歉信君 —— 信真科技·信守真品

www.xinzhenkj.com

微服务架构最强详解

目录如下：

一、微服务架构介绍

微服务架构（Microservice Architecture）是一种架构概念，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。你可以将其看作是在架构层次而非获取服务的

类上应用很多SOLID原则。微服务架构是个很有趣的概念，它的主要作用是将功能分解到离散的各个服务当中，从而降低系统的耦合性，并提供更加灵活的服务支持。

概念：把一个大型的单个应用程序和服务拆分为数个甚至数十个的支持微服务，它可扩展单个组件而不是整个的应用程序堆栈，从而满足服务等级协议。

定义：围绕业务领域组件来创建应用，这些应用可独立地进行开发、管理和迭代。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单。

本质：用一些功能比较明确、业务比较精练的服务去解决更大、更实际的问题。

二、出现和发展

微服务（Microservice）这个概念是2012年出现的，作为加快Web和移动应用程序开发进程的一种方法，2014年开始受到各方的关注，而2015年，可以说是微服务的元年；

越来越多的论坛、社区、blog以及互联网行业巨头开始对微服务进行讨论、实践，可以说这样更近一步推动了微服务的发展和创新。而微服务的流行，Martin Fowler功不可没。

这老头是个奇人，特别擅长抽象归纳和制造概念。特别是微服务这种新生的名词，都有一个特点：**一解释就懂，一问就不知，一讨论就打架。**

Martin Fowler是国际著名的OO专家，敏捷开发方法的创始人之一，现为ThoughtWorks公司的首

席科学家。在面向对象分析设计、UML、模式、软件开发方法学、XP、重构等方面，都是世界顶级的

专家，现为Thought Works公司的首席科学家。Thought Works是一家从事企业应用开发和一一集

成的公司。早在20世纪80年代，Fowler就是使用对象技术构建多层企业应用的倡导者，他著有几

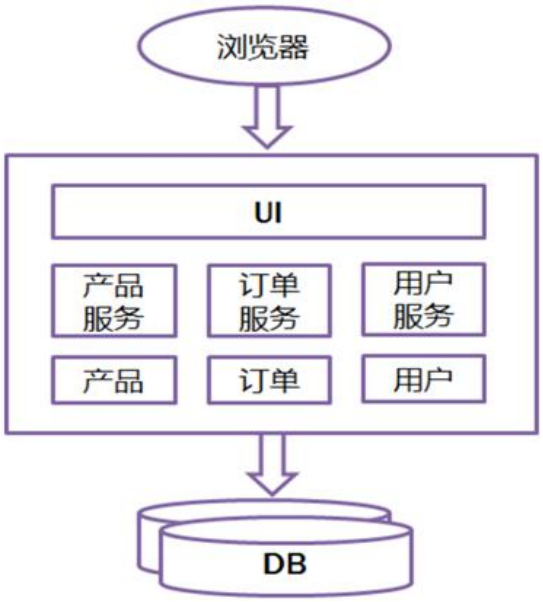
本经典书籍：《企业应用架构模式》、《UML精粹》和《重构》等。

———— 百度百科

三、传统开发模式和微服务的区别

先来看看传统的web开发方式，通过对比比较容易理解什么是Microservice Architecture。和Microservice相对应的，这种方式一般被称为Monolithic（单体式开发）。

所有的功能打包在一个 WAR包里，基本没有外部依赖（除了容器），部署在一个JEE容器（Tomcat，JBoss，WebLogic）里，包含了 DO/DAO，Service，UI等所有逻辑。



