Security Level:

微服务架构设计与实践

华为软件开发云
曾正阳

www.huawei.com

目 录

未来市场对产品的需求


SHORTER DELIVERY TIMES
VOLATILE MARKETS
24/7 SERVICE
24/7
SHORTER PRODUCT LIFE CYCLES
1 year
2 years
3 years
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
- 互联网公司、中小企业、初创公司
- 小团队：2 Pizza Team
- TTM: 按天、周进行升级发布
- DevOps: CI, CD, 全自动化
- 可扩展性：自动弹性伸缩
- 高可用：升级、扩容不中断业务

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

目 录

什么是微服务？

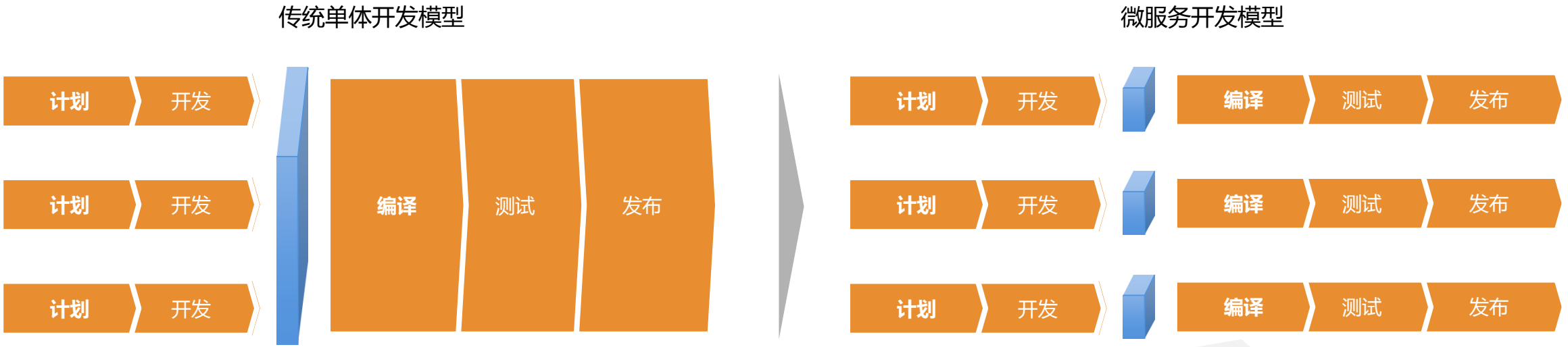
微服务是一个软件架构形式。在这个架构中，复杂的应用程序是由多个小而独立的进程组成，每一个进程通过独立于语言的接口进行相互交流。这些服务较小、高度解耦且专注于完成一个小任务，使得用模块化方法建设系统更加容易。

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

微服务的特点：

- **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为微服务模型



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

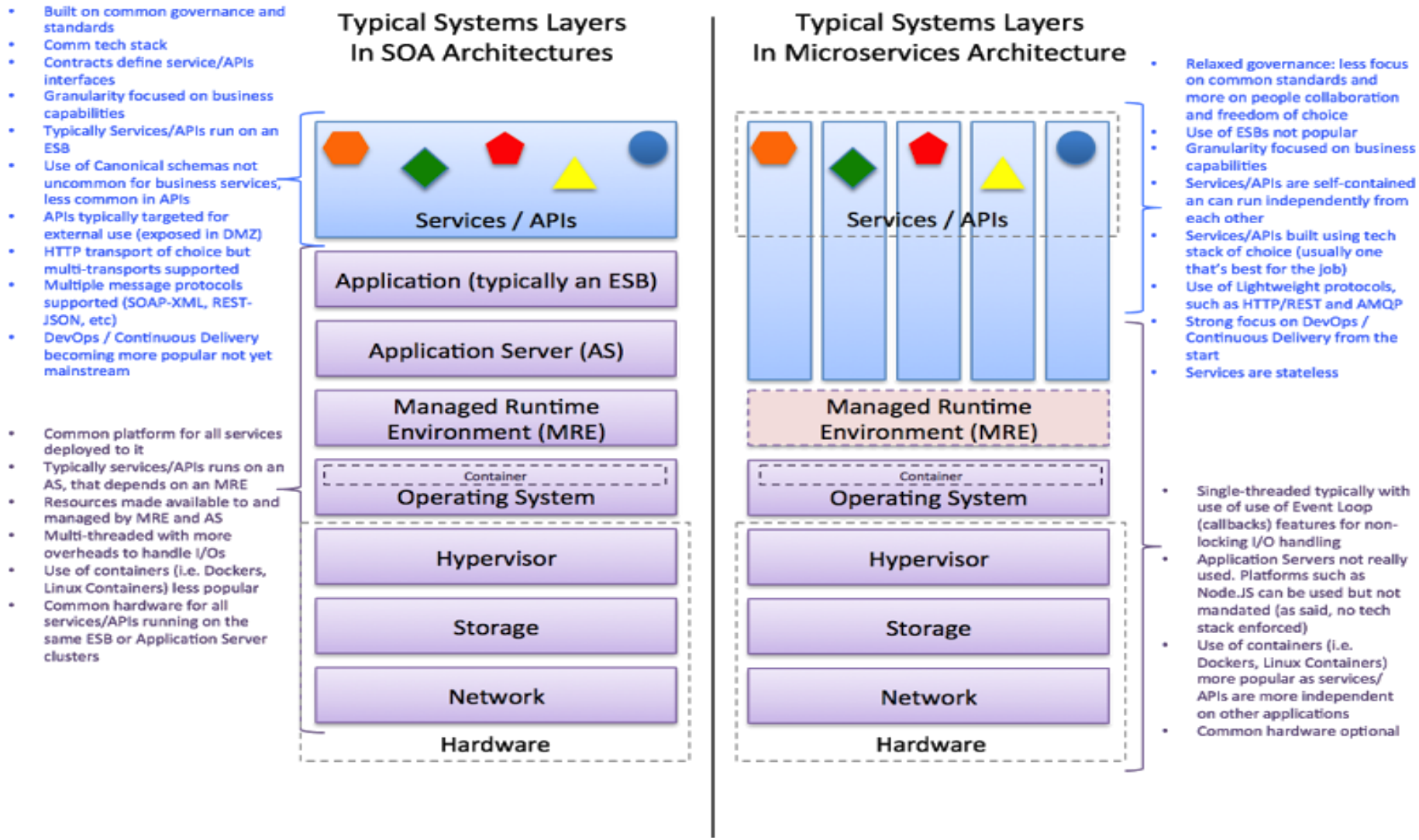
微服务架构的特性——Martin Fowler

- **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 buid 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。
- **聪明的端点和哑管道：**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、监控和发布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。
- **考虑到失败的设计：**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：**发送时要保守，接收时要开放。按照伯斯塔尔法则的思想来设计服务调用，发送的数据要更保守，意味着最小化的传送必要的信息，接收时更开放意味着要最大限度的容忍信息的兼容性。 多余的信息不认识可以忽略，而不应该拒绝或抛出错误。

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？



- **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	• 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	• OLTP中间件。 • Weblogic, Websphere	• 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	• 全局事务。 • 商业关系新数据库如Oracle。	• 一般无全局事务和强一致性要求。 • 开源数据库mysql, prostegsql, NoSql. • 看业务场景和开发人员的能力。
适用场景	• 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。	• 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。

