# 硕 士 学 位 论 文

# 面向 JavaScript 引擎报错机制的类别导向模糊测试方法

**Category-Directed Fuzzing Test Method for Error Reporting Mechanism in JavaScript Engines**


作 者 姓 名：卢 凌

学科、 专业：软件工程

学 号：22017017

指 导 教 师：江 贺


完 成 日 期：2023.3.15


大连理工大学

Dalian University of Technology

# 大连理工大学学位论文独创性声明

作者郑重声明：所呈交的学位论文，是本人在导师的指导下进行研究工作所取得的成果。尽我所知，除文中已经注明引用内容和致谢的地方外，本论文不包含其他个人或集体已经发表的研究成果，也不包含其他已申请学位或其他用途使用过的成果。与我一同工作的同志对本研究所做的贡献均已在论文中做了明确的说明并表示了谢意。

若有不实之处，本人愿意承担相关法律责任。

学位论文题目：面向JavaScript引擎报错机制的类别导向模糊测试方法

作 者 签 名：卢凌 日期：2023 年 6 月 1 日

# 大连理工大学学位论文版权使用授权书

本人完全了解学校有关学位论文知识产权的规定，在校攻读学位期间论文工作的知识产权属于大连理工大学，允许论文被查阅和借阅。学校有权保留论文并向国家有关部门或机构送交论文的复印件和电子版，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。

学位论文题目：面向JavaScript引擎报错机制的类别导向模糊测试方法

作者签名：卢凌　　日期：2023 年 6 月 1 日

导师签名：江贺　　日期：2023 年 6 月 1 日

# 大连理工大学学位论文独创性声明

作者郑重声明：所呈交的学位论文，是本人在导师的指导下进行研究工作所取得的成果。尽我所知，除文中已经注明引用内容和致谢的地方外，本论文不包含其他个人或集体已经发表的研究成果，也不包含其他已申请学位或其他用途使用过的成果。与我一同工作的同志对本研究所做的贡献均已在论文中做了明确的说明并表示了谢意。

若有不实之处，本人愿意承担相关法律责任。

学位论文题目：_____

作 者 签 名 ：_____ 日期：_____年____月日

# 摘　要

报错机制是 JavaScript 引擎中至关重要的一部分。JavaScript 引擎的开发人员不可避免地都会编写出错误的 JavaScript 程序。面对错误的程序，JavaScript 引擎报错机制应输出合理的错误信息，指出错误的原因和位置，帮助开发人员修复错误。然而，JavaScript 引擎报错机制中存在缺陷，存在缺陷的报错机制会抛出令开发人员困惑的错误信息、指向不存在错误的代码片段、阻碍开发人员修复错误的 JavaScript 程序。

为了完善 JavaScript 引擎的报错机制，本文提出了面向 JavaScript 引擎报错机制的类别导向的模糊测试方法 CAFJER。CAFJER 将 JavaScript 引擎报错机制抛出的错误信息分为 61 类，并实现了增改 API、插入函数模板以及更改符号三种变异方法，使变异后的测试用例能够触发所有类别的错误信息。给定一个种子程序，CAFJER 首先为其选择一个目标类别的错误信息，并通过动态分析得到种子文件的上下文信息。之后，CAFJER 随机选择一个目标类别错误信息对应的变异方法，根据种子程序的上下文信息生成能触发目标类别错误信息的测试用例。接着，CAFJER 将生成的测试用例输入不同 JavaScript 引擎中进行差分测试，对报错机制输出的错误信息进行比较，若其中存在差异，说明 CAFJER 可能发现了一个 JavaScript 引擎报错机制中的缺陷，并输出相应的缺陷报告。最后，根据模糊测试的历史数据，CAFJER 自动过滤重复的和无效的缺陷报告，有效减少了人工的参与。

为了验证 CAFJER 的有效性，我们将 CAFJER 与目前先进的相似方法 JEST 和 DIPROM 进行比较，实验结果表明，CAFJER 发现 JavaScript 引擎报错机制中缺陷的能力分别是 JEST 和 DIPROM 的 2.17 倍以及 26.00 倍。3 个月实验中，CAFJER 共向 JavaScript 引擎的开发者提交了 17 个缺陷报告，其中 7 个已被开发者确认，CAFJER 对真实世界中 JavaScript 引擎报错机制缺陷的检测工作得到了开发者的认可。

**关键词：**JavaScript；报错机制；错误信息；差分测试；程序变异

# Category-Directed Fuzzing Test Method for Error Reporting Mechanism in JavaScript Engines

## Abstract

Error reporting mechanism is a crucial part of JavaScript engine. Users of JavaScript engine will inevitably write wrong JavaScript programs. For programs with errors, the error reporting mechanism of JavaScript engine should output reasonable error message, point out location and cause of the error, help developers repair the program. However, there are defects in the error reporting mechanism of JavaScript engine. The error reporting mechanism of JavaScript engine with defects will throw confused error messages, point to right code fragments and hinder users from repairing the wrong JavaScript program.

To improve the error reporting mechanism of JavaScript engine, in this paper, the category directed fuzzy testing method for JavaScript engine error reporting mechanism called CAFJER is proposed. CAFJER divides the error messages thrown by the JavaScript engine error reporting mechanism into 61 categories and implements three mutation methods (adding or modifying APIs, inserting function templates, changing symbols) to generate test cases which can trigger all types of error messages. For a given seed program, CAFJER first selects an error message of the target category for it and dynamically analyzes it to obtain its context information. Secondly, CAFJER randomly selects a mutation method corresponding to the target error category and generates test cases which can trigger the target category error information according to the context information of the seed program. Thirdly, CAFJER inputs the generated test cases into different JavaScript engines for differential testing. Compare the error information output by the error reporting mechanism. If there is any difference, CAFJER may have found a defect in the error reporting mechanism of JavaScript engine and output the corresponding defect report. Finally, CAFJER automatically filters repeated and invalid test cases based on the historical data of the fuzzy test, effectively reducing manual participation.

In order to verify the effectiveness of CAFJER, CAFJER is compared with the current advanced similar methods JEST and DIPROM. The experimental results show that CAFJER has found 2.17 to 26.00 times more unique defects in the JavaScript engine error reporting mechanism. During the three-month experiment, CAFJER also submitted 17 defect reports to developers and 7 of which have been confirmed. CAFJER's detection of defects in JavaScript engine error mechanisms in the real world has been recognized by developers.

**Key Words**：JavaScript; Error reporting mechanism; Error message;Differential test;Program mutation

# 目　　录

# 1 绪论

## 1.1 研究背景与意义

JavaScript 语言是世界上最受欢迎的编程语言之一，广泛应用于客户端编程和服务器端编程。开发者调查分析公司 SlashData 的统计数据显示，目前全球有 1380 万多活跃的 JavaScript 开发人员[1]。GitHub 年度报告显示，从 2014 年开始，JavaScript 成为 GitHub 上最受欢迎的编程语言并持续至今[2]。

编写 JavaScript 代码时，开发人员常常会犯错误，写出错误的 JavaScript 程序，触发 JavaScript 引擎的报错机制，使 JavaScript 引擎的报错机制抛出错误信息。报错机制是 JavaScript 引擎中的重要组成部分，用于帮助开发人员修复错误 JavaScript 程序。面对错误的程序，理想情况下，JavaScript 引擎报错机制应输出合理的错误信息，告知开发人员输入 JavaScript 程序中错误产生的原因，指出错误发生的位置，并向开发人员提出解决错误的建议，帮助开发人员修复错误的 JavaScript 程序。

然而，JavaScript 引擎报错机制中存在缺陷，存在缺陷的 JavaScript 引擎报错机制会输出异常的错误信息。异常的错误信息可能会抛出与错误程序完全无关的错误说明，指向未出错的代码位置，使开发人员陷入困惑，阻碍开发人员对错误程序的修复工作。如下图 1.1 所示，错误代码"var enum;"会触发 JavaScript 引擎 SpiderMonkey 报错机制中的一个缺陷。

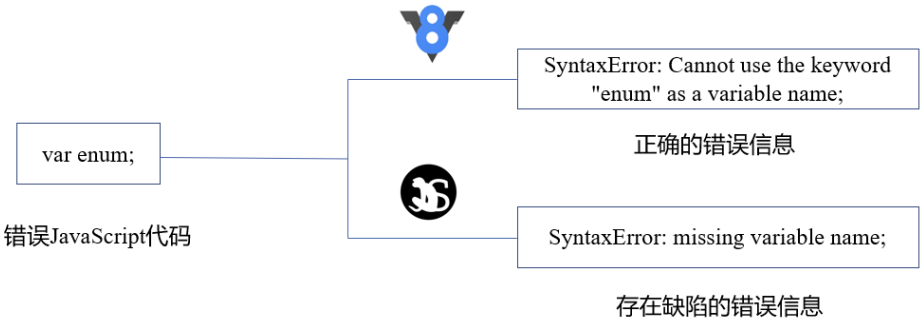


图 1.1 JavaScript 引擎报错机制缺陷示意图

Fig. 1.1 JavaScript engine error reporting mechanism defect diagram

"enum"在 JavaScript 语言的语法规则中是一个被保留的关键词，将"enum"声明为变量名是一种不符合 JavaScript 语言语法规则的行为。编译错误代码"var enum;"时，JavaScript 引擎 V8 的报错机制能发现错误代码中的语法错误，提示开发人员不能将关

键词"enum"声明为变量名(SyntaxError: Cannot use the keyword "enum" as a variable name)。但 JavaScript 引擎 SpiderMonkey 的报错机制中存在一个缺陷。面对错误代码"var enum;"，SpiderMonkey 的报错机制抛出了存在缺陷的错误信息，提示开发人员代码出错的原因在于缺少变量名(SyntaxError: missing variable name)[3]。而错误程序"var enum;"中显然存在变量名"enum"，其错误原因在于将关键词"enum"声明为了变量名而不在于缺少变量名。这一 JavaScript 引擎报错机制中的缺陷会使开发人员陷入迷茫，阻碍开发人员对错误程序的修复工作。

严重情况下，报错机制中的缺陷还会导致 JavaScript 引擎直接崩溃。如下图 1.2 所示，本文发现若在死循环代码中重复声明长度足够长的数组，编写出如下图 1.2 所示的错误 JavaScript 代码。JavaScript 引擎 V8 不能正确编译此错误程序， V8 引擎在检测到函数递归次数达到上限前会先耗尽内存，出现内存溢出（OOM）。内存溢出会使 JavaScript 引擎 V8 直接崩溃，对用户的信息安全产生威胁。

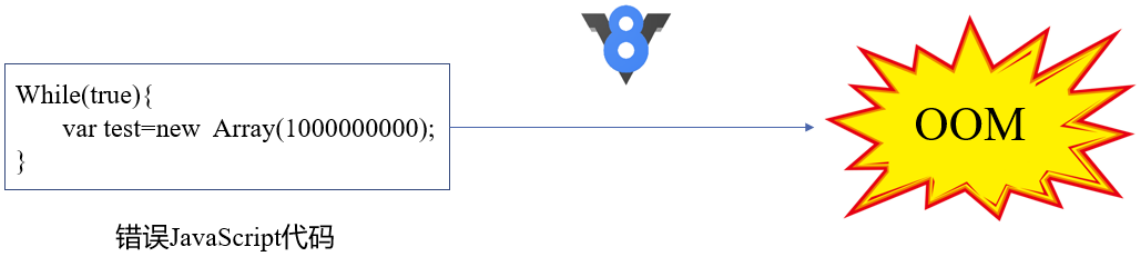


图 1.2　严重 JavaScript 引擎报错机制缺陷示意图

Fig. 1.2　Serious JavaScript engine error reporting mechanism defect diagram

同时，各个 JavaScript 引擎的开发者们也认可并重视 JavaScript 引擎报错机制中的缺陷。在三个 JavaScript 引擎 V8、JSC 以及 SpiderMonkey 的 bug 仓库中，我们共发现了 50 多个被开发者确认的 JavaScript 引擎报错机制中的缺陷。尤其是 JavaScript 引擎 V8 的开发者，设立了一个特别的专题，专门用于收集并讨论 JavaScript 引擎 V8 报错机制中的缺陷。同时，JavaScript 引擎 SpiderMonkey 的开发人员特别重视我们的工作，对于我们提交的报错机制中的缺陷报告作出了极大的肯定，评价我们提交的缺陷报告能使其作出意义重大的改进。

## 1.2 本文的主要研究内容

为确保各个 JavaScript 引擎输出内容的一致性，减少 JavaScript 引擎中的缺陷，现有大量对于 JavaScript 的研究工作，致力于检测各个 JavaScript 引擎编译机制的准确性[4-13]。现有工作依据 JavaScript 语言的语法规则设计了多种变异方法，生成语法正确的测试用例，覆盖尽可能多的语法规则。现有工作将测试用例输入不同 JavaScript 引擎进行差分测试，比较各个 JavaScript 引擎输出的结果间是否存在差异，检测各个 JavaScript 引擎编译机制面对相同 JavaScript 程序时是否表现一致，验证各个 JavaScript 引擎输出的结果是否符合行业规范。

但现有的对 JavaScript 引擎编译机制检测的工作都有所疏漏，忽视了 JavaScript 引擎的报错机制的重要性。JavaScript 引擎检测到输入程序中存在错误时，JavaScript 引擎会立即停止编译并输出错误信息。现有工作认为停止编译的 JavaScript 引擎不会产生严重缺陷，难以对公众的信息安全产生威胁，输出错误信息值得信赖。但实际上，通过本文对 JavaScript 引擎报错机制的检测，我们提交了 17 个 JavaScript 引擎报错机制中的缺陷报告，其中 7 个已被开发者确认。其中最严重的报错机制缺陷会使 JavaScript 引擎直接崩溃。同时，JavaScript 引擎报错机制输出的错误信息中也存在令人困惑的错误说明，会出现指向无关代码的错误位置。面对会触发错误信息的测试用例，现有的工作都直接抛弃了它们，略过对各个 JavaScript 引擎报错机制输出错误信息的比较，忽视了 JavaScript 引擎报错机制缺陷的潜在威胁。

此外，各个 JavaScript 引擎报错机制输出的错误信息都由 JavaScript 引擎开发人员依据个人编程经验编写，没有统一的格式。面对相同的错误 JavaScript 程序，不同 JavaScript 引擎会抛出表述格式不一致的错误信息。因此，为实现 JavaScript 引擎报错机制的检测，我们主要面临以下两个挑战：

挑战一:如何使测试用例覆盖尽可能多的错误信息类别？JavaScript 程序报错的原因是多种多样的。面对不同的错误 JavaScript 程序，JavaScript 引擎报错机制应输出对应类别的错误信息。为了保证检测的全面性，我们需要总结出 JavaScript 引擎报错机制中所有类别的错误信息，并在实验中生成能触发各类错误信息的测试用例，实现对所有类别错误信息的检测。但 JavaScript 语言的官方文档 ECMA-262[14]和各 JavaScript 引擎的说明文档都忽视了错误信息的分类问题，没有明确指出错误信息分类标准。只有部分 JavaScript 社区文档尝试对错误信息类别进行分析。但社区文档没有明确指明各类错误信息的触发条件，也难以保证文档中错误信息分类的时效性和准确性。那么，我们难以对社区文档中的内容进行验证，并删除其中的无效信息，添加其缺少的错误信息类别。

挑战二:如何辨别错误信息间的差异?我们可以将同一错误 JavaScript 程序输入不同 JavaScript 引擎，比较报错机制输出的错误信息，若其中存在差异，说明我们可能发现了一个报错机制中的缺陷。现有工作中，检测 JavaScript 引擎编译机制时，我们可以利用各个 JavaScript 引擎编译同一测试用例，直接比较各个 JavaScript 引擎的输出结果是否完全一致。如果我们发现输出结果不完全一致，说明其中可能存在一个 JavaScript 引擎编译机制中的缺陷。但错误信息由 JavaScript 引擎开发人员按照个人理解编写，各个 JavaScript 引擎会输出格式不同的错误信息，开发人员难以就错误信息的格式形成统一的规范。面对同一错误 JavaScript 程序，各 JavaScript 引擎的报错机制会抛出表达格式不同但含义相同的错误信息。我们难以直接判断差分测试输出的错误信息间是否存在差异。

为解决上述挑战，本文提出了 CAFJER(CAtegory-directed Fuzzing for JavaScript ERror message)，一种用于检测 JavaScript 引擎报错机制缺陷的类别导向的模糊测试工具。从 JavaScript 社区的文档和差分测试输出的错误信息中，我们总结得到了共 61 类错误信息。依据各类错误信息的触发条件，CAFJER 实现了增改 API、插入函数模板以及更改符号三种类别导向的变异方法。面对输入的种子程序，CAFJER 利用 MH 算法为其选定目标类别错误信息，随机选择对应的变异方法，生成能触发选定类别错误信息的测试用例，实现了对全部 61 类错误信息的覆盖，从而解决了挑战一。根据错误信息分类的原则，我们发现相同 JavaScript 引擎的同类错误信息共享独特的表达格式，并为每个 JavaScript 引擎的每类错误信息都设计了对应的正则表达式。正则表达式中固定的普通字符对错误信息中不变的表达格式进行匹配，实现对错误信息类别的识别。正则表达式中灵活的元字符对可变的错误位置等进行匹配，实现对错误位置的提取。利用正则表达式，CAFJER 将复杂的表达格式不同的错误信息简化为其对应的错误类别和错误位置。面对未知错误信息，CAFJER 将其与已有正则表达式进行匹配，从中得到此未知错误信息对应的错误类别和错误位置。差分测试中，不同于现有工作直接比较的方法，CAFJER 利用总结得到的正则表达式，将输入测试用例触发的错误信息转化为对应的错误类别和错误位置。CAFJER 对错误类型和错误位置进行比较，若各错误类型或错误位置间存在差异，说明 CAFJER 可能发现了一个 JavaScript 引擎报错机制中的缺陷，从而解决了挑战二。

