1. 引言

1.1 微服务研究的背景和意义

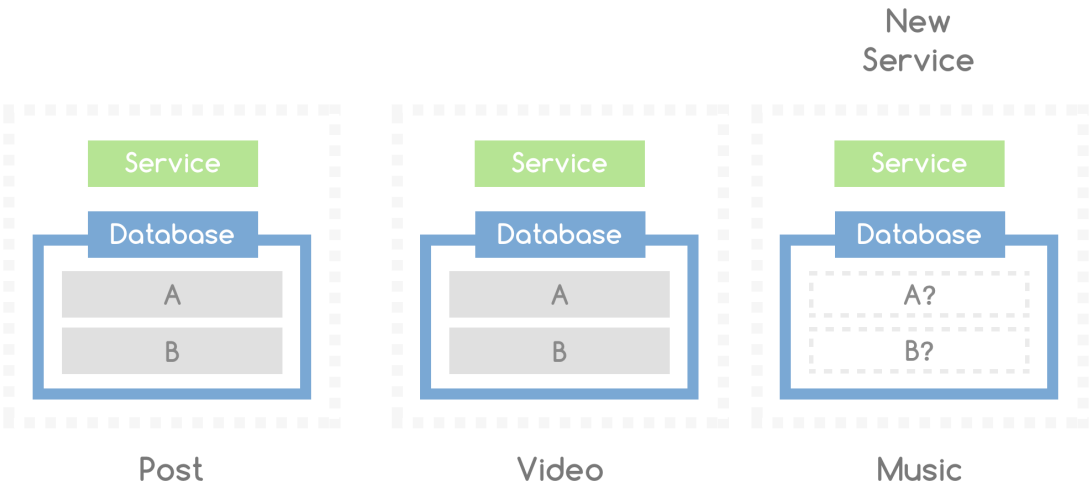
系统架构设计是系统构建过程中的非常关键的一部分，决定着系统的稳定性、鲁棒性、扩展性等一系列问题，定义了在系统内部，如何根据技术、业务和组织及可扩展性、可维护性和灵活性等一系列因素，把系统划分成不同的组成部件，并使这些部件相互协作为用户提供某种特定的服务。

不过伴随着业务的不断发展，功能的持续增加，传统单块架构对应的沟通、管理、协调等成本越来越高，出现了维护成本增加，交付周期长，新人培养久，技术选型成本高，水平和垂直扩展性差，组建全功能团队难等一系列问题。为解决传统系统架构面临的问题，随着领域驱动设计，持续交付技术，虚拟化技术，小型自治团队组建，基础设施智能化自动化，大型集群系统设计等技术和实践的发展，微服务应运而生。

微服务作为一种分布式系统解决方案，聚焦细粒度服务使用的推动。不同的微服务协同工作，每个服务都有自己的生命周期。由于微服务主要围绕业务领域构建模型，而且整合了过去十多年来的新概念和新技术，因此可以避免由传统的分层架构引发的很多问题及陷阱，具有很大的研究价值和实用意义。

1.2 微服务的定义和特点

微服务（英语：Microservices）是一种软体架构风格，它是以专注于单一责任与功能的小型功能区块(Small Building Blocks) 为基础，利用模组化的方式组合出复杂的大型应用程式，各功能区块使用与语言无关 (Lang而且复杂的服务背后是使用简单 URI 来开放介面，任何服务，任何细粒都能被开放（exposed）。这个设计在 HP 的实验室被实现，具有改变复杂软体系统的强大力量。



在微服务架构中，有几个关键的特点：

1. **小型化**：每个微服务都非常小，只关注一个具体的业务功能或需求。这种小型化使得每个服务都更加简单和易于理解，从而降低了开发和维护的难度。此外，小型化也意味着你可以更快地开发和测试每个服务，因为每个服务都相对较小。
2. **独立性**：微服务可以被单独开发和部署，不依赖于其他服务。这使得微服务可以由小型团队（例如2至5人）进行开发。在实际的开发场景中，这种独立性意味着每个服务都可以独立地开发、测试和部署，而不需要依赖其他服务。这使得团队可以并行工作，提高了效率，并且更容易扩展和升级。
3. **松耦合**：微服务之间的关联非常松散，无论是开发还是部署，都是相互独立的。这意味着，修改一个服务不会影响其他服务。在实际的开发中，松耦合可以降低风险和复杂度，使得系统更加灵活，并且可以随时添加、删除或修改某个服务，而不会对整个系统产生太大的影响。
4. **技术多样性**：微服务可以使用不同的编程语言和技术栈进行开发。在实际开发中，这使得团队可以选择最适合他们的服务的语言和技术。这还可以帮助你更好地利用现有的技能和资源，因为你不必限制自己只能使用某种特定的语言和技术。
5. **易于理解与维护**：由于每个微服务只负责一项任务，因此它们相对容易理解、修改和维持。在实际开发中，这使得团队可以更快地开发和调试服务，因为他们只需要关注一个特定的功能。此外，由于每个服务都是独立的，所以你可以更容易地找到和修复错误。
6. **融合新技术**：微服务架构允许快速采用和整合新的技术。在实际开发中，这使得你可以更快地响应市场需求和客户反馈。例如，如果你需要添加一个新的支付选项，你可以快速集成一个新的支付API到你的支付服务中。由于每个服务都是独立的，所以你可以更容易地尝试新的技术，因为你只需要在一个服务中试验，而不是整个系统。

2. 微服务的历史和发展

2.1 微服务的起源

在传统架构下，随着产品的发展，软件的迭代，代码间的逻辑会越来越复杂，代码也越来越多，时间久了代码库就会变得非常庞大，以至于在修改逻辑或者增加新功能时想定位在什么地方做修改都很困难。尽管大家都想在巨大的代码库中做到模块化，但实际上模块之间的界限很难确定，而且更不好处理的是，相似功能的代码在代码库中随处可见，使得修改旧逻辑和增加新功能变得更加异常复杂。而且随着组织架构的变大，参与人员的变多，这一问题将会变得更加棘手。

随着领域驱动设计，按需虚拟化，持续交付，小型自治团队，基础设施自动化，大型集群系统等技术和实践的发展以及为解决传统架构的痛点，微服务应运而生。微服务是一种分布式的系统解决方案，着力推动细粒度的服务的使用，不同的微服务协同工作，每个服务都有自己的生命周期。由于微服务主要围绕业务领域构建模型，所以可以避免由传统的分层架构引发的很多问题。

微服务的由来：

2011年5月，在威尼斯附近的软件架构师小组首次提及了“Microservice”一词，以描述参会者中的许多人在近期探索研究的许多架构风格。

2012年3月，来自ThoughtWorks 的James Lewis 在克拉科夫33rd Degree 会议上的[5]中就此做了相关的案例研究报告，几乎同一时间Fred George也做了与之相同的工作。

来自Netflix的Adrian Cockcroft把这种方法称为“细粒度SOA”，并认为这是一套在Web规模下具有开创意义的架构类型。Joe Walnes，Dan North，Evan Botcher和Graham Tackley也在这篇文章中对此作出了评论。

2012年5月，之前首次提及微服务的软件架构师小组最终决议，以“microservice”为最合适的架构名称。

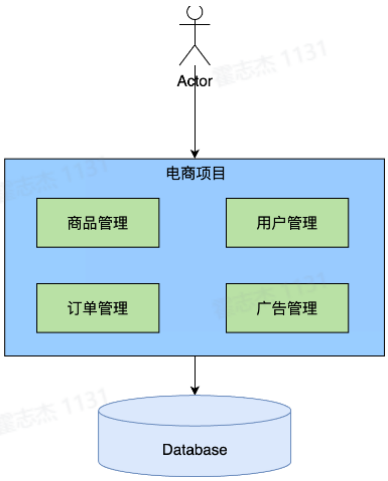
2014年，Martin Fowler 和James Lewis 共同提出微服务的概念。在 Martin Fowler的博客中，他将微服务的定义概括如下：简而言之，微服务架构是将单个应用程序开发为一组小型服务的方法，每个应用程序运行在自己的进程中，并通过轻量级的通讯机制进行通信，通常是基于HTTP资源的API。这些服务是围绕业务功能构建的，并可以通过全自动部署机制独立部署。这些服务应该尽可能少的采用集中式管理，并根据所需，使用不同的编程语言和数据存储。