优点：

①开发简单，集中式管理

②基本不会重复开发

③功能都在本地，没有分布式的管理和调用消耗

缺点：

1、效率低：开发都在同一个项目改代码，相互等待，冲突不断

2、维护难：代码功功能耦合在一起，新人不知道何从下手

3、不灵活：构建时间长，任何小修改都要重构整个项目，耗时

4、稳定性差：一个微小的问题，都可能导致整个应用挂掉

5、扩展性不够：无法满足高并发下的业务需求

常见的系统架构遵循的三个标准和业务驱动力：

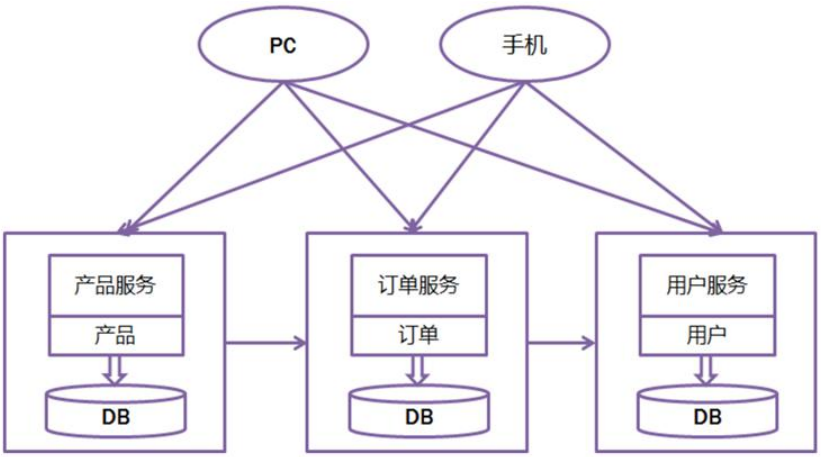
1、提高敏捷性：及时响应业务需求，促进企业发展

2、提升用户体验：提升用户体验，减少用户流失

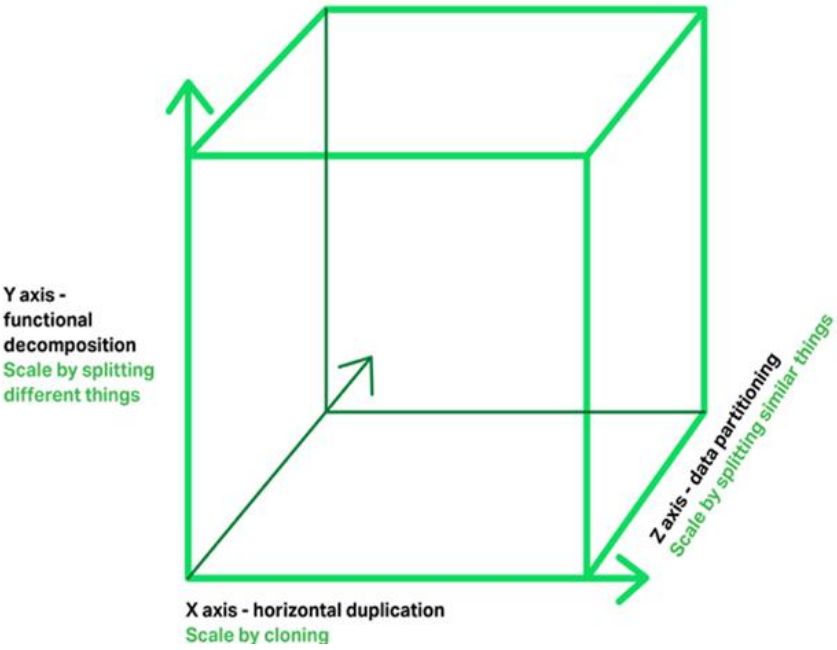
3、降低成本：降低增加产品、客户或业务方案的成本

基于微服务架构的设计：

目的：有效的拆分应用，实现敏捷开发和部署



关于微服务的一个形象表达：



X轴：运行多个负载均衡器之后的运行实例

Y轴：将应用进一步分解为微服务（分库）

Z轴：大数据量时，将服务分区（分表）

四、微服务的具体特征

官方的定义：

1、一些列的独立的服务共同组成系统

2、单独部署，跑在自己的进程中

3、每个服务为独立的业务开发

4、分布式管理

5、非常强调隔离性

大概的标准：

1、分布式服务组成的系统

2、按照业务，而不是技术来划分组织

3、做有生命的产品而不是项目

4、强服务个体和弱通信（ Smart endpoints and dumb pipes ）

5、自动化运维（ DevOps ）

6、高度容错性

7、快速演化和迭代

五、SOA和微服务的区别

1、SOA喜欢重用，微服务喜欢重写

SOA的主要目的是为了企业各个系统更加容易地融合在一起。 说到SOA不得不说ESB(EnterpriseService Bus)。 ESB是什么? 可以把ESB想象成一个连接所有企业级服务的脚手架。

通过service broker，它可以把不同数据格式或模型转成canonical格式，把XML的输入转成CSV传给legacy服务，把SOAP 1.1服务转成 SOAP 1.2等等。 它还可以把一个服务

路由到另一个服务上，也可以集中化管理业务逻辑，规则和验证等等。 它还有一个重要功能是消息队列和事件驱动的消息传递，比如把JMS服务转化成SOAP协议。 各服务间可能有

