

微服务敏捷开发模式探索与实践

姜胜，华为云工具

www.huawei.com

会议议程

- **为什么要用微服务**
- **什么是微服务**
- **微服务架构介绍**
- **微服务框架实践**

未来市场对产品的需求

SHORTER DELIVERY TIMES



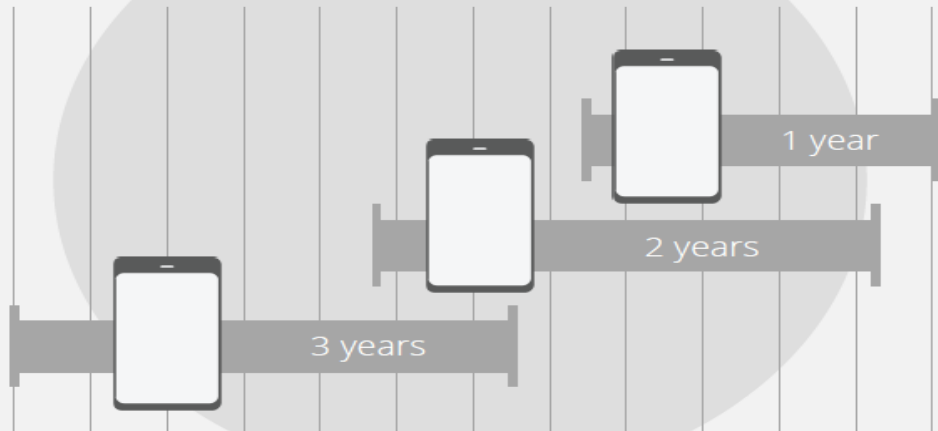
VOLATILE MARKETS



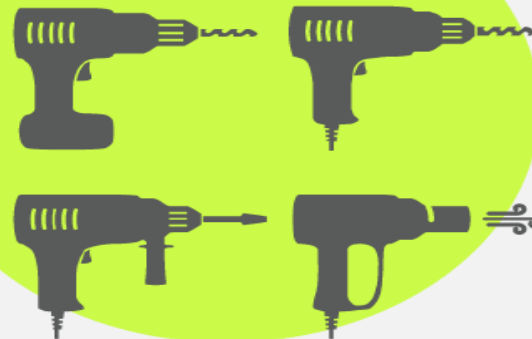
24/7 SERVICE



SHORTER PRODUCT LIFE CYCLES



MORE INDIVIDUALIZED
CUSTOMER WISHES



应用向CloudNative演进，微服务是CloudNative的事实标准

第一代：单体架构



- 紧耦合，
- 系统复杂、错综交互，动一发而牵全身
- 重复制造各种轮子：操作系统，数据库，中间件
- 完全封闭的架构

第二代：SOA架构



- 松耦合
- 在大型、超大型企业中仍然流行
- 通常通过ESB进行系统集成
- 有状态
- 大团队：100~200人
- TTM: 1年、半年、月
- 集中式、计划内停机扩容

第三代：微服务架构



- 解耦
- TTM: 按周，天进行升级发布
- 可扩展性：自动弹性伸缩
- 高可用：升级、扩容不中断业务
- DevOps: CI, CD, 全自动化
- 小团队：2 Pizza Team
- 互联网公司(亚马逊)、中小企业、初创公司

PaaS日趋成熟，微服务是大趋势，Docker等容器技术能更好支撑微服务

会议议程

- 为什么要用微服务
- 什么是微服务
- 微服务架构介绍
- 微服务框架实践

什么是微服务

微服务是一个软件架构形式。在这个架构中，复杂的应用程序是由多个小而独立的进程组成，每一个进程通过独立于语言的接口进行相互交流。这些服务较小、高度解耦且专注于完成一个小任务，使得用模块化方法建设系统更加容易。

- From Wikipedia <https://en.wikipedia.org/wiki/Microservices>

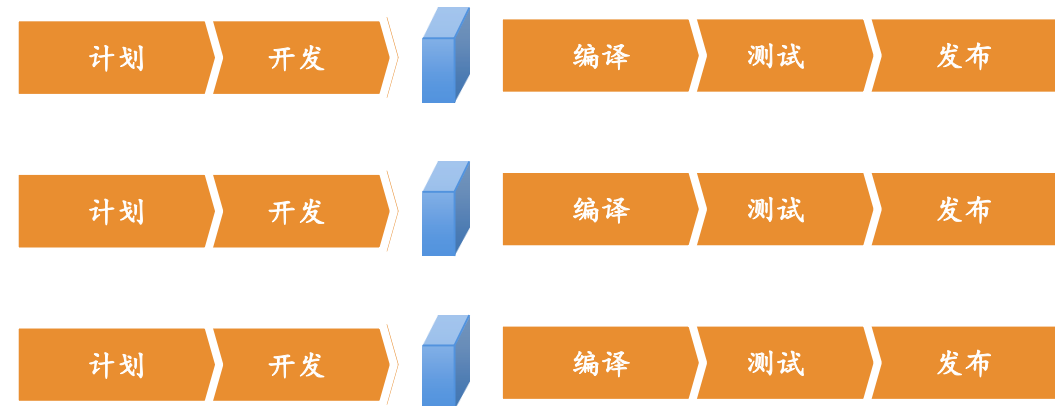
- 每个服务都很简单，只关注于一个业务功能。
- 每个微服务可以由不同的团队独立开发。
- 微服务是松散耦合的。
- 微服务可以通过不同的编程语言与工具进行开发。

开发模式从传统的单体模型转变成微服务模型

传统单体开发模型



微服务开发模型



部署间隔 (工作日)

云服务	0.35秒
业务网站	11.6秒

业界领先的工程工作量

每年8百万次编译
50,000 流水线

几千团队 × 使用微服务架构开发 × 持续交付 × 多个环境 =
每年5千万次部署

微服务架构的特性——Martin Fowler

- 通过服务来实现组件：将**应用运行在不同的进程中**，单一服务的局部变化只需重新部署对应的服务进程。**服务之间的调用是跨进程的**，设计时必须定义清晰的边界和职责。
- 按业务能力来划分服务与组织团队：康威定律（Conway 's law）指出任何设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构。**系统架构与组织沟通结构匹配。**
- 服务即产品：**让开发团队负责整个产品的全部生命周期。You build it, you run it.** 开发团队对软件在生产环境的运行负全部责任，让服务的开发者与服务的使用者（客户）形成每天的交流反馈，来自直接客户端的反馈有助于开发者提升服务的质量。
- 智能端点与傻瓜管道：服务作为智能终端，**所有的业务逻辑在服务内部处理**，而服务间的通信尽可能的轻量化，不添加任何额外的业务规则。
- 分散的治理和数据管理：可以针对不同的业务服务特征选择**不同的技术平台或产品**，有针对性的解决具体的业务问题。
- 基础设施自动化：一系列的多进程服务，意味着开发、调试、测试、集成、发布和监控的复杂度都会相应增大。必须要有合适的**自动化基础设施来支持微服务架构模式**，否则开发、运维成本将大大增加。
- 容错设计：任何时刻对服务的调用都可能因为服务方不可用导致失败，这就要求服务的消费方需要**优雅的处理此类错误。**
- 进化的设计：由于使用微服务架构，整个系统的各组成成分可以独立改进。不需要大家都准备好之后才能做改动。所以整个**系统的设计和改善是通过不断和频繁的迭代逐渐进化的。**

