

## 第十七章

# 代码搜索

作者:Alexander Neubeck 和 Ben St. John 编辑:  
Lisa Carey

代码搜索是 Google 中用于浏览和搜索代码的工具,由前端 UI 和各种后端元素组成。与 Google 的许多开发工具一样,它的出现直接源于对代码库规模的扩展需求。代码搜索最初是用于内部代码的 grep 类型工具<sup>1</sup>与外部代码搜索的排名和 UI 的组合。<sup>2</sup>它作为 Google 开发人员的关键工具的地位因 Kythe/Grok 的集成而得到巩固,<sup>3</sup>增加了交叉引用和跳转到符号定义的功能。

这种集成将重点从搜索转变为浏览代码,后来代码搜索的开发部分遵循了“单击即可回答有关代码的下一个问题”的原则。现在,诸如“这个符号在哪里定义?”,“它在哪里使用?”,“我如何包含它?”,“它何时添加到代码库中?”,甚至诸如“在整个 Fleet 中,它消耗了多少 CPU 周期?”之类的问题,都可以通过一两次单击来回答。

与集成开发环境 (IDE) 或代码编辑器相比,代码搜索针对大规模阅读、理解和探索代码的用例进行了优化。为此,它严重依赖基于云的后端来搜索内容和解析交叉引用。

---

<sup>1</sup> GSearch 最初运行在 Jeff Dean 的个人电脑上,这曾一度引起全公司的困扰,当时他去度假了而且关闭了!

<sup>2</sup>于 2013 年关闭;请参阅[https://en.wikipedia.org/wiki/Google\\_Code\\_Search](https://en.wikipedia.org/wiki/Google_Code_Search)。

<sup>3</sup>现在被称为 Kythe,提供交叉引用(以及其他功能)的服务;使用特定的代码符号(例如函数)的完整构建信息来将其与具有相同名称的其他代码符号区分开来。

在本章中,我们将更详细地介绍代码搜索,包括 Google 员工如何将其用作开发人员工作流程的一部分,为什么我们选择开发一个单独的代码搜索网络工具,以及研究它如何解决在 Google 存储库规模上搜索和浏览代码的挑战。

## 代码搜索用户界面

搜索框是代码搜索 UI 的核心元素（见图17-1）,与网页搜索类似,它提供“建议”,开发人员可以使用这些建议快速导航到文件、符号或目录。对于更复杂的用例,将返回包含代码片段的结果页面。搜索本身可以看作是一种即时的“在文件中查找”（类似 Unix grep命令）,具有相关性排名和一些特定于代码的增强功能,如适当的语法突出显示、范围感知以及注释和字符串文字感知。搜索也可以从命令行进行,并且可以通过远程过程调用 (RPC) API 合并到其他工具中。当需要后期处理或结果集太大而无法手动检查时,这非常有用。

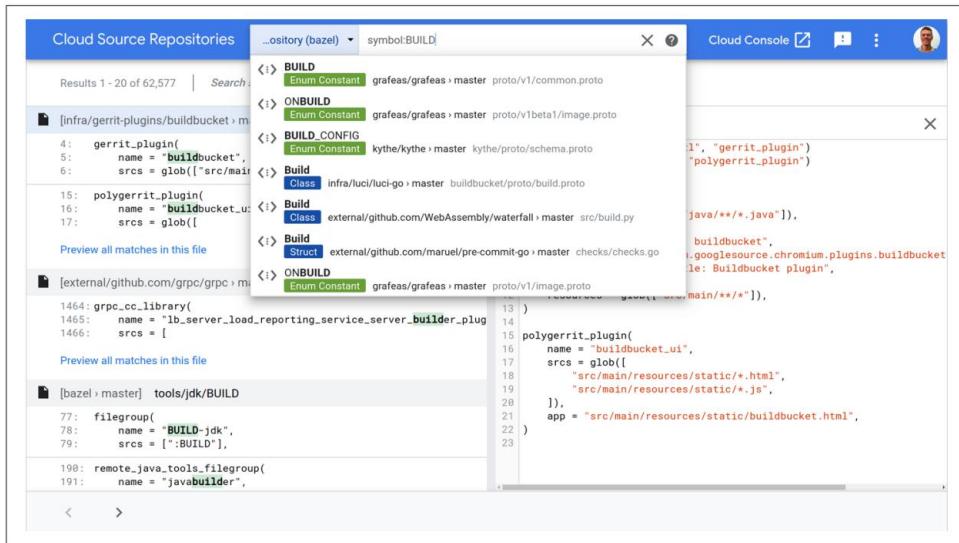


图 17-1. 代码搜索 UI

查看单个文件时,大多数标记都是可点击的,以便用户快速导航到相关信息。例如,函数调用将链接到其函数定义,导入的文件名链接到实际源文件,或注释中的错误 ID 链接到相应的错误报告。这是由基于编译器的索引工具（如Kythe）提供支持的。

单击符号名称将打开一个面板,其中显示使用该符号的所有位置。

通常,将鼠标悬停在函数中的局部变量上将突出显示实现中该变量的所有出现。

代码搜索还通过与 Piper 的集成显示文件的历史记录 (参见第 16 章)。这意味着可以看到文件的旧版本、哪些更改影响了它、谁编写了这些更改、在 Critique 中跳转到这些更改 (参见第 19 章)、文件的不同版本以及经典的“责备”视图 (如果需要)。甚至可以从目录视图中看到已删除的文件。

## Google 员工如何使用代码搜索?

尽管其他工具也提供类似的功能,但 Google 员工仍然大量使用代码搜索 UI 进行搜索和文件查看,并最终理解代码。<sup>4</sup>工程师尝试使用代码搜索完成的任务可以看作是回答有关代码的问题,并且重复的意图变得可见。<sup>5</sup>

### 在哪里?

大约 16% 的代码搜索会尝试回答代码库中特定信息的位置问题;例如,函数定义或配置、API 的所有用法,或者特定文件在存储库中的位置。这些问题非常有针对性,可以通过搜索查询或遵循语义链接 (如“跳转到符号定义”) 非常准确地回答。这类问题通常在重构/清理等大型任务期间或与其他工程师合作项目时出现。因此,必须有效地解决这些小知识空白。

代码搜索提供了两种帮助方式:对结果进行排名和丰富的查询语言。排名解决了常见情况,而搜索可以非常具体 (例如,限制代码路径、排除语言、仅考虑函数) 以处理罕见情况。

用户界面让与同事共享代码搜索结果变得容易。因此,对于代码审查,您只需添加链接即可。例如,“您是否考虑过使用这个专门的哈希映射:cool\_hash.h?”这对于文档、错误报告和事后分析也非常有用,并且是引用代码的规范方式。

---

<sup>4</sup>无处不在的代码浏览器鼓励了一种有趣的良性循环:编写易于浏览的代码。这可能意味着不要将层次结构嵌套得太深,因为层次结构嵌套太深需要多次点击才能从调用点移动到实际实现,并且使用命名类型而不是字符串或整数等通用类型,因为这样很容易找到所有用法。

<sup>5</sup> Sadowski, Caitlin, Kathryn T. Stolee 和 Sebastian Elbaum。“开发人员如何搜索代码:案例研究”载于 2015 年第 10 届软件工程基础联合会议 (ESEC/FSE 2015) 的论文集。<https://doi.org/10.1145/2786805.2786855>。

在 Google 内部。甚至可以引用旧版本的代码，因此链接可以在代码库发展过程中保持有效。

## 什么？

大约四分之一的代码搜索是经典的文件浏览，用于回答代码库中特定部分正在做什么的问题。这些类型的任务通常更具探索性，而不是定位特定结果。这是使用代码搜索来阅读源代码，以便在进行更改之前更好地理解代码，或者能够理解其他人的更改。

为了简化这类任务，代码搜索引入了通过调用层次结构进行浏览和在相关文件之间快速导航的功能（例如，在标头、实现、测试和构建文件之间）。这是通过轻松回答开发人员在查看代码时遇到的众多问题来理解代码。

## 如何？

最常见的用例（约占代码搜索的三分之一）是查看其他人如何完成某件事的示例。通常，开发人员已经找到了特定的 API（例如，如何从远程存储读取文件），并希望了解如何将该 API 应用于特定问题（例如，如何稳健地设置远程连接并处理某些类型的错误）。代码搜索还用于首先找到适合特定问题的库（例如，如何有效地计算整数值的指纹），然后选择最合适的实现。对于这些类型的任务，搜索和交叉引用浏览的组合是典型的。

## 为什么？

与代码的功能相关，有更多针对性查询围绕代码行为为何与预期不同。约 16% 的代码搜索试图回答为何添加某段代码或为何其行为方式不同。

在调试过程中经常会出现这样的问题；例如，为什么在这种特定情况下会出现错误？

这里的一个重要功能是能够搜索和探索特定时间点的代码库的确切状态。调试生产问题时，这可能意味着要处理几周或几个月前的代码库状态，而调试新代码的测试失败通常意味着要处理几分钟前的更改。这两种情况都可以通过代码搜索实现。

## 谁和何时？

大约 8% 的代码搜索会尝试回答有关某人何时或谁引入了某段代码的问题，并与版本控制系统进行交互。例如，可以查看引入特定行的时间（如 Git 的“责备”）并跳转到相关的代码审查。此历史记录面板在找到询问代码或审查代码更改的最佳人选时也非常有用。<sup>6</sup>

## 为什么要使用单独的 Web 工具？

除了 Google 之外，上述大多数调查都是在本地 IDE 中完成的。那么，为什么还要使用另一个工具呢？

### 规模

第一个答案是，Google 代码库非常庞大，以至于完整代码库的本地副本（大多数 IDE 的先决条件）根本无法放在一台机器上。即使在遇到这一根本障碍之前，为每个开发人员构建本地搜索和交叉引用索引也是有成本的，这种成本通常在 IDE 启动时就已支付，从而降低了开发人员的速度。或者，如果没有索引，一次性搜索（例如使用 grep）可能会变得非常慢。集中式搜索索引意味着只需预先完成一次这项工作，并且意味着对这一过程的投资将使每个人都受益。例如，代码搜索索引会随着每次提交的更改而逐步更新，从而能够以线性成本构建索引。<sup>7</sup>

在普通的网络搜索中，快速变化的当前事件与变化较慢的项目（例如稳定的维基百科页面）混杂在一起。同样的技术可以扩展到搜索代码，使索引成为增量式，从而降低成本并允许所有人在立即看到代码库的更改。提交代码更改后，只需重新索引实际涉及的文件，从而允许并行和独立更新全局索引。

不幸的是，交叉引用索引不能以同样的方式立即更新。

它不可能实现增量，因为任何代码更改都可能影响整个代码库，实际上通常会影响数千个文件。许多（几乎所有

---

<sup>6</sup>尽管如此，考虑到机器生成的更改的提交率，天真的“责备”追踪的价值不如在更不愿意改变的生态系统中那么大。

<sup>7</sup>相比之下，“每个开发人员在自己的工作区上都有自己的 IDE 进行索引计算”的模型的扩展速度大致是二次方：开发人员每单位时间产生的代码量大致是恒定的，因此代码库的扩展速度是线性的（即使开发人员数量是固定的）。线性数量的 IDE 每次完成的工作量也是线性的。这不是实现良好扩展的秘诀。

需要构建 8 (或至少分析)完整二进制文件来确定完整的语义结构。它使用大量计算资源来每天生成索引 (当前频率)。即时搜索索引和每日交叉引用索引之间的差异是用户遇到罕见但反复出现的问题的根源。

### 零设置全局代码视图能够即时有效地浏

览整个代码库意味着可以非常轻松地找到要重用的相关库和要复制的好示例。对于在启动时构建索引的 IDE,需要有一个小项目或可见的范围来减少此时间并避免自动完成等工具被噪音淹没。使用代码搜索 Web UI,无需进行任何设置 (例如项目描述、构建环境),因此无论代码出现在哪里,都可以非常轻松快速地了解代码,从而提高开发人员的效率。也不会有遗漏代码依赖项的危险;例如,在更新 API 时,可以减少合并和库版本控制问题。

### 专业化可能令人惊

讶的是,Code Search 的一个优点是它不是 IDE。这意味着用户体验 (UX) 可以针对浏览和理解代码进行优化,而不是编辑代码,这通常是 IDE 的主要内容 (例如键盘快捷键、菜单、鼠标单击甚至屏幕空间)。例如,由于没有编辑器的文本光标,因此每次鼠标单击符号都可以变得有意义 (例如,显示所有用法或跳转到定义),而不是移动光标的方式。

这个优势非常大,开发人员在编辑器中同时打开多个代码搜索选项卡的情况非常普遍。

### 与其他开发者工具集成由于它是查看源代码的主要方

式,因此代码搜索是公开源代码信息的逻辑平台。它使工具创建者无需为其结果创建 UI,并确保所有开发者都能了解他们的工作而无需宣传。许多分析会定期在整个 Google 代码库上运行,其结果通常会显示在代码搜索中。对于

---

<sup>8</sup>凯西检测构建工作流以从源代码中提取语义节点和边。此提取过程会收集每个构建规则的部分交叉引用图。在后续阶段,这些部分图将合并为一个全局图,并针对最常见的查询 (转到定义、查找所有用法、获取文件的所有修饰) 优化其表示。每个阶段 (提取和后处理) 的成本大致与完整构建一样高;例如,在 Chromium 的情况下,Kythe 索引的构建在分布式设置中大约需要六个小时,因此成本太高,无法由每个开发人员在自己的工作站上构建。这种计算成本就是为什么 Kythe 索引每天只计算一次的原因。

例如,对于许多语言,我们可以检测“死”代码(未调用的代码),并在浏览文件时将其标记为“死”代码。

在另一个方向,指向源文件的代码搜索链接被视为其规范的“位置”。这对许多开发人员工具都很有用(见图17-2)。例如,日志文件行通常包含日志语句的文件名和行号。生产日志查看器使用代码搜索链接将日志语句连接回生产代码。根据可用信息,这可以是特定修订版本的文件的直接链接,也可以是具有相应行号的基本文件名搜索。如果只有一个匹配的文件,则在相应的行号处打开它。否则,将呈现每个匹配文件中所需行的片段。

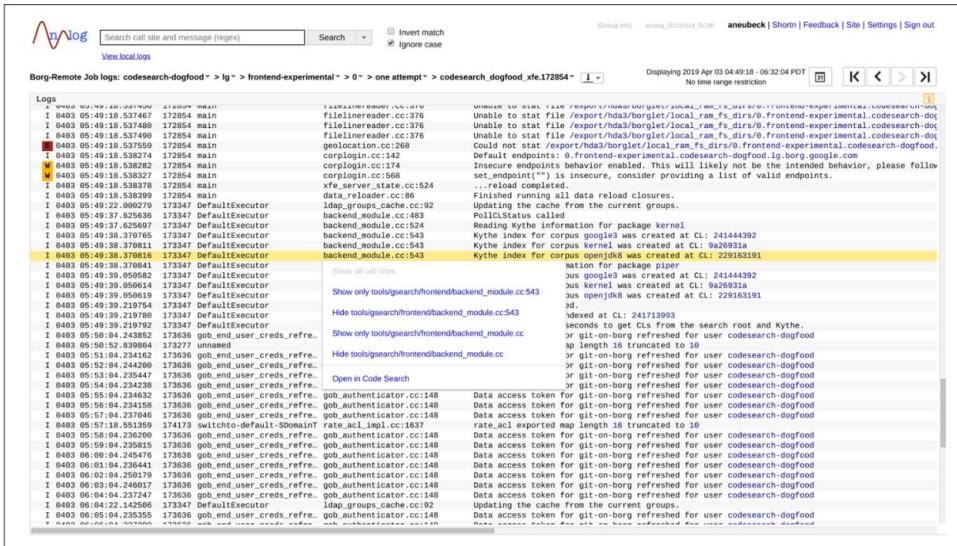


图 17-2. 日志查看器中的代码搜索集成

类似地,无论堆栈框架是在崩溃报告工具中还是在日志输出中显示,它们都会链接回源代码,如图 17-3 所示。根据编程语言的不同,链接将使用文件名或符号搜索。由于崩溃二进制文件构建的存储库快照是已知的,因此搜索实际上可以限制到这个版本。这样,即使相应的代码后来被重构或删除,链接也会在很长一段时间内保持有效。



图 17-3. 堆栈框架中的代码搜索集成

编译错误和测试通常也会引用代码位置（例如，在文件第 1 行测试 X）。鉴于大多数开发都发生在特定的云可见工作区中，并且可以通过代码搜索进行访问和搜索，因此即使对于未提交的代码，也可以对其进行链接。

最后，代码实验室和其他文档引用了 API、示例和实现。此类链接可以是引用特定类或函数的搜索查询，当文件结构发生变化时，这些链接仍然有效。对于代码片段，head 中最新的实现可以轻松嵌入到文档页面中，如图 17-4 所示，而无需使用其他文档标记污染源文件。

Use a [markdown code block](#) and specify `live-snippet` as the language. Live snippets use the Code Search `cs/` query syntax to specify which code to include. For example:

```
```live-snippet
cs/file:google3/corp/g3doc/tests/regression_tests/testLiveSnippets/snippet_test.cc f
```

```

Which renders as:

```
static int Fibonacci(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1;
    for (int i = 0; i < n; ++i) {
        int t = a;
        a = b;
        b += t;
    }
    return b;
}
```

图 17-4. 文档中的代码搜索集成

## API 公开Code

Search 将其搜索、交叉引用和语法突出显示 API 公开给工具,因此工具开发人员可以将这些功能引入他们的工具中,而无需重新实现它们。此外,还编写了插件来为编辑器和 IDE (如 vim、emacs 和 IntelliJ) 提供搜索和交叉引用。这些插件恢复了由于无法本地索引代码库而损失的部分功能,并恢复了开发人员的工作效率。

# 规模对设计的影响

在上一节中,我们了解了代码搜索 UI 的各个方面,以及为什么值得拥有一个单独的代码浏览工具。在以下各节中,我们将介绍实现的幕后情况。我们首先讨论主要挑战 扩展,然后讨论大规模如何使制作一个好的代码搜索和浏览产品变得复杂。之后,我们将详细介绍如何解决其中的一些挑战,以及在构建代码搜索时做出了哪些权衡。

搜索代码的最大<sup>9</sup>扩展挑战是语料库的大小。对于几兆字节的小型存储库,使用grep搜索进行强力搜索即可。当需要搜索数百兆字节时,简单的本地索引可以将搜索速度提高一个数量级或更多。当需要搜索 GB 或 TB 级的源代码时,具有多台机器的云托管解决方案可以使搜索时间保持合理。集中式解决方案的实用性会随着使用它的开发人员数量和代码空间大小的增加而增加。

### 搜索查询延迟尽管我们认为

快速响应的 UI 对用户更有利,但低延迟并非没有代价。为了证明这种努力是值得的,我们可以将其与所有用户节省的工程时间进行比较。在 Google,我们每天在代码搜索中处理来自开发人员的超过一百万个搜索查询。对于一百万个查询,每个搜索请求仅增加一秒就相当于每天有大约 35 名全职工程师闲置。相比之下,搜索后端的构建和维护只需要这些工程师的大约十分之一。这意味着,每天大约有 100,000 个查询 (相当于不到 5,000 名开发人员),仅一秒钟的延迟参数就是盈亏平衡点。

事实上,生产力损失并不会随着延迟而线性增加。**如果延迟低于 200 毫秒,则 UI 被视为响应迅速。**但仅仅一秒钟之后,开发人员的注意力就会开始分散。如果再过 10 秒钟,开发人员就会

---

<sup>9</sup>由于查询是独立的,因此可以通过更多服务器来满足更多用户的需求。

很可能会完全切换上下文,这通常被认为会降低生产力。让开发人员保持高效“流动”状态的最佳方法是将所有频繁操作的端到端延迟控制在 200 毫秒以下,并投资相应的后端。

为了浏览代码库,需要执行大量代码搜索查询。理想情况下,“下一个”文件只需单击一下即可(例如,对于包含的文件或符号定义),但对于一般导航,与使用传统文件树相比,只需搜索所需的文件或符号即可,速度要快得多,理想情况下无需完全指定它,并且会为部分文本提供建议。随着代码库(和文件树)的增长,这种情况变得越来越真实。

正常情况下,导航至另一个文件夹或项目中的特定文件需要多次用户交互。使用搜索时,只需几次按键即可找到相关文件。为了使搜索更有效,可以向搜索后端提供有关搜索上下文(例如,当前查看的文件)的其他信息。上下文可以将搜索限制在特定项目的文件内,或者通过优先选择靠近其他文件或目录的文件来影响排名。在代码搜索 UI<sup>10</sup> 中,用户可以预定义多个上下文并根据需要在它们之间快速切换。在编辑器中,打开或编辑的文件被隐式用作上下文,以按其邻近程度对搜索结果进行优先排序。

人们可以将搜索查询语言(例如,指定文件、使用正则表达式)的功能作为另一个标准;我们将在本章稍后的权衡部分讨论这个问题。

### 索引延迟大多数情况

况下,开发人员不会注意到索引何时过期。他们只关心一小部分代码,即使如此,他们通常也不知道是否有更新的代码。但是,对于他们编写或审查相应更改的情况,不同步可能会造成很多混乱。更改是小修复、重构还是一段全新的代码往往并不重要。开发人员只是希望获得一致的视图,就像他们在小型项目的 IDE 中体验到的那样。

编写代码时,需要立即索引已修改的代码。当添加新文件、函数或类时,无法找到它们会令人沮丧,并且会破坏习惯于完美交叉引用的开发人员的正常工作流。另一个例子是基于搜索和替换的重构。删除的代码立即从搜索结果中消失不仅更加方便,而且后续重构考虑新状态也是必不可少的。使用

---

<sup>10</sup>代码搜索 UI 也有一个经典的文件树,因此也可以通过这种方式导航。

在集中式 VCS 中,如果之前的更改不再是本地修改的文件集的一部分,开发人员可能需要对提交的代码进行即时索引。

相反,有时能够回到代码的先前快照 (即发布版本)很有用。在发生事件期间,索引和正在运行的代码之间的差异可能特别成问题,因为它可能隐藏真实原因或引入无关的干扰。这对于交叉引用来说是一个问题,因为当前用于构建 Google 规模的索引的技术只需要几个小时,而且复杂性意味着只保留一个“版本”的索引。

虽然可以进行一些修补以使新代码与旧索引对齐,但这仍然是一个有待解决的问题。

## 谷歌的实施

Google 的代码搜索具体实现是根据其代码库的独特特性量身定制的,上一节概述了我们为创建强大且响应迅速的索引而设计的约束。下一节概述了代码搜索团队如何实现并将其工具发布给 Google 开发人员。

### 搜索索引

Google 的代码库对 Code Search 来说是一个特殊的挑战,因为它的规模非常庞大。早期采用基于三元组的方法。随后 Russ Cox 开源了一个[简化版本](#)。目前,代码搜索索引了大约 1.5 TB 的内容,每秒处理大约 200 个查询,服务器端搜索延迟中位数小于 50 毫秒,索引延迟中位数 (代码提交和索引中可见性之间的时间) 小于 10 秒。

让我们粗略地估计一下使用基于 grep 的强力解决方案实现此性能所需的资源。我们用于正则表达式匹配的 RE2 库以大约 100 MB/秒的速度处理 RAM 中的数据。给定 50 毫秒的时间窗口,需要 300,000 个核心来处理 1.5 TB 的数据。因为在大多数情况下,简单的子字符串搜索就足够了,所以可以用特殊的子字符串搜索替换正则表达式匹配,在某些条件下,该子字符串搜索可以处理大约 1 GB/秒<sup>11</sup>,从而将核心数量减少 10 倍。到目前为止,我们只研究了在 50 毫秒内处理单个查询的资源要求。如果我们每秒收到 200 个请求,其中 10 个将在该 50 毫秒窗口中同时处于活动状态,这样我们仅用于子字符串搜索就需要 300,000 个核心。

---

<sup>11</sup>请参阅<https://blog.scalyr.com/2014/05/searching-20-gbsec-systems-engineering-before-algorithms> 和 [http://volnitsky.com/project/str\\_search](http://volnitsky.com/project/str_search)。

虽然这个估计忽略了搜索在找到一定数量的结果后可以停止,或者文件限制的评估比内容搜索更有效,但它也没有考虑通信开销、排名或向数万台机器的扩散。但它很好地展示了所涉及的规模以及 Google 代码搜索团队不断投资改进索引的原因。多年来,我们的索引从最初的基于三元组的解决方案,到基于自定义后缀数组的解决方案,再到当前的稀疏 n-gram 解决方案。这个最新的解决方案比蛮力解决方案效率高 500 倍以上,同时还能能够以极快的速度响应正则表达式搜索。

我们从基于后缀数组的解决方案转向基于标记的 n-gram 解决方案的原因之一是利用 Google 的主要索引和搜索堆栈。使用基于后缀数组的解决方案,构建和分发自定义索引本身将成为一项挑战。通过利用“标准”技术,我们可以从核心搜索团队在反向索引构建、编码和服务方面取得的所有进步中受益。即时索引是标准搜索堆栈中存在的另一项功能,在大规模解决它时,它本身就是一个巨大的挑战。

依赖标准技术是在实现简单性和性能之间进行权衡。尽管 Google 的代码搜索实现基于标准反向索引,但实际的检索、匹配和评分都是高度定制和优化的。否则,一些更高级的代码搜索功能将无法实现。为了索引文件修订历史记录,我们提出了一种自定义压缩方案,其中索引完整历史记录只会增加 2.5 倍的资源消耗。

在早期,Code Search 从内存中提供所有数据。随着索引大小的增加,我们将[倒排索引](#)移到了闪存。尽管闪存至少比内存便宜一个数量级,但其访问延迟至少高出两个数量级。因此,在内存中运行良好的索引可能不适合从闪存中提供。例如,原始三元组索引不仅需要从闪存中获取大量反向索引,还需要获取相当大的反向索引。使用 n-gram 方案,可以减少反向索引的数量及其大小,但代价是索引更大。

为了支持本地工作区(与全局存储库的差异很小),我们让多台机器进行简单的强力搜索。工作区数据在第一次请求时加载,然后通过监听文件更改来保持同步。当内存不足时,我们会从机器中删除最不最新的工作区。使用我们的历史索引搜索未更改的文档。因此,搜索被隐式地限制在工作区同步到的存储库状态。

## 排名对于非

常小的代码库,排名不会带来太多好处,因为结果本来就不多。但是代码库越大,结果就越多,排名就越重要。在 Google 的代码库中,任何短字符串都会出现数千次,甚至数百万次。如果没有排名,用户要么必须检查所有这些结果才能找到正确的结果,要么必须进一步优化查询 12,直到结果集减少到只有少数几个文件。这两种选择都会浪费开发人员的时间。

排名通常始于一个评分函数,该函数将每个文件的一组特征(“信号”)映射到某个数字:分数越高,结果越好。然后,搜索的目标是尽可能高效地找到前 N 个结果。通常,人们会区分两种类型的信号:仅依赖于文档的信号(“查询无关”)和依赖于搜索查询及其与文档的匹配方式的信号(“查询相关”)。文件名长度或文件的编程语言是查询无关信号的例子,而匹配是函数定义还是字符串文字则是查询相关信号。

## 查询独立信号一些最重要的查

询独立信号是文件浏览次数和对文件的引用次数。文件浏览次数很重要,因为它们表明开发人员认为哪些文件很重要,因此更有可能想要找到它们。例如,基础库中的实用函数具有很高的浏览次数。无论库是否已经稳定并且不再更改,或者库是否正在积极开发,都无关紧要。此信号的最大缺点是它创建的反馈循环。通过对经常查看的文档进行更高的评分,开发人员查看这些文档的机会就会增加,而其他文档进入前 N 名的机会就会减少。这个问题被称为开发与探索,对此存在各种解决方案(例如,高级 A/B 搜索实验或训练数据的管理)。在实践中,过度展示高分项目似乎并没有什么坏处:当它们不相关时,只需忽略它们,如果需要通用示例,则采用它们。然而,对于新文件来说,这是一个问题,因为它们还没有足够的信息来形成良好的信号。<sup>12</sup>我们还使用对文件的引用次数,这与原始[页面排名算法相似](#),通过将 Web 链接替换为大多数语言中存在的各种“include/import”语句作为引用。我们可以将这个概念扩展到构建

---

<sup>12</sup>与网页搜索相比,在代码搜索查询中添加更多字符总会减少结果集(除了通过正则表达式术语来排除一些罕见的例外)。

<sup>13</sup>这个问题很可能可以通过以某种形式使用新近度作为信号来得到一定程度的纠正,也许可以做一些类似于网页搜索处理新页面的事情,但我们还没有这样做。

依赖项（库/模块级别引用）以及函数和类。  
这种全局相关性通常被称为文档的“优先级”。

在使用引用进行排名时，必须注意两个挑战。首先，您必须能够可靠地提取引用信息。在早期，Google 的代码搜索使用简单的正则表达式提取包含/导入语句，然后应用启发式方法将它们转换为完整的文件路径。随着代码库的复杂性不断增加，这种启发式方法变得容易出错且难以维护。在内部，我们用 Kythe 图中正确的信息替换了这部分。

大规模重构，例如[开源核心库](#)，第二个挑战是，这种变化不会在一次代码更新中自动发生，而是需要分多个阶段进行。通常会引入间接操作，例如隐藏文件的移动。这些间接操作会降低移动文件的页面排名，使开发人员更难发现新位置。此外，文件移动时文件视图通常会丢失，这会使情况更加糟糕。由于这种代码库的全局重组相对罕见（大多数接口很少移动），最简单的解决方案是在这种过渡期间手动提升文件。（或者等到迁移完成，然后自然地提升文件在新位置的排名。）

#### 查询相关信号查询独立信号

可以离线计算，因此计算成本不是主要问题，尽管它可能很高。例如，对于“页面”排名，信号取决于整个语料库，需要类似 MapReduce 的批处理来计算。查询相关信号必须针对每个查询进行计算，计算成本应该很低。这意味着它们仅限于查询和可从索引中快速访问的信息。

与网页搜索不同，我们不仅仅匹配标记。但是，如果有干净的标记匹配（即搜索词与周围有某种形式的中断（例如空格）的内容匹配），则会应用进一步的提升并考虑区分大小写。这意味着，例如，搜索“Point”与“Point \*p”的得分将高于与“appointed to the council”的得分。

为方便起见，默认搜索除了匹配实际文件内容外，还会匹配文件名和限定符号<sup>14</sup>。用户可以指定特定类型的匹配，但这不是必需的。与普通匹配相比，符号和文件名匹配的得分更高

---

<sup>14</sup>在编程语言中，诸如函数“Alert”之类的符号通常在特定范围内定义，例如类（“Monitor”）或命名空间（“absl”）。那么限定名称可能是 absl::Monitor::Alert，即使它没有出现在实际文本中，也可以找到它。

内容匹配以反映开发者的推断意图。与网络搜索一样,开发者可以向搜索中添加更多术语,使查询更加具体。

查询通常会通过文件名提示来“限定”(例如,“base”或“myproject”)。评分会利用这一点,提升查询中大部分内容出现在潜在结果的完整路径中的结果,使此类结果领先于仅包含内容中随机位置的单词的结果。

## 检索

在对文档进行评分之前,需要先找到可能与搜索查询匹配的候选文档。此阶段称为检索。由于检索所有文档并不实际,而只能对检索到的文档进行评分,因此检索和评分必须很好地协同工作才能找到最相关的文档。一个典型的例子是搜索类名。根据类的流行程度,它可能有数千种用法,但可能只有一个定义。如果搜索没有明确限制在类定义上,则在找到具有单个定义的文件之前,固定数量的结果的检索可能会停止。显然,随着代码库的增长,问题变得更具挑战性。

检索阶段的主要挑战是从大量不太有趣的文件中找到少数高度相关的文件。一种效果很好的解决方案称为补充检索。其思想是将原始查询重写为更专业的查询。在我们的示例中,这意味着补充查询将搜索限制为仅定义和文件名,并将新检索到的文档添加到检索阶段的输出中。在补充检索的简单实现中,需要对更多文档进行评分,但获得的额外部分评分信息可用于全面评估检索阶段中最有希望的文档。

## 结果多样性搜索的另

一个方面是结果多样性,这意味着尝试在多个类别中提供最佳结果。一个简单的例子是为一个简单的函数名称提供 Java 和 Python 匹配,而不是用其中一个或另一个填充第一页结果。

当用户意图不明确时,这一点尤其重要。多样性的挑战之一是,结果可以分为许多不同的类别(如函数、类、文件名、本地结果、用法、测试、示例等),但用户界面中没有足够的空间来显示所有结果或所有组合的结果,而且这并不总是可取的。Google 的代码搜索在这方面做得不如网页搜索,但建议结果的下拉列表(如网页搜索的自动完成)经过调整,可在用户当前工作区中提供一组多样化的顶级文件名、定义和匹配项。

## 选定的权衡

在 Google 规模的代码库中实现代码搜索并保持其响应速度需要做出各种权衡。这些将在下一节中说明。

完整性 : 头部存储库我们已经看到,更大的代码库

会给搜索带来负面影响;例如,索引速度更慢、成本更高、查询速度更慢以及结果更嘈杂。可以通过牺牲完整性来降低这些成本吗?换句话说,将一些内容排除在索引之外?答案是肯定的,但要谨慎。非文本文件(二进制文件、图像、视频、声音等)通常不供人类阅读,因此会与文件名分开删除。由于它们很大,所以这可以节省大量资源。更接近边缘的情况涉及生成的 JavaScript 文件。由于混淆和结构的丢失,它们几乎无法被人类阅读,因此将它们从索引中排除通常是一个很好的权衡,以完整性为代价减少索引资源和噪音。从经验上讲,几兆字节的文件很少包含与开发人员相关的信息,因此排除极端情况可能是正确的选择。

但是,从索引中删除文件有一个很大的缺点。开发人员要想依赖代码搜索,就必须能够信任它。不幸的是,如果删除的文件一开始就没有被索引,那么通常无法对特定搜索的不完整搜索结果提供反馈。由此给开发人员带来的困惑和生产力损失是为节省的资源付出的高昂代价。即使开发人员完全了解这些限制,如果他们仍然需要执行搜索,他们也会以一种临时且容易出错的方式进行搜索。考虑到这些罕见但可能很高的成本,我们选择在索引过多方面犯错,设置相当高的限制,这些限制主要是为了防止滥用和保证系统稳定性,而不是为了节省

资源。

另一方面,生成的文件不在代码库中,但通常对索引有用。目前它们还不是,因为索引它们需要集成创建它们的工具和配置,这会带来巨大的复杂性、混乱和延迟。

完整性 : 全部结果与最相关结果普通搜索牺牲了完整性来换取速度,本

质上是在赌排名能确保最靠前的结果包含所有想要的结果。事实上,对于代码搜索而言,排名搜索是更常见的情况,即用户可能在数百万个匹配项中寻找一个特定的东西,例如函数定义。然而,有时开发人员想要所有结果;例如,找到某个特定符号的所有出现情况以进行重构。需要所有结果对于

分析、工具或重构，例如全局搜索和替换。需要提供所有结果，这与网络搜索有着根本区别，因为网络搜索可以采取许多捷径，例如只考虑排名靠前的项目。

能够为非常大的结果集提供所有结果的成本很高，但我们认为这对于工具和开发人员信任结果是必不可少的。但是，对于大多数查询而言，只有少数结果是相关的（要么只有少数匹配项<sup>15</sup>，要么只有少数结果是有趣的），因此我们不想为了潜在的完整性而牺牲平均速度。

为了用一种架构实现这两个目标，我们将代码库拆分成多个分片，并按优先级对文件进行排序。然后，我们通常只需要考虑每个块中与高优先级文件的匹配。这类似于网络搜索的工作方式。但是，如果收到请求，代码搜索可以从每个块中获取所有结果，以保证找到所有结果。<sup>16</sup>这让我们能够同时解决这两种用例，而不会因为不常用的返回大型完整结果集的功能而减慢典型搜索的速度。结果还可以按字母顺序而不是排名顺序提供，这对某些工具很有用。

因此，这里的权衡是更复杂的实现和 API 与更强大的功能，而不是更明显的延迟与完整性。

**完整性：头部、分支、所有历史记录和工作区 与语料库大小**这个维度相关的问题是应该索引哪些代码版本：

具体来说，除了当前代码快照（“头部”）之外，是否还应该索引其他内容。如果索引超过一个文件修订版本，系统复杂性、资源消耗和总体成本将大幅增加。据我们所知，没有 IDE 会索引当前版本以外的任何内容。当我们查看 Git 或 Mercurial 等分布式版本控制系统时，它们的效率很大程度上来自于对历史数据的压缩。但是在构建反向索引时，这些表示的紧凑性就会丧失。另一个问题是，很难有效地索引图结构，而图结构是分布式版本控制系统的基础。

尽管很难索引存储库的多个版本，但这样做可以探索代码的变化方式并找到已删除的代码。在 Google 中，代码搜索会索引（线性）Piper 历史记录。这意味着代码库可以

---

<sup>15</sup>查询分析显示，大约三分之一的用户搜索的结果少于 20 个。

<sup>16</sup>实际上，幕后还会发生更多的事情，以便响应不会变得过于庞大，并且开发人员不会因为进行与几乎所有内容匹配的搜索而导致整个系统崩溃（想象一下搜索字母 “i” 或一个空格）。

搜索代码的任意快照、已删除的代码，甚至搜索由某些人编写的代码。

一个很大的好处是现在可以简单地从代码库中删除过时的代码。

以前，代码通常会被移到标记为过时的目录中，以便以后仍能找到。完整的历史索引还在人们的工作区（未提交的更改）中有效搜索奠定了基础，这些工作区会同步到代码库的特定快照。对于未来，历史索引开辟了在排名时使用有趣信号的可能性，例如作者身份、代码活动等。

工作区与全局存储库有很大不同：

- 每个开发人员都可以拥有自己的工作区。 · 工作区内
- 通常只有少量更改的文件。 · 正在处理的文件经常更改。 · 工作区仅存在相对  
较短的时间段。

为了提供价值，工作区索引必须准确反映工作区的当前状态。

表现力：标记与子字符串与正则表达式规模效应在很大程度上受支持的搜索

功能集的影响。代码搜索支持正则表达式（regex）搜索，这增强了查询语言的功能，允许指定或排除整组术语，并且它们可用于任何文本，这对于不存在更深层次语义工具的文档和语言尤其有用。

开发人员还习惯于在其他工具（例如grep）和上下文中使用正则表达式，因此它们提供了强大的搜索功能，而不会增加开发人员的认知负担。这种能力是有代价的，因为创建索引来有效地查询它们具有挑战性。还有哪些更简单的选择？

基于标记的索引（即单词）具有很好的可扩展性，因为它只存储了实际源代码的一小部分，并且得到了标准搜索引擎的良好支持。缺点是，在处理源代码时，许多用例很难甚至无法使用基于标记的索引有效地实现，因为源代码会为许多在标记化时通常会忽略的字符赋予含义。例如，在大多数基于标记的搜索中，搜索“function()”而不是“function(x)”、“(x ^ y)”或“== myClass”是困难或不可能的。

标记化的另一个问题是代码标识符的标记化定义不明确。

标识符可以以多种方式书写，例如 CamelCase、snake\_case，甚至可以直接混合使用而不需要任何单词分隔符。如果只记得部分单词，那么查找标识符对于基于标记的索引来说是一项挑战。

标记化通常也不关心字母的大小写（“r”与“R”），并且经常会混淆单词；例如，将“searching”和“searched”简化为相同的词干标记搜索。这种缺乏精确度的情况在搜索代码时是一个重大问题。最后，标记化使得无法搜索空格或其他单词分隔符（逗号、括号），而这在代码中非常重要。

搜索能力的下一步<sup>17</sup>是全子串搜索，可以搜索任何字符序列。提供此功能的一种相当有效的方法是通过基于三元组的索引。<sup>18</sup>在其最简单的形式中，生成的索引大小仍然比原始源代码大小小得多。但是，与其他子串索引相比，较小的索引大小是以相对较低的召回率为代价的。这意味着查询速度较慢，因为需要从结果集中过滤掉不匹配项。这时必须在索引大小、搜索延迟和资源消耗之间找到一个良好的折衷点，这在很大程度上取决于代码库大小、资源可用性和每秒搜索次数。

如果有子字符串索引，则很容易将其扩展以允许正则表达式搜索。基本思想是将正则表达式自动机转换为一组子字符串搜索。对于三元组索引，此转换很简单，并且可以推广到其他子字符串索引。由于没有完美的正则表达式索引，因此总是可以构造导致强力搜索的查询。但是，考虑到只有一小部分用户查询是复杂的正则表达式，实际上，通过子字符串索引进行近似非常有效。

## 结论

代码搜索从grep的有机替代品发展成为提高开发人员生产力的核心工具，并在此过程中利用了Google的网络搜索技术。

但是这对你来说意味着什么呢？如果你的项目很小，很容易适应你的IDE，那么可能就没什么意义了。如果你负责大型代码库的工程师的生产力，那么可能会有一些见解。

最重要的一点可能显而易见：理解代码是开发和维护代码的关键，这意味着投资于理解代码将产生可能难以衡量但却是真实存在的红利。我们添加到代码搜索中的每个功能过去和现在都被开发人员用来帮助他们完成日常工作（诚然，有些功能比其他功能更多）。两个最重要的功能，Kythe集成（即添加语义代码理解）和查找工作示例，也最明显地与理解代码有关（而不是例如找到它或查看它）。

---

<sup>17</sup>还有其他中间类型，例如构建前缀/后缀索引，但一般来说，它们提供的功能较少  
搜索查询的表达力，同时仍然具有较高的复杂性和索引成本。

<sup>18</sup> Russ Cox，“[使用三元素引的正则表达式匹配或Google代码搜索的工作原理](#)。”

就工具影响而言,没有人会使用他们不知道其存在的工具,因此让开发人员了解可用的工具也很重要 在 Google,这是“Noogler”培训的一部分,即针对新聘用软件工程师的入职培训。

对您来说,这可能意味着为 IDE 设置标准索引配置文件、分享有关 egrep 的知识、运行 ctags 或设置一些自定义索引工具(如代码搜索)。无论您做什么,它几乎肯定会被使用,而且使用得更多,使用方式也与您预期的不同您的开发人员将从中受益。

## TL;DR

- 帮助开发人员理解代码可以极大地促进工程专业生产力。在 Google,实现这一目标的关键工具是代码搜索。
  - 代码搜索作为其他工具的基础以及所有文档和开发工具链接到的中心标准位置具有额外的价值。
  
  - Google 代码库规模庞大,因此我们开发了一个自定义工具 而不是例如, grep或 IDE 的索引 必要的。
  - 作为一种交互式工具,代码搜索必须快速,允许“问答”工作流程。它应该在搜索、浏览和索引等各个方面都具有较低的延迟。
    - 只有在值得信赖的情况下,它才会被广泛使用,并且只有在索引所有代码、提供所有结果并首先提供所需的结果时,它才
- 会被信任。然而,只要了解它们的局限性,早期功能较弱的版本既有用又可用。

## 第十八章

# 建立系统和建立哲学

作者:Erik Kueer  
由 Lisa Carey 编辑

如果您问 Google 工程师在 Google 工作最喜欢什么（除了免费食物和酷炫产品）,您可能会听到一些令人惊讶的回答:工程师喜欢构建系统。<sup>1</sup> Google 在其发展历程中投入了大量工程精力从头开始创建自己的构建系统,目标是确保我们的工程师能够快速可靠地构建代码。这一努力非常成功,以至于构建系统的主要组件 Blaze 已被离开公司的前 Google 员工多次重新实现。<sup>2</sup> 2015 年,Google 终于开源了 Blaze 的实现,名为 Bazel。

## 构建系统的目

从根本上讲,所有构建系统都有一个简单的目的:将工程师编写的源代码转换为机器可以读取的可执行二进制文件。一个好的构建系统通常会尝试优化两个重要属性:

### 快速

开发人员应该能够键入单个命令来运行构建并返回生成的二进制文件,通常只需几秒钟。

---

<sup>1</sup>在一项内部调查中,83% 的 Google 员工表示对构建系统感到满意,这是第四大  
在接受调查的 19 款工具中,这款工具是最令人满意的。平均满意度为 69%。

<sup>2</sup>请参阅<https://buck.build>和<https://www.pantsbuild.org/index.html>。

正确的

每次任何开发人员在任何机器上运行构建时,他们都应该得到相同的结果（假设源文件和其他输入相同）。

许多较旧的构建系统试图在速度和正确性之间做出权衡,采取可能导致构建不一致的捷径。Bazel 的主要目标是避免在速度和正确性之间做出选择,提供一个结构化的构建系统,以确保始终能够高效且一致地构建代码。

构建系统不仅仅适用于人类;它们还允许机器自动创建构建,无论是用于测试还是发布到生产。事实上,Google 的大多数构建都是自动触发的,而不是由工程师直接触发的。

我们几乎所有的开发工具都以某种方式与构建系统相关联,为使用我们代码库的每个人带来了巨大的价值。以下是利用我们的自动构建系统的工作流程的一小部分示例:

- 代码自动构建、测试并投入生产,无需任何人工干预。不同的团队以不同的速度执行此操作:一些团队每周推送一次,其他团队每天推送一次,而其他团队则以系统可以创建和验证新版本的速度推送一次。(参见第 24 章)。
- 在将开发人员的更改送去进行代码审查时,会自动对其进行测试(参见[第 19 章](#)),以便作者和审查者都可以立即看到由更改导致的任何构建或测试问题。 · 在将更改合并到主干之前,会立即再次对其进行测试,这使得提交重大更改变得更加困难。 · 低级库的作  
者能够在整个代码库中测试他们的更改,确保他们的更改在数百万个测试和二进制文件中都是安全的。 · 工  
程师能够创建大规模更改(LSC),一次涉及数万个源文件(例如,重  
命名一个通用符号),同时仍然能够安全地提交和测试这些更改。我们将在[第 22 章](#)中更详细地讨论 LSC。

所有这一切只有在 Google 对其构建系统的投资下才有可能。

尽管 Google 的规模可能独一无二,但任何规模的组织都可以通过正确使用现代构建系统获得类似的好处。本章介绍了 Google 认为的“现代构建系统”以及如何使用此类系统。

## 如果没有构建系统会发生什么?

构建系统可让您的开发规模化。正如我们将在下一节中说明的那样,如果没有合适的构建环境,我们就会遇到规模化的问题。

## 但我需要的只是一个编译器！

构建系统的需求可能并不明显。毕竟，我们大多数人在刚开始学习编程时可能没有使用构建系统。我们可能首先从命令行调用gcc或javac等工具，或者在集成开发环境（IDE）中调用等效工具。只要我们所有的源代码都在同一个目录中，这样的命令就可以正常工作：

```
javac *.java
```

这将指示Java编译器获取当前目录下的每个Java源文件并将其转换为二进制类文件。在最简单的情况下，这就是我们所需要的。

但是，一旦我们的代码扩展，事情就会迅速变得更加复杂。javac足够智能，可以在当前目录的子目录中查找我们导入的代码。但它无法找到存储在文件系统其他部分的代码（可能是我们几个项目共享的库）。它显然只知道如何构建Java代码。大型系统通常涉及用各种编程语言编写的不同部分，这些部分之间存在依赖关系，这意味着没有一种语言的编译器可以构建整个系统。

一旦我们最终不得不处理来自多种语言或多个编译单元的代码，构建代码就不再是一个一步到位的过程。我们现在需要考虑我们的代码依赖什么，并按照正确的顺序构建这些部分，可能对每个部分使用不同的工具集。如果我们更改任何依赖项，我们需要重复此过程以避免依赖过时的二进制文件。对于即使是中等规模的代码库，这个过程很快就会变得乏味且容易出错。

编译器也不知道如何处理外部依赖项，例如Java中的第三方JAR文件。通常，在没有构建系统的情况下，我们能做的最好的事情就是从互联网上下载依赖项，将其粘贴到硬盘上的lib文件夹中，然后配置编译器以从该目录中读取库。随着时间的推移，我们很容易忘记我们在那里放了什么库、它们来自哪里以及它们是否仍在使用。祝你好运，随着库维护者发布新版本，保持它们的最新状态。

## Shell脚本可以解决问题吗？

假设你的业余项目一开始很简单，你只需使用编译器就可以构建它，但你开始遇到前面描述的一些问题。也许你仍然认为你不需要真正的构建系统，并且可以使用一些简单的shell脚本自动完成繁琐的部分，这些脚本负责以正确的顺序构建事物。这在一段时间内会有所帮助，但很快你就会开始遇到更多的问题：

- 变得乏味。随着系统变得越来越复杂,您开始花费几乎与实际代码一样多的时间处理构建脚本。调试 shell 脚本非常痛苦,因为越来越多的 hack 层层叠加。
- 速度很慢。为了确保你不会意外地依赖过时的库,每次运行构建脚本时,你都会按顺序构建每个依赖项。你考虑添加一些逻辑来检测哪些部分需要重建,但这听起来非常复杂,而且对于脚本来说很容易出错。或者你考虑每次指定哪些部分需要重建,但这样你又回到了原点。
  - 好消息:是时候发布了!最好弄清楚你需要传递给 jar 命令的所有参数,以进行最终构建。并记住如何上传和推送到中央存储库。构建和推送文档更新,并向用户发送通知。嗯,也许这需要另一个脚本……
  - 灾难!您的硬盘崩溃了,现在您需要重建整个系统。您足够聪明,将所有源文件都保留在版本控制中,但您下载的那些库怎么办?您能再次找到它们并确保它们与第一次下载时的版本相同吗?您的脚本可能依赖于安装在特定位置的特定工具 - 您可以恢复相同的环境以便脚本再次运行吗?您很久以前为使编译器正常工作而设置但后来忘记的所有那些环境变量怎么办?
- 尽管存在问题,但您的项目已经足够成功,您可以开始聘请更多工程师。现在您意识到,之前的问题并不需要灾难才会出现。每次有新开发人员加入您的团队时,您都需要经历同样痛苦的引导过程。尽管您尽了最大努力,但每个人的系统仍然存在细微差异。通常,在一个人的机器上有效的方法不一定在另一个人的机器上有效,每次都需要花几个小时调试工具路径或库版本才能找出差异所在。
- 您决定需要自动化构建系统。理论上,这很简单,只需购买一台新电脑,然后将其设置为每晚使用 cron 运行构建脚本即可。您仍然需要经历痛苦的设置过程,但现在您无法像人脑一样能够检测和解决小问题。现在,每天早上当您上班时,您都会发现昨晚的构建失败了,因为昨天开发人员进行了一项更改,该更改在他们的系统上有效,但在自动构建系统上无效。每次都是一个简单的修复,但这种情况发生得太频繁了,以至于您最终每天都要花费大量时间来发现和应用这些简单的修复。