复杂的依赖关系。

微服务通常由重写一个模块开始。要把整个巨石型的应用重写是有很大的风险的，也不一定必要。我们向微服务迁移的时候通常从耦合度最低的模块或对扩展性要求最高的模块开始，

把它们一个一个剥离出来用敏捷地重写，可以尝试最新的技术和语言和框架，然 后单独布署。 它通常不依赖其他服务。微服务中常用的API Gateway的模式主要目的也不是重用代码，

而是减少客户端和服务间的往来。API gateway模式不等同与Facade模式，我们可以使用如future之类的调用，甚至返回不完整数据。

2、SOA喜欢水平服务，微服务喜欢垂直服务

SOA设计喜欢给服务分层(如Service Layers模式)。 我们常常见到一个Entity服务层的设计，美其名曰Data Access Layer。 这种设计要求所有的服务都通过这个Entity服务层

来获取数据。 这种设计非常不灵活，比如每次数据层的改动都可能影响到所有业务层的服务。 而每个微服务通常有它自己独立的data store。 我们在拆分数据库时可以适当的做些

去范式化(denormalization)，让它不需要依赖其他服务的数据。

微服务通常是直接面对用户的，每个微服务通常直接为用户提供某个功能。 类似的功能可能针对手机有一个服务，针对机顶盒是另外一个服务。 在SOA设计模式中这种情况通常会用到

Multi-ChannelEndpoint的模式返回一个大而全的结果兼顾到所有的客户端的需求。

3、SOA喜欢自上而下，微服务喜欢自下而上

SOA架构在设计开始时会先定义好服务合同(service contract)。 它喜欢集中管理所有的服务，包括集中管理业务逻辑，数据，流程，schema，等等。 它使用Enterprise

Inventory和Service Composition等方法来集中管理服务。 SOA架构通常会预先把每个模块服务接口都定义好。 模块系统间的通讯必须遵守这些接口，各服务是针对他们的调用者。

SOA架构适用于TOGAF之类的架构方法论。

微服务则敏捷得多。只要用户用得到，就先把这个服务挖出来。然后针对性的，快速确认业务需求，快速开发迭代。

六、怎么具体实践微服务

要实际的应用微服务，需要解决一下四点问题：

1、客户端如何访问这些服务

2、每个服务之间如何通信

3、如此多的服务，如何实现?

4、服务挂了，如何解决? （备份方案，应急处理机制）

1、客户端如何访问这些服务

原来的Monolithic方式开发，所有的服务都是本地的，UI可以直接调用，现在按功能拆分成独立的服务，跑在独立的一般都在独立的虚拟机上的 Java进程了。客户端UI如何访问他的?

后台有N个服务，前台就需要记住管理N个服务，一个服务下线/更新/升级，前台就要重新部署，这明显不服务我们 拆分的理念，特别当前台是移动应用的时候，通常业务变化的节奏更快。

另外，N个小服务的调用也是一个不小的网络开销。还有一般微服务在系统内部，通常是无 状态的，用户登录信息和权限管理最好有一个统一的地方维护管理（OAuth）。

所以，一般在后台N个服务和UI之间一般会一个代理或者叫API Gateway，他的作用包括：

① 提供统一服务入口，让微服务对前台透明

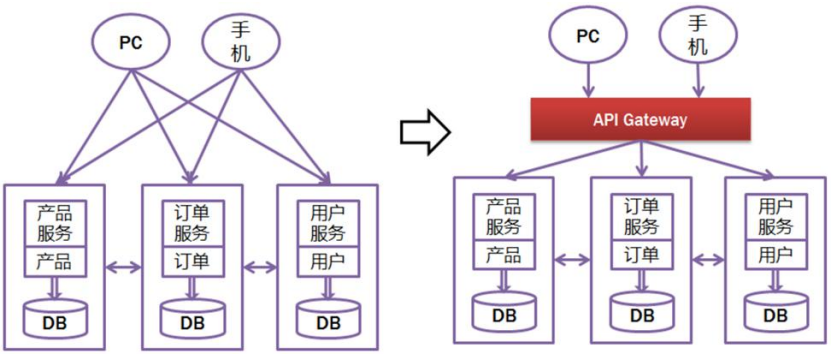
② 聚合后台的服务，节省流量，提升性能

③ 提供安全，过滤，流控等API管理功能

其实这个API Gateway可以有很多广义的实现办法，可以是一个软硬一体的盒子，也可以是一个简单的MVC框架，甚至是一个Node.js的服务端。他们最重要的作 用是为前台（通常是

