# Using Docker

## 项目中使用 Docker

Ding Zhang

Computer School, Wuhan University

Nov. 03 2021

# Outline

# Outline

# 流程

- 需求调研
- 需求调研
- 需求调研
- 开发 & 测试
- 部署
- 开发 & 测试
- 部署
- ...

# Outline

# 问题

- 环境之间会有干扰
- 管理依赖麻烦、易出错
- 部署麻烦
- 不易于升级版本
- 服务之前不能隔离
- ...

# Outline

# Outline

# 什么是 Docker

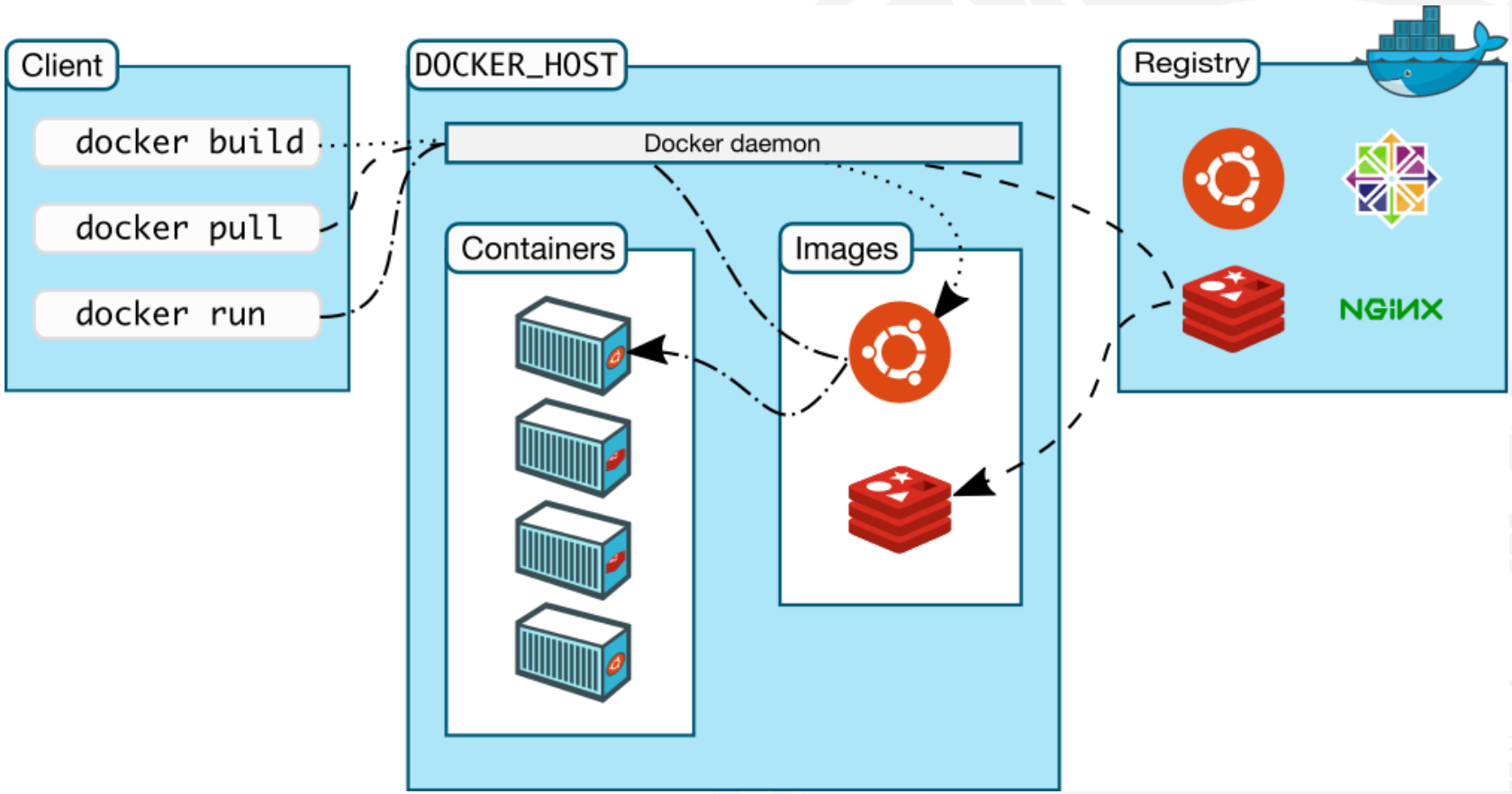
- Docker 将应用程序与该程序的依赖，打包在一个文件里面，打包到一个轻量级、可移植的容器中。
- 容器是完全使用沙箱机制，相互之间不会有任何接口，更重要的是容器性能开销极低。

