第十五章　轻量级虚拟化

第十一章提到，云计算的发展和繁荣离不开虚拟化技术，没有虚拟化技术，云计算模式的成功就无从说起。假设小明想要部署一台 Web 服务器，在现有虚拟化技术的帮助下，他可以向云服务商租用虚拟机，无需自己购买和维护物理主机，复杂和繁琐的日常维护工作也不再需要关注。不过，当租用虚拟机之后，小明仍然需要安装操作系统，配置运行 Web 服务器所需要的环境，之后才能在这台虚拟机上部署 Web 服务器，这些安装和配置的过程依然很繁琐。而当 Web 服务器需要扩容时，小明还得租用更多的虚拟机，然后不断重复上述工作。

那么，有没有办法可以减轻小明的负担，使配置 Web 服务器的工作能够简单高效地完成呢？为了回答这一问题，我们需要首先对虚拟化技术进行回顾。在传统虚拟化技术（即系统虚拟化）中，程序运行在不同的虚拟机中，每个虚拟机中都安装了独立的操作系统，实现了程序之间的隔离。可是，像小明这样的用户部署的服务功能比较单一，并不需要用到操作系统的全部功能，安装完整的操作系统显得过于笨重。而轻量级虚拟化不再需要部署重量级的操作系统，能够在原生的操作系统中直接运行程序。为了实现程序间的隔离，轻量级虚拟化引入了沙盒技术。

容器技术是一种典型的轻量级虚拟化技术。如果小明使用容器技术部署 Web 服务器，他只需部署相应的最小运行环境即可，而不需要再去部署整个操作系统。而当小明需要再次部署该 Web 服务器时，可以将之前的运行环境打包为一个镜像，随后便能随时随地使用这个镜像进行重新部署。部署一个容器的时间是以毫秒为单位进行计算的，一般情况下要快于部署一台虚拟机。同时，当业务处于高峰期导致服务器需要扩容时，容器编排平台（例如 Kubernetes [5] 等）也能够帮助我们实现快速扩展；多余的资源也可以方便地被释放。

一般情况下，轻量级虚拟化的性能要优于系统级虚拟化，其部署速度也要快于系统级虚拟化。而就隔离性而言，系统级虚拟化优于轻量级虚拟化。系统

级虚拟化与轻量级虚拟化各有千秋，有着不同的适用场景。

本章将首先介绍沙盒技术和容器技术。随后我们将会以 Linux 中的命名空间和控制组技术为例，对资源视图隔离和性能隔离进行讨论。最后本章以 AWS Lambda 为例介绍了无服务计算。

15.1 沙盒技术

本节主要知识点

- ❑ 什么是沙盒技术？
- ❑ 常见的沙盒技术是如何隔离共享的系统资源的？

虚拟化技术通常可以增强系统的隔离性，说到隔离性，我们很容易联想到在安全领域常用的沙盒技术。沙盒技术是一种可以为程序提供一个隔离的运行环境的安全机制，通常会用来运行一些无法验证来源或是可能存在安全隐患的程序。沙盒中的程序通常只能访问有限的系统资源，无法对沙盒外的环境造成破坏。所以可以安全地在沙盒中运行和分析带有病毒的程序或其他恶意代码。

由于沙盒技术能够提供隔离环境，所以许多轻量级虚拟化技术也使用到了沙盒技术。本节将会介绍几种常见的沙盒技术。

15.1.1 Chroot

1979 年，Version 7 Unix 引入了 chroot 这一命令。Chroot，即 change root directory，顾名思义，它可以启动一个程序，并将其根目录修改为用户指定的目录。如果用 chroot 来启动一个 shell，在 shell 启动之后，将只能访问这个新的根目录之下的所有内容，系统其余内容对其是不可见的。通过 chroot，我们可以将一些可能存在安全隐患的程序与其他程序隔离开来。不过，chroot 仅仅提供了文件系统的隔离，无法隔离系统中的其他共享资源。目前的大多数 Unix 系统都支持 chroot。

15.1.2 FreeBSD jail

Chroot 技术存在许多局限。首先，它只提供了文件系统的隔离；其次，chroot 比较容易被攻破，在实践中人们发现了许多能够跳出 chroot 环境的方法。因此，FreeBSD 在 2000 年提出了 FreeBSD jail。除了修复 chroot 的问

题、提供与 chroot 完全一致的功能以外，FreeBSD jail 还能够提供更多资源的隔离，例如用户和网络等资源。

15.1.3 Seccomp

Seccomp（Secure Computing Mode）是 linux 内核中的一种限制系统调用的安全机制，最初是为了在集群计算（Grid Computing）中提供一个安全的执行不可信代码的环境，于 2005 年被提出。在 linux 中，用户态的程序需要通过系统调用（Syscall）来使用系统资源。正常情况下，程序可以使用内核提供的所有系统调用。Seccomp 提供了两种使用模式，一种是严格模式（Strict Mode），另外一种是过滤模式（Filter Mode）。在严格模式下，seccomp 可以禁止程序使用除了`exit`、`sigreturn`和`read`、`write`（对已经打开的文件描述符使用）四个系统调用外的任意系统调用。在过滤模式下，用户可以使用 BPF 设定系统调用的白名单或黑名单，从而制定出多种多样的规则。关于 BPF 的详细介绍请参考第17.2.3。

15.2 容器

本节主要知识点

- ❑ 什么是容器技术？容器技术与轻量级虚拟化的关系是什么？
- ❑ 容器的架构与虚拟机的架构有什么区别？
- ❑ 如何构建容器的镜像？这种构建方式有什么好处？

容器技术是一种典型的轻量级虚拟化技术，通常会借助操作系统提供的多种隔离技术来提供进程级的隔离。其不仅具有轻量级和隔离性等优点，还具有易于移植的特性。容器技术可以将一个程序和它的依赖关系打包在一起，形成一个镜像，封装了与运行应用程序相关的所有细节，用户可以利用该镜像在其他环境中部署大量的相同容器。容器的这个特性有助于使软件开发者专注于应用程序本身，免于因开发环境与生产环境不同所带来的困扰。

15.2.1 镜像与容器

本节将首先介绍容器技术中两个常见的概念：镜像和容器。

镜像（Image） 简单来说，容器技术中的镜像是由一个文件系统以及相关的元数据组成的。这个文件系统内包含了一个可执行的程序和该程序依赖的所有数据、二进制文件和库文件。值得一提的是，镜像中的文件系统实际上是由许多层文件系统组合而成的。这种分层设计使得多个不同的镜像能够共用相同的文件系统层，降低了镜像的存储开销。

如图 15.1所示，镜像 1 是一个 Ubuntu 的镜像，我们在镜像 1 的基础上安装了 Fio，然后制作成了镜像 2。随后我们在镜像 2 的基础上制作了镜像 3——一个同时具有 Fio 和 Nginx 的 Ubuntu 系统。镜像 1、2、3 共用了 Ubuntu 这一层镜像。

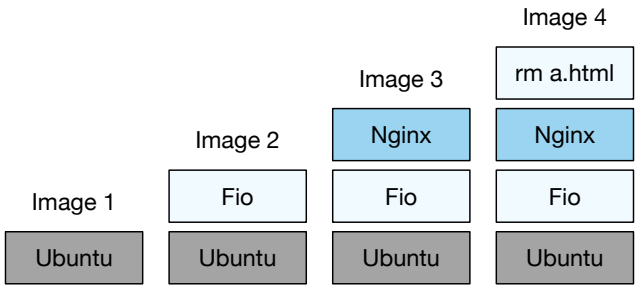

图 15.1: 镜像分层

镜像构建时会从下往上逐层构建，每一层都是在前一层的基础上构建的，在构建完成后便不可以再进行修改。因此，在构建完一层后，如果想要删除这一层的某个文件，只能通过再新建一层来实现。例如在图 15.1中，如果我们想删除镜像 3 中的一个 html 文件 *a.html*，就可以在镜像 3 的基础上添加一层构建镜像 4，该层仅包含一条指令`rm a.html`，会将该文件标记为已删除，但不会真的将其从镜像 3 中删除。因此，虽然这个文件在通过镜像 4 生成的容器中不可见，但它仍然存在于这个镜像中，占用磁盘空间。

容器（Container） 容器和镜像的关系就类似于进程和程序的关系。程序保存在磁盘中，每当运行一个程序时，便会产生一个程序的实例，即进程。同理，容器也是镜像产生的实例。一个程序可以被多次运行得到多个进程，而一个镜像也可以被用来生成多个容器。

当一个镜像被用来生成一个容器时，该容器实际上是在镜像之上添加了一层可读可写的容器层，之后对该容器所做的任何操作都是在这层容器层上进行的，对于下面的只读镜像层没有任何影响。在一台机器上，同一镜像制作而成的多个容器共享所有的只读镜像层，而最上层的容器层则是各容器私有的。现

有的容器也可以被用来制作新的镜像，此时可以读写的容器层会变为只读并添加到原有镜像的最上层，形成一个新的镜像。

容器本质上仍然是一到多个进程，但是与直接在宿主机里的原生进程不同，容器的进程有着属于自己的独立的命名空间。不同的容器之间是相互隔离的。

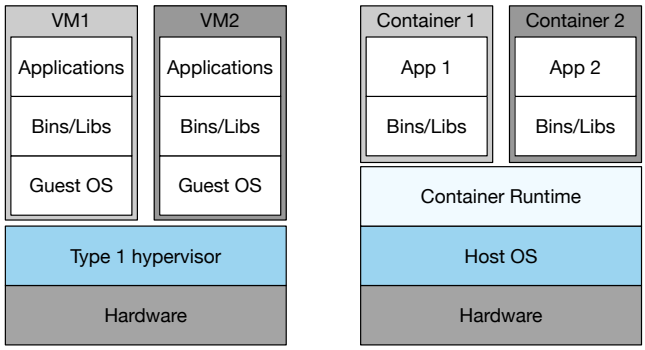

图 15.2: 虚拟机（左）与容器（右）的架构对比

15.2.2 容器的架构

图 15.2展示了容器和 Type-1 虚拟化（详见第十一章）的架构对比。Type-1 虚拟化中的虚拟机统一由虚拟机监控器管理，且可以使用不同的操作系统；容器则运行在同一个操作系统上，由相同的容器运行时（Container Runtime）管理。容器中通常只包含一个应用程序和它依赖的文件（二进制文件和库等），充分体现了“轻量级”的特征。

15.2.3 OCI (Open Container Initiative)

容器镜像要包含哪些东西、运行时系统要支持哪些操作、容器又该如何进行分发呢？这些都需要统一的标准。于是，2015 年 OCI 诞生了。OCI 是一个为了推动容器标准化而成立的开放组织，它的目的是为容器格式和运行时提供一个统一的行业标准。OCI 目前已经定义了两个规范：运行时规范和镜像规范。在 OCI 的标准中，运行容器的过程分为 3 步：

- 第一步：下载一个满足 OCI 镜像规范的容器镜像。
- 第二步：将下载的镜像解压为一个文件系统包（即 File System Bundle，是一组按照特定方式组织的文件，以下简称为 bundle）。OCI 规定了

bundle 的内容，规定了一个容器和它的相关数据是如何存储在文件系统包中的，bundle 包含了运行容器所需要的所有信息。任意兼容 OCI 运行时规范的运行时环境都能够对这个 bundle 进行标准操作（容器创建、运行和停止等操作）。

