链滴

【翻译】Google 在构建静态代码分析工具方面的经验教训

作者：nanolikeyou
原文链接：https://ld246.com/article/1530854065192
来源网站：链滴

原文链接 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fstorage.googleapis
com%2Fpub-tools-public-publication-data%2Fpdf%2F3198e114c4b70702b27e6d88de2c927
4c9ac4c0.pdf" target="_blank" rel="nofollow ugc">https://storage.googleapis.com/pub-tools
public-publication-data/pdf/3198e114c4b70702b27e6d88de2c92734c9ac4c0.pdf
软件 bug 耗费开发者和软件公司大量的时间和金钱。 以 2014 年为例，被广泛使用的 SSL 协议
现中的一个（“goto fail”）bug 导致可接受无效的 SSL 证书，另外一个与日期格式化相关的 bug
致 Twitter 的大范围服务中断。此类错误通常可以被静态分析检测，其实事实上在阅读代码或文档阶
均可以快速识别，可以最终现实是情况仍然在生产环境实施发生。
之前的工作已经完善报道将 bug 检测工具应用于软件开发的经验。但虽然开发人员使用静态分
工具方面有如此多的成功案例，仍有以下原因导致工程师并不总情愿使用静态分析工具或主动忽略工
产生的告警信息：

未合理整合。工具未集成到开发人员的工作流程中或者是程序运行时间太长;

无效的告警。告警信息可行性性差;

不值得信赖。用户因为误报而不再信任结果;

缺陷实际利用场景不清晰。报告的 bug 在理论上是可行的，但缺陷在实际利用场景下并不清晰;<
p>

修复成本过高。修复已检测到的代码缺陷的成本太高或有其他方面的风险;

告警不易理解。使用者并不了解告警信息的的具体信息和原理。

接下来本文描述了我们如何吸取 Google 在先前使用 FindBug 进行 java 语言分析方面以及学术
献中的得到的经验和教训，最终在 Google 公司成功构建了软件工程师日常使用的静态分析基础设施
构。借助吸收于工程师的意见建议，Google 的工具可以在有问题的代码被合入到公司级别的代码仓
之前检测到每天工程师所修复的数千个问题。
在工具作用范围方面，我们专注于将静态分析融入为 Google 核心开发流程的一部分，并服务于
部分 Google 开发人员。许多静态代码分析工具在部署于 google 的 20 亿行代码级别下将相形见绌
因此大规模场景下运行复杂分析的技术的优先级并不高。
当然必须要考量到 Google 外部开发人员在专业领域（例如航空航天和医学设备领域）工作的可
会使用特定的静态分析工具和工作流程。同样开发项目涉及特定类型（例如内核代码和设备驱动程序
的开发人员可能需要进行特定的分析方法。静态分析方面已经有很多卓越的工作成果，我们并不认为
们所反馈的经验和心得是独一无二的，但我们坚信整理和分享我们在提高 google 的代码质量和改善
发体验方面的工作是有所裨益的。
术语定义。我们使用以下术语定义：分析工具对源代码运行一个或多种“检查器”，并识别出可
呈现出软件故障的“缺陷”。如果开发人员在看到问题后没有采取积极行动，如果开发人员遇到识别
的缺陷并未采取适当的修复方式，我们将其视为“实际误报”。如果静态分析并未准确的识别报告到
项缺陷，但开发人员主动采取措施修改此处代码以提高可读性和可维护性，那么这就不算是有效的“
报”。如果一个分析报告了实际的代码错误，但是开发人员因未理解这处代码问题因而没有采取任何
动，就视为这是一个“实际误报”。我们使用这种概念区别强调了研发角度的重要性。开发人员而不
工具作者对感知并直接影响到工具的误报率。
<h3 id="谷歌如何编译构建软件">谷歌如何编译构建软件</h3>
下文我们将概述 Google 软件开发流程的关键点。在 Google，几乎所有开发工具（开发环境除
）都是集中化和标准化的。该基础架构的许多部分都是由内部团队拥有的 Scratch 构建的，保留了具

