基于领域知识的Docker镜像自动构建方法

陈伟[1,2]，叶宏杰[1,2]，周家宏[1,2]，魏峻[1,2]
1. 中国科学院大学，北京 100190；2. 中国科学院软件研究所，北京 100190

摘要
Dockerfile是构建Docker应用镜像的脚本代码，包含软件系统镜像构建所需的软件包及其依赖的下载、安装和配置的所有指令。编写Dockerfile需要丰富的领域知识，否则编写的Dockerfile容易产生镜像构建错误。针对此问题，提出一种基于领域知识的Docker镜像自动构建方法。该方法通过对大规模Dockerfile的自动解析，分析提取构建Docker镜像所需的软件依赖及安装配置等领域知识；在面向特定软件系统构建镜像时，从已构建的领域知识库中分析推断指定软件的依赖关系及安装操作，生成Dockerfile来构建Docker镜像。实验结果表明，该方法具有利用领域知识推断系统依赖关系和软件包安装方式、生成不同软件Dockerfile的能力。

关键词
Docker；Dockerfile；知识图谱；软件包；系统依赖

中图分类号：TP311 文献标识码：A doi: 10.11959/j.issn.2096-0271.2021005

An approach to automatically building Docker images by using domain knowledge

CHEN Wei[1,2], YE Hongjie[1,2], ZHOU Jiahong[1,2], WEI Jun[1,2]
1. University of Chinese Academy of Sciences, Beijing 100190, China
2. Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

Abstract
A Dockerfile builds a Docker image by specifying how to construct a software system by downloading, installing and configuring software packages and their dependencies. However, manually writing a Dockerfile can be error-prone because system dependency resolution requires a lot of domain knowledge. Therefore, an approach to automating Dockerfile generation based on domain knowledge was proposed. The approach automatically parses Dockerfiles and extracts knowledge of building Docker images and stores the knowledge in a graph database. When generating new Dockerfiles, the system dependencies and their installation operations for the designated software based on the knowledge base were inferred. Experiments indicate that it is viable to automate Dockerfile generation for diversified software by inferring system dependencies and software package installations with the domain knowledge.

Key words
Docker, Dockerfile, knowledge graph, software package, system dependency

1 引言

在传统软件开发过程中，开发部署和运行演化两阶段相互割裂，各阶段数据汇聚与知识提炼、关联与运用程度低，难以快速响应需求变化。为此，开发运维一体化（DevOps）[1]被提出，旨在加强开发和运维部门之间的沟通协作，提高软件运行演化过程中生产活动的效率和质量。DevOps的引入对软件产品的开发、测试、交付和运维有重要意义。

Docker[2]是当前主流的容器技术，在DevOps中被广泛使用。Docker容器是Docker镜像的实例，封装了软件应用程序及其系统依赖项（即操作系统和相关软件包），构建了保证软件系统能够正常运行的环境。Docker镜像成为DevOps中软件系统构建和发布的主流制品形式，Docker容器则成为复杂分布式系统部署和运行的主流基本构成。Dockerfile是基于领域特定语言（domain specific language，DSL）编写的脚本代码，用于构建Docker镜像，并实例化Docker容器。Dockerfile包含一组构建Docker镜像的指令序列[2]，声明了构建镜像时使用的操作系统、安装的软件包以及安装顺序等。

尽管Dockerfile被广泛用于构建Docker镜像，但人工编写Dockerfile十分复杂且容易出错，Dockerfile质量问题导致的Docker镜像构建失败案例普遍存在[3]。一方面，Dockerfile指定了镜像构建的系统环境配置，特别是软件包之间的关联和依赖、软件包与操作系统的兼容性、软件下载安装的操作以及顺序等，需要全面的领域知识；另一方面，人工编写Dockerfile时的拼写错误、语法错误、违反最佳实践[4]和Dockerfile坏味（bad smell）[5]等质量问题难以避免。例如，在为Python代码片段构建Docker容器运行环境时，开发人员平均要花费2 h编写Dockerfile，但是仍难以保证Python代码片段正确运行[6]。

本文提出了一种基于领域知识的Dockerfile自动生成方法，用于支持Docker镜像的自动构建。该方法首先自动解析Docker Hub上的大量Dockerfile，从中提取构建Docker镜像所需的细粒度知识，包括基础镜像、操作系统、系统软件包的下载和安装配置等，并通过软件包在Dockerfile中的出现顺序和共现性推断软件包之间的关联，构建领域知识库。在面向特定软件系统构建镜像时，方法基于领域知识分析推断指定软件的系统依赖关系及其安装操作，并生成Dockerfile，用于构建Docker镜像。在最后的实验中，本文选取了100个不同类型的软件系统，并为其构建容器镜像，本文方法能够为其中的73个软件系统成功构建Docker镜像。实验结果表明，本文方法在构建软件镜像时能够应对软件类型的多样性，具有较好的镜像构建成功率。本文的工作主要有以下两点贡献。

- 提出了一种面向Docker镜像构建的领域知识图谱自动构建方法。本文方法提出了Docker镜像构建的领域知识图谱元模型，并基于抽象语法树（abstract language tree，AST）分析技术从大规模Dockerfile中提取各种类型的领域知识实体与关系，实现知识图谱的构建。
- 提出了一种Dockerfile的自动生成方法。本文方法根据知识图谱推断构建目标软件系统镜像所需的基础镜像、需要安装的所有软件包及安装顺序和操作，生成指令，并合成Dockerfile。

