大数据的信息基础设施

计算虚拟化

陈一帅

yschen@bjtu.edu.cn

北京交通大学电子信息工程学院

内容

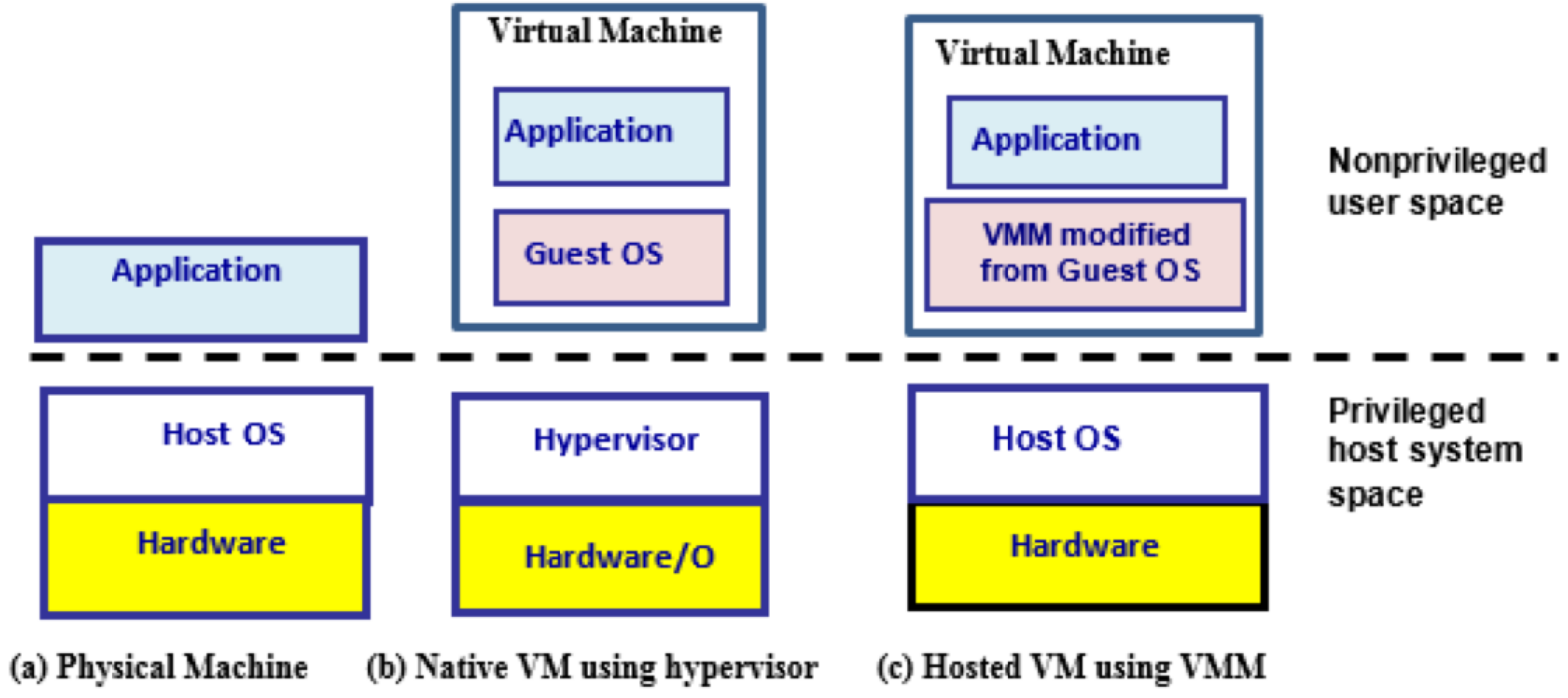
- 虚拟化
- 虚拟机
- 容器
- 微服务
- 微服务实例

虚拟化

- 程序
  - 一些指令
- 操作系统（OS）
  - 允许多个用户程序同时运行，分享资源
  - 进行状态管理、程序上下文切换、I/O 访问控制
- 程序不能进行状态管理、I/O 访问指令，因为会接入其它程序的状态
  - 要通过操作系统执行这些指令
- 提供看起来真实，但实际上是在软件中处理的这些指令的过程，被称为虚拟化