移动应用）提供后台服务的聚合，提供一个统一的服务出口，解除他们之间的耦合，不过API Gateway也有可能成为单点故障点或者性能的瓶颈。

用过Taobao Open Platform（淘宝开放平台）的就能很容易的体会，TAO就是这个API Gateway。



2、每个服务之间如何通信

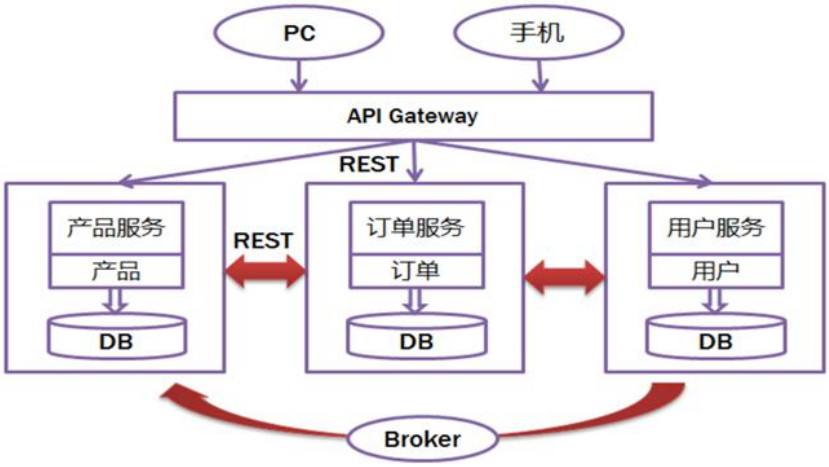
所有的微服务都是独立的Java进程跑在独立的虚拟机上，所以服务间的通信就是IPC（inter process communication），已经有很多成熟的方案。现在基本最通用的有两种方式：

同步调用：

①REST（JAX-RS，Spring Boot）

②RPC（Thrift, Dubbo）

异步消息调用(Kafka, Notify, MetaQ)



同步和异步的区别：

一般同步调用比较简单，一致性强，但是容易出调用问题，性能体验上也会差些，特别是调用层次多的时候。RESTful和RPC的比较也是一个很有意 思的话题。

一般REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要求，只要封装了HTTP的

SDK就能调用，所以相对使用的广一些。RPC也有自己的优点，传输协议更高效，安全更可控，特别在一个公司内部，如果有统一个 的开发规范和统一的服务框架时，

他的开发效率优势更明显些。就看各自的技术积累实际条件，自己的选择了。

而异步消息的方式在分布式系统中有特别广泛的应用，他既能减低调用服务之间的耦合，又能成为调用之间的缓冲，确保消息积压不会冲垮被调用方，同时能保证调用方的

服务体验，继续干自己该干的活，不至于被后台性能拖慢。不过需要付出的代价是一致性的减弱，需要接受数据最终一致性；还有就是后台服务一般要 实现幂等性，因为消息

发送出于性能的考虑一般会有重复（保证消息的被收到且仅收到一次对性能是很大的考验）；最后就是必须引入一个独立的broker，如果公司内部没有技术积累，

对broker分布式管理也是一个很大的挑战。

3、如此多的服务，如何实现?

在微服务架构中，一般每一个服务都是有多个拷贝，来做负载均衡。一个服务随时可能下线，也可能应对临时访问压力增加新的服务节点。服务之间如何相互感知？服务如何管理?

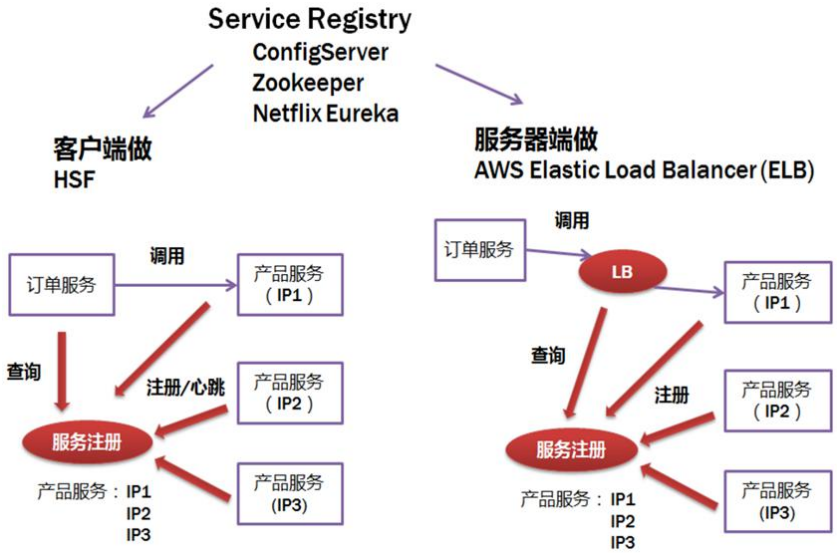
这就是服务发现的问题了。一般有两类做法，也各有优缺点。基本都是通过zookeeper等类似技术做服务注册信息的分布式管理。当服务上线时，服务提供者将自己的服务信息

