KUBERNETES1.4 版本新功能介绍

增强包括调度在内的 Kubernetes 基础架构….

摘要

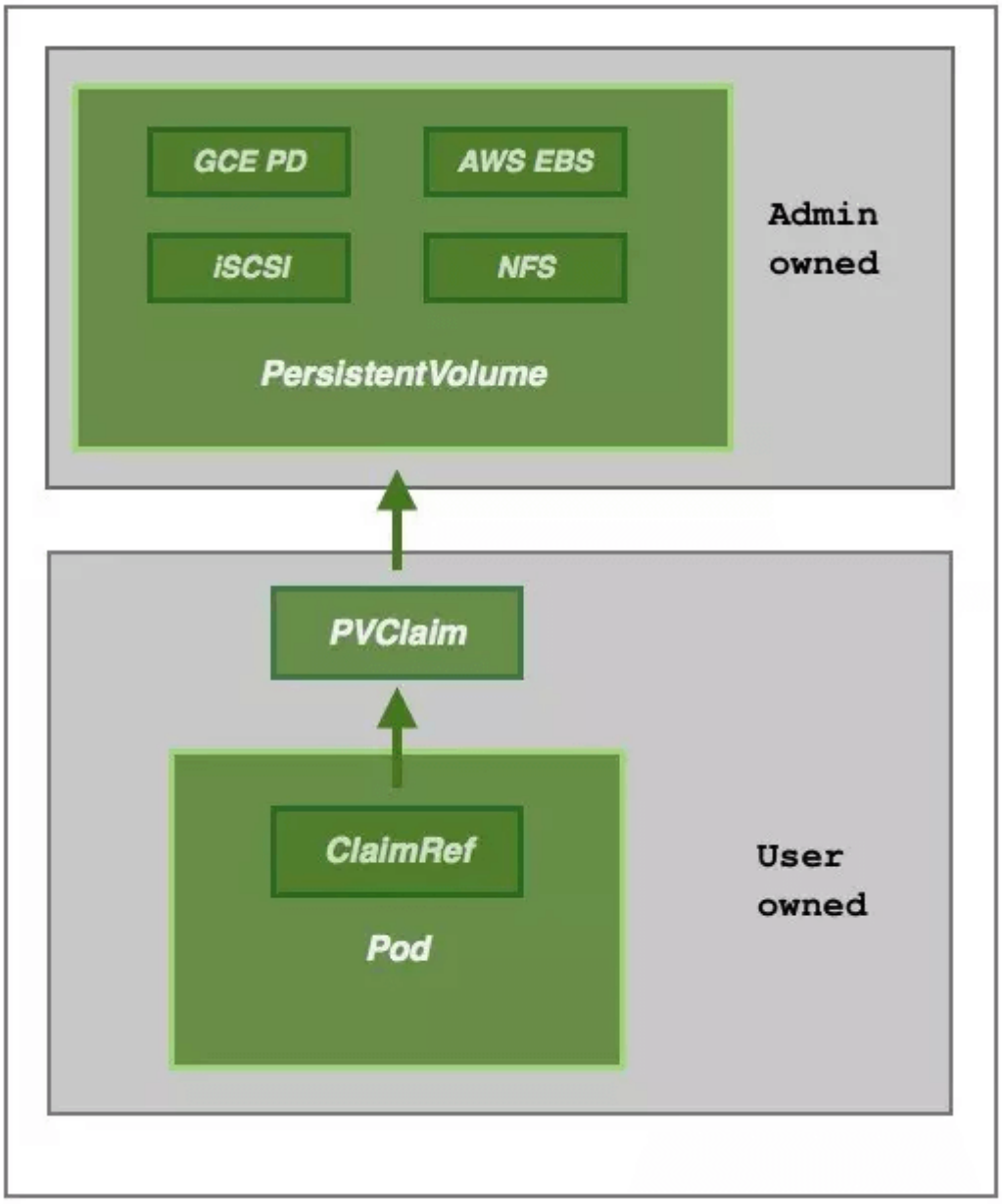
创建 kubernetes 集群只需要两条命令
增强了对有状态应用的支持
增加了集群联盟 API
支持容器安全控制
增强包括调度在内的 Kubernetes 基础架构
通过 Kubernetes DashBoard UI 已经可以实现 90%的命令行操作
……

Kubernetes 中文社区

www.kubernetes.org.cn

目录

Kubernetes 1.4 持久卷



本文描述了目前 kubernetes 中持久卷（PersistentVolume）的情况，建议先熟悉卷（Volume）的概念。 在云环境下，存储资源管理和计算资源管理有区别。持久卷（PersistentVolume）子系统提供了 API，屏蔽了底层的细节，用户和管理员不用关心存储如何提供，如何消耗等。为此，我们在 kubernetes 1.4 中引入了两个新的 API 资源（kind）：PersistentVolume 和 PersistentVolumeClaim。

持久卷(PV) 通常我们把存储做成集群，对外表现为一个网络存储，持久卷(PV)是这个资源中的一个片段，就像 node 和 cluster 的关系。PV 是跟 Volume 一样是卷插件【详见 kubernetes **Volume** 概念】，但其生命周期不依赖任何单个的 pod，底层存储实现可以是 NFS，iSCSI，云存储等，都通过 API 对象对外暴露，对用户透明。

持久卷请求(PVC) 用户通过持久卷请求（Persistent Volume Claim – PVC) 申请存储资源，PVC vs PV 可以类比 pod 和 node 的关系，pod 会消耗 node 的资源，PVC 消耗 PV 的资源。pod 可以申请计算资源的粒度（CPU、内存),PVC 可以申请大小、访问方式（例如 mount rw 一次或 mount ro 多次等多种方式）。

存储级别 经过 API 抽象，用户可以通过 PVC 使用存储资源，通常用户还会关心 PV 的很多属性，例如对不同的应用场景需要不同的性能，仅仅提供存储大小和访问模式不能满足要求。集群管理员一方面要提供不同 PV 的多种属性，一方面要隐藏底层的细节，就引入了 StorageClass 资源。管理员用存储级别 StorageClass 描述存储的分类，不同的分类可以对应不同的质量服务 Qos 等级、备份策略和其他自定义的策略。kubernetes 本身不参与存储级别的划分，StorageClass 概念在有的存储系统里被称为”profiles”。

如果想尝试 PersistentVolumeClaim，请点击详细案例。

生命周期

PV 是集群的资源，PVC 请求资源并检查资源是否可用，PV 和 PVC 生命周期如下：

资源提供

包含静态和动态两种提供方式。

静态

存储集群先创建一定数量的 PV，PV 包含了真实存储的底层细节，用户通过 kubernetes API 使用存储。

动态

在现有 PV 不满足 PVC 的请求时，管理员可以使用依据存储分类 (StorageClass)，描述具体过程为：PV 先创建分类，PVC 请求已创建的某个类的资源，这样就达到动态配置的效果。请求类有效的禁止了动态配置自己。

绑定

用户根据所需存储空间大小和访问模式创建（或在动态部署中已创建）一个 PersistentVolumeClaim。Kubernetes 的 Master 节点循环监控新产生的 PVC，找到与之匹配的 PV（如果有的话），并把他们绑定在一起。动态配置时，循环会一直将 PV 与这个 PVC 绑定，直到 PV 完全匹配 PVC。避免 PVC 请求和得到的 PV 不一

致。绑定一旦形成，PersistentVolumeClaim 绑定就是独有的，不管是使用何种模式绑定的。

如果找不到匹配的 volume，用户请求会一直保持未绑定状态。在匹配的 volume 可用之后，用户请求将会被绑定。比如，一个配置了许多 50Gi PV 的集群不会匹配到一个要求 100Gi 的 PVC。 只有在 100Gi PV 被加到集群之后，这个 PVC 才可以被绑定。

使用

Pods 使用 Claims 做为 volumes。集群检查 Claim，查找绑定的 volume，并为 Pod 挂载这个 volume。对于支持多访问模式的 volume，用户在 Pod 中使用 Claim 时可指定自己想要的访问模式。

用户一旦有了 Claim，并且被绑定，那么被绑定的 PV 将一直属于该用户。用户可通过在 Pod volume 块中包涵 PersistentVolumeClaim 的方式来调度 Pods，并访问他们所要求的 PV。

释放

当用户使用完 volume 后，可以从 API 中删除 PVC 对象，这样资源就可以被回收了。Claim 被删除之后，volume 被认为是‘释放’状态，但它还不能被其它 claim 使用。之前使用者的数据还保留在 volume 上，这个必须按一定策略进行处理。

回收利用

PersistentVolume 的重用策略将告诉集群在 volume 被释放之后该如何进行处理。volume 被释放后可用保留，回收或删除。保留支持手工重分配资源。

删除（需要 volume 插件支持），支持从 Kubernetes 中删除 PersistentVolume 对象，并支持从外部设施中（比如 AWS EBS, GCE PD or Cinder volume）删除对应的存储资源。动态部署的 volumes 会被直接删除。

回收（需要 volume 插件支持），可对 volume 执行基本的删除操作(rm -rf /thevolume/*) ，并使它对新的 claim 可用。

持久存储的种类

Kubernetes 通过插件支持下列存储：

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- FC (Fibre Channel)
- NFS
- iSCSI
- RBD (Ceph 块设备)
- CephFS
- Cinder (openstack 的快存储)
- Glusterfs
- VsphereVolume
- HostPath (只能用于单节点)

持久卷（PV）

每个 PV 定义中包含 spec 和 status，下面是一个简单的例子：

```
apiVersion: v1

kind: PersistentVolume

metadata:

    name: pv0003

    annotations:

        volume.beta.kubernetes.io/storage-class: "slow"

spec:

    capacity:
```

```
        storage: 5Gi
    accessModes:
        - ReadWriteOnce
    persistentVolumeReclaimPolicy: Recycle
    nfs:
        path: /tmp
        server: 172.17.0.2
```

容量

一般来说，PV 通过 capacity 属性定义容量大小。可以通过查看 Kubernetes 资源模型理解容量需求。

目前仅支持容量大小，将来会包括 iops，吞吐量等属性。

访问方式

存储有很多种，每种存储又有自己的特性，PV 访问不同的存储会有不同的访问方式，例如 NFS 支持多个 rw 的客户端，但具体到 NFS PV 可能会用只读方式 export。对不同 PV 的特性，每个 PV 有自己的访问方式。

访问方式包括：

ReadWriteOnce – 被单个节点 mount 为读写 rw 模式

ReadOnlyMany – 被多个节点 mount 为只读 ro 模式

ReadWriteMany – 被多个节点 mount 为读写 rw 模式

在 CLI 下，访问方式被简写为：

RWO – ReadWriteOnce

ROX – ReadOnlyMany

RWX – ReadWriteMany

重要的是，在同一时刻，一个卷只能使用一种访问方式 mount，例如 GCEPersistentDisk 被可以一个节点 mount 为 ReadWriteOnce 也可以被多个节点 mount 为 ReadOnlyMany，但不能同时做。

下图列举了 Kubernetes 支持的存储插件的访问方式:

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	x	–	–
AzureFile	x	x	x
CephFS	x	x	x
Cinder	x	–	–
FC	x	x	–
FlexVolume	x	x	–
GCEPersistentDisk	x	x	–
Glusterfs	x	x	x
HostPath	x	–	–
iSCSI	x	x	–
NFS	x	x	x
RDB	x	x	–
VsphereVolume	x	–	–

