基于缺陷修复历史的两阶段缺陷定位方法①

王 旭[1,2], 张 文[1], 王 青[1]

[1](中国科学院软件研究所 互联网软件技术实验室, 北京 100190)
[2](中国科学院大学, 北京 100049)

摘 要: 缺陷定位是软件缺陷修复的关键步骤. 随着计算机软件的日趋复杂和网络的迅速发展, 如何快速高效的定位缺陷相关代码成为了一个急待解决的问题. 在研究现有基于信息检索技术的缺陷定位方法的基础上, 综合考虑缺陷修复历史信息, 提出了基于缺陷修复历史的两阶段缺陷定位方法. 该方法不再单一依赖文本相似度, 从缺陷修复的局部性现象入手, 更多的考虑了缺陷修复的历史记录、变更信息及代码特征等因素, 结合信息检索和缺陷预测方法来提高缺陷定位的精度. 最后本文以两个开源项目为例, 验证了方法的可行性和有效性.

关键词: 缺陷定位; 信息检索; 缺陷修复; 缺陷报告

Two-Phase Bug Localization Method Based on Defect Repair History

WANG Xu[1,2], ZHANG Wen[1], WANG Qing[1]

[1](Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Science, Beijing 100049, China)

Abstract: Bug location is the key step in the software defect repair. With the rapid development of complex computer software and network, how to fast and efficiently locate bugs related code has become a pressing problem. This paper studies the existing bug localization method based on information retrieval technologies and a two-phase bug localization method based on defect repair history was proposed. The method pays more attention to the defect repair's local phenomenon. This method does not only rely on text similarity, but also more concern about the defect fix behavior, change of information, code characteristics and other factors. Based on information retrieval and defect prediction this method is proposed to improve the accuracy of defect location. At the end of the paper, two open source projects are conducted to validate the feasibility and effectiveness of the method.

Key words: bug localization; information retrieval; defect repair; bug report

随着计算机软件的日趋复杂和网络的迅速发展, 现代软件的规模和复杂程度不断增加, 对于软件可靠性和可用性的要求越来越高. 软件缺陷伴随在软件的开发及使用过程中, 其数目及分布对软件的可靠性、可用性及成本都产生着重要影响, 甚至决定着软件的成败. 缺陷定位是修复软件缺陷的关键步骤, 缺陷定位的及时性和有效性直接影响软件的可靠性和可用性, 因此软件缺陷定位是软件开发及维护过程中不可忽视的一个重要环节. 在软件开发过程中, 通过对软件缺陷的定位, 软件开发人员可以及时地发现, 修复软件项目所产生的缺陷, 控制软件项目的风险. 缺陷定位本身也是一项软件维护任务, 对一些大型的开源软件社区, 如 Mozilla, Eclipse, Apache 等缺陷管理系统的统计研究发现, 每天都会有上千条的软件缺陷报告被使用者和开发人员提交到缺陷跟踪系统中. 处理这些大量的软件缺陷报告, 给软件开发人员带来了沉重的负担. J. Anvik 等人的研究[1]发现, 对于 Eclipse 项目, 平均每天有大约 200 个软件缺陷报告被提交给缺陷跟

① 基金项目:国家自然科学基金(71101138,91218392,61379046,91318301);北京自然科学基金(4122087)
收稿时间:2014-03-04;收到修改稿时间:2014-04-11

踪系统. 对于日平均缺陷报告上千的大的开源项目, 开发者需要利用现有关于缺陷的信息来定位到需要修改的源代码上, 人工缺陷定位将耗费更多的人时. 这将严重影响软件项目的进度, 增大软件开发的成本. 因此, 开发出高效精确的软件缺陷定位技术对降低开源软件的开发成本, 减少依赖专家知识, 缩短修复时间[2]有着重要的意义.

1　研究现状

缺陷定位方法通常被划分为 2 类, 即静态定位方法和动态定位方法. 动态定位方法通常是在给定输入的条件下, 比较正常运行和失败运行时的控制流的不同[3-9]. 而静态定位方法则是主要依赖源代码的文本信息, 不再要求程序的动态运行信息. 通常是用利用缺陷报告来定位相关的源代码文件. 静态定位方法的优点主要是是不要求一个可运行的软件系统, 可以使用在软件开发和维护的任意阶段; 此外不像大多数的动态定位方法要求的那样, 需要一个能够触发缺陷的测试用例[10].

