计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200100075

基于脚本预测和重组的内存泄漏测试加速技术

李　吟[1,2]　李必信[1]
1 东南大学计算机科学与工程学院　南京 211189
2 江苏自动化研究所　江苏 连云港 222006
(leein121999@126.com)

摘　要　内存泄漏是云应用、Web服务、中间件等各类连续工作型软件中的一种常见缺陷,它会导致程序运行速度减慢、资源耗尽崩溃等软件稳定性问题。现有测试一般以较长周期运行测试用例来检测泄漏缺陷,用于检测泄漏的测试用例通常需要运行数小时以上才能产生足以暴露泄漏的内存表现。整个测试过程代价高昂,若对测试用例不加筛选,可能会耗费大量的时间在暴露泄漏可能性低的测试用例上,降低了泄漏发现的效率。为了弥补现有技术的不足,并解决Java Web程序长时间运行的内存泄漏缺陷不易发现、不易诊断及不易修复的难题,文中对内存泄漏的发现技术进行了研究,提出了基于机器学习的内存泄漏测试脚本预测方法,通过构建内存特征模型,对存在内存泄漏的脚本进行训练及预测,基于训练的模型进行脚本内存泄漏风险值预测,并给出相应的参数打分,以指导后续的脚本重组,从而预测获取更可能造成内存泄漏的功能测试脚本。同时,提出了脚本重组优化方法,改善其缺陷揭示能力。对预测和重组后的脚本进行优先测试,可以加速泄漏缺陷的发现。最后通过案例验证表明了所提框架具有较强的泄漏发现能力,重组优化后的测试脚本在发现缺陷的速度方面比普通脚本高出一倍以上,从而缩短了内存膨胀问题的暴露时间,达到了提高测试效率以及保障软件质量的目的。

关键词:内存泄漏;泄漏预测模型;机器学习;测试脚本;脚本重组
中图法分类号　TP311.5

Memory Leak Test Acceleration Based on Script Prediction and Reconstruction

LI Yin[1,2] and LI Bi-xin[1]
1 School of Computer Science and Engineering,Southeast University,Nanjing 211189,China
2 Jiangsu Automation Research Institute,Lianyungang,Jiangsu 222006,China

Abstract　Memory leak is a common defect in continuous working software,such as cloud applications,web service,middleware,etc. It can affect the stability of software applications,lead to run in bad performance and even crash. To clearly observe memory leaks,the test cases toward them need to execute longer time in order to generate significant memory pressure. The cost of memory leaks testing is expensive. If the execution orders of test cases are not optimized,we may waste lots of time on the test cases that are not likely to reveal faults before finding test cases that really containing memory leaks. This seriously reduces the efficiency of fault discovery. In order to make up for the shortcomings of the existing technology and solve the problems of the memory leak of Java Web program while running for a long time,which is not easy to find,diagnose and repair,this paper studies the memory leak detection technology,proposes the memory leak test script prediction method based on machine learning. The method trains and predicts the script with memory leak by building the memory feature model. Then,based on the training model,it predicts the risk value of script memory leak,and gives the corresponding parameter scores,to guide the subsequent script reorganization,can predict and obtain the function test script that is more likely to cause memory leak. At the same time,a script reorganization optimization method is proposed to improve its defect revealing ability. Priority testing of predicted and recombined scripts can accelerate the detection of leakage defects. Finally,a case study shows that the proposed framework has strong leak detection ability. The speed of defect detection of the optimized test script can be more than twice as fast as that of the common script,thus accelerating the exposure time of memory expansion problem,achieving the purpose of improving test efficiency and ensuring software quality.

Keywords　Memory leak,Memory leak prediction model,Machine learning,Test script,test script reconstruction

到稿日期:2020-01-10　**返修日期**:2020-06-28　**本文已加入开放科学计划**(OSID),请扫描上方二维码获取补充信息。
通信作者:李必信(bx.li@seu.edu.cn)

1 引言

内存管理是大型应用软件需要考虑的一个重要因素，它直接影响应用软件的稳定性和效率，不当的内存管理可能会造成内存泄漏。内存泄漏是一种广泛存在于云应用、Web服务、中间件等各类连续工作型软件中的一种典型缺陷。其表现一般是，随着系统的不断运行，软件的内存消耗越来越大，进而影响系统的性能，逐渐无法对外提供服务，甚至会由于资源的耗尽导致系统宕机[1]。这类软件执行表现退化的现象也称为软件老化(Software Aging)[2]。内存泄漏引起的软件老化对于高质量应用系统的可靠性有严重影响。在很多关键领域，软件难以承受不时宕机、重启带来的后果，因此必须对内存泄漏缺陷及其引起的软件老化问题进行详尽的测试[3-4]。

现有的内存泄漏研究的主要工作大多集中在静态测试[5-7]，以及发现泄漏现象后的问题诊断等方面[8-18]。受制于程序语言的不断动态化，静态测试在Web应用等系统上已经越来越难以发挥作用。现有的主流内存泄漏测试研究大多关注动态测试，而在动态测试方面，除了发现泄漏后的分析诊断，如何优化发现过程也是一个值得关注的问题。

内存缺陷是一种非功能性缺陷，具有较强的隐蔽性，导致内存泄漏的测试代价高昂。泄漏的过程非常缓慢，暴露一个内存泄漏现象经常需要运行测试数小时，甚至数天。一个Web应用开始测试时是正常的，运行一段时间后会突然宕机。现有的测试技术在如何测试内存泄漏方面缺乏导向性，无法识别不同测试脚本造成内存泄漏的风险，只是按照功能或者任意的顺序执行测试用例[19-20]。这样可能会耗费大量的时间在检测泄漏风险非常低的测试执行上，需要执行一大批测试用例后才能发现泄漏，大大降低了泄漏发现的效率。另一方面，内存性能的测试往往从功能测试结束开始，虽然在功能测试阶段已经累积了一批用于检测Web应用功能正确性的脚本，但测试人员仍需要为这些脚本增加循环来获得面向内存性能的新脚本，以持续长周期的反复操作。该过程繁琐且费时，因此迫切需要一种能够自动构造循环脚本的方法来优化测试。

