基于逐脚本特征检测的转译器遍条件执行

Rishipal Singh
Bhatia Google 美国加利福
尼亚州桑尼维尔
rishipal@google.com

arXiv:2603.18049v1 [cs.PL] 17 Mar 2026

摘要

随着 ECMAScript 规范不断演进，工业级 JavaScript 编译器面临着一项挑战：既要支持现代语言语法，又要兼容各种各样的执行环境。传统上，编译器通过在单体流水线中运行转译遍次来解决这一问题，其中转译遍次的选择严格依据目标语言级别。这会造成大量计算浪费，因为编译器会执行开销高昂的抽象语法树（AST）遍历，以降级实际输入源代码中可能并不存在的特性。我们提出了一项编译器改进：通过在整个转译过程中准确跟踪并动态维护编译单元中实际存在的语言特性集合，有条件地执行转译器遍次。该改进已在生产环境中的 Google Closure Compiler 中实现。通过在每个 JavaScript 脚本级别填充并维护一个特征集，它可以动态跳过不必要的降级遍次。我们详细介绍了相关架构防护措施，包括策略性遍次排序，以及对转译后代码进行动态验证以确保特征正确性。在大型生产环境单体代码库上对此改进进行评估后，编译时间显著缩短，同时节省了计算资源和内存用量。

CCS 概念： • **软件及其工程 → 编译器**；**源代码生成**；**通用编程语言**。

关键词： 编译器优化、静态分析、程序分析、源到源编译、转译、条件执行、编译器遍次、特征检测、抽象语法树、抽象语法树、AST 遍历、动态分析、遍次调度器、性能优化、编译时间、代码转换

ACM 参考文献格式：
Rishipal Singh Bhatia. 2026. 基于每个脚本的特征检测有条件地执行转译器遍次. 收录于第 *15* 届 *ACM SIGPLAN* 程序分析前沿国际研讨会论文集（*SOAP '26*）。ACM，美国纽约州纽约市，5 页。

预印本。正在第15届 ACM SIGPLAN 程序分析前沿国际研讨会（SOAP '26）上接受审阅。

1 引言与问题陈述

JavaScript 开发已转向现代语言语法（ECMAScript 2022+），这得益于 ECMAScript 规范的快速演进 [1]。然而，生产环境部署仍受制于大量旧版环境，这些环境要求输出 ECMAScript 5 或 ECMAScript 6 级别的代码。其中包括较旧的浏览器引擎、专用嵌入式运行时以及非浏览器环境。转译弥合了这一差距：它将现代语言结构源到源降级为功能等效但兼容性更好的模式。

在大规模工业级 Web 应用程序中，这种转译步骤是构建环境内一项高频且资源密集型的操作。Google Closure Compiler[2], 为全球一些规模最大的 JavaScript 应用程序提供支持，并将转译作为编译过程的核心阶段。历史上，该编译器采用基于范围的执行模型。在该模型中，转译遍次的选择仅由输入语言版本与所请求的输出语言版本之间的差距决定。例如，如果编译器的任务是生成 ES5 输出，它会自动将从当前 ECMAScript 版本（ESNext）一直到 ES5 的整套降级遍次加入队列。

这种方法的根本缺陷在于，它缺乏针对源代码实际内容的细粒度处理。大多数编译单元并未使用现代语言特性的全部范围。例如，一个脚本可能使用 ES2020 的可选链特性，但不包含任何 ES2015 类。尽管如此，基于范围的模型仍会迫使编译器执行一组类降级遍次，从而承担遍次设置和抽象语法树遍历的全部开销。在由数百万个文件组成的大型单体仓库中，这些累积的冗余遍历会造成显著瓶颈，增加构建延迟，并消耗大量不必要的 CPU 周期。

2 背景：TranspilerPipeline

要理解冗余遍历为何代价如此高昂，有必要考察 Google Closure Compiler 转译器的架构。

2.1 增量降级与遍次排序

Google Closure Compiler 将转译拆分为一系列分层、原子的转换遍次，这些遍次依次作用于脚本，一次处理一个脚本。每个遍次负责将其处理的脚本中的特定特性从较新的语言级别降级到较旧的语言级别。例如：

- **RewriteOptionalChainingOperatorpass：**将 ECMAScript 2020 的可选链语法（?.）运算符重写为条件语法（?:）。
- **Es7RewriteExponentialOperatorpass：**将 ECMAScript 7 的指数运算符语法重写为 ECMAScript 5 的 Math.pow 运算。
- **Es6RewriteRestAndSpread 遍次：**将 ECMAScript 6 的剩余与展开语法重写为包含填充代码的 ECMAScript 5 语法。

