基于聚类和偏序序列的 API 用法模式挖掘

王树怡 董 东
(河北师范大学数学与信息科学学院 石家庄 050024)

摘 要 在软件开发过程中,开发人员经常需要遵循特定的 API 用法模式,而这些用法模式几乎没有相关文档作为参考。为了挖掘 API 用法模式,提出基于聚类和频繁闭合偏序序列的 API 用法模式挖掘途径。通过抽象语法树对源代码进行解析,对提取 API 方法调用序列进行层次聚类,最后使用频繁闭合偏序挖掘算法 DFP 进行 API 用法模式的挖掘。实验结果表明,在相同的数据集上,与 SPADE 算法和 BIDE 算法相比,所得候选 API 用法模式集更加精简。

关键词 API 用法模式,序列模式挖掘,层次聚类,偏序
中图法分类号 TP311 **文献标识码** A

Mining of API Usage Pattern Based on Clustering and Partial Order Sequences

WANG Shu-yi DONG Dong
(College of Mathematics and Information Science, Hebei Normal University, Shijiazhuang 050024, China)

Abstract During software development, a developer often needs to follow specific usage patterns of application programming interface (API). However, few of those is well documented for developers to refer to in order to mining the API usage pattern, this paper proposed an approach that discovers the API usage pattern based on clustering and frequent closed partial order sequence mining. After parsing the source code by abstract syntax tree, the extracted API sequences is hierarchically clustered. Finally, API usage patterns by depth-first frequent closed partial order algorithm (DFP) is excauated. The experiment shows that this approach can obtain more succinct candidate API usage pattern compared to SPADE and BIDE on the same dataset.

Keywords API usage pattern, Sequential pattern mining, Hierarchical clustering, Partial order

1 引言

源代码的形成过程是软件系统通过 API 与库进行交互完成的。作为软件的具体实现表示,源代码蕴含着开发人员的开发理念、设计思想、设计模式和编程习惯等信息,展示了使用 API 的细节,包含 API 用法模式[1]。本文将源代码视为有噪声的数据集,引入数据挖掘技术及算法,对其进行分析处理,并结合方法调用之间的关系,转换成偏序序列挖掘问题,最终以偏序序列形式提取源代码中的 API 用法模式。挖掘得到的 API 用法模式,不仅可以降低开发人员的学习成本,帮助开发人员高效地开发软件系统,而且可用于异常检测及软件维护[2]。

在面向对象语言中,应用程序接口(Application Programming Interface, API)是类以及与之相关的方法的集合,包括标准 API、第三方库及框架 API[3]。在特定的使用场景中,一个 API 方法与其他 API 方法之间调用的先后顺序被称为 API 用法模式[4]。

随着软件开发复杂性的增加,API 用法模式越来越多地被人们关注、讨论并研究。基于 API 方法调用的表示形式主要分为以下 3 类:

1)项集表示。Li 等人[5]开发了 PR-Miner 工具,从源代码中提取频繁出现在同一个方法中的方法调用集合,使用频繁项集挖掘算法来挖掘频繁出现的方法调用集合并将其作为 API 用法模式。

2)序列表示。谢涛等人[6]开发了 MAPO 工具,基于 PMD 代码分析器分析代码,提取方法调用序列,通过频繁闭合序列挖掘算法从得到的序列中挖掘频繁闭合序列模式。

3)图表示。Nguyen 等人[7]开发了 GroumMiner 工具,基于图挖掘多对象 API 用法模式。针对面向对象语言,以方法为粒度,把对象集合的用例表示为有标签的有向图。该工具对包含控制及条件结构的 API 用法模式检测及用例异常检测都有很好的帮助。

当然,也存在一些其他形式的 API 用法模式挖掘。Rizky 等人[8]通过分析以往 API 用法模式挖掘发现,大多数的挖掘过程都没有考虑方法调用在实现过程中的角色,不同角色在方法调用序列中的作用有所不同。该文通过对源代码中方法调用的抽象,在更高层次上识别 API 用法模式。Saied 等人[9]提出挖掘多层次的 API 用法模式,通过提取源代码中与方法调用相关的信息,使用增量聚类对方法调用集合进行分层,挖掘 API 用法模式。