为解决上述问题，本文提出了一种基于脚本预测和重组的内存泄漏测试加速技术。本文的主要创新点包括：

(1)提出了一种面向Web应用的测试脚本内存泄漏风险预测方法，能够对引入循环反复前的测试脚本进行内存泄漏的风险评估。该方法提取普通功能测试脚本中的内存相关特征，将特征转化为特征向量，并通过机器学习算法进行脚本预测，可预测引入大剂量循环后导致内存泄漏的可能性。

在后续的内存性能测试中，首先测试风险高的测试脚本以加快内存泄漏发现的速度。

(2)提出了一种脚本自动重组优化方法。对于上一步识别有风险的功能测试脚本，自动通过代码变换技术将其加入循环，并对关键数据进行参数化。如此可无需测试人员编码，直接获得能够进行长周期反复操作的测试脚本程序。

本文选择某型号海军后勤综合保障系统软件作为待测对象来进行方法验证，证实了本文方法具有可行性和有效性。

2 研究背景和相关工作

现有内存泄漏分析测试技术的主要工作可分为静态和动态两个方面。

静态方法不需要运行程序，通过基于源代码的静态测试来检测内存泄漏缺陷。在此方面，研究者提出了基于溢出分析、值流分析、指针分析等多种检测方法，取得了一定成效。然而，静态方法的一个显著缺陷是，对于动态性较强的语言可用性偏弱。例如，大多数静态方法难以应用于基于Java，PHP，Python等动态性较强语言实现的Web应用程序上[3-6]。

动态方法通过监控和执行程序的方法检测内存泄漏(如工具Rational Purify，Valgrind)。目前，该领域的主要研究集中在发现内存泄漏后的分析诊断方面，主要的诊断思路有两类。一类是基于堆内存统计和比较，例如统计一段时间内增长最快的对象或数据结构，或者比较指定周期开始和结束时的内存变化来判定泄漏对象，甚至在内存拓扑图上进行大规模同构数据结构的挖掘来定位泄漏源[7-11]。另一类是基于对象的老化信息。所谓老化对象即虽然存活但长期不用的对象，一些研究以对象从上一次访问到当前时刻之间的内存访问次数为衡量老化的标准[12]，还有一些研究以上一次访问到当前时刻之间的垃圾收集次数来判定老化对象[13]。通过将对象按老化程度排序，可以推测可疑的泄漏源[14-17]。

近年来，如何通过测试发现存在泄漏的执行也得到了越来越多的关注。Shahriar等[18]针对Android应用提出了一种基于变异的泄漏测试方法，构造一些大图、大文件等来检测泄漏，但该方法只针对非常少的几种泄漏模式，缺乏系统性。Yan等[19]提出了一种构造事件循环的方法来发现Android GUI应用中的泄漏执行。但这些方法缺乏导向性，测试的效率不高。特别是对于内存泄漏的发现，一个测试用例通常需要执行较长的时间才能发现泄漏。例如，假设一次界面操作能触发10 kB的内存泄漏，完成操作需要1s，能够观测并判定泄漏的内存规模至少为100 MB，那么一个能够揭示内存泄漏的测试执行需要运行大约3h。如果不带导向性地随意执行测试用例，将会浪费大量的测试资源。

为解决上述问题，本文以Web应用为目标，提出了一种基于机器学习的测试用例(脚本)泄漏风险预测方法，能够预测最可能造成泄漏的脚本。将其优先执行，可以加快发现泄漏的速度。

现有研究的另一不足是未能解决内存泄漏测试脚本的构造问题。那些产生大量循环的内存性能测试脚本大多是手工构造的，其构造过程占用了测试人员的大量时间。为解决该问题，本文基于程序转换技术，提出了一种脚本自动重组方法，能够根据普通的Web应用功能测试脚本(Selenium脚本[21])自动生成带大剂量循环的脚本。如此，在测试人员录制脚本完成功能测试后，内存性能测试可以以较高的自动化程度完成。该算法会在功能测试脚本的基础上，自动选择高

风险脚本进行循环化重组，进而自动执行新脚本以发现泄漏。其测试过程缩短了人工时间，降低了测试成本。

3　基于脚本预测和重组的测试加速框架

在 Java Web 程序测试过程中，普通测试用例集中测试用例的数量往往非常庞大，为了找到应用程序中存在的问题，通常需要将这个测试用例集合中的所有测试用例全部执行，并且每个测试用例需要运行多遍以找出应用程序存在的问题。例如，若要找到某一 Java 应用中的内存膨胀问题，当执行一个测试用例后，观察内存发现并未有大量增加，则并不能判断出此操作是否会导致内存膨胀的发生，需要将该测试用例运行多次（如 10 次、100 次等）才可观察到累计的内存增加，从而人工判断该操作是否会导致内存泄漏。当需要对整个测试用例集合进行测试时，需要对每个测试用例都运行相同的轮次才可以发现程序中存在的内存膨胀现象，使得在测试的过程中需要在检测与泄漏无关的测试执行上消耗大量的时间，并且对膨胀的判断也大大增加了人力操作（在 Java 语言中不易对是否存在泄漏行为进行自动判定），导致整个内存泄漏发现测试的效率低下。

为此，本文提出了基于内存膨胀预测模型的测试用例优先级框架，针对内存泄漏对用户所提供的测试用例集合进行筛选。本节将对提出的基于脚本预测和重组的测试加速框架进行总体描述，该框架分为两个部分：训练部分和测试部分，如图 1 所示。首先，找出 Java Web 应用上已知反复执行后是否会引发内存泄漏问题的 n 个功能测试脚本作为训练脚本，提取这些脚本中的内存泄漏相关程序执行特征，通过机器学习，训练得到从特征到泄漏风险评估值的预测模型。其次，执行不含循环的现有功能测试脚本，获得脚本的执行特征向量，基于内存泄漏预测模型判定脚本引入大量循环后造成泄漏的可能性，并依据该可能性对功能测试脚本进行排序。最后，对高风险脚本进行重组优化，获得带循环的高缺陷揭示能力脚本并对其进行优先测试。