虚拟机

- Virtual Machine（VM）
- 在虚拟机管理程序上运行的 OS
- 是一个完整机器的软件映像，可以将其加载到服务器上并像其他程序一样运行

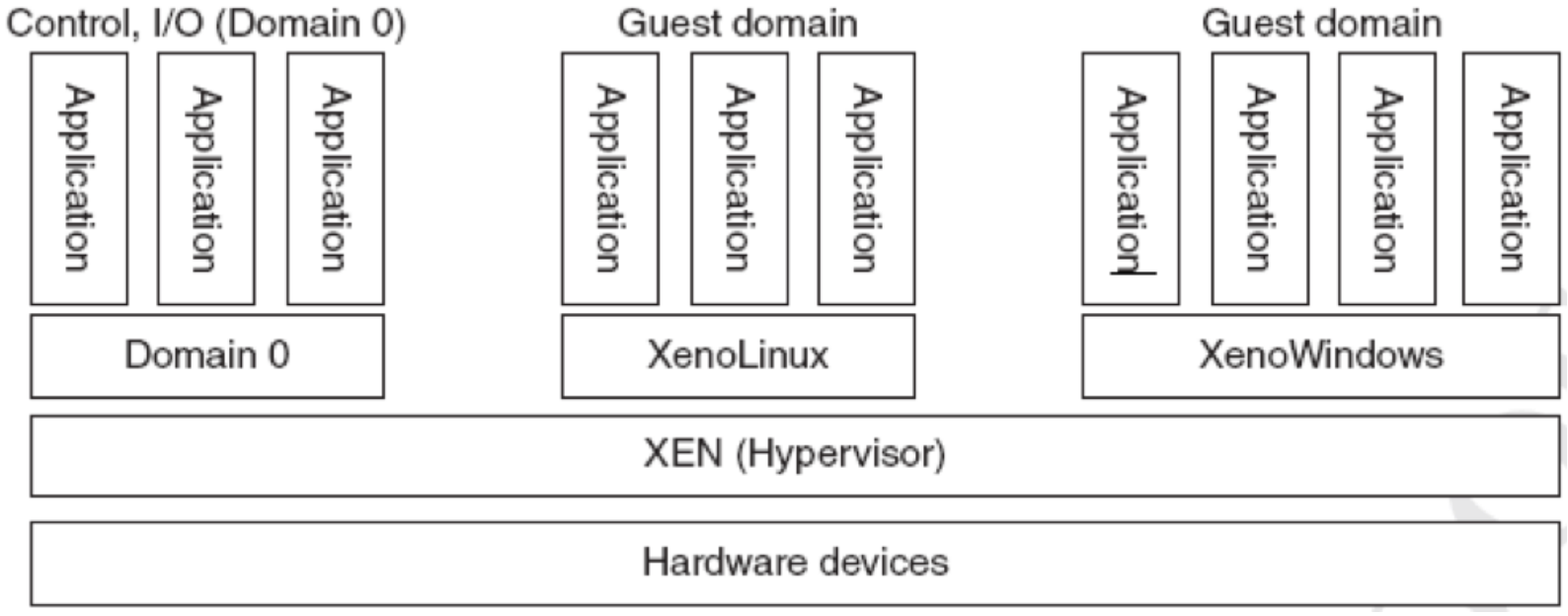


两种虚拟机

- Native 虚拟机
  - 基于虚拟机管理程序 Hypervisor
  - Hypervisor 在 OS 之下，允许多个 OS 同时运行，分享同一个硬件资源
  - 管理和分配服务器资源
  - 如 Citrix Xen，Microsoft Hyper-V，VMWare ESXi
- Hosted 虚拟机
  - 基于 VMM
  - 虚拟机管理程序作为进程运行在主机操作系统上
  - 如 VirtualBox 和 KVM

例：XEN Hypervisor

- Hypervisor 在 OS 之下
- 允许多个 OS 同时运行，分享同一个硬件资源



虚拟机带来的好处

- 云可以选择使用哪个服务器来运行请求的 VM 实例
- 如需要，可在一台服务器上同时运行多个 VM
- 同一服务器上的不同 VM 中运行的用户应用程序之间彼此之间几乎完全没有察觉
- 可以监视每个 VM 的运行状况，记录事件并重启 VM
- 一个 VM 实例崩溃，不会使整个服务器崩溃

虚拟化的好处

- 最小化硬件成本（CapEx）
  - 一台物理硬件上的多个虚拟服务器
- 能够轻松将 VM 移至其他数据中心
  - 提供灾难恢复，硬件维护
  - 跟随太阳（活跃的用户）或跟随月亮（便宜）
- 合并空闲的工作负载，释放未使用的物理资源
  - 用户流量是突发性的和异步的，提高设备利用率，省电
- 更轻松的自动化（降低 OpEx）
  - 简化的硬件和软件供应/管理
- 可扩展、灵活，支持多种操作系统

灵活的计费

- 例：亚马逊 AWS 的三种虚拟机实例的计费方式
  - On-demand instances
  - Reserved instances：1~3 年
  - Spot instances：竞价，闲时用，区域有关
- 弹性配置器
  - Ryan Chard 弹性配置器，成本降低多达 95%

虚拟机的问题

- 成本问题
  - 每个 VM 都需要一个操作系统（OS）
  - 每个操作系统都需要许可证 ⇒ CapEx
  - 每个操作系统都有自己的计算和存储开销
  - 需要维护，更新 ⇒ OpEx
  - VM 成本 = 添加了 CapEx + OpEx
- 性能问题
  - 启动操作系统，需要额外的 Overhead
  - VM 是完整的 OS 实例，因此可能需要几分钟才能启动

内容

- 虚拟化
- 虚拟机
- 容器
- 微服务
- 微服务实例

容器

- 另一种虚拟化方法
- 具有虚拟机的所有良好特性，同时更轻量级
- 在同一操作系统上运行许多应用
  - 这些应用共享操作系统及其开销
- 但不会互相干扰
  - 未经明确许可就无法访问彼此的资源

