# 2021 年云计算技术复习归纳

考试时间：2021-6-8 2:00 PM-4:00 PM
考试地点：A3-207 A3-208
试卷语言：中文
考试题型：选择题、简答题和计算分析题

小贴士：题量和计算量较大，考试时请抓紧时间，并携带计算器。

目录



## 1. 云计算概述

### 1.1 定义

云计算是一种信息技术（IT）范例，它可以无所不在地访问系统资源和高级服务的共享池，这些资源通常可以通过 Internet 以最少的管理工作来快速进行配置。

### 1.2 基本属性

（1）按需随选（on-demand）：资源使用者可以根据需要单方面获取计算服务。

（2）资源池（Resource pooling）：云提供商的计算资源是池化的。

（3）无处不在的网络访问（Ubiquitous network access）：云服务和资源可通过异构网络访问。

（4）位置独立（Location independence）：资源的使用者不用关心资源的位置。

（5）快速弹性（Rapid elasticity）：可以快速而灵活地提供功能。

（6）按需付费（Pay-as-you-go）：服务的使用者按照用量付费。

（7）多租户（Multi-tenancy）：可以在互不可见的多个使用者之间共享应用程序和资源。

### 1.3 概念和术语

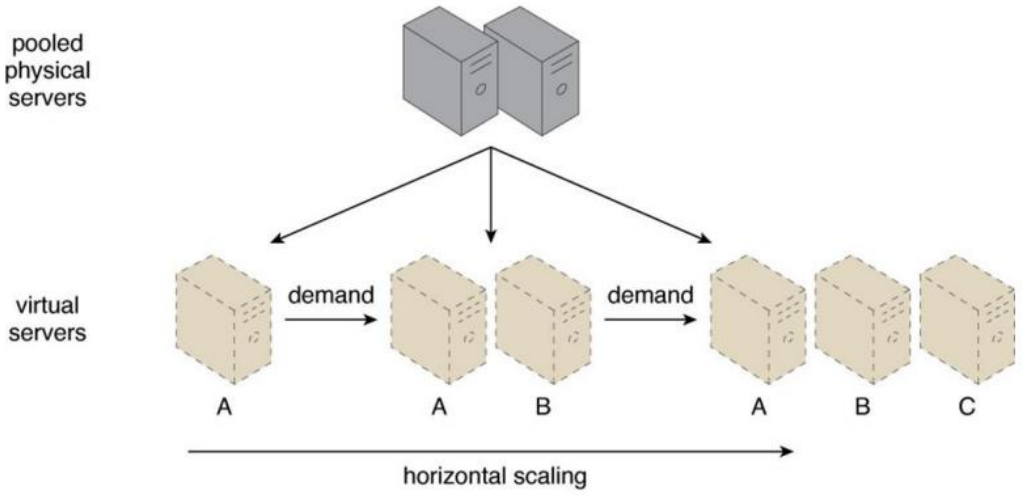
（1）可扩展性

从 IT 资源的角度来看，可扩展是指 IT 资源可以处理增加或减少的使用需求的能力。

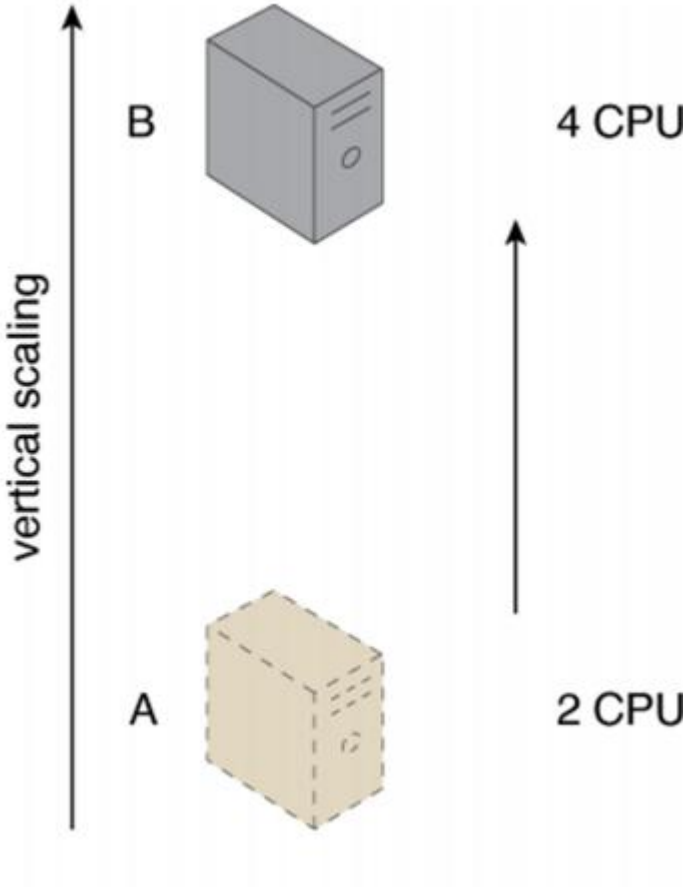
水平扩展（horizontal scaling）：改变 IT 资源的数量。

水平分配资源也称为向外扩展（scaling out）。

水平释放资源也成为向内扩展（scaling in）。




当一个现有 IT 资源被具有更大或更小容量的资源所替代，则称为垂直扩展（vertical scaling）。被具有更大容量的 IT 资源替代，称为向上扩展（scaling up），被具有更小容量的 IT 资源替代，称为向下扩展（scaling down）。




水平扩展和垂直扩展对比：

| 水平扩展 | 垂直扩展 |
| --- | --- |
| 更便宜（使用商品化的硬件组件） | 更昂贵（专用服务器） |
| IT 资源立即可用 | IT 资源通常为立即可用 |
| 资源复制和自动扩展 | 通常需要额外设置 |

| | |
|---|---|
| 需要额外 IT 资源 | 不需要额外 IT 资源 |
| 不受硬件容量限制 | 受限于硬件最大容量 |

（2）云服务

云服务（cloud service）是指任何可以通过云远程访问的 IT 资源。

云服务中的“服务”与其他 IT 领域中的服务技术（比如面向服务的架构，SOA）的“服务”含义更为宽泛。

并非云中所有的 IT 资源都可以被远程访问，其中有公开发布的 API 的软件程序可以专门部署为允许远程客户访问。

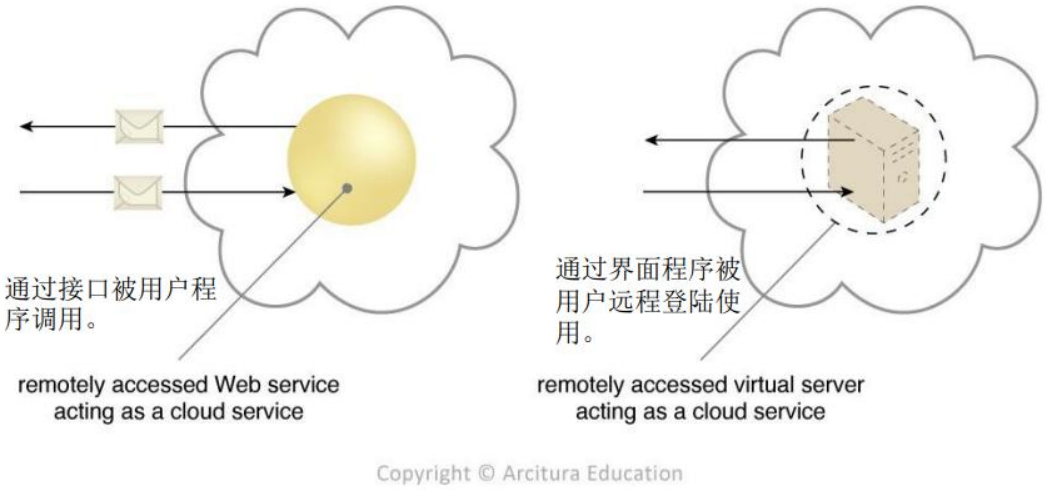



Figure 3.6 –具有已发布技术接口的云服务被云外用户访问（左）。作为云服务的虚拟服务器也可以被云边界之外访问（右）。

（3）云服务用户

云服务用户（cloud service consumer）是一个临时的运行时角色，由访问云服务的软件程序担任。

云服务用户常见类型：

A. 能够通过已发布的服务合同远程访问云服务的软件程序和服务。

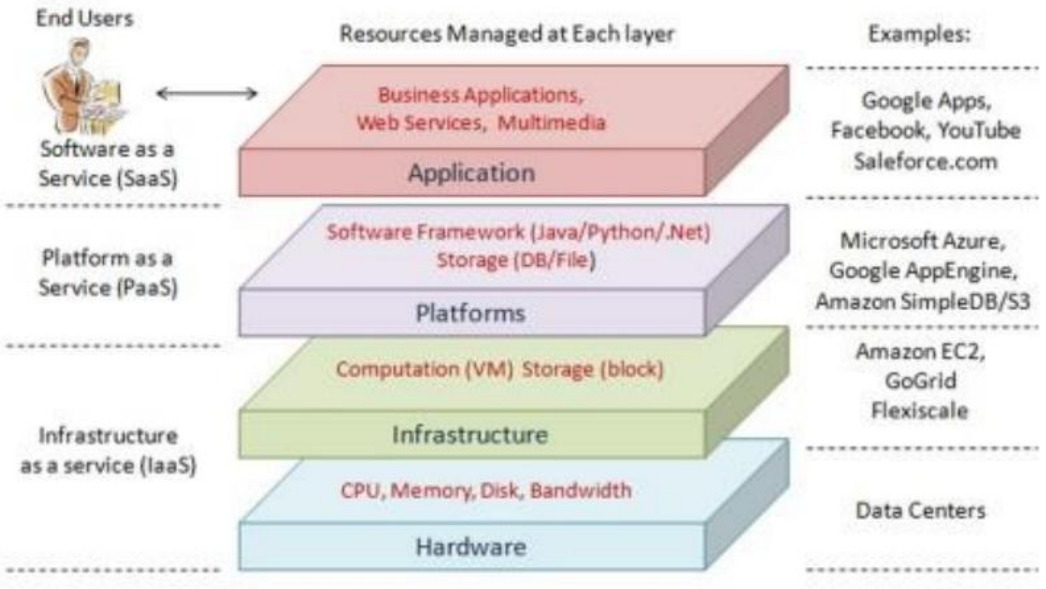
B. 运行某些软件的工作站、便携电脑和移动设备。

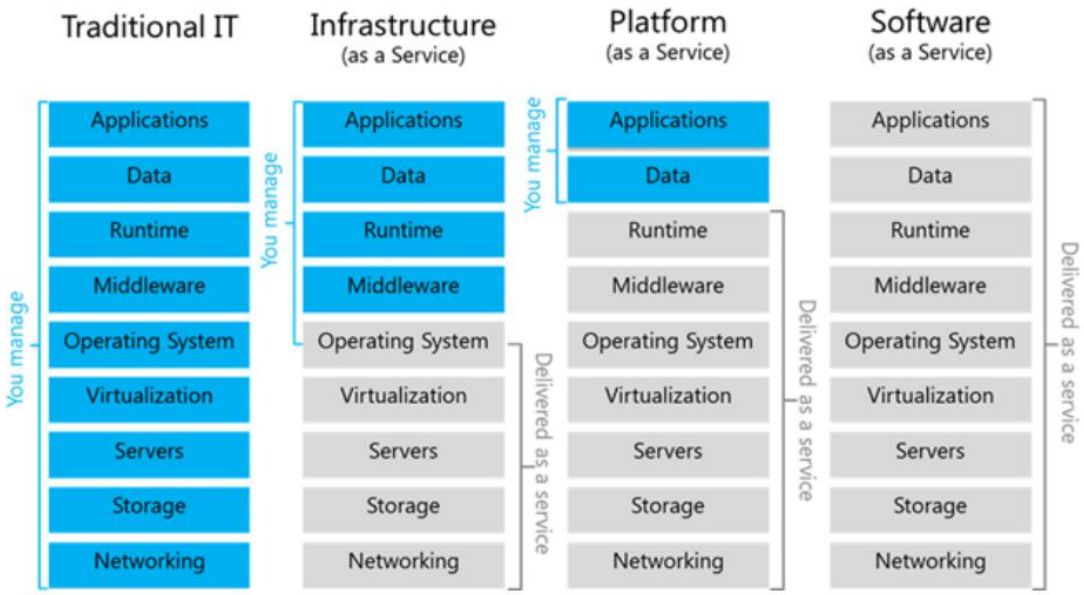

1.4 交付模式（Service Model）

Software as a Service，软件即服务，简称 SaaS，这层的作用是将应用作为服务提供给客户。例如，电子邮件服务。

Platform as a Service，平台即服务，简称 PaaS，这层的作用是将一个开发平台作为服务提供给用户。例如，Google App Engine, Microsoft Azure。

Infrastructure as a Service， 基础设施即服务，简称 IaaS，这层的作用是提供虚拟机或者其他资源作为服务提供给用户。例如，Amazon EC2（云服务器）。




1.5 经济合理性（Economic Justification）

（1）规模经济（规模效应，Economies of scale）

大型数据中心的运行（按单位度量）比小型数据中心便宜。

这里的“大型”指超过十万台服务器；“小型”指少于一万台服务器。

为何会有规模经济（规模经济的原因）：

A. 电力成本（Cost of power）

运行一个数据中心的电力成本占总运营成本的 10%-20%。

大型数据中心每台服务器的电力成本更低：

共享配件：例如机架（rack）和开关（switch）。

价格协商：更大的电力用户能谈出一个好折扣。

地理选择：大型数据中心能搬到电价最低的地方。

获得更廉价的能源：例如风电场和屋顶太阳能。

B. 硬件成本（Hardware costs）

相比更小的采购方，大型数据中心的运营商在购买硬件时可获得高达 30％的折扣。

C. 基础设施劳动力成本（Infrastructure labor costs）

更高效地利用系统管理员。

小型数据中心管理员服务约 150 台服务器。

大型数据中心管理员服务超过 1000 台服务器。

D. 安全性和可靠性（Security and reliability）

维持给定级别的安全性、冗余性和基本的灾难恢复需要固定的投资水平。

大型数据中心能用更多的服务器摊薄这些投资。

（2）设备的利用（Utilization of equipment）

如何提高数据中心的利用率？

（1）虚拟化允许将不同的应用程序置于同一位置。

（2）利用工作负载的变化来提高利用率。

A. 随机访问（Random access）

最终用户随机地访问应用程序。用户越多，负载越有可能变均匀。

B. 每天的访问时间（Time of day）

可以将工作场所相关的服务和消费者相关的服务放在一起。（人在工作时一般不消费，所以，在上班时间，消费者相关的服务负载较低）

不同地理位置之间的时差。（如果用户一般在白天才使用某个网站，这个网站可以同时对中美提供服务来削峰平谷）

C. 资源使用模式（Resource usage patterns）

将 CPU 负载较高的服务和 I/O 负载较高的服务放在一起。

D. 不确定性（Uncertainty）

考虑用量高峰：

新闻事件、营销事件、运动时间

利用公有云来维持足够的容量以支持使用高峰

关键技术是分析负载变化的模式，并利用该模式在资源上分配负载。

## 1.6 基本机制

### 1.6.1 虚拟机（Virtual Machine）

一台虚拟机具有与任何其他虚拟机隔离的地址空间。

从应用程序角度看，它看起来像一台裸机（bare metal machine）。

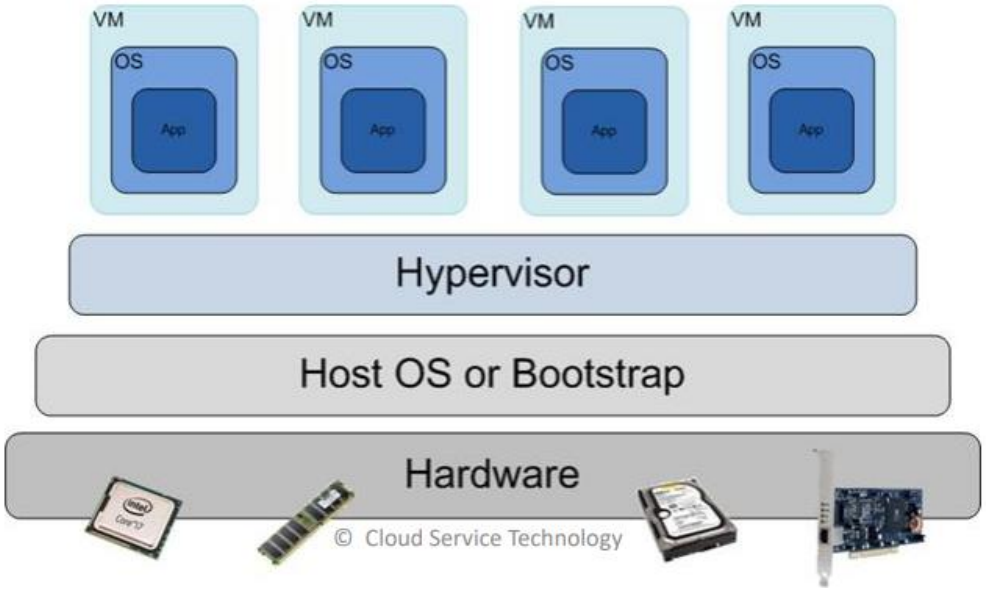
分配了 IP 地址并具有网络功能。

可以加载可以在主机处理器上执行的任何操作系统或应用程序。

### 1.6.2 监控器（Hypervisor）

监控器是用于创建和管理虚拟机的操作系统。

例如，VMWare、Xen、KVM。



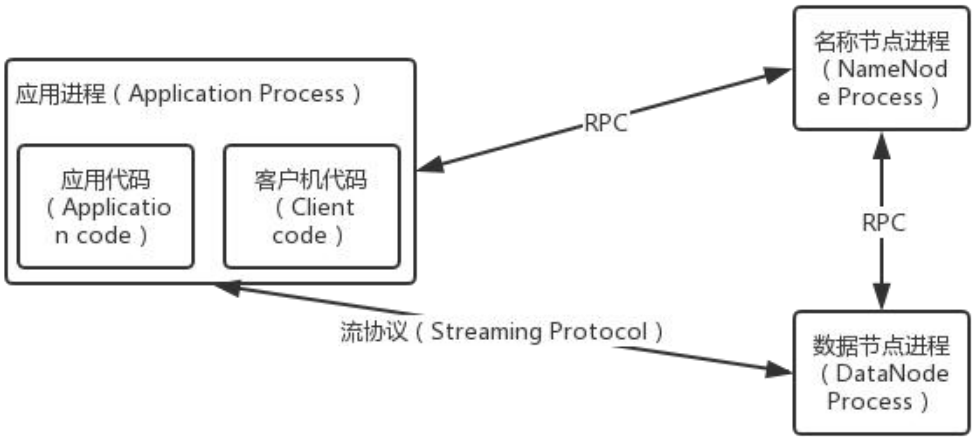

### 1.6.3 文件系统（File System）

每个虚拟机都可以访问文件系统。

HDFS（Hadoop 分布式文件系统）——一种广泛使用的开源云文件系统。

我们说明 HDFS 如何使用冗余（redundancy）来保证可获得性的。

HDFS 的组成：




写入过程参考并行与分布式计算中的 HDFS/GFS。

## 1.7 云架构（Cloud Architecture）

### 1.7.1 安全（Security）

相比非云环境，多租户带来了更多担忧。

疏忽导致的信息共享：可能由于共享资源使用而共享信息。例如，若重新分配磁盘，则磁盘上的信息可能会保留。

虚拟机逃逸：是指脱离虚拟机（监控器）并与主机操作系统进行交互的过程。

拒绝服务（Denial of Service, DoS）攻击：一个用户可以消耗主机服务器的资源，而拒绝其他用户使用。

### 1.7.2 性能（Performance）

两种保证性能的方法：

负载均衡（load balancing）是在多个计算资源之间分配工作负载，以避免单个资源的过载。

自动扩展（auto-scaling）是一种方法，通过该方法通常可以根据活动服务器的数量来衡量计算资源的数量， 根据负载自动伸缩。

### 1.7.3 可得性（Availability）

故障（failure）是云中的常见现象。

当服务器的规模达到数千台时，故障是可以预料到的（一定会发生）。

云提供商确保云本身将保持可用，但有一些明显的例外。

应用程序开发人员必须假设实例将出故障，并在故障的情况下建立检测和更正机制。

## 2. 数据中心网络

### 2.1 设计目标与需求

#### A. 数据中心的整体设计目标

敏捷性——任何服务，任何服务器

将服务器转变为一个大的可取代（fungible）池：

让服务“呼吸”：根据需要动态扩展和收缩他们的足迹

我们已经看到这是如何根据 Google 的 GFS、BigTable、MapReduce 完成的

具有非阻塞核心的等距终端

无限的工作负载移动性

好处：

提高服务开发人员的工作效率

降低成本

实现高性能和可靠性

这些是大多数数据中心基础架构项目的三大动机！

#### B. 数据中心的应用需求

数据中心通常运行两种类型的应用程序：

面向外部（例如，向用户提供网页）

内部计算（例如，用于 Web 索引的 MapReduce）

工作负载通常是不可预测的：

多个服务在数据中心内并发运行。

对新服务的需求可能出现意外情况：

对新服务的需求飙升意味着成功！

但成功也伴随着麻烦（如果没有准备好的话）！

服务器故障是常态。

C. 数据中心的网络需求

统一的高容量：

服务器之间的容量仅受其网络接口控制器（NIC）限制

添加服务器时无需考虑拓扑

换句话说，任何两台服务器之间的容量都很高，无论它们位于哪个机架上！

性能隔离：

一个服务的流量不受其他服务的影响

易于管理：“即插即用”

平面寻址，因此任何服务器都可以拥有任意的 IP 地址

服务器配置与局域网中的配置相同

依赖于广播的传统应用程序必须能工作

可扩展、易于管理、容错且高效的数据中心网络（Data Center Networks, DCN）的要求：

R1：任何 VM 都可以在不更改其 IP 地址的情况下迁移到任何物理机器

R2：管理员不需要先配置任何交换机部署

R3：任何终端主机都应通过任何可用路径与任何其他终端主机进行高效通信

R4：无转发环路

R5：故障检测应快速有效

对网络协议的影响：

整个数据中心由一个二层网络组成（R1 和 R2）【软件定义网络中的大二层网络】

具有数十万个条目的 MAC 转发表（R3）

高效的路由协议，可快速将拓扑变化传播到所有点（R5）

2.2 成本

总成本多变：

大型数据中心超过 2.5 亿美元

服务器成本占主导地位

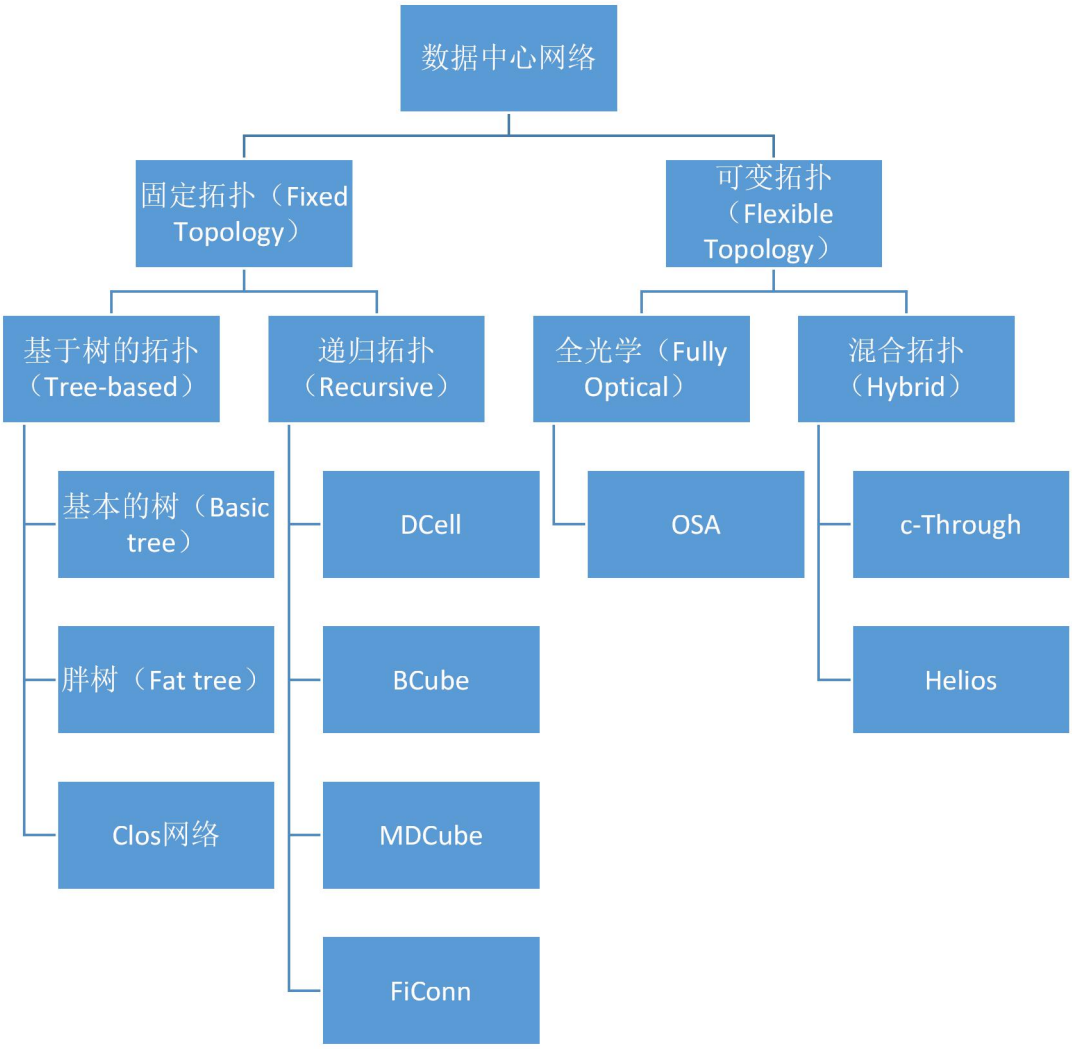
网络成本占比较大

供应时间长：

最多只能做到每季度进行一次新服务器的采购

| 分摊后成本（amortized cost） | 组件（component） | 子组件（sub-component） |
|---|---|---|
| ~45% | 服务器 | CPU、内存、磁盘 |
| ~25% | 电力基础设施 | UPS、散热、电力分配 |
| ~15% | 电力驱动（power draw） | 电力使用成本 |
| ~15% | 网络 | 交换机、线路、网络接入（transit） |

