

一种创建微服务的方法及相关设备

本申请要求于2022年02月14日提交中国国家知识产权局、申请号为202210134482.4、发明名称为“创建微服务的方法、装置、服务器及存储介质”的中国专利申请的优先权，以及要求于2022年06月24日提交中国国家知识产权局、申请号为202210726345.X、发明名称为“一种创建微服务的方法及相关设备”的中国专利申请的优先权，其全部内容通过引用结合在本申请中。

技术领域

本申请涉及软件开发技术领域，尤其涉及一种创建微服务的方法、系统以及计算设备集群、计算机可读存储介质、计算机程序产品。

背景技术

随着用户对软件系统的要求不断提高，业界提出了一种微服务设计理念用于开发软件系统，从而满足用户日益增加的需求。微服务设计是指将单体应用程序（如中大型软件系统）按功能或者业务需求垂直拆分成多个独立的微服务，该微服务也称作子系统，这些子系统以独立部署的进程存在，进程之间通过轻量级、跨语言的同步或异步网络调用进行通信。当用户提出新的需求时，可以增加新的子系统或修改子系统，从而满足上述新的需求，如此降低了大型复杂软件系统版本迭代的难度，且易于维护。

微服务的开发涉及众多系统性的、重复的代码开发工作，从节省微服务开发工作、提高代码规范性角度出发，开发者通常可以采用代码生成工具或基于开发平台内置的功能模块生成微服务的部分代码。例如，开发者可以使用Mybatis生成器(Mybatis Generator, MBG)生成数据操作层的代码。又例如，开发者可以使用JHipster开发平台内置的功能模块生成框架代码。

然而，已有的代码生成工具或开发平台内置的功能模块生成的代码通常是微服务的特定层的简单代码，微服务开发的灵活性较低，而且仍然需要开发者花费大量的时间和精力开发其他层的代码，影响了微服务的开发效率。

发明内容

本申请提供了一种创建微服务的方法，该方法基于领域驱动设计的理念，根据用户定义的业务模型，具体是业务模型中业务对象的元数据以及标识业务对象的关系对象的元数据生成微服务的代码，解决了代码生成工具或开发平台内置的功能模块生成特定层的简单代码的问题，提高了开发效率。本申请还提供了上述方法对应的代码管理系统、计算设备集群、计算机可读存储介质以及计算机程序产品。

第一方面，本申请提供了一种创建微服务的方法。该方法可以由代码管理系统执行。代码管理系统可以是软件系统，该软件系统可以部署在计算设备集群中，计算设备集群通过执行软件系统对应的程序代码，从而执行本申请实施例的创建微服务的方法。在一些实施例中，代码管理系统也可以是具有创建微服务功能的硬件系统，例如代码管理系统可以

是具有创建微服务功能的计算设备集群。

具体地，代码管理系统可以向用户呈现配置界面，接收用户通过所述配置界面配置的所述微服务的业务模型，其中，业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据，然后代码管理系统根据所述微服务的业务模型，生成所述微服务的代码。

该方法中，代码管理系统基于领域驱动设计的理念，根据用户定义的业务模型，具体是业务模型中业务对象的元数据以及标识业务对象的关系对象的元数据生成微服务的代码，解决了代码生成工具或开发平台内置的功能模块生成特定层的简单代码的问题，提高了开发效率。

而且，该方法支持用户灵活定义业务模型，具有较好的扩展能力，能够完成微服务开发的全链路打通，解决微服务开发早期大量的框架性开发工作，及云服务对接、部署系统对接、中间件支持、高性能高可靠场景架构设计等，为开发者提供成熟的产品级解决方案。

在一些可能的实现方式中，代码管理系统可以根据所述微服务的业务模型，生成基础模块代码和扩展模块代码。所述基础模块代码不支持所述用户修改，所述扩展模块代码支持所述用户修改，所述基础模块代码与所述扩展模块代码分离。

该方法通过将微服务的代码分为不支持用户修改的基础模块代码和支持用户修改的扩展模块代码，一方面可以实现基础模块代码重复生成，减少重复开发，提高开发效率，另一方面可以基于扩展模块代码实现定制化开发，满足个性化的需求。

在一些可能的实现方式中，所述微服务的代码包括控制器层代码、业务逻辑层代码、数据聚合层代码、数据持久操作层代码和数据库实体层代码中的至少一种。如此解决了代码生成工具或开发平台内置的功能模块生成特定层的简单代码的问题，提高了开发效率。

在一些可能的实现方式中，代码管理系统可以根据所述业务对象的元数据或所述关系对象的元数据，生成所述微服务的数据持久操作层代码，所述微服务的数据持久操作层代码包括接口代码和查询资源代码。

其中，接口代码可以是应用程序编程接口 API 的代码，查询资源代码可以是 mapper 资源的代码，代码管理系统自动生成上述代码可以实现通过 API 进行数据持久化。

在一些可能的实现方式中，所述多个业务对象的关系包括：一对多关系、多对多关系、递归关系或聚合关系中的一种或多种。该方法中，代码管理系统通过提供多种关系，以供用户对多个业务对象的关系进行定义，能够满足不同业务场景下生成相应的微服务的代码的需求。

在一些可能的实现方式中，多个业务对象包括第一业务对象和第二业务对象，所述第一业务对象和所述第二业务对象的关系为所述聚合关系。进一步地，代码管理系统还可以将所述第一业务对象和所述第二业务对象聚合得到数据传输对象，然后代码管理系统为所述数据传输对象生成查询资源代码。

如此可以满足具有复杂关系的业务对象的微服务的代码生成需求，实现自动化地创建微服务，提高微服务开发效率。

在一些可能的实现方式中，代码管理系统可以通过多种方式对业务对象进行聚合。具体地，代码管理系统可以采用嵌套模式或笛卡尔积模式，聚合所述第一业务对象和所述第

二业务对象得到所述数据传输对象。

嵌套模式是指以其中一个业务对象为基准，将另一个与之相关的业务对象列表组织到一起形成一个聚合对象。笛卡尔积模式是指将待聚合的业务对象的属性聚合到一起形成一个聚合对象。该聚合对象包括具有聚合关系的业务对象的所有属性，并且这些属性分别以业务对象的名称作为属性前缀。

该方法通过聚合业务对象为数据传输对象，基于数据传输对象生成查询资源代码，以便后续提供查询服务。需要说明的是，根据聚合模式不同，代码管理系统可以生成不同的查询方法。不同的查询方法可以通过相应的数据传输对象mapper文件进行持久化存储。

在一些可能的实现方式中，所述业务模型还包括扩展的应用程序接口 API 的元数据。

代码管理系统还可以根据所述扩展的 API 的元数据生成所述扩展的 API 的代码。如此，可以将该微服务或微服务的某些功能通过 API 提供给其他服务使用，具有较高可用性。

在一些可能的实现方式中，代码管理系统可以接收用户通过所述配置界面配置的代码生成策略。相应地，代码管理系统可以根据所述微服务的业务模型，按照所述代码生成策略，生成所述微服务的代码。

该方法中，代码管理系统支持多种代码生成策略，代码管理系统可以按照用户配置的代码生成策略生成微服务的代码，可以提高微服务开发的灵活性，提高微服务开发效率。

在一些可能的实现方式中，所述代码生成策略包括新建或继承。其中，新建可以为从零开始创建微服务，例如是创建一个新的微服务，包括新的应用的微服务或者是应用的新功能对应的微服务。继承可以为在已有微服务基础上创建微服务，例如是对某个微服务进行版本升级。

在一些可能的实现方式中，代码管理系统可以接收用户通过所述配置界面配置的组件扩展策略。相应地，代码管理系统在生成微服务的代码时，可以根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。

代码管理系统通过按照用户配置的组件扩展策略生成扩展组件的代码，如此可以开发出功能更为丰富、强大的微服务，具有较高可用性。

在一些可能的实现方式中，所述扩展组件包括鉴权组件、校验组件、探针组件（也可以称作监控组件）、标注式配置组件中的一种或多种。

其中，鉴权组件可以提供基于 SpringSecurity 的认证和鉴权能力，或者是基于角色的访问控制 RBAC 权限模型的鉴权能力，从而保障数据安全。校验组件可以提供基于 SpringSecurity 的参数校验，或者基于 Jasypt 的安全加解密加强特性，用户使用更加便捷。探针组件可以提供探针特性，用户简单配置即可实现日志系统、监控系统及调用链路系统的自动对接。探针组件也可以提供标准日志、监控、调用链路配置，在系统选择标准模板即可一键式自动分析、呈现。标注式配置组件可以提供简单易用的标注式配置方式，自动为用户生成限流、熔断、降级配置。

进一步地，扩展组件还可以包括多数据源管理组件、分库分表 sharding 组件、分布式缓存 DCS 服务组件、分布式消息服务 DMS 组件、上传下载、导入导出、定时任务组件、元数据组件。

其中，多数据源管理组件支持配置多种类型数据源，并动态切换，支持多种易用的使

用方式，用户体验非常友好。此外，多数据源管理组件可以提供异地容灾能力。多数据源组件还可以通过分布式配置中心，支持数据源的动态管理切换。并且，多数据源组件可以支持随机、轮询的负载均衡算法实现读写分离。

- 5 分库分表组件可以基于开源组件进行二次开发，解决了启动加载的性能难点，大幅缩短启动时间，优化启动性能。而且，分库分表组件对结构化查询语言 SQL 的支持进行了增强，对 SQL 查询的兼容性较好。

具体地，本申请实施例中的分库分表组件定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

- 10 其中，分库分表可以通过数据分片实现。数据分片指按照某个维度将存放在单一数据库中的数据分散地存放至多个数据库或表中，以达到提升性能瓶颈以及可用性的效果。数据分片的拆分方式又分为垂直分片和水平分片。

- 15 垂直分片，又称为纵向拆分，具体是按照业务拆分。垂直分片的经营理念是专库专用。例如，根据业务需求，将用户表和订单表垂直分片到不同的数据库。水平分片，又称为横向拆分。相对于垂直分片，它不再将数据根据业务逻辑分类，而是通过某个字段（或某几个字段），根据某种规则将数据分散至多个数据库或表中，每个分片仅包含数据的一部分。例如：根据主键分片，偶数主键的记录放入 0 库（或表），奇数主键的记录放入 1 库（或表）。水平分片从理论上突破了单机数据量处理的瓶颈，并且扩展相对自由，是分库分表的标准解决方案。

- 20 进一步地，对于同一时刻有大量并发读操作和较少写操作类型的应用系统来说，将数据库拆分为主库和从库，主库负责处理事务性的增删改操作，从库负责处理查询操作，能够有效避免由数据更新导致的行锁，使得整个系统的查询性能得到极大的改善。

- 25 与将数据根据分片键打散至各个数据节点的水平分片不同，读写分离则是根据 SQL 语义的分析，将读操作和写操作分别路由至主库与从库。读写分离的数据节点中的数据内容是一致的，而水平分片的每个数据节点的数据内容却并不相同。将水平分片和读写分离联合使用，能够更加有效的提升系统性能。

分布式缓存组件可以提供多种缓存驱动兼容。例如，该分布式缓存组件可以支持 Redis 的多种驱动兼容。并且，分布式缓存组件支持异地容灾，具有丰富的使用场景，如支持单边读写、写同端读本地等场景。

- 30 分布式消息组件可以提供消息限流、消息异步线程池处理及同步处理、消息偏移量管理、消息异步高可靠机制，以保障消息不丢失等能力。其中，消息限流可以通过在内部 consumer 拉取消息和业务处理中间加入限流组件实现。消息不丢失通过对消息偏移量 offset 进行持久化实现。首先，业务处理后，直接进行 offset 批量持久化，因为同步的原因，offset 自身就是升序的；然后异步地在内存中维护一个有序结构，业务处理后，将 offset 加到内存，通过一定机制触发持久化过程，按照 offset 升序的规则进行 offset 批量持久化。消费者
35 重平衡从数据库拉取 offset 进行指定位置消费，由此保证异步处理的场景下消息不会丢失。此外，服务端的 offset 在重平衡的时候没有使用到，但是为了尽可能让服务端和数据库内保持一致，每次 consumer 拉完消息，可以手动推一次内存中的已持久化的 offset。

