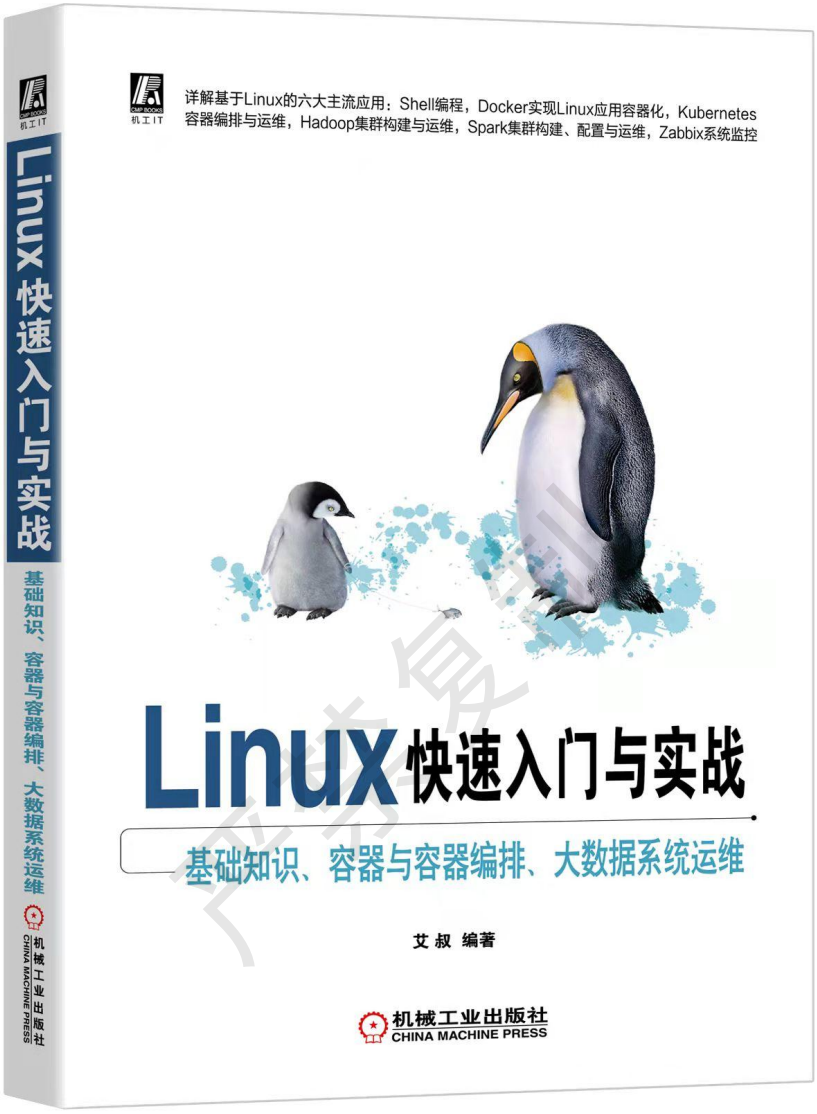
本文内容来源

《Linux 快速入门与实战 基础知识、容器与容器编排、大数据系统运维》



1、面向 Linux、大数据、云计算运维与研发工程师

2、基于 CentOS 8.2 讲解，同样适用于其他主流的 Linux 发行版

3、配套高清视频课程、资源文件、扩展阅读和实践操作电子书

4、零基础讲解 Linux 核心概念、系统组成、运行机制、高频命令和进阶使用，快速打下 Linux 坚实基础，少走弯路少踩坑

5、精选 Linux 在大数据、云原生以及系统监控等领域的实战案例，详解 Linux 下 Shell 编程、Docker、Kubernetes、Hadoop、Spark 和 Zabbix 相关技术，快速累积 Linux 实战经验

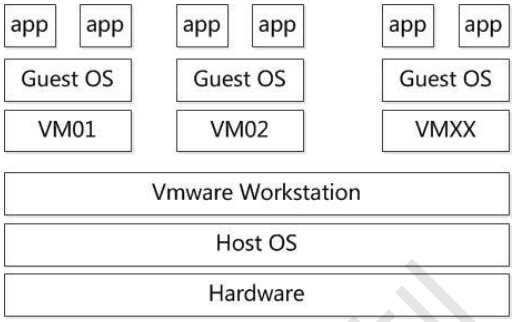
1.1 Docker 容器同虚拟机的区别

Docker 容器和虚拟机的区别是什么，这是初学者经常问到的问题。就使用而言，两者在某种程度上是类似的：假设 Docker 容器和虚拟机上都启动了 SSH 服务，用户使用 PuTTY 可以分别登陆到 Docker 容器和虚拟机的 Linux 系统，而后续的命令行操作就完全一样了。

那么 Docker 容器同虚拟机的区别是什么呢？Docker 容器同虚拟机最主要的区别，就在于两者的实现机制不同，弄清楚这个，将为读者深入理解 Docker 容器的实现机制，更好地使用 Docker 打下基础，具体说明如下。

1. 虚拟机的技术原理

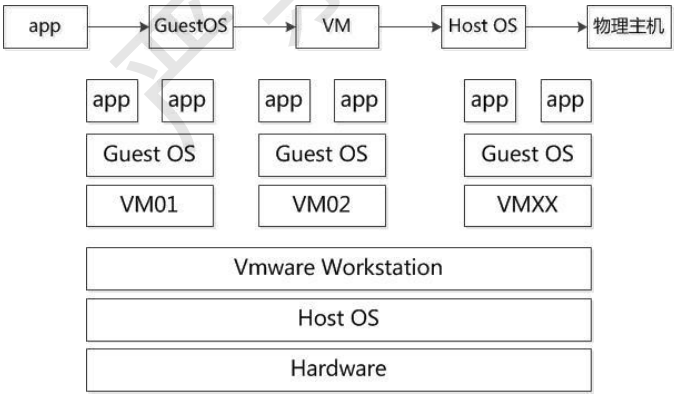
虚拟机（VMware Workstation）的技术原理图如图 1-11 所示。