2.2 微服务的发展历程

2.2.1单体架构

在企业发展的初期，一般公司的网站流量都比较小，只需要一个应用，将所有的功能代码打包成一个服务，部署到服务器上就能支撑公司的业务。这样也能够减少开发、部署和维护的成本。

比如，大家都很熟悉的电商系统，里面涉及的业务主要有：用户管理、商品管理、订单管理、支付管理、库存管理、物流管理等等模块，初期我们会将所有模块写到一个Web项目中，然后统一部署到一个Web服务器中。



优点：

- 架构简单，项目开发和维护成本低。
- 所有项目模块部署到一起，对于小型项目来说，维护方便。

缺点：

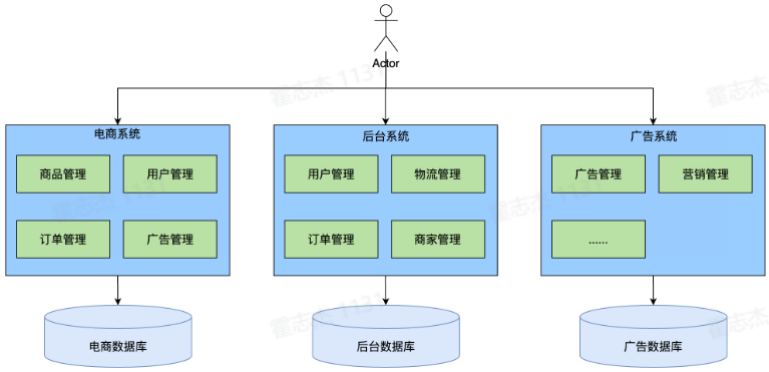
- 所有模块耦合在一起，虽然对于小型项目来说，维护方便。但是，对于大型项目来说，却是不易开发和维护的。
- 项目的各模块之前过于耦合，如果一旦有一个模块出现问题，则整个项目将不可用。
- 无法针对某个具体模块来提升性能。
- 无法对项目进行水平扩展。

2.2.2 垂直应用架构

随着企业业务的不断发展，发现单节点的单体应用不足以支撑业务的发展，于是企业会将单体应用部署多份，分别放

在不同的服务器上。但是，此时会发现不是所有的模块都会有比较大的访问量。如果想针对项目中的某些模块进行优化和性能提升，此时对于单体应用来说，是做不到的。于是乎，垂直应用架构诞生了。垂直应用架构，就是将原来一个项目应用进行拆分，将其拆分为互不想干的几个应用，以此来提升系统的整体性能。

这里，我们同样以电商系统为例，在垂直应用架构下，我们可以将整个电商项目拆分为：电商交易系统、后台管理系统、CMS管理系统等。



我们将单体应用架构拆分为垂直应用架构之后，一旦访问量变大，我们只需要针对访问量大的业务增加服务器节点即可，无需针对整个项目增加服务器节点了。

优点：

- 系统进行了拆分，可根据不同系统的访问情况，有针对性的进行优化。
- 能够实现应用的水平扩展。
- 各系统能够分担整体访问的流量，解决了并发问题。
- 一个系统发生了故障，不应用其他系统的运行情况，提高了整体的容错率。

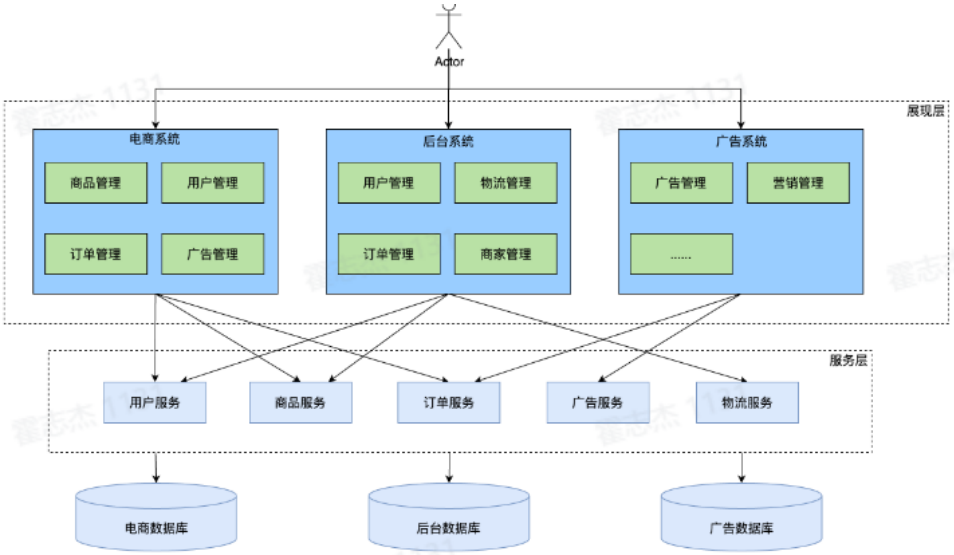
缺点：

- 拆分后的各系统之间相对比较独立，无法进行互相调用。
- 各系统难免存在重叠的业务，会存在重复开发的业务，后期维护比较困难。

2.2.3 分布式架构

我们将系统演变为垂直应用架构之后，当垂直应用越来越多，重复编写的业务代码就会越来越多。此时，我们需要将重复的代码抽象出来，形成统一的服务供其他系统或者业务模块来进行调用。此时，系统就会演变为分布式架构。

