容器化开发及两步法快速构建 Docker 镜像

导读：

作为移动互联新时代的程序员，经常会把程序装进容器内运行，但是慢悠悠的镜像构建过程、国际网络的不稳定、移动联网时的流量狂奔，都让人又爱又恨。本文介绍的两步快速构建容器镜像方法，速度快到以秒为单位，能缓解镜像构建痛点。

一、为什么用容器技术

写作本文源于编写一个 SSH 信任登录认证程序，而信任登录认证又是 K8s 故障诊断程序的一部分。

本文的内容也是从一篇长文中摘录出来的，为了适应互联网读者的阅读习惯（碎片时间阅读），做成几个小专题，内容聚焦更聚焦，也许效果会好一点。少食多餐的好，暴饮暴食万一撑爆了，那就是遗憾加罪过了。

信任登录认证程序运行在 Linux/Unix 环境下，如果要适应不同的操作系统版本，需要考虑一些环境细节，或者说为不同的操作系统版本编写不同的代码，至少某些功能点是这样的。随着操作系统版本升级，还要做适应性升级，否则程序运行时可能会发生异常。这是不希望看到的。

容器化技术在程序与宿主机之间引入新的容器层，把程序与容器的关系，变为程序与容器、容器与宿主机两层关系。程序员只需要考虑程序运行时的容器环境，容器与宿主机之间的关系交由类似于 Docker 的容器管理层实现。容器化技术把程序与运行时的宿主机环境隔离开来，这样程序就能适应不同的宿主机操作系统。例如，假如建立信任登录的程序选择 CentOS 作为容器环境，CentOS 容器能运行在 Windows、RedHat Linux、Debian Linux 和 AIX 等操作系统上，则应用程序也就间接地能在这些操作系统上运行。

信任认证程序只是故障诊断系统的一个功能，如果说修改信任认证程序的源代码以适应不同的运行环境，只需要花费少量的时间，那么修改故障诊断程序适应环境，代价就太高昂了。所以容器化是不得不走的一步，晚走不如早一点走。

二、容器技术选择

从早期的 chroot（1979）、FreeBSD Jails（2000）、Linux VServer（2001）、Solaris Containers（2004）和 LXC（2008），到如今的 Warden（2011）、Docker（2013）、Rocket（2014）和 Windows Containers（2016），容器技术经过四十年的发展，已经是遍地开花，得到广泛地应用。

其中，Docker 容器技术挟后发优势，因为其功能强大、性能优良、开源免费（社区版）、广泛适应性（所有主流操作系统），而受到业界广泛推崇。作者在 Docker 基础上做过大量的开发和应用，所以优先选择 Docker 容器技术。

三、基础镜像选择

选好容器技术后，就要选择容器内运行的操作系统。RedHat Linux 是经典的 Linux 发行版，在国内拥有广泛的用户基础。但是因为版本注册和收费等原因，Redhat 并不适合用作容器操作系统。CentOS 是 Redhat 的社区开源免费版，与 Redhat 有相似的命令集和用户操作习惯。Alpine Linux 是面向安全的轻量级 Linux 发行版，主要用于面向 Serverless 服务而无需用户操作的容器操作系统。

本程序和其所属的故障诊断程序需要用户管理维护，所以既需要一个程序运行环境，也需要用户操作环境，CentOS 恰好能满足这两点需求。CentOS 的版本选择比较新的 CentOS 7.5，开源社区已经有好心人把 CentOS 操作系统构建成公共镜像，从网上拉取对应版本的镜像即可，开箱即用。

基础容器镜像包含操作系统最核心的版本，如果需要更多软件包，例如 SSH 客户端和 SSH 服务器，则需要安装附加软件包，本程序也需要安装进去。这些软件包的安装指令在 Dockerfile 文件中描述。

四、容器驻留程序

在运行时，Docker 从容器镜像启动容器实例后，会自动启动入口点 Entry Point 或者 CMD 声明的程序，入口点程序执行结束退出，容器也就退出了。我们希望容器始终在运行，直到用户主动停止容器。因为诊断程序可能通过 crontab 调度定时运行，也可能在需要时手工启

动运行，但是不会始终在运行，缺少运行状态的进程会引起容器退出。

引入 supervisord 程序，解决容器缺少驻留服务进程问题。supervisord 程序是一个运行在容器内的驻留服务进程，对外开放 Web 管理端口，通过浏览器能访问查询、管理容器内进程等资源。

supervisord 的配置文件 supervisord.conf 的内容如下：

```
## supervisord.conf



[unix_http_server]

file = /supervisord/supervisor.sock



[inet_http_server]

port = *:9001

username = admin

password = adminpass



[supervisord]

logfile = /supervisord/logs/supervisord.log

logfile_maxbytes = 50MB

logfile_backups = 20

loglevel = info

pidfile = /supervisord/supervisord.pid

nodaemon = false
```

```
minfds = 1024

minprocs = 200



[supervisorctl]

serverurl=unix://supervisord/supervisor.sock



[include]

files = /supervisord/include/*.ini
```