# Docker 常见应用场景

- Web 应用的自动化打包和发布。
- 自动化测试和持续集成、发布。
- 在服务型环境中部署和调整数据库或其他的后台应用。
- 从头编译或者扩展现有的 OpenShift 或 Cloud Foundry 平台来搭建自己的 PaaS 环境。

# Outline

# Architecture



# Outline

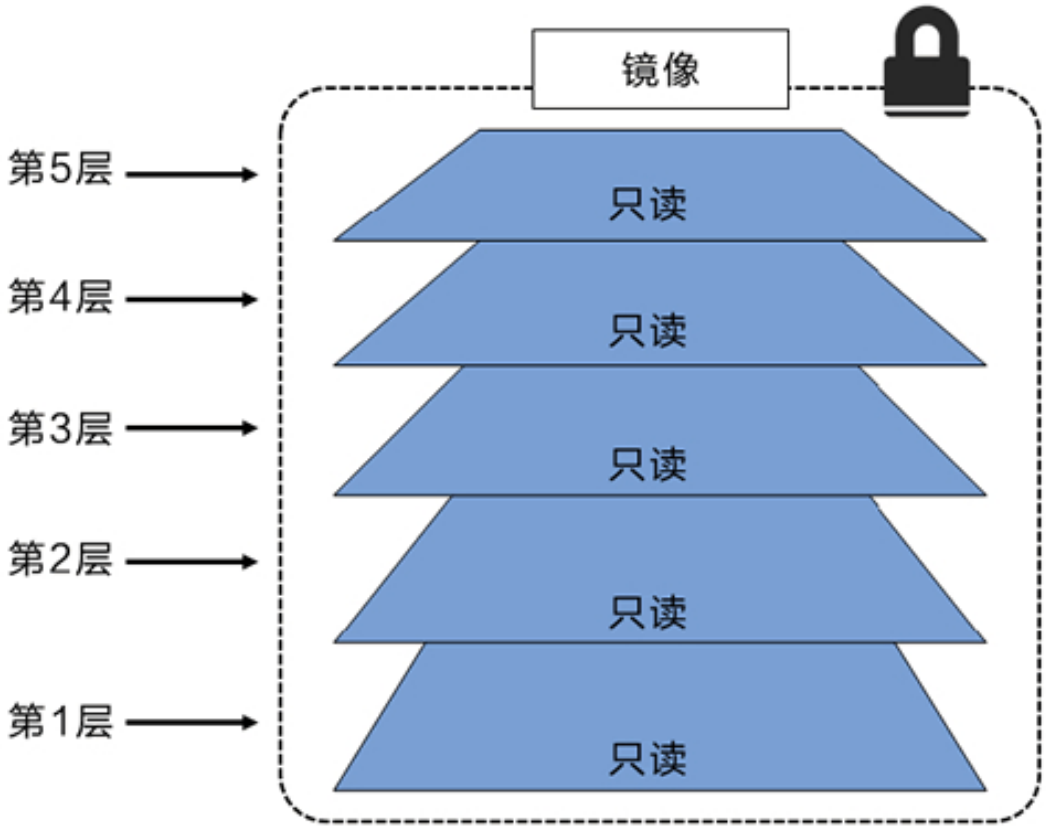
# 镜像

镜像由多个层组成，每层叠加之后，从外部看来就如一个独立的对象。镜像内部是一个精简的操作系统（OS），同时还包含应用运行所必须的文件和依赖包。