- 第三步：使用兼容 OCI 规范的运行时运行这个 bundle。

15.2.4 容器运行时

OCI 的运行时分为两大类——基于传统容器技术的运行时和基于虚拟化技术的运行时。对于这两类符合 OCI 规范的运行时，使用较广的实现有基于容器的 runC [6] 和基于虚拟机的 runV [2]。

runC runC 是一个符合 OCI 运行时规范的用于生成和管理容器的运行时，其核心内容为 libcontainer，包含了对后面将要介绍的 Linux namespace 和 cgroups 等技术的封装。它为创建容器提供了一个原生的 Go 实现，提供了创建、管理和操作容器的方法。在 libcontainer 的实现中，容器自带执行环境，它共享宿主机的内核，并与系统中的其他容器隔离。

runV runV 是一种基于虚拟机的兼容 OCI 运行时规范的实现。由于虚拟机和容器本身的差异，OCI 中对于 namespace、capability、设备等章节不适用于以 runV 为代表的基于虚拟机的容器运行时。

runV 类似于图 15.2中的左图，而 runC 类似于右图。runC 创建的容器共用同一个操作系统，而 runV 则会为每个容器创建一个单独的虚拟机，每个容器享有自己单独的操作系统。与传统容器相比，基于虚拟机的容器提供了更强的隔离性和安全性，同时也继承了容器的快速部署等优点，但性能一般不如传统容器。

与 runV 类似的基于虚拟机的容器运行时引擎还有 Intel 的 Clear Containers [3]。Kata Containers [8] 是一个 2017 年成立的开源社区，它合并了 runV 和 Clear Containers 并进行了扩展，致力于提供基于轻量级虚拟机的容器运行时引擎，提供接近传统容器的性能。同时，Kata Containers 还使用硬件虚拟化技术提供更强的工作负载隔离。

15.2.5 案例分析：Docker

Docker 是 DotCloud（2013 年改名为 Docker 公司）公司提供的一个开源的容器引擎，最初它是基于 LXC（LinuX Containers）实现的，随后从 0.7 版本开始，它将 LXC 替换为自己实现的 libcontainer。2015 年，由 Docker 公司牵头成立了 OCI，定义了我们前面说到的两个规范：运行时规范和镜像规范。由于 Docker 使用及其广泛，因此这两个规范大部分都是基于 Docker 定义的。同时 Docker 公司将自己的镜像格式和 libcontainer 都捐赠给了 OCI。OCI 将 libcontainer 重新封装，形成了 runC，Docker 也从 1.11 版本开始使用 runC 作为自己的运行时。

Docker 使用 Go 语言进行开发，无论是最初使用的 LXC 还是后面自己开发的 libcontainer 和 runC，都使用了 Linux 的 namespace 和 cgroups 技术以实现资源隔离。容器镜像的分层则是使用 Union FS 来实现的。Union FS 可以将多个不同位置的目录合并起来挂载到同一个目录下，从而支持容器镜像的分层设计。启动容器时，通过 Union FS 将这个容器需要用到的每一层挂载到同一个目录，从而成为这个容器的根文件系统。

当我们制作完一个容器镜像后，可以很方便地在本机上运行。如果想要将制作的镜像发布出去供别人使用，我们便需要一个镜像存储和分发的服务，Docker 仓库便是这样的一个服务。最常用的仓库服务器是官方提供的 Docker Hub，在这个仓库中有着大量官方验证过的高质量镜像。当我们制作完一个镜像后，便可以将自己的镜像发布到 Docker Hub 上以供他人使用。

镜像名的格式通常是“用户名/软件名: 版本号”，以本书附带 lab 的构建镜像 “ipads/chcore_builder:v1.0” 为例，“ipads” 是用户名，而 “chcore_builder” 则是对应的镜像名，“v1.0” 是版本号。如果不加版本号，则默认使用 “latest” 的版本号。我们可以使用`docker pull ipads/chcore_builder:v1.0`来将这个镜像拉到本地。

`docker run ipads/chcore_builder:v1.0` 则会使用这个镜像运行容器，若执行这条命令前没有运行`docker pull`命令，这条命令将会自动从镜像仓库拉取对应的镜像到本地。

Docker 默认使用的是 runC 引擎，同时也支持多种兼容 OCI 运行时规范的运行时，例如 runV，可以通过配置更换使用的运行时引擎。Docker 提供了许多子命令来配置和管理容器和镜像，感兴趣的读者可以自行深入了解 Docker 的使用方法。

Docker 与虚拟机的使用过程有哪些区别呢？以运行一个 Ubuntu 环境为例，如果我们要用虚拟机搭建一个 Ubuntu 环境，首先我们得下载一个 Ubuntu

镜像（数百 MB 至数 GB），随后使用 QEMU、VMWare 等虚拟机软件来安装 Ubuntu 系统，安装过程一般需要花上几分钟。如果我们使用 Docker，通过一行`docker run -it ubuntu bash`我们便已经拥有了一个 ubuntu 环境，而整个过程（不计算镜像拉取时间）只需要几秒钟。镜像大小只有数十 MB。

Docker 已经被许多大公司所广泛采用，例如腾讯的 Gaia 平台早在 2014 年 10 月便已上线对 Docker 的支持，同年，淘宝的应用引擎服务 TAE(Taobao App Engine) 也支持了 Docker 技术。

15.2.6 案例分析：Kubernetes 与云际计算

容器技术（以 Docker 为例）为应用程序的打包和运行提供了便利，上一节提到，越来越多的公司正在将 Docker 用于生产环境中，在这类生产环境中通常会有大量的容器运行着不同的服务。如何便捷地管理海量的容器成为了将容器广泛部署于生产环境中的一个自然的问题。

但生产环境中可能存在哪些问题需要容器编排系统来解决呢？首先，由于生产环境中通常运行着海量的容器，少量容器停机的现象十分普遍，当一个容器发生故障时，需要由系统来自动启动另外一个容器，而无需人为干预。其次，提供服务的容器数量需要能够随着用户请求的数量而动态扩展，当用户请求密集或稀疏时，系统要能够提供与其匹配的容器数量，既能够处理用户的请求，又不会浪费过多的系统资源。同时，海量的容器如何均衡地分布在一家云服务商提供的不同机器中、甚至是分布在多个云服务商提供的不同云中也是容器编排系统需要考虑的问题。

Kubernetes [5] 是一个由 Google 所开发的管理和部署容器化服务的开源平台，可以实现动态伸缩、负载均衡、自动部署等功能。2019 年阿里巴巴开始全面推动 Kubernetes 落地，在双十一之前将全部核心应用都迁移到了 Kubernetes 上 [1]，容器规模达到了 200 万。

借助 Kubernetes，阿里巴巴为双十一大考交出了完美的答卷。Kubernetes 的架构如图 15.3所示，主要由节点（Node）和控制平面（Control Plane）组成。

每个节点主要包含以下几个组件，用于维护运行在该节点中的 Pod（一组正在运行应用负载的容器，是 Kubernetes 管理的最小单元）：

- *kubelet* 作为每个节点与控制平面交互的代理，接受控制平面的 Pod 创建请求，与节点上的容器运行时交互并创建与 Pod 相关的容器，以及监控容器的运行健康状况和资源使用情况。

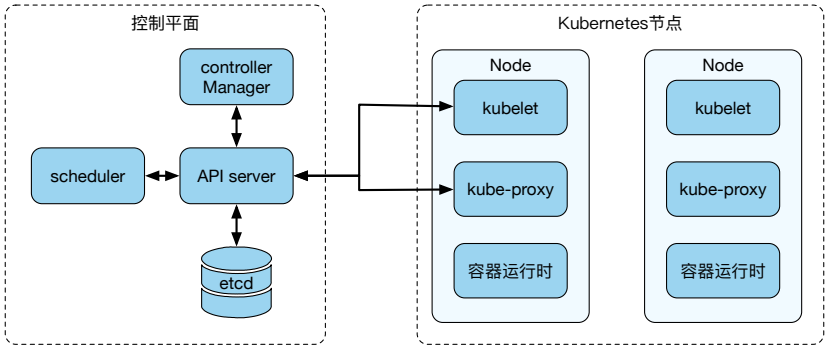

图 15.3: Kubernetes 整体架构

- *kube-proxy* 是每个节点上的网络代理，维护了节点中的网络规则，实现了节点的网络流量转发。
- 容器运行时用于管理容器的生命周期，负责创建、运行和销毁容器。常用的容器运行时包括 Docker、CRI-O 和 containerd 等。

而控制平面主要有以下几个组件：

- *API server* 是 Kubernetes 控制平面的前端，对外暴露了 Kubernetes 的应用程序接口。
- *scheduler* 综合考虑多方面的限制条件（例如硬件资源、数据局部性等），为新创建的 Pod 寻找合适的节点。
- *etcd* 是一个高可用强一致性的键值存储系统，被用于存储 Kubernetes 集群运行状态、集群配置等重要信息，需要定期备份。
- *controller manager* 负责管理 Node controller（负责在节点离线进行通知和响应）、Replication controller（负责保证任意时刻系统中都有正确数量的 Pod 副本）等一系列控制器。

单一云服务提供商存在着多方面的问题。首先是平台锁定问题，对单一的云服务商的强依赖可能会在云服务商出现区域性故障时导致服务中断等严重后果。其次为了满足客户爆发性业务的资源需求，云服务商可能需要大量配置平时用不满的计算资源，从而导致资源利用率低，进而提高了云服务商的成本，而最终成本会转嫁到用户身上。另外，随着越来越多的国家实体参与到全球化的合作项目中来，由于政府监管、数据隐私和政治问题等因素，单一的云服务提供商无法同时满足所有的需求。

对于第一个问题，Kubernetes 能够对跨越多个云服务商的集群进行管理，当某个云服务商的区域出现故障时，Kubernetes 能够将负载迁移到其他健康的可用区域。而其他问题则需要依靠在全球建立新的合作计算模式来解决，就像航空服务业的航空联盟一样。然而，如何提供跨云计算服务，实现多云之间的开放协作和深度合作面临着许多挑战，例如如何在不同的云基础设置之间提供有效的通信、集成多种云服务。

对于这些问题，云际计算（JointCloud）[10] 关注了云服务商之间的横向合作，提出了云际协作环境（JointCloud Collaboration Environment）和对等协作机制（Peer Collaboration Mechanism）以应对这些挑战。

云际协作环境主要提供了三项基于区块链的服务。第一项服务旨在解决云之间的云资源定价、拍卖和其他交易相关问题。第二项服务提供了资源注册、服务搜索等服务，以促进云之间的合作。第三项服务则负责对联合云生态系统中各种资源、服务和云进行监管。基于云际协作环境，只要云服务商实现了对等协作机制并提供了相应的 API，不同的云之间就可以相互合作。

15.3 资源视图隔离

本节主要知识点

- ❑ 资源视图隔离的目标是什么？
- ❑ 有哪些方法可以达到资源视图隔离的效果？
- ❑ 这些资源视图隔离方法各自适用于哪些的系统资源？

15.3.1 资源视图隔离概述

对于一个进程而言，资源视图即该进程所能看到的各类系统资源。为了保证容器的安全隔离与可移植性，容器技术为其中的进程提供了一套统一的资源视图。容器技术会对主机操作系统中的全局资源进行封装，处于某个容器中的进程只能够访问封装后的局部系统资源，然而该进程会认为自己拥有隔离的（Isolated）全局系统资源。

当一个打包完毕的容器运行在不同的主机操作系统上时，这些系统中的全局资源可能各不相同（例如文件目录结构不同、用户群组权限不同等），但容器内部的进程并不会受到影响，它们正常运行所依赖的大部分环境仍然是相同