2 相关工作

DockerizeMe[7]和FRISK[8]与本文

工作相关度较高。DockerizeMe[7]主要解决Python代码中因缺少第三方依赖而导致的导入错误(import error)问题，并通过构建Docker镜像来实现Python代码的运行环境。针对Python包依赖问题，DockerizeMe收集Python软件包索引(Python package index，PyPI)上流行的前1万个Python包及其资源，并从安装和导入包过程的日志中提取包之间的依赖关系，建立Python包依赖库。对于给定的Python代码，DockerizeMe根据依赖库推断代码的第三方依赖包，并将Python:2.7.13作为基础镜像构建容器环境。FRISK[8]的目标是为问答论坛(如Stack Overflow)中问题的解决方案构建复现环境，尤其是面向与服务器端开发相关的问题。FRISK预定义了几个Dockerfile模板，用于创建具有多种语言环境和数据库的服务器端Web框架运行环境，主要面向Express.js、Rails 5、Django、Flask等。通过FRISK，用户可以从模板Dockerfile开始修改，创建符合要求的Dockerfile，进而生成相应的Docker容器环境。但是，DockerizeMe和FRISK都仅仅面向特定的问题场景，难以很好地应对软件系统的多样性及其资源依赖给Docker镜像构建带来的困难。

除了上述两项工作，还有其他工作关注与Docker相关的知识图谱构建和Dockerfile质量问题。DockerPedia[9]是一个面向软件之间依赖关系以及安全漏洞信息的知识图谱，基于轻量级的本体论(ontology)[10]，建立了不同概念之间的联系。Binnacle工具集[4]针对Dockerfiles构建AST，然后使用频繁子树挖掘算法来挖掘Dockerfile编写中的语义规则和最佳实践。Lu Z G等人[11]总结了Dockerfile中的4种坏味模式，并提出了一种基于状态的静态分析方法来检测Dockerfile坏味。Wu Y W等人[5]开展实证研究，总结了开源软件中的Dockerfile坏味。Hassan F等人[12]通过分析软件环境的变化及其影响，向开发人员推荐Dockerfiles的更新。Cito J等人[3]对Docker生态系统、Docker质量问题和Dockerfiles的演变进行了探索性的实证研究。

3 基于领域知识的Docker镜像自动构建方法

如**图1**所示，基于领域知识的Docker镜像自动构建方法主要分为两个阶段：知识图谱构建和Docker镜像自动生成，其中第

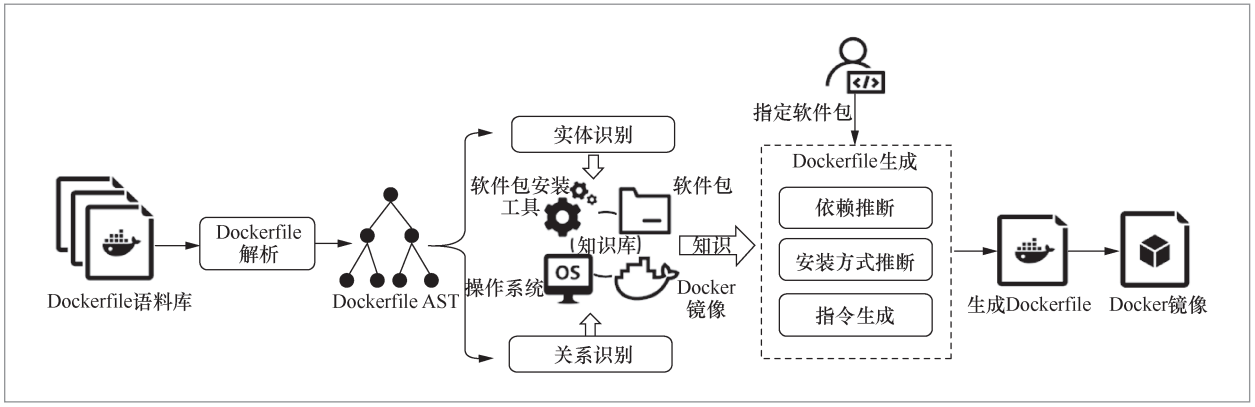


图1 基于领域知识的 Docker 镜像自动生成构建流程

二阶段的重点在于Dockerfile的自动生成。

在知识图谱构建阶段，本文方法从Docker Hub上收集大量Dockerfile，并自动解析，从基于解析构建的Dockerfile AST中抽取出实体和关系，并基于定义的知识图谱元模型构建知识图谱。

在Docker镜像自动生成阶段，对于用户指定需要安装的目标软件，本文方法从知识图谱中推断构建Docker镜像所需的基础镜像和该软件包关联的其他软件包，并确定相应的安装配置顺序和方式。最后，方法根据分析结果构造Dockerfile指令，并合成Dockerfile，进而基于Dockerfile构建镜像。

