Docker 容器简介

一、 什么是 Docker

Docker 使用 Google 公司推出的 Go 语言 进行开发实现，基于 Linux 内核的 cgroup，namespace，以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于 操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。使得 Docker 技术比虚拟机技术更为轻便、快捷。

二、 为什么要使用 Docker

作为一种新兴的虚拟化方式，Docker 跟传统的虚拟化方式相比具有众多的优势。

- **更高效的利用系统资源**

  由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

- **更快速的启动时间**

  传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。

- **一致的运行环境**

  开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 「这段代码在我机器上没问题啊」 这类问题。

- **持续交付和部署**

  对开发和运维（DevOps）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 Dockerfile 来进行镜像构建，并结合 持续集成(Continuous Integration) 系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜

像，甚至结合持续部署(Continuous Delivery/Deployment) 系统进行自动部署。而且使用 Dockerfile 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

- **更轻松的迁移**

由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

- **更轻松的维护和扩展**

Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的 官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

三、 基本概念

Docker 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:18.04 就包含了完整的一套 Ubuntu 18.04 最小系统的 root 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

分层存储

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

Docker 容器

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会混淆容器和虚拟机。

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用 数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器删除或者重新运行之后，数据却不会丢失。

Docker Registry（Docker 仓库）

镜像构建完成后，可以很容易的在当前宿主机上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签。

以 Ubuntu 镜像 为例，ubuntu 是仓库的名字，其内包含有不同的版本标签，如，16.04, 18.04。我们可以通过 ubuntu:14.04，或者 ubuntu:18.04 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 ubuntu，那将视为 ubuntu:latest。

仓库名经常以两段式路径形式出现，比如 jwilder/nginx-proxy，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

Docker Registry 公开服务

Docker Registry 公开服务是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 Docker Hub，这也是默认的 Registry，并拥有大量的高质量的官方镜像。除此以外，还有 CoreOS 的 Quay.io，CoreOS 相关的镜像存储在这里；Google 的 Google Container Registry，Kubernetes 的镜像使用的就是这个服务。

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务（Registry Mirror），这些镜像服务被称为加速器。常见的有 阿里云加速器、DaoCloud 加速器 等。使用加速器会直接从国内的地址下载 Docker Hub 的镜像，比直接从 Docker Hub 下载速度会提高很多。在 安装 Docker 一节中有详细的配置方法。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务。比如时速云镜像仓库、网易云镜像服务、DaoCloud 镜像市场、阿里云镜像库 等。

私有 Docker Registry

除了使用公开服务外，用户还可以在本地搭建私有 Docker Registry。Docker 官方提供了 Docker Registry 镜像，可以直接使用做为私有 Registry 服务。在 私有仓库 一节中，会有进一步的搭建私有 Registry 服务的讲解。

开源的 Docker Registry 镜像只提供了 Docker Registry API 的服务端实现，足以支持 docker 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。在官方的商业化版本 Docker Trusted Registry 中，提供了这些高级功能。

除了官方的 Docker Registry 外，还有第三方软件实现了 Docker Registry API，甚至提供了用户界面以及一些高级功能

四、 Docker 安装部署

Docker 安装部署见 rancher 安装备份文档《2019-rancher2.0 安装备份》

五、 镜像使用

（1）获取镜像

Docker Hub 上有大量的高质量的镜像可以用，这里我们就说一下怎么获取这些镜像。从 Docker 镜像仓库获取镜像的命令是 docker pull。其命令格式为：

```
docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]
```

具体的选项可以通过 docker pull --help 命令看到，这里我们说一下镜像名称的格式。

> 说明：Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub。
>
> 仓库名：如之前所说，这里的仓库名是两段式名称，即 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。

例如：

```
docker pull ubuntu:18.04
18.04: Pulling from library/ubuntu
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
Digest:
sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b492983ae97c3d643fbbe
Status: Downloaded newer image for ubuntu:18.04
```

上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:18.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 18.04 的镜像。