在分布式架构中，我们会将系统整体拆分为服务层和表现层。服务层封装了具体的业务逻辑供表现层调用，表现层则负责处理与页面的交互操作。



优点：

- 将重复的业务代码抽象出来，形成公共的访问服务，提高了代码的复用性。
- 可以有针对性的对系统和服务进行性能优化，以提升整体的访问性能。

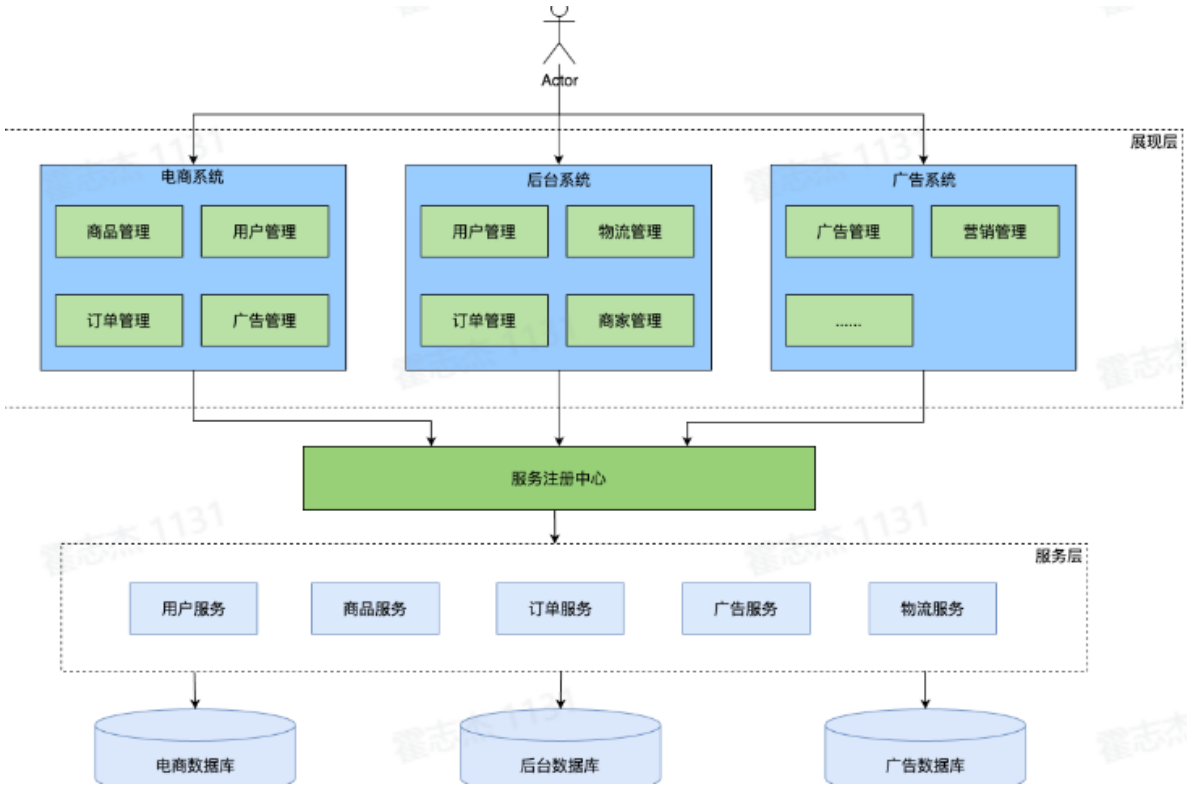
缺点：

- 系统之间的耦合度变高，调用关系变得复杂，难以维护。

2.2.4 SOA架构

SOA架构（Service Oriented Architecture）即面向服务。

在分布式架构下，当部署的服务越来越多，重复的代码就会越来越多，对于容量的评估，小服务资源的浪费等问题比较严重。此时，我们就需要增加一个统一的调度中心来对集群进行实时管理。此时，系统就会演变为SOA（面向服务）的架构。



开始引入“服务注册”的概念

优点：

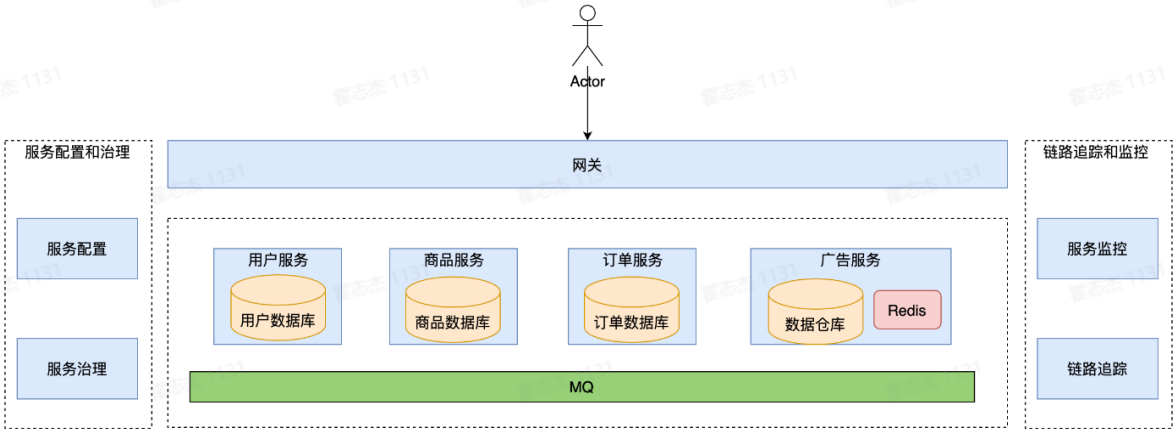
- 使用注册中心解决了各个服务之间的服务依赖和调用关系的自动注册与发现;

缺点：

- 各服务之间存在依赖关系，如果某个服务出现故障可能会造成服务的雪崩;
- 服务之间的依赖与调用关系复杂，测试部署的困难比较大

2.2.5 微服务架构

随着业务的发展，我们在SOA架构的基础上进一步扩展，将其彻底拆分为微服务架构。在微服务架构下，我们将一个大的项目拆分为一个个小的可以独立部署的微服务，每个微服务都有自己的数据库。



优点：

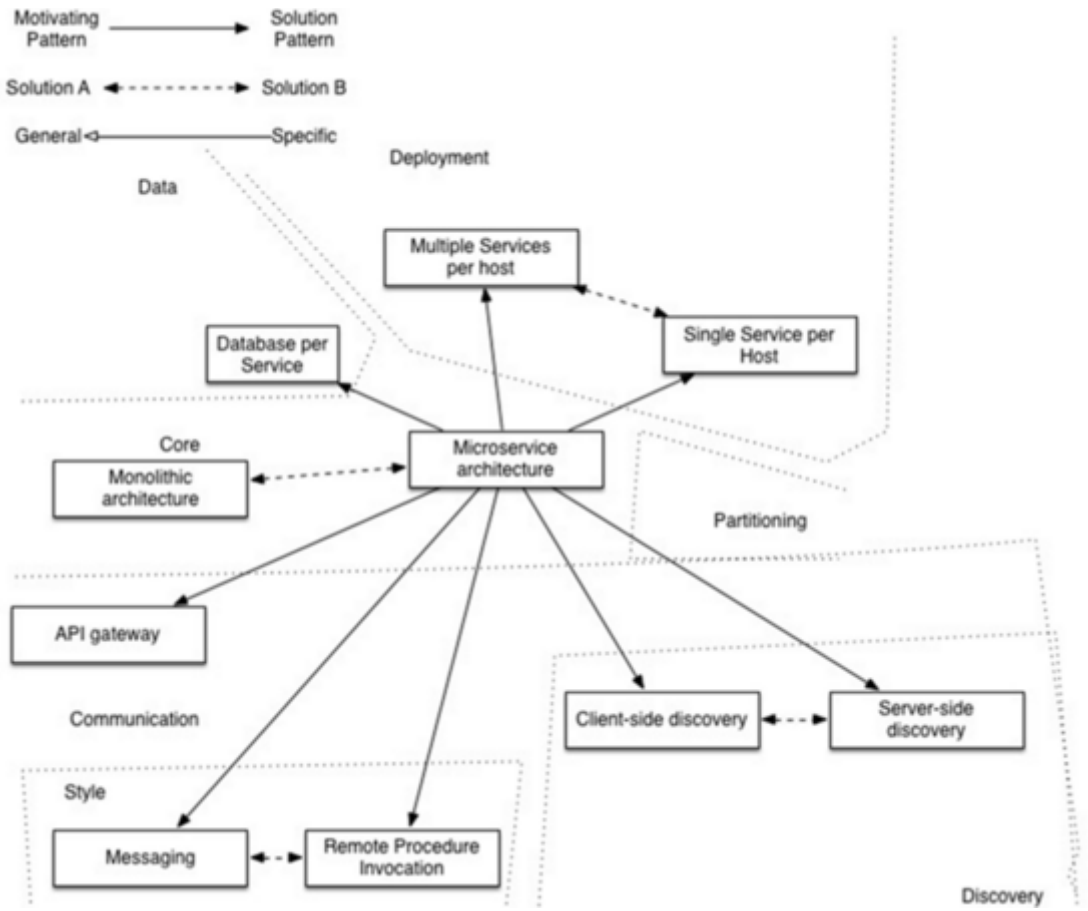
- 服务彻底拆分，各服务独立打包、独立部署和独立升级。
- 每个微服务负责的业务比较清晰，利于后期扩展和维护。
- 微服务之间可以采用REST和RPC协议进行通信。

缺点：

- 治理运维成本增加
- 分布式系统本身复杂性问题

3.微服务架构设计

[微服务技术发展的现状与展望 - 知乎 (zhihu.com)]



如图所示微服务架构设计需要考虑的问题，包括：

1. **API Gateway:**

- 提供统一的入口地址和接口,对外隐藏后端的实现细节和服务实例
- 请求路由,将请求转换为对后端服务的调用
- 访问控制与安全策略,如请求校验、鉴权、访问限流等
- 负载均衡,将请求分配到后端不同服务实例上
- 服务监控与 tracing

