计算机科学
COMPUTER SCIENCE

面向高性能计算系统的容器技术综述

陈轶阳, 王小宁, 卢莎莎, 肖海力

引用本文

陈轶阳, 王小宁, 卢莎莎, 肖海力.面向高性能计算系统的容器技术综述[J]. 计算机科学, 2023, 50(2): 353-363.

CHEN Yiyang, WANG Xiaoning, LU Shasha, XIAO Haili. Survey of Container Technology for High-performance Computing System [J]. Computer Science, 2023, 50(2): 353-363.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220100163

面向高性能计算系统的容器技术综述

陈轶阳[1,2]　王小宁[1]　卢莎莎[1]　肖海力[1]
1 中国科学院计算机网络信息中心　北京 100190
2 中国科学院大学计算机科学与技术学院　北京 100049
(chenyiyang@cnic.cn)

摘　要　容器技术在云计算行业已得到普遍使用,主要用于服务软件环境的快速移植和自动化部署。随着高性能计算、大数据、人工智能技术的深度融合,高性能计算系统的应用软件依赖和配置日益复杂,超算中心对用户自定义软件栈的需求越来越强烈。因此,容器技术在高性能计算系统的应用环境下也发展出多种实现软件,以满足用户自定义软件栈等实际需求。文中总结了容器技术的发展历史,阐述了容器在Linux平台的技术原理,分析并评价了于适用高性能计算系统的容器实现软件,最后展望未来面向高性能计算系统的容器技术研究方向。

关键词:高性能计算;容器;虚拟化;应用软件部署
中图法分类号　TP391

Survey of Container Technology for High-performance Computing System

CHEN Yiyang[1,2],WANG Xiaoning[1],LU Shasha[1] and XIAO Haili[1]
1 Computer Network Information Center,Chinese Academy of Sciences,Beijing 100190,China
2 School of Computer Science and Technology,University of Chinese Academy of Sciences,Beijing 100049,China

Abstract　Container technology has been widely used in the cloud computing industry,mainly for rapid migration and automated deployment of service software environments. With the deep integration of high performance computing,big data and artificial intelligence technologies,the application software dependency and configuration of high performance computing systems are becoming increasingly complex,and the demand for user-defined software stacks in supercomputing centers is getting stronger. Therefore,in the application environment of high-performance computing systems,a variety of container implementations have also been developed to meet the practical needs such as user-defined software stacks. This paper summarizes the development history of container technology,explains the technical principles of containers in Linux platform,analyzes and evaluates the container implementation software for high-performance computing systems,and finally the future research direction of container technology for high-performance computing system is prospected.

Keywords　High performance computing,Container,Virtualization,Application software deployment

1　引言

随着2013年Docker[1]的推出,操作系统虚拟化技术在工业界和学术界获得了许多关注。从私有数据中心到公有云,容器以其轻量、可迁移等特性彻底改变了很多企业服务开发和部署的方式[2],容器技术在高性能计算领域也同样具有较高的应用需求和研究价值。随着高性能计算、大数据、人工智能技术的深度融合,高性能计算系统的应用软件栈日益复杂,因此研究人员希望能够利用容器技术封装和部署应用,避免应用编译和执行环境的相互干扰,从而便于项目移植。

在web of science检索系统中,以检索式TS=("hpc" AND "container")检索论文,每年的论文量和引用量如图1所示。

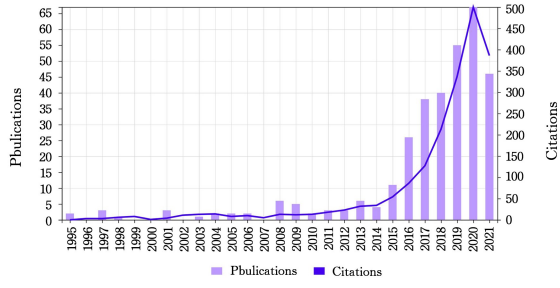

图1　web of science检索的论文和引用数量

Fig. 1　Number of papers and citations indexed by web of science

从2013年开始,HPC和容器相关的论文呈大幅增长趋势。2015年左右,面向高性能计算系统的容器实现Singulari-

到稿日期:2022-01-18　返修日期:2022-07-19
基金项目:中国科学院战略性先导科技专项项目(A类)(XDA19020101)
This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences(XDA19020101).
通信作者:王小宁(wxn@sccas.cn)

ty[3]，Shifter[4]，Charliecloud[5]等相继发布，带动了许多相关工作，可以看到，2015 年后相关论文明显增多。

在谷歌学术上以检索式 hpc AND“container *”搜索论文，结果如图 2 所示，也呈现逐年增长的趋势，可见容器技术是高性能计算行业中一个较新的研究热点。

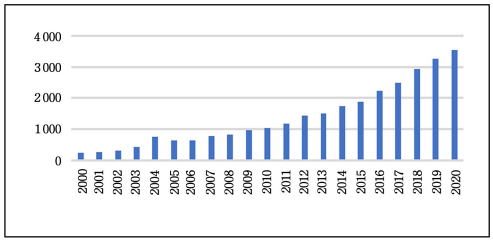

图 2 谷歌学术检索的论文数量

Fig. 2 Number of papers indexed by Google Scholar

相比互联网公司中广泛使用的云服务平台，高性能计算系统有着独特的属性和使用方式。

(1)高性能计算系统是一个多用户共享的计算基础设施，用户在使用高性能计算系统时始终只有普通用户权限。高性能计算系统通常由多个计算节点通过高速互联网络构成，用户在使用之前需要申请相应的计算资源数量，程序运行结束之后释放资源。如果当前系统的剩余资源无法满足用户的需求，用户的计算请求将处于排队状态，直到有满足其需求的资源数量。整个过程中，用户完全是以系统普通用户身份执行任务。

(2)高性能计算系统上运行的应用大多是计算程序，而性能永远是计算程序追求的第一目标。用户申请获得资源之后，给定计算所需的数据文件和参数，启动应用软件开展计算，当计算完成之后即退出释放资源。为了追求性能，计算程序大都依赖硬件实现高效的计算和通信，比如利用 GPU 进行异构计算[6]、利用 InfiniBand 实现高速互联互通[7]。

(3)高性能计算系统往往有着较高的安全防护需求，因此，国内的高性能计算系统大多是隔离于互联网的。用户需要通过 VPN 或者动态口令卡等方式登录高性能计算系统进行操作，并且在系统内部，无论是登录结点还是计算结点均不允许访问外部互联网。

因此，高性能计算系统对于容器实现有着特殊的需求。

(1)适应高性能计算系统的多用户环境，允许非特权用户启动容器，保证容器内进程的权限与引入容器前的普通进程一致，杜绝用户的提权操作。

(2)进行宽松的资源隔离策略，允许容器内进程使用系统的公共资源，包括分布式文件系统、计算加速卡和高速互连网络等。

(3)适配高性能计算系统上的基础设施，比如支持与作业管理系统集成，适应大规模分布式存储架构下的镜像存储。

(4)性能应该与裸机运行接近，没有明显的性能差异。

本文开展了面向高性能计算系统的容器技术综述研究，以期为后续相关研究工作提供指导。本文第 2 节介绍了容器技术的起源与发展；第 3 节整理阐述了容器在 Linux 平台上依赖的核心技术；第 4 节对现有高性能计算系统上的容器实现软件进行了分析；第 5 节总结了现有高性能计算系统中的容器应用现状，并展望未来高性能计算系统中容器技术的发展；最后总结全文。

2 容器技术的起源与发展

容器本质上是一种虚拟化技术。在计算机领域中，虚拟化是一种将计算资源(CPU、内存、磁盘、网络等)抽象、转化、分割，以呈现一个或多个计算资源的技术。主流的虚拟化技术分为基于虚拟机的硬件虚拟化[8]和基于容器的操作系统虚拟化[9]。

