性)往往比删除内容更容

易[1]。因此，在 API 设计和演进的过程中，整体认知负荷和处理工作量会不断增加。在 API 新增某些内容后，往往很难判断能否安全地删除 API，这是因为许多(甚至未知的)客户端可能会依赖 API。正因为如此，API 对外公开的契约也许包括大量相当复杂的数据元素(例如客户或产品主数据的属性)，而且这种复杂性很可能随着软件的演进而增加。变化性管理和“选项控制”必不可少。

- **安全性和隐私性：**对 API 设计来说，安全性和隐私性往往是重要的考虑因素，不仅包括访问控制，而且包括敏感信息的保密性和完整性。例如，API 可能需要具有安全性和隐私性，以免暴露后端服务中包含的机密元素。为支持可观察性和可审计性，建议监控 API 流量和运行时行为。

为满足这些有时相互矛盾(且不断变化)的需求，需要在某些已知选项或新选项之间选择准备采用的架构，而需求是影响架构决策的因素(或标准)之一。权衡利弊的情况不仅存在，而且必须进行处理。我们提出的模式选择将需求作为设计要素来考量，并讨论权衡解决方案。

1.2.5 开发者体验

近年来，将开发者体验与用户体验进行类比和比喻变得颇为流行。Albert Cavalcante 在“What Is DX?”[Cavalcante 2019]一文中写道，开发者体验是用户体验与软件设计原则相结合的产物，愉悦的开发者体验由功能、稳定性、易用性、清晰性这四大支柱构成，即

$$\text{开发者体验} = \text{功能} + \text{稳定性} + \text{易用性} + \text{清晰性}$$

开发者体验涉及开发工作的方方面面，包括工具、库和框架、文档等等。功能支柱指出，某些软件对外公开的处理/数据管理功能之所以具有高优先级，仅仅是因为这些功能激发了客户端开发人员对 API 的兴趣。API 提供的功能应该满足客户端的目标。稳定性指满足期望和各方一致同意的运行时质量目标，如性能、可靠性和可用性。对开发人员而言，可以通过提供文档(教程、示例、参考资料)、社区知识论坛、工具特性(以及其他方式)实现(软件)易用性。清晰性既包括简单性，也涉及可观察性。举例来说，点击工具中的按钮、调用命令行界面(或 SDK 提供的命令)、生成代码等操作所产生的后果应该始终清晰明确。一旦出现问题，客户端开发人员不仅希望了解原因(例如，输入无效或提供者端存在问题)，而且希望找到处理方案(例如，稍后再次尝试调用或修改输入)。

这里有必要提醒开发人员注意：设计 API 的目的不是供自己使用，而是供客户端及其软件使用。话虽如此，设备间通信与人机交互存在本质区别，原因很简单：人类与计算机的思考能力和行为方式有所不同。程序可能(在某种程度上)具备思考能力，但没有喜怒哀乐，也不会意识到自身的存在和所处的环境[2]。因此，并非所有有关用户体验的建议都直接适用于开发者体验。

开发者体验(理所当然)受到广泛关注，它还包括维护者体验以及顾问/教育者/学习者体验，但是我们对运维者体验是否有足够的了解和认识呢？

---

1 想一想大型企业的业务流程、需要填写的相应表单和审批要求：许多活动和数据字段往往是出于好意而添加的，但它们很难取代现有的内容。

2 我们也许可以在图像识别等某些受限的领域中训练程序，但不能指望它们能够建立起价值体系，并像人类那样表现出道德和伦理。

综上所述，至少可以从短期的积极反馈和长期的使用情况这两个方面来衡量 API 成功与否：

第一印象很重要。成功完成首次 API 调用并对响应进行处理的过程越简单、越明确，使用 API 的客户端开发人员就越多，开发者体验(功能、稳定性、易用性和清晰性)也越好。开发人员初次使用 API 时留下的良好印象能否持续下去，成为 API“永葆青春”的基础，具体取决于性能、可靠性、可管理性等运行时质量目标。

下一节(也是本章最后一节)将介绍 API 领域模型，其中涉及本书使用的各种术语。

1.3 远程 API 的领域模型

本书及其模式语言采用一套基本的抽象和概念，为 API 设计和开发构建一个领域模型[Zimmermann 2021b]。我们利用这个领域模型详细介绍我们采用的模式，但并不追求为现有的所有通信概念和集成架构绘制出完整而一致的图景。不过，我们会解释领域模型要素与 HTTP 和其他远程技术中的概念有哪些关系。

