云原生

如何理解云原生

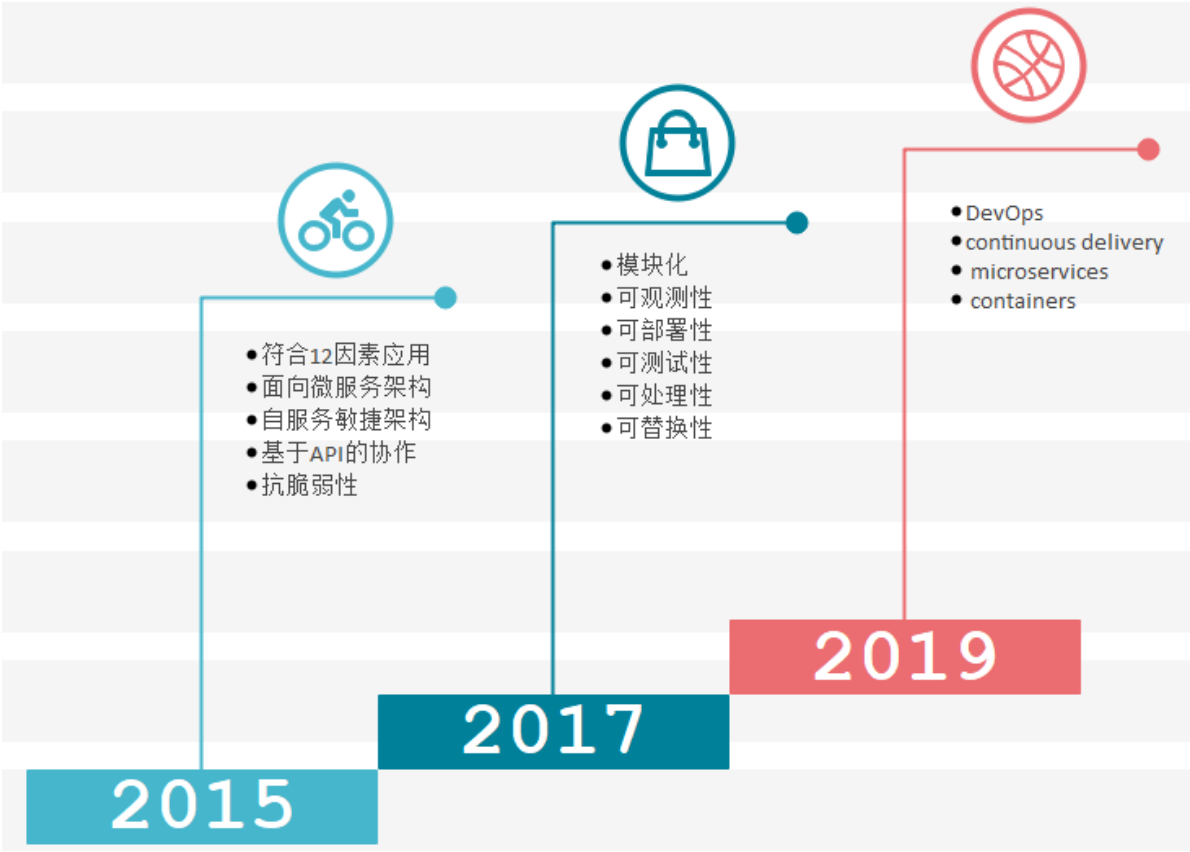
如何理解“云原生”？之所以将这个话题放在前面，是因为，这是对云原生概念的最基本的理解，而这会直接影响到后续的所有认知。
每个人对云原生的理解都可能不同，就如莎士比亚所说：

