调研报告

生态支持

邓胜亮

何纪言

缑慧星

赵敏帆

曾明亮

项目简介

该项目通过对操作系统及相关软件的修改与集成，实现 Docker 在物理机直接装载运行，从而可以在依托 Docker 原有的成熟丰富的生态环境、保持用户接口不变、不借助其他虚拟化技术的前提下，获得更大的计算性能、存取性能和网络性能优化空间、简化部署流程、降低运维成本、获得更高的安全性与隔离性，应对复杂多变的业务情景，更易实现弹性伸缩的目标，并尝试支持多种处理器架构。

项目背景

从传统虚拟化到容器化

传统虚拟化技术

虚拟化技术是指隔离任务，使每个任务看起来独占整个计算机，对计算资源进行抽象。可分为平台虚拟化、资源虚拟化、应用程序虚拟化等类别。通常所说的虚拟机是采用了平台的虚拟化。虚拟化技术经历了从硬件虚拟化、模拟执行、到二进制翻译的发展历程。不同虚拟化技术的区别在于隔离性、性能和虚拟的层次等方面。

虚拟化对象与层次

- 内存虚拟化
  - “机器内存”由虚拟化层管理，虚拟机看到的物理内存是被虚拟化的；
  - 虚拟化层要把两级地址映射“压缩”成一级映射。
- 设备虚拟化（三种方式）
  - 虚拟设备，共享使用；
  - 直接分配，独占访问；
  - 在物理设备的辅助下，虚拟出多个 “小设备”。
- 操作系统级虚拟化

  虚拟机里运行一个或多个进程、虚拟机与主机共享一个内核。
  - 满足：文件系统隔离、命名空间隔离、资源的限制和审计；
  - 隔离执行进程实现虚拟化的容器：轻量，启动更快，内存开销、调度开销更小，访问 I/O 设备不需经过虚拟化层，没有性能损失。

虚拟化技术

- 硬件层面

  由 CPU来负责捕获特权指令,减少二进制翻译的开销。增加 RootMode 供 VMM 使用。特权指令由 CPU 捕获后陷入到 VMM ,处理后再返回虚拟机系统。提供了 VMCALL 指令方便半虚拟化 API 调用。但是模式切换带来的开销依然很大。

- 模拟执行

  需要开辟大空间作为"虚拟机"内存，对指令集所有指令设置对应模拟执行，虽然兼容性好安全性较好，但效率极低。

- 动态二进制翻译

  能翻译的部分翻译成目标架构机器码直接执行并缓存以重复利用，不能翻译部分陷入模拟器模拟行。guest os 在 ring1 ，VMM 在 ring0 。尽管提出了半虚拟化、硬件辅助等技术，但由于特权指令需要模拟执行，所以效率依然很低。

容器化技术

产生

Docker 是 dotCloud 公司开源的一款产品。dotCloud 公司是 2010 年新成立的一家公司，主要基于 PaaS（Platform as a Service，平台即服务）平台为开发者提供服务。但由于 Pass 市场始终处于培育阶段，dotCloud 公司表现的不温不火，于是在 2013 年，dotCloud 把内部使用的 Container 容器技术单独拿出来开源，2013 年 3 月发布 Docker 的 V0.1 版本，并且基本保持每月一个版本的迭代速度，到了 8 月，Docker 已经足够火爆。

解决的问题

在计算机系统和软件数量如云的现今，给开发、测试和运维人员带来了极大的便利，但同时也给他们的维护工作带来了极大地挑战。在 Docker 之前，已有 OpenStack 、Puppet 等解决方案出现，也比较流行，但这些解决方案使用的场景极为有限，如 Puppet 适用于需要大批量部署相同服务的应用场景，OpenStack 适合中大型企业使用。

而 Docker 作为一个开源的应用容器的引擎的出现，让开发者可以打包他们的应用及依赖环境到一个可移植的容器中，然后发布到任何运行有 Docker 引擎的机器上。Docker 集版本控制、克隆继承、环境隔离等特性于一身，提出一整套软件构建、部署和维护的解决方案。使用 Docker 时，软件部署的应用具备类似于 GitHub 的版本控制功能，对应用做一些修改，提交新版本，运行环境可以在多个版本间快速切换，自由选择使用哪个版本对外提供服务，都可以非常方便地帮助开发人员，让大家可以随心所欲地使用软件而又不会深陷到环境配置中。

发展现状

- 2013 年 10 月，dotCloud 公司索性更名为 Docker 股份有限公司，工作的重心也从 PaaS 平台业务转向全面围绕 Docker 来开发；
- 2014年6月9日，Docker 发布了 V1.0 版，Google、IBM、RedHat、Rackspace 等公司的核心人物在 DockerCon 2014 大会上，纷纷表示支持并加入 Docker 的阵营；
- Azure、Google 和亚马逊等都在支持 Docker 技术，这实际上使得 Docker 成为云计算领域的一个重要级成员，也让 Docker 成为云应用部署的事实上的标准；
- 2014 年 12 月，Docker 发布了 Docker 集群管理工具 Machine 和 Swarm，标志着 Docker 开始突破一个标准的容器框架，打造属于 Docker 自己的集群平台和生态圈；
- 2016 年 1 月，Docker 官方计划全面支持自身的 Alpine Linux，使用它构建的基础镜像最小只有 5 M；

