



应用容器化最佳实践

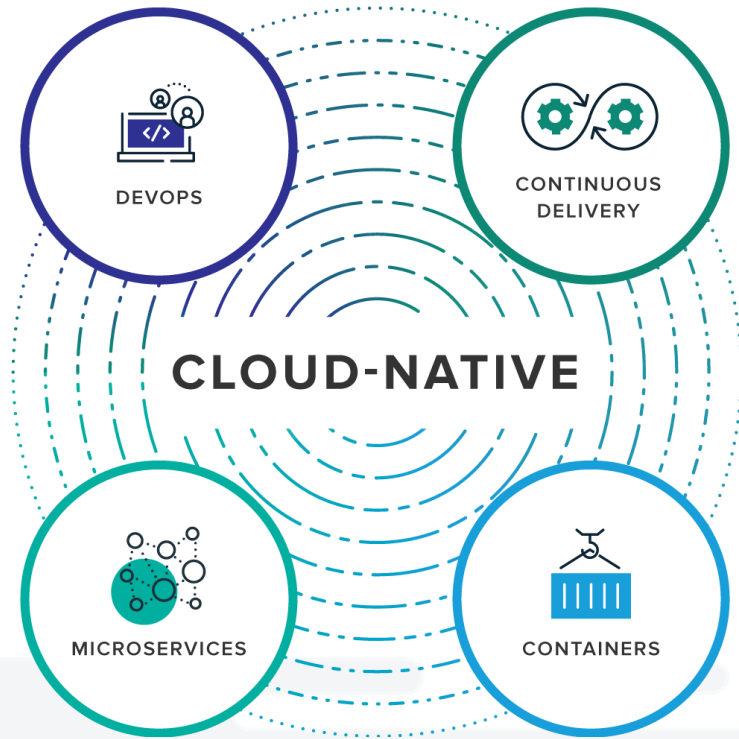
Dehua Ye
dehua@rancher.com

2020.11.14

云原生应用

云原生 (Cloud Native) 是一种充分利用云计算优势，用于构建和部署应用的方式。云原生应用专为云模型而开发，团队可以快速将应用构建和部署到可提供横向扩展和硬件解耦的平台，为企业提供更高的敏捷性、弹性和云间的可移植性。

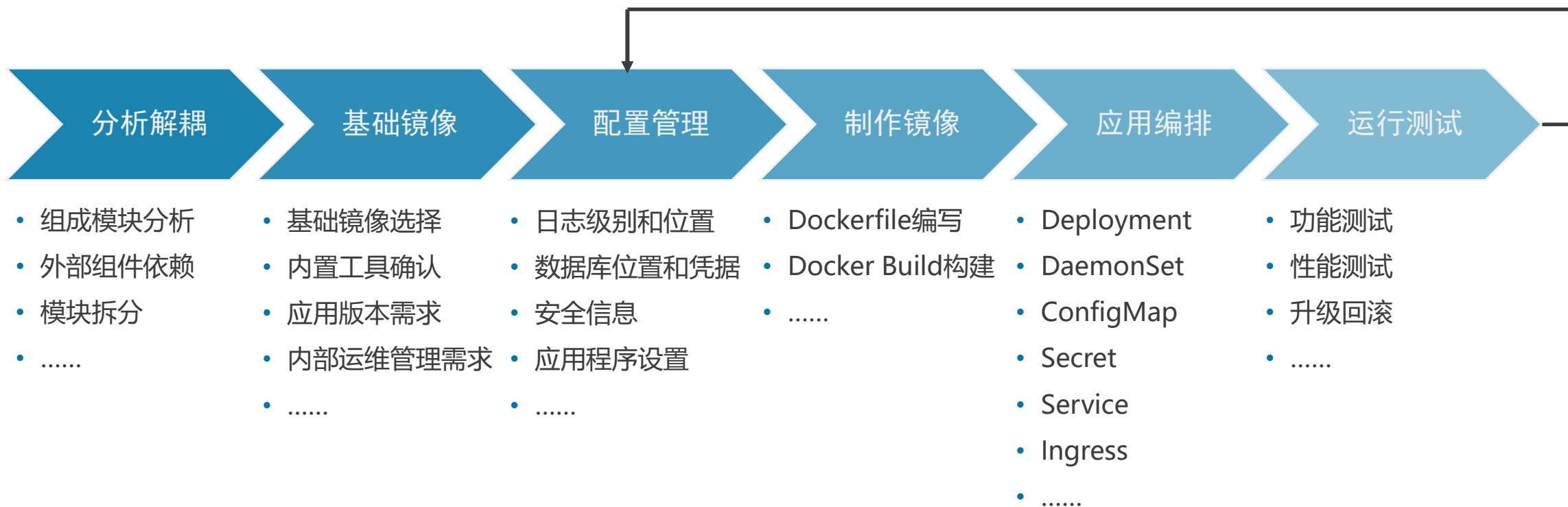
| 云原生应用 | 传统的企业应用 |
|---------|---------|
| 可预测 | 不可预测 |
| 操作系统抽象化 | 依赖操作系统 |
| 合适的容量 | 过多容量 |
| 协作 | 孤立 |
| 持续交付 | 瀑布式开发 |
| 独立 | 依赖 |
| 自动化可扩展性 | 手动扩展 |
| 快速恢复 | 恢复缓慢 |



| 十二因素应用 (Twelve-factor App) | |
|----------------------------|-------------|
| 基准代码 | 依赖 |
| 配置 | 后端服务 |
| 构建、发布、运行 | 进程 |
| 端口绑定 | 并发 |
| 易处理 | 开发环境与线上环境等价 |
| 日志 | 管理进程 |

https://12factor.net/zh_cn/

应用容器化一般流程



应用分析解耦

- ✓ 应用开发语言或平台及对应版本
- ✓ 应用技术架构、运行环境及组件依赖
- ✓ 应用运行包大小、一般启动时长、是否有启停脚本
- ✓ 应用当前软硬件监控、调用链监控、日志分析方案
- ✓ 应用当前配置管理方式、是否有健康检查接口
- ✓ 应用是否实现状态数据外部化管理（如Session会话）
- ✓ 系统部署架构及当前生产高可用方案
- ✓ 系统目前日常及特殊高峰期资源使用情况（CPU、内存等）
- ✓ 系统当前发布方式（是否已实现持续集成或构建管理）
- ✓ 是否有特定的操作系统、GPU或其他底层资源依赖
- ✓ 系统间集成方式（应用层集成、数据库层集成）
- ✓ 业务场景及用户使用情况（用户数、并发数、集中时间段）
- ✓ 系统目前是否有已知的安全漏洞及修复计划
- ✓ 是否可允许停机升级、停机升级窗口时间段及时长
- ✓ 应用研发及维护团队情况
- ✓ 应用近期发布及迭代计划情况



