Kubernetes 入门学习

Kubernetes 是用于自动部署，扩展和管理容器化应用程序的开源系统。
它将组成应用程序的容器组合成逻辑单元，以便于管理和服务发现。Kubernetes 源自Google 15 年生产环境的运维经验，同时凝聚了社区的最佳创意和实践。

Kubernetes 特性

- 自动化上线和回滚
  Kubernetes 会分步骤地将针对应用或其配置的更改上线，同时监视应用程序运行状况以确保你不会同时终止所有实例。如果出现问题，Kubernetes 会为你回滚所作更改。你应该充分利用不断成长的部署方案生态系统。

- 服务发现与负载均衡
  无需修改你的应用程序即可使用陌生的服务发现机制。Kubernetes 为容器提供了自己的 IP 地址和一个 DNS 名称，并且可以在它们之间实现负载均衡。

- 存储编排
  自动挂载所选存储系统，包括本地存储、诸如 GCP 或 AWS 之类公有云提供商所提供的存储或者诸如 NFS、iSCSI、Gluster、Ceph、Cinder 或 Flocker 这类网络存储系统。

- Secret 和配置管理
  部署和更新 Secrets 和应用程序的配置而不必重新构建容器镜像，且不必将软件堆栈配置中的秘密信息暴露出来。

- 自动装箱
  根据资源需求和其他约束自动放置容器，同时避免影响可用性。 将关键性的和尽力而为性质的工作负载进行混合放置，以提高资源利用率并节省更多资源。

- 批量执行
  除了服务之外，Kubernetes 还可以管理你的批处理和 CI 工作负载，在期望时替换掉失效的容器。

- IPv4/IPv6 双协议栈
  为 Pod 和 Service 分配 IPv4 和 IPv6 地址

- 水平扩缩
  使用一个简单的命令、一个 UI 或基于 CPU 使用情况自动对应用程序进行扩缩。

- 自我修复
  重新启动失败的容器，在节点死亡时替换并重新调度容器，杀死不响应用户定义的健康检查的容器，并且在它们准备好服务之前不会将它们公布给客户端。

- 为扩展性设计
  无需更改上游源码即可扩展你的 Kubernetes 集群。

Kubernetes Pods

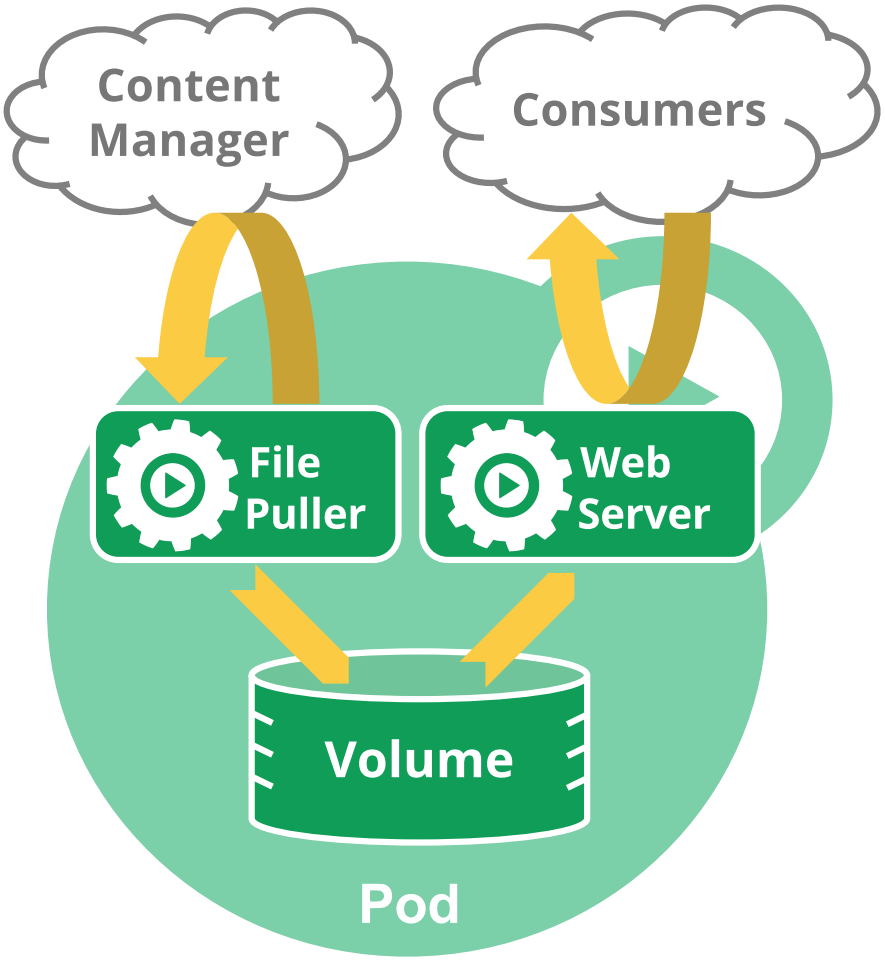
`Pod` 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。

`Pod` 是一组（一个或多个） 容器； 这些容器共享存储、网络、以及怎样运行这些容器的声明。 通常不需要直接创建 Pod，会使用 `Deployment` 或者 `Job` 这类工作负载资源 来创建 `Pod` 。如果 `Pod` 需要跟踪状态，可以考虑 `StatefulSet` 资源。

Kubernetes 集群中的 Pod 主要有两种用法：

- 运行单个容器的 `Pod` 。
  "每个 Pod 一个容器"模型是最常见的 Kubernetes 用例； 在这种情况下，可以将 Pod 看作单个容器的包装器，并且 Kubernetes 直接管理 Pod，而不是容器。

- 运行多个协同工作的容器的 `Pod` 。
  `Pod` 可能封装由多个紧密耦合且需要共享资源的共处容器组成的应用程序。这些位于同一位置的容器可能形成单个内聚的服务单元。例如：一个容器将文件从共享卷提供给公众， 而另一个单独的“边车”（sidecar）容器则刷新或更新这些文件。 Pod 将这些容器和存储资源打包为一个可管理的实体。



每个`Pod`运行给定应用程序的单个实例。如果希望横向扩展应用程序（例如，运行多个实例 以提供更多的资源），则应该使用多个 Pod，每个实例使用一个 Pod。 在 Kubernetes 中，这通常被称为 副本（Replication）。 通常使用一种工作负载资源及其控制器 来创建和管理一组 Pod 副本。

使用 Pod

你很少在 Kubernetes 中直接创建一个个的 Pod，甚至是单实例（Singleton）的 Pod。 这是因为 Pod 被设计成了相对临时性的、用后即抛的一次性实体。 当 Pod 由你或者间接地由 控制器 创建时，它被调度在集群中的节点上运行。 Pod 会保持在该节点上运行，直到 Pod 结束执行、Pod 对象被删除、Pod 因资源不足而被 驱逐 或者节点失效为止。

Pod 和控制器

你可以使用工作负载资源来创建和管理多个 Pod。 资源的控制器能够处理副本的管理、上线，并在 Pod 失效时提供自愈能力。 例如，如果一个节点失败，控制器注意到该节点上的 Pod 已经停止工作，就可以创建替换性的 Pod。调度器会将替身 Pod 调度到一个健康的节点执行。

下面是一些管理一个或者多个 Pod 的工作负载资源的示例：

- Deployment

- StatefulSet
- DaemonSet

Pod 模版

负载资源的控制器通常使用 Pod 模板（Pod Template） 来替你创建 Pod 并管理它们。
Pod 模板是包含在工作负载对象中的规范，用来创建 `Pod` 。这类负载资源包括 `Deployment` 、 `Job` 和 `DaemonSets` 等。
工作负载的控制器会使用负载对象中的 PodTemplate 来生成实际的 Pod。 PodTemplate 是你用来运行应用时指定的负载资源的目标状态的一部分。

```yaml
apiVersion: app/v1
kind: Deployment
metadata:
  name: hello
spec:
  template:
    # 这里是 Pod 模版
    spec:
      replicas: 1
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
    # 以上为 Pod 模版
```

修改 Pod 模版或者切换到新的 Pod 模版都不会对已经存在的 Pod 起作用。 Pod 不会直接收到模版的更新。相反， 新的 Pod 会被创建出来，与更改后的 Pod 模版匹配。

例如，Deployment 控制器针对每个 Deployment 对象确保运行中的 Pod 与当前的 Pod 模版匹配。如果模版被更新，则 Deployment 必须删除现有的 Pod，基于更新后的模版 创建新的 Pod。每个工作负载资源都实现了自己的规则，用来处理对 Pod 模版的更新。

