模块9——生产化集群的管理

操作系统的选择与评估

	CentOS	Ubuntu	CoreOS	Atomic*	Snappy	RancherOS
生态系统和成熟度	通用操作系统		容器优化			
	成熟		最早的容器优化操作系统，不过公司体量小，目前已被收购。	Red Hat出品，品质保证	Canonical 出品，最初为移动设备设计	相对较新，RancherOS中运行的所有服务都是docker 容器。

云原生的原则

不可变基础设施。

可变基础设施的风险

- 在灾难发生的时候，难以重新构建服务。持续过多的手工操作，缺乏记录，会导致很难由标准初始化后的服务器来重新构建起等效的服务。
- 在服务运行过程中，持续的修改服务器，就犹如程序中的可变变量的值发生变化而引入的状态不一致的并发风险。这些对于服务器的修改，同样会引入中间状态，从而导致不可预知的问题。

不可变基础设施

- 不可变的容器镜像
- 不可变的主机操作系统

Atomic

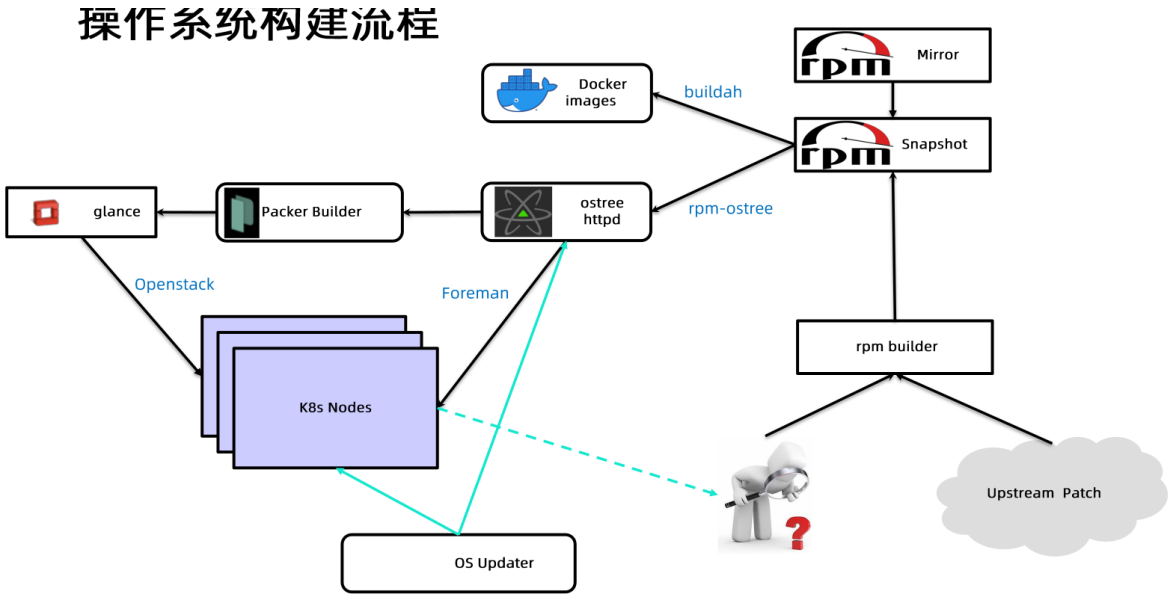
- 由Red Hat支持的软件包安装系统
- 多种Distro
    - Fedora
    - CentOS
    - RHEL
- 优势
    - 不可变操作系统，面向容器优化的基础设施
        - 灵活和安全性较好
        - 只有/etc和 /var可以修改，其他目录均为只读
    - 基于rpm-ostree管理系统包
        - rpm-ostree是一个开源项目，使得生产系统中构建镜像非常简单
        - 支持操作系统升级和回滚的原子操作

最小化主机操作系统

原则：

- 最小化主机操作系统
- 只安装必要的工具
    - 必要：支持系统运行的最小工具集
    - 任何调试工具，比如性能排查，网络排查工具，均可以后期以容器形式运行
- 意义
    - 性能
    - 稳定性
    - 安全保障

- 构建流程



- 操作系统加载
    - 物理机
        - 物理机通常需要通过foreman启动，foreman通过pxe boot，并加载kickstart
        - kickstart通过ostree deploy即可完成操作系统的部署
    - 虚拟机
        - 需要通过镜像工具将ostree构建成qcow2格式，vhd，raw等模式

节点资源管理

状态上报

kubelet周期性地向API Server进行汇报，并更新节点的相关健康和资源使用信息

- 节点基础信息，包括IP地址、操作系统、内核、运行时、kubelet、kube-proxy版本信息。
- 节点资源信息包括CPU、内存、HugePage、临时存储、GPU等注册设备，以及这些资源中可以分配给容器使用的部分。
- 调度器在为Pod选择节点时会将机器的状态信息作为依据。

状态	状态的意义
Ready	节点是否健康
MemoryPressure	节点是否存在内存压力
PIDPressure	节点是否存在比较多的进程
DiskPressure	节点是否存在磁盘压力
NetworkUnavailable	节点网络配置是否正确

```
$ k get no
NAME                          STATUS   ROLES    AGE   VERSION
xiaokai-thinkpad-x13-gen-1    Ready    master   12d   v1.19.15

$ k get no xiaokai-thinkpad-x13-gen-1  -oyaml
```

上述命令可以得到：

- 节点基础信息如下：

```
status:
  addresses:
  - address: 192.168.2.8
    type: InternalIP
  - address: xiaokai-thinkpad-x13-gen-1
    type: Hostname
.....
  nodeInfo:
    architecture: amd64
    bootID: 712dae7f-8c9a-457b-824b-f258986db202
    containerRuntimeVersion: containerd://1.4.12
    kernelVersion: 5.11.0-41-generic
    kubeProxyVersion: v1.19.15
    kubeletVersion: v1.19.15
    machineID: 88962199eaea405198251d6ad25f71d4
    operatingSystem: linux
    osImage: Ubuntu 20.04.3 LTS
    systemUUID: b05e164c-22c2-11b2-a85c-c25a60109855
```

- 节点资源信息

```
status:
  ...
  allocatable:
    cpu: "16"
    ephemeral-storage: "37805329551"
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 15484132Ki
    pods: "110"
```

```
  capacity:
    cpu: "16"
    ephemeral-storage: 41021408Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 15586532Ki
    pods: "110"
```

其中capacity的获取方法如下：

- CPU是从/proc/cpuinfo文件中获取的节点CPU核数
- memery是从/proc/meminfo中获取的节点内存大小
- ephemeral-storage是指节点跟分区的大小
- allocatable和capacity的关系如下：

节点的资源容量(Capacity)			
kube-reserved	system-reserved	eviction-threshold	Allocatable