在一些可能的实现方式中，代码管理系统还可以创建数据库模型。所述数据库模型包括根表对象和分库分表对象。开源组件的配置通常需要用户通过繁琐的配置实现，分库分表、读写分离的自动化配置可实现繁琐的配置简单化。

5 在一些可能的实现方式中，代码管理系统还可以生成所述微服务的代码目录，所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。

该方法通过定义代码分层规范和目录结构分层规范，该代码分层规范与目录结构分层规范一一对应，不同层次的代码放置到不同目录，形成了代码的分层，例如：控制器层、服务层、数据聚合层、数据操作层等。如此可以方便用户通过目录结构查看代码或对代码进行修改，提升用户友好性。

10 第二方面，本申请提供了一种代码管理系统。所述系统包括：

交互模块，用于向用户呈现配置界面；

通信模块，用于接收用户通过所述配置界面配置的所述微服务的业务模型，所述业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据；

15 生成模块，用于根据所述微服务的业务模型，生成所述微服务的代码。

在一些可能的实现方式中，所述生成模块具体用于：

根据所述微服务的业务模型生成基础模块代码和扩展模块代码，所述基础模块代码不支持所述用户修改，所述扩展模块代码支持所述用户修改，所述基础模块代码与所述扩展模块代码分离。

20 在一些可能的实现方式中，所述微服务的代码包括控制器层代码、业务逻辑层代码、数据聚合层代码、数据持久操作层代码和数据库实体层代码中的至少一种。

在一些可能的实现方式中，所述生成模块具体用于：

根据所述业务对象的元数据或所述关系对象的元数据，生成所述微服务的数据持久操作层代码，所述微服务的数据持久操作层代码包括接口代码和查询资源代码。

25 在一些可能的实现方式中，所述多个业务对象的关系包括：一对多关系、多对多关系、递归关系或聚合关系中的一种或多种。

在一些可能的实现方式中，所述多个业务对象包括第一业务对象和第二业务对象，所述第一业务对象和所述第二业务对象的关系为所述聚合关系，所述系统还包括：

聚合模块，用于将所述第一业务对象和所述第二业务对象聚合得到数据传输对象；

30 所述生成模块具体用于：

为所述数据传输对象生成查询资源代码。

在一些可能的实现方式中，所述聚合模块具体用于：

采用嵌套模式或笛卡尔积模式，聚合所述第一业务对象和所述第二业务对象得到所述数据传输对象。

35 在一些可能的实现方式中，所述业务模型还包括扩展的应用程序接口 API 的元数据，所述生成模块还用于：

根据所述扩展的 API 的元数据生成所述扩展的 API 的代码。

在一些可能的实现方式中，所述通信模块还用于：

接收用户通过所述配置界面配置的代码生成策略；
所述生成模块具体用于：
根据所述微服务的业务模型，按照所述代码生成策略，生成所述微服务的代码。
在一些可能的实现方式中，所述代码生成策略包括新建或继承。
5 在一些可能的实现方式中，所述通信模块还用于：
接收用户通过所述配置界面配置的组件扩展策略；
所述生成模块具体用于：
根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。
10 在一些可能的实现方式中，所述扩展组件包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。
在一些可能的实现方式中，所述系统还包括：
创建模块，用于创建数据库模型，所述数据库模型包括根表对象和分库分表对象。
在一些可能的实现方式中，所述微服务的代码包括多层代码，所述生成模块还用于：
15 生成所述微服务的代码目录，所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。
第三方面，本申请提供一种计算设备集群。所述计算设备集群包括至少一台计算设备，所述至少一台计算设备包括至少一个处理器和至少一个存储器。所述至少一个处理器、所述至少一个存储器进行相互的通信。所述至少一个处理器用于执行所述至少一个存储器中存储的指令，以使得计算设备或计算设备集群执行如第一方面或第一方面的任一种实现方式中的代码生成方法。
20 第四方面，本申请提供一种计算机可读存储介质，所述计算机可读存储介质中存储有指令，所述指令指示计算设备或计算设备集群执行上述第一方面或第一方面的任一种实现方式所述的代码生成方法。
25 第五方面，本申请提供了一种包含指令的计算机程序产品，当其在计算设备或计算设备集群上运行时，使得计算设备或计算设备集群执行上述第一方面或第一方面的任一种实现方式所述的代码生成方法。
本申请在上述各方面提供的实现方式的基础上，还可以进行进一步组合以提供更多实现方式。

附图说明

为了更清楚地说明本申请实施例的技术方法，下面将对实施例中所需使用的附图作以简单地介绍。

- 图 1 为本申请实施例提供的一种创建微服务的场景示意图；
- 35 图 2 为本申请实施例提供的一种代码管理系统的架构示意图；
- 图 3 为本申请实施例提供的一种创建微服务的方法的流程图；
- 图 4 为本申请实施例提供的一种元数据的数据文档的示意图；
- 图 5 为本申请实施例提供的一种微服务的代码的结构示意图；

- 图 6 为本申请实施例提供的一种微服务的代码的结构示意图；
图 7 为本申请实施例提供的一种数据库模型的结构示意图；
图 8 为本申请实施例提供的一种创建微服务的方法的流程图；
图 9A 至图 9H 为本申请实施例提供的一种创建微服务的方法中各步骤的界面示意图；
5 图 9I、图 9J 为本申请实施例提供的一种代码生成流程以及代码生成结果的示意图；
图 10 为本申请实施例提供的一种创建微服务的方法的流程图；
图 11A 至图 11D 为本申请实施例提供的一种创建微服务中各步骤的界面示意图；
图 12 为本申请实施例提供的一种创建微服务的方法的流程图；
图 13A 至图 13D 为本申请实施例提供的一种创建微服务中各步骤的界面示意图；
10 图 14 为本申请实施例提供的一种代码管理系统的结构示意图；
图 15 为本申请实施例提供的一种计算设备集群的结构示意图。

具体实施方式

- 本申请实施例中的术语“第一”、“第二”仅用于描述目的，而不能理解为指示或暗示相
15 对重要性或者隐含指明所指示的技术特征的数量。由此，限定有“第一”、“第二”的特征可
以明示或者隐含地包括一个或者更多个该特征。

首先对本申请实施例中所涉及到的一些技术术语进行介绍。

- 微服务设计，是一种将单一应用程序（例如中大型的软件系统）划分成一组小的服务
（也称作微服务）分别进行开发的软件开发技术。采用上述软件开发技术开发的应用程序
20 的架构称作微服务架构。在微服务架构中，微服务之间互相协调、互相配合，从而实现应
用程序的功能。其中，每个微服务运行在独立的进程中，微服务与微服务间可以采用轻量
级的通信机制互相协调或互相配合。轻量级的通信机制例如可以是基于超文本传输协议
（HyperText Transfer Protocol，HTTP）的表征性状态转移接口（RESTful application
programming interface，API）。

- 25 微服务的开发涉及众多系统性的、重复的代码开发工作，从节省微服务开发工作、提
高代码规范性角度出发，开发者在开发微服务时可以借助代码生成工具或者是开发平台内
置的功能模块，自动生成微服务的部分代码。

- 例如，开发者可以通过 MyBatis 生成器（MyBatis Generator，MBG）生成数据操作层
的代码。具体地，MBG 连接一个或多个数据库表，生成用于访问数据库表的数据操作层代
30 码。该代码包括数据库表对应的持久化对象（PO）、操作数据库的接口（Data Access Object，
DAO）或者实现增查更删（create retrieve update delete，CRUD）的结构化查询语言（Structured
Query Language，SQL）语句。又例如，开发者可以通过 JHipster 开发平台内置的功能模块
生成前后端的框架代码。

- 然而，代码生成工具或开发平台内置的功能模块生成的代码通常是微服务的特定层的
35 简单代码，仍然需要开发者花费大量的时间和精力开发其他层的代码，而且微服务开发的
灵活性较低，扩展能力比较有限，影响了微服务的开发效率。

有鉴于此，本申请实施例提供了一种创建微服务的方法。该方法可以由代码管理系统
执行。代码管理系统可以是软件系统，该软件系统可以部署在计算设备集群中，计算设备

集群通过执行软件系统对应的程序代码，从而执行本申请实施例的创建微服务的方法。在一些实施例中，代码管理系统也可以是具有创建微服务功能的硬件系统，例如代码管理系统可以是具有创建微服务功能的计算设备集群。为了便于理解，下文以代码管理系统为软件系统进行示例说明。

5 具体地，代码管理系统可以向用户呈现配置界面，接收用户通过配置界面配置的微服务的业务模型，该业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据，然后代码管理系统根据微服务的业务模型，生成微服务的代码。

该方法中，代码管理系统基于领域驱动设计（Domain Driven Design, DDD）的理念，根据用户定义的业务模型，具体是业务模型中业务对象的元数据以及标识业务对象的关系对象的元数据生成微服务的代码，解决了代码生成工具或开发平台内置的功能模块生成特定层的简单代码的问题，提高了开发效率。

而且，该方法支持用户灵活定义业务模型，具有较好的扩展能力，能够完成微服务开发的全链路打通，解决微服务开发早期大量的框架性开发工作，及云服务对接、部署系统对接、中间件支持、高性能高可靠场景架构设计等，为开发者提供成熟的产品级解决方案。

15 本申请实施例的创建微服务的方法可以应用于对已有的微服务进行改造的场景，也可以应用于创建新的微服务的场景。参见图 1 所示的创建微服务的场景示意图，代码管理系统在创建新的微服务时，可以基于用户配置的业务对象的元数据开始正向设计。代码管理系统在对已有的微服务进行改造，例如是对已有的微服务进行迁移或重构时，可以是复用已有接口和已有数据进行改造。其中，已有数据可以是数据定义语言 (Data Definition Language, DDL) 语句，该 DDL 语句用于描述数据库中要存储的实体。

接下来，结合附图对本申请实施例的代码管理系统的系统架构进行介绍。

参见图 2 所示的代码管理系统的架构图，代码管理系统 20 处于整个应用程序开发框架的上层，代码管理系统 20 可以与应用程序开发框架中的微服务框架 30 对接，微服务框架 25 30 与底层的运行环境 40 对接。

其中，代码管理系统 20 包括代码生成装置 22，代码生成装置 22 用于基于元数据，例如业务模型中业务对象的元数据以及关系对象的元数据，生成微服务的前端代码和后端代码。进一步地，代码管理系统 20 还可以包括代码管理装置 24。代码管理装置 24 用于对代码进行持续集成持续交付（Continuous Integration Continuous Delivery, CICD）。具体地，30 代码管理装置 24 用于生成对代码进行检查、构建、部署和测试的流水线，从而实现微服务的构建、部署、测试。代码管理装置 24 还用于对微服务进行治理、运维。

微服务框架 30 用于提供不同组件或特性、能力以辅助微服务开放。如图 2 所示，微服务框架 30 包括开发框架和开发框架扩展组件。其中，开发框架可以是 Spring Cloud 开发框架，记作 Spring Cloud DevSpore，开发框架扩展组件可以是 Spring Cloud 开发框架扩展组件，记作 DevSpore Component。在该示例中，Spring Cloud DevSpore 依赖云服务，DevSpore Component 可以不依赖云服务。

SpringCloud DevSpore 可以提供一系列高性能、高可靠性、高可用性的特性，包括但不限于分库分表、读写分离、分布式事务、关系/树/本地缓存、文档数据库、工作流、异步调

