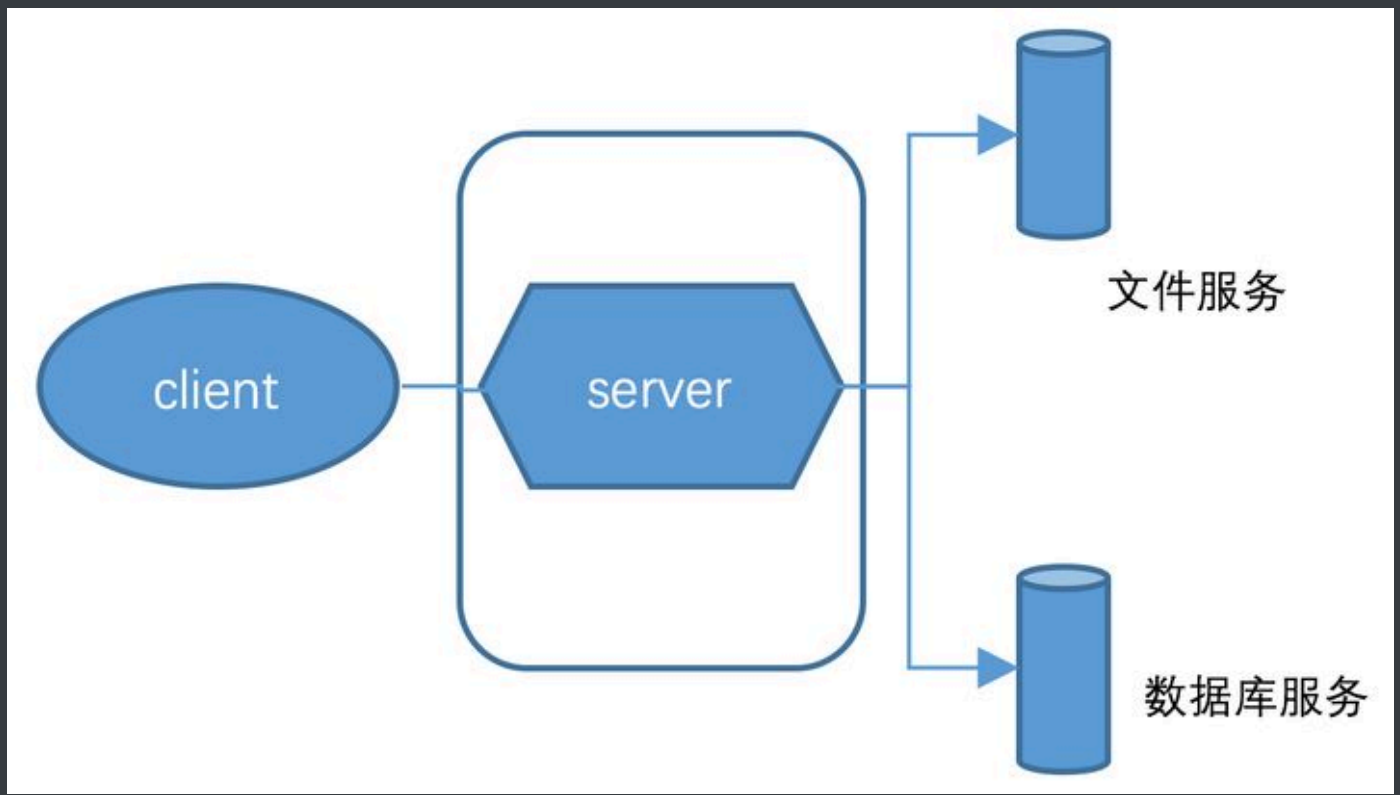
互联网架构演进之路

互联网产品常常面临庞大的用户量，日均数十亿PV的高并发，PB级别的数据存储等问题的挑战，同时要求保证系统的高可用和弹性伸缩，并且能够根据需要进行快速迭代扩展，这些都对于系统架构提出了很高的要求。

互联网架构从简到繁的演进经历了单体架构-集群架构-分布式架构-微服务架构以及最新的service mesh的演进过程。

01. 简单架构

网站架构的发展也是由简到繁，早期互联网产品用户量少，并发量低，数据量小，多数只需要单个应用服务器可以满足需要，而数据库和文件服务部署在外部单个服务器上，这就是最早互联网架构，架构图如下所示。