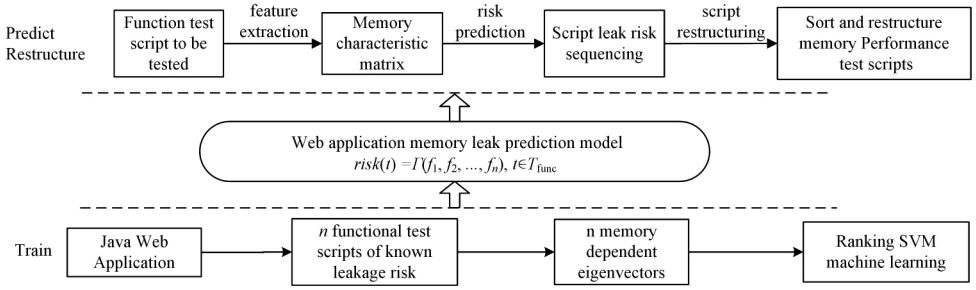

图 1　内存泄漏测试加速框架

Fig. 1　Memory leak test acceleration framework

3.1　内存泄漏风险的代码特征分析

由于功能测试脚本单轮次执行无法引发明显的内存波动，且内存使用可能受垃圾收集等活动的影响，因此很难直接从其内存表现判断其循环执行后引发内存泄漏的可能性，只能通过提取一些执行特征来进行智能化的预测。

与内存泄漏风险相关的代码特征主要有 3 种：对象申请与释放操作、资源申请与释放动作、脚本执行时间长短，如表 1 所列。

表 1　内存泄漏风险相关程序执行特征分类

Table 1　Memory leak risk related procedure execution feature classification

Feature classification of memory leak	Feature parameter
Object application and release	new newarray
Resource application and release	free open××× close×××
Script execution time	elapsedtime

特征描述如下。

（1）对象申请与释放：一个脚本执行过程中申请的对象越多，脚本可能使用的内存越多，达到的内存峰值越大，内存泄漏风险越高。因此，本文将对象申请指令 new，newarray 的出现次数以及相关对象执行过程中被 JVM 释放的对象个数作为一组内存泄漏相关特征。

（2）资源申请与释放：程序运行过程中存在很多资源（如文件、数据库、图片等）的申请与释放操作，这些操作申请的资源尽管不直接是内存空间，但可能导致内存使用增多。例如，打开文件越多，程序占用的内存也会越多。为此，本文将资源的申请次数与释放次数作为重要的执行特征。本文将所有名如 open×××的方法调用操作都当作资源申请，将名如 close×××的操作均当作资源释放。

（3）脚本执行时间：一般情况下脚本执行越长，涉及的内存操作越多，内存泄漏风险越高，因此功能测试脚本的执行时间也可以作为特征加入到训练过程中综合考虑。

3.2　测试脚本集与内存特征向量

本文的预测和重组涉及两个测试集：训练测试集和待预测重组测试集，分别用 T_t（train tests）和 T_c（candidate tests）表示。

$$T_t=\{t_{t1},t_{t2},\cdots,t_{tk}\} \tag{1}$$

$$T_c=\{t_{c1},t_{c2},\cdots,t_{cl}\} \tag{2}$$

两个测试集分别有 k 个和 l 个元素，$T_t\cap T_c=\varnothing$。两个测试集的合集为所有功能测试脚本的集合。

$$T_{\text{func}}=T_t\cup T_c \tag{3}$$

在训练测试集中，已知测试脚本执行后是否会造成内存泄漏现象，执行该测试脚本 100 次，根据获得的泄漏内存数值

计算对应的泄漏风险评估值 γ，γ 的取值为 0～20。即对于训练测试集，有一个已知的风险度标签函数 $R, \forall t \in T_t, R(t) \in \{0, \gamma\}$。

对于待预测重组的测试集，其中的测试脚本未来反复执行时可能引发的泄漏风险未知，因此将通过机器学习获得一个预测函数 $risk(t)$。泄漏可能性越高，$risk(t)$ 的取值就越大。

泄漏风险的预测基于功能测试脚本的执行特征实现。对训练集和待预测重组集中的每个测试脚本 t_i，都可以获得一个特征向量 $\boldsymbol{F}(t_i)$。

$$\boldsymbol{F}(t_i)=[f_1, f_2, f_3, f_4, f_5, f_6] \tag{4}$$

其中，f_1-f_6 分别对应如下特征：

$F=$[新分配对象数，新分配数组数，对象释放数，资源申请数，资源释放数，脚本执行时间]

特征的收集主要通过对 Java Web 应用进行插桩实现。在 Web 容器(本文主要是 Tomcat)加载服务器端 Java 类的过程中，为类中对象创建、函数调用等进行插桩，以统计对象申请与释放的次数、资源申请与释放的次数等信息。Web 应用程序通过 Selenium 测试脚本驱动，脚本执行将被监控，在脚本执行结束后所有特征数据将被记录到文件中以便后续处理。

3.3 内存泄漏预测模型训练

本文通过排序学习(learning to rank)获得内存泄漏预测函数 $risk(t)$，排序学习采用 RankingSVM 算法实现，其输入是一组具有偏序关系的训练样本集以及每个样本的特征向量，输出是一个线性预测函数，该函数为每个参与排序的样本计算一个数据值，按该数据值进行的排序与训练过程中给出的偏序一致。通过训练所得的预测函数，可对新样本进行预测排序。

给定训练样本集测试集 T_t，RankingSVM 排序学习算法首先根据已知的每个训练测试脚本的泄漏风险评估，获得所有训练测试用例之间的偏序关系。

$$t_{t1} < t_{t2} \text{ iff } R(t_{t1}) < R(t_{t2}) \wedge t_{t1} > t_{t2} \text{ iff } R(t_{t1}) > R(t_{t2}) \tag{5}$$

若 $t_{t1} > t_{t2}$，则算法将组合$\langle t_{t1}, t_{t2}\rangle$映射到分类“+1”，而若 $t_{t3} < t_{t4}$ 则算法将组合$\langle t_{t3}, t_{t4}\rangle$映射到分类“−1”，由此可将排序问题映射为分类问题，如图 2 所示。

