Kubernetes中的资源管理

调度器设计和资源Quality of Service

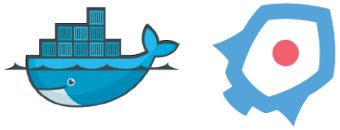
丁海洋（dinghaiyang@huawei.com）
华为技术有限公司
北京，2015.10.31

主要内容

- Kubernetes的调度器设计
- Kubernetes的资源Quality of Service（QoS）
- Kubernetes目前的一些问题

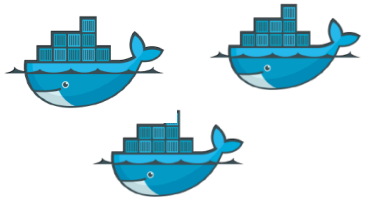
Kubernetes（K8S）中的主要概念

Container



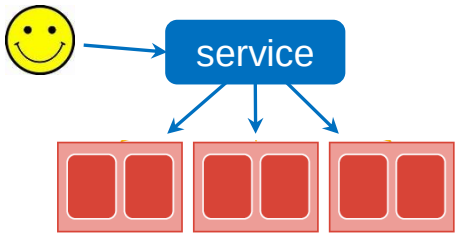
容器，目前主流是Docker。未来K8S也将全面支持RKT（CoreOS）

Pod



一组紧耦合的容器。共享IP、存储等，是K8S中部署的最小单元

Service*

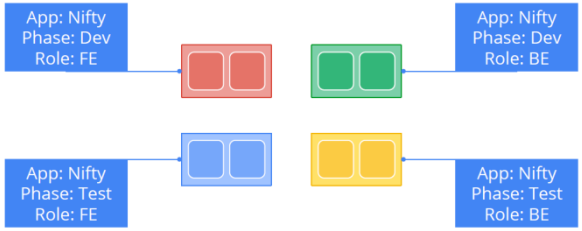


一组提供相同服务的Pod共同连接的一个逻辑层。服务的使用者直接访问Service，而不是具体的Pod

Controller*

以“观察-对比-行动”保证目标对象符合用户的期望。例如Replication Controller，保证群组中Pod数目稳定

Label&
Selector*



K8S中的一切对象都可以标记Label，通过Selector可以灵活的对API对象进行组合

*图片来源：Kubernetes: Architecture and Design, by Tim Hocking, Google

资源管理

资源管理从2个角度看：

- 从应用的角度，为每个Pod寻找合适的部署节点，保证用户体验
  - ✓ 调度器（Scheduler）
  - ✓ 资源QoS（Resource Quality of Service）
  - ✓ 自动弹性（横向/纵向）（Auto-Scaling, Vertical/Horizontal）
- 从集群的角度，提高资源利用率，控制租户的资源配额
  - ✓ 资源超售（ Oversubscription ）
  - ✓ 多租户资源配额管理（LimitRange and Quota）
  - ✓ 资源再平衡（ Rescheduling ）
  - ✓ 实时监控