容器探针

由 kubelet 对容器执行的定期诊断。要执行诊断，kubelet 可以执行三种动作：

1. `ExecAction` （借助容器运行时执行）
2. `TCPSocketAction` （由 kubelet 直接检测）
3. `HTTPGetAction` （由 kubelet 直接检测）

Kubernetes Storage

1. Volumes / 卷

2. Persistent Volumes / 持久卷
3. Projected Volumes / 投射卷
4. Ephemeral Volumes / 临时卷
5. Storage Classes / 存储类
6. Dynamic Volume Provisioning / 动态卷供应
7. Volume Snapshots / 卷快照
8. Volume Snapshot Classes / 卷快照类
9. CSI Volume Cloning / CSI 卷克隆
10. Storage Capacity / 存储容量
11. Node-specific Volume Limits / 特定于节点的卷数限制
12. Volume Health Monitoring / 卷健康监测

Volumes / 卷

Container 中的文件在磁盘上是临时存放的，这给 Container 中运行的较重要的应用程序带来一些问题。 问题之一是当容器崩溃时文件丢失。 kubelet 会重新启动容器，但容器会以干净的状态重启。 第二个问题会在同一 Pod 中运行多个容器并共享文件时出现。 Kubernetes 卷（Volume） 这一抽象概念能够解决这两个问题。

Docker 也有 卷（Volume） 的概念，但对它只有少量且松散的管理。 Docker 卷是磁盘上或者另外一个容器内的一个目录。 Docker 提供卷驱动程序，但是其功能非常有限。

Kubernetes 支持很多类型的卷。 Pod 可以同时使用任意数目的卷类型。 临时卷类型的生命周期与 Pod 相同，但持久卷可以比 Pod 的存活期长。 当 Pod 不再存在时，Kubernetes 也会销毁临时卷；不过 Kubernetes 不会销毁持久卷。 对于给定 Pod 中任何类型的卷，在容器重启期间数据都不会丢失。

卷的核心是一个目录，其中可能存有数据，Pod 中的容器可以访问该目录中的数据。 所采用的特定的卷类型将决定该目录如何形成的、使用何种介质保存数据以及目录中存放的内容。

25种卷类型

- local
  `local` 卷所代表的是某个被挂载的本地存储设备，例如磁盘、分区或者目录。
  `local` 卷只能用作静态创建的持久卷。尚不支持动态配置。
  与 `hostPath` 卷相比， `local` 卷能够以持久和可移植的方式使用，而无需手动将 `Pod` 调度到节点。系统通过查看 `PersistentVolume` 的节点亲和性配置，就能了解卷的节点约束。
  然而，local 卷仍然取决于底层节点的可用性，并不适合所有应用程序。 如果节点变得不健康，那么 local 卷也将变得不可被 Pod 访问。使用它的 Pod 将不能运行。 使用 local 卷的应用程序必须能够容忍这种可用性的降低，以及因底层磁盘的耐用性特征而带来的潜在的数据丢失风险。
  下面是一个使用 `local` 卷和 `nodeAffinity` 的持久卷示例：

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

- hostPath

> 警告：
> HostPath 卷存在许多安全风险，最佳做法是尽可能避免使用 HostPath。 当必须使用 HostPath 卷时，它的范围应仅限于所需的文件或目录，并以只读方式挂载。
> 如果通过 AdmissionPolicy 限制 HostPath 对特定目录的访问，则必须要求 volumeMounts 使用 readOnly 挂载以使策略生效。

`hostPath` 卷能将主机节点文件系统上的文件或目录挂载到你的 `Pod` 中。虽然这不是大多数 `Pod` 需要的，但是它为一些应用程序提供了强大的逃生舱。
例如， `hostPath` 的一些用法有：

- 运行一个需要访问 Docker 内部机制的容器；可使用 hostPath 挂载 /var/lib/docker 路径。
- 在容器中运行 `cAdvisor` 时，以 `hostPath` 方式挂载 /sys。
- 允许 `Pod` 指定给定的 `hostPath` 在运行 Pod 之前是否应该存在，是否应该创建以及应该以什么方式存在。

支持的 type 值如下：

取值	行为
	空字符串（默认）用于向后兼容，这意味着在安装 hostPath 卷之前不会执行任何检查。

取值	行为
DirectoryOrCreate	如果在给定路径上什么都不存在，那么将根据需要创建空目录， 权限设置为 0755，具有与 kubelet 相同的组和属主信息。
Directory	在给定路径上必须存在的目录。
FileOrCreate	如果在给定路径上什么都不存在，那么将在那里根据需要创建空文件， 权限设置为 0644，具有与 kubelet 相同的组和所有权。
File	在给定路径上必须存在的文件。
Socket	在给定路径上必须存在的 UNIX 套接字。
CharDevice	在给定路径上必须存在的字符设备。
BlockDevice	在给定路径上必须存在的块设备。

当使用这种类型的卷时要小心，因为：

- HostPath 卷可能会暴露特权系统凭据（例如 Kubelet）或特权API(例如容器运行时套接字)，可用于容器逃逸或攻击集群的其他部分。
- 具有相同配置（例如基于同一 PodTemplate 创建）的多个 Pod 会由于节点上文件的不同而在不同节点上有不同的行为。
- 下层主机上创建的文件或目录只能由 root 用户写入。你需要在 特权容器 中以 root 身份运行进程，或者修改主机上的文件权限以便容器能够写入 hostPath 卷。

`hostPath` 配置示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # 宿主上目录位置
      path: /data
      # 此字段为可选
      type: Directory
```

- nfs

  nfs 卷能将 NFS (网络文件系统) 挂载到你的 Pod 中。 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，nfs 卷的内容在删除 Pod 时会被保存，卷只是被卸载。 这意味着 nfs 卷可以被预先填充数据，并且这些数据可以在 Pod 之间共享。

```
# 安装NFS和RPC
$ yum -y install nfs-utils rpcbind
# 配置nfs目录
# 新建data目录
$ mkdir /data
# 在文件/etc/exports添加如下内容
$ echo '/data *(rw,no_root_squash,sync)' >> /etc/exports
# 执行命令使配置生效
$ exportfs -r
# 配置完成后，启动 NFS 服务器
$ systemctl enable nfs-server && systemctl start nfs-server
# 验证
$ showmount -e
Export list for instance-cesz58w0:
/data *
```

- secret

  secret 卷用来给 Pod 传递敏感信息，例如密码, SSL证书，RSA秘钥对等

- configMap

  configMap 卷提供了向 Pod 注入配置数据的方法。 ConfigMap 对象中存储的数据可以被 configMap 类型的卷引用，然后被 Pod 中运行的容器化应用使用。

  引用 configMap 对象时，你可以在 volume 中通过它的名称来引用。 你可以自定义 ConfigMap 中特定条目所要使用的路径。 下面的配置显示了如何将名为 log-config 的 ConfigMap 挂载到名为 configmap-pod 的 Pod 中：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
  - name: test
    image: busybox
    volumeMounts:
      - name: config-vol
        mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level
```

- persistentVolumeClaim
  persistentVolumeClaim 卷用来将持久卷（PersistentVolume）挂载到 Pod 中。 持久卷申领（PersistentVolumeClaim）是用户在不知道特定云环境细节的情况下“申领”持久存储（例如 GCE PersistentDisk 或者 iSCSI 卷）的一种方法。
- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- cinder
- downwardAPI
- emptyDir
- fc (光纤通道)
- flocker （已弃用）
- gcePersistentDisk
- gitRepo (已弃用)
- glusterfs
- iscsi
- portworxVolume
- projected （投射）
- quobyte (已弃用)
- rbd
- storageOS (已弃用)
- vsphereVolume

Persistent Volumes / 持久卷

持久卷（PersistentVolume，PV）是集群中的一块存储，可以由管理员事先供应，或者 使用存储类（Storage Class）来动态供应。 持久卷是集群资源，就像节点也是集群资源一样。PV 持久卷和普通的 Volume 一样，也是使用 卷插件来实现的，只是它们拥有独立于任何使用 PV 的 Pod 的生命周期。 此 API 对象中记述了存储的实现细节，无论其背后是 NFS、iSCSI 还是特定于云平台的存储系统。