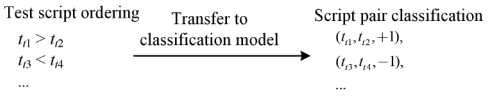

图 2 训练数据转换示意图

Fig. 2 Training data transformation

RankingSVM 算法将基于 SVM 支持向量机技术，获得一个能够匹配训练过程分类数据的从特征向量到实数值的线性函数。该线性函数即泄漏风险预测函数 $risk(t)$，是一个从内存泄漏相关的特征向量到泄漏风险评估值的函数：

$$risk(t)=G(f_1, f_2, \cdots, f_n), t \in T_{\text{func}} \tag{6}$$

本文将特征向量抽取以及从特征向量到泄漏风险的预测函数称为泄漏风险预测模型，利用该模型即可对候选功能测试脚本按照相关风险值进行优先级排序。

3.4 功能测试脚本内存泄漏风险预测

整个训练与预测的过程如图 3 所示。首先，输入一组训练用测试脚本对应特征向量集及每个特征向量对应已知内存泄漏风险，据此训练获得一个内存泄漏风险预测模型。之后，输入待预测排序的 l 个功能测试用例，获得其内存特征矩阵，对矩阵中的每行计算出一个风险评估值，根据评估值得到排序后的测试脚本向量，排在前面的脚本泄漏风险高。

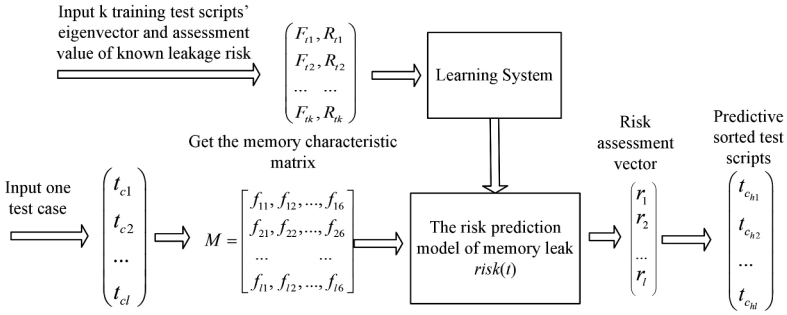

图 3 测试用例脚本预测过程

Fig. 3 Forecasting process of test case scripts

在训练获取的内存泄漏预测模型的基础上，对待预测重组的功能测试脚本进行预测排序。每个测试脚本 t_i 可以获得一个内存泄漏相关执行特征构成的特征向量 $\boldsymbol{F}(t_i)$，所有脚本的特征向量构成内存特征矩阵 $\boldsymbol{M}$，如式(7)所示：

$$\boldsymbol{M}=\begin{bmatrix}\boldsymbol{F}(t_{c1})\\ \boldsymbol{F}(t_{c1})\\ \vdots\\ \boldsymbol{F}(t_{cl})\end{bmatrix}=\begin{bmatrix}f_{11} & f_{12} & \cdots & f_{16}\\ f_{21} & f_{22} & \cdots & f_{26}\\ \vdots & \vdots & & \vdots\\ f_{l1} & f_{l2} & \cdots & f_{l6}\end{bmatrix} \tag{7}$$

以特征矩阵 $\boldsymbol{M}$ 为输入，将预测函数 $risk(t)$ 作用在特征矩阵的每一行，可以得到内存泄漏风险评估向量，如式(8)所示：

$$\boldsymbol{M}=\begin{bmatrix}\boldsymbol{F}(t_{c1})\\ \boldsymbol{F}(t_{c1})\\ \vdots\\ \boldsymbol{F}(t_{cl})\end{bmatrix}=\begin{bmatrix}f_{11} & f_{12} & \cdots & f_{16}\\ f_{21} & f_{22} & \cdots & f_{26}\\ \vdots & \vdots & & \vdots\\ f_{l1} & f_{l2} & \cdots & f_{l6}\end{bmatrix}\xrightarrow{risk(t)}\begin{bmatrix}r_1\\ r_2\\ \vdots\\ r_l\end{bmatrix} \tag{8}$$

其中每个元素 r_i 对应一个测试脚本的内存泄漏风险。

根据内存泄漏风险评估值 $r_1, \cdots, r_l$ 从高到低的顺序，可以对脚本进行排序。排列在前面的功能测试脚本引入大量循环执行，构成面向内存泄漏的性能测试脚本后，更可能揭示内

存泄漏缺陷。将这些高风险测试脚本优先执行,可以提高内存泄漏的发现效率。

3.5　面向测试脚本的重组优化

根据上一节功能脚本的筛选结果,能够得到一组疑似造成内存泄漏的普通功能测试脚本及脚本内存泄漏风险排序结果。然而,仅仅简单地执行这类脚本往往使内存开销非常小,难以真正揭示内存泄漏问题。为此,必须对这类脚本进行重组优化,加入循环操作,使得脚本中的事件反复执行,从而在Web服务器端产生较大的内存压力。

本文通过自动程序变换的方式进行脚本重组。重组前的脚本为通过Selenium IDE录制的功能测试脚本,如图4所示,脚本形式采用python语言描述。

```
class Sina(unittest.TestCase):
    def setUp(self):
        ...
    def test_sina(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_name("SerchKey").send_keys("2016")
        driver.find_element_by_name("SearchSubButton").click()
    def tearDown(self):
        ...
if __name__ == "__main__":
    unittest.main()
```

图4　Web应用测试脚本示例

Fig. 4　Example of Web application test scripts

每个脚本包含一个"_main_"方法和一个class类,"_main_"方法是脚本的执行入口,class类中包含了主体测试方法以及辅助方法*setup*()和*tearDown*()。*setUp*()用来准备测试脚本执行环境,*tearDown*()负责在测试方法执行结束后进行资源回收。测试方法是名如test_×××的成员方法,如图4中的*test_sina*()。该测试方法中包含4条执行语句,第1语句是参数赋值语句,第2句*driver.get*(url)是根据Web应用的URL访问应用首页,第3句和第4句根据name来查找页面元素并进行操作,这两条语句分别完成了在编辑框中输入"2016"、点击name为"SearchSubButton"的按钮两个操作。一个测试脚本可以包含多个测试方法,脚本执行时将按顺序执行每种测试方法。

算法1　脚本重组算法

IN:功能测试脚本集合T_S,循环次数loopcount

OUT:重组后所得脚本output_script

```
BEGIN
    AST={ parse_to_ast(s) | s ∈T_S};
    ast=join_scripts(AST);
    ast=insert_loops_into_ast(ast,loopcount);
    parameterize_the_loops(ast);
    output_script=dump_ast_to_code(ast);
  return output_script;
END
```

