分类号：TP391　　学校代码：10697

密　级：公开　　学　　号：201731960

西北大学
Northwest University

硕士专业学位论文

Dissertation for the Professional Degree of Master

基于类型推断的 JavaScript 引擎模糊测试方法研究

学科名称：软件工程

专业学位类别：工程硕士

作者：曹帅　　指导老师：房鼎益 教授

西北大学学位评定委员会

二〇二〇年

Research on Type-Inference-based JavaScript Engine Fuzzing Test

A thesis submitted to

Northwest University

in partial fulfillment of the requirements

for the degree of Master

in Software Engineering

By

Cao Shuai

Supervisor: Fang Dingyi Professor

2020

西北大学学位论文知识产权声明书

本人完全了解西北大学关于收集、保存、使用学位论文的规定。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。本人授权西北大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时授权中国科学技术信息研究所等机构将本学位论文收录到《中国学位论文全文数据库》或其它相关数据库。

保密论文待解密后适用本声明。

学位论文作者签名：曹帅 指导教师签名：

2020年6月11日　　2020年6月11日

西北大学学位论文独创性声明

本人声明：所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，本论文不包含其他人已经发表或撰写过的研究成果，也不包含为获得西北大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：曹帅

2020年 6月 11日

摘要

JavaScript 是一种基于原型的动态弱类型脚本语言。作为弱类型语言，JavaScript 程序中不能指定每个变量的类型，其执行器——JavaScript 引擎在执行到相应的语句时才能对变量的类型进行判断。由此，许多在强类型语言中可以规避的类型异常都可能被隐藏，难以发现问题所在。在对 JavaScript 引擎进行测试时，如何高效地产生代码覆盖率高的测试用例，并且更快地发现其隐含的缺陷，这些问题都亟待解决。

因此，本文提出了一种基于类型推断的 JavaScript 引擎模糊测试方法。具体研究内容如下：

（1）为了避免测试用例中位置靠前的代码存在异常导致程序过早退出执行，提高代码覆盖率和原始语料的利用率，将原始语料库中的代码拆分为 JavaScript 中的函数，称为预备测试用例。

（2）为了有效地调用这些函数，并进一步提升预备测试用例的代码覆盖率，本文提出了一种参数类型推断方法。首先对函数的每个参数遍历函数体，统计每种数据类型的类型推断因子数，统计得分最高的数据类型即推定为该参数的数据类型。然后据此生成实际参数和函数的调用表达式，即得到了具有高代码覆盖率且能高效触发 JavaScript 引擎崩溃缺陷的测试用例。最后根据类型推断结果，对测试用例进行具有引导性的精确变异，提高了通过满足边界条件进一步提升代码覆盖率和触发更多 JavaScript 引擎缺陷的可能。

（3）为了验证上述方法的有效性，本文实现了原型系统 JSTIFuzz，并使用该原型系统进行了参数类型推断效果评估实验、代码覆盖率提升效果评估实验和模糊测试效果评估实验。实验结果表明，使用 JSTIFuzz 对函数的参数进行类型推断，最高比随机传参的类型精确率高 10.8 倍。并且 JSTIFuzz 能使测试用例的代码覆盖率最高提升 30.33%，使 JavaScript 引擎的代码覆盖率最高提升 8.81%. 在以同等的原始语料库作为输入，并在相同的时间及环境下，JSTIFuzz 可以在测试集上比其他模糊测试工具触发 JavaScript 引擎更多的崩溃缺陷。最后，在针对各 JavaScript 引擎最新版本进行的为期 100 个小时的模糊测试中，本文发现并提交了 Rhino、JerryScript、QuickJS 和 Hermes 等 4 个 JavaScript 引擎的崩溃缺陷共 6 个，其中有 2 个已被确认。

关键词：JavaScript 引擎，类型推断，覆盖率，模糊测试

ABSTRACT

JavaScript is a prototype-based dynamic weakly typed scripting language. As a weakly typed language, the type of each variable cannot be specified in the JavaScript program, and its executor, the JavaScript engine, can judge the type of the variable only when the corresponding statement is executed. Therefore, many type anomalies that can be avoided in strongly typed languages may be hidden, making it difficult to find the problem. When testing the JavaScript engine, how to efficiently generate test cases with high code coverage and find their hidden defects more quickly are all problems that need to be resolved.

Therefore, this thesis proposes a fuzzing method for JavaScript engines based on type inference. The specific research contents are as follows:

(1) In order to avoid the abnormality of the code in the front and cause the program to exit prematurely, improve the code coverage and the utilization rate of the original corpus, the code in the original corpus is split into functions in JavaScript, called pre-test cases.

(2) In order to effectively call these functions and further improve the code coverage of pre-test cases, we propose a parameter type inference method. First traverse the function body for each parameter of the function, the number of type inference factors of each data type is counted, and the data type with the highest statistical score is presumed to be the data type of the parameter. Then the actual parameters and function call expressions are generated accordingly, that is, test cases with high code coverage and efficient triggering of JavaScript engine crash defects are obtained. Finally, based on the type inference results, the test cases are guided and accurately mutated, which improves the possibility of further improving code coverage and triggering more JavaScript engine defects by meeting boundary conditions.

(3) In order to verify the effectiveness of the above methods, this thesis implements the prototype system JSTIFuzz, and uses the prototype system to conduct the parameter type inference effect evaluation experiment, code coverage improvement effect evaluation experiment and the fuzzy test effect evaluation experiment. Experimental results show that

using JSTIFuzz to infer the type of parameters of functions, the highest type accuracy rate is 10.8 times higher than that of random parameter passing. JSTIFuzz can also increase the code coverage of test cases by up to 30.33% and the code coverage of JavaScript engines by up to 8.81%. Using the same original corpus as input and under the same time and environment, JSTIFuzz can trigger more crash defects on the test set than other fuzzing tools. Finally, in a 100-hour fuzz test conducted on the latest version of each JavaScript engine, we found and submitted a total of 6 crash defects of 4 JavaScript engines such as Rhino, JerryScript, QuickJS, and Hermes, of which 2 have been confirmed.

Keywords: JavaScript engine, Type inference, Code coverage, Fuzzy testing

目录

第一章 引言

1.1 研究背景与意义

1.1.1 JavaScript 语言及其引擎简介

JavaScript 是用于实现 Web 应用程序的核心脚本语言[1]。1995 年 5 月系统程序员 Brendan Eich 只用了 10 天就将 JavaScript 开发了出来[2]，它被设计为一种与 Java 足够相似的基于原型的脚本语言。

由于其简单易上手的语言特性，在诞生以来的 20 多年间，JavaScript 迅速成长为一种成熟的 Web 开发语言。如表 1 所示，JavaScript 在各大编程语言中的市场占有率稳定保持在前 10 以内，且有逐年提升的趋势[3]。

表 1 编程语言市场占有率排行表

编程语言	2019	2014	2009	2004	1999
Java	1	2	1	1	3
C	2	1	2	2	1
Python	3	7	6	6	21
C++	4	4	3	3	2
Visual Basic .NET	5	9	-	-	-
C#	6	5	5	8	16
JavaScript	**7**	**8**	**8**	**9**	**9**
PHP	8	6	4	5	31
SQL	9	-	-	89	-
Objective-C	10	3	25	35	-

互联网的发明，使人们获取信息的途径更加多元，更加便捷。人们可以通过编写静态网页来分享自己所掌握的知识、数据。随着越来越多的人进入网络世界，静态网页的弊端也显现了出来——无法产生交互。用户只能阅读他人写在网页上的静态内容，然后更新自己的内容，等待其他人的访问[4]。

JavaScript 语言的出现，使习惯了在互联网上浏览网页的人们开始发现，许多的网页逐渐拥有了各种各样的交互能力。例如人们可以通过一个按钮来订阅网页的内容更新，亦或者通过网页来订机票和火车票。

随着浏览器渲染和计算能力的不断提升，浏览器/服务器（Browser/Server，简称作 B/S）结构[5]被越来越广泛地应用于复杂的网页应用中。如图 1 所示，用户通过浏览器浏览所需的网页。从图中可以看到，JavaScript 引擎承担着以下几个方面的工作：1）通过编译或解释的方法执行 JavaScript 代码。2）当代码中包含与用户进行交互的逻辑时，引擎还需要及时地对 HTML 页面产生影响，即动态地修改页面的图形样式，或从当前页面跳转到其他的页面等。3）作为与用户交互最重要的一部分，JavaScript 引擎还负责向 Web 后端发送 Http 请求，并根据 Web 后端发回的请求响应内容执行 2）中提到的操作。可见 JavaScript 引擎复杂多样的功能及其重要性。

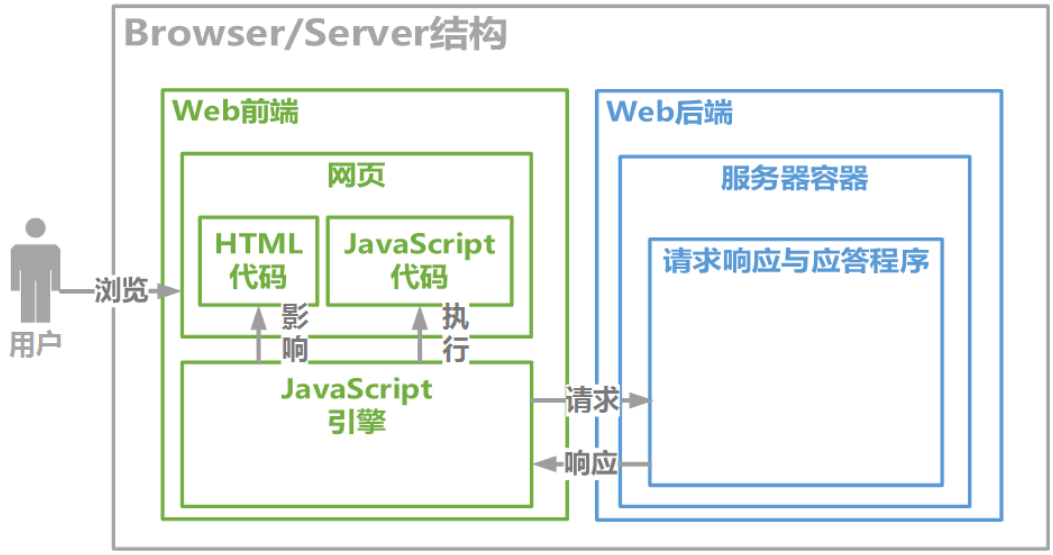

图 1　Browser/Server 结构图

众所周知，JavaScript 语言的标准由欧洲计算机制造联合会（ECMA）维护，称为 ES（ECMAScript）标准，该组织每隔 1~2 年就放出一个新的版本，如 ES 5、ES 6 等，当前已更新至 ES 10[6]。

一个严重的问题是 ECMA 并不实现他们提出的标准，JavaScript 引擎都是由不同的厂商根据该标准分别实现的。由于需求和能力不同，上述厂商有的选择 ES 5，有的选择 ES 6 来实现自己的引擎，甚至有的选择部分实现某一标准。由于这些 JavaScript 引擎的实现者存在理解能力的不同，或者 ES 标准中的描述不够清晰，实际上这些不一致的问题违背了标准制定者的初衷，而且这对 JavaScript 程序的编写者造成了极大的困扰，他需要针对不同的编译器编写不同的代码以确保系统的兼容性。因此，针对 JavaScript 引擎的测试研究具有重要的理论和应用价值。

1.1.2 JavaScript 引擎的测试及其问题

与 Java 等编程语言的代码不同的是，JavaScript 语言的代码不需要指定一个主函数，其引擎具有逐行代码执行的能力。即只要将符合其语言语法的代码文本输入其中，JavaScript 引擎就会按照代码组织的先后顺序执行每一行语句，直到顺利执行结束，或者在某一行处遇到运行时错误，异常退出[7]。

实际上，JavaScript 引擎在执行一份代码时，经常会发生程序在位置靠前的一行发生运行时异常，导致后面的多行代码无法被执行到的问题。

```
1:    print(variable_which_undefined);
2:    var vulnerable_func = function(param1, param2){
3:        return param1 / param2;
4:    };
5:    vulnerable_func(2019, 0);
```

图 2 针对 JavaScript 引擎的测试用例示例

如图 2 所示，这段代码的第 1 行在控制台中输出变量 variable_which_undefined 的值。2-4 行定义了函数 vulnerable_func，其两个参数 param1 和 param2，返回值为 param1 和 param2 相除的结果。第 5 行传入两个参数 2019 和 0 调用了前面定义的函数 vulnerable_func。用 JavaScript 引擎执行这段代码时，会在执行到第 1 行时报出“变量 variable_which_undefined 未被定义”的错误，并提前终止执行。而第 2-5 行原本会执行 2019 / 0 这样的运算逻辑，以测试 JavaScript 引擎如何应对除法运算中除数为 0 的情况。现由于程序在第 1 行提前终止执行，导致这 4 行代码无法被执行到。即在本条共 5 行的测试用例中，只有 1 行被执行，4 行无法被执行到，其代码的行覆盖率为 $1/5 \times 100\% = 20\%$.

由此可见，将整份的 JavaScript 代码作为测试用例来测试 JavaScript 引擎时，其执行结果会受到位置靠前的某些行代码的限制，导致引擎过早地异常退出，进而导致测试用例的代码覆盖率大幅度降低。这在很大程度上也降低了测试的效率。

1.1.3 研究的目的和意义

对一般的 JavaScript 开发人员而言，他们最关心的是自己所编写的代码是否正确并且没有功能上和安全方面的缺陷。也即对这些代码的执行器——JavaScript 引擎——是默认信任的，这就造成了人们对 JavaScript 引擎中可能存在的缺陷的忽视。而若 JavaScript 引擎存在缺陷，其在执行正确编写的程序代码时可能会给出错误的结果，甚至导致引擎自身崩溃，无法继续提供服务。

在 JavaScript 这门编程语言蓬勃发展的同时，其众多引擎所存在的缺陷问题得到了越来越多的关注。关于测试，基于断言的测试方法难以解决测试用例产生效率低的问题。现有的模糊测试方法虽然具备了生成测试用例的能力，但测试用例的代码覆盖率仍然较低。提升 JavaScript 引擎的测试效率成为了一个亟待解决的问题。

本文所提出的基于类型推断的测试用例参数生成方法为弱类型编程语言的测试提供了新的思路，也适用于同类型的 Php 和 VB Script 等编程语言。本文的研究对象是 JavaScript 引擎，其中涉及基于断言的测试、模糊测试等测试方法。本文中所设计并实现的原型系统以及相关的技术不仅适用于 JavaScript 语言及其引擎的测试，同样适用于 C 或者其他语言编译器的测试。

1.2 国内外研究现状

编译器作为各种高级编程语言的执行者，相较于对应的编程语言，获得的关注还很少。多数开发人员不关注编译器的测试，他们默认编译器是健壮的，没有缺陷。而近年来的各项针对编译器的研究正在逐渐推翻这种认识。模糊测试是目前较为流行的一种为处理结构化数据的软件创建测试输入的技术。它已经成功地应用于各个领域，从编译器和解释器到程序分析，再到呈现引擎、图像处理工具和文字处理器[8]。

Veggalam S 等人使用遗传编程来指导模糊测试器生成不常见的输入代码片段。遗传编程是进化算法的一种变体，受到达尔文的进化论的启发，通过重新组合当前最适合的个体的特征，来产生新的个体。使用基于该方法构建的原型系统，他们首先测试了老版本的 JavaScript 引擎，并发现了 40 个缺陷。而后又测试了该引擎的最新版本，最终发现了 17 个缺陷，其中有 4 个是安全相关的缺陷[9]。

Junjie Wang 等人将语料库中海量的用例解析成抽象语法树进行学习，得到概率性的上下文敏感的语法，以表示学习到的语法特征和语义规则，然后利用这些规则为模糊测试生成种子输入。他们的工作使平均的测试用例行覆盖率提升了 20%，函数覆盖率提升了 15%。最终发现了 IE 11 浏览器中的 19 个新的内存损坏缺陷和 32 个拒绝服务缺陷[10]。

Renata Hodovan 等人提出了一种基于原型图的 API 模糊测试方法。他们通过修改 JavaScript 引擎的代码，在其中加入一些功能，使其可以将引擎内部的 API 及其类型信息暴露出来，然后针对性地对这些 API 进行模糊测试。结果表明，该方法在代码总覆盖率方面可以与现有解决方案相媲美，在 JavaScript API 相关模块中甚至可以有更

好的覆盖率[11]。

Malik 等人发现代码中的自然语言内容拥有着丰富的知识，而这些内容通常被类型推断算法所忽略。这些内容包括代码中的注释、注解、函数名和参数名等，它们通常可以显式或隐式得表明函数及其参数的类型。他们通过基于 LSTM[12]的神经模型对包含注解的代码集进行学习，然后使用学习后的模型对不包含注解的函数集进行类型推断。在只考虑最高的一个推断意见时精确率为 84.1%，而在考虑最高的 5 个推断意见时预测类型的精确率达到了 95.5%[13]。类似地，本文所做的工作也需要做类型推断，但使用了完全不同的方法。

当前，JavaScript 语言的快速发展为多种环境下的软件应用开发提供了许多便利，越来越多的 JavaScript 引擎如雨后春笋般出现。但与此同时，各 JavaScript 引擎内部隐含的软件缺陷需要更加高效的方法予以揭示。

1.3 本文研究内容

本文通过对 JavaScript 编程语言及其引擎特性和对模糊测试方法的深入研究，提出了一种基于类型推断的 JavaScript 引擎模糊测试方法。对参数在函数体中的表现进行静态文本分析之后，确定了其数据类型，生成数据类型精确的函数调用表达式。在提高了测试用例和被测试 JavaScript 引擎的代码覆盖率的同时，也提高了通过测试发现 JavaScript 引擎缺陷的效率。

本文的研究内容包括以下几个方面：

（1）JavaScript 引擎软件测试方法研究

对常见的软件测试方法进行了深入的研究，分析了各种软件测试方法的自动化程度、测试用例产生途径和其特异的测试目标。针对发现 JavaScript 引擎崩溃缺陷的目的，详细分析了模糊测试技术及其测试用例生成和变异方法，从测试用例的产生效率和代码覆盖率等层面，对其生成和变异方法进行了对比研究。

（2）基于类型推断的 JavaScript 引擎模糊测试方法设计

本文所涉及的原始 JavaScript 代码（下文中称作“原始语料库”）来自 Github 上数千个 JavaScript 语言相关的代码仓库。为了提高代码覆盖率，将函数所引用的全局变量复制进其函数体中，然后提取了代码中的函数作为预备测试用例。为了避免类型错误带来的测试用例覆盖率降低，设计了一种参数类型推断方法。结合对 JavaScript 编程语言及其引擎特性的分析和（1）中的研究，提出了一种基于类型推断的 JavaScript

引擎模糊测试方法。对参数在函数体中的表现进行静态文本分析之后，确定了其数据类型，向测试模块更加精确地提供函数所需数据类型的实际参数，进一步提升了测试用例的代码覆盖率。根据参数类型推断的结果，对测试用例进行精准的变异，通过触发更多边界条件提升了触发 JavaScript 引擎缺陷的可能。