持久卷申领（PersistentVolumeClaim，PVC）表达的是用户对存储的请求。概念上与 Pod 类似。 Pod 会耗用节点资源，而 PVC 申领会耗用 PV 资源。Pod 可以请求特定数量的资源（CPU 和内存）；同样 PVC 申领也可以请求特定的大小和访问模式 （例如，可以要求 PV 卷能够以 ReadWriteOnce、ReadOnlyMany 或 ReadWriteMany 模式之一来挂载，参见访问模式）。

尽管 PersistentVolumeClaim 允许用户消耗抽象的存储资源，常见的情况是针对不同的 问题用户需要的是具有不同属性（如，性能）的 PersistentVolume 卷。 集群管理员需要能够提供不同性质的 PersistentVolume，并且这些 PV 卷之间的差别不 仅限于卷大小和访问模式，同时又不能将卷是如何实现的这些细节暴露给用户。 为了满足这类需求，就有了 存储类（StorageClass） 资源。

卷和申领的生命周期

- 供应 (PV 卷的供应有两种方式：静态供应或动态供应)

  > 静态供应:
  > 集群管理员创建若干 PV 卷。这些卷对象带有真实存储的细节信息，并且对集群 用户可用（可见）。PV 卷对象存在于 Kubernetes API 中，可供用户消费（使用）。
  > 动态供应:
  > 如果管理员所创建的所有静态 PV 卷都无法与用户的 PersistentVolumeClaim 匹配， 集群可以尝试为该 PVC 申领动态供应一个存储卷。 这一供应操作是基于 StorageClass 来实现的：PVC 申领必须请求某个 存储类，同时集群管理员必须 已经创建并配置了该类，这样动态供应卷的动作才会发生。 如果 PVC 申领指定存储类为 ""，则相当于为自身禁止使用动态供应的卷。
  > 为了基于存储类完成动态的存储供应，集群管理员需要在 API 服务器上启用 DefaultStorageClass 准入控制器。 举例而言，可以通过保证 DefaultStorageClass 出现在 API 服务器组件的 --enable-admission-plugins 标志值中实现这点；该标志的值可以是逗号 分隔的有序列表。关于 API 服务器标志的更多信息，可以参考 kube-apiserver 文档。

- 绑定
  用户创建一个带有特定存储容量和特定访问模式需求的 PersistentVolumeClaim 对象； 在动态供应场景下，这个 PVC 对象可能已经创建完毕。 主控节点中的控制回路监测新的 PVC 对象，寻找与之匹配的 PV 卷（如果可能的话）， 并将二者绑定到一起。 如果为了新的 PVC 申领动态供应了 PV 卷，则控制回路总是将该 PV 卷绑定到这一 PVC 申领。 否则，用户总是能够获得他们所请求的资源，只是所获得的 PV 卷可能会超出所请求的配置。 一旦绑定关系建立，则 PersistentVolumeClaim 绑定就是排他性的，无论该 PVC 申领是 如何与 PV 卷建立的绑定关系。

PVC 申领与 PV 卷之间的绑定是一种一对一的映射，实现上使用 ClaimRef 来记述 PV 卷 与 PVC 申领间的双向绑定关系。

如果找不到匹配的 PV 卷，PVC 申领会无限期地处于未绑定状态。 当与之匹配的 PV 卷可用时，PVC 申领会被绑定。 例如，即使某集群上供应了很多 50 Gi 大小的 PV 卷，也无法与请求 100 Gi 大小的存储的 PVC 匹配。当新的 100 Gi PV 卷被加入到集群时，该 PVC 才有可能被绑定。

- 使用

  Pod 将 PVC 申领当做存储卷来使用。集群会检视 PVC 申领，找到所绑定的卷，并 为 Pod 挂载该卷。对于支持多种访问模式的卷，用户要在 Pod 中以卷的形式使用申领 时指定期望的访问模式。

  一旦用户有了申领对象并且该申领已经被绑定，则所绑定的 PV 卷在用户仍然需要它期间 一直属于该用户。用户通过在 Pod 的 volumes 块中包含 persistentVolumeClaim 节区来调度 Pod，访问所申领的 PV 卷。 相关细节可参阅使用申领作为卷。

- 保护使用中的存储对象

  保护使用中的存储对象（Storage Object in Use Protection）这一功能特性的目的 是确保仍被 Pod 使用的 PersistentVolumeClaim（PVC）对象及其所绑定的 PersistentVolume（PV）对象在系统中不会被删除，因为这样做可能会引起数据丢失。

  如果用户删除被某 Pod 使用的 PVC 对象，该 PVC 申领不会被立即移除。 PVC 对象的移除会被推迟，直至其不再被任何 Pod 使用。 此外，如果管理员删除已绑定到某 PVC 申领的 PV 卷，该 PV 卷也不会被立即移除。 PV 对象的移除也要推迟到该 PV 不再绑定到 PVC。

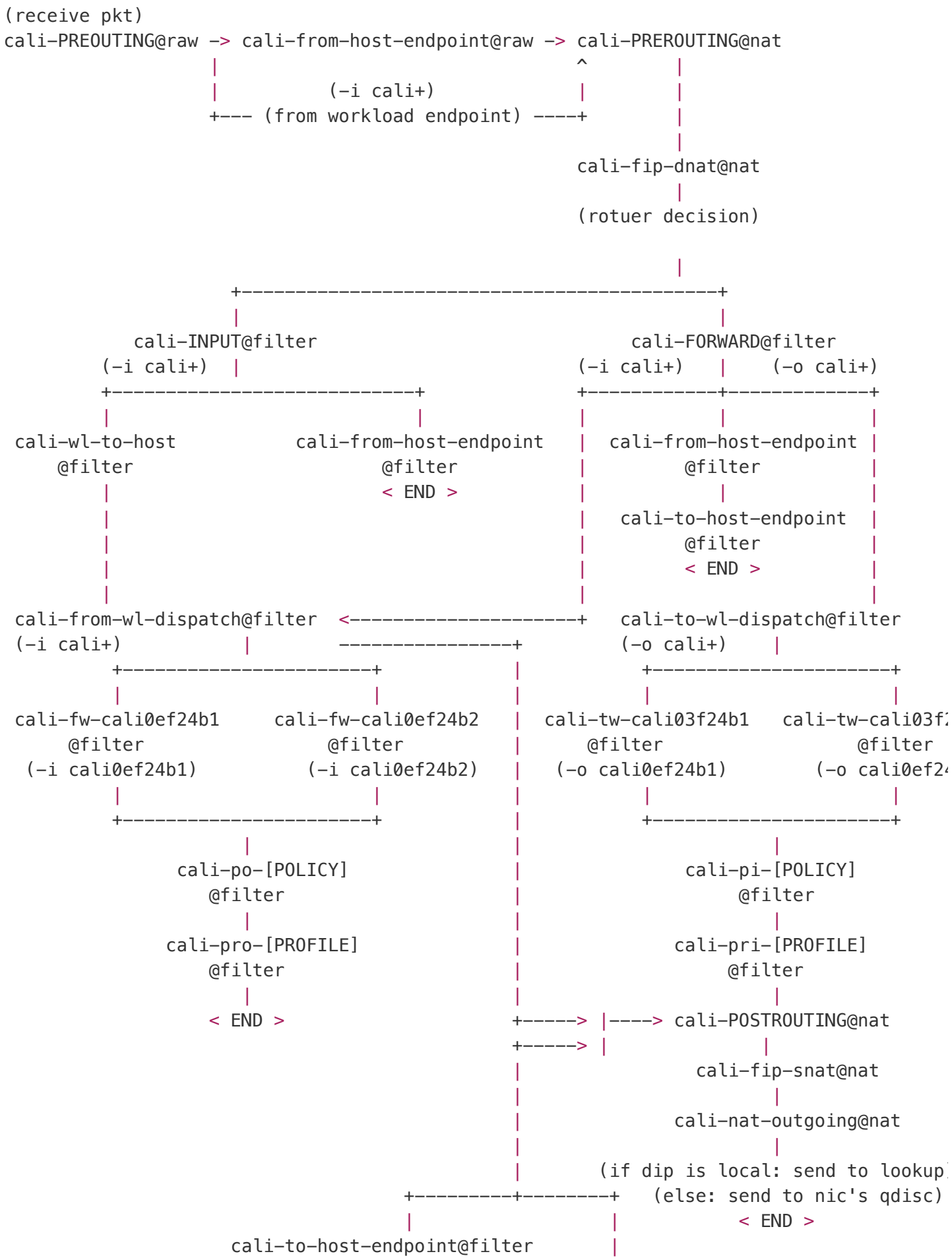
```
$ kubectl describe pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:        <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-hostpath
               volume.beta.kubernetes.io/storage-provisioner=example.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...

$ kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          type=local
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    standard
Status:          Terminating
Claim:
Reclaim Policy:  Delete
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
  Type:          HostPath (bare host directory volume)
  Path:          /tmp/data
  HostPathType:
Events:          <none>
```

