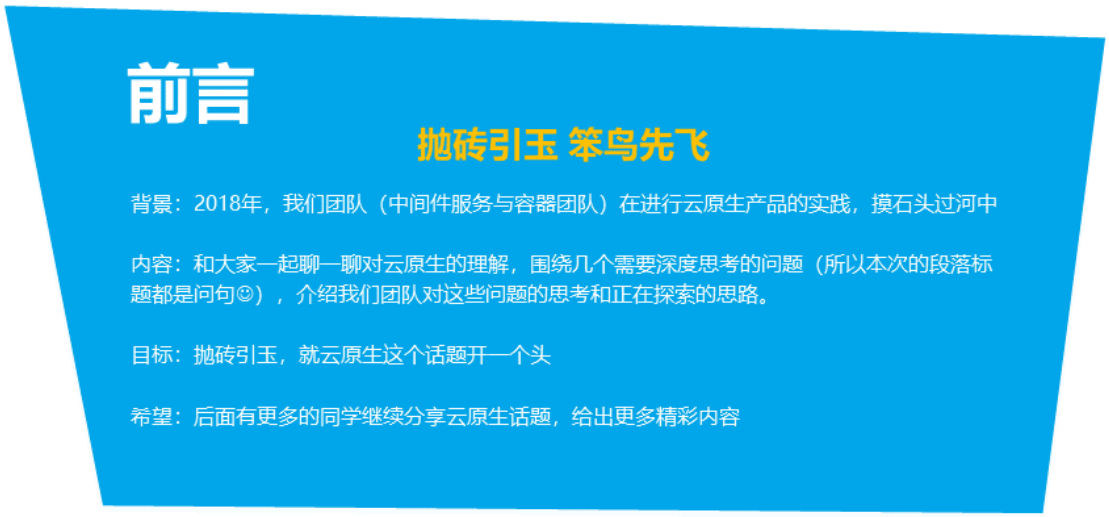
畅谈云原生（上）

今天和大家一起聊一聊云原生这个话题，内容来自蚂蚁金服中间件服务与容器团队。

由于内容比较多，我们分为上下两个半场。

前言



特别指出：这次分享主要是希望起到抛砖引玉的作用，让大家更多的参与到云原生这个话题的讨论，希望后面有更多更好的分享。我们笨鸟先飞，来一个头。

内容主要围绕这几个问题，上半场我们将围绕前三个问题。

如何理解云原生?

第一个话题：如何理解“云原生”？之所以将这个话题放在前面，是因为，这是对云原生概念的最基本的理解，而这会直接影响到后续的所有认知。

吾日三省吾身：如何理解云原生？

Shakespeare： There are a thousand Hamlets in a thousand people's eyes.

每个人对云原生的理解都可能不同，就如莎士比亚所说：一千个人眼中有一千个哈姆雷特。

快速回顾

云原生定义的变化

我们来快速回顾一下云原生这个词汇在近年来定义的变化。

来自Pivotal的定义

Pivotal 是Cloud Native/云原生应用的提出者，并推出了Pivotal Cloud Foundry和Spring系列开发框架，是云原生的先驱者和探路者。

Pivotal

CLOUD FOUNDRY

spring

先看Pivotal，云原生的提出者，是如何定义云原生的。

Pivotal的最初定义

2015年，来自Pivotal公司的Matt Stine编写了一本名为 迁移到云原生应用架构 的电子书，提出云原生应用架构应该具备的几个主要特征：

- 符合12因素应用(Twelve-Factor Applications)
- 面向微服务架构(Microservices)
- 自服务敏捷架构(Self-Service Agile Infrastructure)
- 基于API的协作(API-Based Collaboration)
- 抗脆弱性(Antifragility)

这是2015年，云原生刚刚开始推广时，Matt Stine给出的定义。

Pivotal更新后的定义

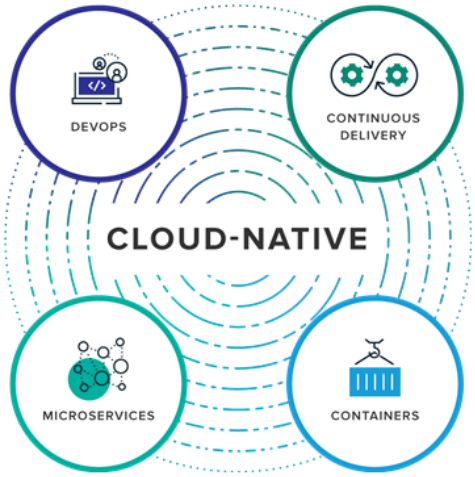
在2017年10月，也是Matt Stine，在接受InfoQ采访时，则对云原生的定义做了小幅调整，将Cloud Native Architectures定义为具有以下六个特质：

- 模块化(Modularity):（通过微服务）
- 可观测性(Observability)
- 可部署性(Deployability)
- 可测试性(Testability)
- 可处理性(Disposability)
- 可替换性(Replaceability)

两年之后，同样是Matt Stine。

Pivotal官网最新定义

在Pivotal最新的官方网站 https://pivotal.io/cloud-native 上，对cloud native的介绍是关注四个要点：



而这是Pivotal最新官方网站的描述。可见Pivotal对云原生的定义一直在变。

CNCF的定义

2015年CNCF建立，开始围绕云原生的概念打造云原生生态体系，起初CNCF对云原生的定义包含以下三个方面：

- 应用容器化(software stack to be Containerized)
- 面向微服务架构(Microservices oriented)
- 应用支持容器的编排调度(Dynamically Orchestrated)

