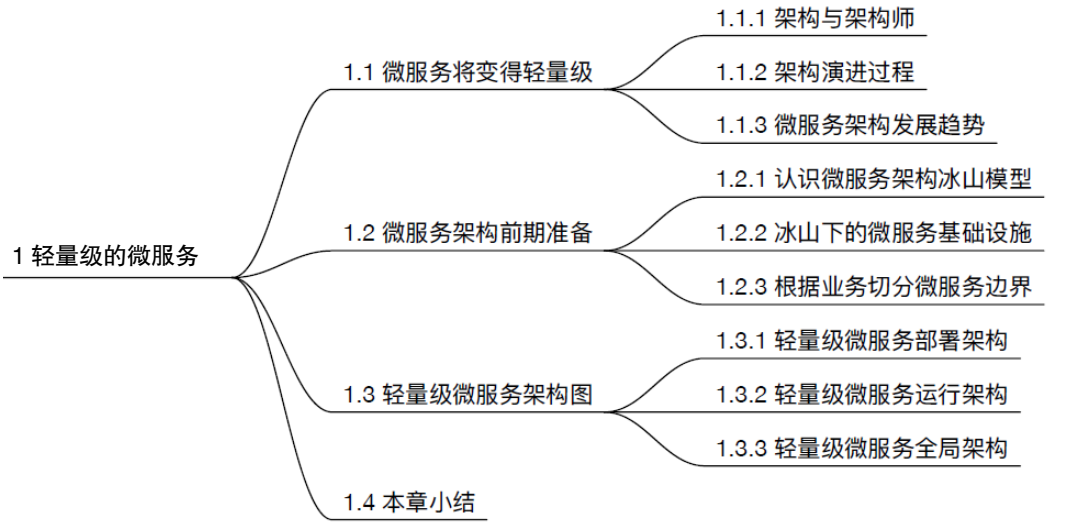
1
chapter

第 1 章
轻量级的微服务


1 轻量级的微服务
1.1 微服务将变得轻量级
1.1.1 架构与架构师
1.1.2 架构演进过程
1.1.3 微服务架构发展趋势
1.2 微服务架构前期准备
1.2.1 认识微服务架构冰山模型
1.2.2 冰山下的微服务基础设施
1.2.3 根据业务切分微服务边界
1.3 轻量级微服务架构图
1.3.1 轻量级微服务部署架构
1.3.2 轻量级微服务运行架构
1.3.3 轻量级微服务全局架构
1.4 本章小结

1.1 微服务将变得轻量级

架构需要由人去设计，这些人被称为架构师。或许很多人并未授予架构师的头衔，但自己却从事着架构的工作。我们认为，架构这项工作永远都需要由人去完成，可能短期内都无法由机器来取代。如果我们不理解什么是架构，或者对架构师的职责感到疑惑，那么很难让架构这项工作有效地落地。我们将在本节重新认识架构，并重新定义架构师的职责。此外，架构演进是一个曲折的过程，但我们却不难看出架构的发展规律，甚至还能推测出架构将来的发展趋势。我们相信，微服务一定不是架构的终点，它或许只是架构从重量级转型为轻量级的桥梁，我们正是设计并建造这座桥梁的工程师。

现在我们从架构与架构师的角度开始出发，开启轻量级微服务的架构探险之旅。

1.1.1 架构与架构师

可能绝大多数的程序员都想成为一名优秀的架构师，每天都能从事技术架构的相关工作，编点框架代码，画点架构图，写点 PPT，帅气地站在讲台上给程序员们进行技术培训。大家普遍认为，架构师的代码比别人写得少，但是工资却比别人拿得多，架构师是技术团队中技术最牛的人，别人搞不定的技术问题，在架构师眼中都是小菜一碟。

这样的人真的是架构师吗？

我们认为，他们不是架构师，而是技术专家。其实架构师与技术专家有着本质的区别，从他们所关注的技术方向来看，架构师更偏向技术广度，而技术专家更偏向技术深度，如图 1-1 所示。