1-11 虚拟机（VMware Workstation）技术原理图

如图 1-11 所示，虚拟机在 Host OS 之上，先虚拟了一台计算机（VM），如 VM01 和 VM02，VM 和商场购买的计算机一样，有 CPU、内存、硬盘和网卡等各种设备；VM 之上再安装操作系统（Guest OS），可以是 Linux 也可以是 Windows，每个 VM 可以运行一个 Guest OS，这些 Guest OS 可以同时运行；然后在 Guest OS 上再安装各种应用，即图 1-12 中的 app。

因此，虚拟机上的 app 的执行路线如图 1-7 所示。



1-12 虚拟机（VMware Workstation）app 执行路线图

VMware Workstation 虚拟的硬件（如 CPU），并不是用纯软件来实现的，这样做效率太低了，真正使用纯软件来模拟实现 CPU 的是另一种虚拟机 Bochs；

较早版本的 VMware Workstation 采用二进制翻译和直接执行技术，分类处理不同类型的指令，以提升指令执行效率；VMware Workstation 9 之后的版本，则结合物理 CPU 的硬件虚拟化特性，进一步提升指令执行的效率。

2. Docker 容器的技术原理

Docker 容器的技术原理如图 1-13 所示，具体说明如下。

- app 之下没有 Guest OS 这一层，所有容器中的所有 app，共享同一个内核，即 Host 主机上的 Linux 内核；
- Docker 容器利用 Linux 内核的 namespaces 技术，实现了程序运行环境的隔离，每个容器都有独立的 root 文件系统、独立的进程编号、独立的网络等等，看起来和独占 Linux 系统没有区别。容器 a 的进程看不到容器 b 中的进程，也看不到容器 b 中目录和文件。总之，这些容器之间是互相隔离的，互相感受不到对方的存在，也互不影响。但在实现上，所有容器中的所有 app 都是共享同一个内核的。因此，Docker 容器实现了轻量级的虚拟化。

由于 Docker 容器隔离（虚拟）的对象是操作系统的核心元素，诸如 root 文件系统、进程编号、用户组、挂载点和网络等等，因此也说 Docker 容器实现了操作系统的虚拟化。

容器中的 app 进程，本质上就是直接在 Host 上运行的进程，只是它们有不同的 namespaces 罢了。因此，Docker 容器中 app 的执行，其调用路线如图。

1-13 Docker 容器 app 的调用路线图

3. Docker 容器同虚拟机的比较

Docker 容器同虚拟机在实现原理、存储开销和使用方式等方面对比如表 1-2 所示。

表 1-2 Docker 容器同虚拟机对比表

对比项	Docker 容器	虚拟机
实现方式	没有 Guest OS，进程直接在 Host 上运行，减少了调用层级，可以做到秒级的启动；不需要运行 Guest OS 及相关服务进程，因此该进程的执行几乎没有额外的系统开销。	有 Guest OS 层，增加了调用层级；进程启动前，要先启动 Guest OS 和相关服务，因此，进程启动速度慢；此外 Guest OS 以及相关服务进程的运行，增加了额外的系统开销。
存储开销	一个 Docker 镜像可以启动多个 Docker 容器，即使启动 100 个容器，也只需要一份镜像文件，有效降低了镜像文件所占用的存储空间。	每个虚拟机镜像只能启动一个虚拟机，因此，启动 100 个虚拟机，需要 100 份虚拟机镜像文件，即便这些虚拟机的系统和配置完全一样，也是如此，因此，其存储空间和虚拟机个数是线性关系。
镜像构建	支持以脚本（Dockerfile）的方式构建镜像，镜像构建可以自动完成；镜像迁移只需传输 Dockerfile，不需要传输镜像本身，大幅降低带宽；可以做到只在需要的时候，动态构建镜像，不用的时候，不需要存储镜像，节省存储空间；还可以复用 Dockerfile 构建新镜像。	镜像的构建需要手工进行；镜像必须提前构建好，镜像迁移需要传输整个镜像文件，占用带宽资源；使用前必须先准备好镜像，占用存储空间；镜像有修改时，可能需要全部重来一遍构建镜像的操作。
使用方式	可以控制在 Docker 容器中启动哪个命令，精度高；支持命令操作 Docker，便于实现自动化管理。	只能启动 Guest OS，无法控制启动 Guest OS 中的哪个命令，精度低；难以实现自动化管理。
应用的类型	因为 Docker 容器共享的是 Linux 内核，因此，Docker 容器中只能运行 Linux 应用程序。	Guest OS 可以安装 Linux/Windows，从而安装对应平台下的应用程序。

基于表 1-2，Docker 容器和虚拟机在不同应用场景下的选择如下所示。

1）如果 Host 是 Windows，需要用到其他版本的 Windows，选择虚拟机；

2）如果 Host 是 Windows，需要用到单机版的 Linux，选择虚拟机更方便；

3）如果 Host 是 Windows，需要多个 Linux 节点组网，如果 Linux 节点小于 3 个，选择虚拟机更方便，否则，选择 Docker 容器；

4）如果 Host 是 Linux，需要多个 Linux 节点组网，选择 Docker 容器。

1.2 Docker 解决了哪些问题

我们评价一种技术的价值时，主要是要看它解决了什么问题。问题本身的价值，就决定了技术的价值，如果问题越基础，解决的难度越大，则解决该问题的技术的价值就越大。那么，Docker 作为目前最热门的技术之一，它解决了什么问题呢？