（3）设计并实现 JSTIFuzz 原型系统

根据本文提出的方法，设计并实现了原型系统 JSTIFuzz。对原型系统的模块设计、关键算法和界面设计进行了详细的介绍。使用 JSTIFuzz 对本文提出的参数类型推断方法的效果进行了评估，并在选定的 JavaScript 引擎的老版本和最新版本上进行了模糊测试的效果评估，还就测试结果与经典的和较新的几种模糊测试工具进行了对比，证明了本文所提出的方法的有效性和实用性。

1.4 本文组织结构

本文划分为五个章节来对基于类型推断的 JavaScript 引擎模糊测试方法研究进行论述，各个章节的研究内容如下所述：

第一章介绍了 JavaScript 语言及其引擎的重要性，引出了针对 JavaScript 引擎的测试中存在的难点，进而得出了本文研究工作的目的和意义。然后介绍了在模糊测试和针对 JavaScript 引擎的测试国内外的研究现状。最后概括了本文的研究内容。

第二章介绍了本文中用到的软件测试相关的理论与技术。首先介绍了几种常见的代码克隆检测技术。接下来介绍了几种常见的软件测试方法，最后对模糊测试技术中两种当前较为新颖的测试用例变异方法进行了简要的介绍。

第三章介绍了本文所设计的基于类型推断的 JavaScript 引擎模糊测试方法。包括语料库预处理、测试用例生成、参数类型推断和模糊测试等四个部分。

第四章从系统模块设计、关键算法设计和系统界面设计等三个方面，详细介绍了根据本文所提出的方法实现的原型系统 JSTIFuzz 的实现细节。

第五章对本文使用原型系统 JSTIFuzz 展开的评估实验进行了介绍和详尽的分析，包含代码覆盖率提升效果评估和模糊测试效果评估两个部分。

第二章 软件测试相关理论与技术

本章涉及很多与软件测试相关的理论与技术。为了避免重复原始语料对测试的效率产生负面影响，采用代码克隆检测技术对其进行过滤。本章还会对基于断言的测试方法和模糊测试方法这两种软件测试方法进行介绍和比较。为了探究对测试用例变异方法的设计对测试效果的影响，还将介绍和分析两种测试用例变异方法。

2.1 代码克隆检测技术

代码克隆检测技术用于检测两段代码的相似程度。在本文中代码克隆检测技术主要运用于对原始的 JavaScript 语料库中的代码进行去重。由于原始语料库从 Github 上收集而来，无法保证其唯一性，因此使用克隆检测技术来尽可能得去除一份语料的多余副本，进而保证测试结果的唯一性。

根据克隆程度的由简单到复杂，代码的克隆行为被分为以下几个等级[14]：1）完整代码片段的克隆，空白、布局和注释可以不同。2）在 1）的基础上，克隆的代码中修改了变量名、变量值等字面量。3）在 1）或 2）的基础上对克隆后的代码进行部分增加、删除和改动。4）语义相同但语法不同。

现有的克隆检测算法有很多种，如基于文本向量化的克隆检测技术、基于哈希算法的克隆检测技术等。

2.1.1 余弦相似度算法

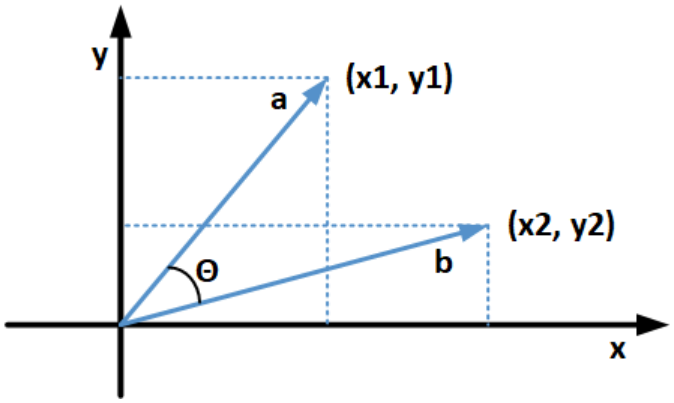

图 3 平面坐标系下的向量 a 和 b

余弦相似度算法[15]使用了数学中三角函数的余弦函数 cos，如图 3 所示，在一个平面直角坐标系中，向量a和 b 之间所成的夹角为θ，此时$\cos(\theta)$的计算过程如公式（2.1）

所示。

$$\cos(\theta) = \frac{a \cdot b}{|a| \cdot |b|} = \frac{(x_1, y_1) \cdot (x_2, y_2)}{\sqrt{x_1^2 + y_1^2} \times \sqrt{x_2^2 + y_2^2}} = \frac{x_1 x_2 + y_1 y_2}{\sqrt{x_1^2 + y_1^2} \times \sqrt{x_2^2 + y_2^2}} \quad (2.1)$$

由余弦函数的定义可知，当θ角趋近于 0 时，$\cos(\theta)$趋近于 1，此时 a 和 b 两个向量就越相似；当θ角趋近于90°时，$\cos(\theta)$趋近于 0，此时 a 和 b 两个向量就越不相似。公式（2.1）是在二维平面坐标系上计算得到的，将其推广到 3 维坐标系中，就得到了公式（2.2）。

$$\cos(\theta) = \frac{x_1 x_2 + y_1 y_2 + z_1 z_2}{\sqrt{x_1^2 + y_1^2 + z_1^2} \times \sqrt{x_2^2 + y_2^2 + z_1^2}} \quad (2.2)$$

进而，在 n 维坐标系中，以d_i表示一个向量在第i个坐标轴上的分量（如以d_1表示 x，d_2表示 y，以此类推），推广得到公式（2.3），两个 n 维向量的余弦相似度公式。

$$\cos(\theta) = \frac{\sum_{i=1}^{n} d_{i_1} d_{i_2}}{\sqrt{\sum_{i=1}^{n} (d_{i_1})^2} \times \sqrt{\sum_{i=1}^{n} (d_{i_2})^2}} \quad (2.3)$$

2.1.2 基于文本向量化的克隆检测技术

要将公式（2.3）推广到两段文本的相似度比较上，就必须先把要进行比较的文本变成向量，这一过程称为文本向量化[16]。

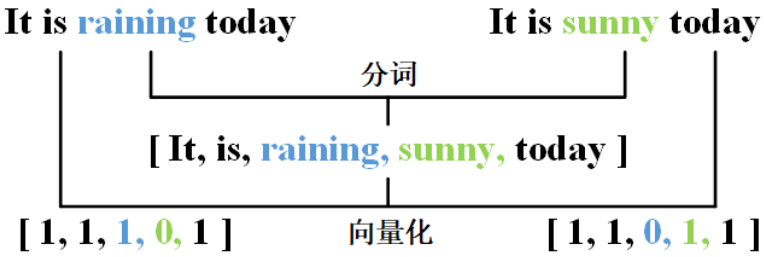

图 4 文本向量化示例

下面对文本向量化技术进行举例介绍。由图 4 所示，以“It is raining today”和“It is sunny today”两个英文短句为例。首先，对两个句子进行分词，即将两个句子拆分成一个一个的单词。得到一个包含 5 个单词的词库[It, is, raining, sunny, today]。然后分别对两个句子统计词频，即把每个句子用到词库中的每个单词的个数进行统计，得到两个句子的 5 维词向量分别为[1, 1, 1, 0, 1]和[1, 1, 0, 1, 1]，“1”表示词库中当前位置的词在句子中出现了 1 次，“0”则表示没有句子中没有用到该单词。

最后计算两个句子的相似度。将上述这两个词向量[1, 1, 1, 0, 1]、[1, 1, 0, 1, 1]和

n=5 代入公式（2.3）得到$\cos(\theta) = \frac{1+1+0+0+1}{\sqrt{1+1+1+1}\times\sqrt{1+1+1+1}} = \frac{3}{4} = 0.75$，即两个例句“It is raining today”和“It is sunny today”的相似度为 75%。

2.1.3 基于哈希算法的克隆检测技术

哈希算法又称为散列算法。对任意长度的输入求哈希值，总能得到长度相同的结果，常用来对敏感信息进行加密。输入中的任何一位发生改变，都有可能得到完全不同的两串哈希值，因此，为了可以检测第 2）类克隆，必须对被检测目标进行一定的变换[17]。

表 2 代码变量名标准化

原始代码示例	标准化代码示例
1: var fibonacci = function (index) {	1: var FUNC = function (FPARAM) {
2: if (index <= 2) { return 1; }	2: if (FPARAM <= 2) { return 1; }
3: return fibonacci(index - 1) +	3: return FUNC(FPARAM - 1) +
4: fibonacci(index - 2);	4: FUNC(FPARAM - 2);
5: }	5: }

如表 2 所示，左边的示例是一个用 JavaScript 语言编写的斐波那契函数，其中“var”、“function”、“if”和“return”称为 JavaScript 语言的保留字，“fibonacci”和“index”是用户自定义的函数名和变量名。在克隆代码时，为了保持上述函数的语义不变，只能修改函数名和变量名等非保留字。为了实现对第 2）类克隆的检测，对所有非保留字进行统一的标准化操作（如将函数名改为 FUNC，将参数名改为 FPARAM），结果如表 2 右边的标准化代码所示。

经过上述的操作，同一份代码的不同克隆副本将会得到相同的标准化结果，由此，只需不断地将待检测的代码进行标准化，然后对标准化后的代码计算哈希值，最后对哈希值进行对比。如发现两串相同的哈希值，即发现了克隆代码。这样便可以实现对第 1）和第 2）类克隆的检测。

2.2 软件测试方法

根据测试用例的生成方法和测试目标的不同，本文将软件测试方法分为基于断言的测试和模糊测试。

2.2.1 基于断言的测试方法

断言在软件测试和验证这类场景中有助于对程序的分析[18]。基于断言的测试一般用于测试目标明确的场景中，即测试人员清楚地知道要测试的是系统的哪个模块，然后有针对性地对相应的模块编写特定的测试用例[19]。在对模块执行测试用例的表现进行预估之后下定断言，接下来调用被测模块执行测试用例。最后将模块的执行结果和断言进行比对，如果一致则该模块通过测试，否则判定该模块存在缺陷。

表 3　基于断言的测试示例

被测模块	测试用例
1:　function divide (dividend, divisor) { 2:　　return dividend / divisor; 3:　}	1:　Assert (result = 5) 2:　divide (20, 4);

如表 3 所示，左边的被测模块是一个简单的除法函数 divide，它返回被除数除以除数的结果。右边的测试用例中调用了 divide 函数，并传入了参数(20, 4)，预期的到的结果是 5，因此设定断言 Assert (result = 5)。显而易见，若运行这个测试用例，该断言的判定结果应当为真。

2.2.2 模糊测试方法

模糊测试方法[20]是一种自动化的软件测试方法，人们已经通过它成功地发现了许多现实软件中的缺陷。在各种类型的模糊测试技术中，基于覆盖率的灰盒模糊测试特别受欢迎，它优先考虑分支探索，以便在难以到达的分支中有效地触发软件缺陷[21]。模糊测试使用修改过的或模糊化的输入对应用程序进行反复测试，目的是在输入解析代码中找到缺陷[22]。与基于断言的测试不同的是，模糊测试不针对系统中的特定模块，而是更加关注整个系统是否存在缺陷。通常情况下模糊测试只关注系统在正常运行过程中是否会发生崩溃[23]。

通常，模糊测试方法还包含一个自动产生测试用例的模块，使其具有自动化测试的能力。产生测试用例的模块分为测试用例生成器和测试用例变异器两种[24]。

（1）基于测试用例生成方法

使用生成方法即根据一定的方法或者规则生成测试用例以供测试，可以看作是全自动的测试。如图 5 所示，将对应编程语言的语法、代码和测试集输入之后，生成器具备了生成测试用例的能力。其首先将输入的代码拆分成小的片段，作为代码语料库。然后在语法正确的前提下，以测试集为指导，将上述语料库中的代码片段随机得拼接成新的测试用例[25]。

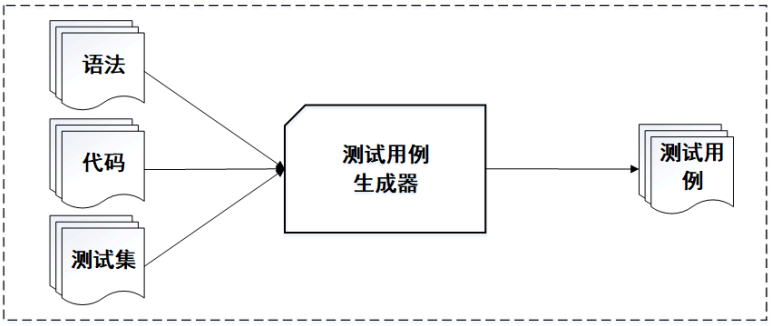

图 5 测试用例生成示例

（2）基于测试用例变异方法

而如果选择变异方法，测试者需要在测试开始时提供一组或多组原始的测试用例，在测试中，变异器会对这些原始测试用例进行随机的变异，即产生了新的测试用例[26]。

图 6 测试用例变异示例

仍以表 3 左侧被测模块的 divide 函数为例，若输入原始测试用例 divide(20, 4)，将返回正确结果 5。如果此时变异器恰好将除数 4 变异为 0，即要求 divide 函数返回 20/0 的结果。根据 divide 函数的逻辑将抛出异常，可能导致被测试系统的崩溃，由此便触发了一个缺陷。该示例如图 6 所示。

2.3 测试用例变异方法

区别于需要人工定义语法或模板的测试用例生成方法，变异方法所需的原始语料来源广泛，测试效果的好与坏取决于变异方法在设计过程中所选择的变异对象和变异方向。即对什么进行变异，把什么变异成什么。本小节将选择几个经典的和当前较新颖的变异方法进行介绍。

2.3.1 基于字节的测试用例变异方法

当前较为知名的模糊测试工具 AFL(American Fuzzy Lop)[27]使用了基于文本的测试用例变异方法。使用 AFL 时需要为它输入多份原始的测试用例，其通过变异方法逐一对这些测试用例进行多种变异，产生多份新的测试用例。以期通过这些变异策略将原本可以正常执行的测试用例，变异成可以触发被测试对象隐含的缺陷的异常用例。AFL 设计了 6 种变异策略[28]，如表 4 所示。

表 4 AFL 的变异策略

变异策略	说明	示例
bitflip	按位翻转，1 变为 0，0 变为 1。 参数 m/n 表示每次翻转相邻的 m 个 bit，按照每 n 个 bit 的步长从头开始。	原始用例： var variable = 1; 变异用例： 1）$ar variable = 1; 2）v_r variable = 1; …
arithmetic	对位进行整数加/减算术运算。 对目标整数会进行+1, +2, ..., +35, -1, -2, ..., -35 的变异。参数 m/n 表示每次对 m 个 bit 进行加减运算，按照每 n 个 bit 的步长从头开始，即对文件的每个 byte 进行整数加减变异。	原始用例： var variable = 1; 变异用例： 1）yar variable = 1; 2）vqr variable = 1; …
interest	把一些特殊内容替换到原文件中。 参数 m/n 表示每次对 m 个 bit 进替换，按照每 n 个 bit 的步长从头开始，即对文件的每个 byte 进行替换。	原始用例： var variable = 1; 变异用例： 1）512ar variable = 1; 2）v-32768r variable = 1; …
dictionary	把自动生成或用户提供的 token 替换/插入到原文件中。	原始用例： var variable = 1; 变异用例： 1）let variable = 1; 2）var variable (1; …
havoc	大破坏，对原文件进行大量随机变异。	原始用例： var variable = 1; 变异用例： 1000 bar=call ;

splice	将两个文件拼接起来得到一个新的文件。	原始用例：
		1）var variable = 1;
		2）var num = 8547;
		print(num % 2 == 0);
		变异用例：
		var variable = 1;
		var num = 8547;
		print(num % 2 == 0);

从表 4 中可以看出，其变异方法所执行是基于代码文本的字节级别的操作。其接近于穷举方法，即将所有可能的情况都做一次尝试，这样的变异操作由于不具有导向

性，且效率十分低下，并且时常不能深入地探究和充分地利用测试用例[29]。

下面列举一个特殊的测试用例，对基于字节的测试用例变异方法效率低下的问题予以更加深入的分析。

```
var snow_white_story = "Once there was a Queen. She was sitting at the window. \nThere was snow outside in the garden--snow on the hill and in the lane, snow on the huts and on the trees: all things were white with snow.\nShe had some cloth in her hand and a needle. The cloth in her hand was as white as the snow.\nThe Queen was making a coat for a little child. She said, \"I want my child to be white as this cloth, white as the snow.\nAnd I shall call her Snow-white.\"\nSome days after that the Queen had a child. The child was white as snow. The Queen called her Snow-white.\nBut the Queen was very ill, and after some days she died. Snow-white lived, and was a very happy and beautiful child.\nOne year after that, the King married another Queen.\nThe new Queen was very beautiful; but she was not a good woman."

print(snow_white_story);
```

图 7 针对基于字节的测试用例变异方法的一个特殊的测试用例示例

如图 7 所示的一个较为极端的测试用例，其语义即定义一个长字符串，然后输出它。如果将其输入到 AFL 中进行变异和测试，AFL 的变异器会对用例的每一位进行一次按位翻转，每个英文 ASCII 字符占 8 位，字符串 snow_white_story 中有数百个字母，即这一操作就要进行上千次。然而这上千次的按位变异只会改变字符串的值，并不能对测试的效果产生太大的影响。其后，剩下的 5 种变异规则会被逐一运用在该用例上，之后才会再开始处理下一个用例。可见，对于这类文本量很大，但语义信息很少的测试用例，AFL 对其进行大量穷举变异的效率十分低下。

2.3.2 基于语法树的测试用例生成和变异方法

另一个最新的模糊测试工具 Nautilus[30]设计了一种基于语法树的上下文无关的树表达式。其在产生测试用例时，首先根据写定的语法规则产生语法树表达式。然后使用上述表达式构建语法树，接着按照设计的变异策略对语法树中的节点进行变异。最后再将语法树转译为代码，这样便达到了对产生的代码的变异的目的[31]。其变异策略如表 5 所示。

表 5 Nautilus 的变异策略

变异策略	说明
Random	选择输入树的一个随机节点，将其替换为一个随机生成的新子树
Rules	依次用所有其他可能的规则生成的子树替换输入树的每个节点。
Random recursive	随机选择输入树的一个子树并重复这个子树 2^n 次(1≤n≤15)。这将使输入树具有高度嵌套结构。
Splicing	用生成的另一棵树替换掉输入树的一个子树。
AFL	由于 AFL 的变异器只对字符串进行操作，因此在使用这种变异之前，输入树会被转换为文本形式。Nautilus 选择了 AFL 的 6 种变异策略中的 3 种： 1）bitflip，按位翻转。 2）arithmetic，对位进行整数加/减算术运算。