- 2016 年 3 月，Docker 在 Github 所有项目中排第7位，在云平台管理领域排名第一，远远超 Openstack 项目。

应用场景

Docker 提供了轻量级的虚拟化，几乎没有任何额外的开销，且容器的启动和停止都能在几秒钟内完成。结合 Docker 的优点，总结出下列应用场景：

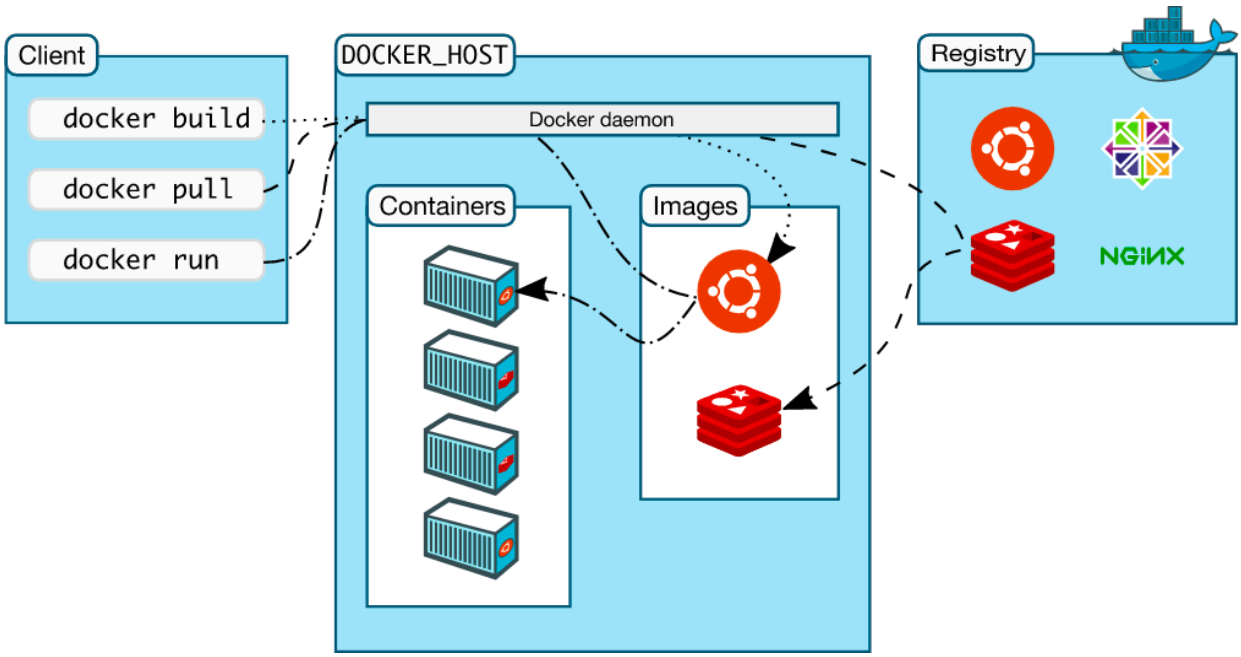
- 简化配置
- 代码流水线管理
- 提高开发效率
- 隔离应用
- 整合服务器
- 调试能力
- 快速部署

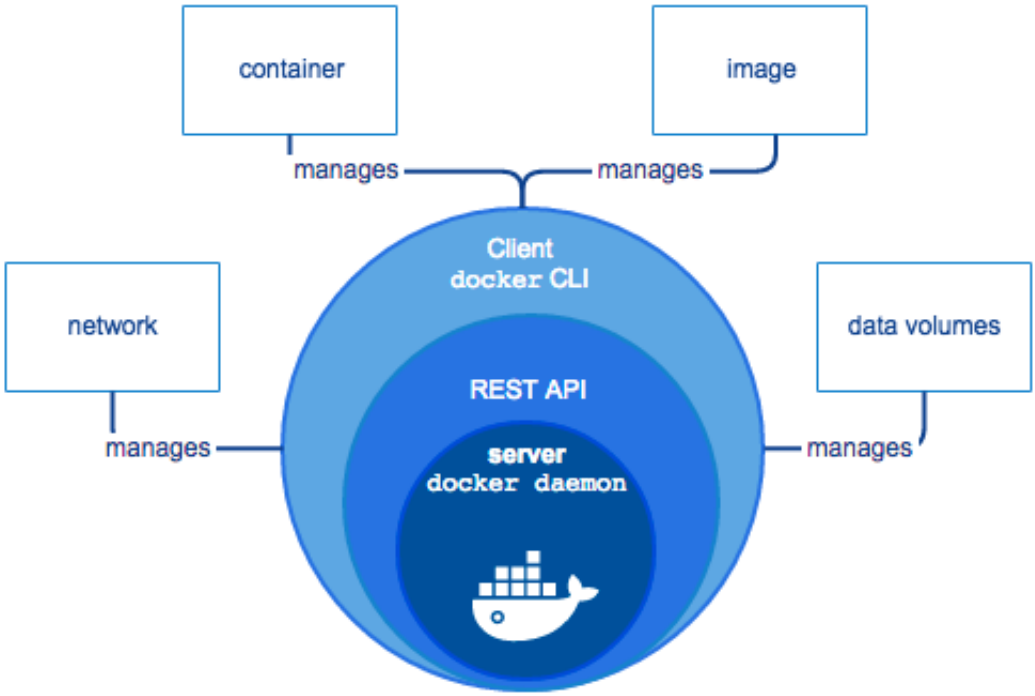
容器化技术的代表：Docker

Docker 的架构

Docker 的组件有：Docker daemon，Docker client，Docker registries 和 Docker objects。

他们的关系如下图：





Docker daemon

守护进程，监听 API，管理 Docker 对象，与其他守护进程通信。

Docker client

客户端，用户与 docker 交互的主要工具，使用 API 与 dockerd 通信，没有 dockerd 就无法工作。

Docker registries

Docker 的仓库，类似于 Git 中仓库的概念，仓库中储存了 images（随后解释）。

其中常用的公开仓库是 Docker Hub，类似于 Git 常用的仓库是 GitHub。

个人也可以搭建私有仓库，如使用 Docker Datacenter (DDC) 搭建 Docker Trusted Registry (DTR)。

Docker objects

images, containers, networks, volumes, plugins 都是 docker 的对象，下面是分别的概述：

Images 镜像

定义：An *image* is a read-only template with instructions for creating a Docker container.

关键词：只读，模板，用于创建容器。

Image 类似于虚拟机中的镜像概念，但不同的是 Docker Image 一般基于另一个 Image。

此外，一般使用 Dockerfile 来定义并构建一个 Image，Dockerfile 中的每一个语句都会创建一个 layer（随后解释），这些 layer 可以被复用，保证当 Dockerfile 添加语句或者是修改语句时，没有修改过的部分的构建结果都可以直接使用，这是 Docker 轻量，体积小，速度快的原因之一。

Containers 容器

定义：A container is a runnable instance of an image.

关键词：可执行，Image 的实例。

Container 的概念类似于虚拟机中创建好的一个虚拟机，你可以通过 Docker 客户端启动，停止，移动，删除这个容器。

通过 Docker 客户端可以管理一个容器的属性，与宿主机器的联系，如网络，储存等等。

容器由镜像和配置参数共同决定，容器默认没有永久储存（除非用户添加了这一资源），当它被移除后不会留下任何数据。

Services 服务

定义：Services allow you to scale containers across multiple Docker daemons, which all work together as a *swarm* with multiple *managers* and *workers*.