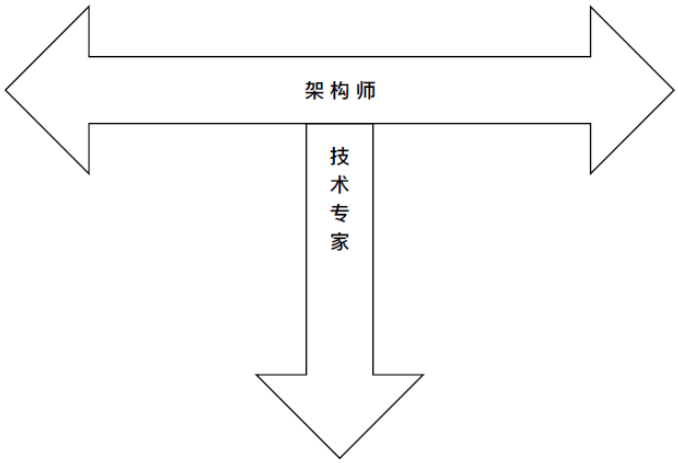


图 1-1 架构师与技术专家的区别

换句话说，架构师需要有较强的综合能力，他们需要接触的技术领域较广，但他们所掌握的技术专业能力却没有技术专家那么深。如果我们想成为一名架构师，那么就不应该把所有的精力都投入在某个技术领域上，而是要学会分散自己所关注的层面，做到在众多技术领域上都要有一定的深度。

架构师除了需要具备在技术上所需的“硬技能”，还需要不断完善自己的“软技能”，比如沟通、组织、学习等技能。有时候软技能可能比硬技能更加重要，甚至软技能还会影响自己的职业发展。如果没有较好的软技能，架构师将无法将自己所设计的架构顺利地移交到程序员们手中，并指导他们将其真正落地。架构师正是通过他们所具备的综合能力来带领技术团队，解决不断出现的技术挑战。

架构师的职责是什么？

我们的回答是：制定规范 + 指导落地。

架构师根据业务需求所制定的合理且可落地的技术规范，我们将这样的规范称为架构。

将架构工作做好犹如我们用两条腿走路一样，左腿迈出去表示“制定规范”，右腿跟上来表示“指导落地”，如图 1-2 所示。

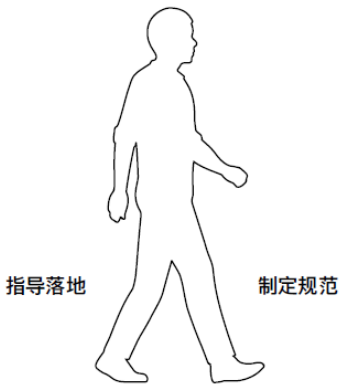


图 1-2　架构师的职责

如果左腿迈出去，右腿没跟上来，那不是架构师，可能是需要拄拐的人。然而，我们身边却有一些这样的不合格的架构师，他们只懂得制定规范，却忽略了指导落地。如果架构无法落地，那么就无法称为架构了。

此外，还有一些架构师认为，架构只是技术层面上的问题，自己设计的架构应该用到市面上最为流行的新技术，比如别人公司在用微服务，那么自己公司也要用起来。如果将架构工作脱离于业务需求，我们认为这不是做架构，而是玩技术。脱离业务来设计架构是对架构的不尊重。微服务是一种应用系统架构，需要架构师围绕业务进行设计。

但是，我们绝不要为了微服务而去微服务。

从事微服务架构工作的架构师，相比传统架构的架构师而言，所要求的技能更加全面。他们不仅仅是系统架构师，也是业务分析师，他们的责任重大且挑战艰巨。

从大的方向来看，微服务架构师需要具备以下基本职责。

（1）分析业务需求并切分微服务边界。

（2）定义架构规范与文档标准。

（3）确保微服务架构顺利落地。

（4）改善微服务架构并提高开发效率。

职责与挑战往往是无法分离的，微服务架构师必须面对并克服这些挑战。

（1）架构需要适应不断变化的业务需求。

（2）架构具备稳定性、扩展性、安全性、容错性等。

（3）使技术团队深刻理解微服务思想。

（4）展现微服务架构的价值。

我们认为，传统架构师转型为微服务架构首先需要做到的是深刻理解业务，而不是表面上了解需求。业务和需求其实是两码事，业务的背后反映了客户的刚需，而需求往往是产品经理根据业务刚需所指定的解决方案。作为微服务架构师，我们需要透过需求的表象去理解业务的本质。其次需要做到的是不断优化架构，让架构变得更加简单，更加轻量级。我们要将昨天最好的表现，当成今天最低的要求，只有在技术上不断要求自己，才能让架构变得更好。

