优秀总结

侯秉韬

《Java系列—微服务》培训收获：

①微服务是通过多个小型服务来构建为单个系统，这些服务是围绕业务来展开的，且理论上来讲各个服务间用到的编程语言、数据库等是可以有差异的。

②会上提到的微服务解决方案之一的 spring cloud alibaba这款开源产品，只需要做一些配置就可以利用阿里中间件来迅速搭建分布式应用系统。比如服务发现、负载均衡这些只需要一些很少的配置代码或者是几句注解就可以完成。

③最后还有提到无服务架构，初衷是单机就能实现系统的所有运行需求，这样的话就不用考虑通信安全、传输协议等诸多问题了，但是这样的话就需要有一台性能无限的主机来支持，近年来各大厂商也相继发布了自己的云产品，可以看出无服务这条路虽任重而道远但从未被放弃。

④最后总结就是没有最好的架构，只有最适合自己公司业务的架构。

张嘉乐

微服务与SpringCloud学习总结

通过微服务架构的发展历程从单体到SOA，从微服务到无服务，证实架构并不是被发明出来的，而是持续演进的结果。架构没有好坏之分只有适合自己的才是最好的，对于小型项目可从单体架构进行开发使用模块化进行管理，随着未来业务发展来改变整体架构，这就需要在项目前期就有良好的设计，这也证实架构是随着业务的变化而进行演进。

关于Spring，这是一个Java开发者必备的框架，它通过IOC/DI与AOP两个核心特性与极好的设计简化开发者的日常工作，SpringBoot又通过自动装配与Starter组件简化Spring，使开发者脱离复杂的xml配置只需要关注业务本身实现即可。

SpringCloud是一个微服务的完整解决方案，其各种框架组件解决微服务面对的核心问题，对于SpringCloud的扩展也有很多，可以根据不同的业务需求与场景来决定使用哪些扩展进行组合使用如：网关的选择（Zuul，SpringGateway），注册中心的选择（Euerka，Nacos等），服务通信的选择等等，非常的灵活，这也归功于Spring强大的生态。

陈宇佳

参加了java微服务架构的培训，我知道了微服务架构 就是 打破之前 单体架构all in one), 把每个功能元素独立处理。把独立出来的功能元素动态组合，需要的功能元素才拿来组合，需要多一些时可以整合多个功能元素。 微服务架构是对功能元素进行复制，而没有对整个应用进行复制。模块之间相互独立，通过接口完成模块间的通信，有效降低了代码的耦合度。在开发新增的业务功能时，你只需要从代码库中下载你需要的模块，并不需要下载所有的代码，开发和测试将会更加简单，并且新功能不会对原有的系统产生任何影响，系统的可扩展性得到了有效地提升。

单体架构比较初级，典型的三级架构，前端(Web/手机端)+中间业务逻辑层+数据库层。所有功能都部署在同一个服务器中的系统，采用的架构就是单体架构。

springboot可以快速的搭建一个 Spring 项目
默认使用 嵌入式 的 Servlet 容器，应用无需打成 WAR 包
有很多 starters 自动依赖和版本控制 的 启动器。 [类似于 npm 依赖包]
自动化配置，简化开发，可以配置修改默认值
无需配置XML ，无代码生成，开箱即用。
生产环境的运行应用监控
与云计算的集成

