阿里云 × CLOUD NATIVE COMPUTING FOUNDATION

云 原 生 技 术 公 开 课

第 04 讲

理解 Pod 与容器设计模式

张磊 阿里巴巴高级技术专家，CNCF 官方大使

关注“ 阿里巴巴云原生 ”公众号
获取第一手技术资料

本节大纲

- 为什么我们需要 Pod?
- Pod 的实现机制
- 详解容器设计模式

1 为什么我们需要 Pod?

先回顾一下容器的基本概念

- 容器的本质是?
  - 一个视图被隔离、资源受限的**进程**
    - 容器里 PID = 1的进程就是应用本身
      - 管理虚拟机 = 管理基础设施；管理容器 = 直接管理应用本身
- 那么 Kubernetes 呢?
  - Kubernetes 就是云时代的**操作系统**!
    - 以此类推，容器镜像其实就是：这个操作系统的**软件安装包**

再来看一个真实操作系统里的例子

- 举例：helloworld 程序
  - helloworld 程序实际上是由一组进程（Linux 里的线程）组成
  - 这 4 个进程共享 helloworld 程序的资源，相互协作，完成 helloworld 程序的工作

```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```

思考

- Kubernetes = 操作系统 (比如：Linux)
- 容器 = 进程 (Linux 线程)
- Pod = ?
  - **进程组** (Linux 线程组)

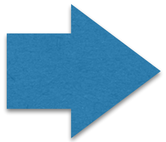
“进程组”

- 举例：
  - helloworld 程序由 4 个进程组成，这些进程之间共享某些文件
- 问题：helloworld 程序如何用容器跑起来呢?
  - 解法一：在一个 Docker 容器中，启动这 4 个进程
    - 疑问：容器 PID = 1 的进程就是应用本身比如 main 进程，那么“谁”来负责管理剩余的 3 个进程?
      - **容器是“单进程”模型!**
      - 除非：
        - 应用进程本身具备“进程管理”能力（这意味着：helloworld 程序需要具备 systemd的能力）
        - 或者，容器的 PID=1进程改成 systemd
          - 这会导致：管理容器 = 管理 systemd != 直接管理应用本身

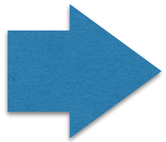
```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```

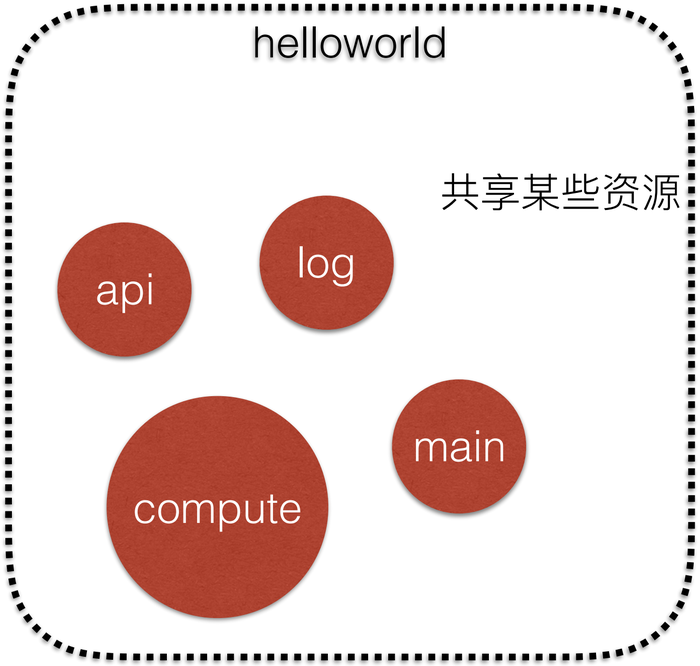
Pod = “进程组”

```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```



