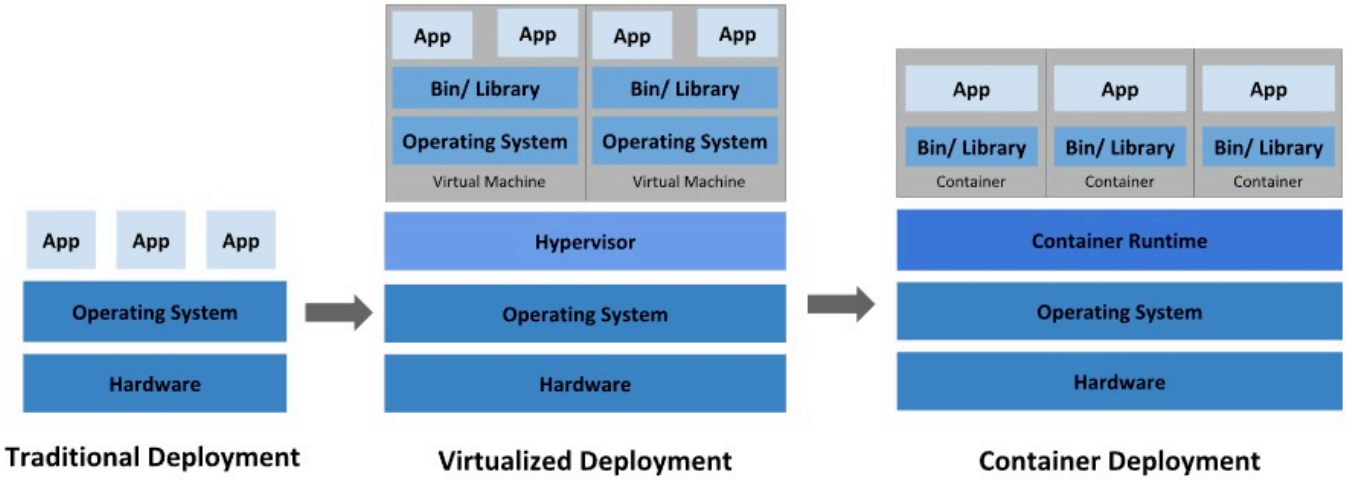
2021-06-15 Richard技术分享

容器化之小鲸鱼的前世今生

应用程序部署发展史



大致来说，在部署应用程序的方式上，我们主要经历了三个时代：

- **传统部署时代**：早期，企业直接将应用程序部署在物理机上。由于物理机上不能为应用程序定义资源使用边界，我们也就很难合理地分配计算资源。例如：如果多个应用程序运行在同一台物理机上，可能发生这样的情况：其中的一个应用程序消耗了大多数的计算资源，导致其他应用程序不能正常运行。应对此问题的一种解决办法是，将每一个应用程序运行在不同的物理机上。然而，这种做法无法大规模实施，因为资源利用率很低，且企业维护更多物理机的成本昂贵。
- **虚拟化部署时代**：针对上述问题，虚拟化技术应运而生。用户可以在单台物理机的CPU上运行多个虚拟机（Virtual Machine）。
    - 虚拟化技术使得应用程序被虚拟机相互分隔开，限制了应用程序之间的非法访问，进而提供了一定程度的安全性；
    - 虚拟化技术提高了物理机的资源利用率，可以更容易地安装或更新应用程序，降低了硬件成本，因此可以更好地规模化实施；
    - 每一个虚拟机可以认为是被虚拟化的物理机之上的一台完整的机器，其中运行了一台机器的所有组件，包括虚拟机自身的操作系统。
- **容器化部署时代**：容器与虚拟机类似，但是降低了隔离层级，共享了操作系统。因此，容器可以认为是轻量级的虚拟机。
    - 与虚拟机相似，每个容器拥有自己的文件系统、CPU、内存、进程空间等；
    - 运行应用程序所需要的资源都被容器包装，并和底层基础架构解耦；
    - 容器化的应用程序可以跨云服务商、跨Linux操作系统发行版进行部署。

容器化越来越流行，主要原因是它带来的诸多好处：

- **敏捷地创建和部署应用程序**：相较于创建虚拟机镜像，创建容器镜像更加容易和快速；
- **持续构建集成**：可以更快更频繁地构建容器镜像、部署容器化的应用程序、并且轻松地回滚应用程序；
- **分离开发和运维的关注点**：在开发构建阶段就完成容器镜像的构建，构建好的镜像可以部署到多种基础设施上。这种做法将开发阶段需要关注的内容包含在如何构建容器镜像的过程中，将部署阶段需要关注的内容聚焦在如何提供基础设施以及如何使用容器镜像的过程中。降低了开发和运维的耦合度；
- **开发、测试、生产不同阶段的环境一致性**：开发阶段在笔记本上运行的容器与测试、生产环境中运行的容器一致；
- **可监控性**：不仅可以查看操作系统级别的资源监控信息，还可以查看应用程序健康状态以及其他信号的监控信息；
- **跨云服务商、跨操作系统发行版的可移植性**：容器可运行在 Ubuntu、RHEL、CoreOS、CentOS等不同的操作系统发行版上，可以运行在私有化部署、Google Kubernetes Engine、AWS、阿里云等不同的云供应商的环境中；