2.3 数据中心拓扑结构（分类）




2.4 常见拓扑结构

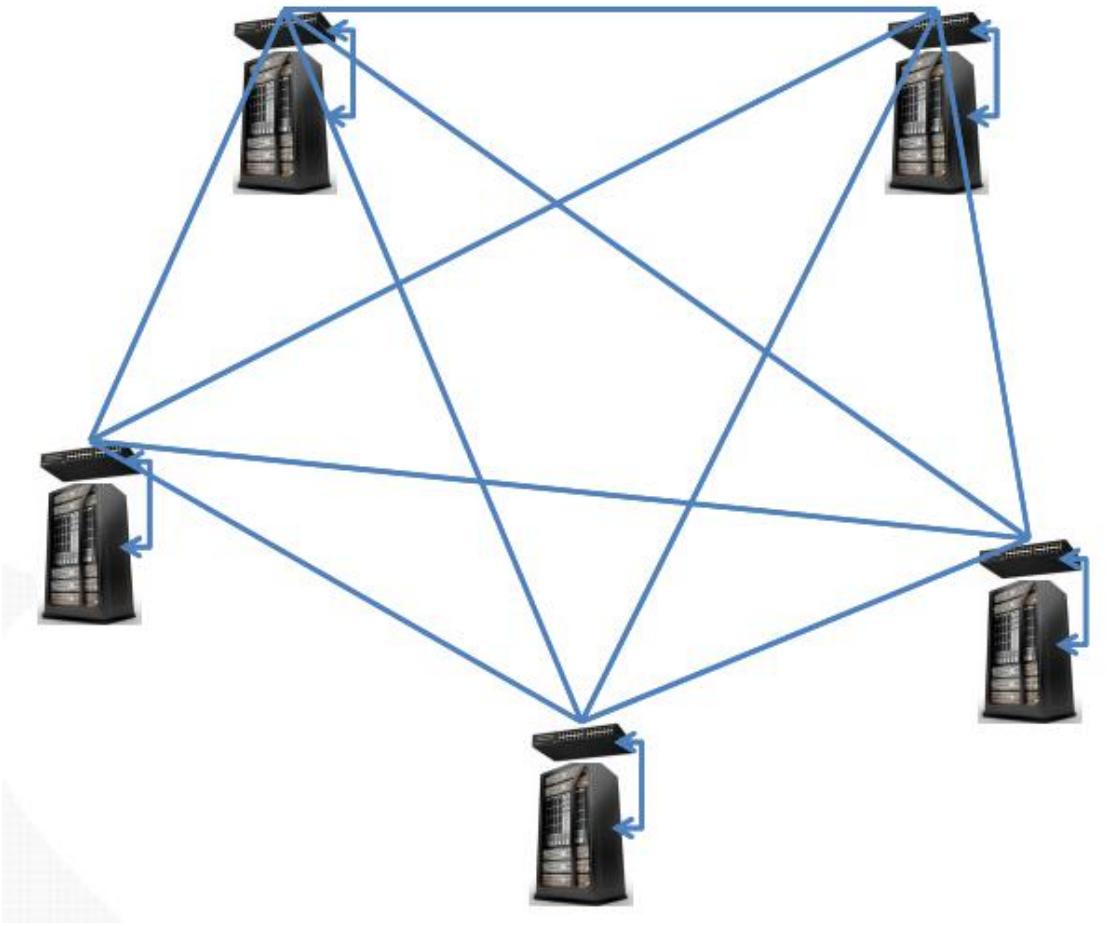
现在典型的数据中心拓扑：

终端主机连接到架顶式（top of rack, ToR）交换机上

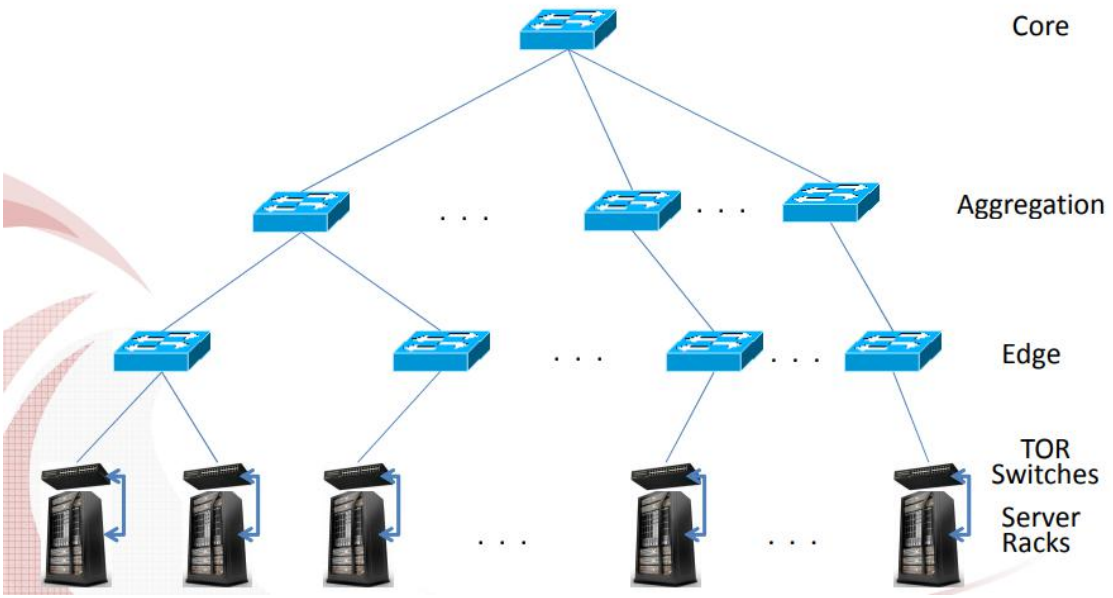
ToR 交换机有 48 个千兆以太网（GigE）端口和高达 410Gbps 的以太网上行链路

ToR 交换机连接到一个或多个列末（end of row, EoR）交换机

全 Mesh 网络：



基本树状拓扑：




2.5 胖树拓扑（Fat-Tree Topology）

A. 基于树的拓扑（Tree-based Topology）

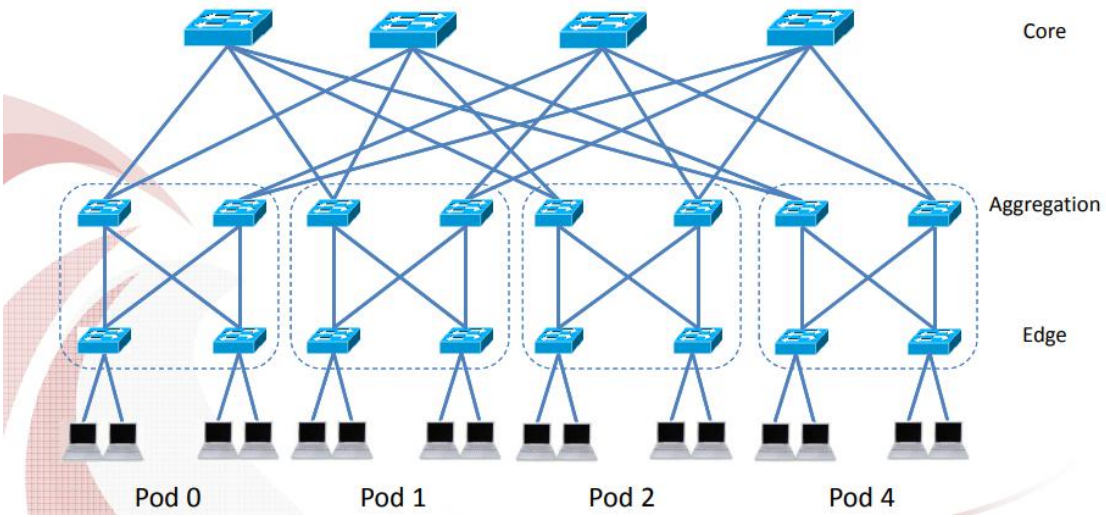
每台电脑一个端口

通过增加更多的交换机和在交换机上增加更多的端口来扩展

基本树、胖树（Fat Tree）、Clos 网络……

也可以归类到以交换机为中心的拓扑

B. 胖树架构及属性（重点）




胖树：一种特殊的 Clos 网络

K 元胖树：三层拓扑（接入层【edge】、汇聚层【aggregation】和核心层【core】）

把胖树分为 K 个 pod

每个 pod 由 $(\frac{K}{2})^2$ 个服务器和 2 层 $\frac{K}{2}$ 个 K 端口的交换机组成

每个接入层交换机连接到 $\frac{K}{2}$ 个服务器和 $\frac{K}{2}$ 个汇聚层交换机上

每个汇聚层交换机连接到 $\frac{K}{2}$ 个接入层交换机和 $\frac{K}{2}$ 个核心层交换机上

$(\frac{K}{2})^2$ 个核心层交换机：每个连接到 K 个 pod 上

每个 pod 支持在 $(\frac{K}{2})^2$ 台主机上的非阻塞操作【在中间层交换机上，上行链路和下行链路的比例为 1：1，这样的网络被称之为非阻塞网络（non-blocking）】

每个源节点和目的节点有 $(\frac{K}{2})^2$ 条路径

胖树的属性：

任意两端都有相同的带宽

每层具有相同的聚合带宽

可以使用具有统一容量的廉价设备构建：

每个端口支持与终端主机相同的速度

如果数据包被沿可用路径统一分发，所有设备都能以链路速度传输【可以将任意的节点分成两对，如果所有节点对都开始通信，那么每对节点仍然能够以全链路带宽进行通信】

可扩展性极佳：K 端口的交换机支持 $\frac{K^2}{4}$ 台服务器

参考：

Fat-Tree Topo Architecture（胖树拓扑结构）

https://blog.csdn.net/u012925450/article/details/108493968

胖树网 https://baike.baidu.com/item/%E8%83%96%E6%A0%91%E7%BD%91

杨旭. 基于 SDN 的胖树网络路由策略研究[D]. 湖北工业大学, 2019.

## 3. 虚拟化技术

### 3.1 计算虚拟化

#### 3.1.1 虚拟机原理

计算（服务器）虚拟化：将一个或多个物理服务器虚拟成多个逻辑上的服务器

服务器虚拟化的层次：

寄居虚拟化：

主机操作系统之上构建虚拟化层

虚拟化层称为虚拟机监控器（VMM）

虚拟化架构系统损耗比较大


寄居虚拟化架构

裸机虚拟化：

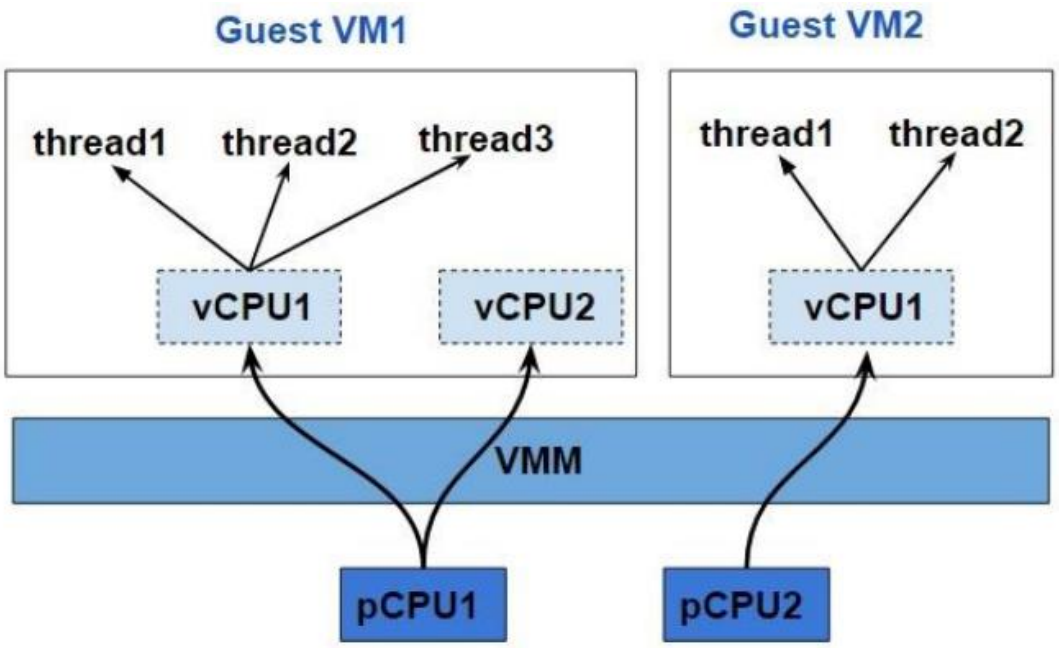
架构中的 VMM 也可以认为是一个操作系统，一般称为 Hypervisor

Hypervisor 实现从虚拟资源到物理资源的映射

Hypervisor 实现了不同虚拟机的运行上下文保护与切换，保证了各个客户虚拟系统的有效隔离


图7-2　裸机虚拟化架构

CPU 虚拟化：

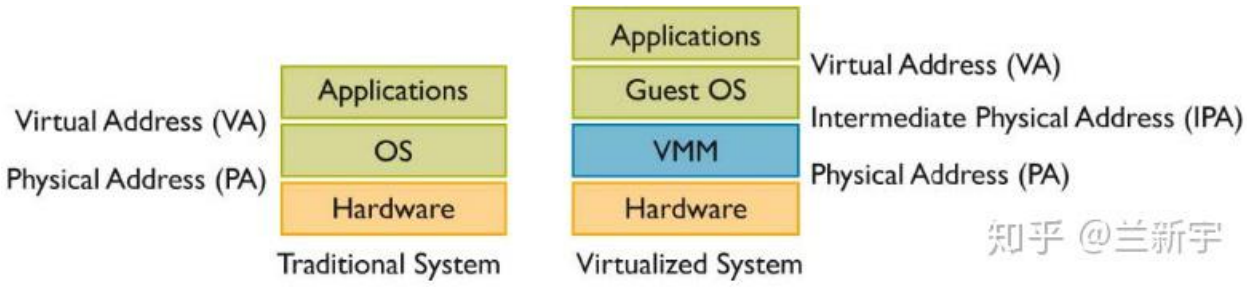


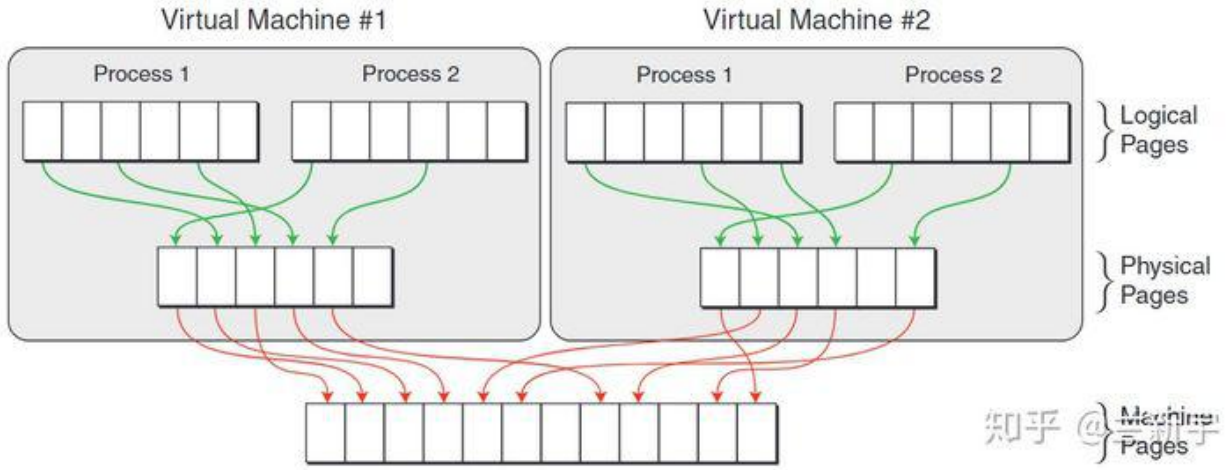

硬件上提供对 vCPU 调度和切换的支持：Intel VT-x（Virtualization Technology for x86）、AMD-V

VMM 调度：VMM 决定当前哪个虚拟 CPU 在物理 CPU 上运行，要保证隔离性、公平性和性能

Guest OS 调度？

内存虚拟化：




内存虚拟化技术把物理内存统一管理，包装成多个虚拟的物理内存提供给若干虚拟机使用，每个虚拟机拥有各自独立的内存空间。

I/O 虚拟化：

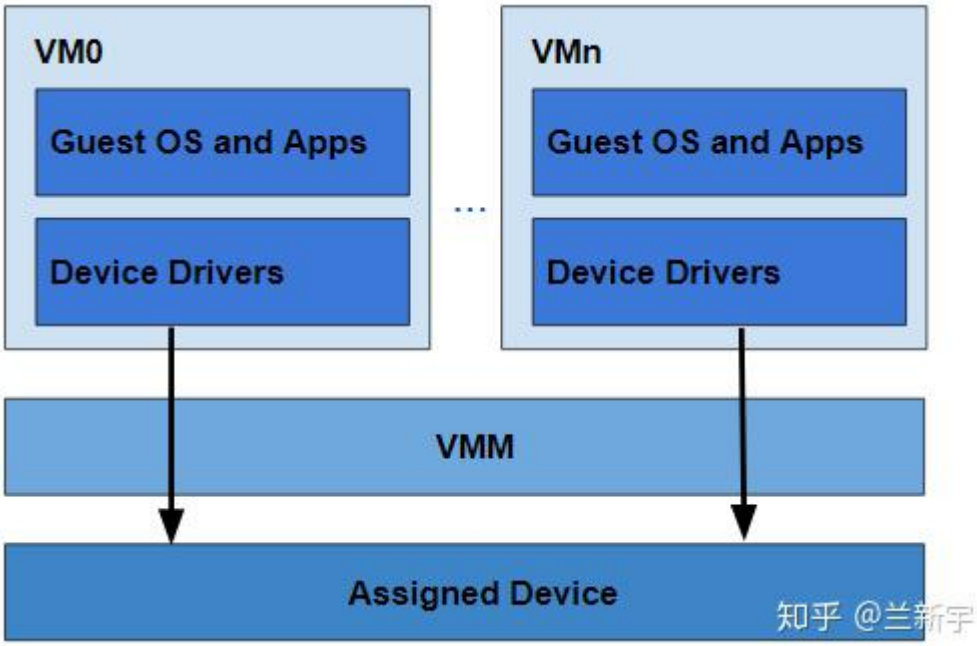
I/O 设备虚拟化技术把真实的设备统一管理起来，包装成多个虚拟设备给若干个虚拟机使用，响应每个虚拟机的设备访问请求和 I/O 请求。

两种实现方法：透传（passthrough），模拟（emulation）。

透传：

guest VM 可以透过 VMM，直接访问 I/O 硬件；guest VM 的 I/O 操作路径几乎和无虚拟化环境下的 I/O 路径相同。

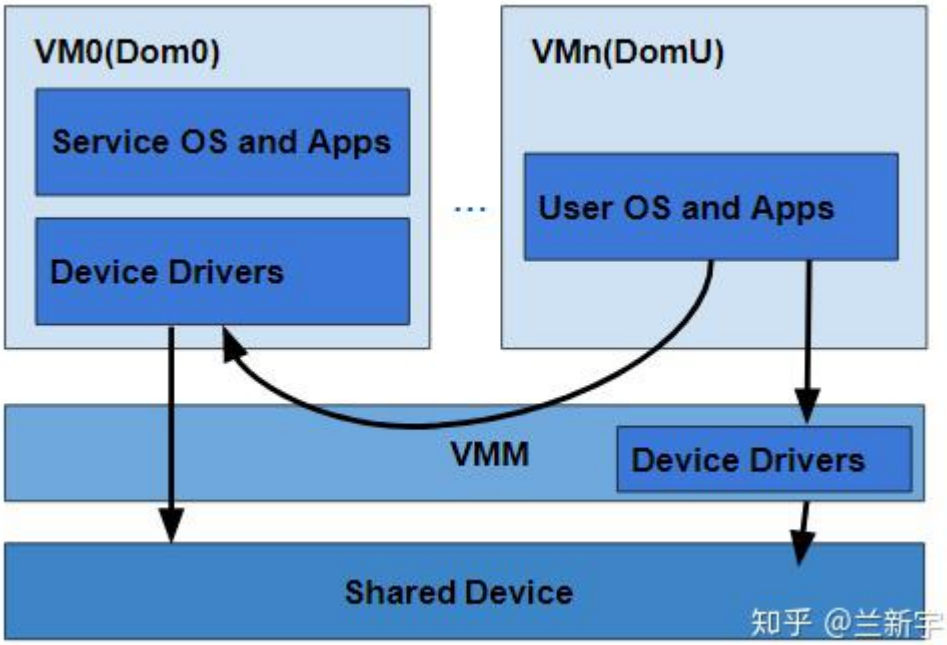
性能高；I/O 设备无法在多台 guest VM 之间共享和动态迁移。

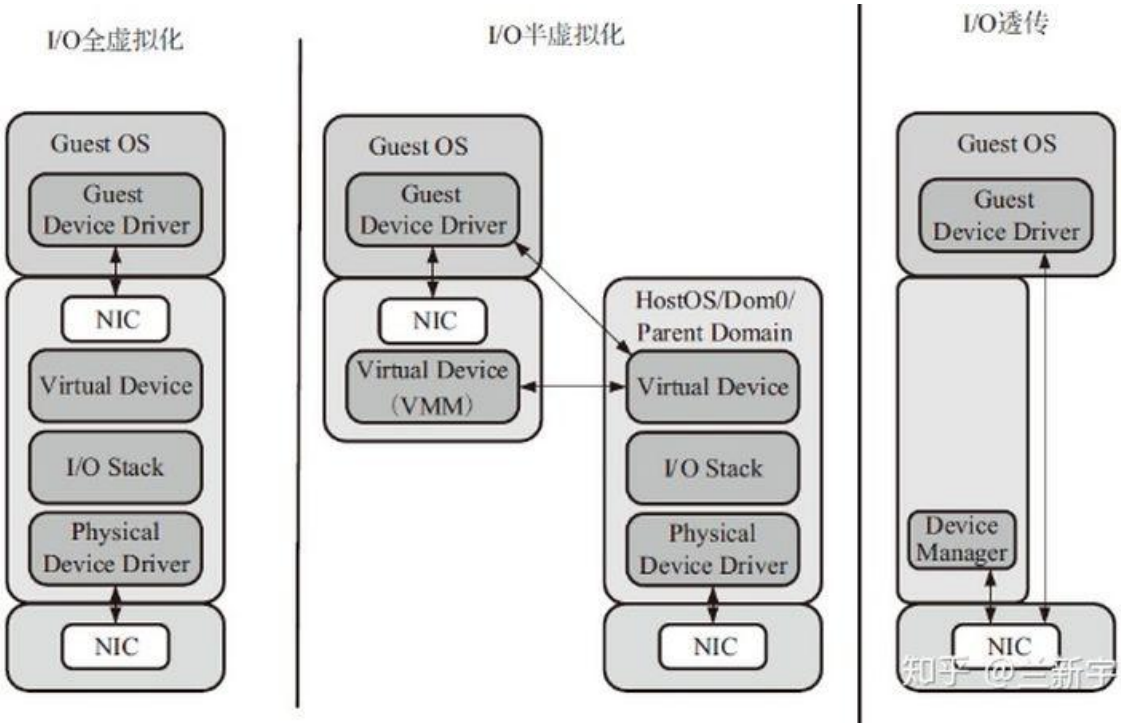



模拟：

在 hypervisor 模型中，VMM 直接就可以提供外设驱动，因此实现对 guest VM 所访问设备的模拟很方便的。

VMM 中没有相应的驱动，那么它就会把这个请求转发给一个拥有该设备驱动程序的 guest VM（Dom0）






参考：

虚拟化技术 - 内存虚拟化 [一] https://zhuanlan.zhihu.com/p/69828213

虚拟化技术 - I/O 虚拟化 [一] https://zhuanlan.zhihu.com/p/69627614

虚拟化技术 - I/O 虚拟化 [二] https://zhuanlan.zhihu.com/p/75649223

### 3.1.2 虚拟机在线迁移机制

（1）预迁移（Pre-Migration）：主机 A 打算迁移其上的一个虚拟机 VM，首先选择一个目的计算机作为 VM 的新主机。

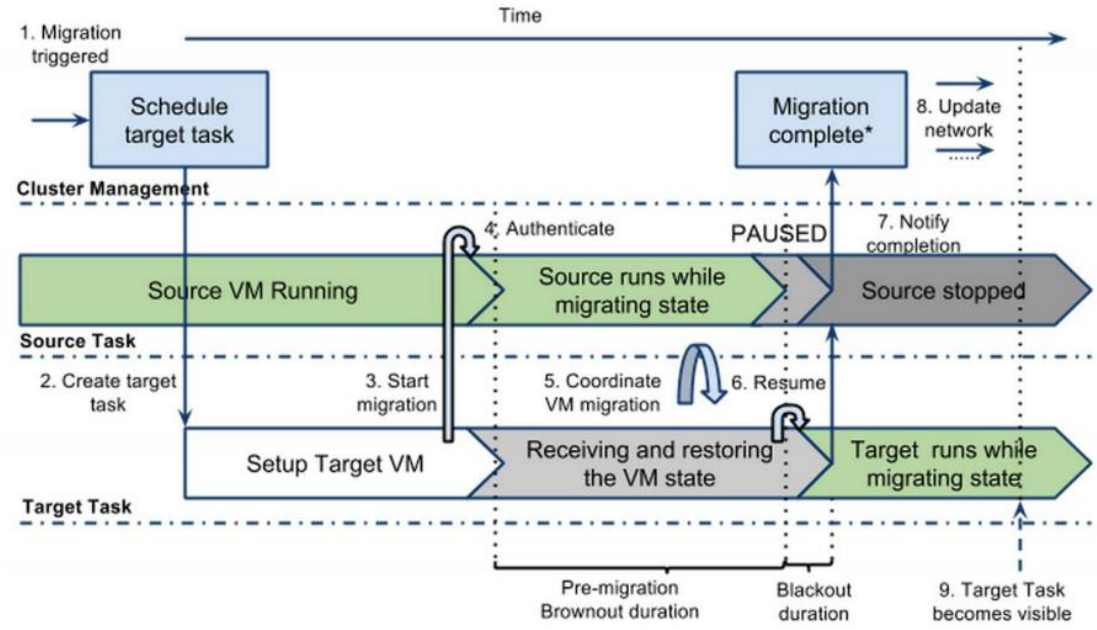
（2）预定资源（Reservation）：主机 A 向主机 B 发起迁移请求，先确认 B 是否有必需的资源，若有，则预定这些资源；若没有，VM 仍在主机 A 中运行，可以继续选择其他计算机作为目的计算机。