1974 年，Popek 等给出了虚拟机的明确定义，就计算机体系结构是否支持高效虚拟化问题提出了他们简化的判断方法，并给出了形式化的证明[10]。他们认为，虚拟机应该能够虚拟化一整套硬件资源，包括一个或多个处理器、内存、外存和其他外设。Hypervisor，即 VMM，是提供计算机虚拟化的软件，一般关注它们 3 个方面的特性。

(1)效率：大部分指令能够由硬件直接执行，无需 VMM 的干预。

(2)等价性：在 VMM 上和等效机器上，同一程序的运行效果应该一致。

(3)资源控制：VMM 能够控制机器上的可用资源。

满足这些条件的控制程序被称为 Virtual Machine Monitor(VMM，又可以称为 Hypervisor)，VMM 生成的环境就是虚拟机[11]。

除了基于虚拟机的虚拟化技术之外，操作系统级别的虚拟化也是一项很重要的技术。Hypervisor 一个重要的作用是在一个共同的硬件上实现不同操作系统的相互隔离。20 世纪 70 年代，Hypervisor 提供的分时功能最终由操作系统直接实现，而 20 世纪 90 年代和 21 世纪的内核开发者也开始考虑直接由操作系统提供环境的隔离功能，操作系统虚拟化由此诞生。相比传统的基于 Hyperviosr 的虚拟化技术，操作系统虚拟化更加轻量化，其由内核来实现虚拟的操作系统环境的创建，因此效率很高，几乎没有额外开销，是今天容器技术的雏形。

容器技术的发展脉络大致可以分为 1970－2008 年的技术萌芽期、2008－2013 年的技术发展期和 2013 年至今的技术标准期(见图 3)。

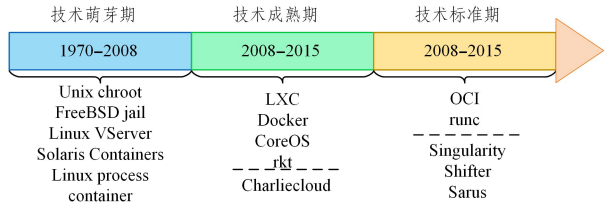

图 3 容器技术发展历程

Fig. 3 Development history of container technology

(1)容器技术萌芽期(1970－2008 年)

1970－2008 年属于容器技术萌芽期，在这个阶段许多操作系统都尝试过添加系统隔离的机制。

早在 1970 年，Unix 系统就引入了 chroot 系统调用，用于改变当前进程的根目录。这个系统调用于 1982 合并进了 BSD 分支。此系统调用的创建，表明当时的人们已经有了对

系统资源进行隔离的需求。虽然它只是简单地隔离了进程的根目录,但仍被认为是操作系统虚拟化技术的起点。

2000 年,FreeBSD 加入了 jail 机制,提供了 jail 系统调用和用户态的 jail 程序[12]。Jail 机制允许系统管理员将 FreeBSD 系统划分成几个独立的环境,这些独立的环境被称为"监狱"(jail)。每个 jail 都拥有自己的文件、进程、用户和超级用户等资源。这些资源隔离之后就虚拟出了一个独立的操作系统。

2001 年,Linux 上也出现了 VServer[13]。Vserver 和 jail 类似,也是在内核对系统资源进行隔离,将每个分区称为安全上下文,并提供了一个类似于 chroot 的实用程序,用于进入安全上下文。它是需要对 Linux 内核进行额外修改的系统级虚拟化技术。

2004 年,Solaris 系统也实现了操作系统级的虚拟化支持,称为 Solaris Containers。Solaris Containers 是资源控制和隔离的组合,一个操作系统可以分出许多独立的 zone,每个 zone 都可以作为完全隔离的虚拟服务器[14]。

2006 年,谷歌开始在 Linux 上开发 process container。这项工作实现了限制、统计和隔离进程集合的资源使用(如 CPU、内存、磁盘 I/O、网络),后来在 Linux 2.6.24 版本中改名为 cgroup,并合入内核主线[15]。

(2)容器技术成熟期(2008－2015 年)

2008－2015 年属于容器技术成熟期,在此阶段 Linux 内核已经能提供内置的资源隔离和资源控制的机制,通用的容器软件开始出现,各商业公司也纷纷推出了自己的容器实现。

2008 年,Linux containers(LXC)发布,这是第一个依赖 Linux 原生内核的完整容器管理实现,其底层使用了内核提供的命名空间[16]和 cgroup[17]机制。命名空间对资源进行隔离,cgroup 对资源进行控制。

2013 年,Docker 作为云计算公司 dotCloud 的开源项目被公开。Docker 基于 LXC 做资源隔离和限制,并且加入写时复制的文件系统作为镜像。这些工具的结合带来了封装应用的全新方式。DevOps[18]、系统管理员、开发人员都开始使用容器做环境的建设、封装和部署。Docker 带来的最大益处就是解决了环境依赖问题,它可以将软件的依赖打包进镜像,然后迅速、便利地移植到其他机器中。而后 CoreOS[19]应运而生,它是专门用于容器环境的轻量级 Linux 发行版。

CoreOS 和 Docker 的公司在商业上的竞争使得 CoreOS 推出了自己的容器引擎 rkt[20](全称 rocket)。rkt 和 Docker 类似,主要运用于软件部署,帮助开发者将应用和依赖包打包到可移植容器中。虽然 Docker 几乎成为了行业的事实标准,但许多公司出于商业上的考虑宁愿自研或二次开发新的容器引擎,也不愿意完全依赖于 Docker。这一定程度上阻碍了整个容器技术的推广。

(3)容器技术标准期(2015－2021 年)

2015－2021 年是容器技术标准期,在此阶段各公司和机构从之前的竞争和合作中寻找平衡,从而推动了标准规范的诞生。

2015 年,在 Docker 公司的牵头下,Linux 基金会启动了开放容器计划(Open Container Initiative,OCI),OCI 旨在围绕容器成立开放的工业标准,解决容器的构建、分发和运行问题[21]。OCI 包含 3 个规范:运行时规范(OCI Runtime Specification)、镜像格式规范(OCI Image Format)和镜像分发规范(OCI Distribution Specification)。Docker 公司从自己的容器引擎实现抽象出 runc,并把它贡献给 OCI 作为标准的容器运行时参考实现。runc 为容器实例的创建提供了所有的基础功能,后续的一些容器都依赖于 runc 作为底层的运行时[22]。

随着容器技术的流行,高性能计算系统环境的用户也开始尝试使用容器技术来解决日益复杂的软件依赖问题。但 Docker 的运行模式更加适合微服务的形式,它的安全模型允许受信任的用户执行任何操作,这在高性能计算传统的多用户环境下是不可接受的[4],于是一些欧美超算中心开始着手开发适用于高性能计算系统的容器软件。

2014 年[1],美国洛斯阿拉莫斯国家实验室(LANL)开始研发 Charliecloud。Charliecloud 利用命名空间实现了资源的隔离,它依赖于 Docker,但是不需要特权操作,其使用简单的方式即满足了 HPC 对用户自定义软件(UDSS)的需求。

2015 年,美国国家能源研究科学计算中心(NERSC,是美国劳伦斯伯克利国家实验室为美国能源部科学办公室运营的高性能计算组织)开始研发 Shifter。资源的隔离基于 Linux 提供的命名空间。Shifter 有个网关进程从仓库拉取 docker 的镜像,并转换成适合 HPC 的格式,比如 squashfs。Shifter 在 NERSC 计算中心内部的集群上已经广泛使用。

同年,美国劳伦斯伯克利国家实验室(LBNL)开发了 Singularity。这是专门为 HPC 环境开发的容器软件,对高速互联网络、加速卡和 MPI 库都有专门的支持。它的安全模型可保持容器内外用户一致,而且兼容 Docker 的容器格式。在学术界尤其是各个超算中心都对 Singularity 有着极大的兴趣,顶级的 HPC 中心,如美国的德克萨斯高级计算中心、圣地亚哥超级计算机中心和橡树岭国家实验室,都已经在集群上安装了 Singularity。

