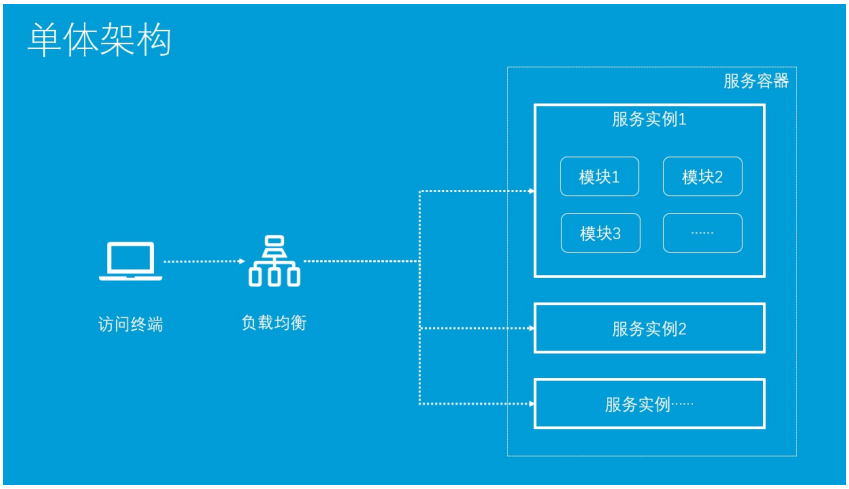
微服务架构设计

微服务（Microservices）一词可追述到2005年[^1]，它并不是一个新鲜的概念。现在很多项目都在谈微服务，但真正成功实施微服务的项目少之又少，这其中有很多对微服务的误解与误用。网络上介绍微服务的文章可谓汗牛充栋，本文尽量避免重复造轮子，在保持结构完整的同时结合笔者的经历突出重点，抛砖引玉，浅谈下微服务的架构设计要明确的问题及正确姿势。

服务架构演绎

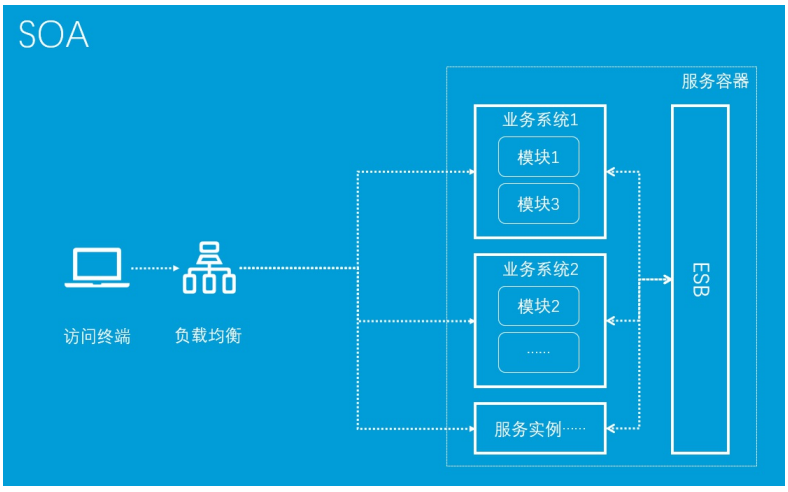
谈到服务架构最简单的当属 单体架构，如下图，一个实例中集成了一个系统的所有功能。



其优点是开发、调试、部署简单，非常适用于小型系统或是项目POC阶段，它的问题也很明显：

1. 扩展性差，一个功能点的变更需要整体部署
2. 无法实现复杂业务，一个容器中实现所功能，服务耦合性高
3. 技术升级困难，牵一发而动全身，无法模块化地实现技术框架的升级
4. 开发效率低，达到一定代码量后编译、部署耗时高，每个成员都需要有完整的环境依赖
5. 不利于安全管理，所有开发都拥有全量代码

于是，出现了 SOA架构，它对 单体架构 做了水平或垂直的拆分，如下图：



SOA架构规避了单体架构的问题，实现了业务与技术的解耦，各系统彼此独立，多通过ESB协调系统间的调度，但它也存在一些问题：

1. SOA拆分的粒度较大， SOA一般按业务域划分系统，但很少涉及系统内细粒度的拆分
2. SOA多需要集中的调度总线，容易产生性能瓶颈

当服务上规模后以上问题会被放大，变得不可忽视，那么我们究竟需要一个怎样的架构呢？

对于大中型系统而言，往往会关注这几个方面：

1. 高度服务化，可扩展：大中型系统必须分层，把各个非业务性的、通用化的功能独立成服务对上提供能力输出，一个理想的系统应该是业务规模与服务的重用度成正比
2. 削峰填谷，易伸缩：大中型系统的硬件成本是个不可忽视的因素，如何实现恰到好处地利用系统资源是架构好差的一个关键指标
3. 小版本，快迭代：系统规模大了，如何避免牵一发而动全身，如何实现对业务的快速响应至关重要
4. 完全分布式，高性能：性能是绕不过去的话题，架构上必须有全盘性的考量，同时也必须避免单点的存在
5. 安全可靠，可维护：架构需要对运维提供友好的支撑，在安全、可维护上做足功夫

