Journal of System Simulation

Volume 32 | Issue 11 Article 16

11-17-2020

# Container-based Automatic Packaging Technology for Complex System Simulation Application

Wang Shuai
*College of Systems Engineering,National University of Defense Technology,Changsha 410073,China;*

Zhu Feng
*College of Systems Engineering,National University of Defense Technology,Changsha 410073,China;*

Yiping Yao
*College of Systems Engineering,National University of Defense Technology,Changsha 410073,China;*

Wenjie Tang
*College of Systems Engineering,National University of Defense Technology,Changsha 410073,China;*

*See next page for additional authors*

# Container-based Automatic Packaging Technology for Complex System Simulation Application


## Abstract

Abstract: Container-based technology provides a new solution for the rapid and flexible deployment of complex system simulation applications. Container supports service-based packaging of simulation applications, which greatly reduces the difficulty of deploying simulation applications. Current packaging technology mainly relies on manually writing Dockerfile, which results in low packaging efficiency and human errors. *A container-based automatic packaging technology for complex system simulation application is proposed, and the reusable library component template is defined. Combined image template is generated by combining simulation application and library component templates. Dockerfile is generated by the combined template after syntax optimization and error detection.* The experiments for complex system simulation application based on SUPE simulation engine show the effectiveness of the proposed packaging technology.



## Keywords

container, complex system simulation, package, Dockerfile



## Authors

Wang Shuai, Zhu Feng, Yiping Yao, Wenjie Tang, and Yuhao Xiao



## Recommended Citation

Wang Shuai, Zhu Feng, Yao Yiping, Tang Wenjie, Xiao Yuhao. Container-based Automatic Packaging Technology for Complex System Simulation Application[J]. Journal of System Simulation, 2020, 32(11): 2199-2207.

第 32 卷第 11 期 系统仿真学报© Vol. 32 No. 11
2020 年 11 月 Journal of System Simulation Nov., 2020

# 基于容器的复杂系统仿真应用自动封装技术

王帅，朱峰*，姚益平，唐文杰，肖雨豪
(国防科技大学系统工程学院，湖南 长沙 410073)

**摘要：**容器技术为复杂系统仿真应用的快速灵活部署提供了一种全新的解决方案。容器支持对仿真应用的服务化封装，可以大大降低仿真应用在不同运行环境上部署的难度。目前主流的容器镜像封装技术主要依赖手动编写 Dockerfile 文件，导致封装效率低且容易引入人为错误。*因此，提出了一种基于容器的复杂系统仿真应用自动封装技术，定义了支持重用的库组件模板，并将仿真应用与库组件模板融合生成组合镜像模板，组合镜像模板在通过语法优化与错误检测后自动生成 Dockerfile。*基于 SUPE 仿真引擎的复杂系统仿真应用封装实验证明了该自动封装技术的有效性。

**关键词：**容器；复杂系统仿真；封装；Dockerfile
中图分类号：TP391.9 文献标识码：A 文章编号：1004-731X (2020) 11-2199-09
DOI: 10.16182/j.issn1004731x.joss.20-FZ0423

## Container-based Automatic Packaging Technology for Complex System Simulation Application

*Wang Shuai, Zhu Feng*, Yao Yiping, Tang Wenjie, Xiao Yuhao*
(College of Systems Engineering, National University of Defense Technology, Changsha 410073, China)

**Abstract:** Container-based technology provides a new solution for the rapid and flexible deployment of complex system simulation applications. Container supports service-based packaging of simulation applications, which greatly reduces the difficulty of deploying simulation applications. Current packaging technology mainly relies on manually writing Dockerfile, which results in low packaging efficiency and human errors. *A container-based automatic packaging technology for complex system simulation application is proposed, and the reusable library component template is defined. Combined image template is generated by combining simulation application and library component templates. Dockerfile is generated by the combined template after syntax optimization and error detection.* The experiments for complex system simulation application based on SUPE simulation engine show the effectiveness of the proposed packaging technology.

