微服务学习 -- Spring Cloud微服务框架实践

INDEX

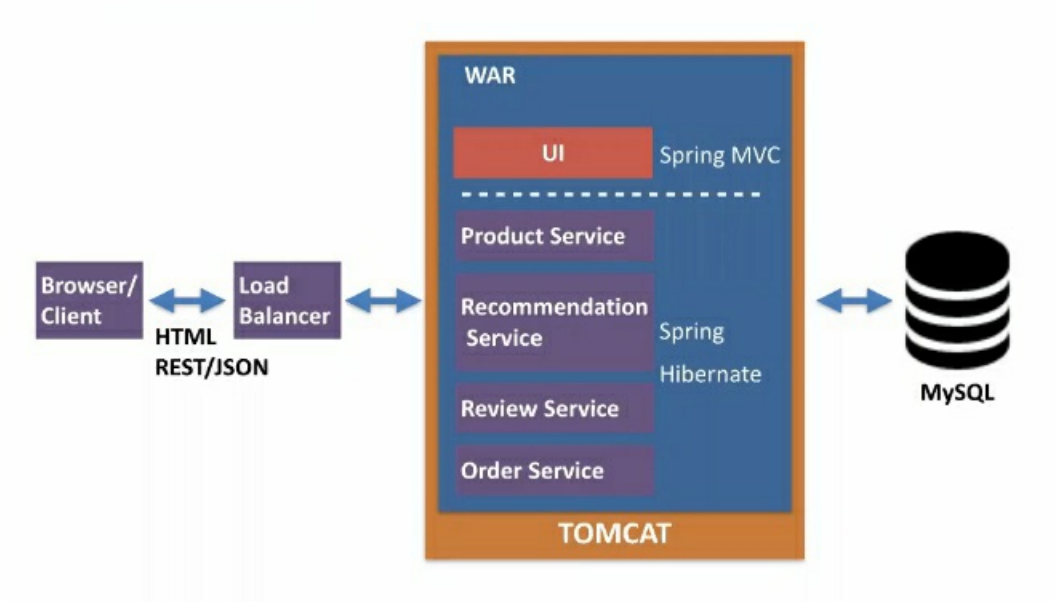
0.微服务整体结构及接口设计原则

0 微服务架构

0.1 微服务的定义

0.1.1 单体应用及其痛点理解

传统的开发模式都是基于单体应用的开发模式，虽然在设计阶段发展了MVC之类的设计思想，在逻辑上有着明确的分层，但是在应用的开发，测试，部署上都有着很强的耦合，相互依赖



单体应用面临的问题：

- 维护的成本增加：随着代码量的增加，代码之间的耦合越来越严重(更多情况是为了改bug而引入的一些代码)，代码的可读性越来越差，而且随着后期可能测试覆盖不到的点越来越多，整个系统的维护不断呈现一种恶性循环的状态
- 交付周期长：单体应用的实现过程，在各个开发，测试，构建，部署的各个阶段都存在着比较大的耦合，任何一个点的问题，都可能会导致整个系统的崩溃(AAA的开发中表现很频繁，如一个业务的权限点文件的错误，最后导致表现的现象可能是整个系统都没有办法使用；或者是CI构建一个完全不了解的模块编译错误，导致需要整个工程重新编译，而往往一次编译就可能需要一个小时)，没有一个很好的机制可以做到并行
- 可伸缩性差：单体应用虽然**易于伸缩**，这种模式是基于clone的 方式，去做水平伸缩，而不是按照业务的需求去实现
- 人员能力培养：单体应用对人员能力要求是比较高，由于整个应用的各方面可能都需要了解，才能将系统跑起来

0.1.2 微服务框架定义

下面是 `Martin Fowler` 对微服务架构的一个定义：

> The microservice architectural style is an approach to developing a single application as a suite of **small services**, each running in its **own process** and communicating with **lightweight mechanisms**, often an HTTP resource API. These services are built around **business capabilities** and **independently deployable** by fully **automated deployment machinery**. There is **a bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。
每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的RESTful API）。
每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、生产环境等。
另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的□言、工具对其进行构建。

然后在其blog中提到了微服务架构的基本特征

- Componentization via Services 以服务来实现组件化(而不是以依赖库的方式)

从独立部署的角度去考虑，依赖于库的方式，任何一个底层库的改变，可能都需要你对整个系统去重新部署；
另外通过RPC的方式，能够让组件更加明确对外提供的接口，做到松耦合

- Organized around Business Capabilities 通过业务来组织服务
- Products not Projects 以做产品的思路而不是做项目的思路

  `you build it， you run it !`
- Smart endpoints and dumb pipes
- Decentralized Governance 服务管理去中心化
- Decentralized Data Management 数据管理去中心化
- Infrastructure Automation 自动化
- Design for failure 面向失效的设计原则
- Evolutionary Design 快速演进的设计原则

0.2 微服务产生背景及认识误区

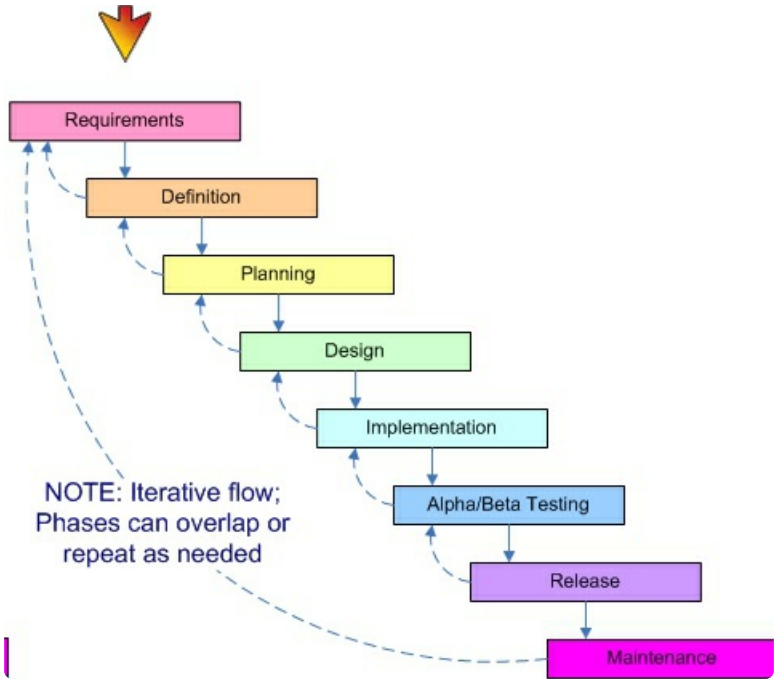
微服务架构并不是新发明的语言或者开发技术，而是过去需求的集合及IT技术演进，基于敏捷，持续交付，DevOps等形成的一种架构风格

0.2.1 微服务产生背景

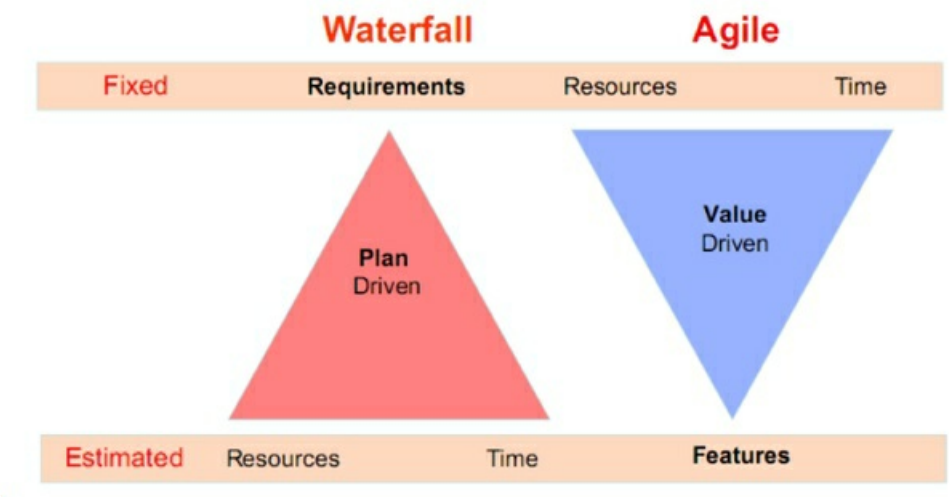
在介绍微服务产生的背景之前，先梳理几个软件开发演进过程中产生的名词，分别是 `敏捷` 、 `DevOps` 、 `持续交付流水线`

0.2.1.1 敏捷(Alige)

传统的交付很多是按照瀑布模型的方式进行开发，但是针对瀑布模型这种预定义的方法，其每个阶段都有着强烈的依赖关系，前一阶段的输入被当成是后一阶段的输入，如果前一阶段的输入质量不高或者未能达标，则对后一阶段的影响是巨大的，甚至是造成项目的停滞，导致整个开发周期的延长；根据现有的调查，70%采用瀑布模型开发的软件项目都以失败而告终。



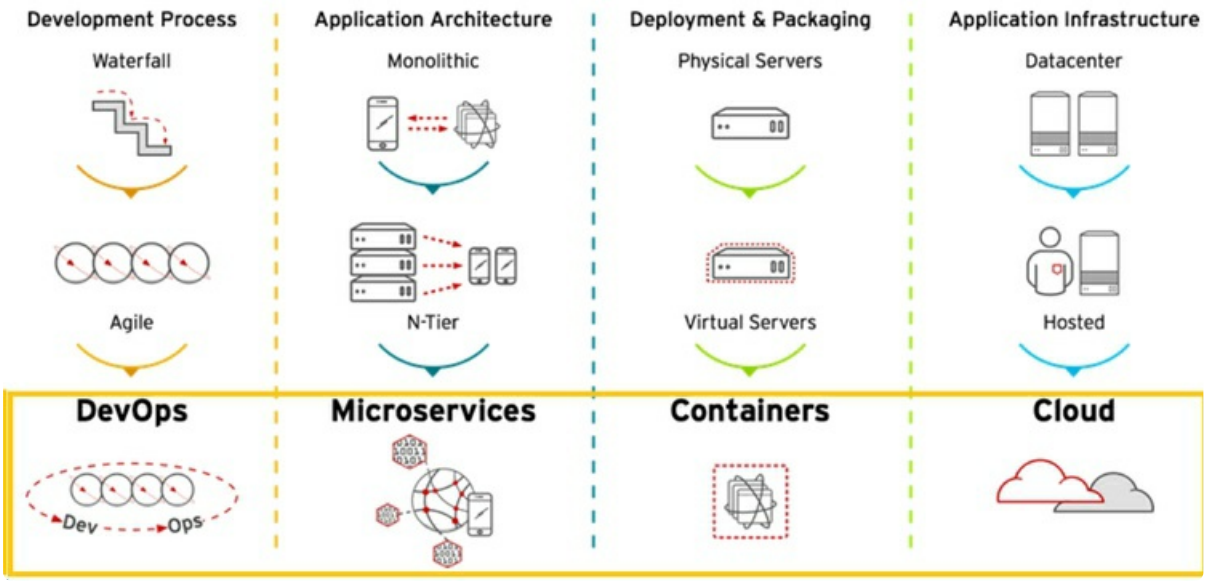
敏捷(Alige)这种开发思路就是在这种背景诞生，其**核心**在于从瀑布开发中"fix Scope，elastic time&resource"转变为"fixed time&source, elastic scope"，体现为由计划驱动转变为由价值驱动



