一种 Java API 文档对异常描述不一致的自动检测方法*

古睿航, 周 宇
(1. 南京航空航天大学 计算机科学与技术学院, 南京 211106; 2. 江苏省软件新技术与产业化协同创新中心, 南京 210023)

摘 要: 应用程序编程接口(application programming interface, API)在软件开发以及代码复用中有着重要作用, API 代码和文档存在的不一致情况会误导 API 的使用者,并降低软件开发效率及其稳定性等。针对 Java API 异常代码及其文档描述不一致的情况,提出了一种基于静态分析代码语法树及方法之间的调用关系的自动检测方法。为验证方法的有效性,利用 JDK 中的 API 源代码包及其相应文档作为测试对象。根据实验结果,该方法的检测结果能达到 71.5% 的准确率以及 85.9% 的召回率,能够较为准确地识别 API 文档对程序异常描述不一致的问题,对 API 文档的编写和维护具有指导性意义。

关键词: API 文档; 程序异常; 约束条件; 不一致性检测
中图分类号: TP311.52 **文献标志码:** A **文章编号:** 1001-3695(2017)07-2032-06
doi:10.3969/j.issn.1001-3695.2017.07.025

Automatically detecting inconsistent description about exceptions in Java API documentation

Gu Ruihang, Zhou Yu
(1. *College of Computer Science & Technology, Nanjing University of Aeronautics & Astronautics, Nanjing* 211106, *China*; 2. *Collaborative Innovation Center of Novel Software Technology & Industrialization, Nanjing* 210023, *China*)

Abstract: Application programming interfaces (API) are playing significant role in software developing and code reusing. However, inconsistent description about exceptions, which exists in API documentation, may mislead developers, and results in lower productiveness or stability of software. Dealing with the inconsistency between Java API code exceptions and the description of their documentation, this paper provided an automatically detecting approach based on static analyzing code abstract syntax tree and invocations among API methods. To investigate the efficiency, this paper evaluated this approach on several packages of JDK source code. Experiment shows a precision rate of 71.5% and a recall rate of 85.9%, which indicates that this approach can detect inconsistent description about exceptions in API documentation with high accuracy. It may be instructive to the writing and maintaining of API documentation.

Key words: API documentation; exceptions; constraints; inconsistency detecting

0 引言

随着软件项目规模的扩大,高效复用代码成为软件行业的追求目标[1],使用 API 是其中一项重要手段[2]。应用程序编程接口(API),一般是一些开放的函数接口,其底层功能的具体实现对开发人员透明,从而使其精力更注重于业务逻辑,进而提高开发的效率。

开发者通过 API 文档了解接口的约束条件,达到正确使用 API 的目的[3]。高质量的 API 文档,应该清楚地描述出其接口被调用时需要满足的相关约束条件[4],其中主要是对参数的约束以及违反约束而抛出的异常。但由于人工撰写文档可能存在错漏、文档和代码更新进度不一致等原因,API 文档的描述和代码功能存在不一致的情况[5]。模糊甚至错误的 API 文档,会造成软件开发者理解困难甚至理解错误[6~8]。例如,在 javax.swing.JTabbedPane 类的 addTab(String title, Component component)方法中,如果传入参数 component 是一个 Window 类的实例,该方法会抛出 IllegalArgumentException 异常;但是在 addTab()方法对应的 Javadoc 文档中对此并没有详细说明。JTabbedPane 类中的其他几个实现类似功能的方法也存在这种文档描述不一致的状况。在 java.awt.JobAttributes 类中的 setPageRanges(int[][] pageRanges)方法中,如果传入参数 pageRanges 为 null,则会抛出 IllegalArgumentException 异常,而该方法文档中也没有对此情况进行说明。

鉴于软件文档在软件工程领域的重要性[9],国内外许多

收稿日期: 2016-09-02; **修回日期:** 2016-11-28 **基金项目:** 江苏省自然科学基金资助项目(BK20151476);国家“973”计划资助项目(2014CB744903);国家“863”计划资助项目(2015AA015303);中央高校基本科研业务基金资助项目(NS2016093);国家自然科学基金资助项目(61202002)

作者简介: 古睿航(1991-),男,四川内江人,硕士,主要研究方向为软件演化;周宇(1981-),男(通信作者),副教授,博士,主要研究方向为软件演化、软件架构、软件验证等(zhouyu@nuaa.edu.cn).

