Dapr v1.0展望：从servicemesh到云原生

前言

Dapr 是一个新兴的云原生开源项目，由微软发起，在今年2月份刚刚发布了 v1.0 正式版本。Dapr 是社区在云原生领域新的方向性的探索，项目前景非常看好。Dapr 在国外有很高的关注度，github已近12000颗星。但在国内 Dapr 知名度非常低，而且现有的少量 Dapr 资料也偏新闻资讯类和简单介绍类，缺乏对 Dapr 的深度解读。

尤其是以下问题需要回答：

- Dapr 是什么？它能解决什么问题？为什么我们需要

关注它?

- Dapr 可以做什么？它和 servicemesh 有什么关系？它和云原生又有什么关系?

在 Dapr v1.0 发布之际，我希望可以通过这篇文章回答上面的问题，期望可以让读者对 Dapr 有一个准确的认知。为了帮助大家深刻理解 Dapr 背后的产品理念和设计思路，我将从 Servicemesh 开始展开，快速回顾 servicemesh 的定义和 Sidecar 模式的特点；然后详细介绍由 sidecar 模式推广而来的 Multi-Runtime 的微服务新架构，在大家对 Multi-Runtime 的本质有深刻理解的基础之上，才正式展开 Dapr 项目的介绍。

希望可以通过这篇文章让读者掌握 Dapr 项目的发展脉搏，了解其核心价值和愿景，领悟 Dapr 项目背后的"道之所在" —— 云原生。

基于这样的考虑，这篇文章内容将分成四个部分：

1. **回顾篇**：快速回顾 Servicemesh和sidecar的原理
2. **演进篇**：介绍 sidecar 模式演进而来的 Multi-Runtime 的新架构
3. **正文**：详细介绍基于 Multi-Runtime 架构的 Dapr
4. **展望篇**：在云原生背景下结合 Dapr 项目探讨应用和中间件的未来形态

回顾：Servicemesh原理和方向

Servicemesh的定义

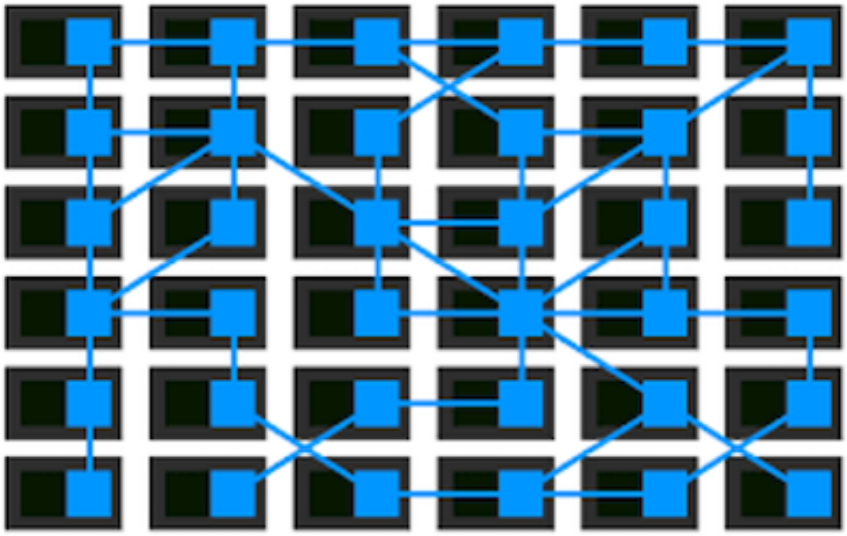
首先，让我们先快速回顾一下“Servicemesh”的定义，这是 Dapr 故事的开始。

以下内容摘录自我在2017年10月的QCon上海做的演讲 "Service Mesh：下一代微服务"：

> Service Mesh是一个基础设施层，用于处理服务间通讯。现代云原生应用有着复杂的服务拓扑，服务网格负责在这些拓扑中实现请求的可靠传递。
>
> 在实践中，服务网格通常实现为一组轻量级网络代理，它们与应用程序部署在一起，而对应用程序透明。

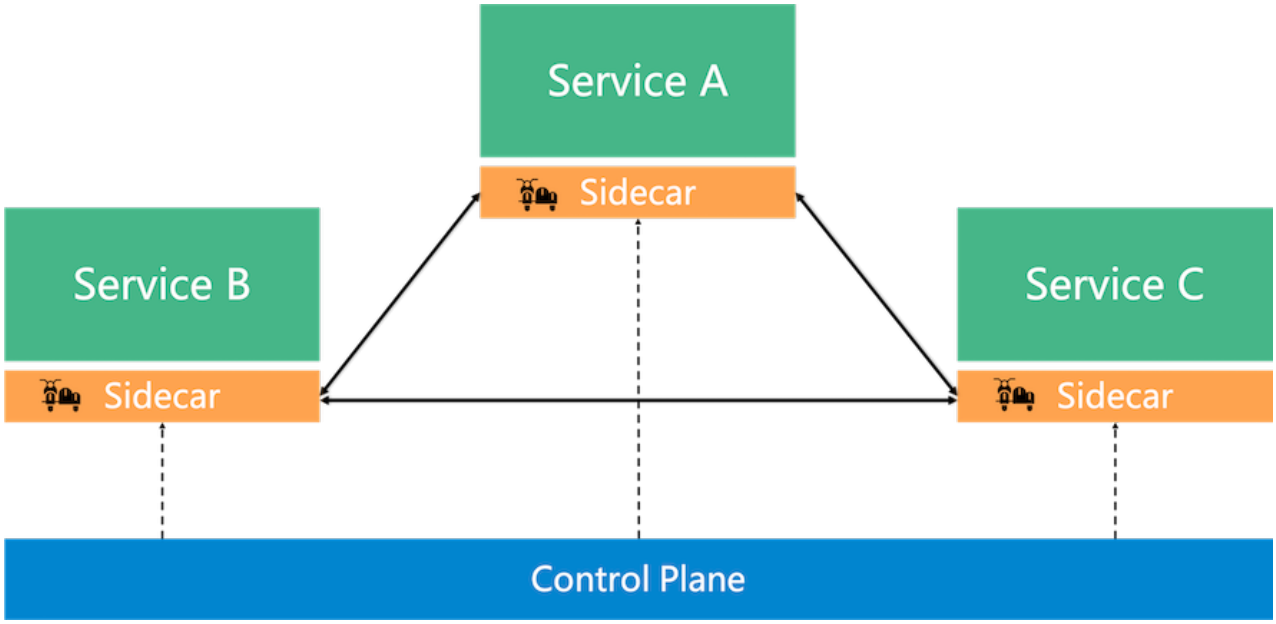
在 Servicemesh 的定义中，简短的描述了 Servicemesh 的关键特征：1. 定位基础设施层 2. 功能是服务间通讯 3. 采用 sidecar 部署 4. 特别强调无侵入、对应用透明。

