计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(7):293−298 [doi: 10.15888/j.cnki.csa.009199]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于序列挖掘的 Dockerfile 规则自动提取工具①

詹威霖[1], 周　宇[1,2]

[1](南京航空航天大学 计算机科学与技术学院, 南京 211106)
[2](南京航空航天大学 高安全系统的软件开发与验证技术工信部重点实验室, 南京 211106)
通信作者: 周　宇, E-mail: zhouyu@nuaa.edu.cn

摘　要: Dockerfile 定义了一组构建容器镜像的指令, 这些指令指示了容器化的应用程序该如何构建. 最近的研究表明 Dockerfile 存在相当多的质量问题. 在本文中, 我们提出了一种新的工具 DMiner (Dockerfile Miner) 来提取高质量 Dockerfile 中的隐含规则, 这些规则将有助于提升 Dockerfile 的质量. DMiner 主要分为 3 个模块, 分别负责 Dockerfile 的采集、过滤, Dockerfile 的解析处理以及 Dockerfile 规则的挖掘提取, DMiner 将 Dockerfile 解析成统一的序列表示并使用序列模式挖掘算法来提取规则. 本工具对现有的 Dockerfile 数据集进行了扩充, 同时新提取出了 9 条在其他工作未曾出现的规则, 在真实数据集上的大量实验证明了该工具的有效性和高效性.

关键词: Dockerfile; 规则挖掘; Docker; 配置文件

引用格式: 詹威霖,周宇.基于序列挖掘的 Dockerfile 规则自动提取工具.计算机系统应用,2023,32(7):293–298. http://www.c-s-a.org.cn/1003-3254/9199.html

Dockerfile Rule Automatic Extraction Tool Based on Sequence Mining

ZHAN Wei-Lin[1], ZHOU Yu[1,2]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)
[2](Key Laboratory of the Ministry of Industry and Information Technology for Software Development and Verification Technology of High Security Systems, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

Abstract: Dockerfile defines a set of instructions for building container images, which instruct how the containerized applications should be built. Recent studies have shown that there are quite a lot of quality problems in Dockerfile. This study proposes a new tool, namely Dockerfile Miner (DMiner) to extract implicit rules from high-quality Dockerfile, and these rules will help to improve the quality of Dockerfile. DMiner is mainly divided into three modules, which are responsible for the collection and filtering of Dockerfile, parsing of Dockerfile, and mining and extraction of Dockerfile rules. DMiner parses Dockerfile into a unified sequential representation and uses a sequential rule mining algorithm to extract rules. This tool expands the existing Dockerfile dataset and extracts nine new rules that have not appeared in other work. A large number of experiments on real datasets show that the tool is effective and efficient.

Key words: Dockerfile; rule mining; Docker; configuration files

云原生环境依靠容器镜像提供可部署的应用程序. 由于相应的执行环境封装在容器镜像中, 因此用户可以在目标平台上直接运行应用程序, 而不需要考虑配置差异. 构建容器镜像的指令按照一套语法规则在 Dockerfile 中按顺序指定. 因此 Dockerfile 的质量对构建的镜像至关重要. 然而最近对大型开源项目的实证研究暴露了对现有 Dockerfile 的质量与其功能或性能的严重担忧, 研究显示, 大型开源项目中的一些 Dockerfile 甚至是错误且无法构建的[1,2].

Dockerfile 可以视作与其他源代码级别的软件制

① 基金项目: 国家自然科学基金 (61972197); 江苏省自然科学基金 (BK20201292)
收稿时间: 2022-12-22; 修改时间: 2023-02-23, 2023-03-14; 采用时间: 2023-03-20; csa 在线出版时间: 2023-05-22
CNKI 网络首发时间: 2023-05-24

© 中国科学院软件研究所 http://www.c-s-a.org.cn

品一样, 需要按照基本原则、规则或其他设计模式仔细设计. 现有的几种工具为 Dockerfile 的检查提供了一些初级的支持, 在语法级别 (例如突出显示关键字、悬停语法等) 做出提示[3]. 事实上, Docker 的官方网站提供了编写 Dockerfile 的最佳实践指南[4]. 但是这份指南的内容是一种抽象层次的指南, 只是提出了一些原则, 缺少具体的示例和规则. 同时最重要的是. 指南更关注特定于 Dockerfile 的指令, 但在 Dockerfile 中, 最常用的是 Shell 脚本 (即由 Run 命令引导的命令), 通常占所有指令的 40% 以上 (一些研究甚至显示, 高达 68.3% 的 Dockerfile 更改集中在 Shell 命令上), 大约 90% 的存储库有使用 Shell 命令[4].