> 可以利用status/capacity配置扩展资源比如GPU等，详见“基于extended resource 扩展节点资源”。

- 节点状态信息

```
  conditions:
  - lastHeartbeatTime: "2021-12-06T04:05:10Z"
    lastTransitionTime: "2021-12-06T04:05:10Z"
    message: Cilium is running on this node
    reason: CiliumIsUp
    status: "False"
    type: NetworkUnavailable
  - lastHeartbeatTime: "2021-12-06T04:24:57Z"
    lastTransitionTime: "2021-12-06T04:04:54Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"
    type: MemoryPressure
  - lastHeartbeatTime: "2021-12-06T04:24:57Z"
    lastTransitionTime: "2021-12-06T04:04:54Z"
    message: kubelet has no disk pressure
    reason: KubeletHasNoDiskPressure
    status: "False"
    type: DiskPressure
  - lastHeartbeatTime: "2021-12-06T04:24:57Z"
    lastTransitionTime: "2021-12-06T04:04:54Z"
    message: kubelet has sufficient PID available
    reason: KubeletHasSufficientPID
    status: "False"
    type: PIDPressure
  - lastHeartbeatTime: "2021-12-06T04:24:57Z"
    lastTransitionTime: "2021-12-06T04:04:54Z"
    message: kubelet is posting ready status. AppArmor enabled
    reason: KubeletReady
    status: "True"
    type: Ready
```

Lease对象“映射”节点状态

早起K8S版本直接上报node对象的状态信息和资源信息，数据包较大，对API SERVER和ETCD造成较大压力。后来引入Lease对象，专门用于**保存**或者说代表节点的健康状态信息（资源信息可能不在持续上报了，因为资源信息的变化频率很低），在默认40s的nodeLeaseDurationSeconds周期内，若Lease对象没有被更新，则对应节点可以判断为不健康。

> 这个原理就和zookeeper的选主过程很像。

```
$ k get lease -A
NAMESPACE         NAME                                HOLDER
                                      AGE
kube-node-lease   xiaokai-thinkpad-x13-gen-1      xiaokai
               13d
kube-system       cilium-operator-resource-lock   xiaokai-EEQPEqFQCm
                13d
kube-system       kube-controller-manager         xiaokai_9bd0b700-7f27-4329-
ac7d-bfeffefaaefa   13d
kube-system       kube-scheduler                  xiaokai_b4193885-cae5-4992-
b710-a4c92e2935c9   13d
tigera-operator   operator-lock                   xiaokai_77f64058-f298-4104-
a61a-577f0c0d3eb4   13d

$ k get lease xiaokai-thinkpad-x13-gen-1 -n kube-node-lease -oyaml
apiVersion: coordination.k8s.io/v1
kind: Lease
metadata:
  creationTimestamp: "2021-11-23T15:04:25Z"
  managedFields:
  - apiVersion: coordination.k8s.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:ownerReferences:
          .: {}
          k:{"uid":"2c44399a-0075-41a0-841a-5ed3c78a2b0b"}:
            .: {}
            f:apiVersion: {}
            f:kind: {}
            f:name: {}
            f:uid: {}
      f:spec:
        f:holderIdentity: {}
        f:leaseDurationSeconds: {}
        f:renewTime: {}
    manager: kubelet
    operation: Update
    time: "2021-11-23T15:04:25Z"
  name: xiaokai-thinkpad-x13-gen-1
  namespace: kube-node-lease
  ownerReferences:
  - apiVersion: v1
    kind: Node
    name: xiaokai-thinkpad-x13-gen-1
    uid: 2c44399a-0075-41a0-841a-5ed3c78a2b0b
```

```
  resourceVersion: "1034186"
  selfLink: /apis/coordination.k8s.io/v1/namespaces/kube-node-
lease/leases/xiaokai-thinkpad-x13-gen-1
  uid: 2bf82b8a-3f04-4c19-9e01-181b4e2c8390
spec:
  holderIdentity: xiaokai-thinkpad-x13-gen-1  #持有人
  leaseDurationSeconds: 40                    #超时时间
  renewTime: "2021-12-07T04:11:10.363792Z"    #上次更新时间
```

上面最后的三个信息和锁的信息非常像：持有人、超时时间。所以Lease可以代替Configmap来做锁，比如控制器的高可用选主的时候，可以用用获取Lease代表的锁来表示主控制器节点，然后定期更新状态。

使用cAdvisor检查节点资源使用情况

- kubelet依赖内嵌的开源软件cAdvisor，周期性检查节点资源使用情况
- CPU是可压缩资源，根据不同进程分配时间配额和权重，CPU可被多个进程竞相使用
- 驱逐策略是基于磁盘和内存资源用量进行的，因为两者属于不可压缩的资源，当此类资源使用耗尽时将无法再申请

检查类型	说明
memory.available	节点当前的可用内存
nodefs.available	节点根分区的可使用磁盘大小
nodefs.inodesFree	节点根分区的可使用inode
imagefs.inodesFree	节点运行时分区的可使用inode
imagefs.available	节点运行时分区的可使用磁盘大小 节点如果没有运行时分区，就不会有相应的资源监控

节点磁盘管理

- 系统分区nodefs：工作目录和容器日志

  containerd & docker：`/var/lib/kubelet/pods`保存的是kubelet pod的data dir。

```
root@xiaokai-ThinkPad-X13-Gen-1:/var/lib/kubelet/pods# ls -l
总用量 56
drwxr-x--- 5 root root 4096 11月 26 22:49 00d45a3e-b7c7-4dcb-b4a7-
ed84336161cb
drwxr-x--- 5 root root 4096 11月 23 23:08 05ca4d4e-e783-4643-b5a7-
6f9421aea2fa
drwxr-x--- 5 root root 4096 11月 23 23:08 169e1c1b-324e-491c-8d30-
83f07feb1a12
drwxr-x--- 5 root root 4096 11月 23 23:04 9e5d7a5b-43a2-446a-87b1-
174e0d21872a
drwxr-x--- 5 root root 4096 11月 23 23:04 ba4750600ce7fb789689157a298e0ab1
drwxr-x--- 5 root root 4096 11月 23 23:07 be6be363-a180-445d-af9d-
0eefde56cbbc
drwxr-x--- 5 root root 4096 12月  2 13:16 bf3ce899-819a-4660-9479-
6838d30dd80b
drwxr-x--- 5 root root 4096 12月  1 19:59 c661c0f7-dc3d-435f-a74a-
7566846097ad
drwxr-x--- 5 root root 4096 11月 23 23:04 d14e619ba51f275718d8a3b18e2ecadb
drwxr-x--- 5 root root 4096 11月 23 23:07 e54516ff-aaad-4989-847b-
771e1d7ccb09
```

```
drwxr-x--- 5 root root 4096 11月 23 23:04 e5527502293600438c26c80c40af0841
drwxr-x--- 5 root root 4096 11月 23 23:04 f188a592111d286635c4a1b1aee9a30d
drwxr-x--- 5 root root 4096 11月 23 23:08 fb029904-9644-4b3d-bf63-
0ba4c7843a46
drwxr-x--- 5 root root 4096 11月 24 12:15 fc1cd0bf-b6ad-4bff-ba24-
57cd3beb12b4
```

- 容器运行时分区imagefs：用来保存镜像
  - **containerd**：`/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs`
  - **docker**：`/var/lib/docker/overlay2/`

