

O'REILLY®

软件 工程 谷歌

从编程中学到的经

验教训

随着时间的推移



由 Titus Winters 策划,
汤姆·曼什雷克 & 海伦·赖特



Software Engineering at Google

Today, software engineers need to know not only how to program effectively but also how to develop proper engineering practices to make their codebase sustainable and healthy. This book emphasizes this difference between programming and software engineering.

How can software engineers manage a living codebase that evolves and responds to changing requirements and demands over the length of its life? Based on their experience at Google, software engineers Titus Winters and Hyrum Wright, along with technical writer Tom Mansreck, present a candid and insightful look at how some of the world's leading practitioners construct and maintain software. This book covers Google's unique engineering culture, processes, and tools and how these aspects contribute to the effectiveness of an engineering organization.

You'll explore three fundamental principles that software organizations should keep in mind when designing, architecting, writing, and maintaining code:

- How *time* affects the sustainability of software and how to make your code resilient over time
- How *scale* affects the viability of software practices within an engineering organization
- What *trade-offs* a typical engineer needs to make when evaluating design and development decisions

"While being upfront about trade-offs, this book explains the Google way of doing software engineering, which makes me most productive and happy."

—Eric Haugh
Software Engineer at Google

Titus Winters, a senior staff software engineer at Google, is the library lead for Google's C++ codebase: 250 million lines of code edited by thousands of distinct engineers per month.

Tom Mansreck is a staff technical writer within Software Engineering at Google. He's a member of the C++ Library Team, developing documentation, launching training classes, and documenting Abseil, Google's open source C++ code.

Hyrum Wright is a staff software engineer at Google, where he leads Google's automated change tooling group. Hyrum has made more individual edits to Google's codebase than any other engineer in the history of the company.

SOFTWARE ENGINEERING

US \$59.99

CAN \$79.99

ISBN: 978-1-492-08279-8



Twitter: @oreillymedia
facebook.com/oreilly

谷歌软件工程

编程过程中获得的经验教训

泰特斯·温特斯、汤姆·曼什莱克和海伦·莱特

北京 波士顿 法纳姆 塞瓦斯托波尔 东京

O'REILLY®

《Google 的软件工程》（作者：
Titus Winters、Tom Mansreck 和 Hyrum Wright）

版权所有 © 2020 Google, LLC。保留所有权利。

在美国印刷。

由 O'Reilly Media, Inc. 出版，地址为 1005 Gravenstein Highway North, Sebastopol, CA 95472。

可以购买 O'Reilly 书籍用于教育、商业或促销用途。大多数书籍都有在线版本(<http://oreilly.com>)。如需了解更多信息，请联系我们的企业/机构销售部门:800-998-9938 或 corporate@oreilly.com。

收购编辑：Ryan Shaw
开发编辑：Alicia Young
制作编辑：克里斯托弗·福彻
文字编辑：Octal Publishing, LLC
校对：Holly Bauer Forsyth

索引编制者：Ellen Troutman-Zaig
室内设计师：David Futato
封面设计师：Karen Montgomery
插画家：Rebecca Demarest

2020 年 3 月： 第一版

第一版修订历史

2020-02-28：首次发布

2020-09-04：第二次发布

请参阅<http://oreilly.com/catalog/errata.csp?isbn=9781492082798>了解发布详情。

O'Reilly 徽标是 O'Reilly Media, Inc. 的注册商标。Google 的软件工程、封面图片和相关商业外观是 O'Reilly Media, Inc. 的商标。

本文表达的观点为作者的观点，不代表出版商的观点。

尽管出版商和作者已尽最大努力确保本作品中包含的信息和说明准确无误，但出版商和作者对错误或遗漏不承担任何责任，包括但不限于因使用或依赖本作品而造成的损害的责任。使用本作品中包含的信息和说明的风险由您自行承担。如果本作品包含或描述的任何代码示例或其他技术受开源许可或他人的知识产权约束，您有责任确保您对其的使用符合此类许可和/或权利。

978-1-492-08279-8

目录

前言
前言
	十九

第一部分论文

1.什么是软件工程?	3
時間與變化 6	
海勒姆定律示	8
例:哈希排序为什么不直接以	9
“没有任何变化”为目标呢?	10
規模与效率	11
无法扩展的政策	12
可扩展的政策	14
示例:编译器升级	14
向左移动	17
权衡与成本	18
例如:标记	19
决策输入	20
示例:分布式构建	20
示例:在时间和规模之间做出选择	22
重新审视决策,犯错	22
软件工程与编程	23
结论	24
TL;DR	24

第二部分文化

2.如何在团队中良好地工作.....	二十七
帮我隐藏我的代码 天才神话 隐藏	二十七
被认为是有害的 早期检	二十八
测 巴士因素 进展速度 简而言之,不要隐	三十
藏 一切都关乎团队 社	31
交互动的三大支柱	31
	三十二
	三十四
	三十四
	三十四
这些支柱为何重要?	三十五
实践中谦逊、尊重和信任 无可指责的事后反思文化 谷歌	三十六
风格 结论	三十九
	41
	四十二
TL;DR	四十二

3.知识共享.....

学习挑战 43	
哲学 奠定基础:	四五五
心理安全指导 大型团体中的心理安全 增长您的知识	四五六
提出问题 了解背	四十六
景 扩展您的问题:询问社区 群组聊天 邮件列表	四十七
YAQS:问答平台 扩展您的知识:您总	四十八
有一些东西可以教	四十八
办公时间 技术讲座和课程	49
	50
	50
	50
	51
	52
	52
	52
文档	53
代码	56
扩展组织的知识	56
培育知识共享文化	56
建立规范的信息来源	58

保持联系	61
可读性:通过代码审查实现标准化指导	62
可读性过程是什么?	63
为什么要有这个过程?	64
结论	66
TL;DR	67
 4.公平工程.....	69
偏见是默认的 理解多样	70
性的必要性 建设多元文化能力 使多样性具有可	72
操作性 拒绝单一方法 挑战既定流程 价值	72
观与结果 保持好奇心,不断前进 结论	74
	75
	76
	77
	78
	79
TL;DR	79
 5.如何领导一个团队. ...	
经理和技术主管 (或两者)	81
工程经理 技术主管 技术主管经理	82
	82
	82
从个人贡献者角色转变为领导角色	83
唯一需要担心的是……嗯,一切仆人式领导 工程经理 经理是个	84
四个字母的词 当今的工程	85
经理 反模式 反模式:雇佣软弱的人 反	86
模式:忽视低绩效者 反模式:忽视人性化问题	86
反模式:成为每个人的朋友 反模式:降低招	87
聘标准 反模式:像	88
对待孩子一样对待您的团队 积极模式	89
放下自我 成为禅师 成为催化剂	89
	90
	91
	92
	92
	93
	93
	94
	96

消除障碍	96
成为一名教师和导师	97
设定明确的目标	97
诚实	98
追踪幸福	99
意想不到的问题	100
其他提示和技巧	101
人就像植物	103
内在动机与外在动机	104
结论	105
TL;DR	105
6.规模领先.....	
始终保持决策	108
飞机的寓言	108
识别盲点	109
确定关键的权衡	109
决定,然后迭代	110
总是要离开	112
你的任务:建立一支“自动驾驶”团队	112
划分问题空间	113
始终保持扩张	116
成功的循环	116
重要与紧急	118
学习投球	119
保护你的能源	120
结论	122
TL;DR	122
7.测量工程生产力.....	.123
为什么要衡量工程生产力?	123
分类:是否值得衡量?	125
选择具有目标和信号的有意义的指标	129
目标信号指标使用数据验证指标采取行动并跟踪结	130
果结论	132
TL;DR	132
	133
	137
	137
	137

第三部分流程

8.风格指南和规则.....	141
为什么要有规则?	142
创建规则	143
指导原则	143
风格指南	151
改变规则	154
流程	155
风格裁决者	156
例外	156
指导	157
应用规则	158
错误检查器	160
代码格式化程序	161
结论	163
TL;DR	163
9.代码审查.....	
代码审查流程	166
Google 的代码审查是如何进行的	167
代码审查的好处	170
代码正确性	171
代码理解	172
代码一致性	173
心理和文化方面的益处	174
知识共享	175
代码审查最佳实践	176
礼貌而专业	176
写下小的改变	177
撰写良好的变更描述	178
尽量减少审阅者	179
尽可能实现自动化	179
代码审查的类型	180
Greenfield 代码审查	180
行为变化、改进和优化	181
错误修复和回滚	181
重构和大规模变更	182
结论	182
TL;DR	183

10.文档.	
什么才算是文档?	185
为什么需要文档?	186
文档就像代码 了解你的受众 受众类型	188
文档类型 参考文档 设计文档	190
教程 概念文档 登陆页面	191
文档评论 文档哲学 谁、什么、何	192
时、何地和为什么 开头、中间和结尾	193
优秀文档的参数	195
何时需要技术作家来弃用文档?	196
何时需要技术作家来弃用文档?	198
何时需要技术作家来弃用文档?	198
何时需要技术作家来弃用文档?	199
何时需要技术作家来弃用文档?	201
何时需要技术作家来弃用文档?	201
何时需要技术作家来弃用文档?	202
何时需要技术作家来弃用文档?	202
何时需要技术作家来弃用文档?	203
何时需要技术作家来弃用文档?	204
结论	204
	205
11.测试概述.	
我们为什么要写测试?	208
Google Web 服务器测试的故事 以现代开发速度	209
进行测试 编写、运行、响应 测试代码的好处 设计测试套件 测试规模 测	210
试范围 碧昂丝规则 关于	212
Google 规模代码覆盖率测试的说明	213
大型测试套件的陷阱	214
大型测试套件的陷阱	215
大型测试套件的陷阱	219
大型测试套件的陷阱	221
大型测试套件的陷阱	222
大型测试套件的陷阱	223
大型测试套件的陷阱	224
谷歌测试的历史	225
入学指导课程	226
测试认证	227
在马桶上进行测试	227
当今的测试文化	228

自动化测试的局限性	229
结论	230
TL;DR	230
12. 单元测试.....	
可维护性的重要性	232
防止脆弱测试，争取不变的测试	233
通过公共 API 进行测试，测试状态，而不是 交互 编写清晰的测试	233
让你的测试完整而简洁	234
测试行为，而不是方法	238
不要在测试中加入逻辑	239
写入清除失败消息	240
测试和代码共享：DAMP，而不是DRY	241
共同价值观	246
共享设置	247
共享帮助程序和验证	248
定义测试基础设施	251
结论	253
TL;DR	254
13. 测试替身....	
测试替身对软件开发的影响	258
Google 的测试替身基本概念示例	258
测试替身接缝模拟框架	259
使用测试替身的技术	259
伪造存根交互测	260
试真实实现更喜欢真实	261
性而不是孤立如何决定何时使用真实实	262
现伪造为	263
什么伪造很重要？	263
什么时候应该写假货？	264
赝品的保真度	265
	266
	269
	270
	270
	271

假货应该经过检验	272
如果没有假货该怎么办	272
存根 过度	272
使用存根的危害 什么时候使用存根合适?	273
	275
交互测试优先于状态测试	275
而不是交互测试何时进行交互测试合适?	275
	277
交互测试的最佳实践结论	277
	280
TL;DR	280
14.更大规模的测试.....	
什么是更大规模的测试?	281
保真度单	282
元测试中的常见差距为什么不进行更	283
大规模的测试?	285
谷歌的大型测试	286
更大的测试和时间	286
在 Google Scale 上进行更大规模的测试	288
大型测试的结构	289
被测系统	290
测试数据	294
确认	295
大型测试的类型	296
一个或多个交互二进制文件的功能测试	297
浏览器和设备测试	297
性能、负载和压力测试	297
部署配置测试	298
探索性测试	298
A/B 差异回归测试	299
验收测试 (UAT)	301
探测器和金丝雀分析	301
灾难恢复与混沌工程	302
用户评价	303
大型测试和开发人员工作流程	304
编写大型测试	305
运行大型测试	305
拥有大型测试	308
结论	309
TL;DR	309

15.弃用.....

为何弃用? 312	313
为何弃用如此困难?	315
设计期间的弃用 弃用类型	316
建议弃用	316
强制折旧	317
弃用警告	318
管理弃用流程	319
流程所有者	320
里程碑	320
弃用工具	321
结论	322
TL;DR	323

第四部分工具

16.版本控制和分支管理

什么是版本控制?	327
为什么版本控制很重要?	329
集中式 VCS 与分布式 VCS 事实来源版本控制与依赖管理 分	331
支管理 正在进行的工	334
作类似于分支 开发分支 发布分支 Google 的版本控制	336
336	336
336	337
339	339
340	340
一个版本场景:多	340
个可用版本 “一个版本”规则 (几乎)没有长寿命分	341
支那么发布分支呢?	342
343	343
344	344
Monorepos	345
版本控制的未来	346
结论	348
TL;DR	349

17.代码搜索.....	352
代码搜索 UI Google 员工如	352
如何使用代码搜索?	353
在哪里?	353
什么?	354
如何?	354
为什么?	354
谁和何时?	355
为什么要使用单独的 Web 工具?	355
规模零	355
设置全局代码视图专业化与其他开发者工具	356
集成API公开规模对	356
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	356
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	359
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	360
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	361
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	361
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	363
设计的影响搜索查询延迟索引延迟Google的实施搜索索引	366
选定的权衡	366
完整性 : 存储库位于头部	366
完整性 : 全部结果与最相关结果	366
完整性 : 头部、分支、全部、历史、	
工作区	367
表达能力 : 标记、子字符串和正则表达式	368
结论	369
TL;DR	370
18.构建体系、构建理念.....	371
构建系统的目的如果没有构建系统	371
会发生什么?	372
但我需要的只是一个编译器!	373
Shell 脚本可以解决问题吗?	373
现代构建系统一切都与依赖关	375
系有关基于任务的构建系统基于工件	375
的构建系统分布式构建时间、规模、	376
权衡	380
	386
	390

处理模块和依赖关系	390
使用细粒度模块和 1:1:1 规则	391
最小化模块可见性	392
管理依赖关系	392
结论	397
TL;DR	397
 19.批评:Google 的代码审查工具.....	399
代码审查工具原则 代码审查流程	399
通知	402
第一阶段:创造变革	402
差异	403
分析结果	404
紧密的工具集成	406
第 2 阶段:请求审核	406
第 3 和第 4 阶段:理解和评论变更	408
评论	408
了解变更状态	410
第五阶段:变更批准 (变更评分)	412
第六阶段:提交变更	413
提交后:跟踪历史	414
结论	415
TL;DR	416
 20.静态分析.....	417
有效静态分析的特征	418
可扩展性	418
可用性	418
静态分析工作的关键经验	419
关注开发者的幸福感	419
使静态分析成为核心开发人员工作流程的一部分	420
让用户有能力做出贡献	420
Tricorder:Google 的静态分析平台	421
集成工具	422
整合反馈渠道	423
建议修复	424
根据项目定制	424
预提交	425
编译器集成	426
编辑和浏览代码时进行分析	427

结论	428
TL;DR	428
21.依赖关系管理. ...	
为什么依赖管理如此困难?	431
冲突的要求和钻石依赖性	431
导入依赖项	433
兼容性承诺	433
导入时的注意事项	436
Google 如何处理导入依赖项	437
依赖管理的理论	439
什么都没有改变 (又称静态依赖模型)	439
语义版本控制	440
捆绑分销模式	441
住在 Head	442
SemVer 的局限性	443
SemVer 可能会过度限制	444
SemVer 可能承诺过多	445
动机最低版本选	446
择那么,SemVer 有效吗?	447
448	448
使用无限资源进行依赖管理导出依赖项结论 TL;DRs	449
452	452
456	456
456	456
22.大规模变更. ...	
什么是大规模变革? 460	460
谁与 LSC 打交道?	461
原子修改的障碍 技术限制 合并冲突 没	463
有闹鬼的墓地 异构性测试 代	463
码审查	463
464	464
464	464
465	465
467	467
LSC 基础设施	468
政策与文化	469
代码库洞察	470
变更管理	470
测试	471

语言支持	471
LSC 流程	472
授权	473
改变创造	473
分片和提交	474
清理	477
结论	477
TL;DR	478
23.持续集成.	
CI 概念	481
快速反馈循环	481
自动化	483
持续测试	485
CI 挑战	490
密封测试	491
Google 的 CI	493
CI 案例研究:Google Takeout	496
但我负担不起 CI	503
结论	503
TL;DR	503
24.持续交付.....	
Google 持续交付的惯用语	506
速度是一项团队运动:如何将部署分解为可管理的件	507
评估隔离中的变化:旗帜保护功能	508
追求敏捷:建立发布列车	509
没有完美的二进制文件	509
满足发布期限	510
质量和用户关注:只发布有用的东西	511
向左移动:尽早做出数据驱动的决策	512
改变团队文化:在部署中建立纪律	513
结论	514
TL;DR	514
25.计算即服务.....	
驯服计算环境 518	518
任务容器化和多租户自动	520
化总结	523

编写用于托管计算的软件	523
为失败而构建	523
批量与服务	525
管理国家	527
连接到服务	528
一次性代码	529
随着时间的推移和规模的 CaaS	530
容器作为一种抽象	530
一项服务统领一切	533
已提交配置	535
选择计算服务	535
集中化与定制化	537
抽象级别:无服务器	539
公共与私人	543
结论	544
TL;DR	545

第五部分结论

后记.....

索引.....

前言

我一直对 Google 做事的细节着迷不已。我曾向 Google 员工朋友们询问公司内部的真实运作方式。他们如何管理如此庞大、单片式的代码存储库而不倒下？数万名工程师如何成功协作完成数千个项目？他们如何保持系统的质量？

与前 Google 员工共事只会增加我的好奇心。如果你曾经与前 Google 工程师（有时被称为“Xoogler”）共事过，你肯定听过这句话：“在 Google，我们……”从 Google 转行到其他公司似乎是一种令人震惊的经历，至少从工程方面来看是如此。据我所知，考虑到公司的规模以及人们对其实力的赞誉，Google 编写代码的系统和流程一定是世界上最好的。

在《谷歌的软件工程》一书中，一群谷歌员工（以及一些 Xooglers）为我们提供了谷歌软件工程的许多实践、工具甚至文化元素的详细蓝图。我们很容易过于关注谷歌为支持编写代码而构建的出色工具，而这本书提供了有关这些工具的大量详细信息。但它也不仅仅是描述工具，还为我们提供了谷歌团队遵循的理念和流程。无论您是否拥有规模和工具，这些都可以根据各种情况进行调整。令我高兴的是，有几章深入探讨了自动化测试的各个方面，而这个话题在我们这个行业中仍然遭遇了太多阻力。

科技的伟大之处在于，做某事从来不仅有一种方法。

相反，我们所有人都必须根据团队和情况做出一系列权衡。我们可以从开源中廉价地获取什么？我们的团队可以构建什么？什么对支持我们的规模有意义？当我盘问我的谷歌员工朋友时，我想听听规模极端的世界：资源丰富，人才和金钱兼具，对软件的要求很高。

构建。这些轶事信息让我想到了一些我可能不会考虑的选择。

在这本书中,我们列出了这些选项,供大家阅读。当然,谷歌是一家独特的公司,如果认为运营软件工程组织的正确方法是完全复制他们的公式,那就太愚蠢了。

从实际应用上看,本书将为您提供有关如何完成事情的想法,并提供大量信息,您可以用来支持采用最佳实践的论据,例如测试、知识共享和建立协作团队。

您可能永远不需要亲自创建 Google,甚至可能不想采用他们在您的组织中应用的相同技术。但如果您不熟悉 Google 开发的实践,那么您就错过了软件工程的视角,这种视角来自二十多年来成千上万名工程师在软件开发方面的合作。这些知识太宝贵了,不容忽视。

卡米尔·福尼耶
《管理者之路》作者

前言

这本书的标题是《谷歌的软件工程》。我们所说的软件工程到底是什么意思？“软件工程”与“编程”或“计算机科学”有何区别？为什么谷歌会以独特的视角为过去 50 年来编写的软件工程文献增添新的内容？

在我们这个行业，“编程”和“软件工程”这两个术语已经互换使用了很长时间，尽管每个术语都有不同的侧重点和不同的含义。大学生往往学习计算机科学，并以“程序员”的身份从事编写代码的工作。

然而，“软件工程”听起来更严肃，好像它意味着应用一些理论知识来构建真实而精确的东西。机械工程师、土木工程师、航空工程师和其他工程学科的工程师都从事工程实践。他们都在现实世界中工作，并运用他们的理论知识来创造真实的东西。软件工程师也创造“真实的东西”，尽管它不像其他工程师创造的东西那样有形。

与那些更为成熟的工程专业不同，当前的软件工程理论或实践远没有那么严格。航空工程师必须遵循严格的指导方针和实践，因为他们的计算错误可能会造成真正的损害；总体而言，编程传统上并没有遵循如此严格的实践。

但是，随着软件越来越融入我们的生活，我们必须采用并依赖更严格的工程方法。我们希望这本书能帮助其他人找到一条通往更可靠软件实践的道路。

随着时间推移而编程

我们认为“软件工程”不仅包括编写代码的行为，还包括组织用于构建和维护代码的所有工具和流程。软件组织可以引入哪些实践来最好地保持其

代码的长期价值如何?工程师如何才能让代码库更具可持续性,让软件工程学科本身更加严格?我们对这些问题还没有根本的答案,但我们希望谷歌过去 20 年的集体经验能够照亮寻找这些答案的可能途径。

我们在本书中分享的一个关键见解是,软件工程可以被认为是“随时间推移而集成的编程”。我们可以在代码中引入哪些实践,使其在从构思到引入到维护到弃用的整个生命周期中具有可持续性(能够对必要的更改做出反应)?

本书强调了我们认为软件组织在设计、架构和编写代码时应该牢记的三个基本原则:

時間與變化

代码在其生命周期内需要如何适应

规模与增长

组织在发展过程中需要如何适应

权衡与成本

组织如何根据时间、变化、规模和增长的经验做出决策

在整个章节中,我们试图回顾这些主题,并指出这些原则如何影响工程实践并使其可持续。(有关完整讨论,请参阅[第 1 章](#)。)

谷歌的观点

谷歌对可持续软件生态系统的增长和发展有着独特的见解,这源于我们的规模和悠久的历史。我们希望,随着贵组织的发展和采用更可持续的做法,我们所学到的经验教训将大有裨益。

我们将本书的主题分为 Google 软件工程格局的三个主要方面:

- 文化
- 流程
- 工具

Google 的文化是独一无二的,但我们在发展工程文化过程中学到的经验教训具有广泛的适用性。我们关于文化的章节([第二部分](#))强调了软件开发企业的集体性质,软件开发是一项团队工作,适当的文化原则对于组织成长和保持健康至关重要。

大多数软件工程师都熟悉我们在“流程”章节（[第 III 部分](#)）中概述的技术，但 Google 庞大且长期存在的代码库为开发最佳实践提供了更完整的压力测试。在这些章节中，我们试图强调我们发现的随着时间的推移和规模扩大后行之有效的方法，并确定我们尚未找到令人满意的答案的领域。

最后，我们的工具章节（[第 IV 部分](#)）说明了我们如何利用对工具基础设施的投资，为我们的代码库在增长和老化过程中带来好处。

在某些情况下，这些工具是 Google 独有的，但我们在适用的情况下指出开源或第三方替代方案。我们预计这些基本见解适用于大多数工程组织。

本书概述的文化、流程和工具描述了典型软件工程师希望在工作中学习的经验教训。Google 当然不会垄断好的建议，我们在此介绍的经验并非旨在指示您的组织应该做什么。本书是我们的观点，但我们希望您会发现它很有用，无论是直接采用这些经验教训，还是在考虑您自己的实践时以它们为起点，专门针对您自己的问题领域。

本书也并非旨在布道。Google 本身仍然不完美地应用了这些页面中的许多概念。我们从失败中吸取的教训是：我们仍然会犯错误，实施不完美的解决方案，并且需要不断改进。然而，Google 工程组织的庞大规模确保了每个问题都有多种解决方案。我们希望本书包含其中最好的解决方案。

这本书不是

本书并非旨在介绍软件设计，这是一门需要专门一本书来阐述的学科（而且这方面的许多内容已经存在）。尽管本书中有一些代码用于说明目的，但这些原则与语言无关，并且这些章节中几乎没有真正的“编程”建议。因此，本书没有涉及软件开发中的许多重要问题：项目管理、API 设计、安全强化、国际化、用户界面框架或其他特定于语言的问题。本书省略了这些问题并不意味着它们不重要。相反，我们选择不在这里讨论它们，因为我们知道我们无法提供应有的待遇。我们试图使本书的讨论更多地涉及工程而不是编程。

告别辞

本文是所有贡献者倾注的心血之作,我们希望您能将其视为:了解大型软件工程组织如何构建其产品的一扇窗口。我们还希望它是众多声音之一,有助于推动我们的行业采用更具前瞻性和可持续性的做法。最重要的是,我们进一步希望您喜欢阅读它,并能从中吸取一些教训,以应对您自己的问题。

汤姆·曼什雷克

本书中使用的约定

本书采用以下印刷约定:

斜体

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

恒定宽度用于程序列

表,以及段落内引用程序元素,例如变量或函数名称、数据库、数据类型、环境变量、语句和关键字。

等宽粗体显示应由用户逐字输入

的命令或其他文本。

等宽斜体显示应由用户提供值或

由上下文确定的值替换的文本。



此元素表示一般说明。

O'Reilly 在线学习

O'REILLY[®] 40 多年来，O'Reilly Media 提供技术和业务培训、知识和见解，帮助公司取得成功。

我们独特的专家和创新者网络通过书籍、文章和我们的在线学习平台分享他们的知识和专长。O'Reilly 的在线学习平台让您按需访问现场培训课程、深入的学习路径、交互式编码环境以及来自 O'Reilly 和 200 多家其他出版商的大量文本和视频。有关更多信息，请访问<http://oreilly.com>。

如何联系我们

请将有关本书的评论和问题发送给出版商：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (美国或加拿大) 707-829-0515 (国际或
本地) 707-829-0104 (传真)

我们为本书建立了一个网页，其中列出了勘误表、示例和任何其他信息。您可以通过<https://oreil.ly/soware-engineering-at-google> 访问该网页。

电子邮件bookquestions@oreilly.com对本书进行评论或询问技术问题。

有关我们的书籍和课程的新闻和更多信息，请访问我们的网站<http://www.oreilly.com>。

在 Facebook 上找到我们：<http://facebook.com/>

[oreilly](#) 在 Twitter 上关注我们：<http://twitter.com/>

[oreillymedia](#) 在 YouTube 上观看我们：<http://www.youtube.com/oreillymedia>

致谢

如果没有无数其他人的努力,这样的书是不可能问世的。本书中的所有知识都来自我们整个职业生涯中在 Google 工作的许多其他人的经验。我们是信使;其他人在我们之前就已在 Google 和其他地方工作,并教会了我们现在呈现给你们的内容。我们无法在此列出你们所有人,但我们确实希望感谢你们。

我们还要感谢 Melody Meckfessel 在该项目初期给予的支持,以及 Daniel Jasper 和 Danny Berlin 在该项目完成前给予的支持。

如果没有我们的策展人、作者和编辑们的大力合作,这本书就不可能问世。虽然作者和编辑们在每一章或注释中都得到了特别的致谢,但我们还是想花点时间感谢那些为每一章做出贡献的人,他们提供了深思熟虑的意见、讨论和评论。

·什么是软件工程? : Sanjay Ghemawat、Andrew Hyatt ·团队合作: Sibley Bacon、
Joshua Morton ·知识共享: Dimitri Glazkov、Kyle Lemons、John
Reese、David Symonds、Andrew Trenk、James Tucker、David Kohlbrenner、Rodrigo Damazio
Bovendorp ·公平工程: Kamau Bobb、Bruce Lee ·如何领导团队: Jon Wiley、Laurent Le Brun ·大
规模领导:
Bryan O Sullivan、Bharat Mediratta、Daniel Jasper、Shaindel

施瓦茨

·衡量工程生产力: Andrea Knight、Collin Green、Caitlin
萨多斯基、马克斯-卡纳特·亚历山大、杨伊蕾
·风格指南和规则: Max Kanat-Alexander、Titus Winters、Matt Austern、
James Dennett ·
代码审查: Max Kanat-Alexander、Brian Ledger、Mark Barolak ·文档: Jonas
Wagner、Smit Hinsu、Geoffrey Romer ·测试概述: Erik Kuefler、Andrew Trenk、
Dillon Bly、Joseph Graves、Neal
诺维茨、杰伊·科贝特、马克·斯特里贝克、布拉德·格林、米什科·赫弗里、安托万
皮卡德、莎拉·斯托克
·单元测试: Andrew Trenk、Adam Bender、Dillon Bly、Joseph Graves、Titus
温特斯、海伦·赖特、奥吉·法克勒 ·测试替身: 约瑟夫·
格雷夫斯、根纳迪·西维尔

·更大的测试： Adam Bender、Andrew Trenk、Erik Kuefler、Matthew Beaumont-Gay ·

弃用： Greg Miller、Andy Shulman ·版本控制和分支管

理： Rachel Potvin、Victoria Clarke ·代码搜索： Jenny Wang ·构建系统和构建理念： Hyrum Wright、Titus Winters、Adam

本德、杰夫·考克斯、雅克·皮纳尔

批评：Google 的代码审查工具： Miko aj D dela、Hermann Loose、Eva 梅、爱丽丝·科伯·索泽克、埃德温·凯宾、帕特里克·希塞尔、奥勒·雷姆森、简·马切克

静态分析： Jeffrey van Gogh、Ciera Jaspan、Emma Söderberg、Edward Aftandilian、Collin Winter、Eric Haugh

·依赖管理： Russ Cox、Nicholas Dunn ·大规模变更： Matthew

Fowles Kulukundis、Adam Zarek ·持续集成： Jeff Listfield、John Penix、Kaushik Sridharan、Sanjeev 丹达

持续交付： Dave Owens、Sheri Shipe、Bobbi Jones、Matt Duftler、布莱恩·苏特

·计算服务： Tim Hockin、Collin Winter、Jarek Ku mierenk

此外，我们还要感谢 Betsy Beyer 分享她在出版原版《站点可靠性工程》一书方面的见解和经验，这使我们的体验更加顺畅。O'Reilly 的 Christopher Guzikowski 和 Alicia Young 在启动和指导该项目出版方面做得非常出色。

策展人还要亲自感谢以下人士：

汤姆·曼什雷克 (Tom Mansreck)：感谢我的爸爸妈妈，他们让我相信自己 并且在厨房的餐桌上陪我做作业。

泰特斯·温特斯：爸爸，为我指引道路。妈妈，为我发声。维多利亚，为我献出真心。

感谢 Raf 对我的支持。此外，还要感谢 Mr Snyder、Ranwa、Z、Mike、Zach、Tom（以及所有 Paynes）、mec、Toby、cgd 和 Melody 的教导、指导和信任。

Hyrum Wright：感谢爸爸妈妈的鼓励。感谢 Bryan 和 Bakerland 的居民们，感谢他们让我初次涉足软件领域。感谢 Dewayne，感谢他继续我的旅程。感谢 Hannah、Jonathan、Charlotte、Spencer 和 Ben 的爱与关心。

感谢 Heather 一直以来的陪伴。

第一部分

论文

第一章

什么是软件工程?