02-dockerfile&compose

docker回顾

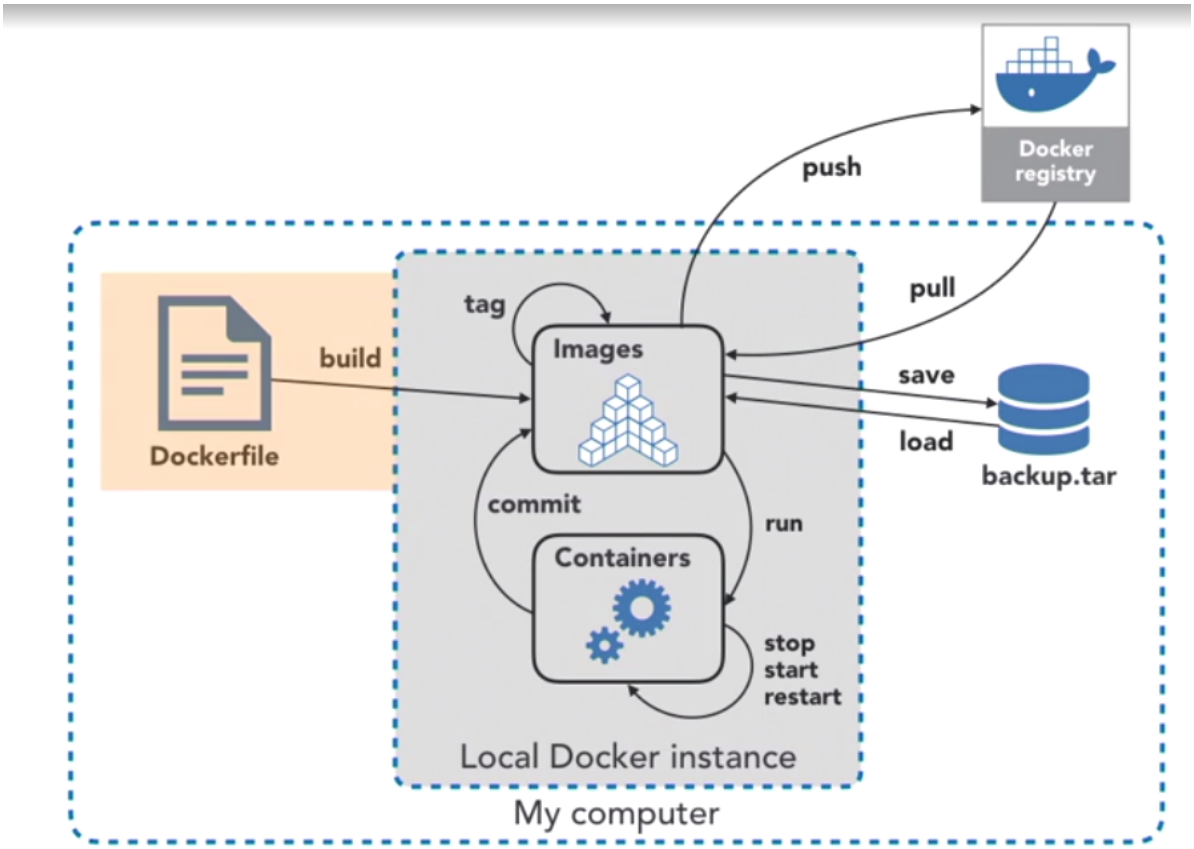
- 01-docker

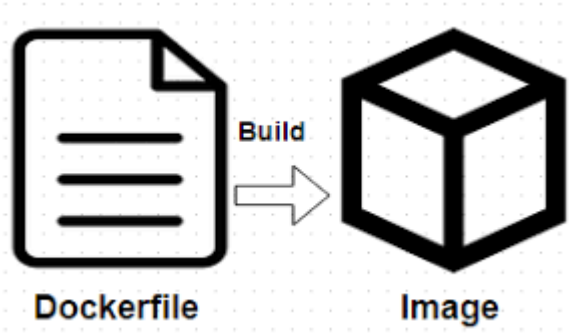
分享目的

- 能够看懂Dockerfile各个指令的用途
- 能够根据需求自行编写Dockerfile文件
- 能够修改docker-compose.yml文件

镜像相关操作

- 为什么我们需要去制作镜像
  - 拉取到https://hub.docker.com镜像后一定符合我们的需要吗？如何去修改呢？
    - exec 进入容器修改，修改完成需要reload
    - 外部挂载方式来覆盖容器内的配置
    - 自己做镜像
- 镜像生成方式
  - Dockerfile
  - 基于容器制作



- Dockerfile介绍：
  - Docker通过读取`Dockerfile`文件中的指令自动构建镜像。`Dcokerfile`是一个文本文件，它包含了构建镜像所需要执行的全部命令。执行`docker build`命令，Docker就会按照文档执行并最终创建一个镜像
  - 

dockerfile编写

格式

- Dockerfile格式：

```
# Comment
INSTRUCTION arguments
```

- 要求：
  - 要有专用的工作目录
  - 文件名首字母要大写Dockerfile文件
  - 依赖的文件需要放到工作目录，不能是工作目录之外的目录

例子：

```
mkdir mynginx
cd mynginx
vi Dockerfile
#描述信息
FROM nginx
RUN echo '<h1>Hi, Docker</h1>' > /usr/share/nginx/html/index.html

docker build -t nginx:mynginx .
docker run -it --rm -p80:80 nginx:mynginx
```