- 回收
  当用户不再使用其存储卷时，他们可以从 API 中将 PVC 对象删除，从而允许 该资源被回收再利用。PersistentVolume 对象的回收策略告诉集群，当其被 从申领中释放时如何处理该数据卷。 目前，数据卷可以被 Retained（保留）、Recycled（回收）或 Deleted（删除）。
- 保留（Retain）
  回收策略 Retain 使得用户可以手动回收资源。当 PersistentVolumeClaim 对象 被删除时，PersistentVolume 卷仍然存在，对应的数据卷被视为"已释放（released）"。 由于卷上仍然存在这前一申领人的数据，该卷还不能用于其他申领。 管理员可以通过下面的步骤来手动回收该卷：
  1. 删除 PersistentVolume 对象。与之相关的、位于外部 2. 基础设施中的存储资产 （例如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）在 PV 删除之后仍然存在。
  2. 根据情况，手动清除所关联的存储资产上的数据。
  3. 手动删除所关联的存储资产。

  如果你希望重用该存储资产，可以基于存储资产的定义创建新的 PersistentVolume 卷对象。

Kubernetes calico iptables

Calico 创建和管理一个扁平的三层网络（不需要 overlay），每个容器会分配一个可路由的 IP。由于通信时不需要解包和封包，网络性能损耗小，易于排查，且易于水平扩展。
小规模部署时可以通过 BGP client 直接互联，大规模下可通过指定的 BGP Route Reflector 来完成，这样保证所有的数据流量都是通过 IP 路由的方式完成互联的。
Calico 基于 iptables 还提供了丰富而灵活的网络 Policy，保证通过各个节点上的 ACL 来提供 Workload 的多租户隔离、安全组以及其他可达性限制等功能。


(receive pkt)
cali-PREOUTING@raw -> cali-from-host-endpoint@raw -> cali-PREROUTING@nat
(-i cali+)
(from workload endpoint)
cali-fip-dnat@nat
(rotuer decision)
cali-INPUT@filter
(-i cali+)
cali-FORWARD@filter
(-i cali+)
(-o cali+)
cali-wl-to-host
@filter
cali-from-host-endpoint
@filter
< END >
cali-from-host-endpoint
@filter
cali-to-host-endpoint
@filter
< END >
cali-from-wl-dispatch@filter
(-i cali+)
cali-to-wl-dispatch@filter
(-o cali+)
cali-fw-cali0ef24b1
@filter
(-i cali0ef24b1)
cali-fw-cali0ef24b2
@filter
(-i cali0ef24b2)
cali-tw-cali03f24b1
@filter
(-o cali0ef24b1)
cali-po-[POLICY]
@filter
cali-pro-[PROFILE]
@filter
< END >
cali-pi-[POLICY]
@filter
cali-pri-[PROFILE]
@filter
cali-POSTROUTING@nat
cali-fip-snat@nat
cali-nat-outgoing@nat
(if dip is local: send to lookup
(else: send to nic's qdisc)
< END >
cali-to-host-endpoint@filter

```
                                        |                  |
                                        +------------------+
                                                  ^ (-o cali+)
                                                  |
                                          cali-OUTPUT@filter
                                                  ^
(send pkt)                                        |
(router descition) -> cali-OUTPUT@nat -> cali-fip-dnat@natopi
```

Kubernetes Deployments

Deployment 用来声明 `Pod` 和 `ReplicaSets`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

以下是 Deployments 的典型用例：

1. 创建 Deployment 以将 ReplicaSet 上线。 ReplicaSet 在后台创建 Pods。 检查 ReplicaSet 的上线状态，查看其是否成功。
2. 通过更新 Deployment 的 PodTemplateSpec，声明 Pod 的新状态 。 新的 ReplicaSet 会被创建，Deployment 以受 控速率将 Pod 从旧 ReplicaSet 迁移到新 ReplicaSet。 每个新的 ReplicaSet 都会更新 Deployment 的修订版本。
3. 如果 Deployment 的当前状态不稳定，回滚到较早的 Deployment 版本。 每次回滚都会更新 Deployment 的修订版本。
4. 扩大 Deployment 规模以承担更多负载。

