一、云计算

> 本文包括:
>
> 1、云计算的概念
>
> 2、云计算的价值
>
> 3、虚拟化

1、云计算的概念

1. 云计算驱动：
   - 技术驱动
   - 需求驱动
   - 商业模式驱动
2. 从商业视角看：云计算 = 信息电厂
   - 企业数据中心通过计算和存储从局域网向互联网迁移，
   - 软件从终端向云端迁移到达互联网数据中心，
   - 互联网数据中心有软件和硬件解耦实现硬件共享到达 PC 端或移动端；
3. 从技术视角看：云计算 = 计算、存储的网络
   - 分为广义和狭义
     - 狭义：IT 基础设施的交互和使用模式，通过按需易扩展的方式获取所需的资源（包括基础架构，平台，软件）；
     - 广义：通过网络按需易扩展的方式获得服务（IT，软件，互联网相关或任意其他服务，例如政务云，教务云，医疗云）
4. 部署模式：
   - 私有云（private cloud）：
   - 公有云（public cloud）：某一组织拥有，向公众
   - 混合云（hybrid cloud）：可以同时包含私有云和共有云

   > 私有云和公有云通过企业防火墙隔离

5. 应用模式：
   - Laas（基础设施及服务）：例如虚拟机的租用，云盘；
   - Paas（系统及平台）：主要面向专业开发者，可以加快Saas
   - Saas(软件及服务)：面向终端用户，用户无需购买，只要向软件提供商租用即可

2、云计算的价值

1. 行业客户对云的认知
   - 云计算必须演进至开放平台
   - IT 交付将基于云来进行
   - 找到合适的战略合作伙伴以启动云战略至关重要
   - 关心云平台的安全性
2. 行业客户反应的困难与挑战
   - 复杂性与平滑过度
   - 可靠性或SLA（服务等级协议）：包括服务类型、服务种类灯
   - 工作负载及可迁移性：业务负荷分担，业务能否热迁移
   - 数据的安全性
   - 厂商锁定
   - 看不到投资回报
3. 云计算的价值：
   1. 智能资源的调度
      - 基于策略地实现负载均衡
      - 通过热迁移来实现节能减排

2. 提高资源的利用率
    - 通过资源共享
    - 通过分时共享：不同业务有不同的峰值/谷底
3. 分布式的计算和存储
4. 统一管理
5. 业务快速部署

4. 云计算带来的机会

    - 对于云计算供应商
    - 对于最终的消费者
        - 随时随地
        - 广泛
    - 对于中小型客户
        - 更高级别产业链
        - 发展
    - 对于企业级的客户
        - 降低设备复杂性
        - 提高敏捷
    - 对于政府机构
        - 提供更广泛地政务

3、虚拟化

- 虚拟化技术的发展由来：

    - 60 年代在大型机有所应用
    - 90 年代在小型机上出现逻辑区分的应用
    - 2000 年，X86 平台虚拟技术开始出现
    - 2001 年，X86 平台虚拟化技术在服务器上开始应用
- 虚拟化技术的推动力：

    - CPU 的处理速度越来越快，超出软件对硬件性能的要求
    - INTER 和 AMD 在 CPU 里加入了虚拟指令
    - 企业成本压力
    - 环保压力
    - 不断增长的业务压力

    > 一个服务器只运行一个应用程序，但消耗资源太大

- 虚拟化的概念
    - 虚拟化是指通过虚拟化技术，将一台计算机虚拟为多台逻辑计算机，在一台计算机上同时运行多个逻辑计算机，每个逻辑计算机可以运行不同的操作系统，并且，应用程序都可以在相互独立的空间内运行而互不影响，从而显著的提高了计算机的工作效率。
    - 物理机：宿主机，运行的OS称为 Host OS
    - 虚拟机：客户机，运行的OS称为 Guest OS
- 虚拟化后的区别就是：
    - 物理机上可以同时运行多个虚拟机，而且还有虚拟机监控器，通过虚拟机监控器的模拟，可以使得虚拟机在上层软件看来就是一个真实的机器。
    - 虚拟机是将物理资源池化了，多个虚拟机直接从资源池种获取资源，与硬件解耦和。
- 虚拟化的主要内容

    - 计算虚拟化：CPU 虚拟化 、内存虚拟化 、 IO 虚拟化
    - 存储虚拟化：裸设备 + 逻辑卷 、存储设备虚拟化 、 主机存储虚拟化 + 文件系统
    - 网络虚拟化: VPN 、 VLAN
- 虚拟化的本质

    - 分区
    - 隔离
    - 封装
    - 独立
- 虚拟化技术的应用案例：--- 热迁移

test

```
软件即服务（SaaS）的云计算，更有利于服务质量的改善，相比于其他的服务模式，可以更快速地从故障中恢复业务的正常运行。B

   A  对
   B  错

IaaS 是下列哪一种服务模式的简称？B

   A  软件即服务
   B  基础设施即服务
   C  平台即服务
   D  硬件即服务

 下列哪一种服务模式的云计算，可以为未来的业务扩展、功能扩展等提供最大的可扩展余地？A

   A  DaaS
   B  SaaS
   C  IaaS
   D  PaaS

 下列哪一种技术是云计算基础架构的基石？A

   A  虚拟化
   B  分布式
   C  并行计算
   D  集中式

下列选项中，亚马逊 AWS 提供了哪些类型的云服务？ABC

   A  SaaS
   B  IaaS
   C  PaaS
   D  DaaS
```

二、OpenStack

> 本文包括：
>
> 1、OpenStack 起源
>
> 2、OpenStack 架构
>
> 3、Nova 介绍
>
> 4、Swift 介绍
>
> 5、Keystone 介绍
>
> 6、Neutron 介绍
>
> 7、Glance 介绍
>
> 8、Cinder 介绍
>
> 9、Ceilometer 介绍
>
> 10、Heat 介绍

- OpenStack 不是云
- OpenStack 不是虚拟化
- OpenStack 是目前最主流的开源云操作系统内核

- 三大组件
  - 计算
  - 网络
  - 存储
- 主要的用户接口
  - Dashboard
  - Horizon

1、OpenStack 起源

- 起源：2010.7
- rack space （美国公司）开源了存储代码 Swift
- NASA 贡献了计算代码 Nova
- OpenStack 不是一个项目，而是一些项目的统称
- OpenStack 项目发展状况：
  - **提供对象存储服务的 Swift**
  - **提供计算服务的 Nova**
  - 提供镜像服务的 Glance
  - 提供网络服务的 Neutron
  - 提供身份认证服务的 Keystone
  - 提供计量服务的 Ceiliometer
  - 提供块存储服务 Cinder
  - 提供编排服务的 Heat
- OpenStack 的会员构成：
  - 白金会员【8 个】
  - 黄金会员【华为。。。】
  - 其他的企业赞助商
- OpenStack 的版本：
  - 按照 26 个字母顺序来进行区分的
- 目前是 Ocata 2017.2.22
  - 视频录制时说的是：Newton 2016.10.06(视频中)

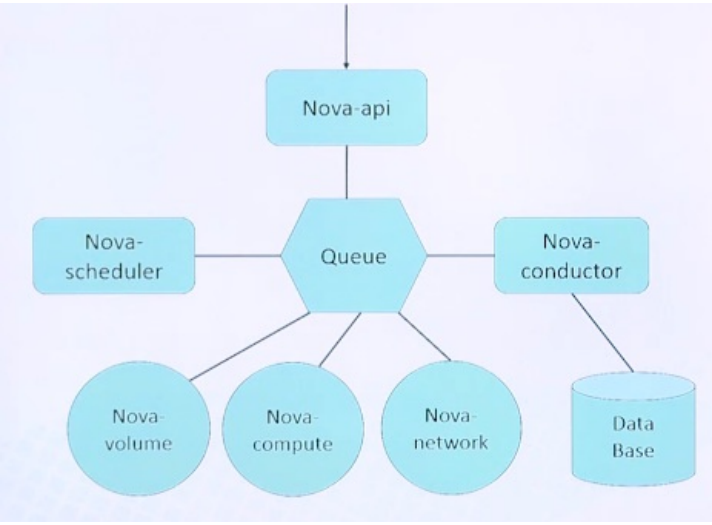
2、OpenStack 架构

1. 提供身份认证服务的 Keystone 组件
   - 主要负责身份服务
   - 管理用户、租户、角色、服务和服务断点
   - 可以支持 SQL，PAM，LDAP 作为后端
2. 提供计算服务的 Nova 组件
   - 主要负责虚拟机实例的调度分配以及实例的创建、起停、迁移、重启等操作，从而来管理云中实例的生命周期。
   - 是整个云中的组织控制器
   - 计算服务：
     - 计算节点运行虚拟机的
   - 分布式控制器
     - 负责处理器调度策略及 API 调用等。
3. 提供镜像服务的 Glance 组件
   - 能够实现镜像的创建、镜像快照管理、以及镜像模板等等。
   - 同时也支持各种格式的镜像格式，如：raw，qcow，vhd，vmdk，iso
   - 镜像文件一般存储在后端，如：Swift，Filesystem，AmazonS3
4. 提供对象存储服务的 Swift
   - 主要提供了存取数据的应用服务。
   - 通常用于保存非结构化的数据，比如通常作为 Glance 组件的存储后端或者作为一些云盘等应用

