第 1 章 Chapter 1

了解微服务

SpringBoot 是一个可使用 Java 构建微服务的微框架，所以在了解 SpringBoot 之前，我们需要先了解什么是微服务。

1.1 什么是微服务

微服务（Microservice）虽然是当下刚兴起的比较流行的新名词，但本质上来说，微服务并非什么新的概念。实际上，很多 SOA 实施成熟度比较好的企业，已经在使用和实施微服务了。只不过，它们只是在闷声发大财，并不介意是否有一个比较时髦的名词来明确表述 SOA 的这个发展演化趋势罢了。

微服务其实就是服务化思路的一种最佳实践方向，遵循 SOA 的思路，各个企业在服务化治理的道路上走的时间长了，踩的坑多了，整个软件交付链路上各个环节的基础设施逐渐成熟了，微服务自然而然就诞生了。

当然，之所以叫微服务，是与之前的服务化思路和实践相比较而来的。早些年的服务实现和实施思路是将很多功能从开发到交付都打包成一个很大的服务单元（一般称为 Monolith），而微服务实现和实施思路则更强调功能趋向单一，服务单元小型化和微型化。如果用“茶壶煮饺子”来打比方的话，原来我们是在一个茶壶里煮很多个饺子，现在（微服务化之后）则基本上是在一个茶

壶煮一个饺子，而这些饺子就是服务的功能，茶壶则是将这些服务功能打包交付的服务单元，如图 1-1 所示。

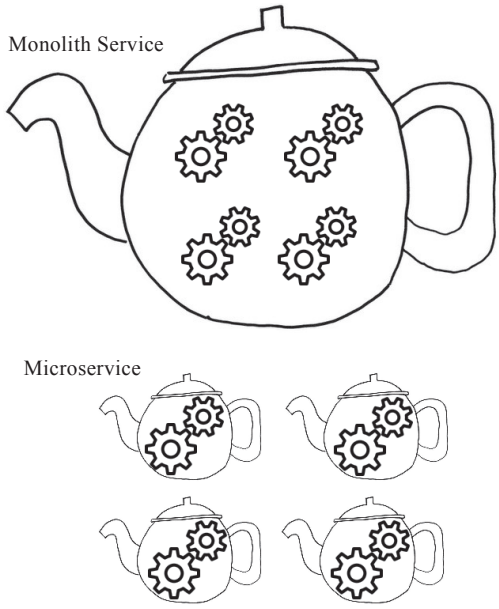

图 1-1　论茶壶里煮“饺子”的不同形式

所以，从思路和理念上来讲，微服务就是要倡导大家尽量将功能进行拆分，将服务粒度做小，使之可以独立承担对外服务的职责，沿着这个思路开发和交付的软件服务实体就叫作“微服务”，而围绕着这个思路和理念构建的一系列基础设施和指导思想，笔者将它称为“微服务体系”。

1.2　微服务因何而生

微服务的概念我们应该大体了解了，那么微服务又是怎么来的？原来将很多功能打包为一个很大的服务单元进行交付的做法不能满足需求吗？

实际上，并非原来“大一统”（Monolith）的服务化实践不能满足要求，也不是不好，只是，它有自己存在的合理场景。对于 Monolith 服务来说，如果团队不大，软件复杂度不高，那么，使用 Monolith 的形式进行服务化治理是比较合适的，而且，这种方式对运维和各种基础设施的要求也不高。

但是，随着软件系统的复杂度持续飙升，软件交付的效率要求更高，投入的人力以及各项资源越来越多，基于 Monolith 的服务化思路就开始“捉襟见肘”。

在开发阶段，如果我们遵循 Monolith 的服务化理念，通常会将所有功能的实现都统一归到一个开发项目下，但随着功能的膨胀，这些功能一定会分发给不同的研发人员进行开发，造成的后果就是，大家在提交代码的时候频繁冲突并需要解决这些冲突，单一的开发项目成为了开发期间所有人的工作瓶颈。

为了减轻这种苦恼，我们自然会将项目按照要开发的功能拆分为不同的项目，从而负责不同功能的研发人员就可以在自己的代码项目上进行开发，从而解决了大家无法在开发阶段并行开发的苦恼。