研究者对软件文档中存在的问题进行了探讨,但是关于软件文档中对程序异常及其参数约束描述不一致情况的研究并不是很多。

本文将重点关注 API 文档中对于程序异常及其参数约束的描述,并对其不一致性[10]进行探讨,提出一种自动化识别这种不一致性问题的方法。本文的主要工作如下:

a)针对问题,提出一种基于静态分析、对 Java 项目中 API 文档描述不一致问题进行自动检测。该方法主要通过分析代码语法树,对程序运行过程中可能抛出的异常及其触发条件进行记录,并通过调用关系的分析找到所有关联的异常信息,以异常类型及其条件所涉及参数为关键字,与文档中的约束描述进行匹配,从而检测出文档对异常的描述不一致情况。

b)以 Java 语言为研究对象,将 JDK 的部分源代码作为测试对象进行实验,尝试检测出其文档对程序异常描述不一致问题。被测代码规模约 50 万行,共有 8 千多个方法以及对应的 API 文档。本文从中检测出约 1 000 条描述不一致信息,其 F-measure 值达到 78%。

1　实例分析

为了更好地阐述本文所关注的问题,这里介绍一个典型的 API 文档对程序异常描述不一致问题的实例。

对于 javax. swing. JTabbedPane 类中的 addTab(String title, Component component)方法(以下简称 addTab( )方法),根据 API 文档中的相关描述,如图 1 所示,其第二个参数应该是 Component 对象。但是,如果将一个 JFrame 对象(JFrame 是 Component 的子类)作为参数传入,则会抛出一个 IllegalArgumentException 异常。经过分析发现,在 addTab( )一个调用到 java. awt. Container 类的 checkNotAWindow(Component comp)方法中存在这样的判断:若参数 comp 是 Window 的子类对象,则抛出异常,而 JFrame 正好是 Window 的子类。

add Tab
public void addTab(String title,
Component component)
Adds a component represented by a title and no icon.Cover method for insertTab.
Parameters:
title--the title to be displayed in this tab
component--the component to be displayed when this tab is clicked
See Also:
insertTab(java.lang.String,javax.swing.Icon,Java.awt.Component,java.lang.String,int)

图1　addTab( )方法对应的API文档

这样的参数约束不符合面向对象编程思想中的里氏替换原则(Liskov substitution principle)。如有特殊原因,是否应该在 addTab( )的文档中对这样的约束条件进行描述呢?

根据 addTab( )的调用路径,如图 2 所示,调查与 addTab( )方法存在相同约束的其他方法。可以发现,虽然 addTab( )以及 insertTab( )两个方法中都没有对 Component 这项参数可能引发的问题进行阐述,但在 addImpl( )方法的文档中是存在相关描述的。同时,另外一些调用 addImpl( )的方法,它们的 API 文档中也是有相应描述的。可以推断,addTab( )以及 insertTab( )方法的 API 文档中,是应该存在相关约束描述的。但是其不存在约束描述,这就形成了一个典型的文档对程序异常描述不一致的问题。

2　不一致检测方法

通过对上述实例的研究,可以总结此类问题具有以下几个特点:

a)抛出异常。此类问题是代码在执行过程中抛出异常,造成程序崩溃。上述例子中以 throws 语句来抛出异常,这种异常很容易被静态分析检测出来,是本文重点关注的目标。

b)递归传递。某 API 文档应该描述的异常其异常代码位置可能不在该方法体中,而在其调用方法中。对于某个方法中抛出的异常,不同的调用方法,其文档可能有不一样的描述。

c)文档没有相关记录。由于这些编程接口经过广泛使用,已经相当成熟,本文认为它们的代码没有问题。造成误解的原因是文档对异常及其参数约束条件的描述不充分。

基于此类问题的特点,本文提出的主要思路是,对于某一个 API 方法,分别分析其执行代码和注释文档,从中提取相对应的约束条件,然后将两者进行比对。而对代码中异常信息及其约束条件的提取还会加入方法间调用关系的分析。