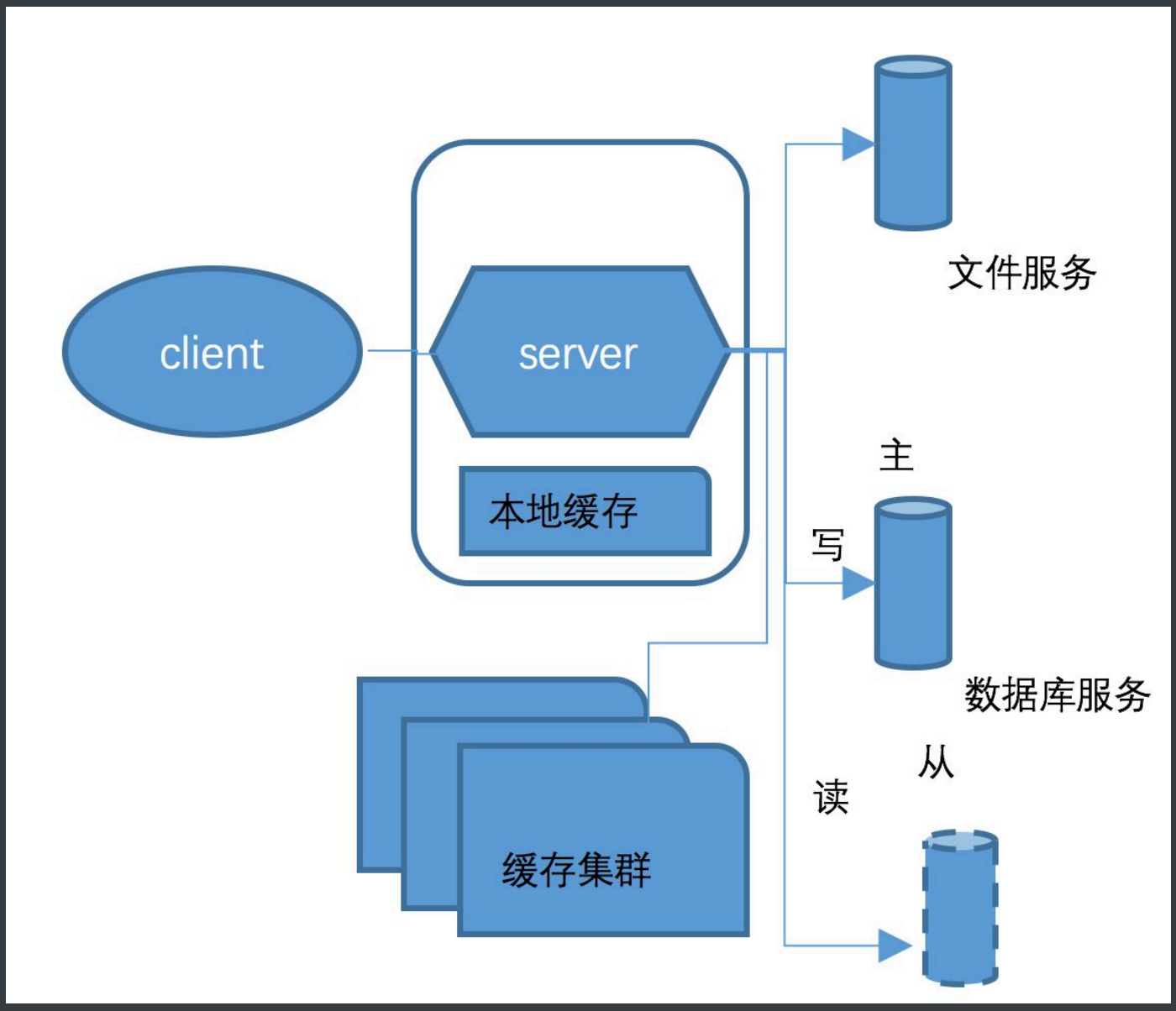


在该架构所有业务逻辑均由单个server处理。并且文件和数据库只部署在外部的单个server中，该架构中存在的问题也比较明显，server应用、文件服务、数据库均为单体服务，缺乏故障转移，在升级维护中必须进行停机，可用性低。在现在的业务场景中已经不再适用。

02.缓存与读写分离

随着访问量的激增，数据量的快速膨胀，对应用服务器和数据库服务器的IO性能要求也越来越高。单体服务总是资源有限，而单个数据库的数据流量越来越大，数据的访问性能也明显下降。 为了进一步降低数据库的访问压力，对数据库进行读写分离、分库分表等优化。 读写分离即将数据库分为主从数据库，从库实时同步主库的更新内容。写请求访问主数据库，读请求访问从数据库。分库分表即是把原本存储于一个库的数据分块存储到多个库上，把原本存储于一个表的数据分块存储到多个表上。不论是读写分离还是分库分表都有效分解了数据库访问压力。

基于二八定律，即大部分的业务数据只集中于数据中的一小部分，而有些数据是需要经常读取，但是更新很少，或是访问量非常大的数据块，这些情况下我们就需要引入缓存层, 如果访问命中缓存，既能减小数据库的访问压力，又可以提高数据访问性能。缓存的发展也是由简单到复杂，由单体到缓存集群，由单点到高可用。本地缓存的特点就是可以提供快速访问，但无法实现共享，无法保证高可用。而缓存集群可以提供高可用、可共享、大容量的存储，缓存层降低数据库的读写IO，有效提升系统响应能力。

