**睿普云网络管理系统设计说明书**

**一、引言**

睿普云网络系统包括内网系统，外网系统，智能路由器系统。该系统为家园云提供网络资源分配管理，其中包括为用户分配 vlan，创建内网的，为主机分配内网 ip 和外网 ip，为主机分配内网和外网的 Mac 地址。

**二、软件总体设计**

**2.1软件需求**

本软件主要有以下几方面的功能

1. 创建公网
2. 删除公网
3. 绑定公网
4. 解绑公网
5. 创建内网网段
6. 绑定内网
7. 解绑内网
8. 创建智能路由器
9. 管理路由器网络
10. Mac 地址分配管理

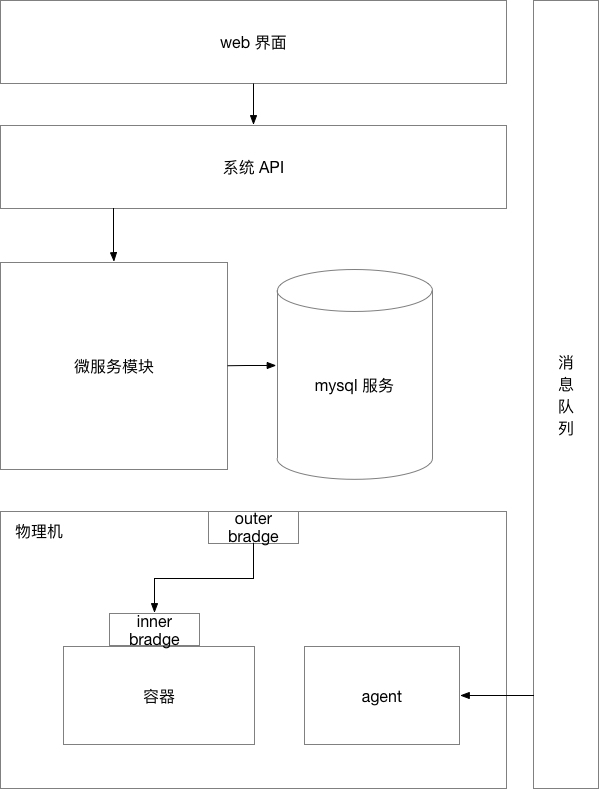
## 2.2条件与限制

开发所用的系统是Linux内核，Centos7.2以及JDK1.8。

硬件环境为CPU：Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz，内存：256GB。

## 2.3总体结构

系统整体结构框架如下图所示，网络接口定义在物理机层和容器之间，在这层实现了容器和物理机网络的打通，实现了外网河网功能 。对于内网局域网管理的智能路由器，则由单独创建的容器实现。



系统整体结构框架图

## 2.4模块功能逻辑关系

系统详细的模块信息所示：

系统详细的模块信息表

|  |  |
| --- | --- |
| **主模块** | |
| 模块名称 | 功能简述 |
| 外网管理 | 实现容器外网的创建 |
| 绑定公网 | 实现容器外网的绑定解绑 |
| 内网管理 | 创建内网局域网端，分配 vlan 和 mac 地址 |
| 绑定内网 | 内网与容器的绑定解绑 |
| 智能路由器 | 智能路由器的创建，网络配置 |
| 智能路由器管理 | 为智能路由器配置网络 |
| 智能路由器开机 | 开启智能路由器 |
| **支撑类模块** | |
| 模块名称 | 功能简述 |
| Mac 管理 | Mac 地址创建分配 |
| Vlan 管理 | Vlan 初始化分配 |

**2.5架构设计**

云网络管理平台的定义是Gartner提出来的，总结起来就是两块，第一是管理，管理公有云、私有云，形成混合云。第二是自服务，镜像划分，计量与计费，负载优化。云网络管理平台最终的目标是应用在云平台上运行时取得最优化的效果。大家熟悉的公有云有上云服务，能最大限度的保证应用的可靠性，我们自己设计一个云网络管理平台时也要考虑这方面，还附加了一些外部系统的对接与管理功能。

说到云计算，大家会想到OpenStack，OpenStack跟云网络管理平台有什么区别，是不是可以基于OpenStack做一些云网络管理平台？目前从我们自己的定义以及市场上的反馈来看，我们认为云网络管理平台是更广的范围。我们可以把OpenStack看作是云网络管理平台下属的资源模块，云网络管理平台可以管理多个OpenStack版本。

有些企业会在不同的数据中心里部署不同的OpenStack资源池管理模块，在OpenStack之上也需要一个云网络管理平台来管理多个OpenStack资源池以及管理不同的OpenStack版本，另外还有很多虚拟化不是由OpenStack+KVM实现的，比如Vmware、Xen等虚拟化，所以云网络管理平台还要针对这种虚拟化做对接。从层次上来看，云网络管理平台是解决用户最后的资源使用一公里的问题，最终提供给用户服务的是云网络管理平台的自服务平台。