实验性质的灵活性。
源代码控制和代码所有权。Google 已开发并使用单源的代码管理系统。并试验单分支存储（几）所有的 Google 专有代码。开发人员使用限制分支的“基于 trunk”的开发方式，通常以版本发布不是特性进行划分。任何工程师经代码所有者的批准都可以更改任何代码。代码所有权基于路径划分;录的所有者对子目录拥有同样权限。
系统构建。 Google 代码库中的所有代码都经一个编译环节独立的 Bazel 版本进行编译，也就是，所有输入都必须在源代码控制中显式声明和存储以便构建环节的易于分发性和并行化处理。在 Google 的构建系统中 Java 规则依赖于 JDK 和经源码管理的 Java 编译器，并且可以通过快捷引入新版本为所有使用者更新这些二进制文件。构建通常来自源代码（通过 head），几乎很少有二进制组件签分支。因为所有开发人员都使用相同的构建系统，所以任何代码都可以编译成功而不报错。
分析工具。 Google 使用的静态分析工具通常并不复杂。Google 基础架构不支持在这样级别的统上运行过程间或基于程序的完整性分析，也没有大规模使用高级静态分析技术（例如 separation logic 技术）。即便是简单的检查器也需要分析基础架构来支持对工作流程进行集成。已部署作为一般开流程的分析器类型包括：

样式检查器（例如 Checkstyle，Pylint，和 Golint）;

可扩展 bug 查找编译器（例如 Error Prone，ClangTidy， Clang Thread SafetyAnalysis， Goet，和 CheckerFramework），包括但不限于抽象语法树模式匹配工具，基于类型的检查器和检测调用变量的分析器。

调用生产服务的分析器（例如检查代码注释中提到的员工是否仍然在 Google 上工作））;

检查构建输出的属性（例如产出二进制文件的大小）。

Google 的 C++ linter 可以捕获“goto fail”漏洞，其检查 if 语句是否后面有括号。基于模式配的检查器会识别出日期格式化错误，那么导致 Twitter 宕机的代码就不会在 Google 编译通过。谷开发人员也使用动态分析工具（如 AddressSanitizer）来寻找缓冲区漏洞、使用 ThreadSanitizerto找数据竞争问题。这些工具在测试环节，有时甚至是有生产流量的环境中运行。
集成开发环境（IDE）。静态分析问题在开发过程早期切入点是集成在 IDE 中。但是 Google 开人员使用各种各样的编辑器，因此在调用构建工具之前很难一致地检测所有开发人员的错误。虽然谷确实使用与流行的内部 IDE 集成的分析，但要求具有可分析的特定 IDE 前途漫漫，路险而艰。
测试。几乎所有 Google 代码都包含相应的测试环节，从单元测试到大规模集成测试。测试活动系统构建中首先需融合的理念，就如编译环节一样都是独立和分布式的。对于大多数项目，开发人员写并维护代码的测试用例;项目通常没有单独的测试或 QA 组。
Google 的持续构建和测试系统会在每次代码提交时运行测试，会及时反馈由于开发人员的代码改导致构建失败或测试用例不通过。它还支持在提交之前测试变更处以避免破坏项目依赖关系。
Code review。每次提交到 Google 的代码都会首先通过 code review。虽然任何开发人员都可对 Google 任何代码处进行更改，但代码的所有者必须在提交合入之前 review 才批准更改。此外，使是代码所有者也必须在提交变更之前 review 其代码。代码审查通过一个与其他开发基础设施紧密成的集中式、基于 Web 的工具进行。静态分析结果可在代码审查中展现。
代码发布。谷歌团队频繁发版，大部分的发布验证和部署过程都是通过“push on green”的方自动完成的，这意味着难以依靠进行辛劳的手动发布验证过程。如果 Google 工程师在生产环境中发错误，同必须中断服务相比而言可以以相对较低的成本回退新版本并将其部署到生产服务器。
<h3 id="从FindBugs学习到的">从 FindBugs 学习到的</h3>
从 2008 年到 2010 年期间早期摸索研究阶段，谷歌的静态分析技术专注于使用 FindBug 进行 Java 分析：由马里兰大学的 William Pugh 和宾夕法尼亚州约克学院的 DavidHovemeyer 创建的独立具。其原理为分析已编译的 Java 类文件并提取可以导致 bug 的代码结构模型。截至 2018 年 1 月，FindBugs 在 google 仅作为极少数工程师使用的命令行工具。一个名为“BugBot”的小型 Google