5. 提供网络服务的 Neutron
    - 基于软件定义网络的思想，实现网路资源的软件化管理。支持各种各样类型的插件，实现多租户网络的隔离。
    - 也可以对硬件以及软件的解决方案进行集成。
6. 提供块存储服务 Cinder
    - 为虚拟机实例提供卷的持久化服务（块存储）
    - 同时支持一些对卷的快照、备份等的一些管理
    - 基于插件的架构，非常易于扩展。
7. 提供 WEB 统一化管理界面服务的 HORIZON 组件
    - 主要提供了自动化仪表板的管理服务
    - 实现对用户、租户、卷、网络等几乎多有资源的图形化管理。
8. 提供监控和计量服务的 Ceilometer
    - 提供对 Opens tack 中平台组件的监控
    - 提供计量服务

3、Nova 介绍

- 什么是 Nova?
    - Nova 是 Open stack 云中的计算组织控制器，管理 opens tack 云中实例的生命周期的所以活动，使得 Nova 称为一个负责管理计算资源、网络、认证所需的可扩展性平台。
- 常用术语
    - KVM：内核虚拟化，Open Stack 默认的 Hypervisor层
    - Qemu: KVM 的替补角色，没有 KVM 的执行效率高，且不支持全虚拟化
    - Flavor：新建虚拟机的配置列表，虚拟机模板
    - Keypair：ssh 连接访问实例的密钥对
    - 安全组：是一个控制访问策略的容器，里面包含了各种各样的安全组规则
    - 安全组规则：用来控制实例访问的具体策略
- Nova 组件
    - Nova API：提供了统一风格的 RestAPI 接口，作为 Nova 组件的入口，接受用户的请求
    - Nova scheduler ：负责调度，将实例分配到具体的计算节点
    - Nova conductor：主要负责与 Nova 数据库进行交互
    - Nova compute：运行在计算节点商，用于虚拟机实例的创建和管理 以及提供消息传递的
    - 消息队列：Nova 各个组件之间的消息传递 和数据库模块：
- **【Nova 各个组件如何协作运行？】**
    1. 首先，用户通过 CLI 命令行或者 horizon 向 Nova 组件提出创建实例的请求时，Nova API 作为 Nova 的入口，将会接受用户的请求，将会以消息队列的方式，将请求发送给 Nova scheduler
    2. Nova scheduler 从消息队列中，侦听到 Nova API 的消息队列后，去数据库中查询，当前计算节点的负载和使用情况
    3. 由于 Nova scheduler 不能直接跟数据库进行交互，因此，将会借助于消息队列的方式通过 Nova conductor 组件，进而与数据库进行交互
    4. 然后将查询到的结果，将虚拟机实例分配到当前负载最小，并且满足启动虚拟机实例的那个计算节点上
    5. 但最终的虚拟机实例的组件，还是要靠 Nova compute 来完成
    6. 实例的创建，离不开镜像、网络等一些资源的配合，因此，Nova compute 将会与 Nova volume、Nova network 等等一些组件通过消息队列的方式实现相互的协作。最终完成虚拟机实例的创建。

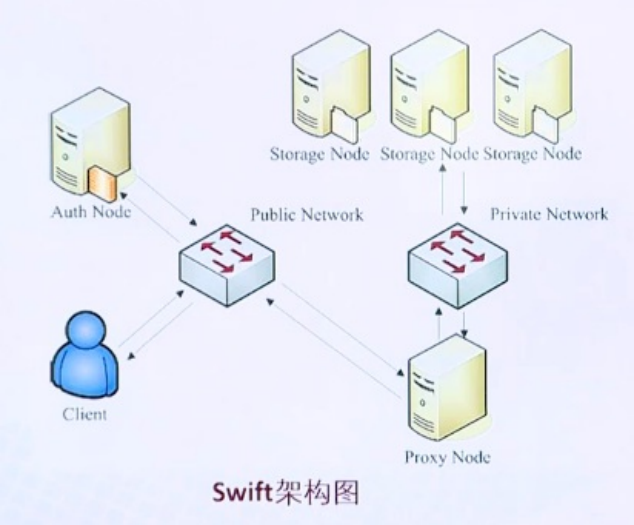


- Nova 的功能特性：
  - 实现实例的生命周期的管理
  - 调动管理平台的网络、存储等资源
  - 提供了统一风格的 RestAPI 接口
  - 支持 KVM、VMware 等透明的 hypervisor
  - 各个模块之间通过消息队列来进行消息传递

4、Swift 介绍

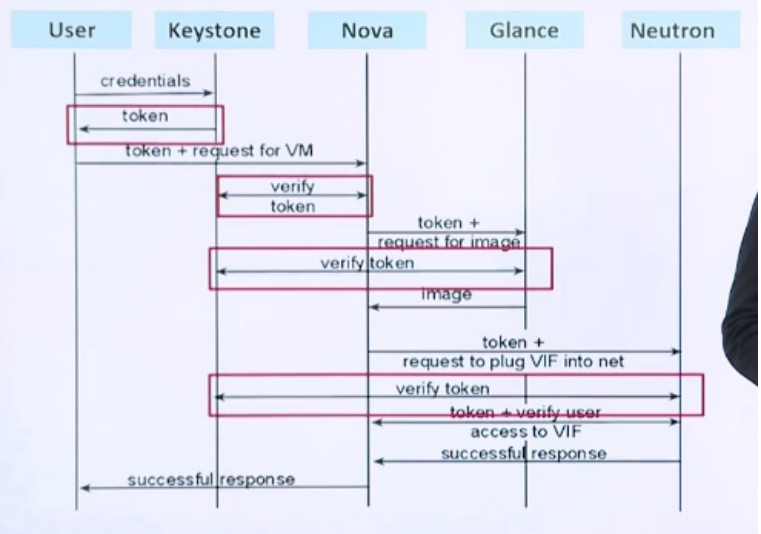
1. 什么是 Swift？
   - Swift 是 提供高可用分布式对象存储的服务，为 nova 组件提供虚拟机镜像存储服务，在数据冗余方面，无需采用 RAID 通过在软件层面，引入一致性散列技术和数据冗余，牺牲一定得数据一致性，来达到高可用和可伸缩性。
   - 支持多租户模式下，容器和对象读写操作，适用于互联网应用场景下非结构化的数据存储，比如，华为云盘等。
2. Swift 中的常用术语【1】
   - Account：用户定义的管理存储区域
   - Container：存储隔间，类似于文件夹或者目录
   - Object：包含了基本的存储尸体和它自身的元数据
   - Ring：环，记录了磁盘上存储的实体名称和物理位置的映射关系。
3. 以上术语之间的关系：
   - 首先，可以创建多个 account
   - 每个 account 里可以创建多个容器 container
   - 每个 container 下可以创建多个 object，【container 之间不能相互嵌套】
4. Swift 的功能
   - Swift 在物理结构上往往会存储对象的多个副本，通常按照物理位置的特点，将对象拷贝到不同的物理位置的特点，将对象拷贝到不同的物理位置上，来保证数据的可靠性。
5. 常用术语 【2】
   - Region: 地域，从地理位置上划分的一个概念（基于灾备）
   - Zone：可用区，按照独立的供网、供电既基础设备划分
   - Node：节点，代表了一台存储服务器
   - Disk：磁盘，代表着物理服务器上的存储设备
   - Cluster：群集，为冗余考虑而设计的架构
6. 以上术语之间的关系：
   - 可以根据不同的物理位置，有不同的 Region，不同的 region 代表两个不同的城市0然后在同一个 region 下，为冗余的考虑，设置了多个可用区，zone
   - 每一个 zone 可以有不同的存储节点 node
   - 在更大的架构上，两个 region 可以构成一个 cluster。
7. Swift 的架构
   - 首先，用户提出一个对象存储服务的申请，由 Swift 的ＡＰＩ接受和处理，收到之后，先去找 keystone 认证节点，对用户的身份进行认证
   - 认证通过后，将请求提交给，名称为 Swift Proxy 的组件，Swift Proxy 是 Swift 的代理，由 Swift Proxy 来确定究竟应该将存储对象放在哪一个满足存储要求的存储节点上