- 随着项目的发展,构建变得越来越慢。有一天,在等待构建完成时,你悲伤地凝视着正在度假的同事闲置的桌面,希望有一种方法可以利用所有浪费的计算能力。

您遇到了一个经典的规模问题。对于一个最多工作几百行代码、最多工作一两周的开发人员（这可能是刚从大学毕业的初级开发人员迄今为止的全部经历）,一个编译器就是您所需要的。脚本也许可以让您走得更远。但是,一旦您需要协调多个开发人员及其机器,即使是完美的构建脚本也不够用,因为很难解释这些机器之间的细微差异。此时,这种简单的方法就失效了,是时候投资真正的构建系统了。

## 现代构建系统

幸运的是,我们遇到的所有问题都已经被现有的通用构建系统多次解决。从根本上讲,它们与我们之前研究的基于脚本的 DIY 方法并没有太大区别:它们在底层运行相同的编译器,您需要了解这些底层工具才能知道构建系统到底在做什么。

但是这些现有的系统已经经历了多年的发展,使得它们比您自己尝试破解的脚本更加强大和灵活。

### 一切都与依赖关系有关在查看前

面描述的问题时,一个主题不断重复:管理自己的代码相当简单,但管理它的依赖关系要困难得多(第 21 章专门介绍这个问题)。依赖关系有各种各样的:有时依赖于某个任务(例如,“在将发布标记为完成之前推送文档”),有时依赖于某个构件(例如,“我需要最新版本的计算机视觉库来构建我的代码”)。有时,你对代码库的另一部分有内部依赖,有时你对另一个团队(在你的组织或第三方)拥有的代码或数据有外部依赖。

但无论如何,“在得到这个之前我需要那个”的想法在构建系统的设计中反复出现,而管理依赖关系可能是构建系统最基本的工作。

基于任务的构建系统我们在上一节开始开

发的 shell 脚本是一个基于任务的原始构建系统的示例。在基于任务的构建系统中,基本工作单元是任务。每个任务都是某种可以执行任何类型逻辑的脚本,任务将其他任务指定为必须在它们之前运行的依赖项。当今使用的大多数主要构建系统 (例如 Ant、Maven、Gradle、Grunt 和 Rake) 都是基于任务的。

大多数现代构建系统都要求工程师创建构建文件来描述如何执行构建,而不是使用 shell 脚本。以下是 [Ant 手册中](#) 的示例:

```
<project name= MyProject default= dist basedir= . >
  <description>简单
    <!-- 为示
    构建设置全局属性 --
  > <property name= src location= src /><property
  name= build location= build /><property
  name= dist location= dist />

  <target name= init ><!--
    创建时间戳 --> <tstamp/><!-- 创建编译使用的
    构建目录结构
  --> <mkdir dir= ${build} /></target>

  <目标名称= 编译 取决于= 初始化
    description= 编译源代码 ><!-- 将 Java 代码
    从 ${src} 编译到 ${build} --> <javac srcdir= ${src} destdir= ${build} /></
    target>

  <target name= dist depends= compile
    description= 生成分发 >
    <!-- 创建分发目录 --> <mkdir dir= ${dist}/lib />

    <!-- 将 ${build} 中的所有内容放入 MyProject-${DSTAMP}.jar 文件中 --> <jar jarfile= ${dist}/lib/MyProject-$
    {DSTAMP}.jar basedir= ${build} /></target>

  <目标名称= 清洁
    description= clean up ><!--
    删除 ${build} 和 ${dist} 目录树 --> <delete dir= ${build} /><delete dir= ${
    dist} /></target></project>
```

构建文件以 XML 编写,定义了一些有关构建的简单元数据以及任务列表（XML3 中的<target>标记）。每个任务执行 Ant 定义的可能命令列表,这些命令包括创建和删除目录、运行javac以及创建 JAR 文件。这组命令可以通过用户提供的插件进行扩展,以涵盖任何类型的逻辑。每个任务还可以通过Depends属性定义其依赖的任务。这些依赖关系形成一个非循环图（参见图 18-1）。

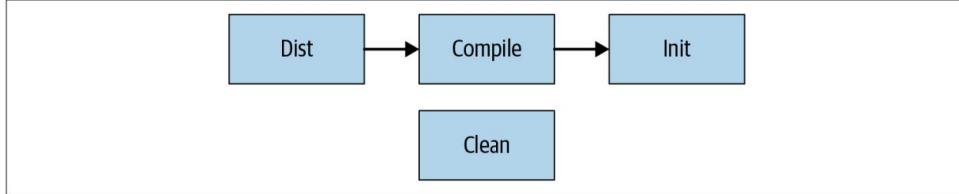


图 18-1. 显示依赖关系的非循环图

用户通过向 Ant 的命令行工具提供任务来执行构建。例如,当用户输入ant dist 时, Ant 将执行以下步骤：

1. 在当前目录下加载一个名为build.xml的文件,并对其进行解析,创建如图18-1所示的图形结构。
2. 查找命令行中提供的名为dist的任务,并发现它依赖于名为compile 的任务。
3. 查找名为compile的任务,发现它对名为 init 的任务有依赖关系。
4. 查找名为init的任务,发现它没有依赖项。
5. 执行init任务中定义的命令。
6. 执行编译任务中定义的命令,前提是该任务的所有命令  
依赖项已运行。
7. 执行dist任务中定义的命令,前提是该任务的所有命令  
依赖项已运行。

最终,Ant 运行dist任务时执行的代码相当于以下 shell 脚本：

```
./createTimestamp.sh
mkdir 构建/
javac src/* -d 构建/
```

---

<sup>3</sup> Ant 使用“目标”一词来表示我们在本章中所说的“任务”,并使用“任务”一词来指代我们称之为“命令”。

```
mkdir -p dist/lib/jar cf
dist/lib/MyProject-$(date -iso-8601).jar build/*
```

当语法被剥离时,构建文件和构建脚本实际上并没有太大区别。但通过这样做我们已经收获颇丰。我们可以在其他目录中创建新的构建文件并将它们链接在一起。我们可以轻松地以任意复杂的方式添加依赖于现有任务的新任务。我们只需要将单个任务的名称传递给ant命令行工具,它就会负责确定需要运行的所有内容。

Ant 是一款非常古老的软件,最初发布于 2000 年。如今,很多人并不认为它是“现代”构建系统!Maven 和 Gradle 等其他工具在随后的几年中对 Ant 进行了改进,并通过添加自动管理外部依赖项和更简洁的语法(无任何 XML)等功能从根本上取代了 Ant。但这些新系统的本质保持不变:它们允许工程师以原则性和模块化的方式将构建脚本编写为任务,并提供用于执行这些任务和管理它们之间依赖关系的工具。

#### 基于任务的构建系统的阴暗面由于这些工具本质

上让工程师将任何脚本定义为一项任务,因此它们功能极其强大,允许您使用它们完成几乎任何您能想到的事情。但这种强大功能也有缺点,随着构建脚本变得越来越复杂,基于任务的构建系统可能变得难以使用。这种系统的问题在于,它们实际上赋予了工程师太多权力,而没有赋予系统足够的权力。由于系统不知道脚本在做什么,因此性能会受到影响,因为它必须非常保守地安排和执行构建步骤。并且系统无法确认每个脚本都在执行其应该执行的操作,因此脚本往往变得越来越复杂,最终成为需要调试的另一件事。

并行化构建步骤的难度。现代开发工作站通常非常强大,具有多个核心,理论上应该能够并行执行多个构建步骤。但是,基于任务的系统通常无法并行化任务执行,即使看起来它们应该能够做到。假设任务 A 依赖于任务 B 和 C。由于任务 B 和 C 彼此没有依赖关系,那么同时运行它们是否安全,以便系统可以更快地转到任务 A?也许可以,如果它们不接触任何相同的资源。但也可能不行 也许两者都使用相同的文件来跟踪它们的状态,同时运行它们会导致冲突。系统通常无法知道,因此要么必须冒这些冲突的风险(导致罕见但很难调试的构建问题),要么必须将整个构建限制在单个进程中的单个线程上运行。这可能会极大地浪费强大的开发人员机器,并且完全排除了在多台机器上分布构建的可能性。

执行增量构建有困难。良好的构建系统将允许工程师执行可靠的增量构建,这样一小段更改就不需要从头开始重建整个代码库。如果构建系统速度慢且由于上述原因无法并行化构建步骤,这一点尤其重要。

但不幸的是,基于任务的构建系统在这里也遇到了困难。由于任务可以做任何事情,因此通常没有办法检查它们是否已经完成。许多任务只是获取一组源文件并运行编译器来创建一组二进制文件;因此,如果底层源文件没有更改,则无需重新运行它们。但如果没有任何其他信息,系统就无法肯定地说这一点。也许任务下载了一个可能已经更改的文件,或者它写入的时间戳在每次运行时都可能不同。为了保证正确性,系统通常必须在每次构建期间重新运行每个任务。

一些构建系统尝试通过让工程师指定需要重新运行任务的条件来实现增量构建。有时这是可行的,但通常这是一个比看起来更棘手的问题。例如,在 C++ 等允许文件直接被其他文件包含的语言中,如果不解析输入源,就不可能确定必须监视更改的整个文件集。工程师往往会选择捷径,而这些捷径可能会导致罕见且令人沮丧的问题,即任务结果被重复使用,即使它不应该被重复使用。当这种情况频繁发生时,工程师会养成在每次构建之前运行清理以获得全新状态的习惯,这完全违背了增量构建的初衷。弄清楚何时需要重新运行任务是非常微妙的,这项工作由机器而不是人类来处理会更好。

维护和调试脚本困难。最后,基于任务的构建系统强加的构建脚本通常很难使用。尽管它们通常受到的审查较少,但构建脚本就像正在构建的系统一样是代码,并且很容易隐藏错误。以下是使用基于任务的构建系统时非常常见的一些错误示例:

- 任务 A 依赖于任务 B 来生成特定文件作为输出。任务 B 的所有者没有意识到其他任务依赖于它,因此他们将其更改为在其他位置生成输出。直到有人尝试运行任务 A 并发现它失败时,才会检测到这种情况。
- 任务 A 依赖于任务 B,而任务 B 又依赖于任务 C,任务 C 生成任务 A 所需的特定文件作为输出。任务 B 的所有者决定不再需要依赖任务 C,这导致任务 A 失败,尽管任务 B 根本不关心任务 C!
- 新任务的开发人员意外地对运行该任务的机器做出了假设,例如工具的位置或特定

环境变量。该任务在他们的机器上可以运行,但每当其他开发人员尝试执行该任务时都会失败。· 任务包含一个不确定的组件,例如从互联网

下载文件或向构建添加时间戳。现在,人们每次运行构建时都会得到不同的结果,这意味着工程师并不总是能够重现和修复彼此的故障或自动构建系统上发生的故障。

- 具有多个依赖项的任务可能会产生竞争条件。如果任务 A 同时依赖于任务 B 和任务 C,并且任务 B 和 C 都修改同一个文件,则任务 A 将根据任务 B 和 C 中哪一个先完成而获得不同的结果。

在本文列出的基于任务的框架中,没有通用的方法来解决这些性能、正确性或可维护性问题。只要工程师能够编写在构建期间运行的任意代码,系统就不可能拥有足够的信息来始终能够快速正确地运行构建。为了解决这个问题,我们需要将一些权力从工程师手中夺走,交给系统,并重新定义系统的角色,不再是运行任务,而是生产工件。这是 Google 使用 Blaze 和 Bazel 所采用的方法,将在下一节中介绍。

### 基于工件的构建系统要设计一个更好的构建系

统,我们需要退一步思考。早期系统的问题在于,它们让工程师自己定义任务,从而赋予了他们太多权力。也许我们可以不让工程师定义任务,而是让系统定义少量任务,工程师可以以有限的方式进行配置。我们或许可以从本章的名称中推断出最重要的任务的名称:构建系统的主要任务应该是构建代码。工程师仍然需要告诉系统要构建什么,但如何构建将留给系统。

这正是 Blaze 及其衍生的其他基于工件的构建系统 (包括 Bazel、Pants 和 Buck) 所采用的方法。与基于任务的构建系统一样,我们仍然有构建文件,但这些构建文件的内容非常不同。

Blaze 中的构建文件不是图灵完备脚本语言中描述如何生成输出的命令集,而是描述要构建的一组工件、它们的依赖项以及影响构建方式的一组有限选项的声明性清单。当工程师在命令行上运行 Blaze 时,他们会指定要构建的一组目标 (“什么”),而 Blaze 负责配置、运行和安排编译步骤 (“如何”)。由于构建系统现在可以完全控制何时运行哪些工具,因此它可以

更加强有力的保证,使其更加高效,同时仍然保证  
正确性。

函数式视角我们很容易将基于工

件的构建系统与函数式编程进行类比。传统的命令式编程语言（例如 Java、C 和 Python）指定要逐个执行的语句列表,就像基于任务的构建系统让程序员定义一系列要执行的步骤一样。相比之下,函数式编程语言（例如 Haskell 和 ML）的结构更像一系列数学等式。在函数式语言中,程序员描述要执行的计算,但将何时以及如何执行该计算的细节留给编译器。这映射到在基于工件的构建系统中声明清单并让系统确定如何执行构建的想法。

许多问题无法用函数式编程轻松表达,但那些可以轻易表达的问题却能从中受益匪浅:该语言通常能够轻松地并行化此类程序,并对其正确性做出强有力的保证,而这在命令式语言中是不可能的。使用函数式编程表达的最简单问题是那些只需使用一系列规则或函数将一个数据转换为另一个数据的问题。这正是构建系统:整个系统实际上是一个数学函数,它将源文件（和编译器等工具）作为输入并生成二进制文件作为输出。因此,基于函数式编程原则构建构建系统效果良好也就不足为奇了。

使用 Bazel 进行具体化。Bazel 是 Google 内部构建工具 Blaze 的开源版本,是基于工件的构建系统的一个很好的例子。以下是 Bazel 中的构建文件（通常名为 BUILD）：

```
java_binary(name = "MyBinary", srcs = ["MyBinary.java"], deps = [":mylib"],)
```

```
java_library(name = "mylib", srcs = ["MyLibrary.java", "MyHelper.java"], visibility = ["//java/com/example/myproduct:_subpackages_"], deps = ["//java/com/example/common", "//java/com/example/myproduct/otherlib", "@com_google_common_guava/guava/jar",
```

```
],
)
```

在 Bazel 中,BUILD 文件定义目标。这里的两种目标是java\_binary和java\_library。每个目标都对应一个可由系统创建的工件:二进制目标生成可直接执行的二进制文件,而库目标生成可由二进制文件或其他库使用的库。每个目标都有一个名称(定义它在命令行和其他目标中的引用方式)、srcs(定义必须编译才能为目标创建工件的源文件)和deps(定义必须在此目标之前构建并链接到其中的其他目标)。依赖项可以位于同一个包中(例如, MyBinary对“:mylib”的依赖),也可以位于同一源层次结构中的不同包中(例如, mylib对“//java/com/example/common”的依赖),也可以位于源层次结构之外的第三方工件中(例如, mylib对“@com\_google\_common\_guava\_guava//jar”的依赖)。每个源层次结构称为工作区,由根目录中的特殊WORKSPACE文件标识。

与 Ant一样,用户使用 Bazel 的命令行工具执行构建。要构建MyBinary目标,用户将运行bazel build :MyBinary。在干净的存储库中首次输入该命令时,Bazel 将执行以下操作:

1. 解析工作区中的每个 BUILD 文件以创建依赖关系图  
在文物之中。
2. 使用该图确定MyBinary 的传递依赖关系;即,递归地确定MyBinary所依赖的每个目标以及这些目标所依赖的每个目标。
3. 按顺序构建(或下载外部依赖项)每个依赖项。Bazel 首先构建每个没有其他依赖项的目标,并跟踪每个目标仍需要构建哪些依赖项。一旦构建了目标的所有依赖项,Bazel 便开始构建该目标。此过程持续进行,直到构建了MyBinary 的每个传递依赖项。
4. 构建MyBinary以生成最终的可执行二进制文件,该二进制文件链接所有依赖项。  
在步骤 3 中构建的密度。

从根本上讲,这里发生的事情似乎与使用基于任务的构建系统时发生的事情没有太大不同。事实上,最终结果是相同的二进制文件,生成它的过程涉及分析一系列步骤以找到它们之间的依赖关系,然后按顺序运行这些步骤。但存在关键差异。第一个出现在步骤 3 中:因为 Bazel 知道每个目标只会生成一个 Java 库,所以它知道它所做的就是运行 Java 编译器而不是任意的用户定义脚本,所以它知道运行它是安全的

并行执行这些步骤。与在多核机器上一次构建一个目标相比,这可以带来一个数量级的性能提升,而且这仅仅是因为基于工件的方法让构建系统负责自己的执行策略,这样它就可以对并行性做出更强的保证。

但是,其好处并不仅限于并行。当开发人员第二次输入`bazel build :MyBinary`而不做任何更改时,这种方法给我们带来的另一个好处就变得显而易见了:Bazel 将在不到 1 秒的时间内退出,并显示一条消息,提示目标已是最新的。这要归功于我们之前讨论过的函数式编程范式 - Bazel 知道每个目标都是运行 Java 编译器的结果,并且它知道 Java 编译器的输出只取决于其输入,因此只要输入没有改变,输出就可以重用。并且这种分析在各个级别都有效;如果`MyBinary.java`发生变化,Bazel 就知道要重建`MyBinary`但重用`mylib`。如果`//java/com/example/common` 的源文件发生变化,Bazel 就知道要重建该库、`mylib`和`MyBinary`,但重用`//java/com/example/myproduct/otherlib`。由于 Bazel 了解它在每个步骤中运行的工具的属性,因此它每次能够只重建最少的工件集,同时保证不会产生过时的构建。

将构建过程重新定义为工件而不是任务,这种做法虽然微妙,但效果却十分强大。通过减少程序员所能获得的灵活性,构建系统可以更清楚地了解构建过程中每一步所做的事情。它可以利用这些知识,通过并行化构建过程并重用其输出,使构建过程更加高效。但这实际上只是第一步,这些并行和重用的构建块将构成分布式、高度可扩展的构建系统的基础,我们将在后面讨论。

### 其他巧妙的 Bazel 技巧基

于 Artifact 的构建系统从根本上解决了基于任务的构建系统固有的并行性和重用性问题。但之前出现过的一些问题我们还没有解决。Bazel 有巧妙的方法来解决这些问题,在继续之前我们应该先讨论一下。

工具作为依赖项。我们之前遇到的一个问题是,构建依赖于我们机器上安装的工具,由于工具版本或位置不同,跨系统重现构建可能很困难。当您的项目使用的语言需要根据构建或编译平台而不同的工具(例如 Windows 与 Linux)时,问题会变得更加困难,并且每个平台都需要一组略有不同的工具来完成相同的工作。

Bazel 通过将工具视为每个目标的依赖项来解决此问题的第一部分。工作区中的每个`java_library`都隐式依赖于 Java 编译器,该编译器默认认为众所周知的编译器,但可以在

工作区级别。每当 Blaze 构建 `java_library` 时,它都会检查以确保指定的编译器在已知位置可用,如果不可用,则下载它。就像任何其他依赖项一样,如果 Java 编译器发生变化,则依赖于它的每个工件都需要重建。Bazel 中定义的每种类型的目标都使用相同的策略来声明它需要运行的工具,确保无论运行的系统上存在什么,Bazel 都能够引导它们。

Bazel 通过使用[工具链](#)解决了问题的第二部分,即平台独立性。目标并非直接依赖于其工具,而是依赖于工具链的类型。工具链包含一组工具和其他属性,定义如何在特定平台上构建目标类型。工作区可以根据主机和目标平台定义要用于工具链类型的特定工具链。有关更多详细信息,请参阅 Bazel 手册。

扩展构建系统。Bazel 提供了多种流行编程语言的目标,但工程师们总是希望做更多的事情 基于任务的系统的好处之一是它们可以灵活地支持任何类型的构建过程,最好不要在基于工件的构建系统中放弃这一点。幸运的是,Bazel 允许通过[添加自定义规则来扩展其支持的目标类型](#)。

要在 Bazel 中定义规则,规则作者需要声明规则所需的输入(以 BUILD 文件中传递的属性的形式)以及规则产生的固定输出集。作者还定义该规则将生成的操作。

每个操作都声明其输入和输出,运行特定的可执行文件或将特定的字符串写入文件,并且可以通过其输入和输出连接到其他操作。这意味着操作是构建系统中最级别的可组合单元 - 只要操作仅使用其声明的输入和输出,它就可以做任何它想做的事情,而 Bazel 会负责安排操作并根据需要缓存其结果。

由于没有办法阻止动作开发人员在其动作中引入不确定性过程之类的操作,因此该系统并非万无一失。

但这种情况在实践中并不常见,将滥用的可能性一直推到操作层面可以大大减少出错的机会。支持许多常见语言和工具的规则在网上广泛可用,大多数项目永远不需要定义自己的规则。即使对于那些需要定义规则的项目,规则定义也只需要在存储库中的一个中心位置进行定义,这意味着大多数工程师将能够使用这些规则,而不必担心它们的实现。

隔离环境。操作听起来可能会遇到与其他系统中的任务相同的问题 - 是否仍然可以编写既写入同一文件又最终相互冲突的操作?实际上,Bazel 通过使用[沙盒使这些冲突不可能发生](#)。在支持的系统上,每个操作都是独立的

通过文件系统沙箱,每个操作都与其他操作隔离。实际上,每个操作只能看到文件系统的受限视图,其中包括它声明的输入和它生成的任何输出。这是由 Linux 上的 LXC 等系统强制执行的,Docker 背后的技术也是如此。这意味着操作不可能相互冲突,因为它们无法读取任何未声明的文件,并且它们写入但未声明的任何文件将在操作完成时被丢弃。Bazel 还使用沙箱来限制操作通过网络进行通信。

使外部依赖项具有确定性。还有一个问题:构建系统通常需要从外部源下载依赖项(无论是工具还是库),而不是直接构建它们。这可以在示例中通过@com\_google\_common\_guava//jar依赖项看到,它从 Maven 下载 JAR 文件。

依赖当前工作区之外的文件是有风险的。这些文件可能随时发生变化,因此可能需要构建系统不断检查它们是否是最新的。如果远程文件发生变化而工作区源代码没有相应变化,也可能导致无法重现的构建。构建可能今天可以运行,明天却会毫无原因地失败,这是由于未注意到的依赖项更改造成的。

最后,当外部依赖项由第三方拥有时,它可能会带来巨大的安全风险:<sup>4</sup>如果攻击者能够渗透到第三方服务器,他们就可以用自己设计的东西替换依赖文件,从而可能让他们完全控制你的构建环境及其输出。

根本问题是,我们希望构建系统能够识别这些文件,而无需将它们签入源代码控制。更新依赖项应该是一个有意识的选择,但该选择应该在一个中心位置进行一次,而不是由个别工程师管理或由系统自动进行。这是因为即使采用“Live at Head”模型,我们仍然希望构建具有确定性,这意味着如果您签出上周的提交,您应该看到当时的依赖项,而不是现在的依赖项。

Bazel 和其他一些构建系统通过要求工作区范围内的清单文件来解决这个问题,该文件列出了工作区中每个外部依赖项的加密哈希值。<sup>5</sup>哈希值是一种简洁的方式来唯一地表示文件,而无需将整个文件签入源代码控制。每当从工作区引用新的外部依赖项时,该依赖项的哈希值都会手动或自动添加到清单中。当 Bazel 运行构建时,它会根据清单中定义的预期哈希值检查其缓存依赖项的实际哈希值,并且只有当哈希值不同时才重新下载文件。

---

<sup>4</sup>这样的“软件供应链”袭击变得越来越常见。

<sup>5</sup> Go 最近增加了对使用完全相同系统的模块的初步支持。

如果我们下载的工件的哈希值与清单中声明的哈希值不同,则构建将失败,除非清单中的哈希值已更新。这可以自动完成,但必须先批准该更改并将其签入源代码控制,然后构建才能接受新的依赖项。这意味着始终会记录依赖项的更新时间,并且外部依赖项的更改必须在工作区源中没有相应更改的情况下进行。这还意味着,在签出旧版本的源代码时,构建保证使用与签入该版本时相同的依赖项(否则,如果这些依赖项不再可用,构建将失败)。

当然,如果远程服务器不可用或开始提供损坏的数据,这仍然会是个问题。如果您没有该依赖项的另一个副本,这可能会导致您的所有构建开始失败。为了避免这个问题,我们建议,对于任何非平凡的项目,您将其所有依赖项镜像到您信任和控制的服务器或服务上。否则,即使签入的哈希保证了构建系统的安全性,您的构建系统的可用性也将永远取决于第三方。

## 分布式构建

Google 的代码库非常庞大,有超过 20 亿行代码,依赖链会变得非常深。在 Google,即便是简单的二进制文件也常常依赖于数以万计的构建目标。在这种规模下,在单台机器上在合理的时间内完成构建根本是不可能的;没有构建系统可以绕过强加于机器硬件的基本物理定律。实现这一点的唯一方法是使用支持分布式构建的构建系统,其中系统完成的工作单元分布在任意数量的可扩展机器上。假设我们将系统的工作分解成足够小的单元(稍后会详细介绍),这将使我们能够以我们愿意支付的速度完成任何规模的任何构建。这种可扩展性是我们通过定义基于工件的构建系统而一直努力实现的目标。

### 远程缓存最简单

的分布式构建类型是仅利用远程缓存,如图 18-2 所示。

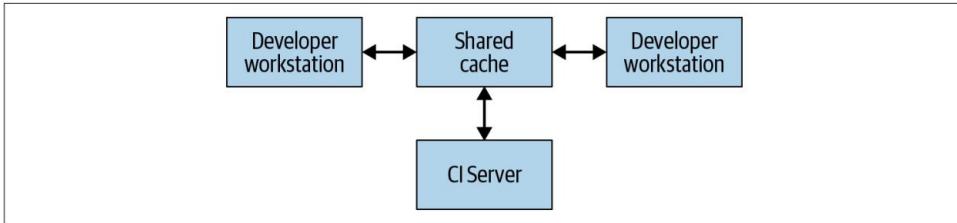


图 18-2. 显示远程缓存的分布式构建

每个执行构建的系统（包括开发人员工作站和持续集成系统）都共享对通用远程缓存服务的引用。

该服务可能是快速的本地短期存储系统（如 Redis）或云服务（如 Google Cloud Storage）。每当用户需要构建工件（无论是直接构建还是作为依赖项构建）时，系统都会首先检查远程缓存以查看该工件是否已存在。如果存在，它可以下载工件而不是构建它。如果不存在，系统将自行构建工件并将结果上传回缓存。这意味着不经常更改的低级依赖项可以构建一次并在用户之间共享，而不必由每个用户重新构建。在 Google，许多工件都是从缓存中提供的，而不是从头开始构建，这大大降低了运行构建系统的成本。

要使远程缓存系统正常工作，构建系统必须保证构建完全可重现。也就是说，对于任何构建目标，必须能够确定该目标的输入集，以便同一组输入在任何机器上都会产生完全相同的输出。这是确保下载工件的结果与自己构建工件的结果相同的唯一方法。

幸运的是，Bazel 提供了这种保证，因此支持[远程缓存](#)。请注意，这要求缓存中的每个工件都以其目标和其输入的哈希值作为关键字 - 这样，不同的工程师可以同时对同一目标进行不同的修改，并且远程缓存将存储所有生成的工件并以适当的方式提供它们而不会发生冲突。

当然，要从远程缓存中获益，下载工件的速度必须比构建工件的速度快。但情况并非总是如此，尤其是当缓存服务器距离构建机器较远时。Google 的网络和构建系统经过精心调整，能够快速共享构建结果。在组织中配置远程缓存时，请谨慎考虑网络延迟并进行实验以确保缓存确实能提高性能。

### 远程执行

远程缓存不是真正的分布式构建。如果缓存丢失，或者你进行了需要重建所有内容的低级更改，你仍然需要在机器上本地执行整个构建。真正的目标是支持远程执行，其中

实际的构建工作可以由任意数量的工人完成。

图 18-3 描绘了一个远程执行系统。

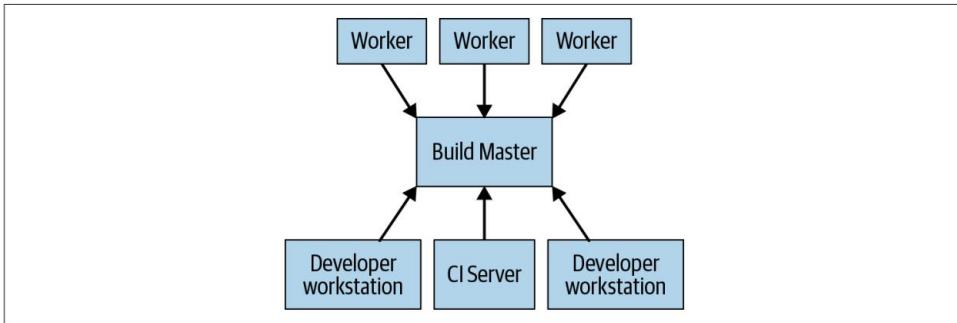


图 18-3. 远程执行系统

每个用户机器上运行的构建工具（用户可以是人工工程师或自动构建系统）将请求发送到中央构建主机。构建主机将请求分解为组件操作，并通过可扩展的工作人员池安排这些操作的执行。每个工作人员根据用户指定的输入执行要求的操作，并写出生成的工作。

这些工件在执行需要它们的操作的其他机器之间共享，直到可以生成最终输出并发送给用户。

实现此类系统最棘手的部分是管理工作器、主服务器和用户本地计算机之间的通信。工作器可能依赖于其他工作器生成的中间产物，最终输出需要发送回用户本地计算机。为此，我们可以基于前面描述的分布式缓存进行构建，让每个工作器将其结果写入缓存并从缓存中读取其依赖项。主服务器会阻止工作器继续执行，直到它们所依赖的所有内容都完成，在这种情况下，它们将能够从缓存中读取输入。最终产品也会被缓存，允许本地计算机下载它。请注意，我们还需要一种单独的方法来导出用户源代码树中的本地更改，以便工作器可以在构建之前应用这些更改。

为了实现这一点，前面描述的基于工件的构建系统的所有部分都需要结合在一起。构建环境必须完全自描述，这样我们就可以无需人工干预即可启动工作程序。构建过程本身必须完全独立，因为每个步骤可能在不同的机器上执行。输出必须完全确定，这样每个工作程序都可以信任它从其他工作程序收到的结果。对于基于任务的系统来说，提供这样的保证极其困难，这使得在其基础上构建可靠的远程执行系统几乎是不可能的。

Google 的分布式构建。自 2008 年以来,Google 一直在使用一种同时采用远程缓存和远程执行的分布式构建系统,如图18-4 所示。

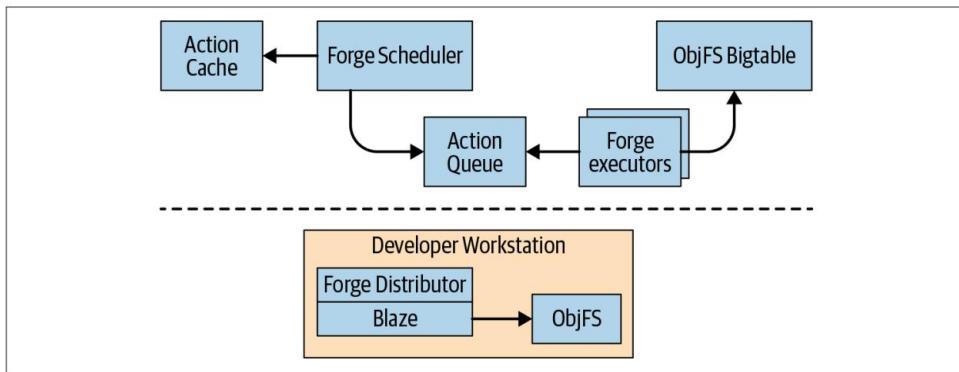


图 18-4. Google 的分布式构建系统

Google 的远程缓存称为 ObjFS。它由一个后端组成,该后端将构建输出存储在Bigtables中分布在我们的生产机器群中,并在每台开发人员的机器上运行一个名为 objfsd 的前端 FUSE 守护程序。FUSE 守护程序允许工程师浏览构建输出,就像它们是存储在工作站上的普通文件一样,但文件内容只会按需下载用户直接请求的少数文件。按需提供文件内容大大减少了网络和磁盘使用量,并且系统能够以两倍的速度构建与我们将所有构建输出存储在开发人员的本地磁盘上相比。

Google 的远程执行系统称为 Forge。Blaze 中的 Forge 客户端称为 Distributor,它将每个操作的请求发送到我们数据中心中运行的作业,称为 Scheduler。Scheduler 维护操作结果的缓存,如果系统的其他用户已经创建了该操作,则允许它立即返回响应。如果没有,它将操作放入队列中。大量 Executor 作业不断从此队列读取操作、执行它们,并将结果直接存储在 ObjFS Bigtables 中。这些结果可供执行器用于将来的操作,或由最终用户通过 objfsd 下载。

最终结果是,该系统可扩展,以有效支持 Google 执行的所有构建。Google 的构建规模非常庞大: Google 每天运行数百万次构建,执行数百万个测试用例,并从数十亿行源代码中生成 PB 级的构建输出。这样的系统不仅让我们的工程师能够快速构建复杂的代码库,还使我们能够实现大量依赖于我们构建的自动化工具和系统。我们花了很多年的时间开发这个系统,但如今开源工具随处可见,任何组织都可以实现类似的系统。尽管这可能需要一些时间

和精力来部署这样的构建系统,最终结果对于工程师来说可能是真正神奇的,并且往往值得付出努力。

## 时间、规模、权衡

构建系统的目的是使代码更容易大规模和长期地使用。

和软件工程中的所有事情一样,选择使用哪种构建系统也需要权衡利弊。使用 shell 脚本或直接调用工具的 DIY 方法仅适用于不需要长期处理代码更改的最小项目,或者像 Go 这样具有内置构建系统的语言。

选择基于任务的构建系统而不是依赖 DIY 脚本可以大大提高项目的扩展能力,让您能够自动执行复杂的构建,并更轻松地在机器上重现这些构建。但代价是,您需要真正开始考虑构建的结构,并处理编写构建文件的开销(尽管自动化工具通常可以提供帮助)。

对于大多数项目来说,这种权衡往往是值得的,但对于特别琐碎的项目(例如,包含在单个源文件中的项目),这种开销可能不会给你带来太多好处。

随着项目规模的进一步扩大,基于任务的构建系统开始遇到一些基本问题,而这些问题可以通过使用基于工件的构建系统来解决。此类构建系统开启了一个全新的规模级别,因为大型构建现在可以分布在许多机器上,成千上万的工程师可以更加确定他们的构建是一致且可重复的。与本书中的许多其他主题一样,这里的权衡是缺乏灵活性:基于工件的系统不允许您使用真实的编程语言编写通用任务,而是要求您在系统约束内工作。对于从一开始就设计为使用基于工件的系统的项目来说,这通常不是问题,但是从现有的基于任务的系统迁移可能会很困难,而且如果构建在速度或正确性方面没有出现问题,那么迁移并不总是值得的。

更改项目的构建系统可能代价高昂,而且随着项目规模的扩大,成本也会增加。这就是为什么 Google 认为几乎每个新项目从一开始就采用基于工件的构建系统(如 Bazel)都会受益匪浅。在 Google 内部,从小型实验项目到 Google Search,基本上所有代码都是使用 Blaze 构建的。

## 处理模块和依赖关系

使用基于工件的构建系统(如 Bazel)的项目被分解为一组模块,模块通过 BUILD 文件表达彼此之间的依赖关系。正确的

这些模块和依赖项的组织会对构建系统的性能和维护所需的工作量产生巨大的影响。

### 使用细粒度模块和 1:1:1 规则在构建基于工件的构建时出现的第一步

一个问题是决定单个模块应包含多少功能。在 Bazel 中，“模块”由指定可构建单元（如 `java_library` 或 `go_binary`）的目标表示。在一个极端，整个项目可以包含在单个模块中，方法是将一个 BUILD 文件放在根目录并递归地将该项目的所有源文件组合在一起。在另一个极端，几乎每个源文件都可以制成自己的模块，实际上要求每个文件在 BUILD 文件中列出它所依赖的所有其他文件。

大多数项目都介于这两个极端之间，选择涉及性能和可维护性之间的权衡。对整个项目使用单个模块可能意味着您永远不需要接触 BUILD 文件（除了添加外部依赖项时），但这意味着构建系统将始终需要一次性构建整个项目。这意味着它将无法并行化或分发构建的各个部分，也无法缓存已经构建的部分。

每个文件一个模块则相反：构建系统在缓存和安排构建步骤方面具有最大的灵活性，但工程师在更改哪些文件引用哪些文件时，需要花费更多的精力来维护依赖项列表。

尽管确切的粒度因语言而异（甚至通常在同一语言中也不同），但 Google 倾向于使用比在基于任务的构建系统中编写的模块小得多的模块。Google 的典型生产二进制文件可能依赖于数万个目标，甚至中等规模的团队也可以在其代码库中拥有数百个目标。对于像 Java 这样具有强大的内置打包概念的语言，每个目录通常包含一个包、目标和 BUILD 文件（Pants 是另一个基于 Blaze 的构建系统，它将此称为 [1:1:1 规则](#)）。打包约定较弱的语言通常会在每个 BUILD 文件中定义多个目标。

较小的构建目标的好处在大规模情况下开始真正显现出来，因为它们可以加快分布式构建速度，并且重建目标的频率更低。在测试开始后，这些优势变得更加引人注目，因为更细粒度的目标意味着构建系统可以更智能地只运行可能受任何给定更改影响的有限测试子集。由于 Google 相信使用较小目标的系统优势，我们通过投资工具来自动管理 BUILD 文件以避免给开发人员带来负担，从而在减轻负面影响方面取得了一些进展。其中许多 [工具](#) 现在是开源的。

## 最小化模块可见性Bazel 和其他构建系

统允许每个目标指定可见性 :指定哪些其他目标可以依赖它的属性。目标可以是公共的 ,在这种情况下 ,它们可以被工作区中的任何其他目标引用 ;私有的 ,在这种情况下 ,它们只能从同一个 BUILD 文件内引用 ;或者仅对明确定义的其他目标列表可见。可见性本质上与依赖性相反 :如果目标 A 想要依赖目标 B ,目标 B 必须使自己对目标 A 可见。

就像大多数编程语言一样 ,通常最好尽可能地降低可见性。通常 ,只有当目标代表 Google 任何团队都可以使用的广泛使用的库时 ,Google 的团队才会将目标公开。需要其他人在使用其代码之前与他们协调的团队将维护客户目标的白名单作为其目标的可见性。每个团队的内部实现目标将仅限于团队拥有的目录 ,并且大多数 BUILD 文件只有一个非私有目标。

## 管理依赖关系模块需要能够

相互引用。将代码库分解为细粒度模块的缺点是您需要管理这些模块之间的依赖关系 (尽管工具可以帮助自动化此操作) 。表达这些依赖关系通常最终会成为 BUILD 文件中的大部分内容。

### 内部依赖关系在分解为

细粒度模块的大型项目中 ,大多数依赖关系可能是内部的 ;也就是说 ,在同一个源存储库中定义和构建的另一个目标上。

内部依赖项与外部依赖项的不同之处在于 ,它们是从源代码构建的 ,而不是在运行构建时作为预构建工件下载的。这也意味着内部依赖项没有 “版本” 的概念 目标及其所有内部依赖项始终在存储库中的同一提交 / 修订中构建。

关于内部依赖关系 ,需要谨慎处理的一个问题是如何处理传递依赖关系 (图 18-5) 。假设目标 A 依赖于目标 B ,而目标 B 又依赖于公共库目标 C 。那么目标 A 是否应该能够使用目标 C 中定义的类 ?

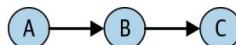


图 18-5. 传递依赖关系

就底层工具而言,这样做没有问题;在构建目标 A 时,B 和 C 都将链接到目标 A,因此 A 知道 C 中定义的任何符号。Blaze 允许这样做很多年,但随着 Google 的发展,我们开始发现问题。假设重构 B,使其不再需要依赖 C。如果随后删除 B 对 C 的依赖,A 以及通过对 B 的依赖使用 C 的任何其他目标都会中断。实际上,目标的依赖关系成为其公共合同的一部分,永远无法安全地更改。这意味着依赖关系会随着时间的推移而积累,Google 的构建速度开始变慢。

谷歌最终通过在 Blaze 中引入“严格传递依赖模式”解决了这个问题。在此模式下,Blaze 会检测目标是否试图引用符号而不直接依赖它,如果是,则会失败并显示错误和可用于自动插入依赖项的 shell 命令。将这一变化推广到谷歌的整个代码库并重构我们数百万个构建目标中的每一个以明确列出它们的依赖项是一项多年的努力,但这是值得的。由于目标具有更少的不必要的依赖项,<sup>6</sup>我们的构建速度现在大大加快,工程师可以删除不需要的依赖项,而不必担心破坏依赖它们的目标。

像往常一样,强制执行严格的传递依赖关系需要权衡。它使构建文件变得更加冗长,因为现在需要在许多地方明确列出常用库,而不是偶然引入,而且工程师需要花费更多精力将依赖关系添加到 BUILD 文件中。从那时起,我们开发了一些工具来减少这种工作量,这些工具可以自动检测许多缺失的依赖关系并将它们添加到 BUILD 文件中,而无需任何开发人员干预。但即使没有这样的工具,我们也发现,随着代码库的扩展,这种权衡是值得的:明确将依赖关系添加到 BUILD 文件是一次性成本,但只要构建目标存在,处理隐式传递依赖关系就会导致持续的问题。Bazel 强制执行严格的传递依赖关系默认在 Java 代码上。

### 外部依赖项如果依赖

项不是内部依赖项,则它必须是外部依赖项。外部依赖项是那些在构建系统之外构建和存储的工件。依赖项直接从工件存储库(通常通过互联网访问)导入并按原样使用,而不是从源代码构建。外部依赖项和内部依赖项之间的最大区别之一是外部依赖项有版本,并且这些版本独立于项目的源代码而存在。

---

<sup>6</sup>当然,实际上删除这些依赖项是一个完全独立的过程。但要求每个目标

明确声明它使用了什么是至关重要的第一步。有关 Google 如何进行此类大规模更改的更多信息,请参阅第 22 章。

自动与手动依赖管理。构建系统可以允许手动或自动管理外部依赖项的版本。手动管理时,构建文件会明确列出要从工件存储库下载的版本,通常使用语义版本字符串例如“1.1.4”。当自动管理时,源文件会指定一系列可接受的版本,而构建系统始终会下载最新版本。例如,Gradle 允许将依赖项版本声明为“1.+”,以指定只要主要版本为 1,依赖项的任何次要版本或修补程序版本都是可接受的。

自动管理依赖项对于小型项目来说可能很方便,但对于规模不小的项目或由多名工程师共同开发的项目来说,这通常会导致灾难。自动管理依赖项的问题在于您无法控制版本更新的时间。无法保证外部各方不会进行重大更新(即使他们声称使用语义版本控制),因此今天可以正常运行的构建明天可能会出现问题,而且没有简单的方法来检测更改的内容或将回滚到正常工作状态。

即使构建没有中断,也可能会存在无法追踪的细微行为或性能变化。

相比之下,由于手动管理的依赖项需要更改源代码控制,因此可以轻松发现和回滚它们,并且可以检出旧版本的存储库以使用旧依赖项进行构建。Bazel 要求手动指定所有依赖项的版本。即使在中等规模下,手动版本管理的开销也非常值得,因为它提供了稳定性。

单一版本规则。库的不同版本通常由不同的工件表示,因此理论上,没有理由不能在构建系统中以不同的名称声明同一外部依赖项的不同版本。

这样,每个目标都可以选择它想要使用的依赖项的版本。

Google 发现这在实践中会导致很多问题,因此我们实施严格的**单一版本规则**适用于我们内部代码库中的所有第三方依赖项。

允许多个版本的最大问题是菱形依赖问题。假设目标 A 依赖于目标 B 和外部库的 v1。如果稍后重构目标 B 以添加对同一外部库的 v2 的依赖,则目标 A 将中断,因为它现在隐式依赖于同一库的两个不同版本。实际上,从目标向具有多个版本的任何第三方库添加新依赖永远是不安全的,因为该目标的任何用户可能已经依赖于不同的版本。遵循单一版本规则可以避免这种冲突 - 如果目标添加了对第三方库的依赖,则任何现有的依赖都已经在同一版本上,因此它们可以愉快地共存。

我们将在第 21 章中在大型 monorepo 的背景下进一步研究这一点。

传递性外部依赖。处理外部依赖的传递性依赖可能特别困难。许多工件存储库（如 Maven Central）允许工件指定对存储库中其他工件特定版本的依赖。Maven 或 Gradle 等构建工具通常会默认递归下载每个传递性依赖，这意味着在项目中添加单个依赖可能会导致总共下载数十个工件。

这非常方便：当添加对新库的依赖时，必须追踪该库的每个传递依赖项并手动添加它们，这将非常麻烦。但这也有一个巨大的缺点：由于不同的库可以依赖同一个第三方库的不同版本，这种策略必然违反单一版本规则并导致菱形依赖问题。如果您的目标依赖于两个使用同一依赖项的不同版本的外部库，那么您无法知道会得到哪一个。这也意味着，如果新版本开始引入某些依赖项的冲突版本，则更新外部依赖项可能会导致整个代码库中看似不相关的故障。

因此，Bazel 不会自动下载传递依赖项。不幸的是，没有灵丹妙药。Bazel 的替代方案是要求一个全局文件，列出存储库的每个外部依赖项以及整个存储库中用于该依赖项的显式版本。幸运的是，[Bazel 提供了工具](#)能够自动生成包含一组 Maven 工件的传递依赖关系的文件。此工具可以运行一次以生成项目的初始 WORKSPACE 文件，然后可以手动更新该文件以调整每个依赖项的版本。

再次强调，这里的“选择”是在便利性和可扩展性之间。小型项目可能更愿意不必担心自己管理传递依赖项，并且可能能够使用自动传递依赖项。随着组织和代码库的增长，这种策略变得越来越没有吸引力，冲突和意外结果变得越来越频繁。在更大的规模下，手动管理依赖项的成本远低于处理自动依赖项管理引起的问题的成本。

使用外部依赖项缓存构建结果。外部依赖项通常由发布稳定版本库的第三方提供，可能不提供源代码。一些组织可能还会选择将自己的部分代码作为工件提供，允许其他代码片段将它们作为第三方依赖项而不是内部依赖项来依赖。如果工件构建速度慢但下载速度快，这在理论上可以加快构建速度。

然而，这也带来了很多开销和复杂性：需要有人负责构建每个工件并将其上传到工件

存储库,客户端需要确保他们保持最新版本。

调试也变得更加困难,因为系统的不同部分将从存储库中的不同点构建,并且不再存在源树的一致视图。

解决工件构建时间过长问题的更好方法是使用支持远程缓存的构建系统,如前所述。这样的构建系统会将每次构建的结果工件保存到工程师共享的位置,因此如果开发人员依赖于其他人最近构建的工件,构建系统将自动下载它而不是构建它。这提供了直接依赖工件的所有性能优势,同时仍确保构建与始终从同一源构建一样一致。这是 Google 内部使用的策略,Bazel 可以配置为使用远程缓存。

外部依赖项的安全性和可靠性。依赖来自第三方来源的工件本身就存在风险。如果第三方来源 (例如工件存储库) 出现故障,则存在可用性风险,因为如果无法下载外部依赖项,您的整个构建可能会陷入停顿。还存在安全风险: 如果第三方系统被攻击者攻陷,攻击者可能会用他们自己设计的工件替换引用的工件,从而允许他们将任意代码注入您的构建中。

通过将您依赖的任何工件镜像到您控制的服务器上并阻止您的构建系统访问第三方工件存储库 (如 Maven Central),可以缓解这两个问题。但代价是这些镜像需要花费精力和资源来维护,因此是否使用它们的选择通常取决于项目的规模。通过要求在源存储库中指定每个第三方工件的哈希值,如果工件被篡改,构建将失败,也可以完全避免安全问题,而且开销很小。

另一种完全避免该问题的方法是将项目的依赖项作为供应商。当项目将其依赖项作为供应商时,它会将它们与项目源代码一起检入源代码控制中,无论是源代码还是二进制文件。这实际上意味着项目的所有外部依赖项都转换为内部依赖项。Google 在内部使用这种方法,将整个 Google 中引用的每个第三方库都检入 Google 源代码树根目录下的 `third_party` 目录中。但是,这仅在 Google 中有效,因为 Google 的源代码控制系统是专门为处理极大的 monorepo 而构建的,因此对于其他组织来说,供应商可能不是一种选择。

## 结论

构建系统是工程组织最重要的部分之一。

每个开发人员每天可能会与它交互数十次或数百次，在许多情况下，它可能是决定其生产力的限速步骤。这意味着值得投入时间和精力来把事情做好。

正如本章所讨论的，谷歌学到的一个更令人惊讶的教训是，限制工程师的权力和灵活性可以提高他们的工作效率。

我们能够开发出满足我们需求的构建系统，不是通过让工程师自由定义构建的执行方式，而是通过开发一个高度结构化的框架来限制个人选择，并将最有趣的决定交给自动化工具。不管你怎么想，工程师们并不反感这一点：Google 员工喜欢这个系统大部分时间独立运行，让他们专注于编写应用程序的有趣部分，而不是纠结于构建逻辑。能够信任构建是一件很强大的事情——增量构建可以正常工作，而且几乎不需要清除构建缓存或运行“清理”步骤。

我们利用这一见解创建了一种全新的基于工件的构建系统，与传统的基于任务的构建系统形成鲜明对比。将构建重新定义为以工件而不是任务为中心，这使得我们的构建能够扩展到像 Google 这样规模的组织。在最极端的情况下，它允许分布式构建系统利用整个计算集群的资源来提高工程师的生产力。虽然您的组织可能不够大，无法从这样的投资中受益，但我们相信基于工件的构建系统可以缩小规模，也可以扩大规模：即使对于小型项目，像 Bazel 这样的构建系统也可以在速度和正确性方面带来显着的好处。

本章的其余部分探讨了如何在基于工件的世界中管理依赖项。我们得出的结论是，细粒度模块比粗粒度模块具有更好的扩展性。我们还讨论了管理依赖项版本的困难，描述了单一版本规则以及所有依赖项都应手动且明确地进行版本控制的观察结果。这些做法避免了钻石依赖问题等常见陷阱，并允许代码库在具有统一构建系统的单个存储库中实现 Google 数十亿行代码的规模。