简而言之，Docker 主要解决了三个层面的问题。

1. 在传统的开发层面，Docker 解决了 Linux 程序运行环境的隔离和迁移问题

Linux 程序的运行环境，是指该进程及其依赖。在传统的开发中，同一台计算机上的进程 a 和进程 b，其运行环境是混杂在一起的，既不能隔离又不能限制资源。

（1）Linux 程序运行环境无法隔离和迁移所带来的问题

1）Linux 程序运行环境无法隔离会导致一系列的问题：例如进程 a 依赖数据库 MySQL，进程 b 依赖 HTTP 服务器 Apache，MySQL 和 Apache 在同一台主机同时运行。如果进程 a 操作 MySQL 时，不小心引起系统崩溃，就会导致进程 b 无法运行；同样的，进程 b 如果引起系统崩溃，也会导致 MySQL 无法工作；此外，如果进程 a 占用 CPU 过高，就会影响到进程 b 的运行；还有，如果进程 a 和进程 b 都依赖 MySQL，但两者所依赖的 MySQL 版本是不一样的，这两个 MySQL 就会因为没有隔离，而导致安装或运行冲突。这些都是因为程序运行环境没有隔离所带来的问题。

与此同时，传统的开发中，很难做到记录并提取指定 Linux 程序的运行环境。这样就导致，我们知道如何构建程序 a 的运行环境，但是，一旦构建好后，却没有办法搞清楚具体的依赖有哪些，更不用说精确提取这些依赖和配置了。这样就会导致 Linux 程序同它的运行环境以及它所在的主机绑定在一起，无法迁移。

2）Linux 程序环境无法迁移也会导致一系列的问题：例如运行一个经典的 LAMP（Linux、Apache、MySQL 和 PHP）网站运行环境是 LAMP 组件和依赖以及相应的配置。在计算机 a 构建好这个环境后，如果要在计算机 b 上也运行该网站，就只能在计算机 b 上将计算机 a 上的构建步骤再重复一遍，而没有办法将网站的运行环境打一个包复制到计算机 b 上。这样就导致，开发阶段，如果是多人协作开发，则每个开发者都要部署一遍运行环境；测试阶段，测试者也要部署运行环境；最后上线运行阶段，运维人员还要部署运行环境。这样既造成了大量的重复工作，还容易导致这些依赖版本的不一致问题。

（2）Docker 容器解决了 Linux 程序运行环境的隔离和迁移问题

1）Docker 使用容器技术，解决了 Linux 程序运行环境的隔离问题。

每个容器都有独立的 root 文件系统、独立进程 ID、用户和网络等等。Linux 程序在容器中运行，是感受不到其他容器的存在的，就好像独占一个 Linux 系统。可以把程序放入不同的容器中，每个容器运行一个程序，即使容器 a 中的进程崩溃，也不会影响到容器 b 中的进程；同时容器 a 和容器 b 有独立的文件系统，因此即使部署了版本不一致的 MySQL 也不会冲突，还能同时运行；最后，Docker 支持对容器的资源进行限制，从而避免一个进程占用资源过多，而对其他进程产生影响。

2）Docker 镜像则解决了程序环境的迁移问题。

Docker 将程序的运行环境打包成了一个文件，在其他计算机上，只要安装了 Docker 引擎，就可以利用该镜像运行容器，直接运行其中的程序，不需要再构建运行环境。这样，可以极大地简化运行环境的部署工作，还能够严格地保证一致性，避免出错。

2. 在虚拟机层面，Docker 解决了虚拟化的开销问题

和 Docker 相比，虚拟机技术同样解决了程序运行环境的隔离和迁移问题。但是，虚拟机的虚拟化开销很大，这是由于虚拟机的实现原理所导致的，虚拟机增加了 Guest OS 这一

层，而且还增加了 Guest OS 以及相应服务进程的开销，此外，虚拟机的镜像存储空间同虚拟机个数成线性关系，更多详细内容可以参考 5.1.2 节中的说明。

而 Docker 利用 Linux 内核的 namespaces 技术，实现了容器的隔离，不同容器中的进程，只是 namespaces 不同，但本质上就是运行在 Host 主机上的进程，既没有增加调用的层级，同时又没有带来 Guest OS 以及服务进程等额外开销。此外，一个 Docker 镜像可以启动多个 Docker 容器，即使运行 100 个 Docker 容器，也只要 1 份镜像，这样可以大大节省存储空间，而虚拟机则需要 100 个镜像。因此，同虚拟机相比，Docker 可以有效解决虚拟化的开销问题。

和虚拟机技术相比，Docker 可以控制在容器中启动哪个命令，而虚拟机则只能控制启动 Guest OS，无法精确到里面的进程；此外 Docker 支持使用命令完成上述操作，而 VMware 只能手动执行；以及 Docker 镜像的开销小，一个镜像可以启动多个容器；还有 Docker 镜像可以使用脚本的方式动态构建等等。

综上所述，从虚拟机的层面，Docker 还解决了虚拟环境（容器）的大规模自动部署的问题。

3. 在容器层面，Docker 解决了完善性和易用性的问题

在 Docker 出现之前，Linux 就已经有容器的概念和技术了，典型的如 LXC（Linux Container），LXC 基于 Linux 的 namespaces 和 cgroups 技术，实现了轻量级的虚拟化，即容器技术，早期的 Docker 版本就是直接基于 LXC 实现的。因此，Docker 和 LXC 都是位于容器层面的，它们也都解决了虚拟化的隔离和开销问题。那么，Docker 相对于 LXC，解决了什么问题呢？又或者说，Docker 在容器层面，还解决了什么问题呢？最主要的，Docker 相对 LXC 解决了容器功能的完善性和易用性问题。

（1）Docker 在完善性方面解决的问题

1）程序运行环境的迁移问题

Docker 将程序的依赖打包成一个镜像文件，这是一个标准单元，该镜像文件在任何安装了 Docker 的计算机上都可以运行容器，并运行其中的程序。这样就很方便地实现了程序运行环境的迁移。而 LXC 没有镜像的概念，其容器的运行和本机有紧密绑定，无法方便地迁移到其他计算机上。

2）程序运行环境的版本管理问题

Docker 镜像实现了版本管理，容器中所做的修改，可以存储为一个镜像版本，所有版本按照提交的时间顺序排列。这样可以对整个程序的运行环境的历史演变过程进行记录和维护。当出现错误时，用户可以根据版本的历史记录，一路向上回溯，定位错误非常方便。在实际开发中，不同角色的人员可以根据自己的需要，使用不同的版本。这样，既保证了效率，又不容易出错。