要实现这几点并不简单，传统的架构只能覆盖某几点，而微服务架构在很大层面上可以满足这些需求或是为其实现提供了可能。

微服务概览

> Microservices is a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services should be fine-grained and the protocols should be lightweight.
>
> https://en.wikipedia.org/wiki/Microservices

这是维基百科对微服务的介绍，很到位，综合而言：

1. 微服务是SOA思想的延续及发展
2. 相比SOA，微服务要求以服务为单位，拆分的粒度更细
3. 微服务更分布式，去中心化，不需要ESB
4. 微服务假定各服务的技术栈可能不同，推荐通用化的HTTP协议交互
5. 各服务可自运行，无需容器

相较SOA架构，它的优势有：

1. 业务响应快：按细粒度业务单元拆分，新增或业务变更只需要修改小部分服务即可提测、发布
2. 代码复用率高：更小的粒度意味着更多的可复用性，避免重复造轮子
3. 可靠性高：去中心化、集群化，降低了单点故障及性能瓶颈
4. 硬件投入少：高峰期自动扩容、低谷时自动缩减，可实现对资源恰到好处的利用
5. 开发成本低：支持不同技术栈服务，开发人员可使用自己擅长的技术，不同服务场景可以选择最合适的技术实现

当然这些优势的前提是基于与业务、团队适配的微服务架构，即架构层面要完善、合理。

微服务的挑战

微服务不是万精油，要实践之前必须明确它的问题，本文会花比较多的篇幅讨论主要的几个挑战。

服务划分

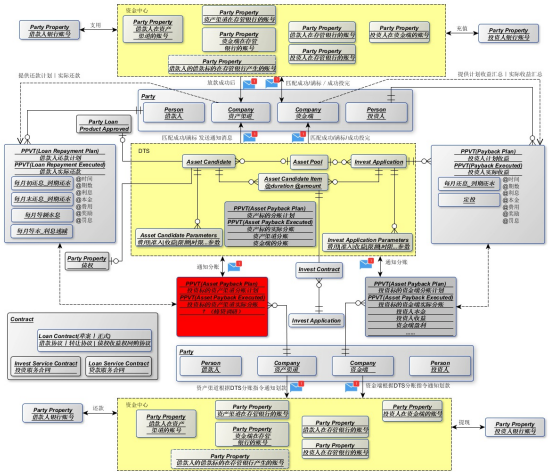
服务的合理划分是微服务成功的重中之重，是所有项目实施之前必须认真思考，严肃对待的。那么怎样划分才算是合理呢？

以业务、技术、团队导向规划服务

我们必须明确的是：服务不是越细越好，服务划分的第一要素是先以业务域水平拆分，再以技术视角垂直拆分，结合团队的规模、能力确定

服务的个数与边界。这之中技术上的垂直拆分比较容易实现，但业务域的拆分可能会存在模糊边界，这时可以考虑结合领域模型来划分业务域。

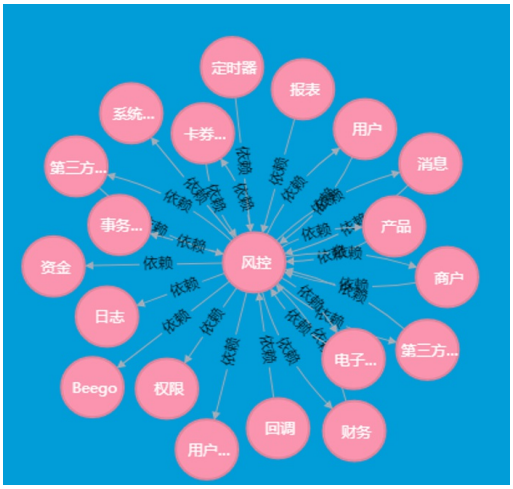
以笔者的经验，在复杂的系统架构中领域驱动设计（DDD）中的领域模型是最好的业务导向的服务划分方案，下图是笔者服务公司的领域模型示例，可以很直观地明确划分边界。关于DDD可参见^Jdon网站。



另外团队的整体能力也是要考虑一个重要因素，一般而言团队的整体能力与服务的数量成正比，反之极容易导致架构失控。

DAG检查

DAG在数学是有向无环图，这里借用此定义表明：服务间不应该存在双向依赖，双向依赖会导致灾难性的后果。



上图是某项目改造前的服务依赖，存在很严重的双向依赖，直接导致了无法自动化部署、需求响应慢等诸多问题。一个设计精良的系统会做服务分层，业务在上，基础能力在下，上层调用下层服务，下层不依赖于上层。

分布式事务检查

分布式事务的成本很高，服务拆分尽量避免产生跨服务事务，能合则合。如实现无法合并优先考虑TCC或基于MQ的柔性事务，尽可能规避2PC等对性能影响很大的事务方案。TCC可完全替换2PC，但开发成本偏高，需要调用各方都同步修改以支持Try、Confirm和Cancel操作，某些场景会调用三方服务，其代码不受我们控制，此时可以考虑使用MQ实现异步消息和补偿性事务。