4 数据收集与知识图谱构建

数据收集与知识图谱的构建主要包括以下步骤：

步骤1 基于网络爬虫获取Docker Hub上的Docker项目及其对应的Dockerfile等数据；

步骤2 解析Dockerfile，并构建AST；

步骤3 从Dockerfile的AST中识别各种类型的实体以及实体间的关系；

步骤4 基于知识图谱元模型，整合解析得到的各类型实体和关系，生成知识图谱。

4.1 知识图谱元模型

领域知识图谱元模型M由实体集合En和关系集合Re构成，即M=(En,Re)。元模型结构如**图2**所示，主要包括8种类型的实体（Docker项目、Dockerfile、Docker镜像、操作系统、操作系统版本、软件包安装工具、软件包版本、软件）；以及8种类型的关系（包含、构建、基于、实例化、使用、安装、兼容、关联），涵盖了Dockerfile自动生成所需的多种知识。元模型表达的语义知识包括：Docker项目包含Docker镜像和构建该镜像所使用的Dockerfile；Docker镜像可以依赖其他镜像，即将其他镜像作为构建时的基础镜像；Docker镜像中包含使用的操作系统信息，以及安装的

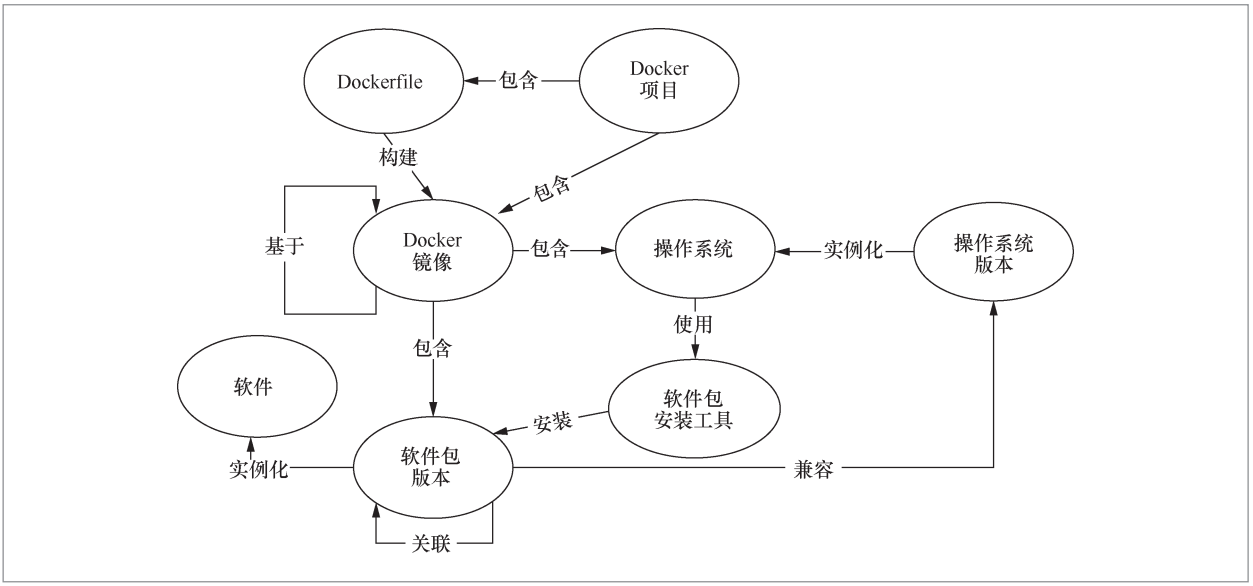


图 2 领域知识图谱元模型

软件包及其版本信息；软件包可以通过包管理软件安装，或者通过下载、配置、编译的方式安装；操作系统有不同的版本，不同操作系统安装软件包的方式不同，不同版本的操作系统可安装的软件包也不同；软件是软件包安装后的实例；软件包之间可能存在关联或依赖关系，这种关系决定了多个软件包在下载安装时需按照一定的先后顺序执行。

4.2 数据获取

构建知识图谱需要将大量的Docker项目和Dockerfile作为知识源。Docker Hub是Docker官方维护的大型公共Docker仓库，包含数以百万计的Docker项目，但是没有提供完整的项目列表和查询接口。针对这些问题，本文设计并实现了一个高效的爬虫工具，自动爬取Docker Hub中的海量项目数据。爬虫工具的工作流程如下：

- 针对缺少完整Docker项目列表的问题，爬虫实现了一个基于英文字母组合的检索关键词生成机制，使得检索结果能够覆盖所有的Docker项目；
- 以不同的关键词在Docker Hub上进行检索，获得以各个关键词开头的Docker项目列表；
- 针对海量数据的爬取效率问题，实现了基于多线程的并行爬取流程，通过多个线程的并行执行来提高Docker Hub上Docker项目的获取效率。

目前本文工作已经收集了约110万个Docker项目的信息和约96万份Dockerfile。

4.3 Dockerfile解析

Dockerfile解析包括两个阶段：Dockerfile预处理；构建Dockerfile对应的AST。

Dockerfile中的指令主要包括FROM指令、ENV指令和RUN指令等。其中，FROM指令用于指定基础镜像，ENV指令用于定义环境变量，RUN指令用于声明构建基础镜像时需要运行的bash命令行指令。例如，在**图3**的Dockerfile中，第1行FROM指令指出当前Docker镜像是基于centos镜像、centos7版本构建的；第3行ENV指令定义了两个环境变量LANG和LC_ALL，取值都为C.UTF-8；第6~8行RUN指令指出构建镜像时需要运行yum install指令，安装wget、bzip2、ca-certificates等软件包。