## 1.3 本文的研究贡献

我们将 CAFJER 部署在三个应用最广泛的 JavaScript 引擎 V8、JSC 和 SpiderMonkey 上。为验证 CAFJER 的有效性，我们将 CAFJER 与目前先进的相似方法 JEST[9] 和 DIPROM[15]进行对比实验。实验时各个方法以相同的配置运行 24 小时，重复 5 次实验取平均值。一轮实验中，CAFJER、JEST 和 DIPROM 分别平均发现了 5.2 个、2.4 个以及 0.2 个 JavaScript 引擎报错机制中的独特缺陷，CAFJER 比 JEST 和 DIPROM 分别多发现了 2.17 倍以及 26.00 倍的独特缺陷数，实验结果表明 CAFJER 能更有效地发现 JavaScript 引擎报错机制中更多的缺陷。3 个月实验中，CAFJER 还在现实世界的 JavaScript 引擎中共发现并提交了 17 个报错机制中的缺陷报告，其中 7 个缺陷报告已被开发者确认。

本文的主要贡献如下:

（1）我们首次分析了各 JavaScript 引擎报错机制输出的错误信息，将所有的错误信息分为 61 类，并总结了各类错误信息的触发条件。

（2）针对 JavaScript 引擎报错机制检测的问题，我们提出了一种类别导向的 JavaScript 程序变异方法，它实现了增改 API、插入函数模板以及更改符号三种变异方法，使生成的测试用例能触发选定类别的错误信息，实现了对全部 61 类错误信息的覆盖。与现有先进方法 JEST 和 DIPROM 相比，我们类别导向的变异方法能更有效地检测 JavaScript 引擎报错机制缺陷，效果分别是 JEST 和 DIPROM 的 2.17 倍以及 26.00 倍，我们类别导向的变异方法能更有效地发现 JavaScript 引擎报错机制中更多的缺陷。

（3）我们实现了一种对 JavaScript 引擎报错机制进行检测的工具 CAFJER，利用 CAFJER 对真实世界中流行的 JavaScript 引擎的报错机制进行了检测。三个月实验中，我们共发现并提交了 17 个缺陷报告，其中 7 个缺陷报告已被开发者确认。

## 1.4 本文的组织结构

本文的结构安排如下：

第一章为绪论部分，首先对 JavaScript 引擎报错机制缺陷检测的研究背景以及研究意义进行了介绍，说明了 JavaScript 引擎报错机制是 JavaScript 引擎中不可或缺的一部分，其中存在的缺陷会影响 JavaScript 引擎的健全性。之后阐明了检测 JavaScript 引擎报错机制时遇到的挑战，简述 CAFJER 的工作机制，介绍 CAFJER 如何解决面临的挑战。最后，对本文的贡献进行总结。

第二章介绍 JavaScript 引擎编译机制缺陷检测和编译器错误信息缺陷检测的相关工作与技术，按照预处理、程序变异和差分测试的顺序介绍了 JavaScript 引擎编译机制的

工作流程，讲解其中的难点以及解决方案。之后，列举了目前用于编译器错误信息缺陷检测工作的先进方法，介绍了错误信息缺陷检测中测试用例的生成方法，以及错误信息检测工作与编译机制检测之间的异同。

第三章对 JavaScript 引擎报错机制的错误信息分类情况进行了详细的介绍。首先对 JavaScript 引擎错误信息的构成进行解析，提出了两条错误信息分类的原则，总结 JavaScript 社区文档。 之后，为各 JavaScript 引擎的各类错误信息构建对应的正则表达式，利用正则表达式对差分测试输出的错误信息进行检测。经过研究总结，我们将 JavaScript 引擎报错机制输出的错误信息分为了 61 类。

第四章描述了 CAFJER 工作的总体流程，依据 CAFJER 的执行顺序，依次介绍了预处理以及选择目标类别错误信息的相关技术，提出了三种类别指导的变异方法，利用差分测试识别 JavaScript 引擎报错机制输出错误信息间的差异，最后过滤无效的和重复的可疑测试用例，并输出缺陷报告。

第五章为实验的设计与分析，介绍了本文的实验环境，提出了研究问题，将 CAFJER 与现有的先进方法 JEST 和 DIPROM 比较，验证 CAFJER 发现 JavaScript 引擎报错机制缺陷的能力。编写了多个 CAFJER 的变体，通过对比实验说明 CAFJER 中各组件的有效性。最后列举了 CAFJER 提交的被 JavaScript 引擎开发者确认的缺陷，证明了 CAFJER 发现真实世界 JavaScript 引擎报错机制缺陷时的有效性。

最后一章对本文工作做出了总结，我们成功将 JavaScript 引擎错误信息细分为 61 类，实现了用于检测 JavaScript 引擎报错机制缺陷的工具 CAFJER，并提交了 7 个被开发者确认的 JavaScript 引擎报错机制中的缺陷报告。并提出了本文未来工作的研究方向。我们期望构建更多的变异方法，实现对错误 JavaScript 程序的变异，并将对报错机制缺陷的检测工作拓展到其他语言中。

# 2 相关工作与技术

## 2.1 JavaScript 引擎编译机制的检测

JavaScript 语言是世界上最流行的编程语言之一，能支撑多种网页布局、嵌入式软件以及智能手机应用程序。JavaScript 引擎用于编译在各个领域中应用的 JavaScript 代码，目前，有多个活跃的 JavaScript 引擎同时在被广泛使用，例如，Google 开发的 V8 引擎、苹果开发的 JavaScriptCore（JSC）引擎以及火狐研发的 SpiderMonkey 引擎。为了维持 JavaScript 程序运行的稳定性，保证不同 JavaScript 引擎编译相同 JavaScript 程序后输出一致的结果，各个 JavaScript 引擎的开发人员齐聚一堂，详细定义了 JavaScript 语言的语法规则，共同商议制订了 JavaScript 语言的规范 ECMA-262。同时，随着科学技术的不断发展，JavaScript 语言的开发人员不断对 JavaScript 引擎提出新的需求，ECMA-262 规范每一年都需要进行重新修正，抛弃过时的内容，增加新的功能。然而，各个 JavaScript 引擎并不能完全实现 ECMA-262 规定的语法内容，这让黑客有机可乘，利用 JavaScript 引擎中的缺陷盗取计算机用户的个人信息，对公众的信息安全产生威胁[16]。因此，为保证各个 JavaScript 引擎的安全系，统一各个 JavaScript 引擎的输出结果，减少 JavaScript 引擎中的缺陷，我们需要对 JavaScript 引擎编译机制进行检测[17]。

JavaScript 引擎编译机制的检测工作主要由三个部分组成，分别是预处理、程序变异和差分测试。

### 2.1.1 预处理

动态类型是 JavaScript 语言的一个显著特点，JavaScript 引擎编译阶段中不需要考虑函数变量执行时的语义,仅依据对程序文本的解析判断各个函数变量的类型信息。JavaScript 语言中共有 6 种数据结构的类型，分别是字符串类型 String、数值类型 Number、布尔值类型 Boolean、空值类型 Null、未定义类型 Undefined 以及对象类型 Object。JavaScript 程序中，用户不需要声明变量的类型，JavaScript 引擎根据程序语句，在编译过程中，自动分析并记录各个变量在程序的各个执行阶段中应该呈现的类型。若某变量应该呈现的类型与其现有的类型不一致，JavaScript 引擎会自动将变量强制转化为目标类型。例如，编译 JavaScript 语句"var v=3;v=true;"。编译第一行 JavaScript 代码时，变量 v 被赋值为一个 Number 型的数值常量 3，JavaScript 引擎根据语法规则推断需要 Number 类型的变量才能被 Number 类型的常量赋值，因此执行第一行代码时变量 v 的类型应为 Number 型，值设置为 3。编译第二行 JavaScript 代码时，变量 v 又被赋值为一个 Boolean 型的常量 true，JavaScript 引擎推断需要 Boolean 类型的变量才能被 Boolean

类型的常量赋值，此时变量 v 的类型被重新设置为 Boolean 型，值被设置为 true。两行 JavaScript 代码都没有直接设定变量 v 的数据类型，但根据赋值语句，JavaScript 引擎自动在编译第一行代码时将变量 v 的类型设置为 Number 型，编译第二行代码时，JavaScript 引擎将变量 v 的类型从 Number 型转化为了 Boolean 型，期间不需要对变量 v 重新进行声明，这就是 JavaScript 语言动态类型的优势所在。

虽然 JavaScript 程序中我们不需要显示声明变量的类型，但 JavaScript 语言的规范 ECMA-262 明确规定了所有 API 的各个参数能接受的类型。例如 ECMA-262 规定使用 length()函数设置一个字符串的长度时，其输入的参数必须是 Number 类型的正整数，如果输入类型不匹配的参数，JavaScript 引擎会强制将输入的值转换为 Number 类型，但类型的强制转换并不是万能的，如果输入的值不能被强制转化为需要的类型，JavaScript 引擎会抛出类型不匹配的错误信息。因此，在对 JavaScript 引擎编译机制的检测工作中，为减少生成测试用例的错误率，对种子程序进行变异操作之前，JavaScript 引擎测试工具需要进行预处理工作，了解种子程序中各个变量在程序执行各个阶段中对应的类型信息，使 JavaScript 引擎检测工具能选择符合 JavaScript 语法的变量进行变异。与之相对应的，检测 JavaScript 引擎报错机制的工作中，我们也需要进行预处理工作，在变异时选择不符合 JavaScript 语法需求的变量，将种子程序变异为不能通过编译的错误 JavaScript 程序。

静态类型的编程语言中，变量在声明阶段会确定其数据类型并在其作用域中不再改变。例如在静态类型的语言 Java 中，一旦在程序中输入"int v=3;"，此时 Java 编译器就记录下变量 v 的类型为 int 型，并且在其生命周期内不能再改变。因此，静态类型的编程语言中，我们可以通过变量的声明语句明确各个变量的类型信息。而 JavaScript 中，变量的声明语句中不包含类型信息，并且执行任意代码后，变量的类型信息有可能被强制改变。面对这种情况，JavaScript 引擎检测工作利用动态分析和静态分析技术，在预处理阶段得到各个变量在程序执行的各个阶段时的类型信息[18]。

（1）动态分析

目前的现有工作利用程序插桩对 JavaScript 程序进行动态分析，程序插桩是在保证被测程序原有逻辑完整性的基础上在程序中插入一些探针，通过探针的执行并抛出程序运行的特征数据，通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等动态信息，从而实现测试目的的方法[19]。在对 JavaScript 程序进行动态分析时，由于 JavaScript 语言动态类型的特点，每个变量的类型信息在每一行代码执行

前后都可能会出现差异，因此，我们需要在程序执行的每一行代码前后进行程序插桩，对种子程序中所有变量的类型信息进行检测并记录。

如下图 2.1 是利用程序插桩进行动态分析预处理的示例图，JavaScript 引擎测试工具在程序运行中的每一行的代码前后都插入探针，输出并记录程序中的变量 v 在每一句程序执行前后的类型变化情况。

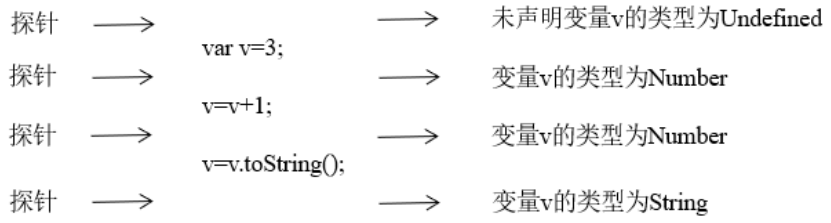


图 2.1　动态分析示意图

Fig. 2.1　Dynamic analysis diagram

预处理阶段中，JavaScript 引擎测试工具也同时收集种子程序中自定义函数的信息。由于 JavaScript 语言动态类型的特点，JavaScript 语言的自定义函数也没有标明输入参数和返回值的类型信息。若 JavaScript 引擎测试工具随意调用自定义函数，极易生成无效的测试用例。为此，种子程序中每出现一个自定义函数时，JavaScript 引擎测试工具都利用程序插桩技术，分析自定义函数的返回值类型，输出自定义函数中每一个参数在当前语境中的类型信息。后续变异阶段中，JavaScript 引擎测试工具调用已知自定义函数时，会对其输入相同类型的参数，生成新的调用语句，实现对种子程序更深层的检测工作。

（2）静态分析

然而，若在种子程序的不可执行部分，例如条件恒为假的 if 语句分支中出现了的对新变量的定义和调用，那么程序插桩技术就不可能得到对应变量的类型信息。这些不可达的语句可能会在程序变异使 if 函数的判断条件发生改变后被执行，如果在预处理阶段不记录下这些不可达部分的变量信息，就可能使变异后的程序发生错误。

SoFi[8]实现了对种子程序中不可达部分变量的预处理操作。SoFi 使用静态分析的方法得到种子程序中不可达部分变量的类型信息。动态分析中利用程序插桩技术编译种子程序得到变量类型信息，而静态分析方法是指不执行程序只分析代码片段得到变量类型信息。SoFi 通过解析 JavaScript 语法制定了 11 条分析规则。执行静态分析时，直接读取目标变量所在语句，通过词法分析规则推断出其相应的类型。例如，JavaScript 语言中要求赋值号的左值和右值应具有相同的类型，如果 SoFi 发现某个赋值号左值是未知

类型的目标变量，赋值号右值为类型为 String 的常量，那么 SoFi 就可以确定此时目标变量的类型也为 String。

预处理阶段中，结合动态分析和静态分析的技术，JavaScript 引擎检测工具可以得到种子程序中各个变量在各个不同位置的类型信息，以及各个自定义函数的返回值类型和各个参数的类型信息。

### 2.1.2 程序变异阶段

程序变异阶段中，JavaScript 引擎编译机制的检测工具根据种子文件和预处理阶段得到的变量信息以及函数信息，设计变异算法，生成测试用例。模糊测试是一种流行的错误检测技术[20-24]，能自动化或半自动化地快速生成大量测试用例。模糊测试向被测试程序输入大量的测试用例，检测被测试程序能否处理各类预期之外的输入，适用于在复杂的、真实的程序中进行检测。模糊测试技术是 JavaScript 引擎缺陷检测的常用方法，也常应用于编译器缺陷的检测工作中[25-27]。模糊测试方法可以分为两大类，分别是基于变异的模糊测试方法和基于生成的模糊测试方法[28-30]。

（1） 基于变异的模糊测试方法

基于变异的模糊测试方法通过修改种子程序生成测试用例。AFL 是最著名的基于变异的模糊测试方法[31]，AFL 的工作原理示意图如下图 2.2 所示。AFL 对种子程序采用随机变异、位翻转等多种语法盲的变异策略，若新生成的测试用例能触发种子程序中新的路径，那么 AFL 就认为新生成的测试用例是对发现缺陷有帮助的，并将它加入种子库中。AFL 语法盲的特性使其可以应用在各种语言的模糊测试中。JavaScript 引擎缺陷检测领域中，Superion[4]，Cerebro[5]和 DIE[6]拓展了 AFL 的研究，依据 JavaScript 语言的语法规则，设计更深层的变异策略。

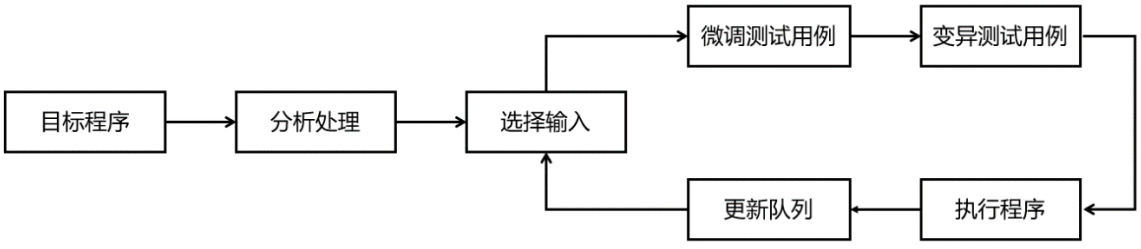


图 2.2 AFL 工作原理示意图

Fig. 2.2 AFL working principle diagram

Superion 基于 JavaScript 语法将种子程序转化为抽象的语法树，在保持输入结构不变的前提下对抽象语法树进行修剪。Superion 利用基于字典的突变方法和基于树的突变方法实现对抽象的语法树的修剪工作。基于字典的突变方法识别种子程序中的字符，在字符间插入标识，利用字典记录标识的位置，实现了以 JavaScript 字符为单位的变异策略。基于树的突变方法对抽象语法树进行解析，随机删除或替换抽象语法树中的结点。Superion 设计的两种有效的变异方法能对测试用例进行修剪，使其更有效地覆盖种子程序中新的路径。

Cerebro 发现 AFL 在选择种子程序时采用盲目的随机选择策略，忽视了不同种子程序揭示 JavaScript 引擎缺陷的潜力不同的事实。Cerebro 提出了一种基于多目标优化的算法，依据代码复杂度、文件大小、代码覆盖范围、执行时间等各项指标评估数据集中各个种子程序，按照发现 JavaScript 引擎缺陷的潜力高低，对数据集中的种子程序进行排序。选择种子程序进行变异时，Cerebro 优先选择发现 JavaScript 引擎缺陷效率高的种子程序进行变异。