到2018年，随着社区对云原生理念的广泛认可和云原生生态的不断扩大，
还有CNCF项目和会员的大量增加，起初的定义已经不再适用

再来看看目前云原生背后最大的推手，CNCF，这是2015年CNCF刚成立时对云原生的定义。

CNCF的2018年更新后的定义：v1.0

云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

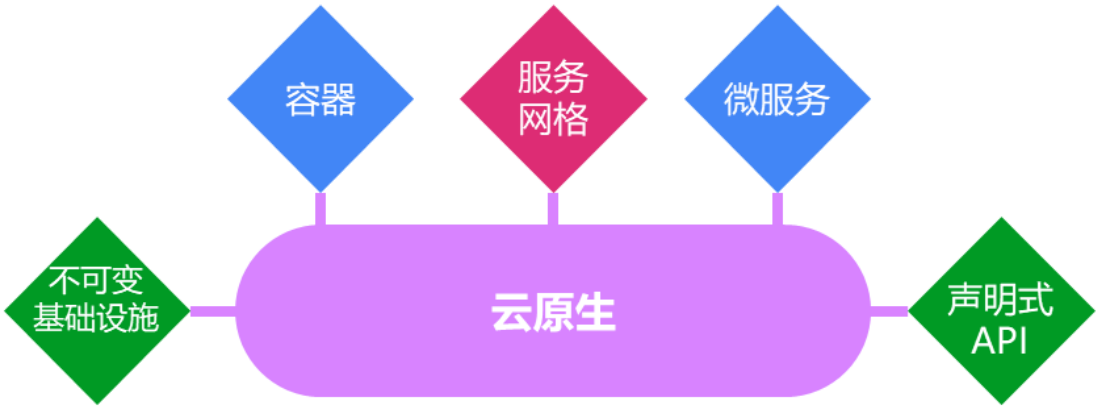
这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

云原生计算基金会（CNCF）致力于培育和维护一个厂商中立的开源生态系统，来推广云原生技术。我们通过将最前沿的模式民主化，让这些创新为大众所用。

https://github.com/cncf/toc/blob/master/DEFINITION.md

2018年CNCF更新了云原生的定义。

云原生代表技术



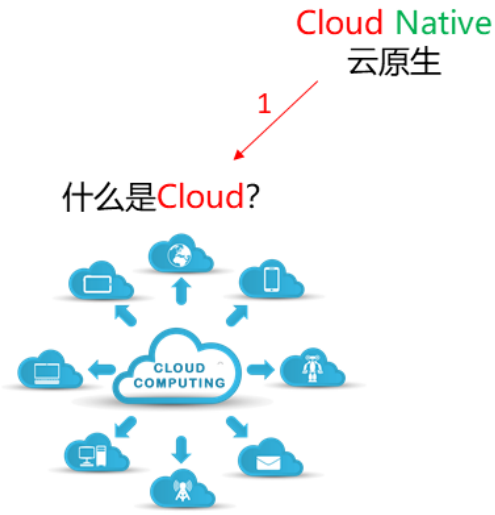
这是新定义中描述的代表技术，其中容器和微服务两项在不同时期的不同定义中都有出现，而服务网格这个在2017年才开始被社区接纳的新热点技术被非常醒目的列出来，和微服务并列，而不是我们通常认为的服务网格只是微服务在实施时的一种新的方式。

云原生定义回顾小结：

- ✓ 云原生的定义一直在变
  - 不同组织有不同的定义：Pivotal & CNCF
  - 同一个组织在不同时间点有不同的定义
  - 同一个人在不同时间点也有不同的定义
- ✓ 云原生的定义未来还会变
  - CNCF最新的定义：版本v1.0

云原生自身的定义一直在变，这让我们该如何才能准确的理解云原生呢?

回到话题：我们该如何理解云原生？



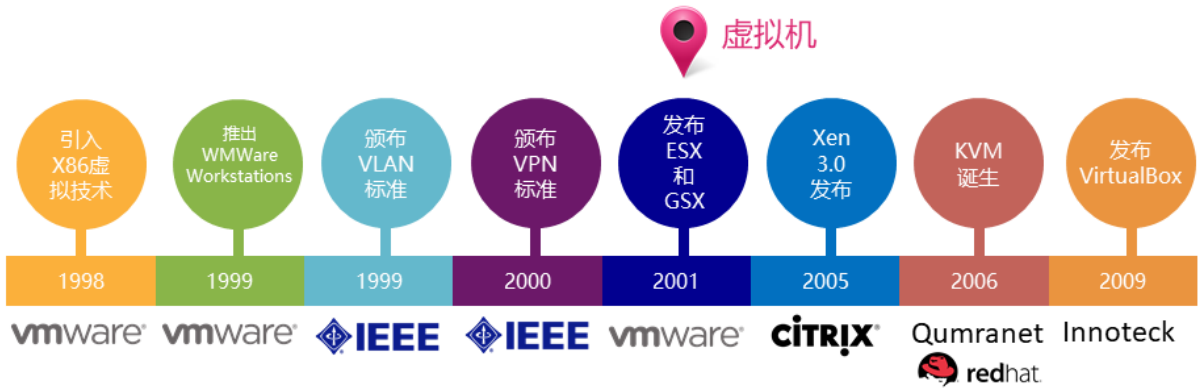
这里我们尝试，将Cloud Native这个词汇拆开来理解，先看看什么是Cloud。