制作容器镜像

Dockerfile常用命令：

- FROM：指定容器基础镜像
- LABEL：给镜像打上标签，比如添加镜像维护者信息
- ADD：向镜像添加文件，可以使用URL
- COPY：向镜像添加文件，不可以使用URL，不会解压缩
- ENV：设置环境变量，可以被后面的指令使用
- RUN：用于容器中运行指定命令，每一次都会生成新的镜像层
- USER：指定RUN、CMD、ENTRYPOINT、容器启动使用的用户
- WORKDIR：RUN、CMD、ENTRYPOINT命令执行目录
- VOLUME：容器创建时将该目录下的数据外挂存储卷
- EXPOSE：声明容器需要对外暴露的端口号
- ENTRYPOINT：容器启动时的指令，多个ENTRYPOINT则只有最后一个生效
- CMD：容器启动时的指令，多个CMD则只有最后一个生效

```
FROM maven:3.5.2-jdk-8-alpine AS MAVEN_BUILD
LABEL MAINTAINER=dehua@rancher.com
COPY pom.xml /build/
COPY src /build/src/
WORKDIR /build/
RUN mvn package
```

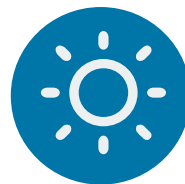
```
FROM openjdk:8-jre-alpine
USER appuser
WORKDIR /app
VOLUME /tmp
EXPOSE 8080
COPY --from=MAVEN_BUILD /build/target/demo-0.1.0.jar /app/
ENTRYPOINT ["java", "-jar", "docker-boot-intro-0.1.0.jar"]
```

Dockerfile最佳实践



使用更小的base镜像

使用体积小的base镜像，如：Alpine、Busybox、Scratch



串联Dockerfile 命令

因为每一个RUN命令对应新的镜像层，我们应该将多个命令通过&&等方式组成同一行命令，减少镜像层数



多阶段构建

支持多个FROM，一个用于编译，另外一个将编译后的可执行文件打入镜像，减少体积



缓存清理

执行一些安装命令如yum install、apt-get install时记得安装完后将缓存清理干净

Dockerfile官方最佳实践文档：https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Dockerfile示例

```
FROM golang:1.11-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

尽量确保镜像的安全、精简、可读、易维护：选择精简安全的系统镜像、串联执行命令减少镜像层数、多阶段构建、缓存清理、只从可信源安装软件、只安装必要的软件、添加维护人员信息、添加必要的备注说明.....

```
FROM buildpack-deps

RUN apt-get update && apt-get install -y \
    ca-certificates \
    curl

# verify gpg and sha256: http://nodejs.org/dist/v0.10.30/SHASUMS256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys
    7937DFD2AB06298B2293C3187D33FF9D0246406D

ENV NODE_VERSION 0.11.14
ENV NPM_VERSION 2.1.18

RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc | sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/local --strip-components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.asc \
    && npm install -g npm@$NPM_VERSION \
    && npm cache clear

CMD [ "node" ]
```

Dockerfile缓存陷阱

以下示例基础镜像基于ubuntu:18.04，其他操作系统类似：

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install -y curl
```

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

```
FROM ubuntu:18.04
RUN apt-get update && apt-get install -y curl
  && rm -rf /var/lib/apt/lists/*
```

```
FROM ubuntu:18.04
RUN apt-get update && apt-get install -y \
  curl \
  nginx \
  && rm -rf /var/lib/apt/lists/*
```

- 请总是将 apt-get update 和 apt-get install 写在同一个RUN语句中，永远不要分开，以避免 apt-get update 缓存问题
- 请避免执行 apt-get upgrade 和 dist-upgrade，因为一方面父级镜像中的有些核心软件包无法在非特权容器中更新升级，另一方面大范围的更新软件及其依赖，会增加镜像大小。所以，建议只更新必要的指定软件，并做好清除动作

通过ConfigMap管理可变应用配置

一般应用普遍会有从配置文件、命令行参数或者环境变量中读取一些配置信息的需求，Kubernetes提供了ConfigMap资源对象来实现配置管理，可以通过以下几种方式来使用ConfigMap配置Pod中的容器：

- 容器 entrypoint 的命令行参数
- 容器的环境变量
- 在只读卷里面添加一个文件，应用读取
- 编写代码在 Pod 中运行，应用通过使用 Kubernetes API 来读取

使用Secret管理敏感信息配置

由于ConfigMap是明文存储，适合用来存储非安全的配置信息，如果涉及安全敏感的数据，推荐使用另一个Secret资源对象。Secret 对象用来保存敏感信息，例如密码、OAuth 令牌和 SSH 密钥，这些信息放在Secret中比放在Pod的定义或者容器镜像中更加安全和灵活。

Secret 主要使用的有以下三种类型：

- Opaque：base64 编码格式的 Secret，用来存储密码、密钥等。但也可以通过base64-decode解码得到原始数据，安全性较弱
- kubernetes.io/dockerconfigjson：用来存储私有docker registry 镜像库的认证信息
- kubernetes.io/service-account-token：使用ServiceAccount 资源对象时，会默认创建一个对应的 Secret 对象，对应的Secret 会自动挂载到Pod 目录 /run/secrets/kubernetes.io/serviceaccount