队与原作者 Pugh 合作，有三次大的尝试试图将 FindBugs 集成到 Google 开发流程中。
我们通过尝试学到了以下几点：
尝试 1：Bug dashboard。最初在 2006 年，FindBugs 被整合为一个集中性的工具来每晚扫描个谷歌代码库，将生成的结果入库方便工程师通过 dashboard 进行检视。尽管 FindBugs 在谷歌的 Java 代码库中发现了数百个错误，但是该 dashboard 效果微乎其微，因为错误信息 dashboard 脱离日常开发流程，而且同现有的其他静态分析结果不能有机结合。
尝试 2：集中改进 bug。
接下来 BugBot 团队开始手动分类每晚查找到的新问题，识别处理相对重要的 bug 报告。 2009 年 5 月，数百名 Google 工程师参加了全公司范围的“Fix it”周活动，聚焦解决 FindBugs 的告警题.总共审查了 3954 个告警信息（占总数 9473 的 42%），但实际上只修复了 16%（640 个）。实 44%的报告结果（1746 个）已经提交了 bug 反馈跟踪。 尽管 Fixit 活动证实 FindBugs 发现的许问题都是现实存在的代码缺陷，但很大一部分并没有重要到需要实际采取修复措施的地步。人工手动类问题、提交 bug 报告在大规模环境下是难以持续进行的。
尝试 3：集成于代码审查。接下来 BugBot 团队集成实现了这样的系统：当 review 人员得到待 r view 通知时，FindBugs 将会自动运行并将扫描结果作为代码审查的注释展示出来，以上代码 review 团队已经针对编码规范/风格问题已经实现完成。谷歌开发人员可以忽略误报，并针对 FindBugs 的果可信度进行筛选。该工具进一步尝试仅显示新的 FindBugs 告警，但有时会因为错误的分类将其视新问题。随着代码审查工具在 2011 年被替换，这种集成随着寿终正寝，原因有两个：实际工作中的误报率导致开发人员对工具失去信心，开发人员自由定制进行过滤导致各方对分析结果观点不一致。
<h3 id="纳入编译流程">纳入编译流程</h3>
在 FindBugs 的实验同时期，Google 的 C++ 开发流程通过向 Clang 编译器添加新的检查规则得到不断进步。Clang 团队实现了的新的编译器检查器，包括修复建议信息，使用 ClangMR 以分布方法在整个 Google 代码库上运行经过更新的编译器来优化检查，并编码实现修复了代码库中的存量ug 问题。一旦代码库已标记被修复存量问题，Clang 团队就会应用新检查器将新问题标记为编译器误（而不是告警，Clang 团队发现谷歌开发人员会忽略告警）来中止构建，对此必须加以解决才能通。Clang 团队通过这一策略非常成功地改进了代码库质量。
我们遵循这个思路并在 javac 编译器基础之上构建了一个简单易用的基于模式分析的 Java 静态析工具，名为 Error Prone。推出的第一个检查规则名为 PreconditionsCheckNotNull，用于检测程运行之初是否方法检测入参为空，比如 checkNotNull（“uid 为 null”，uid）而不是 checkNotNul（uid，“uid was null”）。
为了在不破坏任何连续构建的情况下启动 PreconditionsCheckNotNull 这样的检查器，Error Pr ne 团队使用它对基于 javac 的 MapReduce 程序对整个代码库运行此类检查，类比 ClangMR，使用 lumeJava 构建称之为 JavacFlume。JavacFlume 会生成一系列的修复建议，比对其中的不同，然后用这些到整个代码库进行修复。Error Prone 团队使用内部工具 Rosie，将大规模代码变更拆分为小变更处，每个改变只会影响到单个项目并测试这些变更处，并将它们发送给相应的团队进行代码审查团队仅审查适用于其代码的修复方案，并且仅在批准通过它们进行合入，Rosie 才会提交实际更改。终所以存量问题的修复和变更都得到通过，已有缺陷均得到处理。团队正式开启了编译器错误的方式
当我们对收到这些补丁的开发人员进行调查反馈时，其中 57%收到合入代码的 fix 方案的人的乐得到此类信息，，41%的人持中立态度。只有 2%的人反应较为消极会说：“这样仅仅会加重我的工量”
<h4 id="编译器检查的价值">编译器检查的价值</h4>
编译错误是在开发流程的早期显示并已集成到开发人员工作流程中。我们发现扩展编译检查器有地提高了 Google 的代码质量。因为 Error Prone 中的检查是内部针对 javac 的抽象语法树而不是字码（与 FindBugs 不同）编写的，所以团队外部的开发者可以相对容易地进行检查。利用这些外部贡对于提高 Error Prone 的整体影响力至关重要。截至 2018 年 1 月，共有 162 名作者提供了 733 项查器。
<h4 id="越早报告问题越好">越早报告问题越好</h4>
Google 的集中构建系统会记录所有构建过程和构建结果，因此我们确保所有的用户在指定时间口可看到其中的错误消息。我们向最近遇到编译器错误的开发人员和已收到针对同一问题修复建议进修复的开发人员发送了一项调查反馈。谷歌开发者认为在编译时标记出来的问题（与合入代码时期的丁相对）会捕获更重要的错误;例如，调查参与者认为 74%的问题在编译时被标记为“切实问题”，在合入代码中发现的问题只有 21%。此外，调查参与者认为在编译时发现的问题中有 6%（在合入代