运行

有了镜像后，我们就能够以这个镜像为基础启动并运行一个容器。以上面的 ubuntu:18.04 为例，如果我们打算启动里面的 bash 并且进行交互式操作的话，可以执行下面的命令。

```
$ docker run -it --rm \
    ubuntu:18.04 \
    bash
```

docker run 就是运行容器的命令，具体格式我们会在 容器 一节进行详细讲解，我们这里简要的说明一下上面用到的参数。

- `-it`：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算

进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。

- `--rm`：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 `docker rm`。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。
- `ubuntu:18.04`：这是指用 `ubuntu:18.04` 镜像为基础来启动容器。
- bash：放在镜像名后的是**命令**，这里我们希望有个交互式 Shell，因此用的是 bash。

（2）列出镜像

要想列出已经下载下来的镜像，可以使用 docker image ls 命令。

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
redis               latest              5f515359c7f8        5 days ago          183 MB
nginx               latest              05a60462f8ba        5 days ago          181 MB
mongo               3.2                 fe9198c04d62        5 days ago          342 MB
<none>              <none>              00285df0df87        5 days ago          342 MB
ubuntu              18.04               f753707788c5        4 weeks ago         127 MB
ubuntu              latest              f753707788c5        4 weeks ago         127 MB
```

列表包含了 `仓库名、标签、镜像 ID、创建时间` 以及 所`占用的空间`。

其中仓库名、标签在之前的基础概念章节已经介绍过了。镜像 ID 则是镜像的唯一标识，一个镜像可以对应多个标签。因此，在上面的例子中，我们可以看到 `ubuntu:18.04` 和 `ubuntu:latest` 拥有相同的 ID，因为它们对应的是同一个镜像。

- **镜像体积**

  如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，`ubuntu:18.04` 镜像大小，在这里是 127 MB，但是在 Docker Hub 显示的却是 50 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 `docker image ls` 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

  另外一个需要注意的问题是，`docker image ls` 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复

用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

你可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。

```
$ docker system df

TYPE            TOTAL      ACTIVE      SIZE          RECLAIMABLE
Images          24         0           1.992GB       1.992GB (100%)
Containers      1          0           62.82MB       62.82MB (100%)
Local Volumes   9          0           652.2MB       652.2MB (100%)
Build Cache                            0B            0B
```

- **虚悬镜像**

上面的镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 <none>。：

```
<none>      <none>      00285df0df87        5 days ago          342 MB
```

这个镜像原本是有镜像名和标签的，原来为 `mongo:3.2，` 随着官方镜像维护，发布了新版本后，重新 `docker pull mongo:3.2` 时，`mongo:3.2` 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 <none>。除了 `docker pull` 可能导致这种情况，`docker build` 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 <none> 的镜像。这类无标签镜像也被称为 虚悬镜像(dangling image），可以用下面的命令专门显示这类镜像：

```
$ docker image ls -f dangling=true
REPOSITORY      TAG          IMAGE ID          CREATED          SIZE
<none>          <none>       00285df0df87      5 days ago       342 MB
```

一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除：

```
$ docker image prune
```

- **中间层镜像**

  为了加速镜像构建、重复利用资源，Docker 会利用 中间层镜像。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 `docker image ls` 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 `-a` 参数。

  ```
  $ docker image ls -a
  ```

  这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的中间层镜像也会被连带删除。

- **列出部分镜像**

  不加任何参数的情况下，`docker image ls` 会列出所有顶级镜像，但是有时候我们只希望列出部分镜像。`docker image ls` 有好几个参数可以帮助做到这个事情。根据仓库名列出镜像：

  ```
  $ docker image ls ubuntu
  REPOSITORY      TAG           IMAGE ID        CREATED         SIZE
  ubuntu          18.04         f753707788c5    4 weeks ago     127 MB
  ubuntu          latest        f753707788c5    4 weeks ago     127 MB
  ```

  列出特定的某个镜像，也就是说指定仓库名和标签：