DIE 提出一种保留种子程序中的理想特性的新变异方法，命名为基于要点保留的突变策略。根据 JavaScript 程序的抽象语法树，DIE 注重保留两种要点信息，分别是类型要点和结构要点。预处理阶段中，DIE 额外记录了种子程序抽象语法树中每一个节点的类型信息。程序变异阶段中，替换抽象语法树中的某个子节点时，DIE 保证替换前后的两个节点具有相同的类型信息，实现了基于保留类型要点的变异策略。同时，DIE 在变异时会避免破坏种子程序中的循环判断等结构语句，保证种子程序的数据流不发生改变，实现了基于保留结构要点的变异策略。DIE 通过基于要点保留的突变策略生成了错误率更低的测试用例，有效提高了 JavaScript 引擎的检测的效率。

除此之外，JavaScript 引擎检测工具 IFuzzer[7]使用遗传算法优化种子程序抽象语法树的突变策略，JavaScript 引擎检测工具 SoFi[8]利用静态分析技术挖掘种子程序中不可达部分的上下文信息，都成功发现了许多 JavaScript 引擎编译机制中的缺陷。但现有模糊测试方法都抛弃了错误测试用例，忽视了触发的 JavaScript 引擎报错机制的错误信息。本文提出的 CAFJER 也是一种基于变异的模糊测试方法，用于生成能触发 JavaScript 引擎报错机制的错误 JavaScript 程序。给定一个种子程序和目标类别错误信息，CAFJER 根据种子程序的上下文信息，故意违背正确的 JavaScript 语法，生成错误的程序，使输出的测试用例能触发目标类别的错误信息。

（2）基于生成的模糊测试方法

基于生成的模糊测试方法依据开发人员手动编写的上下文无关语法，从零开始生成测试用例。各个 JavaScript 引擎根据 JavaScript 语言的语法结构等信息构造自己的生成

规则，不一定需要依赖种子程序，直接生成测试用例。在 JavaScript 引擎检测工作中，JEST[9]、COMFORT[10]、CodeAlchemist[11]和 Montage[12]都是成功的利用基于生成的模糊测试方法生成测试用例的 JavaScript 引擎检测工具。

JEST 认为 JavaScript 语言的规范文件 ECMA-262 中也可能存在缺陷，提出了 N+1 版本的差分测试方法，将 N 个 JavaScript 引擎与 1 个 JavaScript 语言规范 ECMA-262 共同进行检测。然而，ECMA-262 只是单纯的文字格式的语法说明，没有办法与各个 JavaScript 引擎共同进行差分测试。为此，JEST 利用信息检索（IR）的方式，从 JavaScript 语言的规范 ECMA-262 中提取多个范式，根据范式生成测试用例。将测试用例输入各个 JavaScript 引擎中进行差分测试。如果某个 JavaScript 引擎的结果与其他 JavaScript 引擎的结果产生差异，JEST 就认为这个产生差异的 JavaScript 引擎中存在缺陷。如果有多个 JavaScript 引擎的输出结果间存在差异，JEST 就认为 JavaScript 语言的规范 ECMA-262 中可能出现了一个缺陷。通过 N+1 版本的差分测试方法，JEST 成功在 ECMA-262 中找到了多个缺陷。

COMFORT 也通过解析 ECMA-262 中的语法规则生成测试用例，但 COMFORT 侧重于分析 ECMA-262 中有关 API 的语法规则。COMFORT 构建了一个自动解析器从 ECMA-262 中提取伪代码格式的 API 规范，根据 ECMA-262 的语法规则，提取每一个 API 的名称，返回值类型，记录下各个参数的类型、取值范围、特殊值和边界值。生成测试用例时，COMFORT 优先生成能触发 API 不同情况的特殊值或边界值，有效揭露了 JavaScript 引擎中的缺陷。

CodeAlchemist 提出了一种语义感知的测试用例生成技术，自动组合语法语义正确的 JavaScript 代码片段。CodeAlchemist 发现，所有的 JavaScript 程序与抽象语法树都可以进行相互转化，并将测试用例的生成任务转化为抽象语法树的构成任务。预处理阶段，CodeAlchemist 将输入的种子程序转化为抽象语法树，将种子程序的抽象语法树进行拆分，得到一个个抽象语法树的结点。测试用例生成阶段，CodeAlchemist 组合多个结点，拼接出一棵新的抽象语法树，将新的抽象语法树再次转化为 JavaScript 程序，作为测试用例进行检测。

Montage 发现 JavaScript 引擎中的缺陷与版本更新的关系，发现新编写的代码中往往更易于出现缺陷。Montage 抛弃了传统的程序变异和代码块拼接的技术，将视线转向神经网络，训练神经网络，利用神经网络生成 JavaScript 测试用例。首先，Montage 将 JavaScript 种子程序转化为抽象语法树，并将抽象语法树进行拆分，得到一些抽象语法树中节点和片段。Montage 将这些节点和片段作为神经网络的训练集，结合自然语言处

理中的语言模型技术，对神经网络进行训练，使其能够生成 JavaScript 程序。利用训练好的神经网络模型，Montage 生成测试用例，输入 JavaScript 引擎中进行检测，发现了多个 JavaScript 引擎中的缺陷。

### 2.1.3 差分测试阶段

差分测试是一种应用广泛的软件测试技术[32-35]，常用于 JavaScript 引擎的检测工作中。差分测试假设多个基于相同规范实现的编译器理应对相同测试程序输出相同结果。若测试的多个编译器面对同一测试用例产生不同的输出时，差分测试采用投票制，若其中大多数编译器输出了相同的结果，少部分或某个编译器输出了不同的结果，差分测试就认为大部分编译器得出的结果是正确答案，少部分或某个编译器输出了不正确的结果，其中可能存在一个缺陷。

JavaScript 引擎编译机制检测工作中，差分测试将测试用例输入多个待测试的 JavaScript 引擎，比较各个 JavaScript 引擎的输出结果，若各个 JavaScript 引擎的输出结果一致，差分测试判定其中不存在缺陷。若某个 JavaScript 引擎的输出结果与其他 JavaScript 引擎的输出结果不一致，差分测试判定大多数 JavaScript 引擎的输出结果是正确答案，这个结果不一致的 JavaScript 引擎输出了错误的结果，其编译机制中可能存在缺陷，生成并输出缺陷报告。

## 2.2 编译器警告信息缺陷的检测

编译器警告信息缺陷的检测工作与 JavaScript 引擎报错机制缺陷的检测工作极为相似，都需要生成违背正确规则的测试用例。不同之处在于，编译器警告信息缺陷的检测工作需要生成能通过编译但不完善的测试用例，JavaScript 引擎报错机制缺陷的检测工作需要生成语法不正确的不能通过编译的测试用例。

Epiphron[36]是首个编译器警告信息缺陷的检测方法，首先提出了编译器警告信息检测问题。对编译器警告信息的检测工作中，Epiphron 发现编译器输出的警告信息是口语化的，是根据开发者的编程经验编写的。由于不同编译器输出的警告信息格式完全不一致，Epiphron 不能对来自不同编译器的警告消息进行直接比较，难以识别不同编译器间不一致的警告行为。各个编译器的警告消息是使用自然语言编写的，不同的编译器可能会以完全不同的格式输出警告信息。为了解决这一挑战，对于每个编译器，Epiphron 设计了特定的解析器，从自然语言描写的警告描述中提取可对齐的警告记录。Epiphron 利用 C 语言语法随机生成测试用例，将测试用例输入不同编译器，直接舍弃掉所有不能触发编译器警告信息的测试用例。差分测试中，Epiphron 通过解析器将警告信息转化为可

对齐的警告记录，比较警告记录是否一致，若各编译器输出的警告记录间存在差异，说明 Epiphron 可能发现了一个编译器警告信息中的缺陷。

目前最先进的编译器警告信息检测方法 DIPROM[15]改进了 Epiphron 的检测方法。DIPROM 发现 Epiphron 中变异方法的低效之处。Epiphron 利用随机的 C 语言语法生成语句，将语句进行随机组合，生成测试用例。这些测试用例触发编译器警告信息的比例非常低。Epiphron 生成了大量的无效测试用例，既不能触发警告信息也不能通过编译。DIPROM 在变异方法方面取得了突破，依据 C 语言的特性，设计了多种对种子程序抽象语法树进行修剪或插入操作的变异策略，例如插入一个控制语句或删除一个声明语句，大大提高了生成的测试用例的有效性。

但 DIPROM 也不能确保测试用例一定会触发警告信息。在 JavaScript 引擎报错机制缺陷检测领域，我们设计的 CAFJER 能依据各类错误信息的触发条件，利用增改 API、插入函数模板以及更改符号这 3 类变异方法，确保生成的测试用例能使 JavaScript 引擎报错机制触发目标类别的错误信息。

## 2.3 本章小结

在本节中，本文首先详细介绍了对 JavaScript 引擎编译机制的检测工作和对编译器错误信息的检测工作。根据 JavaScript 引擎编译机制的检测工作的流程，从预处理、程序变异和差分测试三个部分，介绍了 JavaScript 引擎编译机制的检测工作，阐明了涉及的技术与原理。之后，介绍编译器错误信息的检测工作，描述了生成不完全正确的测试用例和比较不统一的警告信息的解决方法。

# 3 JavaScript 错误信息分类

在本节中，为实现我们类别导向的 JavaScript 引擎报错机制检测工具 CAFJER，我们对 JavaScript 报错机制输出的错误信息进行分类。首先，我们按照自己的标准将 JavaScript 错误信息拆分为错误类型、错误说明以及错误位置三部分。之后，基于错误信息间错误类型和错误说明的表达格式和对应关系，我们提出了基于相同 JavaScript 引擎的错误信息分类原则和基于不同 JavaScript 引擎的错误信息分类原则，从社区文档中提取出有效的 55 类错误信息。接着，我们为各个 JavaScript 引擎的各类错误信息构建对应的正则表达式，通过正则表达式实现对未知错误信息的识别。利用生成的正则表达式，我们对差分测试输出的错误信息进行匹配和识别，若某错误信息与所有已知正则表达式匹配都失败，说明此错误信息可能属于新的错误信息类别。经过研究，我们从差分测试中发现了 6 类新的错误信息类别。经过研究总结，我们将 JavaScript 引擎报错机制触发的错误信息细分为了 61 类。

## 3.1 JavaScript 错误信息构成

在本节中，为实现对 JavaScript 引擎错误信息的分类，我们自己制定标准将 JavaScript 引擎报错机制抛出的错误信息分为了错误类型、错误说明以及错误位置三部分。下图 3.1 所示的是一个错误信息的示例。

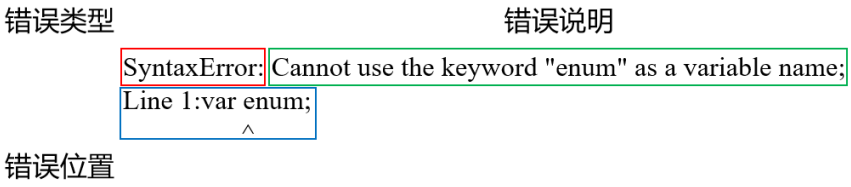


图 3.1 JavaScript 错误信息示意图

Fig. 3.1 JavaScript error message diagram

在 JavaScript 语言中，"enum"是一个被保留的关键词，将"enum"声明为一个变量名会违反 JavaScript 语言的语法规则。当我们编译错误程序"var enum;"时，JavaScript 引擎 V8 就会抛出如图 3.1 所示的错误信息，提示开发人员不能将关键词"enum"声明为一个变量名。在本文中，我们将错误信息第一行冒号之前的内容定义为错误类型，将错误信息第一行冒号之后的内容定义为错误说明，将错误信息第一行之后的内容定义为错误位置。以这个标准看待图 3.1 中所示的错误信息，其错误类型为红色部分的语法错误（SyntaxError），错误说明为图 3.1 中的绿色的部分，阐释了错误产生的原因是不能将

关键词"enum"声明为一个变量名（Cannot use the keyword 'enum' as a variable name;），错误位置对应图 3.1 中的蓝色部分，表明输入错误程序中的字符"enum"使编译产生了错误。

（1）错误类型

在本文我们自己制定的标准中，错误类型是错误信息中最先输出的部分，位于错误信息第一行的冒号前。根据已搜集到的错误信息，我们对其中的错误类型部分进行汇总和分类，发现 JavaScript 引擎错误信息中抛出的错误类型主要有 7 类，它们分别是 RangeError、ReferenceError、SyntaxError、TypeError、URIError、InternalError 以及 EvalError。错误信息中的错误类型部分会告知开发人员错误产生的根本原因。错误类型中的范围错误 RangeError 表示被调用的值不在允许的范围内，引用错误 ReferenceError 表示检测到一个无效的引用，语法错误 SyntaxError 表示输入程序中出现了违背 JavaScript 语法规则的错误代码，类型错误 TypeError 表示输入程序中出现了不符合预期的数据类型或调用了无效的方法，URI 错误 URIError 表示全局 URI 处理函数的使用与定义相违背，内部错误 InternalError 表示出现了内部错误，eval 错误 EvalError 表示发生了异常的 eval（）函数调用。

在对错误信息进行分类的原则中，我们认为如果两条错误信息间的错误类型不同，就说明触发这两条错误信息的根本原因不同，这两条错误信息不可能属于同一类的错误信息类别。

（2）错误说明

在本文我们自己制定的标准中，错误说明是对于输入程序中错误的描述。如图 3.1 中的绿色部分所示，错误说明紧接着错误类型，与错误类型在同一行输出，位于错误信息第一行的冒号后。通常，错误说明部分不会超过三十个单词。错误说明部分中，JavaScript 引擎报错机制用简洁精炼的语言，向开发人员解释错误产生的原因，给出解决错误的建议，一部分错误说明也指出了错误发生的位置。

ECMA-262 规范没有对错误说明进行任何规定，各个 JavaScript 引擎的开发人员也没有就错误说明的输出格式进行统一。错误说明的语句和表达格式由各 JavaScript 引擎开发人员按照个人理解编写。在同一 JavaScript 引擎报错机制中，相似的错误程序会触发格式相同的错误信息。但在不同 JavaScript 引擎之间，面对同一错误程序，不同 JavaScript 引擎会输出语句不同但含义相似的错误说明。

错误说明是进行错误信息分类的主要依据。我们认为，同一个 JavaScript 引擎报错机制中，如果两条错误信息之间至少存在半数以上相同的连续字符，并且对应相同的错

误类型，拥有相同的错误原因，我们就可以将这两条错误信息归为一类。不同 JavaScript 引擎报错机制之间，如果两条错误信息能被同一个错误 JavaScript 程序触发，我们也可以将这两条错误信息归为一类。

（3）错误位置

在本文我们自己制定的标准中，错误位置是错误信息中的最后一部分，由发生错误的代码位置以及引发错误的代码片段组成。如图 3.1 中的蓝色部分所示，错误位置一般在错误类别和错误说明的下一行输出。各个 JavaScript 引擎的开发者们就错误位置的输出精度也难以达成一致。根据 JavaScript 引擎和错误类别的不同，错误位置的详细程度不一。详细的错误位置会精确指向引发错误的 API、符号或函数调用，粗略的错误位置则会输出引发错误的一整行代码。例如，JavaScript 引擎编译错误 JavaScript 语句"var enum;"后，输出的错误位置可以指向"var enum;"语句本身，也可以更详细地指明错误位置是"enum"这一个字符。

错误位置也是检测 JavaScript 引擎报错机制缺陷的有效手段。对于同一个测试用例，如果各 JavaScript 引擎报错机制抛出错误信息中的错误位置有所差异，那么说明其中可能存在一个 JavaScript 引擎报错机制中的缺陷。

## 3.2 JavaScript 错误信息分类原则

基于对错误信息构成的解析，我们依据错误信息中的错误类型和错误说明部分，提出了基于相同 JavaScript 引擎的错误信息分类原则和基于不同 JavaScript 引擎的错误信息分类原则。

### 3.2.1 基于相同 JavaScript 引擎的错误信息分类原则

由于 JavaScript 引擎报错机制输出的错误信息缺乏统一的分类标准，因此面对错误原因相同的相似错误 JavaScript 程序，各个不同的 JavaScript 引擎会抛出表达格式完全不一致的错误信息。

然而，如果我们将目光聚焦于同一个 JavaScript 引擎上。面对错误原因相同的相似错误 JavaScript 程序，我们发现相同 JavaScript 引擎会输出格式一致的相似错误信息。因此，基于相似错误信息中相同的错误类型部分和重合度极大的错误说明部分，我们首先可以基于相同 JavaScript 引擎对错误信息进行分类。

我们提出了一个基于相同 JavaScript 引擎的错误信息分类原则：在相同 JavaScript 引擎报错机制抛出的错误信息中，若存在两条错误信息，其错误类型部分完全相同，其错误说明部分拥有半数以上连续的相同字符，那么我们可以认为这两条错误信息属于同一类错误信息。

下图 3.2 是对相同 JavaScript 引擎抛出的错误信息分类的示例图。在 JavaScript 语言的语法规则中，新建 Map 对象时传入的参数必须是一个可迭代的参数，若新建 Map 对象时输入不可迭代的参数，JavaScript 引擎报错机制就会抛出错误信息，提示开发人员输入的参数是不可迭代的。因此，在运行相似的错误程序"New Map(1);"和"New Map(true);"时，JavaScript 引擎 V8 会发现其中的语法错误，抛出错误信息，提示开发人员"1"是不可迭代的（TypeError: 1 is not iterable;）以及"true"是不可迭代的（TypeError: true is not iterable;）。比较输出的这两条错误信息，我们可以发现它们的错误类型部分都是类型错误 TypeError，它们的错误说明中存在 3 个相同的连续字符（is not iterable），占据了错误说明部分一半以上的字符，因此，根据基于相同 JavaScript 引擎的错误信息分类原则，我们可以将这两条错误信息分为一类。另一方面，JavaScript 引擎编译错误程序"var enum;"时会抛出不同的错误信息，将其与这两条属于同类的错误信息进行比较。它们对应的错误类型分别是类型错误 TypeError 和语法错误 SyntaxError，它们的错误类型部分并不一致。同时，我们发现它们的错误说明部分不存在任何一个相同的字符。因此，根据基于相同 JavaScript 引擎的错误信息分类原则，我们认为它们不属于同类的错误信息。