5. 暂停 Deployment 以应用对 PodTemplateSpec 所作的多项修改， 然后恢复其执行以启动新的上线版本。
6. 使用 Deployment 状态 来判定上线过程是否出现停滞。
7. 清理较旧的不再需要的 ReplicaSet 。

```
# 查看deployments的状态
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

Kubernetes Service

Kubernetes Service 定义了这样一种抽象：逻辑上的一组 Pod，一种可以访问它们的策略 —— 通常称为微服务。 Service 所针对的 Pods 集合通常是通过选择算符来确定的。

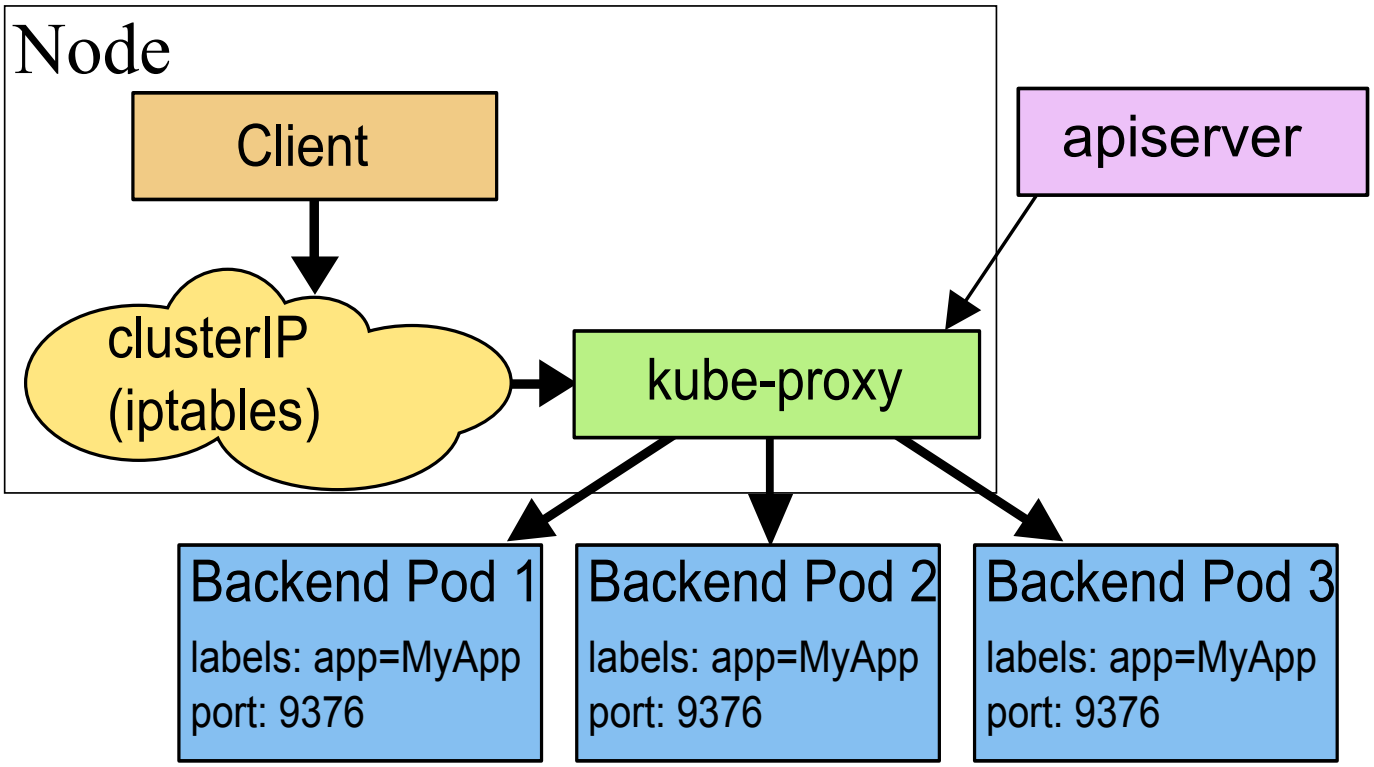
举个例子，考虑一个图片处理后端，它运行了 3 个副本。这些副本是可互换的 —— 前端不需要关心它们调用了哪个后端副本。 然而组成这一组后端程序的 Pod 实际上可能会发生变化， 前端客户端不应该也没必要知道，而且也不需要跟踪这一组后端的状态。

Service 在 Kubernetes 中是一个 REST 对象，和 Pod 类似。 像所有的 REST 对象一样，Service 定义可以基于 POST 方式，请求 API server 创建新的实例。 Service 对象的名称必须是合法的 RFC 1035 标签名称.。

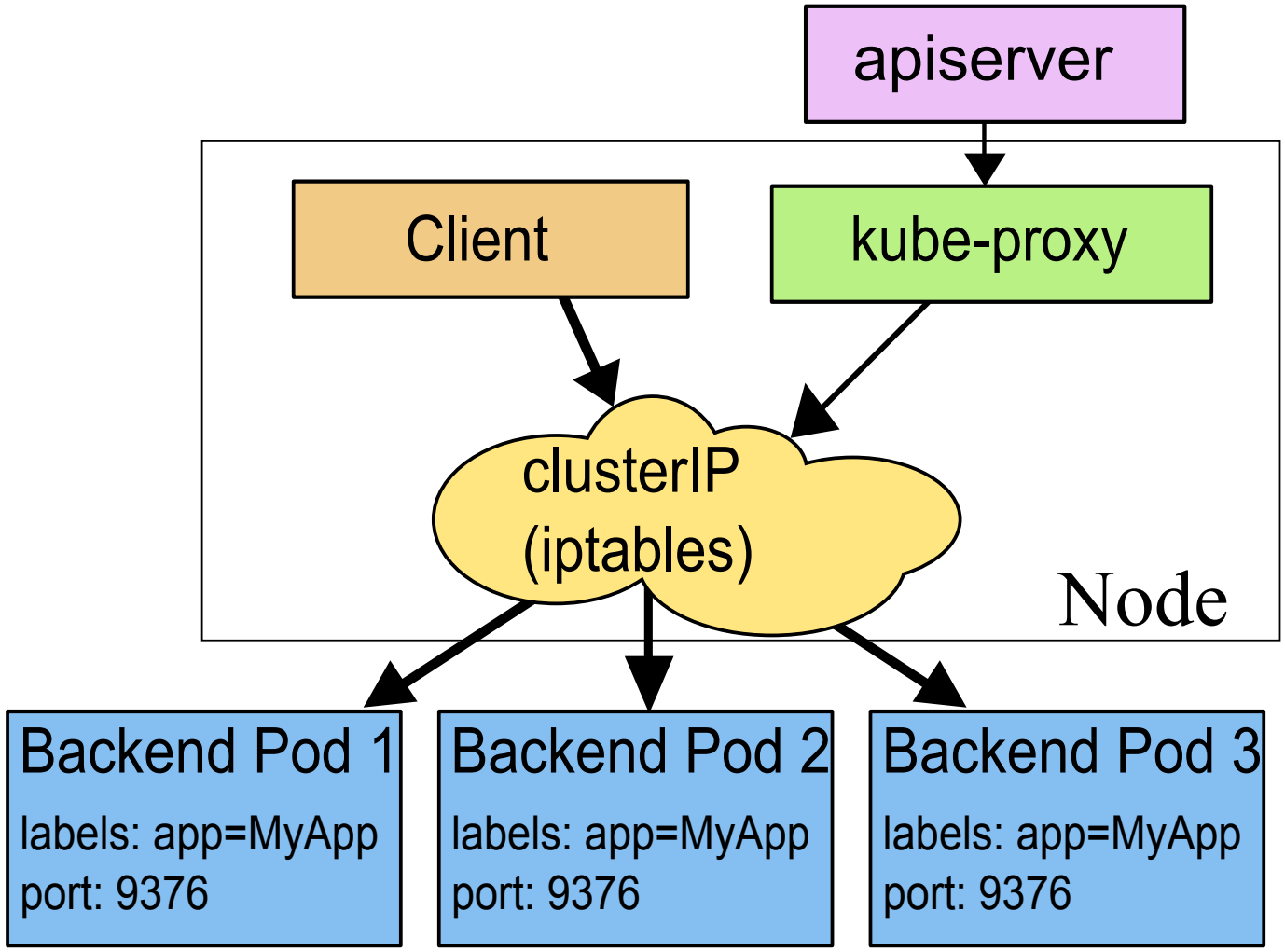
```
# 为一组Pod，它们对外暴露了 9376 端口，同时还被打上 app=MyApp 标签创建了一个名叫my-service的Server
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