注册到ZK（或类似框架），并通过心跳维持长链接，实时更新链接信息。服务调用者通过ZK寻址，根据可定制算法， 找到一个服务，还可以将服务信息缓存在本地以提高性能。

当服务下线时，ZK会发通知给服务客户端。

客户端做：优点是架构简单，扩展灵活，只对服务注册器依赖。缺点是客户端要维护所有调用服务的地址，有技术难度，一般大公司都有成熟的内部框架支持，比如Dubbo。

服务端做：优点是简单，所有服务对于前台调用方透明，一般在小公司在云服务上部署的应用采用的比较多。



4、服务挂了，如何解决

前面提到，Monolithic方式开发一个很大的风险是，把所有鸡蛋放在一个篮子里，一荣俱荣，一损俱损。而分布式最大的特性就是网络是不可靠的。通过微服务拆分能降低这个风险，

不过如果没有特别的保障，结局肯定是噩梦。所以当我们的系统是由一系列的服务调用链组成的时候，我们必须确保任一环节出问题都不至于影响整体链路。**相应的手段有很多**：

①重试机制

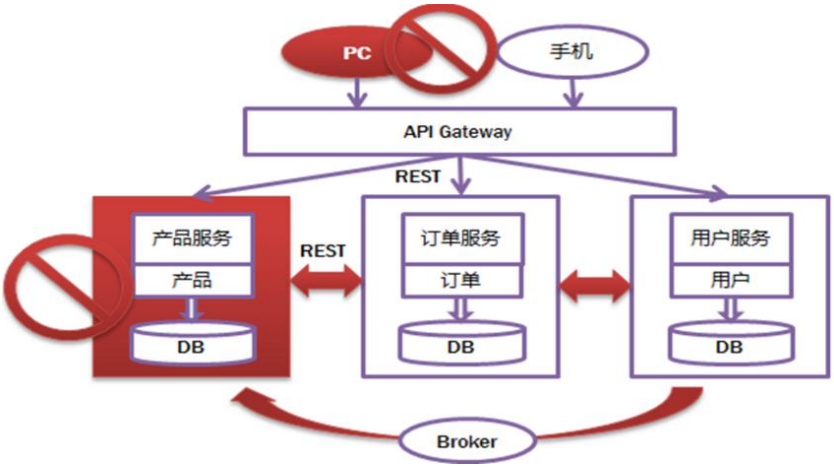
②限流

③熔断机制

④负载均衡

⑤降级（本地缓存）