1.3.1 通信参与者

就抽象层面而言，有两种通信参与者(简称“参与者”)通过 API 相互通信，它们是 API 提供者和 API 客户端。API 客户端可以使用的 API **端点**没有数量限制。API 提供者对外公开 API **契约**，API 客户端则使用契约。API 契约负责约束通信，包括提供具有指定功能的可用端点的相关信息。这些基本概念和它们之间的关系如图 1-3 所示。

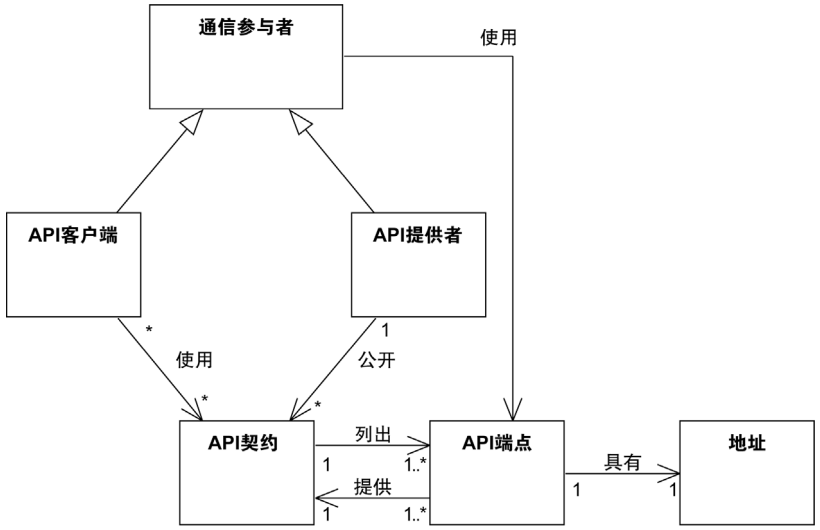

图 1-3 API 设计和演进的领域模型：通信参与者、API 契约、API 端点

请注意，图 1-3 没有显示整个 API，因为 API 是由一系列端点及其提供的契约所构成的。API 端点代表通信信道的提供者一侧，API 至少包括一个这样的端点。每个 API 端点都有唯一的地址(例如统一资源定位符)，通常用于万维网、RESTful HTTP 和基于 HTTP 的简单对象访问协议(Simple Object Access Protocol，SOAP)。在客户端角色中，通信参与者通过 API 端点访问 API。通信参与者可以同时扮演客户端角色和提供者角色。这种情况下，通信参与者既作为 API 提供者提供某些服务，也在实现中使用其他 API 提供的服务[1]。

面向服务的架构把 API 客户端称为*服务使用者*(或*服务消费者*)，API 提供者称为*服务提供者*[Zimmermann 2009]。在 HTTP 中，API 端点对应一组相关资源。带有预先发布 URI 的*根资源*(home resource)是一种入口级别的 URL，用于定位和访问一种或多种相关资源。

1.3.2 API 端点提供描述操作的 API 契约

如图 1-4 所示，操作由 API 契约描述。除端点地址以外，操作标识符用于区分操作，SOAP 消息体中的顶级 XML 标签就起到这种作用。在 RESTful HTTP 中，HTTP 方法(又称 HTTP *动词*)的名称在单个资源内具有唯一性[2]。

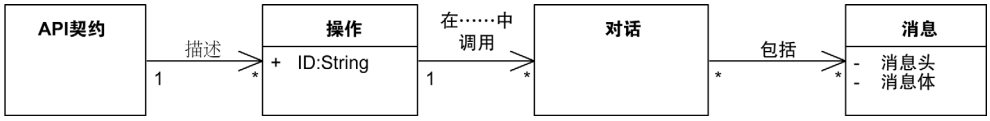

图 1-4 领域模型：操作、对话、消息

1.3.3 消息是对话的组成部分

API 操作由契约描述并由端点提供，可以参与*对话*。对话不同，组合和编排信息的方式也不同，但所有对话都会描述通信参与者之间交换的消息序列。四种主要的对话类型如图 1-5 所示。*请求-应答*消息交换由单条*请求*消息和单条*响应*消息构成。如果没有响应，则表明对话具有*单向*交换性。第三种对话是*事件通知*，包含了触发事件所用的单条消息。最后，对话可能持续很长时间，最初的单条请求消息发送后会收到多条*应答*消息。在这种请求-多应答消息交换中，客户端向提供者发送的一条消息会注册回调，提供者向客户端发送的一条或多条消息则执行回调动作。

三种消息类型是命令消息、文档消息和事件消息[Hohpe 2003]，它们很自然地与对话类型相吻合。例如，文档消息可以通过单向交换对话进行传输；又如，如果客户端关心命令执行结果，那么命令消息需要通过请求-应答对话进行传输。消息可以采用 JSON、XML 等多种格式进行传输。本书主要讨论所有三类消息的内容和结构。

