模块11 将应用迁移至Kubernetes平台

应用接入最佳实践

应用容器化

- 对于Dockerfile要考虑以下的内容
  - 用什么基础镜像——基础镜像越小越好么？如果需要调试就必须登录主机或者在主容器旁边加一个DEBUG容器，如果可以做到这一点，那基础镜像就是越小越好。
  - 需要装什么Utility——lib越少越安全，但是越不方便
  - 容器内需要多少个进程?
    - 主次要分清。如果监控进程不正常了，那整个容器的状态应该如何决策是场景相关的。
    - 防止Fork bomb，K8S有参数配置每个容器里可以占用多少PID
  - 代码和配置分离后配置如何管理
    - 环境变量
    - 配置文件通过volume mount放入容器内
  - 分层的控制。层数越多，效率越低。多个命令叠加到一层；变化慢的层要在下面等。
  - Entrypoint。直接写命令或者写在中括号里是有一些差别的。
- 对于应用本身，有如下点需要考虑
  - 启动速度。虽然容器相较于虚拟机，启动速很快，但有时应用本身启动就很慢，比如很多的JAVA程序，这时就应该考虑加速应用的启动速度，比如做拆分，做微服务化，让应用足够小，使其适合云原生
  - 如何做健康检查，只做端口探活是不够的
  - 启动参数，当这些参数是和主机相关时（比如CPU数量），就会不准确（参见下面的资源监控）
  - 如何做各种探针（liveness\readiness）检查
  - 容器日志驱动Log Driver。云原生容器日志输入到`stdout`和`stderr`，这样就可以使用`kubectl logs`看到日志，然后日志通过runtime的log driver转储到节点上，由日志采集系统进行处理
    - Blocking mode，标准输入buffer满了就会阻塞，影响应用运行速度
    - Nonblocking mode，如果日志狂打，来不及转储就会被丢弃
  - 共用kernel带来的问题
    - 系统参数配置共享，比如elastic search需要特定的参数，这样其它应用也被强迫使用这样的参数
    - 进程数共享（Fork bomb）
    - fd数共享，主机fd被耗光
    - 主机磁盘共享（导致IO下降），虚拟机也存在这个问题
  - 资源监控

    下面的项目对JAVA和Node.js程序都有影响
    - 容器中看到的是主机的资源，比如
      - top
      - java runtime.GetAvailableProcessors()。**影响GC线程的数量。**
      - cat /proc/cpuinfo

    - cat /proc/meminfo。**影响堆内存的分配。**
    - df -k
- 解决方案
    - **在容器内**查询/proc/1/cgroup是否包含kubepods关键字（docker关键字不可靠），如果包含此关键字，说明是运行在Kubernetes之上

```
[root@centos-qos-59db49b6cd-cnz2k /]# cat /proc/1/cgroup
12:blkio:/kubepods.slice/kubepods-podfab1a951_750c_48c1_96a9_b82f8f7da7e0.slice/cri-coope
11:pids:/kubepods.slice/kubepods-podfab1a951_750c_48c1_96a9_b82f8f7da7e0.slice/cri-conpe
10:cpuset:/kubepods.slice/kubepods-podfab1a951_750c_48c1_96a9_b82f8f7da7e0.slice/cri-ccope
```

    - 对于CPU来说，**在容器内**查询

```
cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
```

      配额，分配的 CPU 个数 = quota / period，quota = -1代表 besteffort
        - cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
            - -1
        - cat /sys/fs/cgroup/cpu/cpu.cfs_period_us
            - 100000

      用量
        - cat /sys/fs/cgroup/cpuacct/cpuacct.usage_percpu（按CPU区分）
            - 140669504971 148500278385 149957919463 152786448674
        - cat /sys/fs/cgroup/cpuacct/cpuacct.usage
            - 12081100465458
    - 对内存来说，**在容器内**查询

      配额
        - cat /sys/fs/cgroup/memory/memory.limit_in_bytes
        - 36854771712

      用量
        - cat /sys/fs/cgroup/memory/memory.usage_in_bytes
        - 448450560
    - 或者使用downward-api。从POD的spec中获取分配的值并作为环境变量传给容器

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: centos-qos
spec:
  replicas: 1
  selector:
    matchLabels:
      app: centos
  template:
    metadata:
      labels:
        app: centos
    spec:
```