> nodefs和imagefs可以用单独的参数指定，但一般情况下两者不区分，都是在在rootfs中。

驱逐管理

- kubelet会在系统资源不足时终止一些容器进程，以空出系统资源，保证节点的稳定性。
- 但驱逐只是停止Pod的所有容器进程，并不会直接删除POD，而是留下“痕迹”，以便运维人员发现，否则在另外的节点启动，就会隐藏这个问题。需要手工清理一下，但是不会占用资源。Pod的状态会被标记为：
  - Pod的status.phase会被标记为Failed
  - status.reason会被设置为Evicted
  - status.message则会记录被驱逐的原因

驱逐策略

分为软驱逐和硬驱逐，都可以由参数设定。

kubelet获得节点的可用额信息后，会结合节点的容量信息来判断当前节点运行的Pod是否满足驱逐条件。

驱逐条件可以是绝对值或百分比，当监控资源的可使用额少于设定的数值或百分比时，kubelet就会发起驱逐操作。

kubelet参数evictionMinimumReclaim可以设置每次回收的资源的最小值，以防止小资源的多次回收。

kubelet参数	分类	驱逐方式
evictionSoft	软驱逐	当检测到当前资源达到软驱逐的阈值时，并不会立即启动驱逐操作，而是要等待一个宽限期。 这个宽限期选取EvictionSoftGracePeriod和Pod指定的TerminationGracePeriodSeconds中较小的值
evictionHard	硬驱逐	没有宽限期，一旦检测到满足硬驱逐的条件，就直接中止容器来释放紧张资源

设定的参数如下

```yaml
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
evictionHard:
  memory.available: "500Mi"
  nodefs.available: "1Gi"
  imagefs.available: "100Gi"
evictionMinimumReclaim:
  memory.available: "0Mi"
  nodefs.available: "500Mi"
  imagefs.available: "2Gi"
```

基于内存压力的驱逐

kubelet默认设置了memory.avaiable < 100Mi的硬驱逐条件。

1. **先阻止**。将节点的MemoryPressure状态设置为true，调度器阻止BestEffort Pod调度到内存承压节点。
2. **后驱逐**。
    - 判断Pod内所有容器的内存使用量是否超出了请求的内存量，超出请求资源的pod会成为备选目标。
    - 低优先级的pod被优先驱逐。（防止高优先级的pod被驱逐后到其余节点干扰其它Pod）
    - 计算Pod内所有容器的内存使用量和请求量的差值，**差值越大，越容易被驱逐**，所以如下的三个级别的pod中，从上往下被驱逐的可能性逐渐变小。
        - BestEffort
        - Burstable
        - Garunteed

> OOM Killer行为：
>
> - 系统的OOM Killer可能会采取OOM的方式来中止某些容器的进程，进行必要的内存回收操作
> - 而系统根据进程的oom_score来进行优先级排序，选择待终止的进程，且进程的oom_score越高，越容易被终止。
> - 进程的oom_score是根据当前进程使用的内存占节点总内存的比例值乘以10，再加上oom_score_adj综合得到的
> - 而容器进程的oom_score_adj正是kubelet根据memory.request进行设置的
>
> | Pod QoS类型 | oom_score_adj |
> | --- | --- |
> | Guaranteed | -998 |
> | BestEffort | 1000 |
> | Burstable | min(max(2,1000-(1000×memoryRequestBytes)/machineMemoryCapacityBytes), 999) |
>
> 最后一条的意思是申请少而使用多会被优先杀掉。

```
# crictl  ps | grep cardbill
f7ddeaa7b5499       34bb9631698cd       2 hours ago         Running
    cardbill                  6                   9e45f6413aad8

# crictl inspect f7ddeaa7b5499 | grep pid
 "pid": 8750,
         "pid": 1
         "type": "pid"

# cat /proc/8750/oom_score
24
# cat /proc/8750/oom_score_adj
-998
```

基于磁盘压力的驱逐

1. 首先尝试释放磁盘空间

以下任何一项满足驱逐条件时，它会将节点的DiskPressure状态设置为True，调度器不会再调度任何Pod到该节点上

- nodefs.available
- nodefs.inodesFree
- imagefs.available
- imagefs.inodesFree

驱逐行为

- 有容器运行时分区
    - nodefs达到驱逐阈值，那么kubelet删除已经退出的容器
    - Imagefs达到驱逐阈值，那么kubelet删除所有未使用的镜像。
- 无容器运行时分区
    - kubelet同时删除未运行的容器和未使用的镜像。

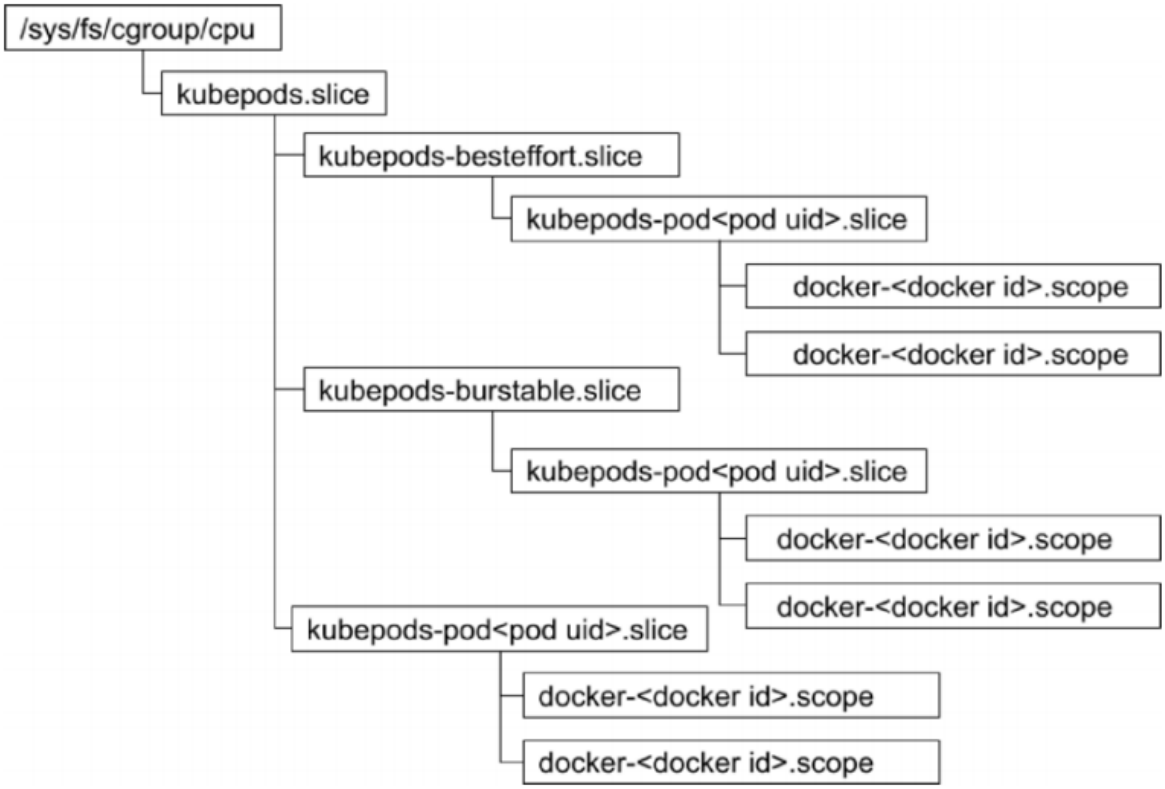
2. 如果释放空间后，仍然满足驱逐条件

   回收已经退出的容器和未使用的镜像后，如果节点依然满足驱逐条件，kubelet就会开始驱逐正在运行的Pod，进一步释放磁盘空间。

   - 判断Pod的磁盘使用量是否超过请求的大小，超出请求资源的Pod会成为备选目标。
   - 查询Pod的调度优先级，低优先级的Pod优先驱逐。
   - 根据磁盘使用超过请求的数量进行排序，差值越小，越不容易被驱逐。