本文提出基于聚类和偏序序列的 API 用法模式挖掘,以偏序序列形式表示 API 方法调用序列。在挖掘的过程中把

本文受河北省自然科学基金(F2013205192)资助。
王树怡(1991-),女,硕士生,主要研究方向为经验软件工程,E-mail:1376515793@qq.com;董 东(1971-),男,硕士,副教授,主要研究方向为经验软件工程,E-mail:dongdong@hebtu.edu.cn(通信作者)。

方法调用之间的顺序关系扩展到方法调用之间的偏序关系；先后通过聚类技术和频繁闭合偏序序列挖掘算法 DFP，使得挖掘得到的候选 API 用法模式集更加精简。

本文第 2 节主要介绍涉及的概念；第 3 节介绍重点挖掘步骤和算法；第 4 节进行实验并分析实验结果；最后对全文进行总结。

2　基本概念

本节主要介绍挖掘过程中涉及的相关概念，包括抽象语法树、层次聚类以及偏序。

2.1　抽象语法树

抽象语法树(AST)作为源代码的一种中间表示形式，是一种依据某种文法生成并在内存中具有统一结构的树型结构。AST 具有较强的描述能力，可以比较准确、形象地描述程序的逻辑结构[10]。

通过遍历抽象语法树，可以获取源代码中的包名、类名、方法名、属性名和相关调用信息，从而生成 API 方法调用序列。

2.2　层次聚类

聚类把一组对象按照相似性归成若干个类别，使得属于同一类别的对象之间的距离尽可能小，而不同类别的对象之间的距离尽可能大。

层次聚类是聚类的一种，又称为树型聚类，它使用数据的联接规则，通过一种层次架构方式反复将数据进行分裂或聚合，以形成一个层次序列的聚类问题解[11]。本文通过计算 API 方法调用序列之间的相似度，利用层次聚合算法，在聚类的开始使每个序列自成一类，之后选择相似度较大的类进行合并，直到满足预定的条件。

2.3　偏序

定义 1　如果集合 A 上的二元关系 R 是自反的、反对称的和传递的，那么称 R 为集合 A 上的偏序，通常用符号“$\leqslant$”表示。当集合 A 中的任意两项 a,b 之间都满足 $R(a,b)$ 或 $R(b,a)$ 时，称 R 为全序[12]。

偏序 R 可以表示为有向无环图(DAG)，其中偏序中的项是有向无环图中的顶点。如果 $(a,b)\in R$ 且 $a\neq b$，则 $\langle a,b\rangle$ 为有向无环图中的一条边，也可记为 ab。若 a 和 b 之间存在 c 且满足 $a\leqslant c\leqslant b$，则边 ab 为冗余边。通常使用哈斯图表示去除冗余边之后的偏序。图 1(a)是包含 6 个顶点、13 条边的偏序 R；图 1(b)是 R 的哈斯图，仅包含 6 条边。其中全序的哈斯图是一条序列，而 DAG 中的每一条路径都是一条序列。

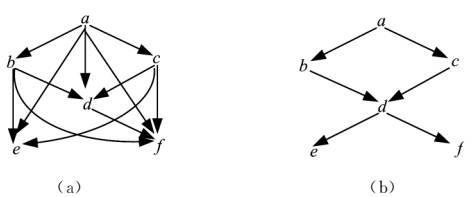

图 1　偏序及哈斯图

定义 2　序列 s 是一个全序集，表示成序列 s 的传递闭包形式，即为：

$$Closure(s)=\{(a_i,a_j)\mid 1\leqslant i<j\leqslant l\}$$

其中，l 为序列 s 的长度。一般情况下，省略 (a_i,a_i)。

由于偏序 R 可以表示为有向无环图，而 DAG 中的每一条路径都是一条序列，因此偏序 R 的传递闭包表示为 DAG 中每条序列传递闭包的并集。

定义 3　序列数据库 SDB 中包含多条序列。对于偏序 R，如果 $R\in Closure(s)$，则称序列 s 支持偏序 R。偏序 R 在 SDB 中的支持度表示为 $sup(R)$。给定一个阈值 min_sup，若 $sup(R)\geqslant min\_sup$，则称偏序 R 为频繁的。对于偏序 R，若不存在 $R\subset R'$，$sup(R)=sup(R')$，则称偏序 R 为闭合的。当偏序 R 既是频繁的又是闭合的，则称偏序 R 为频繁闭合偏序[13]。