```
    containers:
    - command:
      - tail
      - -f
      - /dev/null
      image: centos
      name: centos
      resources:
        requests:
          cpu: 250m
          memory: 1Gi
        limits:
          cpu: 250m
          memory: 1Gi
      env:                # 使用downward-api
        - name: SYSTEM_NAMESPACE_ENV
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: PODIP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: centos
              resource: limits.cpu
              divisor: 1m
```

- 其它方案

lxcfs

- 通过 so 挂载的方式，使容器获得正确的资源信息

Kata

- VM 中跑 container

Virtlet

- 直接启动 VM

> 其中lxcfs的实现就是上面说的，**在容器内**先查询/proc/1/cgroup，然后再查询/sys/fs/cgroup。

将应用迁至Kubernetes

Pod spec

- 初始化需求（init container）。Pod就是由多个（默认）共享网络命名空间的容器组成，要考虑是把一次性的、初始化的工作交给初始化容器进行，比如配置、初始化TOKEN。
- 需要几个Container
- 权限问题。除了默认共享的网络命名空间，还可以共享PID（用一个容器里的进程控制另外一个容器里的进程，可以在Pod spec里写上sharePID）、IPC；MNT虽然不能共享，但是可以让两个容器挂载同一个Volume间接实现。
- 配置管理

- 优雅终止
- 健康检查
  - Liveness Probe
  - Readiness Probe
- DNS策略以及对resolv.conf的影响

> 每个pod都有一个spec.dnsPolicy的属性(参见第8章的笔记)
>
> - 默认配置成ClusterFirst，表示使用集群内的coreDNS服务，此时才会改写容器内的/etc/resolv.conf。
> - 如果配置成default，则容器内的/etc/resolv.conf会和主机上的一模一样。这样**集群内部的所有服务都不能解析**了。
> - 还可以设置为None，表示自定义DNS服务，定义项和/etc/resolv.conf内的项目一样

- imagePullPolicy
- Probe误用会造成严重后果

比如误用导致Pod被重启，但是init进程又无法清理僵尸进程，会导致PID被耗尽。解决方案是

- 单容器进程
- 对于多进程容器，init进程使用tini

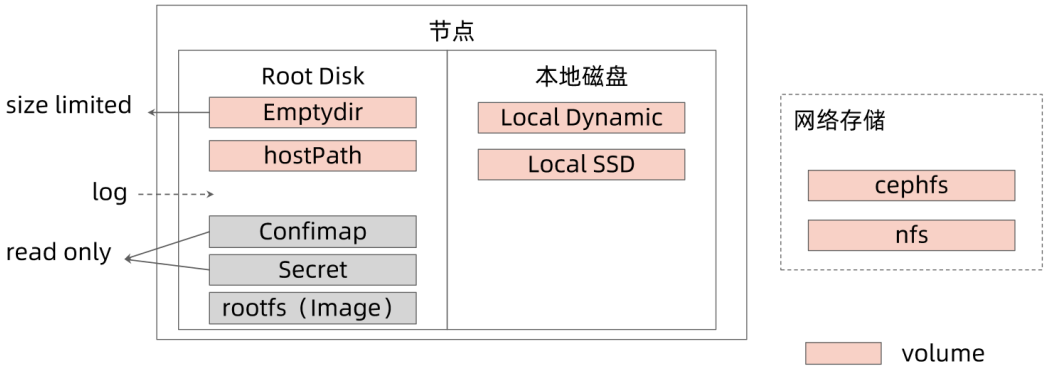
```
ENTRYPOINT ["/tini", "--"]

# Run your program under Tini
CMD ["/your/program", "-and", "-its", "arguments"]
# or docker run your-image /your/program ...
```

- 如果不使用tini，那么
  - 建议采用HttpCheck作为Probe
  - 为Probe执行设置合理的超时时间

- 资源规划
  - 计算资源：CPU/GPU， memory
  - 超售需求
  - 存储资源

    数据管理

    - local-ssd: 独占的本地磁盘，独占 IO，固定大小，读写性能高。
    - Local-dynamic: 基于 LVM，动态分配空间，效率低。



    保存位置

存储卷类型	是应用进程重启后数据是否存在	Pod 重新调度后数据是否存在	容量限制	注意
emptydir	Yes	No	Yes	推荐，了解 emptydir 的 size limit 行为
hostPath	Yes	No	No	需要额外全线，不建议普通应用使用，多开放给节点管理员
local ssd/dynamic	Yes	No	Yes. Quota	无备份
网络存储	Yes	Yes	Yes. Quota	依赖与网络存储稳定性
容器 rootfs	No	No	No	不要写数据

    不要短时间内打印过多日志以防在日志滚动之前节点磁盘就被耗光

- 高可用部署

需要多少实例？

如何控制失败域，部署在几个地区，AZ，集群？

如何进行精细的流量控制？

如何做按地域的顺序更新？

如何回滚？

- 应对基础架构的影响

  应用如果部署在K8S之上，由于K8S是一个动态的环境，POD是会动态转移的。K8S增进了基础架构和应用之间互相理解：容器面向应用，所以基础架构使用K8S的API可以很方便的知道节点上运行这哪些应用，但是也带来一些问题需要考虑

系统管理员	应用	类型	影响	建议
自主中断		Kubelet 升级	• 大部分情况容器不会受影响	
	非自主中断		• 某些版本更新会重建 Container • Pods 会在秒级重建，业务恢复速度依赖应用启动速度	• **多实例部署** • 故障域控制，跨节点跨机架部署
		Kubelet 或者运行时无响应	• 如果是短时无响应，应用不应该受影响 • 如果是长时间无响应，需要驱逐 Pod	• 故障域控制，跨节点跨机架部署 • 合理的健康探针 • 设置合理的 toleration seconds
		节点替换	• 节点会被 dran 掉，节点会被删除 • Pod 会被驱逐	• 故障域控制，跨节点跨机架部署 • 使用 PDB 与基础架构约定驱逐策略 • 使用 preStop 备份关键数据
		节点重启	• Pod 会失效数分钟	• 故障域控制，跨节点跨机架部署 • 设置合理的 toleration seconds
非自主中断		节点 Crash	• Pod 会在15分钟后被驱逐 • Pod 会失效15分钟以上	• 故障域控制，跨节点跨机架部署

  - PodDisruptionBudget

    为了在自主中断是保证应用的高可用，可以使用PDB。PDB只在主动驱逐时有效，当节点承压需要驱逐POD时，是不会考虑PDB的，而是按照特定的算法进行的。

    PDB 是为了自主中断时保障应用的高可用。

    在使用 PDB 时，你需要弄清楚你的应用类型以及你想要的应对措施：

    - 无状态应用：
      - 目标：至少有60%的副本 Available。
      - 方案：创建 PDB Object，指定 minAvailable 为60%，或者 maxUnavailable 为40%。
    - 单实例的有状态应用：
      - 目标：终止这个实例之前必须提前通知客户并取得同意。
      - 方案：创建 PDB Object，并设置 maxUnavailable 为0。
    - 多实例的有状态应用：
      - 目标最少可用的实例数不能少于某个数 N， 例如 etcd。
      - 方案：设置 maxUnavailable=1或者 minAvailable=N，分别允许每次只删除一个实例和每次删除 expected_replicas - minAvailable 个实例。

    PDB代表了基础架构与应用团队之间的契约，应用开发人员针对敏感应用，可定义PDB来确保应用不会被意外中断，比如

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: nginx-deployment
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app: nginx
```