2. **服务间调用**

- 通过HTTP/RESTful或RPC等协议在服务之间调用
- 实现服务解耦合,减少耦合度
- 常用技术如HTTP、gRPC、Dubbo等

3. **服务发现**

- 服务注册如Eureka,服务订阅者可实时获取注册服务列表
- 服务自洽注册与自动续约机制
- 负载均衡策略返回服务实例列表

4. **服务容错**

- 服务降级、熔断、限流等策略
- 重试和超时设置
- 服务实例健康监测
- 自动恢复failed实例

5. **服务部署**

- Docker容器化部署,实现快速弹性伸缩
- 多个实例副本,负载均衡

- 蓝绿部署无停机上新版本
- 服务配置外部化

6. **数据调用**

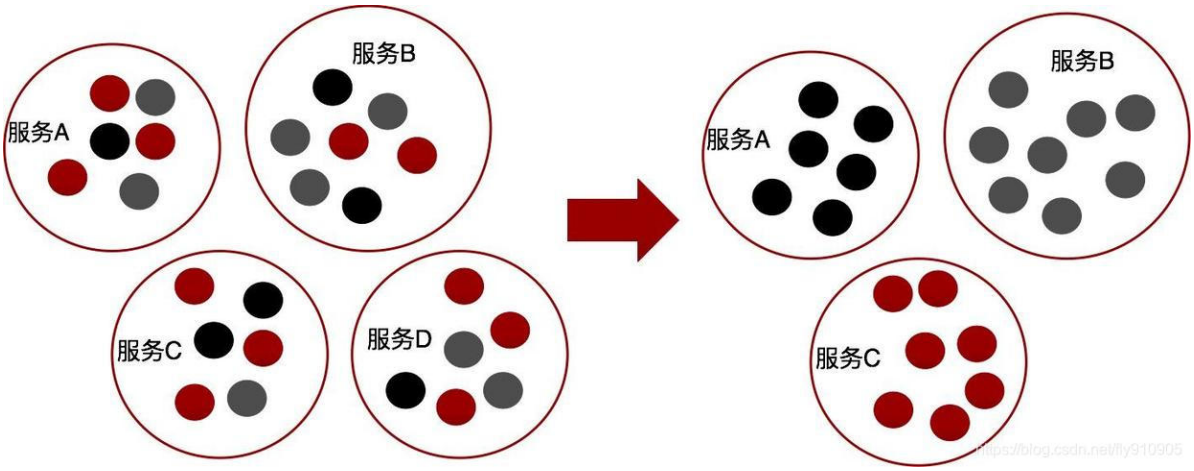
- 本地缓存/分布式缓存Redis等
- 关系数据库对接
- 无关数据库取值与更新
- 数据同步与一致性

以上解释了各组件在微服务体系中的作用与实现原理,希望能更清晰理解微服务架构。请继续提出问题,一起讨论深入。

3.1服务拆分

高内聚低耦合

- 紧密关联的事物应该放在一起，每个服务是针对一个单一职责的业务能力的封装，专注做好一件事情（每次只有一个更改它的理由)。如下图：有四个服务a,b,c,d，但是每个服务职责不单一，a可能在做b的事情，b又在做c的事情，c又同时在做a的事情，通过重新调整，将相关的事物放在一起后，可以减少不必要的服务。
- 轻量级的通信方式
- - 同步RESTful（GET/PUT/POST...），基于http，能让服务间的通信变得标准化并且无状态，关于RESTful API的成熟度，可参`Richardson为REST定义的成熟度模型`
  - 异步（消息队列/发布订阅）
- 避免在服务与服务之间共享数据库



高度自治

- 独立部署运行和扩展
  - 每个服务能够独立被部署并运行在一个进程内
  - 这种运行和部署方式能够赋予系统灵活的代码组织方式和发布节奏，使得快速交付和应对变化成为可能。
- 独立开发和演进
  - 技术选型灵活，不受遗留系统技术栈的约束。
  - 合适的业务问题可以选择合适的技术栈，可以独立的演进
  - 服务与服务之间采取与语言无关的API进行集成
- 独立的团队和自治
  - 团队对服务的整个生命周期负责，工作在独立的上下文中， 谁开发，谁维护。