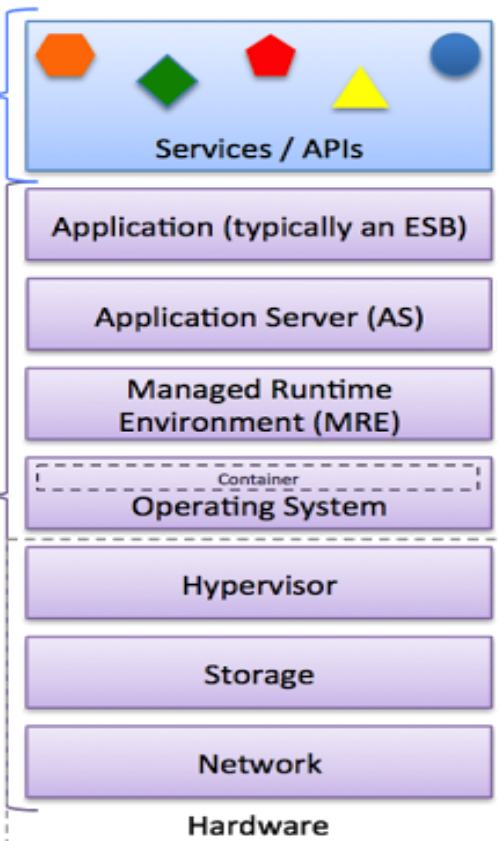
微服务的优缺点

微服务提供的好处	微服务带来的成本
模块化 ：微服务强调模块化的结构，这对于团队特别重要	分布式特性 ：分布式系统的编程难度更大，因为远程调用慢，而且总存在失败的风险。
独立部署 ：简单的服务更容易部署，而且由于它们是自治的，一旦出了问题，不会导致整个系统的故障	最终一致性 ：对于分布式系统来说，保持强一致性很难，这意味着每个人都不得不去处理最终一致性
技术的多样性 ：有了微服务，你可以混合使用多种编程语言、开发框架以及数据存储技术	运维的复杂性 ：你需要一个成熟的运维团队，来管理很多需要定时重新部署的服务

微服务和SOA有什么不同？

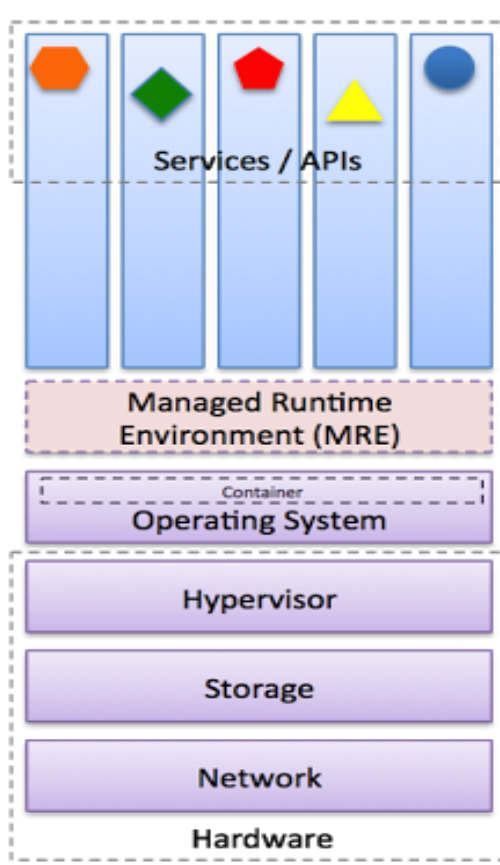
- Built on common governance and standards
- Common tech stack
- Contracts define service/APIs interfaces
- Granularity focused on business capabilities
- Typically Services/APIs run on an ESB
- Use of Canonical schemas not uncommon for business services, less common in APIs
- APIs typically targeted for external use (exposed in DMZ)
- HTTP transport of choice but multi-protocols supported (SOAP/XML, REST/JSON, etc)
- DevOps / Continuous Delivery becoming more popular not yet mainstream

Typical Systems Layers In SOA Architectures



- Common platform for all services deployed to it
- Typically services/APIs runs on an AS, that depends on an MRE
- Resources made available to and managed by MRE and AS
- Multi-threaded with more overheads to handle I/Os
- Use of containers (i.e. Docker, Linux Containers) less popular
- Common hardware for all services/APIs running on the same ESB or Application Server clusters

Typical Systems Layers In Microservices Architecture



- Relaxed governance: less focus on common standards and more on people collaboration and freedom of choice
- Use of ESBs not popular
- Granularity focused on business capabilities
- Services/APIs are self-contained and can run independently from each other
- Services/APIs built using tech stack of choice (usually one that's best for the job)
- Use of Lightweight protocols, such as HTTP/REST and AMQP
- Strong focus on DevOps / Continuous Delivery from the start
- Services are stateless

- Single-threaded typically with use of Event Loop (callbacks) features for non-locking I/O handling
- Application Servers not really used. Platforms such as Node.js can be used but not mandated (as said, no tech stack enforced)
- Use of containers (i.e. Docker, Linux Containers) more popular as services/APIs are more independent on other applications
- Common hardware optional

- 微服务实践：是从更加强调分布式和scalability的互联网场景下演化出来的更加敏捷，**去中心化的服务框架**。
- 另外 SOA本身也在演进，所以微服务和SOA之间一条清晰的界限是没有的。所以这里只比较传统（最早由IBM, BEA等公司实现的）的SOA和微服务框架之间的区别。
- 微服务关键词：高内聚低耦合，**敏捷，去中心化，自动化**。

微服务和SOA有什么不同？

层次	传统SOA	微服务实践
架构层	ESB ,BPEL, SCA等	<ul style="list-style-type: none"> • 去中心化。 • 同步调用通过服务发现机制直接调用，无中心转发，无消息格式转换。 • 异步调用通过不感知内容的消息服务集群。
协议层	Web Service, Soap, WSDL	拥抱简单透明的轻量级协议，http REST, JSON. (有些场景也可以使用XML转换格式)
应用层	<ul style="list-style-type: none"> • OLTP中间件。 • Weblogic, Websphere 	<ul style="list-style-type: none"> • 弱化作为中间层的传统中间件的作用，一切皆服务，由独立的服务提供实现。 • 无语言和框架绑定，倾向于更小，启动更快的语言或者框架，golang，node.js... • 如果使用java语言倾向于jetty，spring这些轻量级的技术。
数据层	<ul style="list-style-type: none"> • 全局事务。 • 商业关系新数据库如Oracle。 	<ul style="list-style-type: none"> • 一般无全局事务和强一致性要求。 • 无状态 • 数据存储的多样性和分布性，MySql, NoSql.
适用场景	<ul style="list-style-type: none"> • 数据有强一致性要求。 • 集成多供应商、异构协议的的商业应用。 	<ul style="list-style-type: none"> • 数据最终一致性场景。 • 基于开源的产品或者构架。 • Scalability很重要。

传统SOA重型框架，各个模块紧耦合，升级周期长、成本高，不能满足产品需求快速变化的场景。

微服务实施要点

1. 自动化文化与环境：**自动构建、自动测试、自动部署。**
2. 围绕业务能力建模服务：**松耦合、高内聚、暴露接口**而隐藏实现细节。
3. 服务协作模型：**中心化**（乐队模型：中心指挥）和**去中心化**（舞蹈模型：群舞自组织），各自场景不同。
4. 服务交互方式：RPC/REST/WS 技术很多但考虑统一。
5. 服务部署：**独立性、失败隔离性、可监控性。**
6. 服务流控：**降级、限流**
7. 服务恢复：多考虑故障发生如何**快速恢复**而非如何避免发生故障。
8. 服务发布：**灰度。**
9. 服务部署：**一服务一主机模型**，需要虚拟化(Hypervisor)、容器化(LXC, Docker)等技术支持，实现硬件资源隔离。
10. 服务配置：**中心化配置**服务支持
11. 康威定律：任何设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构。**系统架构的设计符合组织沟通结构取得的收益最大。**

