一、　　单体架构

单体架构比较初级，典型的三级架构，前端(Web/手机端)+中间业务逻辑层+数据库层。这是一种典型的 Java Spring mvc 或者 Python Drango 框架的应用。

单体架构的应用比较容易部署、测试，　在项目的初期，单体应用可以很好地运行。然而，随着需求的不断增加，　越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。下面是单体架构应用的一些缺点：复杂性高：　以一个百万行级别的单体应用为例，整个项目包含的模块非常多、模块的边界模糊、　依赖关系不清晰、　代码质量参差不齐、　混乱地堆砌在一起。可想而知整个项目非常复杂。　每次修改代码都心惊胆战，　甚至添加一个简单的功能，　或者修改一个 Bug 都会带来隐含的缺陷。技术债务：　随着时间推移、需求变更和人员更迭，会逐渐形成应用程序的技术债务，并且越积　越多。“ 不坏不修”，这在软件开发中非常常见，在单体应用中这种思想更甚。　已使用的系统设计或代码难以被修改，因为应用程序中的其他模块可能会以意料之外的方式使用它。部署频率低：　随着代码的增多，构建和部署的时间也会增加。而在单体应用中，　每次功能的变更或缺陷的修复都会导致需要重新部署整个应用。全量部署的方式耗时长、　影响范围大、　风险高，　这使得单体应用项目上线部署的频率较低。　而部署频率低又导致两次发布之间会有大量的功能变更和缺陷修复，出错率比较高。可靠性差：某个应用 Bug，例如死循环、内存溢出等，　可能会导致整个应用的崩溃。扩展能力受限：单体应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的 CPU；　有的模块则是 IO 密集型的，需要更大的内存。　由于这些模块部署在一起，不得不在硬件的选择上做出妥协。阻碍技术创新：　单体应用往往使用统一的技术平台或方案解决所有的问题，　团队中的每个成员　都必须使用相同的开发语言和框架，要想引入新框架或新技术平台会非常困难。二、分布式应用中级架构，分布式应用，中间层分布式+数据库分布式，是单体架构的并发扩展，将一个大的系统划分为多个业务模块，业务模块分别部署在不同的服务器上，各个业务模块之间通过接口进行数据交互。数据库也大量采用分布式数据库，如redis、ES、solor等。通过LVS/Nginx代理应用，将用户请求均衡的负载到不同的服务器上。

二、分布式架构

该架构相对于单体架构来说，这种架构提供了负载均衡的能力，大大提高了系统负载能力，解决了网站高并发的需求。另外还有以下特点：降低了耦合度：把模块拆分,使用接口通信,降低模块之间的耦合度。责任清晰：把项目拆分成若干个子项目,不同的团队负责不同的子项目。扩展方便：增加功能时只需要再增加一个子项目,调用其他系统的接口就可以。部署方便:可以灵活的进行分布式部署。提高代码的复用性：比如 service 层,如果不采用分布式 rest 服务方式架构就会在手机 wap 商城,微信商城,pc,android，ios 每个端都要写一个 service 层逻辑,开发量大,难以维护一起升级,这时候就可以采用分布式 rest 服务方式,公用一个 service 层。缺点 ：系统之间的交互要使用远程通信,接口开发增大工作量,但是利大于弊。三、微服务架构微服务架构，主要是中间层分解，将系统拆分成很多小应用（微服务），微服务可以部署在不同的服务器上，也可以部署在相同的服务器不同的容器上。当应用的故障不会影响到其他应用，单应用的负载也不会影响到其他应用，其代表框架有 Spring cloud、Dubbo 等。

三、微服务架构
易于开发和维护： 一个微服务只会关注一个特定的业务功能，所以它业务清晰、代码量较少。 开发和维护单个微服务相对简单。而整个应用是由若干个微服务构建而成的，所以整个应用也会被维持在一个可控状态。 单个微服务启动较快： 单个微服务代码量较少， 所以启动会比较快。 局部修改容易部署： 单体应用只要有修改，就得重新部署整个应用，微服务解决了这样的问题。 一般来说，对某个微服务进行修改，只需要重新部署这个服务即可。技术栈不受限：在微服务架构中，可以结合项目业务及团队的特点，合理地选择技术栈。例如某些服务可使用关系型数据库 MySQL；某些微服务有图形计算的需求，可以使用 Neo4j；甚至可根据需要，部分微服务使用 Java 开发，部分微服务使用 Node.js 开发。微服务虽然有很多吸引人的地方，但它并不是免费的午餐，使用它是有代价的。使用微服务架构面临的挑战。运维要求较高：更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行。而在微服务中，需要保证几十甚至几百个服务服务的正常运行与协作，这给运维带来了很大的挑战。分布式固有的复杂性：使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务等都会带来巨大的挑战。接口调整成本高：微服务之间通过接口进行通信。如果修改某一个微服务的 API，可能所有使用了该接口的微服务都需要做调整。重复劳动：很多服务可能都会使用到相同的功能，而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个服务都会开发这一功能，从而导致代码重复。尽管可以使用共享库来解决这个问题（例如可以将这个功能封装成公共组件，需要该功能的微服务引用该组件），但共享库在多语言环境下就不一定行得通了。
四、Serverless 架构当我们还在容器的浪潮中前行时，已经有一些革命先驱悄然布局另外一个云计算战场：Serverless 架构。