容器的隔离

- 容器将应用程序及其所有库依赖关系和数据打包到一个易于管理的单元中
- 每个容器有自己的网络，主机名，域名，进程，用户，文件系统，IPC，互不干扰

Docker

- 允许在包括所有应用程序依赖项的容器中运行应用程序
- 该应用程序可以看到一个完整的私有进程空间，文件系统和网络接口
  - 例如，容器中的“显示进程”（在 Linux 上为 ps）命令将仅显示容器中的进程
- 与同一主机操作系统上其他容器中的应用程序隔离

容器的轻量级

- 在操作系统上提供隔离，轻量级
- 比 VM 轻
  - 是一个应用
  - 应用和它的依赖库、配置、数据的打包
- 启动快
  - 可以在几秒钟内将应用程序初始化并运行
- 比 Process 重
  - 一个容器内能运行多个 Process

容器的低成本

- 在一个操作系统上运行多个容器
- 所有容器共享操作系统
- CapEx 和 OpEx

容器的优点

- 具有虚拟机的所有良好特性
- 可移植
  - 容器从 Image 中生成，也可以另存为 Image
  - 同一个 Image 可以在个人计算机，数据中心或云中运行
  - 可以停下来、保存并移动到另一台计算机上或以后运行
- 可扩展
  - 可以在同一台计算机或不同计算机上运行多个副本
- 灵活
  - 可根据容器构建时的设计来限制或不限制操作系统资源

容器的原理

- Linux 内核功能，绑定和容纳进程的资源调用
  - 基于 namespace 实现名称空间隔离（isolation）
  - 基于控制组（cgroup）实现资源限制

容器的分层实现

- Image 是逐层构建的，每个 Image 都有很多层
- Docker 首先在称为 Alpine 的基础 Linux 内核上运行
  - 其他操作系统功能在该基础之上分层
- 例如：
  - 先安装 Ubuntu OS，一层
  - 然后安装 Python 软件包，一层
  - 安装 Python 的安全补丁，一层

容器的分层实现

- 层是容器高性能和可移植的关键
- 层可以在许多容器之间共享
  - 安装迅速
- 容器实例可以与其他容器实例共享库
  - 启动迅速
  - 相对于 VM，也可以在一台主机上运行更多容器

Docker

- 使用 Google 的 Go 编程语言编写
- 管理容器，提供覆盖网络和安全性
- 提供容器之间的隔离，帮助他们共享操作系统
- 最初由 Docker.com 开发，现已开源
- 可从 Docker.com 下载用于 Linux，Windows 和 Mac
- 两个版本：
  - 社区版（CE）：免费进行实验
  - 企业版（EE）：用于带付费支持的部署
- Docker Swarm 和 Kubernetes 管理大量容器

Image 存储库（Registries）

- Image 存储在 Registries 中
  - 每个 Image 都有几个标签，例如 v2，最新，...
  - 每个 Image 都通过其 256 位哈希值进行标识
- 主机上有本地存储库
- 网络存储库
  - Docker Hub 存储库，经过 Docker 审查的 Image
  - 非官方存储库，未经审核的 Image，谨慎使用
- 启动 Image 时，在本地存储库中找不到的任何组件，会从指定位置下载

Build 自己的容器

- 创建脚本，标识所需的库、源文件和数据
- 在电脑上运行脚本以测试容器
- 将容器上传到网络存储库，从那里可以将其下载到任何云

创建脚本

- 创建 Dockerfile，描述应用程序、依赖包、以及如何运行的
- 例：NodeJS Web服务器

```
FROM Alpine
LABEL maintainer=“xx@gmail.com”
RUN apk add –update nodejs nodejs –npm
COPY . /src
WORKDIR /src
RUN nmp install
EXPOSE 8080
ENTRYPOINT [“node”, “./app.js”]
```

脚本说明

- 从 Alpine Linux 开始
  - 注：也可以从其它容器开始
- 谁写的这个容器
- 使用 apk 安装 nodejs
- 复制应用程序文件
- 设置工作目录
- 安装应用程序依赖项
- 打开端口 8080
- 运行主要应用 app.js

生成容器

- 基于 Dockerfile，使用 docker build 命令构建容器
- 下载
  - 首次运行 docker build 时，会下载所有组件，需要时间
  - 组件下载后，缓存在本地计算机上
- 安装
  - 所有安装（如 pip）均已运行并分层到文件系统中，甚至对 Python 代码也进行了分析以检查错误
  - 由于这种预安装，在运行容器时，一切都已经存在，所以运行起来特别快

Docker Hub

- https://hub.docker.com/
- 公共资源，可以在其中存储你的容器，搜索和下载数百个公共容器
- 创建一个免费的 Docker 帐户，并将容器保存到 Docker Hub

```
docker push yourname/bottlesamp
```

- 然后就可以将容器下载到云中并运行

运行容器

```
Docker run -d -p 8000:8000 yourname/bottlesamp
```

- 下载
  - 首次运行时，会下载所有组件，需要时间
  - 组件下载后，缓存在本地计算机上
- 其它命令
  - ls, exec, stop, start, rm, inspect