这些转换会根据语言规范年份，严格按照特性的时间顺序从新到旧依次执行。例如：

- **级联转换：** 高层级特性（如"异步/等待"，这是 ECMAScript 2017 的一项特性）会先降级为 ECMAScript 2015（ES6）生成器（function*、yield）。
- **逐步降级：** 只有在异步转换完成后，后续的转换流程才能将新生成的生成器降级为兼容 ES5 的状态机。

2.2 转译器转换的成本

在旧版工作流中，编译器识别出输出语言级别后，会实例化并运行达到该目标级别所需的每个转换流程。每个启用的转换流程都包括：

1. **上下文初始化：** 设置该转换流程所需的内部状态。
2. **完整 AST 遍历：** 全面 " 遍历 " 整个树，查找相关节点。 3. **节点重写：**如果发现某项特性，则执行实际转换。 4. **清理与更新：**更新编译器状态，以记录重写已完成，并确保转换流程执行后抽象语法树的完整性。

主要的低效之处在于，无论步骤 3（实际工作）是否发生，范围内的每个遍次都会执行步骤 1、2 和 4。

2.3 语言级别的向下单调性

转译器的架构完整性依赖于向下单调性的假设。ECMAScript 规范按年份（语言级别 L_N）组织特性，而转译器的实现则采用一组细粒度的、用于降低各项语言特性的原子遍历 $P_{f,N}$ $f \in L_N$。

形式上，每个遍次 $P_{f,N}$ 都充当一个转换T： $f \to \{f' \in L_k \mid k < N\}$。随后的一系列遍次会继续迭代地降低级别，直至达到最低目标级别。这种结构依赖于向下单调性假设：对于任何以特性 f为目标的遍次 P_f，流水线中后续的遍次 $P_{f'}$ 都不能将其重新引入抽象语法树 f。

下一节介绍的解决方案增强了转译过程中的特性跟踪，使编译器能够将这一单调性假设转化为编译期间强制执行的不变量。它利用转换的这种细粒度性以及增强后的特性跟踪，实现了在单个语言级别内跳过特定子集遍次的能力；这一点在范围庞大的 ES6→ ES5 降级链中尤为显著。通过准确检测脚本级的特性是否存在，即使同一规范年份的其他特性也存在，我们仍可剪除这些原子遍历。

3 改进：选择性转译

本文提出的改进在 Google Closure Compiler 中通过一种"选择性转译"模型实现 [4, 9]。该方法引入了一个分析"门"，通过确保仅当特定脚本的抽象语法树中存在相应语言特性时才调用降级遍次，从而避免编译器执行冗余工作。

3.1 记录输入特性

首先，编译器会识别并存储解析器在输入脚本中遇到的特性集合。Google Closure Compiler 中的解析器会识别给定输入脚本中特定语言结构的用法，并将其记录为该脚本所属的特性。记录操作会将这些信息作为属性添加到每个 SCRIPT 节点上；该节点是编译器内部文件抽象语法树表示中的顶层节点。为避免增加内存压力，解析器仅在现有抽象语法树结构上操作，不会创建临时对象或节点属性以外的次级数据结构。

我们将编译器识别的所有语言特性集合形式化为 F。对于脚本 S，我们将脚本特性集合 $\Phi(S) \subseteq F$ 定义为：

$$\Phi(S) = \{f \in F \mid \exists n \in S : \text{isFeatureNode}(n, f)\}$$

3.2 将转译器遍次映射到其特性

每个转译器遍次都会明确界定其预期要降级处理的特性集合 $\psi(P) \to f$，并将该 API 暴露给编译器。这样，编译器便可获取转译器遍次与必须触发该遍次的相关特性之间的映射。

3.3 准确跟踪特征集

选择性省略中的一项关键技术挑战，是确保特征集在整个转译过程中始终准确。如果根据过时的特征集跳过了针对某个脚本运行的转译器遍次，或者某个遍次引入了未被跟踪的特性（即未能将新特性注册到脚本的特征集中），编译器可能无法降级处理所需的语法。

