



畅谈云原生 (下)

蚂蚁金服中间件
服务与容器团队

前言

抛砖引玉 笨鸟先飞

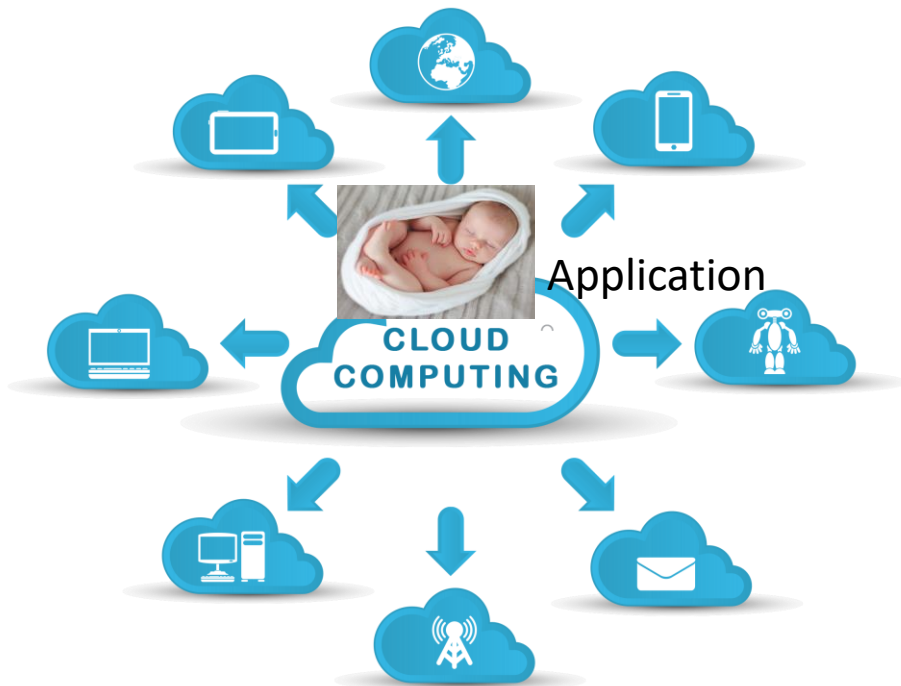
背景：2018年，我们团队（中间件服务与容器团队）在进行云原生产品的实践，摸石头过河中

内容：和大家一起聊一聊对云原生的理解，围绕几个需要深度思考的问题（所以本次的段落标题都是问句☺），介绍我们团队对这些问题的思考和正在探索的思路。

目标：抛砖引玉，就云原生这个话题开一个头

希望：后面有更多的同学继续分享云原生话题，给出更多精彩内容

抛砖引玉：云原生 -> 原生为云设计



云原生：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。

前情回顾2：云原生应用应该是什么样子？



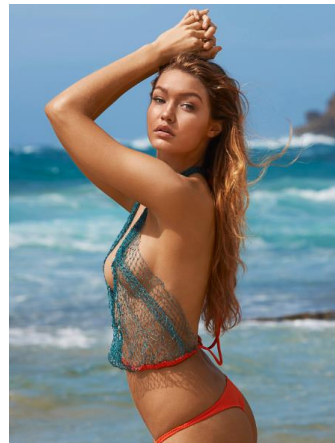
非原生



梦想



务实



理想

轻量化

抛砖引玉：云原生应用应该往轻量化方向努力😊

前情回顾3：云原生下的中间件该如何发展？



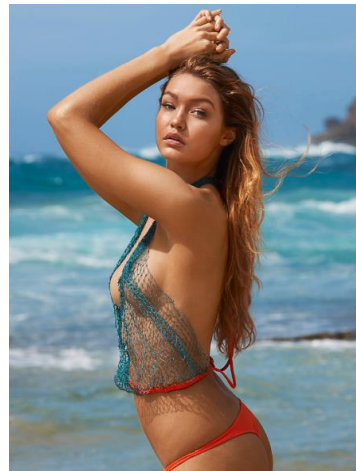
提供御寒衣服



云原生之前：

应用需要实现非常多的能力
(可以通过类库框架简化)

提供温暖的阳光



云原生：

加强和改善应用运行环境
(云)
帮助应用实现轻量化

云原生化



抛砖引玉：下沉到基础设施，赋能方式云原生化

内容

1

如何理解云原生？

2

云原生应用应该是什么样子？

3

云原生下的中间件该如何发展？

4

云和应用该如何衔接？

5

如何让产品更符合云原生？

6

花絮：有哪些有趣的角色转变？

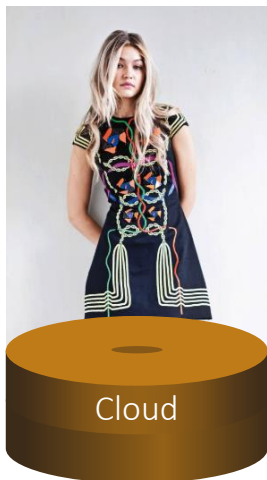
4

云和应用该如何衔接？



非原生

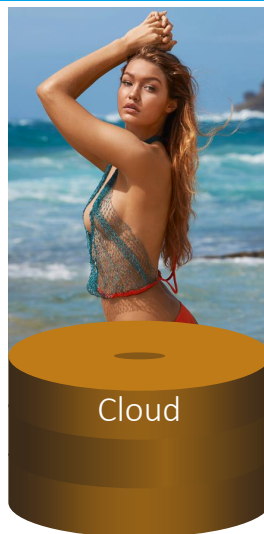
云只能提供**非常有限**的能力，应用需要自行实现



务实版

云原生

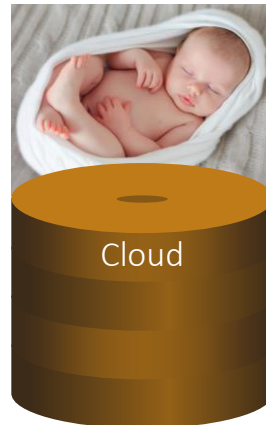
云提供**大部分**能力，部分云未能提供的能力依然需要应用自行实现



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

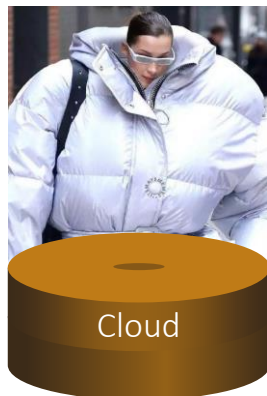
云提供**所有**能力，所有环节都完全解耦

先强调一下：需要应用配合，否则.....



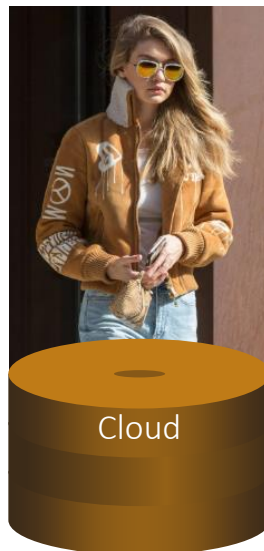
非原生

云只能提供有限的
能力，或者应用未
能利用云的能力



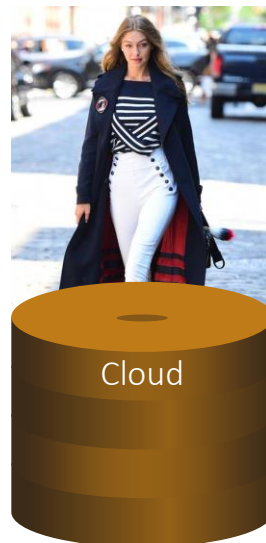
务实版 云原生

云提供大部分能力，
但是应用未能使用，
继续保持非原生应
用的形态



理想版 云原生

云提供绝大部分能
力，但是应用只利
用了部分能力



梦想版 云原生

云提供所有能力，
但是应用只利用了
部分能力

云原生：原生为云设计

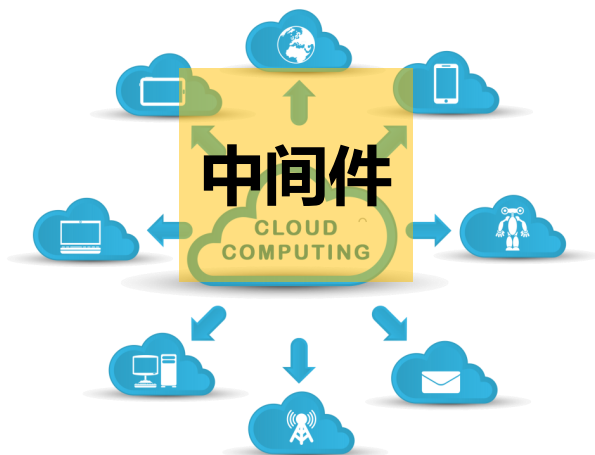
应用原生被设计为在云上以最佳方式运行，充分发挥云的优势

应用和云之间的衔接：赋能方式

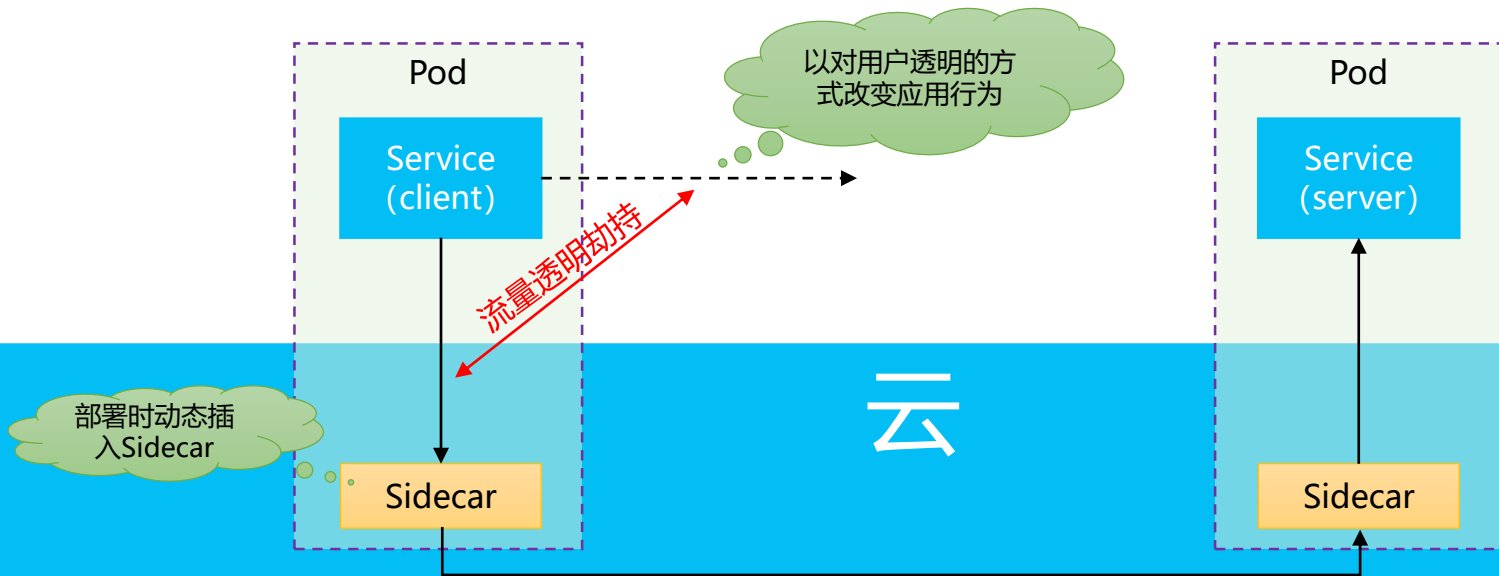


How?

