# 13 | 为什么我们需要Pod?

# 13 | 为什么我们需要Pod?

张磊 2018-09-21

18:42

讲述：张磊 大小：17.13M

你好，我是张磊。今天我和你分享的主题是：为什么我们需要 Pod。

在前面的文章中，我详细介绍了在 Kubernetes 里部署一个应用的过程。在这些讲解中，我提到了这样一个知识点：Pod，是 Kubernetes 项目中最小的 API 对象。如果换一个更专业的说法，我们可以这样描述：Pod，是 Kubernetes 项目的原子调度单位。

不过，我相信你在学习和使用 Kubernetes 项目的过程中，已经不止一次地想要问这样一个问题：为什么我们会需要 Pod?

是啊，我们在前面已经花了很多精力去解读 Linux 容器的原理、分析了 Docker 容器的本质，终于，“Namespace 做隔离，Cgroups 做限制，rootfs 做文件系

统”这样的“三句箴言”可以朗朗上口了，**为什么 Kubernetes 项目又突然搞出一个 Pod 来呢?**

要回答这个问题，我们还是要一起回忆一下我曾经反复强调的一个问题：容器的本质到底是什么?

你现在应该可以不假思索地回答出来：容器的本质是进程。

没错。容器，就是未来云计算系统中的进程；容器镜像就是这个系统里的“.exe”安装包。那么 Kubernetes 呢?

你应该也能立刻回答上来：Kubernetes 就是操作系统!

非常正确。

现在，就让我们登录到一台 Linux 机器里，执行一条如下所示的命令：

```
$ pstree -g
```

□复制代码

这条命令的作用，是展示当前系统中正在运行的进程的树状结构。它的返回结果如下所示：

```
systemd(1)-+-accounts-daemon(1984)-+-{gdbus}(1984)
| `-{gmain}(1984)
|-acpid(2044)
...
|-lxcfs(1936)-+-{lxcfs}(1936)
| `-{lxcfs}(1936)
|-mdadm(2135)
|-ntpd(2358)
|-polkitd(2128)-+-{gdbus}(2128)
| `-{gmain}(2128)
|-rsyslogd(1632)-+-{in:imklog}(1632)
| |-{in:imuxsock) S 1(1632)
| `-{rs:main Q:Reg}(1632)
|-snapd(1942)-+-{snapd}(1942)
| |-{snapd}(1942)
| |-{snapd}(1942)
```

```
| |-{snapd}(1942)
| |-{snapd}(1942)
```

□复制代码

不难发现，在一个真正的操作系统里，进程并不是“孤苦伶仃”地独自运行的，而是以进程组的方式，“有原则地”组织在一起。比如，这里有一个叫作 rsyslogd 的程序，它负责的是 Linux 操作系统里的日志处理。可以看到，rsyslogd 的主程序 main，和它要用到的内核日志模块 imklog 等，同属于 1632 进程组。这些进程相互协作，共同完成 rsyslogd 程序的职责。

> *注意：我在本篇中提到的“进程”，比如，rsyslogd 对应的 imklog，imuxsock 和 main，严格意义上来说，其实是 Linux 操作系统语境下的“线程”。这些线程，或者说，轻量级进程之间，可以共享文件、信号、数据内存、甚至部分代码，从而紧密协作共同完成一个程序的职责。所以同理，我提到的“进程组”，对应的也是 Linux 操作系统语境下的“线程组”。这种命名关系与实际情况的不一致，是 Linux 发展历史中的一个遗留问题。对这个话题感兴趣的同学，可以阅读这篇技术文章来了解一下。*

而 Kubernetes 项目所做的，其实就是将“进程组”的概念映射到了容器技术中，并使其成为了这个云计算“操作系统”里的“一等公民”。

Kubernetes 项目之所以要这么做的原因，我在前面介绍 Kubernetes 和 Borg 的关系时曾经提到过：在 Borg 项目的开发和实践过程中，Google 公司的工程师们发现，他们部署的应用，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。

而如果事先没有“组”的概念，像这样的运维关系就会非常难以处理。

我还是以前面的 rsyslogd 为例子。已知 rsyslogd 由三个进程组成：一个 imklog 模块，一个 imuxsock 模块，一个 rsyslogd 自己的 main 函数主进程。这三个进程一定要运行在同一台机器上，否则，它们之间基于 Socket 的通信和文件交换，都会出现问题。