****我们实现的云网络管理平台主要分三大功能：****

异构的虚拟化管理。OpenStack天生对接KVM平台，在Xen、hyper-v、VMware对接上是相对劣势的一点， 我们实现的是同等能力的管理。

多个OpenStack版本、多个资源池管理还有多数据中心扩展。

调度、计量计费、外部系统集成。

异构的虚拟化管理。OpenStack天生对接KVM平台，在Xen、hyper-v、VMware对接上是相对劣势的一点， 我们实现的是同等能力的管理。

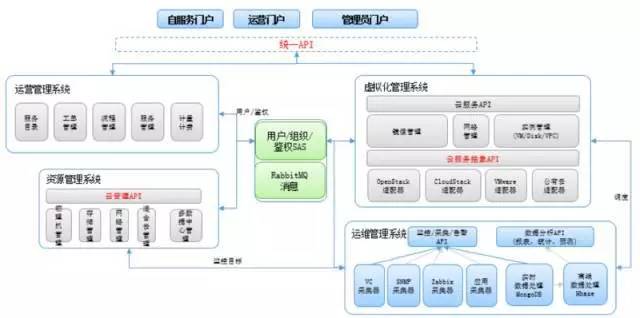
多个OpenStack版本、多个资源池管理还有多数据中心扩展。

调度、计量计费、外部系统集成。

云网络管理平台主要有资源管理、运营、自服务三个功能，资源提供方比如存储、计算、网络。资源池最终交付给用户的是一个一个服务，比如计算服务、存储服务、网络服务等IaaS层面的，中间件服务、数据服务等PaaS层面的。怎么交付给用户的？是云网络管理平台将资源管理起来，通过运营，在公有云上这种运营可以是计量计费，之后通过自服务界面把应用提供给用户使用。

在容器平台的基础架构层面，网络方案的设计一般都是涉及讨论最多，意见碰撞最多的地方，毕竟网络是底层的基础设施，牵一发动全身，所以定会格外谨慎。underlay、overlay、routing的网络模型都比较了遍，当然这些方案都是要Kubernetes CNI（Container Network Interface）插件标准的，主要关注的有：ipvlan（underlay）、Macvlan（underlay）、Flannel（overlay）、Calico（routing）、NSX-T（routing）。  
  
对于underlay的方案，对于传统的网络架构可以无缝适配，网络的管理模式（IP资源管理，流量管理等）也可以保持一致，但从Kubernetes的管理功能和生态圈来看，underlay的网络方案都不是方向，更多是一种适配传统网络架构的过渡方案，对于ip的管理还是要在外部完成，而且Kubernetes也失去了对于容器的隔离控制能力，此外，例如Macvlan要求启用混杂模式，ipvlan要求linux内核版本要在4.1之上等刚性要求，也不适合绝大多数企业的网络管理规范及操作系统使用现状。对于Overlay的方案，对于传统网络和容器通讯及跨Kubernetes集群的容器通讯需求，该类型方案均存在很大弊端，而且基于vxlan的数据封装传输，对于容器ip的流量监控也不易实现（除非支持解析 vxlan 数据包），对于vxlan的解封包，在性能上也会一定损失，性能表现亦不占优。所以，综合应用的通信需求、运维的管理需求、功能的完善度、技术趋势、性能、安全等多方面的因素，我认为routing的网络模型方案更优，routing模式对传统应用适配，接受传统组织结构都更友好。

这是云网络管理整体的定位，它是承上启下的。对下管理资源、对上提供应用使用的界面和API。如果创业公司自己设计一个云网络管理平台的话，五个模块必不可少，一是资源管理，二是运营管理，三是服务提供管理（使用界面与API），还有运维管理与安全管理。



这是我们设计CMP1.0时第一版的架构，拆分一下有资源管理系统、运营管理系统。资源管理系统负责物理机管理、存储管理、网络管理、多数据中心管理。架构图右上角的模块是一个专门的虚拟化管理系统，它也可以看做是一个资源管理模块。

为什么把虚拟化管理系统拆分出来？因为虚拟化管理系统是云网络管理平台一个比较核心的模块，需要对它做更精细的设计，所以单独拆分出来。 运营管理系统最终的目标就是将资源管理系统、虚拟化管理系统管理出来的资源运营成为一种服务，比如服务目录就是将资源发布成云计算产品，工单管理、流程管理、计量计费负责用户申请一个虚拟机或一块存储空间，怎么做计量计费，怎么发布出去。

再往上是用户使用的自服务，运营门户、管理员门户，当然也要开放统一API，因为有些应用使用的不是自服务界面，而是使用API调用虚拟化、存储、网络。要做好一个系统，运维管理，比如监控还有日志分析等等，是必不可少的。在云网络管理平台之外我们单独设计一个运维管理系统，通过Zabbix等采集模块采集系统运行状态，实时呈现给云网络管理平台的管理员，这是CMP总体架构设计的1.0版本。

