容器镜像服务

最佳实践

天翼云科技有限公司

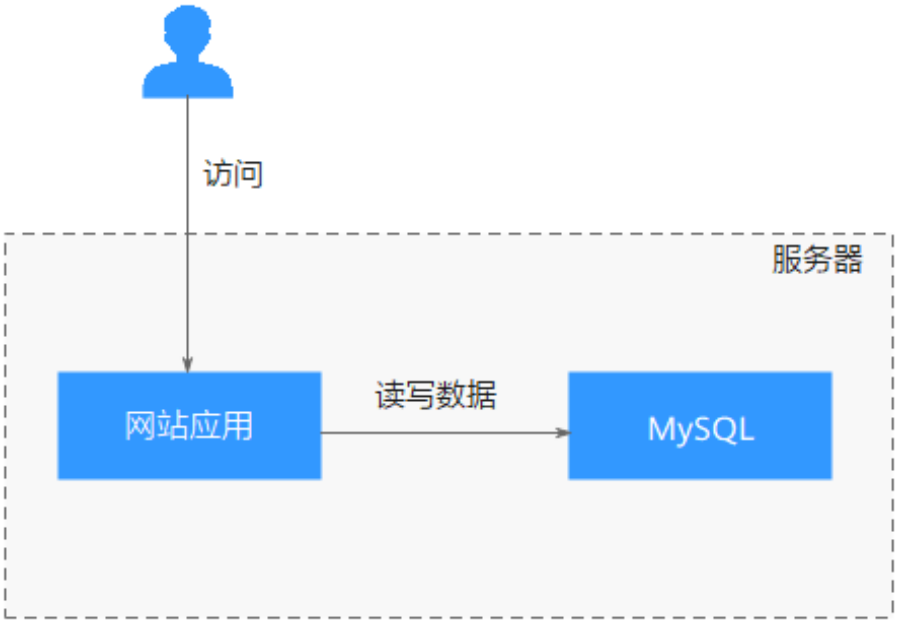
目 录

1 镜像类

1.1 编写高效的 Dockerfile

本章基于容器镜像服务实践所编写，将一个单体应用进行容器改造为例，展示如何写出可读性更好的 Dockerfile，从而提升镜像构建速度，构建层数更少、体积更小的镜像。

下面是一个常见企业门户网站架构，由一个 Web Server 和一个数据库组成，Web Server 提供 Web 服务，数据库保存用户数据。通常情况下，这样一个门户网站安装在一台服务器上。



