CVM技术十年发展史-策划稿

powellli, 2022-01-04

背景

原计划1月初到西安与计算产品中心同事交流，因疫情取消，但答应了HR给全体西安同事讲一门课，定于1月5日下午15:00到16:30。

考虑到受众特性，定下如下几个原则：

1. 以技术作为主线，但不深入进去，以点带面，快速全面介绍计算视角的云业务
2. 通过技术发展史作为叙事推进，强化史诗感和使命感
3. 重点突出不怕困难，敢为人先的价值观

内容概要

1. 切入（Page 1）

讲下背景：本来是要过来，结果因为疫情而无法当面交流，就像罗振宇《时间的朋友》取消了观众席

讲下感想：上次来西安是2021年七八月份的时候，本来是打算开年就来，也是因为疫情。当然这也不一定是坏事，说明我们已经率先一脚踩在元宇宙的门槛上。当时与计算产品中心西安的同事交流，有几点感想：1. 年轻热血 2. 希望做一件伟大的事业。一如十年前的我自己

讲下自己：十年前，被“互联网吞噬一切”而感召，来到腾讯做云

讲下主题：今天的分享主题是《CVM技术十年发展史》，重点在历史而不在技术

2. 云计算的本质是什么（Page 2/3/4）

两个神话

babel tower

西绪弗斯

3. 第一场战争：应用还是机器？(Page 5/6)

AppEngine VS VM，还是SaaS

腾讯云最早是什么时候提供云计算服务的，是有哪些形态

腾讯云计算的几个阶段，也是这次的叙事主线

4. VStation：一切的起点？（Page 7-21）

从物理机到AppEngine，到虚拟机

CVM是起名最差的产品，应该怎么起名呢？

OpenStack的问题：

> 云是一个大型的系统级的工程，它看不惯过往一切无法用软件定义的基石，所以云的终极目标是同时要保证稳定与灵活。具体来看腾讯云当时的系统，流程繁琐、调用冗长，众多模块逐渐形成一个mesh的脆弱结构，容灾能力、可追溯能力、部署能力都存在极大的挑战。与之相比，OpenStack就显得先进得多：restful的接口设计、独立互不依存的模块划分、清晰有意义的分层结构以及完整而灵活的driver设计，是一个逼格满分的解决方案。
>
> 然而，经过仔细分析，OpenStack的诸多闪光点却并未照亮我们急需解决的几个问题，主要包括：
>
> 1. 可维护性
>
> OpenStack是大而全的解决方案，不论是什么云它都希望能帮得上忙。这导致其设计极度考虑灵活性，在一切可能变化的地方都会为了扩展和改进做大量的工作。
>
> 但是，OpenStack的任何使用者都只需要使用它极小的一部分配置集合，而且其它开发者也不一定愿意在理解了它完美的层次再进行开发（参照neturon的部分厂商代码）
>
> 这导致OpenStack的部署与配置十分复杂，而且这让你感觉很疲惫，它无法非常直接地解决你的问题，而总是要正襟危坐、小心翼翼地通过各种包装、代理、驱动才接触到问题，还不一定符合期望。
>
> 而且OpenStack作为docker以前开源社区最热的项目，代表了开源项目的最高峰，有一个显著的特点：尽一切可能把开源的兄弟们拉入伙！于是你看到OpenStack会尽量与各种开源工具融合起来，这无形中增加了运维的成本。OpenStack本来就不简单，需要理解大量你可能未曾预料的巧妙，还需要学习各种其它的开源项目。
>
> 对于云的后台框架，我们必需要对其理解得十分透彻，如果光是打开代码都会导致内存不足，我是不愿意在半夜起来处理故障时面对它们的。
>
> 2. 容灾能力与高可用
>
> 跟上一条类似，早期OpenStack似乎压根没有想过高可用部署，这实在是让人匪夷所思的一个考虑。
>
> 直到我到香港参加OpenStack Summit时，有一个专场是讨论高可用部署，我才发现OpenStack的高可用方案又拉了一堆的大型的开源软件进到你的生产系统。

在腾讯，高可用是后台服务必备的能力，我们从未想过通常不需要花费太多精力的工作在OpenStack上竟然有点让人畏惧。

3. 可追溯能力与流程回滚

可追溯能力是复杂的分布式系统中非常重要的一点。在我们的『全mesh』云平台中经常会遇到一个流程突然中断，却很难找出在哪里失败，为何失败，更重要的是如何回滚。

这在早期OpenStack中我并未看到任何更先进之处。看似随意的rpc、逻辑的散落、精巧但是难以追溯的eventlet都增加了异常发生时你定位的困难和恼怒的情绪。