关键词：多个守护进程，管理多机器集群，属于 Docker 的进阶用法，我们目前暂不涉及。

Underlying technology

Namespaces

Docker 使用命名空间技术来完成实现环境的隔离，也就是我们的 container。这种需要以命名空间来隔离的资源有多种：

- **The pid namespace:** Process isolation (PID: Process ID).
- **The net namespace:** Managing network interfaces (NET: Networking).
- **The ipc namespace:** Managing access to IPC resources (IPC: InterProcess Communication).
- **The mnt namespace:** Managing filesystem mount points (MNT: Mount).
- **The uts namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Control groups

control groups (cgroups) 技术用于限制内存，CPU 等硬件资源。

Union file systems

UnionFS 技术正是使得 docker 轻量快速的一大原因，UnionFS 给构建 Docker 镜像提供了 building blocks（见 Docker images 中的介绍）。

Docker 可以使用 UnionFS 的多种变体，如：AUFS, btrfs, vfs, DeviceMapper.

Docker 的使用

这一部分列举 Docker 的简单使用方法。

验证安装

```
docker version
# or
docker info
```

启动和停止

```
sudo systemctl start docker
sudo systemctl stop docker
```

管理镜像

```
# 列举镜像
docker images ls
# 删除镜像
docker images rm [image_name]
```

拉取镜像

```
docker image pull [image_name]
# example
docker image pull hello-world
```

运行镜像

```
docker container run [image_name]
```

终止容器

```
docker container kill [contain_id]
```

管理容器

```
# 列举正在运行的容器
docker container ls
# 列举所有的容器（正在运行的和结束的）
docker container ls -all
```

Docker 的基础技术：cgroups

功能

cgroups 是 Linux 2.6.24 开始引入的一种资源隔离机制，用于以组为单位对进程使用的多种系统资源进行限制。

术语

cgroups 机制中的主要术语有：层级、子系统、任务和控制组，以下依次解释之：

- 层级（hierarchy）

  表现为一个文件夹，在 `mount` 时用 `-o` 选项指定要附加其上的子系统。

- 子系统（subsystem）

  一个子系统控制一类资源，比如CPU、内存、I/O。

- 任务（task）

  系统中的一个进程。

- 控制组（cgroups）

  资源控制的单位，呈现为一个目录带一系列配置文件，并且是可以嵌套的，子控制组的资源不能超过父控制组。

主要子系统

目前被广泛使用的 cgroups v1 中有如下的子系统：

- blkio

  为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等

等）。

- cpu

  使用调度程序提供对 CPU 的 cgroup 任务访问。

- cpuacct

  自动生成 cgroup 中任务所使用的 CPU 报告。

- cpuset

  为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。

- devices

  可允许或者拒绝 cgroup 中的任务访问设备。

- freezer

  挂起或者恢复 cgroup 中的任务。

- memory

  设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。

- net_cls

  使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序识别从具体 cgroup 中生成的数据包。

- ns

  命名空间子系统。

实现

cgroups 相关的代码相当分散，可以通过 `grep -nr "struct cgroup_subsys ."` 体会一下。

此外，有人评价说：

> cgroups 在内核的实现一直比较混乱，受到很多人的诟病，甚至不乏有人想将其从内核移出。
>
> 来源：http://hustcat.github.io/cgroup-v2-and-writeback-support/

现状

cgroups 分 v1 和 v2 两个版本，v1 实现较早，功能比较多，但是由于它里面的功能都是零零散散的实现的，所以规划的不是很好，导致了一些使用和维护上的不便，v2 的出现就是为了解决 v1 中这方面的问题，在最新的 4.5 内核中，cgroups v2 声称已经可以用于生产环境了，但它所支持的功能还很有限，随着 v2 一起引入内核的还有 cgroups namespace。

cgroups v1 是 systemd、Docker 都在使用的版本。

cgroups v2 在 4.5 版内核中首次出现。

Docker 的基础技术：namespace

Namespace 又称命名空间，它和 Cgroup 一起构成容器技术的两个内核特性。

chroot 与 Namespace

早先 UNIX 有一个叫做 chroot 的系统调用，它通过修改根目录把用户 jail 到一个特定的目录下。chroot 提供了一种简单的隔离模式：chroot 内部的文件系统无法访问外部的内容。Linux Namespace 在此基础上， 提供了对 UTS、IPC、mount、PID、User、Network 的隔离机制。

Linux Namespace 是 Linux 提供的一种内核级别的环境隔离方法，它可以对一类资源进行抽象，并将其封装在一起提供给一个容器中使用，对于这类资源，每个容器都有自己的抽象，而它们彼此之间是不可见的，这样就可以做到访问隔离。

Linux Namespace 种类

Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

mount Namespace

mount Namespace 的作用是隔离挂载点（mount point），一个 mount Namespace 里的进程只能看到看到自己的目录结构，并且把它的挂载点当作根目录，从而看不到它的挂载点之外的目录结构。

mount Namespace 比 chroot 好的地方在于，chroot 只是将子目录设置为应用程序的根目录，在 chroot 的根目录下不会有系统根目录下的其他子目录，而 mount Namespace 则会创建这些子目录的一个实例。

例如：

在 Host OS 下可以看到有如下子目录：

```
-bash-3.2$ hostname
laoar.local
-bash-3.2$ pwd
/
-bash-3.2$ ls
Applications            bin                     private
Incompatible Software   cores                   sbin
Library                 dev                     tmp
Network                 etc                     usr
System                  home                    var
Users                   net
Volumes                 opt
-bash-3.2$
```

而在 Container 中依然可以看到它们：

```
bash-3.2$ docker run -it ubuntu
root@0d31f5e9afc5:/# hostname
0d31f5e9afc5
root@0d31f5e9afc5:/# pwd
/
root@0d31f5e9afc5:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@0d31f5e9afc5:/#
```

Network Namespace

Network Namespace 用来实现网络资源（网络设备、IP 地址、IP 路由器、端口号等）的隔离。

PID Namespace

PID Namespace 用来实现 PID 号这个资源的隔离，这样在不同的 PID Namespace 中可以使用同一个 PID，比如每一个 Namespace 都有它自己的 init 进程（PID 1），init 进程是其它进程的祖先进程。

IPC Namespace

IPC Namespace 用来隔离 System V IPC 标识以及 Posix 消息队列，这样使得不同的 Namespace 之间的进程不能直接通信，就像是在不同的系统中一样。

