新兴软件架构技术

软件架构是指软件系统的基本结构以及创建这种结构和系统的规程。每个结构都包含软件元素、它们之间的关系以及元素和关系的属性。软件系统的架构是一个隐喻，类似于建筑物的架构。它作为系统和开发项目的蓝图，布置设计团队需要执行的任务。

软件架构是指做出基本的结构选择，一旦实现，改变这些选择的代价是高昂的。软件架构选择包括软件设计中可能出现的特定结构选项。例如，控制航天飞机运载火箭的系统要求非常快和非常可靠。因此，需要选择合适的实时计算语言。此外，为了满足可靠性的需要，可以选择有多个冗余和独立生成的程序副本，并在交叉检查结果的同时在独立硬件上运行这些副本。

记录软件架构有助于利益相关者之间的沟通，捕获有关高级设计的早期决策，并允许在项目之间重用设计组件。

软件架构是复杂系统的"智能可理解"抽象。该抽象提供了许多好处：

它为在系统构建之前对软件系统的行为进行分析提供了基础。验证未来软件系统是否满足其利益相关者的需求而无需实际构建它的能力代表了大量的成本节约和风险缓解。已经开发了许多技术来执行此类分析，比如阿塔姆。

它为元素和决策的重用提供了基础。一个完整的软件体系结构或其部分，如单个体系结构策略和决策，可以跨多个系统重用，这些系统的涉众需要相似的质量属性或功能，从而节省设计成本并降低设计错误的风险。

它支持影响系统开发、部署和维护寿命的早期设计决策。正确地做出早期、高影响的决策对于防止进度和预算超支非常重要。

它促进了与涉众的沟通，有助于建立一个更好地满足其需求的系统。从涉众的角度沟通复杂系统，有助于他们理解所述需求的后果以及基于这些需求的设计决策。体系结构使得在系统实现之前就设计决策进行交流的能力，而这些决策仍然相对容易适应。

它有助于风险管理。软件架构有助于减少风险和失败的机会

它可以降低成本。软件架构是在复杂的IT项目中管理风险和成本的一种手段

1.云原生计算架构

CNCF（云原生计算基金会）给出了云原生应用的三大特征：

- 容器化封装：以容器为基础，提高整体开发水平，形成代码和组件重用，简化云原生应用程序的维护。在容器中运行应用程序和进程，并作为应用程序部署的独立单元，实现高水平资源隔离。
- 动态管理：通过集中式的编排调度系统来动态的管理和调度。
- 面向微服务：明确服务间的依赖，互相解耦。

云原生包含了一组应用的模式，用于帮助企业快速，持续，可靠，规模化地交付业务软件。云原生由微服务架构，DevOps 和以容器为代表的敏捷基础架构组成。

云原生的12-Factors经常被直译为12要素，也被称为12原则，由公有云PaaS的先驱Heroku于2012年提出，目的是告诉开发者如何利用云平台提供的便利来开发更具可靠性和扩展性、更加易于维护的云原生应用。具体如下：

