

## 第九章

# 代码审查

作者:汤姆·曼什雷克 (Tom Mansreck) 和凯特琳·萨多夫斯基 (Caitlin Sadowski)  
由 Lisa Carey 编辑

代码审查是指由作者以外的人审查代码的过程,通常是在将代码引入代码库之前。虽然这是一个简单的定义,但整个软件行业对代码审查过程的实施各不相同。一些组织在整个代码库中都有一组精选的“守门人”来审查更改。其他组织将代码审查流程委托给较小的团队,允许不同的团队要求不同级别的代码审查。在 Google,基本上每个更改在提交之前都会经过审查,每个工程师都负责发起审查和审查更改。

代码审查通常需要流程和支持该流程的工具的结合。在 Google,我们使用自定义代码审查工具 Critique 来支持我们的流程。<sup>1</sup> Critique 在 Google 中非常重要,值得在本书中单独写一章。本章重点介绍 Google 的代码审查流程,而不是特定的工具,因为这些基础比工具更古老,而且这些见解中的大多数都可以适用于您可能用于代码审查的任何工具。



有关批评的更多信息,请参阅第 19 章。

---

<sup>1</sup>我们还使用Gerrit审查 Git 代码,主要用于我们的开源项目。然而,Critique 是 Google 典型软件工程师的主要工具。

代码审查的一些好处,例如在代码进入代码库之前检测出代码中的错误,是众所周知的<sup>2</sup>并且有点显而易见(如果衡量不精确的话)。

然而,其他好处则更加微妙。由于Google的代码审查流程非常普遍和广泛,我们注意到了许多更微妙的影响,包括心理方面的影响,这些影响随着时间的推移和规模的扩大为组织带来了许多好处。

## 代码审查流程

代码审查可以在软件开发的许多阶段进行。在Google,代码审查是在将更改提交到代码库之前进行的;此阶段也称为提交前审查。代码审查的主要最终目标是让另一位工程师同意更改,我们将更改标记为“我认为不错”(LGTM)。我们将此LGTM用作必要的权限“位”(与下面提到的其他位结合使用)以允许提交更改。

Google的典型代码审查过程经过以下步骤:

1. 用户在工作区中将更改写入代码库。然后,该作者创建更改的快照:补丁和相应的描述,并上传到代码审查工具。此更改会针对代码库生成diff,用于评估哪些代码已更改。
2. 作者可以使用此初始补丁来应用自动审核评论或进行自我审核。当作者对更改的差异感到满意时,他们会将更改发送给一个或多个审阅者。此过程会通知这些审阅者,要求他们查看快照并发表评论。
3. 审阅者在代码审阅工具中打开更改并针对差异发表评论。有些评论要求明确解决方案。有些仅提供信息。
4. 作者根据反馈修改变更并上传新快照,然后回复审阅者。步骤3和4可以重复多次。
5. 审阅者对变更的最新状态感到满意后,他们同意变更并将其标记为“我认为不错”(LGTM)。默认情况下只需要一个LGTM,尽管惯例可能要求所有审阅者都同意变更。
6. 当变更被标记为LGTM后,作者可以将变更提交到代码库,前提是他们解决了所有评论并且变更得到批准。  
我们将在下一节中讨论批准。

---

<sup>2</sup> Steve McConnell,《代码大全》(雷德蒙德:微软出版社,2004年)。

我们将在本章后面更详细地讨论这一过程。

代码是一种责任记住（并

接受)代码本身就是一种责任很重要。这可能是一种必要的责任,但就其本身而言,代码只是某个地方某人的维护任务。就像飞机携带的燃料一样,它有重量,尽管它当然是飞机飞行所必需的。

当然,新功能通常是必要的,但在开发代码之前,首先应该小心谨慎,确保任何新功能都是合理的。重复的代码不仅浪费精力,而且实际上比没有代码花费的时间更多;在一种代码模式下可以轻松完成的更改,在代码库中有重复时往往需要付出更多努力。编写全新的代码是如此令人反感,以至于我们中的一些人有这样一句话:“如果你从头开始编写,你就做错了!”

对于库或实用程序代码尤其如此。如果你正在编写实用程序,那么在与 Google 规模相当的代码库中,其他人可能也做过类似的事情。因此,第 17 章中讨论的工具对于查找此类实用程序代码和防止引入重复代码至关重要。

理想情况下,这项研究是事先完成的,并且在编写任何新代码之前,任何新事物的设计都已经传达给适当的团队。

当然,新项目会出现,新技术会引入,需要新组件,等等。尽管如此,代码审查并不是重新讨论或辩论先前设计决策的机会。设计决策通常需要时间,需要传播设计提案,在 API 审查或类似会议上讨论设计,甚至可能开发原型。尽管对全新代码的代码审查不应该突然发生,但代码审查过程本身也不应该被视为重新审视先前决策的机会。

## Google 的代码审查是如何进行的

我们已经大致指出了典型的代码审查流程是如何进行的,但细节决定成败。本节详细介绍了 Google 的代码审查工作方式,以及这些做法如何使其随着时间的推移而适当扩展。

审查有三个方面需要对任何给定的变更进行“批准”,  
谷歌:

- 另一位工程师对代码的正确性和理解性进行检查,以确保代码合适并且符合作者的声明。这通常是团队成员,但不一定非得是团队成员。这反映在 LGTM 权限中

“bit” ,在同行评审员同意代码 “看起来不错”后设置。

- 代码所有者之一批准该代码适用于代码库的这一特定部分（并且可以签入特定目录）。如果作者是这样的所有者，则此批准可能是隐式的。Google 的代码库是一个树形结构，特定目录的所有者具有层次结构。（参见第 16 章）。

所有者是其特定目录的守门人。任何工程师都可以提出更改，其他工程师也可以 LGTM 批准，但相关目录的所有者还必须批准将此添加到其代码库部分。此类所有者可能是技术主管或其他被视为该特定代码库领域专家的工程师。通常由每个团队决定分配所有权特权的广度或范围。

- 获得语言 “可读性”<sup>3</sup> 专家的认可，确认代码符合语言风格和最佳实践，检查代码是否以我们期望的方式编写。如果作者具有这样的可读性，这种认可可能也是隐含的。这些工程师是从全公司的工程师人才库中挑选出来的，这些工程师都获得了该编程语言的可读性认证。

虽然这种控制水平听起来很繁重（而且有时确实如此），但大多数评审都是一个人承担所有三个角色，这大大加快了流程。重要的是，作者也可以承担后两个角色，只需要另一位工程师的 LGTM 将代码签入他们自己的代码库，前提是他们已经具备该语言的可读性（所有者通常具备）。

这些要求使代码审查过程变得非常灵活。作为项目所有者且具有该代码语言可读性的技术主管可以提交代码更改，但只需另一位工程师的 LGTM 即可。没有此类权限的实习生可以将相同的更改提交到同一代码库，前提是他们获得具有语言可读性的所有者的批准。上述三个权限 “位”可以任意组合。作者甚至可以向不同的人请求多个 LGTM，方法是明确将更改标记为需要所有审阅者的 LGTM。

实际上，大多数需要多次批准的代码审查通常要经过两个步骤：从同行工程师那里获得 LGTM，然后寻求适当的代码所有者/可读性审查者的批准。这使得两个角色可以专注于代码审查的不同方面，从而节省了审查时间。主要审查者可以专注于代码的正确性和代码更改的总体有效性；代码所有者可以专注于此更改是否适合他们的部分。

---

<sup>3</sup> 在谷歌，“可读性”不仅仅指理解力，还指一套风格和最佳实践，允许其他工程师维护代码。请参阅第 3 章。

代码库的审核员,而不必关注每行代码的细节。换句话说,审批者通常寻找的东西与同行评审者不同。毕竟,有人试图将代码签入他们的项目/目录。他们更关心这样的问题：“这个代码维护起来容易还是困难？”“它会增加我的技术债务吗？”“我们团队是否具备维护它的专业知识？”

如果这三种类型的审查都可以由一位审查员处理,那么为什么不让这些类型的审查员处理所有的代码审查呢?简而言之,就是规模。将这三个角色分开可以增加代码审查过程的灵活性。如果您正在与同行合作开发实用程序库中的新功能,您可以让团队中的某个人审查代码以确保代码的正确性和可理解性。经过几轮(可能持续几天),您的代码让您的同行审查员满意,您将获得 LGTM。

现在,您只需要让图书馆的所有者(所有者通常具有适当的可读性)批准更改。

## 所有权

海伦·莱特

当在专用存储库中与小团队合作时,通常会授予整个团队对存储库中所有内容的访问权限。毕竟,您认识其他工程师,该领域足够狭窄,每个人都可以是专家,而且人数较少可以限制潜在错误的影响。

随着团队规模的扩大,这种方法可能无法扩展。其结果要么是存储库分裂混乱,要么是采用不同的方法来记录存储库不同部分中谁拥有哪些知识和责任。在 Google,我们将这组知识和责任称为所有权,将行使这些权利的人称为所有者。

这一概念不同于拥有一组源代码,而是暗示了一种管理意识,即利用部分代码库以公司的最佳利益行事。(事实上,如果我们能重来一次,“管理者”几乎肯定是一个更好的术语。)

特别命名的 OWNERS 文件列出了对目录及其子目录拥有所有权责任的人员的用户名。这些文件还可能包含对其他 OWNERS 文件或外部访问控制列表的引用,但最终它们会解析为个人列表。每个子目录也可能包含一个单独的 OWNERS 文件,并且关系是层次化的:给定文件通常由目录树中其上方的所有 OWNERS 文件的成员的联合拥有。OWNERS 文件可以包含团队喜欢的任意数量的条目,但我们鼓励使用相对较小且重点突出的列表,以确保责任明确。

Google 代码的所有权赋予了一个人对其权限范围内代码的批准权,但这些权利也伴随着一系列责任,例如了解所拥有的代码或知道如何找到拥有这些代码的人。不同的团队有

授予新成员所有权的标准不同,但我们通常鼓励他们不要将所有权作为入会仪式,并鼓励离职成员尽快放弃所有权。

这种分布式所有权结构支持我们在本书中概述的许多其他实践。例如,根 OWNERS 文件中的一组人员可以充当大规模变更的全局批准者(参见第 22 章),而无需打扰本地团队。同样,OWNERS 文件充当一种文档,使人们和工具只需沿着目录树向上查找,就可以轻松找到负责给定代码段的人员。创建新项目时,没有中央机构必须注册新的所有权特权:新的 OWNERS 文件就足够了。

这种所有权机制简单而强大,在过去二十年中得到了很好的扩展。这是 Google 确保数万名工程师能够在单个存储库中高效操作数十亿行代码的方法之一。

## 代码审查的好处

在整个行业中,代码审查本身并不存在争议,尽管它远非普遍做法。许多(甚至大多数)其他公司和开源项目都有某种形式的代码审查,大多数人认为这个过程与将新代码引入代码库时的健全性检查一样重要。软件工程师了解代码审查的一些更明显的好处,即使他们个人可能不认为它适用于所有情况。但在谷歌,这个过程通常比大多数其他公司更彻底、更广泛。

与许多软件公司一样,Google 的文化是给予工程师广泛的工作自由。人们认识到,对于需要快速响应新技术的动态公司来说,严格的流程往往不太适用,而官僚规则则不太适合具有创造性专业人才。然而,代码审查是一项强制性规定,是 Google 所有软件工程师都必须参与的少数几个一刀切流程之一。Google 要求对代码库的几乎每一次代码更改(无论大小)进行代码审查。这项强制性规定确实会带来成本并影响工程速度,因为它会减慢新代码进入代码库的速度,并会影响任何特定代码更改的生产时间。(这两者都是软件工程师对严格代码审查流程的常见抱怨。)那么,我们为什么需要这个过程呢?为什么我们认为这会带来长期利益?

---

<sup>4</sup>对文档和配置的一些更改可能不需要代码审查,但通常仍需要进行代码审查。  
能够获得这样的评价。

精心设计的代码审查流程和认真对待代码审查的文化可带来以下好处：

- 检查代码正确性
- 确保其他工程师能够理解代码更改 · 确保整个代码库的一致性 · 从心理上促进团队所有权 · 实现知识共享 · 提供代码审查本身的历史记录

这些好处中的许多对于软件组织来说都是至关重要的,它们不仅对作者有益,而且对审阅者也有益。以下各节将详细介绍这些项目。

## 代码正确性

代码审查的一个明显好处是,它允许审查者检查代码更改的“正确性”。让另一双眼睛检查更改有助于确保更改达到预期效果。审查者通常会检查更改是否经过适当测试、设计是否正确以及是否正确高效地运行。在许多情况下,检查代码正确性就是检查特定更改是否会将错误引入代码库。

许多报告指出,代码审查在预防软件未来出现错误方面非常有效。IBM 的一项研究发现,在流程中尽早发现缺陷,可以减少后续修复所需的时间,这一点并不令人意外。<sup>5</sup>如果代码审查流程本身得到精简,以保持其轻量级,那么在代码审查上投入的时间可以节省测试、调试和执行回归分析所花费的时间。后一点很重要;如果代码审查流程过于繁重或无法正确扩展,就会变得不可持续。<sup>6</sup>我们将在本章后面介绍一些保持流程轻量级的最佳实践。

为了防止正确性评估过于主观而非客观,作者通常会尊重他们自己的特定方法,无论是在设计还是在引入的更改的功能方面。审阅者不应该提出

---

<sup>5</sup> “软件检查的进展”,IEEE 软件工程学报,SE-12(7): 744–751,1986 年 7 月。

诚然,这项研究是在强大的工具和自动化测试在软件开发过程中变得如此重要之前进行的,但其结果在现代软件时代仍然似乎具有现实意义。

<sup>6</sup> Rigby, Peter C. 和 Christian Bird.2013 年。“融合软件同行评审实践。”ESEC/FSE 2013:2013 年第 9 届软件工程基础联合会议论文集,2013 年 8 月:202–212.<https://dl.acm.org/doi/10.1145/2491411.2491444>。

由于个人意见而放弃替代方案。审阅者可以提出替代方案,但前提是这些替代方案可以提高理解力(例如,通过降低复杂性)或功能性(例如,通过提高效率)。一般来说,鼓励工程师批准改进代码库的更改,而不是等待对更“完美”的解决方案达成共识。这种关注往往会加快代码审阅速度。

随着工具变得越来越强大,许多正确性检查都通过静态分析和自动测试等技术自动执行(尽管工具可能永远不会完全消除人工代码检查的价值。有关详细信息,请参阅[第 20 章](#))。虽然这种工具有其局限性,但它确实减少了依赖人工代码审查来检查代码的需要。

正确性。

尽管如此,在初始代码审查过程中检查缺陷仍然是一般“左移”策略不可或缺的一部分,旨在尽早发现和解决问题,以便它们不需要在开发周期的后期增加成本和资源。代码审查既不是万能药,也不是检查此类正确性的唯一方法,但它是针对软件中此类问题的纵深防御的一个要素。因此,代码审查不需要“完美”才能取得成果。

令人惊讶的是,检查代码正确性并不是 Google 从代码审查过程中获得的主要好处。检查代码正确性通常可以确保更改有效,但更重要的是确保代码更改易于理解,并且随着时间的推移和代码库本身的扩展变得有意义。

为了评估这些方面,我们需要考虑其他因素,而不仅仅是代码在逻辑上是否“正确”或是否可理解。

代码理解代码审查通常是作者以外的

人第一次检查更改的机会。这种视角允许审查者做一些甚至最好的工程师都无法做到的事情:提供不受作者观点影响的反馈。代码审查通常是对给定更改是否可被更广泛的受众理解的首次测试。这种视角至关重要,因为代码的阅读次数比编写次数多得多,理解和领悟至关重要。

找到一个与作者有不同观点的审阅者通常很有用,尤其是可能需要在工作中维护或使用变更中提出的代码的审阅者。与审阅者在设计决策方面给予作者的尊重不同,使用“顾客永远是对的”这一格言来处理代码理解问题通常很有用。在某种程度上,你现在收到的任何问题都会随着时间的推移成倍增加,因此将每个关于代码理解的问题视为有效的。这并不意味着你需要改变你的

方法或逻辑来回应批评,但这确实意味着你可能需要更清楚地解释它。

总之,代码正确性和代码理解性检查是另一位工程师评定 LGTM 的主要标准,LGTM 是代码审核通过所需的批准点之一。当工程师将代码审核标记为 LGTM 时,他们表示代码符合要求,并且易于理解。但是,Google 还要求代码得到可持续维护,因此在某些情况下我们需要对代码进行额外的批准。

#### 代码一致性在规模化的情况

下,您编写的代码将由其他人依赖并最终由其他人维护。许多其他人将需要阅读您的代码并了解您所做的事情。

其他人(包括自动化工具)可能需要在您转移到另一个项目很久之后重构您的代码。因此,代码需要符合一些一致性标准,以便能够理解和维护。代码还应避免过于复杂;更简单的代码也更容易被其他人理解和维护。审阅者可以在代码审查期间评估此代码是否符合代码库本身的标准。因此,代码审查应该采取行动来确保代码健康。

出于可维护性考虑,代码审查的 LGTM 状态(表示代码正确性和可理解性)与可读性批准状态是分开的。只有成功完成特定编程语言代码可读性培训的个人才能授予可读性批准。例如,Java 代码需要获得“Java 可读性”工程师的批准。

可读性审批者的任务是审查代码,以确保它遵循该特定编程语言的商定最佳实践,与 Google 代码库中该语言的代码库一致,并避免过于复杂。一致且简单的代码更容易理解,并在重构时更容易让工具更新,从而使其更具弹性。如果代码库中的特定模式始终以一种方式完成,则编写工具对其进行重构会更容易。

此外,代码可能只写一次,但会被阅读数十次、数百次甚至数千次。在整个代码库中保持一致的代码可以提高整个工程的理解力,这种一致性甚至会影响代码审查过程本身。一致性有时与功能相冲突;可读性审查者可能更喜欢不太复杂的更改,这种更改可能在功能上不是“更好”,但更容易理解。

有了更一致的代码库,工程师就更容易介入并审查其他人的项目代码。工程师可能偶尔需要看看项目之外的东西

团队在代码审查方面寻求帮助。能够联系专家并请他们审查代码，同时知道代码本身是一致的，这使得这些工程师能够更专注于代码的正确性和理解性。

### 心理和文化益处

#### 代码审查还具有重要的文化益

处：它向软件工程师强调，代码不是“他们的”，而是集体事业的一部分。这种心理益处可能很微妙，但仍然很重要。如果没有代码审查，大多数工程师自然会倾向于个人风格和他们自己的软件设计方法。代码审查过程迫使作者不仅让其他人参与，而且为了更大的利益而妥协。

人类天性就是为自己的技术感到自豪，不愿意将自己的代码向他人批评。对于自己编写的代码，人们有些不愿意接受批评性反馈，这也是很自然的。代码审查流程提供了一种机制，可以缓和原本充满情绪的互动。在最佳状态下，代码审查不仅可以挑战工程师的假设，而且可以以一种既定的中立方式进行，可以缓和任何可能以未经请求的方式针对作者的批评。毕竟，这个过程需要批判性审查（事实上，我们将我们的代码审查工具称为“批评”），所以你不能责怪审查人员尽职尽责，提出批评。因此，代码审查流程本身可以充当“坏警察”，而审查人员仍然可以被视为“好警察”。

当然，并非所有工程师，甚至大多数工程师都需要这样的心理手段。但是，通过代码审查过程来缓冲这种批评，通常可以让大多数工程师更温和地了解团队的期望。许多加入 Google 或新团队的工程师都害怕代码审查。人们很容易认为任何形式的批评性审查都会对一个人的工作表现产生负面影响。但随着时间的推移，几乎所有工程师都会在发送代码审查时期待受到挑战，并开始重视通过此过程提供的建议和问题（尽管不可否认，这有时需要一段时间）。

代码审查的另一个心理益处是验证。即使最有能力的工程师也会患上冒名顶替综合症，并且过于自我批评。像代码审查这样的过程可以验证和认可一个人的工作。通常，这个过程涉及思想交流和知识共享（下一节将介绍），这对审查者和被审查者都有好处。随着工程师在其领域知识方面的成长，他们有时很难得到关于他们如何改进的积极反馈。代码审查过程可以提供这种机制。

启动代码审查的过程也迫使所有作者对他们的更改多加小心。许多软件工程师都不是完美主义者；大多数人会承认，“完成工作”的代码比完美但需要太多时间的代码要好。

开发需要很长时间。如果没有代码审查,我们中的许多人自然会偷工减料,即使完全打算稍后纠正这些缺陷。“当然,我没有完成所有的单元测试,但我可以稍后再做。”代码审查迫使工程师在发送更改之前解决这些问题。收集更改的组件进行代码审查在心理上迫使工程师确保所有事情都井然有序。发送更改之前的短暂反思是通读更改并确保没有遗漏任何内容的最佳时机。

### 知识共享代码审查最重要的好

处之一就是知识共享,但这一点却被低估了。大多数作者都会选择在审查领域内具有专业知识或至少知识渊博的审阅者。审查过程允许审阅者向作者传授领域知识,允许审阅者向作者提供建议、新技术或咨询信息。(审阅者甚至可以将某些评论标记为“仅供参考”,无需采取任何行动;它们只是作为对作者的帮助而添加的。)在代码库的某个领域特别精通的作者通常也会成为所有者,然后他们又可以担任其他工程师的审阅者。

代码审查反馈和确认过程的一部分涉及询问为什么以特定方式进行更改。这种信息交换促进了知识共享。事实上,许多代码审查都涉及双向信息交换:作者和审阅者都可以从代码审查中学习新技术和模式。在 Google,审阅者甚至可以直接在代码审查工具中与作者分享建议的编辑。

工程师可能不会阅读发给他们的每封电子邮件,但他们往往回复发给他们的每封代码审查邮件。这种知识共享也可以跨时区和跨项目进行,利用 Google 的规模将信息快速传播给代码库各个角落的工程师。代码审查是知识传递的最佳时机:它及时且可操作。(Google 的许多工程师都是通过代码审查“认识”其他工程师的!)

考虑到 Google 工程师在代码审查上花费的时间,他们积累的知识相当可观。当然,Google 工程师的主要任务仍然是编程,但他们的大部分时间仍然花在代码审查上。代码审查过程是软件工程师相互交流和交换编码技术信息的主要方式之一。通常,新的模式会在代码审查的背景下进行宣传,有时是通过大规模更改等重构来实现的。

此外,由于每次更改都成为代码库的一部分,因此代码审查可以作为历史记录。任何工程师都可以检查 Google 代码库,确定何时引入了某些特定模式,并在

问题。通常，考古学为工程师提供的见解比原作者和审阅者提供的见解多得多。

## 代码审查最佳实践

无可否认，代码审查会给组织带来摩擦和延迟。这些问题大部分不是代码审查本身的问题，而是他们选择的代码审查实现方式的问题。在 Google，保持代码审查流程顺利运行也不例外，它需要许多最佳实践来确保代码审查值得为该流程付出努力。大多数这些实践都强调保持流程灵活快速，以便代码审查能够适当扩展。

### 礼貌而专业

正如本书“文化”部分所指出的，Google 极力培育一种信任和尊重的文化。这渗透到了我们对代码审查的看法中。例如，软件工程师只需要一位其他工程师的 LGTM 即可满足我们对代码理解的要求。许多工程师发表评论并 LGTM 变更，因为他们知道可以在这些变更完成后提交变更，而无需任何额外的审查。话虽如此，代码审查甚至会给最有能力的工程师带来焦虑和压力。至关重要的是，将所有反馈和批评牢牢地放在专业领域内。

一般来说，审阅者应该听从作者的特定方法，只有在作者的方法有缺陷时才指出替代方案。如果作者可以证明几种方法同样有效，审阅者应该接受作者的偏好。即使在这些情况下，如果在一种方法中发现缺陷，也要将审阅视为一个学习机会（对双方来说！）。所有评论都应严格保持专业性。审阅者应谨慎，不要根据代码作者的特定方法妄下结论。在假设该方法是错误的之前，最好先问一问为什么这样做。

审阅者应及时提供反馈。在 Google，我们期望在 24（工作）小时内收到代码审查的反馈。如果审阅者无法在这段时间内完成审查，那么最好（也是预期）回复说他们至少已经看到了更改并将尽快开始审查。审阅者应避免零零碎碎地回复代码审查。没有什么比从审查中获得反馈、解决它，然后在审查过程中继续获得不相关的进一步反馈更让作者恼火的了。

我们期望审阅者的专业素养，也期望作者的专业素养。请记住，你不是你的代码，你提出的更改不是“你的”，而是团队的。将这段代码签入代码库后，它就不再属于你了。要接受

询问你的方法，并准备好解释为什么你以某种方式做事。请记住，作者的部分责任是确保此代码易于理解且易于维护。

将代码审查中的每个审阅者评论视为 TODO 项非常重要；特定评论可能不需要毫无疑问地被接受，但至少应该得到解决。如果您不同意审阅者的评论，请告知他们，并让他们知道原因，并且在双方都有机会提供替代方案之前不要将评论标记为已解决。如果作者不同意审阅者的意见，保持此类辩论文明的一种常见方法是提供替代方案并要求审阅者 PTAL（请再看一遍）。请记住，代码审查是审阅者和作者的学习机会。这种洞察力通常有助于减少任何分歧的机会。

同样，如果您是代码所有者，并且正在响应代码库中的代码审查，请接受外部作者的更改。只要更改是对代码库的改进，您仍应尊重作者的意见，即更改表明可以且应该改进某些内容。

#### 编写小改动保持代码审查流程灵

活的最重要做法可能是将改动保持在较小范围内。理想情况下，代码审查应该易于理解并专注于单个问题，无论是对于审查者还是作者而言。Google 的代码审查流程不鼓励由完整项目组成的大规模更改，审查者可以理所当然地拒绝此类更改，因为它们太大而无法一次审查完成。较小的改动还可以防止工程师浪费时间等待较大改动的审查，从而减少停机时间。这些小改动对软件开发流程的后续阶段也有好处。如果特定改动足够小以缩小范围，则确定改动中错误的来源要容易得多。

话虽如此，但必须承认，依赖于小改动的代码审查流程有时很难与主要新功能的引入相协调。一组小的增量代码更改可能更容易单独消化，但在更大的方案中更难理解。谷歌的一些工程师承认并不喜欢小改动。存在管理此类代码更改的技术（在集成分支上进行开发，使用不同于 HEAD 的 diff 基础管理更改），但这些技术不可避免地会带来更多开销。将小改动的优化视为优化，并让您的流程适应偶尔出现的较大更改。

“小”变更通常应限制在 200 行代码左右。小变更对审阅者来说应该很容易，而且同样重要的是，不应太麻烦，以致于其他变更被推迟等待广泛的审阅。大多数变更

Google 预计在一天左右的时间内完成审核。<sup>7</sup>（这并不一定意味着审核在一天内结束，但初步反馈会在一天内提供。）Google 大约 35% 的更改都是针对单个文件的。<sup>8</sup>对审核者宽容可以更快地更改代码库，同时也使作者受益。

作者希望快速审核；等待一周左右的全面审核可能会影响后续更改。初步的小规模审核还可以防止后续因错误方法而浪费更多昂贵的精力。

由于代码审查通常规模较小，因此 Google 几乎所有的代码审查都由一个人进行审查。如果不是这样（如果希望团队对通用代码库的所有更改发表意见），那么流程本身就无法扩展。通过保持代码审查规模较小，我们实现了这种优化。多人对任何更改发表评论的情况并不少见（大多数代码审查都会发送给团队成员，但也会抄送给相应的团队），但主要审查者仍然是需要其 LGTM 的人，并且任何更改只需要一个 LGTM。任何其他评论虽然也很重要，但仍然是可选的。

保持较小的变更量还可以让“审批”审阅者更快地批准任何变更。他们可以快速检查主要代码审阅者是否进行了尽职调查，并专注于此变更是否增强了代码库，同时保持了代码的健康。

**撰写良好的变更描述** 变更描述应在第一行以

摘要形式指明变更类型。第一行是主要内容，用于在代码审查工具本身中提供摘要，用作任何相关电子邮件的主题行，并成为 Google 工程师在代码搜索中的历史摘要中看到的可见行（参见第 17 章），因此第一行很重要。

尽管第一行应该是整个更改的摘要，但描述仍应详细说明更改的内容及其原因。“错误修复”的描述对审阅者或未来的代码考古学家没有帮助。如果在更改中进行了多项相关修改，请将它们列在列表中（同时保持列表简洁明了）。描述是此更改的历史记录，而诸如代码搜索之类的工具可让您查找代码库中任何特定更改中谁写了哪一行。尝试修复错误时，深入了解原始更改通常很有用。

---

<sup>7</sup> Caitlin Sadowski、Emma Söderberg、Luke Church、Michal Sipko 和 Alberto Bacchelli，“[现代代码审查：谷歌的一个案例研究](#)。”

<sup>8</sup> 同上。

描述并不是为变更添加文档的唯一机会。

编写公共 API 时,通常不想泄露实现细节,但一定要在实际实现中泄露,此时应该自由地进行注释。如果审阅者不理解你为什么做某事,即使它是正确的,这也表明此类代码需要更好的结构或更好的注释(或两者兼而有之)。如果在代码审查过程中做出了新的决定,请更新更改描述或在实现中添加适当的注释。代码审查不仅仅是你现在做的事情;它是为了记录你所做的事情以供后人参考。

将审阅者保持在最低限度Google 的大多

数代码审查都只由一名审阅者进行。<sup>9</sup>因为代码审查流程允许一个人处理代码的正确性、所有者的接受度和语言的可读性,所以代码审查流程在 Google 规模的组织中可以很好地扩展。

行业和个人都倾向于尝试从各个领域的工程师那里获得额外的意见(和一致同意)。毕竟,每个额外的审阅者都可以为相关的代码审查添加他们自己的独特见解。但我们发现这会导致收益递减;最重要的 LGTM 是第一个,而后续的 LGTM 对等式的贡献并不像您想象的那么多。额外审阅者的成本很快就会超过其价值。

代码审查流程的优化基于我们对工程师的信任,相信他们能正确行事。在某些情况下,让多人审查特定更改可能很有用,但即使在这些情况下,这些审查者也应该关注同一更改的不同方面。

## 尽可能实现自动化

代码审查是一个人工过程,人工输入很重要,但如果代码流程中有可以自动化的部分,请尝试这样做。应该探索自动化机械人工任务的机会;投资适当的工具可以获得回报。在 Google,我们的代码审查工具允许作者在批准后自动提交更改并自动同步到源代码控制系统(通常用于相当简单的更改)。

过去几年自动化方面最重要的技术改进之一是对给定代码更改的自动静态分析(参见第 20 章)。

当前的 Google 代码审查工具不需要作者运行测试、linters 或格式化程序,而是通过以下方式自动提供大部分实用程序:

---

<sup>9</sup>同上。

称为预提交。当更改最初发送给审阅者时,会运行预提交过程。在发送更改之前,预提交过程可以检测现有更改的各种问题,拒绝当前更改 (并避免向审阅者发送尴尬的电子邮件),并要求原作者先修复更改。这种自动化不仅有助于代码审阅过程本身,还允许审阅者专注于比格式更重要的问题。

## 代码审查的类型

所有的代码审查都不一样!不同类型的代码审查需要对审查过程的各个方面给予不同程度的关注。Google 的代码更改通常属于以下类别之一 (尽管有时会重叠) :

- 绿地评审和新功能开发 · 行为变化、改进和优化 · 错误修复和回滚 · 重构和大规模更改

### Greeneld 代码评审

最不常见的代码审查类型是全新代码的审查,即所谓的绿地审查。绿地审查是评估代码是否经得起时间考验的最重要的时机:随着时间和规模改变代码的基本假设,代码将更易于维护。当然,引入全新代码并不令人意外。如本章前面所述,代码是一种负担,因此引入全新代码通常应该解决实际问题,而不仅仅是提供另一种替代方案。在 Google,除了代码审查之外,我们通常要求新代码和 / 或项目进行广泛的设计审查。代码审查不是讨论过去已经做出的设计决策的时候 (同样,代码审查也不是介绍拟议 API 的设计的时候)。

为了确保代码的可持续性,绿地评审应确保 API 符合商定的设计 (这可能需要评审设计文档) 并经过全面测试,所有 API 端点都具有某种形式的单元测试,并且当代码的假设发生变化时,这些测试会失败。(参见第 11 章)。代码还应有适当的所有者 (新项目中的首次评审之一通常是新目录的单个 OWNERS 文件),有足够的注释,并在需要时提供补充文档。绿地评审可能还需要将项目引入持续集成系统。(参见第 23 章)。

行为变化、改进和优化Google 的大多数变化通常属于对代码库中现有代码的广泛修改。这些添加可能包括对 API 端点的修改、对现有实现的改进或对性能等其他因素的优化。这些变化是大多数软件工程师的日常工作。

在每种情况下，适用于全新审查的指导原则也适用：这种更改是否必要？这种更改是否会改善代码库？对代码库的一些最佳修改实际上是删除！删除死代码或过时代码是改善代码库整体代码健康状况的最佳方法之一。

任何行为修改都应包含对任何新 API 行为的适当测试的修订。应在持续集成 (CI) 系统中测试对实现的增强，以确保这些修改不会破坏现有测试的任何基本假设。此外，优化当然应确保它们不会影响这些测试，并且可能需要包含性能基准以供审阅者参考。某些优化可能还需要基准测试。

错误修复和回滚不可避免地，您需

要提交更改以修复代码库中的错误。提交更改时，请避免解决其他问题的诱惑。这不仅会增加代码审查的规模，还会使执行回归测试或让其他人回滚您的更改变得更加困难。错误修复应仅专注于修复指示的错误并（通常）更新相关测试以捕获最初发生的错误。

经常需要通过修改测试来解决错误。错误出现是因为现有测试不充分，或者代码中的某些假设没有得到满足。作为错误修复的审阅者，如果适用，要求更新单元测试非常重要。

有时，像 Google 这样庞大的代码库中的代码更改会导致某些依赖项失败，而这些依赖项要么无法通过测试正确检测到，要么会暴露出代码库中未经测试的部分。在这种情况下，Google 允许此类更改“回滚”，通常由受影响的下游客户执行。回滚包括实际上撤消先前更改的更改。此类回滚可以在几秒钟内创建，因为它们只是将先前的更改恢复到已知状态，但仍需要代码审查。

同样至关重要的是，任何可能导致潜在回滚的更改（包括所有更改！）都应尽可能小且原子化，以便在需要时进行回滚不会导致可以

很难理清。在 Google,我们看到开发人员在提交新代码后很快就开始依赖新代码,因此回滚有时会让这些开发人员感到困扰。小改动有助于缓解这些问题,因为它们具有原子性,而且对小改动的审查往往很快完成。

重构和大规模变更Google 的许多变更都是自动生成

的:变更的作者不是人,而是机器。我们将在**第 22 章**中进一步讨论大规模变更 (LSC) 流程,但即使是机器生成的变更也需要审核。如果变更被认为风险较低,则由拥有整个代码库批准权限的指定审核人员进行审核。但对于变更可能存在风险或需要本地领域专业知识的情况,可能会要求个别工程师作为其正常工作流的一部分审核自动生成的变更。

乍一看,对自动生成的变更的审查应与任何其他代码审查一样处理:审查人员应检查变更的正确性和适用性。但是,我们鼓励审查人员限制相关变更中的评论,并且仅标记特定于其代码的问题,而不是生成变更的底层工具或 LSC。虽然特定变更可能是机器生成的,但生成这些变更的整体流程已经过审查,各个团队不能否决该流程,否则无法在整个组织范围内推广此类变更。如果对底层工具或流程存在疑虑,审查人员可以向 LSC 监督小组上报以获取更多信息。

我们还鼓励自动变更的审核者避免扩大其范围。

在审查新功能或团队成员编写的更改时,要求作者在同一更改中解决相关问题通常是合理的,只要请求仍然遵循先前的建议,将更改保持在较小的范围内。这并不适用于自动生成的更改,因为运行该工具的人可能会进行数百次更改,即使是一小部分带有评论或不相关问题的更改也会限制人们有效操作该工具的规模。

## 结论

代码审查是 Google 最重要、最关键的流程之一。代码审查是将工程师们联系在一起的粘合剂,代码审查流程是开发人员的主要工作流程,几乎所有其他流程 (从测试到静态分析再到 CI) 都必须依赖它。代码审查流程必须适当扩展,因此,最佳实践 (包括小改动、快速反馈和迭代) 对于保持开发人员的满意度和适当的生产速度非常重要。

## TL;DR

- 代码审查有很多好处,包括确保代码的正确性、一致性和整个代码库的一致性。
- 始终通过其他人检查您的假设;为读者进行优化。 · 在保持专业性的同时提供批判性反馈的机会。 · 代码审查对于整个组织的知识共享非常重要。 · 自动化对于扩展流程至关重要。 · 代码审查本身提供了历史记录。



## 第十章

# 文档

作者:汤姆·曼什雷克 (Tom Mansreck)  
编辑:Riona MacNamara

大多数工程师对编写、使用和维护代码的抱怨中,最普遍的挫败感就是缺乏高质量的文档。“这种方法有什么副作用?”“我在第 3 步之后出现错误。”“这个首字母缩略词是什么意思?”

“这个文档是最新的吗?”每一个软件工程师在其职业生涯中都曾抱怨过文档的质量、数量或完全缺乏,谷歌的软件工程师也不例外。

技术作家和项目经理可能会有所帮助,但软件工程师始终需要自己编写大多数文档。因此,工程师需要适当的工具和激励才能有效地完成这项工作。让他们更容易编写高质量文档的关键是引入与组织规模相匹配并与现有工作流程相结合的流程和工具。

总体而言,2010 年代后期的工程文档状态与 20 世纪 80 年代后期的软件测试状态类似。每个人都认识到需要付出更多努力来改进它,但组织尚未认识到它的关键优势。这种情况正在改变,尽管进展缓慢。在 Google,我们最成功的努力是将文档视为代码并纳入传统工程工作流程,使工程师更容易编写和维护简单的文档。

## 什么才算是合格的文档?

当我们提到“文档”时,我们指的是工程师在工作中需要编写的所有补充文本:不仅是独立文档,还有代码注释。(事实上,谷歌工程师编写的大部分文档

以代码注释的形式出现。我们将在本章进一步讨论各种类型的工程文档。

## 为什么需要文档？

质量文档对于工程组织来说具有巨大的益处。

代码和 API 变得更易于理解,从而减少错误。当项目团队的设计目标和团队目标明确时,他们会更加专注。当步骤清晰时,手动流程更容易遵循。如果流程记录清晰,那么让新成员加入团队或代码库所需的努力就会少得多。

但是,由于文档的好处必然都是下游的,因此它们通常不会立即给作者带来好处。与测试不同(正如我们将看到的),测试会迅速为程序员带来好处,而文档通常需要更多的前期工作,并且直到后来才会给作者带来明显的好处。但是,就像在测试方面的投资一样,在文档方面的投资会随着时间的推移而得到回报。毕竟,您可能只写过一次文档,但之后会被阅读数百次,甚至数千次;它的初始成本会分摊到所有未来的读者身上。文档不仅可以随着时间的推移而扩展,而且对于组织的其他部分来说,扩展也是至关重要的。它有助于回答以下问题:

- 为什么要做出这些设计决策? · 为什么  
我们以这种方式实现此代码? · 为什么我以这种方式  
实现此代码,如果你正在查看自己的  
两年后的代码?

