软件复用讨论课03 —— 复用云技术

王思尧 1352896

容器技术

理解

> Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口(类似 iPhone 的 app)。几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。

Docker近两年成了容器技术的代言，从docker的图标中可以看出docker容器技术与集装箱概念类似，首先了解一下集装箱的优势以及其带来的变革

> 集装箱最大的成功在于其产品的标准化以及由此建立的一整套运输体系。无论货物的体积、形状差异有多大，最终都被装载进集装箱里。由于要实现标准尺寸集装箱的运输，堆场、码头、起吊、船舶、汽车乃至公路桥梁、隧道等，都必须适应它在全球范围内的应用而逐渐加以标准化，形成影响国际贸易的全球物流系统。由此带来的是系统效率大幅度提升，运输费大幅度下降，地球上任何一个地方生产的产品都可以快速而低廉地运送到有需求的地方。

从以上介绍可以看出集装箱的本质是通过标准化从而带来效率的提升，Docker

容器技术与集装箱概念十分相似，以下表格类比了容器技术与集装箱

集装箱	容器技术
发货商	应用的发布者，现实中多为应用的生产方，即开发者
客户	使用应用的互联网用户
货物	构成应用的代码、组件、依赖等
集装箱	Docker容器
装卸货	应用的发布、撤销
码头工人	实际操作应用发布过程的人，现实中多为运维人员
散件装卸、运输方式	应用发布过程中逐个安装部署代码、组件、依赖、配置环境等
集装箱装卸、运输方式	把应用运行所需的外部环境、内部代码、组件、依赖打包放进容器，应用发布以容器为单位
港口的码头、起重机、集装箱堆场	应用发布所需的基础设施与工具
轮船/轮船公司	容器运行平台，如可以运行容器的云计算平台

以上可以看出容器技术与集装箱的相似之处，对容器技术也有了一个更为直观的理解，Docker容器技术现在已经变得十分火热，容器技术正在快速改变着公司和用户创建，发布，运行分布式应用的方式，以下是其带来的优点

> - 资源独立、隔离
> **资源隔离是云计算平台的最基本需求。Docker通过linux namespace, cgroup限制了硬件资源与软件运行环境，与宿主机上的其他应用实现了隔离，做到了互不影响。不同应用或服务以“集装箱”（container）为单位装“船”或卸“船”，“集装箱船”（运行container的宿主机或集群 ）上，数千数万个“集装箱”排列整齐，不同公司、不同种类的“货物”（运行应用所需的程序，组件，运行环境，依赖）保持独立。**
> - 环境的一致性

> 开发工程师完成应用开发后build一个docker image，基于这个image创建的container像是一个集装箱，里面打包了各种“散件货物”（运行应用所需的程序，组件，运行环境，依赖）。无论这个集装箱在哪里：开发环境、测试环境、生产环境，都可以确保集装箱里面的“货物”种类与个数完全相同，软件包不会在测试环境缺失，环境变量不会在生产环境忘记配置，开发环境与生产环境不会因为安装了不同版本的依赖导致应用运行异常。这样的一致性得益于“发货”（build docker image）时已经密封到”集装箱“中，而每一个环节都是在运输这个完整的、不需要拆分合并的”集装箱“。
>
> - 轻量化
> 相比传统的虚拟化技术（VM），使用docker在cpu, memory, disk IO, network IO上的性能损耗都有同样水平甚至更优的表现。Container的快速创建、启动、销毁受到很多赞誉。
>
> - Build Once, Run Everywhere
> “货物”（应用）在“汽车”，“火车”，“轮船”（私有云、公有云等服务）之间迁移交换时，只需要迁移符合标准规格和装卸方式的“集装箱”（docker container），削减了耗时费力的人工“装卸”（上线、下线应用），带来的是巨大的时间人力成本节约。这使未来仅有少数几个运维人员运维超大规模装载线上应用的容器集群成本可能，如同60年代后少数几个机器操作员即可在几小时内连装带卸完一艘万级集装箱船。

下面看一下Docker容器技术的隔离性和安全性

就Docker来说，安全性可以概括为两点：

```
不会对主机造成影响
不会对其他容器造成影响
```

所以安全性问题90%以上可以归结为隔离性问题。而Docker的安全问题本质上就是容器技术的安全性问题，这包括共用内核问题以及Namespace还不够完善的限制：

```
/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
```