本文所提出的功能测试脚本重组算法如算法1所示。算法1从一组功能测试脚本重组出一个带循环的新测试脚本。输入为测试脚本集T_s,其中含有一个或多个测试脚本以及一个循环次数设定值*loopcount*。该算法首先将输入脚本解析为一组语法树的集合,然后调用*join_scripts*()方法将所有功能测试脚本拼接成一个大的测试脚本。拼接的思路是将原有脚本的代码复制到一个新的脚本中,然后引入一个新的测试方法,在该方法中调用输入功能测试脚本中的测试方法。完成拼接后,调用*insert_loops_into_ast*()方法为主测试方法体引入循环,使得主要的测试步骤能够多次执行。循环的引入主要通过在测试脚本中加入for循环语句实现。进一步地,使用*parameterize_the_loop*()方法对测试脚本进行参数化。整个方法将配置一个预定义的参数化表,该表是一个常数数据到变量数据的映射。如果脚本中存在匹配参数化表的常数数据,则将其替换为参数变量,同时将一条关于该参数变量的加载语句插入到脚本代码中。如此,脚本在循环执行的过程中可以从参数数据池读取数值,而不是用硬编码的常数作为数据执行测试。参数化使得重组优化后的测试脚本更逼近真实情况。最后调用*dump_ast_to_code*()将抽象语法树写回应用程序,完成整个拼接和循环化流程。

拼接测试方法代码和在脚本中加入循环执行语句能够实现脚本长时间反复执行某些操作的效果,提高系统内存消耗,增加暴露内存泄漏的可能性。对脚本中固化的取值进行参数化,可使脚本并不局限于单独用户帐号、单独数据的操作。

4　工具设计与实现

本文研制了内存泄漏加速检测工具,具体模块和流程如图5所示。该工具包含5个功能模块,即功能测试脚本获取模块、测试脚本插桩模块、内存使用信息采集模块、内存泄漏训练预测模块、测试脚本重组模块。

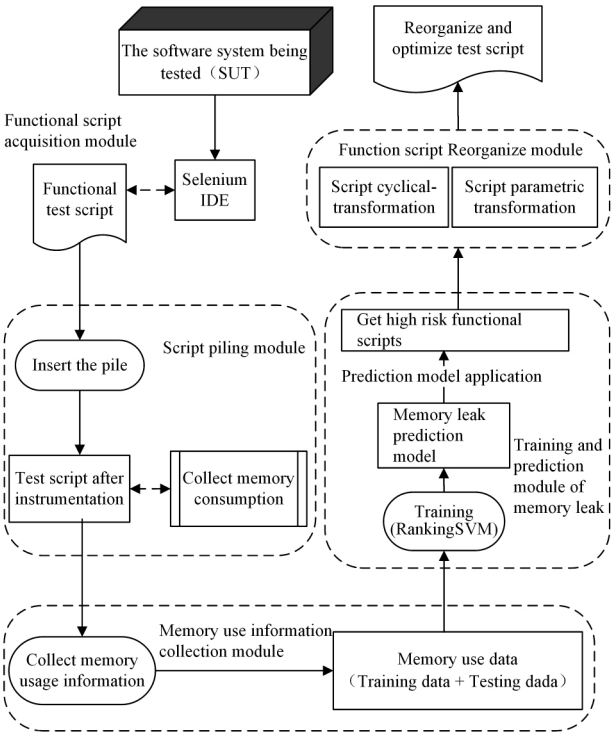

图5　内存泄漏检测工具模块和流程

Fig. 5　Modules and processes of memory leak detection tool

功能脚本获取模块:通过Selenium IDE脚本录制环境,录制待测Web应用的使用过程,从而形成功能测试脚本资源池。

测试脚本插桩模块:对获取的功能测试脚本进行结构分

析和插桩设计，实现运行时跟踪，获取 Web 应用在脚本驱动下的内存使用情况。

内存使用信息采集模块：运行测试脚本，并在运行过程中使用插桩注入探针等机制收集内存使用信息，获得测试脚本的内存泄漏特征向量。

内存泄漏训练预测模块：通过 RankingSVM 算法对测试脚本资源池的功能测试脚本进行训练，获取特征脚本的权重值，生成训练模型；导入功能测试脚本的内存消耗特征向量，基于训练获得的预测模型，对脚本资源池中的脚本进行内存泄漏风险排序。

功能脚本重组模块：从排名后的脚本池中选取多个内存泄漏风险较高的功能测试脚本，引入循环并进行参数化，提高内存消耗水平，由此获得能够快速造成异常泄漏的高资源消耗测试脚本。

工具的执行界面如图 6—图 8 所示，分别从测试脚本内存使用跟踪、测试脚本内存消耗分析及功能测试脚本重组等多个方面对原型系统进行展示。

图 6 所示的测试脚本跟踪执行界面主要展示测试脚本跟踪执行时 Web 应用程序的实时内存消耗，并提供内存相关的实时数据变化，以便测试人员观察脚本执行时 Web 应用程序的运行状态。

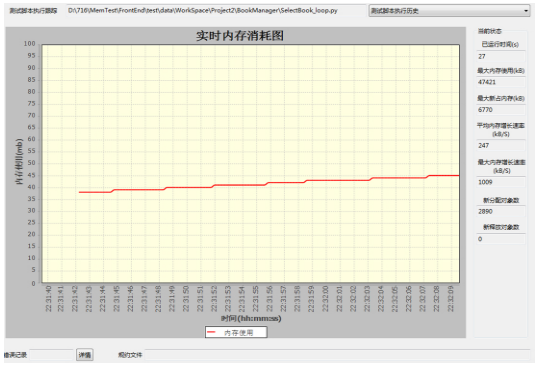

图 6 测试跟踪执行界面

Fig. 6 Test trace execution interface

图 7 所示的测试脚本内存消耗分析界面，给出了 Web 应用程序在不同测试脚本驱动下的内存使用表现，包括对象创建与释放、执行时间等信息。在内存消耗分析界面，测试人员可以根据测试脚本执行时的内存消耗表现对测试脚本进行打分并训练，在得到预测模型后可以根据内存泄漏风险对测试脚本进行排序。