> 一千个人眼中有一千个哈姆雷特

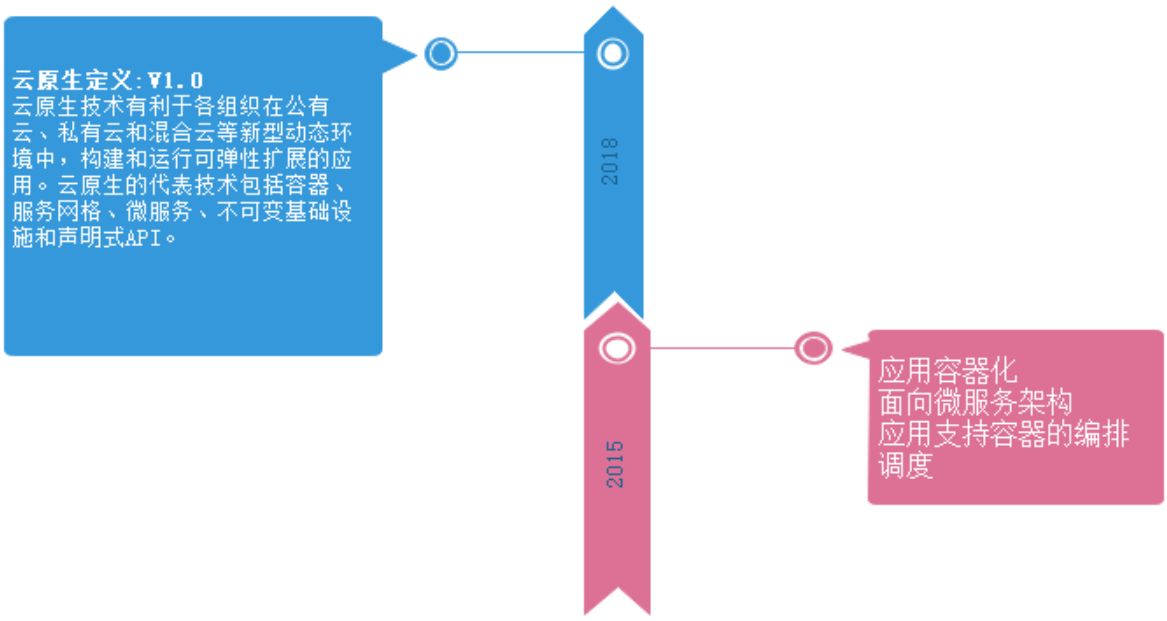
云原生的定义

Pivotal[1] 的定义

pivotal 是云原生应用的提出者，可以说是云原生的先驱或探路者。



CNCF 的定义



最新定义里面容器和微服务都提到了，服务网格是提到了和微服务、容器 并列，和我们理解的是微服务的一种实现有所不同。

cncf 的 [cloud native landscape]

- 自动化、配置中心
- 容器管理
- 容器编排
- 协同、服务发现
- 远程调用
- API 服务网关
- 服务网格
- 监控、日志、链路跟踪
- 数据库
- 流和消息处理
- 应用开发、镜像构建和持续集成
- 云原生存储
- 平台管理
- ...
- ...

> 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?

从字面意思上来看可以分成Cloud(云)和Native(原生)两个部分。

- 云是和本地相对的，传统的应用必须跑在本地服务器上，现在流行的应用都跑在云端，云包含了IaaS,、PaaS和SaaS。

  > 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。

- 原生就是土生土长的意思，我们在开始设计应用的时候就考虑到应用将来是运行云环境里面的，要充分利用云资源的优点，比如云服务的弹性和分布式优势

合起来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。

云原生应用有哪些优势

云原生应用	传统的企业应用
可预测。 云原生应用符合旨在通过可预测行为最大限度提高弹性的框架或“合同”。云平台中使用的高度自动化的容器驱动的基础架构推动着软件编写方式的发展。	**不可预测。** 传统应用的架构或开发方式使其无法实现在云原生平台上运行的所有优势。此类应用通常构建时间更长，大批量发布，只能逐渐扩展，并且会发生更多的单点故障。
操作系统抽象化。 云原生应用架构要求开发人员使用平台作为一种方法，从底层基础架构依赖关系中抽象出来，从而实现应用的简单迁移和扩展。实现云原生应用架构最有效的抽象方法是提供一个形式化的平台。	**依赖操作系统。** 传统的应用架构允许开发人员在应用和底层操作系统、硬件、存储和支持服务之间建立紧密的依赖关系。这些依赖关系使应用在新基础架构间的迁移和扩展变得复杂且充满风险，与云模型相背而驰。
合适的容量。 云原生应用平台可自动进行基础架构调配和配置，根据应用的日常需求在部署时动态分配和重新分配资源。基于云原生运行时的构建方式可优化应用生命周期管理，包括扩展以满足需求、资源利用率、可用资源编排，以及从故障中恢复，最大程度减少停机时间。	**过多容量。** 传统 IT 会为应用设计专用的自定义基础架构解决方案，这延迟了应用的部署。由于基于最坏情况估算容量，解决方案通常容量过大，同时几乎没有能力继续扩展以满足需求。
协作。 云原生可协助 DevOps，从而在开发和运营职能部门之间建立密切协作，将完成的应用代码快速顺畅地转入生产。	**孤立。** 传统 IT 将完成的应用代码从开发人员“隔墙”交接到运营，然后由运营人员在生产中运行此代码。企业的内部问题之严重以至于无暇顾及客户，导致内部冲突产生，交付缓慢折中，员工士气低落。
持续交付。 IT 团队可以在单个软件更新准备就绪后立即将其发布出去。快速发布软件的企业可获得更紧密的反馈循环，并能更有效地响应客户需求。持续交付最适用于其他相关方法，包括测试驱动型开发和持续集成。	**瀑布式开发。** IT 团队定期发布软件，通常间隔几周或几个月，事实上，当代码构建至发布版本时，该版本的许多组件已提前准备就绪，并且除了人工发布工具之外没有依赖关系。如果客户需要的功能被延迟发布，那企业将会错失赢得客户和增加收入的机会。