熟悉 servicemesh 的同学，想必对下面这张图片不会陌生：



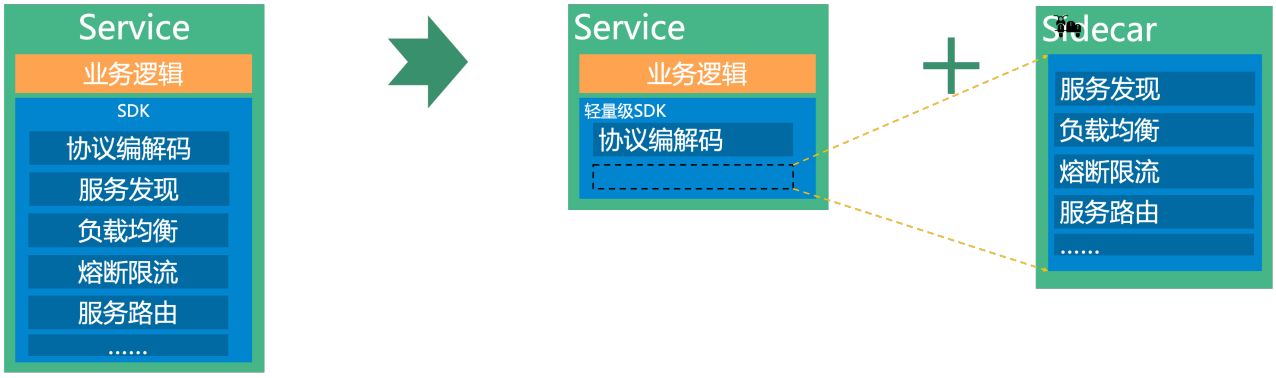
sidecar 模式

和传统RPC框架相比，servicemesh的创新之处在于引入了 sidecar 模式：



引入 sidecar 之后，服务间通讯由 sidecar 接管，而 sidecar 由控制平面统一控制，从而实现了服务间通讯能力的下沉，使得应用得以大幅简化。

我们再来快速回顾一下 Servicemesh 的基本思路：

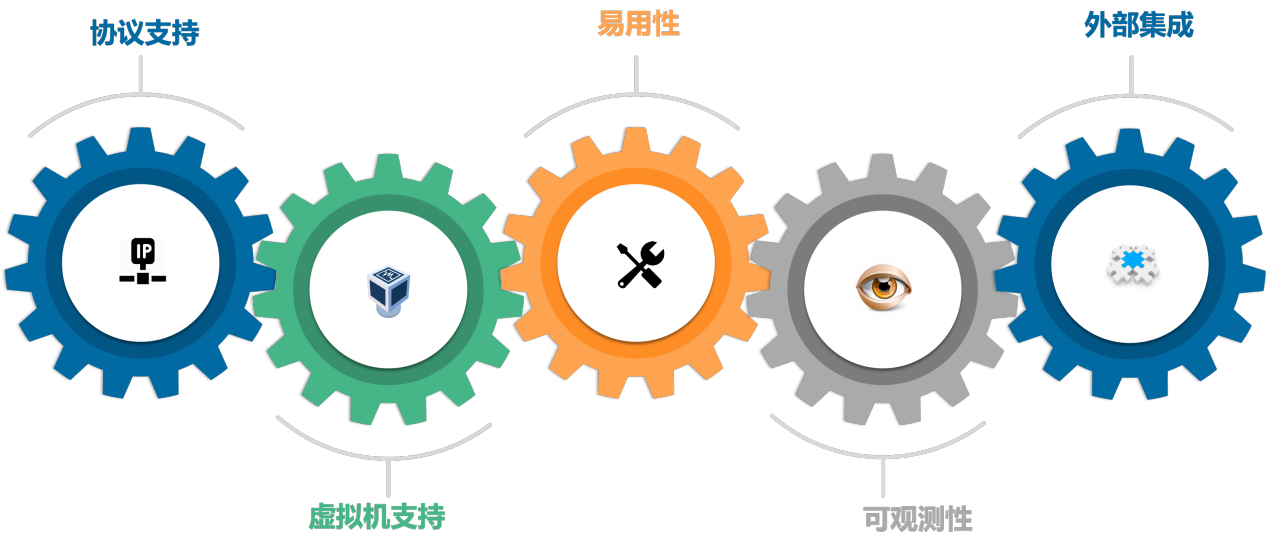


- 引入 sidecar 之前： 业务逻辑和非业务逻辑混合在一个进程内，应用既有业务逻辑，也有各种非业务的功能（体现为各种客户端SDK）。
- 引入 sidecar 之后：客户端SDK的功能剥离，业务进程专注于业务逻辑，而SDK中的大部分功能被拆解为独立进程，以Sidecar的模式运行。

通过引入 sidecar 模式，Servicemesh 成功的实现了 **关注点分离 和 独立维护** 两大目标。

servicemesh的发展趋势

以 Istio 项目为例，我这里总结了最近一两年来 servicemesh 的发展趋势（注意这些内容不是本文的重点，请快速阅读，简单了解即可）：



协议支持

Istio 中通讯协议的支持主要在 HTTP 和 gRPC，各家厂商在提供更多协议支持，包括 Dubbo、Thrift、Redis。也有一些社区力量在做补充，如赵化冰同学的 Aeraki 项目。

虚拟机支持

虚拟机的支持最近成为 Istio 的重要关注点：

- Istio 0.2：Mesh Expansion
- Istio 1.1：ServiceEntry
- Istio 1.6：WorkloadEntry
- Istio 1.8：WorkloadGroup 和智能DNS代理

- Istio 1.9：虚拟机集成

易用性

- Istio 1.5：控制平面单体化，合并多个组件为istiod（这是Istio开源以来最大的架构调整之一）
- Istio 1.7：主推 Operator安装方式，增强istioctl工具，支持在sidecar启动之后再启动应用容器
- Istio 1.8：改善升级和安装,引入istioctl bug-report

可观测性

- Istio 1.8：正式移除Mixer，在Envoy基于wasm重新实现Mixer功能 （Istio最大的架构调整之一）
- Istio 1.9：远程获取和加载wasm模块

外部集成

和非 servicemesh 体系的相互访问，实现应用在两个体系之间的平滑迁移。

- Istio曾计划通过MCP协议提供统一的解决方案
- Istio 1.7：MCP协议被废弃，改为 mcp over xds
- Istio 1.9：Kubernetes Service API支持(alpha)，对外暴露服务

从上面列出的内容，可以看到 Istio 在最近一两年间还是在非常努力的完善自身，虽然过程有些曲折和往复（比如顽固不化的坚持Mixer到最后听从全社区的呼唤彻底废弃了Mixer，开始支持虚拟机后来实质性放弃再到最近重新重视，引入Galley再废弃Galley，引入MCP再变相放弃MCP），但整体上说 Istio 还是在朝 Product Ready 的大方向在努力。

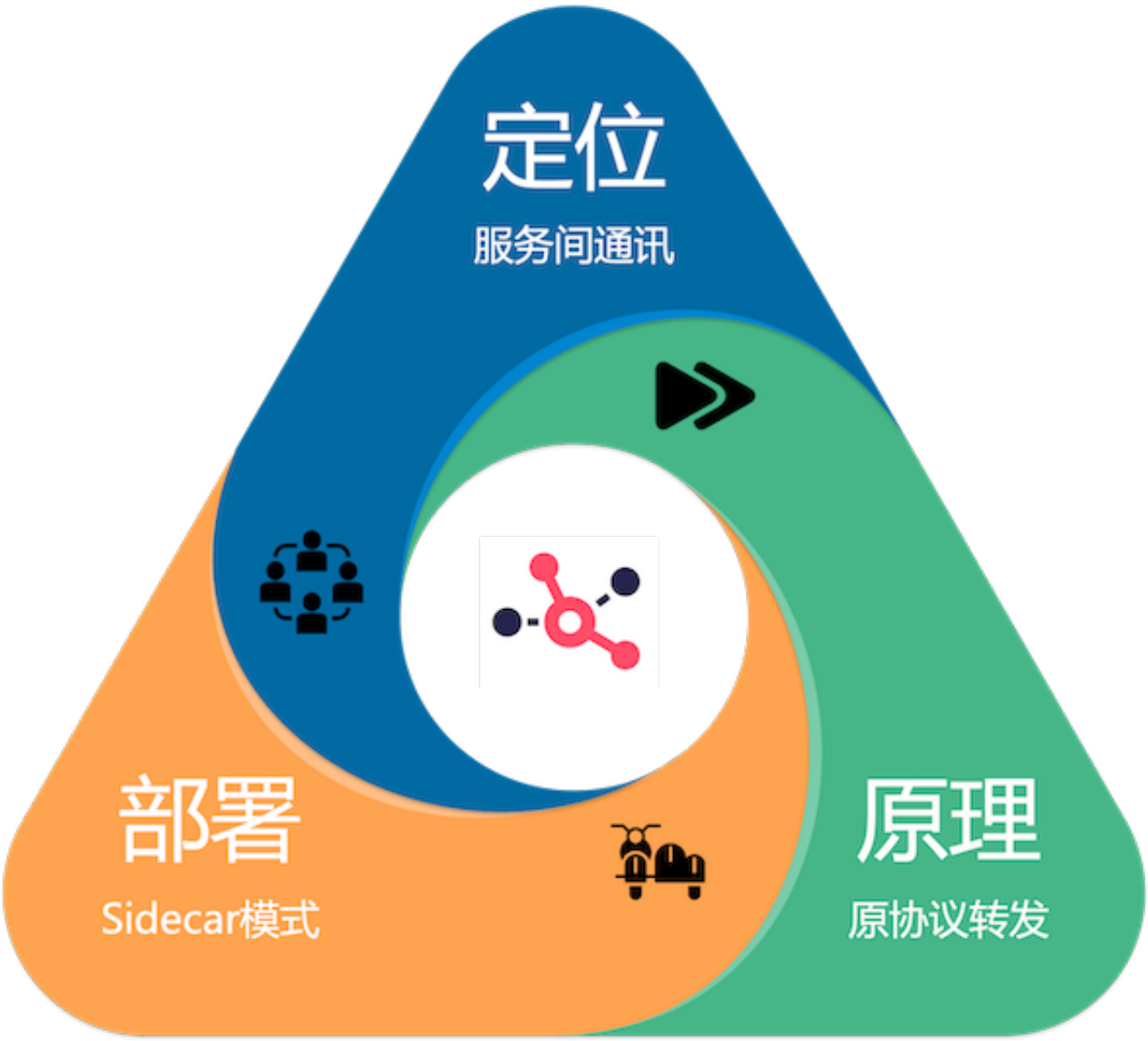
> 备注：当然，社区对 Istio 的演进速度以及 Product Ready 的实际状态还是很不满意的，以至于出现了这个梗：Make Istio Product Ready (Again, and Again...)

servicemesh 回顾总结

我们前面快速回顾了 servicemesh 的定义、sidecar 模式的原理，以及粗略罗列了一下最近一两年间 servicemesh 的发展趋势，主要是为了告知大家这样一个信息：

虽然 Servicemesh 蓬勃发展，但核心元素始终未变

从2016年 Linkerd 的 CEO William Morgon 给出 servicemesh 的定义，到2021年Istio都发布到了1.9版本，整整六年期间，servicemesh 有了很多的变化，但以下三个核心元素始终未变：



1. **定位**：Servicemesh的定位始终是提供 **服务间通讯** 的基础设施层，范围包括HTTP和RPC——支持HTTP1.1/REST，支持HTTP2/gRPC，支持TCP协议。也有一些小的尝试如对redis 、 kafka的支持。
2. **部署**：ServiceMesh 支持 Kubernetes 和虚拟机，但都是采用 **Sidecar 模式**部署，没有采用其他方式如 Node 部署、中心化部署。
3. **原理**： Servicemesh的工作原理是 **原协议转发**，原则上不改变协议内容（通常只是header有些小改动）。为了达到零侵入的目标，还引入了iptables等流量劫