图 1 展示了一个例子. 通常, 我们需要从网络下载压缩文件, 然后再执行解压缩命令. 在图 1 中, wget 命令用来下载远程网站中的压缩文件, 然后使用 unzip 命令来解压缩下载的文件. 执行这样的命令将保留原始的压缩文件, 并创建一个新的文件夹来放置解压缩后的文件, 虽然原始的压缩文件已经不再需要, 但是这种情况下, 原始的压缩文件将仍然保留在最后生成的容器镜像中. 诸如此类的行为, 将不必要的文件留存在容器镜像, 会导致更长的构建时间和更大的容器镜像体积. 因此, 如图 2 所示, 在使用 unzip 之后应使用额外的 rm 命令删除原始的下载文件.

```
RUN wget -O data.zip https://example.com/data.zip && \
    unzip data.zip && \
...
```

图 1　下载后解压

```
RUN wget -O data.zip https://example.com/data.zip && \
    unzip data.zip && rm data.zip && \
...
```

图 2　解压后删除

这个例子说明了在编写 Dockerfile 时应该遵守的一些隐含的规则. 但实际上, 这些规则在很大程度上并没有包括在官方的最佳实践指导指南中. 违反这样的规则并不一定会导致构建失败, 但是会对非功能属性产生负面影响 (增加了产生的镜像的体积), 类似于程序中的“代码气味”概念[5,6].

已经有一些工作和工具来解决这一方面的问题, 两个有代表性的最新工具包括 Hadolint[7] 和 Binnacle[1], 它们试图识别 Dockerfile 中的此类规则, 从而来检查人们在编写 Dockerfile 时的错误. 然而, 它们都受到各种限制, 如人为干预较多, 效率较低等.

为了提高 Dockerfile 的质量, 减少 Dockerfile 中诸如此类“代码气味”的出现, 本文提出了一个新颖的 Dockerfile 规则提取工具 DMiner 来半自动化的提取 Dockerfile 中有助于提升代码质量的潜在规则.

DMiner 能够在大量 Dockerfile 文件中, 通过序列挖掘算法挖掘并提取 Dockerfile 中的隐含规则. 以大量高质量的 Dockerfile 作为输入, 将 Dockerfile 处理成统一的序列化中间表示. 引入 PrefixSpan[7] 序列模式挖掘算法, 设计多种启发式的规则对规则进行约简, 最后在极少人工参与的条件下自动化地得到 Dockerfile 中的隐含规则.

本文着重介绍了 Dockerfile 的规则提取和检测方法 (DRIVE) 中对 Dockerfile 进行规则挖掘和提取的工具 DMiner 的实现, 更多和算法和理论有关的详细信息可以参考 DRIVE[8] 的原始论文.

为了评估该工具的有效性, 本文选择了在 Dockerfile 规则挖掘领域最新的工具 Binnacle 作为在挖掘效果和挖掘效率上的对比, 实验表明, DMiner 不仅可以更高效的挖掘出规则. 而且可以挖掘出更多的隐含规则.

1　工具的设计与实现

DMiner 是一个命令行程序, 其主要架构如图 3 所示, 主要包含 3 大模块, 数据过滤模块、数据解析模块、规则挖掘模块.

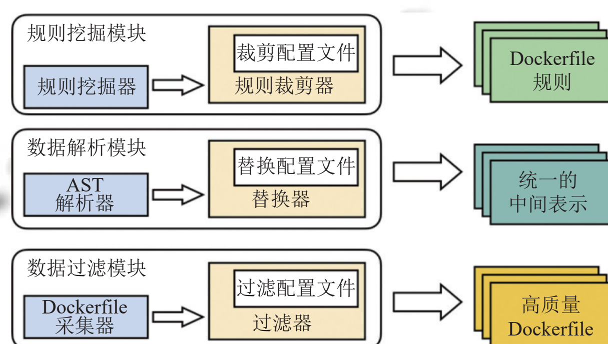

图 3　工具架构

数据过滤模块主要涉及 Dockerfile 的收集与过滤, 通过自动化地采集、过滤得到高质量的 Dockerfile 原始数据, 作为后续模块的工作基础.