UTS Namespace

UTS Namespace 用于封装 `uname()` 这个系统调用。

User Namespace

User Namespace 用于隔离用户 ID 和组 ID。这样可以让一个用户在某个 Container 中拥有 root 权限但是在整个系统里却没有 root 权限。

Namespace API

- `clone()` 实现线程的系统调用，用来创建一个新的进程，同时创建 Namespace，通过 Namespace 的系统调用参数达到隔离；
- `unshare()` 在原先的进程上进行新的 Namespace 隔离；
- `setns()` 加入一个已经存在的 Namespace。

clone()

通过 `clone()` 可以创建一个新的进程，并且同时创建 Namespace。

```
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);
```

- `child_func`

  传入子进程运行的程序主函数。
- `child_stack`

  传入子进程使用的栈空间。
- `flags`

  表示使用哪些 CLONE_* 标志位，可用 `|` 连接。
- `args`

  传入用户参数。

/proc/PID/ns 文件

每一个进程有一个 `/proc/PID/ns` 目录，该目录下每一个命名空间对应一个文件。从 3.8 版本起，每一个这类文件都是一个特殊的符号链接。用户可以在此查看到指向不同 Namespace 号的文件。如果两个进程指向的 Namespace 编号相同，就说明它们在同一个 Namespace 下，否则在不同的 Namespace 里。

setns()

通过 `setns()` 系统调用，可以将进程从原先的 Namespace 中解除，并加入到已有的另一个同类型 Namespace 中。

```
int setns(int fd, int nstype);
```

- `fd` 表示要加入的 Namespace 文件的描述符。它是一个指向 `/proc/PID/ns` 目录的文件描述符。可以通过直接打开该目录下的链接或者打开一个挂载了该目录下链接的文件得到。

- `nstype` 让调用者可以检查 `fd` 指向的 Namespace 类型是否符合实际的要求，填 `0` 表示不检查。

unshare()

`unshare()` 跟 `clone()` 很像，但是 `unshare()` 运行在原先的进程上，不需要启动一个新的进程。它会创建一个新的 Namespace，并将调用者作为 Namespace 的一部分。

```
int unshare(int flags);
```

- `flags` 表示使用哪些 CLONE_* 标志位，可用 `|` 连接。

容器化遇到的问题

性能的进一步提升

为了了解 Docker 技术在性能方面的表现，我们对 Docker 的性能测试进行了以下调研：

我们主要从以下几个方面来衡量 Docker 的性能：

- 计算能力
- 内存访问效率
- 启动时间
- 网络性能
- 存储性能

在总体上，Docker 的性能与 Native 几乎相同，并强于 KVM。

计算能力

在计算能力上，单 CPU 情况下，Docker 与 Native、KVM 的性能差别很微弱，而在双 CPU 情况下，Docker 与 Native 的性能也几乎没有差距，但 KVM 的性能明显弱于前两者。

内存访问效率

大批量连续内存读写时，Docker、Native、KVM 的性能几乎相同。在随机内存读写时，Docker 与 Native 性能基本相同，KVM 明显弱于前两者。

启动时间

Docker 的启动时间在 ms 级别，而 KVM 的启动时间在 second 级别。

网络性能

Docker 和 KVM 在网络性能上稍逊于 Native，但 Docker 的性能强于 KVM。

存储性能

Docker 与 Native 在 IO 读写上没有明显性能差距，但与 KVM 相比有较大优势。

小结

Docker 在除了网络的各个方面都已经逼近原生操作系统的性能，已经没有太多的提升空间，如果要想进一步提升应用程序的性能，必须针对 Docker 对整个操作系统有所改进才能进一步提高应用程序的运行效率。

容器的安全性

为了更少的性能损失和更轻量，Docker 的隔离性和安全性显然不如传统的虚拟化技术。

我们希望的安全性一般包括：

- 容器不会对宿主机器造成影响；
- 一个容器不会对其他容器造成影响。

但是 Docker 的隔离技术其实是不完全的，存在以下问题：

- `/proc`、`/sys` 等未完全隔离；
- `top`, `free`, `iostat` 等命令展示的信息未隔离；
- `root` 用户未隔离；
- `/dev` 设备未隔离；
- 内核模块未隔离；
- SELinux、time、syslog 等所有现有 Namespace 之外的信息都未隔离

另外多个 container，system call 其实都是通过主机的内核处理，这便为Docker留下了潜在的的安全问题。

容器逃逸

因为容器和宿主机是共用内核的，所以如果宿主机的内核没升级的话，使用各种内核漏洞就可以逃逸出虚拟机：

如最有名的 Dirty COW - (CVE-2016-5195) - Docker Container Escape

信息泄漏

- Spectre/Meltdown

  前段时间披露的 Spectre/Meltdown 漏洞，由于是处理器级别的漏洞，容器中完全可以使用相关的技术来攻击。

- 网络信息泄漏

  由于容器的运行环境一般是内网，如果被入侵可能会通过内网访问到内网中机密的信息。

相关管理软件的安全问题

Docker 配套的管理工具也出现过很多问题，最著名的是：`Docker Remote API`（2375 端口）未授权漏洞。

其他还有：

- Kubernetes 未授权访问：

  API Server 默认会开启两个端口：8080 和 6443。

  其中 8080 端口无需认证，应该仅用于测试。6443 端口需要认证，且有 TLS 保护。

- Mesos 未授权访问

`Mesos master` 默认监听 `5050` 端口。

- DCOS 未授权访问

  `API Router` 的默认端口是 `80`（HTTP）和`443`（HTTPS）。

前瞻性及重要性

该项目通过对操作系统及相关软件的修改与集成，实现 Docker 在物理机直接装载运行，从而可以在依托 Docker 原有的成熟丰富的生态环境、保持用户接口不变、不借助其他虚拟化技术的前提下，获得更大的计算性能、存取性能和网络性能优化空间、简化部署流程、降低运维成本、获得更高的安全性与隔离性，应对复杂多变的业务情景，更易实现弹性伸缩的目标，并尝试支持多种处理器架构。

保留容器化技术的优势

该项目完全保留了原有容器化技术的特点，依托 Docker 原有的成熟丰富的生态环境、保持用户接口不变，学习成本低，原有的 Docker 上的解决方案可以继续使用。

取得更大的优化空间

在我们调研的相关工作中可以发现 Unikernel 等高度专能化系统中应用程序往往能发挥出更强大的性能，该项目通过操作系统级别的修改和相关软件的集成，可以在计算性能、存取性能和网络性能拥有更大的优化空间。