如果把应用运行在一个 Docker 容器中，那么很可能写出下面这样的 Dockerfile 来。

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y
```

```
RUN apt-get install -y nodejs ssh mysql
RUN cd /app && npm install

# this should start three processes, mysql and ssh
# in the background and node app in foreground
# isn't it beautifully terrible? <3
CMD mysql & sshd & npm start
```

但是这样 Dockerfile 有很多问题，这里 CMD 命令是错误的，只是为了说明问题而写。

下面的内容中将展示对这个 Dockerfile 进行改造，说明如何写出更好的 Dockerfile，共有如下几种处理方法。

- 一个容器只运行一个进程
- 不要在构建中升级版本
- 将变化频率一样的 RUN 指令合一
- 使用特定的标签
- 删除多余文件
- 选择合适的基础镜像
- 设置 WORKDIR 和 CMD
- 使用 ENTRYPOINT（可选）
- ENTRYPOINT 脚本中使用 exec
- 优先使用 COPY
- 合理调整 COPY 与 RUN 的顺序
- 设置默认的环境变量、映射端口和数据库逻辑卷
- 使用 EXPOSE 暴露端口
- 使用 VOLUME 管理数据库逻辑卷
- 使用 LABEL 设置镜像元数据
- 添加 HEALTHCHECK
- 编写.dockerignore 文件

一个容器只运行一个进程

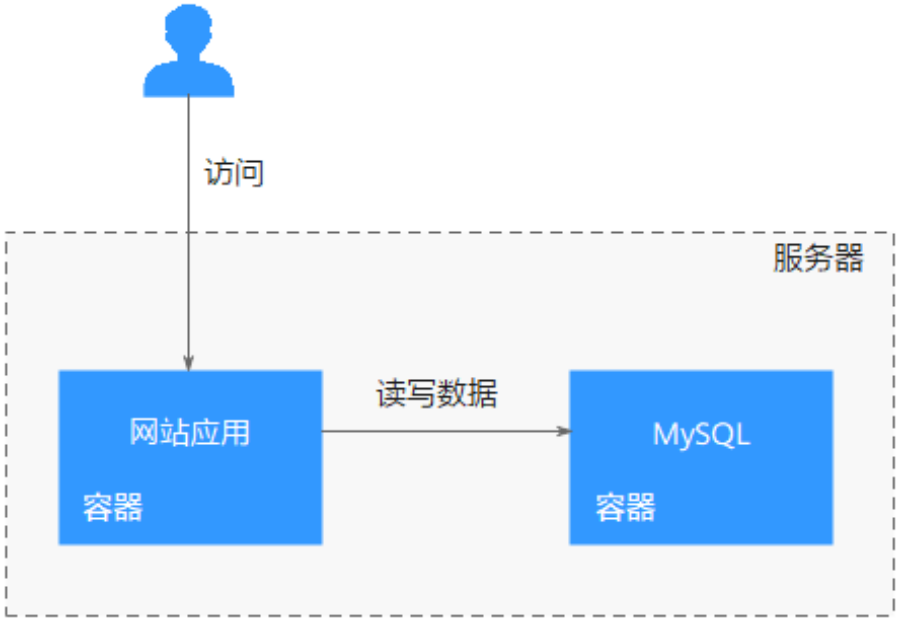
从技术角度讲，Docker 容器中可以运行多个进程，您可以将数据库、前端、后端、ssh 等都运行在同一个 Docker 容器中。但是，这样跟未使用容器前没有太大区别，且这样容器的构建时间非常长（一处修改就要构建全部），镜像体积大，横向扩展时非常浪费资源（不同的应用需要运行的容器数并不相同）。

通常所说的容器化改造是对应用整体微服务进行架构改造，改造后，再容器化。这样做可以带来如下好处：

- 单独扩展：拆分为微服务后，可单独增加或缩减每个微服务的实例数量。
- 提升开发速度：各微服务之间解耦，某个微服务的代码开发不影响其他微服务。
- 通过隔离确保安全：整体应用中，若存在安全漏洞，一旦被攻击，所有功能的权限都可能会被窃取。微服务架构中，若攻击了某个服务，只可获得该服务的访问权限，无法入侵其他服务。

- 提升稳定性：如果其中一个微服务崩溃，其他微服务还可以持续正常运行。

因此，上述企业门户网站可以进行如下改造，Web 应用和 MySQL 运行在不同容器中。



MySQL 运行在独立的镜像中，这样的好处就是，可以对它们分别进行修改，且不会牵一发而动全身。如下面这个例子所示，可以删除 MySQL，只安装 node.js。

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

不要在构建中升级版本

为了降低复杂性、减少依赖、减小文件大小、节约构建时间，你应该避免安装任何不必要的包。例如，不要在数据库镜像中包含一个文本编辑器。

如果基础镜像中的某个包过时了，但你不知道具体是哪一个包，你应该联系它的维护者。如果你确定某个特定的包，比如 foo 需要升级，使用 apt-get install -y foo 就行，该指令会自动升级 foo 包。

apt-get upgrade 会使得镜像构建过程非常不稳定，在构建时不确定哪些包会被安装，此时可能会产生不一致的镜像。因此通常会删掉 apt-get upgrade。

删掉 apt-get upgrade 后，Dockerfile 如下：

```
FROM ubuntu

ADD . /app

RUN apt-get update

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

将变化频率一样的 RUN 指令合一

Docker 镜像是分层的，类似于洋葱，它们都有很多层，为了修改内层，则需要将外面的层都删掉。Docker 镜像有如下特性：

- Dockerfile 中的每个指令都会创建一个新的镜像层。
- 镜像层将被缓存和复用。
- Dockerfile 修改后，复制的文件变化了或者构建镜像时指定的变量不同了，对应的镜像层缓存就会失效。
- 某一层的镜像缓存失效之后，它之后的镜像层缓存都会失效。
- 镜像层是不可变的，如果在某一层中添加一个文件，然后在下一层中删除它，则镜像中依然会包含该文件，只是这个文件在 Docker 容器中不可见。

将变化频率一样的指令合并在一起，目的是为了更好的将镜像分层，避免带来不必要的成本。如本例中将 node.js 安装与 npm 模块安装放在一起的话，则每次修改源代码，都需要重新安装 node.js，这显然不合适。

```
FROM ubuntu

ADD . /app

RUN apt-get update \
    && apt-get install -y nodejs \
    && cd /app \
    && npm install

CMD npm start
```