如图 2 所示，设 $sup(R)=sup(R')$，$Closure(R)=\{(a,b),(a,e),(a,f),(b,e),(b,f),(e,f),(a,c),(c,e),(c,f),(a,d),(d,f)\}$，$Closure((R')=\{(a,b),(a,e),(a,f),(b,e),(b,f),(e,f),(a,c),(c,e),(c,f),(a,d),(d,e),(d,f)\}$，$Closure(R)\subset Closure(R')$，则 $R\subset R'$，因此 R 不是频繁闭合偏序。

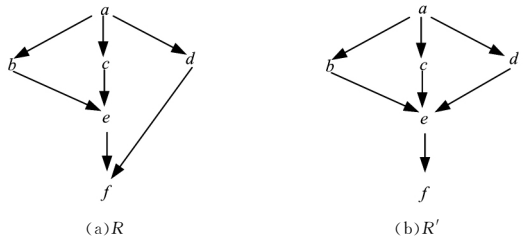

图 2　$R\subset R'$

3　基于聚类和偏序序列的 API 用法模式挖掘

挖掘 API 用法模式的过程如图 3 所示。

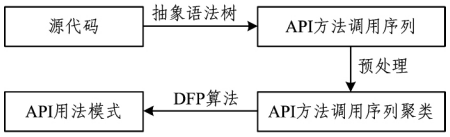

图 3　API 用法模式的挖掘过程

给定一个系统的源代码，首先把源代码转换成中间表示形式，即抽象语法树；接着根据挖掘的需要，从抽象语法树中提取方法调用及相关结点，按照调用的先后顺序形成 API 方法调用序列；通过预处理，对得到的 API 方法调用序列进行筛选，同时为了提高挖掘的覆盖率，对筛选得到的 API 方法调用序列进行层次聚类；最终在每一个聚成的类内使用频繁闭合偏序挖掘算法 DFP 进行 API 用法模式挖掘。

3.1　API 方法调用序列的提取

源代码作为软件的具体实现表示，其中包含很多对库中 API 的方法调用序列，这些序列可以作为库的使用范例，为开发人员提供参考。但源代码中同时包含了很多冗余信息，无法直接进行挖掘。为了消除无关因素的影响，首先对源代码进行预处理。

利用 Eclipse AST 的 API 把源代码转换成抽象语法树。以方法体为粒度，遍历抽象语法树，提取 API 方法调用。为了保证结果的可读性，绑定被调用方法的包名以及类名，并最终把每一项表示为“包名.类名.方法名”的形式。序列的形成过程中，忽略方法体内的控制结构，即把方法体内的方法调用按照代码行序形成一条方法调用序列。

3.2　序列聚类

通过聚类，降低不同 API 使用场景之间的相互影响，提高挖掘 API 用法模式的支持度。对于序列相似度的计算，主要基于序列的 2-gram 集合[14]。例如，定义序列 $s=\{a,b,c\}$ 的

2-gram 集合为：$G(s)=\{a,b,c,ab,bc\}$。这样转换主要基于两点：1)序列中包含单个项的重叠个数越多，序列间的相似度越大；2)序列中相同项组成的方法调用对重叠个数越多，序列间的相似度越大。利用如下公式来计算序列间的相似度：

$$sim(s_i,s_j)=\frac{|I_i\cap I_j|}{|I_i\cup I_j|}$$

其中，s_i 表示 API 方法调用序列，I_i 表示序列 s_i 的 2-gram 集合。该公式表明，如果两条序列的 2-gram 集合中元素的重叠个数越多，则序列间的相似度越大。