作者:Titus Winters
编辑:Tom Mansreck

没有什么东西是建立在石头上的;一切都是建立在沙子上的,但我们必须把沙子当作石头来建造。

豪尔赫·路易斯·博尔赫斯

我们看到编程和软件工程之间存在三个关键差异:时间、规模和权衡。在软件工程项目中,工程师需要更加关注时间的流逝和最终的变更需求。在软件工程组织中,我们需要更加关注规模和效率,既要关注我们生产的软件,也要关注生产软件的组织。最后,作为软件工程师,我们需要做出更复杂、风险更高的决策,这些决策通常基于对时间和增长的不精确估计。

在 Google 内部,我们有时会说:“软件工程是随着时间的推移而形成的编程。”编程无疑是软件工程的重要组成部分:毕竟,编程是你首先生成新软件的方式。如果你接受这种区别,那么我们可能需要区分编程任务(开发)和软件工程任务(开发、修改、维护)。时间的增加为编程增加了一个重要的新维度。立方体不是正方形,距离不是速度。软件工程不是编程。

了解时间对程序的影响的一种方法是思考这个问题:“你的代码的预期寿命¹是多少?”这个问题的合理答案

¹我们指的不是“执行寿命”,而是“维护寿命” 代码将持续运行多久?
构建、执行和维护?该软件将提供价值多长时间?

大约相差 100,000 倍。考虑只需要持续几分钟的代码和想象可以存活几十年的代码同样合理。通常,处于该范围较短端的代码不受时间的影响。对于实用性仅为一小时的程序,不太可能需要适应新版本的底层库、操作系统 (OS)、硬件或语言版本。这些短暂的系统实际上 “只是”一个编程问题,就像一个立方体在一维上压缩到足够远后就是一个正方形一样。随着我们延长时间以允许更长的生命周期,改变变得更加重要。在十年或更长的时间跨度内,大多数组程序依赖关系 (无论是隐式的还是显式的)都可能会发生变化。这种认识是我们区分软件工程和编程的根本原因之一。

这种区别是我们所说的软件可持续性的核心。如果在软件的预期寿命内,你能够对因技术或业务原因而出现的任何有价值的变化做出反应,那么你的项目就是可持续的。

重要的是,我们只关注能力 您可能选择不执行给定的升级,无论是由于缺乏价值还是其他优先事项。²当您从根本上无法对底层技术或产品方向的变化做出反应时,您就是在进行高风险的赌注,希望这种变化永远不会变得至关重要。

对于短期项目来说,这可能是一个安全的选择。但从几十年来看,可能并非如此。³

看待软件工程的另一种方式是考虑规模。有多少人参与?他们在开发和维护过程中扮演什么角色?

编程任务通常是个体创作的行为,但软件工程任务是团队合作。早期对软件工程的定义为这一观点提供了一个很好的定义:“多人开发多版本程序。”⁴这表明软件工程和编程之间的区别在于时间和人员。团队协作带来了新的问题,但也提供了比任何单个程序员更大的潜力来创建有价值的系统。

团队组织、项目组成以及软件项目的政策和实践都主导着软件工程复杂性的这一方面。这些问题也是规模所固有的:随着组织的发展和项目的扩展,软件生产效率是否会提高?我们的开发工作流程是否

² 这也许是对技术债务的一个合理的粗略定义:应该做但实际上没有做的事情
然而 我们的代码和我们所希望的代码之间存在差异。

³ 还要考虑我们是否提前知道一个项目将会长期存在的问题。

⁴ 关于这句话的原始出处存在一些疑问;一致认为这句话最初是由 Brian Randell 或 Margaret Hamilton 说出的,但也可能完全是 Dave Parnas 编造的。对这句话的常见引用是“软件工程技术:北约科学委员会主办的会议报告”,意大利罗马,1969 年 10 月 27-31 日,布鲁塞尔,北约科学事务部。

随着我们的成长,我们的版本控制策略和测试策略是否会让我们的成本相应增加?自软件工程早期以来,围绕沟通和人力扩展的规模问题就一直在讨论,可以追溯到人月神话。5这样的规模问题是政策问题,是软件可持续性问题的基础:重复做我们需要做的事情要花多少钱?

我们还可以说,软件工程与编程的不同之处在于,所需做出的决策的复杂性及其利害关系。在软件工程中,我们经常被迫在几条前进道路之间权衡利弊,有时这些道路的风险很高,而且价值指标往往不完善。软件工程师或软件工程领导者的工作是致力于组织、产品和开发工作流程的可持续性以及对扩展成本的管理。考虑到这些输入,评估您的权衡并做出理性决策。我们有时可能会推迟维护更改,甚至接受扩展性不佳的政策,但我们知道我们需要重新审视这些决定。这些选择应该明确并清楚地说明延期成本。

软件工程中很少有放之四海而皆准的解决方案,本书也是如此。对于“这个软件能存活多久”这个问题,合理的答案有 10 万倍,对于“你的组织中有多少名工程师”这个问题,答案可能也有 1 万倍,至于“你的项目有多少计算资源可用”,答案也不知道有多少,Google 的经验可能与你的不一样。在本书中,我们旨在介绍我们在构建和维护软件时发现的对我们有用的方法,这些软件预计将持续数十年,需要数万名工程师和遍布全球的计算资源。我们发现,大多数在这种规模下是必要的做法,也适用于较小的项目:我们认为,这是一个关于一个工程生态系统的报告,随着规模的扩大,这个生态系统可能会很好。在一些地方,超大规模有其自身的成本,我们更愿意不支付额外的开销。我们称其为警告。希望如果您的组织发展到足够大以至于需要担心这些成本,您可以找到更好的答案。

在我们讨论团队合作、文化、政策和工具的具体细节之前,让我们首先详细阐述时间、规模和权衡这些主要主题。

5 Frederick P. Brooks Jr.《人月神话:软件工程论文集》(波士顿:Addison-Wesley,1995年)。

時間與變化

当新手学习编程时,生成的代码的生命周期通常以小时或天来衡量。编程作业和练习往往是一次性编写的,很少或根本不需要重构,当然也不需要长期维护。这些程序通常在首次制作后不会再重建或执行。

在教学环境中,这并不奇怪。也许在中学或高等教育中,我们可能会发现团队项目课程或实践论文。如果是这样,这些项目很可能是学生代码唯一能存活超过一个月左右的时间。这些开发人员可能需要重构一些代码,也许是应对不断变化的需求,但他们不太可能被要求处理其环境的最大变化。

我们还发现,在常见的行业环境中,开发人员编写的都是短命代码。移动应用程序的生命周期通常很短,⁶不管怎样,完全重写都是比较常见的。早期创业公司的工程师可能会正确地选择关注短期目标,而不是长期投资:公司可能存活的时间不够长,无法从回报缓慢的基础设施投资中获益。连续创业公司的开发人员可能拥有 10 年的开发经验,但几乎没有或根本没有维护任何预计会存在一两年以上的软件的经验。

另一方面,一些成功的项目实际上具有无限的生命周期:我们无法合理地预测 Google 搜索、Linux 内核或 Apache HTTP Server 项目的终点。对于大多数 Google 项目,我们必须假设它们将无限期地存在——我们无法预测何时不再需要升级依赖项、语言版本等。随着它们的生命周期增长,这些长期项目最终会给人与编程任务或初创企业开发不同的感觉。

考虑图 1-1,它展示了处于“预期寿命”范围两端的两个软件项目。对于一个程序员来说,他正在从事一项预期寿命为数小时的任务,可以合理地预期哪些类型的维护?也就是说,如果你正在编写一个只执行一次的 Python 脚本时,操作系统的新版本发布了,你是否应该放下手头的工作去升级?当然不是:升级并不重要。但在另一端,如果 Google 搜索停留在 20 世纪 90 年代的操作系统版本上,那将是一个明显的问题。

⁶ Appcelerator, “除了死亡、税收和移动应用寿命短,没有什么是确定的”, Axway 开发者博客,2012 年 12 月 6 日。

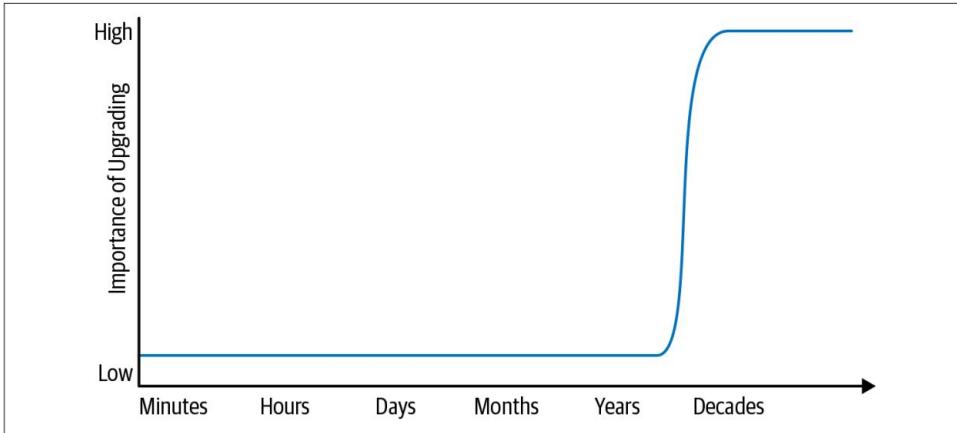


图 1-1. 使用寿命和升级的重要性

预期寿命范围的低点和高点表明存在某个过渡。在一次性项目和持续数十年的项目之间的某个地方,会发生过渡：项目必须开始对不断变化的外部因素做出反应。7对于任何从一开始就没有计划升级的项目来说,这种过渡可能非常痛苦,原因有三个,每个原因都相互叠加：

- 您正在执行该项目尚未完成的任务;更多隐藏假设已经被融入。
- 尝试进行升级的工程师不太可能有这方面的经验这种任务。
- 升级的规模通常比平常更大,相当于几年的一次性升级,而不是逐步升级。

因此,在实际经历一次这样的升级(或中途放弃)之后,高估后续升级的成本并决定“再也不升级”是相当合理的。得出这个结论的公司最终会决定放弃一些东西并重写他们的代码,或者决定再也不升级。与其采取自然的方法避免痛苦的任务,有时更负责任的答案是投资让它变得不那么痛苦。这一切都取决于你的升级成本、它提供的价值以及相关项目的预期寿命。

⁷你自己优先考虑的事情和品味将决定这种转变发生在哪。我们发现大多数项目似乎愿意在五年内升级。对于这一转变,5 到 10 年左右似乎是一个保守的估计。

不仅要完成第一次重大升级,还要达到可以可靠地保持最新状态的程度,这是项目长期可持续发展的本质。可持续发展需要规划和管理所需变更的影响。对于 Google 的许多项目,我们相信我们已经实现了这种可持续发展,这主要是通过反复试验实现的。

那么,具体来说,短期编程与编写预期寿命更长的代码有何不同?随着时间的推移,我们需要更加清楚“碰巧能用”和“可维护”之间的区别。没有完美的解决方案来识别这些问题。这很不幸,因为保持软件长期可维护是一场持久战。

海伦法则

如果你正在维护一个被其他工程师使用的项目,那么关于“它有效”与“它是可维护的”最重要的一课就是我们所说的 Hyrum 的法律:

如果 API 有足够的用户,那么你在合同中承诺什么并不重要:系统的所有可观察到的行为都将依赖于某人。

根据我们的经验,这条公理是任何关于随时间推移改变软件的讨论中的主导因素。它在概念上类似于熵:关于随时间推移改变和维护的讨论必须注意海伦定律⁸,就像讨论效率或热力学时必须注意熵一样。仅仅因为熵永远不会减少并不意味着我们不应该努力提高效率。仅仅因为海伦定律在维护软件时适用并不意味着我们不能为此做计划或尝试更好地理解它。我们可以减轻它,但我们知道它永远无法根除。

海勒姆定律代表了这样的实践知识:即使怀着最好的意图、最好的工程师和可靠的代码审查实践,我们也不能假设完全遵守已发布的合同或最佳实践。作为 API 所有者,通过明确接口承诺,您将获得一些灵活性和自由,但在实践中,特定更改的复杂性和难度还取决于用户认为 API 的某些可观察行为有多有用。如果用户不能依赖这些东西,您的 API 将很容易更改。如果有足够的时间和足够的用户,即使是最无害的更改也会造成某些破坏;⁹您对更改价值的分析必须考虑调查、识别和解决这些破坏的难度。

⁸值得赞扬的是,海伦非常努力地谦虚地称其为“隐性依赖定律”,但“海伦的法律”是谷歌大多数人已经接受的简称。

⁹请参阅“[工作流程](#)”,xkcd 漫画。

示例 : 哈希排序考虑哈希迭代

排序的示例。如果我们将五个元素插入基于哈希的集合 , 我们按什么顺序取出它们 ?

```
>>> for i in { "苹果"、"香蕉"、"胡萝卜"、"榴莲"、"茄子" }: print(i)
...
榴莲
胡萝卜 苹
果 茄子
香蕉
```

大多数程序员都知道哈希表是非明显排序的。很少有人知道他们使用的特定哈希表是否打算永远提供这种特定的排序。这似乎并不奇怪 , 但在过去一二十年里 , 计算行业使用此类类型的经验已经发生了变化 :

- 哈希泛洪¹⁰攻击为非确定性哈希提供了更大的激励迭代。
- 通过研究改进哈希算法或哈希容器需要改变哈希迭代顺序。
- 根据海勒姆定律 , 程序员会根据以下顺序编写程序 : 如果他们有能力这样做 , 就会遍历其中一个哈希表。

因此 , 如果你问任何专家 “ 我可以假设我的哈希容器有一个特定的输出序列吗 ? ” , 那位专家大概会说 “ 不能 ” 。总的来说 , 这是正确的 , 但可能过于简单。更细致的答案是 , “ 如果你的代码是短暂的 , 硬件、语言运行时或数据结构选择都没有变化 , 那么这样的假设是可以的。如果你不知道你的代码会存活多久 , 或者你不能保证你所依赖的东西永远不会改变 , 那么这样的假设就是错误的。 ” 此外 , 即使你自己的实现不依赖于哈希容器顺序 , 它也可能被其他隐式创建这种依赖关系的代码使用。例如 , 如果你的库将值序列化为远程过程调用 (RPC) 响应 , 则 RPC 调用者可能会最终依赖这些值的顺序。

这是 “ 它有效 ” 和 “ 它正确 ” 之间区别的一个非常基本的例子。

对于一个短期程序 , 依赖容器的迭代顺序不会造成任何技术问题。另一方面 , 对于软件工程项目来说 , 这种对既定顺序的依赖是一种风险。如果有足够的时间 , 某些事情就会发生

¹⁰ 一种拒绝服务 (DoS) 攻击 , 其中不受信任的用户知道哈希表的结构 , 并且哈希函数并以降低表上操作的算法性能的方式提供数据。

让改变迭代顺序变得有价值。这种价值可以体现在很多方面,无论是效率、安全性,还是仅仅是让数据结构面向未来以应对未来的变化。当这种价值变得清晰时,您需要权衡这种价值与破坏开发人员或客户的痛苦之间的权衡。

一些语言专门在库版本之间或甚至在同一程序的执行之间随机化哈希顺序,以防止依赖性。

但即便如此,海伦定律仍然会带来一些意外:有些代码使用哈希迭代排序作为低效的随机数生成器。现在删除这种随机性会破坏这些用户。正如每个热力学系统中的熵都会增加一样,海伦定律适用于所有可观察到的行为。

通过思考以“立即可用”和“无限期可用”的心态编写的代码之间的区别,我们可以提取出一些明确的关系。将代码视为具有(高度)可变的生命周期要求的工作,我们可以开始对编程风格进行分类:依赖于其依赖项的脆弱和未发布功能的代码可能被描述为“黑客”或“聪明”,而遵循最佳实践并为未来做好规划的代码更有可能被描述为“干净”和“可维护”。两者都有其目的,但选择哪一种主要取决于相关代码的预期生命周期。我们习惯说,“如果‘聪明’是一种赞美,那就是编程,如果‘聪明’是一种指责,那就是软件工程。”

为什么不直接以“不改变”为目标呢?

所有这些关于时间和应对变化的必要性的讨论都隐含着这样的假设:变化可能是必要的。是吗?

正如本书中的其他所有内容一样,这取决于具体情况。我们很乐意承诺“对于大多数项目,在足够长的时间内,其底层的所有内容都可能需要更改。”如果您有一个用纯C编写的项目,没有外部依赖项(或者只有保证长期稳定性的外部依赖项,如POSIX),您很可能能够避免任何形式的重构或困难的升级。C在提供稳定性方面做得很好 在许多方面,这是它的主要目的。

大多数项目都更容易受到不断变化的底层技术的影响。大多数编程语言和运行时的变化都比C大得多。即使是用纯C实现的库也可能会发生变化以支持新功能,这可能会影响下游用户。从处理器到网络库再到应用程序代码,各种技术都存在安全问题。你的项目所依赖的每一项技术都存在一些(希望是很小的)风险,可能包含严重的错误和安全漏洞,而这些风险可能只有在你开始依赖它之后才会暴露出来。如果你无法部署Heartbleed补丁或减轻

推测执行问题,例如[Meltdown](#) 和 [Spectre](#)因为你已经假设 (或承诺)什么都不会改变,所以这是一场重大的赌博。

效率改进使情况进一步复杂化。我们希望为我们的数据中心配备具有成本效益的计算设备,尤其是提高 CPU 效率。然而,早期 Google 的算法和数据结构在现代设备上效率较低:链表或二叉搜索树仍然可以正常工作,但 CPU 周期与内存延迟之间不断扩大的差距影响了“高效”代码的外观。随着时间的推移,升级到较新的硬件的价值可能会降低,而无需对软件进行设计更改。向后兼容性可确保旧系统仍能运行,但这并不能保证旧的优化仍然有用。不愿意或无法利用这样的机会可能会产生巨大的成本。像这样的效率问题特别微妙:原始设计可能完全合乎逻辑并遵循合理的最佳实践。只有在向后兼容的更改之后,新的、更高效的选择才变得重要。没有犯任何错误,但时间的流逝仍然使改变变得有价值。

上述担忧就是为什么长期项目没有投资于可持续性会面临巨大风险的原因。我们必须有能力应对这些问题并利用这些机会,无论它们是直接影响我们还是仅仅体现在我们赖以建立的技术的过渡闭包中。改变本身并不是一件好事。我们不应该仅仅为了改变而改变。但我们确实需要有能力改变。如果我们考虑到最终的必要性,我们还应该考虑是否投资于降低这种能力的成本。

每位系统管理员都知道,理论上知道可以从磁带恢复是一回事,而实际操作中确切知道如何做以及必要时需要花费多少钱又是另一回事。实践和专业知识是提高效率和可靠性的重要驱动力。

规模与效率

正如《站点可靠性工程 (SRE)》一书¹¹所述, Google 的整体生产系统是人类创造的最复杂的机器之一。构建这样一台机器并保持其平稳运行的复杂性需要我们组织内和全球各地的专家花费无数小时进行思考、讨论和重新设计。因此,我们已经写了一本书,介绍如何让这台机器以如此大规模运行的复杂性。

¹¹ Beyer, B. 等人。[站点可靠性工程:Google 如何运行生产系统](#)。 (波士顿:O'Reilly Media, 2016 年)。

本书的大部分内容都集中在生产这样一台机器的组织的规模复杂性,以及我们用来保持这台机器长期运转的流程上。再次考虑代码库可持续性的概念:“当您能够安全地更改所有应该更改的内容,并且可以在代码库的整个生命周期内进行更改时,您的组织的代码库就是可持续的。”在对能力的讨论中还隐藏着成本之一:如果更改某件事的成本过高,那么很可能可能会被推迟。如果成本随着时间的推移呈超线性增长,那么该操作显然不可扩展。¹²最终,时间会控制住局面,出现一些你必须改变的意外情况。当您的项目范围扩大一倍,您需要再次执行该任务时,所需的劳动力是否会加倍?您是否有足够的人力资源来下次解决这个问题?

人力成本并不是唯一需要扩展的有限资源。正如软件本身需要与计算、内存、存储和带宽等传统资源很好地扩展一样,该软件的开发也需要扩展,无论是在人力投入方面,还是在支持开发工作流程的计算资源方面。如果测试集群的计算成本呈超线性增长,每季度消耗的计算资源更多,那么您就走上了一条不可持续的道路,需要尽快做出改变。

最后,软件组织最宝贵的资产——代码库本身——也需要扩展。如果您的构建系统或版本控制系统随着时间的推移超线性扩展,可能是由于增长和更改日志历史记录的增加,那么您可能会到达无法继续的地步。许多问题,例如“完成完整构建需要多长时间?”,“提取存储库的新副本需要多长时间?”或“升级到新语言版本需要多少钱?”没有得到积极监控,并且变化速度很慢。它们很容易变成**比喻中的煮青蛙**,问题很容易慢慢恶化,永远不会在某个危机时刻显现出来。只有全组织都意识到问题并致力于扩大规模,你才有可能控制住这些问题。

您的组织所依赖的用于生成和维护代码的一切都应在总体成本和资源消耗方面具有可扩展性。特别是,您的组织必须重复做的所有事情都应在人力投入方面具有可扩展性。

从这个意义上来说,许多常见的政策似乎不具有可扩展性。

无法扩展的政策

经过一些练习,就会更容易发现具有不良扩展特性的策略。

最常见的是,可以通过考虑对单个人施加的工作来识别

¹² 在本章中,当我们在非正式语境中使用“可扩展”时,我们的意思是“关于人际交往”。

工程师，并想象组织规模扩大 10 倍或 100 倍。当我们的规模扩大 10 倍时，我们是否会增加 10 倍的工作量，而我们的样本工程师需要跟上这些工作量？我们的工程师必须执行的工作量是否会随着组织规模的扩大而增长？工作量是否会随着代码库的规模而扩大？如果上述任何一种情况属实，我们是否有任何机制来自动化或优化这项工作？如果没有，我们就会面临扩展问题。

考虑一种传统的弃用方法。我们在[第 15 章中更多地讨论了弃用](#)，但弃用的常见方法是扩展问题的一个很好的例子。开发了一个新的 Widget，决定每个人都应该使用新的，停止使用旧的。为了激励这一点，项目负责人说“我们将在 8 月 15 日删除旧的 Widget；确保您已转换为新的 Widget。”

这种方法可能在小型软件环境中有效，但随着依赖关系图的深度和广度的增加，很快就会失败。团队依赖越来越多的小部件，一次构建中断可能会影响公司越来越多的部分。以可扩展的方式解决这些问题意味着改变我们进行弃用的方式：团队可以自行完成迁移工作，而不是将迁移工作推给客户，从而获得规模经济。

2012 年，我们尝试通过制定规则来阻止这种情况：基础架构团队必须自行完成将内部用户迁移到新版本的工作，或者以向后兼容的方式就地进行更新。这项政策被我们称为“用户流失规则”，它具有更好的可扩展性：相关项目不再需要花费越来越多的精力来跟上时代的步伐。我们还了解到，拥有一个专门的专家组来执行变更比要求每个用户付出更多的维护努力更具可扩展性：专家会花一些时间深入了解整个问题，然后将这些专业知识应用于每个子问题。强迫用户对用户流失做出反应意味着每个受影响的团队在提升工作效率、解决眼前问题和丢弃那些现在毫无用处的知识方面都会做得更糟。专业知识具有更好的可扩展性。

开发分支的传统使用是另一个具有内在扩展问题的策略示例。组织可能会发现将大型功能合并到主干中会破坏产品的稳定性，并得出结论：“我们需要对合并时间进行更严格的控制。我们应该减少合并频率。”这很快导致每个团队或每个功能都有单独的开发分支。每当决定任何分支“完成”时，都会对其进行测试并合并到主干中，从而以重新同步和测试的形式触发仍在开发其开发分支的其他工程师进行一些可能代价高昂的工作。这种分支管理可以适用于同时处理 5 到 10 个此类分支的小型组织。随着组织规模（和分支数量）的增加，我们很快就会发现，我们为完成相同任务支付的开销越来越大。随着规模的扩大，我们需要一种不同的方法，我们将在[第 16 章中讨论这个问题](#)。

可扩展的政策

随着组织的发展,什么样的政策可以降低成本?或者更好的是,我们可以实施什么样的政策,在组织发展过程中提供超线性价值?

我们最喜欢的内部政策之一是基础设施团队的强大推动者,保护他们安全地进行基础设施变更的能力。“如果产品由于基础设施变更而出现中断或其他问题,但该问题并未在我们的持续集成(CI)系统中的测试中浮现,那么这不是基础设施变更的错。”更通俗地说,这句话是“如果你喜欢它,你就应该对它进行CI测试”,我们称之为“碧昂丝法则”。¹³从扩展的角度来看,碧昂丝法则意味着复杂的、一次性的定制测试,如果不是由我们的通用CI系统触发的,就不算数。如果没有这条规则,基础设施团队的工程师可能需要追踪每个受影响代码的团队,并询问他们如何运行测试。当有100名工程师时,我们可以这样做。但我们绝对不能再这样做了。

我们发现,随着组织规模的扩大,专业知识和共享交流论坛具有巨大的价值。随着工程师在共享论坛上讨论和回答问题,知识往往会传播开来。新专家不断涌现。如果你有一百名工程师编写Java,那么一位友好而乐于助人的Java专家愿意回答问题,很快就会产生一百名工程师编写更好的Java代码。知识是病毒式的,专家是传播者,为工程师清除常见绊脚石的价值值得一提。我们将在第3章中更详细地介绍这一点。

示例:编译器升级考虑一下升级编

译器这项艰巨的任务。从理论上讲,考虑到语言向后兼容需要付出的努力,编译器升级应该很便宜,但实际上这项操作有多便宜?如果您以前从未进行过这样的升级,您将如何评估您的代码库是否与该更改兼容?

¹³这是对流行歌曲“单身女士”的引用,其中包括这样的歌词“如果你喜欢它,那么你就应该给它戴上戒指。”

根据我们的经验,语言和编译器升级是一项微妙而困难的任务,即使人们普遍期望它们向后兼容。编译器升级几乎总是会导致行为发生微小变化:修复错误编译、调整优化或可能更改任何以前未定义的结果。您将如何评估整个代码库相对于所有这些潜在结果的正确性?