因此，正确的写法是这样的：

```
FROM ubuntu

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

使用特定的标签

当镜像没有指定标签时，将默认使用 latest 标签。因此，FROM ubuntu 指令等同于 FROM ubuntu:latest。当镜像更新时，latest 标签会指向不同的镜像，这时构建镜像有可能失败。

如下示例中使用 16.04 作为标签。

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

删除多余文件

假设更新了 apt-get 源，下载解压并安装了一些软件包，它们都保存在“/var/lib/apt/lists/”目录中。

但是，运行应用时 Docker 镜像中并不需要这些文件。因此最好将它们删除，因为它会使 Docker 镜像变大。

示例 Dockerfile 中，删除“/var/lib/apt/lists/”目录中的文件。

```
FROM ubuntu:16.04

RUN apt-get update \
   && apt-get install -y nodejs \
   && rm -rf /var/lib/apt/lists/*

ADD . /app
RUN cd /app && npm install

CMD npm start
```

选择合适的基础镜像

在示例中，选择了 ubuntu 作为基础镜像。但是只需要运行 node 程序，没有必要使用一个通用的基础镜像，node 镜像应该是更好的选择。

更好的选择是 alpine 版本的 node 镜像。alpine 是一个极小化的 Linux 发行版，只有 4MB，这让它非常适合作为基础镜像。

```
FROM node:7-alpine

ADD . /app
RUN cd /app && npm install

CMD npm start
```

设置 WORKDIR 和 CMD

WORKDIR 指令可以设置默认目录，也就是运行 RUN / CMD / ENTRYPOINT 指令的地方。

CMD 指令可以设置容器创建时执行的默认命令。另外，您应该将命令写在一个数组中，数组中每个元素为命令的每个单词。

```
FROM node:7-alpine

WORKDIR /app
ADD . /app
RUN npm install

CMD ["npm", "start"]
```

使用 ENTRYPOINT（可选）

ENTRYPOINT 指令并不是必须的，因为它会增加复杂度。ENTRYPOINT 是一个脚本，它会默认执行，并且将指定的命令作为其参数。它通常用于构建可执行的 Docker 镜像。

```
FROM node:7-alpine

WORKDIR /app
ADD . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

ENTRYPOINT 脚本中使用 exec

在前文的 ENTRYPOINT 脚本中，使用了 exec 命令运行 node 应用。不使用 exec 的话，则不能顺利地关闭容器，因为 SIGTERM 信号会被 bash 脚本进程吞没。exec 命令启动的进程可以取代脚本进程，因此所有的信号都会正常工作。

优先使用 COPY

COPY 指令非常简单，仅用于将文件拷贝到镜像中。ADD 相对来讲复杂一些，可以用于下载远程文件以及解压压缩包。

```
FROM node:7-alpine

WORKDIR /app

COPY . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

合理调整 COPY 与 RUN 的顺序

将变化最少的部分放在 Dockerfile 的前面，这样可以充分利用镜像缓存。

示例中，源代码会经常变化，则每次构建镜像时都需要重新安装 NPM 模块，这显然不是希望看到的。因此可以先拷贝 package.json，然后安装 NPM 模块，最后才拷贝其余的源代码。这样的话，即使源代码变化，也不需要重新安装 NPM 模块。

```
FROM node:7-alpine

WORKDIR /app

COPY package.json /app
RUN npm install
COPY . /app

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

设置默认的环境变量、映射端口和数据库逻辑卷

运行 Docker 容器时很可能需要一些环境变量。在 Dockerfile 设置默认的环境变量是一种很好的方式。另外，应该在 Dockerfile 中设置映射端口和数据库逻辑卷。示例如下：

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

ENV 指令指定的环境变量在容器中可以使用。如果你只是需要指定构建镜像时的变量，你可以使用 ARG 指令。

使用 EXPOSE 暴露端口

EXPOSE 指令用于指定容器将要监听的端口。因此，你应该为你的应用程序使用常见的端口。例如，提供 Apache web 服务的镜像应该使用 EXPOSE 80，而提供 MongoDB 服务的镜像使用 EXPOSE 27017。

对于外部访问，用户可以在执行 docker run 时使用一个标志来指示如何将指定的端口映射到所选择的端口。

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR
```

```
COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV APP_PORT=3000
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