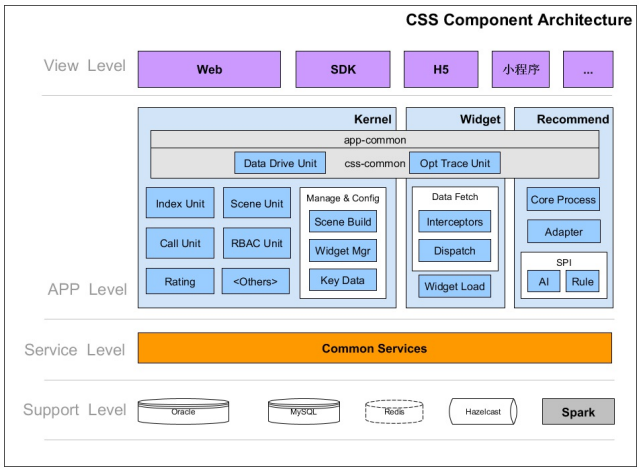
性能分布检查

服务中对于特别耗资源的操作应尽量独立。比如笔者的团队中有项目使用了bcrypt，导致系统注册服务的TPS严重下降，这时就应该考虑把这个加密独立成服务，为这一服务部署更多节点，并且可以为其独立购买计算优化型云主机。

稳定（易变）性检查

一个服务中如存在稳定和不稳定的模块，应该将两者拆分。

比如笔者曾经为某电信10000号做服务化改造，10000号子系统——客服支撑系统(Customer Support System)有很多诸如本月套餐使用情况、销售品、IVR信息、通话足迹、故障分析等功能块，各个功能块要求可以动态添加、修改，此时在设计会把其独立成Widget服务，它的变更不会影响kernel服务。



调用链检查

服务间调用有IO消耗并不宜追踪，应控制调用链路的长度。以笔者的经验，一般的项目在4层以内比较合适，比如（应用服务网关——>应用服务——>公共服务网关——>公共服务）。

服务的划分是微服务成败的关键，实践时遵循上述原则的同时更要灵活机动，因地制宜。

接口调用方式

接口定义上应该明确调用的方式，常见的有RPC和Rest两类：

> PRC(Remote procedure call): 以本地方法调用的形式处理远程方法调用的模型。常用的有：SOAP、RMI、Thrift、Avro、Dubbo协议
>
> Rest(Representational state transfer)：以资源为中心，描述资源状态变更的模型。常见于HTTP协议

RPC的优点有：

1. 可定制协议/传输类型，可实现高性能通讯
2. 可用于强格式约束场景

RPC的缺点有：

1. 各调用方多有语言约束
2. 字段变更各调用方多需要重新部署
3. 防火墙不友好

Rest的优点有：

1. 通用性高，无语言约束
2. 弱格式约束，字段变更不需要所有调用方都重新部署
3. 防火墙友好

Rest的缺点有：

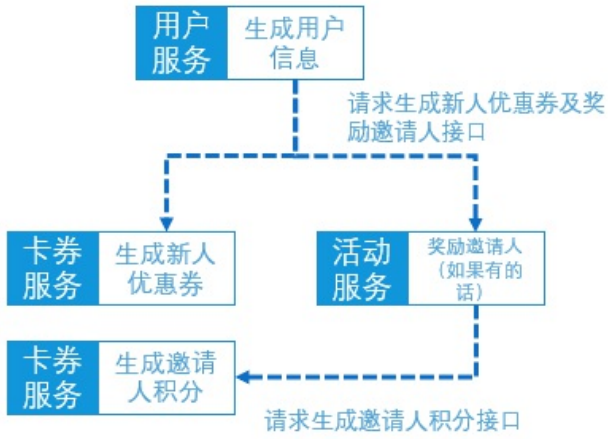
1. 传输效率没有特定的RPC框架高

以微服务架构的初衷看更偏向于使用Rest以方便实现异构系统的通讯。笔者对比Rest的代表Spring Cloud与RPC的代表Dubbo，性能上前者比后者慢1-2倍左右，但对于实际业务调用场景而言，通讯上的这些损失是可以接受的，并且我们还可以用缓存、压缩等方式进一步减少两者的差距。

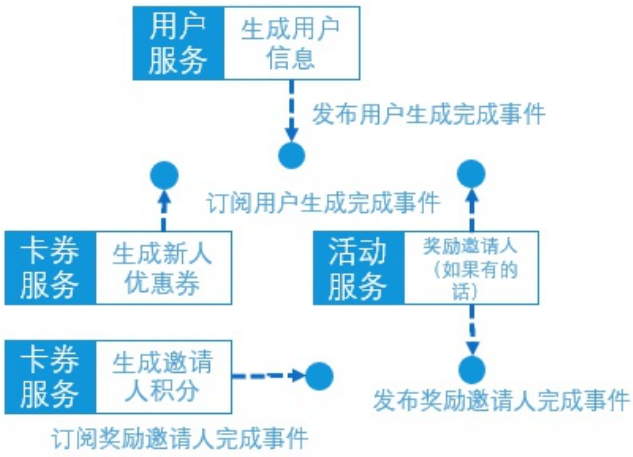
接口定义上还应该明确同步还是异步。同步方式——请求/响应模型，理解直观，使用简单，多用于非耗时方法调用，异步方式——基于事件模型，性能高，服务解耦，多用于耗时方法调用，异步也可以通过回调实现请求/响应模型，但操作上会比较复杂。