但是敏捷关注的点主要是需求到实现这一过程，能够实现快速响应需求，快速实现；而没有办法保证功能的快速测试，快速部署，从而衍生出DevOps，构成一个更大的循环，促进功能能够更快速响应客户需求

02.1.2 DevOps

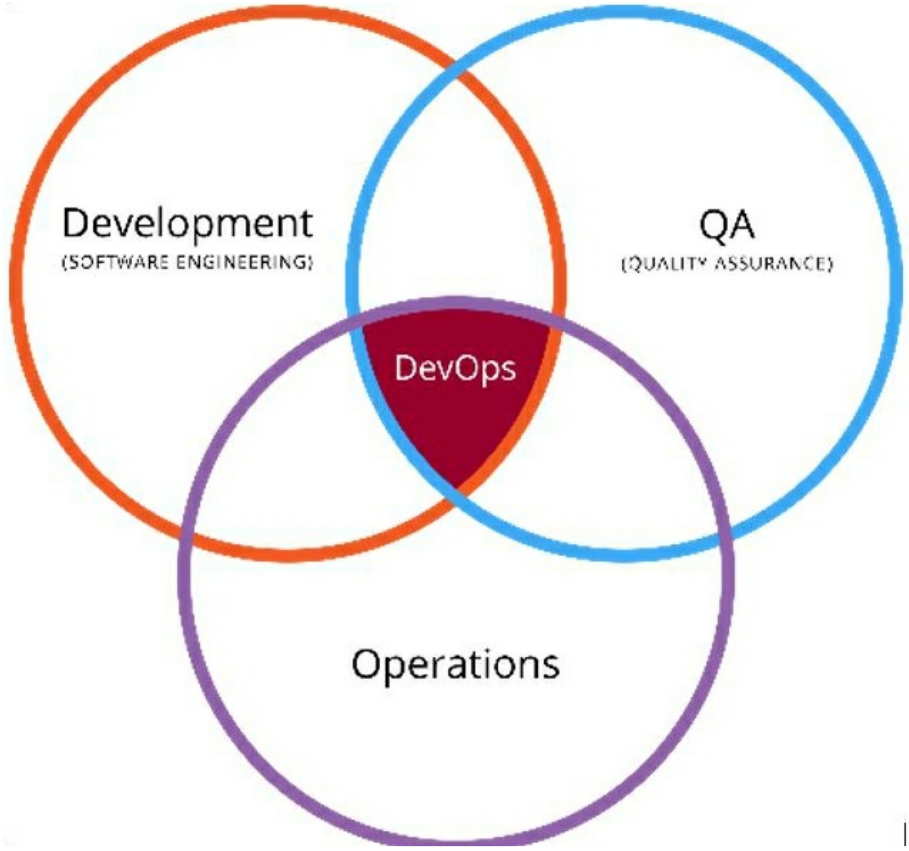
回顾软件行业的研发模式，可以发现大致有三个阶段：瀑布式开发、敏捷开发、DevOps。下面这张图是红帽子分享，从开发模式，应用架构，部署方式，基础设施等角度，展示了软件工程的变化。



DevOps一词的来自于Development和Operations的组合，突出重视软件开发人员和运维人员的沟通合作，通过自动化流程来使得软件构建、测试、发布更加快捷、频繁和可靠。

DevOps是为了填补开发端和运维端之间的信息鸿沟，改善团队之间的协作关系。不过需要澄清的一点是，从开发到运维，中间还有测试环节。DevOps其实包含了三个部分：开发、测试和运维。

换句话说，DevOps希望做到的是软件产品交付过程中**IT工具链的打通**，使得各个团队减少时间损耗，更加高效地协同工作。专家们总结出了下面这个DevOps能力图，良好的闭环可以大大增加整体的产出。

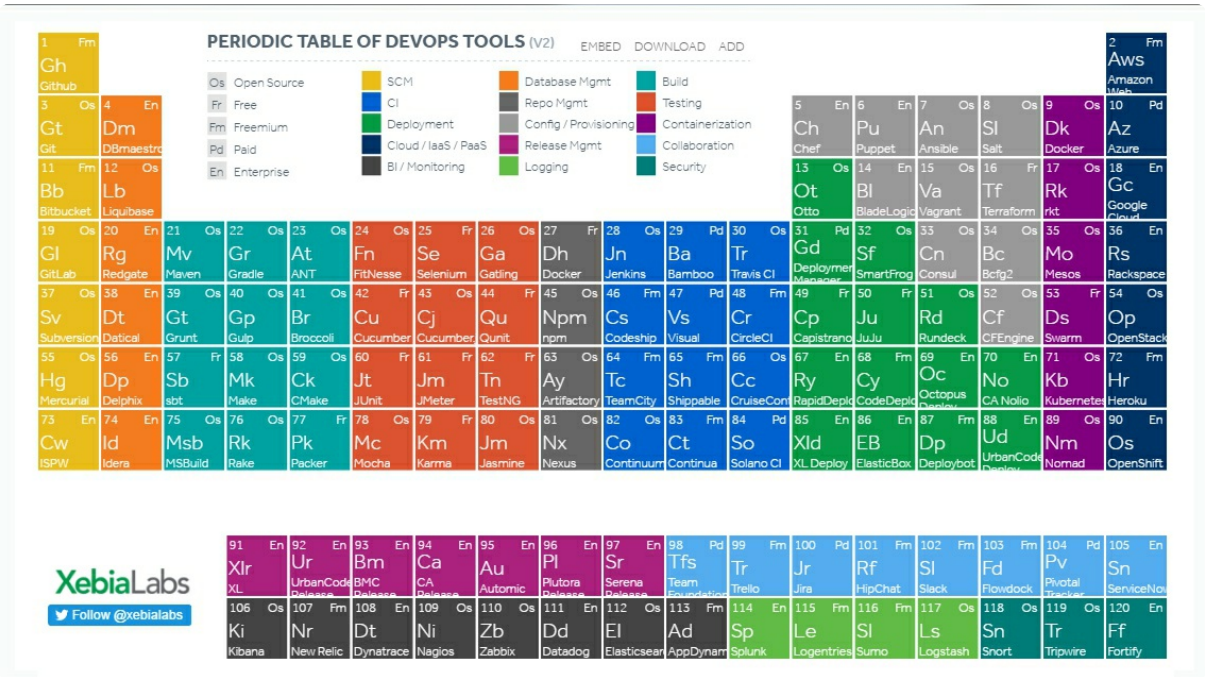


实现DevOps依赖于工具上的准备和组织间的协同，其中工具链方面包含了以下方面：

- 代码管理（SCM）：GitHub、GitLab、BitBucket、SubVersion、TFS
- 构建工具：Ant、Gradle、maven
- 自动部署：Capistrano、CodeDeploy

- 持续集成（CI）：Bamboo、Hudson、Jenkins
- 配置管理：Ansible、Chef、Puppet、SaltStack、ScriptRock GuardRail
- 容器：Docker、LXC、Rkt、第三方厂商如AWS
- 编排：Kubernetes、Apache Mesos、DC/OS
- 服务注册与发现：Zookeeper、etcd、Consul
- 脚本语言：python、ruby、shell
- 日志管理：ELK、Logentries
- 系统监控：Datadog、Graphite、Icinga、Nagios
- 性能监控：AppDynamics、New Relic、Splunk
- 压力测试：JMeter、Blaze Meter、loader.io
- 预警：PagerDuty、pingdom、厂商自带如AWS SNS
- HTTP加速器：Varnish
- 消息总线：ActiveMQ、SQS
- 应用服务器：Tomcat、JBoss
- Web服务器：Apache、Nginx、IIS
- 数据库：MySQL、Oracle、PostgreSQL等关系型数据库；cassandra、mongoDB、redis等NoSQL数据库
- 项目管理（PM）：Jira、Asana、Taiga、Trello、Basecamp、Pivotal Tracker

xebialabs整理了炫酷的DevOps工具链周期表



0.2.1.3 持续集成能力成熟度

如何缩短交付周期，按照持续交付的能力成熟度模型中，包含以下 7点：

1. 持续部署
2. 内建质量
3. 环境管理
4. 数据管理
5. 反馈验证
6. 组织协同

7. 松耦合架构

上述几点，除松耦合架构之外，其他的很多点在之前的开发过程中都有了很大程度的发展，但是如果架构本身不支持，互相之间有着很多的依赖，依然没有办法有效地缩短交付周期。

根据前面提到软件行业的变化， 从下面几个角度来解释微服务架构的产生背景。

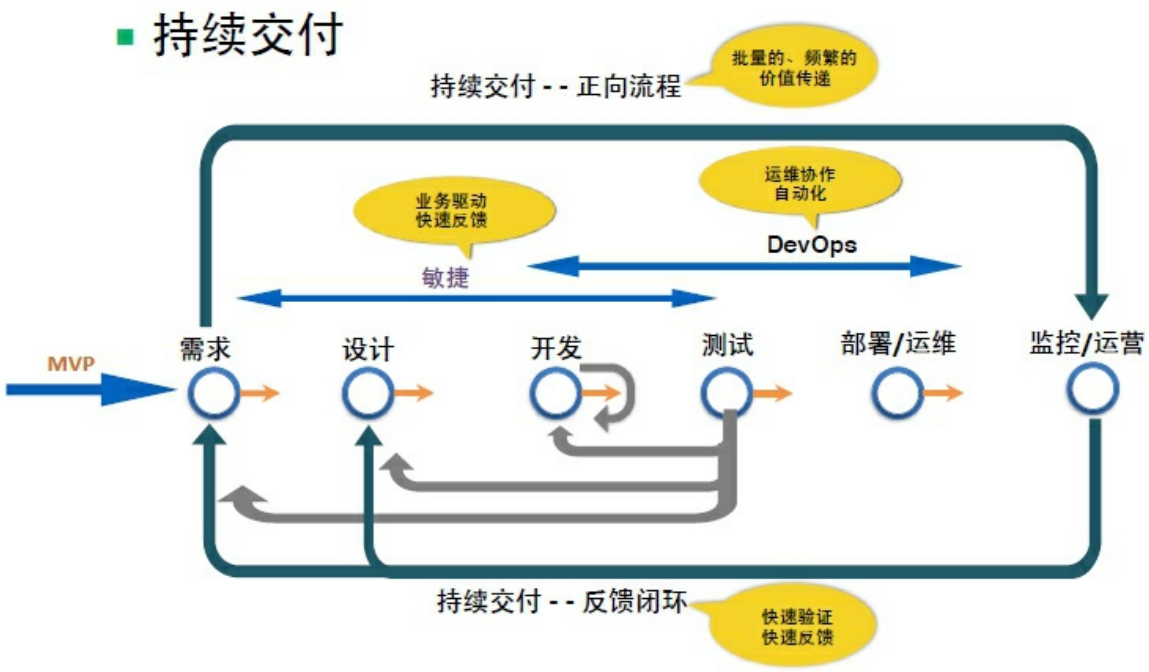
首先，互联网行业以及移动互联网是微服务架构诞生的一个孵化器，由于互联网行业天生就有着**快速交付**的压力，根据MVP(Minimum Viable Product)的理论，希望其产品能够不断的快速迭代；大部分微服务使用的标杆性企业都是互联网企业如，AWS，Netflix等；