## TL;DR

- 功能齐全的构建系统对于在组织规模扩大时保持开发人员的生产力必不可少。 · 功能和灵活性是有代价的。适当

限制构建系统  
使开发人员更容易。

- 围绕工件组织的构建系统往往具有更好的可扩展性和更可靠的特性而不是构建围绕任务组织的系统。
- 定义工件和依赖项时,最好以细粒度模块为目标。细粒度模块能够更好地利用并行性和增量构建。
- 外部依赖项应在源代码控制下明确版本控制。依赖“最新”版本会导致灾难和无法重现的构建。

## 第十九章

# 批评:谷歌的代码审查工具

由凯特琳·萨多斯基、伊尔哈姆·库尼亚和本·罗尔夫斯撰写  
由 Lisa Carey 编辑

正如你在第 9 章中看到的,代码审查是软件开发的一个重要部分,尤其是在大规模工作时。代码审查的主要目标是提高代码库的可读性和可维护性,审查流程从根本上支持了这一点。然而,拥有定义明确的代码审查流程只是代码审查故事的一部分。支持该流程的工具在其成功中也起着重要作用。

在本章中,我们将通过 Google 广受欢迎的内部系统 Critique 来了解成功的代码审查工具。Critique 明确支持代码审查的主要动机,为审查者和作者提供审查视图和对更改发表评论的能力。Critique 还支持把关哪些代码被签入代码库,这将在“评分”更改部分中讨论。Critique 中的代码审查信息在进行代码考古时也很有用,遵循代码审查交互中解释的一些技术决策(例如,当缺少内联注释时)。尽管 Critique 不是 Google 使用的唯一代码审查工具,但它是最受欢迎的工具,遥遥领先。

## 代码审查工具原则

我们上面提到,Critique 提供了支持代码审查目标的功能(我们将在本章后面更详细地介绍此功能),但它为什么如此成功呢?Critique 受到 Google 开发文化的影响,其中包括将代码审查作为工作流程的核心部分。这种文化影响转化为一套指导原则,Critique 旨在强调这些原则:

## Simplicity

Critique 的用户界面 (UI) 旨在简化代码审查,无需进行大量不必要的选择,界面流畅。用户界面加载速度快,导航简单,支持热键,并且有清晰的视觉标记来显示更改是否已审查的总体状态。

## 信任的基础 代码审查不

是为了减慢别人的速度,而是为了赋予别人权力。

尽可能地信任同事才能让事情顺利进行。这可能意味着,例如,信任作者做出的更改,而不需要额外的审核阶段来仔细检查小意见是否得到解决。信任还通过使更改在 Google 上公开 (供查看和审核) 来实现。

## 通用通信

沟通问题很少通过工具来解决。批评优先考虑用户对代码更改进行评论的通用方式,而不是复杂的协议。批评鼓励用户在评论中阐明他们想要什么,甚至建议进行一些编辑,而不是使数据模型和流程变得更加复杂。即使使用最好的代码审查工具,沟通也可能出错,因为用户是人类。

## 工作流集成 Critique 与其

他核心软件开发工具有许多集成点。开发人员可以轻松导航到我们的代码搜索和浏览工具中查看正在审查的代码,在我们的基于 Web 的代码编辑工具中编辑代码,或查看与代码更改相关的测试结果。

在这些指导原则中,简单性可能对该工具的影响最大。我们考虑添加许多有趣的功能,但我们决定不让模型变得更复杂,以支持一小部分用户。

简单性与工作流集成之间也存在着有趣的矛盾。我们考虑过但最终决定不创建一个“代码中心”工具,在一个工具中实现代码编辑、审查和搜索。尽管 Critique 与其他工具有很多接触点,但我们有意识地决定将代码审查作为主要关注点。功能与 Critique 相关联,但在不同的子系统中实现。

## 代码审查流程

代码审查可以在软件开发的许多阶段执行,如图19-1 所示。批评性审查通常在将更改提交到代码库之前进行,也称为预提交审查。尽管第 9 章简要描述了代码审查流程,但在这里我们对其进行扩展,以描述代码审查的关键方面。

每个阶段都有帮助的批评。我们将在以下部分更详细地介绍每个阶段。

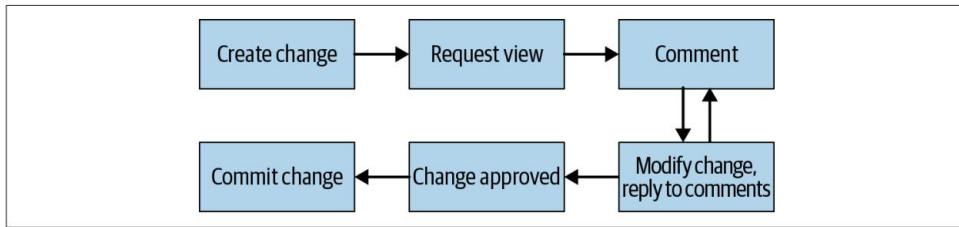


图 19-1. 代码审查流程

典型的审查步骤如下：

1. 创建变更。用户在其工作区中对代码库进行更改。

然后,该作者将快照 (显示特定时间点的补丁) 上传到 Critique,从而触发自动代码分析器的运行 (参见第 20 章)。

2. 请求审阅。当作者对更改的差异和 Critique 中显示的分析器结果感到满意时,他们会将更改发送给一个或多个审阅者。

3. 评论。审阅者在 Critique 中打开更改并起草对差异的评论。评论默认标记为未解决,这意味着它们对于作者来说至关重要。此外,审阅者可以添加可选或信息性的已解决评论。如果存在自动代码分析器的结果,审阅者也可以看到。一旦审阅者起草了一组评论,他们需要发布它们以便作者看到它们;这样做的好处是,审阅者可以在审阅整个更改后自动对更改提出完整的想法。任何人都可以对更改发表评论,并在他们认为必要时提供“驱动审阅”。

4. 修改变更并回复评论。作者修改变更,根据反馈上传新快照,并回复审阅者。

作者至少要解决所有未解决的评论,方法是更改代码或仅回复评论并更改要解决的评论类型。作者和审阅者可以查看任何快照对之间的差异,以查看更改的内容。步骤 3 和 4 可能会重复多次。

5. 变更批准。当审阅者对变更的最新状态感到满意时,他们会批准变更并将其标记为“我认为不错”(LGTM)。

他们可以选择添加要解决的评论。更改被认为适合提交后,用户界面会将其清楚地标记为绿色以显示此状态。

6. 提交变更。如果更改得到批准 (我们将很快讨论),作者可以触发更改的提交过程。如果自动

分析器和其他预提交钩子（称为“预提交”）没有发现任何问题，更改将提交到代码库。

即使在审核流程启动后，整个系统仍提供极大的灵活性，可以偏离常规审核流程。例如，审核者可以取消自己对变更的分配或明确将其分配给其他人，而作者可以完全推迟审核。在紧急情况下，作者可以强制提交其变更并在提交后对其进行审核。

## 通告

随着变更进入前面概述的阶段，Critique 会发布其他支持工具可能使用的事件通知。这种通知模型允许 Critique 专注于成为主要的代码审查工具而不是通用工具，同时仍集成到开发人员的工作流程中。通知可以实现关注点分离，这样 Critique 就可以只发出事件，其他系统就可以基于这些事件构建。

例如，用户可以安装一个 Chrome 扩展程序来使用这些事件通知。当更改需要用户注意时（例如，因为轮到他们审核更改或某些预提交失败），扩展程序会显示一个 Chrome 通知，其中包含一个按钮，可直接转到更改或静音通知。我们发现一些开发人员确实喜欢立即通知更改更新，但其他开发人员选择不使用此扩展程序，因为他们发现这会对他们的流程造成太大的干扰。

Critique 还管理与变更相关的电子邮件；重要的 Critique 事件会触发电子邮件通知。除了显示在 Critique UI 中之外，一些分析器结果还配置为通过电子邮件发送结果。Critique 还处理电子邮件回复并将其转换为评论，支持喜欢基于电子邮件的流程的用户。请注意，对于许多用户来说，电子邮件并不是代码审查的主要功能；他们使用 Critique 的仪表板视图（稍后讨论）来管理审查。

## 第一阶段：创造变革

代码审查工具应在审查过程的所有阶段提供支持，而不应成为提交更改的瓶颈。在预览步骤中，让更改作者在将更改发送出去进行审查之前更容易地完善更改，有助于减少审查人员检查更改所花费的时间。Critique 显示更改差异，并使用旋钮忽略空白更改并突出显示仅移动更改。Critique 还会显示构建、测试和静态分析器的结果，包括样式检查（如第 9 章所述）。

向作者展示变更的差异使他们有机会扮演不同的角色：代码审阅者。评论让变更作者能够像审阅者一样看到其变更的差异，并查看自动分析结果。评论还支持在审阅工具中对变更进行轻量级修改，并建议合适的审阅者。在发出请求时，作者还可以对变更进行初步评论，从而有机会直接向审阅者询问任何未解决的问题。让作者有机会像审阅者一样看到变更可以避免误解。

为了向审阅者提供更多背景信息，作者还可以将更改链接到特定的错误。Critique 使用自动完成服务来显示相关错误，并优先考虑分配给作者的错误。

## 差异化代码

码审查流程的核心是理解代码更改本身。较大的更改通常比较小的更改更难理解。因此，优化更改的差异是优秀代码审查工具的核心要求。

在 Critique 中，这一原则被运用到多个层面（见图19-2）。差异组件从优化的最长公共子序列算法开始，并通过以下方式得到增强：

· 语法高亮 · 交叉引

用（由 Kythe 提供支持；参见第 17 章） · 行内差异显示字

符级分解的差异

字边界（图 19-2） · 一个选项，

可以不同程度地忽略空格差异 · 移动检测，其中从一个地方移动到另一

个地方的代码块被标记为被移动（而不是像简单的 diff 算法那样被标记为在这里被删除并在那里被添加）

The screenshot shows a code editor with two columns of code. The left column is labeled 'A' and the right column is labeled 'B'. Both columns show the same code content, which includes imports, declarations, and various Angular modules. The code is annotated with green highlights and numbers (27, 28, 29, 30, 31, 32, 33, 34, 35, 36) corresponding to specific lines of code. These annotations highlight differences between the two versions of the code, such as moved or deleted lines.

```

 24 @NgModule({
 25   imports: [
 26     AnalysesModule,
 27     CommaSeparatedModule,
 28     CommonModule,
 29     DateModule,
 30     MatDialogModule,
 31     LabeledModule,
 32     LinkifiedModule,
 33     LinkifiedListModule,
 34     MatButtonModule,
 35     MatChipsModule,
 36     MatCollapsesModule,
 37     MatDividerModule,
 38     MatIconModule,
 39     MatInputModule,
 40     MatMenuModule,
 41     MatNativeModule,
 42     MatRippleModule,
 43     MatTabsModule,
 44     MatToolbarModule,
 45     PopupsModule,
 46     ScorePanelModule,
 47     UtilModule,
 48     UserModule,
 49   ],
 50   declarations: [
 51     AnalysisChips,
 52   ]
 53 })
 54 
```

图 19-2. 行内标记显示字符级差异

用户还可以使用各种不同的模式查看差异,例如叠加和并排。在开发 Critique 时,我们认为并排差异非常重要,这样可以简化审核过程。并排差异占用大量空间:为了实现它们,我们必须简化差异视图结构,这样就没有边框,没有填充 - 只有差异和行号。我们还必须尝试各种字体和大小,直到我们拥有一个差异视图,它可以适应 Java 在 Critique 推出时典型的屏幕宽度分辨率(1,440 像素)下的 100 个字符的行限制。

Critique 进一步支持各种自定义工具,这些工具提供由更改产生的工作的差异,例如由更改修改的 UI 的屏幕截图差异或由更改生成的配置文件。

为了使浏览差异的过程更加顺畅,我们非常注意不浪费空间,并投入大量精力确保差异能够快速加载,即使是图像和大文件和/或更改也是如此。我们还提供了键盘快捷键,以便在仅访问修改部分时快速浏览文件。

当用户深入到文件级别时,Critique 会提供一个 UI 小部件,其中紧凑地显示文件的快照版本链;用户可以通过拖放来选择要比较的版本。此小部件会自动折叠类似的快照,从而将焦点吸引到重要的快照上。它可以帮助用户了解更改内文件的演变;例如,哪些快照具有测试覆盖率、哪些快照已经过审核或有评论。为了解决规模问题,Critique 会预取所有内容,因此加载不同的快照非常快。

### 分析结果上传变更快

照会触发代码分析器(见第 20 章)。Critique 在变更页面上显示分析结果,由变更描述下方的分析器状态芯片汇总,如图 19-3 所示,并在“分析”选项卡中详细说明,如图 19-4 所示。

分析人员可以标记特定发现并将其突出显示为红色,以提高可见性。仍在进行的分析人员用黄色筹码表示,否则显示灰色筹码。为简单起见,Critique 没有提供其他选项来标记或突出显示发现 - 可操作性是一个二元选项。如果分析器产生了一些结果(“发现”),单击筹码即可打开发现。与评论一样,发现可以在 diff 中显示,但样式不同,以便于区分。有时,发现还包括修复建议,作者可以预览并选择从 Critique 中应用这些建议。

The screenshot shows the Critique interface for a pull request titled "Change 243497582 by ilham". The status is "Pending". The review summary includes:

- Reviewers:** caitlin
- CC:**
- Bugs:**
- Diffbase:**
- Score:** LGTM - Missing
- Analysis:** Actionable: Presubmit:CheckProtoSyntax
- Done:** Presubmit

The workspace is set to "pizza".

**Files:**

| File                        | Comments | Inline | Modified | Delta |
|-----------------------------|----------|--------|----------|-------|
| pizza/BUILD                 | Added    | Diff   | 3:05 PM  | 7     |
| pizza/PizzaSupplierApp.java | Added    | Diff   | 3:06 PM  | 22    |
| pizza/pizza.proto           | Added    | Hide   | 3:05 PM  | 28    |

**/dev/null**

Mark this file as reviewed

**Presubmit:CheckProtoSyntax** Your change contains one or more new .proto files without a syntax statement. Each .proto file must have a syntax statement in order to submit. [Not useful](#)

```
package google.pizza;
import google.Timestamp;
message Ingredient {
    required int64 id = 1;
    required string name = 2;
    required int64 unit = 3;
    optional Timestamp season = 4;
```

[Create file comment](#) [Snapshot #1 3:05 PM \(text\)](#)

图 19-3. 变更摘要和 di 视图

The screenshot shows the Critique interface for the same pull request. The status is "Pending". The analysis summary includes:

- Reviewers:** caitlin
- CC:**
- Bugs:**
- Diffbase:**
- Score:** LGTM - Missing
- Analysis:** Actionable: Presubmit:CheckProtoSyntax
- Done:** Presubmit

The workspace is set to "pizza".

**Analysis**

**Filters** Only with findings Category status:  Completed  Running  Failed  Include findings on unchanged lines

| Category                   | Status | Snapshot                     | First finding snippet                                                                                                                           |
|----------------------------|--------|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| Presubmit:CheckProtoSyntax | ✓      | 2 (Latest) <b>Actionable</b> | Your change contains one or more new .proto files without a syntax statement. Each .proto file must have a syntax statement in order to submit. |
| Presubmit                  | ✓      | 2 (Latest)                   | Presubmits finished with status SUCCESS. Reported 1 notice(s), 0 warning(s), 1 error(s). NOTES: Presubmits were invoked with ...                |

图 19-4. 分析结果

例如,假设一个 linter 发现行尾有多余的空格,这是样式违规。更改页面将显示该 linter 的 chip。从 chip 中,作者可以快速转到显示有问题代码的 diff,只需单击两次即可了解样式违规。大多数 linter 违规还包括修复建议。单击一下,

作者可以预览修复建议（例如，删除多余的空格），然后再次单击即可将修复应用于更改。

## 紧密的工具集成

Google 在其整体源代码存储库 Piper 上构建了一些工具（参见第 16 章），例如以下内容：

- Cider，一款用于编辑存储在云端的源代码的在线 IDE · Code Search，
  - 一款用于在代码库中搜索代码的工具 · Tricorder，一款用于显示静态分析结果的工具（前面提到过） · Rapid，一款用于打包和部署包含一系列变化 ·
- Zapfhahn，一个测试覆盖率计算工具

此外，还有一些服务提供有关变更元数据的上下文（例如，有关参与变更的用户或链接的 Bug）。Critique 是一个天然的大熔炉，可以通过一键/悬停快速访问，甚至可以嵌入 UI 支持这些系统，尽管我们需要注意不要牺牲简单性。例如，在 Critique 中的变更页面，作者只需单击一次即可开始在 Cider 中进一步编辑变更。支持使用 Kythe 在交叉引用之间导航，或在代码搜索中查看代码的主线状态（参见第 17 章）。Critique 链接到发布工具，以便用户可以看到提交的变更是否在特定发布中。对于这些工具，Critique 更喜欢使用链接而不是嵌入，以免分散核心审查体验。这里的一个例外是测试覆盖率：一行代码是否被测试覆盖的信息通过文件差异视图中行距上的不同背景颜色显示（并非所有项目都使用此覆盖率工具）。

请注意，Critique 与开发人员工作区之间的紧密集成是可能的，因为工作区存储在基于 FUSE 的文件系统中，特定开发人员的计算机之外也可以访问。Source of Truth 托管在云中，所有这些工具都可以访问。

## 第 2 阶段：请求审核

当作者对变更状态感到满意时，他们可以将其发送给评审，如图 19-5 所示。这需要作者挑选评审者。在小团队中，寻找评审者似乎很简单，但即使如此，在团队成员之间均匀分配评审并考虑谁在度假等情况也是很有用的。

为了解决这个问题，团队可以为传入的代码审查提供电子邮件别名。该别名由名为 GwsQ 的工具使用（以最初使用此技术的团队命名）：

Google Web Server)根据与别名关联的配置分配特定审阅者。例如,变更作者可以将审阅分配给 some-team-list-alias,GwsQ 将挑选 some-team-list-alias 的特定成员来执行审阅。

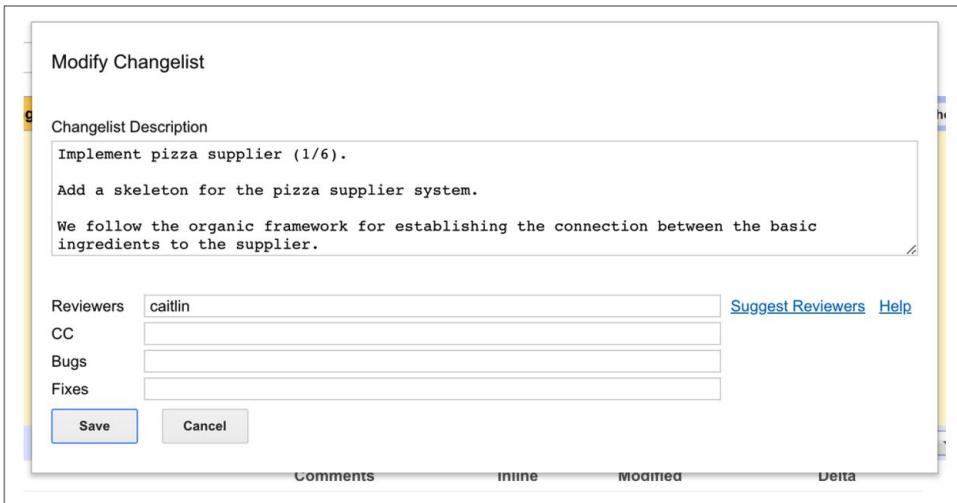


图 19-5. 请求审阅者

考虑到 Google 代码库的规模以及修改它的人数,很难找出最有资格审查您自己项目之外的更改的人。

当达到一定规模时,寻找审阅者是一个需要考虑的问题。评论必须处理规模问题。评论提供了建议足以批准更改的审阅者集的功能。审阅者选择实用程序会考虑以下因素:

- 谁负责修改的代码 (见下一节)    · 谁最熟悉代码 (即谁最近修改了代码)
- 谁可以进行审查 (即不在办公室,最好在同一时间)
- GwsQ 团队别名设置

为变更分配审阅者会触发审阅请求。此请求将运行适用于变更的“预提交”或预提交钩子;团队可以通过多种方式配置与其项目相关的预提交。最常见的钩子包括：

- 自动将电子邮件列表添加到更改中以提高知名度和透明度
- 为项目运行自动测试套件
- 在代码（以强制执行本地代码样式限制）和更改描述（以允许生成发行说明或其他形式的跟踪）上执行项目特定的不变量

由于运行测试需要大量资源,因此在 Google,它们是预提交的一部分（在请求审核和提交更改时运行）,而不是像 Tricorder 检查那样在每个快照中运行。Critique 以与分析器结果显示类似的方式显示运行钩子的结果,并额外强调了失败的结果会阻止更改被发送以供审核或提交的事实。如果预提交失败,Critique 会通过电子邮件通知作者。

## 第三和第四阶段:理解和评论 改变

评审过程开始后,作者和评审人员齐心协力,以达到提交高质量变更的目标。

评论发表评论是用户

在 Critique 中查看更改后进行的第二大常见操作 ([图 19-6](#))。在 Critique 中发表评论对所有人都是免费的。任何人（不仅是更改作者和指定的审阅者）都可以对更改发表评论。

Critique 还提供了通过每个人的状态跟踪审阅进度的功能。审阅者可以使用复选框将最新快照中的单个文件标记为已审阅,从而帮助审阅者跟踪他们已经查看的内容。当作者修改文件时,所有审阅者都会清除该文件的“已审阅”复选框,因为最新快照已更新。

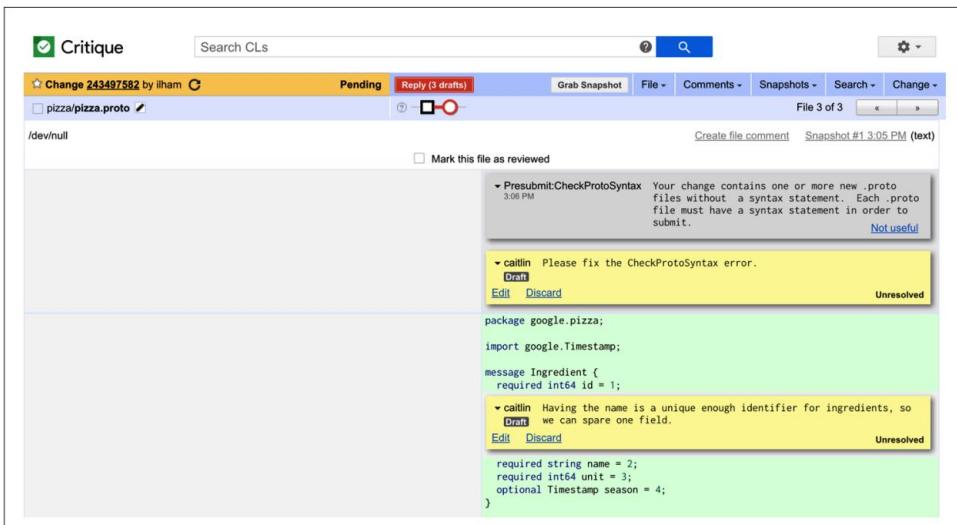


图 19-6. 对 di 视图进行注释

当审阅者看到相关的分析器发现时,他们可以点击“请修复”按钮来创建一条未解决的评论,要求作者解决该发现。审阅者还可以通过内联编辑文件的最新版本来建议修复更改。评论会将此建议转换为带有修复的评论,作者可以应用该修复。

Critique 不会规定用户应该创建哪些评论,但对于一些常见评论,Critique 提供了快捷方式。更改作者可以单击评论面板上的“完成”按钮来指示何时已处理审阅者的评论,或单击“确认”按钮来确认已阅读评论,通常用于信息性或可选评论。如果评论线程未解决,则两者都可以解决评论线程。这些快捷方式简化了工作流程并减少了响应审阅评论所需的时间。

如前所述,评论是即时起草的,但随后会以原子方式“发布”,如图19-7 所示。这样,作者和审阅者就可以确保在发送评论之前对评论感到满意。

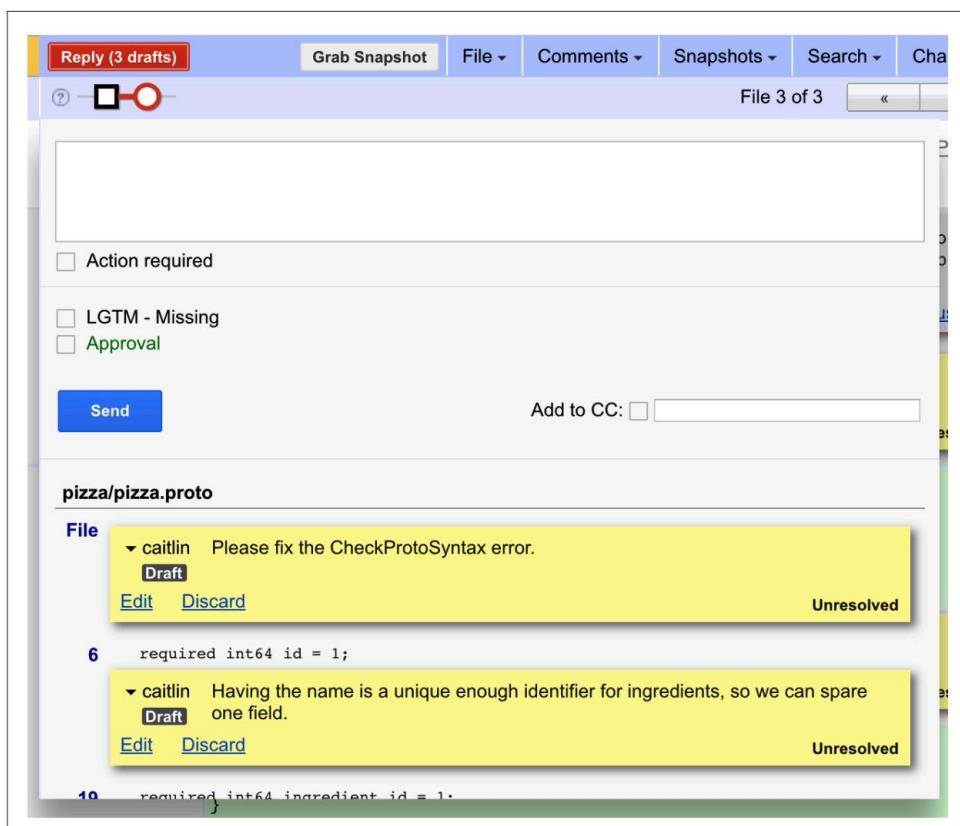


图 19-7. 准备给作者的评论

了解变更评论的状态提供了多种机制,可以清楚

地了解变更目前处于评论和迭代阶段的哪个阶段。这些机制包括确定谁需要采取下一步行动的功能,以及特定开发人员参与的所有变更的审阅/作者状态的仪表板视图。

#### “轮到谁了”功能

加速审核流程的一个重要因素是了解何时轮到您采取行动,尤其是当一个变更被分配了多个审核者时。

如果作者希望软件工程师和负责该功能的用户体验人员或负责该服务的 SRE 审查他们的变更,则可能出现这种情况。通过管理每个变更的注意力集,评论有助于定义谁应该接下来查看变更。

关注集由当前阻止更改的一组人组成。

当评论者或作者处于关注集中时,他们应该及时做出回应。Critique 试图在用户发表评论时智能地更新关注集,但用户也可以自己管理关注集。当更改中有更多的评论者时,它的实用性会进一步增强。Critique 中的关注集通过以粗体显示相关用户名来显示。

在我们实现此功能后,我们的用户很难想象以前的状态。普遍的看法是:没有这个我们怎么过?在我们实现此功能之前,另一种方法是在审阅者和作者之间进行聊天,以了解谁在处理变更。此功能还强调了代码审阅的轮流性质;总是至少有一个人轮流采取行动。

### 仪表板和搜索系统Critique 的

登录页面是用户的仪表板页面,如图19-8所示。仪表板页面分为用户可自定义的部分,每个部分都包含一个变更摘要列表。

| Needs attention 4 Changes |         |         |                 |           |      |                                 |
|---------------------------|---------|---------|-----------------|-----------|------|---------------------------------|
| Change                    | Author  | Status  | Last Action     | Reviewers | Size | Description                     |
| <a href="#">42972248</a>  | ilham   | Pending | Apr 11 by gwsq  | caitlin   | XS   | Implement pizza supplier (6/6). |
| <a href="#">42974683</a>  | ilham   | Pending | Apr 11 by tap   | caitlin   | S    | Implement pizza supplier (5/6). |
| <a href="#">37099895</a>  | ilham   | Pending | Apr 11 by ilham | caitlin   | M    | Implement pizza supplier (4/6). |
| <a href="#">27761071</a>  | caitlin | Pending | Jan 8 by ilham  | ilham     | XS   | Implement pizza maker (3/3).    |

| Incoming reviews 6 Changes |        |            |                  |           |      |                                 |
|----------------------------|--------|------------|------------------|-----------|------|---------------------------------|
| Change                     | Author | Status     | Last Action      | Reviewers | Size | Description                     |
| <a href="#">42972248</a>   | ilham  | Pending    | Apr 11 by gwsq   | caitlin   | XS   | Implement pizza supplier (6/6). |
| <a href="#">42974683</a>   | ilham  | Pending    | Apr 11 by tap    | caitlin   | S    | Implement pizza supplier (5/6). |
| <a href="#">37099895</a>   | ilham  | Pending    | Apr 11 by ilham  | caitlin   | M    | Implement pizza supplier (4/6). |
| <a href="#">42161351</a>   | ilham  | LGTM       | Apr 9 by caitlin | caitlin   | XS   | Implement pizza supplier (3/6). |
| <a href="#">40374250</a>   | ilham  | Unresolved | Apr 4 by caitlin | caitlin   | XS   | Implement pizza supplier (2/6). |
| <a href="#">36387832</a>   | ilham  | Unresolved | Mar 5 by caitlin | caitlin   | L    | Implement pizza supplier (1/6). |

| Outgoing reviews 3 Changes |         |         |                  |           |      |                              |
|----------------------------|---------|---------|------------------|-----------|------|------------------------------|
| Change                     | Author  | Status  | Last Action      | Reviewers | Size | Description                  |
| <a href="#">27761071</a>   | caitlin | Pending | Jan 8 by caitlin | ilham     | XS   | Implement pizza maker (3/3). |
| <a href="#">15068925</a>   | caitlin | Pending | Jan 6 by caitlin | ilham     | S    | Implement pizza maker (2/3). |
| <a href="#">15416497</a>   | caitlin | Pending | Jan 2 by caitlin | ilham     | M    | Implement pizza maker (1/3). |

图 19-8. 仪表板视图

仪表板页面由一个名为“变更列表搜索”的搜索系统提供支持。变更列表搜索索引了 Google 所有用户的所有可用变更(提交前和提交后)的最新状态,并允许其用户通过定期

基于表达式的查询。每个仪表板部分都由对变更列表搜索的查询定义。我们花了很多时间确保变更列表搜索足够快以供交互使用；所有内容都快速编入索引，以便作者和审阅者不会放慢速度，尽管我们在 Google 同时发生了大量并发更改。

为了优化用户体验 (UX)，Critique 的默认仪表板设置是让第一部分显示需要用户注意的更改，尽管这是可自定义的。还有一个搜索栏，用于对所有更改进行自定义查询并浏览结果。作为审阅者，您主要只需要设置注意。作为作者，您主要只需要查看仍在等待审阅的内容，看看是否需要 ping 任何更改。尽管我们在 Critique UI 的其他一些部分避免了可定制性，但我们发现用户喜欢以不同的方式设置他们的仪表板，而不会影响基本体验，类似于每个人组织电子邮件的方式不同。<sup>1</sup>

## 第五阶段：变更批准（变更评分）

要显示审阅者是否认为更改是好的，归根结底就是通过评论提供关注和建议。还需要某种机制来为更改提供高级别的“OK”。在 Google，更改的评分分为三个部分：

- LGTM（“我觉得不错”）
- 认可
- 未解决的评论数量

审阅者的 LGTM 印章意味着“我已经审阅了此更改，认为它符合我们的标准，并且我认为在解决未解决的评论后可以提交它。”审阅者的 Approval 印章意味着“作为守门人，我允许将此更改提交到代码库。”审阅者可以将评论标记为未解决，这意味着作者需要对其采取行动。当更改至少有一个 LGTM、足够的批准并且没有未解决的评论时，作者就可以提交更改了。请注意，无论批准状态如何，每个更改都需要 LGTM，以确保至少有两双眼睛查看了更改。这个简单的评分规则允许 Critique 通知作者何时更改可以提交（以绿色页眉突出显示）。

---

<sup>1</sup>集中式“全局”大规模变更审阅者 (LSC) 特别容易自定义此仪表板  
避免在 LSC 期间发生洪水（参见第 22 章）。

在构建 Critique 的过程中,我们特意决定简化此评级方案。最初,Critique 有“需要更多工作”评级和“LGTM+”<sup>+</sup>。我们已转向的模型是使 LGTM/Approval 始终为正面。如果更改确实需要第二次审阅,则主要审阅者可以添加评论,但无需 LGTM/Approval。在更改过渡到基本良好状态后,审阅者通常会信任作者来处理小幅编辑。无论更改大小,工具都不需要重复 LGTM。

这种评级方案也对代码审查文化产生了积极影响。审查人员不能只是对没有有用反馈的变更不予采纳;审查人员的所有负面反馈都必须与需要修复的具体问题相关(例如,未解决的评论)。“未解决的评论”这个措辞也听起来比较好听。

评论包括一个评分面板,位于分析芯片旁边,其中包含以下信息:

- 谁已 LGTM 变更 · 仍需要哪些批准以及原因 · 仍有多少未解决的评论悬而未决

以这种方式呈现评分信息有助于作者快速了解他们还需要做什么才能使变更得到实现。

LGTM 和 Approval 是硬性要求,只有审阅者才能授予。

审阅者还可以在更改提交之前随时撤销其 LGTM 和批准。未解决的评论是必需的;作者可以在回复时将评论标记为“已解决”。这种区别促进并依赖于作者和审阅者之间的信任和沟通。例如,审阅者可以 LGTM 未解决的评论的更改,而无需稍后检查评论是否真正得到解决,这突显了审阅者对作者的信任。当作者和审阅者之间存在显着的时区差异时,这种信任对于节省时间尤为重要。表现出信任也是建立信任和加强团队的好方法。

## 第六阶段:提交变更

最后但同样重要的一点是,Critique 有一个按钮用于在审查后提交更改,以避免上下文切换到命令行界面。

### 提交后:跟踪历史记录除了将 Critique 的核心

用途用作在将源代码更改提交到存储库之前对其进行审查的工具之外,Critique 还可用作变更考古的工具。对于大多数文件,开发人员可以在代码搜索系统中查看修改特定文件的更改历史记录列表 (参见第 17 章),或直接导航到更改。Google 的任何人都可以浏览一般可查看文件的更改历史记录,包括对更改的评论和演变。这可以进行未来的审计,并用于了解有关进行更改的原因或如何引入错误的更多详细信息。开发人员还可以使用此功能了解更改是如何设计的,并且汇总的代码审查数据可用于生成培训。

Critique 还支持在更改提交后发表评论的功能;例如,当稍后发现问题或额外的背景信息可能对其他时间调查更改的人有用时。Critique 还支持回滚更改并查看特定更改是否已回滚的功能。

#### 案例研究:Gerrit 虽然 Critique

是 Google 最常用的审查工具,但它并不是唯一的工具。由于 Critique 与我们的大型单片存储库和其他内部工具紧密相关,因此无法对外使用。因此,在 Google 中,从事开源项目 (包括 Chrome 和 Android)或无法或不想托管在单片存储库中的内部项目的团队会使用不同的代码审查工具:Gerrit。

Gerrit 是一款独立的开源代码审查工具,与 Git 版本控制系统紧密集成。因此,它为许多 Git 功能提供了 Web UI,包括代码浏览、合并分支、挑选提交,当然还有代码审查。此外,Gerrit 有一个细粒度的权限模型,我们可以使用它来限制对存储库和分支的访问。

Critique 和 Gerrit 的代码审查模型相同,即每个提交都单独审查。Gerrit 支持堆叠提交并上传以供单独审查。它还允许在审查后以原子方式提交链。

由于是开源软件,Gerrit 可以容纳更多变体和更广泛的用例;Gerrit 丰富的插件系统可以紧密集成到自定义环境中。为了支持这些用例,Gerrit 还支持更复杂的评分系统。审阅者可以通过给出 -2 分来否决更改,并且评分系统具有高度可配置性。



您可以在<https://www.gerritcodereview.com>上了解有关 Gerrit 的更多信息并查看其实际运行情况。

## 结论

使用代码审查工具时,有许多隐含的权衡。Critique 内置了许多功能,并与其它工具集成,使用户的审查过程更加无缝。花在代码审查上的时间不是花在编码上的时间,因此审查过程的任何优化都可以提高公司的生产力。

在大多数情况下,只有两个人(作者和审阅者)同意更改才能提交,从而保持较高的速度。尽管代码审查的教育方面很难量化,但 Google 非常重视这些方面。

为了最大限度地缩短审查变更所需的时间,代码审查流程应无缝衔接,简洁地告知用户需要注意的变更,并在人工审查人员介入之前识别潜在问题(问题由分析器和持续集成捕获)。如果可能,在运行时间较长的分析完成之前提供快速分析结果。

Critique 需要通过多种方式来支持规模问题。Critique 工具必须能够扩展到大量产生的审阅请求,而不会降低性能。由于 Critique 是提交更改的关键路径,因此它必须高效加载并可用于特殊情况,例如异常大的更改。2界面必须支持管理大型代码库中的用户活动(例如查找相关更改),并帮助审阅者和作者浏览代码库。例如,Critique 有助于找到合适的审阅者进行更改,而无需弄清楚所有有权/维护者格局(这一功能对于大规模更改(例如可能影响许多文件的 API 迁移)尤为重要)。

Critique 倾向于采用一种自定流程和简单的界面来改进一般的审核工作流程。但是,Critique 确实允许一些可定制性:自定义分析器和预提交提供了有关更改的具体背景,并且可以执行一些团队特定的政策(例如要求多个审核者提供 LGTM)。

---

<sup>2</sup>尽管大多数更改都很小(少于 100 行),但 Critique 有时会用于审查可能涉及数百或数千个文件的大型重构更改,尤其是对于必须原子执行的 LSC(参见第 22 章)。

信任和沟通是代码审查流程的核心。工具可以增强体验,但不能取代它们。与其他工具的紧密集成也是 Critique 成功的关键因素。

## TL;DR

- 信任和沟通是代码审查流程的核心。工具可以增强体验,但无法取代它们。
- 与其他工具紧密集成是获得良好代码审查体验的关键。 · 小型工作流程优化(例如添加明确的“注意集”)可以提高清晰度并大幅减少摩擦。

## 第二十章

# 静态分析

作者:凯特琳·萨多斯基 (Caitlin Sadowski)  
由 Lisa Carey 编辑

静态分析是指通过分析源代码来查找潜在问题（如错误、反模式和其他无需执行程序即可诊断的问题）的程序。“静态”具体是指分析源代码而不是正在运行的程序（称为“动态”分析）。静态分析可以在程序作为生产代码签入之前及早发现其中的错误。例如，静态分析可以识别溢出的常量表达式、从未运行的测试或执行时会崩溃的日志语句中的无效格式字符串。<sup>1</sup>然而，静态分析不仅仅用于查找错误。通过 Google 的静态分析，我们编纂了最佳实践，帮助使代码与现代 API 版本保持同步，并防止或减少技术债务。这些分析的示例包括验证是否遵守命名约定、标记已弃用的 API 的使用或指出使代码更易于阅读的更简单但等效的表达式。静态分析也是 API 弃用过程中不可或缺的工具，它可以防止在将代码库迁移到新 API 时出现倒退（参见第 22 章）。<sup>2</sup>我们还发现有证据表明，静态分析检查可以教育开发人员，并实际上防止反模式进入代码库。<sup>2</sup>

---

<sup>1</sup>请参阅<http://errorprone.info/bugpatterns>。

<sup>2</sup> Caitlin Sadowski 等人。Tricorder 构建程序分析生态系统，国际软件工程会议 (ICSE)，2015 年 5 月。

在本章中,我们将研究什么是有效的静态分析,Google 学到的关于如何使静态分析发挥作用的一些经验教训,以及如何在静态分析工具和流程中实现这些最佳实践。<sup>3</sup>

## 有效静态分析的特征

尽管数十年来静态分析研究一直致力于开发新的分析技术和特定的分析,但对提高静态分析工具的可扩展性和可用性的方法的关注却是一个相对较新的发展。

### 可扩展性因为

现代软件变得越来越大,分析工具必须明确解决扩展问题才能及时产生结果,而不会减慢软件开发过程。Google 的静态分析工具必须扩展到 Google 数十亿行代码库的规模。为此,分析工具是可分片和增量的。我们不会分析整个大型项目,而是专注于受待处理代码更改影响的文件的分析,并且通常只显示已编辑文件或行的分析结果。扩展也有好处:因为我们的代码库非常大,所以有很多唾手可得的错误需要查找。除了确保分析工具可以在大型代码库上运行之外,我们还必须增加可用分析的数量和种类。分析贡献来自整个公司。静态分析可扩展性的另一个组成部分是确保流程可扩展。为此,Google 静态分析基础架构通过直接向相关工程师显示分析结果来避免瓶颈。

### 可用性在考

虑分析可用性时,对于静态分析工具用户来说,重要的是要考虑成本效益。这种“成本”可能是开发人员的时间,也可能是代码质量。修复静态分析警告可能会引入错误。对于不经常修改的代码,为什么要“修复”在生产中运行良好的代码?例如,通过添加对之前死代码的调用来修复死代码警告可能会导致未经测试的(可能有错误的)代码突然运行。这样做的好处不明确,而且成本可能很高。因此,我们通常关注新引入的警告;在正常工作的代码中,现有问题通常只有在特别重要(安全问题、重大错误修复等)时才值得突出显示(和修复)。关注新引入的警告(或有关警告)

---

<sup>3</sup>静态分析理论一个很好的学术参考书是:Flemming Nielson 等著的《程序分析原理》(德国:Springer,2004)。

修改后的行数也意味着查看警告的开发人员拥有最相关的上下文。

此外,开发人员的时间非常宝贵!花在分类分析报告或修复突出问题上的时间与特定分析带来的好处是相权衡的。如果分析作者可以节省时间(例如,通过提供可以自动应用于相关代码的修复程序),则权衡的成本就会降低。任何可以自动修复的问题都应该自动修复。我们还尝试向开发人员展示对代码质量有负面影响的问题报告,以便他们不会浪费时间苦苦寻找不相关的结果。

为了进一步降低审查静态分析结果的成本,我们专注于顺畅的开发人员工作流程集成。将所有内容统一到一个工作流程中的另一个优势是,专门的工具团队可以随工作流程和代码一起更新工具,从而使分析工具能够与源代码一起发展。

我们相信,在使静态分析具有可扩展性和可用性方面我们做出的这些选择和权衡自然而然地源于我们对三个核心原则的关注,我们将在下一节中将其作为经验教训来阐述。

## 静态分析工作的关键经验

关于如何让静态分析工具发挥良好作用,我们在Google学到了三个关键经验。让我们在以下小节中了解它们。

关注开发人员的幸福感我们提到了一些尝试节省

开发人员时间和降低与上述静态分析工具交互成本的方法;我们还会跟踪分析工具的执行情况。如果不衡量这一点,就无法解决问题。我们只部署误报率低的分析工具(稍后会详细介绍)。我们还积极征求并实时采纳使用静态分析结果的开发人员的反馈。在静态分析工具用户和工具开发人员之间培养这种反馈循环会形成一个良性循环,从而建立用户信任并改进我们的工具。用户信任对于静态分析工具的成功极为重要。

对于静态分析,“假阴性”是指一段代码包含分析工具旨在发现的问题,但该工具却错过了它。“假阳性”是指工具错误地将代码标记为存在问题。关于静态分析工具的研究传统上侧重于减少假阴性;在实践中,低假阳性

误报率通常对于开发人员是否愿意使用工具至关重要。谁愿意为了寻找几个真实报告而浏览数百条错误报告?<sup>4</sup>此外,感知是误报率的一个关键方面。如果静态分析工具生成的

警告在技术上是正确的,但被用户误解为误报(例如,由于消息令人困惑),用户的反应会和这些警告实际上是误报一样。同样,在技术上正确但在总体上并不重要的警告也会引起同样的反应。我们将用户感知的误报率称为“有效误报”率。如果开发人员在发现问题后没有采取一些积极措施,则问题就是“有效误报”。这意味着,如果分析错误地报告了问题,但开发人员仍然乐意进行修复以提高代码的可读性或可维护性,这不是有效的误报。例如,我们有一个 Java 分析,它标记了开发人员在哈希表上调用contains方法(相当于containsValue)的情况,而他们实际上想要调用containsKey即使开发人员正确地想要检查值,调用containsValue更清楚。同样,如果分析报告了实际错误,但开发人员没有理解错误并因此没有采取任何措施,那么这是一个有效的误报。

将静态分析纳入核心开发人员工作流程在 Google,我们通过与代码审查工具集成

将静态分析整合到核心工作流程中。基本上,Google 提交的所有代码在提交之前都会经过审查;由于开发人员在发送代码进行审查时已经处于改变心态,因此静态分析工具建议的改进可以在不造成太多干扰的情况下进行。代码审查集成还有其他好处。开发人员通常在发送代码进行审查后切换上下文,并且不会受到审查者的阻拦。有时运行分析,即使分析需要几分钟才能完成。审查者也会施加同行压力,要求他们解决静态分析警告。此外,静态分析可以通过自动突出显示常见问题来节省审查者的时间;静态分析工具有助于代码审查流程(和审查者)扩展。代码审查是分析结果的最佳点。<sup>5</sup>

让用户有权力做出贡献Google 有许多领

域专家,他们的知识可以改进生成的代码。静态分析是一个利用专业知识并大规模应用的机会,让领域专家编写新的分析工具或工具中的单独检查。

---

<sup>4</sup>请注意,对于某些特定分析,审稿人可能愿意容忍更高的假阳性率:一个例子是识别关键问题的安全分析。

<sup>5</sup>有关编辑和浏览时的其他集成点的更多信息,请参阅本章后面的内容代码。

例如,了解某种配置文件上下文的专家可以编写一个分析器来检查这些文件的属性。除了领域专家之外,发现错误并希望防止同一类型错误在代码库的任何其他地方再次出现的开发人员也会提供分析。我们专注于构建一个易于插入的静态分析生态系统,而不是集成一小组现有工具。我们专注于开发简单的 API,供整个 Google 的工程师 (而不仅仅是分析或语言专家) 用于创建分析;例如, Refaster<sup>6</sup>支持通过指定前后代码片段来编写分析器,以演示该分析器期望进行哪些转换。

## Tricorder:Google 的静态分析平台

Tricorder 是我们的静态分析平台,也是 Google 静态分析的核心部分。Tricorder 诞生于 Google 将静态分析与开发人员工作流程集成的几次失败尝试中; Tricorder 与之前的尝试之间的关键区别在于我们坚持不懈地专注于让 Tricorder 只向用户提供有价值的结果。Tricorder 与 Google 的主要代码审查工具 Critique 集成在一起。

Tricorder 警告在 Critique 的差异查看器中显示为灰色注释框,如图20-1 所示。

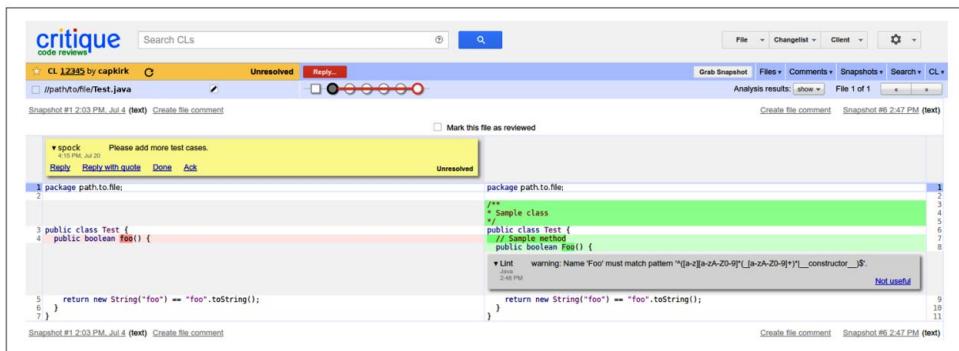


图 20-1. Critique 的 di 查看,显示了来自 Tricorder 的静态分析警告

灰色的

6 Louis Wasserman, “使用 Refaster 实现可扩展、基于示例的重构。”重构工具研讨会, 2013年。

7 Caitlin Sadowski、Jeffrey van Gogh、Ciera Jaspan、Emma Söderberg 和 Collin Winter, 《Tricorder:构建程序分析生态系统》,国际软件工程会议 (ICSE),2015 年 5 月。

8 Caitlin Sadowski、Edward Aftandilian、Alex Eagle、Liam Miller-Cushon 和 Ciera Jaspan, “从在 Google 构建静态分析工具”,ACM 通讯,61 No. 4 (2018 年 4 月) :58–66, <https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext>。

为了扩展,Tricorder 使用微服务架构。Tricorder 系统将分析请求连同有关代码更改的元数据一起发送到分析服务器。这些服务器可以使用该元数据通过基于 FUSE 的文件系统读取更改中的源代码文件版本,并可以访问缓存的构建输入和输出。

然后,分析服务器开始运行每个单独的分析器,并将输出写入存储层;然后,每个类别的最新结果将显示在 Critique 中。由于分析有时需要几分钟才能运行,因此分析服务器还会发布状态更新,让变更作者和审阅者知道分析器正在运行,并在分析器完成后发布完成状态。Tricorder 每天分析超过 50,000 个代码审查变更,并且通常每秒运行多个分析。

整个 Google 的开发人员编写 Tricorder 分析 (称为“分析器”) 或为现有分析贡献单独的“检查”。新的 Tricorder 检查有四个标准:

易于理解 任何工程师都

可以轻松理解输出。

具有可操作性并且易于 x

修复可能需要比编译器检查更多的时间、思考或精力,并且结果应该包括有关如何修复问题的指导。

产生少于 10% 的有效误报开发人员应该感觉到检查至少

在 90% 的时间内指出了实际问题。

有可能对代码质量产生重大影响

这些问题可能不会影响正确性,但开发人员应该认真对待并有意选择修复它们。

Tricorder 分析器可报告 30 多种语言的结果,并支持多种分析类型。Tricorder 包含 100 多个分析器,其中大多数来自 Tricorder 团队之外。其中七个分析器本身就是插件系统,具有数百个额外检查,同样来自 Google 的开发人员。总体有效误报率略低于 5%。