快速回顾

云计算的演进历史

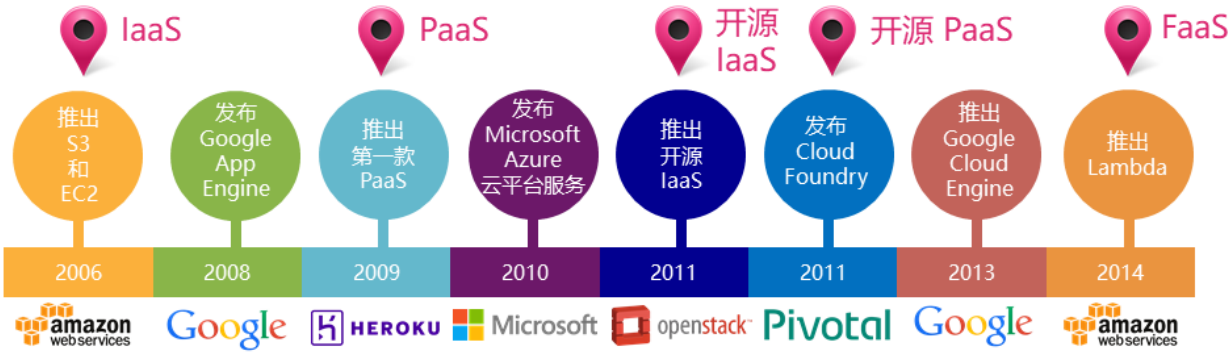
快速回顾一下云计算的历史，来帮助我们对云有个更感性的认识。

虚拟化技术的成熟

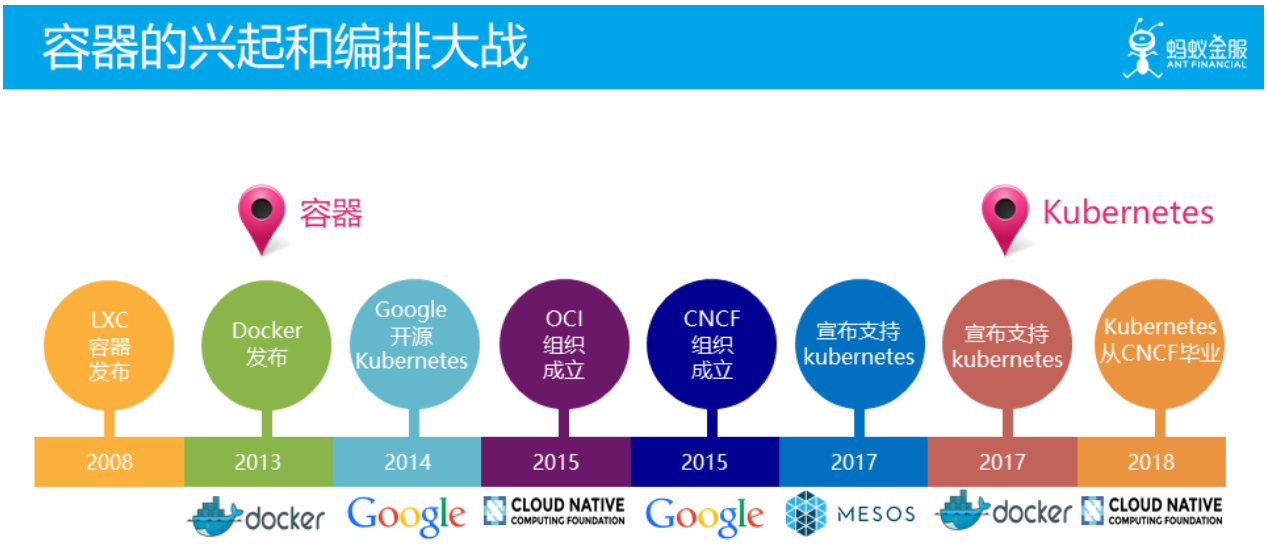


云计算的出现和虚拟化技术的发展和成熟密切相关，2000年前后x86的虚拟机技术成熟后，云计算逐渐发展起来。

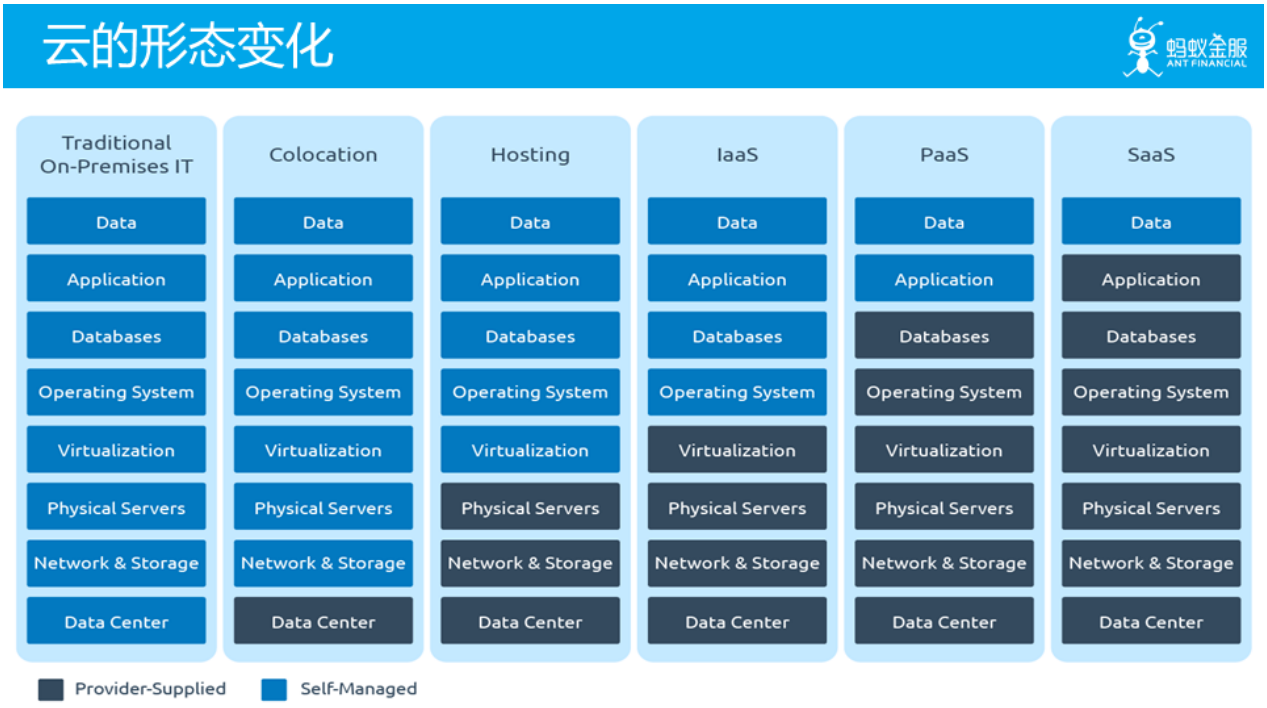
基于虚拟机的云计算



基于虚拟机技术，陆续出现了IaaS/PaaS/FaaS等形态，以及他们的开源版本。



2013年docker出现，容器技术成熟，然后围绕容器编排一场大战，最后在2017年底，kubernetes胜出。2015年CNCF成立，并在近年形成了cloud native生态。