聚类过程中，序列间的相似度并不能直接作为距离使用，需要把序列间的相似度转换成序列间的距离。因此定义序列间的距离函数为：

$$distance(s_i,s_j)=\mathrm{sqrt}(1-(sim(s_i,s_j))^2)$$

其中，$distance(s_i,s_j)$的取值范围是$[0,1]$。该公式表明：序列间相似度越大，距离越小。

层次聚合算法由树状结构的底部开始逐层向上进行聚合。假定序列数据库 $SDB=\{s_1,s_2,\cdots,s_n\}$共有 n 条序列，则：

1)初始化：每一条序列单独成为一类；

2)找两个最近的类：$distance(s_r,s_k)=\min\limits_{\forall s_i,s_j\in SDB,s_i\neq s_j} distance(s_i,s_j)$；

3)合并最近的两个类：现有的类数减 1；

4)若所有的序列都属于同一个类，则终止本算法；否则，返回步骤 2)。

3.3 频繁闭合偏序挖掘算法 DFP

DFP(Depth-first Frequent Closed Partial Order)算法主要包括如下步骤：

1)计算整个序列数据库的检测矩阵，找到可行边的集合 FS。

2)若集合 FS 中存在支持度等于序列数据库大小的边，则将输出作为第一个偏序 R_1。从集合 FS 剩余可行边中选择支持度最大的边 e_i，按深度优先扩展 R_1。以序列数据库中包含边 e_i 的序列集合作为$(R_1\cup\{e_i\})$的伪映射数据库，并检验$(R_1\cup\{e_i\})$的伪映射数据库是否已经遍历过，若遍历过，直接剪枝；否则生成伪映射数据库，遍历该数据库，直到可行边的集合为空。

3)若集合 FS 中不存在支持度等于序列数据库大小的边，则以单个可行边作为一个独立的集合。选择支持度最大的边，以包含该边的序列集合形成其伪映射数据库，按深度优先进行扩展。剪枝同步骤 2)。

4)当可行边集合 FS 为空时，算法结束。

DFP 算法的伪代码如下：

输入：序列数据库 SDB 和最小支持度阈值
输出：频繁闭合偏序集合
方法：
1. 浏览数据库，找出频繁项集；
2. 再次浏览数据库，找到可行边集合 FS；
3. 把支持度等于|SDB|的可行边放入集合 R 中；
4. 如果 R 不为空，则输出 R 为频繁闭合偏序；
5. 令 L=$e_1,\cdots,e_n$ 为支持度小于|SDB|的可行边集合；
6. 对于集合 L 中的可行边 e_i，任选支持度最大的边；
7. 如果 R∪$\{e_i\}$的伪映射数据库 $SDB|_{R\cup\{e_i\}}$ 存在，则
8. 跳转到步骤 6；
9. 否则，形成 R∪$\{e_i\}$的伪映射数据库 $SDB|_{R\cup\{e_i\}}$；
10. 递归挖掘 $SDB|_{R\cup\{e_i\}}$；
11. 否则，如果 R 为空，则：
12. 对于集合 FS 中的可行边 e_i，任选支持度最大的边：
13. 如果$\{e_i\}$的伪映射数据库 $SDB|_{\{e_i\}}$ 存在，则
14. 跳转到步骤 12；
15. 否则，形成$\{e_i\}$的伪映射数据库 $SDB|_{\{e_i\}}$；
16. 递归挖掘 $SDB|_{\{e_i\}}$；