对话的类型还有很多，例如*发布-订阅*机制等更复杂的对话。基本对话可以组合成规模更大的端到端对话场景，这些场景涉及多个 API 客户端与提供者之间的消息交换，甚至可能包括持续数天、数月或数年的托管*业务流程*[Pautasso 2016; Hohpe 2017]。这类高级对话往往见于软件生态系统、企业应用程序以及其他 API 使用场景，但它们不是本书的讨论重点。

1 通信信道的客户端一侧还需要一个网络端点，但本书侧重于探讨 API 而不是通信信道或网络，因此不进行描述。

2 在 OpenAPI 规范中，操作通过 HTTP 方法及其 URI 路径加以标识，还有一个名为 operationId 的附加属性[OpenAPI 2022]。

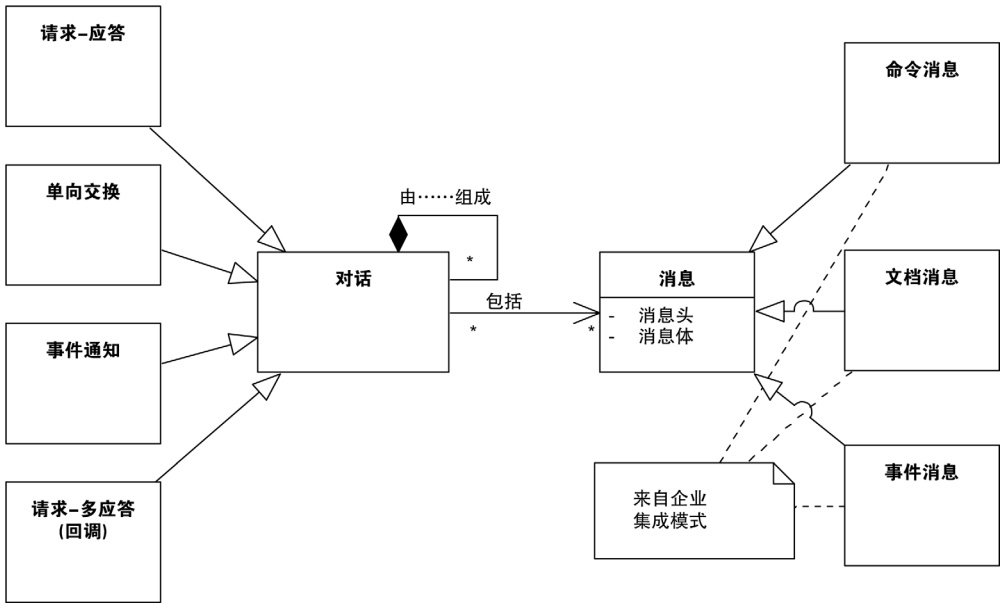

图 1-5 领域模型：对话类型和消息类型

1.3.4 消息结构和表示

如图 1-6 所示，一个或多个*表示元素*(又称参数)构成了通过网络发送的消息的*表示*(请注意，某些技术使用“操作签名”一词来指代参数及其类型)。消息携带元数据和数据，它们保存在消息头和消息体中。地址、消息头和消息体中的表示元素可能已经排序，也可能没有排序；可能会进一步构造为层次结构，也可能不会。这些表示元素通常经过命名，而且可以是静态类型或动态类型。消息既可以包含源地址(以接收返回该地址的应答消息)，也可以包含目标地址。举例来说，返回地址、关联标识符等概念支持消息参与基于内容的消息路由和复杂的长时间对话[Hohpe 2003]。HTTP 资源 API 中的超媒体控件(链接)就包含这样的地址信息。如果消息中没有出现地址，则由通信信道单独承担消息路由的任务。