Dockerfile预处理主要解析环境变量定义指令，并在随后出现的指令中将环境变量替换为对应的值，即通过预处理实现环境变量的实例化。解析环境变量包括以下两个步骤。

步骤1 解析环境变量定义指令。ENV指令的格式可以表示为“ENV key value”，其中，key表示环境变量的名称，value表示环境变量的值。因此，可以构建名-值映射表Map<key,value>来存储环境变量。对于每一条ENV语句，提取其中的key和value，并存入映射表中，实现对环境变量定义指令的解析。

步骤2 替换后续指令中出现的环境变量。在Dockerfile中使用环境变量时，环境变量以“$”开头。以此为依据提取每一条指令中出现的环境变量的名称，在映射表中查找对应的值，完成环境变量实例的替换。

图4是一份带有环境变量的Dockerfile，经过环境变量解析，共解析出ANDROID_NDK_VERSION、COCOS2D_X_VERSION、NDK_HOME 3个环境变量，并进一步对第5、8、9、10、11行环境变量的值进行替换，最终转换成为如**图5**所示的Dockerfile。

完成环境变量替换后，本文方法考虑

Dockerfile指令和嵌套的bash指令的不同语法，设计了以下两阶段的Dockerfile指令解析方法，从而构建AST。

- 以Docker项目的名称为根节点，将每条指令解析为根节点的一个叶节点，用指令的名称表示；
- 关注指令后的文本。有些指令后的文本仍是Dockerfile指令的语法，如FROM、ENV等指令。对于这些指令，仍使用Dockerfile指令的语法解析器进行解析，生成相应的叶节点。有些语句指令后的文本嵌套的是bash指令的语法，如RUN、CMD等指令。对于这些指令，使用bash指令的语法解析器进行解析，生成相应的叶节点。

例如，图3的Dockerfile生成的AST如图6所示。其中，浅色节点是使用Dockerfile语法解析器解析生成的节点，深色节点是使用bash语法解析器解析生成的节点。

4.4 实体和关系识别

得到Dockerfile对应的AST后，本文方法进一步从中识别实体及其之间关系，主要包括：基础镜像和操作系统识别、软件包识别、软件包关联识别。

对于基础镜像和操作系统识别，Dockerfile中的FROM指令声明了当前Docker镜像的基础镜像。本文方法对AST中以FROM节点为根的子树进行分析，得到基础镜像信息。镜像具有传递依赖性，且依赖于某个操作系统镜像。首先判断当前镜像img_{df}是否是操作系统镜像，或者基础镜像img_{base}是否是操作系统镜像，如果是，则得到操作系统信息；否则，解析img_{base}的Dockerfile，判断img_{base}的基础镜像是否是操作系统镜像，重复该过程，直到发现操作系统镜像，得到操作系统信息。

对于软件包识别，根据Dockerfile对软件包的安装方式，本文首先将软件包分为官方软件包（officially packaged software，OPS）和非官方软件包（un-

```
1 FROM centos:centos7
2
3 ENV LANG=C.UTF-8 LC_ALL=C.UTF-8
4 ENV PATH /opt/conda/bin:$PATH
5
6 RUN yum install -y wget bzip2 ca-certificates \
7     libglib2.0-0 libxext6 libsm6 libxrender1 \
8     git mercurial subversion openssh-server openssh-clients
9
10 RUN wget --quiet https://repo.continuum.io/archive/Anaconda3-5.1.0-Linux-x86_64.sh -O ~/anaconda.sh && \
11     /bin/bash ~/anaconda.sh -b -p /opt/conda && \
12     rm ~/anaconda.sh && \
13     ln -s /opt/conda/etc/profile.d/conda.sh /etc/profile.d/conda.sh && \
14     echo ". /opt/conda/etc/profile.d/conda.sh" >> ~/.bashrc && \
15     echo "conda activate base" >> ~/.bashrc
16
17 RUN conda install -y pytorch torchvision cuda91 -c pytorch
18
19 CMD [ "/bin/bash"]
```

图 3 示例Dockerfile(1)

```
1 ......
2 ENV ANDROID_NDK_VERSION r10e
3 ENV COCOS2D_X_VERSION 3.8.1
4 ......
5 ENV NDK_HOME /opt/android-ndk-$ANDROID_NDK_VERSION
6 ......
7 # Install cocos2d-x
8 RUN wget http://cocos2d-x.org/filedown/cocos2d-x-$COCOS2D_X-VERSION.zip && \
9     unzip cocos2d-x-$COCOS2D_X_VERSION.zip && \
10    rm cocos2d-x-$COCOS2D_X_VERSION.zip && \
11    cd cocos2d-x-$COCOS2D_X_VERSION && \
12    python setup.py
13 ......
```