3）程序运行环境的构建问题

Docker 使用 Dockerfile 来构建镜像，这是一个脚本文件，其中包含了镜像的构建步骤，基于该 Dockerfile 和相应的构建命令就可以自动构建出一个 Docker 镜像。这种方式有三个好处，具体说明如下。

- 实现了镜像的自动构建；
- 实现了构建过程的复用，后续如果我们需要在已构建出的镜像上做修改时，则只需要修改 Dockerfile 中对应的部分即可，然后运行命令自动构建即可；
- 镜像迁移时，可以只传输 Dockerfile，不需要传输镜像文件，节省带宽。

（2）Docker 在易用性方面解决的问题

- 早期的 Docker 是基于 LXC 接口实现的，在使用上 Docker 更加接近用户习惯；
- Docker 支持命令参数来简化配置，例如网络设置等等，而 LXC 则需要单独执行一条条的底层命令，更加繁琐和复杂；

- Docker 自身的安装也非常简单，使用 yum 一条命令就可以完成，而 LXC 则需要安装多个 Package 和模板，相对更繁琐。

小结

总之，Docker 解决了程序运行环境的迁移和隔离问题，同时还解决了虚拟化开销的问题，并提供了完善、易用的容器功能。这些问题的解决，大大减少了开发过程中，重复构建程序运行环境的工作量；也大大减少了产品在研发、测试和部署各个阶段因为运行环境不一致而造成的错误；大幅提升了计算机硬件资源的利用率；有效降低了进程之间的相互影响；还可以很方便地实现容器在分布式集群中的大规模自动部署。

1.3 Docker 底层技术

Docker 底层使用了三大技术，分别是：namespaces、cgroups、UnionFS，具体说明如下。

1. namespaces

namespaces 是 Linux 内核的技术，它以一种抽象的形式，将操作系统中的某一类全局资源包装成一个 namespace（如 PID（进程 ID） namespace、User namespace、Network namespace 等），位于该 namespace 内的进程可以访问到这些资源，而 namespace 之外的进程则访问不到这些资源。Namespaces 实现了操作系统全局资源的隔离，Docker 正是利用其特性实现了容器的隔离。

以 PID namespace 为例，默认情况下，Linux 中所有进程的 ID 是统一编号的。但是，如果在一个新 namespace 中启动一个进程的话，Linux 会根据新 namespace 中的进程情况对该进程进行编号，例如新 namespace 中的第一个进程的 ID 为 1。因此，不同的 PID namespace 中都有 ID 为 1 的进程；每个进程只能看到同一个 PID namespace 中的进程，无法看到其他 PID namespace 中的进程，这样就实现了 PID 的隔离；而且它们认为当前操作系统的进程就只有这么多，独占整个系统。然而，实际上是所有 PID namespace 的进程共享一个 Linux 内核，而且所有的进程除了在各自的 PID namespace 中有 ID 外，还有一个全局的进程 ID，因此，PID namespace 还实现了 PID 的虚拟。

除了 PID namespace 外，Linux 内核还支持以下 7 种全局资源 namespace，具体如表 1-3 所示。

表 1-3 namepace 分类表

序号	namespace 名称	说明
1	Cgroup	隔离 Cgroup root directory。
2	IPC	隔离 IPC（进程间通信机制）资源，包括：管道（Pipe）、信号（Signal）、消息队列（Message Queue）、共享内存、信号量（Semaphore）和套接字（Socket）。
3	Network	隔离网络协议栈、IP 路由表、防火墙规则等。
4	Mount	隔离挂载点（Mount point）。
5	PID	隔离进程 ID。
6	User	隔离用户和组。
7	UTS	UTS 是“UNIX Time-sharing System”的缩写，用来隔离主机名和域名。

目前 Docker 用到了以上 IPC、Network、Mount、PID 和 UTS 这 5 种 namespace，基于它们实现了容器，也就是说，每个容器都有自己独立的 IPC、Network、Mount、PID 和 UTS 的 namespace，容器不同，它们的这 5 种 namespace 也就不同。

使用“man namespaces”可以查看 namespace 的更多详细信息。

2. cgroups

cgroups 是“Control cgroups”的缩写，它是 Linux 内核的技术，它可以将进程按照层次关系组织起来，然后限制这些进程对 CPU、内存等资源的使用。其中，内核的 cgroup 接口是通过一个名字为 cgroupfs 的伪文件系统所提供的，所谓 cgroup 就是一组被绑定到了 cgroupfs 的限制（或参数）集合之上的进程。

Docker 利用 crgoups 来限制容器对硬件资源的使用，例如限制容器所使用的 CPU 和内存资源。这样可以使得每个容器都能够分配到合理的资源，而不会因为其他容器占用过多的资源而受到影响。

可以使用“man cgroups”查看 cgroups 的更多详细信息。

3. UnionFS

UnionFS 是一类特别的文件系统，它可以把多个目录合并成一个虚拟目录，供用户使用。Docker 利用 UnionFS 的这个特性，实现了容器的启动和镜像的存储。

UnionFS 有很多种实现，如 AUFS、btrfs、vfs、DeviceMapper 和 OverlayFS。Docker 可以通过配置“storage-driver”参数，指定使用哪种 UnionFS。因此 Docker 也将其使用的 UnionFS，称之为“storage driver”（存储驱动）。

OverlayFS 有 Overlay 和 Overlay2 两种版本，其中 Overlay2 是 Overlay 的升级版，它的性能更强，开销更小，更加稳定，是 Docker 官方推荐的存储驱动。

（1）UnionFS 的工作原理

本小节以 OverlayFS 为例说明 UnionFS 的工作原理，OverlayFS 有 3 类目录：lowerdir、upperdir 和 merged，其中 lowerdir 是多个待合并的只读目录；upperdir 是一个可读写的待合并的目录；merged 是 lowerdir 和 upperdir 合并后的目录，用户在 merged 目录中所做的修改，将写入 upperdir 目录。