的、封装后的局部资源。以一个网络服务器应用为例，如果直接将该应用的可执行文件运行于不同的操作系统上，难免会遇到网页数据的存放目录不同、网络接口变更等麻烦，用户需要手动地进行适应性调整。不仅如此，当多个应用程序直接运行在同一个操作系统上时，可能试图使用同一个挂载点从而造成冲突。而如果利用容器技术对应用程序进行资源视图方面的隔离就可以解决上述遇到的问题，每个容器在易于移植的同时也不会影响容器外部的正常运行。

一个操作系统中包含了多种类型的资源，资源视图隔离需要对几乎每一类资源进行封装。由于资源视图隔离的目标对象是进程，所以不同类型的系统资源根据其与进程的关系以及对应操作系统中进程模型，所需的隔离方法也会有所不同。本小节将待隔离的资源分为独立于进程的资源、绑定于进程的资源、其余特殊资源三类，分别介绍它们所适用的资源视图隔离方法。

上文中提到，FreeBSD 为了改进传统 Unix 系统中 chroot 技术引入了 jail 机制，同样的，Linux 也提出了改进后的资源视图隔离机制**–命名空间**（Namespace）。考虑到 Linux 拥有较强的通用性以及成熟的轻量级虚拟化方案，本节将选取部分 Linux 命名空间作为案例分析，详细介绍资源视图隔离方法在现实中的设计与应用。[1]

15.3.2 双向隔离：独立于进程的资源

在针对容器实现某类资源的视图隔离时，一种直接的想法就是将容器所用到的该类资源与外界完全分离、互不干扰。对于以挂载点、IPC、主机名（Hostname）为代表的资源，它们并非进程自身的属性，在容器的内外两侧不会也不应当有任何的关联。例如，容器内的 IPC 资源应当仅供容器内部的进程之间通信使用，不会连通容器内外两侧的进程，否则就打破了容器的隔离。因此让容器内外互相无法感知对方资源的双向隔离方法对于这类资源是简单有效的。

15.3.3 案例分析：Linux 中挂载点的双向隔离

在轻量级虚拟化中，每个容器都需要一个属于自己的文件系统树，保证容器在对其所属文件系统树中的挂载点进行操作的同时，不会影响到当前容器外的文件系统树，反之，容器外对于挂载点的修改通常也不会反映到容器内部。

[1]本章用于展示的 bash 示例均基于 Ubuntu 18.04 LTS 发行版，由于 Linux 不同发行版存在一些差异，可能会出现相同的指令导致不同的结果。

本小节选取 Linux 的挂载命名空间（Mount Namespace）进行案例分析，详细介绍 Linux 是如何实现实现挂载点资源视图隔离的。

挂载命名空间是 Linux 引入的第一个命名空间，其主要负责对命名空间内的进程所看到的挂载点进行隔离。在 Linux 中，多个文件系统会以一个树状的层级结构呈现给进程，因此不同挂载命名空间中的进程看到的文件系统树结构也会有所不同。

在操作系统启动过程中，Linux 会首先创建一个**初始挂载命名空间**（Initial Mount Namespace），默认情况下所有初始进程都处于该挂载命名空间中并共享同一文件系统树结构。每当创建一个新的挂载命名空间时，新创建的子挂载命名空间会从父挂载命名空间复制一份挂载点的列表，换言之，子挂载命名空间中的初始文件系统树结构与其父命名空间相同，然而随后子挂载命名空间中某个挂载点的变更将对父挂载命名空间不可见，反之亦然。由此实现了不同挂载命名空间之间对于挂载点这一系统资源的隔离。

如图 15.4所示，假设我们有两个挂载命名空间 A 和 B，在初始状态下，A 和 B 中的文件系统树结构是相同的，它们的`/mnt`目录下皆挂载了同一个 EXT4 格式的文件系统。随后 B 中的进程使用`mount`命令将另一个 XFS 格式的文件系统挂载到了 B 中的`/mnt`目录下，此时 B 中的进程将可以通过`/mnt`目录访问到这个新的文件系统。但是 A 中的进程并不受到 B 中挂载操作的影响，在`/mnt`目录下仍然访问到的是旧的 EXT4 格式的文件系统。

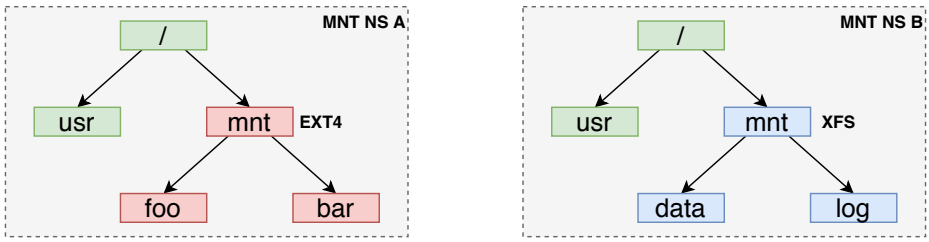

图 15.4: 挂载命名空间图例

输出 15.1在真实环境下展示了挂载命名空间对于挂载点资源的隔离效果。首先通过父进程创建一个子进程，并将子进程放入一个新创建的挂载命名空间，然后在父子进程中均执行 bash，在子进程中对`mnt`目录执行挂载操作，观察两个进程看到的挂载点资源变化。可以看出子进程对于所在命名空间进行的挂载点变更不会影响到处于父进程所在的挂载命名空间。

```
# 父进程bash
```

```
[user@osbook ~] $ stat -f mnt/
File: "mnt"
    ID: 4c425af2b6d3090 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 59828639   Free: 14266302   Available: 11209739
Inodes: Total: 15269888   Free: 14057886

[user@osbook ~] $ sudo unshare --mount --fork /bin/bash
# 子进程bash
[root@osbook ~] # mount msdos.img mnt/
[root@osbook ~] # stat -f mnt/
File: "mnt/"
    ID: 70000000000 Namelen: 1530    Type: msdos
Block size: 2048       Fundamental block size: 2048
Blocks: Total: 65399      Free: 52432      Available: 52432
Inodes: Total: 0          Free: 0

# 另外新开一个bash，与父进程处于同一挂载命名空间，挂载点未受影响
[user@osbook ~] $  stat -f mnt/
File: "mnt"
    ID: 4c425af2b6d3090 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 59828639   Free: 14266302   Available: 11209739
Inodes: Total: 15269888   Free: 14057886
```

输出记录 15.1: 挂载命名空间示例

在 Linux 的实现中，挂载命名空间结构体定义如代码片段 15.1所示。如果当一个子进程被创建时需要将其放入一个新的挂载命名空间内，内核首先会调用`alloc_mnt_ns`函数创建一个新的`mnt_namespace`实例，然后调用`copy_tree`为其拷贝一份父进程的文件系统树副本，之后将新建的挂载命名空间结构体中`root`指针指向文件系统树副本的根目录挂载点。在后续对于挂载点资源进行修改时，由于父子进程使用的是两个独立的文件系统树结构，所以不会相互造成影响。如果一个进程想要改变自己所在的挂载命名空间，只需拷贝一份自己原有的文件系统树副本，其余流程与上文一致。

```
struct mnt_namespace {
  atomic_t              count; // 被引用计数
  struct ns_common      ns;
  struct mount          *root; // 指向根目录的挂载点实例
  // 保存了所有文件系统的挂载点实例，
  // 链表元素是 struct mount 的成员 mnt_list
  struct list_head      list;
  struct user_namespace     *user_ns;
  u64                   seq;    // 用于防止循环引用的序列号
  wait_queue_head_t     poll;   // 轮询的等待队列
  u64                   event; // 事件计数
};
```

代码片段 15.1: 挂载命名空间结构体定义

15.3.4 单向隔离：绑定于进程的资源

然而在轻量级虚拟化场景下，并不是所有的资源都可以保证在容器的内外两侧毫无关联，导致无法使用双向隔离的方法。例如，遵循 POSIX 标准的主机操作系统通过维护一棵进程树，将所有进程串联在一起并记录了进程间的父子关系。一个容器内的进程也不例外，同样会被加入主机操作系统的进程树中，使得容器外的管理进程可以方便地寻找、回收特定进程。这也意味着容器内的进程变更会被反映到容器外，对于以**进程标识符**（Process Identifier，PID）、用户 ID 为代表的资源，它们与进程本身紧密耦合，同样无法避免向容器外暴露相关的信息，因此对于这类资源需要采用单向隔离方法。

15.3.5 案例分析：Linux 中进程标识符的单向隔离

在轻量级虚拟化中，每个容器需要在保证容器外的管理进程可以获取足够进程信息的同时，维持容器内进程对于 PID 资源的统一视图，即容器外能够看到容器内的进程，反之不能。本小节选取 Linux 的 PID 命名空间（PID Namespace）进行案例分析，详细介绍 Linux 是如何实现实现 PID 资源视图隔离的。

进程标识符是 Linux 系统中每个进程都拥有的属性，通常被用来识别和追踪某个进程，PID 命名空间就是负责隔离 PID 这一系统资源的机制。一个 PID 命名空间内的任意进程在当前 PID 命名空间中都有一个唯一的 PID，而不同 PID 命名空间中的进程可以用有相同的 PID，这个功能对于容器的实现而言至关重要。

在 Linux 操作系统中，第一个被创建的进程被称作 **init 进程**，是 PID 为 1 的**守护进程**（Deamon Process），该进程直接或间接地创建了系统中后续的所有进程，其生命周期从系统初始化开始到系统停止运行结束。init 进程承担了一定的进程管理任务，例如内核会将 init 进程设置为所有**孤儿进程**（Orphan Process）的**父进程**（Parent Process）。为了实现轻量级虚拟化，每个容器中都需要有一个 init 进程存在。假设一个宿主操作系统创建了一个 PID 分别为 1000 的普通进程，现在系统想要将其作为容器 A 的 init 进程，那么就可以利用 PID 命名空间使得这个进程在 A 中的 PID 为 1。虽然在宿主操作系统看来这依旧是一个普通进程，但是容器中的所有进程都会认为该进程是它们的 init 进程。

如图 15.5所示，假设我们有五个不同的进程处于 Root PID 命名空间及其下属的子 PID 命名空间 A 中，其中全局 PID 为 2、3 和 4 的进程同时属于两个 PID 命名空间，而 init 进程与 PID 为 5 的进程仅属于 Root PID 命名空间。在 Root PID 命名空间中的进程可以看到图中蓝色的 PID 资源（即全部五个进程的全局 PID），而在 A 中的进程只能看到图中红色的 PID 资源（即属于 A 的三个进程映射后的 PID），无法访问 A 以外的 PID 资源。因此 Root PID 命名空间中的进程认为它们的 init 进程是全局 PID 为 1 的进程，而 A 中的进程认为它们的 init 进程时全局 PID 为 2 的进程。

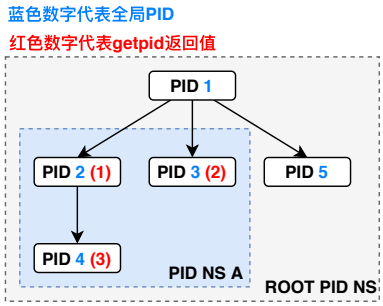

图 15.5: PID 命名空间图例

可移植性也是容器虚拟化必不可少的特性之一，一个容器镜像需要能够在不同的宿主操作系统上进行迁移并继续正常工作，所以在迁移前后容器中的进程要能够正确地相互识别与通信。假设有两个不同的宿主操作系统 A 和 B，一个容器镜像在 A 上运行时，容器内的进程在 A 看来 PID 范围是 1000 到 1100，在迁移到 B 后，容器内的进程在 B 看来 PID 范围是 2000 到 2100。由于容器中的进程有着自己的 PID 命名空间，它们看到自己的 PID 范围一致保持 1 到