虚拟 IP 和 Service 代理

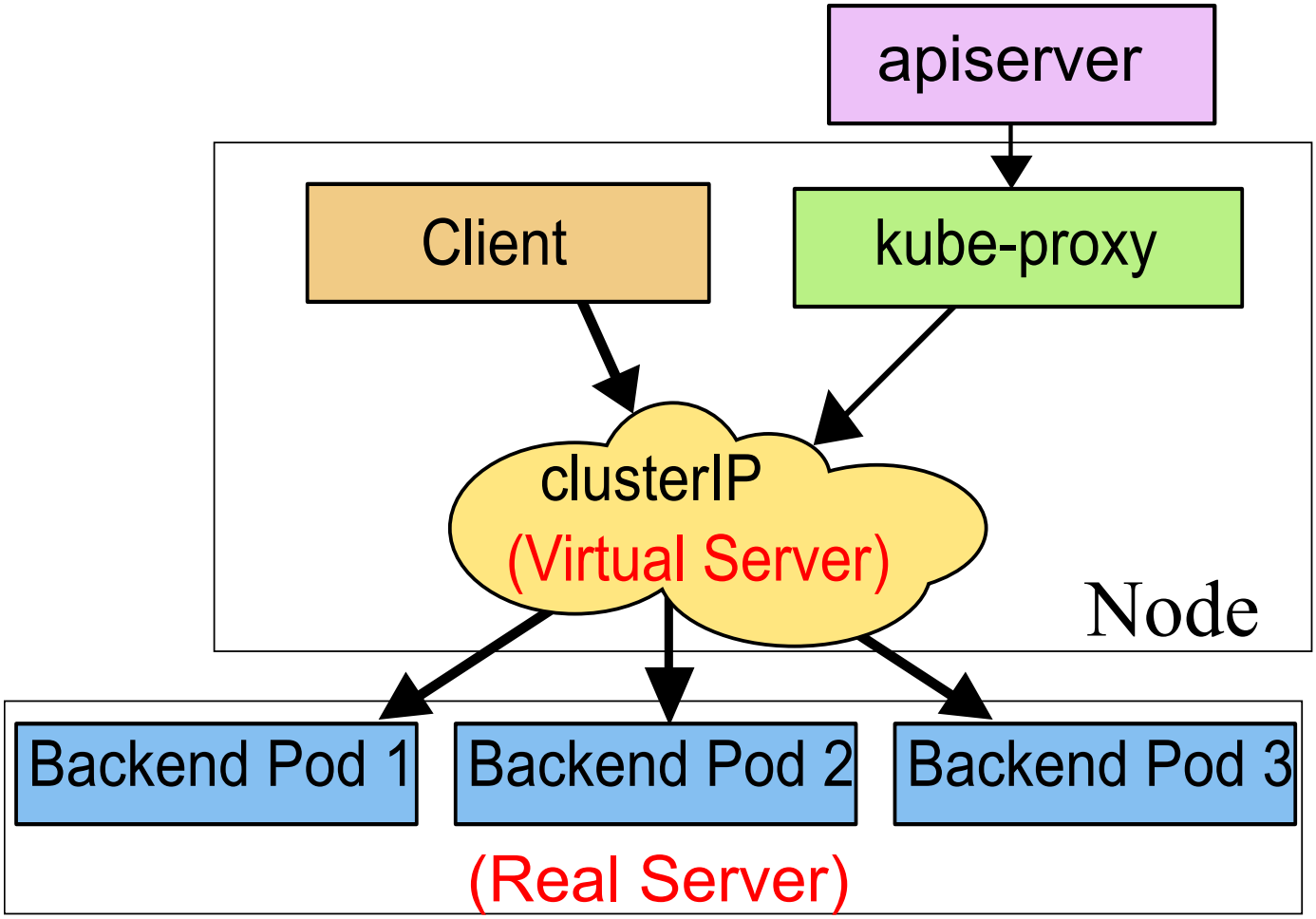
1. userspace 代理模式



2. iptables 代理模式



3. IPVS 代理模式

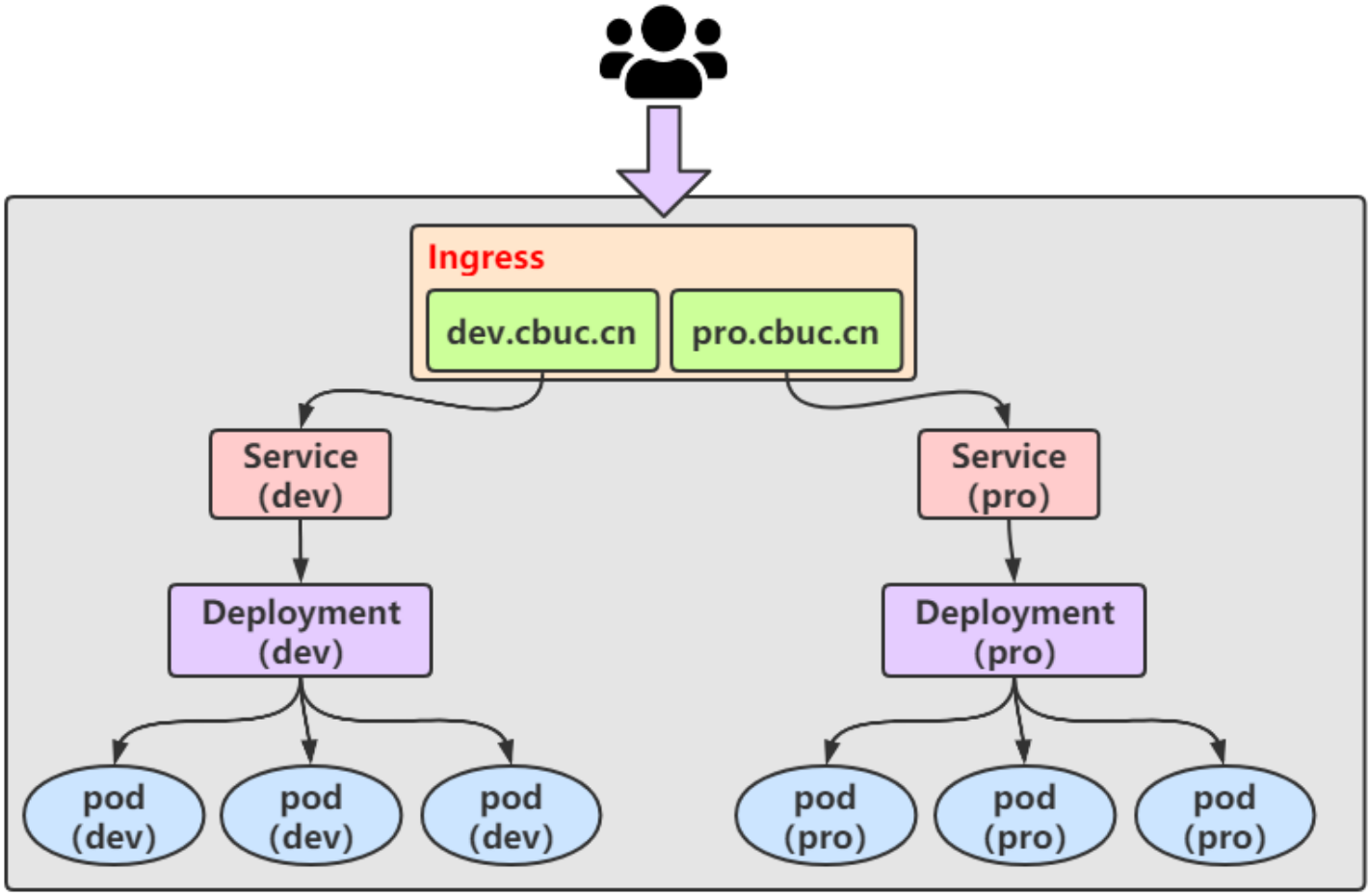


Kubernetes Ingress

Ingress 是对集群中服务的外部访问进行管理的 API 对象，典型的访问方式是 HTTP，Ingress 可以提供负载均衡、SSL 终结和基于名称的虚拟托管，Ingress 类似于 nginx。

Ingress 可为 Service 提供外部可访问的 URL、负载均衡流量、终止 SSL/TLS，以及基于名称的虚拟托管。 Ingress 控制器 通常负责通过负载均衡器来实现 Ingress，尽管它也可以配置边缘路由器或其他前端来帮助处理流量。

Ingress 不会公开任意端口或协议。 将 HTTP 和 HTTPS 以外的服务公开到 Internet 时，通常使用 Service.Type=NodePort 或 Service.Type=LoadBalancer 类型的 Service。


Ingress
dev.cbuc.cn
pro.cbuc.cn
Service (dev)
Service (pro)
Deployment (dev)
Deployment (pro)
pod (dev)
pod (dev)
pod (dev)
pod (pro)
pod (pro)
pod (pro)

```yaml
# 常用的 Ingress 声明
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "foo.bar.com"
    http:
      paths:
      - pathType: Prefix
        path: "/bar"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: "*.foo.com"
    http:
      paths:
      - pathType: Prefix
        path: "/foo"
        backend:
          service:
            name: service2
            port:
              number: 80
```

Kubernetes in timework

Kubernetes 集群安装

- 环境准备

  ```
  # 安装 git client
  $ sudo yum install git -y
  # Clone kubeadm-ha 源码
  $ git clone https://github.com/open-hand/kubeadm-ha.git
  # 安装 Ansible
  $ cd kubeadm-ha && sudo ./install-ansible.sh
  ```

- 配置 ansible inventory 文件
  - 项目 example 文件夹下提供了 6 个 ansible inventory 示例文件，请按需求进行选择并修改。
  - 拷贝项目下的 `example/hosts.m-master.ip.ini` 文件至项目根目录下，命名为 `inventory.ini` 。

```
$ cp example/hosts.m-master.ip.ini inventory.ini
```

修改各服务器的 IP 地址、用户名、密码，并维护好各服务器与角色的关系。

```
[all]
172.16.0.45 ansible_port=22 ansible_user="root" ansible_ssh_pass=""
172.16.0.3 ansible_port=22 ansible_user="root" ansible_ssh_pass=""
[etcd]
172.16.0.45
[kube-master]
172.16.0.3
[kube-worker]
172.16.0.45
172.16.0.3

container_manager="docker"
```

- 部署集群

```
# 在项目根目录下执行
$ ansible-playbook -i inventory.ini 90-init-cluster.yml
```

- 查看等待 pod 的状态为 runnning

```
# 任意master节点下执行
$ kubectl get po --all-namespaces -w
```

- 如果部署失败，想要重置集群，执行

```
# 在项目根目录下执行
$ ansible-playbook -i inventory.ini 99-reset-cluster.yml
```

常用的kubectl命令，使用频率比较高

- 查看集群中的节点 node

```
$ kubectl get node -o wide
NAME          STATUS   ROLES                         AGE   VERSION    INTERNAL-IP
172.16.0.3    Ready    control-plane,master,worker   34d   v1.20.11   172.16.0.3
172.16.0.45   Ready    etcd,worker                   34d   v1.20.11   172.16.0.45
```

- 查看集群中的 namespace/ns

```
$ kubectl get namespace
NAME                   STATUS   AGE
choerodon              Active   34d
choerodon-test         Active   34d
default                Active   34d
ingress-controller     Active   34d
kube-node-lease        Active   34d
kube-public            Active   34d
kube-system            Active   34d
kubernetes-dashboard   Active   34d
prod-tmis              Active   34d
```

- 查看命名空间中的 `ingress/ing`

```
$ kubectl get ingress -n prod-tmis -o wide
NAME                                 CLASS    HOSTS                       ADDRESS
minio-c8d97                          <none>   minio-tmis.timework.cn      10.244.7
scrm-gateway-8c82c                   <none>   api-tmis.timework.cn        10.244.7
tmis-frontend-55cdf                  <none>   tmis-subapp.timework.cn     10.244.7
tmis-management-mobile-d9033         <none>   tmis.timework.cn            10.244.7
tmis-qiankun-main-front-104c0        <none>   tmis.timework.cn            10.244.7
tmis-saas-front-d37f7                <none>   tmis-subapp.timework.cn     10.244.7
websocket-service-node-0c75b-http    <none>   api-tmis.timework.cn        10.244.7
websocket-service-node-0c75b-socketio <none>  api-tmis.timework.cn        10.244.7
```