- 最终将对象存储到指定的存储节点上即可。最终将返回结果返回给用户。



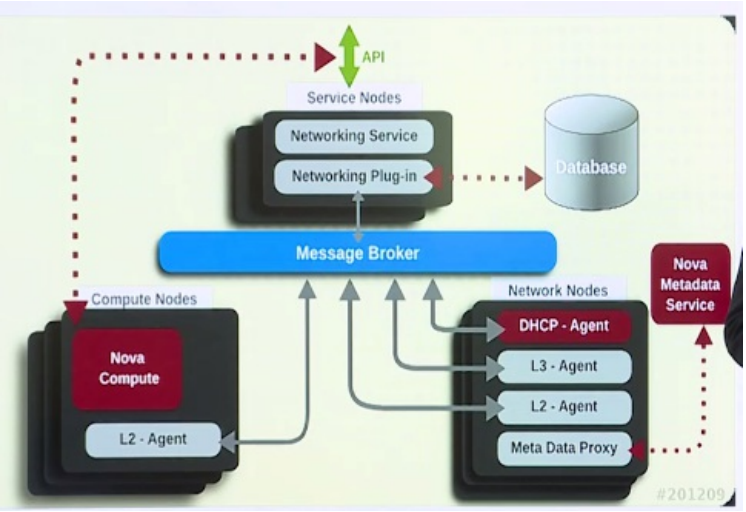
5、Keystone 介绍

- Keystone 介绍
  - 提供身份验证、服务规则和服务令牌功能
  - 任何服务之间相互调用，都需要经过 keystone 的身份验证
- 常用术语：
  - User：Openstack 最基本的用户
  - Project：指分配给使用者的资源的集合
  - Role：代表一组用户可以访问组员的权限
  - Domain：定义管理边界，可以包含多个 project/tenant、user、role 等
  - Endpoint：服务的 URL 路径，暴露出来的访问点
- Keystone 认证模型
  - 首先创建两个域 domain 1 和 2，给两个公司使用
  - 每个公司在 domain 下创建三个子公司 project 1 2 3
  - project 下可以创建多个用户 User，用户 User 可以跨多个 project 存在
- Keystone 认证原理
  - 当用户在创建时，将通过 Keystone 将会创建一个访问令牌 accesstoken
  - 假设当用户提出创建虚拟机实例的请求时，首先将自己的访问令牌和访问请求提交给 Nova 服务
  - Nova 服务为确保用户的访问令牌并没有篡改过，因此首先会将访问令牌交给 keystone 进行验证
  - 验证通过后 Nova 为了启动虚拟机的实例，还需要向 Glance 组件申请相关的镜像资源
  - Glance 为保证访问令牌在传递的过程中没有被篡改过，也需要将访问令牌发送给 keystone 做确认
  - 验证通过后将会发放镜像资源给 nova 组件
  - 当然，虚拟机实例的创建还需要存储、网络等资源，因此 nova 组件还需要给负责各种资源的模块传递申请资源的请求，资源申请的过程中都会伴随这访问令牌的验证
  - 最终，nova 拿到启动虚拟机实例的所有资源后进行实例的启动，然后分配给相关的用户
  - 整个过程来看组件之间资源的调用都离不开 keystone 的验证....



6、Neutron 介绍

- Neutron 的简介：
  - Neutron 是 open stack 中负责提供网络服务的组件，基于软件定义网络的思想，实现软件化的网络资源管理
  - 在实现上，充分利用了 linux 系统中各种与网络相关的技术，支持第三方插件
- Neutron 中常用的术语
  - Bridge-int：综合网桥，常用于实现内部网路通讯功能的网桥
  - Br-ex: 外部网桥，通常用于跟外部网络通讯的网桥。
  - Neutron-server：提供了 API 接口，将配置好的 API 接口，提供给相关的插件，进行后续处理
  - Neutron-L2-agent: 二层代理，用于实现二层网络通讯的代理，用于管理 VLAN 的插件，接受 Neutron-server 的指令来创建 VLAN。
  - Neutron-DHCP－agent: 为子网自动分发 IP 地址
  - Neutron-L3-agent: 负责租户网络和 floating IP 之间的地址转换，通过 linux iptables 中的 NAT 功能来实现 IP 转换
  - Neutron-metadata-agent: 运行在网络节点上，用来响应 nova 组件中的 metadata 请求
  - LBaaS agent: 为堕胎实例和 open vswitch agent 提供负载均衡服务
- Neutron 的架构
  - 当 Neutron 通过 API 接口，接受来自用户或者其他组件的网络请求时，以消息队列的方式提交给 2、3 层代理
  - 其中 Neutron-DHCP－agent 实现子网的创建和 IP 地址的自动分发
  - 而 Neutron-L2-agent 实现相同 VLAN 下，网络的通信
  - Neutron-L3-agent 实现同一个租户网络下，不同子网间的通信



7、Glance 介绍

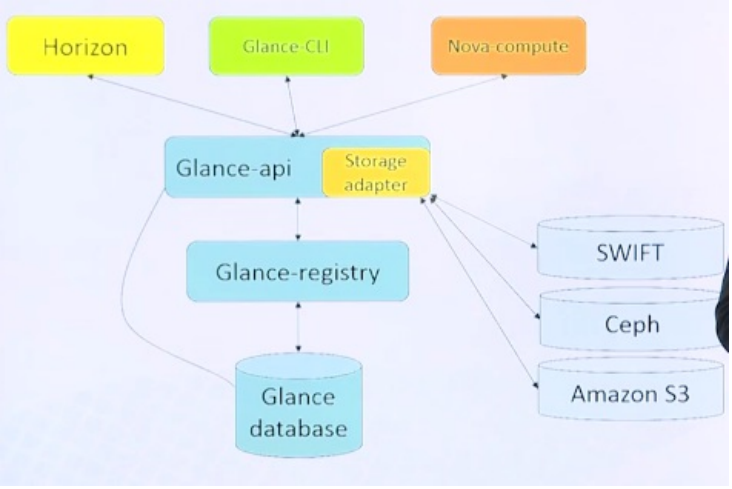
1. Glance 的作用
   - 为 nova 提供镜像服务以便启动实例的组件

- 但不负责镜像的本地存储，
- 可以对镜像做快照、备份、镜像模板等管理

2. Glance 镜像支持的格式：
   - Raw
   - 经常被 vmware、visualbox 使用的 vhd
   - Vdi
   - 光盘 iso
   - openstack 经常使用的 qcow2
   - 亚马逊的 aki 和 ami
3. Glance 组件：
   - Glance-api 负责提供镜像服务的 rest api 服务, 作为镜像服务请求的入口。
   - Glance-registry 负责与 glance 使用的数据库交互，比如镜像的创建、删除、修改等操作
4. Glance 的架构：

   1. 当有来自 horizon、CLI、Nova、Compute 发送过来的镜像请求，由 glance api 接收处理，将请求的消息传递给 Glance-registry 组件
   2. 然后到数据库中查询镜像存储的位置信息，将查询到的结果返回给 api
   3. glance api 接下来将会调用 Storage adapter 组件进行查询，用来查询 glance 后端的存储，比如 SWIFT、Ceph、Amazon S3 等，最终获取镜像返回给用户。