- 基准代码
- 显式声明依赖关系
- 在环境中存储配置
- 把后端服务当作附加资源
- 严格分离构建、发布和运行

- 无状态进程
- 通过端口绑定提供服务
- 通过进程模型进行扩展
- 快速启动和优雅终止
- 开发环境与线上环境等价
- 日志作为事件流
- 管理进程

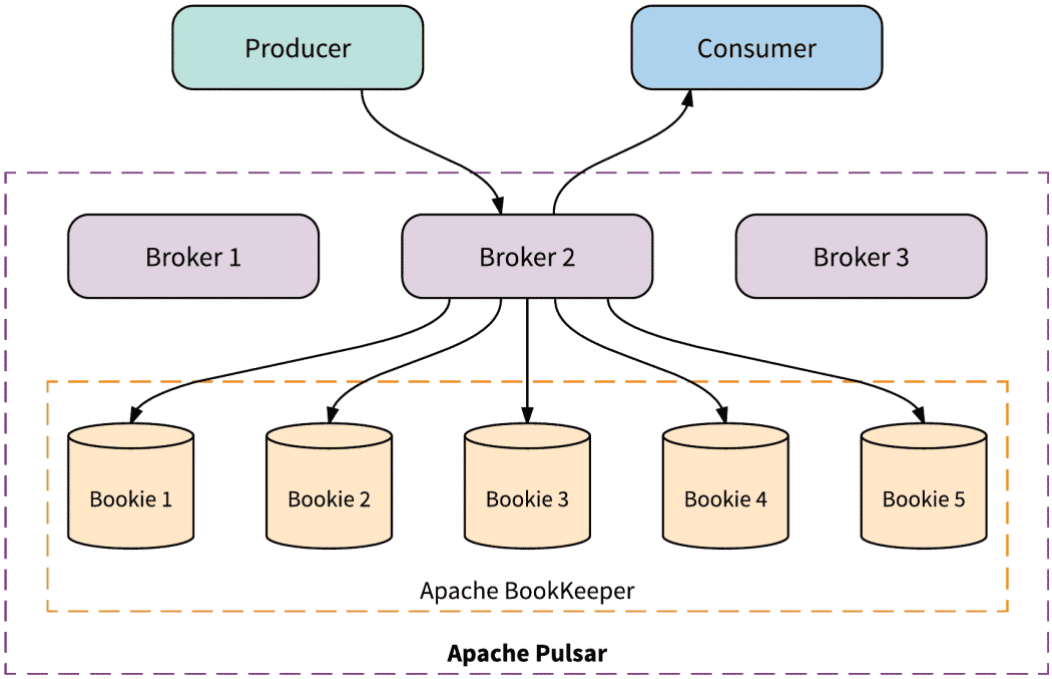
Jimmy Song对云原生架构中运用服务编排的总结是：Kubernetes——让容器应用进入大规模工业生产。这个总结确实很贴切。编排调度的开源组件还有：Kubernetes、Mesos和Docker Swarm。Kubernetes是目前世界上关注度最高的开源项目，它是一个出色的容器编排系统。Kubernetes出身于互联网行业的巨头Google公司，它借鉴了由上百位工程师花费十多年时间打造Borg系统的理念，通过极其简易的安装，以及灵活的网络层对接方式，提供一站式的服务。Mesos则更善于构建一个可靠的平台，用以运行多任务关键工作负载，包括Docker容器、遗留应用程序（例如Java）和分布式数据服务（例如Spark、Kafka、Cassandra、Elastic）。Mesos采用两级调度的架构，开发人员可以很方便的结合公司业务场景自定制MesosFramework。他们为云原生应用提供的强有力的编排和调度能力，它们是云平台上的分布式操作系统。在单机上运行容器，无法发挥它的最大效能，只有形成集群，才能最大程度发挥容器的良好隔离、资源分配与编排管理的优势，而对于容器的编排管理，Swarm、Mesos和Kubernetes的大战已经基本宣告结束，Kubernetes成为了无可争议的赢家。

传统的Web开发方式，一般被称为单体架构（Monolithic）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。单体架构进行演化升级之后，过渡到SOA架构，即面向服务架构。近几年微服务架构（Micro-Service Archeticture）是最流行的架构风格，旨在通过将功能模块分解到各个独立的子系统中以实现解耦，它并没有一成不变的规定，而是需要根据业务来做设计。微服务架构是对SOA的传承，是SOA的具体实践方法。微服务架构中，每个微服务模块只是对简单、独立、明确的任务进行处理，通过REST API返回处理结果给外部。在微服务推广实践角度来看，微服务将整个系统进行拆分，拆分成更小的粒度，保持这些服务独立运行，应用容器化技术将微服务独立运行在容器中。过去设计架构时，是在内存中以参数或对象的方式实现粒度细化。微服务使用各个子服务控制模块的思想代替总线。不同的业务要求，服务控制模块至少包含服务的发布、注册、路由、代理功能。容器化的出现，一定程度上带动了微服务架构。架构演化从单体式应用到分布式，再从分布式架构到云原生架构，微服务在其中有着不可或缺的角色。微服务带给我们很多开发和部署上的灵活性和技术多样性，但是也增加了服务调用的开销、分布式系事务、调试与服务治理方面的难题。

综上所述，云原生应用有三大特征：容器化封装、动态管理、面向微服务。首先由CNCF组织介绍了云原生的概念，然后分别对这三个特征进行详述。云原生架构是当下很火的讨论话题，是不同思想的集合，集目前各种热门技术之大成。