	3）interest，把一些特殊内容替换到原文件中。

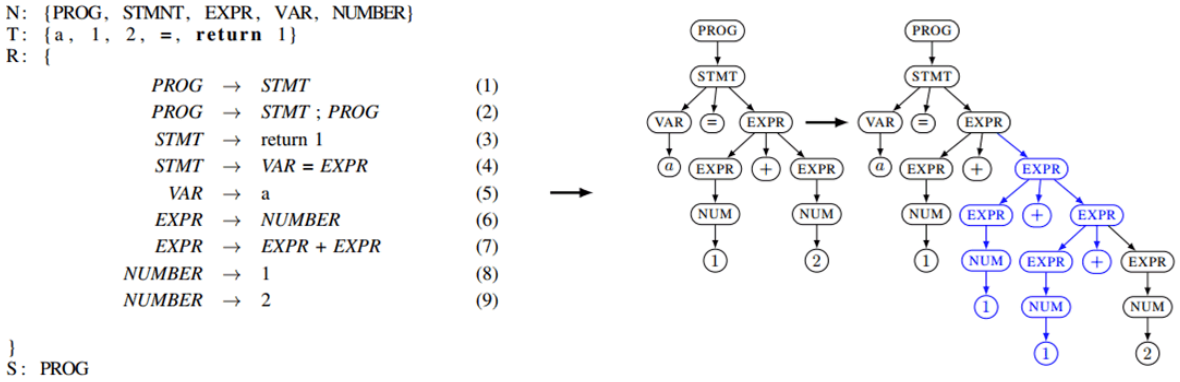

图 8 Nautilus 产生和变异测试用例的示例[31]

从图 8 和上述介绍中可以看出，Nautilus 组合了基于语法的测试用例生成器和测试用例变异器而构建了一个独特的测试用例生成和变异器。除了变异器的变异策略不具有导向性（即不经过其他方法引导的随机变异）外，这类基于语法的代码产生模块还有一个严重的缺陷，即用于生成测试用例的语法的好坏受制于编写语法规则的人的知识储备、实践经验和对这门语言的熟悉程度。如果所设计的语法的语义信息不够复杂和丰富，其所生成的代码所具有的语义信息也会相应地大打折扣。如果这些原始测试用例中很少隐含着可以触发引擎缺陷的语义片段，即便多么复杂的变异器设计都对实际的测试效果无济于事。

2.4 本章小结

本章介绍了软件测试相关的理论与技术。首先介绍了基于文本向量化和哈希算法的两种代码克隆检测技术。前者将代码进行向量化，使用余弦相似度算法计算向量的相似度。后者对代码中的非关键字进行统一，使用哈希算法对两份代码进行比对。本章还介绍了目前比较常见的两种软件测试方法。基于断言的测试方法在使用时目标明确，测试结果没有局限性，但是受限于测试人员的效率低下。模糊测试方法在使用时并不设定确切的测试目标，且具有产生测试用例的能力。最后本文列举了两种测试用例变异方法。其中，基于字节的测试用例变异方法，其按字节进行逐位变异，测试效率较为低下。基于语法树的测试用例生成和变异方法，原始测试用例质量的好坏受制于人工定义的语法规则的严谨性和丰富度，因此会影响用例变异和测试的效果。

第三章 基于类型推断的 JavaScript 引擎模糊测试方法

上一章中提到，一个模糊测试方法中最为核心，最重要的模块是测试用例的产生模块。而测试用例产生模块又分为生成器和变异器两种。生成器的问题在于，基于引导而随机组合起来的代码片段间很难具有较强的语义联系，进而形成一块完整的语义丰富的测试用例。而变异器所需的原始语料来源广泛，测试效果的好与坏取决于如何设计变异的目标，即选择对原始语料中的哪些部分进行什么程度上的变异。由此，本章将系统地介绍本文所提出的基于类型推断的 JavaScript 引擎模糊测试方法，该方法所能产生的测试用例具有更强的语义，并且可以大幅度地提高测试用例的代码覆盖率和模糊测试的效率。

3.1 方法概述

本文所涉及的基于类型推断的 JavaScript 引擎模糊测试方法主要的目标是，针对 JavaScript 语言及其引擎设计一个能高效产生优质测试用例的产生模块，以产生具有强语义，高代码覆盖率[32]的测试用例，以此为基础设计一个模糊测试方法。进而提升发现 JavaScript 引擎缺陷的效率。整个模糊测试方法的结构如图 9 所示。

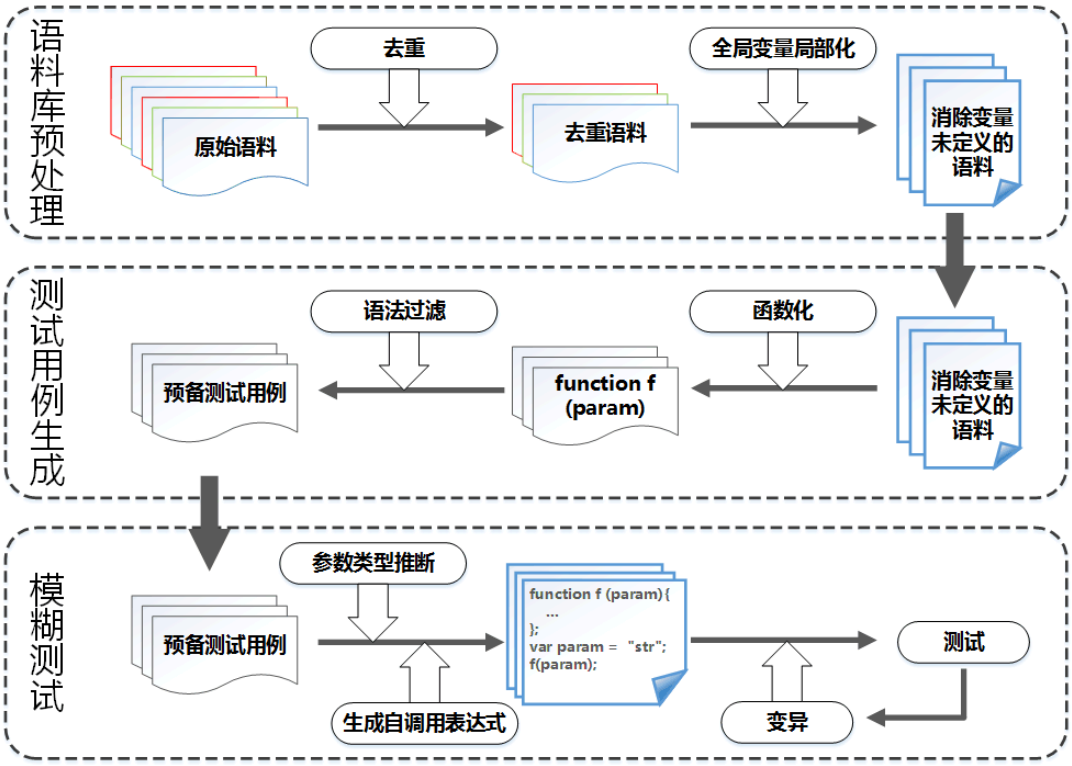

图 9 基于类型推断的 JavaScript 引擎模糊测试方法结构

从图 9 可以看出，本文所设计的方法其流程包含语料库预处理、测试用例生成和模糊测试三个部分：

（1）语料库预处理方法设计

从各种渠道收集而来的原始 JavaScript 代码文件在本文中称作语料库。语料库预处理的目的是对这些原始的 JavaScript 文件进行初步的筛选和处理。语料库去重的步骤可以过滤掉重复的代码文件，进而减少后面测试以及测试结果分析中的重复工作量。而全局变量局部化的步骤可以通过对抽象语法树的操作，解决因变量未定义而导致的测试用例运行提前退出，代码覆盖率降低的问题。

（2）测试用例生成方法设计

经过预处理的语料可以被用来生成预备测试用例。函数化操作解决了原始语料在被执行时，代码覆盖率低的问题。它将（1）中得到的语料拆分成 JavaScript 语言的函数[33]，这是本文中测试方法所需的基本单元。经过函数化得到的函数可能会包含一些语法错误。检查代码的语法错误是引擎在执行代码前所需的必要步骤，包含语法错误的函数并没有执行的收益，反而会带来为尝试它而引入的时间开销，因此需要通过语法检查将其过滤。

（3）模糊测试方法设计

2.2.2 小节中简单介绍过模糊测试，它通过不断输入不同的测试用例让待测目标去执行以发现其隐含的缺陷（以崩溃类型为主）。经过（1）和（2）的处理，得到的预备测试用例尚不能直接用来测试，因为单独的函数不能自动执行，必须从外部调用它，才可以执行其内部的代码逻辑。更进一步，对于占多数比例的有参数的函数而言，调用它们时还需要传入相应的参数，此时便通过本文中设计的类型推断方法推断出参数可能的数据类型，然后根据推断结果传入相应类型的参数并调用该函数，以此来提高测试用例的代码覆盖率，进而提高模糊测试的效率。最后，在测试中对经过了精准传参的函数使用变异器进行参数值范围的变异，以期可以触发更多由数值临界条件引起的引擎崩溃缺陷。

3.2 语料库预处理

语料库中的原始 JavaScript 代码可以有许多来源，如一些开源的代码网站或线上代码托管仓库（如 GitHub）等。为了尽可能地提高语料库中代码的语义丰富程度，需要尽可能多地收集一些代码，而这又同时带来了另一个问题——重复收集的代码比

例提高，这会导致冗余测试等问题。因此需要使用 2.1 小节中提到的相应方法进行去重。而最后的全局变量局部化操作解决了“变量未定义”的问题。整个语料库预处理中对代码所执行的变化如图 10 所示。

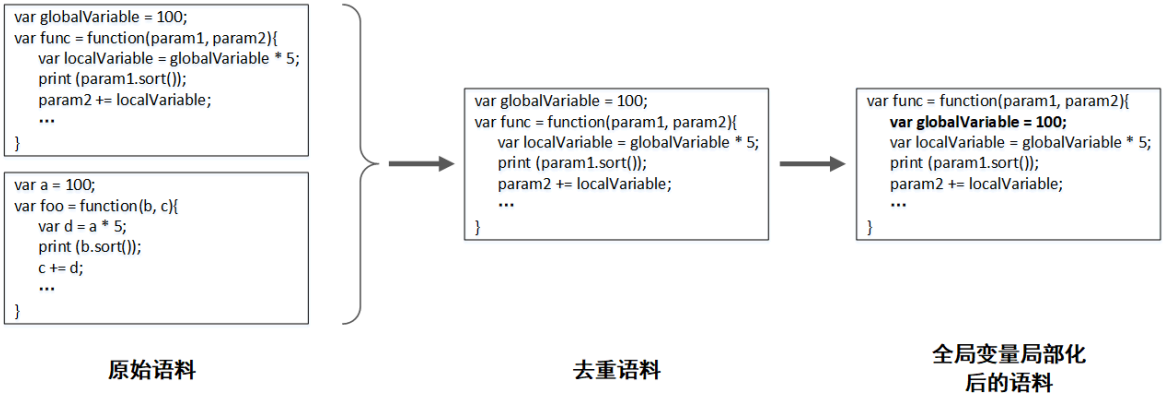

图 10 语料库预处理代码变化

3.2.1 语料库去重

本文中所需的代码克隆检测目的是尽可能地去除重复的 JavaScript 代码，因此仅使用 2.1.2 小节中介绍的基于文本向量化的克隆检测技术来对语料库进行去重。

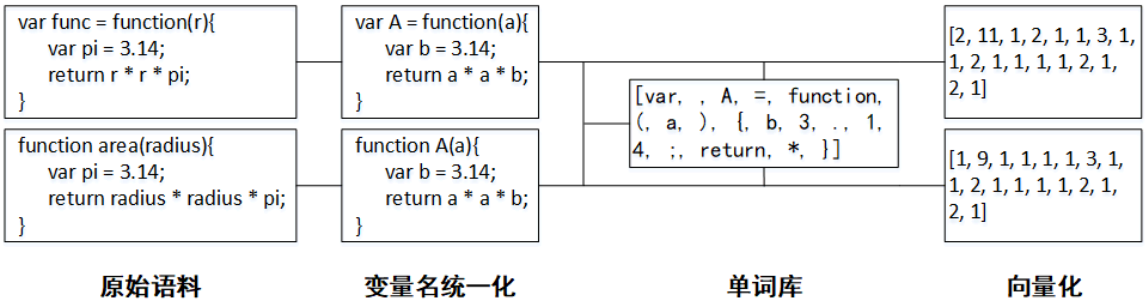

图 11 JavaScript 代码向量化示例

不同的是，为了解决第 2）类代码克隆，即克隆代码之后修改某些变量名、变量值等字面量的情况，在使用文本向量化技术之前，需要对 JavaScript 代码进行变量名统一化替换。如图 11 所示，首先将两段需要进行克隆检测的代码进行变量名统一化处理，即按照一定的规则对代码中的变量进行变量名替换。如图 11 中使用的变量名替换规则为：函数的名称以大写字母来替换（如 A, B, C, …），其他变量名以小写字母来替换（如 a, b, c, …）。接下来对两份代码进行切词，得到一个单词库，如图 11 中第三个部分所示。这个单词库包含了上述两份代码中的所有单词（如 var，A，空格等）。然后分别统计出两份代码中每个单词的词频，得到两个词向量[2, 11, 1, 2, 1, 1, 3, 1, 1, 2, 1, 1, 1, 1, 2, 1, 2, 1]和[1, 9, 1, 1, 1, 1, 3, 1, 1, 2, 1, 1, 1, 1, 2, 1, 2, 1]，最后，将上述两个向量带入公式（2.3）中，得到余弦相似度为$\cos(\theta) \approx 0.9921$，即这两份

JavaScript 代码的相似度为 99.21%，故应该删除其中的一份，仅保留另一份。

值得注意的是，如果不做变量名统一化操作，使用上述步骤计算得到的两份代码的相似度仅为$\cos(\theta) \approx 0.9186$，即 91.86%。由此可见变量名统一化操作可以提升代码克隆检测的检测效果。

3.2.2 全局变量局部化

代码的全局变量[34, 35]是定义在一个大的作用域上的变量，在任何一个更小的局部作用域中可以应用并共享全局变量。如图 12，如果直接对图中的原始语料进行函数化，那么提取函数的时候，第一行的全局变量定义 “var globalVar = 50;”将被丢弃。这样做之后得到的函数在被执行时，引擎会在执行到第二行的 “var localVar = globalVar * 5;” 语句时抛出 “globalVar 未定义” 的异常，程序至此异常终止，这个函数后面的数行代码都会被浪费掉，造成测试效率和效能的损失。

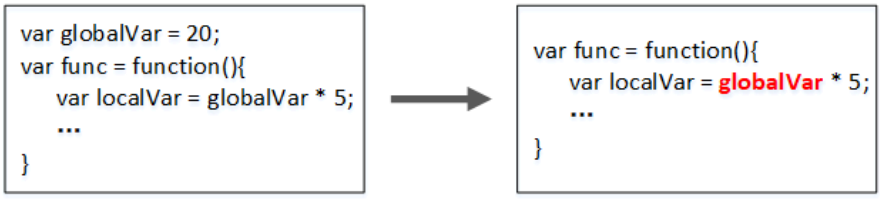

图 12 直接函数化导致的变量未定义问题示例

因此，为了避免因人为的预处理而造成这种 “变量未定义” 情形的增加，需要对代码中的全局变量做局部化预处理操作。其步骤如图 13 所示。

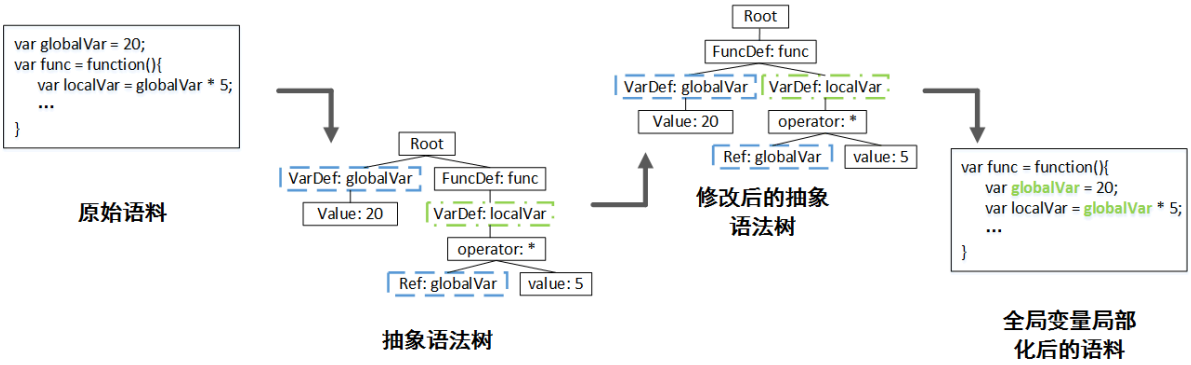

图 13 全局变量局部化过程示例

（1）首先解析出原始语料的 JavaScript 抽象语法树。抽象语法树是代码的树形结构表现形式，与代码一一对应。通过它可以很方便得对代码结构进行分析和做动态的调整，有的 JavaScript 引擎在执行代码之前也会先提取抽象语法树。

（2）然后根据局部变量与全局变量在抽象语法树中的父子节点关系，将全局变量所在节点复制或移动成为相关局部变量的兄弟节点。如图 13 中，“VarDef: localVar”

的子节点“Ref: globalVar”引用了其三层父节点“VarDef: globalVar”，将“VarDef: globalVar”移动到“FuncDef: func”下，作为“VarDef: localVar”的前一个兄弟节点，就完成了全局变量局部化在抽象语法树中的节点变换操作。

（3）最后再将该修改后的抽象语法树文本化为代码形式，得到全局变量局部化后的语料。

现在再进行原始语料的函数化预处理操作，就不会丢失原来的全局语义信息。如此便间接地提高了原始语料代码的利用率，即最终测试用例的代码覆盖率。

3.3 测试用例生成

经过了预处理阶段的各种操作，去除了语料库中重复的语料，并且消除了可能会由人为拆分引入的“变量未定义”问题。但得到的结果仍是整份的 JavaScript 代码。前文中提到，为了提高测试用例的代码覆盖率，需要将语料库预处理阶段得到的结果进行拆分。进而获得本文中执行测试所需的最小单元——JavaScript 的函数。测试用例生成阶段对语料库所做的操作如图 14 所示。

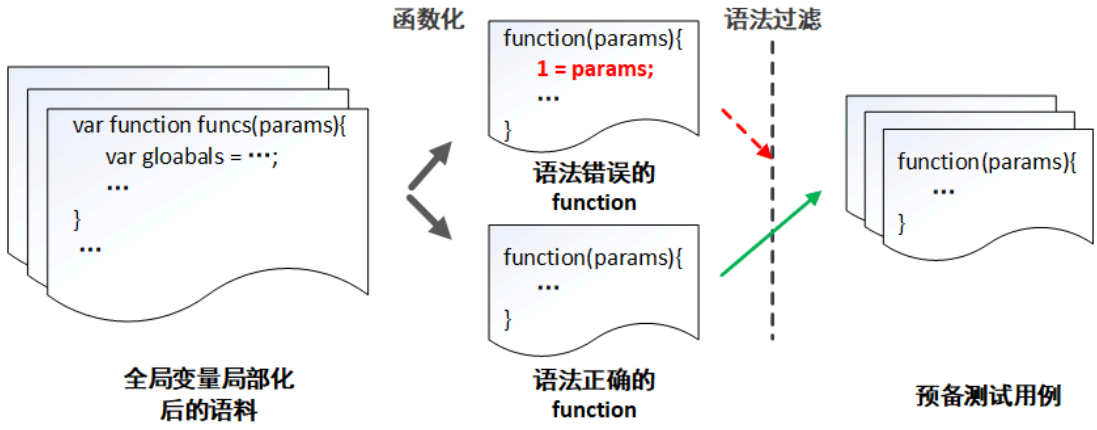

图 14 测试用例生成的代码变化

3.3.1 函数化

函数化操作顾名思义，即将代码转化为函数。在 JavaScript 语言中，函数通过“function”关键字来定义。在收集自各种渠道的 JavaScript 代码中，存在着以下两种可以导致其代码覆盖率低的情况：1）由于代码的前半部分含有语法错误，导致后面的代码无法被引擎执行到。2）有的代码本身就是一个函数库，其中的函数包含丰富的语义信息，但是由于这些函数需要外部的调用才能执行，因此如果不做处理，这些语义丰富的代码无法产生实际的测试作用。这种情况并不在少数，当前较为流行的许