阶段中为 0%）是“重要级别”。这个结果可以通过“幸存者偏差效应”来解释; 也就是说在代码提时错误很可能是由更高昂的手段（如测试和代码审查）捕获的。将尽可能多的检查前置到编译器中是种避免这些成本花销的可靠办法。
<h4 id="编译器检查的标准">编译器检查的标准</h4>
为了规模推广我们的工作，因为中断编译将是一个较大的动作，所以我们已经定义了在编译器中用检查的标准，设置为严格高标注模式。Google 上的编译器检查应当简单易读、可操作的且易于修（尽量实现的错误消息提示应该包括可通用实现的修复建议); 没有产生有效误报（分析动作不应中断建实际上正确的无误代码）; 并仅报告反馈真实的 bug 而非风格或编码规范方面的问题。 衡量满足这标准的分析器的主要目标不仅仅是简单检测问题，而是在整个代码库中自动修复这些编译器错误。 但这些标准也限制了 Error Prone 团队在编译代码时启用的检查范围; 许多问题不能被准确检测或通用问题修复环节仍然是摆在我们面前的问题。
<h3 id="在代码review阶段展示告警信息">在代码 review 阶段展示告警信息</h3>
一旦 Error Prone 团队构建实现了在编译时检测问题所需的基础设施架构，业已证明该方法切实效，我们希望查找出更多有高影响因子的 bug，bug 不局限于我们做的编译器错误检查和为 Java 和 ++ 以外的语言提供分析结果。静态分析结果的第二个集成切入点是 Google 的代码审查工具-Critique 静态分析结果通过使用 Tricorder 在 Google 的程序分析平台的 Critique 中显示。截至 2018 年 1 ，Google 的 C ++ 和 Java 版本的编译器错误均清零，所有分析结果都显示在编译器错误或在代码查阶段。
<h4 id="代码审查检查的标准">代码审查检查的标准</h4>
与编译时检查不同，代码审查期间显示的分析结果允许囊括达到 10%的有效误报率。在代码审期间所期望的反馈并不总是完美无缺的，并且开发者在实际采用之前需评估相应的修复建议。 Google 在代码审计阶段的检查器应符合以下几个标准：
易于理解。对于工程师来说是明白无误易于理解的;
方案可行，易于修复。修复程序可能需要比编译器检查阶段花费更多的时间、思考和工夫，检查果应包括有关如何界定问题的指导内容;
不到 10%的有效误报率。开发人员应该感受到检查器在至少 90%的情况下均找到实际 bug 缺陷
对代码质量产生重大影响。发现的问题可能不会影响程序正确运行，但开发人员应该认真对待它并选择修复它们。
有些问题严重到足以在编译器中标记出来，致力于但降低或开发自动修复方案并非可行。例如有修复问题可能需要对代码进行重构。将这些检查结果作为编译器错误启用将需要手动清理现有的实现这在 Google 这样大规模的代码库上是不可行的。分析工具在代码审查中显示这些检查可避免引入新题，允许开发人员决定是否采取措施进行恰当的修复。代码审查也是报告相对不太重要的问题（如规问题或简化优化代码）的良好时机。根据我们的经验，在编译阶段出报告对开发人员总是难以欣然接的，并且使得快速迭代和调试变得更加困难;举例来说，一处针对代码路径不可达的检测器可能会阻碍处用于调试的代码块。但在代码审查时，开发人员正在细心准备完成他们的代码;他们正处在虚心接受心态，更容易接受可读性和风格细节的问题。
<h4 id="Tricorder">Tricorder</h4>
Tricorder 设计理念旨在易于扩展，并支持包括静态和动态分析工具的众多不同类型的程序分析具。我们在 Tricorder 中展示了一些无法作为编译器错误启用的 Error Prone 检查器。Error Prone 创造了一套新的 C++ 分析组件，它与 Tricorder 集成称之为 ClangTidy。Tricorder 分析器的报告支超过 30 种语言的结果，支持简单的语法分析如样式检查器，利用 Java，JavaScript 和 C ++ 的编译信息，并且可以直接与生产数据集成（例如关于当前正在运行的任务作业）。Tricorder 持续在 Google 取得成功是因为它是支持分析器编写者的一个生态平台的插件模型，并在代码审查过程中高亮显示行的修复方案，并提供反馈渠道以改进分析器并确保分析器开发人员对正向反馈采取措施。
使用户能够做出贡献。截至 2018 年 1 月，Tricorder 包括 146 个分析器，其中 125 个来自 Tricrder 团队之外，7 个插件系统用于数百个额外检查（例如 ErrorProne 和 ClangTidy，它们是包括在个分析插件系统中的两个）。
审阅者参与进来提供修复建议。
Tricorder 检查器可以提供为代码审查工具提供代码 review 人员和开发者可见的合理修复建议审阅者可以通过单击分析结果上的“请修复”按钮来要求开发者修复缺陷代码。直到他们的所有评论手动和自动发现的）都得到解决之前，review 者通常都不会批准合入代码变更。
迭代来自用户的反馈。除了“请修复”按钮，Tricorder 还提供了一个“无用”按钮，评论者或议者可以点击表示他们不认同分析发现的结果。点击动作会自动附带提交 bug 跟踪器中的错误，并