Google 历史上最传奇的编译器升级发生在 2006 年。那时,我们已经运营了几年,拥有几千名工程师。我们大约有 5 年没有更新过编译器了。我们的大多数工程师都没有处理过编译器变更的经验。我们的大多数代码都只接触过一个编译器版本。对于一个(主要)志愿者组成的团队来说,这是一项艰巨而痛苦的任务,最终变成了寻找捷径和简化方法,以便绕过我们不知道如何采用的上游编译器和语言变化。¹⁴最后,2006 年的编译器升级极其痛苦。许多大大小小的海勒姆定律问题都潜入了代码库,加深了我们对特定编译器版本的依赖。打破这些隐式依赖非常痛苦。这些工程师承担了一定的风险:我们还没有碧昂斯规则,也没有普及的持续集成系统,因此很难提前知道变化的影响,或者确保他们不会因回归而受到指责。

这个故事一点也不罕见。许多公司的工程师都可以讲述类似的痛苦升级故事。不同寻常的是,我们事后才意识到这项任务很痛苦,并开始专注于技术和组织变革,以克服扩展问题并将规模转化为我们的优势:自动化(这样一个人就可以做更多的事情)、整合/一致性(这样低级更改的问题范围就有限了)和专业知识(这样少数人就可以做更多的事情)。

您更改基础设施的频率越高,更改就越容易。

我们发现,大多数情况下,当代码作为编译器升级等操作的一部分进行更新时,它会变得更可靠,将来更容易升级。在大多数代码都经过多次升级的生态系统中,它不再依赖于底层实现的细微差别;相反,它依赖于语言或操作系统保证的实际抽象。无论您要升级的是什么,即使考虑到其他因素,代码库的第一次升级也会比以后的升级昂贵得多。

¹⁴具体来说,需要在命名空间 std 中引用 C++ 标准库中的接口,而对 std::string 的优化更改对我们的使用来说是一个严重的悲观因素,因此需要一些额外的解决方法。

通过这次和其他的经验,我们发现了许多影响代码库灵活性的因素:

专业知识

我们知道如何做到这一点;对于某些语言,我们现在已经在许多平台上完成了数百次编译器升级。

稳定性 由

于我们更定期地采用发布版本,因此版本之间的变化较少;对于某些语言,我们现在每周或每两周部署一次编译器升级。

一致性 由于我

们定期升级,因此尚未升级的代码较少。

熟悉度 由于我

们经常这样做,因此我们可以在升级过程中发现冗余并尝试自动化。这与 SRE 对琐事的看法有很大的重叠。

15

政策我

们有像碧昂丝规则这样的流程和政策。这些流程的最终效果是升级仍然可行,因为基础设施团队不需要担心每个未知的用途,只需担心在我们的 CI 系统中可见的用途。

根本的教训不是关于编译器升级的频率或难度,而是一旦我们意识到编译器升级任务是必要的,我们就会找到方法确保以恒定数量的工程师执行这些任务,即使代码库增长也是如此。¹⁶如果我们决定这项任务过于昂贵并且应该在将来避免,我们可能仍在使用十年前的编译器版本。由于错过了优化机会,我们可能会为计算资源多支付 25%。鉴于 2006 年的编译器肯定无助于缓解推测执行漏洞,我们的中央基础设施可能容易受到重大安全风险的攻击。停滞不前是一种选择,但往往不是明智的选择。

15 Beyer 等人,《站点可靠性工程:Google 如何运行生产系统》,第 5 章“消除辛劳”。

16 根据我们的经验,一名普通软件工程师 (SWE) 每单位时间编写的代码行数相当稳定。对于固定数量的 SWE 人员,代码库会呈线性增长 与 SWE 月数成正比。如果您的任务需要的工作量与代码行数成正比,那就令人担忧了。

左移我们发现的

一个普遍事实是，在开发人员工作流程中尽早发现问题通常可以降低成本。考虑一个功能的开发人员工作流程的时间线，从左到右进行，从构思和设计开始，经过实施、审查、测试、提交、金丝雀发布，最终进行生产部署。将问题检测在此时间线上向“左”移动，可以比等待更长时间更便宜地修复问题，如图 1-2 所示。

这一术语似乎源于这样的论点：安全性不能推迟到开发过程的最后，并有必要呼吁“安全性左移”。

在这种情况下，争论相对简单：如果安全问题在产品投入生产后才被发现，那么成本将非常高。如果在部署到生产之前就发现问题，可能仍需要大量工作来识别和解决问题，但成本较低。如果能在原始开发人员将缺陷提交到版本控制之前发现问题，成本甚至更低：他们已经了解该功能；根据新的安全约束进行修改比提交并强迫其他人对其进行分类和修复要便宜。

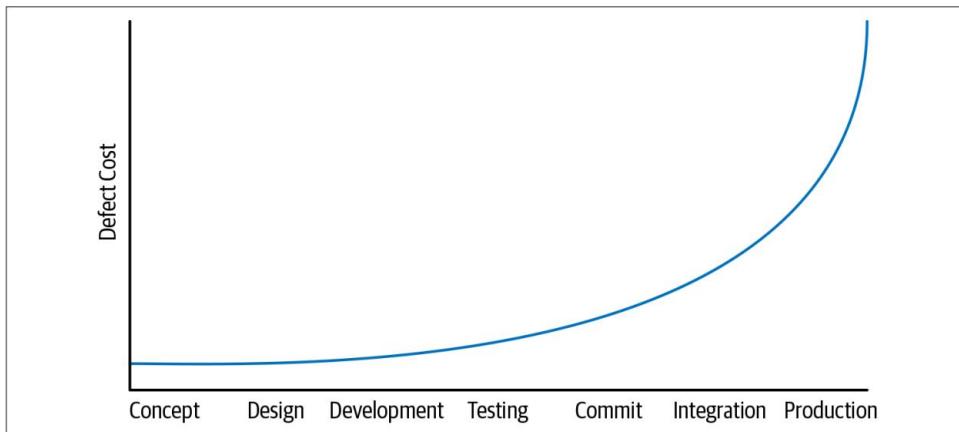


图 1-2. 开发人员工作流程的时间线

本书多次出现相同的基本模式。在提交之前通过静态分析和代码审查发现的错误比投入生产的错误要便宜得多。在开发过程的早期提供强调质量、可靠性和安全性的工具和实践是我们许多基础设施团队的共同目标。没有一个流程或工具需要完美无缺，因此我们可以假设一种纵深防御方法，希望尽可能多地捕获图表左侧的缺陷。

权衡与成本

如果我们了解如何编程,了解我们所维护的软件的生命周期,并了解如何在规模扩大、有更多工程师生产和维护新功能时维护它,那么剩下要做的就是做出正确的决策。

这似乎很明显:在软件工程中,就像在生活中一样,好的选择会带来好的结果。然而,这一观察的后果很容易被忽视。在 Google 内部,人们非常讨厌“因为我这么说”。当决策似乎错误时,任何话题都需要有一个决策者和明确的升级路径,这一点很重要,但目标是达成共识,而不是一致同意。看到一些“我不同意你的指标/估值,但我明白你是如何得出这个结论的”的例子是正常的,也是意料之中的。所有这些都固有这样一种观念,即每件事都需要有理由;“只是因为”、“因为我这么说”或“因为其他人都这样做”是糟糕决策的潜伏之地。只要这样做是有效的,我们就应该能够在两个工程选项的一般成本之间做出决定时解释我们的工作。

我们所说的成本是什么意思?我们在这里谈论的不仅仅是金钱。“成本”大致可以理解为努力,可能涉及以下任何或所有因素:

- 财务成本(例如金钱) · 资源成本(例如 CPU 时间)
- 人员成本(例如工程工作量) · 交易成本(例如采取行动的成本是多少?)
- 机会成本(例如不采取行动的成本是多少?) · 社会成本(例如这种选择会对整个社会产生什么影响?)

从历史上看,人们特别容易忽视社会成本问题。然而,谷歌和其他大型科技公司现在可以可靠地部署拥有数十亿用户的产品。在许多情况下,这些产品具有明显的净收益,但当我们以如此大的规模运营时,即使是可用性、可访问性、公平性或滥用可能性方面的微小差异也会被放大,往往损害已经被边缘化的群体。软件渗透到社会和文化的许多方面;因此,在做出产品和技术决策时,我们应该意识到我们所带来的好处和坏处。我们将在第 4 章中对此进行更多讨论。

除了上述成本(或我们对这些成本的估计)之外,还有偏见:现状偏见、损失厌恶等。当我们评估成本时,我们需要记住前面列出的所有成本:一个组织的健康不仅取决于银行里是否有钱,还取决于其成员是否感到有价值和富有成效。在软件工程等极具创造力和利润丰厚的领域,财务成本通常不是限制因素 人员成本通常是限制因素。效率的提高来自

让工程师感到快乐、专注和投入很容易压倒其他因素,这仅仅是因为专注度和生产力变化很大,而且 10% 到 20% 的差异是很容易想象的。

示例:记号笔在许多组织

中,白板笔被视为贵重物品。它们受到严格控制,并且总是供不应求。任何白板上的记号笔中,几乎一半都是干的,无法使用。您参加的会议中,有多少次因为没有可用的记号笔而中断?您的思路有多少次因为一支记号笔用完了而偏离了轨道?有多少次所有的记号笔都不见了,大概是因为其他团队的记号笔用完了,不得不带着您的记号笔逃走?所有这些都是为了一款售价不到一美元的产品。

Google 倾向于在大多数工作区域放置未上锁的柜子,里面装满了办公用品,包括白板笔。只要一有时间,很容易就能拿到几十支不同颜色的记号笔。在某个时候,我们做出了一个明确的权衡:优化无障碍头脑风暴远比防止有人拿着一堆记号笔四处游荡更重要。

我们的目标是对我们所做的每一件事都保持同样的清醒和清晰的权衡成本/收益,从办公用品和员工福利到开发人员的日常体验,再到如何配置和运行全球规模的服务。我们常说,“谷歌是一种数据驱动的文化。”事实上,这是一种简化:即使没有数据,也可能有证据、先例和论据。做出好的工程决策就是权衡所有可用的输入,并就权衡做出明智的决定。有时,这些决定是基于直觉或公认的最佳实践,但只有在我们用尽了试图衡量或估计真实潜在成本的方法之后。

最后,工程团队的决策应该归结为以下几点:

- 我们这样做是因为我们必须这样做 (法律要求、客户要求)。
- 我们这样做是因为根据目前的证据,这是我们当时能看到的最佳选择 (由一些适当的决策者决定)。

决策不应该是“我们这么做是因为我这么说。”¹⁷

¹⁷这并不是说决策需要一致通过,甚至不需要广泛的共识;最终,必须有人来做决策。这主要是对决策过程应该如何进行的陈述,无论谁真正负责决策。

决策输入

当我们权衡数据时,我们发现两种常见的情况:

- 所有涉及的数量都是可测量的,或者至少可以估算。这通常意味着我们正在评估 CPU 和网络、美元和 RAM 之间的权衡,或者考虑是否要花费两周的工程师时间来节省我们数据中心的 N 个 CPU。
- 有些数量很微妙,或者我们不知道如何测量它们。有时这表现为“我们不知道这需要多少工程师时间。”

有时甚至更加模糊:如何衡量设计不良的 API 的工程成本?或者产品选择的社会影响?

在第一类决策上,没有理由不负责。任何软件工程组织都可以并且应该跟踪计算资源、工程师小时数以及您经常接触的其他数量的当前成本。即使您不想向组织公布确切的金额,您仍然可以制作一个转换表:这么多 CPU 的成本与这么多 RAM 或这么多网络带宽的成本相同。

有了商定的转换表,每个工程师都可以进行自己的分析。“如果我花两周时间将这个链表更改为更高性能的结构,我将使用 5 GB 以上的生产 RAM,但节省 2,000 个 CPU。我应该这样做吗?”这个问题不仅取决于 RAM 和 CPU 的相对成本,还取决于人员成本(软件工程师两周的支持)和机会成本(该工程师在两周内还能生产出什么?)。

对于第二类决策,没有简单的答案。我们依靠经验、领导力和先例来协商这些问题。我们正在投资研究,以帮助我们量化难以量化的事务(见第 7 章)。然而,我们最好的建议是意识到并非所有事情都是可衡量或可预测的,并尝试以同样的优先级和更大的谨慎来对待此类决策。它们通常同样重要,但更难管理。

示例:分布式构建考虑您的构建。

根据完全不科学的 Twitter 民意调查,大约 60% 到 70% 的开发人员在本地构建,即使是当今大型、复杂的构建也是如此。

这直接引出了非笑话,正如[这部“编译”漫画所说明的那样](#) 您的组织在等待构建时浪费了多少生产时间?将其与小团队运行 distcc 之类的东西的成本进行比较。或者,为大团队运行小型构建农场需要多少成本?这些成本需要多少周/月才能实现净收益?

早在 2000 年代中期,Google 完全依赖于本地构建系统:您签出代码并在本地编译。在某些情况下,我们拥有大量本地机器 (您可以在桌面上构建地图!),但随着代码库的增长,编译时间变得越来越长。毫不奇怪,由于时间的浪费,我们的人员成本不断增加,更大、更强大的本地机器的资源成本也不断增加,等等。这些资源成本特别麻烦:当然,我们希望人们能够尽可能快地构建,但大多数时候,高性能桌面开发机器都会闲置。这似乎不是投资这些资源的正确方式。

最终,Google 开发了自己的分布式构建系统。当然,开发这个系统需要付出代价:工程师需要花费时间进行开发,需要花费更多时间来改变个人的习惯和工作流程并学习新系统,当然还需要额外的计算资源。但总体节省显然是值得的:构建速度更快,工程师的时间得到了补偿,硬件投资可以集中在托管共享基础设施 (实际上是我们生产机群的一个子集) 上,而不是功能越来越强大的台式机上。[第 18 章](#) 详细介绍了我们的分布式构建方法及其相关的权衡。

因此,我们构建了一个新系统,将其部署到生产环境中,并加快了每个人的构建速度。这是故事的圆满结局吗?不完全是:提供分布式构建系统极大地提高了工程师的工作效率,但随着时间的推移,分布式构建本身变得臃肿。在前一种情况下,个别工程师受到限制 (因为他们有既得利益,希望尽可能快地进行本地构建),而在分布式构建系统中则不受限制。构建图中臃肿或不必要的依赖关系变得非常普遍。当每个人都直接感受到非最佳构建的痛苦并受到激励保持警惕时,激励机制会更好地协调一致。通过消除这些激励措施并在并行分布式构建中隐藏臃肿的依赖关系,我们创造了一种消耗可能泛滥的情况,几乎没有人受到激励去关注构建臃肿。这让人想起[杰文斯悖论](#):由于资源使用效率的提高,资源消耗可能会增加。

总体而言,添加分布式构建系统所节省的成本远远超过了其构建和维护所带来的负面成本。但是,正如我们看到的消费增加一样,我们并没有预见到所有这些成本。在向前迈进之后,我们发现自己处于一种需要重新概念化系统和我们的使用目标和约束、确定最佳实践 (小依赖关系、机器管理依赖关系) 以及为新生态系统的工具和维护提供资金的境地。即使是“我们将花费 \$\$\$ 购买计算资源以弥补工程师时间”这种相对简单的权衡也会产生无法预见的下游影响。

示例 : 在时间和规模之间做出选择很多时候 , 我们的

主要主题 时间和规模 是重叠的 , 并且协同工作。像碧昂丝规则这样的政策具有很好的可扩展性 , 并有助于我们随着时间的推移保持事物。对操作系统接口的更改可能需要进行许多小的重构才能适应 , 但大多数更改都将具有很好的可扩展性 , 因为它们的形式相似 : 操作系统更改对每个调用者和每个项目的表现并没有什么不同。

有时时间和规模会发生冲突 , 而没有什么比这个基本问题更明显的了 : 我们应该添加依赖项还是分叉 / 重新实现它以更好地满足我们当地的需求 ?

这个问题可能出现在软件堆栈的许多层面上 , 因为通常情况下 , 针对您的狭窄问题空间定制的解决方案可能比需要处理所有可能性的通用实用程序解决方案表现更好。通过分叉或重新实现实用程序代码并针对您的狭窄领域进行定制 , 您可以更轻松地添加新功能 , 或更确定地进行优化 , 无论我们谈论的是微服务、内存缓存、压缩例程还是软件生态系统中的其他任何东西。也许更重要的是 , 您从这种分叉中获得的控制权将您与底层依赖项的更改隔离开来 : 这些更改不是由其他团队或第三方提供商决定的。

您可以控制如何以及何时对时间的流逝和改变的必要性做出反应。

另一方面 , 如果每个开发人员都分叉其软件项目中使用的所有内容 , 而不是重复使用现有的内容 , 那么可扩展性和可持续性都会受到影响。应对底层库中的安全问题不再是更新单个依赖项及其用户的问题 : 现在需要识别该依赖项的每个易受攻击的分支以及这些分支的用户。

与大多数软件工程决策一样 , 这种情况没有一刀切的答案。如果你的项目寿命很短 , 分叉的风险就小一些。如果所讨论的分叉范围有限 , 那么这也会有所帮助 避免跨时间或项目时间边界 (数据结构、序列化格式、网络协议) 操作的接口分叉。一致性具有很大的价值 , 但通用性也有其自身的成本 , 如果你小心谨慎 , 你通常可以通过做自己的事情来取胜。

重新审视决策 , 犯错致力于数据驱动文化的其

中一个不为人知的好处是承认错误的能力和必要性。决策将在某个时候根据可用数据做出 希望是基于良好的数据和一些假设 , 但隐含地基于当前可用的数据。随着新数据的出现、上下文的变化或假设被消除 , 可能会清楚地表明决策是错误的

错误,或者当时有意义但现在不再有意义。这对于一个长期存在的组织来说尤其重要:时间不仅会引发技术依赖性和软件系统的变化,还会引发用于推动决策的数据的变化。

我们坚信数据可以为决策提供参考,但我们也认识到数据会随着时间而变化,新数据也会出现。这本质上意味着,在相关系统的整个生命周期中,决策需要不时重新审视。对于长期项目,在做出初步决策后,有能力改变方向往往至关重要。而且,重要的是,这意味着决策者需要有承认错误的权利。与某些人的直觉相反,承认错误的领导者更受尊重,而不是更少。

要以证据为导向,但也要意识到无法衡量的事物可能仍然有价值。如果你是一名领导者,这就是你被要求做的:运用判断力,断言事情很重要。我们将在第5章和第6章中更多地讨论领导力。

软件工程与编程

当我们将软件工程与编程区分开来时,你可能会问,这其中是否存在内在的价值判断。编程难道比软件工程更糟糕吗?一个由数百人组成的团队预计持续十年的项目难道就一定比一个只能使用一个月、由两个人开发的项目更有价值吗?

当然不是。我们的观点不是软件工程更优越,只是它们代表了两个不同的问题领域,具有不同的约束、价值观和最佳实践。相反,指出这种差异的价值在于认识到某些工具在一个领域很棒,但在另一个领域却不然。对于只持续几天的项目,您可能不需要依赖集成测试(参见第14章)和持续部署(CD)实践(参见第24章)。同样,我们对软件工程项目中的语义版本控制(SemVer)和依赖管理的所有长期关注(参见第21章)并不适用于短期编程项目:使用任何可用的东西来解决手头的任务。

我们认为,区分“编程”和“软件工程”这两个相关但不同的术语非常重要。这种差异很大程度上源于对代码的长期管理、时间对规模的影响以及面对这些想法的决策。编程是直接生成代码的行为。软件工程是一套必要的政策、实践和工具,以使代码在需要使用时有用,并允许跨团队协作。

团队。

结论

本书讨论了所有这些主题:针对组织和单个程序员的政策、如何评估和改进最佳实践,以及可维护软件所需的工具和技术。Google一直努力打造可持续的代码库和文化。我们不一定认为我们的方法是唯一正确的做事方式,但它确实通过实例证明了我们可以做到这一点。我们希望它能为思考一般问题提供一个有用的框架:如何在代码需要继续工作的情况下对其进行维护?

TL;DR

- “软件工程”与“编程”在维度上有所不同:编程就是编写代码。软件工程将其扩展为在代码的有效生命周期内对其进行维护。
- 短寿命代码和长寿命代码的寿命至少相差 100,000 倍。假设相同的最佳实践普遍适用于该范围的两端是愚蠢的。
- 如果在代码的预期生命周期内,我们能够应对依赖关系、技术或产品需求的变化,那么软件就是可持续的。我们可能选择不做任何改变,但我们必须具备这种能力。
- 海勒姆定律:如果 API 有足够数量的用户,那么你在合同中做出什么承诺都并不重要:你的系统所有可观察到的行为都将由某人依赖。
- 您的组织必须重复执行的每项任务在人力投入方面都应具有可扩展性(线性或更佳)。政策是使流程具有可扩展性的极佳工具。
- 流程效率低下和其他软件开发任务往往会扩大
慢慢来。小心煮青蛙的问题。
- 专业知识与规模经济相结合时,回报尤其丰厚。· “因为我这么说”并不是做事的糟糕理由。· 以数据为导向是一个好的开始,但实际上,大多数决策都是基于数据、假设、
先例和论据的混合。客观数据占这些输入的大部分,这是最好的,但很少能全部。
- 随着时间的推移,数据驱动意味着需要在数据发生变化时(或假设被推翻时)改变方向。错误或修改计划是不可避免的。

第二部分

文化

第二章

如何在团队中良好工作

作者:布莱恩·菲茨帕特里克
编辑:Riona MacNamara

因为本章是关于 Google 软件工程的文化和社会方面,所以首先关注一个你肯定能控制的变量是有意义的:你自己。

人天生就不完美 我们喜欢说人类大多是间歇性缺陷的集合。但在你了解同事的缺陷之前,你需要了解自己的缺陷。我们将要求你思考自己的反应、行为和态度 作为回报,我们希望你能真正了解如何成为一名更高效、更成功的软件工程师,花更少的精力处理与人相关的问题,花更多的时间编写出色的代码。

本章的关键思想是软件开发是一项团队工作。要想在工程团队或任何其他创造性合作中取得成功,你需要围绕谦逊、尊重和

相信。

在我们开始之前,让我们先观察一下软件工程师的一般行为方式。

帮我隐藏我的代码

在过去的 20 年里,我和我的同事 Ben¹ 在许多编程会议上发表过演讲。2006 年,我们启动了 Google 的(现已弃用的)开源项目

¹ Ben Collins-Sussman 也是本书的作者之一。

托管服务,起初,我们经常收到很多关于该产品的问题和请求。但在 2008 年中期,我们开始注意到我们收到的请求类型有一种趋势:

“您能让 Google Code 上的 Subversion 隐藏特定分支吗?”

“你能否创建一些一开始不为人知,然后在准备就绪时才公开的开源项目?”

“你好,我想从头重写我的所有代码,你能清除所有历史记录吗?”

您能发现这些请求的共同主题吗?

答案就是不安全感。人们害怕别人看到并评判他们正在进行的工作。从某种意义上说,不安全感只是人性的一部分 没有人喜欢被批评,尤其是那些尚未完成的事情。认识到这个主题让我们意识到软件开发中一个更普遍的趋势:不安全感实际上是一个更大问题的症状。

天才神话

很多人天生就有寻找和崇拜偶像的本能。对于软件工程师来说,偶像可能是 Linus Torvalds、Guido Van Rossum、Bill Gates 他们都是靠英勇事迹改变世界的英雄。Linux 是 Linus 自己写的吧?

实际上,Linus 所做的只是编写了概念验证类 Unix 内核的初始版本,并将其展示给电子邮件列表。这是一项不小的成就,而且绝对是一项令人印象深刻的成就,但这只是冰山一角。Linux 比最初的内核大数百倍,是由数千名聪明人开发的。Linus 真正的成就是领导这些人并协调他们的工作;Linux 不是他最初的想法的辉煌成果,而是社区集体劳动的成果。(Unix 本身并非完全由 Ken Thompson 和 Dennis Ritchie 编写,而是由贝尔实验室的一群聪明人编写的。)

同样,Guido Van Rossum 是否亲自编写了 Python 的全部内容?当然,他编写了第一个版本。但数百人为后续版本做出了贡献,包括创意、功能和错误修复。史蒂夫·乔布斯领导了整个团队开发 Macintosh,尽管比尔·盖茨因编写早期家用电脑的 BASIC 解释器而闻名,但他更大的成就是围绕 MS-DOS 建立了一家成功的公司。然而,他们都成为了社区集体成就的领导者和象征。天才神话是我们人类倾向于将团队的成功归功于一个人/领导者。

那么迈克尔·乔丹又怎么样?

故事还是一样。我们崇拜他,但事实上他并不是靠一己之力赢得每一场比赛的。他真正的天才在于他与团队合作的方式。球队教练菲尔·杰克逊非常聪明,他的执教技巧堪称传奇。

他意识到单凭一个球员是无法赢得总冠军的,因此他围绕乔丹组建了一支“梦之队”。这支队伍就像一台运转良好的机器 至少和迈克尔本人一样令人印象深刻。

那么,为什么我们一再崇拜这些故事中的人物呢?为什么人们会购买名人代言的产品?为什么我们要购买米歇尔·奥巴马的裙子或迈克尔·乔丹的鞋子?

名人是其中很重要的一部分。人类有一种本能,会寻找领导者和榜样,崇拜他们,并试图模仿他们。我们都需要英雄来激励自己,编程界也有自己的英雄。“技术名人”现象几乎已经蔓延到神话中。我们都想写出一些改变世界的东西,比如 Linux,或者设计出下一种出色的编程语言。

许多工程师内心深处都暗自希望自己被视为天才。这种幻想大致如下:

- 你被一个很棒的新概念所震撼。 · 你躲在自己的洞穴里数周或数月,努力实现自己的想法。
- 然后你向世界“发布”你的软件,用你的天才。
- 同行对您的聪明才智感到惊讶。 · 人们排队使用您的软件。 · 名利自然而然地随之而来。

但是请稍等一下:是时候认清现实了。你可能不是天才。

当然,我没有冒犯的意思 我们相信你是一个非常聪明的人。但你知道真正的天才有多么罕见吗?当然,你会写代码,这是一项棘手的技能。但即使你是个天才,这还不够。天才也会犯错,拥有绝妙的想法和精英编程技能并不能保证你的软件会大获成功。更糟糕的是,你可能会发现自己只解决分析问题,而不解决人为问题。天才绝对不是做混蛋的借口:任何人 无论是不是天才 如果社交能力较差,往往会被认为是一个糟糕的朋友。谷歌(以及大多数公司!)的绝大多数工作不需要天才级别的智力,但 100% 的工作都需要最低水平的社交技能。成就或毁掉你的职业生涯的,尤其是在像谷歌这样的公司,取决于你与他人的合作能力。

事实证明,这个天才神话只是我们不安全感的另一种表现。

许多程序员不敢分享他们刚开始的工作,因为这意味着同行会看到他们的错误,并知道代码的作者并不是天才。

引用一位朋友的话：

我知道,如果别人在做某件事之前先查看,我会感到非常不安。
就好像他们会严肃地评判我并认为我是个白痴。

这是程序员中极为普遍的一种感觉,而自然的反应就是躲在山洞里,工作、工作、工作,然后不断完善、完善、完善,确保没有人会看到你的失误,并且当你完成后你仍然有机会揭开你的杰作。躲起来直到你的代码完美无缺。

隐藏工作成果的另一个常见动机是担心其他程序员可能会在你开始工作之前窃取你的想法并付诸实施。通过保密,你可以控制这个想法。

我们知道你现在可能在想什么:那又怎么样?难道人们不应该被允许以他们想要的方式工作吗?

事实上,不是的。在这种情况下,我们断言你做错了,这是一件大事。
原因如下。

隐藏有害

如果你把所有的时间都花在独自工作上,那么你就会增加不必要的失败风险,并错失你的成长潜力。尽管软件开发是一项需要高度集中精力和独处时间的智力工作,但你必须权衡协作和审查的价值(和需求!)。

首先,你怎么知道你是否走在正确的道路上?

想象一下,你是一名自行车设计爱好者,有一天,你突然想到一个绝妙的主意,想出一种全新的变速杆设计方法。你订购了零件,然后花了数周时间躲在车库里,试图制造一个原型。当你的邻居(也是自行车爱好者)问你发生了什么事时,你决定不谈论这件事。你不想让任何人知道你的项目,直到它绝对完美为止。又过了几个月,你仍然无法使原型正常工作。但由于你是在秘密工作,所以不可能向你的机械爱好者朋友征求建议。

然后,有一天,你的邻居从车库里拿出一辆自行车,上面有一个全新的变速装置。原来,他一直在制造与你的发明非常相似的东西,但得到了自行车店里一些朋友的帮助。这时,你很恼火。你向他展示了你的作品。他指出,你的设计有一些简单的缺陷。如果你向他展示的话,这些缺陷可能在第一周就被修复了。这里有很多教训可以借鉴。

早期检测如果你将自

己的绝妙创意秘而不宣,在实施完善之前拒绝向任何人展示任何东西,那么你就是在冒巨大的风险。在早期很容易犯下根本的设计错误。你可能会重新发明轮子。²你还会失去协作的好处:注意到你的邻居通过与他人合作进步得有多快吗?这就是为什么人们在跳入深水区之前会先试水:你需要确保你在做正确的事情,你做得正确,并且以前没有做过。早期失误的可能性很高。你在早期征求的反馈越多,你就越能降低这种风险。³记住那句久经考验的口头禅:“早失败,快速失败,经常失败。”

早期分享不仅是为了防止个人失误和让你的想法得到审查。它还有助于加强我们所谓的项目巴士因子。

巴士因素

巴士因子(名词):在你的项目彻底失败之前需要被巴士撞到的人数。

你的项目中的知识和诀窍分散吗?如果你是唯一了解原型代码如何工作的人,你可能会享受很好的工作保障。但是如果你出事了,整个项目就完蛋了。但是,如果你和同事一起工作,你的巴士系数就翻了一番。如果你有一个小团队一起设计和制作原型,情况就更好了。即使团队成员失踪,项目也不会陷入孤立。请记住:团队成员可能不会真的出事,但生活中仍会发生其他不可预测的事件。有人可能会结婚、搬走、离开公司或休假去照顾生病的亲戚。确保每个责任领域除了有主要和次要所有者之外,还有至少良好的文档,这有助于确保项目在未来取得成功,并提高项目的巴士系数。希望大多数工程师都认识到,成为成功项目的一部分比成为失败项目的关键部分要好。

除了公交车因素,还有整体进度问题。人们很容易忘记,独自工作往往是一项艰巨的任务,进度比人们愿意承认的要慢得多。

独自工作时你能学到多少东西?你的进度有多快?Google 和 Stack Overflow 是意见和信息的重要来源,但它们无法替代实际的人类经验。与其他人一起工作可以直接增加努力背后的集体智慧。当你陷入荒谬的事情时,你浪费了多少时间让自己摆脱困境?想想看

²从字面上理解,如果你实际上是一名自行车设计师。

³我应该指出,如果你仍然
不确定你的总体方向或目标。

如果有几个同事在你身后监督你并立即告诉你如何犯错以及如何解决问题,那体验就会有所不同。

这正是软件工程公司中团队坐在一起（或进行结对编程）的原因。编程很难。软件工程更难。你需要第二双眼睛。

进步的速度这里还有

另一个类比。想想你如何使用编译器。当你坐下来编写一个大型软件时,你是否花了几天的时间编写 10,000 行代码,然后,在编写完最后一行完美的代码后,第一次按下“编译”按钮?当然你不会。你能想象会导致什么样的灾难吗?程序员在紧密的反馈循环中工作效率最高:编写一个新函数,编译。添加测试,编译。重构一些代码,编译。这样,我们就可以生成代码后尽快发现并修复拼写错误和错误。我们希望编译器在我们身边陪伴我们完成每一步;有些环境甚至可以在我们输入代码时编译我们的代码。

这就是我们保持代码质量高并确保软件一点一点正确发展的方式。当前 DevOps 的科技生产力理念明确提出了这些目标:尽早获得反馈、尽早测试、尽早考虑安全性和生产环境。所有这些都与开发人员工作流程中的“左移”理念捆绑在一起;我们越早发现问题,修复它的成本就越低。

不仅在代码层面,而且在整个项目层面也需要同样的快速反馈循环。雄心勃勃的项目发展迅速,必须适应不断变化的环境。项目会遇到不可预测的设计障碍或政治风险,或者我们只是发现事情没有按计划进行。

需求会出乎意料地发生变化。如何获得反馈循环,以便您知道您的计划或设计何时需要更改?答案:通过团队合作。大多数工程师都知道这句话:“众人之眼会让所有错误变得浅显”,但更好的版本可能是:“众人之眼可以确保您的项目保持相关性并按计划进行。”

在洞穴中工作的人们醒来后发现,虽然他们最初的愿景可能已经完成,但世界已经改变,他们的项目已变得无关紧要。

案例研究:工程师和办公室25 年前,传统

观点认为,工程师要想提高工作效率,就需要拥有自己的办公室,并且门可以关上。据说这是他们能够拥有大量不受干扰的时间来专心编写大量代码的唯一方法。

我认为,大多数工程师⁴ 不仅没有必要待在私人办公室,而且非常危险。如今的软件是由团队而非个人编写的,与团队其他成员建立高带宽、随时可用的连接比互联网连接更有价值。你可以拥有世界上所有不受干扰的时间,但如果你用它来做错误的事情,你就是在浪费时间。

不幸的是,现代科技公司(包括谷歌,在某些情况下)似乎已经将钟摆摆到了完全相反的极端。走进他们的办公室,你经常会发现工程师们聚集在巨大的房间里一百人或更多人挤在一起 根本没有墙壁。这种“开放式布局”现在是一个备受争议的话题,因此,对开放式办公室的敌意正在上升。最微不足道的谈话都会公开,人们最终不会说话,因为可能会惹恼几十个邻居。这和私人办公室一样糟糕!