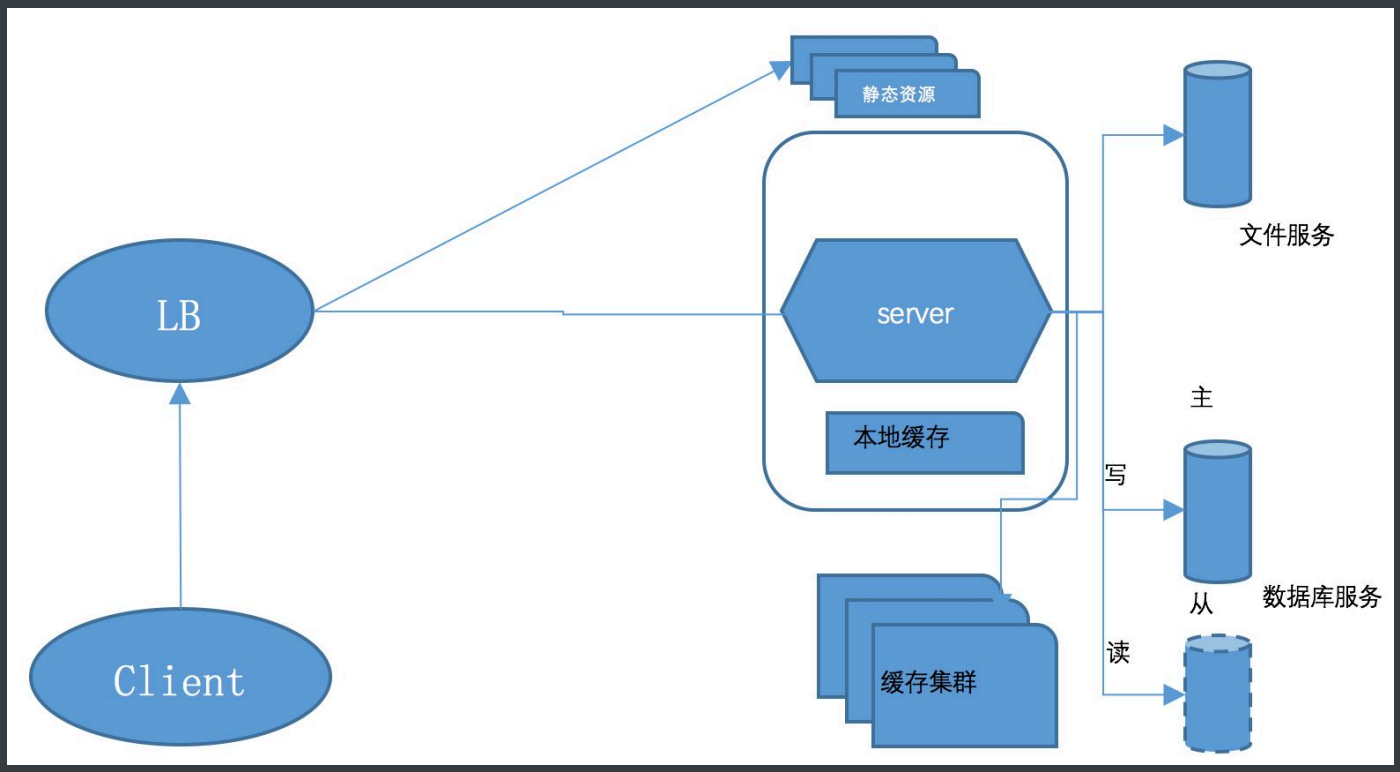


03.动静分离

随着业务的进一步爬坡，需要能够进一步降低服务器的压力，这时候可以采用动静分离技术。动静分离技术是将服务的静态资源与后台应用进行分开部署，提高用户访问静态资源的速度，降低对后台的访问压力。

静态资源一般放在CDN上，部署在网络提供商的机房。用户在访问静态资源时，可以很好的利用CDN的优点，从距离自己最近的网络提供商机房获取数据。

动静分离后，带来的优点是显而易见的：后端应用更为服务化，只需要提供api接口即可，前端可以通过更加专业的技术提高访问效率。但同时静态文件的缓存更新以及前后端的更新成本都是动静分离所带来的问题。