用、事件模型、多数据源管理、缓存增强、数据库存取、消息中间件软件开发工具 (software development kit, SDK) 封装、分布式锁、对象/文件存储。其中, 多数据源管理可以进一步分为关系型数据库服务 (relational database service, RDS) 多数据源管理和分布式缓存服务 (Distributed Cache Service, DCS) 多数据源管理。其中, 一些特性是基于元数据 (metadata) 实现的。DevSpore Component 可以提供一系列简单易用的功能或特性, 包括但不限于表格导入导出、上传下载、定时任务、附件、邮件、异步任务。

微服务框架 30 还可以包括服务治理层、开发框架基础组件层、原子依赖组件层。其中, 服务治理层也可以称作 SpringCloud 层, 用于提供服务治理所需的基本特性, 如配置发现、配置中心、灰度路由、流量治理、本地服务治理中的一种或多种。开发框架基础组件层也称作 SpringBoot DevSpore 层, 用于提供一系列基础功能, 包括但不限于认证鉴权、安全编码、部署、审计、数据库/缓存模拟、国际化、性能监控/告警。原子依赖组件层也称作 SpringBoot 层, 该层可以包括 spring 框架依赖、日志框架、编解码、注解工具、语言增强、API Swagger、通讯框架、安全套件、覆盖率检查、单元测试 (如 junit) 中的一种或多种。

运行环境 40 包括基础设施, 该基础设施可以是实现计算、存储的基础中间件。在一些实施例中, 基础设施包括 RDS、文档数据库服务 (document database service, DDS)、弹性文件服务 (scalable file service, SFS)、分布式消息服务 (distributed message service, DMS)、DCS、云容器引擎 (cloud container engine, CCE)、应用运维管理 (application operations management, AOM)、云日志服务 (log tank service, LTS)、应用性能管理 (application performance management, APM) 中的一种或多种。

接下来, 从代码管理系统 20 的角度对本申请实施例的创建微服务的方法进行介绍。

参见图 3 所示的创建微服务的方法的流程图, 该方法包括:

S302: 代码管理系统 20 向用户呈现配置界面。

配置界面具体为支持用户配置待创建的微服务的相关属性的界面。该配置界面可以是图形化的用户界面 (graphical user interface, GUI) 或者是命令用户界面 (command user interface, CUI)。

进一步地, 待创建的微服务可以为从零开始创建的微服务 (新的微服务) 或者是基于已有的微服务进行改造的微服务。其中, 代码管理系统 20 在创建新的微服务时所呈现的配置界面可以不同于代码管理系统 20 对已有的微服务进行改造时所呈现的配置界面。代码管理系统 20 在创建新的微服务时所呈现的配置界面用于支持用户配置新的微服务的属性, 代码管理系统 20 在对已有的微服务进行改造时所呈现的配置界面用于支持用户配置已有的微服务可复用的属性。

S304: 代码管理系统 20 接收用户通过所述配置界面配置的所述微服务的业务模型。

一个微服务可以包括众多的业务逻辑。基于领域驱动设计 DDD 的理念, 业务逻辑可以抽象成一系列业务对象及组合操作。其中, 上述业务逻辑可以分为领域逻辑和非领域逻辑。领域逻辑可以包括业务对象的新增和修改, 由领域驱动且不易变, 非领域逻辑包括业务对象的查询和删除, 由数据驱动且易变。其中, DDD 是一种通过将实现连接到持续进化的领域模型来满足复杂需求的软件开发方法。DDD 的前提是: 将项目的主要重点放在核心领域

(core domain) 和领域逻辑;把复杂的设计放在有界域(bounded context)的领域模型上;让技术人员以及领域专家合作,以迭代方式来完善特定领域问题的领域模型。领域模型是对业务模型的抽象,DDD是把业务模型翻译成系统架构设计的一种方式。

5 基于上述业务逻辑,用户可以配置微服务的业务模型。业务模型包括多个业务对象(business object, BO)的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据。其中,业务对象是指业务涉及实体对应的对象。业务涉及实体可以包括人、物或者事件,例如电商业务涉及实体可以包括订单、购物车、人、商品等。业务对象的元数据包括业务对象的属性,如业务对象的标识、业务对象的类型、主键类型(primaryKeyType)。其中,业务对象的标识具有唯一性,例如可以是业务对象的编号或者业务对象的名称。

10 多个业务对象之间具有关联关系,标识多个业务对象之间关系的对象即为关系对象。关系对象的元数据包括关系对象的标识、关系对象的类型、关系对象关联的业务对象的名称。关系对象的标识具有唯一性,例如可以为关系对象的编号或者关系对象的名称。关系对象的类型可以包括一对多关系、多对多关系、父子关系(也称作递归关系)或者聚合关系。

15 一对多关系是指一个第一业务对象对应多个第二业务对象的关系。例如,“部门”和“员工”之间的关联关系具体为一对多关系。一对多关系所关联的业务对象的生命周期是完全解耦的,一对多关系的关系对象在两种业务对象被创建后才能创建,关系对象被删除代表两个业务对象特定关系的结束,不影响任何一个业务对象的生命周期。仍以“部门”和“员工”为例,“员工”可以从一个“部门”转移到另一个“部门”,或者一个“部门”被撤销,20 “部门”中的员工仍然在企业,上述情况均可以导致“部门”和“员工”的关系对象被删除。

多对多关系是指多个第一业务对象对应多个第二业务对象的关系。例如,“老师”和“学生”之间的关联关系具体为多对多关系。与一对多关系类似,多对多关系所关联的业务对象的生命周期也是完全解耦的。多对多关系的关系对象通常在两种业务对象被创建后才能25 创建,该关系对象被删除不影响任何一个业务对象的生命周期。关系对象可以存储在对立的关系表中。关系表中除了有分别指向两个业务对象的外键外,还可以包括其它扩展字段。该扩展字段包括但不限于关系的失效时间,关系的权重等。

父子关系是指作为父对象的业务对象和作为子对象的业务对象之间的关系。其中,父对象和子对象为相同类型的对象,父对象和子对象可以存储在同一张数据库表中,子对象的一个字段指向父对象的标识(identifier, ID)。没有父对象的业务对象为根对象,根对象及其子孙对象构成一棵树的关系。在一些场景下,父子关系本身有可能需要独立的关系对象来表示,如需要标注关系的创建时间、失效时间时,可以创建关系对象。关系对象在数据库30 中对应该数据库表,数据库表中父字段和子字段分表指向业务对象中的记录。

聚合关系是指作为根对象的业务对象和作为内部对象(即内部业务对象)的业务对象的关系。根对象通常可以是一个实体 entity,内部对象可以是值对象 value object。根对象和35 内部对象通常可以存储在独立的数据库表中。一个根对象也可以对应多个内部对象,但内部对象的 ID 在根部对象范围内具有意义,也即对内部对象的操作通常需要先获得根对象的 ID,并且内部对象是在根对象被创建后才能创建,若根对象被删除或者释放,内部对象的

生命周期也随之结束。为了便于理解，本申请实施例提供了一个示例。该示例中，根对象可以为“公寓”，内部对象可以为“窗户”。“公寓”具有全局唯一的 ID，“公寓”的 ID 可以根据地址和房间号生成，基于此，根据上述 ID 可以查找某一“公寓”，而“窗户”通常需要在找到房间后才能引用，“公寓”与“窗户”即属于聚合关系。

- 5 在对已有的微服务进行改造的场景中，用户还可以复用已有的微服务的业务模型，例如可以配置业务模型的注解，并作用于代码中，如此，代码管理系统 20 可以根据用户配置的注解逆向生成业务对象的元数据或关系对象的元数据。

图 4 示出了一种元数据的数据文档的示意图，如图 4 所示，业务模型中包括业务对象的元数据，业务对象的元数据包括业务对象定义，业务对象定义可以包括对象字段定义（记作 fields），对象字段定义用于表征该业务对象的属性，如业务对象的 ID、名称、类型、主键类型等等。该业务对象可以为内部对象（记作 Internal BOs）。针对内部对象，业务模型还包括该内部对象的外部对象（即外部业务对象）引用（记作 External BOs reference）。外部对象引用可以是其他微服务的业务对象引用。

- 15 进一步地，业务模型中还包括关系对象（记作 relation objects, Ros）的元数据，关系对象的元数据包括关系对象定义，关系对象定义包括关系对象的属性，如关系对象的 ID、关系对象的类型、关系对象关联的业务对象的 ID。

S306: 代码管理系统 20 根据所述微服务的业务模型，生成所述微服务的代码。

- 20 传统应用开发一般采用模型-视图-控制器（Model-View-Controller, MVC 模式，MVC 模式是软件工程中的一种软件架构模式，该模式下，软件系统分为三个基本部分：模型、视图和控制器。其中，模型可以是技术人员（如程序员）编写程序应有的功能、数据库专家进行数据管理和数据库设计得到。视图可以是界面设计人员进行图形界面设计得到。控制器用于负责转发请求，对请求进行处理。

- 25 在 DDD 理念中，MVC 模式有了进一步的区分。具体地，微服务的代码可以分为如下多层：控制器（记作 controller）层、业务逻辑（记作 controller）层、数据聚合（记作 repository，也称为 DAO）层、数据持久操作（记作 mapper）层、数据库实体（记作 entity）层。其中，entity 层也可以称作模型（记作 model）层。

- 30 controller 层用于业务流程的控制。具体实现时，controller 层可以调用 service 层提供的接口来控制业务流程。针对具体的业务流程，controller 层可以包括不同的控制器。service 层用于实现业务模块的应用逻辑，service 层可以包括用于业务处理的接口以及接口的实现类。其中，配置文件中可以配置接口及实现类的关联关系，从而使得 controller 可以调用接口进行业务处理。repository 层用于进行数据聚合操作，具体是链接 service 层和 mapper 层的数据聚合操作。mapper 层用于实现数据持久化，负责与数据库进行联络的一些任务可以封装在 mapper 层。entity 层，也被称为 model 层、pojo 层。一般情况下，一个实体类可以对应一张数据库表，该实体类的属性和数据库表中的字段一一对应。

- 35 基于此，代码管理系统 20 可以根据所述微服务的业务模型，生成 controller 层代码、service 层代码、repository 层代码、mapper 层代码和 entity 层代码中的至少一种。其中，controller 层可以提供多种接入方式，例如是表述性状态传递（Representational State Transfer, REST）或者是远程过程调用（Remote Procedure Call, RPC），以供使用。

参见图 5 所示的微服务的代码生成示意图, 代码管理系统 20 可以根据元数据生成 controller 层的代码, 如 controller。此外, 代码管理系统 20 还支持生成 controller 层的扩展类, 以满足用户的特殊要求。其中, 该扩展类也称作用户扩展。controller 层还可以提供用法 (usage) 给 service 层。代码管理系统 20 可以自动生成 service 层的代码, 如 service 层的用于业务处理的接口。此外, 用户的特殊业务逻辑可以通过 service 层的扩展类 (可以通过继承得到, 也称作继承扩展) 实现。service 层可以提供 usage 至 Repository 层。代码管理系统 20 可以自动生成 Repository 层的代码。其中, Repository 层为抽象的数据接口层, 代码管理系统 20 通过一致化的接口封装底层实际的数据库类型以及对象关系映射 (Object Relational mapping, O/R mapping) 实现。例如, 对于 MongoDB 等非关系数据库 (non-relational SQL, NoSQL) 和 MySQL、PostgreSQL 等关系型数据库, 代码管理系统 20 可以提供统一的 Spring Data 风格的数据访问接口, 该接口封装底层实际的数据库类型以及 O/R mapping 实现。进一步地, 对于关系型数据库, 代码管理系统 20 可以封装符合 JPA 标准的 O/R mapping 接口以及 iBatis 风格的 O/R mapping 接口的差异。此外, 代码管理系统 20 可以通过继承方式生成 Repository 层的扩展类, 即继承扩展, 或者通过新添方式生成 Repository 的扩展类, 即添加扩展。Repository 层还可以提供 usage 至 mapper 层。mapper 层可以为 O/R mapping 层, 通过面向对象的对象模型 (如业务模型) 和数据库 (如关系型数据库) 的数据结构之间的相互转换, 从而实现数据持久化。代码管理系统 20 可以根据上述 O/R mapping 自动生成 mapper 层的代码, 进一步地, 代码管理系统 20 还可以生成 mapper 层的扩展类, 该扩展类为用户扩展。上述扩展类 (如扩展的 API) 的代码可以根据扩展的 API 的元数据生成。