**Keywords:** container; complex system simulation; package; Dockerfile

## 引言

复杂系统仿真是研究复杂系统最有效的手段之一，并在社会、国防、生物等领域得到广泛运用，如大规模复杂社会仿真，复杂战场环境下作战仿真，生物分子仿真等[1]。随着复杂系统仿真应用的不断发展，仿真实体的规模越来越大，实体之间交互也越来越复杂，不同种类仿真实体的构建往往需要不同领域专业软件的支持，构建应用时需要加载链接不同的库文件[2]，如仿真引擎库 SUPE[3]、消

收稿日期：2020-06-29 修回日期：2020-07-20;
基金项目：国家自然科学基金(61903368)；
作者简介：王帅(1997-)，男，湖北天门，硕士生，研究方向为系统仿真；朱峰(1985-)，男，湖北随州，博士，讲师，研究方向为高性能仿真；姚益平(1963-)，男，湖南邵阳，博士，研究员，研究方向为高性能仿真。



 

息通信库 MPI[4]、运行支撑库 Boost[5]等，导致复杂仿真应用在不同运行环境中部署难度越来越大。云计算支持对仿真应用的服务化封装，为仿真应用的开发与部署提供了一种新的平台架构[6]。为了提高云环境中应用程序开发部署的弹性，以 Docker 为代表的虚拟容器技术应运而生。尤其与基于 Hypervisor 虚拟化相比，基于容器虚拟化的 Docker 更加轻量且可扩展[7]。Docker 可以封装仿真应用以及所依赖的库文件到一个镜像中，用户可以通过镜像生成容器发布到任何 Linux 机器上，实现“一次构建，多次使用”，从而大大降低了仿真应用在不同运行环境上部署的复杂度，提高了仿真应用的封装效率[8]。

在基于容器虚拟化中，目前主流的仿真应用封装方法是通过编写 Dockerfile 文件构建镜像[9-10]，然而，Dockerfile 命令编写涉及较复杂的语法，且手动编写 Dockerfile 文件容易引入人为错误，导致仿真应用封装门槛高、难度较大，失败率较高。另一方面，复杂系统仿真应用的开发集成“永远在路上”[11]，需要根据应用需求不断地加载新的依赖库，这就需要重复多次编写 Dockerfile 文件，增大了仿真应用封装的工作量。因此，如何针对 Dockerfile 的语法进行优化以及实现库组件重用，对于提高复杂系统仿真应用封装效率，降低封装失败率具有重要意义[12]。

为了解决上述问题，设计了一种基于容器的复杂系统仿真应用的自动封装框架，该框架描述了仿真应用与依赖库组合封装的流程。提出了基于 XML 的库组件模板用来描述仿真依赖库组件资源，实现了库组件重用，在仿真应用所依赖库发生改变时，只需改变相应的库组件模板，无需重写 Dockerfile 文件，提高了复杂系统仿真应用的封装效率。优化了 Dockerfile 语法，对 Dockerfile 中的常见的非语法错误进行检测，提高了构建仿真应用镜像的性能，降低了复杂系统仿真应用的封装失败率。

## 1 基于容器的复杂系统仿真应用的自动封装框架

基于容器的复杂系统仿真应用的自动封装框架如图 1 所示，用户首先确定复杂系统仿真应用所需依赖库，将依赖库组件模板与仿真应用生成组合模板，根据组合模板生成对应的 Dockerfile，随后对 Dockerfile 语法进行优化，检测 Dockerfile 是否存在错误，随后根据 Dockerfile 构建 Docker 镜像，Docker 容器就是从镜像中启动的。Docker 镜像是以分层的形式创建的，底层是一个引导文件系统，即 Bootfs，其中包括 Bootloader 以及内核。在 Bootfs 上承载着一个只读的 rootfs 层，包括基础镜像层以及只读层，用户可以在基础镜像上挂载多个只读层，包含了容器启动所需的文件以及目录，容器启动后，Docker 守护进程会在只读层上构造一个读写镜像，以实现对文件系统的读写操作。