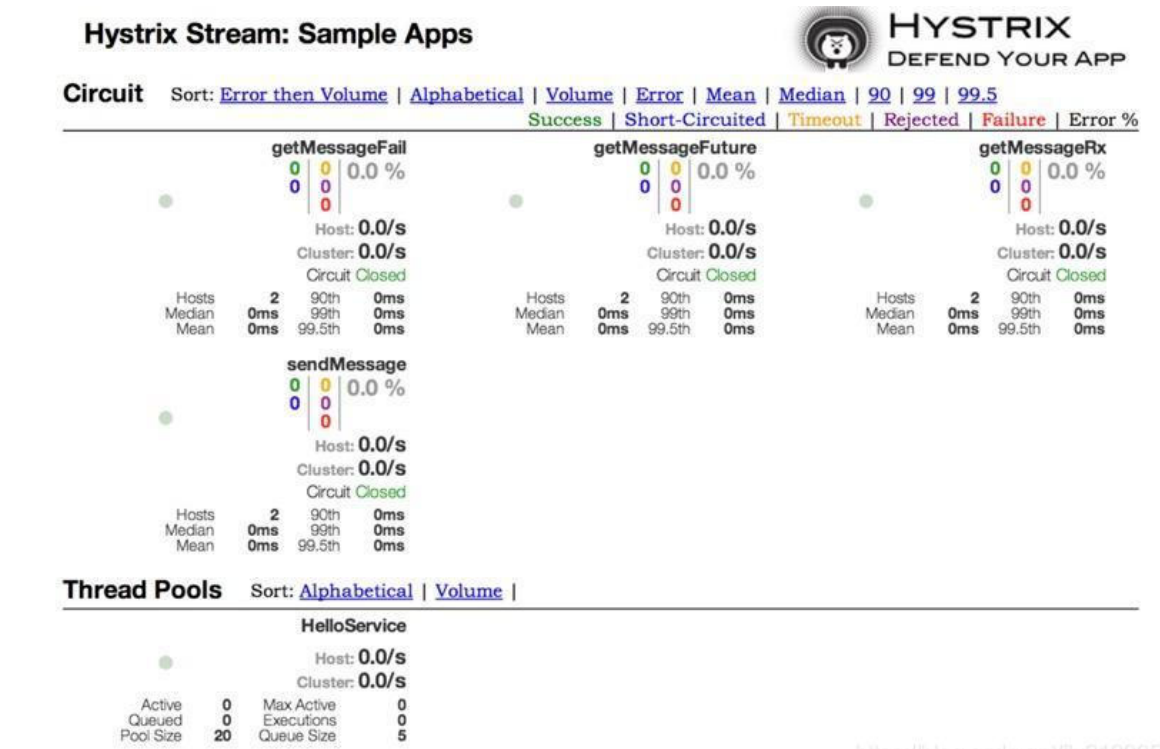
以业务为中心

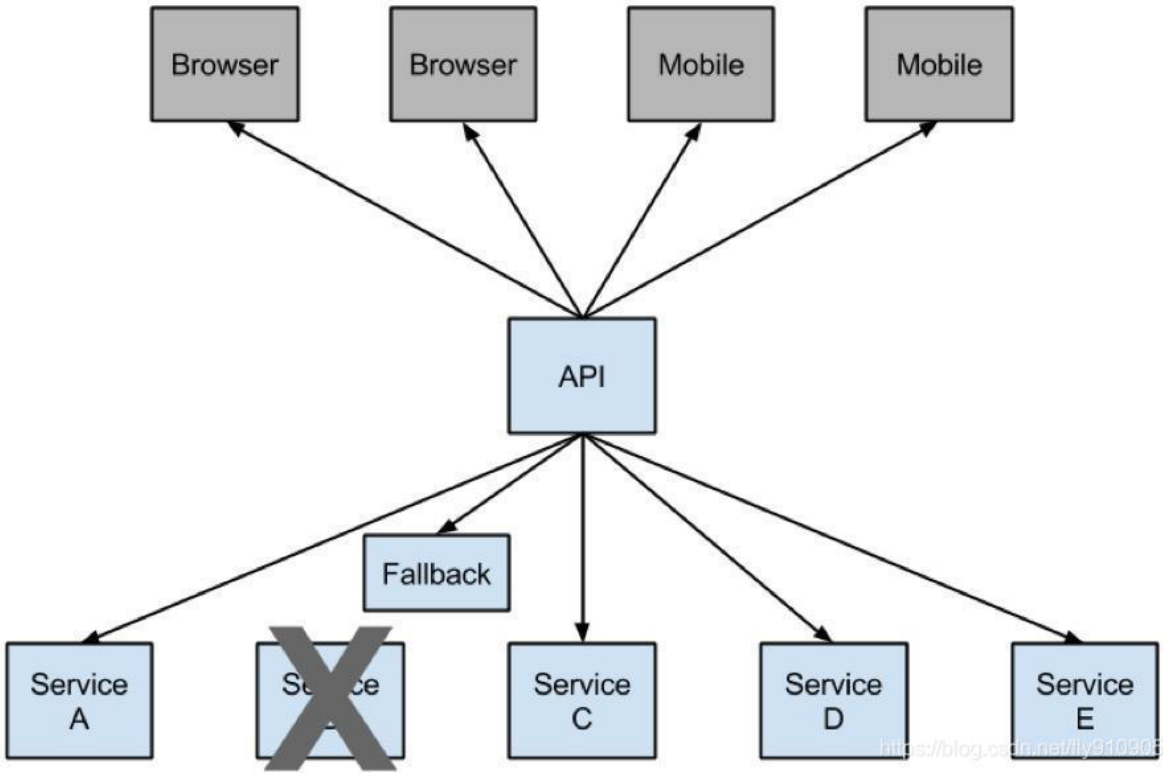
- 每个服务代表了特定的业务逻辑
- 有明显的边界上下文
- 围绕业务组织团队
- 能快速的响应业务的变化
- 隔离实现细节，让业务领域可以被重用

弹性设计

- 设计可容错的系统
  - 拥抱失败，为已知的错误而设计
  - 依赖的服务挂掉
  - 网络连接问题
- 设计具有自我保护能力的系统
  - 服务隔离
  - 服务降级
  - 限制使用资源
  - 防止级联错误

Netfilix 提供了一个比较好的解决方案，具体的应对措施包括：网络超时/限制请求的次数/断路器模式/提供回滚等。





Hystrix记录那些超过预设定的极限值的调用。它实现了circuit break模式，从而避免了无休止的等待无响应的服务。如果一个服务的错误率超过预设值，Hystrix将中断服务，并且在一段时间内所有对该服务的请求会立刻失效。Hystrix可以为请求失败定义一个fallback操作，例如读取缓存或者返回默认值。

3.1.1 领域驱动设计

领域驱动设计主要专注于如何对现实世界的领域进行建模。世界著名软件建模专家Eric Evans在其专著《领域驱动设计——软件核心复杂性应对之道》中引入了新的概念限界上下文（Bounded context）：一个给定的领域都含有多个限界上下文，每个限界上下文中的模型分成两部分，一部分需要与外部通信，另一部分则不需要。每个限界上下文都有明确的接口，该接口决定了暴露哪些模型给其它

的限界上下文[8]。

领域逻辑告诉我们对于模块和服务的划分应该遵循共享特定的模型，而不是共享内部表示这个原则，这样就可以做到松耦合。而松耦合可以保证可以独立的修改及部署单个服务而不需要修改系统的其它部分，这是微服务设计中最重要的一点。除此之外通过共享模型和隐藏模型我们可以更加清楚的识别领域边界，更好地隐藏边界内部实现细节，而边界内部是高相关性的业务，以此做到了高内聚，这样可以避免对多处相关地方做修改和同时发布多个相关微服务而带来的不可控和风险，同时能更快速的交付。

3.2 服务发现

什么是服务发现（Service Discovery），HIGHOPS中的文章《服务发现：六问四专家》中是这么定义的：服务发现跟踪记录大规模分布式系统中的所有服务，使其可以被人们和其它服务发现。

DNS就是一个简单的例子，当然复杂的服务发现需要有更多的功能，比如储存服务的元数据，健康监控，多种查询和实时间更新等。

不同的上下文环境诸如网络网络设备发现，零配置网络（Rendezvous ）发现和 SOA 发现等具有不同的服务发现含义，不过不论是哪一种场景，服务发现都应该提供一种协调机制来发布自己以及在零配置情况下查找其它服务。简单说服务发现就是获取服务地址的服务，复杂的服务提供了多种服务接口和端口，当应用程序需要访问这个服务时，如何确定它的服务地址呢？此时，就需要服务发现了。

3.2.1 三大关键功能

1）高可用：服务元数据存储是服务发现的基础，而数据一致性又是保证服务一致性的关键，而且数据一致性大多依赖分布式算法，同时分布式系统中也要求多数机器可用，所以高可用是必须的功能之一。

2）服务注册：服务实例要想被其它服务知道，必须通过自己或者其它管理组件把服务地址相关元数据存储，同样的当服务地址变化时需要更新，服务停止时需要销毁，这一系列操作也就是服务注册。

3）服务查询：复杂的服务提供了多种服务接口和端口，部署环境也比较复杂，一旦服务组件通过服务注册存储了大量信息后，它就需要提供接口给其它组件或服务进行复杂的查询，比如通过固定的目录获取动态的IP地址等。

3.2.2常用服务发现方式

1）DNS：我们非常熟悉且最简单的一种方式，跟域名和IP映射的原理类似，我们可以将服务域名名称和一到多个机器IP进行关联或者是一个负载均衡器（指向服负载均衡好处是可以避免失效DNS条目问题）。DNS的服务发现方式最大的优点就是它是一种大家熟知的标准形式，技术支持性好。缺点就是当服务节点的启动和销毁变得更加动态时DNS更新条目很难做到高可用和实时性。

2）Zookeeper：最开始是作为Hadoop项目的一部分，基于Paxos算法的ZAB协议，它的应用场景非常广泛，包括配置管理，分布式锁，服务间数据同步，领导人选举，消息队列以及命名空间等。类似于其它分布式系统，它依赖于集群中的运行的大量节点，通常至少是三台，以提供高可用的服务。借助于其核心提供的用于储存信息的分层命名空间，我们可以在此中增删改查新的节点以实现储存服务位置的功能。除此外还可以对节点添加监控功能，以便节点改变时可以得到通知。作为服务发现主要优点是成熟、健壮以及丰富的特性和客户端库。

缺点是复杂性高导致维护成本太高，而且采用Java以及相当数量的依赖使其资源占用率过高。

3）Consul：是强一致性的数据存储，使用Gossip形成动态集群（原生数据中心Gossip系统，不仅能在同一集群中的各个节点上工作，而且还能跨数据中心进行工作）。支持配置管理，服务发现以及一种键值存储，也具备类似zookeeper的服务节点检查功能。主要优点是支持DNS SRV发现服务，这增强了与其它系统的交互性。除此外Consul还支持RESTful HTTP接口，这使集成不同技术栈变得更容易。

4）Doozerd：是比较新的服务发现解决方案，与 ZooKeeper 具有相似的架构和功能，因此可以与 ZooKeeper 互换使用。

5）Eureka：如果需要实现一个 AP（Availability and Partition）系统， Eureka 是一个很好的选择，因为在出现网络分区时，Eureka 选择可用性，而不是一致性。

6）Etcd：采用RESTFul HTTP协议的键值对存储系统，基于Raft算法实现分布式，具有可用于服务发现的分层配置系统。主要优点是，容易部署，设置和使用，有非常好的文档支持。缺点是实现服务发现需要与第三方工具结合。常与Etcd组合使用的工具是Registrator和Confd，Registrator通过检查容器在线或停止来完成相关服务数据的注册和更新，Confd作为轻量级的配置管理工具通过储存在Etcd中的数据来保持配置文件的最新状态。

3.2.3 Etcd及Raft算法

Etcd是一个分布式可靠键值存储系统,它广泛应用于服务发现和配置管理。作为Kubernetes和CoreOS的重要组成部分,Etcd支持简单的客户端API和HTTP接口,提供高可用性和一致性。