降低运维成本

使用物理机器直接装载并运行容器，可以兼容目前现有的云计算管理体系，可以很方便的做到分发启动，配合相关的管理软件，可以大幅度降低运维成本。

更高的安全性和隔离性

以物理机器作为容器中不同应用程序的隔离，相比原有的体系，安全性和隔离性都更高。

多架构支持

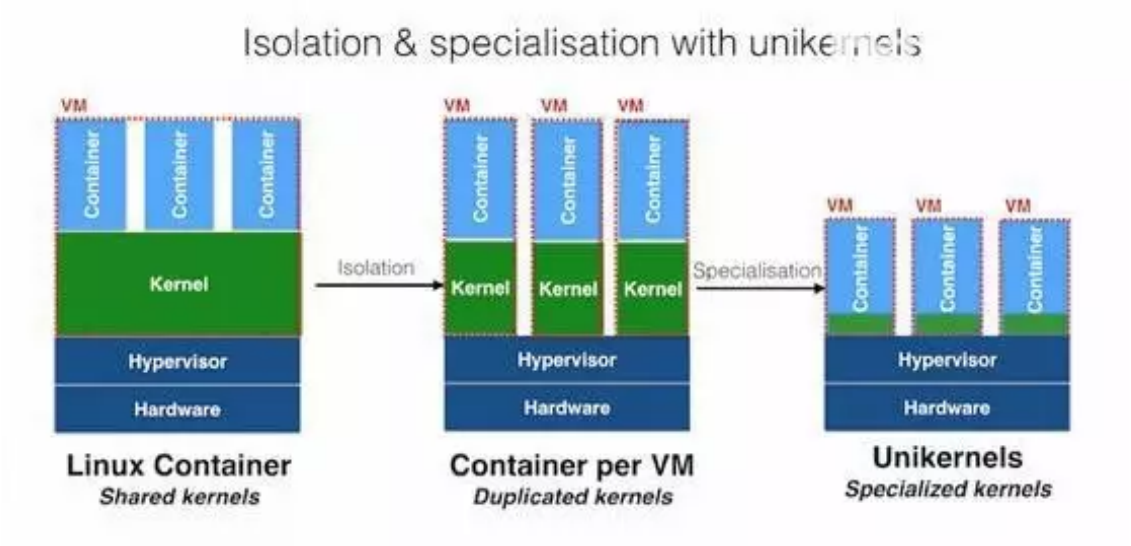
本项目的另一个目的是针对多架构进行支持，希望我们的技术能够在多种处理器架构的机器上直接运行，进一步发挥容器化技术的优势。

相关工作

Unikernel

Unikernel 是精简专属的库操作系统（LibraryOS），它具有能够使用**高级语言**编译并直接运行在商用云平台虚拟机管理程序之上的能力。其代码量和复杂度均远小于传统操作系统。Unikernel 试图抹去现代操作系统带来的一些复杂性，删除应用与硬件中间臃肿的部分，让最“精简”的操作系统运行用户的代码。无核化 Unikernel 允许将所有应用程序包括操作系统都考虑到编译和打包中。例如，如果应用不需要持久磁盘的访问，那么设备驱动程序和 OS 有关磁盘使用的工具就可以不包含在生产 Image 中。因为 Unikernels 是为运行 Xen 等虚拟管理程序而设计的，它们只需要网络和磁盘的一些标准资源的接口，用于显示的数千个设备驱动、磁盘驱动等等都是不需要的。由此，VM image 文件变得更