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
type: Opaque
data:
  username: YWRtaW4= # Base64加密后
  password: YWRtaW4zMjE= # Base64加密后
```

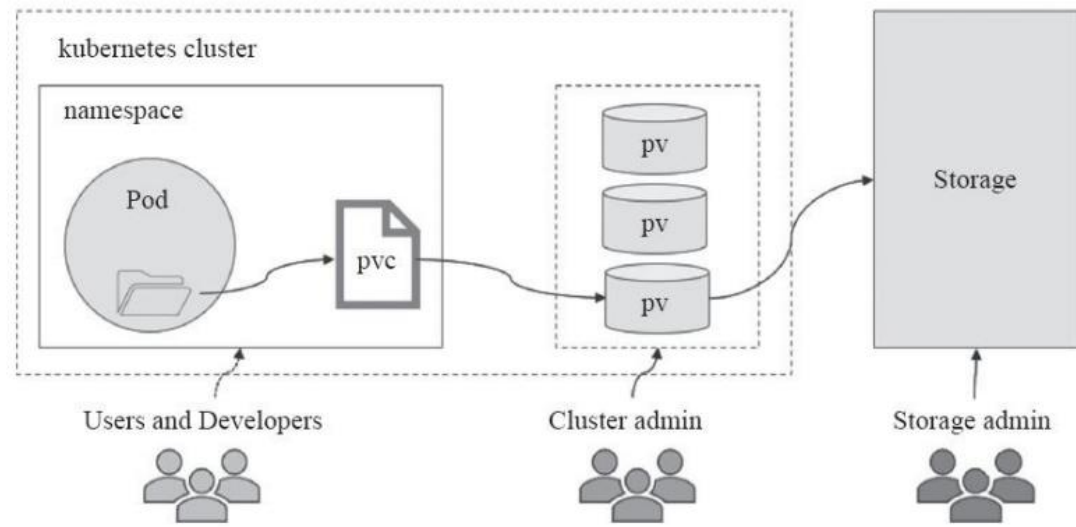
```
apiVersion: v1
kind: Pod
metadata:
  name: demoapp
spec:
  containers:
    - name: secret1
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: demo-secret
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: demo-secret
              key: password
```

通过PV/PVC实现应用数据持久化

虽然可以通过ConfigMap和Secret来实现对应用的配置管理，但不管是有状态应用还是无状态应用，一般都会在应用运行过程中产生业务数据，那么这部分数据的持久化存储就十分重要。

从存储的使用和管理角度来看：

- 存储管理团队：维护存储设备，确保存储的稳定性
- 云平台管理团队：创建StorageClass或PV，根据资源特性和需求做好存储的使用规划
- 应用开发团队：按需创建PVC，应用绑定PVC使用存储资源



| Volume Plugin | ReadWriteOnce | ReadOnlyMany | ReadWriteMany |
|----------------------|---------------|--------------|------------------------------------|
| AWSElasticBlockStore | ✓ | - | - |
| AzureFile | ✓ | ✓ | ✓ |
| AzureDisk | ✓ | - | - |
| CephFS | ✓ | ✓ | ✓ |
| Cinder | ✓ | - | - |
| FC | ✓ | ✓ | - |
| FlexVolume | ✓ | ✓ | - |
| Flocker | ✓ | - | - |
| GCEPersistentDisk | ✓ | ✓ | - |
| Glusterfs | ✓ | ✓ | ✓ |
| HostPath | ✓ | - | - |
| iSCSI | ✓ | ✓ | - |
| Quobyte | ✓ | ✓ | ✓ |
| NFS | ✓ | ✓ | ✓ |
| RBD | ✓ | ✓ | - |
| VsphereVolume | ✓ | - | - (works when pods are collocated) |
| PortworxVolume | ✓ | - | ✓ |
| ScaleIO | ✓ | ✓ | - |
| StorageOS | ✓ | - | - |

基于NFS的PV动态供给使用示例

目前基于NFS的容器云存储方案仍然在被普遍使用，假设存储管理员已经维护好了NFS存储，云平台管理员也已经部署好nfs-client-provisioner并配置了StorageClass，并将其设置为默认缺省存储，那么对于使用者：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1000Mi
```

Step 1: 创建PVC，自动关联StorageClass，动态创建PV

```
kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
    - name: test-pod
      image: busybox
      command:
        - "/bin/sh"
      args:
        - "-c"
        - "touch /mnt/SUCCESS && exit 0 || exit 1"
      volumeMounts:
        - name: mnt
          mountPath: "/mnt"
  volumes:
    - name: mnt
      persistentVolumeClaim:
        claimName: test-pvc
```

Step 2: 创建应用工作负载（Pod、Deployment、StatefulSet等），关联PVC使用

使用Init Container检查依赖服务

一个 Pod 中可以有多多个 Container，也可以有多多个 Init Container，Init Container 会在应用 Container 启动之前启动，并且如果有多多个 Init Container 会依次启动，只有当前一个 Container 运行成功了，才会运行启动下一个 Container，所有 Init Container 都运行结束了，应用才会启动。因此，我们可以借助 Init Container 来检查应用的依赖服务（如：DB、Redis等）是否已经可用，也可以在 Init Container 中执行一些预先的初始化设置工作（如：容器权限设置等）

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: app-server
    image: yedward/app-server:latest
    ports:
      - name: http
        containerPort: 80
        protocol: TCP
    livenessProbe:
      httpGet:
        path: /health
        port: http
        initialDelaySeconds: 60
        periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /api/notice
        port: http
        initialDelaySeconds: 60
        periodSeconds: 10
  initContainers:
  - name: init-redis
    image: busybox:1.31
    command: ['sh', '-c', 'until nslookup redis-server; do echo waiting for redis; sleep 2; done;']
  - name: init-mysql
    image: busybox:1.31
    command: ['sh', '-c', 'until nslookup mysql-server; do echo waiting for mysql; sleep 2; done;']
```

应用配置存活、就绪和启动探测

- livenessProbe: 判断容器是否存活, 即Pod是否为running状态, 如果LivenessProbe探测到容器不健康, 则kubelet将kill掉容器, 并根据容器的重启策略是否重启, 如果没有配置LivenessProbe, 则Kubelet认为容器的LivenessProbe探测的返回值永远成功。
- readinessProbe: 判断容器是否启动完成, 即容器的Ready是否为True, 可以接收请求, 如果ReadinessProbe探测失败, 则容器的Ready将为False, 控制器将此Pod的Endpoint从对应的service的Endpoint列表中移除, 从此不再将任何请求调度此Pod上, 直到下次探测成功。
- startupProbe: 启动探测, 判断容器内的应用程序是否已启动。如果配置了启动探测, 就会先禁用所有其他探测, 直到它成功为止。如果启动探测失败, kubelet将杀死容器, 容器将服从其重启策略。如果容器没有提供启动探测, 则默认状态为成功。