1.1.2 架构演进过程

话说天下大势，分久必合，合久必分，对于架构演进过程而言，也符合这个规律。

最早的应用程序实际上是没有任何架构的，因为那时业务比较简单，没有架构也许是合理的，如图 1-3 所示。

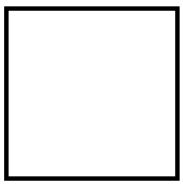


图 1-3 没有架构

但随着业务不断复杂起来，我们意识到架构可以做到水平分层，比如表现层、逻辑层、数

据层等，我们可在不同的层上实现每一层所关注的内容，我们称其为“关注点分离”。但此时的架构更像是一块“铁板”，每一层的无法进行分离，因此我们也将这样的架构称为“单块架构”，如图 1-4 所示。

从此架构发生了更为复杂的变化，层次结构越来越深，而且不再局限于水平方向上的分层，实际上越来越多的应用程序围绕着不同的业务需求来实现，此时出现了“垂直分层架构”，每个垂直应用中实际上都是一个独立的子系统，它们共同组成了整个应用系统。然而，这些子系统一般可以部署在不同的服务器上，这些服务器可以分布在不同的地域中，我们也称其为“分布式架构”，如图 1-5 所示。

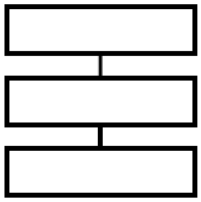


图 1-4　水平分层架构（单块架构）

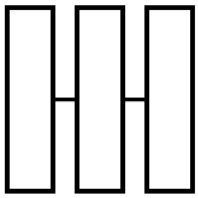


图 1-5　垂直分层架构（分布式架构）

业务并没有停止发展的脚步，架构的演变也是如此。分布式应用之间的调用越来越多，整个系统的复杂度急剧上升，人们希望找到一种途径来降低分布式应用之间的耦合。此时出现了面向服务架构（SOA），人们希望 SOA 能成为解决分布式应用系统复杂性的“银弹”，然而事实却事与愿违。应用的复杂性不仅没有得到解决，反而还让架构变得更加复杂，同时也形成了大量的 SOA 商业产品，这些现象让人们更加恐惧 SOA，将它视为复杂和昂贵的代名词，如图 1-6 所示。

随着互联网行业不断发展，用户对产品的体验要求越来越高，前端的价值逐渐被凸显出来，此时出现了“前后端分离”的趋势，前端工程师专心在界面展现与数据渲染上，后端工程师关注在业务逻辑与数据结构上，前后端只需通过 REST API 进行交互，工作分工更加明确，开发效率更加高效，如图 1-7 所示。

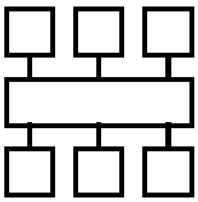


图 1-6　面向服务架构（SOA）

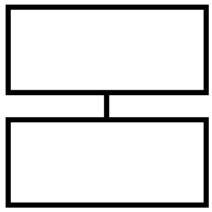


图 1-7　前后端分离架构

似乎很长时间都未出现任何技术能与当年的 SOA 相提并论，除了微服务架构。微服务的概念在 2014 年首次被提出以来，近几年一直是应用架构领域的核心话题。但人们往往还是容易想

到当年的 SOA，认为微服务与 SOA 有着相同的目的，只是实现细节不太一样罢了，如图 1-8 所示。

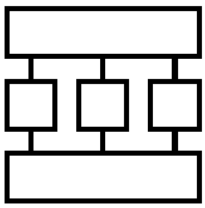


图 1-8　微服务架构

微服务与 SOA 到底有何区别？

我们认为，微服务是 SOA 的一种落地方案。

SOA 是一种面向服务的架构思想，微服务也同样推崇这种思想。微服务是将一个大型的单块架构，拆分为多个细粒度服务的架构。微服务更加考验我们的是，深入理解业务并合理地对服务边界进行切分。微服务的概念相比 SOA 更容易落地的原因不是概念上的创新，而是技术上的突破，尤其是容器与自动化运维技术的普及与应用。