现在，我要把 rsyslogd 这个应用给容器化，由于受限于容器的“单进程模型”，这三个模块必须被分别制作成三个不同的容器。而在这三个容器运行的时候，它们设置的内存配额都是 1 GB。

> *再次强调一下：容器的“单进程模型”，并不是指容器里只能运行“一个”进程，而是指容器没有管理多个进程的能力。这是因为容器里 PID=1 的进程就是应用本身，其他的进程都是这个 PID=1 进程的子进程。可是，用户编写的应用，并不能够像正常操作系统里的 init 进程或者 systemd 那样拥有进程管理的功能。比如，你的应用是一个 Java Web 程序*

> *（PID=1），然后你执行 docker exec 在后台启动了一个 Nginx 进程（PID=3）。可是，当这个 Nginx 进程异常退出的时候，你该怎么知道呢？这个进程退出后的垃圾收集工作，又应该由谁去做呢？*

假设我们的 Kubernetes 集群上有两个节点：node-1 上有 3 GB 可用内存，node-2 有 2.5 GB 可用内存。

这时，假设我要用 Docker Swarm 来运行这个 rsyslogd 程序。为了能够让这三个容器都运行在同一台机器上，我就必须在另外两个容器上设置一个 affinity=main（与 main 容器有亲密性）的约束，即：它们俩必须和 main 容器运行在同一台机器上。

然后，我顺序执行：“docker run main”“docker run imklog”和“docker run imuxsock”，创建这三个容器。

这样，这三个容器都会进入 Swarm 的待调度队列。然后，main 容器和 imklog 容器都先后出队并被调度到了 node-2 上（这个情况是完全有可能的）。

可是，当 imuxsock 容器出队开始被调度时，Swarm 就有点懵了：node-2 上的可用资源只有 0.5 GB 了，并不足以运行 imuxsock 容器；可是，根据 affinity=main 的约束，imuxsock 容器又只能运行在 node-2 上。

这就是一个典型的成组调度（gang scheduling）没有被妥善处理的例子。

在工业界和学术界，关于这个问题的讨论可谓旷日持久，也产生了很多可供选择的解决方案。

比如，Mesos 中就有一个资源囤积（resource hoarding）的机制，会在所有设置了 Affinity 约束的任务都达到时，才开始对它们统一进行调度。而在 Google Omega 论文中，则提出了使用乐观调度处理冲突的方法，即：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。

可是这些方法都谈不上完美。资源囤积带来了不可避免的调度效率损失和死锁的可能性；而乐观调度的复杂程度，则不是常规技术团队所能驾驭的。

但是，到了 Kubernetes 项目里，这样的问题就迎刃而解了：Pod 是 Kubernetes 里的原子调度单位。这就意味着，Kubernetes 项目的调度器，是统一按照 Pod 而非容器的资源需求进行计算的。

所以，像 imklog、imuxsock 和 main 函数主进程这样的三个容器，正是一个典型的由三个容器组成的 Pod。Kubernetes 项目在调度时，自然就会去选择可用内存等于 3 GB 的 node-1 节点进行绑定，而根本不会考虑 node-2。

像这样容器间的紧密协作，我们可以称为“超亲密关系”。这些具有“超亲密关系”容器的典型特征包括但不限于：互相之间会发生直接的文件交换、使用 localhost 或者 Socket 文件进行本地通信、会发生非常频繁的远程调用、需要共享某些 Linux Namespace（比如，一个容器要加入另一个容器的 Network Namespace）等等。

这也就意味着，并不是所有有“关系”的容器都属于同一个 Pod。比如，PHP 应用容器和 MySQL 虽然会发生访问关系，但并没有必要、也不应该部署在同一台机器上，它们更适合做成两个 Pod。