近些年来, 国内外的研究者尝试将信息检索技术应用在缺陷报告上, 试图定位出与缺陷报告相关的源代码文件. 好多现存的基于信息检索技术的缺陷定位方法都是在特征定位的背景下提出的, Poshyvanyk 等人利用潜在语义索引和概率检索模型提出了 PROMESIR 方法[11]; Gay 等人则在现有信息检索方法上增加了显式相关反馈[12]; Lukins 等人则应用 LDA(潜在狄利特雷分布)方法来定位与缺陷相关的文件和方法[10]; Zhou 等人结合历史缺陷报告信息在信息检索的基础上提出了 BugLocator 方法[13]; D kim 等人采用剔除信息不足的缺陷报告来提高缺陷定位精度的方法[14]. 上述方法普遍存在的一个问题就是缺陷修复历史信息没有利用或利用不充分, 从而导致定位精度不高等问题.

为了解决上述问题, 充分利用已知信息提高缺陷定位的精度, 我们考虑软件缺陷修复的局部性现象: ①历史上经常被修改的文件在未来可能还会经常被修改; ②近期被修改的文件比早期修改的文件更加可能被修改. 基于此. 在本文中我们提出了一种基于缺陷修复特征的两阶段缺陷定位方法, 该方法将传统的基于信息检索的缺陷定位方法作为第一阶段, 并在此基础上第二阶段利用代码修复等特征对第一阶段的排序结果进行重排序进而提高定位的准确率. 该方法不同于传统的基于信息检索的缺陷定位方法, 作者试图从代码修复的历史记录及变更信息中寻找规律, 结合信息检索和缺陷预测方法, 提高缺陷定位的精度.

2　两阶段缺陷定位方法

该方法分为信息检索和缺陷预测 2 个阶段: ①信息检索阶段通过文本相似度检索缺陷报告与源代码文件之间的相关性, 从而构建相关源代码文件集合; ②缺陷预测阶段通过在源代码历史缺陷数据上构建预测模型来预测当前相关源代码文件集合中的文件出现缺陷的概率, 从而实现当前相关源代码文件的再排序. 方法的具体流程图如下图所示.

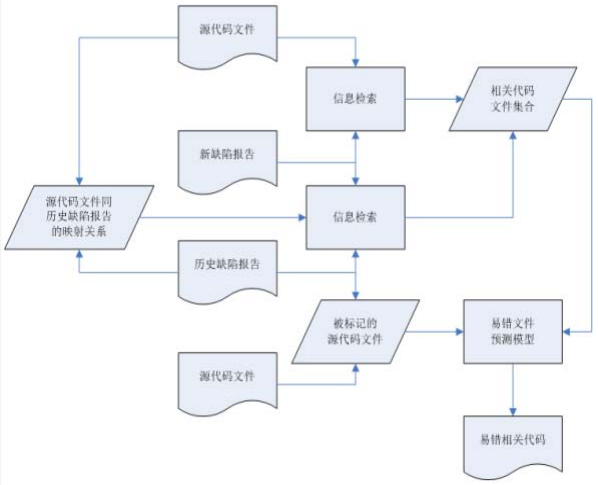

图 1　方法流程图

阶段①用到了传统的信息检索的缺陷定位方法[15]. 传统的信息检索的缺陷定位方法主要包含数据预处理, 构建索引, 构建查询, 检索和排序这 4 个基本步骤. 数据预处理的工作就是对每个源代码文件进行分词, 主要包括去除停用词、词干还原、拆分驼峰词等. 例如“compute”就存在“computes”、“computing”、“computed”等多种变形, 应先进行词根还原. 像是英文中常见的定冠词“a”、“the”就无需放入索引, 需要根据停用词表对这类词进行过滤. 构建索引阶段, 主要就是将第一步分词后的源代码文件构建倒排索引. 每个源代码文件都是由一串词组成, 而缺陷报告通常是若干关键词, 因此如果预先记录这些词出现的位置, 那么只要在索引文件中找到这些词, 也就找到了包含它们的源代码文件. 查询处理主要工作是把缺陷报告