100 从未发生变化，从而保证了容器内进程的正常运行。

输出 15.2在真实环境下展示了 PID 命名空间对于 PID 资源的隔离效果。首先通过父进程创建一个子进程，并将子进程放入一个新创建的 PID 命名空间，然后在子进程中均执行 bash 并查看自己的 PID，然后对比父进程所在 PID 命名空间中看到的子进程 PID。可以看到子进程作为新生成的子 PID 命名空间中的第一个进程，获取到自己的 PID 为 1，而父进程所在的 PID 命名空间中看到的子进程 PID 为 3229。

```
# 父进程bash
[user@osbook ~] $ sudo unshare --pid --fork /bin/bash
# 子进程bash
[root@osbook ~] # echo $$
1

# 另外新开一个bash，与父进程处于同一PID命名空间
[user@osbook ~] $ ps aux | grep unshare
root      3218  0.0  0.0  65608  4188 pts/10   S    21:07   0:00
 sudo unshare --pid --fork /bin/bash
root      3229  0.0  0.0   7456   724 pts/10   S    21:07   0:00
 unshare --pid --fork /bin/bash
```

输出记录 15.2: PID 命名空间示例

PID 命名空间的隔离效果与前面提到的双向隔离效果有些许差别。在使用双向隔离方法对于全局资源进行隔离后，后续对于被隔离资源所做的任何修改都无法影响到包括父命名空间在内的其他命名空间。然而 PID 命名空间不同，由于 Linux 系统需要记录进程之间的关系用于管理（如进程组），一个 PID 命名空间中创建一个新的进程后，所有的父命名空间也可以看到这个新创建的进程。因此，不仅一个新进程所在的 PID 命名空间需要新分配一个 PID，它所属的每一级父 PID 命名空间（如多级嵌套时）都需要给该进程在对应命名空间中分配一个唯一的 PID。例如，一个 PID 命名空间创建了一个进程并分配 PID 为 1000，它的上一级父 PID 命名空间为这个进程分配 PID 为 2000，则父子命名空间中的其他进程可以分别通过这两个 PID 向该新进程发送信号。因此创建一个 PID 命名空间的过程与上文提到的几种命名空间会有所不同。

```
struct pid_namespace {
  struct kref kref;  // 引用计数
  struct idr idr;    // 记录当前命名空间中的 PID 分配情况
  struct rcu_head rcu;
  // 当前命名空间中已分配 PID 的数量
  unsigned int pid_allocated;
  // 当前命名空间的 init 进程
  struct task_struct *child_reaper;
  struct kmem_cache *pid_cachep;
  // 当前命名空间的层级
  unsigned int level;
  struct pid_namespace *parent;       // 父命名空间

  ...
};
```

代码片段 15.2: PID 命名空间结构体定义

在 Linux 的实现中，PID 命名空间结构体定义如代码片段 15.2所示。如果当一个子进程被创建时需要将其放入一个新的 PID 命名空间内，内核中的流程会分为两个主要步骤：

- 分配一个新的 *PID* 命名空间实例

  首先创建一个新的`pid_namespace`实例，然后调用`idr_init`函数初始化当前命名空间中用于分配和管理 PID 资源的树结构，随后对于剩余字段进行初始化，例如将`parent`字段指向父级命名空间实例、将`level`字段赋值为当前 PID 命名空间所在的层级数（即父级命名空间的层级数+1）等。

- 在该进程所属的各级命名空间中分配 *PID*

  首先创建一个新的`pid`实例，然后从当前 PID 命名空间的层级开始到 Root PID 命名空间层级为止，循环调用`idr_alloc_cyclic`函数分配可用的 PID 数值。

这样一来，每个新创建的进程在其所属的所有父级 PID 命名空间中都是可见的，而不存在父子关系的两个 PID 命名空间在各自的树结构中看不到对方的 PID 资源。

15.3.6 特殊资源的隔离

除了上述两种资源，通常情况下容器需要通过网络资源获取与外界交流的能力，典型的容器使用方式包括在容器内运行一个 Nginx 网络服务器并对外提供服务，所以独立的网络资源对于大部分容器来说就是必不可少的。虽然网络接口资源并不依附与进程本身，容器内外分别使用独立的网络接口即可，对于网络资源的修改并不会相互影响。但由于很多网络资源的访问需要硬件网卡，而硬件网卡只会在主机操作系统中生成相应的网络接口，因此需要主机操作系统提供虚拟网卡以及网络转发功能。因此隔离这类较为特殊的系统资源需要容器外的主机操作系统进行配合。

15.3.7 案例分析：Linux 中网络资源的视图隔离

本小节选取 Linux 的网络命名空间（Network Namespace）进行案例分析，详细介绍 Linux 是如何为网络相关资源提供资源视图隔离的。

Linux 系统初始化时创建的所有进程都属于同一个 Root 网络命名空间，它们共享宿主操作系统上的网络资源，每个网络命名空间中都有一套自己的网络资源，包括独立的网络设备、IP 地址、端口号等。如图 15.6所示，假设我们新创建了 A 和 B 两个网络命名空间，它们不会继承父级网络命名空间中的任何网络资源，仅会自动创建一个用于**回环**（Loopback）通信的虚拟网络接口（即图中的 lo 设备）。因为回环接口仅允许同一台主机上的进程间进行网络通信，所以进程在未经过配置的网络命名空间中是无法与网络命名空间外部的网络通信的。

需要注意的是，一个网络设备（包括虚拟设备、物理设备和网桥等）只能出现在一个网络命名空间中，另外物理网络设备只允许分配给 Root 网络命名空间，所以后续的网络命名空间必须借助虚拟网络设备与外界连通。一种常见的做法是利用 Linux 提供的 veth 虚拟网络设备，如下图中我们创建了两对相连的虚拟网络设备分别分配给 A 和 B，其中 veth1 与 A 中的 eth0 相连，veth2 与 B 中的 eth0 相连，这样 A 和 B 中的进程就可以通过各自的 eth0 设备对网络命名空间外进行网络访问了。如果需要连通多个网络命名空间则可以借助 Linux 提供的虚拟网桥（功能类似交换机），在图中我们将 veth1 和 veth2 设备连接到了网桥 br0 上，在配置好相关网络设置之后 A 和 B 中的进程之间就可以通过网络访问到对方了。在实际使用中一个网络命名空间的配置还涉及到路由规则、防火墙等方面的配置，在此不再赘述。

输出 15.3在真实环境下展示了网络命名空间对于网络资源的隔离效果。首

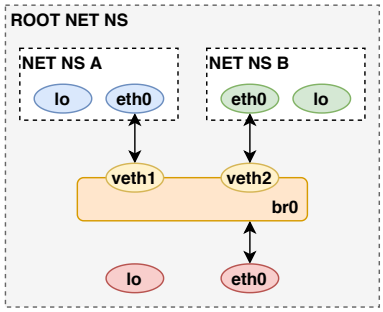

图 15.6: 网络命名空间图例

先通过处于 Root 网络命名空间中的父进程 bash 创建一个网络命名空间，然后创建一个子进程放入其中均执行 bash。在父子进程中均尝试 ping localhost 并查看自己的网络接口，然后对比父进程中的 localhost 连通性以及网络接口。其中父进程可以正常 ping 通 localhost，查看网络接口可以看到 loopback 接口以及外部网络接口 enp16s0 均处于启动（UP）状态，而子进程在新生成的子网络命名空间中的无法 ping 通 localhost，查看网络接口发现只有一个 loopback 接口且处于未启动（DOWN）状态。可以看出父子进程看到的 loopback 接口并不是同一个网络接口，两个网络命名空间中的网络资源是互相隔离的。

```
# 父进程bash
[user@osbook ~] $ ping -c 1 localhost
PING localhost(localhost (::1)) 56 data bytes
64 bytes from localhost (::1): icmp_seq=1 ttl=64 time=0.028 ms

--- localhost ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.028/0.028/0.028/0.000 ms

[user@osbook ~] $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp16s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
state UP group default qlen 1000
    link/ether 34:97:f6:5c:85:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.11.174/16 brd 192.168.255.255 scope global
```

```
    dynamic noprefixroute enp16s0
       valid_lft 574432sec preferred_lft 574432sec
       inet6 fe80::3697:f6ff:fe5c:8584/64 scope link
       valid_lft forever preferred_lft forever

    [user@osbook ~] $ sudo ip netns exec net-ns-demo bash
    # 子进程bash
    [root@osbook ~] # ping -c 1 localhost
    connect: Cannot assign requested address
    [root@osbook ~] # ip a
    1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default
    qlen 1000
       link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

输出记录 15.3: 网络命名空间示例

在 Linux 的实现中，网络命名空间结构体定义如代码片段 15.3所示。如果当一个子进程被创建时需要将其放入一个新的网络命名空间内，内核首先会调用net_alloc创建一个新的net实例，然后调用setup_net初始化该实例中的字段，包括遍历所有的网络模块进行初始化，比如对 loopback 设备进行初始化、配置默认路由表等。新分配的网络命名空间中默认只包含一个 loopback 设备，而内核会将后续新分配的网络设备绑定到当前网络命名空间，使得不同网络命名空间中的进程无法互相访问对方的网络设备。如果一个进程想要改变自己所在的网络命名空间，需要修改自己所绑定的字段，主要流程与上文一致。