不过，相信此时你可能会有**第二个疑问：**

对于初学者来说，一般都是先学会了用 Docker 这种单容器的工具，才会开始接触 Pod。

而如果 Pod 的设计只是出于调度上的考虑，那么 Kubernetes 项目似乎完全没有必要非得把 Pod 作为“一等公民”吧？这不是故意增加用户的学习门槛吗？

没错，如果只是处理“超亲密关系”这样的调度问题，有 Borg 和 Omega 论文珠玉在前，Kubernetes 项目肯定可以在调度器层面给它解决掉。

不过，Pod 在 Kubernetes 项目里还有更重要的意义，那就是：**容器设计模式。**

为了理解这一层含义，我就必须先给你介绍一下Pod 的实现原理。

**首先，关于 Pod 最重要的一个事实是：它只是一个逻辑概念。**

也就是说，Kubernetes 真正处理的，还是宿主机操作系统上 Linux 容器的 Namespace 和 Cgroups，而并不存在一个所谓的 Pod 的边界或者隔离环境。

那么，Pod 又是怎么被“创建”出来的呢？

答案是：Pod，其实是一组共享了某些资源的容器。

具体的说：**Pod 里的所有容器，共享的是同一个 Network Namespace，并且可以声明共享同一个 Volume。**

那这么来看的话，一个有 A、B 两个容器的 Pod，不就是等同于一个容器（容器 A）共享另外一个容器（容器 B）的网络和 Volume 的玩儿法么？

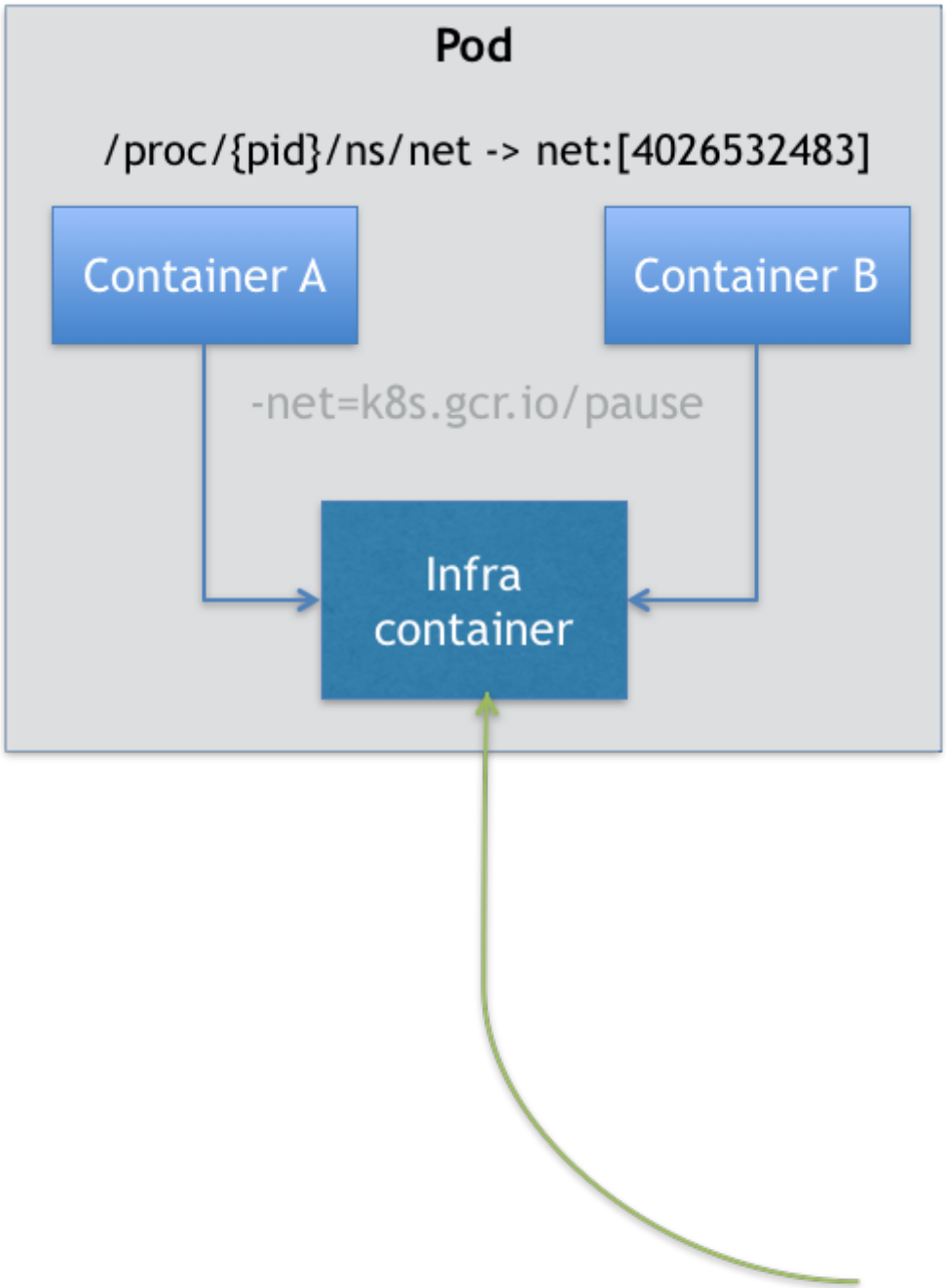
这好像通过 docker run --net --volumes-from 这样的命令就能实现嘛，比如：

```
$ docker run --net=B --volumes-from=B --name=A image-A ...
```