基础架构团对在移除一个节点时应遵循如下流程：

```
# make a node unscheduable ,下面cadmin为一个节点的名字
kubectl cordon cadmin
# 排空一个节点，使得它上面的POD平滑迁移至其它节点
kubectl drain cadmin
# 使一个节点可以重新调度
kubectl uncordon cadmin
```

PDB的示例如下:

对如下的部署

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3   #副本数为3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
```

在Deployment指定的副本数不同时，pdb的`ALLOWER DISRUPTIONS`是不同的

```
root@node1:~/go/src/github.com/cncamp/101/module11/drain-node# k get pdb
NAME               MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
nginx-deployment   1               N/A               2                     2d21h
root@node1:~/go/src/github.com/cncamp/101/module11/drain-node# k scale deployment nginx-deployment --replicas=1
deployment.apps/nginx-deployment scaled
root@node1:~/go/src/github.com/cncamp/101/module11/drain-node# k get pdb
NAME               MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
nginx-deployment   1               N/A               0                     2d21h
```

> 如果要驱除一个POD应该应该用EVICTION而不是删除，**但是EVICTION还没有对应的kubectl命令**，只能直接发送HTTP请求
>
> ```
> curl -v -H 'Content-type: application/json' --key client.key --cert
> client.crt
> https://192.168.34.2:6443/api/v1/namespaces/default/pods/nginx-
> deployment-6799fc88d8-ds499/eviction -d @eviction.json
> ```

> 其中eviction.json是一个文件，内容如下
>
> ```
> {
> "apiVersion": "policy/v1",
> "kind": "Eviction",
> "metadata": {
>  "name": "nginx-deployment-6799fc88d8-ds499",
>  "namespace": "default"
> }
> }
> ```
>
> curl命令和eviction.json中的`nginx-deployment-6799fc88d8-ds499`是POD的名字。

- 部署方式

  多少实例

  更新策略

  - MaxSurge
  - MaxUnavailable（需要考虑 ResourceQuota 的限制）

  深入理解 PodTemplateHash 导致的应用的易变性

- 服务发布

  需要把服务发布至集群内部或者外部，服务的不同类型：

  - ClusterIP（Headless）
  - NodePort
  - LoadBalancer
  - ExternalName

  证书管理和七层负载均衡的需求

  需要 gRPC 负载均衡如何做？

  DNS 需求

  与上下游服务的关系

- 服务发布的挑战

  kube-dns
  - DNS TTL 问题

  Service
  - ClusterIP 只能对内
  - Kube-proxy 支持的 iptables/ipvs 规模有限
  - IPVS 的性能和生产化问题
  - kube-proxy 的 drift 问题
  - 频繁的 Pod 变动（spec change，failover，crashLoop）导致 LB 频繁变更
  - 对外发布的 Service 需要与企业 ELB 即成
  - 不支持 gRPC
  - 不支持自定义 DNS 和高级路由功能

  Ingress
  - Spec 要 deprecate

  其他可选方案？

> 上面说的kube-proxy的drift问题指，kube-proxy构成了一个分布式的负载均衡，但如果某些节点的kube-proxy不正常，则会影响其上的POD。

无状态应用管理

- Replicaset描述Pod模板创建多少个实例
- Deployment描述是部署过程，有一个参数spce.revisionHistoryLimit指定会保存多少个版本，而annotations.deployment.kubernetes.io/revision指示了的当前使用的版本。这两个参数可以用来回滚版本。

Operator

Operator = CRD + Controller.

> Kubernetes对象是可扩展的，扩展的方法有：
>
> - 基于原生对象
>
>   - 生成 types 对象，并通过 client-go 生成相应的 clientset，lister，informer。
>   - 实现对象的 registery backend，即定义对象任何存储进 etcd。
>   - 注册对象的 scheme 至 apiserver。
>   - 创建该对象的 apiservice 生命，注册该对象所对应的 api handler。
>
>   基于原生对象往往需要通过 aggregation apiserver 把不同对象组合起来。
>
> - 基于CRD

使用CRD

用户向 Kubernetes API 服务注册一个带特定 schema 的资源，并定义相关 API

- 注册一系列该资源的实例
- 在 Kubernetes 的其它资源对象中引用这个新注册资源的对象实例
- 用户自定义的 controller 例程需要对这个引用进行释义和实施，让新的资源对象达到预期的状态

定义好上述的资源后，接下来定义认证、鉴权，存储至ETCD，开发Controller

借助 Kubernetes RBAC 和 authentication 机制来保证该扩展资源的 security、access control、authentication 和 multitenancy。

将扩展资源的数据存储到 Kubernetes 的 etcd 集群。

借助 Kubernetes 提供的 controller 模式开发框架，实现新的 controller，并借助 APIServer 监听 etcd 集群关于该资源的状态并定义状态变化的处理逻辑。

开发工具kubebuilder

安装：

```
# 下载链接 https://github.com/kubernetes-sigs/kubebuilder/releases
mv <your_downloading_binary> /usr/local/bin/kubebuilder
```

- **create a kubebuilder project, it requires an empty folder**

  ```
  kubebuilder init --domain cncamp.io
  ```

  上面cncamp.io定义了CRD的域名，会生成很多文件，其中一个是cat PROJECT