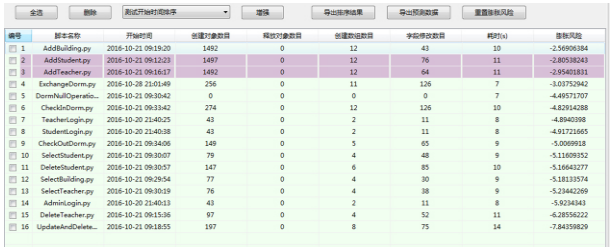

图 7 测试脚本内存消耗分析界面

Fig. 7 Memory consumption analysis of test scripts interface

在图 8 所示的功能测试脚本重组界面，可以对内存泄漏风险较高的测试脚本进行循环化重组，进而得到能够快速暴露内存泄漏风险的性能测试脚本。该界面提供了测试脚本拼接和循环化的配置选项及参数化设定，基于给定设置自动生成重组后的新脚本，最后将重组后的性能测试脚本添加到项目结构树。

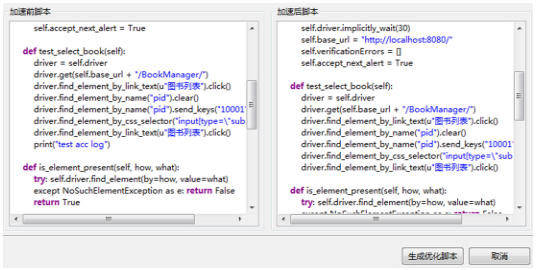

图 8 功能测试脚本重组界面

Fig. 8 Functional test script restructure interface

5 实验评估

为了覆盖上述研究内容，验证工具的有效性，本文选择某型号海军后勤综合保障系统软件作为待测对象，进行软件工具的验证，以验证本文工具的可行性和有效性。在实验过程中，首先对待测的应用系统通过 Selenium IDE[20] 工具进行测试脚本资源池的获取，采用 Soot(2.5.0)[21] 工具进行插桩来记录测试用例执行过程中的内存消耗信息。之后结合待测应用具有内存泄漏风险的代码特征进行分析，获取模型的输入矩阵，通过内存泄漏预测模型进行预测，生成具有内存泄漏的测试用例。

本文选取了后勤综合保障系统的 8 个应用模块作为验证对象，分别为物资保障管理、装备维修管理、装备配置管理、保障指挥管理、技术资料管理、系统管理、政务管理、装备使用管理模块，相关信息如表 2 所列。

表 2 后勤综合保障系统的相关信息

Table 2 Relevant information of logistics support system

Module name	Module description	Number of detected test cases	Number of test scripts with leaks
Material Manager	Material support management	10	2
Equip Manager	Equipment maintenance management	12	4
Config Manager	Equipment configuration management	8	4
Support Manager	Support Command management	6	2
Political Manager	Political management	6	1
Data Manager	Technical files management	10	1
System Manager	System configure management	13	3
Use Manager	Equipment Use management	16	3

其中，将物资保障管理模块、装备维修管理模块、装备配置管理、保障指挥管理及政务管理作为训练模块，将技术资源

管理、系统管理、装备使用管理作为应用验证对象。训练后，　　训练测试集及其特征向量如表3所列。

表3　训练测试集及其特征向量

Table 3　Training test set and its eigenvector

Training model	Training script	new	free	newarray	open	close	Time/ms	Training Score	leak
Material Manager	AddMaterial. py	64687	0	3478	36292	381	10562	5	No
	AddMaterialBack. py	66714	51677	3705	37250	453	12112	3	No
	SelectMaterial. py	40367	0	3131	19360	358	8073	8	Yes
	UpdateMaterial. py	38779	0	3055	18373	349	9296	5	No
	SelectMaterialTwice. py	124967	117394	5221	76395	602	19368	12	Yes
Equip Manager	EquipManagerNull. py	46823	0	7537	7940	420	9544	5	No
	EquipManagerCheckView. py	46311	12097	7486	7902	622	10025	3	No
	EquipManagerComplete. py	197801	130458	30228	40744	1457	25122	20	Yes
	EquipManagerEdit. py	52785	0	8700	9197	651	9333	5	No
	EquipManagerRefresh. py	113112	0	17260	22857	617	14032	12	Yes

实验从对象申请、释放资源申请与释放、执行时间选择3方面选择了 new，newarray，free，open×××，close×××，elapsedtime 等6个特征参数以及第3节提出的排序学习的 RankingSVM 算法，数据集信息如表2所列。

表3中“new”这一列给出了部分 Web 应用运行过程中对象创建的数量，“free”这一列给出了释放的对象数目，“newarray”这一列给出了创建数组数目，“open”这一列给出了 Web 应用运行过程中使用系统资源的次数，“close”这一列给出了关闭资源的次数，“Time”这一列表示测试脚本执行的耗时。训练评分由人工根据测试脚本驱动下 Web 应用的内存消耗表现获得，主要参考是否泄漏以及消耗内存等信息。

由于此系统存在长时间运行的压力，通过前期的测试发现其中存在申请对象未释放泄漏、对象过度持有泄漏、低效数据结构、缓存区对象未释放、重复副本消耗等缺陷，此类缺陷通过软件正确性测试时无法发现，而是在系统长期的运行中才会暴露，增加了后期系统的维护及保障成本，并严重影响了系统的性能。通过实验对象完成验证后，结果信息如表4－表6所列。从表4－表6可以看出，存在内存泄漏的测试脚本预测所得的泄漏风险评分较高，均排在所有测试脚本中的靠前位置，证明了本文提出的预测模型的有效性。

表4　验证用实验对象 DataManager 各测试脚本内存特征向量所对应的泄漏风险评分

Table 4　Leaking risk score corresponding to memory eigenvector of each test script for validation subjects DataManager

Test script	Leaking risk score	Leak
DatareaderCache. py	198.6145207	Yes
DatareaderChangeLG. py	125.2799517	No
DataReaderRegister. py	105.1080021	No
DataReaderLogOn. py	85.10560733	No
DatareaderEdit. py	83.44482305	No
DataReaderReset. py	81.37690457	No
DatareaderUpdateHost. py	74.20430304	No
DataReaderBack. py	55.36386747	No
DatareaderAddHost. py	52.83448091	No
DataReaderNull. py	21.07221193	No

表5　验证用实验对象 SystemManager 各测试脚本内存特征向量所对应的泄漏风险评分

Table 5　Leaking risk score corresponding to memory eigenvector of each test script for validation subjects SystemManager