2.Pulsar消息流中间件

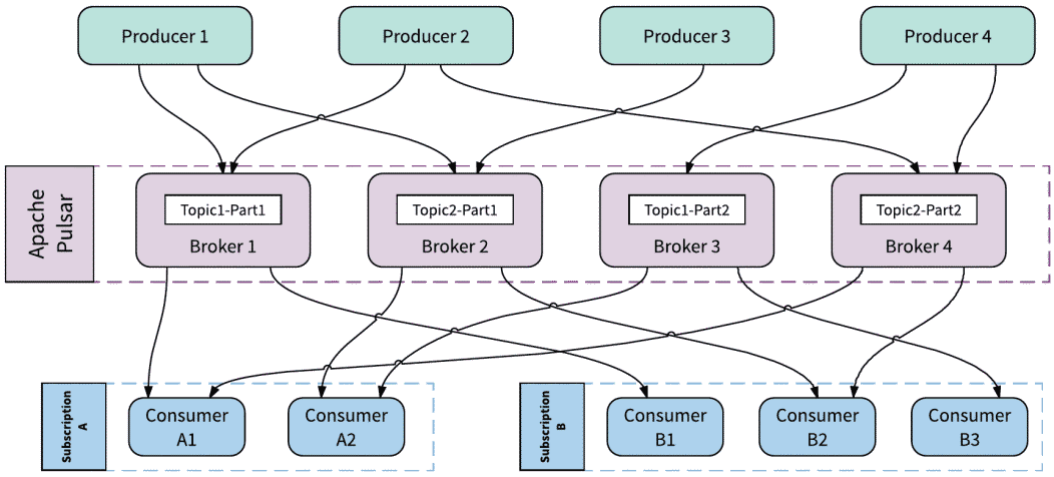
Pulsar 和其他消息系统最根本的不同是采用分层架构。 Apache Pulsar 集群由两层组成：无状态服务层，由一组接收和传递消息的 Broker 组成；以及一个有状态持久层，由一组名为 bookies 的 Apache BookKeeper 存储节点组成，可持久化地存储消息。下图显示了 Pulsar 的主体架构。