也许是我没入对门，即便是我看过很长时间代码，仍然很难快速定位出哪怕是因为配置错误导致的流程失败。

想到以后要天天加班到凌晨，我赶紧跟领导说：我们自己来吧！

4. 性能与稳定性

重要的事情再说一遍：随意的rpc。

消息队列受到影响，甚至无法保障通信的可靠。

凡是有用OpenStack管理过大规模宿主机的，应该都踩过这个大坑。

1. 尊重消息流转。可追溯性是第一优先级。

mesh结构类似于网络工程中节点互相访问的场景，如果都是点到点通信，那么节点规模的线性增长会导致信道的指数级增长。

FDC使用RabbitMQ做为统一的通信总线（类似于以太网的思路），并且明确模块监听的主题名称。强一致性的RabbitMQ保证了消息的准确送达，并且通过主题名称实现系统中的服务发现，还自带了足够稳健的负载均衡与失败重传。而且，由于模块间的通信队列是固定的，我们可直接使用RabbitMQ的已有功能对消息进行追溯、统计、监控。

虽然都用了消息队列，但FDC与OpenStack在这方面的设计原则是完全不同的。OpenStack弱化消息流转的过程，试图让你忘记你处在分布式的系统中，简化代码调用的同时鼓励了不必要的模块间调用，并且在灾难发生时你不得不面对这层外衣下的真实世界。

2. task flow的支持。异常和灾难对云是常态，但不要让用户感知到。

既然我们没有把MQ仅仅当作是一个更好用的socket，而是对消息流转有了严格的编排，那么框架就能很容易地发现异常发生的时间、模块以及模块返回的具体错误信息，实现task flow的功能。

那么框架同样也可以编排异常发生时的消息流转图。对于大部分模块，只需要它的每个原子接口都有对应的回滚接口，那么回滚的逻辑就很简单了，只需要在当前失败步骤开始，逆序调用所有步骤的回滚接口即可。

我在知乎上发招聘广告的时间，都是这样省出来的。

3. 简洁可靠闭环。『Batteries Included』，拿来就用。

为了有更多发广告的时间，我们还希望项目规模能简洁，不要有太多过早优化；能可靠，代码不多但都经得起考验；能闭环，自带电池直接就发光发热。

『Batteries Included』是Python的一个设计哲学。与外部组件的融合往往会带来大量的适配工作，而且你只需要外部组件的一个功能子集却要维护它整个组件。不能拳拳到肉的做法通常是项目偏离需求的开始。对于很多简单直接的需求，直接用几十行几百行代码来实现，至少比引入几万行代码的依赖要好得多，在你需要的时候也能完备的方法替代电池（如Python有哪些黑魔法？ – 李力的回答）。关于『Batteries Included』与『Don't Repeat』并不冲突的探讨这里就先不深入了。

FDC的task flow是自带的，回滚是自带的，通信接口是自带的，容灾和高可用也是自带的。

4. 模块解耦。逻辑上的解耦胜过形式上的解耦，对模块开发者，尽量减少约束。

每一个程序员都有关于框架的梦想，它的意义在于对使用框架者的约束，约束意味着权力，权力意味着阶级。

框架的开发者们说：快看，我找到解决xx问题的最佳实践，只要你按照我的框架，就能写出跟我一样牛逼的代码。这个思想最早体现在巴别塔建造工程上，也体现在网易评论里：如果全国人民都给我1块钱，我就有13亿了！

嗯，回到云的框架。云的框架和大部分的框架还不一样，它更复杂，更无法预料哪里未来要发生变化，毕竟software define everything，那么越是过早优化越是难以回头，越是限制太多就越容易脱离控制。就像是网络硬件厂商想要接入neturon，工程师为了早点下班，根本就不愿意像OpenStack本身的核心开发一样去理解它的庞大而且精细的类层次和设计，直接找到个入口就把代码全塞进去了。

我想起我第一次写CGI时总是理解不了，CGI代码里没有任何与web server、http相关的东西，它是怎么能被浏览器访问到的呢?