  ```
  $ docker image ls ubuntu:18.04
  REPOSITORY      TAG       IMAGE ID        CREATED         SIZE
  ubuntu          18.04     f753707788c5     4 weeks ago      127 MB
  ```

  除此以外，`docker image ls` 还支持强大的过滤器参数 `--filter`，或者简写 `-f`。之前我们已经看到了使用过滤器来列出虚悬镜像的用法，它还有更多的用法。比如，我们希望看到在 `mongo:3.2` 之后建立的镜像，可以用下面的命令：

```
$ docker image ls -f since=mongo:3.2
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
redis               latest              5f515359c7f8        5 days ago          183 MB
nginx               latest              05a60462f8ba        5 days ago          181 MB
```

想查看某个位置之前的镜像也可以，只需要把 since 换成 before 即可。此外，如果镜像构建时，定义了 LABEL，还可以通过 LABEL 来过滤。

```
$ docker image ls -f label=com.example.version=0.1
```

- **以特定格式显示**

默认情况下，`docker image ls` 会输出一个完整的表格，但是我们并非所有时候都会需要这些内容。比如，刚才删除虚悬镜像的时候，我们需要利用 `docker image ls` 把所有的虚悬镜像的 ID 列出来，然后才可以交给 `docker image rm` 命令作为参数来删除指定的这些镜像，这个时候就用到了 -q 参数。

```
docker image ls -q
```

（3）删除本地镜像

如果要删除本地的镜像，可以使用 docker image rm 命令，其格式为：

```
$ docker image rm [选项] <镜像 1> [<镜像 2> ...]
```

用 `docker image ls` 命令来配合

像其它可以承接多个实体的命令一样，可以使用 `docker image ls -q` 来配合使用 `docker image rm`，这样可以成批的删除希望删除的镜像。我们在“镜像列表”章节介绍过很多过滤镜像列表的方式都可以拿过来使用。比如，我们需要删除所有仓库名为 `redis` 的镜像：`$ docker image rm $(docker image ls -q redis)`

或者删除所有在 mongo:3.2 之前的镜像：

```
$ docker image rm $(docker image ls -q -f before=mongo:3.2)
```

```
$ docker rmi $(docker images)
```

充分利用你的想象力和 Linux 命令行的强大，你可以完成很多非常赞的功能。

（4）利用 commit 理解镜像构成

> 注意：`docker commit` 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 `docker commit` 定制镜像，定制镜像应该使用 Dockerfile 来完成。

镜像是容器的基础，每次执行 `docker run` 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，我们所使用的都是来自于 `Docker Hub` 的镜像。直接使用这些镜像是可以满足一定的需求，而当这些镜像无法直接满足需求时，我们就需要定制这些镜像。接下来的几节就将讲解如何定制镜像。

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。现在让我们以定制一个 Web 服务器为例子，来讲解镜像是如何构建的。

```
docker run --name webserver -d -p 8081:80 nginx
```

这条命令会用 nginx 镜像启动一个容器，命名为 `webserver`，并且映射了 `8081` 端口，这样我们可以用浏览器去访问这个 `nginx` 服务器。直接使用浏览器访问地址 http://ip:8081 或者本机 http://localhost:8081 出现如下界面。

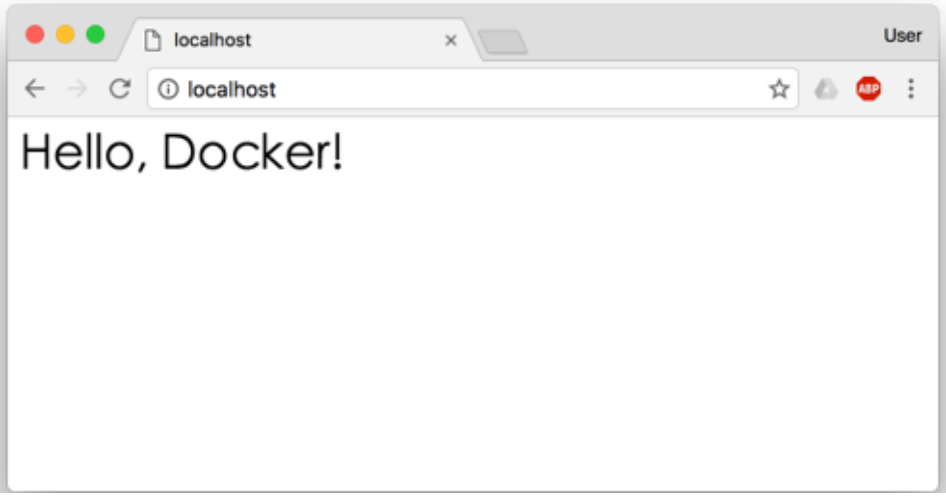
现在，假设我们非常不喜欢这个欢迎页面，我们希望改成欢迎 Docker 的文字，我们可以使用 docker exec 命令进入容器，修改其内容。