容器和资源配置

CPU

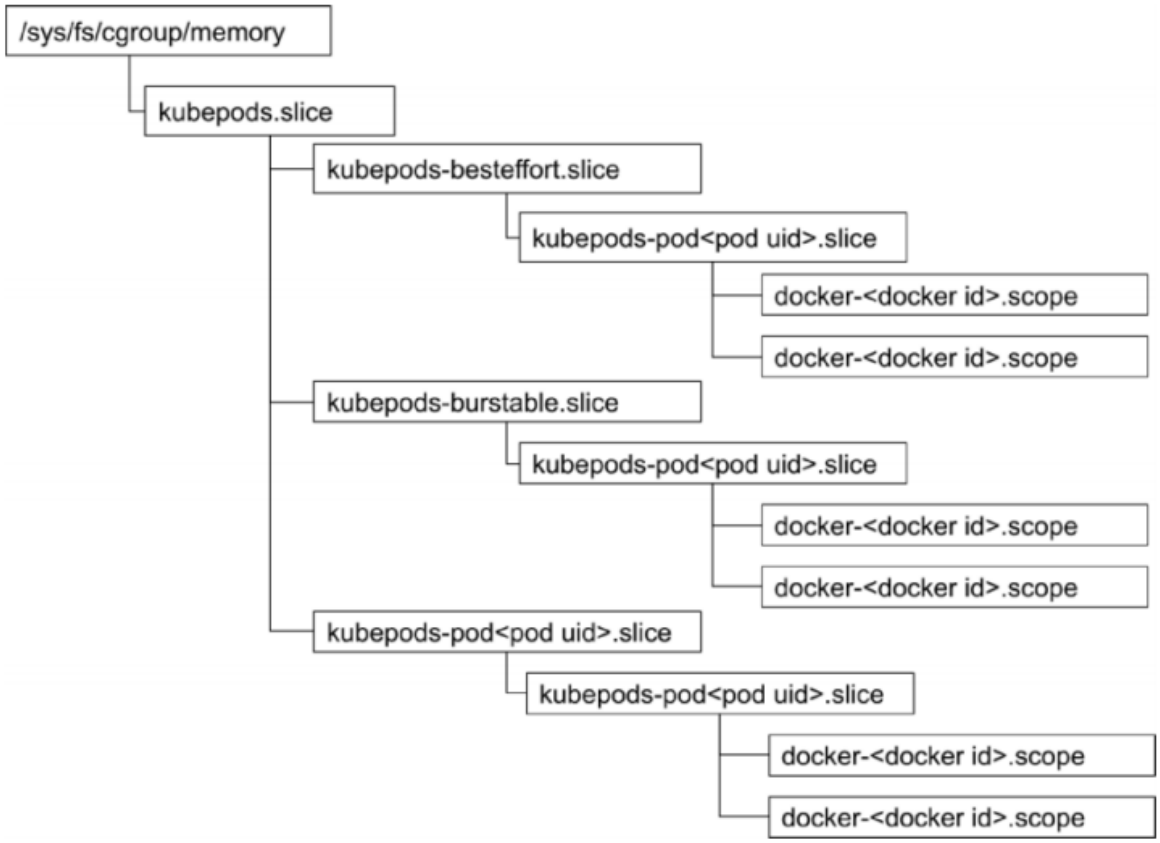


可以看到，对于BestEffort和burstable类型都有一个父目录，而garunteed类型的pod是每个pod自己一个配置。因为前两者每个Pod的设置之间有联动关系（具体没有细讲）。

CGroup类型	参数	QoS类型	值
容器的CGroup	cpu.shares	BestEffort	2
		Burstable	requests.cpu×1024
		Guaranteed	requests.cpu×1024
	cpu.cfs_quota_us	BestEffort	-1
		Burstable	limits.cpu×100
		Guaranteed	limits.cpu×100
Pod的CGroup	cpu.shares	BestEffort	2
		Burstable	Pod所有容器（requests.cpu×1024）之和
		Guaranteed	Pod所有容器（requests.cpu×1024）之和
	cpu.cfs_quota_us	BestEffort	-1
		Burstable	Pod所有容器（limits.cpu×100）之和
		Guaranteed	Pod所有容器（limits.cpu×100）之和

内存

> 类似于CPU，对于BestEffort和burstable类型都有一个父目录，而garunteed类型的pod是每个pod自己一个配置。因为前两者每个Pod的设置之间有联动关系（具体没有细讲）。



CGroup类型	参数	QoS类型	值
容器的CGroup	memory.limit_in_bytes	BestEffort	9223372036854771712
		Burstable	limits.memory
		Guaranteed	limits.memory
Pod的CGroup	memory.limit_in_bytes	BestEffort	9223372036854771712
		Burstable	所有Pod容器（limits.memory）之和
		Guaranteed	所有Pod容器（limits.memory）之和

日志管理

节点上需要通过运行logrotate的定时任务对系统服务日志进行rotate清理，以防止系统服务日志占用大量的磁盘空间。

- logrotate的执行周期不能过长，以防日志短时间内大量增长。
- 同时配置日志的rotate条件，在日志不占用太多空间的情况下，保证有足够的日志可供查看。
- Docker
    - 除了基于系统logrotate管理日志，还可以依赖Docker自带的日志管理功能来设置容器日志的数量和每个日志文件的大小。
    - Docker写入数据之前会对日志大小进行检查和rotate操作，确保日志文件不会超过配置的数量和大小。
- Containerd
    - 日志的管理是通过kubelet定期（默认为10s）执行du命令，来检查容器日志的数量和文件的大小的。
    - 每个容器日志的大小和可以保留的文件个数，可以通过kubelet的配置参数container-log-max-size 和container-log-max-files进行调整。

> 把重要的文字写出来，以便搜索。

- 可以通过kubelet的配置参数container-log-max-size和container-log-max-files调整每个容器日志的大小和可以保留的文件个数。
- docker的日志驱动会把标准输出重定向到文件，但容器内的程序打日志过快导致驱动无法及时将日志放入文件时，就会导致容器进程卡住