小，安全性变得更高，部署变得更快，更加易于维护。



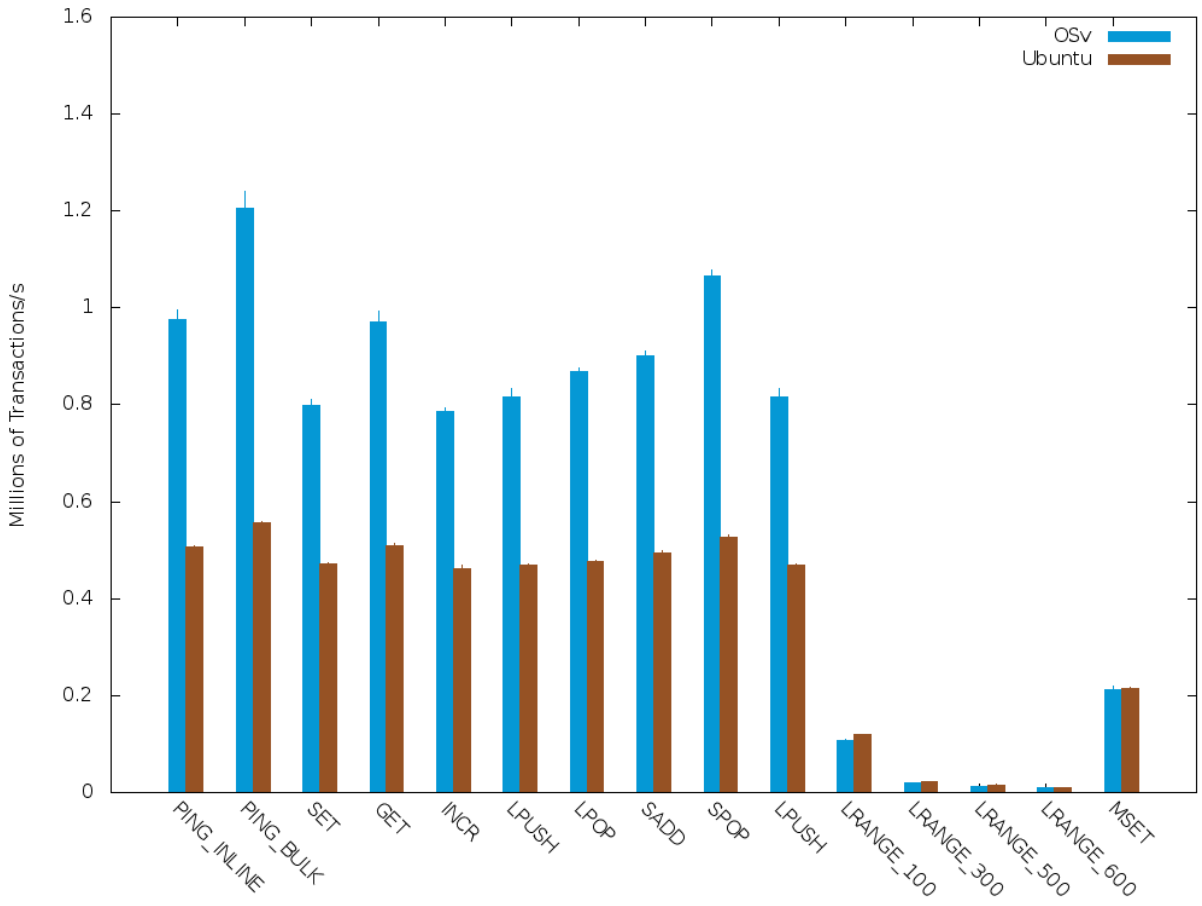
Unikernel 的优点

充分发挥现代虚拟机技术的优势

比传统操作系统更高的运行性能

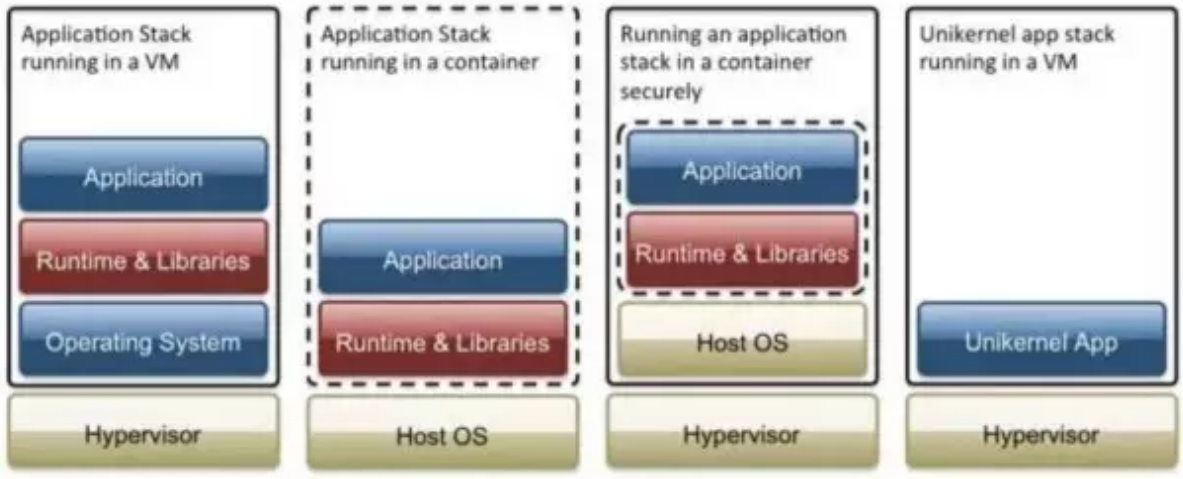
Unikernel 消除了运行时内核态与用户态切换的过程。Unikernel 的核心驱动与应用程序是同时打包构建的，其内存是单地址空间（single address space），不区分系统内核区域和应用服务区域，因此在程序执行效率上远高于通用的服务器操作系统。

下图是在相同的硬件和配置下，在 OSv 与 Ubuntu 上运行 Redis 数据库服务的性能对比：



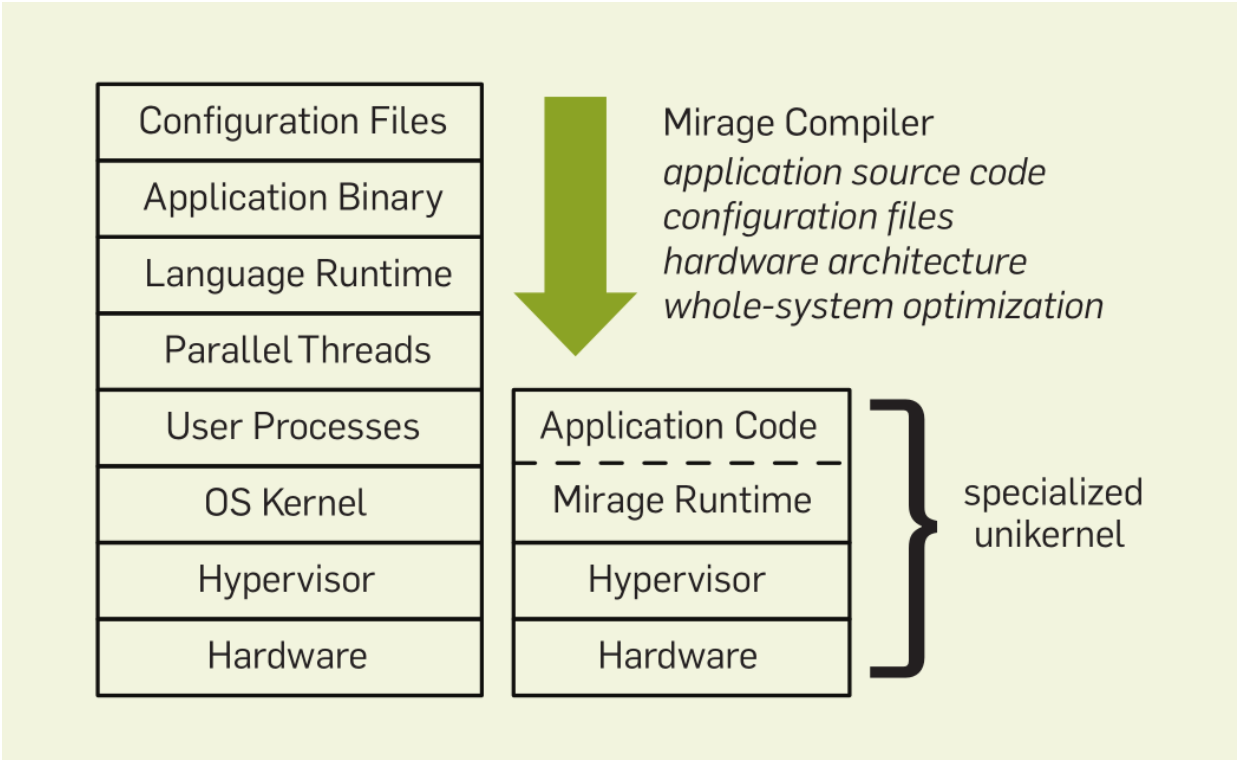
运行微服务时非常高的资源利用效率

传统的操作系统体积是非常大的，无论是 RedHat，SUSE 还是 Ubuntu 亦或是 Windows，他们的思路都是将操作系统做大做全，所以包含了很多对于特定应用而言不必要的服务、驱动、依赖包等等，而且在启动时，就加载了尽可能全的库，这样就浪费了很多硬盘空间、内存空间、CPU运行时间。即使通过一些手段和技术让传统操作系统瘦了下来（例如裁剪内核），仍然会有几十兆到几百兆的大小，这巨大的体积与云市场的“微服务”发展趋势是相悖的。如果使用 Unikernel 技术，可以让每个服务的体积变得非常小，比如 MirageOS 的示例 mirage-skeleton 编译出来的 Xen 虚拟机只有 2M。这样一来，即使每个 Unikernels 只运行一个微服务，资源利用效率依然很高。