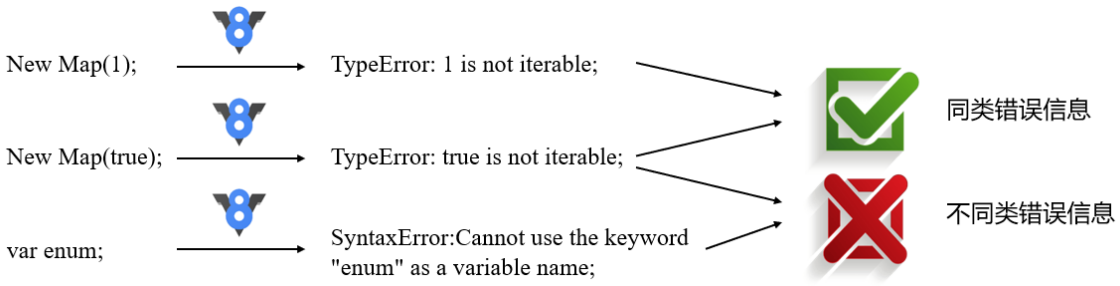


图 3.2　相同 JavaScript 引擎中错误信息分类示意图

Fig. 3.2　Error message classification in the same JavaScript engine diagram

### 3.2.2　基于不同 JavaScript 引擎的错误信息分类原则

基于相同 JavaScript 引擎的错误信息分类原则，我们对相同 JavaScript 引擎的错误信息进行分类。之后，为了使不同 JavaScript 引擎中的错误信息类别间建立对应关系。我们提出了基于不同 JavaScript 引擎的错误信息分类原则：相同错误程序在不同 JavaScript 引擎中触发的错误信息属于同一类错误信息。

下图 3.3 是对不同 JavaScript 引擎抛出的错误信息分类的示例图。JavaScript 语言中的 repeat 函数用于将目标字符串复制若干次，JavaScript 语言的语法规则要求 repeat 函数的参数必须是一个非负数。如果我们将字符串"Test"复制-1 次。使用 JavaScript 引擎编译违反语法规则的错误程序"Test.repeat(-1);"，此时，各个 JavaScript 引擎都会抛出错误信息。JavaScript 引擎 V8 抛出错误信息提示用户出现了范围错误 RangeError，存在无效的输入值（RangeError: Invalid count value;）。JavaScript 引擎 JSC 抛出错误信息提示用户出现了范围错误 RangeError，repeat()函数的参数值必须大于等于 0 且不能为无穷大（RangeError: String.prototype.repeat argument must be greater than or equal to 0 and not be Infinity;）。JavaScript 引擎 Spidermonkey 抛出错误信息也提示用户出现了范围错误 RangeError，repeat()函数的复制次数必须为非负数（RangeError: repeat count must be non-negative;）。这三条错误信息虽然表述格式不同，但根据我们提出的基于不同 JavaScript 引擎的错误信息分类原则，它们可以由同一个错误程序在不同 JavaScript 引擎中触发，因此，我们认为它们属于同一类错误信息。

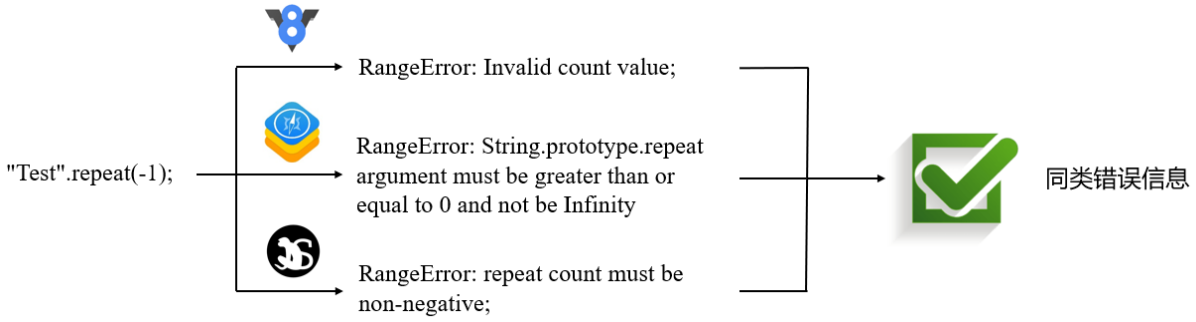


图 3.3　不同 JavaScript 引擎中错误信息分类示意图

Fig. 3.3　Error message classification in different JavaScript engine diagram

### 3.2.3　社区文档的检测

根据我们提出的基于相同 JavaScript 引擎的错误信息分类原则以及基于不同 JavaScript 引擎的错误信息分类原则，我们对 JavaScript 社区文档中收集的错误信息进行分类。

我们在社区文档中总计收集到了 73 类错误信息。首先，我们尝试在最新版本的 JavaScript 引擎中重现这些错误信息，抛弃其中已经被淘汰的 6 类错误信息，得到 67 类有效的错误信息。之后，我们利用基于相同 JavaScript 引擎的错误信息分类原则以及基于不同 JavaScript 引擎的错误信息分类原则，对同类错误信息进行合并。最后，我们总

结社区文档，得到了共 55 类不重复的有效错误信息类别，并记录下各类错误信息的触发条件。

## 3.3 正则表达式的构建

### 3.3.1 正则表达式的对应关系

在基于相同 JavaScript 引擎的错误信息分类原则中，我们提出相同 JavaScript 引擎的同类错误信息对应的错误类型应相同，对应的错误说明中应有一半以上相同的连续字符。根据这一特点，我们可以为每个 JavaScript 引擎中的每类错误信息构建对应的正则表达式。面对新的未知错误信息，我们可以利用正则表达式对其进行匹配，从而快速分析得到其对应的错误信息类别。

正则表达式是一种对字符串进行筛选的逻辑公式，它由不变的普通字符和可以灵活匹配的元字符构成。正则表达式是对其规则定义的特殊字符进行组合所构成一个规则字符串。正则表达式中的元字符可以表达各种字符的排列情况，例如"\d"表示匹配任意数字，"\w"表示匹配任意字符，"\s"表示匹配任意空白字符。当目标字符串与正则表达式进行匹配时，当且仅当目标字符串中拥有与正则表达式中的普通字符排列相同的固定字符，在元字符位置拥有与元字符规定的类型和个数相匹配字符时，正则表达式匹配才能成功。

下图 3.4 是一类错误信息与正则表达式对应关系的示意图，图 3.4 中示例的错误信息类别对应的错误信息分别是"TypeError:1 is not iterable;"以及"TypeError:true is not iterable;"，这类错误信息提示开发人员出现了一个类型错误 TypeError，输入的参数不是一个可迭代的对象。当我们为这一类错误信息构建正则表达式时，我们首先发现两条错误信息中的前半段"TypeError:"和后半段" is not iterable;"是固定不变的字段，所有此类错误信息中都会出现这固定的两段，我们在正则表达式中用普通字符去匹配这两段不变的语句。两条错误信息中的"1"和"true"表示的是错误发生的位置，它会随着错误程序的不同而改变，我们在正则表达式中利用元字符对错误说明的可变部分进行匹配。图 3.4 的示例中，我们利用元字符".*?"去匹配错误信息中不固定的"1"和"true"。".*?"在正则表达式中的含义是匹配任意字符直到下一个符合条件的字符。示例中下一个符合条件的字符是空格，我们利用正则表达式可以识别出与".*?"进行匹配的错误信息中的"1"和"true"，并将其输出，记录为此错误信息对应的错误位置。基于以上两点，我们为错误信息"TypeError:1 is not iterable;"以及"TypeError:true is not iterable;"对应的错误信息类别构建

了正则表达式"TypeError:.*? is not iterable;"，任何能与此正则表达式匹配成功的错误信息都属于此类错误信息。

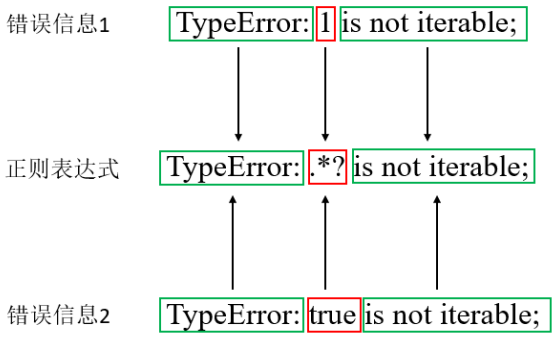


图 3.4　错误信息与正则表达式对应关系示意图

Fig. 3.4　Correspondence between error message and regular expression diagram

我们为各 JavaScript 引擎中的各类错误信息都构建了对应的正则表达式，若某未知错误信息与一条正则表达式匹配成功，说明此未知错误信息与此正则表达式对应相同的错误信息类别。因此，面对任意未知的错误信息，我们可以利用已有的正则表达式，寻找与未知错误信息相匹配的正则表达式，从而分析得到此未知错误信息对应的错误信息类别。

### 3.3.2　测试用例的检测

既然社区文档记录的错误信息类别中存在错误，那么社区文档的错误信息分类中也很有可能存在缺漏。在同一 JavaScript 引擎的同类错误信息中，错误类型和绝大部分错误说明部分始终固定不变，错误位置部分会随着输入程序产生变化。因此，为寻找缺失的错误信息类别，我们可以利用已发现的各类错误信息所构建的对应的正则表达式，对未知错误信息进行检测，判断此未知错误信息是否以被我们收录。任意 JavaScript 引擎的任意已知的错误信息类别都可以与对应的正则表达式进行成功匹配。那么如果存在某未知错误信息，它与现有的所有正则表达式都匹配失败，我们就可以认为此错误信息不属于任何已发现的错误信息类别，属于我们未发现的社区文档中未记载的新错误信息类别。

差分测试中，我们利用正则表达式对输出的所有错误信息进行匹配，如果出现未匹配成功的未知错误信息，说明此错误信息可能对应新的错误信息类别。依据基于相同 JavaScript 引擎的错误信息分类原则和基于不同 JavaScript 引擎的错误信息分类原则，我们将新发现的错误信息和现有的各类错误信息进行比较，若其与现有错误信息间存在较

大差异，我们将新发现的错误信息作为新的错误信息类别进行研究，并为其构造新的正则表达式。若此错误信息与某已知错误信息满足以上的分类原则，我们将这新发现的错误信息作为已知错误信息类别的补充。

差分测试中，我们共发现了 6 类新的错误信息类别。

## 3.4 本章小结

本节中，我们首先将错误信息分为错误类型、错误说明以及错误位置三部分。根据错误信息中错误类型和错误说明部分，我们提出了基于相同 JavaScript 引擎的错误信息分类原则和基于不同 JavaScript 引擎的错误信息分类原则。利用这两个原则对社区文档中的错误信息进行验证及合并，得到了 55 类错误信息类别。之后，我们根据相同 JavaScript 引擎的同类错误信息中错误类型相同以及错误说明中存在半数以上相同连续字符的特点，为每个 JavaScript 引擎对应的每类错误信息构建对应的正则表达式。利用得到的正则表达式对差分测试中输出的每条错误信息进行检测，从中新发现了 6 类错误信息类别。

通过研究，我们在社区文档的记录和差分测试输出的错误信息中共总结得到了 61 类错误信息，具体的错误信息类别如附录 A 所示。

# 4 CAFJER

根据分析得到的 61 类错误信息，我们设计了增改 API、插入函数模板以及更改符号三种类别导向的模糊测试变异方法，实现了用于检测 JavaScript 引擎报错机制缺陷的工具 CAFJER(CAtegory-directed Fuzzing for JavaScript ERror message)。

本节中，我们将按照 CAFJER 的工作流程，依次详细介绍 CAFJER 所涉及的各种技术方法。

## 4.1 总体流程

CAFJER 利用类别导向的模糊测试方法，实现对各 JavaScript 引擎报错机制缺陷的检测。CAFJER 的总体框架如图 4.1 所示。CAFJER 的输入是格式良好的正确 JavaScript 种子程序。给定一个种子程序，CAFJER 首先对其进行动态分析，得到种子程序的上下文信息，并为种子程序选择一个目标类别错误信息。其次，CAFJER 将带上下文信息的种子程序和目标类别错误信息输入变异器，生成能触发目标类别错误信息的测试用例。接着，CAFJER 将生成的测试用例输入不同 JavaScript 引擎中，进行差分测试，对比抛出的错误信息，判断其中是否存在差异，如果抛出的错误信息间存在差异，差分测试就将其作为可疑测试用例输出。最后，CAFJER 过滤掉重复的和无效的可疑测试用例，输出缺陷报告。

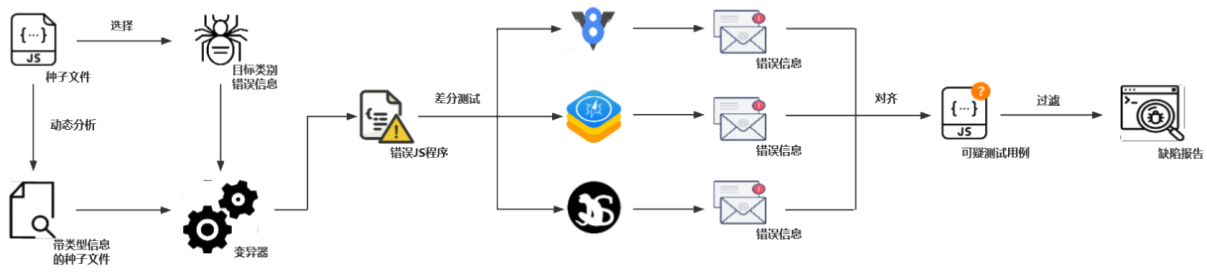


图 4.1 CAFJER 总体框架示意图

Fig. 4.1 CAFJER Overall Framework diagram

## 4.2 预处理

CAFJER 利用工具 esprima[37]和 escodegen[38]，对种子程序采用动态分析的方式进行预处理，得到种子程序的上下文信息。esprima 和 escodegen 是用于将 JavaScript 程序与抽象语法树（AST）之间进行转换的工具，esprima 可以将输入的 JavaScript 程序转化为

抽象语法树，escodegen 可以将抽象语法树转化为对应的 JavaScript 程序。如果 esprima 或 escodegen 在转化过程中报错，说明输入的种子程序中存在错误，输入的种子程序是错误的 JavaScript 程序。JavaScript 引擎在编译时遇到第一个错误时，会立即停止编译并抛出错误信息。如果 CAFJER 对错误的 JavaScript 程序进行变异，依据目标错误信息类别生成的错误代码。此时，插入的变异代码就可能位于原错误程序的不可达部分。JavaScript 引擎编译发现第一个语法错误时会立即停止，并抛出错误信息，有可能会忽视 CAFJER 生成的变异部分，并输出与选定的目标类别错误信息完全无关的错误信息。这种情况下，CAFJER 的变异方法没有对该错误程序的编译过程产生任何影响，对错误 JavaScript 程序进行变异很可能不能触发选定的目标类别错误信息，因此 CAFJER 在预处理部分抛弃了存在错误的种子程序。

预处理阶段中，CAFJER 利用工具 esprima、escodegen 以及动态分析技术，分析得到种子程序的上下文信息。种子程序的上下文信息中包含种子程序的参数信息、API 信息和函数信息。

CAFJER 收集的参数信息包括种子程序中所有的变量名称，以及各个变量在程序运行的不同阶段时对应的实际类型。分析参数信息时，CAFJER 首先将种子程序转化为抽象语法树，依次遍历抽象语法树中的所有结点，寻找种子程序中的变量声明语句，读取各个节点中涉及到的变量，得到种子程序中各个参数的参数名。搜集得到所有变量的参数名后，CAFJER 利用动态分析的程序插桩技术，在种子程序中的每一行代码间插入探针，输出此时所有变量对应的类型信息，记录下各个变量在各个位置对应的类型，总结为种子程序的参数信息。

API 信息包括 API 的名称、API 的参数个数、API 各个参数的类型以及在种子程序中 API 被调用的位置。分析 API 信息时，CAFJER 遍历抽象语法树的各个结点，每发现一个被种子程序调用的 API，依据此 API 的名称和位置，结合参数信息，记录下此 API 调用的参数的个数以及各个参数的类型。

函数信息中的函数指的是种子程序中的自定义函数，CAFJER 搜集的函数信息包括函数的名称、函数的参数个数、函数各个参数的类型以及函数出现的位置。与分析 API 信息的不同之处在于，分析函数信息额外需要监视种子程序中的函数声明语句，得到种子程序中声明的每一个函数的名称以及参数个数。当种子程序中对自定义函数进行调用时，CAFJER 利用程序插桩技术结合已有的参数信息，分析每一个自定义函数的返回值类型和各个参数对应的类型。

预处理阶段中，CAFJER 通过 esprima、escodegen 以及动态分析技术，得到种子程序的参数信息、API 信息和函数信息，为之后的变异部分做好了准备。

## 4.3 目标类别错误信息的选择

CAFJER 通过基于相同 JavaScript 引擎的错误信息分类原则和基于不同 JavaScript 引擎的错误信息分类原则检测 JavaScript 社区文档中错误信息分类的内容，并利用正则表达式匹配查找差分测试是否输出新的错误信息类别，将 JavaScript 引擎报错机制输出的错误信息总结为 61 类。