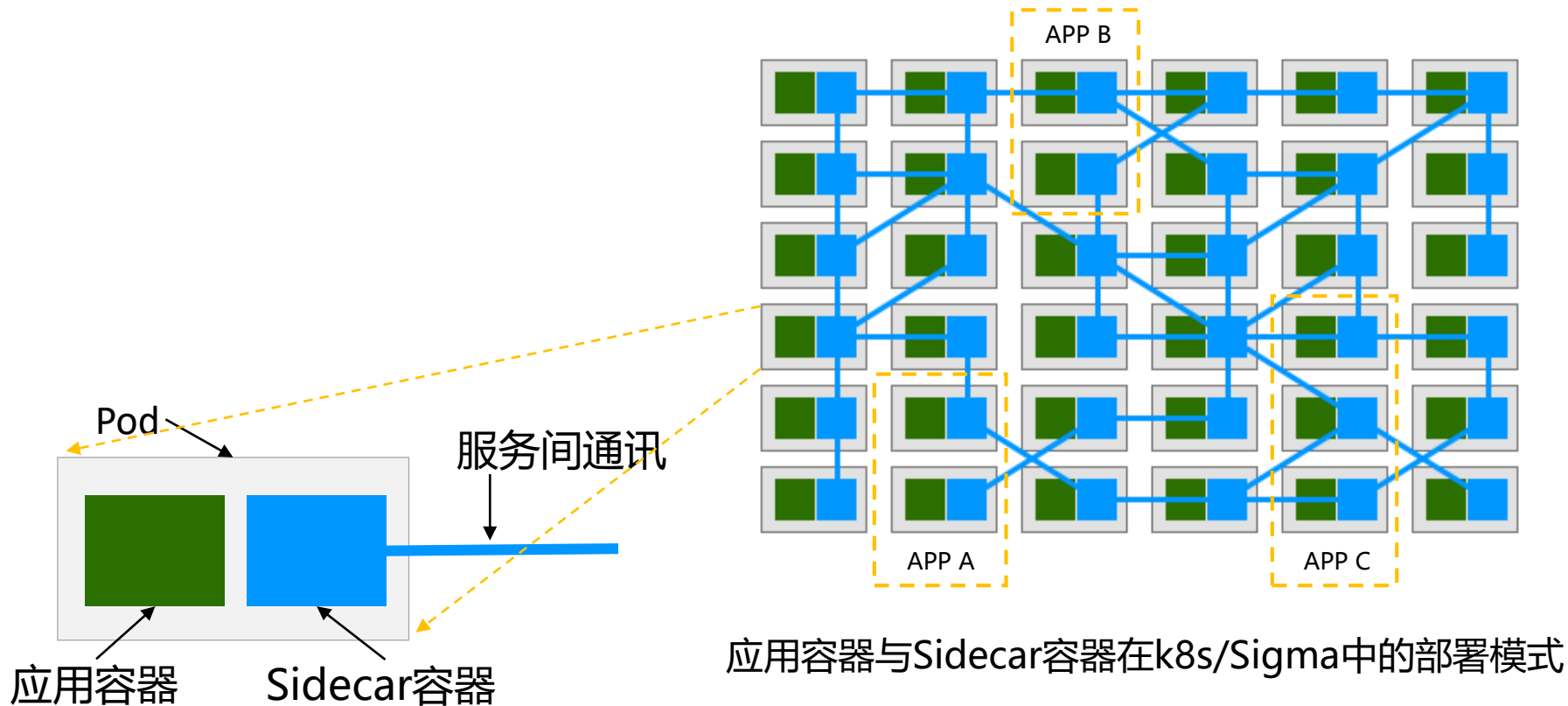
↑ 目标：在运行时为应用 **动态赋能**



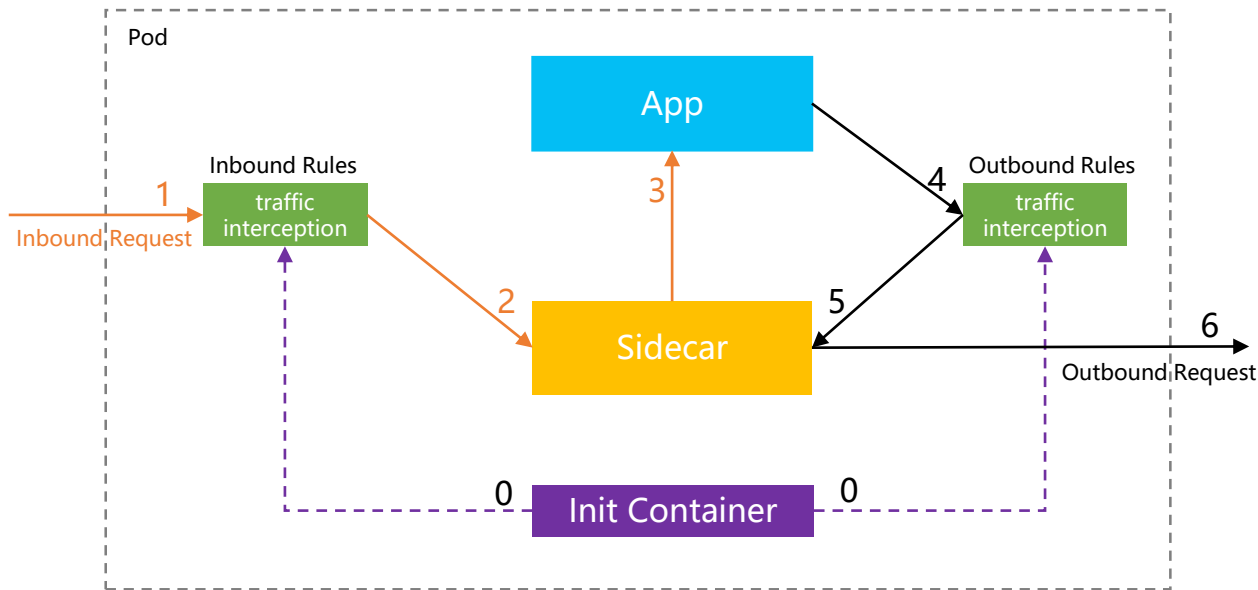
动态赋能的方式：流量透明劫持



透明劫持的部署模式



透明劫持的具体流程：以Istio iptables为例



- Init 流程(0): 在Pod启动时, 通过Init Container特权容器, 开启流量劫持并设置流量劫持规则 (分为 Inbound 规则和 Outbound 规则)。
- Inbound流程(1,2,3): Inbound请求, 被 traffic interception 劫持, 根据 Inbound规则请求被转发到Sidecar, 然后Sidecar再转发给应用
- Outbound流程(4,5,6): APP 发出的 Outbound 请求会被 traffic interception 劫持, 根据 Outbound 规则请求被转发给 Sidecar, 然后 Sidecar 处理之后将请求发送给目的地

	开启http_proxy	iptables透明劫持	ipvs透明劫持	sockmap透明转发
优点	支持度高，各技术栈广泛支持设置http代理	兼容度高，技术栈广泛支持	兼容度高，方案成熟，在LB场景业界已经大规模使用，经过生产验证，sidecar场景下性能足够	绕过协议栈，理论上性能有优势
缺点	http only，其他协议不支持	nf_conntrack 丢包/可维护性/规则数量多了之后性能差；管控性和可观测性不好	ipvs的原始设计是用于load balance，用于透明劫持是创新方式，未经验证；和iptables方案类似，管控性和可观测性不好	兼容度低，最低需要4.17内核，周边生态不完善，与cilium方案严重绑定

对代码无侵入

- 业务代码无需关注细节，也不需要依赖SDK
- 旧有代码可以无需改动就直接运行在service mesh上
- 避免修改代码，重新打包发布上线等复杂流程
- 支持直连/单挑，方便开发调试，和现有系统兼容



↑ 目标：在运行时
为应用 **动态赋能**

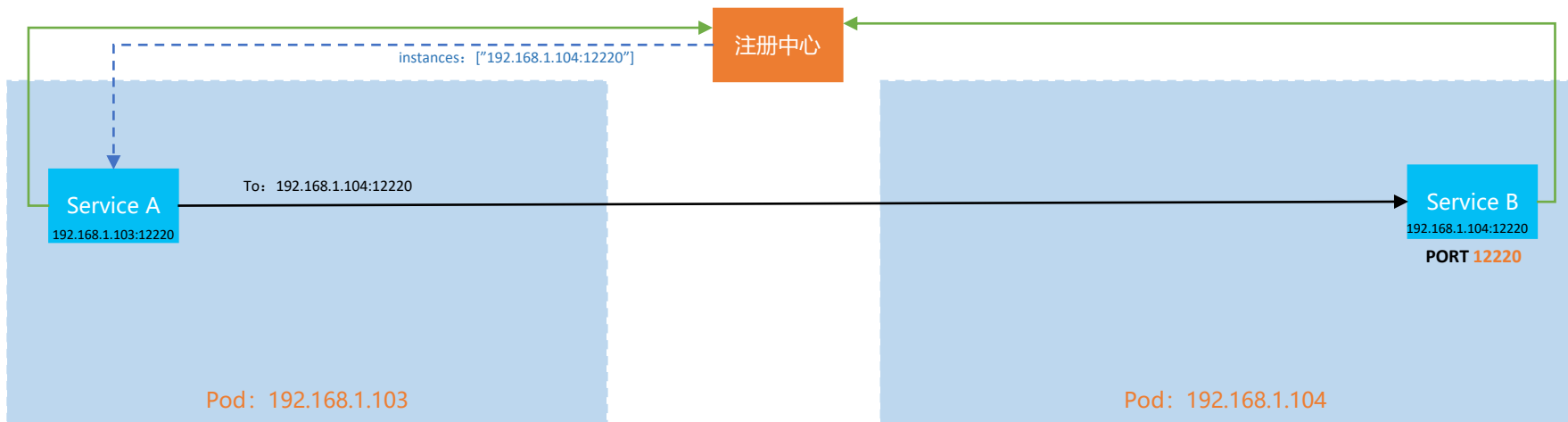
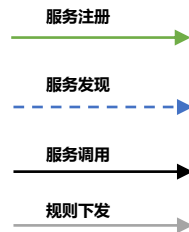
不丢失重要信息

- 原始目标地址和端口信息(original_dest)得以保留
- 更符合12 factor中“Port Binding”的要求
 - 见最后的花絮
- 容许服务在多个端口上绑定多个不同协议而Sidecar只需要一个端口