其指向分析器所属团队。 Tricorder 团队跟进这些“无用”的点击标记，计算“请修复”与“无用”间的点击比率。如果分析器的比例超过 10%那么 Tricorder 团队会禁用该分析器直到作者对其进行进。虽然 Tricorder 团队很少有永久性得禁用分析器，它已经禁用了一些分析器（在几个场景下），到分析器作者删除和修改完成那些结果繁杂无用的检查器。
提交的错误通常能改进分析器效果，从而大大提高开发人员对这些分析器的满意度;例如，Error P one 团队在 2014 年开发了一个检查项，它会标记出来当 Guava 中传递太多参数传递给类似 printf 样的的函数。类似 printf 的函数实际上并不接受所有 printf 说明符，只接受%S。大约每周一次 Error Prone 团队将收到一个“无用”bug，声称该分析不正确，实际上 bug 匹配代码中的格式统配符数与实际传递的参数数量相匹配。而用户试图传递%s 以外的通配占位符时，任何情况下分析器其实都正确误区的。因此团队将代码检视说明文本更改为直接声明该函数仅接受%s 占位符并停止获取有关检查的错误。
Tricorder 的使用规模。截至 2018 年 1 月，Tricorder 已经分析了每天大约 50000 次代码审查更。在高峰时段每秒进行三次分析。review 者每天点击“请修复”超过 5000 次，作者每天应用自动复方案约 3000 次。Tricorder 分析器每天收到 250 次“无用”点击反馈。
代码审查分析的成功表明它在 Google 的开发人员工作流程中占据了“最佳位置”。在编译时显的分析结果必须达到相对到的的质量和准确度，而依靠分析器不可能满足来继续识别严重问题。在 rev ew 和代码合入之后，开发人员进行更改所面临的阻力会有所增加。因此，开发人员对已经测试和发的代码进行修改时会比较纠结并且不太可能去解决低危和不太重要的问题。很多其他软件开发组织中分析项目（例如针对 Android / iOS 应用程序的 Facebook Infer 分析）也强调代码审查是报告分析果的关键切入点。
<h3 id="扩展分析器">扩展分析器</h3>
随着 Google 开发者对 Tricorder 分析器的结果取得认可，他们继续要求深入扩展分析器。 Trico der 以两种方式解决这个问题：允许在项目级定制化并在开发流程的其他环节添加展示分析结果。在节中，我们还讨论 Google 尚未利用更复杂的分析技术作为其核心开发流程的部分原因。
<h4 id="项目级定制">项目级定制</h4>
并非所有请求的分析器对于整个 Google 代码库中都具有同等价值;例如一些分析器有较高的误率有关，因此具有相应的高误报率的检查器可能需要在特定的项目启用配置才有效。这些分析器仅对合的团队才有用。
为了实现这些需求，我们的目标是使 Tricorder 达成可定制化。我们之前为 FindBugs 定制的经实践效果较差;基于用户级别的定制化导致团队内部和团队之间出现差异化导致工具使用率下降。因为个用户都可以看到不同的问题视图，所以没有办法确保每个从事同样项目工作的人都能看到特定的问。如果开发人员从他们团队的代码中删除了所有未使用的导包，那么即使其他一个开发人员在删除未用的导包方面不一致，该变更会很快被回退拒绝。
为了避免此类问题，Tricorder 仅允许在项目级别进行配置，确保对特定项目进行更改的任何人能看到与该项目相关的分析结果一致视图。维护结果视图的一致性使得几种类型的分析器能够执行以动作：
产生二分结果。例如，Tricorder 包括用于协议缓冲区定义的分析器，其识别不向后兼容的变化开发人员团队使用它来确保序列化形式的协议缓冲区中的持久信息，但对于不以此形式存储数据的团而言则很烦人。另一个例子是有分析器建议使用对于不能使用这些库或语言功能的项目，对 Guava 或 ava 代码实现没有意义;
需要特定的设置或代码内注释。例如，如果他们的代码被适当地注释，团队仅可使用 Checker Fr mework 的 null ness 去分析。另一项分析器是在合理配置后，将检查特定 Android 二进制文件的进制大小和函数调用次数的增长，并警告开发人员是否是预期增长或者是否接近限制范围;
支持特定领域的语言（DSL）和特定于团队的编码指南。一些 Google 软件开发团队开发了一些型 DSL 并且希望运行的相关检查器。其他团队已经实现在可读性和可维护性方面的最佳实践并希望续执行这些检查;
同时是资源高度利用化的。按照包含动态分析的结果混合分析的案例。这样的分析为一些团队提部分高价值，但对所有人来说成本太高或耗时太多。
截至 2018 年 1 月，Google 内部大约有 70 个可选分析，其中 2500 个项目至少启用了其中一。整个公司的数十个团队正在积极开发新的分析器，大多数隶属于都在开发工具组之外。
<h4 id="其他工作流程集成点">其他工作流程集成点</h4>
随着开发人员对这些工具的信任度增高，他们还要求进一步集成到工作流程中。Tricorder 现在过提供命令行工具，持续集成系统和代码审阅工具提供分析结果。
命令行支持。Tricorder 团队为开发人员添加了命令行支持，这些开发人员实际上是代码管理员