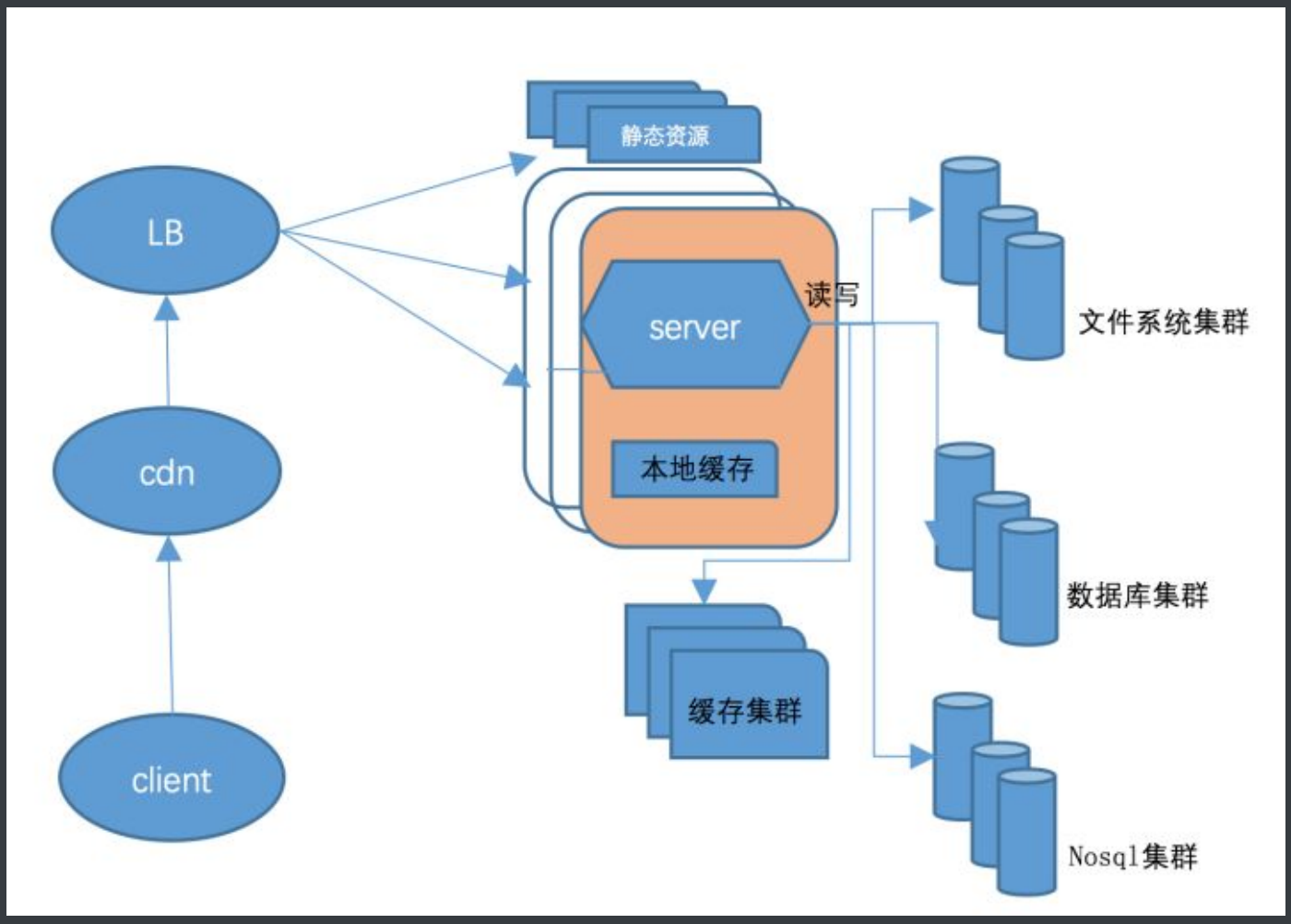


04.集群化高可用架构

随着用户规模和业务量的不断上涨，单个应用服务器将出现性能瓶颈，对于PB级的数据和高并发用户大流量访问，单一或者主备的数据库、文件系统都已经不能满足需求，需要集群化来分担负载。当数据规模达到一定规模，传统关系型数据库性能下滑非常严重，通过分库分表也难以应对，为了支撑海量数据和流量，出现了NoSql数据库，它关注对数据高并发地读写和对海量数据的存储。

同时应用服务器从单体变为集群，客户端也不再是直接接入后端应用，而是转而通过负载均衡设备代理后端多个服务器集群，一方面将访问压力分摊到了多个后端应用上，单个应用不再是性能瓶颈，另一方面可以实现服务路由，以及异常熔断等特性。

