101. 容器入门指南 - 课件

Docker 简介

Docker 的诞生

2013 年 3 月 15 日，在加利福尼亚州圣克拉拉市举办的 Python 开发者大会上，dotCloud 公司的创始人兼 CEO Solomon HyKes 在一个微型演讲中向世人宣布了 Docker，事前没有一点风声。宣布 Docker 时，在 dotCloud 公司之外只有 40 人有机会使用 Docker。

此次宣布的几周之内，Docker 得到了广泛报道。很快，这个项目开源了，源码托管在 Github 中，任何人都可以下载或为此项目做贡献。在接下来的几个月里，业界听说 Docker 的人越来越多，业内人士认为 Docker 为软件的构建、交付和运行提供了革命性的新方式。一年之内，业界几乎没有不知道 Docker 的人，但是很多人都不十分确定 Docker 是什么，也不知道人们为什么如此兴奋。

Docker 致力于轻易封装三个过程：
为任何应用创建便于分发的构建镜像
把构建镜像大规模部署到任何环境中
简化敏捷软件开发团队的工作流程，提升软件的交付效率

Docker 带来的希望

从表面看，Docker 是一种虚拟化技术，不过它并非这么简单。Docker 涉及的领域涵盖软件行业的很多方面，包括 KVM、Xen、OpenStack、Mesos、Capistrano、Fabric、Ansible、Chef、Puppet 和 SaltStack 等技术。如果和这些领域中的领先技术按照功能比较，Docker 最多算是一般的竞争对手，但是 Docker 提供的是这些软件的功能集合，为的是解决工作流程方面众多棘手问题。它不仅可以像 Fabric 等工具那样简单地部署应用，也能轻易的管理虚拟化系统，而且还可以通过使用众多的钩子轻易编排工作流程，实现自动化运维和部署。

传统应用开发模式，团队成员之间很难沟通，处理问题的方式也难统一，而且这两方面通常也很浪费时间，即便对小团队来说，也是如此。可是，在我们生活的世界中，越来越需要团队成员之间有效沟通。因此，如果有工具能降低沟通的复杂度，而且能辅助制作更强健的软件，那么此工具无疑会取得巨大的成功。如今是快速消费时代，软件的推出速度很难符合人们的需求，而且，当公司的开发者由一两个人变成多个开发团队之后，推出新版本时的沟通负担会变得更重，更难以管理。开发者要了解关于软件所处环境的大量复杂知识，而且生产环境的运维团队要不断了解所推出软件的内部细节。一般来说，掌握这些技能当然有益处，因为更好的理解整体环境有利于设计更强健的软件，可是随着团队的壮大，要掌握的技能也越来越多。如果开发者可以直接升级所用软件版本库，然后编写代码，测试新版本，最后推出新版本，整个交付过程所需的时间会大大缩减。如果运维人员不用和应用的多个开发团队协调，直接在宿主系统中升级软件，运维团队的工作效率会更高效。

Docker 能帮助公司把不同团队的业务隔离开，降低开发与运维人员的沟通成本，除了能解决沟通成本问题之外，Docker 还鼓励使用独特的微服务方式处理软件的架构，使用微服务方式能构建更强健的软件。微服务架构的核心应用服务的容器化，采用这种方式，应用不会不小心依赖之前版本遗留的构建问题。事实证明，微服务架构应对应用的弹性伸缩能力非常好，即使不使用 Docker 部署，这种架构也是很成功的。

Docker 式工作流程的好处

Docker 对组织、团队、开发者和运维工程师都有很多好处。Docker 让架构决策变得更简单了，因为从宿主系统的视角来看，本质上所有的应用都一样。使用 Docker 后，工具更易于开发，而且便于在多个应用中共用。除了前文所述，Docker 还有以下好处：

使用开发者已经掌握的技能打包软件
很多公司不得不设立发布和构建工程师职位，让他们负责管理相关的知识和工具，为软件支持的所有平台构建软件包。 RPM、mock、dpkg 和 pbuilder 等工具很复杂，而且要单独学习每个工具的使用方法。而 Docker 打包一切，在一个文件中定义所有需求。

使用标准的镜像格式打包软件应用和所需的操作系统文件系统
过去，往往不仅要打包应用，还要打包应用所需的很多依赖，例如库和守护进程。可是，我们无法确保执行应用的环境始终一致。因为，打包方式很难掌握，而且很多公司不能有效完成打包操作。Docker 的分层镜像能让整个过程变得更行之有效，确保应用运行在预期的环境中。