这是云的形态变化，可以看到：供应商提供的功能越来越多，而客户或者说应用需要自己管理的功能越来越少。

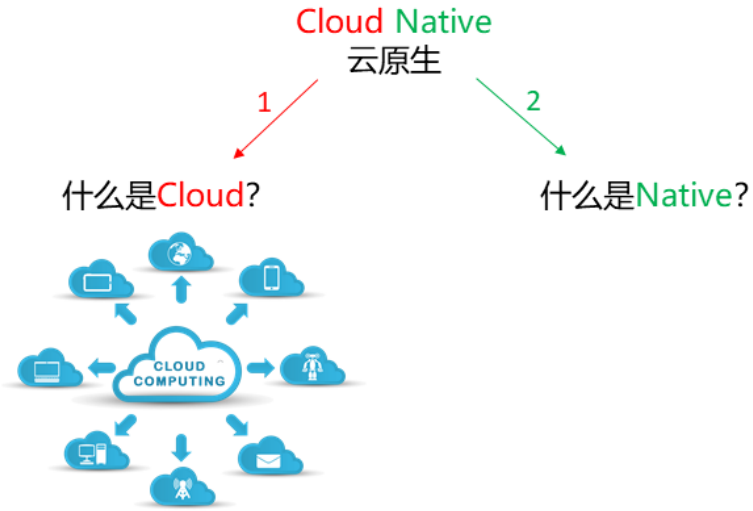
云计算历史回顾小结：

- ✓ 云计算二十年来变化巨大
  - 从物理机到虚拟机到容器
  - IaaS, PaaS, SaaS, Caas, FaaS 等多种形态出现
  - 公有云、私有云、混合云
  - Kubernetes出现并成为事实标准
  - 在低可用率的硬件上搭建高可用率的服务

云上的应用要如何适应？

当云发生如此之大的变化时，云上的应用要如何适应？

回到话题：我们该如何理解云原生？



在回顾完云计算的历史之后，我们对Cloud有更深的认识，接着继续看一下：什么是Native?

字典中的Native

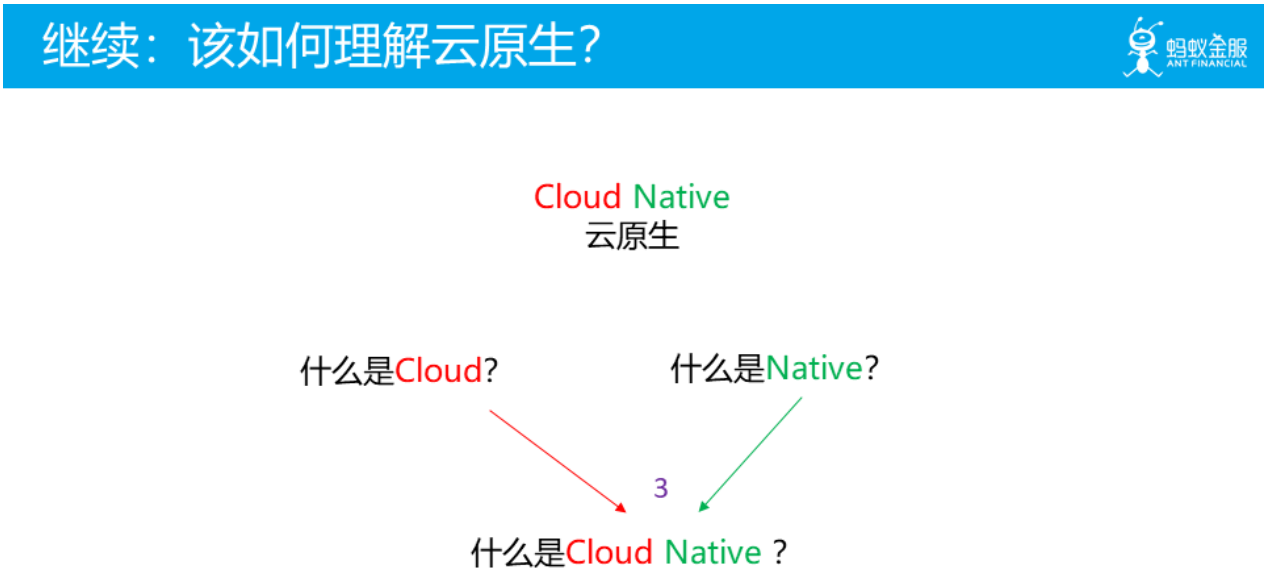
- belonging to a person by birth or to a thing by nature; inherent:
  - native ability; native grace.
- belonging by birth to a people regarded as indigenous to a certain place, especially a preliterate people:
  - Native guides accompanied the expedition through the rain forest.
- born in a particular place or country:
  - a native New Yorker.

这是从字典中摘抄下来的对Native词条的解释，注意其中标红的关键字。

关键字：Born

与生俱来，生而知之；无牵无挂，无拘无束

Native，总是和关键字 Born 联系在一起。



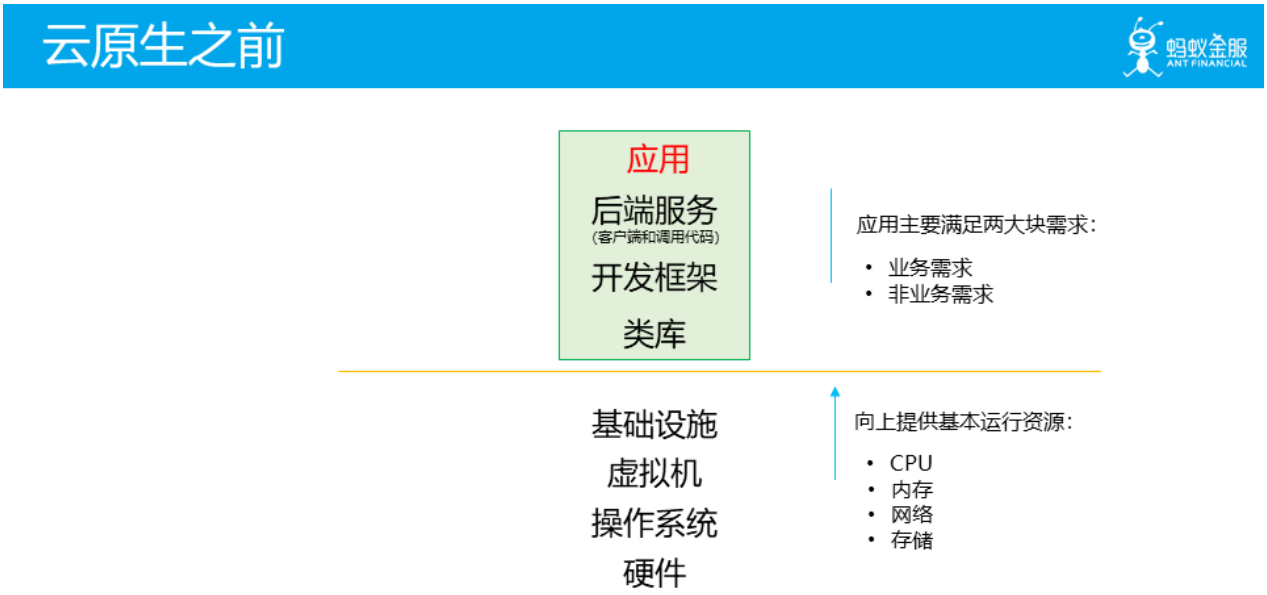
那Cloud和native和在一起，又该如何理解?