3.3.1 合成特征传播。特性在概念上分为两类：源特征（在解析输入源代码时检测到）和合成特性（在降级过程中由转译器遍次引入）。

特征集是一个动态属性。因此，当某个遍次P_i将高级特性 f_{source} 降级时（例如，将异步函数转换为生成器），它必须以编程方式通知编译器自己引入了哪些新特性。随后，编译器会执行内存更新： $\Phi(S) = (\Phi(S) \setminus \{f_{source}\}) \cup \{f_{synthetic}\}$。

此更新会在遍次完成后立即进行，确保调度器在评估下一个遍次的跳过谓词 P_{i+1} 之前，掌握每个脚本的最新特征集。如果没有这种传播，针对合成特性（例如生成器）的遍次可能会因为原始源代码中不包含这些特性而被错误地跳过，最终导致转译结束时出现编译时错误。

3.3.2 已转译特征集。 为防止静默失败，编译器会在整个转译过程中维护一个全局的已转译特征集（T）。此集合记录每个已被相应转译过程转译掉的特性f。当某个转译过程 P_i 完成对编译单元中所有脚本的处理后，特性 f会被添加到 T。

系统强制执行一项严格的不变量：对于编译单元中的任意脚本 S， T中的任何特性都不得存在于 $\Phi(S)$ 中。如果某个转译过程 P_j（其中 $j > i$）试图引入一个已存在于 T 中的特性f，编译器就会标记为编译时违规。这确保了转译的级联过程严格单向，并且在特定的降级过程结束后，特性绝不会被重新引入。

3.3.3 转译器遍控制的特征集。特征集作为抽象语法树中 SCRIPT 节点的持久属性存储。操作语法树的转译器遍必须维护此集合。为尽可能降低开销，在生产模式下，编译器不会在每次重写遍次后单独重新扫描抽象语法树以更新特征集。相反，它依靠转译遍次“拥有”与各自特性相关的特征集属性，而编译器只在转译结束时验证特征集的准确性。这种“所有权”模型既能确保高性能，也能满足选择性省略所需的准确性。

3.4 选择性执行逻辑

在编译器主循环执行期间，若某个由 P 处理的特性既不受目标级别支持，又存在于脚本的特征集中，则满足对脚本 S 执行转译器遍次 P 的条件。

形式化表示如下：

$$\exists f \in \psi(P) : (f \notin \text{目标级别特性}) \wedge (f \in \Phi(S))$$

我们在图 1中展示了这一门控机制及其在编译器执行循环中的集成方式。流程图详细说明了编译器如何针对每个脚本单独评估是否存在目标特性 f_{source}，从而动态省略不需要该转换通道的脚本所对应的整个转换通道，同时确保合成特性能正确传递给后续遍次。