```
apiVersion: v1
kind: Pod
metadata:
  name: app-server
  labels:
    app: app-server
spec:
  containers:
  - name: app-server
    image: yedward/app-server
    ports:
      - containerPort: 8080
    readinessProbe:
      httpGet:
        path: /healthz/readiness
        port: 8080
      failureThreshold: 1
      periodSeconds: 10
    livenessProbe:
      httpGet:
        path: /healthz/liveness
        port: 8080
      failureThreshold: 1
      periodSeconds: 30
    startupProbe:
      httpGet:
        path: /healthz/startup
        port: 8080
      failureThreshold: 30
      periodSeconds: 10
```

选择合适的日志处理方式

优先推荐使用控制台输出日志，但这个推荐是基于容器只作为简单应用的场景，实际的业务场景中需要根据实际情况按需选择最合适的方式：

➤ 业务直写

- 建议在日志量大的场景中使用
- 针对某些业务指标埋点统计场景

➤ DaemonSet

- 一般建议在中小型集群中使用
- 应用日志格式统一，可统一采集
- 一个配置尽可能多的采集同类数据，减少配置数

➤ Sidecar

- 推荐在超大型的集群中使用
- 针对核心的应用采集要给予充分的资源
- 应用日志格式难以统一，需要分别编写规则

```
### 配置根Logger
log4j.rootLogger=debug,stdout

### 输出到控制台
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyy-MM-dd HH\:mm\:ss} %5p %c{1}\: %L - %m%n
```

应用监控指标采集分析

目前Kubernetes生态比较广泛使用的通用监控方案是Exporters + Prometheus + Grafana，通过各种Exporter采集不同维度的监控指标，并通过Prometheus支持的数据格式暴露出来，Prometheus定期拉取数据并用Grafana展现，异常情况使用AlertManager告警。

常用的一些exporter:

- node_exporter
- jmx_exporter
- mysqld_exporter
- redis_exporter
- elasticsearch_exporter
-

自定义指标

配置自定义指标端口，监控系统将通过这些端口采集自定义指标。

| 容器端口 * | Path | 协议 | |
|----------|--------------|------|------|
| 例如: 8080 | 例如: /metrics | HTTP | 删除指标 |

删除标签 可以输入标签名称或者通过表达式匹配标签名

添加标签名

添加自定义指标

注：相比于基于Exporter的这种通用解决方案，业界一些成熟的开发框架或平台也会提供更加原生的监控解决方案，比如针对Spring社区Spring Boot开发框架Metrics采集的micrometer-registry-prometheus组件。此外，还有一些调用链解决方案也能做到更加全面细粒度的监控，比如目前比较流行的Skywalking、Pinpoint等，类似的还有一些成熟的商业解决方案。

应用如何优雅关停

Kubernetes在做Rolling Update的时候默认会向旧的Pod发出一个SIGTERM信号，如果应用没有对SIGTERM信号做处理，就会立即强制退出程序，此时会导致正在处理中的请求没有处理完就被错误的强制关闭，从而影响到业务。

滚动更新一般步骤：

- 启动一个新的POD
- 等待新的POD进入Ready状态
- 创建Endpoint，将新的 pod 添加到负载均衡后端
- 移除与旧的Pod相关的Endpoint，并且将老 pod 状态设置为 Terminating，此时将不会再有新的请求调度到旧的Pod
- 给旧的Pod发送SIGTERM信号，等待到超时时间设置时长，默认terminationGracePeriodSeconds设置为 30 秒
- 超过terminationGracePeriodSeconds等待时间直接强制 Kill 进程并关闭旧的 pod

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    public void shutdown() {
        System.out.println("shutdown")
    }
}
```

```
"lifecycle": {
  "preStop": {
    "httpGet": {
      "path": "/shutdown",
      "port": 8080,
      "scheme": "HTTP"
    }
  }
}
```

避免应用单点故障

哪怕应用部署时指定运行了多个Pod，但仍然无法明确保证不出现单点故障问题，因为也有可能多个Pod会被调度到同一个节点上。Kubernetes允许为Pod设置反亲和策略，从而实现在指定类型的topologyKey（拓扑域）上的反亲和调度，拓扑域可以是节点，机架，云供应商地区，云供应商区域等。

Kubernetes官方使用说明：

- Pod 间亲和与反亲和需要大量的处理，可能会减慢大规模集群中的调度，因此不建议在大规模集群中使用它们
- Pod 反亲和需要对节点进行一致的标记，即集群中的每个节点必须具有适当的标签能够匹配 topologyKey，如果节点缺少指定的 topologyKey 标签，可能会导致意外行为

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: redis-server
          image: redis:3.2-alpine
```

服务质量QoS

QoS 是服务质量 (Quality of Service) 的缩写, 为保证集群资源被有效调度分配的同时提高资源利用率, Kubernetes针对不同服务质量的预期, 通过 QoS 来对Pod进行服务质量管理。对于Pod而言, 服务质量体现在两个具体的指标: CPU 和内存, 当节点上资源紧张时, Kubernetes 会根据预先设置的不同 QoS 类别进行相应处理。

```
containers:
- name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
- name: bar
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
  requests:
    cpu: 100m
    memory: 100Mi
```

Guaranteed (有保证的)

- Pod中所有容器都且仅设置了CPU和内存的limits
- Pod中所有容器都设置了 CPU 和内存的requests和limits, 且单个容器的requests等于limits (requests不等于0)

```
containers:
- name: foo
  resources:
    requests:
      cpu: 10m
      memory: 1Gi
- name: bar
```

Burstable (不稳定的)

- Pod 中只要有一个容器的 requests 和 limits 的设置不相同, 那么该Pod的QoS即为 Burstable
- 注:** 如果容器指定requests而未指定 limits, 则limits的值等于节点资源的最大值, 如果指定 limits 而未指定 requests, 则 requests 等于 limits