□复制代码

但是，你有没有考虑过，如果真这样做的话，容器 B 就必须比容器 A 先启动，这样一个 Pod 里的多个容器就不是对等关系，而是拓扑关系了。

所以，在 Kubernetes 项目里，Pod 的实现需要使用一个中间容器，这个容器叫作 Infra 容器。在这个 Pod 中，Infra 容器永远都是第一个被创建的容器，而其他用户定义的容器，则通过 Join Network Namespace 的方式，与 Infra 容器关联在一起。这样的组织关系，可以用下面这样一个示意图来表达：




如上图所示，这个 Pod 里有两个用户容器 A 和 B，还有一个 Infra 容器。很容易理解，在 Kubernetes 项目里，Infra 容器一定要占用极少的资源，所以它使用的是一个非常特殊的镜像，叫作：k8s.gcr.io/pause。这个镜像是一个用汇编语言编写的、永远处于“暂停”状态的容器，解压后的大小也只有 100~200 KB 左右。

而在 Infra 容器“Hold 住”Network Namespace 后，用户容器就可以加入到 Infra 容器的 Network Namespace 当中了。所以，如果你查看这些容器在宿主机上的 Namespace 文件（这个 Namespace 文件的路径，我已经在前面的内容中介绍过），它们指向的值一定是完全一样的。

这也就意味着，对于 Pod 里的容器 A 和容器 B 来说：

它们可以直接使用 localhost 进行通信；
它们看到的网络设备跟 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；
当然，其他的所有网络资源，都是一个 Pod 一份，并且被该 Pod 中的所有容器共享；
Pod 的生命周期只跟 Infra 容器一致，而与容器 A 和 B 无关。

而对于同一个 Pod 里面的所有用户容器来说，它们的进出流量，也可以认为都是通过 Infra 容器完成的。这一点很重要，因为**将来如果你要为 Kubernetes 开发一个网络插件时，应该重点考虑的是如何配置这个 Pod 的 Network Namespace，而不是每一个用户容器如何使用你的网络配置，这是没有意义的。**

这就意味着，如果你的网络插件需要在容器里安装某些包或者配置才能完成的话，是不可取的：Infra 容器镜像的 rootfs 里几乎什么都没有，没有你随意发挥的空间。当然，这同时也意味着你的网络插件完全不必关心用户容器的启动与否，而只需要关注如何配置 Pod，也就是 Infra 容器的 Network Namespace 即可。

有了这个设计之后，共享 Volume 就简单多了：Kubernetes 项目只要把所有 Volume 的定义都设计在 Pod 层级即可。

这样，一个 Volume 对应的宿主机目录对于 Pod 来说就只有一个，Pod 里的容器只要声明挂载这个 Volume，就一定可以共享这个 Volume 对应的宿主机目录。比如下面这个例子：

```
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"]
```

□复制代码

在这个例子中，debian-container 和 nginx-container 都声明挂载了 shared-data 这个 Volume。而 shared-data 是 hostPath 类型。所以，它对应在宿主机上的目录就是：/data。而这个目录，其实就被同时绑定挂载进了上述两个容器当中。

这就是为什么，nginx-container 可以从它的 /usr/share/nginx/html 目录中，读取到 debian-container 生成的 index.html 文件的原因。