隔离性和安全性非常好

Docker 里面虽然大多数情况只跑一个程序，但是里面含有其它 Coreutils，只要其中任何一个有安全漏洞，整个服务可能就会受到攻击（从 Bash 到 Python 脚本，一些安全漏洞总会能被启动执行）。而 Unikernel 只运行操作系统的核心，甚至抛掉了那些可能是漏洞来源的视频、USB 驱动、系统进程等模块，极大的减小了攻击面。并且，对于 Unikernel 而言，每一个操作系统都是定制用途的，其中包含的核心库均不相同，即使某个服务构建的操作系统由于特定软件问题而遭到入侵，同样的入侵手段在其他的服务中往往不能生效。这无形中增加了攻击者利用系统漏洞的成本。



可伸缩性很强

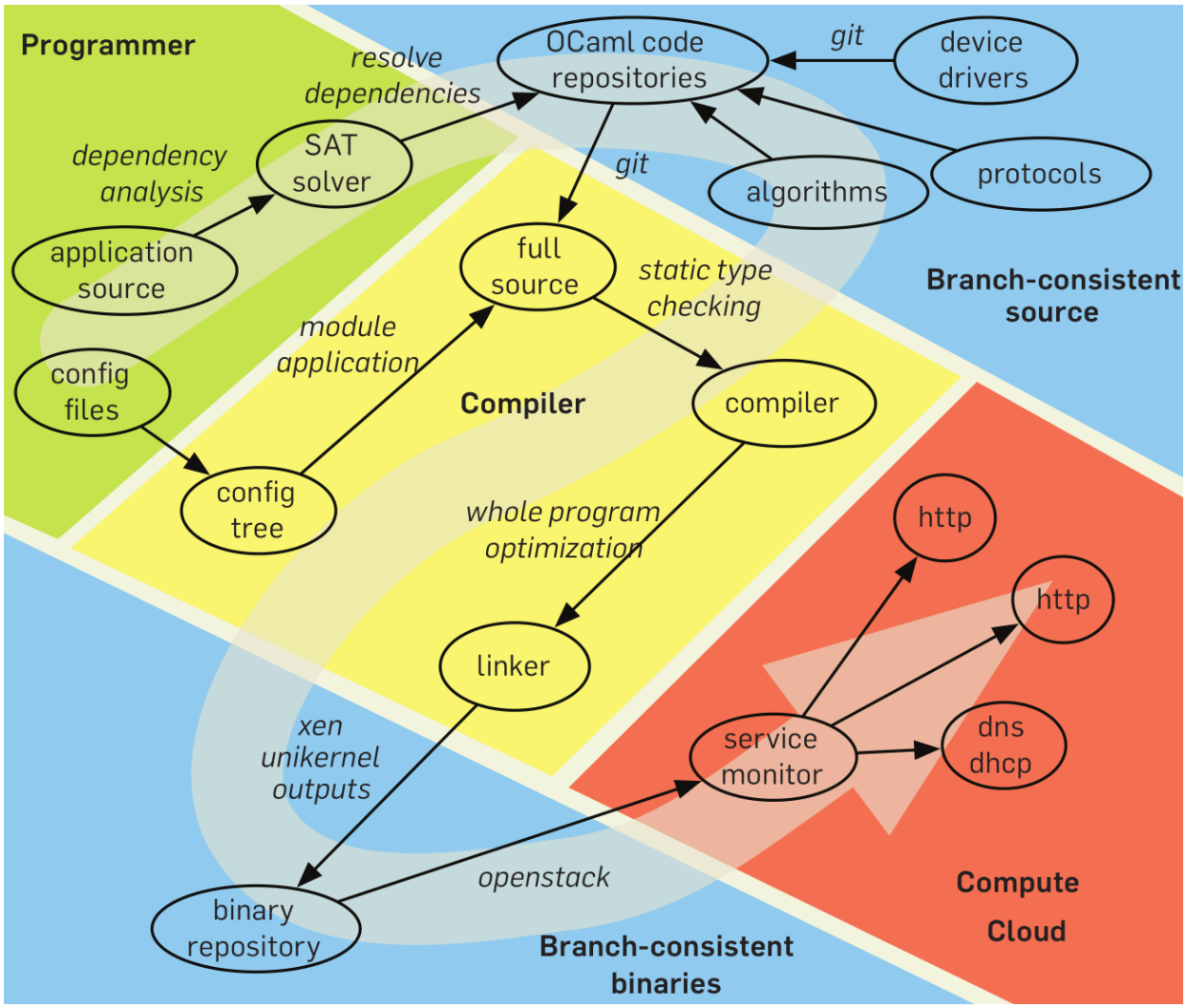
扩展的一个方法就是分配更多的内存和核数，这在现代虚拟机技术下是很容易实现的。另一个方法是运行更多的实例，而 Unikernel 启动时间极快（20ms 以内），这意味着 Unikernel 可以在负载增大时快速开启更多实例来均摊负载。在负载很低时，甚至可以在用户请求时才启动并对用户请求做出响应。

可迁移性很强

现代虚拟机技术支持热迁移。由于 Unikernel 镜像通常很小，迁移速度很快，不会带来网络压力。这就让运行实例的实时迁移成为了可能。

有效降低应用开发工作的复杂度

应用开发者可以完全使用高级语言进行开发工作，无需研究底层代码，也不用研究硬件协议。



有效降低运维工作的复杂度

Unikernel 的镜像体积大概只有传统内核镜像的 4%，而且 Unikernel 系统一旦编译完成就不可改变。这种软件实施方式有点像使用一次性产品，即安装一次，不做修改，用过即扔。这种方式可以保障软件部署的一致性，提高运维效率和降低管理的复杂性。