```
containers:
- name: foo
  resources:
- name: bar
  resources:
```

Best-Effort (尽最大努力)

- 如果Pod中所有容器的resources均未设置 requests 与 limits, 该 Pod 的 QoS 即为 Best-Effort
- 对于Best-Effort类型的Pod, Kubernetes不保证其可靠稳定性

干扰预算PDB

如果服务本身存在单点故障，所有副本都在同一个节点，驱逐的时候肯定就会造成服务不可用，这种情况可以通过反亲和性设置和多副本来应对。但是如果服务本身就被打散在多个节点上，这些节点如果都被同时驱逐的话，那么该服务的所有实例都会被同时删除，此时也会造成服务不可用了，这种情况下可以通过配置 PDB (PodDisruptionBudget) 对象来避免所有副本同时被删除，比如可以设置在驱逐的时候，应用最多只有一个副本不可用，就相当于逐个删除并在其它节点上重建。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: wordpress-pdb
  namespace: kube-example
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
```

应用调度抢占优先级

Kubernetes支持多种资源调度模式，基于nodeName和nodeSelector的服务器资源调度，称其为用户绑定策略；基于PriorityClass的同一Node下不同Pod资源的优先级调度，称其为抢占式调度策略。

```
apiVersion: scheduling.k8s.io/v1alpha1
kind: PriorityClass
metadata:
  name: high-priority
  value: 1000000
  globalDefault: false
  description: "This priority class should be used for
high priority service pods only."
```

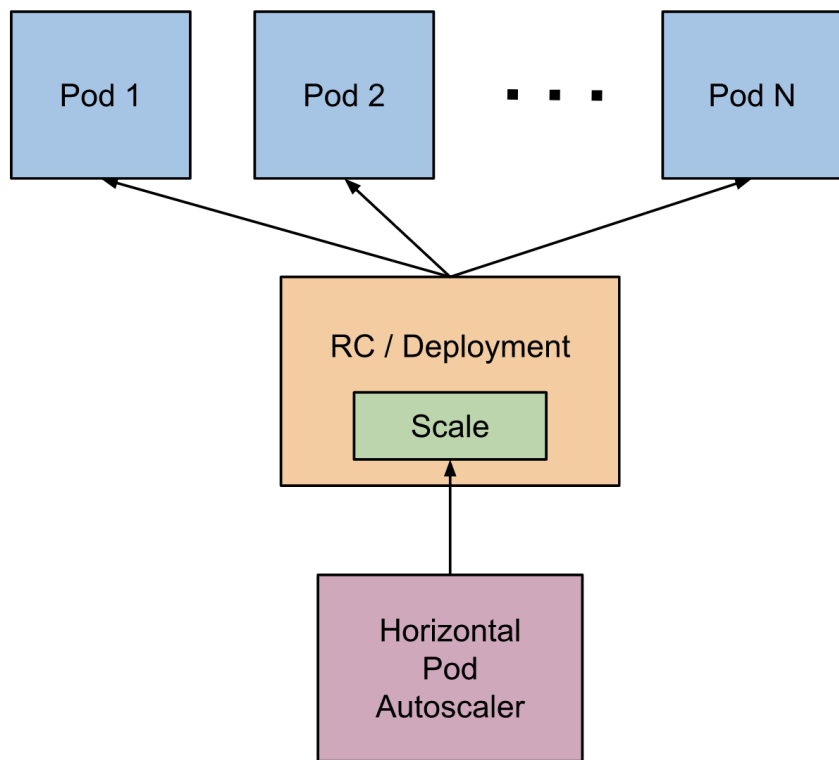
Step 1: 定义PriorityClass

Step 2: 资源对象绑定相应优先级

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
      priorityClassName: high-priority
```

水平自动伸缩HPA

使用 `kubectl scale` 命令可以手动实现 Pod 的扩缩容功能，但应对线上的各种复杂情况，需要能够做到自动化去感知业务，来自动进行扩缩容，Kubernetes提供了Horizontal Pod Autoscaling来应对这种场景。



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: "250m"
```

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 100Mi
    # - type: Pods
    # pods:
    #   metric:
    #     name: packets_per_second
    #   target:
    #     type: AverageValue
    #     averageValue: 100
```

应用容器安全性

容器安全性有诸多考量因素，从应用角度来看，至少需要考虑RBAC授权、密钥凭据管理、SecurityContext等，在做应用容器化时，这些也是应用开发人员所要关注的关键点。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
  namespace: default
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: jenkins
rules:
- apiGroups: ["extensions", "apps"]
  resources: ["deployments", "ingresses"]
  verbs: ["create", "delete", "get", "list", "watch", "patch", "update"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["create", "delete", "get", "list", "watch", "patch", "update"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
- apiGroups: [""]
  resources: ["pods/log", "events"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
```

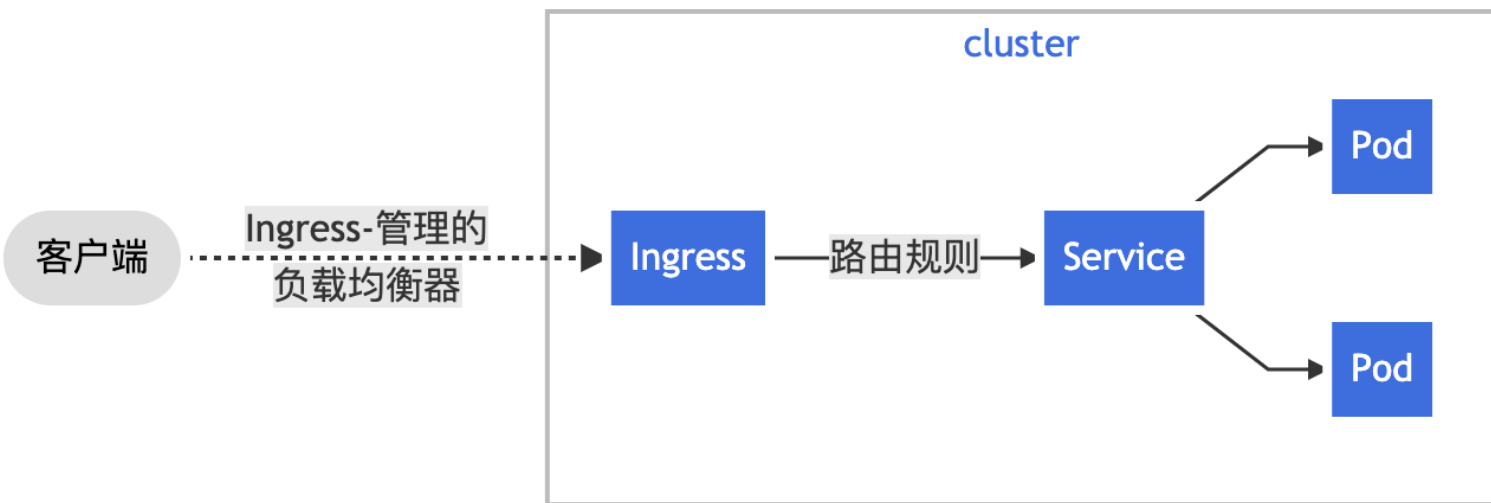
```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: jenkins
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins
  namespace: default
```

```
env:
- name: WORDPRESS_DB_HOST
  value: wordpress-mysql:3306
- name: WORDPRESS_DB_USER
  value: wordpress
- name: WORDPRESS_DB_PASSWORD
  valueFrom:
    secretKeyRef:
      name: wordpress-db-pwd
      key: dbpwd
```