- **以应用程序为中心的管理**：虚拟机时代的考虑的问题是在虚拟硬件上运行一个操作系统，而容器化时代，问题的焦点则是在操作系统的逻辑资源上运行一个应用程序；
- **松耦合、分布式、弹性、无约束的微服务**：应用程序被切分成更小的、独立的微服务，并可以动态部署和管理，而不是一个部署在专属机器上的庞大的单片应用程序；
- **资源隔离**：确保应用程序性能不受干扰；
- **资源利用**：资源高效、高密度利用。

小鲸鱼 - 初出茅庐

2013 年的后端技术领域，已经太久没有出现过令人兴奋的东西了。曾经被人们寄予厚望的云计算技术，也已经从当初虚无缥缈的概念蜕变成了实实在在的虚拟机和账单。而相比于的如日中天 AWS 和盛极一时的 OpenStack，以 Cloud Foundry 为代表的开源 PaaS 项目，却成为了当时云计算技术中的一股清流。

这时，Cloud Foundry 项目已经基本度过了最艰难的概念普及和用户教育阶段，吸引了包括百度、京东、华为、IBM 等一大批国内外技术厂商，开启了以开源 PaaS 为核心构建平台层服务能力的变革。如果你有机会问问当时的云计算从业者们，他们十有八九都会告诉你：PaaS 的时代就要来了！

这个说法其实一点儿没错，如果不是后来一个叫 Docker 的开源项目突然冒出来的话。

事实上，当时还名叫 dotCloud 的 Docker 公司，也是这股 PaaS 热潮中的一份子。只不过相比于 Heroku、Pivotal、Red Hat 等 PaaS 弄潮儿们，dotCloud 公司实在是太微不足道了，而它的主打产品由于跟主流的 Cloud Foundry 社区脱节，长期以来也无人问津。眼看就要被如火如荼的 PaaS 风潮抛弃，dotCloud 公司却做出了这样一个决定：开源自己的容器项目 Docker。

显然，这个决定在当时根本没人在乎。

“容器”这个概念从来就不是什么新鲜的东西，也不是 Docker 公司发明的。即使在当时最热门的 PaaS 项目 Cloud Foundry 中，容器也只是其最底层、最没人关注的那一部分。说到这里，我正好以当时的事实标准 Cloud Foundry 为例，来解说一下 PaaS 技术。

PaaS 项目被大家接纳的一个主要原因，就是它提供了一种名叫“应用托管”的能力。 在当时，虚拟机和云计算已经是比较普遍的技术和服务了，那时主流用户的普遍用法，就是租一批 AWS 或者 OpenStack 的虚拟机，然后像以前管理物理服务器那样，用脚本或者手工的方式在这些机器上部署应用。

当然，这个部署过程难免会碰到云端虚拟机和本地环境不一致的问题，所以当时的云计算服务，比的就是谁能更好地模拟本地服务器环境，能带来更好的“上云”体验。而 PaaS 开源项目的出现，就是当时解决这个问题的一个最佳方案。

举个例子，虚拟机创建好之后，运维人员只需要在这些机器上部署一个 Cloud Foundry 项目，然后开发者只要执行一条命令就能把本地的应用部署到云上，这条命令就是：

```
$ cf push " 我的应用 "
```

是不是很神奇?

事实上，**像 Cloud Foundry 这样的 PaaS 项目，最核心的组件就是一套应用的打包和分发机制。** Cloud Foundry 为每种主流编程语言都定义了一种打包格式，而“cf push”的作用，基本上等同于用户把应用的可执行文件和启动脚本打进一个压缩包内，上传到云上 Cloud Foundry 的存储中。接着，Cloud Foundry 会通过调度器选择一个可以运行这个应用的虚拟机，然后通知这个机器上的 Agent 把应用压缩包下载下来启动。

这时候关键来了，由于需要在一个虚拟机上启动很多个来自不同用户的应用，Cloud Foundry 会调用操作系统的 Cgroups 和 Namespace 机制为每一个应用单独创建一个称作“沙盒”的隔离环境，然后在“沙盒”中启动这些应用进程。这样，就实现了把多个用户的应用互不干涉地在虚拟机里批量地、自动地运行起来的目的。

这，正是 PaaS 项目最核心的能力。 而这些 Cloud Foundry 用来运行应用的隔离环境，或者说“沙盒”，就是所谓的“容器”。

而 Docker 项目，实际上跟 Cloud Foundry 的容器并没有太大不同，所以在它发布后不久，Cloud Foundry 的首席产品经理 James Bayer 就在社区里做了一次详细对比，告诉用户 Docker 实际上只是一个同样使用 Cgroups 和 Namespace 实现的“沙盒”而已，没有什么特别的黑科技，也不需要特别关注。

然而，短短几个月，Docker 项目就迅速崛起了。它的崛起速度如此之快，以至于 Cloud Foundry 以及所有的 PaaS 社区还没来得及成为它的竞争对手，就直接被宣告出局了。那时候，一位多年的 PaaS 从业者曾经如此感慨道：这简直就是一场“降维打击”啊。

难道这一次，连闯荡多年的“老江湖”James Bayer 也看走眼了么？

并没有。

事实上，Docker 项目确实与 Cloud Foundry 的容器在大部分功能和实现原理上都是一样的，可偏偏就是这剩下的一小部分不一样的功能，成了 Docker 项目接下来“呼风唤雨”的不二法宝。

