# 第二章 数据中心技术

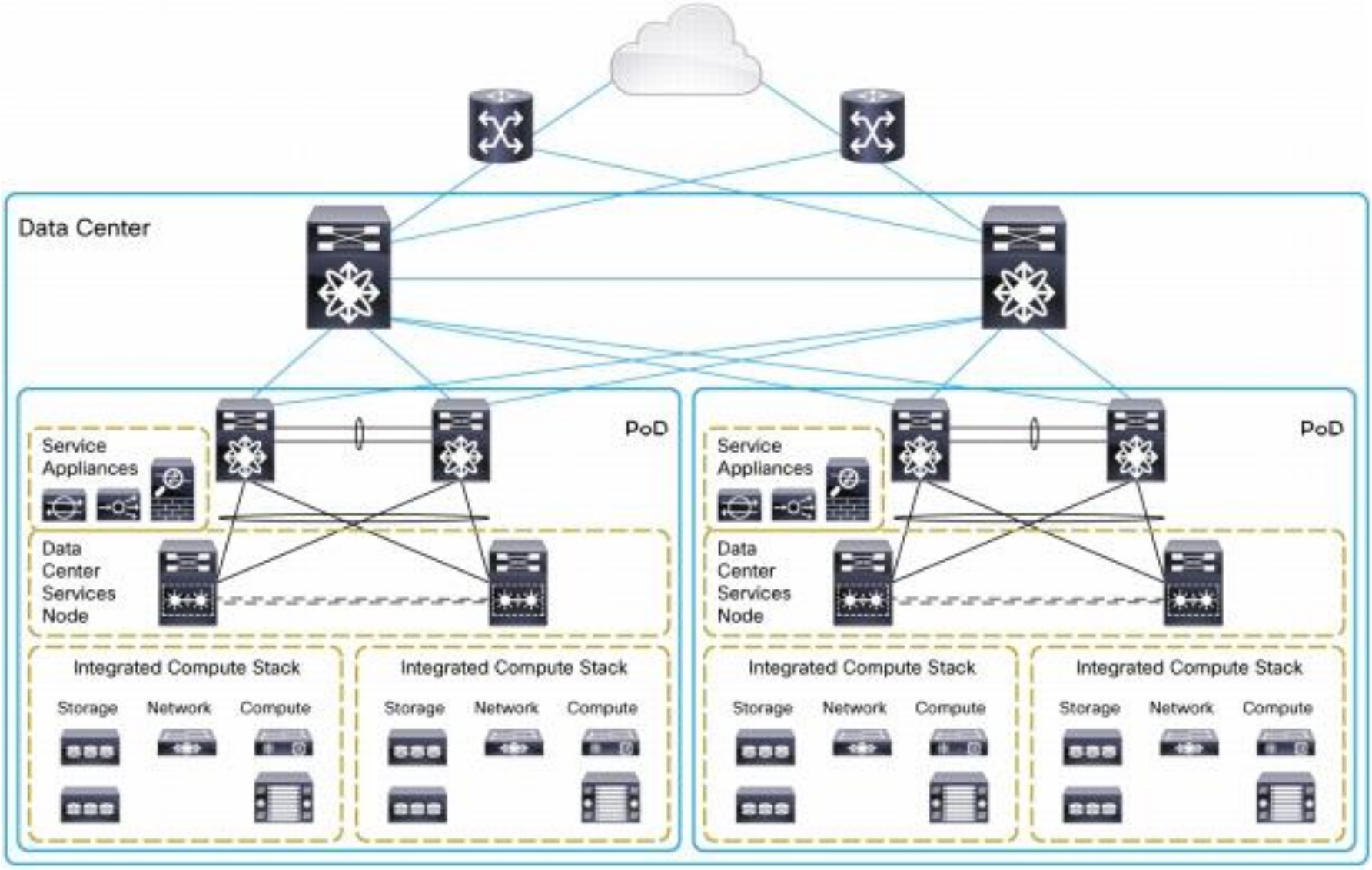
§2.1数据中心网络

§2.2资源池化技术

§2.3系统监控技术

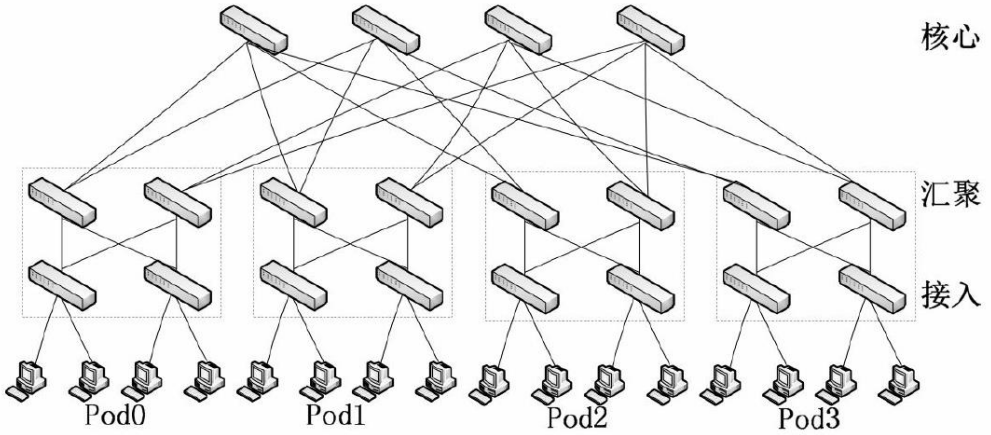
§2.4资源调度技术

# §2.1 数据中心网络


Data Center
PoD
Service Appliances
Data Center Services Node
Integrated Compute Stack
Storage
Network
Compute
PoD
Service Appliances
Data Center Services Node
Integrated Compute Stack
Storage
Network
Compute
Integrated Compute Stack
Storage
Network
Compute
Integrated Compute Stack
Storage
Network
Compute

# 数据中心网络架构

- 改进型胖树网络架构（FatTree）
  - ✓ 三层网络：核心交换机、汇聚交换机、接入交换机
  - ✓ 连接方法：
    - ◆ 每台核心交换机有k个端口，每个端口与一个Pod相连，共k个Pod
    - ◆ 每个Pod有k/2个汇聚交换机和k/2个接入交换
    - ◆ 每个汇聚交换机有k/2个端口连接各个核心，k/2个端口连接接入
      - ✓ 因此需要（k/2）*（k/2）个核心交换机
    - ◆ 每个接入交换机有k/2个端口连接同Pod的各个汇聚，k/2个端口连接主机



# 数据中心网络架构

- 改进型胖树网络架构优缺点分析

✓ 每台核心交换机与每个Pod相连

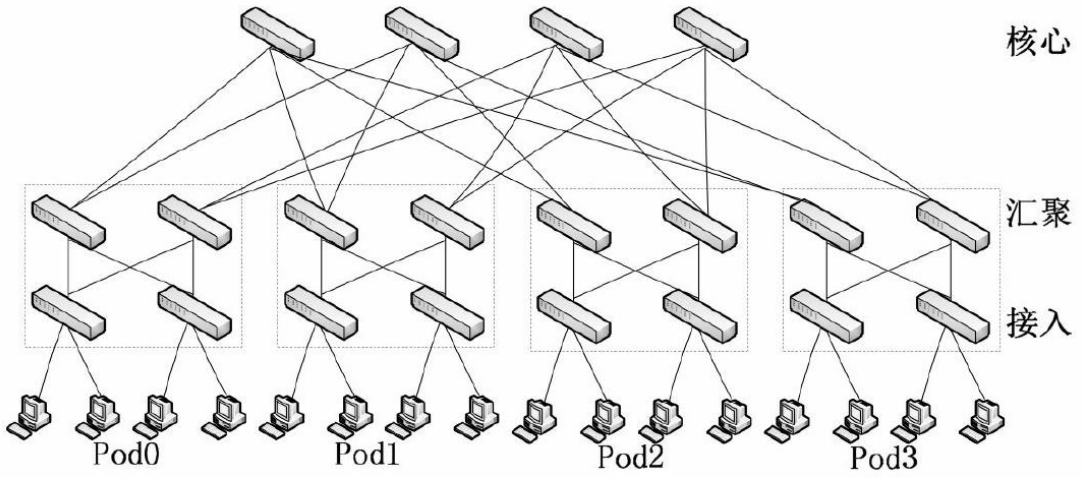
- ◆只要一台核心交换机正常，整个网络就不会断开

✓ 同一Pod内，每个汇聚交换机与所有接入交换机连通

- ◆同一Pod内的流量内部消化

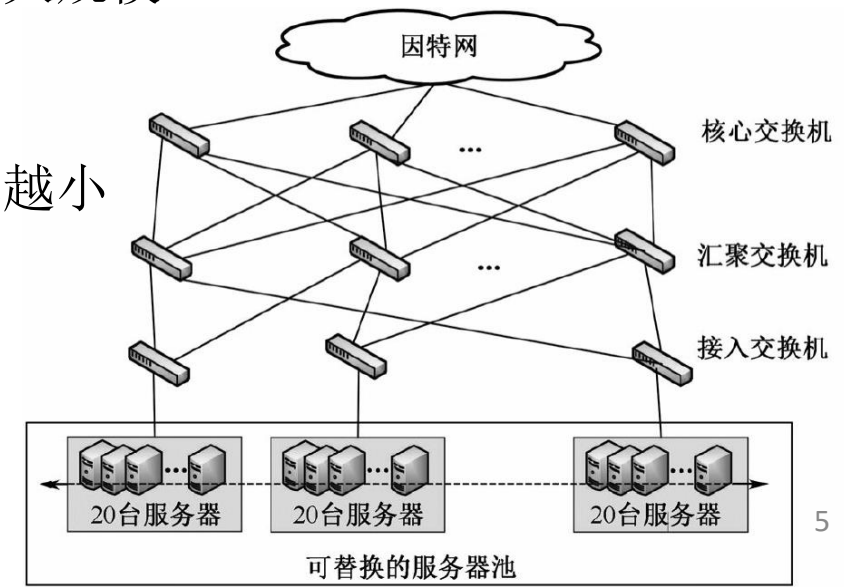
✓ 扩展性

- ◆最大规模为k*（k/2）*（k/2），受限于交换机的端口数
- ◆即：Pod数*每个Pod内的接入交换机数*每个接入用来连接主机的端口数



# 数据中心网络架构

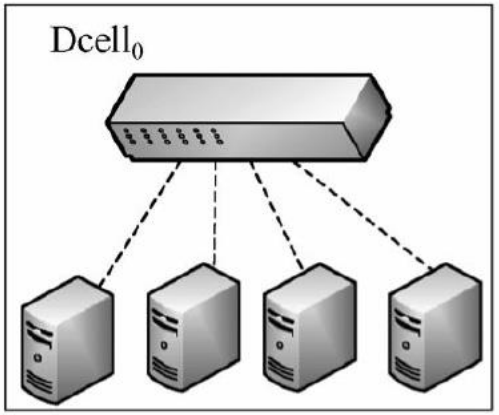
- 微软网络架构（VL2）
  - ✓ 三层网络：核心交换机、汇聚交换机、接入交换机
  - ✓ 连接方法：
    - ◆若干台主机（例如20台）连接到一个接入交换机
    - ◆每台接入与两个汇聚相连
    - ◆每台汇聚与所有核心相连：核心交换机端口数限制网络规模
  - ✓ 优缺点分析
    - ◆节省汇聚和接入的端口用来扩大规模
    - ◆假定交换机端口数k，n个核心
      - ✓ 则规模为k*(k-n)/2*(k-2)
    - ◆核心越多，健壮性越高，规模越小

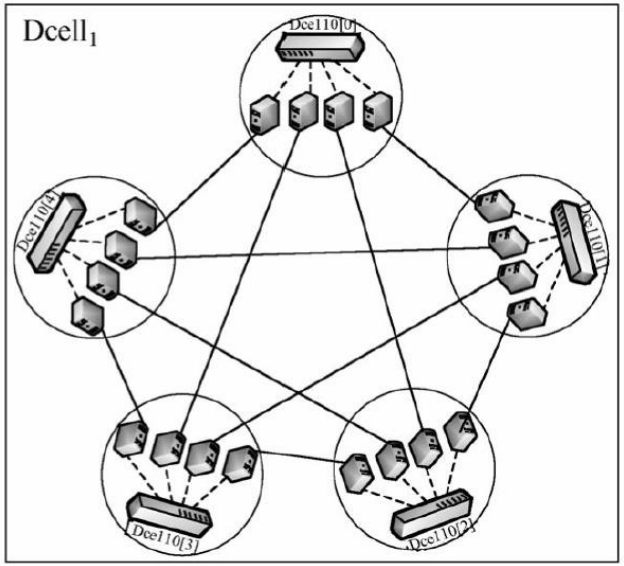


# 数据中心网络架构

第0层：4节点
第1层：4*5=20节点
第2层：20*21=420节点
第3层：420*421=176820

- 递归层次结构之Dcell
- 连接方法：
  - 采用递归方法构建网络第0层由一个交换机连接n个服务器
  - 第1层由n+1个第0层的节点构成：为什么是n+1？
    - 对于一个特定的0层，它的n台服务器分别与其他n个0层中的一台服务器相连
  - 递归关系总结
  - 第k层服务器数为$S_k$：$S_k=(S_{k-1})S_{k-1}$
- 优缺点分析
  - 布线复杂
  - 层数受限于端口数
  - 扩展性好

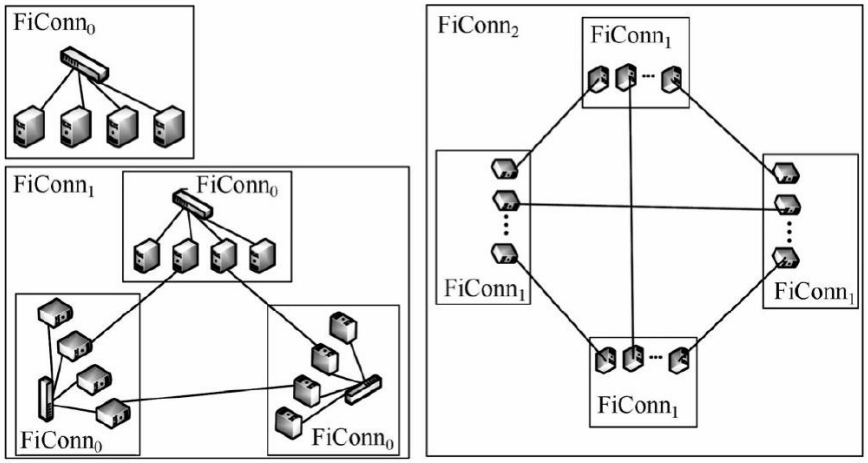


# 数据中心网络架构

- 递归层次结构之FiConn
- ✓ 连接方法：采用递归方法构建网络
  - 第0层由一个交换机连接n个服务器，每个服务器除了跟该交换机连接外，还有一个备用端口待用
  - 第1层构建：对于每个$FiCnn_0$，拿出其中一半还没使用的备用端口，与其它的$FiCnn_0$连接
  - 第2层构建：对于每个$FiCnn_1$，拿出其中一半还没使用的备用端口，与其它的$FiCnn_1$连接
  - 递归关系总结
    - ✓ 第k层服务器数为$S_k$，空闲备用端口$B_k$
    - ✓ 则$S_{k+1} = S_k(\frac{B_k}{2} + 1)$，$B_{k+1} = \frac{B_k}{2}(\frac{B_k}{2} + 1)$

第0层：4节点，$B_0$=4
第1层：4*（2+1）=12节点，$B_1$=6
第2层：12*（3+1）=48节点，$B_2$=12
第3层：48*（6+1）=336节点，$B_3$=42

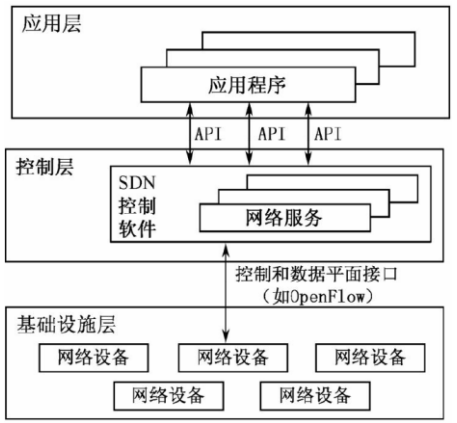


# 数据中心网络管理

- 基于SDN的数据中心技术
- **Software-defined networking (SDN)** technology is an approach to network management that enables dynamic, programmatically efficient network configuration in order to improve network performance and monitoring making it more like cloud computing than traditional network management.