关键字介绍

- FROM 命令指定基于哪个镜像创建
  - FROM指令时最重要的一个且必须为Dockerfile文件开篇的第一个非注释行，用于为镜像文件构建过程指定基准镜像，后续的指令运行在此基准镜像所提供的运行环境
  - 实践中，基准镜像可以时任何可用镜像文件，默认情况下，docker build会在docker主机上查找指定的镜像文件，在其不存在时，则会从docker hub registry上拉去所需要的镜像文件

        - 如果找不到指定的镜像文件，docker build会返回一个错误信息
    - Syntax
        - FROM <registry>[:<tag>]
            - <registry>:指定作为base image的名称
            - <tag>:base image的标签，为可选项，省略时默认为latest；
- MAINTAINER（已废弃） 设置该镜像的作者,格式：Shaokang Li <lisk@docimax.com.cn>
- LABLE：指定kv格式元数据<key>=<value>
- ENV 设置环境变量，键值对
    - 用于为镜像定义所需的环境变量，并可被Dockerfile文件中位于其后的其他指令(ENV,ADD,COPY等)所调用
    - 调用格式为$variable_name或者${variable_name}
    - Syntax:
        - ENV <key> <value>或
        - ENV <key>=<value>
        - 第一种格式中之后的所有内容都会被视作<value>的组成部分，因此一次只能设置一个变量
        - 第二种格式可以用一次设置多个变量，每个变量为一个"<key>=<vaule>"的键值对，如果value中包含空格需要用转义符转移，或者用引号标识，另外反斜线也可以用作续行
        - 定义多个变量时，建议使用第二种方式，以便再同一层完成所有功能
- VOLUME 授权访问从容器内到主机上的目录
- ADD&COPY 复制文件到容器
    - 用于从docker主机复制文件到创建的新镜像文件
    - COPY：
        - Syntax
            - COPY <src> ...<dest> 或
            - COPY ["<src>",...<dest>]
                - <src>:要复制的源文件或目录，支持使用通配符
                - <dest>:目标路径，即正在创建image的文件系统路径；建议为<dest>使用绝对路径，否则COPY指定则以WORKDIR为起始路径。
            - 注意: 在路径中有空白字符时，通常使用第二种格式
    - 文件复制准则
        - <src>必须时build上下文中的路径，不能是其父目录中的文件
        - 如果<src>是目录，则其内部文件或者子目录会被递归复制，但<src>自身不会被复制
        - 如果指定了多个<src>，或在<src>中使用了通配符，则<dest>必须是一个目录，且必须以/结尾
        - 如果<dest>事先不存在，它将会被自动创建，这包括其父目录路径
        - 演示拷贝过程
    - ADD：
        - ADD类似于COPY指令，ADD支持使用TAR文件和URL路径
        - Syntax
            - ADD <src>...<dest>
            - ADD ["<src>"..."<dest>"]
        - 操作准则
            - 同COPY指令