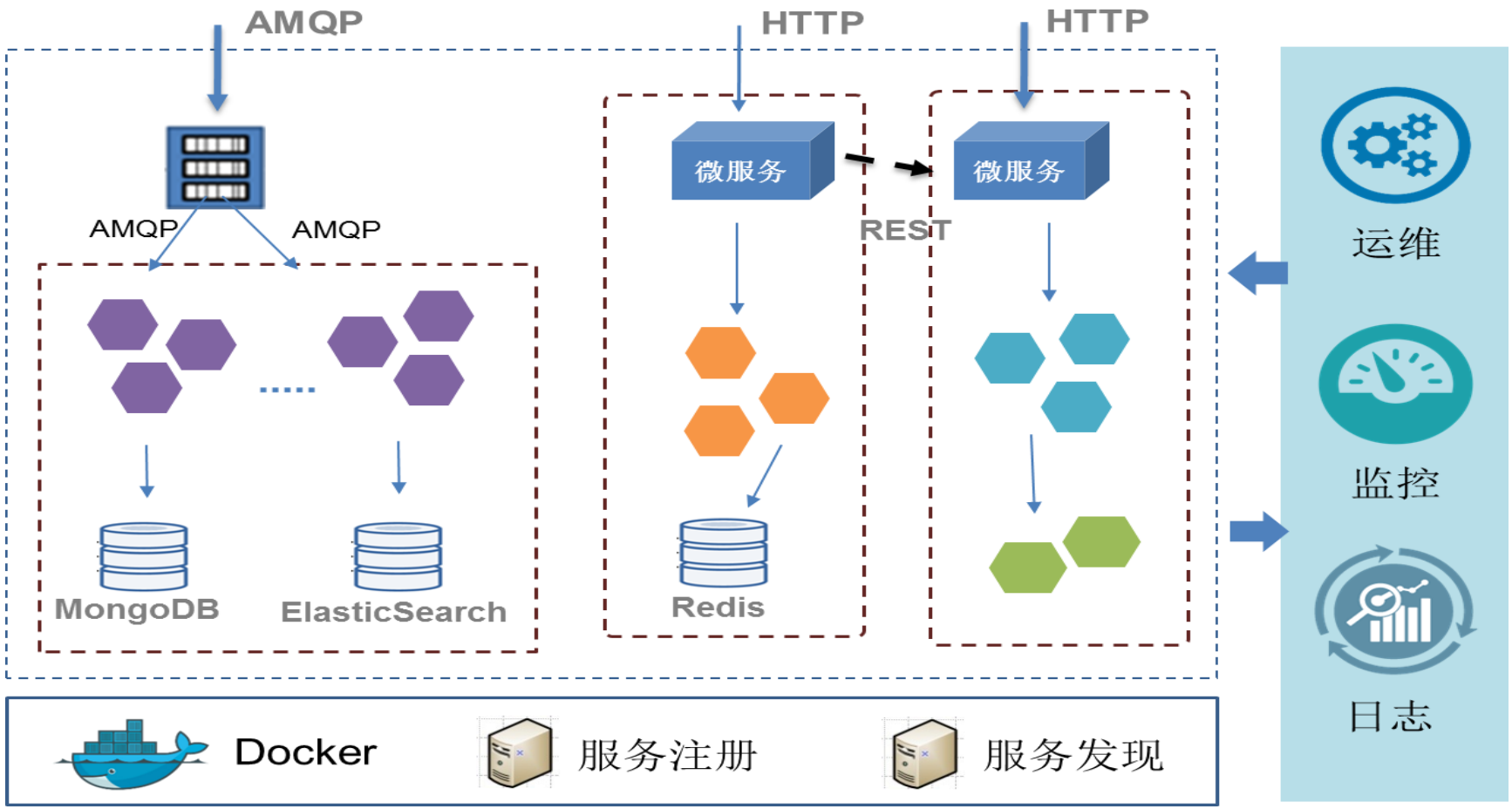
微服务实施要点

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。

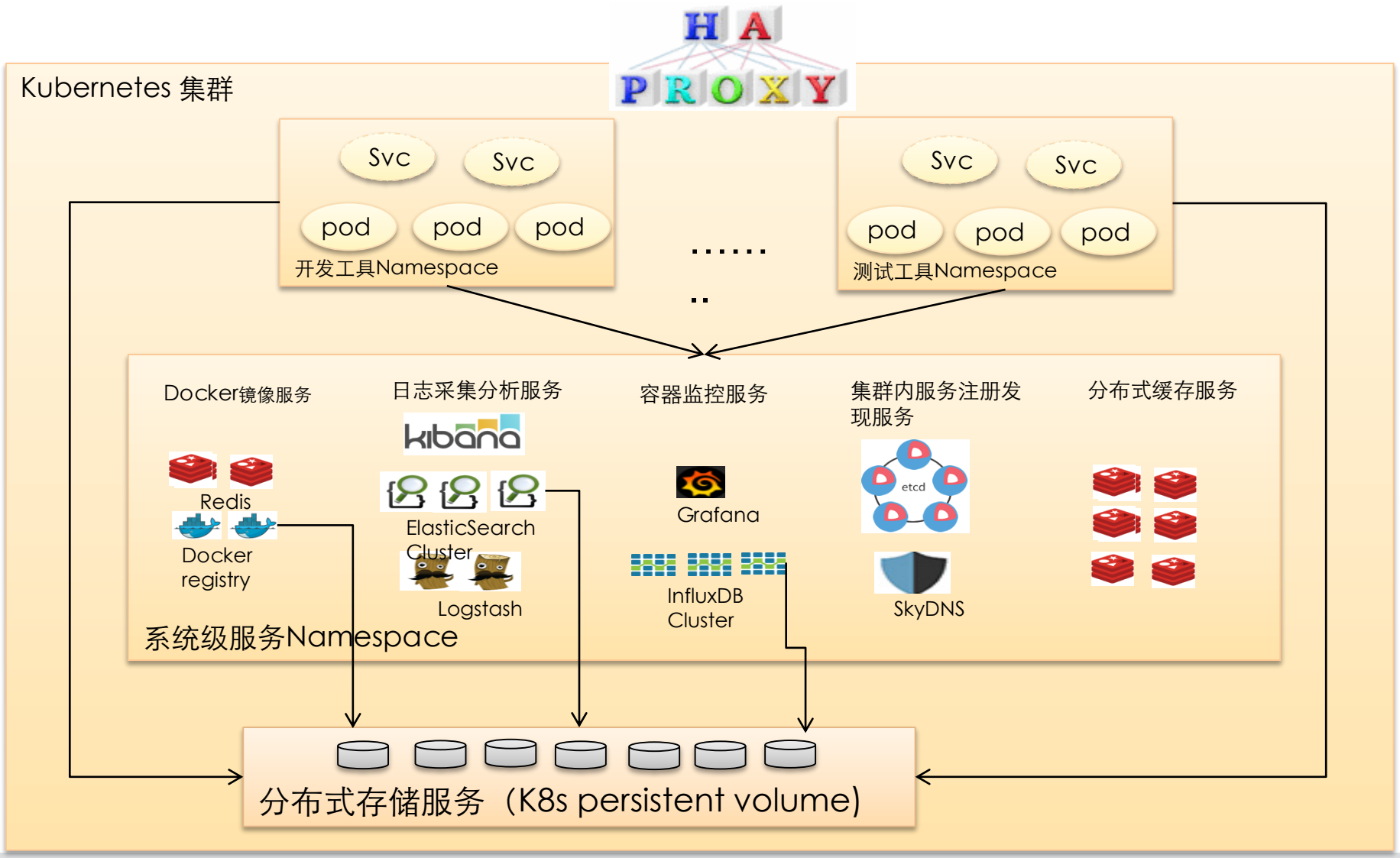
目 录

一个典型的微服务逻辑架构

一个典型的微服务架构

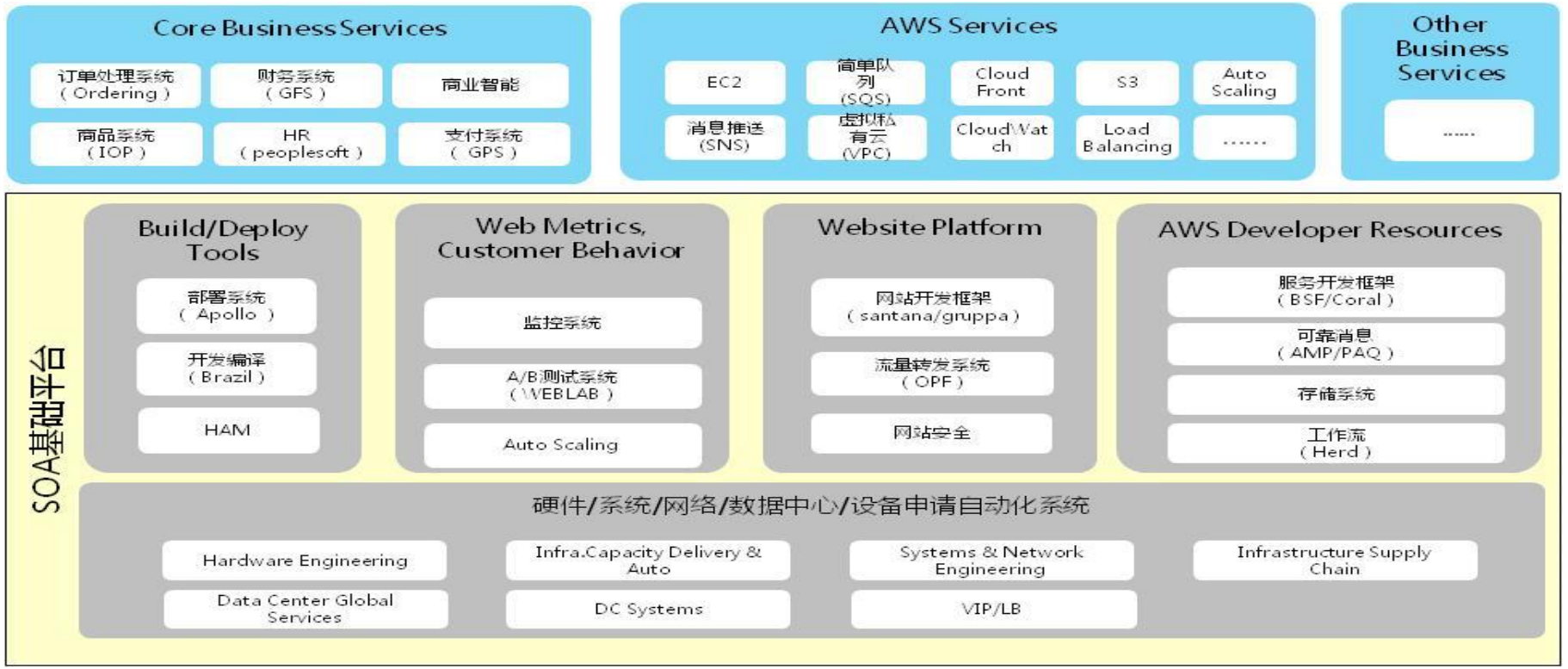

AMQP
HTTP
HTTP
AMQP
AMQP
微服务
微服务
REST
MongoDB
ElasticSearch
Redis
运维
监控
日志
Docker
服务注册
服务发现

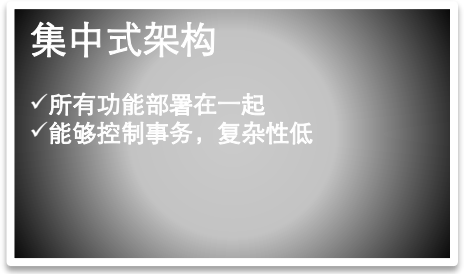
基于K8S容器集群的微服务架构

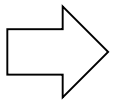


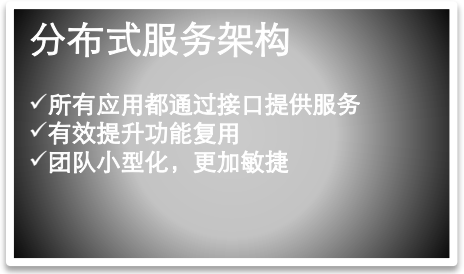
- 自动部署服务
- 灰度发布服务
- 自动水平扩展服务
- 集群内负载均衡服务
- 租户计算资源管理服务
- 增值服务（plugin形式）
- 集群外调用负载均衡服务
- 镜像服务
- 日志监控分析服务
- 容器监控服务
- 分布式缓存服务
- 分布式存储服务（Persistent）