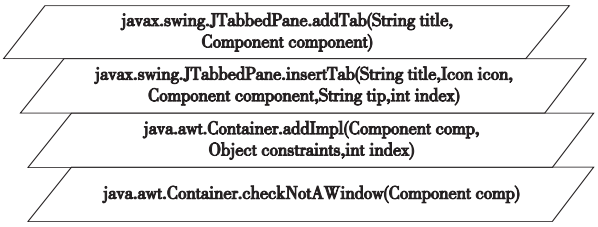

图2　addTab( )方法的调用关系

2.1　数据结构

基于本方法的检测策略,以及便于文章阐述的目的,定义一个基本的元数据结构,用于存储代码中可能产生的异常及其相关的有价值的信息,不妨将其命名为 InfoBox。每个 InfoBox 对应一条抛出异常及其触发条件等相关信息。对于每个 API 方法,可能提取出多个 InfoBox 数据,而这些异常信息都应该在该方法对应的文档中得到表述。其关系应该如以下公式所示。

$$\cup C_i \subseteq C_{\text{doc}}$$

其中:C_i 代表该 API 方法中第 i 个 InfoBox 所包含的异常及其参数约束信息,C_{doc}表示该方法对应的文档中表述的所有异常及其参数约束信息。如果发现某个 InfoBox 不满足条件,则认为存在程序异常与文档描述不一致的情况。

此处所说的文档具体指 Java 源代码中存在的 Javadoc 注释文档,它具有一定的结构特征。基本上,每个 Java 方法对应一块 Javadoc 信息块,而每一个信息块中存在多个条目(directive),每个条目主要分为三个部分:类型描述、关键词、描述语句。其中类型描述一般以@ 开头,常见的有@ param、@ throws 以及@ exception 等,表示该条目主要的内容。关键词随条目类型的不同而具有不同的意义,如@ param 中关键词是相关的参数名,而@ throws 和@ exception 中关键词是抛出异常类型。描述语句是一段自然语言文字信息,是约束条件可能存在的重要位置,也是本文分析的重点。

2.2　执行代码分析

本文采用一种两阶段的处理方法对执行代码进行分析。在第一阶段,暂不考虑调用关系,对所有 API 方法本身进行分析,生成相应的约束条件信息。在第二阶段,针对每个方法,根据调用关系库,遍历匹配,查看其程序异常与文档描述是否一致。

2.2.1　单方法分析

对于每个方法的执行代码,遍历其语法树,定位抛出异常

位置及其触发条件,并进行存储。主要通过以下几个步骤:a)搜索抛出异常;b)回溯中对条件进行补充;c)整理数据,筛选出条件与本方法参数相关的约束。具体的分析方法如算法 1 所示。

算法 1 单方法代码语法树分析

```
Input  AstBlock: code AST of this method; Pars: parameter set.
Output InfoBoxSet: set of (m,c,p), m: trigger; c: set of conditions, p: set of relevant params.
1 GetStatInfoTuples(Statement st)
2 SubTuples←∅
3 if st has sub statements
4   foreach subs of st.subStatements()
5     SubTuples← GetStatInfoTuples (subs)
6 else if st is error trigger statement
7     SubTuples←SubTuples∪{(st,∅,∅)}
8 else if st has conditions
9     SubTuples← GetStatInfoTuples(st.subStatements)
10 foreach tuple in SubTuples
11     tuple.c ← tuple.c∪st.getCondition()
12 returnSubTuples
13 InfoTupleSet ← GetStatInfoTuples(AstBlock)
14 foreach tp in InfoTupleSet
15 tp.p ← tp.c ∩ Pars
```

所需要获取的输出是一个 InfoBox 信息元组集合。对于每个元组(m,c,p)而言,其包含三个主要内容:

a)m 代表当前异常类型。

b)c 代表该触发点的触发条件集合。

c)p 代表该触发点的所有触发条件中,与当前方法参数的交集。

具体地,针对每一条子句,根据其不同的类型进行不同的操作。大概分为三类:

a)可展开语句,即含有子句的语句。对每条子句逐个递归搜索,以完成对语法树的遍历。

b)异常触发语句。对于这样的语句处理方式是,新建一个 InfoBox 元组并将其加入返回集合,此时仅存储异常类型,待回溯时对其触发条件和涉及参数进行补充。

c)条件语句。获取其子句返回的 InfoBox 集合,然后将本句的条件加到每个元组上,并将新的信息集合返回。

在得到了整个信息集合之后,对于集合中每个元组,分析其触发条件,寻找参与条件判断的参数,补充在元组的 p 部分。至此,得到了这个信息集合。

2.2.2 调用关系分析

本文的分析过程中关注并仅分析一种参数直接传递的调用方式。当 funcA(Object p)调用了 funcB(Object q),且 funcA 将其参数 p 作为形参传递给 funcB,称这样的调用关系为直接传递参数的调用关系。其特征在于,对于参与调用关系的两个方法,它们之间传递的参数需要满足相同的约束条件。如果在被调方法中由于参数不满足约束条件而导致异常抛出,那么在主调方法对应的文档中也同样应该对这个异常进行相应的描述。