在运行任何系统的任何环境中测试和部署同一个打包好的构建镜像
开发者把修改后的源码提交到版本控制系统之后，可以构建新的 Docker 镜像，然后使用新的镜像测试，或者把镜像部署到生产环境，整个过程中的任何一步都无需重新编译或重新打包应用。

把软件应用与硬件剥离，不滥用资源
需要在物理硬件和运行其上的软件应用之间建立抽象层时，人们往往会使用企业级虚拟化方案，例如 VMWARE 这么做很消耗资源。虚拟机的监控程序和虚拟机中运行的每个内核都要消耗一定量的硬件系统资源，而且运行其中的应用无法使用这些被虚拟机占用的资源。而容器只是普通的进程，直接与 Linux 内核通信，因此可使用的资源更多，直到系统资源耗尽或配额用完为止。

Docker 发布之前，Linux 容器已经存在很多年了，而且 Docker 使用的很多其他技术也不是全新的。由于 Docker 采用独特的架构方式和工作流程把这些技术融合在一起之后，要比各个技术相加在一起强大得多。Docker 出现之后，终于让已经存在超过十年的 Linux 容器走进了普通技术人员的生活中。Docker 让容器能轻易集成到公司当前使用的工作流程和作业方式中。前面讨论的问题，很多人都感同身受，因此对 Docker 项目感兴趣的人才越来越多。

Docker 不是什么
Docker 能解决众多问题，这些问题是其他种类的传统工具一直致力解决的。可是，广度宽意味着深度浅。例如，有些组织认为使用 Docker 后完全可以摒弃配置管理工具。其实不然，Docker 真正强大的地方在于，虽然可以替代某些更为传统的工具，但是通常与这些工具是兼容的，有时与传统的工具结合在一起使用甚至能增强自身的功能。下面列出 Docker 无法完全取代的工具种类，如果与这些工具结合使用，则能得到更好的效果。

企业级虚拟化平台(VMWARE 或 KVM)
从传统的观点看，容器不是虚拟机。虚拟机包含完整的操作系统，运行在宿主机操作系统之上。虚拟机最大的优点是，在一台宿主机中可以使用虚拟机运行多个完全不同的操作系统。而对容器来说，宿主机和容器共用同一个内核。这意味着容器使用的系统资源更少，但是必须基于相同底层的操作系统。

云平台 (OpenStack 和 CloudStack)
从表面看，采用容器的工作流程与运行云平台有很多相似之处。从传统意义上看，二者都可以按需横向扩展。然而，Docker 不是云平台。
Docker 只能在已经安装 Docker 的宿主机中部署、运行和管理容器，不能创建新的宿主系统、对象存储器和块存储器，以及通常与云平台有关的很多其他资源。

配置管理工具 (Ansible)
虽然 Docker 能大大提升组织管理应用和应用依赖的能力，但不能完全取代更为传统的配置管理工具。Dockerfile 文件用于定义构建时容器里的内容，但是不能持续管理容器的状态，也不能用于管理 Docker 宿主系统。

部署框架(Fabric)
Docker 用于创建自成一体的容器镜像，封装应用的所有依赖，无需改动就能把应用部署到任何环境中，从而简化部署过程中的某些步骤。可是，只使用 Docker 无法自动执行复杂的部署过程，通过还需要使用其他工具把大型的自动化工作流程组织在一起。

工作负载集群管理工具(Kubernetes)
Docker 容器对集群一无所知，因此，必须使用其他编排工具，如 Kubernetes 来管理多 Docker 宿主机，跟踪所有宿主机的当前状态和资源使用情况，确保运行着足够的容器。

开发环境(vagrant)
Vagrant 是虚拟机管理工具，开发者通常用它模拟与应用的生产环境尽量一致的服务器软件环境。

- 镜像 vs 容器

容器基于镜像创建，镜像提供了部署和运行容器所需的一切基础。若想启动容器，必须下载公共镜像或创建自己的镜像。每个 Docker 镜像都包含一个或多个文件系统层，一般来说创建镜像时每个构建步骤都会创建一个文件系统层。Docker 很大程度上依赖存储后端管理镜像。存储后端负责与底层 Linux 文件系统通信，用于构建和管理镜像的多个层。