我们认为折中方案才是最佳解决方案。将4到8人的团队聚集在小房间(或大办公室)中,这样可以轻松(且不尴尬地)进行自发对话。

当然,在任何情况下,个别工程师仍然需要一种方法来过滤噪音和干扰,这就是为什么我见过的大多数团队都开发了一种沟通方式,告诉他们他们目前很忙,你应该限制干扰。我们中的一些人曾经在一个有语音中断协议的团队中工作:如果你想说话,你会说“断点玛丽”,玛丽是你想谈话的人的名字。如果玛丽到了可以停下来的地步,她会转动椅子听。如果玛丽太忙了,她会说“ack”,然后你继续做其他事情,直到她结束当前的头脑状态。

其他团队会将代币或毛绒玩具放在显示器上,以表示只有在紧急情况下才可以打扰他们。还有一些团队会为工程师提供降噪耳机,以便他们更轻松地处理背景噪音 事实上,在许多公司,戴耳机这一行为本身就是一种常见的信号,意思是“除非真的很重要,否则不要打扰我”。许多工程师在编码时倾向于进入只戴耳机的模式,这可能在短时间内有用,但如果一直使用,对协作的危害可能与把自己关在办公室里一样大。

请不要误解我们 我们仍然认为工程师需要不受干扰的时间来专注于编写代码,但我们认为他们同样需要与团队建立高带宽、低摩擦的连接。如果团队中知识较少的人觉得向你提问存在障碍,那就是一个问题:找到正确的平衡是一门艺术。

⁴不过,我承认,内向的人可能比大多数人更需要安宁、安静和独处的时间。

人们可能会受益于更安静的环境,即使他们没有自己的办公室。

简而言之,不要隐藏

因此,“隐藏”归结为:独自工作本质上比与他人合作更危险。尽管你可能担心有人窃取你的想法或认为你不够聪明,但你更应该担心的是,你浪费了大量的时间在错误的事情上。

不要成为另一个统计数据。

团队至上

那么,现在让我们回过头来将所有这些想法放在一起。

我们一直在强调的观点是,在编程领域,孤独的工匠极其罕见 即使他们确实存在,他们也不是在真空中创造超人的成就;他们改变世界的成就几乎总是灵感的火花和英勇的团队努力的结果。

一支伟大的球队会充分发挥其超级明星的作用,但整体总是大于各部分之和。然而打造一支超级明星球队却极其困难。

让我们用更简单的语言来表达这个想法:软件工程是一项团队努力的结果。

这一概念直接违背了我们许多人内心的天才程序员幻想,但当你独自一人待在黑客巢穴中时,光有才华是不够的。你不会通过隐藏和准备你的秘密发明来改变世界或取悦数百万计算机用户。你需要与其他人合作。分享你的愿景。分工合作。向他人学习。组建一支优秀的团队。

考虑一下:您能说出多少个真正由一个人编写的广泛使用的成功软件?(有些人可能会说“LaTeX”,但它很难被“广泛使用”,除非您认为撰写科学论文的人数占所有计算机用户的统计显著比例!)

高效的团队是黄金,也是成功的真正关键。您应该尽一切可能争取这种体验。

社交互动的三大支柱

那么,如果团队合作是生产优秀软件的最佳途径,那么如何建立(或找到)一支优秀的团队呢?

要达到协作的极致,您首先需要学习并接受我所说的社交技能的“三大支柱”。这三个原则不仅仅是为了促进人际关系;它们是所有健康互动和协作的基础:

支柱 1:谦逊 你不是宇宙的中心

（你的代码也不是！）。你既不是无所不知的，也不是绝对正确的。你愿意自我完善。

支柱 2:尊重 你真诚地关心与你共事的其他人。

你善待他们，欣赏他们的能力和成就。

支柱三:信任

你相信其他人有能力并且会做正确的事情，并且你愿意在适当的时候让他们掌控一切。⁵

如果你对几乎所有的社会冲突进行根本原因分析，你最终都可以将其归结为缺乏谦逊、尊重和/或信任。这听起来可能难以置信，但不妨一试。想想你目前生活中一些令人不快或不舒服的社交情况。从最基本的层面上讲，每个人都足够谦逊吗？人们真的互相尊重吗？相互信任吗？

这些支柱为何重要？

当你开始阅读本章时，你可能并没有打算加入某种每周一次的支持小组。我们感同身受。处理社交问题可能很困难：人们很混乱、难以预测，而且经常令人厌烦。与其投入精力分析社交情况并制定战略行动，不如放弃所有努力。与可预测的编译器相处要容易得多，不是吗？为什么要费心处理社交问题呢？

以下是理查德·汉明（Richard Hamming）一次著名演讲中的一段引文：

通过不辞辛劳地给秘书讲笑话并表现出一点友好，我得到了出色的秘书帮助。例如，有一次，由于某种愚蠢的原因，Murray Hill 的所有复印服务都被占用了。不要问我怎么回事，但他们确实被占用了。我想做点什么。我的秘书给 Holmdel 的某个人打了电话，跳上公司的车，花了一小时的时间去复印，然后回来。这是我努力让她高兴、给她讲笑话和表现出友好的回报；正是这些额外的工作后来让我得到了回报。通过意识到你必须使用系统并研究如何让系统完成你的工作，你学会了如何让系统适应你的愿望。

寓意在于：不要低估社交游戏的力量。社交游戏不是欺骗或操纵他人，而是建立关系以完成任务。关系总是比项目更持久。当你与同事的关系更紧密时，他们会更愿意在你需要他们时付出更多努力。

⁵如果你过去曾因为将工作委托给不称职的人而遭受过损失，那么这将是极其困难的。

谦逊、尊重和信任的实践所有这些关于谦逊、尊

重和信任的说教听起来都像是布道。让我们走出迷雾,思考如何在现实生活中运用这些想法。我们将研究一系列你可以开始使用的具体行为和示例。其中许多乍一看可能很明显,但在你开始思考它们之后,你会注意到你(和你的同龄人)经常犯不遵循这些行为的错误。我们自己也确实注意到了这一点!

放下自我好

吧,这是一种更简单的方法,告诉那些不够谦虚的人放下他们的“态度”。没有人愿意和一个总是表现得像房间里最重要的人的人一起工作。即使你知道你是讨论中最聪明的人,也不要当着别人的面炫耀。例如,你是否总是觉得你需要在每个话题上都说第一句或最后一句?你是否觉得有必要对提案或讨论中的每个细节发表评论?或者你认识做这些事情的人吗?

虽然谦虚很重要,但这并不意味着你需要成为受气包;自信并没有错。只是不要表现得像个万事通。

更好的办法是,考虑追求“集体”自我;与其担心自己是否优秀,不如努力建立团队成就感和集体自豪感。例如,Apache 软件基金会长期以来一直致力于围绕软件项目创建社区。这些社区具有非常强烈的身份认同感,并拒绝那些更关心自我推销的人。

自我意识表现在很多方面,很多时候,它会妨碍你的工作效率,拖慢你的进度。这是 Hamming 演讲中的另一个精彩故事,完美地说明了这一点(重点是我们):

约翰·图基几乎总是穿着非常随意。他会走进一个重要的办公室,而对方要花很长时间才能意识到这是个一流的人,他最好听从他的安排。很长一段时间以来,约翰不得不克服这种敌意。这是徒劳的努力!我并没有说你应该顺从;我说的是,“顺从的外表会让你走得很远。”如果你选择以任何方式维护你的自我,“我要按我的方式去做”,那么你在整个职业生涯中都会付出一点小小的代价。而这,在整个一生中,会累积成大量不必要的麻烦。[...]通过意识到你必须使用这个系统,并研究如何让系统为你工作,你就会学会如何让系统适应你的愿望。或者你可以用你的一生来稳步对抗它,就像一场小规模的、未宣战的战争。

学会给予和接受批评几年

前,乔开始了一份程序员的新工作。第一周后,他开始深入研究代码库。因为他关心正在发生的事情,所以他开始温和地询问其他队友的贡献。他通过电子邮件发送简单的代码评论,礼貌地询问设计假设或指出可以改进逻辑的地方。几周后,他被叫到主管办公室。“有什么问题吗?”乔问。“我做错了什么吗?”

主管看起来很担心:“乔,我们收到了很多关于你行为的投诉。显然,你对你的队友非常严厉,到处批评他们。他们很不高兴。你需要收敛一下。”乔完全被搞糊涂了。他认为,他的代码审查肯定应该受到同事的欢迎和赞赏。

然而,在这种情况下,乔应该对团队普遍存在的不安全感更加敏感,应该使用更微妙的方式将代码审查引入到文化中 甚至可以简单到提前与团队讨论这个想法,并要求团队成员尝试几个星期。

在专业的软件工程环境中,批评几乎从来都不是针对个人的 它通常只是改进项目过程的一部分。诀窍是确保你(和你周围的人)明白对某人创造性成果的建设性批评和对某人性格的赤裸裸的攻击之间的区别。后者是无用的 它很琐碎,几乎不可能采取行动。前者可以(也应该!)有所帮助,并提供改进的指导。而且,最重要的是,它充满了尊重:提出建设性批评的人真心关心对方,希望他们改进自己或他们的工作。学会尊重你的同事,礼貌地提出建设性的批评。如果你真的尊重一个人,你就会有动力选择得体、有帮助的措辞 这是一项需要大量练习才能获得的技能。我们在第9章中更详细地介绍这一点。

在对话的另一方,你也需要学会接受批评。

这意味着不仅要对自己的技能保持谦虚,还要相信对方会真心为你着想(以及你的项目的利益!),不会真的认为你是个白痴。编程和其他技能一样:通过练习可以提高。如果一个同事指出你可以提高杂耍技巧的方法,你会认为这是对你人格和人格价值的攻击吗?我们希望不会。同样,你的自我价值不应该与你编写的代码或你构建的任何创意项目联系在一起。我们再重复一遍:你不是你的代码。一遍又一遍地说。

你不是你所创造的。你不仅需要自己相信这一点,还需要让你的同事也相信这一点。

例如,如果你有一个不自信的合作者,不要这样说:“伙计,你那个方法的控制流完全错了。你应该像其他人一样使用标准的xyzzy代码模式。”这种反馈充满了反模式:你告诉某人他们“错了”(好像世界非黑即白),要求他们改变一些东西,并指责他们创造了一些东西

违背了其他人的做法（让他们觉得自己很愚蠢）。你的同事会立即感到被冒犯，他们的反应必然会过于情绪化。

表达同样意思的更好方式可能是：“嘿，我对这一节的控制流感到困惑。我想知道 xyzzy 代码模式是否能使它更清晰、更易于维护？”请注意，你是如何用谦逊的态度来提出关于你而不是他们的问题。他们没有错；你只是在理解代码方面遇到了麻烦。

这个建议只是为了向可怜的你澄清一些事情，同时可能有助于实现项目的长期可持续发展目标。你也没有要求任何东西——你让你的合作者有能力平静地拒绝这个建议。讨论的重点是代码本身，而不是任何人的价值或编码技能。

快速失败并迭代

商界有一个广为人知的都市传说，说的是有一位经理犯了一个错误，损失了 1000 万美元。第二天，他垂头丧气地走进办公室，开始收拾办公桌，当他接到“首席执行官想见你”的电话时，他拖着沉重的脚步走进首席执行官的办公室，悄悄地把一张纸从桌子上推过去。

“这是什么？”首席执行官问道。

“我辞职，”这位高管说道。“我想你叫我来这里就是为了解雇我。”

“解雇你？”首席执行官难以置信地回答道。“我为什么要解雇你？我刚花了 1000 万美元培训你！”⁶

毫无疑问，这是一个极端的故事，但故事中的首席执行官明白，解雇这名高管并不能挽回 1000 万美元的损失，而且会因失去一名宝贵的高管而使损失雪上加霜，他可以非常肯定这名高管不会再犯同样的错误。

在 Google，我们最喜欢的座右铭之一是“失败是一种选择”。人们普遍认为，如果你不时地失败，那说明你不够创新或不敢冒足够的风险。失败被视为学习和改进的黄金机会，以便下次再试。⁷事实上，托马斯·爱迪生经常被引用说：“如果我发现 10,000 种方法行不通，那我并没有失败。我不会气馁，因为放弃每一次错误的尝试都是向前迈出的又一步。”

在 Google X（负责自动驾驶汽车和气球互联网接入等“登月计划”的部门）中，失败被刻意地纳入了激励体系。人们想出各种稀奇古怪的想法，同事们也受到积极的鼓励。

⁶在网上你可以找到这个传说的十几个版本，分别出自不同的著名经理。

⁷同样，如果你一遍又一遍地做同样的事情并且不断失败，那不是失败，而是无能。

以尽快击落它们。个人会得到奖励（甚至竞争），看看他们能在固定时间内推翻或否定多少想法。只有当一个概念真的无法被所有同行在白板上揭穿时，它才会进入早期原型阶段。

无责备事后文化

从错误中汲取教训的关键是记录失败，进行根本原因分析，并撰写“事后分析报告”（谷歌和许多其他公司都这么称呼）。要特别注意，事后分析报告不是一份无用的道歉、借口或指责清单——这不是它的目的。一份合适的事后分析报告应该始终包含对所学知识以及学习经验将带来哪些变化的解释。然后，确保事后分析报告易于获取，并且团队确实遵循了提议的更改。正确记录失败还可以让其他人（现在和将来的人）更容易知道发生了什么，避免重蹈覆辙。不要抹去你的足迹——要像跑道一样照亮那些追随你的人！

良好的事后分析应该包括以下内容：

- 事件的简要摘要 · 事件的时间表，从发现到调查再到解决
- 事件的主要原因 · 影响和损害评估 · 一组行动项目（与所有者一起）以立即解决问题 · 一组行动项目以防止事件再次发生
- 经验教训

学会耐心几年前，

我编写了一个工具，用于将 CVS 存储库转换为 Subversion（后来是 Git）。由于 CVS 的变化无常，我不断发现奇怪的错误。由于我的老朋友兼同事 Karl 对 CVS 非常了解，所以我们决定合作修复这些错误。

当我们开始结对编程时，出现了一个问题：我是一个自下而上的工程师，乐于深入泥潭，通过快速尝试很多东西并略过细节来解决问题。然而，Karl 是一个自上而下的工程师，他希望在着手解决错误之前，全面了解情况并深入研究调用堆栈上几乎每种方法的实现。这导致了一些严重的人际冲突、分歧和偶尔的激烈争论。它发展到了

我们两个人根本无法一起进行结对编程 :这对我们两个来说都太令人沮丧了。

尽管如此 , 我们长期以来一直相互信任和尊重。再加上耐心 , 这帮助我们找到了一种新的合作方式。我们会坐在电脑前 , 找出错误 , 然后分头从两个方向 (自上而下和自下而上) 解决问题 , 然后再一起找出问题所在。我们的耐心和愿意即兴发挥新工作方式的意愿不仅挽救了这个项目 , 也挽救了我们的友谊。

乐于接受影响你越

乐于接受影响 , 你就越能影响别人 ; 你越脆弱 , 你就越显得强大。这些说法听起来像是奇怪的矛盾。但是每个人都能想到自己共事过的人 , 他们就是固执己见 无论别人如何劝说 , 他们都会更加固执。这样的团队成员最终会怎样 ? 根据我们的经验 , 人们不再听取他们的意见或反对意见 ; 相反 , 他们最终会 “ 绕过 ” 他们 , 就像一个每个人都认为理所当然的障碍。你当然不想成为这样的人 , 所以请记住这个想法 : 让别人改变你的想法是可以的。在本书的开篇 , 我们说过 , 工程本质上就是权衡。除非你拥有不变的环境和完美的知识 , 否则你不可能一直对所有事情都正确 , 所以当你看到新证据时 , 你当然应该改变主意。谨慎选择你的战斗 : 要想让别人听到你的声音 , 你首先需要倾听别人的声音。最好在立下决心或坚定地宣布决定之前先倾听别人的声音 如果你不断改变主意 , 人们会认为你优柔寡断。

脆弱性这个概念看起来也很奇怪。如果一个人承认自己对当前话题或问题的解决方案一无所知 , 那么他们在群体中会有什么可信度呢 ? 脆弱性是软弱的表现 , 这会破坏信任 , 对吧 ?

不对。承认自己犯了错误或者能力不足可以长期提升你的地位。事实上 , 愿意表达脆弱是谦逊的外在表现 , 它表明你有责任心和愿意承担责任 , 也表明你信任别人的意见。作为回报 , 人们最终会尊重你的诚实和力量。有时 , 你能做的最好的事情就是说 “ 我不知道 ” 。

例如 , 职业政客以从不承认错误或无知而臭名昭著 , 即使他们在某个问题上明显是错的或不了解。这种行为存在的主要原因是政客经常受到对手的攻击 , 这也是为什么大多数人不相信政客说的一句话。

然而 , 当你编写软件时 , 你不需要不断地

防守 你的队友是合作者,而不是竞争对手。你们都有相同的目标。

作为 Google 人,在行为和

人际交往方面,我们有自己的内部版本的“谦逊、尊重和信任”原则。

从我们文化的早期开始,我们就经常将某些行为称为“Google 风格”或“非 Google 风格”。这个词从未得到明确定义;相反,每个人都只是将其理解为“不作恶”或“做正确的事”或“善待彼此”。随着时间的推移,当我们面试工程师职位的候选人或在内部撰写绩效评估时,人们也开始使用“Google 风格”一词作为文化契合度的非正式测试。人们经常用这个词来表达对他人的看法;例如,“这个人编码很好,但似乎没有 Google 风格。”

当然,我们最终意识到“Googley”一词的含义过于丰富;更糟糕的是,它可能成为招聘或评估中无意识偏见的来源。如果“Googley”对每个员工都有不同的含义,我们就有可能让这个词开始意味着“和我一样”。显然,这不是一个好的招聘测试 我们不想雇佣“和我一样”的人,而是来自不同背景、拥有不同观点和经历的人。面试官想和应聘者(或同事)喝啤酒的个人愿望永远不应被视为其他人的表现或能够在 Google 茁壮成长的有效信号。

谷歌最终解决了这个问题,通过明确定义“谷歌精神”的标准来解决这个问题 我们所寻找的一系列代表强大领导力并体现“谦逊、尊重和信任”的属性和行为:

在模糊的环境中茁壮成

长,即使在环境不断变化的情况下,也能处理相互冲突的信息或方向,建立共识,并在解决问题上取得进展。

重视反馈 谦虚地接

受和给予反馈,并了解反馈对于个人(和团队)发展有多么重要。

挑战现状 能够制定雄心勃

勃的目标并且即使面临来自他人的阻力或惰性也能坚持追求。

以用户为中心 对 Google

产品的用户抱有同理心和尊重,并采取符合他们最佳利益的行动。

关心团队 对同事有同理心
和尊重 ,主动帮助他们 ,提高团队凝聚力。

做正确的事 对所做的每
一件事都有强烈的道德感 ;愿意做出困难或不便的决定来保护团队和产品的完整性。

现在 ,我们已经更好地定义了这些最佳实践行为 ,我们开始避免使用 “Googley”一词。具体说明期望总是更好的
tions !

结论

几乎任何规模的软件项目的基础都是一个运作良好的团队。尽管单人软件开发人员的天才神话仍然存在 ,但事
实是没有人真正独自完成工作。软件组织要经受住时间的考验 ,必须拥有一种健康的文化 ,这种文化植根于谦逊、
信任和尊重 ,以团队而不是个人为中心。此外 ,软件开发的创造性本质要求人们承担风险 ,偶尔也会失败 ;为了让人
们接受失败 ,必须有一个健康的团队环境。

TL;DR

· 了解独立工作的利弊。 · 了解您和您的团队在沟通和人际冲突
上花费的时间。花一点时间了解自己和他人的个性和工作风格 ,可以大大提高工作效率。

· 如果你想与团队或大型组织有效合作 ,请注意
您和他人所喜欢的工作方式。

第三章

知识共享

作者 :Nina Chen 和 Mark Barolak
编辑 :Riona MacNamara

您的组织比互联网上的某个随机人员更了解您的问题领域;您的组织应该能够回答自己的大部分问题。要实现这一点,您既需要知道这些问题答案的专家,也需要传播知识的机制,这正是我们将在本章中探讨的内容。这些机制从非常简单的机制(提出问题;写下您所知道的内容)到更结构化的机制,例如教程和课程。然而,最重要的是,您的组织需要一种学习文化,这需要创造一种心理安全感,让人们承认自己缺乏知识。

学习挑战

在整个组织内共享专业知识并非易事。如果没有强大的学习文化,就会出现挑战。谷歌经历了许多这样的挑战,尤其是在公司规模扩大之后:

缺乏心理安全

在这种环境中,人们害怕冒险或在他人面前犯错,因为他们害怕受到惩罚。这通常表现为一种恐惧文化或一种避免透明的倾向。

信息孤岛 知识碎片化是

指组织内不同部门之间不进行沟通或不使用共享资源。在这样的

环境中,每个群体都会发展出自己的做事方式。¹这通常会导致以下情况:

信息碎片化 每个岛屿对于整体的认识都是不完整的。

信息重复每个岛屿都重新发明了自己的做事方式。

信息偏差每个岛屿都有自己做同一件事的方式,这些方式可能会发生冲突,也可能不会。

单点故障 (SPOF)

当关键信息只能由一个人提供时,就会出现瓶颈。这与[巴士因子](#)有关,第2章将对此进行更详细的讨论。

SPOF 可能出于良好的意图:人们很容易养成“让我来帮你处理这件事”的习惯。但这种方法优化了短期效率(“我来做这件事更快”),代价是长期可扩展性较差(团队永远不会学会如何做任何需要做的事情)。这种心态也往往会导致孤注一掷的专业知识。

全有或全无的专业知识

一群人分为“无所不知”的人和新手,几乎没有中间立场。如果专家总是自己做所有事情,不花时间通过指导或文档培养新专家,这个问题往往会加剧。在这种情况下,知识和责任继续积累在那些已经拥有专业知识的人身上,而新的团队成员或新手只能自谋生路,并且成长速度会更慢。

鹦鹉学舌

式模仿,不加理解。这种行为的典型特征是盲目地复制模式或代码,而不理解其目的,通常认为这些代码是出于未知原因而需要的。

闹鬼墓地 经常用暗语

标记的地方,人们因为害怕出事而避免触碰或改变。与上述鹦鹉学舌不同,闹鬼墓地的特点是人们因为恐惧和迷信而避免采取行动。

¹换句话说,我们不是开发单个全局最大值,而是拥有一堆[局部最大值](#)。

在本章的其余部分,我们将深入探讨 Google 工程组织在应对这些挑战时发现的成功策略。

哲学

软件工程可以定义为多人开发多版本程序的过程。²人是软件工程的核心:代码是重要的输出,但只是构建产品的一小部分。至关重要的是,代码不会凭空而来,专业知识也是如此。每个专家都曾经是新手:组织的成功取决于人才的培养和投资。

专家提供的个性化一对一建议总是非常宝贵的。不同的团队成员有不同的专业领域,因此对于任何给定的问题,最好的队友也会有所不同。但是,如果专家休假或更换团队,团队可能会陷入困境。尽管一个人可能能够为一对多提供个性化帮助,但这种帮助无法扩展,并且仅限于少数“许多人”。

另一方面,记录的知识不仅可以更好地扩展到团队,还可以扩展到整个组织。团队 wiki 等机制使许多作者能够与更大的群体分享他们的专业知识。但是,尽管书面记录比一对一对话更具可扩展性,但这种可扩展性也有一些缺点:它可能更加通用,不太适用于个别学习者的情况,并且需要额外的维护成本来保持信息的相关性和最新性。

部落知识存在于团队成员个人所知与记录内容之间的差距中。人类专家知道这些没有写下来的东西。如果我们记录这些知识并加以维护,那么现在不仅可以直接一对一访问专家的人获得这些知识,而且任何可以找到并查看文档的人都可以获得这些知识。

那么,在一个所有事情都完美且即时记录的魔法世界里,我们不再需要咨询人了,对吗?不完全是。书面知识具有扩展优势,但有针对性的人工帮助也是如此。人类专家可以综合他们的知识。他们可以评估哪些信息适用于个人用例,确定文档是否仍然相关,并知道在哪里可以找到它。或者,如果他们不知道在哪里可以找到答案,他们可能知道谁知道。

² David Lorge Parnas, 软件工程:多版本程序的多人开发 (海德堡:柏林施普林格出版社,2011)。

部落知识和书面知识相辅相成。即使是一支拥有完美文档的专家团队也需要相互沟通、与其他团队协调并随着时间的推移调整策略。没有一种知识共享方法是所有类型学习的正确解决方案，良好的知识共享组合的细节可能会因您的组织而异。机构知识会随着时间的推移而发展，最适合您组织的知识共享方法可能会随着组织的发展而改变。培训、专注于学习和成长，并建立自己的专家队伍：没有所谓的工程专业知识过多。

心理安全

心理安全对于促进学习环境至关重要。

要学习，你必须首先承认有些事情你不懂。我们应该欢迎这样的诚实而不是惩罚它。（谷歌在这方面做得很好，但有时工程师们不愿意承认他们不理解某些事情。）

学习的很大一部分是能够尝试新事物，并且感到可以安心地接受失败。在健康的环境中，人们会感到很自在地提出问题、敢于犯错，并学习新事物。这是对所有 Google 团队的基本期望；事实上，**我们的研究**研究表明，心理安全是有效

团队。

导师制在 Google，我

们试图在“Noogler”（新 Googler）工程师加入公司时就定下基调。建立心理安全感的一个重要方法是为 Noogler 指派一名导师（不是团队成员、经理或技术主管的人），其职责明确包括回答问题和帮助 Noogler 成长。有一位正式指派的导师来寻求帮助可以让新人更轻松，也意味着他们不必担心占用同事太多的时间。

导师是已在 Google 工作一年以上的志愿者，可以提供从使用 Google 基础设施到了解 Google 文化等各方面的建议。至关重要的是，如果学员不知道该向谁寻求建议，导师可以成为他们交谈的安全网。导师与学员不属于同一团队，这可以让学员在遇到棘手情况时更放心地寻求帮助。

导师制使学习正规化并促进学习，但学习本身是一个持续的过程。同事之间总会有相互学习的机会，无论是新员工加入组织，还是经验丰富的工程师学习新技术。在健康的团队中，队友不仅会开放，还会

回答,也要提问:表明他们不知道某事并相互学习。

大群体中的心理安全向附近的队友寻求帮助

比接近一大群陌生人要容易得多。但正如我们所见,一对一的解决方案扩展性不强。小组解决方案更具扩展性,但也更可怕。对于新手来说,提出一个问题并向一大群人询问可能会令人生畏,因为他们知道他们的问题可能会被存档很多年。在大群体中,心理安全的需求被放大了。小组中的每个成员都应发挥作用,创造和维护一个安全的环境,确保新手有信心提出问题,而崭露头角的专家感到有能力帮助那些新手,而不必担心他们的答案受到资深专家的攻击。

实现这种安全和友好的环境的最重要方法是让群体互动是合作性的,而不是对抗性的。

表 3-1列出了一些推荐的群体互动模式(及其相应的反模式)的示例。

表 3-1. 群体互动模式

推荐图案（合作）	反模式（对抗性）
基本问题或错误得到正确的指导	基本问题或错误会被挑出来,提问的人会受到惩罚
给出解释是为了帮助提问者学习	解释是为了展示自己的知识
回应很友善、耐心且有帮助	回应傲慢、刻薄且缺乏建设性
互动是为了找到解决方案而进行的共同讨论	互动是“赢家”与“输家”的争论

这些反模式可能无意中出现:有人可能试图提供帮助,但却意外地表现出傲慢和不友好。我们发现[Recurse Center 的社交规则](#)在这里有帮助:

不要假装惊讶(“什么? !我不敢相信你不知道堆栈是什么!”)
假装惊讶会阻碍心理安全,使得群体成员害怕承认自己缺乏知识。

没有“实际上”
迂腐的修正往往是为了哗众取宠,而不是为了精确。

不要在后座上操纵,不要打断
正在进行的讨论来发表意见而不参与谈话。

没有微妙的“主义”（“这很简单，我的祖母都可以做到！”）

细微的偏见（种族主义、年龄歧视、恐同症）表现可能会让人感到不受欢迎、不受尊重或不安全。

增长你的知识

知识共享从自己开始。认识到自己总有一些东西需要学习是很重要的。以下指南可帮助您增强个人知识。

提出问题

如果您只能从本章中学到一件事，那就是：永远学习；永远提问。

我们告诉 Nooglers，提升可能需要大约六个月的时间。这个延长的时间对于在 Google 庞大而复杂的基础设施上提升是必要的，但它也强化了学习是一个持续、反复的过程的想法。初学者犯的最大错误之一是在遇到困难时不寻求帮助。你可能会想独自努力解决它，或者担心你的问题“太简单”。你会想：“在我向任何人寻求帮助之前，我只需要更加努力。”不要落入这个陷阱！

你的同事往往是最好的信息来源：利用这一宝贵的资源。

不会有神奇的一天，让你突然知道在每种情况下该怎么做。总有更多的东西需要学习。在 Google 工作多年的工程师仍然觉得有些领域他们不知道自己在做什么，这没关系！不要害怕说“我不知道那是什么；你能解释一下吗？”把不知道的事情当作一个机会领域，而不是一个令人恐惧的领域。³无论您是团队新人还是高级领导者，您都应该始终处于一个可以学习的环境中。如果不是，您就会停滞不前（应该寻找新的环境）。

对于担任领导职务的人来说，树立这种行为的榜样尤为重要：重要的是不要错误地将“资历”等同于“无所不知”。事实上，你知道的越多，**你就越知道自己不知道的东西**。公开地提问⁴或表达知识上的差距可以强化别人也可以这样做的想法。

3. **冒名顶替综合症**在高成就者中并不少见，谷歌人也不例外。事实上，本书的大多数作者都患有冒名顶替综合症。我们承认，对失败的恐惧对患有冒名顶替综合症的人来说可能很困难，并且会强化他们避免开拓新领域的倾向。

4. 请参阅“[如何提出好的问题](#)”。

在接受帮助时,耐心和友善的回答可以营造一种环境,让人们感到安全,可以寻求帮助。让克服最初提出问题的犹豫变得更容易,可以尽早定下基调:主动征求意见,让即使是“琐碎”的问题也能轻松得到答案。尽管工程师可能可以自己弄清楚部落知识,但他们并不是在真空中工作。有针对性的帮助可以让工程师更快地提高工作效率,从而提高整个团队的工作效率。

理解背景

学习不仅仅是理解新事物,还包括了解现有事物设计和实现背后的决策。假设您的团队继承了一个存在多年的关键基础设施的遗留代码库。原作者早已离世,代码难以理解。从头开始重写可能比花时间学习现有代码更诱人。但不要想着“我不明白”并就此打住,而是要更深入地思考:您应该问什么问题?