主要内容

- Kubernetes的调度器设计

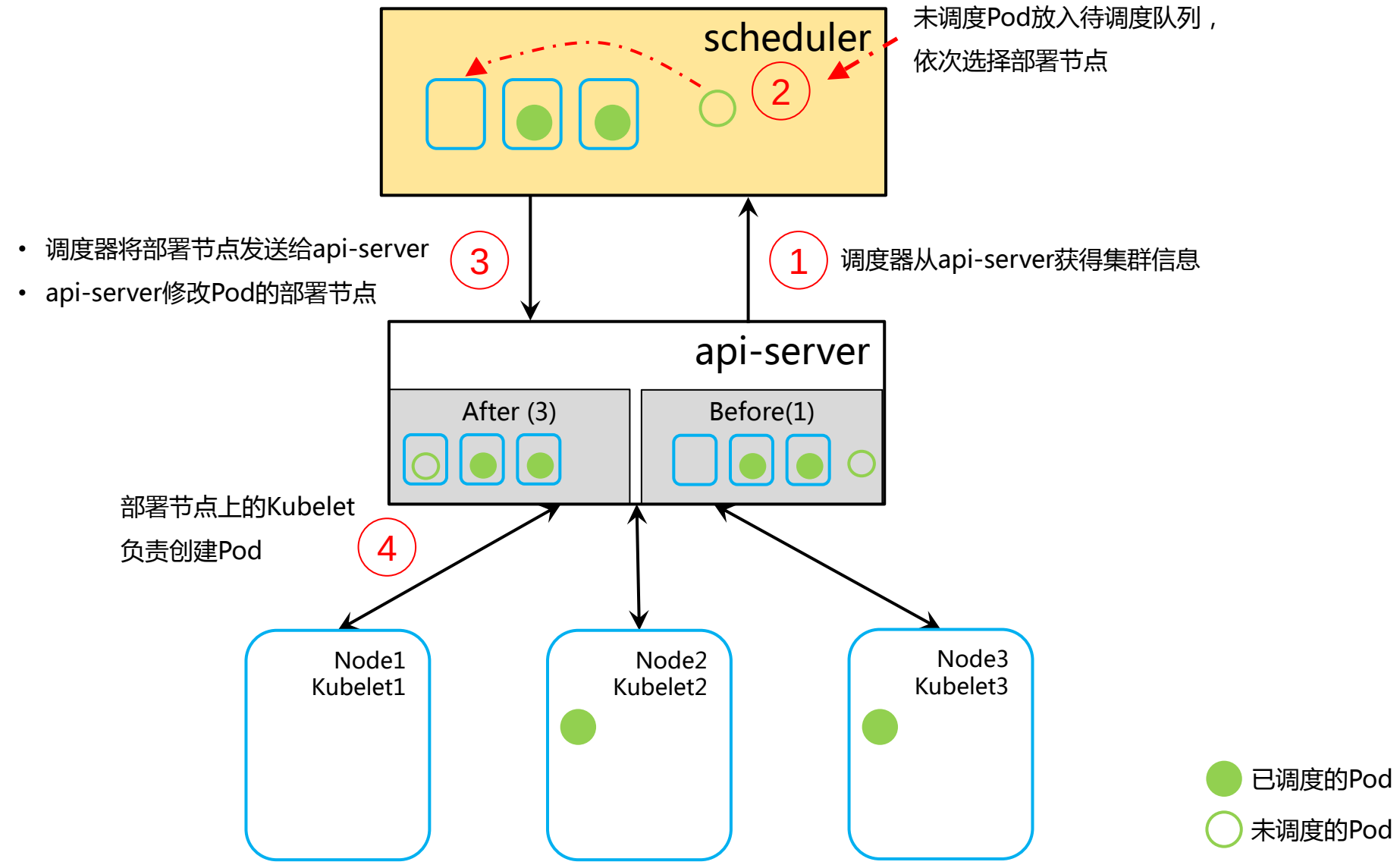
- Kubernetes的资源Quality of Service（QoS）

- Kubernetes目前的一些问题

调度器 我们需要什么样的调度器？

- 把Pod放到合适的节点上，但什么才是“合适”的？
  - ✓ Pod基本的资源需求
  - ✓ Pod的约束条件：多维度的亲和与反亲和（Affinity & Anti-Affinity）
    - 服务分散、跨容灾区域的应用实例、Pod部署到专属节点、Pod部署到已存在镜像的节点等等
  - ✓ 集群整体的资源利用率，资源平衡
- 业务上的需求
  - ✓ 调度的效率，调度的决策要快
  - ✓ 能够为不同类型的应用（Pod）设置不同的调度规则
- 调度器的架构优化
  - ✓ 降低api-server的处理压力：调度器与api-server解耦，作为独立的进程异步运行
  - ✓ 插件式的调度器设计，方便用户添加自定义的调度规则