  ```
  jesse@JESSEMENG-MB0 demo-operator % ls -la
  total 216
  drwxr-xr-x  12 jesse  staff    384 Dec  9 21:39 .
  drwxr-xr-x  14 jesse  staff    448 Dec  9 21:37 ..
  -rw-------   1 jesse  staff    129 Dec  9 21:39 .dockerignore
  -rw-------   1 jesse  staff    367 Dec  9 21:39 .gitignore
  -rw-------   1 jesse  staff    776 Dec  9 21:39 Dockerfile
  -rw-------   1 jesse  staff   4881 Dec  9 21:39 Makefile
  -rw-------   1 jesse  staff    127 Dec  9 21:39 PROJECT
  drwx------   6 jesse  staff    192 Dec  9 21:39 config
  -rw-------   1 jesse  staff    156 Dec  9 21:39 go.mod
  -rw-r--r--   1 jesse  staff  77000 Dec  9 21:39 go.sum
  drwx------   3 jesse  staff     96 Dec  9 21:39 hack
  -rw-------   1 jesse  staff   2780 Dec  9 21:39 main.go
  ```

- **check project layout**

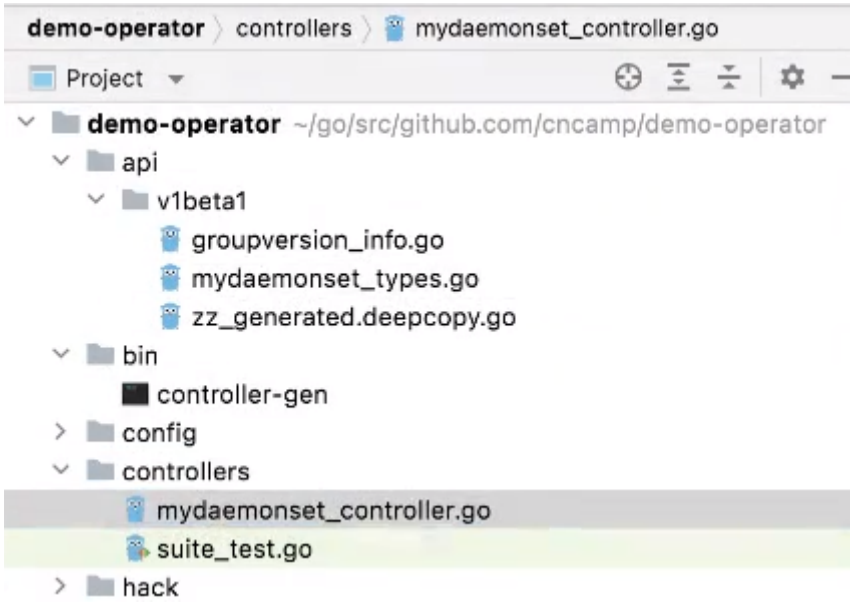
```
cat PROJECT
domain: cncamp.io
layout:
- go.kubebuilder.io/v3
projectName: mysts
repo: github.com/cncamp/demo-operator
version: "3"
```

- create API, create resource[Y], create controller[Y]

```
# 下面相当于是定义了GKV
kubebuilder create api --group apps --version v1beta1 --kind MyDaemonset
```

```
jesse@JESSEMENG-MB0 demo-operator % ls -la
total 216
drwxr-xr-x  15 jesse  staff    480 Dec  9 21:41 .
drwxr-xr-x  14 jesse  staff    448 Dec  9 21:37 ..
-rw-------   1 jesse  staff    129 Dec  9 21:39 .dockerignore
-rw-------   1 jesse  staff    367 Dec  9 21:39 .gitignore
-rw-------   1 jesse  staff    776 Dec  9 21:39 Dockerfile
-rw-------   1 jesse  staff   4881 Dec  9 21:39 Makefile
-rw-------   1 jesse  staff    329 Dec  9 21:41 PROJECT
drwx------   3 jesse  staff     96 Dec  9 21:41 api
drwxr-xr-x   3 jesse  staff     96 Dec  9 21:41 bin
drwx------   8 jesse  staff    256 Dec  9 21:41 config
drwx------   4 jesse  staff    128 Dec  9 21:41 controllers
-rw-------   1 jesse  staff    220 Dec  9 21:41 go.mod
-rw-r--r--   1 jesse  staff  77000 Dec  9 21:39 go.sum
drwx------   3 jesse  staff     96 Dec  9 21:39 hack
-rw-------   1 jesse  staff   3177 Dec  9 21:41 main.go
```

- 打开工程，有类型定义和对应的控制器



在mydaemonset_types.go中定义了API对象常有的spec和status

```
// MyDaemonsetSpec defines the desired state of MyDaemonset
type MyDaemonsetSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
```

```
    // Important: Run "make" to regenerate code after modifying this file

    // Foo is an example field of MyDaemonset. Edit mydaemonset_types.go to
remove/update
    Image string `json:"image,omitempty"`
}

// MyDaemonsetStatus defines the observed state of MyDaemonset
type MyDaemonsetStatus struct {
    AvaiableReplicas int `json:"avaiableReplicas,omitempty"`
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}
```

- make generate生成deepcopy等框架代码
- **check Makefile**

```
Build targets:
    ### create code skeletion
    manifests: generate crd
    generate: generate api functions, like deepCopy

    ### generate crd and install
    run: Run a controller from your host.
    install: Install CRDs into the K8s cluster specified in ~/.kube/config.

    ### docker build and deploy
    docker-build: Build docker image with the manager.
    docker-push: Push docker image with the manager.
    deploy: Deploy controller to the K8s cluster specified in
~/.kube/config.
```

- **generate crd**

```
make manifests
```

- 编写controller逻辑
- **build & install**

```
make build
make docker-build
make docker-push
make deploy
```

- **enable webhooks**

  因为CRD完全是自定义的，如何让K8S知道其中的字段是合法、完整的呢？这就需要定义webhook，让API server调用以做validating和defaulting

  - **install cert-manager**

    ```
    kubectl apply -f https://github.com/jetstack/cert-
    manager/releases/download/v1.6.1/cert-manager.yaml
    ```

  - **create webhooks**

```
kubebuilder create webhook --group apps --version v1beta1 --kind
MyDaemonset --defaulting --programmatic-validation
```

- 编写具体代码
- **enable webhook in** `config/default/kustomization.yaml`
- **redeploy**

Helm

第一，Helm就是一个类似模板引擎的工具，用于管理一堆Spec——即创建一个应用实例的必要的配置组。

配置信息被归类为模板（Template，相对稳定的部分）和值（Value，变化的部分），它们经过渲染生成最终的**可发布**对象。

第二，Helm可以将Template和Value打包，配上对应的描述文件，做成一个chart，然后存放到任何符合OCI规范的仓库中。

Helm的结构

现在Helm只有一个客户端，相关的配置文件都以Secret的形式存在。

使用

- $ helm create myapp