如果文档传达了所有这些好处,为什么工程师普遍认为它“不好”呢?正如我们提到的,其中一个原因是,好处不是立竿见影的,尤其是对作者而言。但还有其他几个原因:

- 工程师通常将写作视为一项与编程不同的技能。(我们将尝试说明事实并非如此,即使确实如此,它也不一定是一项与软件工程不同的技能。) · 一些工程师觉得自己不是有能力的作家。但您不需要熟练掌握英语<sup>2</sup>即可编写可行的  
文档。您只需要跳出自我,从受众的角度看问题。

---

<sup>1</sup>好的,您需要维护它并偶尔修改它。

<sup>2</sup>英语仍然是大多数程序员的主要语言,大多数程序员的技术文档  
mers 依赖于对英语的理解。

- 由于工具支持有限或无法集成到开发人员工作流程中,编写文档通常更加困难。 · 文档被视为额外的负担  
(需要维护的其他内容),而不是使现有代码的维护  
更容易的内容。

并非每个工程团队都需要一名技术作家（即使如此,他们的数量也不够）。这意味着工程师将在很大程度上自己编写大部分文档。因此,我们不应该强迫工程师成为技术作家,而应该考虑如何让工程师更轻松地编写文档。决定在文档上投入多少精力是您的组织在某个时候需要做出的决定。

文档对不同的群体都有好处。即使对于作家来说,文档也有以下好处:

- 它有助于制定 API。编写文档是确定 API 是否合理的最可靠方法之一。通常,文档本身的编写会让工程师重新评估原本不会受到质疑的设计决策。如果你无法解释它,无法定义它,那么你可能设计得不够好。
- 它提供了维护的路线图和历史记录。无论如何,应该避免在代码中使用花招,但是当你盯着两年前写的代码,试图找出错误时,好的注释会大有帮助。 · 它使你的代码看起来更专业,并带来流量。开发人员会自然而然地认为,一个有良好文档的 API 就是一个设计更好的 API。情况并非总是如此,但它们往往高度相关。虽然这种好处听起来很肤浅,但事实并非如此:产品是否有良好的文档通常是产品维护程度的一个相当好的指标。
- 它会减少其他用户的提问。这可能是对编写文档的人来说,随着时间的推移最大的好处。如果你必须向某人解释某件事不止一次,那么记录下来通常是有意义的

过程。

虽然这些好处对于文档编写者来说意义重大,但文档的大部分好处自然会归于读者。Google 的 C++ 风格指南提到了“[为读者优化”的格言](#)。这条格言不仅适用于代码,也适用于代码周围的注释或附加到 API 的文档集。与测试非常相似,您在编写优质文档方面所付出的努力将在其生命周期内带来多次收益。文档在长期内至关重要,并且随着组织规模的扩大,文档会为特别重要的代码带来巨大的收益。

## 文档就像代码

使用单一主要编程语言编写的软件工程师仍然经常使用不同的语言来解决特定问题。工程师可能会编写 shell 脚本或 Python 来运行命令行任务,或者他们可能会用 C++ 编写大部分后端代码,但用 Java 编写一些中间件代码,等等。每种语言都是工具箱中的一个工具。

文档也应该如此:它是一种工具,用不同的语言(通常是英语)编写,用于完成特定任务。编写文档与编写代码没有太大区别。与编程语言一样,它有规则、特定的语法和风格决定,通常是为了实现与代码中类似的目的:加强一致性、提高清晰度和避免(理解)错误。在技术文档中,语法很重要,不是因为需要规则,而是为了标准化语气,避免让读者感到困惑或分心。出于这个原因,Google 要求其许多语言都采用一定的注释风格。

和代码一样,文档也应该有所有者。没有所有者的文档会变得陈旧且难以维护。明确的所有权还可以使通过现有开发人员工作流程处理文档变得更加容易:错误跟踪系统、代码审查工具等。当然,不同所有者的文档仍然可能相互冲突。在这些情况下,指定规范文档很重要:确定主要来源并将其他相关文档合并到该主要来源(或弃用重复项)。

Google 广泛使用“go/链接”(参见第 3 章)使得这个过程变得更加容易。

带有直接 go/ 链接的文档通常会成为权威的真相来源。推广权威文档的另一种方法是将它们直接与它们所记录的代码相关联,方法是将它们直接置于源代码控制之下并与源代码本身放在一起。

文档通常与代码紧密结合,因此应尽可能将其视为代码。也就是说,您的文档应该:

- 有内部政策或规则需要遵守 · 置于源代码控制之下
- 有明确的所有权负责维护文档 · 接受变更评审(并随其记录的代码一起变更) · 跟踪问题,就像跟踪代码中的错误一样
- 定期评估(在某些方面进行测试) · 如果可能,对准确性、新鲜度等方面进行衡量(工具在这里还没有赶上)

工程师越是将文档视为软件开发“必要任务之一”，他们就越不会对编写文档的前期成本心存怨恨，而他们获得的长期利益也越多。此外，简化文档编写任务可以减少这些前期成本。

#### 案例研究:Google Wiki当 Google 规模

较小、精简时，几乎没有技术作家。共享信息的最简单方法是通过我们自己的内部 wiki (GooWiki)。起初，这似乎是一种合理的方法；所有工程师共享一个文档集，并可以根据需要进行更新。

但随着 Google 的规模扩大，wiki 风格方法的问题开始显现。由于文档没有真正的所有者，许多文档变得过时。<sup>3</sup>由于没有制定添加新文档的流程，重复的文档和文档集开始出现。GooWiki 有一个扁平的命名空间，人们不擅长将任何层次结构应用于文档集。有一段时间，有 7 到 10 份文档（取决于您如何计算它们）关于设置 Borg（我们的生产计算环境），其中只有少数文档似乎得到了维护，大多数文档都是特定于具有特定权限和假设的某些团队的。

随着时间的推移，GooWiki 的另一个问题逐渐显现出来：能够修复文档的人并不是使用文档的人。发现文档有问题的新用户要么无法确认文档是否有误，要么没有简单的方法来报告错误。他们知道出了问题（因为文档不起作用），但他们无法“修复”它。相反，最有能力修复文档的人在文档写好后通常不需要查阅文档。随着 Google 的发展，文档变得如此糟糕，以至于文档质量成为 Google 在我们年度开发人员调查中面临的头号开发人员投诉。

改善这种情况的方法是将重要文档移至用于跟踪代码更改的同一类型的源代码控制下。文档开始拥有自己的所有者、源代码树中的规范位置以及识别和修复错误的流程；文档开始得到显著改善。此外，文档的编写和维护方式开始与代码的编写和维护方式相同。文档中的错误可以在我们的错误跟踪软件中报告。可以使用现有的代码审查流程来处理对文档的更改。最终，工程师开始自己修复文档或将更改发送给技术作家（通常是所有者）。

---

<sup>3</sup>当我们弃用 GooWiki 时，我们发现大约 90% 的文档在前几个月。

将文档移至源代码控制最初引起了很多争议。

许多工程师都相信,废除 GooWiki (信息自由的堡垒)会导致文档质量下降,因为文档的门槛 (需要审核、需要文档所有者等)会更高。但事实并非如此。文档变得更好了。

Markdown 作为通用文档格式化语言的引入也起到了帮助作用,因为它让工程师更容易理解如何编辑文档,而无需 HTML 或 CSS 方面的专业知识。Google 最终推出了自己的在代码中嵌入文档的框架: [g3doc](#)。

有了该框架,文档得到了进一步改进,因为文档与工程师开发环境中的源代码并存。现在,工程师可以在同一次更改中更新代码及其相关文档 (我们仍在努力提高这一做法的采用率)。

关键的区别在于,维护文档的体验与维护代码的体验类似:工程师提交错误、更改变更列表中的文档、将更改发送给专家评审等等。利用现有的开发人员工作流程,而不是创建新的工作流程,是一个关键优势。

## 了解你的受众

工程师在编写文档时犯的一个最重要的错误就是只为自己而写。这样做很自然,为自己而写也并非毫无价值:毕竟,几年后您可能需要查看此代码并尝试弄清楚您曾经的意思。您也可能与阅读您文档的人拥有大致相同的技能。但是,如果您只为自己写作,您将做出某些假设,并且考虑到您的文档可能会被非常广泛的受众 (所有工程师、外部开发人员)阅读,即使失去一些读者也是巨大的代价。随着组织的发展,文档中的错误变得更加突出,而您的假设通常不再适用。

相反,在开始写作之前,你应该 (正式或非正式地)确定你的文档需要满足的受众。设计文档可能需要说服决策者。教程可能需要向完全不熟悉代码库的人提供非常明确的说明。API 可能需要为该 API 的任何用户 (无论是专家还是新手)提供完整而准确的参考信息。始终尝试确定主要受众并针对该受众进行写作。

好的文档不需要精致或“完美”。工程师在编写文档时犯的一个错误是假设他们需要成为更好的作家。按照这个标准,很少有软件工程师会写作。像你对待测试或你作为工程师需要做的任何其他过程一样考虑写作。用你的读者期望的语气和风格来写作。如果你能读,你就能写。记住

你的读者现在正站在你曾经站过的地方,只是没有你的新领域知识。所以你不需要成为一名伟大的作家;你只需要找到一个像你一样熟悉这个领域的人。(只要你扎根于此,你就可以随着时间的推移改进这份文档。)

读者类型我们已经指出,你应

该根据你的读者的技能水平和领域知识进行写作。但你的读者究竟是谁?根据以下一个或多个标准,你很可能有多个读者:

- 经验水平 (专业程序员,或者可能不熟悉该语言的初级工程师)。 · 领域知识 (团队成员或组织中的其他工程师
  
- 仅熟悉 API 端点)。
- 目的 (最终用户可能需要您的 API 来执行特定任务并需要快速找到该信息,或者软件专家负责您希望不需要其他人维护的特别复杂的实现的核心)。

在某些情况下,不同的读者需要不同的写作风格,但在大多数情况下,诀窍是尽可能广泛地适用于不同的读者群体。通常,您需要向专家和新手解释一个复杂的主题。为具有领域知识的专家写作可能会让您走捷径,但您会让新手感到困惑;相反,向新手详细解释一切无疑会让专家感到恼火。

显然,编写此类文档需要权衡利弊,没有灵丹妙药,但我们发现,保持文档简短是有帮助的。编写描述性的内容,以便向不熟悉该主题的人解释复杂的问题,但不要让专家感到困惑或恼怒。编写简短的文档通常需要您编写更长的文档(将所有信息写下来),然后进行编辑,尽可能删除重复的信息。这可能听起来很乏味,但请记住,这种成本会分摊到文档的所有读者身上。正如 Blaise Pascal 曾经说过的那样,“如果我有更多时间,我会给你写一封更短的信。”通过保持文档简明了,您将确保它既能满足专家又能满足

新手。

另一个重要的受众区别是基于用户如何遇到文档:

- 寻求者是工程师,他们知道自己想要什么,也想知道他们所寻找的东西是否符合要求。对这类受众来说,一个关键的教学手段是一致性。如果你正在为这一群体编写参考文档 在

代码文件 你需要让你的注释遵循类似的格式,以便读者可以快速浏览参考资料,看看他们是否找到了他们正在寻找的内容。 · 读者可能并不确切知道他们想要什么。他们可能对如何实现他们正在使用的东西只有一个模糊的概念。对于这类读者

来说,关键是清晰。提供概述或介绍(例如,在文件顶部),解释他们正在查看的代码的用途。确定文档何时不适合某些读者也很有用。Google 的许多文档都以“TL;DR 语句”开头,例如“TL;DR:如果您对 Google 的 C++ 编译器不感兴趣,您可以停止阅读。”

最后,一个重要的受众区别是客户(例如,API 的用户)和提供者(例如,项目团队成员)。尽可能将针对一方的文档与针对另一方的文档分开。实现细节对于团队成员来说很重要,因为这样便于维护;最终用户不需要阅读此类信息。通常,工程师会在他们发布的库的参考 API 中表示设计决策。这些推理更适合放在特定文档(设计文档)中,或者最好放在隐藏在接口后面的代码的实现细节中。

## 文档类型

工程师在工作中会编写各种不同类型的文档:设计文档、代码注释、操作方法文档、项目页面等等。这些都算作“文档”。但重要的是要了解不同的类型,不要混淆类型。一般来说,文档应该有一个单一的目的,并坚持这个目的。

正如 API 应该只做一件事并做好一样,避免尝试在一个文档中做几件事。相反,应该更合乎逻辑地将这些部分分开。

软件工程师经常需要撰写的文档主要有以下几种类型:

- 参考文档,包括代码注释 · 设计文档 · 教程
  
- 概念文档 · 登陆页面

在 Google 早期,团队通常会拥有庞大的 wiki 页面,其中包含大量链接(许多链接已损坏或已过时)、一些关于系统如何工作的概念信息、API 参考等,所有这些都散落在一起。这样的文档失败了,因为它们没有单一用途(而且它们也太长了)

没有人会阅读它们;一些臭名昭著的 wiki 页面需要滚动几十屏才能看到。相反,请确保您的文档具有单一目的,并且如果在该页面中添加某些内容没有意义,您可能需要为此目的查找甚至创建另一个文档。

## 参考文档

参考文档是工程师需要编写的最常见的文档类型;事实上,他们经常每天都需要编写某种形式的参考文档。参考文档是指记录代码库中代码使用情况的任何内容。代码注释是工程师必须维护的最常见的参考文档形式。这类注释可以分为两大类:API 注释与实现注释。请记住这两者之间的受众差异:API 注释不需要讨论实现细节或设计决策,也不能假设用户与作者一样精通 API。另一方面,实现注释可以假设读者拥有更多的领域知识,但要小心不要假设太多:人们会离开项目,有时更有条理地说明你为什么以这种方式编写此代码会更安全。

大多数参考文档,即使作为与代码分开的文档提供,也是从代码库本身的注释生成的。(参考文档应该尽可能地单一来源。)某些语言(如 Java 或 Python)具有特定的注释框架(Javadoc,PyDoc,GoDoc),旨在使此类参考文档的生成更加容易。其他语言(如 C++)没有标准的“参考文档”实现,但由于 C++ 将其 API 表面(在头文件或.h 文件中)与实现(.cc 文件)分开,因此头文件通常是记录 C++ API 的自然位置。

Google 采取了这种方法:C++ API 应该将其参考文档放在头文件中。其他参考文档也直接嵌入在 Java,Python 和 Go 源代码中。由于 Google 的代码搜索浏览器(参见第 17 章)非常强大,我们发现提供单独生成的参考文档几乎没有好处。代码搜索中的用户不仅可以轻松搜索代码,而且通常可以在最顶部找到该代码的原始定义。将文档与代码定义放在一起也使文档更容易发现和维护。

我们都知道,代码注释对于文档齐全的 API 至关重要。但究竟什么是“好的”注释呢?在本章前面,我们确定了参考文档的两大受众:搜索者和偶然发现者。搜索者知道自己想要什么;偶然发现者不知道。对于搜索者来说,关键优势是一致的注释代码库,这样他们就可以快速扫描 API 并找到他们要找的东西。对于偶然发现者来说,关键优势是清楚地标识 API 的用途,通常位于文件顶部。

标头。我们将在后面的小节中介绍一些代码注释。后面的代码注释指南适用于 C++，但 Google 也为其他语言制定了类似的规则。

## 文件注释

几乎所有 Google 的代码文件都必须包含文件注释。（某些只包含一个实用函数等的头文件可能不符合此标准。）文件注释应以以下格式的标头开头：

```
// ----- // str_cat.h //
----- // 此头文件包含用于高效连接和附加字符串的函数：
```

StrCat() 和 StrAppend()。这些例程中的大部分工作实际上是通过使用特殊的 AlphaNum 类型来处理的，该类型旨在用作参数类型，以高效管理到字符串的转换并避免上述操作中的复制。

...

通常，文件注释应该以您正在阅读的代码中包含的内容的概述开始。它应该标识代码的主要用例和目标受众（在前面的例子中，是想要连接字符串的开发人员）。任何无法在第一段或第二段中简洁描述的 API 通常都是没有经过深思熟虑的 API 的标志。在这些情况下，请考虑将 API 拆分为单独的组件。

## 课堂评论

大多数现代编程语言都是面向对象的。因此，类注释对于定义代码库中使用的 API “对象”非常重要。Google 的所有公共类（和结构）都必须包含类注释，描述类/结构、该类的重要方法以及该类的用途。通常，类注释应该“名词化”，文档强调其对象方面。也就是说，“Foo 类包含 x,y,z，允许您执行 Bar，并具有以下 Baz 方面”，等等。

课程评论通常应以下形式的评论开头：

```
// ----- // AlphaNum //
----- // // AlphaNum 类作为 StrCat() 和 // StrAppend() 的
主
要参数类型，提供将数字、布尔值和 // 十六进制值（通过 Hex 类型）高效地转换为字符串。
```

## 函数注释

Google 的所有自由函数或类的公共方法都必须包含函数注释,描述函数的功能。函数注释应强调其使用的主动性,以陈述动词开头,描述函数的功能和返回的内容。

函数注释通常应以下形式的注释开头:

```
// StrCat() // //
合
并给定的字符串或数字,不使用分隔符, // 以字符串形式返回合并结果。
```

...

请注意,以声明性动词开头的函数注释可确保整个头文件的一致性。搜索者可以快速扫描 API 并只阅读动词以了解该函数是否合适:“合并、删除、创建”等等。

一些文档样式(和一些文档生成器)要求在函数注释中使用各种形式的样板代码,如“Returns:”、“Throws:”等等,但在 Google,我们发现这些代码不是必需的。在单个散文注释中呈现此类信息通常更清晰,而不会被分成人为的部分边界:

```
// 为具有给定名称和地址的客户创建新记录, // 并返回记录 ID,或者如果具有该名称的记录已经存在,则抛出
“DuplicateEntryError” .int AddCustomer (string name,string address) ;
```

请注意,后置条件、参数、返回值和异常情况是如何自然地记录在一起的(在本例中,用一个句子),因为它们彼此并不独立。添加明确的样板部分会使注释更加冗长和重复,但不会更清晰(甚至可以说更不清晰)。

## 设计文档

Google 的大多数团队在开始任何重大项目之前都要求有一份经过批准的设计文档。软件工程师通常使用团队批准的特定设计文档模板编写拟议的设计文档。此类文档旨在用于协作,因此它们通常在具有良好协作工具的 Google Docs 中共享。一些团队要求在特定的团队会议上讨论和辩论此类设计文档,专家可以在会议上讨论或批评设计的细节。在某些方面,这些设计讨论充当了编写任何代码之前的代码审查形式。

因为设计文档的开发是工程师在部署新系统之前要进行的首要过程之一,所以它也是一个方便的地方来确保

确保涵盖各种问题。Google 的规范设计文档模板要求工程师考虑其设计的各个方面,例如安全隐患、国际化、存储要求和隐私问题等。在大多数情况下,这些设计文档的这些部分由相关领域的专家审查。

一份好的设计文档应该涵盖设计目标、实施策略,并提出关键的设计决策,重点强调各自的权衡。最好的设计文档会提出设计目标,并涵盖备选设计,指出它们的优缺点。

一份好的设计文档一旦获得批准,不仅可作为历史记录,还可作为项目是否成功实现其目标的衡量标准。大多数团队都会将他们的设计文档存档在团队文档中的适当位置,以便以后可以查看。在产品发布之前审查设计文档通常很有用,以确保编写设计文档时所述的目标与发布时所述的目标相同(如果不是,则可以相应地调整文档或产品)。

## 教程

每位软件工程师在加入新团队时,都希望尽快上手。拥有一个教程来指导新项目的设置是非常宝贵的;“Hello World”已经成为确保所有团队成员都能从正确起点开始的最佳方法之一。这适用于文档和代码。大多数项目都应该有一份“Hello World”文档,它不做任何假设,并让工程师实现“真实”的事情。

通常,如果还没有教程,编写教程的最佳时间就是您刚加入团队时。(这也是在您正在关注的任何现有教程中查找错误的最佳时间。)

拿一个记事本或其他方式来做笔记,写下你在此过程中需要做的所有事情,假设没有领域知识或特殊的设置限制;完成后,你可能会知道你在此过程中犯了什么错误以及为什么

然后可以编辑你的步骤以获得更精简的教程。重要的是,写下你在此过程中需要做的所有事情;尽量不要假设任何特定的设置、权限或领域知识。如果你确实需要假设其他设置,请在教程开头明确说明作为一组先决条件。

大多数教程都要求您按顺序执行多个步骤。在这种情况下,请明确编号这些步骤。如果教程的重点是用户(例如,对于外部开发人员文档),则请对用户需要执行的每个操作进行编号。不要对系统可能响应此类用户操作而采取的操作进行编号。

在执行此操作时,明确编号每个步骤至关重要。没有什么比第4步出错更令人恼火的了,因为你忘记告诉某人正确授权他们的用户名。

## 示例 :糟糕的教程

1. 从我们的服务器 `http://example.com` 下载软件包
2. 将 shell 脚本复制到你的主目录
3. 执行 shell 脚本
4. `foobar` 系统将与身份验证系统进行通信
5. 一旦通过身份验证,`foobar` 将启动一个名为 “`baz`” 的新数据库
  
6. 通过在命令行上执行 SQL 命令来测试 “`baz`”
7. 输入 :`CREATE DATABASE my_foobar_db;`

在前面的过程中,步骤 4 和 5 发生在服务器端。目前尚不清楚用户是否需要做任何事情,但实际上他们不需要,因此这些副作用可以作为步骤 3 的一部分提及。此外,目前尚不清楚步骤 6 和步骤 7 是否不同。(它们没有区别。)将所有原子用户操作合并为单个步骤,以便用户知道他们需要在流程的每个步骤中执行某些操作。此外,如果您的教程有用户可见的输入或输出,请在单独的行中表示出来(通常使用等宽粗体字体的惯例)。

## 示例 :让糟糕的教程变得更好

1. 从我们的服务器 `http://example.com` 下载软件包：  
`$ curl -I http://example.com`
2. 将 shell 脚本复制到你的主目录：  
`$ cp foobar.sh ~`
3. 在你的主目录中执行 shell 脚本：  
`$ cd ~; foobar.sh`  
`foobar` 系统首先会与身份验证系统通信。身份验证通过后,`foobar` 将启动一个名为 “`baz`” 的新数据库并打开一个输入 shell。
  
4. 通过在命令行上执行 SQL 命令来测试 “`baz`”：  
`baz:$ 创建数据库 my_foobar_db;`

注意每个步骤如何需要特定的用户干预。如果教程的重点是其他方面(例如,有关“服务器的生命周期”的文档),请从该重点(服务器的作用)的角度对这些步骤进行编号。

### 概念文档有些代码需要更深入的解释

或见解,而仅仅阅读参考文档是无法获得的。在这些情况下,我们需要概念文档来提供 API 或系统的概述。概念文档的一些示例可能是流行 API 的库概述、描述服务器内数据生命周期的文档等等。在几乎所有情况下,概念文档旨在扩充而不是取代参考文档集。这通常会导致一些信息的重复,但目的是:提高清晰度。在这些情况下,概念文档没有必要涵盖所有边缘情况(尽管参考文献应该严格涵盖这些情况)。在这种情况下,为了清晰起见牺牲一些准确性是可以接受的。概念文档的重点是传达理解。

“概念”文档是最难编写的文档形式。因此,它们往往是软件工程师工具箱中最被忽视的文档类型。工程师在编写概念文档时面临的一个问题是,它通常无法直接嵌入源代码中,因为没有规范的位置来放置它。某些 API 具有相对较宽的 API 表面积,在这种情况下,文件注释可能是“概念”解释该 API 的合适位置。但通常,API 与其他 API 和/或模块协同工作。记录这种复杂行为的唯一合理位置是通过单独的概念文档。如果注释是文档的单元测试,那么概念文档就是集成测试。

即使 API 的范围已适当确定,提供单独的概念文档通常也是有意义的。例如,Abseil 的StrFormat 库涵盖了 API 的熟练用户应该理解的各种概念。在这些情况下,无论是内部还是外部,我们都提供格式概念文档。

概念文档需要对广大受众有用:专家和新手都一样。此外,它需要强调清晰度,因此它通常需要牺牲完整性(最好留给参考)和(有时)严格的准确性。这并不是说概念文档应该故意不准确;它只是意味着它应该专注于常见用法,而将罕见用法或副作用留给参考文档。

### 登陆页面大多数工

程师都是团队的成员,大多数团队在公司内联网的某个地方都有一个“团队页面”。通常,这些网站有点混乱:典型的登陆页面可能包含一些有趣的链接,有时包含几个标题为“先读这个!”的文档,以及一些团队和客户的信息。

这些文件一开始很有用,但很快就会变成灾难;因为它们变得

由于维护起来非常麻烦,它们最终会变得过时,只有勇敢者或绝望者才能修复它们。

幸运的是,这些文档看起来令人生畏,但实际上很容易修复:确保着陆页清楚地标明其目的,然后仅包含指向其他页面的链接以获取更多信息。如果着陆页上的某些内容不仅仅是充当交通警察,那么它就没有尽到自己的职责。如果您有单独的设置文档,请从着陆页将其链接到单独的文档。如果着陆页上有太多链接(您的页面不应滚动多个屏幕),请考虑按分类将页面分为不同的部分。

大多数配置不当的登录页面都有两种不同的用途:它们是产品或 API 用户的“转到”页面,或者是团队的主页。不要让页面同时服务于两个主人 这会让人感到困惑。

创建一个单独的“团队页面”作为除主登陆页面之外的内部页面。  
团队需要知道的内容通常与 API 客户需要知道的内容有很大不同。

## 文档评审

在 Google,所有代码都需要审查,我们的代码审查流程得到了很好的理解和认可。一般来说,文档也需要审查(尽管这一点不太被普遍接受)。如果你想“测试”你的文档是否有效,你通常应该让别人来审查它。

一份技术文档受益于三种不同类型的评审,每种评审强调不同的方面:

- 技术审查,确保准确性。此审查通常由主题专家(通常是团队的另一名成员)进行。通常,这是代码审查本身的一部分。
- 为清晰起见,进行受众评审。这通常是不熟悉该领域的人。这可能是您团队的新成员或 API 的客户。
- 为保持一致性,进行写作评审。这通常是技术作家或志愿者。

当然,这些界限有时很模糊,但如果您的文档备受关注或最终可能会对外发布,您可能希望确保它收到更多类型的评论。(我们对这本书使用了类似的审查流程。)

任何文档都会从上述审阅中受益,即使其中一些审阅是临时性的。也就是说,即使只有一位审阅者审阅您的文本也比没有人审阅要好。

重要的是,如果文档与工程工作流程紧密结合,它通常会随着时间的推移而改进。Google 的大多数文档现在都会经过受众审查,因为在某个时候,他们的受众会使用它们,并希望在它们不起作用时让你知道(通过错误或其他形式的反馈)。

#### 案例研究:开发人员指南库如前所述,将大多数(几乎所

有)工程文档包含在共享 wiki 中存在一些问题:重要文档的所有权很少、文档相互竞争、信息过时以及难以提交文档中的错误或问题。但有些文档没有出现这个问题:Google C++ 风格指南由一组精选的高级工程师(风格仲裁者)拥有,他们负责管理它。由于某些人关心它,所以文档保持良好状态。他们隐性拥有该文档。该文档也是规范的:只有一个 C++ 风格指南。

如前所述,直接位于源代码中的文档是促进建立规范文档的一种方式;如果文档与源代码一起存在,通常应该是最适用的(希望如此)。在 Google,每个 API 通常都有一个单独的 g3doc 目录,用于存放此类文档(以 Markdown 文件的形式编写,可在我们的代码搜索浏览器中阅读)。将文档与源代码并存不仅可以确立事实上的所有权,还可以使文档看起来更像是代码的“一部分”。

但是,某些文档集在源代码中无法非常合乎逻辑地存在。例如,面向 Google 员工的“C++ 开发者指南”在源代码中没有明显的位置。没有主“C++”目录供人们查找此类信息。在这种情况下(以及跨越 API 边界的其他情况),在自己的仓库中创建独立的文档集变得很有用。其中许多将相关的现有文档汇总到一个通用集合中,具有通用的导航和外观。此类文档被称为“开发者指南”,并且与代码库中的代码一样,在特定文档仓库中进行源代码控制,此仓库按主题而不是 API 进行组织。通常,技术作家管理这些开发者指南,因为他们更擅长解释跨越 API 边界的主题。

随着时间的推移,这些开发者指南逐渐成为规范。编写竞争性或补充性文档的用户在规范文档集建立后,愿意将他们的文档添加到规范文档集中,然后弃用他们的竞争性文档。最终,C++ 风格指南成为更大的“C++ 开发者指南”的一部分。随着文档集变得更加全面和权威,其质量也得到了提高。工程师开始记录错误,因为他们知道有人在维护这些文档。由于文档在源代码控制下锁定,并有适当的所有者,工程师也开始将变更列表直接发送给技术作家。

引入 go/ 链接（参见第 3 章）实际上使大多数文档更容易成为任何给定主题的规范。例如，我们的 C++ 开发人员指南在 “go/cpp” 中确立了地位。随着更好的内部搜索、go/ 链接以及将多个文档集成到一个通用文档集中，此类规范文档集随着时间的推移变得更加权威和强大。

## 文档哲学

警告：以下部分更多的是关于技术写作最佳实践（和个人观点）的论文，而不是“谷歌是如何做到的”。软件工程师可以选择完全掌握它，但理解这些概念可能会让你更轻松地编写技术信息。

### 谁、什么、何时、何地以及为什么

大多数技术文档回答的是“如何”的问题。这是如何运作的？

如何针对此 API 进行编程？如何设置此服务器？因此，软件工程师倾向于直接跳到任何给定文档的“如何”部分，而忽略与之相关的其他问题：谁、什么、何时、何地和为什么。确实，这些都不如如何重要。设计文档是个例外，因为等效方面通常是因为什么。但如果技术文档没有适当的框架，文档最终会令人困惑。尝试在任何文档的前两段中解决其他问题：

- 之前讨论过 WHO：即受众。但有时您还需要在文档中明确指出并针对受众。例如：“本文档适用于 Secret Wizard 项目的新工程师。” · WHAT 确定本文档的目的：“本文档是旨在在测试环境中启动 Frobber 服务器的教程。”有时，仅仅编写 WHAT 就可以帮助您适当地构建文档。如果您开始添加不适用于 WHAT 的信息，您可能需要将这些信息移到单独的文档中。
- WHEN 标识此文档的创建、审阅或更新时间。源代码中的文档会隐式地标注此日期，其他一些发布方案也会自动执行此操作。但如果一直没有，请确保在文档本身上标注文档的编写日期（或上次修订日期）。
- WHERE 通常也是隐含的，但决定文档应存放在何处。  
通常，首选项应受某种版本控制，最好是其记录的源代码。但其他格式也适用于不同的目的。在 Google，我们经常使用 Google Docs 进行轻松协作，尤其是在

设计问题。然而,在某个时候,任何共享文档都不再只是讨论,而更像是一份稳定的历史记录。此时,将其移至更永久、具有明确所有权、版本控制和责任的地方。·为什么确定文档的目的。总结您希望某人在阅读文档后从中得到什么。一个好的经验法则是将文档的介绍中确定为什么。当您撰写摘要时,请验证您是否满足了最初的期望(并进行相应修改)。

### 开头、中间和结尾所有文档(实际上,文档

的所有部分)都有开头、中间和结尾。

虽然这听起来很愚蠢,但大多数文档通常至少应该包含这三个部分。只有一个部分的文档只说一件事,很少有文档只说一件事。不要害怕在文档中添加章节;它们将流程分解为逻辑部分,并为读者提供文档内容的路线图。

即使是最简单的文档通常也会涉及不止一件事。我们广受欢迎的“每周 C++ 技巧”传统上非常简短,只关注一条小建议。然而,即使在这里,分节也很有帮助。传统上,第一节表示问题,中间部分介绍推荐的解决方案,结论总结要点。如果文档仅由一个部分组成,一些读者无疑会难以找出要点。

大多数工程师讨厌冗余,这是有原因的。但在文档中,冗余通常很有用。隐藏在长篇文字中的重要观点可能难以记住或找出。另一方面,过早将该观点放在更显眼的位置可能会丢失后面提供的背景信息。通常,解决方案是在介绍性段落中介绍和总结该观点,然后使用该部分的其余部分以更详细的方式陈述您的观点。在这种情况下,冗余有助于读者理解所陈述内容的重要性。

### 良好文档的参数

好的文档通常有三个方面:完整性、准确性和清晰度。你很少能在同一份文档中同时满足这三个方面;例如,当你试图使文档更加“完整”时,清晰度就会开始受到影响。如果你试图记录 API 的所有可能用例,你可能会陷入难以理解的混乱之中。

对于编程语言来说,在所有情况下都保持完全准确(并记录所有可能的副作用)也会影响清晰度。对于其他文档,试图清晰地描述一个复杂的主题可能会对文档的准确性产生微妙的影响;你可能会决定忽略概念文档中的一些罕见副作用,例如,

因为该文档的目的是让人们熟悉 API 的使用,而不是提供所有预期行为的教条概述。

在每种情况下,“好文档”的定义都是能够完成其预期任务的文档。因此,您很少希望文档完成多项任务。对于每份文档(以及每种文档类型),确定其重点并适当调整写作。编写概念文档?您可能不需要涵盖 API 的每个部分。编写参考资料?您可能希望它完整,但可能必须牺牲一些清晰度。编写登录页面?专注于组织并将讨论保持在最低限度。所有这些加起来就是质量,诚然,质量很难准确衡量。

如何快速提高文档质量?关注受众的需求。通常,少即是多。例如,工程师经常犯的一个错误是将设计决策或实现细节添加到 API 文档中。就像您理想情况下应该将精心设计的 API 中的接口与实现分开一样,您也应该避免在 API 文档中讨论设计决策。

用户不需要知道这些信息。相反,应该将这些决定放在专门的文档中(通常是设计文档)。

弃用文档就像旧代码会导致问题一样,

旧文档也会导致问题。随着时间的推移,文档会变得陈旧、过时或(通常)被废弃。尽可能避免废弃文档,但是当文档不再有任何用途时,请将其删除或将其标识为过时(并且,如果可用,请指示在哪里获取新信息)。即使对于无人拥有的文档,有人添加一条注释“这不再有效!”也比什么都不说并留下看似权威但不再有效的东西更有帮助。

在 Google,我们经常在文档中附加“新鲜日期”。此类文档会记录文档上次审阅的时间,文档集中的元数据会在文档在三个月内未被触及时发送电子邮件提醒。此类新鲜日期(如下例所示)以及将文档作为错误进行跟踪有助于使文档集随着时间的推移更易于维护,这是文档的主要关注点:

```
<!-- # 文
档新鲜度:有关更多信息,请参阅go/fresh-source。新鲜度:{所有者:`username` 已审核: 2019-02-27 } -->
*-->
```

拥有此类文档的用户有动力保持该新鲜度日期为最新(如果文档受源代码控制,则需要进行代码审查)。因此,这是一种确保文档不时被查看的低成本方法。在 Google,我们发现将文档所有者纳入此新鲜度中

文档本身内带有日期且署名“上次审阅者……”也导致了采用率的提高。

## 什么时候需要技术作家？

当 Google 还年轻且处于发展阶段时，软件工程领域没有足够的技术作家。（现在情况仍然如此。）那些被认为重要的项目往往会展请技术作家，无论该团队是否真的需要。当时的想法是，作家可以减轻团队编写和维护文档的一些负担，并（理论上）让重要项目实现更快的速度。事实证明这是一个错误的假设。

我们了解到，大多数工程团队都可以为自己（他们的团队）完美地编写文档；只有当他们为另一个受众编写文档时，他们才需要帮助，因为写给另一个受众很困难。

团队内部关于文档的反馈循环更加直接，领域知识和假设更加清晰，感知需求更加明显。当然，技术作家通常可以更好地处理语法和组织，但支持单个团队并不是对有限和专业资源的最佳利用；它无法扩展。它引入了一种不正当的激励：成为一个重要的项目，你的软件工程师就不需要编写文档了。

阻止工程师编写文档最终会适得其反。

由于技术作家的资源有限，他们通常应该专注于软件工程师不需要作为其正常职责的一部分来完成的任务。通常，这涉及编写跨越 API 边界的文档。Project Foo 可能清楚地知道 Project Foo 需要什么文档，但它可能不太清楚 Project Bar 需要什么。技术作家更能扮演不熟悉该领域的人的角色。事实上，这是他们的关键角色之一：挑战您的团队对项目实用性的假设。这就是为什么许多（如果不是大多数）软件工程技术作家倾向于关注这种特定类型的 API 文档的原因之一。

## 结论

过去十年，Google 在文档质量方面取得了长足进步，但坦率地说，Google 的文档还不是一流的。相比之下，工程师们逐渐接受了这样的事实：任何代码更改，无论多小，都需要测试。此外，测试工具强大、多样，并在各个环节插入到工程工作流程中。文档的根基却远不及这一水平。

公平地说,处理文档的需要与测试的需要并不相同。测试可以原子化（单元测试）,并可以遵循规定的形式和功能。文档大部分都不能。测试可以自动化,但自动化文档的方案往往缺乏。文档必然是主观的;文档的质量不是由作者来衡量的,而是由读者来衡量的,而且往往是相当异步的。尽管如此,人们认识到文档的重要性,文档开发流程也在不断改进。在本文作者看来,Google 的文档质量比大多数软件工程商店的文档质量都要好。

要改变工程文档的质量,工程师和整个工程组织都需要接受这样一个事实:他们既是问题,也是解决方案。他们不应该对文档现状束手无策,而应该意识到,编写高质量的文档是他们工作的一部分,从长远来看可以节省时间和精力。对于任何一段你预计会存在几个月以上的代码,你在编写该代码文档上投入的额外时间不仅会帮助其他人,还会帮助你维护该代码。

## TL;DR

- 随着时间的推移和规模的扩大,文档变得极为重要。· 文档变更应充分利用现有的开发人员工作流程。· 使文档专注于一个目的。· 为你的读者而非自己写作。



## 第十一章

# 测试概述

作者:亚当·本德 (Adam Bender)  
编辑:Tom Mansreck

测试一直是编程的一部分。事实上,第一次编写计算机程序时,您几乎肯定会向其中输入一些样本数据,以查看其是否按预期运行。长期以来,软件测试的最新进展与此非常相似,主要是手动且容易出错。然而,自 21 世纪初以来,软件行业的测试方法已发生了巨大变化,以应对现代软件系统的规模和复杂性。这种演变的核心是开发人员驱动的自动化测试实践。

自动化测试可以防止错误蔓延并影响用户。在开发周期中越晚发现错误,成本就越高;在许多情况下,成本会呈指数级增长。<sup>1</sup>然而,“发现错误”只是动机的一部分。测试软件的另一个同样重要的原因是支持更改能力。无论是添加新功能、进行专注于代码健康的重构,还是进行更大规模的重新设计,自动化测试都可以快速发现错误,这使得可以放心地更改软件。

能够更快迭代的公司可以更快地适应不断变化的技术、市场条件和客户口味。如果您拥有强大的测试实践,您就不必担心变化。您可以将其作为开发软件的基本素质。您想更改系统的频率越高、速度越快,您就越需要一种快速的方法来测试它们。

---

<sup>1</sup>请参阅“缺陷预防:降低成本并提高质量”。

编写测试还可以改善系统的设计。作为代码的第一个客户端,测试可以告诉你很多有关设计选择的信息。你的系统是否与数据库耦合太紧密?API 是否支持所需的用例?你的系统是否处理了所有的边缘情况?编写自动化测试迫使你在开发周期的早期就面对这些问题。这样做通常会使软件更加模块化,从而为以后提供更大的灵活性。

关于软件测试的话题已经有很多文章进行了探讨,理由很充分:对于如此重要的实践,做好它对很多人来说似乎仍然是一门神秘的技艺。在谷歌,虽然我们已经取得了长足的进步,但我们仍然面临着让整个公司流程可靠地扩展的难题。在本章中,我们将分享我们学到的知识,以帮助进一步讨论。

## 我们为什么要写测试?

为了更好地了解如何最大限度地发挥测试的作用,让我们从头开始。

当我们谈论自动化测试时,我们真正谈论的是什么?

最简单的测试定义如下:

- 您正在测试的单一行为,通常是您正在调用的方法或 API
- 特定输入,传递给 API 的某个值
- 可观察的输出或行为
- 受控环境,例如单个隔离进程

当您执行这样的测试时,将输入传递给系统并验证输出,您将了解系统是否按预期运行。总的来说,数百或数千个简单测试(通常称为测试套件)可以告诉您整个产品是否符合其预期设计,更重要的是,何时不符合。

创建和维护一个健康的测试套件需要付出真正的努力。随着代码库的增长,测试套件也会随之增长。它将开始面临不稳定和缓慢等挑战。如果无法解决这些问题,测试套件就会陷入瘫痪。请记住,测试的价值源于工程师对它们的信任。如果测试成为生产力的瓶颈,不断带来辛劳和不确定性,工程师就会失去信任并开始寻找解决方法。糟糕的测试套件可能比没有测试套件更糟糕。

除了帮助公司快速打造优质产品之外,测试对于确保我们生活中重要产品和服务的安全也变得至关重要。软件与我们的生活比以往任何时候都更加密切相关,缺陷可能会导致

不仅仅是一点小麻烦：它们可能会造成巨额的金钱损失、财产损失，或者最糟糕的是，生命损失。<sup>2</sup>在 Google，我们已确定测试不能是事后才想到的。专注于质量和测试是我们工作

的一部分。我们有时会痛苦地认识到，未能将质量融入我们的产品和服务必然会导致糟糕的结果。因此，我们将测试融入了我们工程文化的核心。

Google Web Server 的故事在 Google 的早期，工程师驱动的测试通常被认为并不重要。团队经常依靠聪明的人来确保软件正确运行。一些系统运行了大型集成测试，但大多数时候都是一片空白。有一种产品似乎遭受了最严重的打击：它被称为 Google Web Server，也称为 GWS。

GWS 是负责为 Google 搜索查询提供服务的 Web 服务器，它对 Google 搜索的重要性不亚于空中交通管制对机场。早在 2005 年，随着项目规模和复杂性的不断扩大，生产力急剧下降。发布版本变得越来越多错误，发布时间也越来越长。团队成员在对服务进行更改时缺乏信心，而且通常只有当功能在生产环境中停止工作时才发现问题。（有一次，超过 80% 的生产版本包含影响用户的错误，必须回滚。）

为了解决这些问题，GWS 的技术主管 (TL) 决定制定一项由工程师驱动的自动化测试政策。作为该政策的一部分，所有新的代码更改都必须包含测试，并且这些测试将持续运行。在制定该政策的一年内，紧急推送的数量下降了一半。尽管该项目每个季度都会经历创纪录的新变化，但这一下降仍然发生了。即使面对前所未有的增长和变化，测试也为 Google 最关键的项目之一带来了新的生产力和信心。

如今，GWS 已进行了数万次测试，几乎每天都会发布新版本，而客户可见的故障相对较少。

GWS 的变化标志着 Google 测试文化的分水岭，因为公司其他部门的团队看到了测试的好处并开始采用类似的策略。

---

<sup>2</sup>参见“宰赫兰的失败”。

GWS 经验教给我们的一个关键见解是,你不能单靠程序员的能力来避免产品缺陷。即使每个工程师只偶尔写出一个 bug,当有足够多的人在同一个项目上工作时,你就会陷入不断增长的缺陷清单中。想象一下一个假设的 100 人团队,其中的工程师非常优秀,以至于他们每个人每月只写一个 bug。

总的来说,这群出色的工程师每天仍然会产生 5 个新错误。更糟糕的是,在一个复杂的系统中,修复一个错误往往会导致另一个错误,因为工程师会适应已知的错误并围绕它们编写代码。

最好的团队会想方设法将其成员的集体智慧转化为整个团队的利益。这正是自动化测试的作用。团队中的工程师编写测试后,该测试将添加到可供其他人使用的公共资源池中。

现在,团队中的其他人都可以运行测试,并在测试检测到问题时受益。相比之下,基于调试的方法每次出现错误时,工程师都必须付出使用调试器深入研究的成本。工程资源的成本是天壤之别,这也是 GWS 能够扭转命运的根本原因。

以现代开发速度进行测试软件系统越来越大,也越来越复杂。谷歌的一个典型应用程序或服务由数千或数百万行代码组成。它使用数百个库或框架,必须通过不可靠的网络交付给

越来越多的平台运行着无数的配置。更糟糕的是,新版本被频繁推送给用户,有时一天会推送好几次。这与每年只更新一两次的包装软件世界相去甚远。

人类手动验证系统中每种行为的能力已经无法跟上大多数软件功能和平台的爆炸式增长。

想象一下,手动测试 Google 搜索的所有功能需要付出多大的努力,例如查找航班、电影时间、相关图片,当然还有网页搜索结果(见图11-1)。即使您可以确定如何解决这个问题,您也需要将工作量乘以 Google 搜索必须支持的每种语言、国家/地区和设备,并且不要忘记检查可访问性和安全性等问题。

试图通过要求人们手动与每个功能交互来评估产品质量是行不通的。谈到测试,有一个明确的答案:自动化。

Google search results for "sfo to london".

**Flights from San Francisco, CA (SFO) to London, United Kingdom (all airports)**

www.google.com/flights

San Francisco, CA (SFO) → London, United Kingdom (all airports)

Sun, September 15 → Mon, September 30

Airline	Flight Time	Type	Price
Multiple airlines	10h 10m+	Connecting	from \$353
British Airways	13h 35m+	Connecting	from \$365
Multiple airlines	10h 15m	Nonstop	from \$422
United	10h 30m	Nonstop	from \$422
Delta	10h 15m	Nonstop	from \$628
Virgin Atlantic	10h 15m	Nonstop	from \$628
Air France	10h 15m	Nonstop	from \$629
KLM	10h 15m	Nonstop	from \$629
Lufthansa	10h 30m	Nonstop	from \$692
Austrian	10h 30m	Nonstop	from \$692
Brussels Airlines	10h 30m	Nonstop	from \$692
Norwegian Air UK	10h 10m	Nonstop	from \$760
American	10h 25m	Nonstop	from \$939
British Airways	10h 25m	Nonstop	from \$939
Iberia	10h 25m	Nonstop	from \$939
Other airlines	10h 15m+	Connecting	from \$415

→ Search flights

**Cheap Flights from San Francisco to London from \$347 - KAYAK**

https://www.kayak.com/Flights/Worldwide/Europe/UnitedKingdom/England

Fly from San Francisco to London on Air Canada from \$347, Finnair from \$348, Lufthansa from \$363...  
Search ... SFO - LHR San Francisco - London Heathrow

How does KAYAK find such low flight prices?

How can Hacker Fares save me money?

Does KAYAK query more flight providers than competitors?

▼ Show more

Google search results for "spaceballs showtimes".

**Spaceballs / On TV soon**

All times are in Pacific Time

Date	Channel
Today, 9:25 AM	Starz Comedy (West)
Today, 5 PM	Starz Comedy (East)
Today, 8 PM	Starz Comedy (West)
Tomorrow, 10:30 PM	Encore (East)
Fri, 1/31, 1:30 AM	Encore (West)

www.fandango.com › spaceballs-836 › movie-overview ▾

**Spaceballs | Fandango**

★★★★ Rating: 83% - 434,978 votes

Showtimes are not available near 95101, watch Spaceballs anytime, anywhere with FandangoNow. Watch Spaceballs anytime, anywhere with FandangoNow.

www.fandango.com › spaceballs-836 › movie-time ▾

**Spaceballs Times - Movie Tickets + Showtimes | Fandango**

★★★★ Rating: 83% - 434,978 votes

SPACEBALLS, 1987, Park Circus/MGM, 96 min. Dir. Mel Brooks. Bill Pullman, John Candy and Rick Moranis head the cast in Mel Brooks' hilarious riff on STAR ...