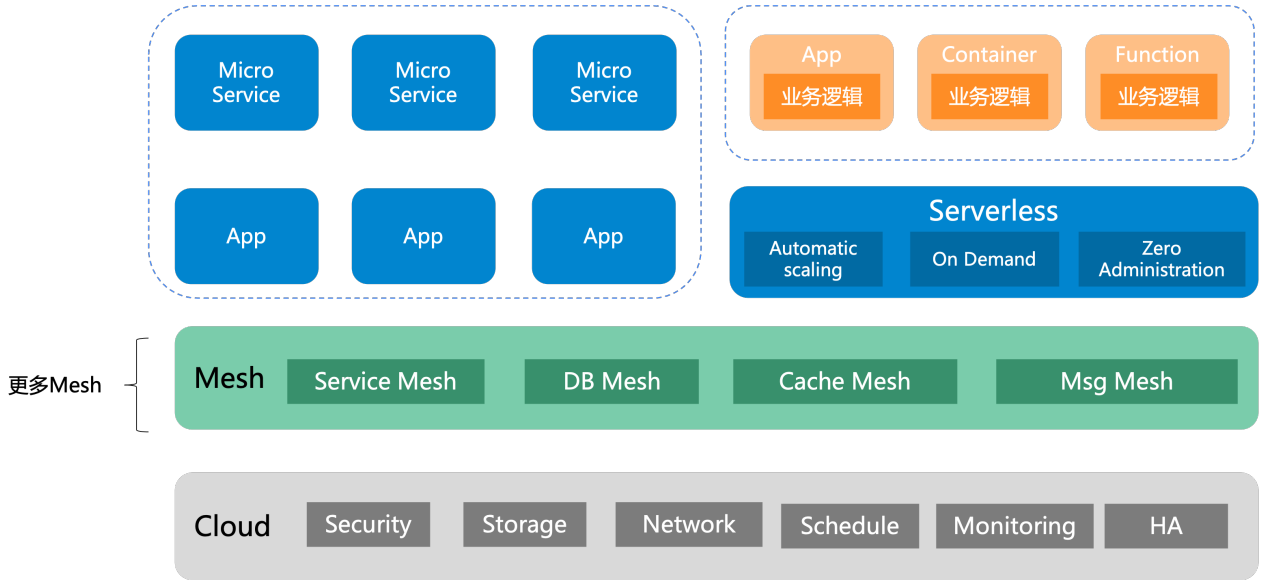
持技术。

演进：云原生分布式应用运行时

在快速完成 servicemesh 的回顾之后，我们开始本文第二部分的内容：当 sidecar 模式进一步推广，上述三个核心元素发生变化时，sidecar 模式将会如何演进?

实践：更多Mesh形态

我之前在蚂蚁金服的中间件团队做 servicemesh 相关的内容，可能很多朋友是从那个时候开始认识我。当时蚂蚁不仅仅做了 servicemesh，还将 servicemesh 的 sidecar 模式推广到其他的中间件领域，陆陆续续探索了更多的 mesh 形态：



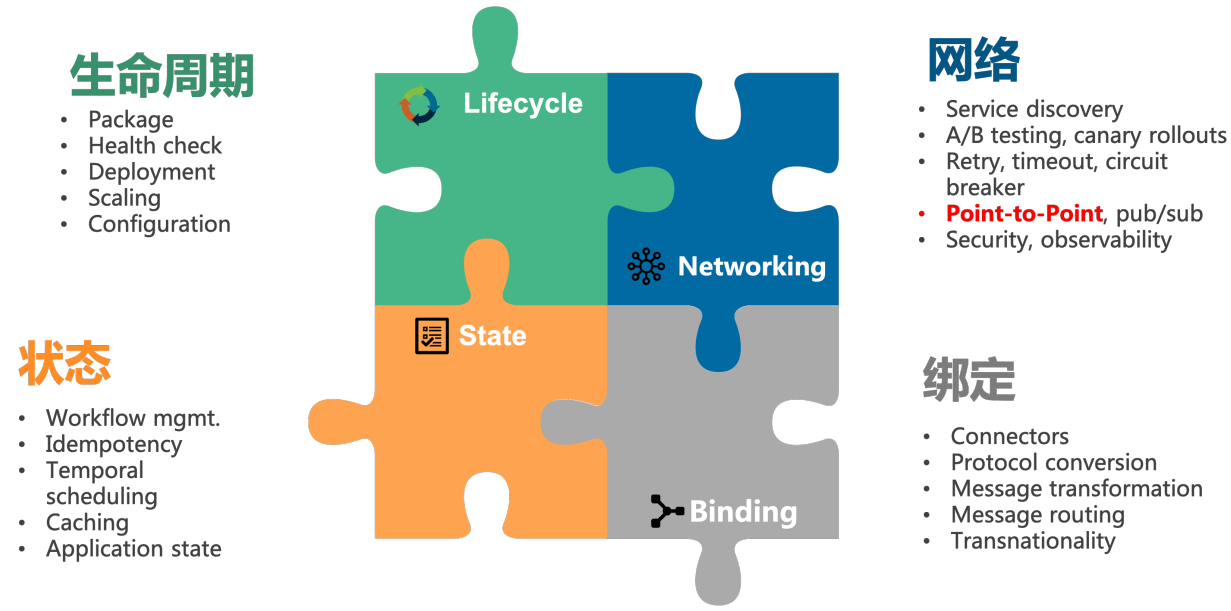
这个图片摘录自我在2019年10月的上海QCon上做的主题演讲 "诗和远方：蚂蚁金服Service Mesh深度实践"，当时我们分享了包括消息Mesh，数据库Mesh等在内的多种mesh形态。

理论升华：Multi-Runtime 理念的提出

最近有越来越多的项目开始引入 sidecar 模式， sidecar 模式也逐渐被大家认可和接受。就在2020年，**Bilgin Ibryam** 提出了 **Multi-Runtime** 的理念，对基于 sidecar 模式的各种产品形态进行了实践总结和理论升华。

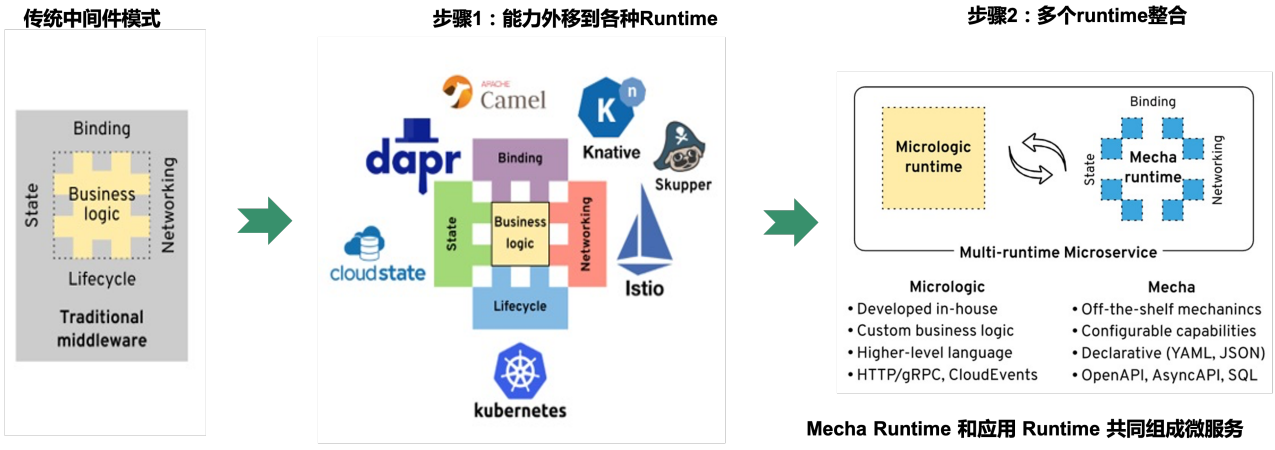
首先我们介绍一下 **Bilgin Ibryam** 同学，他是 "Kubernetes Patterns" 一书的作者，Apache Camel 项目的 committer，目前工作于 redhat 。

2020年初，Bilgin Ibryam 发表文章 "Multi-Runtime Microservices Architecture" ，正式提出了多运行时微服务架构（别名Mecha/机甲，非常帅气的名字）。在这篇文章中，Bilgin Ibryam 首先总结了分布式应用存在的四大类需求，作为 Multi-Runtime 的理论出发点：



这四大类需求中，生命周期管理类的需求主要是通过 PaaS 平台如 kubernetes 来满足，而 servicemesh 提供的主要是网络中的点对点通讯，对于其他通讯模式典型如 pub-sub 的消息通讯模式并没有覆盖到，此外状态类和绑定类的需求大多都和 servicemesh 关系不大。

Multi-Runtime 的理论推导大体是这样的——基于上述四大类需求，如果效仿Servicemesh，从传统中间件模式开始，那么大体会有下面两个步骤：



Mecha Runtime 和应用 Runtime 共同组成微服务

- 步骤一：将应用需要的分布式能力外移到各种

runtime，此时会出现数量众多的各种 sidecar 或者 proxy，如上面中列出来的 istio、knative、cloudstate、camel、dapr等。
- 步骤二：这些 runtime 会逐渐整合，只保留少量甚至只有一两个的 runtime。这种提供多种分布式能力的 runtime 也被称为 Mecha。

步骤二完成后，每个微服务就会由至少一个 Mecha Runtime 和应用 Runtime 共同组成，也就是每个微服务都会有多个（至少两个）runtime，这也就是 Multi-Runtime / Mecha 名字的由来。

Multi-Runtime和云原生分布式应用

将Multi-Runtime / Mecha 的理念引入到云原生分布式应用的方式：

- **能力**：Mecha 是通用的，高度可配置的，可重用的组件，提供分布式原语作为现成的能力。
- **部署**：Mecha 可以与单个Micrologic组件一起部署(Sidecar模式)，也可以部署为多个共享(如Node模式)。
- **协议**：Mecha不对 Micrologic 运行时做任何假设。它与使用开放协议和格式（如HTTP/gRPC，JSON，Protobuf，CloudEvents）的多语言微服务甚至单体一起使用。
- **配置**：Mecha以简单的文本格式（例如YAML，JSON）声明式地配置，指示要启用的功能以及如何将