```
$ cd myapp/
$ tree
.
├── charts
├── Chart.yaml
├── templates
|   ├── deployment.yaml
|   ├── _helpers.tpl        # 一个脚本文件
|   ├── hpa.yaml
|   ├── ingress.yaml
|   ├── NOTES.txt
|   ├── serviceaccount.yaml
|   ├── service.yaml
|   └── tests
|       └── test-connection.yaml
└── values.yaml

3 directories, 10 files
```

这样在目录myapp\templates下就生成了很多**Template**，比如deployment.yaml(里面有很多变量和语句)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "myapp.fullname" . }}
  labels:
    {{- include "myapp.labels" . | nindent 4 }}
spec:
```

```
{{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
{{- end }}
  selector:
    matchLabels:
      {{- include "myapp.selectorLabels" . | nindent 6 }}
  template:
    metadata:
    {{- with .Values.podAnnotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
    {{- end }}
      labels:
        {{- include "myapp.selectorLabels" . | nindent 8 }}
    spec:
      {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      serviceAccountName: {{ include "myapp.serviceAccountName" . }}
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
        - name: {{ .Chart.Name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag |
default .Chart.AppVersion }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
      {{- with .Values.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.affinity }}
      affinity:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.tolerations }}
      tolerations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
```

这些变量的值取自于Chart.yaml

```yaml
apiVersion: v2
name: myapp
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into
versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart
developer. They're included as
# a dependency of application charts to inject those utilities and functions
into the rendering
# pipeline. Library charts do not define any templates and therefore cannot
be deployed.
type: application

# This is the chart version. This version number should be incremented each
time you make changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed. This version
number should be
# incremented each time you make changes to the application. Versions are
not expected to
# follow Semantic Versioning. They should reflect the version the
application is using.
appVersion: 1.16.0
```

或者values.yaml

```yaml
# Default values for myapp.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: ""

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
```

```
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname
template
  name: ""

podAnnotations: {}

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local

resources: {}
  # We usually recommend not to specify default resources and to leave this
as a conscious
  # choice for the user. This also increases chances charts run on
environments with little
  # resources, such as Minikube. If you do want to specify resources,
uncomment the following
  # lines, adjust them as necessary, and remove the curly braces after
'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 100
  targetCPUUtilizationPercentage: 80
  # targetMemoryUtilizationPercentage: 80
```

```
nodeSelector: {}

tolerations: []

affinity: {}
```

- 安装

```
helm install ./myapp
```

复用已存在的成熟Helm Release

- 针对 Helm release repo 的操作

helm repo add grafana https://grafana.github.io/helm-charts

helm repo update

helm repo list

helm search repo grafana

- 从 remote repo 安装 Helm chart

helm upgrade --install loki grafana/loki-stack

下载并从本地安装 helm chart

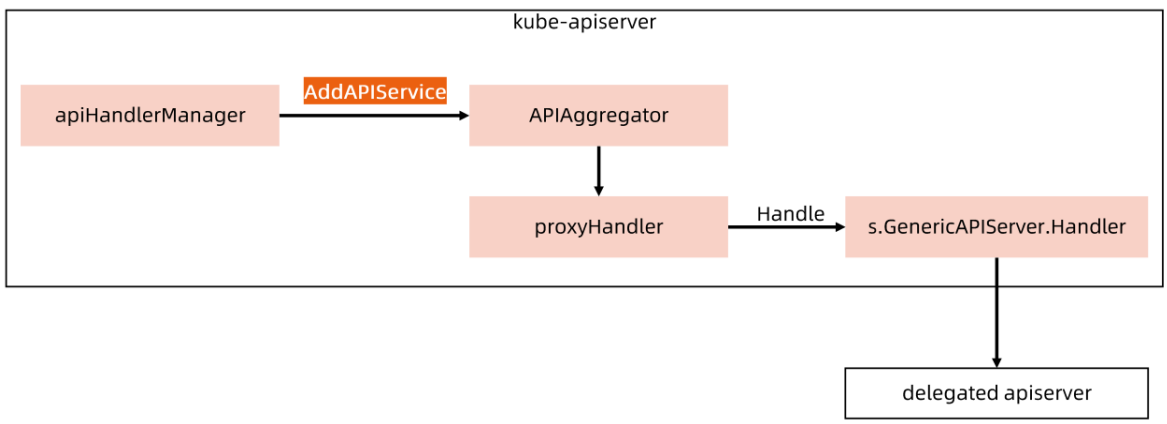
- helm pull grafana/loki-stack

helm upgrade --install loki ./lock-stack

Metrics-server

Aggregated APIServer

K8S的APIserver本质上就是一个跑在GRPC上的HTTP SERVER，它还具有反向代理功能，将请求转发至另外的服务，此时就是一个Aggregated APIServer