（3）预复制（Interative Pre-Copy）：在这一阶段 VM 仍然运行，主机 A 以迭代方式将 VM 的内存页复制到主机 B 上。在第一轮迭代中，所有的页都要从 A 传送到 B，以后的迭代只复制前一轮传送过程中被修改的页面。

（4）停机复制（Stop-and-Copy）：停止主机 A 上的 VM，把它的网络连接重定向到 B。CPU 状态和前一轮传送过程中修改过的页都在这个步骤被传送。最后，主机 A 和主机 B 上有一致的 VM 映像。

（5）提交（Commitment）：主机 B 通知 A 已经成功收到了 VM 的映像，主机 A 对这个消息进行确认，然后主机 A 可以抛弃或销毁其上的 VM。

（6）启动（Activation）：启动迁移到 B 上的 VM，迁移后使用目的计算机的设备驱动，广播新的 IP 地址。




3.1.3 性能指标及函数

迁移时间 vs.停机时间

预复制实时迁移

跳过技术

**Algorithm 1** Modeling the migration process

1: **Input:** memory size $M$, page transfer rate $R$
2: **Output:** pre-copy time $t_p$, downtime $t_d$
3: **function** LIVEMIGRATION(M, R)
4: $i \leftarrow 1$ ▷ $i$ denotes iteration#
5: $V_i \leftarrow M$ ▷ $V_i$ denotes #pages to be transferred
6: **repeat** ▷ starting iterative pre-copy phase
7: $T_i \leftarrow \frac{V_i - W_i}{R}$ ▷ #skipped pages $W_i$ is 0 for KVM
8: $t_p \leftarrow t_p + T_i$ ▷ $T_i$ denotes iteration time
9: $V_{i+1} \leftarrow D_i$ ▷ $D_i$ denotes #dirtied pages in $T_i$
10: $i \leftarrow i + 1$ ▷ moving to next iteration
11: **until** stop-and-copy condition [28] is met
12: $t_d \leftarrow \frac{V_i}{R}$ ▷ transferring remaining pages
13: **return** $t_p, t_d$ ▷ pre-copy time and downtime
14: **end function**

服务质量（Quality of Service, QoS）：在时刻$i$，迁移任务$j$的服务质量$Q_{i,j} = f(d_{i,j})$，其中$f_i$为任务$i$的截止时间，$d_{i,j}$为时刻$j$下迁移任务$i$的服务率（rate of service）。

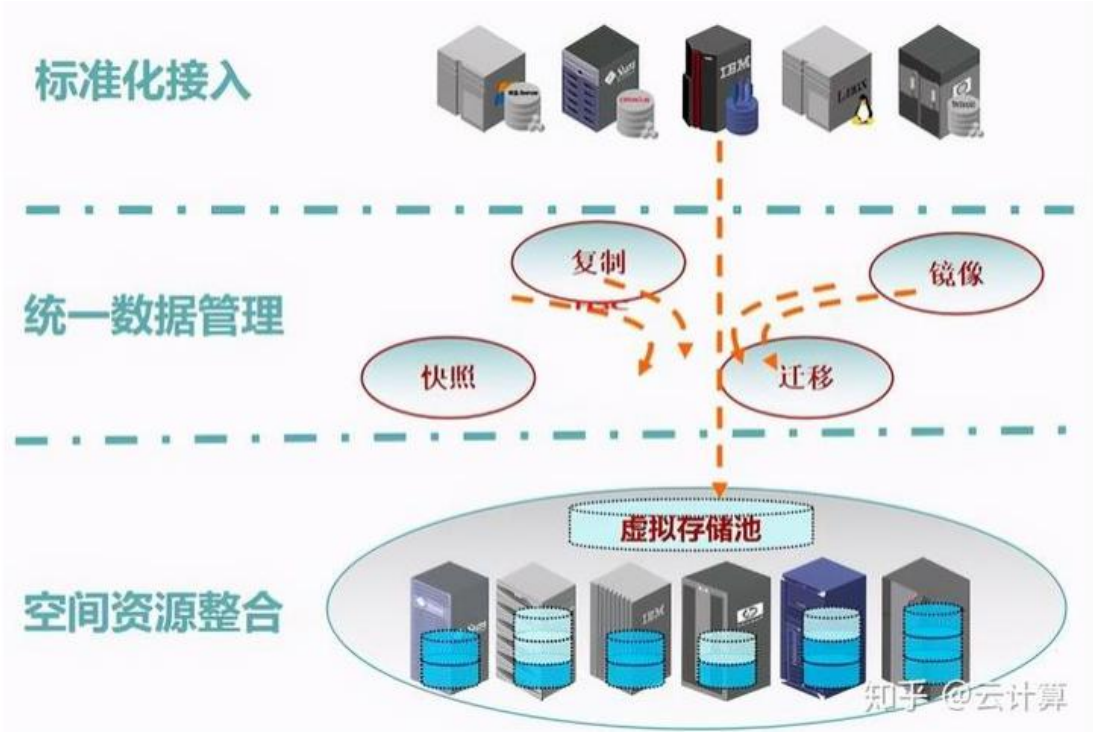
迁移时间（Migration Time）：$t_{Migration}=\frac{M}{b-r_{Dirty}}$，其中，$M$ 为内存大小（Memory Size），$b$ 为带宽，$r_{Dirty}$ 为变脏率（Dirty Rate）。【？】

3.2 存储虚拟化

存储系统成为数据中心的核心平台

存储容量、数据访问性能、数据传输性能、数据管理能力、存储扩展能力

存储虚拟化是指将存储网络中各个分散且异构的存储设备按照一定的策略映射成一个统一的连续编址的逻辑存储空间，称为虚拟存储池，并将虚拟存储池的访问接口提供给应用系统。




存储虚拟化优势：

存储虚拟化将系统中分散的存储资源整合起来

在虚拟层通过使用数据镜像、数据校验和多路径等技术提高了数据的可靠性及系统的可用性

利用负载均衡、数据迁移、数据块重组等技术提升系统的潜在性能

整合和重组底层物理资源

存储技术分类：

RAID（Redundant Array of Independent Disk）技术

磁盘阵列

NAS（Network Attached Storage）网络连接存储技术

专用文件服务器

通过标准网络协议加入网络

SAN（Storage Area Network）存储区域网络技术

专门为存储建立的独立于 TCP/IP 网络之外的专用网络

磁盘阵列连接高速通信网络

FC-SAN 为通过光纤通道协议转发 SCSI 协议，IP-SAN 通过 TCP 协议转发 SCSI 协议

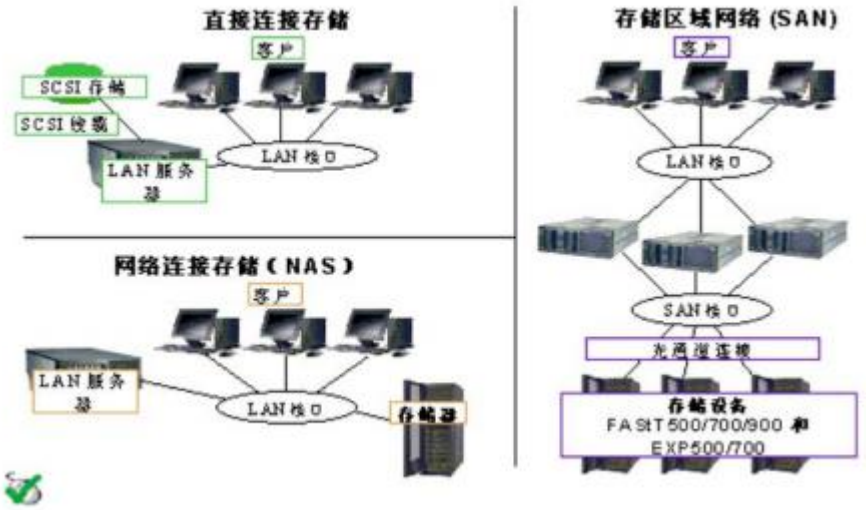
SAN 与 NAS 区别

SAN 存储设备通过光纤连接，而 NAS 存储设备通过 TCP/IP 连接

SAN 通常用于高级解决方案；NAS 解决方案更易于家庭用户或小型企业使用

SAN 存储设备访问数据块，而 NAS 存储设备访问单个文件

SAN 存储设备连接多个存储设备，而 NAS 存储设备作为单个专用设备运行




存储虚拟化实现方式：

（1）基于主机的存储虚拟化

该技术又称为逻辑卷管理，通常由主机操作系统下的逻辑卷管理软件实现

逻辑卷管理软件把多个不同的磁盘阵列映射成一个虚拟的逻辑块空间

当存储需求增加时，逻辑管理软件能把部分逻辑空间映射到新增的磁盘阵列，因此可以在不中断运行的情况下增加或减少物理存储设备

虚拟机主要功能是在系统和应用级上完成多台主机之间的数据存储共享、存储资源管理、数据复制及迁移、集群系统、远程备份、灾难恢复等存储管理任务

当仅需要单个主机服务器访问多个磁盘阵列时，可以使用基于主机的存储虚拟化技术

优点：

支持异构的存储系统

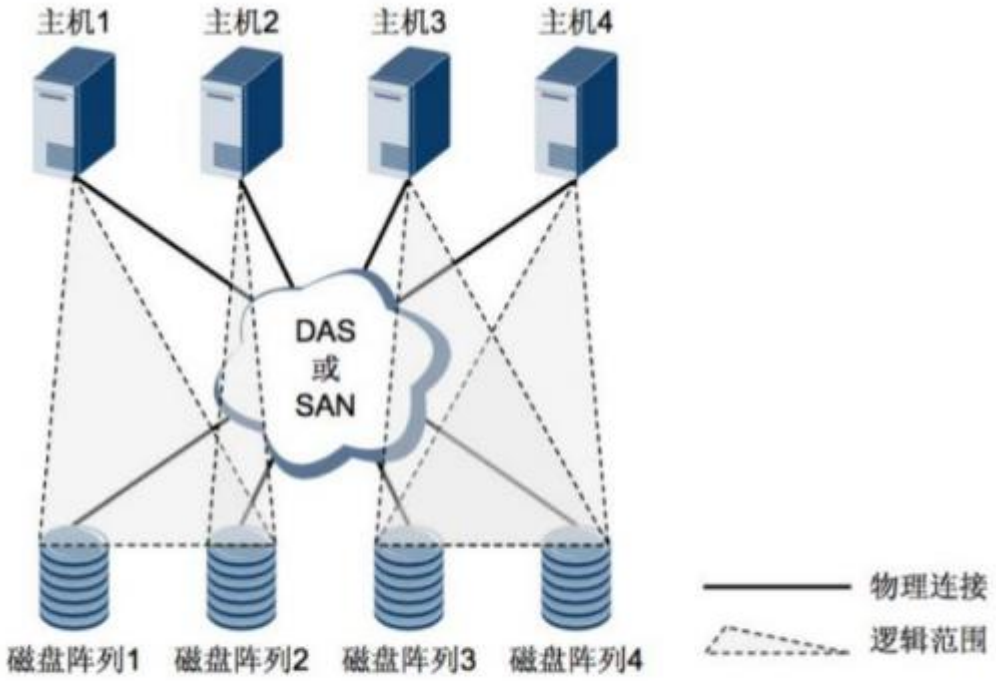
开销低，不需要硬件支持，不修改现有系统架构

缺点：

占用主机资源，降低应用性能

存在操作系统和应用的兼容性问题

主机升级、维护、扩展、迁移复杂




基于主机的存储虚拟化

（2）基于存储设备的存储虚拟化

该技术通过在存储设备上添加虚拟化功能实现，可以将一个阵列上的存储容量划分为多个存储空间（LUN），供不同的主机系统访问

当有多个主机服务器需要访问同一个磁盘阵列时，可以使用基于存储设备的存储虚拟化技术

优点：

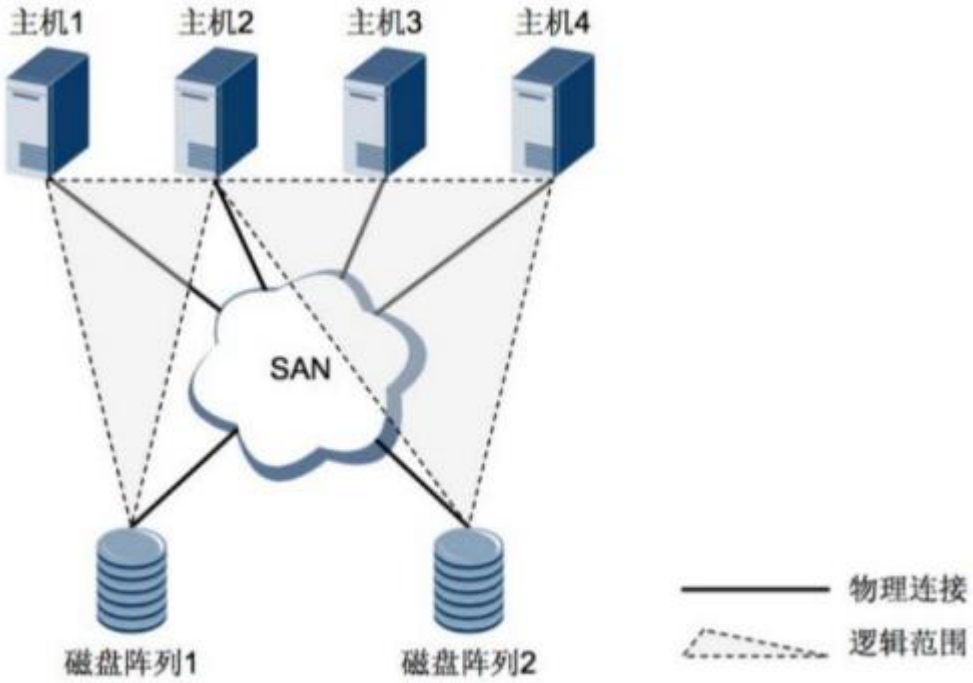
与主机无关，不占用主机资源

数据管理功能丰富

缺点：

一般只能实现对设备内磁盘的虚拟化

不同厂商间的数据管理功能不能互操作




基于存储设备的存储虚拟化

（3）基于网络的存储虚拟化

在网络设备上实现存储虚拟化，如通过在 SAN 中添加虚拟化引擎实现

根据实现位置不同

基于交换机的虚拟化

基于路由器的虚拟化

基于元数据服务器的虚拟化

当多个主机服务器需要访问多个异构存储设备时，可以使用基于网络的存储虚拟化技术

优点：

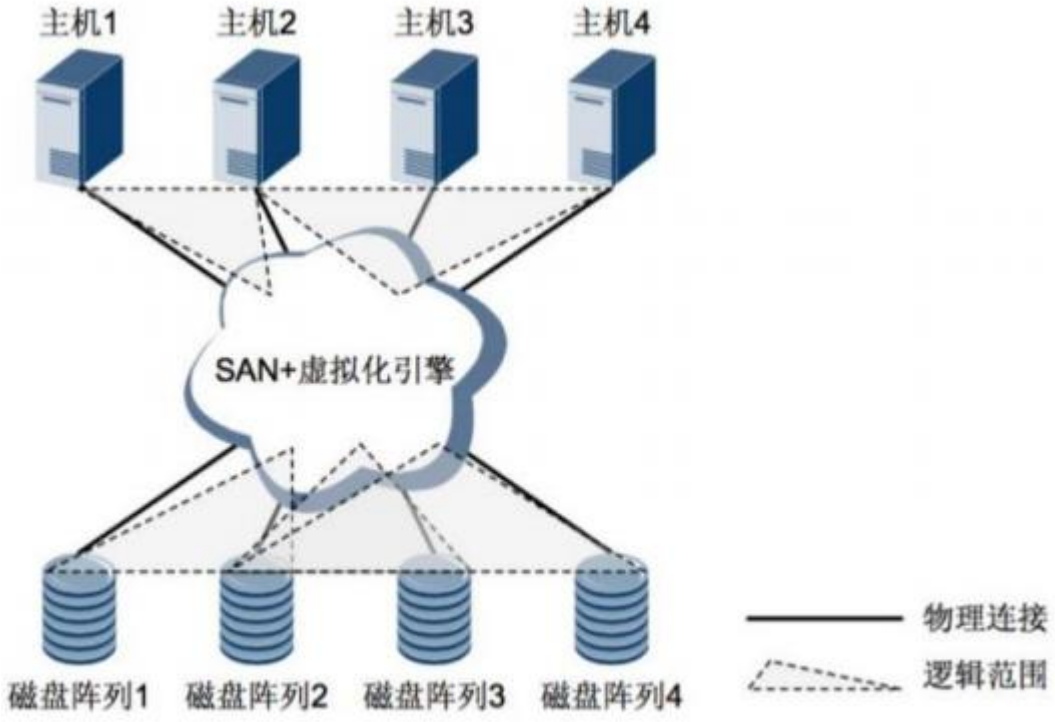
与主机无关，不占用主机资源

支持异构存储设备

统一不同存储设备的数据管理功能，可扩展性好

缺点：

部分厂商数据管理功能弱，成熟度低，仍然存在和不同存储、主机兼容的问题




基于网络的存储虚拟化

技术比较

| 比较项 | 基于主机 | 基于存储设备 | 基于网络 |
|---|---|---|---|
| 对主机的影响 | 大 | 小 | 小 |
| 主机兼容性 | 较差 | 好 | 好 |
| 存储兼容性 | 好 | 较差 | 好 |
| 业务功能 | 较差 | 较好 | 较好 |
| 对性能的影响 | 较大 | 较小 | 较小 |
| 可扩展性 | 较差 | 较好 | 好 |
| 实施影响 | 大 | 较小 | 较小 |

3.3 网络虚拟化

为什么需要网络虚拟化？

传统数据中心：

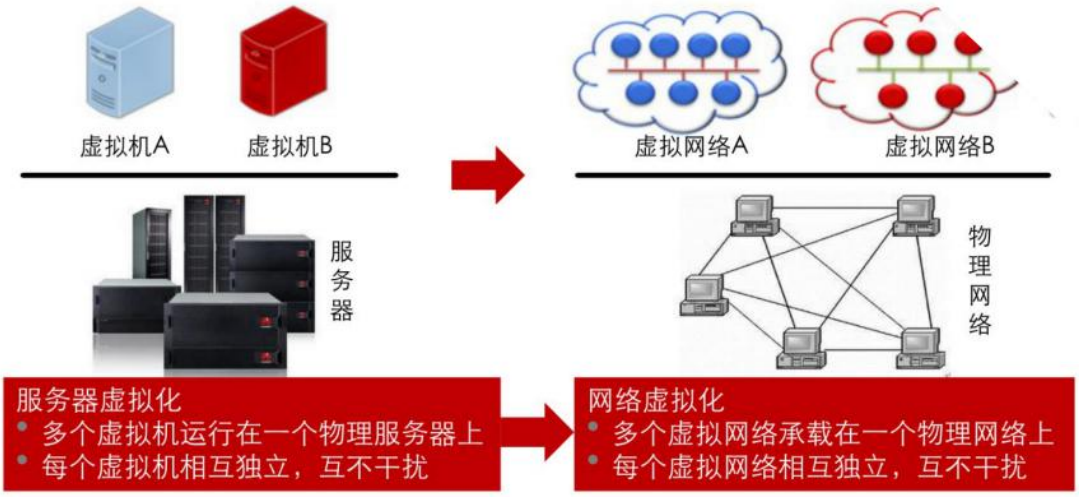
计算机与网络关系固定，很少变动

云数据中心：

虚拟机会频繁跨主机迁移，与网络关系不固定

不再是南北流量为主；东西流量占据 76%

网络虚拟化的产生

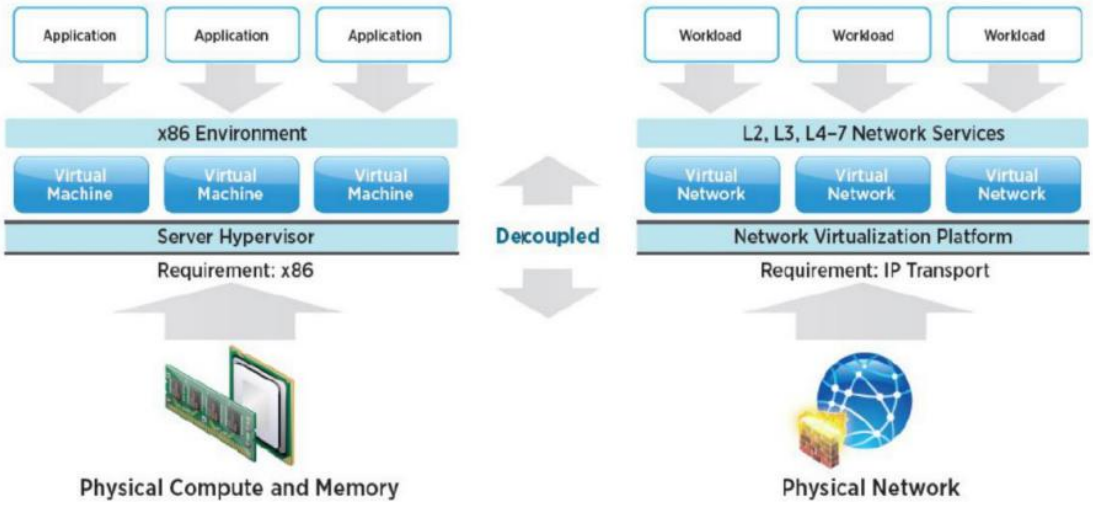



虚拟化技术从服务器虚拟化延伸到网络虚拟化

通过网络虚拟化，基于同一物理网络，虚拟机认为运行于不同的虚拟网络

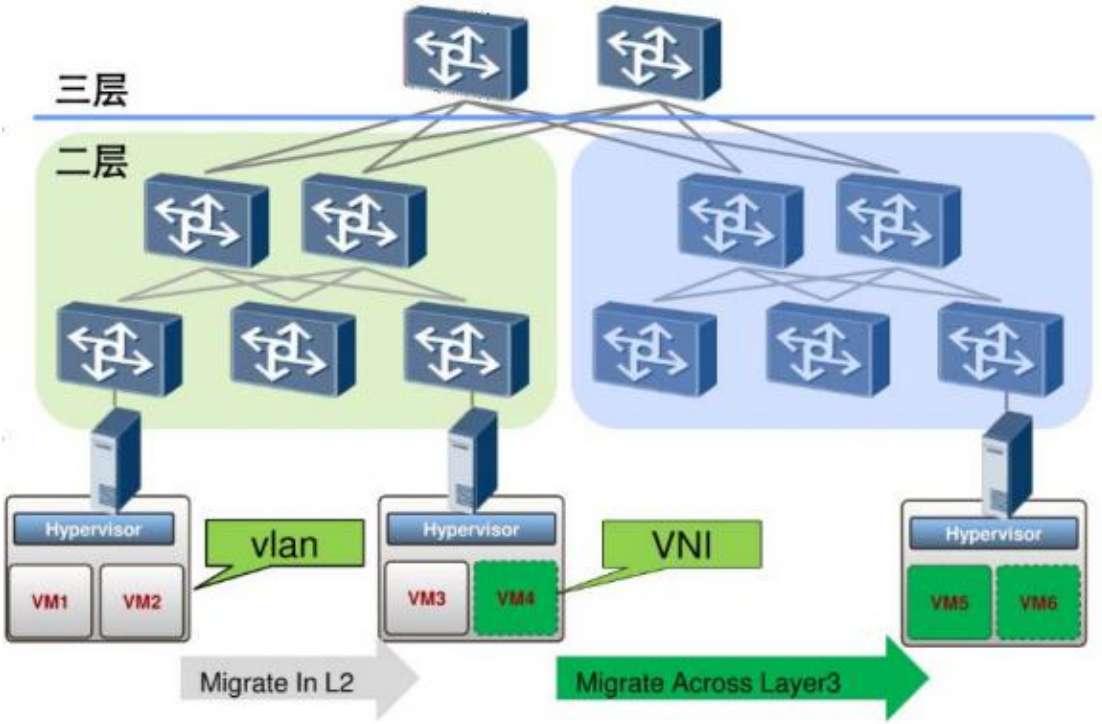
服务器虚拟化使得虚拟机按需可得，网络虚拟化使得虚拟网络灵活配置

网络虚拟化




与服务器虚拟化类似，网络虚拟化可以在很短的时间（秒级）创建 L2、L3 到 L7 的网络服务，如交换、路由、防火墙和负载均衡等。虚拟网络独立于底层网络硬件，可以按照业务需求配置、修改、保存、删除，而无需重新配置底层物理硬件或拓扑。这种网络技术的革新为实现软件定义的数据中心奠定了基础。