五、容器镜像构建

容器镜像的构建过程包含以下步骤：声明基础镜像，这里是 CentOS 7.5；安装附加软件包；编译源代码，这里是 Shell 脚本，不需要编译，这一步忽略；复制 Shell 脚本、上一步编译好的程序文件和其他资源文件到容器内的相应目录下；声明容器启动的入口点程序（Entry Point）；其他操作。

容器镜像的构建过程中，拉取基础镜像、安装附加软件包的比较耗时，因为基础镜像比较大，大约几百兆字节，附加软件包也比较大，大部分从互联网下载（下载过程自动完成），甚至从国外站点下载，受到接入带宽和提供镜像服务站点带宽的约束，所以构建过程的这部分是比较慢的。复制 Shell 脚本在构建环境本地进行，文件比较小，所以非常快。

在开发过程中，因为发布新版本程序，镜像构建过程经常发生，如果每次花费几分钟、十几分钟等待构建完成，这是漫长的煎熬，尤其是对追求极致的架构师，是无法忍受、无法接受的。

如何缩短日常开发过程中的镜像构建时间，是一个亟待解决的问题。前面分析了镜像构建过程，拉基础镜像、安装附加软件包这些步骤耗时很长，而且这些步骤执行的结果每次都是一样的。所以让这些耗时且每次输出结果无变化的步骤只在第一次构建时进行，而在第二

次、第三次...的构建过程不执行这几步，只执行有变化的步骤，将大大缩短镜像构建时间。

六、两步法快速构建镜像

具体实现是将整个构建过程拆分为两步进行，第一步预处理拉基础镜像、安装附件软件包，第二步编译、复制应用程序。因为有两个步骤，称作两步法构建镜像。第一步的输出结果是中间镜像。

Dockerfile 是顺序执行的过程框架，但是逻辑控制能力比较弱。Makefile 能执行编译、docker build 等指令，而且支持 if..else 等控制语句。所以考虑用 Makefile 控制镜像构建过程。

两步法包含一个 Makefile 文件和两个 Dockerfile 文件，也可以称作 1M2D 方法。

先画一张两步法构建容器镜像的时序图，如下图所示：

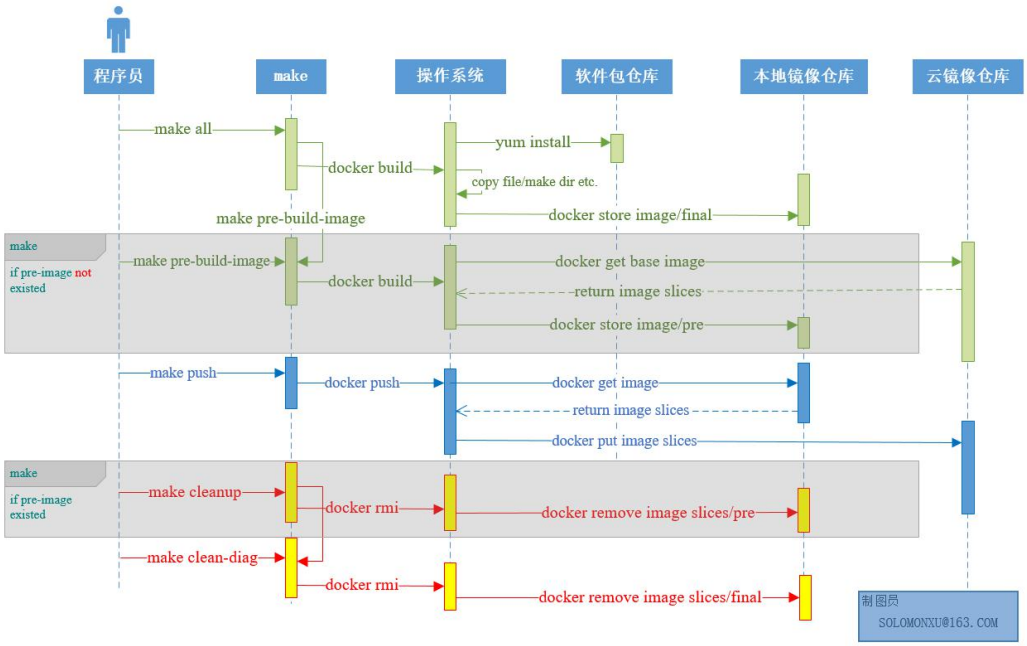

图 两步法构建容器镜像的时序图

Makefile 包含以下阶段：

1、all: pre-build-image

阶段 all 是 Makefile 的总入口，依赖于预构建（中间）镜像，会先调用 pre-build-image 阶段。

2、pre-build-image:

阶段 pre-build-image 构建（中间）镜像阶段。判断中间镜像在本地 Docker 镜像仓库是否存在。如果中间镜像存在，则跳过构建中间镜像阶段。这是两步法的关键点。

3、cleanup: cleanup-diag:

阶段 cleanup 删除中间镜像，只有在中间镜像依赖基础镜像或者附件软件包发生变化时才会执行，这种情况很少发生。

依赖于阶段 cleanup-diag，会预先执行所依赖的阶段，删除最终的应用镜像。

4、cleanup-diag:

阶段 cleanup-diag 删除最终的应用镜像。本阶段没有依赖阶段，可以单独执行，删除最终的应用镜像。

5、push:

阶段 push 推送最终镜像到云端镜像仓库。本阶段没有依赖阶段。

Makefile 文件的内容如下：

```
## Define constants
VERSION=latest
TAG_PRE_BUILD_IMAGE=k8s-diag-pre
K8S_DIAG_IMAGE=k8s-diagnose
#FULL_URL_IMAGE=registry.cn-hangzhou.aliyuncs.com/wise2c-dev/$(K8S_DIAG_IMAGE):$(VERSION)
FULL_URL_IMAGE=registry.cn-hangzhou.aliyuncs.com/solomonxu/$(K8S_DIAG_IMAGE):$(VERSION)


## Check docker image pre-build-image if existed
pre-image-count := $(shell docker images | grep $(TAG_PRE_BUILD_IMAGE) | wc -l)
pre-image-id       := $(shell docker images | grep $(TAG_PRE_BUILD_IMAGE) | grep centos | awk '{print $$3}')
diag-image-id      := $(shell docker images | grep $(K8S_DIAG_IMAGE) | grep $(VERSION) | awk
```

```
'{print $$3}')


## .PHONY
.PHONY: all image-pre


## Build all
all: pre-build-image
#	echo "This will make Docker image $(K8S_DIAG_IMAGE) ..."
	docker build -f Dockerfile -t $(K8S_DIAG_IMAGE):$(VERSION) .


## Build pre-build-image
pre-build-image:
	@if [ "$(pre-image-count)" = "0" ]; then \
		docker build -f Dockerfile.pre -t centos:$(TAG_PRE_BUILD_IMAGE) . ; \
	else \
		echo ""; \
	fi


## Cleanup images
cleanup: cleanup-diag
	@if [ -n "$(pre-image-id)" ]; then \
		echo "This will remove Docker image centos:$(TAG_PRE_BUILD_IMAGE) now, IMAGE ID: $(pre-image-id)."; \
		docker rmi $(pre-image-id) ; \
	fi


## Cleanup image k8s-diagnose
cleanup-diag:
	@if [ -n "$(diag-image-id)" ]; then \
		echo "This will remove Docker image $(K8S_DIAG_IMAGE) now, IMAGE ID:
```

```
$(diag-image-id)."; \
        docker rmi $(diag-image-id) ; \
    fi

## Push to repository
push:
    docker tag $(K8S_DIAG_IMAGE):$(VERSION) $(FULL_URL_IMAGE)
    docker push $(FULL_URL_IMAGE)
```

第一步预处理阶段拉基础镜像、安装软件包，Dockerfile.pre 文件内容如下：

```
FROM docker.io/centos:7.5.1804

MAINTAINER solomonxu<solomonxu@163.com>

## Update CentOS
RUN yum -y update

## Install supervisor etc.
RUN yum install -y openssh-clients openssh-server sshpass
RUN yum install -y net-tools
RUN yum install -y epel-release
RUN yum install -y supervisor
RUN yum install -y which mailx expect
```

第二步复制应用 shell 脚本、暴露端口、声明容器启动入口点，Dockerfile 文件内容如下：

```
FROM centos:k8s-diag-pre

MAINTAINER solomonxu<solomonxu@163.com>
#ARG gitCommit
```

```
## Copy file for k8s-diagnose
ADD ./bin /k8s-diagnose/bin
ADD ./conf /k8s-diagnose/conf

## Add permissions of shell
RUN chmod -R a+x /k8s-diagnose/bin/

## Make dirs for supervisord
RUN mkdir -p /supervisord
RUN mkdir -p /supervisord/include
RUN mkdir -p /supervisord/logs

## Add config for supervisord
ADD ./docker/supervisord.conf /supervisord

USER root
EXPOSE 22 80 9001

## COMMAND
CMD [ "supervisord", "-c", "/supervisord/supervisord.conf" ]
```