如果一个路径不是apiserver的原生路径，那它机会去查找API SERVICE

```
$ k get apiservice
NAME                                   SERVICE   AVAILABLE   AGE
v1.                                    Local     True        4d6h
v1.admissionregistration.k8s.io        Local     True        4d6h
v1.apiextensions.k8s.io                Local     True        4d6h
v1.apps                                Local     True        4d6h
v1.authentication.k8s.io               Local     True        4d6h
v1.authorization.k8s.io                Local     True        4d6h
v1.autoscaling                         Local     True        4d6h
v1.batch                               Local     True        4d6h
v1.certificates.k8s.io                 Local     True        4d6h
v1.coordination.k8s.io                 Local     True        4d6h
v1.crd.projectcalico.org               Local     True        8h
v1.discovery.k8s.io                    Local     True        4d6h
v1.events.k8s.io                       Local     True        4d6h
v1.networking.k8s.io                   Local     True        4d6h
v1.node.k8s.io                         Local     True        4d6h
v1.operator.tigera.io                  Local     True        8h
v1.policy                              Local     True        4d6h
v1.rbac.authorization.k8s.io           Local     True        4d6h
v1.scheduling.k8s.io                   Local     True        4d6h
v1.storage.k8s.io                      Local     True        4d6h
v1beta1.batch                          Local     True        4d6h
v1beta1.discovery.k8s.io               Local     True        4d6h
v1beta1.events.k8s.io                  Local     True        4d6h
v1beta1.flowcontrol.apiserver.k8s.io   Local     True        4d6h
v1beta1.node.k8s.io                    Local     True        4d6h
v1beta1.policy                         Local     True        4d6h
v1beta1.storage.k8s.io                 Local     True        4d6h
v2.cilium.io                           Local     True        3m41s
v2beta1.autoscaling                    Local     True        4d6h
v2beta2.autoscaling                    Local     True        4d6h
```

每个group和version都对应一个apiservice，指明了请求由谁处理，可以配置其不由默认的API SERVER处理。比如如下的定义

```yaml
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  labels:
    k8s-app: metrics-server
  name: v1beta1.metrics.k8s.io  #指定被处理的对象
spec:
  group: metrics.k8s.io
  groupPriorityMinimum: 100
  insecureSkipTLSVerify: true
  service:
    name: metrics-server     # 由kube-system名称空间下的metrics-server进行处理
    namespace: kube-system
  version: v1beta1
  versionPriority: 100
```

创建后

```
$ ks get svc
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
kube-dns         ClusterIP   10.96.0.10     <none>        53/UDP,53/TCP,9153/TCP
  4d6h
metrics-server   ClusterIP   10.106.85.92   <none>        443/TCP
   54s
```

Metrics-Server

Metrics-Server本身是K8S监控体系中的核心组件之一，负责从kublet收集资源指标，然后对这些指标进行聚合，并在Apiserver中通过Metircs API（/apis/metrics.k8s.io/）公开暴露它们。

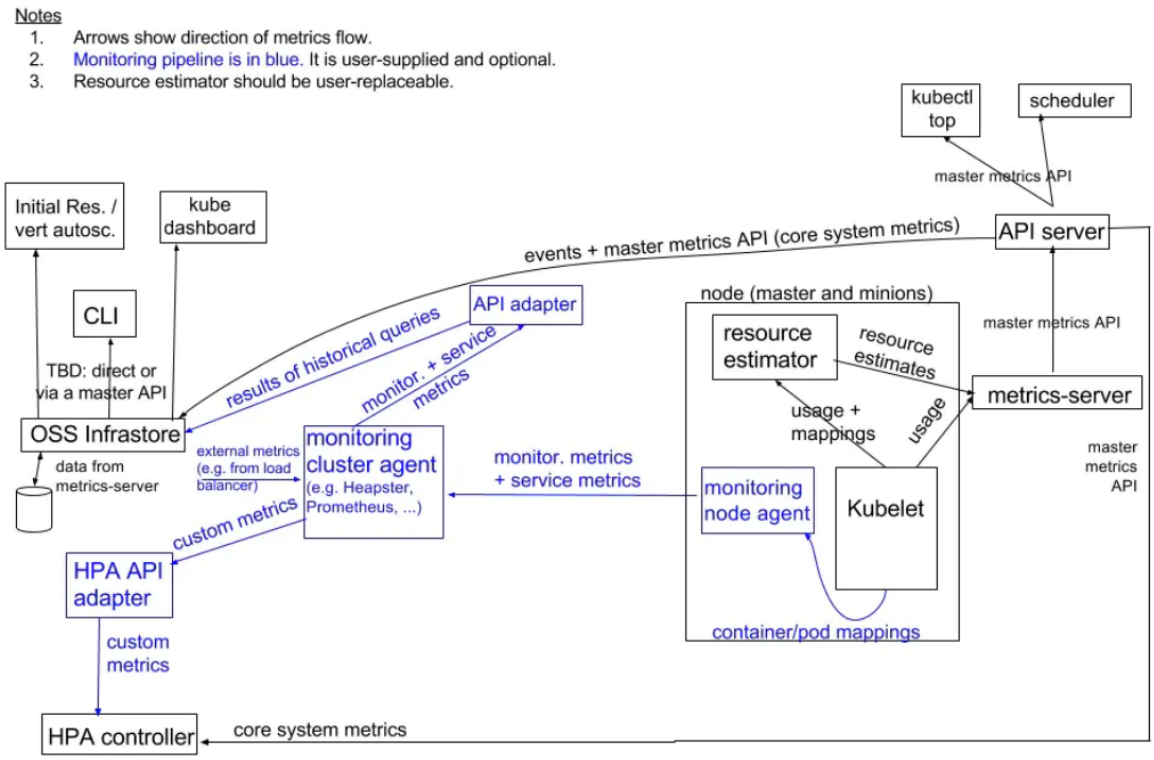
kubectl本身有命令支持查看节点和POD的资源使用情况

```
$ k top no
NAME                          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
xiaokai-thinkpad-x13-gen-1    367m         2%     3271Mi          21%

$ ks top pods
NAME                                                CPU(cores)   MEMORY(bytes)
cilium-operator-7b5c99b568-xclmv                    2m           22Mi

cilium-sh58k                                        7m           219Mi
coredns-7f6cbbb7b8-gdrdg                            5m           13Mi

coredns-7f6cbbb7b8-pw589                            3m           13Mi

etcd-xiaokai-thinkpad-x13-gen-1                     29m          97Mi

kube-apiserver-xiaokai-thinkpad-x13-gen-1           88m          438Mi
kube-controller-manager-xiaokai-thinkpad-x13-gen-1  19m          54Mi

kube-proxy-cnvvf                                    1m           17Mi

kube-scheduler-xiaokai-thinkpad-x13-gen-1           5m           19Mi

metrics-server-6d7c55f98c-mcl2d                     4m           16Mi

# 还可以使用$ ks top pods -v 9查看更详细的信息，从输出信息可以看到ks top pods实际是做了如下请求
# GET https://192.168.34.2:6443/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods 200 OK in 3 milliseconds
# 上面的请求/apis/metrics.k8s.io/v1beta1实际上是交给Metrics Server处理的

$ k get apiservice
...
v1beta1.flowcontrol.apiserver.k8s.io   Local                          True
4d7h
v1beta1.metrics.k8s.io                 kube-system/metrics-server     True
7m48s
v1beta1.node.k8s.io                    Local                          True
4d7h
...
```