- 查看命名空间中的 `service/svc`

```
$ kubectl get service -n prod-tmis -o wide
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
gateway                       ClusterIP   10.244.94.128   <none>        8080/TCP
minio                         ClusterIP   10.244.110.78   <none>        9000/TCP
mysql                         NodePort    10.244.115.18   <none>        3306:30306/T
rabbitmq-ha-9074a             ClusterIP   10.244.125.77   <none>        15672/TCP,56
rabbitmq-ha-9074a-discovery   ClusterIP   None            <none>        15672/TCP,56
redis                         ClusterIP   10.244.117.18   <none>        6379/TCP
register-server               ClusterIP   10.244.69.111   <none>        8000/TCP
tmis-front                    ClusterIP   10.244.103.96   <none>        80/TCP
tmis-management-mobile        ClusterIP   10.244.72.15    <none>        80/TCP
tmis-qiankun-main-front       ClusterIP   10.244.81.135   <none>        80/TCP
tmis-saas-front               ClusterIP   10.244.121.2    <none>        80/TCP
websocket-service-node        ClusterIP   10.244.119.93   <none>        3000/TCP
```

- 查看命名空间中的 `deployment/deploy`

```
$ kubectl get deployment -n prod-tmis -o wide
NAME                              READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS
minio-c8d97                       1/1     1            1           34d   minio
mysql-1fddd                       1/1     1            1           34d   mysql-1fddd
redis-ddc49                       1/1     1            1           34d   redis-ddc49
scrm-admin-87755                  1/1     1            1           34d   scrm-admin-87
scrm-gateway-8c82c                1/1     1            1           34d   scrm-gateway-
scrm-import-7e9d8                 1/1     1            1           34d   scrm-import-7
scrm-platform-f1b01               1/1     1            1           34d   scrm-platform
scrm-register-02d61               1/1     1            1           34d   scrm-register
scrm-scheduler-d40a8              1/1     1            1           34d   scrm-schedule
tmis-file-51f54                   1/1     1            1           34d   tmis-file-51f
tmis-frontend-55cdf               1/1     1            1           34d   tmis-frontend
tmis-iam-3f278                    1/1     1            1           34d   tmis-iam-3f27
tmis-management-mobile-d9033      1/1     1            1           34d   tmis-manageme
tmis-message-ca8e1                1/1     1            1           34d   tmis-message-
tmis-oauth-e8380                  1/1     1            1           34d   tmis-oauth-e8
tmis-qiankun-main-front-104c0     1/1     1            1           34d   tmis-qiankun-
tmis-saas-front-d37f7             1/1     1            1           34d   tmis-saas-fro
tmis-service-3558f                1/1     1            1           34d   tmis-service-
websocket-service-node-0c75b      1/1     1            1           34d   websocket-ser
```

- 查看命名空间中的 pod/po

```
$ kubectl get pod -n prod-tmis -o wide
NAME                                          READY   STATUS    RESTARTS   AGE
minio-c8d97-c5d5b7854-p79b7                   1/1     Running   0          34d
mysql-1fddd-7f6896874f-lj8gk                  1/1     Running   0          34d
rabbitmq-ha-9074a-0                           1/1     Running   0          34d
rabbitmq-ha-9074a-1                           1/1     Running   0          34d
rabbitmq-ha-9074a-2                           1/1     Running   0          34d
redis-ddc49-7cf7c9c576-5wpvk                  1/1     Running   0          34d
scrm-admin-87755-9f594b775-94ljm              1/1     Running   0          34d
scrm-gateway-8c82c-5bb7699794-qbv92           1/1     Running   0          33d
scrm-import-7e9d8-7fc5b5d99c-6npt8            1/1     Running   0          34d
scrm-platform-f1b01-5fff9b779-jgd8v           1/1     Running   0          34d
scrm-register-02d61-f6c866bfd-c6nt9           1/1     Running   0          34d
scrm-scheduler-d40a8-7d5d7dc45c-s8tgv         1/1     Running   0          34d
tmis-file-51f54-75665fd896-kqs85              1/1     Running   0          34d
tmis-frontend-55cdf-8546db6869-jrbbf          1/1     Running   0          4d14h
tmis-iam-3f278-f46667b59-m2k9w                1/1     Running   0          34d
tmis-management-mobile-d9033-7f87667ff4-dzhc9 1/1     Running   0          5d13h
tmis-message-ca8e1-886f965bd-qlkcw            1/1     Running   0          34d
tmis-oauth-e8380-66cb6b485c-78xcv             1/1     Running   0          34d
tmis-qiankun-main-front-104c0-76757f5b88-gcp76 1/1    Running   0          3d6h
tmis-saas-front-d37f7-6588b74c75-rx4q4        1/1     Running   0          34d
tmis-service-3558f-74c6c95d98-8fvj8           1/1     Running   0          4d6h
websocket-service-node-0c75b-6767887999-vgfxh 1/1     Running   0          6d9h
```

- 查看 Pod 日志

```
$ kubectl logs -f --tail 100 register-server-66ddbbb8dd-wfg9d -n test
2022-03-31 11:46:23.766  INFO 8 --- [           main] s.c.a.AnnotationConfigApplicat
2022-03-31 11:46:24.999  INFO 8 --- [           main] f.a.AutowiredAnnotationBeanPos
2022-03-31 11:46:25.124  INFO 8 --- [           main] trationDelegate$BeanPostProces

:: Spring Boot ::       (v1.5.14.RELEASE)

2022-03-31 11:46:26.447  INFO 8 --- [           main] i.c.eureka.EurekaServerApplica
2022-03-31 11:46:26.643  INFO 8 --- [           main] ationConfigEmbeddedWebApplicat
2022-03-31 11:46:31.067  INFO 8 --- [           main] o.s.cloud.context.scope.Generi
```

- 配置SSL证书

```
# 全新配置
$ kubectl create secret tls 证书名称 --cert=pem格式的证书路径 --key=key格式的证书路径 -n r


# 在Ingress中添加
tls:
- hosts:
  - minio-tmis.timework.cn
  secretName: tmis.timework.cn


# 猪齿鱼部署
ingress:
  enabled: true
  annotations: {}
  path: /
  host: minio-tmis.timework.cn
  tls:
    enabled: true
    secretName: tmis.timework.cn
```

- 更新SSL证书

  更新证书的话，可以按照上面的步骤重新创建一个 `secret` 对象，并且更新Ingress或者更新在猪齿鱼上的配置即可，如果没有pem和key格式的证书文件，已知别的Ingress的证书已经修改好，则可以用下面方法

```
# 查看已更新好的证书，并记录tls.crt和tls.key部分
$ kubectl get secret scrm.timework.cn -n scrm-prod -o=yaml
# 修改需要更新的证书`secret`对象，并将tls.crt和tls.key部分进行替换即可
$ kubectl edit secret tmis.timework.cn -n prod-tmis
```