分类

PV 分类通过设置注释 volume.beta.kubernetes.io/storage-class 定义 StorageClass 的名称。一个特定的 PV 分类与 PVC 请求绑定。PV 没有注释或注释为””表示没有分类，只能绑定到通用 PVC。

未来正式版，volume.beta.kubernetes.io/storage-class 注释会成为属性。

回收策略

现有回收策略有：

- Retain – 手动重新使用
- Recycle – 基本的删除操作（“rm -rf /thevolume/*”)
- Delete – 关联的后端存储卷一起删除，后端存储例如 AWS EBS, GCE PD 或 OpenStack Cinder

目前只有 NFS 和 HostPath 支持回收，AWS EBS, GCE PD 和 Cinder volumes 只支持删除。

卷的状态

卷有四种状态，一个卷必属于其中之一:

- Available –闲置状态，没有被绑定到 PVC
- Bound – 绑定到 PVC
- Released – PVC 被删掉，资源没有被在利用
- Failed – 自动回收失败
- CLI 会显示绑定到 PV 的 PVC 名。

持久卷请求（PVC）

PVC 包涵 spec 和 status:

```
kind: PersistentVolumeClaim
```

```yaml
apiVersion: v1

metadata:

    name: myclaim

    annotations:

       volume.beta.kubernetes.io/storage-class: "slow"

spec:

    accessModes:

        - ReadWriteOnce

    resources:

        requests:

            storage: 8Gi

        selector:

            matchLabels:

                release: "stable"

            matchExpressions:

                - {key: environment, operator: In, values: [dev]}
```

访问方式

PVC 使用特定的访问方式式请求存储资源，并统一约定为卷。

资源

PVC 可以像 pod 一样申请一定数量的资源，不过申请的是存储资源，kubernetes 的资源模型同样适用于卷和 PVC。

选择器

PVC 可以指定一个标签选择器来进一步过滤卷集，只有匹配的的卷标签会被绑定到 PVC，Selector 包涵 2 项内容：

- matchLabels – 匹配标签，卷标签必须匹配某个值
- matchExpressions – 匹配表达式，由键值对，操作符构成，操作符包括 In，NotIn，Exists，和 DoesNotExist。

所有请求，必须全部满足 matchLabels 和 matchExpressions 的定义。

分类

PVC 可以用特定的 StorageClass 名称，通过 volume.beta.kubernetes.io/storage-class 注释请求特定的 PV 分类。PV 和 PVC 注释一致时，才能绑定。

PVC 请求可以不带分类。PVC 的注释为空“”表示此请求没有分类要求，所以只能被绑定到没有分类的 PV（没有注释或注释为空“”）。集群有一个 DefaultStorageClass 准入插件的开关决定了无类 PVC 是否相同。

如果准入插件打开，管理员会指定一个缺省的存储类型 StorageClass，所有无类 PVC 会被绑定到缺省 PV， 把 storageclass.beta.kubernetes.io/is-default-class 设置为“true”，就打开了缺省 StorageClass。当没有缺省 PV 或 is-default-class 为 false 或缺省 PV 不唯一，准入插件会禁止创建 PVC。

如果准入插件关闭，就没有缺省 StorageClass 的概念了。无类 PVC 只能被绑定到无类的 PV，这样无类 PV 和 PV 注解为空””是一样的。

当 PVC 指定 Selector 请求 StorageClass，所有请求做与连接：只有 PV 类和标签都符合请求才能绑定到 PVC。需要注意，目前具有非空选择器的 PVC 不能使用动态配置的 PV

未来正式版，volume.beta.kubernetes.io/storage-class 注释会成为属性。

把 PVC 用作卷

pods 通过将 PVC 当做 volume 来访问存储。在 pod 使用 PVC 时，PVC 必须存在于相同的 namespace 下。集群在 pod 的 namespace 中找到 PVC，并用它获取支持 PVC 的 PV。之后,volume 将会被加载至 host 和 pod 中。

```
kind: Pod

apiVersion: v1

metadata:

    name: mypod

spec:

    containers:

        - name: myfrontend



            image: dockerfile/nginx

            volumeMounts:

           - mountPath: "/var/www/html"

               name: mypd

   volumes:

        - name: mypd

            persistentVolumeClaim:

                claimName: myclaim
```

Namespaces 注意事项

PV 的绑定是独占式的，而且因为 PVC 是 namespace 的对象，所以 mount 多种模式（ROX, RWX）的 PVC 只在同一 namespace 中。

存储级别

每个 StorageClasse 都包括 provisioner 和 parameters 字段。当属于 class 的 PV 需要动态提供时，他们会被派上用场。

StorageClass 对象的名字非常重要，决定着用户该如何请求一个特定的 class。在第一次创建 StorageClass 对象时，要设置 class 的名字和其他参数。对象一旦创建将无法更新。

管理员可单独为不需要绑定任何 class 的 PVCs 定义一个默认的 StorageClass。详细请参考 PersistentVolumeClaim 部分。

```
kind: StorageClass

apiVersion: storage.k8s.io/v1beta1

metadata:

    name: standard

provisioner: kubernetes.io/aws-ebs

parameters:

    type: gp2
```

Provisioner

storage classes 的 provisioner 字段决定着哪个 volumn 插件被用来部署 PV。这个字段必须指定。在 beta 版本中，可用的 provisioner 类型有 kubernetes.io/aws-ebs 和 kubernetes.io/gce-pd。

Parameters

Storage classes 的 parameter 字段用来描述属于该 storage class 的 volume。不同的 provisioner 其参数也不同。比如说，io1 对应 type 参数；iopsPerGB 则被指定给 EBS。当参数未被指定时，会用到一些缺省值。

AWS：

```
kind: StorageClass

apiVersion: storage.k8s.io/v1beta1

metadata:

    name: slow

provisioner: kubernetes.io/aws-ebs

parameters:

    type: io1

    zone: us-east-1d

    iopsPerGB: "10"
```

- type：io1, gp2, sc1, st1. 详细请参考 AWS docs for details. 缺省: gp2.
- zone：AWS zone. 如果未指定，将从 Kubernetes cluster 节点所在的 zone 中随机选取一个。
- iopsPerGB：只对 io1 volumes。每秒每 GiB 的 I/O 操作。AWS volume 插件将这个值乘以所请求的 volume 大小来计算该 volume 的 IOPS ，最大值为 20000 IOPS（AWS 所支持的最大值，参考 AWS docs. 该字段为字符格式，即 “10”，非 10） encrypted: 指定 EBS volume 是否应该被加密。合法的值为 “true” or “false”. 该字段为字符格式，即 “true”，非 true.
- kmsKeyId：可选。当 volume 被加密时 Amazon Resource Name 全名会被使用。如未指定，但 encrypted 是 true 时，AWS 会生成一个 key 值。 请参考 AWS docs 有效 ARN 值。

GCE：

```
kind: StorageClass

apiVersion: storage.k8s.io/v1beta1

metadata:
```

```
    name: slow
provisioner: kubernetes.io/gce-pd
parameters:
    type: pd-standard
    zone: us-central1-a
```

- type：pd-standard 或 pd-ssd. 缺省: pd-ssd
- zone：GCE zone。 如未指定，将从控制管理器所在的区域中随机选取一个 zone。

GlusterFS：

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
    name: slow
provisioner: kubernetes.io/glusterfs
parameters:
    endpoint: "glusterfs-cluster"
    resturl: "http://127.0.0.1:8081"
    restauthenabled: "true"
    restuser: "admin"
    restuserkey: "password"
```

- endpoint：glusterfs-cluster 是 endpoint/service 名字，包括 GlusterFS 信任的 IP 地址池。这个参数是必须的。

- resturl：Gluster REST 服务 url，它按需部署 gluster volumes。格式应为 http://IPaddress:Port 。当使用 GlusterFS dynamic provisioner 时，这个参数是必须的。
- restauthenabled : 布尔值，用于指定 REST 服务器上的 Gluster REST 服务认证是否启用。如该值为”true”，你必须为‘restuser’ and ‘restuserkey’ 参数赋值。
- restuser：Gluster REST 服务用户, 用户可以在 Gluster 信任池中创建 volume。
- restuserkey：Gluster REST 服务用户的密码, 用于 REST 服务器认证。

OpenStack Cinder：

```
kind: StorageClass

apiVersion: storage.k8s.io/v1beta1

metadata:

    name: gold

provisioner: kubernetes.io/cinder

parameters:

    type: fast

    availability: nova
```

- type：Cinder 中创建的 VolumeType ，缺省为空。
- availability：可用的 Zone，缺省为空。

可移植配置

如果你在为一个大规模集群编写配置模板或样例，并且需要持久存储，我们建议你使用以下模式：

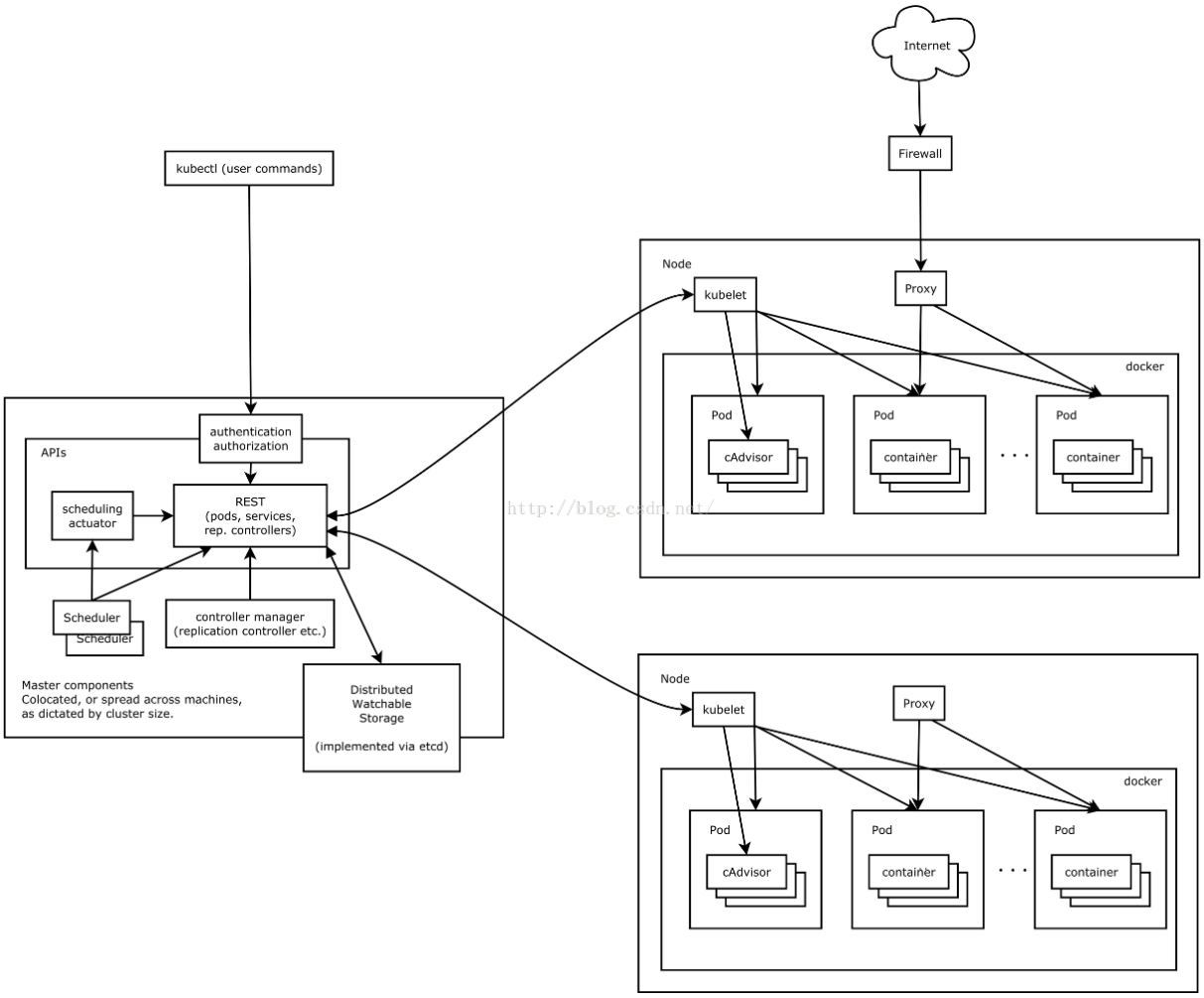
- 在配置组合中一定包涵 PersistentVolumeClaim 对象（以及 Deployments, ConfigMaps 等）

- 一定不要在配置中包涵 PersistentVolume 对象，因为使用配置的用户也许没有权限创建 PersistentVolumes.
- 当创建模板时，让用户可选择是否提供 storage class 名字
- 如果用户提供了 storage class 名字，并且集群版本是 1.4 或以上，把这个值放到 PVC volume.beta.kubernetes.io/storage-class 的注释中. 如果集群的 StorageClasses 已被管理员启用，这会使 PVC 寻找匹配合适的 StorageClass。
- 如果用户未提供 StorageClass 名字，或集群版本是 1.3，那就在 PVC 中加入 volume.alpha.kubernetes.io/storage-class: default 注释。
- 这会使 PV 对一些集群中有默认特性的用户进行自动部署，尽管名字显示 alpha，这个注释的后台是 beta 级别的支持。
- 不要给 volume.beta.kubernetes.io/storage-class: 赋予任何值，包括空值。因为它会阻止 DefaultStorageClass 的准入控制器执行（如果已经启用了的话）。
- 一定要使用监视那些经过一段时间后未被绑定的 PVC，并反馈给用户。因为这或许表明该集群没有动态存储支持（这种情况下用户应该创建一个匹配的 PV），或者该集群没有存储系统 （这种情况下用户不能够部署配置来请求 PVC）。
- 未来我们期待更多的集群启用 DefaultStorageClass，并且有各种存储可用。然而，也许没有什么 storage class 的名字可适用于所有集群，所以不要设置默认名字。到某个阶段后，alpha 注释将失去意义，但 PVC 上未赋值的 storageClass 字段将会达到预期的效果。

kubernetes1.4 增加新节点健康状况类型 DiskPressure

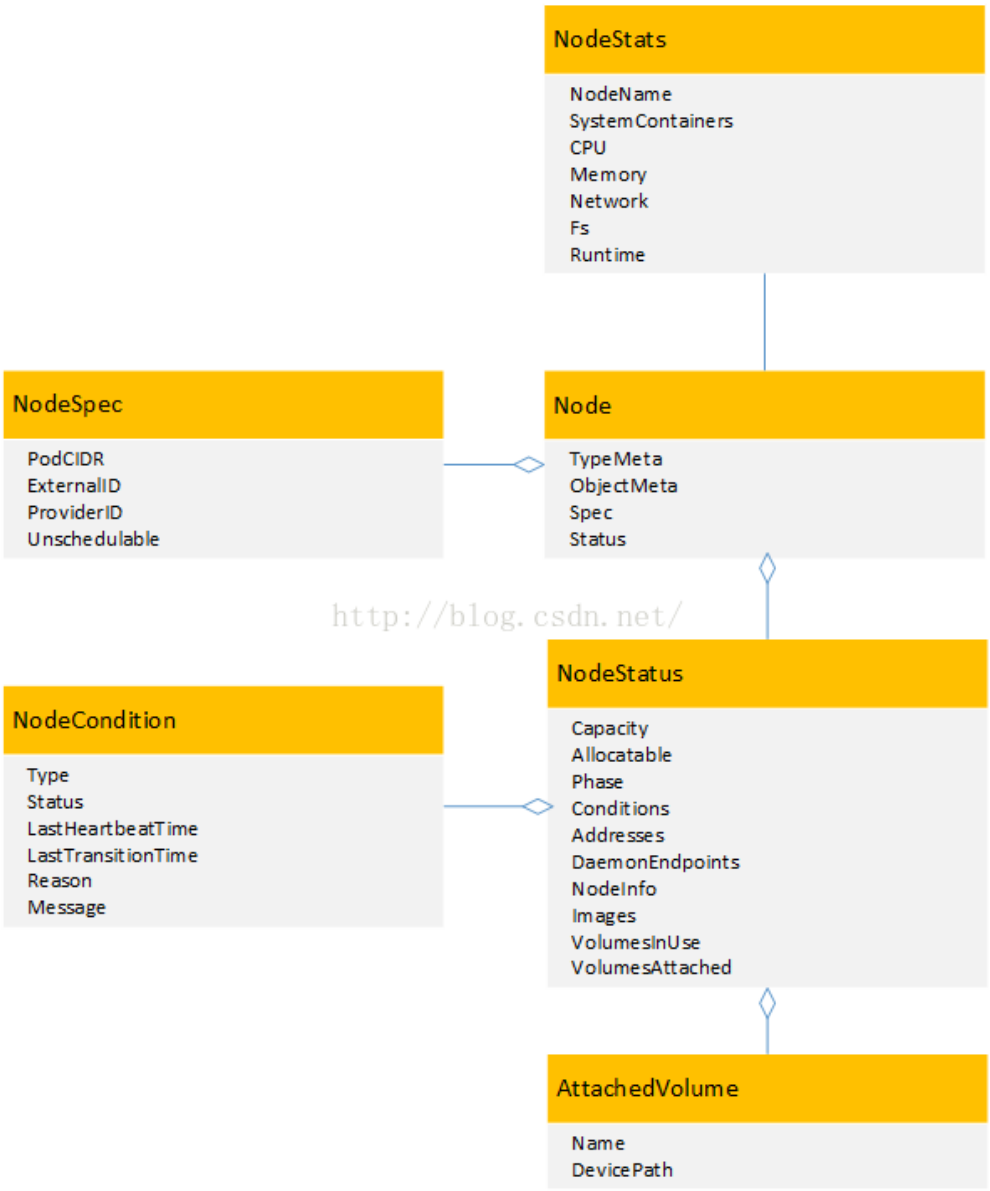
背景资料

在 Kubernetes 架构图中可以看到，节点（Node）是一个由管理节点委托运行任务的 worker。 它能运行一个或多个 Pods，节点（Node）提供了运行容器环境所需要的所有必要条件，在 **Kubernetes** 之前版本中叫做 Minion。



相关结构体

通过下面这张图可以看到在 Kubernetes 中节点（Node）的相关结构体信息：



- 结构体 Node：表示 Kubernetes 中的节点，在节点上面运行 POD。
- 结构体 NodeSpec：存放节点的属性信息。

1. 属性 PodCIDR：表示这个节点上面 POD 可以使用的 IP 范围。
2. 属性 ExternalID：这是一个已经被放弃使用的属性。
3. 属性 ProviderID：当节点是公有云厂商提供的云主机时，这个属性表示公有云系统中对云主机的唯一标识，格式为：
   <ProviderName>://<ProviderSpecificNodeID>
4. 属性 Unschedulable：这是一个布尔型变量，默认是 false。如果为 true 的时候，表示这个节点不能被 Kubernetes 进行调度，也就是 Kubernetes 不能在这个节点上创建新的 POD，但是不会改变这个节点上已经创建的 POD。当

要对节点进行维护，那么就可以将这个节点的 Unschedulable 变量设置成 true，然后对节点进行维护操作。可以通过下面命令来修改这个属性：