本质

Metrics-Server其实是使用下面的命令从cAdvisor获取原始数据的

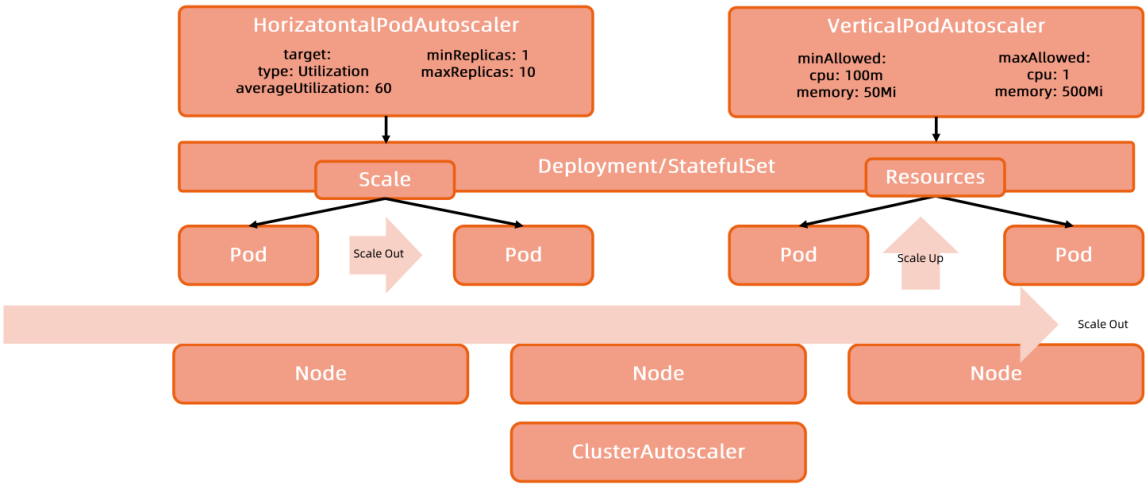
```
 k get --raw "/api/v1/nodes/xiaokai-thinkpad-x13-gen-1/proxy/metrics/resource"
# HELP container_cpu_usage_seconds_total [ALPHA] Cumulative cpu time consumed by
the container in core-seconds
# TYPE container_cpu_usage_seconds_total counter
...
```



在进行HPA和VPA时，使用Metrics-Server提供的指标。

自动扩缩容-HPA

云原生的弹性能力



比如下面是V1版的HorizontalPodAutoscaler定义：

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10                    # 副本集数量的调节范围
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1              # 目标应用
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 50    # 调节目标：CPU使用率
```

V1版不满足需要，比如水平扩展的指标不仅是CPU使用率，还有相应时间、TPS、QPS等，所以社区又出了V2版，支持**自定义的指标**

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

自定义指标只使用原始值，不使用比率

HPA指标类型

内置的指标类型-Resource类型

```
# Resource 类型的指标
  - type: Resource
    resource:
      name: cpu
      # Utilization 类型的目标值，Resource 类型的指标只支持 Utilization 和 AverageValue 类型的目标值
      target:
        type: Utilization
        averageUtilization: 50
```

Pod类型的指标

```
# Pods 类型的指标
 - type: Pods
   pods:
     metric:
       name: packets-per-second
     # AverageValue 类型的目标值，Pods 指标类型下只支持 AverageValue 类型的目标值
     target:
       type: AverageValue
       averageValue: 1k
```

算法细节

期望副本数 = ceil[当前副本数 * (当前指标 / 期望指标)]

当前度量值为 200m，目标设定值为 100m，那么由于 200.0/100.0 == 2.0， 副本数量将会翻倍。如果当前指标为 50m，副本数量将会减半，因为 50.0/100.0 == 0.5。 如果计算出的扩缩比例接近 1.0 （根据--horizontal-pod-autoscaler-tolerance 参数全局配置的容忍值，默认为 0.1），将会放弃本次扩缩。

扩缩容策略

在Spec字段的behavior部分可以指定一个或多个扩缩策略：