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-pod-demo
spec:
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: ["sh", "-c", "sleep 60m"]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /pod/demo
    securityContext:
      allowPrivilegeEscalation: false
```

通过Ingress对外发布应用

通常情况下，Service和Pod仅可在集群内部网络中通过IP地址访问。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和HTTPS。Ingress功能实现依赖于Ingress Controller控制器，社区有非常多不同的Ingress Controller实现，比较常用的有Ingress Nginx Controller。

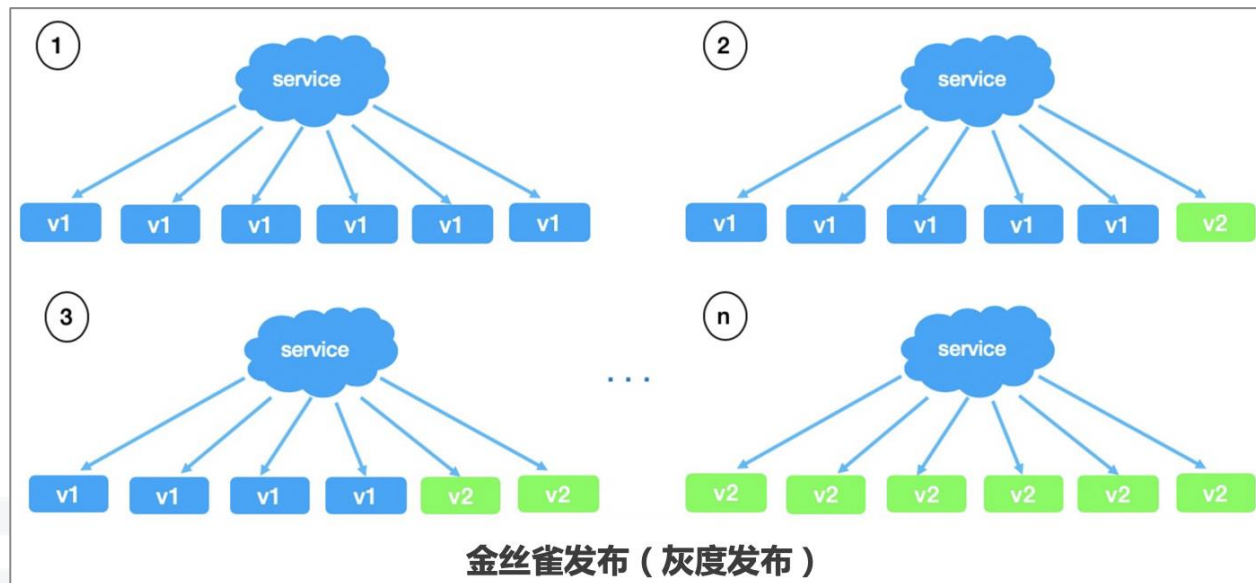
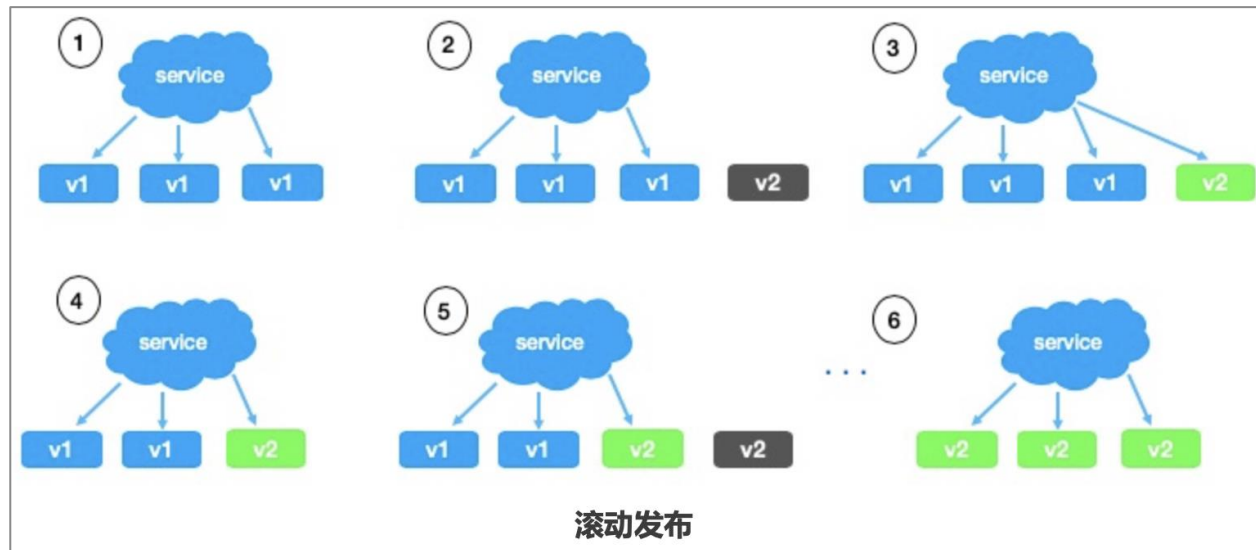
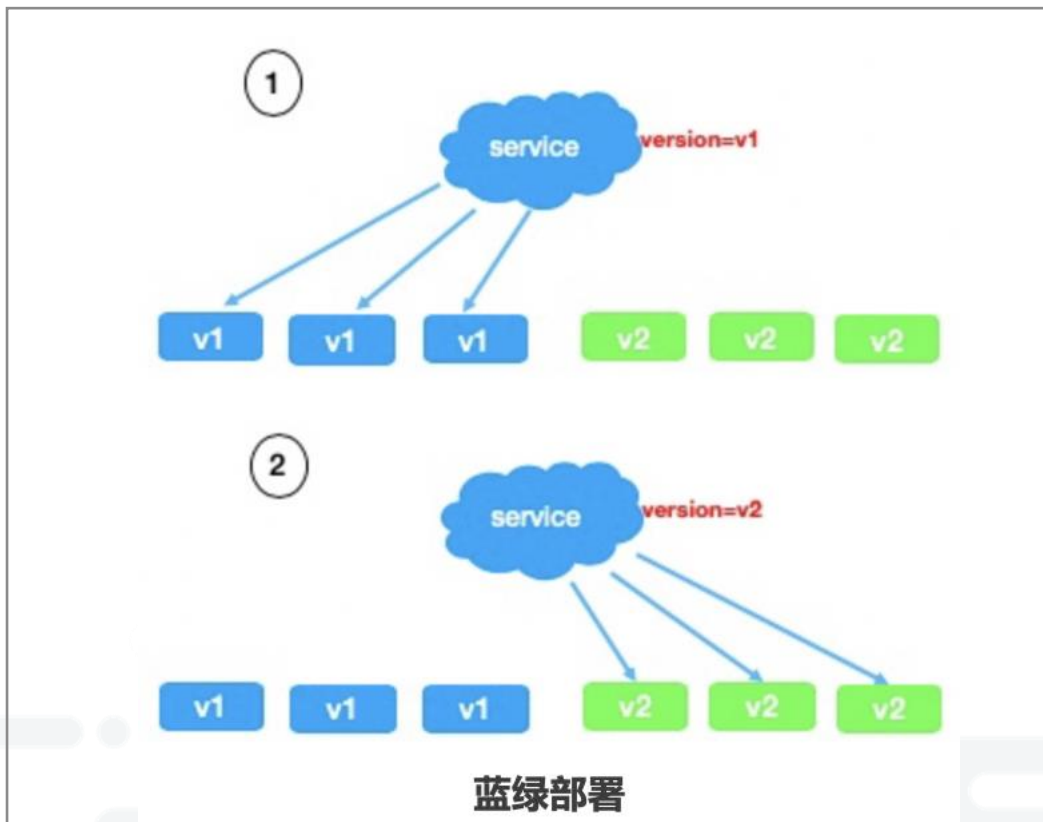