对于每个方法 met,获取其调用方法相关的信息集合 CalleeSet。按照这样的步骤进行调用关系分析:

a)获取 met 方法的参数列表 P;

b)将 met 方法对应的 CalleeSet 集合初始化为空;

c)如果 met 调用了 met_i,获取 met_i 的形参列表 P_i;

d)对于每个 met_i,如果满足 $P \cap P_i \neq \varnothing$,则将当前的 met_i 放入 CalleeSet 集合中。

通过 CalleeSet,可以结合递归的方式,分析与 met 方法有直接或间接调用关系的方法,提取其中存在的约束条件。

由于 try 语句会捕捉异常情况,使得异常不会被抛出,程序也不会因此中止,所以对 try 语句内部的调用关系不再进行递归分析。技术实现上,采用 eclipse 中的 call hierachy 模块进行调用分析。

2.2.3 异常信息整合

对于每个方法,本步骤将其在一定调用深度内的异常信息全部捕捉,以便于后续与文档之间的对比。主要思路如算法 2 所示。

算法 2 加入调用关系的代码分析

```
Input preMet: to be analyzed; TupleSet: of all methods;CallMap: invocation relationship map.
Output TupleSet: a set of (m,c,p), m: trigger; c: set of conditions, p: set of relevant params.
1 GetTuples(MethodInfo met, int depth)
2 if (depth < =0) return ∅
3 TupleSet ← met.getTulpleSet()
4 MethodInfoSet ← get CalleeSet of met
5 foreach subMet in MethodInfoSet
6     SubTupleSet ← GetTuples(subMet, depth - 1)
7     for each subTuple in SubTupleSet
8       refresh subTuple.p with params in met and subMet
9       refresh subTuple.c with params in met and subMet
10      TupleSet ← TupleSet∪SubTupleSet
11    return TupleSet
12 TupleSet ←GetTuples(preMet, depth)
```

对于每个方法按照这样的步骤分析:

a)提取其在 2.2.1 节中已经存储的异常信息集合。

b)获取当前方法的所有调用方法信息,构造一个调用集合。

c)对于调用集合中的每个方法,逐层递归获取其异常信息。

d)根据主调、被调两个方法之间对应的参数名,更新异常信息中每个元组的条件集合与涉及参数集合。

e)通过设置递归深度阈值,避免调用关系的分析形成闭环。

2.3 匹配方法

关注文档中对程序异常及其参数约束条件的描述。对此,提出这样的假设:

假设在大多数情况下,如果文档对某项程序异常及其参数约束条件进行了描述,则该描述是正确的。

基于这个假设,提出一种启发式的对程序异常及其参数约束条件与文档描述不一致性问题的检查方法,即从文档描述中查找程序异常类型、参数信息以及约束关键词,如果描述中存在这些信息,认为该文档与代码之间是一致的,反之则不一致。

对于关键词集,异常类型以及参数信息可以从之前步骤的中间结果中直接获得,约束关键词的提取则需要对约束条件作进一步的解析。一般地,源代码中的约束条件可抽象为树型结构,可以通过树搜索算法获取关键词。

图 3 为一个典型的例子,说明本方法对条件中约束关键词进行提取的步骤:

a)根据德摩根定律,将否定符号分配到子条件表达式中。

b)根据关系运算符,将当前表达式拆分为两个子表达式,对每个子条件表达式,重复步骤 a),递归地进行判断。

c)当前表达式不可再分,则根据运算符号获取关键词。

在最后一步的提取关键词中,只考虑存在运算符的条件表达式,忽略那些不存在运算符的条件表达式。而对于 API 方法中存在的每一条异常信息及其对应的文档,进行这样的匹配:

a)从异常信息的触发条件中,提取当前异常相关关键词集,包括异常类型、参数信息、约束值等。

b)针对每一条文档条目,判断它是否对步骤 a)中提取的关键词集进行了描述。

c)如果该条目对关键词集描述充分,则认为该条目对这个异常信息进行了正确描述。

d)否则,查看下一条目,直到找到符合步骤 c)的条目,或所有条目搜寻完毕。

e)如果所有条目搜寻完毕,仍然找不到符合该异常信息的正确描述,则认为当前文档对程序异常机器参数约束的描述存在不一致性。

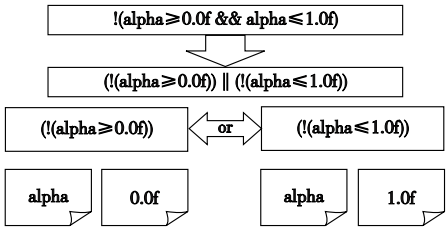

图3　约束条件的树状结构

3　实验

由于 Java JDK 中 API 文档相对完备,为了验证所提出检测方法的有效性以及准确性,将 JDK 部分子项目的源代码为实验对象,对本方法的有效性进行验证。

3.1　准备过程

在准备过程中,通过人工对这些源代码进行分析。对于每个方法,查看其注释文档和执行代码,将该方法中存在的每一条程序异常信息与该方法的注释文档进行匹配,检查文档中是否对这条异常信息的约束条件进行了正确描述,并将结果进行标记,构造一个标准数据集,用于之后比较准确度。

标记的内容是将异常信息按文档是否描述为标准分为正确、模糊、未提及和错误四类。

a)正确是指文档对程序异常及其参数约束描述是正确无误的。

b)模糊是指文档对程序异常及其参数约束进行了描述,但是内容不具体,如 java. awt. Container 中的 add(Component comp, int index)方法,其对应的注释文档中对抛出异常进行如下描述,@ IllegalArgumentException if index is invalid,这里没有说 index 具体应该满足什么要求。

c)未提及是指文档对异常及其参数约束没有相关描述,如示例中 javax. swing. JTabbedPane 的 addTab(String title, Component component)方法,对应文档没有提到 component 不能为 Window 子类这一约束条件。

d)错误是指文档对异常及其参数约束条件描述错误,如 java. awt. event. InputEvent 中 getMaskForButton(int button)方法,其对应文档中指出@ throws IllegalArgumentException if button is less than zero or greater than the number of button masks reserved for buttons,而从代码中可以看出,当 button 小于 0 时就会抛出异常,此时文档中所描述的约束条件是错误的。

以上四类,笔者认为错误和未提及这两种情况属于程序异常及其参数约束与文档描述不一致情况,而由于模糊也对当前的问题进行了阐述,所以不属于不一致。由于大部分约束条件描述集中在@ throws 以及@ exception 这两类条目中,所以重点关注这两类条目。

假设,如果文档描述中有提及参数约束条件,则该约束条件是正确的。基于这个假设,在条目的描述语句中寻找相关的参数名。如果找到,则认为当前文档对这个约束条件是有描述的;否则认为当前文档存在不一致性问题。

在进行正确性标记的同时,也对每条待测对象进行分类处理。基于研究[11],将 API 约束分为空值约束、取值约束以及参数类型约束三类。

a)空值约束指 API 对于参数值为 null 情况下的一些约束条件。通常分为空值允许和空值不允许两种。所谓空值允许,指代码中有考虑到参数值为空时进行相应处理而使得程序不会发生错误或抛出异常;而空值不允许是指如果参数为空值则会发生错误或抛出异常。

b)取值约束指参数需要满足一些取值条件,如方法 java. awt. Component. createBufferStrategy(intnumBuffers, BufferCapabilities caps)对应的文档中要求,numBuffers 取值需大于或等于 1,否则将抛出异常。

c)参数类型约束指参数必须是某种类型,如 java. awt. Container 中 add(Component comp)方法中要求,如果 comp 是 java. awt. Window 的子类,则会抛出异常。

在进行实验之前,提出这样几个问题:

a)关于程序异常及其参数约束与文档描述不一致问题,对于文档正确描述的假设是否合理?

b)对于程序异常及其参数约束与文档描述不一致问题的检测方法,准确率是多少?

c)对于不同类型的不一致性问题,本文方法的准确性是否趋于一致?

d)方法在不同的项目中普适性如何?