Etcd使用Raft算法实现分布式一致性。Raft算法比Paxos更易理解和实现。在Raft中,Etcd节点有Leader、Follower和Candidate三个角色。当Follower检测到Leader失效时,会开始选举过程选择新Leader。 candidate会向其他节点发送选票请求。如果candidate获得超过半数节点选票,它就成为新一轮的Leader。选举完成后,Leader会通过心跳维持与Follower的连接状态。如果连接失效,会重新开始选举。这使得Etcd能够高可用且具有容错能力。

总之Raft 算法不论在教学方面还是实际实现方面都比 Paxos 和其他算法更出众，比其它算法也更加简单和更好理解，而且拥有许多开源实现并被许多公司支持使用，除此之外它的安全性也已经被证明，最重要的它的效率相比其它算法也更加具有竞争力。

3.3服务通信

微服务架构本质上还是分布式系统，而且相比传统的分布式系统，由于服务粒度更小，数据更多元化，交互会更复杂甚至需要经常夸节点，这使得网络成为微服务架构中又一重要问题。不管是同步通信机制还是异步通信机制，各种组件以及服务边界的通信都要根据实际情况来选择合适的机制。当然不管是何种方式，在分布式网络环境中我们都应该知道网络是不可靠的，所以在制定通信机制时可能需要额外的考虑容错和弹性等问题。

3.3.1REST和RESTful

REST（Resource Representational State Transfer，表现层状态转移），通俗解释就是资源在网络中以某种表现形式进行状态转移，资源即数据；某种表现形式诸如JSON，XML，JEPG等；状态变化通常通过HTTP动词诸如GET、POST、PUT、DELETE等来实现，REST出自卡内基梅隆大学Roy Fielding的博士毕业论文。

REST有很多优点，当然无规矩不成方圆，有它的优点也有它的原则：首先，客户端服务器分离，这样的好处是操作简单，同时提高了客户端的简洁和服务端的性能，而且可以让服务器和客户端分别优化和改进；其次，无状态，也就是客户端每个请求独立且要包含服务端需要的所有信息，这样的好处是可以单独查看每个请求，可见性高且更容易故障恢复，还能降低服务端资源使用；第三，可缓存，服务器返回信息时，必须被标记是否可以进行缓存，如果可以进行缓存，客户端可能会复用历史信息发送请求，这样可以减少交互次数和提升速度；第四，系统分层，组件之间透明，这样降低了耦合和系统复杂性，提高了可扩展性；第五，统一接口，提高交互可见性和可单独改进组件；最后，支持按需实现代码，这可以提高可扩展性。

满足上述约束和原则的应用程序或设计风格就是 RESTful，REST并没有规定底层需要支持什么协议，但是最常用的是HTTP，因为HTTP的动词方法能够很好的和资源一起使用，所以REST就是通过使用HTTP协议和uri利用客户端和服务端对资源进行CRUD(Create/Read/Update/Delete)增删改查操作。通过这种风格设计的应用程序HTTP接口也就是RESTful HTTP API是现在Web架构中的常用方法，可以作为在微服务中解决集成问题的RPC替代方案。

3.3.2从Thrift到gRPC

RPC是一种进程间通信模型,允许不同机器上的进程通过调用本地方法的方式进行通信。它隐藏了网络通信的复杂细节,开发者可以通过直接调用本地方法的方式调用远程服务。

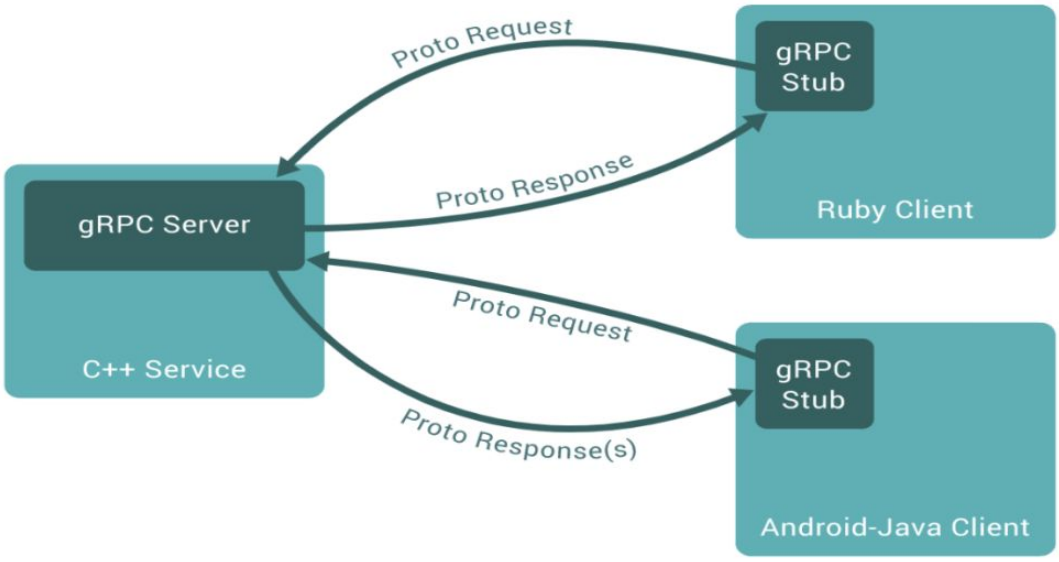
Thrift和gRPC都是实现RPC的常用框架。Thrift通过IDL定义接口,支持多种语言,但不同版本兼容性差,成本较高。

gRPC默认使用Protobuf作为数据格式。它通过Proto文件定义服务接口。服务端和客户端使用Proto编译器和gRPC插件生成必需代码。服务端实现接口并通过RPC框架发布服务。客户端调用接口,服务端处理并返回结果。

之前项目使用Thrift实现微服务间RPC通信。考虑到Thrift不同版本不兼容问题和学习成本高,后来选择了gRPC框架,因为它使用Protobuf格式,跨语言能力强,性能好,维护成本低。

总之,RPC通过本地调用的方式隐藏网络通信细节,极大简化了分布式系统开发。Thrift和gRPC是两种主流的实现方案,gRPC使用于本项目以提高交互效率和维护性。

如下图展示了不同语言的服务端和客户端通过基于Protobuf协议的gRPC进行的远程调用。在后续的第五章的试验中，我们会分别通过PHP的客户端和Golang客户端通过gRPC去调用Python实现的密码评测服务端。此外，最新的Google API将配备其接口的gRPC版本，这意味着可以借助gRPC轻松的将谷歌应用程序构建到一些服务中，而且在最新的Proto3中会有更多的新功能和支持更多的语言，相信gRPC会越来越通用。



3.3.3 消息队列

消息队列是分布式系统中异步通信常用的组件，可以用来解耦异构系统或不同服务之间的耦合从而提高可用性和保障最终一致，也可以将传统的串行业务进行并行处理从而提高吞吐量和降低响应时间，除此外还可以用来进行流量削峰以实现高性能可伸缩架构，当然保证消息顺序作为队列的天然特性也是用途之一。

常用消息队列有ActiveMQ，RabbitMQ，Kafka，除此外还有Activemq的下一代产品更快更强健的Apollo，以及阿里开源的支撑多次双十一活动的Rocketmq，甚至我们非常熟悉的内存数据库Redis也可以基于发布和订阅做简单的消息队列。

3.4 服务监控与治理

日志与监控

当产品环境出错时，需要快速的定位问题，检测可能发生的意外和故障。而日志与监控是快速定位和预防的不二选择，在微服务架构中更是至关重要。

- 高度可观察，我们需要对正在发生的事情有一个整体的视角。
- 聚合你的日志，聚合你的数据，从而当你遇到问题时，可以深入分析原因。
- 当需要重现令人讨厌的问题，或仅仅查看你的系统在生产环境如何交互时，`关联标识`可以帮助你跟踪系统间的调用。

监控主要包括服务可用状态、请求流量、调用链、错误计数，结构化的日志、服务依赖关系可视化等内容，以便发现问题及时修复，实时调整系统负载，必要时进行服务降级，过载保护等等，从而让系统和环境提供高效高质量的服务。