当做查询词, 用它在倒排索引表中搜索相关的源代码文件. 一般我们只抽取缺陷报告中经过预处理后的标题, 详细描述等信息作为查询词. 检索阶段, 将缺陷报告看做查询词, 源代码文件作为文档集, 采用向量空间模型, 逐一计算缺陷报告和源代码文件的文本相似度 *VSMScore*, 并根据相似度结果进行排序.

阶段①除了采用传统的基于信息检索的缺陷定位方法计算缺陷报告和源代码间的文本相似度外, 我们还要通过关键词匹配的方法查找与待定位的缺陷报告相似的已解决的缺陷报告. 由于缺陷报告系统中存放了历史上已解决的缺陷报告与源代码文件的关系, 从而可以计算出待定位的缺陷报告与源代码文件的相关程度 SimiScore. 这里面我们暗含了有一个假设, 即相似的缺陷可能需要修复相似的源代码文件. 最终每个缺陷报告与源代码文件的相关程度得分 Score1 就由 VSMScore 与 SimiScore 加权得到, 计算公式如下:

Score1=α*VSMScore+(1-α)*SimiScore

其中 0≤α≤1, VSMScore 和 SimiScore 均为归一化后的结果, 从而保证 0≤Score1≤1.

阶段①的伪代码如下所示.

输入 源代码集合: S={ s_1, s_2, …, s_m }, s_m 为源代码文件;
历史缺陷报告集合: Bo={$b1$, $b2$, …, br}, br 为源代码文件;
待定位缺陷报告: Bn
1. 对 S、Bo、Bn 进行数据预处理;
2. 创建 S 的倒排索引;
3. 计算 Bn 与{ s_1, s_2, …, s_m }各个源文件之间的文本相似度列表 sim_a = (p_1, p_2, …, p_m);
4. 创建 Bo 的倒排索引;
5. 计算 Bn 与{$b1$, $b2$, …, br}各个历史缺陷报告之间的相似度列表 sim_b = (q_1, q_2, …, q_r);
6. 计算 br 与 s_m 之间的文本相似度列表 sim_{rm}, 若历史缺陷报告 b_r 与 s_m 无映射关系, 则其取值为 0;
7. Bn 与源文档 s_y 的相似度计算式为: $\mathrm{sim_c}(Bn, s_y) = \sum_{x=1}^{r}(\mathrm{sim}_{ry} * q_x)/r$
8. 最终的源文件 s_x 与新缺陷报告的文本相似度为: $\mathrm{sim}(Bn, s_x) = (1-\alpha)*p_x + \alpha*\mathrm{sim_c}(Bn, s_x)$. 其中的 p_x 即为 *SimiScore*, 而 $\mathrm{sim_c}(Bn, s_x)$即为 *VSMScore*, $\mathrm{sim}(Bn, s_x)$即为 *Score1*;
End

阶段②是在阶段①得到的相似度排序基础上, 引入了缺陷预测模型. 研究结果表明, 易错文件的分布符合 2-8 原则, 要识别或者预测这些经常变更的文件, 就需要分类或者回归技术来进行判定. 分类技术和回归技术都属于学习问题, 在学习问题上常常将分类技术划分到解决模式识别问题上, 即用于估计指示函数集合(指示函数即只有 0 或 1 两种取值的函数); 而回归技术则用于解决回归函数估计问题, 即用于估计实函数. 两类技术都可以用来指导测试和其他质量保证活动的计划及执行, 节省昂贵的测试及评审的时间和成本[16]. 构建缺陷预测模型的时候, 由于不同度量元之间存在数据相关性等问题, 通常使用主成分分析技术压缩和降低这些度量元的维度, 选取正交的度量元来估算软件缺陷. 本文使用的基本特征分别是文件长度、文件被修改次数、文件最近一次修改的时间、文件中各函数的平均圈复杂度、文件被其他文件引用的次数(入度)、该文件引用其他文件的次数(出度). 我们将原始数据的各特征项都处理到[0, 1], 避免数值范围较大的属性控制数值范围较小的属性. 构建预测模型中我们采用 SVR 作为分类器, 值得注意的是我们的方法与选取的机器学习技术是独立的.