本文使用准确率—召回率参数来衡量本文的检测结果。以人工标记为准确结果,其中正确和模糊属于文档—代码一致,而错误和未提及属于文档—代码不一致。相关计算公式如下:

$$\text{precision} = \frac{TP}{TP + FP}, \text{recall} = \frac{TP}{TP + FN}$$

$$F\text{-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

需要解释的是,由于希望检测出不一致情况,故 positive 代表检测结果为该 API 中程序异常及其参数约束与文档描述不一致的情况。

3.2　实验 1

为了回答问题 a)~c),对 java. awt. * 包以及 javax. swing. * 包中的源代码为测试对象进行了实验。经过统计,java. awt. * 包代码行数约 17 万行,共有大概 3 千种方法,意味着需要分析 3 千份 API 文档。这些文档中,@ throws 以及@ exception 条目总数为 1 384 条。javax. swing. * 包一共有 37 万行源代码,7 千多种方法,文档中需要分析的条目总数约 1 千条。统计情况如表 1 所示。

在进行实验之前,首先通过人工标记的方式获取一个标准数据集,以评估本文检测方法的准确度。有三名计算机专业研

究生对代码和文档进行分析,对每种方法中所包含的异常信息是否与文档一致进行评估,同时将这些约束类型基于文献[11]的分类方法分为三类。然后利用所提出的检测方法对数据集进行检测,其结果如表 2 所示。

表 1 实验 1 数据总览

源代码包	源代码行数	方法数	@ throws 条目数	@ exception 条目数
java. awt. *	178. 8k	3 010	961	423
javax. swing. *	372. 8k	7 187	448	533
总计	551. 6k	10 197	1 409	956

表 2 实验 1 效果评估

约束	TP	FP	FN	TN	pre/%	rec/%	*F*-m/%
空值约束	23	45	12	182	73. 2	91. 1	81. 1
取值约束	96	95	57	281	67. 3	77. 5	72. 1
类型约束	20	35	3	19	77. 4	97. 6	86. 3
总计	39	175	72	482	71. 5	85. 9	78. 0

需要说明的是,实验剔除了一些无效数据,这些数据均为从私有方法中提取到的不一致信息。因为私有方法并不是开放的编程接口,所以不需要存在对应的 API 文档。而在实验过程中发现,确实有很多私有方法没有对应的 API 文档,而这类数据可能会造成数据的虚高,故而剔除掉这些数据。

如表 2 所示,本文检测方法平均能够达到 71. 5% 的准确率以及 85. 9% 的召回率,对比表 3 模拟随机选择的评估结果可以发现,本方法各项数据均优于随机选择的结果。对于问题 2,本文的检测方法是有效的。

表 3 模拟随机选择结果

约束	TP	FP	FN	TN	pre/%	rec/%	*F*-m/%
空值约束	66	126	69	101	34. 4	48. 9	40. 4
取值约束	118	172	135	204	40. 7	46. 6	43. 5
类型约束	60	31	63	23	65. 9	48. 8	56. 1
总计	244	329	267	328	42. 6	47. 7	45. 0

从表 4 来看,在归类于不一致的类别中,错误子类仅占不到 1%,而未提及子类在不一致类型中占到 99%,也就是说,绝大多数情况下,当文档对某一项约束进行了描述,那么这个描述一般都是正确的,这证明了本文假设的正确性。本文能够正确识别将近 90% 的未提及子类,同时对于文档和代码一致的情况也能达到 80. 8% 的正确率,这就说明基于本文对于此类文档—代码不一致的假设所提出的检测方案是有效合理的。

表 4 各不一致类型的判断准确率

约束	正确判断	错误判断	正确率/%
正确	409	96	80. 9
模糊	73	79	48. 0
未提及	436	51	89. 5
错误	3	21	12. 5

从表 2 来看,对于不同的约束类型,本文的检测方法均能达到不错的效果,其中对空值约束和参数类型约束的检测结果,其召回率指标能达到 90% 以上。平均来看,各类型的准确率均在 70% 左右,其中类型约束的检测准确率能达到 77. 4%。因此证明,本文的方法对于不同的约束类型均能达到一个不错的预测效果。

3. 3 实验 2

为了验证本方法的普遍适应性,对 JDK 中的另外几个子项目的源代码作为测试对象,进行了拓展性实验。与实验 1 一样,在实验之前,首先对数据进行人工标记和分类。本次实验涉及 30 多万行代码,5 千多种方法及其对应的 API 文档。其中@ throws 和@ exception 条目数共计约 4 千条。此次实验的具体数据如表 5 所示。

表 5 实验 2 数据总览

源代码包	源代码行数	方法数	@ throws 条目数	@ exception 条目
java. util. *	212. 1k	3 476	2 547	290
java. security. *	41. 1k	646	164	421
java. lang. *	89. 1k	1 673	754	335
总计	342. 3k	5 795	3 465	1 046