**Spaceballs**

PG 1987 · Fantasy/Parody film · 1h 36m

Play trailer on YouTube

83% Fandango 7.1/10 IMDb 46% Metacritic

92% liked this movie Google users

In a distant galaxy, planet Spaceball has depleted its air supply, leaving its citizens reliant on a product called "Perri-Air." In desperation, Spaceball's leader President Skroob (Mel Brooks) orders the evil Dark Helmet (Rick Moranis) to kidnap Princess Vespa (Daphne Zuniga) of oxygen-rich Druidia... [MORE](#)

图 11-1 两个复杂的 Google 搜索结果的截图

## 编写、运行、反应

最纯粹的自动化测试包括三项活动：编写测试、运行测试和对测试失败做出反应。自动化测试是一小段代码，通常是单个函数或方法，它会调用要测试的大型系统的独立部分。测试代码会设置预期环境，调用系统（通常使用已知输入）并验证结果。有些测试非常小，只执行单个代码路径；其他测试则大得多，可能涉及整个系统，例如移动操作系统或 Web 浏览器。

**示例 11-1** 展示了一个刻意简单的 Java 测试，没有使用任何框架或测试库。这不是编写整个测试套件的方式，但从本质上讲，每个自动化测试看起来都与这个非常简单的示例类似。

### 例 11-1. 一个示例测试

```
// 验证计算器类是否可以处理负结果。
public void main(String[] args) {
    Calculator calculator = new Calculator();
    int expectedResult = -3;
    int actualResult = calculator.subtract(2, 5); // 给定 2, 减去 5。
    assertEquals(expectedResult, actualResult);
}
```

与过去的 QA 流程不同，以前的 QA 流程中，专门的软件测试人员会仔细研究系统的新版本，测试所有可能的行为，而如今构建系统的工程师在为自己的代码编写和运行自动化测试方面发挥着积极而不可或缺的作用。即使在 QA 是重要部门的公司，开发人员编写的测试也很常见。以当今系统开发的速度和规模，唯一的办法就是让整个工程人员共享测试的开发。

当然，编写测试与编写好的测试是不同的。培训数万名工程师编写好的测试可能非常困难。我们将在后续章节中讨论编写好的测试所学到的知识。

编写测试只是自动化测试流程的第一步。编写测试后，你需要运行它们。而且要频繁运行。自动化测试的核心是不断重复相同的操作，只有出现问题时才需要人工干预。我们将在**第 23 章**讨论持续集成 (CI) 和测试。通过将测试表达为代码而不是一系列手动步骤，我们可以在每次代码更改时运行它们。每天轻松运行数千次。与人类测试员不同，机器永远不会感到疲倦或无聊。

将测试表达为代码的另一个好处是，很容易将它们模块化以便在各种环境中执行。在 Firefox 中测试 Gmail 的行为

只要你同时配置了这两个系统,就不需要比在 Chrome 中做更多的事情了。3运行日语或德语用户界面 (UI) 的测试可以使用与英语相同的测试代码。

正在积极开发的产品和服务不可避免地会遇到测试失败。测试流程是否有效取决于它如何解决测试失败。

让失败的测试迅速堆积起来会抵消它们提供的任何价值,因此必须避免这种情况发生。在发生故障后几分钟内优先修复损坏的测试的团队能够保持高度的信心并快速隔离故障,从而从测试中获得更多价值。

总之,健康的自动化测试文化鼓励每个人分担编写测试的工作。这种文化还能确保定期运行测试。最后,也许是最重要的点,它强调快速修复有问题的测试,以保持对流程的高度信心。

测试代码的好处对于来自测试文化不

强的组织的开发人员来说,通过编写测试来提高生产力和速度的想法似乎有些矛盾。毕竟,编写测试所花的时间可能与首先实现一项功能所花的时间一样长(甚至更长!)。相反,在 Google,我们发现投资软件测试可以为开发人员的生产力带来几个关键好处:

更少的调试 正如你

所期望的,经过测试的代码在提交时具有更少的缺陷。至关重要的是,它在整个存在过程中的缺陷也更少;大多数缺陷将在代码提交之前被发现。谷歌的一段代码预计在其生命周期内会被修改数十次。它将被其他团队甚至自动代码维护系统修改。一次编写的测试会持续带来回报,并在项目的整个生命周期中避免代价高昂的缺陷和烦人的调试会话。测试基础设施可以快速检测到破坏测试的项目或项目依赖项的更改,并在问题发布到生产之前进行回滚。

增强对变更的信心 所有软件都会发生变

化。拥有良好测试的团队可以自信地审查和接受项目变更,因为项目的所有重要行为都会得到持续验证。这样的项目鼓励重构。重构的变更

---

3在不同的浏览器和语言中实现正确的行为又是另外一回事了!但理想情况下,最终用户体验对每个人来说都应该是相同的。

在保留现有行为的同时修改代码（理想情况下）不应该要求改变现有的测试。

#### 改进文档软件文档是出了名

的不可靠。从过时的需求到遗漏的边缘案例，文档与代码的关系往往很脆弱。清晰、有针对性的测试每次只执行一种行为，可以作为可执行文档。如果您想知道代码在特定情况下的作用，请查看该案例的测试。更好的是，当需求发生变化并且新代码破坏了现有测试时，我们会得到一个明确的信号，即“文档”已经过时了。请注意，只有注意保持测试清晰简洁，测试才能作为文档发挥最佳作用。

#### 更简单的审查

Google 的所有代码在提交之前都会经过至少一名其他工程师的审查（有关详细信息，请参阅[第 9 章](#)）。如果代码审查包括全面的测试，以证明代码的正确性、极端情况和错误条件，代码审查人员可以花费更少的精力来验证代码是否按预期工作。审查人员无需费力地在脑海中逐一检查代码中的每个案例，而是可以验证每个案例是否都通过了测试。

#### 精心设计 为新代码

编写测试是锻炼代码本身 API 设计的一种实用方法。如果新代码难以测试，通常是因为被测试的代码有太多职责或难以管理的依赖关系。设计良好的代码应该是模块化的，避免紧密耦合，并专注于特定的职责。尽早修复设计问题通常意味着以后的返工更少。

#### 快速、高质量的发布 有了健

康的自动化测试套件，团队就可以自信地发布应用程序的新版本。Google 的许多项目每天都会发布一个新版本投入生产。即使是拥有数百名工程师、每天提交数千个代码更改的大型项目也是如此。如果没有自动化测试，这是不可能的。

## 设计测试套件

如今，Google 的规模非常庞大，但我们的规模并不总是如此之大，而且我们的方法基础早已奠定。多年来，随着我们的代码库不断增长，我们学到了很多关于如何设计和执行测试套件的经验，通常是在通过犯错和事后清理来学习的。

我们很早就学到的一个教训是，工程师们喜欢编写更大的系统级测试，但这些测试比小测试更慢、更不可靠、更难调试。工程师们厌倦了调试系统级测试，

问自己：“为什么我们不能一次只测试一台服务器？”或“为什么我们需要一次测试整个服务器？我们可以单独测试较小的模块。”最终，为了减少痛苦，团队开发了越来越小的测试，结果发现这些测试更快、更稳定，而且通常痛苦更少。

这引起了公司内部关于“小”的确切含义的大量讨论。小是指单元测试吗？集成测试呢，它们的规模有多大？

我们得出的结论是，每个测试用例都有两个不同的维度：大小和范围。大小是指运行测试用例所需的资源：内存、进程和时间等。范围是指我们正在验证的特定代码路径。请注意，执行一行代码与验证它是否按预期工作不同。大小和范围是相互关联但又截然不同的概念。

## 测试规模

在 Google，我们将每个测试分为不同大小，并鼓励工程师始终为给定的功能编写尽可能小的测试。测试的大小不是由其代码行数决定的，而是由其运行方式、允许执行的操作以及消耗的资源数量决定的。事实上，在某些情况下，我们对小、中、大的定义实际上被编码为测试基础架构可以对测试实施的约束。我们稍后会详细介绍，但简而言之，小型测试在单个进程中运行，中型测试在单台机器上运行，大型测试可以在任何需要的地方运行，如图11-2 所示。

4

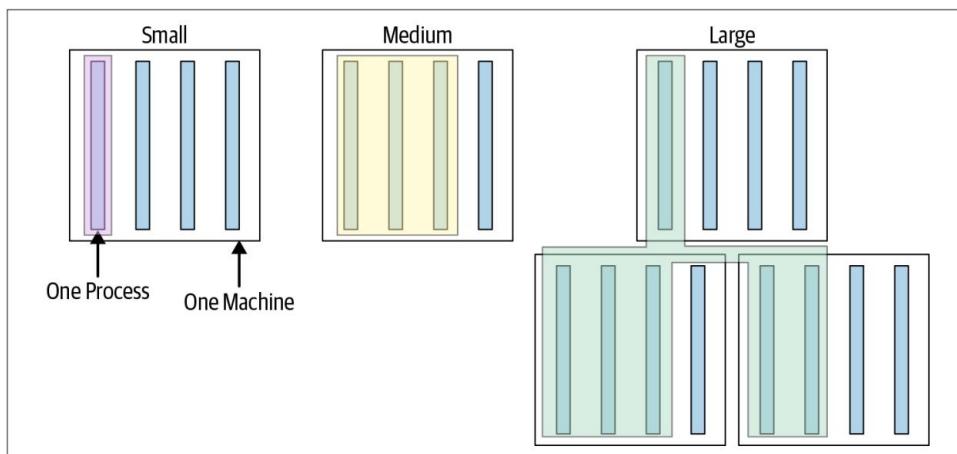


图 11-2. 测试规模

4从技术上讲，Google 的测试规模有四种：小型、中型、大型和巨大。大型和巨大之间的内在区别其实很微妙，而且历史悠久；因此，本书中对大型的大多数描述实际上都适用于我们对巨大概念的理解。

我们之所以做出这种区分,而不是更传统的“单元”或“集成”,是因为无论测试范围如何,我们最希望测试套件具备的品质是速度和确定性。无论范围如何,小型测试几乎总是比涉及更多基础设施或消耗更多资源的测试更快、更确定。对小型测试施加限制使速度和确定性更容易实现。随着测试规模的扩大,许多限制都会放宽。中型测试具有更大的灵活性,但也存在更大的不确定性风险。

大型测试只用于最复杂和最困难的测试场景。让我们仔细看看对每种类型的测试施加的具体限制。

### 小测试

小型测试是三种测试规模中最受限制的。主要限制是小型测试必须在单个进程中运行。在许多语言中,我们甚至进一步限制它们必须在单个线程上运行。这意味着执行测试的代码必须与被测试的代码在同一个进程中运行。您不能运行服务器并让单独的测试进程连接到它。这也意味着您不能在测试中运行第三方程序(例如数据库)。

对小型测试的其他重要限制是,它们不允许休眠、执行 I/O 操作<sup>5</sup>或进行任何其他阻塞调用。这意味着小型测试不允许访问网络或磁盘。测试依赖于这些操作的代码需要使用测试替身(参见[第 13 章](#))将重量级依赖项替换为轻量级的进程内依赖项。

这些限制的目的是确保小型测试无法接触到测试缓慢或不确定性的主要来源。在单个进程上运行且从不进行阻塞调用的测试可以有效地以 CPU 能够处理的速度运行。意外地使这样的测试变得缓慢或不确定是困难的(但肯定不是不可能)。对小型测试的限制提供了一个沙箱,可以防止工程师搬起石头砸自己的脚。

这些限制乍一看似乎有些过分,但想想一整天运行的几百个小测试用例。如果其中有几个非确定性地失败(通常称为[不稳定测试](#)),追查原因会严重影响生产力。以 Google 的规模而言,这样的问题可能会使我们的测试基础设施陷入停顿。

在 Google,我们鼓励工程师尽可能编写小型测试,无论测试范围如何,因为这可以保证整个测试套件快速可靠地运行。有关小型测试与单元测试的更多讨论,请参阅[第 12 章](#)。

---

<sup>5</sup>这项政策有一些回旋余地。如果测试使用密封的内存实现,则允许访问文件系统。

## 中等测试

对小型测试的限制对于许多有趣的测试来说可能过于严格。测试规模的下一个阶梯是中型测试。中型测试可以跨越多个进程,使用线程,并且可以对本地主机进行阻塞调用,包括网络调用。唯一剩下的限制是,中型测试不允许对本地主机以外的任何系统进行网络调用。换句话说,测试必须包含在一台机器中。

运行多个进程的能力带来了很多可能性。例如,您可以运行数据库实例来验证您正在测试的代码是否在更现实的环境中正确集成。或者您可以测试 Web UI 和服务器代码的组合。Web 应用程序的测试通常涉及WebDriver等工具启动一个真实的浏览器并通过测试过程远程控制它。

不幸的是,灵活性的提高也增加了测试变得缓慢和不确定的可能性。跨进程或允许进行阻塞调用的测试依赖于操作系统和第三方进程的快速性和确定性,而这通常不是我们能够保证的。中等测试仍然通过阻止通过网络访问远程机器提供了一些保护,这是大多数系统中速度缓慢和不确定性的最大来源。不过,在编写中等测试时,“安全性”是关闭的,工程师需要更加小心。

## 大型测试最

后,我们有大型测试。大型测试消除了对中型测试施加的本地主机限制,允许测试和被测试的系统跨越多台机器。例如,测试可能针对远程集群中的系统运行。

和以前一样,灵活性的提高伴随着风险的增加。与在单台机器上运行相比,处理跨多台机器的系统和连接它们的网络会大大增加速度缓慢和不确定性的可能性。我们主要将大型测试保留用于全系统端到端测试,这些测试更多地是验证配置而不是代码片段,以及无法使用测试替身的遗留组件的测试。我们将在第 14 章中详细讨论大型测试的用例。Google 的团队经常将大型测试与小型或中型测试隔离开来,只在构建和发布过程中运行它们,以免影响开发人员的工作流程。

### 案例研究:不稳定测试代价高昂如果你有几千个

测试,每个测试都带有一点点不确定性,并且整天都在运行,那么偶尔其中一个测试可能会失败(不稳定测试)。随着测试数量的增加,从统计上讲不稳定测试的数量也会增加。如果每个测试都有 0.1% 的失败率,而你每天运行 10,000 个测试,那么你每天将调查 10 个不稳定测试。每次调查都会占用你的团队原本可以做更有成功的事情的时间。

在某些情况下,您可以通过在测试失败时自动重新运行测试来限制不稳定测试的影响。这实际上是用 CPU 周期换取工程时间。在不稳定程度较低的情况下,这种权衡是合理的。但请记住,重新运行测试只会延迟解决不稳定根本原因的需要。

如果测试不稳定的情况持续增加,你会遇到比生产力下降更糟糕的事情:对测试失去信心。团队在对测试套件失去信任之前,不需要调查许多不稳定情况。发生这种情况后,工程师将停止对测试失败做出反应,从而消除测试套件提供的任何价值。

我们的经验表明,当不稳定性接近 1% 时,测试就开始失去价值。在 Google,我们的不稳定性率徘徊在 0.15% 左右,这意味着每天有数千个不稳定性。我们努力控制不稳定性,包括积极投入工程时间来修复它们。

在大多数情况下,不稳定性是由于测试本身的不确定性行为而产生的。软件提供了许多不确定性来源:时钟时间、线程调度、网络延迟等等。学习如何隔离和稳定随机性的影响并不容易。有时,影响与硬件中断或浏览器渲染引擎等低级问题有关。良好的自动化测试基础设施应该可以帮助工程师识别和缓解任何不确定性行为。

### 所有测试规模所共有的属性

所有测试都应力求做到密封:测试应包含设置、执行和拆除其环境所需的所有信息。测试应尽可能少地假设外部环境,例如测试运行的顺序。例如,它们不应依赖共享数据库。对于较大的测试,此约束变得更具挑战性,但仍应努力确保隔离。

测试应该只包含执行相关行为所需的信息。保持测试清晰简单有助于审查人员验证代码是否按其要求执行。清晰的代码还有助于在失败时诊断故障。我们喜欢说“测试应该一目了然”。因为测试本身没有测试,所以需要人工审查作为正确性的重要检查。因此,我们也**强烈反对使用控制流状态**

测试中的条件和循环之类的内容。更复杂的测试流程本身可能包含错误，并且使确定测试失败的原因变得更加困难。

请记住，只有出现问题时才会重新检查测试。当您被要求修复一个从未见过的有问题的测试时，您会感谢有人花时间让它变得容易理解。代码被阅读的次数远比被编写的次数多，因此请确保您编写的测试是您想要阅读的！

测试规模实践。有了精确的测试规模定义，我们就可以创建工具来执行这些定义。执行使我们能够扩展测试套件，同时仍能保证速度、资源利用率和稳定性。在 Google，这些定义的执行程度因语言而异。例如，我们使用自定义安全管理器运行所有 Java 测试，如果它们试图执行某些被禁止的操作（例如建立网络），则会导致所有标记为小的测试失败

联系。

测试范围虽然

Google 非常重视测试规模，但另一个需要考虑的重要属性是测试范围。测试范围指的是给定测试验证了多少代码。窄范围测试（通常称为“单元测试”）旨在验证代码库中一小块集中部分的逻辑，例如单个类或方法。中等范围测试（通常称为集成测试）旨在验证少数组件之间的交互；例如，服务器与其数据库之间的交互。大范围测试（通常称为功能测试、端到端测试或系统测试）旨在验证系统中几个不同部分之间的交互，或者未在单个类或方法中表达的突发行。

值得注意的是，当我们谈论单元测试的狭义范围时，我们指的是正在验证的代码，而不是正在执行的代码。

一个类具有许多依赖项或它引用的其他类是很常见的，并且这些依赖项在测试目标类时自然会被调用。

尽管还有一些其他的测试策略大量使用测试替身（假的或模拟的）来避免在被测系统之外执行代码，在 Google，我们更愿意在可行的情况下保留真实的依赖关系。[第 13 章](#)更详细地讨论了这个问题。

范围狭窄的测试往往规模较小，范围广泛的测试往往规模中等或较大，但情况并非总是如此。例如，可以编写一个范围广泛的服务器端点测试，该测试涵盖其所有正常解析、请求验证和业务逻辑，但它仍然很小，因为它使用双精度来代替所有进程外依赖项，如数据库或文件系统。同样，可以编写一个范围狭窄的单一方法测试，该测试必须是中等规模。对于

例如,现代 Web 框架通常将 HTML 和 JavaScript 捆绑在一起,而测试日期选择器等 UI 组件通常需要运行整个浏览器,甚至验证单个代码路径。

就像我们鼓励进行小规模的测试一样,在 Google,我们也鼓励工程师编写范围更窄的测试。作为一个非常粗略的指导方针,我们倾向于将大约 80% 的测试混合为小范围的单元测试,以验证我们的大部分业务逻辑;15% 为中等范围的集成测试,以验证两个或多个组件之间的交互;5% 为端到端测试,以验证整个系统。[图 11-3](#)描述了我们如何将其可视化为金字塔。

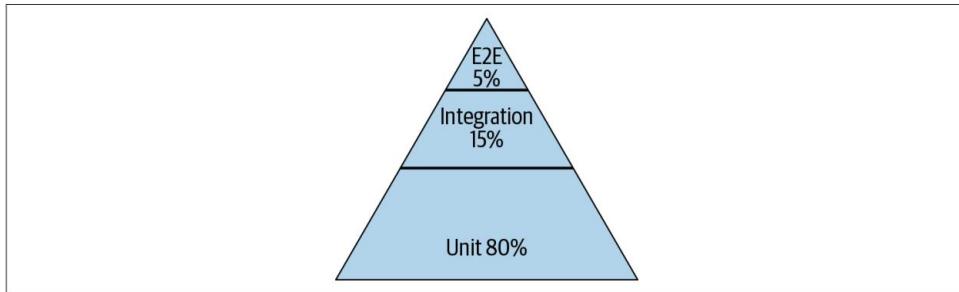


图 11-3。Google 版本的 Mike Cohn 测试金字塔;6 % 是按测试用例数计算的,每个团队的组合都会略有不同

单元测试是一个很好的基础,因为它们快速、稳定,并且大大缩小了范围,减少了识别类或函数可能具有的所有行为所需的认知负荷。此外,它们使故障诊断变得快速而轻松。需要注意的两个反模式是“冰淇淋甜筒”和“沙漏”,如[图11-4 所示](#)。

对于冰淇淋甜筒,工程师编写了许多端到端测试,但很少编写集成或单元测试。此类套件往往速度慢、不可靠且难以使用。这种模式通常出现在以原型开始并迅速投入生产的项目中,从未停下来解决测试债务。

沙漏模式涉及许多端到端测试和许多单元测试,但集成测试很少。它不像冰淇淋甜筒那么糟糕,但仍会导致许多端到端测试失败,而这些失败本可以通过一套中等范围的测试更快、更轻松地发现。当紧密耦合使得单独实例化单个依赖项变得困难时,就会出现沙漏模式。

<sup>6</sup> Mike Cohn,《敏捷成功:使用 Scrum 进行软件开发》(纽约:Addison-Wesley Professional,2009 年)。

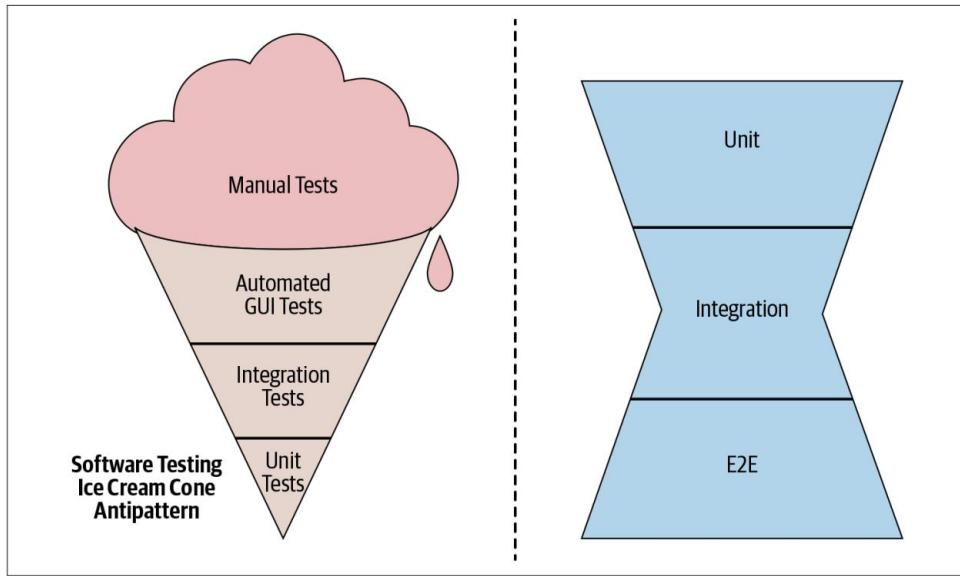


图 11-4. 测试套件反模式

我们推荐的测试组合取决于我们的两个主要目标:工程效率和产品信心。优先使用单元测试可以让我们在开发过程的早期迅速获得高度信心。在产品开发过程中,更大规模的测试可作为健全性检查;不应将其视为发现错误的主要方法。

在考虑自己的组合时,您可能需要不同的平衡。如果您强调集成测试,您可能会发现您的测试套件需要更长的时间来运行,但可以捕获更多组件之间的问题。当您强调单元测试时,您的测试套件可以非常快速地完成,并且您将捕获许多常见的逻辑错误。但是,单元测试无法验证组件之间的交互,例如由不同团队开发的两个系统之间的契约。好的测试套件包含适合当地架构和组织现实的不同测试规模和范围。

碧昂丝规则在培训新员工时,

我们经常被问到,哪些行为或属性需要测试?简单的答案是:测试所有你不想破坏的东西。换句话说,如果你想确信一个系统会表现出某种特定行为,那么确保它会表现出来的唯一方法就是为它编写一个自动化测试。这包括所有常见的测试,如测试性能、行为正确性、可访问性和安全性。它还包括不太明显的属性,如测试系统如何处理故障。

我们对这一普遍哲学有一个名称:碧昂丝规则。简而言之,可以这样表述:“如果你喜欢它,那么你应该对它进行测试。”负责对整个代码库进行更改的基础设施团队经常援引碧昂丝规则。如果不相关的基础设施更改通过了所有测试,但仍然破坏了团队的产品,你就有责任修复它并添加额外的测试。

### 故障测试系统必须考虑

的最重要的原因之一就是故障。故障是不可避免的,但等待真正的灾难发生才能知道系统对灾难的响应程度是痛苦的根源。与其等待故障发生,不如编写自动化测试来模拟常见的故障类型。这包括在单元测试中模拟异常或错误,并在集成和端到端测试中注入远程过程调用(RPC)错误或延迟。它还可以包括使用混沌工程等技术影响实际生产网络的更大规模中断。对不利条件的可预测和可控的响应是可靠系统的标志。

### 关于代码覆盖率的说明代码覆盖

率是衡量哪些功能代码行被哪些测试执行的指标。如果您有 100 行代码,而您的测试执行了其中的 90 行,则代码覆盖率为 90%。<sup>7</sup>代码覆盖率通常被视为了解测试质量的黄金标准指标,这有点不幸。有可能只用几个测试就执行了很多行代码,却从不检查每行代码是否在执行有用的操作。这是因为代码覆盖率仅衡量某行代码是否被调用,而不是结果发生了什么。(我们建议仅测量小型测试的覆盖率,以避免在执行大型测试时发生覆盖率膨胀。)

代码覆盖率还有一个更隐蔽的问题是,就像其他指标一样,它很快就会成为目标。团队通常会为预期的代码覆盖率设定一个标准,例如 80%。乍一看,这听起来非常合理;你肯定希望至少有这么多的覆盖率。实际上,工程师们不是把 80% 当作底线,而是把它当作上限。很快,变化开始出现,覆盖率不超过 80%。毕竟,为什么要做比指标要求更多的工作呢?

<sup>7</sup>请记住,覆盖范围有多种类型(线路、路径、分支等),每种类型对已测试的代码的描述也不同。在这个简单的示例中,使用的是线路覆盖范围。

评估测试套件质量的更好方法是思考所测试的行为。您是否有信心客户期望的一切都能够正常工作？您是否有信心可以捕获依赖项中的重大更改？您的测试是否稳定可靠？这些问题可以让您更全面地思考测试套件。每个产品和团队都会有所不同；有些产品和团队与硬件的交互难以测试，有些则涉及海量数据集。试图用一个数字来回答“我们有足够的测试吗？”这个问题会忽略很多背景信息，而且不太可能有用。代码覆盖率可以提供一些对未测试代码的洞察，但它不能代替对系统测试效果的批判性思考。

## 在 Google 规模上进行测试

到目前为止的大部分指导都可以应用于几乎任何规模的代码库。

然而，我们应该花些时间回顾一下我们在大规模测试中学到的东西。要了解 Google 的测试工作方式，您需要了解我们的开发环境，其中最重要的事实是，Google 的大部分代码都保存在一个单一的整体存储库 (**monorepo**) 中。我们运营的每种产品和服务的几乎每一行代码都存储在一个地方。目前，我们的存储库中有超过 20 亿行代码。

Google 的代码库每周都会经历近 2500 万行的变更。

其中大约一半是由我们 monorepo 中工作的数万名工程师完成的，另一半由我们的自动化系统以配置更新或大规模更改的形式完成（[第22 章](#)）。许多更改都是从当前项目之外发起的。我们对工程师重用代码的能力没有太多限制。

我们代码库的开放性鼓励了一定程度的共同所有权，让每个人都对代码库负责。这种开放性的一个好处是能够直接修复您使用的产品或服务中的错误（当然需要获得批准），而不是抱怨。这也意味着许多人会对其他人拥有的代码库的一部分进行更改。

谷歌的另一个与众不同之处在于，几乎没有团队使用存储库分支。所有更改都提交到存储库头，并立即可供所有人查看。此外，所有软件构建都使用我们的测试基础设施已验证的最后提交的更改来执行。构建产品或服务时，运行它所需的几乎所有依赖项也都是从源代码构建的，同样是从存储库头构建的。谷歌通过使用 CI 系统来管理这种规模的测试。我们的 CI 系统的关键组件之一是我们的测试自动化平台 (TAP)。



有关 TAP 和我们的 CI 理念的更多信息,请参阅[第 23 章](#)。

无论是从规模、单一仓库还是产品数量来看,Google 的工程环境都非常复杂。每周,它都要经历数百万行代码的更改、数十亿个测试用例的运行、数万个二进制文件的构建以及数百种产品的更新  
简直太复杂了!

大型测试套件的缺陷随着代码库的增长,您

将不可避免地需要对现有代码进行更改。

如果编写得不好,自动化测试会使这些更改变得更加困难。脆弱的测试(那些过度指定预期结果或依赖大量复杂的样板的测试)实际上会抵制更改。这些编写不佳的测试甚至在进行不相关的更改时也会失败。

如果您曾经对某个功能进行了五行更改,却发现有数十个不相关的、损坏的测试,那么您已经感受到了脆弱测试的阻力。随着时间的推移,这种阻力会让团队不愿意执行必要的重构来保持代码库的健康。

后续章节将介绍可用于提高测试稳健性和质量的策略。

一些最严重的脆弱测试缺陷是由于模拟对象的误用造成的。

Google 的代码库因滥用模拟框架而遭受了严重损失,以至于一些工程师宣称“不再使用模拟!”尽管这是一个强有力声明,但了解模拟对象的局限性可以帮助您避免滥用它们。



有关有效使用模拟对象的更多信息,请参阅[第 13 章](#)。

除了脆弱测试带来的摩擦之外,测试套件越大,运行速度就越慢。测试套件越慢,运行频率就越低,提供的好处就越少。我们使用了许多技术来加快测试套件的速度,包括并行执行和使用更快的硬件。然而,这些技巧最终会被大量单独缓慢的测试用例所淹没。

测试可能会因多种原因而变慢,例如启动系统的大部分内容、在执行前启动模拟器、处理大型数据集或等待不同的系统同步。测试通常启动得足够快,但随着

系统会不断增长。例如,也许您有一个集成测试,测试一个依赖项需要五秒钟才能响应,但多年来,您逐渐依赖于十几个服务,现在相同的测试需要五分钟。

测试也会因为sleep()和setTimeout()等函数引入的不必要的速度限制而变慢。在检查非确定性行为的结果之前,对这些函数的调用通常被用作简单的启发式方法。在这里或那里休眠半秒钟乍一看似乎并不太危险;但是,如果在广泛使用的实用程序中嵌入了“等待并检查”,很快你就会在每次运行测试套件时增加几分钟的空闲时间。更好的解决方案是以接近微秒的频率主动轮询状态转换。如果测试无法达到稳定状态,你可以将其与超时值结合使用。

如果测试套件不能保证确定性和快速性,那么它就会成为生产力的障碍。在 Google,遇到这些测试的工程师已经找到了绕过速度变慢的方法,有些工程师甚至在提交更改时完全跳过测试。显然,这是一种冒险的做法,应该阻止,但如果测试套件带来的危害大于好处,最终工程师会找到一种方法来完成他们的工作,无论有没有测试。

处理大型测试套件的秘诀是尊重它,激励工程师关心他们的测试;对他们进行坚如磐石的测试给予奖励,就像对他们发布出色的功能给予奖励一样。设定适当的性能目标并重构缓慢或边缘测试。基本上,将测试视为生产代码。当简单的更改开始花费大量时间时,请花精力使测试更可靠。

除了培养适当的文化外,还要投资于测试基础设施,开发 linters、文档或其他帮助,使编写不良测试变得更加困难。减少需要支持的框架和工具的数量,以提高您投入改进的时间的效率。<sup>8</sup>如果您不投资于让测试管理变得简单,最终工程师会认为它们根本不值得拥有。

## 谷歌测试的历史

既然我们已经讨论了 Google 的测试方法,那么了解我们是如何走到这一步可能会很有启发。如前所述,Google 的工程师并不总是接受自动化测试的价值。事实上,直到 2005 年,测试更像是一种好奇心,而不是一种有纪律的实践。大多数测试都是手动完成的,如果有的话。然而,从 2005 年到 2006 年,一场测试革命发生了,改变了

---

<sup>8</sup> Google 支持的每种语言都有一个标准测试框架和一个标准 mocking/stubbing 库。一套基础设施可以运行整个代码库中所有语言的大多数测试。

我们处理软件工程的方式。其影响至今仍在公司内回荡。

我们在本章开头讨论的 GWS 项目的经验起到了催化剂的作用。它清楚地表明了自动化测试的强大。在 2005 年对 GWS 进行改进后,这些做法开始在整个公司传播。工具很原始。然而,后来被称为测试小组的志愿者们并没有因此而放慢脚步。

三个关键举措帮助公司意识到了自动化测试的重要性:入门课程、测试认证计划和厕所测试。每个举措都以完全不同的方式产生了影响,它们共同重塑了 Google 的工程文化。

## 入学指导课程

尽管谷歌早期的很多工程人员都回避测试,但谷歌自动化测试的先驱们知道,按照公司的发展速度,新工程师的数量很快就会超过现有团队成员的数量。如果他们能接触到公司所有的新员工,这可能是引入文化变革的极其有效的途径。幸运的是,所有新工程师员工过去和现在都要经历一个瓶颈:入职培训。

Google 早期的入职培训课程主要涉及医疗福利和 Google 搜索的工作原理等问题,但从 2005 年开始,它还开始包含一个小时的讨论,讨论自动化测试的价值。<sup>9</sup>课程涵盖了测试的各种好处,例如提高生产率、更好的文档和重构支持。它还介绍了如何编写良好的测试。对于当时的许多 Noogler(新 Google 员工)来说,这样的课程是他们第一次接触这些材料。最重要的是,所有这些想法都像公司的标准做法一样呈现。新员工们不知道他们被用作特洛伊木马,将这个想法偷偷带入毫无戒心的团队。

当 Noogler 参加入职培训后加入团队时,他们开始编写测试并询问团队中没有编写测试的人。在短短一两年内,接受过测试教育的工程师人数就超过了未接受测试文化的工程师人数。因此,许多新项目都有了良好的开端。

如今,测试在行业中已经得到越来越广泛的应用,因此大多数新员工在加入时都对自动化测试抱有坚定的期望。尽管如此,入职培训课程仍会设定对测试的期望,并将新员工所学的东西联系起来

---

<sup>9</sup>这门课非常成功,以至于今天仍在教授更新版本。事实上,它是历史最悠久的课程之一。开办有关公司历史的入门课程。

了解 Google 外部的测试以及在我们非常庞大且非常复杂的代码库中进行测试所面临的挑战。

## 测试认证

最初,我们代码库中较大且较复杂的部分似乎难以进行良好的测试实践。一些项目的代码质量非常差,几乎无法测试。为了让项目有明确的前进方向,测试小组设计了一个名为“测试认证”的认证计划。测试认证旨在让团队了解其测试流程的成熟度,更重要的是,提供有关如何改进测试流程的指导手册。

该计划分为五个级别,每个级别都需要采取一些具体行动来改善团队的测试习惯。这些级别的设计使得每个步骤都可以在一个季度内完成,这使其与 Google 的内部规划节奏相得益彰。

测试认证 1 级涵盖基础知识:设置持续构建;开始跟踪代码覆盖率;将所有测试分为小型、中型或大型;识别(但不一定修复)不稳定的测试;并创建一组可以快速运行的快速测试(不一定全面)。随后的每个级别都增加了更多挑战,例如“不发布存在破坏测试的版本”或“删除所有不确定性测试”。到 5 级时,所有测试都已自动化,每次提交之前都会运行快速测试,所有不确定性都已消除,并且每种行为都已涵盖。内部仪表板通过显示每个团队的级别来施加社会压力。不久之后,各个团队就开始相互竞争以攀登阶梯。

到 2015 年测试认证程序被自动化方法取代时(稍后会详细介绍 pH),它已经帮助 1,500 多个项目改善了他们的测试文化。

马桶测试在 Google 测试小组尝试改

进测试的所有方法中,可能没有比马桶测试(TotT)更另类的了。TotT 的目标相当简单:积极提高整个公司的测试意识。问题是,在一家员工遍布全球的公司中,最好的方法是什么?

测试小组考虑过定期发送电子邮件简报的想法,但考虑到谷歌每个人都要处理大量的电子邮件,这个想法很可能会淹没在纷乱的邮件中。经过一番头脑风暴,有人提出了一个在卫生间隔间里张贴传单的主意,以示玩笑。我们很快就意识到这个想法很有创意:不管怎样,卫生间是每个人每天必须至少去一次的地方。

不管是不是玩笑,这个想法实施起来成本很低,所以必须尝试一下。

2006 年 4 月,一篇关于如何改进 Python 测试的短文出现在 Google 的洗手间里。第一篇是由一小群志愿者发布的。说反应两极分化是轻描淡写;有些人认为这是对个人空间的侵犯,并强烈反对。邮件列表中充斥着投诉,但 TotT 的创建者很满意。抱怨的人仍在谈论测试。

最终,风波平息,TotT 迅速成为 Google 文化的一部分。

到目前为止,公司各处的工程师已经制作了数百集,几乎涵盖了测试的所有方面(以及各种其他技术主题)。新集备受期待,一些工程师甚至自愿在自己的大楼周围张贴这些集。我们有意将每集限制在一页以内,挑战作者专注于最重要的可行建议。一集好的内容包含工程师可以立即带回办公桌并尝试的内容。

讽刺的是,尽管 TotT 是一本出现在较为私密的网站上的出版物,但它却产生了巨大的公众影响。大多数外部访问者在访问期间都会看一集,而这样的邂逅往往会展开一些有趣的对话,比如谷歌用户似乎总是在思考代码。此外,TotT 的剧集可以写成很棒的博客文章,这是 TotT 的原始作者很早就意识到的。他们开始[公开发布经过轻微编辑的版本](#),帮助与整个行业分享我们的经验。

尽管 TotT 最初只是一个玩笑,但是在测试小组发起的所有测试计划中,它运行时间最长,影响也最深远。

当下的测试文化如今,Google

的测试文化与 2005 年相比已有了很大的进步。Google 员工仍然会参加测试入门课程,并且 TotT 仍会几乎每周发布一次。

然而,测试的期望已经深深嵌入到开发人员的日常工作流程中。

Google 的每一次代码更改都需要经过代码审查。并且每次更改都应包括功能代码和测试。审查人员需要审查两者的质量和正确性。事实上,如果更改缺少测试,阻止更改是完全合理的。

作为 Test Certified 的替代品,我们的一个工程生产力团队最近推出了一款名为 Project Health (pH) 的工具。pH 工具会持续收集有关项目健康状况的数十个指标,包括测试覆盖率和测试延迟,并在内部提供这些指标。pH 的衡量标准为 1 (最差) 到 5 (最好)。pH-1 项目被视为团队需要解决的问题。几乎每个运行持续构建的团队都会自动获得 pH 分数。

随着时间的推移,测试已成为 Google 工程文化不可或缺的一部分。我们有无数种方式来向全公司的工程师强调测试的价值。通过培训、温和的推动、指导,甚至是一点友好的竞争,我们创造了明确的期望,即测试是每个人的工作。

为什么我们不从强制编写测试开始呢?

测试小组曾考虑向高层领导申请测试授权,但很快决定不这么做。任何关于如何开发代码的授权都会严重违背 Google 文化,而且可能会减慢进度,无论授权的想法是什么。他们认为成功的想法会传播开来,所以重点变成了展示成功。

如果工程师决定自己编写测试,则意味着他们已经完全接受了这个想法,并且很可能会继续做正确的事情 即使没有人强迫他们这样做。

## 自动化测试的局限性

自动化测试并不适合所有测试任务。例如,测试搜索结果的质量通常需要人工判断。我们使用搜索质量评估员进行有针对性的内部研究,他们执行真实查询并记录他们的印象。同样,在自动化测试中很难捕捉到音频和视频质量的细微差别,因此我们经常使用人工判断来评估电话或视频通话系统的性能。

除了定性判断之外,人类还擅长某些创造性评估。例如,寻找复杂的安全漏洞是人类比自动化系统做得更好的事情。人类发现并理解漏洞后,可以将其添加到自动化安全测试系统中,例如 Google 的[云安全扫描器](#)它可以在那里连续、大规模地运行。

这种技术的一个更通用的术语是探索性测试。探索性测试是一种从根本上创造性的努力,其中有人将测试中的应用程序视为一个需要破解的谜题,可能是通过执行一组意想不到的步骤或插入意想不到的数据。在进行探索性测试时,一开始不知道要发现的具体问题。通过探测经常被忽视的代码路径或应用程序的异常响应,逐渐发现这些问题。

与安全漏洞的检测一样,一旦探索性测试发现问题,就应该添加自动化测试,以防止将来出现回归。

使用自动化测试来覆盖众所周知的行为,可以使人工测试人员将昂贵的定性工作重点放在他们能够提供最大价值的产品部分,并避免在后期让测试人员感到无聊。

过程。

## 结论

采用开发人员驱动的自动化测试是 Google 最具变革性的软件工程实践之一。它使我们能够以比我们想象的更快的速度与更大的团队一起构建更大的系统。它帮助我们跟上技术变革的步伐。在过去 15 年里,我们成功地改变了我们的工程文化,将测试提升为一种文化规范。尽管公司自开始以来增长了近 100 倍,但人们对质量和测试的承诺比以往任何时候都更加坚定。

本章旨在帮助您了解 Google 对测试的看法。

在接下来的几章中,我们将更深入地探讨一些关键主题,这些主题有助于我们理解编写良好、稳定和可靠的测试意味着什么。我们将讨论单元测试是什么、为什么以及如何进行单元测试,这是 Google 最常见的测试类型。我们将深入探讨如何通过伪造、存根和交互测试等技术在测试中有效地使用测试替身。

最后,我们将讨论测试更大、更复杂的系统所面临的挑战,例如 Google 的许多系统。

在这三章结束时,您应该对我们使用的测试策略以及更重要的是我们为什么使用它们有更深入和更清晰的了解。

## TL;DR

- 自动化测试是实现软件变革的基础。· 为了扩展测试,测试必须实现自动化。· 平衡的测试套件对于保持良好的测试覆盖率必不可少。· “如果喜欢它,你就应该对它进行测试。”· 改变组织的测试文化需要时间。

## 第十二章

# 单元测试

作者:Erik Kueer  
编辑:Tom Mansreck

上一章介绍了 Google 对测试进行分类的两个主要维度:规模和范围。概括起来,规模是指测试所消耗的资源以及允许测试执行的操作,而范围是指测试旨在验证的代码量。

尽管 Google 对测试规模有明确的定义,但范围往往比较模糊。我们使用单元测试一词来指代范围相对较窄的测试,例如单个类或方法的测试。单元测试通常规模较小,但情况并非总是如此。

除了预防错误之外,测试最重要的目的是提高工程师的工作效率。与范围更广的测试相比,单元测试具有许多特性,使其成为优化生产力的绝佳方法:

- 根据 Google 对测试规模的定义,小型测试通常规模较小。小型测试快速且具有确定性,因此开发人员可以将其作为工作流程的一部分频繁运行,并立即获得反馈。 · 它们通常易于与测试代码同时编写,因此工程师可以将测试重点放在他们正在处理的代码上,而无需设置和了解更大的系统。 · 由于小型测试快速且易于编写,因此它们可以提高测试覆盖率。高测试覆盖率使工程师可以放心地进行更改,而不会破坏任何东西。
- 当测试失败时,它们往往能让人轻松了解问题所在,因为每个测试在概念上都很简单,并且专注于系统的特定部分。 · 它们可以作为文档和示例,向工程师展示如何使用正在测试的系统部分以及该系统的预期工作方式。

由于单元测试具有诸多优势,Google 编写的大多数测试都是单元测试。根据经验,我们鼓励工程师将单元测试和范围更广的测试的比例控制在 80% 左右。这一建议加上单元测试的编写简易性和运行速度,意味着工程师会运行大量单元测试。工程师在普通工作日内(直接或间接)执行数千个单元测试并不罕见。

由于测试可维护性在工程师的生活中占据了很大一部分,因此 Google 非常重视测试可维护性。可维护的测试是“可以正常工作”的测试:编写测试后,工程师无需再考虑它们,直到测试失败,而这些失败表明存在真正的错误,并且原因明确。本章的大部分内容重点介绍可维护性的概念及其实现技术。

## 可维护性的重要性

想象一下这样的场景:玛丽想给产品添加一个简单的新功能,并能够快速实现它,可能只需要几十行代码。但是当她去检查更改时,她从自动测试系统看到满屏的错误。她花了一天的剩余时间逐一检查这些失败。在每种情况下,更改都没有引入实际的错误,但打破了测试对代码内部结构的一些假设,需要更新这些测试。通常,她很难弄清楚测试最初试图做什么,而她为修复它们而添加的黑客行为使这些测试在将来更加难以理解。最终,本来应该很快完成的工作最终花费了几个小时甚至几天的忙碌工作,扼杀了玛丽的生产力并削弱了她的士气。

在这里,测试的效果与预期相反,它消耗了生产力而不是提高生产力,而且没有显著提高被测代码的质量。这种情况太常见了,谷歌工程师每天都在努力解决这个问题。虽然没有灵丹妙药,但谷歌的许多工程师一直在努力开发一套模式和做法来缓解这些问题,我们鼓励公司的其他员工效仿。

Mary 遇到的问题不是她的错,她也无法避免这些问题:糟糕的测试必须在签入之前修复,否则会拖累未来的工程师。从广义上讲,她遇到的问题分为两类。首先,她正在处理的测试很脆弱:它们因无害且无关的更改而中断,而这些更改并没有引入任何真正的错误。其次,测试不明确:测试失败后,很难确定问题出在哪里、如何修复以及这些测试最初应该做什么。

# 防止脆弱测试

按照上述定义,脆弱测试是指在与生产代码无关的变更(不会引入任何实际错误)面前失败的测试。<sup>1</sup>这类测试必须由工程师在工作中诊断和修复。在只有少数工程师的小型代码库中,每次变更都要调整几个测试可能不是什么大问题。但如果团队定期编写脆弱测试,测试维护将不可避免地占用团队越来越多的时间,因为他们被迫在不断增长的测试套件中梳理越来越多的失败。如果一组测试需要工程师针对每次变更手动调整,那么称其为“自动化测试套件”就有点牵强了!