Serverless 架构
2014 年 11 月 14 日，亚马逊 AWS 发布了新产品 Lambda。当时 Lambda 被描述为：一种计算服务，根据时间运行用户的代码，无需关心底层的计算资源。从某种意义上来说，Lambda 姗姗来迟，它像云计算的 PaaS 理念：客户只管业务，无需担心存储和计算资源。在此前不久，2014 年 10 月 22 日，谷歌收购了实时后端数据库创业公司 Firebase。Firebase 声称开发者只需引用一个 API 库文件就可以使用标准 REST API 的各种接口对数据进行读写操作，只需编写 HTML + CSS + JavaScrip 前端代码，不需要服务器端代码（如需整合，也极其简单）。相对于上两者，Facebook 在 2014 年二月收购的 Parse，则侧重于提供一个通用的后台服务。这些服务被称为 Serverless 或 no sever。想到 PaaS（平台即服务）了是吗？很像，用户不需要关心基础设施，只需要关心业务，这是迟到的 PaaS，也是更实用的 PaaS。这很有可能将会变革整个开发过程和传统的应用生命周期，一旦开发者们习惯了这种全自动的云上资源的创建和分配，或许就再也回不到那些需要微应用配置资源的时代里去了。Serverless 架构能够让开发者在构建应用的过程中无需关注计算资源的获取和运维，由平台来按需分配计算资源并保证应用执行的 SLA（服务等级协议），按照调用次数进行计费，有效的节省应用成本。ServerLess 的架构如上图所示。其优点如下所示：低运营成本：在业务突发性极高的场景下，系统为了应对业务高峰，必须构建能够应对峰值需求的系统，这个系统在大部分时间是空闲的，这就导致了严重的资源浪费和成本上升。在微服务架构中，服务需要一直运行，实际上在高负载情况下每个服务都不止一个实例，这样才能完成高可用性；在 Serverless 架构下，服务将根据用户的调用次数进行计费，按照云计算 pay-as-you-go 原则，如果没有东西运行，你就不必付款，节省了使用成本。同时，用户能够通过共享网络、硬盘、CPU 等计算资源，在业务高峰期通过弹性扩容方式有效的应对业务峰值，在业务波谷期将资源分享给其他用户，有效的节约了成本。简化设备运维：在原有的 IT 体系中，开发团队即需要维护

应用程序，同时还要维护硬件基础设施；Serverless 架构中，开发人员面对的将是第三方开发或自定义的 API 和 URL，底层硬件对于开发人员透明化了，技术团队无需再关注运维工作，能够更加专注于应用系统开发。提升可维护性：Serverless 架构中，应用程序将调用多种第三方功能服务，组成最终的应用逻辑。目前，例如登陆鉴权服务，云数据库服务等第三方服务在安全性、可用性、性能方面都进行了大量优化，开发团队直接集成第三方的服务，能够有效的降低开发成本，同时使得应用的运维过程变得更加清晰，有效的提升了应用的可维护性。更快的开发速度：这一点在现在互联网创业公司得到很好的体现，创业公司往往开始由于人员和资金等问题，不可能每个产品线都同时进行，这时候就可以考虑第三方的 Baas 平台，比如使用微信的用户认证、阿里云提供的 RDS，极光的消息推送，第三方支付及地理位置等等，能够很快进行产品开发的速度，把工作重点放在业务实现上，把产品更快的推向市场。但 ServerLess 架构也有其缺点：厂商平台绑定：平台会提供 Serverless 架构给大玩家，比如 AWS Lambda，运行它需要使用 AWS 指定的服务，比如 API 网关，DynamoDB，S3 等等，一旦你在这些服务上开发一个复杂系统，你会粘牢 AWS，以后只好任由他们涨价定价或者下架等操作，个性化需求很难满足，不能进行随意的迁移或者迁移的成本比较大，同时不可避免带来一些损失。Baas 行业内一个比较典型的事件，2016 年 1 月 19 日 Facebook 关闭曾经花巨额资金收购的 Parse，造成用户不得不迁移在这个平台中产生一年多的数据，无疑需要花费比较大的人力和时间成本。成功案例比较少，没有行业标准：目前的情况也只适合简单的应用开发，缺乏大型成功案例的推动。对于 Serverless 缺乏统一的认知以及相应的标准，无法适应所有的云平台。目前微服务架构在四种架构中处于主流地位，很多应用第一、第二种架构的企业也开始慢慢转向微服务架构。到目前为止微服务的技术相对于二三年前已经比较成熟，第四种架构将是未来发展的一种趋势。