```
$ docker exec -it webserver bash

echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

exit
```

我们以交互式终端方式进入 `webserver` 容器，并执行了 bash 命令，也就是获得一个可操作的 Shell。然后，我们用 `<h1>Hello, Docker!</h1>` 覆盖了 `/usr/share/nginx/html/index.html` 的内容。现在我们再刷新浏览器的话，会发现内容被改变了。



我们修改了容器的文件，也就是改动了容器的存储层。我们可以通过 docker diff 命令看到具体的改动。现在我们定制好了变化，我们希望能将其保存下来形成镜像。要知道，当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 `docker commit` 命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

`docker commit 的语法格式为：`

`docker commit [选项] <容器 ID 或容器名> [<仓库名>[:<标签>]]`

我们可以用下面的命令将容器保存为镜像：

```
$ docker commit \
    --author "Tao Wang <twang2218@gmail.com>" \
    --message "修改了默认网页" \
    webserver \
    nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214
```

其中 --author 是指定修改的作者，而 --message 则是记录本次修改的内容。这点和 git 版本控制相似，不过这里这些信息可以省略留空。新的镜像定制好后，我们可以来运行这个镜像。

`docker run --name web2 -d -p 8082:80 nginx:v2`

这里我们命名为新的服务为 web2，并且映射到 8082 端口。我们就可以直接访问 `http://localhost:81` 看到结果，其内容应该和之前修改后的 webserver 一样。至此，

我们第一次完成了定制镜像，使用的是 docker commit 命令，手动操作给旧的镜像添加了新的一层，形成新的镜像，对镜像多层存储应该有了更直观的感觉。

（5）使用 Dockerfile 定制镜像

从刚才的 docker commit 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制。在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

```
$ mkdir mynginx
$ cd mynginx
$ touch Dockerfile
```

其内容为：

```
FROM nginx
RUN echo '<h1>Hello, Docker Test images!</h1>' > /usr/share/nginx/html/index.html
```

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令。

在 Docker Hub 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 node、openjdk、python、ruby、golang 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供

了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。如果你以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 swarm、coreos/etcd。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。

构建镜像

在 Dockerfile 文件所在目录执行：`docker build -t mynginx:v3 .`

从命令的输出结果中，我们可以清晰的看到镜像的构建过程，这里我们使用了 docker build 命令进行镜像构建。其格式为：

`docker build [选项] <上下文路径/URL/->`

在这里我们指定了最终镜像的名称 -t mynginx:v3，构建成功后，我们可以像之前运行 nginx 那样来运行这个镜像，其结果如下：

其它 docker build 的用法

（1）直接用 Git repo 进行构建

或许你已经注意到了，docker build 还支持从 URL 构建，比如可以直接从 Git repo 中构建：

例如：`$ docker build https://github.com/twang2218/gitlab-ce-zh.git#:11.1`

这行命令指定了构建所需的 Git repo，并且指定默认的 master 分支，构建目录为 /11.1/，然后 Docker 就会自己去 git clone 这个项目、切换到指定分支、并进入到指定目录后开始构建。

（2）用给定的 tar 压缩包构建

例如：`$ docker build http://server/context.tar.gz`

如果所给出的 URL 不是个 Git repo，而是个 tar 压缩包，那么 Docker 引擎会下载这个包，并自动解压缩，以其作为上下文，开始构建。

（3）从标准输入中读取 Dockerfile 进行构建

例如：`docker build - < Dockerfile` 或 `cat Dockerfile | docker build -`

如果标准输入传入的是文本文件，则将其视为 Dockerfile，并开始构建。这种形式由于直接从标准输入中读取 Dockerfile 的内容，它没有上下文，因此不可以像其他方法那样可以将本地文件 COPY 进镜像之类的事情。

（4）从标准输入中读取上下文压缩包进行构建

例如：`$ docker build - < context.tar.gz`

RUN 执行命令

RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

- `shell` 格式：RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 RUN 指令就是这种格式。
- `exec` 格式：`RUN ["可执行文件", "参数 1", "参数 2"]`，这更像是函数调用中的格式。