集成工具Tricorder 集成

了许多不同类型的静态分析工具。

容易出错并且整洁有序扩展编译器以分别识别 Java 和 C++ 的 AST 反模式。这些反模式可能代表真正的错误。例如,考虑以下代码片段对 long 类型的字段f进行哈希处理:

```
结果 = 31 * 结果 + (int) (f ^ (f >>> 32));
```

现在考虑f的类型为 int 的情况。代码仍会编译,但右移 32 是无操作,因此f与自身进行异或,不再影响生成的值。我们在 Google 代码库中修复了 31 个此错误,同时在 Error Prone 中启用了编译器错误检查。还有[更多此类示例](#)。AST 反模式还可以提高代码可读性,例如删除智能指针上对.get()的冗余调用。

其他分析器展示语料库中不同文件之间的关系。如果代码库中的其他非代码位置 (例如签入文档内) 引用了源文件,则 Deleted Artifact Analyzer 会发出警告。IfThis-ThenThat 允许开发人员指定必须同时更改两个不同文件的部分内容 (如果没有同时更改,则会发出警告)。Chrome 的 Finch 分析器在 Chrome 中的 A/B 实验配置文件上运行,突出显示常见问题,包括没有正确的批准来启动实验或与影响同一人群的其他当前正在运行的实验发生串扰。Finch 分析器对其他服务进行远程过程调用 (RPC) 以提供此信息。

除了源代码本身之外,一些分析器还在该源代码生成的其他工件上运行;许多项目都启用了二进制大小检查器,当更改严重影响二进制大小时会发出警告。

几乎所有分析器都是过程内的,这意味着分析结果基于过程 (函数) 内的代码。组合或增量过程间分析技术在技术上是可行的,但需要额外的基础设施投资 (例如,在分析器运行时分析和存储方法摘要)。

### 集成的反馈渠道如前所述,在分析消费者

和分析编写者之间建立反馈循环对于跟踪和维持开发人员的满意度至关重要。使用 Tricorder,我们会显示单击分析结果上的“无用”按钮的选项;单击此按钮可以直接向分析器编写者提交错误,说明为什么结果没用,并且会预先填充有关分析结果的信息。代码审阅者还可以通过单击“请修复”按钮要求变更作者解决分析结果。Tricorder 团队会跟踪“无用”点击率高的分析器,特别是相对于审阅者要求修复分析结果的频率,如果分析器不能解决问题并提高“无用”率,他们将禁用分析器。

建立和调整这个反馈回路需要大量工作,但在改善分析结果和改善用户体验 (UX) 方面已获得了多次回报 - 在我们建立清晰的反馈渠道之前,许多开发人员会忽略他们不理解的分析结果。

有时修复非常简单 - 例如更新分析器输出的消息文本!例如,我们曾经推出一个 Error Prone 检查,当向 Guava 中仅接受 %s (而没有其他 printf 说明符)的类似 printf 的函数传递了太多参数时,该检查会进行标记。Error Prone 团队每周都会收到“无用”的错误报告,声称分析不正确,因为格式说明符的数量与参数数量相匹配 - 都是因为用户试图传递 %s以外的说明符。在团队将诊断文本更改为直接说明该函数仅接受 %s 占位符后,错误报告的涌入就停止了。改进分析生成的消息可以在最相关的地方解释哪里出了问题、为什么出了问题以及如何修复它,并且可以让开发人员在阅读消息时学到一些东西。

## 建议的修复Tricorder

检查还会在可能的情况下提供修复,如图20-2 所示。

```
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }

    public String getString() {
        return new String("foo");
    }
}
```

图 20-2.Critique 中静态分析 x 的示例视图

当消息不清楚时,自动修复可作为额外的文档来源,并且如前所述,可降低解决静态分析问题的成本。

可以直接在 Critique 中应用修复,也可以通过命令行工具对整个代码更改进行修复。虽然并非所有分析器都提供修复,但许多分析器都提供修复。我们采取的方法是,特别是样式问题应该自动修复;例如,通过自动重新格式化源代码文件的格式化程序。Google 为每种语言都提供了样式指南,其中指定了格式问题;指出格式错误并不是对人工审阅者时间的有效利用。审阅者每天点击“请修复”数千次,作者每天应用自动修复大约 3,000 次。

Tricorder 分析仪每天会收到 250 次“无用”的点击。

## 针对每个项目进行定制在通过仅显

示高置信度分析结果建立了用户信任基础之后,我们除了默认启用的分析器之外,还增加了在特定项目中运行其他“可选”分析器的功能。Proto Best Practices 分析器就是一个可选分析器的示例。此分析器突出显示可能破坏的数据

**协议**格式变更缓冲区 - Google 的语言无关的数据序列化格式。这些更改仅在序列化数据存储在某个地方（例如，在服务器日志中）时才会中断；没有存储序列化数据的项目的协议缓冲区不需要启用检查。我们还增加了自定义现有分析器的功能，尽管这种自定义通常受到限制，并且许多检查默认在整个代码库中统一应用。

有些分析器甚至一开始是可选的，后来根据用户反馈进行了改进，建立了庞大的用户群，然后在我们能够利用已建立的用户信任后逐渐转变为默认状态。例如，我们有一个分析器，它建议改进 Java 代码的可读性，但通常不会真正改变代码行为。Tricorder 用户最初担心这种分析太“嘈杂”，但最终希望获得更多的分析结果。

使此定制成功的关键见解是专注于项目级定制，而不是用户级定制。项目级定制可确保所有团队成员对其项目的分析结果有一致的看法，并防止出现一个开发人员试图解决问题而另一个开发人员引入问题的情况。

在 Tricorder 开发的早期，一组相对简单的样式检查器（“linters”）在 Critique 中显示结果，Critique 提供用户设置来选择显示结果的置信度级别，并隐藏特定分析的结果。我们从 Critique 中删除了所有这些用户可定制性，然后立即开始收到用户对烦人的分析结果的投诉。我们没有重新启用可定制性，而是询问用户为什么他们感到烦恼，并发现了 linters 的各种错误和误报。例如，C++ linter 也可以在 Objective-C 文件上运行，但产生了不正确、无用的结果。我们修复了 linting 基础架构，以便不再发生这种情况。HTML linter 的误报率极高，有用的信号很少，通常被编写 HTML 的开发人员抑制在视图之外。因为 linter 很少有用，所以我们干脆禁用了这个 linter。简而言之，用户定制导致了隐藏的错误和抑制反馈。

## 预提交

除了代码审查，Google 还有其他静态分析工作流集成点。由于开发人员可以选择忽略代码审查中显示的静态分析警告，因此 Google 还可以添加阻止提交待处理代码更改的分析，我们称之为提交前检查。提交前检查包括对更改内容或元数据的非常简单的可自定义内置检查，例如确保提交消息不显示“请勿提交”或测试文件始终包含在相应的代码文件中。团队还可以指定必须通过的一组测试或验证没有 Tricorder

特定类别的问题。提交前还会检查代码格式是否正确。

提交前检查通常在开发人员将更改发送邮件以供审核时运行，并在提交过程中再次运行，但它们可以在这些时间点之间临时触发。有关 Google 提交前检查的更多详细信息，请参阅[第 23 章](#)。

一些团队编写了自己的自定义预提交。这些是在基本预提交集之上的附加检查，增加了执行比整个公司更高的最佳实践标准的能力，并添加了特定于项目的分析。这使得新项目拥有比拥有大量遗留代码的项目更严格的最佳实践指南（例如）。特定于团队的预提交可能会使大规模变更（LSC）流程（参见[第 22 章](#)）更加困难，因此对于变更描述中带有“CLEANUP=”的变更，有些变更会被跳过。

### 编译器集成虽然使用静态分

析阻止提交很棒，但最好在工作流程的早期就通知开发人员问题。如果可能，我们会尝试将静态分析推入编译器。中断构建是一个无法忽略的警告，但在许多情况下是不可行的。然而，有些分析是高度机械的，没有有效的误报。一个例子是[容易出错的“ERROR”检查](#)。

这些检查均在 Google 的 Java 编译器中启用，从而防止错误实例再次引入我们的代码库。编译器检查需要快速进行，以免减慢构建速度。此外，我们强制执行以下三个标准（C++ 编译器也有类似的标准）：

- 可操作且易于修复（只要可能，错误应该包含建议  
可以机械应用的已解决修复）
- 不产生有效的误报（分析不应该停止构建  
正确的代码）
- 报告仅影响正确性而非风格或最佳实践的问题

要启用新检查，我们首先需要清理代码库中该问题的所有实例，这样我们就不会因为编译器的改进而破坏现有项目的构建。这也意味着部署新的基于编译器的检查的价值必须足够高，以保证修复所有现有实例。Google 已建立基础架构，可通过集群在整个代码库上并行运行各种编译器（如 clang 和 javac）作为 MapReduce 操作。当编译器以这种 MapReduce 方式运行时，运行的静态分析检查必须生成修复程序才能自动执行清理。在准备好并测试将修复程序应用于整个代码库的待处理代码更改后，我们提交该更改并删除问题的所有现有实例。然后，我们在编译器中启用检查，这样就不会在不破坏代码库的情况下提交问题的新实例。

构建。构建中断会在提交后由我们的持续集成 (CI) 系统捕获,或者在提交前通过预提交检查捕获(参见前面的讨论)。

我们还旨在永不发出编译器警告。我们反复发现开发人员忽略编译器警告。我们要么将编译器检查作为错误启用 (并中断构建),要么不在编译器输出中显示它。由于整个代码库使用相同的编译器标志,因此这个决定是全局做出的。不会中断构建的检查要么被抑制,要么在代码审查中显示 (例如通过 Tricorder)。虽然 Google 并非每种语言都有此政策,但最常用的语言都有。Java 和 C++ 编译器都已配置为避免显示编译器警告。Go 编译器将此发挥到了极致;其他语言认为是警告的一些东西 (例如未使用的变量或包导入) 在 Go 中是错误。

### 编辑和浏览代码时进行分析静态分析的另一个潜在集成点

是集成开发环境 (IDE)。但是,IDE 分析需要快速的分析时间 (通常少于 1 秒,理想情况下少于 100 毫秒),因此某些工具不适合在此处集成。此外,还存在确保相同分析在多个 IDE 中以相同方式运行的问题。我们还注意到,IDE 的流行度可能会时起时落 (我们不要求使用单一 IDE);因此 IDE 集成往往比插入审查流程更为混乱。代码审查在显示分析结果方面也具有特定优势。分析可以考虑到更改的整个上下文;某些分析可能对部分代码不准确 (例如,在添加调用站点之前实现函数时的死代码分析)。在代码审查中显示分析结果还意味着代码作者还必须说服审查者,如果他们想要忽略分析结果。也就是说,用于适当分析的 IDE 集成是显示静态分析结果的另一个好地方。

虽然我们主要关注显示新引入的静态分析警告或已编辑代码的警告,但对于某些分析,开发人员实际上确实希望能够在代码浏览期间查看整个代码库的分析结果。一些安全分析就是一个例子。Google 的特定安全团队希望全面了解问题的所有实例。开发人员还喜欢在计划清理时查看代码库的分析结果。换句话说,有时显示结果时,代码浏览是正确的选择。

## 结论

静态分析可以成为改进代码库、尽早发现错误并允许更昂贵的流程（例如人工审核和测试）专注于无法机械验证的问题的绝佳工具。通过提高静态分析基础架构的可扩展性和可用性，我们已使静态分析成为 Google 软件开发的有效组成部分。

## TL;DR

- 关注开发人员的满意度。我们投入了大量精力在我们的工具中建立分析用户和分析编写者之间的反馈渠道，并积极调整分析以减少误报的数量。
- 将静态分析作为核心开发人员工作流程的一部分。Google 静态分析的主要集成点是通过代码审查，其中分析工具提供修复并让审查人员参与。但是，我们还在其他点（通过编译器检查、门控代码提交、在 IDE 中以及在浏览代码时）集成分析。
- 让用户有动力做出贡献。我们可以利用领域专家的专业知识来扩展我们在构建和维护分析工具和平台方面的工作。  
开发人员不断添加新的分析和检查，使他们的生活更轻松，我们的代码库更完善。

## 第21章

# 依赖管理

作者:Titus Winters  
由 Lisa Carey 编辑

依赖管理（对我们无法控制的库、包和依赖关系网络的管理）是软件工程中最难理解、最具挑战性的问题之一。依赖管理主要关注以下问题：如何在外部依赖关系的版本之间进行更新？我们如何描述版本？我们的依赖关系中允许或预期哪些类型的更改？我们如何决定何时依赖其他组织编写的代码是明智的？

相比之下，这里最密切相关的主题是源代码控制。这两个领域都描述了我们如何处理源代码。源代码控制涵盖了较简单的部分：我们在哪里签入内容？我们如何将内容纳入构建？在我们接受基于主干的开发的价值之后，组织的大部分日常源代码控制问题都相当平凡：“我有了一个新东西，我应该将它添加到哪个目录中？”

依赖管理在时间和规模上都增加了额外的复杂性。在基于主干的源代码控制问题中，当您进行更改时，很明显您需要运行测试并且不能破坏现有代码。这是基于您在共享代码库中工作、了解事物的使用方式并可以触发构建和运行测试的想法。依赖管理侧重于在您的组织外部进行更改时出现的问题，而这些更改没有完全访问权限或可见性。由于您的上游依赖项无法与您的私有代码协调，因此它们更有可能破坏您的构建并导致您的测试失败。我们该如何管理呢？我们不应该采用外部依赖项吗？我们是否应该要求外部依赖项的发布之间具有更大的一致性？我们什么时候更新到新版本？

规模使所有这些问题变得更加复杂,因为我们意识到我们实际上并不是在谈论单个依赖项导入,而且在一般情况下,我们依赖于整个外部依赖项网络。当我们开始处理网络时,很容易构建出这样的场景:您的组织对两个依赖项的使用在某个时间点变得无法满足。通常,这种情况发生是因为一个依赖项在没有某些要求的情况下停止工作,而另一个依赖项与同一要求不兼容。关于如何管理单个外部依赖项的简单解决方案通常无法解释管理大型网络的现实情况。我们将在本章中花费大量篇幅讨论这些相互冲突的需求问题的各种形式。

源代码控制和依赖管理是两个相关问题,两者的区别在于“我们的组织是否控制这个子项目的开发 / 更新 / 管理?”例如,如果公司的每个团队都有单独的存储库、目标和开发实践,那么这些团队生成的代码的交互和管理将更多地与依赖管理有关,而不是源代码控制。另一方面,拥有(虚拟?)单一存储库(monorepo)的大型组织可以通过源代码控制策略进一步扩展 - 这是 Google 的方法。独立的开源项目当然算作独立的组织:未知且不一定合作的项目之间的相互依赖关系是一个依赖管理问题。也许我们在这个问题上最好的建议是:在其他条件相同的情况下,优先考虑源代码控制问题而不是依赖管理问题。如果您可以选择更广泛地重新定义“组织”(整个公司而不仅仅是一个团队),那么这通常是一个很好的权衡。与依赖管理问题相比,源代码控制问题更容易思考,处理起来也更便宜。

随着开源软件(OSS)模型不断发展并扩展到新领域,许多流行项目的依赖关系图也随着时间的推移不断扩大,依赖关系管理可能正在成为软件工程政策中最重要的问题。我们不再是建立在 API 之外的一两层上的互不相连的孤岛。现代软件建立在高耸的依赖关系支柱之上;但仅仅因为我们可以构建这些支柱,并不意味着我们已经找到了如何让它们长期保持稳定的方法。

在本章中,我们将研究依赖管理的特殊挑战,探索解决方案(常见和新颖)及其局限性,并研究使用依赖的现实情况,包括我们在 Google 中如何处理事情。首先要承认:我们对这个问题投入了大量思考,并且在重构和维护问题方面拥有丰富的经验

---

<sup>1</sup>这可以是任何语言版本、底层库版本、硬件版本、操作系统、编译器标志、编译器版本等等。

这表明现有方法存在实际缺陷。我们没有第一手证据表明解决方案在大规模组织中效果良好。在某种程度上,本章总结了我们所知道的不起作用的方法(或至少在更大规模下可能不起作用)以及我们认为可能取得更好结果的方法。我们绝对不能声称这里有所有的答案;如果我们可以,我们就不会称其为软件工程中最重要的问题之一。

## 为什么依赖管理如此困难?

甚至定义依赖管理问题也带来了一些不寻常的挑战。这个领域中许多不成熟的解决方案都集中在一个过于狭隘的问题表述上:“我们如何导入本地开发的代码可以依赖的包?”这是一个必要但不充分的表述。诀窍不只是找到一种方法来管理一个依赖关系,而诀窍在于如何管理依赖关系网络及其随时间的变化。这个网络的某些子集对你的第一方代码来说是直接必要的,而有些子集仅由传递依赖项引入。在足够长的一段时间内,该依赖关系网络中的所有节点都会有新版本,其中一些更新将非常重要。<sup>2</sup>我们如何管理依赖关系网络其余部分由此产生的级联升级?或者具体来说,鉴于我们不控制这些依赖关系,如何轻松找到所有依赖关系的相互兼容版本?我们如何分析我们的依赖关系网络?我们如何管理该网络,特别是面对不断增长的依赖关系图?

冲突需求和菱形依赖管理的核心问题凸显了以依赖网络而非单个

依赖的角度进行思考的重要性。大部分困难源于一个问题:当依赖网络中的两个节点有冲突需求,而你的组织同时依赖它们时会发生什么?这种情况可能由多种原因引起,从平台考虑(操作系统[OS]、语言版本、编译器版本等)到更为常见的版本不兼容问题。版本不兼容作为无法满足的版本需求的典型例子是菱形依赖问题。虽然我们通常不会在依赖图中包含诸如“你使用的编译器版本是什么”之类的内容,但大多数这些冲突需求问题都同构于“在依赖图中添加一个表示此需求的(隐藏)节点”。因此,我们将主要从菱形依赖的角度讨论冲突需求,但请记住,libbase实际上可能是

---

<sup>2</sup>例如,安全漏洞、弃用、处于具有安全漏洞的更高级别依赖项的依赖集中等等。

绝对是任何参与构建依赖网络中两个或更多节点的软件。

菱形依赖问题和其他形式的冲突需求至少需要三层依赖,如图21-1 所示。

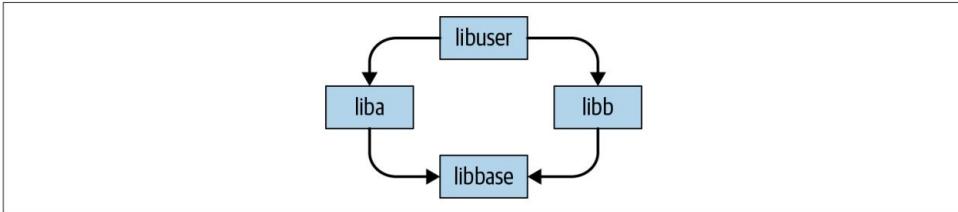


图 21-1 钻石依赖问题

在这个简化的模型中, libbase由liba和libb使用,而liba和libb都由更高级别的组件libuser 使用。如果libbase引入了不兼容的更改,则liba和libb作为不同组织的产品可能不会同时更新。如果liba依赖于新的libbase版本,而libb依赖于旧版本,则libuser (即您的代码)没有通用的方法将所有内容放在一起。这种菱形可以在任何规模上形成:在您的整个依赖关系网络中,如果有一个低级节点需要同时处于两个不兼容的版本 (由于从某个更高级别的节点到这两个版本有两条路径) ,就会出现问题。

不同的编程语言对菱形依赖问题的容忍程度不同。对于某些语言,可以在构建中嵌入依赖项的多个 (独立) 版本:从liba调用libbase可能会调用与从libb 调用libbase不同版本的同一 API。例如,Java 提供了相当完善的机制来重命名此类依赖项提供的符号。<sup>3</sup>与此同时,C++ 在正常构建中对菱形依赖项几乎零容忍,它们很可能触发任意错误和未定义行为 (UB),因为这明显违反了 C++ 的“**单一定义规则**”。您最多可以使用与 Java 的着色类似的想法来隐藏动态链接库 (DLL) 中的某些符号,或者在您单独构建和链接的情况下隐藏它们。但是,在我们所知的所有编程语言中,这些解决方法充其量只是部分解决方案:可以通过调整函数名称来嵌入多个版本,但如果存在在依赖项之间传递的类型,则所有赌注都无效。例如,根本无法以语义一致的方式将libbase v1中定义的映射通过某些库传递给libbase v2 提供的 API。特定于语言的黑客可以隐藏或重命名单独的实体

<sup>3</sup>这称为阴影或版本控制。

编译速度较快的库可以为钻石依赖问题提供一些缓冲,但在一般情况下并不是解决方案。

如果遇到冲突的需求问题,唯一简单的答案就是跳过这些依赖项的版本向前或向后移动,以找到兼容的版本。如果这不可能,我们必须诉诸于本地修补有问题的依赖项,这尤其具有挑战性,因为提供者和消费者中不兼容的原因可能对于最先发现不兼容性的工程师来说都是未知的。这是固有的: liba开发人员仍在以兼容的方式使用libbase v1,而libb开发人员已经升级到 v2。只有同时参与这两个项目的开发人员才有机会发现这个问题,而且肯定不能保证他们对libbase和liba足够熟悉以完成升级。更简单的答案是降级libbase和libb,尽管如果最初由于安全问题而强制升级,那么这不是一个选择。

依赖管理的政策和技术体系很大程度上可以归结为这样一个问题:“我们如何避免冲突的需求,同时仍然允许非协调群体之间进行变更?”如果你有一个针对菱形依赖问题的一般形式的解决方案,可以适应网络各个级别不断变化的需求(依赖关系和平台需求)的现实,那么你就描述了依赖管理解决方案的有趣部分。

## 导入依赖项

从编程角度来说,重用一些现有的基础设施显然比自己构建更好。这是显而易见的,也是技术进步的基本趋势之一。如果每个新手都必须重新实现自己的 JSON 解析器和正则表达式引擎,我们将一事无成。重用是有益的,尤其是与从头开始重新开发高质量软件的成本相比。只要您没有下载带有木马的软件,并且您的外部依赖项满足您的编程任务的要求,您就应该使用它。

兼容性承诺当我们开始考虑时

间时,情况就会出现一些复杂的权衡。仅仅因为您可以避免开发成本并不意味着导入依赖项是正确的选择。在一个了解时间和变化的软件工程组织中,我们还需要考虑其持续的维护成本。即使我们导入依赖项而无意升级它,发现的安全漏洞、不断变化的平台和不断发展的依赖项网络都可能合谋强制进行升级,无论我们的意图如何。到那一天到来时,成本会有多高?有些依赖项比其他依赖项更明确地说明了仅使用该依赖项的预期维护成本:兼容性有多高

假设?假设有多少发展?如何处理变化?发布支持多长时间?

我们建议依赖项提供商应该更清楚地了解这些问题的答案。以拥有数百万用户的大型基础设施项目及其兼容性承诺为例。

## C++

对于 C++ 标准库,该模型是几乎无限的向后兼容性之一。针对旧版本标准库构建的二进制文件有望与较新的标准一起构建和链接;该标准不仅提供 API 兼容性,还提供二进制工件的持续向后兼容性,称为 ABI 兼容性。维护这一原则的程度因平台而异。对于 Linux 上的 gcc 用户来说,大多数代码在大约十年的时间内都运行良好。该标准没有明确指出其对 ABI 兼容性的承诺。关于这一点没有面向公众的政策文件。但是,该标准确实发布了常设[文件 8](#) (SD-8)指出了标准库可以在版本之间进行的一小部分更改类型,隐式定义了需要为哪些类型的更改做好准备。Java 类似:源代码在语言版本之间兼容,旧版本的 JAR 文件可以轻松与

更新版本。

去

并非所有语言都以相同程度的兼容性为优先考虑。Go 编程语言明确承诺大多数版本之间的源代码兼容性,但不保证二进制兼容性。您无法使用某个版本的 Go 语言构建库,然后将该库链接到使用其他版本的 Go 语言构建的 Go 程序中。

## 绳降

Google 的 Abseil 项目与 Go 非常相似,但有一个重要的时间限制。我们不愿意无限期地承诺兼容性:Abseil 是我们内部大多数计算量最大的服务的基础,我们认为这些服务可能会在未来很多年内使用。这意味着我们会谨慎地保留进行更改的权利,特别是在实施细节和 ABI 方面,以便获得更好的性能。我们经历过太多 API 事后变得令人困惑和容易出错的情况;将这种已知错误发布给数以万计的开发人员,让他们无限期地了解情况,这感觉不对。在内部,我们已经有大约 2.5 亿行依赖于此库的 C++ 代码。我们不会轻易更改 API,但必须有可能。为此,Abseil 明确不承诺 ABI 兼容性,但承诺一种略微有限的 API 兼容性:我们不会在没有同时做出重大 API 更改的情况下做出重大更改

提供一个自动重构工具,可以透明地将代码从旧 API 转换为新 API。我们认为,这将大大降低意外成本的风险,使用户受益:无论依赖项是针对哪个版本编写的,该依赖项和 Abseil 的用户都应该能够使用最新版本。最高成本应该是“运行此工具”,并且可能将生成的补丁发送到中级依赖项( `liba` 或 `libb`, 继续我们之前的示例)中进行审查。实际上,该项目足够新,我们不必进行任何重大的 API 破坏性更改。我们无法说这对整个生态系统的效果如何,但从理论上讲,这似乎是稳定性与升级便利性之间的良好平衡。

## 促进

相比之下,Boost C++ 库不保证版本之间的兼容性。当然,大多数代码都不会改变,但是“许多 Boost 库都在积极维护和改进,因此并不总是可以向后兼容以前的版本”。建议用户仅在项目生命周期的某个时期进行升级,在此期间,某些更改不会造成问题。Boost 的目标与标准库或 Abseil 有着根本的不同:Boost 是一个实验性的试验场。Boost 流的某个特定版本可能非常稳定,适用于许多项目,但 Boost 的项目目标并不优先考虑版本之间的兼容性 其他长期项目可能会在保持最新状态方面遇到一些阻力。Boost 开发人员与标准库 4 的开发人员一样专业 这一切都与技术专长无关;这纯粹是一个项目承诺或不承诺什么以及优先考虑什么的问题。

纵观本次讨论中的库,重要的是要认识到这些兼容性问题是软件工程问题,而不是编程问题。您可以下载像 Boost 这样没有兼容性保证的库,并将其深深嵌入到组织中最关键、最长期的系统中;它会正常工作。这里的所有问题都是关于这些依赖项将如何随时间变化、如何跟上更新,以及让开发人员担心维护而不仅仅是让功能正常工作的难度。在 Google 内部,我们不断为工程师提供指导,帮助他们考虑“我让它工作了”和“它以受支持的方式工作”之间的区别。这并不奇怪:毕竟,这是海勒姆定律的基本应用。

更广泛地说:重要的是要意识到依赖管理在编程任务和软件工程任务中具有完全不同的性质。如果你处在一个与长期维护相关的问题空间中,依赖管理就很困难。如果你纯粹是在开发一个不需要更新任何东西的解决方案,那么抓住尽可能多的现成的依赖管理是完全合理的

---

<sup>4</sup> 在许多情况下,这些人群存在很大的重叠。

依赖项随心所欲,无需考虑如何负责任地使用它们或计划升级。通过违反 SD-8 中的所有内容并依赖 Boost 和 Abseil 的二进制兼容性让您的程序今天运行起来是可行的……只要您永远不会升级标准库、Boost 或 Abseil,也不会升级任何依赖于您的内容。

### 导入时的注意事项导入依赖项以用于编程项

目几乎是免费的:假设您花时间确保它能够满足您的需要并且不存在秘密的安全漏洞,那么重用几乎总是比重新实现功能更便宜。

即使该依赖项已采取措施明确其将做出的兼容性承诺,只要我们永远不会升级,那么在该依赖项快照之上构建的任何内容都是没问题的,无论您在使用该 API 时违反了多少规则。但是当我们从编程转向软件工程时,这些依赖项的成本会变得更加昂贵,并且还有许多隐藏的成本和问题需要回答。希望您在导入之前考虑这些成本,并且希望您知道自己是在从事编程项目还是在从事软件工程项目。

当 Google 工程师尝试导入依赖项时,我们鼓励他们首先询问以下(不完整)问题列表:

- 该项目是否有您可以运行的测试? · 这些测试通过了吗?
- 谁在提供该

依赖项?即使在“不提供任何明示或暗示保证”的 OSS 项目中,经验和技能也有很大差异  
依赖 C++ 标准库或 Java 的 Guava 库的兼容性与从 GitHub 或 npm 中随机选择一个项目是完全不同的。声誉不是一切,但值得研究。  
· 该项目追求什么样的兼容性?  
· 该项目是否详细说明了预计支持什么样的用途?  
· 该项目有多受欢迎?  
· 我们将依赖这个项目多久?  
· 项目多久进行一次重大更改?

除此之外,还要添加一些简短的内部问题:

- 在 Google 内部实现该功能有多复杂?  
· 我们有什么动力保持此依赖项最新?  
· 谁将执行升级?

- 我们预计升级有多困难？

我们自己的 Russ Cox 对此进行了更广泛的描述。我们无法给出一个完美的公式来决定从长远来看是进口还是重新实施更便宜；我们自己在这方面常常失败。

## Google 如何处理导入依赖项

简而言之：我们可以做得更好。

任何 Google 项目中的绝大多数依赖项都是内部开发的。这意味着我们内部依赖项管理的绝大部分实际上并不是依赖项管理，而只是源代码控制。这是设计使然。正如我们所提到的，当提供者和消费者属于同一组织并且具有适当的可见性和持续集成（CI；参见第 23 章）时，管理和控制添加依赖项所涉及的复杂性和风险要容易得多。当您可以确切地看到代码是如何被使用的，并且确切地知道任何给定更改的影响时，依赖项管理中的大多数问题就不再是问题了。源代码控制（当您控制相关项目时）比依赖项管理（当您不控制时）容易得多。

当我们处理外部项目时，这种易用性就开始失效了。对于我们从 OSS 生态系统或商业合作伙伴导入的项目，这些依赖项被添加到 monorepo 的一个单独目录中，标记为 third\_party。让我们来看看如何将新的 OSS 项目添加到 third\_party。

Alice 是 Google 的一名软件工程师，她正在从事一个项目，并意识到有一个开源解决方案可用。她非常希望尽快完成并演示这个项目，以便在休假之前完成它。然后，她需要选择是从头开始重新实现该功能，还是下载 OSS 包并将其添加到 third\_party。Alice 很可能认为更快的开发解决方案更有意义：她下载了该包并按照我们的 third\_party 政策中的几个步骤进行操作。这是一个相当简单的检查表：确保它使用我们的构建系统进行构建，确保没有该包的现有版本，并确保至少有两名工程师注册为 OWNERS，以便在需要维护时维护该包。Alice 让她的队友 Bob 说：“是的，我会帮忙。”他们俩都不需要有维护 third\_party 包的经验，而且他们巧妙地避免了了解有关此包实现的任何信息。他们最多只是在用它解决撤离前演示问题的过程中获得了一点对其界面的经验。

从此时起，其他 Google 团队通常都可以在自己的项目中使用该软件包。添加其他依赖项的行为对 Alice 和 Bob 来说是完全透明的：他们可能完全不知道他们下载并承诺维护的软件包已经变得流行。即使他们

正在监控其软件包的新直接使用情况,他们可能不一定会注意到其软件包的传递使用量的增长。如果他们将其用于演示,而 Charlie 在我们的搜索基础架构中添加了依赖项,则该软件包将突然从相当无害的软件包转变为重要 Google 系统的关键基础架构。但是,当 Charlie 考虑是否添加此依赖项时,我们并没有向他发出任何特定信号。

现在,这种情况可能完全没问题。也许该依赖项写得很好,没有安全漏洞,并且不依赖于其他 OSS 项目。它可能会在几年内不更新。这种情况并不一定是明智的:外部更改可能已经对其进行了优化或添加了重要的新功能,或者在发现 CVE5 之前清理了安全漏洞。包存在的时间越长,依赖项(直接和间接)就可能越多。包保持的稳定性越高,我们就越有可能对签入 third\_party 的版本的细节产生 Hyrum 定律依赖。

有一天,Alice 和 Bob 被告知升级至关重要。这可能是因为软件包本身或依赖于它的 OSS 项目中存在安全漏洞,从而迫使他们进行升级。Bob 已转任管理层,有一段时间没有碰过代码库了。Alice 在演示之后就转到了另一个团队,再也没有使用过这个软件包。没有人更改 OWNERS 文件。数千个项目间接依赖于此 - 我们不能删除它,否则会破坏 Search 和其他十几个大团队的构建。没有人对这个软件包的实现细节有任何经验。Alice 所在的团队不一定有丰富的经验来消除随着时间的推移而积累的海勒姆定律的微妙之处。

所有这些都表明:Alice 和此软件包的其他用户将面临昂贵且困难的升级,安全团队施加压力要求立即解决此问题。在这种情况下,没有人有执行升级的经验,而且升级特别困难,因为它涵盖了许多较小的版本,涵盖了从软件包首次引入 third\_party 到安全披露的整个时期。

我们的第三方政策不适用于这些不幸的常见情况。我们大致明白,我们需要更高的所有权标准,我们需要让定期更新变得更容易(也更有回报),让第三方软件包更难同时成为孤儿和重要。困难在于,代码库维护者和第三方领导很难说:“不,你不能使用这个完美解决你的开发问题的东西,因为我们没有资源不断为每个人更新新版本。”受欢迎的项目

---

## 5 个常见的漏洞和暴露

没有兼容性保证（如 Boost）的风险特别大：我们的开发人员可能非常熟悉使用该依赖项来解决 Google 之外的编程问题，但让它根深蒂固地融入我们的代码库结构中是一个巨大的风险。目前，我们的代码库的预期寿命为数十年：上游项目没有明确优先考虑稳定性是一种风险。

## 依赖管理的理论

了解了依赖管理的困难之处及其可能出现的问题之后，让我们更具体地讨论一下我们要解决的问题以及如何解决这些问题。在本章中，我们将回顾这一表述：“我们如何管理来自组织外部（或我们无法完美控制）的代码，如何更新它，如何管理它随时间推移所依赖的东西？”我们需要清楚，任何好的解决方案都会避免任何形式的需求冲突，包括菱形依赖版本冲突，即使在可能添加新依赖项或其他需求（在网络中的任何一点）的动态生态系统中也是如此。我们还需要意识到时间的影响：所有软件都有错误，其中一些对安全至关重要，因此，在足够长的时间内，更新一部分依赖项至关重要。

因此，稳定的依赖管理方案必须能够随着时间和规模的变化而灵活变化：我们不能假设依赖关系图中任何特定节点具有无限稳定性，也不能假设没有添加新的依赖项（无论是在我们控制的代码中还是在我们依赖的代码中）。如果依赖管理解决方案可以防止依赖项之间出现冲突的需求问题，那么它就是一个好的解决方案。如果它能够做到这一点，并且不假设依赖版本或依赖项扇出的稳定性、组织之间的协调或可见性或大量计算资源，那么它就是一个很好的解决方案。

在提出依赖管理解决方案时，我们知道有四个常见选项，它们至少具有一些适当的属性：永不改变、语义版本控制、捆绑所需的一切（不是按项目协调，而是按发行版协调）、或 Live at Head。

### 什么都没有改变（又称静态依赖模型）

确保依赖关系稳定的最简单方法是永远不改变它们：不改变 API，不改变行为，什么都不做。只有在不会破坏用户代码的情况下才允许修复错误。这样一来，兼容性和稳定性就高于一切。显然，由于假设了无限的稳定性，这种方案并不理想。如果我们以某种方式进入一个安全问题和错误修复都不是问题、依赖关系不会改变的世界，那么“无变化”模型就非常有吸引力。如果我们从可满足的约束开始，我们将能够无限期地保持这一属性。

虽然从长远来看,这种模式并不可持续,但实际上,每个组织都是从这里开始的:直到你证明你的项目预期寿命足够长,以至于有必要进行更改,否则我们很容易生活在一个假设什么都不会改变的世界中。同样值得注意的是:这可能是大多数新组织的正确模式。你很少知道你正在启动一个将持续数十年的项目,并且需要能够顺利更新依赖关系。更合理的选择是希望稳定性是一个真正的选择,并假装依赖关系在项目的头几年是完全稳定的。

这种模型的缺点是,在足够长的时间内,它都是错误的,并且没有明确的迹象表明你可以假装它是合法的多长时间。

我们没有针对安全漏洞或其他可能迫使您升级依赖项的严重问题的长期预警系统 - 并且由于依赖链,单次升级理论上可能会成为对整个依赖网络的强制更新。

在这个模型中,版本选择很简单:不需要做任何决定,因为没有版本。

#### 语义版本控制 “我们今天如何管

理依赖关系网络?”的事实标准是语义版本控制 (SemVer)。<sup>6</sup> SemVer 是一种几乎无处不在的做法,它使用三个小数分隔的整数表示某些依赖项 (尤其是库) 的版本号,例如 2.4.72 或 1.1.4。在最常见的约定中,这三个组成数字代表主版本、次版本和补丁版本,这意味着更改的主版本号表示对现有 API 的更改可能会破坏现有用法,更改的次版本号表示纯粹添加的功能,不会破坏现有用法,更改的补丁版本保留用于不影响 API 的实现细节和被视为特别低风险的错误修复。

由于 SemVer 区分了主要版本/次要版本/补丁版本,因此假设版本要求通常可以表示为“任何比其更新的版本”,除非 API 不兼容的更改 (主要版本更改)。通常,我们会看到“需要 libbase  $\geq 1.5$ ”,该要求与 1.5 中的任何 libbase 兼容,包括 1.5.1,以及 1.6 及以后的任何版本,但不兼容 libbase 1.4.9 (缺少 API

---

<sup>6</sup>严格来说,SemVer 仅指将语义应用于主要/次要/补丁版本号的新兴实践,而不是以此方式编号的依赖项之间应用兼容版本要求。不同生态系统对这些要求有许多细微的差异,但总体而言,此处描述的 SemVer 版本号约束系统代表了整个实践。

在 1.5 版中引入)或 2.x 版 ( libbase 中的一些 API 发生了不兼容的更改)。主要版本更改是显著的不兼容性:因为现有功能已更改 (或被删除),所以所有依赖项都可能存在不兼容性。只要一个依赖项使用另一个依赖项,就会存在版本要求 (显式或隐式):我们可能会看到“liba 需要 libbase  $\geq 1.5$ ”和“libb 需要 libbase  $\geq 1.4.7$ ”。

如果我们将这些要求形式化,我们可以将依赖关系网络概念化为软件组件 (节点) 及其之间要求 (边) 的集合。此网络中的边标签会根据源节点的版本而变化,无论是添加 (或删除) 依赖项,还是由于源节点的变化 (例如,需要依赖项中新添加的功能) 而更新 SemVer 要求。由于整个网络会随着时间的推移而异步变化,因此找到一组相互兼容的依赖项以满足应用程序的所有传递要求的过程可能具有挑战性。<sup>7</sup> SemVer 的版本可满足性求解器与逻辑和算法研究中的 SAT 求解器非常相似:给定一组约束 (依赖关系边上的版本要求),我们能否找到一组满足所有约束的相关节点的版本?大多数包管理生态系统都建立在这些类型的图之上,由其 SemVer SAT 求解器管理。

SemVer 及其 SAT 解析器绝不保证存在给定依赖约束集的解决方案。依赖约束无法满足的情况不断出现,正如我们已经看到的:如果较低级别的组件 (libbase) 的主编号增加,并且依赖它的一些库 (但不是全部) 已升级 (libb 但不是 liba),我们将遇到菱形依赖问题。

依赖项管理的 SemVer 解决方案通常基于 SAT 求解器。版本选择就是运行某种算法来找到网络中依赖项的版本分配,以满足所有版本要求约束。

当不存在令人满意的版本分配时,我们通俗地称之为“依赖地狱”。

我们将在本章后面更详细地讨论 SemVer 的一些局限性。

## 捆绑分销模式

作为一个行业,几十年来我们已经看到了强大的依赖关系管理模型的应用:一个组织收集依赖项集合,找到一组相互兼容的依赖项,然后将该集合作为一个单元发布。

例如,Linux 发行版就会发生这种情况 没有任何保证

---

<sup>7</sup>事实上,已经证明应用于依赖网络的 SemVer 约束是 NP 完全的。

发行版中包含的各个部分都是从同一时间点开始的。事实上,较低级别的依赖项可能比较高级别的依赖项更旧,这只是为了说明集成它们所需的时间。

这种“在所有内容周围画一个更大的框并发布该集合”的模型引入了全新的参与者:分发者。尽管所有单个依赖项的维护者可能对其他依赖项知之甚少或一无所知,但这些更高级别的分发者参与了查找、修补和测试一组相互兼容的版本的过程。分发者是负责提出一组要捆绑在一起的版本、测试这些版本以查找依赖关系树中的错误并解决任何问题的工程师。

对于外部用户来说,只要您能够正确依赖其中一个捆绑发行版,这种方法就很好用。这实际上与将依赖关系网络更改为单个聚合依赖关系并为其赋予版本号相同。

而不是说“我依赖这些版本的 72 个库”,而是“我依赖 RedHat 版本 N”,或者“我依赖时间 T 的 NPM 图中的部分”。

在捆绑分发方法中,版本选择由专门的分销商处理。

## 住在 Head

Google<sup>8</sup> 中的一些人一直在推动的模型在理论上是可行的,但却给依赖关系网络中的参与者带来了新的、昂贵的负担。它与当今 OSS 生态系统中存在的模型完全不同,而且目前尚不清楚如何将其作为一个行业从这里转移到那里。在像 Google 这样的组织范围内,它虽然成本高昂但有效,我们认为它将大部分成本和激励放在了正确的地方。我们称这种模式为“Live at Head”。它可以被视为基于主干的开发的依赖管理扩展:基于主干的开发讨论源代码控制策略,我们正在扩展该模型以应用于上游依赖关系。

Live at Head 假设我们可以解除依赖关系、放弃 SemVer,并依靠依赖提供者在提交之前针对整个生态系统测试更改。Live at Head 明确尝试从依赖管理问题中抽出时间和选择:始终依赖所有内容的当前版本,并且永远不会以依赖项难以适应的方式更改任何内容。(无意中)改变 API 或行为的更改通常会被下游依赖项的 CI 捕获,因此不应提交。对于以下情况

---

<sup>8</sup>特别是作者和 Google C++ 社区的其他人。

必须发生这种更改（即出于安全原因），只有在更新下游依赖项或提供自动化工具来执行更新后，才应进行此类中断。（此工具对于闭源下游消费者至关重要：目标是允许任何用户能够在不具备使用或 API 专业知识的情况下更新不断变化的 API 的使用。该属性大大减轻了“大多数旁观者”因中断更改而产生的成本。）开源生态系统中这种责任的哲学转变最初很难推动：将测试和更改所有下游客户的负担放在 API 提供商身上是对 API 提供商职责的重大修改。

Live at Head 模型中的变更不会简化为 SemVer “我认为这是否安全”。相反，测试和 CI 系统用于针对可见的依赖项进行测试，以通过实验确定变更的安全性。因此，对于仅改变效率或实现细节的变更，所有可见受影响的测试都可能通过，这表明该变更不会明显影响用户 - 可以安全提交。修改 API 中更明显可观察部分（语法或语义）的变更通常会导致数百甚至数千个测试失败。

然后由提议变更的作者决定解决这些失败所涉及的工作是否值得提交变更的结果价值。如果做得好，作者将与所有依赖者合作，提前解决测试失败（即，消除测试中的脆弱假设），并可能创建一个工具来执行尽可能多的必要重构。

这里的激励结构和技术假设与其他场景有实质性不同：我们假设存在单元测试和 CI，我们假设 API 提供者将受下游依赖项是否会被破坏的约束，我们假设 API 消费者保持他们的测试通过并以受支持的方式依赖他们的依赖项。这在开源生态系统中（可以提前分发修复程序）比在闭源/闭源依赖项面前效果要好得多。API 提供者在进行更改时会受到激励，以便能够顺利迁移到 API 消费者受到激励以保持他们的测试正常运行，以免被标记为低信号测试并可能被跳过，从而降低该测试提供的保护。

在 Live at Head 方法中，版本选择是通过询问“所有内容的最新稳定版本是什么？”来处理的。如果提供商负责任地做出更改，那么所有内容都将顺利协同工作。

## SemVer 的局限性

Live at Head 方法可能建立在公认的版本控制实践（基于主干的开发）之上，但在规模上尚未得到证实。SemVer 是当今依赖管理的事实标准，但正如我们所建议的，它并非没有

它的局限性。由于它是一种非常流行的方法,因此值得更详细地研究它,并强调我们认为它的潜在缺陷。

在 SemVer 定义中,有很多东西需要解读,即点分三组版本号的真正含义。这是一个承诺吗?还是为发布选择的版本号是一个估计值?也就是说,当 libbase 的维护者发布新版本并选择这是主要版本、次要版本还是补丁版本时,他们在说什么?是否可以证明从 1.1.4 升级到 1.2.0 是安全且容易的,因为只有 API 添加和错误修复?当然不是。libbase 的不良用户可能会做很多事情,这些事情可能会导致构建中断或行为更改,而这也只是一个“简单”的 API 添加。<sup>9</sup>从根本上说,如果只考虑源 API,你无法证明任何有关兼容性的事情;你必须知道你询问的是哪些兼容性。

然而,当我们谈论依赖关系网络和应用于这些网络的 SAT 求解器时,这种“估计”兼容性的想法开始变得薄弱。这个公式的基本问题是传统 SAT 中的节点值与 SemVer 依赖关系图中的版本值之间的差异。三 SAT 图中的一个节点要么是 True,要么是 False。依赖关系图中的版本值 (1.1.14) 由维护者提供,作为对使用先前版本的代码的新版本兼容性的估计。我们正在一个不稳定的基础上构建我们所有的版本满意度逻辑,将估计和自我证明视为绝对的。

正如我们所看到的,即使在有限的情况下它可以正常工作,但从总体上看,它不一定具有足够的保真度来支撑健康的生态系统。

如果我们承认 SemVer 是一种有损估计,并且仅代表可能更改范围的一部分,那么我们可以开始将其视为一种钝器。从理论上讲,它作为一种简写形式效果很好。在实践中,尤其是当我们在其基础上构建 SAT 求解器时,SemVer 可能会(并且确实)因过度限制和保护不足而使我们失望。

SemVer 可能会过度约束考虑一下当

libbase 被识别为不止一个整体时会发生什么:库中几乎总是有独立的接口。即使只有两个函数,我们也可以看到 SemVer 过度约束我们的情况。想象一下,libbase 确实只由两个函数组成,Foo 和 Bar。我们的中级依赖项 liba 和 libb 仅使用 Foo。如果 libbase 的维护者对 Bar 进行了重大更改,他们有责任提高 lib 的主要版本

---

<sup>9</sup> 例如:一个实施不佳的 polyfill 提前添加了新的 libbase API,导致定义冲突。或者,使用语言反射 API 来依赖 libbase 提供的 API 的精确数量,如果该数量发生变化,则会导致崩溃。这些不应该发生,即使意外发生,也非常罕见。关键在于 libbase 提供商无法证明兼容性。

在 SemVer 世界中， liba 和 libb 依赖于 libbase 1.x。 SemVer 依赖解析器不会接受该依赖的 2.x 版本。然而，实际上这些库可以完美地协同工作：只有 Bar 发生了变化，而 Bar 未被使用。当“我做了重大更改；我必须改变主版本号”中固有的压缩不适用于单个原子 API 单元的粒度时，这种压缩是有损的。尽管某些依赖项可能足够细粒度以达到准确度，<sup>10</sup>但这并不是 SemVer 生态系统的常态。

如果由于不必要的严重版本升级或对 SemVer 编号的应用不够细粒度而导致 SemVer 过度约束，自动包管理器和 SAT 解析器将报告您的依赖项无法更新或安装，即使忽略 SemVer 检查后所有内容都可以完美地协同工作。任何在升级过程中经历过依赖地狱的人都会发现这特别令人愤怒：大部分努力完全是在浪费时间。

SemVer 可能过度承诺另一方面，

SemVer 的应用明确假设 API 提供商对兼容性的估计可以完全预测，并且更改分为三类：破坏性（通过修改或删除）、严格添加或不影响 API。如果 SemVer 通过对句法和语义变化进行分类，完美地表示了变更的风险，那么我们如何描述为时间敏感的 API 增加一毫秒延迟的更改？或者更合理的是：我们如何描述改变日志输出格式的更改？或者改变导入外部依赖项的顺序的更改？或者改变结果在“无序”流中返回的顺序的更改？仅仅因为这些更改不是所讨论 API 的语法或契约的一部分，就可以合理地假设这些更改是“安全的”吗？如果文档说“这可能会在未来改变”怎么办？或者 API 被命名为“ForInternalUseByLibBaseOnlyDoNotTouchThisIReallyMeantIt？”<sup>11</sup> SemVer 补丁版本在理论上只改变了实现细节，认为它们是“安全”的更改，这绝对违背了 Google 对海勒姆定律 (Hylrum’s Law) 的经验：“有了足够多的用户，你的系统的每个可观察到的行为都会被某人所依赖。”更改依赖项的导入顺序，或更改“无序”生产者的输出顺序，在规模上，不可避免地会打破某些消费者所依赖的假设（也许是错误的）。“重大更改”一词本身就具有误导性：有些更改在理论上是重大的，但在实践中是安全的（删除未使用的 API）。还有

---

<sup>10</sup> Node 生态系统具有值得注意的依赖关系示例，这些依赖关系仅提供一个 API。

<sup>11</sup> 值得注意的是：根据我们的经验，这样的命名并不能完全解决用户访问私有 API 的问题。最好使用能够很好地控制所有形式的 API 的公共/私有访问的语言。

理论上安全但在实践中会破坏客户端代码的更改（我们之前的任何海勒姆定律示例）。我们可以在任何 SemVer/依赖管理系统中看到这一点，对于这些系统，版本号要求系统允许对补丁号进行限制：如果您可以说 liba 需要 libbase >1.1.14 而不是 liba 需要 libbase 1.1，这显然是承认补丁版本之间存在可观察到的差异。

单独的变更既不是破坏性的也不是非破坏性的，只能根据其使用方式来评估该说法。“这是一项破坏性变更”这一概念没有绝对的真理；一项变更只能对一组（已知或未知的）现有用户和用例产生破坏性影响。我们如何评估变更的现实本质上依赖于依赖管理的 SemVer 公式中不存在的信息：下游用户如何使用此依赖项？

因此，SemVer 约束求解器可能会报告您的依赖项协同工作，但实际上它们并不协同工作，这可能是因为应用了不正确的碰撞，或者是因为您的依赖网络中的某些东西对某些不被视为可观察 API 表面一部分的东西具有 Hyrum 定律依赖性。在这些情况下，您可能存在构建错误或运行时错误，其严重性没有理论上的上限。

## 动机

还有一种观点认为，SemVer 并不总是能激励创建稳定的代码。对于任意依赖项的维护者来说，存在可变的系统激励来避免进行重大更改并升级主要版本。一些项目非常关心兼容性，并会竭尽全力避免升级主要版本。其他项目则更为激进，甚至会故意按照固定的时间表升级主要版本。问题是，任何给定依赖项的大多数用户都是间接用户。他们没有任何重要的理由了解即将发生的更改。即使是大多数直接用户也不会订阅邮件列表或其他发布通知。

所有这些结合起来表明，无论采用流行 API 的不兼容更改会给多少用户带来不便，维护者都只承担了由此产生的版本升级成本的一小部分。对于同时也是用户的维护者来说，也可能存在破坏性的动机：在没有遗留约束的情况下，设计更好的界面总是更容易。这就是为什么我们认为项目应该发布关于兼容性、使用和重大更改的明确意向声明的部分原因。即使这些是尽力而为、不具约束力或被许多用户忽略，它仍然为我们提供了一个起点，让我们可以推断重大更改/主要版本升级是否“值得”，而不会带来这些相互冲突的动机。

结构。

去和Clojure两者都很好地处理了这个问题:在它们标准的软件包管理生态系统中,主版本升级的等价物应该是全新的软件包。这有一定的公平性:如果你愿意破坏软件包的向后兼容性,为什么我们要假装这是同一套 API?