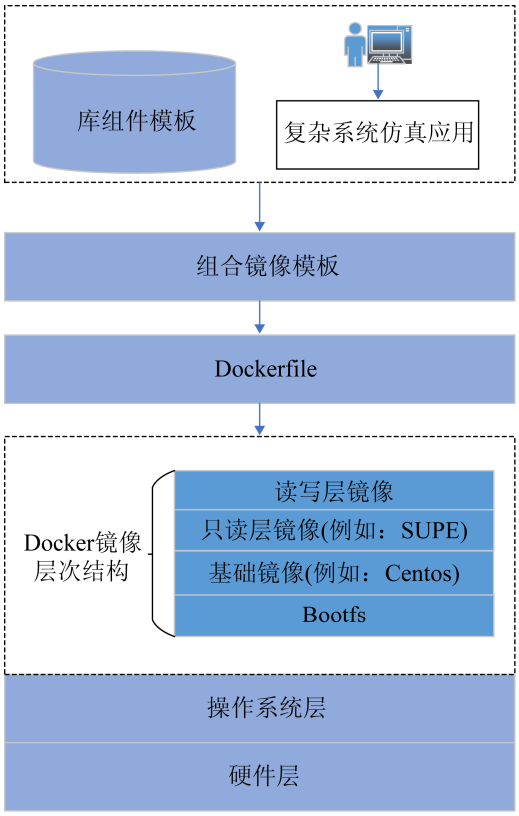


图 1 基于容器的复杂系统仿真应用的自动封装框架
Fig. 1 Container-based automatic packaging framework for complex system simulation applications




DOI: 10.16182/j.issn1004731x.joss.20-FZ0423

## 2 基于容器的复杂系统仿真应用自动封装方法

基于容器的自动封装方法流程图如图 2 所示。首先定义复杂系统仿真应用所需库组件的模板，将仿真应用与库组件模板组合生成组合镜像模板，此时，会进行基本的语法检测，例如：模板是否符合 XML 规范，不可缺省项是否填写等。如果语法存在错误，则需要重新编写库组件模板和组合镜像模板。确认无误后，会根据组合镜像模板生成 Dockerfile，随后会对 Dockerfile 的语法进行优化，最后使用优化后的 Dockerfile 在宿主机上生成镜像，用户即可通过生成的镜像启动容器发布到任何 Linux 机器上，实现仿真应用的自动封装。

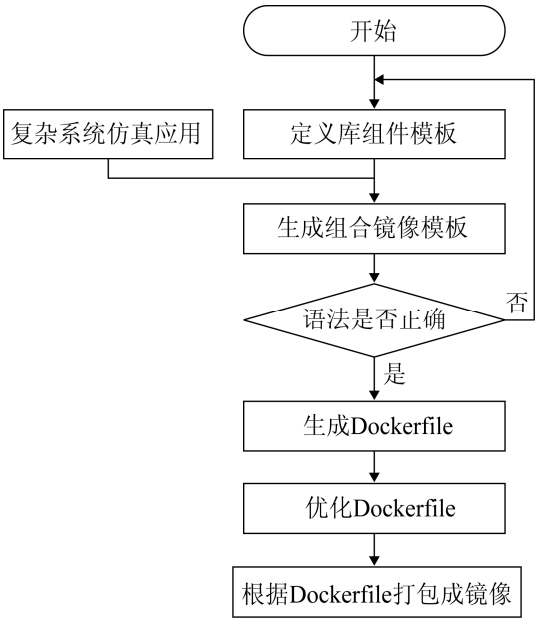


图 2 基于容器的复杂系统仿真应用封装方法流程图
Fig. 2 Flowchart of container-based automatic packaging method for complex system simulation applications

### 2.1 库组件模板

为了实现库组件重用，最重要的是提供模板来描述库组件，面向重用的库组件模板无需或只需少量更改即可应用在新的仿真应用中，避免了重复的底层开发，提高了仿真应用的封装效率。可扩展标记语言 XML 为描述和存储资源信息结构化数据提供了一个解决方案，采用 XML 描述库组件时，其结构化数据一方面可以很好的描述库组件的特征，另一方面便于用户更好的检索所需的库组件，从而有助于提高仿真应用封装效率。因此本文提出了一种基于 XML 的库组件模板。针对 Dockerfile 语法特点，单一库组件模板定义如图 3 所示，其中一个 component 标签表示定义一个库组件模板，在 component 下各标签意义如下：