研究过程中，我们发现各类错误信息的复杂程度不同，揭示 JavaScript 引擎报错机制缺陷的潜力也不同[39]。例如，JavaScript 引擎中存在的内置函数"Number.prototype.toString()"，其接收的输入参数必须是一个 2 到 36 之间的整数。如果输入不符合要求的参数，例如小于 2 的整数 1，那么 JavaScript 引擎报错机制会抛出错误信息，提示开发人员出现了一个范围错误 RangeError，toString()函数的参数值必须在 2 到 36 之间（"RangeError: toString() radix argument must be between 2 and 36"）。这一种错误信息类别特指内置函数"Number.prototype.toString()"所调用的参数不为 2 到 36 之间的整数，判断是否触发此类错误信息的逻辑简单明了，此类错误信息几乎不可能触发 JavaScript 引擎报错机制中的缺陷。

为了辨别各类错误信息触发 JavaScript 引擎报错机制缺陷的潜力大小，变异过程中，CAFJER 记录了各类错误信息被选择为目标错误信息类别的次数，以及变异后生成的可疑测试用例数，可疑测试用例是指差分测试中使 JavaScript 引擎抛出不一致错误信息的测试用例[40,41]。CAFJER 认为，某类错误信息生成的可疑测试用例数在被其选择的总次数中占比越高，此类错误信息发现缺陷的潜力就越大。为此，如公式 4.1 所示，CAFJER 定义了各错误信息类别发现 JavaScript 引擎报错机制缺陷的期望值，具体为此类错误信息生成的可疑测试用例数除以其被选择的次数。按照发现 JavaScript 引擎报错机制缺陷的期望值大小，CAFJER 将 61 种错误信息类别进行排序，并优先选择发现报错机制缺陷期望值大的排序靠前的错误信息类别，将其作为目标类别错误信息。

$$\text{期望值} = \frac{\text{生成的可疑测试用例数}}{\text{被选择次数}} \tag{4.1}$$

CAFJER 采用梅特罗波利斯-黑斯廷斯算法(Metropolis–Hastings algorithm，简称 MH 算法)指导目标类别错误信息的选择[42]。MH 算法是一种马尔科夫方法，用于在难以直接采样的概率分布中抽取随机样本序列。理想情况下，每个错误信息类别都应该有一定的概率被选择为目标错误信息类别。生成的可疑测试用例数在被选择的总次数中占比高的，发现 JavaScript 引擎报错机制缺陷期望值大的错误信息类别会以更大的概率易被选

择。我们可以将目标错误信息类别的选择问题看作一个从概率分布中抽样的问题。给定样本的建议分布，MH 算法会将当前状态随机转化为下一个状态。转换过程中，MH 算法引入接受概率$p$来判断是否执行状态转换。根据现有的研究经验[14,32]，我们将建议分布设置为几何分布，即伯努利试验获得一次成功所需次数 X 的概率分布。设每次选择成功的概率为接受概率$p$，则第 $k$ 次选择首次成功的概率如公式 4.2 所示：

$$Pr(X = k) = (1 - p)^{k-1}p \tag{4.2}$$

CAFJER 接受新目标错误信息类别 $E_b$ 的概率如公式 4.3 所示。选择新的目标类别错误信息时，CAFJER 先记下当前目标错误信息类别，假设当前目标错误信息类别在排序中位列第 $a$ 位，记为 $E_a$。随机选择另一个位列第 $b$ 位的错误信息类别，记为 $E_b$。若 $a$ 大于 $b$，即 CAFJER 判定 $E_b$ 发现报错机制缺陷的期望值比 $E_a$ 大，CAFJER 直接将 $E_b$ 作为下一个目标错误信息类别。若 $a$ 小于 $b$，即 CAFJER 判定 $E_b$ 发现报错机制缺陷的期望值比 $E_a$ 小，根据 $a$，$b$ 的值以及接受概率$p$，CAFJER 以$(1-p)^{b-a}$的概率接受 $E_b$。若 $E_b$ 未被接受，CAFJER 重新选择一个新的错误信息类别作为 $E_b$ 并重复上述过程，直到新的错误信息类别 $E_b$ 被接受。

$$Pb(E_b|E_a) = min(1,(1-p)^{b-a}) \tag{4.3}$$

同时，我们通过以下的公式 4.4、公式 4.5 和公式 4.6 对 MH 算法中的接受概率$p$进行约束：

$$0.95 \leq \sum_{k=1}^{61} Pr(X = k) \leq 1 \tag{4.4}$$

$$\varepsilon < (1-p)^{61-1}p \tag{4.5}$$

$$p > \frac{1}{61} \tag{4.6}$$

公式 4.4 中，我们要求每一类错误信息被选择的概率和大于 0.95 且小于 1，确保累积概率接近于 1。公式 4.5 中，$\varepsilon$表示一个非常小的实数(例如 0.001)，不等号右侧的公式代表发现缺陷期望值最小的错误信息类别被选择的概率，此公式确保发现缺陷期望值最小的错误信息类别仍有被选择的概率。公式 4.6 中，不等号左侧公式表示的是发现缺陷期望值最大的错误信息类别被选择的概率，不等号右侧公式代表的是采用随机选择策略时一个错误信息类别被选择的概率，此公式确保发现缺陷潜力最大的错误信息类别被选择的概率较高，大于采用随机选择算法时此错误信息类别被选择的概率。依据以上公式 4.4、公式 4.5 和公式 4.6，经过研究和计算，我们将 MH 算法中的接受概率$p$的值设置为 0.06。

## 4.4 类别导向的变异方法

本小节中，我们利用研究得到的 61 类错误信息触发条件，设计了 API 模板、函数模板，搜集了符号信息，实现了增改 API、插入函数模板以及更改符号三种类别导向的变异方法。面对种子程序和目标类别错误信息，这三种变异方法能使变异后的测试用例触发选定的目标类别错误信息。

### 4.4.1 增改 API

增改 API 的变异方法是指在种子程序中添加或修改一句 API 语句，使变异后的测试用例能触发目标类别错误信息。CAFJER 根据各类错误信息的触发条件，收集了能触发各类错误信息的 API 及相应的参数要求，将它们制成 API 模板，实现增改 API 的变异方法。CAFJER 共制作了 432 个 API 模板，覆盖了共 40 类错误信息。下图 4.2 所示是一个 API 模板的示例图。

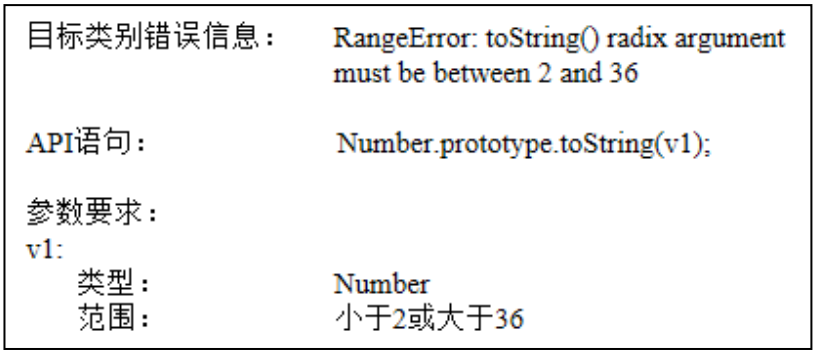


图 4.2 API 模板示例图

Fig. 4.2 API Template Example diagram

CAFJER 中的每个 API 模板都对应一类目标类别错误信息。一个 API 模板由目标类别错误信息、API 语句以及参数要求这三部分组成。API 模板的第一部分是目标类别错误信息。图 4.2 中 API 模板的目标类别错误信息是"RangeError: toString() radix argument must be between 2 and 36"。该类错误信息提示我们"Number. prototype.toString()"函数调用的参数值必须是 2 到 36 之间的整数，向"Number.prototype. toString()"函数输入超范围的参数会触发此类错误信息。API 模板的第二部分是能触发目标类别错误信息的 API 语句，并且此 API 语句是未完成的，其中含有待填充的参数 v1。图 4.2 中示例的 API 语句为"Number.prototype. toString(v1);"，其中 v1 是一个待填充的参数。API 模板的第三部分是参数要求，对 API 语句中待填充的参数进行约束，确保生成的 API 语句满足目标类别错误信息的触发条件。示例 API 语句中只有一个待填充参数 v1，API 模板要求参数 v1

是一个小于 2 或大于 36 的 Number 型数据。我们可以选择将 Number 型的常量 1 赋值给 v1，完成 API 模板中的 API 语句，得到满足条件可以触发目标类别错误信息的语句 "Number. prototype.toString(1);"。完成后的 API 语句中"Number.prototype.toString()"函数调用的参数值为 Number 型的常量 1，不满足其输入的参数值必须是 2 到 36 之间的整数的要求，会使 JavaScript 引擎发生编译错误，使报错机制抛出目标类别的错误信息 "RangeError:toString() radix argument must be between 2 and 36"。

利用 API 模板实现增改 API 的变异方法时，CAFJER 以等可能的概率随机选择增加一个 API 或修改一个 API 的变异策略。CAFJER 中增加一个 API 方法的伪代码如算法 4.1 所示。

表 4.1　增加一个 API

Tab. 4.1　Add an API

| 算法 4.1：增加一个 API |
|---|
| 输入：种子程序 $P$,目标类别错误信息 $E_{ID}$ |
| 输出：变异后的程序 $P_E$ |
| 1. $V_P$,$F_P$=getContext($P$); |
| 2. $T_{API}$=getAPITemplate($E_{ID}$); |
| 3. **foreach** $V_{req}$ in $T_{API}$ : |
| 4. $V$=$V$.addVar(getVar($V_{req}$,$V_P$ ,$F_P$)); |
| 5. $API_p$=combineAPI($T_{API}$ ,$V$); |
| 6. $P_{API}$=addAPI($P$,$API_p$); |
| 7. $P_E$=isStrictMode($P_{API}$,$E_{ID}$); |
| 8. **return** $P_E$; |

对给定的种子程序 $P$ 和目标类别错误信息 $E_{ID}$ 执行增加 API 的变异方法时，CAFJER 首先利用 esprima 和 escodegen 对种子程序进行动态分析，得到种子程序中变量的名称及类型信息 $V_P$，得到种子程序中函数的名称及输入输出的参数类型 $F_P$(line1)。之后，CAFJER 根据输入的目标类别错误信息 $E_{ID}$，从 API 模板库中随机选择一个对应的 API 模板 $T_{API}$(line2)，读取 $T_{API}$ 中对每个参数的要求 $V_{req}$，结合预处理阶段得到的变量信息 $V_P$ 和函数信息 $F_P$，生成满足要求的参数，将它们放入参数列表 $V$ 中(line3，4)。CAFJER 采用多样性的方法随机生成满足 API 模板中参数要求的参数，例如，若 API 模板要求生

成一个 String 类型的参数，CAFJER 既可以直接返回一个 String 类型的字符串，也可以选择变量信息 $V_P$ 中类型为 String 的变量，还可以调用函数信息 $F_P$ 中返回值类型为 String 的函数，或者调用 JavaScript 引擎自带的返回值类型为 String 的 API。之后，CAFJER 将生成的满足参数要求的参数替换 API 语句中对应的待填充参数，得到能触发目标错误信息类别 $E_{ID}$ 的 API 语句 $API_p$，将变异后的语句 $API_p$ 插入种子程序中任意可达的位置(line5，6)。最后，CAFJER 检测目标类别错误信息 $E_{ID}$ 是否需要使用严格模式，如果需要使用严格模式，CAFJER 在种子程序 $P$ 的第一行插入语句"use strict;"激活严格模式，最后输出变异后的测试用例 $P_E$(line7，8)。

当进行更改 API 的变异算法时，在种子程序中，CAFJER 会随机选择一个 API 模板对应的 API 调用语句。之后，CAFJER 利用种子程序中的上下文信息，生成错误 API 语句。最后，将错误 API 语句覆盖选择的 API 语句，生成能触发目标类别错误信息的测试用例。CAFJER 中更改一个 API 方法的伪代码如算法 4.2 所示。

表 4.2　更改一个 API

Tab. 4.2　Change an API

算法 4.2：更改一个 API

输入：种子程序 $P$,目标类别错误信息 $E_{ID}$

输出：变异后的程序 $P_E$

1. $V_P$,$F_P$=getContext($P$);
2. $T_{API}$=getAPITemplate($E_{ID}$,$P$);
3. $API_c$=getChangedAPI($T_{API}$,$P$);
4. **foreach** $V_{req}$ in $T_{API}$ :
5. $V$=$V$.addVar(getVar($V_{req}$,$V_P$ ,$F_P$));
6. $API_p$=combineAPI($T_{API}$ ,$V$);
7. $P_{API}$=changeAPI($P$, $API_c$,$API_p$);
8. $P_E$=isStrictMode($P_{API}$,$E_{ID}$);
9. **return** $P_E$;

与增加 API 的变异方法相似，对给定的种子程序 $P$ 和目标类别错误信息 $E_{ID}$ 执行更改 API 的变异方法。CAFJER 首先利用 esprima 和 escodegen，对种子程序进行动态分析，得到种子程序中变量信息 $V_P$ 以及函数信息 $F_P$(line1)。之后，CAFJER 根据输入的目标类别错误信息 $E_{ID}$ 和种子程序 $P$，从 API 模板库中随机选择一个对应存在的 API 模板

$T_{API}$(line2)。并且在种子程序中的随机选择一个相同 API 的调用语句 $API_c$，将 $API_c$ 作为待替换的 API 语句(line3)。读取 $T_{API}$ 中对每个参数的要求 $V_{req}$，利用变量信息 $V_P$ 和函数信息 $F_P$ 生成满足要求的参数，将它们放入参数列表 $V$ 中(line4，5)。之后，CAFJER 将生成的满足参数要求的参数替换 API 语句中对应的待填充参数，得到能触发目标错误信息类别 $E_{ID}$ 的 API 语句 $API_p$。利用变异后的语句 $API_p$，对待替换的 API 语句 $API_c$ 进行覆盖和替换(line6，7)。最后，CAFJER 检测目标类别错误信息 $E_{ID}$ 是否需要使用严格模式，如果需要使用严格模式，CAFJER 在种子程序 $P$ 的第一行插入语句"use strict;"激活严格模式，最后输出变异后的测试用例 $P_E$(line8，9)。

JavaScript 的严格模式是一种具有限制性的 JavaScript 编译模式，对 JavaScript 程序的编译过程提出了额外的规定。在 JavaScript 程序中添加语句"use strict;"，会使 JavaScript 引擎的编译过程进入严格模式。JavaScript 的严格模式能发现 JavaScript 程序中一部分不合理不严谨的语句，消除部分 JavaScript 程序中潜在的漏洞，保证了 JavaScript 程序运行时的安全可靠。同时，采用严格模式编译 JavaScript 程序，可以提高 JavaScript 引擎编译器的执行效率，有效加快 JavaScript 程序的运行速度。最后，JavaScript 的严格模式中的部分规定也为新版本 JavaScript 语言规范做出了铺垫，指明了 ECMA-262 改进的方向。

JavaScript 的严格模式使 JavaScript 引擎以更严格的条件编译程序，违反这些规定会触发仅在严格模式下出现的错误信息。例如，严格模式下，JavaScript 程序中的 Number 型的常量中不允许出现前导零，假如在 JavaScript 的严格模式下出现语句"var v=01;"，Number 型常量"01"的出现违反了 JavaScript 严格模式下的给定，Number 型常量不允许出现前导"0"，会使 JavaScript 引擎抛出仅在严格模式下出现的错误信息"SyntaxError:0-prefixed octal literals and octal escape seq are deprecated;"。而非严格模式中，JavaScript 引擎不会就出现前导"0"的 Number 型常量报错。CAFJER 记录了仅在 JavaScript 严格模式下出现的错误信息类别，若变异阶段选择了仅在 JavaScript 严格模式下出现的错误信息类别，将其作为目标类别错误信息，CAFJER 会在种子程序第一行额外添加语句"use strict;"，使 JavaScript 引擎的编译过程进入严格模式。

当 CAFJER 选择执行更改 API 的变异策略时，面对种子程序 $P$ 和目标类别错误信息 $E_{ID}$。在预处理的动态分析阶段，CAFJER 额外从抽象语法树中提取种子程序 $P$ 调用的所有 API 信息。如果目标错误信息类型 $E_{ID}$ 对应的某个 API 模板与种子程序 $P$ 调用了相同的 API，CAFJER 就随机选择一个满足这个条件的 API 模板，根据此 API 模板生成错误的 API 调用语句，利用生成的错误 API 语句替换种子程序 $P$ 中相同的 API 语句，

生成测试用例。此时，如果种子程序 $P$ 中未发现与目标错误信息类型 $E_{ID}$ 对应的任意 API 模板调用了相同的 API 语句，CAFJER 就放弃执行修改 API 的变异策略，转而执行插入 API 的变异策略。

### 4.4.2 插入函数模板

部分深层的错误信息类别难以仅靠一句 API 语句触发，CAFJER 将满足复杂错误信息触发条件的代码块装入函数模板，采用插入函数模板的变异方法触发增改 API 无法触发的错误信息类别。CAFJER 中共实现了 34 个 API 模板，覆盖了增改 API 变异方法难以触发的 16 类错误信息。

类似 API 模板，函数模板也由三部分组成，分别是目标类别错误信息、函数语句和参数要求。下图 4.3 所示的是一个函数模板的示例。

目标类别错误信息：　InternalError: too much recursion

函数语句：

```
function Template(){
    //种子文件的代码片段
    if(v1){
        Template();
    }
}
```

参数要求：

v1:

　　类型：　Boolean

　　范围：　等于true

图 4.3　函数模板示例图

Fig. 4.3　Function Template Example diagram

函数模板的第一部分指明了目标类别错误信息，第二部分函数语句指能满足目标类别错误信息触发条件的代码块，第三部分参数要求对函数语句中待填充参数提出了类型和范围的约束。图 4.3 是一个函数模板的示例。函数模板示例中第一部分目标类别错误信息指明对应的目标错误信息是"InternalError: too much recursion"。当 JavaScript 引擎编译过程中循环或迭代次数到达上限时，报错机制会抛出此类错误信息。此类错误信息提示开发人员程序的迭代次数达到上限，常被存在死循环的错误 JavaScript 程序触发。示例函数模板的第二部分函数语句中，CAFJER 首先构建一个自定义 JavaScript 函数 Template，在其中插入种子文件的代码片段丰富函数 Template 的上下文语境，有利于进行更深层的测试。之后，CAFJER 在函数 Template 中编写满足目标类别错误信息触发条

件的代码块，并保留部分待填充参数。示例中，此代码块具体表现为一个 if 语句块，当示例函数模板中 if 语句的条件恒为真时，保证 if 语句中调用函数 Template 自身的语句被反复执行，使程序出现死循环，从而触发目标类别错误信息"InternalError: too much recursion"。函数模板的第三部分是参数要求，示例的函数模板要求待填充参数 v1 的类型为 Boolean 型，值为 true。保证函数 Template 中形成死循环，从而触发目标类别的错误信息"InternalError: too much recursion"。

插入函数模板的伪代码如下表算法 4.3 所示，其工作原理与插入一个 API 的算法相似。

表 4.3　插入函数模板

Tab. 4.3　Insert function template

| 算法 4.3：插入函数模板 |
| --- |
| 输入：种子程序 $P$,目标类别错误信息 $E_{ID}$ |
| 输出：变异后的程序 $P_E$ |
| 1. $V_P$,$F_P$=getContext($P$); |
| 2. $T_F$=getFunctionTemplate($E_{ID}$); |
| 3. $T_P$=getSeedFragment($P$,$T_F$); |
| 4. **foreach** $V_{req}$ in $T_F$ : |
| 5. $V$=$V$.addVar(getVar($V_{req}$,$V_P$ ,$F_P$)); |
| 6. $F_E$=combineFunction($T_P$ ,$V$); |
| 7. $P_F$=addFunction($P$,$F_E$); |
| 8. $P_E$=isStrictMode($P_F$,$E_{ID}$); |
| 9. **return** $P_E$; |

给定一个种子程序 $P$ 和目标类别错误信息 $E_{ID}$。执行插入函数模板的变异方法时，CAFJER 首先用 esprima 和 escodegen 对种子程序进行动态分析，得到种子程序中变量的名称及类型信息 $V_P$，得到种子程序中函数的名称及输入输出的参数类型 $F_P$(line1)。之后，CAFJER 根据目标类别错误信息 $E_{ID}$ 选择一个对应的函数模板 $T_F$，并将种子程序的代码片段插入 $T_F$ 中，丰富其上下文语境，得到 $T_P$(line2，3)。读取对应函数模板 $T_F$ 中对每个参数的要求 $V_{req}$，利用预处理阶段得到的变量信息 $V_P$ 和函数信息 $F_P$，生成满足要求的参数，将它们放入参数列表 $V$ 中(line4，5)。将生成的满足函数模板中参数要求的参数替

换函数模板的函数语句 $T_P$ 中待填充的参数，生成能触发目录表错误信息 $E_{ID}$ 的函数 $F_E$(line6)。将函数 $F_E$ 插入种子程序 $P$ 中，并在种子程序 $P$ 中的任意可达位置调用函数 $F_E$，生成能触发目标类别错误信息 $E_{ID}$ 的测试用例 $P_F$(line7)。最后，CAFJER 检测目标类别错误信息 $E_{ID}$ 是否需要使用严格模式，如果需要使用严格模式，CAFJER 在测试用例 $P_F$ 的第一行插入语句"use strict;"激活严格模式，最后输出变异后的测试用例 $P_E$ (line8，9)。

### 4.4.3 更改符号

增改 API 和插入函数模板这两种变异方法共能覆盖 56 类错误信息，但无论是增改 API 或插入函数模板的变异方法都不能破坏 JavaScript 程序的结构，剩下的 5 类错误信息都与程序中的符号结构有关，不破坏语句结构的 API 模板和函数模板难以触发这些错误信息类别。

我们分析剩下的与符号结构有关的 5 类错误信息，总结这 5 类错误信息分别对应的符号集。例如，括号不匹配这一类错误信息可以对应左小括号、右小括号、左中括号、右中括号、左大括号、右大括号。

给定一个种子程序，并将括号不匹配这类错误信息作为其目标错误信息。此时，更改符号的变异方法首先扫描种子程序，搜集属于目标错误信息对应符号集的所有符号的出现位置，之后随机选择一个属于目标错误信息符号集的符号，将此符号随机替换为其他随机符号，使 JavaScript 引擎报错机制发现测试用例中出现不匹配的括号，抛出括号不匹配的目标类别错误信息，生成能触发目标类别错误信息的测试用例，实现更改符号的变异方法。

插入函数模板的伪代码如算法 4.4 所示。给定一个种子程序 $P$ 和目标类别错误信息 $E_{ID}$。执行更改符号的变异方法时，CAFJER 先通过目标类别错误信息 $E_{ID}$ 得到对应的符号集 $S$(line1)。找出种子程序 $P$ 中所有匹配符号集 $S$ 的符号生成种子程序中的符号集 $S_p$，从种子程序中的符号集 $S_p$ 中随机选出待替换的符号 $S_c$(line2，3)。根据符号集 $S$，利用随机的增删改方法，生成替换符号 $S_E$(line4)。我们可以将替换符号 $S_E$ 设置为空字符串，实现删除的变异操作。我们可以将替换符号 $S_E$ 设置为任意其他符号集 $S$ 之外的字符，实现更改的变异操作。我们可以将替换符号 $S_E$ 设置为多个字符合并的字符串，实现增加的变异操作。接着，我们利用替换符号 $S_E$ 对种子程序中待替换符号 $S_c$ 进行替换，生成测试用例 $P_F$ (line5)。最后，CAFJER 检测目标类别错误信息 $E_{ID}$ 是否需要使用严格模式，如果需要使用严格模式，CAFJER 在测试用例 $P_F$ 的第一行插入语句"use strict;"激活严格模式，最后输出变异后的测试用例 $P_E$ (line6，7)。

表 4.4　更改符号

Tab. 4.4　Change symbol

| 算法 4.4：更改符号 |
| --- |
| 输入：种子程序 $P$,目标类别错误信息 $E_{ID}$ |
| 输出：变异后的程序 $P_E$ |
| 1. $S$=getSymbolSet($E_{ID}$); |
| 2. $S_P$=getSymbol($P$,$S$); |
| 3. $S_c$=getChangedSymbol($P$,$S_p$); |
| 4. $S_E$=mutateSymbol($S$); |
| 5. $P_F$=changeSymbol($P$,$S_c$,$S_E$); |
| 6. $P_E$=isStrictMode($P_F$,$E_{ID}$); |
| 7. **return** $P_E$; |

## 4.5　差分测试

差分测试是检测 JavaScript 引擎缺陷的常用方法。差分测试假设理想中的，就同一规范实现的编译器在面对同一程序时会做出相同的反应，若出现不同的输出结果，那么其中的一个或多个编译器中可能存在缺陷[43-45]。在 JavaScript 语言检测领域，差分测试将同一测试用例输入多个不同 JavaScript 引擎，比较各 JavaScript 引擎输出结果是否相同，若输出结果间存在差异，说明可能发现了一个缺陷。

CAFJER 将变异后的测试用例输入不同 JavaScript 引擎进行差分测试，比较各 JavaScript 引擎报错机制抛出的错误信息是否相同，若输出的错误信息间存在差异，那么说明 CAFJER 可能发现了一个 JavaScript 引擎报错机制的缺陷。但 JavaScript 引擎报错机制输出的错误信息是口语化的，难以直接进行比较。为此，CAFJER 将比较各个 JavaScript 引擎报错机制输出的错误信息间是否存在差异的问题进行转化，转化为比较各个 JavaScript 引擎报错机制输出的错误信息对应的错误信息类别与错误位置是否相同[41]。我们为各 JavaScript 引擎的各类错误信息都构建了对应的正则表达式，用正则表达式固定的普通字符匹配错误信息中错误说明的不变字段和错误类型，用正则表达式的元字符匹配错误说明的灵活字段和错误位置，实现错误信息类别和正则表达式之间的对应关系，从各个 JavaScript 引擎报错机制输出的错误信息中提取对应错误信息类别和错误位置。利用正则表达式匹配，CAFJER 得到各错误信息对应错误信息类别和错误位置。比较时，CAFJER 直接比较各个 JavaScript 引擎报错机制输出的错误信息对应的错误信

息类别和错误位置是否完全相同，若其中存在差异，CAFJER 可能发现了一个 JavaScript 引擎报错机制中的缺陷。

根据各个 JavaScript 引擎报错机制输出错误信息对应的错误信息类别和错误位置的异同，CAFJER 在实验中共发现了三类差分测试结果：

（1）错误信息类别和错误位置都相同。CAFJER 认为这说明各个 JavaScript 引擎报错机制输出的错误信息内容一致，其中不存在缺陷。

（2）错误信息类别不同，错误位置相同。错误信息类别不同说明各个 JavaScript 引擎报错机制认为测试用例中触发错误的根本原因不同，此类可疑测试用例中很可能存在缺陷。

（3）错误信息类别相同，错误位置不同。错误位置不同说明各个 JavaScript 引擎报错机制认为测试用例中触发错误的错误代码位置不同，此类可疑测试用例中可能存在缺陷。

实验过程中，CAFJER 未发现任何测试用例使多个 JavaScript 引擎报错机制抛出错误信息类别和错误位置都不同的错误信息，因此 CAFJER 认为错误信息类别和错误位置都不同的情况很可能不存在。

在差分测试中，CAFJER 发现不同 JavaScript 引擎报错机制输出的错误信息间存在不一致时，CAFJER 收集引发错误信息差异的测试用例和此时各个 JavaScript 引擎抛出的错误信息，将它们打包为一个可疑测试用例，并将可疑测试用例输入 CAFJER 中的下一个组件中进行过滤。

## 4.6 无效测试用例的过滤

差分测试中，我们直接比较各 JavaScript 引擎报错机制输出的错误信息类别和错误位置是否完全相同。但实际上差分测试中发现的可疑测试用例并不都有效，并不都对应着一个新发现的缺陷。各 JavaScript 引擎中一些标准的设置以及错误位置的精度有所差异，这导致差分测试直接比较的方法会输出大量无效的不存在新缺陷的可疑测试用例。根据可疑测试用例触发的错误信息以及错误位置指向的错误代码，CAFJER 过滤以下 4 类无效的可疑测试用例。

（1）已知的缺陷

我们将 CAFJER 发现的缺陷向 JavaScript 引擎的开发人员提交并被确认后，开发人员不可能第一时间对 JavaScript 引擎报错机制中的缺陷进行修复。因此 CAFJER 发现的被确认的缺陷依旧存在 JavaScript 引擎报错机制中，CAFJER 能够通过差分测试再次发

现这些已被确认的缺陷，这类可疑测试用例不能帮助我们找到 JavaScript 引擎报错机制中新的缺陷。

为过滤已知的缺陷导致的可疑测试用例，CAFJER 分析差分测试输出的可疑测试用例中的错误信息和错误代码，判断其是否存在与已知的缺陷相同。例如，把关键词作为变量名进行声明时，SpiderMonkey 会提示缺少变量名而不是提示不能将关键词声明为变量名，这是一个被开发者确认的已知缺陷。如果差分测试中再次抛出此类的可疑测试用例，CAFJER 先确认可疑测试用例中缺少变量名这一错误信息能由已知缺陷触发，根据这一错误信息在错误代码中找到用关键词作为变量名的声明语句，从而判断此测试用例中是否存在已知缺陷，并进行过滤。

（2）对应多个类别的错误信息类别

上文中我们说明了 JavaScript 引擎报错机制抛出的错误信息没有统一的格式，并将所有 JavaScript 引擎抛出的错误信息总结为 61 类，但并不是每个 JavaScript 引擎都实现了全部 61 类错误信息，JavaScript 引擎 V8 中就只包含 54 类错误信息，V8 中的某类错误信息能对应其他 JavaScript 引擎的几类错误信息。例如 V8 的一类错误信息"SyntaxError: Unexpected token"，它表示发现了预期之外的符号。在其他 JavaScript 引擎中，预期之外的符号这类错误信息被细分为预期之外的括号、预期之外的冒号以及预期之外的引号等多类错误信息。V8 的错误信息"SyntaxError:Unexpected token"能对应其他引擎中的多类错误信息。但差分测试直接比较的方法难以识别，仍会认为预期之外的符号与预期之外的括号属于不同类别的错误信息，将其作为可疑测试用例输出。CAFJER 记录下所有一对多的错误信息类别及相应的 JavaScript 引擎。对差分测试输出的每一个可疑测试用例，CAFJER 将其与一对多的错误信息类别表进行匹配，过滤一对多的错误信息类别导致的无效可疑测试用例。

（3）标准设置的差异

各 JavaScript 引擎标准设置的差异也会导致各 JavaScript 引擎输出的错误信息间出现差异，使差分测试输出无效的可疑测试用例。过滤此类可疑测试用例时，CAFJER 检测错误位置指向的代码片段是否涉及有差异的标准。例如，JavaScript 语言的 Promise 对象能执行异步程序。但 JavaScript 引擎 JSC 不同于 V8 和 SpiderMonkey，不会在错误信息中抛出异步程序所触发的错误信息。如果 CAFJER 发现可疑测试用例的报错位置出现在 Promise 对象执行的异步程序中，且错误信息间的差异是由于 JSC 没有抛出错误信息导致的，那么 CAFJER 就认为这是一个标准设置的差异导致的无效可疑测试用例，并将其进行过滤。

（4）错误位置精度的差异

根据 JavaScript 引擎以及错误信息类别的不同，JavaScript 引擎报错机制抛出错误位置的详细程度也不同。详细的错误位置会精确指向引发错误的 API、符号或函数调用，粗略的错误位置可能会包含一整行代码。例如，假设语句"a.b();"中包含一个错误，精度高的报错位置可能指向"b()"，精度低的报错位置可能指向"a.b();"。差分测试比较时认为它们不完全相同，会将其作为可疑测试用例输出，但实际上其中不存在缺陷。CAFJER 将检测所有错误信息类别相同但错误位置不同的可疑测试用例，若其中的错误位置间存在包含关系，就判定错误位置精度的差异使差分测试产生了误报，并过滤此类无效可疑测试用例。

## 4.7 本章小结

本节依据 CAFJER 的流程，详细介绍了 CAFJER 的工作原理以及运用的技术方法。CAFJER 首先利用动态分析技术对种子程序进行预处理，之后，CAFJER 使用 MH 算法，优先选择发现 JavaScript 引擎报错机制缺陷潜力大的目标类别错误信息。变异阶段中，CAFJER 提出了增改 API、插入函数模板以及更改符号三类变异方法，使变异后的测试用例能够触发全部 61 类错误信息。之后，将变异后的测试用例输入差分测试中，比较生成的错误信息对应的错误信息类别和错误位置是否相同，若其中存在差异，将测试用例和输出的错误信息打包为可疑测试用例。最后，CAFJER 对可疑测试用例进行检测，过滤其中无效的可疑测试用例，将剩下的可疑测试用例作为缺陷报告输出。我们人工再次对 CAFJER 输出的缺陷报告进行整理，并向 JavaScript 引擎的开发者提交这些缺陷报告。

# 5 实验分析

本节主要通过实验来验证 CAFJER 检测 JavaScript 引擎报错机制中的缺陷时的有效性。

## 5.1 实验环境

本文的实验部署在一台操作系统为 Ubuntu20.04 Linux，处理器为 Intel(R) Core(TM) i7-10700 CPU @2.90GHz，内存为 32GB 的计算机上。我们基于 Python3.9 开发出了 CAFJER，并将其运行在版本为 2022.2 的 pycharm 上。

本文选用 V8，JSC 和 SpiderMonkey 这三个 JavaScript 引擎作为实验对象，它们都是运营多年的成熟 JavaScript 引擎，拥有活跃的社区。它们的开发人员认可并重视报错机制中的缺陷，bug 仓库中能找到此类的缺陷报告。CAFJER 利用 esprima 和 escodegen 对 JavaScript 程序与抽象语法树进行转化操作，其中 esprima 采用的版本是 v4.0.1，escodegen 采用的版本是 v2.0.0。

## 5.2 研究问题

在本文的研究中，我们利用 CAFJER 检测 JavaScript 引擎报错机制中的缺陷。为了验证 CAFJER 检测 JavaScript 引擎报错机制缺陷时的有效性，我们设计了以下五个问题进行研究:

RQ1：与现有的先进方法相比，CAFJER 发现 JavaScript 引擎报错机制缺陷的能力如何?

RQ2：CAFJER 中类别导向的变异方法对发现 JavaScript 引擎报错机制缺陷的贡献有多大?

RQ3：CAFJER 中选择目标类别错误信息的 MH 算法是否有助于揭示 JavaScript 引擎报错机制中更多的缺陷？

RQ4：CAFJER 总结的 61 类错误信息是否全面？CAFJER 中类别导向的变异方法能否覆盖所有的错误信息类别？

RQ5：CAFJER 发现真实世界 JavaScript 引擎报错机制缺陷的能力如何?