重新打包和重命名所有内容似乎是一项合理的工作,可以期望提供商采取核选项并放弃向后兼容性。

最后,该过程存在人为错误。一般来说,SemVer 版本升级应该适用于语义变化和语法变化;改变 API 的行为和改变其结构同样重要。尽管开发工具来评估某个特定版本是否涉及一组公共 API 的语法变化是可行的,但判断是否存在有意义和有意的语义变化在计算上是不可行的。<sup>12</sup>实际上,即使是识别语法变化的潜在工具也是有限的。在几乎所有情况下,是否针对任何给定的更改升级主要版本、次要版本或修补版本都取决于 API 提供商的人为判断。如果您仅依赖少数专业维护的依赖项,那么您预期遇到这种形式的 SemVer 文书错误的可能性可能很低。<sup>13</sup>如果您的产品下有数千个依赖项的网络,那么您应该为仅由人为错误引起的一一定程度的混乱做好准备。

## 最低版本选择

2018 年,作为为 Go 编程语言构建包管理系统的系列文章的一部分,谷歌的 Russ Cox 描述了 SemVer 依赖管理的一个有趣变体:**最低版本选择 (MVS)**。在更新依赖网络中某个节点的版本时,可能需要将其依赖项更新为较新的版本以满足更新的 SemVer 要求。这随后会触发进一步的更改。在大多数约束满足/版本选择公式中,都会选择这些下游依赖项的最新版本:毕竟,您最终需要更新到这些新版本,对吗?

MVS 做出了相反的选择:当 liba 的规范要求  $\text{libbase} \geq 1.7$  时,我们会直接尝试 libbase 1.7,即使有 1.8 可用。这“会产生高保真构建,其中用户构建的依赖项尽可能接近

---

<sup>12</sup>在无处不在的单元测试的世界里,我们可以识别需要改变测试行为的变化,但仍然很难从算法上将“这是一个行为改变”与“这是一个对非预期/承诺的行为的错误修复”区分开来。

<sup>13</sup>因此,当从长远来看很重要时,请选择维护良好的依赖关系。

作者开发时所针对的版本。”<sup>14</sup>这一点揭示了一个至关重要的事实：当 liba 说它需要 libbase  $\geq 1.7$  时，这几乎肯定意味着 liba 的开发人员安装了 libbase 1.7。假设维护者在发布之前执行了哪怕是最基本的测试，<sup>15</sup>我们至少有传闻证据表明该版本的 liba 和 1.7 版本的 libbase 进行了互操作性测试。这不是 CI，也不是所有东西都经过了单元测试的证明，但总归是件好事。

如果没有从 100% 准确预测未来中得出的准确输入约束，最好尽可能地向前迈出最小的一步。就像通常将一小时的工作投入到项目中比一次性放弃一年的工作更安全一样，在依赖项更新中向前迈出较小的一步也更安全。MVS 只会将每个受影响的依赖项向前推进到所需的程度，并说：“好的，我已经向前走了足够远，可以得到你要求的东西（不会再远了）。你为什么不运行一些测试，看看事情是否顺利？”

MVS 的理念中固有的一点是，承认较新的版本在实践中可能会引入不兼容性，即使理论上版本号并非如此。无论是否使用 MVS，这都承认了 SemVer 的核心问题：将软件更改压缩为版本号会损失一些保真度。MVS 提供了一些额外的实际保真度，尝试生成最接近可能一起测试过的版本的选定版本。这可能足以推动更大的依赖网络正常运行。不幸的是，我们还没有找到一个好方法来实证验证这个想法。在不解决该方法的基本理论和激励问题的情况下，MVS 是否使 SemVer “足够好”尚无定论，但我们仍然相信它代表了当今使用的 SemVer 约束应用的明显改进。

## 那么，SemVer 有效吗？

SemVer 在有限的范围内运行良好。然而，认识到它实际上在说什么以及它不能说什么非常重要。SemVer 可以很好地工作，前提是：

- 您的依赖项提供商是准确且负责任的（以避免 SemVer 冲突中的人为错误）· 您的依赖项是细粒度的（以避免在更新依赖项中未

使用/不相关的 API 时出现错误的过度约束，以及与无法满足 SemVer 要求相关的风险）

---

<sup>14</sup> Russ Cox，“最小版本选择”，2018 年 2 月 21 日，<https://research.swtch.com/vgo-mvs>。

<sup>15</sup> 如果这个假设不成立，你真的应该停止依赖 liba。

- 所有 API 的使用都在预期范围内（以避免因假定兼容的更改而以令人惊讶的方式被破坏,无论是直接的还是在你依赖的代码中）

当您的依赖图中只有少数精心选择且维护良好的依赖项时,SemVer 可能是一个非常合适的解决方案。

然而,我们在 Google 的经验表明,你不太可能在规模上同时拥有这三个属性中的任何一个,并让它们随着时间的推移持续发挥作用。规模往往是 SemVer 弱点的体现。随着依赖网络的扩大,无论是每个依赖的大小还是依赖的数量(以及由于多个项目依赖于同一个外部依赖网络而产生的 monorepo 效应),SemVer 中的复合保真度损失将开始占主导地位。这些失败表现为误报(实际上不兼容的版本,理论上应该可以工作)和误报(SAT 解析器不允许的兼容版本以及由此产生的依赖地狱)。

## 使用 Innite Resources 进行依赖管理

在考虑依赖管理解决方案时,这是一个有用的思维实验:如果我们都能使用无限的计算资源,依赖管理会是什么样子?也就是说,如果我们不受资源限制,而只受到组织间可见性和薄弱协调的限制,我们能期望的最好结果是什么?正如我们目前所看到的,行业依赖 SemVer 有三个原因:

- 它只需要本地信息 (API 提供者不需要知道 par - 下游用户详情)
- 它不假设测试的可用性 (目前在行业中还不是普遍存在的,但在未来十年肯定会朝着这个方向发展)、运行测试的计算资源或监控测试结果的 CI 系统
- 这是现有的做法

对本地信息的“要求”实际上并不是必要的,特别是因为依赖网络往往仅在两种环境中形成:

- 在单个组织内
- 在 OSS 生态系统内,即使项目不一定合作,源代码也是可见的

无论是哪种情况,下游使用情况的重要信息都是可用的,即使这些信息目前还不容易被曝光或采取行动。也就是说,SemVer 的有效主导地位的一部分在于我们选择忽略理论上

我们可以使用它。如果我们可以访问更多的计算资源，并且依赖关系信息很容易浮出水面，那么社区可能会找到它的用途。

尽管 OSS 软件包可以有无数的闭源依赖项，但常见的情况是流行的 OSS 软件包在公开和私人领域都很流行。

依赖网络不会（不能）积极地混合公共和私有依赖关系：通常，有一个公共子集和一个单独的私有子图。<sup>16</sup>

接下来，我们必须记住 SemVer 的目的：“据我估计，这个更改将很容易（或不容易）被采用。”有没有更好的方式来传达这些信息？有的，以实践经验的形式证明这个更改很容易被采用。我们如何获得这样的经验？如果我们的大多数依赖项（或至少是代表性样本）都是公开可见的，我们会对每个提议的更改运行这些依赖项的测试。有了足够多的此类测试，我们至少有一个统计论据，证明这个更改在实际的海勒姆定律意义上是安全的。测试仍然通过，更改是好的——无论是影响 API、修复错误还是介于两者之间的任何事情；无需分类或估计。

那么，想象一下，OSS 生态系统进入了一个变化伴随着是否安全的证据的世界。如果我们将计算成本从等式中剔除，那么“这有多安全”的真相<sup>17</sup> 就来自于在下游依赖项中运行受影响的测试。

即使没有将正式的 CI 应用于整个 OSS 生态系统，我们当然也可以使用这样的依赖关系图和其他次要信号来进行更有针对性的提交前分析。优先考虑使用频繁的依赖项中的测试。优先考虑维护良好的依赖项中的测试。优先考虑曾经提供良好信号和高质量测试结果的依赖项中的测试。除了根据最有可能为我们提供有关实验变更质量的最多信息的项目对测试进行优先排序之外，我们还可以使用变更作者提供的信息来帮助估计风险并选择合适的测试策略。如果目标是“任何人所依赖的都不是重大变更”，那么运行“所有受影响的”测试在理论上是必要的。如果我们认为目标更符合“风险缓解”，统计论证将成为一种更有吸引力（且更具成本效益）的方法。

在第 12 章中，我们确定了四种类型的变更，从纯重构到现有功能的修改。给定一个基于 CI 的依赖项更新模型，我们可以开始将这些类型的变更映射到类似 SemVer 的模型上，变更的作者会评估风险并应用适当级别的测试。例如，仅修改内部 API 的纯重构变更

---

<sup>16</sup>因为公共 OSS 依赖网络通常不能依赖于一堆私有节点，所以图形尽管有固件。

<sup>17</sup>或者与之非常接近的东西。

可能被认为是低风险的，并且证明只在我们自己的项目中运行测试是合理的，也许还会对重要的直接依赖项进行抽样。另一方面，删除已弃用的接口或更改可观察行为的更改可能需要我们尽可能多地进行测试。

要应用这种模式，我们需要对 OSS 生态系统做出哪些改变？不幸的是，我们需要做很多改变：

- 所有依赖项都必须提供单元测试。尽管我们正在不可避免地走向一个单元测试被广泛接受和无处不在的世界，但我们还没有到达那个地步。
- 了解 OSS 生态系统中大多数的依赖关系网络。目前尚不清楚是否有任何机制可用于在该网络上执行图形算法。信息是公开且可用的，但实际上并未被索引或使用。许多包管理系统/依赖关系管理生态系统允许您查看项目的依赖关系，但无法查看反向边，即依赖项。
- 执行 CI 所需的计算资源仍然非常有限。大多数开发人员无权构建和测试计算集群。
- 依赖关系通常以固定方式表达。作为 libbase 的维护者，如果这些依赖关系明确依赖于 libbase 的特定固定版本，我们就无法通过 liba 和 libb 的测试来实验性地运行更改。
- 我们可能希望在 CI 计算中明确包含历史和声誉。如果提议的变更破坏了一个具有长期测试历史的项目，那么它给我们提供的证据形式与最近才添加的、具有不相关测试历史的项目中的破坏不同

原因。

这其中固有的一个规模问题：在提交更改之前，您要针对网络中每个依赖项的哪些版本进行测试？如果我们针对所有历史版本的完整组合进行测试，我们将耗费真正惊人的计算资源量，即使按照 Google 的标准也是如此。此版本选择策略最明显的简化似乎是“测试当前稳定版本”（毕竟，基于主干的开发才是目标）。因此，在无限资源的情况下，依赖项管理模型实际上就是 Live at Head 模型。悬而未决的问题是，该模型是否能够在更实际的资源可用性下有效应用，以及 API 提供商是否愿意承担更大的责任来测试其更改的实际安全性。认识到我们现有的低成本设施对我们正在寻找的难以计算的事实的过度简化仍然是一项有用的练习。

导出依赖项到目前为止,我们只

讨论了依赖项;即依赖于其他人编写的软件。我们还应该考虑如何构建可用作依赖项的软件。这不仅仅是打包软件并将其上传到存储库的机制:我们需要考虑提供软件的好处、成本和风险,无论是对我们自己还是对我们潜在的依赖者而言。

像“开源库”这样无害且充满善意的行为可能会给组织带来两种主要损失。首先,如果实施不当或维护不当,最终可能会损害组织的声誉。正如 Apache 社区所说,我们应该优先考虑“社区高于代码”。如果您提供出色的代码但却是社区中表现不佳的成员,那么这仍然会对您的组织和更广泛的社区造成伤害。

其次,如果不能保持同步,一次善意的发布可能会成为工程效率的负担。假以时日,所有分叉都会变得昂贵。

示例:开源 gflags 对于声誉损失,请

考虑类似 Google 在 2006 年开源我们的 C++ 命令行标志库的经历。回馈开源社区肯定是一种纯粹的善举,不会给我们带来麻烦,对吧?

很遗憾,不行。有多种原因导致这一善举损害了我们的声誉,甚至可能损害了 OSS 社区:

- 当时,我们没有能力执行大规模重构,因此内部使用该库的所有内容都必须保持完全相同 - 我们无法将代码移动到代码库中的新位置。
- 我们将存储库分为“内部开发的代码”(如果需要分叉,可以自由复制,只要正确重命名)和“可能存在法律/许可问题的代码”(可能有更细致入微的使用要求)。
- 如果 OSS 项目接受来自外部开发人员的代码,这通常是一个法律问题 项目发起人并不拥有该贡献,他们只拥有权利  
對它來說。

因此,gflags 项目注定要么是“翻墙”版本,要么是断开连接的分支。为该项目贡献的补丁无法重新合并到 Google 内部的原始源代码中,我们无法将该项目移到 monorepo 中,因为我们尚未掌握这种形式的重构,我们也无法让所有内容在内部依赖于 OSS 版本。

此外,与大多数组织一样,我们的优先事项也随着时间的推移而发生变化。

在该标志库首次发布时,我们感兴趣的是

我们传统领域（Web 应用程序、搜索）之外的产品，包括 Google Earth 之类具有更传统的分发机制的产品：为各种平台预编译的二进制文件。在 2000 年代后期，我们的 monorepo 中的库（尤其是像 flags 这样低级的库）用于各种平台是不寻常的，但并非闻所未闻。随着时间的推移和 Google 的发展，我们的关注点缩小到极少使用内部配置的工具链以外的任何其他东西构建库，然后部署到我们的生产环境中。正确支持像 flags 这样的 OSS 项目所需的“可移植性”问题几乎不可能维持：我们的内部工具根本不支持这些平台，而我们的普通开发人员不必与外部工具交互。

为了保持可移植性，这是一场持久战。

随着原作者和 OSS 支持者转投新公司或新团队，我们最终发现，内部没有人真正支持我们的 OSS 标志项目。没有人能够将这种支持与任何特定团队的优先事项联系起来。鉴于这不是特定团队的工作，也没有人能说明它为什么重要，我们基本上让该项目在外部腐烂也就不足为奇了。<sup>18</sup>随着时间的推移，内部和外部版本逐渐分化，最终一些外部开发人员采用了外部版本并对其进行了分叉，赋予了它一些适当的

注意力。

除了最初的“哦，看，谷歌为开源世界做出了贡献”之外，这一切并没有给我们带来任何好处，但考虑到我们工程组织的优先事项，这一切的每一点都是有意义的。我们这些与它关系密切的人已经学会了“不要在没有长期支持计划（和授权）的情况下发布任何东西。”整个谷歌工程部门是否已经学会了这一点还有待观察。这是一个庞大的组织。

---

<sup>18</sup>这并不是说它是正确的或明智的，只是作为一个组织，我们让一些事情漏网了。

除了含糊其辞的“我们看起来很糟糕”之外,这个故事中还有一些部分说明了我们可能受到因发布不当/维护不善的外部依赖项而导致的技术问题的影响。尽管 flags 库是共享的但被忽略了,但仍有一些 Google 支持的开源项目,或者需要在我们的 monorepo 生态系统之外共享的项目。不出所料,这些其他项目的作者能够识别<sup>19</sup>该库的内部和外部分支之间的通用 API 子集。由于该公共子集在两个版本之间长期保持相当稳定,因此它悄然成为 2008 年至 2017 年期间具有不同寻常的可移植性要求的少数团队的“做法”。他们的代码可以在内部和外部生态系统中构建,根据

环境。

然后,出于不相关的原因,C++ 库团队开始调整内部标志实现中可观察但未记录的部分。此时,所有依赖不受支持的外部分支的稳定性和等效性的人都开始尖叫,说他们的构建和发布突然被破坏了。一个价值数千个总 CPU 的 Google 集群的优化机会会被大大推迟了,这并不是因为很难更新 2.5 亿行代码所依赖的 API,而是因为极少数项目依赖于未承诺和意外的东西。再一次,海勒姆定律影响了软件更改,在这种情况下,即使是由不同组织维护的分叉 API 也是如此。

#### 案例研究:AppEngine一个更

严重的例子是,我们面临更大的意外技术依赖风险,那就是发布 Google 的 AppEngine 服务。该服务允许用户使用几种流行的编程语言之一在现有框架之上编写应用程序。只要应用程序是使用适当的存储/状态管理模型编写的,AppEngine 服务就允许这些应用程序扩展到巨大的使用级别:后端存储和前端管理由 Google 的生产基础设施按需管理和克隆。

最初,AppEngine 对 Python 的支持是使用旧版 Python 解释器运行的 32 位版本。AppEngine 系统本身(当然)在我们的 monorepo 中实现,并使用我们的其他常用工具(使用 Python 和 C++ 构建,以提供后端支持)构建。2014 年,我们开始对 Python 运行时以及我们的 C++ 编译器和标准库安装进行重大更新,结果是我们有效地将“使用当前 C++ 编译器构建的代码”与“使用更新的 Python 版本的代码”绑定在一起。升级其中一个依赖项的项目本质上会同时升级另一个依赖项。对于大多数

<sup>19</sup>通常通过反复试验。

对于一些项目,这不是什么问题。对于一些项目,由于极端情况和海勒姆定律,我们的语言平台专家最终做了一些调查和调试,以解除过渡的阻碍。在海勒姆定律遇到商业实际问题的一个可怕例子中,AppEngine 发现它的许多用户,也就是我们的付费客户,无法(或不会)更新:他们要么不想切换到较新的 Python 版本,要么无法承担从 32 位 Python 迁移到 64 位 Python 所涉及的资源消耗变化。由于有些客户为 AppEngine 服务支付了巨额费用,因此 AppEngine 能够提出强有力的商业理由,即必须推迟强制切换到新语言和编译器版本。这本质上意味着来自 AppEngine 的依赖关系传递闭包中的每一段 C++ 代码都必须与较旧的编译器和标准库版本兼容:对该基础架构进行的任何错误修复或性能优化都必须跨版本兼容。这种情况持续了近三年。

有了足够多的用户,系统的任何“可观察”内容都会被某些人依赖。在 Google,我们将所有内部用户限制在技术栈边界内,并通过 monorepo 和代码索引系统确保他们能够看到自己的使用情况,因此,我们可以更轻松地确保进行有用的更改。当我们从源代码控制转向依赖管理,失去了对代码使用方式的可见性,或者受到外部团体(尤其是付钱给你的团体)的优先考虑时,做出纯粹的工程权衡就会变得更加困难。发布任何类型的 API 都有可能让你面临优先考虑因素和外部人员无法预料的限制。这并不是说你不应该发布 API;它只是提醒你:维护 API 的外部用户比维护内部用户要花费更多。

与外界共享代码,无论是作为开源版本还是作为闭源库版本,都不是简单的慈善行为(在 OSS 情况下)或商业机会(在闭源情况下)。您无法监控的不同组织中具有不同优先级的依赖用户最终将对该代码施加某种形式的海勒姆定律惯性。特别是如果您的工作时间跨度很长,就不可能准确预测可能变得有价值的一组必要或有用的更改。在评估是否发布某些内容时,请注意长期风险:随着时间的推移,外部共享的依赖项的修改成本通常要高得多。

## 结论

依赖管理本质上具有挑战性。我们正在寻找管理复杂 API 表面和依赖网络的解决方案，其中这些依赖关系的维护者通常很少或根本不需要协调。

管理依赖关系网络的事实标准是语义版本控制（SemVer），它提供了采用任何特定更改时可察觉到的风险的有损摘要。SemVer 假设我们可以在不了解相关 API 的使用方式的情况下先验地预测更改的严重程度：海勒姆定律告诉我们事实并非如此。然而，SemVer 在小规模下运行良好，当我们采用 MVS 方法时，效果会更好。随着依赖关系网络规模的扩大，海勒姆定律问题和 SemVer 中的保真度损失使得管理新版本的选择变得越来越困难。

然而，我们有可能走向一个世界，在这个世界中，维护者提供的兼容性估计（SemVer 版本号）被放弃，取而代之的是经验驱动的证据：运行受影响的下游软件包的测试。如果 API 提供商承担更大的责任来针对其用户进行测试，并明确宣传预期的更改类型，我们就有可能在更大规模上实现更高保真度的依赖关系网络。

## TL;DR

- 优先考虑源代码控制问题而不是依赖管理问题：如果你能从组织获得更多代码，从而实现更好的透明度和协调性，那么这些都是重要的简化。
- 对于软件工程项目来说，添加依赖关系并非免费，建立“持续”信任

关系的复杂性也具有挑战性。将依赖关系导入组织需要谨慎，并了解持续支持成本。  
· 依赖关系是一种合同：双方都有所取舍，供应商和消费者在该合同中都拥有一些权利和责任。供应商应该清楚他们长期承诺的内容。

- SemVer 是“人类认为这种变化有多危险？”的有损压缩简写估计。SemVer 和包管理器中的 SAT 解析器会采用这些估计并将其升级为绝对值。这可能会导致过度约束（依赖地狱）或约束不足（应该一起工作但不能一起工作的版本）。
- 相比之下，测试和 CI 提供了一组新版本是否协同工作的实际证据。

- SemVer/包管理中的最小版本更新策略保真度更高。这仍然依赖于人类能够准确评估增量版本风险,但明显提高了 API 提供者和消费者之间的链接经过专家测试的可能性。
- 单元测试、CI 和 (廉价)计算资源有可能改变我们对依赖管理的理解和方法。这一阶段的变化需要行业从根本上改变对依赖管理问题的看法,以及供应商和消费者的责任。
- 提供依赖关系并非免费：“扔掉它然后忘掉它”可能会损害您的声誉并成为兼容性的挑战。以稳定性支持它会限制您的选择并不利于内部使用。在没有稳定性的情况下提供支持可能会损害您的信誉或使您面临重要外部团体依赖某些东西的风险,这些团体会根据 Hyrum 定律依赖某些东西并搞砸您的“不稳定”计划。



## 第22章

# 大规模变革

作者:海伦·莱特  
由 Lisa Carey 编辑

想一想你自己的代码库。在一次同时提交中,你可以可靠地更新多少个文件?哪些因素限制了这个数字?你曾经尝试过提交如此大的更改吗?在紧急情况下,你能在合理的时间内完成吗?你最大的提交大小与代码库的实际大小相比如何?你会如何测试这样的更改?

在提交更改之前需要多少人审核更改?如果确实提交了更改,您是否能够回滚该更改?这些问题的答案可能会让您感到惊讶(您认为的答案以及它们对您的组织的实际结果)。

在 Google,我们早就放弃了在这种类型的大型原子变更中对整个代码库进行全面变更的想法。我们的观察是,随着代码库和在其中工作的工程师数量的增长,最大的原子变更可能反而会减少。运行所有受影响的提交前检查和测试变得困难,更不用说确保变更中的每个文件在提交前都是最新的了。由于对我们的代码库进行全面变更变得越来越困难,考虑到我们普遍希望能够不断改进底层基础设施,我们不得不开发新的方式来推理大规模变更以及如何实施它们。

在本章中,我们将讨论社交和技术方面的技术,这些技术使我们能够保持庞大的 Google 代码库的灵活性并响应底层基础架构的变化。我们还将提供一些实际示例,说明我们如何以及在何处使用这些方法。尽管您的代码库可能看起来不像 Google 的,但了解这些原则并在本地进行调整将有助于您的开发组织扩展,同时仍然能够在整个代码库中进行广泛的更改。

# 什么是大规模变革？

在进一步讨论之前,我们应该深入研究什么才算是大规模变更 (LSC)。根据我们的经验,LSC 是任何一组逻辑上相关但实际上无法作为单个原子单元提交的变更。这可能是因为它涉及的文件太多,以至于底层工具无法一次提交所有文件,或者可能是因为变更太大,以至于总是会发生合并冲突。在许多情况下,LSC 由您的存储库拓扑决定<sup>1</sup>如果您的组织使用一组分布式或联合存储库<sup>2</sup>,那么在它们之间进行原子变更在技术上甚至可能都不可行。<sup>3</sup>我们将在本章后面更详细地讨论原子变更的潜在障碍。

Google 的 LSC 几乎总是使用自动化工具生成的。制作 LSC 的原因各不相同,但更改本身通常分为几个基本类别:

- 使用代码库范围的分析工具清理常见的反模式 · 替换已弃用的库功能的使用 · 启用低级基础设施改进,例如编译器升级 · 将用户从旧系统转移到新系统<sup>3</sup>

在特定组织中,从事这些特定任务的工程师数量可能很少,但对于他们的客户来说,了解 LSC 工具和流程是有用的。就其本质而言,LSC 将影响大量客户,而 LSC 工具很容易缩减到仅进行几十项相关更改的团队。

特定 LSC 背后可能有更广泛的动机。例如,新的语言标准可能会引入更高效的习语来完成给定的任务,内部库接口可能会发生变化,或者新的编译器版本可能需要修复现有问题,而这些问题在新版本中会被标记为错误。

实际上,Google 的大多数 LSC 对功能的影响几乎为零:它们往往是为提高清晰度、优化或未来兼容性而进行的广泛文本更新。但 LSC 理论上并不局限于这种行为保留/重构类的变更。

在所有这些情况下,在像谷歌这样规模的代码库上,基础设施团队可能经常需要更改数十万个对旧代码的单独引用

<sup>1</sup>有关原因的一些想法,请参见第 16 章。

<sup>2</sup>在这个联合的世界中,可以说“我们将尽快提交到每个存储库,以将构建中断的持续时间保持在较短的范围内!”但随着联合存储库数量的增长,这种方法实际上无法扩展。

<sup>3</sup>有关此做法的进一步讨论,请参阅第 15 章。

模式或符号。到目前为止，在最大的案例中，我们已经触及了数百万个引用，我们预计该流程将继续很好地扩展。一般来说，我们发现尽早投资工具以启用 LSC 对许多从事基础设施工作的团队来说是有利的。我们还发现，高效的工具还可以帮助工程师执行较小的更改。使更改数千个文件变得高效的相同工具也可以相当好地缩减到数十个文件。

## 谁与 LSC 打交道？

正如刚才所指出的，构建和管理我们系统的基础设施团队负责执行 LSC 的大部分工作，但工具和资源在整个公司都是可用的。如果你跳过了第 1 章，你可能会想知道为什么基础设施团队要负责这项工作。为什么我们不能只引入一个新的类、函数或系统，并规定所有使用旧系统的人都转移到更新后的模拟系统？虽然这在实践中似乎更容易，但由于几个原因，事实证明它的扩展性不是很好。

首先，构建和管理底层系统的基础设施团队也具备修复数十万个引用所需的领域知识。使用基础设施的团队不太可能具备处理这些迁移的背景知识，而且期望他们每个人都重新学习基础设施团队已经拥有的专业知识，从整体上来说效率很低。集中化还可以在遇到错误时更快地恢复，因为错误通常分为一小组类别，并且运行迁移的团队可以有一本正式或非正式的剧本来自处理它们。

考虑一下，完成一系列您不理解的半机械更改中的第一个更改需要花费多少时间。您可能会花一些时间阅读有关更改的动机和性质的信息，找到一个简单的示例，尝试遵循提供的建议，然后尝试将其应用于您的本地代码。对组织中的每个团队重复此操作会大大增加总体执行成本。通过仅让少数几个集中团队负责 LSC，Google 既将这些成本内部化，又通过使更改更有效地进行来降低这些成本。

其次，没有人喜欢没有资金支持的强制要求。<sup>4</sup> 尽管新系统可能明显优于它所取代的系统，但这些好处往往分散在整个组织中，因此不太可能对各个团队主动更新系统产生足够大的影响。如果新系统足够重要，值得迁移到，

---

<sup>4</sup> “无资金支持的任务”是指“外部实体强加的额外要求，没有平衡的补偿”。有点像 CEO 说每个人都必须在“正式的星期五”穿晚礼服，但却不给你相应的加薪来支付你的正式着装费用。

迁移成本将由组织中的某个部门承担。集中迁移并核算其成本几乎总是比依靠各个团队进行有机迁移更快、更便宜。

此外,拥有需要 LSC 的系统的团队有助于协调激励措施,确保变更得以完成。根据我们的经验,有机迁移不太可能完全成功,部分原因是工程师在编写新代码时倾向于使用现有代码作为示例。拥有一支对移除旧系统有既得利益的团队来负责迁移工作有助于确保迁移工作真正完成。

尽管为运行此类迁移而配备资金和人员的团队似乎是一项额外成本,但实际上它只是将无资金支持的任务所产生的外部性内部化,并具有规模经济的额外好处。

#### 案例研究 :填补漏洞尽管 Google 的

LSC 系统用于高优先级迁移,但我们还发现,只要有它们可用,就会为我们的代码库中的各种小修复提供机会,如果没有它们,这是不可能的。

就像交通基础设施任务包括修建新道路和修复旧道路一样,谷歌的基础设施团队花费大量时间修复现有代码,以及开发新系统并转移用户。

例如,在我们早期的历史中,出现了一个模板库来补充 C++ 标准模板库。这个库被恰当地命名为 Google 模板库,由几个头文件的实现组成。由于时间久远而无法知晓的原因,其中一个头文件被命名为 stl\_util.h,另一个被命名为 map-util.h (请注意文件名中的不同分隔符)。除了让一致性纯粹主义者抓狂之外,这种差异还导致生产力下降,工程师必须记住哪个文件使用了哪个分隔符,并且只有在经过可能很长的编译周期后才发现他们错了。

尽管修复这个单字符更改似乎毫无意义,尤其是在像 Google 这样规模的代码库中,但我们成熟的 LSC 工具和流程使我们能够仅用几周的后台任务努力就完成此操作。库作者可以找到并大量应用此更改,而无需打扰这些文件的最终用户,并且我们能够量化减少由此特定问题导致的构建失败次数。由此带来的生产力 (和幸福感) 的提高远远超过了进行更改所花费的时间。

随着我们在整个代码库中进行更改的能力得到改善,更改的多样性也得到了扩展,我们可以做出一些工程决策,因为我们知道它们在未来并不是一成不变的。有时,填补一些漏洞是值得的。

## 原子变化的障碍

在讨论 Google 实际用于实现 LSC 的过程之前,我们应该先讨论一下为什么许多类型的更改无法以原子方式提交。在理想情况下,所有逻辑更改都可以打包成一个原子提交,可以独立于其他更改进行测试、审查和提交。不幸的是,随着存储库 (以及在其中工作的工程师数量) 的增长,这种理想变得越来越不可行。当使用一组分布式或联合存储库时,即使在小规模下,它也完全不可行。

## 技术限制

首先,大多数版本控制系统 (VCS) 的操作都与更改的大小成线性关系。您的系统可能能够很好地处理小型提交 (例如,大约数十个文件),但可能没有足够的内存或处理能力来一次原子地提交数千个文件。在集中式 VCS 中,提交可能会阻止其他写入者 (在较旧的系统中,是读取者) 在处理时使用系统,这意味着大型提交会使系统的其他用户停滞不前。

简而言之,原子地进行大规模变更可能不仅仅是“困难”或“不明智”:在给定的基础设施下,这可能根本不可能。将大型变更拆分为较小的独立部分可以绕过这些限制,尽管这会使变更的执行更加复杂。<sup>5</sup>

### 合并冲突随着变更规

模的扩大,发生合并冲突的可能性也会增加。我们知道,如果中央存储库中存在文件的较新版本,则每个版本控制系统都需要更新和合并,可能需要手动解决。随着变更中的文件数量增加,遇到合并冲突的概率也会增加,并且会因存储库中工作的工程师数量而加剧。

如果您的公司规模较小,那么您可以在周末无人开发时偷偷修改存储库中的所有文件。或者,您可能有一个非正式的系统,通过在开发团队中传递虚拟 (甚至是物理!) 令牌来获取全局存储库锁。在像 Google 这样的大型跨国公司,这些方法根本不可行:总有人在对存储库进行更改。

---

<sup>5</sup>请参阅<https://ieeexplore.ieee.org/abstract/document/8443579>。

如果变更中的文件很少,则合并冲突的可能性会降低,因此更有可能顺利提交。此属性也适用于以下领域。

不要闹鬼的墓地运行 Google 生产服务的

SRE 有一句口头禅：“不要闹鬼的墓地”。从这个意义上讲,闹鬼的墓地是指那些古老、晦涩或复杂到无人敢进入的系统。闹鬼的墓地通常是业务关键系统,它们被冻结在时间中,因为任何试图改变它们的尝试都可能导致系统以难以理解的方式失败,从而给企业造成真正的损失。它们构成了真正的生存风险,并且会消耗大量资源。

然而,闹鬼墓地不仅仅存在于生产系统中,它们也可以在代码库中找到。许多组织都有一些老旧且无人维护的软件,这些软件是由早已离开团队的人编写的,并且位于一些重要的创收功能的关键路径上。这些系统也处于停滞状态,层层官僚机构建立起来,以防止可能导致不稳定的变化。

没有人想成为翻转错误位的网络支持工程师!!!

代码库的这些部分对于 LSC 流程来说是令人厌恶的,因为它们会阻碍大规模迁移的完成、它们所依赖的其他系统的退役,或它们使用的编译器或库的升级。从 LSC 的角度来看,闹鬼的墓地会阻碍各种有意义的进展。

在 Google,我们发现与此相反的方法是进行良好而传统的测试。当软件经过彻底测试后,我们可以对它进行任意更改,并且有信心知道这些更改是否会带来破坏,无论系统的使用年限或复杂程度如何。编写这些测试需要付出很多努力,但它允许像 Google 这样的代码库在很长一段时间内不断发展,让闹鬼的软件墓地这个概念成为它自己的墓地。

异构LSC 只有在大

部分工作可以由计算机而非人类完成时才能真正发挥作用。尽管人类能够很好地处理模糊性,但计算机依靠一致的环境将适当的代码转换应用到正确的位置。如果您的组织有许多不同的 VCS、持续集成 (CI) 系统、特定于项目的工具或格式指南,则很难对整个代码库进行彻底更改。简化环境以增加一致性将有助于需要在其中移动的人类和进行自动转换的机器人。

例如,Google 的许多项目都配置了提交前测试,在代码库发生更改之前运行。这些检查可能非常复杂,从根据白名单检查新的依赖项,到运行测试,再到确保更改有相关的错误。其中许多检查对于编写新功能的团队来说很重要,但对于 LSC 来说,它们只会增加额外的无关复杂性。

我们决定接受其中的一些复杂性,例如运行提交前测试,将其作为整个代码库的标准。对于其他不一致之处,我们建议团队在 LSC 的部分内容触及他们的项目代码时省略他们的特殊检查。大多数团队都乐于提供帮助,因为这些更改会给他们的项目带来好处。



**第 8 章中提到的一致性给人类带来的许多好处也适用于自动化工具。**

## 测试每一

个变更都应该进行测试(稍后我们将详细讨论这一过程),但变更越大,实际进行适当测试就越困难。Google 的 CI 系统不仅会运行受变更直接影响的测试,还会运行任何依赖于变更文件的测试。<sup>6</sup>这意味着变更会得到广泛覆盖,但我们也观察到,依赖关系图中的测试距离受影响的文件越远,由变更本身导致故障的可能性就越小。

小型、独立的变更更容易验证,因为每个变更都会影响较小的测试集,而且测试失败也更容易诊断和修复。在 25 个文件的变更中找出测试失败的根本原因非常简单;而在 10,000 个文件的变更中找到 1 个则如同大海捞针。

这个决定的权衡是,较小的变化将导致相同的测试运行多次,特别是依赖于大部分代码库的测试。

由于工程师花在追踪测试失败上的时间比运行这些额外测试所需的计算时间要昂贵得多,因此我们做出了明智的决定,这是我们愿意做出的权衡。同样的权衡可能并不适用于所有组织,但值得研究适合您的组织的最佳平衡点。

---

<sup>6</sup>这听起来可能有点矫枉过正,事实也确实如此。我们正在积极研究如何为给定的变更确定“正确”的测试集,平衡运行测试所需的计算时间成本和做出错误选择的人力成本。

## 案例研究:测试 LSC

亚当·本德

如今,项目中有两位数百分比(10%到20%)的变更是由LSC造成的,这很常见,这意味着项目中的大量代码是由全职工作与项目无关的人更改的。如果没有良好的测试,这样的工作就不可能完成,而Google的代码库也会因自身重量而迅速萎缩。LSC使我们能够系统地将整个代码库迁移到较新的API、弃用较旧的API、更改语言版本,并消除流行但危险的做法。

即使是简单的一行签名更改,如果在数百种不同产品和服务的一千个不同位置进行更改,也会变得很复杂。<sup>7</sup>编写更改后,您需要协调数十个团队的代码审查。最后,在审查获得批准后,您需要运行尽可能多的测试,以确保更改是安全的。<sup>8</sup>我们说“尽可能多的”,因为规模较大的LSC可能会触发Google重新运行每个测试,而这可能需要一段时间。事实上,许多LSC必须计划时间来捕捉代码倒退的下游客户,而LSC正在完成整个过程。

测试LSC可能是一个缓慢而令人沮丧的过程。当更改足够大时,您的本地环境几乎肯定会永远与头脑不同步,因为代码库就像沙子一样在您的工作中移动。在这种情况下,您很容易发现自己正在运行并重新运行测试,只是为了确保您的更改继续有效。当项目的测试不稳定或缺少单元测试覆盖时,它可能需要大量的手动干预并减慢整个过程。为了帮助加快速度,我们使用一种称为TAP(测试自动化平台)训练的策略。

### 搭乘TAP列车

LSC的核心见解是它们很少相互影响,大多数受影响的测试都会通过大多数LSC。因此,我们可以一次测试多个更改,并减少执行的测试总数。训练模型已被证明对测试LSC非常有效。

TAP列车利用了两个优势:

- LSC倾向于纯粹的重构,因此范围非常狭窄,保留局部语义。

---

<sup>7</sup>有史以来执行的最大规模的LSC系列在三天内从存储库中删除了超过10亿行代码。这主要是删除已迁移到新位置的存储库过时部分;但是,您有多大信心删除10亿行代码?

<sup>8</sup> LSC通常由一些工具支持,使查找、制作和审查更改变得相对简单向前。

- 个别变更通常比较简单且经过严格审查,因此是正确的经常如此。

该列车模型的另一个优点是它可以同时用于多趟换乘,而不需要每次单独换乘。9列车有五个步骤,每三个小时启动一次:

1. 针对火车上的每一次换乘,随机选择 1,000 个测试样本。
2. 收集所有通过 1,000 次测试的变更,并从所有变更中创建一个超级变更:“火车”。
3. 运行所有直接受该组更改影响的测试的联合。如果 LSC 足够大 (或级别足够低) ,这可能意味着运行 Google 存储库中的每个测试。此过程可能需要六个多小时才能完成。
4. 对于每个失败的非不稳定测试,针对进入训练的每个更改单独重新运行它,以确定哪些更改导致其失败。
5. TAP 会针对每项变更生成一份报告。报告会描述所有通过和未通过的目标,并可用作 LSC 可以安全提交的证据。

## 代码审查

最后,正如我们在第 9 章中提到的,所有更改都需要在提交前进行审查,此政策甚至适用于 LSC。审查大型提交可能很繁琐、繁重,甚至容易出错,特别是如果更改是手动生成的 (我们希望避免此过程,我们将很快讨论)。稍后,我们将了解工具如何在这方面提供帮助,但对于某些类型的更改,我们仍然希望人工明确验证它们是否正确。将 LSC 分成单独的分片可以使这变得容易得多。

### 案例研究 :scoped\_ptr 到 std::unique\_ptr 从最早期开始,Google

的 C++ 代码库就有一个自毁智能指针,用于包装堆分配的 C++ 对象并确保在智能指针超出范围时销毁它们。这种类型被称为 scoped\_ptr ,在 Google 的代码库中被广泛使用,以确保对象生命周期得到适当管理。它并不完美,但考虑到首次引入该类型时当时的 C++ 标准 (C++98) 的局限性,它使程序更安全。

---

<sup>9</sup>可以要求 TAP 进行单次变更 “独立”运行,但这些运行非常昂贵,并且仅在非高峰时段。

在 C++11 中,该语言引入了一种新类型: std::unique\_ptr。它实现了与scoped\_ptr相同的功能,但还可以防止该语言现在可以检测到的其他类型的错误。std ::unique\_ptr确实比scoped\_ptr更好,但谷歌的代码库中有超过 500,000 个对scoped\_ptr的引用分散在数百万个源文件中。迁移到更现代的类型需要谷歌内部迄今为止最大的 LSC 尝试。

在接下来的几个月中,几名工程师同时着手解决这个问题。

使用 Google 的大规模迁移基础架构,我们能够将对scoped\_ptr的引用更改为对std::unique\_ptr的引用,并慢慢调整scoped\_ptr以使其行为更接近std::unique\_ptr。在迁移过程的高峰期,我们不断生成、测试和提交超过 700 个独立更改,每天涉及超过 15,000 个文件。如今,我们有时可以管理 10 倍的吞吐量,改进了我们的实践并改进了我们的工具。

与几乎所有 LSC 一样,这个 LSC 需要追踪各种细微的行为依赖关系(海勒姆定律的另一种表现形式),与其他工程师对抗竞争条件,以及在生成的代码中使用我们的自动化工具无法检测到的用途。当测试基础设施发现这些问题时,我们继续手动处理这些问题。

scoped\_ptr还被用作一些广泛使用的 API 的参数类型,这使得小的独立更改变得困难。我们考虑编写一个调用图分析系统,该系统可以在一次提交中传递性地更改 API 及其调用者,但我们担心由此产生的更改本身太大而无法原子提交。

最后,我们终于能够移除scoped\_ptr,方法是先将其设为std::unique\_ptr的类型别名,然后在旧别名和新别名之间执行文本替换,最后再移除旧的scoped\_ptr别名。如今,Google 的代码库受益于使用与 C++ 生态系统其他部分相同的标准类型,这完全得益于我们为 LSC 提供的技术和工具。

## LSC 基础设施

谷歌已经投资了大量的基础设施来实现 LSC。

该基础设施包括变更创建、变更管理、变更审查和测试工具。然而,对 LSC 最重要的支持可能是围绕大规模变更的文化规范的演变以及对它们的监督。虽然您的组织的技术和社会工具集可能有所不同,但一般原则应该是相同的。

## 政策与文化

正如我们在第 16 章中所描述的，Google 将其大部分源代码存储在一个单一的整体存储库 (monorepo) 中，每个工程师几乎都可以看到所有这些代码。这种高度开放意味着任何工程师都可以编辑任何文件，并将这些编辑发送给可以批准的人进行审查。但是，每个编辑都有成本，既要生成成本，也要审查成本。<sup>10</sup>从历史上看，这些成本在某种程度上是对称的，这限制了单个工程师或团队可以生成的更改范围。随着 Google 的 LSC 工具的改进，

以非常低的成本生成大量更改变得更加容易，单个工程师也同样容易给整个公司的大量审阅者带来负担。尽管我们希望鼓励对我们的代码库进行广泛的改进，但我们希望确保在改进背后有一些监督和深思熟虑，而不是不加区别地进行调整。<sup>11</sup>最终结果是，对于希望在 Google 范围内创建 LSC 的团队和个人来说，这是一个轻量级的审批流程。这个过程由一组经验丰富的工程师监督，他们熟悉各种语言的细微差别，并邀请特定更改领域的专家。这个过程的目标不是禁止 LSC，而是帮助更改作者进行尽可能最好的更改，从而最大限度地利用 Google 的技术和人力资本。有时，这个小组可能会建议清理不值得：例如，清理常见的拼写错误，而没有任何方法

可以防止再次发生。

与这些政策相关的是围绕 LSC 的文化规范的转变。虽然代码所有者对自己的软件有责任感很重要，但他们也需要了解 LSC 是 Google 扩展软件工程实践的重要组成部分。正如产品团队最熟悉自己的软件一样，图书馆基础设施团队也了解基础设施的细微差别，让产品团队相信领域专业知识是迈向社会接受 LSC 的重要一步。由于这种文化转变，本地产品团队逐渐信任 LSC 作者会做出与这些作者领域相关的更改。

有时，本地所有者会质疑作为更广泛的 LSC 的一部分进行的特定提交的目的，变更作者会像回应其他评论一样回应这些评论。从社会角度来看，代码所有者了解其软件发生的变化很重要，但他们也意识到，他们对更广泛的 LSC 没有否决权。随着时间的推移，我们发现，一份好的常见问题解答和

---

<sup>10</sup>计算和存储方面的技术成本显而易见，但审查过程中的人力成本这一变化远远超过技术上的变化。

<sup>11</sup>例如，我们不希望由此产生的工具被用作争夺正确拼写的机制评论中为“灰色”。

良好的历史记录改进使得 Google 内部对 LSC 产生了广泛的认可。

代码库洞察为了进行 LSC,我们

发现能够对代码库进行大规模分析非常有价值,既可以使用传统工具在文本层面进行分析,也可以在语义层面进行分析。例如,Google 使用语义索引工具[Kythe](#)提供了代码库各部分之间的完整链接图,使我们能够提出诸如“此函数的调用者在哪里?”或“哪些类源自这个类?”之类的问题。

Kythe 和类似工具还提供对其数据的编程访问,以便将它们纳入重构工具中。(有关更多示例,请参阅第17章和第20 章。)

我们还使用基于编译器的索引对我们的代码库进行基于抽象语法树的分析和转换。[ClangMR](#) 等[工具](#)[JavacFlume](#) (或[Refaster](#))可以以高度可并行的方式执行转换的程序依赖于这些见解作为其功能的一部分。对于较小的更改,作者可以使用专门的自定义工具、[perl](#)或[sed](#)、正则表达式匹配,甚至是简单的 shell 脚本。

无论您的组织使用什么工具来创建变更,重要的是其人力投入应与代码库呈亚线性增长;换句话说,无论存储库的大小如何,生成所有所需变更的集合所需的人力时间应大致相同。变更创建工具还应适用于整个代码库,以便作者可以确保他们的变更涵盖他们试图修复的所有情况。

与本书中的其他领域一样,早期对工具的投资通常会在短期到中期内获得回报。根据经验法则,我们长期以来认为,如果一项变更需要 500 次以上的编辑,那么工程师学习和执行我们的变更生成工具通常比手动执行该编辑更有效率。对于经验丰富的“代码管理员”,这个数字通常要小得多。

### 变更管理可以说,大型变更基础

设施中最重要的部分是一套工具,它将主变更分割成更小的部分,并管理测试、邮寄、审查和独立提交这些变更的过程。在 Google,这个工具被称为 Rosie,我们将在稍后检查 LSC 流程时更全面地讨论它的使用。在许多方面,Rosie 不仅仅是一个工具,而是一个在 Google 规模上制作 LSC 的完整平台。它提供了将工具生成的大量综合变更分割成更小的分片的能力,这些分片可以独立测试、审查和提交。

测试测试是

支持大规模变更的基础设施的另一个重要部分。如第 11 章所述,测试是我们验证软件是否按预期运行的重要方法之一。在应用非人类编写的变更时,这一点尤为重要。强大的测试文化和基础设施意味着其他工具可以确信这些变更不会产生意外影响。

Google 的 LSC 测试策略与常规变更略有不同,但仍然使用相同的底层 CI 基础架构。测试 LSC 不仅意味着确保大型主变更不会导致故障,还意味着每个分片都可以安全且独立地提交。由于每个分片可以包含任意文件,因此我们不使用基于项目的标准提交前测试。相反,我们会在每个分片可能影响的测试的传递闭包上运行每个分片,这一点我们之前已经讨论过。

语言支持Google 的 LSC 通

常以每种语言为基础完成,某些语言比其他语言更容易支持它们。我们发现,类型别名和转发函数等语言特性对于允许现有用户在我们引入新系统并以非原子方式将用户迁移到新系统时继续使用系统非常有用。对于缺乏这些特性的语言,通常很难逐步迁移系统。<sup>12</sup>我们还发现,静态类型语言比动态类型语言更容易执行大规模自动化更改。基于编译器的工具以及强大的静态分析提供了大量信息,我们可以使用这些信息构建工具来影响 LSC 并在无效转换进入测试阶段之前拒绝它们。不幸的结果是,像 Python、Ruby 和 JavaScript 这样动态类型的语言对维护者来说特别困难。

在许多方面,语言选择与代码寿命问题密切相关:那些更注重开发人员生产力的语言往往更难维护。虽然这不是内在的设计要求,但这正是当前最先进的做法。

最后,值得指出的是,自动语言格式化程序是 LSC 基础架构的重要组成部分。由于我们致力于优化代码以提高可读性,因此我们希望确保自动化工具产生的任何更改对于代码的直接审阅者和未来的读者来说都是可以理解的。所有 LSC 生成工具都会单独运行适合要更改的语言的自动格式化程序,这样特定于更改的工具就不需要

---

<sup>12</sup>事实上,Go 最近引入了这些类型的语言特性,专门用于支持大规模重构 (参见<https://talks.golang.org/2016/refactor.article>)。