Name：定义库组件的名字，该标签对于库组件是唯一且不可缺省的，例如：<name>Supe</name>。

from：指定该库组件的基础镜像，例如 <from>centos:7</from>。

maintainer：指定库组件的维护者信息，例如：<maintainer>John-John@nudt.edu.cn</maintainer>。

content：定义库组件的主要内容，改标签包括 4 种子标签(run，add，workdir，env)，库组件会根据需要灵活改变 4 种子标签的数量和顺序。run 标签指定构建镜像时需要运行的语句，例如：<run>apt-get update</run>。add 标签将宿主机上的文件或目录拷贝到镜像中，例如：<add>src=”/etc/docker/” dest=”/mnt/”</add>, 其中 src 表示宿主机的文件或目录路径，dest 表示指定镜像文件系统中的目录路径。workdir 标签设置当前工作目录，如果存在多个 workdir 标签，则最后一个 workdir 作为当前工作目录。env 标签定义容器文件系统的环境变量，例如：<env>LD_LIBRARY_PATH=/supe/lib/TCP</env>。

expose：声明库组件生成容器运行时暴露的服务端口。例如：<expose> 8080</expose>。

component-cmd：指定容器运行的命令，例如：<component-cmd>”ls”,”-l”</component-cmd>，第一个值为执行的命令，逗号后面的值为命令的参数。

component-entrypoint 与 component-cmd 类似，两者可以一起使用，当指定了 component-entrypoint 的命令时，component-cmd 指定的则是命令的参数。

定义完单一组件模板后，用户可根据仿真应用的需求选择库组件，然后将库组件以及其他仿真资

源生成组合镜像模板，组合模板定义如图 4 所示。在 combined-image 下各标签意义如下：

name：组合镜像的名称，该标签是唯一，不可缺省的，由用户定义。

from：组合镜像的基础镜像，所有的组件必须基于同一个基础镜像。

component：仿真应用需要库组件名称，该名称作为库组件的唯一的标识。可添加多个库组件。

contents：与单一库组件模板的 content 标签类似，组合镜像会根据所需库组件名称获取各库组件模板中的 content 内容，并组合在 contents 标签下。

expose：组合镜像会获取各库组件模板所暴露端口，组合在 expose 标签下。

组合镜像模板中的 combined-image-cmd 与 combined-image-entrypoint 类似。容器启动时只能有一个初始命令，因此 combined-image-cmd 与 combined-image-entrypoint 默认继承最后一个库组件的 component-cmd 与 component-entrypoint 标签。

```
<component>
    <name></name>//库组件名称
    <from></from>//基础镜像
    <maintainer></maintainer >//库组件维护者信息
    <content>
        <run></run>//构建镜像时运行命令
        <add></add>//拷贝文件或目录到镜像中
        <workdir></workdir>//设置工作路径
        <env></env>设置环境变量
    </content>
    <expose> </expose>//声明库组件生成容器运行时暴露的服务端口
    <component-cmd></component-cmd>//指定容器默认的启动命令
    <component-entrypoint></component-entrypoint>//指定容器默认执行的任务
</component>
```

图 3 单一库组件模板
Fig. 3 Single library component template

```
<combined-image>
    <name></name>//组合镜像名称
    <from></from>//基础镜像
    <maintainer></ maintainer >//组合镜像维护者信息
    <component></component>//需要的库组件名称
    <contents>
        <run></run>//构建镜像时运行命令
        <add></add>//拷贝文件或目录到镜像中
        <workdir></workdir>//设置工作路径
        <env></env>设置环境变量
    </contents>组合镜像的内容
    <expose> </expose>//声明组合镜像生成容器运行时暴露服务端口
    <combined-image-cmd></combined-image-cmd>//指定容器默认的启动命令
    <combined-image-entrypoint></combined-image-entrypoint>//指定容器默认执行的任务
</combined-image>
```