图 4 示例Dockerfile(2)

```
1 ......
2 ENV ANDROID_NDK_VERSION r10e
3 ENV COCOS2D_X_VERSION 3.8.1
4 ......
5 ENV NDK_HOME /opt/android-ndk-r10e
7 # Install cocos2d-x
8 RUN wget http://cocos2d-x.org/filedown/cocos2d-x-3.8.1.zip && \
9     unzip cocos2d-x-3.8.1.zip && \
10    rm cocos2d-x-3.8.1.zip && \
11    cd cocos2d-x-3.8.1 && \
12    python setup.py
13 ......
```

图 5 环境变量替换后的Dockerfile(2)

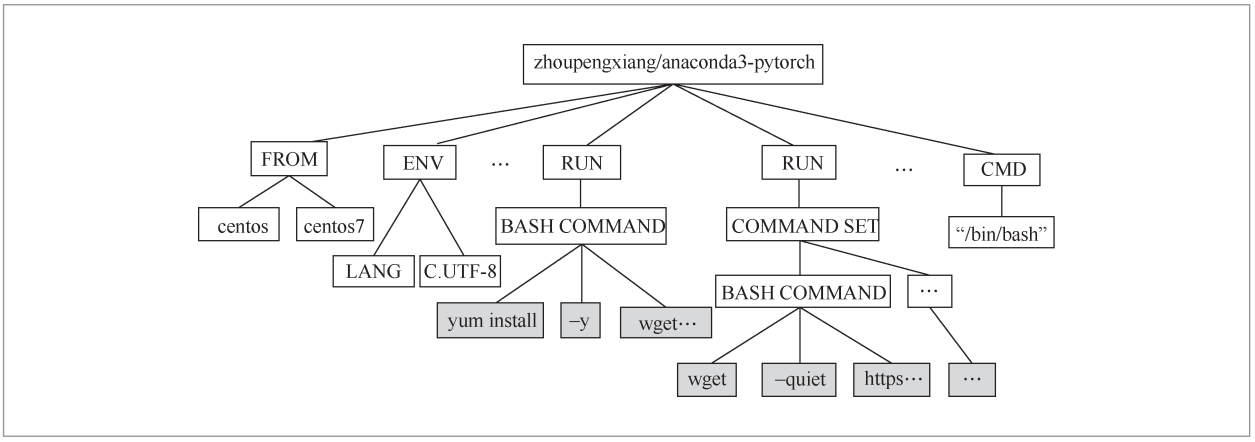


图 6 Dockerfile(1)对应的 AST

officially packaged software，UOPS）两类。OPS指登记在公共仓库中、统一管理的软件包，可以通过apt/apt-get、YUM等包管理工具自动安装、管理和卸载。UOPS指无法通过包管理工具自动下载、安装的软件，通常以压缩文件或Git仓库的形式存在，并由唯一的统一资源定位器（uniform resource locator，URL）指定软件包下载地址，开发者和用户通常需要下载并进行解压和编译，再执行相应的安装操作。以**图3**的Dockerfile为例，第6行中的wget、bzip2等是OPS，可以通过包管理工具YUM进行下载；而第10行中的https://repo.continuum.io/archive/Anaconda3-5.1.0-Linux-x86_64.sh是UOPS，需要通过下载、解压、切换工作目录、运行安装程序等步骤才能安装。

对于OPS，如前所述，apt/apt-get、YUM、dnf等常见的包管理工具提供了OPS的安装能力，可以通过相应的包管理器命令进行下载安装。对RUN节点的子树进行分析，得到bash语句中的指令节点和参数节点，当指令节点是包管理器的安装命令（如apt-get install、yum install等）时，提取参数节点进一步分析。首先通过yum-showduplicates list、apt-cache pkgnames等命令获取各个包管理器中所有可安装的软件包列表，之后通过包名匹配的方式确定当前语句安装的软件包及其版本。例如，对于**图3**第6行（对应**图6**中AST第三棵子树），本文方法可以分析出语句安装了wget、bzip2和ca-certificates等包。

对于UOPS，本文方法关注wget、cURL、Git等常用于下载UOPS的bash指令。由于UOPS并没有对软件进行统一命名，本方法使用下载的URL作为UOPS的唯一标识，并通过URL解析的方式，在下载指令的参数节点中确定安装的UOPS。额外地，本文方法通过爬虫验证下载链接是否可访问，以保证UOPS的有效性。

对于软件包关联识别，经过软件包识别，本文方法可获取当前Dockerfile安装的软件包集合PKG={pkg_1,pkg_2,…,pkg_n}。基于Dockerfile进行Docker镜像构建时，软件包的安装是有序的。本方法以关联<pkg_i,pkg_j>的形式记录两个包pkg_i和pkg_j在当前Dockerfile中的安装顺序，表示pkg_i先于pkg_j安装，作为软件包之间的关联。

4.5 知识图谱构建

基于第4.1节定义的知识图谱元模

型，本文方法将所有Dockerfile解析提取得到的实体和关系整合写入知识图谱$G_{df}=(V,E)$，其中，V为顶点集合，E为边集合；V对应元模型M中的实体集合En，每个顶点v代表一个唯一的实体，其类型为Docker镜像、软件包、操作系统等；E对应M中的关系集合Re，边e_{ij}代表两个顶点v_i、v_j之间的关系（即两个实体之间的关系），并用边的权重表示该关系在所有Dockerfile中出现的频数。当边e_{ij}连接的两个顶点v_i、v_j代表两个软件包时，则e_{ij}表示两者之间的先后安装顺序，如果e_{ij}和e_{ji}同时出现在知识图谱中，则说明两个软件包之间并没有依赖关系，可以以任意顺序安装。