数据解析模块的主要工作包括 Dockerfile 的解析和依照配置文件进行的表示替换, 并转换为序列化的中间表示其输入可以是数据模块过滤得到的 Dockerfile 集合, 也可以是由其他工作得到的 Dockerfile 集合.

规则挖掘模块挖掘数据解析模块解析输出的 Dockerfile 的中间表示, 采用启发式地规则裁剪, 相关

© 中国科学院软件研究所 http://www.c-s-a.org.cn

的规则也是由配置文件驱动的. 最后在裁剪的结果上, 通过一定的人工总结得到最后的 Dockerfile 规则.

DMiner 也根据如上的 3 个模块, 由 3 个可以独立工作的子命令组成, 接受不同的参数和配置文件. 分别是 DMiner spider、DMiner parser 和 DMiner miner.

其中 DMiner spider 可以作为一个通用的 Dockerfile 采集器存在, 根据设定的参数不同, 可以从 GitHub 自动采集不同需求的 Dockerfile.

DMiner parser 的输出将默认作为 DMiner miner 的输入, 但是为了避免过强的耦合也为了工具的可复用性, DMiner miner 可以独立的工作, 其输入序列可以是任意的符合 DMiner 定义的输入.

1.1 数据采集过滤模块

DMiner spider 即数据采集与过滤模块主要由两部分组成, 别是采集器和过滤器. 运行时如图 4 所示.

```
$./DMiner spider --star=1000 --line=4 --waring=2
Mining from GitHub with at least 1000 stars,4 line
The max warning number is 2

522c2b89eaee4c04d98e6da5dfc944eb132de856.Dockerfile saved…
38a65c46e7da3cff225ea250fa48e85a15e18883.Dockerfile saved…
094b187b3c411ba8e52c0fde425582dc552f187f.Dockerfile saved…
fd93dab321565566e8fbd51495182404260b73bc.Dockerfile saved…
fc0912752b0a3c2dd1b42cd661c7dc68602d7a2d.Dockerfile saved…
```

图 4 数据采集器

采集器: 是一个使用 Python 编写的一个分布式爬虫, 利用 GitHub 官方提供的 API[9] 从 GitHub 查询包含 Dockerfile 的所有代码存储库. 该采集器可以根据设定的条件, 自动化地从 GitHub 上采集对应的数据, 在我们的实验中我们选择了截至 2022 年 5 月的所有代码仓库中的 Dockerfile.

过滤器: 由采集器收集到的 Dockerfile 将自动地送到过滤器, 由于收集到的 Dockfile 数量巨大 (大概 40 万份原始的 Dockerfile), 其质量也参差不齐, 因此需要设计一个过滤器对这些 Dockerfile 进行过滤. 过滤器负责对采集到的 Dockerfile 进行过滤. 该过滤器是可以配置的, 可以根据 Dockerfile 使用的语言、所属仓库的知名度 (star 数量)、作者、提交日期等进行过滤. 同时过滤器也集成了 Dockerfile 的语法检查工具, 可以对收集到的 Dockerfile 进行基本的语法正确性检查.

采集器和过滤器都有一些默认配置, 这些默认配置都是可以修改定制的, 其中的默认配置如下.

(1) 选择超过一定星星数量 (GitHub 的社区评价机制, 被认为质量较高) 的仓库所属的 Dockerfile 作为第一个过滤标准. 在我们的工具中, 星星数量的默认配置为 1 000.

(2) 对 Dockerfile 的行数进行检查. 过于简短的 Dockerfile 通常只包含最基本的 Dockerfile 语句, 对 Dockerfile 的潜在规则提取没有帮助. 因此在这一步将过滤掉那些过于简短的 Dockerfile, 在我们的工具中, 这一阈值的默认配置为 4, 即少于 4 行的 Dockerfile 将会被移除.

(3) 利用 Dockerfile 的语法检查器来对这些 Dockerfile 进行进一步过滤, 删去那些有语法错误 (sytax error) 的 Dockerfile, 并对语法检查器警告 (waring) 的数量进行限制, 如果一份 Dockerfile 出现过多的 waring, 我们的工具也会将其移除. 在我们的工具中, waring 的数量上限默认值为 2.