RQ1 中我们将 CAFJER 与目前先进的相似方法 JEST[9]和 DIPROM[15]进行比较，按照相同的配置，比较它们在相同的时间内生成的可疑测试用例数、生成可疑测试用例比例以及发现的独特缺陷数，验证 CAFJER 的有效性。

RQ2 中我们编写了 CAFJER 的变体 $CAFJER_r$，$CAFJER_r$ 将 CAFJER 中类别导向的模糊测试方法改写为随机变异方法，随机增删改种子程序中任意的元素。比较 CAFJER 和 $CAFJER_r$ 的性能，验证 CAFJER 类别导向模糊测试方法的有效性。

RQ3 中我们编写了 CAFJER 的变体 $CAFJER_c$，$CAFJER_c$ 将 CAFJER 中利用 MH 算法选择目标类别错误信息的方法改写为随机选择目标类别错误信息的方法，比较 CAFJER 和 $CAFJER_c$ 的性能，验证 CAFJER 中利用 MH 算法选择目标类别错误信息的有效性。

RQ4 中我们将检测总结得到的 61 类错误信息能否覆盖所有 CAFJER 生成的测试用例以及 CAFJER 的实验过程中是否会出现新的错误信息类别。将 CAFJER 与所有对照组共同运行 24 小时，比较它们触发错误信息类别的数量和速度，验证 CAFJER 对错误信息类别的覆盖率以及覆盖速度。

RQ5 中我们列举了 CAFJER 发现并被确认的 JavaScript 引擎报错机制缺陷，验证 CAFJER 在真实世界流行的 JavaScript 引擎中发现报错机制缺陷的能力。

## 5.3 与现有先进方法的比较

当前没有其他检测 JavaScript 引擎报错机制的有效方法。JEST 是目前先进的 JavaScript 引擎编译机制缺陷的检测方法。JEST 解析 ECMA-262 的语法规则，利用信息检索的方式，从 JavaScript 语言的规范 ECMA-262 中提取多个范式，依此生成测试用例。JEST 在实验过程中会生成大量语法错误的测试用例，本次实验中我们忽视 JEST 生成的语法正确的测试用例，只取 JEST 生成的能触发错误信息的测试用例进行研究。DIPROM 是目前先进的编译器警告信息缺陷的检测方法，通过对种子程序抽象语法树进行多种修剪和插入操作生成测试用例，我们将它改写应用于 JavaScript 引擎报错机制缺陷的检测工作中。本节中我们用 JEST 和 DIPROM 作为对照组，验证 CAFJER 检测 JavaScript 引擎报错机制缺陷时的有效性。

我们根据可疑测试用例数、可疑测试用例比例、独特缺陷数、P 值(P-value)以及效应量(effect size)这五个指标对 CAFJER、JEST 和 DIPROM 进行对比。可疑测试用例数是指差分测试阶段中，发现的能导致各个 JavaScript 引擎报错机制输出不一致错误信息的测试用例数，可疑测试用例比例是指生成的可疑测试用例数在变异生成的测试用例的总数中的占比，独特缺陷数是指在一轮测试中检测方法发现的独特的 JavaScript 引擎报错机制中的缺陷个数。P 值是用来判定假设检验结果的一个参数，P 值是指当原假设为真时，与所得到的样本观察结果相比更极端结果的出现概率。若 P 值很小，说明依据原假设的实验结果情况出现的概率很小，我们有理由怀疑原假设的可靠性。P 值越小，表

明实验的结果越显著，我们越有足够的依据拒绝原假设。效应量是指由某一因素引起的差别，是衡量处理效应大小的指标，它表示不同处理下的总体均值之间差异的大小，可以用来比较两组数据之间的优劣。

我们将三种方法 CAFJER、JEST 和 DIPROM 布置在相同的实验环境中。为了减小偶然性对实验的影响，我们共进行了 5 轮实验，每轮实验持续 24 小时，将得到的可疑测试用例数和独特缺陷数取平均数，用 5 次实验生成的总测试用例数除以发现的可疑测试用例数计算得到可疑测试用例比例，依据独特缺陷数计算 JEST 和 DIPROM 相较于 CAFJER 的 P 值和效应量，实验结果如表 5.1 所示。

表 5.1　CAFJER 与现有方法的性能比较结果

Tab. 5.1　Performance comparison results of CAFJER and existing methods

| | CAFJER | JEST | DIPROM |
|---|---|---|---|
| 可疑测试用例数 | 34699.4 | 26412.0 | 6105.8 |
| 可疑测试用例比例 | 40.98% | 23.40% | 12.10% |
| 独特缺陷数 | 5.2 | 2.4 | 0.2 |
| P-value | - | <0.01 | <0.01 |
| 效应量 | - | 0.953 | 0.987 |

表 5.1 数据显示，一轮实验中，CAFJER 平均发现了 34699.4 个可疑测试用例，而 JEST 和 DIPROM 平均只发现了 26412.0 个以及 6105.8 个可疑测试用例，CAFJER 多发现了 31.38%以及 458.30%的可疑测试用例数。CAFJER 发现可疑测试用例的比例高达 40.98%，显著优于 JEST 的 23.40%和 DIPROM 的 12.10%。由于差分测试的输出中存在 4 类无效的可疑测试用例，因此 CAFJER 一轮实验中平均只发现了 5.2 个独特缺陷，而 JEST 和 DIPROM 一轮实验中平均只发现了 2.4 个以及 0.2 个独特缺陷，CAFJER 发现的独特缺陷数是 JEST 和 DIPROM 的 2.17 倍以及 26.00 倍。实验计算了 JEST 和 DIPROM 相对于 CAFJER 的 P 值和效应量。表 5.1 显示 JEST 和 DIPROM 的 P 值都小于 0.01，说明 CAFJER 的表现优于 JEST 和 DIPROM。在 CAFJER 与对照组的比较中，若效应量大于 0.5 说明 CAFJER 更有效，若效应量等于 0.5 说明 CAFJER 和对照组一样有效，若效应量小于 0.5 说明对照组更有效。表 5.1 中 JEST 和 DIPROM 的效应量都大于 0.9，证明 CAFJER 检测 JavaScript 引擎报错机制缺陷的能力显著优于 JEST 和 DIPROM。

两个对照组中，JEST 能快速生成更多的测试用例，但输出可疑测试用例的比例不高，发现报错机制缺陷的能力不强，找到的独特缺陷数也远少于 CAFJER。另一个对照

组 DIPROM 的表现最差，说明 DIPROM 对种子程序抽象语法树进行修剪和插入操作的变异方法不适用于 JavaScript 引擎报错机制缺陷的检测。

基于以上的数据和分析，我们可以对 RQ1 做出解答，与现有的先进模糊测试方法相比，CAFJER 检测 JavaScript 引擎报错机制的效果更好。

## 5.4 类别导向变异方法的有效性

为验证 CAFJER 类别导向变异方法的有效性，我们编写了 CAFJER 的变体 $CAFJER_r$，与 CAFJER 进行对比实验。$CAFJER_r$ 将 CAFJER 中类别导向的变异方法修改为随机的变异方法，CAFJERr 随机选择种子程序中任意的参数、API 或符号，对它们进行随机的增删改操作。

我们将两种方法 CAFJER 和 $CAFJER_r$ 布置在相同的实验环境中。为减小偶然性对实验的影响，我们共进行了 5 轮实验，每轮实验持续 24 小时。将得到的可疑测试用例数和独特缺陷数取平均数，用 5 次实验生成的总测试用例数除以发现的可疑测试用例数计算得到可疑测试用例比例，按照独特缺陷数计算 $CAFJER_r$ 相较于 CAFJER 的 P 值和效应量。CAFJER 与 $CAFJER_r$ 的性能比较表如表 5.2 所示。

表 5.2　CAFJER 与 $CAFJER_r$ 的性能比较结果

Tab. 5.2　Performance comparison results between CAFJER and $CAFJER_r$

| | CAFJER | $CAFJER_r$ |
|---|---|---|
| 可疑测试用例数 | 34699.4 | 20723.6 |
| 可疑测试用例比例 | 40.98% | 26.46% |
| 独特缺陷数 | 5.2 | 0.6 |
| P-value | - | <0.01 |
| 效应量 | - | 0.982 |

表 5.2 数据显示，每一轮实验中，CAFJER 平均发现了 34699.4 个可疑测试用例，$CAFJER_r$ 平均发现了 20723.6 个可疑测试用例，CAFJER 比 $CAFJER_r$ 多发现 67.44%的可疑测试用例数。CAFJER 发现可疑测试用例的比例为 40.98%，相较于 $CAFJER_r$ 发现可疑测试用例的比例 26.46%提高了 14.52%。在每一轮实验中，CAFJER 平均了发现 5.2 个独特缺陷，$CAFJER_r$ 平均只发现了 0.6 个独特缺陷，CAFJER 发现的独特缺陷数比 $CAFJER_r$ 多 7.67 倍，发现的独特缺陷数远远多于 $CAFJER_r$。因此，我们可以断定 CAFJER 发现报错机制缺陷的能力优于 $CAFJER_r$。

依据 5 轮实验中发现的独特缺陷数，计算 $CAFJER_r$ 相较于 CAFJER 的 P 值和效应量。表 5.2 显示 $CAFJER_r$ 相较于 CAFJER 的 P 值小于 0.01，说明 CAFJER 发现 JavaScript 引擎报错机制缺陷的能力优于 $CAFJER_r$。同时，计算得出 $CAFJER_r$ 相较于 CAFJER 的效应量大于 0.9，也证明了 CAFJER 检测 JavaScript 引擎报错机制缺陷时比 $CAFJER_r$ 更为有效。

实验结果表明 CAFJER 的性能显著优于 $CAFJER_r$。基于以上的数据和分析，我们可以对 RQ2 做出解答，类别导向的变异方法在 CAFJER 的工作中起到了至关重要的作用，有效提高了 CAFJER 检测 JavaScript 引擎报错机制缺陷的能力。

## 5.5 目标类别错误信息选择算法的有效性

为验证 CAFJER 中错误信息类别选择算法的有效性，我们编写了 CAFJER 的变体 $CAFJER_c$ 进行对比实验。$CAFJER_c$ 将 CAFJER 中选择目标错误信息类别的 MH 方法修改为随机选择目标错误信息类别的方法。变异阶段中 $CAFJER_c$ 从 61 个错误信息类别中随机选择一个错误信息类别作为目标错误信息类别。

我们将 CAFJER 和 $CAFJER_c$ 在相同的实验环境中运行 5 轮实验，每一轮实验持续 24 小时，得到平均的可疑测试用例数、可疑测试用例比例以及发现的独特缺陷数。CAFJER 与 $CAFJER_c$ 的性能比较表如表 5.3 所示。

表 5.3 CAFJER 与 $CAFJER_c$ 的性能比较结果

Tab. 5.3 Performance comparison results between CAFJER and $CAFJER_c$

| | CAFJER | $CAFJER_c$ |
|---|---|---|
| 可疑测试用例数 | 34699.4 | 32252.4 |
| 可疑测试用例比例 | 40.98% | 34.22% |
| 独特缺陷数 | 5.2 | 5.0 |

表 5.3 数据显示，每一轮实验中，CAFJER 平均发现了 34699.4 个可疑测试用例，$CAFJER_c$ 平均发现了 32252.4 个可疑测试用例。CAFJER 发现可疑测试用例的比例为 40.98%，高于 $CAFJER_c$ 的 34.22%。CAFJER 平均在每一轮实验中发现了 5.2 个独特的缺陷，$CAFJER_c$ 只平均发现了 5.0 个独特的缺陷。从这两组数据中我们可以发现实际上 $CAFJER_c$ 平均在每一轮实验中生成了比 CAFJER 更多的测试用例，但 CAFJER 的可疑测试用例比例比 $CAFJER_c$ 高 6.76%，发现的独特缺陷数比 $CAFJER_c$ 多 0.2。CAFJER 生成的测试用例更有希望发现 JavaScript 引擎报错机制中的缺陷。

实验结果表明在可疑测试用例数、可疑测试用例比例和发现的独特缺陷数这三方面，CAFJER 的表现都略优于 CAFJER$_c$。基于以上的数据和分析，我们可以对 RQ3 做出解答，CAFJER 中目标错误信息类别选择算法能够在一定程度上提高 CAFJER 检测 JavaScript 引擎报错机制缺陷的能力。

## 5.6 错误信息类别的覆盖情况

我们通过对 JavaScript 引擎社区文档的验证与对 CAFJER 生成测试用例触发的错误信息类别的研究，一共总结得到了 61 类错误信息。本节中，我们根据上述实验研究，分析每一种变异方法平均在每一轮实验中触发的错误信息种类数，寻找是否存在未发现的错误信息类别，比较 CAFJER 与其他各种方法覆盖错误信息类别的数量以及覆盖各类错误信息的速度。

首先，我们将 5 类实验方法 CAFJER、CAFJER$_c$、CAFJER$_r$、JEST 以及 DIPROM 运行 5 轮，每轮持续 24 小时，统计各种变异方法平均在一轮实验中覆盖的错误信息类别数以及发现的新错误信息类别数，实验数据如表 5.4 和图 5.1 所示。

表 5.4 错误信息类别覆盖情况

Tab. 5.4 Error message category coverage

| | CAFJER | CAFJER$_c$ | CAFJER$_r$ | JEST | DIPROM |
|---|---|---|---|---|---|
| 覆盖错误信息类别数 | 61.0 | 61.0 | 35.0 | 12.0 | 16.0 |
| 新错误信息类别数 | 0 | 0 | 0 | 0 | 0 |

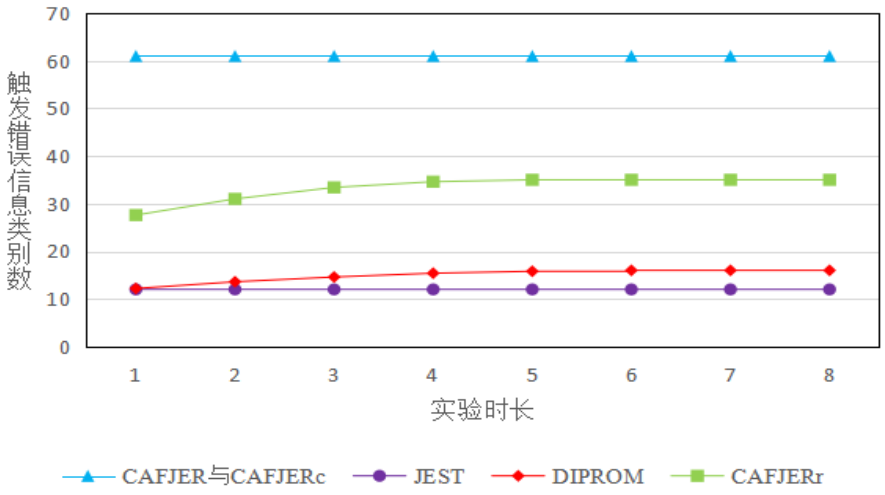


图 5.1 触发错误信息类别数与实验时长关系图

Fig. 5.1 Relationship between the number of error message categories and experiment time diagram

从表 5.4 中我们可以发现，实验中的所有 5 种变异方法都没有发现新的错误信息类别。据此我们可以得出结论，本文总结得到的 61 种错误信息类别能够覆盖各个 JavaScript 引擎报错机制输出的所有错误信息，本文分析得到的 61 类错误信息的分类方法是全面的。

5 种变异方法中，CAFJER 采用 MH 算法选择目标错误信息类别，$CAFJER_c$ 采用随机方法以平均的概率选择目标错误信息类别，CAFJER 和 $CAFJER_c$ 都成功实现了对所有 61 类错误信息的覆盖。$CAFJER_r$ 将目标类别错误信息导向的变异方法替换为随机选择种子程序中的任意参数、API 或符号进行增删改的变异方法，覆盖了 35 类错误信息。目前先进的相似变异方法 JEST 和 DIPROM 分别只覆盖了 12 类错误信息以及 16 类错误信息。CAFJER 覆盖的错误信息类别数与 $CAFJER_c$ 相同，且远远大于 $CAFJER_r$、JEST 和 DIPROM 覆盖的错误信息类别数。

同时，如图 5.1 所示，CAFJER 和 $CAFJER_c$ 在实验开始的第一个小时中就成功覆盖了所有的 61 类错误信息，$CAFJER_r$ 在第一个小时平均覆盖了 27.6 类错误信息，JEST 在第一个小时平均覆盖了 12 类错误信息，DIPROM 在第一个小时平均覆盖了 12.2 类错误信息，它们覆盖的错误信息类别数通常在实验的第五个小时到达最大值。CAFJER 对错误信息类别数的覆盖速度也是最快的。

根据实验结果，我们总结得到的 61 类错误信息覆盖了所有实验中生成的错误信息。CAFJER 中类别导向的变异方法能以最快的速度覆盖所有类别的错误信息。通过 CAFJER 和 $CAFJER_c$ 实验数据的对比，说明 MH 算法对错误信息类别覆盖的贡献不大。通过 CAFJER 和 $CAFJER_r$ 实验数据的对比，说明 CAFJER 中类别导向的变异方法有效提高了 CAFJER 对错误信息类别的覆盖率。通过 CAFJER 与 JEST 以及 DIPROM 实验数据的对比，说明 CAFJER 对错误信息类别的覆盖率和覆盖速度远远优于现有方法。

基于以上的数据和分析，我们可以对 RQ4 做出解答，CAFJER 中总结得到的 61 类错误信息覆盖了所有的测试用例，且 CAFJER 中类别导向的变异方法能以最快的速度触发 JavaScript 引擎报错机制所有的错误信息类别。

## 5.7 被确认的缺陷

我们将 CAFJER 连续运行了三个月，在这期间共向开发者提交了 17 个缺陷报告，其中 7 个缺陷报告已被开发者确认。被确认的缺陷 2 个存在 V8 中，1 个存在 JSC 中，4 个存在 SpiderMonkey 中。