关注格式化细节。应用自动格式化,例如[google-java-format](#)或[clang-format](#),对我们的代码库来说,自动生成的更改将“适应”人工编写的代码,从而减少未来的开发摩擦。如果没有自动格式化,大规模的自动化更改永远不会成为 Google 公认的现状。

### 案例研究 :Operation RoseHub LSC 已成

为 Google 内部文化的重要组成部分,但它们开始对更广泛的世界产生影响。迄今为止最著名的案例可能是“[Operation RoseHub](#)”。

2017 年初,Apache Commons 库中存在一个漏洞,任何在传递类路径中含有该库的易受攻击版本的 Java 应用程序都可能受到远程执行攻击。这个漏洞被称为 Mad Gadget。除此之外,它还允许贪婪的黑客加密旧金山市交通局的系统并关闭其运营。由于该漏洞的唯一要求是类路径中的某个位置存在错误的库,因此任何依赖于 GitHub 上众多开源项目之一的应用程序都可能受到攻击。

为了解决这个问题,一些有进取心的 Google 员工推出了自己的 LSC 流程。通过使用[BigQuery 等工具](#),志愿者们确定了受影响的项目,并发送了 2,600 多个补丁,以将他们的 Commons 库版本升级为可解决 Mad Gadget 问题的版本。没有使用自动化工具来管理该过程,而是有 50 多名人员使这个 LSC 得以运行。

## LSC 流程

有了这些基础设施,我们现在可以讨论实际制造 LSC 的过程。这大致分为四个阶段 (它们之间的界限非常模糊) :

1. 授权
2. 变更创建
3. 分片管理
4. 清理

通常,这些步骤发生在编写新系统、类或函数之后,但在设计新系统时牢记这些步骤也很重要。在 Google,我们的目标是在设计后续系统时考虑从旧系统迁移的路径,以便系统维护人员可以自动将其用户迁移到新系统。

## 授权

我们要求潜在作者填写一份简短的文档,解释拟议变更的原因、其对整个代码库的估计影响（即,大型变更将生成多少个较小的分片）,以及对潜在审阅者可能提出的任何问题的回答。这个过程还迫使作者思考如何以常见问题解答和拟议变更描述的形式向不熟悉变更的工程师描述变更。作者还会从正在重构的 API 的所有者那里获得“领域审查”。

然后,该提案被转发到一个电子邮件列表,其中包含大约十几个负责监督整个过程的人。讨论后,委员会会就如何推进给出反馈。例如,委员会做出的最常见更改之一是将 LSC 的所有代码审查交给一个“全局审批人”。许多首次撰写 LSC 的作者倾向于认为本地项目所有者应该审查所有内容,但对于大多数机械 LSC 来说,让一位专家了解变更的性质并围绕正确审查构建自动化更便宜。

变更获得批准后,作者可以继续提交变更。从历史上看,委员会的批准非常宽松,<sup>13</sup>通常不仅批准特定变更,还批准一系列相关变更。委员会成员可以自行决定快速处理明显的变更,而无需经过充分审议。

此流程旨在提供监督和升级途径,而不会对 LSC 作者造成太大负担。该委员会还被授权作为 LSC 担忧或冲突的升级机构:不同意变更的当地业主可以向该组织上诉,然后该组织可以仲裁任何冲突。实际上,很少需要这样做。

## 变更创建在获得所

需批准后,LSC 作者将开始进行实际的代码编辑。有时,这些可以全面生成为单个大型全局变更,随后将其分成许多较小的独立部分。通常,由于底层版本控制系统的技术限制,变更的规模太大,无法容纳在单个全局变更中。

---

<sup>13</sup>委员会直接拒绝的变更只有那些被视为危险的变更,例如将所有NULL实例转换为nullptr,或极低价值的变更,例如将拼写从英式英语更改为美式英语,反之亦然。随着我们对此类变更的经验增加,LSC 的成本下降,批准门槛也随之降低。

变更生成过程应尽可能自动化,以便当用户回到旧用法14或更改的代码中发生文本合并冲突时,可以更新父变更。偶尔,对于技术工具无法生成全局变更的罕见情况,我们会将变更生成分派给人工(参见第 472 页的“**案例研究:Operation RoseHub**” )。虽然这比自动生成变更要耗费更多的人力,但对于时间敏感的应用程序,这可以使全局变更更快地发生。

请记住,我们优化了代码库的可读性,因此无论使用什么工具生成更改,我们都希望最终的更改尽可能地像人工生成的更改。这一要求导致了样式指南和自动格式化工具的必要性(参见第 8 章)。<sup>15</sup>

分片和提交在生成全局更改后,作

者开始运行 Rosie。Rosie 会进行大规模更改,并根据项目边界和所有权规则将其分片为可以原子提交的更改。然后,它将每个单独分片的更改放入独立的测试邮件提交管道中。Rosie 可能是 Google 开发者基础架构其他部分的重度用户,因此它会限制任何给定 LSC 的未完成分片数量,以较低优先级运行,并与基础架构的其余部分进行通信,了解在我们的共享测试基础架构上可以生成多少负载。

下面我们将详细讨论每个分片的具体测试邮件提交过程。

### 牛与宠物

当我们提到分布式计算环境中的单个机器时,我们经常使用“牛和宠物”的类比,但相同的原则也适用于代码库内的变化。

在 Google,与大多数组织一样,代码库的典型更改都是由致力于特定功能或错误修复的个别工程师手工完成的。工程师可能要花几天或几周的时间来完成单个更改的创建、测试和审核。他们非常了解更改,并在更改最终提交到主存储库时感到自豪。创建这样的更改就像拥有和饲养一只心爱的宠物一样。

---

<sup>14</sup>发生这种情况的原因有很多:从现有示例中复制粘贴、提交已修改的更改、在开发中已经存在一段时间了,或者只是依赖旧习惯。

<sup>15</sup>实际上,这正是 C++ 的 clang-format 最初的工作背后的原因。

相比之下,有效处理 LSC 需要高度自动化,并会产生大量单独的更改。在这种环境下,我们发现将特定更改视为牲畜很有用:匿名且不露面的提交可能会在任何给定时间回滚或以其他方式拒绝,除非整个牛群受到影响,否则成本很低。这种情况通常是由于测试未发现的不可预见的问题,甚至是合并冲突等简单问题而发生的。

对于“宠爱”的提交,很难不把拒绝当成是针对个人的,但当处理大量变更作为大规模变更的一部分时,这只是工作的性质。自动化意味着可以以非常低的成本更新工具并生成新的变更,因此偶尔损失一些牛不是问题。

### 测试每

个独立分片都通过 Google 的 CI 框架 TAP 运行进行测试。

我们运行依赖于给定更改中的每个文件的测试,这通常会给我们的 CI 系统带来高负载。

这听起来计算成本可能很高,但实际上,在我们的代码库中数百万个测试中,绝大多数分片仅影响不到一千个测试。

对于影响较大的测试,我们可以将它们组合在一起:首先对所有分片运行所有受影响测试的联合,然后对每个单独的分片运行其受影响测试与第一次运行失败的测试的交集。大多数联合测试都会导致代码库中的几乎所有测试都运行,因此对该批分片添加额外的更改几乎是免费的。

运行如此大量的测试的缺点之一是,独立的低概率事件在足够大的规模下几乎是必然发生的。不稳定和脆弱的测试(如第 11 章中讨论的测试)通常不会损害编写和维护它们的团队,但对于 LSC 作者来说尤其困难。虽然不稳定测试对单个团队的影响相当小,但它们会严重影响 LSC 系统的吞吐量。自动不稳定检测和消除系统有助于解决这个问题,但要确保编写不稳定测试的团队承担成本,可能需要持续努力。

根据我们对 LSC 的经验,作为保留语义的机器生成更改,我们现在对单个更改的正确性比对近期存在任何不稳定历史的测试更有信心以至于在通过我们的自动化工具提交时,最近不稳定的测试现在会被忽略。从理论上讲,这意味着单个分片可能导致回归,而这种回归只能由不稳定测试从不稳定到失败来检测。

实际上,我们很少看到这种情况,因此通过人机交流而不是自动化来处理它更容易。

对于任何 LSC 流程,各个分片都应该可以独立提交。这意味着它们之间没有任何相互依赖性,或者分片机制可以

将相关更改（例如对头文件及其实现的更改）分组在一起。

与任何其他更改一样，大规模更改分片在审核和提交之前也必须通过特定于项目的检查。

### 邮寄审阅者在

Rosie 通过测试验证了更改是安全的之后，它会将更改邮寄给适当的审阅者。在像 Google 这样拥有数千名工程师的大公司中，审阅者发现本身就是一个具有挑战性的问题。回想一下[第 9 章](#)，存储库中的代码是通过 OWNERS 文件组织的，这些文件列出了对存储库中特定子树具有批准权限的用户。Rosie 使用所有者检测服务，该服务可以理解这些 OWNERS 文件，并根据每个所有者审阅相关特定分片的预期能力对其进行加权。如果某个所有者没有响应，Rosie 会自动添加其他审阅者，以便及时审阅更改。

作为邮件流程的一部分，Rosie 还运行每个项目的预提交工具，这些工具可能会执行额外的检查。对于 LSC，我们有选择地禁用某些检查，例如针对非标准变更描述格式的检查。虽然这些检查对于特定项目的个别变更很有用，但它们是整个代码库的异质性来源，并可能给 LSC 流程带来重大阻力。这种异质性是扩展我们的流程和系统的障碍，不能指望 LSC 工具和作者了解每个团队的特殊政策。

我们还积极忽略在相关变更之前存在的提交前检查失败。在处理单个项目时，工程师可以轻松修复这些问题并继续其原始工作，但当在 Google 的代码库中创建 LSC 时，该技术无法扩展。作为他们与基础设施之间的社会契约的一部分，本地代码所有者有责任确保其代码库中不存在任何预先存在的故障

真正的团队。

### 审查与其

他更改一样，Rosie 生成的更改也需要经过标准代码审查流程。在实践中，我们发现当地所有者通常不会像对待常规更改那样严格地对待 LSC —— 他们太信任生成 LSC 的工程师了。理想情况下，这些更改会像其他更改一样进行审查，但在实践中，当地项目所有者已经开始信任基础设施团队，以至于这些更改通常只进行粗略的审查。我们只将需要当地所有者审查以供了解背景信息（而不仅仅是获得批准权限）的更改发送给他们。所有其他更改都可以交给“全局审批人”：具有所有权的人，可以批准整个存储库中的任何更改。

使用全局审批者时，所有单个分片都分配给该人，而不是分配给不同项目的各个所有者。全局审批者通常

对他们正在审查的语言和/或库有特定的了解，并与大规模变更作者合作，了解预期的变更类型。他们知道变更的细节是什么，以及可能存在哪些潜在的失败模式，并可以相应地定制他们的工作流程。

全局审阅者不再逐一审阅每个变更，而是使用一套单独的基于模式的工具来审阅每个变更，并自动批准符合他们期望的变更。因此，他们只需要手动检查由于合并冲突或工具故障而导致异常的一小部分，这使得流程能够很好地扩展。

提交最后，

提交单个更改。与邮寄步骤一样，我们确保更改通过各种项目提交前检查，然后才真正提交到存储库。

借助 Rosie，我们每天能够在整个 Google 代码库中高效地创建、测试、审查和提交数千项变更，并让团队能够高效地迁移用户。过去需要最终做出的技术决策（例如广泛使用的符号的名称或代码库中流行类的位置）不再需要最终做出。

清理不同

的 LSC 对“完成”有不同的定义，从完全移除旧系统到仅迁移高价值引用，让旧系统自然消失。<sup>16</sup>在几乎所有情况下，重要的是要有一个系统来防止再次引入大规模变更努力移除的符号或系统。在 Google，我们使用第 20 章和第 19 章中提到的 Tricorder 框架在审查时标记工程师引入已弃用对象的新用途，这已被证明是一种防止倒退的有效方法。

我们在第 15 章中详细讨论整个弃用过程。

## 结论

LSC 是 Google 软件工程生态系统的重要组成部分。在设计时，它们开辟了更多的可能性，因为它们知道一些设计决策不需要像以前那样固定不变。LSC 流程还允许核心基础设施的维护者将大量 Google 代码库从旧系统、语言版本和库习语迁移到新系统、语言版本和库习语，同时保持代码库的完整性。

---

<sup>16</sup>可悲的是，我们最想有机分解的系统正是那些最能抵御分解的系统。

它们是代码生态系统的塑料六包装环。

无论是在空间上还是在时间上,都是持续的。而这一切只需要几十名工程师支持数万名工程师就能实现。

无论你的组织规模如何,考虑如何在你的源代码集合中进行这种彻底的更改都是合理的。无论是出于选择还是必要,拥有这种能力将使你的组织在扩展时具有更大的灵活性,同时保持源代码随着时间的推移具有可塑性。

## TL;DR

- LSC 流程使得重新思考某些技术不变性成为可能决定。
- 大规模重构的传统模式被打破。 · 制造 LSC 意味着养成制造 LSC 的习惯。

# 持续集成

作者:Rachel Tannenbaum  
由 Lisa Carey 编辑

持续集成 (CI) 通常被定义为 “一种软件开发实践, 团队成员频繁集成他们的工作 [...] 每次集成都通过自动构建 (包括测试) 进行验证, 以尽快检测集成错误。”<sup>1</sup> 简而言之, CI 的基本目标是尽早自动捕获有问题的变化。

实际上, “频繁集成工作”对现代分布式应用程序意味着什么? 当今的系统除了存储库中最新版本的代码外, 还有许多活动部件。事实上, 随着最近微服务的趋势, 破坏应用程序的更改不太可能存在于项目的直接代码库中, 而更可能存在于网络调用另一端的松散耦合的微服务中。传统的持续构建测试二进制文件中的更改, 而这种扩展可能会测试对上游微服务的更改。依赖关系只是从函数调用堆栈转移到 HTTP 请求或远程过程调用 (RPC)。

除了代码依赖关系之外, 应用程序可能会定期提取数据或更新机器学习模型。它可能在不断发展的操作系统、运行时、云托管服务和设备上执行。它可能是位于不断增长的平台之上的功能, 也可能是必须适应不断增长的功能基础的平台。所有这些都应被视为依赖关系, 我们也应该致力于“持续集成”它们的变化。更复杂的是, 这些不断变化的组件通常由我们团队、组织或公司以外的开发人员拥有, 并按照自己的时间表部署。

---

<sup>1</sup> <https://www.martinfowler.com/articles/continuousIntegration.html>

因此,在当今世界,特别是在大规模开发时,对 CI 的更好的定义可能是这样的:

持续集成 (2) :持续组装和测试我们整个复杂且快速发展的生态系统。

从测试的角度来概念化 CI 是很自然的,因为两者紧密相关,我们将在本章中一直这样做。在前面的章节中,我们讨论了从单元到集成再到更大范围的系统等各种测试。

从测试角度来看,CI 是一个范例,用于指导以下内容:

- 在开发/发布工作流程中以代码形式运行哪些测试 (以及其他)的变化不断融入其中
- 如何在每个点组成被测系统 (SUT),平衡保真度和设置成本等问题

例如,我们在提交前运行哪些测试,我们将哪些测试留到提交后,我们将哪些测试留到稍后的阶段部署?因此,我们如何在每个阶段表示我们的 SUT?正如您所想象的,提交前 SUT 的要求可能与测试阶段环境的要求有很大不同。例如,对于从提交前等待审核的代码构建的应用程序来说,与实际生产后端进行通信可能会很危险 (想想安全和配额漏洞),而这对于阶段环境来说通常是可以接受的。

为什么我们要尝试优化这种通常很微妙的平衡,即在“正确的时间”用 CI 测试“正确的事物”?大量的先前工作已经证实了 CI 对工程组织和整个业务的好处。<sup>2</sup>这些结果由一个强有力地保证驱动:可验证且及时的证明,表明应用程序可以进入下一阶段。我们不需要只是希望所有贡献者都非常小心、负责和彻底;相反,我们可以保证我们的应用程序在从构建到发布的各个阶段的工作状态,从而提高我们对产品的信心和质量,以及我们的生产力

團隊。

在本章的其余部分,我们将介绍一些关键的 CI 概念、最佳实践和挑战,然后介绍我们的持续构建工具 TAP,并深入研究一个应用程序的 CI 转换,从而了解我们如何在 Google 管理 CI。

---

<sup>2</sup> Forsgren, Nicole 等人 (2018 年)。加速:精益软件和 DevOps 的科学:构建和扩展高效技术组织。IT 革命。

# CI 概念

首先,让我们先了解一下 CI 的一些核心概念。

快速反馈循环如第 11 章所述,错

误越晚被发现,其成本几乎呈指数增长。图 23-1 显示了有问题的代码更改在其生命周期中可能被捕获的所有位置。

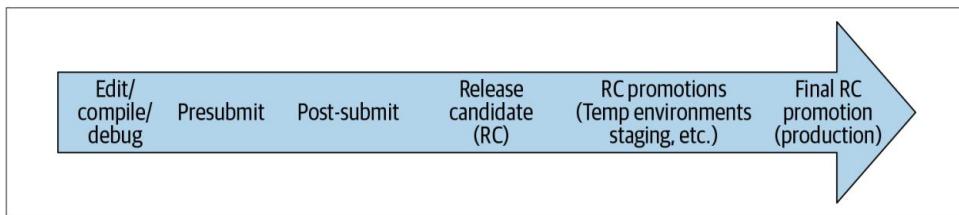


图 23-1. 代码更改的生命周期

一般而言,随着问题在我们的图表中向“右”发展,它们的成本会变得更加高昂,原因如下:

- 它们必须由可能不熟悉有问题的代码更改的工程师进行分类。
- 代码变更作者需要做更多工作来回忆和调查  
變化。
- 它们对其他人产生负面影响,无论是对工程师的工作,还是最终对  
最终用户。

为了最大限度地降低错误成本,CI 鼓励我们使用快速反馈循环。<sup>3</sup>每次我们将代码(或其他)更改集成到测试场景中并观察结果时,我们都会获得一个新的反馈循环。反馈可以采用多种形式;以下是一些常见的形式(按从快到慢的顺序排列):

- 本地开发的编辑-编译-调试循环 · 在提交前向代码变更
- 作者自动提供测试结果 · 两个项目变更之间的集成错误,在两个项目提  
交并一起测试后检测到(即提交后)

---

<sup>3</sup>有时这也被称为“测试左移”。

当上游服务部署其最新更改时,我们的项目与上游微服务依赖项之间存在不兼容性,由我们的暂存环境中的 QA 测试人员检测到

- 在外部用户之前选择使用某项功能的内部用户的错误报告 · 外部用户或媒体的错误或中断报告

金丝雀发布 (即先部署到一小部分生产环境)有助于最大限度地减少在生产环境中出现的问题,在部署到所有生产环境之前,先在生产环境中建立子集初始反馈循环。然而,金丝雀发布也会带来问题,尤其是当同时部署多个版本时,部署之间的兼容性问题。这有时被称为版本偏差,即分布式系统的状态,其中包含多个不兼容的代码、数据和/或配置版本。与我们在本书中讨论的许多问题一样,版本偏差是尝试开发和管理软件时可能出现的另一个具有挑战性的问题。

随着时间的推移。

实验和功能标记是非常强大的反馈循环。它们通过隔离可在生产中动态切换的模块化组件中的更改来降低部署风险。严重依赖功能标记保护是持续交付的常见范例,我们将在[第 24 章中进一步探讨](#)。

### 可访问且可操作的反馈

同样重要的是,CI 的反馈应广泛可用。除了我们在代码可见性方面的开放文化之外,我们对测试报告也有类似的感受。我们有一个统一的测试报告系统,任何人都可以轻松查找构建或测试运行,包括所有日志 (不包括用户个人身份信息 [PII]),无论是单个工程师的本地运行还是自动开发或暂存构建。

除了日志之外,我们的测试报告系统还提供构建或测试目标何时开始失败的详细历史记录,包括每次运行时构建中断的位置、运行位置和运行者。我们还有一个不稳定性分类系统,该系统使用统计数据在 Google 范围内对不稳定性进行分类,因此工程师无需自己弄清楚这一点,即可确定他们的更改是否破坏了另一个项目的测试 (如果测试不稳定性:可能不会)。

测试历史记录的可视性使工程师能够分享和协作反馈,这是不同团队诊断系统间集成故障并从中吸取教训的基本要求。同样,Google 的漏洞 (例如工单或问题)也是公开的,所有人都可以查看和学习完整的评论历史记录 (客户 PII 再次除外)。

最后,CI 测试的任何反馈不仅应易于访问,还应具有可操作性 便于查找和修复问题。我们将看一个改进用户反馈的示例。

本章后面的案例研究中不友好的反馈。通过提高测试输出的可读性，您可以自动理解反馈。

## 自动化

众所周知，[自动化开发相关任务可以节省工程资源](#)从长远来看。直观地说，因为我们通过将流程定义为代码来实现流程自动化，所以在签入更改时进行同行评审将降低出错的可能性。当然，自动化流程与任何其他软件一样，都会有错误；但如果有效实施，它们仍然比工程师手动尝试更快、更容易、更可靠。

具体来说，CI 通过持续构建和持续交付来自动化构建和发布流程。整个过程中都会应用持续测试，我们将在下一节中讨论。

### 持续构建

持续构建 (CB) 集成 head<sup>4</sup> 上的最新代码更改，并运行自动构建和测试。由于 CB 不仅构建代码，还运行测试，因此“中断构建”或“构建失败”既包括中断测试，也包括中断编译。

提交变更后，CB 应运行所有相关测试。如果变更通过了所有测试，CB 会将其标记为通过或“绿色”，因为它通常会显示在用户界面 (UI) 中。此过程有效地在存储库中引入了两个不同版本的 head：真实 head（即提交的最新变更）和绿色 head（即 CB 已验证的最新变更）。工程师能够在本地开发中同步到任一版本。在编码变更时，通常会与绿色 head 同步以使用由 CB 验证的稳定环境，但有一个流程要求在提交之前将变更同步到真实 head。

### 持续交付持续交付 (CD)

[第 24 章将更详细地讨论](#)的第一步是发布自动化，它不断将最新的代码和配置从头组装到发布候选版本中。在 Google，大多数团队在绿色而非真正头版时完成这些工作。

---

<sup>4</sup> Head 是我们 monorepo 中最新版本的代码。在其他工作流程中，这也被称为 master、mainline 或 trunk。相应地，在 head 进行集成也称为基于 trunk 的开发。

候选发布版本 (RC) :由自动化流程5创建的具有凝聚力的可部署单元,由通过持续构建的代码、配置和其他依赖项组成。

请注意,我们在候选版本中包含了配置。这非常重要,尽管在候选版本发布时,配置在不同环境之间可能会略有不同。我们不一定提倡您将配置编译到二进制文件中。实际上,对于许多场景,我们建议使用动态配置,例如实验或功能标志。<sup>6</sup>

相反,我们说您拥有的任何静态配置都应作为候选版本的一部分进行推广,以便可以与相应的代码一起进行测试。请记住,很大部分生产错误是由“愚蠢”的配置问题引起的,因此测试配置和测试代码同样重要(并与将使用它的相同代码一起测试)。版本偏差通常会在此候选版本推广过程中被捕获。当然,这假设您的静态配置处于版本控制中。在Google,静态配置与代码一起处于版本控制中,因此要经过相同的代码审查流程。

然后我们将 CD 定义如下:

持续交付 (CD):持续组装候选版本,然后在一系列环境中推广和测试这些候选版本 - 有时会达到生产水平,有时则不会。

推广和部署过程通常取决于团队。我们将展示我们的案例研究如何引导这一过程。

对于希望从生产中的新更改(例如持续部署)中不断获得反馈的Google团队来说,通常无法持续将整个二进制文件(通常非常大)推送到绿色状态。因此,通过实验或功能标记进行选择性持续部署是一种常见策略。<sup>7</sup>随着RC在环境中的进展,其工作件(例如二进制文件、容器)理想情况下不应重新编译或重建。使用Docker等容器有助于从本地开发开始在环境之间强制RC的一致性。同样,使用Kubernetes(或在我们的例子中通常是Borg)等编排工具,有助于确保部署之间的一致性。通过确保

---

<sup>5</sup>在Google,发布自动化由与TAP不同的系统管理。我们不会重点介绍发布自动化如何自动化组装RC,但如果您感兴趣,我们建议您参阅[站点可靠性工程](#)(O'Reilly)其中详细讨论了我们的发布自动化技术(称为Rapid的系统)。

<sup>6</sup>第24章将进一步讨论带有实验和功能标志的6 CD。

<sup>7</sup>我们称之为“空中相撞”,因为发生这种情况的概率极低;然而,当这种情况发生时,结果可能会相当令人惊讶。

通过我们在环境之间的发布和部署,我们实现了更高保真的早期测试,并且减少了生产中的意外。

持续测试让我们看看在代

码更改的整个生命周期中应用持续测试(CT)时CB和CD如何配合,如图23-2所示。

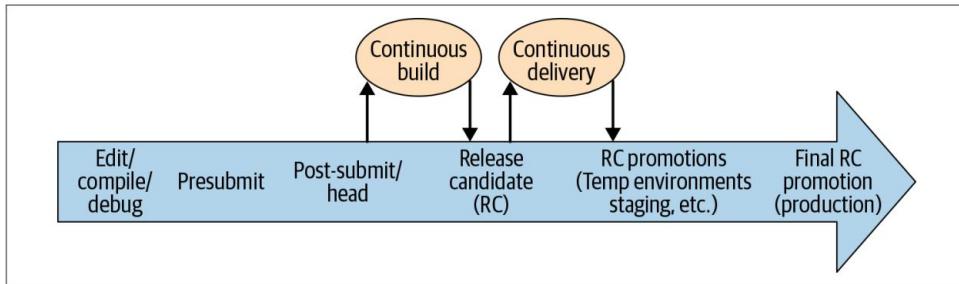


图23-2. CB和CD代码更改的生命周期

向右的箭头表示单个代码更改从本地开发到生产的进展。同样,我们在CI中的主要目标之一是确定在此进展中要测试什么。在本章后面,我们将介绍不同的测试阶段,并提供一些关于在提交前和提交后以及在RC及以后要测试什么的注意事项。我们将展示,随着向右移动,代码更改将接受范围越来越大的自动化测试。

为什么预提交还不够为了尽

快发现有问题的变化,并能够在提交前运行自动测试,您可能会想:为什么不在提交前运行所有测试?

主要原因是成本太高。工程师的生产力极其宝贵,在代码提交期间等待很长时间来运行每个测试可能会造成严重破坏。此外,通过消除预先提交的约束,如果测试通过的频率远高于失败的频率,则可以获得很大的效率提升。例如,可以将运行的测试限制在特定范围内,或者根据预测其检测到故障的可能性的模型进行选择。

类似地,如果工程师在提交前因为与代码更改无关的不稳定或不稳定性导致的故障而受阻,那么他们的代价将是高昂的。

另一个原因是,在我们运行提交前测试以确认更改安全时,底层存储库可能以与正在测试的更改不兼容的方式进行了更改。也就是说,两个涉及完全不同文件的更改可能会导致测试失败。我们称之为“空中相撞”,

虽然这种情况很少见,但在我们的规模下,这种情况几乎每天都会发生。小型存储库或项目的 CI 系统可以通过序列化提交来避免此问题,这样即将输入的内容和刚刚输入的内容之间就没有区别了。

### 提交前与提交后那么,提交前

应该运行哪些测试?我们的一般经验法则是:只运行快速、可靠的测试。您可以接受提交前的一些覆盖范围损失,但这意味着您需要捕获提交后出现的任何问题,并接受一定数量的回滚。在提交后,您可以接受更长的时间和一些不稳定性,只要您有适当的机制来处理它。



我们将[在第 493 页的“Google 的 CI”](#)中展示 TAP 和我们的案例研究如何处理故障管理。

我们不想让工程师们因为等待时间太长而浪费宝贵的生产力,因为测试速度太慢或测试太多。我们通常将提交前测试限制在发生更改的项目上。我们还会同时运行测试,因此还需要考虑资源决策。最后,我们不想在提交前运行不可靠的测试,因为让许多工程师受到影响,调试与他们的代码更改无关的相同问题的成本太高了。

Google 的大多数团队都在 presubmit<sup>8</sup> 上运行小型测试(如单元测试)。这些测试显然是最快、最可靠的测试。是否以及如何在 presubmit 上运行更大范围的测试是一个更有趣的问题,而且每个团队都不一样。对于确实想要运行这些测试的团队来说,密封测试是一种行之有效的方法,可以减少其固有的不稳定性。另一种选择是允许大范围的测试在 presubmit 上不可靠,但当它们开始失败时,就积极禁用它们。

### 发布候选测试在代码更改

通过 CB 后(如果出现失败,则可能需要多个周期),它将很快遇到 CD 并被纳入待定的发布候选中。

随着 CD 构建 RC,它将针对整个候选版本运行更大规模的测试。我们通过在一系列测试环境中推广候选版本并在每次部署时对其进行测试来测试候选版本。这可以包括沙盒、临时环境的组合

---

<sup>8</sup> Google 的每个团队都会配置一部分项目测试,在提交前(而不是提交后)运行。

事实上,我们的持续构建实际上优化了一些提交前的测试,这些测试将在幕后保存以供提交后使用。我们将在本章后面进一步讨论这一点。

开发环境和共享测试环境,如开发或暂存环境。在共享环境中也通常包括一些 RC 的手动 QA 测试。

有几个原因说明为什么对 RC 运行全面的自动化测试套件很重要,即使它与 CB 在提交后针对代码运行的套件相同 (假设 CD 在绿色处切断) :

作为健全性检查,我们仔细检查在 RC 中剪切和重新编译代码时没有发生任何奇怪的事情。

对于可审计性,如果工程师想要检查 RC 的测试结果,他们随时可用并且与 RC 相关联,因此他们无需挖掘 CB 日志来找到它们。

允许挑选如果您对 RC 应用挑选修复,您的源代码现在已经与 CB 测试的最新版本有所不同。

对于紧急推动在这种情况下,CD 可以从真实头部切断并运行必要的最少测试,以对紧急推动充满信心,而无需等待完整的 CB 通过。

生产测试我们持续、

自动化的测试流程会一直持续到最终部署环境:生产环境。我们应该对生产环境 (有时称为探测器) 运行与之前对候选版本进行的相同测试套件,以验证:1)根据我们的测试,生产环境的工作状态;2)根据生产环境,我们测试的相关性。

在应用程序进展的每个步骤中进行持续测试,每个步骤都有其自身的权衡,提醒我们“纵深防御”方法在捕获错误方面的价值 我们依赖的质量和稳定性不仅是一项技术或政策,而是多种测试方法的结合。

### CI 正在发出警报

泰特斯·温特斯

与负责任地运行生产系统一样,可持续地维护软件系统也需要持续的自动监控。正如我们使用监控和警报系统来了解生产系统如何响应变化一样,CI 揭示了我们的软件如何响应其环境的变化。生产监控依赖于正在运行的系统的被动警报和主动探测器,而 CI

使用单元测试和集成测试来检测软件部署之前的变化。

通过对这两个领域进行比较,我们可以将一个领域的知识应用到另一个领域。

在开发人员工作流程中,CI 和警报都具有相同的总体目的 - 尽快合理地识别问题。CI 强调开发人员工作流程的早期阶段,并通过发现测试失败来发现问题。警报侧重于同一工作流程的后期,并通过监控指标并在指标超过某个阈值时报告来发现问题。两者都是“尽快自动识别问题”的形式。

管理良好的警报系统有助于确保满足您的服务级别目标 (SLO)。良好的 CI 系统有助于确保您的构建处于良好状态 - 代码可以编译、测试可以通过,并且您可以在需要时部署新版本。这两个领域的最佳实践政策都非常注重保真度和可操作警报的理念:只有当重要的底层不变量被违反时,测试才会失败,而不是因为测试脆弱或不稳定。每隔几次 CI 运行就会失败的不稳定测试与每隔几分钟发出一次虚假警报并为值班人员生成页面一样成问题。如果它不可操作,就不应该发出警报。如果它实际上没有违反 SUT 的不变量,就不应该是测试失败。

CI 和警报共享一个底层概念框架。例如,本地信号 (单元测试、孤立统计数据监控/基于原因的警报) 和交叉依赖信号 (集成和发布测试、黑盒探测) 之间存在类似的关系。聚合系统是否正常工作的最高保真度指标是端到端信号,但我们为这种保真度付出的代价是不稳定、资源成本增加以及调试根本原因的难度。

类似地,我们看到两个领域的故障模式之间存在潜在的联系。

脆弱的基于原因的警报会在超过任意阈值 (比如,过去一小时内的重试次数) 时触发,而该阈值与最终用户所看到的系统健康状况之间不一定存在根本联系。脆弱的测试在违反任意测试要求或不变量时失败,而该不变量与被测软件的正确性之间不一定存在根本联系。在大多数情况下,这些测试很容易编写,并且可能有助于调试更大的问题。在这两种情况下,它们都是整体健康状况 / 正确性的粗略代理,无法捕捉整体行为。如果你没有简单的端到端探测,但你确实可以轻松收集一些汇总统计数据,那么团队将根据任意统计数据编写阈值警报。如果你没有一种高级的方式来表示“如果解码后的图像与此解码后的图像不大致相同,则测试失败”,团队将构建断言字节流相同的测试。

基于原因的警报和脆弱性测试仍然有价值;但它们并不是在警报场景中识别潜在问题的理想方式。如果发生实际故障,拥有更多可用的调试详细信息可能会很有用。当 SRE 调试中断时,拥有以下形式的信息会很有用:“一小时前,用户开始遇到更多失败的请求。大约在同一时间,重试次数

开始出现问题。让我们从那里开始调查吧。”同样,脆弱测试仍然可以提供额外的调试信息：“图像渲染管道开始吐出垃圾。其中一个单元测试表明我们从 JPEG 压缩器获得了不同的字节。让我们从那里开始调查吧。”

尽管监控和警报被视为 SRE/生产管理领域的一部分,其中“错误预算”的见解得到了很好的理解,<sup>9</sup>但 CI 的观点仍然倾向于绝对化。将 CI 定义为警报的“左移”开始提出推理这些策略的方法,并提出更好的最佳实践:

- 实现 CI 100% 的绿色率,就像实现生产服务 100% 的正常运行时间一样,成本非常高昂。如果这确实是你的目标,那么最大的问题之一就是测试和提交之间的竞争条件。
- 将每个警报视为同等的警报原因通常并不是正确的方法。如果在生产中触发警报但服务实际上并未受到影响,则使警报静音是正确的选择。测试失败也是如此:在我们的 CI 系统学会如何说“已知此测试因无关原因而失败”之前,我们应该更自由地接受禁用失败测试的更改。并非所有测试失败都预示着即将出现生产问题。
- 那些说“如果我们最新的 CI 结果不理想,则任何人都无法提交”的政策可能是错误的。如果 CI 报告了问题,则在让人们提交或加剧问题之前,绝对应该调查此类失败。但如果根本原因很容易理解并且显然不会影响生产,则阻止提交是不合理的。

“CI 即警报”这一见解很新,我们仍在研究如何全面进行比较。考虑到所涉及的风险更高,SRE 投入大量精力研究监控和警报方面的最佳实践,而 CI 则被视为一种奢侈功能,这并不令人意外。<sup>10</sup>在接下来的几年里,软件工程的任务将是看看现有的 SRE 实践可以在 CI 环境中重新概念化哪些方面,以帮助重新制定测试和 CI 格局。也许测试方面的最佳实践可以帮助明确监控和警报方面的目标和政策。

<sup>9</sup>以 100% 正常运行时间为目是一个错误的目标。选择 99.9% 或 99.999% 之类的指标作为业务或产品权衡,定义并监控您的实际正常运行时间,并使用该“预算”作为您愿意以多大力度推动风险发布的输入。

<sup>10</sup>我们认为 CI 实际上对软件工程生态系统至关重要:必不可少,而非奢侈品。但这一点尚未得到普遍理解。

## CI 挑战我们讨

论了一些 CI 中已建立的最佳实践，并介绍了其中涉及的一些挑战，例如不稳定、缓慢、冲突或提交前测试过多可能会对工程师生产力造成潜在影响。实施 CI 时的一些常见额外挑战包括：

- 提交前优化，包括考虑到我们已经描述的潜在问题，在提交前要运行哪些测试，以及如何运行这些测试。

查找原因并隔离故障：哪个代码或其他更改导致了问题，它发生在哪个系统中？“集成上游微服务”是分布式架构中隔离故障的一种方法，当你想弄清楚问题是源自你自己的服务器还是后端时。在这种方法中，你将稳定的服务器与上游微服务的新服务器组合在一起。（因此，你将微服务的最新更改集成到测试中。）由于版本偏差，这种方法可能特别具有挑战性：这些环境不仅通常不兼容，而且你还可能遇到误报 - 发生在特定阶段组合中但在生产中实际上不会发现的问题。

- 资源限制：测试需要资源才能运行，大型测试可能非常昂贵。此外，在整个过程中插入自动化测试的基础设施成本可能相当高昂。

### 还有故障管理的挑战 测试失败时该怎么办。

虽然较小的问题通常可以快速修复，但我们的许多团队发现，当涉及大型端到端测试时，很难拥有始终有效的测试套件。它们天生就容易损坏或不稳定，难以调试；需要有一种机制来暂时禁用并跟踪它们，以便发布能够继续进行。Google 的一种常用技术是使用由值班工程师或发布工程师提交的 bug “热门列表”，并分类到适当的团队。更好的是，这些 bug 可以自动生成和归档。我们的一些大型产品，如 Google Web Server (GWS) 和 Google Assistant，就是这样做的。这些热门列表应该经过精心策划，以确保立即修复任何阻碍发布的 bug。

未发布的阻碍因素也应该修复；它们不那么紧急，但也应该优先处理，以便测试套件保持有用，而不仅仅是一堆日益增多的无效旧测试。通常，端到端测试失败所发现的问题实际上是测试问题，而不是代码问题。

不稳定的测试给这个过程带来了另一个问题。它们像失败的测试一样削弱信心，但找到一个可以回滚的更改通常更困难，因为失败不会一直发生。一些团队依靠工具暂时从提交前移除此类不稳定的测试，同时调查和修复不稳定问题。这可以保持信心，同时留出更多时间来解决问题。

测试不稳定性是我们在预提交的背景下已经研究过的另一个重大挑战。处理此问题的一种策略是允许多次尝试测试。这是团队使用的常见测试配置设置。此外，在测试代码中，可以在各种特定点引入重试。

有助于解决测试不稳定性（和其他 CI 挑战）的另一种方法是密封测试，我们将在下一节中讨论。

### 密封测试因为与实时

后端对话不可靠，所以我们经常使用密封后端对于更大范围的测试。当我们想在提交前运行这些测试时，这尤其有用，因为稳定性是最重要的。在第 11 章中，我们介绍了密封测试的概念：

密封测试：测试针对完全独立的测试环境（即应用服务器和资源）运行（即没有生产后端等外部依赖）。

密封测试有两个重要特性：更高的确定性（即稳定性）和隔离性。密封服务器仍然容易受到一些非确定性因素的影响，例如系统时间、随机数生成和竞争条件。但是，测试的内容不会根据外部依赖关系而改变，因此当您使用相同的应用程序和测试代码两次运行测试时，应该会得到相同的结果。如果密封测试失败，您就知道这是由于应用程序代码或测试的改造成成的（但有一点需要注意：它们也可能由于密封测试环境的重组而失败，但这种情况不应该经常改变）。因此，当 CI 系统在数小时或数天后重新运行测试以提供更多信号时，密封性使测试失败的范围更容易缩小。

另一个重要属性是隔离性，这意味着生产中的问题不应影响这些测试。我们通常也在同一台机器上运行这些测试，因此我们不必担心网络连接问题。反之亦然：运行密封测试引起的问题不应影响生产。

密封测试的成功不应该取决于运行测试的用户。这允许人们重现 CI 系统运行的测试，并允许人们（例如库开发人员）运行其他团队拥有的测试。

一种密封后端是假的。如第 13 章所述，这些后端比运行真正的后端更便宜，但它们需要维护，并且保真度有限。

实现值得进行预提交的集成测试的最干净的方法是使用完全封闭的设置（即启动整个堆栈沙盒<sup>11</sup>），并且Google为数据库等常用组件提供了开箱即用的沙盒配置，以简化此过程。对于组件较少的小型应用程序来说，这更可行，但Google也有例外，甚至有一个（DisplayAds开发的）应用程序在每次预提交时以及提交后持续从头启动大约400台服务器。不过，自创建该系统以来，记录/重放已成为大型系统更受欢迎的范例，并且往往比启动大型沙盒堆栈更便宜。

记录/重放（参见第14章）系统记录实时后端响应，缓存它们，并在封闭的测试环境中重放它们。记录/重放是减少测试不稳定性的有效工具，但它的一个缺点是会导致测试变得脆弱：很难在以下两点之间取得平衡：

#### 误报测试可能不应

该通过，因为我们过多地访问缓存并错过了在捕获新响应时会出现的问题。

#### 误判 测试失败，但

可能不应该失败，因为我们访问缓存的次数太少。这需要更新响应，这可能需要很长时间，并导致必须修复的测试失败，其中许多可能不是实际问题。此过程通常会阻止提交，这并不理想。

理想情况下，记录/重放系统应该只检测有问题的更改，并且只在请求发生有意义的更改时检测缓存未命中。如果该更改导致问题，代码更改作者将使用更新后的响应重新运行测试，发现测试仍然失败，从而提醒问题。实际上，在大型且不断变化的系统中，了解请求何时发生有意义的更改可能非常困难。

#### 密封的 Google Assistant Google

Assistant为工程师提供了一个运行端到端测试的框架，包括一个测试装置，该装置具有设置查询的功能，指定是否在手机或智能家居设备上进行模拟，以及在与Google Assistant交换的过程中验证响应。

---

<sup>11</sup>在实践中，建立一个完全沙盒测试环境通常很困难，但通过最小化外部依赖可以实现所需的稳定性。

其最成功的案例之一是使其测试套件在提交前完全密封。当团队以前在提交前运行非密封测试时,测试经常会失败。有时,团队会看到超过 50 个代码更改被绕过并忽略测试结果。在将提交前测试改为密封测试后,团队将运行时间缩短了 14 倍,几乎没有出现任何问题。它仍然会看到故障,但这些故障往往很容易找到并回滚。

现在,非封闭测试已被推迟到提交后阶段,这反而会导致失败在那里累积。调试失败的端到端测试仍然很困难,有些团队甚至没有时间去尝试,所以他们只是禁用它们。这比让所有人停止所有开发要好,但它可能会导致生产失败。

团队当前面临的挑战之一是继续微调其缓存机制,以便提交前可以捕获更多类型的问题(这些问题在过去仅在提交后发现),而不会引入过多的脆弱性。

另一个问题是,鉴于组件正在转移到自己的微服务中,如何对分散式助手进行提交前测试。由于助手的堆栈庞大而复杂,因此在提交前运行封闭堆栈的成本(就工程工作、协调和资源而言)将非常高。

最后,团队利用这种分散化,制定出一种巧妙的提交后故障隔离策略。对于 Assistant 中的 N 个微服务,团队将运行一个提交后环境,其中包含在 head 构建的微服务,以及其他 N-1 个服务的生产版本(或接近生产版本),以将问题隔离到新建的服务器。这种设置通常需要花费  $O(N^2)$  的成本,但团队利用一项名为热交换的酷炫功能将此成本降低到  $O(N)$ 。本质上,热交换允许请求指示服务器“交换”要调用的后端地址,而不是通常的地址。因此,只需要运行 N 个服务器,每个在 head 切割的微服务运行一个服务器 并且它们可以重复使用交换到这 N 个“环境”中的每一个的同一组生产后端。

正如我们在本节中看到的,密封测试既可以减少大范围测试中的不稳定性,又有助于隔离故障,从而解决我们在上一节中确定的两个重大 CI 挑战。但是,密封后端也可能更昂贵,因为它们会占用更多资源并且设置速度较慢。许多团队在其测试环境中同时使用密封后端和实时后端。

## Google 的 CI

现在让我们更详细地了解 Google 是如何实施 CI 的。首先,我们将了解 Google 绝大多数团队使用的全球持续构建(TAP),以及它如何实现我们在上一节中讨论的一些实践并解决一些挑战。我们还将介绍一款应用程序 Google Takeout,以及 CI 转型如何帮助它扩展为平台和服务。

## TAP:Google 的全球持续构建

亚当·本德

我们对整个代码库进行大规模持续构建,称为测试自动化平台 (TAP)。它负责运行我们大部分的自动化测试。

由于我们使用了 monorepo,TAP 成为了 Google 几乎所有变更的网关。它每天负责处理超过 50,000 个唯一变更并运行超过 40 亿个独立测试用例。

TAP 是 Google 开发基础设施的核心。从概念上讲,该过程非常简单。当工程师尝试提交代码时,TAP 会运行相关测试并报告成功或失败。如果测试通过,则允许将更改纳入代码库。

### 提交前优化

为了快速一致地发现问题,确保对每个更改进行测试非常重要。如果没有 CB,运行测试通常由个别工程师自行决定,这通常会导致少数积极主动的工程师尝试运行所有测试并跟踪失败情况。

如前所述,等待很长时间才能在提交前运行所有测试可能会造成严重破坏,有时甚至需要数小时。为了最大限度地减少等待时间,Google 的 CB 方法允许将可能具有破坏性的更改放入存储库 (请记住,这些更改会立即被公司其他人看到!)。我们只要求每个团队创建一个快速的测试子集,通常是项目的单元测试,可以在提交更改之前 (通常在其发送给代码审查之前) 运行 - 提交前。根据经验,通过提交前测试的更改通过其余测试的可能性非常高 (95% 以上),我们乐观地允许它被集成,以便其他工程师可以开始使用它。

提交更改后,我们使用 TAP 异步运行所有可能受影响的测试,包括更大、更慢的测试。

当更改导致 TAP 中的测试失败时,必须快速修复更改,以防止阻碍其他工程师。我们建立了一种文化规范,强烈反对在已知失败的测试之上提交任何新工作,尽管不稳定的测试会使这变得困难。因此,当提交的更改破坏了团队在 TAP 中的构建时,该更改可能会阻止团队取得进展或构建新版本。因此,快速处理破坏是必不可少的。

为了处理此类破坏,每个团队都配备了一名“构建警察”。构建警察的职责是确保其特定项目中的所有测试都通过,无论谁破坏了这些测试。当构建警察收到其项目中测试失败的通知时,他们会放下手头的工作并修复构建。这通常是通过识别有问题的更改并确定是否需要回滚 (首选解决方案) 或是否可以继续修复 (风险更大的提议) 来完成的。

在实践中,允许在验证所有测试之前提交更改的权衡确实得到了回报;提交更改的平均等待时间约为 11 分钟,通常在后台运行。结合 Build Cop 的纪律,我们能够高效地检测和解决由较长时间运行的测试检测到的中断,同时将干扰降到最低。

### 找到罪魁祸首

在 Google,我们在处理大型测试套件时面临的问题之一是找到导致测试失败的具体更改。从概念上讲,这应该非常简单:找到一个更改,运行测试,如果有测试失败,则将该更改标记为不好。不幸的是,由于不稳定性普遍存在以及测试基础设施本身偶尔出现问题,要确信失败是真实的并不容易。更复杂的是,TAP 每天必须评估如此多的更改(每秒超过一个),以至于它无法再对每个更改运行每个测试。相反,它会回退到将相关更改一起批处理,从而减少要运行的唯一测试的总数。虽然这种方法可以加快测试运行速度,但它会掩盖批次中的哪个更改导致测试失败。

为了加快故障识别速度,我们使用两种不同的方法。首先,TAP 会自动将失败的批次拆分为单独的更改,并针对每个更改单独重新运行测试。这个过程有时需要一段时间才能确定故障原因,因此,我们还创建了罪魁祸首查找工具,个人开发人员可以使用这些工具对一批更改进行二分搜索,并确定哪一个可能是罪魁祸首。

### 故障管理

在隔离重大变更后,尽快修复它很重要。失败的测试会很快开始削弱对测试套件的信心。如前所述,修复损坏的构建是 Build Cop 的责任。Build Cop 最有效的工具是回滚。

回滚更改通常是修复构建最快、最安全的方法,因为它可以快速将系统恢复到已知的良好状态。<sup>12</sup>事实上,TAP 最近已升级为在高度确信更改是罪魁祸首时自动回滚更改。

快速回滚与测试套件协同工作以确保持续的生产力。

测试让我们有信心去改变,回滚让我们有信心去撤销。没有测试,回滚就无法安全地完成。没有回滚,损坏的测试就无法快速修复,从而降低对系统的信心。

<sup>12</sup>只需两次点击即可撤销对 Google 代码库的任何更改!