以序列数据库 SDB 为例，如表 1 所列，设支持度阈值为 $min\_sup=3$，简述 DFP 算法。

表 1 序列数据库 SDB

序列号	序列
1	$abcdef$
2	$acbde$
3	$dabce$
4	$dcabe$
5	$abcd$
6	$dcbae$

(1)遍历序列数据库 SDB，生成检测矩阵$\{cnt[x,y]\}$，如表 2 所列。其中 x 和 y 均为序列数据库中的频繁项，且 $cnt[x,y]$包含两部分：边 xy 的支持度及以边 xy 为子序列的序列中存在于 x,y 之间的项的集合。即若该集合为空，该边不冗余，否则该边冗余。从检测矩阵中选择支持度大于或等于给定阈值且不冗余的边作为可行边的集合，则序列数据库 SDB 的可行边集合为 FS：$\{ab:5,ac:4,ae:5,bd:3,bd:3,be:5,cb:3,cd:3,ce:5,da:3,dc:3,de:5\}$。

表 2 序列数据库 SDB 的检测矩阵

	a	b	c	d	e
a		5,Ø	4,Ø	3,{b,c}	5,Ø
b	1,Ø		3,Ø	3,Ø	5,Ø
c	2,Ø	3,Ø		3,Ø	5,Ø
d	3,Ø	3,{c}	3,Ø		5,Ø
e	0,Ø	0,Ø	0,Ø	0,Ø	

(2)由于不存在支持度大小等于序列数据库大小的边，因此以单个边为独立的集合，挑选支持度最大的边开始扩展。以集合 FS 中的边 ae 为例进行扩展，其伪映射数据库 $SDB|_{\{ae\}}$ 由包含边 ae 的序列号为 1,2,3,4,6 的序列组成，并生成$SDB|_{\{ae\}}$的检测矩阵，则伪映射数据库 $SDB|_{\{ae\}}$ 的可行边集合为 FS_1：$\{ab:4,ac:3,ae:5,be:5,cb:3,ce:5,da:3,dc:3,de:5\}$，选择与边 ae 支持度一样的边一起输出作为 $R_1=\{ae,be,ce,de\}$。若不存在，则直接输出边 ae 为 R_1。

(3)用 FS_1 中剩余的边扩展 R_1，分别为子集$(R_1\cup\{ab\})$，$(R_1\cup\{ac\})$，$(R_1\cup\{cb\})$，$(R_1\cup\{da\})$，$(R_1\cup\{dc\})$。仍从支持度最大的开始扩展，首先考虑子集$(R_1\cup\{ab\})$，其伪映射数据库 $SDB|_{R_1\cup\{ab\}}$ 由序列号为 1,2,3,4 的序列组成，可行边集合为 FS_2：$\{ab:4,ac:3,be:4,ce:4,de:4\}$。从 FS_2 中选择与$(R_1\cup\{ab\})$支持度一样的边输出为 $R_2=\{ab,be,ce,de\}$，之后继续用 FS_2 中剩余的边扩展 R_2，直到可行边集合为空。继续用下一个子集扩展 R_1，直至为空。

(4)继续按支持度高低从集合 FS 中选择可行边，按照步骤 1—步骤 3进行扩展，直到可行边集合 FS 为空，整个挖掘过程结束。扩展过程中，例如边 be,ce,de 的伪映射数据库与边 ae 的伪映射数据库一样，都由序列号为 1,2,3,4,6 的序列组成，则直接剪枝，无需再次遍历，以简化挖掘过程。

使用 DFP 算法挖掘序列数据库 SDB 生成的部分偏序如图 4 所示。

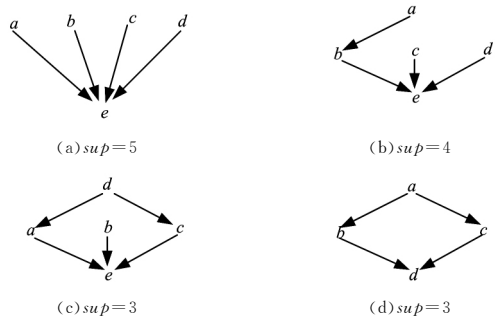

图 4　序列数据库 SDB 部分偏序

DFP 算法是在 Frecpo 算法[15]基础上提出的频繁闭合偏序挖掘算法。与 Frecpo 算法的不同之处如下：

1)针对 Frecpo 算法只能从支持度等于序列数据库大小的可行边开始扩展的限制，DFP 算法提出当序列数据库的可行边集合中不存在支持度等于序列数据库大小的边时，以单个可行边为一个独立的集合，以包含该边的序列形成其伪映射数据库，按照深度优先方式挖掘频繁闭合偏序，使得 DFP 算法适用的数据集更加一般化，同时保证挖掘结果的完整性。