```
1  struct net {
2    refcount_t          passive; // 决定该命名空间的释放时机
3    refcount_t          count;   // 决定该命名空间的关闭时机
4
5    ...
6
7    struct list_head    list; // 所有网络命名空间实例的链表
8    struct list_head    exit_list;
9    struct list_node    cleanup_list;
10
11   ...
12
13   struct idr    netns_ids;
14   struct ns_common  ns;
15
16   ...
17 };
```

代码片段 15.3: 网络命名空间结构体定义

虽然不同网络命名空间中的进程看不到彼此的网络设备，但是一个进程可以将当前网络命名空间中的网络设备移动到子命名空间中。例如上文中提到的veth 虚拟网络设备对的经典用法，一个进程在当前网络命名空间中创建了一对相连的虚拟网络设备后，将其中一个设备移动到子命名空间中去，使得子网络命名空间中的进程可以与父命名空间进行网络连接。内核中的流程会先从父命名空间中关闭并释放该虚拟设备，然后再将该虚拟设备添加到子网络命名空间中，最后进行新设备已添加的通知工作。

15.4 性能隔离

本节主要知识点

- ❑ 为什么需要性能隔离？
- ❑ 有哪些资源控制模型？
- ❑ 哪些资源需要性能隔离？

本节中我们将会以借助操作系统实现的进程级隔离为例来讨论轻量级虚拟化中的性能隔离方案。系统级虚拟化的隔离边界是虚拟机，在进程级隔离中，隔离的边界是任务组。

安全隔离虽然可以让不同的进程互相不可见，但是对于宿主操作系统来说，这些被隔离的进程与正常的进程都是平等的，它们对系统资源的竞争与普通的进程并无任何区别。如果一个被隔离的进程大量使用了某种物理资源，它仍会对其他进程的执行造成干扰。因此，安全隔离并不足以实现轻量级虚拟化。

在安全隔离的基础上，我们还需要添加对有限的物理资源进行划分控制的功能，也就是性能隔离。性能隔离指的是将操作系统中的各种物理资源按照某种规则进行划分。第 六章中介绍的任务调度实际上就是对有限的 CPU 时间片资源进行划分。6.3.3节提到，当小明和小红共用一台机器时，若各自的任务数量不同，则可能无法均分 CPU 资源，而好的调度策略能够一定程度上解决 CPU 资源分配不均的问题。

在轻量级虚拟化的性能隔离中，包括 CPU、内存、磁盘 I/O 等资源在内的资源隔离问题是多用户或者多个不同的应用共用同一设备时十分常见的问题。在多用户共用同一台物理机器时，除了 CPU 资源以外，内存、磁盘、网络等资源都存在竞争，因此都存在性能隔离的需求。考虑这样一个场景，假设磁盘

的最大读写速度为 500MB/s，小明和小红同时都在使用该磁盘。如果没有性能隔离机制，就可能会出现小红和小明都在频繁访问磁盘，但两人的读写磁盘速度分别为 450MB/s 和 50MB/s 的情况，这对小明来说显然是不够公平的。本节将从资源控制模型和典型资源的隔离方案出发，分析性能隔离机制是如何解决以上问题的。

15.4.1 资源控制模型

性能隔离会按照什么样的模型来划分资源呢？说到这里，首先得思考一下，资源控制的目的是什么。显而易见的是，类似于系统级虚拟化可以为每个虚拟机分配一定数量的 CPU、内存等资源，限制一个被隔离的进程所能使用的物理资源的最大值是一种直观的资源控制模型。与此同时，有一些进程可能会大量使用某种物理资源，如果没有其他进程也需要该物理资源，那么我们可以让它充分利用这种资源。但当许多进程都需要该物理资源时，将会导致大量的竞争，许多进程阻塞在请求该资源上，成为整个系统的性能瓶颈。这时就需要控制那些大量使用某种物理资源的进程，不让其过度使用该资源。那么这又该使用怎样的资源控制模型呢？下面将会介绍一些常见的资源控制模型。

最大值 最大值是一种十分直观的资源控制模型，它允许用户直接设置一个任务组能使用的物理资源的最大值。例如，用户可以设置一个任务组最多能使用多少物理内存，或者每秒最多能读写块设备上多少数据。假设用户给一个任务组设定了 1GB 的最大内存限制，当该任务组的内存占用量已经达到 1GB 以后，操作系统可能会终止其继续申请内存的任务。通过类似的方法，可以确保一个任务组不会使用超出其最大限制的物理资源，这防止了一个任务组使用过多物理资源的情况，减少了不必要的资源竞争。

回到最开始的例子中，小明和小红有着各自的任务组，并希望能公平地使用磁盘 I/O 资源，所以可以给他们分别设定 250MB/s 的最大磁盘访问速度。当他们频繁访问磁盘时，现在小红只能够使用 250MB/s 的磁盘 I/O 资源了，由于减小了资源竞争，小明的磁盘访问速度也能够上升到 250MB/s 了。

比例 在实际场景中，系统中各个任务组对各种物理资源的需求是不断变化的，因此直接限制一个任务组所能使用的物理资源最大值并不总是合理的。在前面的例子中，为了公平，我们给小明和小红都按照最大值的资源控制模型设置了 250MB/s 的磁盘 I/O 上限。假如某一时刻小明不再需要大量读写磁盘了，但由于资源控制模型的限制，小红仍然无法使用超过 250MB/s 的磁盘带

宽，这就造成了资源浪费。实际上，资源控制的目的只是让小明和小红尽可能公平地使用磁盘 I/O 资源，只有当他们同时需要访问磁盘时才需要进行限制。为了解决这一问题，比例（权重）这种资源控制模型应运而生。

还是以磁盘 I/O 为例，用户可以给不同的任务组设定权重，当它们需要使用磁盘时，它们的最大磁盘 I/O 带宽会按照权重进行分配。比如，我们可以为小明和小红设定同样的磁盘 I/O 权重值。当只有小红访问磁盘时，操作系统会将所有的磁盘 I/O 带宽都分配给小红，此时小红的读写磁盘速度可以达到 500MB/s；而当小红和小明同时访问磁盘时，操作系统则会按照他们的权重比对磁盘资源进行划分，此时小红和小明的最大读写磁盘速度均为 250MB/s，与最大值的资源控制模型达到了相同的效果。因此，比例的资源控制模型既可以使任务组按照一定的限制使用物理资源，同时也不会造成物理资源的浪费。

总的来说，不同的资源控制模型有着不同的优点、适用于不同的场景，具体使用哪种资源控制模型还需要根据需求决定。

15.4.2 CPU 性能隔离

第6.2.2节中介绍的操作系统进程调度可以对进程使用的 CPU 时间片进行调度。在某种程度上，操作系统进程调度实现了一定程度的 CPU 性能隔离。当使用系统虚拟化技术来创建一台虚拟机时，通常需要为虚拟机指定 CPU 数量。假如我们在一台有 8 个物理 CPU 的物理机上使用系统级虚拟化创建了一台虚拟机，并为它分配了 2 个 CPU，无论这台虚拟机中有多少进程，他们最多只能使用 2 个物理 CPU，而无法使用更多的 CPU。操作系统进程调度能够为进程级隔离提供类似的支持吗？

一般情况下，进程级隔离是以任务组为粒度进行隔离的，但传统任务调度的粒度是单个任务，所以第一步我们需要让任务调度提供以任务组的粒度来进行调度的机制。以6.6.1节中介绍的完全公平调度器（CFS）为例，Linux 2.6.24 引入了组调度的概念，CFS 的调度单位也从进程变为了调度实体（Sched Entity），一个调度实体可以是一个任务，也可以是多个任务，即任务组。通过组调度，任务组便能够公平地利用 CPU 资源。

小思考

有什么办法能够让进程级隔离提供类似于系统级虚拟化的 CPU 隔离吗？

通过系统级虚拟化为某台虚拟机分配 2 个 CPU 可以确保该虚拟机不会使用多于 2 个 CPU 的资源。如何在操作系统提供的进程级隔离中实现类似的功

能呢？让我们思考一下操作系统为 CPU 资源分配提供了什么技术，首先想到的就是任务调度。

可以通过任务调度为一个任务组分配固定数量的 CPU 吗？假如我们在任务调度器中记录下每个任务组当前有多少个任务正在使用 CPU，我们便能够在调度新的任务时检查当前任务组是否还能继续分配新的 CPU 时间片资源。例如我们为一个任务组分配了 1 个 CPU，当这个任务组中第一个任务被调度到时，调度器发现此时这个任务组没有任务正在使用 CPU，所以允许此次调度，即该任务能够正常获取时间片。当第一个任务正在运行，还没释放 CPU 时间片时，第二个任务被调度到了，此时调度器发现这个任务组已经有一个任务正在使用 CPU 了，所以禁止了本次调度，转而选取其他的任务进行调度。如此我们便保证了这个任务组不会同时使用超过 1 个的 CPU。

有时我们想为任务组分配的时间片资源并不是一个整数，例如只想让一个任务组使用半个 CPU，这时又该怎么做呢？任务组使用多少个 CPU，本质上是允许任务组使用多少的 CPU 时间片，任务调度器也是按照时间片的粒度进行调度的，CPU 性能隔离完全可以以时间片的粒度来实现。假如只允许任务组使用 0.1 个 CPU，那么只需要在一个单位时间内只允许该任务组使用 0.1 个单位时间的时间片即可。这种做法需要任务调度器记录每个任务组在当前的周期内已经使用了多少时间片资源，在使用了到达上限的 CPU 时间片资源后，不再允许该任务组进行调度即可。

15.4.3 内存性能隔离

任何程序运行都需要用到内存。在使用系统虚拟化技术创建一台虚拟机时，除了 CPU 数量以外，通常还需要为这台虚拟机指定分配的内存大小以限制其能使用的最大内存。内存大小的隔离也是进程级虚拟化很重要的一部分。如何限制一个任务最多能使用的内存大小呢？要解决这个问题，首先要知道的就是每个任务当前使用了多少内存。

第四章介绍了操作系统的内存管理机制。任务使用的所有内存都是通过内存管理机制来分配和释放的。如果内存管理机制能够在分配内存和回收内存的时候维护任务到其内存使用量的关系，我们便能够知道每个任务内存的实时使用量。每次分配内存时，内存管理机制可以检查当前任务是否能够分配一定大小的内存。例如任务 A 的最大内存使用量为 100MB，它首先分配了 60MB 的内存，由于原先它的内存使用量为 0，加上这 60MB 的内存它也不会超出限制，所以内存分配器允许了此次分配，给任务 A 分配了 60MB 的内存。随后任务 A 又要分配 50MB 的内存，内存分配器发现任务 A 已经使用了 60MB 内