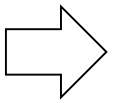
亚马逊的服务化架构

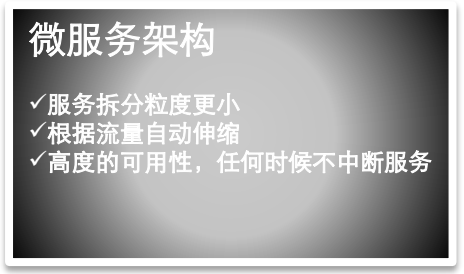






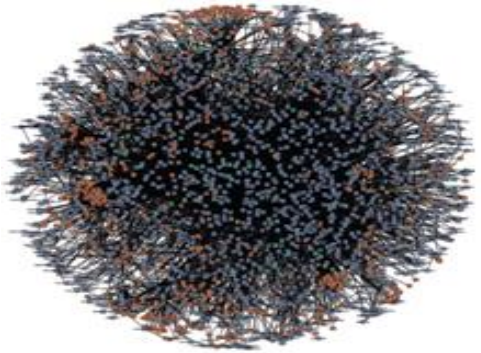






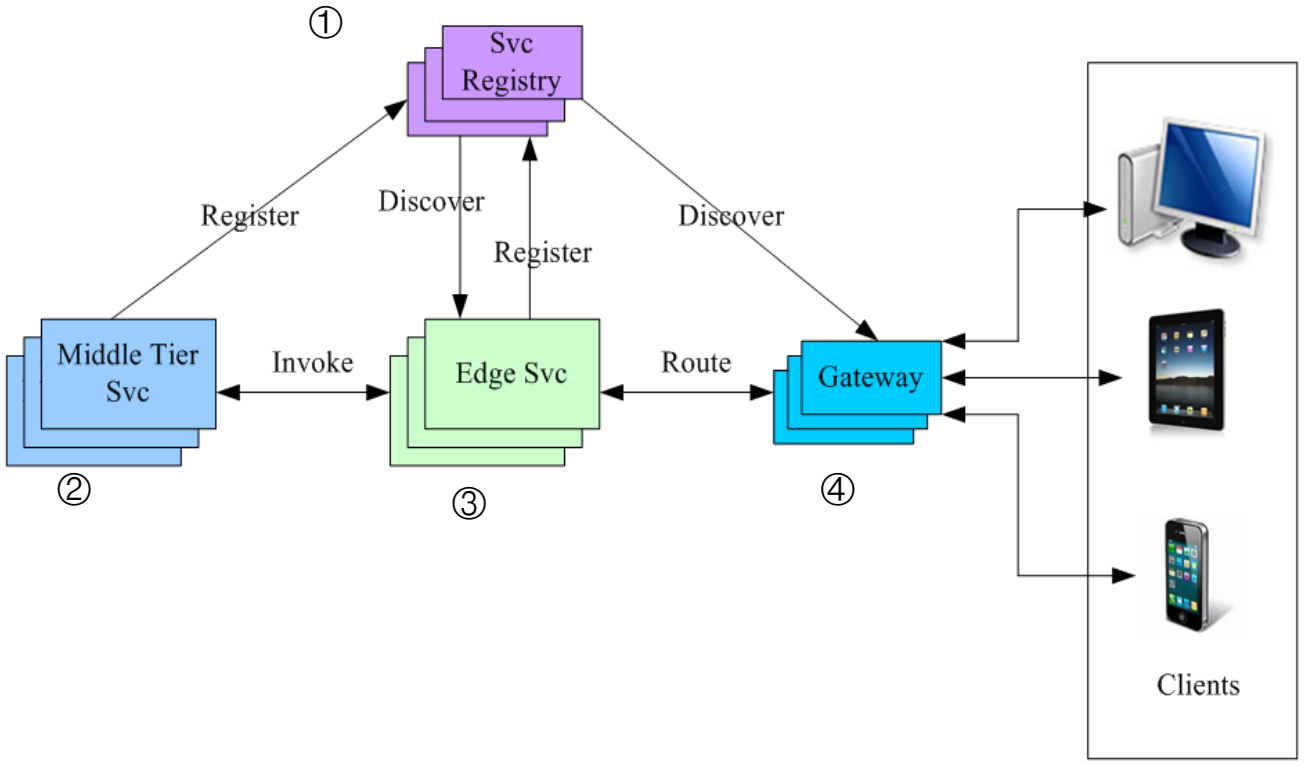
亚马逊架构演进 amazons-success-internal-apis

亚马逊服务依赖关系图



- 服务总数约为4000+，服务间调用连接约为10000+
- 所有应用的开发和运行都在aws上
- 所有应用都采用分布式服务架构
- 最复杂的部分沉淀在标准化的基础服务、框架、中间件、工具中，做到最稳定
- 让开发人员专注于业务，提升开发效率
- 高度自动化的持续发布能力
- 更快的交付速度
- 平均部署时间- 11.6秒
- 平均主机同时收到的部署任务 - 10,000
- 0.001%的部署导致停机
- 瞬时回滚

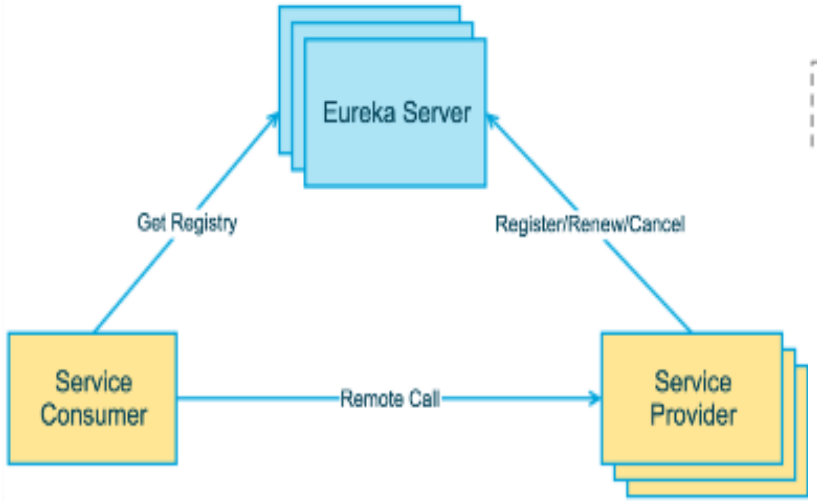
微服务架构核心组件（简化模型）

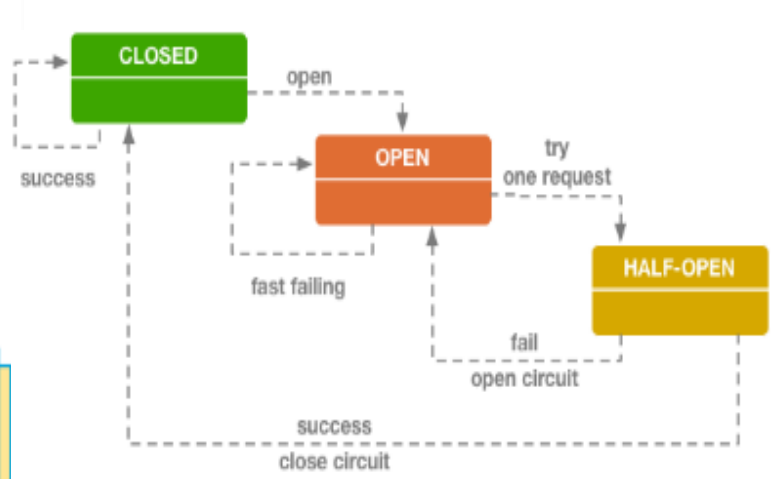


- **服务注册**：为服务提供方提供服务地址注册服务，并接受服务提供方定期上报心跳以表明服务的存活状态；为服务消费方的LB组件提供目标服务地址列表查询服务。
- **后端服务**：也称中间层服务Middle Tier Service ，提供具体业务服务。
- **前端服务**：也称边缘服务Edge Service，前端服务的作用是对后端业务服务做必要的聚合和裁剪后暴露给外部不同的设备，如PC，Pad或者Phone ）。
- **服务网关(Service Gateway)**：网关是连接企业内部和外部系统的一道门。
- **服务发现**：后端服务启动时，会将地址信息注册到服务注册表，前端服务通过查询服务注册表，就可以发现然后调用后端服务。前端服务启动时，也会将地址信息注册到服务注册表，这样网关通过查询服务注册表，就可以将请求路由到目标前端服务。

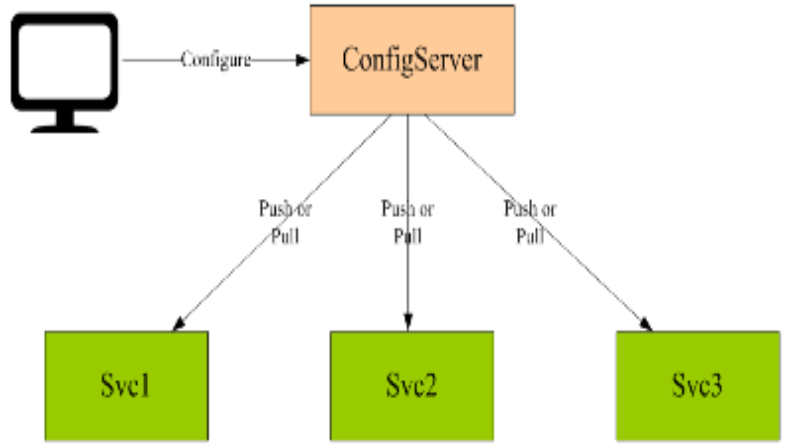
设计模式：网关类似Proxy代理或者Façade门面模式，而服务注册表和服务自注册自发现类似IoC依赖注入模式。微服务可以理解为基于网关代理和注册表IoC构建的分布式系统。