我们可以把镜像理解成容器的生成模板，同一个镜像可以创建N个容器。容器是镜像的实例，是一个真实的系统进程。

- 查找镜像 & 运行容器

可以使用如下命令镜像仓库中搜索镜像：
例如：搜索 python 镜像

docker search python

```
➜ ~ docker search python
NAME                          DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
python                        Python is an interpreted, interactive, obj...   1782      [OK]
kaggle/python                 Docker image for Python scripts run on Kaggle   55                   [OK]
google/python                 Please use gcr.io/google-appengine/python ...   34                   [OK]
tsutomu7/alpine-python        Python3.6 and scientific packages on alpine.    6                    [OK]
vimagick/python               mini python                                     3                    [OK]
pandada8/alpine-python        An alpine based python image                    3                    [OK]
tsuru/python                  Image for the Python platform in tsuru PaaS.    2                    [OK]
lucidfrontier45/python-uwsgi  Python with uWSGI                               2                    [OK]
beevelop/nodejs-python        Node.js with Python                             2                    [OK]
1science/python               Python Docker images based on Alpine Linux      1                    [OK]
```

下载镜像
docker pull python

```
➜ ~ docker pull python
Using default tag: latest
latest: Pulling from library/python
6d827a3ef358: Already exists
2726297beaf1: Downloading [==================================>                ] 12.96 MB/18.61 MB
7d27bd3d7fec: Downloading [=================>                                 ] 14.94 MB/42.57 MB
44ae682c18a3: Downloading [===>                                               ] 9.174 MB/129.9 MB
```

运行 python 容器
docker run -ti python bash

```
➜ ~ docker run -ti python bash
root@bfebc541a670:/# python
Python 3.6.0 (default, Feb 28 2017, 22:20:46)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

以交互模式运行 python 容器。

- 基于镜像的工作流
    - Dockerfile 及最佳实践

如果自己构建镜像，需要了解 Dockerfile 文件，这个文件用于描述创建一个镜像所需的全部步骤，通常都放在应用源码仓库的根目录里。

Dockerfile是一种被docker程序解释的脚本，此文件由一条条指令组成，每条指令对应linux下面的一条命令。Docker程序将这些Dockerfile指令翻译成真正的Linux命令，Dockerfile有自己的书写格式和支持命令，Dockerfile有自己的书写格式和支持的命令，Docker程序解决这些命令间的依赖关系，类似天Makefile。

相比镜像这种黑盒子，Dockerfile这种显而易见的脚本更容易被使用者接受，它明确的表明image是怎么构建的。有了Dockerfile，当我们需要定制自己额外的需求时，只需要在Dockerfile上添加或修改指令，重新生成image即可。

Dockerfile 是由一个个指令组成的，每个指令都对应着最终镜像的一层。每行的第一个单词就是命令，后面所有的字符串是这个命令的参数，关于 Dockerfile 支持的命令以及它们的用法，可以参考官方文档，这里不再赘述。

当运行 docker build 命令的时候，整个的构建过程是这样的：

- 读取 Dockerfile 文件发送到 docker daemon
- 读取当前目录的所有文件（context），发送到 docker daemon
- 对 Dockerfile 进行解析，处理成命令加上对应参数的结构
- 按照顺序循环遍历所有的命令，对每个命令调用对应的处理函数进行处理
- 每个命令（除了 FROM）都会在一个容器执行，执行的结果会生成一个新的镜像
- 为最后生成的镜像打上标签

Dockerfile的书写规则及指令使用方法，指令忽略大小写，建议使用大写 使用#作为注释，每一行只支持一条指令，每条指令可以携带多个参数。

Dockerfile的指令根据作用可以分为两种，构建指令和设置指令:
构建指令用于构建image，其指定的操作不会在运行image的容器上执行.
设置指令用于设置image的属性，其指定的操作将在运行image的容器中执行.

FROM指的是基础镜像-指令有两种格式：
FROM <image>
FROM <image>:<tag>

MAINTAINER (指定镜像创建者信息)
RUN (安装软件使用)
构建指令，RUN可以运行任何被基础image支持的命令，如基础image选择了ubuntu，那么软件管理部分只能使用ubuntu命令.此指令有两种格式：
RUN <command> (the command is run in a shell - `/bin/sh -c`)
RUN ["executable","param1","param2"...] (exec form)
CMD (设置container启动时执行的操作)
设置指令，用于container启动时指定的操作，该操作可以是执行自定义脚本，也可以是执行系统命令，该指令只能在文件中存在一次，如果有多个，则只执行最后一条，此指令有三种格式：
CMD ["executable","param1","param2"] (like an exec, this is the preferred form)
CMD command param1 param2 (as a shell)
当Dockerfile指定了ENTRYPOINT,那么使用下面的格式：
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

ENTRYPOINT (设置container启动时执行的操作)
设置指令，指定容器启动时执行的命令，可以多次设置，但是只有最后一个有效，此命令有两种格式：
ENTRYPOINT ["executable","param1","param2"] (like an exec, the preferred form)
ENTRYPOINT command param1 param2 (as a shell)

该指令使用分两种情况，一种是独自使用，一种和CMD配合使用。
1.当独自使用时，如果你还使用CMD命令且CMD是一个完整的可执行命令，那么CMD指令和ENTRYPOINT会互相覆盖只有最后一个CMD或ENTRYPOINT有效！

CMD echo "Hello world" #此指令将不会执行,只有ENTRYPOINT指令被执行
ENTRYPOINT ls -l

2.和CMD指令配合使用来指定ENTRYPOINT的默认参数，这时CMD指令不是一个完整的可执行命令，仅仅是参数部分，ENTRYPOINT指令只能使用JSON方式指定执行命令，而不能指定参数。