```
apiVersion: v1
kind: Pod
metadata:
  name: helloworld
spec:
  containers:
  - name: api
    image: api
    ports:
    - containerPort: 80
  - name: main
    image: main
  - name: log
    image: log
    volumeMounts:
    - name: log-storage
  - name: compute
    image: compute
    volumeMounts:
    - name: data-storage
```





Pod：一个逻辑单位，多个容器的组合
Kubernetes 的原子调度单位

来自 Google Borg 的思考

- Google 的工程师们发现，在 Borg 项目部署的应用，往往都存在着类似于“进程和进程组”的关系。更具体地说，就是这些应用之间有着密切的协作关系，使得它们必须部署在同一台机器上并且共享某些信息

- Large-scale cluster management at Google with Borg, EuroSys’15

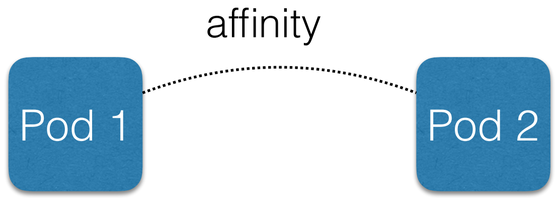
为什么 Pod 必须是原子调度单位?

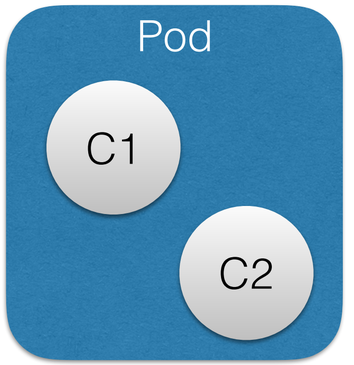
- 举例：两个容器紧密协作
  - App：业务容器，写日志文件
  - LogCollector: 转发日志文件到 ElasticSearch 中
- 内存要求:
  - App: 1G
  - LogCollector: 0.5G
- 当前可用内存:
  - Node_A: 1.25G
  - Node_B: 2G
- 如果 App 先被调度到了 Node_A 上，会怎么样?

- Task co-scheduling 问题
  - Mesos：资源囤积（resource hoarding）：
    - 所有设置了Affinity约束的任务都达到时，才开始统一进行调度
    - 调度效率损失和死锁
  - Google Omega：乐观调度处理冲突：
    - 先不管这些冲突，而是通过精心设计的回滚机制在出现了冲突之后解决问题。
    - 复杂
  - Kubernetes：Pod

再次理解 Pod

- **亲密关系 - 调度解决**
  - 两个应用需要运行在同一台宿主机上
- **超亲密关系 - Pod 解决**
  - 会发生直接的文件交换
  - 使用localhost或者Socket文件进行本地通信
  - 会发生非常频繁的RPC调用
  - 会共享某些Linux Namespace（比如，一个容器要加入另一个容器的Network Namespace）
  - …





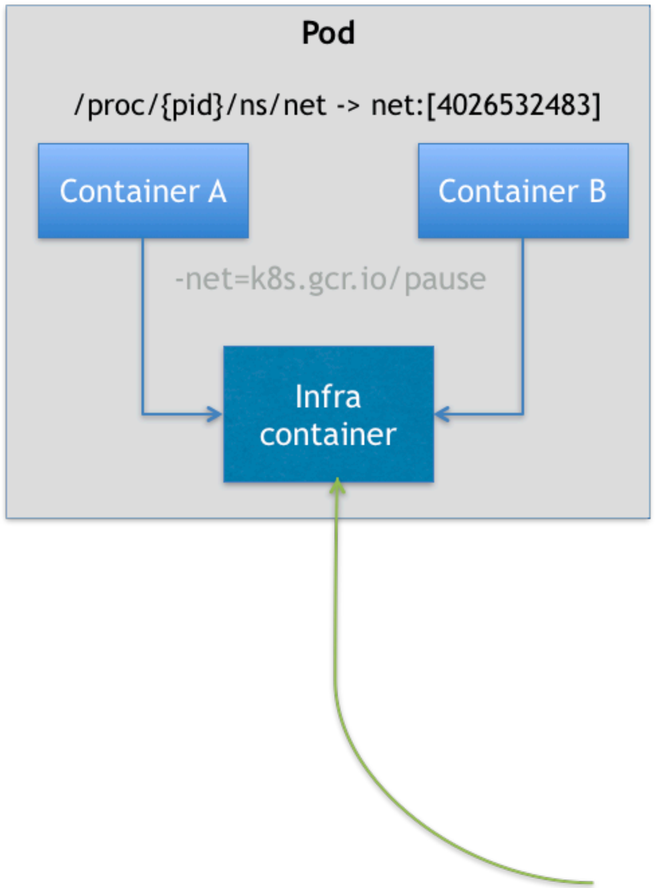
2 Pod 的实现机制

Pod 要解决的问题

- 如何让一个 Pod 里的多个容器之间最高效的共享某些资源和数据?
  - 容器之间原本是被 Linux Namespace 和 cgroups 隔离开的

1. 共享网络

- 容器 A 和 B
  - 通过 Infra Container 的方式共享同一个 Network Namespace：
    - 镜像：k8s.gcr.io/pause；汇编语言编写的、永远处于“暂停”；大小100~200 KB
  - 直接使用localhost进行通信
  - 看到的网络设备跟Infra容器看到的完全一样
  - 一个Pod只有一个IP地址，也就是这个Pod的Network Namespace对应的IP地址
    - 所有网络资源，都是一个Pod一份，并且被该Pod中的所有容器共享
  - 整个 Pod的生命周期跟Infra容器一致，而与容器A和B无关



2. 共享存储

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    hostPath:
      path: /data
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the debian container > /
pod-data/index.html"]
```