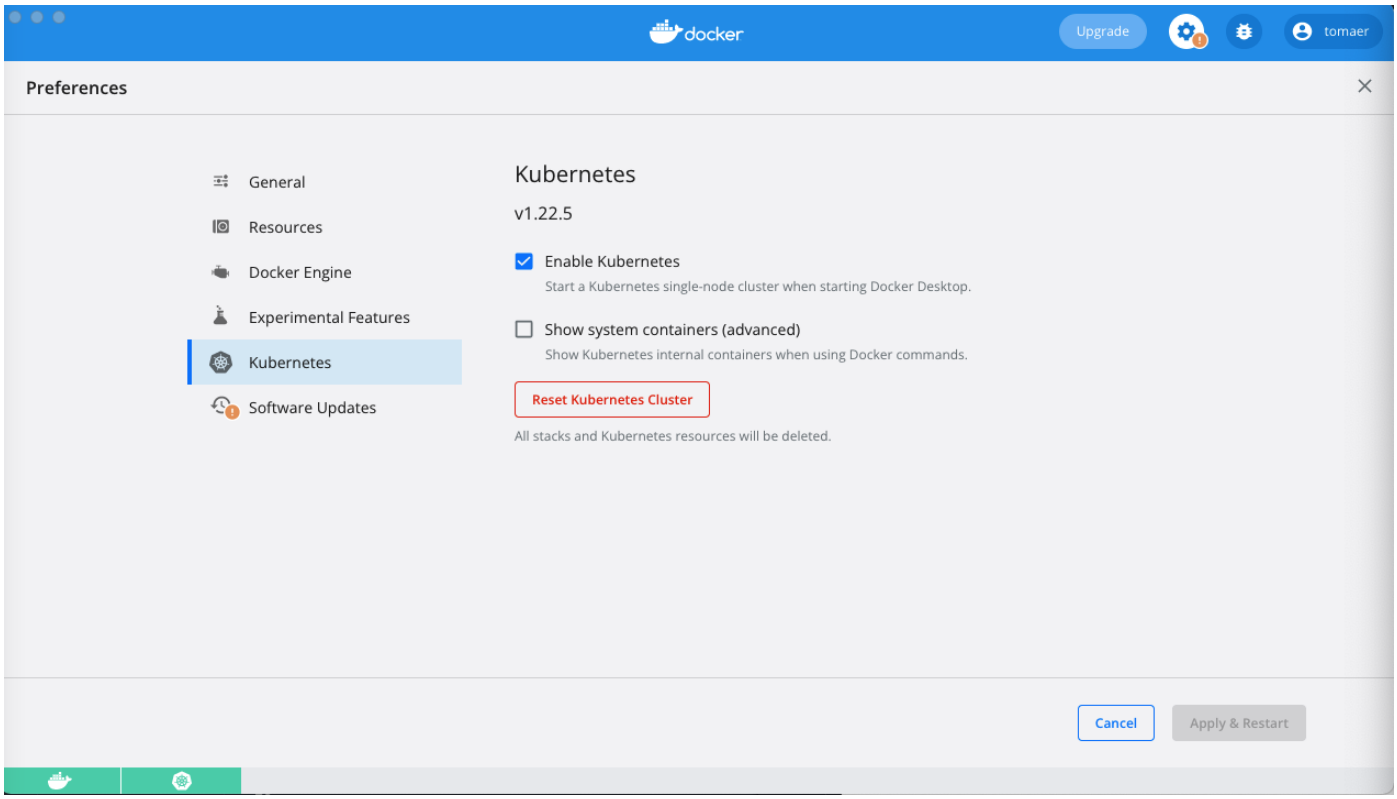
```yaml
# 替换完成的Secret对象
apiVersion: v1
kind: Secret
type: kubernetes.io/tls
metadata:
  name: tmis.timework.cn
  namespace: prod-tmis
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUZnVENDQkdtZ0F3SUJBZ0lRQ3RIVHFol
  tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcEFJQkFBS0NBUUVBNys5cjhSM
```

本地伪集群环境的安装

- 安装Docker Desktop
- 配置proxy

```
{
    "registry-mirrors": [
        "https://6n5qzb8y.mirror.aliyuncs.com"
    ],
    "features": {
        "buildkit": true
    },
    "experimental": false,
    "builder": {
        "gc": {
            "defaultKeepStorage": "20GB",
            "enabled": true
        }
    }
}
```

- 安装 Kubernetes



实战部分

- 在本地部署MySQL数据库
  - 创建 pv mysql_pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-data-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/Users/tomaer/.k8s_volume/mysql/data"


# 创建MySQL data目录并且创建pv
$ mkdir -p /Users/tomaer/.k8s_volume/mysql/data && kubectl apply -f mysql_pv.yml
```

  - 创建 pvc ， 并且与pv绑定 mysql_pvc.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-data-pv-claim
  namespace: test
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

```
# 创建pvc 并且查看是否绑定成功
$ kubectl apply -f mysql_pvc.yml && kubectl get pv
NAME                   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
mysql-data-pv-volume   10Gi       RWO            Retain           Bound    test/
```

- 创建 `deployment` mysql_deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: test
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7.18
        name: mysql
        args: ["--character-set-server=utf8mb4", "--collation-server=utf8mb4_ger
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: password
        - name: TZ
          value: Asia/Shanghai
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-data-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-data-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-data-pv-claim
```

```
$ kubectl apply -f mysql_deployment.yml
```

- 创建 `service` mysql_service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: test
spec:
  type: NodePort
  ports:
  - port: 3306
    targetPort: 3306
    nodePort: 30036
  selector:
    app: mysql


$ kubectl apply -f mysql_service.yml
```

- 在本地部署Redis
  - 创建 pv redis_pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: redis-data-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 3Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/Users/tomaer/.k8s_volume/redis/data"


# 创建Redis data目录并且创建pv
$ mkdir -p /Users/tomaer/.k8s_volume/redis/data && kubectl apply -f redis_pv.yml
```

  - 创建 pvc ,并且与pv绑定 redis_pvc.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: redis-data-pv-claim
  namespace: test
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

```
# 创建pvc 并且查看是否绑定成功
$ kubectl apply -f redis_pvc.yml && kubectl get pv
NAME                    CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
mysql-data-pv-volume    10Gi       RWO            Retain           Bound    test/
redis-data-pv-volume    3Gi        RWO            Retain           Bound    test/
```

- 创建 `deployment` redis_deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  namespace: test
spec:
  selector:
    matchLabels:
      app: redis
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - image: redis
          name: redis
          args: ["--protected-mode", "no", "--appendonly", "yes", "--dir", "/dat
          env:
          - name: TZ
            value: Asia/Shanghai
          ports:
          - containerPort: 6379
            name: redis
          volumeMounts:
          - name: redis-data-persistent-storage
            mountPath: /data
      volumes:
        - name: redis-data-persistent-storage
          persistentVolumeClaim:
            claimName: redis-data-pv-claim


$ kubectl apply -f redis_deployment.yml
```

- 创建 `Service` , redis_service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  namespace: test
spec:
  type: NodePort
  ports:
  - port: 6379
    targetPort: 6379
    nodePort: 30079
  selector:
    app: redis


$ kubectl apply -f redis_service.yml
```

- 在服务器上部署注册中心
  - 创建deployment register_deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: register-server
  namespace: test
spec:
  selector:
    matchLabels:
      app: register-server
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: register-server
    spec:
      containers:
        - image: registry.gobuildrun.com/operation-br-app-engine/eureka-server:2
          name: register-server
          env:
          - name: TZ
            value: Asia/Shanghai
          ports:
          - containerPort: 8000
            name: register-server


$ kubectl apply -f register_deployment.yml
```

- 创建service register_service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: register-server
  namespace: test
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 8000
    targetPort: 8000
    protocol: TCP
  selector:
    app: register-server


$ kubectl apply -f register_service.yml
```

- 创建ingress register_ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
  namespace: test
spec:
  rules:
  - host: register-tmis.timework.com
    http:
      paths:
      - pathType: ImplementationSpecific
        path: /
        backend:
          service:
            name: register-server
            port:
              number: 8000


$ kubectl apply -f register_ingress.yml
```

附录: Kubectl命令

```
Usage:
  kubectl [flags] [options]
```

```
Basic Commands (Beginner):
  create        Create a resource from a file or from stdin
  expose        Take a replication controller, service, deployment or pod and expose it
  run           Run a particular image on the cluster
  set           Set specific features on objects

Basic Commands (Intermediate):
  explain       Get documentation for a resource
  get           Display one or many resources
  edit          Edit a resource on the server
  delete        Delete resources by file names, stdin, resources and names, or by resou

Deploy Commands:
  rollout       Manage the rollout of a resource
  scale         Set a new size for a deployment, replica set, or replication controller
  autoscale     Auto-scale a deployment, replica set, stateful set, or replication cont

Cluster Management Commands:
  certificate   Modify certificate resources.
  cluster-info  Display cluster information
  top           Display resource (CPU/memory) usage
  cordon        Mark node as unschedulable
  uncordon      Mark node as schedulable
  drain         Drain node in preparation for maintenance
  taint         Update the taints on one or more nodes

Troubleshooting and Debugging Commands:
  describe      Show details of a specific resource or group of resources
  logs          Print the logs for a container in a pod
  attach        Attach to a running container
  exec          Execute a command in a container
  port-forward  Forward one or more local ports to a pod
  proxy         Run a proxy to the Kubernetes API server
  cp            Copy files and directories to and from containers
  auth          Inspect authorization
  debug         Create debugging sessions for troubleshooting workloads and nodes

Advanced Commands:
  diff          Diff the live version against a would-be applied version
  apply         Apply a configuration to a resource by file name or stdin
  patch         Update fields of a resource
  replace       Replace a resource by file name or stdin
  wait          Experimental: Wait for a specific condition on one or many resources
  kustomize     Build a kustomization target from a directory or URL.

Settings Commands:
  label         Update the labels on a resource
  annotate      Update the annotations on a resource
  completion    Output shell completion code for the specified shell (bash or zsh)
```

```
Other Commands:
  api-resources Print the supported API resources on the server
  api-versions  Print the supported API versions on the server, in the form of "group/ve
  config        Modify kubeconfig files
  plugin        Provides utilities for interacting with plugins
  version       Print the client and server version information
```