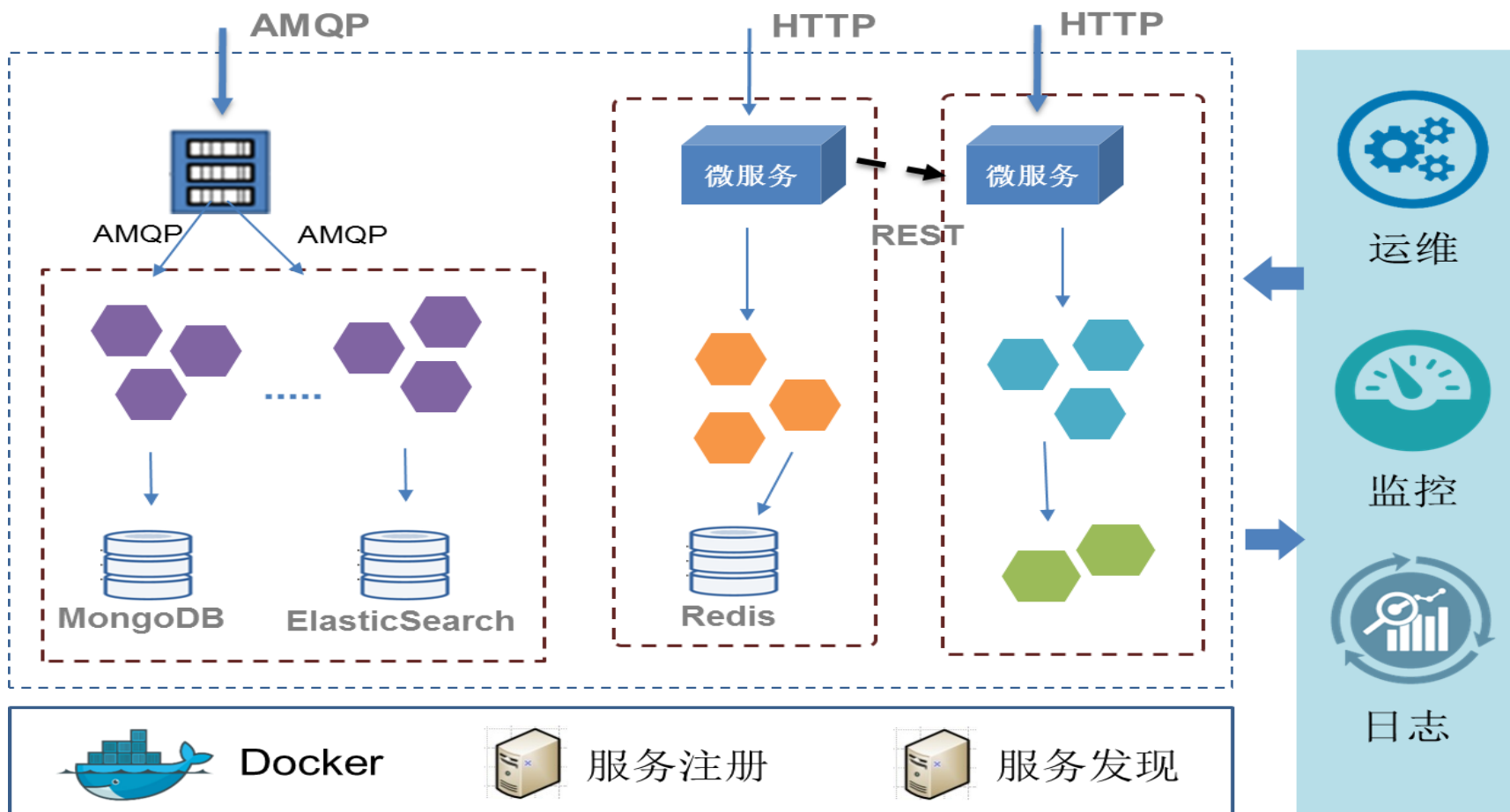
目 录

- 为什么要用微服务
- 什么是微服务
- **微服务架构介绍**
- 微服务框架实践

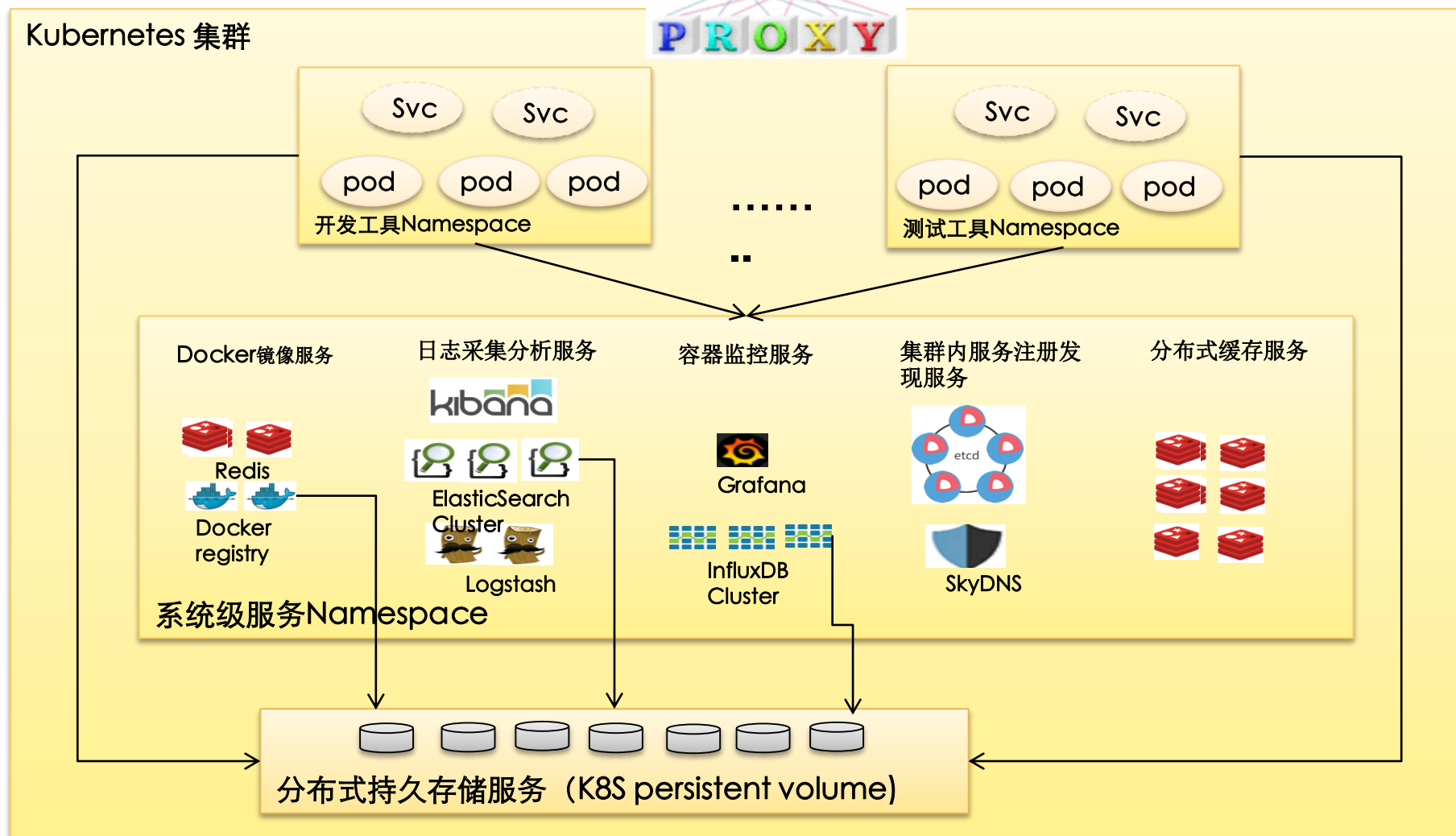
一个典型的微服务逻辑架构



一个典型的基于微服务的应用架构

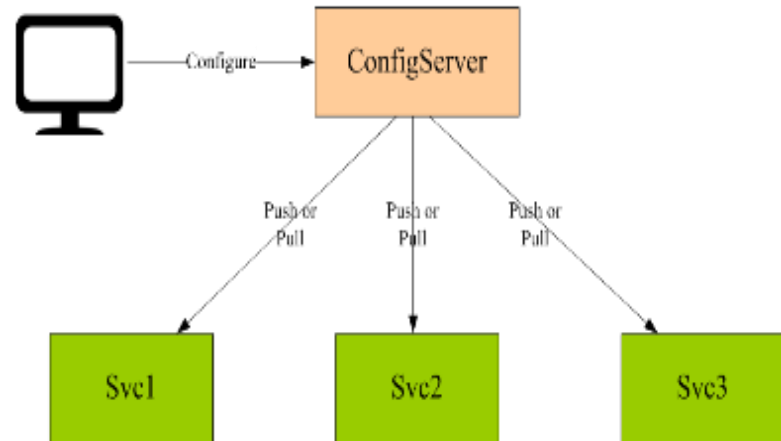
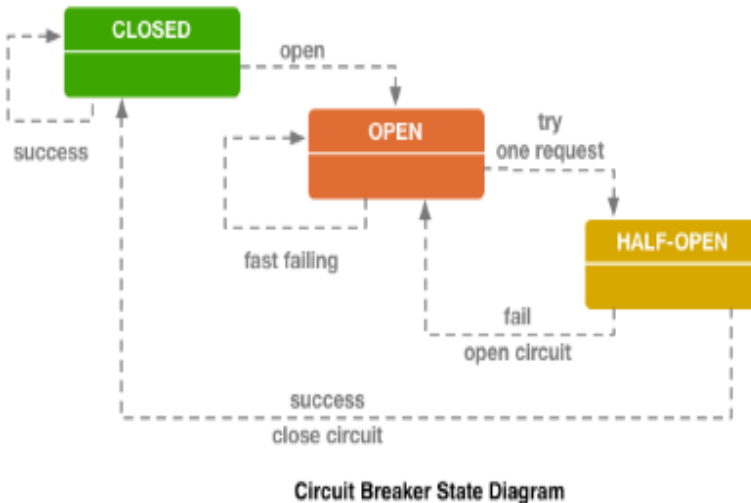