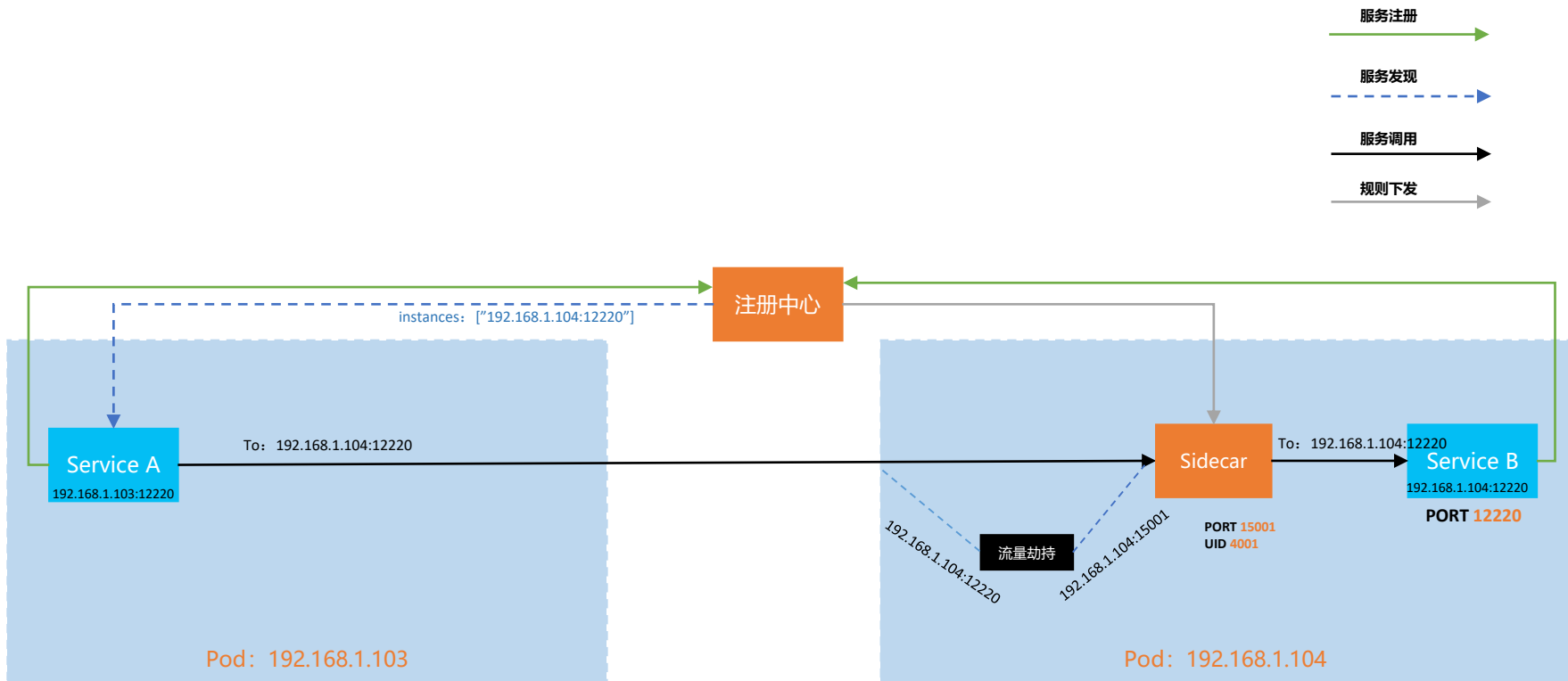
透明劫持的适用场景之一：平滑迁移



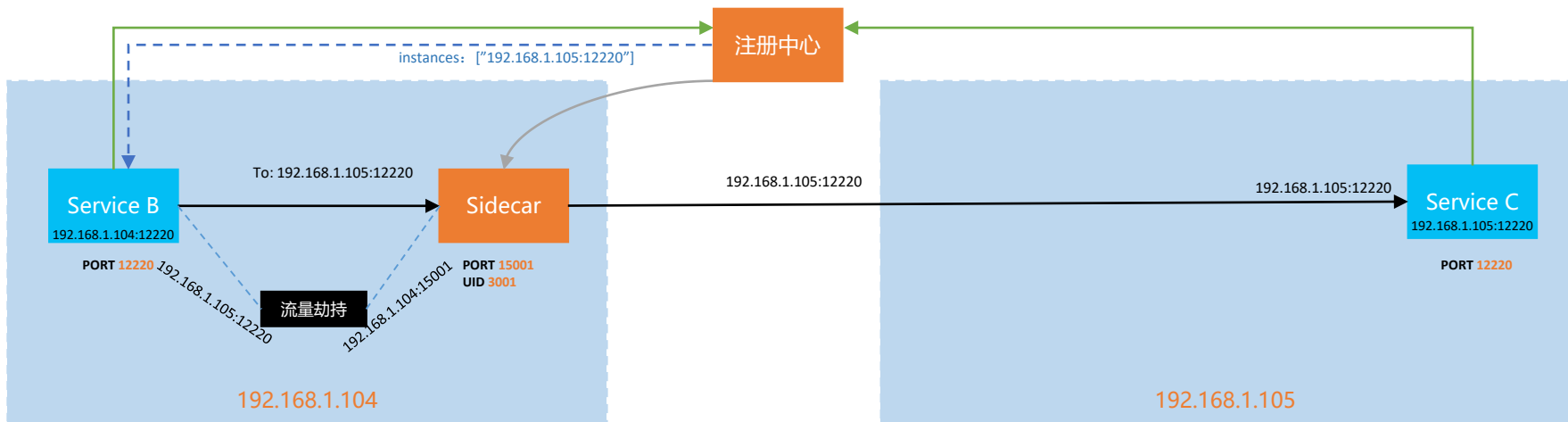
背景：将现有服务迁移到Service Mesh



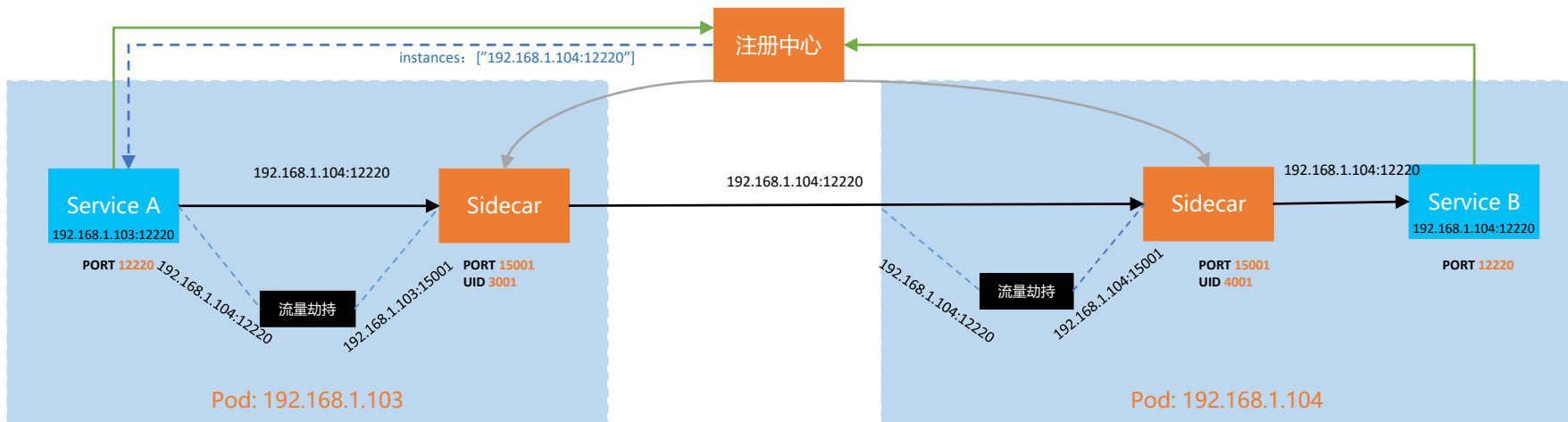
将服务迁移到Service Mesh: 注入Sidecar(Inbound)



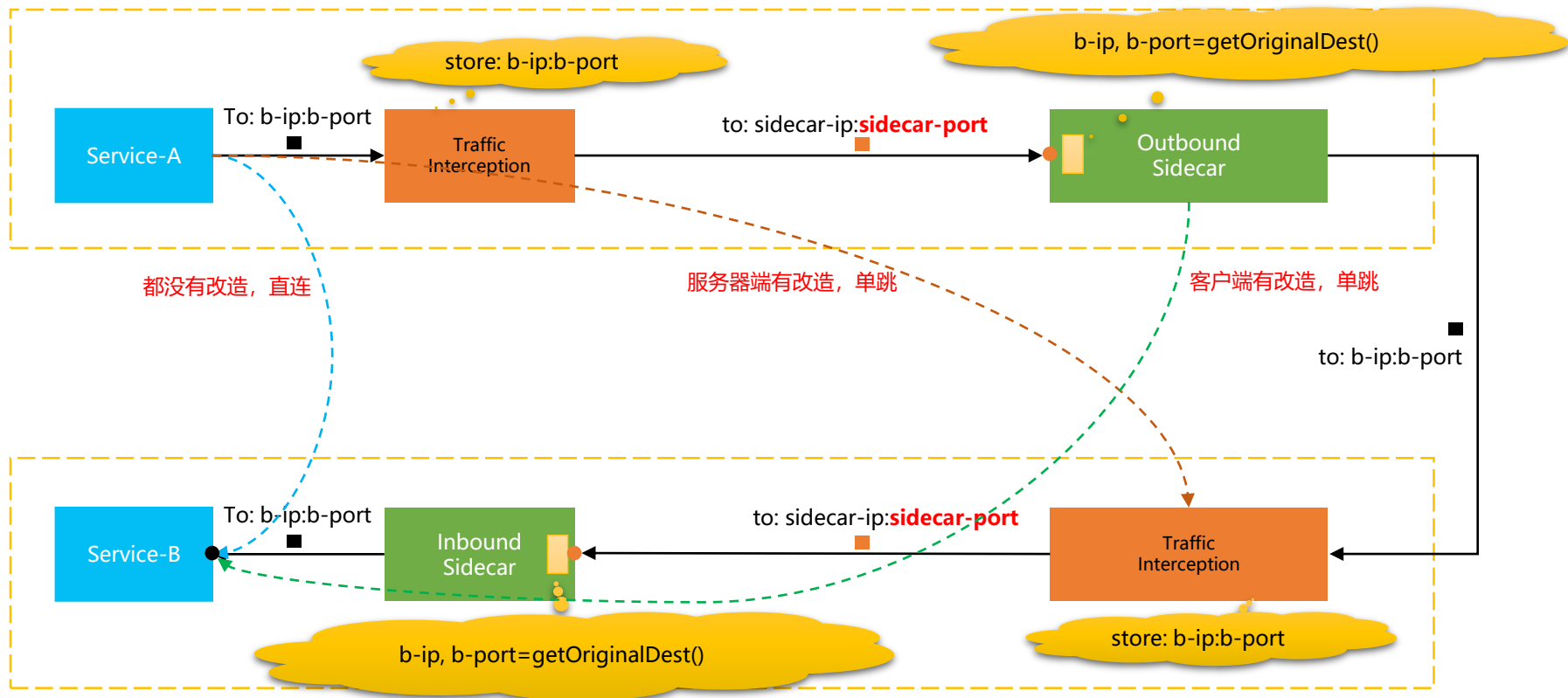
将服务迁移到Service Mesh: 注入Sidecar (outbound)