### 资源限制

尽管工程师可以在本地运行测试,但大多数测试执行都在名为 Forge 的分布式构建和测试系统中进行。Forge 允许工程师在我们的数据中心运行他们的构建和测试,从而最大限度地提高并行性。在我们的规模下,运行工程师按需执行的所有测试以及作为 CB 流程的一部分运行的所有测试所需的资源是巨大的。即使考虑到我们拥有的计算资源量,像 Forge 和 TAP 这样的系统也受到资源限制。为了解决这些限制,从事 TAP 工作的工程师想出了一些巧妙的方法来确定哪些测试应该在什么时候运行,以确保花费最少的资源来验证给定的更改。

确定需要运行哪些测试的主要机制是分析每个变更的下游依赖关系图。Google 的分布式构建工具 Forge 和 Blaze 维护着全局依赖关系图的近乎实时的版本,并将其提供给 TAP。因此,TAP 可以快速确定哪些测试是任何变更的下游测试,并运行最小集以确保变更安全的。

影响使用 TAP 的另一个因素是测试运行的速度。TAP 通常能够比测试较多的更快地运行测试较少的变更。这种偏见鼓励工程师编写小而有针对性的变更。在繁忙的日子里,触发 100 个测试的变更和触发 1,000 个测试的变更之间的等待时间差异可能只有几十分钟。希望减少等待时间的工程师最终会进行较小、有针对性的变更,这对每个人来说都是有益的。

### CI 案例研究:Google Takeout Google

Takeout 最初于 2011 年推出,是一款数据备份和下载产品。其创始人率先提出了“数据解放”的理念,即用户应该能够以可用的格式轻松随身携带数据,无论他们走到哪里。他们首先将 Takeout 与一些 Google 产品集成,生成用户照片、联系人列表等的存档,供用户根据需要下载。然而,Takeout 并没有长期保持小规模,而是成长为一个平台和各种 Google 产品的一项服务。正如我们所看到的,有效的 CI 对于保持任何大型项目的健康发展至关重要,但在应用程序快速增长时尤其重要。

#### 场景 #1:持续中断的开发部署问题:随着

Takeout 成为 Google 范围内强大的数据获取、存档和下载工具,公司的其他团队开始转向它,请求 API,以便他们自己的应用程序也可以提供备份和下载功能,包括 Google Drive (文件夹下载由 Takeout 提供) 和 Gmail (用于 ZIP 文件预览)。总而言之,Takeout 从最初的 Google Takeout 产品的后端发展到为至少 10 种其他 Google 产品提供 API,提供广泛的功能。

该团队决定将每个新 API 部署为自定义实例,使用相同的原始 Takeout 二进制文件,但配置它们的工作方式略有不同。例如,Drive 批量下载的环境拥有最大的队列、为从 Drive API 获取文件保留的配额最多,以及一些自定义身份验证逻辑,以允许未登录用户下载公共文件夹。

不久之后,Takeout 就面临“标志问题”。为其中一个实例添加的标志会破坏其他实例,并且当服务器由于配置不兼容而无法启动时,它们的部署也会中断。除了功能配置之外,还有安全和 ACL 配置。例如,消费者 Drive 下载服务不应有权访问加密企业 Gmail 导出的密钥。配置很快变得复杂,导致几乎每晚都出现故障。

虽然已经做出了一些努力来理清和模块化配置,但这暴露出更大的问题是,当 Takeout 工程师想要更改代码时,手动测试每个服务器是否在每个配置下启动是不切实际的。他们直到第二天部署时才发现配置失败。在提交前和提交后(通过 TAP)运行了单元测试,但这些测试不足以发现这类问题。

团队采取了哪些措施。团队为每个在提交前运行的实例创建了临时的沙盒微型环境,并测试了所有服务器在启动时是否健康。在提交前运行临时环境可防止 95% 的服务器因配置不当而损坏,并将夜间部署失败率降低 50%。

虽然这些新的沙盒预提交测试大大减少了部署失败,但它们并没有完全消除部署失败。特别是,Takeout 的端到端测试仍然会频繁中断部署,并且这些测试很难在预提交上运行(因为它们使用测试帐户,这些帐户在某些方面仍然表现得像真实帐户,并受到相同的安全和隐私保护)。将它们重新设计为预提交友好型将是一项艰巨的任务。

如果团队无法在提交前运行端到端测试,那么什么时候可以运行它们呢?它希望比第二天的开发部署更快地获得端到端测试结果,并决定每两小时一次是一个很好的起点。但团队不想这么频繁地进行完整的开发部署。这会产生开销并破坏工程师在开发中测试的长期运行的流程。为这些测试创建新的共享测试环境似乎也需要太多的资源配置开销,此外,查找罪魁祸首(即查找导致失败的部署)可能会涉及一些不必要的手动工作。

因此,团队重用了提交前的沙盒环境,并轻松将其扩展到新的提交后环境。与提交前环境不同,提交后环境符合使用测试帐户的安全保障措施(首先,因为代码具有

已获得批准),因此可以在那里运行端到端测试。提交后的 CI 每两小时运行一次,从 green head 获取最新代码和配置,创建 RC,并针对它运行已在 dev 中运行的相同端到端测试套件。

吸取教训。更快的反馈循环可防止开发部署中出现问题:

- 将不同 Takeout 产品的测试从“夜间部署后”移至预提交,可防止 95% 的服务器因配置不当而损坏,并将夜间部署失败率降低 50%。 · 虽然端到端测试无法一直移动到预提交,但它们仍从“夜间部署后”移至“两小时内提交后”。这有效地将“罪魁祸首”减少了 12 倍。

场景 #2:难以辨认的测试日志问题:随

着 Takeout 整合了更多 Google 产品,它逐渐成长为一个成熟的平台,允许产品团队将带有产品特定数据获取代码的插件直接插入 Takeout 的二进制文件中。例如,Google Photos 插件知道如何获取照片、相册元数据等。Takeout 从最初的“少数”产品扩展到现在集成了 90 多种产品。

Takeout 的端到端测试将故障转储到日志中,这种方法无法扩展到 90 个产品插件。随着更多产品的集成,引入了更多故障。

尽管团队通过添加提交后 CI 更早、更频繁地运行测试,但多个失败仍然会堆积在内部,很容易被忽略。查看这些日志变成了令人沮丧的时间浪费,而且测试几乎总是失败。

团队做了什么。团队将测试重构为一个动态的、基于配置的套件(使用参数化的测试运行器)在更友好的用户界面中报告结果,清楚地显示单个测试结果为绿色或红色:无需再查阅日志。

他们还使故障调试变得更加容易,最值得注意的是,通过在错误消息中直接显示故障信息以及日志链接。例如,如果 Takeout 无法从 Gmail 获取文件,则测试将动态构建一个链接,在 Takeout 日志中搜索该文件的 ID,并将其包含在测试失败消息中。

这使得产品插件工程师的大部分调试过程自动化,并且不需要 Takeout 团队协助向他们发送日志,如图23-3 所示。

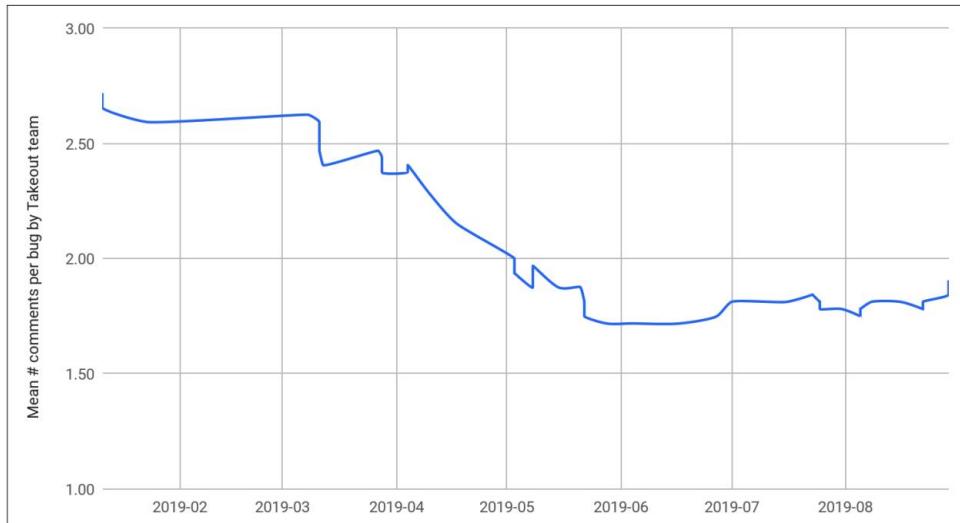


图 23-3。团队参与调试客户端故障

经验教训。CI 提供的可访问、可操作的反馈减少了测试失败并提高了生产率。这些举措将 Takeout 团队参与调试客户端（产品插件）测试失败的时间减少了 35%。

### 场景 3: 调试“所有 Google”

问题：Takeout CI 的一个有趣的副作用是，团队没有预料到，因为它以存档的形式验证了 90 多个面向最终用户的产品的输出，所以他们基本上是在测试“所有 Google”，并发现与 Takeout 无关的问题。这是一件好事。Takeout 能够帮助提高 Google 产品的整体质量。然而，这给他们的 CI 流程带来了一个问题：他们需要更好的故障隔离，以便确定哪些问题出在他们的构建中（哪些是少数），哪些问题出在他们调用的产品 API 背后的松散耦合的微服务中。

团队的做法。团队的解决方案是针对生产环境持续运行完全相同的测试套件，就像在提交后 CI 中所做的那样。这种方法实施起来成本低廉，并且允许团队隔离哪些故障是构建过程中新出现的，哪些故障是生产过程中出现的；例如，在“Google”的其他地方发布微服务的结果。

吸取的教训。针对生产环境和提交后的 CI（使用新构建的二进制文件，但使用相同的实时后端）运行相同的测试套件是一种隔离故障的廉价方法。

剩下的挑战。展望未来,随着 Takeout 与更多产品集成,以及这些产品变得越来越复杂,测试“Google 的所有产品”的负担 (显然,这是夸张的说法,因为大多数产品问题都是由各自的团队发现的)将越来越重。此 CI 与产品之间的手动比较会浪费 Build Cop 的时间。

未来的改进。这提供了一个有趣的机会,可以在 Takeout 的提交后 CI 中尝试使用记录/重放进行密封测试。理论上,这将消除 Takeout CI 中出现的后端产品 API 故障,这将使套件更稳定,更有效地捕获 Takeout 更改的最后两个小时内的故障 - 这是它的预期目的。

#### 场景 #4:保持绿色问题:由于平台

支持更多产品插件,每个插件都包含端到端测试,这些测试会失败,端到端测试套件几乎总是会损坏。故障无法全部立即修复。许多故障是由于产品插件二进制文件中的 bug 造成的,而 Takeout 团队无法控制这些 bug。有些故障比其他故障更重要——低优先级 bug 和测试代码中的 bug 不需要阻止发布,而高优先级 bug 则需要。团队可以通过注释掉测试来轻松禁用测试,但这会使故障很容易被遗忘。

一个常见的失败原因:产品插件推出新功能时,测试可能会中断。例如,YouTube 插件的播放列表抓取功能可能会在开发版中启用几个月,然后才在生产版中启用。

Takeout 测试只知道要检查一个结果,因此通常需要在特定环境中禁用该测试,并在功能推出时手动进行管理。

团队的做法。团队想出了一个策略性方法来禁用失败的测试,即给失败的测试贴上相关错误标签,并将其提交给负责的团队 (通常是产品插件团队)。当失败的测试被贴上错误标签时,团队的测试框架将抑制其失败。这使得测试套件保持绿色,并且仍然确保除了已知问题之外的所有其他测试都通过了,如图23-4 所示。

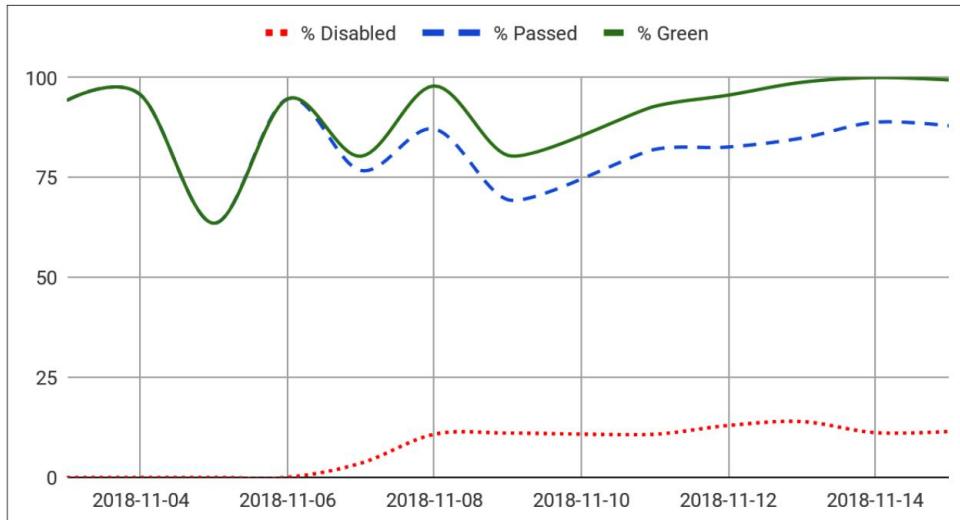


图 23-4. 通过（不负责任的）测试禁用实现绿色

对于推出问题，团队为插件工程师添加了功能，让他们可以指定功能标志的名称或代码更改的 ID，从而启用特定功能以及启用和不启用该功能时预期的输出。测试可以查询测试环境以确定给定功能是否已启用，并相应地验证预期输出。

当禁用测试的 bug 标签开始累积且未更新时，团队会自动进行清理。测试现在将通过查询 bug 系统的 API 来检查 bug 是否已关闭。如果标记失败的测试实际上通过了，并且通过时间超过了配置的时间限制，则测试将提示清理标签（并将 bug 标记为已修复，如果尚未修复）。此策略有一个例外：不稳定的测试。对于这些测试，团队将允许将测试标记为不稳定，并且如果测试通过，系统不会提示清理标记为“不稳定”的失败。

这些变化构成了一个基本上可自我维护的测试套件，如图23-5 所示。

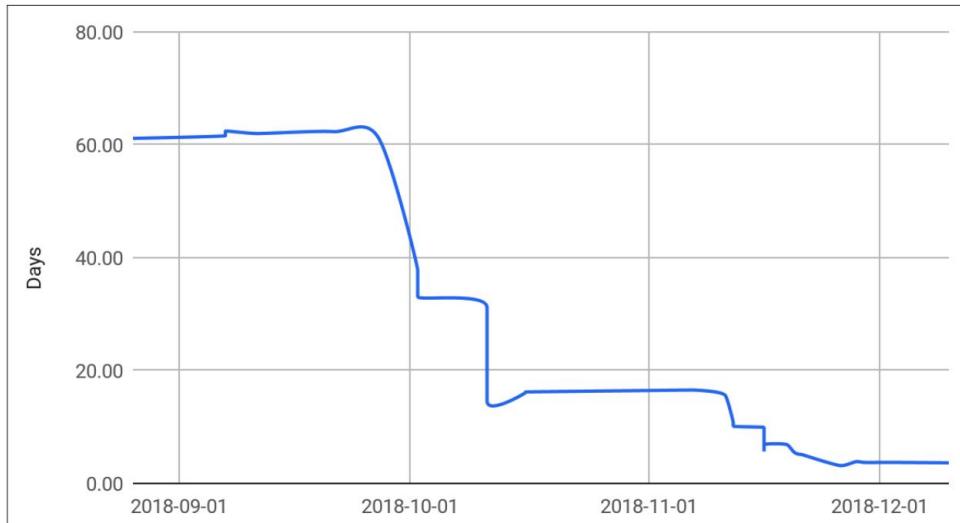


图 23-5. 关闭 bug 的平均时间,aer x 已提交

经验教训。禁用无法立即修复的失败测试是保持套件绿色的实用方法,这让您确信自己了解所有测试失败。此外,自动化测试套件的维护(包括推出管理和更新已修复测试的跟踪错误)可保持套件清洁并防止技术债务。在 DevOps 术语中,我们可以将图 23-5 中的指标称为 MTTCU:平均清理时间。

未来的改进。自动化归档和标记错误将是下一步的有益举措。这仍然是一个手动且繁琐的过程。如前所述,我们的一些大型团队已经这样做了。

进一步的挑战。我们描述的场景远非 Takeout 面临的唯一 CI 挑战,还有更多问题需要解决。例如,我们在第 490 页的“CI 挑战”中提到了将故障与上游服务隔离的难度。这是 Takeout 仍然面临的问题,因为上游服务偶尔会出现故障,例如,当 Takeout 的“Drive 文件夹下载”API 使用的流媒体基础设施中的安全更新在部署到生产环境时破坏了存档解密。上游服务是自行准备和测试的,但没有简单的方法可以在它们投入生产后自动使用 CI 检查它们是否与 Takeout 兼容。最初的解决方案是创建一个“上游准备”CI 环境,以针对其上游依赖项的准备版本测试生产 Takeout 二进制文件。然而,事实证明这很难维护,并且准备版本和生产版本之间存在额外的兼容性问题。

## 但我负担不起 CI

您可能会认为这一切都很好,但您既没有时间也没有钱来构建这一切。我们当然承认,Google 可能比典型的初创公司拥有更多的资源来实施 CI。然而,我们的许多产品发展如此之快,以至于他们也没有时间开发 CI 系统 (至少不是一个合适的系统)。

在您自己的产品和组织中,尝试考虑您为在生产中发现和处理的问题已经付出的成本。当然,这些问题会对最终用户或客户产生负面影响,但也会影响团队。频繁的生产救火工作令人紧张和沮丧。尽管构建 CI 系统的成本很高,但它不一定是新的成本,而是将成本转移到更早的 (也是更可取的)阶段,从而降低了问题发生得太靠右的发生率,从而降低了成本。CI 会带来更稳定的产品和更快乐的开发者文化,工程师对“系统”会发现问题更有信心,他们可以将更多精力放在功能上而不是修复上。

## 结论

尽管我们已经描述了我们的 CI 流程以及我们实现自动化的一些方法,但这并不意味着我们已经开发出了完美的 CI 系统。毕竟,CI 系统本身只是软件,永远不会完整,应该根据应用程序和工程师不断变化的需求进行调整。我们试图通过 Takeout CI 的发展和我们指出的未来改进领域来说明这一点。

## TL;DR

- CI 系统决定何时使用哪些测试。· 随着代码库的老化和规模不断扩大。
- CI 应优化提交前更快、更可靠的测试,以及提交后更慢、更不确定的测试。· 可访问、可操作的反馈使 CI 系统变得更加高效。



## 第 24 章

# 持续交付

作者:Radha Narayan、Bobbi Jones、  
Sheri Shipe 和 David Owens  
由 Lisa Carey 编辑

鉴于技术格局瞬息万变,任何产品的竞争优势都在于其快速进入市场的能力。组织的速度是其与其他参与者竞争、保持产品和服务质量或适应新法规的关键因素。部署时间是速度的瓶颈。部署不会只在首次发布时发生一次。

教育界有句俗语:没有哪套教案能在第一次与学生接触后就立于不败之地。同样,没有哪款软件在第一次发布时是完美的,唯一可以保证的就是你必须更新它。而且要快。

软件产品的长期生命周期包括快速探索新想法、快速响应环境变化或用户问题,以及大规模提高开发人员的速度。从 Eric Raymond 的《大教堂与集市》到 Eric Reis 的《精益创业》,任何组织长期成功的关键始终在于能够尽快将想法付诸实施并交到用户手中,并快速响应他们的反馈。Martin Fowler 在他的《持续交付》一书中写道(又名 CD)指出:“任何软件工作的最大风险在于你最终构建出的东西没有用。你越早、越频繁地将可用的软件呈现给真正的用户,你就能越快得到反馈,从而了解它的真正价值。”

在交付用户价值之前,长时间处于开发状态的工作风险高、成本高,甚至会打击士气。在 Google,我们努力尽早发布,或“发布并迭代”,以便团队能够快速看到其工作的影响,并更快地适应不断变化的市场。代码的价值不是在提交时实现的,而是在功能可供用户使用时实现的。减少

“代码完成”和用户反馈之间的时间最大限度地减少了正在进行的工作的成本。

如果你意识到发射永远不会成功,而是开始了一个学习周期,在这个周期中,你会得到非凡的结果,然后你会修复下一个最重要的问题,衡量它的进展情况,修复下一个问题等等 它永远不会完成。

David Weekly,前谷歌产品经理

在 Google,本书中介绍的实践可让数百名(有时甚至数千名)工程师快速解决问题,独立开发新功能而不必担心发布,并通过 A/B 实验了解新功能的有效性。本章重点介绍快速创新的关键杠杆,包括管理风险、大规模提高开发人员的速度以及了解您发布的每项功能的成本和价值权衡。

## Google 持续交付的惯用语

持续交付 (CD) 和敏捷方法的核心原则是,随着时间的推移,小批量的变更会带来更高的质量;换句话说,越快越安全。乍一看,这对团队来说似乎极具争议,特别是如果设置 CD 的先决条件(例如持续集成(CI)和测试)尚未到位。由于所有团队可能需要一段时间才能实现 CD 的理想,因此我们专注于开发在实现最终目标的过程中独立交付价值的各个方面。以下是其中一些:

敏捷

频繁小批量发布

自动化

减少或消除频繁发布的重复开销

隔离

努力实现模块化架构,以隔离变更并使故障排除更容易

可靠性

测量崩溃或延迟等关键健康指标并不断改进

数据驱动的决策

对健康指标进行 A/B 测试以确保质量

分阶段推出

在向所有人发布更改之前,先向少数用户推出更改

一开始,频繁发布新版本的软件似乎有风险。随着用户群的增长,你可能会担心如果测试中没有发现任何错误,用户会愤怒地反对,而且你可能只是在开发中加入了太多新代码。

您的产品需要进行全面测试。但这正是 CD 可以提供帮助的地方。理想情况下，一个版本与下一个版本之间的变化很少，因此解决问题是微不足道的。在极限情况下，使用 CD，每个更改都会经过 QA 管道并自动部署到生产中。对于许多团队来说，这通常不切实际，因此通常会将文化变革工作转向 CD 作为中间步骤，在此期间，团队可以在不实际这样做的情况下随时做好部署准备，从而建立起他们将来更频繁发布的信心。

## 速度是一项团队运动：如何将部署分解为可管理的部分

当团队规模较小时，代码库中的变更会以一定的速率出现。随着团队规模的扩大或分裂为子团队，我们看到一种反模式出现：子团队会分支代码以避免影响他人，但随后却在集成和罪魁祸首查找方面遇到困难。在 Google，我们更希望团队继续在共享代码库中开发，并设置 CI 测试、自动回滚和罪魁祸首查找，以快速识别问题。[第 23 章将详细讨论这一点。](#)

我们的代码库之一 YouTube 是一个庞大的单片 Python 应用程序。发布过程非常繁琐，需要 Build Cops、发布经理和其他志愿者的参与。几乎每个版本都有多个精心挑选的更改和重新发布。每个版本都有一个由远程 QA 团队运行的 50 小时手动回归测试周期。当发布的运营成本如此之高时，就会开始出现一个周期，在这个周期中，您要等到能够对其进行更多测试后才能发布版本。与此同时，有人想添加一个几乎准备好的功能，很快您就会面临一个繁琐、容易出错且缓慢的发布过程。最糟糕的是，上次发布版本的专家已经精疲力竭并离开了团队，现在甚至没有人知道如何排除您尝试发布更新时发生的那些奇怪崩溃，让您一想到按下按钮就会感到恐慌。

如果您的发布成本高昂且有时存在风险，那么本能的做法是放慢发布节奏并延长稳定期。然而，这只能带来短期的稳定性提升，随着时间的推移，它会降低速度并让团队和用户感到沮丧。

答案是降低成本、加强纪律、降低风险，但关键是要抵制明显的运营修复，并投资于长期的架构变更。这个问题的明显运营修复导致了一些传统方法：恢复到几乎没有学习或迭代空间的传统规划模型，在开发过程中增加更多的治理和监督，并实施风险审查或奖励低风险（通常是低价值）的功能。

不过,回报最高的投资是迁移到微服务架构,这可以使大型产品团队有能力保持活力和创新,同时降低风险。在某些情况下,在谷歌,答案是从头开始重写应用程序,而不是简单地迁移它,将所需的模块化建立到新的架构中。虽然这两种选择都可能需要数月时间,而且短期内可能会很痛苦,但在运营成本和认知简单性方面获得的价值将在应用程序的多年生命周期内得到回报。

## 评估隔离中的变化:旗帜保护功能

可靠的持续发布的关键是确保工程师“标记保护”所有更改。随着产品的发展,二进制文件中将共存于处于不同开发阶段的多个功能。标记保护可用于逐个功能地控制产品中功能代码的包含或表达,并且可以针对发布和开发版本以不同的方式表达。如果语言允许,则为某个版本禁用的功能标记应允许构建工具从版本中删除该功能。例如,已经发送给客户的稳定功能可能同时针对开发和发布版本启用。正在开发的功能可能仅针对开发启用,以保护用户免受未完成功能的影响。新功能代码与旧代码路径一起存在于二进制文件中 - 两者都可以运行,但新代码受标记保护。如果新代码正常运行,您可以删除旧代码路径并在后续版本中完全启动该功能。如果出现问题,可以通过动态配置更新独立于二进制文件版本更新标记值。

在以前的二进制发布时代,我们必须将新闻稿与二进制推出时间紧密结合起来。我们必须先成功推出新功能或新功能,然后才能发布新闻稿。这意味着该功能在宣布之前就已经公开,提前被发现的风险非常大。

这就是标志保护的美妙之处。如果新代码带有标志,则可以在新闻发布之前立即更新标志以启用您的功能,从而最大限度地降低功能泄露的风险。请注意,标志保护的代码并不是真正敏感功能的完美安全网。如果代码没有很好地混淆,仍然可能被抓取和分析,并且并非所有功能都可以在不增加复杂性的情况下隐藏在标志后面。此外,即使是标志配置更改也必须小心推出。一次性为 100% 的用户启用标志并不是一个好主意,因此管理安全配置推出的配置服务是一项很好的投资。尽管如此,控制级别和将特定功能的命运与整体产品发布分离的能力是应用程序长期可持续性的有力杠杆。

# 追求敏捷:建立发布列车

Google 的搜索二进制文件是其第一个也是最古老的。它的代码库庞大而复杂,可以追溯到 Google 的起源 通过我们的代码库搜索仍然可以找到至少早在 2003 年甚至更早编写的代码。当智能手机开始流行时,一个又一个的移动功能被塞进了主要为服务器部署编写的代码堆中。尽管搜索体验变得更加生动和互动,但部署可行的版本变得越来越困难。有一段时间,我们每周只将搜索二进制文件发布到生产环境中一次,甚至连达到这个目标都很难,而且往往取决于运气。

当我们的一位贡献作者 Sheri Shipe 接手提高搜索发布速度的项目时,每个发布周期都需要一组工程师花上几天的时间才能完成。他们构建二进制文件、集成数据,然后开始测试。每个错误都必须手动分类,以确保它不会影响搜索质量、用户体验 (UX) 和/或收入。这个过程非常繁琐且耗时,并且无法根据变化量或变化率进行扩展。因此,开发人员永远无法知道他们的功能何时会投入生产。这使得新闻稿和公开发布的时机变得具有挑战性。

发布不是凭空而来的,可靠的发布使相关因素更容易同步。在几年的时间里,一群敬业的工程师实施了一个持续发布流程,简化了将搜索二进制文件发布到世界的所有过程。我们尽可能地实现自动化,设置提交功能的截止日期,并简化插件和数据与二进制文件的集成。我们现在可以每隔一天持续将一个新的搜索二进制文件发布到生产中。

为了在发布周期中实现可预测性,我们做出了哪些权衡?它们归结为我们融入系统的两个主要想法。

## 没有完美的二进制文件首

先,没有完美的二进制文件,尤其是对于包含数十或数百名开发人员独立开发数十个主要功能的构建。尽管不可能修复所有错误,但我们始终需要权衡以下问题:如果某行向左移动了两个像素,是否会影响广告显示和潜在收入?如果方框的阴影略有改变会怎样?

这会让视障用户难以阅读文本吗?本书的其余部分可以说是关于尽量减少发布时意外结果的集合,但最终我们必须承认软件从根本上来说是复杂的。没有完美的二元性 每次将新更改发布到生产中时都必须做出决策和权衡。具有明确阈值的关键绩效指标指标允许

即使功能并不完美,也可以发布,并且可以为有争议的发布决策提供清晰度。

一个错误涉及一种罕见的方言,这种方言只在菲律宾的一个岛上使用。如果用户用这种方言提出搜索问题,他们得到的不是问题的答案,而是空白网页。我们必须确定修复这个错误的成本是否值得推迟发布一项重要的新功能。

我们跑遍了各个办公室,试图确定到底有多少人会说这种语言,是否每次用户用这种语言搜索时都会这样,以及这些人是否经常使用 Google。我们采访过的每个质量工程师都把问题推给了更资深的人。最后,在掌握了数据后,我们向 Google 搜索的高级副总裁提出了这个问题。我们是否应该推迟发布一个关键版本来修复一个只影响菲律宾一个小岛的错误?事实证明,无论您的岛屿有多小,您都应该获得可靠而准确的搜索结果:我们推迟了发布并修复了错误。

## 满足发布期限

第二个想法是如果你错过了发布列车,它就会在没有你的情况下离开。

俗话说,“截止日期是确定的,但生活却不是。”在发布时间表的某个时刻,你必须下定决心,拒绝开发人员及其新功能。一般来说,在截止日期过后,无论你如何恳求或乞求,都无法将新功能纳入今天的发布版本。

也有罕见的例外。情况通常是这样的。星期五晚上很晚的时候,六名软件工程师惊慌失措地冲进发布经理的办公室。

他们与 NBA 签订了合同,刚刚完成了该功能。但它必须在明天的大赛之前上线。发布必须停止,我们必须挑选该功能放入二进制文件中,否则我们将违反合同!一位睡眼惺忪的发布工程师摇摇头,说剪切和测试新的二进制文件需要四个小时。今天是他们孩子的生日,他们还需要去拿气球。

定期发布意味着,如果开发人员错过了发布列车,他们可以在几小时而不是几天内赶上下一班列车。这限制了开发人员的恐慌,并大大改善了发布工程师的工作与生活平衡。

---

<sup>1</sup>记住 SRE “错误预算”公式:完美很少是最好的目标。了解允许的误差空间以及最近花费了多少预算,并以此来调整速度和稳定性之间的权衡。

# 质量和用户关注:只发布有用的东西

膨胀是大多数软件开发生命周期中令人遗憾的副作用,产品越成功,其代码库通常就越膨胀。

快速、高效的发布流程的一个缺点是,这种臃肿往往会被放大,并可能给产品团队甚至用户带来挑战。特别是如果软件交付给客户,比如移动应用程序,这可能意味着用户的设备要为空间、下载和数据成本付出代价,即使这些功能他们从未使用过,而开发人员则要为较慢的构建、复杂的部署和罕见的错误付出代价。在本节中,我们将讨论动态部署如何允许您只发布使用过的内容,从而在用户价值和功能成本之间做出必要的权衡。在 Google,这通常意味着要配备专门的团队来持续提高产品的效率。

虽然有些产品是基于 Web 并在云端运行,但许多都是客户端应用程序,它们使用用户设备(手机或平板电脑)上的共享资源。这种选择本身就展示了原生应用程序之间的权衡,原生应用程序可能性能更高、更能适应不稳定的连接,但也更难更新,更容易受到平台级问题的影响。反对频繁、持续部署原生应用程序的一个常见论点是,用户不喜欢频繁更新,必须为数据成本和中断付费。可能还有其他限制因素,例如访问网络或限制更新所需的重启次数。

尽管在产品更新频率方面需要权衡,但目标是让这些选择是有意的。通过顺畅、运行良好的 CD 流程,可行版本的创建频率可以与用户接收版本的频率区分开来。您可能无需实际执行即可实现每周、每天或每小时部署的目标,并且您应该根据用户的特定需求和更大的组织目标有意选择发布流程,并确定最能支持产品长期可持续性的人员配备和工具模型。

在本章前面,我们讨论了如何保持代码模块化。这样可以实现动态、可配置的部署,从而更好地利用受限资源,例如用户设备上的空间。如果不这样做,每个用户都必须收到他们永远不会使用的代码,以支持他们不需要的翻译或适用于其他类型设备的架构。动态部署允许应用保持较小的规模,同时只将代码发送到能给用户带来价值的设备,而 A/B 实验允许在功能成本与其对用户和您的业务的价值之间进行有意的权衡。

建立这些流程需要前期成本,而识别和消除导致发布频率低于预期的摩擦是一项艰苦的工作

流程。但从风险管理、开发人员速度和快速创新等方面来看，长期收益是如此之高，以至于这些初始成本是值得的。

## 向左移动：尽早做出数据驱动的决策

如果您是为所有用户开发，那么您可能在智能屏幕、扬声器或 Android 和 iOS 手机和平板电脑上拥有客户端，并且您的软件可能足够灵活，允许用户自定义他们的体验。即使您只为 Android 设备开发，超过 20 亿台 Android 设备的多样性也可能让您难以确定发布版本。随着创新的步伐，当有人读到本章时，全新的设备类别可能已经出现。

我们的一位发布经理分享了一条扭转局面的智慧，他说，我们的客户市场多样性不是问题，而是事实。接受这一点后，我们可以通过以下方式改变我们的发布资格模型：

- 如果全面测试实际上不可行，则以代表性测试为目标  
反而。
- 分阶段逐步增加用户群的百分比，以便快速  
修复。
- 自动化的 A/B 发布可以提供具有统计意义的结果来证明发布的质量，而无需疲惫的人工查看仪表板并做出决策。

在开发 Android 客户端时，Google 应用会使用专门的测试轨道，并分阶段向越来越多的用户流量发布，仔细监控这些渠道中的问题。由于 Play Store 提供无限的测试轨道，我们还可以在计划发布的每个国家/地区设立 QA 团队，从而实现全球范围内一夜之间完成关键功能的测试。

我们在部署 Android 时注意到的一个问题是，只需推送更新，用户指标就会发生统计上显著的变化。这意味着，即使我们不对产品进行任何更改，推送更新也可能以难以预测的方式影响设备和用户行为。因此，虽然将更新发布到一小部分用户流量中可以让我们了解崩溃或稳定性问题，但它几乎无法告诉我们新版本的应用是否真的比旧版本更好。

Dan Siroker 和 Pete Koomen 已经讨论过对功能进行 A/B 测试<sup>2</sup>的价值,但在 Google,我们的一些大型应用程序也会对其部署进行 A/B 测试。这意味着发送两个版本的产品:一个是所需更新,基线是安慰剂(旧版本会再次发送)。当两个版本同时向足够大的相似用户群推出时,您可以将一个版本与另一个版本进行比较,以查看软件的最新版本是否确实比前一个版本有所改进。有了足够大的用户群,您应该能够在几天甚至几小时内获得具有统计意义的结果。一旦有足够的数据确保保护栏指标不会受到影响,自动化指标管道就可以将版本推送到更多的流量,从而实现最快的发布。

显然,这种方法并不适用于所有应用,而且如果用户群不够大,则会产生很大的开销。在这些情况下,建议的最佳做法是争取发布中性变更版本。所有新功能都经过标记保护,因此在推出期间测试的唯一变更是部署本身的稳定性。

## 改变团队文化:建立纪律部署

尽管“始终部署”有助于解决影响开发人员速度的几个问题,但也有一些做法可以解决规模问题。最初发布产品的团队可能不到 10 人,每个人轮流负责部署和生产监控。随着时间的推移,你的团队可能会发展到数百人,子团队负责特定的功能。随着这种情况的发生和组织规模的扩大,每次部署的更改数量和每次发布尝试的风险量都会超线性增加。每次发布都包含数月的汗水和泪水。成功发布成为一项高接触和劳动密集型的工作。开发人员经常会被困在试图决定哪个更糟糕:放弃包含一个季度的新功能和错误修复的版本,还是推出对其质量没有信心的版本。

在规模上,复杂性的增加通常表现为发布延迟的增加。即使你每天都发布,一个版本也可能需要一周或更长时间才能完全安全地推出,当你尝试调试任何问题时,你就会落后一周。这就是“始终部署”可以让开发项目恢复有效形式的地方。频繁的发布列车可以将与已知良好位置的偏差降到最低,最近发布

---

<sup>2</sup> Dan Siroker 和 Pete Koomen,《A/B 测试:将点击转化为客户的最有效方法》(霍博肯:威利,2013 年)。

变更有助于解决问题。但是,团队如何确保庞大且快速扩展的代码库所固有的复杂性不会拖累进度呢?

在 Google 地图上,我们认为功能非常重要,但很少有功能重要到需要推迟发布。如果发布频率很高,那么功能错过发布所带来的痛苦与发布中的所有新功能因延迟而带来的痛苦相比就小得多,尤其是如果匆忙将尚未准备好的功能纳入其中,用户可能会感受到痛苦。

一项发布责任是保护产品不被开发人员侵害。

在权衡利弊时,开发人员对推出新功能的热情和紧迫感绝不能凌驾于现有产品的用户体验之上。这意味着,新功能必须通过具有强大契约的接口与其他组件隔离,关注点分离,严格测试,尽早并经常沟通,以及新功能接受惯例。

## 结论

多年来,在我们所有的软件产品中,我们发现,与直觉相反,更快更安全。产品的健康状况和开发速度实际上并不相互对立,更频繁、小批量发布的产品质量更好。它们可以更快地适应在野外遇到的 bug 和意想不到的市场变化。不仅如此,更快更便宜,因为拥有可预测的、频繁的发布序列会迫使您降低每次发布的成本,并使任何放弃发布的成本非常低。

即使您实际上并没有将这些版本推送给用户,只需拥有能够实现持续部署的结构就可以产生大部分价值。

我们的意思是什么?我们实际上不会每天发布截然不同的搜索、地图或 YouTube 版本,但要做到这一点,需要一个强大、记录良好的持续部署流程、准确且实时的用户满意度和产品健康状况指标,以及一个协调的团队,该团队对哪些功能可以加入或退出以及原因有明确的政策。在实践中,要做到这一点,通常还需要可以在生产中配置的二进制文件、像代码一样管理的配置(在版本控制中),以及允许安全措施(如试运行验证、回滚/前滚机制和可靠的修补)的工具链。

## TL;DR

- 速度是一项团队运动:协作开发代码的大型团队的最佳工作流程需要架构的模块化和近乎持续的集成。

- 评估隔离变化:标记保护任何功能,以便能够隔离问题  
lems早早。
- 以现实为基准:使用分阶段推出来解决设备多样性和用户群广度的问题。在与生产  
环境不相似的合成环境中进行发布资格认证可能会导致后期出现意外。
- 只发布需要使用的功能:监控任何功能的成本和价值,以  
知道它是否仍然具有相关性并且可以提供足够的用户价值。
- 持续:通过 CI 和持续部署,更早地对所有变更做出更快、更数据驱动的决策。· 更快更安全:尽早、频繁、小批  
量地发货,以降低每次发布的风险并最大限度地缩短

上市时间。



## 第 25 章

# 计算即服务

作者:Onufry Wojtaszczyk  
由 Lisa Carey 编辑

我不是试图去理解计算机,而是试图去理解程序。

芭芭拉·利斯科夫

在完成编写代码的艰苦工作后,您需要一些硬件来运行它。因此,您需要购买或租用该硬件。这本质上是计算即服务 (CaaS),其中“计算”是实际运行程序所需的计算能力的简称。

本章将介绍如何将这个简单的概念 (只需给我硬件来运行我的东西<sup>1</sup>)映射到一个可以随着组织的发展和成长而生存和扩展的系统。由于主题复杂,因此它有点长,分为四个部分:

第 518 页的“驯服计算环境”介绍了 Google 如何找到此问题的解决方案,并解释了 CaaS 的一些关键概念。<sup>·第 523 页</sup>  
的“为托管计算编写软件”展示了托管计算解决方案如何影响工程师编写软件的方式。我们认为,“牛,而不是宠  
物”/灵活的调度模型是 Google 在过去 15 年中取得成功的基础,也是软件工程师工具箱中的重要工具。

---

<sup>1</sup>免责声明:对于某些应用程序,“运行它的硬件”是客户的硬件 (例如,对于  
例如,十年前买的一款包装精装游戏)。这带来了非常不同的挑战,我们在本章中不会涉及。

- 第 530 页的“随着时间而变化的 CaaS”深入探讨了 Google 学到的一些经验教训,即随着组织的成长和发展,计算架构的各种选择如何发挥作用。
- 最后,第 535 页的“选择计算服务”主要针对那些将决定在其组织中使用什么计算服务的工程师。

## 驯服计算环境

Google 的内部 Borg 系统<sup>2</sup>是当今许多 CaaS 架构 (如 Kubernetes 或 Mesos) 的前身。为了更好地理解此类服务的具体方面如何满足不断发展的组织的需求,我们将追溯 Borg 的演变以及 Google 工程师为驯服计算环境所做的努力。

### 劳动自动化

想象一下你是世纪之交的一名大学生。如果你想部署一些新的、漂亮的代码,你会通过 SFTP 将代码上传到大学计算机实验室的一台机器上,通过 SSH 进入这台机器,编译并运行代码。这个解决方案很简单,很诱人,但随着时间的推移和规模扩大,它会遇到相当大的问题。但是,因为许多项目都是从这个系统开始的,所以多个组织最终采用了一些流程,这些流程是该系统的简化演变,至少对于某些任务而言是这样——机器数量在增加 (因此你通过 SFTP 和 SSH 进入其中的许多机器),但底层技术仍然存在。例如,2002 年,Google 最资深的工程师之一 Jeff Dean 写了以下内容,介绍在发布过程中运行自动数据处理任务:

[运行任务] 是一项耗时的后勤噩梦。它目前需要获取 50 多台机器的列表,在每台机器上启动一个进程,并在每台机器上监控其进度。如果其中一台机器死机,则不支持自动将计算迁移到另一台机器,并且以临时方式监控作业的进度 [...] 此外,由于进程可能相互干扰,因此有一个复杂的、人为实现的“注册”文件来限制机器的使用,这会导致调度不够理想,并加剧对稀缺机器资源的争用。这是 Google 努力驯服计算环境的早期触发因素,这很好地解释了为什么简单的解决方案在更大规模下变得无法维护。

---

<sup>2</sup> Abhishek Verma,Luis Pedrosa, Madhukar R Korupolu,David Oppenheimer,Eric Tune 和 John Wilkes, “Google 与 Borg 的大规模集群管理”, EuroSys, 文章编号:18 (2015 年 4 月) :1-17。

## 简单的自动化组织

可以做一些简单的事情来减轻一些痛苦。将二进制文件部署到 50 多台机器上并在那里启动它的过程可以通过 shell 脚本轻松实现自动化，然后 如果这是一个可重用的解决方案 通过更容易维护的语言编写的更强大的代码来并行执行部署（特别是因为“50 多台”机器可能会随着时间的推移而增长）。

更有趣的是，每台机器的监控也可以自动化。最初，负责该流程的人希望知道（并能够干预）其中一个副本是否出了问题。这意味着从流程中导出一些监控指标（如“流程处于活动状态”和“处理的文档数量”）通过将其写入共享存储，或调用监控服务，他们可以一目了然地看到异常情况。例如，该领域当前的开源解决方案是在 Graphana 或 Prometheus 等监控工具中设置仪表板。

如果检测到异常，通常的缓解策略是通过 SSH 进入机器，终止进程（如果进程仍然存在），然后重新启动。这很繁琐，可能容易出错（请确保连接到正确的机器，并确保终止正确的进程），但可以自动化：

- 无需手动监控故障，可以使用机器上的代理来检测异常（例如“该进程在过去五分钟内没有报告其处于活动状态”或“该进程在过去十分钟内没有处理任何文档”），并在检测到异常时终止该进程。
- 无需在死亡后登录机器重新启动该过程，将整个执行包装在“while true; do run && break; done” shell 脚本中就足够了。

云世界的等价物是设置一个自动修复策略（在健康检查失败后终止并重新创建虚拟机或容器）。

这些相对简单的改进解决了 Jeff Dean 之前描述的部分问题，但不是全部；人为实现的节流和转移到新机器需要更复杂的解决方案。

## 自动调度下一步自然是

实现机器分配的自动化。这需要第一个真正的“服务”，最终将发展成为“计算即服务”。也就是说，为了实现调度自动化，我们需要一个中央服务，它知道可用的机器的完整列表，并可以根据需求选择一些空闲的机器，自动将二进制文件部署到这些机器上。这样就不需要手动

维护“注册”文件，而将机器列表的维护委托给计算机。该系统让人强烈地联想到早期的分时架构。

这个想法的自然延伸是将这种调度与对机器故障的反应结合起来。通过扫描机器日志以查找表示健康状况不佳的表达式（例如，大量磁盘读取错误），我们可以识别出损坏的机器，向人类发出需要修理这些机器的信号，并避免在此期间将任何工作安排到这些机器上。为了进一步消除劳累，自动化可以在涉及人类之前先尝试一些修复，例如重新启动机器，希望任何错误都会消失，或者运行自动磁盘

扫描。

Jeff 的引言中还有最后一个抱怨，即如果计算所在的机器坏了，就需要人工将计算迁移到另一台机器上。这里的解决方案很简单：因为我们已经有了调度自动化和检测机器故障的能力，所以我们只需让调度程序分配一台新机器，并在这台新机器上重新启动工作，放弃旧机器。执行此操作的信号可能来自机器自省守护程序或来自对单个进程的监控。

所有这些改进都系统地应对了组织规模的不断增长。当集群只有一台机器时，SFTP 和 SSH 是完美的解决方案，但在数百或数千台机器的规模下，自动化需要接管。

我们开始引用的是 2002 年“Global WorkQueue”的一份设计文档，这是 Google 早期针对某些工作负载的 CaaS 内部解决方案。

容器化和多租户到目前为止，我们隐式地假设了机器和机

器上运行的程序之间存在一对一的映射。从计算资源（RAM、CPU）消耗的角度来看，这在很多方面都是非常低效的：

- 很可能存在比机器类型（具有不同的资源可用性）更多不同类型的作业（具有不同的资源需求），因此许多作业需要使用相同的机器类型（需要为其中最大的作业提供相同的机器类型）。
- 机器需要很长时间才能部署，而程序资源需求会随着时间的推移而增长。如果获取新的、更大的机器需要花费您的组织数月时间，您还需要使它们足够大以适应预期的增长

现有资源需求超过配置新机器所需的时间,这会导致浪费,因为新机器没有得到充分利用。3 · 即使新机器到达,您仍然有旧机器(扔掉它们可能会造成浪费),因此您必须管理不能自行适应您需求的异构机器群。

自然的解决方案是指定每个程序的资源需求(CPU、RAM、磁盘空间),然后要求调度程序将程序的副本打包到可用的机器池中。

邻居家的狗在我的内存里叫如果每个人都

能很好地配合,上述解决方案会非常有效。但是,如果我在配置中指定数据处理管道的每个副本将消耗一个CPU和200 MB的内存,然后由于错误或有机增长,它开始消耗更多,那么调度到的机器将耗尽资源。

对于CPU来说,这将导致相邻的服务作业出现延迟问题;对于RAM来说,这将导致内核因内存不足而终止运行,或者由于磁盘交换而导致可怕的延迟。4

同一台计算机上的两个程序在其他方面也可能出现不良交互。许多程序都希望在一台机器上安装某些特定版本的依赖项      这些依赖项可能会与其他程序的版本要求相冲突。

程序可能希望某些系统范围的资源(例如/tmp)可供其自己独用。安全性是一个问题      程序可能正在处理敏感数据,需要确保同一台计算机上的其他程序无法访问

访问它。

因此,多租户计算服务必须提供一定程度的隔离,某种程度的保证,即进程能够安全进行而不受机器上其他租户的干扰。

隔离的一个经典解决方案是使用虚拟机(VM)。然而,这些虚拟机在资源使用方面(它们需要资源来运行内部的完整操作系统)和启动时间(同样,它们需要启动完整的操作系统)会带来很大的开销。5这使得它们成为批处理作业的不完美解决方案

---

3请注意,如果您的组织从公共云租用机器,则此点和下一点的适用性较低  
提供商。

4谷歌很久以前就决定,由于磁盘交换导致的延迟下降是如此可怕,以至于一个  
内存耗尽并迁移到另一台机器是普遍更可取的做法    所以在谷歌的案例中,它总是因内存不足而终止。

5尽管正在进行大量研究来降低这种开销,但它永远不会像本地运行的进程那样低。

容器化预计占用较少的资源并且运行时间较短。

这促使 Google 的工程师在 2003 年设计 Borg 时寻求不同的解决方案,最终采用了容器 一种基于 cgroups 的轻量级机制 (由 Google 工程师于 2007 年贡献给 Linux 内核)和 chroot jails、bind mounts 和/或 union/overlay 文件系统,用于文件系统隔离。开源容器实现包括 Docker 和 LMCTFY。

随着时间的推移和组织的发展,越来越多的潜在隔离故障被发现。举一个具体的例子,2011 年,从事 Borg 工作的工程师发现进程 ID 空间 (默认设置为 32,000 个 PID)耗尽正在成为隔离故障,并且必须对单个副本可以生成的进程/线程总数进行限制。我们将在本章后面更详细地介绍这个例子。

适当规模和自动扩展 2006 年的

Borg 根据工程师在配置中提供的参数 (例如副本数量和资源需求)来安排工作。

从远处看这个问题,要求人类确定资源需求数字的想法有些缺陷:这些数字不是人类每天接触的数字。因此,随着时间的推移,这些配置参数本身就成为效率低下的根源。工程师需要在首次推出服务时花时间确定它们,随着您的组织积累越来越多的服务,确定它们的成本也会增加。此外,随着时间的推移,程序不断发展 (可能会增长),但配置参数却跟不上。

这最终会导致中断 结果是,随着时间的推移,新版本对资源的需求会侵蚀为意外峰值或中断而留下的余地,而当这种峰值或中断实际发生时,剩余的余地却不够。

自然的解决方案是自动设置这些参数。不幸的是,事实证明要做好这一点出奇地困难。例如,Google 最近才达到整个 Borg 集群中超过一半的资源使用情况由合理规模自动化决定的程度。尽管它只占使用量的一半,但它是配置中较大的一部分,这意味着大多数工程师不需要担心调整容器大小这种繁琐且容易出错的负担。我们认为这是“简单的事情应该简单,复杂的事情应该可能”这一理念的成功应用 仅仅因为 Borg 工作负载的某些部分太复杂而无法通过合理规模进行妥善管理,并不意味着处理简单的情况没有很大价值。

## 概括

随着您的组织不断发展壮大并且您的产品越来越受欢迎,您将在以下所有方面都获得成长:

- 需要管理的不同应用程序的数量 · 需要运行的应用程序副本数量
- 最大应用程序的大小

为了有效地管理规模,自动化必不可少,这将使您能够解决所有这些增长轴。随着时间的推移,您应该期望自动化本身变得更加复杂,既要处理新类型的需求(例如, GPU 和 TPU 的调度是 Borg 在过去 10 年中发生的重大变化),又要处理更大的规模。在较小规模下可以手动执行的操作将需要自动化,以避免组织在负载下崩溃。

一个例子是自动化管理我们的数据中心,这也是 Google 仍在研究的一个转变。十年前,每个数据中心都是一个独立的实体。我们手动管理它们。启动一个数据中心是一个复杂的手动过程,需要一套专门的技能,需要数周时间(从所有机器准备就绪的那一刻起),而且本身就存在风险。然而,随着 Google 管理的数据中心数量的增长,我们转向了一种模式,在这个模式中,启动数据中心是一个自动化的过程,不需要人工

干涉。

## 编写用于托管计算的软件

从手动管理机器列表到自动调度和调整规模的转变使 Google 的机器群管理变得容易得多,但它也彻底改变了我们编写和思考软件的方式。

### 为失败而构建架构假设一名

工程师要处理一批一百万份文档并验证其正确性。如果处理一份文档需要一秒钟,那么整个工作将需要一台机器大约 12 天。这可能太长了。因此,我们将工作分片到 200 台机器上,从而将运行时间缩短到更易于管理的水平

100分钟。

如第 519 页的“自动调度”中讨论的那样，在 Borg 世界中，调度程序可以单方面杀死 200 个工作进程中的一个，并将其移动到另一台机器上。<sup>6</sup>“将其移动到另一台机器”意味着你的工作进程的新实例可以自动被消灭，而不需要人工通过 SSH 进入机器并调整某些环境变量或安装软件包。