脆弱的测试会给任何规模的代码库带来麻烦,但在Google这样的规模下,问题尤为严重。单个工程师在工作期间可能一天内轻松运行数千个测试,而一次大规模更改(参见[第22章](#))可能会触发数十万个测试。在这种规模下,即使影响一小部分测试的虚假破坏也会浪费大量的工程时间。

谷歌的各个团队在测试套件的脆弱性方面存在很大差异,但我们已经确定了一些可以使测试更能抵御变化的实践和模式。

## 力求测试不变在讨论避免脆弱测试的

模式之前,我们需要回答一个问题:编写测试后,我们预计需要多久更改一次测试?花在更新旧测试上的时间是无法花在更有价值的工作上的时间。因此,理想的测试是不变的:编写测试后,除非被测系统的要求发生变化,否则测试永远不需要更改。

在实践中,情况会怎样?我们需要考虑工程师对生产代码所做的更改类型,以及我们应该如何期望测试对这些更改做出响应。从根本上讲,有四种类型的更改:

### 纯重构 当工程师重

构系统内部而不修改其接口时,无论是出于性能、清晰度还是其他原因,系统的测试都不需要更改。在这种情况下,测试的作用是确保重构不会改变系统的行为。重构期间需要更改的测试表明,要么更改正在影响系统的行为,并且不是纯重构,要么测试的编写级别不合适

---

<sup>1</sup>请注意,这与不稳定测试略有不同,不稳定测试会非确定性地失败,而不会对生产代码。

抽象。Google 依靠大规模变更（第 22 章中描述）来进行此类重构，这使得此案例对我们来说尤为重要。

#### 新功能 当工程

师向现有系统添加新功能或行为时，系统的现有行为应保持不受影响。工程师必须编写新测试来涵盖新行为，但他们不需要更改任何现有测试。与重构一样，在添加新功能时对现有测试的更改表明该功能或不适当的测试会产生意想不到的后果。

#### 修复错误

修复错误很像添加新功能：错误的存在表明初始测试套件中缺少一个案例，错误修复应包括缺少的测试案例。同样，错误修复通常不需要更新现有测试。

#### 行为变更 改变系统

的现有行为是我们预计必须更新系统现有测试的一种情况。请注意，此类变更的成本往往比其他三种类型的变更高得多。系统的用户可能依赖于其当前行为，而对该行为的更改需要与这些用户进行协调，以避免混淆或破坏。在这种情况下更改测试表明我们正在破坏系统的明确契约，而前几种情况下的更改表明我们正在破坏非预期的契约。低级库通常会投入大量精力来避免改变行为，以免破坏用户。

要点是，编写测试后，在重构系统、修复错误或添加新功能时，您不需要再次触及该测试。这种理解使得大规模使用系统成为可能：扩展系统只需要编写与您正在进行的更改相关的少量新测试，而不必触及曾经针对系统编写的每个测试。只有系统行为的重大变化才需要回头更改其测试，在这种情况下，更新这些测试的成本往往相对于更新所有系统用户的成本而言很小。

#### 通过公共 API 进行测试

现在我们了解了我们的目标，让我们看一些实践，以确保除非被测试系统的要求发生变化，否则测试不需要改变。

迄今为止，确保这一点的最重要的方法是编写测试，以与用户相同的方式调用被测试系统；也就是说，调用其公共 API 而不是其实现细节。如果测试与系统用户的工作方式相同，那么根据定义，破坏测试的更改也可能破坏用户。作为额外的好处，此类测试可以作为用户的有用示例和文档。

考虑示例 12-1，它验证交易并将其保存到数据库。

### 例 12-1. 事务 API

```
public void processTransaction (Transaction交易) { if (isValid (交易) ) {

    保存到数据库 (事务) ; }

}

私有布尔值isValid (Transaction t){返回t.getAmount () <
    t.getSender () .getBalance () ;}

私有void saveToDatabase (事务t){
    字符串 s = t.getSender() + ， + t.getRecipient() + ， + t.getAmount();数据库.put(t.getId(), s);}

公共无效设置帐户余额 (字符串帐户名称,整数余额) {
    // 直接将余额写入数据库
}

公共无效获取帐户余额 (字符串帐户名称) {
    // 从数据库读取交易以确定账户余额
}
```

测试此代码的一个诱人的方法是删除“私有”可见性修饰符并直接测试实现逻辑，如示例 12-2 所示。

### 例 12-2. 对事务 API 实现的简单测试

```
@Test
public void emptyAccountShouldNotBeValid() {
    断言 (处理器.isValid (newTransaction () .setSender (EMPTY_ACCOUNT) )) 是假的 () ;
}

@Test
public void shouldSaveSerializedData() {
    处理器.saveToDatabase (newTransaction () .setId
        (123) .setSender
        (我) 。 setRecipient (你) 。 setAmount (100) ) ;
    assertThat (数据库.get (123) ) .isEqualTo (我,你,100) ;
}
```

这个测试与交易处理器的交互方式与真实用户大不相同：它窥视系统的内部状态，并调用未发布的方法。

经常作为系统 API 的一部分暴露。因此,测试很脆弱,几乎任何被测系统的重构 (例如重命名其方法、将它们分解为辅助类或更改序列化格式)都会导致测试中断,即使这种更改对于类的实际用户是不可见的。

相反,通过仅针对类的公共 API 进行测试就可以实现相同的测试覆盖率,如示例 12-3 所示。<sup>2</sup>

### 例 12-3. 测试公共 API

```
@Test
public void shouldTransferFunds() {
    处理器.setAccountBalance(“我”, 150);处理
    器.setAccountBalance(“你”, 20);

    处理器.processTransaction (newTransaction () .setSender
        ( 我 ) .setRecipient
        ( 你 ) .setAmount
        (100) );

    断言 (processor.getAccountBalance ( 我 )) .isEqualTo (50);断言
    (processor.getAccountBalance ( 你 )) .isEqualTo (120);
}

@Test
public void shouldNotPerformInvalidTransactions() {
    处理器.setAccountBalance(“我”, 50);处理
    器.setAccountBalance(“你”, 20);

    处理器.processTransaction (newTransaction () .setSender
        ( 我 ) .setRecipient
        ( 你 ) .setAmount
        (100) );

    断言 (processor.getAccountBalance ( 我 )) .isEqualTo (50);断言
    (processor.getAccountBalance ( 你 )) .isEqualTo (20);
}
```

仅使用公共 API 的测试,顾名思义,是以与用户相同的方式访问被测系统。此类测试更切合实际,更不易损坏,因为它们形成了明确的契约:如果此类测试失败,则意味着系统的现有用户也会失败。仅测试这些契约意味着您可以自由地对系统进行任何内部重构,而不必担心对测试进行繁琐的更改。

---

<sup>2</sup>这有时被称为“先使用前门原则”。

人们并不总是很清楚什么构成了“公共 API”，而这个问题实际上触及了单元测试中“单元”的核心。单元可以小到单个函数，也可以大到一组相关的包 / 模块。当我们在这种情况下说“公共 API”时，我们实际上是在谈论该单元向拥有代码的团队之外的第三方公开的 API。这并不总是与某些编程语言提供的可见性概念相符；例如，Java 中的类可能将自己定义为“公共”，以便同一单元中的其他包可以访问，但不打算供单元之外的其他方使用。有些语言（如 Python）没有内置的可见性概念（通常依赖于在私有方法名称前加上下划线之类的约定），并且构建系统（如 Bazel）可以进一步限制谁可以依赖编程语言声明为公开的 API。

定义一个单元的适当范围以及因此应该被视为公共 API 是一门艺术而非科学，但这里有一些经验法则：

- 如果某个方法或类仅用于支持一两个其他类（即，它是一个“辅助类”），那么它可能不应被视为自己的单元，并且应通过这些类而不是直接测试其功能。
- 如果某个包或类设计为任何人都可以访问而无需咨询其所有者，那么它几乎肯定是一个应该直接测试的单元，其中对它的测试以与用户相同的方式访问该单元。
- 如果某个包或类只能由其所有者访问，但它旨在提供在一系列上下文中有用的一般功能（即，它是一个“支持库”），那么它也应被视为一个单元并直接进行测试。鉴于支持库的代码将同时由其自己的测试和用户的测试覆盖，这通常会在测试中产生一些冗余。

然而，这种冗余是有价值的：如果没有冗余，当库中的一个用户（及其测试）被删除时，就可能出现测试覆盖率的缺口。

在 Google，我们发现工程师有时需要被说服通过公共 API 进行测试比针对实现细节进行测试更好。这种不情愿是可以理解的，因为编写专注于您刚刚编写的代码的测试通常比弄清楚该代码如何影响整个系统要容易得多。尽管如此，我们发现鼓励这种做法很有价值，因为额外的前期努力可以减轻维护负担，从而获得数倍的回报。

针对公共 API 进行测试并不能完全防止脆弱性，但它是您可以做的最重要的事情，以确保您的测试仅在系统发生重大更改时才会失败。

## 测试状态,而不是交互

测试通常依赖于实现细节的另一种方式涉及的不是测试调用系统的哪些方法,而是如何验证这些调用的结果。通常,有两种方法可以验证被测系统是否按预期运行。通过状态测试,您可以观察系统本身,看看调用后系统是什么样子。通过交互测试,您可以检查系统在调用时是否对其协作者采取了预期的操作序列。许多测试将执行状态和交互验证的组合。

交互测试往往比状态测试更脆弱,原因与测试私有方法比测试公共方法更脆弱相同:交互测试检查系统如何得出其结果,而通常你只需要关心结果是什么。[示例 12-4](#)演示了一个使用测试替身(第 13 章将进一步解释)来验证系统如何与数据库交互的测试。

### 例 12-4. 脆性相互作用测试

```
@Test
public void shouldWriteToDatabase()
    {accounts.createUser( foobar );验证(数据
        库).put( foobar );
}
```

该测试验证是否对数据库 API 进行了特定调用,但有几种不同的方式可能会出错:

- 如果测试系统中的错误导致记录在写入后不久从数据库中删除,则测试将通过,即使我们希望它失败。
- 如果重构被测系统以调用略有不同的 API 来编写等效记录,则测试将失败,即使我们希望它通过。

直接针对系统状态进行测试就没那么容易了,如[示例 12-5 所示](#)。

### 例 12-5. 测试状态

```
@Test
public void shouldCreateUsers()
    {account.createUser( foobar );
    assertThat(accounts.getUser( foobar )).isNotNull();
}
```

这个测试更准确地表达了我们所关心的:与被测系统交互之后的状态。

交互测试出现问题的最常见原因是过度依赖模拟框架。这些框架可以轻松创建测试替身来记录和验证针对它们的每个调用，并在测试中使用这些替身代替真实对象。这种策略直接导致交互测试变得脆弱，因此我们倾向于使用真实对象而不是模拟对象，只要真实对象快速且确定性强即可。



有关测试替身和模拟框架的更广泛的讨论、何时使用它们以及更安全的替代方案，请参阅[第 13 章](#)。

## 编写清晰的测试

即使我们完全避免了脆弱性，我们的测试迟早也会失败。失败是一件好事——测试失败为工程师提供了有用的信号，也是单元测试提供价值的主要方式之一。

测试失败通常有两个原因<sup>3</sup>：

- 被测系统有问题或不完整。这个结果正是测试的目的是：提醒您注意错误，以便您可以修复它们。
- 测试本身存在缺陷。在这种情况下，被测系统没有任何问题，但测试指定不正确。如果这是一个现有测试，而不是您刚刚编写的测试，则意味着测试很脆弱。上一节讨论了如何避免脆弱测试，但很少可能完全消除它们。

当测试失败时，工程师的首要任务是确定失败属于哪种情况，然后诊断实际问题。工程师能够做到这一点的速度取决于测试的清晰度。清晰的测试是指工程师能够立即清楚其存在的目的和失败的原因。当测试失败的原因不明显或难以弄清楚最初编写测试的原因时，测试就无法达到清晰的程度。清晰的测试还带来其他好处，例如记录被测系统并更容易作为新测试的基础

测试。

测试清晰度随着时间的推移变得非常重要。测试的寿命通常比编写测试的工程师更长，并且系统的要求和理解会随着时间而发生微妙的变化。失败的测试可能是几年前由一位开发人员编写的，这是完全可能的。

---

<sup>3</sup>这也是测试可能“不稳定”的两个原因。要么被测系统存在不确定性问题故障，或者测试存在缺陷，有时应该通过却失败了。

工程师不再是团队的一员,无法弄清楚其目的或如何修复它。这与不明确的生产代码形成鲜明对比,通过查看调用它的内容以及删除它时会造成什么问题,您通常可以确定其目的。对于不明确的测试,您可能永远无法理解其目的,因为删除测试除了(可能)在测试覆盖率中引入一个微妙的漏洞外,不会产生任何影响。

最糟糕的情况是,当工程师无法找到修复方法时,这些晦涩难懂的测试最终会被删除。删除此类测试不仅会导致测试覆盖率出现漏洞,还表明该测试在其存在的整个时期内(可能是数年)都没有提供任何价值。

为了使测试套件能够随着时间的推移而扩展和发挥作用,套件中的每个测试都应尽可能清晰。本节探讨了实现清晰度的测试技巧和思考方式。

使您的测试完整而简洁有助于测试实现清晰度的两个高级属

性是**完整性和简洁性**。如果测试主体包含了读者理解其结果所需的全部信息,则测试是完整的。如果测试不包含其他分散注意力或不相关的信息,则测试是简洁的。[示例 12-6](#)展示了一个既不完整也不简洁的测试:

### 例 12-6. 一个不完整且混乱的测试

```
@Test
public void shouldPerformAddition() {
    计算器 calculator = new Calculator(new RoundingStrategy());
    “未使用”， ENABLE_COSINE_FEATURE, 0.01, calculusEngine, false);
    int result = calculator.calculate(new TestCalculation()); assertThat(result).isEqualTo(5);
    这个数字是从哪里来的?
}
```

测试将大量不相关的信息传递到构造函数中,而测试的真正重要部分隐藏在辅助方法中。通过明确辅助方法的输入,可以使测试更加完整,通过使用另一个辅助方法隐藏构造计算器的不相关细节,可以使测试更加简洁,如[示例 12-7 所示](#)。

### 例 12-7. 一个完整、简洁的测试

```
@Test
public void shouldPerformAddition() {
    计算器 calculator = new Calculator(); int result =
    calculator.calculate(new Calculation(2, Operation.PLUS, 3));
```

```
    断言(结果).isEqualTo(5);
}
```

我们稍后讨论的想法,尤其是围绕代码共享的想法,将与完整性和简洁性紧密相关。特别是,如果能使测试更清晰,那么违反 DRY (不要重复自己)原则往往是值得的。请记住:测试主体应包含理解测试所需的所有信息,而不包含任何无关或分散注意力的信息。

测试行为,而不是方法许多工程师的

第一反应是尝试将测试结构与代码结构相匹配,以便每个生产方法都有相应的测试方法。这种模式一开始可能很方便,但随着时间的推移,它会导致问题:随着被测试方法变得越来越复杂,其测试也会变得越来越复杂,并且越来越难以推理。例如,考虑[示例 12-8 中的代码片段](#),它显示了事务的结果。

### 例 12-8. 交易片段

```
public void displayTransactionResults(用户 user, 交易 transaction) {
    ui.showMessageDialog( 您购买了 if +交易.获取商品名称());
    (user.getBalance() < LOW_BALANCE_THRESHOLD) {
        ui.showMessageDialog( 警告:您的余额不足! );
    }
}
```

找到一个涵盖该方法可能显示的两条消息的测试并不罕见,如[示例 12-9 所示](#)。

### 例 12-9. 方法驱动测试

```
@Test
public void testDisplayTransactionResults() {
    transactionProcessor.displayTransactionResults
        (newUserWithBalance
            (LOW_BALANCE_THRESHOLD.plus(美元(2))), 
            新的交易("某些商品", 美元(3)));
    assertThat(ui.getText()).contains( 您购买了某些商品 );
    assertThat(ui.getText()).contains( 您的余额不足 );
}
```

通过这样的测试,测试很可能一开始只涵盖第一种方法。

后来,一位工程师在添加第二条消息时扩展了测试(违反了我们之前讨论过的不变测试的想法)。这种修改开创了一个坏先例:随着测试方法变得越来越复杂,并实现更多

功能,其单元测试将变得越来越复杂,并且越来越难以处理。

问题是,围绕方法构建测试自然会导致测试不明确,因为单个方法通常在幕后会执行几项不同的操作,并且可能存在一些棘手的边缘和极端情况。有一种更好的方法:不要为每种方法编写测试,而是为每种行为编写测试。4行为是系统对在特定状态下如何响应一系列输入做出的任何保证。5行为通常可以使用“**给定**”、“**何时**”和“**然后**”等词来表达:“假设银行账户为空,当试图从中提款时,交易将被拒绝。”方法和行为之间的映射是多对多的:大多数非平凡方法实现多个行为,而一些行为依赖于多个方法的交互。前面的示例可以使用行为驱动测试重写,如**示例 12-10 所示**。

#### 例 12-10. 行为驱动的测试

```

@Test
public void displayTransactionResults_showsItemName()
    { transactionProcessor.displayTransactionResults( new User(), new
        Transaction( Some Item )); assertThat(ui.getText()).contains( 您
        购买了某些商品 );
    }

@Test
public void displayTransactionResults_showsLowBalanceWarning()

    { transactionProcessor.displayTransactionResults( newUserWithBalance( LOW_BALANCE_THRESHOLD.plus(dollars(2))), 
        新的交易 ( “某些项目”, 美元 (3) ) ; assertThat (ui.getText
        () ).包含 ( “您的余额不足” );
    }
}

```

拆分单个测试所需的额外样板是**值得的**,由此产生的测试比原始测试更清晰。行为驱动测试往往比面向方法的测试更清晰,原因有几个。首先,它们读起来更像自然语言,可以自然地理解,而不需要费力的心理分析。其次,它们更清楚地表达**因果关系**因为每个测试的范围都比较有限。最后,由于每个测试都很简短且描述性强,因此更容易看到哪些功能已经测试过,并鼓励工程师添加新的精简测试方法,而不是堆积在现有方法上。

---

<sup>4</sup>请参阅<https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html>和<https://dannorth.net/introduction-bdd/>。

<sup>5</sup>此外,特征(就产品意义而言)可以表示为行为的集合。

构建测试以强调行为将测试与行为而非

方法相结合的想法会显著影响测试的结构。请记住,每个行为都有三个部分:定义系统设置方式的“给定”组件、定义要对系统采取的操作的“时间”组件以及验证结果的“然后”组件。<sup>6</sup>当这种结构明确时,测试最清晰。一些框架(如Cucumber)和斯波克直接嵌入given/when/then。其他语言可以使用户空格和可选注释来突出结构,如示例 12-11 所示。

## 例 12-11. 一个结构良好的测试

```
@Test
public void transferFundsShouldMoveMoneyBetweenAccounts() {
    // 给定两个账户,初始余额分别为 150 美元和 20 美元
    账户account1 = newAccountWithBalance(usd(150));
    账户account2 = newAccountWithBalance(usd(20));

    // 从第一个账户向第二个账户转账 100 美元bank.transferFunds(account1, account2, usd(100));

    // 然后新的账户余额应该反映转移assertThat(account1.getBalance()).isEqualTo(usd(50));
    assertThat(account2.getBalance()).isEqualTo(usd(120));
}
```

在简单的测试中,这种级别的描述并不总是必要的,通常省略注释并依靠空格使各部分清晰就足够了。但是,明确的注释可以使更复杂的测试更容易理解。这种模式使得在三个粒度级别上阅读测试成为可能:

1. 读者可以通过查看测试方法名称(如下所述)来了解正在测试的行为的粗略描述。
2. 如果这还不够,读者可以查看给定/何时/然后注释以获得行为的正式描述。
3. 最后,读者可以查看实际代码,以准确了解该行为是如何实现的表达式。

这种模式最常见的违反方式是将断言散布在对被测系统的多个调用之间(即组合“when”和“then”块)。合并

---

<sup>6</sup>这些部分有时被称为“安排”、“行动”和“断言”。

以这种方式使用的“then”和“when”块会使测试不太清晰,因为它很难区分正在执行的操作和预期结果。

当测试确实需要验证多步骤流程中的每个步骤时,可以定义交替的when/then块序列。也可以通过使用“and”一词将长块分开,使其更具描述性。[示例 12-12](#)显示了相对复杂的行为驱动测试可能是什么样子。

### 例 12-12. 在测试中交替使用 when/then 块

```
@Test
public void shouldTimeOutConnections() {
    // 给定两个用户
    用户 user1 = newUser();
    用户 user2 = newUser();

    // 以及一个超时时间为 10 分钟的空连接池Pool pool = newPool(Duration.minutes(10));

    // 当将两个用户都连接到池时pool.connect(user1);
    pool.connect(user2);

    // 那么池应该有两个连接assertThat(pool.getConnections()).hasSize(2);

    // 等待20分钟时
    clock.advance(Duration.minutes(20));

    // 那么池应该没有连接assertThat(pool.getConnections()).isEmpty();

    // 并且每个用户都应该断开连接
    assertThat(user1.isConnected()).isFalse();
    assertThat(user2.isConnected()).isFalse();
}
```

编写此类测试时,请务必小心,确保您不会无意中同时测试多个行为。每个测试应仅涵盖一个行为,并且绝大多数单元测试只需要一个“when”和一个“then”块。

### 以被测试的行为命名测试面向方法的测试

通常以被测试的方法命名(例如, updateBalance方法的测试通常称为testUpdateBalance)。通过更集中的行为驱动测试,我们拥有更多的灵活性,并且有机会在测试名称中传达有用的信息。测试名称非常重要:它通常是故障报告中可见的第一个或唯一一个标记,因此它是您与测试者沟通的最佳机会。

测试中断时描述问题。这也是表达测试意图的最直接方式。

测试的名称应该概括测试的行为。好的名称既能描述系统上正在采取的操作,又能描述预期结果。测试名称有时会包含其他信息,例如在对其采取行动之前系统或环境的状态。某些语言和框架允许测试相互嵌套并使用字符串命名,从而使这一点比其他语言和框架更容易实现,例如示例 12-13 中使用了 Jasmine。

### 例 12-13. 一些嵌套命名模式的示例

```
描述 ( “乘法” ,函数 (){  
    describe( 用正数 ,function(){  
        var positiveNumber = 10; it( 与  
        另一个正数相加为正数 ,function() {  
            expect(positiveNumber * 10).toBeGreaterThan(0);}); it( 带有负  
        数 ,function() { expect(positiveNumber * -10).toBeLessThan(0);});});  
        describe( 带有负数 ,function() {  
  
            var negativeNumber = 10; it( 与  
            一个正数一起为负数 ,function() { expect(negativeNumber *  
            10).toBeLessThan(0);}); it( 与另一个负数一起为正数 ,  
            function() { expect(negativeNumber * -10).toBeGreaterThan(0);});});});
```

其他语言要求我们将所有这些信息编码在方法名称中,从而产生如**示例 12-14 所示的方法命名模式**。

#### 例 12-14. 一些示例方法命名模式

```
multiplyingTwoPositiveNumbersShouldReturnAPositiveNumber
multiply_positiveAndNegative_returnsNegative
divide_byZero_throwsException
```

此类名称比我们通常希望在生产代码中为方法编写的名称要冗长得多,但用例不同:我们永远不需要编写调用这些方法的代码,并且它们的名称经常需要在报告中供人阅读。因此,额外的冗长是合理的。

只要在单个测试类中一致使用,许多不同的命名策略都是可以接受的。如果您遇到困难,一个很好的技巧是尝试以“should”一词开头作为测试名称。当与被测试类的名称一起使用时,此命名方案允许将测试名称读作一个句子。例如,名为shouldNotAllowWithdrawalsWhenBalanceIsEmpty的BankAccount类的测试可以读作“BankAccount 在余额为空时不应允许提款”。通过阅读套件中所有测试方法的名称,您应该对被测系统实现的行为有很好的了解。这样的名称还有助于确保测试专注于单一行为:如果您需要在测试名称中使用“and”,则很有可能您实际上正在测试多种行为,并且应该编写多个测试!

#### 不要在测试中加入逻辑清晰的测试

在检查时显然是正确的;也就是说,只需看一眼就可以明显看出测试正在做正确的事情。这在测试代码中是可能的,因为每个测试只需要处理一组特定的输入,而生产代码必须通用化才能处理任何输入。对于生产代码,我们可以编写测试来确保复杂的逻辑是正确的。但测试代码没有这种奢侈。如果你觉得需要编写测试来验证你的测试,那就出问题了!

复杂性通常以逻辑的形式引入。逻辑通过编程语言的命令式部分(如运算符、循环和条件)定义。当一段代码包含逻辑时,您需要进行一些心理计算来确定其结果,而不仅仅是从屏幕上读取它。不需要太多逻辑就可以使测试更难以推理。例如,**示例 12-15 中的测试对您来说看起来正确吗?**

### 例子 12-15. 隐藏错误的逻辑

```
@Test
public void shouldNavigateToAlbumsPage() {
    字符串baseUrl = "http://photos.google.com/" ; Navigator nav = new
    Navigator (baseUrl) ; nav.goToAlbumPage () ; assertThat
        (nav.getCurrentUrl () ) .
    isEqualTo (baseUrl + "/albums" ) ;
}
```

这里没有太多逻辑:实际上只有一个字符串连接。但如果我们通过删除那一点逻辑来简化测试,错误就会立即显现出来,如示例 12-16 所示。

### 示例 12-16. 没有逻辑的测试暴露了 bug

```
@Test
public void shouldNavigateToPhotosPage() { Navigator nav =
    new Navigator( http://photos.google.com/ ) ; nav.goToPhotosPage();

    assertThat(nav.getCurrentUrl()).isEqualTo( http://
        photos.google.com//albums ) ; // 啊呀!
}
```

当写出整个字符串时,我们可以立即看到 URL 中应该有两个斜杠,而不是一个。如果生产代码犯了类似的错误,则此测试将无法检测到错误。复制基本 URL 是为了让测试更具描述性和意义而付出的一点小代价(请参阅本章后面关于 DAMP 与 DRY 测试的讨论)。

如果人类不擅长发现字符串连接中的错误,那么我们在发现来自更复杂的编程结构(如循环和条件)的错误方面就更糟糕了。教训很明显:在测试代码中,坚持使用直线代码而不是巧妙的逻辑,并且当它使测试更具描述性和意义时,考虑容忍一些重复。我们将在本章后面讨论有关重复和代码共享的想法。

#### 编写清晰的失败信息清晰度的最后一个

方面与测试的编写方式无关,而是与测试失败时工程师看到的内容有关。在理想情况下,工程师只需阅读日志或报告中的失败信息即可诊断问题,而无需查看测试本身。良好的失败信息包含的信息与测试名称大致相同:它应该清楚地表达期望结果、实际结果以及任何相关参数。

以下是错误失败消息的示例：

测试失败:帐户已关闭

测试失败是因为账户被关闭了,还是账户预计会被关闭而测试失败是因为账户没有被关闭?更好的失败消息可以清楚地区分预期状态和实际状态,并提供有关结果的更多背景信息:

```
预期帐户处于 CLOSED 状态,但收到帐户 <{name: my-account ,state:  
OPEN }
```

好的库可以帮助更轻松地编写有用的失败消息。考虑Java 测试中[示例 12-17](#)中的断言,其中第一个使用经典的 JUnit 断言,第二个使用[Truth](#), Google 开发的断言库:

#### 例 12-17. 使用 Truth 库的断言

```
设置 <String> colors = ImmutableSet.of( red , green , blue );  
assertTrue(colors.contains( orange ));// JUnit  
assertThat(colors).contains( orange );// 真值
```

因为第一个断言只接收一个布尔值,所以它只能给出一个通用的错误消息,如“预期为 `<true>`,但结果为 `<false>`”,这在失败的测试输出中并不是很有用。因为第二个断言明确接收了断言的主题,所以它能够给出**更有用的错误消息**:断言错误:`<[红色,绿色,蓝色]>`应该包含`<橙色>`。”

并非所有语言都有这样的帮助程序,但应该始终能够手动指定失败消息中的重要信息。例如,Go 中的测试断言通常看起来像[示例 12-18](#)。

#### 示例 12-18. Go 中的测试断言

```
结果:= Add(2, 3)如果结果!=  
5 {  
    t.Errorf( Add(2, 3) = %v, 想要 %v ,结果, 5 )
```

## 测试和代码共享 :DAMP,而不是DRY

编写清晰的测试和避免脆弱性的最后一个方面与代码共享有关。大多数软件都试图实现一项名为 DRY 的原则——“不要重复自己”。DRY 指出,如果每个概念都以规范的方式在一个地方表示,并且代码重复被保持在最低限度,软件将更容易维护。这种方法在使更改更容易方面尤其有价值,因为工程师只需要更新一段代码,而不是追踪多个引用。缺点

这种合并的缺点是它会使代码不清楚,需要读者遵循引用链才能理解代码在做什么。

在正常的生产代码中,这个缺点通常是为了让代码更易于更改和使用而付出的一点小代价。但是这种成本/收益分析在测试代码的背景下会有所不同。好的测试被设计为稳定的,事实上,当被测试的系统发生变化时,你通常希望它们会中断。所以 DRY 在测试代码方面并没有那么多好处。同时,测试的复杂性成本更大:生产代码有测试套件的好处,可以确保它在变得复杂时继续工作,而测试必须独立存在,如果它们不是不言而喻的正确,就会有错误的风险。如前所述,如果测试开始变得足够复杂,以至于感觉它们需要自己的测试来确保它们正常工作,那就出问题了。

测试代码不应完全遵循 DRY 原则,而应努力遵循DAMP 原则,即推广“描述性且有意义的短语”。测试中存在少量重复是可以的,只要这种重复能让测试更简单、更清晰即可。为了说明这一点,示例 12-19 展示了一些过于遵循 DRY 原则的测试。

### 示例 12-19. 过于 DRY 的测试

```

@Test
public void shouldAllowMultipleUsers() {
    列表 <User> users = createUsers(false, false);
    论坛 forum = createForumAndRegisterUsers(用户); validateForumAndUsers(论
    坛,用户);
}

@Test
public void shouldNotAllowBannedUsers() {
    列表<用户>用户=创建用户(true);
    论坛 forum = createForumAndRegisterUsers(用户); validateForumAndUsers(论
    坛,用户);
}

// 更多测试...

私有静态List<User> createUsers (boolean ... banned) { List<User> users = new
ArrayList<>(); for (boolean isBanned : banned)
{ users.add(newUser() .setState(isBanned ?
    State.BANNED :
    State.NORMAL) .build()); }

}
返回用户;
}

私有静态论坛createForumAndRegisterUsers (List <User> users) {
    论坛 forum = new Forum();
}

```

```

对于 (用户用户 : 用户) {尝试

    { forum.register (用户) ;
    }捕获 (BannedUserException 被忽略) {}

}

返回论坛;
}

private static void validateForumAndUsers (Forum forum, List<User> users) {
    断言 (forum.isReachable () ) .isTrue () ;对于 (用户用户 : 用户)
    (用户) {断言 (forum.hasRegisteredUser
        (用户) )
        .isEqualTo(用户.getState() == State.BANNED);
    }
}

```

根据前面关于清晰度的讨论,这段代码中的问题应该显而易见。首先,虽然测试主体非常简洁,但它们并不完整:重要的细节隐藏在辅助方法中,读者必须滚动到文件的完全不同部分才能看到。这些辅助方法还充满了逻辑,使得它们很难一目了然地验证(你发现错误了吗?)。当使用 DAMP 重写测试时,测试变得更加清晰,如示例 12-20 所示。

### 示例 12-20. 测试应该是 DAMP

```

@Test
public void shouldAllowMultipleUsers() {用户用户1 =
    newUser().setState(State.NORMAL).build();用户用户2 =
    newUser().setState(State.NORMAL).build();

    论坛 forum = new Forum();
    forum.register(user1);
    forum.register(user2);

    断言 (forum.hasRegisteredUser (用户1) ) .isTrue () ;断言
    (forum.hasRegisteredUser (用户2) ) .isTrue () ;
}

@Test
public void shouldNotRegisterBannedUsers() {
    用户 user = newUser().setState(State.BANNED).build();

    论坛 forum = new Forum();尝试

    { forum.register(user);
    }捕获 (BannedUserException 被忽略) {}

    断言 (forum.hasRegisteredUser (用户) ) .isFalse () ;
}

```

这些测试有更多重复,测试主体也更长一些,但额外的冗长是值得的。每个单独的测试都更有意义,无需离开测试主体就可以完全理解。这些测试的读者可以确信测试会按照其声称的方式进行,并且不会隐藏任何错误。

DAMP 不是 DRY 的替代品,而是 DRY 的补充。辅助方法和测试基础结构仍可以帮助使测试更简洁,分解出与被测试的特定行为无关的重复步骤。重要的是,这种重构应该着眼于使测试更具描述性和意义,而不仅仅是减少了重复。本节的其余部分将探讨跨测试共享代码的常见模式。

## 共同价值观

许多测试都是通过定义一组测试要使用的共享值,然后定义涵盖这些值如何相互作用的各种情况的测试来构建的。

**示例 12-21**说明了此类测试的具体内容。

### 示例 12-21. 具有歧义名称的共享值

```
私有静态最终帐户ACCOUNT_1 = Account.newBuilder() . setState (AccountState.OPEN) .
    setBalance (50) . build () ;

私有静态最终帐户ACCOUNT_2 = Account.newBuilder() . setState (AccountState.CLOSED) .
    setBalance (0) . build () ;

私有静态最终Item ITEM =
    Item.newBuilder() . setName( Cheeseburger ) . setPrice(100) . build();

// 数百行其他测试...

@Test
public void canBuyItem_returnsFalseForClosedAccounts() {
    断言 (store.canBuyItem (ITEM, ACCOUNT_1) ) .isFalse () ;
}

@Test
public void canBuyItem_returnsFalseWhenBalanceInsufficient()
    { assertThat(store.canBuyItem(ITEM, ACCOUNT_2)).isFalse();
}
```

这种策略可以使测试非常简洁,但随着测试套件的增长,它会带来问题。首先,很难理解为什么为测试选择了某个特定值。在**示例 12-21 中**,测试名称幸运地阐明了正在测试哪些场景,但您仍然需要向上滚动到定义以确认ACCOUNT\_1和ACCOUNT\_2适用于这些场景。更具描述性的常量名称(例如,

测试用例的名称（例如，CLOSED\_ACCOUNT和ACCOUNT\_WITH\_LOW\_BALANCE）有一点帮助，但它们仍然使查看被测试值的确切细节变得更加困难，并且重用这些值的便利性可以鼓励工程师这样做，即使名称不能准确描述测试需要什么。

工程师通常倾向于使用共享常量，因为在每个测试中构造单独的值可能会很冗长。实现此目标的更好方法是使用辅助方法构造数据（参见示例 12-22）要求测试作者仅指定他们关心的值，并为所有其他值设置合理的默认值<sup>7</sup>。这种构造在支持命名参数的语言中很容易实现，但没有命名参数的语言可以使用诸如 Builder 模式之类的构造来模拟它们（通常借助 AutoValue 等工具）：

## 例 12-22. 使用辅助方法共享值

```
# 辅助方法通过为每个参数定义任意默认值来包装构造函数。def newContact( firstName= Grace , lastName= Hopper ,
phoneNumber= 555-123-4567 ):
return Contact(firstName,
lastName, phoneNumber)

# 测试调用助手，仅为它们关心的参数指定值。def test.FullNameShouldCombineFirstAndLastNames(self):
    def contact = newContact(firstName= Ada , lastName= Lovelace )
    self.assertEqual(contact.fullName(), Ada Lovelace )

//像 Java 这样的不支持命名参数的语言可以通过返回一个可变的“构建器”对象来模拟它们，该对象表示正在构造的值。private static
Contact.Builder newContact() { return

    Contact.newBuilder().setFirstName( Grace ).setLastName( Hopper ).setPhoneNumber( 555-123-4567 );
}

//然后测试调用构建器上的方法来覆盖它们关心的参数//，然后调用 build() 从构建器中获取真实值。

@Test
public void fullNameShouldCombineFirstAndLastNames() {
    联系人 contact =
        newContact().setFirstName( Ada )
```

---

<sup>7</sup>在许多情况下，稍微随机化返回不属于以下类型的字段的默认值甚至会很有用：明确设置。这有助于确保两个不同的实例不会意外地比较为相等，并且使工程师更难以对默认值进行硬编码依赖。

```

.setLastName
(" Lovelace" ) .build () ;
断言 (contact.getFullName () ) .isEqualTo ( "Ada Lovelace" ) ;
}

```

使用辅助方法构建这些值允许每个测试创建它所需的精确值,而不必担心指定不相关的信息或与其他测试冲突。

### 共享设置测试共享代码

码的相关方法是通过设置/初始化逻辑。许多测试框架允许工程师在套件中的每个测试运行之前定义要执行的方法。

如果使用得当,这些方法可以避免重复繁琐且不相关的初始化逻辑,从而使测试更清晰、更简洁。如果使用不当,这些方法可能会将重要细节隐藏在单独的初始化方法中,从而损害测试的整体性。

设置方法的最佳用例是构造测试对象及其协作者。当大多数测试不关心用于构造这些对象的特定参数并可以让它们保持默认状态时,这很有用。同样的想法也适用于对测试替身的返回值进行存根,我们将在[第 13 章中更详细地探讨这个概念。](#)

使用 setup 方法的一个风险是,如果测试开始依赖于 setup 中使用的特定值,则它们可能会导致测试不明确。例如,[示例 12-23](#)中的测试似乎不完整,因为测试的读者需要去寻找字符串 “Donald Knuth”的来源。

### 示例 12-23. 设置方法中值的依赖关系

[私人名称服务名称服务;私人用户商店用户商店;](#)

```

@Before
public void setUp()
{
    nameService = new NameService();
    nameService.set( user1 , Donald Knuth );
    userStore = new UserStore(nameService);
}

// [... 数百行测试...]

@Test
public void shouldReturnNameFromService() { UserDetails user
    = userStore.get( user1 );
    assertThat(user.getName()).isEqualTo( Donald Knuth );
}

```

此类明确关注特定值的测试应直接声明这些值,必要时可覆盖设置方法中定义的默认值。结果测试包含稍多的重复内容,如[示例 12-24 所示](#),但结果更具描述性和意义。

### 示例 12-24. 覆盖设置 mMethods 中的值

私人名称服务名称服务;私人用户商店用户商店;

```
@Before
public void setUp() {
    nameService = new NameService();
    nameService.set( user1 , Donald Knuth ); userStore =
        new UserStore(nameService);
}

@Test
public void shouldReturnNameFromService() {
    nameService.set( user1 , 玛格丽特汉密尔顿 ); UserDetails user
    = userStore.get( user1 );
    assertThat(user.getName()).isEqualTo( 玛格丽特汉密尔顿 );
}
```

### 共享帮助程序和验证 代码在测试间

共享的最后一一种常见方式是通过从测试方法主体调用的“帮助程序方法”。我们已经讨论了帮助程序方法如何成为简洁构建测试值的有用方法。这种用法是合理的,但其他类型的帮助程序方法可能很危险。

一种常见的辅助程序类型是针对被测系统执行一组通用断言的方法。极端的例子是在每个测试方法结束时调用的验证方法,该方法针对被测系统执行一组固定检查。这种验证策略可能是一个坏习惯,因为使用这种方法的测试不太受行为驱动。使用这样的测试,很难确定任何特定测试的意图,也很难推断作者在编写测试时考虑的具体情况。当引入错误时,这种策略也会使它们更难定位,因为它们经常会导致大量测试开始失败。

然而,更集中的验证方法仍然有用。最好的验证辅助方法断言关于其输入的单一概念事实,这与涵盖一系列条件的通用验证方法不同。当它们验证的条件在概念上很简单但需要循环或条件逻辑来实现时,这种方法特别有用,如果将其包含在测试方法的主体中,则会降低清晰度。例如,示例 12-25 中的辅助方法可能在涵盖几个不同情况的测试中很有用

帐户访问。

### 示例 12-25. 一个概念上简单的测试

```
private void assertUserHasAccessToAccount(用户 user, 帐户 account) {
    对于 (long userId : account.getUsersWithAccess () ) {如果 (user.getId () == userId) {返回;
    }
}
失败 (用户.getName () + " 无法访问 " + 帐户.getName());
```

定义测试基础设施到目前为止,我们讨论的

技术涵盖了在单个测试类或套件中跨方法共享代码。有时,在多个测试套件之间共享代码也很有用。我们将这种代码称为测试基础设施。虽然它通常在集成或端到端测试中更有价值,但在某些情况下,精心设计的测试基础设施可以使单元测试更容易编写。

与单个测试套件中的代码共享相比,自定义测试基础架构必须更加谨慎。在许多方面,测试基础架构代码与生产代码的相似性要高于其他测试代码,因为它可能有许多依赖它的调用者,并且很难在不引入中断的情况下进行更改。大多数工程师在测试自己的功能时不会对通用测试基础架构进行更改。测试基础架构需要被视为独立的产品,因此,测试基础架构必须始终有自己的测试。

当然,大多数工程师使用的大多数测试基础设施都是以 JUnit 等知名第三方库的形式出现的。有大量此类库可用,组织内应尽早普遍地对它们进行标准化。例如,谷歌多年前规定 Mockito 是新 Java 测试中应使用的唯一模拟框架,并禁止新测试使用其他模拟框架。这项法令当时引起了一些习惯使用其他框架的人的抱怨,但今天,它被普遍视为一项很好的举措,使我们的测试更易于理解和使用。

## 结论

单元测试是我们软件工程师必须使用的最强大的工具之一,它可以确保我们的系统在遇到意外变化时能够持续运行。但是能力越大,责任越大,如果对单元测试使用不当,可能会导致系统需要付出更多努力来维护,需要付出更多努力来更改,而实际上并没有提高我们对该系统的信心。

Google 的单元测试远非完美,但我们发现遵循本章概述的实践的测试比不遵循的测试价值高出几个数量级。我们希望它们能帮助您提高自己的测试质量!