经常浏览并清理团队代码库中的各种告警分析。这些开发人员也非常熟悉每个分析器将生成的修复类，并且高度信任特定分析器。因此开发人员可以使用命令行工具自动应用给定分析中的所有修复并进清理变更;
代码提交门槛。有些团队希望特定的分析器可以阻止代码提交而不是仅仅出现在代码审查工具中通常要求阻止提交的能力是由具有高度定制检查器且保证没有误报的团队提出，通常用在自定义 DSL 或库。
代码展示结果。代码展示最适合显示大型项目（或整个代码库）中问题规模。例如，浏览有关已用 API 的代码时的分析结果可以显示迁移工作需要多少工作量;或者某些安全和隐私分析是全球性的需要专业团队在确定是否存在问题之前审查结果。由于默认情况下不显示分析结果，因此代码浏览器许特定团队启用分析视图，然后扫描整个代码库并审核结果，而这并不会干扰其他开发人员对这些分器的注意力。如果分析结果具有关联修复，则开发人员只需单击代码浏览工具即可应用此修复。代码览器也非常适合显示生产数据利用的分析结果，因为在代码提交和运行之前这些数据均不可用。
<h4 id="复杂分析">复杂分析</h4>
在 Google 上广泛部署的所有静态分析都相对简单，尽管有些团队针对特定领域（例如 Android 应用程序）的项目特定分析框架进行过程间分析。 Google 规模的过程分析在技术上是可行的。但是施起来这样的分析非常具有挑战性。上面说到所有 Google 的代码都存贮在单独的整体的源码仓库中因此从概念上讲代码仓库中的任何代码都可以是任意二进制文件的一部分。因此可以想象这样一种情，其中特定代码审查的分析结果将需要分析整个代码仓库。尽管 Facebook 的 Infer 专注于过程间分，将基于分离逻辑的分析器扩展到数百万行的代码库，但将这种分析器扩展到 Google 的数十亿行代仓库仍然需要大量的工程化工作。截至 2018 年 1 月，实施一个更复杂的分析系统并不是 Google 的先考虑因素：
大量投资。前期基础设施投资将是令人望而却步的;
需要努力降低误报率。分析团队必须开发技术，以显著降低许多分析器的误报率和/或严格限制显示出来哪些错误信息，就如图 infer 做的一样;
还有更多要实施。分析团队仍然有更多“简易”的分析器需要去实现和集成;
高昂的前期成本。我们发现这种“简单”分析器的性价比很高，这是 FindBugs 的核心动机。相之下，即使确定更复杂的检查器的成本 ROI，前期成本也很高。
请注意，对于在专业领域（例如航空航天和医疗设备）或特定项目（例如设备驱动程序和手机应程序）上工作的 Google 以外的开发人员，此 ROI 可能会有很大差异。
<h3 id="心得">心得</h3>
我们尝试将静态分析融入到 Google 工作流程中的学费教会以下我们宝贵的经验教训：
发现 bug 缺陷很容易。当代码库足够庞大时，它几乎包含任何可以想象得到的代码模式。即使具有完整测试覆盖率和严格的代码审查流程的成熟代码库中错误也在若隐若现。有时候问题在本地检中并不明显，有时候由看似人畜无害的重构所引入错误。例如考虑以下代码片段使用类型为 long 的段 f，
<blockquote>
result =
31 * result