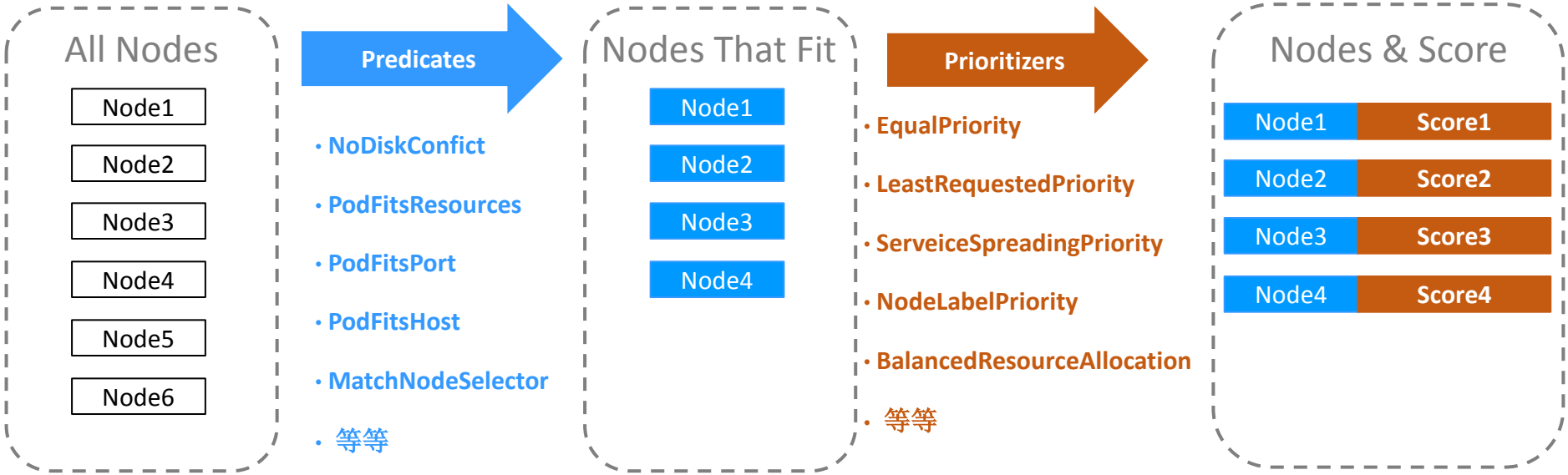
调度器架构与基本调度过程



调度器节点选择的过程

为待部署的Pod选择节点包括过滤和打分两个步骤：

- 过滤：
  - ✓ 用一系列过滤函数（predicate），把不满足Pod基本部署条件的节点筛选出去
  - ✓ 资源剩余、部署约束（Label检查，服务分散等）
- 打分：
  - ✓ 用一系列打分函数（prioritizer），为满足条件的节点打分，选择分数最高的
  - ✓ 资源剩余、资源平衡、服务分散、Label检查等等



调度器目前全部的Predicate和Prioritizer列表

Predicate函数列表

函数名	功能
NoDiskConflict	检查Node是否可以满足Pod对硬盘的需求
PodMatchesNodeLabels	检查Node的Label是否满足Pod的要求
CheckNodeLabelPresence	检查Node是否含有（或不含有）Pod指定的Label
PodFitsHost	检查Pod是否指定了具体的部署Node
PodFitsHostPorts	Pod要求的Node端口是否已经被占用
PodFitsResources	检查Node是否满足Pod的资源需求

Prioritizer函数列表

函数名	功能
NodeLabelPriority	检查Node是否存在Pod需要的Label，有则10分，没有则0分
BalanceResourceAllocation	选择在部署后CPU和内存占用率相近的Node
Spreading	同一个Service下的Pod尽可能的分散在集群中。Node上运行的通Service下的Pod数目越少，分数越高。
Anti-Affinity	与Spreading类似，但是分散在拥有特定Label的所有Node中
LeastRequestPriority	**将Pod部署到剩余CPU和内存最多的Node上**

如何衡量剩余资源“最多”？

LeastRequestPriority在计算时考虑容器 *“需要”* 的资源的多少，与 *“需要”* 相对应的的是对资源的 *“限制”* 。

这就涉及到K8s中资源Quality of Service的设计问题

主要内容

- Kubernetes的调度器设计

➢Kubernetes的资源Quality of Service（QoS）

- Kubernetes目前的一些问题

资源的Quality of Service(QoS)