Pulsar 非常核心的设计逻辑就是消息服务和消息存储的分离, 即Broker 消息服务层和 BookKeeper 消息存储层。

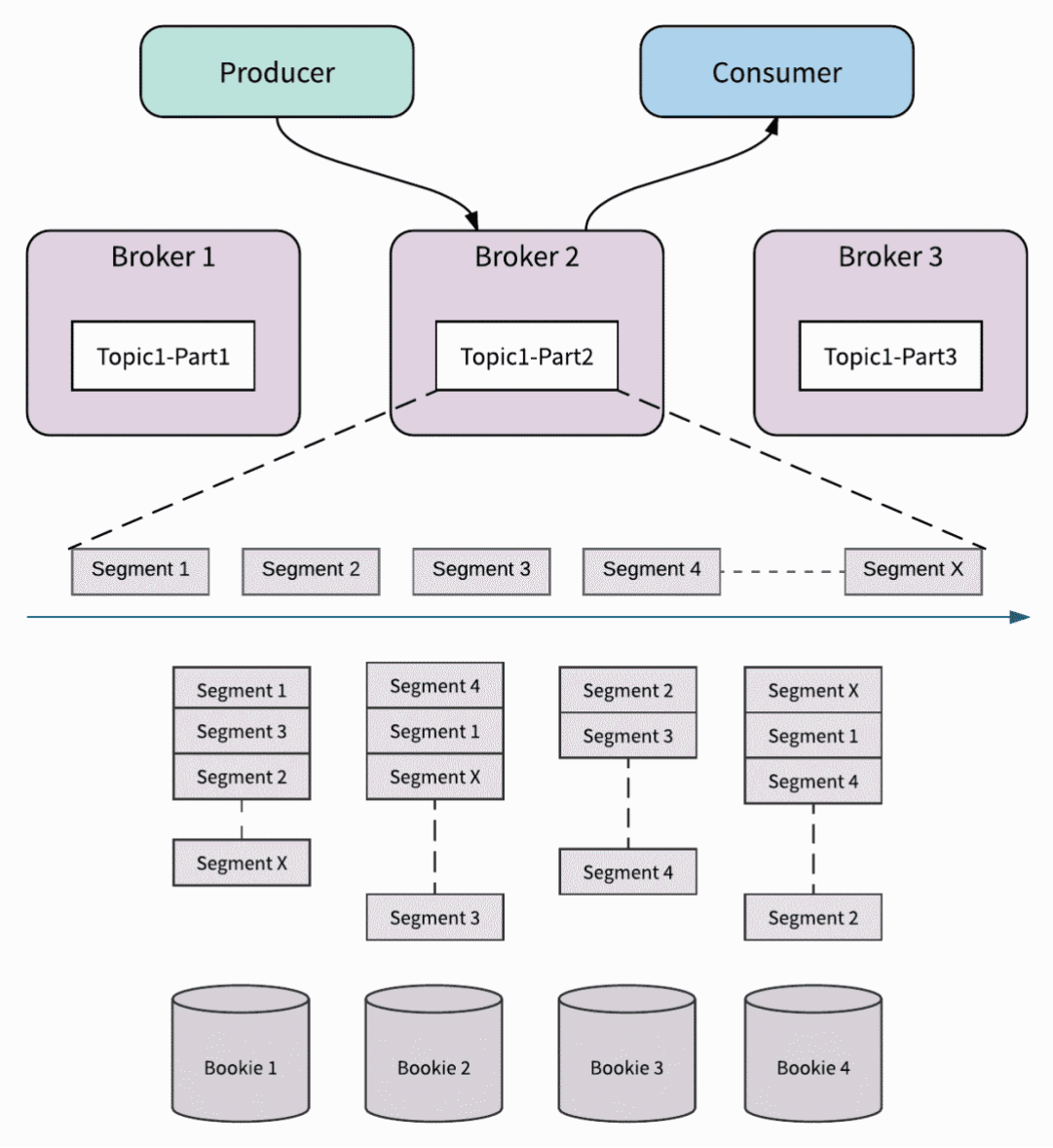
Broker

Broker 不存储消息本身，每个 topic 的消息都存储到了分布式日志存储系统 BookKeeper 中。 每个 partitioned topic 会被分配给某个 broker, 而生产者和消费者都连接到这个 broker 分别发送和消息消息。
如果一个 Broker 失败，Pulsar 会自动将其拥有的主题分区移动到群集中剩余的某一个可用 Broker 中。由于 Broker 是无状态的，当发生 Topic 的迁移时，Pulsar 只是将所有权从一个 Broker 转移到另一个 Broker，在这个过程中，不会有任何数据复制发生。Broker 层的示意图如下所示。



BookKeeper

BookKeeper 作为 puslar 的持久化层，负责存储每个 topic 的 partitions，本质上是分布式日志。每个分布式日志又被分为 Segment 分段。 每个 Segment 分段作为 BookKeeper 中的一个 Ledger，均匀分布并存储在 BookKeeper 群集中的多个 Bookie（BookKeeper 的存储节点）中。 Segment 创建的条件是基于配置的 serment 大小；基于配置的滚动时间, 或者 segment 的所有者发生了变化。通过 Segment 分段的方式，topic partition 中的消息可以均匀和平衡地分布在群集中的所有 Bookie 中。 因此一个 topic partition 的容量不受一个节点容量的限制；相反，它可以扩展到整个 BookKeeper 集群的总容量。 下图展示了一个 topic partition 存储示意