考虑到一些代码可以重复生成, 代码管理系统 20 可以根据微服务的业务模型生成基础 (base) 模块代码和扩展模块代码。其中, 扩展模块代码也称作 service 模块代码。基础模块代码可以是每次编译时自动生成, 不支持所述用户修改, 通常为固定模式的代码。service 模块代码支持所述用户修改, 例如支持用户修改或增加业务逻辑, 通常不会重复生成。代码管理系统 20 通过将 base 模块代码 (如图 5 中的自动生成代码) 与 service 模块代码 (如图 5 中的继承扩展、用户扩展) 分离, 可以实现重复生成代码, 提高代码生成效率。

在本实施例中, base 模块代码定义一系列的接口以及对应的默认实现, service 模块代码可继承、扩展以覆盖或扩展默认实现, 达到扩展业务逻辑的目标。

如图 6 所示, base 模块代码中的 service 层定义了用于业务处理的接口, 并提供默认的实现代码, service 模块代码中的 service 层扩展实现了该接口, 可实现业务逻辑编写。在扩展实现该接口时, 用户可以修改现有方法如默认方法以实现业务逻辑, 无需增加新方法。

类似地, base 模块代码中的 repository 层定义了 repository 抽象类, service 模块代码中的 repository 层定义了 repository 实现类, 用户可以增加新方法实现业务逻辑, 或者是对已有方法进行修改, 从而实现业务逻辑。base 模块代码中的 mapper 层定义了自动实现的 mapper 接口 (接口代码) 和 mapper 资源 (也称作查询资源代码), service 模块代码中的 mapper 层定义了扩展 mapper 接口。其中, 扩展 mapper 接口初始为空接口, 用户可以通过添加新方法增加数据库操作。

代码管理系统 20 根据业务对象的元数据或关系对象的元数据可以生成规范化的接口代码 (API) 和查询资源代码 (mapper 资源)。其中, API 以 REST 方式接入时, 该 API 符

合 RestAPI 命名及格式规范。

由于微服务的 API 可以用于对业务对象进行操作，因此，微服务的服务路径（例如是 API 路径）可以通过服务版本路径和业务对象路径表示。在一些实施例中，服务路径可以表示为：/{服务版本路径}/{业务对象路径}。其中，服务版本路径可以为/{小写服务名}/{服务版本号}。例如：微服务 uct 的服务版本路径可以为/uct/v1，业务对象的路径通常可以为/{对象类型名}s。操作业务对象的代码可以包括操作关键字和服务路径。

下面以 uct 服务 v1 版本接口的 project 对象为例进行说明。具体地，微服务的 API 可以用于获取对象列表，获取对象列表的代码可以为 GET /uct/v1/projects? 查询条件。新增业务对象的代码可以为 POST /uct/v1/projects，删除业务对象的代码可以为 DELETE /uct/v1/projects/{project_id}，修改业务对象的代码可以为：PUT /uct/v1/projects/{project_id}，获取业务对象的代码可以为：GET /uct/v1/projects/{project_id}。

在一些可能的实现方式中，微服务的 API 还支持对业务对象的批量操作，如批量新增业务对象、批量删除业务对象、批量修改业务对象。其中，批量新增业务对象的代码可以为 POST /uct/v1/projects/batch-create，批量删除业务对象的代码可以为 POST /uct/v1/projects/batch-delete，批量修改业务对象的代码可以为 POST /uct/v1/projects/batch-update。

当操作的业务对象为内部对象时，通常需要先找到根对象。基于此，微服务中用于操作内部对象的 API 的服务路径可以表示为/{服务版本路径}/{根对象类型名}s/{根对象的 ID}/{BO}s。相应地，操作内部对象的代码可以包括操作关键字和上述携带有根对象的 ID 的服务路径。

下面以 uct 服务 v1 版本接口的 privilege 对象为例进行说明。获取内部对象列表的代码可以为 GET /uct/v1/roles/{role_id}/privileges。新增内部对象的代码可以为 POST /uct/v1/roles/{role_id}/privileges，删除内部对象的代码可以为 DELETE /uct/v1/roles/{role_id}/privileges/{privilege_id}，修改内部对象的代码可以为：PUT /uct/v1/roles/{role_id}/privileges/{privilege_id}，获取业务对象的代码可以为：GET /uct/v1/roles/{role_id}/privileges/{privilege_id}。

相应地，批量新增内部对象的代码可以为 POST /uct/v1/roles/{role_id}/privileges/batch-create，批量删除业务对象的代码可以为 POST /uct/v1/roles/{role_id}/privileges/batch-delete，批量修改业务对象的代码可以为 POST /uct/v1/roles/{role_id}/privileges/batch-update。

以上描述了独立的业务对象（如基本业务对象）以及具有聚合关系的业务对象的操作方法，下面对具有一对多关系的业务对象以及具有多对多关系的业务对象的操作方法进行说明。

具体地，具有一对多关系的业务对象的操作路径通常是独立的，也即不存在路径依赖，因而与具有聚合关系的业务对象相比，对具有一对多关系的业务对象的操作不会在 {secondary} 对应的表（table）中创建外键 foreign key。与独立的业务对象相比，对具有一对多关系的业务对象的操作会在 {secondary} 对应的 table 中自动增加命名为 {secondary}_id 的一列。

需要说明的是，根对象（Root BO）与分表对象（Sharding BO）之间若未定义关系，则代码管理系统 20 可以自动添加一对多关系；若定义有聚合关系或一对多关系，则不自动添加关系。

与一对多关系类似，具有多对多关系的业务对象的操作路径通常是独立的，一般在关系操作中体现依赖关系，用于操作上述业务对象的 API 的服务路径可以表示为/{服务版本路径}/{primary name}-{secondary name}-relations。

为了便于理解，以用户组 UserGroup 和用户 User 进行示例说明。批量新建关系的代码可以为 POST /uct/v1/user-group-user-relations，批量删除关系的代码可以为 DELETE /uct/v1/user-group-user-relations，批量修改关系的代码可以为 PUT /uct/v1/user-group-user-relations，查询关系的代码可以为 GET /uct/v1/user-group-user-relations，查询关联到特定用户组的所有用户的代码可以为 GET /uct/v1/user-groups/{id}/users，查询关联到特定用户的所有用户组的代码可以为 GET /uct/v1/users/{id}/user-groups。

除了一对多关系、多对多关系、聚合关系，业务对象之间的关系也可以是递归关系。其中，业务对象之间的递归关系可以使用关系对象 RO 维护，也可以通过在对象中增加父节点标识 parentID 字段进行维护。

当业务对象包括 parentID 字段时，也即不使用关系对象进行递归关系维护时，用于新增业务对象、删除业务对象或修改业务对象的 API 可以和前述用于新增、删除或修改独立的业务对象的 API 的代码一致。

进一步地，业务对象之间基于 parentID 实现了树关系维护。微服务可以提供用于关系查询的 API。例如，获取父节点的代码可以为 GET ../{BO type}s/{id}/parents，该 API 的返回值为父节点的标识 parentID。获取直接子节点的代码可以为 GET ../{BO type}s/{id}/children，该 API 的返回值为所有子节点的 ID。获取所有后代节点的代码可以为 GET ../{BO type}s/{id}/descendants，该 API 的返回值为所有后代节点的 ID，其中，后代节点也可以称作子孙节点，包括子节点以及子节点的子节点。获取所有祖先节点的代码可以为 GET ../{BO type}s/{id}/ancestors，该返回值可以为业务对象标识数组，记作 array<BusinessObject>，该数组包括所有祖先节点的 ID，其中，祖先节点包括父节点以及父节点的父节点，层次越高的节点在数组中的位置越靠前。

当业务对象使用关系对象维护业务对象之间的递归关系时，用于新增、删除或修改递归关系的 API 可以通过关系表实现，该 API 的代码可以为 /{服务版本路径}/{bo name}-relations。

为了便于理解，以英文对象 role 进行示例说明。查询关系的代码可以为 GET /uct/v1/role-relations?parent_id=xxx&child_id=xxx，批量新增关系的代码可以为 POST /uct/v1/role-relations，批量修改关系的代码可以为 PUT /uct/v1/role-relations，批量删除关系的代码可以为 DELETE /uct/v1/role-relations。

以上对代码管理系统 20 生成 API 的代码进行了介绍。进一步地，代码管理系统 20 还可以生成 API 的实现方法的代码，也称作查询资源代码。在本申请实施例，代码管理系统 20 不仅可以生成根据业务模型中的元数据生成持久化操作层中对数据库进行简单操作的方法的代码，还可以生成根据业务模型中的元数据生成持久化操作层中对数据库进行复杂操作的

方法的代码。其中，简单操作包括但不限于新增、删除、查询或者修改业务对象，或者新增、删除、查询或者修改关系。复杂操作包括但不限于关联操作，如查询关联到特定用户组的所有用户等复杂查询。

下面分别对不同操作的方法的代码进行介绍。

- 5 第一种情况，代码管理系统 20 为业务对象 BO 生成基本的 CRUD 操作以及批量操作的方法。

为了便于理解，以业务对象"Person"示例说明。该示例中，"Person"的定义可以为：

```
{
  "service": {
10    "name": "TestDemo",
    "packageName": "com.xx.cloud", //xx 具体为云平台的名称
    "groupId": "com.xx.tools" // xx 具体为云平台的名称
    ...
  }
15  ...
  "model": {
    "objects": {
      "fixedFields": [
20        {
          "name": "creator"
        },
        ...
      ],
      "businessObjects": [
25        {
          "name": "Person",
          "chinessName": "",
          "type": "PERSIST",
          "dbVender": "MYSQL",
30          "primaryKeyType": "AUTO_INCREASE_NUMBER",
          ...
          "fields": [
            {...},
            ...
35          ]
        },
        ...
      ]
    }
40  },
  "generatorPolicy": {
    "mapperPackageName": "testmapper",
    ...
  }
}
```

```
}  
}
```

代码管理系统20可以通过DevSpore-codegen, 为"Person"业务对象分别生成接口代码和查询资源代码, 其中, 接口代码在dao接口文件中持久化存储, 该文件的名称为
5 PersonMapper.java, 查询资源代码在sql资源文件中持久化存储, 该sql资源文件的名称为PersonMapper.xml。

参见图4, 代码管理系统20获取的元数据文档中还包括服务级元数据, 该服务级元数据中的字段用于表示微服务的全局属性, 例如服务级元数据可以包括微服务的名称(记作“name”)、包名称(记作“packageName”)。进一步地, 元数据文档中还可以包括用户通过所述配置界面配置的代码生成策略(记作“generatorPolicy”), 代码生成策略中定义
10 查询资源文件包名称(记作“mapperPackageName”)。代码管理系统20可以根据元数据文档中微服务的名称、包名称以及查询资源文件包名称生成dao接口文件的包名。该包名可以表示为{service.packageName}.{generatorPolicy.mapperPackageName}。其中, dao接口文件的包名可以映射到文件系统, 获得目录结构。dao接口文件如PersonMapper.java文件可以存储
15 在该目录结构下。

需要说明的是, 当元数据文档中未配置“mapperPackageName”, 则代码管理系统20可以基于默认查询资源文件包名如“mapper”生成dao接口文件的包名。例如, 元数据文档中未配置“mapperPackageName”时, PersonMapper.java文件的包名可以为
com.xxx.cloud.testdemo.mapper.PersonMapper.java。当元数据文档中配置
20 “mapperPackageName”, 则代码管理系统20基于配置好的“mapperPackageName”生成dao接口文件的包名。仍以业务对象“Person”为例, 元数据文档中“mapperPackageName”被配置为“testmapper”, 相应地, PersonMapper.java文件的包名可以为
com.xxx.cloud.testdemo.testmapper.PersonMapper.java。