(int) (f ^ (f >>> 32));

</blockquote>
想象下如果开发人员将 f 的类型更改为 int 会发生什么。代码继续编译，但是向右偏移 32 变为 n-op 操作，字段与自身进行异或，变量的 hash 值变为常量 0.结果是 f 不再影响 hashCode 方法生成值。任何能够计算 f 类型的工具都可以正确地检测到向右偏移超过 31 的情况，我们在 Google 的代库中修复了了 31 项出现该错误的代码，同时在 Error Pone 中将该检查纳入编译器错误。
由于发现错误很容易，Google 使用简单的工具来检测错误类型。接下来分析编写者根据运行 Google 代码的结果进行微调。
大多数开发人员都不会如他们所想的那样使用静态分析工具。随着许多商业工具的发展，Google 最初依赖 FindBugs 的实施，工程师选择访问集中的 dashboard 来查看他们项目中所发现的问题，是其中很少有人真正去这样查看。查找已合入代码中的错误（可能已部署并在没有用户可感知到问题情况下运行）为时已晚。为了确保大多数或所有工程师都能看到静态分析警告，必须将分析工具集成工作流程中，并默认为每个人启用。Error Prone 等项目不提供错误 dashboard，而是通过额外的检器扩展编译器，并且在代码审查时展示分析结果。