云计算网络虚拟化的特性




隔离性

不同租户的流量，相互之间不能访问

不同租户的 IP/MAC 地址可以独立规划，甚至可以相互重叠

可移动性

虚拟机可以跨二层/三层迁移，甚至可以跨广域网进行迁移

逻辑网络和物理网络解耦，不受物理网络限制

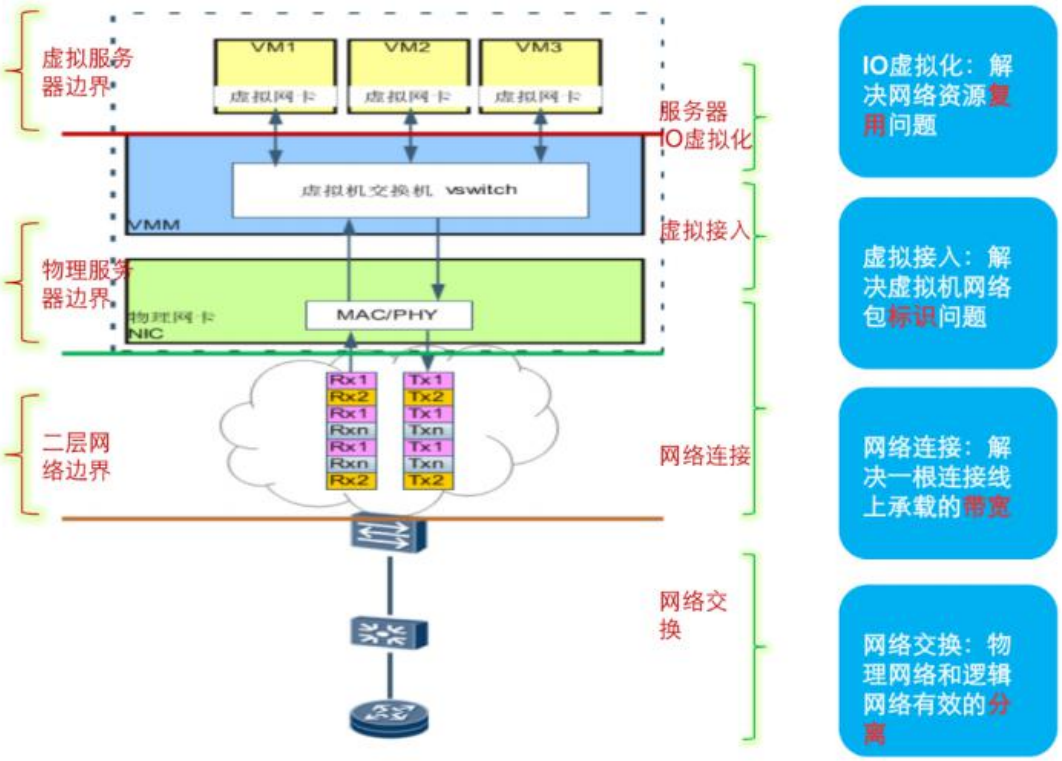
逻辑网络可以跨越二层/三层物理网络

可扩展性

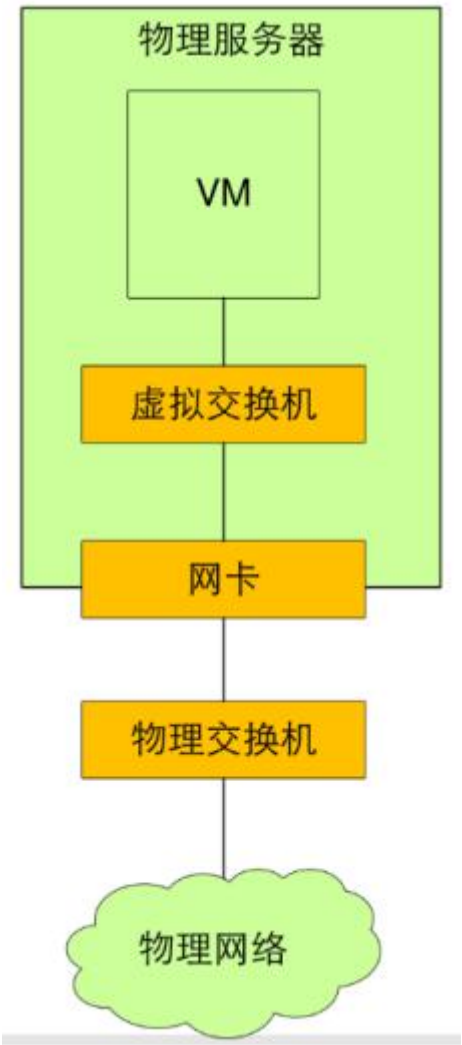
逻辑网络规模可扩展

逻辑网络数量可扩展

网络虚拟化的层次


虚拟服务器边界
VM1
VM2
VM3
虚拟网卡
虚拟网卡
虚拟网卡
服务器IO虚拟化
虚拟机交换机 vswitch
VMM
虚拟接入
物理服务器边界
物理网卡
NIC
MAC/PHY
Rx1
Rx2
Rx1
Rxn
Rx1
Rxn
Rx2
Tx1
Tx2
Tx1
Txn
Tx1
Txn
Tx2
二层网络边界
网络连接
网络交换
IO虚拟化：解决网络资源复用问题
虚拟接入：解决虚拟机网络包标识问题
网络连接：解决一根连接线上承载的带宽
网络交换：物理网络和逻辑网络有效的分离

虚拟交换实现方式




虚拟交换机可以在三个层次实现：

基于服务器 CPU：以软件形式运行在服务器上，实现虚拟交换功能

基于物理网卡：某些物理网卡支持硬件虚拟化功能，通过硬件本身提供的虚拟化功能实现虚拟交换

基于物理交换机：某些物理交换机可通过特殊协议，感知虚拟机的存在，在交换机层实现虚拟交换

基于 CPU 实现的虚拟交换机

服务器内部的通信性能

同一服务器上的虚拟机间报文转发性能好，时延低

虚拟交换机实现虚拟机之间报文的二层软件转发

报文不出服务器，转发路径短，性能高

跨服务器通信性能

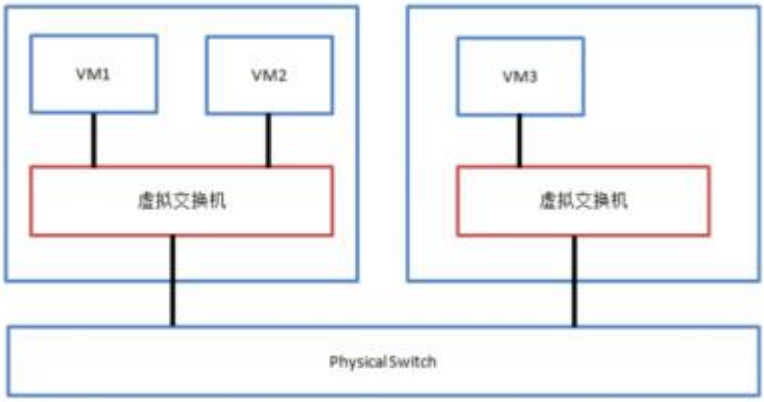
需要经物理交换机进行转发，相比物理交换机实现虚拟交换，由于虚拟交换模块的消耗，性能稍低于物理交换机实现虚拟交换

扩展灵活

由于采用纯软件实现，相比采用 L3 芯片的物理交换机，功能扩展灵活、快速，可以更好地满足云计算的网络需求扩展

规格容量大

服务器内存大，相比物理交换机，在 L2 交换容量、ACL 容量等，远大于物理交换机

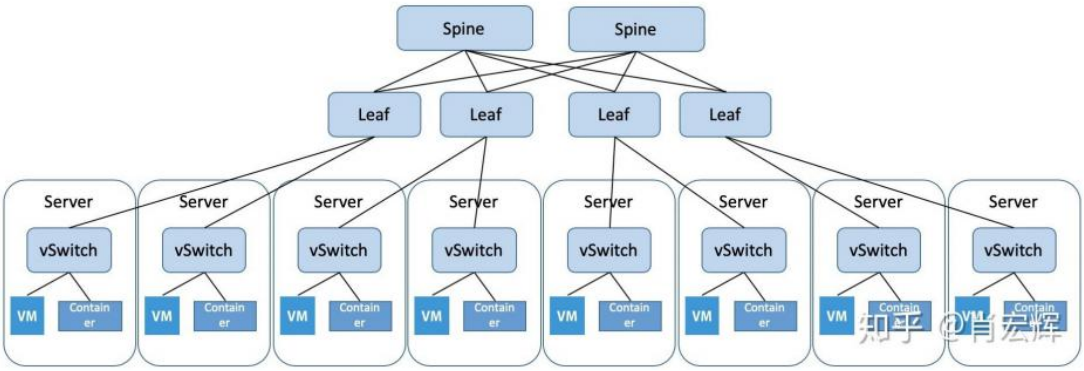


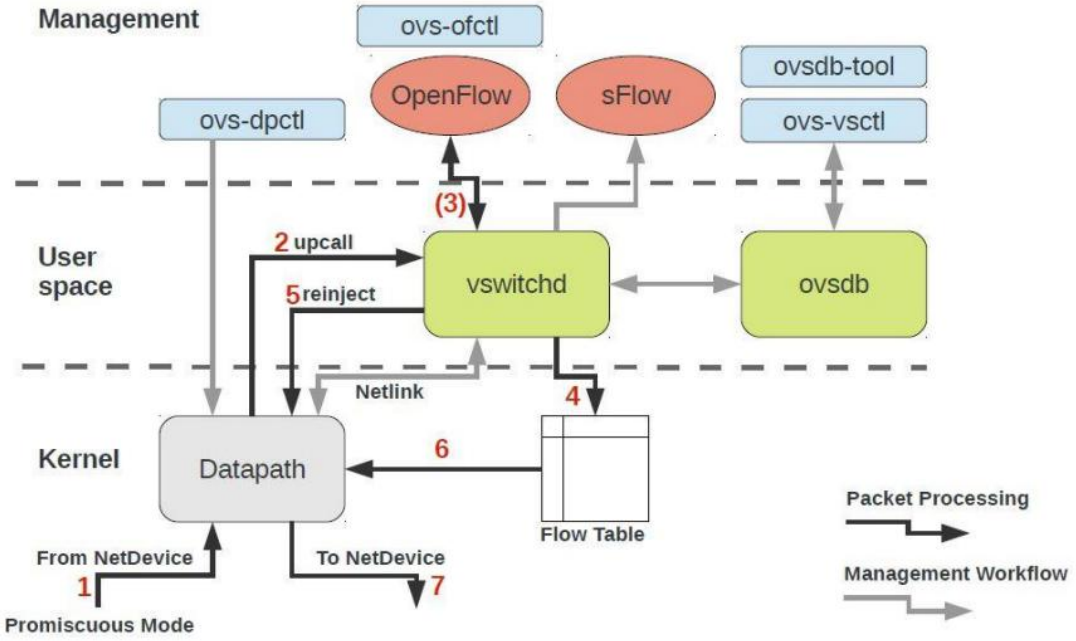

OpenVSwitch（OVS）

OpenVSwitch 是基于软件实现的开源虚拟交换机

80%的 x86 workload 已经是虚拟化；vSwitch 连接服务器内容器和虚（拟？）机

物理网络（underlay）基础之上，再定义一个独立的 overlay 网络；不受物理网络设备控制，完全由 vSwitch 控制






3.4 OpenStack

OpenStack 版本

一年两个版本

城市地点命名

OpenStack 的所属分类

云操作系统；IaaS 服务


OpenStack 优势

快速——OpenStack 安装部署所需要的时间很少

灵活——OpenStack 获得了各大领导厂商的广泛支持，兼容性和适用性极强，使用起来非常方便可靠

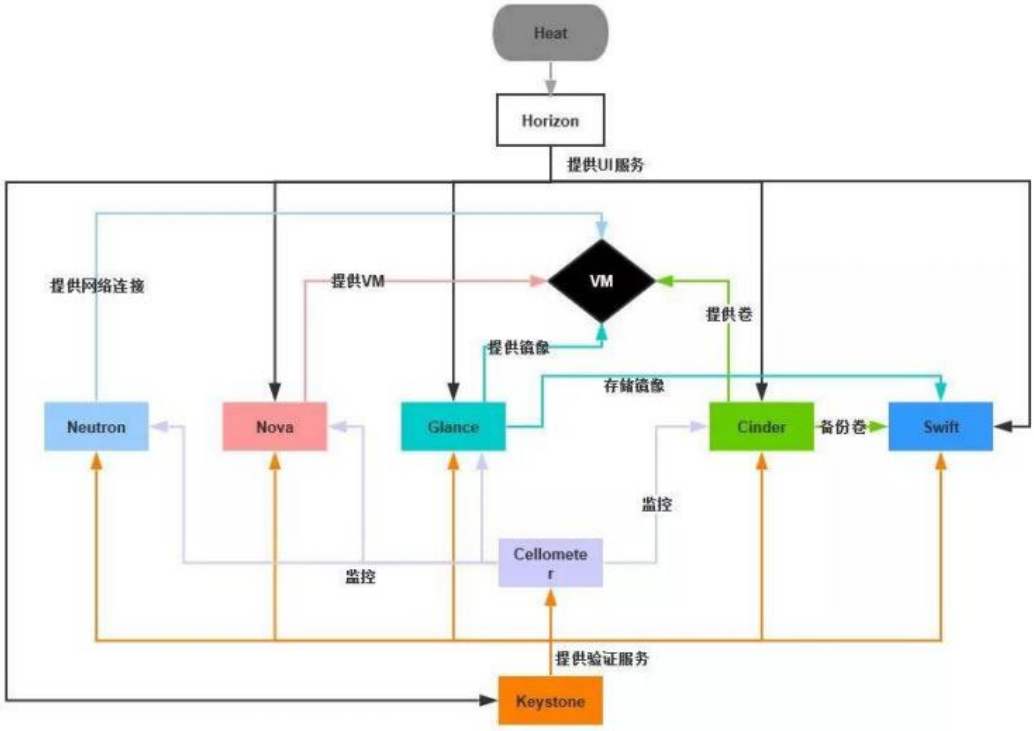
便宜——作为开源项目，OpenStack 的使用成本相对低廉，还能获得源源不断的更新

OpenStack 各组件服务

| | 组件名称 | 服务类型 |
|---|---|---|
| | Horizon | Dashboard，WEB前端服务 |
| | Nova | Compute，计算服务 |
| | Neutron | Networking，网络服务 |
| 存储服务 Storage Services | Swift | Object Storage，对象存储服务 |
| | Cinder | Block Storage，块存储服务 |
| 共享服务 Shared Services | Keystone | Identity service，认证服务 |
| | Glance | Image Service，镜像服务 |
| | Ceilometer | Telemetry，监控服务 |
| 更高级服务 Higher-level services | Heat | Orchestration，集群服务 |
| | Trove | Database Service，数据库服务 |

@鲜枣课堂

OpenStack 组件逻辑关系图




OpenStack 各服务组件

认证服务 Keystone：

为所有的 OpenStack 组件提供认证和访问策略服务

计算设施 Nova：

计算的弹性控制器，管理云实例生命期所需的各种动作；

Nova 本身并不提供任何虚拟能力，使用 libvirt API 与虚拟机的宿主机进行交互

镜像服务器 Glance：

虚拟机镜像发现、注册、检索系统

对象存储服务 Swift：

一种分布式、持续虚拟对象存储，它类似于 Amazon Web Service 的 S3 简单存储服务

快存储服务 Cinder：

为虚拟机提供块级存储，以及可用作存储磁盘的裸卷

文件共享存储服务 Manila：

远程文件系统存储，类似 Linux 上的网络文件系统（NFS）

网络服务 Neutron：

为 Openstack 管理的设备提供网络服务

OpenStack 部署

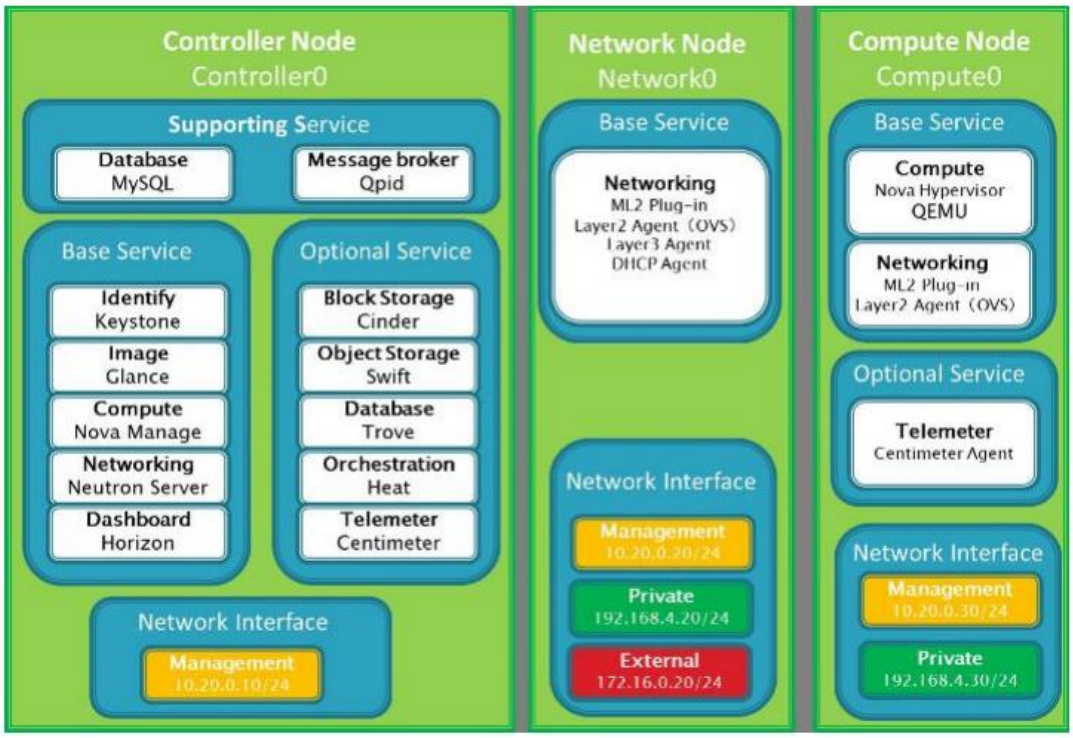
控制节点：对其余节点的控制，包含虚拟机建立，迁移，网络分配，存储分配等等

计算节点：虚拟机运行

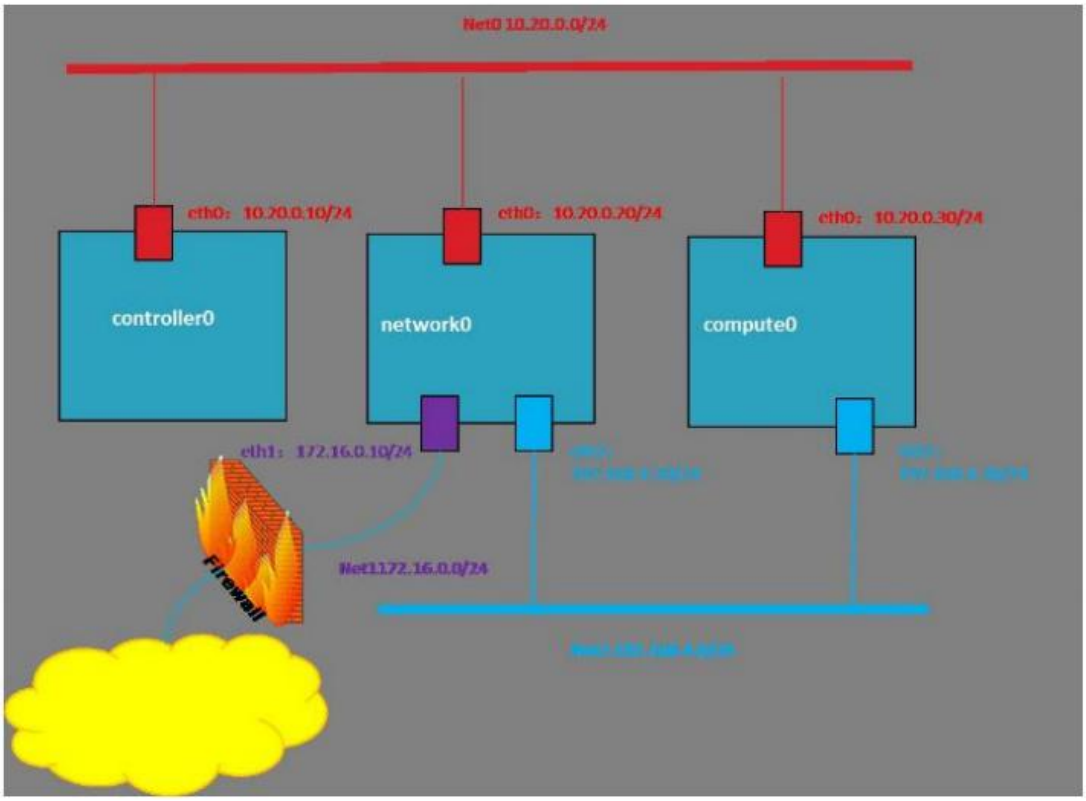
网络节点：负责对外网络与内网络之间的通信

存储节点：负责对虚拟机的额外存储管理等等

OpenStack 架构图——部署




OpenStack 的网络拓扑结构图




3.5 SWIFT 存储系统原理

对象存储服务 Swift

OpenStack 开源云计算项目的子项目之一

可扩展的对象存储系统

扩展性、冗余性、持久性

比如复制和存档数据，图像或视频服务，存储次级静态数据，开发数据存储整合的新应用，存储容量难以估计的数据，为 Web 应用创建基于云的弹性存储。

OpenStack Object Storage（Swift）

分布式对象存储集群，存储容量可达 PB 级

适用永久类型的静态数据的长期存储

例如虚拟机镜像、图片存储、邮件存储和存档备份

软件层面引入一致性哈希技术和数据冗余性

牺牲一定程度的数据一致性来达到高可用性

互联网的应用场景下非结构化数据存储问题

成为云存储的开放标准，打破 Amazon S3 在市场上的垄断地位

Swift 特性

（1）高数据持久性：数据的可靠性，是指数据存储到系统中后，到某一天数据丢失的可能性。

（2）完全对称的系统架构：“对称”意味着 Swift 中各节点可以完全对等，能极大地降低系统维护成本。

（3）无限的可扩展性：一是数据存储容量无限可扩展，二是 Swift 性能（如 QPS、吞吐量等）可线性提升。

（4）无单点故障：整个 Swift 集群中，也没有一个角色是单点的，并且在架构和设计上保证无单点业务是有效的。

（5）简单、可依赖：简单体现在实现易懂、架构优美、代码整洁；可依赖是指 Swift 经测试、分析之后，可以放心大胆地将 Swift 用于最核心的存储业务上。

Swift 应用场景

Swift 提供的服务与 Amazon S3 相同，适用于许多应用场景。

Swift v.s. HDFS

相似的目的——实现冗余、快速分布式存储

技术差异：

Swift 元数据呈分布式，跨集群复制；HDFS 使用了中央系统来维护文件元数据（Namenode），容易成为单一故障点，很难扩展到大规模环境

Swift 考虑到多租户架构，HDFS 没有

Swift 支持数据的并发写操作；HDFS 每次只能有一个文件写入

Swift 用 Python 来编写，而 HDFS 用 Java 来编写

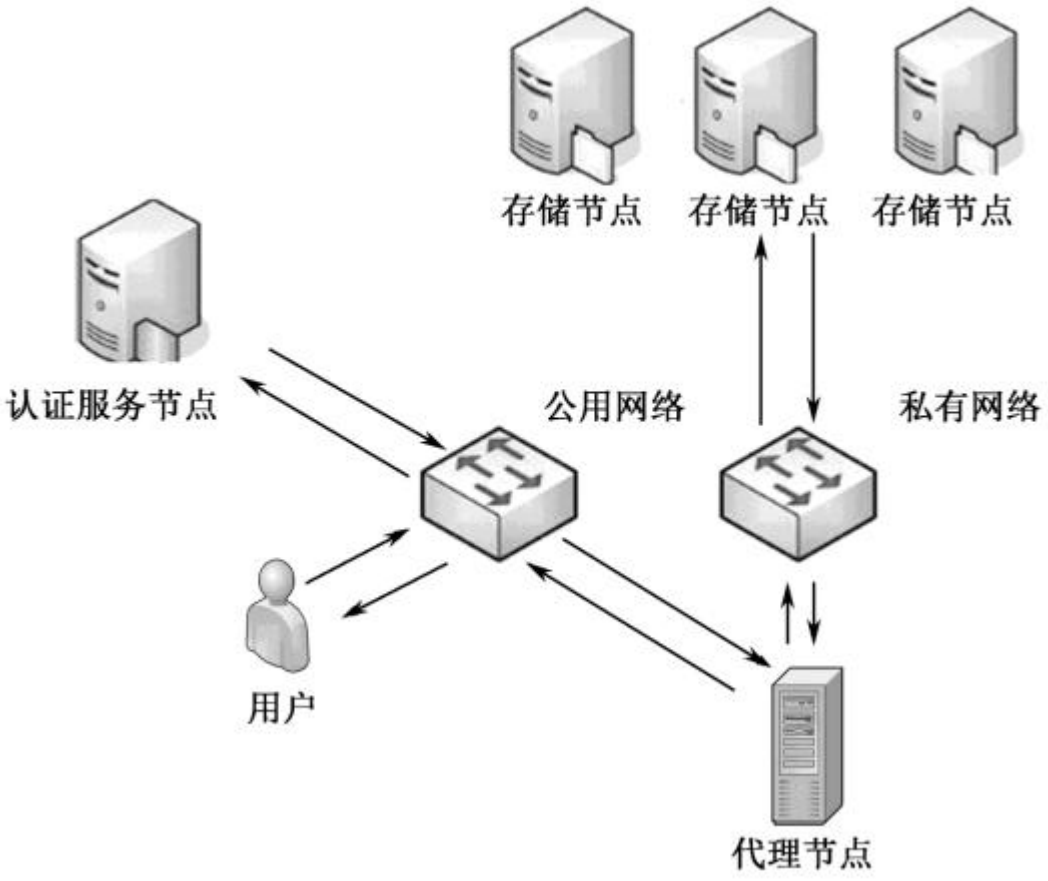
Swift 通用的存储解决方案，可靠存储数量多且大小不一的文件；HDFS 存储数量中等的大文件以支持数据处理