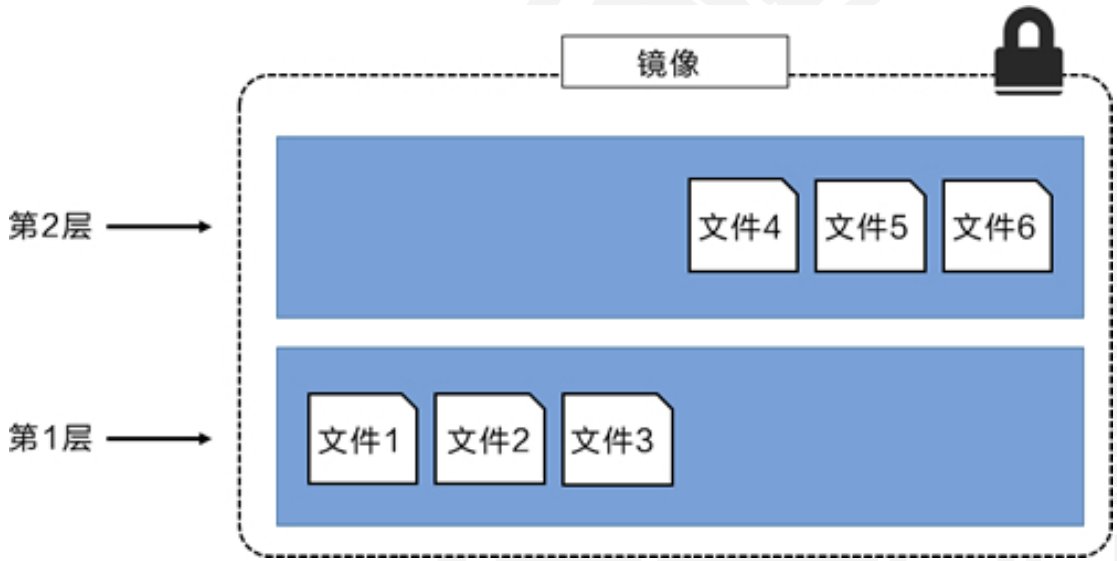
# 镜像和分层

- Docker 镜像由一些松耦合的只读镜像层组成。
- Docker 负责堆叠这些镜像层，并且将它们表示为单个统一的对象。
- 查看镜像分层的方式可以通过 `docker image inspect` 命令。
- 所有的 Docker 镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