考虑一下“[切斯特森栅栏](#)”的原理:在删除或更改某些内容之前,首先要了解它存在的原因。

在改革事物而非扭曲事物的问题上,有一条简单明了的原则;这条原则可能被称为悖论。在这种情况下,存在着某种制度或法律;为了简单起见,我们假设在道路上竖起了一道篱笆或一道大门。更现代的改革者会兴高采烈地走到它面前说:“我看不出这有什么用处;让我们把它清理掉吧。”对此,更聪明的改革者会回答:“如果你看不出这有什么用处,我当然不会让你把它清理掉。走开,好好想想。然后,当你回来告诉我你确实看到了它的用处时,我可能会允许你把它摧毁。”

这并不意味着代码不能缺乏清晰度,或者现有的设计模式不能出错,但工程师往往会比通常需要的更快地得出“这很糟糕!”的结论,尤其是对于不熟悉的代码、语言或范例。谷歌也不能幸免于此。寻找并理解背景,尤其是对于看起来不寻常的决定。在您理解了代码的背景和目的之后,请考虑您的更改是否仍然有意义。如果有意义,请继续进行;如果没有意义,请记录您的理由以供将来的读者阅读。

许多 Google 风格指南明确包含上下文,以帮助读者了解风格指南背后的原理,而不仅仅是记住一长串随意的规则。更微妙的是,了解给定指南背后的原理可让作者做出明智的决定,确定何时不应应用该指南或该指南是否需要更新。请参阅第 8 章。

扩大您的问题范围 :询问社区

获得一对一的帮助带宽很大,但规模必然有限。作为一名学习者,记住每一个细节可能很困难。为未来的自己做点好事:当你从一对—讨论中学到一些东西时,把它写下来。

未来的新人可能会有和你一样的问题。也帮他们一个忙,分享你写下的内容。

虽然分享您收到的答案很有用,但从更大的社区而不是个人那里寻求帮助也是有益的。在本节中,我们将研究不同形式的基于社区的学习。这些方法(群聊、邮件列表和问答系统)各有优缺点,相互补充。但它们都使知识寻求者能够从更广泛的同行和专家社区获得帮助,并确保答案可供该社区的现有和未来成员广泛使用。

群聊当你有问题时,

有时很难从合适的人那里获得帮助。也许你不确定谁知道答案,或者你想问的人很忙。在这些情况下,群聊非常有用,因为你可以同时向许多人提问,并与任何有空的人进行快速的来回对话。作为额外奖励,群聊的其他成员可以从问答中学习,许多形式的群聊都可以自动存档并在以后搜索。

群聊通常专注于主题或团队。主题驱动的群聊通常是开放的,任何人都可以加入并提问。它们往往吸引专家,而且规模可能很大,因此问题通常能很快得到解答。另一方面,团队导向的聊天往往规模较小,并且限制成员资格。因此,它们可能不像主题驱动的聊天那样具有影响力,但它们规模较小,可以让新加入者感到更安全。

虽然群聊非常适合快速提问,但它们没有提供太多结构,这使得很难从您没有积极参与的对话中提取有意义的信息。一旦您需要在小组外分享信息,或使其可供以后参考,您应该写一份文档或发送电子邮件列表。

邮件列表Google

的大多数主题都有一个 topic-users@ 或 topic-discuss@ Google Groups 邮件列表,公司中的任何人都可以加入或发送电子邮件。在公共邮件列表中提问就像在群聊中提问一样:问题会传达给很多人,他们可以

有可能回答该问题,关注该列表的任何人都可以从答案中学习。然而,与群聊不同,公共邮件列表易于与更广泛的受众共享:它们被打包成可搜索的档案,并且电子邮件线程比群聊提供更多的结构。在 Google,邮件列表也被编入索引,并通过 Google 的内部网搜索引擎 Moma 找到。

当你在邮件列表中找到问题的答案时,你可能会忍不住继续工作。但不要这么做!你永远不知道将来什么时候会有人需要同样的信息,因此,最佳做法是将答案发布回列表。

邮件列表并非没有缺点。它们非常适合需要大量上下文的复杂问题,但对于群聊擅长的快速来回交流来说,它们却很笨拙。关于特定问题的帖子通常在处于活动状态时最有用。电子邮件存档是不可变的,很难确定在旧讨论帖子中发现的答案是否仍然与当前情况相关。此外,信噪比可能低于其他媒介(如正式文档),因为某人在其特定工作流程中遇到的问题可能不适用于您。

Google 的电子邮件

Google 文化以电子邮件为中心,电子邮件泛滥。Google 工程师每天会收到数百封电子邮件(甚至更多),这些电子邮件的可操作性程度各不相同。新用户可能要花几天时间设置电子邮件过滤器,以处理来自他们自动订阅的群组的大量通知;有些人干脆放弃,不再尝试跟上潮流。有些群组默认将大型邮件列表抄送给每个讨论,而不尝试将信息发送给可能对其特别感兴趣的人;因此,信噪比可能是一个真正的问题。

Google 默认倾向于基于电子邮件的工作流程。这并不一定是因为电子邮件是一种比其他通信方式更好的媒介(通常并非如此),而是因为我们的文化习惯于此。当您的组织考虑鼓励或投资哪种形式的沟通时,请记住这一点。

YAQS:问答平台

YAQS(“又一个问题系统”)是 Google 内部版本的 Stack Overflow 流程式的网站,让 Google 员工可以轻松链接到现有或正在进行的代码以及讨论机密信息。

与 Stack Overflow 一样,YAQS 具有邮件列表的许多相同优点,并进行了改进:标记为有用的答案会在用户界面中进行推广,并且

用户可以编辑问题和答案,这样当代码和事实发生变化时,它们仍然准确且有用。因此,一些邮件列表已被 YAQS 取代,而其他邮件列表则演变为更一般的讨论列表,不再专注于解决问题。

扩展你的知识： 你总有一些东西可以教

教学并不局限于专家,专业知识也并非二元状态,即您是新手还是专家。专业知识是您所知道内容的多维向量:每个人在不同领域的专业知识水平各不相同。这就是为什么多样性对组织成功至关重要的原因之一:不同的人会带来不同的观点和专业知识(参见第 4 章)。Google 工程师以各种方式教导他人,例如办公时间、技术讲座、授课、编写文档和审查代码。

办公时间

有时,找个人聊天真的很重要,在这种情况下,办公时间可能是一个很好的解决方案。办公时间是定期(通常每周一次)安排的活动,在此期间,一人或多人会回答有关特定主题的问题。办公时间几乎从来都不是知识共享的首选:如果您有一个紧急问题,等待下一次会议才能得到答复会很痛苦;如果您举办办公时间,这会占用时间,需要定期宣传。也就是说,办公时间确实为人们提供了一种与专家面对面交谈的方式。如果问题仍然不够明确,工程师还不知道要问什么问题(比如当他们刚开始设计一项新服务时),或者问题太专业而没有相关文档,办公时间就特别有用。

技术讲座和课程

Google 拥有强大的内部和外部技术讲座和课程文化。我们的 engEDU(工程教育)团队专注于为众多受众提供计算机科学教育,从 Google 工程师到世界各地的学生。在更基层的层面上,我们的 g2g(Googler2Googler)计划让 Google 员工

5 <https://talksat.withgoogle.com> 以及 <https://www.youtube.com/GoogleTechTalks>,仅举几例。

报名参加或参加谷歌员工的讲座和课程。6该计划非常成功,有数千名谷歌员工参与教授各种主题,从技术性(例如“理解现代CPU中的矢量化”)到纯粹为了娱乐(例如“初学者摇摆舞”)。

技术讲座通常由演讲者直接向观众进行演讲。另一方面,课堂可以包含讲座部分,但通常以课堂练习为中心,因此需要与会者更积极地参与。因此,讲师指导的课程通常比技术讲座要求更高,创建和维护成本更高,并且只用于讨论最重要或最困难的主题。

尽管如此,课程创建后,可以相对轻松地扩展,因为许多教师可以使用相同的课程材料来教授课程。我们发现,当存在以下情况时,课程往往效果最佳:

- 该主题非常复杂,经常会引起误解。

创建课程需要付出很多努力,因此只有当它们能够满足特定需求时才应该进行开发。

- 主题相对稳定。更新课堂材料是一项艰巨的工作,因此如果主题发展迅速,其他形式的知识共享将更具成本效益。· 有老师可以回答问题并提供个性化帮助,对主题大有裨益。如果学生无需指导即可轻松学

习,那么文档或录音等自助媒介会更有效。Google的许多入门课程也有自学版本。· 有足够的需求定期提供课程。否则,潜在的学习者将通过其他方式获取所需的信息,而不是等待课程。

在谷歌,这对于小型、地理位置偏远的办公室来说尤其是一个问题。

文档

文档是书面知识,其主要目的是帮助读者学习一些东西。并非所有书面知识都必须是文档,尽管它可以作为纸质记录使用。例如,可以在邮件列表主题中找到问题的答案,但该主题上原始问题的主要目的是寻求答案,其次才是为其他人记录讨论。

6 g2g计划的详情载于:Laszlo Bock,《工作规则!来自Google内部的洞察将改变你的生活和领导方式》(纽约:十二本书,2015年)。其中包括对该计划不同方面的描述,以及如何评估影响,以及在设立类似计划时应关注哪些方面的建议。

在本节中,我们将重点关注发现为正式文档做出贡献和创建正式文档的机会,从修正拼写错误等小事到记录部落知识等大事。



有关文档的更全面讨论,请参见第 10 章。

更新文档第一次学习某样

东西是寻找改进现有文档和培训材料的方法的最佳时机。等到您吸收并理解了新流程或系统时,您可能已经忘记了“入门”文档中有哪些难点或缺少哪些简单步骤。在此阶段,如果您发现文档中有错误或遗漏,请修复它!让营地比您发现时更干净⁷,并尝试自己更新文档,即使该文档由组织的不同部门拥有。

在 Google,工程师们感到有权更新文档,无论文档的所有者是谁 我们经常这样做 即使只是纠正拼写错误。随着 g3doc⁸ 的推出,社区维护水平显著提高,这使得 Google 员工更容易找到文档所有者来审查他们的建议。

它还留下了与代码无异的可审计的变更历史痕迹。

创建文档随着您的熟练

程度不断提高,编写您自己的文档并更新现有文档。

例如,如果您设置了新的开发流程,请记录这些步骤。然后,您可以让其他人更容易地遵循您的路径,只需将他们指向您的文档即可。

更好的是,让人们自己更容易找到文档。任何无法发现或无法搜索的文档都可能存在。

这是 g3doc 的另一个亮点,因为文档可以预见地位于源代码的旁边,而不是位于某个(无法找到的)文档或网页中。

最后,确保有反馈机制。如果读者没有简单直接的方式来指出文档是否过时或不准确,他们可能不会费心告诉任何人,而下一个新人也会遇到同样的问题。

⁷参见“童子军规则”和 Kevlin Henney,《每个程序员都应该知道的 97 件事》(波士顿·奥莱利,2010 年)。

⁸g3doc 代表“google3 文档”。google3 是 Google monodoc 当前版本的名称
石器源储存库。

lem。如果人们觉得有人会真正注意到并考虑他们的建议,他们就会更愿意贡献更改。在 Google,您可以直接从文档本身提交文档错误。

此外,Google 员工还可以轻松地在 g3doc 页面上发表评论。其他 Google 员工可以查看并回复这些评论,而且由于发表评论会自动向文档所有者提交错误报告,因此读者无需弄清楚该向谁报告接触。

促进文档编写传统上,鼓励工程

师编写文档可能很困难。编写文档需要花费时间和精力,而这些时间和精力本可以花在编码上,而且这项工作带来的好处不是立竿见影的,而且大部分都归他人所有。像这样的不对称权衡对整个组织来说是有利的,因为许多人可以从少数人的时间投入中受益,但如果没有任何激励措施,鼓励这种行为可能会很困难。我们将在[第 57 页的“激励和认可”一节中讨论其中一些结构性激励措施。](#)

然而,文档作者通常可以直接从编写文档中受益。

假设团队成员总是请你帮忙调试某些类型的生产故障。记录你的流程需要前期投入时间,但完成这项工作后,你可以通过让团队成员查看文档并仅在需要时提供实际帮助来节省未来的时间。

编写文档还有助于您的团队和组织扩大规模。首先,文档中的信息将被规范化为参考:团队成员可以参考共享文档,甚至可以自行更新。其次,规范化可能会传播到团队之外。文档的某些部分可能并非团队配置所独有,对于寻求解决类似问题的其他团队来说很有用。

代码

从元层面来看,代码就是知识,因此编写代码的行为本身可以被视为一种知识转录形式。尽管知识共享可能不是生产代码的直接意图,但它通常是一种新兴的副作用,可以通过代码的可读性和清晰度来促进。

代码文档是共享知识的一种方式;清晰的文档不仅有利于库的使用者,也有利于未来的维护者。同样,实现注释可以跨时间传递知识:您明确地为未来的读者(包括未来的您!)编写这些注释。在权衡方面,代码注释与一般文档有相同的缺点:它们需要积极维护,否则很快就会过时,任何读过与代码直接矛盾的注释的人都可以证明这一点。

代码审查(参见第9章)通常是作者和审查人员的学习机会。例如,审查人员的建议可能会让作者了解一种新的测试模式,或者审查人员可能会通过看到作者在代码中使用新库来了解它。Google通过可读性流程标准化了通过代码审查进行的指导,如本章末尾的案例研究中所述。

扩展组织的知识

随着组织的发展,确保整个组织内专业知识得到适当共享变得越来越困难。有些事情,比如文化,在每个发展阶段都很重要,而其他事情,比如建立规范的信息来源,可能对更成熟的组织更有益。

培养知识共享文化组织文化是人性中比较模糊的东西,许多公司都将其视为

事后才考虑的事情。但在Google,我们认为,首先关注文化和环境⁹比只关注环境的输出(如代码)会产生更好的结果。

进行重大的组织变革并非易事,关于这一主题的书籍数不胜数。我们并不假装知道所有答案,但我们可以分享Google为创建促进学习的文化所采取的具体步骤。

请参阅《工作规则!10》一书,以更深入地了解Google的企业文化。

⁹拉兹洛·博克,《工作规则!:来自谷歌内部的洞察将改变你的生活和领导方式》(纽约:十二本书,2015年)。

¹⁰同上。

尊重少数

人的不良行为可能会让整个团队或社区不受欢迎。在这样的环境中,新手会学会将问题带到其他地方,潜在的新专家会停止尝试,没有成长空间。在最糟糕的情况下,团队会减少到最有害的成员。从这种状态中恢复可能很困难。

知识共享可以而且应该以善意和尊重的方式进行。在科技领域,对“聪明的混蛋”的容忍 或者更糟的是,对“聪明的混蛋”的崇敬 既普遍又有害,但成为专家和善良并不相互排斥。谷歌软件工程职位阶梯的领导力部分清楚地概述了这一点:

尽管高层领导需要一定的技术领导力,但并非所有的领导力都针对技术问题。领导者要提高周围人的素质,提高团队的心理安全感,创造团队合作的文化,缓解团队内部的紧张局势,树立谷歌文化和价值观的榜样,让谷歌成为一个更有活力、更令人兴奋的工作场所。混蛋不是好的领导者。

这一期望是由高层领导塑造的:Urs Hözle (技术基础设施高级副总裁)和 Ben Treynor Sloss (Google SRE 创始人兼副总裁)撰写了一份经常被引用的内部文件 (“No Jerks”),解释了为什么 Google 员工应该关心工作中的尊重行为以及如何解决这个问题。

激励和认可良好的文化

必须得到积极培育,而鼓励知识共享的文化则需要致力于在系统层面上对其进行认可和奖励。

组织经常犯的一个错误是口头上支持一套价值观,而实际上却奖励那些不执行这些价值观的行为。人们对激励的反应比陈词滥调要强烈,因此,通过建立薪酬和奖励制度,兑现承诺非常重要。

谷歌采用了多种认可机制,从绩效评估和晋升标准等全公司范围的标准到谷歌员工之间的同行奖励。

我们的软件工程阶梯用于校准公司范围内的薪酬和晋升等奖励,它通过明确指出这些期望来鼓励工程师分享知识。在更高级别,阶梯明确指出了更广泛影响力的重要性,并且随着资历的提高,这种期望也会增加。在最高级别,领导力的例子包括:

- 通过担任初级员工的导师来培养未来的领导者,帮助他们在技术和 Google 职位上都得到发展

- 通过代码和设计审查、工程教育和开发以及为该领域的其他人提供专家指导来维护和发展 Google 的软件社区



有关领导力的更多信息,请参阅第5章和第 6 章。

工作阶梯期望是一种自上而下引导文化的方式,但文化也是自下而上形成的。在谷歌,同事奖金计划是我们接受自下而上文化的一种方式。同事奖金是一种金钱奖励和正式认可,任何谷歌员工都可以授予其他谷歌员工,以表彰他们超乎寻常的工作。¹¹例如,当拉维向朱莉娅发送同事奖金,以表彰她是邮件列表的顶级贡献者 定期回答让许多读者受益的问题 他是在公开表彰她的知识共享工作及其对团队以外的影响。

由于同事奖金是由员工而非管理层推动的,因此它们可以产生重要而强大的基层效应。

与同行奖金类似的是荣誉:对贡献的公开承认(通常影响力或努力程度比值得获得同行奖金的贡献要小),这提高了同行贡献的知名度。

当一名 Google 员工向另一名 Google 员工授予同事奖励或表扬时,他们可以选择将奖励电子邮件抄送给其他团体或个人,以提升对同事工作的认可。获奖者的经理通常会将奖励电子邮件转发给团队,以庆祝彼此的成就。

一个可以让人们正式且轻松地认可同事的系统是鼓励同事继续做他们所做的事情的有力工具。重要的不是奖金:而是同事的认可。

建立规范的信息源规范的信息源是集中的、公司范围内的信息

库,它提供了一种标准化和传播专家知识的方法。它们最适合与组织内所有工程师相关的信息,否则很容易出现信息孤岛。例如,建立

¹¹同事奖金包括现金奖励和证书,并且是 Google 员工奖励的永久组成部分
在名为 gThanks 的内部工具中记录。

基本的开发人员工作流程应该是规范的,而运行本地 Frobber 实例的指南仅与从事 Frobber 的工程师更相关。

建立规范的信息源需要的投资比维护团队文档等本地化信息更高,但也有更广泛的好处。为整个组织提供集中的参考资源,使广泛需要的信息更容易、更可预测地找到,并解决了当多个团队在处理类似问题时可能出现的信息碎片化问题,这些团队会制作自己的指南(通常是相互冲突的)。

由于规范信息具有高度可见性,旨在在组织层面提供共同理解,因此,由主题专家积极维护和审查内容非常重要。主题越复杂,规范内容拥有明确的所有者就越重要。善意的读者可能会发现某些内容已经过时,但缺乏专业知识来进行修复所需的重大结构更改,即使工具可以轻松建议更新。

创建和维护集中的、规范的信息源既昂贵又耗时,而且并非所有内容都需要在组织层面共享。在考虑要为这种资源投入多少精力时,请考虑您的受众。谁会从这些信息中受益?您?您的团队?您的产品领域?

全部都是工程师吗?

开发者指南Google

为工程师提供了广泛而深入的官方指导,包括[风格指南](#)、官方软件工程最佳实践12、代码审查13和测试指南14以及每周小贴士(TotW15)。信息量如此之大,以至于期望工程师从头到尾读完它是不切实际的,更不用说一次性吸收这么多信息了。相反,已经熟悉指南的人类专家可

以向同事工程师发送链接,然后同事工程师可以阅读参考资料并了解更多信息。专家无需亲自解释全公司的实践,从而节省了时间,而学习者现在知道有一个值得信赖的规范信息来源,他们可以在必要时访问。这样的过程可以扩展知识,因为它使人类专家能够利用通用的可扩展知识来识别和解决特定的信息需求

资源。

12例如有关谷歌软件工程的书籍。

13请参阅第9章。

14请参阅第11章。

15适用于多种语言。外部可用于C++,网址为<https://abseil.io/tips>。

go/ 链接

go/ 链接（有时称为 goto/ 链接）是 Google 的内部 URL 缩短器。¹⁶ 大多数 Google 内部参考资料至少有一个内部 go/ 链接。例如，“go/spanner”提供有关 Spanner 的信息，“go/python”是 Google 的 Python 开发人员指南。内容可以存在于任何存储库（g3doc、Google Drive、Google Sites 等）中，但指向它的 go/ 链接提供了一种可预测、令人难忘的访问方式。这带来了一些好处：

- go/ 链接非常短，因此很容易在对话中分享它们（“你应该看看 go/frobber！”）。这比必须找到链接然后向所有感兴趣的各方发送消息要容易得多。拥有一种低摩擦的方式来分享参考资料，更有可能首先分享这些知识。
- go/ 链接提供内容的永久链接，即使底层 URL 发生变化。当所有者将内容移动到其他存储库（例如，将内容从 Google 文档移动到 g3doc）时，他们只需更新 go/ 链接的目标 URL。go/ 链接本身保持不变。

go/ 链接已深深植根于 Google 文化，因此形成了一个良性循环：Google 员工在查找有关 Frobber 的信息时，很可能会先查看 go/frobber。如果 go/ 链接未指向 Frobber 开发人员指南（如预期的那样），Google 员工通常会自行配置该链接。因此，Google 员工通常可以在第一次尝试时猜出正确的 go/ 链接。

代码实验室

Google 代码实验室是指导性的、实践性的教程，通过结合解释、最佳实践示例代码和代码练习，向工程师传授新概念或流程。¹⁷ 谷歌广泛使用的技术的规范代码实验室集合可在 go/codelab 上找到。这些代码实验室在发布前要经过几轮正式审查和测试。代码实验室是静态文档和讲师指导课程之间的一个有趣的折中点，它们具有各自的优点和缺点。它们的实践性质使它们比传统文档更具吸引力，但工程师仍然可以按需访问它们并自行完成；但它们的维护成本很高，并且无法根据学习者的特定需求进行量身定制。

¹⁶ 个 go/ 链接与 Go 语言无关。

¹⁷ 外部代码实验室可在<https://codelabs.developers.google.com> 上找到。

静态分析静态

分析工具是一种强大的方法,可以共享可通过编程方式检查的最佳实践。每种编程语言都有自己特定的静态分析工具,但它们具有相同的通用目的:提醒代码作者和审阅者如何改进代码以遵循样式和最佳实践。一些工具更进一步,可以自动将这些改进应用于代码。



有关静态分析工具及其在 Google 中的使用方式的详细信息,请参阅[第 20 章](#)。

设置静态分析工具需要前期投资,但一旦到位,它们就会高效扩展。当工具中添加了对最佳实践的检查时,使用该工具的每个工程师都会意识到该最佳实践。这也让工程师可以腾出时间教授其他东西:原本用于手动教授(现已实现自动化)最佳实践的时间和精力可以用来教授其他东西。静态分析工具增强了工程师的知识。它们使组织能够应用更多最佳实践,并且比其他方式更一致地应用它们。

保持联系有些信息对于完

成工作至关重要,例如了解如何执行典型的开发工作流程。其他信息,例如流行生产力工具的更新,不那么重要,但仍然有用。对于这类知识,信息共享媒介的正式性取决于所传递信息的重要性。例如,用户希望官方文档保持最新,但通常对新闻通讯内容没有这样的期望,因此对所有者的维护和保养较少。

时事通讯

Google 有许多公司范围内的新闻通讯发送给所有工程师,包括 EngNews (工程新闻)、Ownd (隐私/安全新闻) 和 Google's Greatest Hits (本季度最有趣的中断报告)。这些是传达工程师感兴趣但不是任务关键的信息的好方法。对于这种类型的更新,我们发现,当新闻通讯发送频率较低且包含更多有用、有趣的内容时,它们的参与度会更高。否则,新闻通讯可能会被视为垃圾邮件。

尽管大多数 Google 简报都是通过电子邮件发送的,但有些简报的分发方式更有创意。厕所测试(测试技巧)和厕所学习(产品

厕所测试和厕所学习是张贴在厕所隔间内的单页通讯。这种独特的传递媒介使《厕所测试》和《厕所学习》从其他通讯中脱颖而出，所有期刊都在线存档。



请参阅[第 11 章](#)了解马桶测试的历史。

社区

Google 员工喜欢围绕各种主题组建跨组织社区来分享知识。这些开放的渠道让您更容易从自己圈子之外的其他人那里学习，避免信息孤岛和重复。Google 群组尤其受欢迎：Google 拥有数千个内部群组，其正式程度各不相同。有些群组专门用于故障排除；其他群组，如代码健康群组，则更适合讨论和指导。内部 Google+ 也是 Google 员工中流行的非正式信息来源，因为人们会发布有趣的技术故障或有关他们正在从事的项目的详细信息。

可读性：标准化指导 通过代码审查

在 Google，“可读性”不仅仅指代码的可读性；它是一种标准化的、Google 范围内的指导流程，用于传播编程语言的最佳实践。可读性涵盖广泛的专业知识，包括但不限于语言习语、代码结构、API 设计、通用库的适当使用、文档和测试覆盖率。

可读性最初是由一个人完成的。在 Google 早期，Craig Silverstein（员工 ID #3）会亲自与每一位新员工坐下来，逐行审查他们第一次提交的主要代码。这是一次挑剔的审查，涵盖了从代码改进方式到空格约定等所有内容。这让 Google 的代码库具有统一的外观，但更重要的是，它教授了最佳实践，强调了可用的共享基础架构，并向新员工展示了在 Google 编写代码的感觉。

不可避免的是，Google 的招聘速度超出了个人能力范围。许多工程师发现这个过程很有价值，他们自愿抽出时间来扩大该计划。如今，大约 20% 的 Google 工程师随时参与可读性过程，无论是作为审阅者还是代码作者。

可读性过程是什么？

在 Google，代码审查是强制性的。每个变更列表 (CL)¹⁸ 都需要可读性批准，这表明拥有该语言可读性认证的人已批准了该 CL。认证作者隐式地提供了他们自己的 CL 的可读性批准；否则，一个或多个合格的审阅者必须明确地为 CL 提供可读性批准。在 Google 发展到无法再强制要求每位工程师接受以达到所需严格程度的最佳实践的代码审查之后，添加了此要求。



请参阅[第 9 章](#)，了解 Google 代码审查流程的概述以及在此上下文中批准的含义。

在 Google 内部，拥有可读性认证通常被称为对该语言“具有可读性”。拥有可读性的工程师已证明他们始终如一地编写清晰、惯用且可维护的代码，这些代码体现了 Google 对特定语言的最佳实践和编码风格。他们通过可读性流程提交 CL 来做到这一点，在此过程中，一组集中的可读性审阅者会审查 CL 并就其展示各个掌握领域的程度提供反馈。随着作者内化可读性指南，他们收到的关于其 CL 的评论会越来越少，直到他们最终完成该流程并正式获得可读性。可读性带来了更大的责任：拥有可读性的工程师被信任继续将他们的知识应用到自己的代码中，并充当其他工程师代码的审阅者。

大约 1% 到 2% 的 Google 工程师是可读性审阅者。所有审阅者都是志愿者，任何具有可读性的人都可以自我提名成为可读性审阅者。可读性审阅者必须达到最高标准，因为他们不仅需要具备深厚的语言专业知识，还需要具备通过代码审查进行教学的能力。他们应该将可读性首先视为一个指导和合作的过程，而不是一个把关或对抗的过程。鼓励可读性审阅者和 CL 作者在审阅过程中进行讨论。审阅者为他们的评论提供相关的引用，以便作者可以了解风格指南中的理由（“Chesterson 的围栏”）。如果任何给定指南的理由不清楚，作者应该要求澄清（“提出问题”）。

¹⁸ 变更列表是版本控制系统中构成变更的文件列表。变更列表与使用[变更集](#)。

可读性是一个刻意以人为本的过程，旨在以标准化但个性化的方式扩展知识。作为书面知识和部落知识的互补融合，可读性结合了书面文档的优势（可以通过可引用的参考文献访问）和专家人工审阅者的优势（他们知道要引用哪些指南）。规范指南和语言建议都有全面的记录。这很好！但信息量太大¹⁹，可能会让人不知所措，尤其是对新手来说。

为什么要有个过程？

阅读代码的次数远多于编写代码的次数，这种影响在 Google 的规模和我们（非常大的）monorepo 中被放大了。任何工程师都可以查看和学习其他团队的代码知识财富，以及 Kythe 等强大的工具使整个代码库中的引用变得容易找到（参见第 17 章）。记录的最佳实践（参见第 8 章）的一个重要特征是它们为所有 Google 代码提供了一致的标准。可读性是这些标准的执行和传播机制。