这些方法基本都很明确通用，比如Netflix的Hystrix：https://github.com/Netflix/Hystrix



七、常见的设计模式和应用

有一个图非常好的总结微服务架构需要考虑的问题，包括：

1、API Gateway

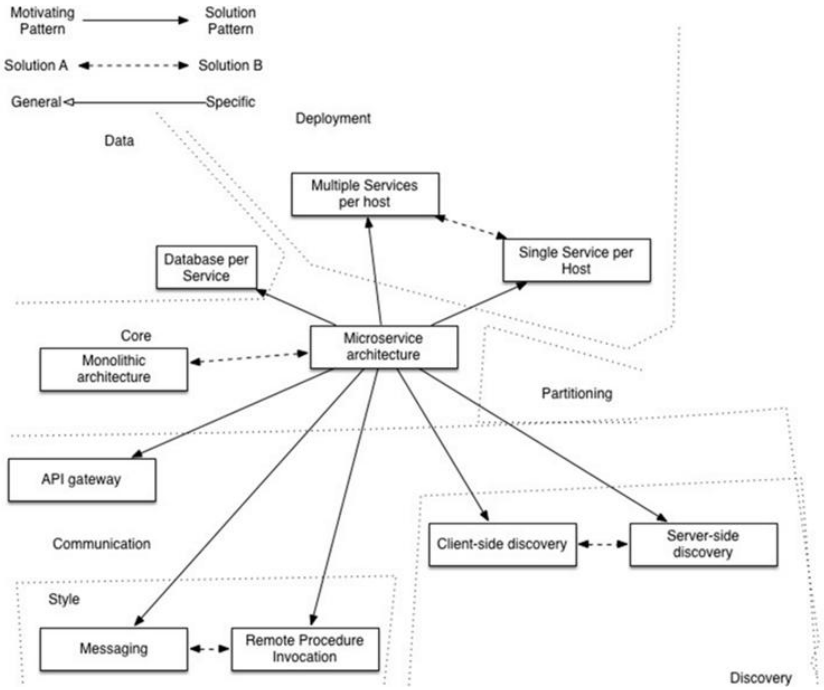
2、服务间调用

3、服务发现

4、服务容错

5、服务部署

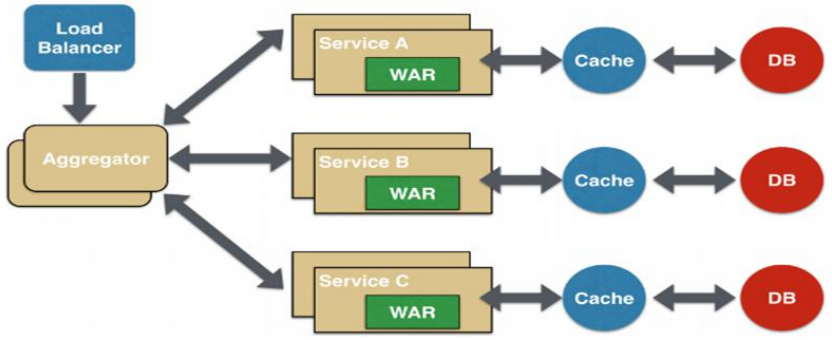
6、数据调用



六种常见的微服务架构设计模式：

1、聚合器微服务设计模式

这是一种最常见也最简单的设计模式：

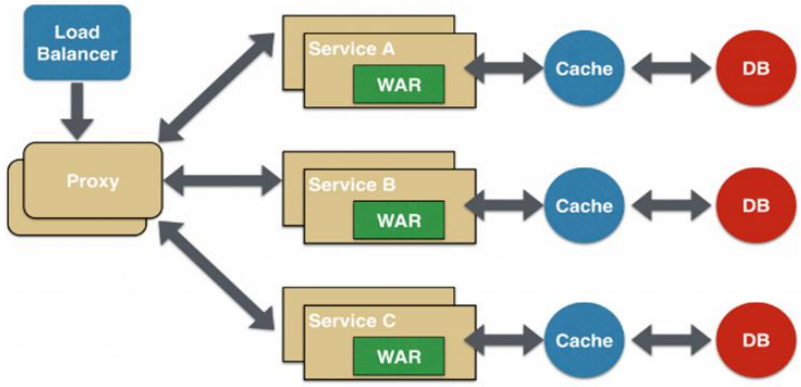


聚合器调用多个服务实现应用程序所需的功能。它可以是一个简单的Web页面，将检索到的数据进行处理展示。它也可以是一个更高层次的组合微服务，对检索到的数据增加业务逻辑后进一步

发布成一个新的微服务，这符合DRY原则。另外，每个服务都有自己的缓存和数据库。如果聚合器是一个组合服务，那么它也有自己的缓存和数据库。聚合器可以沿X轴和Z轴独立扩展。