查询资源文件默认生成位置可以为classpath:/mapper/。例如PersonMapper.xml文件的生成位置可以为 TestDemo /base/src/main/resources/mapper/PersonMapper.xml。需要说明的是,
25 查询资源文件的生成位置通常不支持配置。

代码管理系统20为业务对象BO生成新增业务对象的方法可以包括两种场景, 下面分别对这两种场景进行介绍。

一种场景为直接新增业务对象。具体地, 代码管理系统20可以插入一条BO对象, 返回
30 插入行数, 成功为1, 失败为0, 具体如下所示:

```
Integer insert({Bo} {bo});
```

以业务对象“Person”为例, 直接新增业务对象的方法可以为Integer insert(Person person)。

另一种场景为根据条件新增业务对象。具体地, 代码管理系统20可以插入一条BO对象, 对象中没有的属性则不添加, 返回插入行数, 成功为1, 失败为0, 具体如下所示:

```
Integer insertSelective({Bo} {bo});
```

以业务对象“Person”为例, 根据条件新增业务对象的方法可以为Integer insertSelective(Person person)。

进一步地, 代码管理系统20可以批量新增业务对象。具体地, 批量新增业务对象的操作具有原子性, 也即操作结果为全部成功或者全部失败。当全部成功时, 返回插入行数,

全部失败时，返回0，具体如下所示：

```
Integer insertBatch(List<{Bo}> {bo}List);
```

以业务对象“Person”为例，批量新增业务对象的方法可以为 Integer insertBatch(List<Person> personList)。

- 5 代码管理系统20可以使用主键为业务对象BO生成删除业务对象的方法，该方法返回删除行数，成功为1，失败为0，具体如下所示：

```
Integer deleteByPrimaryKey({Bo.primaryKeyType} id);
```

以业务对象“Person”为例，删除该业务对象的方法可以为 Integer deleteByPrimaryKey(Long id)。

- 10 当需要批量删除业务对象时，代码管理系统20也可以使用主键列表批量删除业务对象，此操作具有原子性，若全部成功则返回删除行数。需要说明，若传入不存在的主键不会报错，也不会统计在返回个数中。该方法具体如下所示：

```
Integer deleteByPrimaryKeys(List<{Bo.primaryKeyType}> ids);
```

- 15 以业务对象“Person”为例，批量删除该业务对象的方法可以为 Integer deleteByPrimaryKeys(List<Long> ids);

代码管理系统20可以为业务对象生成更新业务对象的方法，若成功返回1，若失败返回0，具体如下所示：

```
Integer update({Bo} {bo});
```

以业务对象“Person”为例，更新该业务对象的方法可以为 Integer update(Person person)。

- 20 代码管理系统20也可以批量更新业务对象。该操作具备原子性，操作结果为全部成功或者全部失败，若成功返回更新行数，若失败返回0，具体如下所示：

```
Integer updateBatch(List<Bo> boList);
```

以业务对象“Person”为例，批量更新该业务对象的方法可以为 Integer updateBatch(List<Person> personList)。

- 25 代码管理系统20还可以部分更新业务对象。更新的业务对象中包括没有值的属性时则不更新该属性。更新基准是以主键为条件，若更新成功返回1，若失败或无符合条件的记录返回0，具体如下所示：

```
Integer updateSelective({Bo} {bo});
```

- 30 以业务对象“Person”为例，部分更新该业务对象的方法可以为 Integer updateSelective(Person person)。

代码管理系统20还可以使用主键获取业务对象，如果为空则返回null，具体如下所示：

```
{Bo} selectByPrimaryKey({Bo.primaryKeyType} id);
```

以业务对象“Person”为例，获取该业务对象的方法可以为 Person selectByPrimaryKey(Long id)。

- 35 进一步地，代码管理系统20可以使用主键列表获取业务对象列表，返回主键实际存在的业务对象数量，若全不存在时则返回空列表，具体如下所示：

```
List<{Bo}> selectByPrimaryKeys(List<{Bo.primaryKeyType}> ids);
```

以业务对象“Person”为例，获取业务对象列表的方法可以为 List<Person>

selectByPrimaryKeys(List<Long> ids)。

代码管理系统20还可以根据查询条件查询业务对象个数，该方法如下所示：

```
Long countByExample({Bo}Criteria {bo}Criteria);
```

以业务对象“Person”为例，根据查询条件查询业务对象个数的方法可以为Long
5 countByExample(PersonCriteria personCriteria)。

代码管理系统20还可以根据查询条件查询业务对象主键列表，若无符合条件的业务对象时返回空列表，具体如下所示：

```
List<{Bo.primaryKeyType}> selectIDsByExample({Bo}Criteria {bo}Criteria);
```

如果只需要业务对象的主键ID,则调用此函数可能只需要读取覆盖索引就可以得到，不需要读取表记录，如此性能相比selectByExample有显著提升，用户可以优先使用
10 selectIDsByExample。

仍以业务对象“Person”为例，根据查询条件查询业务对象主键列表的方法可以为List<Long> selectIDsByExample(PersonCriteria personCriteria)。

类似地，代码管理系统还可以根据查询条件查询业务对象列表，若没有命中的业务对象则返回空列表，该方法具体为：
15

```
List<{Bo}>selectByExample({Bo}Criteria{bo}Criteria);
```

以业务对象“Person”为例，根据查询条件查询业务对象列表的方法可以为List<Person> selectByExample(PersonCriteria personCriteria)。

第二种情况，业务对象包括第一业务对象和第二业务对象，该第一业务对象与第二业务对象具有聚合关系，代码管理系统20可以根据上述第一业务对象和第二业务对象生成如图4所示的数据传输对象（data transfer object，DTO）。代码管理系统20可以为DTO生成查询方法。
20

其中，代码管理系统20可以将所述第一业务对象和所述第二业务对象聚合得到数据传输对象DAT。在聚合第一业务对象和第二业务对象时，代码管理系统20可以根据需求选择相应的模式。例如，代码管理系统20可以采用嵌套（nested）模式或笛卡尔积(Cartesian)模式，聚合所述第一业务对象和所述第二业务对象得到所述数据传输对象。
25

Nested模式是指以其中一个业务对象为基准，将另一个与之相关的业务对象列表组织到一起形成一个聚合对象。例如，User<-m2m->Role的聚合对象UserRoleNested可以为：

```
UserRoleNested:{  
30     User user;    List<Role> roles;  
}
```

Cartesian模式是指将待聚合的业务对象的属性聚合到一起形成一个聚合对象。该聚合对象包括具有聚合关系的业务对象的所有属性，并且这些属性分别以业务对象的名称作为属性前缀。例如，User<-m2m->Role的聚合对象UserRoleCartesian可以为：

```
UserRoleCartesian{  
35     userId;  
     userName;  
     userCreator;
```

```

        user...
        roleId;
        roleName;
        role...
5      }

```

根据聚合模式不同，代码管理系统20可以生成不同的查询方法。不同的查询方法可以通过相应的DTO mapper文件进行持久化存储。

当聚合模式为Nested模式时，代码管理系统20在查询时可以从关系双方的BoMapper中查找到符合条件的记录，然后将结果设置到聚合对象中。以查询UserRoleNested为例，该方法可以为：

```

10      UserRoleNested userRoleNested = new UserRoleNested();
      User user = userMapper.selectByPrimaryKey(userId);
      List<Role> roles = roleMapper.getRolesByUserId(userId);
      userRoleNested.setUser(user);
15     userRoleNested.setRole(roles);

```

当聚合模式为 Cartesian模式时，代码管理系统20可以根据其中一个业务对象的主键如 primary主键或者secondary主键直接查询Cartesian模式的聚合对象。下面分别对根据primary主键直接查询Cartesian模式的聚合对象以及根据secondary主键直接查询Cartesian模式的聚合对象进行说明。

根据primary主键查询聚合对象列表的方法可以为：

```

      List<{Primary}{Secondary}Cartesian> get{Primary}{Secondary}CartesianBy {Primary}Id
      ({Primary.keyType} {Primary}Id);

```

以查询UserRoleNested为例，第一业务对象为“user”，第二业务对象为“Role”，则根据 primary 主键查询聚合对象列表的方法可以为 List<UserRoleCartesian> getUserRoleCartesianByUserId(Long userId)。

根据Secondary主键查询聚合对象列表的方法可以为：

```

      List<{Primary}{Secondary}Cartesian>get{Primary}{Secondary}CartesianBy{Secondary}Id
      ({Secondary.keyType}{Secondary}Id);

```

以查询UserRoleNested为例，第一业务对象为“user”，第二业务对象为“Role”，则根据 Secondary 主键查询聚合对象列表的方法可以为 List<UserRoleCartesian> getUserRoleCartesianByRoleId(Long roleId)。

第三种情况，代码管理系统20可以为关系对象RO生成基本的CRUD方法及批量操作的方法，用于管理业务对象BO之间的关系。下面根据BO对象之间关系分别进行说明。

假如有以下User<-->Role多对多关系定义：

```

35    {
        "name": "rel-user-role",
        "primary": "User",
        "secondary": "Role",

```

```
"relationType": "MANY2MANY",
"apiTypes": [
  "NESTED",
  "CARTESIAN"
5 ],
"tableName": "user_role_new",
"fields": [
  {
    "name": "userId",
10    "type": "LONG",
    "length": 200,
    "remark": "用户id",
    "defaultValue": null,
    ...
15  },
  {
    "name": "roleId",
    "type": "LONG",
    "length": 200,
20    "remark": "角色id",
    ...
    },
  {
    "name": "creator",
25    "type": "STRING",
    "length": 200,
    "remark": "创建者",
    ...
    },
30  {
    "name": "createTime",
    "type": "STRING",
    "length": 200,
    ...
35  }
  ]
}
```

其中，fields为用户定义的关系表字段，关系表中至少包含两个字段，两个字段分别为userId和roleId，userId和roleId关联各自的业务对象。除上述两个字段，用户还可以添加其

它附加描述字段，包括但不限于关系的创建者、关系的生效或者关系的过期时间等。

代码管理系统20可以通过DevSpore为关系对象生成关系操作的dao接口文件和查询资源文件。其中，dao接口文件可以为UserRoleRelationMapper.java，查询资源文件可以为sql资源文件UserRoleRelationMapper.xml。与单个业务对象相同，UserRoleRelationMapper.java
5 可以生成在 {packageName}. {mapperPackageName} 或默认的 mapper} 包路径下，UserRoleRelationMapper.xml可以生成在默认位置如classpath:/mapper路径下。

关系操作的方法包括新增关系对象的方法、删除关系对象的方法、更新关系对象的方法或者查询关系对象的方法。下面对关系操作的方法分别进行详细说明。

针对新增关系对象的方法，代码管理系统20可以插入一条RO对象，返回插入行数，若
10 成功为1，失败为0，具体如下所示：

```
int insert({Primary}{Secondary}Relation relation);
```

以关系对象UserRoleRelation为例，代码管理系统20新增该关系对象的方法可以为int insert (UserRoleRelation relation)。

针对新增关系对象的方法，代码管理系统20也可以根据关系属性按需添加关系对象。
15 具体地，代码管理系统20可以根据关系实体记录，有条件地插入一条关系，若字段存在则不插入对应的字段，插入成功返回1，插入失败返回0，具体如下所示：

```
int insertSelective({Primary}{Secondary}Relation relation);
```

以关系对象UserRoleRelation为例，代码管理系统20可以选择性地新增关系对象，该方法可以为int insertSelective (UserRoleRelation relation)。