微服务架构核心组件介绍





Circuit Breaker State Diagram



1、**服务注册、发现、负载均衡和健康检查**：服务自注册一般做在服务器端框架中，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，服务发现和负载均衡则集成在服务客户端框架中。

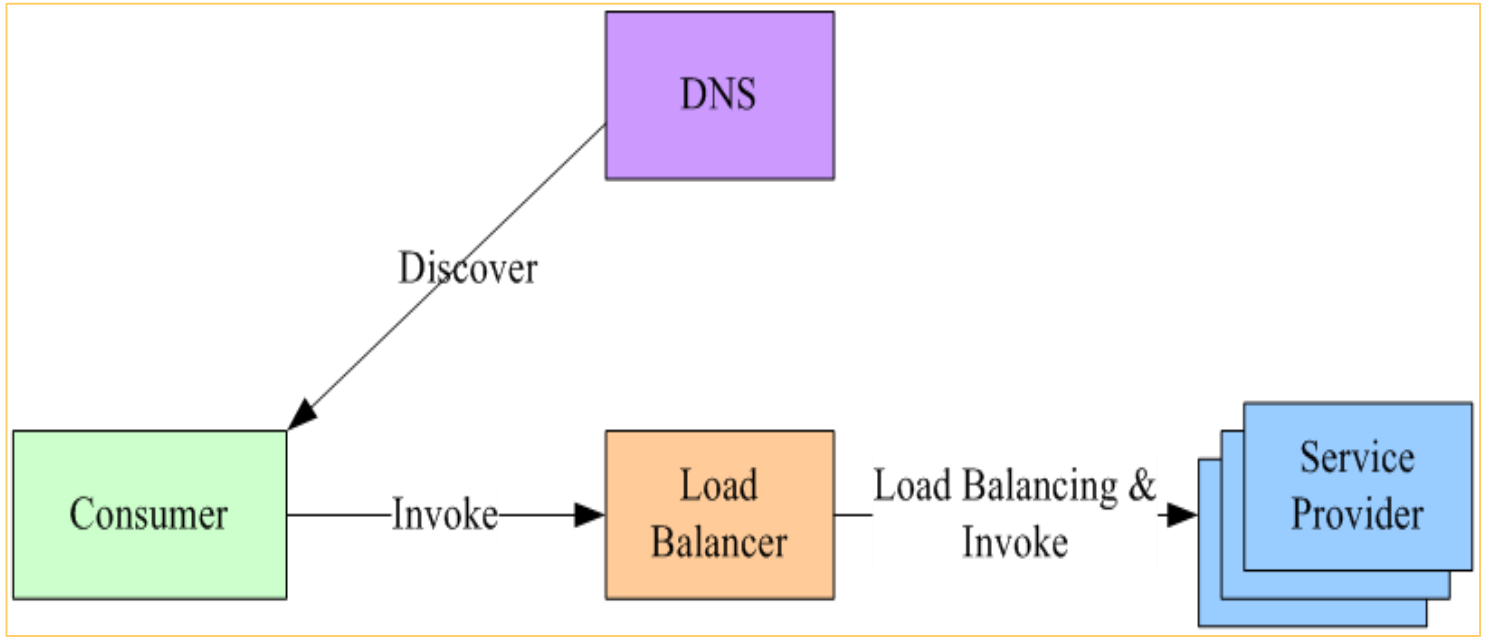
2、**限流和容错：**框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。

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。

传统LB方案：集中式LB



F5、 LVS、 Haproxy、 Nginx

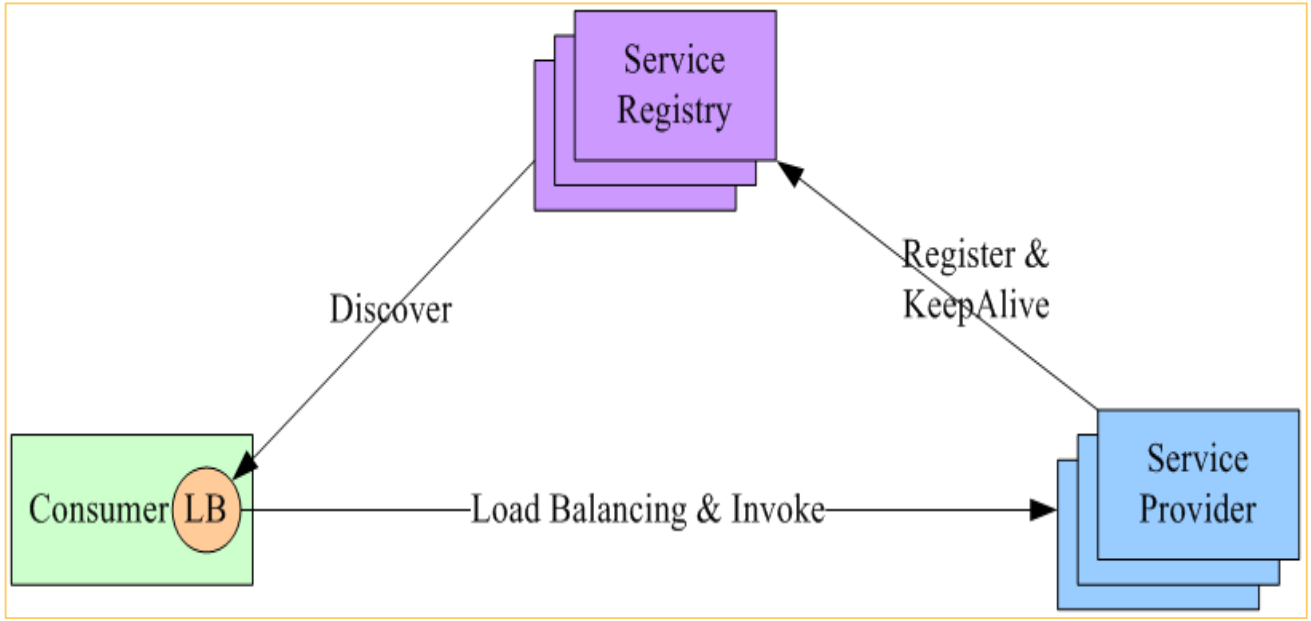
特点：

- 服务消费者和服务提供者之间有一个独立的LB。
- 具备健康检查能力，能自动移除不健康的服务实例。
- 服务消费方通过DNS发现LB，域名指向LB。

缺点：

- 集中式LB依赖DNS服务，且存在单点故障问题。
- 流量通过集中LB，造成性能开销。
- 流量很大时，集中LB成为性能瓶颈。

Eureka客户端分布式LB方案：进程内LB



进程内LB

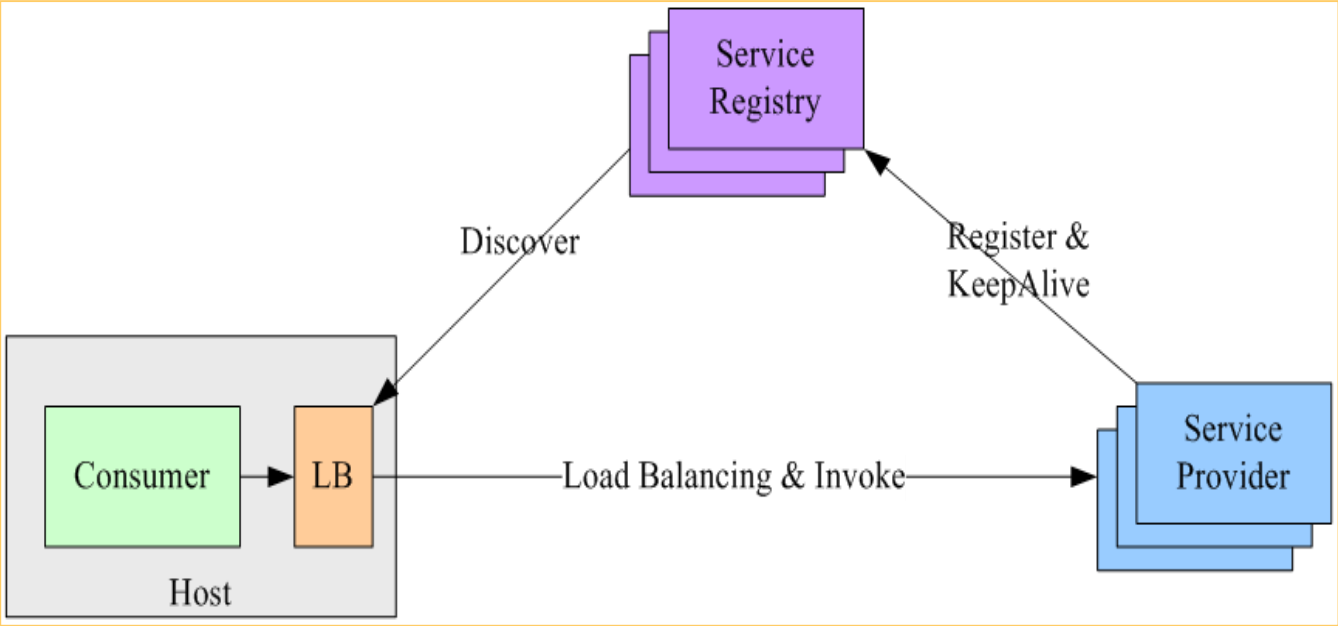
特点：

- 软负载或者客户端负载，无单点故障问题。
- LB和服务发现功能，在每一个消费者的进程内；不依赖DNS服务。
- 消费方和服务提供方之间直接调用，没有额外开销，性能好；不存在集中式性能瓶颈。