2)在挖掘的过程中，DFP 算法按照支持度从高到低的顺序选择可行边进行扩展，加快了剪枝的速度，并保证了结果的正确性。

3)Frecpo 算法中最耗时的步骤是判断生成的偏序是否是闭合的，因为需要不断地和已经生成的偏序进行传递闭包集合的比较。但只有在伪映射数据库相同时，才会出现偏序不闭合或重复的情况，因此 DFP 算法在剪枝的过程中省去了偏序间传递闭包集合的比较，只判断伪映射数据库中序列的序列号是否相等，若相等，则直接剪枝，无需再次遍历，有效地缩短了算法的运行时间，保证了算法的运行效率。

4　实验设计和结果分析

本节着重介绍实验使用的数据集和实验评价标准，并对实验结果进行对比分析。

4.1　实验数据集

GEF 是 Eclipse 的图形化编辑框架，为开发者提供图形化编辑模型的功能。本实验主要研究 GEF 框架中的 API 用法模式。因此，实验数据集选用 10 个使用 GEF 框架的开源数据集作为代码库，包括 Work flow，Visual OCL，jLibrary (Client)等[16]。该代码库对 GEF 框架中类的覆盖率达到 70%以上，对类中方法的覆盖率在 65%以上。

4.2　实验设计

首先对使用的 10 个源代码库进行转换，提取 API 方法调用序列。通过预处理，筛选出序列中包含以“org. eclipse. gef”开头的方法调用的序列。由于偏序具有反对称性，对于得到的 API 方法调用序列中存在诸如“abbbc”或“abcab”类型的序列(即序列中含有重复项)，不能直接进行挖掘。因此，针对“abbbc”类型的序列，直接把重复的项合并成一项，从而形成序列“abc”。而对于“abcab”类型的序列，从重复项的前一项进行截断，形成两条序列，即“abc”和“cab”，既去除了重复项，又保证了不冗余边的支持度。

以 API 方法调用序列作为聚类算法的输入，对于相似性的计算，使用 R 语言 ngram 包中的 ngram()函数，编程计算得到序列间的相似度，并使用 R 语言提供的层次聚类函数 hclust(d，method＝“single”)，按照最小距离法进行层次聚类；再对得到的类进行筛选，选择序列条数超过 3 条的类作为 DFP 算法的输入，挖掘偏序序列形式的 API 用法模式。

4.3　评价标准

实验采用预处理前后序列的条数及最终聚类个数作为对挖掘过程中使用序列预处理技术结果的评价。在相同的数据集、不同的支持度下，使用 DFP 算法与传统序列挖掘算法 SPADE[17]及频繁闭合序列挖掘算法 BIDE[18]进行 API 用法模式的挖掘，采用所得到的 API 用法模式个数作为评价 DFP 算法挖掘结果简洁性的标准。

4.4　实验结果分析

表 3 列出了预处理前后得到的序列条数及最终的聚类个数。由表 3 可知，通过序列预处理技术，减少了得到的方法调用序列的条数，在此基础上再对方法调用序列进行聚类，增加了挖掘 API 用法模式的支持度。

表 3　预处理前后的序列条数

预处理前的序列条数	预处理后的序列条数	聚类个数
2651	406	86

图 5 示出 DFP 算法、SPADE 算法和 BIDE 算法在相同数据集、不同支持度下挖掘得到的 API 用法模式个数对比结果。

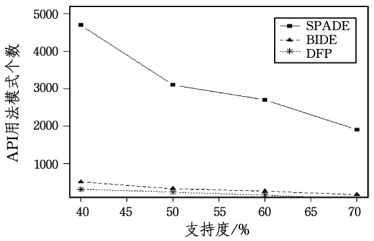

图 5　API 用法模式个数的对比

由图 5 可知，DFP 算法得到的结果集最精简。其原因在于，传统序列挖掘算法 SPADE 挖掘的是支持度大于给定阈值的所有频繁序列，没有考虑序列之间的闭合关系，造成挖掘结果的过度冗余，同时算法的时间复杂度也很高。而频繁闭合序列挖掘算法 BIDE 是对传统序列挖掘算法的改进，即在相同支持度下，频繁闭合序列算法只选择最长的序列，有效地减少了冗余，但候选 API 模式集仍然较大。