```
kubectl patch nodes $NODENAME -p '{"spec": {"unschedulable": true}}'
```


这里面有一个例外，就是如果使用 daemonSet 控制器来创建 POD 的时候，不会关心 Unschedulable 这个属性是否被设置成了 true，这是因为 daemonSet 控制器认为任何节点上的 daemonSet POD 都是必须要创建的，同这个属性无关。

•结构体 NodeStatus：存放节点当前状态信息。

1. 属性 Capacity：存放节点上所有资源总量

2. 属性 Allocatable：存放节点上可以被调度使用的可用资源数量

3. 属性 Phase：存放节点当前所处在什么阶段，一共有三个取值，分别是“Pending”、“Running”和“Terminated”，分别表示如下阶段：

1)Pending：表示节点已经被添加到 Kubernetes 集群中，但是还没有被 Kubernetes 进行配置

2)Running：表示节点已经被 Kubernetes 配置完成，可以由 Kubernetes 进行调度使用

3)Terminated：表示节点已经从 Kubernetes 集群中被删除掉了

•结构体 NodeCondition：存放节点健康状况。

1.属性 Type：节点健康状况类型，包括 Ready、OutOfDisk、MemoryPressure、DiskPressure 和 NetworkUnavailable，分别表示：

1)Ready：表示节点是健康的，可以随时在上面创建 POD

2)OutOfDisk：表示这个节点没有空闲的磁盘空间了，已经不能在上面创建 POD 了

3)MemoryPressure：表示这个节点上可用内存已经很少了

4)DiskPressure：表示这个节点上可用磁盘空间已经很少了

5) NetworkUnavailable：表示这个节点上网络没有被正确配置

2. 属性 Status：表示某种类型健康状况的当前状态，目前只有 True、False 和 Unknown，在 kubernetes 将来版本中还会继续添加新的状态。

1)True：表示当前类型的健康状况确实存在

2)False：表示当前类型的健康状况不存在

3)Unknown：表示 kubernetes 无法确定当前类型的健康状况是否存在

1. 属性 LastHeartbeatTime：表示上一次更新状态的时间
2. 属性 LastTransitionTime：表示上一次状态变化的时间
3. 属性 Reason：表示上一次状态变化的简单原因
4. 属性 Message：表示上一次状态变化的详细原因

- 结构体 AttachedVolume：存放挂载到节点上的数据卷信息。

1. 属性 Name：挂载到节点上数据卷的名称
2. 属性 DevicePath：挂载到节点上数据卷的有效路径

- 结构体 NodeStats：存放节点的统计数据。

1. 属性 NodeName：表示这个节点名称。
2. 属性 SystemContainers：这是一个数组型变量，存放这个节点上所有系统容器的统计信息，这里系统容器指的是以 Daemon 状态运行的 kubelet 和 Docker 容器。
3. 属性 StartTime：表示开始对节点进行统计的时间。
4. 属性 CPU：表示 CPU 相关的统计数据。
5. 属性 Memory：表示内存相关的统计数据
6. 属性 Network：表示网络相关的统计数据。
7. 属性 Fs：表示 Kubernetes 模块使用文件系统相关的统计数据。
8. 属性 Runtime：表示用户容器运行时的统计数据。

新特性

在 Kubernetes1.4 中，结构体 NodeCondition 新增了一个健康状况类型 DiskPressure，用来表示这个节点上可用磁盘空间已经很少了。

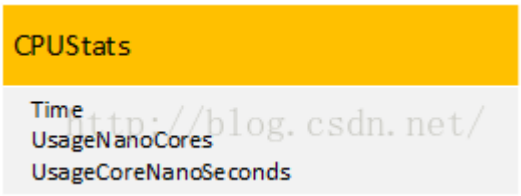
新增了这个健康状况类型 DiskPressure 后，在两个方面会提升 Kubernetes 的使用：

1.可以在调度 POD 的时候进行参考，如果节点上确实发生了 DiskPressure 这件事，那么就会由 scheduler 模块将 POD 调度到其他节点上。

2.可以在控制节点的时候进行参考，如果节点上确实发生了 DiskPressure 这件事，那么就会由 kubelet 模块回收这个节点上的所有 POD，将这些 POD 驱逐到其他节点上。

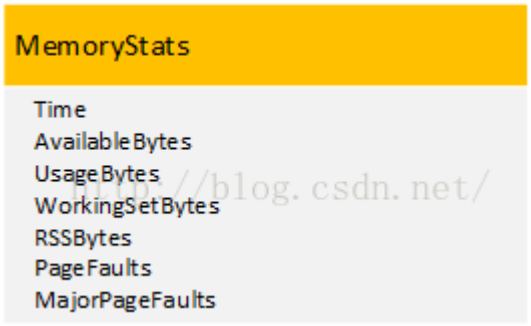
下面介绍 kubelet 模块在控制节点的时候是如何参考使用 DiskPressure 的，这就需要更细致的分析结构体 NodeStats 中的五个属性：

1. 属性 CPU：表示 CPU 相关的统计数据。这个属性对应结构体 CPUStats：



其中属性 Time 表示统计数据更新时间，属性 UsageNanoCores 表示节点上所有 CPU 在采样窗口内的使用量，属性 UsageCoreNanoSeconds 表示节点上所有 CPU 历史使用总量。

2. 属性 Memory：表示内存相关的统计数据。这个属性对应结构体 MemoryStats：



其中属性 Time 表示统计数据更新时间，属性 AvailableBytes 表示可以使用的内存总量，属性 UsageBytes 表示已经被分配的内存大小，属性 WorkingSetBytes 表示已经被使用的内存大小，已经被使用的内存大小<=已经被分配的内存大小，也就是说属性 WorkingSetBytes<=UsageBytes，属性 RSSBytes 表示匿名缓存和 swap

缓存（包括 Linuxtransparent huge pages）大小，属性 PageFaults 表示次要内存缺页错误数，属性 MajorPageFaults 表示主要内存缺页错误数。

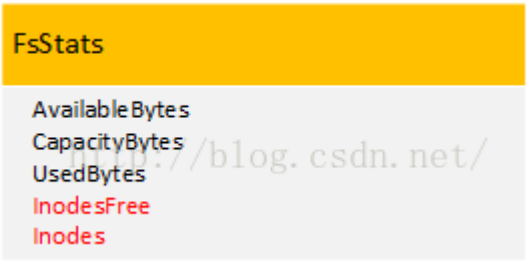
什么叫错缺页错误：

节点上物理内存是有限，但是应用程序的使用需求是无限的，操作系统为了解决这个矛盾，使用了虚拟内存的设计。简单的描述就是，给应用程序一个与物理内存无关的虚拟地址空间，并提供一套映射机制，将虚拟地址映射到物理内存。当然应用程序是不知道有这个映射机制存在的，他唯一需要做的就是尽情的使用自己的虚拟地址空间。操作系统提供的映射机制是运行时动态进行虚拟地址和物理地址之间的映射的，当一个虚拟地址没有对应的物理内存时候，映射机制就分配物理内存，构建映射表，满足应用程序的需求，这个过程就叫缺页错误。与直接访问物理内存不同，缺页错误过程大部分是由软件完成的，消耗时间比较久，所以是影响性能的一个关键指标。Linux 把缺页错误又进一步分为次要缺页错误和主要缺页错误。前面提到的分配物理内存，构建映射表过程可以看做是次要缺页错误。主要缺页错误是由 swap 机制引入的，对于 swap 情况，地址映射好了后，还需要从外部存储读取数据，这个过程涉及到 IO 操作，耗时更久。

3. 属性 Network：表示网络相关的统计数据。这个属性对应结构体 NetworkStats：

其中属性 Time 表示统计数据更新时间，属性 RxBytes 表示接收到的字节数，属性 RxErrors 表示接收错误数，属性 TxBytes 表示发送出去的字节数，属性 TxErrors 表示发送错误数。

4. 属性 Fs：表示 Kubernetes 模块使用文件系统相关的统计数据。这个属性对应结构体 FsStats：



在 kubernetes1.4 中新增加了 InodesFree 和 Inodes 这两个属性。其中属性 AvailableBytes 表示文件系统可以使用的存储空间大小，属性 CapacityBytes 表示文件系统存储空间总量，属性 UsedBytes 表示使用文件系统上特定任务所占用的存储空间大小，所以 UsedByte 并不等于 CapacityBytes-AvailableBytes，属性 InodesFree 表示文件系统上空闲 inode 数量，属性 Inodes 表示文件系统上 inode 总量。

什么是 inode：

文件储存在硬盘上，硬盘的最小存储单位叫做”扇区”。每个扇区储存 512 字节。操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个”块”。这种由多个扇区组成的”块”，是文件存取的最小单位。”块”的大小，最常见的是 4KB，即连续八个扇区组成一个块。文件数据都储存在”块”中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做 inode，中文称作”索引节点”。

5. 属性 Runtime：表示用户容器运行时的统计数据。这个属性对应结构体 RuntimeStats:

其中属性 ImageFs 对应结构体 FsStats，这个属性表示文件系统上容器使用镜像所占用空间的统计数据。对于节点来说可以有两种文件系统，一种是 root 文件系统，提供给 kubelet 存放日志、提供数据卷等使用；一种是 image 文件系统，提供给容器存储镜像、读写操作使用。image 文件系统可以就是 root 文件系统，也可以是单独提供给容器镜像使用的文件系统，默认情况下就是 root 文件系统。

在 Kubernetes1.4 之前版本中只有 MemoryPressure 这个健康状况类型，对应节点上结构体 MemoryStats 的 AvailableBytes 属性。在 Kubernetes1.4 中，结构体

NodeCondition 新增了一个健康状况类型 DiskPressure，其实这个 DiskPressure 就对应 root 文件系统和 image 文件系统。更具体的说，就是对应 root 文件系统中和 image 文件系统的 AvailableBytes 属性和 inodesFree 属性。

当满足 DiskPressure 发生的条件时，kubelet 模块回收这个节点上的所有 POD，将这些 POD 驱逐到其他节点上。这需要在启动 kubelet 模块时增加 eviction 相关的参数，比如

–eviction-hard：表示立即进行 POD 资源回收，并将 POD 驱逐到其他空闲节点上

–eviction-soft：表示先观察一段时间，之后进行 POD 资源回收，并将 POD 驱逐到其他空闲节点上

–eviction-soft-grace-period：表示观察时长

下面是 kubelet 模块设置使用 DiskPressure 的例子：

```
--eviction-hard="nodefs.available<1Gi,nodefs.inodesFree<1,imagefs.available<10Gi,imagefs.inodesFree<10"
```

当节点上 root 文件系统可用空间小于 1G 或者空闲 inode 数小于 1，或者当节点上 image 文件系统可用空间小于 10G 或者空闲 inode 数小于 10，那么就会设置 DiskPressure 为 true，kubelet 模块接口会立即回收这个节点上的 POD，并将 POD 驱逐到其他空闲节点上。

```
--eviction-soft="nodefs.available<1.5Gi,nodefs.inodesFree<10,imagefs.available<20Gi,imagefs.inodesFree<100"

--eviction-soft-grace-period="nodefs.available=1m,imagefs.available=2m"
```

当节点上 root 文件系统可用空间小于 1.5G 或者空闲 inode 数小于 10，并且 1 分钟后还是这样；或者当 image 文件系统可用空间小于 20G 或者空闲 inode 数小于 100，并且 2 分钟后还是这样，那么就会设置 DiskPressure 为 true，kubelet 模块接口会回收这个节点上的 POD，并将 POD 驱逐到其他空闲节点上。

总结

我们可以看到 kubernetes 正在快速的丰富和完善自身功能，从只有 MemoryPressure，到增加了 DiskPressure，kubernete 给使用者提供了更多方式来处理应用容器化会遇到的问题。我们通过上面结构体的介绍可以发现，还可以继续丰富其他类型的 Pressure，比如 CPUPress、NetworkPress，这些都需要 kubernets 社区继续去完善，相信随着 kubernetes 新版本的发布，功能会变得越来越强大。

kubernetes1.4 支持 Docker 新特性

（一）背景资料

- 在 Kubernetes1.2 中这个第三方组件就是 go-dockerclient，这是一个 GO 语言写的 docker 客户端，支持 Dockerremote API，这个项目在 https://github.com/fsouza/go-dockerclient 中。
- 在 Kubernetes1.3 中直接使用 docker 公司提供的 client 来实现，通过这个 client 可以实现同 DockerDeamon 之间的通讯，这个客户端项目在 https://github.com/docker/engine-api/中，感兴趣的话可以去看看。
- 在 Kubernetes1.4 中延用了 1.3 中的方式，直接使用 docker 公司提供的 client 来实现。

（二）支持 Docker 版本

- 对于 Kubernetes1.4 需要使用 Docker 版本至少是 1.9.x，Docker1.9.x 对应的 API 版本是 1.21。
- 下面是 Docker 版本同 API 版本对应关系，其中红色字体的部分是 Kubernetes1.4 不支持的。

Docker 版本	API 版本
1.12x	1.24
1.11.x	1.23

1.10.x	1.22
1.9.x	1.21
1.8.x	1.20
1.7.x	1.19
1.6.x	1.18
1.5.x	1.17
1.4.x	1.16
1.3.x	1.15
1.2.x	1.14

（三）调用 Docker API

下面表格展现了 Docker 最新版本所有的 API 列表，同时也展现了 **Kubernetes1.4 版本**和 1.3 版本都使用了哪些 API。

- 第一列是 Docker 1.24 版本 API 列表
- 第二列是这些 API 使用方式
- 第三列是 Kubernetes1.4 中使用到的 API
- 第四列是 Kubernetes1.3 中使用到的 API
- 红色字体部分为 1.4 版本比 1.3 版本增加的调用 API，也就是说 1.4 版本比 1.3 版本增加的操作 Docker 的功能

Docker API 1.24	使用方式	Kubernetes1.4	Kubernetes1.3
Get Container stats	GET /containers/(id)/stats		

based on resource usage			
Update a container	POST /containers/(id)/update		
Rename a container	POST /containers/(id)/rename		
Retrieving information about files and folders in a container	HEAD /containers/(id)/archive		
List containers	GET /containers/json	√	√
Inspect a container	GET /containers/(id)/json	√	√
Inspect changes on a container’s filesystem	GET /containers/(id)/changes	√	√
Create a container	POST /containers/create	√	√
Start a container	POST /containers/(id)/start	√	√
Stop a container	POST /containers/(id)/stop	√	√
Restart a container	POST /containers/(id)/restart		

Pause a container	POST /containers/(id)/pause		
Unpause a container	POST /containers/(id)/unpause		
List processes running inside a container	GET /containers/(id)/top		
Kill a container	POST /containers/(id)/kill	√	√
Remove a container	DELETE /containers/(id)	√	√
Get an archive of a filesystem resource in a container	GET /containers/(id)/archive		
Extract an archive of files or folders to a directory in a container	PUT /containers/(id)/archive		
Copy files or folders from a container	POST /containers/(id)/copy，以后会被删除掉，使用 archive 代替		
Wait a container	POST /containers/(id)/wait		
Create a new image from a	POST /commit		

container’ s changes			
Attach to a container	POST /containers/(id)/attach	√	√
Attach to a container (websocket)	GET /containers/(id or name)/attach/ws		
Get container logs	GET /containers/(id)/logs	√	√
Resize a container TTY	POST /containers/(id)/resize	√	
Export a container	GET /containers/(id)/export		
List Images	GET /images/json	√	√
Inspect an image	GET /images/(name)/json	√	√
Get the history of an image	GET /images/(name)/history	√	√
Push an image on the registry	POST /images/(name)/push		
Build image from a Dockerfile	POST /build		
Create an image	POST /images/create	√	√

Load a tarball with a set of images and tags into docker	POST /images/load		
Get a tarball containing all images in a repository	GET /images/(name)/get		
Get a tarball containing all images	GET /images/get		
Tag an image into a repository	POST /images/(name)/tag		
Remove an image	DELETE /images/(name)	√	√
Search images	GET /images/search		
Monitor Docker’s events	GET /events		
Show the docker version information	GET /version	√	√
Display system-wide information	GET /info	√	√

Ping the docker server	GET /_ping		
List volumes	GET /volumes		
Create a volume	POST /volumes/create		
Inspect a volume	GET /volumes/(name)		
Remove a volume	DELETE /volumes/(name)		
List networks	GET /networks		
Inspect network	GET /networks/<network-id>		
Create a network	POST /networks/create		
Remove a network	DELETE /networks/(id)		
Connect a container to a network	POST /networks/(id)/connect		
Disconnect a container from a network	POST /networks/(id)/disconnect		
Check auth configuration	POST /auth		

Exec Create	POST /containers/(id)/exec	√	√
Exec Start	POST /exec/(id)/start	√	√
Exec Resize	POST /exec/(id)/resize	√	
Exec Inspect	GET /exec/(id)/json	√	√
List plugins	GET /plugins		
Install a plugin	POST /plugins/pull?name=<plugin name>		
Inspect a plugin	GET /plugins/(plugin name)		
Enable a plugin	POST /plugins/(plugin name)/enable		
Disable a plugin	POST /plugins/(plugin name)/disable		
Remove a plugin	DELETE /plugins/(plugin name)		
List nodes	GET /nodes		
Inspect a node	GET /nodes/<id>		
Remove a node	DELETE /nodes/<id>		
Update a node	POST /nodes/<id>/update		

Inspect swarm	GET /swarm		
Initialize a new swarm	POST /swarm/init		
Join an existing swarm	POST /swarm/join		
Leave a swarm	POST /swarm/leave		
Update a swarm	POST /swarm/update		
List services	GET /services		
Create a service	POST /services/create		
Remove a service	DELETE /services/(id or name)		
Inspect one or more services	GET /services/(id or name)		
Update a service	POST /services/(id or name)/update		
List tasks	GET /tasks		
Inspect a task	GET /tasks/(task id)		

1) 从表格中可以看到，Kubernetes1.4 中调用了 Docker 的 Resize a container TTY 接口，用来配置 Docker 容器的虚拟终端（TTY），重新设置 Docker 容器的虚拟终端之后，需要重新启动容器才能生效。

HTTP 请求例子：