```
FROM debian:stretch
RUN apt-get update
RUN apt-get install -y gcc libc6-dev make wget
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

Dockerfile 中每一个指令都会建立一层，RUN 也不例外。每一个 RUN 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，commit 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多

运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误。

Union FS 是有最大层数限制的，比如 AUFS，曾经是最大不得超过 42 层，现在是不得超过 127 层。上面的 Dockerfile 正确的写法应该是这样：

```
FROM debian:stretch
RUN buildDeps='gcc libc6-dev make wget' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -rf /var/lib/apt/lists/* \
    && rm redis.tar.gz \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
```

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 && 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 \ 的命令换行方式，以及行首 # 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

（6）Dockerfile 指令详解

- **COPY 复制文件**

  格式：

  - COPY [--chown=<user>:<group>] <源路径>... <目标路径>
  - COPY [--chown=<user>:<group>] ["<源路径 1>",... "<目标路径>"]

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。COPY 指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

```
COPY package.json /usr/src/app/
```

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。在使用该指令的时候还可以加上 `--chown=<user>:<group>` 选项来改变文件的所属用户及所属组。

例如：

```
COPY --chown=55:mygroup files* /mydir/
COPY --chown=bin files* /mydir/
COPY --chown=1 files* /mydir/
COPY --chown=10:11 files* /mydir/
```

- **ADD 更高级的复制文件**

  ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。比如 `<源路径>` 可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 `<目标路径>` 去。下载后的文件权限自动设置为 `600`，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或

者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip, bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。在某些情况下，这个自动解压缩的功能非常有用，比如官方镜像 ubuntu 中：

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
```

但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 ADD 命令了。

- **CMD 容器启动命令**

CMD 指令的格式和 RUN 相似，也是两种格式：

- shell 格式：CMD <命令>
- exec 格式：CMD ["可执行文件", "参数 1", "参数 2"...]
- 参数列表格式：CMD ["参数 1", "参数 2"...]。在指定了 ENTRYPOINT 指令后，用 CMD 指定具体的参数。

之前介绍容器的时候曾经说过，Docker 不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。CMD 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，ubuntu 镜像默认的 CMD 是 /bin/bash，如果我们直接 docker run -it ubuntu 的话，会直接进入 bash。我们也可以在运行时指定运行别的命令，如 docker run -it ubuntu cat /etc/os-release。这就是用 cat /etc/os-release 命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。

在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 "，而不要使用单引号。如果使用 shell 格式的话，实际的命令会被包装为 sh -c 的参数的形式进行执行。比如：CMD echo $HOME

在实际执行中，会将其变更为：CMD ["sh", "-c", "echo $HOME"]

这就是为什么我们可以使用环境变量的原因，因为这些环境变量会被 shell

进行解析处理。提到 CMD 就不得不提容器中应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。Docker 不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样，用 upstart/systemd 去启动后台服务，容器内没有后台服务的概念。

一些初学者将 CMD 写为：