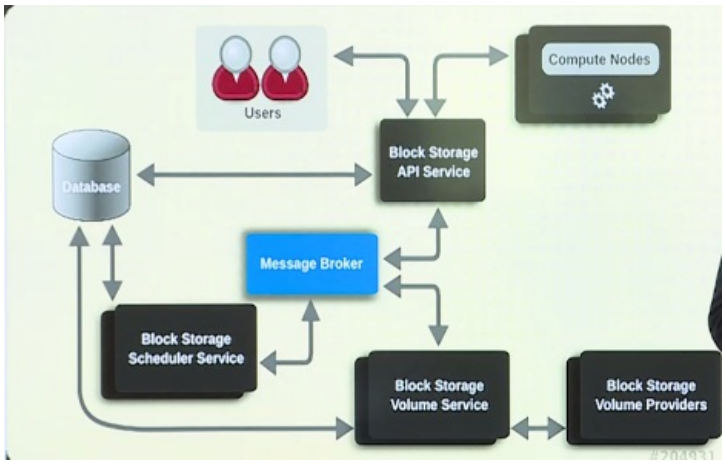
   > glance api 也可以直接于数据库交互，但只传输少量数据



8、Cinder 介绍

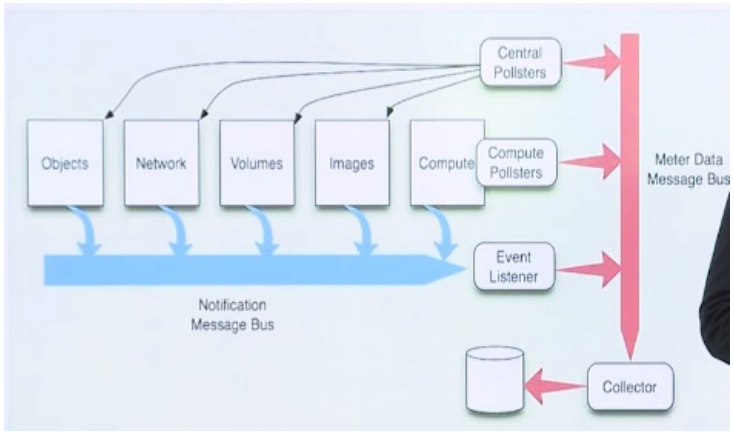
- Cinder 的简介
  - 为虚拟机实例提供 volume 卷的块存储服务，可将卷挂载到实例上，作为虚拟机实例的本地磁盘来使用
  - 一个 volume 卷可以同时挂载到多个实例上
  - 共享的卷同时只能被一个实例进行写操作，其他只能进行只读操作
- 支持的文件系统类型
  - LVM / ISCSI
  - NFS
  - NetAPP NFS
  - Gluster
  - DELL Equall Logic
- 常用术语
  - Volume 备份：volume 卷的备份，存放在备份的设备中
  - Volume 快照：卷在某个时间点的状态
  - Cinder API：为 Cinder 请求提供统一风格的 Rest API 服务，是 Cinder 服务的入口
  - Cinder Scheduler：负责为新建卷制定块存储设备
  - Cinder Volume：负责与存储的快设备交互，实现卷的创建、删除、修改等操作
  - Cinder Backup：备份服务负责通过驱动和后端的备份设备打交道。
- Cinder 架构

- 当有用户或者 nova compute 提出创建卷的服务的请求时，首先由 Cinder API 接收请求，然后以消息队列的方式发送给 Cinder Scheduler 进行调用
- Cinder Scheduler 侦听到来自 Cinder API 的消息队列后，到数据库中去查询当前存储节点的状态信息。并根据预定策略，选择卷的最佳 volume service 节点，然后将调度的结果发布出来，给 volume service 来调用
- volume service 收到来自 volume schedule 的调度结果后会去查找 volume Provider。
- volume Provider 在特定的存储节点上创建相关的卷，然后将相关的结果返回给用户，同时将修改的数据写入到数据库中。



9、Ceilometer 介绍

- Ceilometer 的简介：
  - Ceilometer 是 open stack 中的一个子项目，为计费、监控等其他的服务提供数据支撑。
- Ceilometer 存在的理由：
  - IaaS
  - 更多的公司利用 open stack 做自己的公有云平台。而作为共有云，计量和监控，这两个基础的服务往往是必不可少的，计量是为了获取平台中用户对自己的使用情况，监控是为了确保资源处于一个健康的状态
  - 因此，Ceilometer 在项目提出之初，是为了计量、计费而生。
- Ceilometer 的核心概念
  - Ceilometer-agent-compute: 运行在计算节点上，是收集计算节点上信息的代理
  - Ceilometer-agent-central：运行在控制节点上，轮询服务的非持续化数据
  - Ceilometer-collector: 运行在一个或者多个控制节点上，监听 Message Bus【消息总线】，将收到的信息写入到数据库中
  - Storage: 数据存储，支持 mongo DB，mysql 等等。用于存储收集到的样本数据
  - API server: 运行在控制节点上，提供对数据库的数据的访问
  - Message Bus: 计量数据的消息总线，收集数据给 Ceilometer-collector
- Ceilometer 架构
  - Ceilometer 采用了两种数据采集的方式，其中一种是消费了 openstack 内各个服务自动发出的 notification 消息，【图中的蓝色箭头】，另外一种是调用各个服务的 API，去主动轮询获取数据。【图中的黑色箭头】

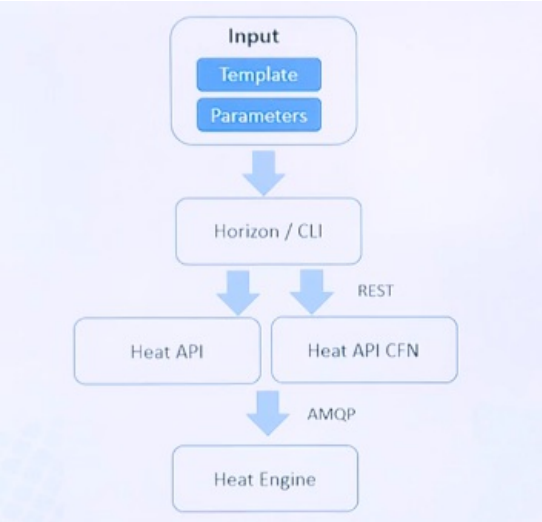


- 为什么采用两种数据采集的方式？【也是工作架构】
  - 第一种:
    - 因为在 openstack 中，大部分事件都会发出 notification 消息
    - 比如创建删除 instance 实例的时候，这些计量计费的信息时，都会发出 notification 消息

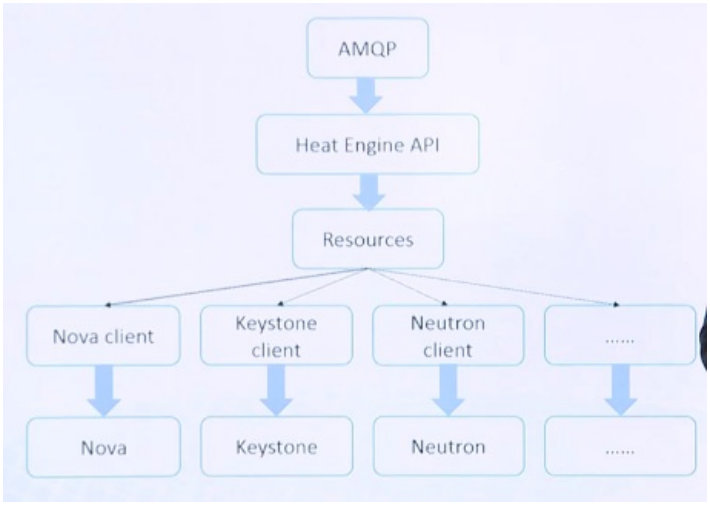
        - 而作为 Ceilometer 组件，就是 notification 消息的最大的消费者
        - 因此，第一种方式，是 Ceilometer 的首要的数据来源。
    - 第二种:
        - 但是，也有一些计量的消息，是 notification 获取不到的，比如一些 instance 的 CPU 的运行时间，或者是 CPU 的使用率等等
        - 因此，Ceilometer 增加了第二种方式，即为周期性的调用相关的 API，去轮询这些消息。