我们始终相信，微服务并不是架构发展的终点，也许是新架构时代的起点。

1.1.3　微服务架构发展趋势

微服务架构所涉及的范围相当广泛，我们不妨从多个角度来推测微服务架构的发展趋势。

从微服务开发角度来看，我们认为微服务的开发框架将变得更加多样化。

开发人员可使用更加适合的开发框架来完成微服务业务实现，而不再拘泥于某一种编程语言，只需确保对外提供统一的 API 接口方式即可。甚至可将查询与修改操作相分离，查询操作可以用一种更加轻量级的编程语言来实现，而修改操作会涉及事务，一般需要借助开发框架的事务特性来保证。

我们坚信，微服务必将坚持走轻量级技术路线。

究竟什么是轻量级？

我们认为，轻量级必须包含三个特征：易用、快速、稳定。

我们希望微服务架构中所涉及的技术都能够快速上手，运行时不占用过多的系统资源且性能突出，而且能够长期稳定地运行。

从微服务部署角度来看，我们认为微服务的部署过程将变得更加自动化。

部署微服务不再通过手工的方式去完成，因为这样既低效又容易出错，我们更加倾向于使

用软件工具将其自动完成。要实现自动化部署这个目标，我们往往无法一步到位，最合理的方式是“先让它跑起来，再让它跑得快”。也就是说，早期的自动化部署方案也许不够完备，或多或少会存在一些人工参与的情况，其实这些都再正常不过了，但我们需要不断优化，努力通过自动化技术来取代重复性的人工操作。

最后想说的是，自动化虽好，但不要为了自动化而去自动化，或许有些环节通过手工处理才是最有效的方式。

1.2　微服务架构前期准备

搭建微服务架构绝不是一件轻松的事情，我们不仅要对微服务概念有着深刻的理解，还要研究大量的技术工具，并掌握它们的优缺点，最终需要结合技术团队对技术的了解程度，选择最为合适的技术选型。这些纯技术的技术工具只是微服务的基础设施，我们还需要在此基础设施上围绕真实的业务场景，对微服务边界进行合理地切分。我们将在本节介绍微服务的冰山模型，大家将从这座冰山中看到微服务架构的全貌，随后我们将深入冰山之下的世界，去探索微服务基础设施的八大中心，最后我们还会介绍一些关于切分微服务边界的原则和技巧。

我们现在就从微服务冰山模型开始吧，这座冰山似乎比我们想象中的要大很多。

1.2.1　认识微服务架构冰山模型

有些人认为，使用了 Spring Boot 开发框架就是拥有了微服务，其实这样的认识是不正确的。Spring Boot 只是一款微服务的开发框架，而且仅能用于 Java 应用程序中，毫无疑问，它只是微服务的冰山一角。此外，我们建议大家结合自身的业务场景，选择更为合适的编程语言以及开发框架来实现微服务，而不要拘泥于一种编程语言。

还有一些人认为，用上了 Docker 就进入了微服务时代，其实这样的认识也是不正确的。Docker 只是一种封装微服务应用程序的容器化技术，它改变了应用程序的交付方式，也加速了微服务架构的落地速度。可以肯定地说，如果没有 Docker 容器技术，或许今天我们无法听到微服务的概念。

如果将微服务架构中所涉及的技术栈比喻为一座冰山的话，那么冰山之上最显而易见的部分就是微服务的开发框架与容器技术了，Spring Boot 与 Docker 都属于冰山之上的技术。

冰山之下到底有哪些技术呢？

我们认为冰山之下的技术是整个微服务架构的基石，它们构成了整个微服务架构的基础设施。比如我们在上册中学习的 ZooKeeper 服务注册表、Node.js 服务网关、Jenkins 持续部署系统等，它们都属于冰山之下的部分。