**资源管理系统**

首先看资源管理系统。 虚拟化管理系统可以看作是资源管理系统的子模块。目前市场上常用的虚拟化大的用户里最多的是VMware。我们统计过电信运营商、银行等等，VMware占有量大概在80%以上。第二个是中小企业，用微软hyper-v的比较多，还有citrix、kvm、苏研虚拟化。

我们在做虚拟化管理系统时是做虚拟化适配层，针对每一个虚拟化平台做一个driver，这个driver可以下发指令、并反馈状态，同时开放API针对管理层使用。比如虚拟机的基础管理，像开机关机都可以通过driver做。在基于KVM去做的时候引用了OpenStack虚拟化管理的nova模块，主要目的是有了nova我们也没有必要完全重写针对KVM虚拟化管理的功能。

我们在设计私有云的时候，用户对应用支撑这块有很高的要求，比如数据库服务，他会明确要求放在物理机上。 所以资源管理模块需要做物理机管理，目前是基于IPMI实现的，可以做物理机的分配、自动安装、自动服务提供。跟虚拟化的区别就是物理机的分配单位是一台一台的物理机，虚拟化的分配单位可以在物理机上做更小的切分。

还有存储模块。目前说云计算的时候大家一提存储往往是提分布式存储或者云存储，类似对象存储的实现。如果在传统企业做存储管理并将存储形成存储服务的话，还有一块就是传统存储，也叫SAN存储，包括FC存储和iSCSI存储。这种存储管理的实现我们是通过标准的SMIS协议，如果没有这个协议就要对接cmd line/shell脚本实现。

资源管理的网络管理。云计算里面经常介绍SDN的管理，除此之外还有路由、交换机、防火墙，这块的管理需要在上面自动划分网络服务，划分一个VPC或者子网，需要通过网络管理模块API做对接和呈现。

**服务容错**

微服务之间存在错综复杂的依赖关系，一次请求可能会依赖多个后端服务，在实际生产中这些服务可能会产生故障或者延迟，在一个高流量的系统中，一旦某个服务产生延迟，可能会在短时间内耗尽系统资源，将整个系统拖垮，因此一个服务如果不能对其故障进行隔离和容错，这本身就是灾难性的。**我们的微服务架构中使用了Hystrix组件来进行容错处理。**Hystrix是Netflix的一款开源组件，**它通过熔断模式、隔离模式、回退（fallback）和限流等机制对服务进行弹性容错保护，保证系统的稳定性。**

1. **熔断模式**：熔断模式原理类似于电路熔断器，当电路发生短路时，熔断器熔断，保护电路避免遭受灾难性损失。当服务异常或者大量延时，满足熔断条件时服务调用方会主动启动熔断，执行fallback逻辑直接返回，不会继续调用服务进一步拖垮系统。熔断器默认配置服务调用错误率阀值为50%，超过阀值将自动启动熔断模式。服务隔离一段时间以后，熔断器会进入半熔断状态，即允许少量请求进行尝试，如果仍然调用失败，则回到熔断状态，如果调用成功，则关闭熔断模式。
2. **隔离模式**：Hystrix默认采用线程隔离，不同的服务使用不同的线程池，彼此之间不受影响，当一个服务出现故障耗尽它的线程池资源，其他的服务正常运行不受影响，达到隔离的效果。例如我们通过andThreadPoolKey配置某个服务使用命名为TestThreadPool的线程池，实现与其他命名的线程池隔离。
3. **回退（fallback）**：fallback机制其实是一种服务故障时的容错方式，原理类似Java中的异常处理。只需要继承HystixCommand并重写getFallBack()方法，在此方法中编写处理逻辑，比如可以直接抛异常（快速失败），可以返回空值或缺省值，也可以返回备份数据等。当服务调用出现异常时，会转向执行getFallBack()。有以下几种情况会触发fallback：

程序抛出非HystrixBadRequestExcepption异常，当抛出HystrixBadRequestExcepption异常时，调用程序可以捕获异常，没有触发fallback，当抛出其他异常时，会触发fallback；

程序运行超时；

熔断启动；

线程池已满。

1. **限流**： 限流是指对服务的并发访问量进行限制，设置单位时间内的并发数，超出限制的请求拒绝并fallback，防止后台服务被冲垮。

Hystix使用命令模式HystrixCommand包装依赖调用逻辑，这样相关的调用就自动处于Hystrix的弹性容错保护之下。调用程序需要继承HystrixCommand并将调用逻辑写在run()中，使用execute()（同步阻塞）或queue()（异步非阻塞）来触发执行run()。

**动态配置中心**

微服务有很多依赖配置，某些配置参数在服务运行期间可能还要动态修改，比如：根据访问流量动态调整熔断阀值。传统的实现信息配置的方法，比如放在xml、yml等配置文件中，和应用一起打包，每次修改都要重新提交代码、打包构建、生成新的镜像、重新启动服务，效率太低，这样显然是不合理的，因此我们需要搭建一个动态配置中心服务支持微服务动态配置。