七、构建镜像测试

1、第一次构建镜像

第一次构建镜像时，本地已有基础镜像 centos:7.5.1804，省略了从云镜像仓库拉取的时间。但是仍需要执行更新 update、安装软件包等步骤。

第一次构建镜像比较顺利，耗时 2 分 21 秒。有几次安装软件包遇到国内软件仓库不可用，连接国外软件仓库网络不好，第一次构建花了十几分钟。

```
[root@dev-10 k8s-diagnose]# date; make; date
Thu Aug 29 22:30:44 CST 2019
Sending build context to Docker daemon  212.4MB
Step 1/8 : FROM docker.io/centos:7.5.1804
 ---> cf49811e3cdb
Step 2/8 : MAINTAINER solomonxu<solomonxu@163.com>
 ---> Running in 9a4511e239e5
...
Step 12/12 : CMD [ "supervisord", "-c", "/supervisord/supervisord.conf" ]
 ---> Running in f5caa70dde50
Removing intermediate container f5caa70dde50
 ---> ccf57e4cf9a4
Successfully built ccf57e4cf9a4
Successfully tagged k8s-diagnose:latest
Thu Aug 29 22:33:05 CST 2019
```

2、第二次构建镜像

第二次构建从中间镜像开始，不需要安装软件包等，只需要复制 bin、conf 等几个子目录下的文件到镜像内，耗时很短。

第二次构建耗时 2 秒左右。测试过几次，有时相差 1 秒，有时相差 2 秒。

```
[root@dev-10 k8s-diagnose]# date; make; date
Thu Aug 29 22:34:26 CST 2019
docker build -f Dockerfile -t k8s-diagnose:latest .
Sending build context to Docker daemon  212.4MB
Step 1/12 : FROM centos:k8s-diag-pre
 ---> 9278f7715546
Step 2/12 : MAINTAINER solomonxu<solomonxu@163.com>
 ---> Using cache
```

```
 ---> f972deed6ace
Step 3/12 : ADD ./bin /k8s-diagnose/bin
 ---> Using cache
 ---> bd6e94808793
Step 4/12 : ADD ./conf /k8s-diagnose/conf
 ---> Using cache
 ---> 9aaf55dbc034
Step 5/12 : RUN chmod -R a+x /k8s-diagnose/bin/
 ---> Using cache
 ---> 4ae828d62733
Step 6/12 : RUN mkdir -p /supervisord
 ---> Using cache
 ---> c5b32d52c423
Step 7/12 : RUN mkdir -p /supervisord/include
 ---> Using cache
 ---> e971140bd63b
Step 8/12 : RUN mkdir -p /supervisord/logs
 ---> Using cache
 ---> 4e1a4caa0148
Step 9/12 : ADD ./docker/supervisord.conf /supervisord
 ---> Using cache
 ---> 7b6b3419e487
Step 10/12 : USER root
 ---> Using cache
 ---> 5e97076ee31f
Step 11/12 : EXPOSE 22 80 9001
 ---> Using cache
 ---> e622b241366d
Step 12/12 : CMD [ "supervisord", "-c", "/supervisord/supervisord.conf" ]
```

```
 ---> Using cache
 ---> ccf57e4cf9a4
Successfully built ccf57e4cf9a4
Successfully tagged k8s-diagnose:latest
Thu Aug 29 22:34:28 CST 2019
```

3、删除最终镜像

删除最终镜像，保留中间镜像。

```
[root@dev-10 k8s-diagnose]# make cleanup-diag
This will remove Docker image k8s-diagnose now, IMAGE ID: ccf57e4cf9a4.
Untagged: k8s-diagnose:latest
Deleted: sha256:ccf57e4cf9a4ebf3157a27b106ea5bde50f72949d4b69de202c88efa9b541bac
...
```