Unikernel 的缺点

当然，Unikernel 也存在一些问题，其中最大的问题在于 Unikernel 是完全不可调试的。在 Unikernel 这种模式下，启动之后直接进入到应用程序中，出问题之后唯一的解法就是重启，想要对其中的内容进行重新定制的唯一办法就是修改源码然后重新编译。

另外，有安全专家认为运行内容少不能证明更安全，虽然 Unikernel 减少了系统攻击面，但是它也缺失了通用操作系统内建的大量安全机制，包括特权隔离、保护 Ring、强制访问控制、审计等。

Unikernel 相比 Traditional OS

- Unikernel 和操作系统都可以运行于 Bare Metal Architecture；
- Unikernel 代码量和复杂度小于操作系统；

- Unikernel 可以便捷的锁核，特定 CPU 服务于特定应用，减少线程切换开销；
- Unikernel 可自定义模块，选择性抛弃应用不需要的内核模块，例如软驱，USB 驱动等；
- Unikernel 一旦打包完成，内核模块的独立升级比较困难。

Unikernel 相比 KVM

- Unikernel 可以运行于 Hypervisor 或是 Bare Metal Architecture，KVM 运行于Hosted Architecture；
- Unikernel 可以选择性的打包内核模块，KVM 虚拟出完整的操作系统；
- Unikernel 可以在一台宿主机上运行上千个应用，而这是虚拟机所无法办到的；
- Unikernel 与 KVM 具有类似级别的计算隔离性。

Unikernel 相比 Docker

- 运行于 Hosted Architecture，Unikernel 相比于 Docker 具有更好的隔离性、安全性；
- 运行于 Bare Metal Architecture，Unikernel 相比于 Docker 具有更快的启动时间，但目前缺少编排管理系统的支持；
- Unikernel 可满足定制化内核的需求。

Unikernel 的运行环境

Unikernel 与嵌入式操作系统非常相似，构建的镜像可以运行在虚拟化和物理环境（甚至可以运行在 FPGA 上）中。不过由于 Unikernel 在编译时需要指定特定的驱动程序，所以制作的镜像在物理环境中不具备通用性。虚拟化技术向上层提供了统一的硬件层，屏蔽了底层物理硬件的差异，Unikernel 主要还是应用在虚拟化环境中。

CoreOS

随着近年来容器技术的兴起，与之配套的技术也发展迅猛，CoreOS 正是以“容器化”为背景而诞生的一个操作系统。

CoreOS 面向云，是一款轻量级的操作系统，基于裁剪过的 Linux 内核，最大的特定是可以很方便的组建大型云服务集群，并且拥有优秀的动态缩放和管理集群的特点。

CoreOS 的设计目标是安全性、一致性、可靠性，这些设计目标都体现在了 CoreOS 的一系列特点中：

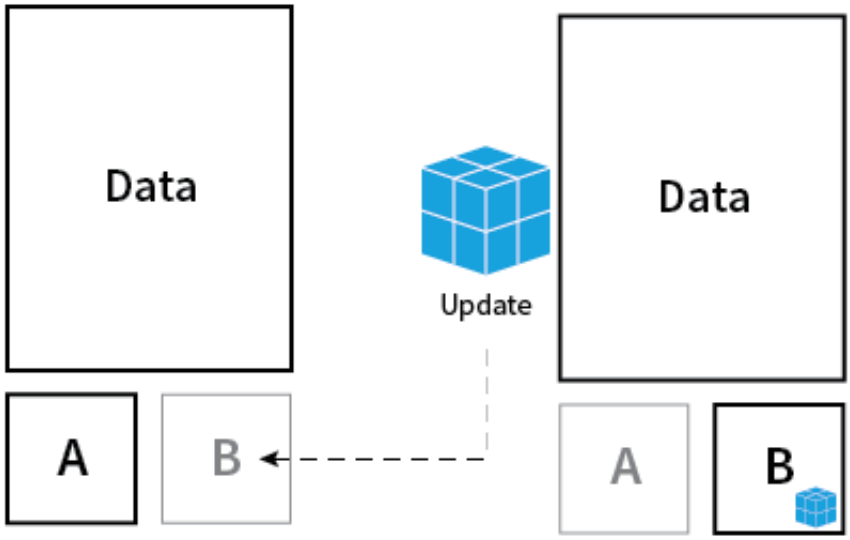
更新机制

云服务中一个重要的话题就是更新，一个大型的云服务集群常常面临着需要频繁更新的问题。

而 CoreOS 的更新机制也正是它相比于传统云服务操作系统最大的优势之一。

为了保持一致性和可靠性，CoreOS 没有采用和其他发行版类似的基于包的管理，如操作系统级别的更新，是以整个操作系统作为一个整体，作为一个最小单位进行更新。

这样的更新是通过一个叫做“主动/被动分区”的技术完成的，简单来说就是将更新由被动分区 B 完全准备好以后，重新启动之后 B 分区就会变成主动引导分区，并引导系统启动。



这样的更新机制好处是很多的：

- 安全回滚

  如果升级失败，系统可以很方便的从原被动分区回滚到原主动分区。

- 签名验证

  由于以整个分区为粒度进行更新，验证程序的完整性和可靠性可以很方便的通过签名来完成。

- 快速执行

  CoreOS 的更新配合它的轻量级可以实现快速重启，通常只需要数秒钟的时间，这一点很好地保证了应用中断的时间不会很长。另外如果云平台支持 kexec，则可以进一步缩短重启时间。

更新策略

CoreOS 预设了四种更新策略以适合不同需求，这里列举如下，有待之后进一步调研：

- Best-effort
- Etcd-lock
- Reboot
- Off

运行环境

CoreOS 可以运行在 Vagrant, Amazon EC2, QEMU/KVM, VMware, OpenStack 等主流的云平台，也可以运行在裸机。

由于 CoreOS 内核面向容器技术且经过裁剪，比起一般的系统运行需要的硬件资源可以更少。

集群发现

CoreOS 使用 etcd 专门处理集群问题，etcd 运行在集群的每一个机器上，并且自动协调，许多 CoreOS 的机器组成一个集群，他们的 etcd 会互相连接。

集群调度

fleet 在 CoreOS 集群中的地位就如同 init 对于一个单一机器的地位，他可以与每一个节点的 systemd init 系统交互，可以在一台中心主机上启动和管理调度每一个节点。

生态支持

CoreOS 经过了 Kubernetes 认证，可以很方便的和 Kubernetes 一起工作。