10、Heat 介绍

- Heat 简介
    - Heat 是一个基于模板来创建相关资源的服务，是 OpenStack 核心项目之一。可以通过. yaml 文件生成模板，通过 Heat-agent 组件在 OpenStack 中创建相关的资源。
    - 除此之外，支持自由的 Haut 模板。以及亚马逊的 cloudy formation 格式的模板。
    - 模板支持丰富的资源类型，不仅支持了常用的基础架构，包括计算、网络、存储、 镜像等功能，还包括了 Ceilometer 警报等高级资源。
    - 提供基于模板的编排服务。
- 常用术语
    - Stack：Heat 要用到的所有设施和资源的集合。
    - Heat template：是以. yaml 结尾的文件，用于创建 stack。
    - Heat-api：提供 rest api 服务，Heat 入口将 api 请求发送给 heat engine 去执行。
    - Heat-api-cfn：支持亚马逊格式访问的 rest api。
    - Heat-engine：Heat 的核心模块，接收 API 请求，在 openstack 中创建资源。
    - Heat-cfntools、Heat-init：在镜像中安装完成虚拟实例操作任务的工具。
    - Heat-api-cloudwatch： 监控编排服务。
    - Resource：底层各种服务抽象的集合。（例：计算、存储、网络）
    - Heat-client：用于调用访问其它各个组件的 client 工具
- Heat 架构



1. 当用户在 Horizon 中或者命令行（CLI）中提交包含模板和参数的，创建实例的请求时，Heat 服务接收请求，调用 Heat-API/Heat-API-cfn
2. 然后 API/API-cfn 首先验证模板的正确性，然后通过消息队列异步传输给 Heat Engine 进行处理。



1. 当 Heat Engine 拿到请求后，会把请求解析为各种资源类型，而每一种资源都有相关的 Client 与相关的服务对应。
2. Client 通过发送 rest 请求给其它的服务，从而获取相关的资源，最终完成请求的处理。

- 在整个过程中，Heat Engine 的作用分为三层：

1. 处理 Heat 层面的请求，然后根据模板和输入的参数来创建 Stack。
2. 解析 Stack 中各种资源的依赖关系以及 Stack 的嵌套关系。
3. 根据解析出来的关系，依次调用各种服务的 Clinet，来创建各种资源。

test

```
下列关于 Nova 组件各个功能模块的描述哪一项是错误的？A

   A  nova-scheduler 直接访问数据库从而决定将实例分配到具体的计算节点上
   B  nova-conductor 负责提供 nova 数据库连接的服务
   C  nova-compute 是运算在计算节点上的服务，主要用于创建和管理虚拟机实例
   D  nova-api 模块能提供 rest 风格的 API 服务接口

Openstack 组件中下列哪一项可以以云盘的存储方式对外提供服务？A

   A  swift
   B  celiometer
   C  cinder
   D  glance

Openstack 的开源项目最早是由 rackspace 公司和 NASA 提供的，该两家机构提供项目对应正确的是下列哪一项？A

   A  swift 和 nova
   B  neutron 和 nova
   C  cinder 和 nova
   D  swift 和 neutron

) 下列关于虚拟机实例管理中网络配置的描述，哪些是正确的？D

   A  指定租户内，普通_member_角色下用户可以创建私有网络并指定子网的网段
   B  指定租户内，普通_member_角色下用户可以为自己创建的实例绑定 floating IP 地址
   C  指定租户内，普通_member_角色下用户申请 floating IP 地址
   D  指定租户内，普通_member_角色下用户可以创建外网网络并指定外网的网段地址

下列关于创建虚拟机实例时指定镜像位置的描述错误的有哪些？AD

   A  通过 --copy-from 指定一个 NFS 或 SMB 的共享路径
   B  通过 --file 指定本地系统的位置
   C  通过 --location 指定一个 URL 路径
   D  所有说法都正确
```

三、Docker

> 本文包括：
>
> 1、Docker 发展历程
>
> 2、容器技术精髓剖析
>
> 3、Docker 核心技术
>
> 4、Docker 平台架构
>
> 5、Docker 平台的对比
>
> 6、Docker 生态圈及企业应用案例

1、Docker 发展历程

1. 容器和 Docker
   - 容器：在 linux 中，容器技术是一种进程隔离的技术，应用可以运行在每个相互隔离的容器中。
   - 容器与虚拟机的区别：在容器中，各个应用共用一个 kernel
   - Docker：Docker 是一家公司，在 2013 年之前公司名叫 dotCloud，Docker 仅仅是一个容器管理的产品。在 13 年，将 Docker 开源，Docker 风靡全球，公司也更名为 Docker。
2. 容器技术的演进
   - 1979 年，有了容器技术的雏形，root 技术的引进开启了进程隔离技术
   - 2000 年，FreeBSD Jails 将计算机分为多个独立的小型计算系统。
   - 2006 年，谷歌 Process Containers 技术，在进程隔离的基础上，进行了计算资源的限制
   - 2018 年，LXC，第一个完整的容器管理工具
   - 2013 年，LMCTFY 实现了 linux 应用程序的程序化，成为 libcontainer 的重要组成部分。
   - 2013 年，Docker 最初使用的是 LXC，后来被替换成 libcontainer
3. Docker 技术的迅猛发展
   - 在开源之后迎来迅猛的发展，在现在也保持着迅猛的发展势头
4. Docker 技术发展迅猛的原因总结：
   1. 应用架构正在发生变革 --- 微服务化
      - 在互联网时代，为了实现更快的开发迭代和更好的弹性伸缩，互联网应用不再采用传统的 3 层架构，而是采用微服务的方式，解耦，快速交付
   2. 基础架构系统也在发生变革 --- 虚拟化、混合云
      - 从硬件服务器到虚拟机到企业私有云，从本地数据中心到灾备数据中心再到公有云。
5. 他山之石，可以攻玉【容器技术的作用】
   - 一次封装，多次部署，随时迁移，不需要关注底层环境
6. Docker 定义的标准 + 服务应用
   - 基础设施标准化（Docker Engine）：有了 Docker Engine，可将 Docker 容器跑起来
   - 应用交付的标准化（Docker Image）: 提供了一套应用快速打包为轻量级 Docker Image 的方法。开发人员在代码完成之后，可以将其打包为镜像
   - 运维管理的标准化（Docker container）： 运维人员不在需要将应用准备系统、运行环境、组件和基础软件包, 容器时代，应用都运行在一个个的 Docker container 中。标准运维将关注容器，而不是复杂的系统环境。
   - 分发部署标准化（Docker Registry）：指的是容器化之后不同版本的应用镜像都存储在镜像仓库中。

2、容器技术精髓剖析

1. namespace 技术，包含了六项隔离，下面是 namespace 及其对应的隔离内容：
   - UTS ——主机名与域名
   - IPC ——通信（信号量、消息队列和共享内容）
   - PID ——进程编号，通过 PID 技术，同个主机上的不同容器可以有相同的 PID 进程。
   - Network ——网络设备、网络栈、端口
   - Mount ——让文件拥有自己的文件系统
   - User ——用户和用户组，实现用户权限的隔离
2. cgroups 技术

- 通过 cgroups 可以为容器设定系统资源的配额，包括 CPU、内存、I/O 等等。
- 对于不同的系统资源，cgroups 提供了统一的接口，对资源进行控制和统计。
- 限制的具体方式不尽相同，实际的流程很复杂。

3. 其它相关 Linux Kernel 技术
   - selinux 和 apparmor：增强对容器的访问控制
   - capabilities：将超级用户 root 的权限分割成各种不同的 capability 权限，从而更严格地控制容器的权限。
   - netlink：完成 Docker 的网络环境配置和创建
   - 这些技术从安全、隔离、防火墙、访问等方面为容器的成熟落地打下了坚实的基础。
4. 容器管理
   1. lxc
      - 是第一个完整意义上的容器管理技术。通过 lxc 可以方便的创建、启动和停止一个容器。
      - 还可以通过 lxc 来操纵容器中的应用，也可以查看容器的运行状态。
      - Docker 的出现把 2008 年的 lxc 的复杂的使用方式简化为自己的一套命令体系。
   2. libcontainer
      - Docker 后来开发了原生的 libcontainer 代替了 lxc。
      - libcontainer 实际上反向定义了一组接口标准。
      - 反向定义：libcontainer 并不是为了调用底层的 Linux Kernel 技术而设计的，而是 Linux Kernel 技术符合了定义出来的 libcontainer 标准，Docker 引擎才能运行起来。如果此后还有新技术符合这套标准，Docker 引擎还是可以正常运行。
      - 这样的设计思路为 Docker 的跨平台实现和全面化应用带来了可能。
5. Docker 技术原理
   - Docker 构造：Client-Server
   - Docker 结构
     - 装好了 Docker 工具之后，也就同时装好了 Client 端和 Server 端。
     - Client 端可以是 Docker 命令行工具，也可以是 GitHub 上开源的图形化工具。通过 Client 工具可以发起创建、管理容器的指令到 Server 端。
6. Docker Daemon
   - Docker Daemon 通过 libcontainer、lxc 的技术来完成容器管理操作。
   - Docker Daemon 的三个重要组件：
     - execdriver：存储了容器定义的配置信息，libcontainer 拿到配置信息以后调用底层的 namespace 等技术来完成容器的创建和管理。
     - networkdriver：完成容器网络环境的配置，包括了容器的 IP 地址、端口、防火墙策略，以及与主机的端口映射等。
     - graphdriver: 负责对容器镜像的管理。