然后是由于前面提到的单体应用的带来各种痛点；

接着是以容器化等新技术的出现，带来对传统部署模型不断简化，能够有效提升运维效率；

DevOps解决的是开发和运维之间的本身冲突，开发崇尚的是不断更新，拥抱变化，而运维则是希望稳定，使得系统保持稳定运行；其代表的工具链的不断成熟

最后是在以快速交付的思想下和技术能力不断成熟情况下产生的持续交付的概念，对微服务框架有着极大的促进；下面这张图是对前面概念的一个汇总



0.2.2 微服务认识误区

微服务能让开发变得更加简单

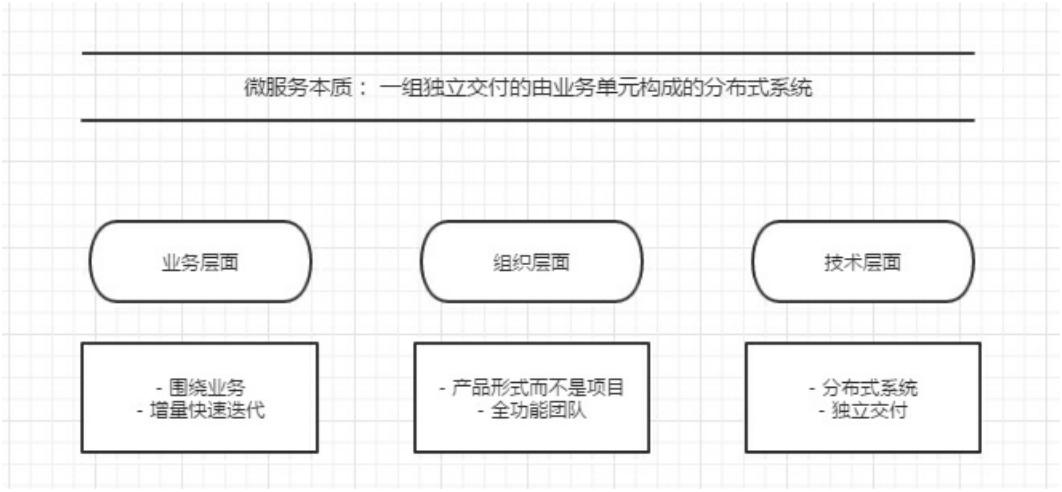
微服务架构并不是银弹，微服务面临的挑战

- 分布式系统的复杂度
    1. 时延/性能问题
    2. 事务一致性问题
    3. 服务治理 （服务的动态部署，autoscaling)
    4. 前后台通信
- 运维成本
    1. 环境配置
    2. 部署/监控/告警
    3. 问题定位

- 测试
    1. 合理的测试策略
- 团队协作

0.2.3 微服务的本质

> 微服务是一组**独立交付**的由**特定业务单元**构成的**分布式**系统



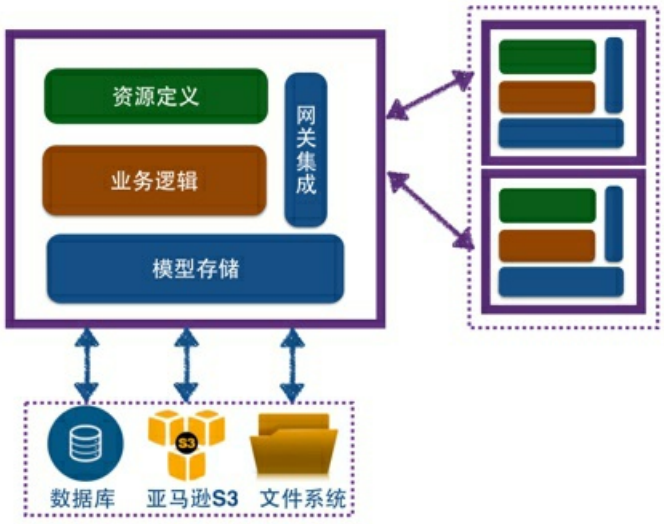
不要仅仅局限于将技术层面，更重要的是根据业务，按照合理的组织协作实现功能交付。

微服务有着一下优点：

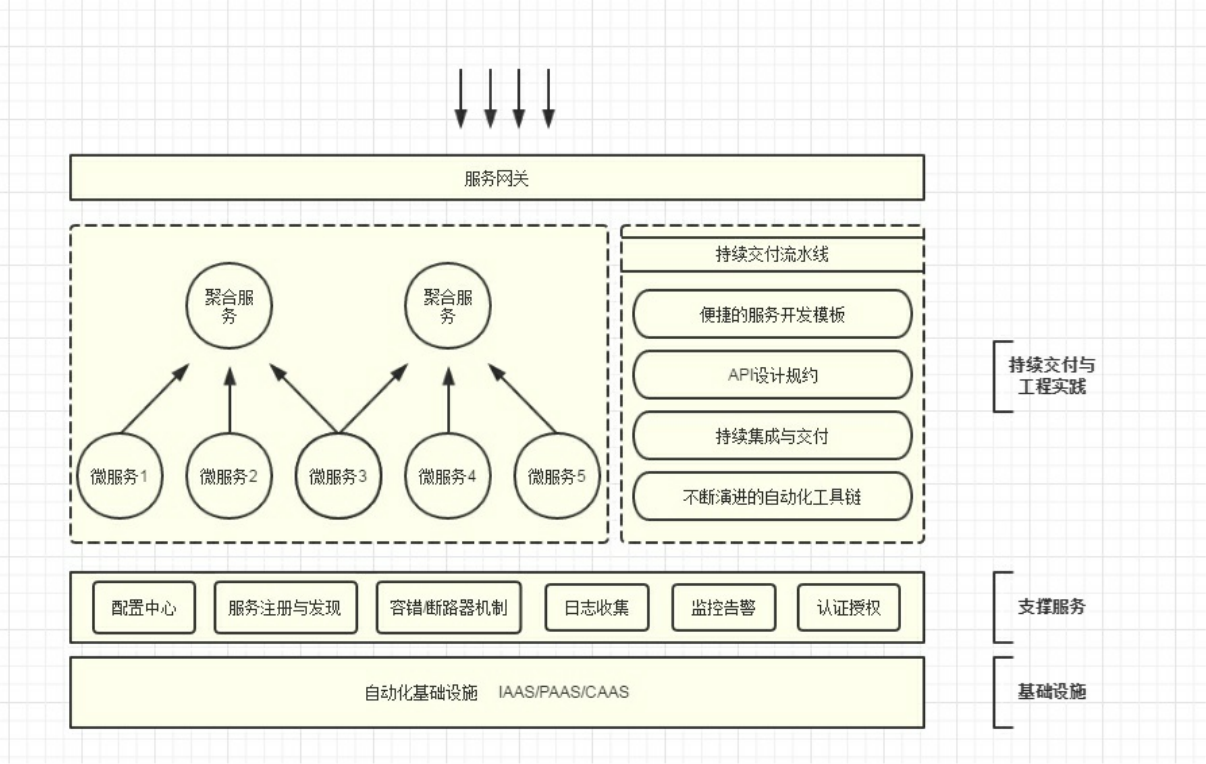
- 独立性：独立部署，灵活按需伸缩
- 技术实现多样性：不受技术框架限制，多种技术栈
- 组织优化：康威定律，逆康威定律
- 边界性：有着明确的边界，松耦合高内聚

0.2.4 微服务框架一般构成

单一微服务内部构成



微服务生态系统的整体架构图如下：



0.3 微服务设计原则

和Maritin Fowler提出的微服务框架特点一致，在未付开发过程中主要关注以下4个原则

- 围绕业务
    1. 构建服务

        > 解决什么是微服务，如何划分服务粒度的疑问；微服务的拆分需要贴合具体的业务和团队自动化能力；基于DDD(领域驱动设计)或者面向对象的设计方法

    2. 构建团队
- 去中心化
    1. 解决技术栈的限制
    2. 服务管理
    3. 数据管理 ： 业务与数据分离，各个微服务可以有自己独立的数据库；各业务自己通过接口访问其他业务数据
- 自动化

  持续交付流水线
    1. 自动化基础设施
    2. 自动化测试
    3. 自动化部署
- 演进式架构

支持增量式变更作为第一原则；尽量降低变更的成本，使得架构的演进更加容易

0.4 接口设计原则

0.5 遗留问题

问题： 精益创业的思路 (精益思想，MVP)

问题： 持续交付能力成熟度模型

问题： CAP原则

问题： 事务一致性：ebay最终一致性 -> 事务消息和数据消息 每个事务有自己的消息，通过event table记录消息，通过定时任务去执行同步event table，**两阶段提交** 同构数据库是可以解决数据一致性问题，但是因为微服务本身可能涉及到各种各样的技术栈，可能无法使用相同的数据库，所以一般强调的最终一致性

问题： SOA的概念与落地方案

SOA 企业级自定向下实施

	SOA实现	微服务架构实现
实施策略	企业级，自顶向下开展实施	团队级，自底向上开展实施
实施粒度	服务由多个子系统组成，粒度粗	一个系统被拆分成多个服务，粒度细
核心理念	企业服务总线，集中式的服务架构	去中心化，松散的服务架构
集成方式	集成方式复杂（*ESB/WS/SOAP*）	集成方式灵活（*HTTP/REST/JSON*）
部署策略	部署集中，相互依赖	部署独立
实施周期	实施周期长	实施周期短

问题： 领域驱动设计的概念与实践

问题： 数据库存储过程的概念

问题： 各个微服务的数据库有大量冗余的情况？重新考虑是不是拆分服务，或者以同一个服务不同的Endpoint的方式去做

问题： 前后端分离的概念，让前端和后端独立运行？

0.6 参考

Devops的前世今生

华为架构师8年经验谈：从单体架构到微服务的服务化演进之路

1.服务注册与服务发现

1 服务注册和服务发现

服务注册和发现的问题，其实是所有的分布式系统都面临的问题，而不是在微服务架构中所特有的。

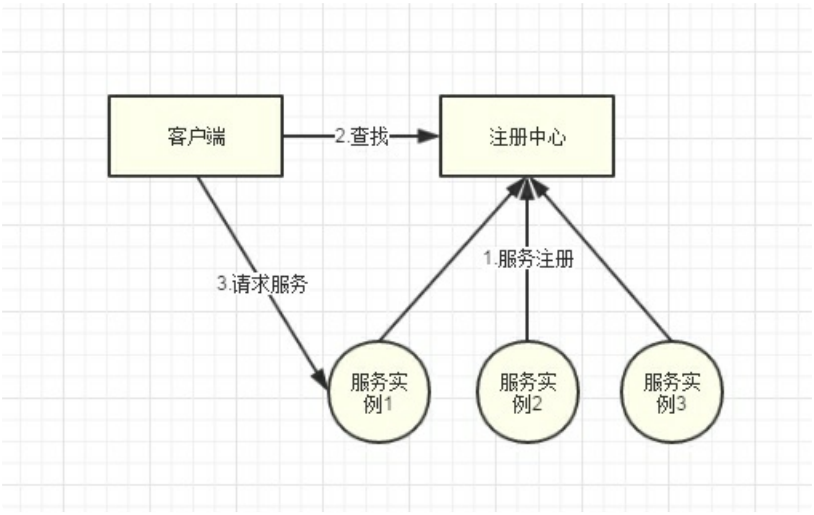
1.1 为什么需要服务注册和服务发现

- 微服务其实根据业务划分之后的一个细粒度的分布式系统架构，经常是部署在如AWS之类云环境中，会涉及到大量实例重启，IP更换等情况？
- 另外针对应用，可能会出现业务有大量增加的情况，就需要对应的伸缩机制实现**auto scalling**，此时涉及到系统的水平伸缩，后台服务数量发生变化，如何将请求转发到伸缩后节点上？
- 针对PAAS平台等依赖于容器的环境，会有一个节点包含多个实例绑定不同端口的情况，如何去感知各个服务端口的变化？