阶段②使用到的缺陷预测模型依赖阶段①确定的相关源代码集合, 得到的预测结果 *Score*2 用来表征源代码文件的易错程度. 最终每个源代码文件的得分计算公式如下:

Score= (1-*β*)**Score1*+*β***Score2*

其中 0≤*β*≤1, *Score1* 和 *Score2* 均为归一化后的结果, 从而保证 0≤*Score*≤1.

阶段②的伪代码如下所示.

输入 阶段①获得的各个源文件的文本相似度值:sim = (sim_1, sim_2, …, sim_m), 待定位的缺陷报告: Bn
1. 获得 S={ s_1, s_2, …, s_m }各个文档的:
a. 源文件长度{a_1, a_2, …, a_m};
b. 源文件被修改次数{b_1, b_2, …, b_m};
c. 源文件最近一次修改的时间{c_1, c_2, …, c_m};
d. 源文件中各函数的平均圈复杂度{d_1, d_2, …, d_m};
e. 源文件被其他文件引用的次数(入度){e_1, e_2, …, e_m};
f. 该源文件引用其他文件的次数(出度){f_1, f_2, …,f_m};

2. 将上述特征, 作为缺陷预测模型使用的特征, 对源文件进行回归.

3. 最终的源文件与新缺陷报告的文本相似度为: $Score = (1-\beta)*Score1+\beta*Score2$;

4 源代码文件按照 *Score* 值排序, 返回得分最高的前 *n* 个源文件, 作为最终的定位到的文件列表.

End

3 实验与结果分析

大型的开源软件项目, 例如 Tomcat, Android 都使用软件缺陷跟踪系统, 如 Bugzilla, JIRA 等开放缺陷跟踪系统, 来记录和追踪软件产品的缺陷. 这些缺陷跟踪系统负责记录、存储和跟踪开发人员和用户提交的缺陷报告, 包括提交的缺陷标题、缺陷的描述、提交人、提交缺陷的日期及提交者定义的严重程度等信息. 当一个缺陷报告被提交到缺陷跟踪系统以后, 软件项目的管理人员通过查看待处理的缺陷报告, 手动的将缺陷报告分发给相应的开发人员进行修复. 开发人员可以对缺陷报告发表评论, 提供修复该缺陷的方案等. 为了验证方法的有效性, 本文选取了 Tomcat6、Android 项目, 正是由于这两个开源项目都是为业界所熟知的, 而且都具备完善的缺陷追踪系统和版本控制系统.

对于每个系统, 我们都要从缺陷追踪系统中收集缺陷报告, 为了评估本文提出的两阶段缺陷定位方法, 我们只收集已经修复完成的缺陷. 为了建立缺陷报告与源代码文件之间的映射关系, 需要从文件日志上找出变更对应的缺陷 ID, 关联缺陷报告的描述和修改前后的源代码[17]. 图 2 是以一个 Tomcat 的缺陷报告为例, 展示了缺陷报告与源代码文件之间的映射关系, 并以 xml 格式格式保存起来.

图 2　缺陷报告与源代码文件间的映射关系图

实验对比了四种缺陷定位方法, 方法 1 采用的是传统的基于信息检索的缺陷定位方法. 方法 2 在方法 1 的基础上加入了历史已修复的缺陷报告与源代码文件间的关系. 方法 3 在方法 2 的基础上引入了缺陷预测模型. 方法 4 即本文提出的两阶段缺陷定位方法, 在方法 3 的基础上同时考虑了缺陷修复行为的特征及缺陷报告与源代码文件的文本相似度.

为了评估本文提出的基于缺陷修复历史的两阶段缺陷定位方法的有效性, 在参考缺陷定位领域相关论文的基础上, 我们引入了如下 3 种指标:

TopN: 表征缺陷报告定位成功的概率. 在给定一个待定位的缺陷报告基础上, 如果返回的前 N 个源代码文件包含至少一个与待定位的缺陷报告相关的应该修复的缺陷, 那么我们认为定位成功.