明白了 Pod 的实现原理后，我们再来讨论“容器设计模式”，就容易多了。

Pod 这种“超亲密关系”容器的设计思想，实际上就是希望，当用户想在一个容器里跑多个功能并不相关的应用时，应该优先考虑它们是不是更应该被描述成一个 Pod 里的多个容器。

为了能够掌握这种思考方式，你就应该尽量尝试使用它来描述一些用单个容器难以解决的问题。

**第一个最典型的例子是：WAR 包与 Web 服务器。**

我们现在有一个 Java Web 应用的 WAR 包，它需要被放在 Tomcat 的 webapps 目录下运行起来。

假如，你现在只能用 Docker 来做这件事情，那该如何处理这个组合关系呢？

> 一种方法是，把 WAR 包直接放在 Tomcat 镜像的 webapps 目录下，做成一个新的镜像运行起来。可是，这时候，如果你要更新 WAR 包的内容，或者要升级 Tomcat 镜像，就要重新制作一个新的发布镜像，非常麻烦。
> 另一种方法是，你压根儿不管 WAR 包，永远只发布一个 Tomcat 容器。不过，这个容器的 webapps 目录，就必须声明一个 hostPath 类型的 Volume，从而把宿主机上的 WAR 包挂载进 Tomcat 容器当中运行起来。不过，这样你就必须要解决一个问题，即：如何让每一台宿主机，都预先准备好这个存储有 WAR 包的目录呢？这样来看，你只能独立维护一套分布式存储系统了。

实际上，有了 Pod 之后，这样的问题就很容易解决了。我们可以把 WAR 包和 Tomcat 分别做成镜像，然后把它们作为一个 Pod 里的两个容器“组合”在一起。这个 Pod 的配置文件如下所示：

```
apiVersion: v1
kind: Pod
metadata:
name: javaweb-2
spec:
initContainers:
- image: geektime/sample:v2
name: war
command: ["cp", "/sample.war", "/app"]
volumeMounts:
- mountPath: /app
name: app-volume
containers:
- image: geektime/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: {}
```

复制代码

在这个 Pod 中，我们定义了两个容器，第一个容器使用的镜像是 geektime/sample:v2，这个镜像里只有一个 WAR 包（sample.war）放在根目录下。而第二个容器则使用的是一个标准的 Tomcat 镜像。

不过，你可能已经注意到，WAR 包容器的类型不再是一个普通容器，而是一个 Init Container 类型的容器。

在 Pod 中，所有 Init Container 定义的容器，都会比 spec.containers 定义的用户容器先启动。并且，Init Container 容器会按顺序逐一启动，而直到它们都启动并且退出了，用户容器才会启动。

所以，这个 Init Container 类型的 WAR 包容器启动后，我执行了一句"cp /sample.war /app"，把应用的 WAR 包拷贝到 /app 目录下，然后退出。

而后这个 /app 目录，就挂载了一个名叫 app-volume 的 Volume。

接下来就很关键了。Tomcat 容器，同样声明了挂载 app-volume 到自己的 webapps 目录下。

所以，等 Tomcat 容器启动时，它的 webapps 目录下就一定会存在 sample.war 文件：这个文件正是 WAR 包容器启动时拷贝到这个 Volume 里面的，而这个 Volume 是被这两个容器共享的。

像这样，我们就用一种“组合”方式，解决了 WAR 包与 Tomcat 容器之间耦合关系的问题。

实际上，这个所谓的“组合”操作，正是容器设计模式里最常用的一种模式，它的名字叫：sidecar。

顾名思义，sidecar 指的就是我们可以在一个 Pod 中，启动一个辅助容器，来完成一些独立于主进程（主容器）之外的工作。

比如，在我们的这个应用 Pod 中，Tomcat 容器是我们要使用的主容器，而 WAR 包容器的存在，只是为了给它提供一个 WAR 包而已。所以，我们用 Init

Container 的方式优先运行 WAR 包容器，扮演了一个 sidecar 的角色。

**第二个例子，则是容器的日志收集。**

比如，我现在有一个应用，需要不断地把日志文件输出到容器的 /var/log 目录中。

这时，我就可以把一个 Pod 里的 Volume 挂载到应用容器的 /var/log 目录上。

然后，我在这个 Pod 里同时运行一个 sidecar 容器，它也声明挂载同一个 Volume 到自己的 /var/log 目录上。

这样，接下来 sidecar 容器就只需要做一件事儿，那就是不断地从自己的 /var/log 目录里读取日志文件，转发到 MongoDB 或者 Elasticsearch 中存储起来。这样，一个最基本的日志收集工作就完成了。