2、代理微服务设计模式

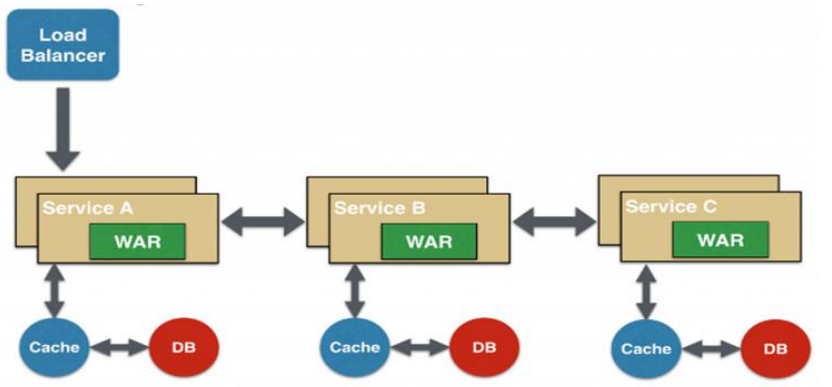
这是聚合模式的一个变种，如下图所示：



在这种情况下，客户端并不聚合数据，但会根据业务需求的差别调用不同的微服务。代理可以仅仅委派请求，也可以进行数据转换工作。

3、链式微服务设计模式

这种模式在接收到请求后会产生一个经过合并的响应，如下图所示：

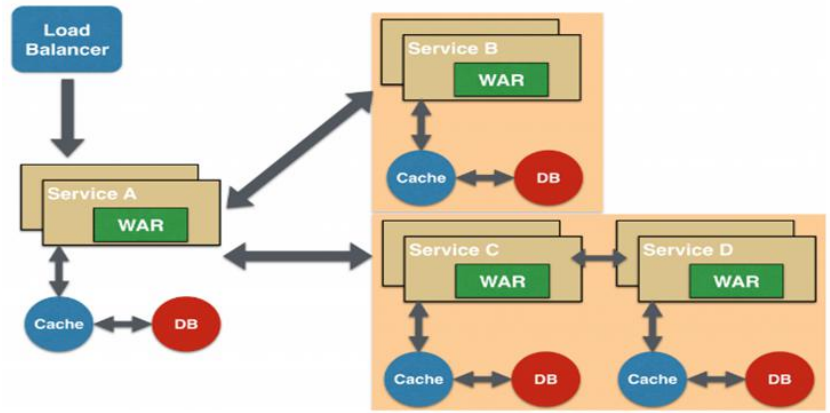


在这种情况下，服务A接收到请求后会与服务B进行通信，类似地，服务B会同服务C进行通信。所有服务都使用同步消息传递。在整个链式调用完成之前，客户端会一直阻塞。

因此，服务调用链不宜过长，以免客户端长时间等待。

4、分支微服务设计模式

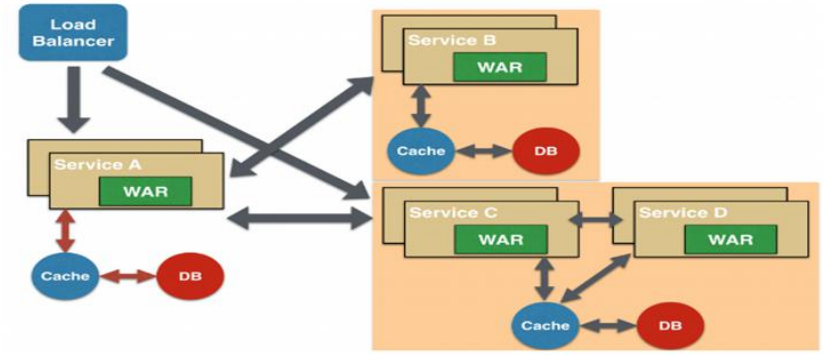
这种模式是聚合器模式的扩展，允许同时调用两个微服务链，如下图所示：



5、**数据共享微服务设计模式**

自治是微服务的设计原则之一，就是说微服务是全栈式服务。但在重构现有的"单体应用（monolithic application）"时，SQL数据库反规范化可能会导致数据重复和不一致。