```
CMD service nginx start
```

然后发现容器执行后就立即退出了。甚至在容器内去使用 systemctl 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 service nginx start 命令，则是希望 upstart 来以后台守护进程形式启动 nginx 服务。而刚才说了 CMD service nginx start 会被理解为 CMD ["sh", "-c", "service nginx start"]，因此主进程实际上是 sh。那么当 service nginx start 命令结束后，sh 也就结束了，sh 作为主进程退出了，自然就会令容器退出。正确的做法是直接执行 nginx 可执行文件，并且要求以前台形式运行。比如：

```
CMD ["nginx", "-g", "daemon off;"]
```

- **ENTRYPOINT 入口点**

  ENTRYPOINT 的格式和 RUN 指令格式一样，分为 exec 格式和 shell 格式。ENTRYPOINT 的目的和 CMD 一样，都是在指定容器启动程序及参数。ENTRYPOINT 在运行时也可以替代，不过比 CMD 要略显繁琐，需要通过 docker run 的参数 --entrypoint 来指定。当指定了 ENTRYPOINT 后，CMD 的含义就发生了改变，不再是直接的运行其命令，而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令，换句话说实际执行时，将变为：<ENTRYPOINT> "<CMD>"

```
FROM ubuntu:18.04
RUN apt-get update  && apt-get install -y curl \
        && rm -rf /var/lib/apt/lists/*
ENTRYPOINT [ "curl", "-s", "https://ip.cn" ]
```

- **ENV 设置环境变量**

  格式有两种：

  - ENV <key> <value>
  - ENV <key1>=<value1> <key2>=<value2>...

  这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 RUN，还是运行时的应用，都可以直接使用这里定义的环境变量。

  例如：`ENV VERSION=1.0 DEBUG=on  NAME="Happy Feet"`

  或则官方 node 镜像 Dockerfile 中，就有类似这样的代码：

```
ENV NODE_VERSION 7.2.0


RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \
  && ln -s /usr/local/bin/node /usr/local/bin/nodejs
```

  在这里先定义了环境变量 NODE_VERSION，其后的 RUN 这层里，多次使用 $NODE_VERSION 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新 7.2.0 即可，Dockerfile 构建维护变得更轻松了。

- **ARG 构建参数**

  格式：`ARG <参数名>[=<默认值>]`

  构建参数和 ENV 的效果一样，都是设置环境变量。所不同的是，ARG 所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 ARG 保存密码之类的信息，因为 `docker history` 还是可以看到所有值的。

  Dockerfile 中的 ARG 指令是定义参数名称，以及定义其默认值。该默认值可以在构建命令 `docker build` 中用 `--build-arg <参数名>=<值>` 来覆

盖。在 1.13 之前的版本，要求 `--build-arg` 中的参数名，必须在 Dockerfile 中用 ARG 定义过了，换句话说，就是 `--build-arg` 指定的参数，必须在 Dockerfile 中使用了。如果对应参数没有被使用，则会报错退出构建。从 1.13 开始，这种严格的限制被放开，不再报错退出，而是显示警告信息，并继续构建。这对于使用 CI 系统，用同样的构建流程构建不同的 Dockerfile 的时候比较有帮助，避免构建命令必须根据每个 Dockerfile 的内容修改。

- **VOLUME 定义匿名卷**

  格式为：

  `VOLUME ["<路径 1>", "<路径 2>"...]` 或则 `VOLUME <路径>`

  之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中，后面的章节我们会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

  例如：`VOLUME /data`

  这里的 /data 目录就会在运行时自动挂载为匿名卷，任何向 /data 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。

  例如：

  ```
  docker run -d -v mydata:/data xxxx
  ```

  在这行命令中，就使用了 mydata 这个命名卷挂载到了 /data 这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

- **EXPOSE 声明端口**

格式为：`EXPOSE <端口 1> [<端口 2>...]`

EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是

`docker run -P` 时，会自动随机映射 EXPOSE 的端口。

要将 EXPOSE 和在运行时使用 `-p <宿主端口>:<容器端口>` 区分开来。-p，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

- **WORKDIR 指定工作目录**

格式为：`WORKDIR <工作目录路径>`

使用 WORKDIR 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如该目录不存在，WORKDIR 会帮你建立目录。例如：