服务编排与协同

服务编排(Orchestration)与协同(Choreography)在SOA就存在，编排多指有统一的中心节点负责服务调用处理，协同多指无中心化调度，调用以消息形式传递。以用户注册流程为例：



上图是服务编排的处理方式，可见用户服务要感知卡券服务及活动服务的存在并与它们有直接的交互，活动服务也要感知卡券服务，这导致责任链不明确。



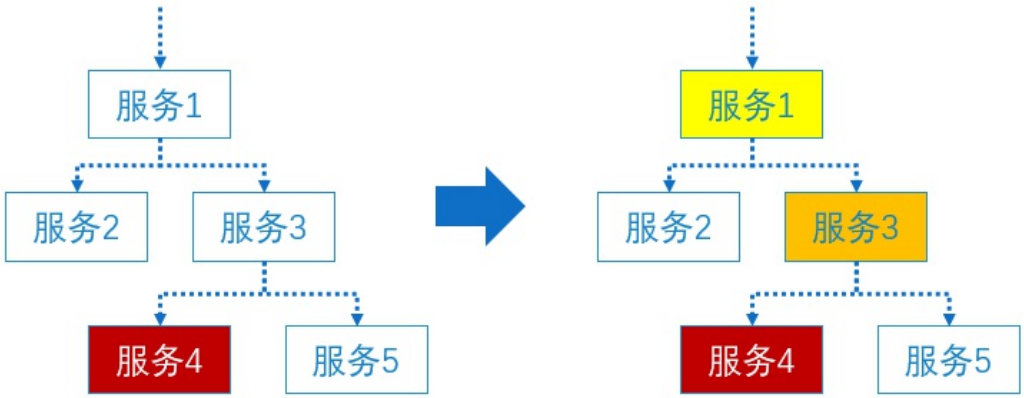
上图是服务协同，所有调用都是事件驱动，由消息触发，各司其职，所有服务都完全解耦。

服务编排实现简单，但存在服务耦合，同步调用影响性能等问题，服务协同解决了上述问题，但需要额外的监控处理，不大适用一致性要求高的场景及请求/响应模型。

从微服务所面对的场景分析，服务协同或是事件驱动（EDA）更为合适，但实现的难度过大，纵观现状还是以服务编排为主。

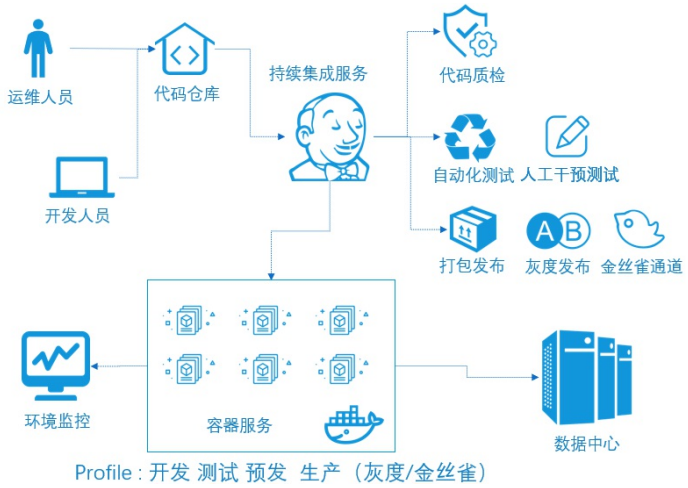
服务监控

实施微服务有一句深入人心的话：无监控不服务。微服务由众多独立的服务组成并且服务间调用频繁，必须要有强大的监控系统才能确保服务的可靠性。监控要全方位的，从服务调用、JVM到系统、网络、数据库、中间件等缺一不可。谈到服务监控有必要说一下断路保护，这也是必须考虑的，断路保护可以是简单暴力的超时判定，也可以引入Hystrix等实现智能保护。下图是缺失断路保护引发的服务雪崩，关于雪崩有很多文章，此处不赘述：



服务部署

微服务带来的服务数量提升及快速响应要求直接导致了运维方式的升级，成功的微服务架构必须有一套成熟的持续部署（CD）流程。



如上图是一个CD流程的示例，对于一个成熟的微服务可能有几十个组件，每个组件做HA，最终可能有上百个实例，同时还要考虑开发、测试、UAT/预发、生产、灰度等环境，人工部署将是灾难性的。