这个功能，就是 Docker 镜像。

恐怕连 Docker 项目的作者 Solomon Hykes 自己当时都没想到，这个小小的创新，在短短几年内就如此迅速地改变了整个云计算领域的发展历程。

PaaS 之所以能够帮助用户大规模部署应用到集群里，是因为它提供了一套应用打包的功能。可偏偏就是这个打包功能，却成了 PaaS 日后不断遭到用户诟病的一个“软肋”。

出现这个问题的根本原因是，一旦用上了 PaaS，用户就必须为每种语言、每种框架，甚至每个版本的应用维护一个打好的包。这个打包过程，没有任何章法可循，更麻烦的是，明明在本地运行得好好的应用，却需要做很多修改和配置工作才能在 PaaS 里运行起来。而这些修改和配置，并没有什么经验可以借鉴，基本上得靠不断试错，直到你摸清楚了本地应用和远端 PaaS 匹配的“脾气”才能够搞定。

最后结局就是，“cf push”确实是能一键部署了，但是为了实现这个一键部署，用户为每个应用打包的工作可谓一波三折，费尽心机。

而**Docker 镜像解决的，恰恰就是打包这个根本性的问题。** 所谓 Docker 镜像，其实就是一个压缩包。但是这个压缩包里的内容，比 PaaS 的应用可执行文件 + 启停脚本的组合就要丰富多了。实际上，大多数 Docker 镜像是直接由一个完整操作系统的所有文件和目录构成的，所以这个压缩包里的内容跟你本地开发和测试环境用的操作系统是完全一样的。

这就有意思了：假设你的应用在本地运行时，能看见的环境是 CentOS 7.2 操作系统的所有文件和目录，那么只要用 CentOS 7.2 的 ISO 做一个压缩包，再把你的应用可执行文件也压缩进去，那么无论在哪里解压这个压缩包，都可以得到与你本地测试时一样的环境。当然，你的应用也在里面！

这就是 Docker 镜像最厉害的地方：只要有这个压缩包在手，你就可以使用某种技术创建一个“沙盒”，在“沙盒”中解压这个压缩包，然后就可以运行你的程序了。

更重要的是，这个压缩包包含了完整的操作系统文件和目录，也就是包含了这个应用运行所需要的所有依赖，所以你可以先用这个压缩包在本地进行开发和测试，完成之后，再把这个压缩包上传到云端运行。

在这个过程中，你完全不需要进行任何配置或者修改，因为这个压缩包赋予了你一种极其宝贵的能力：**本地环境和云端环境的高度一致！**

这，**正是 Docker 镜像的精髓。**

那么，有了 Docker 镜像这个利器，PaaS 里最核心的打包系统一下子就没了用武之地，最让用户抓狂的打包过程也随之消失了。相比之下，在当今的互联网里，Docker 镜像需要的操作系统文件和目录，可谓唾手可得。

所以，你只需要提供一个下载好的操作系统文件与目录，然后使用它制作一个压缩包即可，这个命令就是：

```
$ docker build " 我的镜像 "
```

一旦镜像制作完成，用户就可以让 Docker 创建一个“沙盒”来解压这个镜像，然后在“沙盒”中运行自己的应用，这个命令就是：

```
$ docker run " 我的镜像 "
```

当然，docker run 创建的“沙盒”，也是使用 Cgroups 和 Namespace 机制创建出来的隔离环境。

所以，**Docker 项目给 PaaS 世界带来的“降维打击”，其实是提供了一种非常便利的打包机制。这种机制直接打包了应用运行所需要的整个操作系统，从而保证了本地环境和云端环境的高度一致，避免了用户通过“试错”来匹配两种不同运行环境之间差异的痛苦过程。**

而对于开发者们来说，在终于体验到了生产力解放所带来的痛快之后，他们自然选择了用脚投票，直接宣告了 PaaS 时代的结束。

不过，Docker 项目固然解决了应用打包的难题，但正如前面所介绍的那样，它并不能代替 PaaS 完成大规模部署应用的职责。

遗憾的是，考虑到 Docker 公司是一个与自己有潜在竞争关系的商业实体，再加上对 Docker 项目普及程度的错误判断，Cloud Foundry 项目并没有第一时间使用 Docker 作为自己的核心依赖，去替换自己那套饱受诟病的打包流程。

反倒是一些机敏的创业公司，纷纷在第一时间推出了 Docker 容器集群管理的开源项目（比如 Deis 和 Flynn），它们一般称自己为 CaaS，即 Container-as-a-Service，用来跟“过时”的 PaaS 们划清界限。

Docker 为什么出现?

一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，不同版本环境的兼容，对运维人员是极大的考验！

环境配置如此麻烦，换一台机器，就要重来一次，费力费时。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。解决开发人员说的“在我的机器上可正常工作”的问题。

之前在服务器配置一个应用的运行环境，要安装各种软件，就拿一个基本的工程项目的环境来说吧，Java/Tomcat/MySQL/JDBC驱动包等。安装和配置这些东西有多麻烦就不说了，它还不能跨平台。假如我们是在 Windows 上安装的这些环境，到了 Linux 又得重新装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。

传统上认为，软件编码开发/测试结束后，所产出的成果即是程序或是能够编译执行的二进制字节码文件等（Java为例）。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让维运团队得以部署应用程式，**开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，仍然常常发生部署失败的状况。**