Swift 系统框图

使用 OpenStack 的认证服务 Keystone，目的在于实现统一 OpenStack 各个项目间的认证

Swift 主要有三个组成部分：Proxy Server、Storage Server 和 Consistency Server。

其中 Storage 和 Consistency 服务均允许在 Storage Node 上。



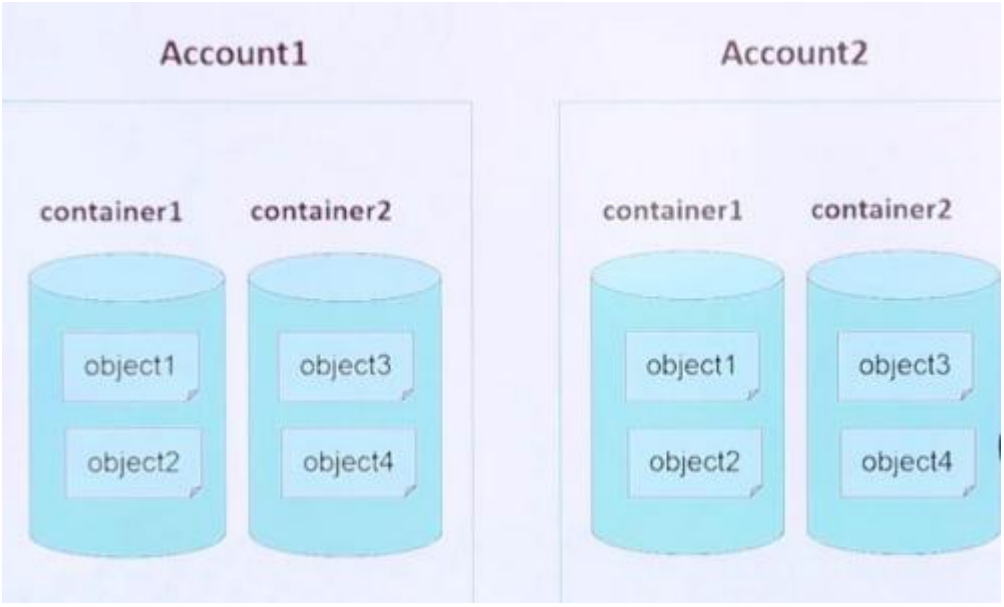

存储逻辑结构

Account：用户定义的管理存储区域

Container：容器，存储的隔间，类似于子文件夹

Object：对象，包含基本的存储实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，有 Account 环、Container 环和 Object 环




物理存储结构

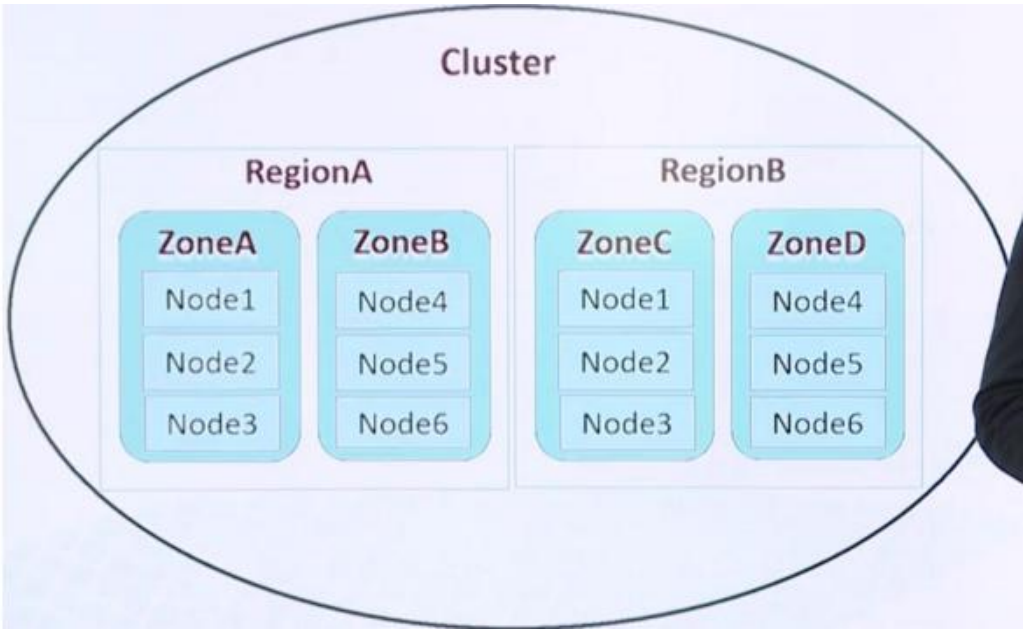
Region：地域，不同城市的地理位置

Zone：可用区，物理的网络，供电，空调等基础设施的隔离，不同的可用区可能是不同数据中心机房，也可能是同一个数据中心，不同的供电、供水、网络、接入等等一些隔离的系统

Node：结点，代表着一台存储服务器

Disk：磁盘，代表着物理的存储设备

Cluster：指的是群集，是为冗余考虑而设计的架构




3.6 一致性哈希算法

一致性哈希（Consistent Hashing）

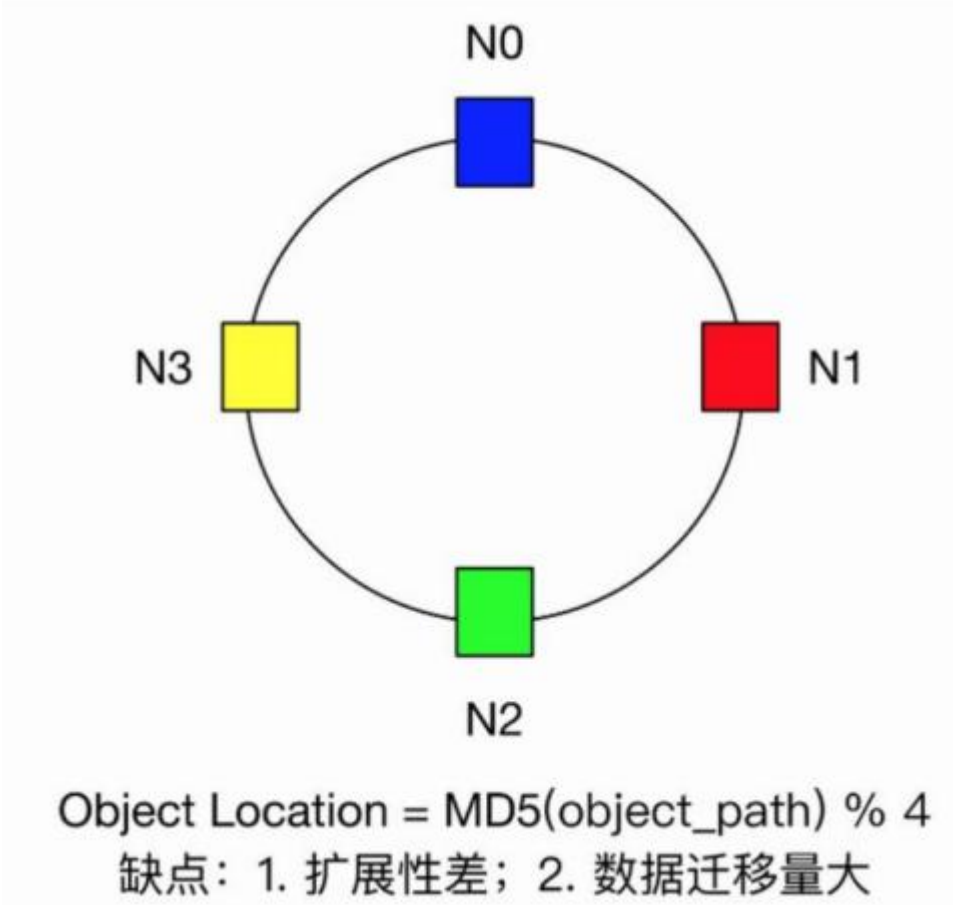
寻址问题：如何将海量级别的对象，分布到成千上万台服务器和硬盘设备上？

（1）普通哈希算法

对对象名称哈希计算并取模，将对象放置在合适物理节点上

缺点：可扩展性差。

一旦面临添加和删除节点时，模值就会改变，所有 location 值都要重新计算，导致大量的数据迁移。



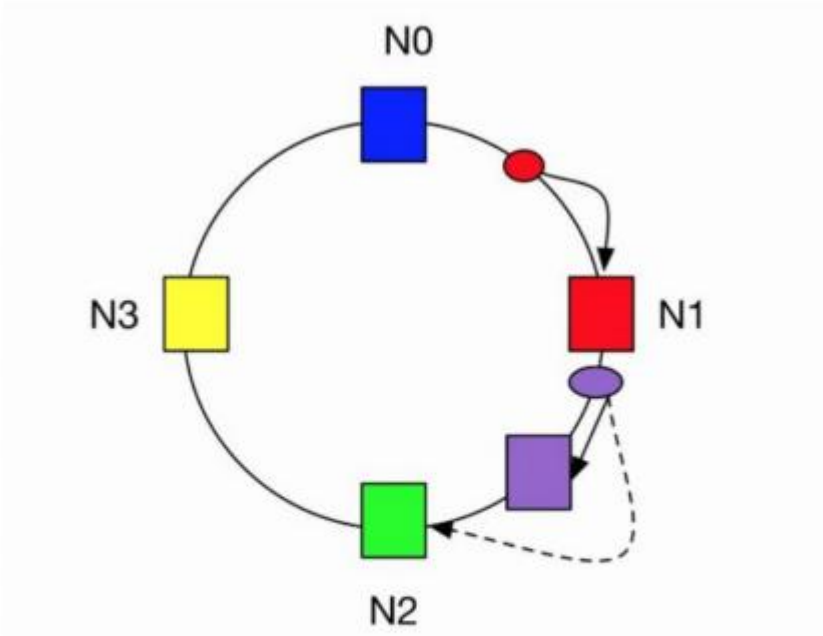

（2）一致性哈希算法

模值采用固定值 1024；

当添加一个新的节点 N4，通过计算节点 N4 的哈希值，计算它在[0-1024)环上的位置；

哈希值落在(hash(N1), hash(N4)]之间的对象，就需要从 N2 节点迁移到 N4 节点上。

缺点：数据迁移量减少，但添加和删除操作，造成物理节点的哈希值在[0-1024)之间不均匀分布，导致节点上数据分布不均匀。 数据迁移是在两个节点之间进行，影响用户对节点 N2 上的对象的 IO 操作




Node Location = MD5(HostName) % 1024
Object Location = MD5(object_path) % 1024
缺点：1. 扩展性仍然较差；2. 数据分布不均匀

（3）改进型一致性哈希算法

环上的方格子不再代表一个物理节点，而是一个虚拟节点。

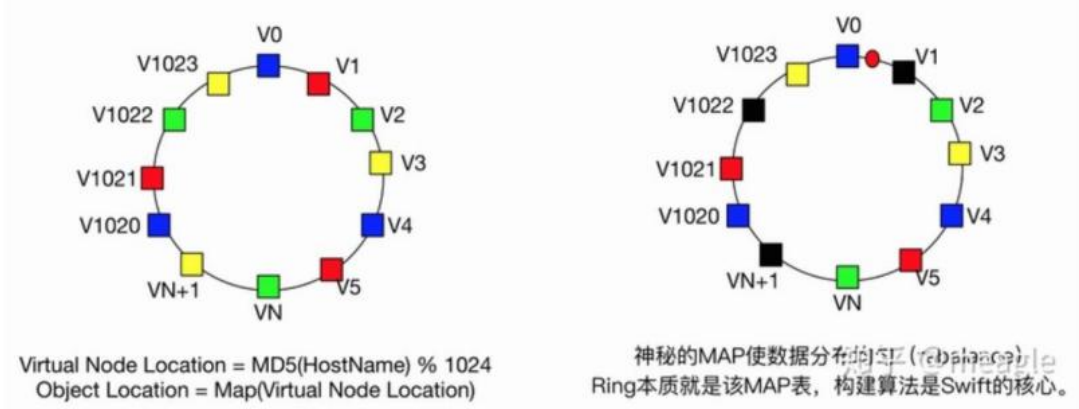
虚拟节点和物理节点之间有一张映射表，如 N0 ==> (V0, V4, V1024)。一个物理节点对应一个虚拟节点集合。

对象名称到虚拟节点之间的映射关系不会变，即模值不变。

添加节点时 N4 时，N1、N2、N3 节点都会贡献一部分虚拟节点给 N4，形成 N4 ==> (V1, VN+1, V1022)。即数据迁移时，N1、N2、N3 都会向 N4 节点写数据。

如果 hash(object_name) % 1024==V1，那么这个文件将会从 N1 上迁移到 N4 上；

每个节点拥有的大致相当的虚拟节点数目，数据分布更加平衡。




3.7 Quorum 仲裁协议

（1）定义 N：数据的副本总数；W：写操作被确认接受的副本数量；R：读操作的副本数量。

（2）强一致性：R+W>N，以保证对副本的读写操作会产生交集，从而保证可以读取到最新版本；

如果 W=N，R=1，则需要全部更新，适合大量读少量写操作场景下的强一致性；

如果 R=N，W=1，则只更新一个副本，通过读取全部副本来得到最新版本，适合大量写少量读场景下的强一致性。

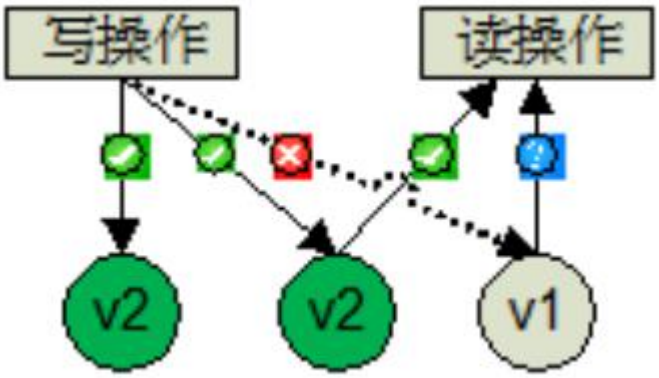
（3）弱一致性：R+W≤N，如果读写操作的副本集合不产生交集，就可能会读到脏数据；适合对一致性要求比较低的场景。

Swift 中 Quorum 仲裁协议配置

Swift 默认配置是 N=3，W=2>N/2，R=1 或 2

当 R=1 时意味着某一个读操作成功便立刻返回，此种情况下可能会读取到旧版本（弱一致性模型）

当 R=2 时，同时读取 2 个副本的元数据信息，然后比较时间戳来确定哪个是最新版本（强一致性模型）；如果数据出现了不一致，后台服务进程会在一定时间窗口内通过检测和复制协议来完成数据同步，从而保证达到最终一致性




4. 云计算框架与系统

4.1 Google 云计算

4.1.1 GFS

文件系统基础

什么是文件系统？

用于持久地存储数据的系统

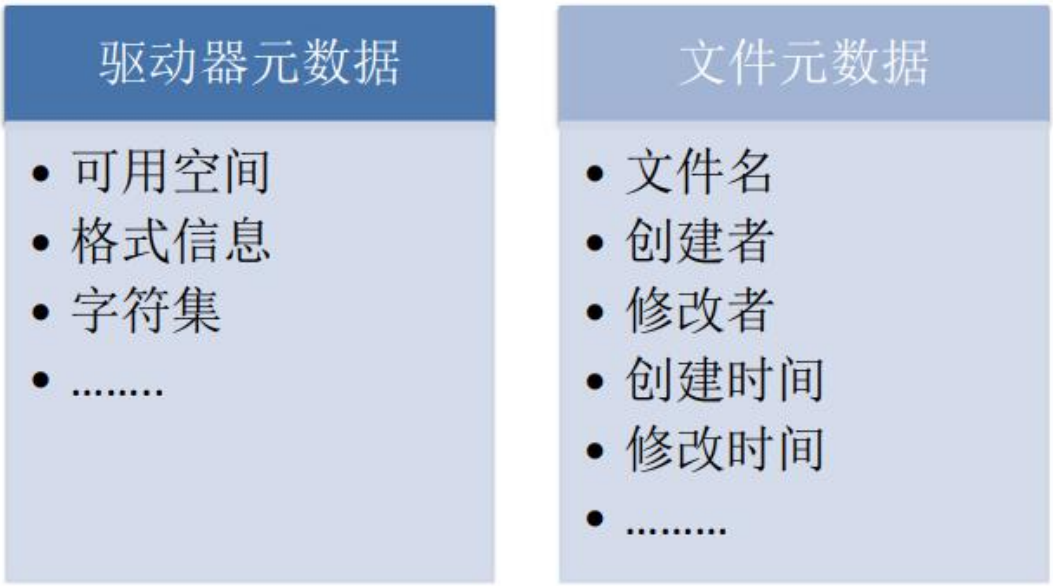
规范路径

定位文件的最短绝对路径

文件系统的存储内容

主要内容：用户的实际数据

元数据：驱动器元数据与文件元数据




GFS 的假设与目标

硬件出错是正常而非异常

系统应当由大量廉价、易损的硬件组成

必须保持文件系统整体的可靠性

主要负载是流数据读写

主要用于程序处理批量数据，而非与用户的交互或随机读写

数据写主要是“追加写”，“插入写”非常少

需要存储大尺寸的文件

存储的文件尺寸可能是 GB 或 TB 量级，而且应当能支持存储成千上万的大尺寸文件

GFS 的设计思路

将文件划分为若干块（Chunk）存储

每个块固定大小（64M）

通过冗余来提高可靠性

每个数据块至少在 3 个数据块服务器上冗余

数据块损坏概率？

通过单个 master 来协调数据访问、元数据存储

结构简单，容易保持元数据一致性

GFS 的架构

单一 Master，若干 ChunkServer

单一 Master 问题

分布式系统设计告诉我们：

这是单点故障

这是性能瓶颈

GFS 的解决办法

单点故障问题：采用多个（如 3 个）影子 Master 节点进行热备，一旦主节点损坏，立刻选举一个新的主节点服务

性能瓶颈问题：

尽可能减少数据存取中 Master 的参与程度

不使用 Master 读取数据，仅用于保存元数据

客户端缓存元数据

采用大尺寸的数据块（64M）

数据修改顺序交由 Primary Chunk Server 完成

Master 节点的任务

存储元数据

与 ChunkServer 进行周期性通信

发送指令，搜集状态，跟踪数据块的完好性

数据块创建、复制及负载均衡

对 ChunkServer 的空间使用和访问速度进行负载均衡，平滑数据存储和访问请求的负载

对数据块进行复制、分散到 ChunkServer 上

一旦数据块冗余数小于最低数，就发起复制操作

垃圾回收

在日志中记录删除操作，并将文件改名隐藏

缓慢地回收隐藏文件

与传统文件删除相比更简单、更安全

陈旧数据块删除

探测陈旧的数据块，并删除

GFS 架构的特点

不缓存数据

GFS 的文件操作大部分是流式读写，不存在大量的重复读写，使用 Cache 对性能提高不大

Chunk Server 上的数据存取使用本地文件系统，如果某个 Chunk 读取频繁，文件系统具有缓存

从可行性看，Cache 与实际数据的一致性维护也极其复杂

GFS 的容错方法（GFS 的容错机制）

Chunk Server 容错

每个 Chunk 有多个存储副本（通常是 3 个），分别存储于不同的服务器上

每个 Chunk 又划分为若干 Block（64KB），每个 Block 对应一个 32bit 的校验码，保证数据正确（若某个 Block 错误，则转移至其他 Chunk 副本）

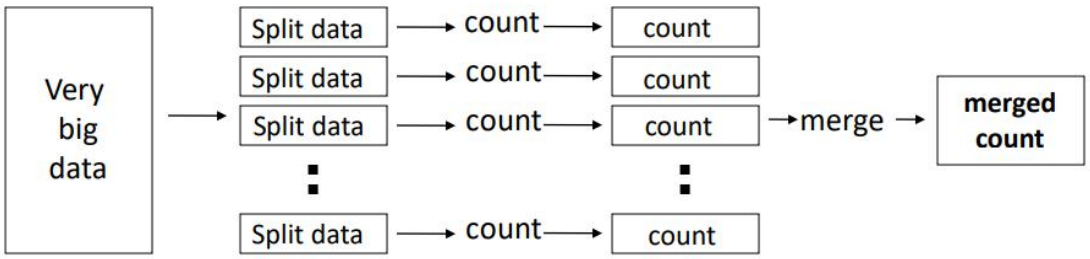
Master 容错

三类元数据：命名空间（目录结构）、Chunk 与文件名的映射以及 Chunk 副本的位置信息

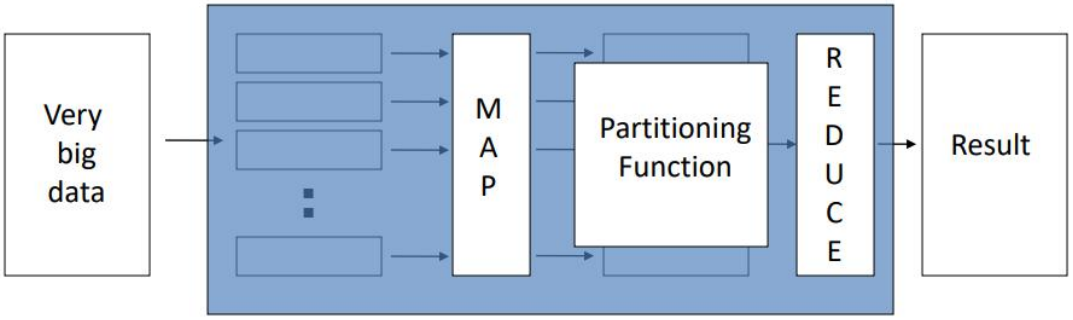
前两类通过日志提供容错，Chunk 副本信息存储于 Chunk Server，Master 出现故障时可恢复