2018 年,瑞士国家超级计算中心(SCSC)研发了 Sarus 容器软件。Sarus 也主要用于 HPC 工作场景,它为多用户使用提供了安全性,它的容器文件系统为并行存储量身定制,并且能够与作业管理系统集成。Sarus 与 OCI 镜像兼容,并且提供了用户范围的镜像管理。

从目前来看,操作系统虚拟化的容器技术已成为一种流行的开发、测试、部署应用的方式,它们允许应用程序拥有自己完整的软件堆栈,甚至是操作系统发行版。许多研究表明,如果操作正确,容器几乎不会引入性能损失[23-29]。这更加推进了容器在高性能计算环境的普及。

容器运行时包含 3 个要素:

(1)经过内核隔离的进程(或协作进程组);

(2)某些内核资源的独立视图;

(3)独立的根文件系统。

相比传统的虚拟机(见图 4),容器只依靠宿主机内核

[1] Charliecloud,Shifter,Singularity 和 Sarus 的年份以 git 的提交记录的最早时间为准

提供的虚拟化功能，并且通常都会集成一个容器引擎与内核交互，用于初始化、管理和分配容器。容器运行的应用程序有自己的根文件系统，但与宿主操作系统共享内核。与虚拟机相比，容器技术无须运行 Guest OS，其以进程的形式直接运行在 Host OS 上，从而减小了运行开销，节约了硬件资源。同时，容器内进程对宿主机可见，便于统一管理。容器允许用户使用不同的软件、库，甚至不同的 Linux 发行版，而无须重新安装系统。

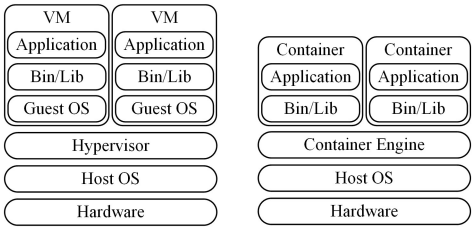

图 4 虚拟机 vs. 容器

Fig. 4 Virtual machine vs. container

3 容器核心技术

本节将介绍 Linux 系统对容器软件实现提供的底层系统机制。如图 5 所示，通常的容器软件都使用命名空间做资源隔离，用 cgroup 做资源控制，并且都会将根文件系统 rootfs 存储到各自定义的镜像中。

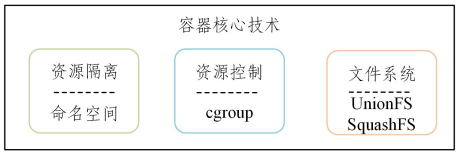

图 5 容器核心技术

Fig. 5 Core technology of container

3.1 资源隔离技术

命名空间是 Linux 内核支持的特性，它以进程为单位对一部分内核的资源进行隔离，实现了一种抽象，在同一个命名空间下的进程被认为是独占了对应的隔离的资源。这些资源对其他的命名空间不可见，对资源的修改只影响同一命名空间下的进程。这些资源包括进程 ID、主机名、用户 ID、文件系统挂载、网络通信和进程间通信。

Linux 上命名空间机制受到贝尔实验室开发的分布式操作系统 plan 9 的启发，并从 2.4.19 版本加入内核。随着更新，内核支持越来越多的命名空间子系统。各个类型的命名空间进入内核的版本如表 1 所列，其中用户命名空间在 3.8 版本被重新实现。

表 1 Linux 命名空间

Table 1 Linux namespace

Namespace 类型	引入内核版本	隔离的资源
Mount namespace	2.4.19	文件系统挂载点
UTS namespace	2.6.19	主机名、域名
IPC namespace	2.6.19	SystemV IPC 和 POSIX 消息队列
PID namespace	2.6.14	进程号
Network namespace	2.6.24	网络设备、协议栈、端口等网络资源
User namespace	2.6.23 & 3.8	用户 ID 和组 ID

每个进程在 proc 文件系统下都会有个子目录/proc/[pid]/ns/，里面存储了命名空间相关的信息。这个目录下每个文件都是符号链接，指向的虚拟文件指代当前进程所在的命名空间，这也符合 Unix 里一切皆文件的设计理念。如果两个进程在同一个命名空间下，那么它们的/proc/[pid]/ns/文件夹下对应的命名空间符号链接指向的虚拟文件的设备号和 inode 号相同。虚拟文件的命名方式为“命名空间类型:inode 号”。

(1)挂载(Mount)命名空间[30]

挂载命名空间主要用于文件系统挂载点的隔离。在创建新的挂载命名空间时，当前命名空间的挂载点都会被复制，但之后对挂载点的操作不再相互传递。有时需要让一些挂载点在命名空间之间传播，可以使用文件系统的共享子树机制。

容器软件可以通过开启挂载命名空间，并调用 pivot_root 改变 rootfs，实现文件系统根目录的隔离。

(2)UTS 命名空间[31]

UTS 命名空间用于主机名和域名的隔离。容器可以通过该命名空间定义新的主机名或域名，主要用于服务之间的通信。

(3)IPC 命名空间[32]

IPC 是 Linux 提供的进程间通信(Inter-Process Communication)的机制。IPC 命名空间用于 System V IPC 对象和 POSIX 消息队列的隔离。这些 IPC 都跨进程创建全局标识符，以便两个进程通过标识符进行通信。IPC 命名空间隔离这些标识符，只有一个命名空间下的进程允许通过 System V IPC 和 POSXI 消息队列进行通信。

容器可以通过创建新的 IPC 命名空间对运行的进程加以限制，使它们只能通过一组特定的 IPC 资源和同一个容器实例内的进程进行通信，不允许干扰其他容器和主机中的进程。

(4)进程号(PID)命名空间[33]

进程号命名空间主要用于隔离进程标识。进程号命名空间是嵌套的，这意味着当创建一个新进程时，从其当前进程号命名空间到初始进程号命名空间，这个进程都有一个独立的进程号。因此，在初始进程号命名空间能够看到所有进程，尽管它们的进程号与在其他命名空间内看到的不同。创建新的进程号命名空间时，第一个进程的进程号为 1。它与 init 进程类似，比如会接受命名空间下的孤儿进程、屏蔽信号等。

通过将容器中运行的进程纳入独立的 PID 命名空间中，可以限制容器和容器、容器和宿主机之间进程的可见性，从而实现进程隔离。

(5)网络(Network)命名空间[34]

网络命名空间会对系统的网络资源进行隔离，包括网络设备、协议栈、路由表、防火墙规则、端口号等。每个新创建的网络命名空间都只有一个本地环回设备。一个物理设备只能存在于一个网络命名空间，可以通过添加虚拟网络设备，实现跨容器、跨主机的网络通信。容器可以通过网络命名空间给每一个实例创建一个独立的网络栈，从而使每个实例都有自己的 IP 地址、IP 路由表、网络设备等，这使得容器可以通过各自的网络接口与外部进行通信，并且无法窃听或操纵其他容器或主机的网络流量。

(6)用户(User)命名空间[35]

用户命名空间用于隔离用户和组 ID，其核心在于权限的

隔离。利用用户命名空间能够安全地实现非特权进程使用传统的特权功能。与进程号命名空间一样，用户命名空间是嵌套的，每个新的用户命名空间都被视为创建它的用户命名空间的子项。新用户命名空间中的第一个进程拥有新命名空间中的所有能力，但它在父用户命名空间中的能力为 0。因此，用户可以任意操纵新用户命名空间内的进程和内核资源，比如无特权地开启其他命名空间，这个特性可以用于支持非特权用户启动容器。

在创建用户命名空间后，在/proc/[PID]/uid_map 和/proc/[PID]/gid_map 这两个文件写入配置，可以实现父子用户命名空间之间用户和组 ID 的映射。该映射是双向的，当在用户命名空间内部发起基于 UID 的访问决策时，会将子用户命名空间内的 UID 转换为外部父命名空间对应的 UID，并将转换后的 UID 用于访问控制。将任何目录绑定挂载到容器中都是安全的，因为控制访问由用户在主机上的真实的、无特权的 ID 决定，而不是新用户命名空间中的虚拟 ID。