可以看出, 第 1 个模块主要负责 Dockerfile 的采集和过滤, 为之后 Dockerfile 规则挖掘做准备. 我们使用数据过滤采集模块收集了一个规模更大也更新的高质量 Dockerfile 数据集, 从 2022 年 5 月之前 GitHub 上开源仓库中所有的约 40 万份 Dockerfile 中筛选得到了 1 761 份高质量的 Dockerfile 文件并开源[8], 以供本文的实验以及未来的其他工作使用.

1.2 数据解析模块

DMiner parser 即数据解析模块主要由 3 个组件构成, 与原论文中提到的 3 步解析相对应. 其一是实现对 Dockerfile 自身的指令解析的模块, 其二是对 Dockerfile 中的 Shell 脚本进行解析的 Shell 解析模块, 其三是根据预定义的配置对一些变量进行替换的模块.

使用 DMiner parser 解析一份 Dockerfile 的示例如图 5 所示. DMiner 的行为由命令行参数指定, 在这里 parser 格式化输出并高亮了原始的 Dockerfile 输入文件以及最后得到的序列化的中间表示. 图 5 中的格式化输出是为了提高可读性, 实际上最终进行挖掘的序列仅仅使用空格分隔, 且一条序列即代表一份 Dockerfile.

Dockerfile 指令解析模块将根据 Dockerfile 的语法说明解析 Dockerfile, 解析特定的一些指令如 FROM, COPY 等, 得到 Dockerfile 的抽象语法树, 在这一步中, 如果 RUN 命令所引导的 Shell 脚本以附加的脚本文件的形式存在, 会将脚本文件读取替换作为 RUN 命令的参数, 但是这一步 RUN 命令中的 Shell 脚本仅作为参数字符串保留, 待后续步骤再进行解析, 遍历抽象语法的树的每个节点可以得到 Dockerfile 的基本序列表示, 这意味着, 除了 Dockerfile 中的 Shell 脚本没有解析以外, 其他所有的命令都将被完整地解析.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
$./DMiner parse -print-origin -color example.Dockerfile
Original Dockerfile:
FROM python:3.7-slim

RUN apt-get update &&
    apt-get install -y —no-install-recommends curl &&
    rm -r /var/lib/apt/lists/*

COPY requirements.txt ./

RUN pip install \
    --no-cache-dir
    -r requirements.txt

Parsed Dockerfile:
FROM-IMAGE-[python]-TAG-[SPECIFIC]

SC-[apt-get]
   SC-[apt-get]-ARG-[update]

SC-[apt-get]
   SC-[apt-get]-ARG-[install]
   SC-[apt-get]-ARG-[-y]
   SC-[apt-get]-ARG-[--no-install-recommends]
   SC-[apt-get]-ARG-[curl]

SC-[rm]
   SC-[rm]-ARG-[-r]
   SC-[rm]-ARG-[PATH-APT-LIST]

COPY-[FILE-PIP-REQUIREMENT.TXT]-[PATH-NORMAL]