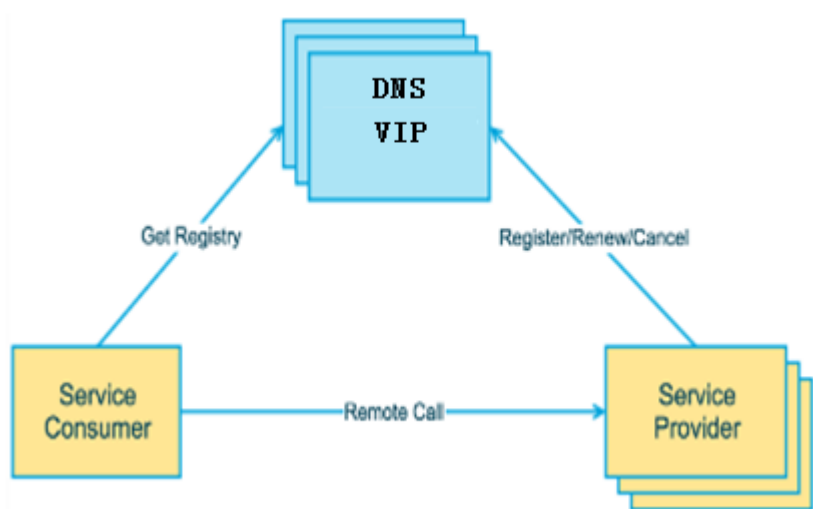


基于K8S容器集群的微服务架构



- 自动部署服务
- 灰度发布服务
- 自动水平扩展服务
- 集群内负载均衡服务
- 租户计算资源管理服务
- 增值服务（插件形式）
- 集群外调用负载均衡服务
- 镜像服务
- 日志监控分析服务
- 容器监控服务
- 分布式缓存服务
- 分布式持久存储服务

微服务架构核心组件介绍

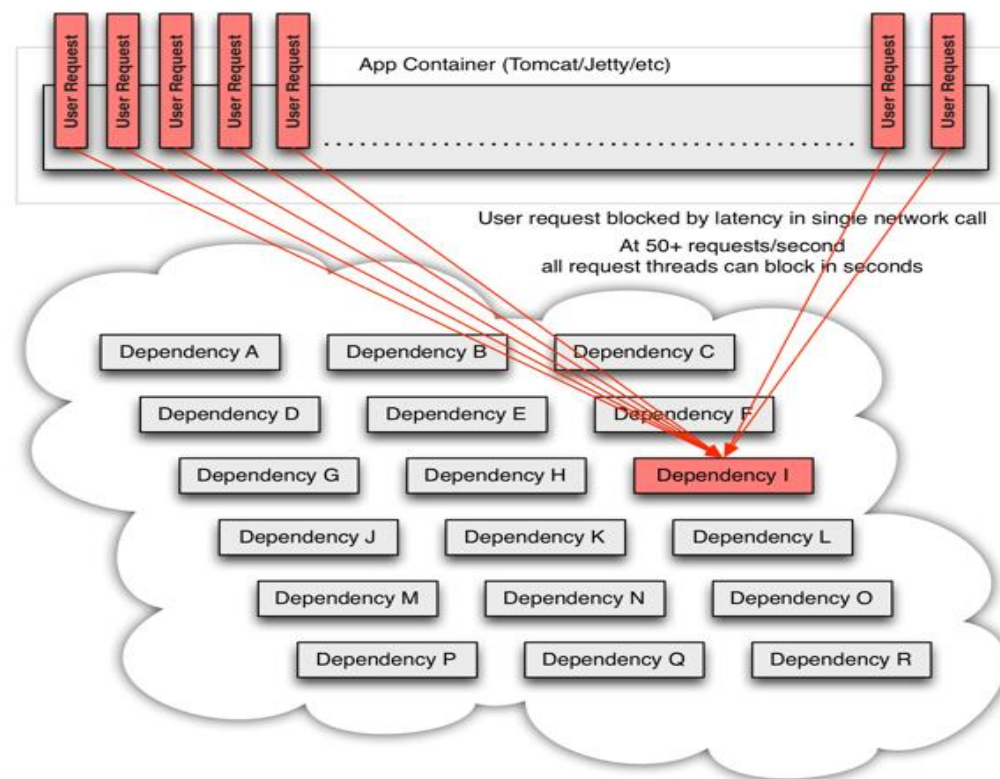
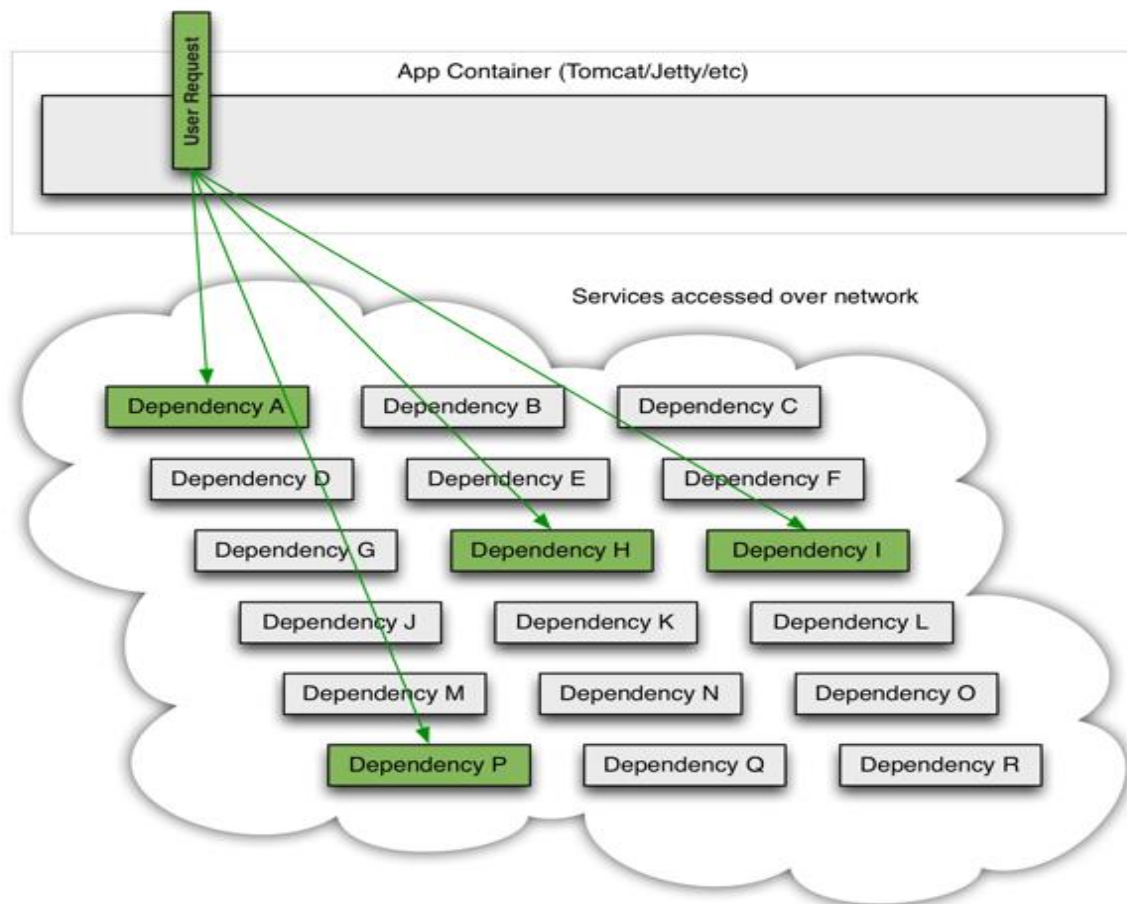