Docker、K8S等技术的兴起为微服务提供了很好的运维支撑，可以很方便地实现服务规模的伸缩。

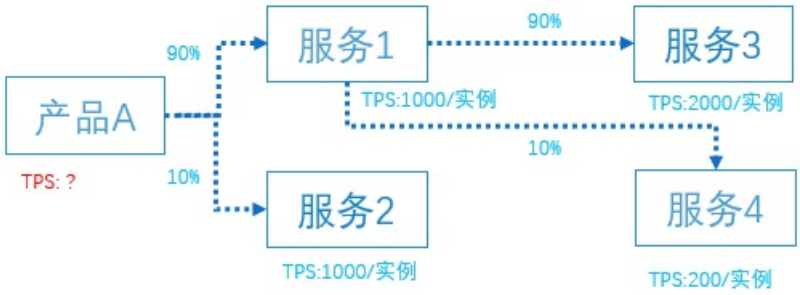
可测试性

由于存在很多服务间的调用，单元测试可能会需求更多的Mock场景，性能/可靠性测试需要更多的关注网络、服务升降级、分布式事务/锁等因素，给测试带来很多挑战。

对于微服务的测试可以尝试使用CDCT(消费者驱动契约测试)，本文不对此展开讨论，CDCT的内容可参见^CDCT。

SLO/SLI设定

单服务SLO设定比较简单，但多个服务叠加后的系统整体SLO设定就比较困难了。



如上图，我们不能简单地汇总每个服务的SLO来确定整体SLO，并不是取平均/最低值（木桶理论不适用），需要结合业务评估。

其它

除了上述要考虑的问题外，微服务实施还需要明确诸如以下内容：

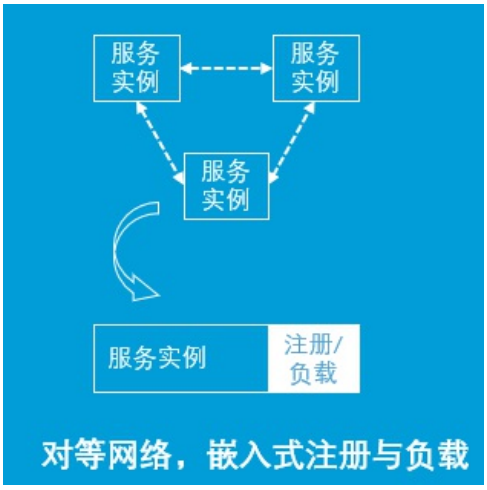
1. 幂等处理：微服务设计以假定网络不稳定为前提，要求服务接口考虑幂等，常用的幂等处理可由数据库、分布式缓存实现，对于海量数据处理需要考虑使用Bloom Filter以减少幂等处理的数据体量
2. 时序处理：对于一些顺序敏感的服务需要有一种统一的机制处理顺序，实现上可以使用MQ来保证顺序
3. 分布式锁：程序上应该尽量避免使用锁，分布式环境更甚，可考虑使用MQ、分桶等方案规避，如必须使用务必确保设置超时时间
4. CP与AP的取舍：服务要明确可否接受数据有一定时间的不同步，这会涉及到诸如数据库HBase（CP）与Cassandra（AP），分布式服务Zookeeper（CP）与Eureka（AP）选型
5. Leader选举（共识）：我们有时会需要Master-Slave的服务架构，这时需要考虑如何选举，如何处理脑裂问题

上述每个问题都可以展开讨论，每个问题都有适用于不同业务场景的解决方案，微服务架构并不能人云亦云、亦步亦趋，选择之前请三思。

微服务的核心技术

服务发现与调用

服务注册发现及调用是微服务最核心的技术，主要关注两点：服务注册表、负载策略，实现有三种方案。



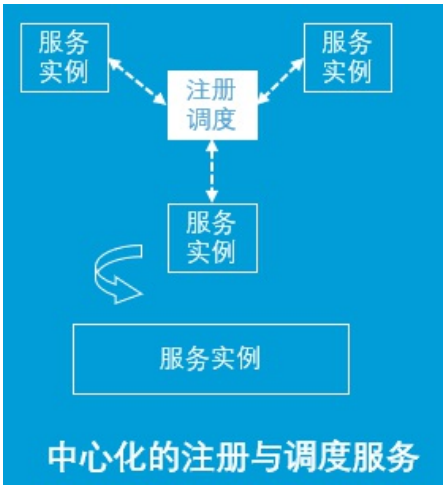
对等网络，嵌入式注册与负载

优点：

1. 服务自治，完全对等，无需额外服务
2. 服务调用直连，速度快

缺点：

1. 对服务有侵入，不存粹
2. 需要为不同语言开发完整的注册负载实现
3. 服务数量多时不利于管控



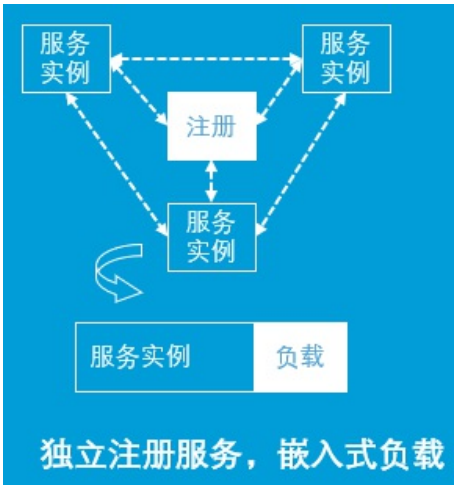
中心化的注册与调度服务

优点:

1. 服务无侵入
2. 中心化服务，方便管控
3. 利于服务编排

缺点:

1. 类似ESB，每次服务调用都要走总线，性能差



独立注册服务，嵌入式负载

优点:

1. 中心化注册服务，方便管控
2. 服务调用直连，速度快

缺点:

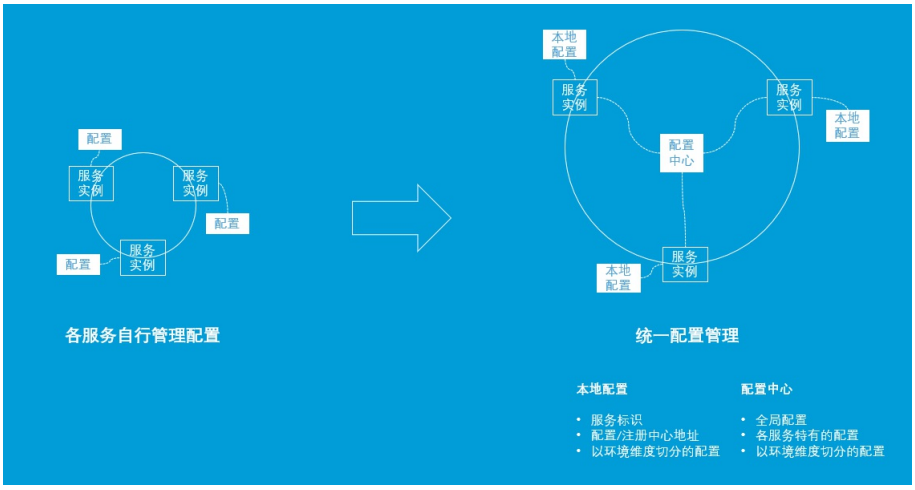
1. 服务要保存本地注册表执行负载策略
2. 有一定侵入，语言相关

在实际使用中笔者倾向于使用独立注册服务，嵌入式负载，这也是Spring Cloud的方案。

统一配置管理

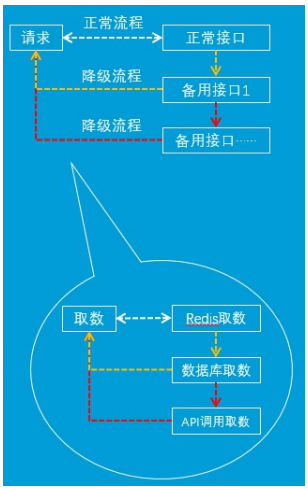
没有统一配置管理，对于微服务而言是无法接受的，统一配置管理需要解决的问题有：

1. 集中化的配置管理，通用配置只存在一份，一改全改
2. 动态更新，对于一些简单的配置可以不重启刷新，且实现分批次刷新、失败回滚处理等功能

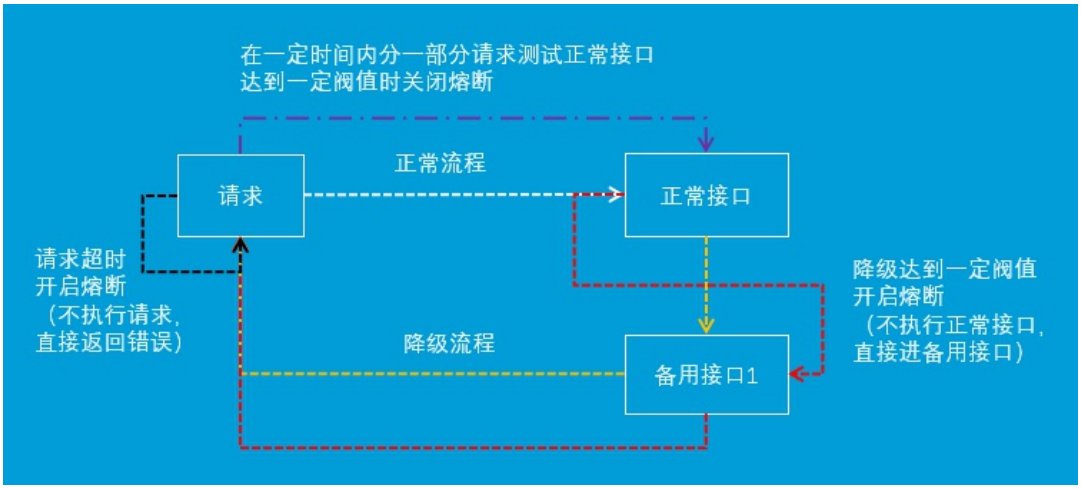


链路保护与降级

上文有讲到过断路保护，智能链路保护与降级是微服务的核心技术之一。

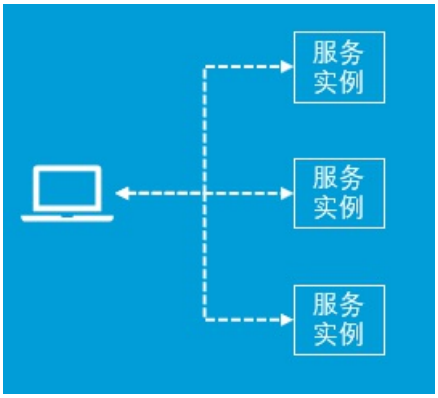