存，如果分配了这 50MB 内存，就超出了它的最大内存——100MB，所以内存分配器拒绝执行此次内存分配请求。通过这种方式，便能对任务的内存使用量进行限制。要想实现任务组粒度的内存使用量限制，只需要保证同一个任务组内所有任务的内存实时使用量之和不超过最大值即可。

这种方法在每次内存分配和释放时都需要修改一个计数器，通常会带来不小的开销，尤其是在内存分配十分频繁的情况下。

小思考

还有什么和内存相关的资源可能会受到竞争呢?

在生活中我们可能会遇到当自己在学习或者工作时，邻居家却发出巨大的噪音，使我们无法安心学习或工作。邻居家与我们家是物理隔开的，就像已经实现了内存隔离的情况一样，我们使用的内存和邻居使用的内存互不干扰。但实际情况却是邻居虽然没有使用分配给我们的内存，却能通过其他方式影响我们使用自己的内存，这是为什么呢?

除了内存空间的隔离以外，内存带宽也是对内存性能影响很大的一个因素。如果邻居占用了大量的内存带宽，势必会对我们造成一定的影响，如何防止吵闹的邻居（Noisy Neighbors）呢？即如何实现内存带宽的隔离呢？这个问题在操作系统层面并不是一个容易的问题，常用的解决方案一般是通过限制 CPU 时间来限制内存带宽，例如当任务使用完带宽限额后便不再调度该任务，直到下一计时周期。这种方法并不能直接对内存带宽进行限制和隔离，而是通过限制其他的资源来间接实现对内存带宽的限制，容易与 CPU 隔离的需求产生冲突和矛盾。所以这种方法不够通用，只在特定场景下可行。

吵闹的邻居的现象在云场景中经常出现，为了解决这个问题，Intel 从硬件层面提供了内存带宽分配技术（Memory Bandwidth Allocation） [4]，该技术可以直接从硬件层面给访存操作添加一定的延迟以实现内存带宽的限制。通过给吵闹的邻居添加访存延迟，便可以减少其噪音，使其变得不再吵闹。

15.4.4 I/O 性能隔离

本节将会以磁盘 I/O 为例来介绍 I/O 性能隔离。由于磁盘带宽通常较小，如果系统中存在许多大量访问磁盘的应用程序，磁盘带宽很容易成为性能瓶颈。应用程序想要进行磁盘 I/O，通常会经过图 9.1所示的存储栈，理论上来说，从上至下的每一层都有可能被用来实现磁盘 I/O 隔离，只是实现的难度和开销不一样。虚拟文件系统会调用具体的文件系统接口来处理用户的各种请

求，在需要访问存储设备时，具体的文件系统会创建对存储块的访问请求并转发给 I/O 调度器。I/O 调度器则会根据具体的策略来合理地将文件系统的访问请求发送给设备驱动，最后再由设备驱动来访问存储设备。

对于最大值的资源控制模型而言，文件系统、块抽象和 I/O 调度都有能力维护一个从任务到已发出的 I/O 量的关系，所以都能够稍作修改以支持最大值的资源控制模型，即在固定的周期只允许一个任务发出不多于最大限制值的 I/O 请求。

从文件系统和块抽象实现按比例分配的资源控制模型较为复杂，而 I/O 调度器却可以很容易地按照比例来进行调度，Linux 提供的 CFQ（类似于第 refsubsubsec:sched-cfs 介绍的 CFS 任务调度器）等 I/O 调度器都提供了按照权重调度的机制，感兴趣的读者可以去更深入地了解这些调度器的具体实现。

15.5 案例分析：Linux Control Groups

本节主要知识点

- ❑ 什么是控制组技术？
- ❑ 为什么需要控制组技术？
- ❑ 如何使用控制组来限制任务使用的资源量？
- ❑ 控制组在 *Linux* 中是怎样实现的？针对哪些资源进行了限制？

为了解决性能隔离的问题，Linux 提出了名为 Cgroups 的解决方案。

15.5.1 Cgroups 架构和接口

Cgroups 是 Control Groups（控制组）的缩写，是 Linux 内核（从 Linux 2.6.24 开始）提供的一种可以用来将任务分组，并对多种物理资源进行限制、监控和隔离的功能。其中分组功能是在内核中的核心 cgroup 代码中实现的，而资源跟踪和限制则是在每一种资源（例如 CPU、内存）的子系统中分别实现的。Cgroups 通过一个叫做 cgroupfs 的伪文件系统提供了用户接口。

接下来本节会通过图 15.7中的例子介绍 cgroups 中的一些常用术语和基本的知识。

任务（Task） 任务是 cgroups 中用于管理的基本单位。在 cgroups v1 中，一个任务就是系统中的一个线程。例如在图 15.7中，我们有 task1 和 task2 两个任务。

控制组（Cgroup） 简单来说，控制组是一些按照某种标准划分的任务的集合。一个任务可以加入到一个控制组，也可以在不同的控制组之间进行迁移。Cgroups 中的资源限制都是以控制组为单位实现的。一个控制组内的所有任务可以使用 cgroups 为该控制组分配的资源，同时受到 cgroups 为该控制组所设定的限制。在图 15.7中，我们有 3 个控制组，cgroup1 只有 task1 一个任务，cgroup2 有 task2 一个任务，而 cgroup3 则由 task1 和 task2 共同组成。

子系统（Subsystem） 子系统是一个内核组件，每个子系统会对应一种类型的系统资源。内核中有许多已经实现的子系统，例如 CPU 和内存子系统等。通过这些子系统，用户可以监控和限制一个控制组使用的对应类型物理资源。例如，用户可以限制一个控制组所能使用的 CPU 时间或内存大小、获取该控制组已使用的 CPU 时间和内存等。由于子系统可以控制物理资源分配，它有时候也会被称作资源控制器（Resource Controllers）。在图 15.7中，我们有着 3 个子系统，分别是 cpu、cpuacct 和 memory，cpu 子系统负责限制 cpu 资源，cpuacct 子系统负责对使用的 cpu 资源进行监控和数据统计，而 memory 子系统负责对内存使用情况进行监控和限制。

层级（Hierarchy） 内核将控制组组织成树状结构，每一棵由控制组构成的树称为一个层级结构，它们拥有各自的 cgroups 文件系统。用户可以通过在一个 cgroups 文件系统中创建、删除和重命名子目录来构造或修改这一层级结构。在一个层级结构中，每个子目录都是一个控制组，子目录下有着许多文件，其中有一些文件是所有子系统都拥有的，例如 cgroup.procs 等，向该文件中写入一个进程的 PID 可以将这个进程加入该控制组；还有一些文件则是各个子系统所独有的，例如 cpu 子系统有着一些以 cpu 开头的文件，通过读写这些文件，我们便能与 cgroups 进行交互。任务在每个层级结构中可以属于一个控制组，同时我们可以为每个层级结构附加一些子系统，但在同一时刻，每个子系统只能被附加到一个层级结构中。这是因为同一任务在不同的层级结构中可能会属于不同的控制组，若一个任务属于的两个控制组对于同一种系统资源的限制不一致，则会产生矛盾。如图 15.7所示，我们将 cpu 和 cpuacct 两个子系统附加到了层级结构 A 中，将 memory 子系统附加到了层级结构 B 中。层级

结构 A 被挂载到了*/cpu,cpuacct* 目录下，而层级结构 B 被挂载到了*/memory* 目录下。

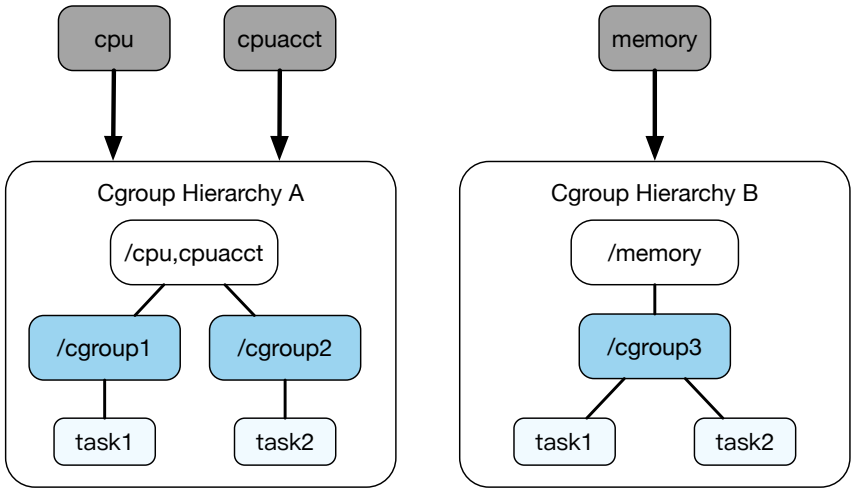

图 15.7: cgroups 概览

在介绍完了这些常用术语之后，让我们再来回顾一下图 15.7所示的例子，系统中有 3 个子系统 cpu、cpuacct 和 memory。我们将 cpu 子系统和 cpuacct 子系统挂载到*/cpu,cpuacct* 目录下，形成了 cgroup 层级结构 A，memory 子系统被挂载到了*/memory* 目录下，形成了层级结构 B。在层级结构 *A* 中，我们建立了两个控制组 *cgroup1* 和 *cgroup2*，然后将任务 *task1* 加入了控制组 *cgroup1*，将任务 *task2* 加入了控制组 *cgroup2*。在层级结构 *B* 中，只有一个控制组 *cgroup3*，两个任务都属于这个控制组。

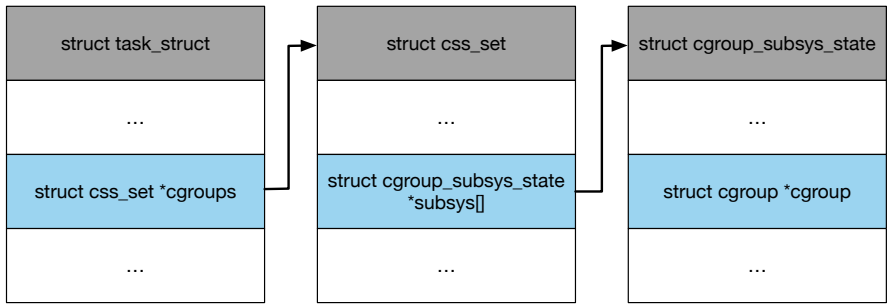

图 15.8: kernel 如何找到一个任务所属于的 cgroup

15.5.2 CPU 控制器

CPU 控制器（子系统）主要用于控制一个控制组中所有任务可以使用的 CPU 时间片，读取与 CPU 相关的数据则依赖于另外一个子系统——cpuacct 子系统，在 cgroups 中这两个子系统通常会一起出现。CPU 控制器有基于 Linux CFS 调度器和实时调度器的两种实现，本节仅介绍前一种。

用户可以通过调节控制组在一个周期内的 CPU 配额来限制该控制组所能使用的 CPU 时间。其中周期和配额都是可以修改的。用户可以通过以下文件来使用 CPU 控制器：

- `cpu.cfs_period_us`。该文件保存了周期长度（单位为微秒）。通过读写该文件可以获取或修改周期长度。
- `cpu.cfs_quota_us`。该文件保存了一个周期内控制组所能使用的 CPU 时间配额（单位为微秒）。可以通过读写该文件获取或修改控制组的配额。
- `cpu.stat`。该文件为只读文件，保存了 CPU 时间相关的统计数据。

在每个周期内，一个控制组被分配了一定的 CPU 时间配额，当这个控制组的任何任务变为运行状态时，这个控制组的配额会随之减少。一旦该控制组用完了配额之后，其组内的任务都无法再被调度，直到下一个周期开始，所有控制组的配额被重置之后，这个控制组的任务才能继续运行。在多核系统中，配额是可以大于周期长度的，这是由于一个控制组内的所有任务共享同一个配额，而一个控制组内的不同任务可以同时运行在不同的 CPU 上。

前面已经介绍过，通过向`cpu.cfs_quota_us`文件写入一个值可以修改一个控制组的 CPU 时间配额，那么具体而言，设置的配额该如何生效呢？在“/kernel/sched/core.c” 中有如代码片段 15.4所示的一段代码。