OverlayFS 使用示例如下，该 mount 命令会将目录 1、目录 2、目录 3 的内容合并到目录 5 显示，而我们在目录 5 中的修改，将会保存到目录 3 中。

```
mount -t overlay example -o lowerdir=./1:./2,upperdir=./3,workdir=./4      ./5
```

上述命令和参数说明如下。

- “-t overlay”表示挂载的文件系统类型为 OverlayFS；
- example 是自定义的挂载名称；
- “-o”后面接一系列的选项，lowerdir 包括目录 1 和目录 2，upperdir 为目录 3，workdir 是目录 4，是保存中间结果的工作目录，merged 为目录 5。

目录 1~5 挂载前如下所示，其中目录 1 中有两个文件：1G 和 aaa；目录 2 中有 1 个文件 bbb，目录 3、目录 4 和目录 5 都为空；

```
1
├── 1G
└── aaa
2
└── bbb
3
4
5
```

执行上述挂载命令后，目录 5 的内容如下，可以看到目录 1 和目录 2 的内容都合并到了目录 5 之中。

```
[root@localhost ~]# ls 5
```

1G aaa bbb

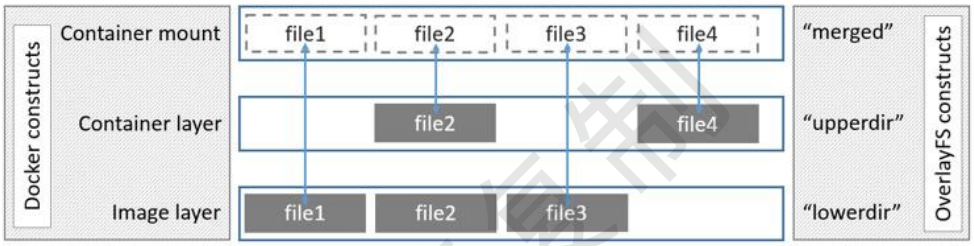
如果在目录 5 中增加/删除/修改文件，即使被修改的文件原来位于目录 1 或目录 2，那么所做的修改都会在目录 3（upperdir）中记录，目录 1 和目录 2（lowerdir）是不会做任何修改的。

lowerdir 也是分层的，最左边的目录为最上层的目录，按照自左向右到顺序依次向下叠加，目录越往下，优先级越低。上例中，目录 1 在目录 2 左侧，因此，目录 1 在上，目录 2 在下，如果目录 1 和目录 2 都有一个同名文件 aaa，那么，在目录 5 中看到的 aaa 的内容，来自目录 1 的 aaa 文件。

如果后续对 aaa 进行了修改，则修改后的 aaa 会存储到目录 3 中，目录 5 中看到的 aaa 的内容，则来自目录 3 的 aaa。

（2）UnionFS 在 Docker 中的应用

OverlayFS 在 Docker 中的应用如图 1-14 所示[9]。其中，左侧表示 Docker 的存储体系，分为 3 层：“Image layer”、“Container layer”和“Container mount”；右侧则是 OverlayFS 的存储体系，也分为 3 层：lowerdir、upperdir 和 merged；Docker 存储体系的每一层和 OverlayFS 的每一层是一一对应的，中间的部分则是每一层的具体内容。

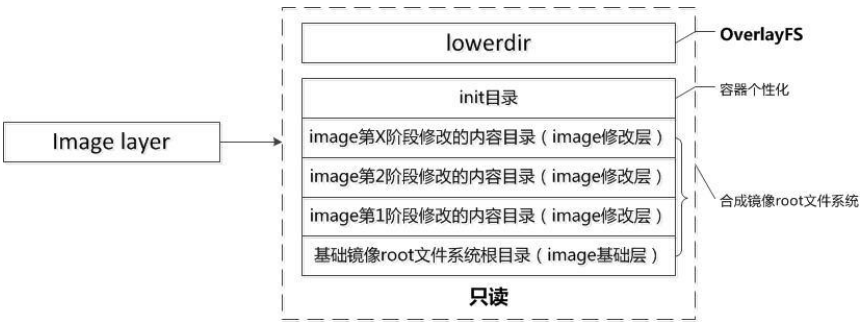


1-14 OverlayFS 在 Docker 中的应用原理图

结合图 1-14，对 OverlayFS 在 Docker 中的应用说明如下。

1）Image layer

Docker 使用“Image layer”层来存储镜像数据。Docker 镜像本质上由一个基础镜像（root 文件系统）加上若干次修改（文件的增/删/改）得到。如图 1-15 所示，OverlayFS 正好可以实现这样的镜像：将基础镜像 root 文件系统的根目录，放置在 lowerdir 的最底层，然后按照修改的顺序，将每一个阶段的修改（该修改保存在一个目录中），依次作为 lowerdir 的一层目录，向上叠加。Docker 镜像有分层的概念，因此，将基础镜像 root 文件系统的根目录称之为 image 基础层，而各阶段修改的内容目录则统称为 image 修改层。



1-15 Docker 镜像分层存储原理图

图 1-15 中“Image layer”的文件有 file1、file2 和 file3，但实际上，file1 可能来自 image 基础层，file2 则可能来自某个 image 修改层，file3 则可能来自另一个 image 修改层。

lowerdir 中的目录，是以只读（read-only）的方式挂载的，因此，用户在容器中的修改，不会更新到 lowerdir 中的目录，也不会更新到镜像。这也就是为什么在容器 a 的根目录下创建文件 c，然后停止/删除容器 a，再用同样的镜像运行容器 b，容器 b 的根目录下并没有文件 c 的原因。