与批量新增业务对象类似，代码管理系统20可以批量新增关系对象。具体地，代码管理系统20可对以根据主键（如两个主键组成的联合主键）批量插入关系对象，并返回插入关系对象的条数，具体如下所示：

```
Integer insertBatch{Primary}By{Secondary}Id(List<{Primary}{Secondary}Key>relations);
```

以关系对象UserRoleRelation为例，批量新增该关系对象的方法可以为：

25 UserRole.java: Integer insertBatchUserByRoleId(List<UserRoleKey> relations);

```
Integer insertBatch{Secondary}By{Primary}Id(List<{Primary}{Secondary}Key> relations
```

```
RoleMapper.java: Integer insertBatchRoleByUserId(List<UserRoleKey> relations);
```

需要说明的是，代码管理系统20在批量插入关系对象时，通常支持插入主键，而不支持插入主键之外的其他信息。

30 针对删除关系对象的方法，代码管理系统20可以根据关系表的主键（例如为两个主键形成的联合主键）删除一条记录，返回删除行数，并在删除成功时返回1，失败或没有符合条件的记录时返回0，具体如下所示：

```
Int deleteByPrimaryKey(@Param("{Primary}Id") {Primary.keyType} {primary}Id ,  
@Param("{Secondary}Id") {Secondary.keyType} {secondary}Id);
```

35 以删除 UserRoleRelation 为例，代码管理系统20删除关系对象的方法可以为int deleteByPrimaryKey(@Param("userId") Long userId, @Param("roleId") Long roleId)。

与批量删除业务对象类似，代码管理系统20可以批量删除关系对象。具体地，代码管理系统20可以根据主键列表批量删除关系对象，其中，该方法的参数为PrimarySecondaryKey，

属性为外键，返回删除行数，若无符合条件的记录则返回0，具体如下所示：

```
Integer deleteBatch{Primary}By{Secondary}d(List<{Primary}{Secondary}Key> relations);
Integer deleteBatch{Secondary}By{Primary}d(List<{Primary}{Secondary}Key> relations);
```

下面以UserRoleKey为例，对批量删除关系对象进行说明。

5 该示例中定义有UserRoleKey，具体如下所示：

```
UserRoleKey{
    Long userId;
    Long roleId;
    //getter and setter
10 }
```

然后，代码管理系统20可以生成如下代码，以按照PrimaryKey删除UserMapper.java，按照SencondaryKey删除RoleMapper.java，具体如下所示：

UserMapper.java:

```
Integer deleteBatchUserByRoleId(List<UserRoleKey> relations);
```

15 RoleMapper.java:

```
Integer deleteBatchRoleById(List<UserRoleKey> relations)。
```

代码管理系统20还可以为关系对象生成更新关系对象的方法。具体地，代码管理系统20可以根据主键更新一条记录，更新成功返回1，更新失败或无符合条件的记录返回0，具体如下所示：

20 int update({Primary}{Secondary}Relation record);

以关系对象UserRoleRelation为例，代码管理系统20生成的更新关系对象的方法可以为int update(UserRoleRelation record)。

代码管理系统20也可以根据属性按需更新关系对象。其中，更新基准可以以联合主键为条件，其他属性如果未配置，则不更新内容，更新成功返回1，更新失败或无符合条件的记录返回0，具体如下所示：

25 int updateSelective({Primary}{Secondary}Relation record);

以关系对象UserRoleRelation为例，代码管理系统20根据属性按需更新该关系对象的方法可以为int updateSelective(UserRoleRelation record)。

代码管理系统20可以查询符合条件的关系对象个数，无符合条件的记录返回0，否则返回符合条件的个数，具体如下所示：

```
long countByExample({Primary}{Secondary}RelationCriteria example);
```

以关系对象UserRoleRelation为例，代码管理系统20查询符合条件的关系对象个数的方法可以为long countByExample(UserRoleRelationCriteria example)。

代码管理系统20还可以查询符合条件的关系记录。具体地，代码管理系统20可以根据给定的关系属性条件，查询符合条件的关系记录，返回符合条件的关系记录，若无符合条件的关系记录则返回空列表，如下所示：

```
List<{Primary}{Secondary}Relation>selectByExample({Primary}{Secondary}Relation
Criteria example);
```

以关系对象UserRoleRelation为例, 代码管理系统20查询符合条件的关系记录的方法可以为List<UserRoleRelation> selectByExample(UserRoleRelationCriteria example)。

代码管理系统20也可以根据主键(如联合主键)查询关系记录, 并返回查询到的关系记录, 如没有则返回null, 具体如下所示:

5 {Primary}{Secondary}RelationselectByPrimaryKey(@Param("{Primary}Id"){Primary.
KeyType}{primary}Id, @Param("{Secondary}Id"){Secondary.keyType}{secondary}Id);

以关系对象UserRoleRelation为例, 代码管理系统20根据主键查询关系记录的方法可以为 UserRoleRelation selectByPrimaryKey(@Param("userId") Long userId, @Param("roleId") Long roleId)。

10 代码管理系统20可以批量查询关系对象。具体地, 代码管理系统20可以根据多个主键(联合主键)批量查询关系信息, 其中, 参数为关系对象, 关系对象的属性中包括主键, 查询时以主键为条件查询, 若无符合条件的记录, 则返回空列表, 如下所示:

List<Primary>{Secondary}Relation>selectByPrimaryKeys(List<Primary>{Secondary}Relation>ids);

15 以关系对象UserRoleRelation为例, 代码管理系统20批量查询关系对象的方法可以为List<UserRoleRelation> selectByPrimaryKeys(List<UserRoleRelation> ids)。

进一步地, 代码管理系统20可以根据关系双方中一方的Id查询与之关联的另一方业务对象列表。例如, 代码管理系统20根据在关系中对方的Id, 查询本方的业务对象列表, 返回查询到的本方的业务对象列表, 若无符合条件的记录则返回空列表, 如下所示:

20 List<{Primary}> get{Primary}By{Secondary}Id({Secondary}.keyType id)

以关系对象UserRoleRelation为例, 代码管理系统20 根据对方的Id查询本方的业务对象列表的方法可以为:

UserMapper.java:

List<User> getUsersByRoleId(Long id);

25 List<{Secondary}> get{Secondary}By{Primary}Id({Primary}.keyType id)

RoleMapper.java:

List<Role> getRolesByUserId(Long id)。

代码管理系统20可以根据关系中对方的Id查询与之关联的本方业务对象主键列表, 返回查询到的本方业务对象主键列表, 若无符合条件的记录则返回空列表, 如下所示:

30 List<{Primary}.keyType> get{Primary}IdsBy{Secondary}Id({Secondary} id) ;

以关系对象UserRoleRelation为例, 代码管理系统20可以根据关系中对方的Id查询与之关联的本方业务对象主键列表的方法可以为:

UserMapper.java:

List<Long> selectUserIdsByRoleId(Long id);

35 List<{Secondary}.keyType>get{Secondary}IdsBy{Primary}Id({Primary}id) ;

roleMapper.java:

List<Long> selectRoleIdsByUserId(Long id)。

针对多对多中的递归关系, 代码管理系统20还可以为其生成以下方法:根据主键查询字

业务对象、根据主键查询父业务对象、根据主键查询子业务对象主键、根据主键查询父业务对象主键。

其中，代码管理系统20根据主键查询其子业务对象(由于无法使用存储过程，当前该方法只能查询一层关系)，若没有则返回null，具体如下所示：

5 List<{Bo}> getChild{Bo}sById({Bo.keyType} {bo}Id);

以当前业务对象为Roles，进行示例说明。代码管理系统20根据主键查询子业务对象的方法可以为 List<Role> getChildRolesById(Long roleId)。

类似地，代码管理系统20根据主键查询其父业务对象(由于无法使用存储过程，当前该方法只能查询一层关系)，若没有则返回null，具体如下所示：

10 List<{Bo}> getParent{Bo}sById({Bo.keyType} {bo}Id);

以当前业务对象为Roles，进行示例说明。代码管理系统20根据主键查询其父业务对象的方法可以为List<Role> getParentRolesById(Long roleId)。

代码管理系统20还可以根据主键查询其子业务对象主键(由于无法使用存储过程，当前该方法只能查询一层关系)，若没有则返回null，具体如下所示：

15 List<{Bo}> get{Bo}ChildIdsById({Bo.keyType} {bo}Id);

以当前业务对象为Roles，进行示例说明。代码管理系统20根据主键查询其子业务对象主键的方法可以为List<Long> selectRoleChildIdsById(Long id)。

代码管理系统20根据主键查询其父业务对象主键(由于无法使用存储过程，当前该方法只能查询一层关系)，若没有则返回null，具体如下所示：

20 List<{Bo}> get{Bo}ParentIdsById({Bo.keyType} {bo}Id);

以当前业务对象为Roles，进行示例说明。代码管理系统20根据主键查询其父业务对象主键的方法可以为List<Long> selectRoleParentIdsById(Long id)。

对于三个业务对象，可以构建两个关系对象。其中，两个关系对象中包括同一个业务对象时，例如Person-o2m->User 与 User<-m2m->Role中都包括User业务对象，则代码管理系统20可以为一个业务对象（如Role）生成根据另一个业务对象（如Person）的主键经过共同业务对象（如User）查询间接关联的Role业务对象及其对应的主键的方法。

又例如，关系对象 Company aggregate->CompanyAccount 与关系对象 Workspace o2m->Company 中包括Company业务对象，则代码管理系统20可以为CompanyAccount生成根据Workspace主键经过Company查询间接关联的CompanyAccount业务对象及主键的方法。

30 在一些可能的实现方式中，代码管理系统20可以根据两个关系对象中经过中间业务对象查询对方业务对象记录的方法，若不存在符合条件的记录则返回空列表，具体如下所示：

 List<{Bo}> get{Bo}sThru{SecondBo}By{ThirdBo}Id({ThirdBo.keyType} {thirdBo}Id);

以 Person-o2m->User 与 User<-m2m->Role为例，代码管理系统20生成的查询对方业务对象记录的方法可以为：

35 RoleMapper.java:

 List<Role> getRolesThruUserByPersonId(Long personId);

 CompanyAccount.java:

 List<CompanyAccount> getCompanyAccountsThruCompanyByWorkspaceId(Long

workspaceId)。

基于三个业务对象构成的两个关系对象中有同一个业务对象时，例如：Person-o2m->User User<-m2m->Role中都包括User业务对象，其中User与Role为多对多关系，则代码管理系统20为Role生成根据Person主键经过User查询间接关联的Role业务对象并且带有UserId的目标对象：RoleWithUserId。

代码管理系统20可以根据两个关系对象中经过中间业务对象查询对方业务对象主键记录的方法，若不存在符合条件的记录则返回空列表，若查询的目标对象和中间业务对象的关系为多对多关系，则为其生成查询带有中间对象主键的目标对象，具体如下所示：

```
List<{Bo}With{SecondBo}Id>get{Bo}sThru{SecondBo}By{ThirdBo}Id2({ThirdBo.key  
Type}{thirdBo}Id);
```

以 Person-o2m->User 与 User<-m2m->Role为例，代码管理系统20可以生成查询对方业务对象主键记录的方法可以为：

roleMapper.java:

```
List<RoleWithUserId> getRolesThruUserByPersonId2(Long personId)。
```

在一些可能的实现方式中，代码管理系统20还可以接收用户通过所述配置界面配置的组件扩展策略，如图4所示，元数据文档中还包括组件扩展策略，代码管理系统20还可以根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。其中，扩展组件可以包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。

本申请实施例还定义了基于业务对象设计的数据库模型，该数据库模型为分库分表模型。参见图7所示的数据库模型的结构示意图，数据库模型如逻辑数据库1至逻辑数据库包括根表对象和分库分表对象。其中，根表对象可以记作root表，分库分表对象可以记作sharding表。用户通过预置对象类型，可以为后续的分库分表操作提供依据，可实现便捷的分库分表操作。