和命名空间相关的系统调用有 3 个，分别是 clone[36]，setns[37]和 unshare[38]。

clone 和 fork 类似，都是用来复制 Liux 内核中的 task_struct 结构体，生成一个子进程。clone 通过 flags 参数，能够更加精细地控制复制后修改的数据。常见的用法是更精细地控制子进程和父进程之间共享的资源，比如通过共享虚地址空间、文件描述符表等来实现线程。同样，命名空间的创建也可以通过 falgs 来控制，分别对应创建的命名空间。创建出的子进程处于新的命名空间中，它的子孙也默认在这个命名空间里。

unshare 和 clone 类似，目的是让一个进程无须创建子进程就实现执行上下文的分离。如果 flags 是 CLONE_NEW*，那么相应的命名空间就会被内核创建，当前进程自动加入新创建的命名空间。

setns 是为了命名空间专门新增的系统调用，它让调用的线程加入其他的命名空间，通过指向目录/proc/[pid]/ns/下的符号链接的文件描述符来指定加入的命名空间。

除了创建用户命名空间外，其余命名空间的创建都需要有 CAP_SYS_ADMIN 能力。使用 setns 加入命名空间同样需要 CAP_SYS_ADMIN 能力。容器启动时都会创建命名空间，因此也需要 CAP_SYS_ADMIN 能力。为了允许非特权用户启动容器，一般有 3 种方式：一是让容器程序文件所有者为 root 用户，并设置 SETUID 位；二是创建用户命名空间来获取临时的 CAP_SYS_ADMIN 能力；三是通过 C/S 架构，请求拥有 root 权限的守护进程执行特权操作。

3.2 资源控制技术

Linux cgroup 是一种限制进程资源的机制。cgroup 最早是从上文提到的 process container[15]演化而来，当时它作为 cgroup v1 合入了 Linux 2.6.24。随着时间的推移，已经添加了各种 cgroup 控制器，以允许管理各种类型的资源，然而这些控制器和 cgroup 层次结构的管理之间产生了许多不一致。于是 cgroup v2 得以逐步开发，并在 Linux 4.5 上正式发布。目前，cgroups v2 只实现了 cgroups v1 中可用控制器的一个子集。

cgroup 的控制单位是进程集合。如图 6 所示，主要提供了以下功能特性：

(1)资源限制。限制进程允许使用的特定资源，如内存或 CPU 的数量。

(2)优先级。允许一部分进程更优先使用需要竞争的资源，如磁盘和网卡等设备。

(3)统计。监视和报告资源使用情况。

(4)控制。统一控制 cgroup 中所有进程的状态(比如启动或停止)。

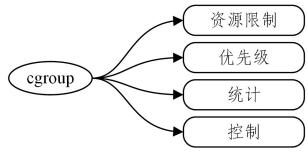

图 6 cgroup 资源控制技术

Fig. 6 cgroup resource control technology

容器可以利用 cgroup 来管理一个实例内进程的资源使用情况。例如控制资源量(CPU、内存和磁盘 I/O 等)，确保每个容器获得公平的资源份额，防止某个容器消耗所有资源。还可以通过设备白名单控制器限制容器实例允许访问的设备集，从而避免访问一些重要的设备节点对宿主机的潜在影响。

3.3 容器文件系统

容器的一个很重要的特性就是在运行时给容器内的进程提供一个与宿主机操作系统隔离的 rootfs，包含应用程序所需要的代码、库、工具、依赖项、配置等运行所需的文件。容器软件都采用了镜像来提供完整且封装的容器环境，而容器环境的核心就是 rootfs。除此之外，如图 7 所示，镜像中还会有诸如签名验证、环境变量等其他信息。启动容器实例时，容器引擎会将这个 rootfs 挂载到 Host OS 的文件系统上，然后通过 chroot 或者 pivot_root 这两个系统调用，将容器的根目录切换到镜像里 rootfs 的挂载点。镜像对 rootfs 的存储方式一般分为两类，一种是以 Docker 为代表的分层式存储，默认依赖于联合文件系统(UnionFS[39])；还有一种是以 Singularity 为代表的单文件式存储，默认依赖于 SquashFS[40]。

图 7 容器镜像的组成

Fig. 7 Composition of container image

(1)UinonFS

UnionFS 是一类文件系统的统称。UnionFS 不会直接操作硬盘去存储数据，而是通常依赖于其他文件系统(如 ext4)来完成存储操作。它是可堆叠的统一文件系统，能够合并多个目录(分支)的内容，将多个目录挂载到一个挂载点，同时保持它们的物理内容为分开的状态。这些目录在挂载时需要设置是否可读写。针对只读目录进行操作时，修改文件会导致文件被复制到可写目录中，并对这个副本进行修改；删除文件也会在可写目录中留下删除的标记。

Docker 最开始就参考和利用了 UnionFS 分层的技术来实现容器的镜像。如图 8 所示，构成 Docker 镜像的每个文件都称为层，每一层都是目录的压缩包。这些层形成了一系列中间图像，其中每一层都依赖于紧邻其下方的层，这个层级

结构是 Docker 镜像概念的关键核心。在容器启动时，Docker 引擎会利用 UnionFS 将这些层挂载到一个挂载点，并将它作为启动后容器的 rootfs。

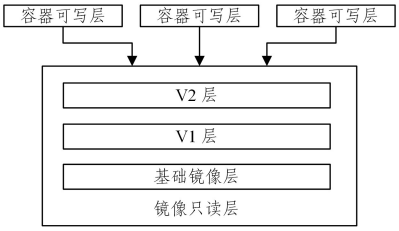

图 8 Docker 镜像分层

Fig. 8 Docker image layers

在通过如 Dockfile 的定义文件构建镜像时，最底层通常是 Linux 发行版的基础镜像，构建过程中会不可避免地对基础的文件系统产生修改。这些修改不会直接原地生效，而是在最高层记录与下层的差异(diff)。将这些层的堆叠关系记录下来，就构成了一个新的镜像。

在运行时，容器可以针对镜像启动一个或多个实例。启动后，容器引擎会在镜像之上给每个实例创建一个可写层，或者叫容器层；原有的镜像被称为镜像层。容器实例运行过程中，对镜像层的文件系统做任何更改都只在容器层记录，这样能保证任意数量的同类容器实例都共享对相同底层镜像的访问，同时保持自己的独立文件系统视图。容器层默认情况下不做持久存储，一旦应用程序运行完成，它就会被丢弃。这一事实给运行应用程序(如数据库、日志)带来了问题。一般的做法是将宿主机系统内的某些文件或者目录挂载到容器里，容器对文件的修改操作由此可以传播到宿主机系统上，并实现持久存储。当然，用户也可以选择将临时的可写层保留下来，这就相当于构建了一个新的镜像。

Docker 中管理镜像层和容器层内容的组件被称为存储驱动程序。这里有多种存储驱动器可供选择，它们的实现各不相同，适用于不同的工作负载和应用场景，但几乎所有的驱动程序都采用了某一个 UnionFS 来实现可堆叠的镜像层和写时复制的策略。比较常用的有 aufs 和 overlay2 存储驱动器，它们分别是基于联合文件系统 AUFS 和 OverlayFS 来实现的[41]。在 Docker 18.06 版本之前，默认的存储驱动程序是 aufs，之后的版本默认使用 overlay2。

(2)SquashFS

SquashFS 是 Linux 内置支持的一个只读文件系统，可以压缩文件、inode 和目录，将它们打包成一个连续的字节段，并写入其他设备、分区或普通文件。SquashFS 一般用于存储各个 Linux 的 Live CD 或嵌入式系统的固件。以 Singularity 为代表的容器基于 SquashFS 来存储 rootfs。构建镜像时，会将容器运行时的 rootfs 压缩成 SquashFS 格式的字节流，并将这部分嵌入到镜像里。容器运行镜像时，容器引擎默认让内核将这个 SquashFS 文件挂载，容器进程最终会将自己的 rootfs 切换到这个挂载点。