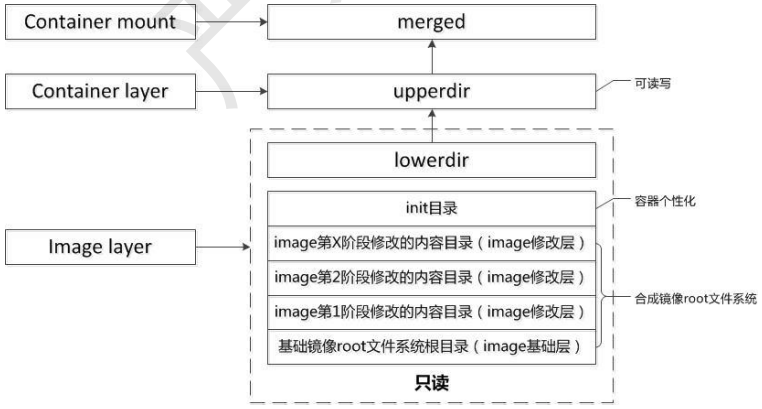
在 lowerdir 中还有一个特殊的目录，就是 init 目录，该目录中保存了容器的个性化信息，例如主机名、Host 信息等。容器的个性化信息是通过“docker run”传入参数指定的，Docker 会在运行容器时，新建一个 init 目录（如 d5ac51ee2dcac5263ceb2fd2438e6f3d13ce8169c0c3ef53c520637044ca141b-init），然后将参数中的个性化信息，写成配置文件，存储在 init 下的 diff 目录中，最后将 diff 目录作为 lowerdir 中的最上层目录，同 lowerdir 中的其他目录合并。正是因为有了 init 目录及其个性化信息，使得 Docker：1. 可以在容器启动时，配置容器的个性化信息；2. 支持一个镜像启动多个容器。

2）Container layer

Docker 使用“Container layer”层存储用户对容器的修改（即 root 文件系统中的增/删/改）。如图 1-16 所示，“Container layer”对应 OverlayFS 的 upperdir 目录。Docker 运行容器时，会为每个容器新建一个目录，指定该目录为 upperdir，然后和“Image layer”对应的 lowerdir 中的目录合并，构成容器的 root 文件系统。用户在容器中所做的修改，都会存储到该容器的 upperdir 中，如图 1-16 的“Container layer”所示，file2 是容器中已有的，并被修改过的文件，file4 则是容器中新建的文件。如果容器删除，则 upperdir 也会被删除，这就是为什么容器的修改不会自动保存到镜像的原因；如果运行命令将容器保存到镜像，那么 Docker 会将此 upperdir 作为 lowerdir 的一层新目录，下次再通过该镜像运行容器时，就会加入这一层新的目录；由于每次容器的存储，都会形成一个新的目录，这样就可以很方便地实现镜像的版本管理。

3）Container mount

“Container mount”是容器的 root 文件系统，它对应 OverlayFS 的 merged 目录，它由 lowerdir 中的目录，加上 upperdir 目录合并而成，具体组成如图 1-16 所示。



1-16 Container mount 组成图

（3）小结

总之，Docker 基于 UnionFS（OverlayFS），很巧妙地实现了以下功能。

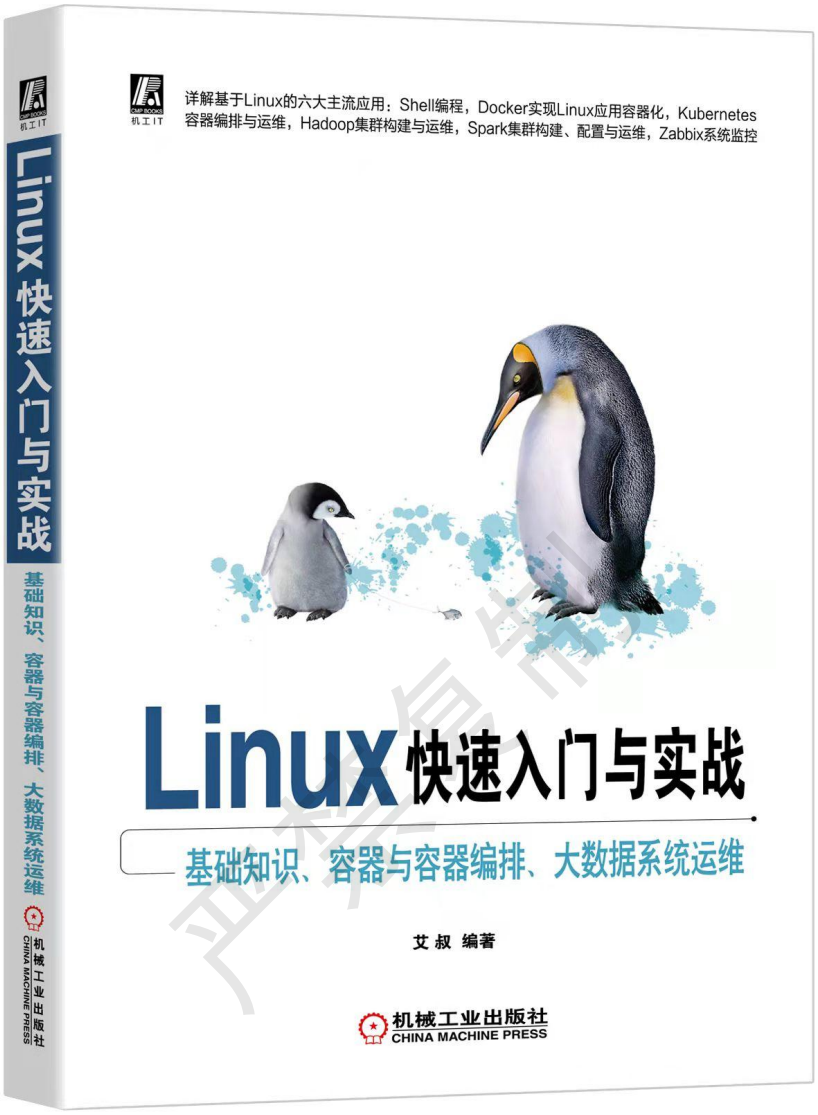
1）**镜像的分层存储**。这样既可以充分复用之前的镜像成果，避免了重复占用存储空间，也提升了镜像的存储速度（只做变更存储，不是整体存储），同时还做到了镜像的版本管理；