开发者的感受至关重要。根据我们的经验和材料积累，许多尝试将静态分析集成到软件开发组织尝试都失败了。在 Google 管理层通常没有授权工程师使用静态分析工具。从事静态分析的工程师必通过有效实际数据证明其影响力。要使静态分析项目取得成功开发者必须感知到他们从中受益并享受用它的价值。
为了构建成功的分析平台，我们构建了可为开发人员提供高价值的工具。 Tricorder 团队会仔细阅已修复的问题，实际调研以了解开发人员的感受，使得通过分析工具提交 bug 更为便捷，并使用有这些数据来持续改进。开发人员需要建立对分析工具的信任。如果一个工具浪费了开发人员时间的报和反馈低级别问题，那么开发人员就会失去信心并忽视结果。
不局限于发现错误，修复它们。要推广静态分析工具，一种典型的方法是列举代码库中存在的大问题。目的是通过指出纠正潜在错误或去在未来阻止 bug 发生来影响采取措施。但是如果开发人员到不激励他们采取行动，那么这种潜在预期结果仍将无法实现。这是一个基本缺陷：分析工具通过它识别的问题数来衡量它们的实用性，而流程集成会由于只有极少数的 bug 修复而失败。相反 Google 静态分析团队会同找 bug 一样也负责相应地修复工作，将其作为是否成功闭环的标准。专注于修复误确保了工具提供可行的建议并最大限度地减少误报。在许多情况下，修复错误就像通过自动化工具到它们一样容易。即使对于难以解决的问题，过去五年的研究也凸显了自动创建静态分析问题修复方的新技术。
分析器开发需群策齐力。虽然特定的静态分析工具需要专家开发人员编写分析，但专家可能很少际上并不知道哪些检查会产生较大的影响因子。此外分析器专家通常不是特定领域专家（例如那些使用 API，语言和安全方面的专家）。通过 FindBugs 集成只有少数 Google 员工了解如何编写新检查器因此小型 BugBot 团队必须自己完成所有工作。这限制了添加新检查的速度并事实上不能由其他人从们的领域知识贡献而获益。像 Tricorder 这样的团队现在专注于降低开发人员提供的检查标准，不需事先具备静态分析经验。例如 Google 工具 Refaster 允许开发人员通过在代码片段之前和之后指定例来编写检查器。由于贡献者在自己调试错误代码之后经常有动力做出贡献，因此新的检查会逐步节开发人员时间。
<h3 id="结论">结论</h3>
我们的体会心得是重视集成于开发流程是静态分析工具实施的关键。虽然检查器工具作者可能认开发人员应该面对他们编写的代码中存在缺陷列表感到高兴，但实际上我们并未发现这样的列表会激开发人员去修复这些缺陷。作为分析工具开发人员，我们必须通过实际纠正的缺陷方面来定义衡量效，而不是给开发人员提供数字。这意味着我们的责任远远超出了分析工具本身。
我们倡导一个专注于尽早推动工作流程集成的系统。尽可能将检查器作为编译器错误启用。为了免中断构建工具编写者首先承担修复代码库中所有现有问题的任务，允许我们不断前行一步一步地提高 Google 代码库的质量。由于我们在编译器中呈现出错误告警，开发人员在编写代码后立即对其进行理，这样他们仍然可以进行及时更改。为实现这一目标，我们开发了基础架构用于运行分析并在整个大的 Google 代码库中生成修复程序。我们还受益于代码审查和允许更改数百个文件的提交自动化，然还有工程文化，其通常容许合入遗留代码的变更，因为改进代码胜过对修改风险的厌恶。
代码审查是提交代码之前显示分析警告的最佳切入点。为了确保开发人员能够接受分析结果 Tricorder 仅在开发人员在提交更改之前的修改代码阶段才会展示问题，并且 Tricorder 团队应用一系列标来选择要显示的告警。Tricorder 进一步在代码审查工具中收集统计数据，该工具用于检测分析器产大量无效告警的根因。
为了克服告警被忽视，我们努力重新赢得得谷歌工程师的信任，发现谷歌开发人员有强烈的偏见忽视静态分析，任何误报率不理想的报告都给他们不作为的理由。分析团队非常谨慎只有在根据描述观标准对其进行审查后才能将检查结果作为错误或警告显示，因此开发人员很少被分析结果淹没，混或烦恼。调查和反馈渠道是这一过程的重要质量控制方法。现在开发人员已经对分析结果重新抱有信感，Tricorder 团队正在满足在 Google 开发人员工作流程中更多介入更多分析的需求。
我们在 Google 上构建了一个成功的静态分析基础架构，在编译时和代码审查期间可防止每天有百个错误进入 Google 代码库。我们希望其他人可以从我们的经验中获益，将静态分析成功整合到他自己的工作流程。