缺点：

- LB客户库的方式集成到服务调用方进程里面，受技术栈约束。
- LB客户库版本升级问题。

Kubernetes分布式LB方案：主机独立LB进程



主机独立LB进程

特点：

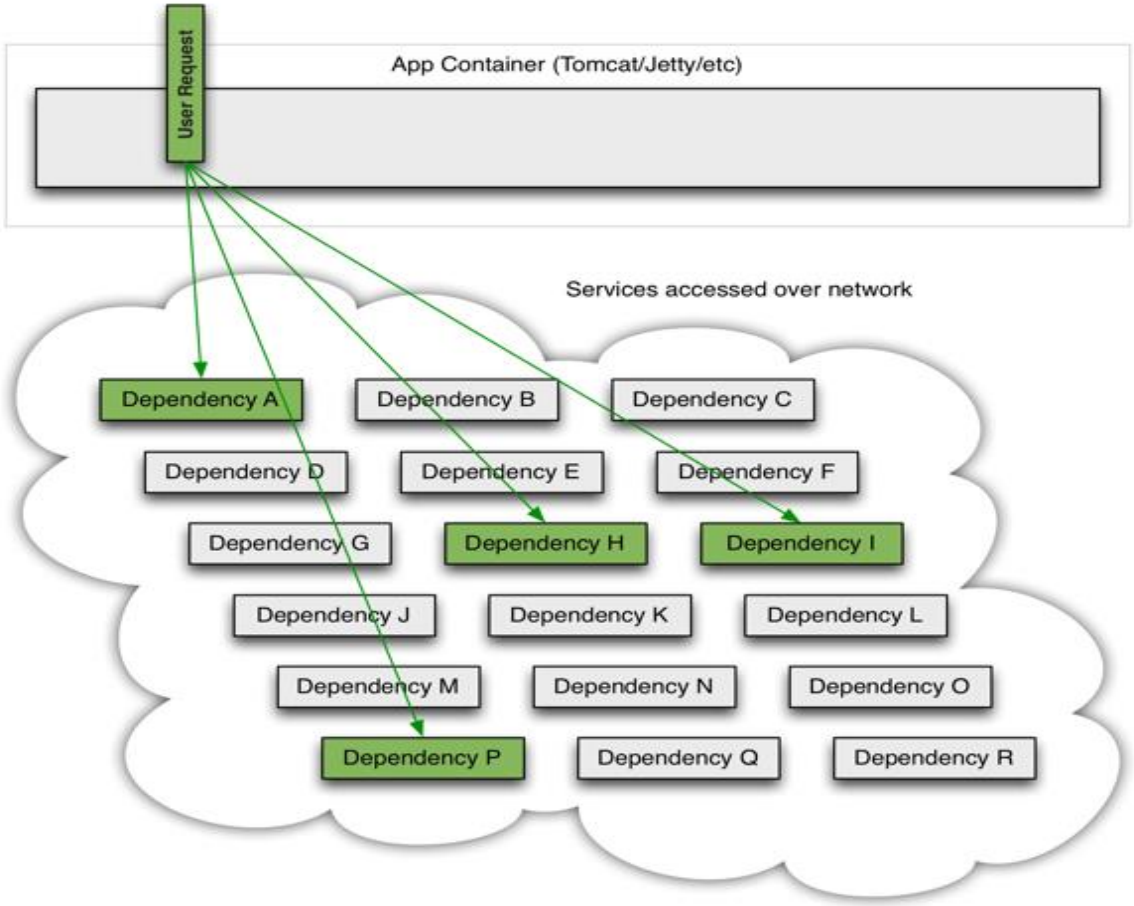
- 软负载或者客户端负载，无单点故障问题。
- LB和服务发现功能，从进程内移到一个独立进程；不依赖DNS服务。
- 服务调用方和LB之间是同一主机的进程间调用，性能开销较低；不受技术栈约束。

缺点：

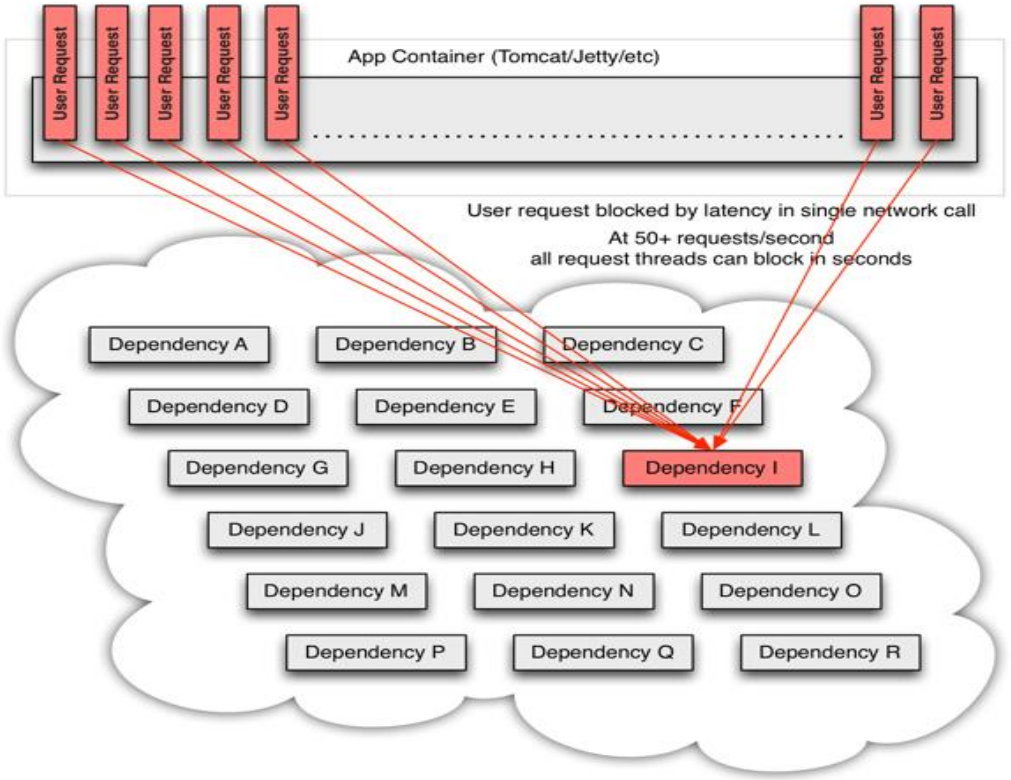
- 部署较复杂，节点多，问题定位复杂。

服务限流与容错

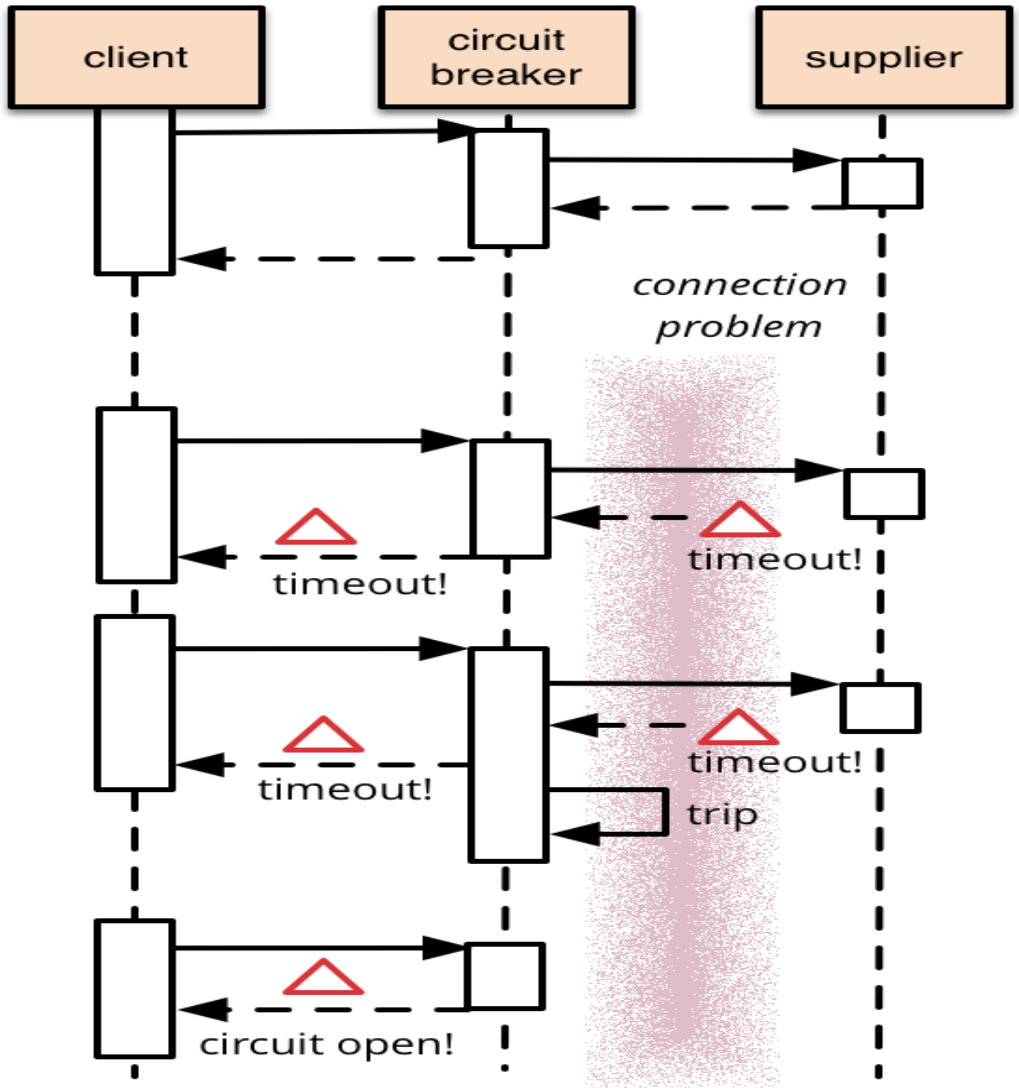
- **微服务依赖关系：**微服务化以后，服务之间会有错综复杂的依赖关系，一个前端请求一般会依赖于多个后端服务。