        - 如果<src>为URL且<dest>不以/结尾，则<src>指定的文件将被下载并直接创建<dest>;如果<dest>以/结果，则文件名URL指定的文件将直接下载并保存为<dest>/<filename>
        - 如果<src>是本地系统上的压缩格式tar文件，它将被展开为一个目录，其行为类似于“tar -x”命令；然而，通过URL获取的tar文件不会自动展开
        - 如果<src>有多个，或其间直接使用了通配符，则<dest>必须是一个以/结尾的目录路径；如果<dest>不以/结尾，则其被视作一个普通文件，<src>的内容将被直接写入到<dest>
- WORKDIR 指定RUN/CMD/ENTRYPOINT命令的工作目录
- EXPOSE 指定容器在运行时监听的端口
- RUN 在shell或exec的环境下执行命令
    - 用于指定docker build过程中运行的应用陈旭，可以是任何指令
    - Syntax
        - RUN <command>
        - RUN ["<executable>", "<param1>", "<param2>"]
        - 第一种格式通常是一个shell命令，且以"/bin/sh -c"来运行，这意味着此进程在容器中的PID不为1，不能接受Unix信号，因此当使用docker stop <container>命令停止容器时此进程接收不到sigterm信号
        - 第二种语法格式中的参数是一个JSON格式的数组，其中executable为要运行的命令，后面则是传递给命令的选项或参数；然而，此种格式指定的命令不会以"/bin/sh -c"来发起，因此常见的shell操作如变量替换以及通配符(?,*等)替换将不会进行；不过，如果要运行的命令依赖于此shell的话可以将其替换为下面类似的格式
            - RUN ["/bin/bash", "-c", "<executable>", "<param1>"]
- CMD 容器默认的执行命令
    - 类似于RUN指令，CMD指令也可用于运行任何命令或应用程序，不过二者的运行时间点不同
        - RUN指令运行与镜像文件构建过程中，而CMD指令运行于基于Dockerfile构建出来的新镜像文件启动一个容器时
        - CMD指令的首要目的在于为启动容器指定默认的运行程序，且运行结束后，容器也将终止；CMD指定的命令也可以被docker run命令运行选项所覆盖。
        - 在Dockerfile中可以存在多个CMD指令，但是只有最后一个才会生效
        - Syntax：
            - CMD <command>
            - CMD ["<executable>", "<param1>", "<param2>"]
            - CMD ["param1", "param2"]
            - 前两种语法格式的意义同RUN
            - 第三种则用于为ENTRYPOINT指令提供默认参数

```
FROM busybox
LABEL maintainer="Shaokang Li <lisk@docimax.com.cn>"

ENV WEB_DOC_ROOT="/data/web/html/"

RUN mkdir -p $WEB_DOC_ROOT && \
    echo "<h1>Busybox http server.</h1>" > ${WEB_DOC_ROOT}/index.html
CMD /bin/httpd -f -h ${WEB_DOC_ROOT}
CMD ["/bin/httpd", "-f", "-h ${WEB_DOC_ROOT}"]
```

- ENTRYPOINT 配置给容器一个可执行的命令

    - 类似于CMD命令功能，用于为容器指定默认运行程序，从而使得容器像是一个单独的可执行程序
    - 与CMD不同的是，由ENTRYPOINT启动的应用程序不会被docker run命令行指定的参数所覆盖，而且，这些命令行参数会被当做参数传递给ENTRYPOINT指定的程序
        - 不过，docker run命令的--entrypoint选项可以覆盖ENTRYPOINT指令指定的程序
    - Syntax：
        - ENTRYPOINT <command>
        - ENTRYPOINT ["<executable>", "<param1>", "<param2>"]
        - docker run 命令传入的命令参数会覆盖CMD指令的内容并且附加到ENTRYPOINT命令最后作为其参数使用
        - Dockerfile文件中也可以存在多个ENTRYPOINT指令，但是只有最后一个生效

构建上下文

- 在执行docker build时命令最后有个点，表示当前目录，而Dockerfile就在当前目录，因此不少人以为这个路径是指Dockerfile所在路径
- 当我们进行镜像构建的时候，并非所有定制都会通过 `RUN` 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 `COPY` 指令、`ADD` 指令等。而 `docker build` 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

  这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，`docker build` 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件