3、Docker 核心技术

1. VM Vs. Docker
   - 应用在 Docker 中跑起来是什么样的？
   - Docker 和虚拟机的结构对比
     - 虚拟机下，是硬件→操作系统→Hypervisor→虚拟机操作系统→配置虚拟机依赖环境→安装 App
     - 容器时代，硬件→操作系统→Docker Engine→运行所需要的依赖环境→安装 App
     - 容器的运行不需要安装虚拟机的操作系统，是比虚拟机更加轻量的虚拟化技术。
   - Docker 和虚拟机对比的详细信息
     - Docker 容器共用一个 Kernel，而虚拟机使用自己操作系统的 Kernel→虚拟机拥有比 Docker 更好的隔离性。
     - Docker 有更多的优势：
       - 虚拟机操作系统的存在，占用了更多的计算资源；
       - 空间占用上，虚拟机是 GB 级，Docker 可以小至几 MB；
       - Docker 的启动时间更快（毫秒级）；
       - Docker 有更强的快速扩展能力、跨平台迁移能力（虚拟机无法从VMware 迁移到 KVM）
2. Docker 的重要概念：容器、镜像、仓库
   - 容器：与虚拟机一样都是承载相关应用的载体
     - Docker 容器的状态
       - 三种状态：Running、Stopped 和 Paused，与虚拟机的运行、关机、挂起状态相似。
   - 镜像：当容器安装了特定的应用之后，就可以将其打包成镜像。需要应用的时候，将镜像下载下来，就可以快速运行起来。
     - 镜像的生成
       - 当把镜像下载到本地之后，可以使用 Docker run 的命令，启动一个基于这个镜像的容器。对容器修改之后，可以将其 commit 回去，生成一个新版本的镜像。

        - Docker 镜像是一种层级结构的文件系统，最上层往往是可写的，存储了已经运行的容器的修改信息当对容器进行 kill 的时候，修改信息就会被删掉。当容器被 commit 成镜像的时候，这些修改信息也会保存成新的层级。
        - 镜像的生成除了使用 commit 之外，还可以使用 Dockerfile（更标准、更常用）。这种方式 build 生成出来的镜像更加干净、透明。
    - 仓库：存储镜像的地方。
        - Dockerhub 和私有仓库
            - Dockerhub 是 Docker 的官方仓库，存放着各种官方的标准镜像。可以使用 pull 命令直接从 Dockerhub 中下载镜像到本地进行使用。
            - 还可以构建自己的镜像仓库，用于存放常用的镜像以及企业自定义的应用镜像。可以从私有仓库中下载、上传镜像。
3. Docker 核心技术：Build, Ship, Run
    - Docker 的主要操作
        - 首先利用 Dockerfile 将组建 Build 成为一个镜像，然后将镜像上传到企业自定义的镜像仓库中。
        - 当我们需要镜像的时候，可以从任何地方连接到镜像仓库中，将镜像下载到本地，然后一键将其 Run 起来。
        - 这就是 "Build once, run everywhere"：一次构建，任何地方都能运行。
4. Docker 数据卷
    - 保存 Docker 数据
        - 容器一旦关闭，修改信息就会丢失，这对于一些有状态的应用来说往往是不可接受的。
        - 可以通过给文件挂载文件目录或者存储来解决，从而可以存储容器运行中的一些数据。这样，当容器崩溃，重启容器的时候，依然可以访问之前容器存储下来的一些数据。
        - 这种方式也可以解决一些主机和容器之间的数据访问。
5. Docker 网络
    - 实现容器之间的通信、容器与外部之间的通信。
    - 四种模式：
        - Bridged：表示容器可以与主机上的容器，主机上的外部进行通信
        - Host：表示容器只能与主机通信
        - Container：表示容器只能与容器通信
        - None：没有网络连接
    - 比较常用的是 Birdged 模式。主机会生成一个 Docker 网桥，每个容器可以拥有自己的虚拟网卡，容器网卡通过网桥连接到主机的物理网卡，与外部进行通信。

4、Docker 平台架构

1. 容器编排（orchestration）
    - 这里的编排泛指广义的编排，用于管理容器下面的主机，管理容器以及日期之间的逻辑关系，即为我们所知的应用架构。
    - 集群管理：【关键词：配置管理、资源视图、节点增删、高可用】
    - 容器调度：【关键词：容器部署、调度策略、互斥】
    - 故障恢复：【关键词：主机检查、容器检查】
    - 应用编排：应用是一个个细微的容器来组成，每一个容器之间具有一定的逻辑关系，我们需要使用简单明了的编排语句将这些容器关联起来，比如容器之间的端口访问，容器的启动顺序等，从而实现整个应用的逻辑架构。
2. 编排主流的 3 大工具
    - Swarm: Docker 2014 年发布 内置于 Docker，和 compose 一起使用
    - Mesos: Apache 2007 年发布 一般会结合 marathon、Aookeeper 一起使用
    - Kubernets: Google 2014 年发布 结合 Etcd 一起使用
3. 负载均衡和服务发现
    - 负载均衡：请求到达负载均衡器之后，负载均衡器平均的分配到后面的容器上。
        - 常用的负载均衡的技术：haproxy,LVS,F5,Nginx
    - 服务发现：服务发现会自动将容器的配置信息上传至配置中心（config center），包含了容器的 IP、 端口、对外的域名等。
        - 常用的服务发现的技术：Etcd,Zookeeper,Consul
    - 负载均衡器会周期性的从配置中心获取配置信息，并且将容器加入到相关的负载均衡访问架构中
4. 日志管理
    - 日志：包含主机、编排工具、日志、容器、容器中的应用等等相关的日志
    - 对日志处理平台的要求：集中化、海量存储、灵活过滤、快速查询、伸缩性架构、高可用、强大的 UI
    - 日志管理软件：ELK， 包含了 3 个组件:
        - Logstash，用于收集各种各样的日志
        - ElasticSearch: 主要用于存储和搜索日志
        - Kbana: 用于界面展示的管理工具
5. Docker 监控
    - 包括 主机 镜像 容器 应用等维度进行监控

- 构建相关的告警, 跟踪，监控等监控流程体系，即 WANT 原则(Watching(监控)、Answer（响应）、Notify（通知）、Track（跟踪），总结起来就是 WANT)
- 常用的监控工具：Zabbix, Nagios, cAdvisor, Datadog, Scout 等

6. Docker 平台架构
   - 平台层：底层需要对计算，网络，存储等资源进行管理，结合容器引擎和容器编排的工具实现对容器化应用的落地
   - 能力层：需要实现容器化应用的弹性架构，负载均衡等等，需要实现平台的统一监控和统一的日志管理，需要有严格的权限体系实现对用户，租户的管理，需要结合当前的 deverups 理念来实现对应用的持续交付和版本控制
   - 最上层：平台还需提供应用的 Web UI 和功能齐全的 API 服务
7. Docker 平台技术体系
   - Orchestration(编排): Swaim, Kubernetes, Mesos
   - Cluster Management(集群管理): Etcd, Zookeeper, Haproxy
   - CI/CD(持续集成): Jenkins, GitLab, Registry
   - Database(数据库): redis, MySQL, mongoDB
   - ELK&CMDB(日子和配置管理): Logstash, Chef, puppet
   - Monitor(监控): ZABBIX, Nagios, sysdig