```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: mynginx
  namespace: default
spec:
  rules:
  - host: mynginx.yedward.net
    http:
      paths:
      - backend:
          serviceName: mynginx-svc
          servicePort: 80
```

注：除了通过Ingress方式进行服务对外发布以外，Service还提供了NodePort、LoadBalance两种服务类型，针对不同的应用场景可以选择最适合的方案。

应用发布策略

- 滚动发布：逐个替换，直到所有实例都被替换完成
- 蓝绿部署：蓝/绿版本均部署，负载控制流量切换
- 金丝雀发布（灰度发布）：多个版本均部署，根据路由规则，让指定用户路由到指定的后端

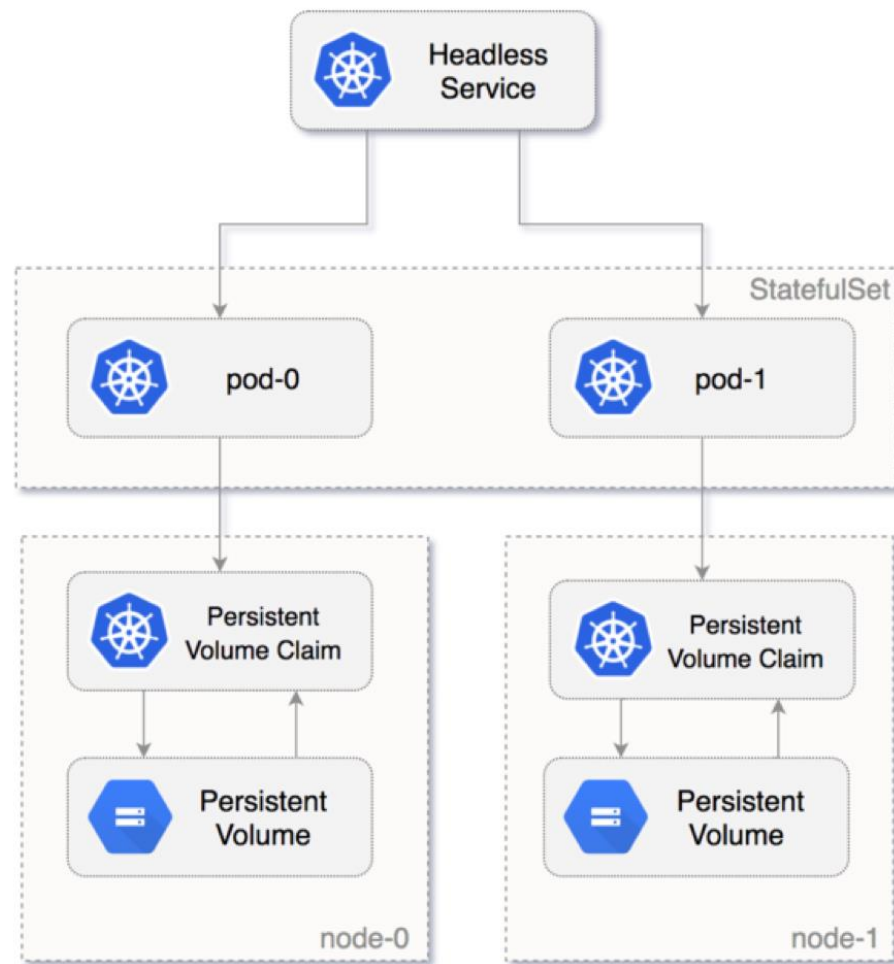


特殊对待有状态应用StatefulSet

对于无状态应用容器化前面有了全面的说明，而有状态应用的容器化部署却相对比较麻烦。Kubernetes提供了StatefulSet资源对象来稳定的管理有状态应用，稳定意味着Pod调度或重调度的整个过程是有持久性的，StatefulSet的目标应用一般具备以下特点：

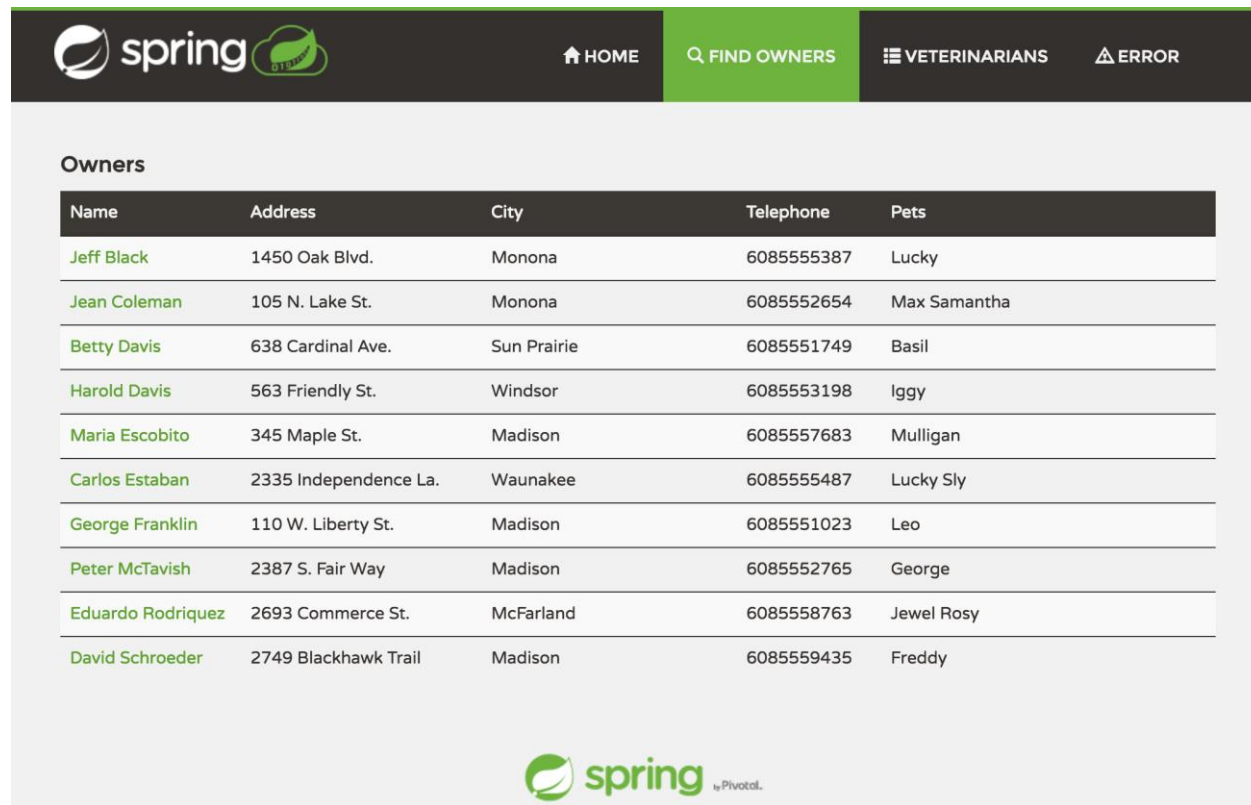
- 稳定的、唯一的网络标识符，即Pod重新调度后PodName和HostName不变，基于Headless Service实现
- 稳定的、持久的存储，即Pod重新调度后还能访问到相同的持久化数据，基于PVC实现
- 有序部署、有序扩展，即Pod是有顺序的，从0到N-1，下一个Pod运行前所有之前的Pod必须都是Running或Ready状态
- 有序收缩、有序删除，即Pod时有序的，从N-1到0，与扩容相反

注：对于一般的比较简单的有状态应用，可以通过StatefulSet方式来部署，但对于复杂的场景，还需要提供自动备份、故障检测、恢复自愈等自动化运维能力时，建议通过Operator方式来部署管理，例如Etcd Operator



实践：容器化部署Spring Petclinic项目

- 项目地址: <https://projects.spring.io/spring-petclinic/>
- 操作文档: <http://note.youdao.com/s/IYSLUjUh>
- 涉及知识点:
 - Dockerfile编写
 - Kubernetes部署文件编写
 - Kubernetes应用发布