传统虚拟机与Docker安全性分析比较

> 其实传统虚拟机系统也绝非100%安全，只需攻破Hypervisor便足以令整个虚拟机毁于一旦，问题是有谁能随随便便就攻破吗？如上所述，Docker的隔离性主要运用Namespace 技术。传统上Linux中的PID是唯一且独立的，在正常情况下，用户不会看见重复的PID。然而在Docker采用了Namespace，从而令相同的PID可于不同的Namespace中独立存在。举个例子，A Container 之中PID=1是A程序，而B Container之中的PID=1同样可以是A程序。虽然Docker可透过Namespace的方式分隔出看似是独立的空间，然而Linux内核（Kernel）却不能Namespace，所以即使有多个Container，所有的system call其实都是通过主机的内核处理，这便为Docker留下了不可否认的安全问题。
>
> 传统的虚拟机同样地很多操作都需要通过内核处理，但这只是虚拟机的内核，并非宿主主机内核。因此万一出现问题时，最多只影响到虚拟系统本身。当然你可以说黑客可以先Hack虚拟机的内核，然后再找寻Hypervisor的漏洞同时不能被发现，之后再攻破SELinux，然后向主机内核发动攻击。文字表达起来都嫌繁复，实际执行更是如此。所以Docker是很好用，但在迁移业务系统至其上时，需要注意安全性。

解决安全性问题

在接纳了“容器并不是全封闭”这种思想以后，开源社区尤其是红帽公司，连同Docker一起改进Docker的安全性，改进项主要包括保护宿主不受容器内部运行进程的入侵、防止容器之间相互破坏。开源社区在解决Docker安全性问题上的努力包括：

```
1. Audit namespace
    作用：隔离审计功能
    未合入原因：意义不大，而且会增加audit的复杂度，难以维护。
2. Syslognamespace
    作用：隔离系统日志
    未合入原因：很难完美的区分哪些log应该属于某个container。
3. Device namespace
    作用：隔离设备（支持设备同时在多个容器中使用）
    未合入原因：几乎要修改所有驱动，改动太大。
4. Time namespace
    作用：使每个容器有自己的系统时间
    未合入原因：一些设计细节上未达成一致，而且感觉应用场景不多。
5. Task count cgroup
    作用：限制cgroup中的进程数，可以解决fork bomb的问题
    未合入原因：不太必要，增加了复杂性，kmemlimit可以实现类似的效果
。(最近可能会被合入)
6. 隔离/proc/meminfo的信息显示
    作用：在容器中看到属于自己的meminfo信息
    未合入原因：cgroupfs已经导出了所有信息，/proc展现的工作可以由用
户态实现，比如fuse。
```

不过，从08年cgroup/ns基本成型后，至今还没有新的namespace加入内核，cgroup在子系统上做了简单的补充，多数工作都是对原有subsystem的完善。内核社区对容器技术要求的隔离性，本的原则是够用就好，不能把内核搞的太复杂。

一些企业也做了很多工作，比如一些项目团队采用了层叠式的安全机制，这些可选的安全机制具体如下：

- 文件系统级防护

  文件系统只读：有些Linux系统的内核文件系统必须要mount到容器环境里，否则容器里的进程就会罢工。这给恶意进程非常大的便利，但是大部分运行在容器里的App其实并不需要向文件系统写入数据。基于这种情况，开发者可以在mount时使用只读模式。比如下面几个： /sys 、/proc/sys 、/proc/sysrq-trigger 、 /proc/irq、/proc/bus

  写入时复制（Copy-On-Write）：Docker采用的就是这样的文件系统。所有运行的容器可以先共享一个基本文件系统镜像，一旦需要向文件系统写数据，就引导它写到与该容器相关的另一个特定文件系统中。这样的机制