- 1、服务注册、发现、负载均衡和健康检查：**服务一般会有一个VirtualIP作为对外的接入点，可以通过DNS做解析。VirtualIP会对应到LB。LB会控制一组机器Cluster。健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制。失败的机器会及时从VIP里移除。并以新的机器作为补充。
- 2、限流和容错：**框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。
- 3、安全：**安全和访问控制逻辑可以在框架层统一进行封装，具体业务服务根据需要加载相关安全插件。
- 4、监控与日志：**框架一方面要记录重要的框架层日志、metrics和调用链数据，还要将日志、metrics等接口暴露出来，让业务层能根据需要记录业务日志。所有日志一般集中到后台日志系统，做进一步分析和处理。
- 5、配置：**除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。

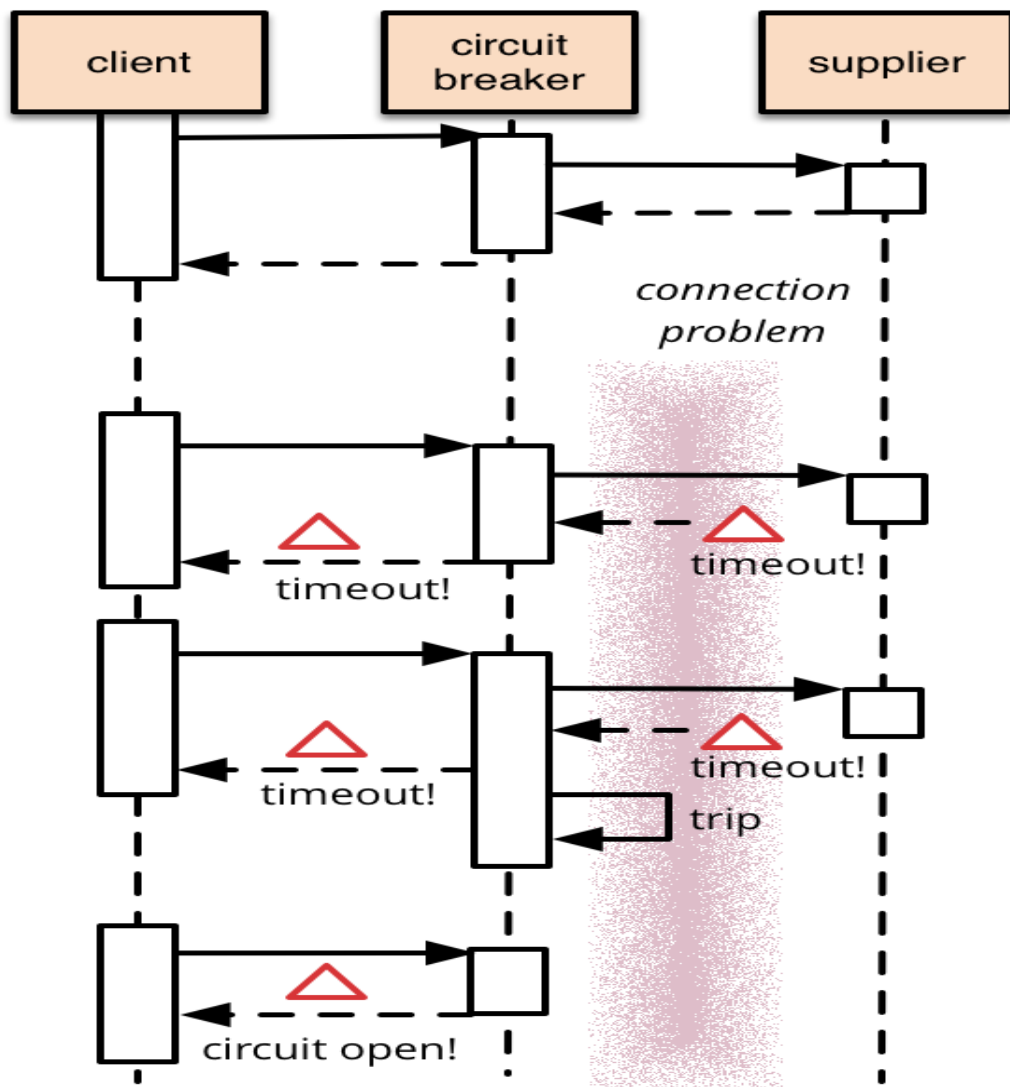
服务限流与容错

- **微服务依赖关系**：微服务化以后，服务之间会有错综复杂的依赖关系，一个前端请求一般会依赖于多个后端服务。

- **雪崩效应**：实际生产环境中，服务可能会出错或者延迟，如果不能对其依赖的服务故障进行容错和隔离，很快会导致应用资源(线程，队列等)被耗尽，造成雪崩效应(Cascading Failure)，可致整个系统瘫痪。