### 4.1.2 MapReduce

Distributed Word Count




Map+Reduce




Map：

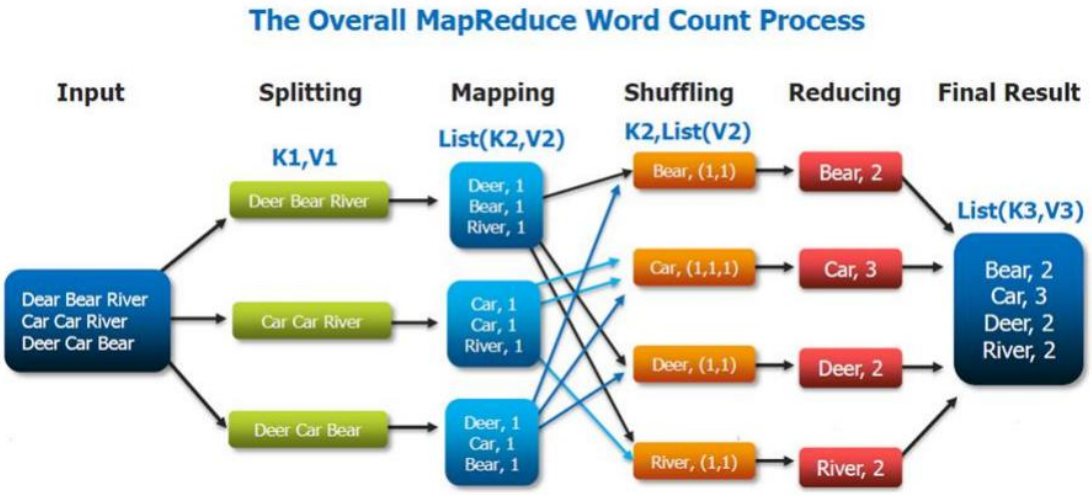
接受键值对输入

发射（emit）键值对中间值

Reduce：

接受键值对中间值

发射键值对输出




模型中的功能组件

Map

处理一个键值对以生成键值对中间值

Reduce

合并所有有相同键的中间值

Partition

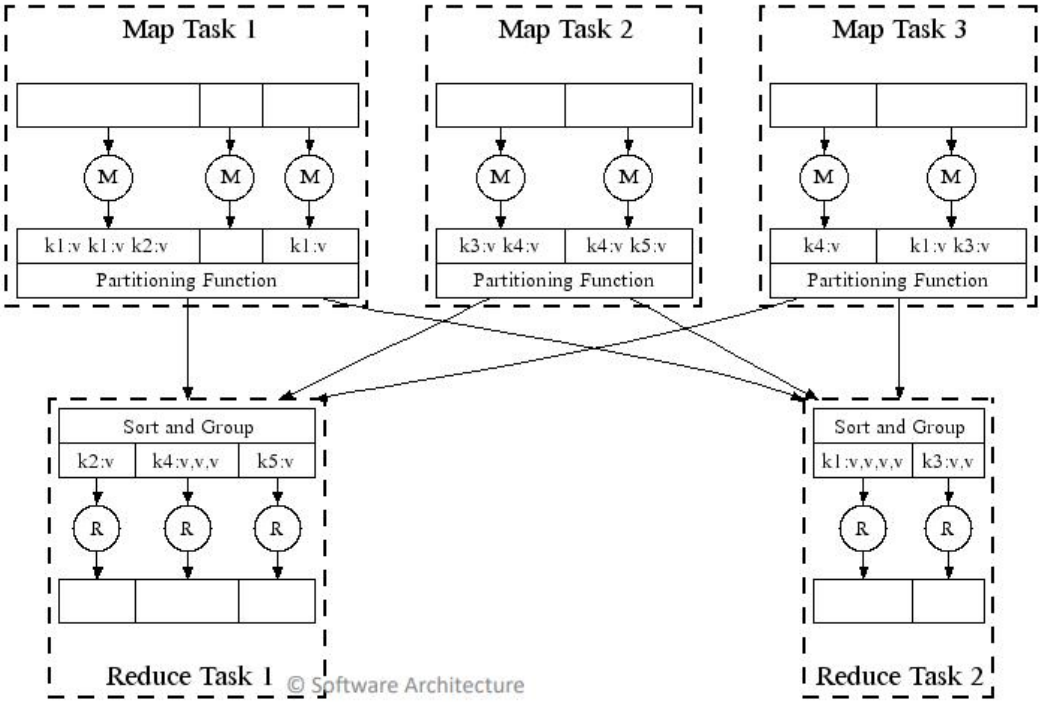
默认为 hash(key) mod R

良好地均衡负载

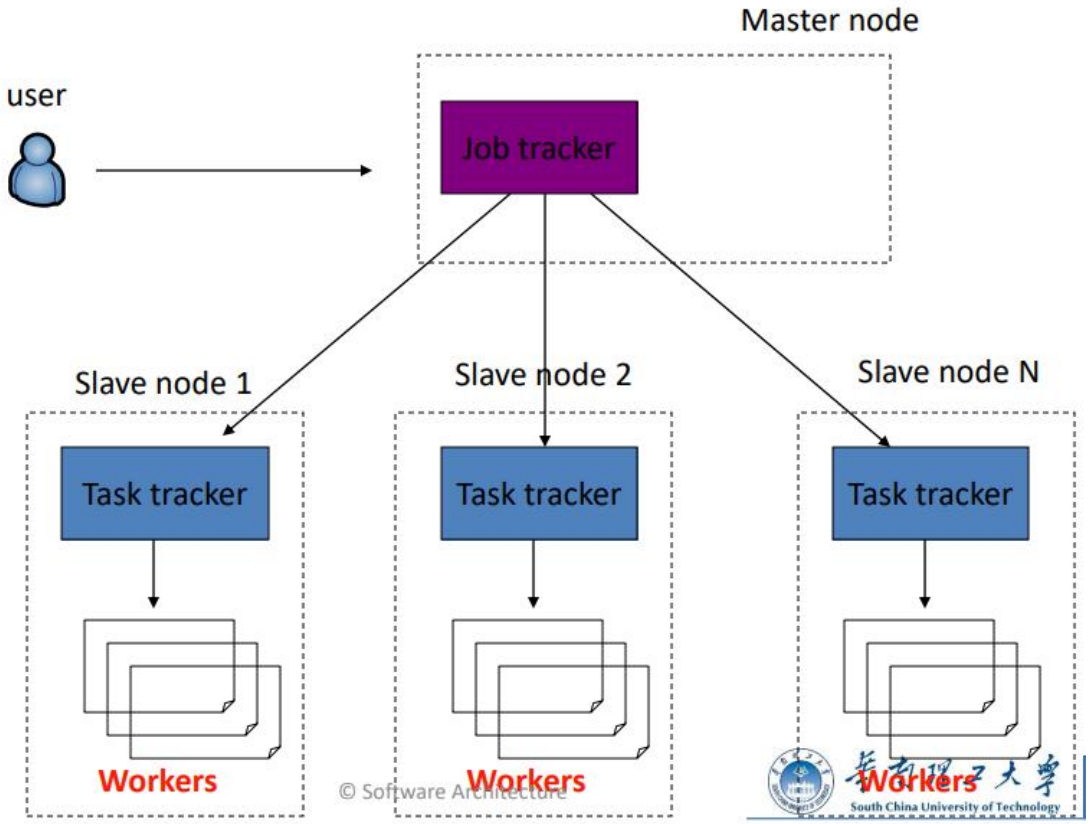
Shuffling

如何在网络中减少通信的交通流量？

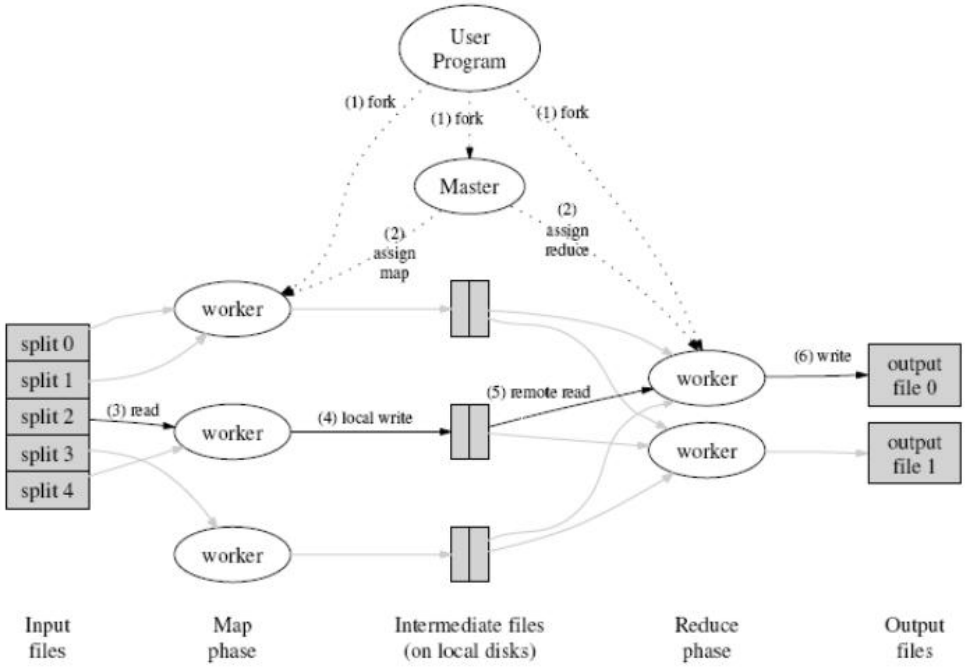
并行执行




架构概览




如何工作？




MapReduce 的容错

Worker 故障

Master 周期性的 ping 每个 worker。如果 master 在一个确定的时间段内没有收到 worker 返回的信息，那么它将把这个 worker 标记成失效

重新执行该节点上已经执行或尚未执行的 Map 任务

重新执行该节点上未完成的 Reduce 任务，已完成的不再执行

Master 故障

定期写入检查点数据

从检查点恢复

MapReduce 的优化

任务备份机制

慢的 workers 会严重地拖延整个执行完成的时间

由于其他的任务占用了资源

磁盘损坏

解决方案：在临近结束的时候，启动多个进程来执行尚未完成的任务

谁先完成，就算谁

可以十分显著地提高执行效率

本地处理

Master 调度策略：

向 GFS 询问获得输入文件 chunk 副本的位置信息

Map tasks 的输入数据通常按 64MB 来划分（GFS chunk 大小）

按照 chunk 所在的机器或机器所在机架的范围进行调度

效果

绝大部分机器从本地读取文件作为输入，节省大量带宽

需要强调的点

Map 完成后 Reduce 才开始

Master 必须交流中间文件的位置

基于数据位置调度任务

若 map worker 在 reduce 前的任何时候失败，任务必须完全重新运行

MapReduce 库为我们完成了大部分困难的工作！

如何使用

用户需要完成的清单：

表明：

输入输出文件

M：map 任务的数量

R：reduce 任务的数量

W：机器的数量

编写 map 和 reduce 函数

提交作业（job）

### 4.1.3 BigTable

BigTable：基于 GFS 和 Chubby 的分布式存储系统

对数据进行结构化存储和管理

Google 的需求

数据存储可靠性

高速数据检索与读取

存储海量的记录（若干 TB）

可以保存记录的多个版本

假设

与写操作相比，数据记录读操作占绝大多数工作负载

单个节点故障损坏是常见的

磁盘是廉价的

可以不提供标准接口

Google 既能控制数据库设计，又能进行应用系统设计

设计目标

具有广泛的适应性

支持 Google 系列产品的存储需求

具有很强的可扩展性

根据需要随时加入或撤销服务器

应对不断增多的访问请求

高可用性

单个节点易损，但要确保几乎所有的情况下系统都可用

简单性

简单的底层系统可减少系统出错概率，为上层开发带来便利

逻辑视图

总体上，与关系数据库中的表类似

(row: string, column: string, time:int64)->string

| Row Key | Time Stamp | Column Contents | Column Anchor | | Column “mine” |
|---|---|---|---|---|---|
| | | | cnnsi.com | my.look.ca | |
| "com.cnn.www” | T9 | | CNN | | |
| | T8 | | | CNN.COM | |
| | T6 | “<html>..” | | | Text/html |
| | T6 | “<html>..” | | | |
| | T3 | “<html>..” | | | |

数据模型

行

每行数据有一个可排序的关键字和任意列项

字符串、整数、二进制串甚至可串行化的结构都可以作为行键

表按照行键的“逐字节排序”顺序对行进行有序化处理

表内数据非常‘稀疏’，不同的行的列的数目可以大不相同

URL 是较为常见的行键，存储时需要倒排

统一地址域的网页连续存储，便于查找、分析和压缩

mp3.baidu.com/index.asp→com.baidu.mp3/index.asp

列

特定含义的数据的集合，如图片、链接等

可将多个列归并为一组，称为族（family）

采用 族:限定词 的语法规则进行定义

“fileattr: owning_group”, “fileattr: owning_user”, etc

同一个族的数据被压缩在一起保存

族是 BigTable 中访问控制的基本单元

时间戳

保存不同时期的数据，如“网页快照”

“A big table”

表中的列可以不受限制地增长

表中的数据几乎可以无限地增加

通过(row, col, timestamp)查询

通过(row, col, MOST_RECENT)查询

无数据校验

每行都可存储任意数目的列

BigTable 不对列的最少数目进行约束

任意类型的数据均可存储

BigTable 将所有数据均看作为字符串

数据的有效性校验由构建于其上的应用系统完成

一致性

针对同一行的多个操作可以分组合并

不支持对多行进行修改的操作符

物理视图

逻辑上的“表”被划分为若干子表（Tablet）

每个 Tablet 由多个 SSTable 文件组成

SSTable 文件存储在 GFS 之上

每个子表存储了 table 的一部分行

元数据：起始行键、终止行键

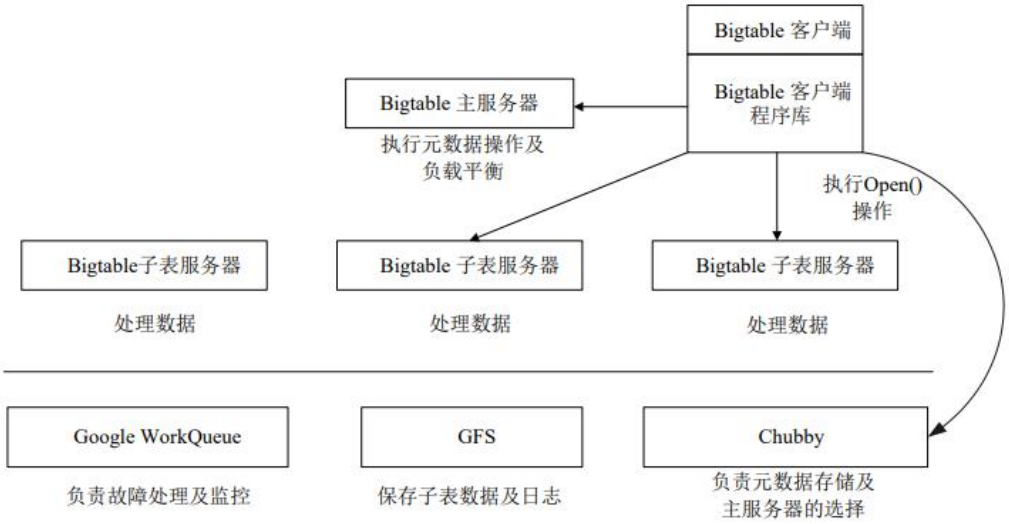
如果子表体积超过阈值（如 200M），则进行分割


BigTable 系统架构

Chubby：元数据存储和选择主服务器

主服务器：子服务器的负载均衡以及状态监控

子表服务器：子表的读写请求，以及子表的分裂或合并；并不真实存储数据，而相当于连接 Bigtable 和 GFS 的代理




主节点的职责

为每个子表服务器分配子表

与 GFS 垃圾回收进行交互，收回废弃的 SSTable

探测子表服务器的故障与恢复

负载均衡

并没有存储元数据

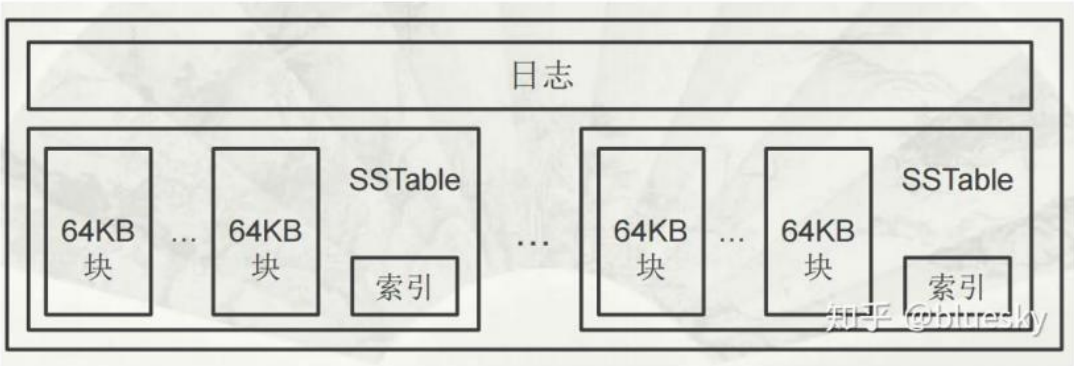
有效缓解单点故障

子表服务器

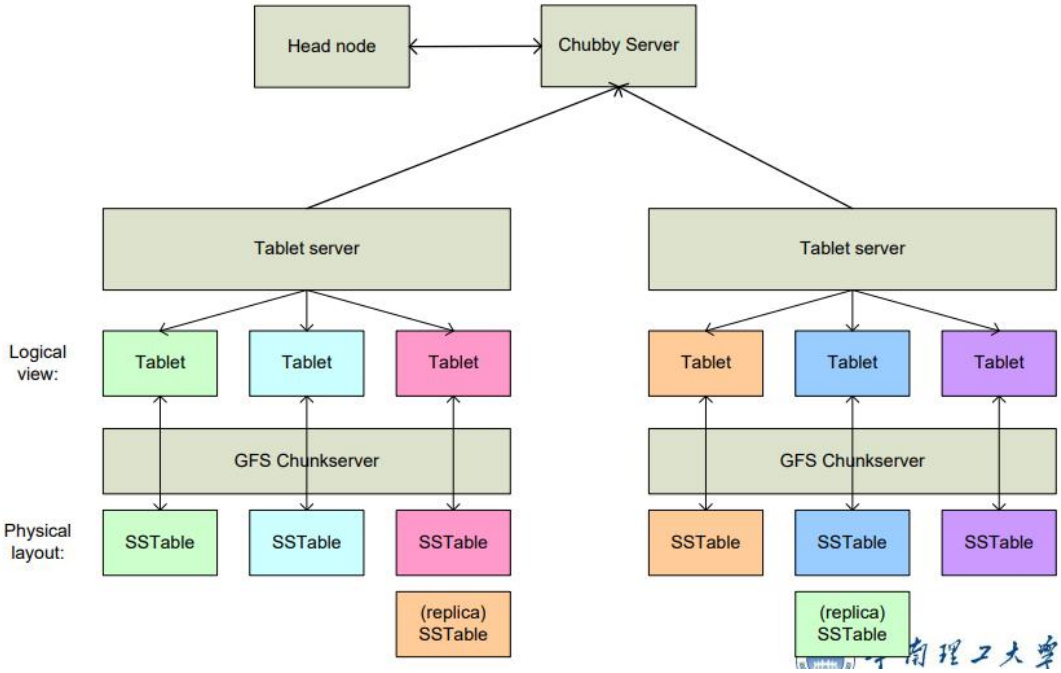
保存一个日志文件和若干数量的子表

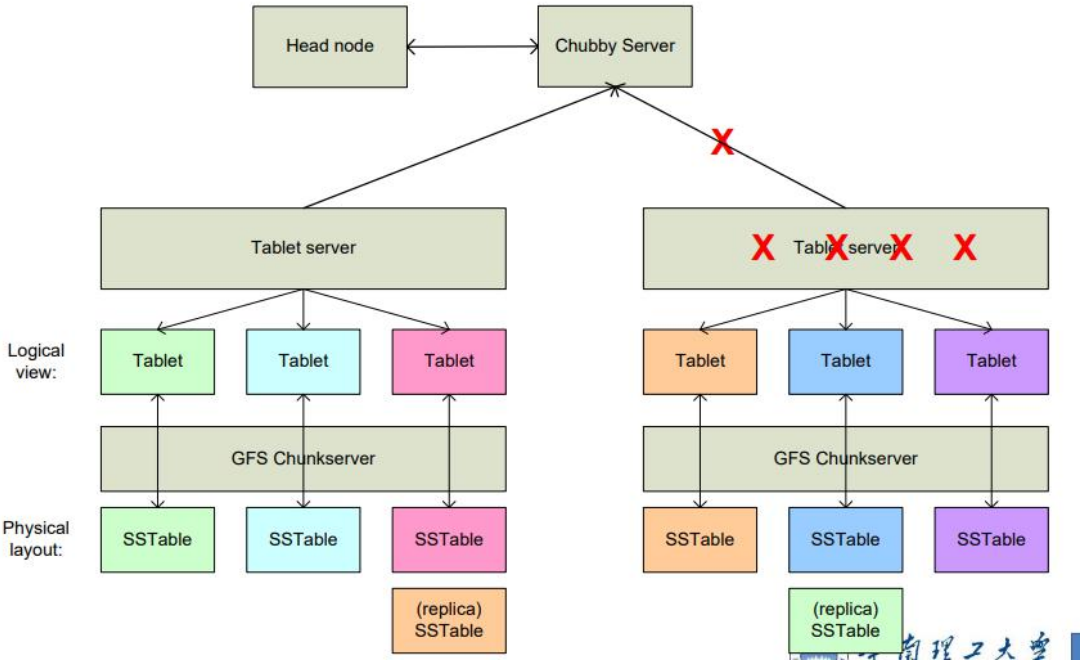
Bigtable 规定将日志的内容按照键值进行排序

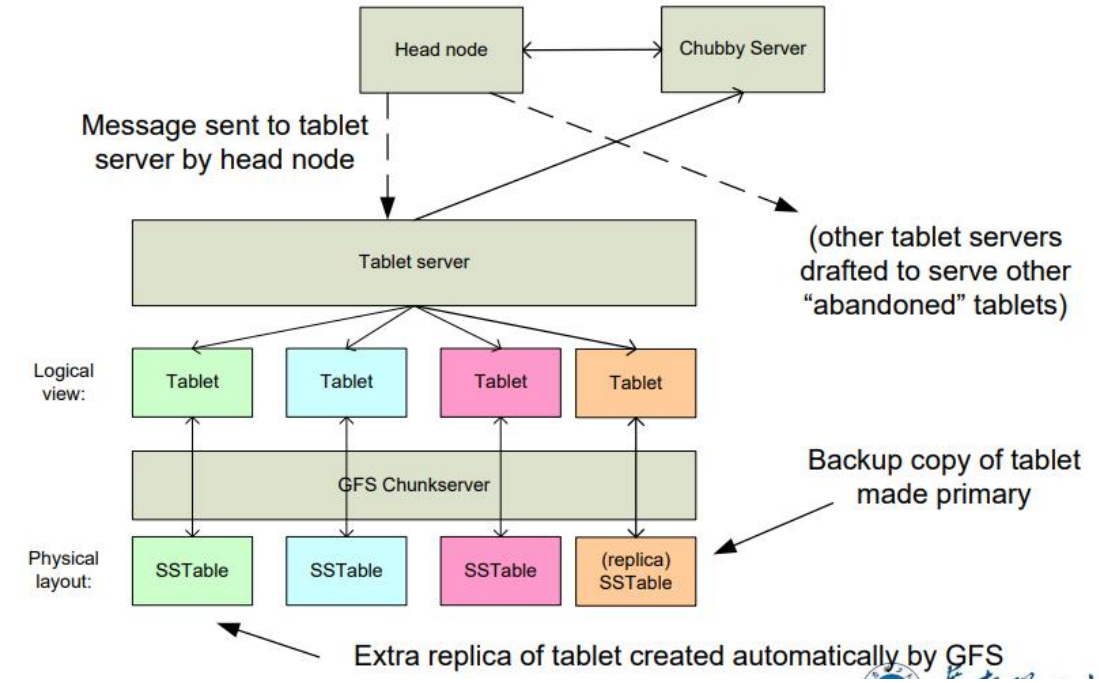
子表数量从几十到上千不等，通常 100 个左右




子表服务器故障


Head node
Chubby Server
Tablet server
Tablet server
Logical view:
Tablet
Tablet
Tablet
Tablet
Tablet
Tablet
GFS Chunkserver
GFS Chunkserver
Physical layout:
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
(replica) SSTable
(replica) SSTable

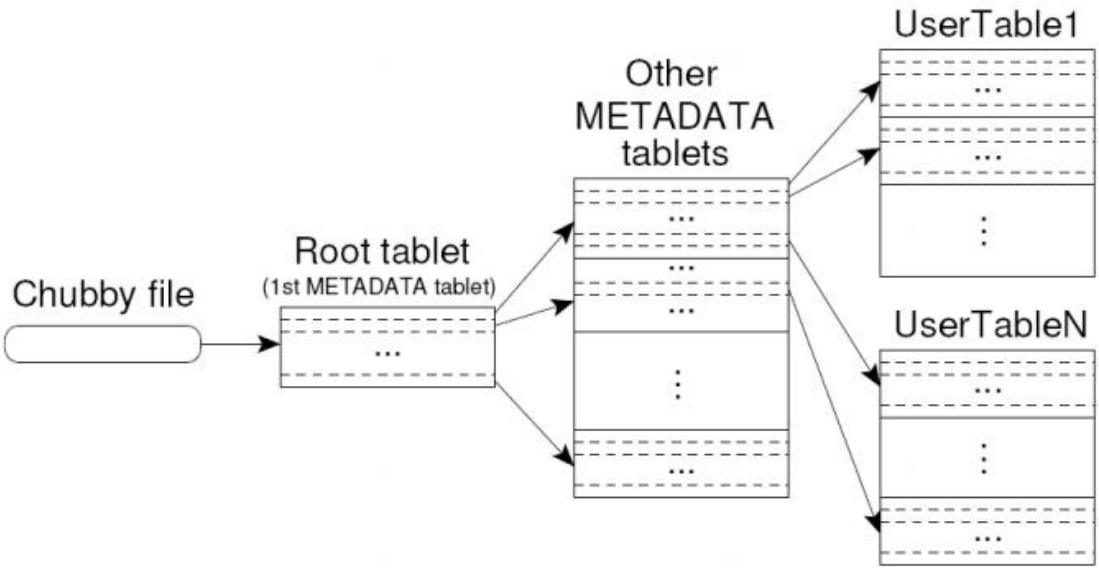

Head node
Chubby Server
X
Tablet server
X X X X
Logical view:
Tablet
Tablet
Tablet
Tablet
Tablet
Tablet
GFS Chunkserver
GFS Chunkserver
Physical layout:
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
(replica) SSTable
(replica) SSTable