到了软件交付阶段，如果我们遵循 Monolith 的服务化理念，那么，我们一定是将所有这些开发阶段并行开发的项目集合到一起进行交付，这就涉及服务化早期实践中比较有名的“火车模型”，即交付的服务就像一辆火车，而这个服务相关的所有功能对应的项目成果，就是要装上火车车厢的一件件货物，交付的列车只有等到所有项目都开发测试完成后才可以装车出发，完成整个服务的交付。很显然，只要有一个车厢没有准备好货物（即功能项目未开发测试完成），火车就不能发车，服务就不能交付，这大大降低了服务的交付效率。如果每个功能项目可以各自独立交付，那么就不需要都等同一辆火车，各自出发就可以了。顺着这个思路，自然而然地，大家逐渐各自独立，每一个功能或者少数相近的功能作为单一项目开发完成后将作为一个独立的服务单元进行交付，从而在服务交付阶段，大家也能够并行不悖，各自演化而不受影响。

所以，随着服务和系统的复杂度逐渐飙升，为了能够在整个软件的交付链路上高效扩展，将独立的功能和服务单元进行拆分，从而形成一个一个的微服务是自然而然发生的事情。这就像打不同的战役一样，在双方兵力不多、战场复杂度不高的情况下，Monolith 的统一指挥调度方式是合适的；而一旦要打大的战役（类似于系统复杂度提升），双方一定会投入大量的兵力（软件研发团队的规模增长），如果还是在狭小甚至固定的战场上进行厮杀，显然施展不开！所以，小战役有小战役的打法，大战役有大战役的战法，而微服务实际上就是一种帮助扩展组织能力、提升团队效率的应对“大战役”的方法，它帮助我们从软件开发到交付，进而到团队和组织层面多方位进行扩展。

总的来说，一方面微服务可以帮助我们应对飙升的系统复杂度；另一个方面，微服务可以帮助我们进行更大范围的扩展，从开发阶段项目并行开发的扩展，到交付阶段并行交付的扩展，再到相应的组织结构和组织能力的扩展，皆因微服务而受惠。

1.3 微服务会带来哪些好处

显然，随着系统复杂度的提升，以及对系统扩展性的要求越来越高，微服务化是一个很好的方向，但除此之外，微服务还会给我们带来哪些好处？

1.3.1 独立，独立，还是独立

我们说微服务打响的是各自的独立战争，所以，每一个微服务都是一个小王国，这些微服务跳出了“大一统”（Monolith）王国的统治，开始从各个层面打造自己的独立能力，从而保障自己的小王国可以持续稳固的运转。

首先，在开发层面，每个微服务基本上都是各自独立的项目（project），而对应各自独立项目的研发团队基本上也是独立对应，这样的结构保证了微服务的并行研发，并且各自快速迭代，不会因为所有研发都投入一个近乎单点的项目，从而造成开发阶段的瓶颈。开发阶段的独立，保证了微服务的研发可以高效进行。

服务开发期间的形态，跟服务交付期间的形态原则上是不需要完全高度统一的，即使我们在开发的时候都是各自进行，但交付的时候还是可以一起交付，不过这不是微服务的做法。在微服务治理体系下，各个微服务交付期间也是各自独立交付的，从而使得每个微服务从开发到交付整条链路上都是独立进行，这大大加快了微服务的迭代和交付效率。

服务交付之后需要部署运行，对微服务来说，它们运行期间也是各自独立的。

微服务独立运行可以带来两个比较明显的好处，第一个就是可扩展性。我们可以快速地添加服务集群的实例，提升整个微服务集群的服务能力，而在传统 Monolith 模式下，为了能够提升服务能力，很多时候必须强化和扩展单一结点的服务能力来达成。如果单结点服务能力已经扩展到了极限，再寻求扩展的

话，就得从软件到硬件整体进行重构。

软件行业有句话：“ Threads don't scale, Processes do!”，很明确地道出了原来 Monolith 服务与微服务在扩展（Scale）层面的差异。