```
FROM ubuntu
CMD ["-1"]
ENTRYPOINT ["/usr/bin/ls"]
```

USER(设置container容器的用户)

设置指令，设置启动容器的用户，默认是root用户

```
ENTRYPOINT ["memcached"]
USER daemon
```

或

```
ENTRYPOINT ["memcached","-u","daemon"]
```

EXPOSE (指定容器需要映射到宿主机器的端口)

设置指令，该指令会将容器中的端口映射成宿主机器中的某个端口，当需要访问容器的时候，可以不是用容器的IP地址而是使用宿主机器的IP地址和映射后的端口。

要完成整个操作需要两个步骤：

首先在Dockerfile使用EXPOSE设置需要映射的容器端口，然后在运行容器的时候指定-p选项加上EXPOSE设置的端口，这样EXPOSE设置的端口号会被随机映射成宿主机器中的一个端口号。

格式：

EXPOSE <port> [<port>...]

ENV (用于设置环境变量)

构建指令，在image中设置一个环境变量

格式：

ENV <key> <value>

设置了后，后续的RUN命令都可以使用，container启动后，可以通过docker inspect查看这个环境变量，也可以通过docker run --env key=value设置或修改环境变量

假如安装了JAVA程序，需要设置JAVA_HOME,那么可以在Dockerfile中这样写：

ENV JAVA_HOME /path/to/java/dirent

ADD (从src复制文件到container的dest路径)

构建指令，所有拷贝到容器中的文件和文件夹权限为0755，uid和gid为0

如果是一个目录，那么会将该目录下的所有文件添加到容器中，不包括目录

如果文件是可识别的压缩格式，则docker会帮忙解压缩(注意压缩格式)

如果<src>是文件且<dest>中不使用斜杠结束，则会将<dest>视为文件，<src>的内容会写入<dest>

如果<src>是文件且<dest>中使用斜杠结束，则会将<src>文件拷贝到<dest>目录下

格式：

ADD <src> <dest>

VOLUME (指定挂载点)

设置指令，使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也可以共享给其他容器使用。

我们知道容器使用的是AUFS,这种文件系统不能持久化数据，当容器关闭后，所有的更改都会丢失，当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令

格式：

```
VOLUME ["<mountpoint>"]
```

例：

```
FROM base
VOLUME ["/tmp/data"]
```

运行通过该Dockerfile生成image的容器，/tmp/data目录中的数据在容器关闭后，里面的数据还存在

例如另一个容器也有持久化数据的需求，且想使用上面容器共享的/tmp/data目录，那么可以运行下面的命令启动一个容器

```
docker run -t -i -rm --volumes-from container1 image2 bash
```

container1是第一个容器的ID，image2是第二个容器运行image的名字

WORKDIR (切换目录)

设置指令，可以多次切换(相当于cd命令)，对RUN,CMD,ENTRYPOINT生效

格式：

```
WORKDIR /path/to/workdir
#在/p1/p2下执行 vim a.txt
WORKDIR /p1 WORKDIR p2 RUN vim a.txt
```

ONBUILD (在子镜像中执行)

格式：