云原生应用	传统的企业应用
独立。 微服务架构将应用分解成小型松散耦合的独立运行的服务。这些服务映射到更小的独立开发团队，可以频繁进行独立的更新、扩展和故障转移/重新启动操作，而不影响其他服务。	**依赖。** 一体化架构将许多分散的服务捆绑在一个部署包中，使服务之间出现不必要的依赖关系，导致开发和部署过程丧失敏捷性。
自动化可扩展性。 大规模基础架构自动化可消除因人为错误造成的停机。计算机自动化无需面对此类挑战，可以在任何规模的部署中始终如一地应用同一组规则。云原生还超越了基于以虚拟化为导向的传统编排而构建的专用自动化。全面的云原生架构包括适用于团队的自动化和编排，而不要求他们将自动化作为自定义方法来编写。换句话说，自动化可轻松构建和运行易于管理的应用。	**手动扩展。** 手动基础架构包括人工运营人员，他们负责手动构建和管理服务器、网络及存储配置。由于复杂程度较高，运营人员无法快速地大规模正确诊断问题，并且很容易执行错误实施。手动构建的自动化方法可能会将人为错误的硬编码到基础架构中。
快速恢复。 容器运行时和编排程序可在虚拟机上提供动态的高密度虚拟化覆盖，与托管微服务非常匹配。编排可动态管理容器在虚拟机群集间的放置，以便在发生故障时提供弹性扩展和恢复/重新启动功能。	**恢复缓慢。** 基于虚拟机的基础架构对于基于微服务的应用来说是一个缓慢而低效的基础，因为单个虚拟机启动或关闭的速度很慢，甚至在向其部署应用代码之前就存在很大的开销。