经过对约22万份高质量的Dockerfile进行分析，本文方法建立了一个含有约90万个顶点和约290万条边的知识图谱。

5 Dockerfile自动生成方法

给定一个软件包（尤其是UOPS，因为典型的OPS可以通过特定的包管理器自动下载安装），生成对应的Dockerfile需要考虑以下几方面：基础镜像、需要安装的软件包、软件包的安装顺序。因此，本文方法的任务是根据给定的软件包s，在知识图谱G_{df}中挖掘提取出三元组$K_s=(img_{base}, PKG_s, seq_s)$，其中，$img_{base}$表示安装s时使用的基础镜像；$PKG_s$表示安装s时需要安装的所有软件集合（包括s本身）；seq_s表示PKG_s中所有软件的安装顺序集合，安装顺序以软件对$<pkg_i,pkg_j>$的形式出现。

5.1 基础镜像推荐

基础镜像推荐包括两步：在G_{df}中找到候选基础镜像集合；候选基础镜像排序。

首先，本文方法在G_{df}中进行搜索，找到所有安装了给定软件包s的镜像。如果用户指定了操作系统，则再根据操作系统筛选出满足用户需求的镜像，生成候选镜像集合。

得到候选镜像集合后，本文方法根据镜像的流行度进行排序。镜像的流行度指一个镜像被选作其他镜像的基础镜像的次数。排序后，将流行度最高的镜像作为安装s时使用的基础镜像img_{base}。

5.2 关联软件包分析

软件包间的关联具有传递性，故在G_{df}中，从s对应的顶点开始，采用广度优先搜索（breadth first search，BFS）算法找到所有与s关联的包，生成G_{df}的子图G_s。子图中所有顶点即需要安装的软件包集合PKG_s。

部分关联出现次数较少，可信度较低。因此，本文方法提出关联度$cor(pkg_i,pkg_j)$这一指标，评判两个软件包pkg_i、pkg_j之间存在关联的可信度。BFS只会搜索关联度高于设定阈值的边。关联度计算方法如下。

- 当软件包pkg_i和pkg_j之间只存在一条边时，$cor(pkg_i, pkg_j)$的计算方法如式（1）所示。其中，$w(i, j)$表示知识图谱中边e_{ij}的权重，即所有Dockerfile中pkg_i和pkg_j共同被安装的次数。$|pkg_i|$表示在所有Dockerfile中，pkg_i被安装的次数。若软件包pkg_i和pkg_j之间只存在一条边，且关联度高于阈值，则说明pkg_i和pkg_j之间存在依赖关系，pkg_i需要在pkg_j之前安装。
- 当软件包pkg_i和pkg_j之间存在两条边时，$cor(pkg_i,pkg_j)$的计算方法如式（2）所示。其中，$w(i, j)+w(j, i)$表示在所有Dockerfile中pkg_i和pkg_j共同被安装的次数，$|pkg_i|+|pkg_j|$表示所有Dockerfile中pkg_i被安装的次数和pkg_j被安装的次数之

和。软件包pkg_i和pkg_j之间存在两条边，且关联度高于阈值，只能说明两个包之间存在关联，两个软件包可以以任意顺序安装。

$$\mathrm{cor}(\mathrm{pkg}_i,\mathrm{pkg}_j)=\frac{w(i,j)}{|\mathrm{pkg}_i|} \quad (1)$$

$$\mathrm{cor}(\mathrm{pkg}_i,\mathrm{pkg}_j)=\frac{w(i,j)+w(j,i)}{|\mathrm{pkg}_i|+|\mathrm{pkg}_j|} \quad (2)$$

5.3 软件包安装顺序推断

为了确定软件包的安装顺序，需要对子图G_s中的各个顶点进行拓扑排序。排序前需要消除G_s中的环。如果环中的顶点数为2，则删除环中的所有边，因为这两个软件包可以以任意顺序安装；如果环中的顶点数大于2，则删除环中关联度最小的边。消除环后，本文方法对各个顶点进行拓扑排序，排序的结果即软件包的安装顺序seq_s。

5.4 Dockerfile生成

根据基础镜像、需要安装的软件包及安装顺序，本方法生成Dockerfile的步骤如下。

- 根据基础镜像img_{base}，生成指令FROM img_{base}。
- 根据软件包的安装顺序，逐条生成各个软件包的安装指令。
- 对于OPS，在知识图谱中找到该软件包对应的包管理器，生成运行包管理器安装该软件包的RUN指令。例如，若软件包pkg的包管理器是apt-get，则会生成指令RUN apt-get install -y pkg[=version]，以安装该软件包。
- 对于UOPS，本文发现在Dockerfile中，每个UOPS安装语句前后通常会有空行，形成一个独立的指令块（如**图3**中第9~16行对anaconda的安装）。因此，以空行进行划分可以得到UOPS的安装方式。从中选取使用频率最高的安装方式，生成对UOPS的安装指令块。