频繁闭合偏序综合相同支持度下频繁闭合序列的信息，把支持度相同的相关序列作为一个模式输出，使得结果集更加精简，即偏序中的每一条路径都是一条频繁闭合序列。同时，频繁闭合偏序是相同支持度下最具代表性的偏序，完整地保存了序列所要表达的全部信息；且以 DAG 形式表达偏序，不仅有助于结果的理解，更包含了序列挖掘中没有包含的信息，使得候选 API 用法模式的准确率更高。

结束语　本文提出的基于聚类和偏序序列的 API 用法模式挖掘考虑了不同 API 方法调用序列的使用场景，同时将方法调用之间的顺序关系扩展到方法调用之间的偏序关系，并使用频繁闭合偏序挖掘算法 DFP 进行挖掘。但在 API 方法调用序列的形成过程中没有考虑程序的控制结构，对挖掘结果的精确度有一定的影响，之后将着重增加对控制结构的考虑，以提高候选 API 用法模式的查全率和查准率。另外，本文只是对 API 用法模式进行了挖掘，而如何将挖掘得到的结果运用到软件开发的过程中，是后续需要重点研究的方向。

参 考 文 献

[1] KHATOON S, MAHMOOD A, LI G. An evaluation of source code mining techniques[C]// International Conference on Fuzzy Systems and Knowledge Discovery. 2011:1929-1933.

[2] PICCIONI M, FURIA C A, MEYER B. An Empirical Study of API Usability[C]// Empirical Software Engineering and Measurement. New York: ACM, 2013:35-44.

[3] ROBILLARD M P. What makes apis hard to learn? Answers from developers[J]. IEEE Software, 2009, 26(6):27-34.

[4] THUMMALAPENTA S, XIE V. PARSEWeb: a programmer assistant for reusing open source code on the Web[C]// Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2007:204-213.

[5] LI Z, ZHOU Y. PR-Miner: automatically extractingimplicit programming rules and detecting violations in large software code [C]// European Software Engineering Conference/Foundations of Software Engineering. New York: ACM, 2005:306-315.

[6] XIE T, PEI J. MAPO: Mining API usages from open source repositories[C]// Proceedings of the 2006 international workshop on Mining software repositories. New York: ACM, 2006:54-57.

[7] NGUYEN T T, NGUYEN H A, PHAM N H, et al. Graph-based Mining of Multiple Object Usage Patterns[C]// European Software Engineering Conference/Foundations of Software Engineering. ACM, 2009:383-392.

[8] AKBAR R J, OMORI T, MARUYAMA K. Mining API Usage Patterns by Applying Method Categorization to Improve Code Completion[J]. IEICE Transactions on Information and Systems, 2014, 97(5):1069-1083.

[9] SAIED M A, BENOMA R O, SAHRAOU H, et al. Mining multi-level API usage patterns[C]// 2015 IEEE 22nd International Confe-rence on Software Analysis. 2015:23-32.

[10] 廖兴,尹俊文,蔡放. 基于 Java 语言的抽象语法树的创建与遍历[J]. 长沙大学学报, 2004, 18(4):50-53.

[11] 孙吉贵,刘杰,赵连雨. 聚类算法研究[J]. 软件学报, 2008, 19(1):48-61.

[12] ACHARYA M, XIE T, PEI J. Mining API patterns as partial orders from source code: from usage scenarios to specification [C]// European Software Engineering Conference/Foundations of Software Engineering. New York: ACM, 2007:25-34.