MRR: 是把相关源代码文件在缺陷定位方法给出结果中的排序取倒数作为它的准确度, 再对所有的问题取平均.

MAP: 表征定位方法在全部相关源代码文件上性能的单值指标. 缺陷定位方法检索出来的相关源代码文件越靠前, MAP 值就会越高. 如果缺陷定位方法没有返回相关源代码文件, 则准确率默认为 0.

实验 1 在 Tomcat6 数据集上, 使用的原始缺陷报告有 610 个, 根据 2013 年最新研究[18]经过筛选之后确认其中 51 个缺陷报告描述的为真实缺陷. 将这些缺陷报告分成两组, 第一组 20 个作为历史数据集来训练缺陷预测模型; 其余 31 个放在第二组作为实验中需要进行缺陷定位的缺陷报告. 实验结果如下表所示.

表 1　Tomcat6 项目上四种实验方法对比

Tomcat6	方法 1	方法 2	方法 3	方法 4
TOP1	0.38	0.50	0.17	0.52
TOP5	0.63	0.71	0.50	0.72
TOP10	0.79	0.71	0.67	0.75
MRR	0.51	0.59	0.32	0.59
MAP	0.56	0.61	0.31	0.61

实验 2 在 Android 数据集上, 使用 57 个缺陷报告, 其中 28 个放在一组作为历史缺陷报告来训练缺陷预测模型, 其余 29 个放在第二组作为实验中需要进行缺陷定位的缺陷报告. 由于该数据集没有对缺陷报告进行人工筛选, 因此 Android 数据集的质量相对 tomcat6 数据集质量要差一些. 实验结果如下表所示.

表 2 Android 项目上四种实验方法对比

Android	方法 1	方法 2	方法 3	方法 4
TOP1	0.09	0.11	0.05	0.12
TOP5	0.17	0.19	0.08	0.20
TOP10	0.18	0.19	0.07	0.20
MRR	0.13	0.14	0.05	0.15
MAP	0.11	0.12	0.06	0.13

在实验的过程中, 为了取得理想的效果, 需要根据实际情况设定超参 α、β 的值. 超参 α、β 分别负责调整阶段一中 *VSMScore* 和 *SimiScore* 的权重, 阶段二中 *Score*1 和 *Score*2 的权重. 我们进行了多次实验来评估超参对结果的影响, 我们发现在初期随着超参值的增加缺陷定位性能会提高, 但是到达某个拐点之后, 随着超参值的增加缺陷定位性能会下降. 图 3、图 4 显示了在 Android 上超参 α 及 β 对缺陷定位性能的影响. 为了获得性能最优值, 我们将两阶段方法中 α 设为 0.2, β 设为 0.3.

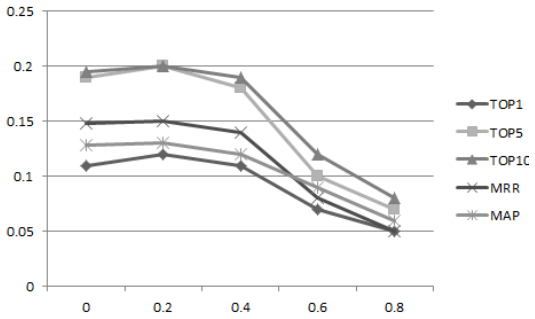

图 3 超参 α 对定位性能的影响关系图

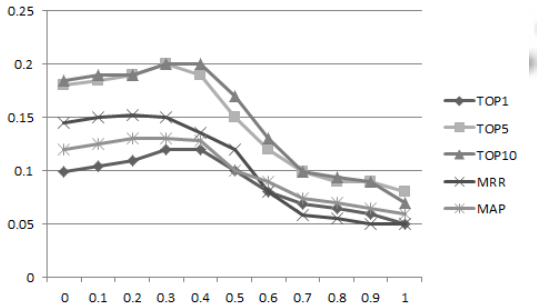

图 4 超参 β 对定位性能的影响关系图

可以看出, 单纯的使用本文所提出的特征的效果有限, 而使用两阶段方法, 能够有效改善缺陷定位方法.