Docker Mount 主机文件系统

- 在容器操作系统中挂载主机目录
  - 容器可以与主机共享数据
  - 多个容器也可以共享该目录，进行通信
- 例
  - 启动一个 Linux Ubuntu 容器，把 Mac 电脑的/tmp 目录挂载为容器里的/localtmp 目录

```
docker run -it -v /tmp:/localtmp ubuntu
```

- 注
  - 如果在 Windows 10 上运行，需要访问 Docker 设置，提供 Docker 权限以查看和修改驱动器 C

Docker 安全

- 使用了 Linux 中所有内置的安全机制，并且更多
  - 加密节点 ID
  - 相互认证
  - 加密集群存储
  - 加密网络流量
  - 签名镜像
  - 安全扫描检测漏洞

内容

- 虚拟化
- 虚拟机
- 容器
- 微服务
- 微服务实例

微服务

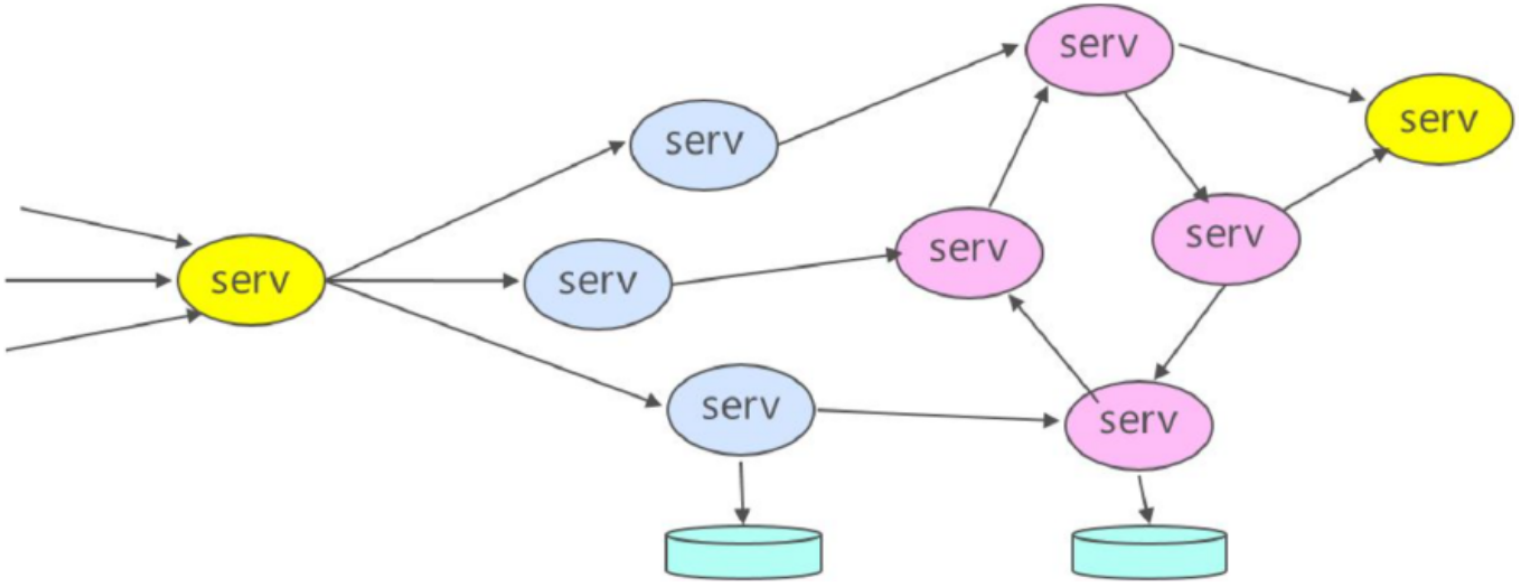
- 在一个容器中运行 Web 服务器
- 在另一个容器中运行数据库服务器
- 两个容器可以彼此发现并通信
- 微服务

微服务

- 消息 + 参与者
  - 大型云应用程序的主要设计范式
  - 计算由许多进行消息通信的参与者执行
  - 每个参与者都有自己的内部状态
  - 收到消息后开始行动
  - 根据消息，更改内部状态，向其他参与者发送消息
  - 参与者可实现为功能单一的 Web 服务 （Unix 设计原则）
- 如果服务无状态，就是微服务
  - 微服务通常被实现为容器实例