对于 Java 开发者来说，早些年（当然现在也依然存在），我们遵循 Java EE 规范开发的 Web 应用，都需要以 WAR 包的形式部署到 TOMCAT、Jetty、RESIN 等 Web 容器中运行，即使每个 WAR 包提供的都是独立的微服务，但因为它们都是统一部署运行在一个 Web 容器中，所以扩展能力受限于 Web 容器作为一个进程（process）的现状。无论如何调整 Web 容器内部实现的线程（thread）设置，还是会受限于 Web 容器整体的扩展能力。所以，现在很多情况下，大家都是一个 TOMCAT 只部署一个 WAR，然后通过复制和扩展多个 TOMCAT 实例来扩展整个应用服务集群。

当然，说到在 TOMCAT 实例中只部署一个 WAR 包这样的做法，实际上不单单只是因为扩展的因素，还涉及微服务运行期间给我们带来的第二个好处，即隔离性。

隔离性实际上是可扩展性的基础，当我们将每个微服务都隔离为独立的运行单元之后，任何一个或者多个微服务的失败都将只影响自己或者少量其他微服务，而不会大面积地波及整个服务运行体系。在架构设计上有一种实践模式，即隔板模式（Bulkhead Pattern），这种架构设计模式的首要目的就是为了隔离系统中的各个功能单元和实体，使得系统不会因为一个单元或者服务的失败而导致整体失败。这种思路在造船行业、兵工行业都有类似的应用场景。现在任何大型船舶在设计上都会有隔舱，目的就是即使有少量进水，也可以只将进水部位隔离在小范围，不会扩散而导致船舶大面积进水，从而沉没。当年泰坦尼克号虽然沉了，但不意味着他们没有做隔舱设计，只能说，伤害度已经远远超出隔舱可以提供的基础保障范围。在坦克的设计上，现在一般也会将弹药舱和乘员舱隔离，从而可以保障当坦克受创之后，将伤害尽量限定在指定区域，尽量减少对车乘成员的伤害。

前面我们提到，现在大家基本上弱化了 Java EE 的 Web 容器早期采用的“一个 Web 容器部署多个 WAR 包”的做法，转而使用“一个 Web 容器只部署一个 WAR 包”的做法，这实际上正是综合考虑了 Web 容器的设计和实现现状与真实需求之后做出的合理实践选择。这些 Web 容器内部大多通过类加载器

（Classloader）以及线程来实现一定程度上的依赖和功能隔离，但这些机制从基因上决定了这些做法不是最好的隔离手段。而进程（Process）拥有天然的隔离特性，所以，一个 WAR 包只部署运行在一个 Web 容器进程中才是最好的隔离方式。

现在回想一下，好像自从各个微服务打响独立战争并且独立之后，无论从哪个层面来看，各自“活”得都挺好。

1.3.2 多语言生态

微服务独立之后，给了对应的团队和组织快速迭代和交付的能力，同时，也给团队和组织带来了更多的灵活性，实际上，对应交付不同微服务的团队或者组织来说，现在可以基于不同的计算机语言生态构建这些微服务，如图 1-2 所示。

微服务的提供者既可以使用 Java 或者 Go 等静态语言完成微服务的开发和交付，也可以使用 Python 或者 Ruby 等动态语言完成微服务的开发和交付，对于团队内部拥有繁荣且有差异的语言文化来说，多语言生态下的微服务开发和交付将可以最大化的发挥团队和组织内部各成员的优势。当然，对于多语言生态下的微服务研发来说，有一点需要注意：为了让服务的访问者可以用统一的接口访问所有这些用不同语言开发和交互的微服务，应该尽量统一微服务的服务接口和协议。

在微服务的生态下，互通性应该是需要重点关注的因素，没有互通，不但服务的访问者和用户无法很好地使用这些微服务，微服务和微服务之间也无法相互信赖和互助，这将大大损耗微服务研发体系带来的诸多好处，而多语言生态也会变成一种障碍和负累，而不是益处。