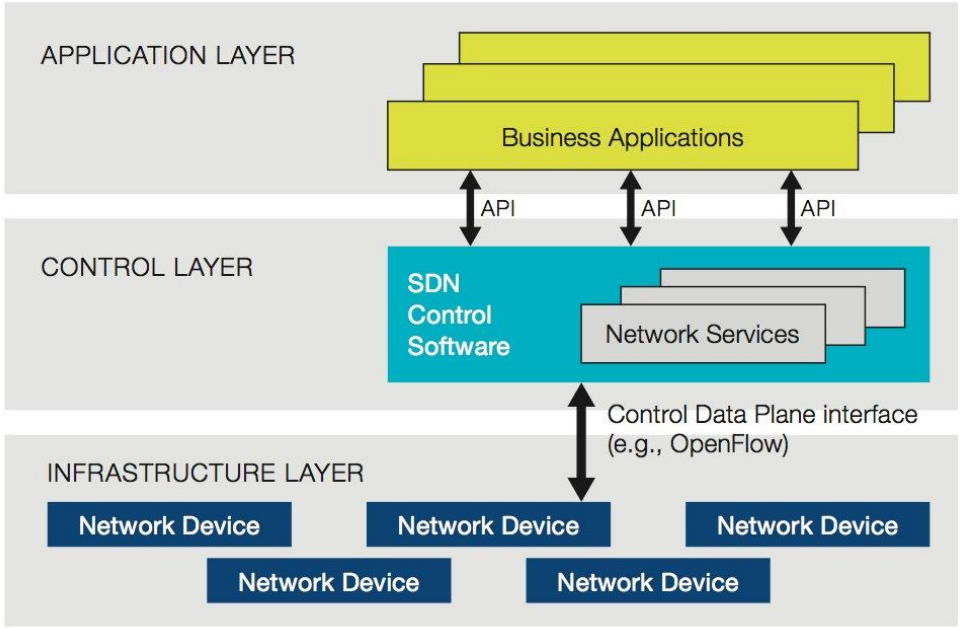
# 数据中心网络管理

- 软件定义网络（SDN，Software Defined Networking）
  - ✓ 一种新型的网络技术
  - ✓ 将网络的控制与数据转发进行分离
  - ✓ 数据流的接入、路由等都由控制器来控制
  - ✓ 交换机只是按控制器所设定的规则进行数据分组的转发
  - ✓ 通过开放可编程软件模式来实现网络的自动化控制功能
- SDN主要优势
  - ✓ 实现集中控制，因而能够通过达到最优性能
  - ✓ 网络设备功能简化，从封闭走向开放
    - ✓ 底层的网络设备能够专注于数据转发,功能简化
    - ✓ 可以在廉价的裸机或白盒服务器上实现
  - ✓ 方便网络运行维护，仅需要通过软件的更新来实现
    - ✓ 网络功能的升级无须再针对每一个硬件设备进行配置



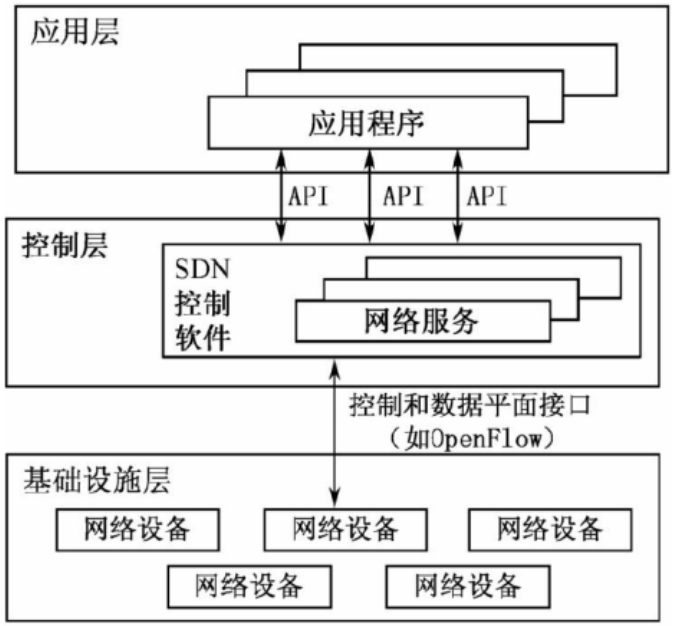
# SDN技术

- 三个层/面
  - 数据层
  - 控制层
  - 应用层
- 两个接口
  - **南向接口**
    - 控制面跟数据转发面之间的接口
    - 传统网络的南向接口存在于各个设备商的私有代码中，对外不可见，如存在于交换机内部
  - **北向接口**
    - 控制器跟应用程序之间的接口
    - 传统网络里，指交换机控制面跟网管软件之间的接口



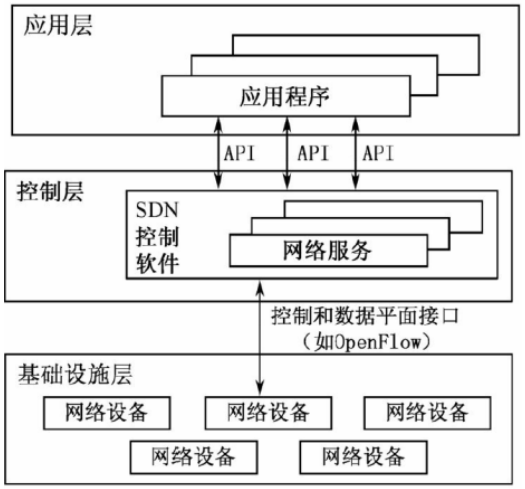
# SDN控制器

- 控制器是整个SDN网络的核心大脑
- 负责数据平面资源的编排、维护网络拓扑和状态信息等
- 向应用层提供北向API接口。
- 其核心技术包括
  - 链路发现和拓扑管理
  - 高可用和分布式状态管理
  - 自动化部署以及无丢包升级
- 开源控制器
  - OpenDaylight
  - ONOS
  - NOX/POX
  - OpenContrail
  - Ryu
  - Floodlight



# SDN数据平面

- 数据平面负责数据处理、转发和状态收集等。
- 其核心设备为交换机，可以是物理交换机，也可以是虚拟交换机。
- 转发设备将数据平面与控制平面完全解耦
- 所有数据包的控制策略由远端的控制器通过南向接口协议下发，网络的配置管理同样也由控制器完成。



# SDN+DataCenter

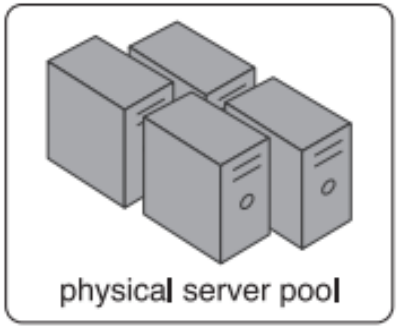
- SDN数据中心更智能高效灵活，提升上层业务的SLA
- 可以快速部署业务平台
- 可以根据应用和服务变化动态配置网络资源
  - 如：服务、虚拟机的迁移
- 简化网络运维，提升运维效率

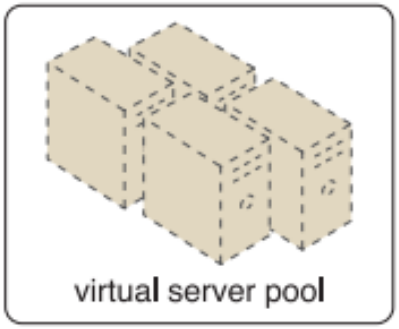
# §2.2资源池化技术

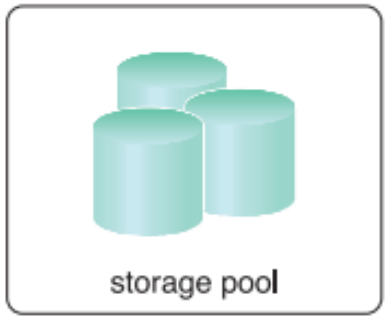
- 概念
  - 资源池是由计算、存储、网络等资源构成的可以统一管理分配的资源集合。
  - 资源池是灵活管理资源的逻辑抽象。
- 特点
  - 灵活、弹性的资源部署，提高资源利用率
  - 有效的提高扩展和容错能力，提升运维效率

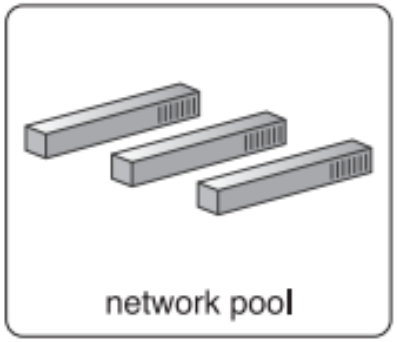
# 资源池架构

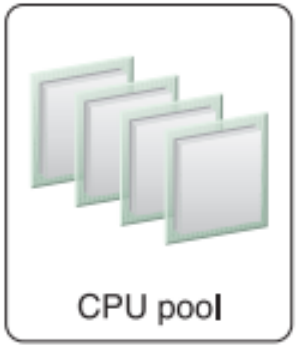
- 用一个或多个资源池
- 相同的IT资源由一个系统进行分组和维护
  - 需要保持同步
- 常见的资源池种类：

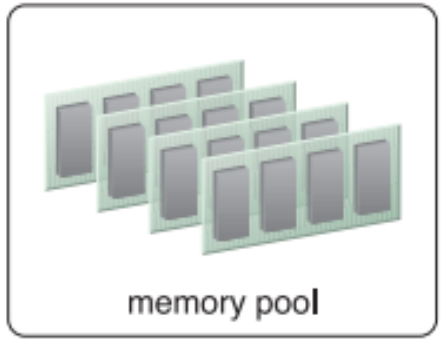


# 复杂资源池示例

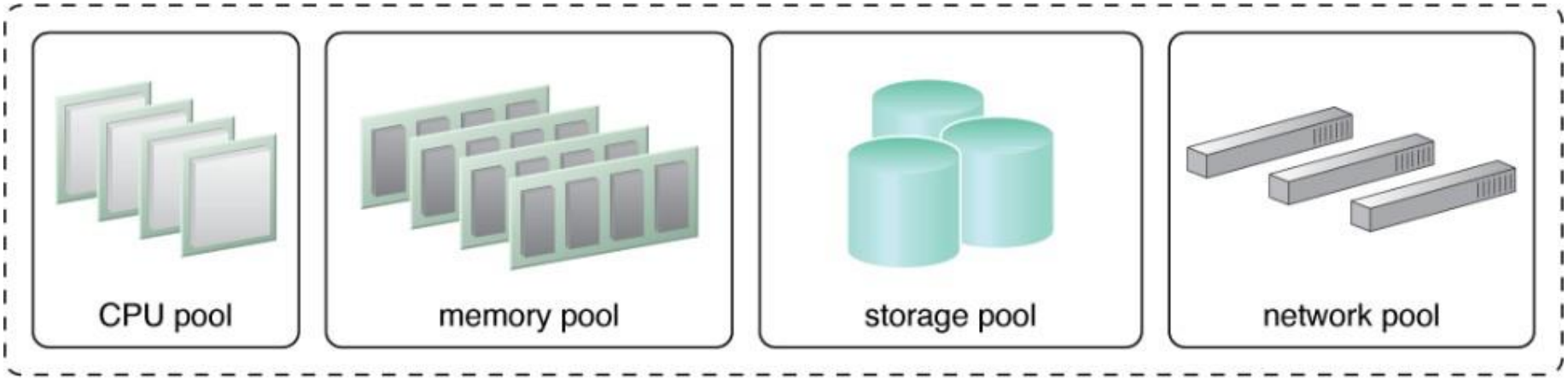





Figure11.2 该资源池由4个子资源池组成，分别是：CPU池、内存池、云存储设备池和虚拟网络设备池。

# 层次资源池架构

- 资源池可以建立层次结构，形成资源池之间的父子（parent）、兄弟（sibling）和嵌套（nested）关系，从而有利于构成不同的资源池需求。

- 同级资源池（Sibling pools）之间是互相隔离的，云用户只能访问各自的资源池。

- 嵌套资源池（Nested pools）可以用于向同一个云用户组织的不同部门或者不同组分配资源池。

# 同级资源池（sibling pools）

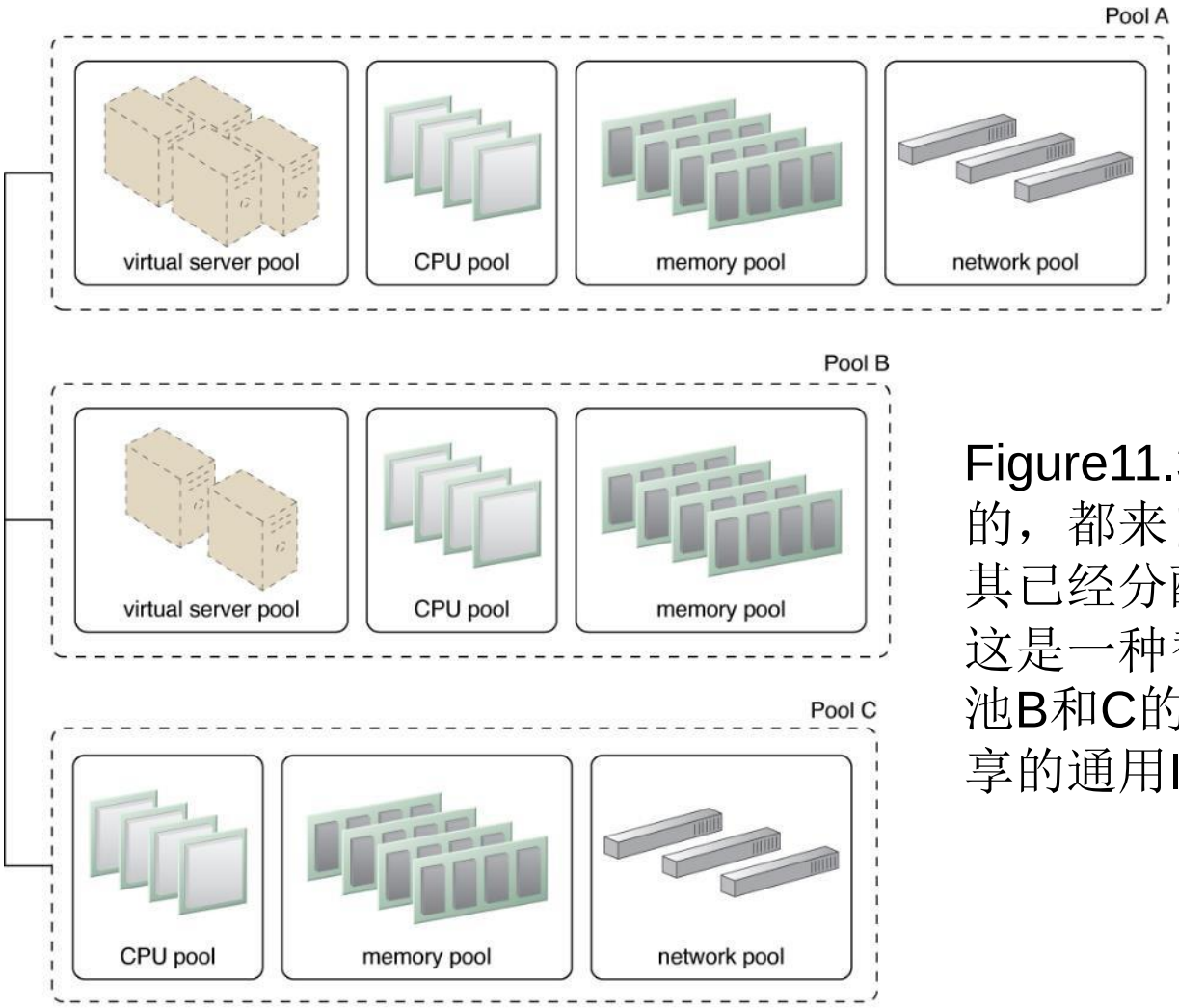





Figure11.3 资源池B和C是同级的，都来自于较大的资源池A，其已经分配给云用户了。
这是一种替代方法，使得资源池B和C的IT资源不需要从云共享的通用IT资源储备池中获得。