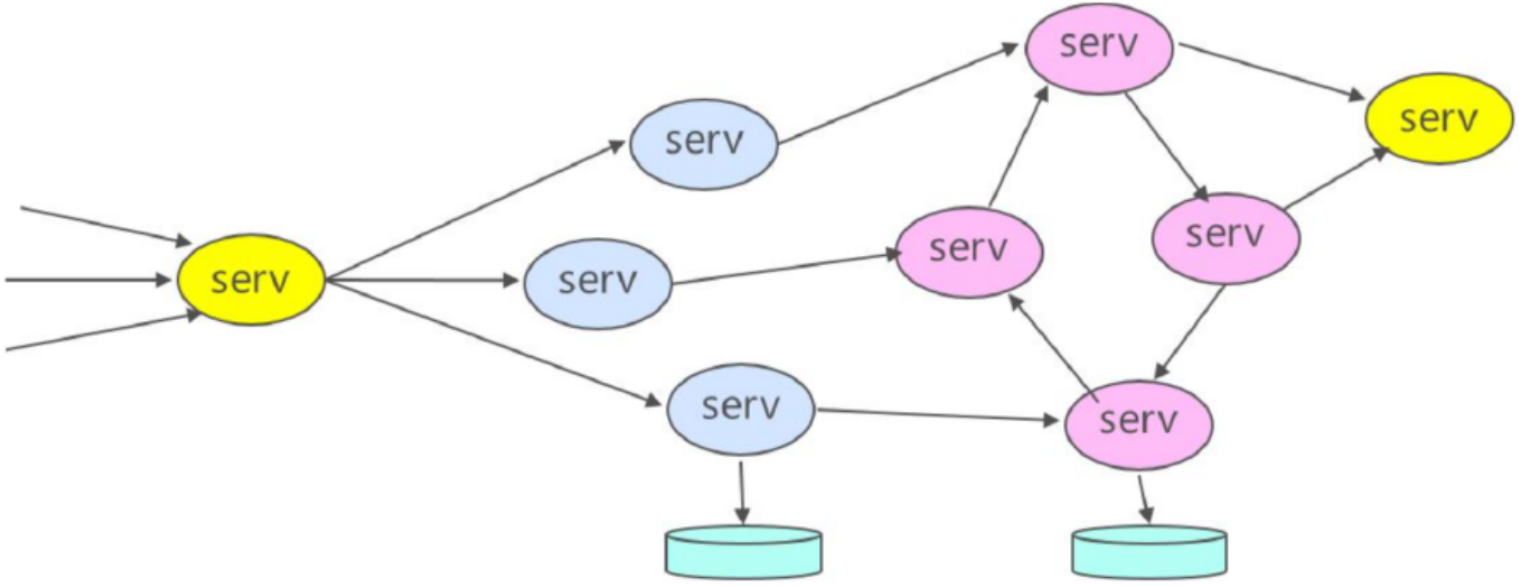
微服务系统模型

- 微服务构成群（Swarm）
- 微服务通过网络进行异步通信



微服务适合大规模互联网应用

- 大型在线服务的需求
  - 需要支持数千个并发用户
  - 在维护和升级的同时，还必须保持每天 24 小时在线
- 微服务集群，可伸缩



微服务

- 每天 Billion 级请求，使系统可扩展性成为核心要求
- Unix 和 Linux 的设计哲学
  - 一个程序只作一个事情，做得很好
  - 和其它的程序合作
- 服务器集成式的软件应用被分解，变成微服务
  - 基于容器的微服务模型
  - 更好管理、迁移、扩展、资源调度
- 大数据是其中的核心模块

Scale

- 最初可以部署在单个 VM 实例上
- 随着业务增长，它可能需要扩展以在高峰时间使用 100 甚至 10,000 台服务器
- 然后在业务缓慢时进行缩减

微服务组件设计

- 将应用程序划分为小型、独立的微服务组件
- 每个微服务必须能独立于其他微服务进行管理
  - 复制，扩展，升级和部署
- 每个微服务仅具单一功能，在有限上下文中运行
  - 责任有限，对其他服务的依赖也有限
- 所有微服务都应能支持不断发生故障和恢复
- 尽可能重用现有可信服务，如数据库，缓存和目录

微服务组件间通信

- 组件之间通过简单，轻便的机制进行通信
- 通信机制
  - REST Web 服务调用
  - RPC 机制（例如 Google 的 Swift）
  - 高级消息队列协议（AMQP）

微服务的广泛应用

- 微服务理念已被广泛采用
  - Netflix，Google，Microsoft，Spotify 和 Amazon
- Web 服务器或移动应用程序的后端
  - 接受来自远程客户端的连接，并根据客户端请求执行一些计算并返回响应
- 远程传感控制系统
  - 当地球传感器检测到以明显方式发生的地面震动时，会发出地震警告
- 大数据系统

微服务的管理

- 大规模微服务应用，需要管理大量分布式通信服务
- 管理平台
  - 启动实例
  - 停止实例
  - 创建新版本
  - 扩展实例数量

管理工具

- Mesos
- Docker Swarm
  - Docker 容器调度程序
- Docker Kubernetes
  - Google 云容器调度程序

Mesosphere

- 集群管理
- 基于 Berkeley Mesos 系统
- 数据中心、分布式操作系统（DCOS）
- Web 界面
- 可以管理云中应用程序

Mesosphere

- 组成部分
  - Apache Mesos 分布式系统内核
  - Marathon 初始化系统，监视程序和服务，自动修复故障
  - Mesos-DNS 服务发现实用程序
  - ZooKeeper 高性能协调服务，管理已安装的 DCOS 服务
- 结构
  - 一个主节点，一个备份主节点，一组运行容器的工作线程