记得时任黑猫宅急便社长的小仓昌男在其所著的《黑猫宅急便的经营学》中提到一个故事，日本国铁曾经采用不同于国际标准的集装箱和铁路规格，然后发现货物的运输效率很低，经过考察发现，原来是货物从国际标准集装箱卸载之后，在通过日本国铁运输之前，需要先拆箱，重新装入日本国铁规格的集装箱，然后装载到日本国铁上进行运输。但是，如果日本国铁采用国际标准的集装箱规格，那么货物集装箱从远洋轮船上卸载之后就可以直接装上国铁，这将大大加快运输效率（日本，国铁改革后也证明确实如此）。日本国铁在前期采用私有方案时，只关注了自己的利益和效率，舍弃了互通，也带来了效率的低下。所以，在开发和交付微服务的时候，尤其是在多语言生态下开发和交付微

服务，我们从一开始就要将互通性作为首要考虑因素，从而不会因为执迷于某些服务或者系统的单点效率而失去了整个微服务体系的整体效率。

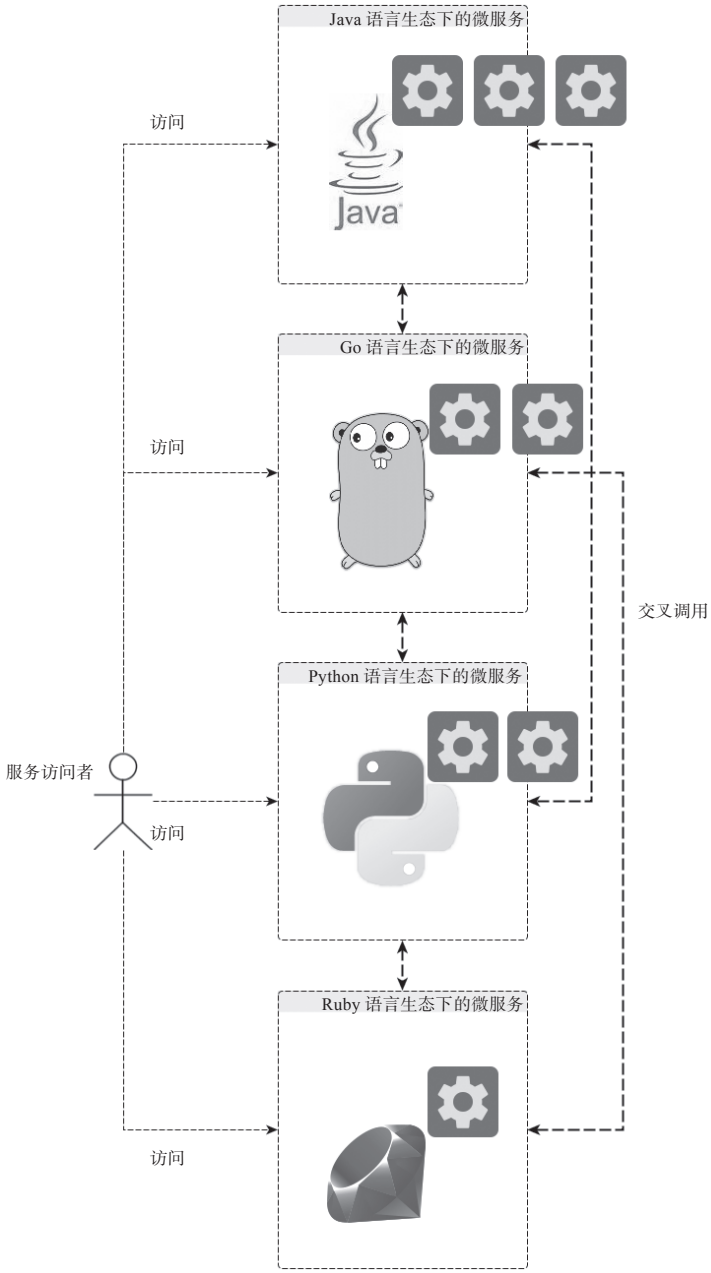

图 1-2　多语言的微服务生态

1.4 微服务会带来哪些挑战

微服务给我们带来的并非只有好处，还有相应的一些挑战。

服务“微”化之后，一个显著的特点就是服务的数量增多了。如果将软件开发和交付也作为一种生产模式看待，那么数量众多的微服务实际上就类似于传统生产线上的产品，而在传统生产模型下，为了能够高效地生产大量产品，通常采用的就是标准化生产。