(3)优势分析

使用 UnionFS 将镜像分层存储和 SquashFS 单个文件存储各有各的优势。UnionFS 分层存储复用了基础镜像，节省了存储的磁盘空间；构建时只生成新的镜像层，节省了计算工作量；上传和分发时能够跳过云端或本地已有的镜像层，节省了网络带宽。而 SquashFS 单个图像文件使得环境的移植更加便利，该文件可以被复制、共享、存档，用户也可以使用标准的 UNIX 文件权限来管理自己的镜像。

4 现有高性能计算系统的容器实现

Docker 是目前最流行的容器技术，普遍适用于微服务和云计算平台。Docker 是个 C/S 架构的软件。Docker daemon 以 root 权限运行，以便拥有管理适当资源所需的访问权限。如果系统上存在名为 docker 的组，Docker 会将套接字的所有权应用于该组。因此，属于 docker 组的任何用户都可以运行 Docker，而无须获得系统管理员权限。Docker 为了保持各个容器实例的独立性，采取了十分严格的资源隔离和控制的策略，默认启用全部的命名空间和 cgroup 功能。

虽然 Docker 提供了很多特性，但是它更适用于服务进程而不是计算程序的进程，而不适用于高性能计算系统的软硬件环境。Docker 需要在高性能计算系统的每个节点上有一个以系统管理员权限运行的 Dockerdaemon，运行的容器的 UID 也默认为 root。虽然能够采用用户命名空间来改变这个策略，但是用户理论上可以强制守护进程授予用户升级权限。此外，每个 Docker 引擎在本地都有自己的镜像存储，无法共享。假如一个作业在 100 个节点上启用了容器，那么依赖的镜像需要复制 100 次，每个节点一个[42]。Docker 默认的强隔离性使得进程无法充分利用硬件资源进行加速和互联。Docker 早期对 MPI 的支持和作业管理系统的集成也都不够完善。

为了解决 Docker 在高性能计算环境中存在的一些问题，同时利用容器技术满足自定义软件栈的需求，适用于高性能计算环境的容器实现相继出现，主要有 Singularity，Shifter，Charliecloud 和 Sarus。

4.1 Singularity

2015 年，美国劳伦斯伯克利国家实验室(LBNL)开发了 Singularity 的初始版本。Singularity 放弃了许多 Docker 支持的容器特性，专注于实现软件的可移植性。Singularity 的常用工作方式如图 9 所示，用户可以自己构建镜像，并允许交互式修改。之后将镜像传输到运行环境上，用户可以以无特权的方式运行镜像中的程序。

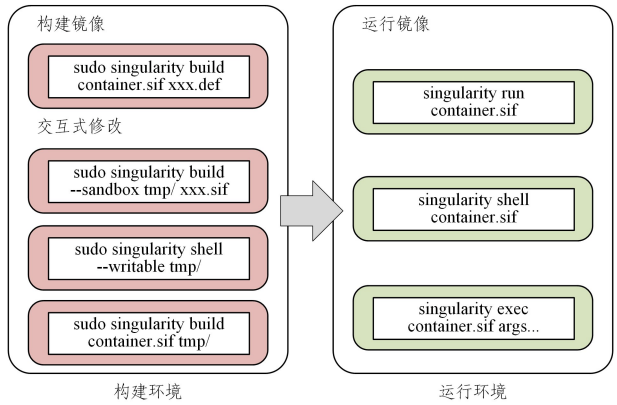

图 9 Singularity 工作流程

Fig. 9 Singularity workflow

Singularity 通过将整个程序和依赖都封装到单个镜像

文件中来实现快速移植。Singularity 的工作流程基本独立，不依赖于 Docker 的组件，但为了利用 Docker 庞大的生态，Singularity 支持拉取 dockerhub 的镜像，并转换成其专有的镜像格式。Singularity 也有自己的官方镜像库 Singularity-hub。Singularity 没有采用 Docker 那样复杂的 C/S 架构，而是把启动容器、拉取镜像、创建镜像等核心功能都放进了命令行的客户端里实现。Singularity 默认使用 SETUID 来实现特权操作。为了在没有 root 权限的机器上安装并正常使用，从 3.5 版本开始，Singularity 采用用户命名空间实现了 fakeroot 特性，可以让非特权用户执行一些受限的特权操作，启动容器后，容器内进程的 UID 保持不变。Singularity 利用内核的 NO_NEW_PRIV 标志，禁止普通用户通过 sudo 启动容器从而具备系统管理员权限，避免了越权的安全隐患。

高性能计算系统中的工作流程不需要对容器进行严格的隔离。在基于作业管理系统的传统成熟工作模式下，不同用户的作业都会直接作为一系列进程在节点上运行。利用传统的 Linux 权限模型就足以保障用户各自的文件和进程的安全，并维持整个系统环境的正常运作。Singularity 的核心功能就是赋予容器独立的文件系统视图，从而解决复杂依赖问题，因此容器引擎在启动容器进程时，会默认创建新的挂载命名空间。Singularity 默认不启用 cgroup 做资源限制，因为全局的作业管理系统会统筹规划每个用户的作业，包括限制作业可使用的资源，不需要在容器这一层面再做重复的工作。为了让容器运行时在高性能计算系统上能够正常工作，Singularity 还需要额外为容器提供一些资源，比如默认会挂载用户的家目录，从而允许封装的程序读取输入文件和持久化地保存计算结果；一些设备也需要挂载到容器供其运行时使用，典型的有高速互联网络和加速卡。

Singularity 针对高性能计算系统做了很多支持，并且单文件的镜像软件也给移植带来了便利，现已成为现在高性能环境上最流行的容器软件。

4.2 Shifter

Shifter 是美国国家能源研究科学计算中心（NERSC）和 Cray 联合开发的容器软件。为了利用 Docker 庞大的生态，Shifter 复用了 Docker 在镜像创建和镜像分发相关的组件，自研了容器引擎并自定义了镜像的格式。

Shifter 的架构比 Docker 简单，容器命令行客户端是 SETUID 的程序，以此来允许用户特权操作。Shifter 有以下重要组件：命令行用户工具、镜像网关、称为“udiRoot”的运行时启动器和工作负载管理器集成（WLM）组件。

Shifter 的工作流程如图 10 所示。用户命令行工具根据用户的选项来和其他组件交互。网关用于接受命令行工具发来的 restful 请求，然后从各种来源（包括私有 Docker 仓库、公开的 Dockerhub 和本地文件系统）导入镜像，将镜像转化成 udiRoot 组件可以使用的格式，最后把转化后的镜像传输到与目标 HPC 关联的全局文件系统中。镜像文件基于 Squashfs 格式。UdiRoot 组件用于实例化和销毁容器。Shifter 和 Singularity 的思路理念一致，核心也是给容器内的进程构建一个独立的根文件系统视图，因此也不需要隔离和限制很多的资源，它将资源管理工作都交给了系统上运行的作业管理系统，如 SLURM，ALPS 等。启动后在默认情况下，文件系统视图的根目录是挂载的 SquashFS，因此镜像内的文件和目录都呈现只读的权限，用户也需要额外挂载 host 上的目录来实现文件输入或者持久化存储。这个挂载的 SquashFS 一般都会在分布式文件系统的某个目录中，而且是只读，以此可以允许多个容器进程的 rootfs 共享，这在运行 MPI 这类并行程序的使用场景下能够节省运行时的存储开销。工作负载管理器集成（WLM）组件用于与 Slurm 等工作管理系统集成，例如上文提到的作业中多个进程的 rootfs 共享宿主机文件系统下的目录，就需要 WLM 的支持。

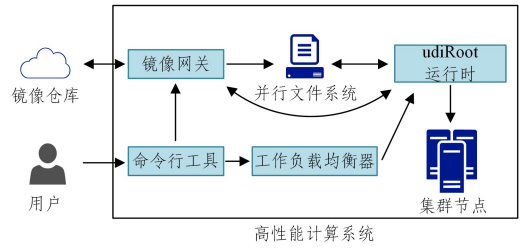

图 10 Shifter 工作流程[43]