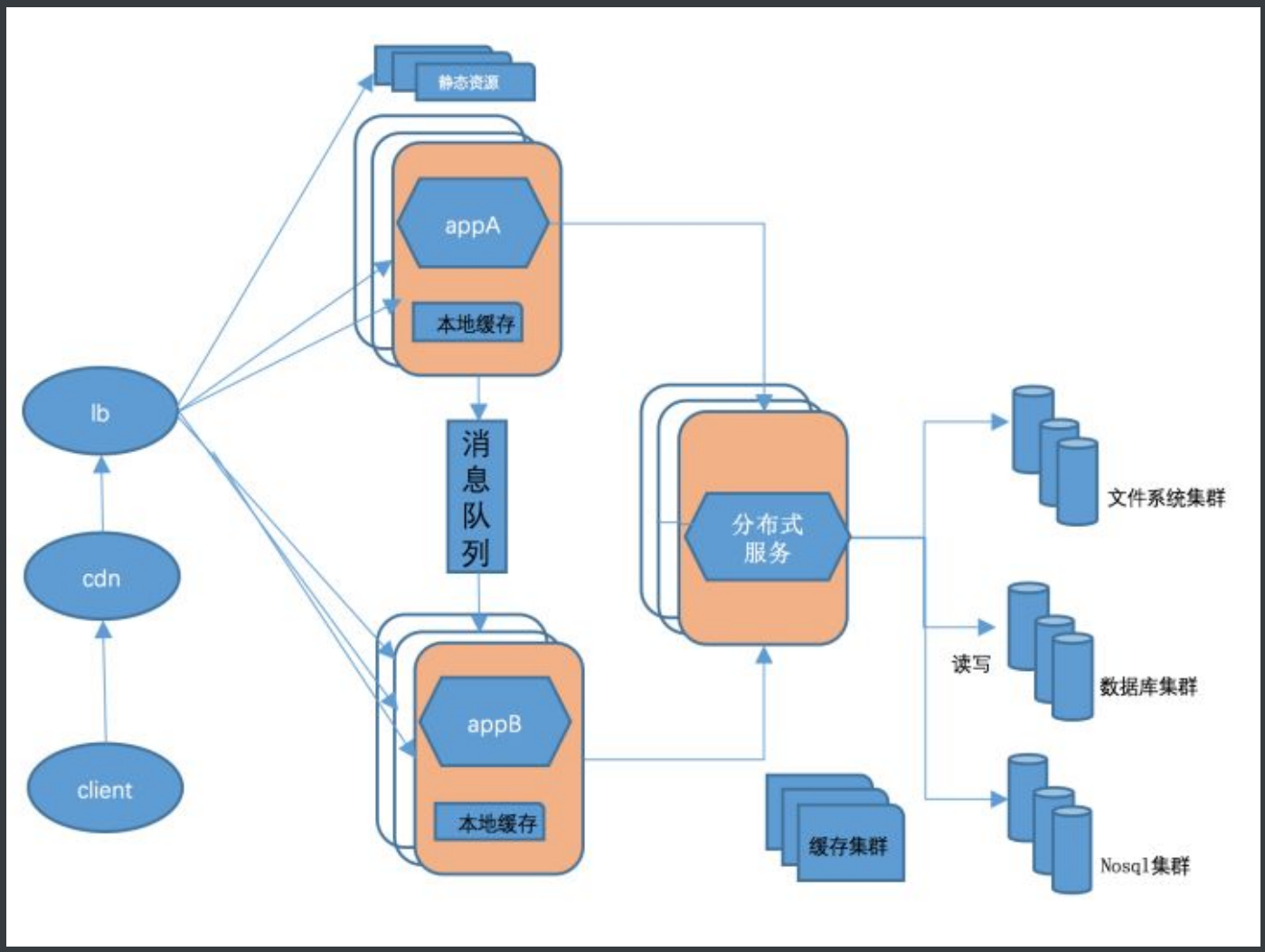
为了进一步降低服务端压力，降低客户端访问延迟，客户端与负载均衡设备之间加入CDN对静态资源进行缓存加速，利用GSLB调度体系，将用户请求精准调度至最优接入节点。以达到最优的访问性能。



05.业务拆分与分布式

同时随着业务场景的复杂化，存储规模越来越庞大。将业务进行拆分并分而治之成为更好的选择。大型的业务场景被细分为单个更小的业务子场景，按照系统业务功能进行划分，例如对于电商系统，按功能维度我们可以拆分为商品中心，订单中心，用户中心，购物车，结算等功能模块。每一个子模块由不同的业务团队负责。每个子业务模块进行独立开发、部署、运维。