4 结语

对于大型和持续演化的软件系统, 项目组每天会收到大量的缺陷报告. 近些年, 许多研究者尝试使用自动化的缺陷定位方法来搜寻同缺陷报告相关的源代码文件. 静态缺陷定位方法由于其适用范围广, 使用要求低的特点, 在近年的缺陷定位研究中受到了广泛的重视. 本文针对传统的基于信息检索的缺陷定位方法的不足, 从缺陷修复的特点出发, 提出了一种基于缺陷修复历史的两阶段缺陷定位方法. 在多个数据集的实验中, 该算法都与传统的基于信息检索技术的缺陷定位算法相当或略优, 特别在准确率方面, 方法的优势更加明显.

该方法的定位精度目前还是存在着依赖具有良好的命名规范的源代码和高质量的缺陷报告等问题. 在未来的研究中, 我们期待从两方面改进现有方法: 一方面, 寻找更多更有效的特征集来提高定位的精度, 并且扩大实验范围, 在规模更大的软件系统中去验证方法的可用性和有效性; 另一方面, 尝试着加入动态定位方法并形成自动化的动静结合的缺陷定位工具, 相信会显著提高缺陷定位的精度和效率.

参考文献

1 Anvik J, Hiew L, Murphy GC. Who should fix this bug? Proc. of the 28th International Conference on Software Engineering. May, 2006. 361–370.

2 Nguyen AT, Nguyen TT, Al-Kofahi J, et al. A topic-based approach for narrowing the search space of buggy files from a bug report. Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering. Nov, 2011. 263–272.

3 Abreu R, Zoeteweij P, Golsteijn R, et al. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009, 82(11): 1780–1792.

4 Jones JA, Harrold MJ. Empirical evaluation of the Tarantula automatic fault-localization technique. Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering. 2005. 273–282.

5 Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. Proc. of the 24th International Conference on Software Engineering. May, 2002. 467–477.

6 Liblit B, Naik M, Zheng AX, et al. Scalable statistical bug isolation. Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. June, 2005. 15–26.

7 Liu C, Fei L, Yan X, et al. Statistical debugging: a hypothesis testing-based approach. IEEE Trans. on Software Engineering, 2010, 32 (10): 831–848.

8 Dallmeier V, Lindig C, Zeller A. Lightweight bug localization with AMPLE. Proc. of the 6th International Symposium on Automated Analysis-driven Debugging.

9 Renieres M, Reiss S. Fault localization with nearest neighbor queries. Proc. of the 18th IEEE Int. Conf. on Automated Software Engineering. 2003. 30–39.

10 Lukins S, Kraft N, Etzkorn L. Bug localization using latent Dirichlet allocation. Information and Software Technology, 2010, 52(9): 972–990.

11 Poshyvanyk D, Guéhéneuc Y, Marcus A, et al. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. on Software Engineering, 2007, 33(6): 420– 432.

12 Gay G, Haiduc S, Marcus A, et al. On the use of relevance feedback in IR-based concept location. Proc. of the 25th IEEE International Conference on Software Maintenance. Sep, 2009. 351–360.

13 Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. Proc. of the 34th International Conference on Software Engineering. June, 2012. 14–24.

14 Kim D, Tao Y, Kim S, et al. Where should we fix this bug: A two-phase recommendation model. IEEE Trans. on Software Engineering, 2013, 39(11): 1597–1610.

15 Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. Proc. of the 8th Working Conference on Mining Software Repositories. May, 2011. 43–52.

16 王青,伍书剑,李明树.软件缺陷预测技术.软件学报,2008,19(7):1565–1580.

17 Bachmann A, Bernstein A. Software process data quality and characteristics: a historical view on open and closed source projects. Proc. of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops. 2009. 119–128.

18 Herzig K, Just S, Zeller A. It's not a bug, it's a feature: How misclassification impacts bug prediction(accepted for ICSE, to appear), 2013.