可读性计划的主要优势之一是，它让工程师接触到的不仅仅是他们自己团队的部落知识。为了获得特定语言的可读性，工程师必须将 CL 发送给一组集中的可读性审阅者，这些审阅者负责审阅整个公司的代码。集中化流程会带来重大的权衡：随着组织的增长，该计划只能线性扩展，而不是亚线性扩展，但它可以更轻松地实施一致性、避免孤岛，并避免（通常是无意的）偏离既定规范。

代码库范围内一致性的价值怎么强调都不为过：即使有成千上万的工程师在几十年间编写代码，它也能确保给定语言的代码在整个语料库中看起来相似。这使读者能够专注于代码的功能，而不会被代码看起来与他们习惯的代码不同的原因所困扰。大规模变更作者（参见第 22 章）可以更轻松地在整个 monorepo 中进行变更，跨越数千个团队的界限。人们可以更换团队，并确信新团队使用给定语言的方式与他们之前的团队没有太大不同。

¹⁹截至 2019 年，仅 Google C++ 风格指南就有 40 页。构成完整指南的次要材料最佳实践的语料库要长很多倍。

20有关 Google 使用 monorepo 的原因，请参阅<https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-single-repository/full-text>。还要注意，并非所有 Google 代码都存在于 monorepo 中；此处描述的可读性仅适用于 monorepo，因为它是存储库内一致性的概念。

这些好处伴随着一些代价:与文档和类等其他媒介相比,可读性是一个重量级的过程,因为它是强制性的,并由 Google 工具强制执行 (参见第 19 章)。这些代价不小,包括以下内容:

- 对于没有任何具有可读性团队成员的团队来说,摩擦增加,因为他们需要从团队之外寻找审阅者来对 CL 进行可读性批准。
- 对于需要可读性的作者,可能会进行额外几轮代码审查
审查。
- 人为驱动的流程在扩展方面的缺点。由于依赖人工审阅人员进行专门的代码审阅,因此只能随组织增长而线性扩展。

那么,问题就是收益是否大于成本。还有时间因素:收益与成本的全面影响并不在同一时间范围内。

该计划刻意在增加短期代码审查延迟和前期成本,以换取更高质量的代码、存储库范围内的代码一致性和更专业的工程师技能等长期回报。长期收益的预期是,代码的寿命可能长达数年,甚至数十年。²¹

与大多数(或许是所有)工程流程一样,总有改进的空间。一些成本可以通过工具来降低。许多可读性注释解决了可以通过静态分析工具静态检测并自动注释的问题。随着我们继续在静态分析方面投入,可读性审阅者可以越来越多地关注高阶领域,例如特定代码块是否可被不熟悉代码库的外部读者理解,而不是自动检测某行是否有尾随空格。

但仅有抱负是不够的。可读性是一个有争议的计划:一些工程师抱怨说,这是一个不必要的官僚障碍,并且浪费了工程师的时间。可读性的权衡值得吗?为了找到答案,我们求助于我们值得信赖的工程生产力研究(EPR)团队。

²¹因此,已知时间跨度较短的代码不受可读性要求的限制。例如,experimental/ 目录(明确指定用于实验代码,不能推送到生产环境)和 Area 120 程序,谷歌实验产品的研讨会。

EPR 团队对可读性进行了深入研究,包括但不限于人们是否受到流程的阻碍、是否学到了什么、或者毕业后是否改变了行为。这些研究表明,可读性对工程速度有净积极影响。具有可读性的作者编写的 CL 在审查和提交方面所花的时间比不具有可读性的作者编写的 CL 在统计上明显要少。²² 在缺乏更多客观的代码质量衡量标准的情况下,具有可读性的工程师自述对代码质量的满意度高于不具有可读性的工程师。完成该计划的绝大多数工程师都表示对该流程感到满意并认为这是值得的。他们报告说,他们从审阅者那里学到了东西,并改变了自己的行为,以避免在编写和审阅代码时出现可读性问题。



如需深入了解这项研究以及 Google 内部工程生产力研究,请参阅第7章。

Google 拥有非常强大的代码审查文化,而可读性是这种文化的自然延伸。可读性从一位工程师的热情发展成为由人类专家指导所有 Google 工程师的正式计划。它随着 Google 的发展而发展和变化,并将随着 Google 需求的变化而继续发展。

结论

知识在某种程度上是软件工程组织最重要的(尽管是无形的)资本,而共享这些知识对于使组织在面对变化时具有弹性和冗余性至关重要。促进开放和诚实的知识共享的文化可以在整个组织中有效地传播这些知识,并使该组织能够随着时间的推移而扩大规模。在大多数情况下,对更轻松的知识共享的投资会在公司的整个生命周期内获得多倍的回报。

²²这包括控制多种因素,包括在谷歌的任职时间,以及与已经具有可读性的作者相比,不具有可读性的作者的 CL 通常需要额外几轮审查的事实。

TL;DR

- 心理安全是促进知识共享的基础环境。
- 从小事做起:提出问题并写下来。 · 让人们能够轻松地从人类专家和文献资料中获得所需的帮助。
- 在系统层面,鼓励和奖励那些花时间教导和拓展专业知识的人,而不仅仅是他们自己、他们的团队或他们的组织。 · 没有灵丹妙药:增强知识共享文化需要多种策略的组合,最适合您组织的确切组合可能会随着时间的推移而改变。

第四章

公平工程

作者:德玛·罗德里格斯 (Demma Rodriguez)
编辑:Riona MacNamara

在前面的章节中,我们探讨了编程与软件工程之间的对比。前者是编写代码来解决当前的问题,而后者是将代码、工具、政策和流程更广泛地应用于可能跨越数十年甚至一生的动态和模糊问题。在本章中,我们将讨论工程师在为广大用户设计产品时的独特职责。此外,我们将评估一个组织如何通过包容多样性来设计适合所有人的系统,并避免对我们的利益造成伤害。

用户。

软件工程领域虽然很新,但人们对它对弱势群体和多元化社会的影响的理解仍然很新。我们写这章并不是因为我们知道所有的答案。我们不知道。事实上,了解如何设计能够赋予我们所有用户权力并尊重他们产品仍然是 Google 正在学习的事情。我们在保护最脆弱的用户方面曾多次公开失败,因此我们写这章是因为通往更公平产品的道路始于评估我们自己的失败并鼓励增长。

我们之所以写下这一章,也是因为那些做出影响世界的开发决策的人与那些不得不接受并遵守这些决策的人之间的权力越来越不平衡,这些决策有时会让全球已经被边缘化的社区处于不利地位。与下一代软件工程师分享和反思我们迄今为止所学到的东西很重要。更重要的是,我们要帮助影响下一代工程师,让他们比现在的我们更优秀。

拿起这本书就意味着您很可能立志成为一名杰出的工程师。

您想解决问题。您渴望打造能够为最广泛的人群（包括最难接触的人群）带来积极成果的产品。要做到这一点，您需要考虑如何利用您构建的工具来改变人类的发展轨迹，希望是朝着更好的方向发展。

偏见是默认的

如果工程师不关注不同国籍、民族、种族、性别、年龄、社会经济地位、能力和信仰体系的用户，即使是最有才华的员工也会不经意地辜负用户。这种辜负往往是无意的；所有人都有一定的偏见。社会科学家在过去几十年中已经认识到，大多数人都表现出无意识的偏见，强化和传播现有的刻板印象。无意识的偏见是阴险的，往往比有意的排斥行为更难缓解。即使我们想做正确的事，我们也可能不会认识到自己的偏见。同样，我们的组织也必须认识到这种偏见的存在，并努力在员工队伍、产品开发和用户拓展方面解决它。

由于存在偏见，Google 有时无法公平地代表其产品中的用户。过去几年发布的产品没有足够关注代表性不足的群体。许多用户将我们在这些情况下缺乏认识归因于我们的工程师大多是男性、白人或亚洲人，当然不能代表使用我们产品的所有社区。我们的员工队伍中缺乏此类用户的代表性¹意味着我们通常不具备必要的多样性来了解我们产品的使用会如何影响代表性不足或弱势用户。

案例研究：谷歌未能实现种族包容
2015 年，软件工程师 Jacky Alciné 指出
2，Google Photos 中的图像识别算法将他的黑人朋友归类为“大猩猩”。谷歌对这些错误的反应
迟缓，解决起来也不彻底。

是什么导致了如此巨大的失败？有以下几点：

- 图像识别算法依赖于提供“正确”（通常意味着“完整”）的数据集。输入到 Google 图像识别算法的照片数据显然不完整。简而言之，这些数据并不代表总体。

¹ Google 2019 年多元化报告。 2

@jackyalcine, 2015 年。“Google Photos, 你们都搞砸了。我的朋友不是大猩猩。”Twitter, 2015 年 6 月 29 日。

<https://twitter.com/jackyalcine/status/615329515909156865>。

- Google 本身（以及整个科技行业）过去和现在都没有多少黑人代表，³这会影响此类算法设计和此类数据集收集的主观决策。组织本身的无意识偏见可能导致更具代表性的产品被搁置。
- Google 的图像识别目标市场没有充分涵盖这些代表性不足的群体。Google 的测试没有发现这些错误；结果，我们的用户发现了，这既让 Google 感到尴尬，也损害了我们的用户。

截至 2018 年，谷歌仍未充分解决根本问题。⁴

在这个例子中，我们的产品设计和执行不充分，未能适当考虑所有种族群体，结果辜负了我们的用户，并给 Google 带来了负面报道。其他技术也存在类似的缺陷：自动完成功能可能会返回令人反感或种族主义的结果。Google 的广告系统可能会被操纵以显示种族主义或令人反感的广告。YouTube 可能无法捕捉仇恨言论，尽管从技术上讲，仇恨言论在该平台上是被禁止的。

在所有这些情况下，技术本身并不是真正的罪魁祸首。例如，自动完成功能的设计目的并非针对用户或歧视。但它的设计也不够灵活，无法排除被视为仇恨言论的歧视性语言。结果，该算法返回的结果对我们的用户造成了伤害。对谷歌本身的伤害也显而易见：降低了用户对公司的信任和参与度。例如，黑人、拉丁裔和犹太裔求职者可能会对谷歌作为一个平台甚至作为一个包容性环境本身失去信心，从而破坏谷歌提高招聘代表性的目标。

这怎么会发生呢？毕竟，谷歌聘请的技术人员都受过良好的教育和/或拥有专业经验，他们是编写最佳代码并测试其工作的优秀程序员。“为每个人打造”是谷歌的品牌宣言，但事实是，我们还有很长的路要走，才能宣称我们做到了。解决这些问题的一种方法是帮助软件工程组织本身看起来像我们为之打造产品的人群。

³ 2018-2019 年的许多报告都指出科技行业缺乏多样性。一些知名机构包括美国国家妇女与信息技术中心、以及技术的多样性。

⁴ Tom Simonite，“当谈到大猩猩时，Google Photos 仍然视而不见”，连线杂志，2018 年 1 月 11 日。

理解多样性的必要性

在 Google,我们认为,要成为一名出色的工程师,您还需要专注于将不同的观点带入产品设计和实施中。这也意味着负责招聘或面试其他工程师的 Google 员工必须为建立更具代表性的员工队伍做出贡献。例如,如果您面试公司其他职位的工程师,那么了解招聘过程中出现偏见的结果非常重要。了解如何预测和预防伤害有着重要的先决条件。为了达到为每个人打造的地步,我们首先必须了解我们的代表性人群。我们需要鼓励工程师接受更广泛的教育和培训。

首先要打破这样的观念:只要拥有计算机科学学位和/或工作经验,你就具备成为一名优秀工程师所需的所有技能。计算机科学学位往往是必要的基础。

然而,单凭学位(即使加上工作经验)并不能让你成为一名工程师。打破只有拥有计算机科学学位的人才能设计和制造产品的观念也很重要。今天,**大多数程序员都拥有计算机科学学位**;他们成功地制定了规范,建立了变革理论,并应用了解决问题的方法。然而,正如上述例子所表明的,这种方法对于包容和公平的工程来说是不够的。

工程师应该首先将所有工作集中在他们试图影响的完整生态系统框架内。至少,他们需要了解用户的人口统计数据。工程师应该关注与自己不同的人,尤其是那些可能试图使用他们的产品造成伤害的人。

最难考虑的用户是那些被流程和他们使用技术的环境剥夺了权利的用户。为了应对这一挑战,工程团队需要代表其现有和未来的用户。

由于工程团队缺乏多元化代表,个体工程师需要学习如何为所有用户构建产品。

建设多元文化能力

杰出工程师的标志之一是能够理解产品如何有利于和不利于不同人群。工程师需要具备技术才能,但他们也应该有辨别力,知道何时该制造,何时不该制造。辨别力包括培养识别和拒绝导致不良结果的功能或产品的能力。这是一个崇高而艰难的目标,因为成为一名高绩效工程师需要大量的个人主义。然而,要想成功,我们必须扩展我们的

将关注点从我们自己的社区转向下一个十亿用户,或者转向那些可能因我们的产品而失去权利或被抛弃的现有用户。

随着时间的推移,你可能会构建出数十亿人每天使用的工具。影响人们对生命价值看法的工具、监视人类活动的工具、捕获和保存敏感数据(如孩子和亲人的图像以及其他类型的敏感数据)的工具。作为一名工程师,你可能拥有比你意识到的更大的权力:真正改变社会的权力。在成为一名杰出工程师的过程中,至关重要的是,你要明白在不造成伤害的情况下行使权力所需的内在责任。第一步是认识到由许多社会和教育因素造成的偏见的默认状态。认识到这一点之后,你将能够考虑经常被遗忘的用例或用户,他们可能会从你构建的产品中受益或受到伤害。

行业继续向前发展,以越来越快的速度为人工智能(AI)和机器学习构建新的用例。为了保持竞争力,我们致力于规模化和高效化,打造一支高素质的工程和技术队伍。然而,我们需要停下来思考这样一个事实:今天,有些人有能力设计技术的未来,而另一些人却没有。我们需要了解,我们构建的软件系统是否会消除整个人口共享繁荣和平等使用技术的潜力。

从历史上看,当公司面临一个抉择:是完成一个能够推动市场主导地位和收入增长的战略目标,还是一个可能减缓实现这一目标的势头的目标时,它们往往会选择速度和股东价值。这种倾向因许多公司重视个人绩效和卓越表现,但往往无法有效地在所有领域推动产品公平性问责制而加剧。关注代表性不足的用户显然是促进公平的机会。为了继续在技术领域保持竞争力,我们需要学会为全球公平性而进行设计。

如今,当公司设计技术来扫描、捕捉和识别街上行人时,我们感到担忧。我们担心隐私问题,以及政府现在和将来会如何使用这些信息。然而,大多数技术人员并不具备代表性不足的群体的必要视角,无法理解种族差异对面部识别的影响,也无法理解应用人工智能如何导致有害和不准确的结果。

目前,人工智能驱动的面部识别软件仍然不利于有色人种或少数民族。我们的研究不够全面,没有涵盖足够广泛的不同肤色。如果训练数据和软件创建者都只代表一小部分人,我们就不能指望输出是有效的。在这种情况下,我们应该愿意推迟开发,以便尝试获得更完整、更准确的数据,以及更全面、更包容的产品。

然而,数据科学本身对于人类而言很难评估。即使我们确实有代表性,训练集仍然可能存在偏见并产生无效结果。2016 年完成的一项研究发现,超过 1.17 亿美国成年人被纳入执法部门的面部识别数据库。⁵由于对黑人社区的监管不力以及逮捕结果的差异,在面部识别中使用此类数据库可能会出现种族偏见的错误率。尽管该软件的开发和部署速度不断加快,但独立测试的速度却没有加快。

为了纠正这一严重失误,我们需要保持正直,放慢速度,确保我们的输入包含尽可能少的偏见。谷歌现在在人工智能的背景下提供统计培训,以帮助确保数据集本质上没有偏见。

因此,将你的行业经验重点转移到更全面、多元文化、种族和性别研究教育上不仅是你的责任,也是你的雇主的责任。科技公司必须确保其员工不断接受专业发展,而且这种发展是全面的、多学科的。要求不是一个人独自承担起学习其他文化或其他人口结构的责任。改变要求我们每个人,无论是个人还是团队领导者,都投资于持续的专业发展,这不仅可以培养我们的软件开发和领导技能,还可以培养我们理解整个人类多样化经历的能力。

让多元化变得可行

如果我们愿意承认,我们所有人都应对科技行业存在的系统性歧视负责,那么系统性公平和公正是可以实现的。我们要为系统的失败负责。推迟或抽象个人责任是无效的,而且根据你的角色,这可能是不负责任的。将你所在的公司或团队的动态完全归咎于造成不平等的最大社会问题也是不负责任的。多元化支持者和反对者都喜欢这样一句话:“我们正在努力解决(插入系统性歧视话题),但问责很难。”

我们如何对抗(数百年来)历史性的歧视?”这一研究方向是转向更哲学或学术的对话,远离改善工作条件或成果的集中努力。建设多元文化能力的一部分需要更全面地了解社会不平等制度如何影响工作场所,特别是在技术领域。

⁵ Stephen Gaines 和 Sara Williams。“永恒的阵容:美国不受监管的警察人脸识别。”

乔治城大学法学院隐私与技术中心,2016 年 10 月 18 日。

如果您是一名工程经理,致力于招聘更多来自代表性不足的群体的人,那么回顾世界上歧视的历史影响是一项有用的学术练习。然而,至关重要的是要超越学术对话,关注可以采取的可量化和可操作的步骤,以推动公平和公正。例如,作为招聘软件工程师经理,您有责任确保您的候选人名单是平衡的。候选人评审池中是否有女性或其他代表性不足的群体?在您雇用某人之后,您提供了哪些成长机会,机会分配是否公平?每位技术主管或软件工程经理都有办法增强团队的公平性。重要的是,我们必须承认,尽管存在重大的系统性挑战,但我们都是系统的一部分。这是我们要解决的问题。

拒绝单一方法

我们不能延续那些只提供单一理念或方法来解决技术部门不平等问题的解决方案。我们的问题复杂且多因素。因此,我们必须打破提高职场代表性的单一方法,即使这些方法是由我们钦佩的人或拥有机构权力的人所倡导的。

科技行业普遍持有一个观点是,劳动力缺乏代表性的问题只能通过修复招聘渠道来解决。是的,这是一个根本性的举措,但这不是我们需要立即解决的问题。例如,我们需要认识到晋升和留任方面的系统性不平等,同时关注更具代表性的招聘和跨种族、性别、社会经济和移民身份的教育差距。

在科技行业,许多来自代表性不足的群体的人每天都在错过机会和晋升。黑人+谷歌员工的流失率超过了所有其他群体的流失率并阻碍了代表性目标的实现。如果我们想推动变革并提高代表性,我们需要评估我们是否正在创建一个所有有抱负的工程师和其他技术专业人员都能蓬勃发展的生态系统。

全面了解整个问题空间对于确定如何解决问题至关重要。

从关键数据迁移到招聘具有代表性的员工,这一点适用于所有事情。例如,如果你是一名工程经理,想要招聘更多女性,那么不要只专注于建立人才管道。关注招聘、留任和晋升生态系统的其他方面,以及它对女性的包容性。考虑一下你的招聘人员是否表现出识别女性和男性优秀候选人的能力。如果你管理着一支多元化的工程团队,那么要关注心理安全,并投资于提高团队的多元文化能力,让新团队成员感到受欢迎。

如今,一种常见的方法是先为大多数用例构建,然后将解决极端情况的改进和功能留到以后。但这种方法是有缺陷的;它让那些已经在技术方面占优势的用户抢占先机,从而加剧了不平等。将所有用户群体的考虑都推迟到设计接近完成时,这降低了优秀工程师的标准。相反,通过从一开始就构建包容性设计并提高开发标准,使工具令人愉悦,并让那些难以获得技术的人能够使用,我们可以增强所有用户的体验。

为最不像你的用户设计不仅是明智的,也是最佳实践。所有技术人员,无论领域如何,在开发避免不利于用户或低估用户的产品时,都应该考虑一些务实且直接的后续步骤。首先要进行更全面的用户体验研究。这项研究应该针对多语言、多文化、跨越多个国家、社会经济阶层、能力和年龄范围的用户群体进行。首先关注最困难或代表性最低的用例。

挑战既定流程

挑战自己建立更公平的系统不仅仅是设计更具包容性的产品规范。建立公平的系统有时意味着挑战导致无效结果的既定流程。

考虑一个最近评估公平影响的案例。在谷歌,几个工程团队合作建立了一个全球招聘申请系统。该系统支持外部招聘和内部流动。参与的工程师和产品经理很好地倾听了他们认为是核心用户群的要求:招聘人员。招聘人员专注于最大限度地减少招聘经理和申请人浪费的时间,他们向开发团队展示了专注于这些人的规模和效率的用例。为了提高效率,招聘人员要求工程团队添加一项功能,一旦内部调动人员对某个工作表示感兴趣,就会向招聘经理和招聘人员突出显示绩效评级(特别是较低的评级)。

从表面上看,加快评估流程并帮助求职者节省时间是一个伟大的目标。那么潜在的公平问题在哪里呢?提出了以下公平问题:

- 发展评估是否是绩效的预测性衡量标准? · 向未来经理人提供的绩效评估是否没有个人偏见?
- 整个组织的绩效评估分数是否标准化?

如果对其中任何一个问题的回答是“否”,那么绩效评级仍然可能导致不公平的、因而无效的结果。

当一位杰出的工程师质疑过去的表现是否能预测未来的表现时,评审小组决定进行彻底的评审。最后,评审小组确定,如果候选人找到了新团队,他们很可能会克服糟糕的表现评级。事实上,他们获得满意或模范表现评级的可能性与从未获得过糟糕评级的候选人一样大。简而言之,绩效评级仅表明一个人在接受评估时在其既定角色中的表现。评级虽然是衡量特定时期绩效的重要方法,但不能预测未来的表现,不应被用来衡量候选人是否准备好担任未来的角色,也不应用于评估内部候选人是否有资格加入其他团队。

(但是,它们可用于评估员工在当前团队中的位置是否合适;因此,它们可以提供机会来评估如何更好地支持内部候选人向前发展。)

这个分析肯定占用了大量的项目时间,但积极的结果是一个更加公平的内部流动过程。

价值与结果

Google 在招聘方面有着良好的投资记录。如上例所示,我们还不断评估我们的流程,以提高公平性和包容性。更广泛地说,我们的核心价值观是基于尊重和对多元化和包容性员工队伍的坚定承诺。然而,年复一年,我们在招聘代表全球用户的代表性员工队伍方面也未能如愿。尽管制定了政策和计划来帮助和支持包容性举措并促进招聘和晋升方面的卓越表现,但改善公平结果的努力仍在继续。失败点不在于公司的价值观、意图或投资,而在于在实施层面应用这些政策。

旧习惯很难改变。你今天可能习惯为之设计的用户(也就是你习惯从他们那里获得反馈的用户)可能并不代表你需要接触的所有用户。我们在各种产品中都经常看到这种情况,从不适合女性身体的可穿戴设备到不适合深色肤色人群的视频会议软件。

那么,出路何在?

1.好好审视一下自己。在 Google,我们有“为每个人打造”的品牌口号。如果我们没有代表性的员工队伍或以社区反馈为中心开展工作的参与模式,我们怎么才能为每个人打造产品呢?我们

不能。事实是,我们有时公开未能保护我们最脆弱的用户免受种族主义、反犹太主义和恐同内容的侵害。

2.不要为每个人而建。我们要与每个人一起建设。我们还没有为每个人建设。这项工作不是在真空中进行的,当技术还不能代表整个人口时,这项工作当然不会发生。话虽如此,我们不能收拾行李回家。那么我们如何为每个人建设呢?我们要与我们的用户一起建设。我们需要让全人类的用户参与进来,并有意将最脆弱的群体置于我们设计的中心。他们不应该是事后才想到的。

3.为使用产品最困难的用户进行设计。

为那些面临更多挑战的人打造产品将使产品更适合所有人。另一种思考方式是:不要为了短期速度而牺牲公平性。

4.不要假设公平;要在整个系统中衡量公平。要认识到决策者也会受到偏见的影响,并且可能对不公平的原因了解不足。您可能没有专业知识来识别或衡量公平问题的范围。迎合一个用户群可能意味着剥夺另一个用户群的权利;这些权衡可能很难发现,而且不可能逆转。

与多元化、公平性和包容性方面的主题专家个人或团队合作。

5.改变是可能的。我们今天面临的技术问题,从监视到虚假信息到网络骚扰,确实令人难以承受。

我们不能用过去失败的方法或仅靠我们已有的技能来解决这些问题。我们需要做出改变。

保持好奇,不断前进

实现公平的道路漫长而复杂。然而,我们可以也应该从简单地构建工具和服务过渡到加深对我们设计的产品如何影响人类的理解。挑战我们的教育、影响我们的团队和经理以及进行全面的用户研究都是取得进步的方法。虽然改变令人不舒服,实现高绩效的道路也可能很痛苦,但通过合作和创造力是可能的。

最后,作为未来的杰出工程师,我们应该首先关注受偏见和歧视影响最大的用户。通过专注于持续改进和承认失败,我们可以共同努力加速进步。成为一名工程师是一个复杂而持续的过程。目标是做出改变,推动人类前进,而不会进一步剥夺弱势群体的权利。作为未来的杰出工程师,我们相信我们可以防止系统在未来出现故障。

结论

开发软件和发展软件组织是一项团队工作。随着软件组织的扩大,它必须响应并充分设计其用户群,在当今互联的计算世界中,这涉及到本地和世界各地的每个人。必须付出更多努力,使设计软件的开发团队和他们生产的产品都能反映出如此多样化和包罗万象的用户群体的价值观。而且,如果工程组织想要扩大规模,就不能忽视代表性不足的群体;来自这些群体的工程师不仅可以增强组织本身,还可以为设计和实现真正对整个世界有用的软件提供独特且必要的视角。

TL;DR

- 偏差是默认值。
- 多样性对于为全面的用户群进行适当设计是必不可少的。 · 包容性不仅对于改善代表性不足的群体的招聘渠道至关重要,而且对于为所有人提供真正支持性的工作环境也至关重要。 · 产品速度必须根据提供对所有用户真正有用的产品来评估。 放慢速度总比发布可能对某些用户造成伤害的产品要好。

第五章

如何领导团队

作者:布莱恩·菲茨帕特里克
编辑:Riona MacNamara

到目前为止,我们已经讨论了软件编写团队的文化和组成方面的很多内容,在本章中,我们将研究最终负责使一切正常运转的人员。

没有领导者,任何团队都无法正常运作,尤其是在谷歌,工程几乎完全是团队工作。在谷歌,我们认识到两种不同的领导角色。经理是人的领导者,而技术主管领导技术工作。虽然这两个角色的职责需要类似的规划技能,但它们需要的人际交往技能却截然不同。

没有船长的船只不过是一间漂浮的候船室:除非有人抓住舵并启动引擎,否则它只会随波逐流,漫无目的地漂流。软件就像那艘船:如果没有驾驶它,你就只剩下一群工程师在浪费宝贵的时间,只是坐在那里等待某事发生(或者更糟的是,仍然在编写你不需要的代码)。虽然本章是关于人员管理和技术领导力的,但如果你是一个个人贡献者,它仍然值得一读,因为它可能会帮助你更好地了解你自己的领导者。

经理和技术主管(或两者)

虽然每个工程团队通常都有一位领导者,但他们以不同的方式获得这些领导者。这在谷歌确实如此;有时是一位经验丰富的经理来管理一个团队,有时是一位个人贡献者被提拔到领导职位(通常是在较小的团队中)。

在新兴团队中,这两个角色有时会由同一个人担任:技术主管经理 (TLM)。在较大的团队中,经验丰富的人事经理将接任管理角色,而经验丰富的高级工程师将接任技术主管角色。尽管经理和技术主管在工程团队的成长和生产力中都发挥着重要作用,但成功胜任每个角色所需的人事技能却大不相同。

工程经理许多公司聘请受过培训的人力资源

经理来管理他们的工程团队,而这些人可能对软件工程知之甚少。然而,谷歌很早就决定,其软件工程经理应该有工程背景。这意味着要聘请曾经是软件工程师的经理,或者培训软件工程师成为经理(稍后会详细介绍)。

从最高层来看,工程经理要对团队中每个人(包括技术主管)的绩效、生产力和幸福感负责,同时还要确保他们负责的产品能够满足业务需求。由于业务需求和团队成员个人的需求并不总是一致的,这通常会让经理陷入困境。

技术主管

团队的技术主管 (TL) 通常会向团队经理汇报工作,负责(令人惊讶的是!)产品的技术方面,包括技术决策和选择、架构、优先级、速度和一般项目管理(尽管在较大的团队中,他们可能会有项目经理协助处理这些事务)。TL 通常会与工程经理携手合作,以确保团队配备足够的产品人员,并确保工程师能够从事与他们的技能和技能水平最匹配的任务。大多数 TL 也是个人贡献者,这通常会迫使他们选择是自己快速完成某件事还是委托给团队成员(有时)更慢地完成。随着团队规模和能力的扩大,后者通常是正确的决定。

技术主管经理在小型和新兴团队中,工程

经理需要具备强大的技术技能,因此通常默认会配备一名技术主管经理:即一名能够同时处理团队人员和技术需求的人员。有时,技术主管经理是更资深的人员,但更多时候,该角色由最近才成为个人贡献者的人担任。

在 Google,规模较大、成熟的团队通常会有两位领导者（一位是技术总监,另一位是工程经理）作为合作伙伴一起工作。理论上讲，很难同时（出色地）做好两份工作而不至于完全精疲力竭,所以最好让两位专家专注于每一项工作。

TLM 的工作很棘手,通常需要 TLM 学习如何平衡个人工作、授权和人员管理。因此,它通常需要经验丰富的 TLM 提供大量指导和帮助。(事实上,我们建议新任 TLM 除了参加 Google 提供的有关此主题的许多课程外,还要寻找一位资深导师,以便在他们成长为 TLM 的过程中定期为他们提供建议。)

案例研究:无需授权即可产生影响人们普遍认为,你

可以让下属为你的产品完成工作,但当你需要让组织外的人（有时甚至是产品领域外的人）去做你认为需要做的事情时,情况就不同了。这种“无需授权即可产生影响”是你可以培养的最强大的领导特质之一。

例如,多年来,高级工程研究员、可能是谷歌内部最知名的谷歌人杰夫·迪恩 (Jeff Dean) 只领导了谷歌一小部分工程团队,但他对技术决策和方向的影响力却延伸到整个工程组织甚至更远（这要归功于他在公司外的写作和演讲）。

另一个例子是我创办的一个名为“数据解放阵线”的团队:我们只有不到六名工程师,却成功让 50 多种 Google 产品通过我们推出的一款名为 Google Takeout 的产品导出数据。当时,Google 高管层并没有正式指示所有产品都成为 Takeout 的一部分,那么我们如何让数百名工程师为这项工作做出贡献呢?我们的方法就是确定公司的战略需求,展示它与公司使命和现有优先事项之间的关联,并与一小群工程师合作开发一种工具,使团队能够快速轻松地与 Takeout 集成。

从个人贡献者角色转变为领导角色

无论他们是否被正式任命,如果你的产品要走向世界,就需要有人来掌舵,如果你是那种积极进取、没有耐心的人,那么这个人可能就是你。你可能会发现自己被卷入帮助

你的团队解决冲突、做出决策并协调人员。这种事情总是在发生,而且往往是偶然发生的。也许你从未打算成为一名“领导者”,但不知何故它还是发生了。有些人将这种痛苦称为“管理倦怠症”。

即使你发誓永远不会成为一名经理,但在你职业生涯的某个阶段,你很可能发现自己处于领导地位,特别是如果你在自己的角色中取得了成功。本章的其余部分旨在帮助你了解当这种情况发生时该怎么做。

我们在这里并不是试图说服你成为一名经理,而是要帮助你说明为什么最好的领导者会用谦逊、尊重和信任的原则来为他们的团队服务。了解领导力的来龙去脉是影响工作方向的一项重要技能。如果你想为你的项目掌舵,而不是随波逐流,你需要知道如何导航,否则你会把自己(和你的项目)推到沙洲上。

唯一需要担心的是……嗯,一切除了大多数人听到“经

理”这个词时普遍感到的不适之外,大多数人不想成为经理还有许多原因。在软件开发领域,你会听到的最大原因是你花在编写代码上的时间少得多。无论你是成为TL还是工程经理,情况都是如此,我将在本章后面详细讨论这一点,但首先,让我们介绍一下为什么我们大多数人不愿意成为经理的更多原因。

如果你的大部分职业生涯都在编写代码,那么在一天结束时,你通常会指着某样东西

无论是代码、设计文档还是刚刚关闭的一堆错误说:“这就是我今天所做的事情。”但是在忙碌的“管理”一天结束时,你通常会发现自己在想:“我今天什么都没做。”这就相当于你花了数年时间计算每天摘的苹果数量,然后换了一份种香蕉的工作,每天结束时却对自己说:“我什么苹果都没摘”,开心地忽略了旁边茂盛的香蕉树。量化管理工作比计算你生产出的小部件更困难,但让你的团队快乐高效地工作是衡量你工作水平的重要标准。不要陷入种香蕉时数苹果的陷阱。¹另一个不能成为经理的重要原因往往是不言而喻的,但根源于著名的“彼得原理”,即“在等级制度中,每个员工都倾向于升到他无法胜任的水平。”谷歌通常会通过要求一个人在一段时间内从事高于其当前水平的工作(即“超过”

¹另一个需要适应的区别是,我们作为管理者所做的事情通常会在一段时间内产生回报。
更长的时间线。

大多数人都曾遇到过无法胜任工作或管理能力很差的经理,²我们也知道有些人只为糟糕的经理工作。如果你在整个职业生涯中只接触过糟糕的经理,那你为什么还想当经理呢?你为什么想晋升到一个你觉得无法胜任的职位呢?