4、第三次构建镜像

第三次构建与第二次构建类似，因为保留了中间镜像，速度也是很快的。第二次构件时时间不够精确，第三次把时间精确到纳秒，观察精确的构建时间。

第三次构建耗时 1.779 秒。测试了三遍，时间在 1.772 秒到 1.779 秒之间变化，变化范围很小。

```
[root@dev-10 k8s-diagnose]# date "+%Y-%m-%d %H:%M:%S.%N"; make; date "+%Y-%m-%d %H:%M:%S.%N";
2019-08-29 23:01:51.802823460
docker build -f Dockerfile -t k8s-diagnose:latest .
Sending build context to Docker daemon  212.4MB
Step 1/12 : FROM centos:k8s-diag-pre
 ---> 9278f7715546
...
Successfully built f6a104888e0d
Successfully tagged k8s-diagnose:latest
```

```
2019-08-29 23:01:53.581900905
```

两步法将本程序的最终镜像构建时间从 2 分 21 秒缩短到 1.779 秒。说把时间缩短到原来的 1/80，可能意义不是很典型。因为第一次构建时间严重依赖于网络，时间变化幅度大，第二次及以后的构建时间非常稳定，但是也会随着复制程序文件的数量和大小有小幅度变化。尽管如此，可以看得出两步法压缩镜像构建时间效果明显。

5、推送镜像到云镜像仓库

推送已构建的最终镜像到云镜像仓库，读者可以自行修改仓库地址。

make push

6、删除所有的构建镜像

make cleanup

八、从镜像启动容器

从前述过程构建的容器镜像启动一个新容器，然后登录到容器内部，查询容器内运行的进程。docker-compose 执行当前目录下的 docker-compose.yaml 文件，根据 yaml 文件指引，启动容器运行。

```
[root@dev-10 k8s-diagnose]# docker-compose up &
[1] 26526
[root@dev-10 k8s-diagnose]# Creating network "k8sdiagnose_default" with the default driver
Creating k8sdiagnose_supervisor_1 ... done
Attaching to k8sdiagnose_supervisor_1
[root@dev-10 k8s-diagnose]# docker exec -it k8sdiagnose_supervisor_1 bash
[root@30ffac589d2f /]#
[root@30ffac589d2f /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 15:16 ?        00:00:00 /usr/bin/bash -c supervisord -c
```

```
/supervisord/supervisord.conf while true; do sleep 100; done
root         7     1  0 15:16 ?        00:00:00 /usr/bin/python /usr/bin/supervisord -c
/supervisord/supervisord.conf
root         8     1  0 15:16 ?        00:00:00 sleep 100
root         9     0  0 15:16 pts/0    00:00:00 bash
root        22     9  0 15:16 pts/0    00:00:00 ps -ef
```

九、工作小结

本文把镜像构建拆分为两步：构建预构建镜像，构建最终镜像，把 Shell 脚本程序的镜像构建时间从 2~10 分钟，缩短到 2 秒以内，极大地方便了快速应用开发 RAD 和快速版本迭代，适于流行的敏捷开发方法。

对于经常在移动环境办公的读者，先在 WiFi 环境构建好预镜像，在预构建阶段下载完所有软件包。这样在移动联网时，构建最终镜像就无需担心移动流量消耗，因为工作都在本地进行（下载源码除外），不会产生移动流量。即使需要推送镜像到云镜像仓库也不是大问题，云镜像仓库会把镜像分解为若干个分片，推送镜像时只推送变化的分片，也就是程序员开发的部分，而这些体积是很小的。

十、下一步工作

本文介绍的容器镜像构建方法基于 Shell 脚本，在实际开发工作中，经常遇到的是 Java、Golang、Python 等语言编写的应用程序。高级语言程序编写完源代码之后，需要先编译成可执行的二进制的可行性文件，才能启动可执行程序运行。

为了将编译环境与宿主机环境隔离，程序员经常采用容器化编译：先构建一个编译时容器，然后在容器内编译源代码。为了精简运行时环境的体积，有些编译时需要的软件包，在运行时容器内不会出现，例如：Golang 的第三方源码包、Go 编译器，Java 的开发工具包 JDK，就不应该出现在运行时容器。所以，编译时容器与运行时容器不宜共用。

构建编译时容器镜像也可以拆分为两步，以加速编译过程。这样，构建编译时镜像分两

步走，构建运行时镜像也分两步，一共四个步骤，可以快速完成源码下载、源码编译、运行时容器镜像构建。

将来的小目标是：1 分钟内完成小型 Golang 程序镜像构建，2~5 分钟内完成中大型 Golang 程序构建；30 秒内完成小型 Java 程序镜像构建，1~2 分钟内完成中大型 Java 程序构建。

十一、源码下载

源代码托管在 github.com 源码仓库，源代码随时可能会更新。

用下面的命令可以直接克隆源码：

git clone https://github.com/solomonxu/docker-fast-build-shell.git

Cantact to the author:

Email: xumeng@wise2c.com

Wechat: solomonxu9999