比如在汽车产业，在福特 T 型车没有出来之前，大多汽车企业的生产效率都不高，而福特在引入标准化生产线之后，福特 T 型车得以大量生产并以低成本优势快速普及。

在其他行业也是同样的道理，个性化生产虽然会深得个别用户的喜欢，但生产成本通常也会很高，生产效率因为受限于个性化需求，也无法从“熟能生巧”中获益，所以，最终用户需要为生产成本和效率付出更多的溢价才能获得最终产品。而相对于个性化生产来说，标准化生产走的是另一条路，通过生产标准产品，使得整条生产链路可重复，从而提升了生产效率，可以为更广层面的用户提供大量“物美价廉”的标准产品。

微服务的研发和交付其实就类似于产品的生产链路，而数量大这一特点则决定了，我们无法通过个性化的生产模式来支撑整个微服务的交付链路和研发体系，虽然微服务化之后，我们可以投入相应的人力和团队对应各个微服务的开发和交付，可扩展性上绝对没有问题，但这不意味着现实情况下我们就能这样做，因为这些都涉及人力和资源成本，而这往往是受限的。所以，使用标准化的思路来开发和交付微服务就变成了自然而然的选择：

- 通过标准化，我们可以重复使用开发阶段打造的一系列环境和工具支持。
- 通过标准化，我们可以复用支持整个微服务交付链路的各项基础设施。
- 通过标准化，我们可以减少采购差异导致的成本上升，同时更加高效地利用硬件资源。
- 通过标准化，我们可以用标准的协议和格式来治理和维护数量庞大的微服务。

如果你还对使用标准化的思路来构建微服务体系存有疑惑，那么，不妨再

结合微服务的多语言生态特性思考一番：

- 增加一种语言生态用于微服务的开发和交付，我们是否要围绕着这种语言生态和微服务的需求重新搭建一套研发 / 测试环境？
- 我们是否还要围绕着这种语言生态打造一系列的工具来提升日常开发的效率？
- 增加一种语言生态，我们是不是还要围绕这种语言生态搭建一套针对微服务的交付链路基础设施？
- 增加一种语言生态，我们是否还要围绕它提供特定的硬件环境以及运维支撑工具和平台？

多语言生态虽然灵活度高了，不同语种和思路的团队成员也能够百花齐放了，但是不是也同样带来了以上一系列的成本？

所以，很多事情你能做，并不意味着你一定要做。适度的收缩语言生态的选择范围，并围绕主要的语言生态构建一套标准化的微服务交付体系，或许是更为合理的做法。

要实施高效可重复的标准化微服务生产，我们需要有类似传统行业生产线的基础设施。否则，高效可重复的开发和交付大量的微服务就无从谈起，所以，完备的微服务研发和交付体系基础设施建设就成为了实施微服务的终极挑战。一个公司或者组织要很好地或者说成熟地实施微服务化战略，为交付链路提供完备支撑的基础设施建设必不可少！

1.5　本章小结

在带领大家探索本书的主角 SpringBoot 微框架之前，本章首先为大家介绍了 SpringBoot 微框架服务的核心场景，即微服务。然后一起探索了微服务的概念以及由来，并探讨了微服务可以为我们带来哪些好处，以及同时又为我们带来哪些挑战。

总的来说，微服务化虽然是当下流行的趋势，但并非任何场景都合适，我们还是要审慎地在“大一统”（Monolith）服务架构和微服务架构之间做出选择，而一旦确定选择了微服务化之路，那么，就应该围绕团队和组织的主要语言生态以及微服务方向积极探索高效的微服务开发和交付模式。

SpringBoot 微框架实际上就是为 Java 语言生态而生的一种微服务最佳实践，在第 2 章中我们将从回顾 SpringBoot 的起源开始，逐步揭开 SpringBoot 微框架的神秘面纱。