2）**容器的个性化**。Docker 通过增加一个 init 目录，将容器的设置（启动参数/配置文件）放置在 init 目录；

3）**一个镜像对应多个容器**。镜像的 layer 是只读的，init 目录用于容器个性化，upperdir 则用于容器的修改，实现了读写分离。如果不是这样，一个镜像只能对应一个容器。

上述内容来源

《Linux 快速入门与实战 基础知识、容器与容器编排、大数据系统运维》



1、面向 Linux、大数据、云计算运维与研发工程师

2、基于 CentOS 8.2 讲解，同样适用于其他主流的 Linux 发行版

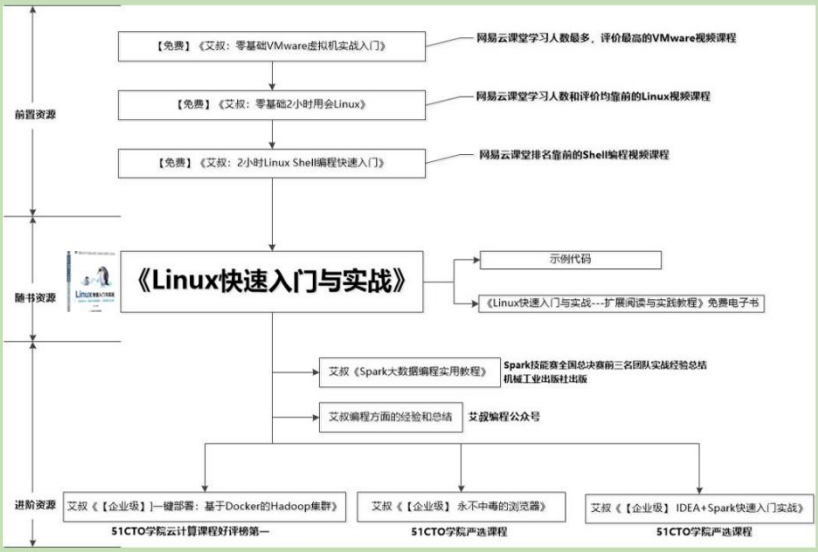
3、配套高清视频课程、资源文件、扩展阅读和实践操作电子书

4、零基础讲解 Linux 核心概念、系统组成、运行机制、高频命令和进阶使用，快速打下 Linux 坚实基础，少走弯路少踩坑

5、精选 Linux 在大数据、云原生以及系统监控等领域的实战案例，详解 Linux 下 Shell 编程、Docker、Kubernetes、Hadoop、Spark 和 Zabbix 相关技术，快速累积 Linux 实战经验

目录

配套高清视频、扩展阅读与实践电子书、示例代码



扫码了解本书

当当

京东

（作者简介）

文艾（艾叔）：原解放军理工大学–奇虎360云计算联合实验室技术负责人，系统分析师，51CTO学院严选讲师；具有多年Linux下的开发、运维和教学经验，对Linux下的Docker、Kubernetes、Hadoop和Spark等系统有深入研究和丰富的实践经验；带领团队完成了华为、中兴和奇虎360等公司的多个校企合作Linux相关项目；指导零基础本科生参加国家级科技创新竞赛和编程大赛，共获得全国特等奖1次，一等奖2次，二等奖2次；通过“艾叔编程”公众号和网易云课堂开设了一系列Linux相关的免费课程，已帮助8万多名学习者入门编程并深受好评。

加入艾叔 Linux+大数据+云原生学习群

一本书 

从零开始，全面学习

Linux+Shell+Docker+Kuberbetes(k8s)+Hadoop+Spark+Zabbix

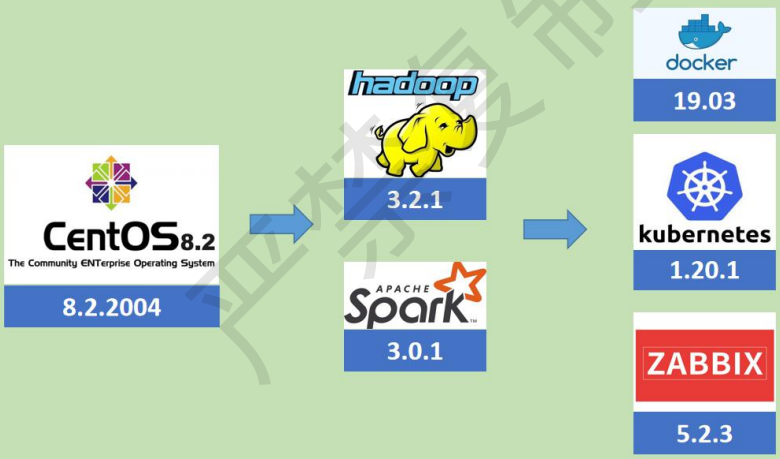
Linux基础-》系统运维-》容器-》容器编排-》大数据平台运维

面向Linux初学者、大数据、云计算运维与研发工程师

基于CentOS 8.2讲解，同样适用于其他主流的Linux发行版

艾叔编程

精选当前最热门开源技术和平台，开启高薪之门



精选Linux实战案例，快速累积经验，少走弯路，少踩坑

项目实战（一）：Kubernetes(k8s)容器编排实践

- 使用kubeadm快速构建Kubernetes集群
- Kubernetes Pod调度---容器在指定的节点上运行
- 故障调试实践
- Kubernetes集群中容器的高可用实践
- Kubernetes Pod多容器运行
- Kubernetes容器数据的持久化存储
- Kubernetes容器服务暴露实践
- Kubernetes容器规模自动伸缩实践---HPA
- 节点性能数据采集与监控实践

艾叔编程

实战项目（二）：基于Docker单机构建100个节点的分布式集群

综合实战

- Linux使用
- Docker核心技术
- Shell编程

- Docker快速安装与配置
- Docker常用命令
- 容器运行
- Docker容器只读运行
- Docker持久化存储
- Docker日志查看
- Docker镜像构建
- Dockerfile编写
- Docker自定义bridge网络使用与热插拔实践
- Docker host、overlay、MACVLAN网络原理与使用