本次实验的检测结果评估如表 6 所示。可以看到,本方法依然能保持较好的检测效果。其中,java. util. * 包的数据规模最大,分析结果也最好,分别能达到 73. 3% 的准确率以及 92. 6% 的召回率。对另外几个项目而言,召回率表现都不错,javax. xml. * 以及 java. security. * 均能达到 80% 以上。但就准确性而言表现显得不足,其中 java. security. * 项目中,检测准确率仅能达到 60. 9%。深究其原因,发现在这个项目中有很多约束条件存在于其他条目中,而出于简化实验的目的,本文对这些条目没有进行进一步的检测。

表 6 实验 2 效果评估

源代码包	TP	FP	FN	TN	pre/%	rec/%	*F*-m/%
java. util. *	313	114	25	456	73. 3	92. 6	81. 8
java. security. *	39	25	5	89	60. 9	88. 6	72. 2
java. lang. *	84	42	42	58	66. 7	66. 7	66. 7
总计	436	181	72	603	70. 7	85. 8	77. 5

3. 4 总结与展望

以 JDK 的几个子项目的源代码为实验对象进行实验。通过实验,验证了本文关于 Java API 程序异常及其参数约束与文档描述不一致问题假设的正确性,以及基于此假设基础上所提出检测方法的合理性和准确性。同时,对于不同约束条件类型、不同的项目,也验证了本文所提出检测方法的普遍适用性,并收到了良好的效果。

今后将继续深入研究,丰富对更多条目的检测方案。对于文档中描述错误的情况,将尝试添加自然语言分析技术,利用自然语言描述构造约束条件[12],并将之与从执行代码中提取的约束条件进行比对,从而进一步提高对文档—代码不一致性问题的检测有效性和准确性。

对实验内容进行扩充,对更多的 API 及其文档进行检测,探索本方法的通用性,比如对 Java 语言的其他开源项目、对 Android 项目进行拓展性实验。

4 相关研究

API 文档是开发者理解接口功能及其约束条件,进而正确使用编程接口的重要指引,而与此相关的文档自动生成技术[13]、文档追踪技术[14,15]等均受到学界广泛关注,而文档对 API 参数约束的相关研究[10,11,16]相对较少。本章主要回顾一些对软件文档进行的相关研究工作。

Buse 等人[13]提出一种利用静态分析源代码并自动生成 API 文档的方法。该方法对代码中异常语句进行定位,对触发条件进行分析,并转换成自然语言再现,进而生成 API 文档。在有效性实验中,该方法平均能对 40.73% 的 API 生成自动文档;而在对 Weka 和 FreeCol 的源代码实验中,文档生成率高达 88.9% 和 75.2%,证明其方法的可行性。

Tan 等人[10,16]对已有文档中存在的不一致性问题进行了研究。从代码相应的文档中提取约束条件,生成测试用例,通过动态分析方法对代码进行测试,捕获异常,以此验证代码行为是否与文档描述一致。经评估,该方法拥有平均为 99% 的准确度。比较局限的是,该研究关注的是参数值为空的约束条件,对其他类型的参数约束并未进行深入探讨。

Saied 等人[11]就文档中对 API 参数约束的描述进行了相关研究,并将参数约束类型划分成四类。利用符号执行的方法分析源代码的控制流,通过对每个分支参数可能的取值进行分析,最终确定 API 参数的约束条件。另一方面,从源代码 Javadoc 注释中提取相应的约束描述,与上一步骤中获得的约束条件进行匹配,从而检测文档描述的不一致问题。但是这样的方法没有考虑到 API 之间存在的调用关系可能对约束条件的影响。

Zhong 等人[17]对文档中的代码示例问题进行了研究。通过一些启发式算法从文档中抽取代码片段,然后对代码片段中可能存在的问题进行检测,包括代码中关键词是否合法、类名和方法名是否过时或不存在以及代码是否符合语法规范等。该方法主要通过分析源代码、提取关键词,并查看代码示例中关键词是否存在于集合中,以此进行合法性判断。在五个开源项目的检测中,该方法能检测出超过 1 000 项文档错误,同时具有很可观的准确率和召回率。

Manperrus 等人[18]对软件文档的种类进行了广泛的调查,并根据文档的描述内容,对 23 种 API 条目分别进行探讨。本文的研究仅针对其中与参数约束条件相关的条目。