图 4 组合镜像模板
Fig. 4 Combined image template




DOI: 10.16182/j.issn1004731x.joss.20-FZ0423

### 2.2 Dockerfile 文件生成

在生成组合镜像模板后，会根据组合镜像模板的内容自动生成 Dockerfile 文件，为了更好的适配 Dockerfile 语法，组合镜像模板的标签与 Dockerfile 的命令一一对应，其对应关系如表 1 所示。

表 1 组合镜像模板与 Dockerfile 命令对应关系
Tab. 1 Correspondence between combined image template and Dockerfile command

| 组合镜像模板标签 | Dockerfile 命令 | 备注 |
|---|---|---|
| from | FROM | 基础镜像 |
| env | ENV | 设置环境变量 |
| add | ADD | 将文件或目录从源地址复制到目标地址 |
| run | RUN | 指定镜像运行命令 |
| workdir | WORKDIR | 设置工作目录 |
| expose | EXPOSE | 容器运行时暴露的服务端口 |
| combined-image-cmd | CMD | 指定构建容器初始命令 |
| combined-image-entrypoint | ENTRYPOINT | 指定容器默认执行的任务 |

### 2.3 Dockerfile 语法优化与错误检测

Dockerfile 的复杂性要求用户深入了解 Dockerfile 的语法，带来了很高的学习成本，用户使用 Dockerfile 生成的镜像可能会存在构建时间长，占用空间大的问题。并且 Dockerfile 并没有对构建镜像中可能存在的错误进行检测，导致构建镜像失败率较高。因此本节主要通过分析 Dockerfile 语言的特点进行语法优化与错误检测，提高仿真应用封装效率。

Dockerfile 语法优化与错误检测规则如下：

(1) 将多条 Dockerfile 命令合并为一条。Dockerfile 以基础镜像为基础构建镜像。Dockerfile 命令实际上是描述如何在基础镜像上构建新的镜像，且每条命令都会构建一层新的镜像，因此，应该将连贯的命令(例如，RUN，ADD 等)合并以减少生成镜像的层数。

(2) Dockerfile 中 CMD 与 ENTRYPOINT 命令有 2 种格式，exec 和 shell 格式。当使用 shell 格式时，容器启动后会首先调用 shell 命令，即自动在指定命令前加/bin/sh-c，当组合使用 CMD 与 ENTRYPOINT 命令时，CMD 指令会作为 ENTRYPOINT 指令的参数，此时/bin/sh -c 也会作为 ENTRYPOINT 的参数，导致未知的错误，因此需要统一 CMD 与 ENTRYPOINT 命令格式为 exec。

(3) 指定基础镜像标签。如果 Dockerfile 中没有说明具体的基础镜像标签，则构建镜像时将会继承 latest 版本的基础镜像(例如，From centos 指令等同于 From centos：latest)，当基础镜像更新时，构建镜像可能会出现版本问题，导致构建镜像失败。因此必须指定基础镜像标签。

(4) 在编写 Dockerfile 中，引用的文件和目录不存在是常见非语法错误的一种，因此，该模块会检测 Dockerfile 文件中引用的文件或目录路径是否存在。

## 3 实验与分析

为了更好的描述本文提出的复杂系统仿真应用的自动封装技术的有效性，本节将本文提出的技术分别对 Phold[13]基准与 Social Opinion System (SOS)[14]封装引擎库 SUPE。以及验证 Dockerfile 语法优化与错误检测的有效性。

### 3.1 实验环境

我们将实验在 Intel(R) Core(TM) i5-7500，16 GB 内存，NVIDIA GeForce GTX1050 Ti 显卡以及操作系统为 Centos7 的电脑上进行验证，Docker 版本为 1.3.1。