[13] CASAS-GARRIGA G. Summarizing Sequential Data with Closed Partial Orders[C]// 5th SIAM International Conference on Data Mining. 2005.

[14] WANG J, XIE T, ZHANG D, et al. Mining succinct and high-coverage api usage patterns from source code[C]// Working Conference on Mining Software Repositories. 2013:319-328.

[15] PEI J, WANG H, YU P, et al. Discovering frequent closed partial orders from strings[J]. IEEE Transactions on Knowledge and Data Engineering, 2006, 18(11):1467-1481.

[16] ZHONG H, XIE T, PEI J, et al. MAPO: mining and recommending API usage patterns[C]// the 23rd European Conference on ECOOP. 2009:318-343.

[17] ZAKI M. SPADE: An Efficient Algorithm for Mining Frequent Sequences[J]. Machine Learning, 2001, 42(1):31-60.

[18] WANG J, HAN J. BIDE: efficient mining of frequent closed sequences[C]// 20th International Conference on Data Engineering. 2004:79-91.

[19] MICHAIL A. Data mining library reuse patterns using generalized association rules[C]// Proceedings of the 22nd International Conference on Software Engineering. 2000:167-176.

[20] SAHAVECHAPHAN N, CLAYPOOL K. XSnippet: mining For sample code[J]. ACM SIGPLAN Notices, 2006, 41(10):413-430.

[21] HSU S K, LIN S J. MACs: Mining API code snippets for code reuse[J]. Expert Systems with Applications, 2011, 38(6):7291-7301.

(上接第 462 页)

[7] 金宏,王宏安,王强,等. 改进的最小空闲时间优先调度算法[J]. 软件学报, 2004, 15(8):1116-1123.

[8] Semghouni S, Amanton L, Sade B, et al. On new scheduling policy for the improvement of firm RTDBSs performance[J]. Data & Knowledge Engineering, 2007, 63(2):414-432.

[9] 王永炎,王强,王宏安,等. 基于优先级表的实时调度算法及其实现[J]. 软件学报, 2004, 15(3):360-370.

[10] 夏家莉,陈辉,杨兵. 一种动态优先级实时任务调度算法[J]. 计算机学报, 2012, 35(12):2686-2695.

[11] MARCO C, GIORGIO B, LUI S. Handling execution overruns in hard real-time control systems[J]. IEEE Transactions on Computer, 2002, 51(7):110-118.

[12] JENSEN E D, LOCKE C D, TODUDA H. A time-driven scheduling model for real-time operating systems[C]// the 6th IEEE Real-time System Symp, 1985. San Diego: IEEE Computer Society Press, 1985:112-122.

[13] BUTTAZZO G, SPURI M, SENSINI F. Value vs. Deadline scheduling in overload conditions[C]// the 19th IEEE Real-Time System Symp, 1995. Pisa: IEEE Computer Society Press, 1995: 90-99.

[14] BIYABANI S R, STANKOVIC J A, RAMAMRITHAM K. The integration of deadline and criticalness in hard real-time scheduling[C]// the 9th IEEE Real-Time System Symp, 1988. Huntsville: IEEE Computer Society Press, 1988:152-160.

[15] TSENG S M, CHIN Y H, YANG W P. Scheduling value-based transactions in real-time main-memory databases[M]// LIN K J, ed. the First International Workshop on Real-Time Databases: Issues and Applications. Newport Beach: Kluwer Academic Publishers, 1996:111-117.

[16] BURNS A, PRASAD D, BONDAVALLI A, et al. The meaning and role of value in scheduling flexible real-time systems[J]. Journal of Systems Architecture, 2000, 46(4):305-325.

[17] 胥光辉,徐永森. 同步阻塞线程的唤醒问题研究[J]. 计算机科学, 2002, 29(12):49-50.

[18] LI T, XU K, SHEN M, et al. Towards Minimal Tardiness of Data-intensive Applications in Heterogeneous Networks [C]// IEEE International Conference on Computer Communication and Networks (ICCCN), 2016. Hawaii: IEEE Computer Society Press, 2016:1-9.