Test script	Leaking risk score	Leak
SelectRegisterOrder. py	138.38180884	No
UpdateRegisterOrder. py	108.77978143	Yes
AddRegisterBack. py	82.32304936	Yes
SelectUserByGender. py	55.17494556	Yes
AddRegisterOrder. py	52.73076823	No
UpdateUserInfo. py	47.63482919	No
AddUser. py	46.72325386	No
AddUserBack. py	42.89640579	No
ShowView. py	78.48572485	No
LogOut. py	35.35232347	No
Login. py	27.29218495	No
Login_loop. py	14.99561579	No
ShowFormAndRefresh. py	9.85587854	No

表6　验证用实验对象 UseManager 各测试脚本内存特征向量所对应的泄漏风险评分

Table 6　Leaking risk score corresponding to memory eigenvector of each test script for validation subjects UseManager

Test script	Leaking risk score	Leak
AddBuilding. py	97.92846031	Yes
AddMaterial. py	97.61142266	Yes
AddManager. py	85.27254214	No
ExchangeDorm. py	84.37630807	Yes
DormNullOperation. py	83.01274182	No
CheckInDorm. py	80.06665381	No
ManagerLogin. py	70.09832455	No
MaterialLogin. py	64.12162326	No
CheckOutDorm. py	52.36767310	No
SelectMaterial. py	51.61553205	No
DeleteMaterial. py	51.35113752	No
SelectBuilding. py	45.54200944	No
SelectManager. py	44.61657608	No
AdminLogin. py	33.75532452	No
DeleteManager. py	27.99997884	No
UpdateAndDeleteBuilding. py	27.71183356	No

表7列出了预测过程中的一些相关统计信息。N_{Leak} 表示测试脚本集合中真实存在的内存泄漏的测试脚本数目。RN_{Leak}”表示前 N_{Leak} 个排序在前的脚本中包含的内存

泄漏的脚本数。$Accuracy$ 表示脚本排序准确率，其计算公式如式(9)所示：

$$A=1-\frac{LastLeak_{\text{index}}-N_{\text{leak}}}{N} \tag{9}$$

其中，$LastLeak_{\text{index}}$ 表示使用预测模型预测排序之后，最后一个真实泄漏的脚本在所有排序后脚本中的位置。N_{leak} 表示真实泄漏的测试脚本数目。

式(9)的分子是错误地排在泄漏脚本之前的非泄漏脚本的数目，分母 N 表示脚本的总数。因此，排错的脚本越多，预测精度就越低。

表 7 实验对象结果信息

Table 7 Experimental object result information

Application program	N_{Leak}	RN_{Leak}	$Accuracy$/%	LeakF	NLeakF	NLeakS	LeakS
DataManager	1	1	100	(124093,20014,4959,913,605,10000)	(42480,11382,6806,4276,986,13000)	88.24	198.61
SystemManager	3	2	87.5	(47448,25076,6117,792,719,15333)	(20721,5768,4010,458,407,11400)	56.97	82.09
UseManager	3	3	100	(9360,15331,6110,15289,15333,792)	(977,19872,510,619,2345,88)	62.34	93.3

表 7 中，LeakF 表示存在内存泄漏的向量的平均值，NLeakF 表示不存在内存泄漏的向量的平均值，NLeakS 表示不存在泄漏的脚本的平均泄漏风险预测评分，LeakS 表示存在泄漏的脚本的平均泄漏风险预测评分。从表 7 也可以看出，存在内存泄漏的测试脚本的预测分数明显大于不存在内存泄漏的测试脚本的预测分数。

为了验证内存泄漏脚本的准确性及其发现内存泄漏现象的时效性，分别采用传统的内存性能检测方法和重组生成的内存泄漏脚本进行内存泄漏测试，并对比发现内存泄漏问题所需的代价。

传统的增加循环进行内存性能测试的方法中，发现内存泄漏问题所需代价的计算公式如下：

$$T_{\text{no\_rank}}=\sum_{i=0}^{M=N/(2\times C)} L_{\text{loop}}\times T_i \tag{10}$$

其中，$T_{\text{no\_rank}}$ 是使用传统循环测试方法的测试总耗时。M 是发现第一个含内存泄漏问题的脚本，所需执行的总的脚本数。设总计有 N 个测试脚本，其中 C 个脚本存在内存泄漏问题，测试者具有平均化的发现问题可能性，概率密度过半即能够发现缺陷，则 $M=N/(2\times C)$。T_i 表示一个未进行增强的测试脚本一次执行的时间，L_{loop} 是增强过程中设定的循环轮次。

通过本文方法进行预测，只需将每个功能测试脚本运行一轮，即可进行预测排序，之后可按从高泄漏风险到低泄漏风险的顺序来增强和检测测试脚本。发现第一个内存泄漏的脚本的耗时可用式(11)来计算。

$$T_{\text{ranked}}=\sum_{i=0}^{N} T_i+T_{\text{prev}}+T_{\text{bloat}} \tag{11}$$

其中，T_{ranked} 是预测排序方法发现一个含有内存泄漏问题脚本所需要的总时间，T_i 表示一个未增强的功能测试脚本一轮的执行时间，N 表示测试脚本的总数，$\sum_{i=0}^{N} T_i$ 是所有未增强测试脚本一轮执行的时间之和，T_{prev} 是在发现真实内存泄漏问题之前进行的循环增强脚本执行的累计时间，T_{bloat} 是第一个训练增强的膨胀测试脚本的执行时间。

在实验过程中，对测试对象进行了多次实验测试，实验表明，对功能测试脚本至少引入 100 次循环才能够产生相对明显的可观察的内存泄漏现象，该次数是发现 Java Web 应用内存泄漏的合适循环次数，按该多次数录制所得的功能测试脚本进行循环增强，将其转化为性能测试适合的脚本，进行内存泄漏测试。通过对 ConfigManger，DataManger 进行实验，得到各个功能测试脚本的执行时间和 100 次之后的执行时间，将其代入式(10)进行计算，统计结果如表 8 所列。

表 8 增强型脚本与传统循环测试耗时对比

Table 8 Time consuming comparison between enhanced script and tranditional loop test