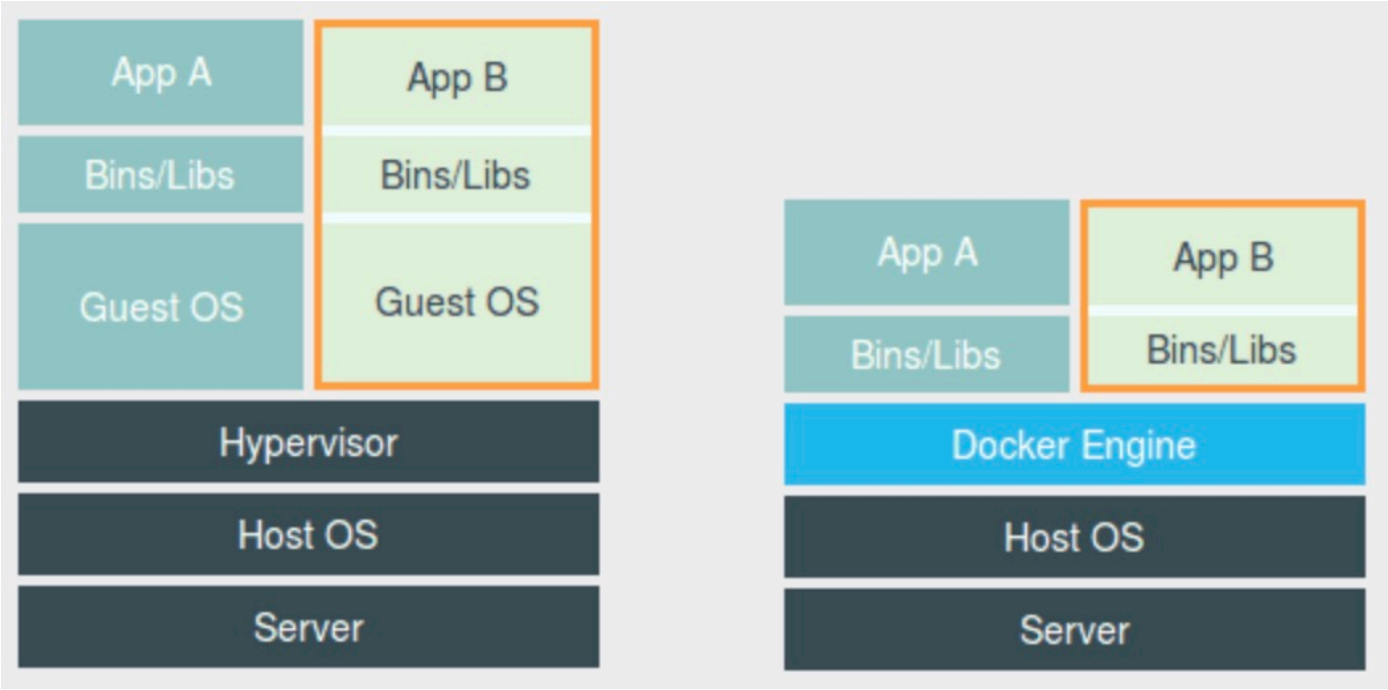
Docker之所以发展如此迅速，也是因为它对此给出了一个标准化的解决方案。

Docker镜像的设计，**使得Docker得以打破过去「程序即应用」的观念。通过Docker镜像 (images) 将应用程序所需要的系统环境，由下而上打包，达到应用程序跨平台间的无缝接轨运作。**

Docker的思想来自于集装箱，集装箱解决了什么问题？在一艘大船上，可以把货物规整的摆放起来。并且各种各样的货物被集装箱标准化了，集装箱和集装箱之间不会互相影响。那么我就不需要专门运送水果的船和专门运送化学品的船了。只要这些货物在集装箱里封装的好好的，那我就可以用一艘大船把他们都运走。

Docker 就是类似的理念。

为什么Docker比VM 快



这幅图的左边，画出了虚拟机的工作原理。其中，名为 Hypervisor 的软件是虚拟机最主要的部分。它通过硬件虚拟化功能，模拟出了运行一个操作系统需要的各种硬件，比如 CPU、内存、I/O 设备等等。然后，它在这些虚拟的硬件上安装了一个新的操作系统，即 Guest OS。

这样，用户的应用进程就可以运行在这个虚拟的机器中，它能看到的自然也只有 Guest OS 的文件和目录，以及这个机器里的虚拟设备。这就是为什么虚拟机也能起到将不同的应用进程相互隔离的作用。

而这幅图的右边，则用一个名为 Docker Engine 的软件替换了 Hypervisor。这也是为什么，很多人会把 Docker 项目称为“轻量级”虚拟化技术的原因，实际上就是把虚拟机的概念套在了容器上。

可是这样的说法，却并不严谨。

在理解了 Namespace 的工作方式之后，你就会明白，跟真实存在的虚拟机不同，在使用 Docker 的时候，并没有一个真正的“Docker 容器”运行在宿主机里面。Docker 项目帮助用户启动的，还是原来的应用进程，只不过在创建这些进程时，Docker 为它们加上了各种各样的 Namespace 参数。

这时，这些进程就会觉得自己是各自 PID Namespace 里的第 1 号进程，只能看到各自 Mount Namespace 里挂载的目录和文件，只能访问到各自 Network Namespace 里的网络设备，就仿佛运行在一个个“容器”里面，与世隔绝。

Docker 技术原理

Namespace

Namespace 是 Linux 内核的一项功能，该功能对内核资源进行隔离，使得容器中的进程都可以在单独的命名空间中运行，并且只可以访问当前容器命名空间的资源。Namespace 可以隔离进程 ID、主机名、用户 ID、文件名、网络访问和进程间通信等相关资源。