以上是微服务架构中可能会出现的一些常见情况，服务注册和发现将有效解决这些问题

1.2 服务注册和发现的核心思想

个人感觉类似于**web-service**的思想，都会一个**registry**的概念；基本框架如下图



流程如下：

1. 各个服务的实例在启动之后，向服务注册中心注册
2. 客户端通过接口向注册中心查询可用服务实例列表
3. 客户端通过列表调用服务实例(可以自己访问或者通过负载均衡)

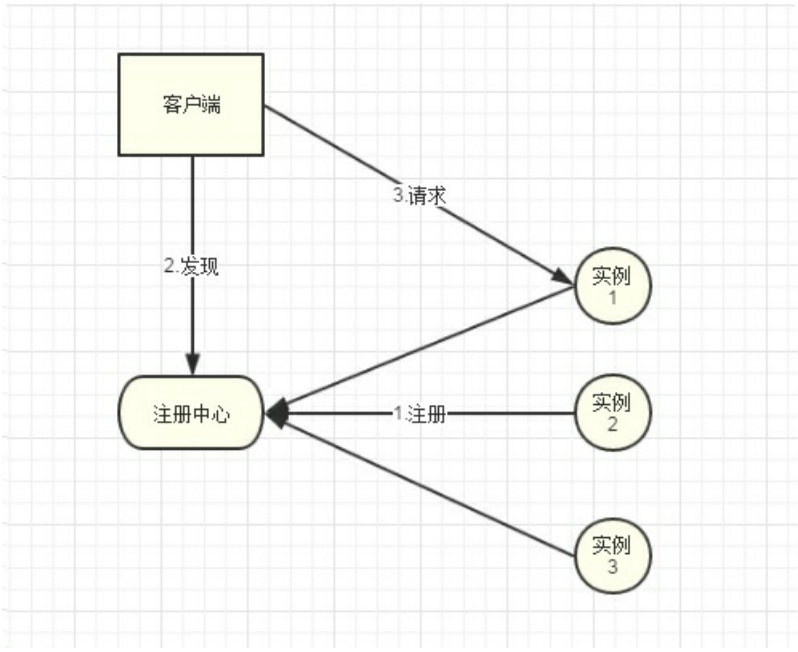
1.2.1 服务注册与发现的方式

由于服务的提供方通常会有多个实例，则针对调用方去调用服务时，就会用多种方式，如上节第3点中提到的，具体可以分为：

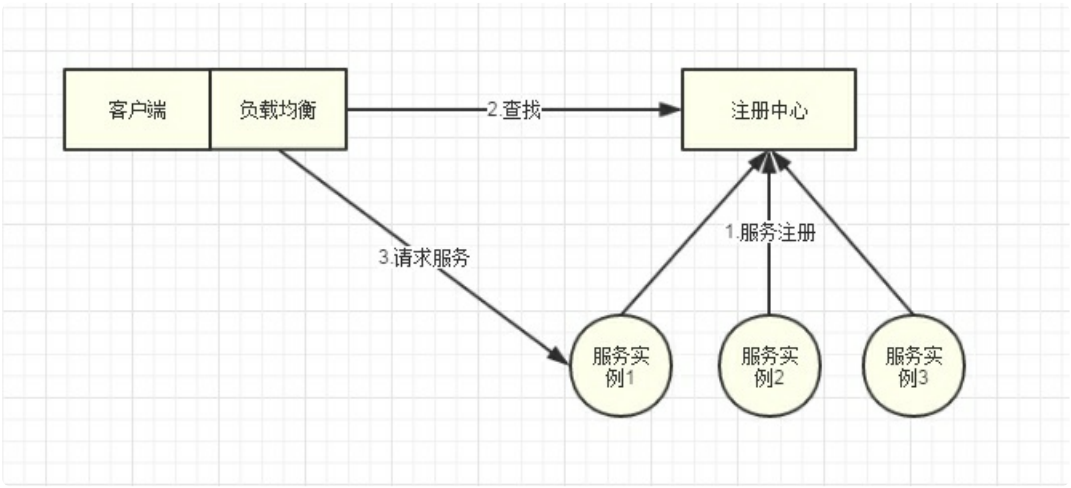
- **客户端服务发现模式**

客户端模式又分为两种情况，主要是和具体服务实例之间通信的差别：

1. 客户端模式第一种情况是指调用方通过注册中心直接去取到所有的服务实例信息，然后自己去判断选择调用那个具体的服务实例，如下图



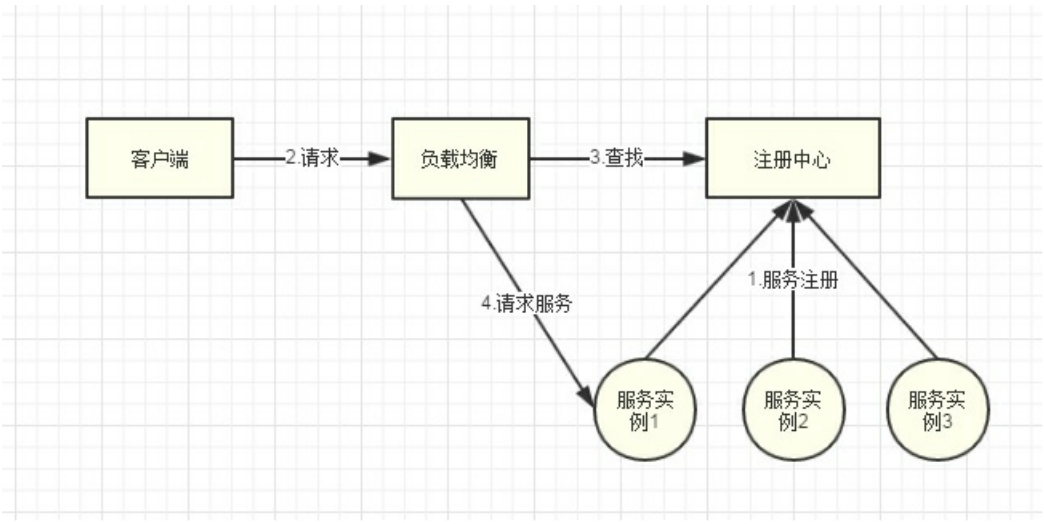
1. 客户端模式第二种情况则是利用**客户端负载均衡**实现对服务实例的选择，如下图



利用客户端负载均衡，在客户端部署一个负载均衡，在部署模式该负载均衡是和客户端部署在一起的，通过轮循等机制达到提升服务可用性的效果；通常这种方式是在客户端的代码中集成负载均衡的SDK实现

- **服务端服务发现模式**

服务端模式，重点强调的是负载均衡是一个独立的组件，独立部署，尤其是针对云环境，如AWS提供的ELB(Elastic Load Balancer)服务；客户端不直接和注册中心交互，而是**只感知到负载均衡对外暴露的接口**，客户端请求不会到达注册中心，而是通过负载均衡，然后调用服务实例



1.3 常用的服务注册与发现方案

- DNS

在服务化过程中，最原始的方案即为通过DNS(Domain Name Server)去实现服务发现；通过域名的方式将服务和一组IP绑定，调用方通过访问DNS域名解析得到服务的一个真实地址

DNS的优势在于配置方式比较简单，可以实现最简单的静态服务注册的配置；而针对前面提到的微服务架构及云环境中的情况，当有大量动态更新时，会对一致性有着很强的要求，DNS针对这种场景就比较无力

- Dubbo/DubboX

Dubbo是有阿里开源的分布式服务框架，其主要实现了服务注册与发现，服务间调用等功能；

其中服务注册与发现采用的是zookeeper实现，服务间调用为一套自研的RPC框架，对restful风格的支持不太完善；停止更新后，由当当在其基础上继续演进为DubboX，主要是提高其对restful的支持度；在微服务的配置中心，断路器等组件上功能缺失

- zookeeper

zookeeper是一个分布式的应用管理框架，最开始的是作为hadoop的组件之一出现，其主要功能是帮助hadoop集群中维护各个组件的一致性，比较强调于CAP原则中CP，更加强调于分布式集群中各个节点的数据一致性，在比较极端情况下，zookeeper中可能会出现由于为优先保证各节点数据一致性，而忽略其服务可用性的情况，导致调用方暂时无法访问到服务，需要使用者去考虑才有**fast fail**之类的策略以保障服务可用性

zookeeper工作原理是共享配置状态，在每个集群中**选举出领袖**，客户端可以连接到任何一台服务器获取数据

- Eureka

我司PAAS采用的组件之一？，由netflixOSS开源，spring集成到spring cloud框架中；eureka的设计更加强调的AP原则，更多去保证服务发现的可用性，并不能保证服务信息的有效性；另外，eureka同时实现了前面提到的客户端负载均衡的功能，通过ribbon组件帮助客户端在获取到了服务实例信息之后，调用实例

- consul

服务化集中配置工具之一，作为一个K-V键值对，也可以做服务注册与发现

- etcd

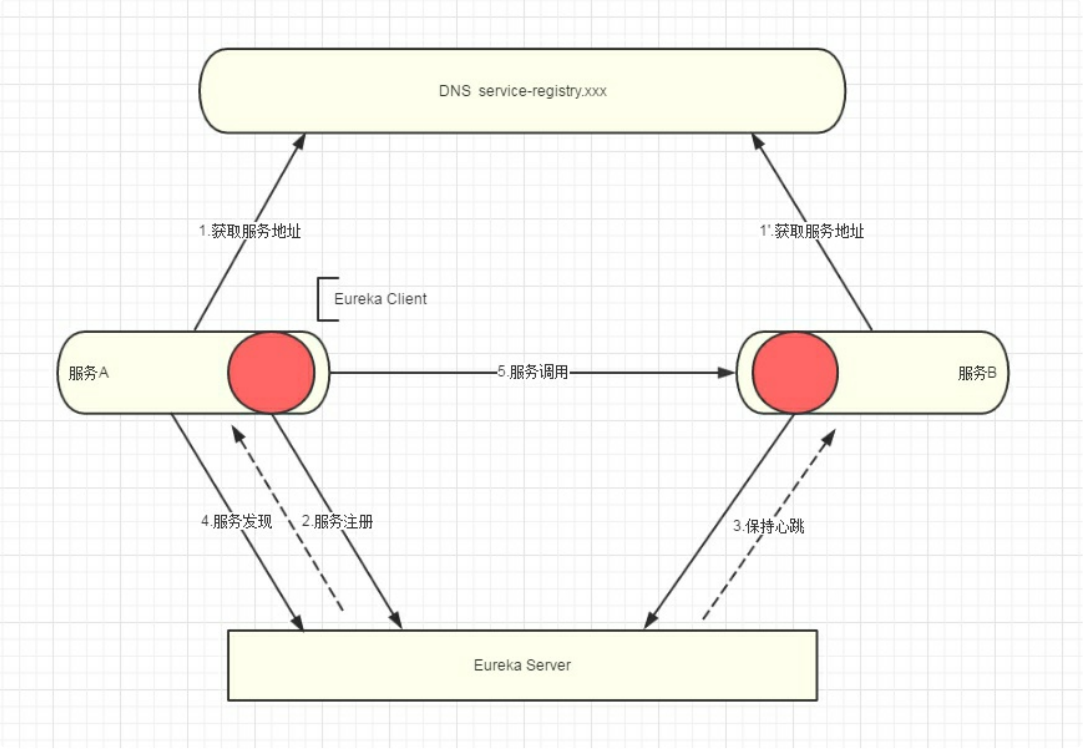
我司PAAS组件之一，K-V键值对存储

	说明	实现语言
Doozer	基于Paxos 协议，使用Go 语言实现，强一致性	Go
Etcd	受 Zookeeper 和 Doozer 启发，使用Raft协议实现	Go
Consul	基于Raft 协议。提供可以直接使用的工具	Go
SmartStack	来自Airbnb，依赖Zookeeper和HAproxy	Ruby