我们可通过一幅图来描绘微服务架构这座冰山，如图 1-9 所示。

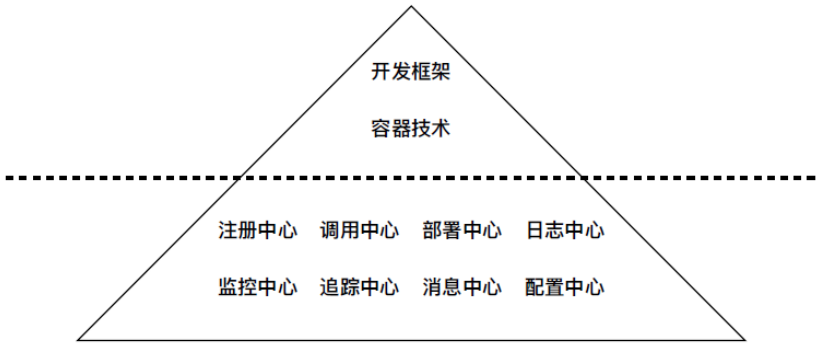


图 1-9　微服务架构冰山模型

由于服务注册表集中管理了微服务相关的服务配置，因此我们也将它称为“注册中心”；由于服务网关是前端应用程序的唯一入口，因此我们也将其称为“调用中心”；同样，我们也将持续部署系统称为“部署中心”。这些中心都汇集在微服务冰山模型之下，除此之外，还有其他功能的中心，这些中心共同构成了微服务的基础设施。

1.2.2　冰山下的微服务基础设施

冰山下的微服务基础设施，实际包括了八大中心。

（1）注册中心：用于注册微服务相关配置信息的中心，我们选用 ZooKeeper 实现。

（2）调用中心：用于提供给前端调用的统一入口，我们选用 Node.js 实现。

（3）部署中心：用于编译并打包微服务源码并将其部署到 Docker 引擎中，我们选用 Jenkins 实现。

（4）日志中心：用于收集并管理微服务应用程序中产生的日志，第 2 章中将详细介绍。

（5）监控中心：用于监控微服务的实时运行状况，第 3 章中将详细介绍。

（6）追踪中心：用于最终微服务的调用轨迹，第 3 章中将详细介绍。

（7）消息中心：用于解耦微服务之间的调用关系，第 5 章中将详细介绍。

（8）配置中心：用于管理微服务应用程序所需的配置参数，第 7 章中将详细介绍。

也许大家看到以上八大中心后会产生一些疑惑：很多人说微服务是去中心化的，为何我们还要提供这些中心呢？

我们认为，中心分为两类：一类是含有业务意义的中心；另一类是不含业务意义的中心（只是技术层面的中心）。

在微服务架构中我们应该去除的是含有业务意义的中心，而不是去除技术层面上的广义中心。比如，我们所设计的调用中心内部是不可能带有任何业务流程的，它只是一个纯技术层面的框架。对于其他中心也是如此，绝对不含有任何的业务，否则我们就应该将其去除。

1.2.3　根据业务切分微服务边界

凡是学习过微服务的人都知道，我们需要根据业务来切分微服务边界。道理可能大家都懂，但或许仍然不知道应该怎么去做。例如，切分微服务边界有哪些关键性步骤，以及包含哪些重要性原则？

经过大量的微服务实践，我们总结了以下五个步骤，可帮助大家有效地切分微服务边界。

第一步：梳理业务流程。

在切分微服务之前，我们要做的第一件事情就是梳理业务流程。不妨找业务专家咨询，通过与他们沟通从而了解真实的业务流程，并将其绘制成流程图。对于过于复杂的业务流程，我们也可单独绘制流程图，并增加相关的流程说明。当然也能提供相应的状态图，用于说明业务流程中所涉及状态的变化过程。

花再多时间去分析业务流程都不过分，现在所花的每一分钟都是相当值得的。

第二步：抽取公共服务。

在业务流程中与业务不太相关的部分，我们可考虑将其剥离出来，并形成公共服务。例如，邮件发送、文件上传、其他第三方接口等。每种公共服务都对应一个微服务，每个微服务都有相关 API，每个 API 都有自己的输入与输出。这些 API 一定要形成文档，以便其他服务调用。

一般情况下，抽取的公共服务都不太会变化，我们一定要想办法将不变的东西从可变的世界中抽取出来。

第三步：定义业务服务。

当公共服务抽取完毕后，业务流程中剩下来的部分就是业务服务了。建议刚开始实施微服务时，不要将业务服务的边界切得太细，可以考虑先“大切几块”，但需要确保每个服务之间尽量不要有依赖关系。换句话说，每个服务都是独立的，虽然此时服务的块头可能比较大。