Docker 主要用到以下五种命名空间。

- pid namespace：用于隔离进程 ID。
- net namespace：隔离网络接口，在虚拟的 net namespace 内用户可以拥有自己独立的 IP、路由、端口等。
- mnt namespace：文件系统挂载点隔离。
- ipc namespace：信号量，消息队列和共享内存的隔离。
- uts namespace：主机名和域名的隔离。

Cgroups

Cgroups 是一种 Linux 内核功能，可以限制和隔离进程的资源使用情况（CPU、内存、磁盘 I/O、网络等）。在容器的实现中，Cgroups 通常用来限制容器的 CPU 和内存等资源的使用。

联合文件系统

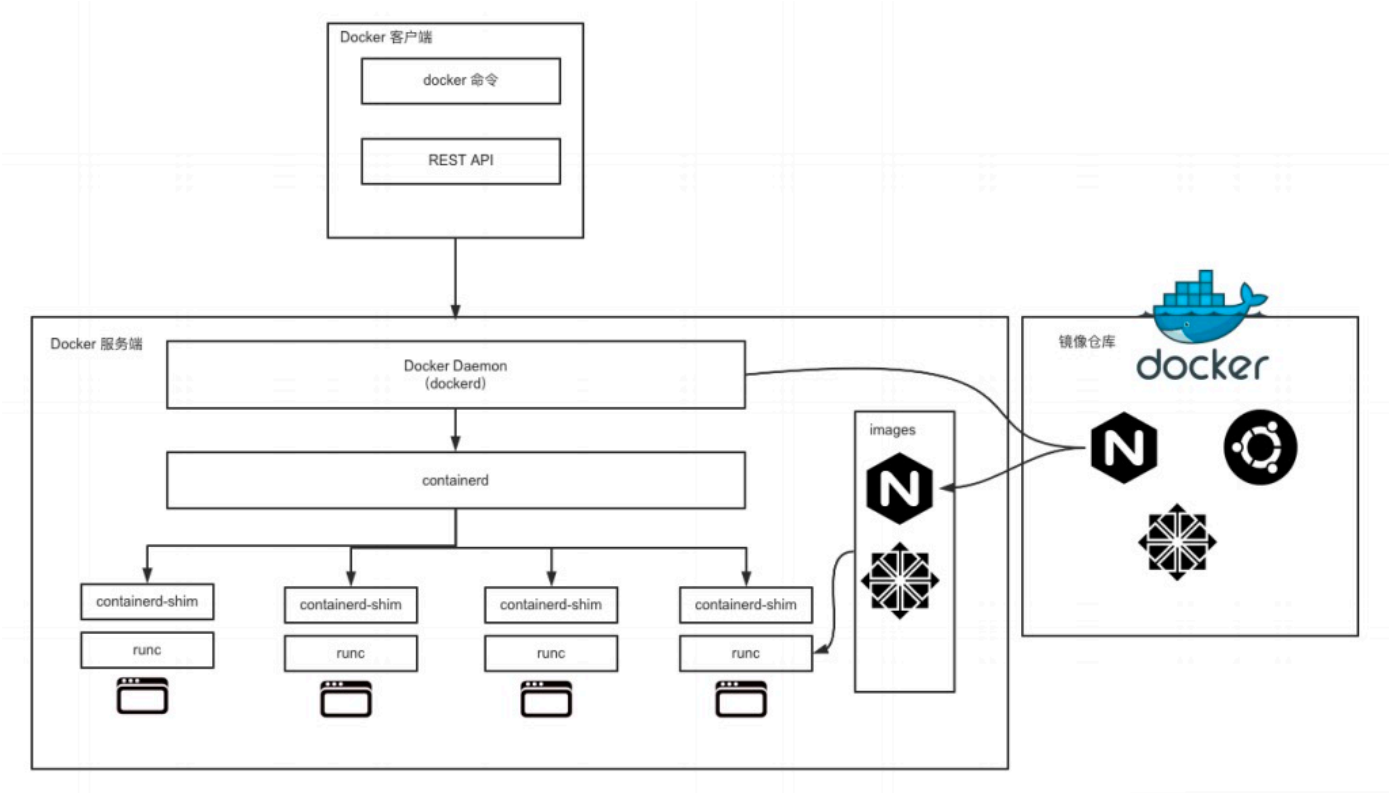
联合文件系统，又叫 UnionFS，是一种通过创建文件层进程操作的文件系统，因此，联合文件系统非常轻快。Docker 使用联合文件系统为容器提供构建层，使得容器可以实现写时复制以及镜像的分层构建和存储。常用的联合文件系统有 AUFS、Overlay 和 Devicemapper 等。

> **Docker 就是利用 Linux 的 Namespace 、Cgroups 和联合文件系统三大机制来实现的， 所以它的原理是使用 Namespace 做主机名、网络、PID 等资源的隔离，使用 Cgroups 对进程或者进程组做资源（例如：CPU、内存等）的限制，联合文件系统用于镜像构建和容器运行环境。**

Namespace 演示

。。。

Docker 核心架构



Docker 整体架构采用 C/S（客户端 / 服务器）模式，主要由客户端和服务端两大部分组成。客户端负责发送操作指令，服务端负责接收和处理指令。客户端和服务端通信有多种方式，既可以在同一台机器上通过UNIX套接字通信，也可以通过网络连接远程通信。