我们使用Spring Cloud的**configserver服务帮我们实现动态配置中心的搭建。**我们开发的微服务代码都存放在Git服务器私有仓库里面，所有需要动态配置的配置文件存放在Git服务器下的configserver（配置中心，也是一个微服务）服务中，部署到Docker容器中的微服务从Git服务器动态读取配置文件的信息。

当本地Git仓库修改代码后push到Git服务器仓库，Git服务端hooks(post-receive，在服务端完成代码更新后会自动调用)自动检测是否有配置文件更新，如果有，Git服务端通过消息队列给配置中心（configserver，一个部署在容器中的微服务）发消息，通知配置中心刷新对应的配置文件。这样微服务就能获取到最新的配置文件信息，实现动态配置。

以上这些框架或组件是支撑实施微服务架构的核心，在实际生产中，我们还会用到很多其他的组件，比如日志服务组件、消息服务组件等等，根据业务需要自行选择使用。在我们的微服务架构实施案例中，参考使用了很多Spring Cloud Netflix框架的开源组件，**主要包括Zuul（服务网关）、Eureka（服务注册与发现）、Hystrix（服务容错）、Ribbon（客户端负载均衡）等。**这些优秀的开源组件，为我们实施微服务架构提供了捷径。

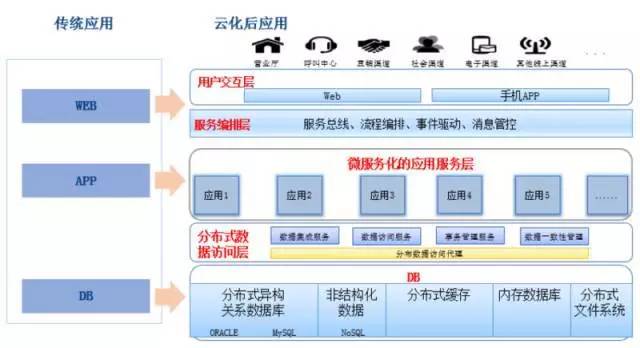
最后是多数据中心管理。比如要帮一个公有云的客户实现一个云网络管理平台，这个公有云可能会在全国部署多个数据中心。涉及到一个云网络管理平台对多个数据中心的管理，这是一种分布式架构的实现。比如说在数据中心里把OpenStack或者vCenter作为一个资源池模块，最终是一个云网络管理平台去管理多个数据中心，并做整体平台的运营和服务。

刚才简单的给大家总结了资源管理这一部分，包括计算资源、存储资源、网络资源的统一管理。

**运营管理**

管理完之后是运营的过程。运营包括将管理完的资源发布成服务，需要一种服务模板的规划设计。服务模板包括把一个虚拟机定义成一个服务，包括虚拟机的定义、发布、审核以及在界面上的呈现。在服务目录之后，我们有了服务的概念，还要做计量计费，用户使用资源时要做统计，公有云上可以收费用，私有云可以做跨部门之间的资源使用统计，这是订单管理和用户管理、资源使用计量计费三块合一，也就是要确定一个用户或者一个部门使用的资源时间时长以及最终的计量计费的过程。最后是运营门户，运营管理员可以运营这些服务。比如公有云厂商定期有新的云服务发布，基本是在运营管理层面上做的。

我们做了一个总结， 应用在使用时主要可以分为两种类型：实时交易类型和在线批处理。这两种应用迁移到云平台的时候都有自己的特定要求。



首先看传统的交易系统，比如电商系统或者是CRM人力资源管理系统，它分三层：Web、App、DB。以前中大型的企业，比如银行或者运营商都是放在小型机上承载的，Web用X86服务器，App可能用weblogic、websphere放在小型机上运行，还有DB用Oracle或者DB2的数据库，都放在小型机上。

阿里一直在提的去IOE，实际上是一种分布式架构改造，改造之后传统的Oracle数据库、DB2数据库，逐步去做分布式数据库或者是关联不复杂的数据逐步往非结构化数据、内存数据库做迁移，还有NoSQL数据库，实际上数据库是在逐步做新技术的引入，实现模块化分布式改造。

如果我们把原先单一的数据库改造成分布式数据库，需要分布式数据库的数据路由和集中访问。到应用这层，以前我们说weblogic、websphere，现在就改成微服务，比如一个一个的应用，每个应用单一化改造成单一的服务。有了微服务之后，在微服务上层还需要微服务的治理，比如服务编排、服务访问路由。再上层就是用户交互层。