SC-[pip]
   SC-[pip]-ARG-[install]
   SC-[pip]-ARG-[--no-cache-dir]
   SC-[pip]-ARG-[-r]
   SC-[pip]-ARG-[FILE-PIP-REQUIREMENT.TXT]
```

图 5 Dockerfile 解析器

Shell 解析模块将解析 Dockerfile 中所有没有完成解析的 Shell 脚本, 这一模块输出的最终结果是完成全部解析并转换成基本的序列表示的 Dockerfile. 虽然这里的解析器是两个独立的模块, 但是在实际的工具实现中, 这两个模块的功能是耦合在一起的. 将 Dockerfile 作为输入, 将直接输出完全解析完成之后的序列表示.

替换模块将根据配置文件中预定义的启发式规则, 自动地检查解析器输出的序列表示, 替换其中的一些文件名、URL 等变量. 这一配置文件中的具体规则可以参考 DRIVE 论文[8].

通过如此的解析与替换, 得到了便于后续挖掘应用序列挖掘算法的通用中间表示, 这一中间表示的既保留了 Dockerfile 文件的语义信息, 同时又尽可能地提升了利用序列挖掘算法的效果.

1.3 规则挖掘模块

DMiner 即规则挖掘模块主要由一个分布式并行的 PrefixSpan 频繁序列挖掘器和一个可定制的序列裁剪器组成.

经过第 1.2 节中解析模块处理的 Dockerfile 数据集被转换成了一种序列化的中间表示. miner 会根据文献 [9] 提到的方法, 自动化的数据集进行分组, 拆分成若干个小数据集. 即将使用了同一个 Shell 命令的若干个 Dockerfile 的将被划分到同一个数据集. 这里的数据集之间并不互斥, 即一份 Dockerfile 会出现在其所有使用到的 Shell 命令所代表的小数据集中.

得益于之前的 Dockerfile 的解析工作, 我们可以非常方便地从序列化的中间表示中提取出 Dockerifle 中使用到的 Shell 命令. 分析图 5 可知 Dockerfile 中的 Shell 脚本中的所有出现的命令都被解析成了“SC-[命令名称]”的形式, 这里的 SC 是 Shell Command 的缩写. 那么提取序列中的这一项即可得到 Dockerfile 中所使用到的 Shell 命令.

这样的分析可以得到所有使用到的 Shell 命令, 在这些命令的基础上, miner 会对 Shell 命令出现的频率进行检查, 这里的频率指的是该命令在多少份 Dockerfile 中出现过. 即使一份 Dockerfile 多次使用同一 Shell 命令, 对该 Shell 命令的频率计数也只加一.

对于那些使用频率过低的命令, miner 将不会创造该命令的分组. 而其他命令, miner 将会把所有使用到该命令的 Dockerfile 放到同一个分组. 在实现中就是一个文本文件代表一个分组, 文件的每一行代表着一份 Dockerfile 的序列表示. 那么显然, 文件的行数就代表这该分组的数据集的大小.

为了充分利用我们分组的特性, 我们用 Go 语言实现了一个修改地分布式并行的 PrefixSpan 频繁序列挖掘器. 该挖掘器可以自动地根据所使用的机器的 CPU 核数对每个分组数据集进行并行化地挖掘. 该挖掘器的输入包括输入文件 (一行代表一个序列) 以及一个子序列的最小支持度 (百分数值). 而挖掘器的输出即该输入文件对应支持度的所有频繁子序列.

在得到了频繁子序列之后, DMiner 需要对输出的频繁子序列进行筛选过滤. 因此 DMiner 中同时包含了一个可定制的序列过滤器, 默认将使用 DRIVE 论文提出的算法[8] 对输出的频繁子序列进行过滤, 之后人工地对过滤后的频繁子序列进行分析. 得到由该子序列所代表的 Dockerfile 潜在规则. 所以 DMiner 也提供了一个简单的 Web UI 来帮助专家对于每个命令分组的数据集的输出结果进行分析总结. 该部分的展示如图 6 所示. 图 6 展示了最终在 pip 这个命令分组中的第 3 个频繁子序列, 从这个子序列中我们可以得到一条 Dockerfile 中的潜在规则: pip 命令应该使用 requirements.txt 文件中来安装指定的依赖.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

图 6 辅助总结规则的 Web UI

2 实验评估

在这一部分中, 我们进行实验来评估我们的工具, 为此我们提出了两个研究问题 (RQ).

RQ1: 我们的规则挖掘工具 DMiner 的效果?

RQ2: 我们的规则挖掘工具 DMiner 的效率?

所有实验是均在同一台服务器上完成, 该服务器配备了 Intel Xeon 2.3 GHz 32 核 CPU 和 32 GB RAM, 运行 Arch Linux. 程序的原型是用 Go v1.18 和 Python v3.10.4 实现的. 最新的和 Dockerfile 规则相关的工具包括 Binnacle[1] 和 Hadolint[10]. Binnacle 对 Dockerfile 进行分层解析, 并使用挖掘频繁子树的方法来对 Dockerfile 中的规则进行提取. Hadolint 是一个 Dockerfile 的规则检测工具, 其中的 Dockerfile 规则都来自于社区贡献, 并不能主动地挖掘并提取规则. 因此本文实验中比较的基线是主要是目前最新的工具 Binnacle[1]. 为了保证实验的效果, 我们的实验中主要使用了两个数据集.

D1: 我们的数据过滤模块筛选出的高质量 Dockerfile 集合, 它包含 1 761 份 Dockerfile.