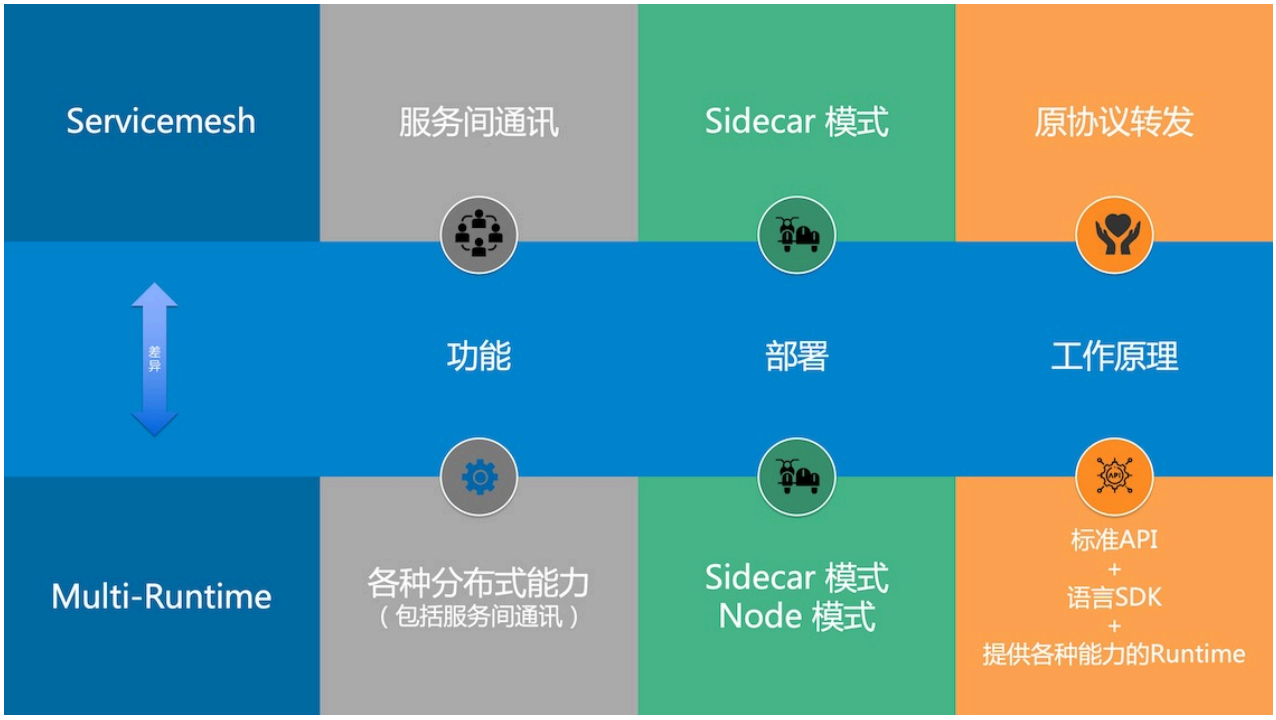
其绑定到Micrologic端点。
- **整合**：与其依靠多个代理来实现不同的目的（例如网络代理，缓存代理，绑定代理），不如使用一个Mecha提供所有这些能力。

Multi-Runtime的特点和差异

虽然同为 sidecar 模式，但是和 servicemesh 相比，Multi-Runtime 有自身的特点：

- **提供能力的方式和范围**： Multi-Runtime提供的是分布式能力，体现为应用需要的各种分布式原语，并不局限于单纯的服务间点对点通讯的网络代理
- **Runtime**部署的方式**： Multi-Runtime的部署模型，不局限于Sidecar模式，Node模式在某些场景下（如Edge/IoT，Serverless FaaS）可能会是更好的选择。
- **和App的交互方式**： Multi-Runtime 和应用之间的交互是开放而有 API 标准的，Runtime 和 Micrologic 之间的“协议”体现在API上，而不是原生的TCP通讯协议。另外Multi-Runtime 不要求无侵入，还会提供各种语言的SDK以简化开发。

Multi-Runtime 和 servicemesh 的差异总结如下图所示：



Multi-Runtime 的本质

至此我介绍了 Multi-Runtime 架构的由来，相信读者对 Multi-Runtime 的特点以及和 servicemesh 的差异已经有所了解。为了加深大家的理解，我来进一步分享一下我个人对 Multi-Runtime 的感悟：

Multi-Runtime的本质是面向云原生应用的 *分布式能力抽象层*

何为 "**分布式能力抽象层**"？

如上图所示，左侧是分布式应用存在的四大类需求：生命周期、网络、状态、绑定。从需求上说 Multi-Runtime 要为分布式应用提供这四大类需求下所列出的各种具体的分布式能力。以 Sidecar 模式为应用提供这些能力容易理解，但关键在于 Multi-Runtime 提供这些能力的方式。和 servicemesh 采用原协议转发不同，Multi-Runtime 的方式是：

- 将能力抽象为API：很多分布式能力没有类似 HTTP 这种业界通用的协议，因此 Multi-Runtime 的实现方式是将这些能力抽象为和通讯协议无关的API，只用于描述应用对分布式能力的需求和意图，尽量避免和某个实现绑定。
- 为每种能力提供多种实现：Multi-Runtime 中的能力一般都提供有多种实现，包括开源产品和公有云商业产品。
- 开发时：这里我们引入一个"面对能力编程"的概念，类似于编程语言中的"不要面对实现编程，要面向接口编程"。Multi-Runtime 中提倡面向"能力(Capability)"编程，即应用开发者面向的应该是已经抽象好的分布式能力原语，而不是底层提供这些能力的具体实现。
- 运行时：通过配置在运行时选择具体实现，不影响抽

象层API的定义，也不影响遵循"面对能力编程"原则而开发完成的应用。

> 备注：分布式能力的通用标准API，将会是Multi-Runtime成败的关键，Dapr的API在设计和实践中也遇到很大的挑战。关于这个话题，我稍后将单独写文章来阐述和分析。

介绍：分布式应用运行时Dapr

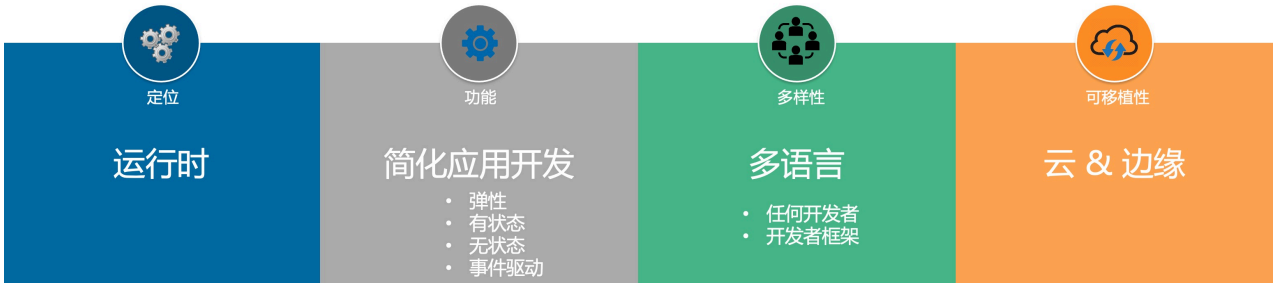
在快速回顾 servicemesh 和详细介绍 multi-runtime 架构之后，我们已经为了解 Dapr 奠定了良好的基础。现在终于可以开始本文的正式内容，让我们一起来了解 Dapr 项目。

什么是Dapr?

Dapr 是一个开源项目，由微软发起，下面是来自 Dapr 官方网站的权威介绍：

> Dapr is a portable, event-driven runtime that makes it easy for any developer to build resilient, stateless and stateful applications that run on the cloud and edge and embraces the diversity of languages and developer frameworks.
>
> Dapr是一个可移植的、事件驱动的运行时，它使任何开发者都能轻松地构建运行在云和边缘的弹性、无状态和有状态的应用程序，并拥抱语言和开发者框架的多样性。

参考并对照 servicemesh 的定义，我们对上述 Dapr 定义的分析如下：



- **定位**：Dapr 将自身定义为运行时（runtime），而不是 servicmesh 中的 proxy。
- **功能**：Dapr为应用提供各种分布式能力，以简化应用的开发。上面定义中提及的关键点有弹性、支持有状态和无状态、事件驱动。
- **多语言**：对多语言的支持是 sidecar 模型的天然优势，Dapr 也不例外，考虑到 Dapr 为应用提交的分布式能力的数量，这可能比 servicemesh 只提供服务间