**传统应用云化改造对云网络管理平台功能设计的新需求**

传统的应用在改造成云化架构时带来了很多需求：

第一个关键点是Web接入，比如从单机变成负载均衡+后端多节点。从客户流量变化上大家都知道有弹性伸缩，节点数量随着流量变化而变化。

第二是X86集群部署，小型机的年代是单机或者是HA的部署，到了云化的时候，从Web到App到数据层，都是x86集群化部署，所以要有集群管理功能。

第三是数据分布式部署，原先的单一数据库在存储几亿条数据之后，查询交易上延迟性和响应上比较大，所以会考虑数据库的拆分，比如数据库表的垂直拆分或者水平拆分。拆分之后会带来新的问题，比如数据对外的统一访问，原先是一个表如果拆成多个表，每个表跨库join、跨表join都是新的问题，所以在云平台上要构建数据路由层或者统一数据访问引擎。开源的有Hibernate、Shards、Ibatis-Sharding，但是开源的访问引擎如果集成到云平台上，云平台对他们的监控、调用逻辑的呈现是必要的需求。

第四是数据平台化。以前App跟数据库逻辑一般是绑死的，比如在数据库里写一些存储过程，这种存储过程往往跟中间件层有很强的逻辑关联性。微服务化之后，需要后台单一的每个不同的数据库存储，需要有集中的存储平台的概念，最终要做存储分析、订单的历史趋势呈现，都需要数据平台来做。

还有一块是联机分析处理。以前做数据仓库就是大的Oracle数据仓库，Hadoop技术出现之后大家就想用Hadoop做Oracle的数据仓库功能。最开始的架构就是在原先的Oracle DW之外，逐步引入Hadoop技术，如果要做关联复杂的历史数据分析，比如银行的复杂的用户数据画像需要MPP数据库，如果不复杂也可以用Hadoop做。最开始的联机处理分析引入的是Hadoop并列割裂式的架构。如果需要在Hadoop分析，可以把数据仓库上的数据导到Hadoop上进行分析。这种方式有一个问题，相互独立的信息系统缺乏数据交互性，演进路线受限。后来大家提企业级数据平台，是从ETL层到数据汇总，统一做拉开。

第一个关键点是Web接入，比如从单机变成负载均衡+后端多节点。从客户流量变化上大家都知道有弹性伸缩，节点数量随着流量变化而变化。

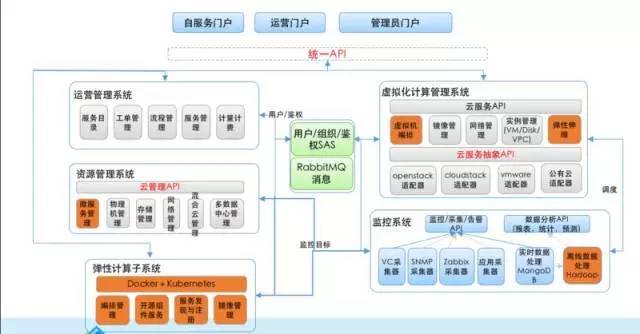
第二是X86集群部署，小型机的年代是单机或者是HA的部署，到了云化的时候，从Web到App到数据层，都是x86集群化部署，所以要有集群管理功能。

第三是数据分布式部署，原先的单一数据库在存储几亿条数据之后，查询交易上延迟性和响应上比较大，所以会考虑数据库的拆分，比如数据库表的垂直拆分或者水平拆分。拆分之后会带来新的问题，比如数据对外的统一访问，原先是一个表如果拆成多个表，每个表跨库join、跨表join都是新的问题，所以在云平台上要构建数据路由层或者统一数据访问引擎。开源的有Hibernate、Shards、Ibatis-Sharding，但是开源的访问引擎如果集成到云平台上，云平台对他们的监控、调用逻辑的呈现是必要的需求。

第四是数据平台化。以前App跟数据库逻辑一般是绑死的，比如在数据库里写一些存储过程，这种存储过程往往跟中间件层有很强的逻辑关联性。微服务化之后，需要后台单一的每个不同的数据库存储，需要有集中的存储平台的概念，最终要做存储分析、订单的历史趋势呈现，都需要数据平台来做。

还有一块是联机分析处理。以前做数据仓库就是大的Oracle数据仓库，Hadoop技术出现之后大家就想用Hadoop做Oracle的数据仓库功能。最开始的架构就是在原先的Oracle DW之外，逐步引入Hadoop技术，如果要做关联复杂的历史数据分析，比如银行的复杂的用户数据画像需要MPP数据库，如果不复杂也可以用Hadoop做。最开始的联机处理分析引入的是Hadoop并列割裂式的架构。如果需要在Hadoop分析，可以把数据仓库上的数据导到Hadoop上进行分析。这种方式有一个问题，相互独立的信息系统缺乏数据交互性，演进路线受限。后来大家提企业级数据平台，是从ETL层到数据汇总，统一做拉开。