- **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



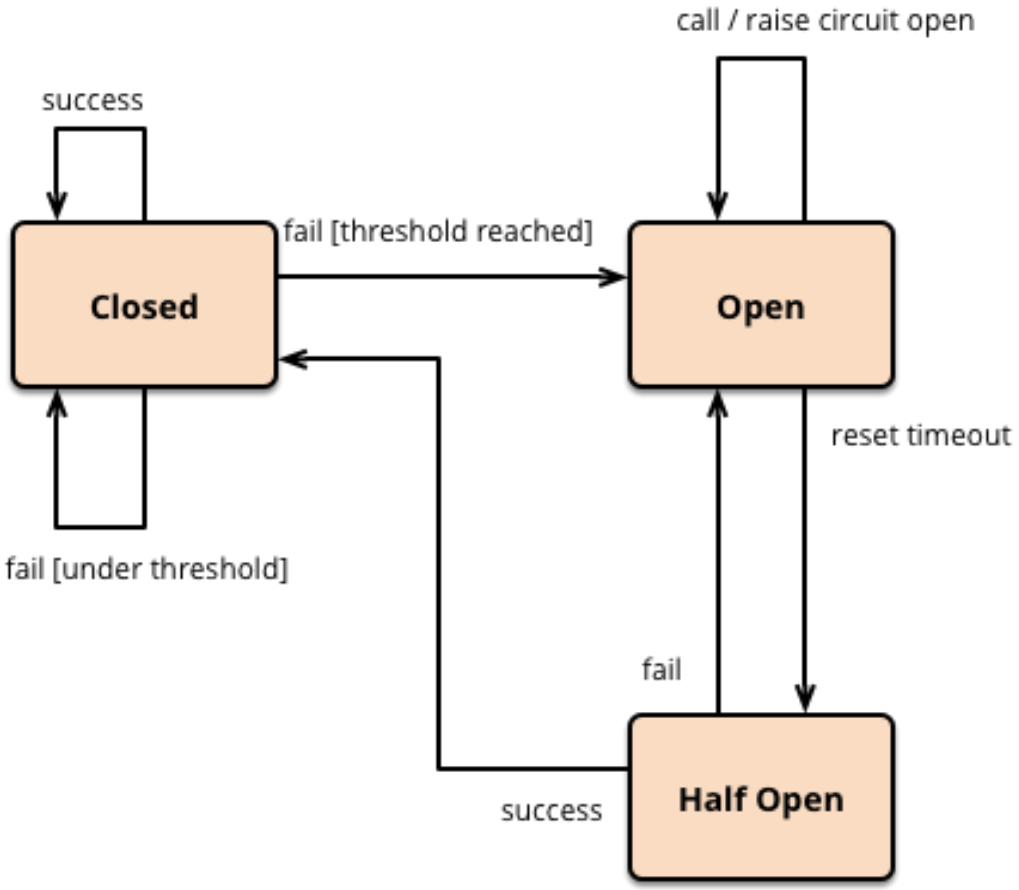
服务故障：网络连接、响应超时或者其他突发事件，导致服务无法响应请求，服务处于不可用状态。

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。

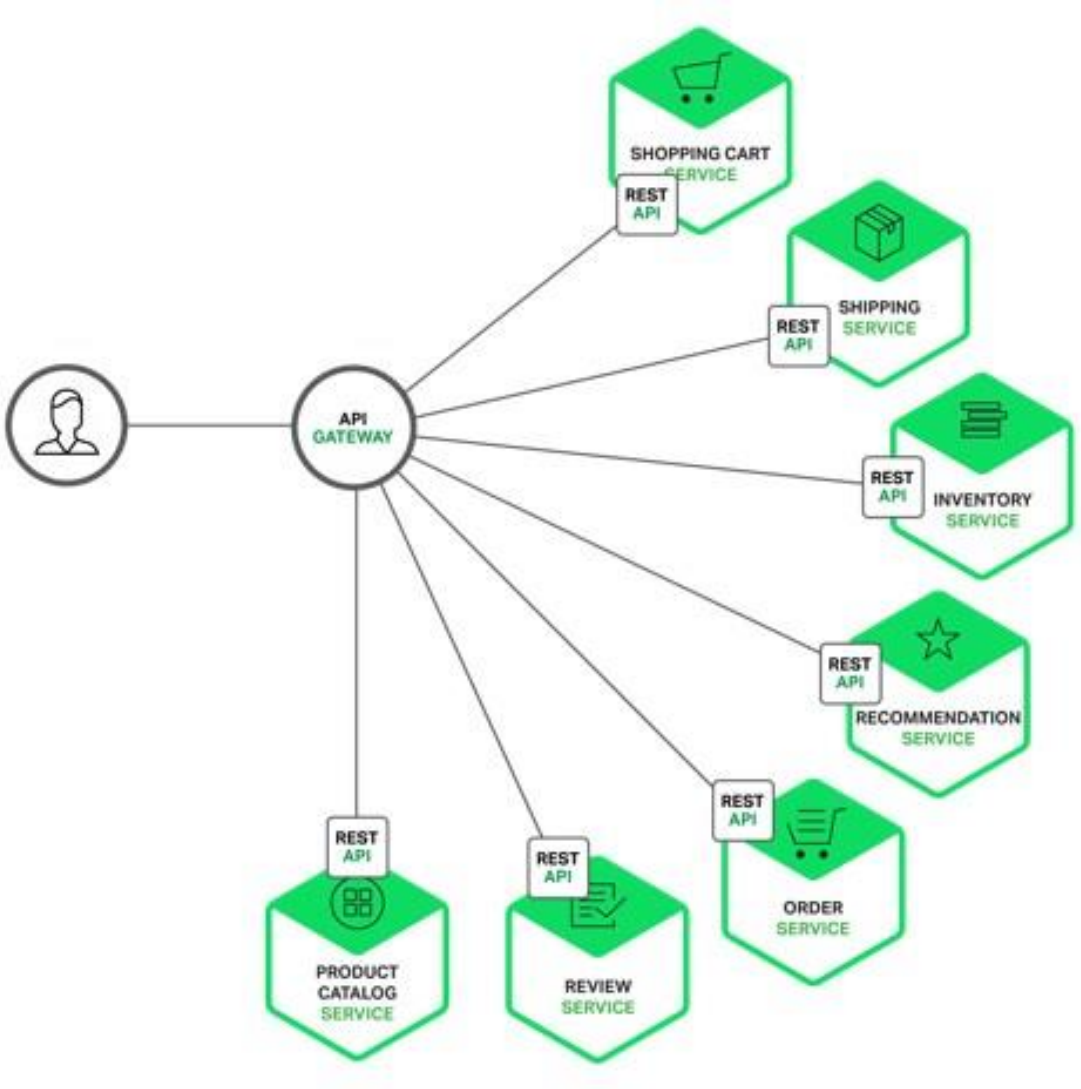
对于需要长时间解决的故障问题，不断重试就没有太大意义了，可以使用断路器模式解决。

断路器设计模式--状态机控制



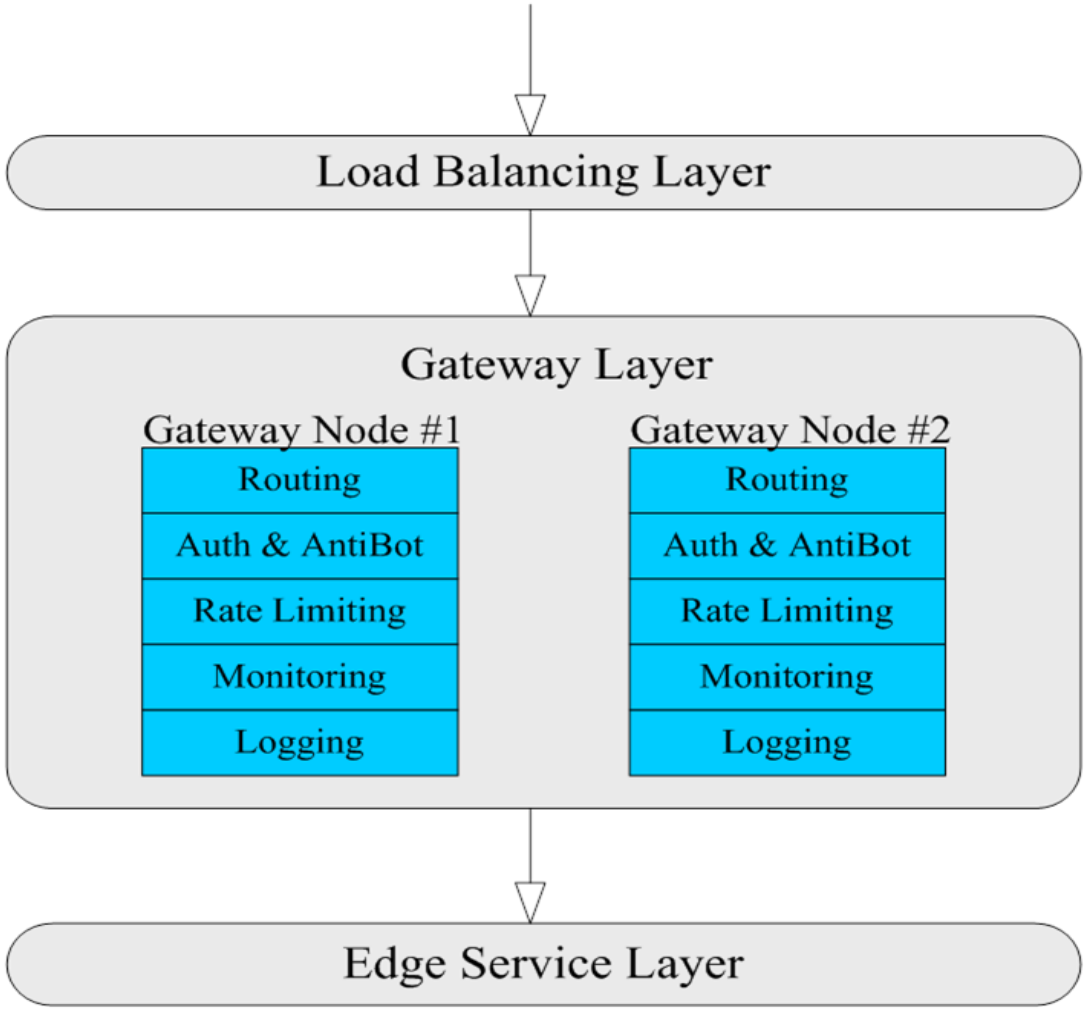
- **异常处理**：通过断路器调用的应用程序，必须能够处理在操作不可用时可能被抛出的异常，应用程序会暂时降级其功能。
- **异常类型**：一些请求故障严重类型高于其他故障。断路器应能检查发生的异常类型，并根据这些异常本质调整策略。
- **日志记录**：一个断路器应记录所有失败的请求（如果可能的话记录所有请求），以使管理员能够监视它封装下受保护操作的运行状态。
- **可恢复性**：应该配置断路器成与受保护操作最匹配的恢复模式。避免断路器处于开状态的时间过长，或者从开状态到半开状态切换过快。
- **手动复位**：如果受保护操作暂时不可用，管理员可以强制断路器进入放状态（并重新启动超时定时器）。
- **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