这里我们抛出一个我们自己的理解：云原生代表着原生为云设计。详细的解释是：应用原生被设计为在云上以最佳方式运行，充分发挥云的优势。

这个理解有点空泛，但是考虑到云原生的定义和特征在这些年间不停的变化，以及完全可以预料到的在未来的必然变化，我觉得，对云原生的理解似乎也只能回到云原生的出发点，而不是如何具体实现。

云原生应用应该是什么样子?

那在这么一个云原生理解的背景下，我再来介绍一下我对云原生应用的设想，也就是我觉得云原生应用应该是什么样子。



在云原生之前，底层平台负责向上提供基本运行资源。而应用需要满足业务需求和非业务需求，为了更好的代码复用，通用型好的非业务需求的实现往往会以类库和开发框架的方式提供，另外在SOA/微服务时代部分功能会以后端服务的方式存在，这样在应用中就被简化为对其客户端的调用代码。

然后应用将这些功能，连同自身的业务实现代码，一起打包。

传统非云原生应用

背景：美国今年冬天特别冷，然后出现了这种极度夸张的超大号搞笑衣服。

Gigi Hadid

这是传统非云原生应用的一个形象表示：在业务需求的代码实现之后，包裹厚厚的一层非业务需求的实现（当然以类库和框架的形式出现时代码量没这么夸张）。

云原生：云的支持应该让应用更多关注业务

而云的出现，可以在提供各种资源之外，还提供各种能力，从而帮助应用，使得应用可以专注于业务需求的实现。

我们觉得：理想的云原生应用应该是这个样子 ☺

Gigi Hadid

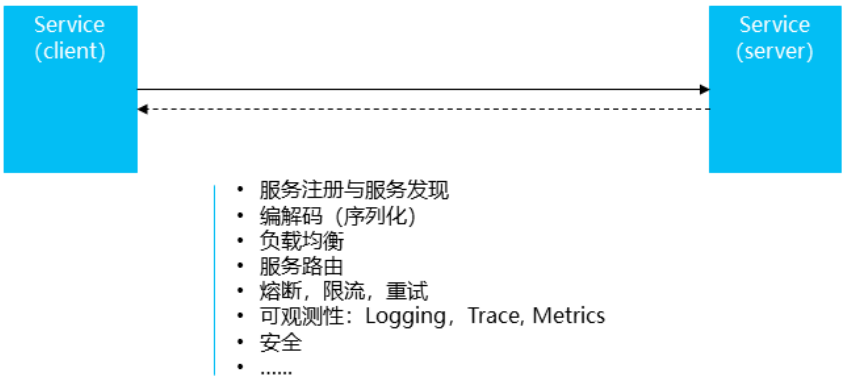
在我们设想中，理想的云原生应用应该是这个样子：业务需求的实现占主体，只有少量的非业务需求相关的功能。

问题：“衣服”那里去了？

- ✓ 云
- ✓ 基础设施
- ✓ 下沉到基础设施的中间件
  - 后面会详细解释什么是下沉

非业务需求相关的功能都被移到云，或者说基础设施中去了，以及下沉到基础设施的中间件。

以服务间通讯为例

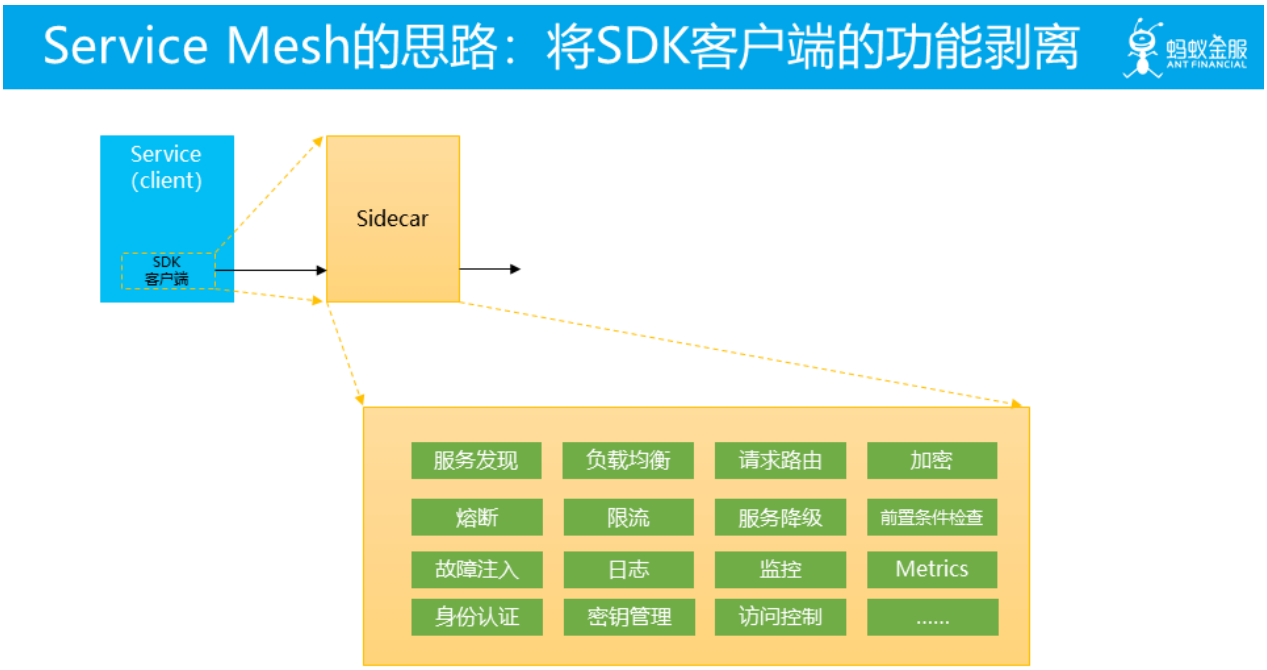


- 服务注册与服务发现
- 编解码（序列化）
- 负载均衡
- 服务路由
- 熔断，限流，重试
- 可观测性：Logging，Trace, Metrics
- 安全
- ……