```
ONBUILD <Dockerfile关键字>
```

ONBUILD指定的命令在构建镜像时并不执行，而是在它的子镜像中执行

编写 Dockerfile 的一些最佳实践

1. 使用统一的 base 镜像

有些文章讲优化镜像会提倡使用尽量小的基础镜像，比如 busybox 或者 alpine 等。我更推荐使用统一的大家比较熟悉的基础镜像，比如 ubuntu，centos 等，因为基础镜像只需要下载一次可以共享，并不会造成太多的存储空间浪费。它的好处是这些镜像的生态比较完整，方便我们安装软件，除了问题进行调试。

2. 动静分离

经常变化的内容和基本不会变化的内容要分开，把不怎么变化的内容放在下层，创建出来不同基础镜像供上层使用。比如可以创建各种语言的基础镜像，python2.7、python3.4、go1.7、java7等等，这些镜像包含了最基本的语言库，每个组可以在上面继续构建应用级别的镜像。

3. 最小原则：只安装必需的东西

很多人构建镜像的时候，都有一种冲动——把可能用到的东西都打包到镜像中。要遏制这种想法，镜像中应该只包含必需的东西，任何可以有也可以没有的东西都不要放到里面。因为镜像的扩展很容易，而且运行容器的时候也很方便地对其进行修改。这样可以保证镜像尽可能小，构建的时候尽可能快，也保证未来的更快传输、更省网络资源。

4. 一个原则：每个镜像只有一个功能

不要在容器里运行多个不同功能的进程，每个镜像中只安装一个应用的软件包和文件，需要交互的程序通过 pod（kubernetes 提供的特性） 或者容器之间的网络进行交流。这样可以保证模块化，不同的应用可以分开维护和升级，也能减小单个镜像的大小。

5. 使用更少的层

虽然看起来把不同的命令尽量分开来，写在多个命令中容易阅读和理解。但是这样会导致出现太多的镜像层，而不好管理和分析镜像，而且镜像的层是有限的。尽量把相关的内容放到同一个层，使用换行符进行分割，这样可以进一步减小镜像大小，并且方便查看镜像历史。

6. 减少每层的内容

尽管只安装必须的内容，在这个过程中也可能会产生额外的内容或者临时文件，我们要尽量让每层安装的东西保持最小。

比如使用 --no-install-recommends 参数告诉 apt-get 不要安装推荐的软件包
安装完软件包，清楚 /var/lib/apt/list/ 缓存
删除中间文件：比如下载的压缩包
删除临时文件：如果命令产生了临时文件，也要及时删除
7. 不要在 Dockerfile 中修改文件的权限

因为 docker 镜像是分层的，任何修改都会新增一个层，修改文件或者目录权限也是如此。如果修改大文件或者目录的权限，会把这些文件复制一份，这样很容易导致镜像很大。

解决方案也很简单，要么在添加到 Dockerfile 之前就把文件的权限和用户设置好，要么在容器启动脚本（entrypoint）做这些修改。

8. 利用 cache 来加快构建速度

如果 Docker 发现某个层已经存在了，它会直接使用已经存在的层，而不会重新运行一次。如果你连续运行 docker build 多次，会发现第二次运行很快就结束了。

不过从 1.10 版本开始，Content Addressable Storage 的引入导致缓存功能的实效，目前引入了 --cache-from 参数可以手动指定一个镜像来使用它的缓存。

9. 版本控制和自动构建

最好把 Dockerfile 和对应的应用代码一起放到版本控制中，然后能够自动构建镜像。这样的好处是可以追踪各个版本镜像的内容，方便了解不同镜像有什么区别，对于调试和回滚都有好处。

另外，如果运行镜像的参数或者环境变量很多，也要有对应的文档给予说明，并且文档要随着 Dockerfile 变化而更新，这样任何人都能参考着文档很容易地使用镜像，而不是下载了镜像不知道怎么用。

        - 创建自己的基础镜像

当我们需要创建自己的基础镜像时，具体过程取决于要打包的 Linux 发行版。我们下面有一些例子，鼓励大家提交自己的新的镜像。

        - 镜像交付与管理 - registry

        - 配置 harbor 镜像仓库
- Docker 存储基础
    - 了解镜像、容器和存储
- Docker 网络配置
    - 容器网络
    - 网络命令实战
    - 默认桥接网络
        - 旧的容器链接
        - 将容器端口绑定到主机
- Docker 存储练习
    - 添加容器卷
    - 查找容器卷 - Volume
    - 宿主机目录挂载到容器
    - 将共享存储卷作为容器数据卷装载
    - 创建和安装数据卷容器
    - 备份、还原或迁移数据卷
    - 移除数据卷
    - 使用共享卷的重要提示
- Docker 网络练习
    - 查看容器默认网络
    - 创建你自己的桥接网络
    - 将容器加入网络