我们先确保这些大块头服务可以运行在微服务基础设施上，再不断将它们进行细化，拆解为更小的服务。

第四步：设计数据模型。

深入到每个业务服务中，我们首先要做的是定义它底层所涉及的数据模型，也称为“领域模型”。此时会涉及数据库表结构设计，以及数据模型与关系设计。在数据层面上的设计是至关

重要的，如果该部分设计得不到位，将增加后期实现微服务的成本。

数据模型的设计同样也需要进行文档化，这些文档将指导后端工程师顺利地完成微服务实现。

第五步：定义服务接口。

底层的数据模型设计完毕后，我们将视角转换到顶层的服务接口上。服务接口实际上就是一组 API，这些 API 需做到职责单一，而且需要通过名称就能识别出它的业务含义。建议确保每个 API 的命名是全局唯一的，也建议每个 API 都有各自的版本号，版本号可以用自增长的方式来体现。

服务接口也需要进行文档化，这些文档一般由后端工程师编写，并提供给前端与测试工程师阅读。

1.3 轻量级微服务架构图

作为一名微服务架构师，我们不仅需要深入理解业务，并能准确地划分服务边界，而且还需要深入理解微服务，并能合理地选择技术选型。我们认为，微服务架构实际上分为两大部分，其中一部分对应部署阶段，另一部分对应运行阶段。这两部分包含了大量的技术工具，我们需要结合多方面因素来考虑，选择最为合适的技术选型来搭建微服务架构，并确保它能保持轻量级。

本节将着眼于微服务的部署与运行两大阶段，通过图文方式来描述轻量级微服务架构。

1.3.1 轻量级微服务部署架构

当开发人员完成了微服务的细节实现后，首先要做的是确保自己所写代码的可用性，他们往往会借助单元测试工具来保证这一环节不出问题。当他们将源码提交并推送到代码仓库后，此时部署中心将从代码仓库中获取源码，并执行编译与打包操作。

不仅如此，部署中心还需从配置中心获取对应运行环境的配置参数，并生成相应的配置文件，并将这些配置文件与应用程序一同复制到 Docker 镜像中，最后还需将此镜像上传到镜像仓库，以便后续可从镜像仓库中下载指定的镜像，从而运行相应的 Docker 容器。

此外，部署中心还可扫描源码并自动生成 API 文档站点，以便其他技术人员可随时从文档站点中查看最新部署服务所包含的 API 文档。当然，我们还可以对所需完成的微服务仅提供简单的代码实现，这样就能将微服务文档的输出时间尽可能提前，以便其他技术人员能够更早地了解到微服务的相关 API 信息，随后再去完成更加细节的代码实现，这是我们推荐的工作方法。

当 Docker 镜像上传到镜像仓库后，部署中心可在不同的运行环境下根据特定的镜像来启动相应的 Docker 容器。为了便于描述，我们将该容器称为“服务容器”，它包含了承载微服务的应用程序及其配置文件。

当服务容器启动后，会自动将其配置信息写入注册中心，与此同时，部署中心也会连接到注册中心，并设置服务的版本号，以便于在后续调用服务时可根据版本号来识别当前可用的服务。