Docker卷管理

- K8S尚未将Dockerfile中通过VOLUME指令生命的存储卷纳入管控范围，不建议使用；
- 容器进程在可写层或emptyDir卷进行大量读写，由于底层都在一块盘上，就会到会磁盘IO过高，从而影响其它容器进程甚至系统进程。
- Docker和Containerd运行时都基于CGroup1，只支持对Direct I/O限速，而对Buffered I/O还不具备有效的支持，对于有特殊I/O需求的容器，建议使用独立的磁盘空间。

网络资源限制

因为所有容器共享同一块物理网卡或者虚拟机网卡，所以可以利用CNI插件进行限制（底层都是利用Linux Traffic Control）

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/ingress-bandwidth: 10MB
    kubernetes.io/egress-bandwidth: 10MB
...
```

进程数

kubelet默认不限制Pod可以创建的子进程数量，但可以通过启动参数podPidsLimit开启限制，还可以由reserved参数为系统进程预留进程数。

- kubelet通过系统调用周期性地获取当前系统的PID的使用量，并读取/proc/sys/kernel/pid_max，获取系统支持的PID上限。
- 如果当前的可用进程数少于设定阈值，那么kubelet会将节点对象的PIDPressure标记为True
- kube-scheduler在进行调度时，会从备选节点中对处于NodeUnderPIDPressure状态的节点进行过滤。

节点异常检测

当基础架构守护程序、硬件、内核、容器运行时出问题，k8s服务组件并不会感知以上问题，就会导致pod仍然调度到该节点上。

node-problem-detector

社区引入守护进程node-problem-detector解决上述问题。

download helm chart and unzip

```
helm pull deliveryhero/node-problem-detector
tar -zxvf node-problem-detector-2.0.9.tgz
```

change image

```
vi node-problem-detector/values.yaml
```

```
image:
  repository: cncamp/node-problem-detector
  tag: v0.8.10
  pullPolicy: IfNotPresent
```

helm chart install

```
helm install npd ./node-problem-detector
```

```
sudo sh -c "echo 'kernel: BUG: unable to handle kernel NULL pointer dereference
at TESTING' >> /dev/kmsg"
```

故障分类

Problem Daemon Types	NodeCondition	Description	Configs
系统日志监控	KernelDeadlock ReadonlyFilesystem FrequentKubeletRestart FrequentDockerRestart FrequentContainerdRestart	通过监控系统日志来汇报问题并输出系统指标数据	filelog, kmsg, kernel abrt systemd
CustomPluginMonitor	按需定义	自定义插件监控允许用户自定义监控脚本，并运行这些脚本来进行监控	比如ntp服务监控
HealthChecker	KubeletUnhealthy ContainerRuntimeUnhealthy	针对kubelet和运行时的健康检查	kubelet docker

问题汇报手段

- 永久性故障：通过设置NodeCondition来改变节点的状态
- 临时故障：通过Event来提醒相关对象，比如通知当前节点运行的所有pod

然后自定义一个控制器，监听NPD汇报的condition，taint node，阻止pod调度到故障节点。问题修复后，重启NPD pod来清理错误事件。

使用插件pod启用NPD

如果你使用的是自定义集群引导解决方案，不需要覆盖默认配置， 可以利用插件 Pod 进一步自动化部署。

创建 node-strick-detector.yaml，并在控制平面节点上保存配置到插件 Pod 的目录 /etc/kubernetes/addons/node-problem-detector。

> 这里的重点是插件POD的目录`/etc/kubernetes/addons`.

常用节点问题排查手段

查看日志

针对使用systemd拉起的服务

```
journalctl -afu kubelet -S "2019-08-26 15:00:00"
-u unit，对应的systemd拉起的组件，如kubelet
-f follow，跟踪最新日志
-a show all，现实所有日之列
-S since，从某一时间开始 -S "2019-08-26 15:00:00"
```

对于标准的容器日志

```
kubectl logs -f -c <containername> <podname>
kubectl logs -f --all-containers <podname>
kubectl logs -f -c <podname> --previous  #在pod重启后，可以看到重启之前的日志。
```

如果容器日志被shell转储到文件，则需通过exec

```
kubectl exec -it xxx -- tail -f /path/to/log
```

基于extended resource 扩展节点资源

原理就是利用节点的status/capacity字段用自定义的名字填上资源的数量。

- 集群操作员可以向 API 服务器提交 PATCH HTTP 请求， 以在集群中节点的 status.capacity 中为其配置可用数量。
- 完成此操作后，节点的 status.capacity 字段中将包含新资源。
- kubelet 会异步地对 status.allocatable 字段执行自动更新操作，使之包含新资源。
- 调度器在评估 Pod 是否适合在某节点上执行时会使用节点的 status.allocatable 值， 在更新节点容量使之包含新资源之后和请求该资源的第一个 Pod 被调度到该节点之间， 可能会有短暂的延迟。

命令如下：

```
curl --key admin.key --cert admin.crt --header "Content-Type: application/json-patch+json" \
--request PATCH -k \
--data '[{"op": "add", "path": "/status/capacity/cncamp.com~1reclaimed-cpu", "value": "2"}]' \
https://192.168.34.2:6443/api/v1/nodes/cadmin/status
```

上面命令里的admin.crt和admin.key是从`${HOME}/.kube/config`中拿出来的，执行完后，节点上就会有类型为cncamp.com~1reclaimed-cpu的资源，然后下面的pod就可以申请使用这种资源了

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
```

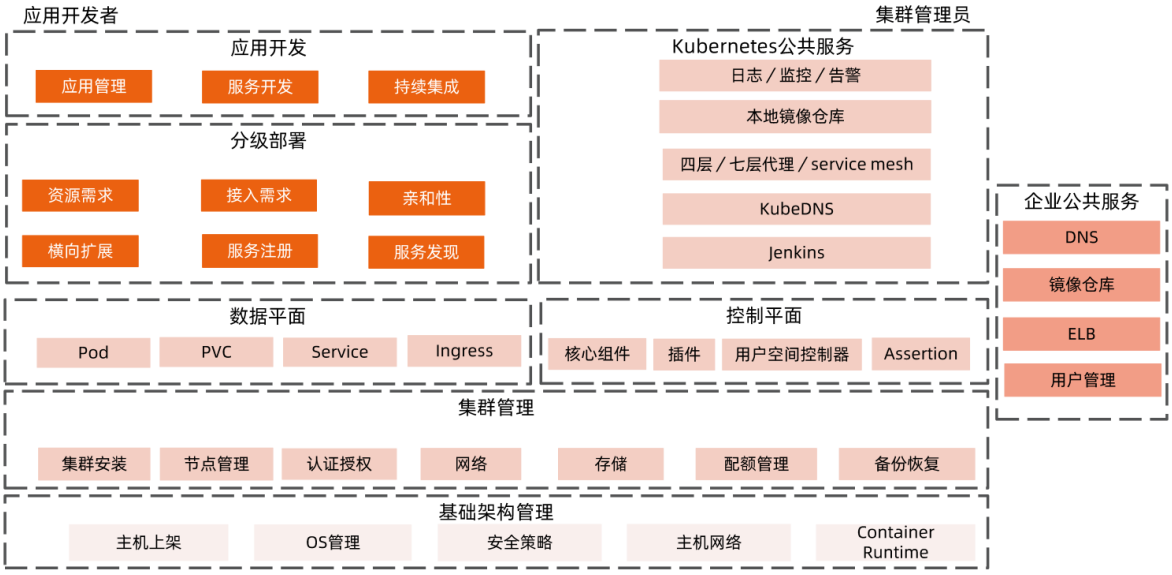
```
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            limits:
              cncamp.com/reclaimed-cpu: 3
            requests:
              cncamp.com/reclaimed-cpu: 3
```