Fig. 10 Shifter workflow

4.3 Charliecloud

Charliecloud 是美国洛斯阿拉莫斯国家实验室（LANL）研发的容器软件。Charliecloud 是一个轻量级的容器实现，它依赖于 Docker 实现镜像管理，只独立实现了核心的容器运行时。Charliecloud 和之前分析过的 Singularity 以及 Shifter 一样，默认只创建新的挂载命名空间，资源控制也交给上层的作业管理系统完成。

由于 Charliecloud 的轻量实现，它的工作模式需要依赖用户额外的机器和 Docker。它的工作流程如图 11 所示。首先，用户需要一台拥有 root 权限的机器，在此机器上利用 Docker 完成前期的准备工作，即通过 Docker 构建或者拉取镜像；然后利用 Docker 自带的 export 命令导出镜像，导出的格式为 tarball 压缩包；并利用网络将镜像的压缩包从自己的机器传输到高性能计算机器上。之后，用户在超算集群的登陆节点上操作。为了启动容器，用户需要解压镜像压缩包，通常解压到 tmpFS 或者共享文件系统上，解压之后是一个目录，这个目录就是后续启动容器的 rootfs；然后利用容器引擎启动进程。容器引擎会创建新的挂载命名空间，绑定挂载主机上必要的目录，使用 pivot_root 将容器 rootfs 更改为镜像解压后的目录，最后执行命令给定的程序。这里创建挂载命名空间是特权操作，Charliecloud 通过启用用户命名空间，让用户能够无需特权完成这项操作。

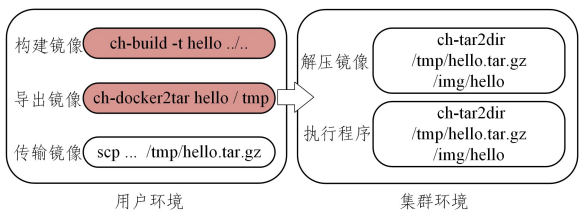

图 11 Charliecloud 工作流程

Fig. 11 Charliecloud workflow

Charliecloud 巧妙地将一部分工作转移到高性能计算系统之外的机器上，从而避免了对 Docker 的依赖。它使用了

用户命名空间来无特权启动容器。用户命名空间需要 Linux 内核为 3.8 以上，因此 Charliecloud 对一些老的高性能计算系统支持不够好。

4.4 Sarus

Sarus 是瑞士国家超级计算中心(CSCS)2018 年研发的容器软件，它符合 OCI 运行时规范，并能够直接使用 Docker 庞大的镜像仓库。

Sarus 也没有采用 Docker 的 C/S 架构，只简单地分为命令行组件、镜像管理器、运行时组件 3 个部分。Sarus 的工作流如图 12 所示。命令行组件用于收集命令行参数并执行用户请求的操作，一般会调用镜像管理器或运行时组件。镜像管理器负责从 Dockerhub 下载镜像，将其转换成 Sarus 自己的格式，并存储在本地的分布式文件系统上；运行时组件用于启动容器，它会从镜像中提取出 rootfs 目录，把主机系统上必要的目录也挂载到相应的位置，然后生成容器进程。容器进程根据配置创建不同的命名空间，并进行 rootfs 的切换，最后执行相应的命令。

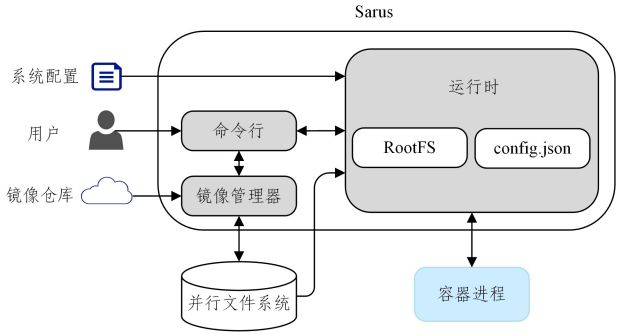

图 12 Sarus 工作流程[44]

Fig. 12 Sarus workflow

Sarus 运行时符合 OCI 标准，可以在容器启动时设置 OCI hook 函数，这些 hook 回调函数将在容器生命周期的不同阶段执行，以拓展容器运行时提供的功能。开发或运维人员可以创建 hook 函数实现新功能，让容器与高性能计算环境的其他基础设施集成。Sarus 利用 hook 拓展了许多功能：1)为了保证 MPI 程序运行时容器内和宿主机上的库 ABI 兼容，并且有效利用 MPI 库针对集群硬件调优后的性能优势，Sarus 可以将容器内的 MPI 库替换成主机系统上的版本；2)Sarus 集成了 NVIDIA container runtime 中的 hook，会在容器中挂载主机上的 GPU 设备、cuda 运行库依赖等，保证了 CUDA 程序的正常执行；3)为了支持 Apache Spark 这类需要 ssh 服务的程序，Sarus 提供了一个在容器内提供 OpenSSH 软件栈的 hook；4)为了防止特殊情况下 Slurm 分配的资源不可用而导致进程启动不同步，Sarus 还提供了一个 hook，可以在对用户程序非侵入的情况下提供同步屏障。

4.5 小结

这些容器软件都能够实现自定义软件栈，解决了程序复杂依赖的问题。它们都能够满足高性能计算系统的特殊需求，但采取了不同的实现方式。

表 2 列出了 4 款容器默认配置下的特性，可以看出，它们之间存在一些共性和差异。它们需要在多用户的环境下杜绝特权滥用，必须避免系统管理员权限的守护进程来包揽一切操作，但又得允许普通用户完成诸如创建命名空间、挂载主机目录等必要的特权操作，因此几乎都选择了使用 SETUID 或用户命名空间来遵循最小特权原则，容器实例内的用户与主机上保持一致。其中，Singularity 默认采用 SETUID，但也允许通过配置使用用户命名空间；Shifter 和 Sarus 采用了 SETUID；Charliecloud 采用了用户命名空间。它们都采取了宽松的资源隔离策略，默认只创建挂载命名空间来隔离挂载点，保证进程对高性能计算系统软硬件资源的正常使用，如使用 GPU 加速计算卡、绑定集群上的文件系统目录、访问高速互连网络等。4 款容器都支持与作业管理系统集成，同时也默认将容器内进程的资源控制交由作业管理系统负责。Singularity 和 Sarus 支持通过配置启用 cgroup。为了复用 Docker 的生态，它们都支持从 Docker 仓库拉取镜像，其中，Charliecloud 拉取镜像依赖于 Docker 本身，镜像格式也与 Docker 一致。其余的 3 个容器都会将 Docker 镜像转换成自定义的基于 SquashFS 存储根目录的格式，Singularity 还支持从自定义的脚本构建镜像。Sarus 开发时 OCI 规范已经落地，因此它兼容 OCI 运行时规范，但不兼容镜像规范；Singularity 从 3.1.0 版本开始支持 OCI 运行时规范。

表 2 4 款容器实现的比较总结

Table 2 Comparison of four container implementations

	Singularity	Shifter	Charliecloud	Sarus
特权模式	SUID/UserNS	SUID	UserNS	SUID
创建的命名空间	MountNS/UserNS	MountNS	MountNS/UserNS	MountNS
Cgroup 资源控制	可选	不支持	不支持	可选
构建镜像	支持	不支持	不支持	不支持
镜像文件系统	SquashFS	SquashFS	UnionFS	SquashFS
OCI 兼容	兼容 OCI runtime 规范	不兼容	不兼容	兼容 OCI runtime 规范

5 总结与展望

5.1 容器技术在高性能计算系统的应用现状

随着容器技术的发展，很多超算中心或者超算云的厂商已经在它们的高性能计算系统上支持了容器软件。美国国家能源研究科学计算中心(NERSC)支持在超算上使用 Shifter。根据他们 2014 年[45]和 2018 年[46]的系统运行统计报告，容器的使用量从 2014 年的 1%增加到了 2018 年的 8%。美国国家航空航天局(NASA)的超级计算中心在机器上加入了 Singularity 模块[47]；上海交通大学的校级计算公共服务平台也部署了 Singularity[1)]；亚马逊公司的云服务 AWS 研发了超算云，利用脚本 AWS ParallelCluster[48]快速创建高性能计算集群，并提供了不同的容器运行时[49]，支持 runc，Singularity 和 Sarus。阿里云的弹性高性能计算 E-HPC[50]平台也支持 Singularity 来完成高性能计算作业[51]。