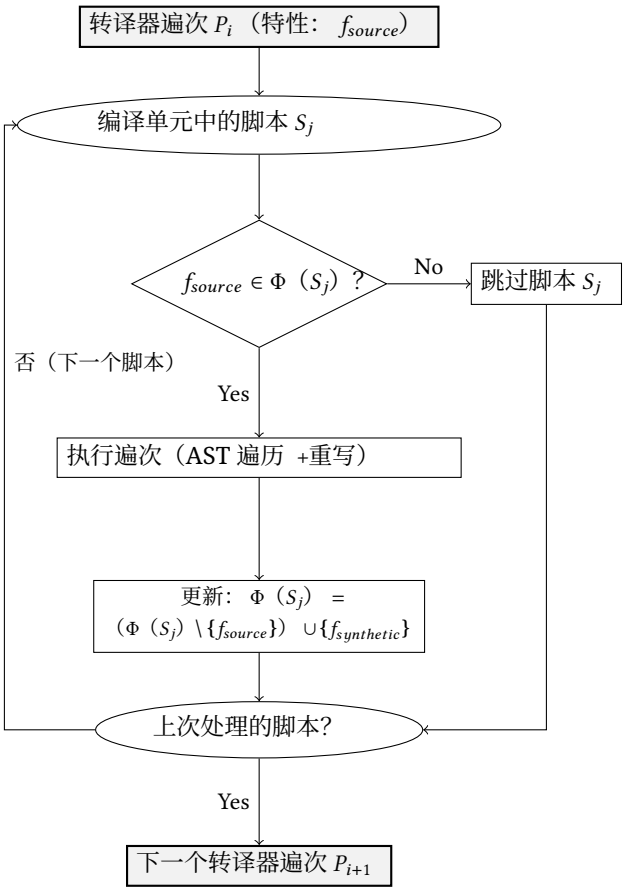

图1. 选择性转译逻辑：编译器在每个转译器遍次中遍历脚本，根据各自的特性集 $\Phi(S)$，跳过缺少目标特征 f_{source} 的脚本。

4 架构防护措施

在编译器中动态剪除转译器遍次会带来“特性泄漏”的风险。我们实施了四层防护措施，以确保正确性：

- **策略性遍排序：**如第 3 节所述，遍次按逆时间顺序运行。这可确保降级代码不会重新引入来自更高层级的特性。
- **动态特性跟踪：**脚本节点的 Node. FEATURE_SET 属性会实时更新。如果某项转换引入了需要后续降级遍处理的新特性，该特性就会被添加到集合中。
- **AST 验证：**开发期间，每个遍次之后都会注入一个 ASTValidator，用于将当前抽象语法树与“已转译移除”集合进行交叉核对。如果某个遍次试图引入已标记为移除的特性，编译器就会报错。此外，如果某个遍次试图引入允许的特性，却未明确向编译器报告这一变更，也会被捕获为编译器错误。
- **转译后验证：**在转译阶段结束时，会运行生产级后置流程 TranspileCheckUnsupportedFeaturesRemoved，以确保没有不受支持的特性遗漏。

5 评估结果

5.1 评估方法

为了对该解决方案进行可信且可复现的评估，我们从内部单体代码库中选取了具有代表性的大规模生产应用样本进行评估。这些应用涵盖多个领域，包括高流量搜索服务、地理空间制图工具以及复杂的 Web 生产力套件。多样性确保评估结果涵盖各种编码模式，从旧版 ES5 到现代 ES2022 JavaScript。

评估在标准化的分布式构建环境中进行，使用基于云的工作节点。每个工作节点都是一台配置规格统一的 Linux 虚拟机（例如，配备 32 核高性能 CPU 和 64GB 内存）。为减少外部干扰，每项基准测试均执行 3 次，并记录平均值。

5.2 评估结果

- **编译时间改进：**对于通过 Google Closure Compiler 编译的部分大型生产环境 Web 应用，总转译时间减少了 50%。在通过 Google Closure Compiler 编译的所有生产环境 Web 应用中，我们观察到，编译的特定阶段（包含转译）的平均编译时间减少了 12%。 [3]。
- **节省的计算量：**这项解决方案改进了编译流程中的一个阶段（其中包含转译步骤），该阶段每天执行约 900,000 次，每次用于编译 Google 的生产环境 Web 应用程序。
- **冗余遍次分析：**对于一个典型的、目标输出级别为 ES5 的生产环境 Web 应用，旧模型中有 30%–40% 的转译遍次会进行冗余遍历。
- **可扩展性：**随着新的特性不断被引入输入 JavaScript 语言（例如，假设存在 ES2027 特性集合），即使输入源代码中很少使用现代语言特性，旧模型仍会继续执行全部转译遍次（ES2027 → ES5）。而在新模型中，编译成本与未使用的新特性脱钩。

表 1. 选择性转译对生产工作负载的影响

指标	结果
转译时间（大型应用）	减少 50%
平均编译阶段延迟	减少 12%
跳过的冗余转译遍历	30% – 40%
每日生产执行次数	≈900,000

5.3 在开源项目中的实现

选择性转译模型已在 Google Closure Compiler 的开源发行版中提供，并默认启用。核心门控逻辑封装在 TranspilationPasses.java 工具类的 doesScriptHaveAnyOfTheseFeatures 方法中。

出于基准测试或调试目的，可以通过修改上述方法使其返回恒定值 true 来切换此优化。这样实际上会将编译器恢复为传统的基于范围的执行模型，强制在每个脚本上执行范围内所有可用的转译器遍次，而不考虑脚本内容。在实践中，构建复杂 Web 项目时禁用此特性会导致构建延迟显著增加，因为冗余遍历被重新引入。

5.4 正确性验证

此优化的一项关键要求是保持语义透明；跳过冗余遍次不得导致编译器生成不同的输出代码。我们采用了多种测试方法，以验证选择性转译模型能够产生与传统的基于范围的模型相同的结果：

- **差分测试：** 对于部分选定的内部应用，我们分别使用传统转译模型和选择性转译模型编译源代码。随后，我们对最终的压缩后的生产包进行了逐位比较，以确保二者完全一致。