如上图，可以为一些核心（高可用）接口定义多个备用接口，例如数据获取时正常流程使用Redis取数，失败后走数据库，如数据库也失败则从原始API中（可能是第三方接口）获取数据。



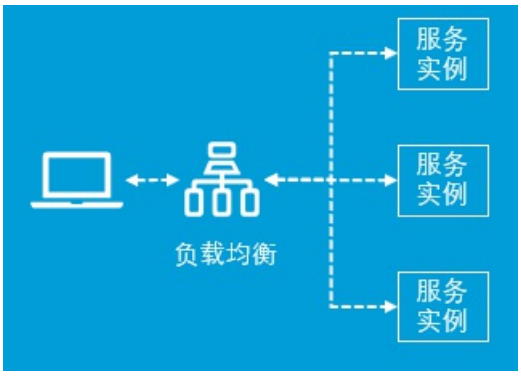
所谓智能体现在失败与重试的策略上，可以确保在满足一定阀值时自动恢复。

服务网关



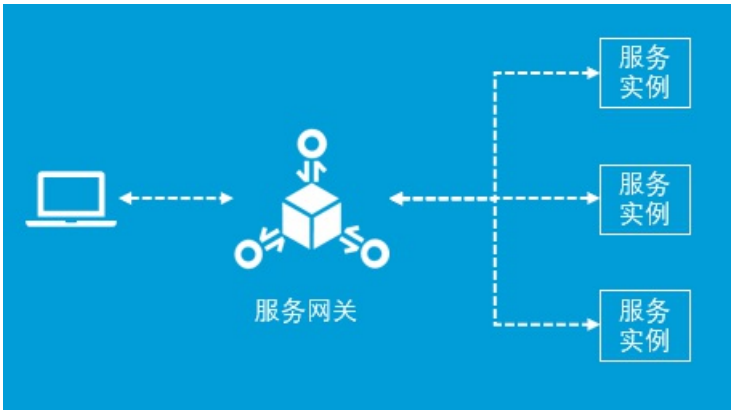
在没有服务网关时，所有服务都对外暴露，这导致的问题有：

1. 对外暴露了服务地址，不安全
2. 终端要设置的地址太多了，违反迪米特法则
3. 缺少负载均衡功能，稳定性是个问题



我们可以使用反向代理做网关，如Nginx，但它也有一定的问题：

1. 只实现了负载均衡，功能单一
2. 服务动态感知能力弱



所以微服务一般需要独立的服务网关服务，以实现如下功能：

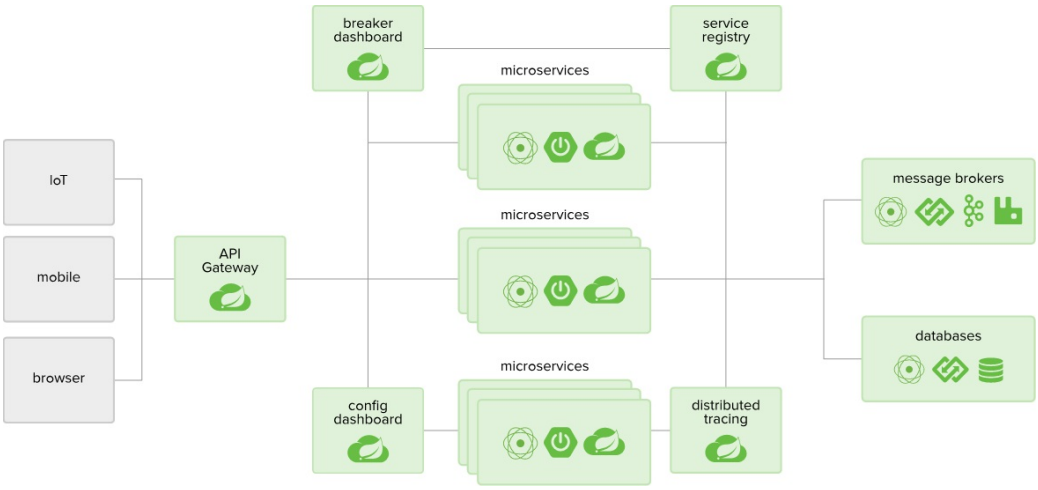
1. 对外隐藏服务细节，提供统一的接口
2. 服务级别的负载均衡，与服务注册中心配合实现高效的动态感知
3. 可添加日志、权限等功能，实现统一拦截处理