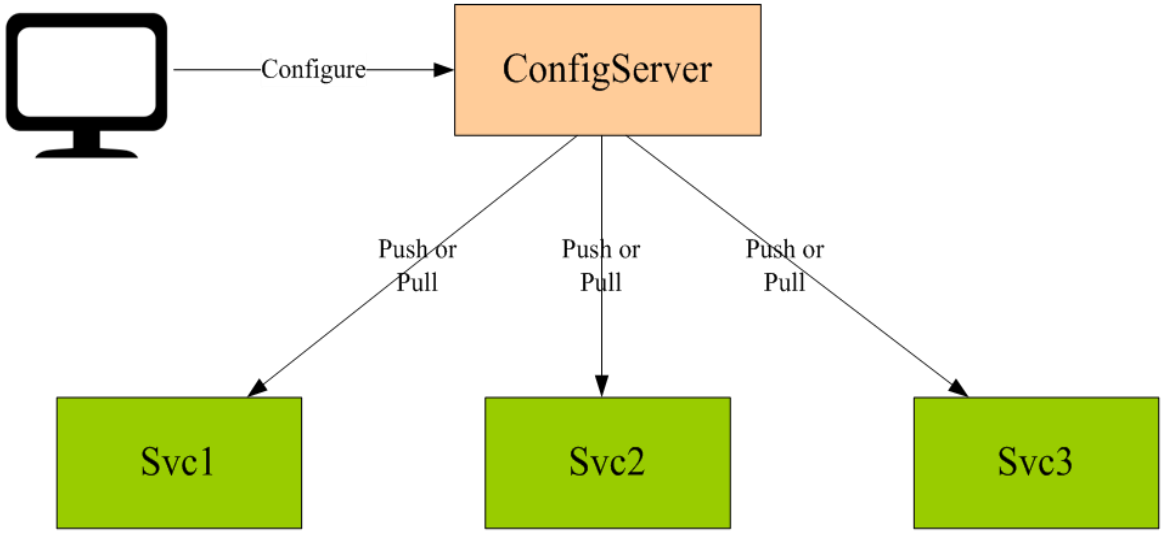
- 微服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到，这就涉及服务的前端路由，对应的组件是服务网关(Service Gateway)。
- 网关是连接系统内部和外部系统的一道门。
- 网关还可以实现线上引流，线上压测，线上调试(Surgical debugging)，金丝雀测试(Canary Testing)，数据中心双活(Active-Active HA)等高级功能。