上文对代码管理系统20生成微服务的代码进行了介绍，微服务的代码包括多层代码，代码管理系统20还可以生成所述微服务的代码目录，该所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。

基于上述内容描述，本申请实施例提供了一种创建微服务的方法。该方法中，代码管理系统20基于领域驱动设计的理念，根据用户定义的业务模型，具体是业务模型中业务对象的元数据以及标识业务对象的关系对象的元数据生成微服务的代码，解决了代码生成工具或开发平台内置的功能模块生成特定层的简单代码的问题，提高了开发效率。而且，该方法支持用户灵活定义业务模型，具有较好的扩展能力，能够完成微服务开发的全链路打通，解决微服务开发早期大量的框架性开发工作，及云服务对接、部署系统对接、中间件支持、高性能高可靠场景架构设计等，为开发者提供成熟的产品级解决方案。

接下来，结合具体应用场景对本申请实施例的创建微服务的方法进行介绍。

参见图8所示的创建微服务的方法的流程图，该方法包容如下步骤：

第一步：用户通过门户系统（portal）触发创建项目的操作，门户系统响应于创建项目

的操作，向应用后端服务发生创建项目的指令，以使得应用后端服务创建项目。

其中，项目可以是一个应用，应用可以包括一个或多个微服务。微服务可以对接各种云服务，如微服务可以对接数据库、存储、日志、监控、负载均衡等云服务，并和上述云服务一同以组件形式提供给用户使用。

5 参见图 9A 所示的门户系统提供的应用创建界面的示意图，应用创建界面包括应用新建控件，用户触发该应用新建控件，触发创建项目或应用的操作，相应地，门户系统可以响应于用户的上述操作，向应用后端服务发送创建项目的指令。接着参见图 9B 所示的门户系统提供的应用配置界面的示意图，用户可以通过简单的配置，如应用名称、应用介绍、关联项目（作为创建应用的模板），实现应用的快速创建。

10 第二步：用户通过门户系统触发创建微服务的操作，门户系统响应于创建微服务的操作，创建微服务，并向云平台的基础服务申请与该微服务对应的基础服务。

本实施例中，微服务可以组件形式提供给用户。参见图 9C 所示的微服务创建界面的示意图，微服务创建界面包括新建组件控件，该控件被触发时，可以跳转至图 9D 至图 9F 所示的微服务配置界面，用户可以通过微服务配置界面进行简单的配置，从而实现微服务的快速创建。

15 如图 9D 至图 9F 所示，用户可以通过微服务配置界面配置开发语言、应用模板、应用托管方式、应用基本信息（如组件名、组件描述、组、包名）、对接的云服务系统选择等（包括应用托管、日志系统、监控系统、调用链路系统等）、云服务申请等（包括数据库、缓存、负载均衡、API 网关、容器集群等）。

20 第三步：应用后端服务根据业务设计，获得业务模型的元数据，然后调用代码生成服务，以便于根据业务模型的元数据，通过代码生成服务生成微服务的代码，从而创建微服务。

接着参见图 9G 所示的微服务创建流程示意图，用户在通过上述配置界面完成上述配置后，应用后端服务可以创建组件、生成工程框架、创建代码仓库、提交代码、创建流水线，直至创建成功。其中，应用后端服务在调用代码生成服务生成微服务的代码时，还可以查询代码生成状态，并在代码生成状态为成功时，向代码仓库提交代码。其中，创建成功后，可以在图 9H 的界面中的概览区域显示微服务的信息，例如显示代码仓、流水线。进一步地，概览区域还可以显示告警统计信息以及最新告警，以便于用户能够根据告警统计信息或最新告警对微服务的代码进行调整。此外，概览区域还可以显示实例数量、接口
25 时延分布等，以便根据接口时延分布调整部署情况。

图 9I 和图 9J 分别示出了代码生成流程以及代码生成结果，如图 9I 所示，用户可以触发新建微服务的操作，然后通过门户系统的配置界面配置业务对象以及关系对象的元数据，代码生成服务可以基于上述元数据自动生成代码。需要说明的是，代码生成服务可以根据配置文件中的元数据生成 spring mvc 框架代码、流水线代码、数据持久操作层代码（如
35 batis.xml）和结构化查询语言（Structured Query Language, SQL）查询语句中数据库模式定义语言（Data Definition Language, DDL）代码中的一种或多种。代码生成服务还可以根据配置文件中的元数据生成中间文件，例如生成 swagger.yaml，然后基于中间文件，通过代码生成服务中的代码生成插件生成服务层代码和模型层代码中的一种或多种。进一步地，用

户还可以添加扩展接口、实现扩展业务逻辑，生成微服务的扩展组件的代码。该扩展组件包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。图 9J 示出了代码生成服务基于代码分层规范生成的微服务的各层代码，以及基于目录规范生成目录。

该方法通过将代码生成整合到整个应用开发流程中，可提供给企业或开发者使用，通过流程化的创建应用，自动完成应用创建的各项服务对接，如此提高了创建效率。

接下来，参见图 10 所示的创建微服务的方法的流程图，该方法包括如下步骤：

第一步：用户通过前端界面如门户系统或者客户端触发生成空项目的操作，然后用户通过向云 IDE 传入代码仓地址和 token 打开项目，接着向 IDE 插件导入 DDL，以编辑业务。

第二步：IDE 插件根据用户编辑的业务模型的元数据，调用代码生成服务，生成微服务的代码。

第三步：代码生成微服务向代码仓库上传代码，IDE 插件从代码仓下载代码，云 IDE 从 IDE 插件打开下载的代码，实现业务逻辑。

下面对 IDE 插件使用流程进行详细说明。

参见图 11A 至图 11B 的微服务配置界面的示意图，用户可以通过图 11A 所示的微服务配置界面配置项目基本信息，例如是配置微服务名称、组标识（group ID）、版本等信息，以及通过图 11B 所示的微服务配置界面配置框架组件，如数据库组件、认证服务组件、鉴权服务组件、缓存服务组件等等。

接着参见图 11C 所示的微服务业务设计界面的示意图，该界面支持从零开始设计，也支持导入已有数据进行设计，具体用于设计业务对象以及业务对象之间的关系。关系使用连接线表示，用户定义连接线的属性，从而实现关系设计。

然后参见图 11D 所示的创建流程界面示意图，代码生成服务基于业务设计生成代码，用户通过在 IDE 加载代码以打开项目，实现业务逻辑。

该方法将应用创建过程内置到 IDE 插件中，开发者只需要在插件应用市场下载插件安装即可使用。如此增加用户使用便捷性，提升用户友好度。

接着，参见图 12 所示的创建微服务的方法的流程图，该方法包括如下步骤：

第一步：用户触发创建微服务的操作，配置微服务的基本信息。

参见图 13A 所示的微服务配置界面的示意图，用户可以通过该微服务配置界面配置创建类型、微服务名称、项目、工作空间、服务号、描述中的一种或多种，其中，创建类型包括新建微服务，或者是从 UADP 选择。项目是指该微服务所属应用。

第二步：用户选择业务模型及服务配置。

参见图 13B 所示的微服务配置界面的示意图，用户可以通过该微服务配置界面配置框架配置。框架配置具体包括服务类型、代码模板，其中，代码模板包括自定义字段，如数据库模型、模型分支、打包方式、认证方式、缓存方式、版本、Group ID。

在代码模板自定义字段中，“数据库模型”可选择 DbDesigner 设计好的数据模型，也可以点击“数据库模型”下拉列表后面的“新建”创建一个空数据库模型来创建微服务。

第三步：用户选择代码仓。

参见图 13C 所示的微服务配置界面的示意图,用户可以通过该界面进行代码仓库配置,具体地,用户选择需要的仓库类型及仓库组织,若组织不存在,用户可直接手动输入组织名称,创建微服务时会同步在代码库中创建组织。

第四步:用户配置流水线,以创建微服务。

5 当在图 13C 的界面选择了“现在创建流水线”,则在图 13D 的界面中可配置该流水线。具体地,用户可以通过图 13D 的微服务配置界面选择代码检查规则集、部署系统,部署系统配置为 container 时,表示采用容器化部署,部署系统配置为 general 时,表示采用虚拟机部署。

10 该方法通过将应用创建服务化,支持在前端直接导入数据模型进行应用创建,保障了业务设计的一致性,能够实现根据业务设计直接生成应用中微服务的代码,提高了创建微服务的效率。

15 基于本申请实施例提供的创建微服务的方法,本申请实施例还提供了一种如前述的代码管理系统 20。下面将从功能模块化的角度,结合附图对本申请实施例提供的代码管理系统 20 进行介绍。

参见图 14 所示的代码管理系统 20 的结构示意图,该系统 20 包括:

交互模块 1402,用于向用户呈现配置界面;

20 通信模块 1404,用于接收用户通过所述配置界面配置的所述微服务的业务模型,所述业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据;

生成模块 1406,用于根据所述微服务的业务模型,生成所述微服务的代码。

在一些可能的实现方式中,所述生成模块 1406 具体用于:

25 根据所述微服务的业务模型生成基础模块代码和扩展模块代码,所述基础模块代码不支持所述用户修改,所述扩展模块代码支持所述用户修改,所述基础模块代码与所述扩展模块代码分离。

在一些可能的实现方式中,所述微服务的代码包括控制器层代码、业务逻辑层代码、数据聚合层代码、数据持久操作层代码和数据库实体层代码中的至少一种。

在一些可能的实现方式中,所述生成模块 1406 具体用于:

30 根据所述业务对象的元数据或所述关系对象的元数据,生成所述微服务的数据持久操作层代码,所述微服务的数据持久操作层代码包括接口代码和查询资源代码。

在一些可能的实现方式中,所述多个业务对象的关系包括:一对多关系、多对多关系、递归关系或聚合关系中的一种或多种。

在一些可能的实现方式中,所述多个业务对象包括第一业务对象和第二业务对象,所述第一业务对象和所述第二业务对象的关系为所述聚合关系,所述系统 20 还包括:

35 聚合模块 1408,用于将所述第一业务对象和所述第二业务对象聚合得到数据传输对象;所述生成模块 1406 具体用于:

为所述数据传输对象生成查询资源代码。

在一些可能的实现方式中,所述聚合模块 1408 具体用于:

采用嵌套模式或笛卡尔积模式，聚合所述第一业务对象和所述第二业务对象得到所述数据传输对象。

在一些可能的实现方式中，所述业务模型还包括扩展的应用程序接口 API 的元数据，所述生成模块 1406 还用于：

5 根据所述扩展的 API 的元数据生成所述扩展的 API 的代码。

在一些可能的实现方式中，所述通信模块 1404 还用于：

接收用户通过所述配置界面配置的代码生成策略；

所述生成模块 1406 具体用于：

根据所述微服务的业务模型，按照所述代码生成策略，生成所述微服务的代码。

10 在一些可能的实现方式中，所述代码生成策略包括新建或继承。

在一些可能的实现方式中，所述通信模块 1404 还用于：

接收用户通过所述配置界面配置的组件扩展策略；

所述生成模块 1406 具体用于：

15 根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。

在一些可能的实现方式中，所述扩展组件包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。

在一些可能的实现方式中，所述系统 20 还包括：

创建模块 1409，用于创建数据库模型，所述数据库模型包括根表对象和分库分表对象。

20 在一些可能的实现方式中，所述微服务的代码包括多层代码，所述生成模块 1406 还用于：

生成所述微服务的代码目录，所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。

25 需要说明的是，代码管理系统 20 可以包括代码生成装置 202 和代码管理装置 204。上述交互模块 1402、通信模块 1404、生成模块 1406、聚合模块 1408、创建模块 1409 可以是代码生成装置 202 中的功能模块，代码生成装置 202 的上述模块用于实现本申请实施例的创建微服务的方法。进一步地，代码管理系统 20 还可以包括流水线管理模块、运行治理模块、运维管理模块（图 14 中未示出），这些模块可以是代码管理装置 204 中的模块，用于实现流水线自动生成、微服务治理、微服务运维等功能。