- shared-data 对应在宿主机上的目录会被同时绑定挂载进了上述两个容器当中

3 详解容器设计模式

举例： WAR 包 + Tomcat 的容器化

- 方法一：把WAR包和Tomcat打包进一个镜像
  - 无论是WAR 包和 Tomcat 更新都需要重新制作镜像
- 方法二：镜像里只打包Tomcat。使用数据卷（hostPath）从宿主机上将WAR包挂载进Tomcat容器
  - 需要维护一套分布式存储系统
- 有没有更通用的方法?

InitContainer

- Init Container 会比spec.containers 定义的用户容器先启动，并且严格按照定义顺序依次执行
- /app 是一个Volume
- Tomcat容器，同样声明了挂载该 Volume 到自己的webapps目录下
- 故当 Tomcat容器启动时，它的 webapps目录下就一定会存在 sample.war文件

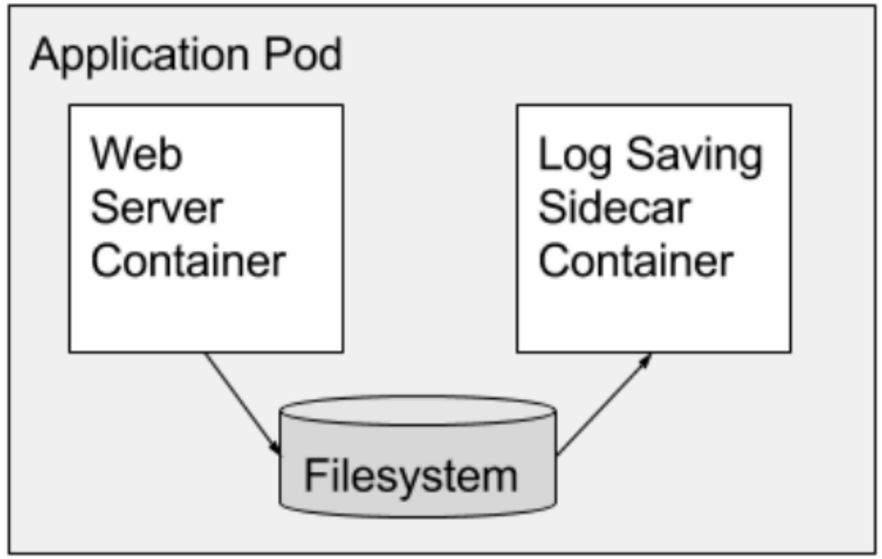
```
apiVersion: v1
kind: Pod
metadata:
  name: javaweb-2
spec:
  initContainers:
  - image: resouer/sample:v2
    name: war
    command: ["cp", "/sample.war", "/app"]
    volumeMounts:
    - mountPath: /app
      name: app-volume
  containers:
  - image: resouer/tomcat:7.0
    name: tomcat
    command: ["sh","-c","/root/apache-tomcat-7.0.42-v2/
bin/start.sh"]
    volumeMounts:
    - mountPath: /root/apache-tomcat-7.0.42-v2/webapps
      name: app-volume
    ports:
    - containerPort: 8080
      hostPort: 8001
  volumes:
  - name: app-volume
    emptyDir: {}
```

容器设计模式：Sidecar

- 通过在 Pod 里定义专门容器，来执行主业务容器需要的辅助工作
  - 比如：
    - 原本需要 SSH 进去执行的脚本
    - 日志收集
    - Debug 应用
    - 应用监控
    - …
- 优势：
  - 将辅助功能同主业务容器解耦，实现独立发布和能力重用