从“工程师必须手动监控 100 项任务中的每一个，并在它们发生故障时处理它们”到“如果其中一个任务出现问题，系统的架构是让其他任务承担负载，同时自动调度程序会终止它并在新机器上重新实例化它”的转变，多年后被描述为“宠物与牛”的类比。<sup>7</sup>如果你的服务器是一只宠物，当它坏了，人类会来看它（通常是惊慌失措），了解出了什么问题，并希望它恢复健康。它很难更换。如果你的服务器是牛，你将它们命名为 replica001 到

replica100，如果其中一个出现故障，自动化将删除它并在其位置提供一个新的。“牛”的显着特征是很容易消除相关作业的新实例 - 它不需要手动设置，并且可以完全自动完成。

这实现了前面描述的自我修复特性 在发生故障的情况下，自动化可以接管并将不健康的作业替换为新的健康作业，而无需人工干预。请注意，虽然最初的比喻指的是服务器（VM），但同样适用于容器：如果您可以在没有人工干预的情况下从映像中生成新版本的容器，您的自动化将能够在需要时自动修复您的服务。

如果您的服务器是宠物，您的维护负担将随着您的舰队规模线性增长，甚至超线性增长，这是任何组织都不应轻视的负担。另一方面，如果您的服务器是牛，您的系统将能够在发生故障后恢复到稳定状态，您无需花费整个周末来照顾宠物服务器或容器使其恢复健康。

不过，让虚拟机或容器成为牛群并不足以保证您的系统在出现故障时表现良好。在拥有 200 台机器的情况下，其中一个副本被 Borg 杀死是相当可能的，可能不止一次，而且每次都会延长整个持续时间 50 分钟（或无论损失了多少处理时间）。为了优雅地处理这种情况，处理架构需要

---

<sup>6</sup> 调度程序不会随意执行此操作，而会出于具体原因（例如需要更新内核，或者机器上的磁盘出现问题，或者需要重新调整以使数据中心中工作负载的整体分布更好）。但是，拥有计算服务的意义在于，作为软件作者，我不应该知道或关心为什么会发生这种情况。

<sup>7</sup> [兰迪·拜亚斯](#) 将“宠物与牛”的比喻归功于 [比尔·贝克](#) 并且它已经成为一种非常流行的描述“复制软件单元”概念的方式。类似地，它也可以用于描述服务器以外的概念；例如，参见第 22 章。

不同之处在于：我们不是静态地分配工作，而是将整个一百万个文档集分成 1,000 个块，每个块包含 1,000 个文档。每当一个工作者完成一个特定块时，它都会报告结果并拾取另一个。这意味着，如果工作者在完成该块后但在报告之前死亡，我们最多会在工作者故障时丢失一个工作块。幸运的是，这非常符合当时 Google 的标准数据处理架构：在计算开始时，工作不会平均分配给工作者集；而是在整个处理过程中动态分配，以解决工作者故障的问题。

类似地，对于为用户流量提供服务的系统，理想情况下，您不希望重新安排容器导致向用户提供错误。当 Borg 调度程序计划出于维护原因重新安排容器时，它会提前向容器发出信号，告知其意图。容器可以通过拒绝新请求来对此做出反应，同时仍有时间完成正在进行的请求。这反过来又要求负载平衡器系统理解“我无法接受新请求”的响应（并将流量重定向到其他副本）。

总结一下：将您的容器或服务器视为牛意味着您的服务可以自动恢复到健康状态，但需要额外的努力来确保它在经历中等率的故障时能够顺利运行。

### 批处理与服务全局工作队列

(我们在本章第一部分中描述过)解决了 Google 工程师所称的“批处理作业”问题。这些程序需要完成某些特定任务（如数据处理）并运行至完成。批处理作业的典型示例是日志分析或机器学习模型学习。批处理作业与“服务作业”形成对比。这些程序需要无限期运行并处理传入的请求，典型示例是从预建索引中处理实际用户搜索查询的作业。

这两类工作通常具有不同的特点，<sup>8</sup>尤其是：

- 批处理作业主要关注处理吞吐量。服务作业关心的是处理单个请求的延迟。· 批处理作业的寿命较短（几分钟，最多几小时）。服务作业的寿命通常较长（默认情况下，仅在发布新版本时重新启动）。

---

<sup>8</sup>与所有分类一样，这一分类并不完美；有些类型的程序不能完全归入任何类别，或者具有服务作业和批处理作业的典型特征。但是，与大多数有用的分类一样，它仍然捕捉到了许多现实生活中存在的区别。

- 由于服务类工作寿命较长,因此启动时间也更长  
次。

到目前为止,我们的大多数示例都是关于批处理作业的。正如我们所见,为了使批处理作业能够经受住故障,我们需要确保将工作分散成小块并动态分配给工作者。在 Google 中,执行此操作的典型框架是 MapReduce<sup>9</sup>,后来被 Flume 取代。<sup>10</sup>

从很多方面来看,服务作业比批处理作业更适合抗故障。它们的工作自然地被分成小块(单个用户请求),并动态分配给工作者 通过跨服务器集群进行负载平衡来处理大量请求流的策略自互联网流量服务的早期就一直被使用。

但是,也有多个服务应用程序并不自然地符合该模式。典型的例子是任何你直观地描述为特定系统的“领导者”的服务器。这样的服务器通常会维护系统状态(在内存中或在其本地文件系统中),并且如果运行它的机器发生故障,新创建的实例通常将无法重新创建系统状态。另一个例子是,当你有大量数据需要服务时 一台机器无法容纳 因此你决定将数据分片到 100 台服务器中,每台服务器保存 1% 的数据,并处理对该部分数据的请求。

这类似于静态地将工作分配给批处理作业工作者;如果其中一个服务器出现故障,您将(暂时)失去提供部分数据的能力。最后一个例子是,如果您的服务器通过其主机名被系统的其他部分所知。在这种情况下,无论您的服务器如何构造,如果此特定主机失去网络连接,系统的其他部分将无法联系它。<sup>11</sup>

---

<sup>9</sup>请参阅 Jeffrey Dean 和 Sanjay Ghemawat,《MapReduce:大型集群上的简化数据处理》,第六届操作系统设计和实  
施研讨会(OSDI),2004 年。

<sup>10</sup> Craig Chambers,Ashish Raniwala,Frances Perry,Stephen Adams,Robert Henry,Robert Bradshaw 和  
Nathan Weizenbaum,“Flume -Java:简单、高效的数据并行管道”,ACM SIGPLAN 编程语言设计和实现会议(PLDI),2010  
年。

<sup>11</sup>另请参阅 Atul Adya 等人撰写的《数据中心应用程序的自动分片》,OSDI,2019 年;以及 Atul Adya,Daniel Myers,  
Henry Qin 和 Robert Grandl 撰写的《快速键值存储:一个时代已逝的想法》,  
HotOS XVII,2019 年。

## 管理状态前面描述中

的一个常见主题是,当尝试将作业视为牲畜时,状态是问题的根源。<sup>12</sup>每当你替换一个牲畜作业时,您都会丢失所有进程内状态(如果作业移动到另一台机器,则本地存储中的所有内容也会丢失)。这意味着进程内状态应被视为瞬态,而“真实存储”需要发生在其他地方。

处理这个问题最简单的方法是将所有存储提取到外部存储系统。这意味着,任何超出服务单个请求(在服务作业的情况下)或处理一个数据块(在批处理的情况下)范围的东西都需要存储在机器之外的持久存储中。如果你的所有本地状态都是不可变的,那么让你的应用程序具有抗故障能力应该是相对容易的。

不幸的是,大多数应用程序都没有那么简单。一个自然而然的问题是:“这些持久的、持久的存储解决方案是如何实现的 它们是牛吗?”答案应该是“是的”。持久状态可以由牛通过状态复制来管理。在另一个层面上,RAID 阵列是一个类似的概念;我们将磁盘视为瞬态的(接受其中一个磁盘可能消失的事实),同时仍保持状态。在服务器世界中,这可以通过多个副本保存单个数据并进行同步来实现,以确保每条数据都被复制足够多次(通常为 3 到 5 次)。请注意,正确设置这一点很困难(需要某种共识处理方式来处理写入),因此 Google 开发了许多专门的存储解决方案<sup>13</sup>,这些解决方案使大多数应用程序采用所有状态都是瞬态的模型。

牛可以使用的其他类型的本地存储包括“可重新创建”的数据,这些数据在本地保存以改善服务延迟。缓存是这里最明显的例子:缓存只不过是将状态保存在临时位置的临时本地存储,但状态不会一直消失,这平均而言可以实现更好的性能特征。谷歌生产基础设施的一个重要教训是配置缓存以满足您的延迟目标,但为核心应用程序配置总负载。这使我们能够避免在缓存层丢失时发生中断,因为非缓存路径被配置来处理总负载

---

<sup>12</sup>请注意,除了分布式状态之外,设置有效的“服务器即牛”解决方案还有其他要求,例如发现和负载平衡系统(以便在数据中心内移动的应用程序可以有效访问)。由于本书不是关于构建完整的 CaaS 基础设施,而是关于这种基础设施与软件工程艺术的关系,因此我们不会在这里详细介绍。

<sup>13</sup>例如,请参阅 Sanjay Ghemawat,Howard Gobioff 和 Shun-Tak Leung 的《Google 文件系统》,Pro-2003 年第 19 届 ACM 操作系统研讨会论文集;Fay Chang 等人的“Bigtable:一种用于结构化数据的分布式存储系统”,第 7 届 USENIX 操作系统设计和实现研讨会(OSDI);或 James C. Corbett 等人的“Spanner:Google 的全球分布式数据库”,OSDI,2012 年。

(尽管延迟更高)。但是,这里显然存在一个权衡:在冗余上花费多少才能降低缓存容量丢失时发生中断的风险。

与缓存类似,在应用程序预热时,数据可能会从外部存储拉到本地,以改善请求服务延迟。

使用本地存储的另一种情况是批量写入(这次是在写入数据多于读取数据的情况下)。这是监控数据的常见策略(例如,考虑从队列中收集CPU利用率统计信息以指导自动扩展系统),但它可以在任何可以接受一小部分数据消失的地方使用,要么是因为我们不需要100%的数据覆盖率(这是监控情况),要么是因为可以重新创建消失的数据(这是批量作业的情况,该作业以块为单位处理数据并为每个块写入一些输出)。请注意,在许多情况下,即使特定计算需要很长时间,也可以通过定期将状态检查点发送到持久存储来将其拆分为较小的时间窗口。

### 连接到服务如前所述,如果系

统中的任何东西都硬编码了程序运行所在的主机的名称(甚至在启动时作为配置参数提供),则程序副本就不是牛。但是,要连接到您的应用程序,另一个应用程序确实需要从某个地方获取您的地址。在哪里?

答案是增加一个间接层;也就是说,其他应用程序通过某个标识符来引用您的应用程序,该标识符在特定“后端”实例重新启动后仍然有效。当调度程序将您的应用程序放置在特定机器上时,该标识符可以由另一个系统解析。现在,为了避免在向您的应用程序发出请求的关键路径上进行分布式存储查找,客户端可能会在启动时查找可以找到您的应用程序的地址并建立连接,然后在后台对其进行监视。这通常称为服务发现,许多计算产品都有内置或模块化解决方案。大多数此类解决方案还包括某种形式的负载平衡,这进一步减少了与特定后端的耦合。

这种模型的后果是,在某些情况下,你可能需要重复你的请求,因为你正在与之通信的服务器可能会在它回答之前被关闭。<sup>14</sup>由于网络问题,重试请求是网络通信(例如,移动应用程序到服务器)的标准做法,但它可能不太直观

---

<sup>14</sup>请注意,重试需要正确实现 使用退避、优雅降级和避免抖动等級联故障的工具。因此,这应该是远程过程调用库的一部分,而不是由每个开发人员手动实现。例如,请参阅[第 22 章:解决級联故障](#)在 SRE 书中。

对于诸如服务器与其数据库通信之类的事情。这使得以优雅的方式设计服务器的 API 非常重要，以便优雅地处理此类故障。对于变异请求，处理重复请求很棘手。您想要保证的属性是幂等性的某种变体 - 即两次发出请求的结果与一次发出请求的结果相同。帮助实现幂等性的一个有用工具是客户端分配的标识符：如果您正在创建某些东西（例如，将披萨送到特定地址的订单），则客户端会为该订单分配某个标识符；如果已经记录了具有该标识符的订单，则服务器假定这是一个重复的请求并报告成功（它还可能验证订单的参数是否匹配）。

我们还发现，调度程序有时会因为网络问题而与某台机器失去联系。然后，调度程序认为该机器上的所有工作都已丢失，并将其重新安排到其他机器上，然后

机器回来了！现在我们在两台不同的机器上有两个程序，都认为它们是“replica072”。消除歧义的方法是检查地址解析系统引用了哪一个（另一个应该自行终止或被终止）；但这也是幂等性的另一种情况：执行相同工作并担任相同角色的两个副本是请求重复的另一个潜在来源。

## 一次性代码

之前的大部分讨论都集中在生产质量作业上，无论是服务于用户流量的作业，还是产生生产数据的数据处理管道。然而，软件工程师的生活还涉及运行一次性分析、探索性原型、自定义数据处理管道等。这些需要计算

资源。

通常，工程师的工作站是满足计算资源需求的令人满意的解决方案。例如，如果有人想自动浏览服务在过去一天生成的 1 GB 日志，以检查可疑行 A 是否总是出现在错误行 B 之前，他们只需下载日志，编写一个简短的 Python 脚本，然后让它运行一两分钟即可。

但是，如果他们想自动浏览该服务在过去一年中生成的 1 TB 日志（用于类似目的），等待大约一天才能得到结果可能是不可接受的。计算服务允许工程师在几分钟内在分布式环境中运行分析（利用几百个核心），这意味着现在进行分析和明天进行分析之间的区别。对于需要迭代的任务（例如，如果我需要在查看结果后优化查询），区别可能是在一天内完成和根本无法完成。

这种方法有时会引发一个担忧，即允许工程师在分布式环境中只运行一次性作业可能会浪费资源。这

当然,这是一种权衡,但应该有意识地进行权衡。工程师运行的处理成本不太可能比工程师编写处理代码所花费的时间更昂贵。确切的权衡值因组织的计算环境及其向工程师支付的费用而异,但一千个核心小时的成本不太可能接近一天的工程工作。在这方面,计算资源类似于我们在本书开篇讨论过的标记;公司在制定获取更多计算资源的流程时有一点节省机会,但这一过程在工程机会和时间上的损失可能比它节省的成本要多得多。

话虽如此,计算资源与标记不同,因为很容易意外占用过多的资源。虽然不太可能有人会拿走一千个标记,但完全有可能有人会不经意间编写一个占用一千台机器的程序。<sup>15</sup>对此的自然解决方案是为各个工程师设置资源使用配额。Google 使用的另一种方法是观察到,由于我们有效地免费运行低优先级批处理工作负载(请参阅后面的多租户部分),我们可以为工程师提供几乎无限的低优先级批处理配额,这对于大多数一次性工程任务来说已经足够了。

## 随着时间的推移和规模的 CaaS

我们上面讨论了 CaaS 在 Google 的发展历程以及实现它所需的基本部分——“只需为我提供运行我的东西所需的资源”这一简单使命如何发展为像 Borg 这样的实际架构。CaaS 架构如何影响软件在时间和规模上的生命周期的几个方面值得我们 反式仔细研究。

### 容器作为一种抽象

正如我们之前描述的,容器主要被视为一种隔离机制,一种实现多租户的方式,同时最大限度地减少共享一台机器的不同任务之间的干扰。这是最初的动机,至少在谷歌是这样。但事实证明,容器在抽象计算环境方面也发挥了非常重要的作用。

容器在部署的软件和运行该软件的实际机器之间提供了抽象边界。这意味着,随着时间的推移,机器发生变化,只有容器软件(可能由一个团队管理)需要

---

<sup>15</sup>这种事情在谷歌发生过多次;例如,有人在休假时忘记关闭占用一千台谷歌计算引擎虚拟机的负载测试基础设施,或者一名新员工在其工作站上调试主二进制文件,却没有意识到它在后台产生了 8,000 个全机工作者。

进行调整,而应用软件 (随着组织的发展,由每个团队进行管理)可以保持不变。

让我们讨论一下容器化抽象如何帮助组织管理变更的两个例子。

文件系统抽象提供了一种整合非公司编写的软件的方法,而无需管理自定义机器配置。这可能是组织在其数据中心运行的开源软件,也可能是它想要加入其 CaaS 的收购。如果没有文件系统抽象,加入需要不同文件系统布局的二进制文件 (例如,需要 /bin/foo/bar 处的辅助二进制文件) 将需要修改机群中所有机器的基本布局,或对机群进行碎片化,或修改软件 (由于许可方面的考虑,这可能很困难,甚至不可能)。

尽管如果导入外部软件是一生中只会发生一次的事情,这些解决方案可能是可行的,但如果导入软件成为一种常见的 (甚至是罕见的) 做法,那么它就不是一个可持续的解决方案。

某种文件系统抽象也有助于依赖管理,因为它允许软件预先声明和预先打包软件运行时所需的依赖项 (例如,特定版本的库)。依赖于机器上安装的软件会产生漏洞抽象,迫使每个人都使用相同版本的预编译库,并使升级任何组件变得非常困难,甚至不可能。

容器还提供了一种管理机器上命名资源的简单方法。

典型的例子是网络端口;其他命名资源包括专门的目标;例如, GPU 和其他加速器。

Google 最初并未将网络端口作为容器抽象的一部分,因此二进制文件必须自行搜索未使用的端口。因此, `PickUnusedPortOrDie` 函数在 Google C++ 代码库中有超过 20,000 次使用。

Docker 是在 Linux 命名空间推出后构建的,它使用命名空间为容器提供虚拟专用网卡,这意味着应用程序可以监听任何它们想要的端口。然后,Docker 网络堆栈将机器上的端口映射到容器内的端口。Kubernetes 最初是在 Docker 之上构建的,它更进一步,要求网络实现将容器 (Kubernetes 术语中的“pod”) 视为主机网络上可用的“真实”IP 地址。现在,每个应用程序都可以监听任何它们想要的端口,而不必担心冲突。

这些改进在处理并非设计为在特定计算堆栈上运行的软件时尤其重要。尽管许多流行的开源程序都有用于指定使用哪个端口的配置参数,但它们之间对于如何配置这些参数并不一致。

容器和隐式依赖关系和任何抽象一样,隐

式依赖关系的海勒姆定律也适用于容器抽象。它可能比平常更适用,因为用户数量巨大(在Google,所有生产软件和许多其他软件都将在Borg上运行),而且用户在使用文件系统等东西时并不觉得他们在使用API(甚至不太可能考虑这个API是否稳定、是否已版本化等)。

为了说明这一点,让我们回到2011年Borg经历的进程ID空间耗尽的例子。您可能想知道为什么进程ID是会耗尽的。它们不是可以从32位或64位空间分配的整数ID吗?在Linux中,它们实际上被分配在[0,...,PID\_MAX - 1]范围内,其中PID\_MAX默认为32,000。但是,可以通过简单的配置更改来提高PID\_MAX(提高到相当高的限值)。问题解决了吗?

嗯,不是。根据海勒姆定律,Borg上运行的进程的PID被限制在0...32,000范围内,这一事实成为人们开始依赖的隐式API保证;例如,日志存储过程依赖于PID可以存储为五位数字的事实,并且无法存储六位数字的PID,因为记录名称超过了允许的最大长度。处理这个问题变成了一个漫长的两阶段项目。首先,对单个容器可以使用的PID数量设置一个临时上限(以便单个线程泄漏作业不会导致整个机器无法使用)。其次,为线程和进程拆分PID空间。

(因为事实证明,很少有用户依赖分配给线程而不是进程的PID的32,000保证。因此,我们可以增加线程的限制,并将进程的限制保持在32,000。)第三阶段是将PID命名空间引入Borg,为每个容器提供自己完整的PID空间。可以预见的是(再次是海勒姆定律),许多系统最终假设三元组{主机名、时间戳、PID}唯一地标识了一个进程,如果引入PID命名空间,这种情况就会被打破。八年后,识别所有这些地方并修复它们(并反向移植任何相关数据)的努力仍在进行中。

这里的重点不是您应该在PID命名空间中运行容器。

虽然这是个好主意,但这并不是这里有趣的教训。当Borg的容器被构建时,PID命名空间还不存在;即使有,也不可能指望2003年设计Borg的工程师认识到引入它们的价值。

即使现在,机器上肯定也有一些资源没有得到充分隔离,这可能有一天会造成问题。这强调了设计一个随着时间的推移可以证明可维护的容器系统的挑战,因此使用由更广泛社区开发和使用的容器系统的价值,这些类型的问题已经发生在其他人身上,并且吸取了经验教训。

## 一项服务统领一切

如前所述,最初的 WorkQueue 设计仅针对某些批处理作业,这些作业最终都共享由 WorkQueue 管理的机器池,并且使用不同的架构来处理作业,每个特定的服务作业都在其自己的专用机器池中运行。开源的等效方案是为每种类型的工作负载运行一个单独的 Kubernetes 集群(并为所有批处理作业运行一个池)。

2003 年,Borg 项目启动,旨在(并最终成功)构建一个计算服务,将这些不同的池合并为一个大型池。Borg 的池涵盖服务和批处理作业,并成为任何数据中心的唯一池(相当于在每个地理位置为所有工作负载运行一个大型 Kubernetes 集群)。这里有两个值得讨论的显著效率提升。

第一个变化是服务机器变成了牛群(用 Borg 设计文档的话来说:“机器是匿名的:只要机器具备正确的特性,程序并不关心它们在哪台机器上运行”)。如果每个管理服务作业的团队都必须管理自己的机器池(自己的集群),那么维护和管理该机器池的组织开销将与每个团队相同。随着时间的推移,这些机器池的管理实践将不尽相同,使得全公司范围的变更(比如迁移到新的服务器架构或切换数据中心)变得越来越复杂。统一的管理基础设施-即为组织中所有工作负载提供通用的计算服务-使 Google 能够避免这种线性扩展因素;对于机器群中的物理机器,并没有 n 种不同的管理实践,只有 Borg。<sup>16</sup>第二个变化更微妙,可能并不适用于每个组织,但与 Google 非常相关。批处理和服务作业的不同需求结果是互补的。服务作业通常需要超额配置,因为它们需要有能力处理用户流量,且不会显著降低延迟,即使在使用量激增或部分基础设施中断的情况下也是如此。这意味着仅运行服务作业的机器将无法充分利用。人们很容易试图通过超额使用机器来利用这种空闲时间,但这首先违背了空闲时间的目的,因为如果确实发生峰值/中断,我们需要的资源将不可用。

但是,这种推理仅适用于服务作业!如果我们在一台机器上有许多服务作业,并且这些作业请求的 RAM 和 CPU 总计为

---

<sup>16</sup>任何复杂系统都有例外。并非 Google 拥有的所有机器都由 Borg 管理,也并非每个数据中心都由单个 Borg 单元覆盖。但大多数工程师都在不接触非 Borg 机器或非标准单元的环境中工作。

机器的总容量,即使资源的实际利用率只有容量的 30%,也无法再将服务作业放入其中。但我们可以(在 Borg 中,我们会)将批处理作业放在备用的 70% 中,策略是如果任何服务作业需要内存或 CPU,我们将从批处理作业中回收它们(如果是 CPU,则冻结它们;如果是 RAM,则杀死它们)。因为批处理作业关注的是吞吐量(以数百个工作器为单位进行总体衡量,而不是针对单个任务),而且它们的各个副本无论如何都是牛,它们会非常乐意吸收这些服务作业的备用容量。

根据给定机器池中的工作负载情况,这意味着要么所有批处理工作负载都在免费资源上有效运行(因为我们无论如何都在为服务作业的空闲时间付费),要么所有服务工作负载实际上只支付它们使用的资源,而不是它们需要的空闲容量(因为批处理作业正在空闲时间运行)。在 Google 的案例中,大多数时候,我们都能免费有效地运行批处理。

服务作业的多租户前面,我们讨

论了计算服务必须满足的一系列要求,才能适合运行服务作业。如前所述,让服务作业由通用计算解决方案管理有多种优势,但也带来了挑战。值得重复的一项特殊要求是发现服务,在第 528 页的“连接到服务”中进行了讨论。当我们想要将托管计算解决方案的范围扩展到服务任务时,还有许多其他新要求,例如:

- 需要限制作业的重新安排:尽管杀死并重新启动 50% 的批处理作业副本可能是可以接受的(因为这会在处理过程中引起暂时的波动,而我们真正关心的是吞吐量),但杀死并重新启动 50% 的服务作业副本是不可接受的(因为剩余的作业可能太少,无法在等待重新启动的作业再次恢复时为用户流量提供服务)。
- 批处理作业通常可以在没有警告的情况下终止。我们失去的是一些已经执行的处理,这些处理可以重做。当服务作业在没有任何警告的情况下终止时,我们可能会面临一些面向用户的流量返回错误或(在最好的情况下)延迟增加的风险;最好提前几秒钟发出警告,以便作业可以完成正在处理的请求,而不接受新的请求。

出于前面提到的效率原因,Borg 同时涵盖批处理和服务作业,但多种计算产品将这两个概念分开 - 通常是用于批处理作业的共享机器池,以及用于服务作业的专用、稳定的机器池。

然而,无论这两种类型的工作是否使用相同的计算架构,这两类工作都能从像牛一样的待遇中受益。

### 已提交的配置Borg 调度程序接收复制

服务或批处理作业的配置,以远程过程调用 (RPC) 的内容形式在单元中运行。服务操作员可以使用发送这些 RPC 的命令行界面 (CLI) 来管理它,并将 CLI 的参数存储在共享文档中或头部。

依赖文档和部落知识而非提交到存储库的代码通常不是一个好主意,因为文档和部落知识都有随时间推移而退化的趋势 (参见第 3 章)。然而,进化的下一个自然步骤 将 CLI 的执行包装在本地开发的脚本中 仍然不如使用专用的配置语言来指定服务的配置。

随着时间的推移,逻辑服务的运行时存在通常会在多个轴上超越一个数据中心的一组复制容器:

- 它将把它的存在扩展到多个数据中心 (为了用户亲和力和抵抗故障)。
- 除了生产环境/配置之外,它还将分为暂存和开发环境。
- 它将以以下形式累积不同类型的额外复制容器

附加服务,例如伴随服务的 memcached。

如果这种复杂的设置可以用一种标准化的配置语言来表达,从而轻松表达标准操作 (例如“将我的服务更新到新版本的二进制文件,但在任何给定时间内占用的容量不超过 5%”),那么服务的管理就会大大简化。

标准化配置语言提供了其他团队可以轻松纳入其服务定义中的标准配置。像往常一样,我们强调这种标准配置的价值会随着时间的推移和规模的扩大而不断增长。如果每个团队都编写不同的自定义代码片段来支持他们的 memcached 服务,那么执行组织范围内的任务 (例如,出于性能或许可原因)或向所有 memcache 部署推送安全更新)就会变得非常困难。还请注意,这种标准化配置语言是部署自动化的必要条件 (参见第 24 章)。

## 选择计算服务

任何组织都不太可能走上谷歌走过的道路,从头开始构建自己的计算架构。如今,现代计算产品既可以在开源世界 (如 Kubernetes 或 Mesos) 中使用,也可以在不同的

抽象级别,OpenWhisk 或 Knative),或作为公有云托管产品(同样,复杂程度不同,从 Google Cloud Platform 的托管实例组或 Amazon Web Services Elastic Compute Cloud [Amazon EC2] 自动扩展;到类似于 Borg 的托管容器,如 Microsoft Azure Kubernetes Service [AKS] 或 Google Kubernetes Engine [GKE];到无服务器产品,如 AWS Lambda 或 Google 的 Cloud Functions)。

但是,大多数组织都会选择计算服务,就像 Google 内部所做的那样。请注意,计算基础设施具有很高的锁定因素。其中一个原因是代码将以利用系统所有属性的方式编写(海勒姆定律);因此,例如,如果您选择基于 VM 的产品,团队将调整其特定的 VM 映像;如果您选择特定的基于容器的解决方案,团队将调用集群管理器的 API。如果您的架构允许代码将 VM(或容器)视为宠物,团队将这样做,然后转向依赖于将它们视为牛(甚至是不同形式的宠物)的解决方案将很困难。

为了说明计算解决方案的最小细节如何最终被锁定,请考虑 Borg 如何运行用户在配置中提供的命令。在大多数情况下,命令将是二进制文件的执行(可能后跟许多参数)。但是,为了方便起见,Borg 的作者还包括传入 shell 脚本的可能性;例如, while true; do ./但是,尽管二进制文件执行可以通过简单的 fork-and-exec 来完成(这是 Borg 做的),但 shell 脚本需要由 Bash 之类的 shell 运行。因此,Borg 实际上执行了/usr/bin/bash -c \$USER\_COMMAND,这在简单的二进制文件执行的情况下也有效。

17

某个时候,Borg 团队意识到,在 Google 的规模下,这个 Bash 包装器所消耗的资源(主要是内存)是不可忽略的,因此决定改用更轻量级的 shell:ash。因此,该团队对进程运行器代码进行了更改,改为运行/usr/bin/ash -c \$USER\_COMMAND。

您可能会认为这不是一个冒险的改变;毕竟,我们控制环境,我们知道这两个二进制文件都存在,所以这不可能不起作用。实际上,这不起作用的原因在于,Borg 工程师并不是第一个注意到运行 Bash 的额外内存开销的人。一些团队在限制内存使用方面很有创意,他们用一段自定义的“执行第二个参数”代码替换了(在他们的自定义文件系统覆盖中)Bash 命令。当然,这些团队非常清楚他们的内存使用情况,所以当 Borg 团队将进程运行器更改为使用 ash(未被覆盖)时

---

17这个特定的命令在 Borg 下是有害的,因为它阻止了 Borg 处理启动失败。但是,更复杂的包装器(例如,将环境的部分内容回显到日志记录中)仍在使用,以帮助调试启动问题。

自定义代码），它们的内存使用量增加（因为它开始包含 ash 使用量而不是自定义代码使用量），这导致了警报、回滚更改以及一定程度的不满。

计算服务选择难以随时间推移而改变的另一个原因是，任何计算服务选择最终都会被一个庞大的辅助服务生态系统所包围。用于日志记录、监控、调试、警报、可视化、实时分析、配置语言和元语言、用户界面等的工具。这些工具需要作为计算服务更改的一部分进行重写，对于中型或大型组织来说，理解和列举这些工具可能都是一项挑战。

因此，选择计算架构非常重要。与大多数软件工程选择一样，这涉及权衡。让我们讨论一下。

## 集中化与定制化

从计算堆栈的管理开销角度（以及从资源效率角度），组织能做的最好的事情就是采用单一的 CaaS 解决方案来管理其整个机器群，并只使用那里为每个人提供的工具。这确保随着组织的发展，管理机器群的成本仍然可控。这条路基本上就是谷歌对 Borg 所做的。

## 需要定制

然而，一个不断发展的组织将有越来越多样化的数据需求。例如，当谷歌在 2012 年推出谷歌计算引擎（“虚拟机即服务”公共云产品）时，虚拟机和谷歌的大多数其他产品一样，都是由 Borg 管理的。这意味着每个虚拟机都在由 Borg 控制的单独容器中运行。然而，“牛群”任务管理方法并不适合云的工作负载，因为每个特定的容器实际上都是某个特定用户正在运行的虚拟机，而云的用户通常不会将虚拟机视为牛群。<sup>18</sup>

调和这种差异需要双方付出大量工作。云组织确保支持虚拟机的实时迁移；也就是说，能够在一台机器上运行虚拟机，在另一台机器上启动该虚拟机的副本，将该副本变成完美的映像，最后将所有流量重定向到副本，而无需

---

<sup>18</sup>我的邮件服务器不能与您的图形渲染作业互换，即使这两个任务都在同一种形式的 VM 中运行。

从而导致服务在一段明显的不可用时间内无法使用。<sup>19</sup>另一方面,Borg 必须进行调整以避免随意杀死包含虚拟机的容器（以提供将虚拟机内容迁移到新机器的时间）,同时考虑到整个迁移过程更加昂贵,Borg 的调度算法也进行了调整以优化,从而降低需要重新调度的风险。<sup>20</sup>当然,这些修改只针对运行云端工作负载的机器,从而导致 Google 内部计算产品出现（小幅但仍然明显的）分叉。

一个不同的例子（但也会导致分歧）来自搜索。

2011 年左右,一个为 Google 搜索网络流量提供服务的复制容器在本地磁盘上建立了一个巨大的索引,存储了 Google 网络索引中不常访问的部分（更常见的查询由其他容器的内存缓存提供）。在特定机器上建立此索引需要多个硬盘的容量,并且需要几个小时才能填充数据。然而,当时,Borg 认为,如果某个容器中存储数据的任何磁盘出现故障,容器将无法继续运行,需要重新安排到另一台机器上。这种组合（以及与其他硬件相比,旋转磁盘的故障率相对较高）导致了严重的可用性问题;容器一直处于关闭状态,然后需要很长时间才能重新启动。为了解决这个问题,Borg 必须增加容器自行处理磁盘故障的功能,选择退出 Borg 的默认处理方式;而搜索团队必须调整流程以在部分数据丢失的情况下继续运行。

多个其他分支,涵盖文件系统形状、文件系统访问、内存控制、分配和访问、CPU/内存局部性、特殊硬件、特殊调度约束等领域,导致 Borg 的 API 界面变得庞大而笨重,行为交集变得难以预测,甚至更难测试。如果容器同时请求 Cloud 的特殊驱逐处理和 Search 的磁盘故障处理,没有人真正知道是否会发生预期的事情（在许多情况下,“预期”的含义甚至不明显）。

---

<sup>19</sup>这并不是使用户虚拟机能够实时迁移的唯一动机;它还提供了相当多面向用户的好处,因为这意味着可以在不中断虚拟机的情况下修补主机操作系统并更新主机硬件。另一种选择（其他主要云供应商使用）是提供“维护事件通知”,这意味着云提供商可以重新启动或停止虚拟机,然后再启动虚拟机。

<sup>20</sup>这一点尤其重要,因为并非所有客户虚拟机都选择进行实时迁移;对于某些工作即使是迁移期间的短暂性能下降,负载也是不可接受的。这些客户将收到维护事件通知,除非绝对必要,否则 Borg 将避免驱逐这些虚拟机的容器。

2012 年之后,Borg 团队投入了大量时间来清理 Borg 的 API。他们发现 Borg 提供的某些功能已经完全不再使用。<sup>21</sup>最令人担忧的功能是那些被多个容器使用的功能,但尚不清楚是否是故意的。在项目之间复制配置文件的过程导致原本仅供高级用户使用的功能使用泛滥。某些功能被引入白名单以限制其传播并明确标记为仅供高级用户使用。然而,清理工作仍在进行中,一些更改(如使用标签来识别容器组)仍未完全完成。<sup>22</sup>

像往常一样,在权衡利弊时,尽管有办法投入精力并获得定制化的一些好处,同时又不会遭受最糟糕的负面影响(例如前面提到的白名单功能),但最终还是需要做出艰难的选择。这些选择通常以多个小问题的形式出现:我们是否接受扩展显式(或更糟的是隐式)API 界面以适应我们基础设施的特定用户,或者我们是否要给该用户带来很大的不便,但保持更高的一致性?

## 抽象级别:无服务器

谷歌驯服计算环境的描述很容易被理解为增加和改进抽象的故事。更高级的 Borg 版本承担了更多的管理责任,并将容器与底层环境进一步隔离。很容易给人留下这是一个简单故事的印象:更多的抽象是好的;更少的抽象是坏的。

当然,事情没那么简单。这里的环境很复杂,有多种产品。在第 518 页的“驯服计算环境”中,我们讨论了从处理在裸机上运行的宠物(由您的组织拥有或从主机托管中心租用)到将容器作为牲畜进行管理的进展。在这两者之间,作为替代路径,是基于 VM 的产品,其中 VM 可以从裸机的更灵活替代品(在 Google Compute Engine [GCE] 或 Amazon EC2 等基础设施即服务产品中)发展为容器的更强大替代品(具有自动缩放、适当大小和其他管理工具)。

根据谷歌的经验,选择管理牛(而不是宠物)是大规模管理的解决方案。重申一下,如果你的每个团队在每个数据中心只需要一台宠物机器,那么你的管理成本将随着你的

---

<sup>21</sup>一个很好的提醒是,随着时间的推移,监控和跟踪功能的使用情况是有价值的。

<sup>22</sup>这意味着,Kubernetes 受益于清理 Borg 的经验,但一开始并没有受到广泛现有用户群的阻碍,从一开始就在很多方面(如其对标签的处理)都更加现代。话虽如此,Kubernetes 现在在各种类型的应用程序中被广泛采用,因此也遭遇了一些相同的问题。

组织的增长（因为团队数量和团队占用的数据中心数量都可能增长）。在选择管理牲畜之后，容器自然而然地成为管理的选择；它们更轻量（意味着更小的资源开销和启动时间）并且可配置性足够强，如果您需要为特定类型的工作负载提供专用硬件访问，您可以（如果您愿意）轻松地打通一个洞。

虚拟机作为“牛”的优势主要在于能够使用我们自己的操作系统，如果您的工作负载需要运行多种操作系统，这一点就很重要。多个组织还将拥有管理虚拟机的现有经验，以及基于虚拟机的现有配置和工作负载，因此可能会选择使用虚拟机而不是容器来降低迁移成本。

### 什么是无服务器？

无服务器产品是更高层次的抽象。<sup>23</sup>假设一个组织正在提供 Web 内容，并使用（或愿意采用）一个通用的服务器框架来处理 HTTP 请求和提供响应。框架的关键定义特征是控制反转 - 因此，用户只需负责编写某种“操作”或“处理程序” - 所选语言中的函数，它接受请求参数并返回响应。

在 Borg 世界中，运行此代码的方式是建立一个复制容器，每个副本包含一个由框架代码和函数组成的服务器。如果流量增加，您将通过扩展（添加副本或扩展到新数据中心）来处理此问题。如果流量减少，您将缩小规模。请注意，需要最低限度的存在（Google 通常假设服务器运行的每个数据中心至少有三个副本）。

但是，如果多个不同的团队使用同一个框架，则可以采用不同的方法：我们不仅可以使计算机成为多租户，还可以使框架服务器本身成为多租户。通过这种方法，我们最终会运行更多的框架服务器，根据需要在不同的服务器上动态加载/卸载操作代码，并动态将请求定向到已加载相关操作代码的服务器。各个团队不再运行服务器，因此称为“无服务器”。

大多数关于无服务器框架的讨论都将其与“虚拟机作为宠物”模型进行比较。在这种情况下，无服务器概念是一场真正的革命，因为它带来了牲畜管理的所有好处：自动扩展、更低的开销、无需明确配置服务器。然而，如前所述，转向共享、多租户、

---

<sup>23</sup> FaaS（功能即服务）和 PaaS（平台即服务）是与 Serverless 相关的术语。这三个术语之间有区别，但相似之处较多，界限有些模糊。

基于cattle的模型应该已经成为组织扩展计划的目标;因此无服务器架构的自然比较点应该是“持久容器”架构,如Borg、Kubernetes或Mesosphere。

### 优点和缺点

首先请注意,无服务器架构要求您的代码真正无状态;我们不太可能在无服务器架构中运行用户的虚拟机或实现Spanner。我们之前讨论过的所有管理本地状态的方法(除了不使用它)都不适用。在容器化世界中,您可能在启动时花费几秒钟或几分钟来设置与其他服务的连接、从冷存储中填充缓存等等,并且您期望在典型情况下,在终止之前会给您一个宽限期。在无服务器模型中,没有真正地在请求之间持久化的本地状态;您想要使用的所有内容都应该在请求范围内设置。

实际上,大多数组织都有一些需求,而真正的无状态工作负载无法满足这些需求。这可能导致组织依赖特定解决方案(无论是自主开发还是第三方解决方案)来解决特定问题(例如托管数据库解决方案,它经常与公有云无服务器产品搭配使用),或者拥有两种解决方案:基于容器的解决方案和无服务器的解决方案。值得一提的是,许多或大多数无服务器框架都构建在其他计算层之上:AppEngine在Borg上运行,Knative在Kubernetes上运行,Lambda在Amazon EC2上运行。

托管无服务器模型对于灵活扩展资源成本很有吸引力,尤其是在低流量端。例如,在Kubernetes中,您的复制容器无法缩减到零容器(因为假设在请求服务时启动容器和节点的速度太慢)。这意味着在持久集群模型中,仅拥有一个应用程序的成本最低。另一方面,无服务器应用程序可以轻松缩减到零;因此,仅拥有它的成本就会随着流量而增长。

在流量极高的情况下,无论采用哪种计算解决方案,您都必然会受到底层基础设施的限制。如果您的应用程序需要使用100,000个核心来处理其流量,则无论支持您所使用的基础设施的物理设备是什么,都需要有100,000个物理核心可用。在流量稍低的情况下,您的应用程序确实有足够的流量来让多台服务器保持繁忙,但不足以给基础设施提供商带来问题,持久性容器解决方案和无服务器解决方案都可以扩展以处理它,尽管无服务器解决方案的扩展将比持久性容器解决方案更具响应性和更细粒度。

最后,采用无服务器解决方案意味着对环境的控制会有所丧失。从某种程度上来说,这是件好事:拥有控制权意味着必须行使控制权,而这意味着管理开销。但当然,这也意味着如果你

需要一些您使用的框架中没有的额外功能,这将成为您的一个问题。

举一个具体的例子,Google Code Jam 团队 (举办一场有数千名参与者的编程竞赛,前端在 Google AppEngine 上运行)有一个定制脚本,在竞赛开始前几分钟使竞赛网页出现人为的流量高峰,以便预热足够多的应用实例来满足竞赛开始时的实际流量。

这虽然有效,但人们希望通过选择无服务器解决方案来避免这种手动调整 (以及黑客攻击)。

## 权衡

在这种权衡之下,谷歌的选择是不对无服务器解决方案进行大量投资。

Google 的持久容器解决方案 Borg 足够先进,可以提供大多数无服务器优势 (如自动扩展、适用于不同类型应用程序的各种框架、部署工具、统一日志记录和监控工具等)。唯一缺少的是更积极的扩展 (特别是缩减到零的能力),但 Google 的绝大部分资源占用来自高流量服务,因此过度配置小型服务的成本相对较低。与此同时,Google 运行着多个在“真正无状态”世界中无法运行的应用程序,从 GCE 到 BigQuery 等自研数据库系统或 Spanner,用于需要很长时间填充缓存的服务器,如前面提到的长尾搜索服务作业。因此,为所有这些事物建立一个通用的统一架构的好处超过了为部分工作负载使用单独的无服务器堆栈的潜在收益。

然而,谷歌的选择并不一定是每个组织的正确选择:其他组织已经成功构建了混合容器/无服务器架构,或者利用第三方解决方案进行存储的纯无服务器架构。

然而,无服务器的主要吸引力不在于大型组织做出的选择,而在于较小的组织或团队;在这种情况下,比较本质上是不公平的。无服务器模型虽然限制性更强,但它允许基础设施供应商承担更大份额的整体管理开销,从而减少用户的管理开销。如果集群不在多个团队之间共享,那么在共享无服务器架构 (如 AWS Lambda 或 Google 的 Cloud Run)上运行一个团队的代码比在托管容器服务 (如 GKE 或 AKS)上运行代码要简单得多 (也更便宜)。如果您的团队希望获得托管计算产品的好处,但您的大型组织不愿意或无法迁移到基于持久容器的解决方案,那么公共云提供商提供的无服务器产品可能会对您有吸引力,因为成本 (资源和

只有当集群真正共享（在组织中的多个团队之间）时，共享集群的“成本管理”才能很好地摊销。

但请注意，随着组织的发展和托管技术的采用，您可能会超越纯无服务器解决方案的限制。

这使得存在突破路径（如从 KNative 到 Kubernetes）的解决方案具有吸引力，因为它们提供了一条通往像 Google 这样的统一计算架构的自然路径（如果您的组织决定走这条路的话）。

## 公共与私人

谷歌刚起步时，CaaS 产品主要是自产的；如果你想要一个，就自己开发。在公有云和私有云领域，你唯一的选择就是拥有机器和租用机器，但所有集群的管理都由你决定。

在公共云时代，有更便宜的选择，但也有更多的选择，组织必须做出选择。

使用公有云的组织实际上是将（部分）管理开销外包给公有云提供商。对于许多组织来说，这是一个有吸引力的提议：他们可以专注于在特定专业领域提供价值，而不需要培养重要的基础设施专业知识。虽然云提供商（当然）收取的费用高于金属裸机成本以收回管理费用，但他们已经积累了专业知识，并且正在与多个客户共享。

此外，公有云是一种更轻松地扩展基础架构的方法。随着抽象级别的增长（从主机托管、购买虚拟机时间，到托管容器和无服务器产品），扩展的难易程度也在增加（从必须签署主机托管空间租赁协议，到需要运行 CLI 来获取更多虚拟机，再到资源占用量会随着收到的流量自动变化的自动扩展工具）。特别是对于年轻的组织或产品而言，预测资源需求是一项挑战，因此不必预先配置资源的优势非常显著。

选择云提供商时，一个重要的担忧是担心被锁定：提供商可能会突然提高价格，甚至可能失败，让组织陷入非常困难的境地。Zimki 是首批提供无服务器服务的提供商之一，它是一个用于运行 JavaScript 的平台即服务环境，于 2007 年提前三个月通知关闭。

部分缓解措施是使用基于开源架构（如 Kubernetes）的公共云解决方案。这是为了确保保存在迁移路径，即使特定基础设施提供商由于某种原因变得不可接受。虽然这可以减轻很大一部分风险，但这并不是完美的

策略。根据海勒姆定律,很难保证不使用特定供应商特有的部件。

该策略有两种可能的扩展。一种是使用较低级别的公共云解决方案（如 Amazon EC2）,并在其上运行更高级别的开源解决方案（如 Open-Whisk 或 KNative）。这试图确保如果您想要迁移出去,您可以对更高级别的解决方案、在其基础上构建的工具以及您拥有的隐式依赖项进行任何调整。另一种是运行多云;也就是说,使用来自两个或更多不同云提供商的基于相同开源解决方案的托管服务（例如,Kubernetes 的 GKE 和 AKS）。这为从其中一个云提供商迁移提供了一条更简单的途径,也使得依赖其中一个云提供商提供的特定实现细节变得更加困难。

另一个相关策略（更多地用于管理迁移,而不是用于管理锁定）是在混合云中运行;也就是说,将部分总体工作负载放在私有基础架构上,将部分工作负载放在公共云提供商上运行。可以使用的方法之一是使用公共云来处理溢出。组织可以在私有云上运行其大部分典型工作负载,但在资源短缺的情况下,将部分工作负载扩展到公共云。同样,为了使此方法有效,需要在两个空间中使用相同的开源计算基础架构解决方案。

多云和混合云策略都要求多个环境能够很好地连接在一起,通过不同环境中的机器之间的直接网络连接以及两者中可用的通用 API。

## 结论

在构建、改进和运行其计算基础架构的过程中,Google 了解到精心设计的通用计算基础架构的价值。为整个组织提供单一基础架构（例如,每个区域一个或少数几个共享的 Kubernetes 集群）可显著提高管理和资源成本效率,并允许在该基础架构之上开发共享工具。在构建这样的架构时,容器是允许在不同任务之间共享物理（或虚拟）机器（从而提高资源效率）以及在应用程序和操作系统之间提供抽象层以提供随时间推移的弹性的关键工具。

要充分利用基于容器的架构,需要在设计应用程序时使用“牛群”模型:设计应用程序以包含可以轻松自动替换的节点,从而可以扩展到数千个实例。编写与该模型兼容的软件需要不同的思维模式;例如,将所有本地存储（包括磁盘）视为临时存储并避免对主机名进行硬编码。

也就是说,尽管谷歌总体上对其架构的选择感到满意和成功,但其他组织将从广泛的计算服务中进行选择 - 从手动管理的虚拟机或机器的“宠物”模型,到“牛”复制容器,再到抽象的“无服务器”模型,所有这些都以托管和开源风格提供;您的选择是多种因素的复杂权衡。

## TL;DR

- 规模需要通用的基础设施来运行生产中的工作负载。 · 计算解决方案可以为软件提供标准化、稳定的抽象和环境。
- 软件需要适应分布式、托管的计算环境。 · 应慎重选择组织的计算解决方案,以提供适当的抽象级别。



第五部分

---

## 结论



---

## 后记

Google 的软件工程一直是开发和维护庞大且不断发展的代码库的非凡实验。我在这里工作期间，亲眼目睹工程团队在这方面取得突破，推动 Google 成为一家触及数十亿用户的公司和科技行业的领导者。如果没有本书中概述的原则，这一切都不可能实现，因此我很高兴看到这些页面得以实现。

如果说过去 50 年（或前面几页）证明了什么，那就是软件工程远非停滞不前。在技术不断变化的情况下，软件工程功能在特定组织中发挥着特别重要的作用。今天，软件工程原则不仅仅是关于如何有效地运营一个组织；它们是关于如何成为一家对用户和整个世界更负责任的公司。

常见软件工程问题的解决方案并不总是隐藏在显而易见的地方。大多数问题都需要一定程度的坚定敏捷性，以找到既能解决当前问题又能经受住技术系统不可避免的变化的解决方案。自 2008 年加入 Google 以来，我有幸与软件工程团队共事并从他们身上学到了很多东西，这种敏捷性是这些团队的共同品质。

可持续性的理念也是软件工程的核心。在代码库的预期寿命内，我们必须能够对变化做出反应和适应，无论是产品方向、技术平台、底层库、操作系统还是其他方面。

今天，我们依靠本书概述的原则来实现改变软件生态系统各个部分的关键灵活性。

我们当然无法证明我们所找到的实现可持续性的方法适用于每个组织，但我认为分享这些关键经验很重要。软件工程是一门新学科，因此很少有组织有机会同时实现可持续性和规模化。通过提供我们所见所闻以及沿途遇到的坎坷，我们希望展示可持续性的价值和

代码健康长期规划的可行性。时间的流逝和变化的重要性不容忽视。

本书概述了我们与软件工程相关的一些关键指导原则。从高层次来看，它还阐明了技术对社会的影响。作为软件工程师，我们有责任确保我们的代码设计具有包容性、公平性和可访问性。单纯为了创新而构建已不再可接受；只帮助一组用户的技术根本不是创新。

在 Google，我们的责任一直是为内部和外部的开发者提供一条光明的道路。随着人工智能、量子计算和环境计算等新技术的兴起，我们作为一家公司仍有很多东西需要学习。我特别期待看到行业在未来几年将软件工程带向何方，我相信这本书将有助于塑造这条道路。

阿西姆·侯赛因  
谷歌工程副总裁