艾叔编程

实战项目（三）：一键部署基于Docker的Hadoop+Spark集群

它是**Docker+Hadoop+Spark**的深度综合应用

只用**一个命令**，就可以实现**任意节点Hadoop+Spark集群**的部署

即使是1000个节点的集群，也可以轻松运维

和VMware相比

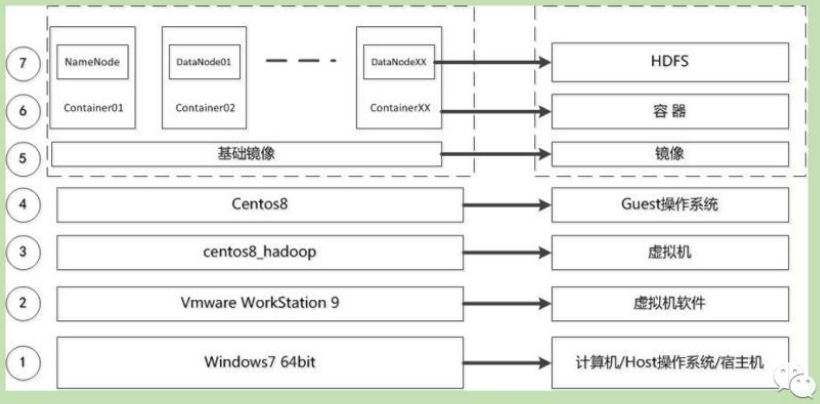
不管集群规模多大，它都只需要一个存储镜像

和VMware相比，它开销更小

可以在一台笔记本上，轻松运行10个节点以上的分布式系统

艾叔编程

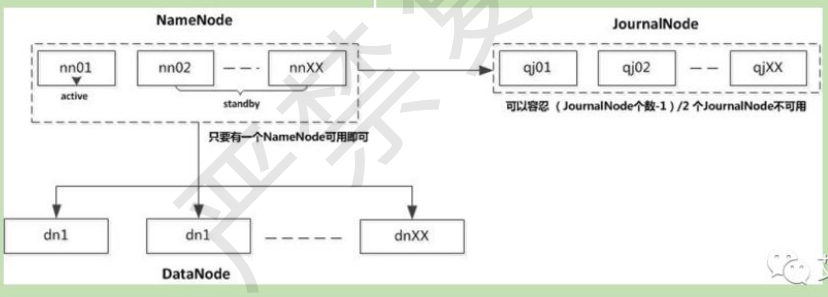
实战项目（三）：一键部署基于Docker的Hadoop+Spark集群



实战项目（四）：HDFS高可用性（HA）实战

综合实战

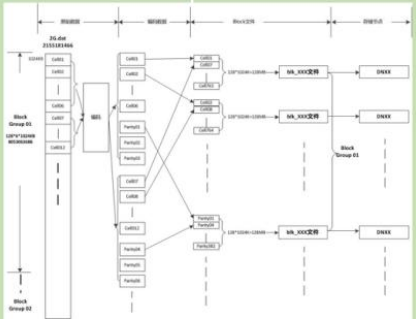
Linux使用 | Docker核心技术 | Shell编程 | HDFS HA



实战项目（五）：HDFS纠删码（Erasure Coding）实战

综合实战

Linux使用 | Docker核心技术 | Shell编程 | HDFS EC



实战项目（六）：使用Zabbix监控Hadoop分布式文件系统HDFS

综合实战

Linux使用 | Shell编程 | Docker核心技术 | HDFS集群构建 | Zabbix核心技术

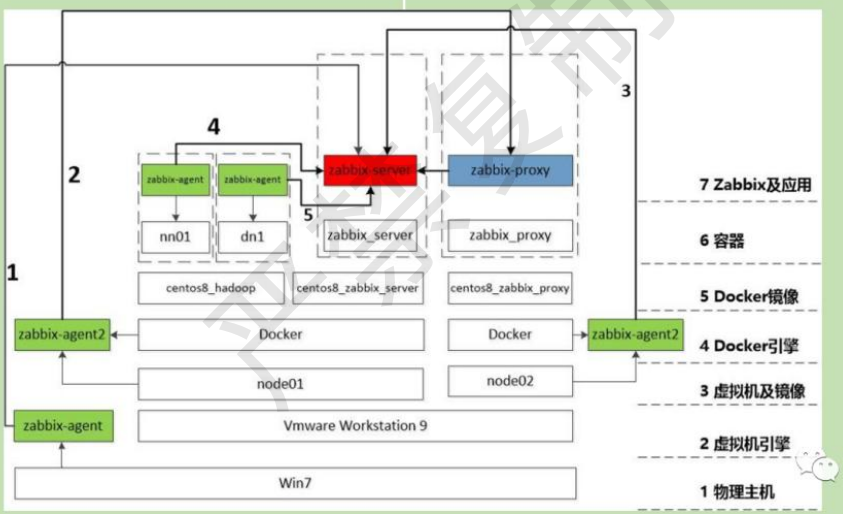
Docker安装、环境构建 | Docker常用命令 | Dockerfile编写 | Docker镜像构建

Docker持久化存储 | 容器运行

Zabbix快速部署与使用 | Zabbix监控流程 | Zabbix监控Linux | Zabbix监控Windows | 监控日志

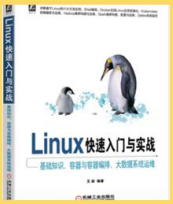
Zabbix监控数据库 | Zabbix监控Docker容器 | Zabbix监控HDFS | Zabbix Proxy监控内网节点

实战项目（六）：使用Zabbix监控Hadoop分布式文件系统HDFS



如果你想在面试、找工作时

有可以**展示的高水平作品**



强烈推荐学习此书！！！

扫码了解本书

当当

京东