1.4 spring cloud Eureka实现服务注册与发现

1.4.1 eureka的主要组件

1.4.2 eureka的工作原理



1.4.3 服务注册与发现代码实现

1.5 Eureka源码剖析

1.6 遗留问题

SOA/web-service/EBS

zookeeper工作原理

Dubbo简单实例

yaml语法示例

eureka的组件和具体实现过程

eureka作为注册中心可能出现的问题？

1.7 参考

2.集中化配置中心

2 集中配置管理

整个服务的交付过程中依赖于大量的配置文件，这些配置涉及到一个服务的方方面面，主要包括以下几类

- **基础设施的连接信息** - 这块主要是针对一些服务依赖的基础信息，如数据库的连接地址，连接池的大小，连接超时配置等
- **协作服务的连接信息** - 例如服务调用方依赖的服务的地址端口信息，在服务注册与发现的过程，调用方需要知道注册中心的地址
- **影响业务行为的参数** - 例如某些参数可能需要在业务运行过程中动态调整
- **业务特性的功能开关** - 当开发流程中，有功能重大，但是影响范围比较小的特性需要做的时候，可以通过在配置文件中引入一个toggle，当开关打开时，代码才调用该特性；这种做法可以不用在重新拉分支，避免因为特性比较复杂，开发周期长导致分支合入主干时，差异过大，简化版本管理
- **开发环境/生成环境的差异配置**

2.1 应用配置管理总结

2.1.1 常见配置管理方式

- **与服务在同一个包中**

将配置文件和代码一起打包，将配置文件保存在classpath内，通常如jar包中对应的resource目录下文件；这种方式在比较灵活的场景下，会比较被动；修改配置文件可能需要重新编译构建，重新发布，重启等操作；不太符合快速响应的理念

springboot的application.properties在这种方式上提供了一定的灵活性，通过指定不同 `profile` 可以做到在不同场景下调用不同的配置信息

- **与服务分离，保存在单独的配置文件中**

将配置文件从应用中剥离开，例如springboot 可以在jvm的命令行中指定参数将对应配置信息传入

```
java -jar xxxxx.jar   --spring.config.location=xxx
```

- **通过环境变量方式控制**

环境变量的方式比较灵活，可以根据不同场景自定义；但是当有大量的配置参数时又比较高的管理成本，很难做版本控制，并且在环境启动之后，对环境变量比较难跟踪

- **使用云平台提供能力**

目前很多paas平台都提供单独的配置管理功能，利用该功能可以向服务提供配置信息；但是该方式和平台的耦合度比较高

2.1.2 配置管理实现思路

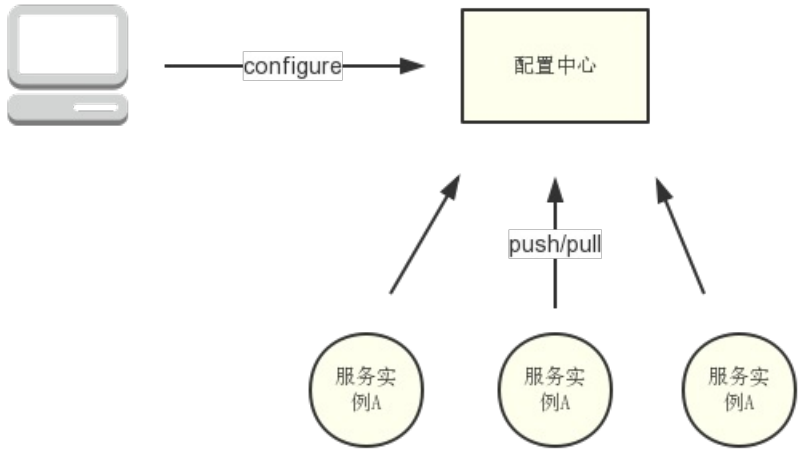
在配置管理中，问题主要有以下几点：

- 针对单个实例，如何能够将配置的改动做到动态更新？
- 多实例时，各个实例之间配置信息如何同步？
- 如何对配置信息进行版本管理，可以有效地升级，回滚，跟踪变化？
- 配置中心如何保证自己的可用性，有哪些实用的容错机制？

针对这些问题，一个通用的配置中心需要提供一下功能

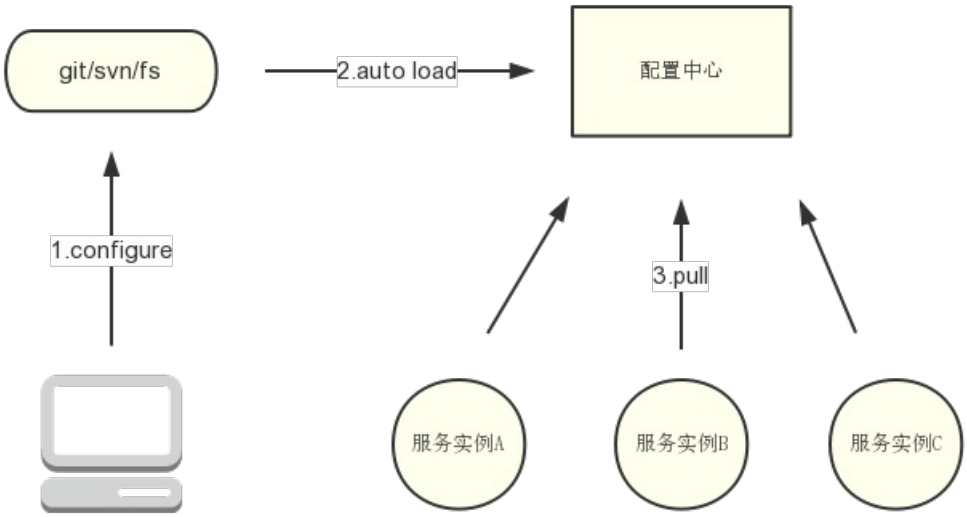
- 具有版本管理功能的存储
- 配置信息的pull/push机制
- 可用性保障及fast fail机制

整体功能如下图所示，配置动态的存储在配置服务器中，用户可以通过一个配置界面去动态调整配置文件，为了做到版本控制以及回滚可以通过git管理配置文件；在配置更新之后，各个服务实例可以通过**定期拉取(Scheduled Pull)**的方式或者**服务器推送(Server-side Push)**的方式更新动态配置，拉方式比较可靠，但会有延迟同时有无效网络开销(假设配置不常更新)，服务器推方式能及时更新配置，但是实现较复杂，一般在服务和配置服务器之间要建立长连接；可用性的保证一般采用负载均衡加多实例，以及针对配置中心服务不可用，通过快速失效避免长时间阻塞



2.2 Spring Cloud Config使用

spring cloud config的实现思路基本和上图一致，允许通过git/svn/本地文件系统作为配置文件存储；config client采用主动拉取的方式更新配置，对外提供rest接口用户通过该rest接口触发更新；多实例之间可以通过消息队列同步到各个实例，目前支持redis、kafka、rabbitMQ等



2.2.1 具体配置

2.2.1.1 config-server配置

第一步，先引入spring cloud config依赖，具体配置如下

```
{
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'

jar {
    baseName = 'config-server'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
    compile('org.springframework.cloud:spring-cloud-config-server') //增加config server依赖
    compile("org.springframework.cloud:spring-cloud-starter-bus-amqp")
    testCompile('org.springframework.boot:spring-boot-starter-test')
```

```
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Brixton.SR6" //增加spring cloud
    }
}
```

第二步，配置对应配置文件存储方式及地址

- 使用github

```
spring:
  cloud:
    config:
      server:
        git:  #svn 方式类似
          uri: file://${user.home}/Work/training/microservice-online-training/training/training-code/centr
alized-config/config-repo
          username:username
          password:password
```

- 使用本地文件系统

```
# 如果使用设置native属性，config server会默认从src/main/resource下检索配置文件
spring.profiles.active=native
# 或者使用如下配置方式
spring.cloud.config.server.native.uri=file://${user.home}/Work/training/
```

第三步，在config server的application代码中添加 `@EnableConfigServer` 注解

```
@SpringBootApplication
@EnableConfigServer // 使得该应用具有config server的能力
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

第四步，定义配置文件仓库config-repo中的配置文件，在仓库中可以定义多个配置文件，一般配置文件规则如下

```
{label}/{application}-{profile}.{yml|properties}
label 表示版本控制信息，git默认的为master，svn默认为trunk
application 为config-client对应的应用名
profile 表示config-clien对应应用在启动时激活的profile
```

第五步，访问config server对外提供的rest接口，一般接口的规则如下

```
http://{server-host}:{server:port}/{application}/{profile}/[{label}]}
```

2.2.1.2 config-client配置

第一步，增加对应依赖