30 根据本申请实施例的代码管理系统 20 可对应于执行本申请实施例中描述的方法，并且代码管理系统 20 的各个模块/单元的上述和其它操作和/或功能分别为了实现图 3 所示实施例中的各个方法的相应流程，为了简洁，在此不再赘述。

35 本申请实施例还提供一种计算设备集群。该计算设备集群包括至少一台计算设备，该至少一台计算设备中的任一计算设备可以来自云环境或者边缘环境，也可以是终端设备。该计算设备集群具体用于实现如图 14 所示实施例中代码管理系统 20 的功能。

图 15 提供了一种计算设备集群的结构示意图，如图 15 所示，计算设备集群 150 包括多台计算设备 1500，计算设备 1500 包括总线 1501、处理器 1502、通信接口 1503 和存储

器 1504。处理器 1502、存储器 1504 和通信接口 1503 之间通过总线 1501 通信。

总线 1501 可以是外设部件互连标准 (peripheral component interconnect, PCI) 总线或扩展工业标准结构 (extended industry standard architecture, EISA) 总线等。总线可以分为地址总线、数据总线、控制总线等。为便于表示, 图 15 中仅用一条粗线表示, 但并不表示仅有一根总线或一种类型的总线。

处理器 1502 可以为中央处理器 (central processing unit, CPU)、图形处理器 (graphics processing unit, GPU)、微处理器 (micro processor, MP) 或者数字信号处理器 (digital signal processor, DSP) 等处理器中的任意一种或多种。

通信接口 1503 用于与外部通信。例如, 通信接口 1503 用于接收用户通过配置界面配置的微服务的业务模型, 接收用户通过配置界面配置的代码生成策略, 接收用户通过配置界面配置的组件扩展策略等等。

存储器 1504 可以包括易失性存储器 (volatile memory), 例如随机存取存储器 (random access memory, RAM)。存储器 1504 还可以包括非易失性存储器 (non-volatile memory), 例如只读存储器 (read-only memory, ROM), 快闪存储器, 硬盘驱动器 (hard disk drive, HDD) 或固态驱动器 (solid state drive, SSD)。

存储器 1504 中存储有计算机可读指令, 处理器 1502 执行该计算机可读指令, 以使得计算设备集群 150 执行前述创建微服务的方法 (或实现前述代码管理系统 20 的功能)。

具体地, 在实现图 14 所示系统的实施例的情况下, 且图 14 中所描述的代码管理系统 20 的各模块如交互模块 1402、通信模块 1404、生成模块 1406、聚合模块 1408、创建模块 1409 的功能为通过软件实现的情况下, 执行图 14 中各模块的功能所需的软件或程序代码可以存储在计算设备集群 150 中的至少一个存储器 1504 中。至少一个处理器 1502 执行存储器 1504 中存储的程序代码, 以使得计算设备集群 150 执行前述创建微服务的方法。

本申请实施例还提供了一种计算机可读存储介质。所述计算机可读存储介质可以是计算设备能够存储的任何可用介质或者是包含一个或多个可用介质的数据中心等数据存储设备。所述可用介质可以是磁性介质, (例如, 软盘、硬盘、磁带)、光介质 (例如, DVD)、或者半导体介质 (例如固态硬盘) 等。该计算机可读存储介质包括指令, 所述指令指示计算设备或计算设备集群执行上述创建微服务的方法。

本申请实施例还提供了一种计算机程序产品。所述计算机程序产品包括一个或多个计算机指令。在计算设备上加载和执行所述计算机指令时, 全部或部分地产生按照本申请实施例所述的流程或功能。所述计算机指令可以存储在计算机可读存储介质中, 或者从一个计算机可读存储介质向另一计算机可读存储介质传输, 例如, 所述计算机指令可以从一个网站站点、计算设备或数据中心通过有线 (例如同轴电缆、光纤、数字用户线 (DSL)) 或无线 (例如红外、无线、微波等) 方式向另一个网站站点、计算设备或数据中心进行传输。所述计算机程序产品可以为一个软件安装包, 在需要使用前述创建微服务的方法的任一方法的情况下, 可以下载该计算机程序产品并在计算设备或计算设备集群上执行该计算机程序产品。

上述各个附图对应的流程或结构的描述各有侧重, 某个流程或结构中没有详述的部分, 可以参见其他流程或结构的相关描述。

权 利 要 求

1. 一种创建微服务的方法，其特征在于，所述方法包括：
向用户呈现配置界面；
接收用户通过所述配置界面配置的所述微服务的业务模型，所述业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据；
根据所述微服务的业务模型，生成所述微服务的代码。
2. 根据权利要求 1 所述的方法，其特征在于，所述根据所述微服务的业务模型，生成所述微服务的代码，包括：
根据所述微服务的业务模型生成基础模块代码和扩展模块代码，所述基础模块代码不支持所述用户修改，所述扩展模块代码支持所述用户修改，所述基础模块代码与所述扩展模块代码分离。
3. 根据权利要求 1 或 2 所述的方法，其特征在于，所述微服务的代码包括控制器层代码、业务逻辑层代码、数据聚合层代码、数据持久操作层代码和数据库实体层代码中的至少一种。
4. 根据权利要求 3 所述的方法，其特征在于，所述根据所述微服务的业务模型，生成所述微服务的代码，包括：
根据所述业务对象的元数据或所述关系对象的元数据，生成所述微服务的数据持久操作层代码，所述微服务的数据持久操作层代码包括接口代码和查询资源代码。
5. 根据权利要求 1 至 4 任一项所述的方法，其特征在于，所述多个业务对象的关系包括：一对多关系、多对多关系、递归关系或聚合关系中的一种或多种。
6. 根据权利要求 5 所述的方法，其特征在于，所述多个业务对象包括第一业务对象和第二业务对象，所述第一业务对象和所述第二业务对象的关系为所述聚合关系，所述方法还包括：
将所述第一业务对象和所述第二业务对象聚合得到数据传输对象；
为所述数据传输对象生成查询资源代码。
7. 根据权利要求 6 所述的方法，其特征在于，所述将所述第一业务对象和所述第二业务对象聚合得到数据传输对象，包括：
采用嵌套模式或笛卡尔积模式，聚合所述第一业务对象和所述第二业务对象得到所述数据传输对象。
8. 根据权利要求 4、6 或 7 所述的方法，其特征在于，所述业务模型还包括扩展的应用程序接口 API 的元数据，所述方法还包括：
根据所述扩展的 API 的元数据生成所述扩展的 API 的代码。
9. 根据权利要求 1 至 8 任一项所述的方法，其特征在于，所述方法还包括：
接收用户通过所述配置界面配置的代码生成策略；
所述根据所述微服务的业务模型，生成所述微服务的代码，包括：
根据所述微服务的业务模型，按照所述代码生成策略，生成所述微服务的代码。
10. 根据权利要求 9 所述的方法，其特征在于，所述代码生成策略包括新建或继承。
11. 根据权利要求 1 至 8 任一项所述的方法，其特征在于，所述方法还包括：

接收用户通过所述配置界面配置的组件扩展策略；

所述根据所述微服务的业务模型，生成所述微服务的代码，包括：

根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。

- 5 12. 根据权利要求 11 所述的方法，其特征在于，所述扩展组件包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。

13. 根据权利要求 1 至 12 任一项所述的方法，其特征在于，所述方法还包括：

创建数据库模型，所述数据库模型包括根表对象和分库分表对象。

- 10 14. 根据权利要求 1 至 13 任一项所述的方法，其特征在于，所述微服务的代码包括多层代码，所述方法还包括：

生成所述微服务的代码目录，所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。

15. 一种代码管理系统，其特征在于，所述系统包括：

交互模块，用于向用户呈现配置界面；

- 15 通信模块，用于接收用户通过所述配置界面配置的所述微服务的业务模型，所述业务模型包括多个业务对象的元数据和用于标识所述多个业务对象之间关系的关系对象的元数据；

生成模块，用于根据所述微服务的业务模型，生成所述微服务的代码。

16. 根据权利要求 15 所述的系统，其特征在于，所述生成模块具体用于：

- 20 根据所述微服务的业务模型生成基础模块代码和扩展模块代码，所述基础模块代码不支持所述用户修改，所述扩展模块代码支持所述用户修改，所述基础模块代码与所述扩展模块代码分离。

- 25 17. 根据权利要求 15 或 16 所述的系统，其特征在于，所述微服务的代码包括控制器层代码、业务逻辑层代码、数据聚合层代码、数据持久操作层代码和数据库实体层代码中的至少一种。

18. 根据权利要求 17 所述的系统，其特征在于，所述根据所述微服务的业务模型，生成所述微服务的代码，包括：

根据所述业务对象的元数据或所述关系对象的元数据，生成所述微服务的数据持久操作层代码，所述微服务的数据持久操作层代码包括接口代码和查询资源代码。

- 30 19. 根据权利要求 15 至 18 任一项所述的系统，其特征在于，所述多个业务对象的关系包括：一对多关系、多对多关系、递归关系或聚合关系中的一种或多种。

20. 根据权利要求 19 所述的系统，其特征在于，所述多个业务对象包括第一业务对象和第二业务对象，所述第一业务对象和所述第二业务对象的关系为所述聚合关系，所述系统还包括：

- 35 聚合模块，用于将所述第一业务对象和所述第二业务对象聚合得到数据传输对象；

所述生成模块，具体用于：

为所述数据传输对象生成查询资源代码。

21. 根据权利要求 20 所述的系统，其特征在于，所述聚合模块具体用于：

采用嵌套模式或笛卡尔积模式，聚合所述第一业务对象和所述第二业务对象得到所述数据传输对象。

22. 根据权利要求 18、20 或 21 所述的系统，其特征在于，所述业务模型还包括扩展的应用程序接口 API 的元数据，所述生成模块还用于：

5 根据所述扩展的 API 的元数据生成所述扩展的 API 的代码。

23. 根据权利要求 15 至 22 任一项所述的系统，其特征在于，所述通信模块还用于：

接收用户通过所述配置界面配置的代码生成策略；

所述生成模块具体用于：

根据所述微服务的业务模型，按照所述代码生成策略，生成所述微服务的代码。

10 24. 根据权利要求 23 所述的系统，其特征在于，所述代码生成策略包括新建或继承。

25. 根据权利要求 15 至 22 任一项所述的系统，其特征在于，所述通信模块还用于：

接收用户通过所述配置界面配置的组件扩展策略；

所述生成模块具体用于：

15 根据所述微服务的业务模型，按照所述组件扩展策略，生成所述微服务的扩展组件的代码。

26. 根据权利要求 25 所述的系统，其特征在于，所述扩展组件包括鉴权组件、校验组件、探针组件、标注式配置组件中的一种或多种。

27. 根据权利要求 15 至 26 任一项所述的系统，其特征在于，所述系统还包括：

创建模块，用于创建数据库模型，所述数据库模型包括根表对象和分库分表对象。

20 28. 根据权利要求 15 至 27 任一项所述的系统，其特征在于，所述微服务的代码包括多层代码，所述生成模块还用于：

生成所述微服务的代码目录，所述代码目录包括多个子目录，所述多个子目录与所述多层代码一一对应。

25 29. 一种计算设备集群，其特征在于，所述计算设备集群包括至少一台计算设备，所述至少一台计算设备包括至少一个处理器和至少一个存储器，所述至少一个存储器中存储有计算机可读指令；所述至少一个处理器执行所述计算机可读指令，以使得所述计算设备集群执行如权利要求 1 至 14 中任一项所述的方法。

30. 一种计算机可读存储介质，其特征在于，包括计算机可读指令；所述计算机可读指令用于实现权利要求 1 至 14 任一项所述的方法。

30 31. 一种计算机程序产品，其特征在于，包括计算机可读指令；所述计算机可读指令用于实现权利要求 1 至 14 任一项所述的方法。