数据访问方式

采用类似 B+树的三层查询体系

客户端缓存子表的位置信息；缓存找不到子表的位置信息，需要查找此三层结构；包括访问一次 Chubby 服务，访问两次子服务器




数据写的流程

任何对子表的写操作都会记录到一个存储在 GFS 之上的 commit log 中

每个子表服务器上所有子表变化对应于一个 commit log

新的数据存储到子表服务器的内存（memtable）

次压缩

旧数据存储在 SSTable 中，而新数据存放在 memtable 中

当 memtable 体积超过一定阈值，将形成 SSTable，并写入 GFS

每个 tablet 对应多个 SSTable

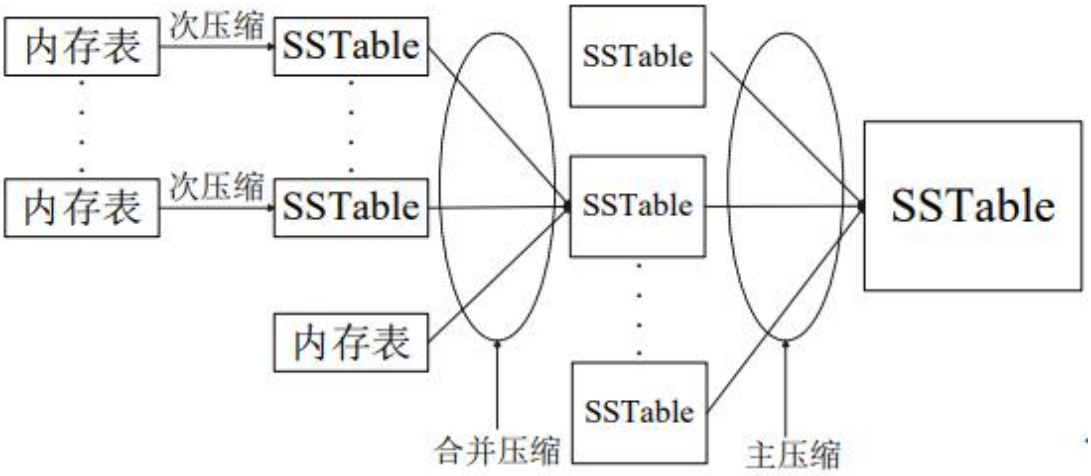
合并压缩

tablet 含有多个 SSTable 导致查询效率低

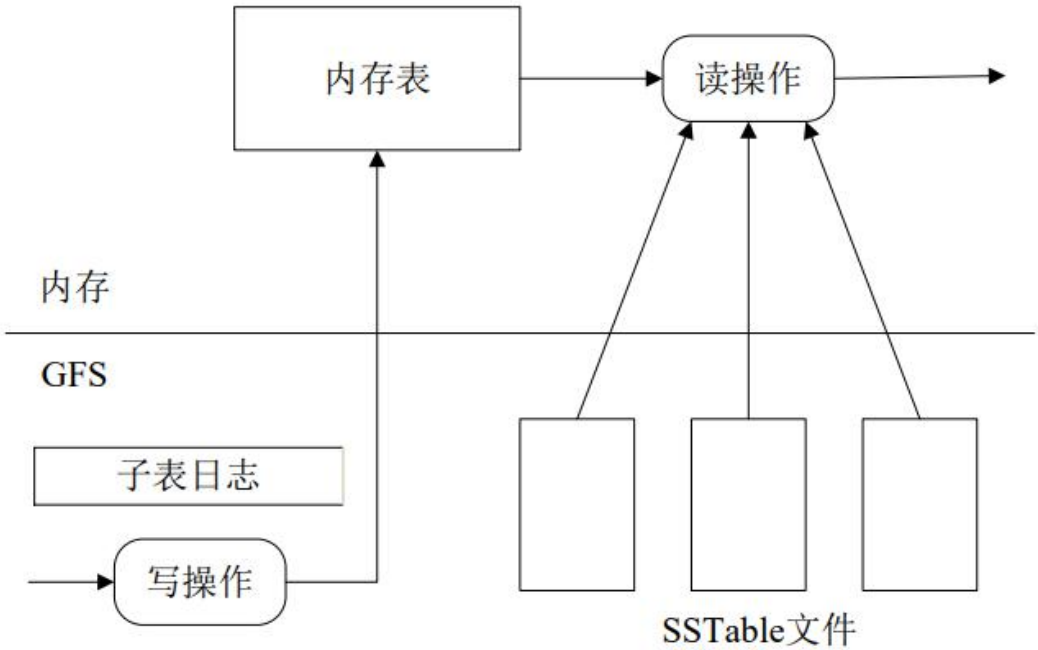
合并压缩操作读取多个 SSTable，创建一个新的 SSTable 来保持其中的最新数据

旧的 SSTable 删除

如果合并压缩操作完成后，tablet 只包含一个 SSTable，那么该操作也称为主压缩




数据存储与读取流程




子表服务器故障恢复

新的故障

子表服务器内存中的 memtable 丢失

恢复方法

按照 tablet 将该服务器对应的日志分片

为每个失效 tablet 分配新的子表服务器

新子表服务器读取对应的分段 commit log，并按照日志修改 tablet

删除 commit log 中已实施的内容

重新对外提供服务

BigTable 和 GFS 的关系

集群包括主服务器和子服务器，主服务器负责将子表分配给子服务器，而具体的数据服务则全权由子服务器负责

子服务器没有真的存储数据（除了内存中 memtable 的数据），数据的真实位置只有 GFS 才知道

主服务器将子表分配给子服务器的意思是，子服务器获取了子表的所有 SSTable 文件名，子服务器通过一些索引机制可以知道所需要的数据在哪个 SSTable 文件，然后从 GFS 中读取 SSTable 文件的数据，这个 SSTable 文件可能分布在好几台 chunkserver

## 4.2 Hadoop 开源平台

### 4.2.1 HDFS

是 GFS 的开源实现

容量大：TB 或者是 PB 级别的

将数据保存到大量的节点当中

支持很大单个文件

高可靠性、快速访问、高可扩展

大量的数据复制

简单加入更多服务器

HDFS 是针对 MapReduce 设计

数据尽可能根据其本地局部性进行访问与计算

HDFS 设计

基于块的文件存储

块进行复制的形式放置，按照块的方式随机选择存储节点

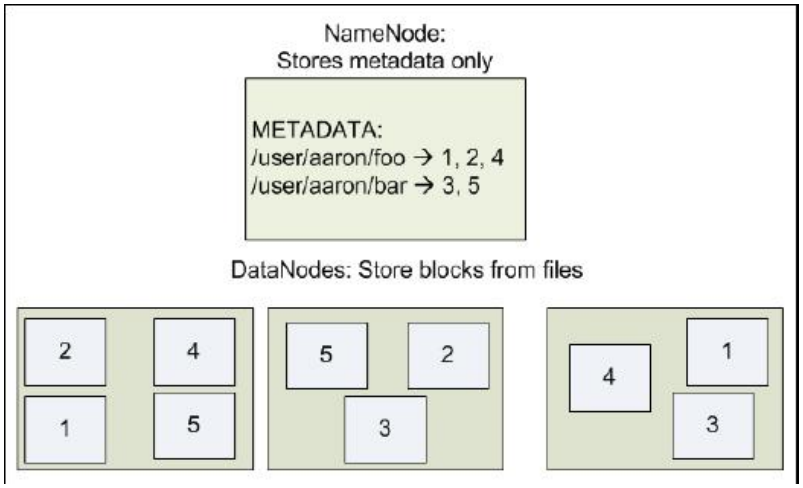
副本的默认数目是 3

默认的块的大小是 64MB

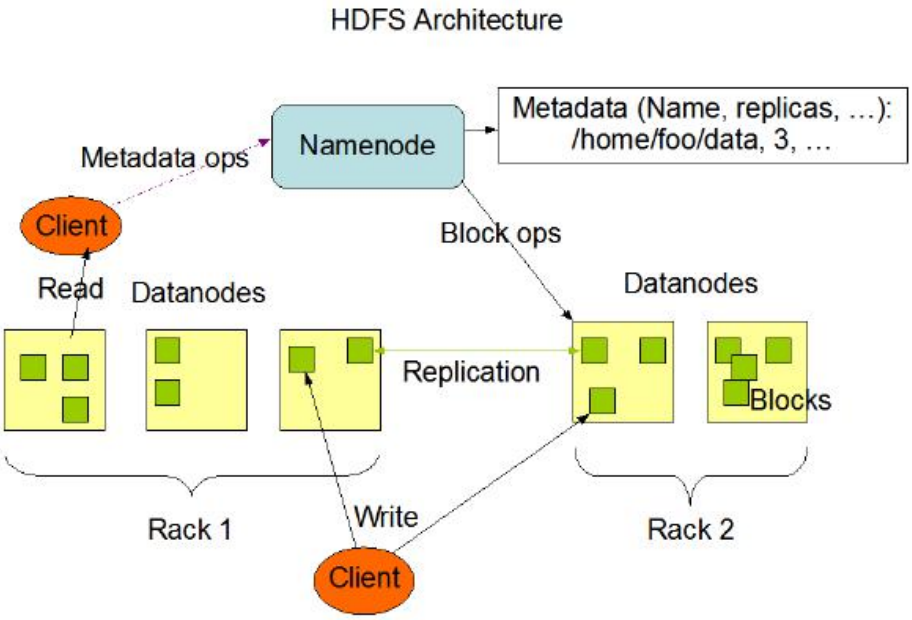
减少元数据的量

有利于顺序读写（在磁盘上数据顺序存放）

HDFS 数据分布设计




HDFS 体系结构




HDFS 可靠性

磁盘数据错误

心跳

重新分布

数据完整性：校验和

元数据磁盘故障：检查点

快照：用于回滚，尚未实现

负载均衡

加入一个新节点的步骤

配置新节点上的 hadoop 程序

在 Master 的 slaves 文件中加入新的 slave 节点

启动 slave 节点上的 DataNode，会自动去联系 NameNode，加入到集群中

Balancer 类用来做负载均衡

默认的均衡参数是 10%范围内

### 4.2.2 Hadoop

Hadoop VS. Google

技术架构的比较

数据结构化管理组件：Hbase→BigTable

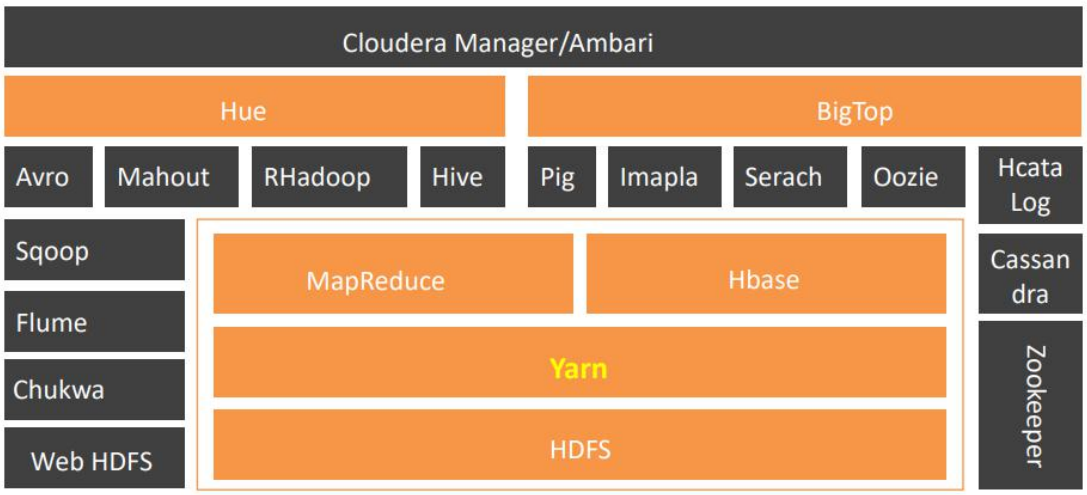
并行计算模型：MapReduce→MapReduce

分布式文件系统：HDFS→GFS

分布式锁服务：ZooKeeper →Chubby


Hadoop 2.0 相关项目

近几年工业界围绕 Hadoop 进行了大量的外围产品开发，下图描述了各个产品项目之间的层次关系。



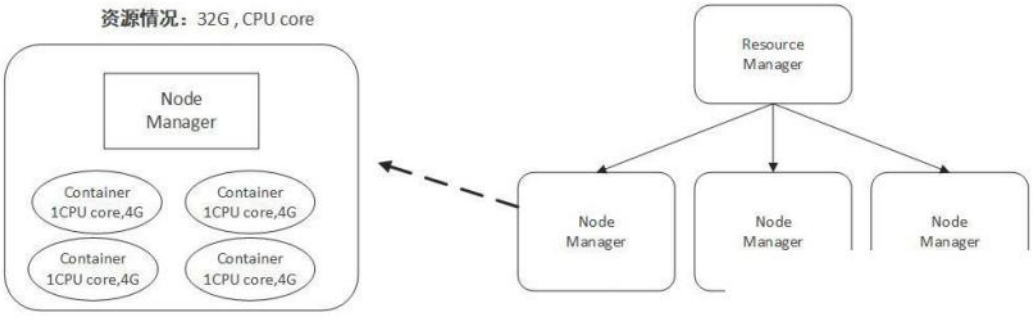

### 4.2.3 Yarn

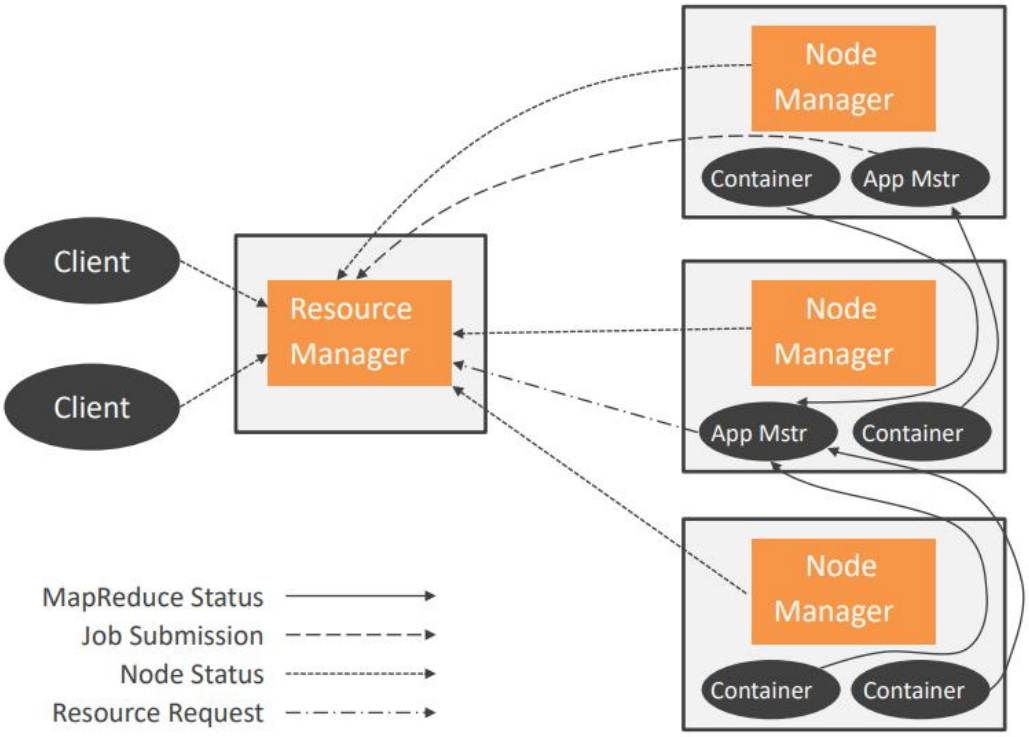
Yarn：大数据集群“分布式操作系统”


架构 Yarn 体系架构

将 MRv1 版 JobTracker 的两大功能——资源管理和任务管理


Yarn 依旧是 master/slave 结构；主进程 ResourceManager 是整个集群资源仲裁中心；从进程 NodeManager 管理本机资源






Yarn 工作流程

Client 向 Yarn 提交 Application（例如，MapReduce 作业）

ResourceManager 向 NodeManager 通信，为该 Application 分配第一个容器，并在该容器中运行应用程序对应的 ApplicationMaster

ApplicationMaster 启动以后，基于对作业拆分的 task 情况，向 ResourceManager 申请要运行程序的容器

申请到容器后，ApplicationMaster 通知对应的 NodeManager，将任务分发到对应容器中运行

容器中运行的任务会向 ApplicationMaster 发送心跳，汇报自身情况；当程序运行完成后，ApplicationMaster 再向 ResourceManager 注销并释放容器资源

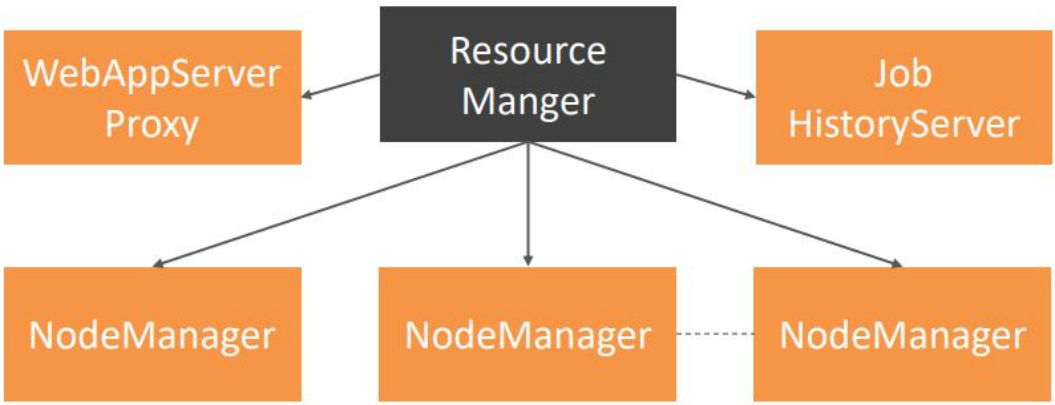
Yarn 体系架构

从 Yarn 架构和 Yarn 任务执行过程能看出 Yarn 具有巨大优势：

减轻了 ResourceManager 的资源消耗，ApplicationMaster 可分布于集群中任意一台机器，设计更加优美

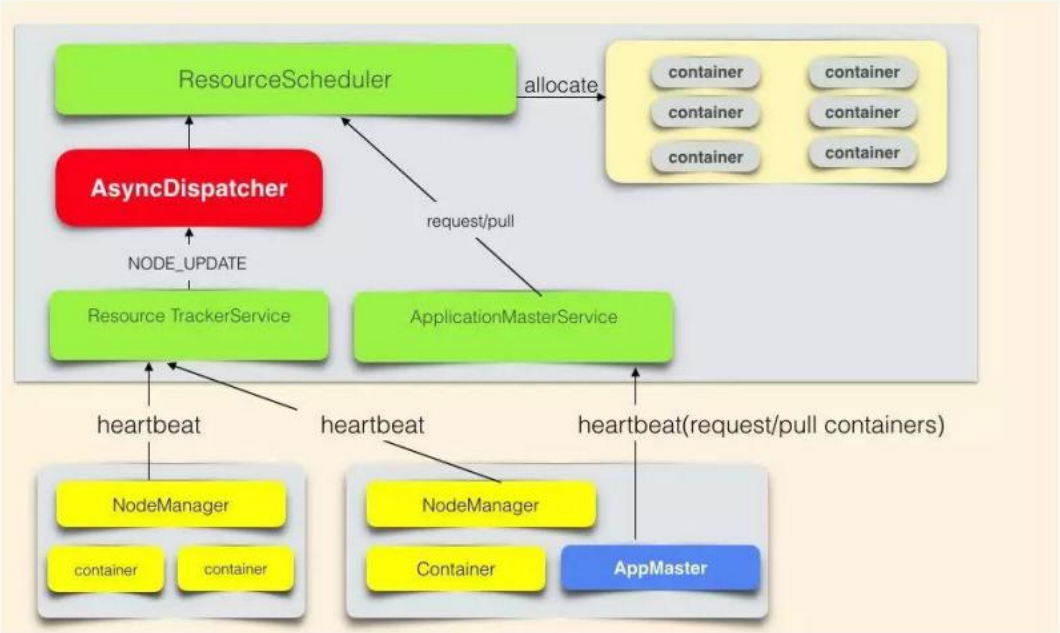
Yarn 能让更多计算框架可以接入到 HDFS 中，而不单单是 MapReduce；Yarn 的包容，使得其他计算框架能专注于计算性能的提升

除了 ResourceManager 和 NodeManager 两个实体外，Yarn 还包括 WebAppProxyServer 和 JobHistoryServer 两个实体




Yarn典型拓扑

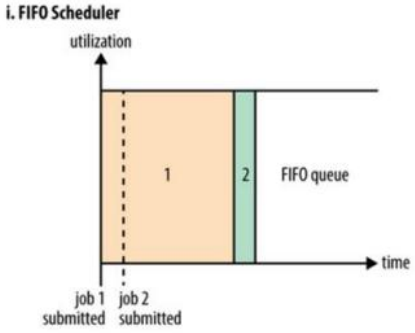

Yarn 调度框架



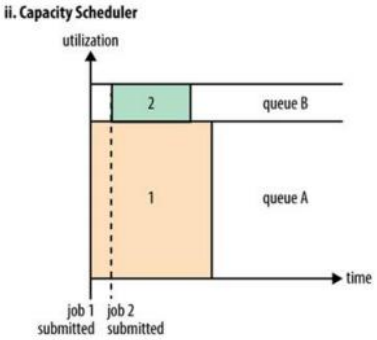

NodeManager 心跳：监控集群可用资源

AppMaster 心跳：告知 YARN 资源需求（List[ResourceRequest]），并取回上次心跳之后，调度器已经分配好的资源（List[Container]）