使用 VOLUME 管理数据库逻辑卷

VOLUME 指令用于暴露任何数据库存储文件、配置文件或容器创建的文件和目录。强烈建议使用 VOLUME 来管理镜像中的可变部分和用户可以改变的部分。

下面示例中填写一个媒体目录。

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

使用 LABEL 设置镜像元数据

你可以给镜像添加标签来帮助组织镜像、记录许可信息、辅助自动化构建等。每个标签一行，由 LABEL 开头加上一个或多个标签对。

须知

如果你的字符串中包含空格，必须将字符串放入引号中或者对空格使用转义。如果字符串内容本身就包含引号，必须对引号使用转义。

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"
```

添加 HEALTHCHECK

运行容器时，可以指定--restart always 选项。这样的话，容器崩溃时，docker daemon 会重启容器。对于需要长时间运行的容器，这个选项非常有用。但是，如果容器的确在运行，但是不可用怎么办？使用 HEALTHCHECK 指令可以让 Docker 周期性的检查容器的健康状况。只需要指定一个命令，如果一切正常的话返回 0，否则返回 1。当请求失败时，curl --fail 命令返回非 0 状态。示例如下：

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"

ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
   APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit 1

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

编写.dockerignore 文件

.dockerignore 的作用和语法类似于.gitignore，可以忽略一些不需要的文件，这样可以有效加快镜像构建时间，同时减少 Docker 镜像的大小。

构建镜像时，Docker 需要先准备 context，将所有需要的文件收集到进程中。默认的 context 包含 Dockerfile 目录中的所有文件，但是实际上，并不需要.git 目录等内容。

示例如下：

```
.git/
```

1.2 创建 JDK8 基础镜像并上传至 SWR

场景概述

在项目中，我们通常基于相同的基础镜像创建镜像。本章节将以 JDK8 基础镜像为例，介绍如何在 CCE 节点上创建 JDK8 基础镜像并上传至 SWR。

操作步骤

步骤 1 购买一个 CCE 集群。

- 登录 CCE 控制台

- 在购买 CCE 集群页面配置各项集群参数，详细请参考 CCE 帮助文档 > 集群 > 创建集群。
- 在订购确认界面中会显示集群的资源配置、计费模式等信息，单击“提交”。

  等待集群创建成功。创建成功后在集群管理下会显示一个运行中的集群，且集群节点数量为 0。

步骤 2 创建 CCE 节点。

集群创建成功后，您还需要在集群中创建运行工作负载的节点。CCE 节点默认安装了 Linux 操作系统和 Docker。我们可以用它创建基础镜像。

在下面的步骤中，我们将以 Centos7.6 为例，详细介绍如何创建 JDK8 基础镜像，并将它上传到 SWR。

- 登录 CCE 控制台。
- 单击 1 中创建的集群，进入集群控制台。
- 在左侧菜单栏选择节点管理，单击右上角“创建节点”，在节点创建页面中配置集群工作节点的参数。
- 在网络配置中，选择“自动创建”1 个弹性公网 IP，带宽为 1Mbit/s。
- 在页面最下方选择节点的数量和计费模式，单击“下一步: 规格确认”。
- 查看节点规格无误后，阅读页面上的使用说明，勾选“我已阅读并知晓上述使用说明”，单击“提交”。

  等待节点创建成功。创建成功后在节点管理下会显示一个运行中的节点。

图1-1 CCE 节点示例

步骤 3 下载 JDK 软件包。

- 节点创建成功后，单击节点名称，进入云服务器详情页。
- 在云服务器详情页，单击右上角“远程登录”。
- 选择一种登录方式，单击“登录”。
- 以 root 用户登录弹性云服务器。
- 新建一个目录 image。

  mkdir image
- 进入 image 目录。

  cd image

- 下载 JDK 软件包。

  wget https://builds.openlogic.com/downloadJDK/openlogic-openjdk/8u352-b08/openlogic-openjdk-8u352-b08-linux-x64.tar.gz

步骤 4 **构建一个镜像。**

- 执行 **vi dockerfile** 命令，编写一个 Dockerfile，并写入以下信息：