6 实验与分析

本文所有实验都在一台8核3.50 GHz、32 GB内存的机器上进行，操作系统为Ubuntu 18.04.01 LTS，使用的Docker版本为19.03.6，设置的关联度阈值为0.5。

6.1 实验方法

本文通过实验验证笔者提出的方法是否能够为给定的软件包生成Dockerfile，并成功构建Docker镜像。本文在开源社区（如GitHub、Apache Software Foundation等）中随机选取了100个UOPS进行实验，即使用本文方法生成Dockerfile，并验证是否能根据该Dockerfile成功构建镜像和运行UOPS。在100个UOPS中，49个来自GitHub，其余51个来自其他仓库，并涵盖了各种类型的软件，如系统软件、开发工具、应用软件等。经过检验，这100个UOPS的下载链接均是有效的。**表1**列出了选取的100个UOPS的详细信息。此外，为了进一步说明方法的有效性，本文尝试生成8个常见的基于Docker的Web框架（包括Express.js、Rails 5和Django等）的Dockerfile，与FRISK[8]进行对比。

本文使用以下两个指标分析实验结果。

- 构建成功率（build success rate，BSR）：表示Dockerfile成功构建镜像的比率，计算方法如式（3）所示，其中$|DF_{total}|$表示生成Dockerfile的总数，$|DF_{bs}|$表示基于生成的Dockerfile能够成功构建镜像的数量。

表1 实验UOPS详细信息

类别	软件包数量/个	代表软件包	描述
系统软件	20	Tomcat, Nginx, Apache Spark, Apache Storm, Redis, etcd	服务器、平台、数据库等
开发工具	26	JDK, Python, Maven, Anaconda, Node.js, NVM, SBT	程序开发工具、开发环境、构建工具等
常用工具	35	pinpoint-agent, BTSync, Dehydrated, Puppet, libiconv	提供通用功能的工具和库
应用程序	15	WordPress, Tiny Tiny RSS, Magento2	网络应用、商业应用、视频软件等
其他	4	files and keys	其他不属于上述分类的软件包

● 配置成功率（configuration success rate，CSR）：表示Dockerfile成功构建镜像，并使得给定软件能够正确运行的比率，计算方法如式（4）所示，其中$|DF_{cs}|$表示成功运行的镜像的数量。

$$BSR = \frac{|DF_{bs}|}{|DF_{total}|} \quad (3)$$

$$CSR = \frac{|DF_{cs}|}{|DF_{total}|} \quad (4)$$

6.2 实验结果与分析

通过本文方法生成Dockerfile后，使用“docker build”命令构建Docker镜像，人工观察构建结果，并统计分析构建失败的原因。结果显示，在100个软件包中，73个软件包对应的Dockerfile能够成功构建镜像（BSR=73%），59个软件包对应的Dockerfile不仅可以成功构建镜像，而且能正确运行镜像中的软件（CSR=59%）。另外，对于8个常见的Web框架，本文方法均成功生成Dockerfile，并使得框架能够正确运行。结果表明，本文方法具有利用领域知识推断系统依赖关系和软件包安装方式的能力，能够自动生成不同软件的Dockerfile。

本文对生成的100份Dockerfile进行分析，发现以下两点。

● 最常被安装的软件包是cURL，其次是wget、tar、Git和GNU Make等，分布如**图7**所示。这些软件包主要用于下载、解压和编译UOPS。

● Ubuntu操作系统镜像最常被作为基础镜像，被作为基础镜像的比率达到47%。

本文对构建失败的Dockerfile进行分析。构建失败的主要原因如下。

● 基础镜像获取失败：Docker Hub上存储的基础镜像丢失或无法访问，无法拉取基础镜像构建新的镜像。5份Dockerfile构建失败的原因是基础镜像获取失败。

● 依赖缺失：没能在知识图谱中建立软件包完整的依赖关系，导致软件包无法成功安装。6份Dockerfile构建失败的原因是依赖关系缺失。

● 文件路径错误：构建Docker镜像时，访问了已经不存在的文件路径。6份Dockerfile构建失败的原因是文件路径错误。

● 其他错误：包括字符集编码错误、授权无效等。10份Dockerfile构建失败的原因是其他错误。

同时，本文对配置失败的Dockerfile进行分析，发现配置失败的主要原因是不完整配置，即在软件包安装指令中，缺少一些必要的指令（如环境配置指令、文件操作指令等），使得Docker镜像无法正确运行。