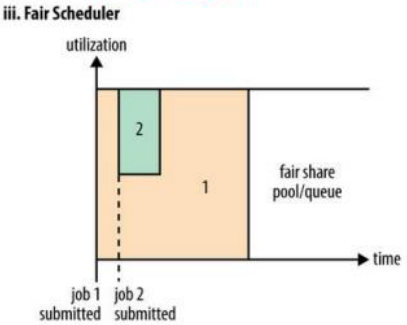
Yarn 常见调度策略




FIFO调度器




容量调度器




公平调度器

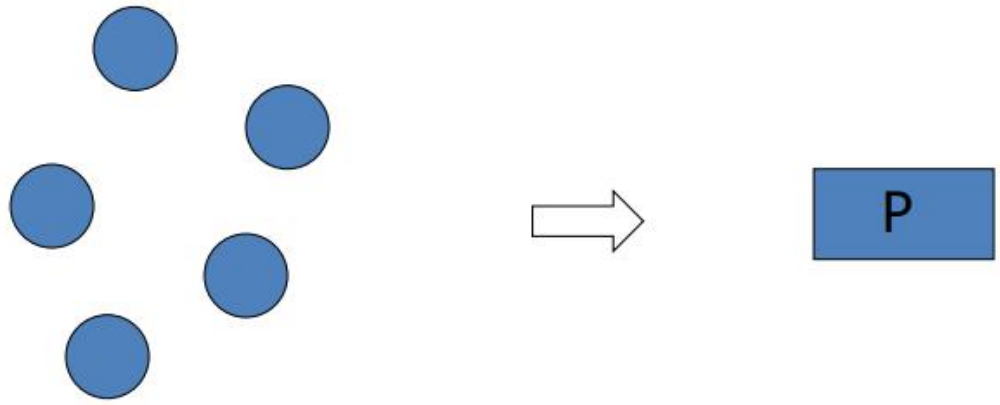
延迟调度器

*Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling.* **EuroSys'10**


5. 调度模型及算法

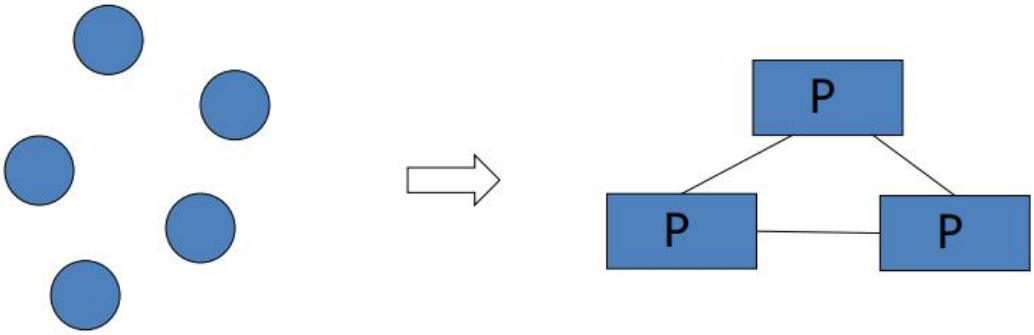
5.1 常见调度问题

（1）单处理器上的任务调度



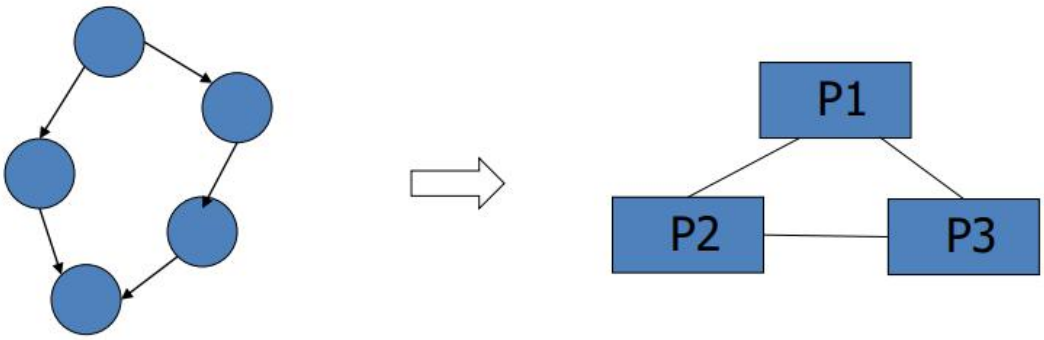

给定：发布时间、每项任务的工作量或截止日期
确定每项任务的执行时间
目标：任务的平均完成时间或满足截止时间

（2）多处理器上的任务调度



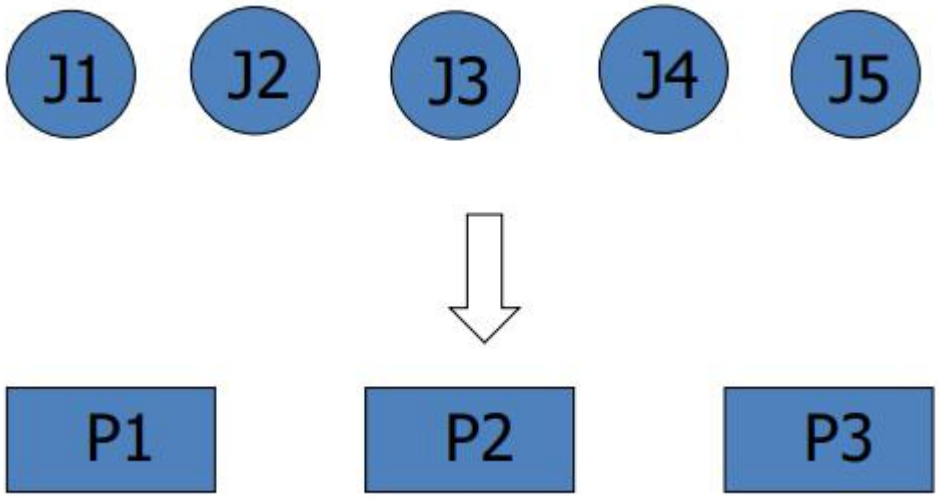

给定：发布时间、每项任务的工作量
确定每项任务的执行位置和时间
目标：最大完成时间（make-span）……

（3）异构处理器上的 DAG 调度




给定：处理每个处理器上每个任务的时间、边上的通信时间
确定每个任务执行的位置和时间
目标：最大完成时间（make-span）……

（4）作业车间调度（Job Shop Problem, JSP）




给定：每个处理器上每个作业的处理时间
约束：每个作业都由每个处理器执行一次
目标：最大完成时间（make-span）……

参考：作业车间调度问题描述
https://blog.csdn.net/weixin_36909758/article/details/88037578

5.2 调度方法

在线调度：

以机器为中心的方法

在机器空闲时触发计划

对于每台空闲机器，根据某些策略选择任务，例如，

先来先服务（FCFS），

最短作业优先（SJF），

最早的截止时间优先（EDF），

等待时间最长的工作优先……

以任务为中心的方法

在新任务到达时触发计划

对于每个计划任务，根据某些策略选择机器，例如，最早的完成时间……

5.3 列调度（List Scheduling）算法

（1）任务选择

通过为每个任务分配优先级来构建有序的任务列表，并按优先级顺序选择任务。

（2）处理器选择

将每个所选任务安排到处理器，以最小化预定义的代价函数（cost function）。

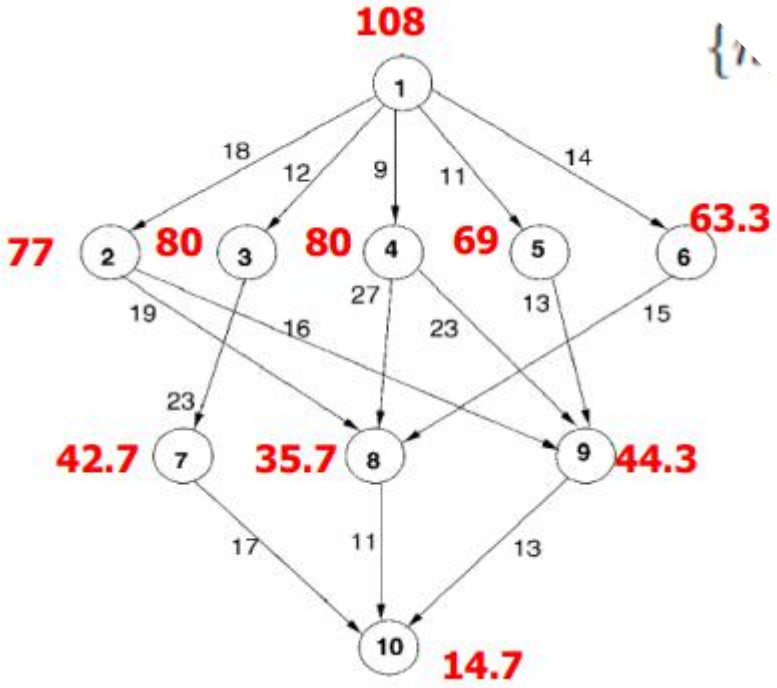
（3）重复步骤 1 和步骤 2，直到安排了所有任务。

例子：

步骤 1：任务选择——降序（upward rank）

计算出节点 i 到出口的最长路径的长度，长度较大的排在前面

（计算时间按各个核心计算时间的平均值计算，假设都需要传输数据）




$$\{n_1, n_3, n_4, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}\}$$

计算开销

| 任务 | P1 | P2 | P3 |
|---|---|---|---|
| 1 | 14 | 16 | 9 |
| 2 | 13 | 19 | 18 |
| 3 | 11 | 13 | 19 |
| 4 | 13 | 8 | 17 |
| 5 | 12 | 13 | 10 |
| 6 | 13 | 16 | 9 |
| 7 | 7 | 15 | 11 |
| 8 | 5 | 11 | 14 |
| 9 | 18 | 12 | 20 |
| 10 | 21 | 7 | 16 |

步骤 2：处理器选择——最早完成时间（Earliest Finish Time）

【若某一任务及其依赖的任务在同一核心，数据传输的时间为 0】

$$EST(n_i, p_j) = \max\{avail[j], \max_{n_m \in pred(n_i)}(AFT(n_m) + c_{m,i})\}$$

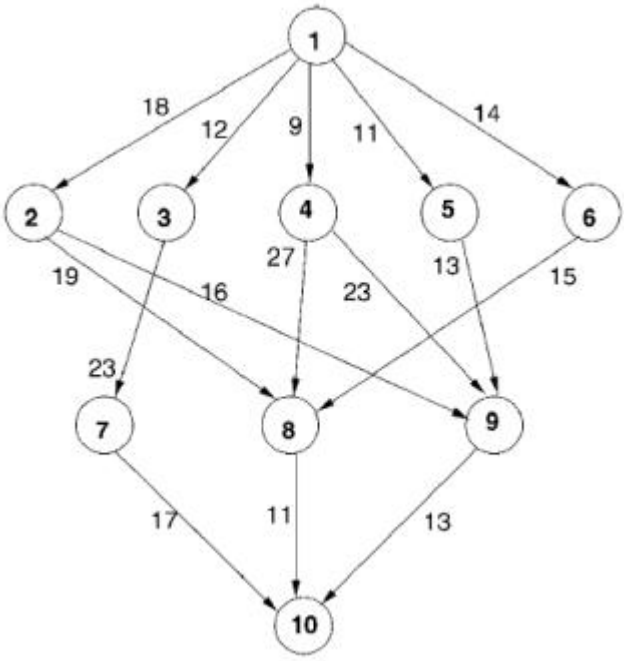
$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j)$$

$$EST(n_{entry}, p_j) = 0$$

$$AFT(n_i) = \min_{\forall j} EFT(n_i, p_j)$$

对于每个任务，选择可以在最早的时间完成该任务的机器。

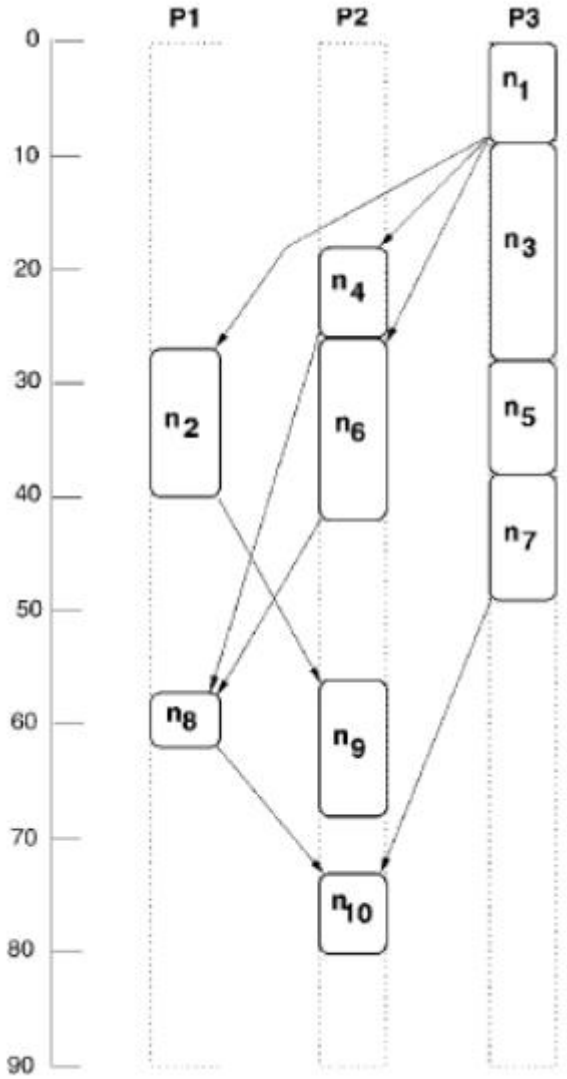
$$\{n_1, n_3, n_4, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}\}$$




| 任务 | P1 | P2 | P3 |
|---|---|---|---|
| 1 | 14 | 16 | 9 |
| 2 | 13 | 19 | 18 |
| 3 | 11 | 13 | 19 |
| 4 | 13 | 8 | 17 |
| 5 | 12 | 13 | 10 |
| 6 | 13 | 16 | 9 |
| 7 | 7 | 15 | 11 |
| 8 | 5 | 11 | 14 |
| 9 | 18 | 12 | 20 |
| 10 | 21 | 7 | 16 |

EFT 表格

| 任务 | P1 | P2 | P3 |
|---|---|---|---|
| 1 | 14 | 16 | 9 |
| 3 | 32 | 34 | 28 |
| 4 | 31 | 26 | 45 |
| 2 | 40 | 46 | 46 |
| 5 | 52 | 39 | 38 |
| 6 | …… | …… | …… |
| …… | | | |




6. 微服务架构及部署

6.1 微服务及无服务器计算的基本概念

A. 微服务的基本概念

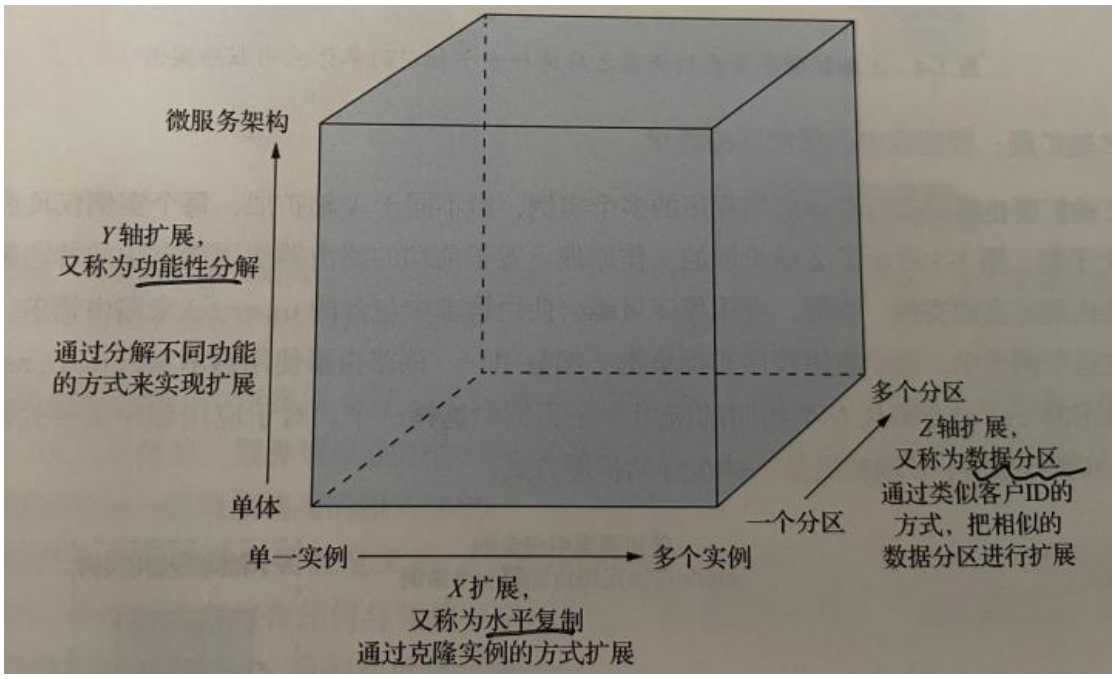
微服务定义

微服务架构是一种架构模式或者说是一种架构风格，它提倡将单一应用程序划分成一组小的服务，每个服务运行独立的进程中，服务之间互相协调、互相配合，为用户提供最终价值。

服务之间采用轻量级的通信机制互相沟通（通常是基于 HTTP 的 RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。

另外，应尽量避免统一的、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，选择合适的语言、工具对其进行构建，可以有一个非常轻量级的集中式管理来协调这些服务。可以使用不同的语言来编写服务，也可以使用不同的数据存储。

扩展立方体和服务




微服务与 SOA 异同

| | SOA | 微服务 |
|---|---|---|
| 服务间通信 | Enterprise Service Bus（ESB），采用重量级协议，如 SOAP 及其他 WS 标准 | 消息代理，采用轻量级协议，如 REST 或 gRPC |
| 数据管理 | 全局数据模型并共享数据库 | 每个服务都有自己的数据模型和数据库 |
| 典型服务的规模 | 较大的单体应用 | 较小的服务 |

微服务的优势

使大型的服务应用程序可以持续交付和持续部署
每个服务都相对较小并容易维护
服务可以独立部署
服务可以独立扩展
微服务架构可以实现团队的自治
更容易实验和采纳新的技术
更好的容错性

微服务的部署方式

虚拟机
容器
无服务器模式（Serverless）

B. 无服务器计算的基本概念

什么是 Serverless

无服务器架构：开发者实现的业务逻辑运行在无状态的计算容器中，它由事件触发，

完全被第三方管理，其业务层面的状态存储在数据库和其他存储资源中

无需管理基础设施

按用付费

事件驱动

自动化构建部署

无服务器计算=FaaS+BaaS

Serverless 的使用场景

异步、并发、易于并行化成独立工作单元的工作负载

低频请求，但具有较大不可预测的扩容需求的工作负载

无状态，短期运行，对冷启动延迟不敏感的工作负载

业务需求变化迅速，要求快速开发实现的场景

Knative

Knative 是谷歌发起的开源项目，致力于将 Serverless 标准化。它基于 K8S 平台，用于部署和管理现代无服务器工作负载。

## 7. 移动云计算

### 7.1 定义

基于云计算的定义，移动云计算是指通过移动网络以按需、易扩展的方式获得所需的基础设施、平台、软件（或应用）等的一种 IT 资源或（信息）服务的交付与使用模式。移动云计算是云计算技术在移动互联网中的应用。

### 7.2 三个视角

（1）移动云计算——AI 视角

AI 应用的重要计算基础设施智能交通系统

犯罪追捕

Face++

（2）移动云计算——云计算视角

云计算（Cloud Computing）：通过网络向客户提供动态可配置资源的共享池作为服务

移动云计算（Mobile Cloud Computing）：通过移动设备向用户提供云计算服务，例如智能手机和汽车

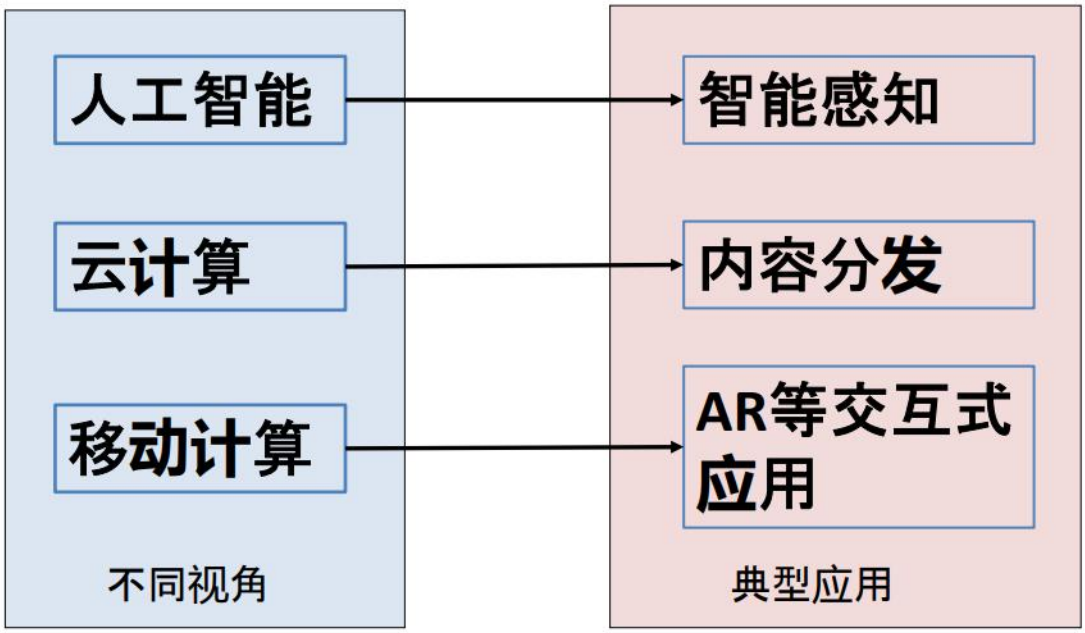
（3）移动云计算——移动计算视角

移动设备计算资源、电池受限

利用云计算资源节约（解决？）移动设备资源受限问题

Computation Offloading——将 Mobile App.的复杂计算迁移到云上

增强现实（AR）+计算迁移（computation offloading）——选取计算复杂部分迁移到云上完成




7.3 性能建模与优化

性能建模

系统性能指标是什么？响应时间、吞吐率

用户响应时间

本地计算时间

数据传输（等待）时间

云上计算（等待）时间

影响因素

本地设备计算能力

网络带宽、延迟、拥塞

云处理能力、负载

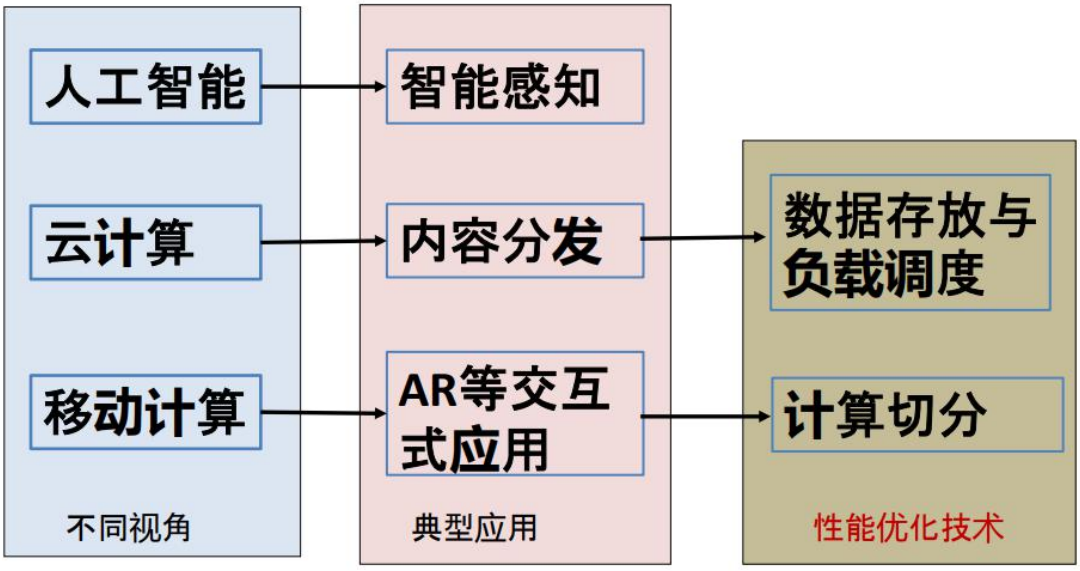
计算模块部署

性能建模方法

系统分析：通过系统架构和模型，分析和推导影响性能各因素与性能指标之间的直接关系

数据挖掘：测量系统在不同因素下性能，基于测量数据，采用数据挖掘方法建立性能模型

移动云性能优化技术




计算切分（Computation Partitioning）

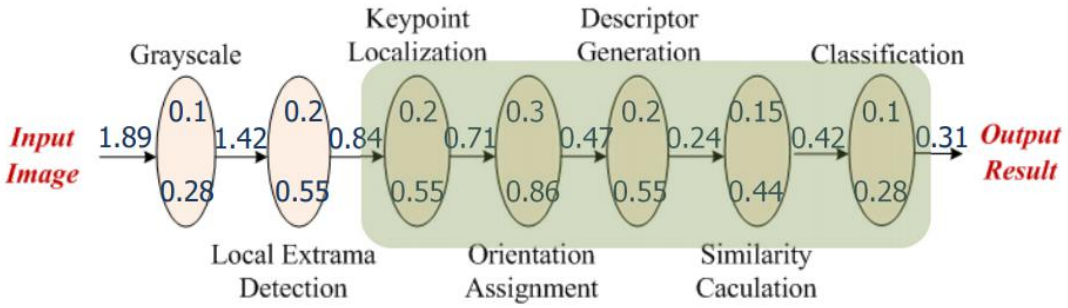
数据存放和负载分配【Data (Service) Placement and Load Dispatching】

7.4 计算切分

计算分区将应用程序的工作负载分解为较小的工作负载，并决定哪些部分在本地执行，哪些部分转交到云端

| | 性能指标 | 重要因素 |
|---|---|---|
| 工作流 | 最大完成时间（make-span） | 移动设备上的计算资源：CPU、内存、工作负载、电池<br>网络：连接度（connectivity）、传输率（transmission rate）<br>云资源：服务器容量和工作负载<br>移动云计算（MCC）系统的计算安排（placement） |
| 数据流 | 吞吐量（throughput） | |

例子

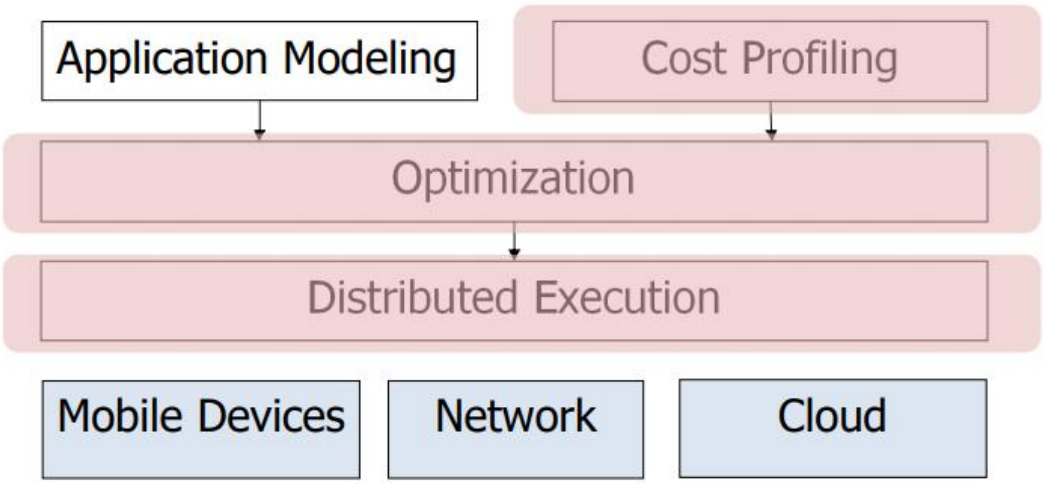



最优划分：0.28+0.55+0.84+0.2+0.3+0.2+0.15+0.1+0.31=2.93

本地执行：0.28+0.55+0.55+0.86+0.55+0.44+0.28=3.51

远程执行：1.89+0.1+0.2+0.2+0.3+0.2+0.15+0.1+0.31=3.45

技术难点




应用模型

如何表示应用的结构：3 种主要方法

过程调用（Procedure calls）

应用：一组过程

以功能为中心的、同步的

服务调用（Service invocation）

应用：一个服务调用图

以消息为中心的、异步的

数据流（Dataflow）

应用：一个有向无环图（directed acyclic graph）

边代表数据流；节点是数据的处理函数

开销建模

估计应用中每个组件的执行开销并权衡上传的开销和潜在的收益

运行开销能被以下指标中的一个或多个的加权度量：

执行时间（本地的和远程的）

能耗

网络中的数据传输量

概述（profiling）是一种手机并估计应用组件开销的方法

基于预测的概述（Prediction-based profiling）

基于模型的概述（Model-based profiling）

基于预测的概述

记录每个应用实例的开销，并根据历史记录预测当前开销

精确度依赖过去记录的新鲜程度

低开销概述

例如，Odessy、ThinkAir

基于模型的概述

通过建模和度量设备和网络参数来估计开销，例如，CloneCloud

需要应用的离线概述以及设备和网络参数的在线度量
应用信息（静态）——计算负载、数据传输量
设备信息——计算容量、CPU 负载、内存使用率等等
网络信息——带宽、RTT/延迟
高概述成本
适用于动态环境

优化
获得最优的计算切分——能在线或离线解决
在线优化解决每次应用执行的运行时优化问题。
离线优化在离线阶段计算不同设备和网络状态下最优的划分
为给定的设备和网络状态指标搜索最匹配的划分
避免解出优化的开销，但需要大量的离线测试例子
优化能在移动端或云端解决

分布式执行
在移动设备和云网络执行所划分的计算组件
3 个执行方法
客户机服务器交流法
虚拟机迁移
容器迁移，例如，docker 和 kubernetes

其他参考资料
[1] 老师的课件
[2] 课本
[3] 往届的复习总结