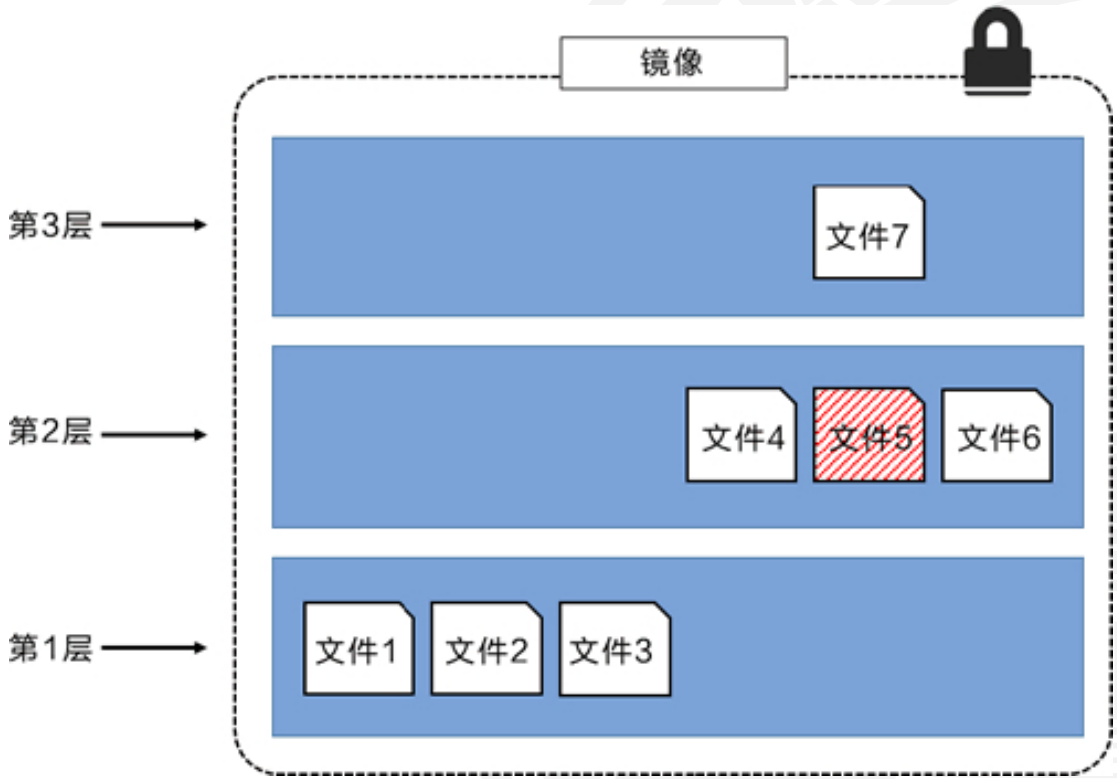
# 镜像和分层



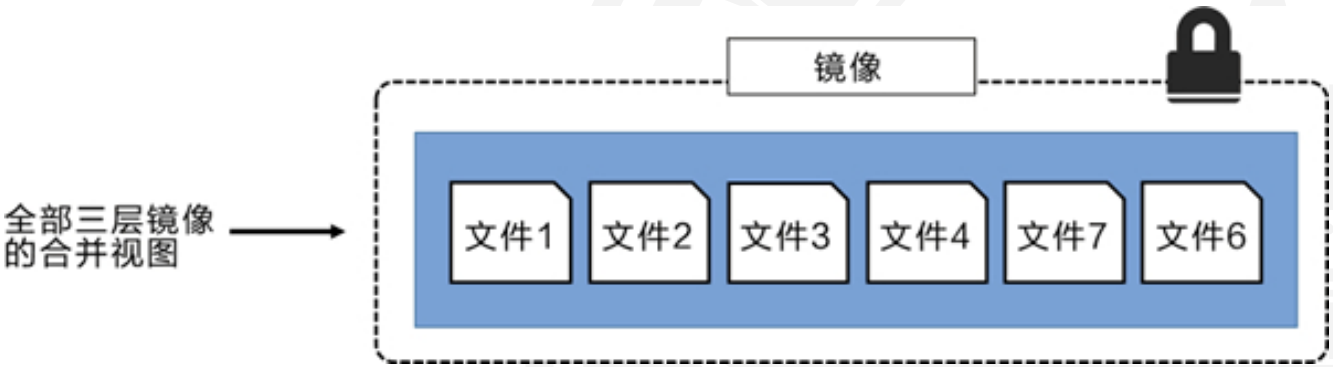
# 镜像和分层



# 镜像和分层



# 镜像和分层



# Outline

# 容器

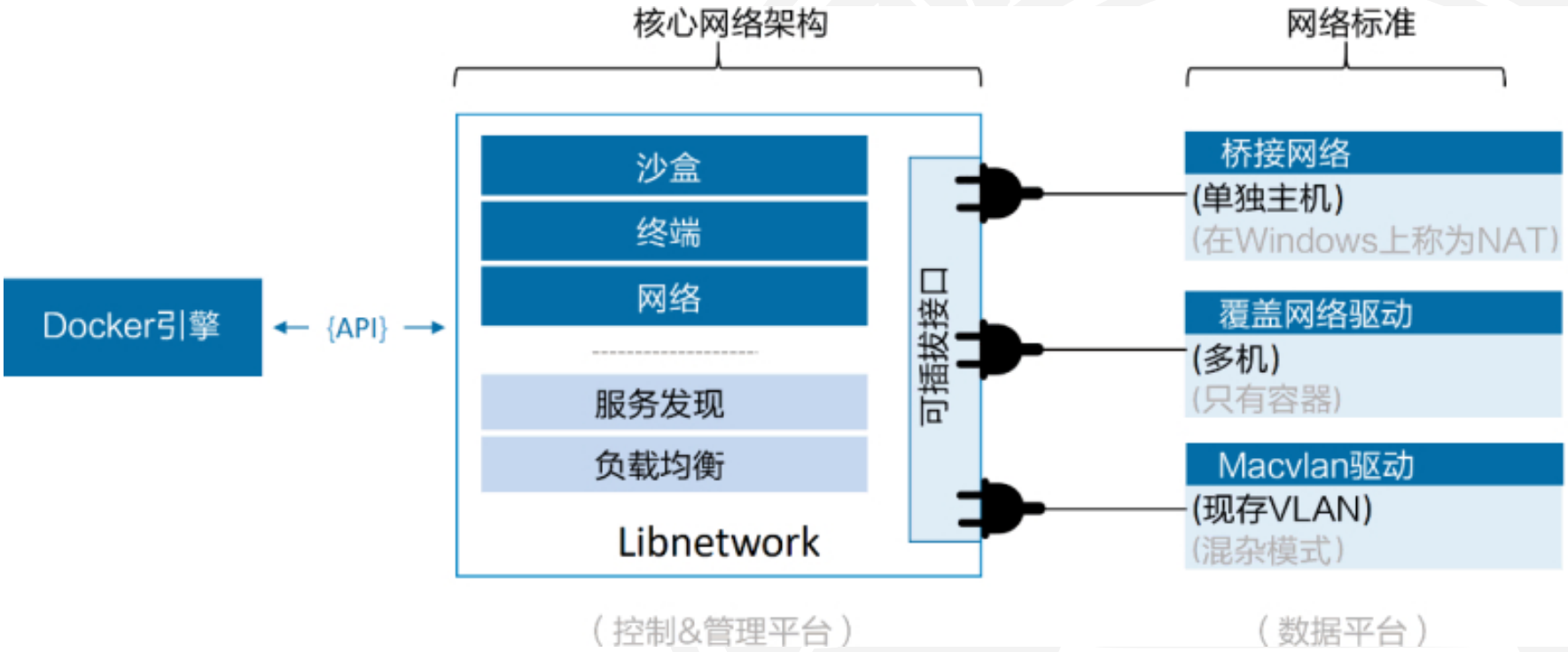
- 容器是镜像的运行时实例。
- 虚拟机和容器最大的区别是容器更快并且更轻量级。与虚拟机运行在完整的操作系统之上相比，容器会共享其所在主机的操作系统/内核。

# Outline

# 网络

- Docker 网络架构源自一种叫作容器网络模型（CNM）的方案，该方案是开源的并且支持插接式连接。
- Docker 封装了一系列本地驱动，覆盖了大部分常见的网络需求。其中包括单机桥接网络（Single-Host Bridge Network）、多机覆盖网络（Multi-Host Overlay），并且支持接入现有 VLAN。