正是因为分层架构和以 segment 为中心存储的设计思想让 Pular 相比其他消息队列产品有了以下优势

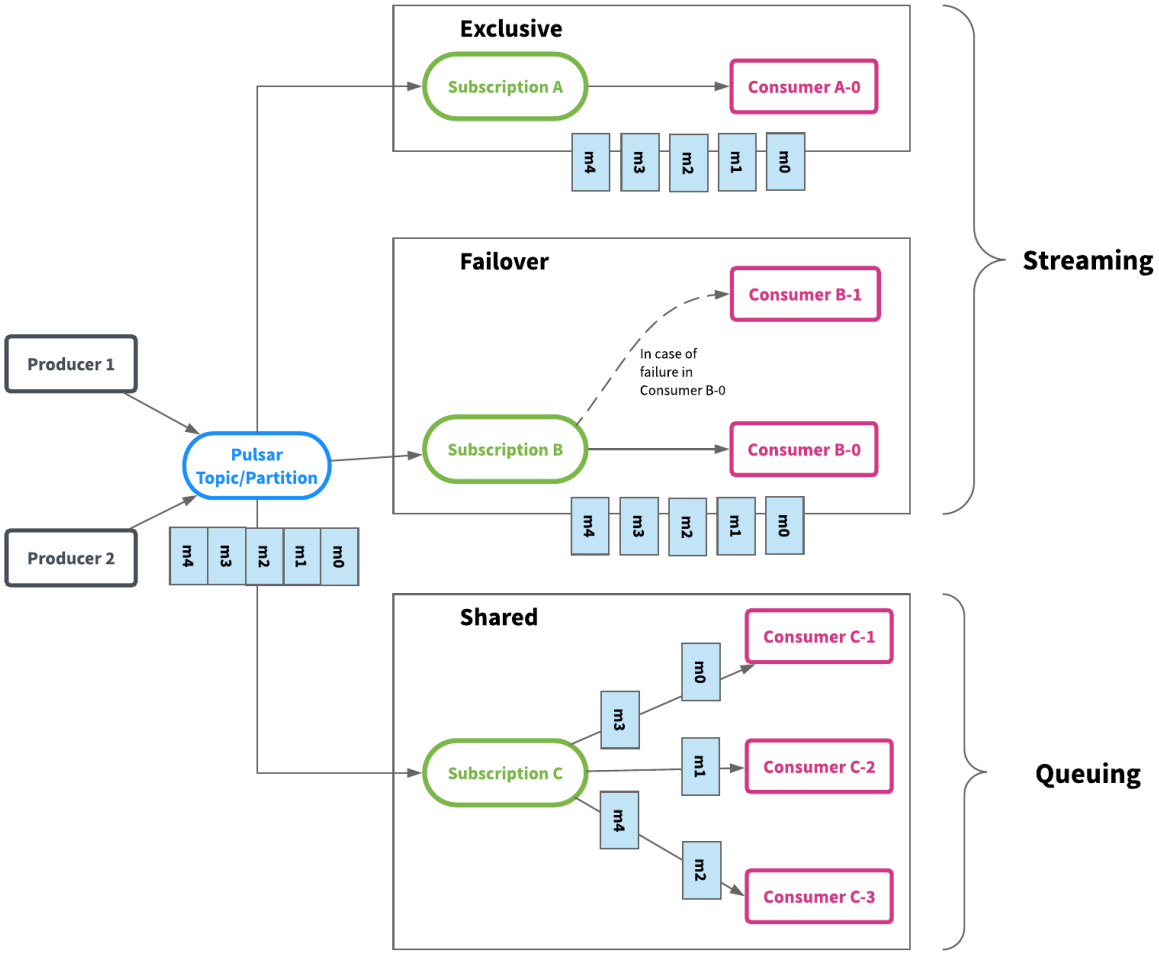
- 无限制的 topic partition 存储；
  topic patition 可以扩展到整个 BookKeeper 集群的总容量，只需添加 Bookie 节点即可扩展集群容量。
- 即时扩展，无需数据迁移；
  - 无缝 Broker 故障恢复
  - 无缝集群容量扩展
  - 无缝 Bookie 故障恢复
- 独立的扩展性

消息模型

消息系统的模型可以分为两类 队列式(queuing)和流式(streaming)。

queuing 模型主要是采用无序或者共享的方式来消费消息，当一条消息从队列发送出来后，多个消费者中的只有一个（任何一个都有可能）接收和消费这条消息。queuing 模型通常与无状态应用程序一起结合使用, 无状态应用程序不关心排序，但需要能够确认（ack）或删除单条消息，以及尽可能地扩展消费并行的能力。RabbitMQ 就是典型的 queuing 模型。 streaming模型要求消息的消费严格排序或独占消息消费，对于一个 message channel 只会有一个消费者消费消息。流模型通常与有状态应用程序相关联。有状态的应用程序更加关注消息的顺序及其状态。消息的消费顺序决定了有状态应用程序的状态。消息的顺序将影响应用程序处理逻辑的正确性。kafka 就是 streaming 模型。 Pulsar 的消息模型既支持 queuing，也支持 streaming。在 Pulsar 的消息消费模型中，Topic 是用于发送消息的通道, 消费