```
{
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'

jar {
    baseName = 'event-composite-service'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.springframework.cloud:spring-cloud-starter-config")
    compile("org.springframework.cloud:spring-cloud-config-client")
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Brixton.SR6"
    }
}
```

第二步，配置config-server地址

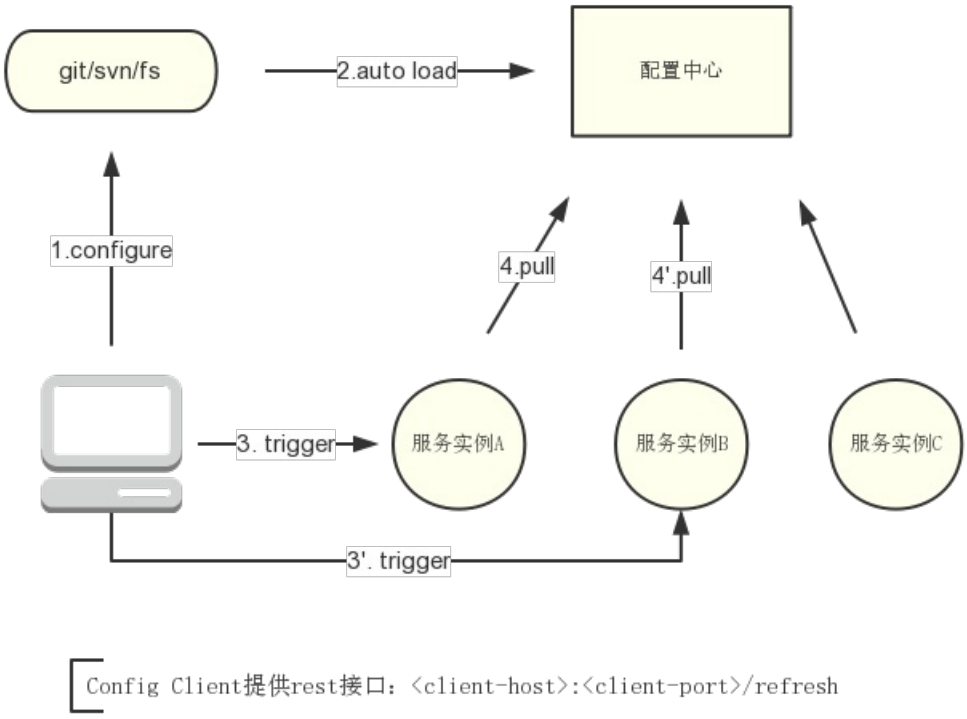
```
# 在对应application的bootstrap配置文件中添加config-server的地址，表示在服务启动前就指定对应server
spring:
    application:
        name: config-client
cloud:
    config:
        uri: http://{server-host}:{server:port}
```

第三步，在需要使用配置信息的变量上增加 `@Value("${configuration.xxxx}")` 获取对应配置值

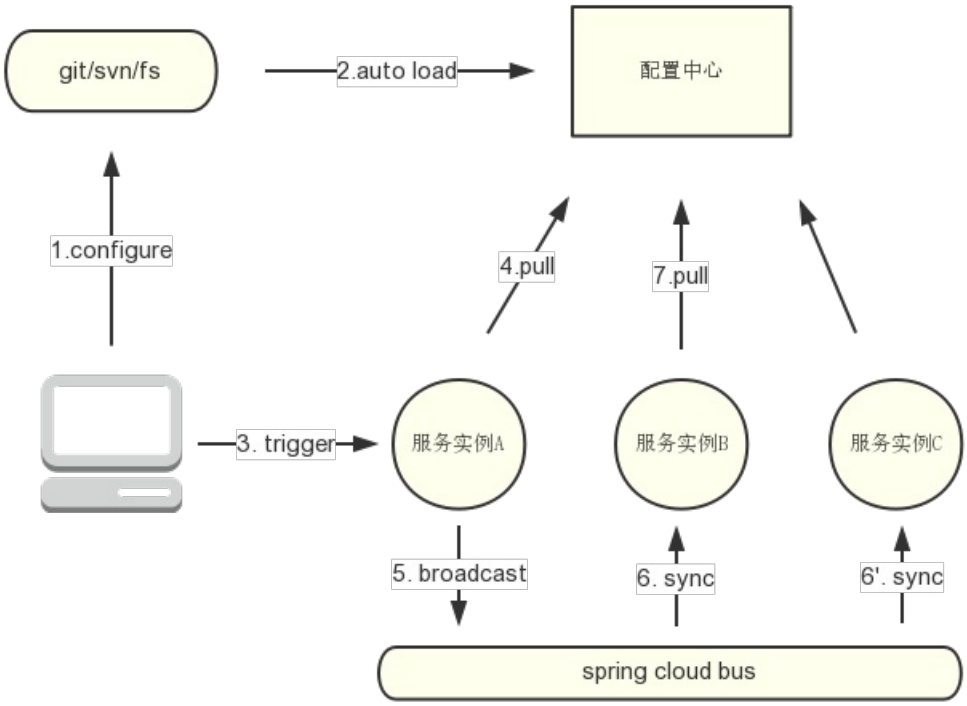
```
@Value("${configuration.https.enable}")
boolean httpsEnable; //通过配置文件判断是否采用https协议
```

第四步，当config-server包含多个配置文件是，获取到对应配置的优先级可以参考如下示例

2.2.2 单实例和多实例更新



针对多实例配置同步的情况，spring还提供了spring cloud bus通过消息代理的方式将配置同步到各个实例



2.2.3 如何确保配置中心的可用性

常用保证可用性方法有

- 负载均衡+多实例
- 启用fast fail机制，通过配置spring.cloud.config.failFast=true，默认为关闭
- 在config-client端增加默认配置，当config-server超时，选择默认配置

2.3 Spring Cloud Config源码分析

2.4 遗留问题

2.5 参考

实施微服务，我们需要哪些基础框架

Spring Cloud构建微服务架构（四）分布式配置中心

Spring Cloud构建微服务架构（七）消息总线

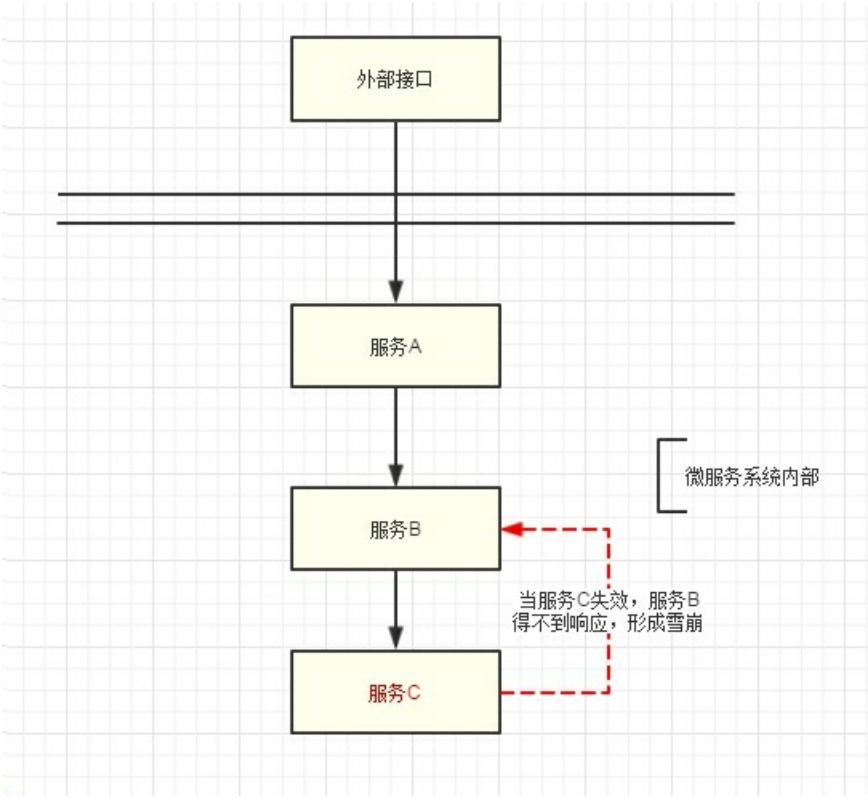
3.熔断器机制

3 熔断器机制

针对分布式系统常常出现的一种情况是由于某个服务不可用导致整个系统不可用，微服务作为一个分布式系统架构，一样也面临这种问题。

3.1 雪崩效应(故障蔓延)

由于微服务架构，按照独立部署的原则，通常都是以独立的进程存在，如果需要和其他服务进行通信，一般通过请求对应的注册服务器以**服务注册与发现**的方式进行；这种模式当服务提供者出现异常或者网络异常时，会导致服务调用方一直得不到响应，根据一般的线程池模型(*之后分析下httpclient与okhttp的线程池代码*)实现会导致调用方阻塞，线程被阻塞，资源得不到释放，从而导致上级调用方出现异常，形成故障蔓延(Cascade Failure)，又称为雪崩效应，下图是一个简单示例。



服务雪崩效应是一种因**服务提供者**的不可用导致**服务调用者**的不可用,并将不可用逐渐放大 的过程.

3.2 雪崩效应形成原因

简化服务调用链，单纯只考虑服务的调用双方，在**服务调用方**，**服务提供方**，**通信机制**这3块都有可能会出现异常，雪崩效应可能发生在各个节点，各个阶段；可能有以下原因

- 硬件故障：硬件故障可能为硬件损坏造成的服务器主机宕机， 网络硬件故障造成的服务提供者的不可访问
- 缓存击穿：缓存击穿一般发生在缓存应用重启， 所有缓存被清空时,以及短时间内大量缓存失效时. 大量的缓存不命中， 使请求直击后端,造成服务提供者超负荷运行,引起服务不可用.（*在使用缓存的时候注意考虑这点*）

- 用户请求激增：在秒杀和大促开始前,如果准备不充分,用户发起大量请求也会造成服务提供者的不可用.
- 重试机制：在服务提供者不可用后，用户由于忍受不了界面上长时间的等待,而不断刷新页面甚至提交表单；以及一般在服务调用端会存在大量服务异常后的重试逻辑；这些情况都会导致重试流量加大
- 程序bug

以上这些点都会导致服务崩溃，当服务调用者使用**同步调用**时，会产生大量的等待线程占用系统资源．一旦线程资源被耗尽,服务调用者提供的服务也将处于不可用状态，于是服务雪崩效应产生了

3.3 雪崩效应的处理方法

针对雪崩效应，一般针对以下几个点进行处理

- 流量控制

流量控制 的具体措施包括：
- 网关限流：目前很多项目就通过nginx+lua在入口，对流量进行限制，结合之前分析的nginx rate limit等以及开源OpenResty方案
- 用户交互限流：限制用户在重试时的频率，考虑在前端的点击按钮对用户进行限制
- 关闭重试：取消调用方的重试机制?

- 改进缓存模式

通过缓存预加载，尽量采用异步方式进行缓存刷新，减少缓存击穿的可能

- 自动扩容机制

通过监控机制，指定对应的指标，当服务达到阈值时，对服务进行动态扩容；典型的如AWS的autoScalling机制，利用cloud watch监控当前实例的各项指标

- 服务调用者降级服务

服务降级指的是在服务出现状况时，就算对外体现是服务质量下降，也要保证生产环境任然能够运行；只要能够工作，就是万幸。(区分服务降级和服务断路的区别)
服务降级可以考虑3点，首先，对服务调用方的资源进行隔离，使用不同的线程去调用不同的服务，避免大家共用同一资源，从而当某个服务崩溃时，不会影响其他服务；
其次，我们根据具体业务,将依赖服务分为：强依赖和若依赖．强依赖服务不可用会导致当前业务中止,而弱依赖服务的不可用不会导致当前业务的中止；
最后，针对不同的服务，可以用不同的降级措施。

常见的降级措施包括了限时(fast fail)、短路、预留退路(fallback)

限时
针对服务超时，可以通过超时控制保证接口的返回，可以通过设置超时时间为1s，尽快返回结果，因为大多数情况下，接口超时一方面影响用户体验，一方面可能是由于后端依赖出现了问题，如负载过高，机器故障等。互联网实践中通常采用，当系统故障时，fail fast。

电路熔断
这里的电路熔断是对于后端服务的保护，当错误、超时、负载上升到一定的高度，那么负载再高下去，对后端来说肯定是无法承受，好像和电路熔断一样

，这三个因素超过了阈值，客户端就可以停止对后端服务的调用，这个保护的措施，帮助了运维人员能迅速通过增加机器和优化系统，帮助系统度过难关。

fallback
有些情况下，即使服务出错，对用户而言，也希望是透明的，无感的，设置一些fallback，做一些服务降级，保证用户的体验，即使这个服务实际上是挂掉的，返回内容是空的或者是旧的，在此故障期间，程序员能赶紧修复，对用户几乎没有造成不良体验。

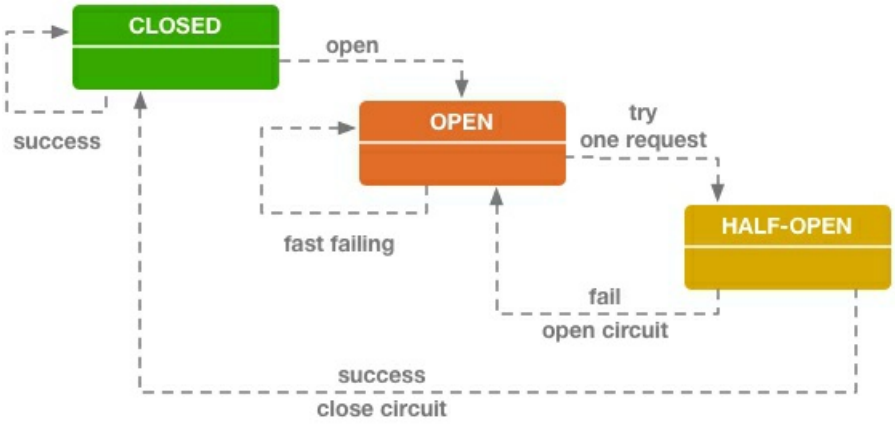
3.3.1 断路器原理

服务的健康状况 = 请求失败数 / 请求总数. 熔断器开关由关闭到打开的状态转换是通过当前服务健康状况和设定阈值比较决定的.

当熔断器开关关闭时, 请求被允许通过熔断器. 如果当前健康状况高于设定阈值, 开关继续保持关闭. 如果当前健康状况低于设定阈值, 开关则切换为打开状态.

当熔断器开关打开时, 请求被禁止通过.

当熔断器开关处于打开状态, 经过一段时间后, 熔断器会自动进入半开状态, 这时熔断器只允许一个请求通过. 当该请求调用成功时, 熔断器恢复到关闭状态. 若该请求失败, 熔断器继续保持打开状态, 接下来的请求被禁止通过.



Circuit Breaker State Diagram

3.4 netflix hystrix使用

Netflix Hystrix 是一个帮助解决分布式系统交互时超时处理和容错的类库, 它同样拥有保护系统的能力.

3.4.1 hystrix的设计原则

Hystrix的设计原则包括:

资源隔离

熔断器

命令模式

3.5 hystrix源码剖析

3.6 遗留问题

3.7 参考

Spring Cloud构建微服务架构（三）断路器

防雪崩利器：熔断器 Hystrix 的原理与使用

Hystrix 使用与分析

4.微服务网关

4 微服务网关

4.1 微服务设计原则

4.1 接口设计原则

4.1 遗留问题

4.1 参考

5.微服务的部署策略

5 微服务部署策略

5.1 微服务设计原则

5.1 接口设计原则

5.1 遗留问题

5.1 参考

6.微服务测试

6 微服务测试

6.1 微服务设计原则

6.1 接口设计原则

6.1 遗留问题

6.1 参考

7.微服务监控告警

7 微服务日志与监控告警

7.1 微服务设计原则

7.1 微服务调用链跟踪

zapkin

7.1 遗留问题

7.1 参考

8.微服务安全

8 微服务安全

8.1 微服务设计原则

8.1 微服务调用链跟踪

zapkin

8.1 遗留问题

8.1 参考

A.附录

其他知识点串联

docker的基础使用

docker-compose的基础使用

Jenkins搭建完整的持续交付流水线

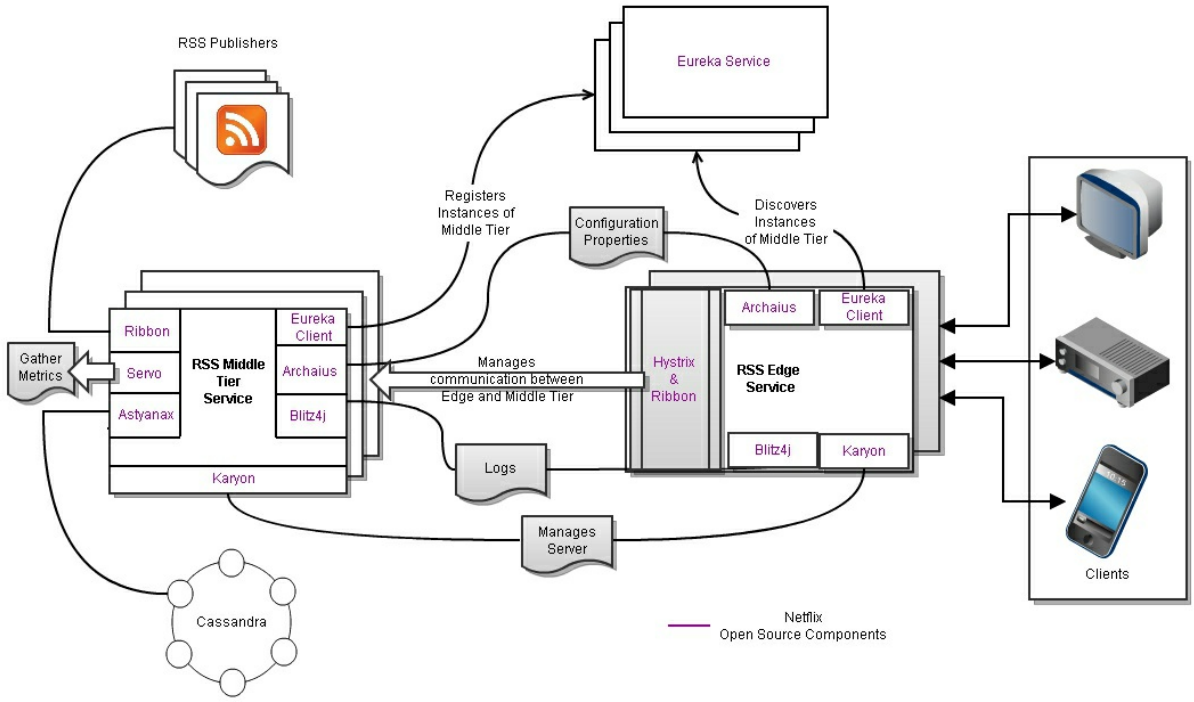
ELK基础使用

prometheus基础使用

Rancher基础使用

安全相关协议汇总分析

ms-with-springcloud



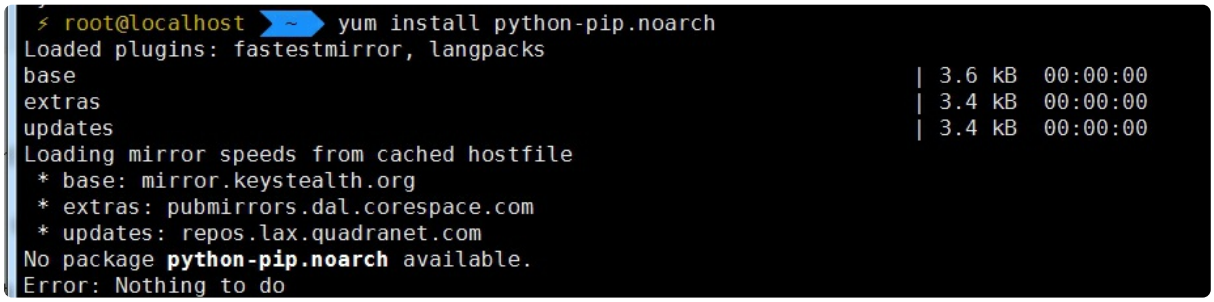
环境搭建

环境搭建

pip安装

在使用centos7时，通过yum安装python-pip时，默认时没有pip安装包，提示如下错误信息