# 嵌套资源池（nested pools）

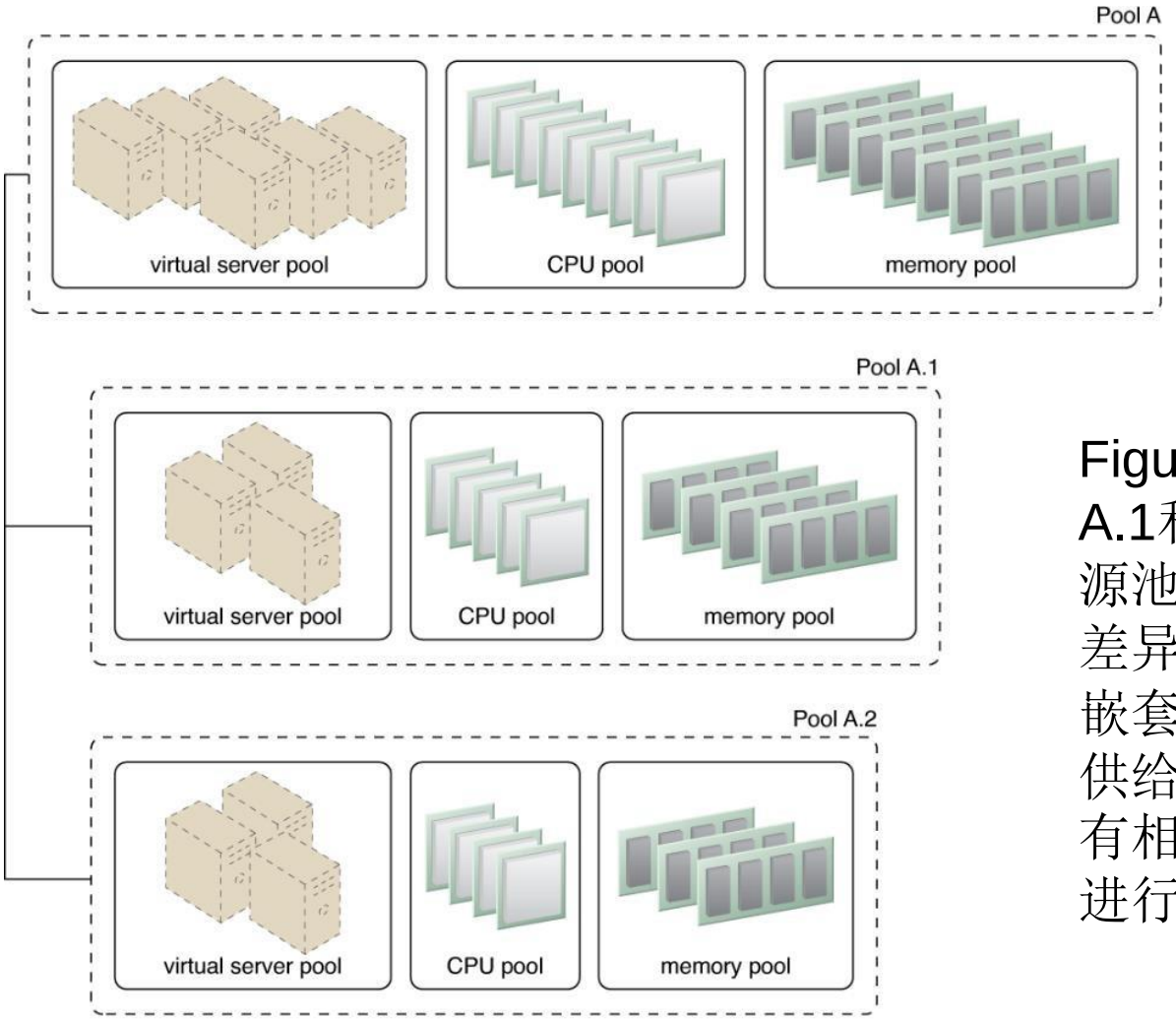





Figure11.4 嵌套的资源池A.1和A.2包含的IT资源和资源池A相同，只是数量上有差异。

嵌套资源池通常用于云服务供给，这些云服务需要用具有相同配置的同类型IT资源进行快速实例化。

# §2.3系统监控技术

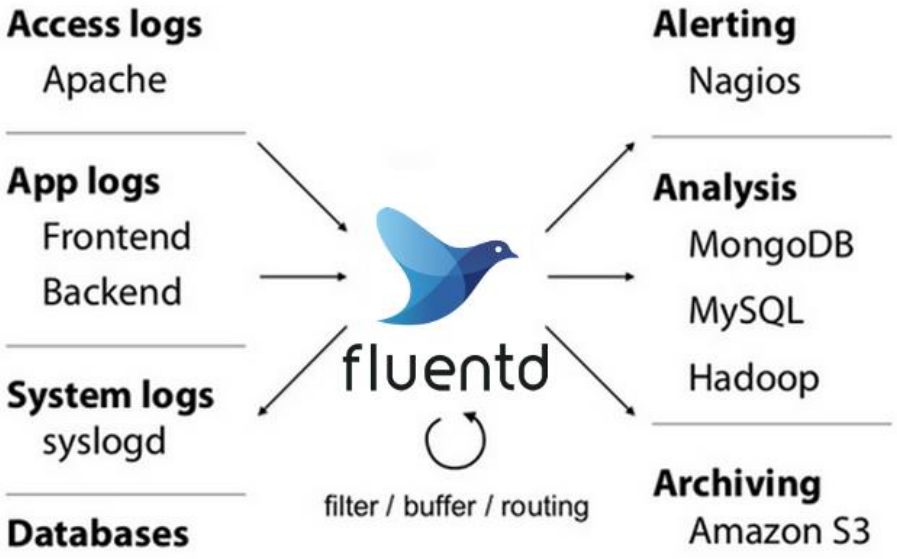
- 功能
  - 状态监控
  - 性能监控
  - 容量监控
  - 安全监控
  - 使用度量
- 常用方法
  - 日志分析
  - 报嗅探
  - 探针采集

# 主机系统监控

- Linux系统运维命令
  - 内存： top, free ,vmstat, mpstat, iostat, sar, pmap
  - CPU： top, vmstat, mpstat, iostat, sar
  - I/O ： vmstat, mpstat, iostat, sar
  - 进程： ipcs, ipcrm
  - 系统运行负载：uptime, w

# 集群资源监控工具

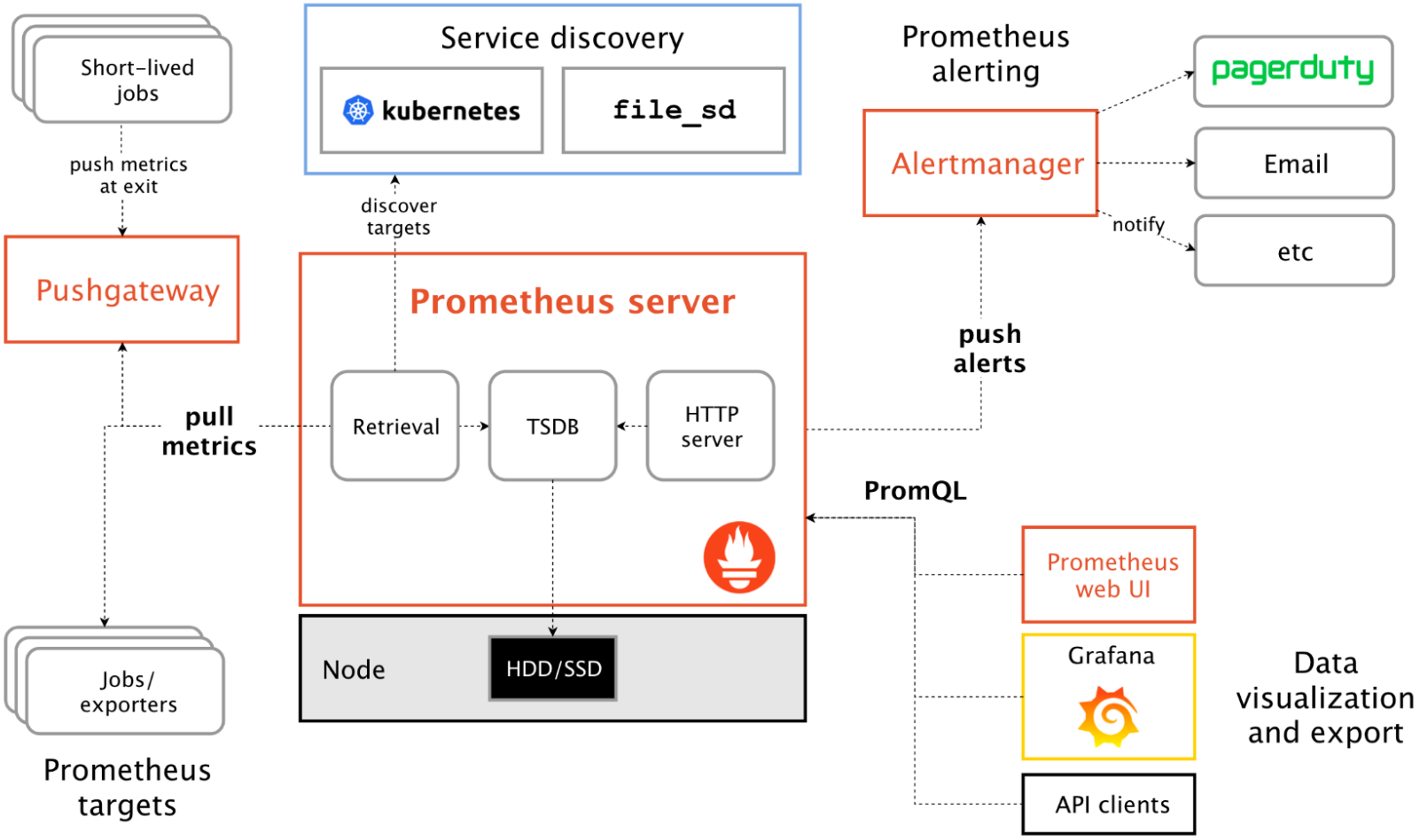
- **Elasticsearch + Kibana + Logstash(ELK stack)**
- LOGalyze
- Fluentd
- Prometheus+Grafana



# 集群监控工具—Prometheus

- 起源： BorgMon 谷歌内部监控系统
- 工作原理：使用抓取pull的方式，获取监控数据，存于TSDB中。以便后续可以按照时间进行检索。
- 适合做虚拟化环境监控系统，比如VM、Docker、Kubernetes等。

# 集群监控工具—Prometheus



# 集群监控工具—Prometheus

- 基本组件：
  - Prometheus Server ： 包括HTTP Server 与 一个内置的时序数据库。主要负责数据采集和存储，提供PromQL查询语言的支持
  - Alertmanager可以根据 Metrics 信息灵活地设置报警
  - Push Gateway： 允许被监控对象以 Push 的方式向 Prometheus 推送 Metrics 数据

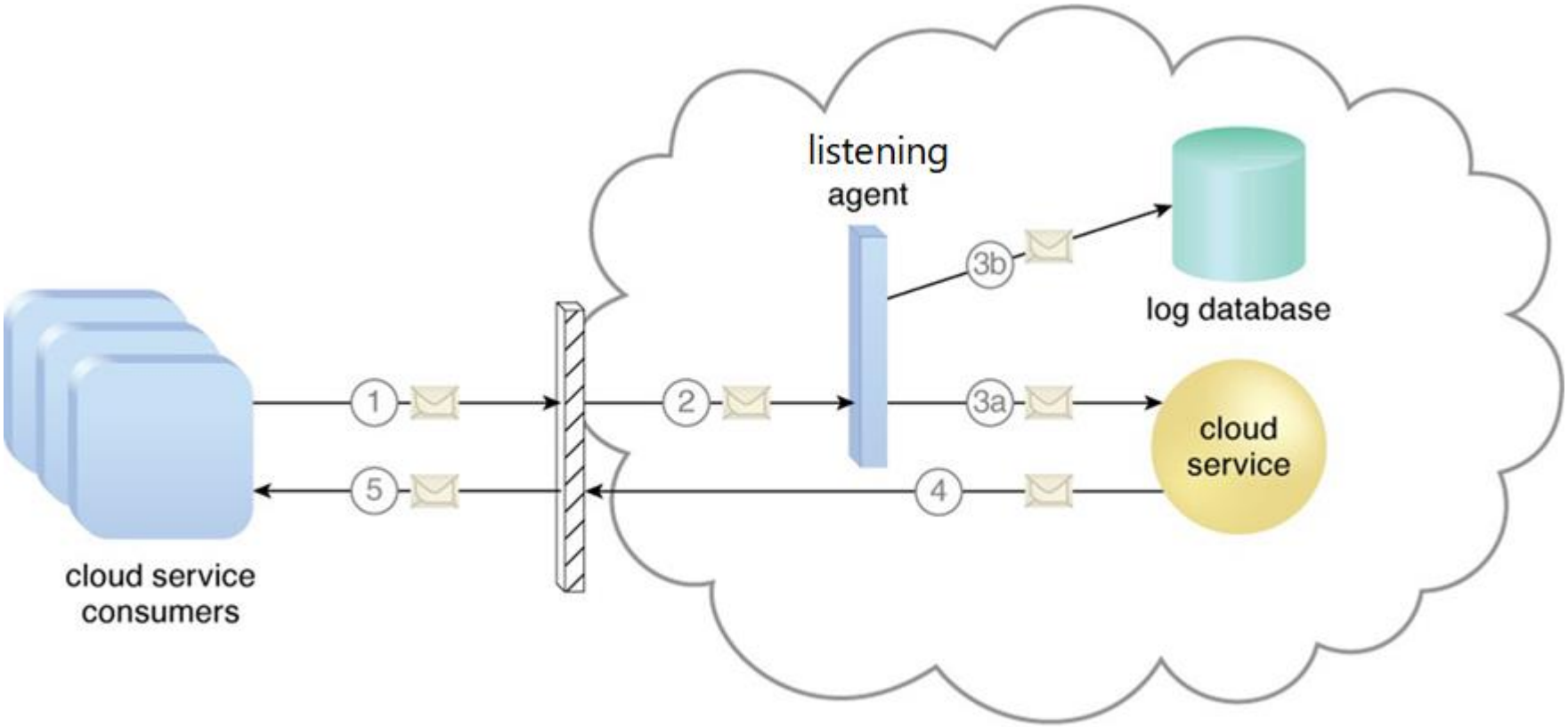
# 云使用监控

- 云使用监控用于收集和处理IT资源的使用数据
- 使用数据发送到日志数据库，以便进行后续处理和报告。
- 三种种常见的实现形式：（基于代理）
  - 监听代理（listening agent）
  - 资源代理（resource agent）
  - 轮询代理（polling agent）