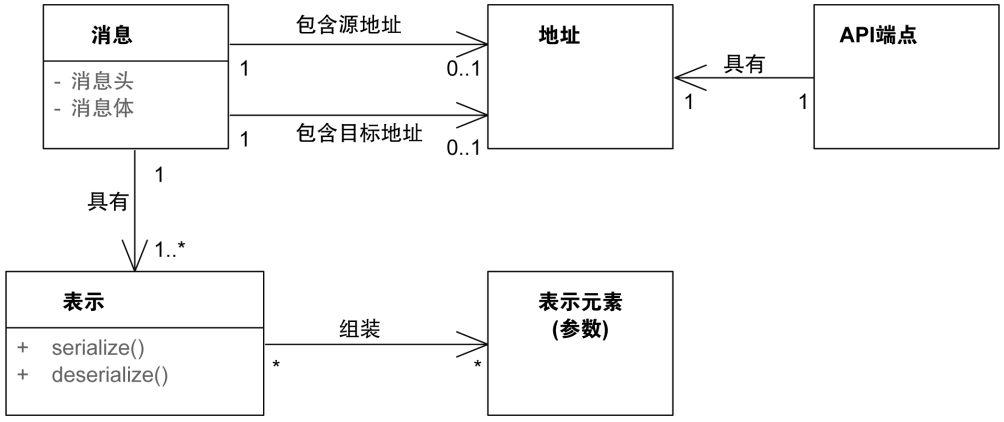

图 1-6 领域模型：消息细节

消息表示也称为*数据传输表示*(Data Transfer Representation，DTR)。在设计 DTR 时，应该避免对客户端和服务器端的编程范式(例如面向对象编程、命令式编程或函数式编程)做出任何假设。客户端与服务器之间交换的是普通消息(例如不包括任何远程对象存根或处理程序)[1]。将编程语言表示转换为可以通过网络发送的 DTR 的过程称为*序列化*或*封送*，相反的操作则称为*反序列化*或*解封送*。这些术语往往见于分布式计算技术和中间件平台[Voelter 2004]。纯文本和二进制格式一般用于收发 DTR，如前所述，JSON 和 XML 是常见的格式。

1.3.5 API 契约

如图 1-7 所示，全部端点操作均由 API 契约(参见图 1-3)指定。API 契约详细描述了所有可能出现的对话和消息，直至协议级别的消息表示(参数、消息体)和网络地址。API 客户端和 API 提供者必须就契约中指定的共享知识协商一致后才能交换数据，因此 API 契约对于实现任何具备互操作性、可测试性和可演进性的运行时通信都至关重要。

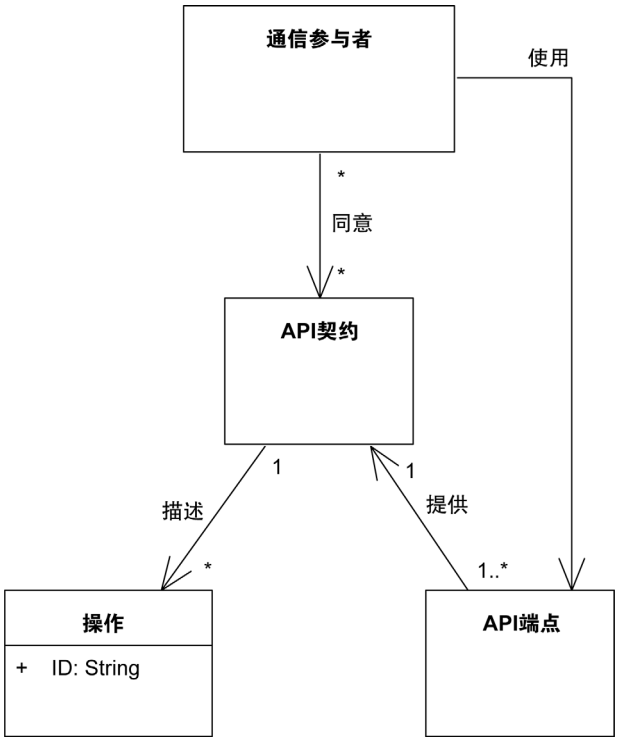

图 1-7 领域模型：API 契约描述了操作(这些操作由包含消息的对话进行调用)

实际上，这种协议可能具有高度不对称性，原因在于许多 API(尤其是公共 API)是由 API 提供者按原样提供的，导致 API 客户端无法自行修改。客户端可以按规定使用，也可以完全不用，此时通信参与者之间不会就契约进行协商或达成正式协议。如果 API 客户端为服务付费，

1 数据传输对象(Data Transfer Object，DTO)是一种程序级别的模式，DTR 相当于 DTO 在网络传输中的对应形式[Fowler 2002; Daigneau 2011]。

那么情况可能有所不同：这类 API 契约也许是各方实际协商的产物，并附有法律合同(甚至会写入法律合同)。API 契约既可以只列出最基本的内容，也可以被纳入更全面的 API DESCRIPTION 或 SERVICE LEVEL AGREEMENT(我们采用的两种模式)。

1.3.6 全书使用的领域模型

在讨论模式语言时，本书采用领域模型中的抽象概念作为词汇表。这是因为根据定义，模式文本必须与平台和技术无关(说明性示例除外)。此外，领域模型中的任何概念和关系都可能成为决定是否采用某种模式的驱动因素。举例来说，消息每次出现时，必须确定其参数结构。第 3 章将深入探讨这些内容，并指导开发人员对所有领域模型元素和模式进行决策。