```
POST/containers/4fa6e0f0c678/resize?h=40&w=80 HTTP/1.1
```

返回响应例子：

```
HTTP/1.1 200 OK

Content-Length: 0

Content-Type: text/plain; charset=utf-8
```

请求参数：

h – 虚拟终端高度

w – 虚拟终端宽度

HTTP 返回响应状态值：

200 – 设置成功

404 – 没有找到指定 Docker 容器

500 – 不能够重新设置虚拟终端参数

2) 从表格中还可以看到，Kubernetes1.4 中调用了 Docker 的 Exec Resize 接口，如果在 Docker 容器中执行 exec 命令时指定了虚拟终端（tty），那么通过这个 API 接口就可以重新设置虚拟终端（tty）。

HTTP 请求例子：

```
POST/exec/e90e34656806/resize?h=40&w=80 HTTP/1.1

Content-Type: text/plain
```

返回响应例子：

```
HTTP/1.1201 Created

Content-Type: text/plain
```

请求参数：

h –虚拟终端高度

w –虚拟终端宽度

HTTP 返回响应状态值：

201 –设置成功

404 –没有找到指定 exec 实例

3)　Kubernetes1.4 新增加了上面两个接口调用，可以看看这两个接口调用在源代码中的位置：

```
func AttachContainer(client DockerInterface,containerID kubecontainer.Contai
nerID, stdin io.Reader, stdout, stderrio.WriteCloser, tty bool, resize <-cha
n term.Size) error {

<span style="color:#FF0000;"> kubecontainer.HandleResizing(resize, func(siz
e term.Size) {

 client.ResizeContainerTTY(containerID.ID,int(size.Height), int(size.Widt
h))

       })</span>

       opts:= dockertypes.ContainerAttachOptions{

              Stream:true,

              Stdin:  stdin != nil,

              Stdout:stdout != nil,

              Stderr:stderr != nil,

       }

       ……

}

func (*NativeExecHandler)ExecInContainer(client DockerInterface, container *
dockertypes.ContainerJSON,cmd []string, stdin io.Reader, stdout, stderr io.W
riteCloser, tty bool, resize<-chan term.Size) error {
```

```
        ……

       <span style="color:#FF0000;"> kubecontainer.HandleResizing(resize, f
unc(size term.Size) {

               client.ResizeExecTTY(execObj.ID, int(size.Height),int(size.W
idth))

        })</span>

        startOpts:= dockertypes.ExecStartCheck{Detach: false, Tty: tty}

        streamOpts:= StreamOptions{

               InputStream:  stdin,

               OutputStream:stdout,

               ErrorStream:  stderr,

               RawTerminal:  tty,

        }

        err= client.StartExec(execObj.ID, startOpts, streamOpts)