D2: Binnacle 工具采用的高质量 Dockerfile 数据集[11], 是目前已知最新的公开数据集, 包含 405 份 Dockerfile.

DMiner 在挖掘时, 只需要接受两个参数, 一个是需要设置频繁子序列的最低支持度, 其二是待挖掘的 Dockerfile 集合. 根据序列挖掘算法的特性[12] 和实验验证, 当支持度设置得过高时, 挖掘出的频繁子序列的数量会变少, 在进行 Dockerfile 规则筛选过滤时, 丢失了较低支持度时能挖掘出的规则. 而支持度过低的频繁子序列并不能反映出潜在的 Dockerfile 规则. 因此, 经过我们的多次实验, 选择将最低支持度设置为 40%.

2.1 RQ1: 挖掘方法的效果

如前所述, D1 是我们的方法收集的高质量 Dockerfile 数据集. 在规则挖掘阶段, 我们根据命令对解析的 Dockerfile 进行分组, 在 D1 数据集上得到了 77 个命令组. 然后从每组数据中挖掘出频繁模式. 每组在经过序列过滤器之前的输出的频繁模式的平均个数为 4 515. 在通过序列过滤之后, 每组的平均模式只剩下 4 个, 减少了 99.9%.

然后, 我们人工检查每组输出的模式. 由于我们的方法设置了支持度的阈值, 我们验证了这些规则的置信度[12]. 通过检查这些挖掘出的规则, 其中有 25 个与之前的相关工作发现的规则[1] 相一致, 我们还发现 9 条在以前的相关工作中没有被识别到 (在 Binnacle 或 Hadolint 中提及) 的新规则, 意味着这几条规则是我们的工具所新发现的, 这也证明了我们的方法的有效性, 新发现的规则在表 1 中列出.

表 1 DMiner 挖掘出的新规则

序号	规则描述	置信度 (%)
1	git clone 使用 HTTPS 的超链接	96
2	使用unzip解压文件之后应当删除原压缩文件	70
3	chown 使用参数 -r	61
4	Go的项目应当使用多步构建方法	91
5	Java的项目应当使用多步构建方法	72
6	在使用conda安装依赖之后, 应当清除conda的缓存	72
7	在下载完文件之后, 使用SHA校验下载的文件的完整性	56
8	在下载完文件之后, 使用GPG校验下载的文件的完整性	42
9	在Shell脚本中set -eux以打印命令并快速失败	57

由于 DMiner 和 Binnacle 使用了不同的数据集来挖掘规则, 为了进行公平的比较, 我们还在相同的数据集上进行的对比. 即 D1 和 D2. 比较结果如表 2 所示.

可以发现, 在 D2 这个较小的数据集上, 两者挖掘出的规则数量没有太大差异. 而当数据集变大, DMiner 在挖掘这些隐含规则方面表现出优势. 这是因为 Binnacle 使用频繁子树挖掘的算法, 只能挖掘预定义的局部子树[13,14], 当数据集扩大, 预定义的子树没有扩大, 导致效果变化不明显. 而我们的方法在替换变量后, 既保留了文本的语义, 又可以挖掘命令之间的关系, 从而可以更从容地应对更加巨大的数据集. 结果表明, 我们的方法能更有效地识别 Dockerfile 中的隐含语义规则.

表 2 挖掘出的规则个数对比 (个)

数据集	DMiner	Binnacle
D1 (1 761)	34	22
D2 (405)	18	17

2.2 RQ2: 挖掘方法的效率

在这个 RQ 中, 我们主要考虑方法的性能, 主要包括运行时间和运行时所消耗的内存. 作为比较, 我们在上面提到的两个数据集上运行 Binnacle 和 DMiner, 并

© 中国科学院软件研究所 http://www.c-s-a.org.cn

分别在解析和规则挖掘阶段收集它们的运行时间和内存占用, 运行时间和运行内存占用的对比分别如表 3 和表 4 所示.

表 3 时间消耗对比 (s)

数据集	DMiner		Binnacle	
	解析	挖掘	解析	挖掘
D2 (405)	3	201	62	1 028
D1 (1 761)	14	257	264	1 386

表 4 内存消耗对比 (MB)

数据集	DMiner		Binnacle	
	解析	挖掘	解析	挖掘
D2 (405)	10	10	13	17
D1 (1 761)	34	37	56	295

可以看出, 在数据预处理部分和规则挖掘部分, DMiner 的运行效率都高于 Binnacle. 在数据预处理中, DMiner 可以高效地将 Dockerfile 转换为序列形式. 而在规则挖掘部分, 一方面因为我们选择的序列挖掘算法在速度上具有优势, 而且我们设计的挖掘工具在每个命令分组之间的挖掘工作相互独立, 并行运行. 因此, 可以显著加快工具的运行时间.

而在内存的使用量方面, DMiner 有着更少的内存占用. 当数据集规模不大时, DMiner 和 Binnacle 占用的内存相近, 但随着数据集规模增长, 由于我们的算法对 Dockerfile 进行了分组, 在分组挖掘时不会使得占用的内存增大太多, 而 Binnacle 采用基于频繁子树的挖掘方法[14], 需要将所有的 Dockerfile 解析的结果缓存在内存中, 从而导致占用的内存增加.

通过以上分析, 我们可以得出结论, 我们的方法能够非常高效地提取 Dockerfile 规则.

3 结论与展望

本文基于序列模式挖掘技术, 提出了一种从高质量 Dockerfile 中高效挖掘隐含规则的新工具 DMiner. 我们证明了我们的工具相对于最先进的基线的有效性. DMiner 可以用更少的时间, 更高的效率发现更多有用的隐含规则.