Pandita 等人[12]从自然语言的角度入手,尝试将文档中描述的约束条件转换成逻辑表达式。利用自然语言处理框架 Stanford parser 对文档描述进行处理,分析句子成分以及各成分之间的依赖关系,根据一些启发式规则,生成一阶逻辑表达式。

5　结束语

软件 API 文档作为编程接口的指导文件,在软件复用中扮演着重要的角色。虽然目前有许多关于软件文档方面的研究,但是很少关注到 API 文档中对异常及其参数约束的描述与执行代码中存在不一致的问题。本文提出了一种自动化的检测方法,利用静态分析技术,尝试结合代码语法树分析、调用关系分析,利用启发式关键词匹配的方式,对程序异常及其参数约束与文档描述不一致问题进行检测。同时,本文利用 JDK 源代码中的几个包作为实验对象对本方法进行实验,并获得了良好的效果。在以后的研究中,将进一步尝试利用自然语言分析技术,更准确地提取 API 文档中的约束条件,以达到更好的检测效果。

参考文献:

[1] Kidwell P. The mythical man-month: essays on software engineering [J]. IEEE Annals of the History of Computing, 1996, 18(4): 57-60.

[2] Robillard M P, Bodden E, Kawrykow D, *et al*. Automated API property inference techniques[J]. IEEE Trans on Software Engineering, 2013, 39(5): 613-637.

[3] Dagenais B, Robillard M P. Creating and evolving developer documentation: understanding the decisions of open source contributors [C]//Proc of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2010: 127-136.

[4] Kramer D. API documentation from source code comments: a case study of Javadoc[C]//Proc of International Conference on Documentation. 1999: 147-153.

[5] Shi Lin, Zhong Hao, Xie Tao, *et al*. An empirical study on evolution of API documentation[C]//Proc of the 14th International Conference on Fundamental Approaches to Software Engineering. Berlin: Springer-Verlag, 2011: 416-431.

[6] Bloch J. How to design a good API and why it matters[C]//Proc of Companion to the ACMSIGPLAN Symposium on Object-Oriented Programming System. 2006: 506-507.

[7] Endrikat S, Hanenberg S, Robbes R, *et al*. How do API documentation and static typing affect API usability? [C]//Proc of the 36th International Conference on Software Engineering. 2014: 632-642.

[8] Uddin G, Robillard M P. How API documentation fails[J]. IEEE Software, 2015, 32(4): 68-75.

[9] Li Hongwei, Xing Zhenchang, Peng Xin, *et al*. What help do developers seek, when and how? [C]//Proc of Working Conference on Reverse Engineering. 2013: 142-151.

[10] Tan S H, Marinov D, Tan Lin, *et al*. @ tComment: testing Javadoc comments to detect comment-code inconsistencies[C]//Proc of the 5th International Conference on Software Testing, Verification and Validation. 2012: 260-269.

[11] Saied M A, Sahraoui H, Dufour B. An observational study on API usage constraints and their documentation[C]// Proc of IEEE International Conference on Software Analysis, Evolution and Reengineering. 2015: 33-42.

[12] Pandita R, Xiao Xusheng, Zhong Hao, *et al*. Inferring method specifications from natural language API descriptions[C]//Proc of International Conference on Software Engineering. 2012: 815-825.

[13] Buse R P L, Weimer W R. Automatic documentation inference for exceptions[C]//Proc of International Symposium on Software Testing and Analysis. New York: ACM Press, 2008: 273-282.

[14] Marcus A, Maletic J I. Recovering documentation-to-source-code traceability links using latent semantic indexing[C]//Proc of the 25th International Conference on Software Engineering. 2003: 125-135.

[15] 杨丙贤,刘超. 基于软件结构的文档与代码间可追踪性研究[J]. 计算机科学与探索, 2014, 8(6): 694-703.

[16] Tan Lin, Zhou Yuanyuan, Padioleau Y. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs[D]. Urbana, Illinois: University of Illinois at Urbana-Champaign, 2015.

[17] Zhong Hao, Su Zhendong. Detecting API documentation errors[C]// Proc of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. 2013: 803-816.

[18] Monperrus M, Eichberg M, TEKES E, *et al*. What should developers be aware of an empirical study on the directives of API documentation [J]. Empirical Software Engineering, 2012, 17(6): 703-737.