5、Docker 平台的对比

1. Mesos 介绍
   - 两个重要角色，一个是 Slave, 安装集群节点上的，还有 Master, 作为集群的管理节点，slave 会将节点的资源使用情况周期性的报告给我们的 master 节点。
   - Mesos 需要配合架构的 FrameWork 进行资源调度，过程如下：
     - master 会定期将计算机节点的资源使用情况周期性的报告给我们的 Framework Scheduler
     - Framework Scheduler 进行调配后，下发部署的任务给集群节点
     - 节点上的 framework Executor 获取任务进行容器的部署（在容器的架构中，Executor 就是容器的引擎）
     - 节点会将部署结果反馈给 master
     - Master 会更新主机资源的状态给 Framework Scheduler
   - Mesos+Marathon+Zookeeper
     - Marathon 是一个可以调用 Docker 引擎的 framework，可以将容器按照一定的调度策略部署到合适的主机上。
     - Zookeeper 的作用是保证 Marathon 和 Mesos 来管理节点的高可用性，即当 master 节点宕机之后，可以快速的选取出新的 master 节点，从而不影响逻辑架构。Zookeeper 本身是一个分布式的高可用的架构
2. kubernetes 的介绍（Goolge 的开源容器管理项目）
   - 重点：mesos 中的最小的单元是容器，但是 kubernetes 的最小的单元是 Pod。容器被封装在 Pod 中，一个 Pod 中可以存放一个或者多个容器。
   - 设计 Pod 的目的：将需要紧密联系的 Docker 容器放置在一个独立的空间（Pod）内。
     - 同个 pod 中的容器可方便的共享存储
     - 同个 pod 中的容器可直接访问另一个容器
   - kubernetes 整体架构：在部署 kubernetes 之前，需要先部署 Etcd 作为集群的管理工具
   - 2 个重要的角色：minion（普通节点）和 master（管理节点）
3. Swarm 的介绍（Docker 公司自己的容器管理工具）
   - 1.12 版本之后，Swarm 已经封装进 Docker 引擎中，并且自带服务发现的功能
   - 2 个重要角色：manager node 和 worker node
   - Swarm1.12 之后，自带服务发现的功能，可以做到 manager node 的高可用性。
   - Swarm 也内置了负载均衡的技术，使用的是 LVS，但是依然支持和其他服务发现的工具，如 ETCD，Haproxy
4. Docker 平台对比【表格无法显示】

	Mesos	Swarm	Kubernetes
应用定义	应用组定义的容器集合	Compose中定义的容器逻辑集合	Pods的集合
应用扩展	可以容器或应用组为单位进行扩展	以容器为单位进行扩展	以pod为单位进行扩展
高可用	一般依赖于Zookeeper	依赖于自己或etcd等工具	依赖于Etcd
负载均衡	可依赖于marathon-lb	自身集成lvs	可依赖于haproxy
日志和监控	使用外部工具	使用外部工具	可使用自带工具，也可使用外部工具

6、Docker 生态圈及企业应用案例

1. 应用场景：快速交付与 CICD

- 企业应用的开发上线流程一般是：代码、构建编译、测试、发布、部署
- 遇到的问题：可能因为环境的问题导致上线延迟，测试不通过等。
- 快速交付：Docker，通过 Docker 可以大大的提高环境交付的质量和速度，开发人员写好代码之后，交付的不在是一大堆的附属文档，而是一个个的镜像存储到镜像仓库中。在测试环境、预生产环境以及生产环境将镜像仓库中的镜像拉取出来即可。保证部署出来的所有应用都是标准的、统一的。即为实现了应用的快速交付。
- CICD：持续集成和持续部署（Constant Integration Constant Deployment）当我们的代码更新时，开发人员可以构建一个新的镜像版本到镜像仓库中，运维人员可以快速的将我们的镜像应用到测试环境、预生产环境以及生产环境。甚至可以通过金 KISS 实现整个更新的自动化，从而实现了持续集成持续部署，实现了应用开发环境的快速迭代。

2. 应用场景：云间迁移
   - 应用容器化之后，对底层环境的要求将大大的降低，应用可以实现从本地数据中心到 ALS，阿里云、公有云等迁移
3. 应用场景：弹性扩展
   - 企业应用容器化之后，应用的扩展就是拉取镜像部署更多的容器的简单的过程，我们可以部署相关的监控系统，当发现应用访问慢或者是资源紧张的时候，在弹性扩展的策略下，应用会自动增加相应的容器实例，从而减轻应用访问的压力。当集群中的主机资源不足的时候，还可以使用 IaaS 接口，自动的增加主机的数量，以便于创建更多的 Docker 容器。
4. 应用案例：
   - 平安 Padis 平台
   - 京东 618（基于 openstack 和 Docker）
   - 天猫双十一
5. Docker 巨大生态势能
   - 从安全架构领域，操作系统领域、网络、存储、安全、安全、监控、日志等方面，越来越多的公司卷入到 Dockers 的发展潮流当中。
6. 基于 Docker 的产品：
   - 红帽 openshift
   - 阿里云容器服务
   - Azure 容器服务
   - 网易蜂巢
   - 道客云、有容云、希云、时速云等

test

```
Docker 容器本质上是宿主机上的进程，宿主机只能是硬件服务器。B

   A 对
   B 错

下列哪个不是 Docker 引擎的组件？C

   A Docker daemon
   B Docker CLI
   C Docker registries
   D Rest API 接口

下列关于 Docker 的 namespace 特征特性描述中哪一项是不正确的？C

   A 不同用户的进程通过 pid namespace 隔离开
   B 通过 net namespace 实现网络隔离
   C namespace 不允许每个 container 拥有独立的 hostname 和 domain name
   D mnt namespace 允许不同 namespace 的进程看到的文件结构不同

在 Docker 中，关于容器的描述中不正确的哪项？D

   A Docker 利用容器来运行应用
   B 容器看做是一个简易版的 Linux 环境
   C 每个容器都是相互隔离的
   D Docker 容器比较适合运行多个进程

 Docker 作为成功的容器解决方案，具有下列哪些特性？ACD

   A 简单易用
   B 无需维护
   C 跨平台
   D 可移植
```

四、微服务

> 本文包括：
>
> 1、从单体架构到微服务架构的演进
>
> 2、基于 Docker 的微服务架构设计
>
> 3、基于容器的微服务架构剖析
>
> 4、微服务架构设计模式
>
> 5、微服务云架构管理

1、从单体架构到微服务架构的演进

1. Monolithic 单体式架构解析
   - Monolithic 单体式架构指的是：尽管是模块化逻辑，但是最终还是会打包并且部署为一个单一应用，具体的格式依赖于具体的语言和框架
   - 例如，部分 java 应用会被大包围 WAR 格式，部署在 Tomcat 上或者 JIT 上，而另外一些 java 应用会被打包为自包含的 jar 格式
   - 同样，Reals 和 node.js 会被打包为层级目录。
2. Monolithic 单体式架构的优缺点
   - 优点：开发工具 IDE 和其他工具都擅长开发一个简单应用，这类应用也很易于调试和部署。只需要把打包应用拷贝到服务器端，通过在负载均衡器后端运行多个拷贝，就可以轻松是实现多个扩展
   - 缺点：单体式架构一旦随着时间的推移，逐渐的变大，敏捷开发和部署举步维艰。任何单个开发者都很难搞懂它。修正 bug 和正确的添加新功能变得非常困难且很耗时。
3. Monolithic 单体式架构面临的挑战
   - 随着市场变化快用户需求变化快、用户访问量增加的同时，单块架构应用的维护成本、人员的培养成本、缺陷修复成本、技术架构演进的成本、系统扩展成本等都在增加。
   - 单块架构的曾经的优势已逐渐不在适应互联网时代的快速变化。
4. 微服务架构模式倡导的做法
   - Microservice 微服务架构是一种架构模式，提倡将 Monolithic 单体式架构应用划分为一系列小的服务，服务之间相互协调，相互配合，为用户提供服务。
   - 每个服务运行于其独立的进程中，服务之间采用轻量级的协议进行通信，每个服务都围绕着具体业务进行构建，并能够独立部署。
   - 微服务架构的优点：每个服务能够内聚，代码容易理解，开发效率高，服务之间可以独立部署，使得持续部署成为可能，容易正对每个服务组件开发团队，容错性也大大提高。
5. 向微服务架构演进的推荐顺序
   - 先规划，然后是中间件和数据库，最后是服务和应用