Docker 客户端

Docker 客户端其实是一种泛称。其中 docker 命令是 Docker 用户与 Docker 服务端交互的主要方式。除了使用 docker 命令的方式，还可以使用直接请求 REST API 的方式与 Docker 服务端交互，甚至还可以使用各种语言的 SDK 与 Docker 服务端交互。目前社区维护着 Go、Java、Python、PHP 等数十种语言的 SDK，足以满足你的日常需求。

Docker 服务端

Docker 服务端是 Docker 所有后台服务的统称。其中 dockerd 是一个非常重要的后台管理进程，它负责响应和处理来自 Docker 客户端的请求，然后将客户端的请求转化为 Docker 的具体操作。例如镜像、容器、网络和挂载卷等具体对象的操作和管理。

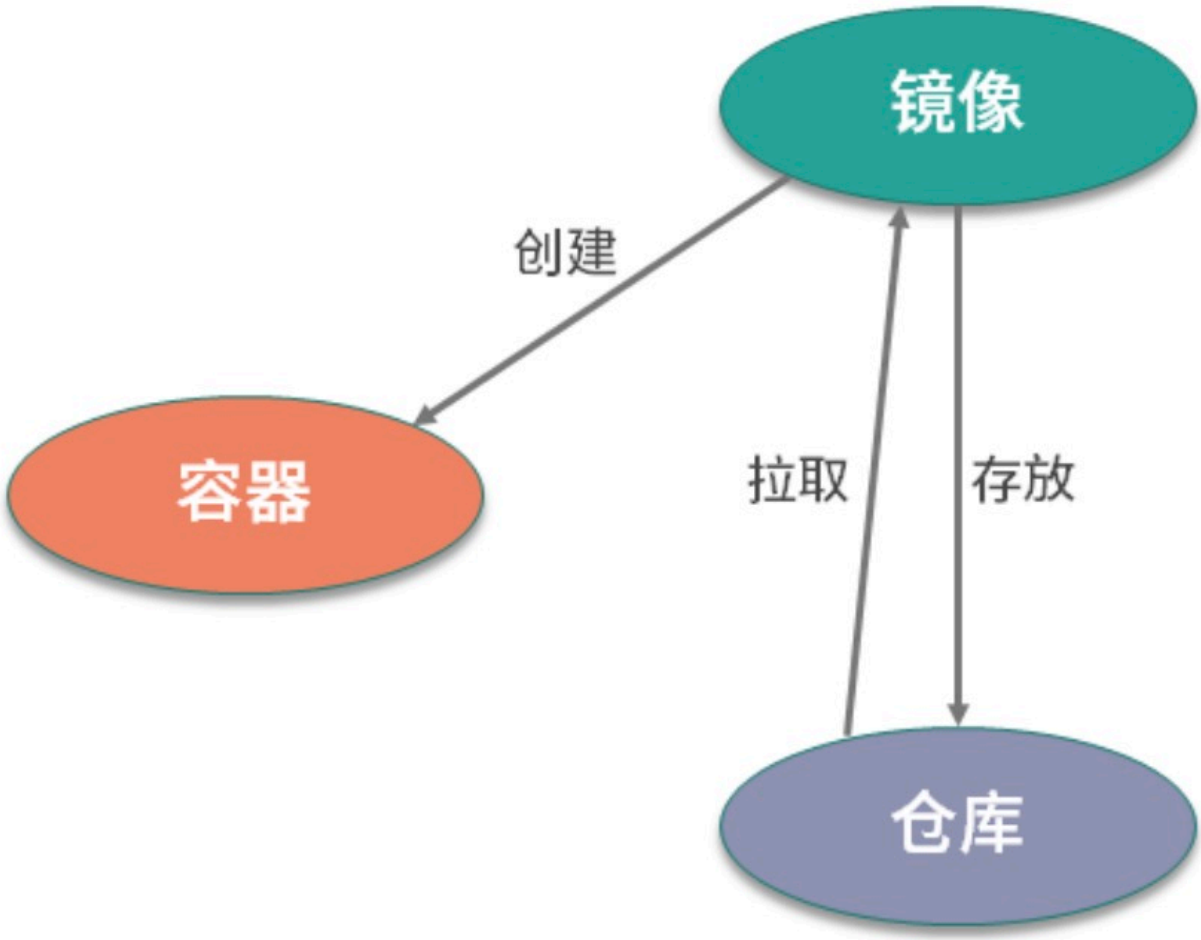
Docker 从诞生到现在，服务端经历了多次架构重构。起初，服务端的组件是全部集成在 docker 二进制里。但是从 1.11 版本开始， dockerd 已经成了独立的二进制，此时的容器也不是直接由 dockerd 来启动了，而是集成了 containerd、runC 等多个组件。

虽然 Docker 的架构在不停重构，但是各个模块的基本功能和定位并没有变化。它和一般的 C/S 架构系统一样，Docker 服务端模块负责和 Docker 客户端交互，并管理 Docker 的容器、镜像、网络等资源。

runC 和 containerd

- `runC`是 Docker 官方按照 OCI 容器运行时标准的一个实现。通俗地讲，runC 是一个用来运行容器的轻量级工具，是真正用来运行容器的。
- `containerd`是 Docker 服务端的一个核心组件，它是从`dockerd`中剥离出来的，它的诞生完全遵循 OCI 标准，是容器标准化后的产物。`containerd`通过 containerd-shim 启动并管理 runC，可以说`containerd`真正管理了容器的生命周期。