在未来, 我们计划用更多的功能来增强 DMiner, 比如对 Dockerfile 是否符合规则的检测, 并开发成熟的工具 (作为主流 IDE 的插件) 来实时地检测用户编写的 Dockerfile, 以提供更好的可用性.同时这种数据驱动的工具也可以扩展到其他密切相关的领域, 如其他配置文件模式地挖掘和代码气味地检测等.

参考文献

1 Henkel J, Bird C, Lahiri SK, *et al*. Learning from, understanding, and supporting DevOps artifacts for Docker. Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering. Seoul: IEEE, 2020. 38–49.

2 Henkel J, Silva D, Teixeira L, *et al*. Shipwright: A human-in-the-loop system for Dockerfile repair. Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings. Madrid: IEEE, 2021. 198–199.

3 Visual Studio Code. Docker in visual studio code. https://code.visualstudio.com/docs/containers/overview. [2022-10-06].

4 Cito J, Schermann G, Wittern JE, *et al*. An empirical analysis of the Docker container ecosystem on GitHub. Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR). Buenos Aires: IEEE, 2017. 323–333.

5 Rasool G, Arshad Z. A review of code smell mining techniques. Journal of Software: Evolution and Process, 2015, 27(11): 867–895. [doi: 10.1002/smr.1737]

6 Wu YW, Zhang Y, Wang T, *et al*. Characterizing the occurrence of Dockerfile smells in open-source software: An empirical study. IEEE Access, 2020, 8: 34127–34139. [doi: 10.1109/ACCESS.2020.2973750]

7 Pei J, Han JW, Mortazavi-Asl B, *et al*. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. Proceedings of the 17th International Conference on Data Engineering. Heidelberg: IEEE, 2001. 215–224.

8 Zhou Y, Zhan WL, Li Z, *et al*. DRIVE: Dockerfile rule mining and violation detection. arXiv:2212.05648, 2022.

9 GitHub REST API. GitHub docs. https://ghdocs-prod.azurewebsites.net/en/rest. [2022-10-06].

10 hadolint/hadolint. Dockerfile linter, validate inline bash, written in Haskell. https://github.com/hadolint/hadolint. [2022-10-06].

11 Henkel J, Bird C, Lahiri SK, *et al*. A dataset of Dockerfiles. Proceedings of the 17th International Conference on Mining Software Repositories. Seoul: ACM, 2020. 528–532.

12 Fournier-Viger P, Lin JCW, Kiran RU, *et al*. A survey of sequential pattern mining. Data Science and Pattern Recognition, 2017, 1(1): 54–77.

13 Jiménez A, Berzal F, Cubero JC. Frequent tree pattern mining: A survey. Intelligent Data Analysis, 2010, 14(6): 603–622. [doi: 10.3233/IDA-2010-0443]

14 Mazuran M, Quintarelli E, Tanca L. Mining tree-based association rules from XML documents. Proceedings of the 17th Italian Symposium on Advanced Database Systems. Camogli, 2009. 109–116.

(校对责编: 牛欣悦)

© 中国科学院软件研究所 http://www.c-s-a.org.cn