断路器（CircuitBreaker）设计模式



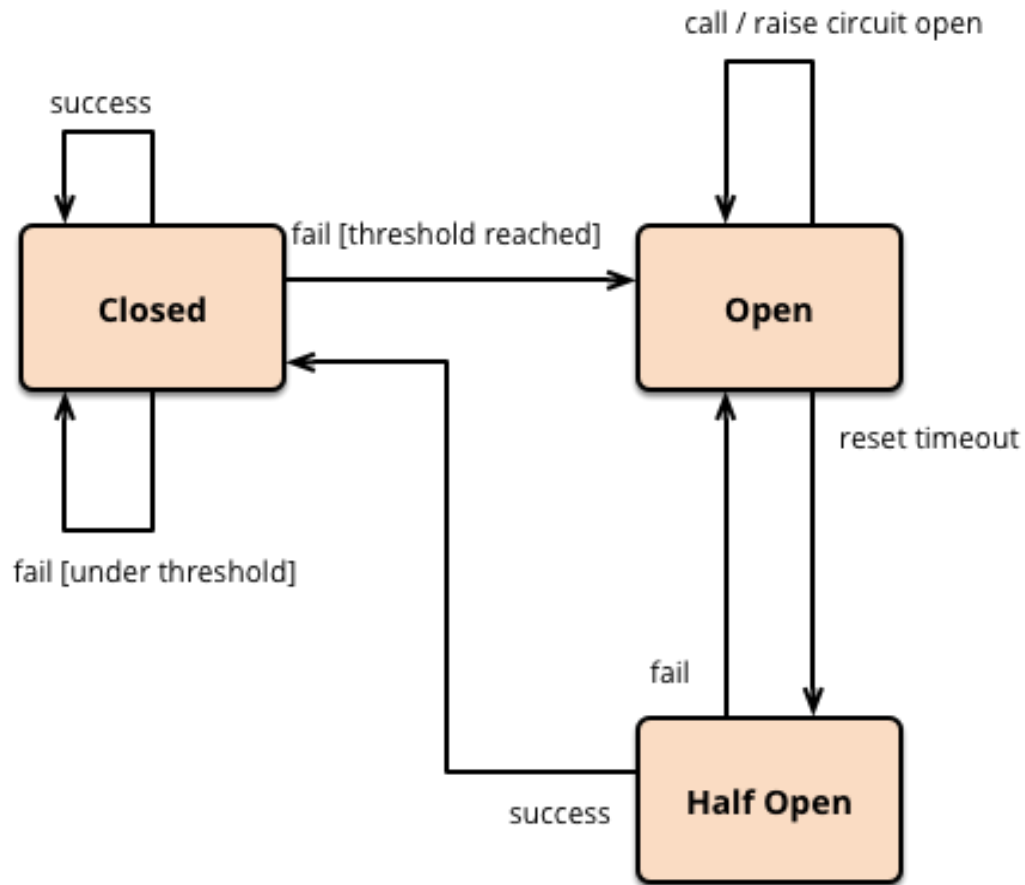
服务故障：网络连接、响应超时或者其他突发事件，导致服务无法响应请求，服务处于不可用状态。

分布式服务的容错通常有两种方法：

- **重试机制：**可预期的短暂故障，重试模式可以修复故障。
- **断路器模式：**将服务消费者封装在一个断路器对象中，当调用失败次数或者失败率达到断路门限，断路器将跳闸，后继调用请求将不会发往被依赖的服务，而由断路器对象快速返回错误。

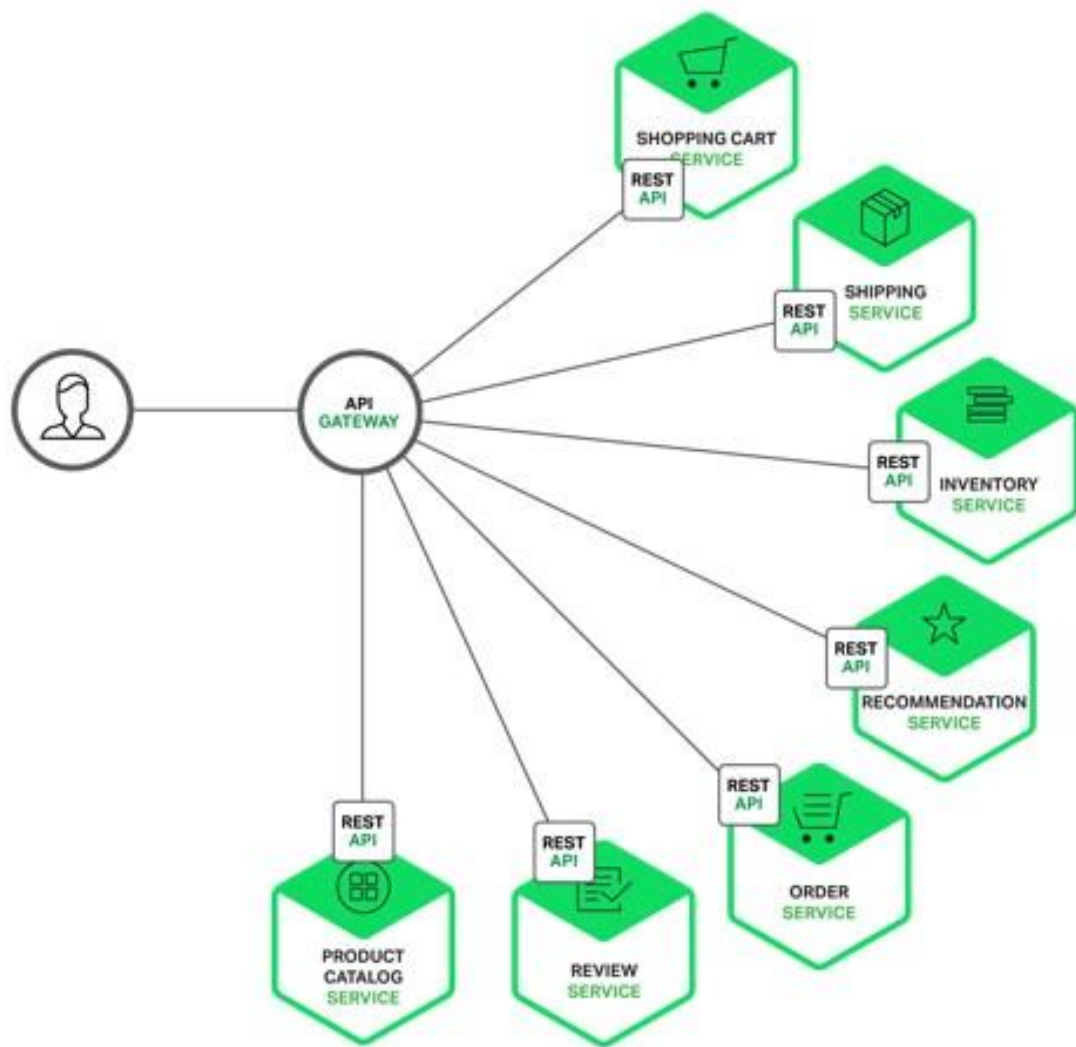
对于需要长时间解决的故障问题，不断重试就没有太大意义了，可以使用断路器模式解决。