## TL;DR

- 努力实现不变的测试。 · 通过公共 API 进行测试。 · 测试状态,而不是交互。
- 让您的测试完整而简洁。 · 测试行为,而不是方法。
- 构建测试以强调行为。 · 以被测试的行为命名测试。 · 不要在测试中加入逻辑。 · 编写清晰的失败信息。 · 共享测试代码
- 时遵循 DAMP 而不是 DRY。

## 第十三章

## 测试替身

作者:Andrew Trenk 和 Dillon Bly  
编辑:Tom Mansreck

单元测试是保持开发人员工作效率和减少代码缺陷的重要工具。虽然对于简单的代码来说,编写单元测试很容易,但随着代码变得越来越复杂,编写单元测试变得越来越困难。

例如,想象一下尝试为一个函数编写测试,该函数将请求发送到外部服务器,然后将响应存储在数据库中。编写少量测试可能只需付出一些努力即可完成。但如果需要编写数百或数千个这样的测试,您的测试套件可能需要数小时才能运行,并且可能会因随机网络故障或测试覆盖彼此的数据等问题而变得不稳定。

在这种情况下,测试替身就派上用场了。**测试替身**是一个对象或函数,可以在测试中代替实际实现,类似于电影中的替身演员。测试替身的使用通常被称为模拟,但我们在本章中避免使用该术语,因为我们将会看到,该术语还用于指代测试替身的更具体方面。

也许最明显的测试替身类型是对象的更简单实现,其行为与真实实现类似,例如内存数据库。

其他类型的测试替身可以验证系统的特定细节,比如通过轻松触发罕见的错误情况,或确保调用重量级函数而不实际执行该函数的实现。

前两章介绍了小型测试的概念,并讨论了为什么它们应该成为测试套件中测试的主体。然而,由于跨多个进程或机器进行通信,生产代码通常不符合小型测试的限制。测试替身可以比真实测试更轻量级

实现,允许您编写许多执行快速且稳定的小测试。

## 测试替身对软件开发的影响

使用测试替身会给软件开发带来一些复杂性,需要做出一些权衡。本章将更深入地讨论这里介绍的概念:

### 可测试性

要使用测试替身,代码库需要设计为可测试的。测试应该能够用测试替身替换实际实现。例如,调用数据库的代码需要足够灵活,以便能够使用测试替身代替实际数据库。如果代码库在设计时没有考虑测试,而您后来决定需要测试,则可能需要做出重大承诺来重构代码以支持使用测试替身。

### 适用性 虽然

正确使用测试替身可以大大提高工程速度,但使用不当会导致测试变得脆弱、复杂且效率低下。当在大型代码库中不正确地使用测试替身时,这些缺点会被放大,可能会导致工程师生产力的重大损失。在许多情况下,测试替身并不合适,工程师应该选择使用真实的实现。

### 保真度

保真度指的是测试替身的行为与它所替代的真实实现的行为的相似程度。如果测试替身的行为与真实实现有显著差异,那么使用该测试替身的测试可能不会提供太大价值。例如,想象一下尝试用测试替身为数据库编写一个测试,该测试替身会忽略添加到数据库的任何数据并始终返回空结果。但完美保真度可能并不可行;测试替身通常需要比真实实现简单得多,才适合用于测试。在许多情况下,即使没有完美保真度,使用测试替身也是合适的。使用测试替身的单元测试通常需要通过执行真实实现的更大范围的测试来补充。

## 谷歌的测试替身

在 Google,我们已经看到了无数个例子,这些例子表明测试替身可以为代码库带来生产力和软件质量方面的好处,以及如果使用不当可能会造成的负面影响。我们在 Google 遵循的做法是根据这些经验随着时间的推移而演变的。从历史上看,我们几乎没有关于如何

有效地使用测试替身,但随着我们看到许多团队的代码库中出现常见模式和反模式,最佳实践也在不断发展。

我们从惨痛经历中吸取的一个教训是过度使用模拟框架的危险,它允许你轻松创建测试替身(我们将在本章后面详细讨论模拟框架)。当模拟框架首次在 Google 投入使用时,它们看起来就像一把能打穿所有钉子的锤子。它们使得针对孤立的代码片段编写高度集中的测试变得非常容易,而不必担心如何构建该代码的依赖关系。直到几年后经过无数次测试,我们才开始意识到这种测试的成本:虽然这些测试很容易编写,但我们遭受了巨大的损失,因为它们需要不断的努力来维护,而且很少发现错误。Google 的钟摆现在已经开始摆向另一个方向,许多工程师避免使用模拟框架,转而编写更现实的测试。

尽管本章讨论的实践在 Google 内部得到了普遍认可,但实际应用情况在各个团队之间却存在很大差异。这种差异源于工程师对这些实践的了解不一致、现有代码库中存在不遵循这些实践的情形,或者团队只考虑短期内最简单的做法而不考虑长期影响。

## 基本概念

在深入探讨如何有效使用测试替身之前,我们先来介绍一些与之相关的基本概念。这些概念为我们本章后面将要讨论的最佳实践奠定了基础。

测试替身示例想象一下一个需要

处理信用卡付款的电子商务网站。其核心可能类似于[示例 13-1 所示的代码](#)。

### 例 13-1. 信用卡服务

```
类PaymentProcessor {私有
    CreditCardService creditCardService;
    ...
    boolean makePayment(信用卡creditCard,金额) {
        if (creditCard.isExpired()) { return false; } boolean success =
            creditCardService.chargeCreditCard(creditCard, amount);返回成功;
    }
}
```

在测试中使用真实的信用卡服务是不可行的（想象一下运行测试所产生的所有交易费用！），但可以使用测试替身来代替它来模拟真实系统的行为。[示例 13-2](#)中的代码展示了一个非常简单的测试替身。

### 示例 13-2. 一个简单的测试替身

```
TestDoubleCreditCardService类实现CreditCardService {
    @Override
    public boolean chargeCreditCard(CreditCard creditCard, Money amount) {
        返回 true;
    }
}
```

虽然这个测试替身看起来不太有用，但在测试中使用它仍然可以让我们测试makePayment()方法中的一些逻辑。例如，在[示例 13-3 中](#)，我们可以验证当信用卡过期时该方法是否正常运行，因为测试执行的代码路径不依赖于信用卡服务的行为。

### 示例 13-3. 使用测试替身

```
@Test public void cardIsExpired_returnFalse() {
    布尔成功= paymentProcessor.makePayment(EXPIRED_CARD, AMOUNT); assertThat(success).isFalse(); }
```

本章接下来的部分将讨论如何在比这更复杂的情况下使用测试替身。

## 接缝

代码被认为是可测试的如果代码的编写方式使得为代码编写单元测试成为可能。[接缝](#)是一种通过允许使用测试替身来使代码可测试的方法。它可以对被测系统使用不同的依赖项，而不是使用生产环境中使用的依赖项。

[依赖注入](#)是一种引入接缝的常用技术。简而言之，当一个类使用依赖注入时，它需要使用的任何类（即该类的依赖项）都会传递给它，而不是直接实例化，从而可以在测试中替换这些依赖项。

[例 13-4](#)展示了一个依赖注入的例子。构造函数并没有创建CreditCardService 的实例，而是接受一个实例作为参数。

## 例 13-4. 依赖注入

```
类PaymentProcessor {私有
    CreditCardService creditCardService;

    付款处理器（信用卡服务信用卡服务） {
        这个.信用卡服务=信用卡服务;
    }
    ...
}
```

调用此构造函数的代码负责创建适当的Credit CardService实例。生产代码可以传入与外部服务器通信的CreditCardService实现,而测试可以传入测试替身,如[示例 13-5 所示](#)。

## 示例 13-5 传递测试替身

```
PaymentProcessor paymentProcessor = new
    PaymentProcessor(new TestDoubleCreditCardService());
```

为了减少与手动指定构造函数相关的样板代码,可以使用自动依赖注入框架来自动构建对象图。在 Google, [Guice](#)和[Dagger](#)是常用于 Java 代码的自动化依赖注入框架。

使用动态类型语言(例如 Python 或 JavaScript),可以动态替换单个函数或对象方法。依赖注入在这些语言中并不那么重要,因为此功能使得可以在测试中使用依赖项的实际实现,同时仅覆盖依赖项中不适合测试的函数或方法。

编写可测试代码需要前期投资。这在代码库的生命周期早期尤其重要,因为越晚考虑可测试性,就越难应用于代码库。没有考虑测试的代码通常需要重构或重写,然后才能添加适当的测试。

模拟框架模拟框架是一个软件库,它

使在测试中创建测试替身变得更加容易;它允许您用模拟替换对象,模拟是测试替身,其行为在测试中内联指定。使用模拟框架可以减少样板代码,因为您不需要在每次需要测试替身时定义一个新类。

**例 13-6**演示了 Mockito 的使用， Java 的模拟框架。

Mockito 为CreditCardService创建了一个测试替身并指示它返回一个特定的值。

#### 例 13-6. 模拟框架

```
类PaymentProcessorTest {
    ...
    付款处理器付款处理器;

    // 仅用一行代码创建 CreditCardService 的测试替身。
    @Mock CreditCardService mockCreditCardService; @Before
    public void setUp() { // 将测试替身传递给
        被测系统。 paymentProcessor = new
        PaymentProcessor(mockCreditCardService); }

    @Test public void chargeCreditCardFails_returnFalse() {
        // 为测试替身赋予一些行为:每次调用 chargeCreditCard() 方法时,它都会返回 false。方法参数
        使用 // “any()”告诉测试替身无论传递了哪些参数都返回 false。when
        (mockCreditCardService.chargeCreditCard(any(), any()))

        .thenReturn(false);  
布尔成
        功= paymentProcessor.makePayment(CREDIT_CARD, AMOUNT);断言(成功).isFalse(); }

}
```

大多数主流编程语言都有模拟框架。在 Google,我们使用 Mockito for Java,**这是 Googletest 的 googletest 组件**对于 C++,以及unittest.mock对于 Python。

尽管模拟框架有助于更轻松地使用测试替身,但它们也存在一些重大问题,因为过度使用它们通常会使代码库更难维护。我们将在本章后面介绍其中一些问题。

## 使用测试替身的技巧

使用测试替身主要有三种技术。本节简要介绍这些技术,让您快速了解它们是什么以及它们有何不同。本章后面的部分将详细介绍如何有效地应用它们。

了解这些技术之间的区别的工程师在需要使用测试替身时更有可能知道使用哪种适当的技术。

## 伪造假

货是 API 的轻量级实现,其行为与真实实现类似,但不适合生产;例如内存数据库。[示例 13-7](#)给出了伪造的示例。

### 示例 13-7. 一个简单的伪造

```
// 制造假货既快捷又简单。
AuthorizationService fakeAuthorizationService = new FakeAuthorizationService();
AccessManager accessManager = new AccessManager(fakeAuthorizationService);

// 未知的用户 ID 不应具有访问权限.assertFalse
(accessManager.userHasAccess(USER_ID));

// 将用户 ID 添加到授权服务后,应该具有访问权限。
fakeAuthorizationService.addAuthorizedUser
(new User(USER_ID)); assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

当你需要使用测试替身时,使用假货往往是理想的技术,但是对于你需要在测试中使用的对象,可能不存在假货,而且编写假货可能很有挑战性,因为你需要确保它现在和将来具有与真实实现类似的行为。

## 存根

向一个函数赋予行为的过程,该函数本身没有行为。你向函数指定要返回的确切值(也就是说,你对返回值进行存根)。

[例 13-8](#)说明了存根。Mockito 模拟框架中的 `when(...).thenReturn(...)` 方法调用指定了 `lookupUser()` 方法的行为。

### 例 13-8. 存根

```
// 传递由模拟框架创建的测试替身。
访问管理器 accessManager = new AccessManager(mockAuthorizationService) :
    // 如果返回 null,则用户 ID 不应具有访问权限.when
    (mockAuthorizationService.lookupUser(USER_ID)).thenReturn(null);
    assertThat(accessManager.userHasAccess(USER_ID)).isFalse();

    // 如果返回非空值,则用户 ID 应该具有访问权限.when
    (mockAuthorizationService.lookupUser(USER_ID)).thenReturn(USER);
    assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

存根通常通过模拟框架来完成,以减少手动创建硬编码返回值的新类所需的样板。

尽管存根是一种快速而简单的技术,但它有局限性,我们将在本章后面讨论。

交互测试  
**交互测试**是一种

无需实际调用函数实现即可验证函数调用方式的方法。如果函数调用方式不正确,则测试失败 例如,如果函数根本没有被调用,调用次数过多,或者调用时使用了错误的参数。

**例 13-9**展示了**一个交互测试实例**。Mockito模拟框架中的**verify(...)**方法用于验证**lookupUser()**是否按预期被调用。

### 例 13-9. 交互测试

```
// 传递由模拟框架创建的测试替身。
访问管理器 accessManager =新的AccessManager(mockAuthorizationService);
accessManager.userHasAccess(USER_ID);

// 如果 accessManager.userHasAccess(USER_ID) 没有调用 // mockAuthorizationService.lookupUser(USER_ID).
verify(mockAuthorizationService).lookupUser(USER_ID), 则测试将
失败；
```

与存根类似,交互测试通常通过模拟框架完成。与手动创建包含代码的新类(用于跟踪函数调用频率和传入的参数)相比,这减少了样板代码。

交互测试有时也称为**模拟**。我们在本章中避免使用该术语,因为它可能与模拟框架混淆,模拟框架可用于存根以及交互测试。

正如本章后面所讨论的,交互测试在某些情况下很有用,但应尽可能避免,因为过度使用很容易导致测试脆弱。

## 实际实施

尽管测试替身是极为宝贵的测试工具,但我们测试的首选是使用被测系统依赖项的真实实现;即与生产代码中使用的实现相同。在以下情况下,测试保真度更高:

它们执行代码的方式与在生产中执行的方式相同，并且使用真实的实现有助于实现这一点。

在 Google，我们对真实实现的偏好随着时间的推移而发展，因为我们发现过度使用模拟框架会导致测试中出现与真实实现不同步的重复代码，并使重构变得困难。

我们将在本章后面更详细地讨论这个主题。

在测试中优先考虑真实实现的做法被称为[经典测试](#)。还有一种称为模拟测试的测试风格，这种风格倾向于使用模拟框架而不是实际实现。尽管软件行业中的一些人（包括[第一个模拟框架的创建者](#)）也使用模拟测试，在 Google，我们发现这种测试方式很难扩展。它要求工程师在设计被测系统时遵循严格的指导方针，大多数 Google 工程师的默认行为都是以更适合传统测试风格的方式编写代码。

## 宁愿现实，也不愿孤立

使用真实实现作为依赖项可以使被测系统更加真实，因为这些真实实现中的所有代码都将在测试中执行。相反，使用测试替身的测试将被测系统与其依赖项隔离，因此测试不会执行被测系统依赖项中的代码。

我们更喜欢现实测试，因为它们能让我们更有信心地相信被测系统运行正常。如果单元测试过于依赖测试替身，工程师可能需要运行集成测试或手动验证其功能是否按预期运行，才能获得同样的信心水平。执行这些额外的任务会减慢开发速度，如果工程师完全跳过这些任务，甚至会导致错误溜走，因为与运行单元测试相比，执行这些任务太耗时了。

用测试替身替换类的所有依赖项会任意地将被测系统与作者碰巧直接放入类中的实现隔离开来，并排除碰巧位于不同类中的实现。但是，好的测试应该独立于实现——应该根据被测试的 API 来编写，而不是根据实现的结构来编写。

如果实际实现中存在错误，使用实际实现可能会导致测试失败。这很好！在这种情况下，您希望测试失败，因为这表明您的代码在生产中无法正常工作。有时，实际实现中的错误可能会导致一连串的测试失败，因为使用实际实现的其他测试也可能会失败。但使用好的开发人员工具，例如

持续集成 (CI) 系统,通常很容易追踪导致失败的变化。

### 案例研究:@DoNotMock在

Google,我们已经看到足够多的测试过度依赖模拟框架,从而促使在 Java 中创建@DoNotMock 注释,该注释是ErrorProne的一部分静态分析工具。此注释是 API 所有者声明“此类型不应被模拟,因为存在更好的替代方案”的一种方式。

如果工程师尝试使用模拟框架创建已注释为@DoNotMock 的类或接口的实例 (如[示例 13-10](#)所示),他们将看到一个错误,指示他们使用更合适的测试策略,例如真实实现或伪造实现。此注释最常用于足够简单、可按原样使用的值对象,以及具有精心设计的伪造对象的 API。

#### 例 13-10. @DoNotMock 注释

```
@DoNotMock( 使用SimpleQuery.create() 而不是模拟。 ) public abstract class
Query { public abstract String
    getValue();
}
```

API 所有者为什么要关心这个问题?简而言之,它严重限制了 API 所有者随时间推移更改其实现的能力。正如我们将在本章后面探讨的那样,每次使用模拟框架进行存根或交互测试时,它都会复制 API 提供的行为。

当 API 所有者想要更改他们的 API 时,他们可能会发现它在整个 Google 代码库中已被模拟了数千次甚至数万次!

这些测试替身很可能会表现出违反被模拟类型的 API 契约的行为 - 例如,对于永远不能返回 null 的方法返回 null。如果测试使用的是真实实现或虚假实现,API 所有者可以更改其实现,而无需先修复数千个有缺陷的测试。

如何决定何时使用真实实现如果真实实现快速、确定且依赖关系简单,则优先选择真  
实实现。例如,应该将真实实现用于[值对象](#)。示例包括金额、日期、地理地址或列表或地图等集合类。

然而,对于更复杂的代码,使用真实的实现通常是不可行的。

由于需要做出权衡,因此对于何时使用真实实现或测试替身可能没有确切的答案,因此您需要考虑以下因素。

### 执行时间

单元测试最重要的品质之一是速度要快 您希望能够在开发过程中不断运行它们,以便快速获得有关代码是否正常工作的反馈(并且您还希望它们在CI系统中运行时能够快速完成)。因此,当实际实现速度较慢时,测试替身非常有用。

对于单元测试来说,多慢才算太慢?如果实际实现将每个测试用例的运行时间增加一毫秒,那么很少有人会将其归类为慢。

但是如果它增加了10毫秒、100毫秒、1秒等等,会怎么样?

这里没有确切的答案 这取决于工程师是否感觉到生产力损失,以及有多少测试使用了实际实现(如果有5个测试用例,每个测试用例额外花费一秒钟可能是合理的,但如果500个则不合理)。

对于边界情况,使用真实的实现通常更简单,直到它变得太慢而无法使用,此时可以更新测试以使用测试替身。

测试的并行化也有助于减少执行时间。在Google,我们的测试基础设施使得将测试套件中的测试拆分到多台服务器上执行变得非常简单。这会增加CPU时间成本,但可以节省大量开发人员时间。我们将在[第18章中进一步讨论这一点。](#)

需要注意的另一个权衡是:使用真实实现可能会导致构建时间增加,因为测试需要构建真实实现及其所有依赖项。使用像[Bazel](#)这样的高度可扩展的构建系统可以提供帮助,因为它可以缓存未改变的构建工件。

### 决定论

测试是[确定性的](#)如果对于被测系统的给定版本,运行测试总是产生相同的结果;也就是说,测试要么总是通过,要么总是失败。相反,测试是[非确定性的](#)如果它的结果可以改变,即使被测系统保持不变。

[测试中的不确定性](#)会导致不稳定性 即使测试系统没有发生任何变化,测试也偶尔会失败。如[第11章所述](#),如果开发人员开始不信任测试结果并忽略失败,不稳定性就会损害测试套件的健康。如果使用实际实现很少导致不稳定性,则可能不需要响应,因为对工程师的干扰很小。但如果稳定性发生了,

经常,也许是时候用测试替身来代替真实的实现,因为这样做可以提高测试的保真度。

实际实现可能比测试替身复杂得多,这增加了其不确定性的可能性。例如,如果被测系统的输出因线程执行顺序的不同而不同,则使用多线程的实际实现可能偶尔会导致测试失败。

不确定性的一个常见原因是代码不够**密封**;也就是说,它依赖于测试无法控制的外部服务。例如,如果 HTTP 服务器过载或网页内容发生变化,则尝试从 HTTP 服务器读取网页内容的测试可能会失败。相反,应该使用测试替身来防止测试依赖于外部服务器。如果使用测试替身不可行,另一个选择是使用服务器的封闭实例,其生命周期由测试控制。下一章将更详细地讨论封闭实例。

不确定性的另一个例子是依赖于系统时钟的代码,因为被测系统的输出可能根据当前时间而不同。

测试可以使用对特定时间进行硬编码的测试替身,而不必依赖系统时钟。

#### 依赖项构建使用真实实现

时,需要构建其所有依赖项。例如,一个对象需要构建其整个依赖项树:它所依赖的所有对象、这些依赖对象所依赖的所有对象等等。测试替身通常没有依赖项,因此与构建真实实现相比,构建测试替身要简单得多。

举一个极端的例子,想象一下尝试在测试中创建以下代码片段中的对象。确定如何构造每个单独的对象将非常耗时。测试还需要不断维护,因为当这些对象的构造函数的签名被修改时,它们需要更新:

```
Foo foo = new Foo(new A(new B(new C()), new D()), new E(), ..., new Z());
```

使用测试替身可能很诱人,因为构建一个测试替身并不困难。

例如,使用 Mockito 模拟框架时,构建测试替身只需以下步骤:

```
@Mock Foo mockFoo;
```

尽管创建这个测试替身要简单得多,但使用真实实现有显著的好处,正如本节前面所讨论的那样。以这种方式过度使用测试替身也常常会带来重大的弊端,我们将在本章后面讨论。因此,在考虑是使用真实实现还是测试替身时,需要做出权衡。

理想的解决方案是使用与生产代码相同的对象构造代码,例如工厂方法或自动依赖项注入,而不是在测试中手动构造对象。为了支持测试用例,对象构造代码需要足够灵活,以便能够使用测试替身,而不是对将用于生产的实现进行硬编码。

## 假装

如果在测试中使用真实实现不可行,最好的选择通常是使用假实现来代替。假实现优于其他测试替身技术,因为它的行为与真实实现类似:被测系统甚至无法分辨它是在与真实实现交互还是与假实现交互。

**例 13-11**说明了一个伪文件系统。

### 例子 13-11. 一个伪造的文件系统

```
// 这个伪造的实现了 FileSystem 接口。这个接口也被 // 真正的实现使用。 public class FakeFileSystem implements
FileSystem {

    // 存储文件名到文件内容的映射。文件存储在内存中而不是磁盘上,因为测试不需要执行磁盘 I/O。 private Map<String,
    String> files = new HashMap<>(); @Override public void writeFile(String fileName, String content) {

        // 将文件名和内容添加到地图中。files.add (fileName, content);

    }

    @Override
    public String readFile(String fileName) { String content =
        files.get(fileName); // 如果找不到文件,真正的实现将抛出此异常,
        因此伪造的也必须抛出该异常。 if (contents == null) { throw new FileNotFoundException(fileName); }
        return content;
    }

}
```

## 假货为何如此重要？

伪造的程序可以成为一种强大的测试工具：它们执行速度很快，并且允许您有效地测试代码，而没有使用真实实现的缺点。

单个伪造程序就能彻底改善 API 的测试体验。如果将这一比例扩大到针对各种 API 的大量伪造程序，伪造程序可以极大地提高整个软件组织的工程速度。

在另一个极端，在一个很少出现伪造的软件组织中，速度会更慢，因为工程师最终可能会难以使用真正的实现，从而导致测试缓慢且不稳定。或者工程师可能会求助于其他测试替身技术，如存根测试或交互测试，正如我们将在本章后面讨论的那样，这可能会导致测试不明确、脆弱且效率低下。

## 什么时候应该写假货？

伪造需要付出更多努力和更多领域经验才能创建，因为它的行为需要与真实实现类似。伪造还需要维护：每当真实实现的行为发生变化时，伪造也必须更新以匹配此行为。因此，拥有真实实现的团队应该编写和维护一个伪造。

如果一个团队正在考虑编写一个假程序，那么需要权衡一下，使用假程序带来的生产效率提升是否超过编写和维护它的成本。如果只有少数用户，这可能不值得他们花时间，而如果有数百名用户，它可以带来明显的生产效率提升。

为了减少需要维护的伪代码数量，通常应仅在无法用于测试的代码根部创建伪代码。例如，如果数据库无法用于测试，则应为数据库 API 本身创建伪代码，而不是为调用数据库 API 的每个类创建伪代码。

如果需要在不同的编程语言中重复实现伪造服务，维护伪造服务可能会很麻烦，例如对于具有允许从不同语言调用该服务的客户端库的服务。这种情况的一个解决方案是创建一个单一的伪造服务实现，并让测试配置客户端库以向该伪造服务发送请求。与将伪造服务完全写入内存相比，这种方法更为繁重，因为它需要测试跨进程通信。但是，只要测试仍能快速执行，这可以是一个合理的权衡。

### 伪造品的逼真度伪造品制

作中最重要的概念或许就是逼真度;换句话说,就是伪造品的行为与真实实现的行为有多接近。如果伪造品的行为与真实实现的行为不匹配,则使用该伪造品进行的测试将毫无用处。使用伪造品时测试可能会通过,但相同的代码路径在真实实现中可能无法正常工作。

完美保真度并不总是可行的。毕竟,伪造是必要的,因为真实的实现在某些方面并不合适。例如,伪造的数据库在硬盘存储方面通常与真实数据库不保真,因为伪造的数据库会将所有内容存储在内存中。

但最重要的是,伪造程序应忠实行真实实现的 API 契约。对于 API 的任何给定输入,伪造程序应返回相同的输出并执行与其对应的真实实现相同的状态更改。例如,对于 database.save(itemId) 的真实实现,如果项目在其 ID 尚不存在时成功保存,但当 ID 已存在时产生错误,则伪造程序必须符合相同的行为。

一种思考方式是,伪造品必须与真实实现完全一致,但这只是从测试的角度来看的。例如,哈希 API 的伪造品不需要保证给定输入的哈希值与真实实现生成的哈希值完全相同 - 测试可能不关心特定的哈希值,只关心给定输入的哈希值是唯一的。如果哈希 API 的契约不能保证将返回哪些特定的哈希值,那么即使伪造品与真实实现不完全一致,它仍然符合契约。

其他完美保真度通常对伪造品无用的例子包括延迟和资源消耗。但是,如果您需要明确测试这些约束 (例如,验证函数调用延迟的性能测试),则不能使用伪造品,因此您需要诉诸其他机制,例如使用真实实现而不是伪造品。

伪代码可能不需要具有其对应真实实现的 100% 的功能,特别是如果大多数测试不需要这种行为 (例如,用于罕见边缘情况的错误处理代码)。在这种情况下,最好让伪代码快速失败;例如,如果执行了不受支持的代码路径,则引发错误。这种失败会告诉工程师,在这种情况下伪代码是不合适的。

## 假货应该经过检验

伪造程序必须有自己的测试,以确保其符合其对应的真实实现的 API。未经测试的伪造程序最初可能会提供逼真的行为,但如果 没有 测试,随着真实实现的发展,这种行为可能会随着时间的推移而发生变化。

为伪造品编写测试的一种方法是针对 API 的公共接口编写测试,然后针对真实实现和伪造品运行这些测试(这些测试称为[契约测试](#))。针对真实实现运行的测试可能会更慢,但它们的缺点被最小化,因为它们只需要由假的所有者运行。

## 如果没有假货该怎么办

如果没有假货,请先让 API 所有者创建一个。所有者可能不熟悉假货的概念,或者他们可能没有意识到假货为 API 用户带来的好处。

如果 API 的所有者不愿意或无法创建伪造的 API,您可以自己编写。一种方法是将对 API 的所有调用包装在一个类中,然后创建不与 API 通信的类的伪造版本。这样做也比为整个 API 创建伪造的 API 简单得多,因为您通常只需要使用 API 行为的子集。在 Google,一些团队甚至将他们的伪造贡献给 API 的所有者,这使得其他团队可以从伪造中受益。

最后,你可以决定使用真实的实现(并处理本章前面提到的真实实现的权衡),或者采用其他测试双重技术(并处理我们将在本章后面提到的权衡)。

在某些情况下,你可以将伪造视为一种优化:如果使用真实实现的测试速度太慢,你可以创建一个伪造程序来加快测试速度。但如果伪造程序带来的加速效果无法抵消创建和维护伪造程序所需的工作量,那么最好还是坚持使用真实实现。

## 存根

如本章前面所述,存根是一种测试方法,用于对本身没有行为的函数进行硬编码。它通常是在测试中替换实际实现的一种快速简便的方法。例如,[示例 13-12](#)中的代码使用存根来模拟信用卡服务器的响应。

### 例 13-12. 使用存根来模拟响应

```
@Test public void getTransactionCount() {
    transactionCounter = new TransactionCounter(mockCreditCardServer); // 使用存根返回三个
    交易。when (mockCreditCardServer.getTransactions()).thenReturn(
        newList(TRANSACTION_1, TRANSACTION_2, TRANSACTION_3) );断言
    (transactionCounter.getTransactionCount () ) .isEqualTo (3) ;
}
```

过度使用存根的危险因为存根在测试中应用起来非常容

易,所以当使用实际实现并不简单时,人们很容易就会使用这种技术。然而,过度使用存根可能会导致需要维护这些测试的工程师的生产力大幅下降。

### 测试变得不明确

存根涉及编写额外的代码来定义被存根函数的行为。这些额外的代码会偏离测试的初衷,如果您不熟悉被测系统的实现,这些代码可能很难理解。

一个关键的迹象表明,存根不适合测试的是,如果您发现自己在心理上逐步完成被测系统,以了解测试中某些功能被存根的原因。

### 测试变得脆弱

存根会将代码的实现细节泄露到测试中。当生产代码中的实现细节发生变化时,您需要更新测试以反映这些变化。理想情况下,良好的测试应该仅在 API 面向用户的行为发生变化时才需要更改;它应该不受 API 实现更改的影响。

### 测试效果变差

使用存根时,无法确保被存根的函数的行为与真实实现类似,例如以下代码片段中所示的语句,该语句对add()方法的部分契约进行了硬编码 (“如果传入 1 和 2,则将返回 3” ) :

当 (stubCalculator.add (1,2) )时。然后返回 (3) ;

如果被测系统依赖于实际实现的契约,那么存根是一个糟糕的选择,因为您将被迫复制契约的细节,并且无法保证契约的正确性 (即,存根函数与实际实现具有一致性) 。

此外,使用存根无法存储状态,这会使测试代码的某些方面变得困难。例如,如果您在真实实现或虚假实现上调用`database.save(item)`,您可能能够通过调用`database.get(item.id())`来检索该项目,因为这两个调用都在访问内部状态,但使用存根,没有办法做到这一点。

过度使用存根的示例  
[示例 13-13](#)说明了一个过度使用存根的测试。

### 例子 13-13. 过度使用存根

```
@Test public void creditCardIsCharged() {
    // 传入由模拟框架创建的测试替身。 paymentProcessor = new PaymentProcessor(mockCreditCardServer,
    mockTransactionProcessor); //
    为这些测试替身设置存根。 when(mockCreditCardServer.isServerAvailable()).thenReturn(true);

    when(mockTransactionProcessor.beginTransaction()).thenReturn(transaction);
    when(mockCreditCardServer.initTransaction(transaction)).thenReturn(true);
    when(mockCreditCardServer.pay(transaction, creditCard, 500))

    然后返回 (false) ;
    when(mockTransactionProcessor.endTransaction()).thenReturn(true); // 调用测试系统。
    paymentProcessor.processPayment
    (creditCard, Money.dollars(500)); // 无法判断 pay() 方法是否真正执行了交易,因此测试唯一能做的就是
    验证 pay() 方法是否被调用.verify (mockCreditCardServer).pay(transaction, creditCard, 500);

}
```

[示例 13-14](#)重写了相同的测试,但避免使用存根。请注意,测试更短,并且实现细节(例如如何使用交易处理器)未在测试中公开。不需要特殊设置,因为信用卡服务器知道如何操作。

### 示例 13-14. 重构测试以避免存根

```
@Test public void creditCardIsCharged() { paymentProcessor
    = new
        PaymentProcessor(creditCardServer, transactionProcessor);
    // 调用测试系统.paymentProcessor.processPayment
    (creditCard, Money.dollars(500)); // 查询信用卡服务器状态以查看付款是否通过.assertThat
    (creditCardServer.getMostRecentCharge(creditCard)).isEqualTo(500);

}
```

我们显然不希望这样的测试与外部信用卡服务器通信,因此假信用卡服务器更合适。如果没有假的,另一个选择是使用与密封信用卡服务器通信的真实实现,尽管这会增加测试的执行时间。(我们将在下一章中探讨密封服务器。)

## 什么时候适合使用存根?

存根并非是真实实现的全面替代,而是在需要函数返回特定值以使被测系统进入特定状态时才适用,例如[示例 13-12](#)要求被测系统返回非空事务列表。由于函数的行为是在测试中内联定义的,因此存根可以模拟各种返回值或错误,而这些返回值或错误可能无法从真实实现或伪造实现中触发。

为了确保其目的明确,每个存根函数都应与测试的断言有直接关系。因此,测试通常应该存根少量函数,因为存根过多函数会导致测试不太明确。

需要对许多功能进行存根的测试可能表明存根被过度使用,或者被测系统过于复杂而应进行重构。

请注意,即使存根是合适的,真实的实现或伪造的实现仍然是首选,因为它们不会暴露实现细节,并且与存根相比,它们可以为您提供有关代码正确性的更多保证。但存根可以是一种合理的技术,只要其使用受到限制,以使测试不会变得过于复杂。

## 交互测试

正如本章前面所讨论的,交互测试是一种无需实际调用函数实现即可验证函数如何被调用的方法。

模拟框架使交互测试变得简单。但是,为了保持测试的实用性、可读性和对变化的适应性,只在必要时执行交互测试非常重要。

优先进行状态测试而不是交互测试与交互测试相反,优先通过[状态测试来测试代码](#)。

通过状态测试,您可以调用被测系统并验证是否返回了正确的值,或者被测系统中的某些其他状态是否正确更改。[示例 13-15](#)给出了状态测试的示例。

### 例子 13-15. 状态测试

```
@Test public void sortNumbers() {
    NumberSorter numberSorter = new NumberSorter(quicksort, bubbleSort); // 调用被测系统。

    List sortedList = numberSorter.sortNumbers(newList(3, 1, 2)); // 验证返回的列表是否
    已排序。使用哪种排序算法并不重要，只要返回正确的结果即可。assertThat(sortedList).isEqualTo(newList(1,
    2, 3));

}
```

**例 13-16**说明了类似的测试场景,但使用了交互测试。

请注意,此测试无法确定数字是否真正排序,因为测试替身不知道如何对数字进行排序 - 它只能告诉你,被测系统已尝试对数字进行排序。

### 示例 13-16. 交互测试

```
@Test public void sortNumbers_quicksortIsUsed() {
    // 传入由模拟框架创建的测试替身。
    NumberSorter numberSorter =
        new NumberSorter(mockQuicksort, mockBubbleSort);

    // 调用被测系统.numberSorter.sortNumbers
    (newList(3, 1, 2));

    // 验证 numberSorter.sortNumbers() 是否使用了快速排序。如果从未调用 mockQuicksort.sort() (例
    如,如果使用了 mockBubbleSort)或者使用错误的参数调用,则测试将失败。verify
    (mockQuicksort).sort(newList(3, 1, 2));

}
```

在谷歌,我们发现强调状态测试更具可扩展性;它降低了测试脆弱性,使得随着时间的推移更改和维护代码变得更容易。

交互测试的主要问题是它无法告诉您被测系统是否正常运行;它只能验证某些函数是否按预期调用。它要求您对代码的行为做出假设;例如,“如果调用database.save(item),我们假设该项目将保存到数据库中。”状态测试是首选,因为它实际上验证了这一假设 (例如通过将项目保存到数据库,然后查询数据库以验证该项目是否存在)。

交互测试的另一个缺点是它利用了被测系统的实现细节 为了验证某个函数是否被调用,您需要向测试暴露被测系统调用该函数的事实。与存根类似,这些额外的代码会使测试变得脆弱,因为它会将生产代码的实现细节泄露到测试中。谷歌的一些人开玩笑地提到过度使用交互的测试

作为**变化检测测试**进行测试因为它们无法响应生产代码的任何改变,即使被测系统的行为保持不变。

## 何时适合进行交互测试?

在某些情况下需要进行交互测试:

- 您无法执行状态测试,因为您无法使用真实实现或虚假实现(例如,如果真实实现太慢且不存在虚假实现)。作为后备,您可以执行交互测试来验证是否调用了某些函数。虽然不是最理想的,但这确实提供了一些基本的信心,即被测系统是否按预期运行。

- 函数调用次数或调用顺序的差异可能会导致不希望的

行为。交互测试很有用,因为使用状态测试来验证此行为可能很困难。例如,如果您希望缓存功能减少对数据库的调用次数,则可以验证数据库对象的访问次数不会超过预期。使用 Mockito,代码可能看起来类似于以下内容:

```
验证(databaseReader,atMostOnce()) .selectRecords();
```

交互测试并不能完全替代状态测试。如果您无法在单元测试中执行状态测试,强烈建议您使用执行状态测试的最大范围的测试来补充您的测试套件。例如,如果您有一个通过交互测试验证数据库使用情况的单元测试,请考虑添加一个可以针对真实数据库执行状态测试的集成测试。更大范围的测试是降低风险的重要策略,我们将在下一章中讨论它。

交互测试的最佳实践在执行交互测试时,遵循这些实践可以减少上述缺点的一些影响。

### 只对状态改变函数进行交互测试

当被测系统调用依赖项上的函数时,该调用属于以下两类之一:

状态改变函数对被

测系统外部的世界有副作用。

示例: `sendEmail()`、`saveRecord()`、`logAccess()`。

### 非状态改变函数没有副作用

用；它们返回有关被测系统外部世界的信息，并且不会修改任何内容。示例：getUser()、findResults()、readFile()。

一般来说，您应该只对状态改变的函数执行交互测试。对非状态改变的函数执行交互测试通常是否余的，因为被测系统将使用函数的返回值来执行您可以断言的其他工作。交互本身并不是正确性的重要细节，因为它没有副作用。

对非状态改变函数执行交互测试会使你的测试变得脆弱，因为你需要在交互模式发生变化时更新测试。

由于附加断言使得确定哪些断言对于确保代码的正确性至关重要变得更加困难，因此这也使测试的可读性降低。相比之下，状态改变交互表示您的代码正在执行一些有用的操作来改变其他地方的状态。

**例 13-17**演示了状态改变和非状态改变函数的交互测试。

### 示例 13-17 状态改变和非状态改变交互

```
@Test public void grantUserPermission() {
    // 用户授权者 = new UserAuthorizer(mockUserService, mockPermissionDatabase);
    // 当 (mockPermissionService.getPermission(FAKE_USER)) 时。然后返回 (空) ;
    // 调用被测系统。userAuthorizer.grantPermission
    // (USER_ACCESS);

    // addPermission() 是状态改变的，因此执行 // 交互测试来验证它是否被调用是合理的。verify
    // (mockPermissionDatabase).addPermission(FAKE_USER, USER_ACCESS);

    // getPermission() 不会改变状态，因此不需要此行代码。一个线索表明可能不需要交互测试： //
    // getPermission() 已在此测试中被存根。 verify(mockPermissionDatabase).getPermission(FAKE_USER);
}
```

### 避免过度规范在第 12

**章中**，我们讨论了为什么测试行为比测试方法更有用。这意味着测试方法应该专注于验证方法或类的一个行为，而不是试图在单个测试中验证多个行为。

在执行交互测试时,我们应尽量应用相同的原则,避免过度指定要验证的函数和参数。这样可以使测试更清晰、更简洁。它还可以使测试对超出每个测试范围的行为所做的更改具有弹性,因此如果对函数调用方式进行更改,则失败的测试会更少。

**例 13-18**说明了过度指定的交互测试。测试的目的是验证用户的名字是否包含在问候提示中,但如果与相关的功能被改变,测试就会失败。

### 示例 13-18. 过度指定的交互测试

```
@Test public void displayGreeting_renderUserName() {
    when(mockUserService.getUserName()).thenReturn( Fake User );
    userGreeter.displayGreeting(); // 调用被测系统。

    // 如果 setText() 的任何参数发生变化,测试将失败.verify(userPrompt).setText( Fake User , Good
    morning! , Version 2.1 );

    // 如果不调用 setIcon(),测试将失败,即使这个 // 行为对于测试来说是偶然的,因为它与 // 验证用户名
    无关.verify(userPrompt).setIcon(IMAGE_SUNSHINE);

}
```

**示例 13-19**说明了交互测试在指定相关参数和函数时更加谨慎。被测试的行为被分成单独的测试,每个测试验证确保其测试行为正确所需的最少内容。

### 示例 13-19. 明确指定的交互测试

```
@Test public void displayGreeting_renderUserName() {
    when(mockUserService.getUserName()).thenReturn( Fake User );
    userGreeter.displayGreeting(); // 调用被测系统.verify(userPrompt).setText(eq( Fake
    User ), any(), any());
}

@Test public void displayGreeting_timeIsMorning_useMorningSettings() {
    setTimeOfDay(TIME_MORNING);
    userGreeter.displayGreeting(); // 调用被测系统.verify(userPrompt).setText(any(),
    eq( 早上好! , any())); verify(userPrompt).setIcon(IMAGE_SUNSHINE);

}
```

## 结论

我们了解到,测试替身对于工程速度至关重要,因为它们可以帮助全面测试您的代码并确保您的测试快速运行。另一方面,误用它们会严重影响生产力,因为它们会导致测试不明确、脆弱且效率低下。这就是为什么工程师了解如何有效应用测试替身的最佳实践很重要。

关于是否使用真实实现或测试替身,或者使用哪种测试替身技术,通常没有确切的答案。工程师在决定适合其用例的方法时可能需要做出一些权衡。

尽管测试替身非常适合处理难以在测试中使用的依赖项,但如果您想要最大限度地提高对代码的信心,有时您仍然需要在测试中使用这些依赖项。下一章将介绍更大范围的测试,无论这些依赖项是否适合单元测试,例如,即使它们很慢或不确定,它们都会被使用。

## TL;DR

- 应优先考虑真实实现,而不是测试替身。 · 如果无法在测试中使用真实实现,则假冒通常是理想的解决方案。
- 过度使用存根会导致测试不清晰且脆弱。
- 应尽可能避免交互测试:它会导致测试很脆弱,因为它会暴露被测系统的实现细节。

## 第十四章

# 更大规模的测试

作者:约瑟夫·格雷夫斯  
编辑:Tom Mansreck

在前面的章节中,我们讲述了 Google 是如何建立测试文化的,以及小型单元测试如何成为开发人员工作流程的基本组成部分。但是其他类型的测试呢?事实证明,Google 确实使用了许多大型测试,这些测试构成了健康软件工程所必需的风险缓解策略的重要组成部分。但这些测试带来了额外的挑战,以确保它们是宝贵的资产而不是资源消耗。在本章中,我们将讨论“大型测试”的含义、执行这些测试的时间以及保持其有效性的最佳实践。

## 什么是更大规模的测试?

如前所述,Google 对测试规模有具体概念。小型测试仅限于一个线程、一个进程、一台机器。大型测试没有相同的限制。但 Google 也有测试范围的概念。单元测试的范围必然小于集成测试。而最大范围的测试(有时称为端到端或系统测试)通常涉及几个实际依赖项和较少的测试替身。

大型测试具有许多小型测试所不具备的特征。它们不受相同约束的约束;因此,它们可以表现出以下特征:

- 它们可能很慢。我们的大型测试默认超时时间为 15 分钟或 1 小时,但我们也有持续数小时甚至数天的测试。
- 它们可能是非密封的。大型测试可能与其他测试共享资源,并且交通。

- 它们可能是非确定性的。如果大型测试是非封闭的，则几乎不可能保证确定性：其他测试或用户状态可能会干扰它。

那么为什么要进行更大规模的测试呢？回想一下你的编码过程。你如何确认你编写的程序确实可以运行？你可能在编写和运行单元测试，但你是否发现自己在运行实际的二进制文件并亲自尝试？当你与他人分享此代码时，他们如何测试它？通过运行你的单元测试，还是自己尝试？

另外，您如何知道您的代码在升级期间继续工作？假设您有一个使用 Google Maps API 的网站，并且有一个新的 API 版本。

您的单元测试可能不会帮助您了解是否存在兼容性问题。您可能会运行它并尝试一下，看看是否有任何问题。

单元测试可以让您对单个函数、对象和模块充满信心，但大型测试可以让您更加确信整个系统是否按预期运行。

并且实际的自动化测试可以以手动测试无法实现的方式进行扩展。

#### 保真度更大

规模测试存在的主要原因是为了解决保真度问题。保真度是测试反映被测系统 (SUT) 真实行为的属性。

设想保真度的一种方式是从环境的角度。如图14-1所示，单元测试将测试和一小部分代码捆绑在一起作为可运行单元，这确保代码经过测试，但与生产代码的运行方式有很大不同。

生产本身自然是测试中保真度最高的环境。还有一系列临时选项。大型测试的关键是找到合适的方案，因为保真度的提高也伴随着成本的增加和（在生产的情况下）失败风险的增加。

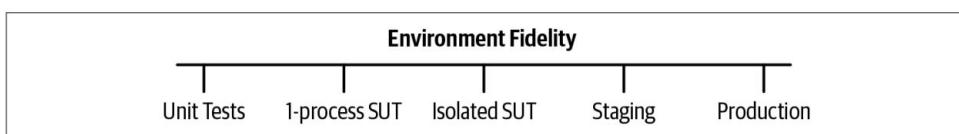


图 14-1. 保真度提升的规模

测试还可以通过测试内容对现实的忠实程度来衡量。如果测试数据本身看起来不切实际，许多手工制作的大型测试都会被工程师拒绝。从生产中复制的测试数据更忠实于现实（以这种方式捕获），但一个巨大的挑战是如何在发布新代码之前创建真实的测试流量。这在人工智能 (AI) 中尤其成问题，因为“种子”数据经常受到内在偏差的影响。而且，由于单元测试的大多数数据都是手工制作的，因此它涵盖的案例范围很窄，并且倾向于符合

作者的偏见。数据遗漏的场景代表了测试中的保真度差距。

### 单元测试中的常见缺陷当较小的测试

失败时,可能还需要进行较大的测试。以下小节介绍了单元测试无法提供良好风险缓解覆盖范围的一些特定领域。

#### 不忠的双打

单个单元测试通常涵盖一个类或模块。测试替身 (如第 13 章所述) 通常用于消除重量级或难以测试的依赖关系。但是当这些依赖关系被替换时,替换和替身可能不一致。

Google 的几乎所有单元测试都是由编写被测单元的同一位工程师编写的。当这些单元测试需要替身并且使用的替身是模拟时,编写单元测试的工程师会定义模拟及其预期行为。但该工程师通常不会编写被模拟的内容,因此可能会对其实际行为产生误解。被测单元与给定对等体之间的关系是一种行为契约,如果工程师误解了实际行为,则对契约的理解无效。

此外,模拟会变得陈旧。如果这个基于模拟的单元测试对于实际实现的作者不可见,并且实际实现发生变化,则没有信号表明测试 (以及被测试的代码) 应该更新以跟上变化。

请注意,如第 13 章所述,如果团队为自己的服务提供虚假信息,这种担忧就会大大缓解。

#### 配置问题单元测试涵

盖给定二进制文件中的代码。但该二进制文件在执行方式方面通常不是完全自给自足的。通常二进制文件具有某种部署配置或启动脚本。此外,实际的最终用户服务生产实例有自己的配置文件或配置数据库。

如果这些文件存在问题,或者这些存储定义的状态与相关二进制文件之间存在兼容性问题,则可能会导致严重的用户问题。单靠单元测试无法验证这种兼容性。<sup>1</sup>顺便说一句,这是确保您的配置和代码都处于版本控制中的一个很好的理由,因为这样,

---

<sup>1</sup>有关更多信息,请参阅第 483 页的“持续交付”和第 25 章。

配置更改可以被识别为错误的来源,而不是引入随机的外部缺陷,并且可以纳入大型测试中。

在 Google,配置更改是导致严重中断的首要原因。

这是我们表现不佳的一个领域,并导致了一些最令人尴尬的错误。例如,2013 年,由于未经测试的网络配置推送不当,谷歌发生了全球性中断。配置往往是用配置语言编写的,而不是生产代码语言。它们的生产推出周期通常比二进制文件更快,而且测试起来也更困难。所有这些都会导致更高的故障可能性。但至少在这种情况下(以及其他情况下),配置是版本控制的,我们可以快速识别罪魁祸首并缓解问题。

#### 负载下出现的问题

在 Google,单元测试力求小而快,因为它们需要适应我们的标准测试执行基础架构,并且作为顺畅的开发人员工作流程的一部分多次运行。但性能、负载和压力测试通常需要向给定的二进制文件发送大量流量。在典型的单元测试模型中,这些流量很难测试。而且我们的流量很大,通常每秒有数千或数百万个查询(就广告、**实时竞价而言**)!