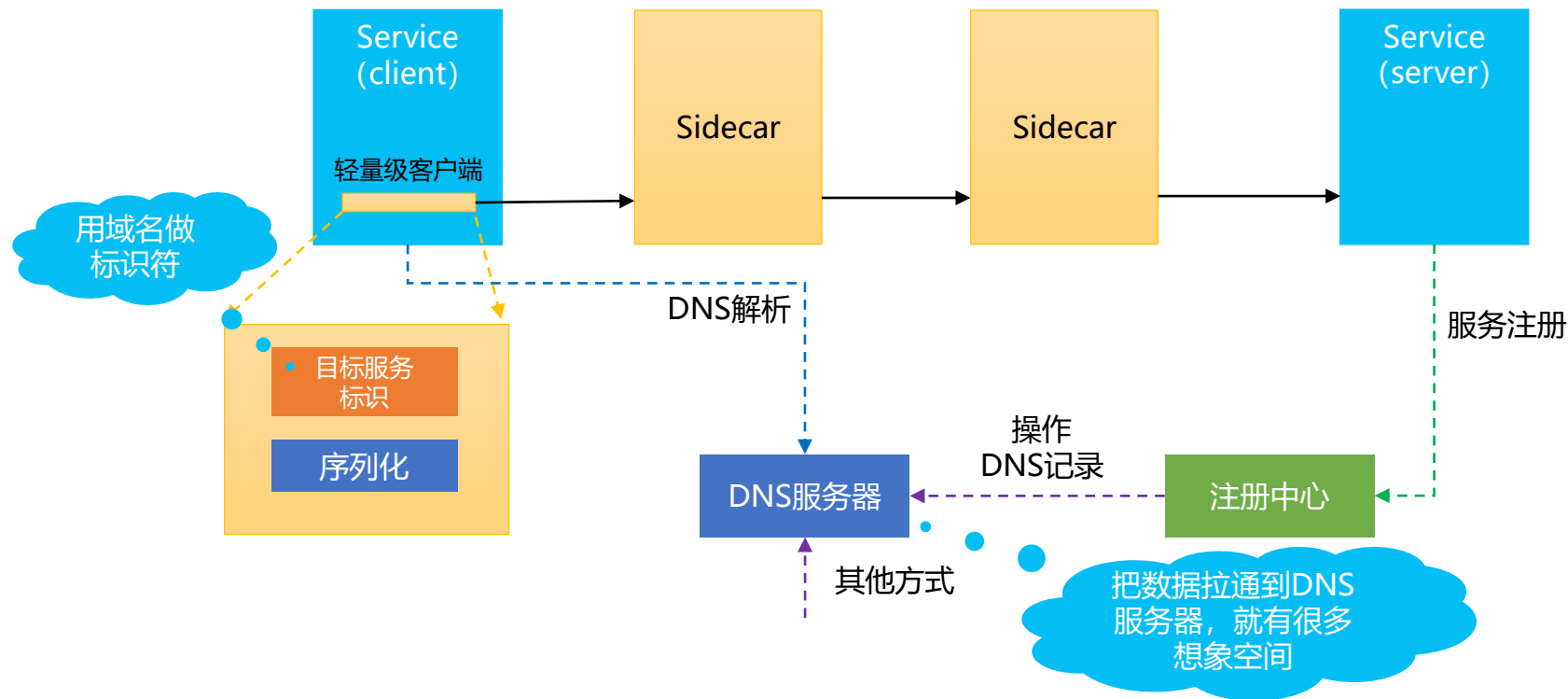
将服务迁移到Service Mesh: Inbound & Outbound



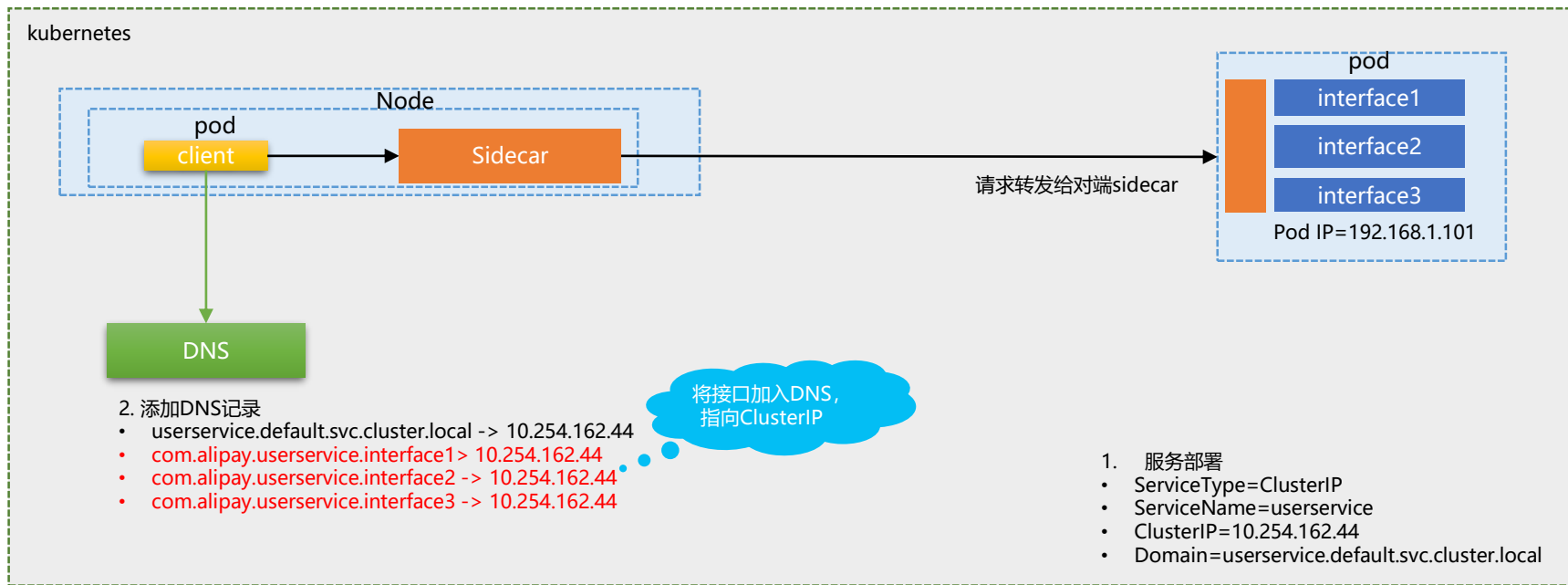
透明劫持带来的升级弹性：对应用透明，支持直连/单跳/双跳



动态赋能的方式：DNS



SOFAMesh的例子：使用DNS解决实现接口访问



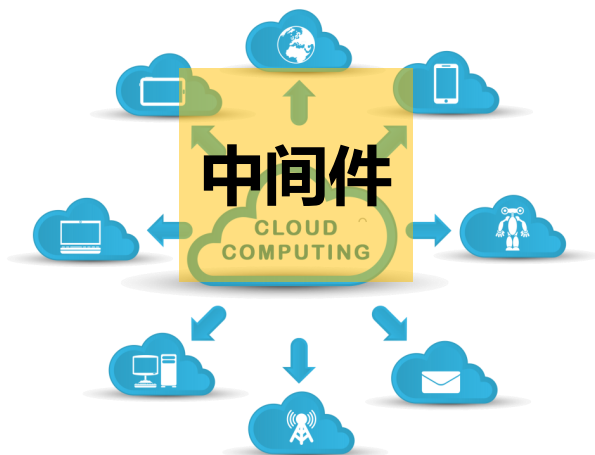
结合使用：服务注册 + DNS信息同步 + 流量透明劫持 + Sidecar逻辑处理

应用和云之间的衔接：能力控制方式



应用要
如何控制
这些能力？

↑ 目标：在运行时为应用 **动态赋能**



✓ 能力是动态赋予应用的

- 应用无法直接控制这些能力
- 应用也不应该知道这些能力的具体实现方式
- 我们的理解：命令式在这种场景下不合适

✓ 那应用能做什么？

- 应用肯定知道自己要达到的控制目标，即目标状态
- 我们的理解：声明式非常符合这个场景

应用声明

当我访问某个服务时：

- 要用轮询的负载均衡算法
- 要10%的流量去v2版本，其他的去v1版本
- 要开启链路加密
- 要.....

✓ 简单

- 应用不需要关心细节
- 实现方式/流程/细节都是能力提供方内部完成，对应用是透明的

✓ 自描述

- 声明描述的就是应用期望的目标状态

把方便留给别人，把麻烦留给自己

对于用的人，
Declarative模式
省力省心

Declarative模式设计
和实现的难度是远高于
Imperative模式的

关于云和应用如何衔接这个问题
目前我们能给出的方案远不够理想

期望未来会有更多更好的做法
欢迎探讨，欢迎指教

5

如何让**产品**更符合云原生？

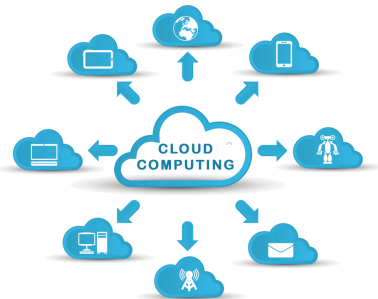
总结前面的规律：不仅提供功能，更强调体验

更舒适



少做事

更方便



Traditional On-Premises IT	Colocation	Hosting	IaaS	PaaS	SaaS
Data	Data	Data	Data	Data	Data
Application	Application	Application	Application	Application	Application
Databases	Databases	Databases	Databases	Databases	Databases
Operating System	Operating System	Operating System	Operating System	Operating System	Operating System
Virtualization	Virtualization	Virtualization	Virtualization	Virtualization	Virtualization
Physical Servers	Physical Servers	Physical Servers	Physical Servers	Physical Servers	Physical Servers
Network & Storage	Network & Storage	Network & Storage	Network & Storage	Network & Storage	Network & Storage
Data Center	Data Center	Data Center	Data Center	Data Center	Data Center