```
yum install python-pip.noarch
Loaded plugins: fastestmirror, langpacks
base                                                     | 3.6 kB  00:00:00
extras                                                   | 3.4 kB  00:00:00
updates                                                  | 3.4 kB  00:00:00
Loading mirror speeds from cached hostfile
 * base: mirror.keystealth.org
 * extras: pubmirrors.dal.corespace.com
 * updates: repos.lax.quadranet.com
No package python-pip.noarch available.
Error: Nothing to do
```



该错误说明没有对应的软件包；这是由于类似于centos之类的linux衍生发行版，源的内容会比较滞后，或者缺少一些扩展源，这时可以通过增加EPEL拓展源解决，EPEL是由 Fedora 社区打造，为 RHEL 及衍生发行版如 CentOS、Scientific Linux 等提供高质量软件包的项目。

所以具体安装步骤为：

- 安装EPEL扩展源

```
yum -y install epel-release
```

```
 ✘ ⚡ root@localhost ~ yum -y install epel-release
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirrors.unifiedlayer.com
 * extras: mirror.pac-12.org
 * updates: centos.sonn.com
Resolving Dependencies
--> Running transaction check
---> Package epel-release.noarch 0:7-9 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package                  Arch               Version          Repository
================================================================================
```

- 安装python-pip

```
yum install python-pip
```

```
 ⚡ root@localhost ~ yum install python-pip
Loaded plugins: fastestmirror, langpacks
epel/x86_64/metalink                                        |  14 kB  00:00:00
epel                                                        | 4.3 kB  00:00:00
(1/3): epel/x86_64/group_gz                                 | 170 kB  00:00:03
(2/3): epel/x86_64/updateinfo                               | 787 kB  00:00:14
(3/3): epel/x86_64/primary_db                               | 4.7 MB  00:00:36
Loading mirror speeds from cached hostfile
 * base: mirror.scalabledns.com
 * epel: mirror.hmc.edu
 * extras: mirror.scalabledns.com
 * updates: centos.sonn.com
Resolving Dependencies
--> Running transaction check
---> Package python2-pip.noarch 0:8.1.2-5.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package                Arch              Version                 Repository        Size
================================================================================
Installing:
```

- 更新pip

```
pip install --upgrade pip
```

```
⚡ root@localhost ~ pip install --upgrade pip
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/ssl_.py:318: SN
IMissingWarning: An HTTPS request has been made, but the SNI (Subject Name Indication) exte
nsion to TLS is not available on this platform. This may cause the server to present an inc
orrect TLS certificate, which can cause validation failures. You can upgrade to a newer ver
sion of Python to solve this. For more information, see https://urllib3.readthedocs.org/en/
latest/security.html#snimissingwarning.
  SNIMissingWarning
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/ssl_.py:122: In
securePlatformWarning: A true SSLContext object is not available. This prevents urllib3 fro
m configuring SSL appropriately and may cause certain SSL connections to fail. You can upgr
ade to a newer version of Python to solve this. For more information, see https://urllib3.r
eadthedocs.org/en/latest/security.html#insecureplatformwarning.
  InsecurePlatformWarning
Collecting pip
  Downloading pip-9.0.1-py2.py3-none-any.whl (1.3MB)
    100% |████████████████████████████████| 1.3MB 195kB/s
Installing collected packages: pip
  Found existing installation: pip 8.1.2
```

- 清楚yum缓存

```
yum clean all
```

```
⚡ root@localhost ~ yum clean all
Loaded plugins: fastestmirror, langpacks
Cleaning repos: base epel extras updates
Cleaning up everything
Cleaning up list of fastest mirrors
⚡ root@localhost ~
```

docker-compose安装

- 通过pip安装docker-compose