FDC的模块要求类似于CGI，模块本身不需要理解MQ，也不需要理解task flow，也不需要考虑容灾和高可用，这些都完全由框架和RabbitMQ做完了。甚至模块只有一个入口，以回调函数或者可执行文件的方式提供，框架负载输入，模块返回输出即可。模块甚至也不知道自己将会被框架attach进去，这对现有系统的接入和改造的成本也是极低的。

5. 高性能，可平行扩展。至少要支持十万级别的宿主机。

其实这点真的不难，真的。

把大象关冰箱，总共分几步?

ZStack

张鑫：OpenStack -> CloudStack -> Zstack

梁胜：OpenStack -> CloudStack -> Rancher

CloudStack无法响应用户需求
ZStack卖产品而不做项目
一个整体架构（monolithic architecture）-> 单体架构
微服务作为java线程存在
一致性Hash的问题

阿里云

挖煤

微服务：RPC的问题，强一致性到最终一致性的妥协。需要的功能有配置管理、服务发现、熔断器、智能路由、微代理、控制总线、一次性令牌、全局锁、领导选举、分布式会话、集群状态
MF的特点：控制权的上移，上层决策和调度，更适合已有系统的接入
RPC -> FLOW
RESTFul -> MQ

5. 大规模调度能力(Page 22-27)

每周三上午10点的崩溃故事

卧底qq群，阿里云清退

腾讯云是否清退呢?

引入大规模分布式调度

在正式介绍分布式调度之前，我们首先来明确调度的含义。事实上，在分布式系统中，调度的概念比较广泛。主要包括以下 2种：
任务之间的调度，负责管理任务间的关系，典型系统 /组件包括 Hadoop YARN 中的 Application Master 和 Airflow。
任务资源调度，负责为任务分配资源，典型系统包括 Google Borg、Mesos、Kubernetes 等。
在腾讯云中，不同的产品会解决不同的调度问题，例如批量计算 Batch 主要解决任务之间的调度问题，本质上属于工作流产品；云主机 CVM 则主要解决任务的资源调度问题。本文主要关注 CVM 产品，聚焦讨论的是第二种调度——任务资源调度。

Page25
在异构化、规模化的背景下，针对调度质量、吞吐率等问题，公司和高校都做了很多工作
Google 和 UC Berkeley的同学，针对这些问题，其实就提出了他们认为的调度系统的演变规律：即从统一调度架构、演化出两级调度架构，进而又会出现共享状态的调度架构

如图所示，左侧的架构即为统一调度架构，下方是集群的宿主机；中间是集群状态信息，用于保存宿主机的资源状态；上方是统一的调度器，负责接收调度请求，并在集群状态信息的基础上进行调度决策。许多调度系统最初都被设计为这种架构，例如第一代 Hadoop MapReduce。
这种架构，设计简单，可以便捷的保持资源数据一致性，但是当宿主机规模增大时，调度器处理单次资源调度请求的时间会开始增加。当资源调度请求增大到一定程度时，调度器的吞吐量不足，调度请求开始排队，造成任务阻塞积压。

如果发生调度冲突，VStation 会选择次优宿主机。相比 Google Omega 重新调度的做法，对调度冲突的处理代价显著减小。在公有云海量并发创建的场景下，VStation 在调度决策和调度吞吐率进行权衡，选择次优解来保证调度吞吐率

榜一大哥：相同配置买一万台

6. 裸金属化/异构化/私有化（Page 28-32）

略

7. 新的征程 （page 33-40）

良心云

Orca取自“orchestra”，管弦乐、编排。寓意云操作系统像是交响乐园指挥手编排、协调和管理整个云计算中的资源和服务。

另外，Orca刚好也是虎鲸的英文名，海洋系，是海豚科下最大的物种，分布于几乎所有的海洋区域，寓意云原生操作系统可以无处不在。

云原生操作系统Orca品牌还有丰富的品牌特质，开源开放、稳定可靠、创新创造、行业先进，首字母拼成ORCA，为品牌赋予更多内涵

基于腾讯20多年云架构技术积累与海量业务锤炼，腾讯云遨驰Orca秉承【开源创新】的产品理念，打造行业首家【全域治理】的云原生操作系统，

为用户提供高度标准化【无限算力】，以最贴合用户便利的方式【触手可及】

8. 结语

前栽树，后乘凉，求仁而得仁，痛饮一杯春风桃李

朝装逼，夕作死，一日复一日，仗剑十年夜雨江湖