■ Provider-Supplied ■ Self-Managed

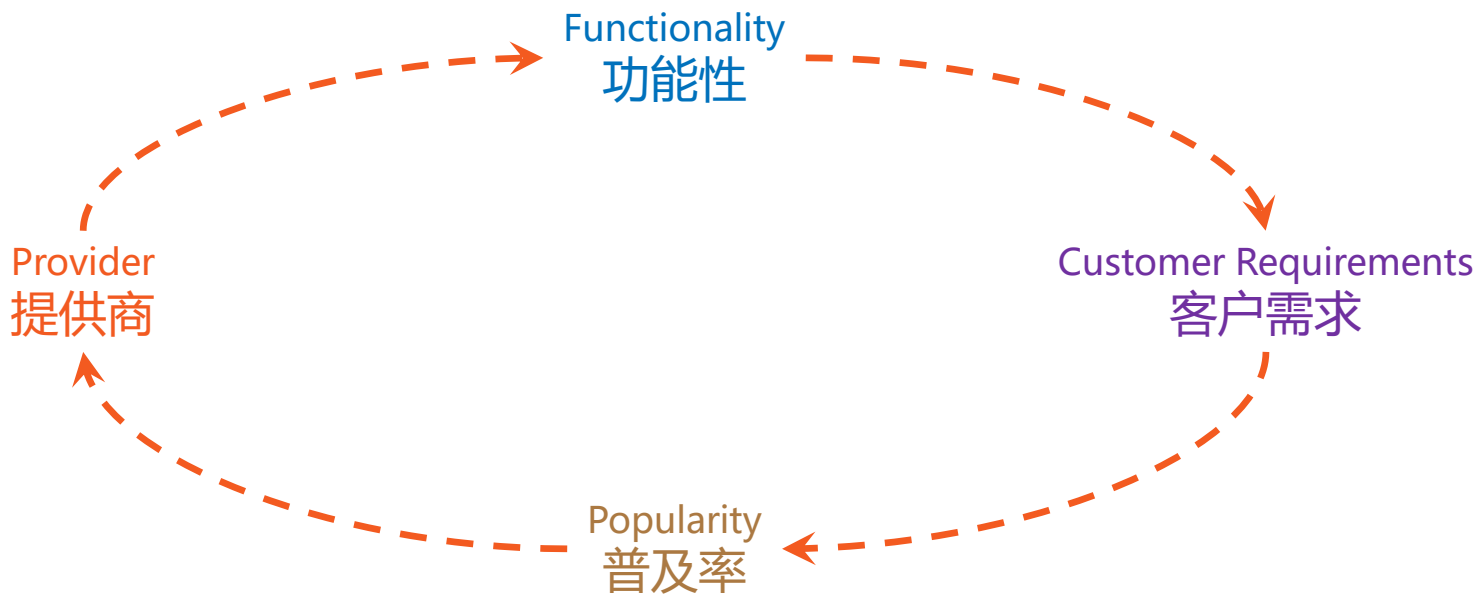
把方便留给别人

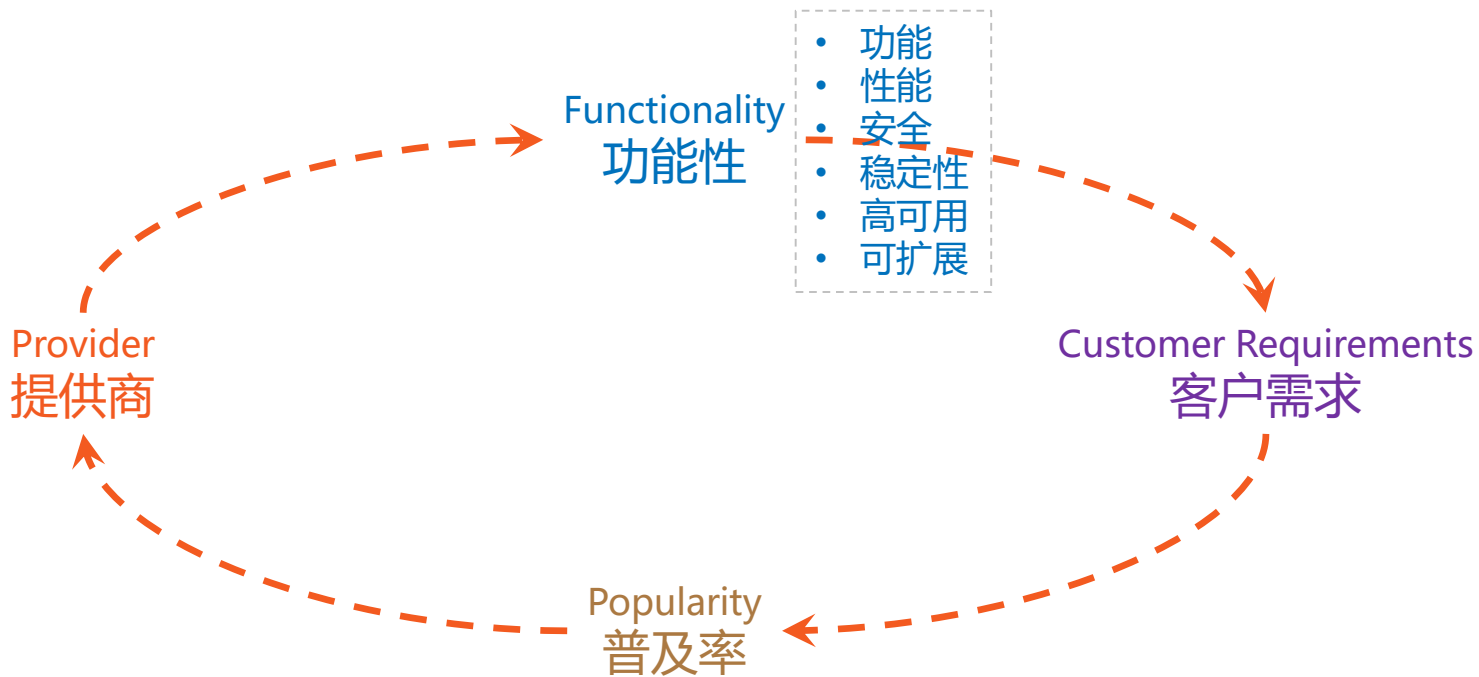
把麻烦留给自己



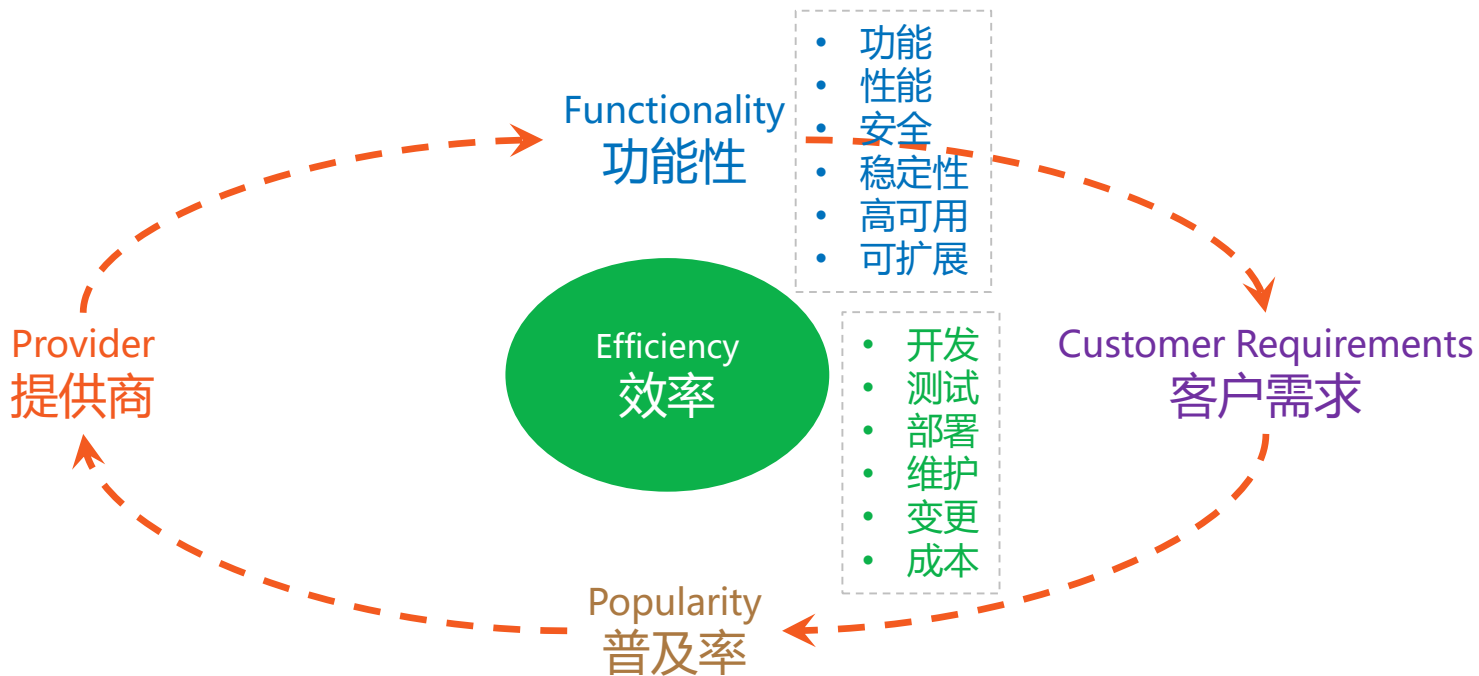
创意来源：亚马逊飞轮理论

云计算和云原生出现之前的大循环

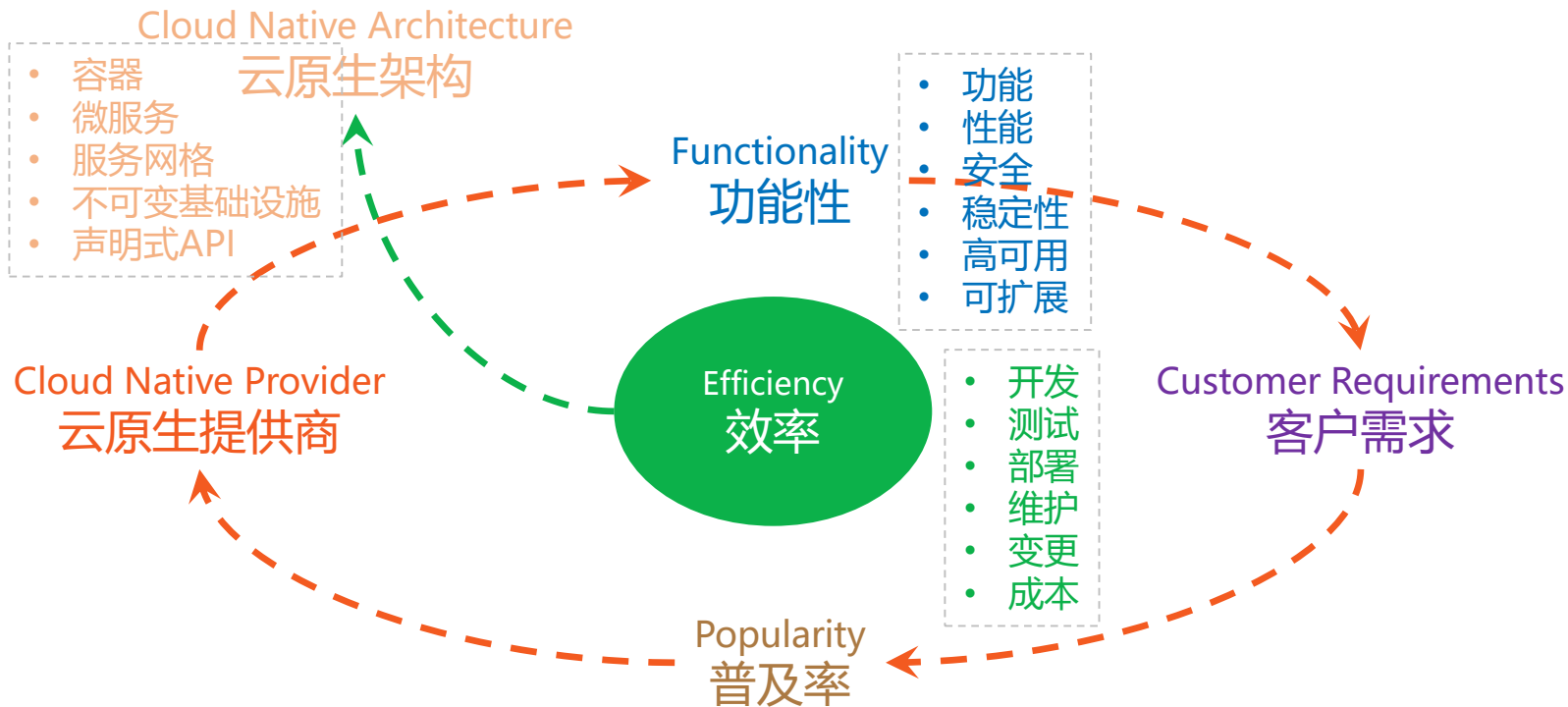




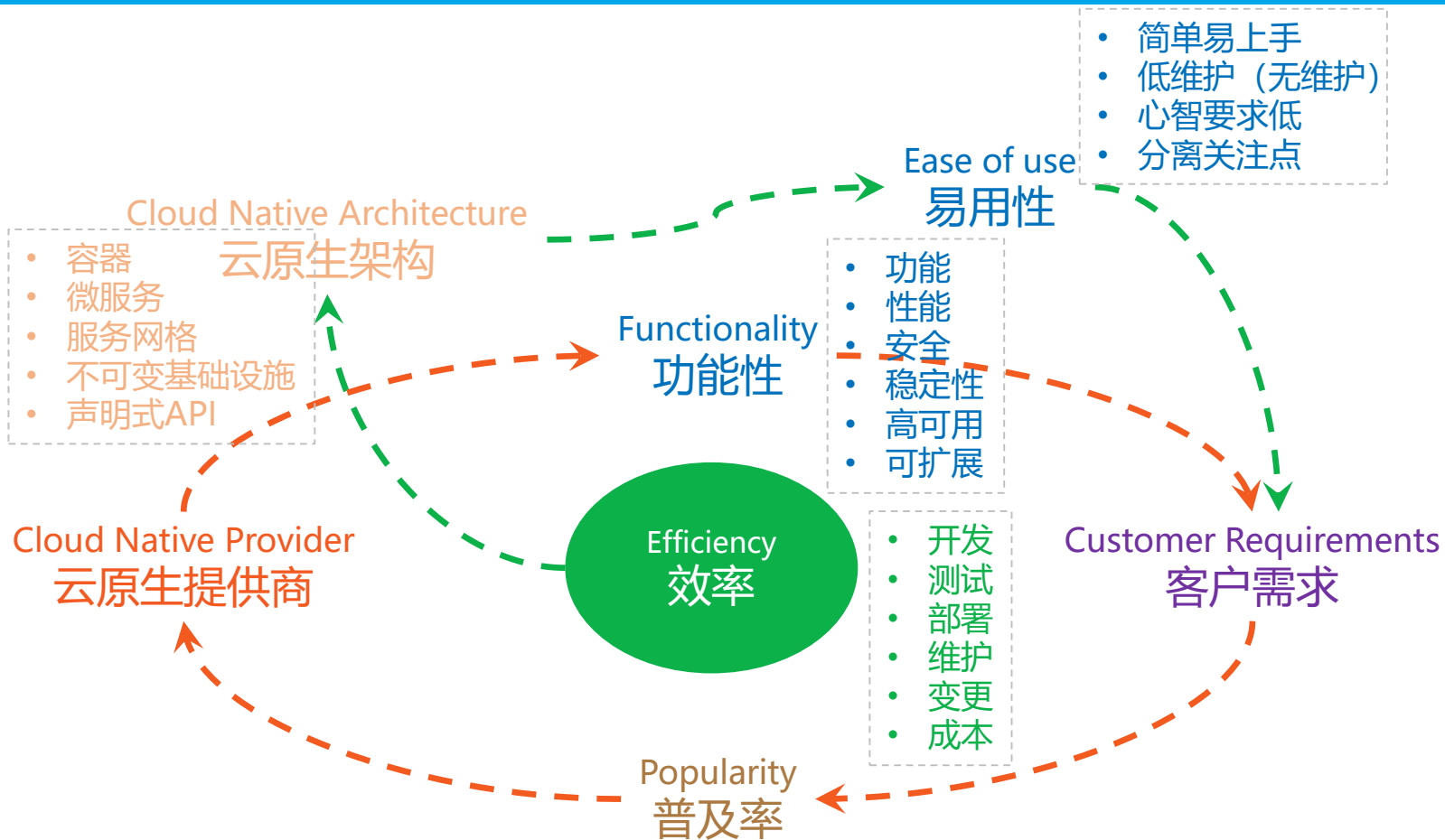
功能性之外，更多是对效率的追求



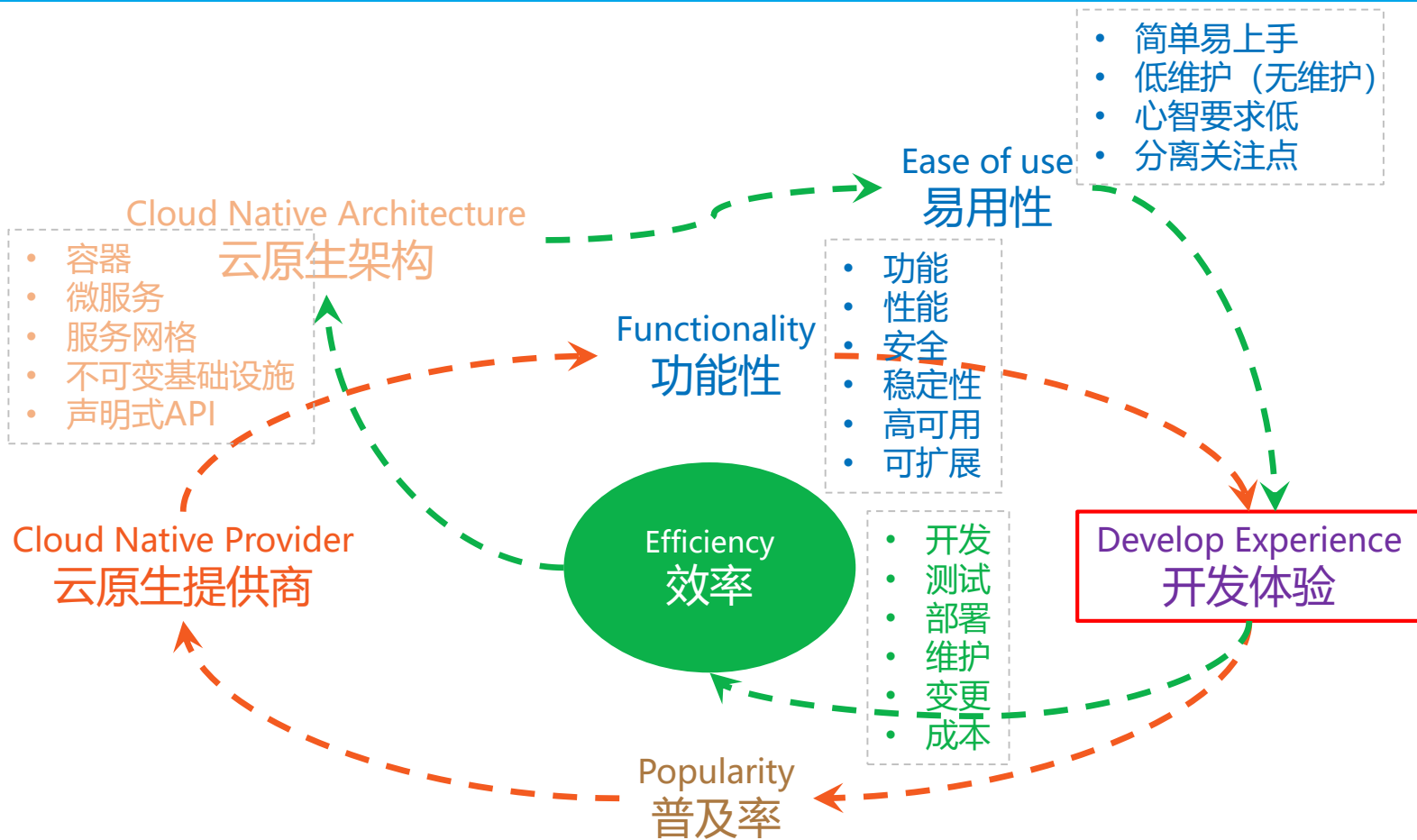
推动云计算和云原生架构的产生和发展