多 JavaScript 框架如 jQuery[36]、Bootstrap[37]等内部含有对象操作、渲染操作、网络请求等多种功能的大量函数，这些含有丰富语义的代码是用来测试 JavaScript 引擎的很好的测试用例来源。

3.3.2 语法过滤

经过函数化的操作，得到了大量的函数，但这些函数并非完全符合 JavaScript 的语法规则。语法检查是一种静态检查，即在引擎执行代码之前对代码文本进行的检测，如果代码中有任何的语法错误，引擎不应当执行这段代码，而应该直接抛出异常，退出执行。

因此，包含语法错误的函数并不会被执行，而且还会因为引擎的启动和退出时延降低测试的效率，因此需要对包含语法错误的函数进行过滤删除。如图 14 中的示例，含有为常量 1 赋值操作的函数无法通过语法检测。通过了语法过滤的函数在本文中称为预备测试用例。

3.4 参数类型推断

经过了上述的一系列操作，得到了经过去重、消除变量未定义，并进行了语法过滤的函数集合，称为预备测试用例。现在，需要讨论的是如何使用这些预备测试用例，更进一步的即如何调用这些经过了一系列优化处理的函数？为了解决这个问题，本文设计了一种参数类型推断方法。

3.4.1 参数类型推断的作用

根据前文中的说明，一个函数如果不被显式得调用，其函数体中的代码逻辑不能得到执行，将其作为测试用例是无效的。因此需要为每一个测试函数生成调用语句，使其真正地发挥测试 JavaScript 引擎的作用。针对一些没有声明参数（即没有参数）的函数，可以直接用其函数名生成调用语句。例如一个函数声明为“var test_func = function(){ … };”，可以为其生成调用语句“test_func();”，即可使被测试的引擎执行函数体中的代码逻辑。

但针对一些具有参数的函数，如果不传入参数或者随机传入参数，就有很大的可能性会再次引入“变量未定义”的问题，导致 JavaScript 引擎对函数体中的程序执行提前结束，造成测试用例代码覆盖率降低。因此，本文提出了参数类型推断方法，就是要解决有参函数其参数的数据类型无法确定的问题，为后续步骤中精准地传参，进而

为提高测试用例的代码覆盖率做有力的保障。

3.4.2 参数类型推断方法说明

参数类型推断方法通过对函数的参数进行预先的静态文本分析，来确定其可能的数据类型。如图 15 中的例子所示，该函数有一个参数 param，函数体中有三行代码。代码语义是，首先将 param 转换为小写形式，然后获取它的长度，最后返回它的后二分之一。

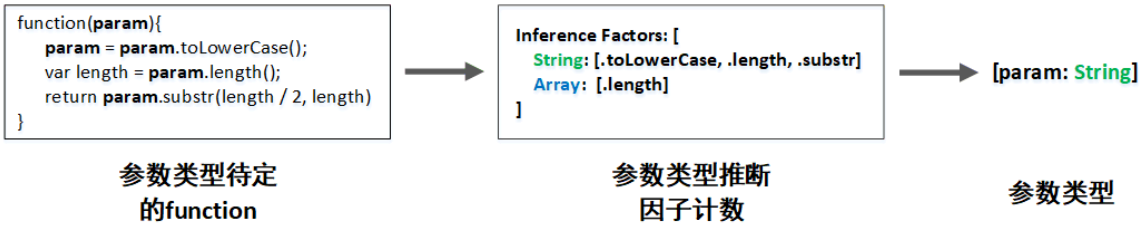

图 15 参数类型推断示例

在对参数 param 进行的类型推断过程中，首先对参数类型待定的函数的函数体进行遍历扫描，统计函数体中与参数 param 相关的各种数据类型的推断因子，然后对各类型推断因子的个数进行比较，匹配的推断因子数量最多的数据类型即被推定为该参数 param 的类型。如图中的例子，与 param 相关的 String 类型的推断因子有[.toLowerCase, .length, .substr]等 3 个，而 Array 类型的推断因子只有[.length]这 1 个，因此推定参数 param 所需的数据类型为 String，即字符串类型。

3.4.3 类型推断因子

本小节对类型推断因子进行详细的说明。为了尽可能精确地标识一种数据类型的变量在 JavaScript 代码中所具有的特有表现，本文提出了类型推断因子的概念。类型推断因子，即一种数据类型在代码层面所具有的独有功能的集合。如图 15 中的代码示例，“.toLowerString”和“substr”是 JavaScript 语言中 String 类型特有的函数，其语义分别为获得字符串的小写字母形式和获取字符串的子串。而“.length”是 String 类型和 Array 类型都具有的函数，即分别表示获得字符串和数组的长度。因此“.length”被分别设计为 String 和 Array 两种数据类型的推断因子。

本文中所设计的类型推断因子不仅仅包含各数据类型所独有的函数，还包含了从一些类型的变量所特有的操作中截取的文本片段，如“ == true”判断变量值是否为真，常与 Boolean 类型的操作相关，所以作为 Boolean 类型的一个推断因子；“++”自增操作可以使 Number 类型的变量的值自增 1，因此作为 Number 类型的推断因子；

“[”和“]”常常与 Array 类型的变量一同出现，用于指示下标为指定值的数组元素，因此可以作为 Array 类型的推断因子。

总之，除了上述简单介绍的以外，本文针对 JavaScript 语言的 Array、Boolean、Number、String 和 Function 这 5 种数据类型设计了许多的类型推断因子，具体如表 6 所示。

表 6 各数据类型的推断因子

数据类型	前缀推断因子	后缀推断因子	环绕推断因子
Array		".length", ".concat", ".join", ".pop", ".push", ".reverse", ".shift", ".slice", ".sort", ".splice", ".toSource", ".toLocaleString", ".unshift", "[", " ["	
Boolean	"true==", "true ==", "false==", "false =="	".toSource", "=true", "= true", "=false", "= false", "==true", "== true", "==false", "== false"	
Number	"+=", "+= ", "-=", "-= ", "*=", "*= ", "/=", "/= ", "++", "++ ", "--", "-- ", "%", "% ", "%=", "%= ", "<<", "<< ", ">>", ">> ", "<<=", "<<= ", ">>=", ">>= ", ">>>", ">>> ", "&=", "&= ", "^=", "^= ", "|=", "|= "	".toLocaleString", ".toFixed", ".toExponential", ".toPrecision", "+=", "+= ", "-=", " -=", "*=", " *=", "/=", " /=", "++", " ++", "--", " --", "%", " %", "%=", "%= ", "<<", "<< ", ">>", ">> ", "<<=", "<<= ", ">>=", ">>= ", ">>>", ">>> ", "&=", "&= ", "^=", "^= ", "|=", "|= "	
Function		".call", "(", " (", "=function", " =function", "= function", " = function"	["function", "("], ["function ", "("], ["function", " ("], ["function ", " ("]
String		".length", ".anchor", ".big", ".blink", ".bold", ".charAt", ".charCodeAt", ".concat", ".fixed", ".fontsize", ".indexOf", ".italics", ".lastIndexOf", ".link", ".localeCompare", ".match", ".replace", ".search", ".slice", ".small", ".split", ".strike", ".sub", ".substr", ".substring", ".sup", ".toLocaleLowerCase", ".toSource", ".toLowerCase", ".toUpperCase", ".toLocaleUpperCase", "[", " [", "=\"", " =\"", "= \"", " = \""	

从表 6 中可以看到，对于不同的数据类型，本文设计了前缀、后缀和环绕三种

类型推断因子。除了大多数以“.”号开头的表示函数名的后缀推断因子外，Boolean 和 Number 具有一些前缀推断因子，同时为 Function 类型设计了一些环绕推断因子。

需要注意的是，当在一些特殊的情况下，如果一个函数的参数在两个及两个以上的数据类型上统计出来的类型推断因子数相同，则可以认为这些数据类型都可能是这个参数所需的，然后对该参数给出一个包含所有可能类型的推断结果集合。另外，当一个参数在所有数据类型上统计出来的推断因子数都为 0，那么将为它推定结果“none”，即没有类型。上述推断结果将作为下一个步骤的输入。

3.5 模糊测试

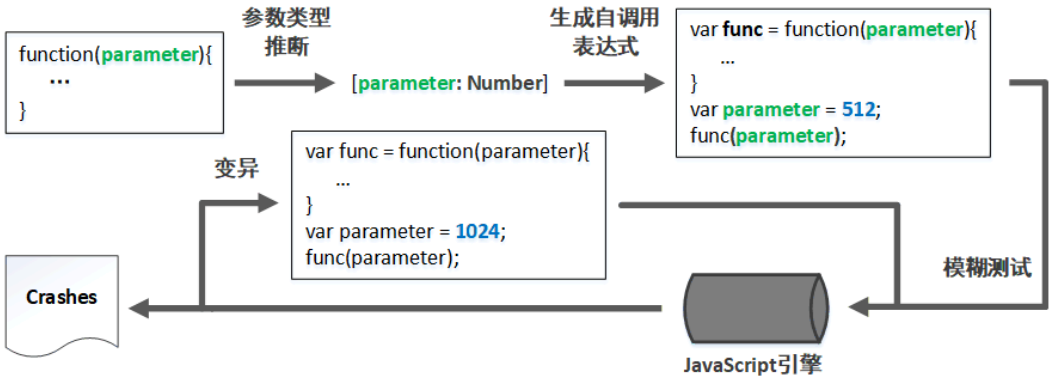

图 16 模糊测试阶段的代码变化

进行模糊测试是上述所有步骤最终的目的，整个模糊测试过程中测试用例的变化如图 16 所示。首先，通过参数类型推断操作来推断出函数的参数 parameter 可能的数据类型，然后根据推断结果生成相应类型的实际参数，最后使用这个参数调用该函数执行函数体中的内部代码逻辑。为了提高每个测试用例的测试效能，更多地触发因数值边界条件引发的引擎缺陷，对一个测试用例所传的参数还会经过多轮的变异。最终当一些测试用例达到临界条件时，将触发 JavaScript 引擎的缺陷，引起崩溃。

3.5.1 生成自调用表达式

前文中曾提到过，未被显式调用的函数，其函数体中的代码逻辑不会被执行，这也是本文对原始 JavaScript 语料进行函数化的初衷，即让尽量多的测试用例得到执行，提高测试用例的代码覆盖率。经过 3.4 小节中所做的参数类型推断，已经推定了函数中参数的数据类型，现在需要根据上述推断结果生成相应的实际参数，然后使用这些实际参数产生调用这些函数的代码语句，即自调用表达式。

针对不同的数据类型和推断结果，本文设计了不同的参数生成策略。具体的说明

和示例如表 7 所示。

表 7 各数据类型参数的生成策略及示例

推断结果	代表类型	生成策略	示例
[param: Boolean]	布尔类型	以等概率随机生成 true 或 false。	true 或 false

[param: Number]	数值类型	以等概率随机生成一个数值范围在$[-10^{10}, 10^{10}]$之间的整数或浮点数。	var param = -809;
			或
			var param = 66.1059660754950772;

[param: String]	字符串类型	长度为[1, 128]之间的 ASCII 字符集随机字符组成的字符串。	var param = "U%$&T<1}R\\G~;PNR5fm2Kq_c5l0)";
[param: Array]	数组类型	以等概率生成以下两种数组之一： （1）成员类型相同的数组； （2）成员类型不同的数组。 数组的长度为一个[0, 15]之间的随机数。	var param = [-1938764, 2317144958, 76, 6921, -127496057, 1940] 或 var param = [null, true, ";+x", -914];

[param: function]	函数类型	预备测试用例库中的一个随机的函数。	var param = function(a, b){
			return a * b;
			};

[param: none]	随机类型	随机分配一种数据类型的值。	

需要进一步说明的是，数组类型的成员仍可以是数组类型，即可以生成多维嵌套数组。当类型推断无法推定参数的数据类型时，将会为[param: none]的推断结果随机生成上述其他类型的一个值。

最后，使用函数名和参数名生成函数的调用表达式，即自调用表达式。如图 16 的右上方所示，一个最终的测试用例由函数、实际参数列表和自调用表达式三个部分组成。

3.5.2 执行模糊测试

经过了上述一系列的处理，得到了最终的测试用例。接下来介绍本文的最终目标，即使用这些测试用例对 JavaScript 引擎展开模糊测试。在本文第二章中曾提及，模糊测试是一种黑盒测试，它通过输入大量的测试用例，来发掘被测试对象中隐含的代码缺陷，通常为崩溃类型的缺陷。

模糊测试方法一般都具有一个测试用例的产生模块它是模糊测试方法最重要的模块。框架测试效能的高低取决于其产生测试用例的质量。在本文所设计的模糊测试方法中，测试用例产生模块由基于类型推断的 JavaScript 测试用例生成模块和变异模块两部分组成。在前文中已经介绍了基于类型推断的 JavaScript 测试用例生成模块，

有了它便可以更加精确地指导变异器对测试用例进行变异，从而避免了现有的工作中对测试用例进行的按位变异导致的测试效率低下的问题。

了解了测试用例产生模块，接下来介绍测试模块。在执行模糊测试时，每次创建一个进程，在这个进程中调用 JavaScript 引擎去执行测试用例。然后在进程中监控 JavaScript 引擎的执行情况，即是否正常执行结束、异常退出或者崩溃。这些进程状态的监控和判断是通过系统的进程返回码和信号[38]来确定的。

表 8 Linux 操作系统常见的进程返回码和信号

信号	返回码	指代的进程状态
SIGHUP	1	进程挂起信号
SIGINT	2	进程终止信号, 在用户键入 INTR 字符(通常是 Ctrl-C)时发出
SIGQUIT	3	进程退出信号, 在用户键入 QUIT 字符(通常是 Ctrl-\)时发出
SIGILL	4	非法指令信号，程序指令不合法
SIGABRT	6	进程终止信号
SIGBUS	7	总线错误信号
SIGFPE	8	浮点异常信号
SIGSEGV	11	段违例信号，进程执行了无效的内存引用，或发生段错误
SIGSYS	31	非法系统调用信号

在 Linux 操作系统中，进程中执行一段正常的程序其返回码为 0，非 0 的返回码可能意味着程序的执行发生了某些异常或错误。常见的 Linux 操作系统进程返回码和信号如表 8 所示，除了 SIGINT（返回码 2）和 SIGQUIT（返回码 3）是由用户输入导致的进程终止和退出外，其他的信号都可能是程序中的错误引发的。

本文所设计的模糊测试方法即首先创建一个进程，然后在进程中调用被测试的 JavaScript 引擎去逐个执行前面的步骤中准备好的测试用例。最后通过监控进程的返回码来判断 JavaScript 引擎是否发生了崩溃，即是否触发了其隐含的缺陷。如果返回码为 0 则表示该 JavaScript 引擎执行当前测试用例状态正常。如果返回码不为 0，则认为该引擎在执行当前测试用例时可能发生了崩溃，随即保存当前测试用例，最后进行人工的检验，如果可以复现崩溃情形，则向该引擎的维护者提交缺陷报告。

3.5.3 测试用例变异

在 2.3 小节中，本文介绍了两种测试用例变异方法，并分析了现有的测试用例变异方法存在的问题，即其使用的变异策略对每一份测试用例进行无差别的随机穷举变异，效率低下，降低了模糊测试的整体效率。

本文设计了一种基于类型推断的具有引导性的测试用例变异器。为了使测试用例

变异器具有引导性，进而有针对性地对代码进行变异，本文对函数的参数进行类型推断，当函数的参数所需的数据类型被成功推定后，变异器就可以有针对性地对这一类的参数的值进行有效的变异。

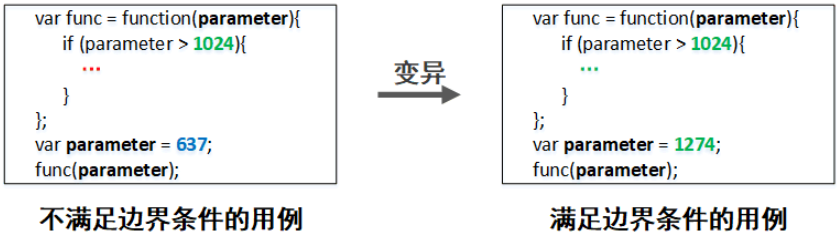

图 17 基于类型推断的测试用例变异示例

如图 17 的示例所示，函数 func 的参数 parameter 所需的数据类型被推断为 Number 类型，首次传参时给出的值为 637，它不能通过函数体中的 if(parameter > 1024)的条件判断，因此无法深入执行下面的代码块。而当变异器针将 Number 类型的 parameter 的值由 637 变异为其 2 倍的 1274 时，就可以进入到条件判断的代码块中执行，提高了测试用例的代码覆盖率。进而可以执行到更多的测试用例代码逻辑，提高了触发引擎缺陷的可能。

表 9 基于类型推断的测试用例变异策略

数据类型	变异策略
Boolean	1）翻转：true -> false，false -> true
String	1）范围删除：在输入字符串中随机删除其中的一部分； 2）范围插入：在输入字符串中随机插入一段字符串； 3）范围复制：将字符串中随机的一段复制一份插入到其中； 4）按位变异：对输入字符串的随机一位，在 ASCII 字符集内进行随机替换； 5）字节变异：对输入字符串中的随机一位，与范围(0, 255)内的一个随机数做异或运算； 6）随机交换：随机交换输入字符串中的两位； 7）算术运算变异：对输入字符串中的随机一位进行算数+1 或- 1 的操作。
Number	1）四则混合运算变异：对输入数值进行+、-、*或/ 2 的操作； 2）平方运算：对输入数值求平方值； 3）平方根运算：对输入数值求平方根值； 4）镜像变异：对输入数值取反，如 mutate(200) = -200； 5）浮点变换：对整型输入数值添加随机的小数部分，对浮点型输入数值消除其小数部分。
Array	1）翻转数组：将输入数组按位倒置，如 mutate([1, 2, 3]) = [3, 2, 1]； 2）数组减半：将输入数组的长度缩短为原来的一半； 3）数组倍增：将数组的长度延长为原来的两倍，新的一般元素为原数组的复制； 4）随机交换：随机交换输入数组中的两个指定下标的值。

Function	1）函数替换：从预备测试用例库中随机抽取一个函数替换输入函数； 2）函数倍增：将输入函数的函数体复制一份添加在原函数体后面； 3）函数合并：从预备测试用例库中随机抽取一个函数，将其函数体与输入函数的函数体合并；