> 扩展资源无法超兽也无法驱逐。

构建和管理高可用集群

k8s融合了基础设施管理和应用管理，磨平了IAAS\PAAS\SAAS之间的区别。

Kubernetes高可用层级



生产化集群管理

- 如何设定单个集群规模
    - 社区声明单一集群可支持5000节点，在如此规模的集群中，大规模部署应用是有诸多挑战的。应该更多还是更少？如何权衡？
- 如何根据地域划分集群
    - 不同地域的计算节点划分到同一集群
    - 将同一地域的节点划分到同一集群
- 如何规划集群的网络
    - 企业办公环境、测试环境、预生产环境和生产环境应如何进行网络分离
    - 不同租户之间应如何进行网络隔离
- 如何自动化搭建集群
    - 如何自动化搭建和升级集群，包括自动化部署控制平面和数据平面的核心组件
    - 如何与企业的公共服务集成

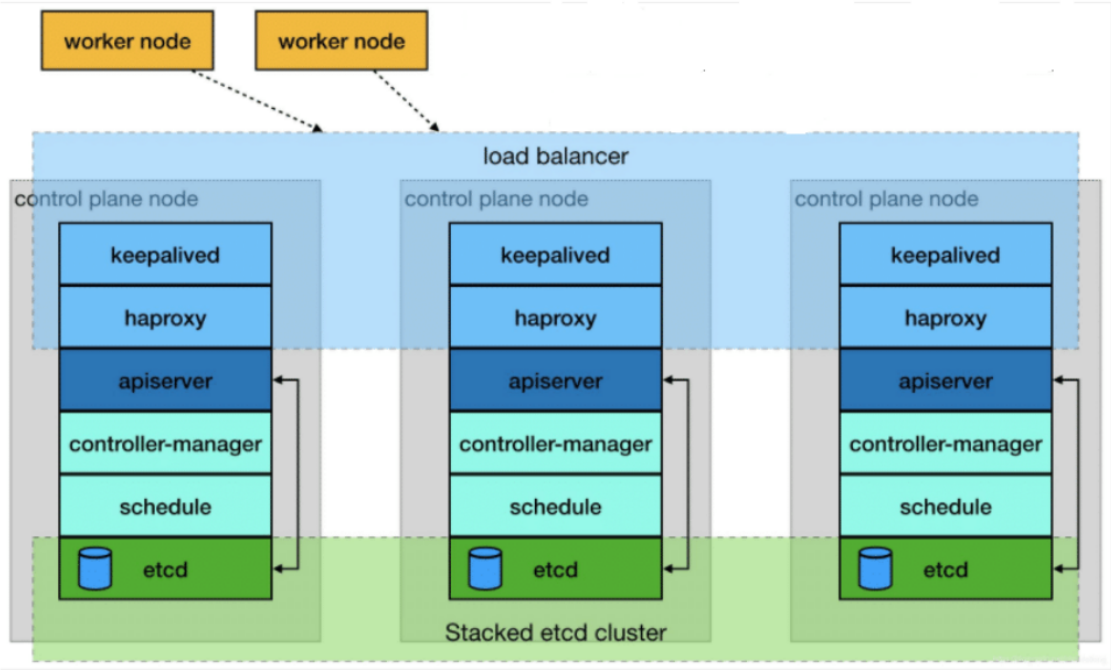
其中，根据地域划分集群（本质上是基于K8S的分区架构）：

- 不同地域的计算节点划分到同一集群，通过给POD或节点打标签，使得POD分布在不同的数据中心里。这样会：
    - 增加复杂度，深圳的请求应该优先发到深圳服务节点
    - 这样不同地域的节点在一个集群中，但是对网络延迟较大的两个数据中心来说，ETCD的心跳和选主就慢了，给控制面带来压力，使得集群规模下降。

控制平面高可用

- 针对大规模的集群，应该为控制平面组件划分单独节点，减少业务容器对控制平面容器或守护进程的干扰和资源抢占
- 控制平面所在的节点，应确保在不同机架上，以防止因为某些机架的交换机或电源出问题，造成所有的控制面节点都无法工作
- 保证控制平面的每个组件有足够的CPU、内存和磁盘资源，过于严苛的资源限制会导致系统效率低下，降低集群可用性
- 应尽可能地减少或消除外部依赖。在Kubneretes初期版本中存在较多Cloud Provider API的调用，导致在运营过程中，当Cloud Provider API出现故障时，会使得Kubernetes集群也无法正常工作。
- 应尽可能地将控制平面和数据平面解耦，确保控制平面组件出现故障时，将业务影响降到最低。
- Kubernetes还有一些核心插件，是以普通的Pod形式加载运行的，可能会被调度到任意工作节点，与普通应用竞争资源。这些插件是否正常运行也决定了集群的可用性。