- **代码大小测试：** 我们运行了代码大小回归测试套件，以验证跳过冗余处理阶段（选择性转译）不会影响 Google Closure Compiler 生成的最终输出代码大小。
- **编译器测试套件：** 我们在启用这项改进的情况下，运行了现有的 Google Closure 编译器测试套件（包含数千个单元测试）。

6 与相关工作的比较

Babel 和 SWC： 最受欢迎的转译器 [5, 6] 采用基于插件的架构，用户需要显式启用转换。尽管 SWC [6] 凭借基于 Rust 的实现提供了极快的速度，但对于已配置的预设，它通常仍采用基于范围的执行方式。选择性转译则不同，它通过深入检查抽象语法树，而非依赖静态配置，自动完成选择。

LLVM PassManager： LLVM [7] 采用 PassManager，可根据分析结果跳过某些优化遍次。我们将这一概念扩展到源到源转换场景中的高级语言特性。

即时（JIT）编译器： 像 V8 [8] 这样的引擎会通过动态性能分析来优化“热点”代码路径。相比之下，选择性转译是一种静态优化，能够改善整个代码库的构建时体验。

7 未来工作

选择性转译的成功表明，编译器设计中存在一种更广泛的模式。可能的扩展包括：

- **选择性优化：** 对高开销的优化遍次采用类似的基于特性的门控后，如果抽象语法树中不存在这些优化所针对的语言结构，我们就可以跳过高复杂度优化。**注意事项：** 与转译不同，每个转译器遍次都有明确的按时间顺序划分的规范边界；要更新“特征集”，使其准确反映优化遍次结束时的抽象语法树状态，则复杂得多。
- **细粒度作用域分析：** 在函数级而非脚本级执行特性扫描，使遍次能够跳过抽象语法树中的特定子树。**权衡：** 虽然这能减少大型“单体”文件中的更多工作，但存储和查询逐作用域特征的开销会显著增加内存消耗和复杂度。对于许多中等规模的文件，进行更细粒度分析的成本可能会超过降级遍次节省的时间。

8 结论

本文所述的选择性转译模型，代表了工业级编译器管理不断扩展的 ECMAScript 规范方式的一次重大架构转变。通过从僵化的基于范围的执行模型转向动态的特性门控方法，我们成功地将转译的计算成本与未使用的现代语言特性是否存在解耦。我们在 Google Closure Compiler 中的实现表明，轻量级静态分析结合可靠的特性跟踪和保护措施，能够在不损害输出语义完整性的前提下，大幅节省构建时间资源。

观测到的编译延迟和 CPU 周期减少，证实在脚本级维护动态特征集所带来的开销，与跳过冗余抽象语法树遍历所节省的时间相比微不足道。此外，这种新模型为编译器的持续发展提供了一条可行路径；随着 TC39 委员会 [10]不断引入新语法，编译器的性能将与开发者实际使用的特性数量相匹配，而不是与标准中定义的特性总数相匹配。

最终，本文所述的方法为现代编译器在语言快速演进的时代实现高性能源到源转换提供了蓝图。通过严格执行向下单调性，并借助架构 safeguards 验证特征集，我们确保编译器始终兼具速度与正确性，在为开发者提供即时价值的同时，能够扩展以满足未来 Web 开发的需求。

参考资料

[1] ECMA 国际。（2024年）。ECMA-262：ECMAScript 2024 语言规范。取自 https://tc39.es/ecma262/ [2] Google。（2024年）。Closure 编译器。GitHub 仓库。 https://github.com/google/closure-compiler [3] Bhatia, Rishipal Singh，Perfys：避免不必要的转译。2024年。内部文档。（引用时称为“perfys”）。[4] Bhatia, Rishipal Singh。避免不必要的转译。2023年。内部设计文档。 [5] Babel 团队。2026年。Babel：面向下一代 JavaScript 的编译器。 https://babeljs.io/ [6] SWC 团队。2026年。SWC：基于 Rust 的 Web 平台。 https://swc.rs/ [7] LLVM 项目。2026年。LLVM 编译器基础设施。 https://llvm.org/ [8] V8 项目。2026年。V8 JavaScript 引擎。 https://v8.dev/[9] Bhatia, Rishipal Singh 和 Harker, Laura，《基于逐脚本特征检测的编译器遍次条件执行》，技术披露共享平台，（2026年1月27日） https://www.tdcommons.org/dpubs_series/TC..E 9223 [10] 39 2026 cma International's TC39 https://tc39.es/