Webapplication	Number of real expansion scripts	Total scripts	Traditional cycle test time/s	Enhanced script test time/s	Remarks
DataManager	1	10	6701	734	6701 s 是执行 10/2=5 个循环增强的测试脚本的耗时。所有脚本执行一轮的耗时，加上第一个泄漏脚本循环增强后的耗时，得到 734s
SystemManager	3	13	7758	5658	7758 s 是执行 13/2=7 个循环增强的测试脚本耗时。所有脚本执行一轮的耗时，加上前 4 个循环增强非膨胀脚本的耗时，再加上第一个循环增强膨胀脚本的耗时，得到 5658 s
UseManager	3	16	3020	1151	3020 s 是执行 16/(3*2)=3 个循环增强的测试脚本的耗时。所有脚本执行一轮的耗时，加上第一个膨胀脚本循环增强后的耗时，得到 1151 s
Average	2	13	5826	2514	5826>2*2514

从表 8 可以看出，平均情况下，本文提出的方法的内存泄漏时间比传统循环测试发现内存泄漏的时间缩短一半以上，能够节省 50% 的测试时间，内存泄漏的发现速度提高了一倍以上。

综上所述，系统运行过程中存在各类内存泄漏的缺陷，例如重复副本消耗，用一个集合模拟当前活跃的用户 session，同时在内存中，存在一个运行记录的多个冗余副本持续执行，导致缓存区对象未释放，当系统频繁加载新图片时，可能会造成内存空间溢出，从而形成内存泄漏造成的膨胀问题。这些问题通过常规的测试手段均无法发现，因此迫切需要加速内存膨胀问题的暴露时间，减少测试过程对人工介入的依赖，达到提高测试效率以及保障软件质量的目的。

结束语　由于海军装备软件中 Java 开发的程序多为 B/S 架构下的 Web 应用，如何有效地保障 Java Web 应用工作的稳定性及性能已经成为该类软件测试迫切需要解决的难题。当前 Java Web 应用内存泄漏研究主要关注发现泄漏现象后如何诊断泄漏对象、泄漏语句等问题，但是关于如何通过测试技术发现潜在的泄漏现象的相关研究较为缺乏，缺少成熟有效的解决方案。本文提出了一种脚本预测和重组的内存泄漏测试加速技术，通过对测试用例集合进行预测筛选，找到其中高内存泄漏风险的功能测试脚本，并采用引入循环的方式对这些用例进行自动重组优化，构造检测错误能力更强的性能测试脚本，然后优先执行对高风险功能测试脚本重组优化所得的性能测试脚本，从而加速内存泄漏问题的暴露过程，减少测试过程对人工介入的依赖，提高测试效率。

在今后的工作中，将进一步完善研究内容，考虑更多发现内存泄漏现象之后的工作，增加泄漏现象的诊断和修复模块，以更全面地支持内存泄漏的测试工作。

参 考 文 献

[1] ŠOR V，SRIRAMAS N. Memory leak detection in Java：Taxonomy and classification of approaches[J]. Journal of Systems and Software (JSS)，2014，96：139-151.

[2] VALENTIM N A，MACEDO A，MATIASR. A Systematic Mapping Review of the First 20 Years of Software Aging and Rejuvenation Research[C]// IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2016.

[3] HEINE D L，LAM M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector[C]// ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2003.

[4] XIE Y，AIKEN A. Context-and path-sensitive memory leak detection[C]// 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). 2005：115-125.

[5] XU Z，ZHANG J，XU Z. Memory leak detection based on memory state transition graph[C]// Proceedings of the Asia-Pacific Software Engineering Conference (APSEC). 2011：33-40.

[6] SUI Y，YE D，XUE J. Static memory leak detection using full-sparse value-flow analysis[C]// Proceedings of the International Symposium on Software Testing and Analysis. 2012.

[7] LI Q，PAN M X，LIX D. Benchmark of tools for memory leak [J]. Computer Science and Exploration，2010，4(1)：29-35.

[8] JUMP M，MCKINLEY K S. Detecting memory leaks in managed languages with Cork[J]. Software：Practice and Experience，2010，40(1)：1-22.

[9] MAXWELL E K，BACK G，RAMAKRISHNANN. Diagnosing memory leaks using graph mining on heap dumps[C]// Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). 2010：115-124.

[10] RAYSIDE D，MENDE L L. Object ownership profiling：a technique for finding and fixing memory leaks [C]// The 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2007：194-203.

[11] ŠOR V，SRIRAMAS N，SALNIKOV-TARNOVSKI N. Memory leak detection in Plumbr. Software[M]. Practice and Experience (SPE)，2014.

[12] CHILIMBI T，HAUSWIRTHM. Low-overhead memory leak detection using adaptive statistical profiling[C]// The 11th International Conference on Architectural Support for Programming Languages and Operating Systems. 2004.

[13] BOND M D，MCKINLEY K S. Bell：bit-encoding online memory leak detection[C]// The 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006：61-72.

[14] XU G，ROUNTEV A. Precise memory leak detection for Java software using container profiling[J]. ACM Transactions on Software Engineering and Methodology (TOSEM)，2013，22(3)：17.

[15] JUNG C，LEE S，RAMAN E，et al. Automated memory leak detection for production use[C]// International Conference on Software Engineering (ICSE). 2014.

[16] LEE S，JUNG C，PANDES. Detecting memory leaks through introspective dynamic behavior modelling using machine learning [C]// International Conference on Software Engineering (ICSE). 2014.

[17] JIA X X，WU J，JIN M Z，et al. Overviewon memory leak of Java program [J]. Computer Research and Application，2006(9)：1-4.

[18] SHAHRIAR H，NORTH S，MAWANG I E. Testing of Memory Leak in Android Applications[C]// International Symposium on High-Assurance Systems Engineering (HASE). 2014.

[19] YAN D，YANG S，ROUNTEV A. Systematic testing for resource leaks in Android applications[C]// 24th International Symposium on Software Reliability Engineering (ISSRE). 2013.

[20] GUNDECHA U. Selenium Testing Tools Cookbook (2 edition) [M]. Packt Publishing，2015.

[21] BODDEN E. Invoke Dynamic support in Soot. ACM SIGPLAN [C]// International Workshop on the State of the Art in Java Program Analysis. 2012：51-55.

LI Yin，born in 1988，master. His main research interests include bigdata，software reliability and software testing，etc.

LI Bi-xin，born in 1969，Ph. D，professor. His main research interests include software analysis，testing and verification，and empirical software engineering，etc.