比如商业解决方案`splunk，sumologic`，以及开源产品ELK他们都可以用于日志的收集，聚合，展现，并提供搜索功能，基于一定条件，触发邮件警告。

`Spring boot admin`也可以用于服务可用性的监控，`hystrix`除了提供熔断器机制外，它还收集了一些请求的基本信息（比如请求响应时间，访问计算，错误统计等），并提供成的dashboard将信息可视化。

关于性能监控和调用链追踪，考虑使用dynatrace和zipkin/Sleuth

自动化

在微服务架构下，面临如下挑战：

- 分布式系统
- 多服务，多实例
- 手动测试，部署，发布太消耗时间
- 反馈周期太长

传统的手工运维方式必然要被淘汰，微服务的实施是有一定的先决条件：那就是自动化，当服务规模化后需要更多`自动化`和`标准化`的手段来提升效能和降低成本。

- 自动化测试必不可少，因为对比单块系统，确保我们大量的服务正常工作是一个更复杂的过程。
- 调用一个统一的命令行，以相同的方式把系统部署到各个环境。
- 考虑使用环境定义来帮助你明确不同环境间的差异，但同时保持使用统一的方式进行部署的能力。
- 考虑创建自定义镜像来加快部署，并且拥抱全自动化创建不可变服务器的实践。

`自动化一切可以自动化的`，降低部署和发布的难度, 比如： 在持续集成和持续交付中，自动化编译，测试，安全扫描，打包，集成测试，部署，随着服务越来越多，在发布过程中，需要进一步自动化蓝绿部署（做到老版本到新版本的平滑过渡）还可以使用pipeline as code的实践，用代码来描述你的流水线。关于部署有很多选择，可以使用虚拟机，容器docker，或者流行的无服务架构lambda（AWS Lambda也有一些明显的局限。它并不适合被用来部署长期运行的服务，请求需要在 300 秒内完成，当然你可以通过hack的方式延迟时间)。

然后， 可以采用基础设施及代码的实践，比如亚马逊的`cloudformation`，还有`terrform`，通过代码来描述计算和网络等基础设施， 可以快速为一个全新的服务，构架它所需要的环境，保持各`环境的一致性`。

4. 微服务实施与部署

接下来介绍关于微服务架构目前国内外主要的解决方案。

4.1 微服务架构方案Spring Cloud

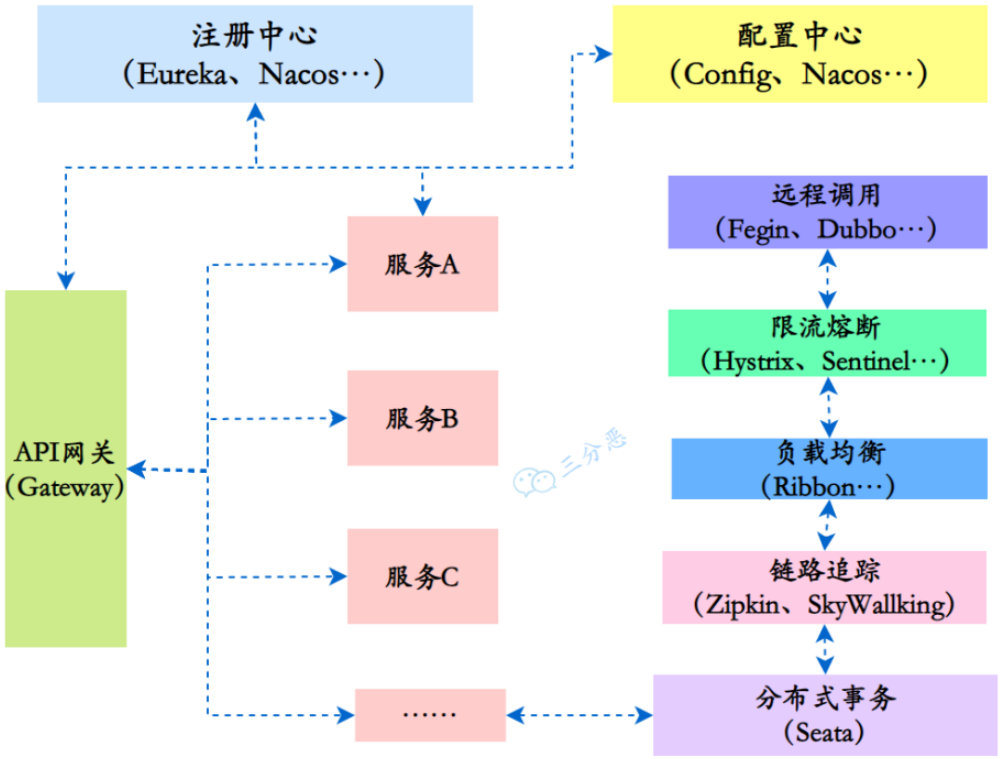
Spring Cloud并不是一个项目而是一组项目的集合在Spring Cloud中包含了很多的子项目每一个子项目都是一种微服务开发过程中遇到的问题的一种解决方案它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

Spring Cloud Alibaba 与 Spring Cloud 生态其他方案之间对比图如下：

	Spring Cloud Alibaba	Spring Cloud Netflix	Spring Cloud 官方	Spring Cloud Zookeeper	Spring Cloud Consul	Spring Cloud Kubernetes
分布式配置	Nacos Config	Archaius	Spring Cloud Config	Zookeeper	Consul	ConfigMap
服务注册发现	Nacos Discovery	Eureka	-	Zookeeper	Consul	API Server
服务熔断	Sentinel	Hystrix	-	-	-	-
服务调用	Dubbo RPC	Feign	OpenFeign RestTemplate	-	-	-
服务路由	Dubbo+Servlet	Zuul	Spring Cloud Gateway	-	-	-
分布式消息	SCS RocketMQ Binder	-	SCS RabbitMQ Binder	-	SCS Consul Binder	-
消息总线	RocketMQ Bus	-	RabbitMQ Bus	-	Consul Bus	-
负载均衡	Dubbo LoadBalance	Ribbon	Spring Cloud LoadBalancer	-	-	-
分布式事务	Seata	-	-	-	-	-
Sidercar	Spring Cloud Alibaba Sidecar	Spring Cloud Netflix Sidecar	-	-	-	-

日常开发中一般考虑如下三种方案：

- Dubbo在一开始的定位是RPC框架，近几年随着微服务的兴起，才开始向着微服务解决方案更新迭代。但目前为止，Dubbo还相当不完善，需要借助很多第三方组件。
- Spring Cloud Netflix是一站式微服务解决方案，但已停止开发，进入维护阶段。
- Spring Cloud Alibaba是在Spring cloud netflix基础上封装了阿里巴巴的微服务解决方案，是目前最完善的微服务解决方案。