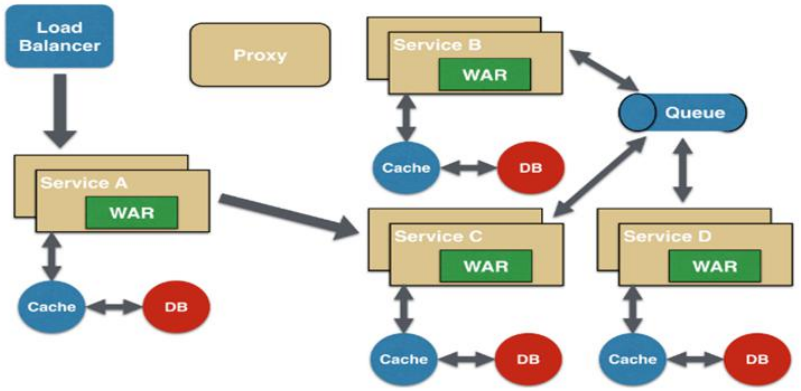
因此，在单体应用到微服务架构的过渡阶段，可以使用这种设计模式，如下图所示：



在这种情况下，部分微服务可能会共享缓存和数据库存储。不过，这只有在两个服务之间存在强耦合关系时才可以。对于基于微服务的新建应用程序而言，这是一种反模式。

6、异步消息传递微服务设计模式

虽然REST设计模式非常流行，但它是同步的，会造成阻塞。因此部分基于微服务的架构可能会选择使用消息队列代替REST请求/响应，如下图所示：



八、优点和缺点

1、微服务的优点：

关键点：复杂度可控，独立按需扩展，技术选型灵活，容错，可用性高

①它解决了复杂性的问题。它会将一种怪异的整体应用程序分解成一组服务。虽然功能总量不变，但应用程序已分解为可管理的块或服务。每个服务都以RPC或消息驱动的API的

形式定义了一个明确的边界；Microservice架构模式实现了一个模块化水平。

②这种架构使每个服务都能够由专注于该服务的团队独立开发。开发人员可以自由选择任何有用的技术，只要该服务符合API合同。当然，大多数组织都希望避免完全无政府状态并

限制技术选择。然而，这种自由意味着开发人员不再有义务使用在新项目开始时存在的可能过时的技术。在编写新服务时，他们可以选择使用当前的技术。此外，由于服务相对较小，

因此使用当前技术重写旧服务变得可行。

③Microservice架构模式使每个微服务都能独立部署。开发人员不需要协调部署本地服务的变更。这些变化可以在测试后尽快部署。例如，UI团队可以执行A | B测试，并快速迭代

UI更改。Microservice架构模式使连续部署成为可能。

④Microservice架构模式使每个服务都可以独立调整。您可以仅部署满足其容量和可用性限制的每个服务的实例数。此外，您可以使用最符合服务资源要求的硬件。

2、微服务的缺点

关键点（挑战）：多服务运维难度，系统部署依赖，服务间通信成本，数据一致性，系统集成测试，重复工作，性能监控等

①一个缺点是名称本身。术语microservice过度强调服务规模。但重要的是要记住，这是一种手段，而不是主要目标。微服务的目标是充分分解应用程序，以便于敏捷应用程序开发和部署。

②微服务器的另一个主要缺点是分布式系统而产生的复杂性。开发人员需要选择和实现基于消息传递或RPC的进程间通信机制。此外，他们还必须编写代码来处理部分故障，

因为请求的目的地可能很慢或不可用。

③微服务器的另一个挑战是分区数据库架构。更新多个业务实体的业务交易是相当普遍的。但是，在基于微服务器的应用程序中，您需要更新不同服务所拥有的多个数据库。使用分布式事务

通常不是一个选择，而不仅仅是因为CAP定理。许多今天高度可扩展的NoSQL数据库都不支持它们。你最终不得不使用最终的一致性方法，这对开发人员来说更具挑战性。

④测试微服务应用程序也更复杂。服务类似的测试类将需要启动该服务及其所依赖的任何服务（或至少为这些服务配置存根）。再次，重要的是不要低估这样做的复杂性。

⑤Microservice架构模式的另一个主要挑战是实现跨越多个服务的更改。例如，我们假设您正在实施一个需要更改服务A，B和C的故事，其中A取决于B和B取决于C.在单片应用程序中，

您可以简单地更改相应的模块，整合更改，并一次性部署。相比之下，在Microservice架构模式中，您需要仔细规划和协调对每个服务的更改。例如，您需要更新服务C，然后更新服务B，

然后再维修A.幸运的是，大多数更改通常仅影响一个服务，而需要协调的多服务变更相对较少。

⑥部署基于微服务的应用程序也更复杂。单一应用程序简单地部署在传统负载平衡器后面的一组相同的服务器上。每个应用程序实例都配置有基础架构服务（如数据库和消息代理）

的位置（主机和端口）。相比之下，微服务应用通常由大量服务组成。例如，每个服务将有多个运行时实例。更多的移动部件需要进行配置，部署，扩展和监控。此外，您还需要实现服务

发现机制，使服务能够发现需要与之通信的任何其他服务的位置（主机和端口）。传统的基于故障单和手动操作的方法无法扩展到这种复杂程度。因此，成功部署微服务应用程序需要

开发人员更好地控制部署方法，并实现高水平的自动化。

九、思考：意识的转变

微服务对我们的思考，更多的是思维上的转变。对于微服务架构：**技术上不是问题，意识比工具重要。**

关于微服务的几点设计出发点：

1、应用程序的核心是业务逻辑，按照业务或客户需求组织资源（这是最难的）

2、做有生命的产品，而不是项目

3、头狼战队，全栈化

4、后台服务贯彻Single Responsibility Principle（单一职责原则）

5、VM->Docker （to PE）

6、DevOps (to PE)