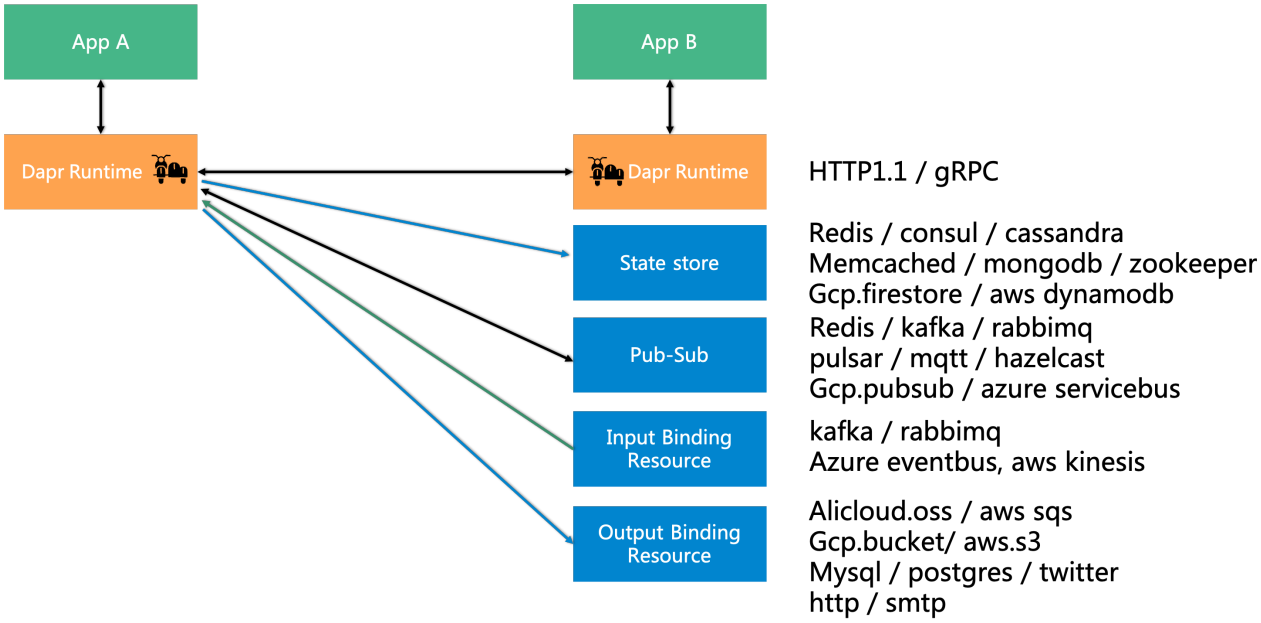
通讯能力对应用的价值更高。而且由于 Dapr 语言 SDK 的存在，Dapr 可以非常方便的和各编程语言的主流开发框架集成，如 Java 下和 Spring 框架集成。

- **可移植性**：Dapr 适用的场景包括各种云（公有云，私有云，混合云）和边缘网络，Multi-Runtime 架构的几个关键特性如"面向能力编程"、标准API、可运行时配置实现等为 Dapr 带来了绝佳的跨云跨平台的可移植性。

我们将在后面的介绍中详细展开 Dapr 的这些特性。在开始之前，这里有一个小小的花絮—— "Dapr" 项目名字的由来：

Dapr Sidecar 的功能和架构

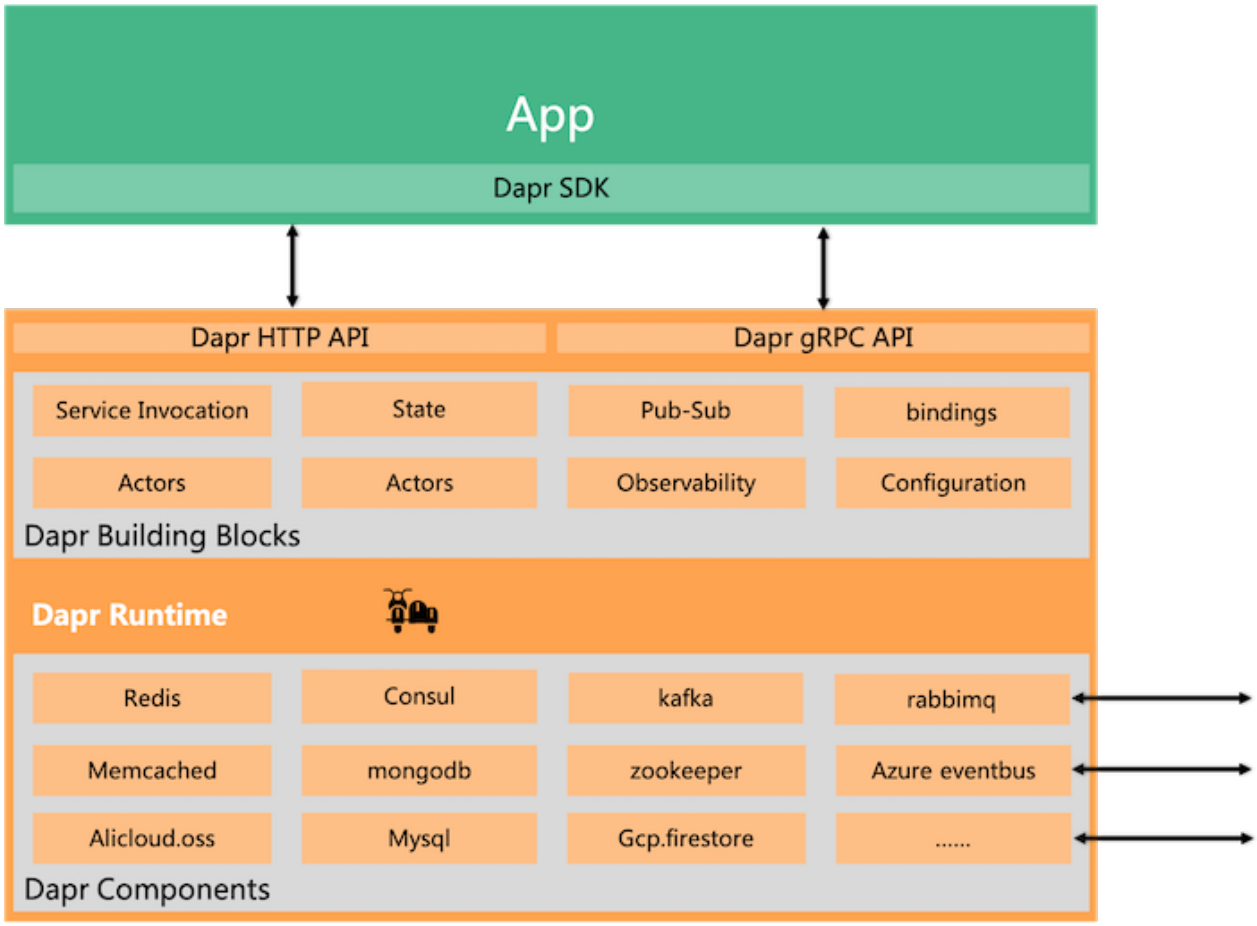
和 servicemesh 类似，Dapr 同样基于 Sidecar 模式，但提供的功能和使用场景要比 ServiceMesh 的复杂多，如下图所示：



Dapr 的 sidecar，除了可以和 servicemesh 一样支持服务间通讯（目前支持 HTTP1.1/REST协议和 gRPC 协议外，还可以支持到更多的功能，如 state（状态管理）、pub-sub（消息通讯），resource binding（资源绑定，包括输入和输出）。

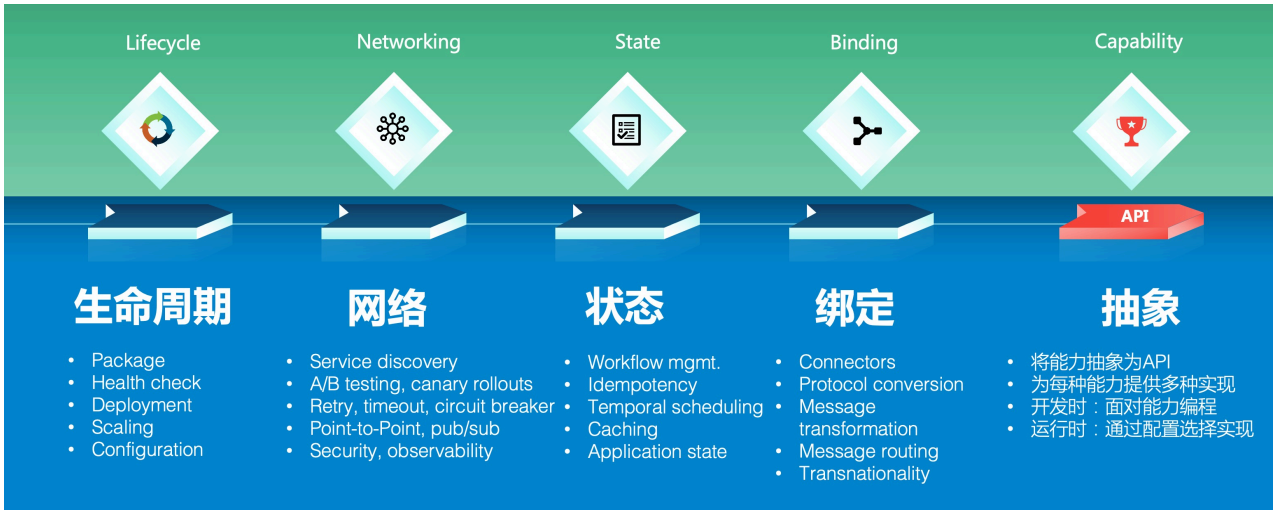
每个功能都有多种实现，在上图中我简单摘录了这几个能力的常见实现，可以看到实现中既有开源产品，也有公有云的商业产品。注意这只是目前 Dapr 实现中的一小部分，目前各种实现（在Dapr中被称为组件，我们下面会介绍）已经有超过70个，而且还在不断的增加中。