可以看到，容器软件在高性能计算环境中已经得到了

1) https://docs.hpc.sjtu.edu.cn/container/index.html

一定程度的普及，但是根据 NERSC 公开的统计数据来看，使用率依然不够高[46]，分析其原因，我们认为有如下几点：

(1)容器构建的学习成本和操作成本较高

基于容器运行程序和直接运行程序差别不大，用户可以很快适应；相比之下，构建自定义的容器镜像需要一定的学习门槛，其过程相当于所有的软件包重新安装一遍，也需要一定的操作成本。尤其在高性能计算系统中，高速网络、加速卡、并行编程环境等，都给用户自主构建容器镜像造成了困难。镜像仓库可以提供丰富的、常用的应用软件栈镜像，优质的应用镜像虽然方便用户使用，但是仍然无法满足多元化的用户需求，同时仓库中的镜像质量参差不齐，也可能存在安全隐患。这些都成为容器应用困境的重要因素。

(2)兼容性和容器可移植性的两难问题

以多数高性能计算程序依赖 MPI[52]为例，MPI 程序启动后会利用 orted 守护进程来统筹其余工作进程[53]。Orted 一般位于集群主机上，而工作进程是容器启动的实例，它们之间会通过 PMI(Process Management Interface)[54]进行通信。如果集群上的 MPI 库和容器内的 MPI 库不兼容，就会导致整个程序执行错误。一种解决方案是挂载集群文件系统上的 MPI 库到容器实例内，但不同的高性能计算系统上 MPI 版本不尽相同，会对容器镜像的可移植性造成一定限制。

(3)容器镜像占用存储空间较大

一个只包含操作系统的基础容器镜像约 70 MB，应用于 ATLAS 等业务流程封装形成的 CVMFS 镜像文件 3.5 TB[55]。在高性能计算系统中，用户存储空间往往会被设定一定的限额，因此较大的存储空间占用也为容器应用推广造成了一定的困难。

5.2 未来工作展望

(1)面向国产计算系统的 DevOps 工具链

未来工作将面向国产异构计算系统开发适用高性能计算应用研发和部署的 DevOps 工具链，方便开发者构建、测试和部署高性能计算应用程序。开发者提交代码到仓库时，可以利用一些无特权的方式在高性能计算系统上自动构建容器[56]。测试环境需要与实际的集群系统一致，可以选择利用一些测试节点，或者通过私有的 IaaS 系统创建集群。测试过程中可以通过容器实现软件栈的任意组合，借此找出性能最佳的软件栈，作为当前程序的依赖。管理员也可以借此功能找出性能最优的组合，将其直接配置成集群的默认基础库。部署阶段直接把测试通过的镜像用于执行作业，这种方式可以最大化地利用容器，并且减少在新系统上开发或移植复杂代码的工作量。

(2)面向融合计算的工作流

云服务商已经推出了一些面向较小规模计算的 HPC cloud[57]，高性能计算系统上云可能会给未来高性能计算的商业模式带来影响。AWS 提供的高性能云服务集群，甚至在超算 TOP500 上从 2010 年到 2017 年都榜上有名[58]。2011 年 11 月排名最高，为第 42 名。云服务平台利用 Kubernetes[59]等编排系统进行使用快速、可扩展的服务部署，或者在作业管理系统中直接集成对于容器的控制。因此，高性能计算领域资源和云服务平台资源协同服务于高性能计算以及大数据和人工智能的融合应用成为一种趋势[60]，需要容器技术更好地服务并完善其中的工作流程。

(3)面向大规模应用计算的容器优化

大规模应用计算始终是高性能计算领域追求的目标，随着 E 级计算时代的到来以及异构计算的兴起，大规模应用计算面临更多的挑战。采用容器技术运行大规模应用计算，将便于应用软件栈的自定义构建，但是在容器镜像存储、网络管理[61]、细粒度计算资源管理、面向应用级容错的运行时迁移等方面都还需要开展进一步的研究和深度优化工作，借助容器技术的优势最大化地发挥计算资源的能力。

结束语 本文总结了容器技术的发展历史，介绍了 Linux 系统对容器技术的底层支持，并分析了高性能计算系统上常见的容器实现，最后思考了容器技术在高性能计算领域面临的问题和未来发展方向。容器技术在高性能计算系统上正在逐渐成熟，未来容器可能会改变高性能计算软件传统的开发和部署模式，为高性能计算以及大数据与人工智能计算的深度融合提供强有力的技术手段。

参 考 文 献

[1] MERKEL D. Docker: lightweight Linux containers for consistent development and deployment [J]. Linux Journal, 2014, 2014(239): 76-91.

[2] JARAMILLO D, NGUYEN D V, SMART R. Leveraging microservices architecture by using Docker technology [C] // SoutheastCon 2016. 2016: 1-5.

[3] KURTZER G M, SOCHAT V, BAUER M W. Singularity: Scientific containers for mobility of compute [J]. PLOS ONE, 2017, 12(5): e0177459.

[4] JACOBSE N, DOUGLAS M, CANON R, et al. Contain this, unleashing docker for hpc [C] // Proceedings of the Cray User Group. 2015: 33-49.

[5] REID P, TIM R. Charliecloud: unprivileged containers for user-defined software stacks in HPC[C] // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2017: 1-10.

[6] TIWARI D, GUPTA S, ROGERS J, et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation [C] // 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). 2015: 331-342.

[7] LIU J, WU J, PANDA D K. High performance RDMA-based MPI implementation over InfiniBand [J]. International Journal of Parallel Programming, 2004, 32(3): 167-198.

[8] PEARCE M, ZEADALLY S, HUNT R. Virtualization: Issues, security threats, and solutions [J]. ACM Computing Surveys, 2013, 45(2): 1-39.

[9] LAADAN O, NIEH J. Operating system virtualization: practice and experience [C] // Proceedings of the 3rd Annual Haifa Experimental Systems Conference. 2010: 1-12.

[10] POPEK G J, GOLDBERG R P. Formal requirements for virtualizable third generation architectures [J]. Communications of the ACM, 1974, 17(7): 412-421.

[11] BUZEN J P, GAGLIARDI U O. The evolution of virtual machine architecture[C]// National Computer Conference. 1973: 291-299.

[12] KAMP P H, WATSON R N. Jails: Confining the omnipotent root[C]// Proceedings of the 2nd International SANE Conference. 2000: 116-127.

[13] DES LIGNERIS B. Virtualization of Linux based computers: the Linux-VServer project[C]// 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05). 2005: 340-346.

[14] TUCKER A, COMAY D. Solaris Zones: Operating System Support for Server Consolidation[C]// Virtual Machine Research and Technology Symposium. 2004: 241-254.

[15] MENAGE P B. Adding generic process containers to the linux kernel[C]// Proceedings of the Linux Symposium. 2007: 45-57.

[16] MICHAEL K. namespaces(7) — Linux manual page [EB/OL]. (2021-08-27) [2021-12-27]. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[17] MICHAEL K. cgroups(7)—Linux manual page [EB/OL]. (2021-08-27) [2021-12-27]. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[18] LEITE L, ROCHA C, KON F, et al. A survey of DevOps concepts and challenges [J]. ACM Computing Surveys (CSUR), 2019, 52(6): 1-35.

[19] JAKE E. CoreOS: A different kind of Linux distribution [EB/OL]. (2014-04-09) [2021-12-27]. https://lwn.net/Articles/593928/.

[20] NATHAN W. The Rocket containerization system [EB/OL]. (2014-12-03) [2021-12-27]. https://lwn.net/Articles/624349/.