针对前文中提到的 Boolean、Number、String、Array 和 Function 这 5 种数据类型，本文分别设计了许多相应的变异策略，以使测试用例可以在变异的过程中具有更加广泛的值范围，进而进入更多的条件代码块中触发 JavaScript 引擎中更多的临界条件。具体的设计如表 9 所示。

3.6 本章小结

本章主要介绍了本文所涉及的基于类型推断的 JavaScript 引擎模糊测试方法的设计。首先从整体介绍了框架中包含的三大模块，即语料库预处理模块、测试用例生成模块和模糊测试模块。其中在语料库预处理的过程中，需要对从 GitHub 上收集的原始语料库进行去重和全局变量局部化的操作，以减少重复语料并提前消除“变量未定义问题”出现的可能。在测试用例生成阶段，需要对预处理后的语料进行函数化和语法过滤操作，其中前者用于提高测试用例的代码覆盖率。而后者删除了不符合语法规则的语料减少了测试效率的浪费。在最后也是最重要的模糊测试阶段，经过了参数类型推断后，确定了预备测试用例的参数所需的数据类型，然后对特定的类型生成特定的实际参数，进而生成自调用表达式，用这些生成的参数去调用目标函数，以进一步提高测试用例的代码覆盖率。进而，针对每种数据类型的值进行具有引导性的测试用例变异，相较于使用随机变异策略的模糊测试方法，具有更高的变异效率。最终设计了一套基于 Linux 系统进程返回码和信号的方法，以监控测试结果，判断是否触发了被测试 JavaScript 引擎的崩溃缺陷。

第四章 JSTIFuzz 原型系统的设计与实现

上一章中介绍了本文所设计的基于类型推断的 JavaScript 引擎模糊测试方法，根据该方法，本文设计并实现了相应的原型系统，称为 JSTIFuzz。本章将对 JSTIFuzz 原型系统的设计与实现进行详细的介绍。

4.1 系统模块设计

JSTIFuzz 原型系统共分为语料库搜集模块、语料库预处理模块、测试用例生成模块、模糊测试模块和测试结果处理模块等 5 个主要模块。整个系统的模块层级关系如图 18 所示。

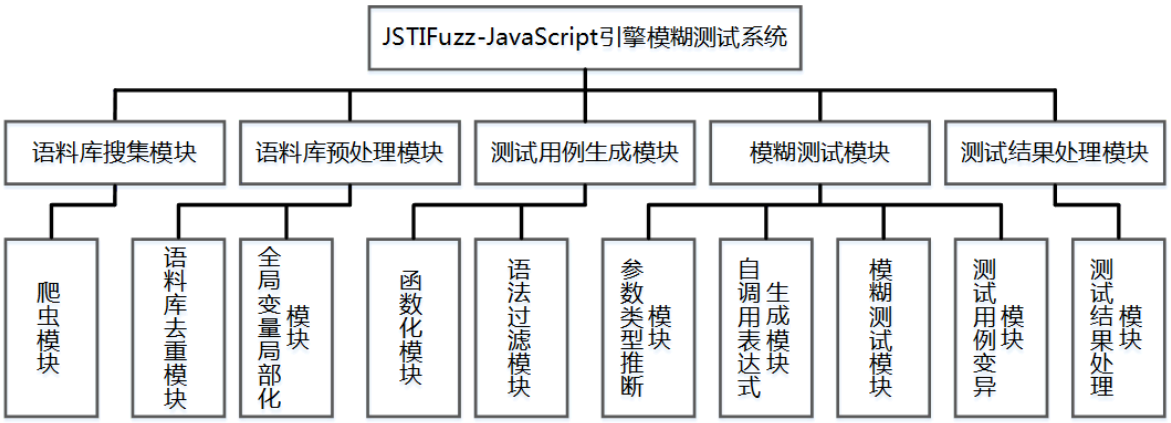

图 18 JSTIFuzz 原型系统模块结构

其中语料库搜集模块实际上由一个网页爬虫模块构成，其主要的工作职责为从 Github 等线上代码托管网站中爬取 JavaScript 代码，构成原始的语料库。测试结果处理模块的主要作用是在发现被测试 JavaScript 引擎发生崩溃时，保存测试用例、发生崩溃的引擎和系统环境参数，并过滤重复的测试结果。

系统中负责 JavaScript 引擎模糊测试的核心模块详细介绍如下。

4.1.1 语料库预处理模块

语料库预处理模块包含语料库去重模块和全局变量局部化模块两个子模块，其整体的作用是对语料库进行基本的处理，使其格式统一，内容纯净且尽可能地正确，以符合下一阶段所需的输入的标准。其中语料库去重模块用于过滤掉重复的 JavaScript 代码，全局变量局部化模块用于提前消除后续步骤中可能出现的“变量未定义问题”。其具体的工作流程如图 19 所示，详细描述如下：

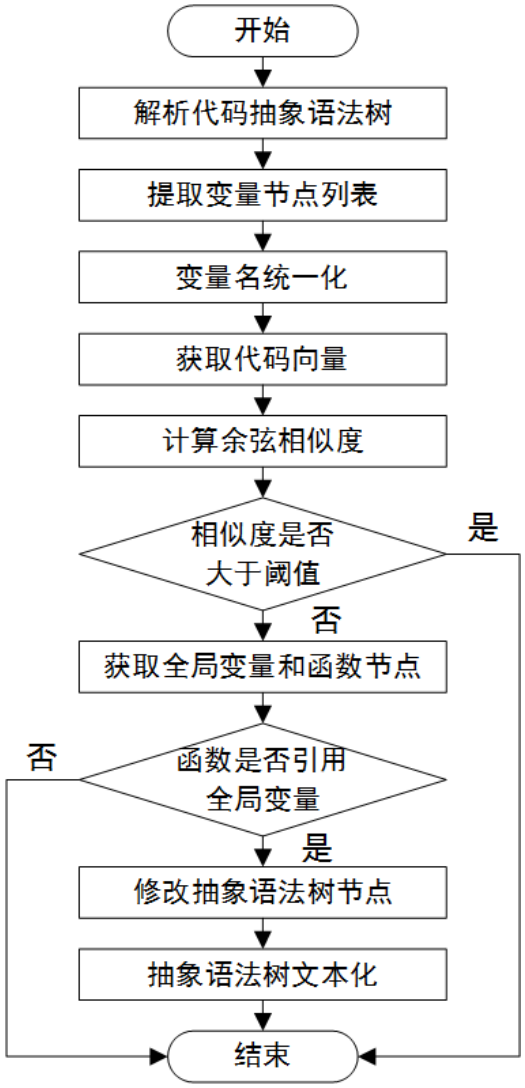

图 19 语料库预处理模块流程图

（1）解析语料库中的代码得到抽象语法树。遍历抽象语法树得到树中所有变量节点组成的列表。

（2）遍历变量节点列表，按照普通变量名为小写字母，函数名为大写字母的规则修改所有的变量名，进行统一化。

（3）将代码向量化，得到对应的 n 维向量。用当前的代码向量与代码向量库中的所有向量分别计算余弦相似度，如果某次的计算结果大于阈值，则判定当前代码为重复语料，将其删除。

（4）从抽象语法树中获取全部的全局变量节点组成的列表和全部函数节点组成的列表。

（5）遍历函数节点列表，如果其中的函数节点引用了某个全局变量节点，则修改抽象语法树，将该全局变量节点复制进函数节点，将其变为函数的局部变量。

（6）将抽象语法树文本化，得到预处理完成的语料。

4.1.2 测试用例生成模块

测试用例生成模块包含函数化模块和语法过滤模块两个子模块。其中函数化模块将经过了预处理的语料库中的代码切分成函数，用于后续步骤中的处理和测试。语法过滤模块将包含语法错误的函数从测试用例集合中过滤出去。其具体的工作流程如图 20 所示，详细描述如下：

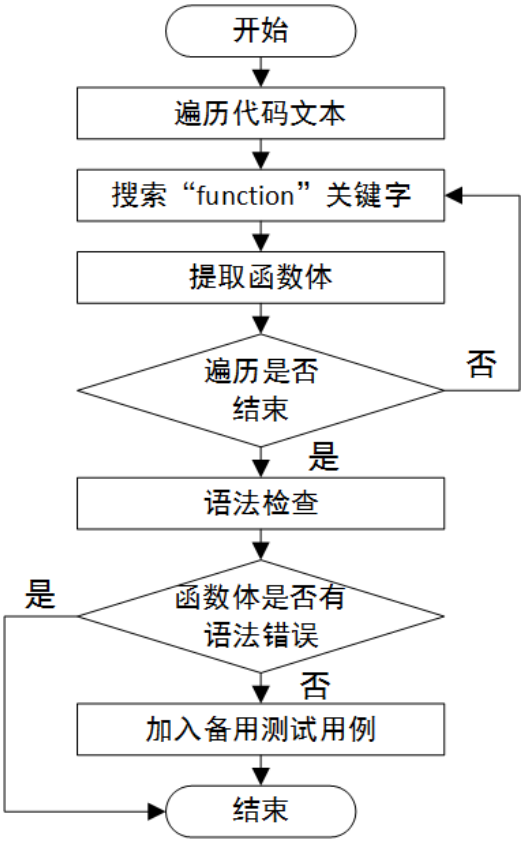

图 20 测试用例生成模块流程图

（1）遍历预处理后的语料库中的代码文本，搜索“function”关键字，若搜索到结果，提取对应函数的函数体。

（2）继续遍历，直到提取到了该份预备测试用例中的所有函数体。

（3）对提取到的函数体进行语法检查，若函数体中有语法错误，则放弃该函数体，否则将其加入预备测试用例库中。

4.1.3 模糊测试模块

模糊测试是本文的最终目标，故模糊测试模块是 JSTIFuzz 原型系统中最重要的模块。其包含参数类型推断模块、自调用表达式生成模块、模糊测试模块和测试用例变异模块等 4 个子模块。其具体的工作流程如图 21 所示，详细描述如下：

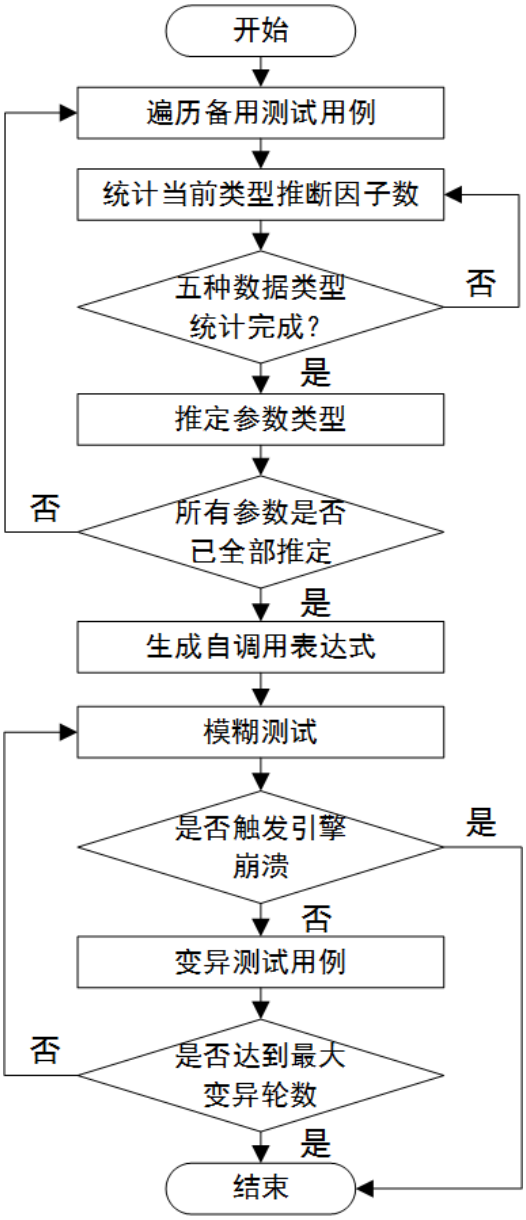

图 21 模糊测试模块流程图

（1）遍历预备测试用例，针对该函数的每一个参数，统计函数体中该参数对五种数据类型的推断因子，推定统计得分最高的数据类型为该参数的数据类型。直到完成对每一个参数的类型推断。

（2）根据参数类型推断的结果，首先生成参数定义表达式，即生成相应数据类型的实际参数。然后根据参数名列表生成自调用表达式，使用上述实际参数调用预备测试用例（函数）。即得到了最终的测试用例。

（3）使用测试用例对 JavaScript 引擎执行模糊测试。如果引擎发生了崩溃，则结束当前测试用例的执行流程。否则对测试用例进行变异，然后继续进行模糊测试。

（4）持续执行第（3）步，直到 JavaScript 引擎发生了崩溃或变异次数达到了预

设的最大轮数。

4.2 关键算法设计

4.2.1 全局变量局部化算法

全局变量局部化算法整体上先将代码解析成抽象语法树，然后在抽象语法树上识别全局变量节点，最后将与每个函数节点有关的全局变量节点复制并插入函数节点的函数体中最前面的位置。其详细的算法如算法 1 所示。

算法 1　全局变量局部化算法

输入：code_string，语料库中 JavaScript 代码的字符串

输出：完成操作的 JavaScript 代码的字符串

```
1:  Dim ast = parseScript(code_string)
2:  Dim global_variables = getGlobalVariables(ast)
3:  Dim functions = getFunctions(ast)
4:  For Each func In functions
5:      Dim undefined_ids = getUndefinedIds(func)
6:      Dim needed_globale_variables = []
7:      For Each undefined_id In undefined_ids
8:          For Each global_variable In global_variables
9:              If undefined_id == global_variable.name Then
10:                 needed_globale_variables.add(global_variable)
11:             End If
12:         Next
13:     Next
14:     func.body.insert(0, needed_globale_variables)
15: Next
16: Return func.toString()
```

算法的解析如下：

1-3 行：定义变量 ast 表示代码经过解析后得到的抽象语法树对象。定义变量 global_variables 表示从抽象语法树中搜索到的全部全局变量的数组。定义变量

functions 表示从抽象语法树中搜索到的全部函数的数组。

4-6 行：进入循环，对 functions 中每个函数 func 进行处理。定义变量 undefined_ids 表示当前 func 中依赖的全局变量名。定义变量 needed_globale_variables 作为当前 func 所需的全部全局变量数组。

7-13 行：进入两层嵌套子循环，对当前 func 所需的全局变量名和 global_variables 中的全局变量的名字进行比较，如果有一致的，就将该全局变量节点添加到数组 needed_globale_variables 中。

14 行：将 needed_globale_variables 数组中的所有节点按顺序插入到当前 func 的最前面，使这些节点成为当前 func 内部的局部变量节点。

4.2.2 函数化算法

通过静态文本匹配的方法，从整段的 JavaScript 代码中抽取出函数，其主要的算法逻辑即先找到“function”关键字，然后根据左大括号’{’和右大括号’}’在数量上的匹配来确定一个函数是否截取完整。其主要的算法如算法 2 所示。

算法 2　函数化算法

输入：code_string，语料库中 JavaScript 代码的字符串

输出：拆分得到的函数体集合

```
1:  Dim index = 0, function_index = 0, functions = {}
2:  Do While index < code_string.length And function_index > -1
3:      function_index = code_string.find('function', index)
4:      If function_index > -1 Then
5:          Do While function_index < code_string.length And code_string[function_index] != '{'
6:              function_body += code_string[function_index++]
7:          Loop
8:          Dim function_body += "{"
9:          Dim open_brace = 1, close_brace = 0
10:         Do While ++function_index < code_string.length And open_brace != close_brace
11:             Dim current_character = code_string[function_index]
12:             function_body += current_character
13:             If current_character == '{' Then
```

```
14:                 open_brace += 1
15:             End If
16:             If current_character == '}' Then
17:                 close_brace += 1
18:             End If
19:             function_index += 1
20:         Loop
21:         functions.add(function_body)
22:         index = function_index + 1
23:     End If
24: Loop
25: Return functions
```

算法的解析如下：

1 行：定义遍历索引变量 index，初值为 0。定义函数索引变量 function_index，初值为 0。定义函数体集合 functions。

2-4 行：进入子循环，对代码字符串 code_string 进行遍历。优先搜索 code_string 中的"function"关键字。

5-7 行：跳过函数名等字符，直接找到以左大括号'{'起始的函数体。

10-20 行：进入子循环，以左大括号'{'和右大括号'}'的数量是否匹配为判断依据，抽取函数体。循环直到上述两种大括号的数量一致时结束，函数体变量 function_body 中即本次抽取出的函数体。

时间复杂度分析：设代码中的字符数为 n，由于上述算法只会固定地对所有字符进行一趟遍历，所以其时间复杂度为O(n).

4.2.3 参数类型推断算法

在章节 3.4.1 中曾介绍过，本文针对 Array、Boolean、Number、Function 和 String 等 5 中 JavaScript 中的数据类型分别设计了一些前缀、后缀和环绕推断因子，以此来推断一个函数的参数所需的数据类型是什么。其具体的算法如下：

算法 3　参数类型推断算法

输入：param_name，待推定类型的参数名；callable，待推定参数所在的函数体

输出：参数类型推断结果集合

```
1:  Dim factors = [array_factors, boolean_factors, number_factors, string_factors, function_factors]
2:  Dim counter = [0, 0, 0, 0, 0]
3:  Dim types = ["array", "boolean", "number", "string", "function"]
4:  For Each factor In factors
5:      counter[i] += infer_left(param_name, callable, factor[0])
6:      counter[i] += infer_right(param_name, callable, factor[1])
7:      counter[i] += infer_around(param_name, callable, factor[2])
8:  Next
9:  Dim max_counter = max(counter)
10: If max_counter > 0 Then
11:     Dim result = []
12:         For i = 0 To 4
13:             If counter[i] == max_counter Then
14:                 result.append(types[i])
15:             End If
16:         Next
17: Else
18:     result.append("none")
19: End If
20: Return result
```

算法的解析如下：

1-3 行：定义推断因子二维数组 factors，包含每种数据类型的前缀、后缀和环绕推断因子，详细的推断因子设计见章节 3.4.3。定义数据类型计数器数组 counter，表示当前参数在每种类型上统计到的推断因子数，也称作得分。定义推断结果数组 types，表示包含 5 种数据类型文本形式的数组。

4-8 行：进入循环，针对每种数据类型，统计待推断参数 param_name 在函数体 callable 中，与前缀、后缀和环绕推断因子的组合出现的次数，即得分。如在 callable 中出现"param_name.join"这个组合一次，则此时 counter 中代表数组 Array 类型的 0

号位置数值加 1。

9-19 行：将计数器中得分最高的一个和或多个所对应的数据类型添加到结果集合 result 中，作为推断的结果，如 counter = [3, 0, 1, 3, 0]，则 result = ["array", "string"]，即待推断参数的数据类型可能是 array 或 string。

时间复杂度分析：设函数体的字符数为 n，5 种数据类型包含的推断因子数分别为 a, b, c, d, e，由于对每种数据类型的所有推断因子都遍历一趟函数体 callable，因此时间复杂度为$O((a+b+c+d+e)n)$. 又因为 a, b, c, d, e 都是已知的常数，因此该算法的时间复杂度化简为$O(n)$.