在 Dapr 的架构中，有三个主要组成部分：API，Building Blocks 和Components，如下图所示：



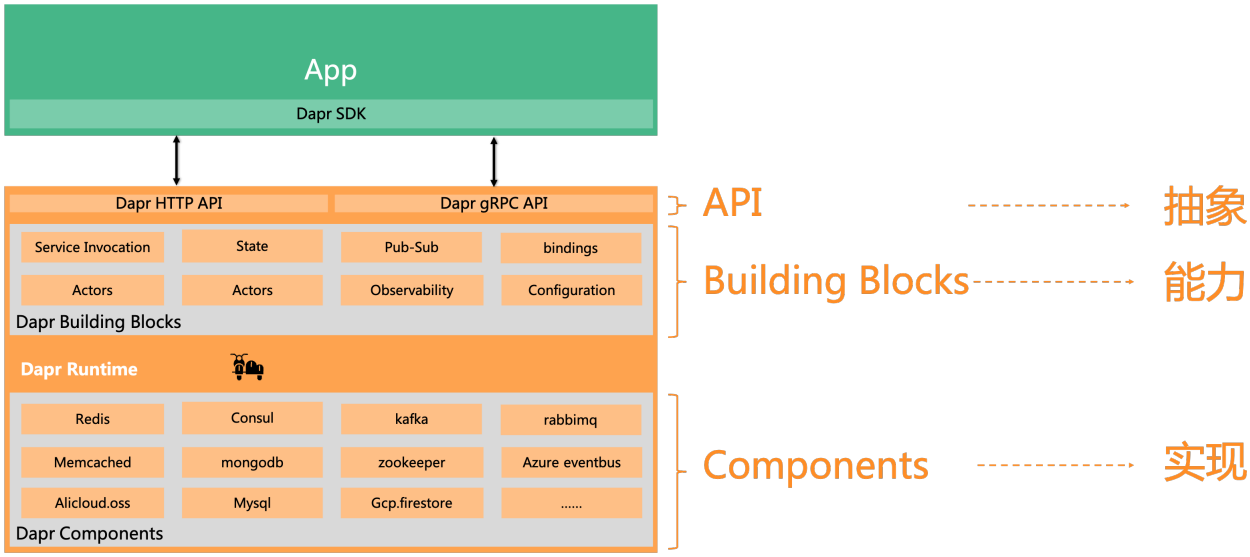
- Dapr API：Dapr 提供两种API，HTTP1.1/REST 和 HTTP2/gRPC，两者在功能上是对等的。
- Dapr Building Blocks：翻译为构建块，这是 Dapr 对外提供能力的基本单元，每个构建块对外提供一种分布式能力。
- Dapr components：组件层，这是 Dapr 的能力实现层，每个组件都会实现特定构建块的能力。

为了帮助大家理解 Dapr 的架构，我们回顾一下前面重点阐述的 Multi-Runtime 的本质：



Multi-Runtime的本质是面向云原生应用的分布式能力抽象层

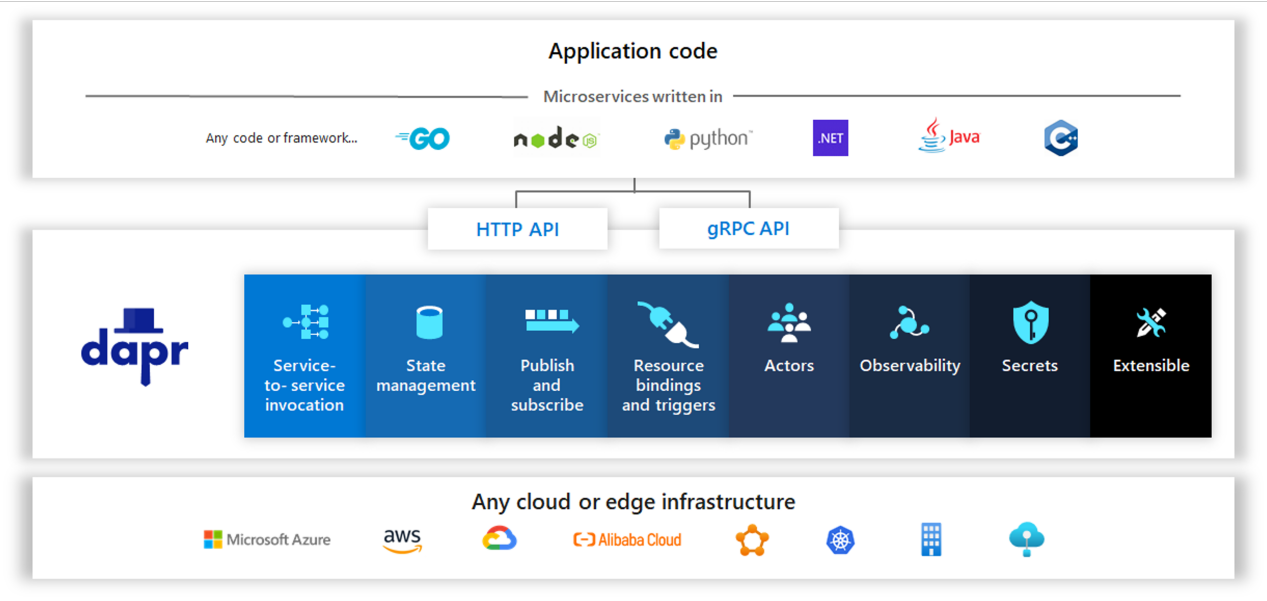
结合 Multi-Runtime 理念，我们再来理解 Dapr Runtime 的架构：



- Dapr Building Blocks 提供"**能力**"
- Dapr API 提供对分布式能力的"**抽象**"，对外暴露 Building Block 的能力
- Dapr Components 是 Building Block 能力的具体"实现"

Dapr 的愿景和现有能力

下图来自 Dapr 官方，比较完善的概括了 Dapr 的能力和层次架构：



- 居中蓝色的是 Dapr Runtime：这里列出了 Dapr 目前已经提供的构建块
- Dapr Runtime 对外通过远程调用提供能力，目前有 HTTP API 和 gRPC API
- 由于 Sidecar 模式的天然优势，Dapr 支持各种编程语言，而且 Dapr 官方为主流语言（典型如 Java、golang、c++、nodejs、.net、python）提供了 SDK。这些 SDK 封装了通过 HTTP API 或者 gRPC API 和 Dapr Runtime 进行交互的能力。
- 最下方是可以支持 Dapr 的云平台或者边缘网络，由于每个能力都可以由不同的组件来完成，因此理论上只要 Dapr 的支持做的足够完善，就可以实现在任何