但是,有很多理由可以考虑成为 TL 或经理。首先,这是扩展自己的一种方式。即使你擅长编写代码,你编写的代码量仍然有上限。想象一下,在你的领导下,一个由优秀工程师组成的团队可以编写多少代码!其次,你可能真的很擅长它 许多发现自己陷入项目领导真空的人发现,他们非常擅长提供团队或公司所需的指导、帮助和空中掩护。总得有人来领导,为什么不是你呢?

仆人式领导似乎有一种疾

病会困扰管理者,他们会忘记管理者对他们所做的所有可怕的事情,并突然开始做同样的事情来“管理”下属。这种疾病的症状包括但不限于微观管理、忽视表现不佳的人和雇用软弱的人。如果不及时治疗,这种疾病可能会毁掉整个团队。

我刚成为谷歌经理时收到的最好建议来自当时的工程总监史蒂夫·温特。他说:“最重要的是,抵制管理的冲动。”新晋经理最大的冲动之一就是积极“管理”员工,因为这就是经理的职责,对吧?这通常会带来灾难性的后果。

治愈“管理”疾病的方法是自由运用“仆人式领导”,这是一种很好的表达方式,表明作为领导者,你能做的最重要的事情就是为你的团队服务,就像管家或管家照顾家庭的健康和幸福一样。作为仆人式领导者,你应该努力创造一种谦逊、尊重和信任的氛围。这可能意味着消除团队成员自己无法消除的官僚障碍,帮助团队达成共识,甚至在团队在办公室加班时为他们买晚餐。仆人式领导者填补漏洞,为团队铺平道路,并在必要时为他们提供建议,但仍然不怕弄脏自己的手。仆人式领导者所做的唯一管理就是管理团队的技术和社会健康;尽管很诱人

²公司不应该强迫员工进入管理层作为职业道路的一部分,还有一个原因是:如果一名工程师能够编写大量出色的代码,但根本不想管理人员或领导团队,那么强迫他们进入管理或 TL 角色,您就会失去一名优秀的工程师,并获得一名糟糕的经理。这不仅是一个坏主意,而且会造成严重伤害。

虽然可能只关注团队的技术健康,但团队的社会健康也同样重要(但通常管理起来更加困难)。

工程经理

那么,现代软件公司对经理的期望究竟是什么?在计算机时代之前,“管理”和“劳动”可能扮演着几乎对立的角色,经理掌握着所有的权力,而劳动则需要集体行动来实现自己的目的。但现代软件公司的工作方式并非如此。

经理是一个四个字母的词在谈论 Google

工程经理的核心职责之前,让我们先回顾一下经理的历史。当今“尖头发经理”的概念部分是延续下来的,首先来自军事等级制度,后来被工业革命所采用 距今已有 100 多年历史!工厂开始在各地涌现,它们需要工人(通常是非熟练工人)来维持机器的运转。因此,这些工人需要主管来管理他们,而且由于很容易用其他急需工作的人来取代这些工人,经理们没有什么动力善待员工或改善他们的工作条件。不管是否人道,当员工除了执行机械任务之外无事可做时,这种方法多年来一直很有效。

管理者对待员工的方式通常与马车夫对待骡子的方式相同:他们交替用胡萝卜引导员工前进,当这种方法不起作用时,就用棍子鞭打员工,以此激励员工。这种胡萝卜加大棒的管理方法在从工厂³过渡到现代办公室的过程中仍然存在,在 20 世纪中叶,当员工在同一份工作上工作多年时,强硬的管理者作为骡车夫的刻板印象盛行。

如今,这种现象在某些行业仍在继续,甚至在需要创造性思维和解决问题的行业也是如此,尽管许多研究表明,过时的“胡萝卜加大棒”政策对创造性人才的生产力是无效的,而且有害的。几年前的流水线工人可以在几天内接受培训并随时被替换,而从事大型代码库工作的软件工程师可能需要数月才能适应新团队。与可替换的流水线工人不同,这些人需要培养、时间和空间来思考和创造。

³要了解有关优化工厂工人流动的更多有趣信息,请阅读《科学管理》或《泰勒主义》,尤其是其对工人士气的影响。

当今的工程经理大多数人仍然使用“经

理”这个头衔,尽管它往往是一种不合时宜的称呼。这个头衔本身往往会鼓励新经理管理他们的下属。经理最终可能会表现得像父母⁴,因此员工的反应就像孩子一样。

以谦逊、尊重和信任为背景来阐述这一点:如果经理明确表示信任员工,员工就会感受到不辜负这种信任的积极压力。就这么简单。优秀的经理会为团队开辟道路,关注他们的安全和福祉,同时确保满足他们的需求。如果您从本章中记住了一件事,那就是:

传统的经理担心如何完成任务,而优秀的经理则担心完成什么任务(并且相信他们的团队能够找到如何完成任务的方法)。

几年前,一位新工程师杰瑞加入了我的团队。杰瑞的前任经理(在另一家公司)坚持要求他每天从9:00到5:00必须坐在办公桌前,并认为如果他不在那里,就是工作不够(当然,这是一个荒谬的假设)。杰瑞第一天和我一起工作,下午4:40来找我,结结巴巴地道歉说他必须提前15分钟离开,因为他有一个无法重新安排的约会。我看着他,笑了笑,直截了当地告诉他,“听着,只要你完成了你的工作,我不在乎你什么时候离开办公室。”杰瑞茫然地盯着我看了几秒钟,点点头,然后走了。我把杰瑞当成大人一样对待;他总是能完成工作,我从来不用担心他坐在办公桌前,因为他不需要保姆来完成工作。如果你的员工对他们的工作如此不感兴趣,以至于他们实际上需要传统经理的照顾才能被说服去工作,那么这才是你真正的问题。

失败是一种选择激励团

队的另一种方法是让他们感到安全和有保障,这样他们就可以通过建立心理安全来承担更大的风险。这意味着你的团队成员觉得他们可以做自己,而不必担心你或他们的团队成员的负面影响。风险是一个迷人的东西;大多数人都不善于评估风险,大多数公司都会不惜一切代价避免风险。因此,通常的做法是保守地工作,专注于较小的成功,即使承担更大的风险可能意味着成倍的成功。谷歌有一句俗语说,如果你试图实现一个不可能的目标,你很有可能会失败,但如果你试图实现不可能的事情却失败了,你很可能会取得比你仅仅尝试做某事时取得的成就更大的成就。

⁴如果你有孩子,你很可能会清晰地记得,你第一次对你的孩子说了什么话,让你停下来并惊呼(甚至大声喊出):“天哪,我成了我妈妈了。”

知道你能完成。建立一种接受冒险的文化的一个好方法是让你的团队知道失败是可以的。

所以,让我们先把这个问题搞清楚:失败是可以的。事实上,我们喜欢把失败看作是一种快速学习的方式(前提是你不是在同一件事上反复失败)。此外,重要的是将失败视为学习的机会,而不是指责或推卸责任。快速失败是好的,因为风险不大。慢慢失败也可以教会我们宝贵的教训,但它更痛苦,因为风险更大,损失也更大(通常是工程时间)。以影响客户的方式失败可能是我们遇到的最不受欢迎的失败,但这也是我们拥有最多结构来从失败中吸取教训的失败。如前所述,每次Google发生重大生产故障时,我们都会进行事后分析。此程序是一种记录导致实际故障的事件并制定一系列步骤以防止将来发生故障的方法。这既不是指责的机会,也不是旨在引入不必要的官僚检查;相反,目标是集中精力解决问题的核心,一劳永逸地解决问题。这非常困难,但非常有效(而且具有宣泄作用)。

个人的成功和失败有些不同。赞扬个人的成功是一回事,但在失败的情况下寻求个人责任是分裂团队和阻止所有人冒险的好方法。失败是可以的,但作为一个团队失败并从失败中吸取教训。如果一个人成功了,在团队面前表扬他们。如果一个人失败了,私下里提出建设性的批评。⁵无论如何,利用这个机会,用谦逊、尊重和信任的态度帮助你的团队从失败中吸取教训。

反模式

在我们介绍成功的技术总监和项目经理的一系列“设计模式”之前,我们将回顾一下如果你想成为一名成功的经理,你不想遵循的一系列模式。我们在职业生涯中遇到过一些糟糕的经理,而且在很多情况下,我们自己也遇到过这些破坏性的模式。

⁵公开批评个人不仅无效(这会让别人处于防御状态),而且很少有必要,而且通常只是卑鄙或残酷的。你可以肯定团队的其他成员已经知道某个人什么时候失败了,所以没有必要再提起这件事。

反模式：雇佣软弱的人

如果你是一名经理，并且你对自己的角色没有安全感（无论出于何种原因），那么确保没有人质疑你的权威或威胁到你的工作方法之一就是雇佣你可以欺负的人。你可以雇佣不如你聪明或有抱负的人，或者比你更没有安全感的人来达到这个目的。尽管这会巩固你作为团队领导者和决策者的地位，但也意味着你要做更多的工作。如果你不牵着他们，你的团队就无法采取行动。如果你组建了一支软弱的团队，你可能无法休假；一旦你离开房间，生产力就会急剧下降。但这肯定是让你在工作中感到安全所付出的一点小代价，对吧？

相反，你应该努力雇佣比你聪明、可以替代你的人。这可能很难，因为这些人会定期挑战你（除了在你犯错时让你知道）。这些人也会不断给你留下深刻印象，并创造伟大的事情。他们将能够在更大程度上自我指导，有些人也会渴望领导团队。你不应该把这看作是篡夺权力的企图；相反，把它看作是一个机会，让你领导一个额外的团队，探索新的机会，甚至休假，而不必担心每天检查团队以确保他们完成工作。这也是一个学习和成长的好机会。当你身边都是比你聪明的人时，扩展你的专业知识要容易得多。

反模式：忽视低绩效者

在我担任谷歌经理的职业生涯初期，有一次我向我的团队发放奖金信，我笑着告诉我的经理，“我喜欢做经理！”

我的经理，一位资深的业内人士，毫不犹豫地回答道：“有时候你可以扮演牙仙子，有时候你必须扮演牙医。”

拔牙从来都不是一件有趣的事。我们见过团队领导做了所有正确的事情来组建非常强大的团队，但这些团队却因为一两个表现不佳的人而无法出类拔萃（并最终分崩离析）。我们知道，人为因素是编写软件最具挑战性的部分，但与人打交道最困难的部分是处理那些没有达到预期的人。有时，人们没有达到预期是因为他们工作时间不够长或不够努力，但最困难的情况是，无论他们工作多长时间或多努力，某人都无法胜任自己的工作。

Google 的站点可靠性工程 (SRE) 团队有一句座右铭：“希望不是策略”。

在处理表现不佳的员工时，人们最常滥用希望这一策略。大多数团队领导咬紧牙关，避开目光，只希望表现不佳的员工

表现者要么奇迹般地进步,要么就消失。然而,这种情况极其罕见。

当领导者满怀希望,而表现不佳的人却没有进步(或离开)时,团队中的高绩效者却浪费宝贵的时间来拉拢表现不佳的人,团队士气也随之消散。即使你忽略了他们,你也可以肯定团队知道表现不佳的人在那里。事实上,团队非常清楚谁是表现不佳的人,因为他们必须支持他们。

忽视低绩效者不仅会阻止新的高绩效者加入您的团队,而且还会鼓励现有的高绩效者离开。最终,您将得到一个由低绩效者组成的团队,因为他们是唯一不能自愿离开的人。最后,让低绩效者留在团队中甚至不会给他们带来任何好处;通常,在您的团队中表现不佳的人实际上可能会在其他地方产生很大的影响。

尽快处理低绩效员工的好处是,你可以把自己置于帮助他们提高或摆脱困境的位置。如果你立即处理低绩效员工,你通常会发现他们只需要一些鼓励或指导就能进入更高的生产力状态。如果你等得太久才处理低绩效员工,他们与团队的关系就会变得非常糟糕,你会非常沮丧,无法帮助他们。

如何有效地指导低绩效员工?最好的比喻是想象你正在帮助一个跛脚的人重新学会走路,然后慢跑,然后与团队的其他成员一起奔跑。这几乎总是需要暂时的微观管理,但仍然需要大量的谦逊、尊重和信任。尤其是尊重。设定一个特定的时间框架(比如两个月)和一些你希望他们在这段时间内实现的非常具体的目标。让目标小、渐进、可衡量,这样就有机会取得许多小成功。每周与团队成员会面检查进度,并确保你对每个即将到来的里程碑设定非常明确的期望,以便轻松衡量成功或失败。如果低绩效者无法跟上,你们双方在过程的早期就会非常明显地意识到这一点。

此时,员工通常会承认事情进展不顺利,并决定辞职;在其他情况下,员工会下定决心,并“提高水平”以满足期望。无论哪种方式,通过直接与表现不佳的员工合作,您都在催化重要且必要的变化。

反模式:忽视人为问题经理对其团队有两个

主要关注领域:社交和技术。

在谷歌,经理们更擅长技术是很常见的,而且由于大多数经理都是从技术岗位晋升而来的(技术岗位的主要目标是解决技术问题),他们往往会忽视人性问题。你很容易把所有的精力都集中在团队的技术方面

因为,作为个人贡献者,你花费了绝大多数时间来解决技术问题。当你还是学生时,你的课程都是关于学习工作的技术细节。但现在你是一名经理,你忽视了团队的人性因素,后果自负。

让我们先看一个领导者忽视团队中的人为因素的例子。几年前,杰克生了他的第一个孩子。杰克和凯蒂已经一起工作了好几年,既远程工作,又在同一间办公室工作,所以在新生儿出生后的几周里,杰克在家工作。这对夫妻来说效果很好,凯蒂也完全没问题,因为她已经习惯了和杰克一起远程工作。他们一如既往地高效工作,直到他们的经理帕布罗(在另一个办公室工作)发现杰克一周大部分时间都在家工作。帕布罗对杰克不去办公室和凯蒂一起工作感到很不高兴,尽管杰克和往常一样高效,凯蒂对这种情况也没什么意见。杰克试图向帕布罗解释,他和来办公室一样高效,而且他大部分时间在家工作几周对他和他的妻子来说要轻松得多。帕布罗的回答是:“老兄,人们总是有孩子。你需要去办公室。”不用说,杰克(通常是一位温文尔雅的工程师)被激怒了,对帕布罗失去了很多尊重。

帕布罗本可以用很多不同的方式来处理这件事:他可以表示理解,杰克想花更多时间待在家里陪伴妻子,如果他的工作效率和团队不受影响,就让他继续这样做一段时间。他可以和杰克商量,让杰克每周去办公室一两天,直到事情平息下来。不管最终结果如何,在这种情况下,一点点同理心都会让杰克高兴很多。

反模式:成为每个人的朋友大多数人第一次

尝试领导任何一种职位都是当他们成为他们以前是成员的团队的经理或 TL 时。许多领导不想失去他们与团队培养的友谊,因此他们有时会在成为团队领导后加倍努力地维持与团队成员的友谊。这可能是灾难的根源,并导致许多友谊破裂。

不要将友谊与软弱领导相混淆:当你掌握某人职业生涯的大权时,他们可能会感受到压力,而人为地回报友谊。

请记住,即使你不是团队的密友(或是一个铁杆硬汉),你也可以领导一个团队并达成共识。同样,你也可以成为一个强硬的领导者,而不必抛弃现有的友谊。我们发现,与你的团队共进午餐是一种有效的方式,可以与他们保持社交联系,而无需

让他们感到不舒服 这让你有机会在正常工作环境之外进行非正式的对话。

有时,要接替曾经的好朋友和同事担任管理职位可能很棘手。如果被管理的朋友不善于自我管理,也不努力工作,那么每个人都会感到压力。我们建议您尽可能避免陷入这种情况,但如果做不到,请特别注意与这些人的关系。

反模式:降低招聘标准史蒂夫·乔布斯曾经说过:“A类人会雇

佣其他A类人;B类人会雇佣C类人。”人们很容易成为这句格言的受害者,尤其是当你试图快速招聘时。我在Google之外看到的一种常见做法是,一个团队需要招聘5名工程师,因此它会筛选一堆申请,面试40或50人,然后挑选出最好的5名候选人,无论他们是否符合招聘标准。

这是建立一支平庸团队最快的方法之一。

寻找合适人选的成本 无论是支付招聘人员费用、投放广告还是四处寻找推荐人 与处理一个你根本不应该雇用的员工的成本相比都是微不足道的。

这种“成本”表现为团队生产力下降、团队压力、管理员工升迁或离职所花费的时间,以及解雇员工所涉及的文书工作和压力。当然,这是假设您试图避免将他们留在团队中的巨大成本。如果您管理的团队在招聘方面没有发言权,并且您对团队的招聘不满意,那么您需要全力以赴争取更高质量的工程师。如果您仍然得到不合格的工程师,也许是时候寻找另一份工作了。没有优秀团队的原材料,你注定要失败。

反面模式:像对待孩子一样对待您的团队向您的团队表明您不信任

它的最好方法就是像对待孩子一样对待团队成员 人们倾向于按照您对待他们的方式行事,因此如果您像对待孩子或囚犯一样对待他们,那么当他们表现出这种行为时请不要感到惊讶。您可以通过对他们进行微观管理或根本不尊重他们的能力并且不给他们机会对自己的工作负责来体现这种行为。如果由于您不信任他们而必须永久地对他们进行微观管理,那么您的招聘就失败了。好吧,除非您的目标是建立一支您可以用余生来照顾的团队,否则这肯定会失败。如果您雇用值得信任的人并向这些人表明您信任他们,他们通常会挺身而出(坚持我们之前提到的基本前提,即您雇用了好人)。

这种信任程度的结果一直延伸到办公和电脑用品等更普通的事物。再举一个例子，谷歌为员工提供了装满各种办公用品（例如，笔、笔记本和其他“传统”创作工具）的柜子，员工可以根据需要随意取用。IT 部门开设了许多“技术站”，提供自助服务区，就像一个小型电子产品商店。这些包含许多电脑配件和小玩意（电源、电缆、鼠标、USB 驱动器等），这些物品很容易被拿走并大量带走，但因为谷歌员工被信任可以检查这些物品，所以他们感到有责任做正确的事。许多来自典型公司的人听到这个消息后都惊恐不已，惊呼谷歌肯定因为有人“偷”这些物品而损失了巨额资金。这当然是可能的，但如果员工表现得像孩子一样，或者不得不浪费宝贵的时间正式要求廉价办公用品，那成本又如何呢？这肯定比几支笔和 USB 电缆的价格要贵。

积极模式

现在我们已经介绍了反模式，让我们来谈谈成功领导和管理的积极模式，这些模式是我们从谷歌的经历、观察其他成功的领导者以及最重要的，从我们自己的领导力导师那里学到的。

这些模式不仅是我们实施后取得巨大成功的模式，也是我们在所追随的领导者身上一直最尊重的模式。

放弃自我在前几

章中，我们第一次探讨谦逊、尊重和信任时谈到了“放弃自我”，但当你是团队领导者时，这一点尤其重要。这种模式经常被误解为鼓励人们成为受气包，让别人欺负他们，但事实并非如此。当然，谦逊和让别人利用你之间只有一线之隔，但谦逊并不等同于缺乏信心。你仍然可以有自信和意见，而不必成为一个自大狂。任何团队都难以处理强烈的个人自我意识，尤其是团队领导者。相反，你应该努力培养强大的集体团队自我意识和身份认同。

“放弃自我”的一部分是信任：你需要信任你的团队。这意味着尊重团队成员的能力和先前的成就，即使他们是团队的新成员。

如果你没有对团队进行微观管理，那么你可以肯定，一线工作人员比你更了解工作细节。这意味着，尽管你可能是推动团队达成共识并帮助设定方向的人，但如何实现目标的具体细节最好由将产品组合在一起的人来决定。这不仅让他们有更大的意识

不仅如此,他们还会对产品的成功(或失败)有更强的责任感和责任感。如果你有一个优秀的团队,并让他们为工作质量和速度设定标准,那么他们取得的成就将比你用胡萝卜加大棒监督团队成员要大得多。

大多数新担任领导职务的人都感到自己肩负着巨大的责任,要把每件事都做好,了解所有事情,并掌握所有的答案。我们可以向你保证,你不会把每件事都做好,也不会掌握所有的答案,如果你表现得像这样,你很快就会失去团队的尊重。这在很大程度上取决于你对自己的角色是否有基本的安全感。回想一下当你还是一个个体贡献者的时候,你在很远的地方就能闻到不安全感。试着欣赏探究:当有人质疑你做出的决定或声明时,请记住,这个人通常只是想更好地了解你。如果你鼓励探究,你就更有可能得到那种建设性的批评,这会让你成为更好团队的更好领导者。找到会给你好的建设性批评的人非常困难,而从“为你工作”的人那里得到这种批评就更加困难了。想想你作为一个团队试图完成的大局,并公开接受反馈和批评,避免领土侵犯的冲动。

放下自我的最后一部分很简单,但许多工程师宁愿被火烧也不愿这样做:犯错时道歉。我们并不是说你应该像在爆米花上撒盐一样在整个谈话过程中都说“对不起”你需要真诚地道歉。你肯定会犯错,无论你是否承认,你的团队都会知道你犯了错误。无论他们是否与你交谈,你的团队成员都会知道(有一件事是肯定的:他们会互相谈论这件事)。道歉不需要花钱。

人们非常尊重那些在犯错时道歉的领导者,而且与普遍的看法相反,道歉并不会让您变得脆弱。事实上,当你道歉时,你通常会赢得人们的尊重,因为道歉告诉人们你头脑冷静,善于评估情况,而且回到谦逊、尊重和信任谦逊。

做一名禅师

作为一名工程师,你可能已经形成了极好的怀疑和愤世嫉俗的意识,但当你试图领导一个团队时,这可能是一种负担。这并不是说你应该在每个转折点都天真地乐观,但你最好不要大声怀疑,同时让你的团队知道你意识到工作中的复杂性和障碍。当你领导更多的人时,调节你的反应和保持冷静就变得更加重要,因为你的团队会(有意识和无意识地)向你寻求如何行动和对周围发生的事情做出反应的线索。

直观地了解这一现象的一个简单方法是将公司的组织结构图看作一连串的齿轮，个人贡献者是末端只有几个齿的小齿轮，他们上面的每个经理是另一个齿轮，首席执行官是最大的齿轮，有数百个齿。这意味着，每当该人的“经理齿轮”（可能有几十个齿）旋转一圈时，“个人齿轮”就会旋转两到三圈。首席执行官只需轻轻一动，就能让倒霉的员工（位于六七个齿轮链末端）疯狂旋转！无论你是否愿意，你在链条上的位置越高，你下面的齿轮就能旋转得越快。

对此问题的另一种思考方式是“领导者永远在舞台上”这一格言。

这意味着，如果你处于明显的领导地位，你就会一直受到监视：不仅是在你主持会议或发表演讲时，甚至在你坐在办公桌前回复电子邮件时也是如此。你的同事会从你的肢体语言、你对闲聊的反应以及你吃午饭时的信号中寻找微妙的线索。

他们读懂的是信心还是恐惧？作为领导者，你的工作是激励他人，但激励他人是一项全天候的工作。你对一切事物（无论多么琐碎）的明显态度都会被无意识地注意到，并感染你的团队。

谷歌早期的管理者之一、工程副总裁比尔·库夫兰（Bill Coughran）确实掌握了时刻保持冷静的能力。无论发生什么事，无论发生什么疯狂的事情，无论风暴有多大，比尔都不会惊慌失措。

大多数时候，他会将一只手臂放在胸前，用手托着下巴，询问问题，通常是问给完全惊慌失措的工程师。这样做可以让他们平静下来，并帮助他们专注于解决问题，而不是像无头苍蝇一样四处乱窜。我们中的一些人曾经开玩笑说，如果有人来告诉比尔，公司有 19 个办事处遭到外星人袭击，比尔的回答会是：“知道他们为什么不把办事处数整整 20 个吗？”

这给我们带来了另一个禅宗管理技巧：提问。当团队成员向你征求建议时，通常会非常兴奋，因为你终于有机会解决某些问题了。这正是你在担任领导职务之前多年来所做的，所以你通常会立即进入解决方案模式，但这是你最不应该处于的位置。征求建议的人通常不希望你解决他们的问题，而是希望你帮助他们解决问题，而做到这一点的最简单方法就是向这个人提问。这并不是说你应该用一个魔术 8 球来代替自己，那样会让人抓狂，也毫无帮助。相反，你可以表现出谦逊、尊重和信任，并尝试通过尝试改进和探索问题来帮助这个人自己解决问题。这通常会引导员工找到答案，⁶而这个人的答案又会回到我们在本章前面讨论的所有权和责任。无论你是否拥有

⁶另请参阅“橡皮鸭调试”。

答案是,使用这种技巧几乎总会给员工留下你这样做的印象。很棘手,是吧?苏格拉底会为你感到骄傲的。

成为催化剂在

化学中,催化剂是一种加速化学反应但本身不会在反应中消耗的物质。催化剂(例如酶)发挥作用的方式之一是将反应物拉近:当催化剂帮助反应物聚集在一起时,反应物不会在溶液中随机弹跳,而是更有可能发生有利的相互作用。作为领导者,您经常需要扮演这个角色,您可以通过多种方式来实现这一目标。

团队领导者最常做的事情之一就是建立共识。这可能意味着你要从头到尾推动整个过程,或者只是轻轻地推动它朝着正确的方向发展以加快进程。建立团队共识是一项领导技能,非正式领导者经常使用这种技能,因为这是一种无需实际权力就能领导的方式。如果你有权力,你可以指挥和指示方向,但总体而言,这不如建立共识有效。⁷如果你的团队希望快速行动,有时会自愿将权力和方向让给一个或多个团队领导。尽管这看起来像独裁或寡头政治,但当它是自愿完成时,它就是一种共识。

消除障碍

有时,您的团队已经就您需要做的事情达成了共识,但遇到了障碍并陷入困境。这可能是技术或组织障碍,但介入帮助团队重新开始是一种常见的领导技巧。

有一些障碍,尽管你的团队成员几乎不可能克服,但你却可以轻松处理,并且让你的团队明白你很高兴(并且能够)帮助解决这些障碍是很有价值的。

有一次,一个团队花了数周时间试图解决与谷歌法律部门的障碍。当团队最终陷入集体困境并向经理提出问题时,经理在不到两个小时内就解决了问题,因为他知道该联系谁来讨论这个问题。还有一次,一个团队需要一些服务器资源,但无法分配。幸运的是,团队经理与公司其他团队保持沟通,并在当天下午就设法为团队提供了所需的资源。还有一次,一名工程师在处理一段晦涩难懂的 Java 代码时遇到了麻烦。尽管团队经理不是 Java 专家,但她还是能够让这位工程师与另一位工程师取得联系。