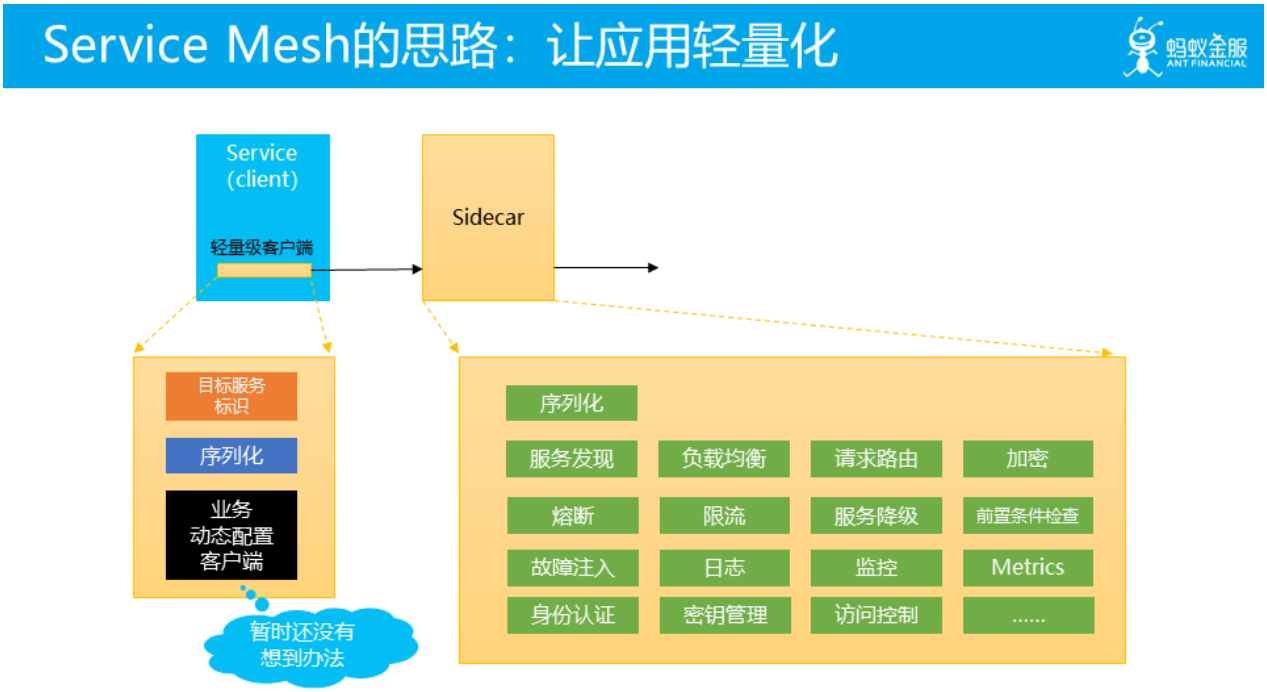
以服务间通讯为例：需要实现上面列举的各种功能。

SDK的思路

SDK的思路：通过一个胖客户端，在这个客户端中实现各种功能。



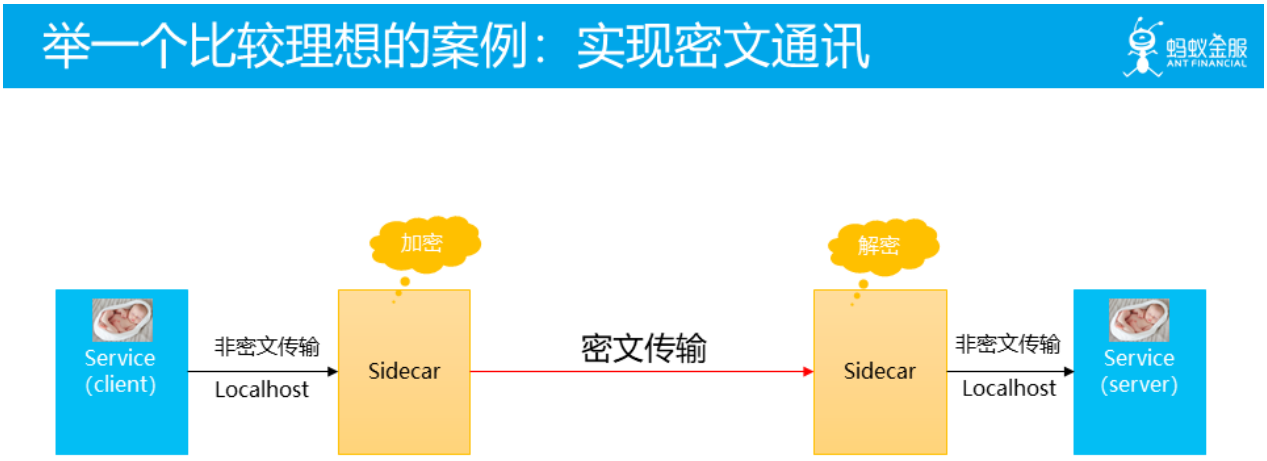
Service Mesh的思路，体现在将SDK客户端的功能剥离出来，放到Sidecar中。



通过这种方式，实现应用的轻量化。此时绝大部分的功能都在剥离，应用中只留下一个轻量级的客户端。这个轻量级客户端中还保留有少数功能和信息，比如目标服务的标识（指出要调用的目标），序列化的实现。

这里特别指出，有一个功能是我们努力尝试但是始终没有找到办法的：业务动态配置的客户端。也就是如何获取和应用**业务逻辑**实现相关的**动态**配置信息。如果有哪位同学对此有研究，希望可以指教。

我们的想法，云原生应用应该超轻量化的方向努力，尽量将业务需求之外的功能剥离出来。当然要实现理想中的状态还是比较难的，但是及时是比较务实的形态，也能比非云原生下要轻量很多。



在这里举一个效果比较理想的实际案例，在service mesh中实现密文通讯。

由于客户端和服务器端两个sidecar的存在，因此我们可以通过Sidecar之间的协商与合作分别实现加密和解密，从而实现远程通讯的密文传输，而这个加密和解密的过程对于原有应用是透明的。

云原生下的中间件该如何发展?