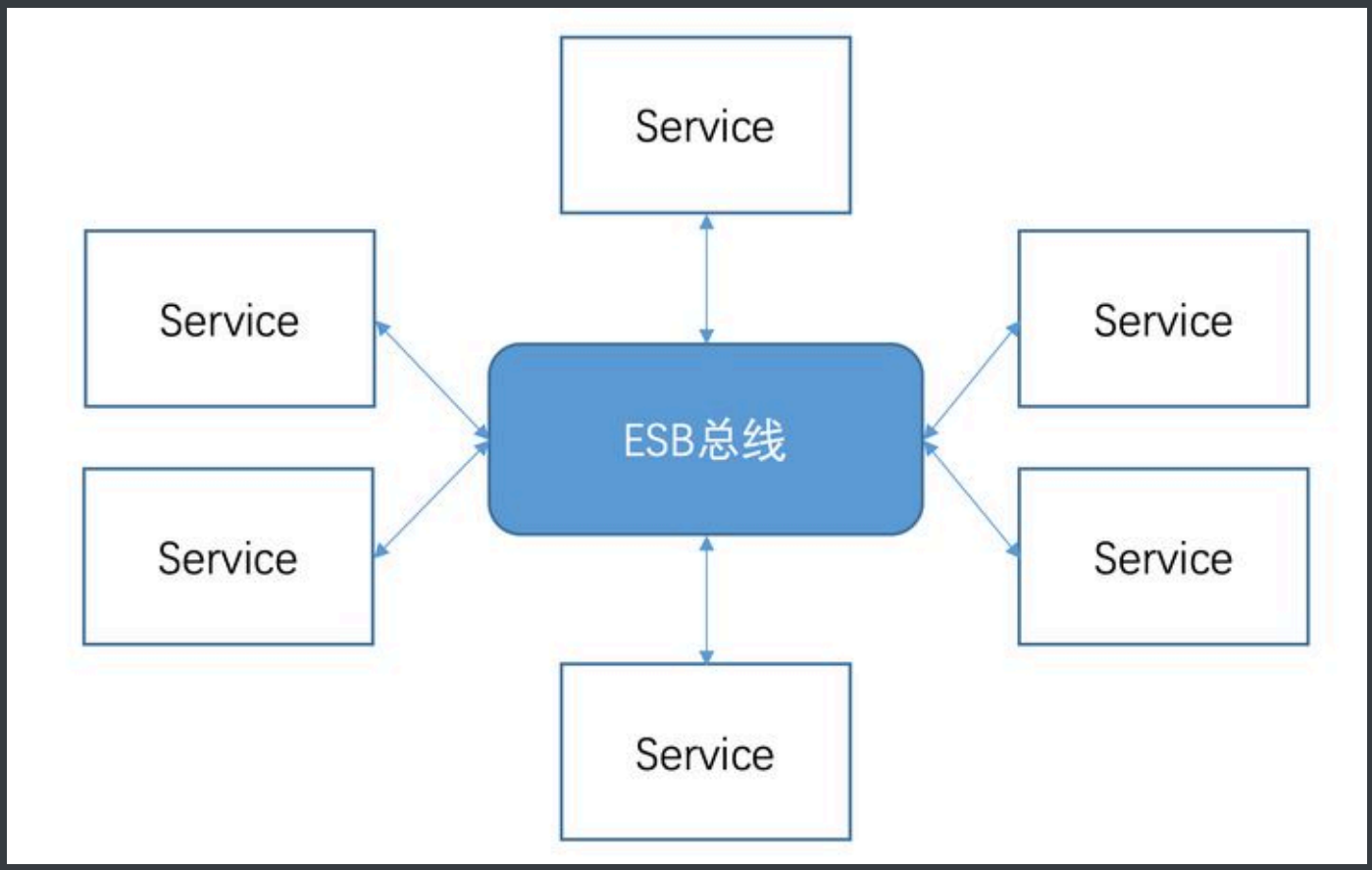
随着对业务场景越来越细的划分，模块越来越多，整个应用的复杂度也越来越高，大量的独立子应用对数据库的独立访问，导致后端数据库的压力越来越大，需要将各个子应用的重叠逻辑再进行抽取为独立子服务，子应用服务之间通过RPC或者消息系统进行相互通信，因此出现了分布式应用，到目前为止的分布式架构已经能够应对大部分高并发，大流量的业务场景。



06.SOA

分布式架构出现之后，随着应用的更细力度的拆分，逐渐呈现了一个相互依赖、公共的模块，这些模块大都依赖于相同的逻辑或者功能代码。这时候就需要把这些应用的公用模块提出来，采用独立部署统一管理的方式，提高重用度，这就是服务导向架构（SOA）。

SOA其实是一种理念，它的主要特性：面向服务的计算，服务之间松散耦合，支持服务的封装，服务注册和自动发现。但是SOA并没有定义出具体的实现方式，目前一种主要的实现方式是企业服务总线（Enterprise Service Bus，ESB），ESB的根本目标是解决异构系统的连通性，通过协议转换、消息解析、消息路由把服务提供者的数据传递到服务消费者。所以SOA的中心化虽然重，但是会解决一些公用逻辑的问题。