Docker Swarm

- 可管理数千个容器
- Swarm
  - 通过网络协作的一组节点
  - 一个服务可能在一个 Swarm 上运行
- 主机的两种模式
  - 单引擎模式：不参与
  - 群体模式：参与 Swarm

Swarm 管理

- 每个 Swarm 都有一些 Manager，将任务分派给 Worker
  - Manager 也是 Worker（即执行任务）
- Manager 们 选择一个 Leader，它真正跟踪 Swarm
  - 分配任务，重新分配失败的 Worker 的任务，...
- 其他 Manager 在 Leader 失败时重新选举 Leader
- 可以根据需要扩大或缩小服务

Kubernetes

- Google 开源的容器管理器
- Linux Foundation 中的 Cloud Native Computing Foundation（CNCF）项目
- 非常流行
- 类似 Swarm
- 将 Swarm 称为集群（Cluster）

基于 Pod 的 Kubernetes

- Pod 是一个或多个容器，以及这些容器共享的一组资源的集合
- Kubernetes 中调度的基本单位是 Pod
- 启动后，一个 pod 驻留在单个服务器或 VM 上
  - 因此共享相同的 IP 和端口空间，可以通过 localhost 之类的常规方式找到彼此
  - 还可以共享本地的存储卷

Kubernetes 命令

- 启动 Jupyter 并启动其端口 8888

```
kubectl run jupyter --image=jupyter/scipy-notebook
                    --port=8888
```

- 在外部显示，并连接负载均衡器

```
kubectl expose deployment jupyter
                  --type=LoadBalancer
```

- 获取服务说明

```
kubectl describe services jupyter
```

内容

- 虚拟化
- 虚拟机
- 容器
- 微服务
- 微服务实例

实战：云中创建微服务

- 以 Amazon EC2 Container Service (ECS) 为例
- 需要定义三个组件
  - 一个或多个 EC2 Clusters
  - 任务定义：指定容器信息，例如任务中包含多少个容器，将使用哪些资源，如何链接它们以及将使用哪些主机端口
  - Docker 镜像仓库：Amazon 托管的 Docker 映像存储库。将映像存储在此处可能会使它们在需要时更快地加载，但也可使用公共 Docker Hub 存储库

云中创建微服务

- 进入 Amazon ECS 控制台
- 创建集群（Cluster）
  - 命名，给出所需的 EC2 实例类型并提供实例数
- 提供任务定义
- 指定 Docker 镜像
- 创建服务，设定 8 个 微服务
  - 第一次创建服务时，需花大约一分钟从公共 Hub 下载 2 GB Docker 映像并将其加载到集群的 VM 中。在随后的运行中，由于该映像是本地映像，因此仅花费了几秒钟即可启动该服务

提供输入，保存输出

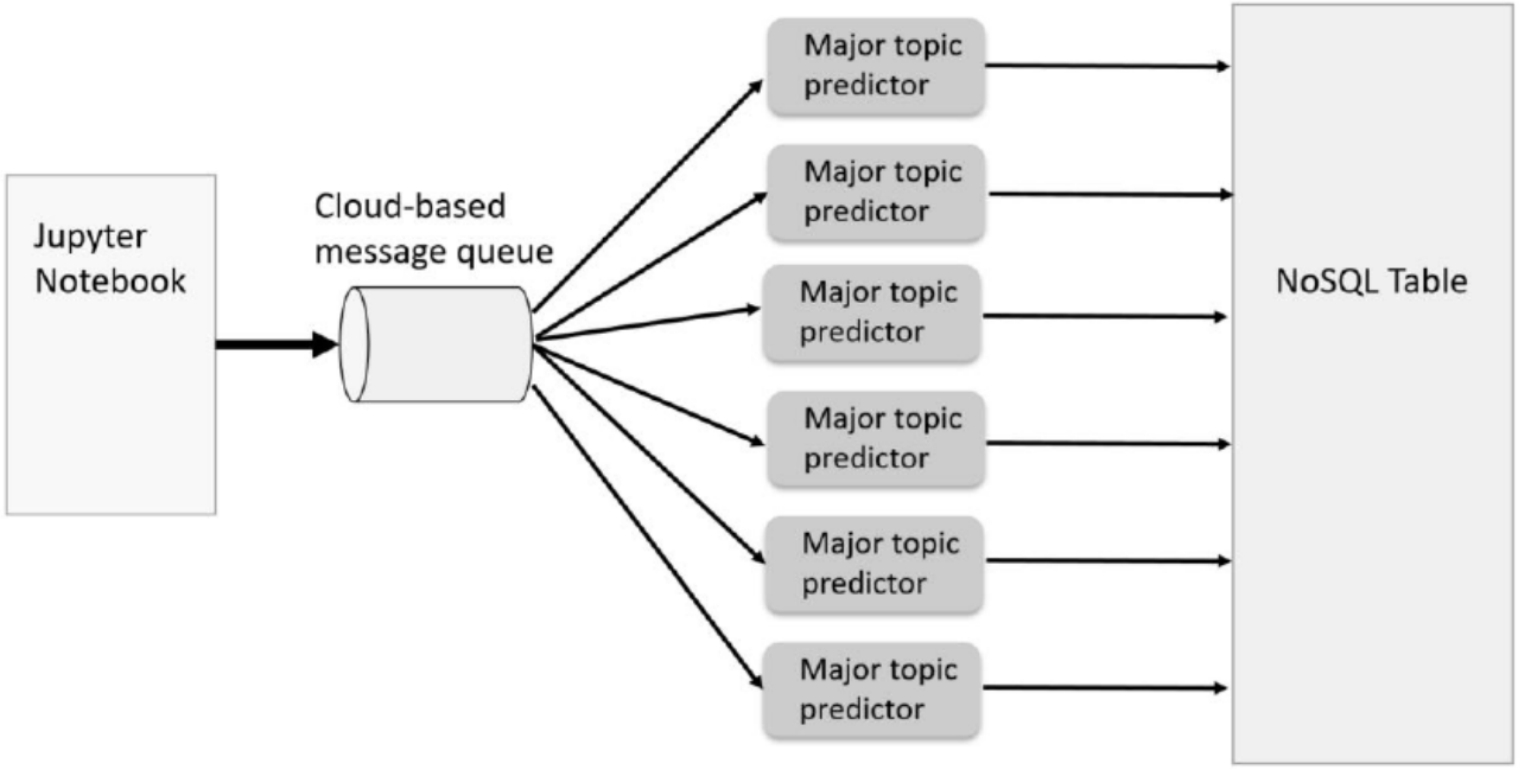
- 使用 Amazon SQS service 的队列服务，将消息送入集群
- 将结果存到 DynamoDB NoSQL DB 里