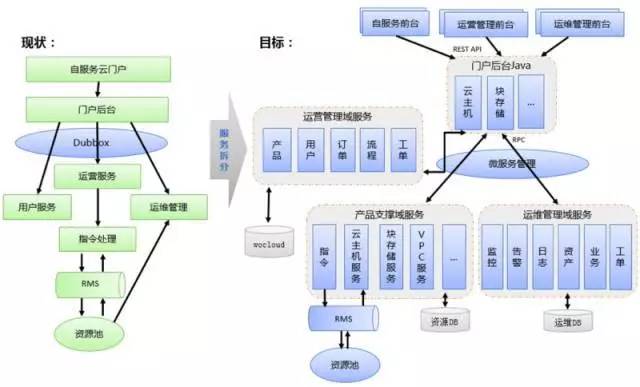
总结一下交易系统跟批处理系统在上云过程中对云网络管理平台的新的需求。第一是弹性计算，Web节点实时弹性伸缩。第二是App中间件层逐渐引入微服务，微服务架构有很多，设计出的微服务也很多，需要对微服务做统一管控。第三是越来越多的模块，需要对产生的数据监控做更多分析。比如一个用户三年前上云，现在数据中心积累了海量的监控数据，监控数据对他也是很有用的，可以做历史趋势的分析。第四是需要支持大数据类、开源数据库类的开源组件对接与管理。



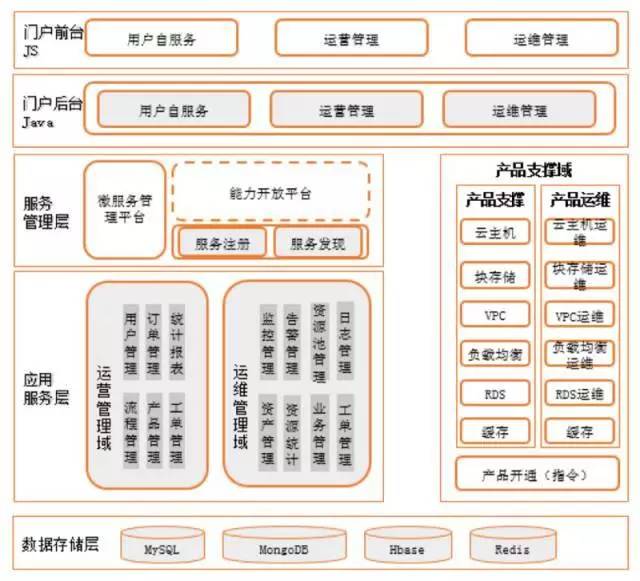
我们做了几点，第一是虚拟机编排跟弹性伸缩。大家知道Docker有自带的弹性伸缩，在虚拟机这一层也需要弹性伸缩。第二是资源管理系统引入了微服务管理。第三是开源组件的支持，引入了Docker+Kubernetes，把hadoop和spark的一些内容往弹性计算体系迁移，这样云网络管理平台新的针对应用的设计演进了一步。这是2013年1.3版本的设计，到2014、2015年3.1版本的设计，逐步引入弹性子计算系统。

云网络管理平台最开始1.0的设计只是对资源的管理，到了2.0、3.0的设计时，我们逐步发现上云的过程中，应用会对云网络管理平台提出新的需求，如何调整、如何增加新的技术实现对应用的支撑。接下来看一下容器跟微服务引入之后对云网络管理平台本身设计架构的改变。

**容器与微服务化对云网络管理平台新的架构设计的支撑**



两张图的对比。左边是1.0的架构，它的部署架构包括自服务门户、门户后台、用户服务、运营服务、运维管理。运营服务里面有订单管理、计费管理、AZ配置。所有业务都是耦合的，如果要升级一个功能整个服务都要升级。这是一个不利点，而且开发测试团队也不明晰，升级任何功能点都要对模块进行重构。所以我们在微服务上考虑引入这样一个架构，运营服务、产品支撑和运维管理里面每一个功能点都单独做成一个微服务。



微服务的系统架构主要是两个分离： 第一是客户交互与业务逻辑分离。前台的JS与后台java分离。 第二就是微服务化，把这些服务都通过服务管理、服务注册、服务发现管理起来，前台对服务的使用都通过微服务管理平台发现相应的服务，导流到各个服务节点上。云网络管理平台最终提供很多产品，像云主机产品、块存储产品等等，我们会把这些产品做成一个一个的服务，通过微服务化可以实现数据中心里每个服务的自治，不同服务的升级可以完全基于自己去做，只需要保证对外API是统一的。

再一个就是分布式，我们做了无状态化可以做更好的云网络管理平台本身的弹性。比如一个大的运营商是集团级的，用户量有上千万，每次促销时云网络管理平台自身的压力也是一个比较大的点。每次促销都是先备很多虚拟机上去，现在微服务化之后通过无状态化可以实施自动的扩展。

微服务化之后每一个管理平台最终的状态是什么样的？第一是运营管理门户，每个门户有订单查询、产品目录、工单等等呈现。在门户选择一个服务就会通过REST API的形式到运营管理API上，每一个API都是一个单独的服务。点了一个服务之后，这个服务的逻辑实现是在后台，通过微服务管理平台上找相应的已经注册的服务，然后服务将最终的逻辑实现。