避免了一个容器看到另一个容器的数据，而且容器也无法通过修改文件系统的内容来影响其他容器。

- Capability机制

  Linux对Capability机制阐述的还是比较清楚的，即为了进行权限检查，传统的UNIX对进程实现了两种不同的归类，高权限进程（用户ID为0，超级用户或者root），以及低权限进程（UID不为0的）。高权限进程完全避免了各种权限检查，而低权限进程则要接受所有权限检查，会被检查如UID、GID和组清单是否有效。从2.2内核开始，Linux把原来和超级用户相关的高级权限划分成为不同的单元，称为Capability，这样就可以独立对特定的Capability进行使能或禁止。通常来讲，不合理的禁止Capability，会导致应用崩溃，因此对于Docker这样的容器，既要安全，又要保证其可用性。开发者需要从功能性、可用性以及安全性多方面综合权衡Capability的设置。目前Docker安装时默认开启的Capability列表一直是开发社区争议的焦点，作为普通开发者，可以通过命令行来改变其默认设置。

- NameSpace机制

  Docker提供的一些命名空间也从某种程度上提供了安全保护，比如PID命名空间，它会将全部未运行在开发者当前容器里的进程隐藏。如果恶意程序看都看不见这些进程，攻击起来应该也会麻烦一些。另外，如果开发者终止pid是1的进程命名空间，容器里面所有的进程就会被全部自动终止，这意味着管理员可以非常容易地关掉容器。此外还有网络命名空间，方便管理员通过路由规则和iptable来构建容器的网络环境，这样容器内部的进程就只能使用管理员许可的特定网络。如只能访问公网的、只能访问本地的和两个容器之间用于过滤内容的容器。

- Cgroups机制

  主要是针对拒绝服务攻击。恶意进程会通过占有系统全部资源来进行系统攻击。Cgroups机制可以避免这种情况的发生，如CPU的cgroups可以在一个Docker容器试图破坏CPU的时候登录并制止恶意进程。管理员需要设计更多的cgroups，用于控制那些打开过多文件或者过多子进程等资源的进程。

- SELinux

  SELinux是一个标签系统，进程有标签，每个文件、目录、系统对象都有标签。SELinux通过撰写标签进程和标签对象之间访问规则来进行安全保

护。它实现的是一种叫做MAC（Mandatory Access Control）的系统，即对象的所有者不能控制别人访问对象。

容器技术的限制

- 不能应用在所有场景当中

  虽然容器技术拥有很强的兼容性，但是仍然不能完全取代现有的虚拟机环境。就像虚拟化技术刚刚出现的时候，一些传统的应用程序更加适合运行在物理环境当中一样，现在，一些应用程序并不适合运行在容器虚拟化环境当中。 比如，容器技术非常适合用于开发微服务类型的应用程序——这种方式将复杂的应用程序拆分为基本的组成单元，每个组成单元部署在独立的容器当中，之后将相关容器链接在一起，形成统一的应用程序。可以通过增加新的组成单元容器的方式对应用程序进行扩展，而不再需要对整个应用程序进行重新开发。

- 难以解决依赖关系问题

  大多数虚拟机都是相对独立的，每台虚拟机都包含自己的操作系统、驱动和应用程序组件。只要拥有合适的hypervisor，还可以将虚拟机迁移到其他任何虚拟化平台当中。但是对比来说，容器运行在物理操作系统之上，相互之间共享大量底层的操作系统内核、库文件以及二进制文件。Bittman进一步解释说容器之间的现有依赖关系可能会限其在服务器之间的可移植性。比如，位于Linux操作系统上的Docker容器就不能运行在当前版本的Windows Server操作系统上。

- 较差的隔离性

  基于hypervisor的虚拟机拥有完善的隔离特性，由于系统硬件资源完全是虚拟的，由hypervisor分配给虚拟机使用，因此bug、病毒或者入侵有可能影响一台虚拟机，但是不会蔓延到其他虚拟机上。 容器的隔离性较差因为其共享同一个操作系统内核以及其他组件，在开始运行之前就已经获得了统一的底层授权（对于Linux环境来说通常是root权限）。因此，漏洞和攻击更加有可能进入到底层的操作系统，或者转移到其他容器当中——潜在的传播行为远比最初的事件更加严重。

