上节课我们主要介绍了部署系统中各耗时环节的一些常用优化方案。课后思考题是：课程中提到了几种利用缓存的优化方案呢？如果你认真学习了课程内容，不难发现我一共提到了三种基于缓存的优化方案，它们分别是：多项目共用依赖缓存、依赖安装目录的缓存以及构建过程的持久化缓存备份。这些缓存方案不仅可以运用到传统的部署方式中，在今天介绍的容器化部署方案中也有各自的用武之地。

下面我就来介绍本节课的第一个话题：什么是容器化呢？

容器化的基本概念

什么是容器化

容器化（Containerization）通常是指以 Docker 技术为代表，将操作系统内核虚拟化的技术。和传统的虚拟机相比，容器化具有占用空间更小、性能开销更低、启动更快、支持弹性伸缩以及支持容器间互联等优势。

下面介绍 Docker 的几个基本概念。

Docker

通常我们提到的 Docker 是指运行在 Linux/Windows/macOS 中开源的虚拟化引擎，用于创建、管理和编排容器（此外，Docker 也是发布和维护该开源项目的公司名称）。

镜像

Docker 中的镜像（Image）是指用于创建容器实例的基础虚拟化模板。对于开发人员来说，可以把镜像理解为编程语言中的类：通过一个镜像可以创建多个容器实例，镜像之间也存在继承关系。通过 Docker 引擎可以构建、删除镜像，还可以将本地镜像 push 到远程仓库或者从远程拉取。例如一个基于 node:14 的镜像，在创建时不光包含了运行 node14 版本所需的 Linux 系统环境，还包含了额外打入到镜像内的 Yarn 程序。

容器

容器（Container）是 Docker 中的核心功能单元。通常一个容器内包含了一个或多个应用程序以及运行它们所需要的完整相关环境依赖。例如，基于上面例子中的 node:14 镜像，就可以创建出对应版本 NodeJS 的独立运行环境。通过 Docker 引擎可以对容器进行创建、删除、停止、恢复、与容器交互等操作。

数据挂载与数据卷

通常情况下容器内的基础数据来自创建容器的镜像。创建容器后，在容器内执行的命令如果导致容器内的数据产生变化，这些变更的数据会在容器删除的同时被清除，无法持久化保留。如果要解决持久化保留数据，可以采取两种方式：挂载容器的宿主环境（即执行启动容器命令所在的服务器）的目录或使用数据卷。数据卷可以理解为通过 Docker 引擎创建的宿主环境下的独立磁盘空间，用于在容器内读写数据，生命周期独立，不受容器生命周期的影响。

网络

Docker 容器的网络有多种驱动类型，例如 bridge、host、overlay 等。其中默认的 bridge 类型类似网桥，用于点对点访问容器间端口或者将容器端口映射到宿主环境下。而 host 则是直接使用宿主环境的网络。更多网络模型介绍可参照官方说明文档。

容器化的构建部署

在了解了容器化的基本概念后，我们再来看看什么是容器化的构建部署。

顾名思义，容器化的构建部署是把原先在部署服务器中执行的项目部署流程的各个环节，改为使用容器化的技术来完成，具体可以划分为操作镜像和操作容器两个阶段。

镜像阶段

镜像阶段的主要目标是创建一个用于部署代码的容器环境的工作镜像。以前端项目为例，工作镜像中一般会包含：特定版本的 node 环境、git、项目构建所需的其他依赖库等。有了这样的环境就可以在对应的容器中执行各部署环节。

构建镜像的具体内容写在 Dockerfile 文件中，例如下面的代码：

```
FROM node:12-slim


RUN apt-get update


RUN apt-get -y install git


RUN apt-get install -y build-essential


RUN apt-get install -y curl
```

这是一个基本的包含 node、git 等依赖程序的部署工作环境的镜像内容。

然后在 Dockerfile 所在目录下执行构建命令，即可创建相应镜像，如下所示：

```
docker build --network host --tag foo:bar .
```

容器阶段

容器阶段的主要目标就是基于项目的工作镜像创建执行部署过程的容器，并操作容器执行相应的各部署环节：获取代码、安装依赖、执行构建、产物打包、推送产物等。操作分为两个部分，创建容器与执行命令，如下面的代码所示：

```
docker run -dit --name container_1 foo:bar bash


docker exec -it container_1 xxxx
```

容器化部署过程的优势

与传统的在单台服务器上以目录区分不同部署项目的方式相比，容器化的构建部署过程有以下优点：

环境隔离

每个项目在独立的容器内执行构建部署过程，保证容器与宿主环境之间，容器与容器之间的环境隔离，防止原先共用一台服务器时可能产生的互相影响（例如项目脚本中修改了全局配置或环境变量等）。同时，环境隔离还可以保证每个项目都可以自由定制专属的环境依赖，而不必担心对其他项目产生影响。