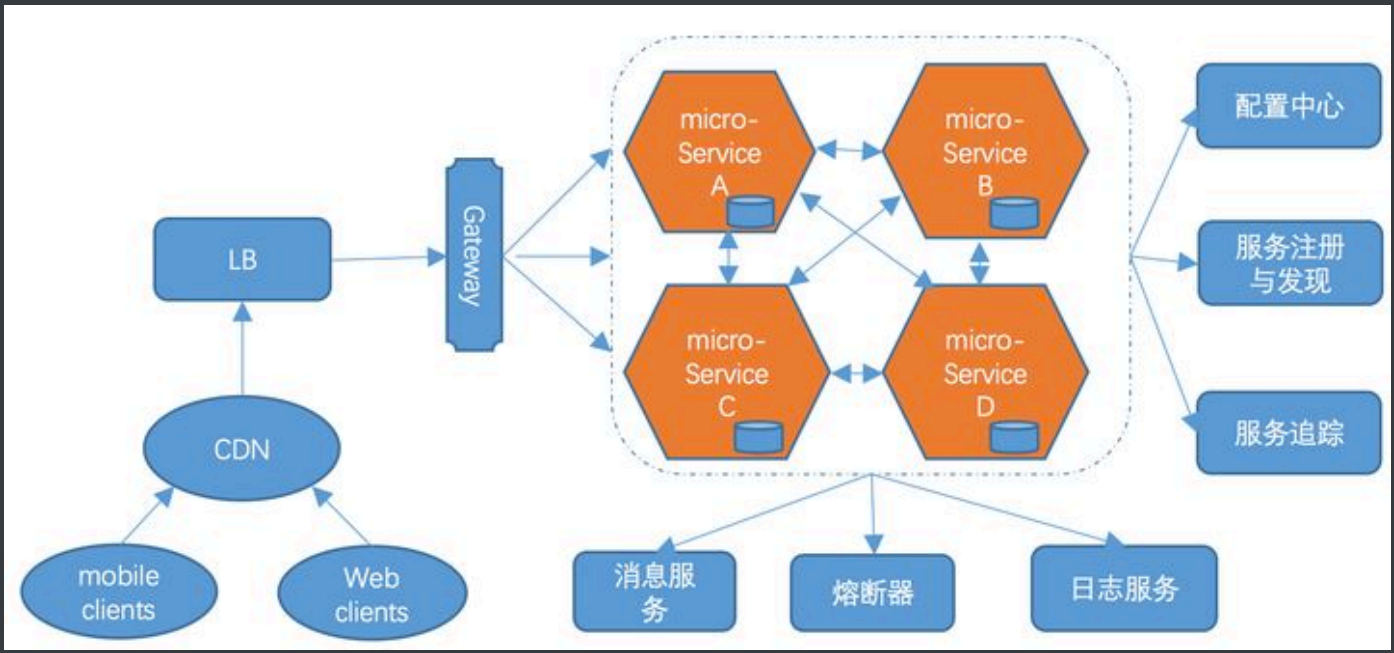
07.微服务

基于SOA的系统架构实现了松耦合，系统之间通过服务接口（Service API）和中心化管理的企业服务总线（ESB）进行交互, 但这种拆分往往是业务系统的一种垂直拆分，拆分的子系统随着业务的复杂耦合仍然面临难以开发和维护的问题。因此更细粒度的拆分变得必要，将子系统功能进一步拆分，变成一项项的服务。系统分布式架构，服务去中心化实现，也即微服务的思想。

敏捷开发专家Martin Fowler 在2014年定义了微服务架构。微服务架构风格是一个系统，由一套独立运行的微服务组成， 这些服务是围绕业务功能构建的，服务关注单一业务，服务间采用轻量级的通信机制（通常是HTTP资源的API），可以全自动独立部署，可以使用不同的编程语言和数据存储技术。

微服务架构通过业务拆分实现服务组件化，通过组件组合快速开发系统，业务单一的服务组件又可以独立部署，使得整个系统变得清晰灵活。但是大量的分布式服务又使得架构实现面临问题，如服务注册发现，服务统一接入和权限控制，服务的负载均衡，服务配置的集中管理，服务熔断，服务监控等。所以在微服务架构中除了业务服务组件外通常会引入服务注册发现组件来进行服务治理，引入服务网关组件来提供统一入口和权限控制，引入负载均衡组件来提供客

户端或服务器端的负载均衡，引入集中配置组件来提供服务集中管理，引入熔断器组件来提供服务熔断，引入服务追踪组件来提供服务监控等。因此微服务架构是由这些基础的服务组件和业务微服务组件共同组成。基本的微服务架构如图所示：