- 潜在的蔓延问题

  就像虚拟机生命周期管理对于hypervisor环境来说十分重要一样，生命周期管理对于容器来说也是至关重要的。容器可以被大量快速复制，这是容器

技术的重要优势之一，但是也有可能在管理员没有注意到的情况下消耗大量计算资源。如果应用程序所在的容器不再使用时能够被及时删除，那么情况还不算太坏。但是如果对一个容器化应用程序进行扩展之后忘记将其缩减回之前的规模，那么将会为企业带来大量的（并且不必要的）云计算开销。Bittman还表示云提供商十分高兴看到这种情况发生——因为他们就是通过出租计算资源而获利的——因此用户需要自己关注容器的部署情况。

- 缺乏工具

  对于这个行业来说，用于监控和管理容器的工具种类仍然十分缺乏。这并不是一种最近产生的现象，在基于hypervisor虚拟化的早期也曾经出现过可用工具十分匮乏的情况。就像优秀的虚拟机监控和管理工具逐渐增多一样，容器管理领域也在不断出现新的工具。其中包括谷歌的开源Docker管理工具Kubernetes，此外DockerUI使用基于web的前端界面替换Linux的命令行功能，Logspout能够将容器日志汇集到一个集中位置。

Docker容器应用场景

- 作为云主机使用

  相比虚拟机来说，容器使用的是一系列非常轻量级的虚拟化技术，使得其启动、部署、升级跟管理进程一样迅速，用起来灵活又感觉跟虚拟机一样没什么区别，所以有些人直接使用Docker的Ubuntu等镜像创建容器，当作轻量的虚拟机来使用。

  容器云主机也完全能像普通主机一样随意启动、稳定运行、关机、重启，所以在上面随意搭建博客、小网站等完全不在话下。除了常用的托管服务业务，你完全可以自定义任何用法，包括在上面使用任何云服务提供商的云硬盘、云数据库，部署各种你需要的服务。

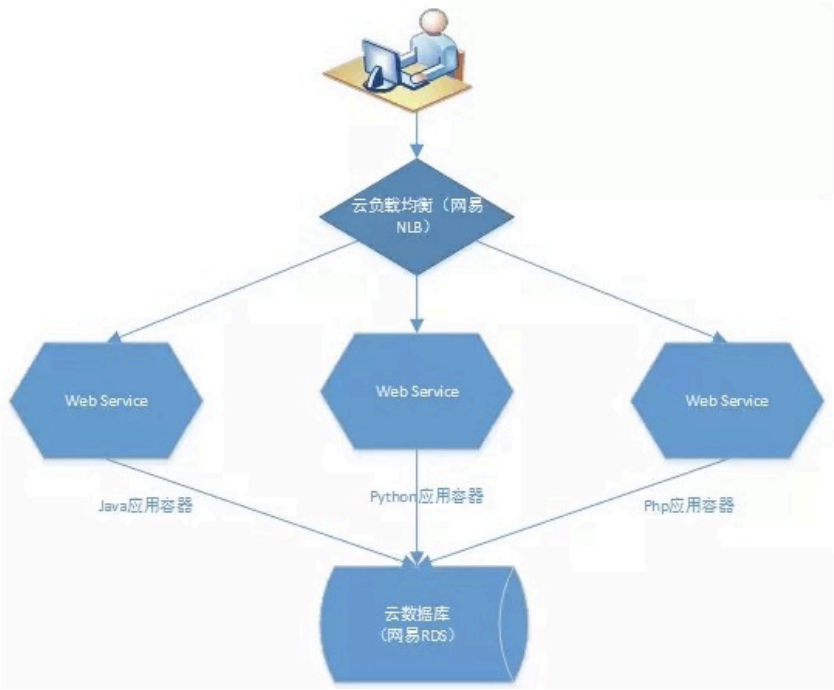
- 作为服务使用

  Docker容器最重要价值在于提供一整套平台无关的标准化技术，简化服务的部署、升级、维护，只要把需要运维的各种服务打包成标准的集装箱，就可以在任何能运行Docker的环境下跑起来，达到开箱即用的效果，这个特点才是Docker容器风靡全球的根本原因。