```
FROM ubuntu
MAINTAINER hello
RUN mkdir /mydir
RUN echo hello world > /mydir/test.txt
WORKDIR /mydir
CMD ["more" ,"test.txt"]
```

- **USER 指定当前用户**

格式：`USER <用户名>[:<用户组>]`

USER 指令和 WORKDIR 相似，都是改变环境状态并影响以后的层。WORKDIR 是改变工作目录，USER 则是改变之后层的执行 RUN, CMD 以及 ENTRYPOINT 这类命令的身份。当然，和 WORKDIR 一样，USER 只是帮助你切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

```
RUN groupadd -r redis && useradd -r -g redis redis
USER redis
RUN [ "redis-server" ]
```

如果以 root 执行的脚本，在执行期间希望改变身份，比如希望以某个已经建立好的用户来运行某个服务进程，不要使用 su 或者 sudo，这些都需要比较麻烦的配置，而且在 TTY 缺失的环境下经常出错。建议使用 gosu

```
# 建立 redis 用户，并使用 gosu 换另一个用户执行命令
RUN groupadd -r redis && useradd -r -g redis redis
# 下载 gosu
RUN wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/1.7/gosu-amd64" \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true
# 设置 CMD，并以另外的用户执行
CMD [ "exec", "gosu", "redis", "redis-server" ]
```

六、 操作容器

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。

（一）启动容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是将在终止状态（stopped）的容器重新启动。因为 Docker 的容器实在太轻量级了，很多时候用户都是随时删除和新创建容器。

（1）新建并启动

所需要的命令主要为 `docker run`

例如：下面的命令输出一个 “Hello World”，之后终止容器。

```
docker run ubuntu:18.04 /bin/echo 'Hello world'
```

这跟在本地直接执行 `/bin/echo 'hello world'` 几乎感觉不出任何区别。

下面的命令则启动一个 `bash` 终端，允许用户进行交互。

```
docker run -t -i ubuntu:18.04 /bin/bash
```

其中，`-t` 选项让 Docker 分配一个伪终端（`pseudo-tty`）并绑定到容器的标准输入上， `-i` 则让容器的标准输入保持打开。当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器

- 执行用户指定的应用程序
- 执行完毕后容器被终止

（2）启动已终止容器

可以利用 `docker container start` 命令，直接将一个已经终止的容器启动运行。容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。除此之外，并没有其它的资源。可以在伪终端中利用 `ps` 或 `top` 来查看进程信息。可见，容器中仅运行了指定的 `bash` 应用。这种特点使得 `Docker` 对资源的利用率极高，是货真价实的轻量级虚拟化。

（二）守护态运行

（1）后台运行

更多的时候，需要让 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加 -d 参数来实现。下面举两个例子来说明一下。

如果不使用 -d 参数运行容器，容器会把输出的结果打印到宿主机上面，例如：

```
$ docker run ubuntu:18.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
hello world
hello world
hello world
hello world
```

如果使用了 -d 参数运行容器，此时容器会在后台运行并不会把输出的结果打印到宿主机上面(输出结果可以用 docker logs 查看)。

例如：

```
$ docker run -d ubuntu:18.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"

77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

> 注： 容器是否会长久运行，是和 docker run 指定的命令有关，和 -d 参数无关。

使用 -d 参数启动后会返回一个唯一的 id，也可以通过 `docker container ls` 命令来查看容器信息。

要获取容器的输出信息，可以通过 `docker container logs` 命令。

例如：

```
$ docker container logs [container ID or NAMES]
hello world
hello world
hello world
```