```
pip install docker-compose
```

```
⚡ root@localhost ~ pip install docker-compose
Collecting docker-compose
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/ssl_.py:318: SN
IMissingWarning: An HTTPS request has been made, but the SNI (Subject Name Indication) exte
nsion to TLS is not available on this platform. This may cause the server to present an inc
orrect TLS certificate, which can cause validation failures. You can upgrade to a newer ver
sion of Python to solve this. For more information, see https://urllib3.readthedocs.io/en/l
atest/security.html#snimissingwarning.
  SNIMissingWarning
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/ssl_.py:122: In
securePlatformWarning: A true SSLContext object is not available. This prevents urllib3 fro
m configuring SSL appropriately and may cause certain SSL connections to fail. You can upgr
ade to a newer version of Python to solve this. For more information, see https://urllib3.r
eadthedocs.io/en/latest/security.html#insecureplatformwarning.
  InsecurePlatformWarning
  Downloading docker_compose-1.12.0-py2.py3-none-any.whl (91kB)
    100% |████████████████████████████████| 92kB 42kB/s
Collecting cached-property<2,>=1.2.0 (from docker-compose)
  Downloading cached_property-1.3.0-py2.py3-none-any.whl
Collecting backports.ssl-match-hostname>=3.5; python_version < "3.5" (from docker-compose)
```

- 解决安装中的坑。。。

- 验证结果

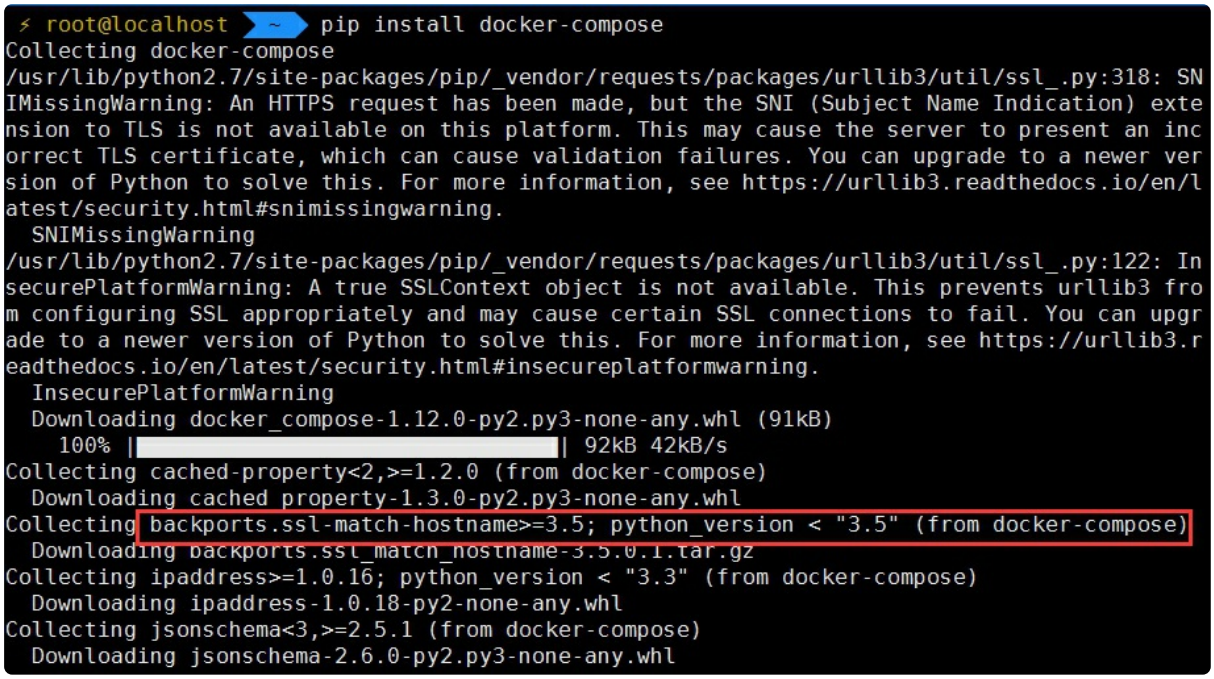
```
docker-compose --version
```

安装问题

问题 1 ：docker-compose运行时，可能会出现如下报错

```
pkg_resources.DistributionNotFound: backports.ssl-match-hostname>=3.5
```

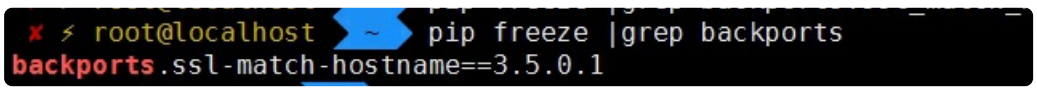
解决方法：问题是由如下截图原因导致，docker-compose要求的backports版本3.5以上，可以通过升级backports版本解决




```
pip install -U backports.ssl_match_hostname
```

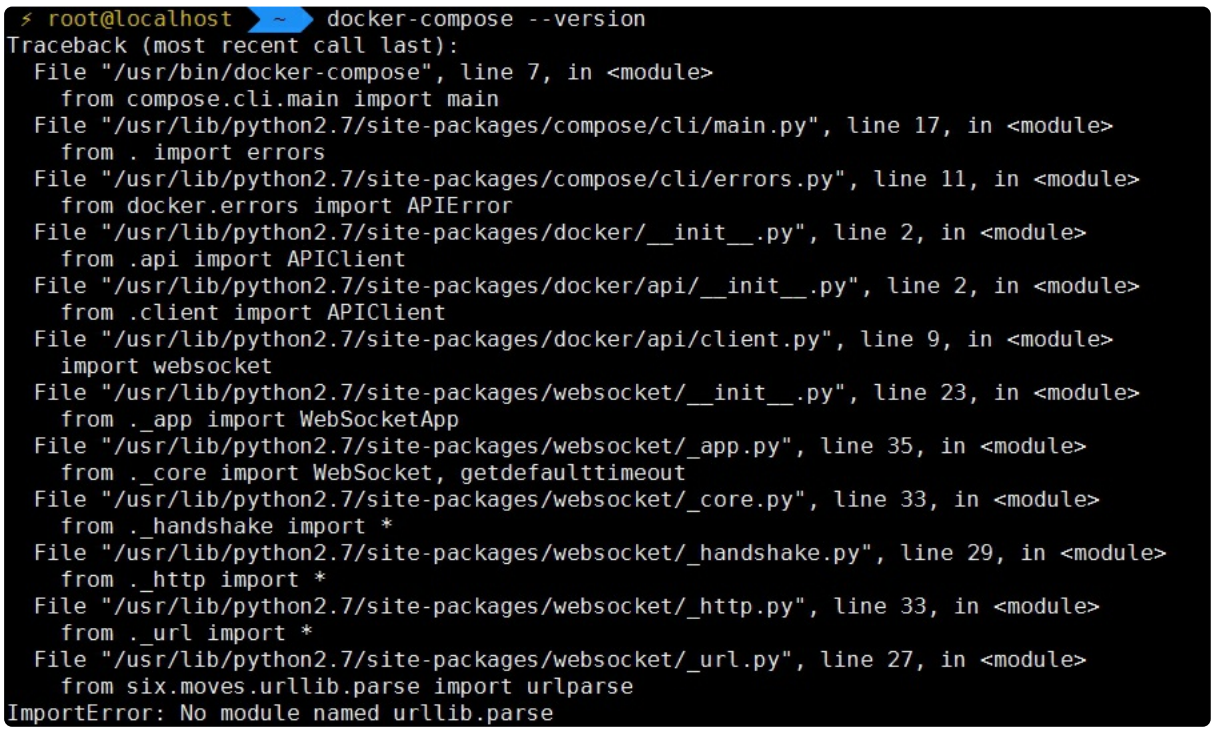
验证后升级后版本

```
pip freeze |grep backports
```



问题2： docker-compose运行期间出现urlprase错误，错误截图如下

```
docker-compose --version
Traceback (most recent call last):
  File "/usr/bin/docker-compose", line 7, in <module>
    from compose.cli.main import main
  File "/usr/lib/python2.7/site-packages/compose/cli/main.py", line 17, in <module>
    from . import errors
  File "/usr/lib/python2.7/site-packages/compose/cli/errors.py", line 11, in <module>
    from docker.errors import APIError
  File "/usr/lib/python2.7/site-packages/docker/__init__.py", line 2, in <module>
    from .api import APIClient
  File "/usr/lib/python2.7/site-packages/docker/api/__init__.py", line 2, in <module>
    from .client import APIClient
  File "/usr/lib/python2.7/site-packages/docker/api/client.py", line 9, in <module>
    import websocket
  File "/usr/lib/python2.7/site-packages/websocket/__init__.py", line 23, in <module>
    from ._app import WebSocketApp
  File "/usr/lib/python2.7/site-packages/websocket/_app.py", line 35, in <module>
    from ._core import WebSocket, getdefaulttimeout
  File "/usr/lib/python2.7/site-packages/websocket/_core.py", line 33, in <module>
    from ._handshake import *
  File "/usr/lib/python2.7/site-packages/websocket/_handshake.py", line 29, in <module>
    from ._http import *
  File "/usr/lib/python2.7/site-packages/websocket/_http.py", line 33, in <module>
    from ._url import *
  File "/usr/lib/python2.7/site-packages/websocket/_url.py", line 27, in <module>
    from six.moves.urllib.parse import urlparse
ImportError: No module named urllib.parse
```



解决方法：具体问题没有定位清楚，通过google找到解决方案，更新websocket库

```
pip install -U websocket
```

```
                                  --no-index.
 ⚡ root@localhost ~ pip install -U websocket
Collecting websocket
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/s
IMissingWarning: An HTTPS request has been made, but the SNI (Subject Name In
nsion to TLS is not available on this platform. This may cause the server to
orrect TLS certificate, which can cause validation failures. You can upgrade
sion of Python to solve this. For more information, see https://urllib3.readt
atest/security.html#snimissingwarning.
  SNIMissingWarning
/usr/lib/python2.7/site-packages/pip/_vendor/requests/packages/urllib3/util/s
securePlatformWarning: A true SSLContext object is not available. This prever
m configuring SSL appropriately and may cause certain SSL connections to fail
ade to a newer version of Python to solve this. For more information, see htt
eadthedocs.io/en/latest/security.html#insecureplatformwarning.
  InsecurePlatformWarning
  Downloading websocket-0.2.1.tar.gz (195kB)
    100% |████████████████████████████████| 204kB 296kB/s
Collecting gevent (from websocket)
  Downloading gevent-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl (1.8MB)
    100% |████████████████████████████████| 1.8MB 79kB/s
Collecting greenlet (from websocket)
  Downloading greenlet-0.4.12-cp27-cp27mu-manylinux1_x86_64.whl (41kB)
    100% |████████████████████████████████| 51kB 190kB/s
Installing collected packages: greenlet, gevent, websocket
  Running setup.py install for websocket ... done
Successfully installed gevent-1.2.1 greenlet-0.4.12 websocket-0.2.1
```

验证结果，运行正常

```
 ⚡ root@localhost ~ docker-compose --version
docker-compose version 1.12.0, build b31ff33
```

参考

ImportError: No module named urllib.parse