```
FROM centos                                                                    #
使用 centos 作为基础镜像
RUN useradd -d /home/springboot -m springboot
#在工作目录下创建一个用户
ADD ./openlogic-openjdk-8u352-b08-linux-x64.tar.gz /home/springboot
#拷贝 jdk 软件包到镜像，并自动解压
RUN chown springboot:springboot /home/springboot/openlogic-openjdk-8u352-b08-
linux-x64 -R
USER springboot
#指定用户为 springboot
ENV  JAVA_HOME=/home/springboot/openlogic-openjdk-8u352-b08-linux-x64
#设置环境变量
ENV   PATH=$JAVA_HOME/bin:$PATH  \
    CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
WORKDIR /home/springboot/
#指定镜像的工作目录
```

说明

以上仅为示例，如果您想详细了解 Dockerfile 的详细使用帮助，请查看 **Dockerfile reference**。

- 按 **ESC**,输入**:wq**,保存 Dockerfile，并退出编辑。
- 执行下面的命令，构建一个镜像。

  docker build -t openjdk:8 .

- 使用 **docker images** 命令，查看镜像是否构建成功。

步骤 5 **登录容器镜像服务控制台，并创建一个组织。**

示例：这里我们创建一个名为 test 的组织。

步骤 6 **上传镜像到 5 的组织下。**

- 以 root 用户登录容器镜像服务控制台。
- 为镜像打标签。

  *示例如下：**docker tag openjdk:8 registry.cn-jssz1.ctyun.cn/test/0penjdk:v8.8***

- 上传镜像到步骤 5 的组织下。

docker push registry.cn-jssz1.ctyun.cn/test/0penjdk:v8.8

镜像上传成功后，我们可以在容器镜像服务控制台-“我的镜像”中找到刚刚上传成功的镜像。

步骤 7 **(可选）镜像上传成功后，你可以使用已上传的镜像在 CCE 中部署工作负载。**

----结束

2 使用类

2.1 配置 SWR 访问网络

2.1.1 方案概述

当我们使用云上的 ECS 或 CCE 节点作为安装容器引擎的客户端时，如何配置网络，才能保证上传下载的最佳运行速度？这里将介绍几种常见方案，用户可以根据自己的实际使用场景来选择。

2.1.2 内网访问

当您使用的安装容器引擎的客户端为云上的 ECS 或 CCE 节点，且机器与容器镜像仓库在同一 region 时，上传下载镜像走内网链路。

您无需进行任何访问配置，直接通过容器镜像的上传下载地址访问 SWR 即可。

2.1.3 公网访问

场景描述

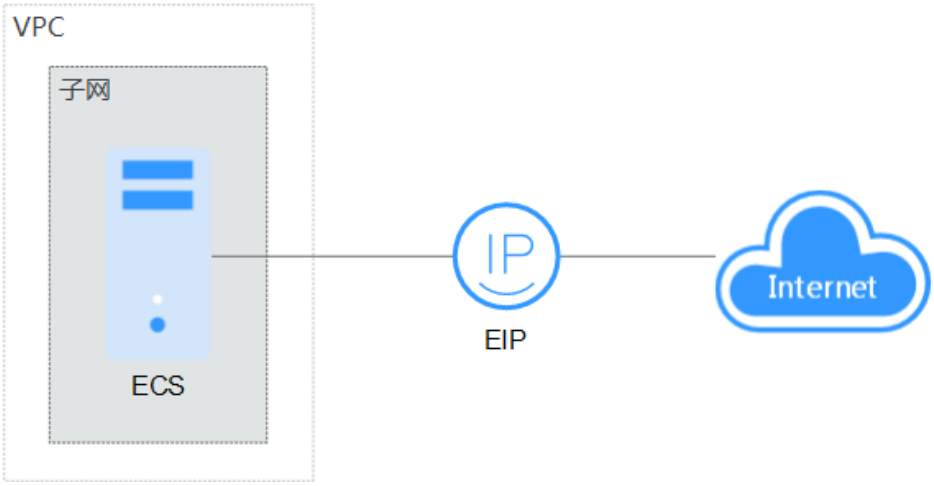
该场景下安装容器引擎的机器为云上的 ECS 或 CCE 节点，机器与容器镜像仓库不在同一区域，上传下载镜像走公网链路，机器需要绑定弹性公网 IP。在此场景下，又分为 2 种情形。

- 单个 ECS 访问公网
- 多个 ECS 访问公网

单个 ECS 访问公网

当您的某台 ECS 需要主动访问公网，可以为 ECS 绑定 EIP，即可实现公网访问。天翼云提供多种计费方式（按需、按流量等）供您选择，无需使用时支持灵活解绑。

图2-1 组网图



步骤 2 登录管理控制台。

步骤 3 单击管理控制台左上角的，选择区域和项目。

步骤 4 单击，选择“计算 > 弹性云服务器”。

步骤 5 在弹性云服务器列表中，选中要绑定弹性公网 IP 的机器，单击“操作”列下的“更多 > 网络设置 > 绑定弹性公网 IP”。

步骤 6 选择一个弹性公网 IP，单击“确定”。