我们可将以上过程绘制为一幅架构图，如图 1-10 所示。

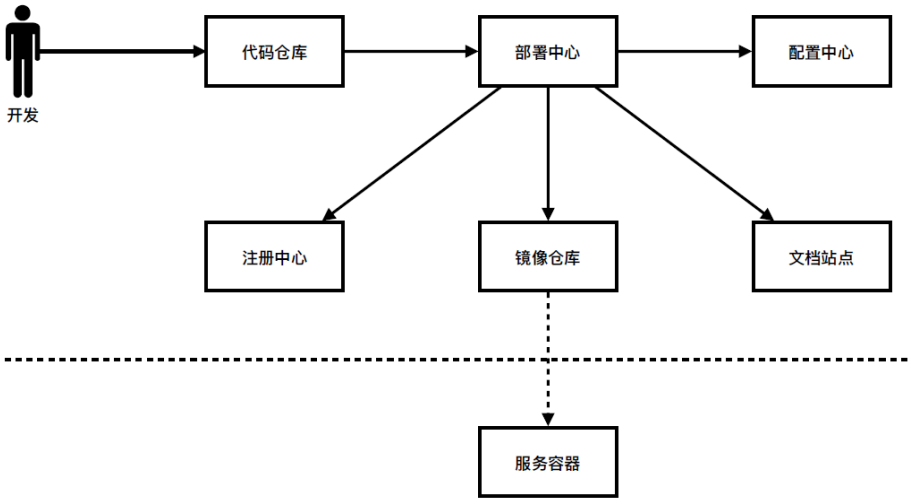


图 1-10　轻量级微服务部署架构

可见，在微服务部署阶段中，部署中心是其中的主角，它支配其他组件，使服务可以成功部署，我们需要确保它的稳定性。

1.3.2　轻量级微服务运行架构

当用户通过浏览器或移动端访问应用系统时，请求首先将进入服务网关，因为它是所有请求调用的中心，我们也将其称为“调用中心”。它虽然是不带任何业务的中心，但我们需要确保它所做的事情足够少，让它不会成为整个应用系统的调用瓶颈。

随后调用中心将连接注册中心，并通过服务名称从注册中心中获取服务所在的 IP 地址与端口号（即服务地址），该过程称为“服务发现”，进而调用中心可根据服务地址以反向代理的方式来调用具体的服务容器，该过程称为“服务调用”。

在服务容器中可能会触发一些事件，这些事件将以消息的方式写入消息中心，以便其他服务可监听消息中心并从中获取相应的消息。该方案可解决服务之间的耦合问题，同时能将同步

调用转为异步调用，提高整个应用系统的吞吐率。

在服务容器运行时会产生大量的日志，我们可将这些日志统一写入日志中心，并能在日志中心所提供的控制台上查询具体的日志信息。此外，日志中心也能帮助我们快速地定位并分析系统出现的异常状况。

为了观察服务容器是否运行正常，我们可借助监控中心所输出的图形化数据来判断。监控中心将不断地收集服务容器中的运行状态，包括 CPU、内存、硬盘、网络，以及应用程序的 JVM 内存使用情况。

由于微服务很难切得干净，服务之间难免会出现少量的调用关系，我们可将每次调用所产生的相关信息写入追踪中心，并通过追踪中心提供的图形化界面来查看服务之间的调用轨迹以及所产生的调用延时，从而可分析出服务调用所产生的性能瓶颈。