### 3.2 仿真引擎库 SUPE 封装

其封装步骤如下所示：

step 1：根据单一库组件模板规则建立 SUPE 引擎库组件模板。SUPE 引擎库组件模板如图 5 所示。

step 2：将 step 1 中的 SUPE 引擎库组件模板分别与 Phold 基准以及 SOS 结合为组合镜像模板，

Phold 与 SOS 组合镜像模板分别如图 6~7 所示。

step 3：将 step 2 完成的 Phold 组合镜像模板与 SOS 组合镜像模板根据组合镜像模板与 Dockerfile 命令对应关系解析后生成 Dockerfile 文件，并根据 2.3 节中的规则对 Dockerfile 进行语法优化与错误检测。Phold 与 SOS 组合镜像模板经过语法优化以及错误检测后生成的 Dockerfile 文件分别如图 8~9 所示。

```
<component>
    <name>SUPE</name>
    <from>centos:7</from>
    <maintainer>shuai wang-wangshuai18@bit.edu.cn</ maintainer >
    <content>
        <add>src=”yh-supe.tar.gz” dest=”/mnt/yh-supe/” </add>
    <run> yum -y update</run>
    <run>yum install -y tar</run>
    <run> yum install -y wget</run>
    <run>yum clean all</run>
        <workdir>/home</workdir>
        <env>LD_LIBRARY_PATH = /mnt/yh-supe//lib/TCP</env>
    </content>
    <expose> 8080 </expose>
    <component-cmd></component-cmd>
    <component-entrypoint></component-entrypoint>
</component>
```

图 5 SUPE 组件模板
Fig. 5 SUPE component template

```
<combined-image>
    <name>SUPE Phold</name>
    <from> centos:7</from>
    <maintainer> shuai wang-wangshuai18@bit.edu.cn </ maintainer >
    <component> SUPE </component>
    <contents>
        <add>src=”yh-supe.tar.gz” dest=”/mnt/yh-supe/” </add>
        <add>src=”PHOLD.tar.gz” dest=”/mnt/yh-supe/” </add>
        <run>yum -y update</run>
        <run>yum install -y tar</run>
        <run>yum clean all</run>
        <workdir>/home</workdir>
        <env>LD_LIBRARY_PATH = /mnt/yh-supe/lib/TCP</env>
    </contents>
    <expose>8080</expose>
    <combined-image-cmd>”/mnt/yh-supe/PHOLD/PholdTCP”,”1”,”1”</combined-image-cmd>
    <combined-image-entrypoint></combined-image-entrypoint>
</combined-image>
```

图 6 Phold 组合镜像模板
Fig. 6 Phold combined image component template



https://dc-china-simulation.researchcommons.org/journal/vol32/iss11/16
DOI: 10.16182/j.issn1004731x.joss.20-FZ0423

```
<combined-image>
    <name>SUPE SOS</name>
    <from> centos:7</from>
    <maintainer> shuai wang-wangshuai18@bit.edu.cn </ maintainer >
    <component> SUPE </component>
    <contents>
        <add>src=”yh-supe.tar.gz” dest=”/mnt/yh-supe/” </add>
        <add>src=”Opinion_Poll.tar.gz” dest=”/mnt/yh-supe/” </add>
        <add>src=”configure.tar.gz” dest=”/mnt/yh-supe/”</add>
        <run>yum -y update</run>
        <run>yum install -y tar</run>
        <run>yum clean all</run>
        <run>mv /mnt/yh-supe/configure /mnt/yh-supe/Opinion_Poll/src/bin</run>
        <workdir>/home</workdir>
        <env>LD_LIBRARY_PATH = /mnt/yh-supe/lib/TCP</env>
    </contents>
    <expose>8080</expose>
    <combined-image-cmd>”/mnt/yh-supe/Opinion_Poll /src/bin/CrowdBehaviorTCP”,”1”,”1”</combined-image-cmd>
    <combined-image-entrypoint></combined-image-entrypoint>
</combined-image>
```