#### 未预料到的行为、输入和副作用单元测试受限于编

写它们的工程师的想象力。也就是说,它们只能测试预期的行为和输入。但是,用户在产品中发现的问题大多是未预料到的(否则它们不太可能作为问题传递给最终用户)。这一事实表明,需要不同的测试技术来测试未预料到的行为。

**海伦法则**这里有一个重要的考虑因素:即使我们可以 100% 地测试是否符合严格的指定契约,有效用户契约也适用于所有可见行为,而不仅仅是规定的契约。单元测试不太可能单独测试公共 API 中未指定的所有可见行为。

#### 突发行为和“真空效应”

单元测试仅限于其覆盖的范围(尤其是在广泛使用测试替身的情况下),因此如果行为在此范围之外发生变化,则无法检测到。而且由于单元测试旨在快速可靠,因此它们会刻意消除真实依赖关系、网络和数据的混乱。单元测试就像理论物理学中的一个问题:隐藏在真空中,巧妙地隐藏在现实世界的混乱中,这对速度和可靠性很有好处,但却会遗漏某些缺陷类别。

## 为什么不进行更大规模的测试？

在前面的章节中,我们讨论了许多开发人员友好型测试的属性。

具体来说,需要如下:

可靠它—

定不能不稳定并且必须提供有用的通过/失败信号。

快速

它需要足够快以免打断开发人员的工作流程。

可扩展

Google 需要能够有效地运行提交前和提交后所有这些有用的受影响测试。

好的单元测试具备所有这些特性。大型测试通常会违反所有这些约束。例如,大型测试通常更不稳定,因为它们比小型单元测试使用更多的基础设施。它们通常也更慢,无论是设置还是运行。而且它们很难扩展,因为需要资源和时间,但通常还因为它们不是孤立的 这些测试可能会相互冲突。

此外,更大的测试还带来了另外两个挑战。首先,所有权问题。单元测试显然归拥有该单元的工程师 (和团队) 所有。更大的测试跨越多个单元,因此可以跨越多个所有者。这带来了长期所有权挑战:谁负责维护测试,谁负责在测试失败时诊断问题?如果没有明确的所有权,测试就会失败。

更大规模测试面临的第二个挑战是标准化 (或缺乏标准化)。

与单元测试不同,大型测试在编写、运行和调试的基础设施和流程方面缺乏标准化。大型测试的方法是系统架构决策的产物,因此引入了所需测试类型的差异。例如,我们在 Google Ads 中构建和运行 AB diff 回归测试的方式与在搜索后端中构建和运行此类测试的方式完全不同,而后者又不同于 Drive。它们使用不同的平台、不同的语言、不同的基础设施、不同的库和竞争的测试框架。

标准化的缺失会产生重大影响。由于大型测试有多种运行方式,因此在大规模变更期间通常会跳过这些测试。(参见第 22 章。)基础设施没有标准的方法来运行这些测试,并且要求执行 LSC 的人员了解每个团队的本地测试细节是不可扩展的。由于大型测试在各个团队的实施上有所不同,因此实际测试这些团队之间集成的测试需要统一不兼容的基础设施。而且由于标准化的缺失,我们无法教授

对 Nooglers (新 Google 员工)或更有经验的工程师采取单一方法,这不仅使这种情况持续存在,而且还导致对此类测试动机缺乏理解。

## 谷歌的大型测试

当我们之前讨论 Google 的测试历史时 (参见第 11 章),我们提到了 Google Web Server (GWS) 在 2003 年强制要求进行自动化测试,以及这是如何成为一个分水岭时刻。然而,我们实际上在此之前就已经使用了自动化测试,但常见的做法是使用自动化的大规模测试。例如,AdWords 早在 2001 年就创建了一个端到端测试来验证产品场景。同样,在 2002 年,搜索为其索引代码编写了一个类似的“回归测试”,而 AdSense (当时甚至还没有公开发布)在 AdWords 测试的基础上创建了自己的变体。

2002 年左右还存在其他“更大”的测试模式。Google 搜索前端严重依赖手动 QA 端到端测试场景的手动版本。Gmail 有自己的“本地演示”环境版本——一个脚本,用于在本地启动端到端 Gmail 环境,其中包含一些生成的测试用户和邮件数据,用于本地手动测试。

当 C/J Build (我们的第一个持续构建框架)发布时,它并没有区分单元测试和其他测试,但有两个关键的发展导致了分裂。首先,Google 专注于单元测试,因为我们希望鼓励测试金字塔,并确保绝大多数书面测试都是单元测试。其次,当 TAP 取代 C/J Build 成为我们正式的持续构建系统时,它只能对符合 TAP 资格要求的测试这样做:可在一次更改时构建的密封测试,可以在最长时间限制内在我们的构建/测试集群上运行。虽然大多数单元测试都满足此要求,但大型测试大多不满足。

然而,这并没有阻止对其他类型测试的需求,它们继续填补覆盖范围的空白。C/J Build 甚至坚持了多年专门处理这些类型的测试,直到新系统取代了它。

更大规模的测试和时间在本书中,

我们一直在研究时间对软件工程的影响,因为谷歌已经开发了运行了 20 多年的软件。时间维度如何影响更大规模的测试?我们知道,代码的预期寿命越长,某些活动就越有意义,各种形式的测试在各个层面都是有意义的活动,但适当的测试类型会随着代码的预期寿命而变化。

正如我们之前指出的,单元测试对于预期寿命从几小时开始的软件来说才有意义。在几分钟级别 (对于小脚本),手动

测试最为常见,SUT 通常在本地运行,但本地演示可能是生产,尤其是一次性脚本、演示或实验。在较长的生命周期中,手动测试仍然存在,但 SUT 通常会有所不同,因为生产实例通常是在云托管的,而不是本地托管的。

其余的大型测试都为更长寿命的软件提供了价值,但主要关注点变成了随着时间的推移这些测试的可维护性。

顺便说一句,这种时间影响可能是开发“冰淇淋甜筒”测试反模式的原因之一,如[第 11 章](#)所述,并在图 14-2 中再次显示。

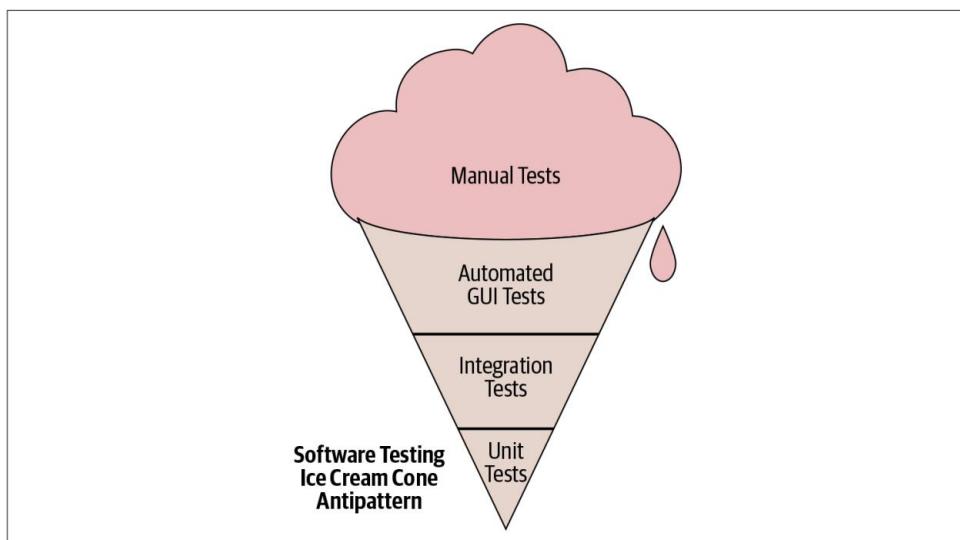


图 14-2。冰淇淋甜筒测试反模式

当开发从手动测试开始(工程师认为代码只需要运行几分钟)时,这些手动测试就会累积起来并占据初始整体测试组合的主导地位。例如,破解脚本或应用程序并通过运行它进行测试,然后继续向其中添加功能,但继续通过手动运行进行测试,这是很典型的做法。这个原型最终变得可用并与其他人共享,但实际上不存在自动化测试。

更糟糕的是,如果代码难以进行单元测试(因为它最初的实现方式),那么唯一可以编写的自动化测试是端到端的测试,而我们在几天之内就无意中创建了“遗留代码”。

对于长期健康而言,在开发的最初几天内通过构建单元测试来迈向测试金字塔至关重要,然后在那之后通过引入自动化集成测试并摆脱手动端到端测试来达到顶峰。我们成功地将单元测试作为提交的要求,但是

弥补单元测试和手动测试之间的差距对于长期健康是必要的。

在 Google 规模上进行更大规模的测试看起来,在

更大规模的软件中,更大规模的测试应该更必要、更合适,但尽管如此,编写、运行、维护和调试这些测试的复杂性也会随着规模的增长而增加,甚至比单元测试还要复杂。

在由微服务或独立服务器组成的系统中,互连模式看起来像一个图:让该图中的节点数为 N。每次向该图添加新节点时,都会对通过该图的不同执行路径的数量产生乘法效应。

**图 14-3**描绘了一个想象中的 SUT:该系统由一个包含用户的社交网络、一个社交图谱、一个帖子流和一些混杂的广告组成。广告由广告商创建并在社交交流的背景下投放。仅此 SUT 就包含两组用户、两个 UI、三个数据库、一个索引管道和六台服务器。

图中列举了 14 条边。测试所有端到端可能性已经很困难了。想象一下,如果我们在这个组合中添加更多服务、管道和数据库:照片和图像、机器学习照片分析等等?

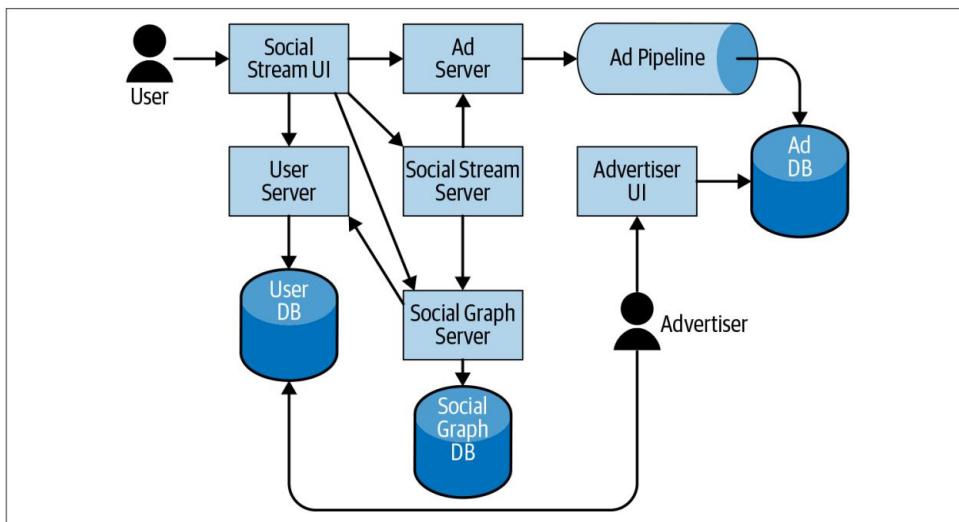


图 14-3 一个相当小的 SUT 示例:带有广告的社交网络

端到端测试不同场景的速度可能会呈指数级或组合式增长,具体取决于被测系统的结构,而且这种增长无法扩展。因此,随着系统的增长,我们必须找到替代的更大规模测试策略,以保持可管理性。

然而,为了达到这一规模,必须做出一些决策,因此此类测试的价值也随之提升。这是保真度的影响:随着软件层数的增加(N层),如果服务替身的保真度较低(1- $\epsilon$ ) ,那么将它们组合在一起时出现bug的几率将以N为指数级增长。再次查看此示例SUT,如果我们用替身替换用户服务器和广告服务器,并且这些替身的保真度较低(例如,准确率为10%) ,则出现bug的可能性为99% ( $1 - (0.1 \cdot 0.1)$ ) 。而且这还只是两个低保真度替身的情况。

因此,以在这种规模下运行良好但保持合理高保真度的方式实施更大规模的测试变得至关重要。

### 提示：“尽可能小的测试”

即使对于集成测试,规模越小越好 少量大型测试比庞大的测试要好。而且,由于测试的范围通常与SUT的范围相关,因此找到缩小SUT的方法有助于缩小测试规模。

当用户旅程可能需要来自许多内部系统的贡献时,实现此测试比率的一种方法是“链接”测试,如图14-4所示,不是具体执行测试,而是创建多个较小的成对集成测试来代表整体场景。这是通过确保将一个测试的输出持久化到数据存储库中,将其用作另一个测试的输入来实现的。

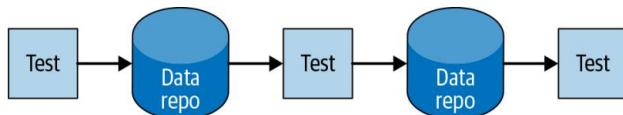


图 14-4. 链式测试

## 大型测试的结构

尽管大型测试不受小型测试约束,并且可以由任何内容组成,但大多数大型测试都表现出共同的模式。大型测试通常由以下阶段的工作流程组成:

- 获取被测系统 · 植入必要的测试数据
- 使用被测系统执行操作
- 验证行为

被测系统大型测试的一个关键组件

是上述 SUT（见图14-5）。典型的单元测试将注意力集中在一个类或模块上。此外，测试代码与被测代码在同一个进程（或 Java 虚拟机 [JVM] 在 Java 的情况下）中运行。对于较大的测试，SUT 通常非常不同；一个或多个单独的进程，测试代码通常（但并非总是）在其自己的进程中。

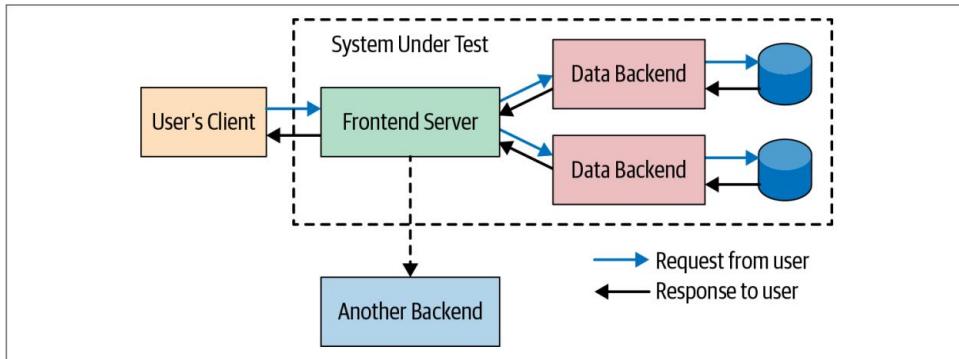


图 14-5. 被测系统 (SUT) 示例

在 Google, 我们使用多种不同形式的 SUT, SUT 的范围是大型测试本身范围的主要驱动因素之一 (SUT 越大, 测试规模越大)。每种 SUT 形式都可以根据两个主要因素来判断:

密封性 这是

SUT 与除测试之外的其他组件的使用和交互的隔离。密封性高的 SUT 受并发性和基础设施不稳定源的影响最小。

## 保真度

SUT 反映被测生产系统的准确性。保真度高的 SUT 将包含与生产版本相似的二进制文件（依赖类似的配置、使用类似的基础架构并具有类似的整体拓扑）。

这两个因素常常直接冲突。

以下是一些 SUT 的示例：

## 单进程 SUT 整个测试

系统被打包成一个二进制文件（即使在生产环境中，这些是多个单独的二进制文件）。此外，测试代码可以打包成与 SUT 相同的二进制文件。如果一切都是单线程的，这样的测试-SUT 组合可以是一个“小型”测试，但它对生产拓扑和配置的忠实度最低。

### 单机 SUT 被测系统由一个

或多个单独的二进制文件组成（与生产相同），测试是其自己的二进制文件。但所有内容都在一台机器上运行。这用于“中型”测试。理想情况下，我们在本地运行这些二进制文件时使用每个二进制文件的生产启动配置，以提高保真度。

### 多机 SUT 被测系统分布

在多台机器上（很像生产云部署）。这比单机 SUT 的保真度更高，但使用多机 SUT 会使测试规模“庞大”，而且这种组合容易受到网络和机器不稳定的影响。

### 共享环境（暂存和生产）

该测试不运行独立的 SUT，而是仅使用共享环境。

这种方式成本最低，因为这些共享环境通常已经存在，但测试可能会与其他同时使用的情况发生冲突，因此必须等待代码推送到这些环境。生产也增加了对最终用户影响的风险。

### 混合一些

SUT 代表一种混合：可能可以运行部分 SUT，但让其与共享环境交互。通常，被测试的东西是明确运行的，但其后端是共享的。对于像 Google 这样庞大的公司来说，运行 Google 所有互连服务的多个副本几乎是不可能的，因此需要进行一些混合。

### 密封 SUT 的好处

大型测试中的 SUT 可能是不可靠性和长周转时间的主要原因。例如，生产中测试使用实际的生产系统部署。如前所述，这很受欢迎，因为环境没有额外的开销成本，但生产测试在代码到达该环境之前无法运行，这意味着这些测试本身无法阻止代码发布到该环境。本质上，SUT 为时已晚。

最常见的第一个替代方案是创建一个巨大的共享暂存环境并在其中运行测试。这通常是作为某些发布推广过程的一部分完成的，但它再次将测试执行限制在仅在代码可用时。作为替代方案，一些团队将允许工程师在暂存环境中“预留”时间，并利用该时间窗口部署待处理的代码并运行测试，但这不会随着工程师数量或服务数量的增加而扩展，因为环境、用户数量以及用户冲突的可能性都会迅速增加。

生长。

下一步是支持云隔离或机器隔离的 SUT。这样的环境通过避免代码发布的冲突和预留要求来改善这种情况。

案例研究:生产测试和 Webdriver Torso 的风险我们提到过,生产测试可能会有风险。

生产测试中发生的一个有趣事件被称为 Webdriver Torso 事件。我们需要一种方法来验证 YouTube 生产中的视频渲染是否正常工作,因此创建了自动化脚本来生成测试视频、上传视频并验证上传质量。这项工作是在 Google 旗下的 YouTube 频道 Webdriver Torso 中完成的。但这个频道是公开的,大多数视频也是公开的。

随后,该频道在 [《连线》杂志的一篇文章](#) 中被公开,这导致该消息在媒体上广为流传,随后人们开始努力破解这个谜团。最后,一位博主将一切都归咎于 Google。最终,我们坦白承认了这一点,并做了一些有趣的事情,包括 Rickroll 和复活节彩蛋,所以一切都很顺利。但我们确实需要考虑最终用户发现我们在生产中包含的任何测试数据的可能性,并为此做好准备。

在问题边界处减小 SUT 的大小有一些特别痛苦的测试边界

可能值得避免。涉及前端和后端的测试变得很痛苦,因为用户界面 (UI) 测试众所周知不可靠且成本高昂:

- UI 的外观和感觉经常发生变化,这会使 UI 测试变得脆弱,但实际上并不影响底层行为。· UI 经常具有难以测试的异步行为。

尽管对服务的 UI 进行端到端测试直至其后端很有用,但这些测试对 UI 和后端的维护成本都是成倍增加的。相反,如果后端提供公共 API,则通常更容易将测试拆分为 UI/API 边界上的连接测试,并使用公共 API 来驱动端到端测试。无论 UI 是浏览器、命令行界面 (CLI)、桌面应用程序还是移动应用程序,都是如此。

另一个特殊边界是针对第三方依赖项。第三方系统可能没有用于测试的公共共享环境,在某些情况下,将流量发送给第三方会产生成本。因此,不建议让自动化测试使用真正的第三方 API,并且该依赖项是拆分测试的重要接缝。

为了解决这个尺寸问题,我们通过将其数据库替换为内存数据库并删除我们真正关心的 SUT 范围之外的一台服务器,缩小了此 SUT 的大小,如图 14-6 所示。此 SUT 更适合安装在一台机器上。

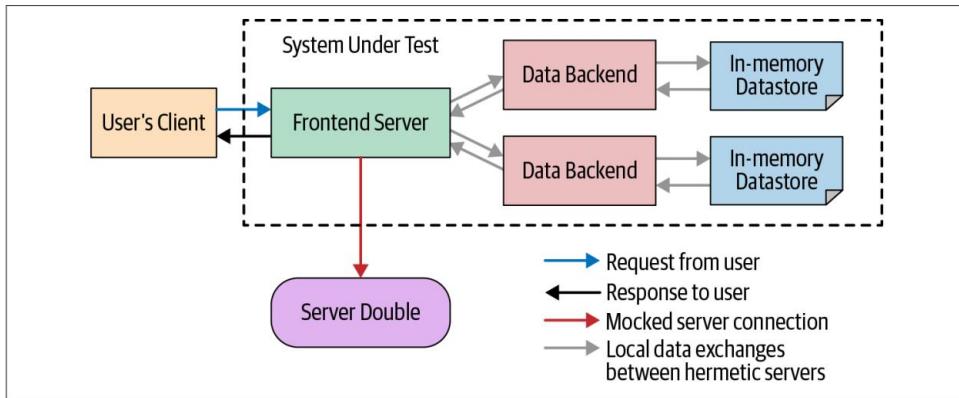


图 14-6. 缩小尺寸的 SUT

关键是要确定保真度和成本/可靠性之间的权衡,并确定合理的界限。如果我们可以运行少量二进制文件和测试,并将其全部打包到执行常规编译、链接和单元测试的同一台机器中,那么我们就可以为工程师提供最简单、最稳定的“集成”测试。

### 记录/重放代理在上

一章中,我们讨论了测试替身和可用于将被测类与其难以测试的依赖项分离的方法。我们还可以使用具有等效 API 的模拟、存根或伪造服务器或进程来复制整个服务器和进程。但是,无法保证所使用的测试替身实际上符合其所替代的真实事物的契约。

处理 SUT 的依赖但辅助服务的一种方法是使用测试替身,但如何知道替身反映了依赖项的实际行为?谷歌之外一种日益流行的方法是使用**消费者驱动契约**框架测试。这些测试为客户端和服务提供者定义了一个契约,这个契约可以驱动自动化测试。也就是说,客户端定义了一个服务的模拟,表示对于这些输入参数,我会得到一个特定的输出。然后,真正的服务在真正的测试中使用这个输入/输出对,以确保它在给定这些输入的情况下产生那个输出。两个用于消费者驱动契约测试的公共工具是**Pact Contract Testing**和**Spring Cloud Contracts**。Google 对协议缓冲区的严重依赖意味着我们不会在内部使用这些缓冲区。

在 Google,我们的做法有些不同。**我们最受欢迎的方法** (有一个公共 API)是使用较大的测试来生成较小的测试,方法是在运行较大的测试时记录到这些外部服务的流量,并在运行较小的测试时重放它。较大的测试或“**记录模式**”测试在提交后持续运行,但其主要目的是生成这些流量日志 (但必须通过才能生成日志)。较小的测试或“**重放模式**”测试用于开发和提交前测试。

记录/重放工作原理的一个有趣方面是,由于不确定性,必须通过匹配器匹配请求以确定要重放哪个响应。这使得它们与存根和模拟非常相似,因为参数匹配用于确定结果行为。

对于新的测试或者客户行为发生重大变化的测试会发生什么?

在这些情况下,请求可能不再与记录的流量文件中的内容相匹配,因此测试无法在重放模式下通过。在这种情况下,工程师必须在记录模式下运行测试以生成新的流量,因此让运行记录测试变得简单、快速和稳定非常重要。

## 测试数据

测试需要数据,大型测试需要两种不同类型的数据:

**种子数据**

被测系统中预先初始化的数据反映了测试开始时 SUT 的状态

**测试轨迹**

测试本身在执行过程中发送给被测系统的数据

由于独立且更大的 SUT 的概念,设置 SUT 状态的工作通常比单元测试中完成的设置工作复杂几个数量级。

例如:

**域数据**

某些数据库包含预填充到表中并用作环境配置的数据。如果未提供域数据,使用此类数据库的实际服务二进制文件可能会在启动时失败。

**现实基线 要使 SUT**

被视为现实的,它可能需要在启动时提供一组现实的基础数据,包括质量和数量。例如,社交网络的大型测试可能需要一个现实的社交图作为测试的基础状态:必须存在足够多具有现实个人资料的测试用户以及这些用户之间足够的互连,测试才能被接受。

### 种子 API 用于种子

子数据的 API 可能很复杂。可能可以直接写入数据存储,但这样做可能会绕过执行写入操作的实际二进制文件所执行的触发器和检查。

数据可以通过不同的方式生成,例如:

#### 手工创建的数据 与小

型测试一样,我们可以手动创建大型测试的测试数据。但是,在大型 SUT 中为多个服务设置数据可能需要更多工作,并且我们可能需要为大型测试创建大量数据。

#### 复制数据 我们

可以复制数据,通常是从生产中复制。例如,我们可以先从生产地图数据的副本开始测试地球地图,以提供基准,然后测试对它的更改。

#### 采样数据 复制数

据可能会提供过多数据,无法合理处理。采样数据可以减少数据量,从而减少测试时间并使其更容易推理。“智能采样”包括复制实现最大覆盖范围所需的最少数据的技术。

## 验证

在 SUT 运行并且向其发送流量后,我们仍然必须验证其行为。

有几种不同的方法可以做到这一点:

#### 手动 就像

在本地测试二进制文件一样,手动验证使用人工与 SUT 交互来确定其是否正常运行。此验证可以包括通过执行一致测试计划中定义的操作来测试回归,也可以是探索性的,通过不同的交互路径来识别可能的新故障。

请注意,手动回归测试不能以亚线性方式扩展:系统越大,经过的流程越多,手动测试所需的人力就越多。

#### 断言与单元测试

非常相似,这些是针对系统预期行为的明确检查。例如,对于 xyzzy 的 Google 搜索集成测试,断言可能如下:

```
assertThat(response.Contains( 巨大的洞穴 ))
```

#### A/B 比较（差异）

A/B 测试不定义明确的断言,而是运行 SUT 的两个副本,发送相同的数据,然后比较输出。预期行为没有明确定义:必须由人工手动检查差异,以确保任何更改都是预期的。

## 大型测试的类型

现在,我们可以将这些不同的方法组合到 SUT、数据和断言中,以创建不同类型的大型测试。然后,每个测试都有不同的属性,包括它可以缓解哪些风险;编写、维护和调试测试需要多少工作量;以及运行测试所需的资源成本。

下面列出了 Google 使用的不同类型的大型测试,包括它们的组成、用途和局限性:

· 一个或多个二进制文件的功能测试 · 浏览器和设备

测试 · 性能、负载和压力测试 · 部

署配置测试 · 探索性测试 · A/B 差异 (回归)

测试 · 用户验收测试 (UAT) · 探测器和金

丝雀分析 · 灾难恢复和混

沌工程

· 用户评价

鉴于组合数量如此之多,因此测试范围也如此之广,我们如何管理何时做什么?起草测试计划是软件设计的一部分,而测试计划的一个关键部分是战略性地概述需要进行哪些类型的测试以及每种测试的数量。该测试策略确定了主要风险向量以及减轻这些风险向量所需的测试方法。

在谷歌,我们有一个专门的工程职位“测试工程师”,我们在寻找一个好的测试工程师时所考虑的因素之一就是能够为我们的产品制定测试策略。

## 一个或多个交互二进制文件的功能测试

此类测试具有以下特点：

- SUT:单机密或云部署隔离 · 数据:手工制作
- 验证:断言

正如我们目前所见,单元测试无法真正准确地测试复杂系统,因为它们的打包方式与实际代码的打包方式不同。许多功能测试场景与给定二进制文件的交互方式与该二进制文件中的类的交互方式不同,这些功能测试需要单独的 SUT,因此是规范的大型测试。

测试多个二进制文件的交互比测试单个二进制文件更复杂,这并不令人意外。一个常见的用例是在微服务环境中,当服务部署为多个单独的二进制文件时。在这种情况下,功能测试可以通过调出由所有相关二进制文件组成的 SUT 并通过已发布的 API 与其交互来覆盖二进制文件之间的实际交互。

## 浏览器和设备测试测试 Web UI

和移动应用程序是对一个或多个交互二进制文件进行功能测试的特殊情况。可以对底层代码进行单元测试,但对于最终用户来说,公共 API 就是应用程序本身。通过前端与应用程序作为第三方进行交互的测试可以提供额外的覆盖层。

## 性能、负载和压力测试

此类测试具有以下特点：

- SUT:云部署隔离 · 数据:手工制作或  
从生产中多路复用 · 验证:差异 (性能指标)

虽然可以测试小单元的性能、负载和压力,但此类测试通常需要同时向外部 API 发送流量。该定义意味着此类测试是多线程测试,通常在被测二进制文件的范围内进行测试。但是,这些测试对于确保版本之间性能没有下降以及系统可以处理预期的流量峰值至关重要。

随着负载测试规模的扩大,输入数据的范围也随之扩大,最终很难生成在负载下触发错误所需的负载规模。负载和压力处理是系统的“高度突发”属性;也就是说,这些复杂的行为属于整个系统,而不是单个成员。因此,重要的是让这些测试尽可能接近生产。每个 SUT 所需的资源与生产所需的资源类似,并且很难减轻来自生产拓扑的噪声。

消除性能测试中的噪音的一个研究领域是修改部署拓扑 各种二进制文件在机器网络上的分布方式。运行二进制文件的机器会影响性能特征;因此,如果在性能差异测试中,基本版本在快速机器(或具有快速网络的机器)上运行,而新版本在慢速机器上运行,则可能会出现性能回归。此特性意味着最佳部署是在同一台机器上运行两个版本。如果一台机器无法容纳二进制文件的两个版本,则另一种方法是通过执行多次运行并消除峰值和谷值来进行校准。

## 部署配置测试

此类测试具有以下特点:

- SUT:单机密封或云部署隔离
- 数据:无
- 验证:断言(不会崩溃)

很多时候,缺陷的根源不是代码,而是配置:数据文件、数据库、选项定义等等。更大规模的测试可以测试 SUT 与其配置文件的集成,因为这些配置文件是在启动给定二进制文件时读取的。

此类测试实际上是 SUT 的烟雾测试,无需太多额外数据或验证。如果 SUT 成功启动,则测试通过。否则,测试失败。

## 探索性测试

此类测试具有以下特点:

- SUT:生产或共享阶段
- 数据:生产或已知的  
测试范围
- 验证方式:手动

探索性测试<sup>2</sup>是一种手动测试,其重点不是通过重复已知测试流程来寻找行为回归,而是通过尝试新的用户场景来寻找可疑行为。经过培训的用户/测试人员通过产品的公共 API 与产品交互,寻找系统中的新路径,以及哪些行为偏离预期或直观行为,或者是否存在安全漏洞。

探索性测试对于新系统和已发布的系统都很有用,可以发现意料之外的行为和副作用。通过让测试人员遵循系统中不同的可达路径,我们可以增加系统覆盖率,并且当这些测试人员发现错误时,可以捕获新的自动化功能测试。从某种意义上说,这有点像功能集成测试的手动“模糊测试”版本。

### 限制

手动测试无法实现亚线性扩展;也就是说,手动测试需要人工投入时间。探索性测试发现的任何缺陷都应通过运行频率更高的自动化测试进行复制。

### Bug bash

我们用于手动探索性测试的一个常见方法是**Bug bash**。一个由工程师和相关人员(经理、产品经理、测试工程师以及任何熟悉该产品的人)组成的团队会安排一次“会议”,但在这次会议上,所有参与者都会手动测试产品。可能会有一些关于 bug 清理的特定重点领域和/或使用系统的起点的发布指南,但目标是提供足够的交互多样性来记录可疑的产品行为和直接的 bug。

## A/B 差异回归测试

此类测试具有以下特点:

- SUT:两个云部署的隔离环境 · 数据:通常从生产中多路复用或采样
- 验证:A/B 差异比较

单元测试涵盖了一小段代码的预期行为路径。但是,对于给定的面向公众的产品,预测许多可能的故障模式是不可能的。

此外,正如海勒姆定律所述,实际的公共 API 不是声明的 API,而是

---

<sup>2</sup> James A. Whittaker,探索性软件测试:指导测试设计的技巧、窍门、导览和技术(纽约:Addison-Wesley Professional,2009 年)。

产品的所有用户可见方面。鉴于这两个特性,A/B 差异测试可能是 Google 最常见的大型测试形式,这并不奇怪。这种方法的概念可以追溯到 1998 年。在 Google,我们自 2001 年以来一直基于此模型对我们的大多数产品进行测试,从广告、搜索和地图开始。

A/B 差异测试通过向公共 API 发送流量并比较新旧版本之间的响应 (尤其是在迁移期间) 来进行。任何行为偏差都必须协调为预期或未预期 (回归)。在这种情况下,SUT 由两组真实二进制文件组成:一组在候选版本上运行,另一组在基础版本上运行。第三个二进制文件发送流量并比较结果。

还有其他变体。我们使用 AA 测试 (将系统与其自身进行比较) 来识别不确定的行为、噪音和不稳定因素,并帮助从 AB 差异中去除这些因素。我们偶尔也会使用 ABC 测试,比较上一个生产版本、基线版本和待定更改,以便一目了然地看到即时更改的影响,以及下一个即将发布的版本的累积影响。

A/B 差异测试是一种廉价但可自动化的方法,可以检测任何已启动系统的意外副作用。

## 限制

差异测试确实带来了一些需要解决的挑战:

### 批准 必须有

人充分了解结果,才能知道是否有任何差异。与典型测试不同,尚不清楚差异是好事还是坏事 (或者基线版本是否有效),因此该过程中通常有一个手动步骤。

## 噪音

对于 diff 测试,任何在结果中引入意外噪音的因素都会导致对结果进行更多的手动调查。有必要对噪音进行补救,这是构建良好 diff 测试的一大复杂性来源。

## 覆盖范围

为 diff 测试生成足够有用的流量可能是一个具有挑战性的问题。测试数据必须覆盖足够多的场景才能识别极端情况的差异,但手动整理此类数据非常困难。

## 设置配

置和维护一个 SUT 相当具有挑战性。同时创建两个 SUT 会使复杂性加倍,尤其是当它们具有相互依赖关系时。

验收测试 (UAT)

此类测试具有以下特点：

- SUT:机器密封或云部署隔离 · 数据:手工制作
- 验证:断言

单元测试的一个关键方面是它们是由编写被测代码的开发人员编写的。但这使得对产品预期行为的误解不仅反映在代码中,还反映在单元测试中。这样的单元测试验证代码是“按实现运行”而不是“按预期运行”。

对于有特定最终客户或客户代理（客户委员会甚至产品经理）的情况,UAT 是通过公共 API 执行产品的自动化测试,以确保特定 **用户旅程的整体行为** 符合预期。存在多个公共框架（例如 Cucumber 和 RSpec）,以使此类测试能够以用户友好的语言编写/读取,通常是在“可运行规范”的上下文中。

Google 实际上并没有进行大量的自动化 UAT,也很少使用规范语言。Google 的许多产品过去都是由软件工程师自己创建的。几乎不需要可运行的规范语言,因为定义预期产品行为的人通常精通实际的编码语言。

## 探测器和金丝雀分析

此类测试具有以下特点：

- SUT:生产 · 数据:生  
产
- 验证:断言和 A/B  
差异（指标）

探测器和金丝雀分析是确保生产环境本身健康的方法。从这些方面来看,它们是生产监控的一种形式,但它们在结构上与其他大型测试非常相似。

探测器是针对生产环境运行编码断言的功能测试。通常这些测试执行众所周知的确定性只读操作,以便即使生产数据随时间发生变化,断言仍然有效。例如,探测器可能会在[www.google.com](http://www.google.com)上执行 Google 搜索并验证结果是否返回,但实际上并不验证结果的内容。在这方面,

它们是生产系统的“烟雾测试”，但它们可以早期发现重大问题。

金丝雀分析与之类似，只不过它关注的是发布何时被推送到生产环境。如果发布是分阶段进行的，我们可以针对升级的（金丝雀）服务运行探测器断言，并比较金丝雀和生产基线部分的健康指标，确保它们没有出问题。

探测器应在任何实时系统中使用。如果生产推出过程包括将二进制文件部署到生产机器的有限子集的阶段（金丝雀阶段），则应在该过程中使用金丝雀分析。

#### 限制

此时（生产中）发现的任何问题都已经影响到最终用户。

如果探测器执行可变（写入）操作，它将修改生产状态。  
这可能导致以下三种结果之一：不确定性和断言失败、将来无法写入或用户可见的副作用。

## 灾难恢复与混沌工程

此类测试具有以下特点：

- SUT: 生产 · 数据: 生产和用户制作（故障注入） · 验证: 手动和 A/B 差异（指标）

这些测试您的系统对意外变化或故障的反应程度。

多年来，谷歌每年都会举办一场名为“**DiRT**”的战争游戏灾难恢复测试（Disaster Recovery Testing）是将近乎全球规模的故障注入我们的基础架构的测试。我们模拟了从数据中心火灾到恶意攻击等各种情况。在一个令人难忘的案例中，我们模拟了一场地震，这场地震使我们位于加利福尼亚州山景城的总部与公司其他部门完全隔绝。这样做不仅暴露了技术缺陷，还揭示了在所有关键决策者都无法联系的情况下经营公司的挑战。3 DiRT 测试的影响需要整个公司的大量协调；相比之下，混沌工程更像是对技术基础架构的“持续测试”。Netflix 使混沌工程流行起来，混沌工程涉及编写程序

---

3 在这次测试中，几乎没有人能完成任何事情，所以很多人放弃了工作，去了一个我们的许多咖啡馆，这样做最终对我们的咖啡馆团队发起了 DDoS 攻击！

不断将背景级别的故障引入您的系统并观察会发生什么。有些故障可能非常严重,但在大多数情况下,混沌测试工具旨在在事情失控之前恢复功能。混沌工程的目标是帮助团队打破稳定性和可靠性的假设,并帮助他们应对构建弹性的挑战。今天,Google 的团队每周使用我们自己开发的名为 Catzilla 的系统执行数千次混沌测试。

这些类型的故障和负面测试对于具有足够的理论容错能力来支持它们的实时生产系统来说是有意义的,并且测试本身的成本和风险也是可以承受的。

## 限制

此时 (生产中)发现的任何问题都已经影响到最终用户。

DiRT 的运行成本相当高,因此我们偶尔会进行协调演习。当我们造成这种程度的中断时,我们实际上会造成痛苦并对员工绩效产生负面影响。

如果探测器执行可变 (写入) 操作,它将修改生产状态。

这可能导致不确定性和断言失败、将来写入能力失败或用户可见的副作用。

## 用户评价

此类测试具有以下特点:

- SUT: 生产 · 数据: 生产
- 验证: 手动和 A/B 差异 (指标)

基于生产的测试可以收集大量有关用户行为的数据。

我们有几种不同的方法来收集有关即将推出的功能的受欢迎程度和问题的指标,这为我们提供了 UAT 的替代方案:

Dogfooding 可以

使用有限的推广和实验,将生产中的功能提供给部分用户。我们有时会与自己的员工一起这样做 (使用我们自己的 dogfood), 他们会在实际部署环境中给我们提供宝贵的反馈。

## 实验 在不知情的情况下

下,将新行为作为实验提供给部分用户。然后,在总体层面上,就某些期望指标将实验组与对照组进行比较。例如,在 YouTube 中,我们

进行了一项有限的实验,改变了视频点赞的工作方式（消除了点踩）,只有一部分用户看到了这一变化。

对于谷歌来说,这是一个**非常重要的方法**。Noogler 加入公司后听到的第一个故事是关于谷歌发起一项实验,改变谷歌搜索中 AdWords 广告的背景阴影颜色,并注意到实验组用户的广告点击次数与对照组相比显着增加。

#### 评分员评估 人工评分员

会看到给定操作的结果,并选出哪个操作“更好”以及原因。然后根据反馈确定给定的更改是积极的、中性的还是消极的。例如,Google 过去一直使用评分员评估来处理搜索查询（我们已经发布了我们为评分员提供的指导方针）。在某些情况下,来自这些评级数据的反馈可以帮助确定算法更改是否可行。评分员评估对于机器学习系统等非确定性系统至关重要,因为这些系统没有明确的正确答案,只有更好或更坏的概念。

## 大型测试和开发人员工作流程

我们讨论了什么是大型测试、为什么要进行大型测试、何时进行大型测试以及进行多少测试,但我们还没有过多讨论谁来编写测试。谁来编写测试?