断路器设计模式--状态机控制



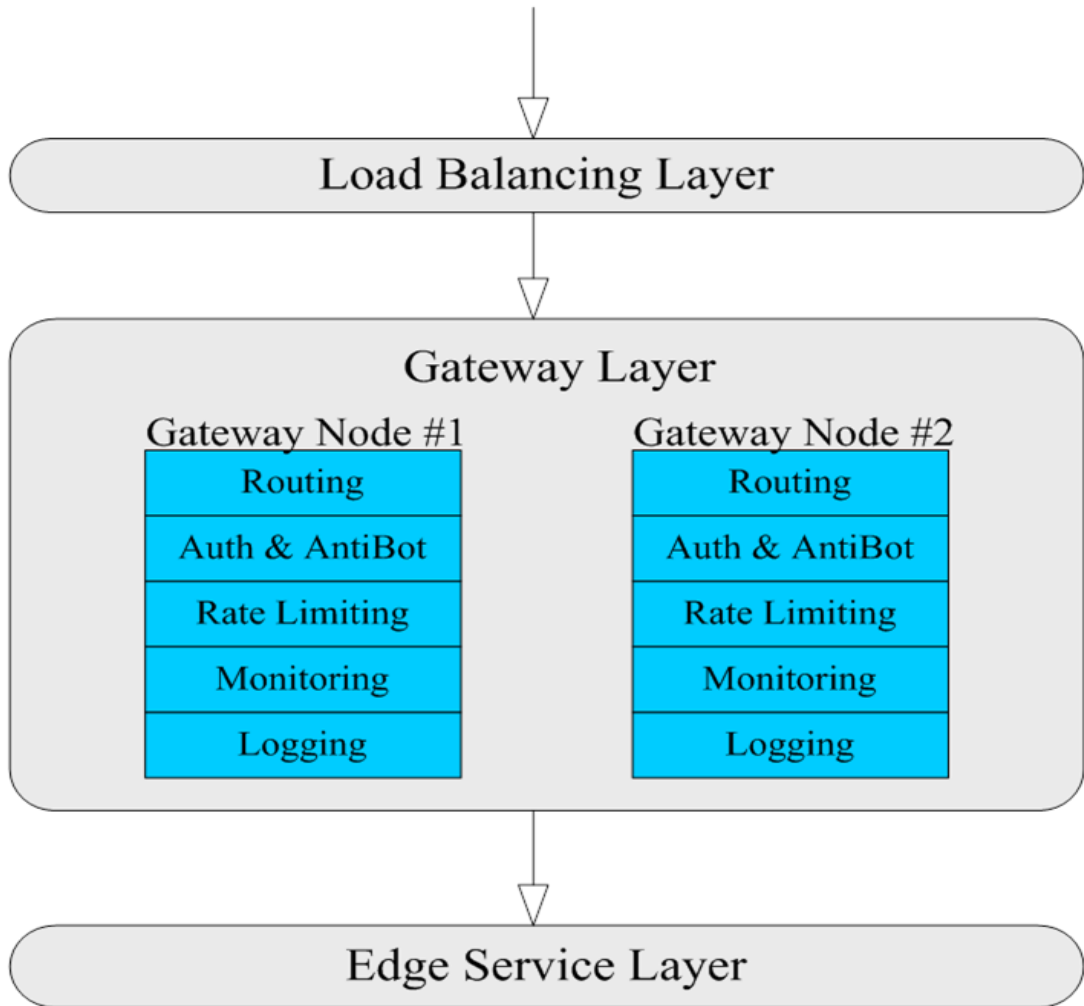
- **异常处理**：通过断路器调用的应用程序，必须能够处理在操作不可用时可能被抛出的异常，应用程序会暂时降级其功能。
- **异常类型**：一些请求故障严重类型高于其他故障。断路器应能检查发生的异常类型，并根据这些异常本质调整策略。
- **日志记录**：一个断路器应记录所有失败的请求（如果可能的话记录所有请求），以使管理员能够监视它封装下受保护操作的运行状态。
- **可恢复性**：应该配置断路器成与受保护操作最匹配的恢复模式。避免断路器处于开状态的时间过长，或者从开状态到半开状态切换过快。
- **手动复位**：如果受保护操作暂时不可用，管理员可以强制断路器进入放状态（并重新启动超时定时器）。
- **加速断路**：有时失败响应包含足够的信息，用于判定应当立即跳闸并保持最小时间量的跳闸状态。
- **重试失败请求**：在开状态下，断路器可以不是快速地简单返回失败，而是将每个请求的信息记录日志，并在远程服务重新可用时安排重试。

服务网关



- 微服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到，这就涉及服务的前端路由，对应的组件是服务网关(Service Gateway)。
- 网关是连接系统内部和外部系统的一道门。
- 网关还可以实现线上引流，线上压测，线上调试 (Surgical debugging)，金丝雀测试(Canary Testing)，数据中心双活(Active-Active HA)等高级功能。

服务网关架构及其主要功能

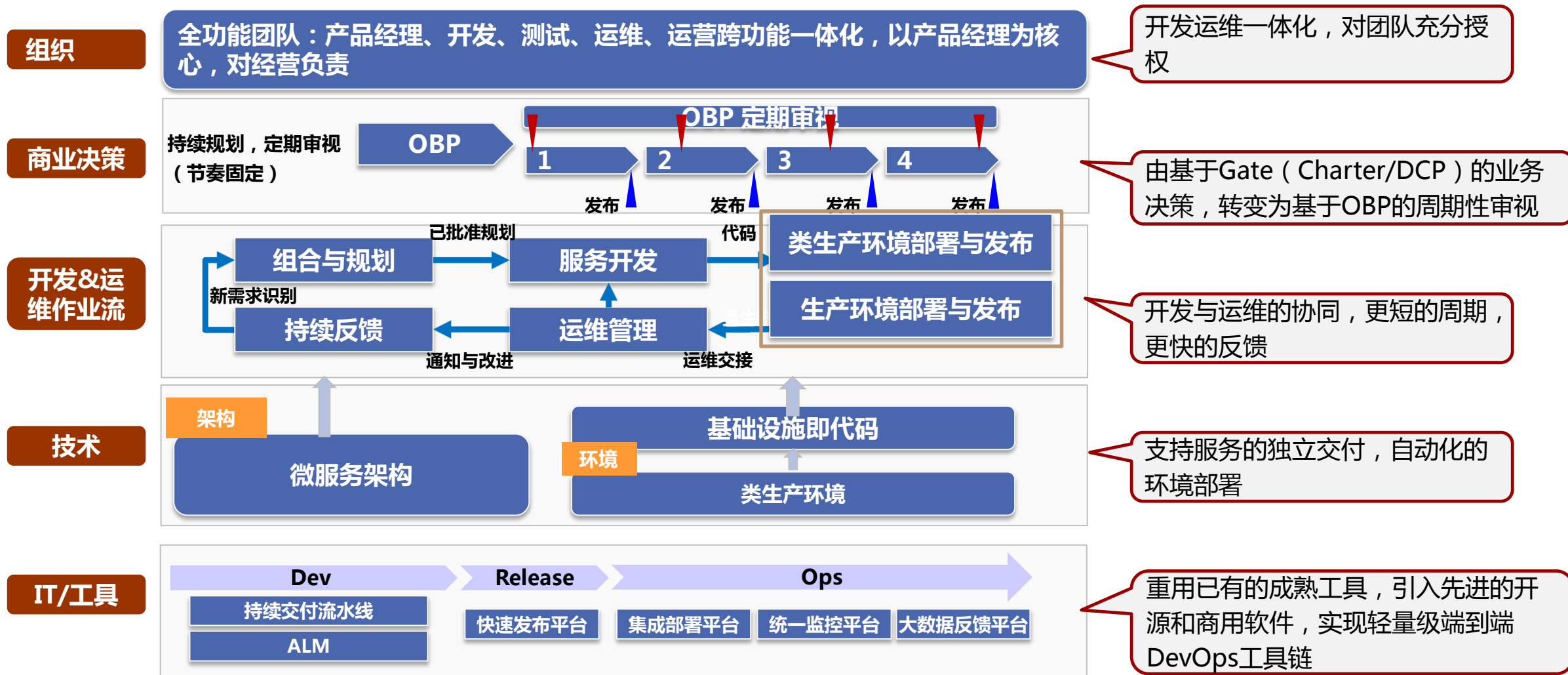


- **服务反向路由**：网关要负责将外部请求反向路由到内部具体的微服务，这样虽然企业内部是复杂的分布式微服务结构，但是外部系统从网关上看到的就像是一个统一的完整服务，网关屏蔽了后台服务的复杂性，同时也屏蔽了后台服务的升级和变化。
- **安全认证和防爬虫**：所有外部请求必须经过网关，网关可以集中对访问进行安全控制，比如用户认证和授权，同时还可以分析访问模式实现防爬虫功能，网关是连接企业内外系统的安全之门。
- **限流和容错**：在流量高峰期，网关可以限制流量，保护后台系统不被大流量冲垮，在内部系统出现故障时，网关可以集中做容错，保持外部良好的用户体验。
- **监控**：网关可以集中监控访问量、调用延迟、错误计数和访问模式，为后端的性能优化或者扩容提供数据支持。
- **日志**：网关可以收集所有的访问日志，进入后台系统做进一步分析。