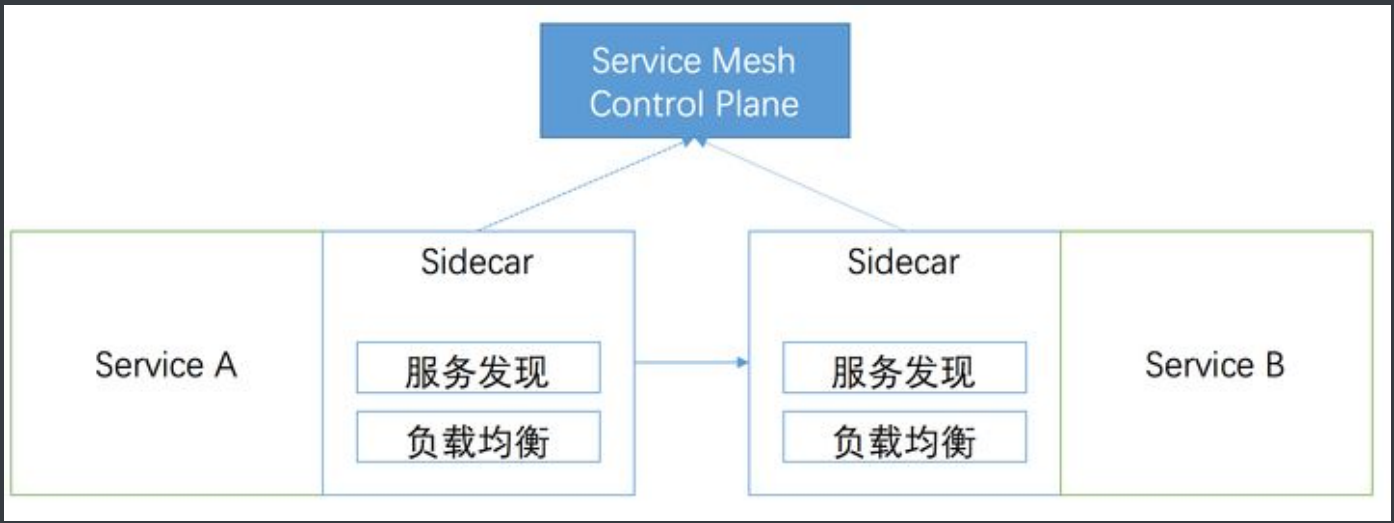


微服务架构在业界已经有了很多成熟的实践， Netflix推出了经过生产验证的NetflixOSS，Pivotal 将 NetflixOSS 开源微服务组件集成到其 Spring 体系，推出 Spring Cloud，阿里推出了服务治理能力丰富的生产级分布式服务框架Dubbo，谷歌推出了基于 protobuf 的强契约编程模型的 RPC 框架gRPC。 服务发现组件除dubbo外还有Eureka，Consul和Zookeeper, 服务网关组件有Spring Cloud体系的Zuul，基于 Nginx/OpenResty 的 API 网关 Kong， 配置中心组件有Spring Cloud 自带 Spring Cloud Config和携程的Apollo，服务监控组件有Zipkin、Pinpoint和CAT，服务熔断组件有Netflix 的 Hystrix等。

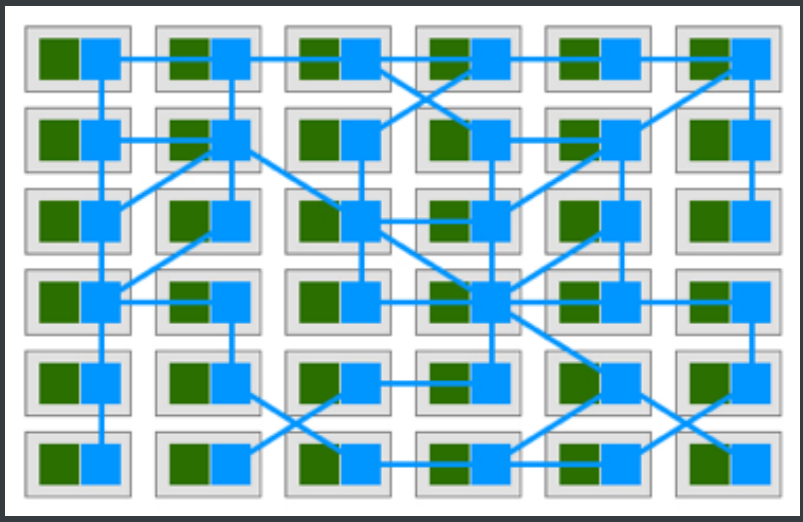
微服务的交付除了依赖成熟的框架构建系统外，自动化部署也需要自动化部署工具的支持。容器化技术已经成为自动化部署微服务的一种理想手段。Docker 和其他类似的 Linux 容器技术使得软件开发和持续交付变得很容易，大大降低了微服务运维的难度，也使得系统宕机风险大大降低。

采用微服务架构后，项目可以快速迭代与持续交付。但是也带了一些弊端，开发人员除了需要关注业务逻辑实现外还需要考虑业务的一系列问题，比如服务注册，服务发现，服务通讯，负载均衡，服务熔断，服务超时等，这些是非常重要的。大多数时候，我们需要依赖第三方库或者组件来提供这些服务，像上文提到的Hystrix, Eureka这些组件，在其服务组织中起到了广泛的应用。但这些组件的使用是侵入式的，这要求开发人员需要额外的精力去关注一些非业务逻辑层面的事情。而服务网格架构（Service Mesh）的出现，让业务开发人员得以真正解脱。

服务网格架构是在微服务架构盛行的今天的必然产物，服务网格架构可以说是微服务架构的一种实现，将开发人员从服务之间的交互中释放出来，专心维护业务逻辑。服务网格是一个基础设施层，用于代理服务间的通信，来负责消息直接按需传递，开发人员不需要在考虑服务发现等非业务逻辑，服务的治理也上升到统一的层面。服务网格对业务是相对透明的，与业务始终部署在一起。



上图是一个简单的服务网格例子，不同的业务之间通过服务网格进行通讯，服务网格根据预先定义好的规则进行流量治理：可以配置A/B测试，金丝雀发布，同样也可以根据服务网格的进行流量的统计。



借用服务网格的一个经典的图例，在微服务大量部署之后，每个服务节点的sidecar进程自然就形成了一个网格。

总而言之，微服务架构实现了业务的解耦，使系统具备了良好的扩展性和伸缩性，单个服务组件可以独立部署，便于组件的开发测试部署和维护，服务自治也可以将数据管理去中心化，容器化部署平台也降低了测试、部署和运维的难度，使软件开发迭代更加敏捷快速，但分布式系统固有的难题（如分布式事务等）在微服务架构中也依然存在。