# 网络


核心网络架构
网络标准
Docker引擎
{API}
沙盒
终端
网络
服务发现
负载均衡
Libnetwork
可插拔接口
桥接网络
(单独主机)
(在Windows上称为NAT)
覆盖网络驱动
(多机)
(只有容器)
Macvlan驱动
(现存VLAN)
(混杂模式)
( 控制&管理平台 )
( 数据平台 )

# 网络驱动

- Linux
  - Bridge
  - Overlay
  - Macvlan
- Windows
  - NAT
  - Overlay
  - Transport
  - L2 Bridge

# Outline

# 容器与非持久数据

- 每个容器都被自动分配了本地存储。
- 非持久存储属于容器的一部分，并且与容器的生命周期一致，容器创建时会创建非持久化存储，同时该存储也会随容器的删除而删除。
- Linux 系统中，该存储的目录在 `/var/lib/docker/<storage-driver>/` 之下，是容器的一部分。
- Windows 系统中位于 `C:\ProgramData\Docker\windowsfilter\` 目录之下。

# 容器与持久化数据

- 卷会挂载到容器文件系统的某个目录之下，任何写到该目录下的内容都会写到卷中。即使容器被删除，卷与其上面的数据仍然存在。

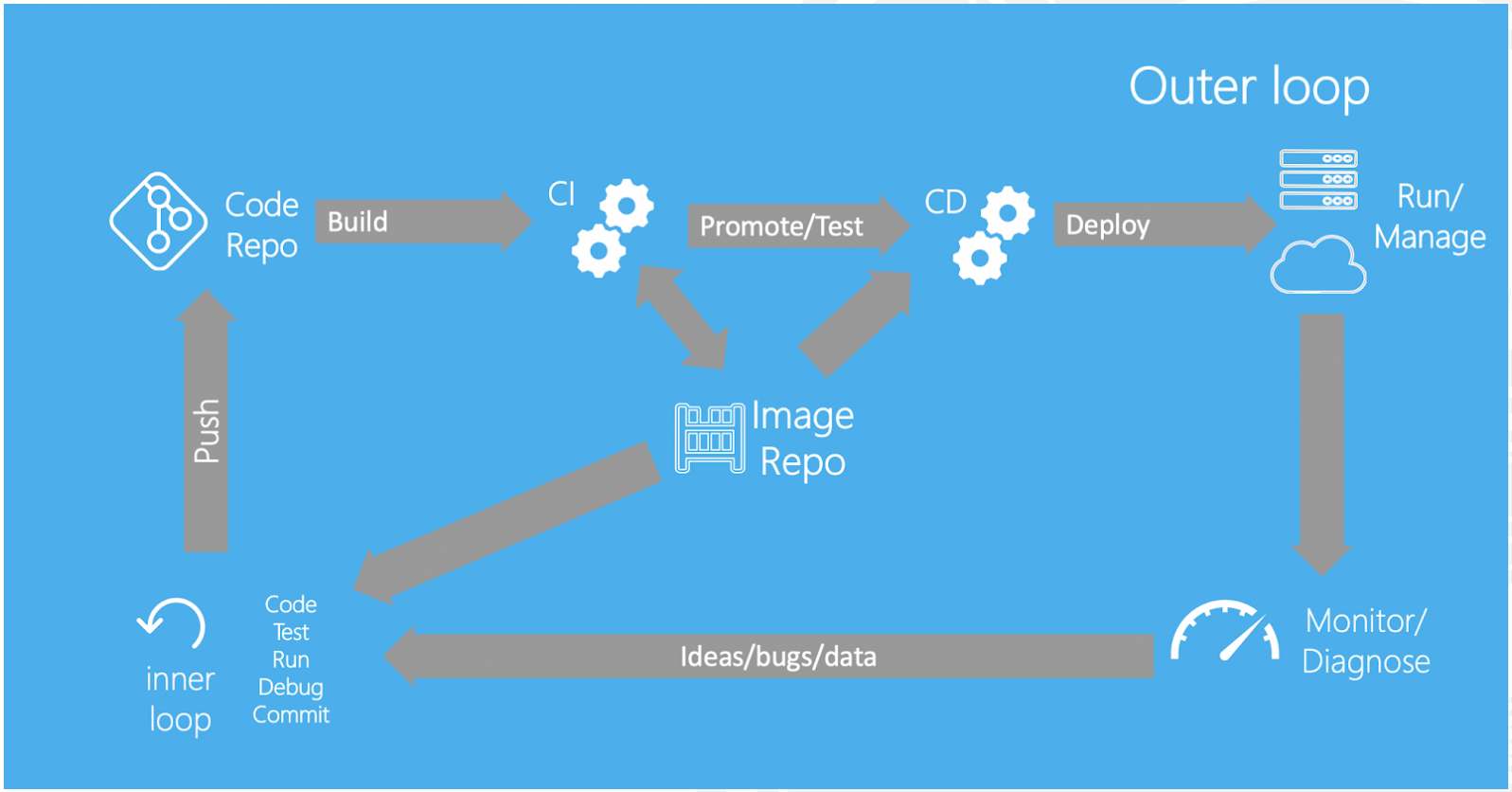
# Outline

# 实验室为什么可以使用/需要 Docker

- 不用担心操作系统是否一致（当然需要都是 Linux 或者都是 Windows）
- 可以进行自动化，减少人力手动操作
- 部署操作简单，一条命令解决（客户服务器上安装了兼容的 Docker 版本）
- 可以保证程序/服务在我们这边与客户那边保持完全一致

# Outline

# Docker Workflow



# Outline

# Outline

# Dockerfile

- 文档地址：
  `https://docs.docker.com/engine/reference/builder`

# Dockerfile

- FROM [--platform=<platform>] <image> [AS <name>]
- FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
- FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
- LABEL <key>=<value> <key>=<value> <key>=<value> ...
- ENV <key>=<value> ...
- SHELL ["executable", "parameters"]
- EXPOSE <port> [<port>/<protocol>...]
- RUN
- COPY/ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]