步骤 7 完成绑定后，可以在云服务器列表页查看已绑定的弹性公网 IP。

说明

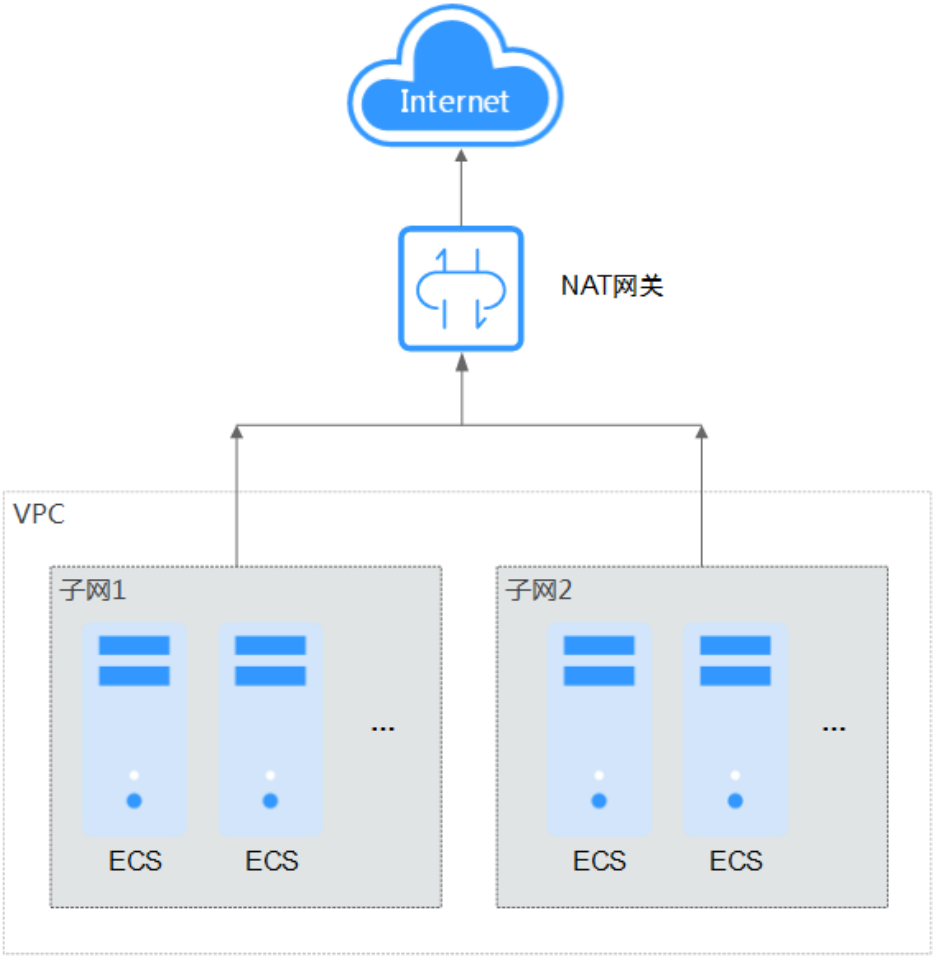
如果当前区域没有可用的弹性公网 IP，则弹性公网 IP 列表为空，请购买弹性公网 IP 后重新执行绑定操作。

----结束

多个 ECS 访问公网

当您的 VPC 内 ECS 都有公网访问需求时，可以使用 NAT 网关服务，按子网配置 SNAT 规则，轻松构建 VPC 的公网出口。对比 EIP 访问公网，在未配置 SNAT 规则时，外部用户无法通过公网直接访问 NAT 网关的公网地址，保证了 ECS 的相对安全。

图2-2 组网图



步骤 1 参考弹性公网 IP 和带宽帮助文档，创建 EIP 并绑定带宽。

步骤 2 创建 NAT 网关，具体步骤详见 NAT 网关帮助中心。

a. 登录管理控制台。

b. 在管理控制台左上角单击，选择区域和项目。

c. 在控制台首页，单击左上角的，在展开的列表中单击“网络 > NAT 网关”。

d. 在 NAT 网关页面，单击右上角的“创建公网 NAT 网关”。

e. 根据界面提示配置参数。

步骤 3 配置 SNAT 规则，为子网绑定弹性公网 IP，具体请参见 NAT 网关 > 添加 SNAT 规则。

a. 登录管理控制台。

b. 在管理控制台左上角单击，选择区域和项目。

c. 在控制台首页，单击左上角的 ☰，在展开的列表中单击“网络 >NAT 网关”。

d. 在 NAT 网关页面，单击需要添加 SNAT 规则的 NAT 网关名称。

e. 在 SNAT 规则页签中，单击“添加 SNAT 规则”。

f. 根据界面提示配置参数，将弹性公网 IP 与 ECS 所在子网进行关联。

图2-3 添加 SNAT 规则

----结束

3 迁移类

3.1 容器镜像迁移

3.1.1 方案概述

应用现状

随着容器化技术的发展，越来越多的企业使用容器代替了虚拟机完成应用的运行部署。目前许多企业选择自建 Kubernetes 集群，但是自建集群往往有着沉重的运维负担，需要运维人员自己配置管理系统和监控解决方案。企业自运维大批镜像资源，意味着要付出高昂的运维、人力、管理成本，且效率不高。

容器镜像服务支持 Linux、ARM 等多架构容器镜像托管。企业可以将镜像仓库迁移到容器镜像服务，节省运维成本。

如何把已有的镜像仓库平滑地迁移到容器镜像服务？这里将介绍几种常见的方案，用户可以根据自己的实际使用场景来选择。

迁移方案

表3-1 迁移方案及适用场景对比

方案类型	适用场景	注意事项
使用 docker 命令迁移镜像至 SWR	待迁移的镜像数量较少	• 依赖磁盘存储，需要及时进行本地镜像的清理，而且落盘形成多余的时间开销，难以胜任生产场景中大量镜像的迁移。 • 依赖 docker 程序，docker daemon 对 pull/push 的并发数进行了严格的限制，没法进行高并发同步。 • 一些功能只能经过 HTTP api 进行操作，单纯使用 docker

方案类型	适用场景	注意事项
		cli 没法做到，使脚本变得复杂。
使用 image-syncer 迁移镜像至 SWR	待迁移的镜像数量庞大	• 支持多对多镜像仓库同步。 • 支持基于 Docker Registry V2 搭建的 docker 镜像仓库服务(如 Docker Hub、Quay、Harbor 等)。 • 同步只通过内存和网络，不依赖磁盘存储，同步速度快。 • 增量同步，经过对同步过的镜像 blob 信息落盘，不重复同步已同步的镜像。 • 并发同步，能够经过配置文件调整并发数。 • 自动重试失败的同步任务，能够解决大部分镜像同步中的网络抖动问题。 • 不依赖 docker 以及其余程序。

3.1.2 使用 docker 命令迁移镜像至 SWR

场景描述

容器镜像服务提供了简便、易用的镜像托管和高效分发业务。当要迁移的镜像数量较少时，企业可以通过简单的 docker pull、docker push 命令行，将之前维护的镜像迁移到 SWR 上。

操作步骤

步骤 1 从源仓库下载镜像。

使用 docker pull 命令下载镜像。

示例：**docker pull nginx:latest**

使用 **docker images** 命令查看是否下载成功。