[21] DEBAB R, HIDOUCI W K. Containers Runtimes War: A Comparative Study[C]// Proceedings of the Future Technologies Conference. 2020: 135-161.

[22] KUMAR R, THANGARAJU B. Performance analysis between RunC and kata container runtime[C]// 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). 2020: 1-4.

[23] XAVIER M G, NEVES M V, ROSSI F D, et al. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments[C]// 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. 2013: 233-240.

[24] LIU P N, GUITART J. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study [J]. Journal of Supercomputing, 2021, 77(6): 6273-6312.

[25] ABRAHAM S, PAUL A K, KHAN R I S, et al. On the Use of Containers in High Performance Computing Environments [C]// IEEE 13th International Conference on Cloud Computing (CLOUD). 2020: 284-293.

[26] JAERYUN L, CHAE Y, TAK B. Comparative Analysis of Container for High Performance Computing [J]. Journal of The Korea Society of Computer and Information, 2020, 25(9): 11-20.

[27] TORREZ A, RANDLES T, PRIEDHORSKY R, et al. HPC container runtimes have minimal or no performance impact[C]// 1st IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). 2019: 37-42.

[28] YONG C H, LEE G W, HUH E N. Proposal of Container-Based HPC Structures and Performance Analysis [J]. Journal of Information Processing Systems, 2018, 14(6): 1398-1404.

[29] ZHANG J, LU X Y, PANDA D K, et al. Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds? [C]// 10th International Conference on Utility and Cloud Computing (UCC) / 4th International Conference on Big Data Computing, Applications and Technologies (BDCAT). 2017: 151-160.

[30] MICHAEL K. mount_namespaces(7)—inux manual page [EB/OL]. (2021-08-27) [2021-12-27]. https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.

[31] MICHAEL K. uts_namespaces(7)—Linux manual page [EB/OL]. (2019-11-19) [2021-12-27]. https://man7.org/linux/man-pages/man7/uts_namespaces.7.html.

[32] MICHAEL K. ipc_namespaces(7)—Linux manual page [EB/OL]. (2019-08-02) [2021-12-27]. https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html.

[33] MICHAEL K. pid_namespaces(7)—Linux manual page [EB/OL]. (2020-11-01) [2021-12-27]. https://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[34] MICHAEL K. network_namespaces(7)—Linux manual page [EB/OL]. (2020-06-09) [2021-12-27]. https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[35] MICHAEL K. user_namespaces(7)—Linux manual page [EB/OL]. (2021-08-27) [2021-12-27]. https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[36] MICHAEL K. clone(2)—Linux manual page [EB/OL]. (2021-03-22) [2021-12-27]. https://man7.org/linux/man-pages/man2/clone.2.html.

[37] MICHAEL K. setns(2)—Linux manual page [EB/OL]. (2020-08-13) [2021-12-27]. https://man7.org/linux/man-pages/man2/setns.2.html.

[38] MICHAEL K. unshare(2)—Linux manual page [EB/OL]. (2021-03-22) [2021-12-27]. https://man7.org/linux/man-pages/man2/unshare.2.html.

[39] WRIGHT C P. Kernel korner: unionfs: bringing filesystems together [J]. Linux Journal, 2004, 128: 24-29.

[40] RIOUX P, KIAR G, HUTTON A, et al. Deploying large fixed file datasets with SquashFS and Singularity[C]// Practice and Experience in Advanced Research Computing. 2020: 72-76.

[41] XU Q, AWASTHI M, MALLADI K T, et al. Docker characterization on high performance SSDs[C]// 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2017: 133-134.

[42] VEIGA V S, SIMON M, AZAB A, et al. Evaluation and Benchmarking of Singularity MPI Containers on EU Research e-Infrastructures[C]// 1st IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). 2019: 1-10.

[43] ALBERTO M，LUCAS B，FELIPE A C，et al. Shifter at CSCS—Docker Containers for HPC [EB/OL]. (2018-04-09) [2021-12-27]. http://www.hpcadvisorycouncil.com/events/2018/swiss-workshop/pdf/Monday09April/Madonna_ShifterDockerContainers_Mon_090418.pdf.

[44] BENEDICIC L，CRUZ F A，MADONNA A，et al. Sarus：Highly Scalable Docker Containers for HPC Systems[C]// IEEE International Conference on High Performance Computing，Data，and Analytics. 2019：46-60.

[45] BRIAN A，WAHID B，TINA B，et al. 2014 NERSC Workload Analysis [EB/OL]. (2014-11-05) [2021-12-27]. https://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_v1.1.pdf.

[46] AUSTIN B. NERSC-10 Workload Analysis (Data from 2018) [EB/OL]. (2020-04-01) [2021-12-27]. https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf.

[47] MICHELLE M. Enabling User Defined Software Stacks with Singularity [EB/OL]. (2021-11-15) [2021-12-27]. https://www.nas.nasa.gov/hecc/support/kb/enabling-user-defined-software-stacks-with-singularity_637.html.

[48] FRANCESCO D，MARTINO，ENRICO U，et al. AWS ParallelCluster [EB/OL]. https://github.com/aws/aws-parallelcluster.

[49] CHRISTIAN K. HPC Container Engine State-of-Art [EB/OL]. (2021-02-06) [2021-12-27]. https://containers-on-pcluster.workshop.aws/.

[50] ALIBABA-CLOUD. Elastic High Performance Computing [EB/OL]. https://www.alibabacloud.com/product/ehpc.

[51] ALIBABA-CLOUD. E-HPC：Use high-performance container applications [EB/OL]. https://help.aliyun.com/document_detail/102579.html?spm=5176.21213303.J_6704733920.37.5dab3eda2NILb8&scm=20140722.S_help%40%40%E6%96%87%E6%A1%A3%40%40102579.S_0%2Bos0.ID_102579-RL_singularity-OR_helpmain-V_2-P0_6.

[52] GROPP W，LUSK E，DOSS N，et al. A high-performance，portable implementation of the MPI message passing interface standard [J]. Parallel Computing，1996，22(6)：789-828.

[53] WOFFORD Q，BRIDGES P G，WIDENER P. A Layered Approach for Modular Container Construction and Orchestration in HPC Environments[C]// Proceedings of the 11th Workshop on Scientific Cloud Computing. 2020：1-8.

[54] BALAJI P，BUNTINAS D，GOODELL D，et al. PMI：A scalable parallel process-management interface for extreme-scale systems [C]// European MPI Users' Group Meeting. 2010：31-41.

[55] GERHARDT L，BHIMJI W，CANON S，et al. Shifter：Containers for HPC [J]. Journal of Physics：Conference Series，2017，898：082021.

[56] PRIEDHORSKY R，CANON R S，RANDLES T，et al. Minimizing privilege for building HPC containers[C]// IEEE International Conference on High Performance Computing，Data，and Analytics. 2021：1-14.

[57] NETTO M A，CALHEIROS R N，RODRIGUES E R，et al. HPC cloud for scientific and business applications：taxonomy，vision，and research challenges [J]. ACM Computing Surveys，2018，51(1)：1-29.

[58] TOP500. ORG. Amazon Web Services [EB/OL]. https://www.top500.org/site/50321/.

[59] MEDEL V，RANA O，BAÑARES J Á，et al. Modelling performance & resource management in kubernetes[C]// Proceedings of the 9th International Conference on Utility and Cloud Computing. 2016：257-262.

[60] ZHENG W M. Architecture and evaluation of high-performance computers for processing artificial intelligence applications[J]. Journal of Chongqing University of Posts and Telecommunications(Natural Science Edition)，2021，33(2)：171-175.

[61] ZANG D，YANG Z G，WANG J，et al. Design of High-Performance Container Network Based on Network Interface Card Virtualization[J]. Computer Engineering，2022，48(7)：214-219.

CHEN Yiyang，born in 1997，postgraduate. His main research interests include high performance computing and cloud service.

WANG Xiaoning，born in 1981，Ph.D，associate professor，Ph.D supervisor，master supervisor. Her main research interests include high performance computing，grid computing and cloud service.

（责任编辑：何杨）