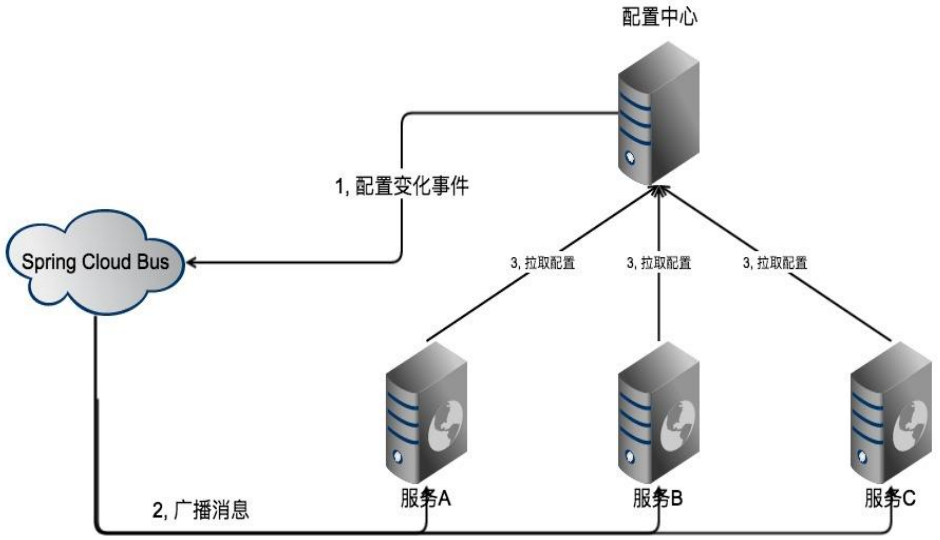
服务网关架构及其主要功能



- **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。

服务统一配置管理—堆拉模式





拉：具体服务通过定期拉(Scheduled Pull)的方式更新动态配置

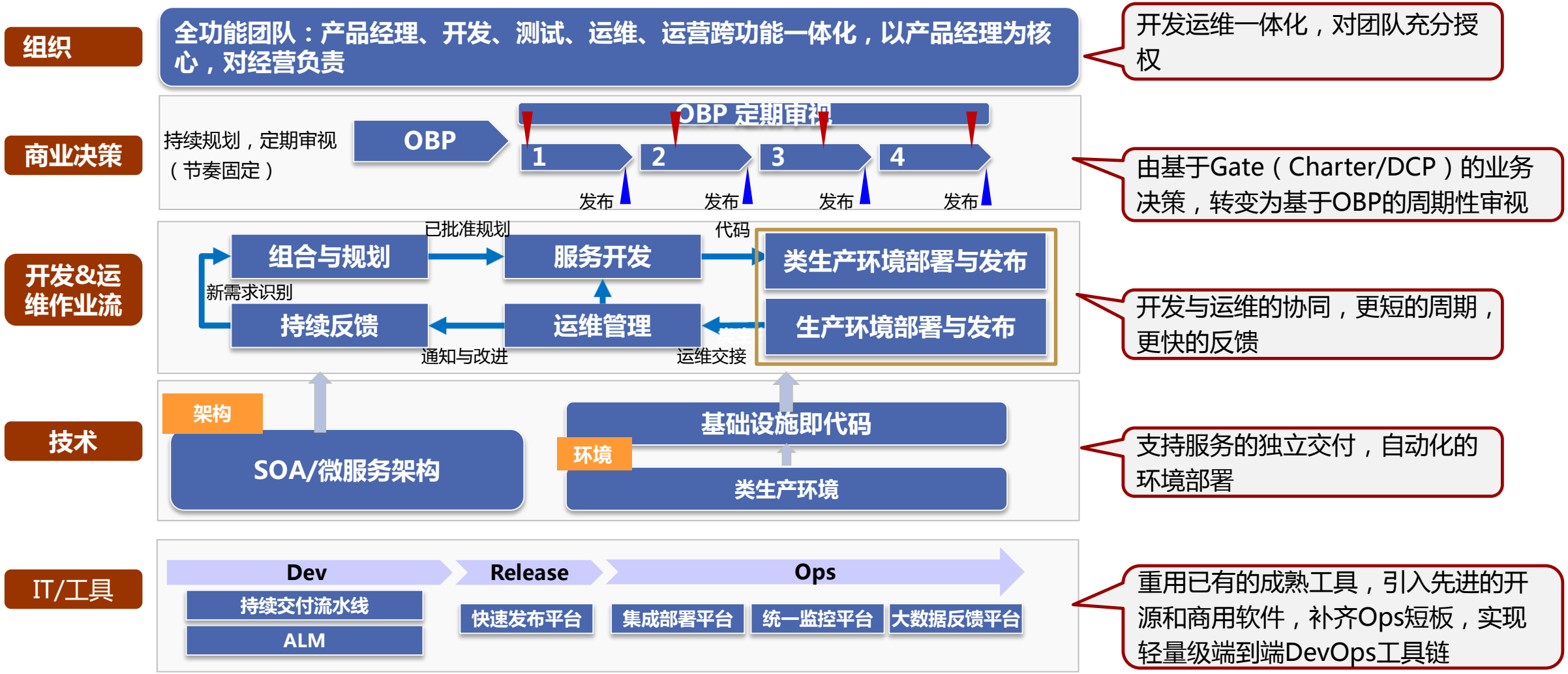
推：服务端推(Server-side Push)的方式更新动态配置

主要功能如下：

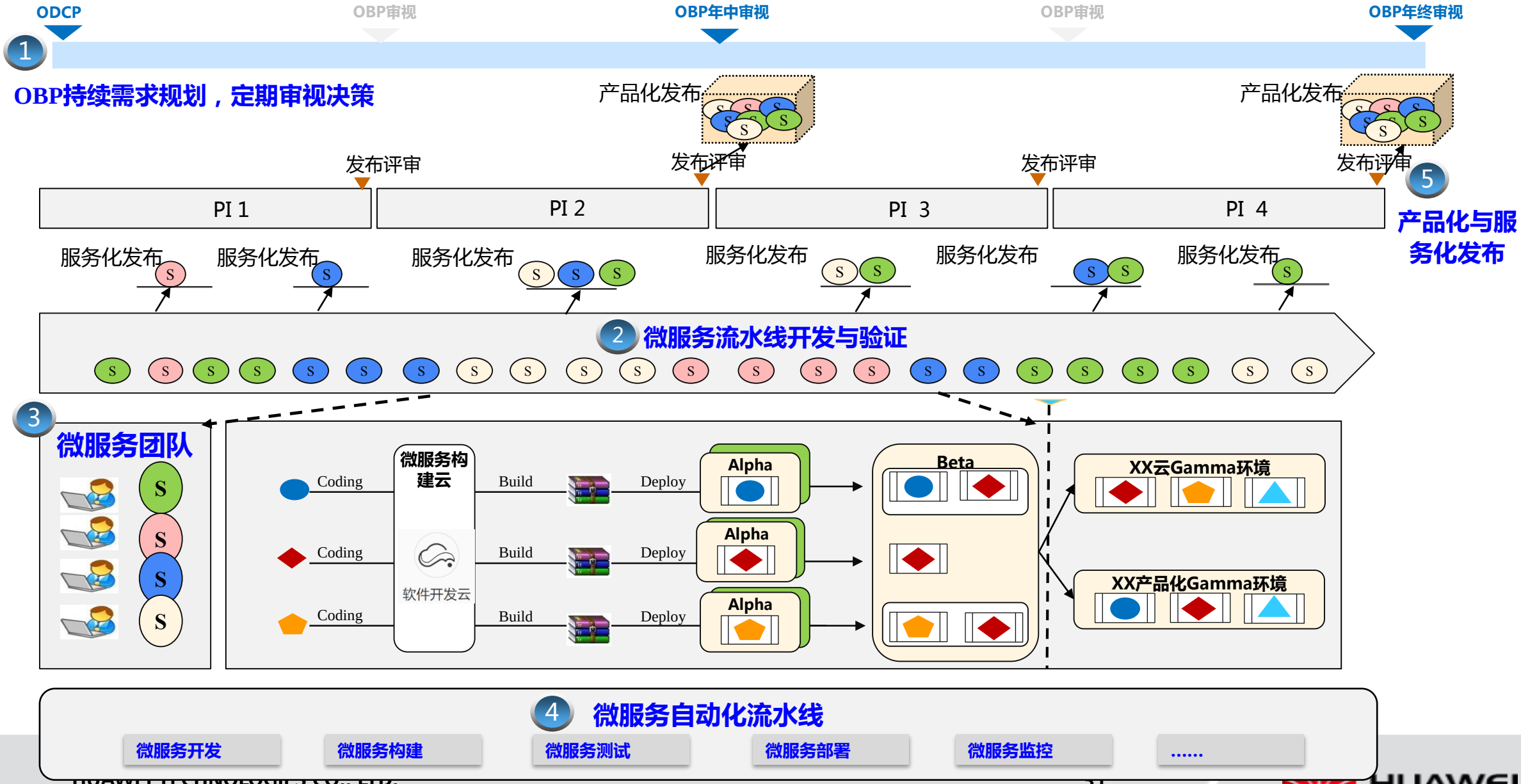
- **集中化的配置文件管理**：不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息。配置中心支持配置的版本控制和审计。
- **动态化的配置更新**：当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置。
- **HTTP REST接口**：非Java语言的服务也能享受到上述功能。

目 录

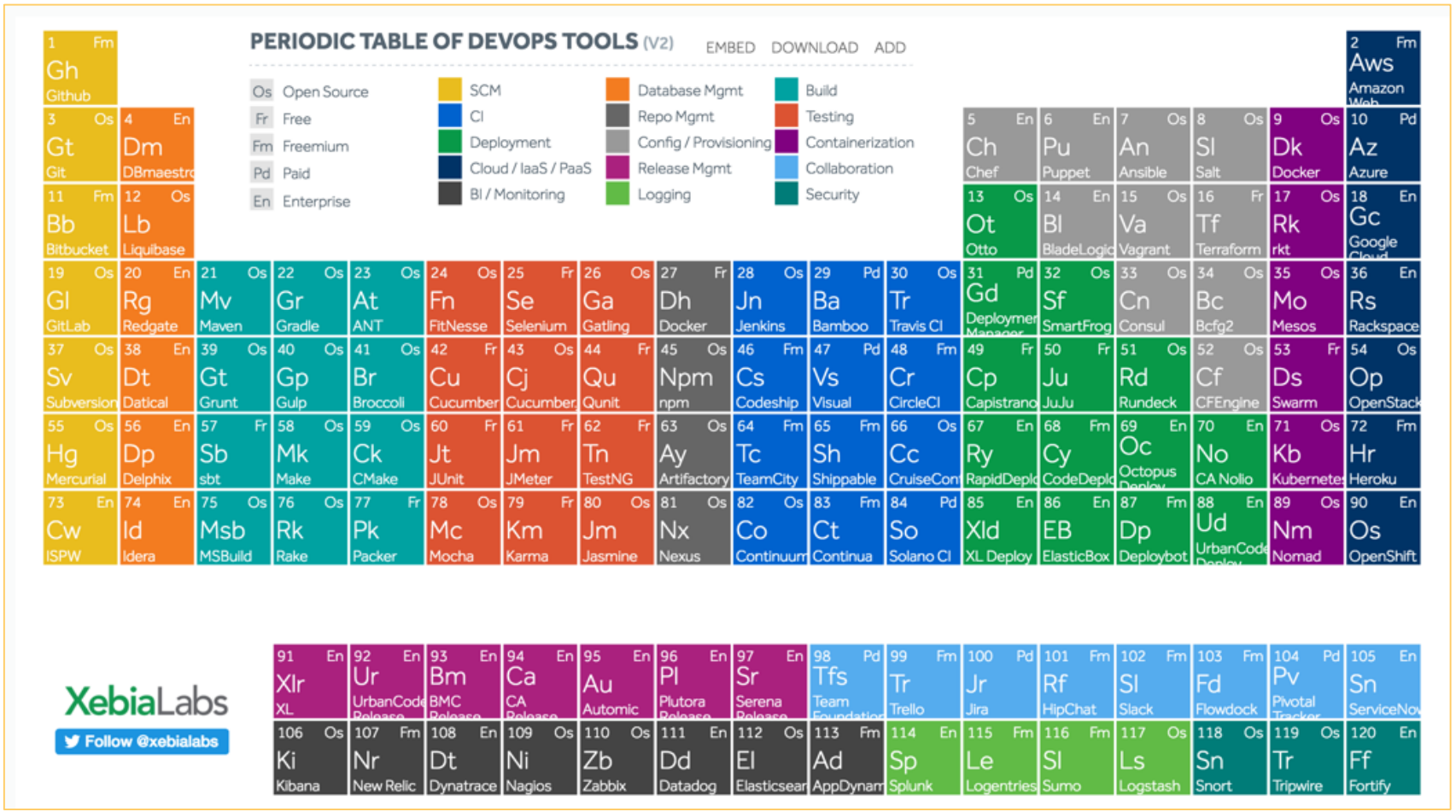
微服务开发模式总体方案



基于微服务架构的持续规划、并行流水线开发和持续发布



选择合适的工具链



来源：XebiaLabs，15类，120种工具

- ✓ 开源/商业？
- ✓ 哪个工具？
- ✓ 工具集成？
- ✓ 部署与维护？

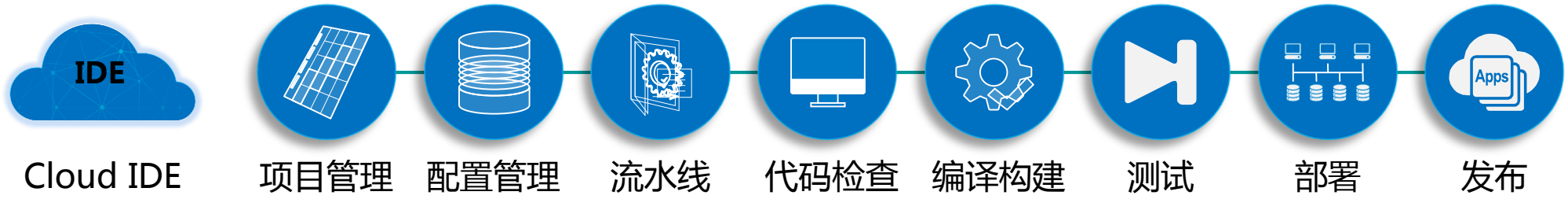
华为软件开发云

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

Web开发

微服务开发

移动应用开发



Cloud IDE　项目管理　配置管理　流水线　代码检查　编译构建　测试　部署　发布

谢谢

微服务化架构判断标准

名称	微服务化架构	非微服务化架构
服务独立性	服务可独立发布、安装、运行	只支持产品整体打包，无法支持单个服务发布、安装、及运行
接口是服务与外界联系的唯一方式	除接口之外的资源，都应该由服务自身提供，服务提供者不能依赖于服务消费者。	服务A包含服务B（B作为服务提供者，依赖于A进行部署）
服务调用方式	服务的调用只能通过协议接口，建议采用REST方式	使用直接链结程序、直接读取其他团队的数据库、使用编程语言的函数调用机制、使用进程间通信、使用共享内存模式、使用别人模块的后门，以上任何一种调用方式都不是服务化架构；
服务可以在运行期复用	可在运行期直接部署服务，通过接口调用来实现复用。	只能通过源代码复用； 只能通过在产品中重新编译、链接来复用； 只能通过系统停机，将服务重新打包来复用
服务间无启动顺序依赖	服务之间没有启动顺序关系；即所依赖的服务未启动时，本服务可以启动起来但不工作	服务必须按依赖关系顺序启动
服务的规模	每个服务有一个设计、开发、测试全功能团队完成，团队规模小于10人	一个开发单元的规模很大，往往几十上百人
产品与服务的关系	产品的主要功能是有一系列的服务组成	一个产品如果只提供服务接口，而不是由多个服务组成，不是服务化架构； 一个产品如果只提供个别服务，也不是服务化架构；
服务注册、发现机制	服务可以动态注册，被消费者动态发现，减少服务的人工管理工作量。	内部模块间的联系主要依靠开发态的编译链接或配置。
服务Governance机制	服务提供自身的正式的接口描述，并通过服务库发布	不提供正式的接口描述，没有接口管理机制