Docker 核心概念



镜像（image）

镜像是什么呢？通俗地讲，它是一个只读的文件和文件夹组合。它包含了容器运行时所需要的所有基础文件和配置信息，是容器启动的基础。所以你想启动一个容器，那首先必须要有一个镜像。镜像是 Docker 容器启动的先决条件。

如果你想要使用一个镜像，你可以用这两种方式：

自己创建镜像。通常情况下，一个镜像是基于一个基础镜像构建的，你可以在基础镜像上添加一些用户自定义的内容。例如你可以基于centos 镜像制作你自己的业务镜像，首先安装 nginx 服务，然后部署你的应用程序，最后做一些自定义配置，这样一个业务镜像就做好了。

从功能镜像仓库拉取别人制作好的镜像。一些常用的软件或者系统都会有官方已经制作好的镜像，例如nginx、ubuntu、centos、mysql等，你可以到 Docker Hub 搜索并下载它们。

> Docker 镜像（Image）就是一个只读的模板。镜像可以用来创建 Docker 容器，一个镜像可以创建很多容器。 就好似 Java 中的 类和对象，类就是镜像，容器就是对象！

容器（container）

容器是什么呢？容器是 Docker 的另一个核心概念。通俗地讲，容器是镜像的运行实体。镜像是静态的只读文件，而容器带有运行时需要的可写文件层，并且容器中的进程属于运行状态。即容器运行着真正的应用进程。容器有初建、运行、停止、暂停和删除五种状态。

虽然容器的本质是主机上运行的一个进程，但是容器有自己独立的命名空间隔离和资源限制。也就是说，在容器内部，无法看到主机上的进程、环境变量、网络等信息，这是容器与直接运行在主机上进程的本质区别。

> Docker 利用容器（Container）独立运行的一个或一组应用。容器是用镜像创建的运行实例。容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。

仓库（repository）

Docker 的镜像仓库类似于代码仓库，用来存储和分发 Docker 镜像。镜像仓库分为公共镜像仓库和私有镜像仓库。

目前，Docker Hub 是 Docker 官方的公开镜像仓库，它不仅有很多应用或者操作系统的官方镜像，还有很多组织或者个人开发的镜像供我们免费存放、下载、研究和使用。除了公开镜像仓库，你也可以构建自己的私有镜像仓库。

仓库(Repository)和仓库注册服务器（Registry）是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

Docker 安装

- Install Docker Desktop on Mac
- Install Docker Desktop on Windows
- Install Docker Engine on CentOS

Dockerfile 构建镜像

Dockerfile 类似 Makefile，Makefile 是通过 make 构建整个工程项目，而 Dockerfile 是通过 Docker build 来构建整个镜像。

Dockerfile 常用的指令：

Dockerfile 指令	指令简介
FROM	Dockerfile 除了注释第一行必须是 FROM ，FROM 后面跟镜像名称，代表我们要基于哪个基础镜像构建我们的容器。
RUN	RUN 后面跟一个具体的命令，类似于 Linux 命令行执行命令。
ADD	拷贝本机文件或者远程文件到镜像内
COPY	拷贝本机文件到镜像内
USER	指定容器启动的用户
ENTRYPOINT	容器的启动命令
CMD	CMD 为 ENTRYPOINT 指令提供默认参数，也可以单独使用 CMD 指定容器启动参数
ENV	指定容器运行时的环境变量，格式为 key=value
ARG	定义外部变量，构建镜像时可以使用 build-arg = 的格式传递参数用于构建
EXPOSE	指定容器监听的端口，格式为 [port]/tcp 或者 [port]/udp
WORKDIR	为 Dockerfile 中跟在其后的所有 RUN、CMD、ENTRYPOINT、COPY 和 ADD 命令设置工作目录。

一个最简单的 Dockerfile 如下：

```
FROM nginx:alpine

COPY dist/ /usr/share/nginx/html

EXPOSE 80
```

Docker Compose 服务编排

Docker Compose 是 Docker 三剑客之一，前身是 Orchard 公司开发的 Fig，2014 年 Docker 收购了 Orchard 公司，然后将 Fig 重命名为 Docker Compose。现阶段 Docker Compose 是 Docker 官方的单机多容器管理系统，它本质是一个 Python 脚本，它通过解析用户编写的 yaml 文件，调用 Docker API 实现动态的创建和管理多个容器。

Docker Compose 文件主要分为三部分： services（服务）、networks（网络） 和 volumes（数据卷）。

- services（服务）：服务定义了容器启动的各项配置，就像我们执行`docker run`命令时传递的容器启动的参数一样，指定了容器应该如何启动，例如容器的启动参数，容器的镜像和环境变量等；
- networks（网络）：网络定义了容器的网络配置；
- volumes（数据卷）：数据卷定义了容器的卷配置。

一个典型的 Docker Compose 文件结构如下：

```yaml
version: "3"
services:
  nginx:
    ## ... 省略部分配置
  networks:
    frontend:
    backend:
  volumes:
    db-data:
```