# 云使用监控

- 监听代理是一个中间的事件驱动程序，对数据流进行透明的监控和分析。
- 资源代理是一种处理模块，在资源软件级别监控预定义的且可观测事件的使用指标，比如：启动、暂停、恢复和垂直扩展。
- 轮询代理是一种处理模块，通过轮询IT资源来周期性地监控IT资源状态（比如正常运行时间和停机时间）。

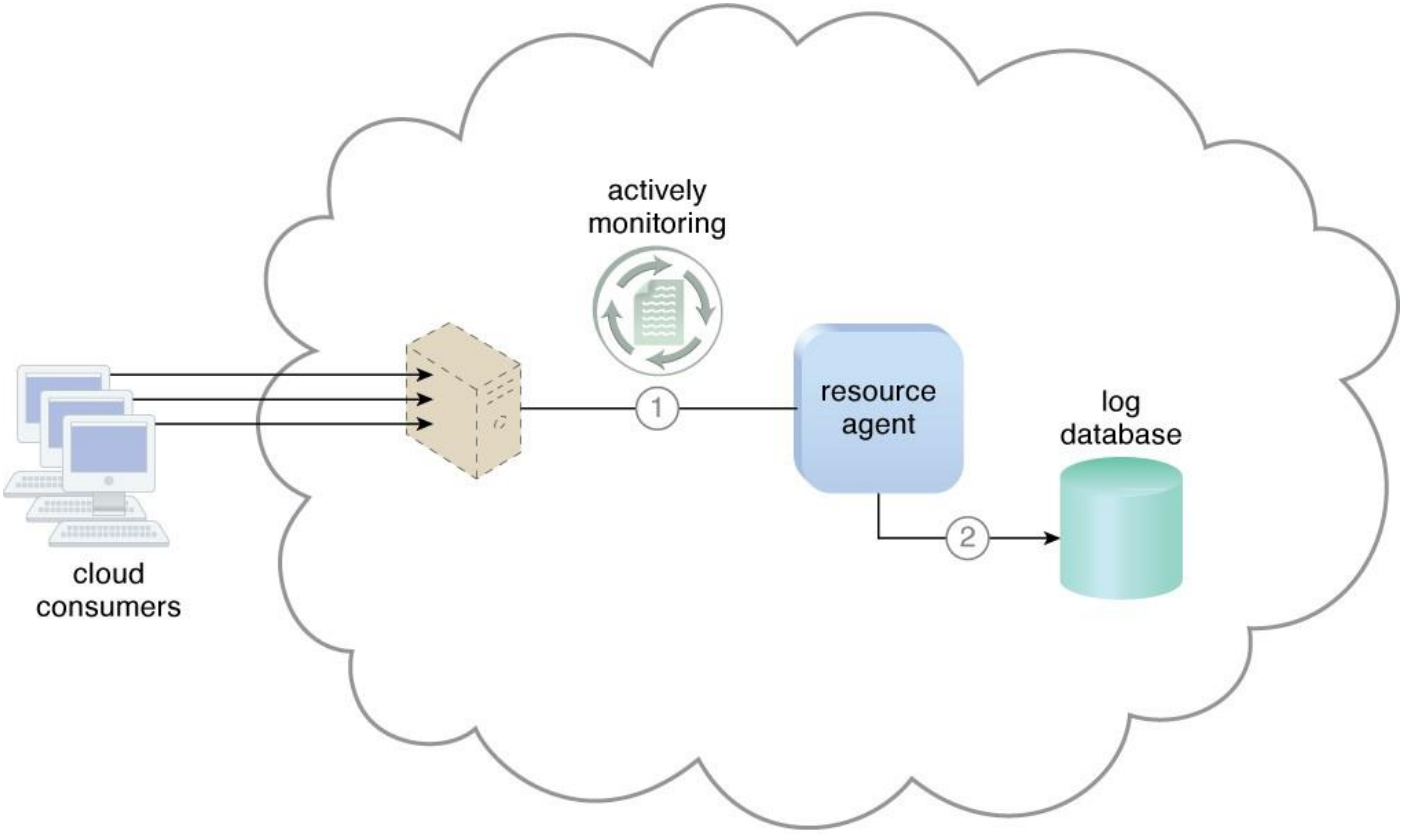
# 监听代理






云服务用户向云服务发送请求消息（1）监控代理拦截此消息，收集相关使用数据（2），然后将其继续发往云服务（3a）。监控代理将收集到的使用数据存入日志数据库（3b）。云服务产生应答消息（4），并将其发送回云服务用户，此时监控代理不会进行拦截（5）。

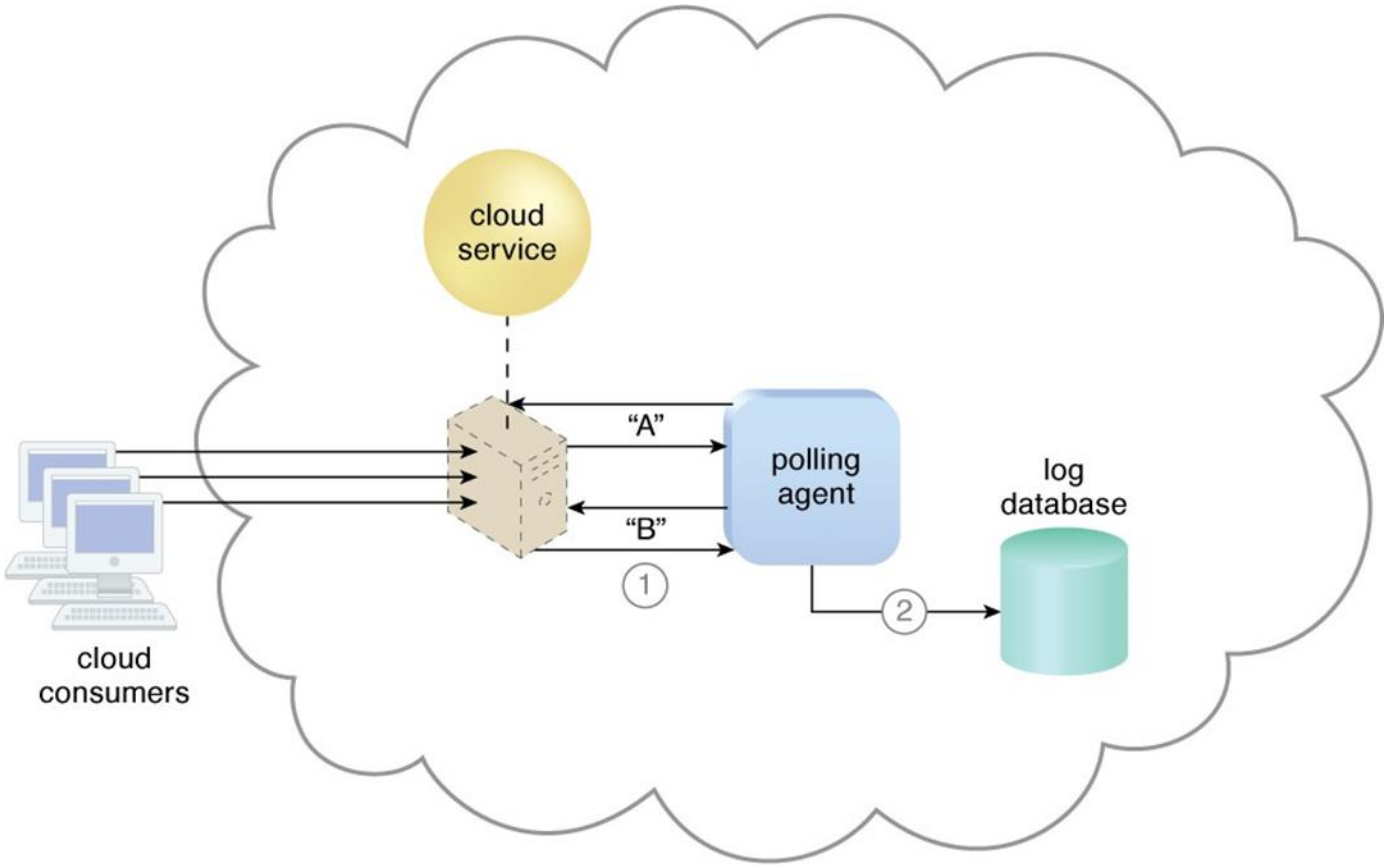
# 资源代理






资源代理主动监控虚拟服务器，并检测到使用的增加（1）。资源代理从底层资源管理程序收到通知，虚拟服务器正在进行扩展，按照其监控指标，资源代理将收集的使用数据存入日志数据库（2）。

# 轮询代理






轮询代理监控虚拟服务器上的云服务状态，它周期性地发送轮询消息，并在数个轮询周期后接收到使用状态为“A”的轮询响应消息。当代理接收到使用状态为“B”时（1），轮询代理就将新的使用状态记录到日志数据库中（2）。

# §2.4资源调度技术

- 基本概念：
  - 调度，即为需要运行的任务或服务找到合适的资源，满足其运行需求
- 调度的两种场景/类型
  - （在线）服务调度、（离线/后台）任务调度
- 调度算法：
  - 单资源调度算法
    - FCFS 、带优先级的FCFS、Round-Robin
    - Max-min Fair Share Algorithm等
  - 多资源调度算法
    - Assets Fairness、DRF(Domain Resource Fairness)等

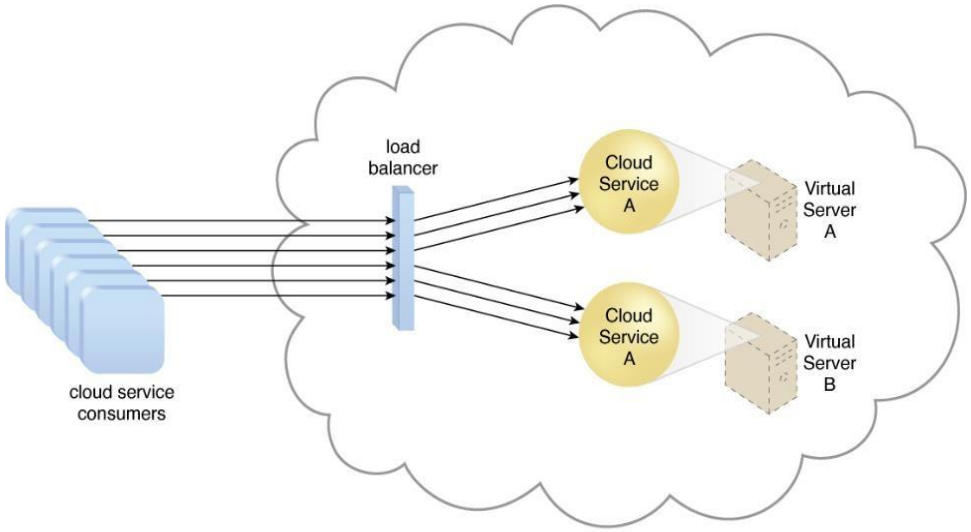
# 服务调度技术

- 调度方法
  - 基于DNS
  - 基于虚拟IP
  - 基于链路聚合：用于整合链路提高网络传输能力
  - 基于应用：用于分配到分布式调度器
- 调度策略
  - 轮转、负载水平，…
  - 同一用户的多个请求调度到同一服务器
  - 同一租户的请求调度到尽量少的一组服务器
  - 尽量实现不同类型负载的互补
  - 。。。

# 负载均衡

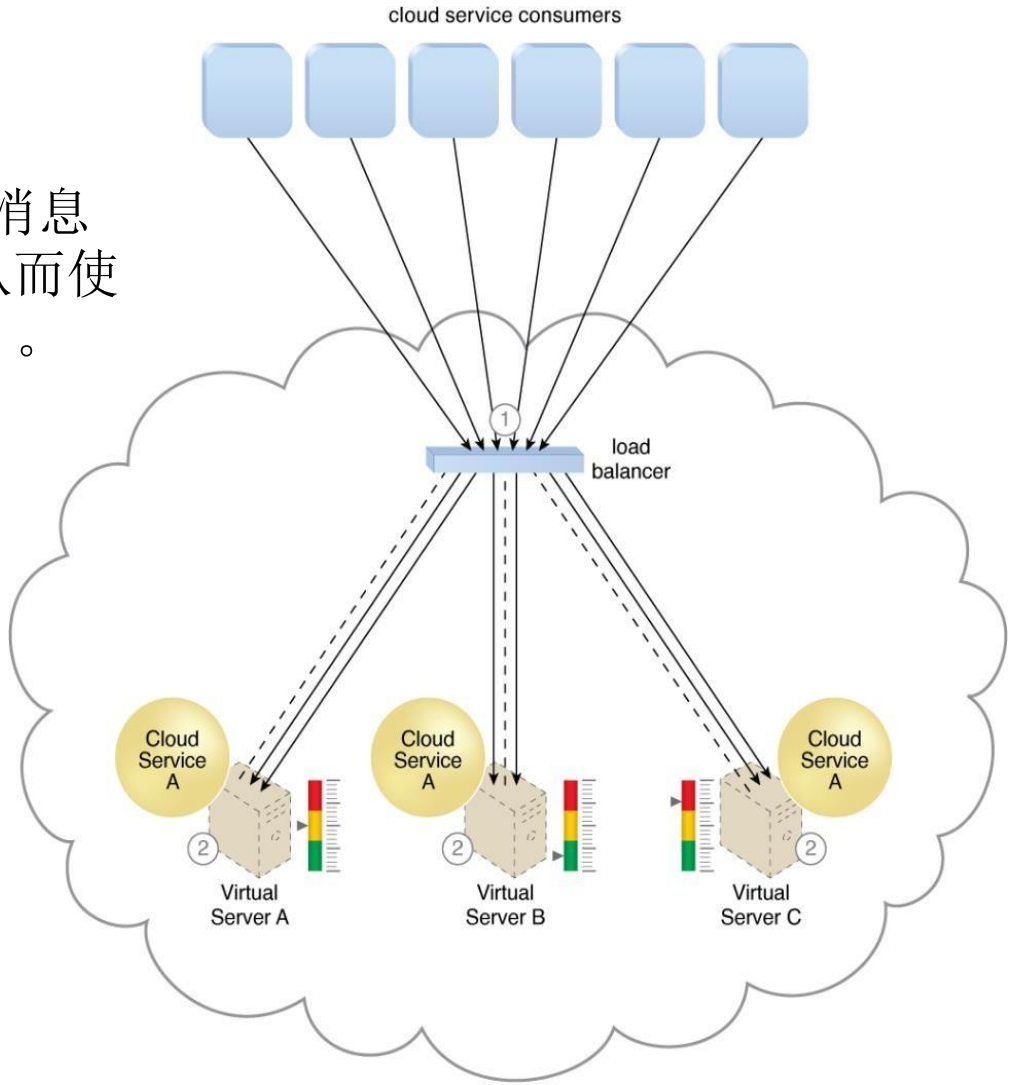
- 负载均衡器通常位于云数据中心前端的通讯路径上。
- 设计成一个透明的代理或是一个代理的组件