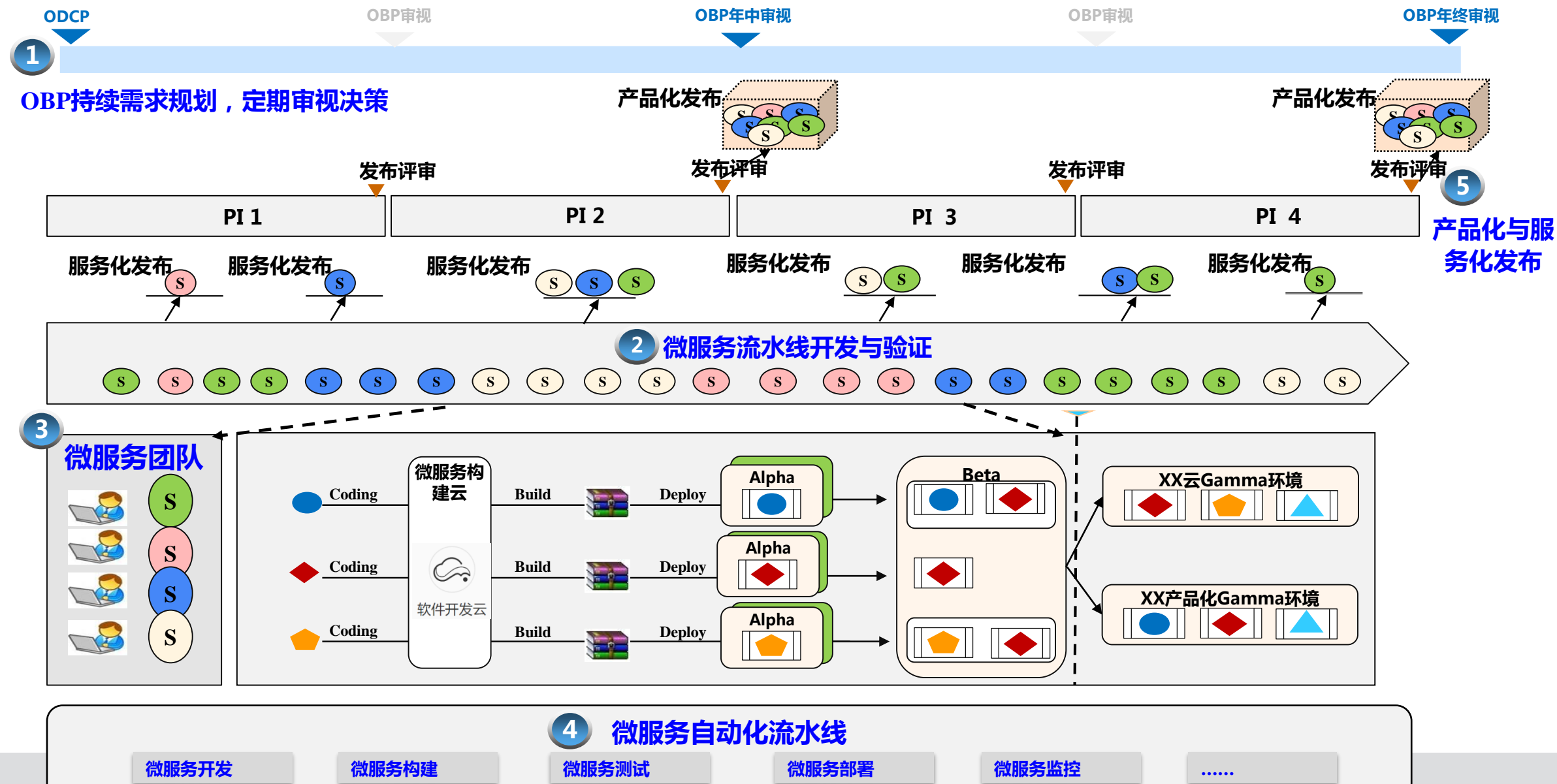
目 录

- 为什么要用微服务
- 什么是微服务
- 微服务架构介绍
- **微服务框架实践**

微服务开发模式总体方案



基于微服务架构的持续规划、并行流水线开发和持续发布

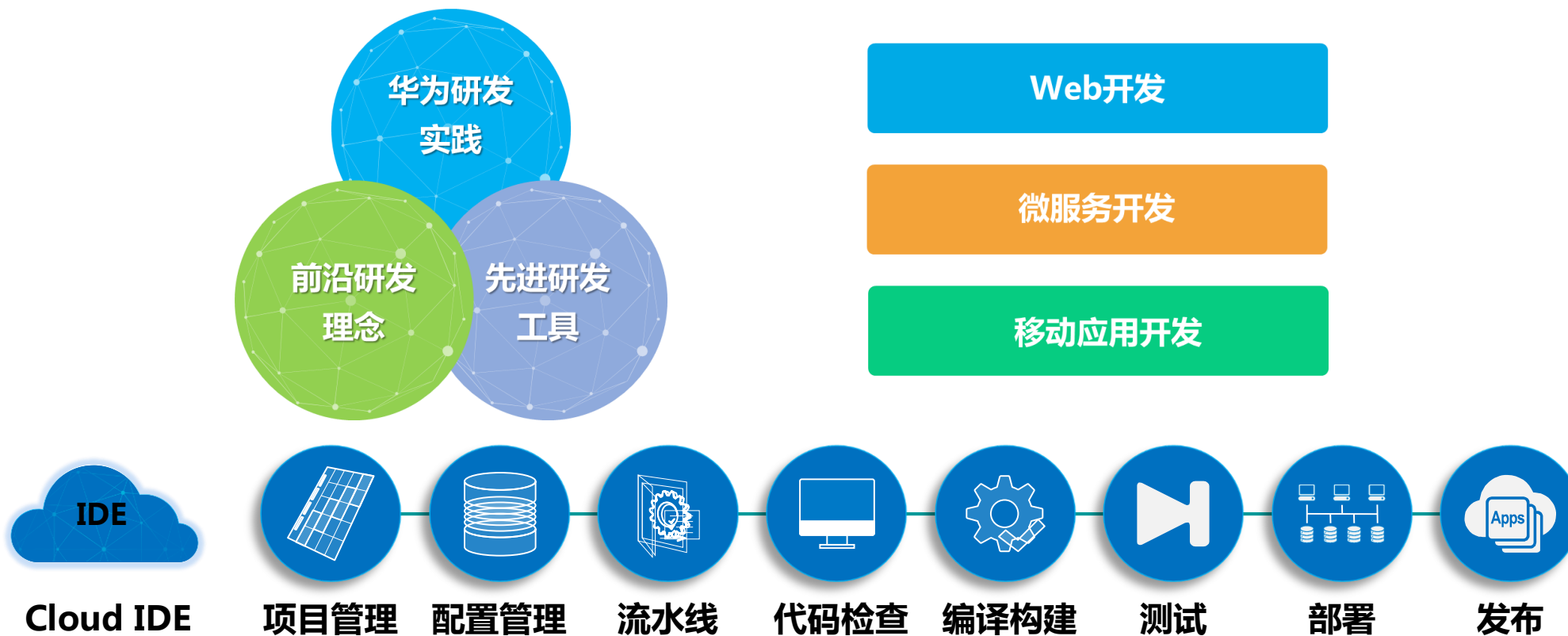


微服务关键工具链的总结

- 初创工具：创建初始文件和环境设置
- 代码存储和调试工具
- 代码审视工具
- 编译工具：产生目标版本集
- 资源管理：为流水线申请和管理环境资源
- 持续交付流水线：在流水线定义阶段和门禁
- 部署工具：把版本集部署到持续交付流水线的各阶段的目标环境
- 仪表盘：为实时监控和报警显示服务指标
- 工单管理系统：管理与跟踪线上服务问题工单
- 反馈系统：大数据分析**DevOps**数据，改进研发效率与服务功能

华为软件开发云

软件开发云（DevCloud）是一站式云端DevOps平台，集华为研发实践、前沿研发理念、先进研发工具为一体的研发云平台；面向开发者提供研发工具服务，让软件开发简单高效



谢谢