```
1 static struct cftype cpu_legacy_files[] = {
2   {
3     .name = "cfs_quota_us",
4     .read_s64 = cpu_cfs_quota_read_s64,
5     .write_s64 = cpu_cfs_quota_write_s64
6   }
7 };
```

代码片段 15.4: CPU 控制器文件接口

如代码片段 15.4中的`write_s64`所示，当我们向 *cpu.cfs_quota_us* 文件中写入一个值时，cgroups 会调用`cpu_cfs_quota_write_s64`函数，随后的

调用过程如代码片段 15.5所示，最后会调用到函数tg_set_cfs_banwidth，这个函数最终会修改task_group中的cfs_bandwidth结构体中的period和quota。

```
1  static int cpu_cfs_quota_write_s64(
2             struct cgroup_subsys_state *css,
3             struct cftype *cftype,
4             s64 cfs_quota_us)
5  {
6    return tg_set_cfs_quota(css_tg(css), cfs_quota_us);
7  }
8
9  static int tg_set_cfs_quota(
10            struct task_group *tg,
11            long cfs_quota_us)
12 {
13   u64 quota, period;
14   ...// 省略给 quota 和 period 赋值的代码
15   return tg_set_cfs_bandwidth(tg, period, quota);
16 }
```

代码片段 15.5: 修改 *cpu.cfs_quota_us* 的函数调用过程

6.6.1节中已经介绍过 CFS 调度器的工作原理。除了按照任务为调度实体进行调度外，CFS 调度器还提供了按组调度的功能，也就是以task_group（任务组）为调度实体。cfs_bandwidth是task_group结构体中的一个域，其中包含了这个调度实体所剩下的运行时间runtime。系统会启动一个高精度定时器，每隔 *period* 将cfs_bandwidth中的runtime重置为 *quota*。其中 *quota* 和 *period* 是我们自己配置的，修改过程上面已经介绍了。当调度组中的任务需要进行调度时，它会先以运行队列cfs_rq的身份向调度组申请一个时间片，调度组会从runtime中给它分配一段时间。如果runtime不够用了，调度组会将这个调度实体从它的运行队列中移除（dequeue_entity）。

在下一个周期runtime被重置之后，也就是有了足够的时间片之后，调度组会将之前出列的调度实体重新加入原本的运行队列（enqueue_entity）。通过这种办法，我们就实现了精确控制一个控制组内所有任务使用的 CPU 时间。

基于周期和配额的方法按照绝对值来限制控制组的 CPU 时间，类似于前面介绍过的最大值资源控制模型。这种方法简单直接，但限制最大值在某些情况下可能会造成资源浪费。所以除了这种方法外，CPU 控制器还提供了按照比例来设置不同控制组的 CPU 时间的方法。以下文件是 cgroups 按照相对值

来设置控制组的 CPU 时间的接口：

- `cpu.shares`。该文件保存 CPU 的相对值，默认为 1024。通过读写该文件可以获取/修改控制组所能使用的 CPU 时间。

假如我们有两个控制组 A 和 B，A 的 shares 值为 300，B 的 shares 值为 100，那么当 A 和 B 都在满负载运行时，A 能够获得 75% 的 CPU 时间，B 能够获得 25% 的 CPU 时间。当控制组 A 没有任务需要 CPU 时，控制组 B 能够使用超过 25% 的 CPU 时间，这样可以使 CPU 资源得到更充分的利用。

修改 *cpu.shares* 的原理与前面修改 *cpu.cfs_quota_us* 的原理相同，最后将这个值更新到了该`task_group`中每一个调度实体的`load.weight`中（即调度的权重）。根据第6.6.1节介绍的 CFS 调度原理，`load.weight`越大，`vruntime`就越小，也就意味着这个调度实体能够使用的 CPU 时间越多。

15.5.3 内存控制器

内存控制器可以控制一个控制组中所有任务所能使用的物理内存总量。内存控制器中常用的文件如下：

- `memory.limit_in_bytes`。该文件保存控制组的物理内存限制，单位为字节。用户可以通过读写该文件获取 / 修改当前限制。
- `memory.stat`。该文件为只读，保存内存相关的统计数据。

当一个控制组用尽了自己的内存资源后，如果组内的某个任务继续申请内存，系统的默认操作是直接杀掉该任务。由于默认行为显得过于简单粗暴，内存控制器提供了`memory.oom_control`文件作为修改默认行为的接口。该文件的默认值是 0，代表当内存不足时终止申请内存的任务；通过向这个文件写入 1 可以使系统在内存不足时不终止继续申请内存的任务，而是使任务进入等待状态，直到有额外的内存可以使用。

到这里我们已经了解了如何使用 cgroups 的接口去操作内存控制器，那么内存控制器究竟是如何实现的，如何限制一个控制组所能使用的内存大小呢？

一种直观的做法是，每当要为一个控制组分配新的物理内存时，操作系统首先检查一下这个控制组已使用的内存是否达到了上限，只有当这个控制组已使用的内存低于限制值时，才为其分配新的物理内存。因此，操作系统需要记录每个控制组已经使用了多少内存。

在 Linux 中，对于每个控制组，内存控制器中都有一个结构体`mem_cgroup`与其对应。在每个`mem_cgroup`中又有一个名

为memory的page_counter结构体，它跟踪了每个mem_cgroup的内存使用情况，也就是每个控制组的内存使用情况。每当我们要为这个mem_cgroup分配新的内存时，便会调用page_counter_try_charge函数（如代码片段 15.6所示）来尝试对这些新的内存进行 charge 操作，即将这些页加入到该page_counter里。如果加上这些页之后，这个page_counter的值大于了我们给它设定的最大值，则会导致这次charge失败，随后根据其他设置可能会进一步触发内存不足的操作oom（Out of Memory）。如果没有超过最大值的限制，则此次charge成功，新的页成功被加入mem_cgroup的计数器中。

```
1  bool page_counter_try_charge(
2       struct page_counter *counter,
3       unsigned long nr_pages,
4       struct page_counter **fail)
5  {
6    ...
7    for (c = counter; c; c = c->parent) {
8      long new;
9      new = atomic_long_add_return(nr_pages, &c->usage);
10     if (new > c->max) {
11       atomic_long_sub(nr_pages, &c->usage);
12       ...
13       goto failed;
14     }
15   }
16   ...
17 failed:
18   for (c = counter; c != *fail; c = c->parent)
19     page_counter_cancel(c, nr_pages);
20   return false;
21 }
```

代码片段 15.6: 内存 charge 的函数调用流程

反之，每当内存要被释放时，需要使用相反的流程来进行uncharge操作。通过这些操作，我们成功地记录了每个控制组所使用的内存大小，也成功地限制了每个控制组所能使用的最大内存。

那么如何修改一个控制组所能使用的最大内存呢？前面已经介绍过，我们可以通过向 *memory.limit_in_bytes* 中写入一个值来修改这个控制组所能使用的最大内存。那么这个值又是如何生效，如何具体限制该控制组能使用的内存的呢？

“/mm/memcontrol.c” 中有这样一段代码（如代码片段 15.7所示）。

```
static struct cftype mem_cgroup_legacy_files[] = {
  {
    .name = "limit_in_bytes",
    .private = MEMFILE_PRIVATE(_MEM, RES_LIMIT),
    .write = mem_cgroup_write,
    .read_u64 = mem_cgroup_read_u64
  }
};
```

代码片段 15.7: 内存控制器文件接口

当我们向内存控制器文件接口 *memory.limit_in_bytes* 中写入一个值时，cgroups 会调用mem_cgroup_write函数。随后的调用过程如代码片段 15.8所示，最终会修改该控制组对应的mem_cgroup中名为memory的page_counter的最大值。