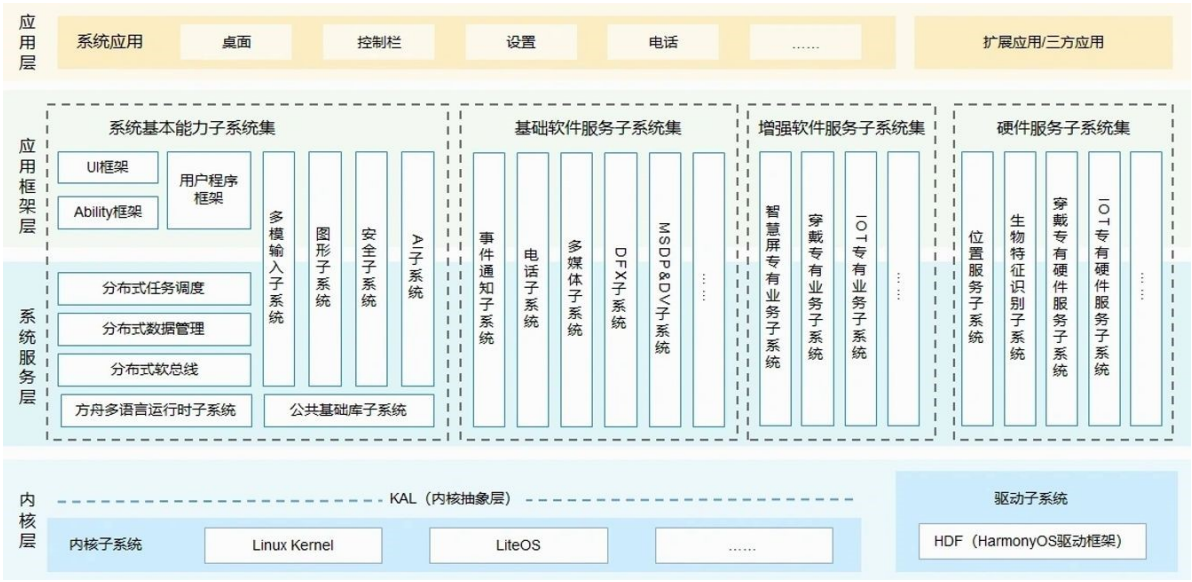
者被组合在一起消费消息，每个消费组是一个订阅。每组消费者可以拥有自己不同的消费方式： 独占（Exclusive），故障切换（Failover）或共享（Share）。Pulsar 通过这种模型，将队列模型和流模型这两种模型结合在了一起，提供了统一的 API 接口。 这种模型，既不会影响消息系统的性能，也不会带来额外的开销，同时还为用户提供了更多灵活性，方便用户程序以最匹配模式来使用消息系统。Exclusive 和 Failover 订阅，仅允许一个消费者来使用和消费每个对主题的订阅。这两种模式都按主题分区顺序使用消息。它们最适用于需要严格消息顺序的流（Stream）用例。 Share 订阅允许每个主题分区有多个消费者。同一订阅中的每个消费者仅接收主题分区的一部分消息。共享订阅最适用于不需要保证消息顺序的队列（Queue）的使用模式，并且可以按照需要任意扩展消费者的数量。 下图展示了3个不同类型的订阅和消息的流向。



Pulsar 将高性能的 streaming（Kafka）和灵活的 queuing（RabbitMQ）结合到一个统一的消息模型和 API 中。 Pulsar 使用统一的 API 为用户提供一个支持 streaming 和 queuing 的系统，且具有同样的高性能。

3.HarmonyOS分布式架构

HarmonyOS整体的分层结构自下而上依次为：内核层、系统服务层、应用框架层、应用层。HarmonyOS基于微内核设计，系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，各功能模块组织符合“抽屉式”设计，即功能模块采用AOP的设计思想，可根据实际需求裁剪某些非必要的子系统或功能/模块。HarmonyOS的设计实现模块化耦合，对应不同设备可实现弹性部署，使其可以方便、智能的适配GB、MB、KB等由低到高的不同内存规模设备，可以便捷的在诸如手机、智慧屏、车机、穿戴设备等IoT设备间实现数据的流转与迁移，同时兼具了小程序的按需使用，过期自动清理的突出优点。



内核层：

内核层基于Linux系统设计，主要包括内核子系统和驱动子系统（HDF， HarmonyOS驱动程序）。

内核子系统：HarmonyOS采用多内核设计，支持针对不同资源受限设备选用适合的OS内核。内核抽象层（KAL，KernelAbstract Layer）通过屏蔽多内核差异，对上层提供基础的内核能力，包括进程/线程管理、内存管理、文件系统、网络管理和外设管理等。

驱动子系统：HarmonyOS驱动框架（HDF）是HarmonyOS硬件生态开放的基础，提供统一外设访问能力和驱动开发、管理框架。

系统服务层：

系统服务层是HarmonyOS的核心能力集合，通过框架层对应用程序提供服务。该层包含以下几个部分：

系统基本能力子系统集：为分布式应用在HarmonyOS多设备上的运行、调度、迁移等操作提供了基础能力，由分布式软总线、分布式数据管理、分布式任务调度、方舟多语言运行时、公共基础库、多模输入、图形、安全、AI等子系统组成。其中，方舟运行时提供了C/C++/JS多语言运行时和基础的系统类库，也为使用自研的方舟编译器静态化的Java程序（即应用程序或框架层中使用Java语言开发的部分）提供运行时。