4.2.4 测试用例变异算法

在章节 3.4.4 中本文介绍过，有了参数类型推断的结果，即可以精确地引导测试用例的变异，即仅需要对最终测试用例中的参数定义表达式进行变异。由此，通过参数数值范围的变化，触发测试用例中更多的边界条件，提升变异器变异测试用例的效率。其具体的算法如下：

算法 4　测试用例变异算法

输入：function_body，待变异测试用例的函数体；params，存放参数名、参数类型和参数旧值的三元组列表；self_calling，测试用例的自调用表达式

输出：经过变异后的新测试用例字符串

```
1:  Dim param_def AS String()
2:  For i = 1 To params.length - 1
3:      Dim new_value
4:      Select Case params[i].type
5:          Case "Array"
6:              new_value = mutateArray(params[i].value)
7:          Case "String"
8:              new_value = mutateString(params[i].value)
9:          Case "Boolean"
10:             new_value = mutateBoolean(params[i].value)
11:         Case "Number"
12:             new_value = mutateNumber(params[i].value)
```

```
13:             Case "Function"
14:                 new_value = mutateFunction(params[i].value)
15:         End Select
16:         param_def += ("var " +params[i].name + " = " + new_value)
17:     Next
18:     Return function_body + param_def + self_calling
```

算法的解析如下：

1 行：设置参数定义表达式变量 param_def，表示变异后的新参数定义表达式。

2-3 行：进入循环，按顺序遍历三元组列表 params 中的每个三元组 params[i]。三元组形如(name, type, value)，即表示当前参数名、类型和旧值。定义变量 new_value 表示参数的新值。

4-15 行：进入条件选择，根据三元组中的参数类型 params[i].type 选择相应的函数对旧值进行变异。

16 行：根据参数的新值拼接新的参数定义表达式。

17 行：将函数体、新的参数定义表达式和自调用表达式拼接起来，产生经过变异后的新测试用例。

4.3 系统界面设计

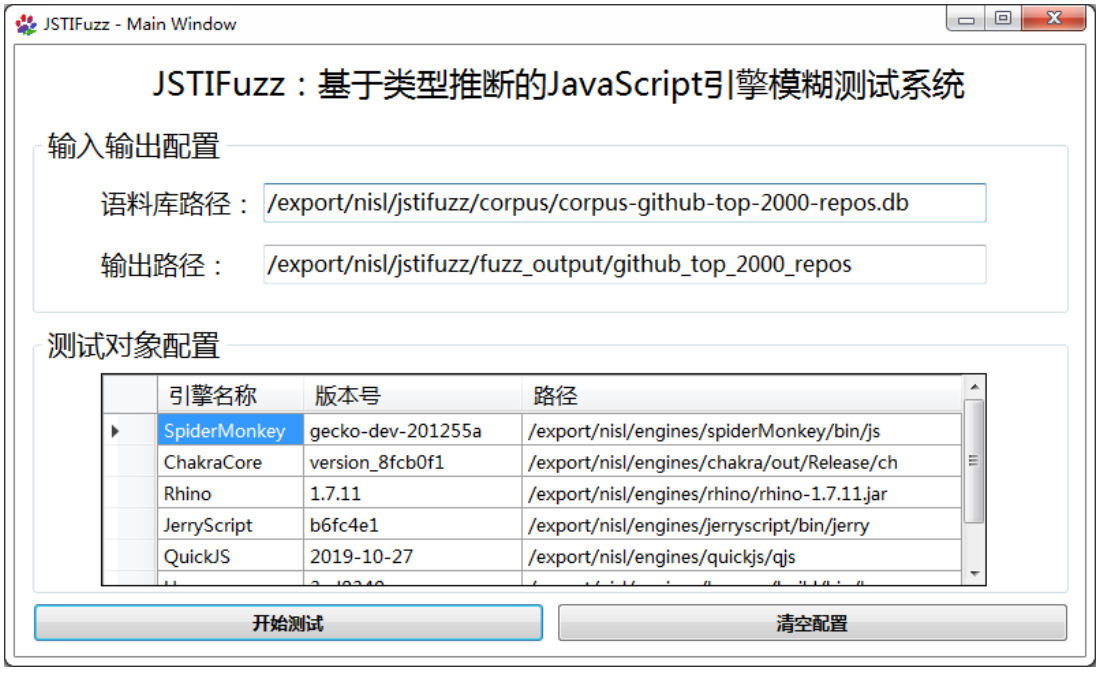

图 22 JSTIFuzz 系统主界面图

打开 JSTIFuzz 原型系统，其主界面上包含输入输出配置区域、测试对象配置区域两个部分。如图 22 所示，在输入输出配置区域中设置好了语料库的路径、系统输出路径，并在测试对象配置区域中设置好了 JavaScript 引擎的信息之后，点击界面左下角的“开始测试”按钮即可开始对这些引擎的模糊测试。

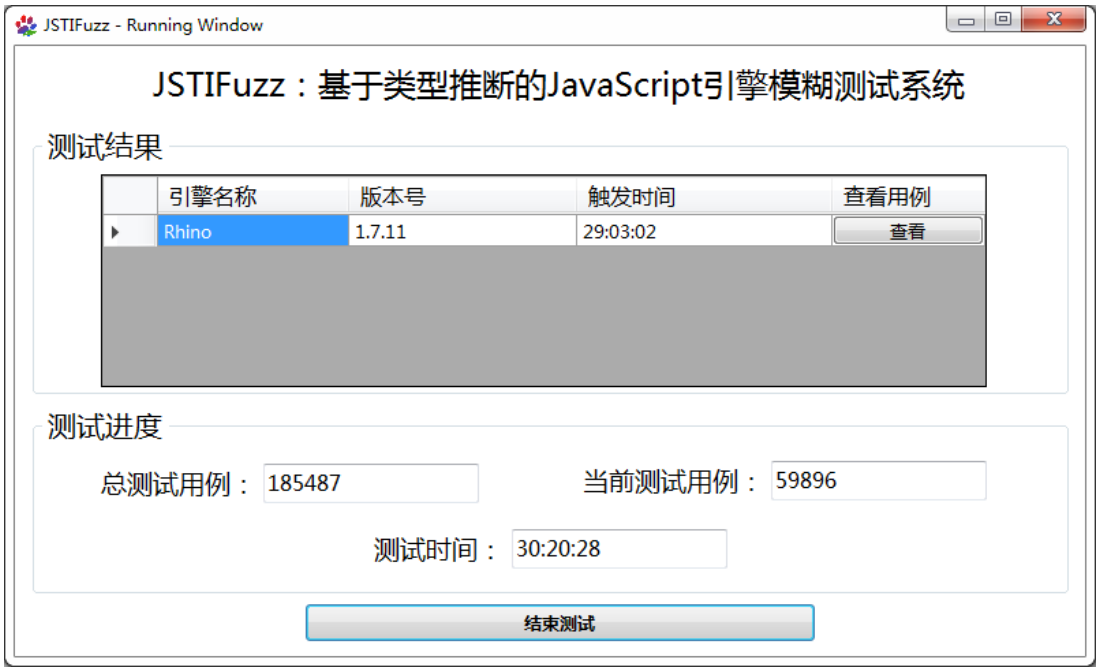

图 23 JSTIFuzz 系统测试执行界面图

点击主界面中的“开始测试”按钮后，系统跳转到测试执行界面，如图 23 所示。在该界面的上半部分，可以实时地看到当前的测试结果，点击列表中的“查看”按钮可以看到触发该缺陷的测试用例。在该界面的下半部分，可以看到当前测试用例执行进度和测试总时长。点击最下方的“结束测试”按钮即可终止测试。

4.4 本章小结

在本章中，本文对 JSTIFuzz 原型系统的设计与实现进行了详细的介绍。首先对语料库预处理模块、测试用例生成模块和模糊测试模块等三个关键模块的工作流程进行了详细的介绍。然后对全局变量局部化算法、函数化算法、参数类型推断算法和测试用例变异算法等四个关键算法进行了详尽的说明。最后，本文还对系统的界面进行了展示，并对界面上分块的功能进行了简单的介绍。

第五章 系统实验评估与分析

在原型系统 JSTIFuzz 实现的基础上，本章将使用收集到的数万份原始语料，与知名的模糊测试工具进行测试用例变异效率的对比，并对知名的 JavaScript 引擎展开真实环境下的模糊测试，以验证本文所提出的方法以及所实现的 JSTIFuzz 原型系统的有效性。

5.1 实验设计

5.1.1 实验环境和实验步骤

本文的实验环境如下：

操作系统：Linux Ubuntu 16.04.6 LTS；处理器：Intel Core i7-7820x，3.6GHz；内存：64GB。软件环境：Python 3.7.1。

具体的实验步骤设计如下：

（1）测试用例产生及变异：使用从 Github 上收集到的数万份原始语料生成测试用例，然后使用这些测试用例对选定的 JavaScript 引擎进行模糊测试，其间配合本文所设计的变异器对测试用例进行变异。

（2）参数类型推断效果评估：在参数类型推断阶段，人工判断对函数参数进行类型推断的精确程度，并与随机传参（即不进行类型推断而随机设定参数类型）的结果进行比较。从参数类型推断的精确性方面验证本文方法的有效性。

（3）代码覆盖率提升效果评估：在测试用例产生和变异的各个步骤中，对测试用例的代码覆盖率进行统计，并计算出每个步骤执行前和执行后的覆盖率差值。同时，统计 JavaScript 引擎在执行每个步骤的处理产出（各阶段产生的测试用例）时，其引擎自身代码的覆盖率的变化情况。从代码覆盖率方面验证本文方法的有效性。

（4）模糊测试效果评估：首先使用 JSTIFuzz 原型系统与选定的知名模糊测试工具，在相同的测试集进行测试，统计并比较各工具复现测试集中缺陷的情况。最后使用 JSTIFuzz 原型系统对选定的 JavaScript 引擎的最新版本进行模糊测试，统计其发现的各引擎的缺陷。从真实环境下模糊测试的效果方面验证本文方法的有效性。

5.1.2 测试对象和对比工具介绍

在实验中，本文选择了 6 个 JavaScript 引擎进行测试，其分别为用于浏览器中的

JavaScript 引擎两种，用于 JDK 内嵌的、物联网设备中的、嵌入式设备中的、可移动设备中的 JavaScript 引擎各一种。详细的说明如表 10 所示。

表 10 测试对象说明

JavaScript 引擎	类型	说明
SpiderMonkey[39]	浏览器 JavaScript 引擎	知名浏览器 FireFox 中的 JavaScript 引擎，由 C 语言实现。
ChakraCore[40]	浏览器 JavaScript 引擎	知名浏览器 Windows Edge 中的 JavaScript 引擎，由 C 语言实现。
Rhino[41]	JDK 内嵌 JavaScript 引擎	SpiderMonkey 引擎的 Java 实现版本。曾于 Java Development Kit 中内嵌的 JavaScript 引擎，用于执行 Java 代码中嵌套的 JavaScript 脚本，现被 Nashorn 引擎替代。
JerryScript[42]	物联网设备 JavaScript 引擎	由知名的三星公司开发的 JavaScript 引擎。用于物联网设备中，特点是其较为轻量级，能够运行在物联网设备中非常有限的内存中。
QuickJS[43]	嵌入式设备 JavaScript 引擎	轻量级的嵌入式 JavaScript 引擎，支持 ES2020。可以将 JavaScript 源码编译成其他的可执行文件。
Hermes[44]	可移动设备 JavaScript 引擎	由知名的 Facebook 公司开发的 JavaScript 引擎。用于在可移动设备中提升应用软件的执行性能。

同时，为了和现有的工作进行对比，本文选择了 AFL 和 CodeAlchemist 这两个模糊测试工具进行对比实验，其中 AFL-fast 是 AFL 的效率优化版本。包含本文所实现的 JSTIFuzz 在内，4 个模糊测试工具的说明如表 11 所示。

表 11 进行对比的模糊测试工具说明

模糊测试工具	测试用例产生方式	说明
JSTIFuzz	对原始语料以函数为单位进行拆分，然后对函数的参数进行类型推断，最后根据推断结果进行传参产生测试用例，并对测试用例进行精准的变异。	根据本文所提出的方法实现的模糊测试工具。
CodeAlchemist[45]	对原始语料以块为单位进行拆分，对拆分的代码块按类型进行装配，以产生新的测试用例[46]。	当前较新的模糊测试工具
AFL	对原始语料文本进行逐位的随机变异，具体的变异方法如表 4 所示。	经典的老牌模糊测试工具
AFL-fast[47]	对原始语料文本进行逐位的随机变异，具体的变异方法如表 4 所示。	AFL 的效率优化版本

5.2 参数类型推断效果评估

在本小节中，笔者通过人工分析的方法对本文提出的参数类型推断方法的有效性进行评估。实验分为对照组和实验组，两组的输入数据都相同，即 1 万份预备测试用例。对照组中，本文对 1 万份备用测试用例进行随机传参，即等概率地将函数的每个参数的数据类型设定为 Array、Boolean、Number、Function 和 String 等 5 种数据类型中的一种。而实验组使用本文所提出的参数类型推断方法，对函数的每个参数进行类型推断。根据人工分析，对两组实验的产出的类型精确率进行统计。

表 12 参数类型推断效果评估实验结果

分组	类型精确率				
	Array 类型	**Boolean 类型**	**Number 类型**	**Function 类型**	**String 类型**
对照组	17.86%	7.69%	14.71%	9.35%	18.42%
实验组	96.30%	83.33%	92.41%	98.33%	97.30%

经过人工分析和统计，1 万份输入数据中，共有参数 16074 个，其中可以进行类型推断的参数有 5903 个。即每个函数平均约有 1.6 个参数，其中约有 0.6 个可以进行参数类型推断。

实验的结果如表 12 所示。由于对照组使用的是随机策略，不具有逻辑性，因此类型的精确率普遍偏低，由 7.69%到 18.42%. 使用了本文所提出的参数类型推断方法，实验组的类型精确率最高达到了 98.33%. 对两组实验的类型精确率进行比较，其中 Array 类型，实验组是对照组的 5.4 倍；Boolean 类型，10.8 倍；Number 类型，6.3 倍；Function 类型，10.5 倍；最后，对于 String 类型，实验组是对照组的 5.3 倍。

在两组实验中，Boolean 类型的精确率较其他 4 中数据类型低一些。对照组的结果受限于随机性，而在类型推断中，从表 6 可以看出，由于 Boolean 类型只有“真”或“假”两种值，因此其类型推断因子也较其他 4 种数据类型更少。而当类型推断因子越多时，对该数据类型的类型推断也越精确。

对于 Function 类型，在对照组中其类型准确率为 9.35%，偏低。原因在于，JavaScript 语言中使用函数来实现其他编程语言中的“类”的功能。对于面向对象的语言而言，通过实例化类来操作对象的情况比对 Array、Boolean、Number、Function 和 String 等 5 种基础数据类型的使用更加普遍。在实验的统计中，Function 类型的可推断参数其数量也是其他 4 种数据类型的数倍，因此在对照组中其类型准确率偏低。从表 6 可以看出，Function 数据类型的类型推断因子较少。而不同于其他 4 种数据类

型，对 Function 数据类型的参数的操作具有较大的局限性，一般即调用它们。而调用函数的方式有通过函数名直接调用和通过其“.call”函数进行调用两种。在实验分析中发现，其“(”和“.call”这两种类型推断因子共提供了接近全部的类型推断贡献。

5.3 代码覆盖率提升效果评估

代码覆盖率用于描述程序源代码被执行的程度。覆盖率高的程序在测试期间执行了更多的源代码，与覆盖率低的程序相比，它包含未检测到的软件错误的几率更低[48]。它是衡量测试用例质量的重要指标。

在本小节的所有评估实验中，本文所使用的实验输入为从 Github 网站上收集到的 1 万份原始语料，即 1 万份包含各种功能和操作的 JavaScript 代码。

5.3.1 全局变量局部化效果评估

由于本文的方法将函数作为测试用例的基本单元，所以在后面的函数化操作时，会抛弃原始语料中函数以外的内容。因此就可能导致函数中引用全局变量时引擎报出“变量未定义”的问题。本文曾在 3.2.2 小节中介绍过，全局变量局部化操作可以提前消除上述问题出现的可能。

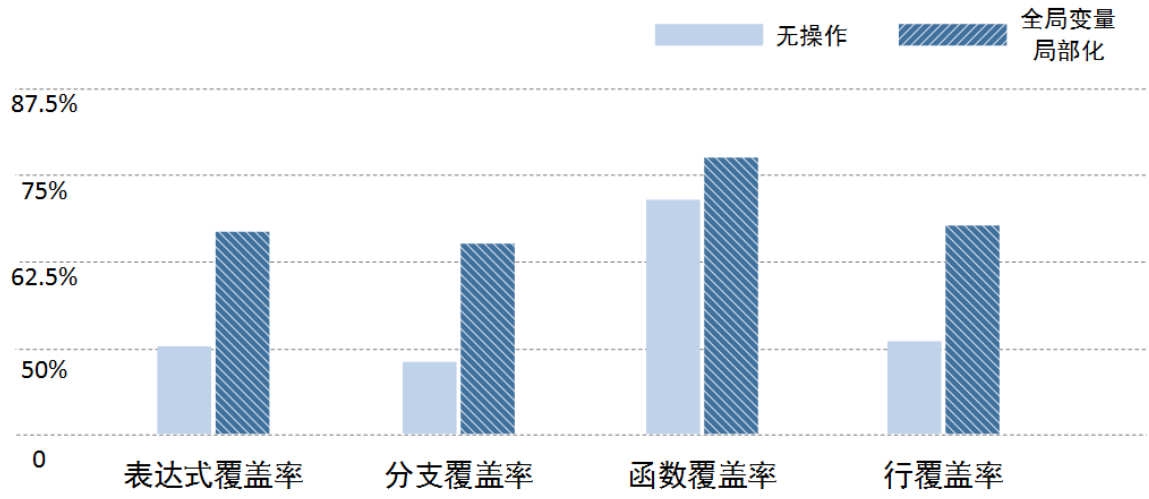

图 24 全局变量局部化操作前后的代码覆盖率对比

图 24 描述了对本组实验设置对照组和实验组，首先对实验组的数据做全局变量局部化操作，而对照组的数据不做任何操作。然后分别对两组数据进行函数化操作，进而分别对两组在表达式覆盖率、分支覆盖率、函数覆盖率和行覆盖率这 4 个维度上进行了统计，最后求得均值如图所示。

从图中可见，在先做了全局变量局部化操作后，再执行函数化操作，得到的测试

用例其表达式覆盖率、分支覆盖率、函数覆盖率和行覆盖率都有一定的提升，其中表达式覆盖率提升了 15.25%，分支覆盖率提升了 17.08%，函数覆盖率提升了 5.79%，行覆盖率提升了 15.17%。消除了变量未定义的问题，JavaScript 引擎在执行代码时就不会过早地退出，进而可以执行到测试用例中更多的代码，因此在 4 种代码覆盖率统计维度上都有提升。函数覆盖率提升效果相对较低的原因在于，JavaScript 语言虽然允许函数的嵌套，但这种现象在实际的代码中并不常见。因此，引擎可以多执行到的这部分代码中，函数只占很少的一部分。