- multi-layer network switch
- dedicated hardware appliance
- dedicated software (in server OS))
- service agent



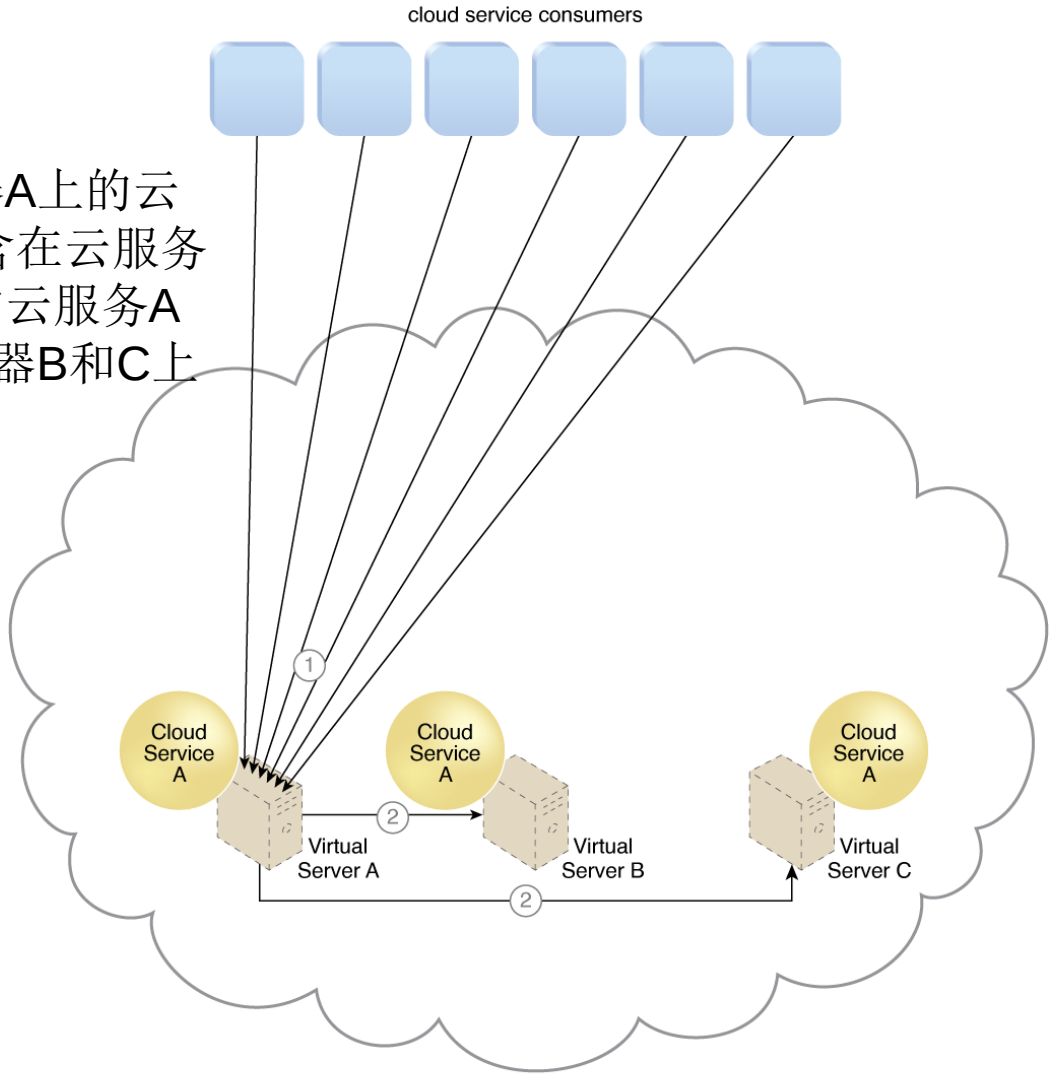
# 服务负载均衡两种架构（1/2）

负载均衡器截获云服务用户发送的消息（1）并将其转发给虚拟服务器，从而使工作负载的处理得到水平扩展（2）。



# 服务负载均衡两种架构（2/2）

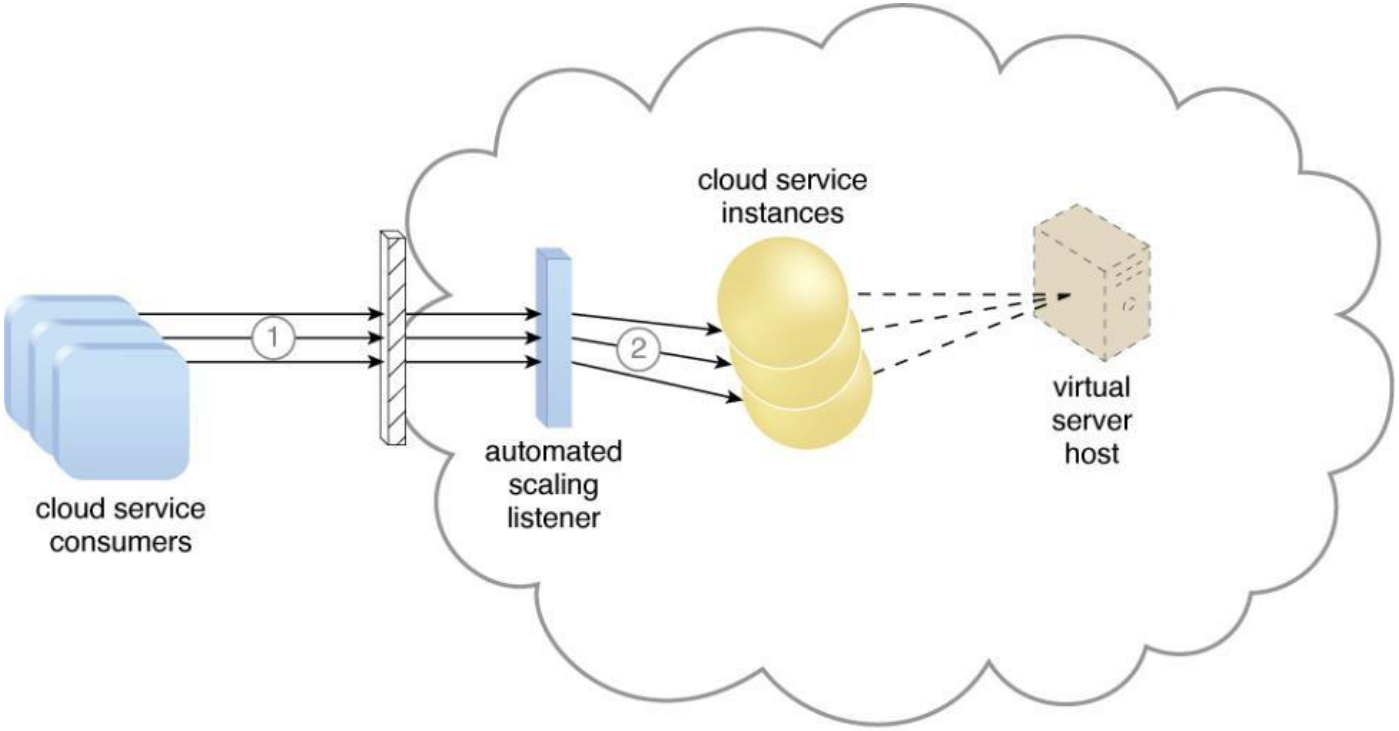
云服务用户的请求发送给虚拟服务器A上的云服务A（1）。内置负载均衡逻辑包含在云服务实现中，它可以将请求分配给相邻的云服务A，这些云服务A的实现位于虚拟服务器B和C上（2）。



# 服务动态扩展

- 动态水平扩展（Dynamic Horizontal Scaling）
  - 向内或向外扩展IT资源实例
  - 自动扩展监听器请求资源复制，并发信号启动IT资源复制
- 动态垂直扩展（Dynamic Vertical Scaling）
  - 调整单个IT资源的处理容量
  - 向上或向下扩展IT资源实例
- 动态重定位（Dynamic Relocation）
  - 将IT资源重放置到更大/更小容量的主机上

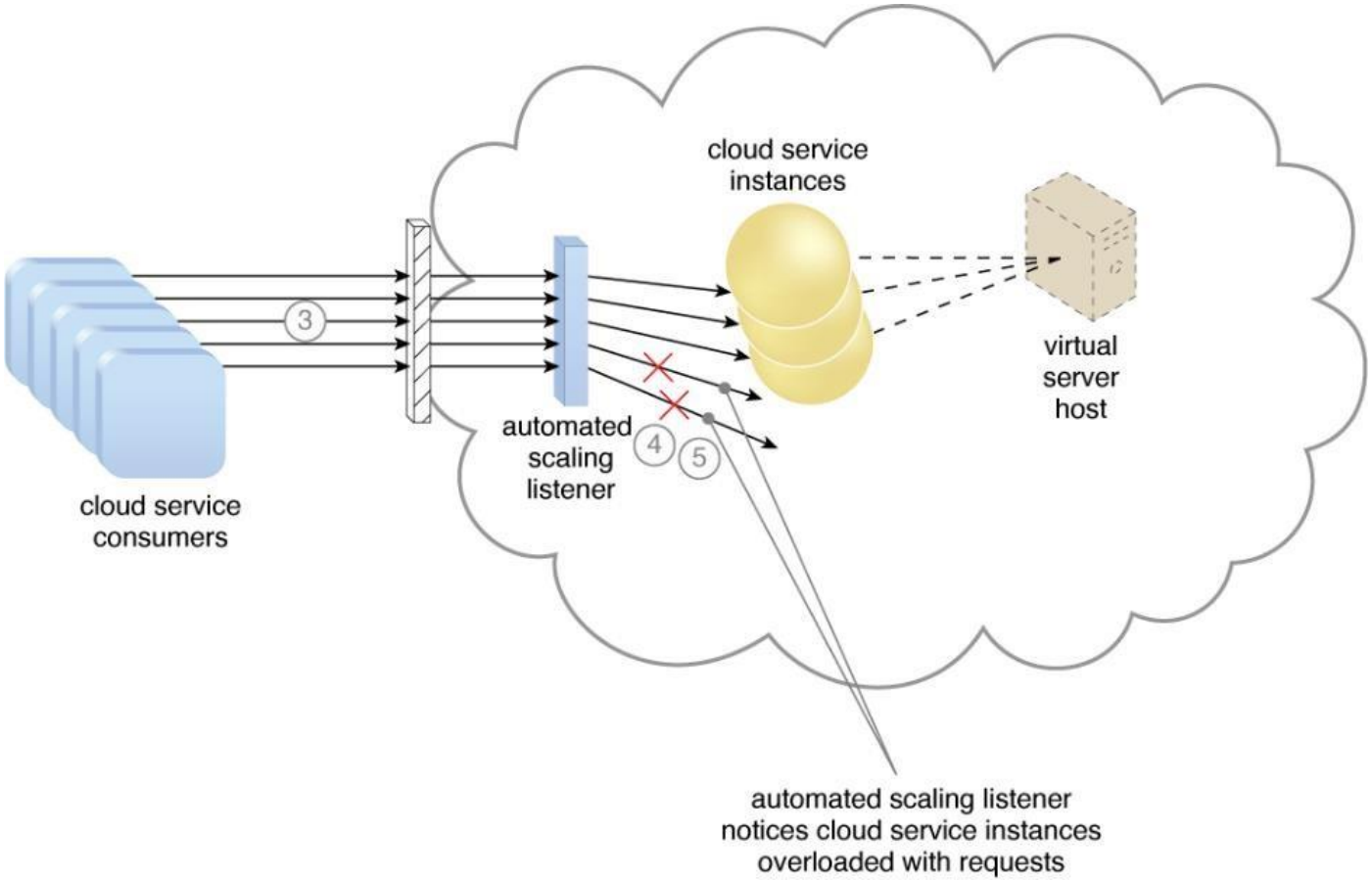
# 动态水平扩展过程（1/3）





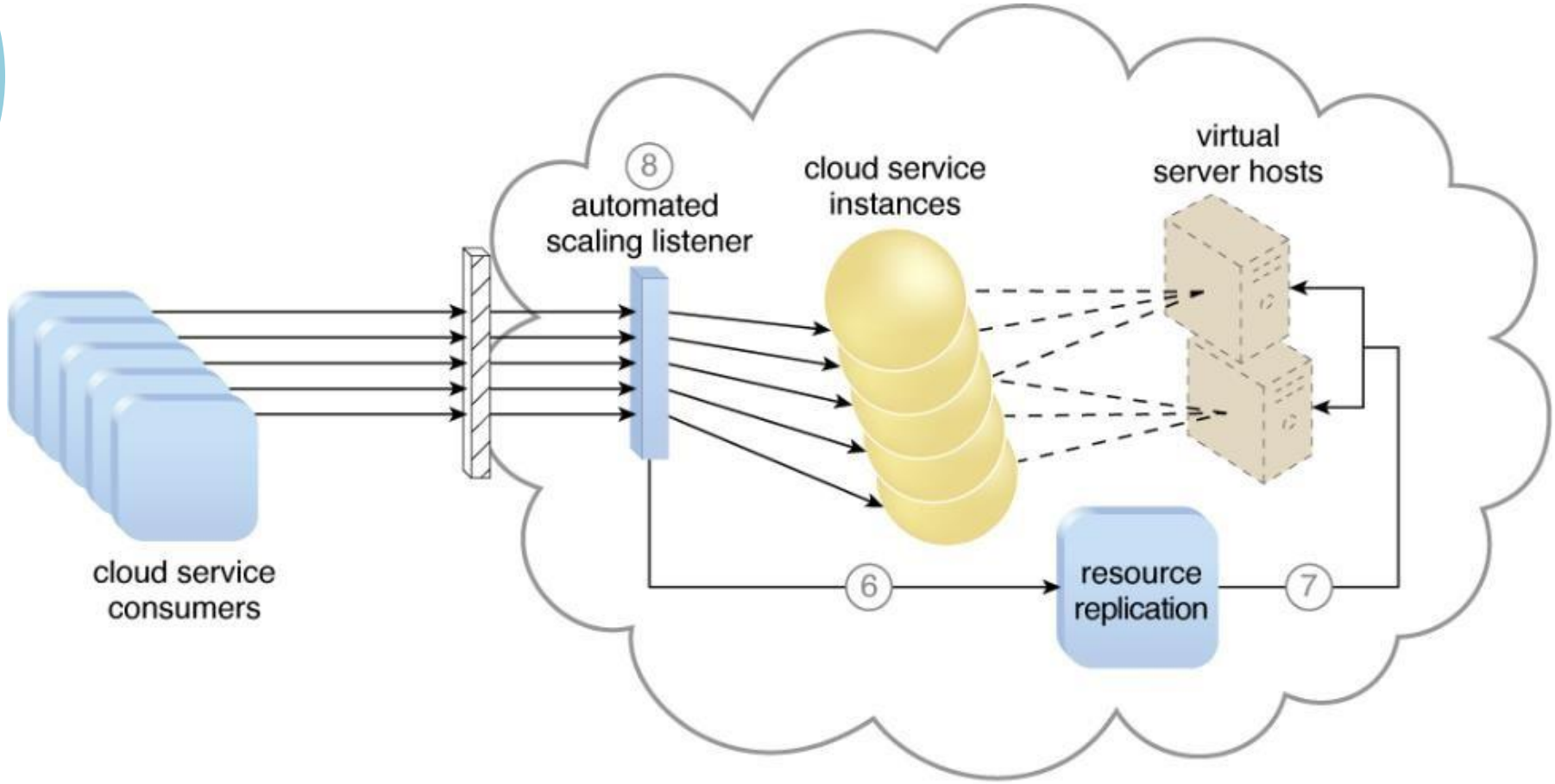

云服务用户想云服务发送请求（1）。自动扩展监听器监视该云服务，判断预定义的容量阈值是否已经被超过（2）。

# 动态水平扩展过程（2/3）




云服务用户的请求数量增加（3）。工作负载已超过性能阈值。根据预定义规则，自动扩展监听器决定下一步的操作（4）。如果云服务的实现被认为适合扩展，则自动扩展监听器启动扩展过程（5）。