下表 5.5 展示了所有 CAFJER 提交后被确认的缺陷。优先级 P1 表示此缺陷最受开发者重视，P5 表示此缺陷的影响最小。在 CAFJER 发现的被开发者确认的缺陷中，三个优先级为 P2，三个优先级为 P3，一个优先级为 P5，说明 CAFJER 发现的缺陷得到了各个 JavaScript 引擎开发者的重视。CAFJER 发现的缺陷中，有 5 个缺陷抛出了不正确的错误类别，有 2 个缺陷抛出了不正确的错误位置，错误类别的差异是 JavaScript 引擎报错机制缺陷发生的主要诱因。本文认为，报错机制产生缺陷的首要原因是 ECMA-262 没有对错误信息进行详细定义。ECMA-262 忽视了错误信息类别的重要性，没有对错误信息类别进行官方的统一规范，各个 JavaScript 引擎开发者也没有就错误信息类别的表述格式达成一致，因此错误信息只能由开发人员按照个人理解编写，不同 JavaScript 引擎中同类错误信息表达格式差异极大。受限于开发人员自身的编程经验，编写各类错误信息的过程中难免会出现失误，给 JavaScript 引擎报错机制留下了缺陷。因此，CAFJER 发现了 5 个错误类别不正确的缺陷。而 ECMA-262 中规定了 JavaScript 正确的语法规则，各 JavaScript 引擎开发人员能根据 ECMA-262 规范确定错误位置。但真实世界 JavaScript 程序中存在众多复杂的情况，开发人员难免有所疏漏，CAFJER 仍发现了 2 个错误位置不正确的缺陷。基于以上发现，本文认为若 ECMA-262 对错误信息类别进行规范统一，使各 JavaScript 引擎开发人员在编写错误说明时有据可依，就能有效减少报错机制中的缺陷数量。

表 5.5　被确认的缺陷

Tab. 5.5　Identified defects

| | ID | 优先级 | 缺陷类别 | 引擎 |
|---|---|---|---|---|
| 1 | 12918 | P3 | 错误类别不正确 | V8 |
| 2 | 13187 | P2 | 错误位置不正确 | V8 |
| 3 | 244018 | P2 | 错误类别不正确 | JSC |
| 4 | 1771690 | P3 | 错误类别不正确 | SpiderMonkey |
| 5 | 1775215 | P3 | 错误类别不正确 | SpiderMonkey |
| 6 | 1780916 | P2 | 错误类别不正确 | SpiderMonkey |
| 7 | 1782849 | P5 | 错误位置不正确 | SpiderMonkey |

基于以上的数据和分析，我们可以对 RQ3 做出解答，CAFJER 能有效地发现真实世界 JavaScript 引擎报错机制中的缺陷。

## 5.8 本章小结

本章详细介绍了本文研究的实验设计，对实验结果进行了分析。

5.1 小节中我们介绍了 CAFJER 运行的实验环境和运用的各个软件及其版本号。5.2 小节中我们提出了 5 个研究问题，对 CAFJER 和 61 类错误信息的有效性进行研究。5.3 小节中我们将 CAFJER 与目前相似的变异方法 JEST 和 DIPROM 进行比较，证明 CAFJER 的有效性。5.4 小节中将 CAFJER 类别导向的变异方法修改为随机变异的方法编写了 CAFJER 的变体 $CAFJER_r$，比较 CAFJER 和 $CAFJER_r$ 的表现证明 CAFJER 中类别导向的变异方法的有效性。5.5 小节中将 CAFJER 选择目标错误信息类别的 MH 方法修改为随机的方法编写了 CAFJER 的变体 $CAFJER_c$，比较 CAFJER 和 $CAFJER_c$ 的表现证明 CAFJER 中利用 MH 算法选择目标错误信息类别的有效性。5.6 小节中检测差分测试中是否发现新的错误信息类别，实验结果表明得到的 61 类错误信息覆盖了所有的测试用例，证明了总结得到的 61 类错误信息的全面性。比较 CAFJER 与其他方法覆盖错误信息的类别数量和覆盖速度，证明 CAFJER 能快速地覆盖最多类别的错误信息。5.7 小节中介绍了 CAFJER 发现的被 JavaScript 引擎开发者确认的缺陷，证明 CAFJER 在真实世界中发现 JavaScript 引擎缺陷的能力。

# 结　论

为检测 JavaScript 引擎报错机制中的缺陷，本文提出了类别导向的模糊测试方法 CAFJER。我们分析 JavaScript 社区中的文档和差分测试输出的错误信息，将 JavaScript 错误信息分为 61 类，并设计了能触发每类错误信息的变异方法。给定一个种子程序，CAFJER 首先对其进行动态分析，为其选择一个目标类别的错误信息。其次，CAFJER 使用类别导向的变异方法生成测试用例。接着，CAFJER 将生成的测试用例输入不同 JavaScript 引擎进行差分测试，比较抛出的错误信息间是否存在差异。最后，CAFJER 自动过滤重复的和无效的测试用例，输出缺陷报告。

我们将目前最先进的 JavaScript 引擎编译机制缺陷的检测方法 JEST 和目前最先进的编译器警告信息缺陷的检测方法 DIPROM 加以改写应用到 JavaScript 引擎报错机制缺陷的检测工作中。与 JEST 和 DIPROM 相比，CAFJER 多输出了 31.38%和 458.30%的可疑测试用例数，多发现了 2.17 倍和 26.0 倍的独特缺陷数。CAFJER 发现 JavaScript 引擎报错机制缺陷的性能显著优于现有方法。真实世界中，CAFJER 三个月内向开发者提交了 17 个缺陷报告，其中 7 个已被确认。

下一步的工作中，我们将会更加深入挖掘各类错误信息的触发条件，搜集更多与各类错误信息相关的 API 函数或复杂语境，丰富变异算法中的 API 模板和函数模板的数量和类型，使 CAFJER 生成更全面更有效的测试用例，实现对各个 JavaScript 引擎报错机制更深层次的检测。我们也计划尝试对错误的 JavaScript 种子程序进行变异，改变种子程序中的错误原因或错误位置，使生成的测试用例触发不同类别的错误信息。现有的各项工作都对正确的种子程序进行变异，对错误种子程序进行变异操作极有潜力构建特殊的语境发现 JavaScript 引擎中的缺陷。此外，我们还计划将对报错机制缺陷的研究工作拓展到其他语言中，对 Java、Python 等流行语言编译器抛出的错误信息进行检测。

# 参 考 文 献

[1] Slashdata.State of the developer nation [EB/OL].(2021-11-22) [2022-12-21]. https://slashdatawebsitecms.s3.amazonaws.com/sample_reports/VZtJWxZw5Q9NDSAQ.pdf

[2] Github. Github Annual Report in 2021 [EB/OL]. (2022-01-27) [2022-12-21]. https://octoverse.github.com/2022/top-programming-languages.

[3] SpiderMonkey. SpiderMonkey bug#1775215 [EB/OL]. (2022-06-20) [2022-12-21]. https://bugzilla.mozilla.org/show_bug.cgi?id=1775215.

[4] Wang J, Chen B, Wei L, et al. Superion: Grammar-aware greybox fuzzing[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 724-735.

[5] Li Y , Xue Y , Chen H , et al.Cerebro: context-aware adaptive fuzzing for effective vulnerability detection[C]//the 2019 27th ACM Joint Meeting. ACM, 2019: 533-544.

[6] Park S , Xu W, Yun I , et al.Fuzzing JavaScript Engines with Aspect-preserving Mutation[C]//2020 IEEE Symposium on Security and Privacy (SP).IEEE,2020: 1629-1642.

[7] Mathis B, Gopinath R, Zeller A. Learning input tokens for effective fuzzing[C]//Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. 2020: 27-37.

[8] He X, Xie X, Li Y, et al. Sofi: Reflection-augmented fuzzing for javascript engines[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021: 2229-2242.

[9] Park J, An S, Youn D, et al. Jest: N+1-version differential testing of both javascript engines and specification[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 13-24.

[10] Ye G, Tang Z, Tan S H, et al. Automated conformance testing for JavaScript engines via deep compiler fuzzing[C]//Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation. 2021: 435-450.

[11] Han H S, Oh D H, Cha S K. Codealchemist: Semantics-Aware code Generation to Find Vulnerabilities in Javascript Engines[C]// The 2019 Annual Network and Distributed System Security Symposium, 2019.

[12] Lee S, Han H S, Cha S K, et al. Montage: A neural network language model-guided javascript engine fuzzer[C]//Proceedings of the 29th USENIX Conference on Security Symposium. 2020: 2613-2630.

[13] Arteca E, Harner S, Pradel M, et al. Nessie: automatically testing JavaScript APIs with asynchronous callbacks[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1494-1505.

[14] Ecma International. ECMA-262 The specification of JavaScript language[S]. https://tc39.es/ecma262/,2021.

[15] Tang Y, Jiang H, Zhou Z, et al. Detecting compiler warning defects via diversity-guided program mutation[J]. IEEE Transactions on Software Engineering, 2021, 48(11): 4411-4432.

[16] Sayed B, Traoré I, Abdelhalim A. If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript[J]. Computers & Security, 2018, 75: 92-117.

[17] 黄子杰,陈军华,高建华.检测 JavaScript 类的内聚耦合 Code Smell[J].软件学报,2021,32(08):2505-2521.DOI:10.13328/j.cnki.jos.006082.

[18] 魏苗,吴毅坚,沈立炜等.基于静态分析的 JavaScript 类型失配缺陷查找[J].计算机科学,2017,44(04):223-228.

[19] 龚伟刚,游伟,李赞等.一种基于动态插桩的 JavaScript 反事实执行方法[J].计算机科学,2017,44(11):22-26+49.

[20] Chen J, Hu W, Hao D, et al. An empirical comparison of compiler testing techniques[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 180-190.

[21] Holler C. Grammar-based interpreter fuzz testing[D]. Saarbrücken, Saarland University, 2011.

[22] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.

[23] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

[24] Zhu X, Wen S, Camtepe S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys (CSUR), 2022, 54(11s): 1-36.

[25] Han H S, Cha S K. Imf: Inferred model-based fuzzer[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2345-2358.

[26] Holler C, Herzig K, Zeller A. Fuzzing with Code Fragments[C]//USENIX Security Symposium. 2012: 445-458.

[27] Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers[C]//Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011: 283-294.

[28] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

[29] Pham V T, Böhme M, Santosa A E, et al. Smart greybox fuzzing[J]. IEEE Transactions on Software Engineering, 2019, 47(9): 1980-1997.

[30] Veggalam S, Rawat S, Haller I, et al. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming[C]//Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21. Springer International Publishing, 2016: 581-601.

[31] Gan S, Zhang C, Qin X, et al. Collafl: Path sensitive fuzzing[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 679-696.

[32] Chen J, Bai Y, Hao D, et al. Learning to prioritize test programs for compiler testing[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017: 700-711.

[33] Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs[J]. ACM Sigplan Notices, 2014, 49(6): 216-226.

[34] Ofenbeck G, Rompf T, Püschel M. RandIR: differential testing for embedded compilers[C]//Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala. 2016: 21-30.

[35] Song S, Hur J, Kim S, et al. R2Z2: detecting rendering regressions in web browsers through differential fuzz testing[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1818-1829.

[36] Sun C, Le V, Su Z. Finding and analyzing compiler warning defects[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 203-213.

[37] Hidayat A. Esprima javascript parser[J]. last accessed on, 2016: 01-30.

[38] Suzuki Y. escodegen: Ecmascript code generator[J]. 2017: 01-14.

[39] Lyu C, Ji S, Zhang C, et al. MOPT: Optimized Mutation Scheduling for Fuzzers[C]//USENIX Security Symposium. 2019: 1949-1966.

[40] Chen Y, Su T, Su Z. Deep differential testing of JVM implementations[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 1257-1268.

[41] Chen Y, Su T, Sun C, et al. Coverage-directed differential testing of JVM implementations[C]//proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2016: 85-99.

[42] Metropolis N, Rosenbluth A W, Rosenbluth M N, et al. Equation of state calculations by fast computing machines[J]. The journal of chemical physics, 1953, 21(6): 1087-1092.

[43] Cha S K, Woo M, Brumley D. Program-adaptive mutational fuzzing[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 725-741.

[44] Rebert A, Cha S K, Avgerinos T, et al. Optimizing seed selection for fuzzing[C]//23rd USENIX Security Symposium (USENIX Security 14). 2014: 861-875.

[45] Woo M, Cha S K, Gottlieb S, et al. Scheduling black-box mutational fuzzing[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013: 511-522.

# 附录 A　61 类错误信息

如下是本文归纳总结得到了 61 类错误信息：

| 编号 | 错误信息 |
|---|---|
| 1 | RangeError:invalid date |
| 2 | RangeError:argument is not a valid code point |
| 3 | RangeError:invalid array length |
| 4 | RangeError:precision is out of range |
| 5 | RangeError: toString() radix argument must be between 2 and 36 |
| 6 | RangeError:repeat count must be less than infinity |
| 7 | RangeError:"x" is not defined |
| 8 | ReferenceError:assignment to undeclared variable "x" |
| 9 | ReferenceError:deprecated caller or arguments usage |
| 10 | ReferenceError:invalid assignment left-hand side |
| 11 | ReferenceError:can't access lexical declaration "x" before initialization |
| 12 | SyntaxError:"0"-prefixed octal literals and octal escape seq are deprecated |
| 13 | SyntaxError: a declaration in the head of a for-of loop can't have an initializer |
| 14 | SyntaxError: for-in loop head declarations may not have initializers |
| 15 | SyntaxError: identifier starts immediately after numeric literal |
| 16 | SyntaxError: "use strict" not allowed in function with non-simple parameter |
| 17 | SyntaxError: "x" is a reserved identifier |
| 18 | SyntaxError: Unexpected token |
| 19 | SyntaxError: applying the "delete" operator to an unqualified name is deprecated |
| 20 | SyntaxError: Function statements require a function name |
| 21 | SyntaxError: invalid regular expression flag "x" |
| 22 | SyntaxError: missing = in const declaration |
| 23 | SyntaxError: illegal character "x" |
| 24 | SyntaxError: missing ) after argument list |
| 25 | SyntaxError: missing ) after condition |
| 26 | SyntaxError: missing : after property id |
| 27 | SyntaxError: missing] after element list |
| 28 | SyntaxError: missing formal parameter |
| 29 | SyntaxError: missing name after . operator |
| 30 | SyntaxError: missing } after function body |
| 31 | SyntaxError: missing } after property list |
| 32 | SyntaxError: redeclaration of formal parameter "x" |
| 33 | SyntaxError: return not in function |
| 34 | SyntaxError: yield expression is only valid in generators |
| 35 | SyntaxError: "" string literal contains an unescaped line break |

| | |
|---|---|
| 36 | SyntaxError: malformed Unicode character escape sequence |
| 37 | SyntaxError: private names aren't valid in this context |
| 38 | SyntaxError: missing catch or finally after try |
| 39 | TypeError:"x" is not iterable |
| 40 | TypeError: can't access property "x" of "y" |
| 41 | TypeError: "x" is not a constructor |
| 42 | TypeError: "x" not a function |
| 43 | TypeError: "x" is read-only |
| 44 | TypeError: More arguments needed |
| 45 | TypeError: reduce of empty array with no initial value |
| 46 | TypeError: can't assign to property "x" on "y": not an object |
| 47 | TypeError: can't define property "x":"obj" is not extensible |
| 48 | TypeError: can't delete non-configurable array element |
| 49 | TypeError: can't redefine non-configurable property "foo" |
| 50 | TypeError: cyclic object value |
| 51 | TypeError: invalid "in" operand "x" |
| 52 | TypeError: invalid "instanceof" operand "x" |
| 53 | TypeError: invalid Array.prototype.sort argument |
| 54 | TypeError: invalid assignment to const "x" |
| 55 | TypeError: property "x" is non-configurable and can't be deleted |
| 56 | TypeError: setting getter-only property "x" |
| 57 | TypeError: iterable for Object.fromEntries should have array-like objects |
| 58 | TypeError: property descriptor's get field is neither undefined nor a function |
| 59 | TypeError: "x" is not a non-null object |
| 60 | URIError: malformed URI sequence |
| 61 | InternalError:too much recursion |

# 攻读硕士学位期间发表学术论文情况

1 面向 JavaScript 引擎报错机制的类别导向模糊测试方法. 卢凌，周志德，任志磊，江贺计算机科学，2023 年。主办单位：国家科技部西南信息中心。（本硕士学位论文第四章以及第五章内容）

# 致　　谢

三年研究生的生涯弹指而过，在这里我要对帮助过我的老师同学们表达感谢。

首先，我特别感谢我的研究生指导老师。在整个研究生期间，我受到了江老师的许多指导和帮助，让我对这个研究领域有了深刻的认知。特别是在论文编写时期，即使我因为疫情不能返校，江老师仍多次审阅我的文章，通过线上会议向我提出了指导性的意见。同时江老师是一个富有人格魅力的人，在老师言传身教下，让我更热爱技术，在面对困难时也不轻言放弃。

其次，我还很感谢周师兄，周师兄帮助我找到了研究的方向，向我提供了许多相关领域的文献，让我能够更好的进行研究工作，让我能够更好的应对困难。接着，我还要感谢各个一路相识的朋友，认识你们是我的幸运，同时还要感谢我的同学和师兄师姐们，感谢他们在整个研究生期间的帮助。

然后，我还要感谢我的母校，大连理工大学，感谢您为我提供了一个学习的机会和地点，研究生三年，让我在这里遇见了一位又一位优秀的教师，一位又一位优秀的同学。

# 大连理工大学学位论文版权使用授权书

本人完全了解学校有关学位论文知识产权的规定，在校攻读学位期间论文工作的知识产权属于大连理工大学，允许论文被查阅和借阅。学校有权保留论文并向国家有关部门或机构送交论文的复印件和电子版，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。

学位论文题目：_____

作 者 签 名 ：_____ 日期：_____年____月日

导 师 签 名 ：_____ 日期：_____年____月日