2、基于 Docker 的微服务架构设计

- 微服务架构设计需要遵循的模式
  - 比较知名的 “12-factor”
- “12-factor” 为构建如下的 SaaS 应用提供了方法论
  - 使用标准化流程自动配置，从而使得新的开发者话费最少的成本加入这个项目；和操作系统之间尽可能的划清界限，在各个系统中提供最大的可移植性，适合部署在现在的云计算平台，从而在服务器和系统管理方面节省资源。
  - 将开发环境和可生产环境的差异降至最低，并使用实施交付实现敏捷开发。可以在工具、架构、开发流程不发生明显变化的前提下实施扩展。
- “12-factor” 中比较常用的几个：
  - 基准代码
    - 基准代码和应用之间总是保持一一对应的关系：一旦有多个基准代码，就不能称为一个应用，而是一个分布式系统。分布式系统中的每一个组件都是一个应用，每一个应用可以使用 “12-factor” 进行开发。
    - 多个应用共享一份基准代码是有悖于 “12-factor” 原则的，解决方案是将基准代码拆分为独立的类库，然后使用依赖管理策略去使用管理他们。
  - （显示声明）依赖关系
    - 应用程序不会隐式依赖系统级的类库，他通过依赖清单，确切的声明所有的依赖项。
    - 此外，在运行过程中通过依赖隔离工具来确保程序不会调用系统中存在倒是清单中未声明的依赖项。这一做法会统一应用到生产和开发环境。

- 配置（config）
  - 推荐将应用的配置存储于环境变量中，环境变量可以非常方便的在不同的部署之间做修改，却不动一行代码。
  - 与配置文件不同，一不小心把他们签入代码库的概率微乎其微
  - 与一些传统的解决配置问题的机制（比如 Java 的属性配置文件）相比，环境变量与语言和系统无关。
- 后端服务
  - “12-factor” 应用不会区别对待本地或者第三方服务，对应用程序而言，两种都是附加资源。
  - “12-factor” 应用支持任意部署，如：可以在不进行惹任何代码改动的情况下，将本地的 mysql 数据库换成第三方服务。

3、基于容器的微服务架构剖析

1. Docker 的本质
   - Docker 是一个微容器，一个云计算的微 PaaS 容器，类似 JVM，但是比其更加强大的容器，直接基于各种内核，支持各种语言；
   - 比 VM 虚拟机更加的轻量，能够在 linux 或者 IaaS 云平台上直接运行，带着你的应用运行到各种运行环境。
   - Docker 本质是一个允许你创建镜像，并且让这个实例运行在容器中的软件。
2. Docker 流行起来的因素
   - Docker 的细粒度松耦合能够让我们用一个 Docker 容器装载一个场景功能，即按照功能角色分类。
   - 每个 Docker 里面装一个应用或服务，一个服务器上可以运行多个 Docker，一个系统级别的服务，比如 mysql 数据库
3. 集群分布的微服务
   - 让每个 Docker 中运行一个微服务
   - 通过分布式集群
   - 比如 Spring Boot
4. Docker 对传统 PaaS 平台的挑战
   - Docker 比 Java EE 服务器更强的地方在于：Docker 是基于 linux 内核，可以装载各种语言应用，是一种崭新的 PaaS 微平台。
5. Docker 的安全性
   - Docker 利用容器将资源进行有效的隔离，因此，与 linux 系统和 hypevisor 有着相同的安全运行管理和配置管理级别。
   - 但当 Docker 运行在云供应商平台上时，Docker 变得更加的复杂，你需要知道云供应商平台正在做什么

4、微服务架构设计模式

1. 客户端如何访问这些服务？
   - 原来的 Monolithic 方式开发，所有的服务都是本地的，UI 可以直接调用，而现在按照功能拆分成独立的服务，通常都是运行在虚拟机上的 Java 进程。
   - 后台有 N 个服务，前台就需要记住管理 N 个服务，一个服务下线、更新、升级，前台就需要重新部署，这明显不符合拆分的理念。特别当前台是移动应用的时候，通常业务变化的节奏更快。
   - 另外，N 个小服务的调用也是一个不小的网络开销。所以，后台 N 个服务和 UI 之间会通过一个代理，或者叫 API Gateway 统一提供服务入口，让服务对前台透明，聚合后台的服务，节省流量，提升性能，提供安全、过滤流控等 API 管理功能。
2. 服务之间如何进行通信？
   - 所有的服务都是独立的 Java 进程，跑在独立的虚拟机上，所以服务间的通信就是 IPC（Inter process communication）已经有很多的成熟的方案。
   - 基本最通用的是两种方式：同步调用 REST 和 IPC 异步消息调用
3. 这么多的服务，怎么找？
   - 服务发现
   - 一般每个服务都有多个拷贝来做负载均衡，一个服务随时可能下线，也可能应对临时访问压力，增加新的服务节点。
   - 如何发现：基本都是通过 Zookeeper 等类似的技术做服务注册信息的分布式管理。
   - 解释：当服务上限时，服务提供者将自己的服务信息注册到 ZK 或者类似的管理框架，并通过心跳维持长链接，实时更新链接信息，服务调用者通过 ZK 寻址，根据可定制算法找到一个服务，还可以将服务信息缓存在本地，以便提高性能。当服务下线时，ZK 会发通知给服务的客户端。
4. 服务挂了怎么办？
   - 分布式一个最大的缺点是：网络是不可靠的
   - 一个提供高容错性的工具：HYSTRIX
   - 当系统是由一系列的系统调用链组成的时候，我们必须确保任一环节出问题，都不至于影响整体的电路，相应的手段有：重试机制、限流、负载均衡、降级（本地缓存）

5、微服务云架构管理

1. 微服务简化

- 基于容器技术的云服务将极大的简化容器化微服务创建、集成、部署、运维的整个流程，从而推动微服务在云端的大规模实践。

2. 微服务的创建
   - 假设用户的微服务程序，存储于 github 等代码托管服务中，用户可以将这个代码仓库构建成容器镜像，并保存在镜像仓库中，用户可以将这个微服务一键部署到容器云平台。
   - 云平台提供了持续集成的功能，用户可以选择是否使用，每当微服务的代码有变化时，就构建一个新的微服务的镜像，以便以后部署使用。
3. 微服务的集成
   - 用户可以自由组合、复用数以万计的容器化微服务，像搭积木一样轻松集成应用。比如，用户需要一个通用的 mysql 数据库服务，他无需构建镜像，可以直接在镜像仓库中选择合适的数据库服务镜像，并与其微服务连接起来。
4. 微服务的部署
   - 微服务由于组件数量众多，云端部署成为实践上的一个难点
   - 容器云平台容器为应用发布的载体，用户不必指定传统部署方式中繁琐的步骤，只需要提供容器镜像和简单的容器配置，平台会将整个部署流程自动化。
5. 微服务的运维
   - 微服务由于独立进程众多，部署后的运维、管理成为实践上的另一个难点
   - 容器云平台完全屏蔽底层云主机和基础架构运维，让用户专注于应用。
   - 通过容器编排、自动修复、自动扩展、监控日志等高级应用生命周期服务，实现容器化服务的智能托管。

test

微服务架构每个服务都有自己的数据库，而不像传统架构下多个服务共享一个数据库。A

A 对
B 错

应用想要做到持续部署就必须缩小本地与线上差异。开发人员应该反对在不同环境间使用不同的后端服务，即使适配器已经可以几乎消除使用上的差异。A

A 对
B 错

微服务被定义为特定背景下，为了开发出速度更快、更有弹性且用户体验更佳的应用而采用的自动化系统，下列哪一种场景环境不适合微服务？A

A 必须理解其他部件运作原理
B 松耦合
C 运行环境可以是 Docker 容器
D 面向服务的架构

关于微服务 12 要素中的 “Dependences 依赖”，下列哪一项理解是不正确的？B

A 应用程序不会隐式依赖系统级的类库。
B 这一做法应用到生产和开发环境时可以不一致。
C 在运行过程中通过依赖隔离工具来确保程序不会调用系统中存在但清单中未声明的依赖项。
D 一定通过依赖清单 ，确切地声明所有依赖项。

单体式应用的特点不包括下列哪一项？C

A 尽管也是模块化逻辑，但是最终它还是会打包并部署为单体式应用。
B 具体的格式依赖于应用语言和框架。
C 每一个单体式应用的模块都是微型六角形应用，都有自己的业务逻辑和适配器。
D 单体式应用在不同模块发生资源冲突时，扩展将会非常困难。