跟第一个例子一样，这个例子中的 sidecar 的主要工作也是使用共享的 Volume 来完成对文件的操作。

但不要忘记，Pod 的另一个重要特性是，它的所有容器都共享同一个 Network Namespace。这就使得很多与 Pod 网络相关的配置和管理，也都可以交给 sidecar 完成，而完全无须干涉用户容器。这里最典型的例子莫过于 Istio 这个微服务治理项目了。

Istio 项目使用 sidecar 容器完成微服务治理的原理，我在后面很快会讲解到。

> *备注：Kubernetes 社区曾经把“容器设计模式”这个理论，整理成了一篇小论文，你可以点击链接浏览。*

# 总结

在本篇文章中我重点分享了 Kubernetes 项目中 Pod 的实现原理。

Pod 是 Kubernetes 项目与其他单容器项目相比最大的不同，也是一位容器技术初学者需要面对的第一个与常规认知不一致的知识点。

事实上，直到现在，仍有很多人把容器跟虚拟机相提并论，他们把容器当做性能更好的虚拟机，喜欢讨论如何把应用从虚拟机无缝地迁移到容器中。

但实际上，无论是从具体的实现原理，还是从使用方法、特性、功能等方面，容器与虚拟机几乎没有任何相似的地方；也不存在一种普遍的方法，能够把虚拟机里的应用无缝迁移到容器中。因为，容器的性能优势，必然伴随着相应缺陷，即：它不能像虚拟机那样，完全模拟本地物理机环境中的部署方法。

所以，这个“上云”工作的完成，最终还是要靠深入理解容器的本质，即：进程。

实际上，一个运行在虚拟机里的应用，哪怕再简单，也是被管理在 systemd 或者 supervisord 之下的**一组进程，而不是一个进程**。这跟本地物理机上应用的运行方式其实是一样的。这也是为什么，从物理机到虚拟机之间的应用迁移，往往并不困难。

可是对于容器来说，一个容器永远只能管理一个进程。更确切地说，一个容器，就是一个进程。这是容器技术的“天性”，不可能被修改。所以，将一个原本运行在虚拟机里的应用，“无缝迁移”到容器中的想法，实际上跟容器的本质是相悖的。

这也是当初 Swarm 项目无法成长起来的重要原因之一：一旦到了真正的生产环境上，Swarm 这种单容器的工作方式，就难以描述真实世界里复杂的应用架构了。

所以，你现在可以这么理解 Pod 的本质：

> *Pod，实际上是在扮演传统基础设施里“虚拟机”的角色；而容器，则是这个虚拟机里运行的用户程序。*

所以下一次，当你需要把一个运行在虚拟机里的应用迁移到 Docker 容器中时，一定要仔细分析到底有哪些进程（组件）运行在这个虚拟机里。

然后，你就可以把整个虚拟机想象成为一个 Pod，把这些进程分别做成容器镜像，把有顺序关系的容器，定义为 Init Container。这才是更加合理的、松耦合的容器编排诀窍，也是从传统应用架构，到“微服务架构”最自然的过渡方式。

> *注意：Pod 这个概念，提供的是一种编排思想，而不是具体的技术方案。所以，如果愿意的话，你完全可以使用虚拟机来作为 Pod 的实现，然后把用户容器都运行在这个虚拟机里。比如，Mirantis 公司的virtlet 项目就在干这个事情。甚至，你可以去实现一个带有 Init 进程的容器项目，来模拟传统应用的运行方式。这些工作，在 Kubernetes 中都是非常轻松的，也是我们后面讲解 CRI 时会提到的内容。*

相反的，如果强行把整个应用塞到一个容器里，甚至不惜使用 Docker In Docker 这种在生产环境中后患无穷的解决方案，恐怕最后往往会得不偿失。

# 思考题

除了 Network Namespace 外，Pod 里的容器还可以共享哪些 Namespace 呢？你能说出共享这些 Namesapce 的具体应用场景吗？