云原生涉及到的面非常广，对开发测试运维都会有影响，我们这里将聚焦在中间件，给出我们的一些粗浅的想法，因为我们来自中间件部门。大家也可以将中间件自行替换为自己关心的领域，尝试思考一下这个问题。

前面我们讲到云原生应用的理想形态和轻量化方向，这里隐含了一个前提：不管云原生应用的形态如何变化，云原生应用最终对外提供的功能应该是保持一致的。

应用形态依赖于云提供的能力

非原生

云只能提供**非常有限**的能力，应用需要自行实现

务实版
云原生

云提供**大部分**能力，部分云未能提供的能力依然需要应用自行实现

理想版
云原生

云提供绝大部分能力，只是在某些特定环节无法完全剥离和解耦

梦想版
云原生

云提供**所有**能力，所有环节都完全解耦

而要实现这一点，应用需要依赖于云提供的能力，来替换因应用轻量化而剥离的原有能力，云提供的能力是应用形态演变的基础和前提。

云的能力

现实：还远没有实现

理想：由云来提供应用需要的所有能力

理想状态下，我们期望云能够提供应用需要的所有能力，这样应用就可以以最原生化的形态出现。但是现实是这一点远还没有做到，我们依然需要在云之外额外提供一些功能，比如原有中间件的功能。

中间件的做法

应用
后端服务
(客户端和调用代码)
开发框架
类库

在云原生前：通常以类库和框架的形式

云原生：下沉到基础设施，成为云的一部分

在云原生之前，中间件的做法通常是以类库和框架的形式出现，近年来也有服务形式。而在云原生时代，我们的想法是让中间件下沉到基础设施，成为云的一部分。

解释：什么是“下沉”？

问题：“衣服”那里去了？

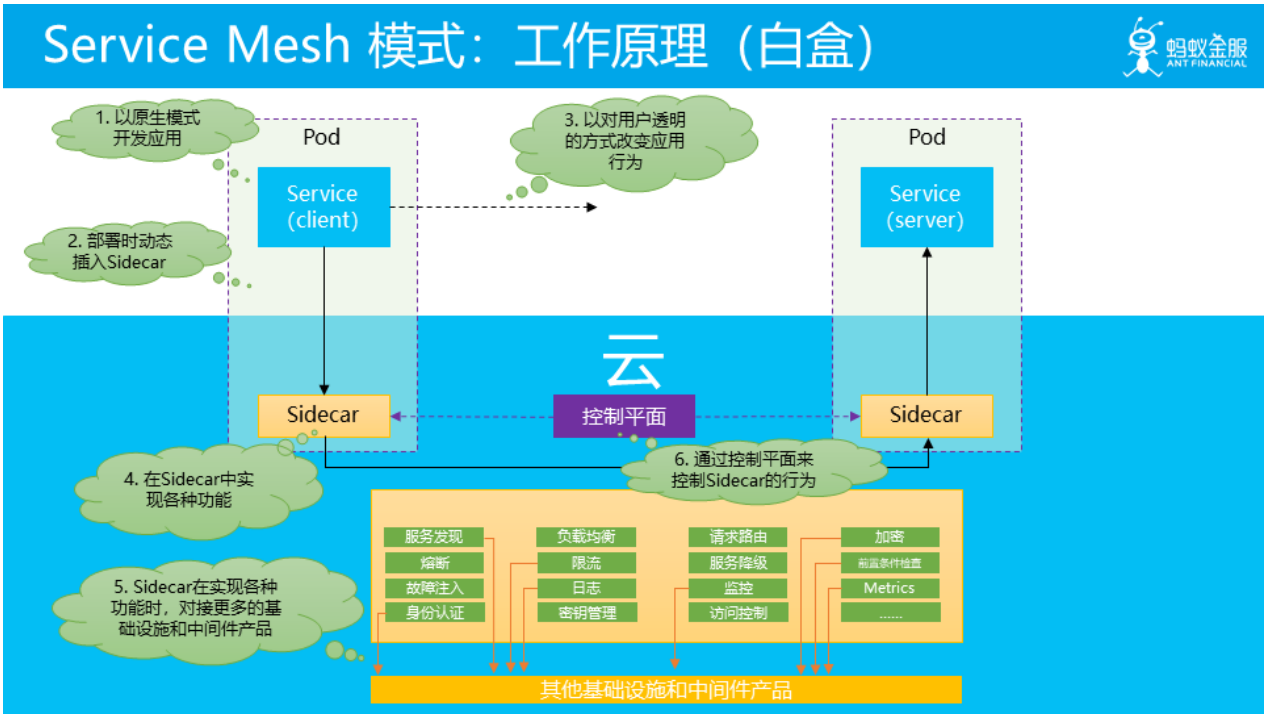
- ✓ 云
- ✓ 基础设施
- ✓ 下沉到基础设施的中间件

  - 是下沉，不是消失
  - 产品依然存在，功能继续提供
  - 但是，不和应用直接耦合（轻量化）
  - 也不被应用直接感知（原生）
  - 在运行时为应用动态赋能
    - 具体实现后面展开

在这里解释一下，在前面就提到了“下沉”，什么是**下沉**?