图 7 SOS 组合镜像模板
Fig. 7 SOS combined image component template

```
FROM centos:7
MAINTAINER shuai wang "wangshuai18@bit.edu.cn"
WORKDIR /home
RUN yum -y update &&\
    yum install -y tar &&\
    yum clean all
ADD yh-supe.tar.gz /mnt/yh-supe/
ADD PHOLD.tar.gz /mnt/yh-supe/
ENV LD_LIBRARY_PATH = /mnt/yh-supe/lib/TCP
EXPOSE 8080
CMD ["/mnt/yh-supe/PHOLD/PholdTCP ",”1”,”1”]
```

图 8 Phold Dockerfile 文件
Fig. 8 Phold Dockerfile

step 4：使用 Docker build 命令将 Dockerfile 打包成镜像，完成自动化封装。

传统的 Dockerfile 封装方式重用性较低，且当仿真应用所需依赖库发生改变时，需要重写 Dockerfile 文件，导致仿真应用开发部署工作量较大，流程较复杂。使用基于 XML 的库组件模板，用户无需重写 Dockerfile 文件，只需根据需要选择所需依赖库组件模板即可自动生成 Dockerfile 文件，同时大大提高了库组件的可重用性，同一个模板可以被多个仿真应用所使用，提高了仿真应用封装效率。

```
FROM centos:7
MAINTAINER shuai wang "wangshuai18@bit.edu.cn"
WORKDIR/home
ADD yh-supe.tar.gz /mnt/yh-supe/
ADD Opinion_Poll.tar.gz /mnt/yh-supe/
ADD configure.tar.gz /mnt/yh-supe/
RUN yum -y update &&\
    yum install -y tar &&\
    yum clean all &&\
    mv/mnt/yh-supe/configure/mnt/yh-supe/Opinion_Poll/src/
    bin
ENV LD_LIBRARY_PATH = /mnt/yh-supe/lib/TCP
EXPOSE 8080
CMD ["/mnt/yh-supe/Opinion_Poll/src/bin/CrowdBehavior
TCP",”1”,”1”]
```

图 9 SOS Dockerfile 文件
Fig. 9 SOS Dockerfile

### 3.3 Dockerfile 语法优化与错误检测

为了证明Dockerfile语法优化与错误检测的有效性，将 Dockerfile 语法优化与错误检测规则应用在生成 SUPE 镜像，Phold 组合镜像与 SOS 组合镜像中，两种镜像的基础镜像均为 centos: 7。

从图 10 中，可以看出经过优化后的镜像大小明显减少，其中对于 Dockerfile 命令较多的 SOS 镜像优化效果更为显著。正是由于将多条命令合成一条，镜像层数减小，镜像大小也随之减小。镜像生成时间减少幅度较少，是因为下载传输文件比较耗时。因此，Docker 语法优化与错误检测能有效解决传统 Dockerfile 打包镜像占用空间大的问题，对减少镜像构建时间也有一定的效果。

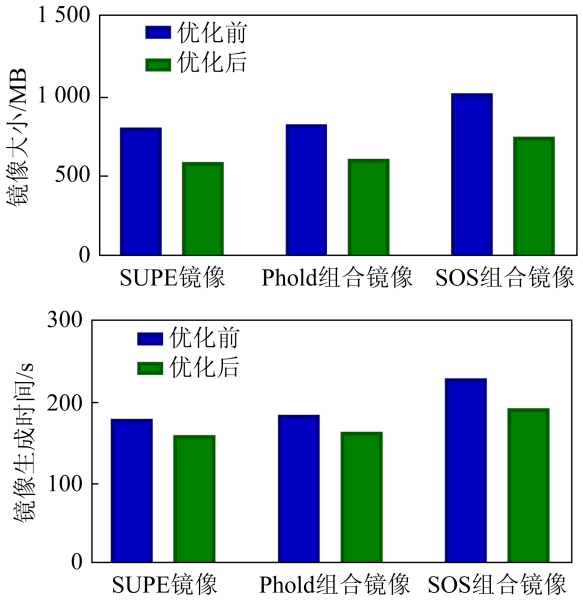


图 10 SUPE 镜像与组合镜像性能对比
Fig. 10 Performance comparisons of SUPE image and combined images