谁负责运行测试并调查失败原因?谁负责测试?我们如何才能容忍这种情况?

尽管标准单元测试基础设施可能不适用,但将大型测试集成到开发人员工作流程中仍然至关重要。实现这一点的一种方法是确保存在用于提交前和提交后执行的自动化机制,即使这些机制与单元测试机制不同。在 Google,许多此类大型测试不属于 TAP。它们不密封、太不稳定和/或太耗资源。

但我们仍需要防止它们崩溃,否则它们不会发出任何信号,变得难以分类。因此,我们所做的就是为这些内容进行单独的提交后持续构建。我们还鼓励在提交前运行这些测试,因为这会直接向作者提供反馈。

需要手动批准差异的 A/B 差异测试也可以纳入此类工作流程。对于提交前,在批准更改之前,可以要求在 UI 中批准任何差异,这是代码审查要求。如果提交的代码存在未解决的差异,我们会自动记录阻止发布的错误。

在某些情况下,测试规模过大或过于繁琐,以至于提交前执行会给开发人员带来太多麻烦。这些测试仍在提交后运行,并且也是发布过程的一部分。不运行这些提交前测试的缺点是污点会进入 monorepo,我们需要找出罪魁祸首更改才能将其回滚。但我们

需要在开发人员的痛苦和所产生的变更延迟以及持续构建的可靠性之间做出权衡。

编写大型测试虽然大型测试的结构

相当标准,但创建这样的测试仍然存在挑战,特别是如果团队中有人是第一次这样做。

编写此类测试的最佳方法是拥有清晰的库、文档和示例。单元测试很容易编写,因为有原生语言支持 (JUnit 曾经很深奥,但现在已成为主流)。我们将这些断言库重用于功能集成测试,但随着时间的推移,我们也创建了用于与 SUT 交互、运行 A/B 差异、播种测试数据和编排测试工作流的库。

大型测试的维护成本更高,无论是资源还是人力,但并非所有大型测试都是一样的。A/B diff 测试受欢迎的一个原因是它们在维护验证步骤方面的人力成本较低。同样,生产 SUT 的维护成本也低于孤立的密封 SUT。而且由于所有这些

必须维护编写的基础设施和代码,成本节省就会增加。

然而,必须从整体上考虑这一成本。如果手动协调差异或支持和保护生产测试的成本超过节省的成本,那么这种方法就无效了。

运行大型测试我们上面提到过,

我们的大型测试不适合 TAP,因此我们为它们提供了替代的连续构建和预提交。我们的工程师面临的最初挑战之一是如何运行非标准测试以及如何对其进行迭代。

我们尽可能地尝试以工程师熟悉的方式运行大型测试。我们的预提交基础设施在运行这些测试和运行 TAP 测试之前放置了一个通用 API,我们的代码审查基础设施显示这两组结果。但许多大型测试都是定制的,因此需要特定的文档来说明如何按需运行它们。这可能会让不熟悉的工程师感到沮丧。

加快测试速度工程师

不会等待缓慢的测试。测试速度越慢,工程师运行它的频率就越低,失败后等待再次通过的时间就越长。

加快测试速度的最佳方法通常是缩小测试范围或将大型测试拆分为两个可以并行运行的小型测试。但您还可以使用其他一些技巧来加快大型测试的速度。

一些简单的测试会使用基于时间的休眠来等待不确定的操作发生,这在大型测试中很常见。但是,这些测试没有线程限制,而实际生产用户希望等待的时间尽可能短,因此测试最好以实际生产用户的方式做出反应。方法包括:

- 在时间窗口内反复轮询状态转换,以接近微妙的频率完成事件。如果测试未能达到稳定状态,您可以将其与超时值结合使用。
- 实现事件处理程序。 · 订阅事件完成通知系统。

请注意,当运行这些测试的集群超载时,依赖于睡眠和超时的测试都将开始失败,由于这些测试需要更频繁地重新运行,从而进一步增加负载,因此这种情况会不断加剧。

#### 降低内部系统超时和延迟

生产系统通常采用分布式部署拓扑结构进行配置,但 SUT 可能部署在单台机器上(或至少是一组共置机器)。如果生产代码中存在硬编码超时或(尤其是)睡眠语句来解释生产系统延迟,则应在运行测试时对其进行调整并减少这些操作。

#### 优化测试构建时间 我们的 monorepo

的一个缺点是,大型测试的所有依赖项都是构建并作为输入提供的,但对于某些较大的测试来说,这可能不是必需的。如果 SUT 由真正是测试焦点的核心部分和一些其他必要的对等二进制依赖项组成,则可能可以使用已知良好版本的这些其他二进制文件的预构建版本。我们的构建系统(基于 monorepo)无法轻松支持此模型,但这种方法实际上更能反映不同服务以不同版本发布的生产环境。

#### 消除不稳定因素不稳

定因素对于单元测试来说已经够糟糕了,对于更大规模的测试来说,它可能会让其变得无法使用。团队应该将消除此类测试的不稳定性视为重中之重。但如何才能消除此类测试的不稳定性呢?

减少不稳定性首先要缩小测试范围 封闭式 SUT 不会面临生产或共享暂存环境中多用户和真实世界中的不稳定性风险,单机封闭式 SUT 不会存在分布式 SUT 的网络和部署不稳定性问题。但您可以通过测试设计和实施以及其他技术来缓解其他不稳定性问题。

在某些情况下,您需要在这些方面与测试速度之间取得平衡。

就像使测试具有响应性或事件驱动性可以加快速度一样,它也可以消除不稳定因素。定时休眠需要超时维护,这些超时可以嵌入到测试代码中。增加内部系统超时可以减少不稳定因素,而如果系统以不确定的方式运行,则减少内部超时可能会导致不稳定因素。这里的关键是确定一种权衡,既可以定义最终用户可容忍的系统行为(例如,我们允许的最大超时为 n 秒),又可以很好地处理不稳定的测试执行行为。

内部系统超时的一个更大问题是,超过该超时时间会导致难以分类的错误。生产系统通常会尝试通过妥善处理可能的内部系统问题来限制最终用户遭受灾难性故障的可能性。例如,如果 Google 无法在给定的时间限制内投放广告,我们不会返回 500,我们只是不会投放广告。但在测试运行者看来,当存在不稳定的超时问题时,广告投放代码可能会出现问题。在这种情况下,重要的是使故障模式显而易见,并使其易于针对测试场景调整此类内部超时。

### 让测试易于理解一个特定情况

是,当测试结果对于运行测试的工程师来说难以理解时,测试就很难集成到开发人员的工作流程中。即使是单元测试也会产生一些混乱 - 如果我所做的更改破坏了你的测试,如果我对你的代码不熟悉,可能很难理解为什么 - 但对于更大的测试,这种混乱是无法克服的。自信的测试必须提供明确的通过/失败信号,并且必须提供有意义的错误输出以帮助分类失败的根源。需要人工调查的测试,如 A/B 差异测试,需要特殊处理才能有意义,否则可能会在提交前被跳过。

这在实践中如何发挥作用?一个失败的大型测试应该做到以下几点:

#### 有一条消息清楚地表明故障原因

最糟糕的情况是出现错误,只显示“断言失败”和堆栈跟踪。好的错误会预测测试运行者对代码的不熟悉,并提供一条提供上下文的消息:“在 test>ReturnsOneFullPageOfSearchResultsForAPopularQuery 中,预期有 10 个搜索结果,但只得到 1 个。”对于失败的性能或 A/B 差异测试,输出中应该清楚地解释正在测量的内容以及为什么该行为被视为可疑。

### 尽量减少找出差异根本原因所需的努力

堆栈跟踪对于大型测试来说没什么用,因为调用链可能跨越多个进程边界。相反,有必要生成跨调用链的跟踪,或者投资于可以缩小罪魁祸首范围的自动化。测试应该为此生成某种工作。例如, [Dapper](#)是Google使用的一个框架,用于将单个请求ID与RPC调用链中的所有请求关联起来,并且该请求的所有关联日志都可以通过该ID进行关联,以方便追踪。

### 提供支持和联系信息。

通过让测试的所有者和支持者能够轻松联系,测试运行者能够轻松获得帮助。

### 负责大型测试

必须有记录在案的负责人能够充分审查测试变更并在测试失败时提供支持的工程师。如果没有适当的责任感,测试可能会成为以下问题的牺牲品:

- 贡献者修改和更新测试变得更加困难 · 解决测试失败需要更长的时间

并且测试失败。

特定项目内组件的集成测试应由项目负责人负责。以功能为中心的测试(涵盖一组服务中的特定业务功能的测试)应由“功能所有者”负责;在某些情况下,此所有者可能是负责端到端功能实现的软件工程师;在其他情况下,可能是负责业务场景描述的产品经理或“测试工程师”。负责测试的人必须有权确保其整体健康,并且必须既有能力支持其维护,又有这样做的动机。

如果以结构化的方式记录这些信息,则可以围绕测试负责人构建自动化。我们使用的一些方法包括:

#### 常规代码所有权 在许多情

况下,较大的测试是位于代码库中特定位置的独立代码工件。在这种情况下,我们可以使用 monorepo 中已有的 OWNERS ([第9章](#))信息来向自动化提示特定测试的所有者是测试代码的所有者。

#### 每个测试注解 在某些情况

下,可以将多个测试方法添加到单个测试类或模块中,并且每个测试方法可以具有不同的特性所有者。我们使用

每种语言的结构化注释来记录每种情况下的测试所有者,以便如果特定测试方法失败,我们可以确定要联系的所有者。

## 结论

全面的测试套件需要更大规模的测试,既要确保测试与被测系统的保真度相匹配,又要解决单元测试无法充分覆盖的问题。由于此类测试必然更复杂且运行速度更慢,因此必须小心确保此类大型测试得到适当的拥有、维护良好,并在必要时运行(例如在部署到生产之前)。总体而言,此类大型测试仍必须尽可能小(同时仍保持保真度),以避免开发人员产生摩擦。对于大多数软件项目来说,全面的测试策略是必要的,它可以识别系统的风险,并进行更大规模的测试来应对这些风险。

## TL;DR

- 大型测试涵盖单元测试无法涵盖的内容。
- 大型测试由测试系统、数据、操作和验证组成。
- 良好的设计包括识别风险的测试策略和大型测试减轻这些影响。
- 必须对更大规模的测试付出额外的努力,以防止它们在开发人员的工作流程中造成摩擦。



## 第十五章

# 弃用

作者:海伦·莱特  
编辑:Tom Mansreck

我喜欢最后期限。我喜欢它们飞逝而过的嗖嗖声。

道格拉斯·亚当斯

所有系统都会老化。尽管软件是一种数字资产,物理部分本身不会退化,但随着时间的推移,新技术、库、技巧、语言和其他环境变化会使现有系统过时。旧系统需要持续维护、深奥的专业知识,并且随着它们与周围生态系统的分离,通常需要更多的工作。投入精力关闭过时的系统通常比让它们无限期地与替代它们的系统一起运行更好。但仍然运行的过时系统的数量表明,在实践中,这样做并非易事。我们将有序迁移并最终移除过时系统的过程称为弃用。

弃用是另一个更准确地说属于软件工程学科而非编程学科的话题,因为它需要思考如何随时间管理系统。对于长期运行的软件生态系统,正确规划和执行弃用可以减少资源成本,并通过消除系统中随时间积累的冗余和复杂性来提高速度。另一方面,不当弃用系统的成本可能比保持原样更高。

虽然弃用系统需要额外的努力,但可以在系统设计期间规划弃用,以便最终更容易退役和移除它。弃用可能会影响从单个函数调用到整个软件堆栈的系统。为了具体起见,下面的大部分内容都集中在代码级弃用上。

与本书讨论的大多数其他主题不同,Google 仍在学习如何最好地弃用和删除软件系统。本章介绍

我们在弃用大型且使用频繁的内部系统时所得到的教训。

有时,它会按预期工作,有时则不然,但删除过时系统的普遍问题仍然是业界的一个困难且不断发展的問題。

本章主要讨论弃用技术系统,而不是最终用户产品。这种区分有些武断,因为面向外部的 API 只是另一种产品,而内部 API 的消费者可能认为自己是最终用户。虽然许多原则都适用于拒绝公共产品,但我们在乎里关注的是弃用和删除过时系统的技术和政策方面,在这种情况下,系统所有者可以查看其

使用。

## 为何弃用?

我们对弃用问题的讨论始于一个基本前提,即代码是一种负债,而不是资产。毕竟,如果代码是一种资产,我们为什么还要花时间尝试关闭和删除过时的系统?代码是有成本的,其中一些成本是在创建系统的过程中承担的,但许多其他成本是在系统在其整个生命周期内维护时承担的。这些持续的成本,例如保持系统运行所需的运营资源或随着周围生态系统的发展不断更新其代码库的努力,意味着值得评估在保持老化系统运行和努力关闭它之间的权衡。

系统年限的长短并不能成为其过时的理由。一个系统可以经过数年精心设计,成为软件形式和功能的典范。一些软件系统,例如 LaTeX 排版系统,经过数十年的改进,尽管仍然会发生变化,但变化很少。仅仅因为某样东西老旧,并不意味着它已经过时。

弃用最适合于明显过时且存在提供类似功能的替代系统的系统。新系统可能更有效地利用资源、具有更好的安全属性、以更可持续的方式构建,或者只是修复错误。使用两个系统来完成同一件事似乎不是一个紧迫的问题,但随着时间的推移,维护它们的成本可能会大幅增长。用户可能需要使用新系统,但仍有使用过时系统的依赖项。

这两个系统可能需要相互接口,这需要复杂的转换代码。随着两个系统的发展,它们可能会相互依赖,最终很难移除任何一个。从长远来看,我们发现,让多个系统执行相同的功能也会阻碍新系统的发展,因为它仍然需要保持兼容性。

用旧系统替换旧系统。花费精力移除旧系统是值得的,因为替换系统现在可以更快地发展。

之前我们曾断言“代码是负债,而不是资产”。如果这是真的,为什么我们要在这本书中花大部分篇幅来讨论构建可以存活数十年的软件系统的最有效方法?既然代码最终只会出现在资产负债表的负债方,为什么还要花那么多精力去编写更多代码呢?

代码本身并不带来价值:它提供的功能才带来价值。

如果该功能满足了用户需求,那么它就是一笔资产:实现该功能的代码只是实现该目的的一种手段。如果我们能从一行可维护、可理解的代码中获得与 10,000 行错综复杂的意大利面条式代码相同的功能,我们更愿意选择前者。代码本身是有成本的 在保持相同功能量的情况下,代码越简单越好。

我们不应该关注我们能写出多少代码,或者我们的代码库有多大,而应该关注每单位代码能提供多少功能,并尽量最大化这个指标。最简单的方法之一不是编写更多代码并希望获得更多功能;而是删除不再需要的多余代码和系统。弃用政策和程序使这成为可能。

尽管弃用很有用,但我们在 Google 了解到,从执行弃用的团队以及这些团队的客户的角度来看,组织对于同时进行的合理弃用工作量有一个限制。例如,虽然每个人都喜欢新铺的道路,但如果公共工程部门决定同时关闭每条道路进行铺路,那么没有人会去任何地方。通过集中精力,铺路队可以更快地完成特定工作,同时让其他交通得以进展。

同样,谨慎选择弃用项目并承诺完成它们也很重要。

## 为何弃用如此困难?

我们在本书的其他地方提到过海伦定律,但值得在这里重申它的适用性:系统的用户越多,用户以意想不到和无法预见的方式使用它的可能性就越高,并且弃用和删除这样的系统就越困难。他们的使用只是“碰巧有效”而不是“保证有效”。在这种情况下,删除系统可以被认为是最终的改变:我们不仅仅是改变行为,而是完全删除该行为!

这种彻底的改变将会产生一批意想不到的依赖者。

更复杂的是,通常情况下,除非有提供相同(或更好!)功能的新系统可用,否则不会弃用。新系统

可能会更好,但也有不同之处:毕竟,如果它与旧系统完全相同,它就不会给迁移到它的用户带来任何好处(尽管它可能使运营它的团队受益)。这种功能差异意味着旧系统和新系统之间很少出现一对一的匹配,并且每次使用旧系统都必须在新系统的背景下进行评估。

另一个令人惊讶的不愿弃用的原因是对旧系统的情感依恋,尤其是那些弃用者参与创建的系统。这种厌恶改变的一个例子发生在谷歌系统性地删除旧代码时:我们偶尔会遇到“我喜欢这个代码!”这样的抵制。说服工程师拆除他们花了数年时间建立的东西可能很困难。

这是一种可以理解的反应,但最终会弄巧成拙:如果一个系统过时了,它会给组织带来净成本,应该被移除。我们解决Google内部保留旧代码问题的方法之一是确保源代码存储库不仅可以在主干中搜索,还可以在历史记录中搜索。即使是被删除的代码也可以再次找到(参见第17章)。

Google内部流传着一个老笑话:做事有两种方式:一种是过时的,一种是尚未准备好的。这通常是新解决方案“几乎”完成的结果,也是在复杂且快节奏的技术环境中工作的不幸现实。

Google工程师已经习惯了在这种环境下工作,但它仍然令人不安。良好的文档、大量的指示牌以及帮助完成弃用和迁移过程的专家团队都让您更容易知道是否应该使用存在所有缺陷的旧版本,还是使用存在所有不确定性的新版本。

最后,资助和执行弃用工作在政治上可能很困难;组建团队并花时间移除过时的系统需要花费真金白银,而什么都不做、让系统无人看管地缓慢运行的成本却不易观察到。很难说服相关利益相关者弃用工作是值得的,特别是如果它们对新功能开发产生负面影响的话。研究技术(如第7章中描述的技术)可以提供具体证据证明弃用是值得的。

鉴于弃用和删除过时的软件系统的难度,用户通常更容易在原地开发系统,而不是完全替换它。增量式并不能完全避免弃用过程,但它确实将其分解为更小、更易于管理的部分,从而产生增量效益。在Google内部,我们观察到迁移到全新系统的成本极高,而且成本经常被低估。增量式弃用工作

通过就地重构可以保持现有系统运行,同时更容易地向用户提供价值。

设计期间的弃用与许多工程活动一样,软

件系统的弃用可以在首次构建时进行规划。编程语言、软件架构、团队组成,甚至公司政策和文化的选择都会影响系统在达到使用寿命后最终被移除的难易程度。

设计系统以便最终可以弃用的概念在软件工程中可能是激进的,但在其他工程学科中却很常见。

以核电站为例,这是一个极其复杂的工程。作为核电站设计的一部分,必须考虑到它在生产服务终生后最终退役的问题,甚至为此拨款。<sup>1</sup>当工程师知道核电站最终需要退役时,建造核电站的许多设计选择都会受到影响。

不幸的是,软件系统很少经过如此深思熟虑的设计。许多软件工程师都热衷于构建和发布新系统,而不是维护现有系统。包括谷歌在内的许多公司的企业文化都强调快速构建和发布新产品,这通常会阻碍人们从一开始就考虑弃用。尽管软件工程师是数据驱动的自动机这一流行概念,但从心理上来说,为我们辛苦打造的作品的最终消亡做好规划可能很困难。

那么,在设计未来更容易弃用的系统时,我们应该考虑哪些因素呢?以下是我们鼓励 Google 工程团队提出的几个问题:

- 我的消费者从我的产品转向潜在产品有多容易?  
替代品?
- 如何逐步替换系统的各个部件?

其中许多问题与系统如何提供和使用依赖项有关。

有关如何管理这些依赖关系的更详细讨论,请参见第 16 章。

---

1 “促进核电站退役的设计和建造”,技术报告系列

国际原子能机构,维也纳,第 382 号 (1997 年)。

最后,我们应该指出,当组织第一次决定构建项目时,就会决定是否长期支持该项目。软件系统存在后,剩下的唯一选择是支持它、小心地弃用它,或者在某些外部事件导致其崩溃时让它停止运行。这些都是有效的选择,它们之间的权衡将因组织而异。一家只有单个项目的创业公司在公司破产时会毫不留情地将其关闭,但一家大公司在考虑删除旧项目时,需要更仔细地考虑其投资组合和声誉的影响。如前所述,Google 仍在学习如何最好地利用我们自己的内部和外部产品进行这些权衡。

简而言之,不要启动组织不承诺在预期生命周期内提供支持的项目。即使组织选择弃用和删除该项目,仍然会产生成本,但可以通过规划和投资工具和政策来减轻这些成本。

## 弃用类型

弃用并不是单一的过程,而是一个连续的过程,从“我们希望有一天能关闭它”到“这个系统明天就要消失了,客户最好做好准备。”广义上讲,我们将这个连续过程分为两个独立的领域:建议性和强制性。

建议性弃用建议性弃用是指那些

没有截止日期并且对组织来说不是高优先级的弃用(并且公司不愿意为其投入资源)。

这些也可以称为期望弃用:团队知道系统已被替换,尽管他们希望客户最终会迁移到新系统,但他们没有立即提供支持以帮助迁移客户或删除旧系统的计划。这种弃用通常缺乏执行力:我们希望客户迁移,但不能强迫他们迁移。正如我们在 SRE 的朋友会毫不犹豫地告诉你的那样:“希望不是一种策略。”

建议弃用是一种很好的工具,可用于宣传新系统的存在并鼓励早期采用者开始试用。这样的新系统不应处于测试阶段:它应该已准备好用于生产用途和负载,并应准备好无限期地支持新用户。当然,任何新系统都会经历成长的烦恼,但在旧系统以任何方式弃用后,新系统将成为组织基础设施的重要组成部分。

在 Google,我们看到一种情况是,当新系统为用户提供令人信服的好处时,建议弃用会带来很大的好处。在这些情况下,

只需通知用户这个新系统并向他们提供自助工具以迁移到该系统,通常就能鼓励用户采用。然而,好处不能只是渐进的:它们必须是变革性的。用户会犹豫是否要为了边际效益而自行迁移,即使是经过巨大改进的新系统,仅通过建议性弃用工作也无法获得全面采用。

建议性弃用允许系统作者推动用户朝着期望的方向发展,但不应指望他们完成大部分迁移工作。人们往往倾向于简单地在旧系统上放置弃用警告,然后不再做任何进一步的努力。我们在 Google 的经验是,这可能会导致过时系统的新用途(略有)减少,但很少会导致团队主动迁移。旧系统的现有用途对其产生了某种概念(或技术)吸引力:无论我们说多少次“请使用新系统”,旧系统的相对多的用途往往会占据大量新用途。除非更积极地鼓励用户迁移,否则旧系统将继续需要维护和其他资源。

#### 强制弃用这种积极的鼓励以强制

弃用的形式出现。这种弃用通常伴随着一个移除过时系统的最后期限:如果用户在该期限之后继续依赖它,他们会发现他们自己的系统不再工作。

与直觉相反,强制弃用工作扩大规模的最佳方式是将迁移用户的专业知识集中在一个专家团队中通常是负责完全删除旧系统的团队。这个团队有动力帮助其他人从过时的系统迁移,并可以积累经验和工具,然后可以在整个组织中使用。许多这样的迁移都可以使用[第 22 章中讨论的相同工具](#)来实现。

要使强制弃用真正发挥作用,其时间表需要有一个执行机制。这并不意味着时间表不能改变,而是赋予运行弃用流程的团队权力,在通过迁移工作充分警告不合规用户后,让他们放弃这些用户。如果没有这种权力,客户团队很容易忽略弃用工作,转而关注功能或其他更紧迫的工作。

同时,在没有人员配备的情况下强制弃用可能会让客户团队觉得他们心胸狭窄,这通常会阻碍弃用的完成。客户只是将这种弃用工作视为一项没有资金支持的任务,要求他们把自己的优先事项放在一边,只是为了保持服务正常运行而工作。这很像“原地踏步”现象,并在基础设施维护人员和他们的客户之间造成了摩擦。这是因为

原因在于我们强烈主张强制折旧应由专门的团队积极负责直至完成。

还值得注意的是,即使有政策支持,强制弃用仍会面临政治障碍。想象一下,当旧系统的最后一个用户是整个组织所依赖的关键基础设施时,试图强制执行弃用工作。你有多愿意为了设定一个任意的最后期限而破坏该基础设施(以及依赖它的所有人)?如果该团队可以否决其进展,很难相信弃用真的是强制性的。

Google 的单体式存储库和依赖关系图让我们能够深入了解整个生态系统中系统的使用情况。即便如此,一些团队可能甚至不知道他们依赖某个过时的系统,而且很难通过分析发现这些依赖关系。也可以通过增加测试频率和持续时间来动态地找到它们,在此期间旧系统会暂时关闭。这些有意的更改提供了一种机制,可以通过查看哪些地方出现了问题来发现非预期的依赖关系,从而提醒团队需要为即将到来的最后期限做好准备。在 Google 内部,我们偶尔会更改仅实现符号的名称,以查看哪些用户在不知不觉中依赖它们。

在 Google,当某个系统计划弃用和移除时,团队通常会在停用前的几个月和几周内宣布计划停机时间,并延长停机时间。与 Google 的灾难恢复测试(DiRT)练习类似,这些事件通常会发现正在运行的系统之间的未知依赖关系。这种渐进式方法允许那些依赖关系的团队发现并规划系统的最终移除,甚至可以与弃用团队合作调整他们的时间表。(同样的原则也适用于静态代码依赖关系,但静态分析工具提供的语义信息通常足以检测过时系统的所有依赖关系。)

### 弃用警告无论是建议性弃用还

是强制弃用,通常都可以通过编程方式将系统标记为弃用,以便警告用户其使用情况并鼓励用户离开。人们往往倾向于将某些东西标记为弃用,并希望其用途最终消失,但请记住:“希望不是一种策略。”

弃用警告可以帮助防止新的用途,但很少导致现有系统的迁移。

实践中通常发生的情况是,这些警告会随着时间的推移而累积。如果它们在传递上下文中使用(例如,库 A 依赖于库 B,而库 B 又依赖于库 C,而库 C 发出警告,该警告在库 A 构建时出现),这些警告很快就会让系统用户不知所措,以至于他们完全忽略它们。在医疗保健领域,这种现象被称为“**警报疲劳**”。

向用户发出的任何弃用警告都需要具备两个属性：可操作性和相关性。如果用户可以利用警告实际执行某些相关操作，那么该警告就是可操作的，这不仅是在理论上，而且在实践中，因为考虑到我们期望普通工程师在该问题领域的专业知识。例如，某个工具可能会警告应将对给定函数的调用替换为对其更新对应函数的调用，或者一封电子邮件可能会概述将数据从旧系统移动到新系统所需的步骤。在每种情况下，警告都提供了工程师可以执行的后续步骤，以不再依赖弃用的系统。<sup>2</sup>警告可以是可操作的，但仍然令人讨厌。为了有用，弃用警告还应该具有相关性。如果警告在用户实际执行指示的操作时出现，则警告是相关的。关于使用弃用函数的警告最好在工程师编写使用该函数的代码时发出，而不是在代码被签入存储库几周后发出。同样，数据迁移电子邮件最好在删除旧系统前几个月发送，而不是在删除前的周末才想到发送。

重要的是抵制对所有可能事物发出弃用警告的冲动。

警告本身并不坏，但简单的工具通常会产生大量警告消息，可能会让毫无戒心的工程师不知所措。在 Google 内部，我们非常宽容地将旧函数标记为已弃用，但会利用 ErrorProne 等工具或 clang-tidy 以确保警告以有针对性的方式出现。如第 20 章所述，我们将这些警告限制在新更改的行中，以此警告人们弃用符号的新用途。只有强制弃用时才会添加更具侵入性的警告，例如依赖图中的弃用目标，团队正在积极地将用户移走。无论哪种情况，工具在向适当的人适时呈现适当的信息方面都发挥着重要作用，从而允许添加更多警告而不会让用户感到疲劳。

## 管理弃用流程

虽然由于我们是在解构系统而不是构建系统，所以它们看起来就像是不同类型的项目，但弃用项目在管理和运行方式上与其他软件工程项目类似。我们不会花太多精力来讨论这些管理工作之间的相似之处，但值得指出它们的不同之处。

---

<sup>2</sup>请参阅 <https://abseil.io/docs/cpp/tools/api-upgrades> 举个例子。

## 流程所有者

我们在 Google 了解到,如果没有明确的所有者,弃用流程就不太可能取得有意义的进展,无论系统可能生成多少警告和警报。让明确的项目所有者负责管理和运行弃用流程似乎是一种资源浪费,但替代方案甚至更糟糕:永远不要弃用任何东西,或者将弃用工作委托给系统用户。第二种情况只是建议性的弃用,永远不会自然完成,第一种情况是承诺无限期地维护每个旧系统。集中弃用工作有助于更好地确保专业知识实际上通过使其更加透明来降低成本。

废弃项目通常会在确定所有权和协调激励机制时带来问题。每个规模合理的组织都有仍在积极使用但无人明确拥有或维护的项目,Google 也不例外。项目有时会进入这种状态,因为它们已被弃用:原始所有者已转向后续项目,而过时的项目则被搁置在地下室,仍然是关键项目的依赖项,并希望它最终会逐渐消失。

此类项目不太可能自行消失。尽管我们抱有最大的希望,但我们发现这些项目仍然需要弃用专家来移除它们,并防止它们在不合时宜的时候失败。这些团队应该将移除作为他们的首要目标,而不仅仅是其他工作的副项目。在优先级相互竞争的情况下,弃用工作几乎总是被认为具有较低的优先级,并且很少得到它所需的关注。这些重要但不紧急的清理任务是对 20% 时间的极好利用,并为工程师提供了接触代码库其他部分的机会。

## 里程碑

在构建新系统时,项目里程碑通常非常明确:“下个季度推出 frobnazzer 功能。”按照增量开发实践,团队逐步构建并向用户交付功能,用户每次利用新功能时都会获得收益。最终目标可能是推出整个系统,但增量里程碑有助于让团队了解进度,并确保他们不必等到流程结束才能为组织创造价值。

相比之下,人们常常会觉得弃用流程的唯一里程碑就是彻底移除过时的系统。团队会觉得,直到关灯回家,他们才取得任何进展。虽然这可能是团队最有意义的一步,但如果团队正确地完成了工作,它通常是团队外部最不引人注意的一步,因为到那时,过时的系统已经没有任何用户了。弃用项目经理应该抵制以下诱惑:

使其成为唯一可衡量的里程碑,特别是考虑到它可能不会发生在所有弃用项目中。

与构建新系统类似,管理从事弃用工作的团队应该涉及具体的增量里程碑,这些里程碑是可衡量的,并为用户带来价值。用于评估弃用进度的指标会有所不同,但庆祝弃用过程中的增量成就仍然有利于士气。我们发现识别适当的增量里程碑(例如删除关键子组件)很有用,就像我们识别构建新产品的成就一样。

#### 弃用工具本书其他部分深入讨论

了用于管理弃用过程的许多工具,例如大规模变更(LSC)过程([第22章](#))或我们的代码审查工具([第19章](#))。我们不会讨论这些工具的具体细节,而是简要概述这些工具在管理过时系统的弃用时如何有用。这些工具可分为发现、迁移和倒退预防工具。

#### 发现在弃用

过程的早期阶段,事实上在整个过程中,了解过时系统是如何使用的以及由谁使用是很有用的。弃用的大部分初始工作是确定谁在使用旧系统,以及以哪些意想不到的方式使用。根据使用类型,此过程可能需要在获得新信息后重新审视弃用决定。我们还在整个弃用过程中使用这些工具来了解工作的进展情况。

在Google内部,我们使用代码搜索([参见第17章](#))和Kythe([参见第23章](#))等工具来静态确定哪些客户使用了给定的库,并且经常对现有使用情况进行抽样,以了解客户意外依赖哪些类型的行为。由于运行时依赖项通常需要一些静态库或瘦客户端使用,因此这种技术可以产生启动和运行弃用过程所需的大量信息。生产中的日志记录和运行时抽样有助于发现动态依赖项的问题。

最后,我们将全局测试套件视为一个预言机,以确定是否已删除对旧符号的所有引用。如[第11章所述](#),测试是一种防止生态系统演变过程中系统发生不必要的行为变化的机制。弃用是这种演变的重要组成部分,客户有责任进行充分的测试,以确保删除过时的系统不会对他们造成损害。

## 迁移

Google 的大部分弃用工作都是通过使用我们前面提到的同一套代码生成和审查工具来完成的。LSC 流程和工具在管理实际更新代码库以引用新库或运行时服务的大量工作时特别有用。

## 防止倒退最后,一个经

常被忽视的弃用基础设施是用于防止添加正在被主动删除的东西的新用途的工具。即使是建议性弃用,在用户编写新代码时,警告用户避开弃用的系统而选择新系统也是有用的。如果没有倒退预防,弃用可能会变成一场打地鼠游戏,用户不断添加他们熟悉的系统的新用途(或在代码库的其他地方找到示例),而弃用团队不断迁移这些新用途。这个过程既适得其反,又令人沮丧。

为了在微观层面上防止弃用倒退,我们使用 Tricorder 静态分析框架通知用户他们正在向弃用系统添加调用,并向他们提供关于适当替换的反馈。弃用系统的所有者可以向弃用符号添加编译器注释(例如@deprecated Java 注释),Tricorder 会在审查时显示这些符号的新用途。这些注释将消息控制权交给拥有弃用系统的团队,同时自动提醒变更作者。在有限的情况下,该工具还会建议一键修复以迁移到建议的替换。

从宏观层面上讲,我们在构建系统中使用可见性白名单,以确保不会将新的依赖项引入到已弃用的系统中。自动化工具会定期检查这些白名单,并在依赖系统从过时系统中迁移出来时将其删除。

## 结论

弃用可能感觉像是马戏团游行刚刚穿过城镇后清理街道的脏活,但这些努力通过减少维护开销和工程师的认知负担改善了整个软件生态系统。

随着时间的推移,可扩展地维护复杂的软件系统不仅仅是构建和运行软件:我们还必须能够删除过时或未使用的系统。

完整的弃用流程需要通过政策和工具成功应对社会和技术挑战。以有组织且管理良好的方式进行弃用通常被忽视为组织带来的好处,但对于组织的长期可持续性而言却至关重要。

## TL;DR

- 软件系统有持续的维护成本,应将其与移除成本进行权衡。 · 移除东西通常比一开始构建它们更困难,因为

现有用户的使用方式经常超出系统最初的设计。

- 如果将调节成本计算在内,改进现有系统通常比更换新系统更便宜。

- 很难诚实地评估决定是否弃用所涉及的成本:除了保留旧系统所涉及的直接维护成本之外,还有多个类似的系统可供选择且可能需要互操作的生态系统成本。旧系统可能会暗中拖累新功能的开发。这些生态系统成本是分散的,难以衡量。弃用和移除成本通常也同样分散。



## 第四部分

# 工具



## 第十六章

# 版本控制和分支管理

作者:Titus Winters  
由 Lisa Carey 编辑

在整个行业中,也许没有一种软件工程工具像版本控制一样被广泛采用。很难想象有哪个规模大于几个人的软件组织不依赖正式的版本控制系统 (VCS) 来管理其源代码并协调工程师之间的活动。

在本章中,我们将探讨为何版本控制的使用已成为软件工程中如此明确的规范,并介绍版本控制和分支管理的各种可能方法,包括我们如何在整个 Google 范围内大规模实施。我们还将分析各种方法的优缺点;尽管我们认为每个人都应该使用版本控制,但某些版本控制策略和流程可能比其他策略和流程更适合您的组织(或总体而言)。特别是,我们发现 DevOps<sup>1</sup> 所推广的“基于主干的开发”(一个存储库,没有开发分支)是一种特别可扩展的策略方法,我们将提供一些建议来解释其原因。

## 什么是版本控制?



本节对许多读者来说可能有点基础:毕竟,版本控制的使用相当普遍。如果您想跳过这一节,我们建议您跳到[第 334 页的“事实来源”一节](#)。

<sup>1</sup> DevOps 研究协会在本章初稿撰写至出版期间被 Google 收购,并在年度“DevOps 现状报告”和《Accelerate》一书中广泛发表了关于这一问题的文章。

据我们所知,它推广了基于主干的开发这一术语。

VCS 是一种跟踪文件随时间变化的修订（版本）的系统。VCS 维护着所管理文件集的一些元数据，文件和元数据的副本统称为存储库<sup>2</sup>（简称 repo）。VCS 允许多个开发人员同时处理同一组文件，从而帮助协调团队活动。早期的 VCS 通过一次授予一个人编辑文件的权限来实现这一点。这种锁定方式足以建立排序（商定的“哪个较新”，这是 VCS 的一个重要功能）。更先进的系统可确保一次提交的对文件集合的更改被视为单个单元（当逻辑更改涉及多个文件时具有原子性）。像 CVS（20 世纪 90 年代流行的 VCS）这样的系统没有这种提交原子性，因此容易发生损坏和丢失更改。确保原子性可以消除先前更改被无意覆盖的可能性，但需要跟踪上次同步到哪个版本。在提交时，如果自上次本地开发人员同步以来提交中的任何文件在头部被修改，则提交将被拒绝。特别是在这种跟踪更改的 VCS 中，开发人员的托管文件工作副本因此需要自己的元数据。根据 VCS 的设计，此存储库副本可以是存储库本身，也可以包含少量的元数据。这种精简的副本通常是“客户端”或“工作区”。

这看起来非常复杂：为什么需要 VCS？这种工具有什么特别之处，让它成为软件开发和软件工程中为数不多的几乎通用的工具之一？

想象一下没有 VCS 的工作。对于一个（非常）小的分布式开发人员团队来说，他们在有限范围的项目上工作，并且对版本控制一无所知，最简单、基础设施最低的解决方案就是来回传递项目的副本。当编辑不同时（人们在不同的时区工作，或者至少工作时间不同）时，这种方法最有效。

如果人们不知道哪个版本是最新的，我们就会立即遇到一个恼人的问题：跟踪哪个版本是最新的。任何尝试在非网络环境中进行协作的人都可能记得来回复制名为 Presentation v5 - nal - redlines - Josh's version v2 的文件的恐怖经历。正如我们将看到的，当没有一个一致认可的真相来源时，协作就会变得非常困难，并且容易出错。

引入共享存储需要稍微多一点的基础设施（访问共享存储），但提供了一个简单而明显的解决方案。在共享驱动器中协调工作可能在人数较少的情况下足以应付一段时间，但仍然需要带外协作以避免覆盖彼此的工作。此外，直接在共享存储中工作意味着任何不

---

<sup>2</sup>尽管对于什么是存储库、什么不是存储库的正式概念会根据您选择的 VCS 而略有不同，并且术语也会有所不同。

保持构建持续运行将开始妨碍团队中的每个人。如果我在您启动构建的同时对系统的某些部分进行更改,您的构建将无法工作。显然,这不能很好地扩展。

实际上,缺乏文件锁定和合并跟踪将不可避免地导致冲突和工作被覆盖。这样的系统很可能引入带外协调来决定谁在处理任何给定的文件。如果文件锁定被编码在软件中,我们已经开始重新发明早期的版本控制,如 RCS (等等)。当你意识到一次授予一个文件的写权限太粗粒度,并且你开始想要行级跟踪时,我们肯定会重新发明版本控制。我们似乎不可避免地需要某种结构化的机制来管理这些协作。因为我们似乎只是在这个假设中重新发明轮子,所以我们不妨使用现成的工具。

## 为什么版本控制很重要?

虽然版本控制现在几乎无处不在,但情况并非总是如此。最早的 VCS 可以追溯到 20 世纪 70 年代 (SCCS) 和 80 年代 (RCS),比第一次提到软件工程作为一门独立学科晚了很多年。团队参与“[多版本软件的多人开发](#)”当时业界还没有正式的版本控制概念。版本控制是为了应对数字协作的新挑战而发展起来的。经过数十年的演变和传播,版本控制的可靠、一致使用才发展成为如今的常态。<sup>3</sup>那么它是如何变得如此重要的?鉴于它似乎是一个不言而喻的解决方案,为什么有人会抵制 VCS 的想法?