高可用集群架构



高可用集群安装

各种方法比较

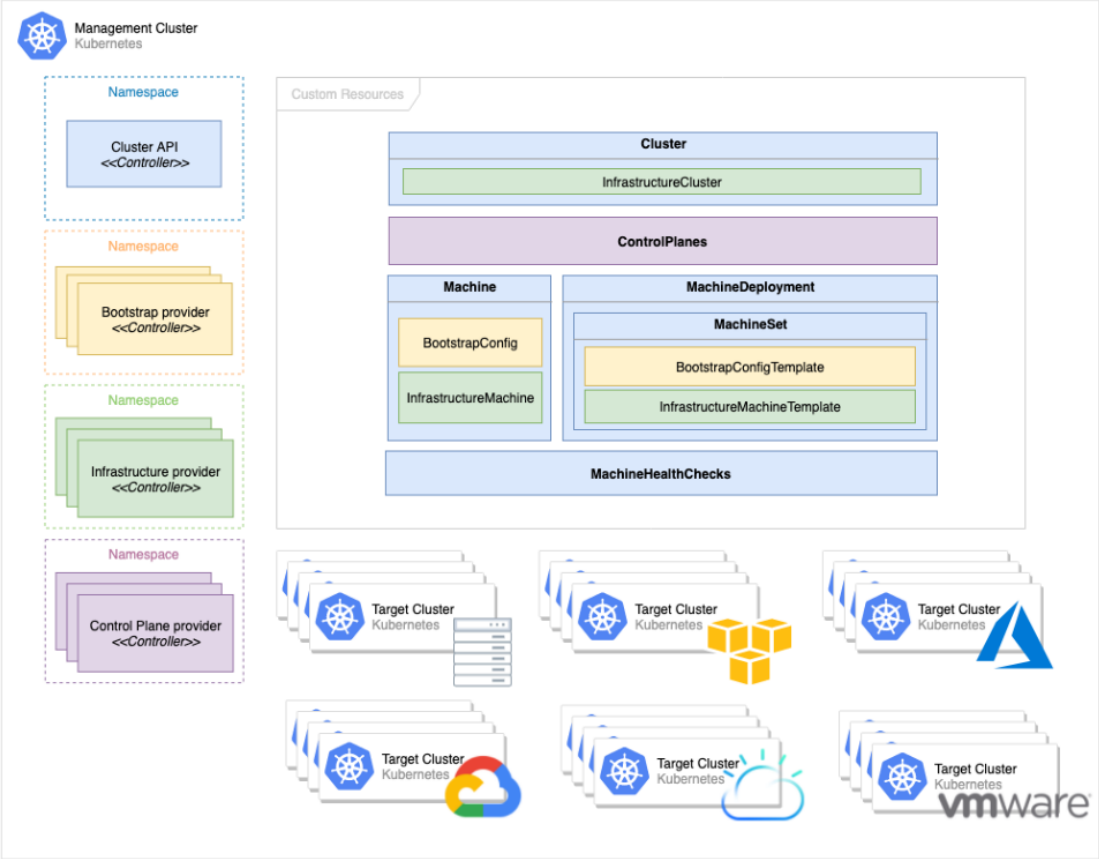
安装工具	方法	优势	缺点
二进制	下载二进制文件，并通过设置systemd来管理	灵活度强	复杂，需要关心每一个组件的配置 对系统服务的依赖性过多
Kubeadm	Kubeadm是一个搭建集群的命令行工具 管理节点通过kubeadm init初始化 计算节点通过kubeadm join加入	相比二进制，控制面板组件的安装和配置被封装起来了 管理集群的生命周期，比如升级，证书管理等	操作系统层面的配置无自动化 运行时安装配置等复杂步骤依然是必须的 CNI插件等需要手工安装
Kubespray	通过Ansible-playbook完成集群搭建	自动完成操作系统层面的配置 利用了kubeadm作为集群管理工具	缺少基于声明式API的支持
KOPS	基于声明式API的集群管理工具	基于社区标准的Cluster API进行集群管理 节点的操作系统安装等全自动化	与云环境深度集成 灵活性差

重点是KOPS，其它的方法就不记录了。

基于声明式API管理集群

管理不仅仅是搭建，更主要的是管理，比如集群扩缩容、节点健康检查和自动修复、kubernetes升级、操作系统升级等。

- Kubernetes Cluster API结构



- 参与角色
  - 管理集群
    - 管理workload 集群的集群，用来存放Cluster API对象的地方
  - Workload集群
    - 真正开放给用户用来运行应用的集群，由管理集群管理
  - Infrastructure provider
    - 提供不同云的基础架构管理，包括计算节点，网络等。目前流行的公有云多与Cluster API集成了。
  - Bootstrap provider
    - 证书生成
    - 控制面组件安装和初始化，监控节点的创建
    - 将主节点和计算节点加入集群
  - Control plane
    - Kubernetes控制平面组件
- 涉及模型
  - Machine
    - 计算节点，用来描述可以运行Kubernetes组件的机器对象（注意与Kubernetes Node）的差异
    - 一个新Machine被创建以后，对应的控制器会创建一个计算节点，安装好操作系统并更新Machine的状态
    - 当一个Machine被删除后，对应的控制器会删除掉该节点并回收计算资源。
    - 当Machine属性被更新以后（比如Kubernetes版本更新），对应的控制器会删除旧节点并创建新节点
  - Machine Immutability (In-place Upgrade vs. Replace)
    - 不可变基础架构
  - MachineDeployment
    - 提供针对Machine和MachineSet的声明式更新，类似于Kubernetes Deployment
  - MachineSet
    - 维护一个稳定的机器集合，类似Kubernetes ReplicaSet
  - MachineHealthCheck
    - 定义节点应该被标记为不可用的条件

其中MachineDepolyment、MachineSet、Machine相当于Deployment、ReplicaSet、Pod之间的关系。

实验

如果需要运行老师给的例子，需要安装cluster api：

- 先安装kind

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.11.1/kind-linux-amd64
chmod +x ./kind
mv kind /usr/bin/
```

- 安装Cluster API

> 参考《The Cluster API Book》

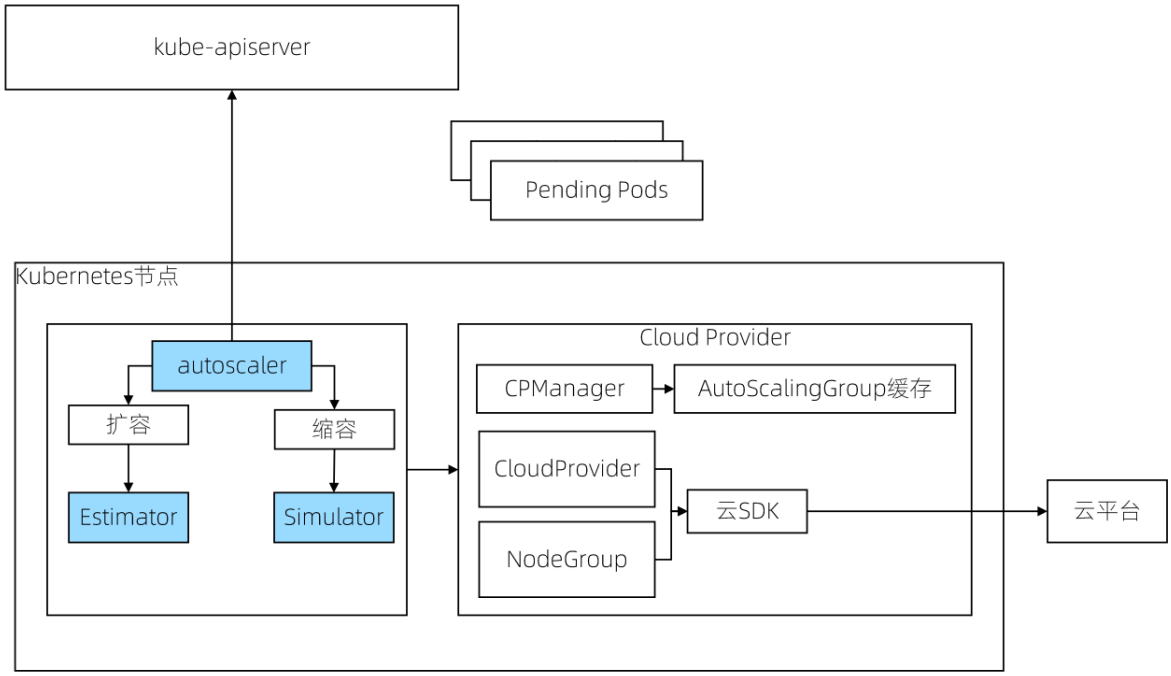
```
curl -L https://github.com/kubernetes-sigs/cluster-
api/releases/download/v1.0.2/clusterctl-linux-amd64 -o clusterctl