资源管理平台也是同样的概念，比如说资产管理、故障告警等功能最终会落在一个个服务的API上，再通过服务发现、服务管理功能导流在后台服务做统一处理。

微服务化之后有一个比较重要的就是对服务能力的监控。目前我们是两种方式，一种方式是用ceilometer实时采集呈现，还有一种是实时呈现完之后历史数据的存储，目前是用Spark集群做。需要对每个Hypervisor做实时呈现跟监控，这是做平台必不可少的模块。

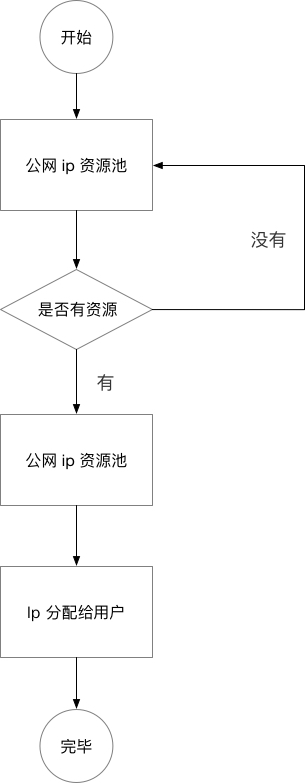
再就是日志，比如说调用的日志，从用户申请一个虚拟机的服务到每个服务之间的调用是7步左右，这7步任何一步出现了问题都要做产品回退，因为这会影响最终使用，所以需要对每一步的调用日志进行监控和采集，以确保可以排查哪一步出了问题。日志我们采用了开源的Elastic Search，做了一些封装和改造。最终在日志管理的界面上呈现。

最后就是微服务化之后系统模块之间的关系图。以前就Web、App、DB三个服务，微服务化之后变成30几个服务，每个服务之间也有调用关系，这种关系图是维护系统运行的最重要的模块。CMP1.0的时候我们很少强调总架构师的功能，微服务化引入之后我们在后台研发上专门设立总架构师的功能，他来维护模块之间的关系定义及关系服务之间的架构设计。

**三、软件功能描述**

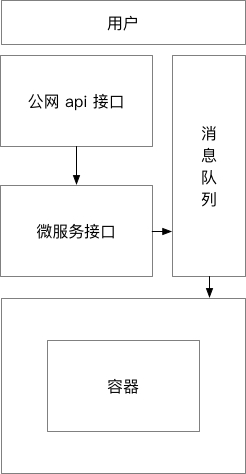
**3.1 公网创建**

公网创建的时候，是从公网 ip 资源池中选取一个未使用的，分配给用户使用。公网 ip 是从网络服务商申请的 ip 或者 IP地址段，用户购买之后，可以按照带宽或者流量进行收费。当用户删除之后，ip 地址回归资源池，其他用户可以继续使用。流程如下图所示：