- WORKDIR /path/to/workdir
- VOLUME ["/data"]
- HEALTHCHECK
- ENTRYPOINT/CMD ["executable", "param1", "param2"]

# Outline

# Buildx

- 由项目 Moby BuildKit 提供。
- 使用前需要设置环境变量 `DOCKER_BUILDKIT=1`
- 设置为默认 builder：`docker buildx install`，同时会将 `docker buildx` 设置为 `docker build` 的 alias。
- 取消设置为默认 builder：`docker buildx uninstall`

# Buildx Dockerfile 前端语法

- 文档地址：`https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/syntax.md`
- Build Mounts: `RUN --mount=...`
  - `RUN --mount=type=bind target=<target_path>`（默认）
  - `RUN --mount=type=cache`
  - `RUN --mount=type=tmpfs`
  - `RUN --mount=type=secret`
  - `RUN --mount=type=ssh`
- Network modes: `RUN --network=none|host|default`
- Security context: `RUN --security=insecure|sandbox`

# Outline

# Outline

# 为什么会有这个东西

- 一个基于 Docker 的 Python 工具。
- 允许用户基于一个 YAML 文件定义多容器应用，进行多容器管理。
- 可以对应用的全生命周期进行管理。

- 使用它时，首先编写定义多容器（多服务）应用的 YAML 文件，然后将其交由 docker-compose 命令处理，Docker Compose 就会基于 Docker 引擎 API 完成应用的部署。

# Docker Compose 命令

- `docker-compose up [-d]`
- `docker-compose down`
- `docker-compose start`
- `docker-compose pause`
- `docker-compose stop`
- `docker-compose build [--no-cache]`
- `docker-compose logs [--tf][--tail=<num>]`

# Outline

# docker-compose.yml

- 文档地址：`https://docs.docker.com/compose/compose-file/compose-file-v3`

# Thanks For Listening

Reporter:Ding Zhang