Docker 实战

环境

- MacOS Big Sur: 11.4
- Docker: 3.4.0
- Docker Engine: 20.10.7
- Compose: 1.29.2

> 以下演示适用与与 Mac 和 Linux 系统。Windows 系统需要做微调，如：路径

前端

docker-demo/frontend

后端

docker-demo/backend/docker-compose.yml

```yaml

# compose 语法版本号
version: '3'

services:
  # 指定 service 名
  gateway-service:
    # 指定使用的镜像，默认从 docker hub
    image: registry.cn-beijing.aliyuncs.com/xinlc/gateway:1.0.0-DEV
    # 指定启动后的容器名
    container_name: recruit-gateway
    # 重启规则，这里指定除非手动停止，还有失败自动重启等等
    restart: unless-stopped
    # 指定开放的端口号，主机:容器
    ports:
      - "5800:80"
    # 指定网络
    networks:
      - net-recruit-dev
```

```
    # 加载环境变量文件
    env_file:
      - ./test.env
    # 定义数据卷所挂载路径设置 HOST:CONTAINER，ro 代表只读
    volumes:
      - /usr/share/zoneinfo/Asia/Shanghai:/etc/localtime:ro
      - /etc/timezone:/etc/timezone:ro
    # 指定 log 使用的驱动，日志文件最大 20MB，最多两个文件，新日志会自动替换旧日志
    logging:
      driver: "json-file"
      options:
        max-size: "20m"
        max-file: "2"
# 配置网络
networks:
  net-recruit-dev:
    driver: bridge

```

Dockerfile 文件：

```
# 第一段构建所依赖的基础镜像
FROM openjdk:8-jre-alpine AS builder

# 切换到工作目录
WORKDIR application
# 定义外部变量，jar包存在位置
ARG JAR_FILE=target/*.jar
# 重命名jar包
COPY ${JAR_FILE} app.jar

# 提取分层信息
RUN java -Djarmode=layertools -jar app.jar extract

# 第二阶段
FROM openjdk:8-jre-alpine

# apk 镜像源
#RUN echo -e "https://mirror.tuna.tsinghua.edu.cn/alpine/v3.9/main\n\
#https://mirror.tuna.tsinghua.edu.cn/alpine/v3.9/community" > /etc/apk/repositories

# 安装字体库，easyexcel 依赖
# RUN apk --update add ttf-dejavu && \
#       rm -rf /var/cache/apk/*

WORKDIR application

# 制作镜像层，注意顺序，不经常变动的应放在前面
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
```

```
COPY --from=builder application/application/ ./

# 定义环境变量
ENV TZ="Asia/Shanghai" PORT=80 JAVA_OPTS=" " SPRING_OPTS=" "

# 声明数据卷
VOLUME ["/log", "/data"]

# 声明对外暴露端口
EXPOSE $PORT

# 容器启动执行命令
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -Djava.security.egd=file:/dev/./urandom
org.springframework.boot.loader.JarLauncher ${SPRING_OPTS}"]

```

这个 dockerfile 表示先进行一次临时镜像构建标记为builder，并加载一次全量jar包，然后执行`java -Djarmode=layertools -jar app.jar extract`命令将jar包分解为分层打包目录，再次构建一个新镜像，按照(`java -Djarmode=layertools -jar app.jar list`) list 的目录顺序分批将分层目录加载到 docker 镜像中。

附录

Docker 常用命令

```
docker run [options] image [command][arg...]
-d                  # 后台运行容器
-e                  # 设置环境变量
--expose/-p         # 宿主端口：容器端口映射
--name              # 指定容器名称
-v                  # 宿主目录：容器目录，挂载磁盘卷

docker pull <image> # 下载镜像
docker search <image> # 搜索镜像
docker start/stop <ID> # 启动/停止容器
docker ps <ID>      # 查看运行中的容器
docker ps -a        # 查看所有容器
dokcer ps -l        # 查看最后一次创建的容器
docker rm <ID>      # 删除容器
docker rmi <ID>     # 删除镜像
docker logs <ID>    # 查看容器日志
docker images       # 显示镜像列表
docker inspect <ID> # 查看容器信息
docker exec -it <ID> bash # 登录到容器中
docker help         # 终极命令
```

Docker Compose 常用命令

```
docker-compose [-f <arg>...] [options] [--] [COMMAND] [ARGS...]
  -f, --file <FILE>             # 指定 docker-compose 文件，默认为 docker-compose.yml
  -p, --project-name <NAME>     # 指定项目名称，默认使用当前目录名称作为项目名称
  --verbose                     # 输出调试信息
  --log-level <LEVEL>           # 日志级别 (DEBUG, INFO, WARNING, ERROR, CRITICAL)

  build               # 构建服务
  config              # 校验和查看 Compose 文件
  up                  # 创建并且启动服务
  down                # 停止服务，并且删除相关资源
  exec                # 在一个运行的容器中运行指定命令
  images              # 列出镜像
  logs                # 查看容器输出
  port                # 打印容器端口所映射出的公共端口
  ps                  # 列出项目中的容器列表
  pull                # 拉取服务中的所有镜像
  restart             # 重启服务
  rm                  # 删除项目中已经停止的容器
  start               # 启动服务
  stop                # 停止服务
  top                 # 限制服务中正在运行中的进程信息
```

参考

- 《由浅入深吃透 Docker》
- 《深入剖析 Kubernetes》
- 《Spring Boot & Kubernetes 云原生微服务实践》
- Install Docker Engine
- Docker CLI reference
- Docker Compose CLI reference
- Dockerfile reference
- Compose file version 3 reference
- Deploy to Swarm