多环境构建

使用容器化部署，可以方便地针对同一个项目生成多套不同的构建环境，达到类似 Github Actions 中的矩阵构建效果，使项目可以同时检测多套环境下的集成过程。

便于调试

用户可通过 Xterm+SSH 的方式，通过浏览器访问部署系统中的容器环境。同传统的部署方式相比，用容器化的方式可以在部署遇到问题时让用户第一时间进入容器环境中进行现场调试，效率和便捷性大大提升。之前介绍过的部署工具 CircleCI 中就提供了这一调试功能。

环境一致性与迁移效率

由于部署过程的工作环境以 Docker 镜像的方式独立制作和存储，因此可以在支持 Docker 引擎的任意服务器中使用。使用时提供一致的工作环境，无须考虑不同服务器操作系统的差异。在迁移时也可以做到一键迁移，无须重复安装环境依赖。

容器化部署过程的挑战及建议

尽管容器化的部署方式有上述优势，但也面临一些问题，例如缓存方面和性能方面的问题等。

缓存问题

项目在独立容器中构建部署时，首先面临的就是缓存方面的问题：

- **依赖缓存**：默认情况下，容器内的依赖缓存目录与宿主环境缓存目录不互通，这就导致每次依赖安装时，都无法享受宿主环境缓存带来的效率提升。同时，每次部署流程都在新容器中进行，这也导致在依赖安装的过程中，也无法读取历史依赖安装后的缓存数据。要解决这类问题，可以从两方面入手：生成容器时挂载宿主环境依赖缓存目录，以及上节课中提到的安装目录缓存。
- **构建缓存**：在传统的服务器中执行部署时，可以通过留存历史构建目录的方式来保留构建缓存数据。而在容器化的情况下，每次部署过程都会基于新容器环境重新执行各部署环节，构建过程的缓存数据也会随着部署结束、容器移除而消失。因此在这种部署方式中，尤其需要重视持久化缓存数据的留存问题。你同样可以考虑从两个方向解决：在宿主环境中创建构建缓存目录并挂载到容器中，并在项目构建配置中将缓存目录设置为该目录，这样就可以将缓存直接写入宿主服务器目录中。或者按照上节课提到的持久化缓存的备份与还原方案，将缓存备份到宿主服务器或远程存储服务器中，之后在新部署流程中进行还原使用。

性能问题

通常情况下，与传统的服务器部署相比，容器化部署并没有明显的性能差异，但是在实践中也存在一些性能方面的特殊情况：

- **容器资源限制**：在创建容器时可以通过参数来限制容器使用的 CPU 核心数和内存大小。和在普通服务器中执行部署时，完整使用所有系统资源的方式相比，限制系统资源的方式会在一定程度上导致执行过程性能的降低。在多项目使用的容器化集群方案中，这种限制通常是必要的，只是对于性能降低明显的项目而言，可以考虑在设置中分配更多的资源以提升执行效率。
- **copy-to-write**：性能问题的另一方面则体现在它独特的数据存储方式上。和传统的磁盘读写方式不同，容器中的数据是分层的，环境的数据来自镜像层，而新增的数据则来自写入容器层。在读取数据时，数据来自镜像层还是容器层并没有差别，但当修改数据时，如果修改或删除的是镜像层的数据，容器会先将数据从镜像层复制到容器层，然后进行相应操作。这种先复制后写入的模式称为 copy-to-write。因此，如果在容器的部署流程中涉及对镜像层数据的修改时，执行起来会比在普通服务器中的操作耗费更多时间。举个极端的例子，如果我们把项目的依赖安装过程写入构建镜像中，然后在容器内移动 node_modules 目录，会发现这个操作耗费的时间几乎等同于复制完整目录的时间。而同样的移动操作在普通服务器中几乎是瞬间完成的。因此，在使用容器化部署时，需要尽量避免将可变数据写入镜像中。

总结

这节课我们首先了解了以 Docker 为代表的容器化技术的基本概念：镜像、容器、数据挂载和网络等。然后讨论了容器化的构建部署需要经历的流程，先创建镜像，然后根据镜像创建容器，最后在容器内执行相关部署环节。接着分析了容器化部署的优势和劣势：容器化部署具有隔离性高、支持多环境矩阵执行、易于调试和环境标准化等优势，同时在使用时也需要额外注意对应的缓存和性能问题。

本节课的思考题是：容器化技术不仅可以应用在部署过程中，还更广泛地被应用在部署后的项目服务运行中。试比较这两种场景下对容器化技术需求的差异性。

下节课我们将进入部署效率模块的最后一节课：如何搭建基本的前端高效部署系统。