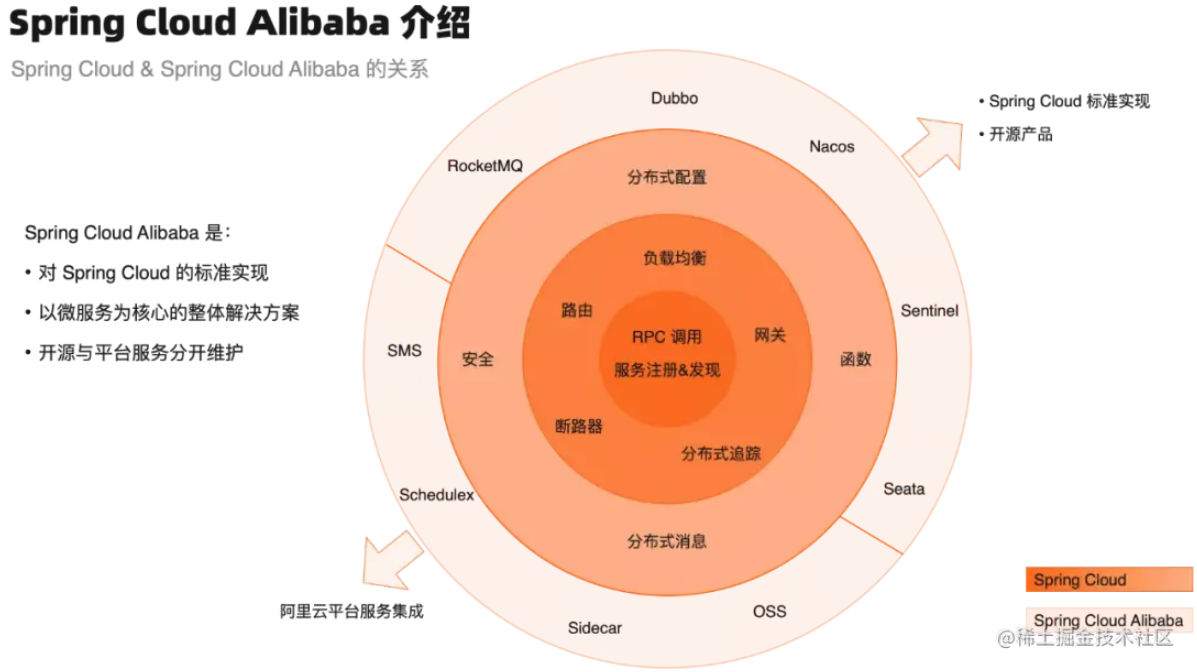


4.2 Spring Cloud Alibaba

Spring Cloud Alibaba 是阿里巴巴结合自身丰富的微服务实践而推出的微服务开发的一站式解决方案，是 Spring Cloud 第二代实现的主要组成部分。吸收了 Spring Cloud Netflix 微服务框架的核心架构思想，并进行了高性能改进。自 Spring Cloud Netflix 进入停更维护后，Spring Cloud Alibaba 逐渐代替它成为主流的微服务框架。

同时 Spring Cloud Alibaba 也是国内首个进入 Spring 社区的开源项目。2018 年 7 月，Spring Cloud Alibaba 正式开源，并进入 Spring Cloud 孵化器中孵化；2019 年 7 月，Spring Cloud 官方宣布 Spring Cloud Alibaba 毕业，并将仓库迁移到 Alibaba Github OSS 下。

在 Spring Cloud 众多的实现方案中，Spring Cloud Alibaba 凭借其支持组件最多，方案最完善，在 Spring Cloud 生态家族中扮演了重要角色。



4.2.1Spring Cloud Alibaba功能与组件

主要功能

- **服务限流降级：** 默认支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Spring Cloud Gateway, Zuul, Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
- **服务注册与发现：** 适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持。
- **分布式配置管理：** 支持分布式系统中的外部化配置，配置更改时自动刷新。
- **消息驱动能力：** 基于 Spring Cloud Stream 为微服务应用构建消息驱动能力。
- **分布式事务：** 使用 @GlobalTransactional 注解， 高效并且对业务零侵入地解决分布式事务问题。
- **阿里云对象存储：** 阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **分布式任务调度：** 提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker（schedulerx-client）上执行。
  阿里云短信服务：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

组件

- [Sentinel]：阿里巴巴源产品，把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。
- [Nacos]：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- [RocketMQ]：一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。
- [Dubbo]：Apache Dubbo 是一款高性能 Java RPC 框架。
- [Seata]：阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。
- [Alibaba Cloud OSS] : 阿里云对象存储服务（Object Storage Service，简称 OSS），是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- [Alibaba Cloud SchedulerX]: 阿里中间件团队开发的一款分布式任务调度产品，提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。

- [Alibaba Cloud SMS] : 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

5.微服务的优势和挑战

5.1 微服务的优势

微服务的优势包括：

- 单一职责：微服务架构中的每个节点高度服务化，都是具有业务逻辑的，符合高内聚、低耦合原则以及单一职责原则的单元，包括数据库和数据模型；不同的服务通过“管道”的方式灵活组合，从而构建出庞大的系统。
- 轻量级通信：通过REST API模式或者RPC框架，事件流和消息代理的组合相互通信，实现服务间互相协作的轻量级通信机制。
- 独立性：在微服务架构中，每个服务都是独立的业务单元，与其他服务高度解耦，只需要改变当前服务本身，就可以完成独立的开发、测试、部署、运维。
- 进程隔离：在微服务架构中，应用程序由多个服务组成，每个服务都是高度自治的独立业务实体，可以运行在独立的进程中，不同的服务能非常容易地部署到不同的主机上，实现高度自治和高度隔离。

5.2 微服务面临的挑战

微服务的挑战包括：

- 分布式固有复杂性：微服务架构是基于分布式的系统，而构建分布式系统必然会带来额外的开销。分布式系统的挑战包括性能、可靠性、分布式通信、数据一致性等问题。
- 服务的依赖管理和测试：在微服务架构中，服务数量众多，每个服务都是独立的业务单元，服务主要通过接口进行交互，如何保证它的正常，是测试面临的主要挑战。所以单元测试和单个服务链路的可用性非常重要。
- 有效的配置版本管理：在微服务架构中，配置可以写在yaml文件，分布式系统中需要统一进行配置管理，同一个服务在不同的场景下对配置的值要求还可能不一样，所以需要引入配置的版本管理、环境管理。
- 自动化的部署流程：在微服务架构中，每个服务都独立部署，交付周期短且频率高，人工部署已经无法适应业务的快速变化。有效地构建自动化部署体系，配合服务网格、容器技术，是微服务面临的另一个挑战。
- 对于DevOps更高的要求：在微服务架构的实施过程中，开发人员和运维人员的角色发生了变化，开发者也将承担起整个服务的生命周期的责任，包括部署、链路追踪、监控；因此，按需调整组织架构、构建全功能的团队，也是一个不小的挑战。
- 运维成本：微服务架构中，每个服务都需要独立地配置、部署、监控和收集日志，成本呈指数级增长。

5.3 如何克服微服务的挑战

1. 拆分服务的原则

在拆分服务时，应遵循单一职责原则，每个服务都应该有一个明确的责任，并且服务之间应该相互独立，避免出现紧耦合的关系。

2. 设计良好的API接口

为了实现服务之间的通信，需要设计良好的API接口，这些接口应该易于使用、易于理解，并且应该具有良好的文档和版本控制。

3. 采用自动化部署和测试

采用自动化部署和测试可以降低部署和测试的成本，同时也可以提高部署和测试的速度和质量。

4. 实时监控和日志记录

实时监控和日志记录可以帮助我们及时发现和解决问题，同时也可以提供有价值的业务数据，帮助我们做出更明智的决策。

5. 选择合适的技术栈

选择合适的技术栈可以充分利用各种技术的优势，同时也可以解决各种技术栈的兼容性问题。

6.总结与展望

在面对当前业务功能复杂数据访问量巨大的背景下，微服务架构作为目前应对大规模系统的主要解决方案是有它自身的强大优势的。但微服务的实施是有一定的先决条件：基础的运维能力（如监控、快速配置、快速部署）需提前构建，否则就会陷入较被动的局面。推荐采用CI/CI改进基础设施及运维的实践，通过自动化运维使得可以快速安全的响应和处理微服务对服务部署的要求，通过容器技术保证服务环境之间拥有更高的一致性，降低“在我的环境工作，而你的环境不工作”的可能，也是为后续的发布策略和运维提供更好的支撑。

想要更好的实施微服务，首先需要考虑构建团队DevOps能力，这是保证微服务架构在持续交付和应对复杂运维问题的动力之源。

其次保持服务持续演进，使之能够快速、低成本地被拆分和合并，以快速响应业务的变化；同时要保持团队和架构对齐。微服务看似是技术层面的变革，但它对团队结构和组织文化有很强的要求和影响。识别和构建匹配架构的团队是解决问题的另一大支柱。

所以成功实施微服务并不是一件孤立的事情，它关联很多其他事情，架构、工具到团队协同，需要同步建设，它是一个系统工程。

在IT世界没有什么技术是永不过时的，微服务架构的演进就是一个例子，从单体程序到微服务架构，我们不知道下一个技术迭代点是什么时候，但我们知道微服务架构肯定还会更新，所以说IT人应该建立终身学习习惯。