（三）终止

可以使用 `docker container stop` 来终止一个运行中的容器。此外，当 Docker 容器中指定的应用终结时，容器也自动终止。例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 `docker container ls -a` 命令看到。处于终止状态的容器，可以通过 `docker container start` 命令来重新启动。此外，`docker container restart` 命令会将一个运行态的容器终止，然后再重新启动它。

批量终止所有容器：`docker stop $(docker ps -a -q)`

（四）进入容器

在使用 -d 参数时，容器启动后会进入后台。某些时候需要进入容器进行操作，包括使用 `docker attach` 命令或 `docker exec` 命令，推荐大家使用 `docker exec` 命令，原因会在下面说明。

（1）**attach** 命令

下面示例如何使用 docker attach 命令。

```
$ docker run -dit ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
```

注意： 如果从这个 stdin 中 exit，会导致容器的停止。

（2）**exec** 命令

`docker exec` 后边可以跟多个参数，这里主要说明 `-i -t` 参数。

只用 `-i` 参数时，由于没有分配伪终端，界面没有我们熟悉的 Linux 命令提示符，但命令执行结果仍然可以返回。当 `-i -t` 参数一起使用时，则可

以看到我们熟悉的 Linux 命令提示符。

例如进入 ubuntu：`docker exec -it ubuntu bash`

（五）导入导出容器

（1）导出容器

如果要导出本地某个容器，可以使用 `docker export` 命令。这样将导出容器快照到本地文件。

```
$ docker container ls -a

CONTAINER ID     IMAGE         COMMAND    CREATED         STATUS                    PORTS     NAMES

7691a814370e     ubuntu:18.04  "/bin/bash"  36 hours ago  Exited (0) 21 hours ago       test

$ docker export 7691a814370e > ubuntu.tar
```

（2）导入容器快照

可以使用 `docker import` 从容器快照文件中再导入为镜像，例如

```
$ cat ubuntu.tar | docker import - test/ubuntu:v1.0

$ docker image ls

REPOSITORY         TAG                 IMAGE ID            CREATED              VIRTUAL SIZE

test/ubuntu         v1.0                9d37a6082e97        About a minute ago   171.3 MB
```

此外，也可以通过指定 URL 或者某个目录来导入，例如：

```
$ docker import http://example.com/exampleimage.tgz example/imagerepo
```

注：用户既可以使用 docker load 来导入镜像存储文件到本地镜像库，也可以使用 docker import 来导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

（3）`docker load` 与 `docker import` 区别

首先，想要清楚的了解 `docker load` 与 `docker import` 命令的区别，就必须了解镜像与容器的区别：

- 镜像：用来启动容器的只读模板，是容器启动所需的 rootfs，类似于虚拟

机所使用的镜像。

- 容器：Docker 容器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。

> 镜像是容器的基础，可以简单的理解为镜像是我们启动虚拟机时需要的镜像，容器时虚拟机成功启动后，运行的服务。

（4）`docker save` 与 `docker export` 区别

- `docker save images_name`：将一个镜像导出为文件，再使用 `docker load` 命令将文件导入为一个镜像，会保存该镜像的的所有历史记录。比 `docker export` 命令导出的文件大，很好理解，因为会保存镜像的所有历史记录。
- `docker export container_id`：将一个容器导出为文件，再使用 `docker import` 命令将容器导入成为一个新的镜像，但是相比 `docker save` 命令，容器文件会丢失所有元数据和历史记录，仅保存容器当时的状态，相当于虚拟机快照。

（六）删除容器

（1）删除终止状态的容器

可以使用 `docker container rm` 来删除一个处于终止状态的容器。例如：

```
$ docker container rm  [container id/name]
id/name
```

如果要删除一个运行中的容器，可以添加 `-f` 参数，或者使用 `docker container stop [id/name]` 停止容器，然后在进行删除操作。

批量删除所有容器：`docker rm $(docker ps -a -q)`

（2）清理所有处于终止状态的容器

用 `docker container ls -a` 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦，用下面的命令可以清理掉所有处于终止状态的容器。

```
$ docker container prune
```

其它容器案例请参阅《docker 实战案例》。