dockerfile多阶构建

- 在Docker 17.05之前，构建Docker镜像通查采用两种方式：
    - 全部放入一个Dockerfile中，包括项目及其依赖库的编译、测试、打包等流程。带来的问题有镜像层次多，镜像体积大，部署时间长，而且有源代码泄露的风险。
    - 分散到多个Dockerfile中，事先在一个Dockerfile中将项目及其依赖库编译测试打包好，再将其拷贝到运行环境，这种方式需要编写两个Dockerfile和一些编译脚本才能将两个阶段整合起来，规避了第一种存在风险但是复杂度提升了不少。
- 在Docker 17.05开始支持了多阶段构建，使用多阶段构建我们在一个Dockerfile中处理上面所说的这些问题。
- 举例：

```
#See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile to build your images for faster debugging.

#Depending on the operating system of the host machines(s) that will build or run the containers, the image specified in the FROM statement may need to be changed.
#For more information, please see https://aka.ms/containercompat
#EXPOSE 443

FROM mcr.microsoft.com/dotnet/aspnet:5.0-buster-slim AS base
RUN apt-get update && apt-get install -y fontconfig
COPY ["Tools/simsun.ttc", "/usr/share/fonts/"]
WORKDIR /app
EXPOSE 8051
```

```
FROM mcr.microsoft.com/dotnet/sdk:5.0-buster-slim AS build
WORKDIR /src
COPY ["generationcenter/GenerationCenter.csproj", "generationcenter/"]
COPY ["CompanyManageRepository/CompanyManageRepository.csproj",
"CompanyManageRepository/"]
COPY ["ConfigMode/ConfigModel.csproj", "ConfigMode/"]
COPY ["Extensions/Extensions.csproj", "Extensions/"]
COPY ["CompanyManageApollo/CompanyManageApollo.csproj", "CompanyManageApollo/"]
COPY ["CompanyManageIRepository/CompanyManageIRepository.csproj",
"CompanyManageIRepository/"]
COPY ["Help/Utility.csproj", "Help/"]
COPY ["Tools/Tools.csproj", "Tools/"]
COPY ["MysqlModel/MysqlModel.csproj", "MysqlModel/"]
COPY ["CompanyRabbitMQ/CompanyRabbitMQ.csproj", "CompanyRabbitMQ/"]
COPY ["CompanyManageWebSocket/CompanyManageWebSocket.csproj",
"CompanyManageWebSocket/"]
COPY
["CompanyManageRepositoryIServices/CompanyManageRepositoryIServices.csproj",
"CompanyManageRepositoryIServices/"]
COPY ["CompanyManageRepositoryServices/CompanyManageRepositoryServices.csproj",
"CompanyManageRepositoryServices/"]
RUN dotnet restore "./generationcenter/GenerationCenter.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "generationcenter/GenerationCenter.csproj" -c Release -o
/app/build

FROM build AS publish
RUN dotnet publish "generationcenter/GenerationCenter.csproj" -c Release -o
/app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "GenerationCenter.dll"]
```

docker-compose

docker compose介绍

- 随着开发者对Docker了解的深入，使用其进行分布式部署变得复杂。开发者需要在开发，测试以及生产环境中的可移植应用，这些应用需要在不同的平台提供商之间迁移，比如在不同的云平台或者私有数据中心部署，同时，应用应该是可组合的，一个应用可以分解为多个服务。 Docker公司在2014年12月发布了三款用于解决多容器分布式软件可移植部署的问题。
- `Docker Machine`为本地，私有数据中心及公有云平台提供Docker引擎，实现从零到Docker的一键部署。
- `Docker Compose`是一个编排多容器分布式部署的工具，提供命令集管理容器化应用的完整开发周期，包括服务构建，启动和停止。
- `Docker Swarm为Docker`容器提供了原生的集群，它将多个Docker引擎的资源汇聚在一起，并提供Docker标准的API，使Docker可以轻松扩展到多台主机。

Compose是用来编排和管理多容器应用的工具，使用它，你可以通过定义一个`YAML`文件来定义你的应用的所有服务，然后通过一条命令，你就可以创建并启动所有的服务。

应用

- 安装：https://github.com/docker/compose/releases

```
curl -L "https://github.com/docker/compose/releases/download/v2.2.3/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

- 常用命令
  - `docker-compose up -d` 用于部署一个 Compose 应用
  - `docker-compose down` 停止并删除运行中的 Compose 应用
  - `docker-compose up -d  <application_name>` 更新yaml文件中的单个服务
  - `docker-compose -f <dockercompose_file> up -d` 指定非标准命名的Compose文件

> 参考文章：
>
> https://jiajially.gitbooks.io/dockerguide/content/dockerIND.html
>
> https://zhuanlan.zhihu.com/p/79949030
>
> https://dockerdocs.cn/develop/develop-images/dockerfile_best-practices/index.html #最佳实践
>
> https://dockerdocs.cn/compose/
>
> https://yeasy.gitbook.io/docker_practice/container/run
>
> https://yeasy.gitbook.io/docker_practice/image/build#jing-xiang-gou-jian-shang-xia-wen-context #docker build上下文介绍
>
> https://yeasy.gitbook.io/docker_practice/image/multistage-builds #dockerfile多阶构建