基础软件服务子系统集：为HarmonyOS提供公共的、通用的软件服务，由事件通知、电话、多媒体、DFX、MSDP&DV等子系统组成。

增强软件服务子系统集：为HarmonyOS提供针对不同设备的、差异化的能力增强型软件服务，由智慧屏专有业务、穿戴专有业务、IoT专有业务等子系统组成。

硬件服务子系统集：为HarmonyOS提供硬件服务，由位置服务、生物特征识别、穿戴专有硬件服务、IoT专有硬件服务等子系统组成。

根据不同设备形态的部署环境，基础软件服务子系统集、增强软件服务子系统集、硬件服务子系统集内部可以按子系统粒度裁剪，每个子系统内部又可以按功能粒度裁剪。

应用框架层：

框架层为HarmonyOS的应用程序提供了Java/C/C++/JS等多语言的用户程序框架和Ability框架，以及各种软硬件服务对外开放的多语言框架API；同时为采用HarmonyOS的设备提供了C/C++/JS等多语言的框架API，不同设备支持的API与系统的组件化裁剪程度相关。

应用层：

应用层包括系统应用和第三方非系统应用。HarmonyOS的应用由一个或多个FA（Feature Ability）或PA（Particle Ability）组成。其中，FA有UI界面，提供与用户交互的能力；而PA无UI界面，提供后台运行任务的能力以及统一的数据访问抽象。基于FA/PA开发的应用，能够实现特定的业务功能，支持跨设备调度与分发，为用户提供一致、高效的应用体验。

HarmonyOS最大的特征就是将分布式架构应用于终端OS，使用户可以方便的实现同一账户跨设备、跨终端的调用。其分布式架构包括分布式任务调度、分布式数据管理、硬件能力虚拟化、分布式软总线。尤其是创新性的分布式软总线技术，更是使鸿蒙系统的端到端时延小于20ms，有效吞吐高达1.2Gbps、抗丢包率高达25%。HarmonyOS根据应用及设备类别，智能分配慢速、快速、超快速车道，从而保证不同应用在不同设备上的的快捷、刘畅运行。在车机系统方面，经测相比谷歌Fusion系统有3-5倍的性能提升。在安全方面，HarmonyOS采用微内核技术，从源头认证系统安全，该级别的安全技术以往仅应用于航空、军工等领域。其自研的方舟编译器，兼容Android，保证在Android上开发的应用可以“非常容易”的迁移到鸿蒙系统上，并且打破了Android系统只能在虚拟机上运行，而无法直接和系统底层直接通信的桎梏，从而大大提升性能。

4.Serverless架构

如同许多新的概念一样，Serverless目前还没有一个普遍公认的权威的定义。最新的一个定义是这样描述的：“无服务器架构是基于互联网的系统，其中应用开发不使用常规的服务进程。相反，它们仅依赖于第三方服务（例如AWS Lambda服务），客户端逻辑和服务托管远程过程调用的组合。”

最开始，“无服务器”架构试图帮助开发者摆脱运行后端应用程序所需的服务器设备的设置和管理工作。这项技术的目标并不是为了实现真正意义上的“无服务器”，而是指由第三方云计算供应商负责后端基础结构的维护，以服务的方式为开发者提供所需功能，例如数据库、消息，以及身份验证等。简单地说，这个架构的就是要让开发人员关注代码的运行而不需要管理任何的基础设施。程序代码被部署在诸如AWS Lambda这样的平台之上，通过事件驱动的方法去触发对函数的调用。很明显，这是一种完全针对程序员的架构技术。其技术特点包括了事件驱动的调用方式，以及有一定限制的程序运行方式，例如AWS Lambda的函数的运行时间默认为3秒到5分钟。从这种架构技术出现的两年多时间来看，这个技术已经有了非常广泛的应用，例如移动应用的后端和物联网应用等。简而言之，无服务器架构的出现不是为了取代传统的应用。然而，从具有高度灵活性的使用模式及事件驱动的特点出发，开发人员和架构师应该重视这个新的计算范例，它可以帮助我们达到减少部署、提高扩展性并减少代码后面的基础设施的维护负担。