- 实践中，容器根据其承载的应用不同可以粗略的分成两类（Google，Borg）：
  - 重要的：Long Running Service（Borg中的prod）
    - ✓ 长时间运行的，例如Web服务
    - ✓ 延时十分敏感的，例如需要快速响应用户请求的应用
    - ✓ 生产环境的监控、控制进程等
  - 相对不重要的：Batch jobs（Borg中的non-prod）
    - ✓ 仅仅运行一段时间便会停止，而且对计算时间不敏感的，例如Map-Reduce等计算类任务
- K8S中，为容器划分了3个优先级类型，这种优先级是通过资源的QoS定义的
  - Guaranteed，拥有最高的优先级，类似prod
  - Best-Effort，拥有最低的优先级，类似non-prod*
  - Burstable，优先级介于以上两者之间

资源QoS是根据对资源的“需要”和“限制”定义的，为什么呢？

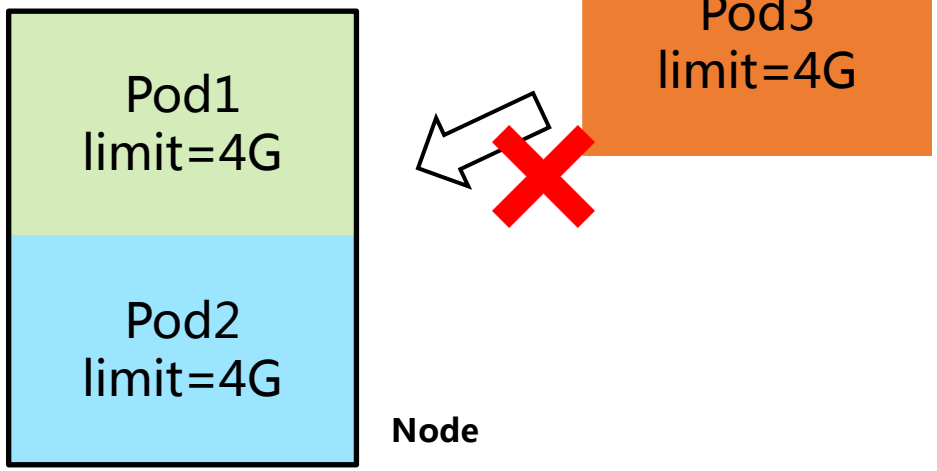
*【为防止误解】实际上Borg中对容器优先级的分类要细致很多。prod和non-pro仅仅是两个大类的名称

资源的Quality of Service

为什么使用request和limit

- 容器的cgroup可以为容器设置资源使用的上限，即限制的资源（ limit ）
- 但是仅靠limit来调度容器（Pod），资源使用率很低。为什么？

  - Node共有8G内存
  - 两个Pod都有可能使用到上限。如果部署Pod3，有其自己的上限（4G），那么当3个Pod都按照上限需求时，减少谁的需求呢？
  - 问题：无法保证任意Pod/容器资源使用的下限，因而不能部署Pod3



资源的Quality of Service

为什么使用request和limit

- 容器的cgroup可以为容器设置资源使用的上限，即限制的资源（ limit ）
- 但是仅靠limit来调度容器（Pod），资源使用率很低。为什么？

- Node共有8G内存
- 两个Pod都有可能使用到上限。如果部署Pod3，有其自己的上限（4G），那么当3个Pod都按照上限需求时，减少谁的需求呢？
- 问题：无法保证任意Pod/容器资源使用的下限，因而不能部署Pod3

- 但是，通常各个Pod不能使用到limit的资源，空闲的资源就浪费了，降低了资源使用率
- 解决方案：为每个容器（Pod）添加资源使用的下限，即，在需要的时候，容器（Pod）会保证得到这个下限数量的资源，称为“需求（request）”

资源的Quality of Service

为什么使用request和limit

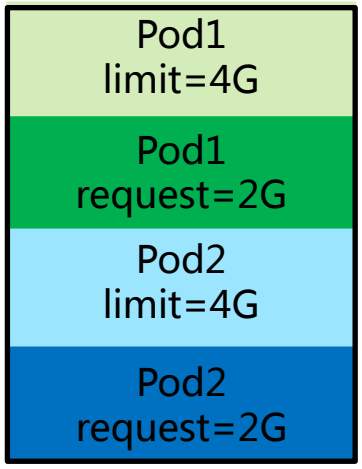
- 容器的cgroup可以为容器设置资源使用的上限，即限制的资源（ limit ）
- 但是仅靠limit来调度容器（Pod），资源使用率很低。为什么？