回想一下,软件工程是随着时间的推移而集成的编程;我们在源代码的即时生成和随着时间的推移维护该产品的行为之间划清了界限(在维度上)。这一基本区别在很大程度上解释了 VCS 的重要性和犹豫:从最基本的层面上讲,版本控制是工程师管理原始源代码和时间之间相互作用的主要工具。我们可以将 VCS 概念化为一种扩展标准文件系统的方法。文件系统是从文件名到内容的映射。VCS 对其进行了扩展,以提供从(文件名、时间)到内容的映射,以及跟踪上次同步点和审计历史所需的数据。版本控制使时间考虑成为操作的一个明确部分:在程序中是不必要的。

---

<sup>3</sup>事实上,我曾经多次公开演讲,以“采用版本控制”作为典型例子,说明软件工程规范如何随着时间的推移而演变。以我的经验来看,在 20 世纪 90 年代,版本控制被广泛理解为最佳实践,但并未得到普遍遵循。在 21 世纪初,不采用版本控制的专业团体仍然很常见。如今,即使在从事个人项目的大学生中,Git 等工具的使用似乎也很普遍。采用率的上升可能部分归因于工具的用户体验更好(没有人想回到 RCS),但经验和规范的变化也发挥着重要作用。

ming 任务，在软件工程任务中至关重要。在大多数情况下，VCS 还允许对该映射进行额外输入（分支名称），以允许并行映射；因此：

VCS(文件名, 时间, 分支) => 文件内容

在默认用法中，该分支输入将具有通常理解的默认值：我们称之为“头部”、“默认”或“主干”来表示主分支。

对于是否持续使用版本控制的犹豫（轻微的）几乎直接来自于将编程和软件工程混为一谈。我们教授编程，培训程序员，面试工作时都基于编程问题和技术。即使是在像 Google 这样的地方，新员工对多人编写的代码或超过几周的代码几乎没有或根本没有经验，这完全是正常的。鉴于这种经验和对问题的理解，版本控制似乎是一个陌生的解决方案。版本控制正在解决我们的新员工不一定经历过的问题：“撤销”，不是针对单个文件，而是针对整个项目，这增加了许多复杂性，但有时好处并不明显。

在一些软件团队中，当管理层将技术人员的工作视为“软件开发”（坐下来编写代码）而不是“软件工程”（编写代码，使其在一段较长时间内保持工作和有用）时，也会出现同样的结果。由于编程的思维模型是主要任务，并且对代码和时间流逝之间的相互作用了解甚少，因此很容易将“返回上一个版本以撤消错误”描述为一种奇怪的、高开销的奢侈。

除了允许单独存储和引用不同时期的版本之外，版本控制还能帮助我们缩小单个开发人员和多个开发人员流程之间的差距。从实践角度来看，这就是版本控制对软件工程如此重要的原因，因为它使我们能够扩大团队和组织规模，尽管我们只是偶尔将其用作“撤消”按钮。开发本质上是一个分支合并的过程，无论是在不同时间点协调多个开发人员还是单个开发人员。VCS 消除了“哪个更新？”的问题。使用现代版本控制可以自动执行容易出错的操作，例如跟踪已应用哪些更改集。版本控制是我们协调多个开发人员和/或多个时间点的方式。

由于 VCS 已完全融入软件工程流程，甚至法律和监管实践也已跟上。VCS 允许对每行代码的每次更改进行正式记录，这对于满足审计要求越来越有必要。当内部开发与适当使用第三方来源相结合时，VCS 有助于追踪每行代码的出处和起源。

除了跟踪源代码随时间变化以及处理同步/分支/合并操作的技术和监管方面,版本控制还会触发一些非技术性的行为变化。提交到版本控制并生成提交日志的仪式会触发反思的时刻:自上次提交以来,您完成了什么?源代码是否处于您满意的状态?与提交、撰写摘要和标记任务完成相关的内省时刻可能对许多人来说本身就很有价值。提交过程的开始是检查清单、运行静态分析（参见第 20 章）、检查测试覆盖率、运行测试和动态分析等的最佳时机。

与任何过程一样,版本控制也需要一些开销:必须有人配置和管理您的版本控制系统,并且各个开发人员必须使用它。

但不要误会:这些几乎总是相当便宜。有趣的是,大多数经验丰富的软件工程师都会本能地对任何持续一两天以上的项目使用版本控制,即使是单个开发人员的项目。该结果的一致性表明,价值(包括风险降低)与开销之间的权衡一定相当容易。但我们承诺承认背景很重要,并鼓励工程领导者独立思考。考虑替代方案总是值得的,即使是在像版本控制这样基本的东西上。

事实上,很难想象任何可以被视为现代软件工程的任务不立即采用 VCS。既然您了解版本控制的价值和必要性,您现在可能会问您需要哪种类型的版本控制。

## 集中式 VCS 与分布式 VCS

从最简单的层面来说,所有现代 VCS 都是彼此等效的:只要您的系统具有原子提交对一批文件的更改的概念,其他一切都只是 UI。您可以使用另一个现代 VCS 和一堆简单的 shell 脚本构建与任何现代 VCS 相同的一般语义(而非工作流)。因此,关于哪个 VCS “更好”的争论主要是一个用户体验问题。核心功能是相同的,差异在于用户体验、命名、边缘情况功能和性能。选择 VCS 就像选择文件系统格式:在选择足够现代的格式时,差异相当小,而迄今为止更重要的问题是您在系统中填充的内容以及您使用它的方式。但是,VCS 中的主要架构差异会使配置、策略和扩展决策更容易或更困难,因此重要的是要注意重大架构差异,主要是集中式还是分散式之间的决定。

## 集中式 VCS

在集中式 VCS 实现中,模型是单个中央存储库之一 (可能存储在组织的某个共享计算资源上)。尽管开发人员可以在本地工作站上签出和访问文件,但与这些文件的版本控制状态交互的操作需要通过通信实现。

集中到中央服务器 (添加文件、同步、更新现有文件等)。开发人员提交的任何代码都会提交到该中央存储库中。第一个 VCS 实现都是集中式 VCS。

回顾 20 世纪 70 年代和 80 年代初期,我们发现最早的 VCS (例如 RCS) 专注于锁定和防止多个同时编辑。您可以复制存储库的内容,但如果想编辑文件,则可能需要获取 VCS 强制执行的锁定,以确保只有您在进行编辑。完成编辑后,即可释放锁定。当任何给定的更改都是快速完成的,或者很少有超过一个人在任何给定时间想要锁定文件时,该模型可以正常工作。调整配置文件等小编辑可以正常工作,在工作时间不连贯或很少长时间处理重叠文件的小团队中工作也可以正常工作。这种简单的锁定在规模方面存在固有问题:对于少数人来说,它可以正常工作,但如果任何这些锁发生争用,则可能会在更大的群体中崩溃。<sup>4</sup>

为了解决这一扩展问题,20 世纪 90 年代和 21 世纪初流行的 VCS 在更高层次上运作。这些更现代的集中式 VCS 避免了独占锁定,但会跟踪您已同步的更改,要求您的编辑基于提交中每个文件的最新版本。CVS 通过 (主要)一次对成批文件进行操作并允许多个开发人员同时签出一个文件来包装和改进 RCS:只要您的基本版本包含存储库中的所有更改,您就可以提交。Subversion 通过提供提交的真正原子性、版本跟踪和对不寻常操作 (重命名、使用符号链接等) 的更好的跟踪而进一步发展。集中式存储库/签出客户端模型今天在 Subversion 以及大多数商业 VCS 中继续使用。

## 分布式 VCS

从 2000 年代中期开始,许多流行的 VCS 都遵循分布式版本控制系统 (DVCS) 范式,例如 Git 和 Mercurial 等系统。

---

<sup>4</sup> 故事:为了说明这一点,我查找了 Google 员工对我最近一个项目中一个半热门文件所进行的未提交/未提交编辑的信息。在撰写本文时,有 27 项更改处于待处理状态,其中 12 项来自我的团队的成员,5 项来自相关团队的成员,10 项来自我从未见过的工程师。这基本上按预期工作。需要带外协调的技术系统或政策肯定无法扩展到分布式位置的 24/7 软件工程。

DVCS 与更传统的集中式 VCS (Subversion、CVS)之间的主要概念差异在于以下问题：“您可以在哪里提交？”或者“这些文件的哪些副本算作存储库？”

DVCS 世界不强制中央存储库的约束：如果您有存储库的副本（克隆、分叉），那么您就拥有一个可以提交的存储库以及查询有关修订历史等信息所需的所有元数据。标准工作流程是克隆一些现有存储库，进行一些编辑，在本地提交它们，然后将一些提交推送到另一个存储库，该存储库可能是也可能不是克隆的原始来源。任何中心化概念都纯粹是概念性的，是政策问题，而不是技术或底层协议的基础。

DVCS 模型允许更好的离线操作和协作，而无需固有地将某个特定的存储库声明为事实来源。一个存储库没有必要“领先”或“落后”，因为更改本身并不投射到线性时间轴中。然而，考虑到常见用法，集中式和 DVCS 模型在很大程度上可以互换：集中式 VCS 通过技术提供明确定义的中央存储库，而大多数 DVCS 生态系统都根据政策为项目定义中央存储库。也就是说，大多数 DVCS 项目都是围绕一个概念上的事实来源（例如 GitHub 上的特定存储库）构建的。DVCS 模型倾向于假设更分布式的用例，并且在开源世界中得到了特别广泛的采用。

一般而言，当今占主导地位的源代码控制系统是 Git，它实现了 DVCS。<sup>5</sup>如有疑问，请使用它——效仿其他人的做法有一定的价值。如果您的用例预计会有所不同，请收集一些数据并评估利弊。

Google 与 DVCS 的关系很复杂：我们的主要存储库基于（大规模）定制的内部集中式 VCS。我们定期尝试集成更多标准外部选项，并匹配我们的工程师（尤其是 Nooglers）对外部开发的期望工作流程。不幸的是，这些转向 Git 等更常见工具的尝试因代码库和用户群的庞大而受阻，更不用说海勒姆定律效应将我们与特定的 VCS 和该 VCS 的界面绑定在一起了。<sup>6</sup>这也许并不奇怪：大多数现有工具无法很好地适应 50,000 名工程师和数千万

---

<sup>5</sup> Stack Overflow 开发人员调查结果，2018 年。

<sup>6</sup> 单调递增的版本号（而不是提交哈希值）尤其麻烦。在 Google 开发者生态系统中成长起来的许多系统和脚本都假设提交的数字顺序与时间顺序相同——消除这些隐藏的依赖关系非常困难。

提交。<sup>7</sup> DVCS 模型通常（但并非总是）包括历史记录和元数据的传输，需要大量数据来启动存储库才能运行。

在我们的工作流程中，代码库的中心化和云存储似乎对扩展至关重要。DVCS 模型是围绕下载整个代码库并在本地访问的想法构建的。实际上，随着时间的推移和组织规模的扩大，任何特定的开发人员都将操作存储库中相对较小比例的文件，以及这些文件的一小部分版本。随着我们的成长（文件数量和工程师数量），这种传输几乎完全是浪费。大多数文件的唯一本地化需求发生在构建时，但分布式（和可重现的）构建系统似乎也更适合该任务（参见第 18 章）。

## 真相之源

集中式 VCS（Subversion、CVS、Perforce 等）将真实来源概念融入系统设计中：最近在主干上提交的内容是当前版本。当开发人员去签出项目时，默认情况下，他们会看到主干版本。当您的更改在该版本上重新提交时，您的更改即“完成”。

然而，与集中式 VCS 不同，DVCS 系统中没有固有的概念，即分布式存储库的哪个副本是唯一的事实来源。理论上，可以在没有集中或协调的情况下传递提交标记和 PR，从而允许不同的开发分支不受控制地传播，从而冒着概念回归 Presentation v5 - nal - redlines - Josh 的版本 v2 的风险。因此，与集中式 VCS 相比，DVCS 需要更明确的政策和规范。

使用 DVCS 进行良好管理的项目会将特定存储库中的特定分支声明为真实来源，从而避免出现更多混乱的可能性。我们在实践中看到了这一点，托管式 DVCS 解决方案（如 GitHub 或 GitLab）的传播 用户可以克隆和分叉项目的存储库，但仍然只有一个主存储库：当项目位于该存储库的主干分支中时，它们就“完成”了。

即使在 DVCS 世界中，集中化和真相来源也重新被使用，这并非偶然。为了帮助说明真相来源这一理念的重要性，让我们想象一下，当我们没有明确的真相来源时会发生什么。

---

<sup>7</sup>事实上，截至 Monorepo 论文发表之时，存储库本身大约有 86 TB 的数据和元数据（不考虑发布分支）。直接将其安装到开发人员工作站上将是……具有挑战性的。

### 场景:没有明确的事实来源

想象一下,您的团队充分遵守 DVCS 理念,避免将特定的分支+存储库定义为最终的真相来源。

在某些方面,这让人想起了 Presentation v5 - nal - redlines - Josh 的版本 v2 模型 从队友的存储库中提取后,不一定清楚哪些更改存在,哪些不存在。在某些方面,它比这更好,因为 DVCS 模型以比那些临时命名方案更精细的粒度跟踪各个补丁的合并,但 DVCS 知道哪些更改被合并与每个工程师确保他们拥有所有过去/相关的更改之间存在差异。

考虑一下如何确保发布版本包含每个开发人员在过去几周开发的所有功能。有哪些(非集中式、可扩展的)机制可以做到这一点?我们能否设计出比每个人都签字更好的政策?随着团队规模的扩大,是否有只需要次线性人力投入的政策?随着团队开发人员数量的增加,这种方法还能继续发挥作用吗?就我们所知:可能不会。如果没有一个集中式事实来源,就会有人保留一份可能准备好包含在下一个版本中的功能列表。最终,这种记账方式将重现集中式事实来源的模型。

进一步想象一下:当一个新的开发人员加入团队时,他们从哪里获得最新的、已知良好的代码副本?

DVCS 支持许多出色的工作流程和有趣的使用模型。但如果您担心随着团队的壮大,系统需要花费线性人力来管理,那么将一个存储库(和一个分支)定义为最终的真相来源就非常重要。

事实来源具有一定的相对性。也就是说,对于给定项目,不同组织的真相来源可能不同。这个警告很重要:Google 或 RedHat 的工程师对 Linux 内核补丁使用不同的事实来源是合理的,这与 Linus (Linux 内核维护者)本人的做法仍然不同。当组织及其事实来源是层级结构的(并且对组织外部的人来说是不可见的)时,DVCS 可以正常工作 - 这也许是 DVCS 模型最实用的效果。RedHat 工程师可以提交到本地事实来源存储库,并且可以定期从那里向上游推送更改,而 Linus 对事实来源的概念完全不同。只要对于应该将更改推送到何处没有选择或不确定性,我们就可以避免 DVCS 模型中一大类混乱的扩展问题。

在所有这些思考中,我们为主干分支赋予了特殊意义。但当然,VCS 中的“主干”只是技术默认设置,组织可以

在此基础上选择不同的策略。也许默认分支已被放弃,所有工作实际上都发生在某个自定义开发分支上 除了需要在更多操作中提供分支名称外,这种方法本身并没有什么问题;它只是非标准的。在讨论版本控制时,有一个(经常不言而喻的)事实:对于任何给定的组织来说,技术只是其中的一部分;在此基础上几乎总是有同等数量的政策和使用惯例。

在版本控制中,没有哪个主题比讨论如何使用和管理分支更具策略和惯例。我们将在下一节中更详细地讨论分支管理。

版本控制与依赖管理在讨论版本控制策略和依赖管理时,有很多概念上的相似

之处(参见第21章)。两者的区别主要体现在两个方面:VCS策略主要涉及如何管理自己的代码,而且通常粒度更细。依赖管理更具挑战性,因为我们主要关注由其他组织管理和控制的项目,粒度更高,而这些情况意味着您无法完全控制。

我们将在本书后面讨论更多这些高级问题。

## 分支机构管理

能够在版本控制中跟踪不同的修订版本,为管理这些不同版本开辟了各种不同的方法。总的来说,这些不同的方法属于分支管理,与单个“主干”相对。

正在进行的工作类似于分支任何组织关于分支管理政策

的讨论都应该至少承认,组织中每一项正在进行的工作都相当于一个分支。在DVCS中情况更明显,开发人员更有可能在将代码推送回上游事实来源之前进行大量本地暂存提交。对于集中式VCS来说仍然如此:未提交的本地更改在概念上与分支上已提交的更改没有什么不同,只是可能更难找到和进行比较。一些集中式系统甚至明确了这一点。例如,当使用Perforce时,每个更改都会被赋予两个修订号:一个表示创建更改的隐式分支点,另一个表示重新提交更改的位置,如图16-1所示。Perforce用户可以查询谁对给定文件有未完成的更改,检查其他用户未提交的更改中的待处理更改等等。

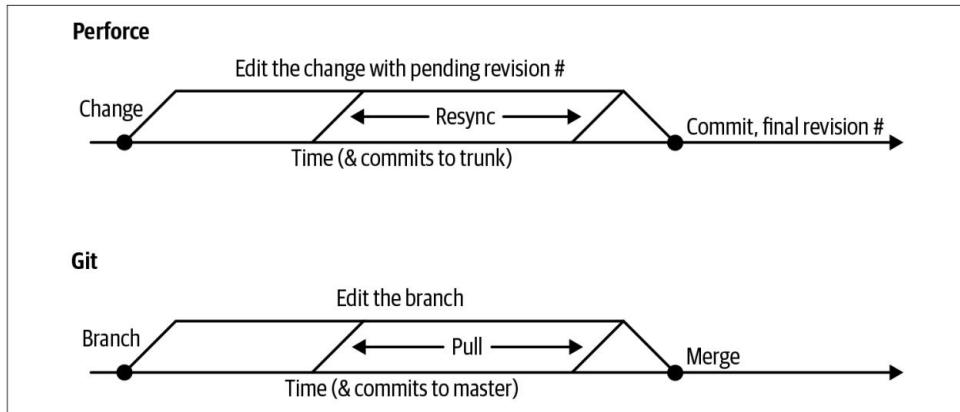


图 16-1. Perforce 中的两个修订版本号

这种“未提交的工作类似于分支”的想法在考虑重构任务时尤其重要。想象一下，有人告诉开发人员，“将 Widget 重命名为 OldWidget”。根据组织的分支管理政策和理解，什么算作分支，哪些分支很重要，这可能会有几种解释：

- 在 Source of Truth 存储库的主干分支上重命名 Widget
- 在 Source of Truth 存储库的所有分支上重命名 Widget
- 在 Source of Truth 存储库的所有分支上重命名 Widget，并查找所有对引用 Widget 的文件有未完成更改的开发人员

如果我们进行推测，尝试支持“在任何地方重命名，即使在未完成的更改中”用例是商业集中式 VCS 倾向于跟踪诸如“哪些工程师打开此文件进行编辑？”之类的内容的部分原因。（我们不认为这是一种执行重构任务的可扩展方式，但我们理解这种观点。）

## 开发分支

在一致性单元测试出现之前（见第 11 章），当引入任何特定更改都存在导致系统其他功能退化的风险时，对主干进行特殊处理是有意义的。您的技术主管可能会说：“我们不会提交主干，直到新更改经过一轮完整的测试。我们的团队改用特定于功能的开发分支。”

开发分支（通常为“dev 分支”）是“已完成但未提交”和“新工作基于此”之间的中间点。这些分支试图解决的问题（产品不稳定）是合理的，但

我们发现,通过更广泛地使用测试、持续集成 (CI) (参见第 23 章) 和彻底的代码审查等质量执行实践,可以更好地解决问题。

我们认为,大量使用开发分支作为实现产品稳定性的手段的版本控制策略本质上是错误的。同一组提交最终将合并到主干。小型合并比大型合并更容易。由编写这些更改的工程师进行的合并比批量处理不相关的更改并稍后合并更容易 (如果团队共享开发分支,最终会发生这种情况)。如果合并后的提交前测试发现了任何新问题,则同样的论点也适用:如果只有一名工程师参与,则更容易确定谁的更改导致了回归。合并大型开发分支意味着在该测试运行中发生了更多更改,这使得故障更难隔离。分类和找出问题的根源很困难;修复它就更糟了。

除了缺乏专业知识和合并单个分支的固有问题之外,依赖开发分支还存在很大的扩展风险。这对于软件组织来说是一种非常常见的生产力损失。当有多个分支长期处于孤立状态时,协调合并操作的成本会比基于主干的开发高得多(并且可能更具风险)。

### 我们是怎么对开发分支上瘾的?

很容易看出组织是如何陷入这个陷阱的:他们看到“合并这个长期存在的开发分支会降低稳定性”,并得出结论“分支合并是有风险的”。

他们没有用“更好的测试”和“不要使用基于分支的开发策略”来解决这个问题,而是专注于减缓和协调症状:分支合并。团队开始根据其他正在进行的分支开发新的分支。在长期存在的开发分支上工作的团队可能会也可能不会定期将该分支与主开发分支同步。随着组织规模的扩大,开发分支的数量也会增加,并且需要投入更多的精力来协调该分支合并策略。他们投入越来越多的精力来协调分支合并。这项任务本质上无法扩展。一些不幸的工程师成为构建主管/合并协调员/内容管理工程师,专注于充当单点协调员来合并组织中所有不同的分支。定期安排的会议试图确保组织已经“制定出本周的合并策略”。<sup>8</sup>未被选中进行合并的团队通常需要在每次大型合并之后重新同步和重新测试。

---

<sup>8</sup>最近的非正式 Twitter 民意调查显示,大约 25% 的软件工程师受到过“定期计划中的”合并战略会议。

合并和重新测试的所有努力都是纯粹的开销。替代方案需要不同的范例:基于主干的开发,严重依赖测试和 CI,保持构建绿色,并在运行时禁用不完整/未经测试的功能。每个人都负责同步到主干并提交;没有“合并策略”会议,没有大型/昂贵的合并。并且,不要激烈讨论应该使用哪个版本的库 只能有一个。必须有一个单一的真相来源。最终,将有一个用于发布的单一修订版:缩小到单一真相来源只是“左移”方法,用于识别包含和不包含的内容。

## 发布分支

如果产品的发布间隔 (或发布生命周期)超过几个小时,则创建一个发布分支来代表产品发布版本的确切代码可能是明智之举。如果在该产品实际发布到公众面前和下一个发布周期之间发现任何严重缺陷,则可以从主干中挑选修复程序 (最小限度、有针对性的合并)到您的发布分支。

与开发分支相比,发布分支通常比较温和:麻烦的不是分支的技术,而是使用方式。开发分支和发布分支之间的主要区别在于预期的最终状态:开发分支预计会合并回主干,甚至可能被另一个团队进一步分支。发布分支预计最终会被放弃。

在 Google 的 DevOps 研究与评估 (DORA) 组织确定的运作最完善的技术组织中,发布分支几乎不存在。已实现持续部署 (CD) (每天多次从主干发布的能力)的组织可能倾向于跳过发布分支:只需添加修复并重新部署就容易得多。因此,精选和分支似乎是不必要的开销。显然,这更适用于以数字方式部署的组织 (例如 Web 服务和应用程序),而不是向客户推送任何形式的有形发布的组织;确切了解已向客户推送的内容通常很有价值。

同一项 DORA 研究还表明,“基于主干的开发”、“没有长期存在的开发分支”与良好的技术成果之间存在很强的正相关性。这两个想法背后的思想似乎很明确:分支会拖累生产力。在许多情况下,我们认为复杂的分支和合并策略是一种安全支柱 一种保持主干稳定的尝试。正如我们在本书中看到的那样,还有其他方法可以实现这一结果。

## Google 的版本控制

在 Google,我们绝大部分源代码都存放在一个存储库 (mono-repo) 中,大约有 50,000 名工程师共享该存储库。Google 旗下的几乎所有项目都存放在那里,除了 Chromium 和 Android 等大型开源项目。

这包括面向公众的产品,如搜索、Gmail、我们的广告产品、我们的 Google 云平台产品,以及支持和开发所有这些产品所需的内部基础设施。

我们依靠内部开发的名为 Piper 的集中式 VCS,它在我们的生产环境中作为分布式微服务运行。这使我们能够使用 Google 标准存储、通信和计算即服务技术来提供全球可用的 VCS,存储超过 80 TB 的内容和元数据。

然后,每天有数千名工程师同时编辑和提交 Piper monorepo。在人工和利用版本控制 (或改进签入 VCS 的内容) 的半自动化流程之间,我们每个工作日通常会处理 60,000 到 70,000 次提交到存储库。二进制文件相当常见,因为不会传输完整的存储库,因此二进制文件的正常成本并不适用。由于从最早的构想开始就专注于 Google 规模,因此这个 VCS 生态系统中的操作在人力规模上仍然很便宜:在主干上创建一个新的客户端、添加一个文件并向 Piper 提交一个 (未审核的) 更改可能总共需要 15 秒。这种低延迟交互和易于理解/设计良好的扩展简化了许多开发人员的体验。

由于 Piper 是内部产品,因此我们能够对其进行自定义并执行我们选择的任何源代码控制策略。例如,我们在 monorepo 中有一个细粒度所有权的概念:在文件层次结构的每个级别,我们都可以找到 OWNERS 文件,其中列出了允许批准存储库子树内提交的工程师的用户名 (除了在树中更高级别列出的 OWNERS)。在具有许多存储库的环境中,这可能通过拥有单独的存储库来实现,这些存储库具有文件系统权限强制控制提交访问,或者通过 Git “提交钩子”(提交时触发的操作)进行单独的权限检查。通过控制 VCS,我们可以使所有权和批准的概念更加明确,并在尝试提交操作期间由 VCS 强制执行。该模型也很灵活:所有权只是一个文本文件,与存储库的物理分离无关,因此在团队转移或组织重组后更新它很简单。

## 一个版本

仅凭 Piper 的惊人扩展能力无法实现我们所依赖的那种协作。正如我们之前所说:版本控制也与政策有关。除了我们的 VCS,Google 版本控制政策的一个关键功能就是我们所说的“一个版本”。这扩展了我们

之前提到的“确保开发人员知道哪个分支和存储库是他们的可信来源”这一原则被修改为“对于存储库中的每个依赖项,只能选择一个版本”。<sup>9</sup>对于第三方软件包,这意味着在稳定状态下,该软件包只能有一个版本签入我们的存储库。<sup>10</sup>对于内部软件包,这意味着如果不重新打包/重命名,就不能进行分叉:在技术上必须安全地将原始软件包和分叉软件包混合到同一个项目中,而无需付出特殊努力。这对于我们的生态系统来说是一项强大的功能:很少有软件包带有这样的限制:“如果包含此软件包(A),则不能包含其他软件包(B)”。

将单个存储库中单个分支上的单个副本作为我们的“事实来源”这一概念是直观的,但在应用中也有一些微妙的深度。让我们研究一下这样一个场景:我们有一个 monorepo (因此可以说已经履行了“单一事实来源”的法律条文),但允许我们的库的分支在主干上传播。

场景:多个可用版本想象一下以下场景:某个团队在通用

基础架构代码(在我们的例子中是 Abseil 或 Guava 等)中发现了一个错误。团队决定不就地修复它,而是分叉该基础架构并对其进行调整以解决该错误而不重命名库或符号。它会通知他们附近的其他团队,“嘿,我们这里有一个改进版本的 Abseil:请查看。”其他一些团队构建了依赖这个新分叉的库。

正如我们将在第 21 章中看到的,我们现在处于一个危险的境地。如果代码库中的任何项目同时依赖于 Abseil 的原始版本和分叉版本,那么在最好的情况下,构建会失败。在最坏的情况下,我们将面临难以理解的运行时错误,这些错误源于链接同一个库的两个不匹配版本。“分叉”实际上为代码库添加了一个着色/分区属性:任何给定目标的传递依赖项集都必须包含此库的一个副本。从代码库的“原始风格”分区添加到“新分叉”分区的任何链接都可能会破坏一切。这意味着最终像“添加新依赖项”这样简单的事情变成了一项操作,可能需要对整个代码库运行所有测试,以确保我们没有违反这些分区要求之一。这很昂贵、不幸,并且扩展性不佳。

---

<sup>9</sup>例如,在升级操作期间,可能会签入两个版本,但如果开发人员在现有包上添加新的依赖项,不应选择依赖哪个版本。

<sup>10</sup>话虽如此,我们在许多情况下都失败了,因为外部软件包有时会将自己的依赖项的副本固定在其源版本中。您可以在第 21 章中详细了解这一切是如何出错的。

在某些情况下,我们也许能够以某种方式将一些东西拼凑在一起,使生成的可执行文件能够正常运行。例如,Java有一个相对标准的做法,称为着色,它会调整库内部依赖项的名称,以向应用程序的其余部分隐藏这些依赖项。处理函数时,这在技术上是可行的,即使理论上有点像 hack。处理可以从一个包传递到另一个包的类型时,遮蔽解决方案在理论上和实践上都行不通。据我们所知,任何允许多个独立版本的库在同一个二进制文件中运行的技术技巧都有这个限制:该方法适用于函数,但没有好的(有效的)解决方案来遮蔽类型。任何提供词汇表类型(或任何更高级别构造)的库的多个版本都会失败。遮蔽和相关方法正在修补底层问题:需要相同依赖项的多个版本。(我们将在第 21 章讨论如何尽量减少这种情况。)

任何允许在同一代码库中使用多个版本的政策系统都存在这些代价高昂的不兼容问题。您可能会暂时摆脱这种情况(我们确实有一些违反此政策的小问题),但一般而言,任任何多版本情况都有可能导致大问题。

### “单一版本”规则

记住这个例子,在单一事实来源模型的基础上,我们希望能够充分理解这个看似简单的源代码控制和分支管理规则的深度:

开发人员绝不能选择“我应该依赖这个组件的那个版本?”

通俗地说,这有点像“单一版本规则”。在实践中,“单一版本”并不是一成不变的,<sup>11</sup>但围绕限制添加新依赖项时可以选择的版本来表述这一点,传达了一种非常有力的理解。

对于个人开发者来说,缺乏选择似乎是一种任意的障碍。

然而,我们一次又一次地看到,对于一个组织来说,一致性是高效扩展的关键要素。一致性对于组织的各个层面都具有深远的重要性。

从某个角度来看,这是关于一致性以及确保能够利用一致性“瓶颈”的讨论的直接副作用。

---

<sup>11</sup>例如,如果有定期更新的外部/第三方库,则可能无法通过一次原子更改来更新该库并更新对其的所有使用。因此,通常需要添加该库的新版本,防止新用户添加对旧库的依赖,并逐步将使用情况从旧库切换到新库。

(几乎)没有长寿命分支我们的“单一版本规则”

隐含着一些更深层次的想法和政策;其中最重要的是:开发分支应该最小化,或者最好是短命的。这源于过去20年发表的大量研究成果,从敏捷流程到DORA关于基于主干的开发的研究成果,甚至凤凰项目12关于“减少在制品”的经验教训。当我们看待处理工作视为类似于开发分支的想法时,这进一步强调了应该以小增量的方式针对主干进行工作,并定期提交。

举一个反例:在一个严重依赖长期开发分支的开发社区中,不难想象选择的机会会悄悄回归。

想象一下这样的场景:某个基础设施团队正在开发一个比旧Widget更好的新Widget。兴奋感与日俱增。其他新启动的项目会问:“我们可以依赖您的新Widget吗?”显然,如果您已经投资了代码库可见性策略,则可以解决这个问题,但当新Widget被“允许”但仅存在于并行分支中时,就会出现深层问题。请记住:新开发在添加依赖项时不能有选择。应该将新Widget提交到主干,在准备好之前从运行时禁用,并且如果可能的话,通过可见性对其他开发人员隐藏 或者应该将两个Widget选项设计为可以共存,链接到同一个程序中。

有趣的是,已经有证据表明这在行业中很重要。在Accelerate和最新的DevOps报告中,DORA指出,基于主干的开发与高绩效软件组织之间存在预测关系。谷歌并不是唯一发现这一点的组织 当这些政策制定时,我们也不一定想到了预期的结果 似乎其他方法都不起作用。DORA的结果肯定符合我们的经验。

我们针对大规模变更(LSC;参见第22章)的政策和工具更加强调了基于主干的开发的重要性:在修改签入到主干分支的所有内容时,在整个代码库中应用的广泛/浅显的变更已经是一项艰巨(通常乏味)的任务。如果有无限数量的额外开发分支可能需要同时重构,那么执行这些类型的变更将是一项非常繁重的任务,因为要找到一组不断扩大的隐藏分支。在DVCS模型中,甚至可能无法识别所有这些分支。

---

12 Kevin Behr、Gene Kim 和 George Spafford, 《凤凰计划》(波特兰:IT Revolution Press,2018年)。

当然,我们的经验并不具有普遍性。你可能会发现自己处于一些特殊情况,需要与主干并行 (并定期合并)的寿命更长的开发分支。

这些情况应该很少见,而且应该理解为代价高昂。在 Google monorepo 中工作的大约 1,000 个团队中,只有几个拥有这样的开发分支。<sup>13</sup>通常,这些存在的原因非常具体 (而且非常不寻常)。大多数原因归结为“我们有一个不寻常的

长期兼容性要求。”通常,这是为了确保静态数据在各个版本之间的兼容性:某种文件格式的读取器和写入器需要随着时间的推移就该格式达成一致,即使读取器或写入器实现发生了修改。其他时候,长期存在的开发分支可能来自于承诺 API 长期兼容性 - 当一个版本不够用时,我们需要承诺旧版本的微服务客户端仍然可以与较新的服务器一起使用 (反之亦然)。这可能是一个非常具有挑战性的要求,对于积极发展的 API,你不应该轻易承诺,并且你应该谨慎对待以确保这段时间不会意外开始增长。任何形式的跨时间依赖性都比时间不变的代码昂贵和复杂得多。

在内部,Google 生产服务很少做出此类承诺。<sup>14</sup>我们还从“构建范围”对潜在版本偏差的限制中受益匪浅:生产中的每项工作最多每六个月需要重建和重新部署一次。(通常频率要高得多。)

我们确信还有其他情况可能需要长期存在的开发分支。

只要确保它们很少出现即可。如果你采用本书中讨论的其他工具和实践,许多工具和实践都会对长期存在的开发分支施加压力。在主干上运行良好但在开发分支上失败 (或需要更多努力) 的自动化和工具可以帮助鼓励开发人员保持最新状态。

## 那么发布分支呢?

许多 Google 团队使用发布分支,但选择性有限。如果您打算每月发布一次版本,并继续为下一个版本而努力,那么创建发布分支是完全合理的。同样,如果您要将设备发送给客户,那么确切了解“现场”发布的版本也很重要。请谨慎并保持理性,将选择性保持在最低限度,并且不要计划与主干重新合并。鉴于很少有团队能够达到 CD 所承诺的快速发布节奏 (参见第 24 章),从而消除对发布分支的需求或渴望,因此我们的各个团队对发布分支都有各种政策。一般而言,

---

<sup>13</sup>很难得到准确的数字,但这样的球队数量几乎肯定少于 10 支。

<sup>14</sup>云接口则是另一回事。

根据我们的经验,发布分支不会造成任何大范围的成本。或者,至少,除了 VCS 的额外固有成本之外,没有明显的成本。

## Monorepos

2016 年,我们发表了一篇关于 Google monorepo 方法的论文 (被广泛引用和讨论)。<sup>15</sup> monorepo 方法具有一些固有的优势,其中最主要的是遵守“一个版本”很简单:违反“一个版本”通常比做正确的事情更困难。它没有决定任何东西的哪些版本是官方的,也没有发现哪些存储库是重要的过程。构建工具以了解构建状态 (参见第 23 章)也不需要发现重要的存储库存在的位置。一致性有助于扩大引入新工具和优化的影响。总的来说,工程师可以看到其他人都在做什么,并以此为指导自己在代码和系统设计方面的选择。这些都是非常好的事情。

考虑到所有这些以及我们对单一版本规则优点的信念,我们有理由问一问,单一仓库是否是唯一正确的方法。相比之下,开源社区似乎可以很好地使用“多仓库”方法,该方法建立在看似无限数量的非协调和非同步项目存储库上。

简而言之:不,我们认为我们所描述的 monorepo 方法对每个人来说都是完美的答案。继续比较文件系统格式和 VCS,很容易想象决定是使用 10 个驱动器来提供一个非常大的逻辑文件系统还是使用 10 个单独访问的较小文件系统。在文件系统世界中,两者各有利弊。评估文件系统选择时的技术问题包括中断恢复能力、大小限制、性能特征等。

可用性问题可能更多地集中在跨文件系统边界引用文件、添加符号链接和同步文件的能力上。

一组非常类似的问题决定了是选择单一存储库还是更细粒度的存储库集合。关于如何存储源代码 (或存储文件)的具体决定很容易引起争议,在某些情况下,组织和工作流程的细节比其他方面更重要。

这些都是你需要自己做出的决定。

重要的不是我们是否专注于 monorepo;而是尽可能地遵守单一版本原则:开发人员在向组织中已在使用的某个库添加依赖项时必须没有选择。违反单一版本规则的选择会导致合并策略讨论、钻石依赖、工作损失和精力浪费。

---

<sup>15</sup> Rachel Potvin 和 Josh Levenberg, “谷歌为何将数十亿行代码存储在一个存储库中”, ACM 通讯, 59 No. 7 (2016 年) :78-87。

软件工程工具（包括 VCS 和构建系统）越来越多地提供机制，以巧妙地融合细粒度存储库和单一存储库，从而提供类似于单一存储库的体验。对提交的顺序达成一致，并了解依赖关系图。Git 子模块、具有外部依赖项的 Bazel 和 CMake 子项目都允许现代开发人员合成一些弱近似于单一存储库行为的东西，而无需单一存储库的成本和缺点。<sup>16</sup>例如，细粒度存储库在规模（Git 在几百万次提交后通常会出现性能问题，并且当存储库包含大型二进制文件时克隆速度往往很慢）和存储（VCS 元数据可能会累积，尤其是当您的版本控制系统中有二进制文件时）方面更易于处理。联合/虚拟单一存储库（VMR）样式存储库中的细粒度存储库可以更容易地隔离实验性或绝密项目，同时仍保留一个版本并允许访问常用实用程序。

换句话说：如果您组织中的每个项目都有相同的保密性、法律、隐私和安全要求，那么真正的 monorepo 是一个不错的选择。

否则，以 monorepo 的功能为目标，但允许自己灵活地以不同的方式实现该体验。如果您可以使用分离的存储库进行管理并遵守一个版本，或者您的工作负载完全断开连接以允许真正独立的存储库，那就太好了。否则，以某种方式合成像 VMR 这样的东西可能代表两全其美。

毕竟，你选择的文件系统格式并不重要，重要的是你写入它。

## 版本控制的未来

Google 并不是唯一一家公开讨论 monorepo 方法好处的组织。Microsoft、Facebook、Netflix 和 Uber 也公开提到了他们对该方法的依赖。DORA 已广泛发表了有关它的文章。所有这些成功的、长寿的公司都可能被误导了，或者至少它们的情况足够不同，以至于不适用于一般的小型组织。虽然有可能，但我们认为可能性不大。

大多数反对 monorepos 的论点都集中在拥有单个大型存储库的技术限制上。如果从上游克隆存储库既快速又便宜，开发人员更有可能将更改保持在较小且独立的范围内（并避免

---

<sup>16</sup>我们认为我们还没有看到任何事情能够特别顺利地做到这一点，但是存储库间依赖/虚拟单一存储库的想法显然已经存在。

<sup>17</sup>或者您有意愿和能力定制您的 VCS，并在您的代码库/组织的整个生命周期内维护该定制。不过，也许不要将此作为一种选择；这会花费很多开销。

提交到错误的正在进行的分支会导致错误。如果克隆存储库（或执行其他常见的 VCS 操作）会浪费开发人员的数小时时间，那么您可以很容易地理解为什么组织会回避对如此大型的存储库/操作的依赖。幸运的是，我们通过专注于提供可大规模扩展的 VCS 避免了这个陷阱。

回顾过去几年 Git 的重大改进，显然我们做了很多工作来支持更大的存储库：浅克隆、稀疏分支、更好的优化等等。我们预计这种情况会继续下去，而“但我们需要保持存储库较小”的重要性会逐渐减弱。

反对 monorepos 的另一个主要论点是，它与开源软件（OSS）世界中的开发方式不符。虽然确实如此，但 OSS 世界中的许多实践（正确地）源于优先考虑自由、缺乏协调和缺乏计算资源。OSS 世界中的独立项目实际上是独立的组织，碰巧能够看到彼此的代码。在组织的边界内，我们可以做出更多假设：我们可以假设计算资源的可用性，我们可以假设协调，我们可以假设存在一定程度的集中权限。

关于 monorepo 方法的一个不太常见但可能更合理的担忧是，随着组织规模的扩大，每段代码都遵守完全相同的法律、合规性、监管、保密和隐私要求的可能性越来越小。manyrepo 方法的一个固有优势是，单独的存储库显然能够拥有不同的授权开发人员、可见性、权限等。将该功能整合到 monorepo 中是可以做到的，但在定制和维护方面需要一些持续的持有成本。

与此同时，业界似乎在一次又一次地发明轻量级的存储库间链接。有时，这是在 VCS（Git 子模块）或构建系统中。只要一组存储库对“什么是主干”、“哪个更改先发生”以及描述依赖关系的机制有一致的理解，我们就可以轻松地想象将一组不同的物理存储库拼接成一个更大的 VMR。尽管 Piper 为我们做得很好，但投资于高度可扩展的 VMR 和工具来管理它并依靠现成的定制来满足每个存储库的策略要求可能是一项更好的投资。

一旦有人在 OSS 社区中构建了足够大的兼容且相互依赖的项目，并发布了这些软件包的 VMR 视图，我们怀疑 OSS 开发人员的做法将开始改变。我们在可以合成虚拟 monorepo 的工具中看到了这一点，以及（例如）大型 Linux 发行版所做的工作，这些发行版发现并发布了数千个软件包的相互兼容的修订版。借助单元测试、CI 和自动版本升级，这些修订版的新提交使软件包所有者能够

更新其软件包的主干（当然，以不间断的方式），我们认为该模型将在开源世界中流行起来。毕竟，这只是一个效率问题：采用单一版本规则的（虚拟）单一存储库方法将软件开发的复杂性降低到一个（困难的）维度：时间。

我们预计版本控制和依赖管理将在未来 10 到 20 年内朝这个方向发展：VCS 将专注于允许更大的存储库具有更好的性能扩展，同时也通过提供更好的机制将更大的存储库跨项目和组织边界拼接在一起，从而消除对更大存储库的需求。某些人（也许是现有的包管理组或 Linux 分销商）将催化事实上的标准虚拟单一存储库。根据该单一存储库中的实用程序，可以轻松访问一组兼容的依赖项作为一个单元。我们将更普遍地认识到版本号是时间戳，允许版本偏差会增加维度复杂性（时间），这会带来很多成本，而我们可以学会避免这种情况。它从逻辑上像单一存储库这样的东西开始。

## 结论

版本控制系统是技术（尤其是共享计算资源和计算机网络）带来的协作挑战和机遇的自然延伸。从历史上看，它们的发展与我们当时理解的软件工程规范步调一致。

早期系统提供简单的文件粒度锁定。随着典型的软件工程项目和团队规模不断扩大，这种方法的扩展问题变得明显，我们对版本控制的理解也随之改变以应对这些挑战。然后，随着开发越来越多地转向具有分布式贡献者的 OSS 模型，VCS 变得更加分散。我们预计 VCS 技术将发生转变，假设网络可用性不变，更加注重存储和在云中构建，以避免传输不必要的文件和工作。这对于大型、长期的软件工程项目来说越来越重要，即使这意味着与简单的单开发 / 单机编程项目相比方法有所改变。这种向云端的转变将使 DVCS 方法的出现具体化：即使我们允许分布式开发，仍然必须集中承认某些东西是真相的来源。

当前的 DVCS 去中心化是该技术对行业（尤其是开源社区）需求的明智反应。但是，DVCS 配置需要严格控制，并与适合您组织的分支管理策略相结合。它还经常会引入意外的扩展问题：完美保真离线操作需要更多的本地数据。无法控制分支混战的潜在复杂性可能会导致开发人员和代码部署之间产生无限的开销。但是，复杂的技术不需要以复杂的方式使用：

我们在 monorepo 和基于主干的开发模型中看到,保持分支策略简单通常会带来更好的工程结果。

选择会导致成本。我们高度赞同此处提出的单一版本规则:组织内的开发人员不得选择提交到哪里,或依赖现有组件的那个版本。据我们所知,很少有政策会对组织产生如此大的影响:虽然这可能会让个别开发人员感到烦恼,但从总体上看,最终结果要好得多。

## TL;DR

- 对于任何大于“只有一个开发人员的玩具项目,永远不会更新”的软件开发项目,请使用版本控制。
- 当选择“哪个版本”时,存在固有的扩展问题

我应该依赖这个吗?

- 单一版本规则对于组织效率而言极其重要。  
消除在何处提交或依赖什么的选择可以带来显著的简化。
- 在某些语言中,您可能需要花费一些精力通过诸如阴影、单独编译、链接器隐藏等技术方法来避免这种情况。  
让这些方法发挥作用的工作完全是浪费劳动力 您的软件工程师没有生产任何东西,他们只是在解决技术债务。· 先前的研究 (DORA/State of DevOps/Accelerate)表明,基于主干的开发是高绩效开发组织的预测因素。长期存在的开发分支并不是一个好的默认计划。· 使用对您有意义的任何版本控制系统。如果您的组织希望为不同的项目优先考虑不同的存储库,那么将存储库间依赖关系取消固定/“置于头部” / “基于主干”可能仍然是明智的做法。

VCS 和构建系统设施的数量不断增加,允许您拥有小型、细粒度的存储库以及整个组织的一致的“虚拟”头/主干概念。