```
1  static ssize_t mem_cgroup_write(
2                 struct kernfs_open_file *of,
3                 char *buf, size_t nbytes,
4                 loff_t off)
5  {
6    struct mem_cgroup *memcg =
       ↪  mem_cgroup_from_css(of_css(of));
7    ...
8    ret = mem_cgroup_resize_max(memcg, nr_pages, false);
9    ...
10 }
11
12 static int mem_cgroup_resize_max(
13             struct mem_cgroup *memcg,
14             unsigned long max, bool memsw)
15 {
16   ...
17   struct page_counter *counter = memsw ? &memcg->memsw
       ↪  : &memcg->memory;
18   ret = page_counter_set_max(counter, max);
19   ...
20 }
21
22 int page_counter_set_max(
23     struct page_counter *counter,
24     unsigned long nr_pages)
25 {
26   ...
27   old = xchg(&counter->max, nr_pages);
28   ...
29 }
```

代码片段 15.8: 修改 *memory.limit_in_bytes* 的函数调用过程

而前面已经介绍了内存控制器就是使用这个page_counter中的最大值来限制控制组所能使用的内存上限的。至此，我们便对从向 *memory.limit_in_bytes* 这个文件中写入最大值开始到内存控制器真正对这个控制组施加限制的整个流程有了清晰的认识。

15.5.4 存储控制器

存储控制器主要用于控制一个控制组中所有任务可以使用的块设备 I/O 速率，实现于图 9.1中的块设备和 I/O 调度层。存储控制器支持两种资源调度策略：I/O 调度权重和最大 IOPS/BPS 限制。

I/O 调度权重 I/O 调度权重策略是基于 CFQ（Complete Fair Queueing）调度算法实现的，在使用这种策略之前必须确保所使用的 I/O 调度算法是 CFQ 调度算法。（新的内核所使用的 Budget Fair Queueing 调度算法也支持该策略）

在该策略中，以下两个文件被用来控制具体的调度策略：

- *blkio.weight*。该文件保存每个控制组的权重，是一个从 10-1000 的整数。通过读写这个文件，用户可以获取或修改控制组当前的 I/O 调度权重值。
- *blkio.weight_device*。该文件允许控制组对不同的设备设定不同的权重值，当这个值被设定后，会覆盖掉 *blkio.weight* 的限制。

在 CFQ 调度中，每个控制组都有一个结构体`cfq_group`与之对应。与 CPU 调度中的 CFS 调度器选择`vruntime`最小的调度实体进行调度类似，CFQ 调度算法会选择`vdisktime`最小的`cfq_group`进行调度。每用完一个时间片后，这个`cfq_group`的`vdisktime`会增加，其增加的幅度与我们设置的`weight`成反比，也就是`weight`越大，`vdisktime`增加的越少。而在进行调度时，CFQ 会选择`vdisktime`最小的`cfq_group`进行调度。因此，设置的`weight`越大，其 I/O 请求被 CFQ 调度算法服务的机会就越大，直观来说就是该控制组所占用的 I/O 带宽越大。

最大 BPS/IOPS 限制 该策略又细分为最大带宽限制策略和最大 IOPS 限制策略，两者都是限制控制组使用设备 I/O 的速率，只是对这个速率的衡量方法不同。该策略主要会使用到如下几个文件：

- *blkio.throttle.read_bps_device* 和 *blkio.throttle.write_bps_device* 。这两个文件允许控制组对不同的设备分别设置读写带宽的上限值，单位为字节每秒。
- *blkio.throttle.read_iops_device* 和 *blkio.throttle.write_iops_device* 这两个文件与上述文件功能类似，只是单位变为了每秒的 I/O 操作数量。

如果带宽策略和 IOPS 策略同时使用，那么控制组会同时受到这两个策略的限制，即设备 I/O 的带宽不能超过最大带宽限制，同时每秒进行的 I/O 操作数量也不能超过 IOPS 所设定的上限。

除了对块设备 I/O 速率进行控制以外，存储控制器还可以报告许多与块设备 I/O 相关的统计数据，本节不一一赘述。

15.5.5 其他控制器

除了上述几种子系统以外，cgroups 中还有着许多子系统。下面我们将会简单介绍其他的一些常用的子系统，不再对其实现原理进行介绍。

CPUSET 控制器 CPUSET 控制器提供了一种为控制组里的任务指定 CPU 节点和内存节点的机制。当系统中存在多个 CPU 节点和内存节点时，我们可以方便地使用 CPUSET 控制器来为每个控制组分配运行节点。

PID 控制器 进程标识符 PID 是一种有限的基本资源。PID 控制器提供了一种限定控制组所能拥有的最多 PID 数量的机制。在 Linux 中，PID 既能表示进程又能表示线程，所以 PID 控制器可以限制控制组所能拥有的进程或线程的最大数量。当一个控制组的进程数达到设定的上限后，这个控制组内的所有任务都无法再创建新的进程或线程。

设备控制器 设备控制器提供了一种设置控制组中任务对于设备的访问权限的机制。允许或拒绝控制组中的任务访问设备的机制。在图 15.9中，控制组 A 能够完全访问设备 A 和设备 B，而控制组 B 无法访问设备 A，对于设备 B 仅有只读权限。

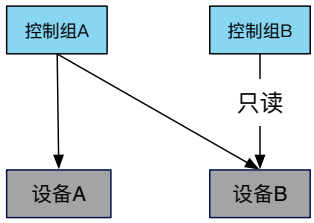

图 15.9: 设备控制器实例

FREEZER 控制器 通过该控制器，用户可以冻结（挂起）或者恢复一个控制组中的任务。

除了上述控制器以外，cgroups 中还包含许多其他控制器，本节不再赘述。

15.6 案例分析：Serverless 和 AWS Lambda

15.6.1 Serverless

Serverless（无服务计算）是一种新型的云计算模型，使用者无需向云服务商租赁虚拟机或容器，只需要提交一段写好的代码，便可以自动部署工作。Serverless 主要具有如下一些特点：

无状态 Serverless 是无状态的，即无法在运行环境中保存状态，一般 serverless 函数会利用外部的存储节点来读取或保存数据。

事件驱动 Serverless 中代码的运行是由事件驱动的，每当有定义好的事件发生时，便会自动运行这段代码。事件的定义可以有很多种，例如一次 http 请求或是一次数据库修改，都可以成为触发的事件。有多少事件触发，这段定义好的代码便会被运行多少次。由于 serverless 是无状态的，所以适合按需启动的事件驱动模型。

自动弹性伸缩 在事件驱动模型下，每当有新的事件触发，serverless 便会自动生成一个实例来运行这段代码，当这段代码运行完毕，这个实例就会被收回。不同实例之间是相互隔离的，互不干扰。当事件触发得十分频繁时，serverless 会自动生成大量的实例来支持实际的负载，可以容易地实现高并发。

细粒度计费 Serverless 按照使用量计费，客户的代码实际上使用了多少资源，就需要付多少钱。当没有任何事件发生时，便不会有任何实例运行代码，也就不会消耗任何资源。当事件触发不频繁、代码运行时间较短时，使用 serverless 可以大大节约客户的成本。

快速部署 在 serverless 的模式下，应用程序开发者只需要关注应用程序本身的逻辑，省去了服务器管理和运维的成本。同时，serverless 一般会由多个函数组成函数链，每个函数自身的功能都非常简单，也会为开发者提供一定的便利。

Serverless 同时也有着一些缺点。首先，由于 serverless 是事件驱动，只有当事件触发时才会生成实例来运行函数，这就意味着每次运行函数都需要等待实例启动，造成函数的性能下降。这个缺点可以使用许多方法来缓解，例如事先准备好已经启动的实例等。同时，由于 serverless 是无状态的，所有的存

取数据都需要通过网络访问存储节点，这也意味着当函数需要使用大量数据时，读取数据可能会成为性能瓶颈，针对这一问题，可以通过将函数迁移到存储节点上来解决。

15.6.2 AWS Lambda

AWS Lambda 是什么 AWS Lambda（简称为 Lambda）是 Amazon 推出的 serverless 服务，允许用户上传一段代码而无需配置或管理任何服务器。当有请求到来时，AWS Lambda 会自动执行用户上传的代码并自动扩展。用户只需要按照代码实际运行时间来付费，计费方式是使用的内存量 * 时间，时间精确到 100ms。Lambda 目前支持 Node.js、Python、Ruby、Java、Go、C#、PowerShell 等编程语言。

AWS Lambda 的架构 图 15.10展示了 Lambda 的高层架构。对于函数调用的请求会首先到达前端，在前端进行验证和检查，通过之后会加载相应函数的元数据。随后前端会让 Worker Manager 来为这个请求分配对应的 Worker。Worker Manager 负责函数到 Worker 的路由工作，其最核心的思路是将同一个函数路由到尽可能少的 Worker 上。当 Worker Manager 确定了负责处理这个函数的 Worker 之后，它会让前端将工作负载直接发给对应的 Worker，而不是通过自己来发送，这样可以减少 Round-Trips（即网络包的来回数量）。

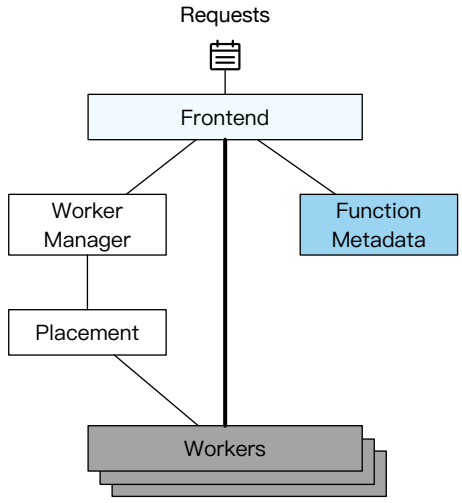

图 15.10: Lambda 简化高层架构图

Worker 是负责执行函数的模块。每个 Worker 上有着许多 slot（插槽），每个 slot 为一个函数提供预先加载好的执行环境。当 Worker Manager 为某个函数寻找 Worker 时，它首先会检查是否还有可供这个函数执行的 slot 存在，如果有，它就会告知前端，让前端将这个调用请求分配到这个可用的 slot 上；如果没有现成的 slot，这时 Worker Manager 就会调用 Placement 服务，来为这个函数创建一个新的 slot。

Placement 负责 slot 的管理工作，包括 slot 的创建、资源限制和回收等。当 Worker Manager 需要创建一个新的 slot 时，Placement 会综合考虑所有 Worker 的共享资源利用率，再决定应该在哪个 Worker 上创建。但前面已经讲过，如果 Worker Manager 找到了能够使用的 slot，它可以直接告诉前端这个 slot 可用，这个过程没有经过 Placement，那么它是如何保证在这个过程中 Placement 不会回收这个 slot 呢？实际上，Placement 使用了一个基于时间的租约（Lease）将这个 slot 借给了 Worker Manager，使得 Worker Manager 在一段时间内可以自主决策，而不需要每次都调用 Placement 服务。

现在我们知道了 Lambda 大致是怎么接收并部署函数的，但具体每个函数是在哪里执行的？Lambda 又是如何提供函数级别的隔离的呢？最初 Lambda 选择使用虚拟机来提供不同用户之间的隔离，同时使用 Linux 容器来隔离同一个用户的不同函数。不同用户的函数会运行在不同的虚拟机内，而同一个用户的不同函数会运行在同一个虚拟机内的不同容器内。但 Amazon 对于这种架构并不满意。首先，虚拟机本身比较笨重，如果使用虚拟机运行函数，那么一台物理主机就无法服务于成百上千个不同的用户；同时，容器的隔离性相比虚拟机较弱，不足以满足函数对于安全的需求。因此，Amazon 开始为 serverless 服务探索新的架构，他们对于新的架构有以下 6 个要求 [9]：

- 强隔离性：必须安全地在同一硬件上运行多个不同的函数，能够抵御不同类型的攻击。
- 低开销、高密度：可以在一台机器上以最少的开销运行数千个函数。
- 高性能：函数的运行状态和性能都要与在本地运行相似。
- 兼容性：允许函数包含原生的二进制文件和库，用户不需要对函数作任何修改。
- 快速启动和清理：可以快速生成新的实例并回收无用的实例。
- 软分配支持：即使给函数分配了更多的资源，函数也只会按需使用资源。

基于这些要求，Amazon 开发了轻量级虚拟化运行环境 AWS Firecracker[7]，并于 2018 年 12 月开源了 FireCracker 的实现。同年，Firecracker 被用到 Lambda 生产环境中。FireCracker 没有完整地模拟整个计算机硬件平台，只实现了运行函数的必要功能，将 KVM+QEMU 架构中的 QEMU 完全替换，构建了一个全新的虚拟机监控器。另外，Firecracker 使用 MicroVM 运行函数，其同时具有传统虚拟机的安全性以及容器的速度和资源利用率。通过 Firecracker 启动一个 MicroVM 只需要 5MB 的内存和低于 125ms 的启动时间，均远小于传统虚拟机。Firecracker 为实现容器和函数级别的隔离提供了一个新的选择。对于 Firecracker 的具体实现感兴趣的读者可以自行参考其官网或论文 [9]。

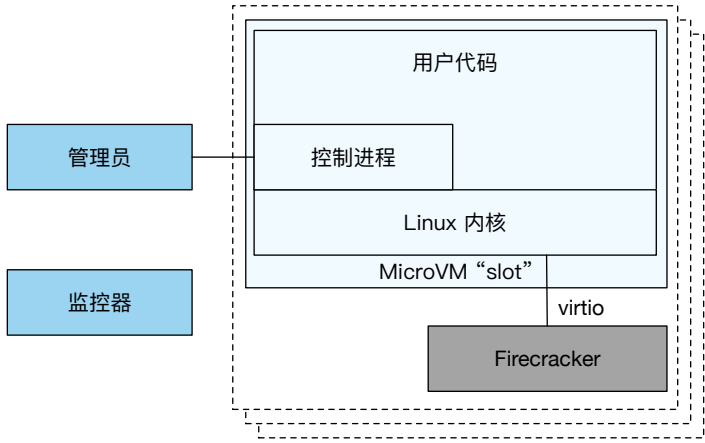

图 15.11: Lambda worker 架构

Firecracker 为 Lambda Worker 在同一物理设备上运行许多不同的负载提供了关键的安全边界。如图 15.11所示，每个 Lambda Worker 由成百上千个 MicroVM 组成，具体的数量由每个 MicroVM 需要的资源多少决定。每个 MicroVM 都是一个只运行唯一客户代码的沙盒（即前面我们所说的 slot），里面包含着一个最小的 Linux Kernel 和一个控制进程（Lambda Shim）。

每个 Worker 有着唯一的管理员（Micro Manager），它管理着这台机器上所有的 MicroVM，同时为 Placement 提供了 slot 的管理和同步相关的 API。当一个调用请求到达前端后，Worker Manager 会告诉前端这个请求该由哪个 slot 进行处理，之后前端会直接将这个请求转发给该 Worker 对应的管理员。管理员通过一个 TCP/IP 套接字与对应的 MicroVM（即 slot）中的控制进程进行通讯，将请求发给该 MicroVM。当请求完成后，管理员通过该套接字接收函数处理的返回结果或出错信息，然后将结果返回给前端，再由前端将结果返回

给用户。

每个 Worker 上还运行了一些用于监控和日志记录的进程，从而提供了人工或自动的监控和预警功能。比如，这些进程会告知 Placement 每个 Worker 上的负载量如何，便于 Placement 决定新的 slot 应该放在哪个 Worker 上。

前面已经提到，Firecracker 的启动时间是 125ms。虽然 125ms 的启动时间很快了，但是对于 Lambda 这种 serverless 服务来说仍然不够快。因此，Micro Manager 提供了一个由已经提前启动的 MicroVM 组成的 MicroVM 池（MicroVM Pool），用于更快响应 Placement 对新的 slot 的请求，从而提供更快的启动速度。

15.7 思考题

1. 如果我们想利用 cgroups 让一个任务只能用 1GB 的内存，我们应该怎么操作。

2. 假设我们有两个任务 A 和 B，我们想要让这两个任务的 CPU 使用时间比例为 2:1，且磁盘 I/O 速度比例为 1:2，我们该如何利用 cgroups 实现这一目标?

3. 如何使用 chroot 运行一个 shell?

4. AWS Lambda 采用的第一种隔离方案是什么？现有方案与原有方案相比有哪些好处?

5. 以 root 身份创建一个目录`ns-demo-dir`，然后在当前目录下创建一个文件`delete-me`。以普通用户身份使用 unshare 命令创建一个新的用户命名空间，将当前用户映射为 root 用户后在命名空间内执行 bash，在该 bash 中以 root 用户身份尝试删除 delete-me 文件，删除操作是否能够成功？请解释原因。

6. 使用 iproute2 工具，搭建与图 15.6类似的架构，实现两个网络命名空间与外部网络连通。提示：使用 ip 命令，创建一个网桥、两对 veth 虚拟设备对、两个网络命名空间。

7. 思考 Docker 是如何使用命名空间来创造隔离环境的。了解 Docker 的配置选项，例如`--network`、`--ipc`等，这些参数应该如何搭配不同命名空间来实现?

参考文献

[1] Alibaba economy cloud native practice. https://developer.aliyun.com/article/728327.

[2] Hypervisor-based runtime for oci. https://github.com/hyperhq/runv.

[3] Intel clear containers: Now part of kata containers. https://clearlinux.org/news-blogs/intel-clear-containers-now-part-kata-containers.

[4] Introduction to memory bandwidth allocation. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[5] Production-grade container orchestration. https://kubernetes.io/zh/.

[6] runc. https://github.com/opencontainers/runc.

[7] Secure and fast microvms for serverless computing. https://firecracker-microvm.github.io/.

[8] The speed of containers, the security of vms. https://katacontainers.io/.

[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[10] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1846–1855, 2017.

轻量级虚拟化：扫码反馈