# 动态水平扩展过程（3/3）




自动扩展监听器想资源复制机制发送信号（6），创建更多的云服务实例（7）。增加的工作负载可以得到满足，自动扩展监听器根据请求，继续监控并增加或减少IT资源（8）。

# 任务调度技术

- 为计算任务分配相应的资源
  - 多任务/多租户场景
  - 资源稀缺有限
  - 任务规模，优先级不同，资源需求类型不同
  - 使得资源分配尽量公平
- 资源公平算法
  - Max-min Fairness
  - Asset Fairness
  - Dominant Resource Fairness
  - …

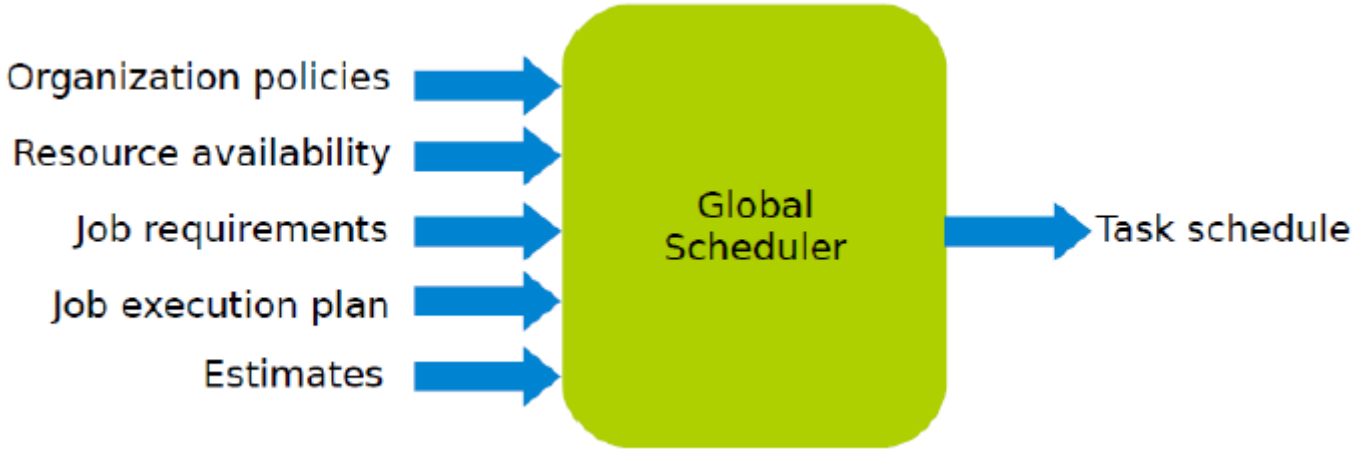
# 调度框架

- Global scheduler
- Distributed scheduler

# Global Scheduler (1/2)

- Job requirements
  - Response time
  - Throughput
  - Availability
- Job execution plan
  - Task DAG
  - Inputs/outputs

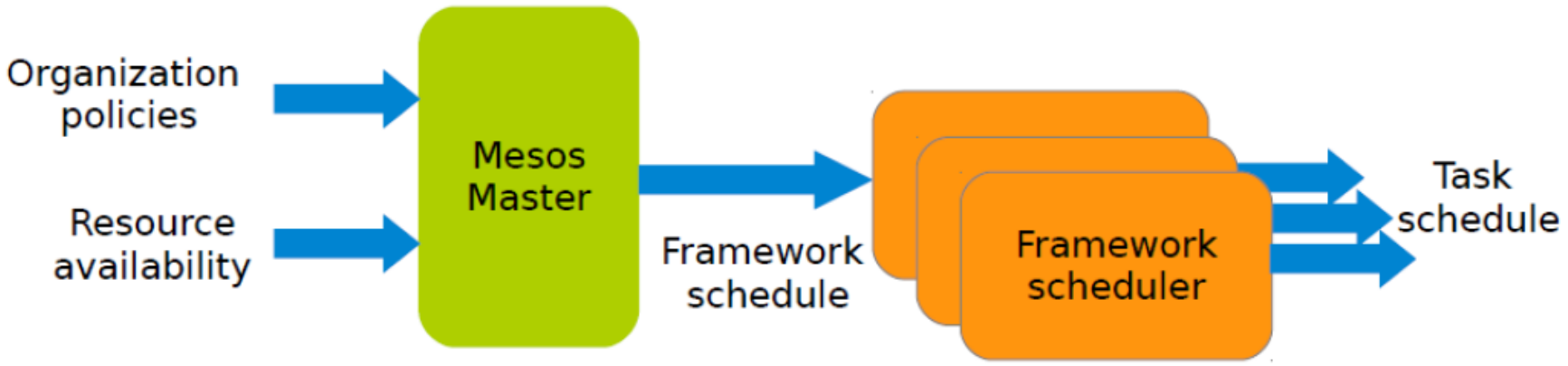
- Estimates
  - Task duration
  - Input sizes
  - Transfer sizes



# Global Scheduler (2/2)

- Advantages
  - Can achieve optimal schedule.
- Disadvantages
  - Complexity: hard to scale and ensure resilience.
  - Hard to anticipate future frameworks requirements.
  - Need to refactor existing frameworks.

# Distributed Scheduler (1/3)



# Distributed Scheduler (2/3)

- Unit of allocation: resource offer
  - Vector of available resources on a node
  - For example, node1: <1CPU, 1GB>, node2: <4CPU, 16GB>

- Master sends resource offers to frameworks.

- Frameworks select which offers to accept and which tasks to run.

# Distributed Scheduler (3/3)

- Advantages
  - Simple: easier to scale and make resilient.
  - Easy to port existing frameworks, support new ones.

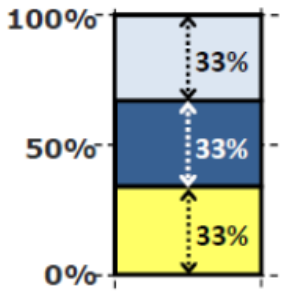
- Disadvantages
  - Distributed scheduling decision: not optimal.

# 调度算法

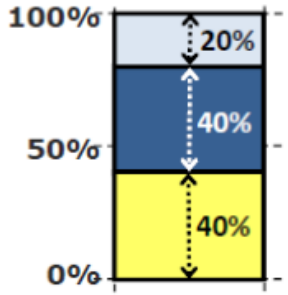
- 单资源调度
  - 调度时只考虑一种资源，如CPU
- 多资源调度
  - 调度时考虑多种资源，如CPU+Memory

# 单资源调度: Fair Sharing

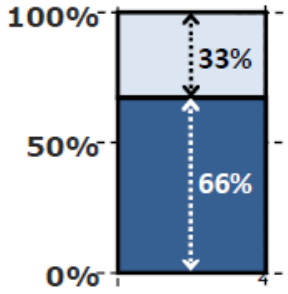
- n users want to share a resource, e.g., CPU.
  - Solution: allocate each 1/n of the shared resource.




- Generalized by max-min fairness.
  - Handles if a user wants less than its fair share.
  - E.g., user 1 wants no more than 20%.




- Generalized by weighted max-min fairness.
  - Give weights to users according to importance.
  - E.g., user 1 gets weight 1, user 2 weight 2.



# Max-Min Fairness

- 1 resource: CPU
- Total resources: 20 CPU
- User 1 has x tasks and wants <1CPU> per task
- User 2 has y tasks and wants <2CPU> per task

$$\max(x, y) \text{ (maximize allocation)}$$
subject to
$$x + 2y \leq 20 \text{ (CPU constraint)}$$
$$x = 2y$$
so
$$x = 10$$
$$y = 5$$

# Why is Fair Sharing Useful?

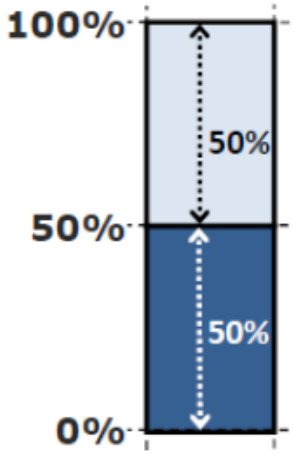
- Proportional allocation: user 1 gets weight 2, user 2 weight 1.
- Priorities: give user 1 weight 1000, user 2 weight 1.
- Reservations: ensure user 1 gets 10% of a resource, so give user 1 weight 10, sum weights ≤ 100.
- Isolation policy: users cannot affect others beyond their fair share.

# Properties of Max-Min Fairness

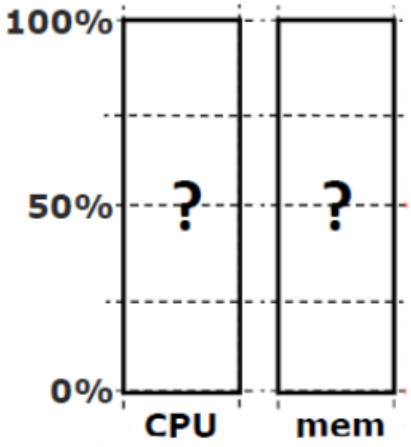
- Share guarantee
  - Each user can get at least 1/n of the resource.
  - But will get less if her demand is less.
- Strategy proof
  - Users are not better off by asking for more than they need.
  - Users have no reason to lie.
- Max-Min fairness is the only reasonable mechanism with these two properties.
  - Widely used: OS, networking, datacenters, ...

# 多资源调度

- Single resource example
  - 1 resource: CPU
  - User 1 wants <1CPU> per task
  - User 2 wants <2CPU> per task




- Multi-resource example
  - 2 resources: CPUs and mem
  - User 1 wants <1CPU; 4GB> per task
  - User 2 wants <2CPU; 1GB> per task
  - What is a fair allocation?



# A Natural Policy (1/2)

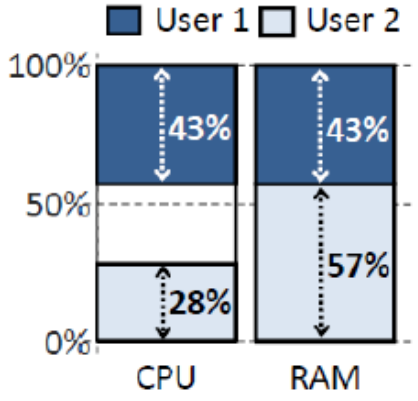
- Asset fairness: give weights to resources (e.g., 1 CPU = 1 GB) and equalize total value given to each user.
- Total resources: 28 CPU and 56GB RAM (e.g., 1 CPU = 2 GB)
  - User 1 has x tasks and wants <1CPU; 2GB> per task
  - User 2 has y tasks and wants <1CPU; 4GB> per task

- Asset fairness yields:

$$\max(x, y)$$
$$x + y \leq 28$$
$$2x + 4y \leq 56$$
$$4x = 6y$$

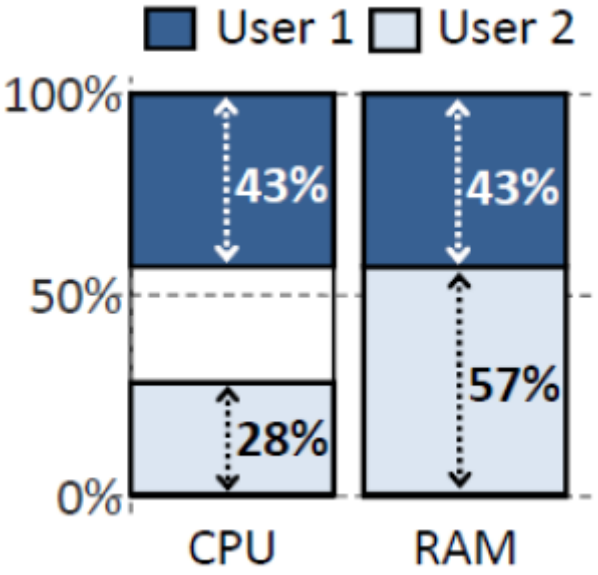
User 1: $x = 12$: $\langle 43\%CPU, 43\%GB \rangle$ $(\sum = 86\%)$

User 2: $y = 8$: $\langle 28\%CPU, 57\%GB \rangle$ $(\sum = 86\%)$



# A Natural Policy (2/2)

- Problem: violates share guarantee.
- User 1 gets less than 50% of both CPU and RAM.
- Better off in a separate cluster with half the resources.



# Challenge

- Can we find a fair sharing policy that provides:
  - Share guarantee
  - Strategy-proofness

- Can we generalize max-min fairness to multiple resources?