```
# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
nginx                         latest    22f2bf2e2b4f   5 hours ago     22.8MB
```

步骤 2 将步骤 1 中下载的镜像上传到 SWR。

- 登录到目标端容器所在虚拟机，并登录 SWR。
- 给镜像打标签。

  docker tag [镜像名称:版本名称] [镜像仓库地址]/[组织名称]/[镜像名称:版本名称]

  示例：

  docker tag nginx:v1 registry.cn-jssz1.ctyun.cn/cloud-develop/nginx:v1
- 上传镜像至目标镜像仓库。

  docker push [镜像仓库地址]/[组织名称]/[镜像名称:版本名称]

  示例：

  docker push registry.cn-jssz1.ctyun.cn/cloud-develop/nginx:v1
- 终端显示如下信息，表明上传镜像成功。

```
The push refers to repository [registry.cn-jssz1.ctyun.cn/cloud-
develop/nginx:v1]
fbce26647e70: Pushed
fb04ab8effa8: Pushed
8f736d52032f: Pushed
009f1d338b57: Pushed
678bbd796838: Pushed
d1279c519351: Pushed
f68ef921efae: Pushed
v1: digest:
sha256:0cdfc7910db531bfa7726de4c19ec556bc9190aad9bd3de93787e8bce3385f8d size:
1780
```

  返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

3.1.3 使用 image-syncer 迁移镜像至 SWR

场景描述

当我们处理数量较少的镜像迁移任务时，使用命令行迁移就可以解决这个问题。但是实际生产中涉及到成千上百个镜像，几 T 的镜像仓库数据时，迁移过程就变得耗时很是漫长，甚至丢失数据。这时，我们可以使用开源镜像迁移工具 **image-syncer** 来处理这个任务。

操作步骤

步骤 1 下载 image-syncer，解压并运行工具。

以 v1.3.1 版本为例，您也可以选择其他版本。

wget https://github.com/AliyunContainerService/image-syncer/releases/download/v1.3.1/image-syncer-v1.3.1-linux-amd64.tar.gz

tar -zvxf image-syncer-v1.3.1-linux-amd64.tar.gz

步骤 2 创建镜像仓库的认证信息文件 **auth.json**。

image-syncer 支持基于 Docker Registry V2 搭建的 docker 镜像仓库，按格式填写即可。此处以苏州的镜像仓库迁移到广州 4 为示例，

将源仓库及目标仓库认证信息写入，示例如下。