服务网关是服务的门面，对性能有很高的要求，网关的处理逻辑要求简单快捷，Spring Cloud有Zuul网关，但其性能并不适用高并发场景（特指Zuul 1.x，2.x基于事件驱动，尚未发布），笔者会在网关中实现统一的鉴权处理，这一功能使用Vertx实现，鉴权只与Redis有一次交互，以实现单机3w多的TPS（阿里云4核8G，系统未调优）处理能力。

微服务框架介绍

Spring Cloud http://projects.spring.io/spring-cloud/

Spring Cloud无疑是微服务的集大成者，它拥有完整的微服务生态，并且以Spring之名一举成为微服务的后起之秀。下图是Sping Cloud的生态组件：



Dubbo http://dubbo.io/

Dubbo本身不算严格的微服务，它只实现了服务调用及治理，但由于国内用户众多，此处还是有必要提及，使用Dubbo还需要引用其它的技术框架以实现服务网关、统一配置、HTTP服务等。

Vert.x http://vertx.io/

这是笔者最推崇的微服务框架，它有完整的组件生态，支持主流JVM语言（Java、JavaScript、Groovy、JRuby、Scala等），并且它是Reactive的，完全异步化、事件驱动。笔者使用此框架近四年，对其可靠性、高性能有着深刻的印象。

lagom https://www.lagomframework.com/

这是微服务的新秀，它有与Vert.x相同的Reactive特性，同时更有Event Sourcing和CQRS支持，通讯是使用Akka，开发上基于Play!。就技术而言，笔者挺看好它，但从社区、市场上看只能成为一个小众框架。

如何迁移到微服务架构

TBD

展望未来

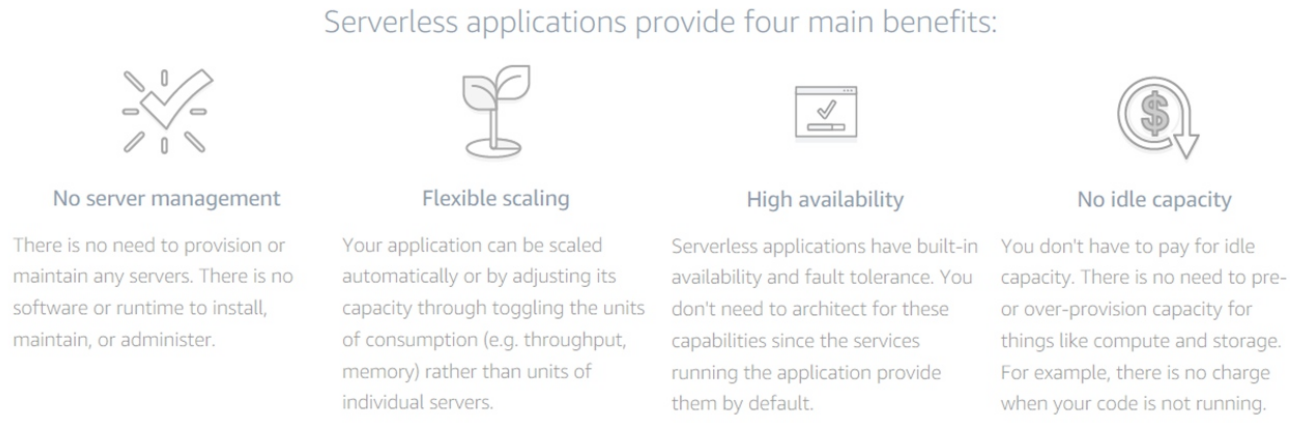
笔者认为微服务有如下几个重大问题：

设计视角 服务拆分最关键，但其粒度及分层很难把握 **部署视角** 严重依赖支撑服务（注册与发现、统一配置、网关等） **运维视角** 服务众多监控困难，服务伸缩架构复杂

如果有一种方式，它可以……：

1. 简化服务拆分，仅需要聚焦业务维度
2. 不用自己运维支撑服务且可以确保这些服务稳定
3. 有强大的服务监控功能
4. 不用关心服务地伸缩，解决方案中天然支持

有吗？目前看得到的无服务器架构(Serverless)也许可以很好地实现上述目标，解决微服务中的问题。



对于Serverless各大云厂商也都在逐步跟进：

> https://aws.amazon.com/serverless/
> https://azure.microsoft.com/en-us/services/functions/
> https://www.aliyun.com/product/fc
> https://cloud.tencent.com/product/scf

Serverless目前缺少统一的标准，各个厂商都有自己的规范，这严重限制了其成长，有一个项目：https://serverless.com/ 试图抽象统一的规范来适配不同的厂商，但这治标不治本。期待看到统一的Serverless specification。

另外，服务化道路上另一个新的方向是Service Mesh，其代表Istio是Google/IBM/Lyft联合开发的开源项目（https://github.com/istio/istio），截至本文撰写时已有3443次Commits 5869个Stars，社区很活跃。这一方向很值得关注。

作者：蒋震宇 2018年2月13日