  - Web应用服务

Web应用服务是使用最广泛的一类服务，典型的架构是前端一个Tomcat + Java服务，后端MySQl数据库。

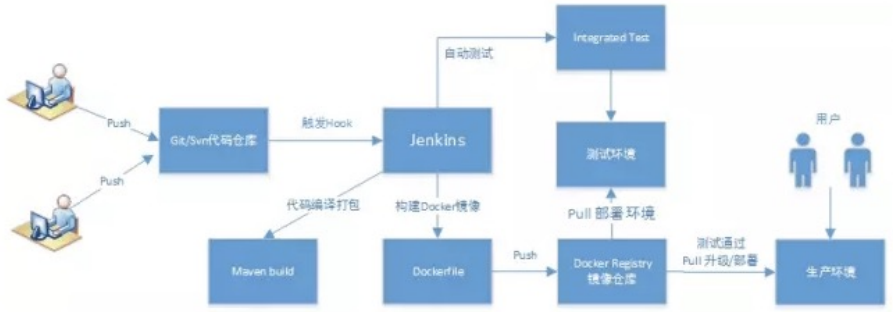
前端的Java Web服务器是最适合使用Docker容器的，先将Java运行环境、web服务器直接打包成一个通用的基础Docker镜像，之后再将自定义应用代码或编译程序包加入到该基础镜像中就能产生一个新的应用镜像，最后通过Docker服务立马就能以容器的形式启动Web应用服务。



- 持续集成和持续部署

  互联网行业倡导敏捷开发，持续集成部署CI/CD便是最典型的开发模式。

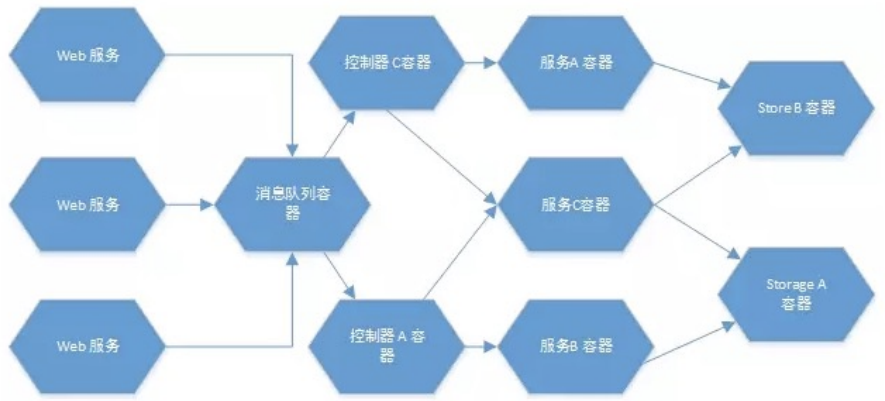
  使用Docker容器云平台，就能实现从代码编写完成推送到git/svn后，自动触发后端CaaS平台将代码下载、编译并构建成测试Docker镜像，再替换测试环境容器服务，自动在Jenkins或者Hudson中运行单元/集成测试，最后测试通过后，马上就能自动将新版本镜像更新到线上，完成服务升级。整个过程全自动化，一气呵成，最大程度地简化了运维成本，而且保证线上、线下环境完全一致，线上服务版本与git/svn发布分支也实现统一。



- 微服务架构使用

  微服务架构将传统分布式服务继续拆分解耦，形成一些更小服务模块，服务模块之间独立部署升级，这些特性与容器的轻量、高效部署不谋而合。

  如下图所示，每个容器里可以使用完全不同环境的镜像服务，容器启动即产生了一个独立的微服务主机节点（独立的网络IP），上层服务与下层服务之间服务发现通过环境变量注入、配置文件挂载等多种方式灵活解决，而且还可以直接将云平台提供的各种云服务与自定义的微服务整合组成一个强大的服务集群。



- 其他应用场景

  - 跑一次性/定时任务

    有些情况下用户只是需要执行一次性的任务，例如计算出某个结果即可，如果采用传统的服务模式，服务器需要一直运行，造成的极大的资源浪费。而容器的快速创建、销毁能很灵活满足这种完全按需付费的场景。只要制作好一次性程序运行的Docker镜像，当需要运行的时候实时通过镜像创建出容器来执行任务，程

序执行完成容器自动退出释放资源。

- 科学计算服务
- 游戏和网联网等

> http://blog.csdn.net/gaoyingju/article/details/49616295
> http://www.freebuf.com/articles/system/69809.html
> http://www.searchvirtual.com.cn/showcontent_89711.htm
> http://dockone.io/article/1282