```
{
    "registry.cn-jssz1.ctyun.cn": {
        "username": "cn-jssz1@F1I3Q……",
        "password": "2fd4c869ea0……"
    },
    "registry.cn-gdgz1.ctyun.cn": {
        "username": "cn-gdgz1@4N3FA……",
        "password": "f1c82b57855f9d35……"
    }
}
```

其中 registry.cn-jssz1.ctyun.cn 为镜像仓库地址，username、password 可以在登录命令中获取，获取方法如下：

登录 SWR 控制台，在右上角单击“登录指令”，在弹出的窗口中获取登录指令，如下图所示。

图3-1 登录指令

在上图中，cn-jssz1@4N3FA……为 **username**；

f1c82b57855f9d3564ee……为 **password**；

registry.cn-jssz1.ctyun.cn 为**镜像仓库地址**。

注意

因安全性要求，以上示例中所有 username 和 password 均有部分内容进行省略，请以控制台获取到的实际用户名和密码为准。

步骤 3 创建同步镜像描述文件 **images.json**。

如下示例，左边是源仓库的地址，右边是目的仓库地址。image-syncer 还支持其他描述方式，具体请参见 README-zh_CN.md。

```
{
    "registry.cn-jssz1.ctyun.cn/org-ss/canary-consumer": "registry.cn-gdgz1.ctyun.cn/dev-container/canary-consumer"
}
```

步骤 4 执行如下命令将镜像迁移至 SWR。

./image-syncer --auth=./auth.json --images=./images.json --namespace=dev-container --registry=registry.cn-gdgz1.ctyun.cn --retries=3 --log=./log

表3-2 命令行参数说明

参数	说明
--config	设置用户提供的配置文件路径，使用之前需要创建此文件，默认为当前工作目录下的 config.json 文件。这个参数与 --auth 和--images 的作用相同，分解成两个参数可以更好地区分认证信息与镜像仓库同步规则。建议使用 --auth 和 --images。
--images	设置用户提供的镜像同步规则文件所在路径，使用之前需要创建此文件，默认为当前工作目录下的 images.json 文件。
--auth	设置用户提供的认证文件所在路径，使用之前需要创建此认证文件，默认为当前工作目录下的 auth.json 文件。
--log	打印出来的 log 文件路径，默认打印到标准错误输出，如果将日志打印到文件将不会有命令行输出，此时需要通过 cat 对应的日志文件查看。
--namespace	设置默认的目标 namespace，当配置文件内一条 images 规则的目标仓库为空，并且默认 registry 也不为空时有效，可以通过环境变量 DEFAULT_NAMESPACE 设置，同时传入命令行参数会优先使用命令行参数值。
--proc	并发数，进行镜像同步的并发 goroutine 数量，默认为 5。
--retries	失败同步任务的重试次数，默认为 2，重试会在所有任务都被执行一遍之后开始，并且也会重新尝试对应次数生成失败任务的生成。一些偶尔出现的网络错误比如 io timeout、TLS handshake timeout，都可以通过设置重试次数来减少失败的任务数量。
--registry	设置默认的目标 registry，当配置文件内一条 images 规则的目标仓库为空，并且默认 namespace 也不为空时有效，可以通过环境变量 DEFAULT_REGISTRY 设置，同时传入命令行参数会优先使用命令行参数值。

迁移命令执行后，可登录目标镜像仓库，查看已迁移的镜像。

----结束