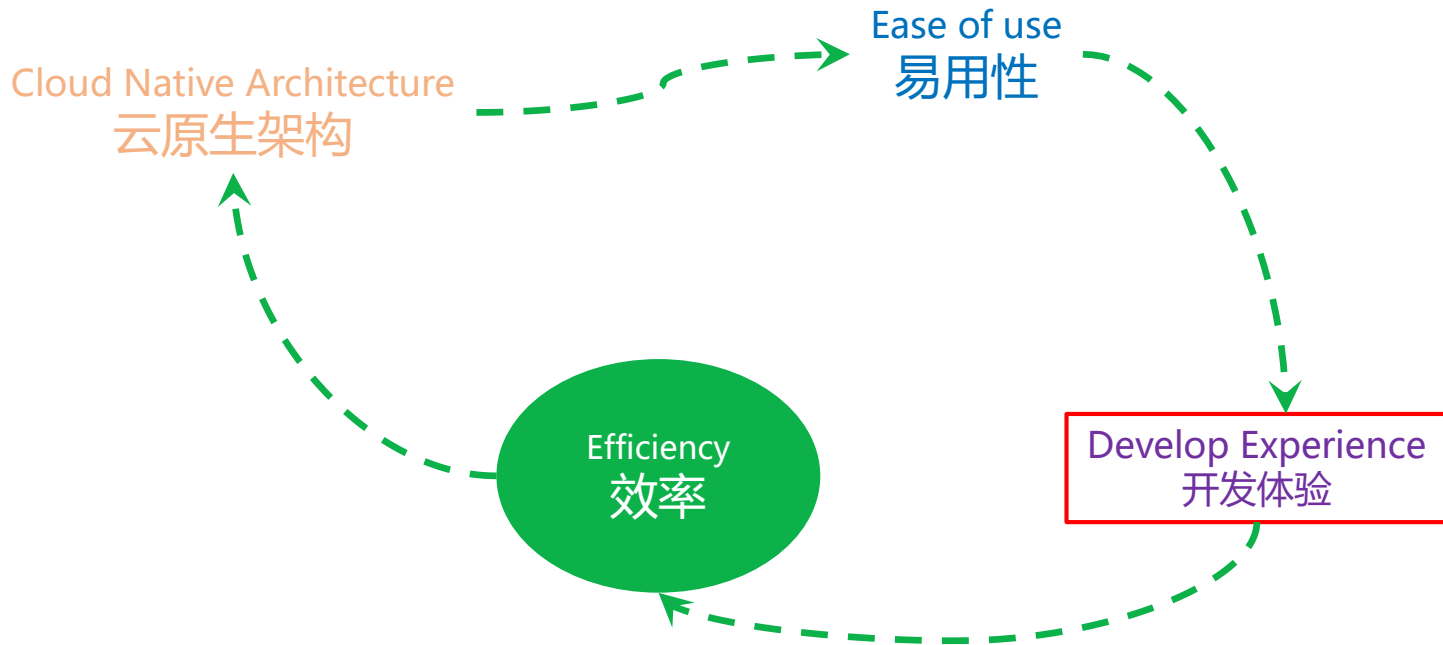
易用性方面有质的飞跃



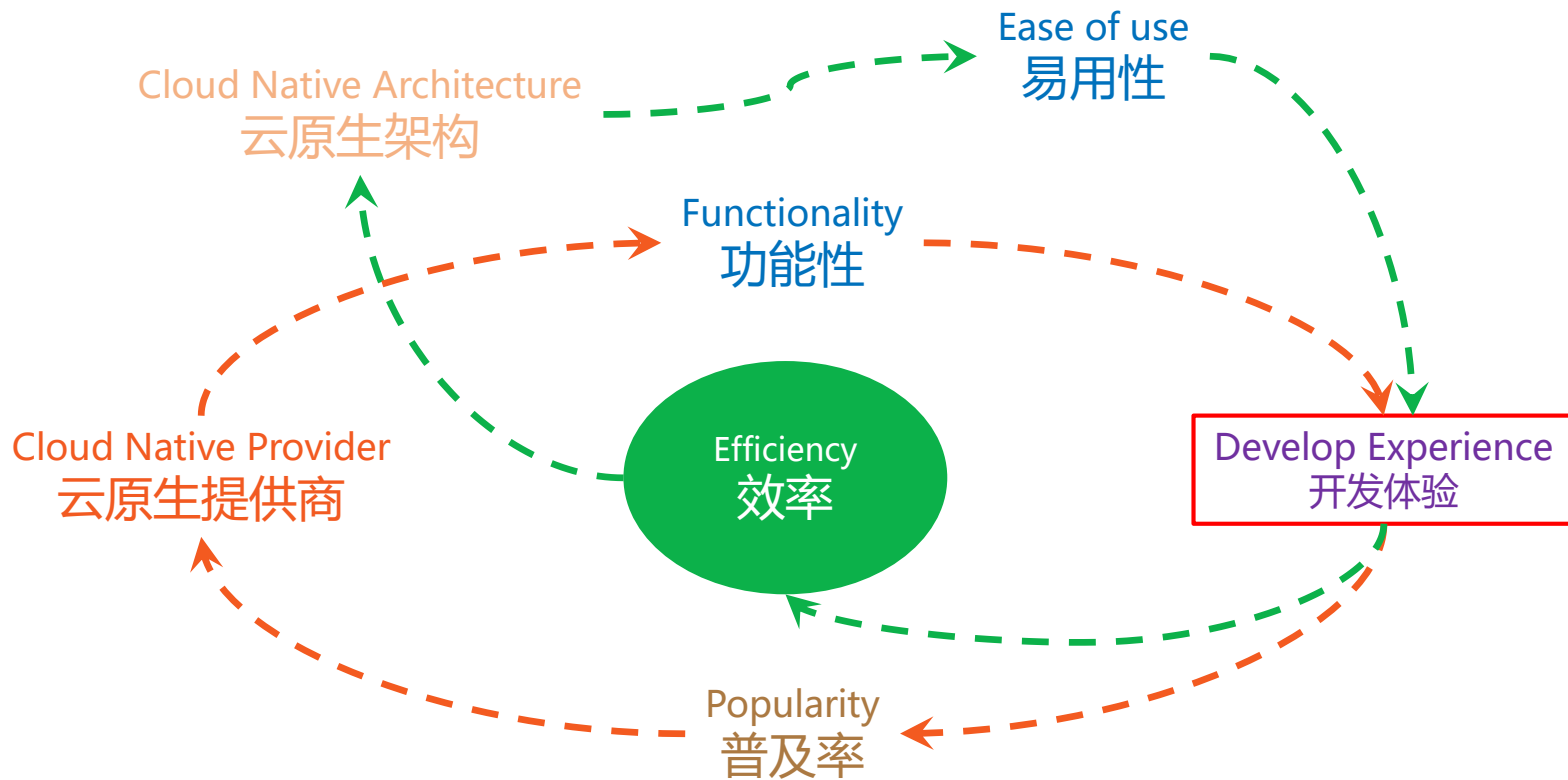
极大的改善了开发体验，并实现了效率的提升



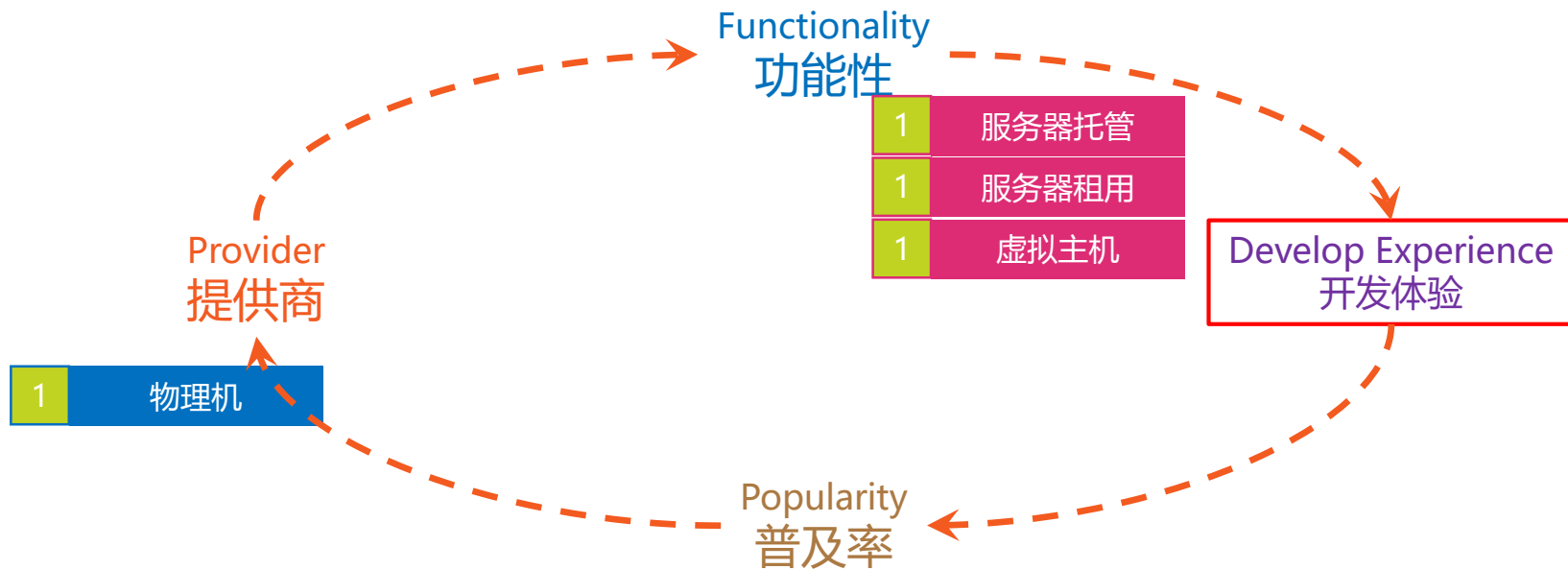
围绕易用性，新的闭环产生



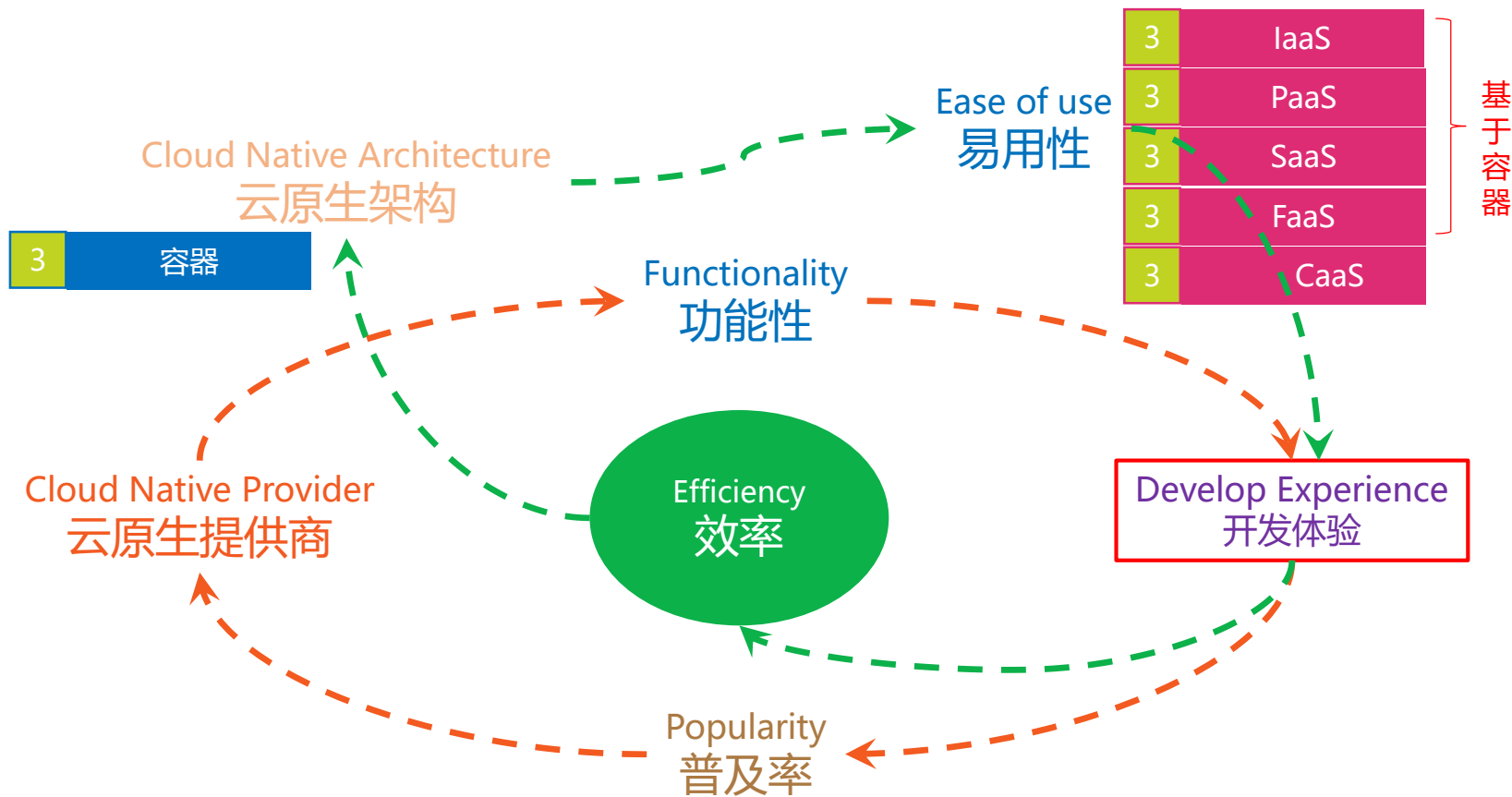
完整的云原生架构飞轮理论



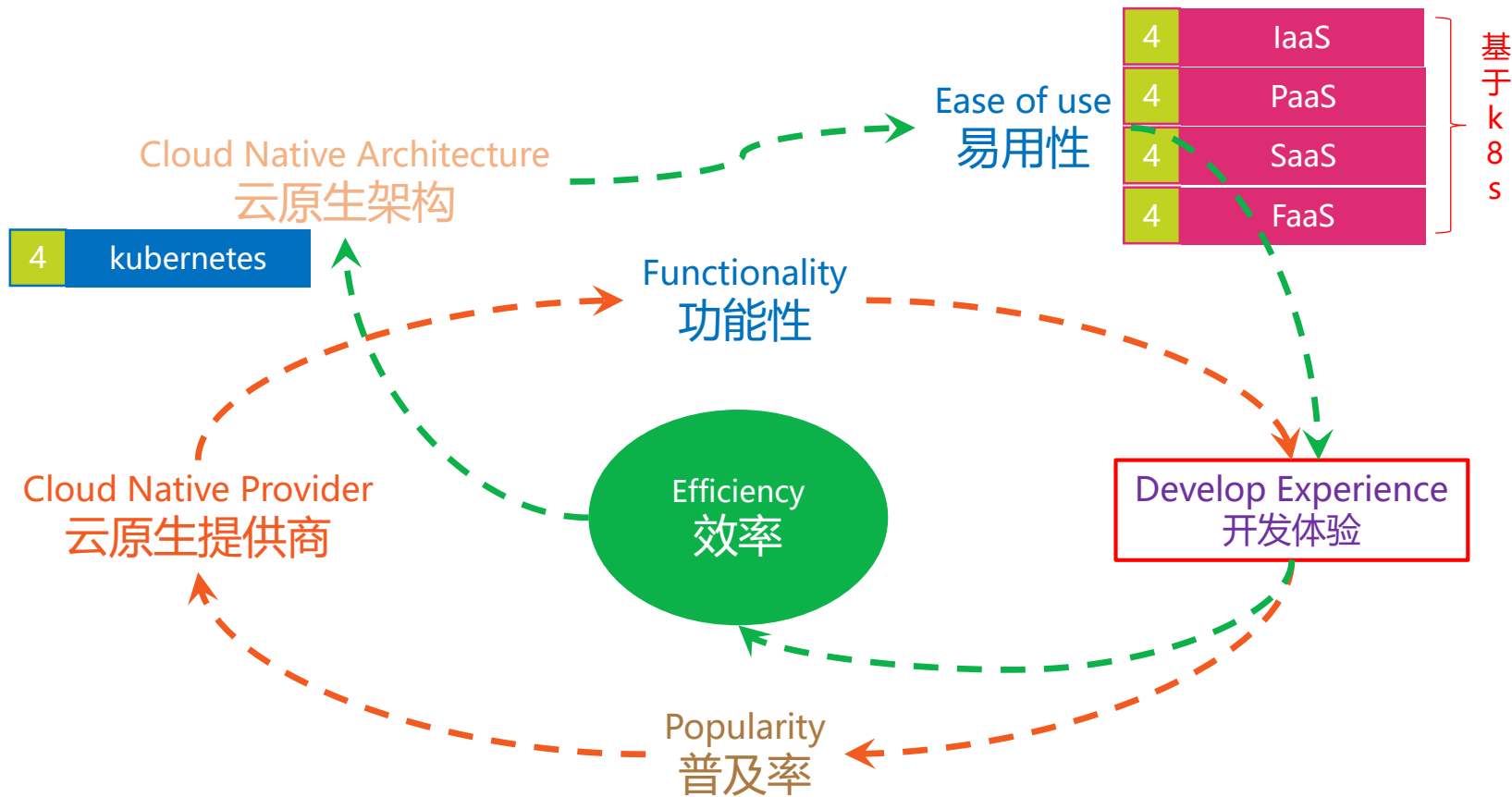
案例：虚拟化技术



虚拟化技术：Container




虚拟化技术: kubernetes



✓ 功能/性能之外

- 关注易用性
- 关注开发者体验

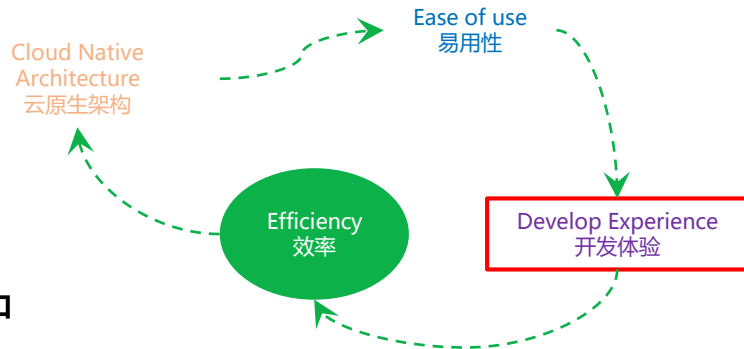
- 把应用和应用开发者当成  来呵护

✓ 依托云原生架构

- 基于云，基于容器，基于kubernetes

✓ 顺势而为

- 顺着飞轮的方向，迎合云原生和社区方向





Kubernetes是云计算和云原生时代的Linux

- 以kubernetes为底座进行能力建设
- 把kubernetes当kubernetes用

- CRD + Controller
- 如果k8s底座的能力不够
 - 补充和加强k8s的能力，实现新的Controller
- 如果k8s的抽象不够
 - 复杂场景，现有CRD不适用或不够用
 - 定义新的抽象，添加新的CRD
- 加固k8s底座（Controller）+ 扩展k8s抽象（CRD） -》新的云原生基础设施
- 声明式API，声明式API，声明式API!