****

**3.2公网绑定功能**

公网绑定是在容器创建完成之后，为容器外网网卡配置指定的 ip 地址。

****

# 四、接口设计

## 4.1接口列表

**内网相关接口**

POST /ip/intranet/getGatewayIpPara 查询网关下的所有网段

POST /ip/intranet/getIntrIpByPara 查询网段下所有ip

POST /ip/intranet/getContainerByIntranetIp 查询内网ip绑定的主机

POST /ip/intranet/getIntranetIpById 根据id查询内网ip

POST /ip/intranet/getIntrIpParaByUserId 查询用户的所有网段

POST /ip/intranet/getIpParaList 查询内网ip段列表（带分页）

POST /ip/intranet/deleteIpPara 批量删除ip段

POST /ip/intranet/bindIpPara 网关绑定内网ip段

POST /ip/intranet/getUnusePara 查询未使用的ip段

POST /ip/intranet/getParaByContainer 获取网关的ip段

POST /ip/intranet/unBindPara 解绑网关ip段

POST /ip/intranet/unBindParaActive 解绑ip段回调

POST /ip/intranet/createContainerNetworkCallback 绑定webconsole网卡回调

POST /ip/intranet/createWebconsoleNetwork 给webconsole添加网卡,确保webconsole和每个内网端都通

POST /ip/intranet/getCountIp 获取网段内ip

POST /ip/intranet/recoverWebconsole 恢复webconsole容器数据

POST /ip/intranet/createAllWebConsoleNetwork ip全部下发一遍

POST /ip/intranet/addIpData 修复ip数据

POST /ip/intranet/addIpDataFromMQ 修复ip数据

POST /ip/intranet/deleteIntranetIpPara 删除内网ip段

POST /ip/intranet/getIntranetIpList 查询内网ip列表（带分页）

POST /ip/intranet/getIntrIpParaList 查询内网ip段列表（带分页）

POST /ip/intranet/bindIntranetIp 给容器绑定内网ip

POST /ip/intranet/bindIntranetIpActive 给容器绑定内网ip回调

POST /ip/intranet/bindIpParaActive 网关绑定内网ip段回调

POST /ip/intranet/getContainerBandIntranetIp 查询容器绑定的内网ip

POST /ip/intranet/removeBindContainerIntranetId 解除绑定内ip

POST /ip/intranet/removeBindContainerIntranetIdActive 解除绑定内ip回调

POST /ip/intranet/getUserGateWay 查询用户下的网关

POST /ip/intranet/distIpPara 用户初始化分配内网ip网段

POST /ip/intranet/createIntranetIp 通过网段创建ip

POST /ip/intranet/getIntranetIp 获取用户可用的内网ip

**公网 ip相关接口**

# POST /ip/public/createPublicIpInfo 创建公网ip

# POST /ip/public/getPublicIp 从公网ip表拿一个未使用的ip

# POST /ip/public/getPublicIpList 公网IP信息列表分页

# POST /ip/public/selectPublicIpInfoList 查询公网ip信息不分页

# POST /ip/public/deletePublicIpInfo 删除公网ip

# POST /ip/public/updatePublicIpInfo 修改公网ip

# POST /ip/public/updatePublicIpInfoActive 修改公网ip回调

# POST /ip/public/bindContainer 绑定公网ip

# POST /ip/public/bindContainerActive 绑定公网ip

# POST /ip/public/unBindContainer 解除绑定公网ip

# POST /ip/public/unBindContainerTimeOut 解除绑定公网ip,过期解绑,解绑之后回收公网ip

# POST /ip/public/getPublicInBandContainer 查询公网ip下绑定的主机

# POST /ip/public/updateIcp 设置icp

# POST /ip/public/getPublicIpInfo 根据主键id查询公网ip

# POST /ip/public/getContainerPublicIp 查询容器下所有公网ip

# POST /ip/public/deletePublicIpInfos 批量删除公网ip

# POST /ip/public/getCostPurchase 计算购买ip宽带费用

# POST /ip/public/getExpansionCostPurchase 计算扩容费用

# GET /ip/public/findOutTimePublicIp 查询过期公网ip

# POST /ip/public/removeBindContainerActive 解除绑定公网ip回调

DNS 服务接口

# POST /dns/saveUserDns 修改用户默认dns

# POST /dns/addOrUpdateDefaultDNS 添加默认dns

# POST /dns/getDefaultDNS 查找默认dns

# POST /dns/newDnsGroup 新建dns组

# POST /dns/getUserDnsGroup 获取用户dns组

# POST /dns/deleteDnsGroup 删除dns组

# POST /dns/getContainerDns 查询容器的dns

# POST /dns/modifyDnsToChannelCallback 下发dns回调

# POST /dns/getUserDns 查询用户默认dns

# POST /dns/modifyUserDns 修改dns

# 4.2出错处理设计

## 出错输出信息

接口返回结果数据格式为JSON，返回的错误信息在最外层根据HTTP统一错误码进行判断对应结果如下：

200 - 服务器成功返回网页

404 - 请求的网页不存在

503 - 服务不可用

1xx（临时响应）表示临时响应并需要请求者继续执行操作的状态代码。

100 （继续） 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。

101 （切换协议） 请求者已要求服务器切换协议，服务器已确认并准备切换。

2xx （成功）

表示成功处理了请求的状态代码。

200 （成功） 服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。

201 （已创建） 请求成功并且服务器创建了新的资源。

202 （已接受） 服务器已接受请求，但尚未处理。

203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 （无内容） 服务器成功处理了请求，但没有返回任何内容。

205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206 （部分内容） 服务器成功处理了部分 GET 请求。

3xx （重定向）

表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。

300 （多种选择） 针对请求，服务器可执行多种操作。服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。

301 （永久移动） 请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

303 （查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。

304 （未修改） 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。

305 （使用代理） 请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理。

307 （临时重定向） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

4xx（请求错误）

这些状态代码表示请求可能出错，妨碍了服务器的处理。

400 （错误请求） 服务器不理解请求的语法。

401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 （禁止） 服务器拒绝请求。

404 （未找到） 服务器找不到请求的网页。

405 （方法禁用） 禁用请求中指定的方法。

406 （不接受） 无法使用请求的内容特性响应请求的网页。

407 （需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。

408 （请求超时） 服务器等候请求时发生超时。

409 （冲突） 服务器在完成请求时发生冲突。服务器必须在响应中包含有关冲突的信息。

410 （已删除） 如果请求的资源已永久删除，服务器就会返回此响应。

411 （需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。

412 （未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。

413 （请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。

414 （请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。

415 （不支持的媒体类型） 请求的格式不受请求页面的支持。

416 （请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。

417 （未满足期望值） 服务器未满足"期望"请求标头字段的要求。

5xx（服务器错误）

这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

501 （尚未实施） 服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。

502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。通常，这只是暂时状态。

504 （网关超时） 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505 （HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

如果最外层返回码为200，其他业务错误类型在第二次中进行标识。