5.3.2 类型推断传参效果评估

在 3.4 和 3.5.1 小节中曾介绍过，本文所提出的方法首先对函数中的参数进行类型推断，在得知了每个参数的数据类型后，就可以根据相应的类型更加精准地传递函数所需的参数，进而通过这些参数去调用函数，提升函数体中的代码覆盖率。

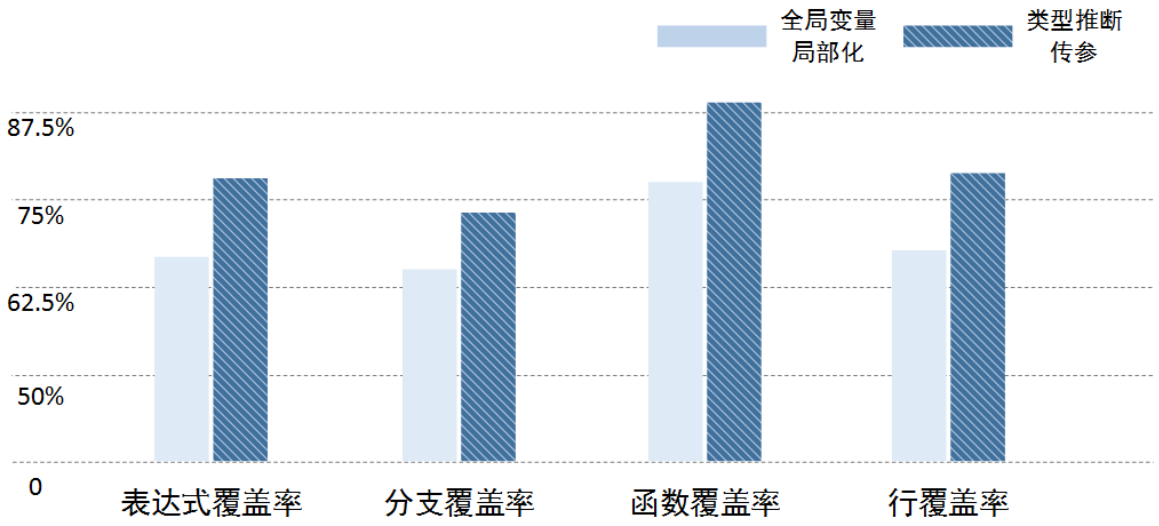

图 25 类型推断传参操作前后的代码覆盖率对比

图 25 描述了对本组实验设置对照组和实验组，实验输入为 5.3.1 小节中实验组的结果，即全局变量局部化后的测试用例。首先对实验组的数据做类型推断传参操作，而对照组的数据不做任何操作。然后分别对两组在表达式覆盖率、分支覆盖率、函数覆盖率和行覆盖率这 4 个维度上进行了统计，最后求得均值如图所示。

从图中可见，执行了本文提出的类型推断传参操作之后，测试用例的表达式覆盖率、分支覆盖率、函数覆盖率和行覆盖率都有进一步的提升。其中表达式覆盖率提升了 9.86%，分支覆盖率提升了 7.08%，函数覆盖率提升了 10.35%，行覆盖率提升了 9.95%。在不传入参数或随机传入参数时，引擎对于这些测试函数的执行将终止于首次调用形参的时候，因为此时参数为空或者其类型和所需的类型不匹配。类型推断子

模块和自调用表达式生成子模块配合作用为函数生成其所需类型的参数，进一步提升了测试用例的代码覆盖率。

5.3.3 测试用例变异效果评估

表 13 测试用例变异 5 次的覆盖率变化

度量标准	测试用例变异的覆盖率变化			
	表达式覆盖率	分支覆盖率	函数覆盖率	行覆盖率
中位数	78.98% (+1.37%)	74.77% (+3.14%)	88.12% (+0.92%)	78.63% (+1.34%)
最大值	81.72% (+5.11%)	77.05% (+5.42%)	91.08% (+3.88%)	82.36% (+5.07%)
最小值	75.71% (-0.90%)	70.15% (-1.48%)	86.41% (-0.79%)	76.44%(-0.85%)

参考已有的对于模糊测试的指导[49]，由于模糊测试中的各种操作具有随机性，因此实验需要做 5 组以上，然后根据各种度量标准来进行效果的评判。表 13 描述了对 5.3.2 中实验组的结果（即经过类型推断传参之后的测试用例），使用本文所设计的测试用例变异模块，进行了 5 组测试用例变异实验的部分结果。由于单纯的平均值不能很好地代表数据的整体情况，所以选择了中位数、最大值和最小值三种度量标准来描述这 5 组实验的结果。

从表 13 中可以看出，由于变异结果具有随机性，在 5 组实验结果中，测试用例的表达式覆盖率有-0.90% ~ 5.11%的变化范围，其中位数为 1.37%；分支覆盖率有范围在-1.48% ~ 5.43%的变化，其中位数为 3.14%；函数覆盖率有-0.79% ~ 3.88%的变化范围，其中位数为 0.92%；最后，行覆盖率有范围在-0.85% ~ 5.07%的变化，其中位数为 1.34%。测试用例的变异会导致函数的参数发生随机的变化，由上述实验结果中亦可看出，覆盖率有可能会升高，也有可能会降低。但表达式覆盖率、分支覆盖率、函数覆盖率和行覆盖率的中位数都大于 0，因此变异模块对测试用例所执行的变异操作整体上有正向的作用和效果。

5.3.4 测试用例覆盖率提升效果总结

在前面三个小节中，本文分别介绍了 JSTIFuzz 原型系统中几个子模块在产生和处理测试用例的过程中，对测试用例代码覆盖率的提升效果。在本小节中将各个子模块的代码覆盖率贡献进行了总结。

如图 26 所示，在先后经过了全局变量局部化、类型推断传参等两项操作后，测试用例的表达式覆盖率有了 25.11%的提升，分支覆盖率有了 24.16%的提升，函数覆盖率有了 16.14%的提升，最后行覆盖率有了 25.12%的提升。最终，由于测试用例的

变异存在较强的随机性，因此在最好的情况下，表达式覆盖率共有 30.33%的提升，由未做处理时的 51.50%提升到 81.83%；分支覆盖率共有 29.58%的提升；函数覆盖率共有 20.02%的提升；行覆盖率共有了 30.19%的提升。

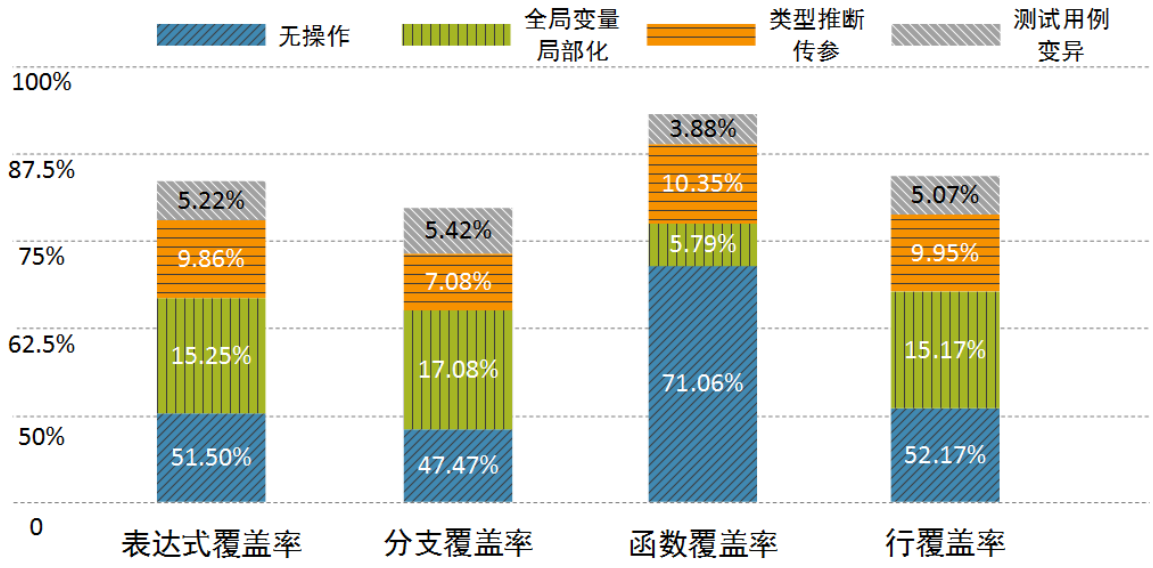

图 26 JSTIFuzz 原型系统对测试用例覆盖率提升的整体效果

5.3.5 JavaScript 引擎代码覆盖率提升效果评估

在做模糊测试时，本文所测试的对象是 JavaScript 引擎，最终的目标也是用产生的测试用例尽可能多得覆盖到 JavaScript 引擎中的代码。在前面的小节中，本文评估了 JSTIFuzz 原型系统对测试用例覆盖率提升的效果，在本小节中，将评估原型系统所产生的测试用例对 JavaScript 引擎本身的代码覆盖率的提升效果。

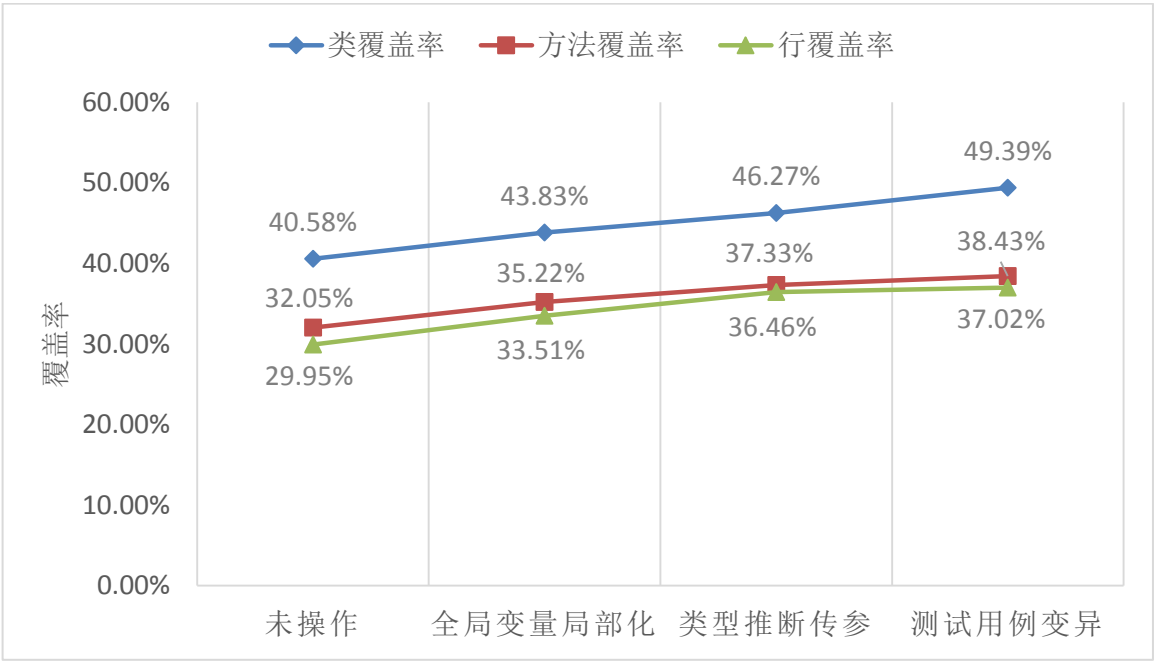

图 27 JavaScript 引擎代码覆盖率提升效果

图 27 描述了本文使用开源 JavaScript 引擎 Rhino 进行的覆盖率统计结果。由表 10 中可知，Rhino 是 SpiderMonkey 引擎的 Java 版本。曾于 JDK[50]中内嵌的 JavaScript 引擎，用于执行 Java 代码中嵌套的 JavaScript 脚本。笔者获取了 Rhino 的源代码，用其执行在 JSTIFuzz 原型系统运行的各个过程中产生和处理过的测试用例，并通过集成开发环境 IDEA[51]统计 Rhino 源代码的覆盖率，共分为类覆盖率、方法覆盖率和行覆盖率三个统计维度。

如图所示，先后经过全局变量局部化、类型推断传参和测试用例变异等三个操作，Rhino 引擎的类覆盖率由 40.58%提升到了 49.39%，提升了 8.81%；方法覆盖率提升了 6.38%；最后，行覆盖率提升了 7.07%。类覆盖率比方法覆盖率和行覆盖率高的原因在于，一个类通常包含许多方法，进而包含许多分之，类中即使只有一行或者一个方法被覆盖到，都算作该类被覆盖到。假设 JavaScript 引擎的源码中包含 x 个类、y 个方法、z 个行，显然可得 x < y < z. 若此时，由于测试用例的变化使得引擎中多出一个类的一个方法中的一行被覆盖到，那么覆盖率的提升情况$1/x > 1/y > 1/z$是显而易见的。

由此可以说明，JavaScript 引擎的代码覆盖率与测试用例的代码覆盖率呈正相关关系，即 JSTIFuzz 原型系统所产生的测试用例可以有效得提升 JavaScript 引擎的代码覆盖率。

5.4 模糊测试效果评估

在 5.3 小节中，本文从代码覆盖率的维度上对 JSTIFuzz 原型系统进行了多方面的评估。而一个模糊测试工具最终的目标仍是高效得触发被测试的 JavaScript 引擎中的缺陷。在本小节中，本文使用 JSTIFuzz 原型系统在测试集和真实环境下分别对 5.1.2 小节中介绍的测试对象展开模糊测试，并和几个知名的模糊测试工具进行比较，以评估本文所设计的方法和所实现的 JSTIFuzz 原型系统的模糊测试效果。

5.4.1 测试集上的缺陷复现效果评估

表 14 各模糊测试工具在测试集上的缺陷复现情况

	JSTIFuzz	CodeAlchemist	AFL	AFL-fast
ChakraCore	5	3	0	0
JerryScript	8	-	0	0
Rhino	7	-	-	-

为了验证 JSTIFuzz 在实际模糊测试中的表现，本文首先在收集到的测试集上进行了缺陷的复现实验。其中 Chakra 的版本为 v1.7.2，已知崩溃缺陷 9 个；JerryScript 的版本为 2.0，已知崩溃缺陷 17 个；Rhino 的版本为 1.7.7.2，已知崩溃缺陷 13 个。

表 14 描述的是本文使用 4 款模糊测试工具在上述测试集上进行的缺陷复现实验结果。实验中对每个 JavaScript 引擎使用 5 个 CPU 核心进行 100 个小时的测试。由于 CodeAlchemist 不支持 JerryScript 和 Rhino，AFL 及其快速版本 AFL-fast 不支持可执行文件为 Jar 文件的 Rhino，所以表中没有对应的测试结果。从表中可以看出，JSTIFuzz 在 Chakra 的测试集上复现的崩溃缺陷比 CodeAlchemist 多 2 个，其中 CodeAlchemist 触发的 2 个缺陷中，1 个是其独占的外，剩下的 2 个和 JSTIFuzz 重叠，且有 3 个是 JSTIFuzz 独占的缺陷。最后 JSFIFuzz 在 JerryScript 和 Rhino 的测试集上各复现了 8 个和 7 个。而 AFL 和 AFL-fast 都没有触发任何崩溃缺陷。

由上述的对比实验结果可以得出，JSTIFuzz 支持更多的 JavaScript 引擎，且比当前较新的和经典的模糊测试工具都具有更好的测试效果。

5.4.2 真实环境下的模糊测试效果评估

检验一个模糊测试工具是否有效，最终都需要使用其在真实环境下进行测试的实践。而所谓真实环境，即区别于 5.4.1 小节复现实验中人为设置的测试集环境，使用最新版本的 JavaScript 引擎进行测试。

表 15 真实环境下的模糊测试结果

序号	JavaScript 引擎	描述	状态
1	Rhino-1.7.11	由引擎内部逻辑异常造成的栈溢出	已提交[52]
2	JerryScript- 40f7b1c	由引擎内部逻辑异常造成的引用计数超限	已提交[53]
3	JerryScript- 84a56ef	由引擎内部内存管理错误造成的段错误	已确认[54]
4	QuickJS-2019-10-27	由引擎内部递归层数过多导致的段错误	已修复
5	QuickJS-2019-10-27	由引擎内部递归层数过多导致的段错误	已修复
6	Hermes-3ed8340	由引擎内部内存管理错误造成的段错误	已确认[55]

表 15 描述了本文使用 JSTIFuzz 原型系统在真实环境下对 5.1.2 小节中介绍的测试对象进行的模糊测试实验结果。在该实验中，笔者对每个 JavaScript 引擎使用 5 个 CPU 核心进行了 100 个小时的测试。

从表中可以得出，在 100 个小时的测试中，本文在 4 个 JavaScript 引擎的 5 个版本上共触发了 6 个崩溃缺陷，其中 Rhino 有 1 个，JerryScript 有两个，QuickJS 有两

个，Hermes 有 1 个。针对这 6 份测试结果，笔者都已经向相应的开发者或维护者提交了缺陷报告。其中结果 3 已被 JerryScript 引擎的开发者确认是一个缺陷；结果 6 已被 Hermes 的维护者确认是一个缺陷，并已在最新发布的版本中得到了修复。结果 1 和 2 在提交后暂未得到维护者的回复；结果 4 和 5 比较特殊，即在笔者发现了这两份测试结果时，立即对其进行验证，发现在开发者最近发布的一个软件补丁中，这两个崩溃缺陷已被修复。即开发者在发布了本文测试的版本（测试时的最新版本）之后自行发现了这两个缺陷，而本文的测试结果略晚于开发者发布软件补丁的时间。

从上述结果中可以看出， JSTIFuzz 原型系统在真实的环境下和较短的测试周期中，可以触发多种 JavaScript 引擎中的崩溃缺陷。针对各测试结果的具体情况，本文会选择其中的几个测试结果所对应的测试用例进行具体的分析和说明。

5.4.3 对测试用例的研究和分析（一）

本文的 5.1.2 小节中曾介绍过，JerryScript 是一款由三星公司开发的 JavaScript 引擎，其被设计用于物联网设备中，特点是轻量级。表 15 中的结果 4 是在其 84a56ef 版本上触发的一个崩溃缺陷，对应的测试用例如图 28 所示。

```
1  var JSTIFuzzingFunc = function (size) {
2      var array = new Array(size);
3      while (size--) {
4          array[size] = 0;
5      }
6  };
7
8  var JSTIParameter0 = 904862;
9
10 JSTIFuzzingFunc(JSTIParameter0);
```

图 28 触发 JavaScript 引擎崩溃缺陷的测试用例示例（一）

首先对该测试用例的语义进行说明。第 1 行声明一个测试函数 JSTIFuzzingFunc，其有一个参数 size。第 2 行函数体内，定义一个长度为 size 的数组 array。第 3-5 行有限循环中，对 array 中的每一位赋初值为 0。第 8 行定义一个值为 904862 的 Number 类型变量 JSTIParameter0。第 10 行传入 JSTIParameter0，调用 JSTIFuzzingFunc。

下面对该用例造成 JerryScript 引擎崩溃的原因进行分析。该引擎内部通过哈希表来存储 JavaScript 的数组数据结构，但其原定大小上限为 65535（即$2^{16}-1$）。上述测试用例中，传入的数组大小为 904862，超过了上限 10 倍以上，在逐位赋初值为 0 的过程中，引擎内部为哈希表申请的内存空间逐渐变大并最终超过了 65535，导致了内