笔者认为，可以从以下方面进一步改进，减少构建失败和配置失败。

● 完善知识图谱：继续从Docker Hub和GitHub等开源社区收集Docker项目，解析Dockerfile，并提取软件包之间的关联，

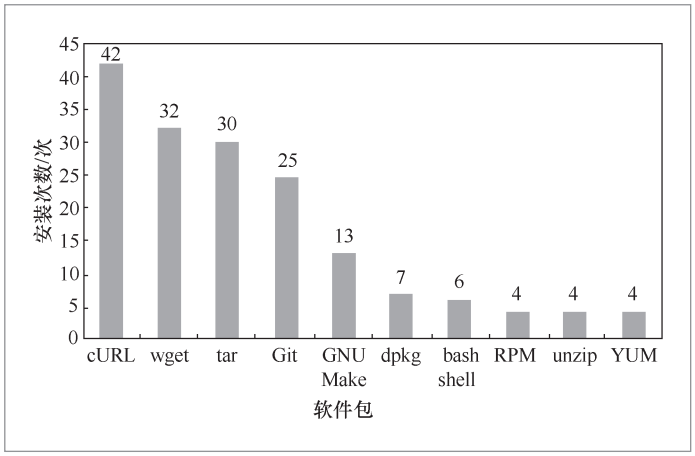


图 7 常用软件包统计

进一步提高知识图谱的完整性。

● 资源有效性检测：在使用资源（包括基础镜像和软件包等）前预先访问，以确保资源的有效和可访问。

● UOPS配置模式总结：UOPS的安装配置主要包括下载、解压、编译和建立链接等步骤，因此可以进一步总结UOPS的配置模式，用于完善软件安装所必需的相关指令。

7 结束语

本文提出了一种基于领域知识的Docker镜像自动生成方法。该方法通过对数十万的Dockerfile进行解析，提取其中与镜像构建相关的实体和关系等知识，构建Docker领域知识图谱。对于给定需要构建镜像的软件包，该方法通过知识图谱推断目标软件的基础镜像、所有需要安装的依赖软件包以及安装顺序，在此基础上生成Dockerfile，并进一步构建面向目标软件的Docker镜像。实验结果显示，该方法具有利用领域知识推断系统依赖关系和软件包安装方式的能力，能够自动生成面向不同软件的Dockerfile和Docker镜像。在未来的研究中，笔者认为可以从提高知识图谱完整性、Dockerfile优化、语言层包依赖解析等方面着手，进一步提高Docker镜像的自动生成能力。

参考文献：

[1] EBERT C, GALLARDO G, HERNANTES J, et al. DevOps[J]. IEEE Software, 2016, 33(3): 94-100.

[2] MERKEL D. Docker: lightweight Linux containers for consistent development and deployment[J]. Linux Journal, 2014.

[3] CITO J, SCHERMANN G, WITTERN J E, et al. An empirical analysis of the Docker container ecosystem on GitHub[C]//2017 IEEE/ACM 14th International Conference on Mining Software Repositories. Piscataway: IEEE Press, 2017: 323-333.

[4] HENKEL J, BIRD C, LAHIRI S K, et al. Learning from, understanding, and supporting DevOps artifacts for Docker[C]// The ACM/IEEE 42nd International Conference on Software Engineering. New York: ACM Press, 2020: 38-49.

[5] WU Y W, ZHANG Y, WANG T, et al. Characterizing the occurrence of Dockerfile smells in open-source software: an empirical study[J]. IEEE Access, 2020, 8: 34127-34139.

[6] HORTON E, PARNIN C. Gistable: evaluating the executability of Python code snippets on GitHub[C]//2018 IEEE International Conference on Software Maintenance and Evolution. Piscataway: IEEE Press, 2018: 217-227.

[7] HORTON E, PARNIN C. DockerizeMe: automatic inference of environment dependencies for python code snippets[C]//2019 IEEE/ACM 41st International Conference on Software Engineering. New York: ACM Press, 2019: 328-338.

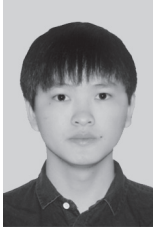
[8] MELO L, WIESE I S, AMORIM M D. Using Docker to assist Q&A forum users[J]. IEEE Transactions on Software

Engineering, 2019: 1.
[9] OSORIO M, BUIL-ARANDA C, VARGAS H. DockerPedia: a knowledge graph of Docker images[C]//International Semantic Web Conference (P&D/Industry/BlueSky). [S.l.:s.n.], 2018.
[10] HUO D, NABRZYSKI J, VARDEMAN C F. Smart container: an ontology towards conceptualizing Docker[C]//International Semantic Web Conference (Posters & Demos). [S.l.:s.n.], 2015.
[11] LU Z G, XU J W, WU Y W, et al. An empirical case study on the temporary file smell in Dockerfiles[J]. IEEE Access, 2019, 7: 63650-63659.
[12] HASSAN F, RODRIGUEZ R, WANG X Y. RUDSEA: recommending updates of Dockerfiles via software environment analysis[C]//The 33rd ACM/IEEE International Conference on Automated Software Engineering. New York: ACM Press, 2018: 796-801.

作者简介

陈伟(1980-),男,博士,中国科学院软件研究所副研究员,中国计算机学会会员,主要研究方向为软件工程、网络分布式计算等。

叶宏杰(1996-),男,中国科学院软件研究所硕士生,主要研究方向为软件工程。



周家宏(1995-),男,中国科学院软件研究所硕士生,主要研究方向为机器学习、知识图谱等。

魏峻(1970-),男,博士,中国科学院软件研究所研究员、博士生导师,主要研究方向为软件工程、网络分布式计算等。

收稿日期: 2020-10-20
通信作者: 陈伟, chenwei@iscas.ac.cn
基金项目: 国家重点研发计划基金资助项目(No.2016YFB1000800);国家自然科学基金资助项目(No.61732019)
Foundation Items: The National Key Research and Development Program of China(No.2016YFB1000800), The National Natural Science Foundation of China(No.61732019)