- **Proposed Solution**: Dominant Resource Fairness (DRF)

# Dominant Resource Fairness (DRF) (1/2)

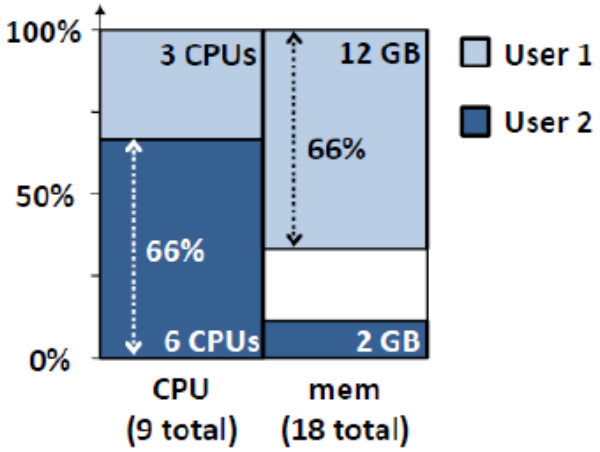
- Dominant resource of a user: the resource that user has the biggest share of.
  - Total resources: <8CPU; 5GB>
  - User 1 allocation: <2CPU; 1GB>, 2/8 = 25%CPU and 1/5 = 20%RAM
  - Dominant resource of User 1 is CPU (25% > 20%)

- Dominant share of a user: the fraction of the dominant resource she is allocated.
  - User 1 dominant share is 25%.

# Dominant Resource Fairness (DRF) (2/2)

- Apply max-min fairness to dominant shares: give every user an equal share of her dominant resource.
- Equalize the dominant share of the users.
  - Total resources: <9CPU; 18GB>
  - User 1 wants <1CPU; 4GB>; Dominant resource: RAM $1/9 < 4/18$
  - User 2 wants <3CPU; 1GB>; Dominant resource: CPU $3/9 > 1/18$

- $\max(x, y)$
  $x + 3y \leq 9$
  $4x + y \leq 18$
  $\frac{4x}{18} = \frac{3y}{9}$
  User 1: $x = 3$: $\langle 33\%CPU, 66\%GB \rangle$
  User 2: $y = 2$: $\langle 66\%CPU, 16\%GB \rangle$



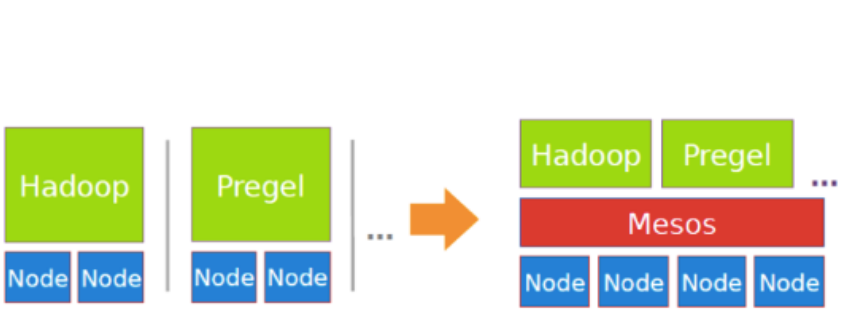
# Online DRF Scheduler

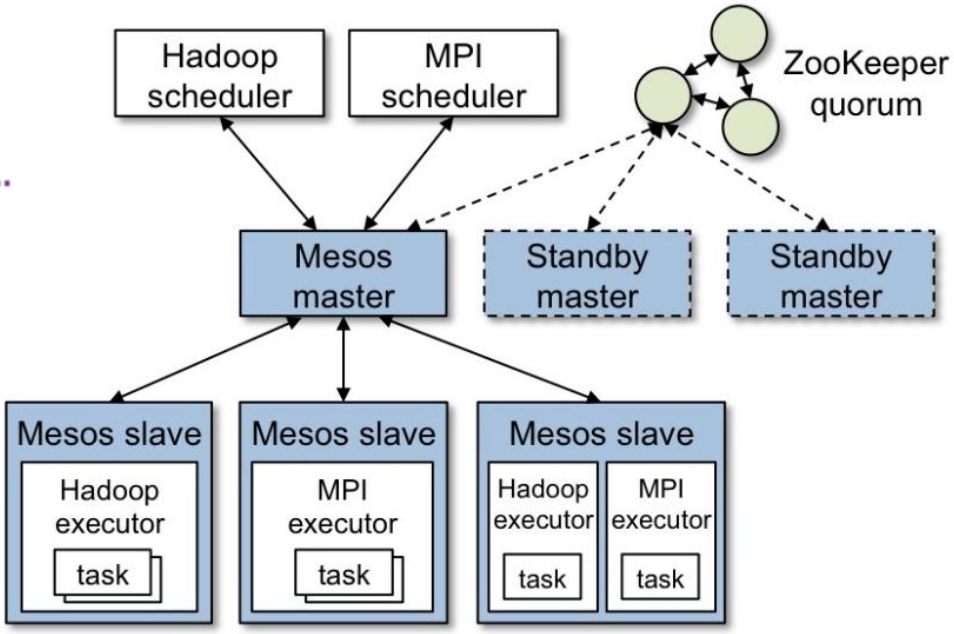
- Whenever there are available resources and tasks to run:
  - Schedule a task to the user with the smallest dominant share.

# 资源调度工具——Mesos

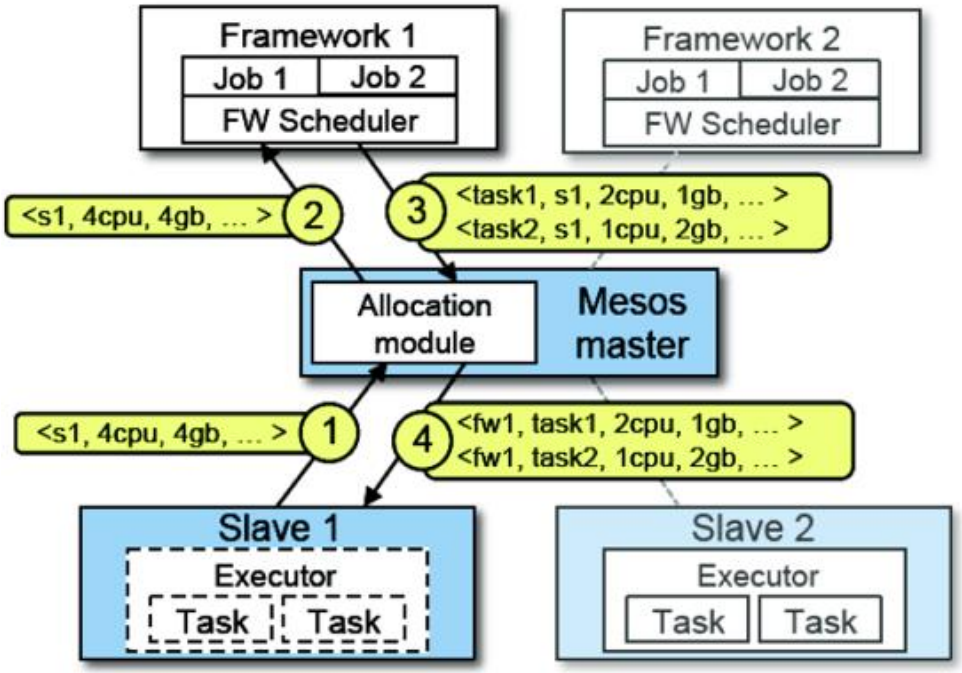
一个通用的、底层资源调度工具

在其上，可以运行各种任务调度器



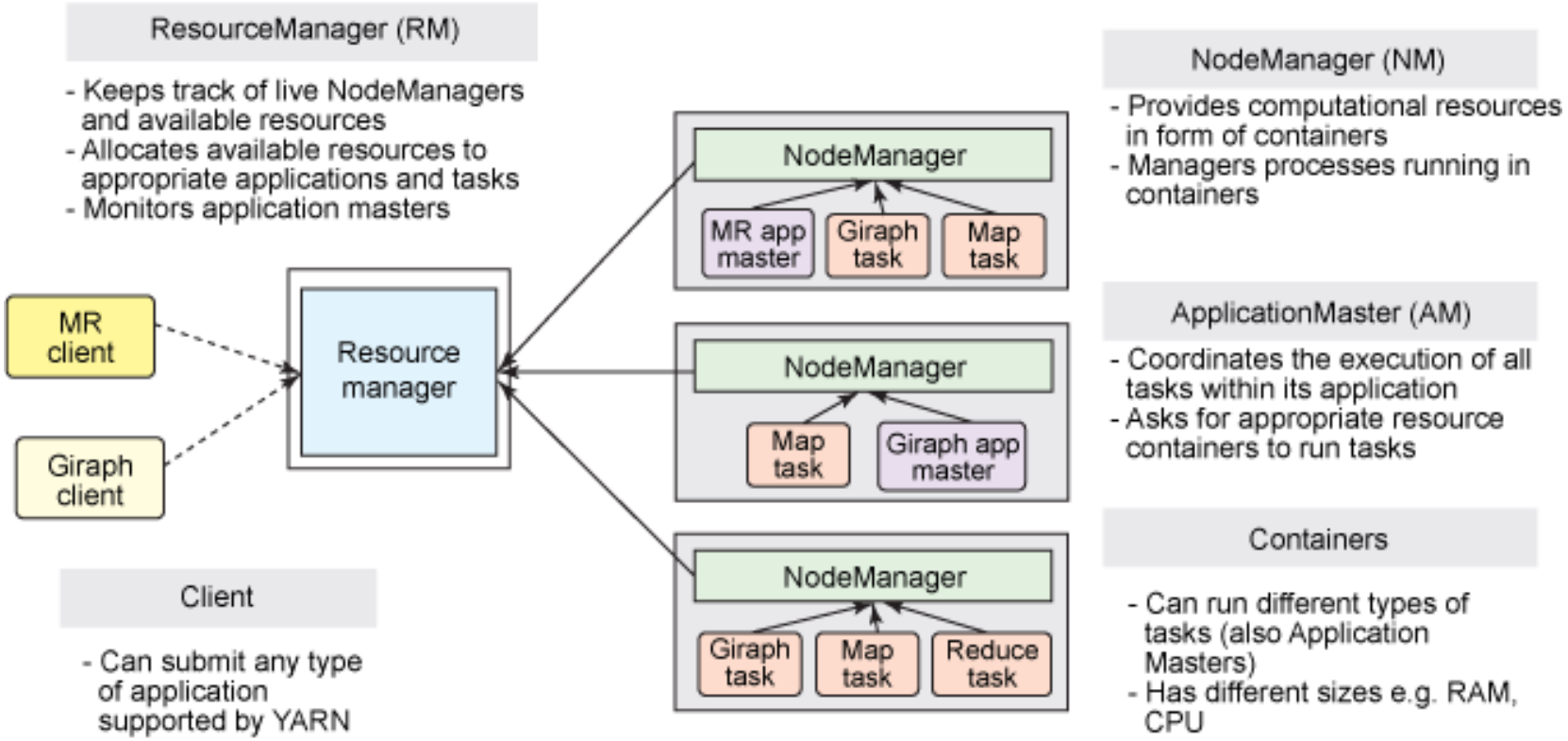



# 资源调度工具——Mesos




1. Slaver向Master汇报它的资源现状。
2. 由Master结点来决定要向哪个Framework给Offer
3. Framework接到这个Offer之后。开始考虑是否要接受这个Resource offer，并且接受多少。然后得出结论，也是给出一组向量<task n ,xxCPU, xxxGB>.
4. 然后经过Mesos的转发，到那个Slaver里面的相应的Framework的Executor来创建出相应的Task

# 资源调度工具——Yarn


ResourceManager (RM)
- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters
MR client
Giraph client
Resource manager
Client
- Can submit any type of application supported by YARN
NodeManager
MR app master
Giraph task
Map task
NodeManager
Map task
Giraph app master
NodeManager
Giraph task
Map task
Reduce task
NodeManager (NM)
- Provides computational resources in form of containers
- Managers processes running in containers
ApplicationMaster (AM)
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks
Containers
- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

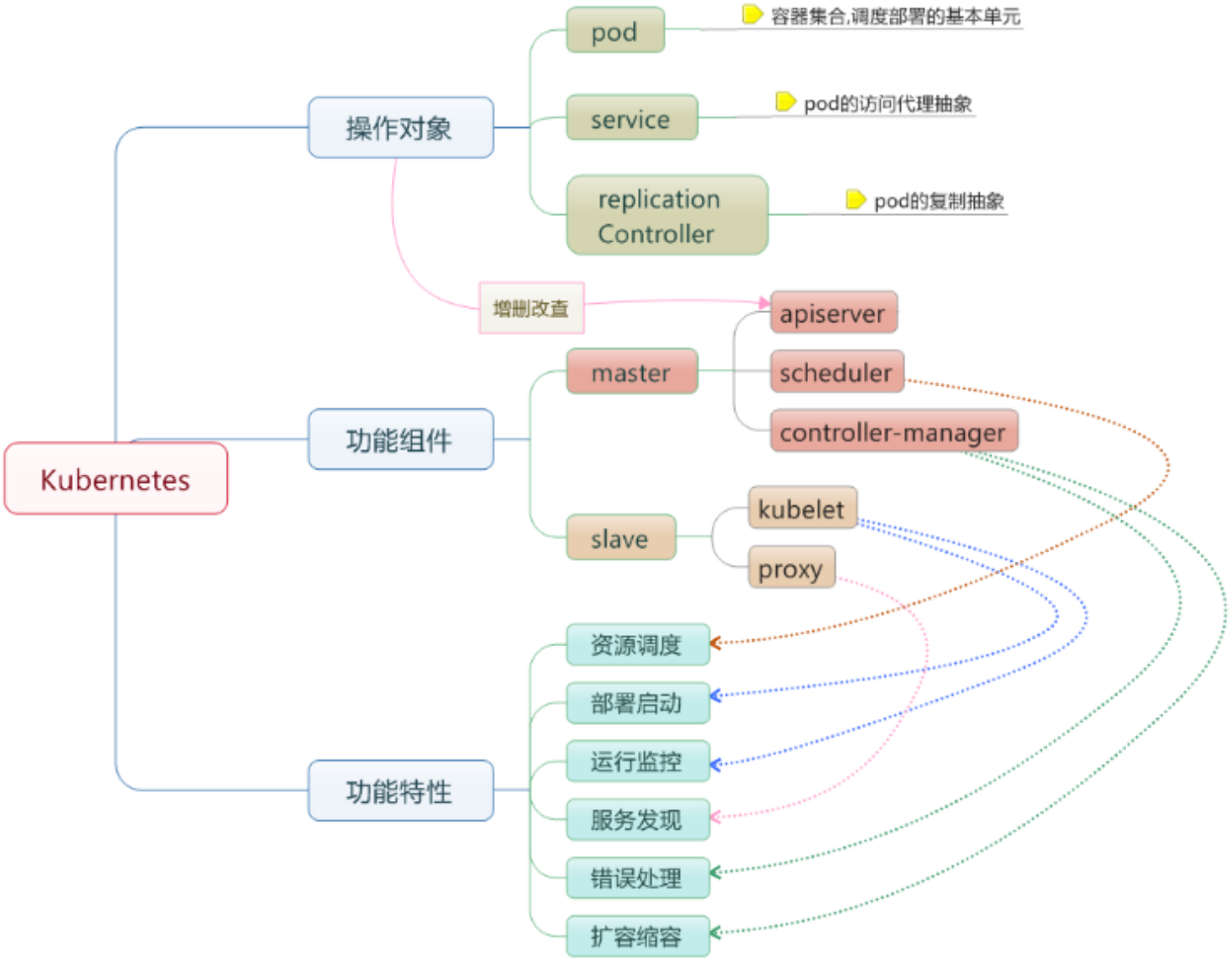
# 资源调度工具—Kubernetes

- Kubernetes 是 Docker生态圈中重要一员，前身是 Borg，是Google大规模容器管理技术的开源版本。是一套自动化容器管理平台提供应用部署、维护、 扩展机制等功能，用于容器的部署、自动化调度和集群管理。