我们可将以上过程绘制为一幅架构图，如图 1-11 所示。

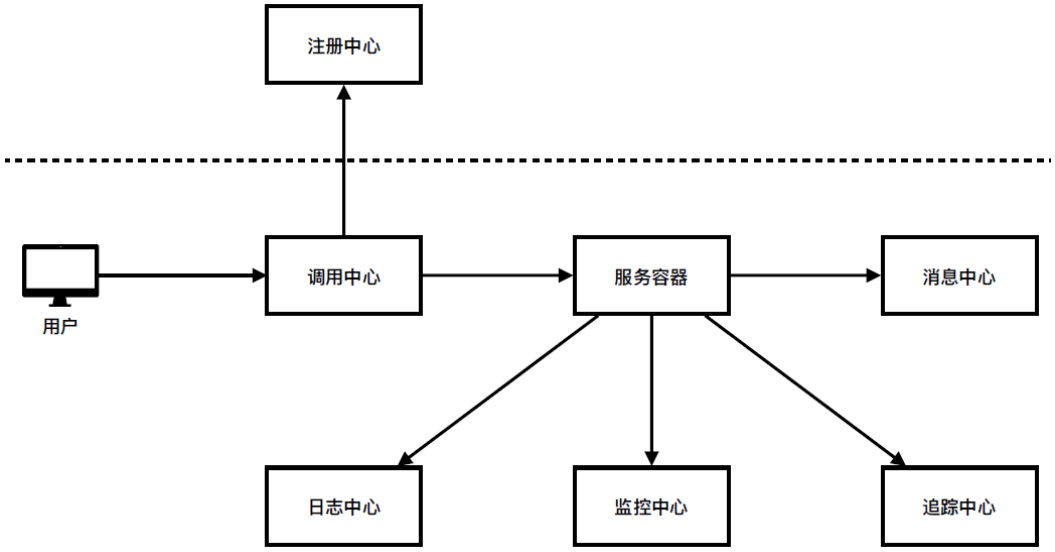


图 1-11　轻量级微服务运行架构

可见，在微服务运行阶段中，调用中心是其中的主角，注册中心作为它的数据来源，服务容器作为它的调用目标，它需要具备良好的高性能与高可用等特性。

1.3.3　轻量级微服务全局架构

我们用一张图将轻量级微服务的部署架构与运行架构进行整合，它就是轻量级微服务架构的全貌，如图 1-12 所示。

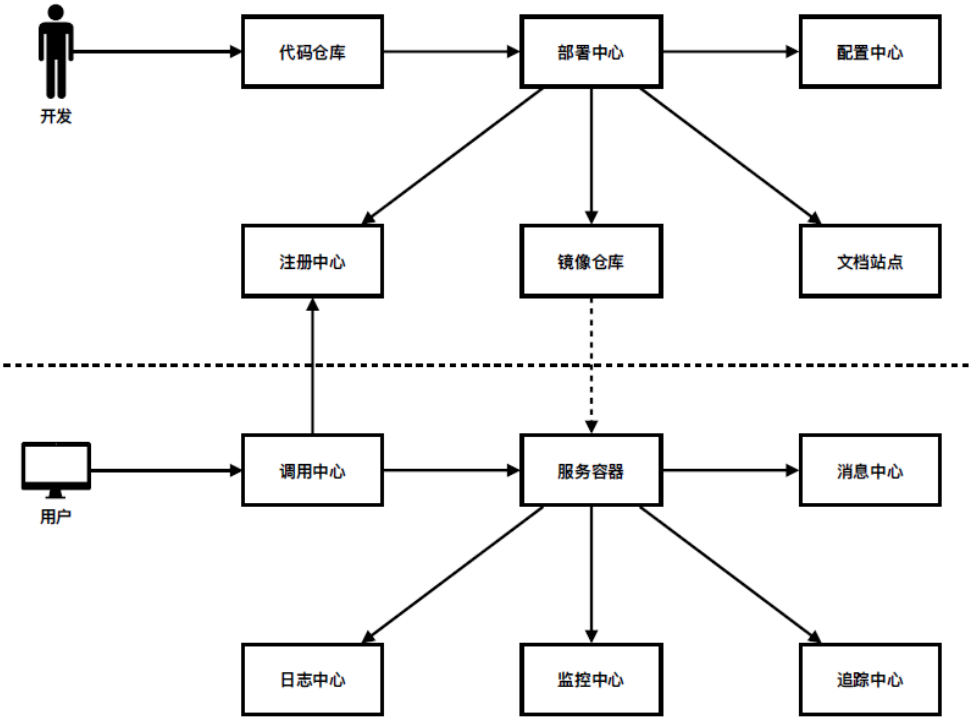


图 1-12　轻量级微服务全局架构

图 1-12 所示的架构图中包括 12 个组件，其中的代码仓库、部署中心、注册中心、镜像仓库、调用中心、服务容器这 6 个组件已在上册中进行描述，剩下的配置中心、文档站点、消息中心、日志中心、监控中心、追踪中心这 6 个组件将在下册后续章节中深入探索。

1.4　本章小结

本章从宏观上描述了轻量级微服务架构，为后续探险历程提供了明确的蓝图。首先我们从架构与架构师开始讲起，简单回顾了架构演进的过程与微服务的发展趋势，我们认为，微服务的出现是必然的，而且它将朝着轻量级方向发展。随后我们探讨了在搭建微服务架构之前需要准备的工作，也认识了微服务架构的“冰山模型”，本书将重点集中在冰山之下的微服务基础设施中，此外还介绍了切分微服务边界的方法和技巧，我们认为，合理地切分微服务边界是微服务架构师的职责之一。最后我们从部署与运行两个角度来观察微服务架构，并以一幅架构全景图来结束本章，随后的章节将围绕这张架构图的相关部分进行展开，我们会选择最为合适的技术选型来搭建这款轻量级微服务架构。

下一章我们将重点关注在微服务日志方面，并实现轻量级微服务架构中的“日志中心”。