示例：文档分类

- 从各种 Feed 中提取文档
- 将文档送入云的消息队列中
- 使用一组微服务将这些文档分类为物理，生物学，数学，金融和计算机科学各类
- 使用第二组将这些文档再分类为子主题领域
- 结果推入 NoSQL 表

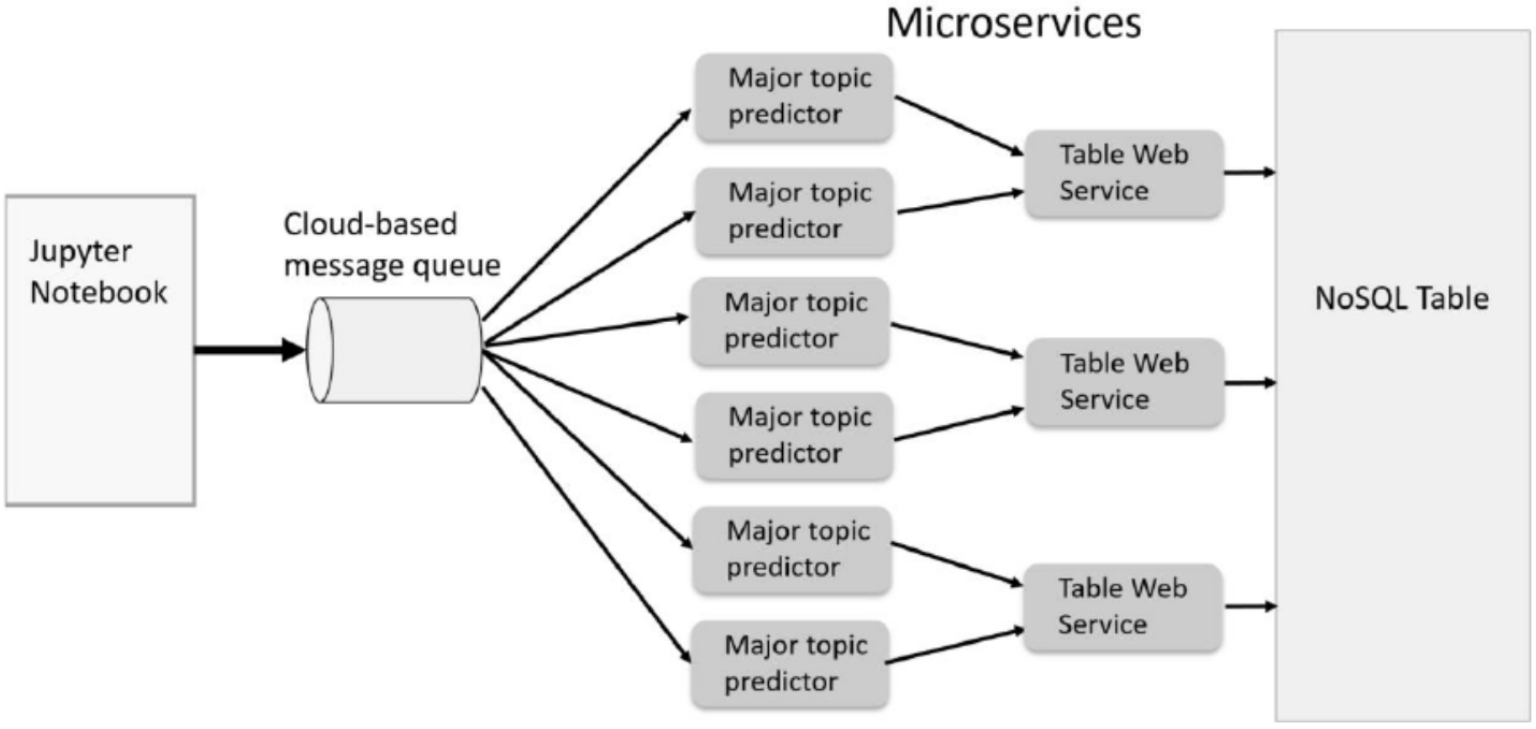
基础架构

- 一组微服务，完成所有分类



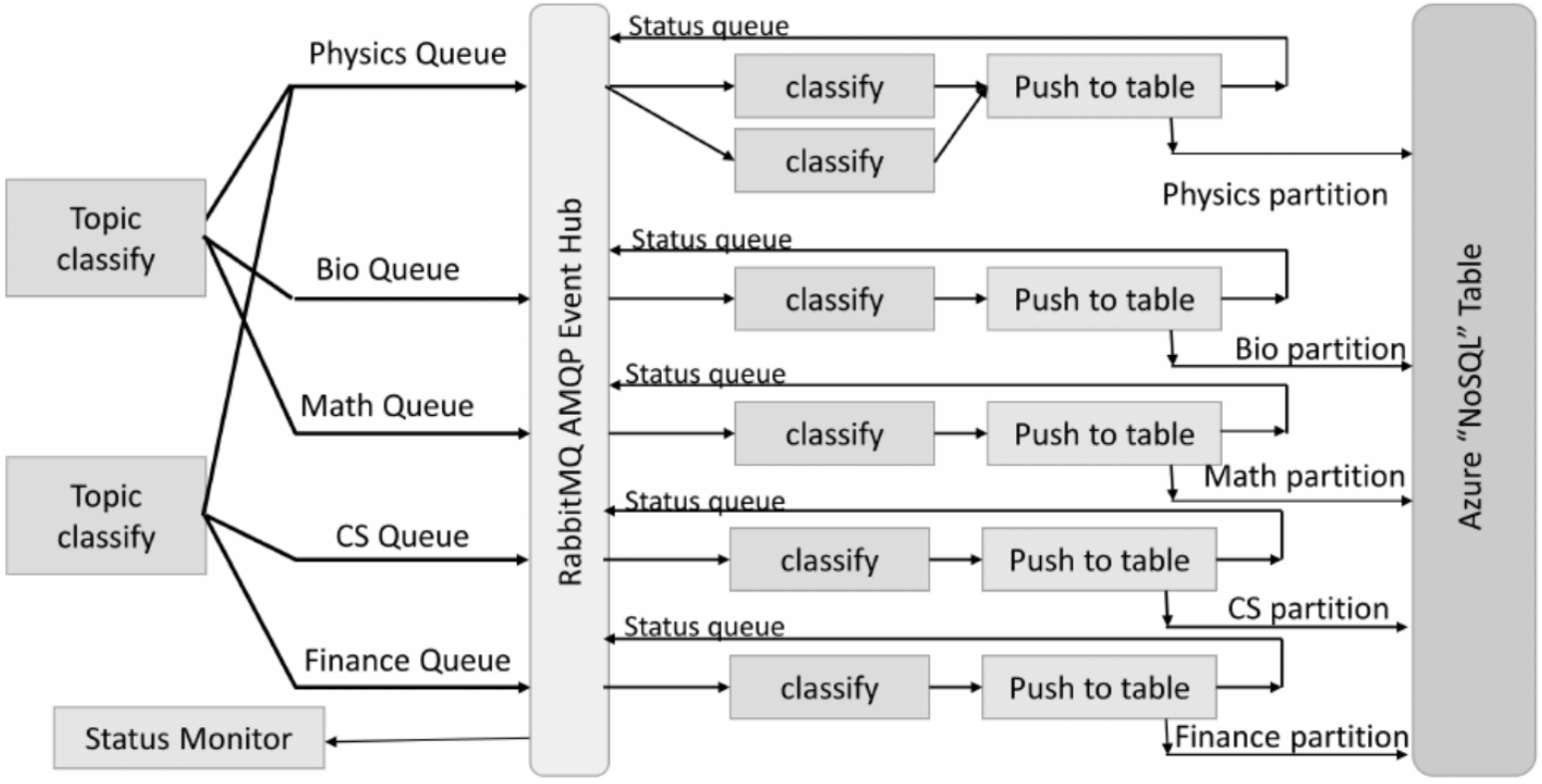
升级架构

- 两组微服务：主题选择器服务，表存储 Web 服务
- 根据工作量，配置不同的容器数



最终架构

- 完整主题领域分类器的示意图



小结

- 虚拟化
- 虚拟机
- 容器
  - Docker
- 微服务
  - Mesos，Kubernete
- 微服务实例
  - 文档分类

练习

- 匈牙利布达佩斯理工大学，微服务练习
- 起始代码和作业说明
  - https://github.com/ricsinaruto/simple-soa-projects
- docs目录
  - 2 Rest API 练习
  - 4 Microservices 练习
- Java 编程
- PostMan 测试
  - https://www.getpostman.com/