```
behavior:
 scaleDown:
  policies:
  - type: Pods
   value: 4
   periodSeconds: 60
  - type: Percent
   value: 10
   periodSeconds: 60
```

注意事项

- 多个冲突的HPA会同时创建到同一个应用，导致无法预期的行为，所以一定要消息维护HPA规则；
- HPA管理的是Deployment的replicas字段

问题

HPA本质是一个自动控制器，但是因为下面各种延迟（相当于惯性）的存在，用“自动化”专业的背景的我看来，现行控制算法的问题归结于一个：根本就不是一个合格的自动控制算法，比如没有对指标做数字信号处理，导致副本数量频繁变化——抖动（Thrashing）：

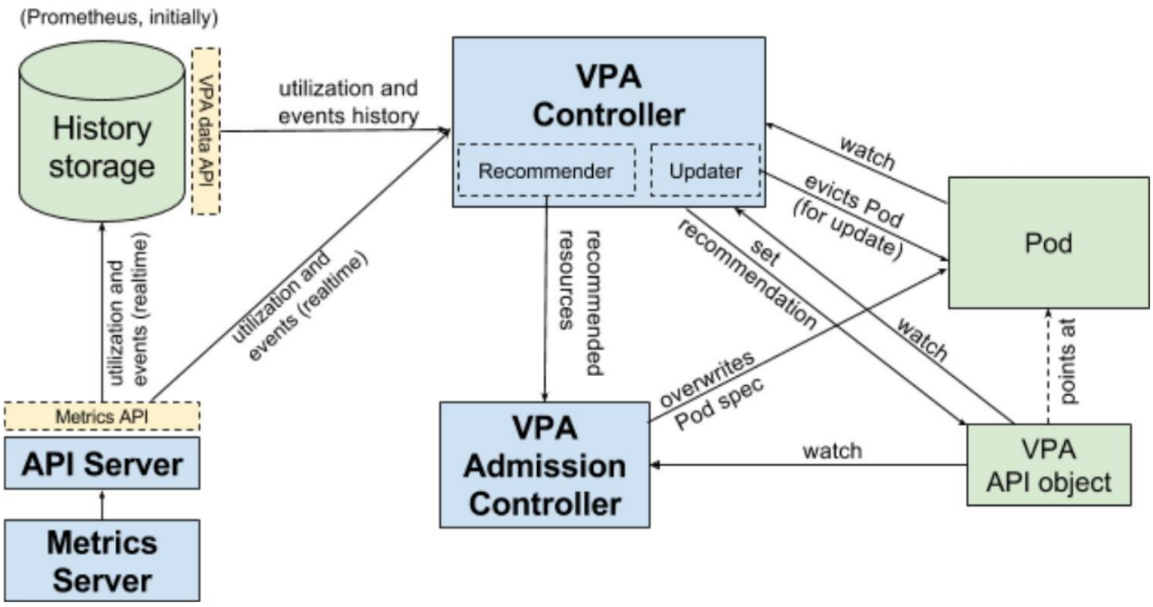
- 应用指标数据已经超出阈值；
- HPA 定期执行指标收集滞后效应；
- HPA 控制 Deployment 进行扩容的时间；
- Pod 调度，运行时启动挂载存储和网络的时间；
- 应用启动到服务就绪的时间。

自动扩缩容-VPA

意义

- Pod 资源用其所需，提升集群节点使用效率；
- 不必运行基准测试任务来确定 CPU 和内存请求的合适值；
- VPA 可以随时调整 CPU 和内存请求，无需人为操作，因此可以减少维护时间。

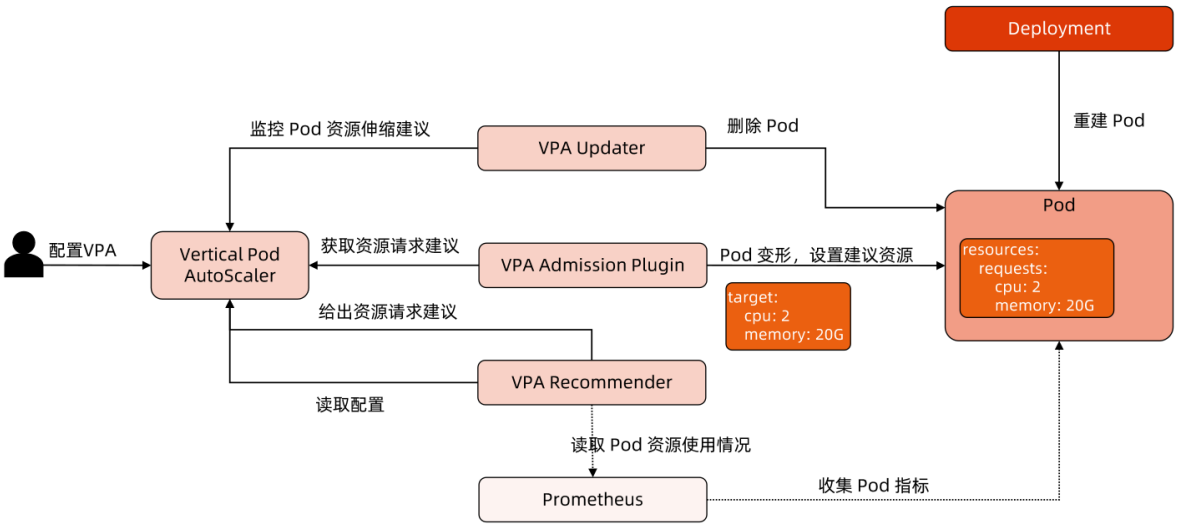
架构



注意：

- VPA调整POD的资源并不是原地调整，而是会先做驱逐，然后调整。即使现在社区提出的原地调整PR被合并，也很难解决节点上资源不足时调整的需要；
- 同时，对于JAVA应用，堆内存涨上去以后，就不会再释放了；
- VPA的成熟度不足！

工作原理



Recommender的目标

这里内存和CPUS使用率被看做独立的**随机变量**。

推荐模型（MVP）假设内存和 CPU 利用率是独立的随机变量，其分布等于过去 N 天观察到的变量（推荐值为 N=8 以捕获每周峰值）。

- 对于 CPU，目标是将容器使用率超过请求的高百分比（例如 95%）时的时间部分保持在某个阈值（例如 1% 的时间）以下。在此模型中，“CPU 使用率”被定义为在短时间间隔内测量的平均使用率。测量间隔越短，针对尖峰、延迟敏感的工作负载的建议质量就越高。最小合理分辨率为 1/min，推荐为 1/sec。
- 对于内存，目标是将特定时间窗口内容器使用率超过请求的概率保持在某个阈值以下（例如，24 小时内低于 1%）。窗口必须很长（≥ 24 小时）以确保由 OOM 引起的驱逐不会明显影响（a）服务应用程序的可用性（b）批处理计算的进度（更高级的模型可以允许用户指定 SLO 来控制它）。

基本上就是一个根据随机变量时间序列调整CPU使用率和内存使用率的自动控制过程。