- ✓ Istio 中定义的 CRD 数量多达50+
- ✓ 典型如用于网络和路由的抽象
 - VirtualService
 - DestinationRule
 - Gateway
 - ServiceEntry
 - EnvoyFilter
 - ServiceDependency
- ✓ Mixer adapter 和 adapter template



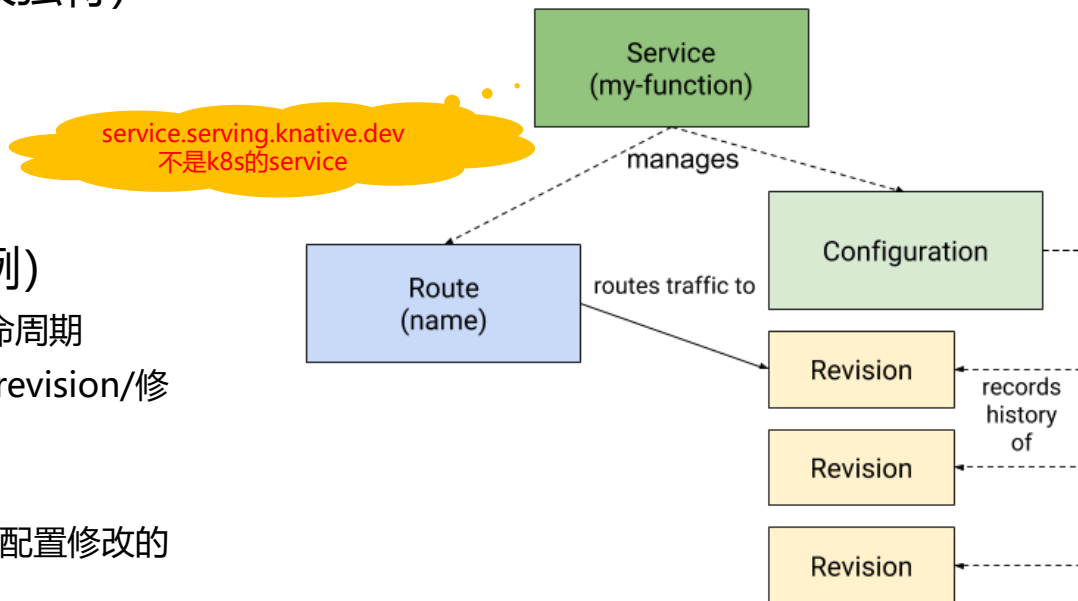


✓ Knative处理抽象的方式（比较独特）

- kubernetes 和 istio 本身的概念非常多
- 理解和管理，比较困难
- knative 提供了更高一层的抽象
- 基于 kubernetes 的 CRD 实现

✓ 抽象概念（以Serving模块为例）

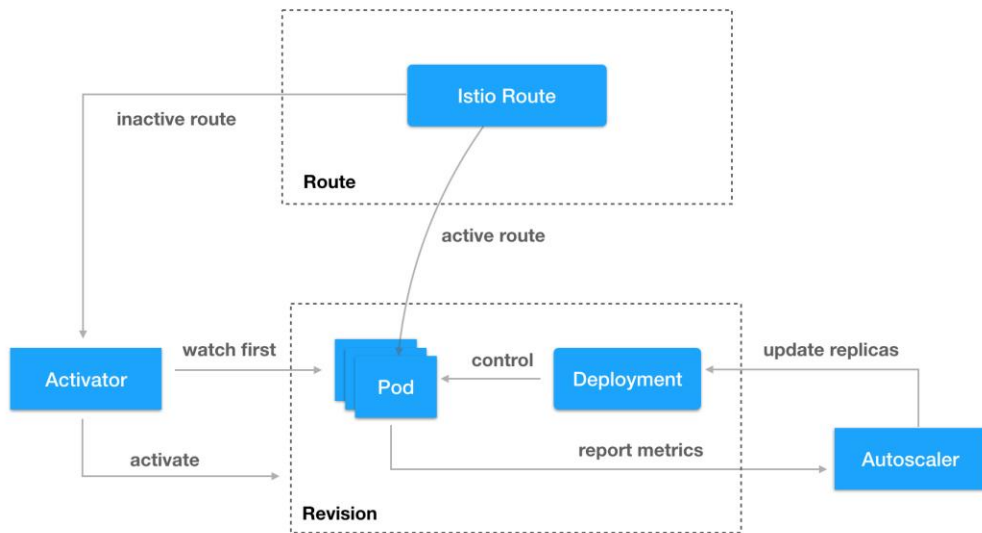
- Service: 自动管理工作负载的整个生命周期
- Route: 将网络端点映射到一个或多个revision/修订版本
- Configuration: 维护部署所需的状况
- Revision: 每次对工作负载进行代码和配置修改的时间点快照



- ✓ 在云原生基础设施上生长云原生产品
 - 尽量利用k8s和基础设施的能力
 - 尽量下沉通用能力到基础设施和k8s
 - 尽量将元数据收口到k8s



利用k8s能力的例子：Knative的 Autoscaler 的实现



- ✓ 目前 autoscaler 是knative自行实现的
- ✓ 计划转向采用 k8s 的原生能力
 - HPA (Horizontal Pod Autoscaler)
 - Custom Metrics

✓ 业界标准 (CNCF社区)

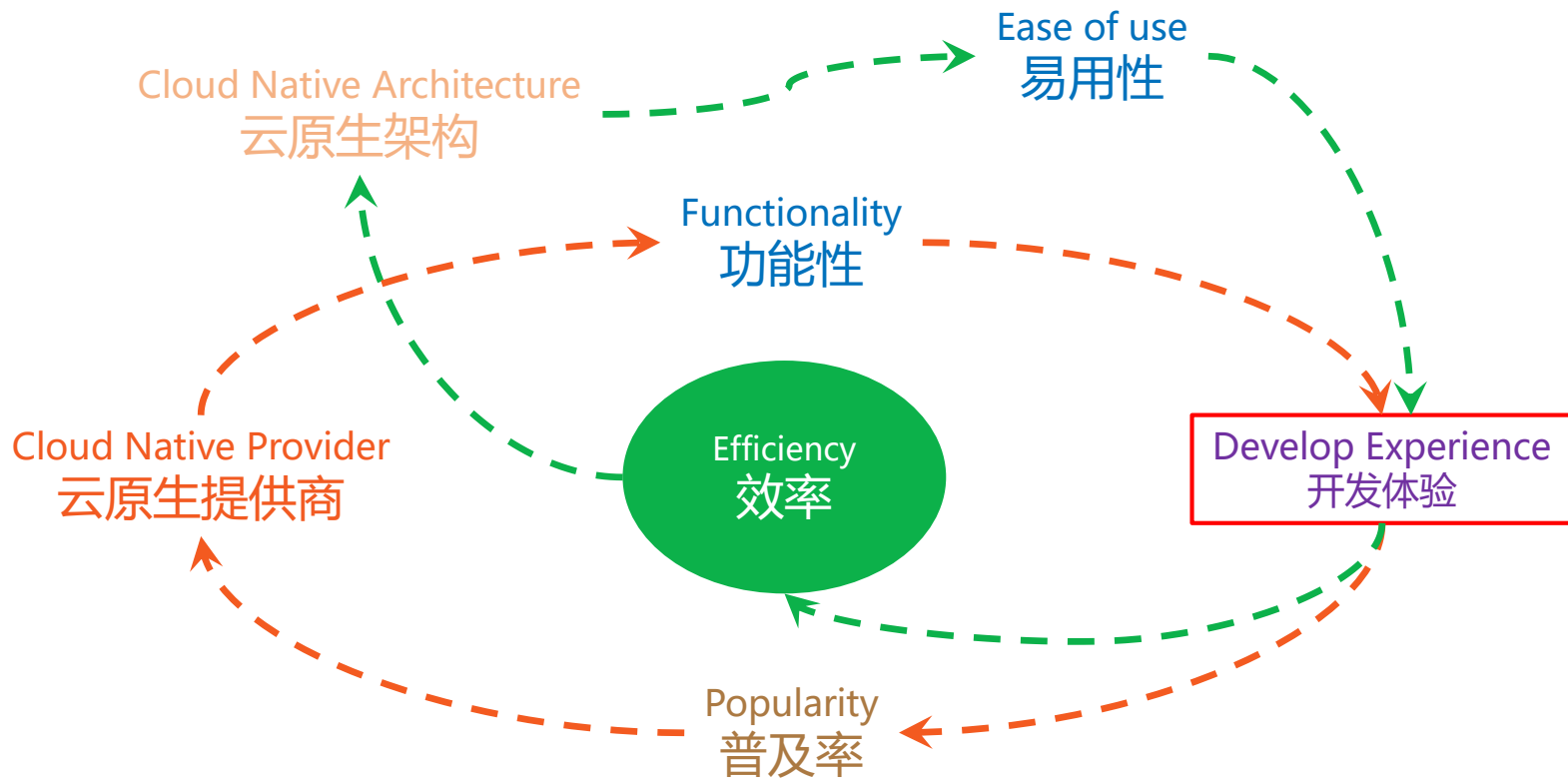
- CNI
- OpenTracing
- Open Policy Agent
- CloudEvents
- OpenMetrics
- SPIFFE

✓ 事实标准 (CNCF社区)

- Kubernetes
- Prometheus
- Envoy xDS协议
- gRPC



尽量不要逆势而为：顺着飞轮转



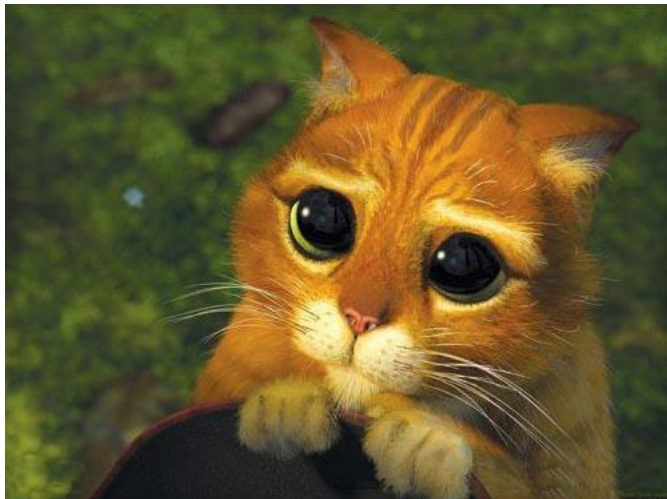
我们这次只带来了一些比较基础的想法
更具体更深入的建议应该结合实际产品讲

期望后续有同学带来深入分享

6

花絮：有哪些有趣的角色转变？

宠物



牲口

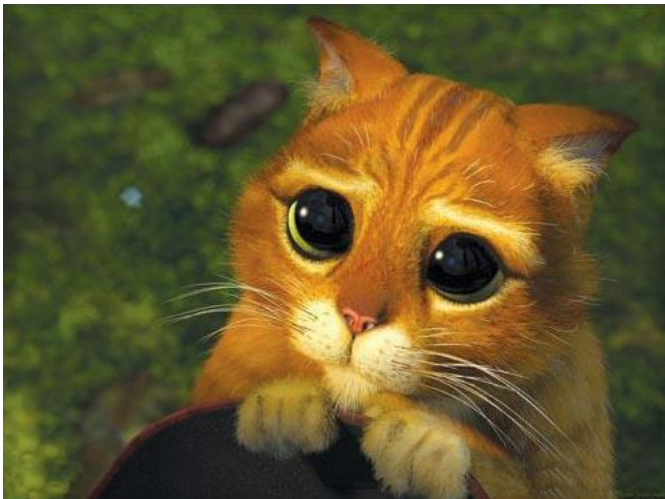


问：如何判断某物是宠物还是牲口？

答：简单评估一下：如果它发生失败无法工作，你是倾向于让它恢复，还是倾向于简单抛弃然后拿另一个替换。

在云原生时代，有哪些概念发生了角色转变？

从宠物到牲口



IP address!

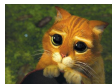
从牲口到宠物



Port?



✓ 在云原生(容器)之前


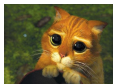


- IP地址非常重要
- 几乎等同于一台机器（物理机或虚拟机）的标识
- 固定，不轻易变更
- 外部通过IP地址进行访问

✓ 云原生(容器)时代



- 容器被频繁创建和销毁
- 容器的IP地址不再固定，而是动态变化
- 容器的IP地址不再适合作为机器标识
- 在k8s中取而代之的是 Label 和 Label Selector

- ✓ 当牲口的用法 
 - 端口之作为服务注册信息的一个普通组成部分
 - 应用动态选择端口进行监听，在服务注册信息中携带短信信息
 - 访问应用时，通过服务发现得到地址和端口信息
- ✓ 当宠物的用法 
 - 端口明确和协议绑定
 - 提供服务的应用应该显式的绑定并监听该端口
 - 客户端也明确的使用该端口，而不是在服务发现时动态决定

12 Factor: Port Binding



✓ Port Binding原则

- export services via port binding, The twelve-factor app is completely self-contained
- Bind every service to a port and listen on that port; **don't rely on runtime server injection**
- Twelve factor apps are self-contained, stateless and share-nothing processes and **don't depend on any runtime injection** for creating web-facing services. The only thing they should do is to bind to a port on the underlying execution environment and the app services are exported over that port.

✓ 详细解释

- Port Binding 指的是应用应该通过端口绑定的方式来导出服务
- 核心思想：十二因素应用程序应该是完全自包含的（self-contained），无状态和无共享的进程。
- 因此不应该依赖于任何运行时注入端口，或者运行时动态设置端口。
- 端口应该是和协议绑定的，提供服务的应用应该显式的绑定并监听该端口
- 而使用该服务的客户端应用，就应该明确的使用该端口。

✓ 目前Envoy / Istio 等都遵循该原则

期待更多分享