        iferr != nil {

               returnerr

        }

  ……

  }
```

这两处开发开发人员的注释如下：

Have to start this before the call toclient.AttachToContainer because client.AttachToContainer is a blocking call:-(Otherwise, resize events don’t get processed and the terminal neverresizes.

Have to start this before the call toclient.StartExec because client.StartExec is a blocking call Otherwise,resize events don’t get processed and the terminal never resizes.

通过注释可以发现，因为 attach 和 start exec 两个接口都是可以阻塞的，所以通过增加设置虚拟终端（tty）来判断向 attach 和 start exec 两个接口发送的请求是否阻塞。

4) 从表格中还可以看到，Kubernetes 没有使用到 Docker 的网络接口，也没有使用到 Docker 的卷接口，原因是 Kubernetes 自己定义了 Service 和 POD，自己实现了 POD 之间的网络和挂载到 POD 上的卷。

5) 从表格中也可以看到，Kubernetes 对 Docker 容器的管理只有很少的功能，甚至都没有使用到 Docker 的重启接口，还是因为 Kubernetes 自己定义的 POD，Kubernetes 以 POD 为基本操作单元，而且是 Kubernetes 从容器集群管理角度设计的，所以不存在对 POD 里面单个 Docker 容器的重启操作。

（四）对 Docker 其他操作

1) Linux ext4 文件系统要求文件名字符个数不能超过 255，在 Kubernetes1.4 中进行了控制。

2) 由于 Docker1.12 支持了通过—sysctl 参数来设置内核参数，所以在 Kubernetes1.4 可以将安全的 sysctl 命令放入白名单列表中，这样就可以对容器内核参数进行配置操作，下面是 Kubernetes1.4 对内核参数的默认设置：

- sysctl -wvm.overcommit_memory=1

表示节点上有多少物理内存就用多少，不进行内存超分配

- sysctl -w vm.panic_on_oom=0

表示当节点上物理内存耗尽时，内核触发 OOM killer 杀掉最耗内存的进程

- sysctl -w kernel/panic=10

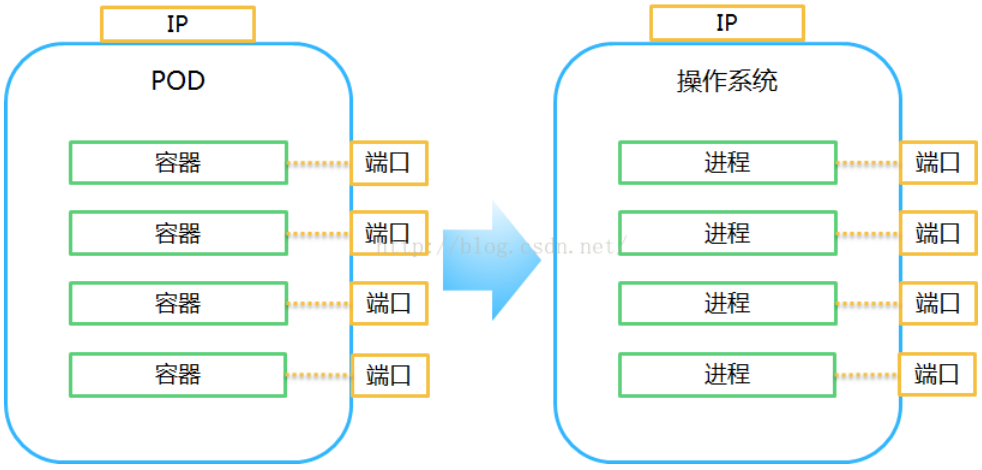
表示当内核 panic 时，等待 10 秒后在重启

- sysctl -wkernel/panic_on_oops=1

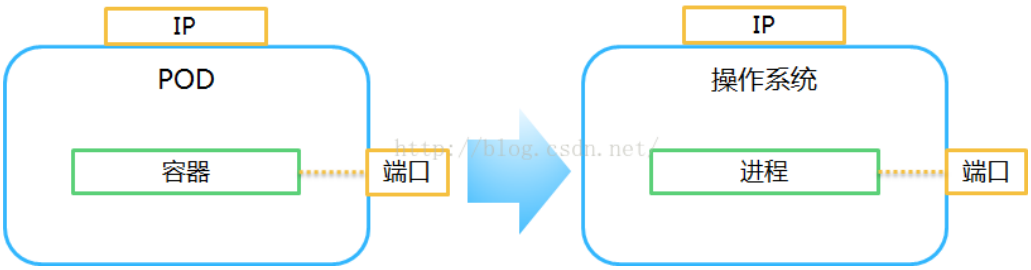
设置当内核 oops 发生时，采用 panic 方式处理。不同于 panic 会导致 OS 重启，设备驱动引发的 oops 通常不会导致 OS 重启，但是可以通过设置这个参数来指定 oops 发生时进行 OS 重启

（五）K8S 操作 Docker 的网络实现

我们可以用下面这张图来把 POD 和容器之间的关系形象化，此图仅供说明问题时的参考，并无实际意义。



我们可以把 POD 看作是机器里面的操作系统，把容器看作是里面的进程，在操作系统内部进程间是可以通过 IPC（Inter-Process Communication）进行通讯的，不同操作系统之间的进程是通过操作系统 IP 和端口进行通讯的，那么对应到 POD 和容器，就变成了 POD 内部容器间事可以通过 IPC（Inter-Process Communication）进行通讯的，不同 POD 之间的容器是通过 POD IP 和端口进行通讯的。从集群的角度来考虑问题，Kubernetes 基本操作单元是 POD，不需要关注到 POD 中的容器，那么我们可以想象一下，如果我们要按照虚拟机的使用方式来使用容器，那样的话应该如何使用 Kubernetes 呢？可以看下面的图：



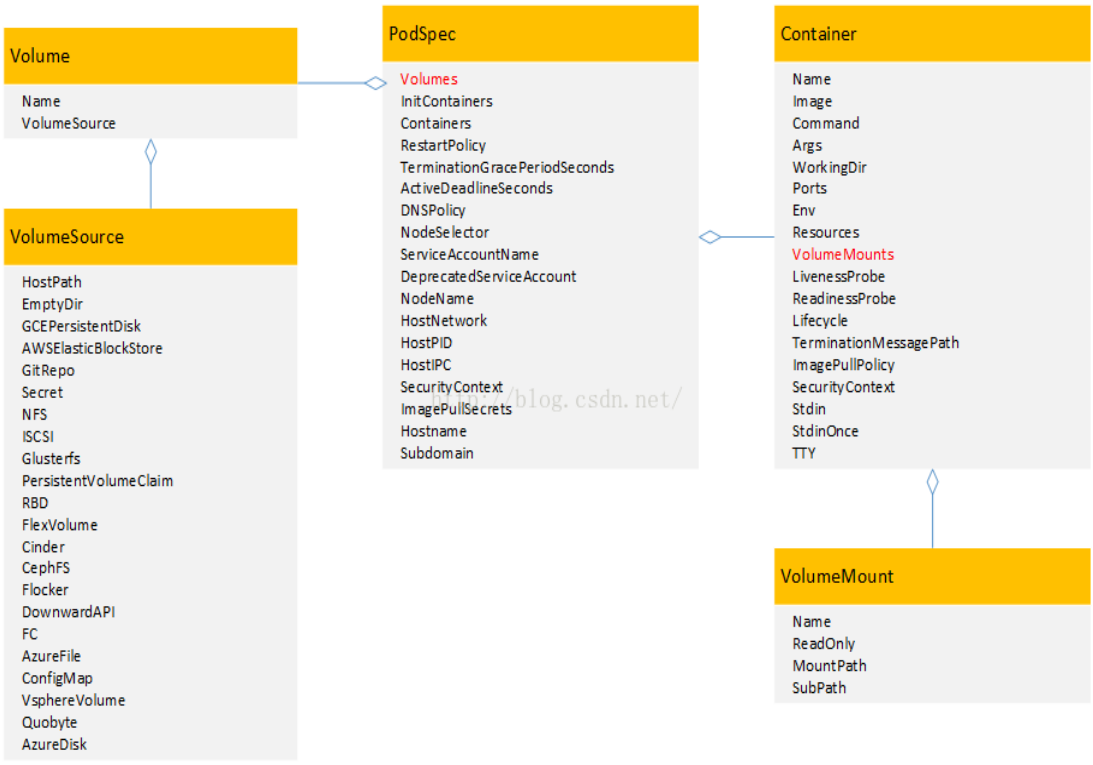
这样我们就实现了像虚拟机那样使用容器，我们可以认为 POD 就是一个虚拟机，只不过在这个虚拟机中只有一个容器。但是如果要对这个虚拟机进行操作的时

候我们发现问题来了，从前面的表格中可以看到，Kubernetes 对 Docker 容器的管理只有很少的功能，没有使用到 Docker 的重启接口，但是如果把容器当作虚拟机用，必然要使用重启功能，抛开 Kubernetes 的设计理念，我们自己可以扩展 Kubernetes 对 **POD** 重启的实现，实现把容器当作虚拟机来使用的需求。

kubernetes1.4 支持两种新的卷插件 Quobyte 和 AzureDisk

背景介绍

在 Kubernetes 中卷的作用在于提供给 **POD** 持久化存储，这些持久化存储可以挂载到 POD 中的容器上，进而给容器提供持久化存储。

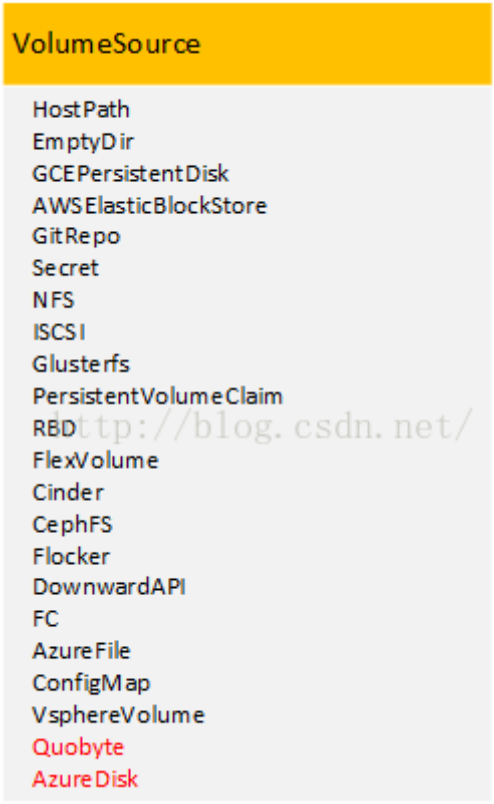


从图中可以看到结构体 PodSpec 有个属性是 Volumes，通过这个 Volumes 属性可以关联到结构体 Volume 和结构体 VolumeSource，而且这个 Volumes 属性是一个数组类型，就是说 POD 可以关联到多个不同类型的卷上面。

结构体 Container 表示 POD 中的容器，这个结构体有一个属性 VolumeMounts，通过这个属性让容器知道具体挂载的存储路径，这个 VolumeMounts 属性也是一个数组类型，就是说容器可以挂载多个存储路径。

支持新的卷插件

Kubernetes 一共支持 22 种卷插件。在 Kubernetes1.4 中又新增了两种新的卷插件：Quobyte 和 AzureDisk。

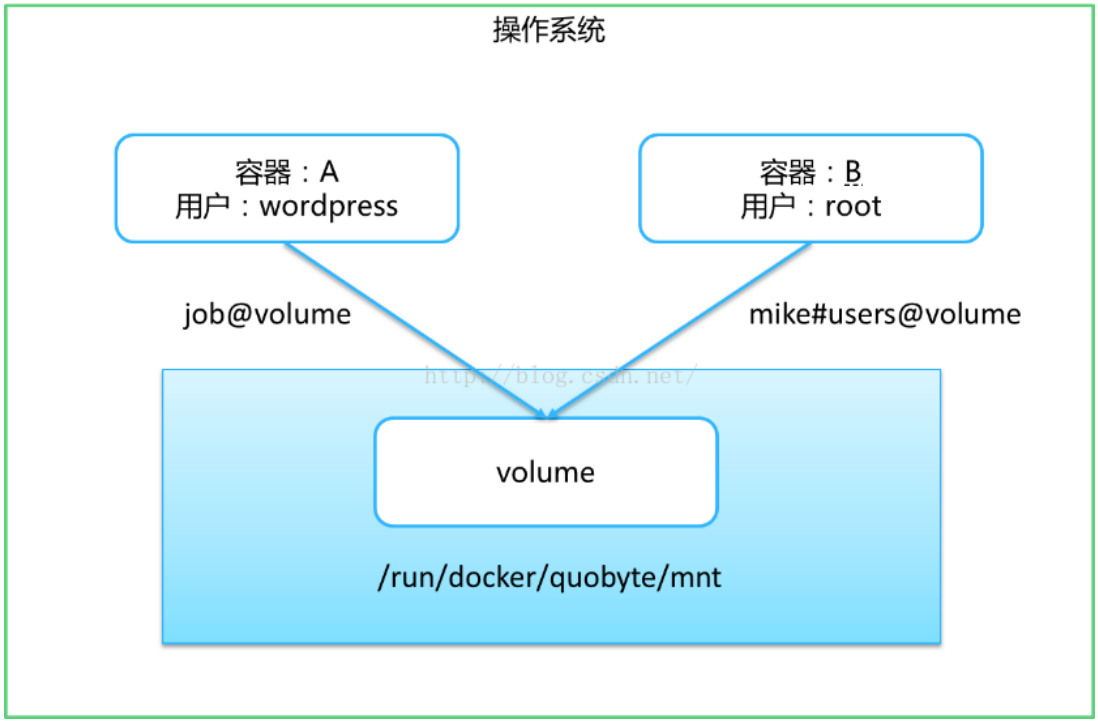


Quobyte

这是 Quobyte 公司推出的分布式文件系统。要想在 **kubernetes** 中使用 Quobyte 存储，需要提前部署 Quobyte 软件，要求必须是 1.3 以及更高版本，并且在 kubernetes 管理的节点上面部署 Quobyte 客户端。

为什么要使用 1.3 以及更高版本？因为这些版本中 Quobyte 提供了一个特性：fixed-user 挂载，这个新特性允许所有 Quobyte 卷都会被挂载到一个路径下面，同时又可以按照不同用户来进行分别使用。所有对 Quobyte 卷的访问都是区分用户和

组的，当然也可以不区分，区分的好处就是可以实现多租户隔离。下图是一个 fixed-user 挂载的示例：



容器 A 设计成只能通过 wordpress 这个用户来执行操作，但是对于 quobyte 来说，容器 A 实际上是通过用户 job 来执行实际读写操作的；容器 B 设计成通过 root 用户来执行操作，但是对于 quobyte 来说，容器 B 实际上是通过 root 组中的用户 mike 来执行操作的。也就是说做了个用户映射，而不是使用容器内的用户来执行读写操作。

如何启用 fixed-user 挂载这个特性?安装完 Quobyte 客户端后，在配置文件 /etc/quobyte/client.cfg 中增加一行 allow-usermapping-in-volumename。

如果不使用 kubernetes，直接使用 Docker，那么也可以不安装 Quobyte 客户端，而是使用 Quobyte 提供的 Docker 插件，在 Docker 中，所有 Quobyte 卷都会挂载到/run/docker/quobyte/mnt 目录下。这个插件在下面环境中已经测试过：

OS	Docker Version
CentOS 7.2	1.10.3
Ubuntu 16.04	1.11.2
Ubuntu 16.04	1.12.0
CoreOS 1097.0.0	1.11.2

但是比较遗憾的是 Quobyte 提供的 Docker 插件现在还不支持 fixed-user 挂载这个特性，如果想使用这个特性，就必须安装 Quobyte 客户端。

下面是 quobyte 卷对应的结构体：

- 变量 Registry：QuoByte 注册服务入口，如果配置了多个注册服务入口，那么每个注册服务入口可以通过分号分割。

- 变量 Volume：QuoByte 已经创建好的卷。

- 变量 ReadOnly：这是一个布尔型变量，默认是 false，表示可以对 QuoByte 已经创建好的卷进行读写操作，如果配置成 true，那么就表示只能对 QuoByte 已经创建好的卷进行只读操作。

- 变量 User：操作 QuoByte 卷的用户。

- 变量 Group：操作 QuoByte 卷的组。

下面是使用 Quobyte 卷创建 POD 的示例文件 quobyte-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: quobyte
spec:
  containers:
  - name: quobyte
    image: kubernetes/pause
    volumeMounts:
    - mountPath: /mnt
      name: quobytevolume
  volumes:
  - name: quobytevolume
    quobyte:
      registry: registry:7861
      volume: testVolume
      readOnly: false
      user: root
      group: root
```