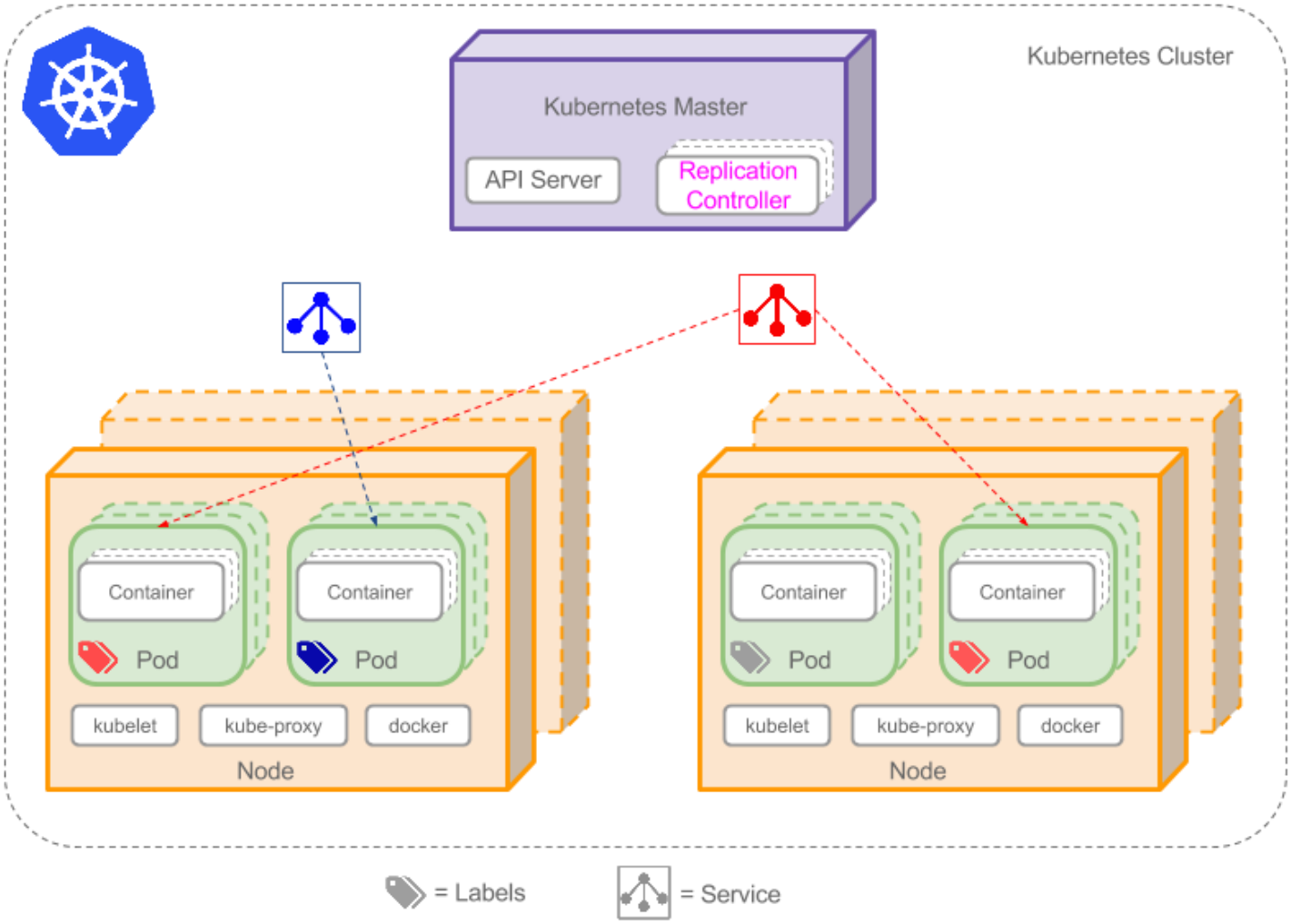
- 目前 kubernetes 有以下的特性：

- 容器的自动化部署，自动化扩展或者缩容
- 容器成组，对外提供服务，支持负载均衡
- 服务的健康检查，自动重启
- 以集群的方式运行、管理跨机器的容器
- 解决Docker跨机器容器之间的通讯问题

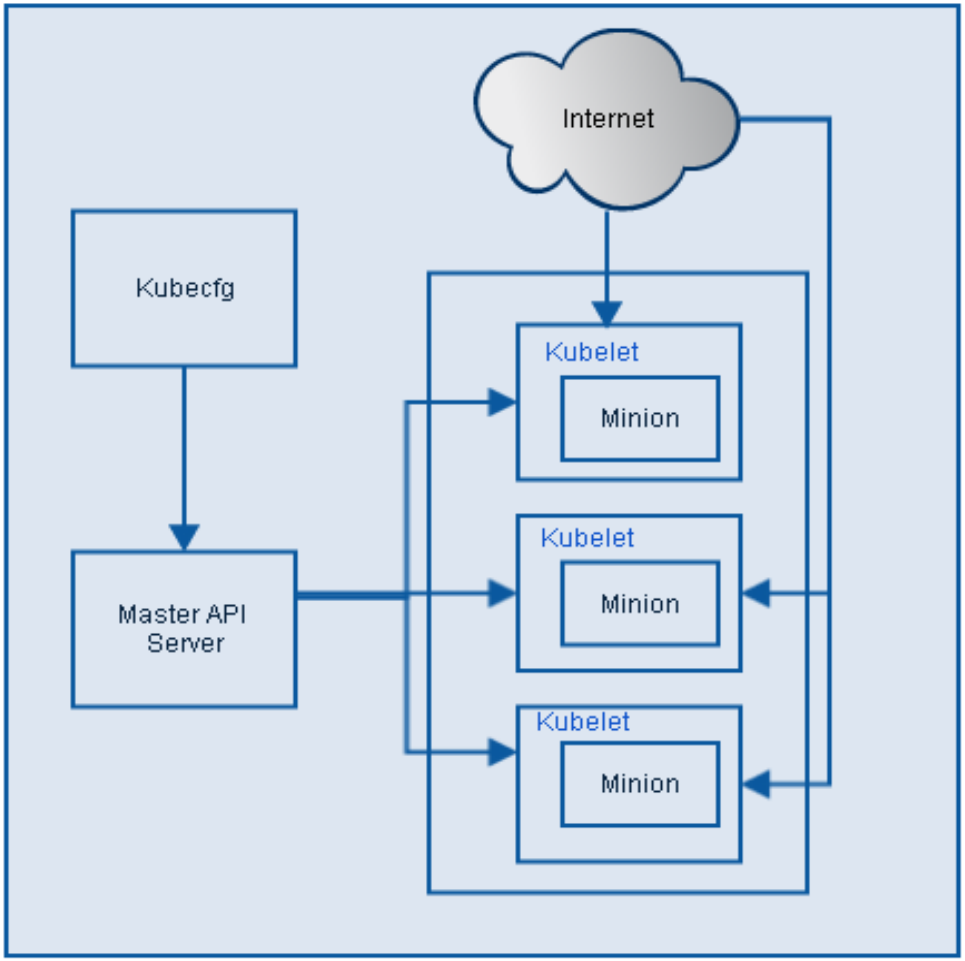
# Kubernetes


Kubernetes
操作对象
pod
容器集合,调度部署的基本单元
service
pod的访问代理抽象
replication Controller
pod的复制抽象
增删改查
apiserver
功能组件
master
scheduler
controller-manager
slave
kubelet
proxy
功能特性
资源调度
部署启动
运行监控
服务发现
错误处理
扩容缩容

# Kubernetes


Kubernetes Cluster
Kubernetes Master
API Server
Replication Controller
Container
Container
Pod
Pod
kubelet
kube-proxy
docker
Node
Container
Container
Pod
Pod
kubelet
kube-proxy
docker
Node
= Labels
= Service

# Kubernetes构件




主要包括

kubecfg

Master API Server

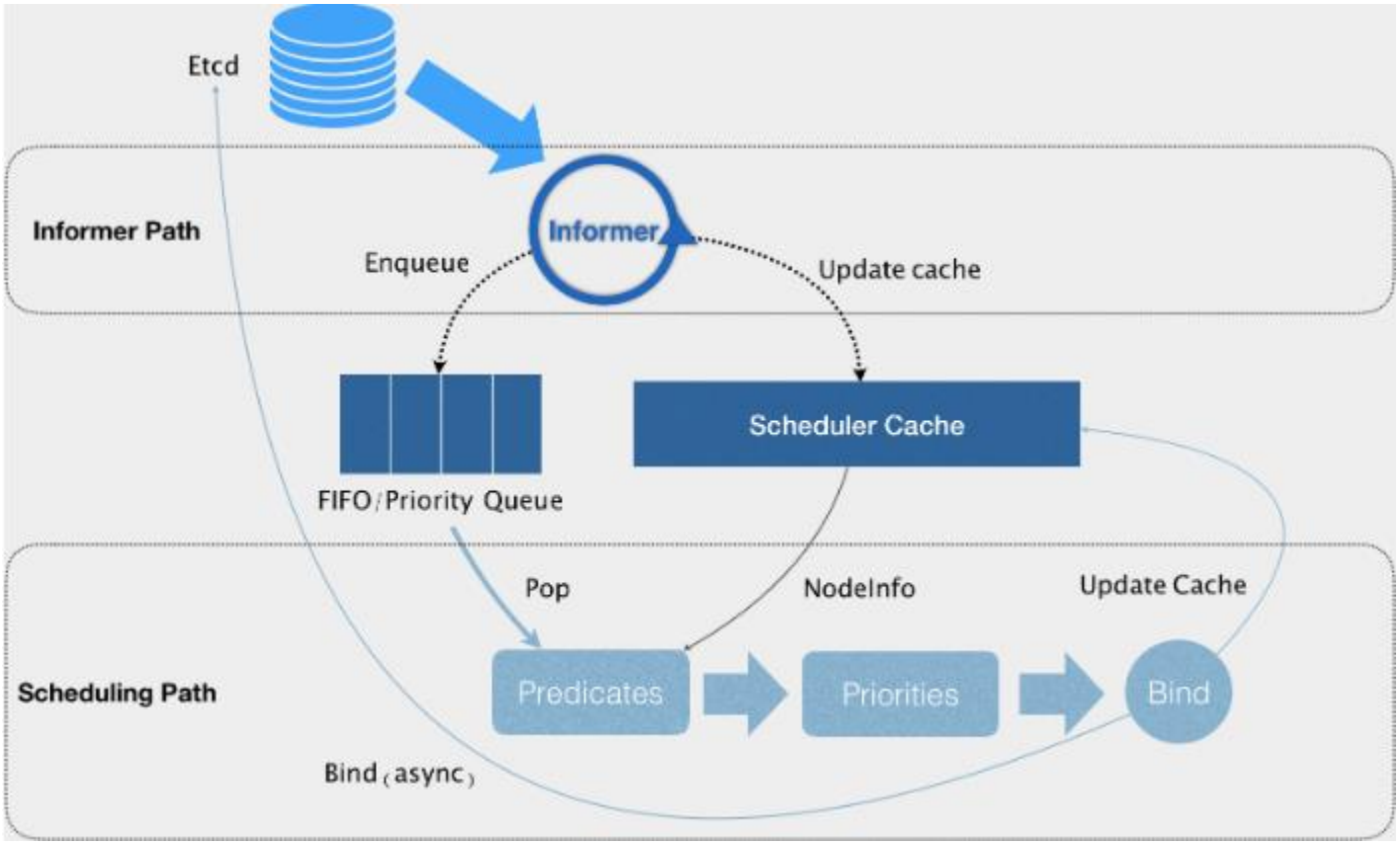
Kubelet

Minion(Host)

Proxy

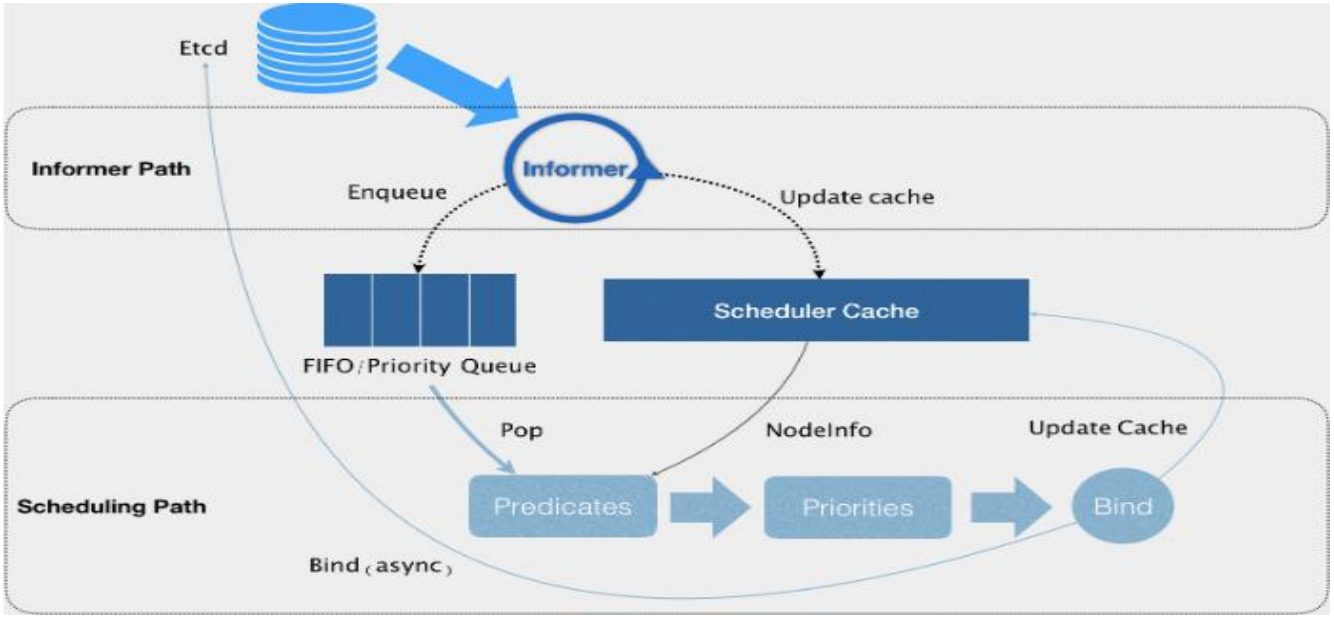
# Kubernetes的资源调度

调度在Kubernetes中的定义： Pod到节点的映射




Kubernetes的调度器的核心，实际上就是两个**相互独立的控制循环**。

# Kubernetes的调度




控制循环1——Informer Path：启动一系列informer，监听etcd中与调度相关的API的对象的变化。当一个待调度的Pod被创建时，将被监听，放入调度队列。

控制循环2——Scheduling Path：不断从调度队列中出队Pod，使用Predicate算法过滤出可以运行该Pod的Node。再使用Priorities算法对节点进行打分，调度到得分最高的节点。

# 本章小节

- 数据中心的管理运维涉及诸多内容
- 数据中心网络
- 资源池化
- 系统/资源监控
- 任务/服务调度