存的段错误。

最后对该测试用例的产生过程进行说明。首先，通过函数化操作，从一份原始语料中拆分得到了测试函数。接下来，根据第 3 行的“size--”， 由表 6 可见“--”是仅属于 Number 数据类型的后缀推断因子，因此类型推断模块推断参数 size 的类型为 Number。接下来自调用表达式生成模块对参数 size 生成一个 Number 类型的参数 JSTIParameter0，为其赋值为 904862。最终生成自调用表达式调用测试函数。

在该测试用例产生的过程中，类型推断模块的工作起到了最为重要的作用。根据与参数 size 相关的唯一推断因子“--”，推断出其需要传入 Number 类型的值。如果没有类型推断模块或者推断错误成其他的数据类型，该测试用例都会在第 2 或 3 行触发运行时错误提前退出，无法再触发 JerryScript 引擎的崩溃缺陷。

5.4.4 对测试用例的研究和分析（二）

同样在 5.1.2 小节中曾介绍过，Hermes 是 Facebook 公司开发的一款 JavaScript 引擎，其主要用于提升移动端应用程序的性能。表 15 中的结果 6 是在其 3ed8340 版本上触发的一个崩溃缺陷，对应的测试用例如图 29 所示。

```
1  var JSTIFuzzingFunc = function (value, replacer, space) {
2      return JSON.stringify(value, replacer, space);
3  };
4
5  var JSTIParameter0 = true;
6
7  var JSTIParameter1 = function (b) {
8      return { d1: b };
9  };
10
11 var JSTIParameter2 = 1711499276.6637239937014953;
12
13 JSTIFuzzingFunc(JSTIParameter0, JSTIParameter1, JSTIParameter2);
```

图 29 触发 JavaScript 引擎崩溃缺陷的测试用例示例（二）

首先对该用例的语义进行说明。第 1 行声明一个测试函数 JSTIFuzzingFunc，其有三个参数 value，replacer 和 space。第 2 行按顺序传入上述三个参数调用 API 函数 JSON.stringify，并将结果返回。该 API 函数的作用是将 value 转化为 JSON 字符串，并使用 replacer 对转换结果进行映射。结果 JSON 字符串的缩进量由 space 的值来控制。第 5 行定义 Boolean 类型的参数 JSTIParameter0，其值为“true”。第 7 行定义 Function 类型的参数 JSTIParameter1，其函数体在第 8 行，定义一个键值对并返回。第 11 行定义 Number 型的参数 JSTIParameter2，其值如图所示。最后，在第 13 行中顺序传入

上述三个参数，调用测试函数。

下面对该用例造成 JerryScript 引擎崩溃的原因进行分析。由于 replacer 对应的参数 JSTIParameter1 所对应的值是一个函数，这个函数的返回值是一个固定的键值对“{ d1: b }”，这个键值对又会被递归地输入到上述函数中进行映射，导致了无限递归，最终造成了 Hermes 引擎内部的栈溢出异常。

最后对该测试用例的产生过程进行说明。首先，通过函数化操作，从一份原始语料中拆分得到了测试函数。接着类型推断模块对该函数的三个参数的数据类型进行推断，发现函数体中都没有上述三个参数有效的推断因子，因此分别为三个参数随机确定数据类型。接下来自调用表达式生成模块为三个参数生成对应类型的值。最终生成自调用表达式调用测试函数。

JSON.stringify 这个 API 函数需要三个参数 value，replacer 和 space。其中 value 可以是任意数据类型，replacer 的类型限定为 Function 或 Array，space 的类型必须为 Number。而本文的类型推断模块随机确定的类型为 Boolean、Function 和 Number，因此通过类型匹配的参数调用了 JSON.stringify 函数，触发了 Hermes 因实现时未设置最大递归层数而导致的崩溃缺陷。

可见，在类型推断模块无法准确推断出参数类型时，会使用随机策略，尽最大的可能利用测试用例去触发 JavaScript 引擎中的缺陷。

5.5 本章小结

本章主要从三个部分的实验评估了 JSTIFuzz 原型系统的有效性和实用性。参数类型推断效果评估实验的结果显示，本文所提出的参数类型推断的精确率最高可以达到随机传参的 10 倍以上。代码覆盖率提升效果评估实验首先从原型系统在产生测试用例的各个主要步骤中的代码覆盖率提升效果，然后评估了使用这些测试用例对 JavaScript 引擎的代码覆盖率提升效果。模糊测试效果评估实验首先在测试集上对包括 JSTIFuzz 原型系统在内的几款对比工具进行了缺陷复现效果的评估，然后在真实环境下，对原型系统进行模糊测试的效果进行了评估。实验结果表明，JSTIFuzz 原型系统可以有效得产生具有高代码覆盖率的测试用例，并可以有效地发现 JavaScript 引擎的崩溃缺陷。

总结与展望

总结

随着软件开发行业去客户端运动的进行，Web 浏览器的性能得到了不断的提升，其中 JavaScript 引擎的性能提升占据了极其重要的位置。随着 JavaScript 语言应用场景的不断扩展，越来越多新的、运用在不同环境中的 JavaScript 引擎如雨后春笋般被人们开发出来。而实现的多元化也意味着新的 JavaScript 引擎中可能隐含着更多的软件缺陷。本文通过研究和分析 JavaScript 语言的特性和现有模糊测试方法中存在的问题，在模糊测试的测试用例产生过程中引入了类型推断技术，提出了一种基于类型推断的模糊测试方法。同时在研究的基础上实现了原型系统 JSTIFuzz，并通过设计和展开参数类型推断效果评估、代码覆盖率提升效果评估和模糊测试效果评估实验证明了本文所提出的方法的实用性和有效性。

本文主要包含以下的研究工作：

（1）本文研究了 JavaScript 语言本身所具有的特性。由于 JavaScript 是一种弱类型的脚本语言，在其他强类型语言中由语法检查即可检出的代码错误，在 JavaScript 中则必须在运行时才能得以发现。由此举例分析了测试用例覆盖率的高低在很大程度上取决于测试用例中是否会发生数据类型错误。

（2）本文研究了模糊测试方法以及当前的模糊测试方法的设计。设计模糊测试方法时需要一个测试用例的产生算法，当前最主流的算法有基于模板的生成算法和变异算法两种。前者的效果会受到编写算法的人的知识水平的限制，而现有的变异算法采用的是按位随机变异的思路，用在测试中的效率十分低下。

（3）本文基于 JavaScript 语言弱类型的特点和模糊测试的用例变异算法提出了基于类型推断的模糊测试方法。通过本文提出的类型推断方法，解决了函数的参数其数据类型无法确定的问题。配合以精确的参数生成方法调用测试函数，大幅度地提高了测试用例以及 JavaScript 引擎内部的代码覆盖率，进而提升了测试的效率。

（4）本文设计并实现了原型系统 JSTIFuzz。对重要模块的设计及其算法进行了详尽的描述。选择了多款同类的模糊测试工具和各种类型的 JavaScript 引擎，从参数类型推断效果、测试用例的代码覆盖率提升效果、在测试集上的缺陷复现情况对比和真实环境下的模糊测试等实验中，验证了原型系统的实用性和有效性。

展望

本文所提出的基于类型推断的模糊测试方法虽然在一定程度上可以有效且高效地对 JavaScript 引擎展开测试，但仍存在一些美中不足和可以继续改进的地方，具体包含以下几个方面：

（1）本文所采用的原始语料皆收集自 Github 等在线代码仓库，其获取的过程需要耗费大量的时间和精力，同时收集到的代码数量相当有限。近年来，机器学习领域中自然语言处理的技术日趋成熟。后续的研究会考虑使用自然语言处理技术让机器去学习 JavaScript 语言的语法，然后生成源源不断的原始语料。由此还可以消除整个流程中预处理的操作。

（2）本文所提出的类型推断方法目前仅包含 Array、Boolean、Number、Function 和 String 等 5 种数据类型，也仅能生成相应的 5 类参数。而 JavaScript 语言的设计中包含了面向对象的思想，Object（对象）在这门编程语言中也占据了很重要的位置。实际的 JavaScript 代码中也经常使用到 Object 数据类型，除了使用一些内置的类来创建对象，用户也可以自定义对象。在后续的研究中，笔者会考虑如何对 Object 数据类型进行有效的推断，并根据推断的结果生成具有精确属性的对象，进一步地提高代码覆盖率，并提高触发引擎缺陷的效率。

（3）本文所设计的测试用例变异方法基于类型推断传参的结果，对 5 种类型的参数分别进行精准的变异，相对于传统的测试用例变异算法具有更高的导向性。但变异的方向仍具有不确定性。后续的研究笔者会考虑使用测试用例的覆盖率或其他的指标作为衡量变异效果的标准，进一步提高测试用例变异算法的引导性和方向性。

参考文献

[1] 房鼎益，党舒凡，王怀军，等. 具有时间多样性的 JavaScript 代码保护方法[J]. 计算机应用，2015, 35(1): 72-76.

[2] JavaScript[EB/OL]. https://en.wikipedia.org/wiki/JavaScript#Beginnings_at_Netscape

[3] TIOBE Index[EB/OL]. https://www.tiobe.com/tiobe-index/, 2019

[4] Static web page[EB/OL]. https://en.wikipedia.org/wiki/ Static_web_page

[5] 浏览器/服务器结构[EB/OL]. https://zh.wikipedia.org/wiki/浏览器-服务器

[6] ECMAScript[EB/OL]. http://en.volupedia.org/wiki/ECMAScript#History

[7] Nordio M , Calcagno C , Furia C A . Javanni: a verifier for javascript.[C]// International Conference on Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2013: 231-234.

[8] Patra J, Pradel M. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data[J]. TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664, 2016.

[9] Veggalam S, Rawat S, Haller I, et al. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming[C]//European Symposium on Research in Computer Security. Springer, Cham, 2016: 581-601.

[10] Wang J, Chen B, Wei L, et al. Skyfire: Data-driven seed generation for fuzzing[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 579-594.

[11] Renáta Hodován, Ákos Kiss. Fuzzing JavaScript Engine APIs[C]// 12th International Conference on Integrated Formal Methods (iFM 2016). Springer International Publishing, 2016: 425-438.

[12] Greff K, Srivastava R K, Koutník J, et al. LSTM: A search space odyssey[J]. IEEE transactions on neural networks and learning systems, 2016, 28(10): 2222-2232.

[13] Malik R S, Patra J, Pradel M. NL2Type: inferring JavaScript function types from natural language information[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 304-315.

[14] White M , Tufano M , Vendome C , et al. Deep learning code fragments for code clone detection[C]// the 31st IEEE/ACM International Conference. ACM, 2016: 87-98.

[15] Ye J. Improved cosine similarity measures of simplified neutrosophic sets for medical diagnoses[J]. Artificial Intelligence in Medicine, 2015, 63(3): 171-179.

[16] 唐明，朱磊，邹显春. 基于 Word2Vec 的一种文档向量表示[J]. 计算机科学，2016，43(6): 214-217.

[17] Kim S , Woo S , Lee H , et al. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery[C]// Security & Privacy. IEEE, 2017: 595-614.

[18] Wang C, Jiang Y, Zhao X, et al. Weak-assert: a weakness-oriented assertion recommendation toolkit for program analysis[C]//2018 IEEE/ACM 40th International Conference on Software

Engineering: Companion (ICSE-Companion). IEEE, 2018: 69-72.

[19] Assertion base testing[EB/OL]. https://en.wikipedia.org/wiki/Test_assertion

[20] 张雄, 李舟军. 模糊测试技术研究综述[J]. 计算机科学, 2016, 43(5): 1-8, 26.

[21] Peng C, Jianzhong L, and Hao C. Matryoshka: Fuzzing Deeply Nested Branches[C]// ACM Sigsac Conference on Computer and Communications Security. ACM, 2019:499-513.

[22] Godefroid P, Peleg H, Singh R. Learn&fuzz: Machine learning for input fuzzing[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017: 50-59.

[23] Lin Y D , Liao F Z , Huang S K , et al. Browser fuzzing by scheduled mutation and generation of document object models[C]// 2015 International Carnahan Conference on Security Technology (ICCST). IEEE, 2015: 1-6.

[24] Fuzzing[EB/OL]. https://en.wikipedia.org/wiki/Fuzzing#Type of fuzzers

[25] Holler C , Herzig K , Zeller A . Fuzzing with Code Fragments[J]. Proc Usenix Security, 2012:445--458.

[26] 霍玮, 戴戈, 史记, 等. 基于模式生成的浏览器模糊测试技术[J]. 软件学报, 2018, 29(5): 1275-1287.

[27] AFL[DB/OL]. http://lcamtuf.coredump.cx/afl/

[28] AFL(American Fuzzy Lop)实现细节与文件变异[EB/OL]. https://paper.seebug.org/496/

[29] Lemieux C, Sen K. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage[J]. arXiv preprint arXiv:1709.07101, 2017.

[30] Nautilus - A grammar based feedback fuzzer[DB/OL]. https://github.com/RUB-SysSec/nautilus

[31] Aschermann C, Frassetto T, Holz T, et al. NAUTILUS: Fishing for Deep Bugs with Grammars[C]//NDSS. 2019.

[32] Zhang B, Ye J, Bi X, et al. Ffuzz: Towards full system high coverage fuzz testing on binary executables[J]. PloS one, 2018, 13(5).

[33] JavaScript function[EB/OL]. https://www.w3school.com.cn/js/js_functions.asp

[34] Global variable[EB/OL]. https://en.wikipedia.org/wiki/Global_variable

[35] JS 的作用域和全局变量[EB/OL]. https://blog.csdn.net/weixin_43187545/article/details/88858145

[36] jQuery[EB/OL]. https://api.jquery.com/category/core/

[37] Bootstrap[EB/OL]. https://getbootstrap.com/docs/4.4/getting-started/introduction/

[38] Linux 错误代码及其含义[EB/OL]. https://blog.csdn.net/u013457167/article/details/79196306?depth_1-utm_source=distribute.pc_relevant.none-task&utm_source=distribute.pc_relevant.none-task

[39] SpiderMonkey: The Mozilla JavaScript runtime[EB/OL]. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[40] ChakraCore - the core part of the Chakra JavaScript engine that powers Microsoft Edge[DB/OL]. https://github.com/Microsoft/ChakraCore

[41] Rhino - an open-source implementation of JavaScript written entirely in Java[DB/OL]. https://github.com/mozilla/rhino

[42] JerryScript - Ultra-lightweight JavaScript engine for the Internet of Things.[DB/OL]. https://github.com/jerryscript-project/jerryscript

[43] QuickJS - a small and embeddable Javascript engine.[DB/OL]. https://bellard.org/quickjs/

[44] Hermes - a small and lightweight JavaScript engine optimized for running React Native on Android.[EB/OL]. https://hermesengine.dev/

[45] CodeAlchemist - Semantics-aware Code Generation for Finding JS engine Vulnerabilities [DB/OL]. https://github.com/SoftSec-KAIST/CodeAlchemist

[46] Han H S, Oh D H, Cha S K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines[C]//NDSS. 2019.

[47] AFLFast (extends AFL with Power Schedules)[DB/OL]. https://github.com/mboehme/aflfast

[48] Code coverage[EB/OL]. http://en.volupedia.org/wiki/Code_coverage

[49] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2123-2138.

[50] Java SE Development Kit - For Java Developers. Includes a complete JRE plus tools for developing, debugging, and monitoring Java applications.[EB/OL]. https://www.oracle.com/java/technologies/javase-downloads.html

[51] IntelliJ IDEA[EB/OL]. https://www.jetbrains.com/idea/features/

[52] Commited Bug 1[DB/OL]. https://github.com/mozilla/rhino/issues/570

[53] Commited Bug 2[DB/OL]. https://github.com/jerryscript-project/jerryscript/issues/2905

[54] Confirmed Bug 1[DB/OL]. https://github.com/jerryscript-project/jerryscript/issues/3216

[55] Confirmed Bug 2[DB/OL]. https://github.com/facebook/hermes/issues/188

致谢

有人说，美好的时光总是短暂的。在西北大学求学的七个春秋是我人生中一段最美好、最充实的年华。不知不觉中我已从刚刚走进校门时那个懵懂的少年成长为了一名亟待走出校园为社会做出贡献的青年人。从本科到研究生的这些年，我学会了如何与他人交流协作，懂得了如何承担自己应负的责任，养成了自觉读书学习的良好习惯。同时，我的心中也深深地烙上了西北大学“公诚勤朴”的校训。

回首既往，无论是忧伤还是欣喜，总有一群人为我提供着无私的陪伴、支持与帮助。首先我要向我的导师房鼎益教授致以真诚的感谢和崇高的敬意。在我本科期间您讲述的自己在艰苦的条件下求学软件的经历和您治学严谨的态度，使我坚定了若要攻读硕士必定继续拜您为师的信念。事实证明我的决定没有错，这些年来您的谆谆教诲和孜孜不倦的教导使学生受益匪浅。其次，我要感谢汤战勇教授对实验室的严格管理和对我的科研工作和毕业设计的悉心指导。您对实验室规定的严格要求使我养成了按时作息的好习惯，同时为我们营造了干净整洁、舒适安全的科研工作环境。您的指导使我清楚地认识到了自己的缺点与所长，在工作中可以发挥自己所擅长的方面，逐渐弥补自己的短板，并按时顺利地完成自己的计划和任务。我要感谢叶贵鑫、张洁和翟双娇三位博士，感谢你们在科研工作和实验室日常工作中对我充当大组长、小组长等角色时所提供的支持和帮助。我还要感谢薛超、李振和田超雄三位学长，除了在日常科研工作中为我提供了数不胜数的帮助外，他们还带领着我进行健身运动，让我在学习的同时能具有强健的体魄。我还要感谢常原海和王帅两位同学，他们和我一起互相分享学习到的知识、生活中的乐趣，使我在疲惫不堪的时候可以快速恢复状态。我还要感谢柯鑫、李梦、李青佩和孔维星等几位同学，感谢你们在实验室的日常工作中和我并肩作战。我还要感谢我的组员姚厚友、田洋、李笑和瞿兴，你们的无数个等待结果的日日夜夜和没有休息的周末，换来了我们小组工作的稳步推进。我还要感谢我的女朋友王玉莉，感谢你的支持与陪伴、倾听与理解，你是我一首唱不完的歌。最后，我要感谢我的父母，是你们 20 年如一日含辛茹苦的抚养与教育，让我可以无忧无虑地完成学业。

最后，祝愿母校西北大学越办越好，祝愿从我们的 NISL 实验室走出去的学生都是祖国的栋梁之材。祝愿房老师和汤老师身体安康，桃李满天下。

攻读硕士学位期间取得的科研成果

1. 发表学术论文

[1] Tang Z, Li M, Ye G, Cao S, et al. VMGuards: A novel virtual machine based code protection system with VM security as the first class design concern[J]. Applied Sciences, 2018, 8(5): 771.

2. 申请（授权）专利

[1] 房鼎益，曹帅，叶贵鑫等. 一种基于类型推断的具有引导性的测试用例变异方法: 中国, 202010200651.0[P]. 2020-03-20. (受理)

3. 参与科研项目及科研获奖

[1] 腾讯公司合作项目，JavaScript 前端代码保护方法研究。