接着通过下面命令创建 POD：

```
$kubectl create -f ./quobyte-pod.yaml
```


在 POD 创建成功后，可以检查确认 quobyte 卷 testVolume 已经被挂载上了：

```
$ mount | grep quobyte
quobyte@10.10.105.78:7861/ on /var/lib/kubelet/plugins/kubernetes.io~quobyte type fuse
(rw,nosuid,nodev,noatime,user_id=0,group_id=0,default_permissions,allow_other)

$ docker inspect --format '{{ range .Mounts }}{{ if eq .Destination "/mnt"}}{{ .Source }}{{ end }}{{ end
}}' 66ba20703cc3
/var/lib/kubelet/plugins/kubernetes.io~quobyte/root#root@testVolume
```


AzureDisk

Azure 是微软提供的公有云服务，如果使用 Azure 上面的虚拟机来作为 Kubernetes 集群使用时，那么可以通过 AzureDisk 这种类型的卷插件来挂载 Azure 提供的数据磁盘。

下面是 Azure 上数据磁盘的介绍：

数据磁盘是附加到虚拟机的 VHD，用于存储应用程序数据或其他需要保留的数据。数据磁盘注册为 SCSI 驱动器并且带有所选择的字母标记。每个数据磁盘的最大容量为 1023 GB。虚拟机的大小决定了可附加的磁盘数目，以及可用来托管磁盘的存储类型。Azure 中使用的 VHD 是在 Azure 的标准或高级存储帐户中作为页 Blob 存储的 .vhd 文件。

Azure 数据磁盘的核心就是 Blob，Blob 就是保存大型二进制对象，比如用来存储文件、图片、文档等二进制格式的文件。

Blob 分为两种类型：

1、Block Blob （块 Blob）。这种类型适合存储二进制文件，支持断点续传，可以最大以 4M 为一个区块单位，单一文件最大可以存储 200GB，且区块不会连续存储，可能会在不同的存储服务器分块存放。为了适应文件的上传和下载而专门进行了优化。Block Blob 可以通过 2 种方式创建。不超过 64MB 的 Block Blobs 可以通过调用 PutBlob 操作进行上传。大于 64M 的 Block Blobs 必须分块上传，且每块的大小不能超过 4MB。Block Blob 可以近似理解为网盘。

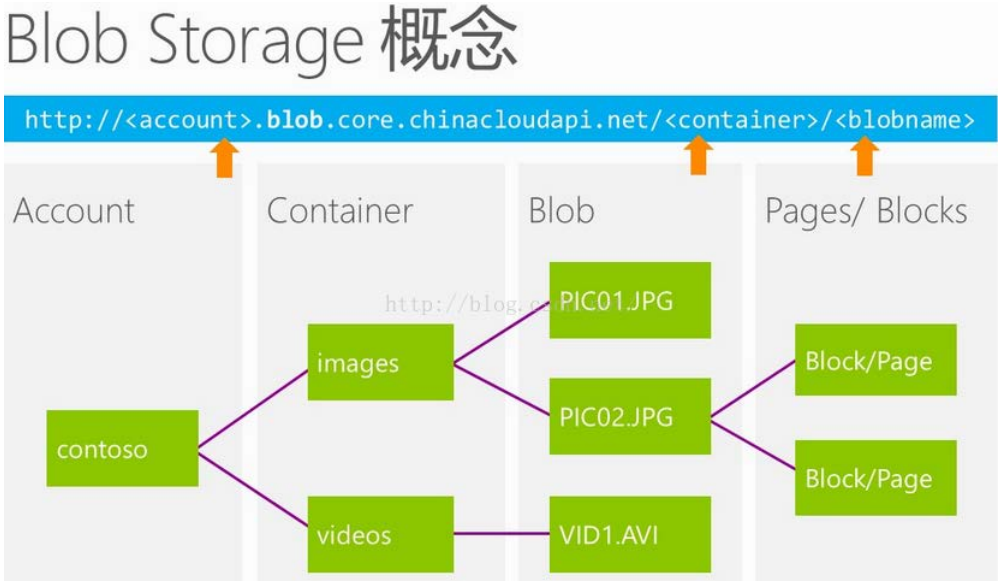
2、Page Blob （页 Blob)。这类存储优化了随机访问。它会在存储区中划分一个连续的区域供应用程序存放数据，可以用来存放 VHD，单一文件最大可以存储 1TB。

Blob 服务由 Blob 本身以及其收纳容器 (Container) 构成，容器可以视为一般本机上的文件夹。

你可以通过 REST API 来访问 Blob：

http://.blob.core.chinacloudapi.cn//

- accountname 表示哪个 Azure 存储账号下的资源，是全局唯一的。
- blob.core.chinacloudapi.cn 表示 azure chinablob 存储资源，是固定的。
- containername 表示容器的名字，可以认为是访问某一文件夹下的资源
- blobname 表示要访问的资源名称，可以认为是一个 mp3 文件，或者是一个 jpg 文件。



举例说明：

保存在 leizhangstorage 存储账号下，containername 为 photo，blobname 为 myphoto.jpg。则这个 URL 地址为：

http://leizhangstorage.blob.core.chinacloudapi.cn/photo/myphoto.jpg

保存在 leizhangstorage 存储账号下，containername 为 vhd，blobname 为 myvm.vhd。则这个 URL 地址为：

http://leizhangstorage.blob.core.chinacloudapi.cn/vhd/myvm.vhd

Container 的命名规则:

- containername 只能是一级目录，没有办法在 containername 下再设置下一级别 containername
- 必须以英文或数字开头，且名称内只能有英文、数字及 dash(-)
- 不能以 dash(-) 开头或结尾，dash(-) 不能连续出现
- 所有英文的字符必须是小写
- 长度为 3-63 之间

Blob 的命名规则：

- 除了 url 的保留字符以外，其他的字符组合都可以使用

- 长度为 1-1024 个字符

- 尽量避免以 dot(.) 或者是 forward slash(/) 结尾。否则会造成 Blob Service 误判。

下面是 Kubernetes 中 AzureDisk 卷对应的结构体：

- 变量 DiskName：必选参数，表示数据磁盘的名称。

- 变量 DataDiskURI：必选参数，表示数据磁盘的访问路径。

- 变量 CachingMode：可选参数，表示数据磁盘缓冲模式，可以选择 None、ReadOnly 和 ReadWrite，默认是 None。

- 变量 FSType：可选参数，表示数据磁盘挂载到操作系统上之后格式化成的文件系统类型，比如：”ext4″、”xfs”、”ntfs”，默认是”ext4″。

- 变量 ReadOnly：可选参数，这是一个布尔型变量，表示数据磁盘是否为只读使用，默认是 false，表示可以进行读写操作。

下面是使用 AzureDisk 卷创建 POD 的示例文件 azuredisk-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
 name: azure
spec:
 containers:
  - image: kubernetes/pause
    name: azure
    volumeMounts:
      - name: azure
        mountPath: /mnt/azure
 volumes:
      - name: azure
        azureDisk:
          diskName: test.vhd
          diskURI: https://someaccount.blob.microsoft.net/vhds/test.vhd
```

接着通过下面命令创建 POD：

```
$kubectl create -f ./azuredisk-pod.yaml
```


总结

Kubernetes1.4 一共支持 22 种卷插件，从这些卷插件就可以看出 Kubernetes 社区参与厂家越来越多了，这 22 种卷插件可以覆盖 GoogleCompute Engine 公有云、Amazon WebService 公有云、Microsoft Azure 公有云、基于 OpenStack 的公有云、基于 VMware vSphere 的私有云，在 Kubernetes 1.4 版本中又加入了第三方商用分布式存储厂商 Quobyte 的支持，可以看出来，Kubernetes 的影响力在扩大，将来一定会有更多厂商提供对 Kubernetes 的支持。

从这么多钟卷插件也可以看出来，Kubernetes1.3 开始推出了跨云的 Kubernetes 集群管理特性：“集群联盟”，也就意味着 Kubernetes 将来会利用目世界范围内公有云 IaaS 资源，成为一个可以横跨不同公有云 IaaS 资源的超级集群管理工具，实现在全球快速部署和管理应用。

kubernetes1.4 支持 sysctl 命令

背景介绍

sysctl 是一个允许改变正在运行中的 Linux 系统内核参数的接口。可以通过 sysctl 修改 Linux 系统内核中的 TCP/IP 堆栈和虚拟内存系统的高级选项，而且不需要重新启动 Linux 系统，就可以实现优化 Linux 系统和提高应用进程运行性能。

通过 Linux 系统中的/proc 虚拟文件系统来实现动态配置 Linux 系统内核参数，在/proc/sys 目录下有 Linux 系统绝大多数的内核参数，这些内核参数可以在 Linux 系统运行时进行修改，并且不需要重新启动 Linux 系统便可以立刻生效，但是这种修改在重新启动 Linux 系统后便会失效，要是想永久生效的话，需要更改配置文件/etc/sysctl.conf 中相应的内核参数配置项。

可以通过下面命令获取 sysctl 可以操作的所有内核参数配置项和已经配置的数值：

sysctl –a