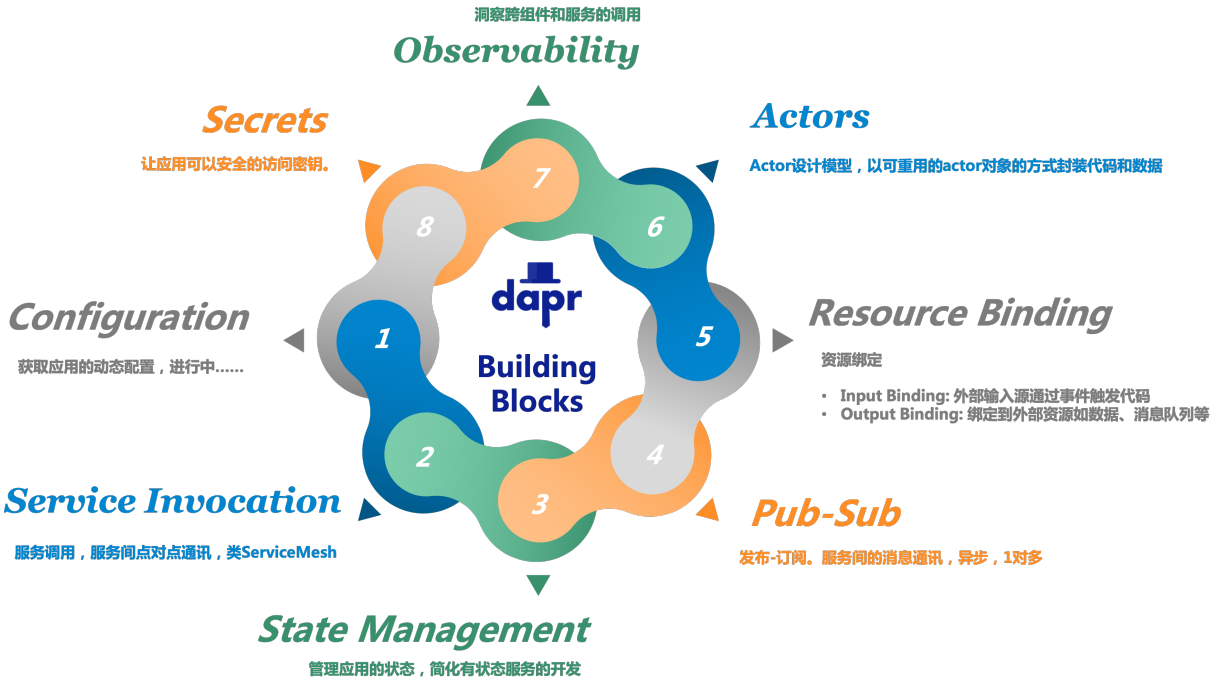
平台上，总是能找到基于开源产品或者基于云厂商商业化产品的可用组件。

结合以上几点，Dapr 提出了这样一个愿景：

Any language, any framework, anywhere

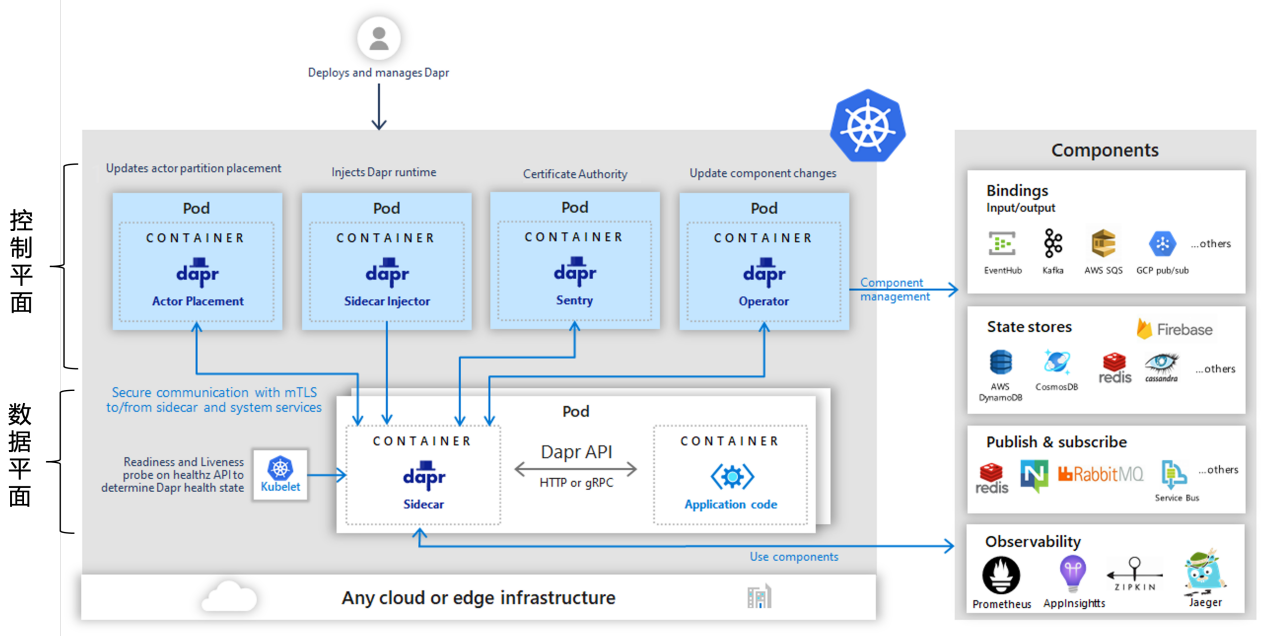
即：可以使用任意编程语言开发，可以和任意框架集成，可以部署在任意平台。

下图是 Dapr 目前已有的构建块和他们提供的能力的简单描述：



Dapr 的控制平面

和 servicemesh 的架构类似，Dapr 也有控制平面的概念：



Dapr 的控制平面组件有：

- Dapr Actor Placement
- Dapr Sidecar Injector
- Dapr Sentry
- Dapr Operator

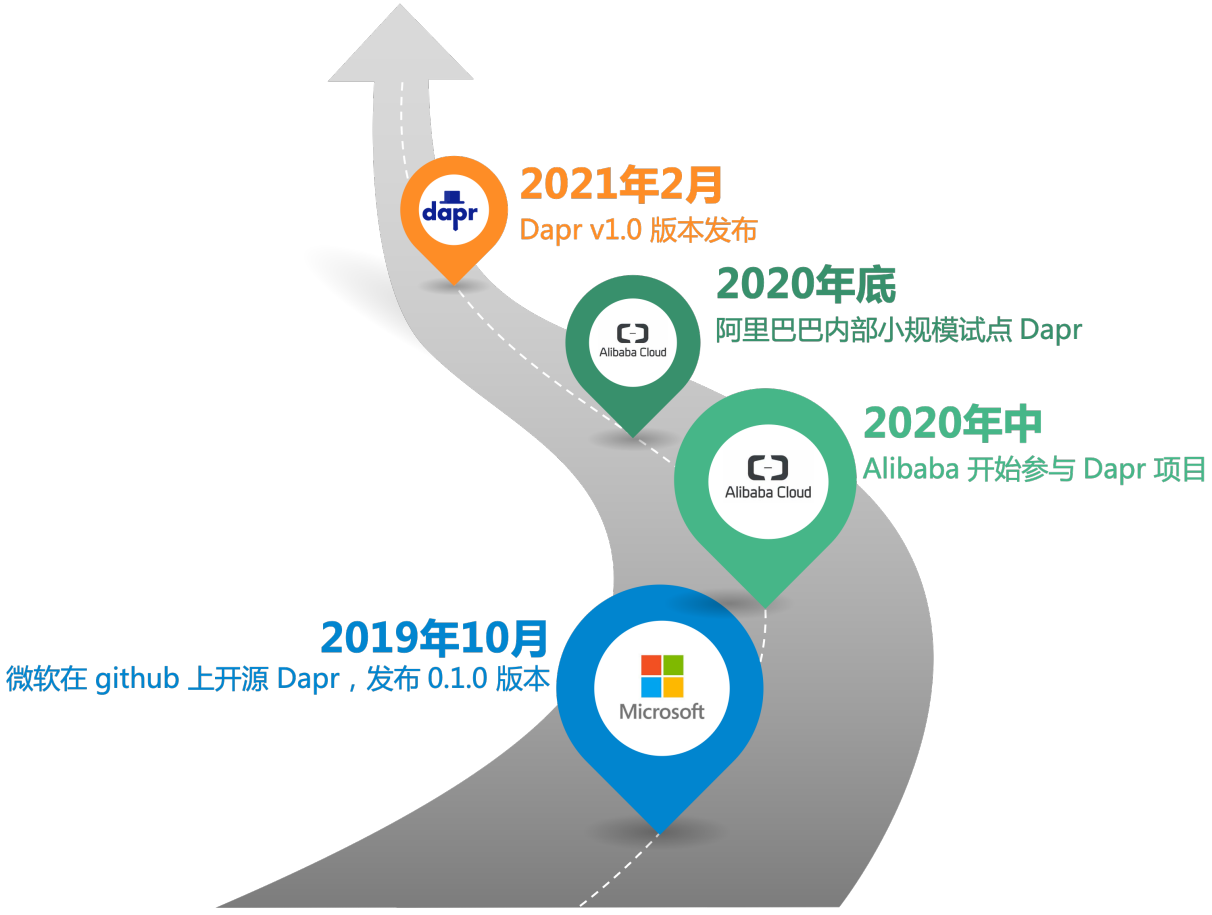
比较有意思的是：Istio 为了简化运维，已经将微服务架构的控制平面进行了合并，控制平面回归到传统的单体模式。而 Dapr 的控制平面目前还是微服务架构，不知道未来会不会效仿 Istio。

> 备注：出于控制篇幅的考虑，本文不对 Dapr 的构建块和控制平面进行详细展开，稍后预计会另有单独文章做详细介绍，对 Dapr 有兴趣的同学可以关注。

Dapr 的发展历程和阿里巴巴的参与

Dapr 是一个非常新的开源项目，发展至今也才大约一年半的时间，不过社区关注度还不错（主要是国外），在 github 上目前有接近 12000 颗星(类比：envoy 16000，istio 26000，linkerd 7000)。Dapr 项目的主要里程碑是：

- 2019年10月：微软在 github 上开源了 Dapr，发布 0.1.0 版本
- 2021年2月：Dapr v1.0 版本发布



阿里巴巴深度参与 Dapr 项目，不仅仅以终端用户的身份成为 Dapr 的早期采用者，也通过全面参与 Dapr 的开源开发和代码贡献成为目前 Dapr 项目中的主要贡献公司之一，仅次于微软：

- 2020年中：阿里巴巴开始参与 Dapr 项目，在内部试用功能并进行代码开发
- 2020年底：阿里巴巴内部小规模试点 Dapr，目前已有十几个应用在使用 Dapr 。

> 备注：关于 Dapr 在阿里巴巴的实践，请参阅我们刚刚发表在 Dapr 官方博客上的文章 "How Alibaba is using Dapr"

目前我们已经有两位 Dapr Committer 和一位 Dapr Maintainer，在2021年预计我们会在 Dapr 项目上有更多的投入，包括更多的开源代码贡献和落地实践，身体力行的推动 Dapr 项目的发展。欢迎更多的国内贡献者和国内公司一起加入到 Dapr 社区。

Dapr 快速体验

在 Dapr 的官方文档中提供了 Dapr 安装和 quickstudy 的内容，可以帮助大家快速的安装和体验 Dapr 的能力和使用方式。

为了更加快捷和方便的体验 Dapr，我们通过 **阿里云知行动手实验室** 提供了一个超级简单的 Dapr 入门教程，只要大约十分钟就可以快速体验 Dapr 的开发、部署过程：