云原生核心技术

- 容器
- **k8s**
- 微服务/服务网格

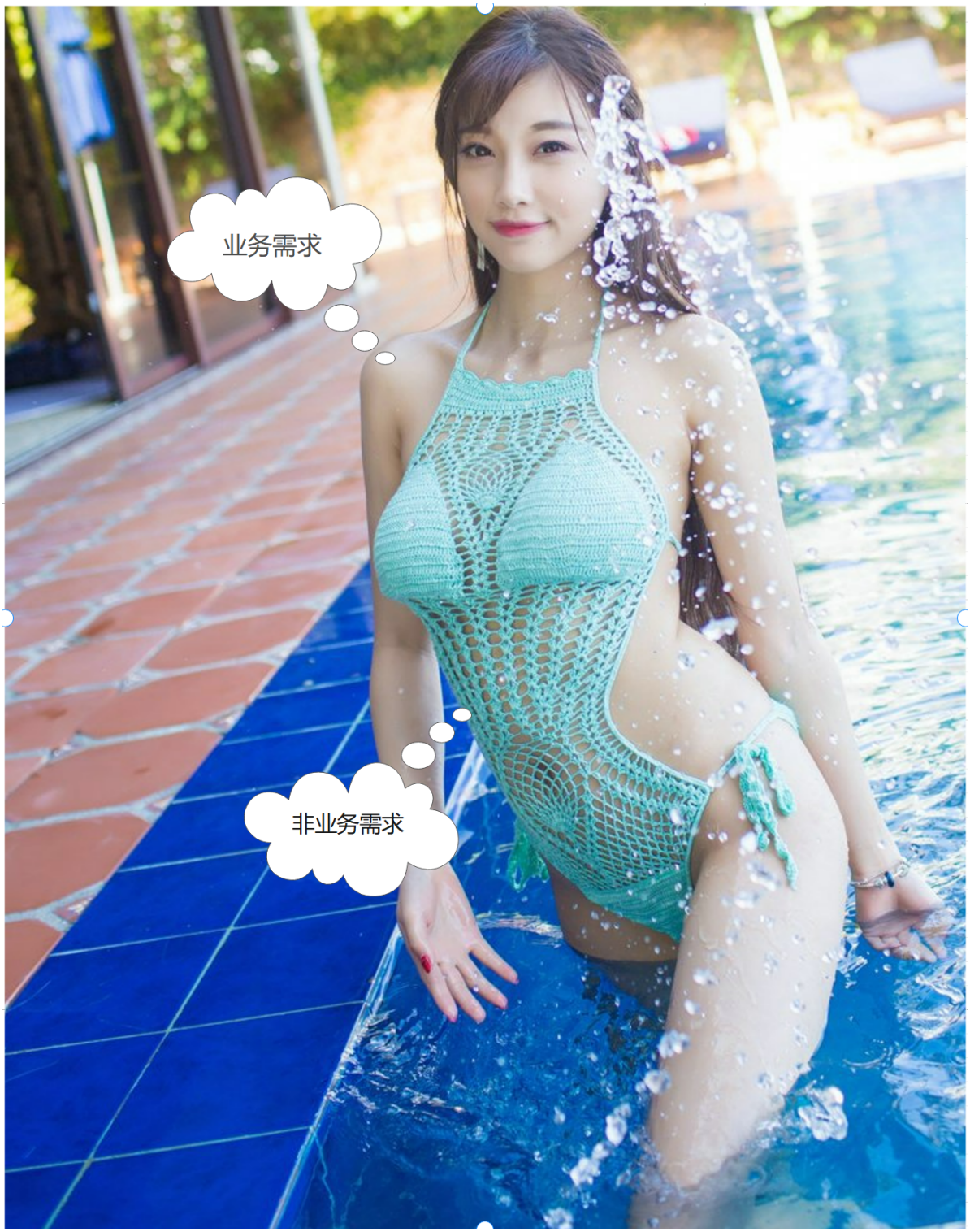
如何让自己的应用符合云原生

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

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码量没这么夸张）

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。

我们设想中云原生应该是这样的：



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。

我们的想法，云原生应用应该朝**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

"衣服"去哪了?

> - 云
> - 基础设施
> - 下沉到基础设施

轻量化

> 体积小，启动快、占用资源少

- java
- spring

> `GraalVM` 编译器

微服务：可扩展性、可升级、`易于维护?`、故障和资源隔离。

- 服务治理 ： 弹性收缩、故障隔离
- 流量控制 ： 路由、熔断、限流
- 应用可观测 ： 监控指标、故障追踪

spring cloud：提供了服务发现、负载均衡、失效转移、动态扩容、数据分片、调用链路监控等核心功能，一度是微服务的不二选择

问题：。。

侵入性强

k8s具备完善的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和在线扩容、可扩展的资源自动调度机制、多粒度的资源配额管理能力。还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。可以说基础设施上已经提供比较齐全的功能。

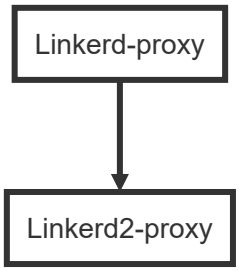
还有必要使用 spring cloud 自带的组件吗？？

spring native / spring one

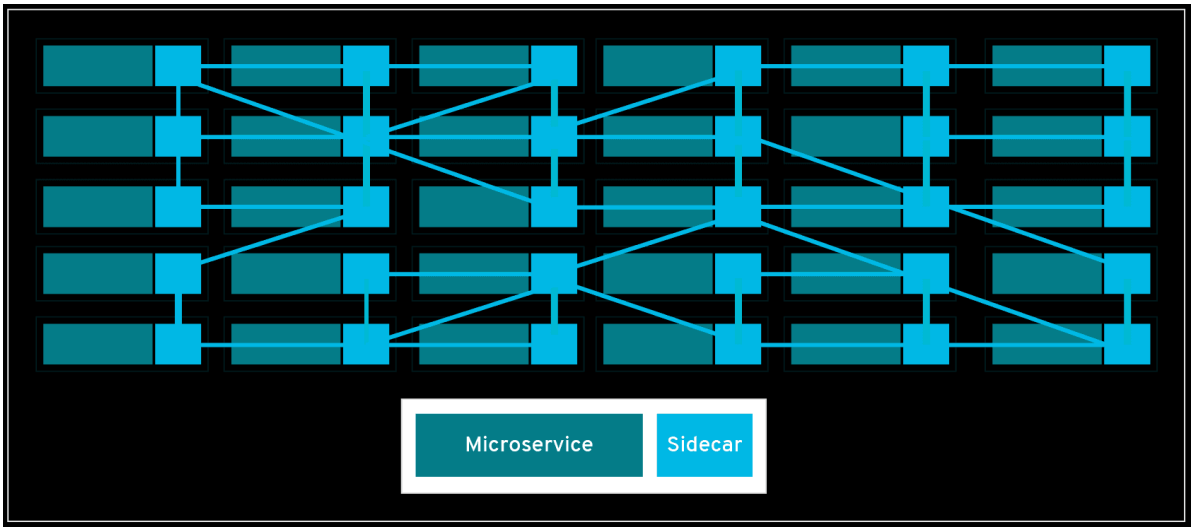
当然这里并不是说，面向服务的架构已经过时，事实上面向服务的架构非常成功，但是在云原生或 `Serverless` 环境下，可能事件驱动架构可能更合理一些，或者是两者的配合使用

> Logstash pk Filebeat

Linkerd-proxy pk envoy



Service Mesh



1. Pivotal是由 EMC 和 VMware 合资成立的软件公司，主要销售系列软件工具和咨询服务，提供PaaS解决方案。19年被VMware并购. ↩