## 4 结论

针对目前仿真应用封装效率低，难度大的问题，提出了基于容器的复杂系统仿真应用自动封装技术，设计了基于 XML 的库组件模板与 Dockerfile 语法优化与错误检测模块，提高了仿真应用封装效率。基于容器的复杂系统仿真应用的自动封装框架可实现对仿真应用的自动化封装。通过基于 SUPE 仿真引擎的复杂系统仿真应用封装实验验证了该技术的有效性。

## 参考文献:

[1] 刘晓平, 唐益明, 郑利平. 复杂系统与复杂系统仿真研究综述[J]. 系统仿真学报, 2008, 20(23): 6303-6315.
Liu Xiaoping, Tang Yiming, Zheng Liping. Survey of Complex System and Complex System Simulation[J]. Journal of System Simulation, 2008, 20(23): 6303-6315.

[2] Lan C H, Hsui C Y. The deployment of artificial reef ecosystem: Modelling, simulation and application[J]. Simulation Modelling Practice and Theory (S1569-190X), 2006, 14(5): 663-675.

[3] Hou B, Yao Y, Wang B. Modeling and simulation of large-scale social networks using parallel discrete event simulation[J]. Simulation: Trans. Soc. Model. Simul. Int. (S1741-3133), 2013, 89(10): 1173-1183.

[4] Zhou H, Toth G. Efficient OpenMP parallelization to a complex MPI parallel magnetohydrodynamics code[J]. Journal of Parallel and Distributed Computing (S0743-7315), 2020, 139: 65-74.

[5] Daniella N, Wainer G. A kernel for embedded systems development and simulation using the boost library[C]. Proceedings of the Symposium on Theory of Modeling & Simulation. Pasadena: ACM, 2016: 13.

[6] Wan X, Guan X, Wang T, et al. Application deployment using Microservice and Docker containers:Framework and optimization[J]. Journal of Network and Computer Applications (S1084-8045), 2018, 119: 97-109.

[7] 刘海洋, 胡晓峰, 雷旭. 基于图形集群的远程实时渲染系统研究[J]. 系统仿真学报, 2019, 31(5): 886-892.
Liu Haiyang, Hu Xiaofeng, Lei Xu. Remote Real-time Rendering System Based on Graphics Cluster[J]. Journal of System Simulation, 2019, 31(5): 886-892.

[8] Benedictis M D, Lioy A. Integrity verification of Docker containers for a lightweight cloud environment[J]. Future Generation Computer Systems (S0167-739X), 2019, 97: 236-246.

[9] 耿朋, 陈伟, 魏峻. 面向 Dockerfile 的容器镜像构建工具[J]. 计算机系统应用, 2016, 25(11): 14-21.
Geng Peng, Chen Wei, Wei Jun. Tool for Building Docker Image on Dockerfile[J]. Computer Systems Applications. 2016, 25(11): 14-21.

[10] 李伟. 基于 Docker 的镜像组合技术研究与实现[D]. 广州: 华南理工大学, 2017.
Li Wei. Research and Implementation of Image Combination Based on Docker[D]. Guangzhou: South

China University of Technology, 2017.

[11] Zhu F, Yao Y, Tang W, et al. A hierarchical composite framework of parallel discrete event simulation for modelling complex adaptive systems[J]. Simulation Modelling Practice and Theory (S1569-190X), 2017, 77: 141-156.

[12] 陈李鸿. 仿真系统云平台的环境构建与资源调度[D]. 武汉: 华中科技大学, 2019.
Chen Lihong. Environment Construction and Resource Scheduling of the Cloud Platform Simulation[D]. Wuhan: Huazhong University of Science and Technology, 2019.

[13] Fujimoto R M. Performance of Time Warp under synthetic workloads[C]. Proceedings of the SCS Multiconference on Distributed Simulation. San Diego: ACM, 1990, 22: 23-28.

[14] Zhu F, Yao Y, Tang W, et al. A hierarchical composite framework of parallel discrete event simulation for modelling complex adaptive systems[J]. Simulation Modelling Practice and Theory (S1569-190X), 2017, 77: 141-156.