https://start.aliyun.com/course?id=gImrX5Aj

有兴趣的同学可以实际体验一下。

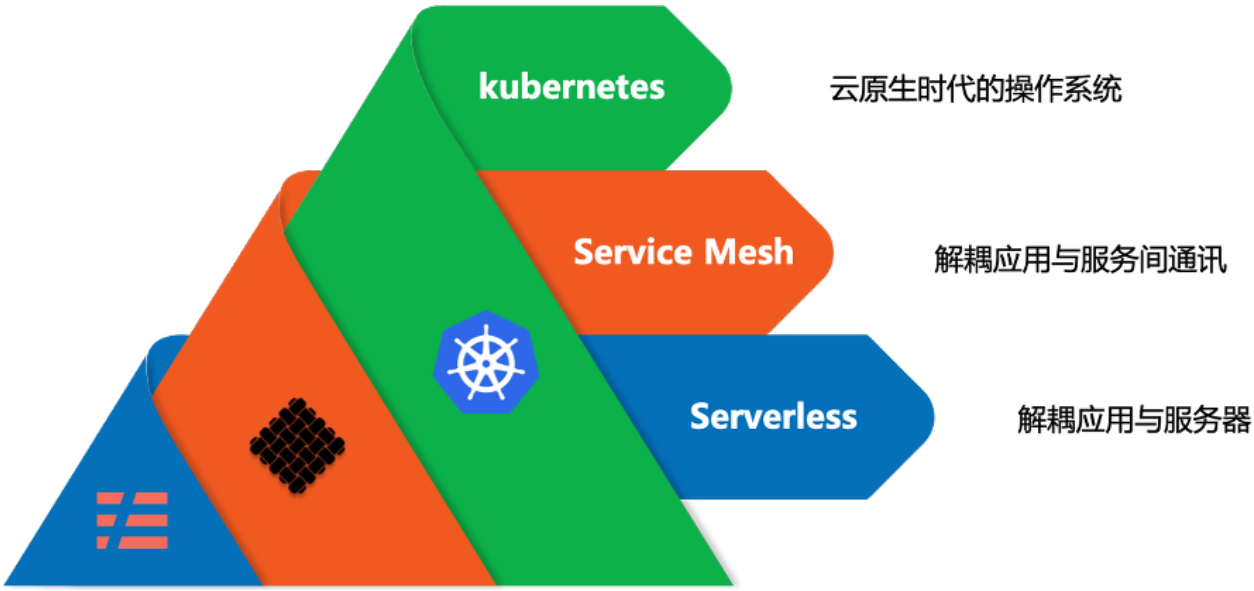
展望：应用和中间件的未来形态

在本文的最后部分，我们展望一下应用和中间的未来形态。

云原生的时代背景

首先要申明的是，我们阐述的所有这些内容，都是基于一个大的前提：云原生。

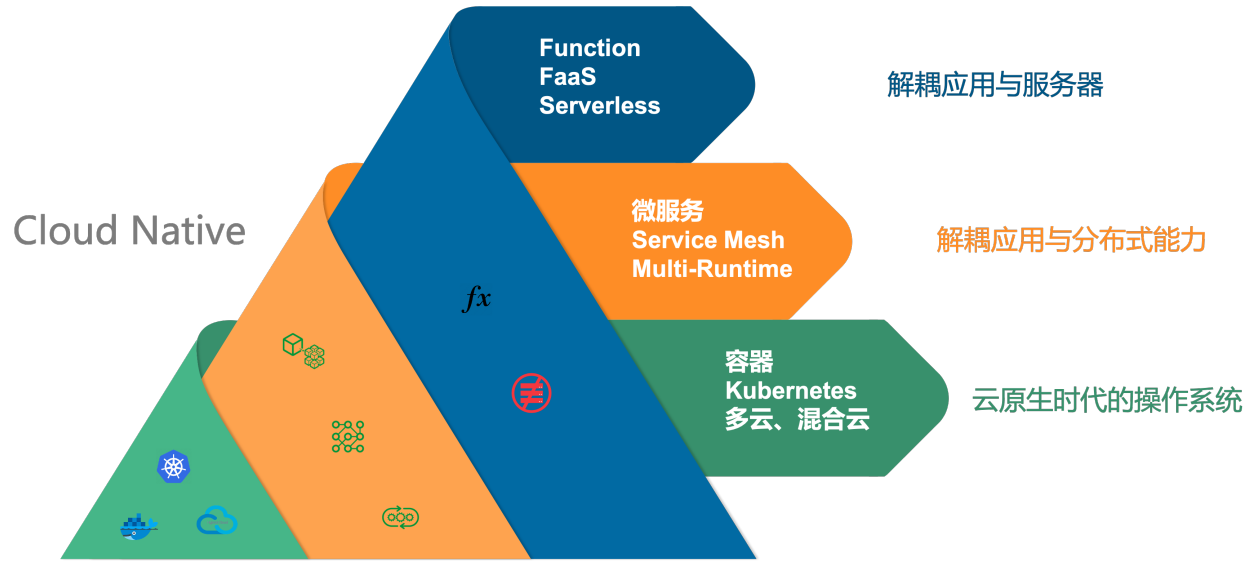
下面这张图片，摘录自我在2019年10月QCon大会上的演讲 "诗和远方：蚂蚁金服Service Mesh深度实践"：



当时（2019年）我们刚完成了 kubernetes 和 servicemesh 的探索和大规模落地，并开始 serverless 的新探索，我在文中做了一个云原生落地总结和是否采纳 servciemesh 的建议，大体可以概括为（直接援引原文）：

> - 有一点我们是非常明确的：**Mesh化是云原生落地的关键步骤**
> - 如果云原生是你的诗和远方，那么**ServiceMesh就是必由之路**。
> - **kubernetes / servicemesh / serverless** 是当下云原生落地实践的三驾马车，相辅相成，相得益彰。

两年之后的今天，回顾当时对云原生发展战略大方向的判断，感触良多。上面这张图片我稍加调整，增加了 Multi-Runtime/容器/多云/混合云的内容，修改如下图：



和2019年相比，云原生的理念得到了更广泛的认可和采纳：多云、混合云成为未来云平台的主流方向；servicemesh有了更多的落地实践，有更多的公司使用servicemesh；serverless 同样在过去两年间快速发展。

云原生的历史大潮还在进行中，而在云原生背景下，应用和中间件将何去何从？

应用的期望就是中间件的方向

让我们畅想云原生背景下处于最理想状态的业务应用，就当是个甜美的梦吧：

- 应用可以使用任意喜爱而适合的语言编写，可以快速开发和快速迭代。
- 应用需要的能力都可以通过标准的API提供，无需关心底层具体实现。

- 应用可以部署到任意的云端，不管是公有云、私有云还是混合云，没有平台和厂商限制，无需代码改造。
- 应用可以根据流量弹性伸缩，顶住波峰的压力，也能在空闲时释放资源。
- ……

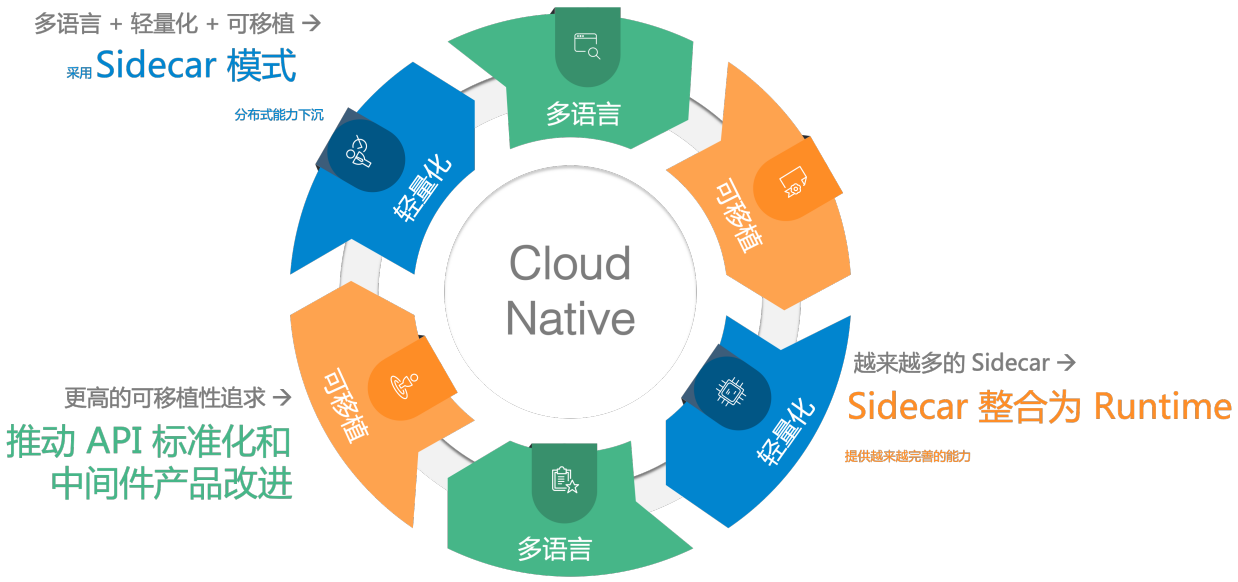
我个人的对云原生应用未来形态的看法是：Serverless 会是云上应用的理想形态和主流发展方向；而多语言支持、跨云的可移植性和应用轻量化将会是云原生应用的三个核心诉求。

应用对云原生的期望，就是中间件前进的方向！

过去几年间，中间件在云原生的美好目标推动下摸索着前进，未来几年也必将还是如此。servicemesh 探索了 sidecar 模式，Dapr 将 sidecar 模式推广到更大的领域：

- 完善的多语言支持和应用轻量化的需求推动中间件将更多的能力从应用中分离出来
- Sidecar 模式会推广到更大的领域，越来越多的中间件产品会 开始 Mesh 化，整合到 Runtime。
- 对厂商锁定的天然厌恶和规避，会加剧对可移植性的追求，从而进一步促使为下沉到 Runtime 中的分布式能力提供标准而业界通用的 API。
- API 的标准化和社区认可，将成为 Runtime 普及的最大挑战，但同时也将推动各种中间件产品改进自身实现，实现中间件产品和社区标准API之间的磨合与完善。

在云原生需求推动下，多语言支持、跨云的可移植性和应用轻量化，预计将成为未来几年间中间件产品的突破点和重点发展方向，如下图所示：



在目前的云原生领域，Dapr 项目是一个非常引人注目的新生力量。Dapr 是探路者，开启 Multi-Runtime 理念的全新探索，而这必然是一个艰难的过程。 非常期待有更多的个人和公司，和我们一起加入 Dapr 社区，一起探索，共同成长！

> 备注：关于 Dapr API 标准化的话题，以及 Dapr 在定义 API 和实现 API 遇到的挑战，在现场曾有一段热烈的讨论，我将稍后整理出单独的文章，结合 state api 的深度实践和新的 configuration api的设计过程，深入展开，敬请关注。

尾声

在这片文章的最后，让我们用这么一段话来总结全文：

Dapr 在 Servicemesh 的基础上进一步扩展 Sidecar 模式的使用场景，一方面提供天然的多语言解决方案，满足云原生下应用对分布式能力的需求，帮助应用轻量化和serverless化，另一方面提供面向应用的分布式能力抽象层和标准API，为多云、混合云部署提供绝佳的可移植性，避免厂商锁定。

Dapr将引领云原生时代应用和中间件的未来。

附录：参考资料

本文相关的参考资料如下：

- Dapr 官网 和 Dapr 官方文档：部分 Dapr 介绍内容和图片摘录自 dapr 官方网站
- Multi-Runtime Microservices Architecture: multi-runtime 介绍的内容和图片部分援引自 Bilgin Ibryam 的这篇文章