Sidecar：应用与日志收集

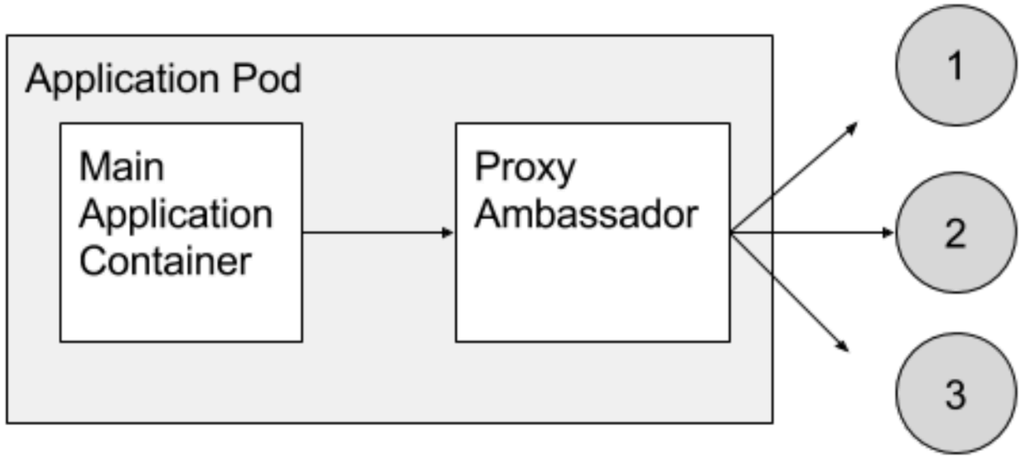
- 业务容器将日志写在 Volume 里
- 日志容器共享该 Volume 从而将日志转发到远程存储当中
  - Fluentd 等



图片来源于论文：https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_burns.pdf

Sidecar：代理容器

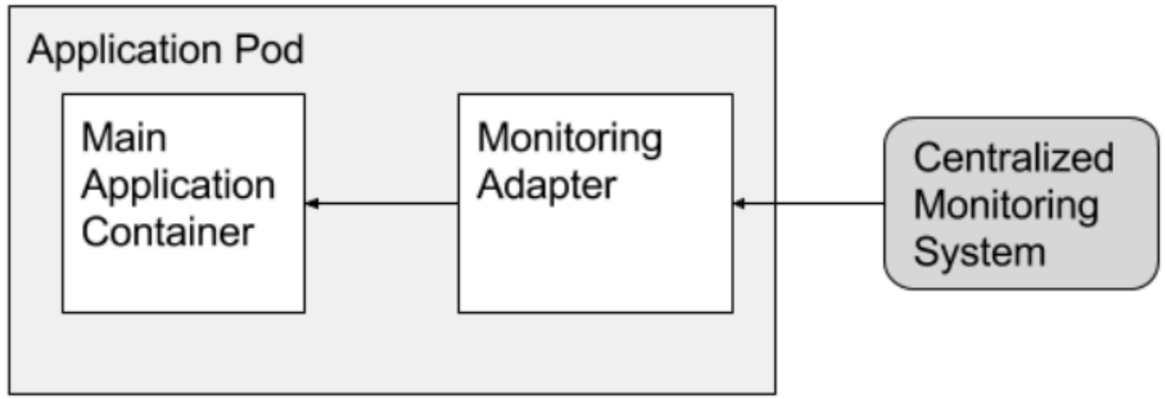
- 代理容器对业务容器屏蔽被代理的服务集群，简化业务代码的实现逻辑
  - 提示：
    - 容器之间通过 localhost 直接通信
    - 代理容器的代码可以被全公司重用



图片来源于论文：https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_burns.pdf

Sidecar：适配器容器

- 适配器容器将业务容器暴露出来的接口转换为另一种格式
- 举例：
  - 业务容器暴露出来的监控接口是 /metrics
  - Monitoring Adapter 将其转换为 /healthz 以适配新的监控系统
  - 提示：
    - 容器之间通过 localhost 直接通信
    - 代理容器的代码可以被全公司重用



图片来源于论文：https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_burns.pdf

本节总结

- Pod 是 Kubernetes 项目里实现“容器设计模式”的核心机制
- “容器设计模式”是 Google Borg 的大规模容器集群管理最佳实践之一
  - 也是 Kubernetes 进行复杂应用编排的基础依赖之一
- 所有“设计模式”的本质都是：解耦和重用

关注“ 阿里巴巴云原生公众号 ”
获取第一手技术资料