The screenshot displays the Spring Petclinic application's 'Owners' page. The page features a dark navigation bar with the Spring logo, a home icon, and links for 'FIND OWNERS', 'VETERINARIANS', and 'ERROR'. Below the navigation bar, the title 'Owners' is centered. A table lists ten pet owners with columns for Name, Address, City, Telephone, and Pets. The table is styled with a dark header and alternating light and dark rows. The Spring logo and 'by Pivotal.' text are visible at the bottom right of the page.

| Name | Address | City | Telephone | Pets |
|-------------------|-----------------------|-------------|------------|--------------|
| Jeff Black | 1450 Oak Blvd. | Monona | 6085555387 | Lucky |
| Jean Coleman | 105 N. Lake St. | Monona | 6085552654 | Max Samantha |
| Betty Davis | 638 Cardinal Ave. | Sun Prairie | 6085551749 | Basil |
| Harold Davis | 563 Friendly St. | Windsor | 6085553198 | Iggy |
| Maria Escobito | 345 Maple St. | Madison | 6085557683 | Mulligan |
| Carlos Estaban | 2335 Independence La. | Waunakee | 6085555487 | Lucky Sly |
| George Franklin | 110 W. Liberty St. | Madison | 6085551023 | Leo |
| Peter McTavish | 2387 S. Fair Way | Madison | 6085552765 | George |
| Eduardo Rodriquez | 2693 Commerce St. | McFarland | 6085558763 | Jewel Rosy |
| David Schroeder | 2749 Blackhawk Trail | Madison | 6085559435 | Freddy |



我们致力于实现计算无处不在
computing everywhere.

We give customers the freedom to compute everywhere they want to - from the data center, to the cloud, to the edge, and beyond.