- Node共有8G内存
- 两个Pod都有可能使用到上限。如果部署Pod3，有其自己的上限（4G），那么当3个Pod都按照上限需求时，减少谁的需求呢？
- 问题：无法保证任意Pod/容器资源使用的下限，因而不能部署Pod3

- 但是，通常各个Pod不能使用到limit的资源，空闲的资源就浪费了，降低了资源使用率
- 解决方案：为每个容器（Pod）添加资源使用的下限，即，在需要的时候，容器（Pod）会保证得到这个下限数量的资源，称为“需求（request）”

如何使用request

- request<= limit
- request表示容器（Pod）可以获取资源的下限（除非有系统级别的应用所要资源）
- 对于有资源request的Pod，其【可用资源】=【节点总资源】-【已部署Pod的总request】



被其它具有request的Pod视为可用资源

- 调度时保证所有Pod的request的和不超过节点总资源
- 调度不考虑limit
- 问题：尚不能考虑资源的实时使用

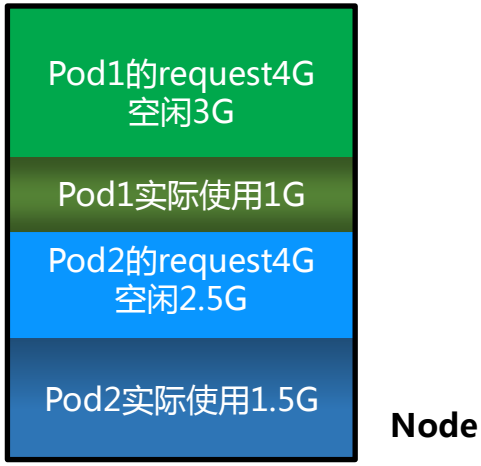
资源的Quality of Service

更进一步的资源超售

- 如果节点已经被request塞满了，还能部署部署Pod吗？
- 如果能，怎样保证所有Pod的request在需要时都可以被满足？

答案是：能

- 可以被request=0的Pod占用
- 需要对资源使用进行实时监控



- 定义容器的优先级
- 可以kill低优先级的容器为高优先级的容器释放资源
- request=0的容器的优先级最低，但是却承担着使用集群空闲资源、提高资源利用率的重要任务

小节：

- 仅使用limit资源利用率很低，不能达到超售的目的
- 为了资源超售，需要为容器（Pod）定义资源的request和limit
- 为了保证相对重要的容器（Pod）能够按需使用资源，根据资源的QoS类型定义划分优先级

资源的Quality of Service

总结：K8S中为容器申请的资源设置了三个类型的QoS，优先级从高到低：

Guaranteed > Burstable > Best-Effort（再次提醒，真的不是中文“尽全力”的意思！）

K8S中QoS分类简介：

优先级类/QoS Class	特点	划分依据
Guaranteed	• 最高优先级，最后考虑被杀掉的Container • 调度时考虑节点上其它Pod的request • 如果试图使用超过limit的资源，会被kill	request==limit!=0
Burstable	• 次高优先级，通过杀掉Best-Effort获取资源 • 调度时考虑节点上其它Pod的request • 如果试图使用超过request的资源，可能被kill • 如果试图使用超过limit的资源，会被kill	0<request<limit<∞
Best-Effort	• 最低优先级 • 高优先级的两类Pod在无法使用request要求的资源时，此类容器首先被kill • 调度时不考虑节点中其它Pod的request，而是考虑节点资源的实时使用情况（待实装）	request==0

通过资源QoS，K8S实现了：

- 资源超售，提高了集群的资源利用率
- 保证重要的容器能随时获得需要的资源
- 其它：
  - ✓ 容器的out-of-resource killing是在Kubelet端实现的
  - ✓ 通过资源QoS定义容器的优先级是K8S采用的方法，实际上优先级不是一定要和资源QoS联系起来
  - ✓ 目前K8S的QoS设计和开发还在进行中，见后面分析·