Serverless有以下几个特点：

- Serverless意味无维护，Serverless不代表完全去除服务器，而是代表去除有关对服务器运行状态的关心和担心，它们是否在工作，应用是否跑起来正常运行等等。Serverless代表的是你不要关心运营维护问题。有了Serverless，可以几乎无需Devops了。
- Serverless不代表某个具体技术，有些人会给他们的语言框架取名为Serverless，Serverless其实去除维护的担心，如果你了解某个具体服务器技术当然有帮助，但不是必须的。
- Serverless中的服务或功能代表的只是微功能或微服务，Serverless是思维方式的转变，从过去：“构建一个框架运行在一台服务器上，对多个事件进行响应。”变为：“构建或使用一个微服务或微功能来响应一个事件。”，你可以使用 django or node.js 和express等实现，但是serverless本身超越这些框架概念。框架变得也不那么重要了。

Serverless架构主要有如下几个优点：

降低人力成本

不需要再自己维护服务器，操心服务器的各种性能指标和资源利用率，而是关心应用程序本身的状态和逻辑。而且 serverless 应用本身的部署也十分容易，我们只要上传基本的代码但愿，例如 Javascript 或 Python 的源代码的 zip 文件，以及基于JVM的语言的纯 JAR 文件。不需使用 Puppet、Chef、Ansible 或 Docker 来进行配置管理，降低了运维成本。同时，对于运维来说，也不再需要监控那些更底层的如磁盘使用量、CPU 使用率等底层和长期的指标信息，而是监控应用程序本身的度量，这将更加直观和有效。在此看来有人可能会提出 “NoOps” 的说法，其实这是不存在的，只要有应用存在的

一天就会有 Ops，只是人员的角色会有所转变，部署将变得更加自动化，监控将更加面向应用程序本身，更底层的运维依然需要专业的人员去做。

降低风险

对于组件越多越复杂的系统，出故障的风险就越大。我们使用 BaaS 或 FaaS 将它们外包出去，让专业人员来处理这些故障，有时候比我们自己来修复更可靠，利用专业人员的知识来降低停机的风险，缩短故障修复的时间，让我们的系统稳定性更高。

减少资源开销

我们在申请主机资源一般会评估一个峰值最大开销来申请资源，往往导致过度的配置，这意味着即使在主机闲置的状态下也要始终支付峰值容量的开销。对于某些应用来说这是不得已的做法，比如数据库这种很难扩展的应用，而对于普通应用这就显得不太合理了，虽然我们都觉得即使浪费了资源也比当峰值到来时应用程序因为资源不足而挂掉好。解决这个问题最好的办法就是，不计划到底需要使用多少资源，而是根据实际需要来请求资源，当然前提必须是整个资源池是充足的（公有云显然更适合）。根据使用时间来付费，根据每次申请的计算资源来付费，让计费的粒度更小，将更有利于降低资源的开销。这是对应用程序本身的优化，例如让每次请求耗时更短，让每次消耗的资源更少将能够显著节省成本。

增加缩放的灵活性

以 AWS Lamba 为例，当平台接收到第一个触发函数的事件时，它将启动一个容器来运行你的代码。如果此时收到了新的事件，而第一个容器仍在处理上一个事件，平台将启动第二个代码实例来处理第二个事件。AWS lambad 的这种自动的零管理水平缩放，将持续到有足够的代码实例来处理所有的工作负载。但是，AWS 仍然只会向您收取代码的执行时间，无论它需要启动多少个容器实例要满足你的负载请求。例如，假设所有事件的总执行时间是相同的，在一个容器中按顺序调用Lambda 100 次与在 100 个不同容器中同时调用 100 次 Lambda 的成本是 一样的。当然 AWS Lambada 也不会无限制的扩展实例个数，如果有人对你发起了 DDos 攻击怎么办，那么不就会产生高昂的成本吗？AWS 是有默认限制的，默认执行 Lambada 函数最大并发数是 1000。

缩短创新周期

小团队的开发人员正可以在几天之内从头开始开发应用程序并部署到生产。使用短而简单的函数和事件来粘合强大的驱动数据存储和服务的 API。完成的应用程序具有高度可用性和可扩展性，利用率高，成本低，部署速度快。以 Docker 为代表的容器技术仅仅是缩短了应用程序的迭代周期，而 serverless 技术是直接缩短了创新周期，从概念到最小可行性部署的时间，让初级开发人员也能在很短的时间内完成以前通常要经验丰富的工程师才能完成的项目。

综上所述，serverless技术有着非常广阔的应用空间。