我们的想法是：云原生下的中间件，功能应该继续提供，但是中间件给应用的赋能方式，应该云原生化：

- 在云原生之前，应用需要实现非常多的能力，即使是以通过类库和框架的方式简化，其思路是加强应用能力，方式如左图所示，通过提供更大更厚的衣物来实现御寒御寒能力。
- 云原生则是另外的思路，主张加强和改善应用运行环境（即云）来帮助应用，如右图所示，通过提供温暖的阳光，来让轻量化成为可能。

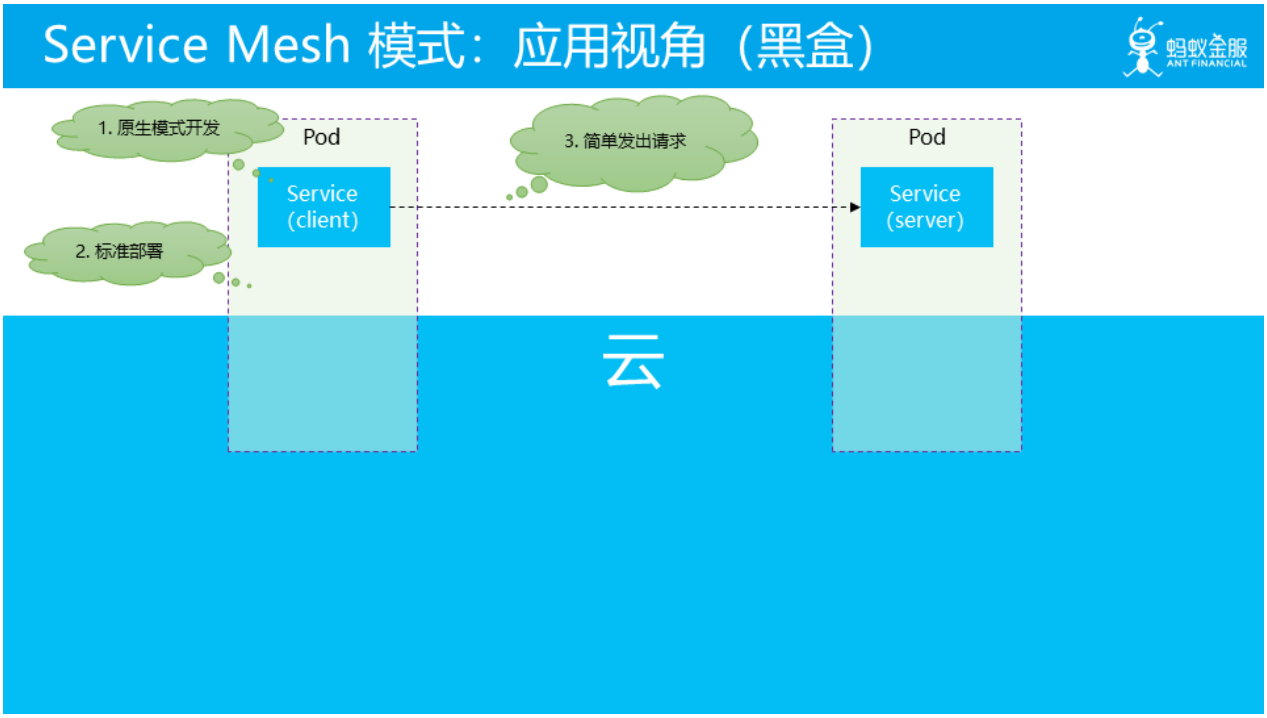


我们以Service Mesh模式为例来详细讲解，首先我们以白盒的视角来看Service Mesh的工作原理：

1. 以原生模式开发应用：应用只需具备最基本的能力，如客户端简单发一个请求给服务器端
2. 部署时动态插入Sidecar：当我们将开发的云原生应用部署到云上，具体说是部署在k8s的pod中时，我们会自动在pod中再部署一个Sidecar，用于实现为应用赋能
3. 在运行时，我们会改变云原生应用的行为：如前面所说客户端简单发一个请求给服务器端，在这里

会被改变为将请求劫持到Sidecar，注意这个改变对应用而言是透明无感知的

4. 在Sidecar中实现各种功能：Sidecar里面就可以实现原有SDK客户端实现的各种功能，如服务发现，负载均衡，路由等等
5. Sidecar在实现这些功能时，可以对接更多的基础设施，也可以对接其他的中间件产品，这种情况下，Service Mesh产品会成为应用和基础设施/中间件之间的桥梁
6. 可以通过控制平面来控制Sidecar的行为，而这些控制可以独立于应用之外



我们再以应用的视角，将云和下沉到云中的Service Mesh产品视为黑盒，来看Service Mesh模式：

1. 以原生模式开发应用
2. 以标准模式部署应用：底下发生了什么不关心
3. 客户端简单发一个请求给服务器端：底下是如何实现的同样不关心，应用只知道请求最终顺利发送完成

Service Mesh产品的存在和具体工作模式，对于运行于其上的云原生应用来说是透明无感知的，但是在运行时这些能力都动态赋能给了应用，从而帮助应用在轻量化的同时依然可以继续提供原有的功能。

Mesh模式可以应用于更多的场景：不仅仅是服务间通讯

Mesh模式不仅仅可以用于服务间通讯，也可以应用于更多的场景：

- Database mesh：用于数据库访问
- Message mesh：用于消息系统

Mesh模式之外的探索

- ✓ 中间件下沉到基础设施的方式
  - 不只有Mesh模式一种
  - 也不是每个中间件都需要改造为Mesh模式
    - 有些是可以通过和mesh集成来间接提供能力的
- ✓ 更多的模式有待进一步探索和实践
  - DNS（探索中）
- ✓ 基本工作原理
  - 将功能实现从应用中剥离出来
    - 轻量化
  - 在运行时为应用 动态赋能

中间件下沉到基础设施的方式，不只是有Mesh模式一种，而且也不是每个中间件都需要改造为Mesh模式，比如前面我们提到有些中间件是可以通过与Mesh集成的方式来间接为应用提供能力，典型如监控，日志，追踪等。

我们也在探索mesh模式之外的更多模式，比如DNS模式，目前还在探索中。

简单归纳一下我们目前总结的云原生赋能（Cloud Empower）的基本工作原理：

- 首先要将功能实现从应用中剥离出来：这是应用轻量化的前提和基础
- 然后在运行时为应用 **动态赋能**：给应用的赋能方式也要云原生化，要求在运行时动态提供能力，

而应用无感知

本次畅谈云原生分享的上半场内容就到这里结果了，欢迎继续观看下半场内容。

如开头所说，这次分享是希望起到一个抛砖引玉的作用，期待后面会有更多同学出来就云原生这个话题进行更多的分享和讨论。也希望能有同学介绍更多云原生的实现模式，更多云原生的发展思路，拭目以待。