```
kernel.sched_latency_ns = 24000000
kernel.sched_migration_cost_ns = 500000
kernel.sched_min_granularity_ns = 10000000
kernel.sched_nr_migrate = 32
kernel.sched_rr_timeslice_ms = 100
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
kernel.sched_shares_window_ns = 10000000
kernel.sched_time_avg_ms = 1000
kernel.sched_tunable_scaling = 1
kernel.sched_wakeup_granularity_ns = 15000000
kernel.sem = 250        32000   32      128
kernel.shm_rmid_forced = 0
kernel.shmall = 268435456
kernel.shmmax = 4294967295
kernel.shmmni = 4096
kernel.softlockup_panic = 0
kernel.stack_tracer_enabled = 0
kernel.sysrq = 16
kernel.tainted = 0
kernel.threads-max = 126947
kernel.timer_migration = 1
kernel.unknown_nmi_panic = 0
kernel.usermodehelper.bset = 4294967295 31
kernel.usermodehelper.inheritable = 4294967295  31
kernel.version = #1 SMP Fri Mar 6 11:36:42 UTC 2015
kernel.watchdog = 1
kernel.watchdog_thresh = 10
```


这些内核参数主要包括下面几类配置项：

- 全局内核配置项：以“**kernel.**”为配置项前缀，举例：

1. **kernel.shmmax** = 33554432（共享内存段的最大尺寸，以字节为单位）
2. **kernel.threads-max** = 139264（Linux 内核所能使用的线程最大数量）

- 网络配置项：以“**net.**”为配置项前缀，举例：

1. **net.ipv4.ipfrag_low_thresh** = 196608（用于 IP 分片汇聚的最小内存用量）
2. **net.ipv4.ipfrag_high_thresh** = 262144（用于 IP 分片汇聚的最大内存用量）

- 虚拟内存配置项：以“**vm.**”为配置项前缀，举例：

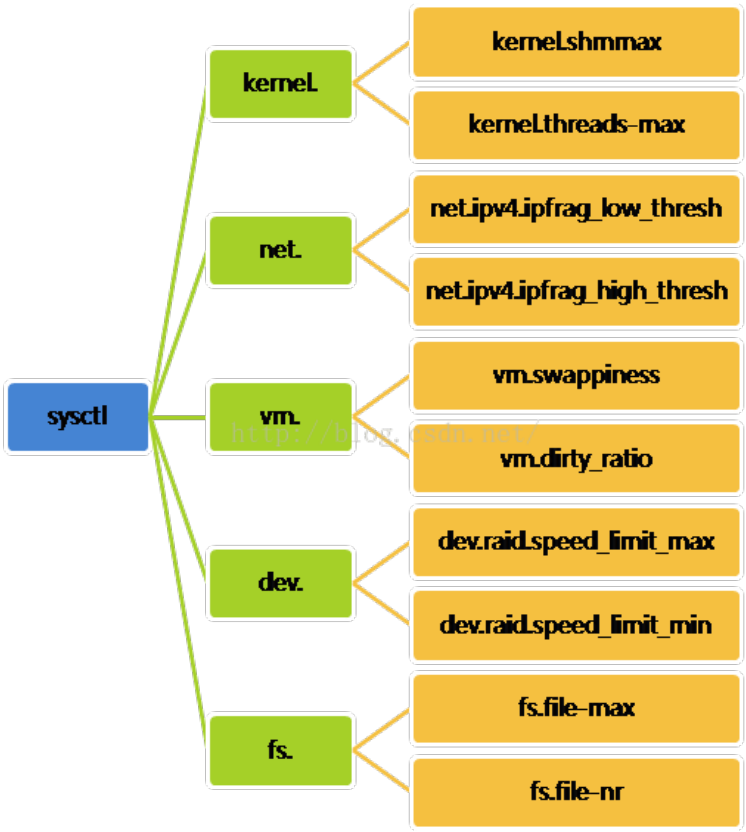
1. **vm.swappiness** = 60（减少系统对于 swap 频繁的写入，将加快应用程序之间的切换，有助于提升系统性能）
2. **vm.dirty_ratio** = 40（该文件表示如果进程产生的废数据到达系统整体内存的百分比，此时进程自信把废数据写回磁盘）

- 设备专用配置项：以“**dev.**”为配置项前缀，举例：

1. **dev.raid.speed_limit_max** = 200000（需要初始化同步 RAID 的同步最大速度限制）
2. **dev.raid.speed_limit_min** = 1000（需要初始化同步 RAID 的同步最小速度限制）

- 文件系统专用配置项：以“**fs.**”为配置项前缀

1. **fs.file-max** = 779703（可以分配的文件句柄的最大数目）
2. **fs.file-nr** = 3930 0 779703（已分配文件句柄的数目，已使用文件句柄的数目，文件句柄的最大数目，该文件是只读的，仅用于显示信息）



容器相关内核参数

上面介绍了通过 sysctl 可以操作 Linux 系统内核参数，在这些内核参数中，有些不但是操作系统全局级别的内核参数，还是命名空间级别的内核参数。对于容器来说，是通过命名空间实现隔离的，那么就意味着这些命名空间级别的参数是容器相关的内核参数。

Linux 系统命名空间的分类如下：

命名空间	隔离内容
UTS	主机名与域名
IPC	信号量、消息队列和共享内存
PID	进程编号
Network	网络设备、网络栈、端口等等
Mount	挂载点（文件系统）
User	用户和用户组

命名空间级别的内核参数包括：

- kernel.shm*（内核中共享内存相关参数），举例：

1. **kernel.shmall** = 3774873（可以使用的共享内存的总量）
2. **kernel.shmmax** = 15461882265（单个共享内存段的最大值）

- kernel.msg*（内核中 SystemV 消息队列相关参数）

1. **kernel.msgmnb** = 16384（每个消息队列的最大字节限制）
2. **kernel.msgmni** = 128（同时运行的最大的消息队列个数）

- kernel.sem（内核中信号量参数）

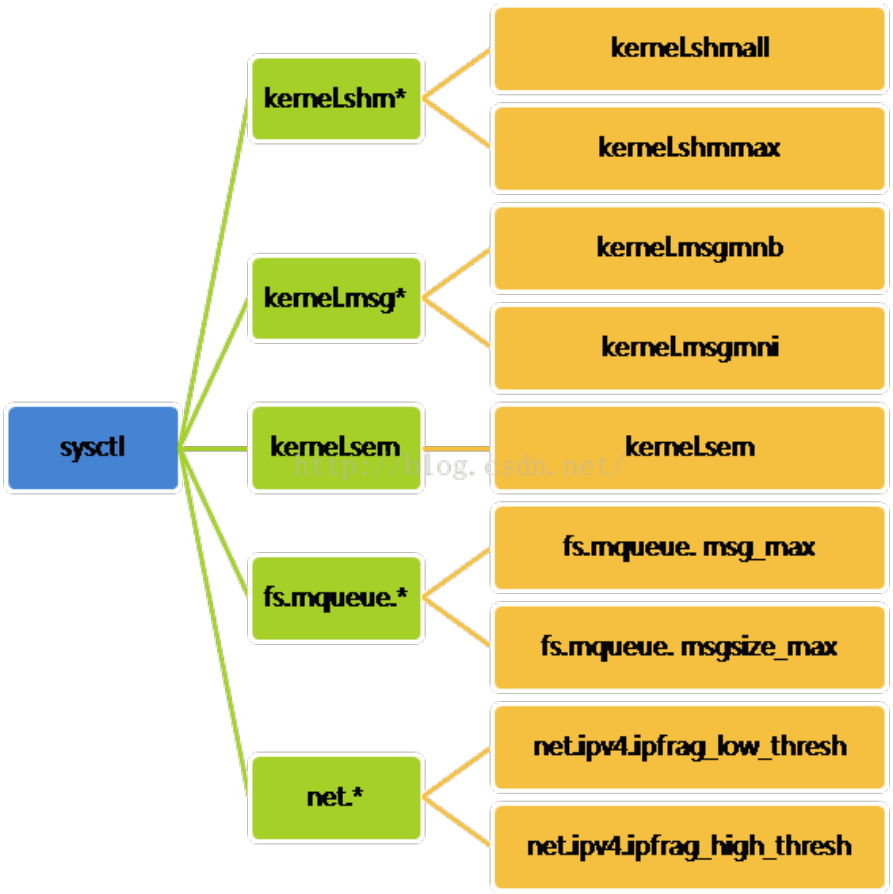
1. **kernel.sem** = 250 32000 100 128（每个信号集中的最大信号量数目、系统范围内的最大信号量总数目、每个信号发生时的最大系统操作数目、系统范围内的最大信号集总数目）

- fs.mqueue.*（内核中 POSIX 消息队列相关参数）

1. **fs.mqueue. msg_max** = 32678（队列里缓存的软最大消息数目）
2. **fs.mqueue. msgsize_max** = 8192（最大消息长度上限）

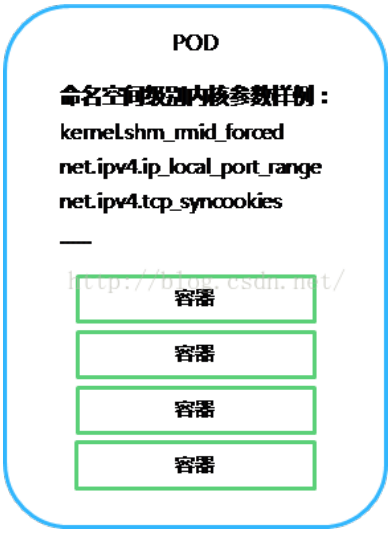
- net.*（内核中网络配置项相关参数）

1. **net.ipv4.ipfrag_low_thresh** = 196608（用于 IP 分片汇聚的最小内存用量）
2. **net.ipv4.ipfrag_high_thresh** = 262144（用于 IP 分片汇聚的最大内存用量）



新特性

因为 sysctl 可以修改命名空间级别的内核参数，所以在 **Kubernetes1.4** 中通过 sysctl 来配置 POD 中 Linux 内核参数的功能，通过修改 POD 中 Linux 内核参数，可以优化 POD 性能，提升 POD 中容器运行效率。在 Kubernetes1.4 中这还是一个阿尔法特性。

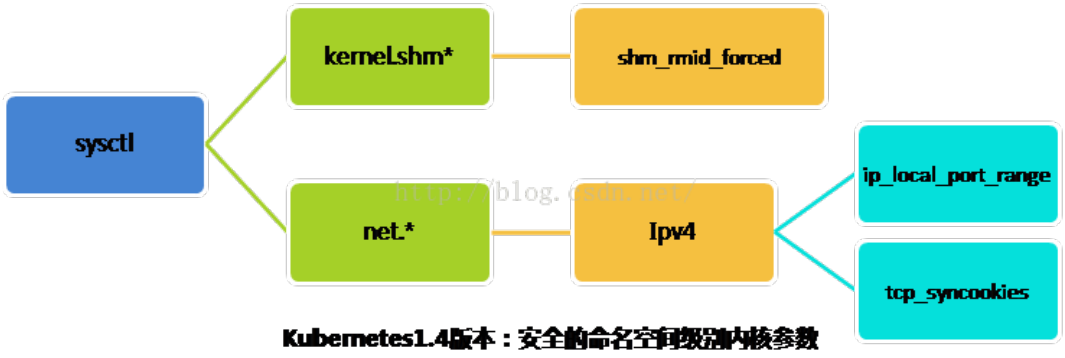


修改 Linux 内核参数存在安全风险，修改错误很可能会降低系统性能，甚至会引起系统崩溃，所以需要谨慎对待。在 Kubernetes1.4 中将命名空间级别的内核参数分成了两类，一类是安全的内核参数，一类是不安全的内核参数。所谓安全的命名空间级别内核参数，就是要能够实现相同节点上不同 POD 之间的完全隔离，要满足如下条件：

1. 不能对相同节点上其他 **POD** 产生任何影响
2. 不能对节点上操作系统健康造成影响
3. 不能在 POD 资源限制以外获取更多的 CPU 和内存资源

根据上面三个条件可以发现，大多数的命名空间级别内核参数都不是安全的。在 Kubernetes1.4 中，认为下面的命名空间级别内核参数是安全的：

1. kernel.shm_rmid_forced = 1（表示是否强制将共享内存和一个进程联系在一起，这样的话可以通过杀死进程来释放共享内存）
2. net.ipv4.ip_local_port_range =1024 65000（表示允许使用的端口范围）
3. net.ipv4.tcp_syncookies = 1（表示是否打开 TCP 同步标签，同步标签可以防止一个套接字在有过多试图连接时引起过载）



Kubernetes1.4版本：安全的命名空间级别内核参数

在 Kubernetes 以后的版本中，还会继续扩充安全的命名空间级别内核参数。在 Kubernetes 中，所有安全的命名空间级别内核参数默认都是启用状态的，所有不安全的命名空间级别内核参数默认都是禁用状态的，如果想设置不安全的内核参数，需要 Kubernetes 管理员手工启用。如果管理员没有手工启用不安全的内核参数，那么 Kubernetes 仍然会进行调度，将这些带有不安全内核参数的 POD 分配到节点上，但是这些 POD 在启动时会失败。

在启动 kubelet 时通过增加参数“experimental-allowed-unsafe-sysctls”来启用不安全的命名空间级别内核参数：

可以在 POD 配置文件中设置已经被启用的命名空间级别内核参数：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  annotations:
    security.alpha.kubernetes.io/sysctls: kernel.shm_rmid_forced=1
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.min_pmtu=1000,kernel.msgmax=1 2 3
spec:
  ...
```

上面的配置文件在 POD 中设置了安全的命名空间级内核参数：kernel.shm_rmid_forced，并且在 POD 中设置了两个不安全的命名空间级内核参数：net.ipv4.route.min_pmte 和 kernet.msgmax。

在 annotations 中的“security.alpha.kubernetes.io/sysctls”参数上设置安全的命名空间级内核参数，在 annotations 中的“security.alpha.kubernetes.io/unsafe-sysctls”参数上设置不安全的命名空间级内核参数。