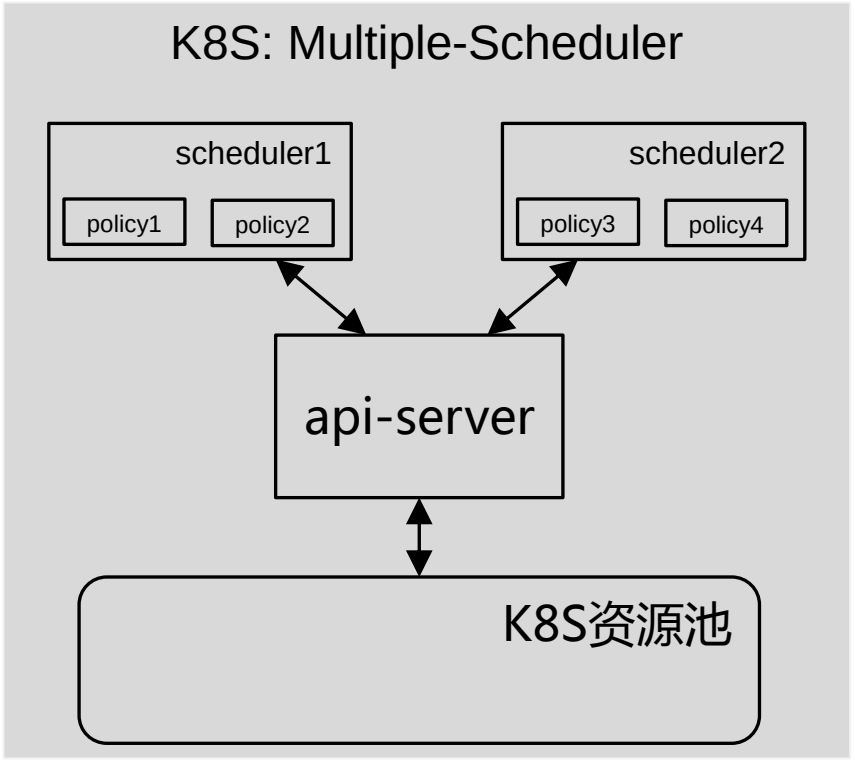
主要内容

- Kubernetes的调度器设计
- Kubernetes的资源Quality of Service（QoS）
- ➢Kubernetes目前的一些问题

Kubernetes目前的问题

调度器存在的一些问题：

- 一个集群中只有一个调度器 → 未来支持多调度器
- 一个调度器只能有一个调度规则 → 我们希望未来一个调度器可以支持多个调度规则
- 更换调度器或者调度规则需要重新启动调度器 → 动态启动新的调度器
- 支持的约束类型不够多
  - 指定不同类型Pod之间的反亲和
  - 某类Node仅接受特定类型Pod的部署请求
  - 更多维度的亲和与反亲和
  - “Not in” Selector等等



Kubernetes目前的问题

资源QoS存在的一些问题：

PS：目前资源QoS还在不断的开发中

- 尚不支持RKT
- 目前仅支持Memory，一般不会因为CPU使用量选择kill容器
- QoS目前定义在计算资源层面，未来应该定义在容器甚至是Pod层面
- 不支持使用实时的资源使用情况调度Pod，Best-Effort容器（Pod）不能完全达到使用空闲资源的目的
- 一些更好的Kubelet端资源控制和kill容器的方式，例如：
  - 两个Burstable竞争资源，一个超过了Request，而另一个还没达到Request（部署的比较晚），目前的方式是kill（并restart）超过了Request的容器，但是这个方案过于昂贵，且可能产生链式反应
  - 一个Pod中的容器被kill，整个Pod是否应该重启
  - Burstable被kill后允许重新启动一定的次数
  - 等

华为与Kubernetes

- K8S是容器集群管理的领导者，我们会持续在社区中进行贡献
- 与社区和Google建立良好的合作关系
- 主导一些重要特性的开发

#	Company	Commits
1	Google	6254
	*independent	2087
2	Red Hat	2070
3	FathomDB	318
4	Huawei	263
5	CoreOS	184
6	Zhejiang University	76
7	Canonical	50
8	Amadeus	49
9	VMware	25

谢谢