chmod +x ./clusterctl

sudo mv ./clusterctl /usr/local/bin/clusterctl
clusterctl version
```

Cluster Autoscaler

Cluster Autoscaler架构



架构中的角色有：

Autoscaler：核心模块，负责整体扩缩容功能

Estimator：负责评估计算扩容节点

Simulator：负责模拟调度，计算缩容节点

Cloud-Provider：与云交互进行节点的增删操作，每个支持CA的主流厂商都实现自己的plugin实现动态缩放

扩展机制

为了自动创建和初始化 Node，Cluster Autoscaler 要求 Node 必须属于某个 Node Group，比如

- GCE/GKE 中的 Managed instance groups（MIG）
- AWS 中的 Autoscaling Groups
- Cluster API Node

当集群中有多个 Node Group 时，可以通过 --expander=<option> 选项配置选择 Node Group 的策略，支持如下四种方式

- random：随机选择
- most-pods：选择容量最大（可以创建最多 Pod）的 Node Group
- least-waste：以最小浪费原则选择，即选择有最少可用资源的 Node Group
- price：选择最便宜的 Node Group

多租户集群管理

- 认证，实现多租户的基础

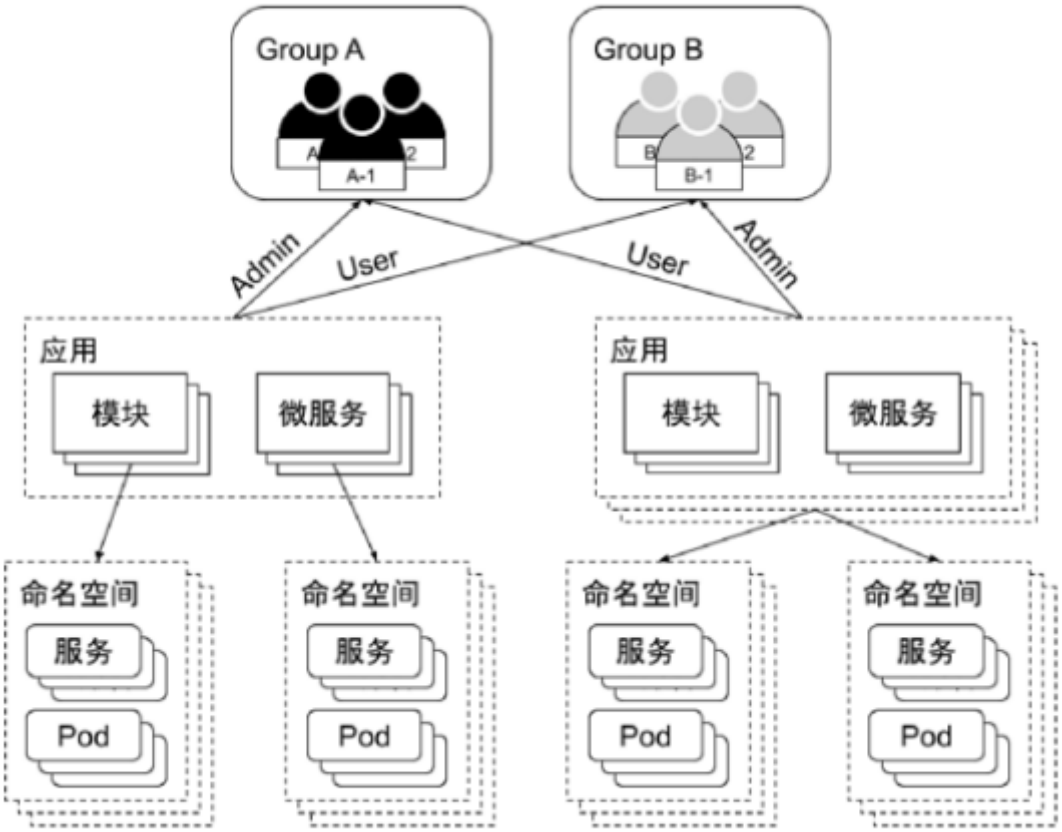
  可以管理两类用户

  - 用来标识和管理系统组件的ServiceAccount
  - 外部用户的认证，需要通过K8S的认证扩展来对接企业、供应商的认证服务
- （**资源**）隔离（自己将下面的三种隔离分为"资源"隔离）

  这里**隔离**指保证用户的工作负载彼此之间有尽可能安全的隔离，减少用户工作负载之间的影响。

  - 权限隔离。**目的**是阻止对宿主机及其它用户的容器进行读取、写入等操作，**手段**是不给普通用户容器priviledged、sys_admin、net_admin等高级管理权限。
  - 网络隔离。**目的**是让容器之间只能通过容器开放的端口进行通信，而不能通过其它方式进行访问；**手段**是让不通过的POD运行在不同的Network Namespace中，拥有独立的网络协议栈。
  - 数据隔离。**目的**是容器相互之间无法访问对方的文件系统、进程、IPC等信息，**手段**有：
    - 对于不同pod中的容器，运行在不同的MNT、UTS、PID、IPC Namespace上；
    - 同一个Pod的容器，其MNT、PID Namespace也不共享。
- 租户隔离

  租户隔离是基于Namespace的，Namespace 属于且仅属于一个租户。这里命名空间和租户之间是N:1的关系，即一个租户可以拥有多个命名空间



> 注意：**一个租户可以有多个成员**！！

  - 权限：即命名空间中的Role与RoleBinding，表示租户在当前命名空间中定义了哪些权限、权限授权给了哪些用户。
  - POD安全策略：特殊权限指集群级的资源定义——PodSecurityPolicy，描述了工作负载和基础设施之间、工作负载和工作负载之间的关联关系，并通过RoleBinding完成授权。
  - 网络策略。
  - 网络空间中权限的客体指Pod、Service、PersistentVolumeClaim，是租户应用落地到K8S中的实体。
- 权限隔离

- 基于Namespace的权限隔离
    - 创建一个namespace-admin ClusterRole，拥有所有对象的所有权限
    - 为用户开辟新namespace，并在该namespace创建rolebinding绑定namespace-admin ClusterRole，用户即可拥有当前namespace所有对象操作权限
- 自动化解决方案
    - 当Namespace创建时，通过mutatingwebhook将namespace变形，将用户信息记录至namespace annotation
    - 创建一个控制器，监控namespace，创建rolebinding为该用户绑定namespace-admin的权限

- Quota管理
    1. 开启ResourceQuota准入插件；
    2. 在用户Namespace创建ResourceQuota对象进行配额限制

```
apiVersion: v1
kind: ResourceQuota
metadata:
 name: high-qos-limit-requests
spec:
 hard:
  limits.cpu: 8
  limits.memory: 24Gi
  pods: 10
  requests.cpu: 4
  requests.memory: 12Gi
 scopes:
 - NotBestEffort
```

- 节点资源管理
  - 通过为不同节点设置不同taint来识别不同租户的计算资源。
  - 不同租户在创建Pod时，增加Toleration关键字，确保其能调度至某个被Taint的节点。