最后要指出的是，我们采用微服务领域特定语言(Microservice Domain-Specific Language，MDSL)对部分示例进行建模，MDSL 就是以这个领域模型为基础设计而成的，相关讨论参见附录 C。

1.4 本章小结

本章讨论了以下内容：

- API 的定义以及设计高质量 API 的重要性及所面临的挑战。
- API 设计中的期望质量目标，包括耦合和粒度方面的考虑因素，以及积极的开发者体验由哪些要素组成。
- 本书使用的 API 领域术语和概念。

无论是模块化程序内部使用的本地 API，还是连接操作系统进程和分布式系统的远程 API，都已存在很长时间。目前，大多数远程 API 采用 RESTful HTTP、gRPC、GraphQL 等基于消息的协议。远程 API 提供了通过应用程序集成的协议来访问服务器端资源的途径(“访问”“协议”和“集成”这三个词的首字母合起来是“API”)。远程 API 起到重要的桥梁作用，它们把多个系统连接在一起，同时尽可能保持各个系统的独立性，以最大限度减少后续调整所带来的影响。API 及其实现甚至可以具有独立的控制权和所有权。本地 API 和远程 API 都应该以满足客户端对信息或集成的实际需求为己任，并具有明确的目的性。

如果用现实事物作为类比，那么不妨把 API 看作建筑物的入口和大厅。例如，在一幢摩天大楼的大堂里，接待来访者的工作人员既要把他们带到相应的电梯口，也要检查他们是否有权从正门进入。首次前往某处时，第一印象很重要——人类使用软件是这样，API 客户端使用 API 也是这样。因此，API 门户就像其背后应用程序对应的一套“名片”(或构建地图)，用于将服务介绍给可能有兴趣使用它们来编写自己的应用程序的开发人员。名片和入口大厅都会影响来访者体验，API 则会影响开发者体验。

对于本地 API，需要确保它能正确使用；而对于远程 API，还要把分布式计算的谬误考虑在内。举例来说，当最终用户界面(例如基于浏览器的单页应用程序)和分布式云应用程序中的

后端服务需要借助远程 API 进行通信时，就不能认为网络是可靠的。

在架构决策过程中，必须考虑一系列质量属性。从客户端的角度来说，API 的开发质量包括具有良好的开发者体验、可负担的成本和足够的性能；从提供者的角度来说，API 的开发质量包括具有可持续性且易于更改和维护。在整个 API 生命周期中，以下三类质量属性尤为重要。

1. 开发质量：从开发人员的角度来说，API 应该易于发现、学习和理解，并能方便地用来构建应用程序。这些要素统称为 API 应提供积极的开发者体验，通过功能、稳定性、易用性、清晰性这四大支柱进行定义。

2. 操作质量：API 及其实现应该是可靠的，并满足规定的性能、可靠性和安全性要求。API 在运行时应该具备可管理性。

3. 管理质量：API 应该可以随时间的推移进行演进和维护，最好能够同时兼顾扩展性和向后兼容性，从而在调整 API 时不会影响现有的客户端。为此，必须在敏捷性与稳定性之间找到平衡。

实现高质量的 API 设计和演进具有一定的难度(也很有趣)，原因如下：

- API 应该具有长久的生命力，需要从短期和长期两个方面来衡量 API 成功与否。
- 各方需要就 API 对外公开的功能和相关质量属性协商一致、达成共识。
- API 的粒度既取决于对外公开的端点数量和操作数量，也取决于这些操作中请求和响应消息包含的数据契约。在少量富操作和大量窄操作之间做出选择十分重要。
- 需要实施耦合控制。零耦合意味着断开连接；API 客户端与 API 提供者之间的了解越深入，双方的耦合性就越高，独立演进就越困难。
- 虽然 API 技术“你方唱罢我登台”，但 API 设计的基本概念和相关的架构决策及其选项和标准保持不变。

本书侧重于讨论用于连接系统及其部件的远程 API。API 提供者对外公开 API 端点，端点提供操作，而操作通过消息交换进行调用。这些交换中的消息构成了对话，它们包含简单或结构化的消息表示元素。这些概念定义在 API 设计和演进采用的领域模型中。本书致力于将这些概念付诸实践，从而设计出能够满足客户端需求的高质量 API。

第 2 章将介绍一个虚构的大型 API 和服务设计示例，第 3 章将以决策驱动因素为基础分析本节讨论的设计挑战和要求。第 II 部分将详细阐述模式及其解决方案，并讨论 API 的成功要素和质量属性。