⁷试图达成 100% 的共识也可能有害。你需要能够决定继续进行,即使并非所有人都持有相同观点,或者仍存在一些不确定性。

工程师知道问题所在。您不需要知道所有答案来帮助消除障碍,但认识知道答案的人通常会有所帮助。

在许多情况下,认识对的人比认识对的人更有价值
回答。

成为一名教师和导师

作为 TL,最难处理的事情之一就是看着一个初级团队成员花 3 个小时处理你知道自己可以在 20 分钟内完成的事情。教导人们并让他们有机会自学一开始可能非常困难,但这是有效领导的重要组成部分。这对于新员工来说尤其重要,因为他们除了学习团队的技术和代码库外,还要学习团队的文化和承担的适当责任。一位好的导师必须平衡学员学习的时间和他们对产品的贡献时间,作为有效努力的一部分,随着团队的成长扩大团队规模。

与经理的角色非常相似,大多数人不会申请导师的角色 他们通常是在领导者寻找某人来指导新团队成员时才成为导师。成为导师不需要太多的正规教育或准备。首先,你需要具备三样东西:熟悉团队流程和系统、能够向他人解释事情,以及能够判断你的学员需要多少帮助。最后一件事可能是最重要的 你应该向你的学员提供足够的信息,但如果你过度解释事情或喋喋不休,你的学员可能会不理你,而不是礼貌地告诉你他们明白了。

设定明确的目标

这是那些听起来很明显但却被大量领导者忽视的模式之一。如果你要让你的团队朝着一个方向快速前进,你需要确保每个团队成员都理解并同意这个方向。想象一下你的产品是一辆大卡车 (而不是一系列管子)。

每个团队成员手里都拿着一根绳子,绳子绑在卡车前面,在处理产品时,他们会把卡车拉向自己的方向。如果你的目的是尽快把卡车 (或产品)拉向北方,你就不能让团队成员四处拉 你希望他们所有人都把卡车拉向北方。如果你要有明确的目标,你需要设定明确的优先事项,并帮助你的团队决定在时机成熟时应该如何权衡利弊。

设定明确目标并让团队将产品推向同一方向的最简单方法是为团队创建简明的使命宣言。在帮助团队确定方向和目标后,你可以退后一步,赋予团队更多自主权,定期检查以确保每个人都在正确的轨道上。这不是

不仅能腾出时间来处理其他领导任务,还能大幅提高团队效率。团队可以在没有明确目标的情况下取得成功(而且确实如此),但由于每个团队成员将产品拉向略有不同的方向,他们通常会浪费大量精力。这会让你感到沮丧,减缓团队的进度,并迫使你使用越来越多的精力来纠正方向。

诚实

这并不意味着我们假设你在对团队撒谎,但值得一提,因为你不可避免地会发现自己处于无法告诉你团队某些事情的境地,甚至更糟的是,你需要告诉每个人他们不想听到的事情。我们认识的一位经理告诉新团队成员:“我不会骗你,但当我不能告诉你某些事情或者我只是不知道时,我会告诉你。”

如果团队成员就您不能分享的事情找您,您可以告诉他们您知道答案,但您无权说任何话。更常见的情况是,当团队成员问您一些您不知道答案的问题时,您可以告诉对方您不知道。这是另一个在您阅读时似乎显而易见的事情,但许多担任经理职位的人认为,如果他们不知道某件事的答案,就证明他们很软弱或与时俱进。实际上,这只能证明他们是人。

给出严厉的反馈……确实很难。第一次告诉你的下属他们犯了错误或没有按预期完成工作时,压力会非常大。大多数管理文本建议你在提供严厉的反馈时使用“赞美三明治”来减轻打击。赞美三明治看起来是这样的:

您是团队中可靠的成员,也是我们最聪明的工程师之一。话虽如此,您的代码很复杂,团队中的其他人几乎不可能理解。但您潜力巨大,而且您的T恤系列非常酷。

当然,这可以减轻打击,但这种拐弯抹角的做法,大多数人离开会议时只会想:“太好了!我有很酷的T恤!”我们强烈建议不要使用三明治恭维法,不是因为我们认为你应该不必要地残忍或苛刻,而是因为大多数人不会听到批评信息,即需要改变某些事情。在这里可以运用尊重:在提出建设性批评时要善良和富有同理心,而不要诉诸三明治恭维法。事实上,如果你想让接受者听到批评而不是立即采取防御措施,善良和同理心是至关重要的。

几年前,一位同事从另一位经理那里招募了一位团队成员蒂姆,这位经理坚称蒂姆很难共事。他说蒂姆从不回应反馈或批评,而是一直做那些他被告知不该做的事情。我们的同事旁听了几次这位经理与蒂姆的会议,观察了这位经理和蒂姆之间的互动,并注意到这位经理

大量使用赞美三明治以免伤害蒂姆的感情。

当他们把蒂姆带入团队时,他们与他坐下来,非常清楚地解释说,蒂姆需要做出一些改变,以便更有效地与团队:

我们非常确定您没有意识到这一点,但您与团队互动的方式正在疏远和激怒他们,如果您想要有效,您需要提高您的沟通技巧,我们致力于帮助您做到这一点。

他们没有给 Tim 任何赞美或粉饰问题,但同样重要的是,他们并不刻薄 他们只是根据 Tim 在前一个团队的表现,陈述他们所看到的事实。瞧,在几周内(经过几次“复习”会议后),Tim 的表现有了显著改善。Tim 只是需要非常明确的反馈和指导。

当你提供直接反馈或批评时,你的表达方式是确保你的信息被听到而不是被转移的关键。如果你让接收者处于守势,他们不会考虑如何改变,而是会考虑如何与你争论,以表明你错了。我们的同事 Ben 曾经管理过一位工程师,我们称他为 Dean。Dean 有非常强烈的意愿,会与团队的其他成员争论任何事情。这可能是团队使命这么大的事情,也可能是网页上小部件的位置这么小的事情;Dean 会以同样的信念和激烈程度争论,他拒绝放过任何事情。经过几个月的这种行为,Ben 会见了 Dean,向他解释说他太好斗了。现在,如果 Ben 只是说:“Dean,别再这样混蛋了”,你可以肯定 Dean 会完全无视它。Ben 认真思考如何让 Dean 明白他的行为对团队产生了不利影响,他想出了以下比喻:

每次做出决定就像是一列火车穿过城镇 当你跳到火车前面阻止它时,你会减慢火车的速度,并可能惹恼驾驶火车的工程师。每 15 分钟就会有一列新火车经过,如果你跳到每列火车前面,你不仅会花很多时间阻止火车,而且最终驾驶火车的工程师之一会生气到直接撞倒你。所以,虽然跳到一些火车前面是可以的,但要挑选你想要阻止的火车,以确保你只阻止真正重要的火车。

这个轶事不仅给情况注入了一丝幽默,也使本和迪恩更容易讨论迪恩的“火车停下来”对团队的影响以及迪恩为此付出的精力。

追踪幸福感作为领导

者,长期来看,让你的团队更有效率(并且不太可能离开)的方法之一是花一些时间来衡量他们的幸福感。我们合作过的最佳领导者都是业余心理学家,他们会不时关注团队成员的福利,确保他们得到认可

做，并努力确保他们对自己的工作感到满意。我们知道有一位 TLM 会制作一个电子表格，记录所有需要完成的繁琐、吃力不讨好的任务，并确保这些任务在团队中平均分配。另一位 TLM 会关注其团队的工作时间，并利用补偿时间和有趣的团队郊游来避免倦怠和疲惫。还有一位 TLM 会通过处理技术问题来与团队成员进行一对一的会议，以此来打破僵局，然后花一些时间确保每位工程师都拥有完成工作所需的一切。

热身完毕后，他和工程师聊了一会儿，询问他们是否喜欢这份工作以及对下一步工作的期待。

追踪团队幸福感的一个简单好方法是在每次一对一会议结束时询问团队成员：“你需要什么？”这个问题是一种很好的总结方法，可以确保每个团队成员都拥有他们需要的东西，从而提高工作效率和幸福感，尽管你可能需要仔细探究才能获得详细信息。

如果每次一对一会议时你都问这个问题，你会发现你的团队最终会记住这一点，有时甚至会向你提供一份清单，列出使每个人的工作变得更好所需的事项。

意想不到的问题

在我加入谷歌后不久，我与当时的首席执行官埃里克·施密特进行了第一次会面，最后埃里克问我：“你需要什么吗？”我曾准备过无数种应对难题或挑战的防御性回答，但这次我完全没有准备。所以我坐在那里，目瞪口呆。下次再被问到这个问题时，你肯定已经准备好了！

作为一名领导者，关注一下团队在办公室之外的幸福感也是值得的。我们的同事 Mekka 在进行一对一沟通时，会要求下属以 1 到 10 的等级来评价他们的幸福感，而他的下属通常会以此来讨论办公室内外的幸福感。不要以为人们除了工作之外没有生活——对人们投入工作的时间抱有不切实际的期望会导致人们失去对你的尊重，更糟的是，会导致他们精疲力竭。我们并不是提倡你窥探团队成员的私生活，但是对团队成员所经历的个人情况保持敏感可以让你深入了解为什么他们在任何特定时间的效率可能会更高或更低。对目前在家中过得艰难的团队成员给予一点额外的宽容，可以让他们更愿意在你的团队稍后有紧迫的最后期限时投入更长的时间。

⁸谷歌还开展了一项名为“Googlegeist”的年度员工调查，对许多公司的员工幸福感进行评估。

尺寸。这提供了良好的反馈，但并不是我们所说的“简单”。

追踪团队成员幸福感的一个重要部分是追踪他们的职业生涯。如果你问团队成员他们认为自己五年后的职业生涯会如何,大多数时候你只会耸耸肩,一脸茫然。当被问到这个问题时,大多数人不会说太多,但通常每个人都希望在未来五年内做几件事:升职、学习新东西、开展重要工作、与聪明人一起工作。无论他们是否口头表达,大多数人都在考虑这些事情。如果你想成为一名有效的领导者,你应该考虑如何帮助实现所有这些目标,并让你的团队知道你在考虑这些事情。其中最重要的部分是将这些隐含的目标明确化,这样当你提供职业建议时,你就有一套真正的衡量标准来评估情况和机会。

追踪幸福感不仅意味着监测职业生涯,还意味着为团队成员提供提升自己的机会,让他们的工作得到认可,并在此过程中享受一点乐趣。

其他提示和技巧

以下是 Google 为您担任领导职务时推荐的其他各种提示和技巧:

授权,但要亲自动手 从个人贡献者角

色转变为领导角色时,实现平衡是最困难的事情之一。一开始,你倾向于自己做所有的工作,而担任领导角色很长时间后,你很容易养成不自己做任何工作的习惯。如果你是领导新手,你可能需要努力将工作委派给团队中的其他工程师,即使他们完成这项工作需要比你长得多的时间。这不仅是你保持理智的一种方式,也是团队其他成员学习的方式。如果你已经领导团队一段时间或者你接手了一个新团队,那么获得团队尊重和了解他们正在做的事情的最简单方法之一就是亲自动手——通常是承担别人不愿意做的繁重任务。你可以有一份简历和一长串的成就列表,但没有什么比投入并真正做一些艰苦的工作更能让你的团队知道你有多熟练、多专注(和多谦逊)。

寻找替代自己的人除非你

想在职业生涯的余下时间里一直做同样的工作,否则就寻找替代自己的人。正如我们前面提到的,这始于招聘过程:如果你想让团队成员替代你,你需要聘请能够替代你的人,我们通常总结为“聘请比你更聪明的人”。在拥有能够完成你工作的团队成员后,你需要给他们机会承担更多的责任或偶尔领导团队。如果你这样做,你很快就会发现谁最有能力

领导团队以及谁想领导团队。请记住,有些人更喜欢成为高绩效的个人贡献者,这没关系。我们一直对那些将他们最好的工程师带入管理职位的公司感到惊讶 违背他们的意愿。这通常会从您的团队中减去一位优秀的工程师,并增加一位水平不佳的经理。

知道何时制造波澜

你会 (不可避免地且经常地)遇到困难的情况,你身体里的每一个细胞都在尖叫着让你什么也不做。这可能是你团队中技术水平不够的工程师。可能是每次遇到问题都会抢先一步。可能是每周工作 30 小时的缺乏动力的员工。“只要等一会儿,情况就会好起来的”,你会告诉自己。“一切都会好起来的”,你会这样合理化。不要落入这个陷阱 这些情况需要你现在就做出最大的努力。这些问题很少会自行解决,你等待解决的时间越长,它们对团队其他成员的负面影响就越大,你就越会因为这些问题彻夜难眠。等待只会拖延不可避免的事情,并在此过程中造成无法估量的损失。所以,行动起来,迅速行动。

保护你的团队免受混乱

当你担任领导角色时,你通常会发现的第一件事是,你的团队之外是一个混乱和不确定 (甚至是疯狂)的世界,当你作为个人贡献者时,你从未见过这样的世界。当我在 1990 年代第一次成为一名经理时 (在重新成为个人贡献者之前),我对公司中发生的大量不确定性和组织混乱感到震惊。我问另一位经理,是什么导致了原本平静的公司突然出现这种动荡,另一位经理歇斯底里地嘲笑我的天真:混乱一直存在,但我的前任经理保护了我和团队的其他成员免受其害。

为您的团队提供空中掩护

尽管让您的团队了解公司 “上层”发生的事情很重要,但保护他们免受团队外部可能强加给您的大量不确定性和无聊要求的影响也同样重要。尽可能多地与您的团队分享信息,但不要让组织疯狂的事情分散他们的注意力,因为这些疯狂的事情极不可能真正影响到他们。

当你的团队表现良好时,让他们知道

许多新团队领导可能过于专注于处理团队成员的缺点,而忽视了经常提供积极的反馈。

就像你让别人知道他们犯了什么错误一样,你也一定要让他们知道

当他们表现出色时,一定要让他们 (和团队的其他成员) 知道。

最后,当最优秀的领导者拥有敢于尝试新事物的冒险精神的团队成员时,他们会经常了解并运用以下技巧:

对容易撤销的事情说“是”很容易。如果您的团队成员想

花一两天时间尝试使用可以加速产品开发的新工具或库⁹ (并且您的期限并不紧迫),您很容易说“当然,试一试”。另一方面,如果他们想做一些事情,比如推出一款您必须在未来 10 年内支持的产品,您可能需要多考虑一下。真正优秀的领导者能够很好地判断什么时候可以撤销某件事,但撤销的事情比您想象的要多 (这适用于技术和非技术决策)。

人就像植物

我妻子是六个孩子中最小的一个,她的母亲面临着一项艰巨的任务,那就是如何养育六个截然不同的孩子,他们每个人都有不同的需求。我问我婆婆她是如何做到这一点的 (明白我的意思了吗?),她回答说,孩子们就像植物:有些像仙人掌,需要少量的水但需要大量的阳光;有些像非洲紫罗兰,需要散射光和湿润的土壤;还有一些像西红柿,如果你给它们一点肥料,它们就会长得非常好。如果你有六个孩子,给每个孩子相同量的水、光照和肥料,他们都会得到同等的待遇,但很可能他们都不会得到他们真正需要的东西。

因此,您的团队成员也像植物一样:有些需要更多的光照,有些需要更多的水 (有些需要更多的……肥料)。作为他们的领导者,您的工作就是确定谁需要什么,然后满足他们的要求 只不过,您的团队需要的不是光照、水和肥料,而是不同程度的激励和指导。

为了让所有团队成员都得到他们需要的东西,你需要激励那些陷入困境的人,并为那些心不在焉或不确定该做什么的人提供更有力的方向。当然,有些人“迷失了方向”,既需要激励也需要方向。所以,有了这种激励和方向的结合,你可以让你的团队快乐而高效。你不想给他们太多 因为如果他们不需要激励或方向,而你却试图给他们,你只会惹恼他们。

⁹为了更好地理解技术变革到底有多么“不可撤销”,请参阅第 22 章。

给出指导相当简单 它需要对需要做的事情有基本的了解、一些简单的组织技能,以及足够的协调能力来将其分解为可管理的任务。有了这些工具,你就可以为需要指导帮助的工程师提供足够的指导。然而,动机则有点复杂,值得解释一下。

内在动机与外在动机

动机有两种类型:外在动机,源自外部力量(如金钱补偿)和内在动机,来自内部。丹·平克(Dan Pink)在其著作《驱动力》(Drive)中解释说,让人们最快乐、最有效率的方法不是从外部激励他们(例如,给他们一大堆现金);而是需要努力提高他们的内在动机。丹声称,你可以通过给予人们三样东西来提高内在动机:自主权、掌控力和目标。¹⁰当一个人有能力独立行动而不需要别人微观管理时,他就拥有了自主权。¹¹对于自主的员工(谷歌努力雇佣大多数自主的工程师),你可能会给他们一个产品发展的大方向,但让他们自己决定如何实现目标。这有助于提高激励,不仅是因为他们与产品的关系更密切(而且可能比你更了解如何制造产品),

还因为这让他们对产品有更强的主人翁意识。他们对产品成功的期望越大,他们希望看到产品成功的兴趣就越大。

精通的最基本形式只是意味着你需要给某人机会来提高现有技能并学习新技能。提供足够的机会来精通不仅有助于激励人们,而且随着时间的推移,他们也会变得更好,从而组建更强大的团队。¹²员工的技能就像刀刃:你可以花费数万美元为你的团队找到技能最精湛的人,但如果你多年不磨刀,你最终会得到一把钝刀,效率低下,在某些情况下甚至毫无用处。谷歌为工程师提供了充足的机会来学习新事物和掌握他们的技艺,以保持他们的敏锐、高效和有效。

¹⁰观看Dan 的精彩 TED 演讲关于这个问题。

¹¹这意味着这些人的薪酬足够高,收入不会成为压力的来源。

¹²这假设你的团队中有人不需要微观管理。

¹³当然,这也意味着他们是更有价值、更有市场价值的员工,所以如果他们不喜欢自己的工作,他们就更容易离开你。请参阅第 99 页“追踪幸福感”中的模式。

当然,如果一个人毫无理由地工作,世界上所有的自主权和掌控力都无法激励他,这就是为什么你需要给他们的工作以目的。许多人致力于开发具有重大意义的产品,但他们却无法了解他们的产品可能对公司、客户甚至世界产生的积极影响。即使在产品的影响可能小得多的情况下,你也可以通过寻找他们努力的原因并向他们说明这个原因来激励你的团队。如果你能帮助他们在工作中看到这个目的,你会看到他们的积极性和生产力大大提高。¹⁴我们认识的一位经理密切关注 Google 收到的有关其产品(“影响较小”的产品之一)的电子邮件反馈,每当她看到客户发来的消息,谈论公司产品如何帮助他们个人或帮助他们的业务时,她都会立即将其转发给工程团队。这不仅能激励团队,而且还经常启发团队成员思考如何使他们的产品变得更好。

结论

领导团队与担任软件工程师是不同的任务。因此,优秀的软件工程师不一定能成为优秀的管理者,这没关系。高效的组织为个人贡献者和人事管理者提供了富有成效的职业道路。尽管谷歌发现软件工程经验本身对管理者来说是无价的,但高效的管理者所能带来的最重要的技能是社交技能。优秀的管理者通过帮助他们出色地工作、让他们专注于正确的目标以及使他们免受团队外部问题的困扰来激励他们的工程团队,同时遵循谦逊、信任和尊重的三大支柱。

TL;DR

- 不要以传统意义上的“管理”;专注于领导力、影响力和服务组建团队。· 尽可能委派任务;不要自己动手。· 特别注意团队的重点、方向和速度。

¹⁴ Adam M. Grant, “任务重要性的意义:工作绩效影响、关系机制和边界条件”,《应用心理学杂志》,第 93 卷,第 1 期 (2018 年), http://bit.ly/task_significance。

第六章

规模领先

作者:Ben Collins-Sussman
编辑:Riona MacNamara

在第5章中,我们讨论了从“个人贡献者”转变为团队的明确领导者意味着什么。从领导一个团队到领导一组相关团队是一个自然的过程,本章讨论了如何在工程领导的道路上继续有效前进。

随着你角色的演变,所有最佳实践仍然适用。你仍然是一名“仆人式领导者”,只是服务于一个更大的群体。也就是说,你正在解决的问题范围变得更大、更抽象。你逐渐被迫成为“更高层次的人”。

也就是说,你越来越无法深入了解事物的技术或工程细节,你被迫“广泛”而不是“深入”。这个过程每一步都令人沮丧:你为失去这些细节而感到悲伤,你逐渐意识到你之前的工程专业知识与你的工作越来越不相关。

相反,您的效率比以往任何时候都更取决于您的一般技术直觉和激励工程师朝着好的方向前进的能力。

这个过程往往令人沮丧 直到有一天你注意到,你作为领导者的影响力实际上比你作为个人贡献者时要大得多。这是一个令人满意但苦乐参半的认识。

那么,假设我们理解了领导力的基本知识,那么如何才能成为一名真正优秀的领导者呢?这就是我们在这里讨论的内容,使用我们所谓的“领导力的三个始终”:始终在决策、始终在离开、始终在扩展。

始终保持决策

管理一个团队意味着要在更高的层次上做出更多的决策。

你的工作更多的是制定高层战略,而不是解决任何具体的工程任务。在这个层面上,你所做的大多数决定都是为了找到正确的权衡。

飞机的寓言Lindsay Jones是我们的朋友,他是一

名专业的戏剧声音设计师和作曲家。他一生都在美国各地飞行,从一个制作公司跳到另一个制作公司,他有很多关于航空旅行的疯狂(和真实)故事。这是我们最喜欢的故事之一:

现在是早上 6 点,我们都登上了飞机,准备出发。机长通过广播系统向我们解释说,不知何故,有人给油箱加了 10,000 加仑的油。现在,我已经坐飞机很长时间了,我不知道这种事是可能的。我的意思是,如果我给汽车加了 1 加仑的油,我的鞋子上就会沾满汽油,对吧?

好吧,不管怎样,机长说我们有两个选择:我们要么等卡车来把飞机上的燃料吸出来,这将需要一个多小时,要么现在就让二十个人下飞机来平衡重量。

没人动。

现在,在头等舱里,坐在我对面过道上的这个家伙,他非常愤怒。

他让我想起了《陆军野战医院》中的弗兰克·伯恩斯;他非常慷慨,到处发牢骚,要求知道谁应该对此负责。这是一次精彩的展示,就像他是马克斯兄弟电影中的玛格丽特·杜蒙特一样。

于是,他抓起钱包,掏出一大叠现金!他说:“我不能迟到这个会议!我会给现在下飞机的任何人 40 美元!”

果然,人们接受了他的建议。他给 20 个人发了 40 美元(顺便说一下,是 800 美元现金!)然后他们都离开了。

现在,一切准备就绪,我们准备出发前往跑道,机长又通过广播通知我们。飞机的计算机停止工作了。没人知道原因。现在我们必须被拖回登机口。

弗兰克·伯恩斯气疯了。说真的,我以为他要中风了。他又骂又叫。其他人都面面相觑。

我们回到登机口,这个家伙要求换一班航班。他们建议给他订 9:30 的航班,但太晚了。他说:“难道 9:30 之前没有其他航班吗?”

登机口工作人员说:“嗯,8 点还有一班航班,但现在都满员了。他们现在正在关闭登机门。”

然后他说:“满了? ! 哈意思? 飞机上一个座位都没有了? ! ? ! ”

登机口工作人员说：“先生，飞机的座位一直敞开着，直到突然冒出 20 名乘客，把所有座位都占满了。他们是我见过的最开心的乘客，他们一路笑着走下登机桥。”

9:30 的航班非常安静。

这个故事当然是关于权衡的。尽管本书的大部分内容都集中在工程系统中的各种技术权衡上，但事实证明，权衡也适用于人类行为。作为领导者，您需要决定您的团队每周应该做什么。有时权衡是显而易见的（“如果我们从事这个项目，就会推迟另一个项目……”）；有时权衡会产生不可预见的后果，可能会反过来伤害你，就像前面的故事一样。

从最高层次来看，作为领导者（无论是单个团队还是大型组织的领导者），你的工作是引导人们解决困难、模糊的问题。模糊是指问题没有明显的解决方案，甚至可能无法解决。

无论哪种方式，都需要探索、导航问题，并（希望）将其控制在可控状态。如果编写代码类似于砍树，那么作为领导者，您的工作就是“透过树木看森林”，找到一条可行的穿过森林的道路，引导工程师走向重要的树木。这个过程有三个主要步骤。首先，您需要识别盲点；接下来，您需要识别权衡；然后您需要决定并迭代解决方案。

识别盲点当你第一次着手解

决一个问题时，你经常会发现一群人已经为此奋斗多年。这些人已经沉浸在这个问题中太久了，以至于他们戴着“盲点”。也就是说，他们不再能看到森林。他们在不知不觉中对问题（或解决方案）做出了一堆假设。“我们一直都是这样做的，”他们会说，他们已经失去了批判性地考虑现状的能力。有时，你会发现奇怪的应对机制或合理化已经发展成为维持现状辩护。这就是你用新眼光拥有巨大优势的地方。你可以看到这些盲点，提出问题，然后考虑新的策略。（当然，不熟悉问题并不是优秀领导力的必要条件，但它往往是一种优势。）

确定关键权衡 顾名思义，重要且模糊

的问题没有“灵丹妙药”解决方案。没有一种答案可以永远适用于所有情况。只有当下最好的答案，而且几乎肯定会涉及在一个方向或另一个方向上做出权衡。你的工作是指出权衡，向每个人解释，然后帮助决定如何平衡它们。

决定,然后迭代

在您了解权衡利弊及其运作方式后,您就有能力了。您可以使用这些信息为这个月做出最佳决策。下个月,您可能需要重新评估和重新平衡权衡利弊;这是一个反复的过程。这就是我们所说的“始终做出决定”的意思。

这里存在风险。如果你不将流程设计为不断重新平衡权衡,你的团队很可能会陷入寻找完美解决方案的陷阱,这可能会导致一些人所说的“分析瘫痪”。你需要让你的团队适应迭代。一种方法是降低风险,通过解释来平息紧张情绪:“我们将尝试这个决定,看看结果如何。”

下个月,我们可以撤销更改或做出不同的决定。”这使人们能够保持灵活性并从他们的选择中学习。

案例研究:解决网络搜索的“延迟”问题在管理团队时,自然会倾向于放弃单一产品,而是拥有整个“类别”的产品,或者可能是跨产品的更广泛问题。谷歌的一个很好的例子与我们最古老的产品网络搜索有关。

多年来,数千名 Google 工程师一直致力于解决改善搜索结果这一普遍问题,即提高结果页面的“质量”。但事实证明,这种对质量的追求有一个副作用:它逐渐使产品变慢。

曾几何时,Google 的搜索结果不过是一页包含 10 个蓝色链接的页面,每个链接代表一个相关网站。然而,在过去十年中,为了提高“质量”,Google 进行了数千次微小的更改,从而带来了越来越丰富的结果:图像、视频、带有 Wikipedia 事实的框,甚至交互式 UI 元素。这意味着服务器需要做更多的工作来生成信息:通过网络发送的字节更多;客户端(通常是手机)被要求呈现越来越复杂的 HTML 和数据。尽管十年来网络和计算机的速度显著提高,但搜索页面的速度却越来越慢:其延迟增加了。这似乎不是什么大问题,但产品的延迟会直接影响(总体而言)用户的参与度和使用频率。即使渲染时间增加 10 毫秒也很重要。延迟慢慢增加。这不是某个特定工程团队的错,而是代表了对公共资源的长期集体毒害。在某种程度上,Web 搜索的总体延迟会增加,直到其效果开始抵消因结果“质量”的提高而带来的用户参与度的提高。

多年来,许多领导人一直在努力解决这个问题,但未能系统地解决这个问题。每个人都带着眼罩,认为唯一的办法是

处理延迟的一种方法就是每两三年宣布一次延迟“黄色代码”¹ ,在此期间每个人都会放下手中的一切来优化代码,加快产品速度。

虽然这种策略暂时有效,但仅仅一两个月后,延迟就会再次开始上升,并很快回到之前的水平。

那么,到底是什么发生了变化呢?在某个时候,我们退后一步,找出了盲点,并重新全面评估了权衡利弊。结果发现,追求“质量”的成本不是一种,而是两种。第一种成本是用户的成本:更高的质量通常意味着发送更多的数据,这意味着更长的延迟。第二种成本是 Google 的成本:更高的质量意味着需要做更多的工作来生成数据,这会占用我们服务器的更多 CPU 时间。我们称之为“服务容量”。尽管领导层经常在质量和容量之间的权衡中谨慎行事,但他们从未将延迟视为微积分中的重要因素。正如那句老笑话所说,“好、快、便宜 选两个”。描述权衡利弊的一个简单方法是在好(质量)、快(延迟)和便宜(容量)之间画一个张力三角形,如图6-1 所示。

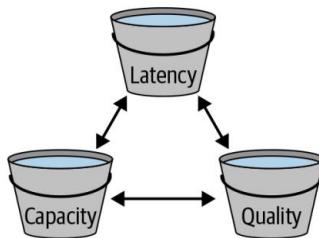


图 6-1. Web 搜索中的 Trade-os;选择两个!

这正是这里发生的情况。通过故意损害另外两个特征中的至少一个,可以轻松改善其中任何一个特征。例如,您可以通过在搜索结果页面上放入更多数据来提高质量。但这样做会损害容量和延迟。您还可以通过更改服务集群上的流量负载在延迟和容量之间进行直接权衡。如果向集群发送更多查询,则可以提高容量,因为您可以更好地利用 CPU 硬件投入更多资金。但更高的负载会增加计算机内的资源争用,使查询的平均延迟更糟。如果您故意减少集群的流量(使其“运行得更冷”),您的整体服务容量会减少,但每个查询都会变得更快。

这里的重点是,这种洞察力 更好地理解所有的权衡 使我们能够开始尝试新的平衡方法。现在,我们可以将延迟视为第一个问题,而不是将其视为不可避免的意外副作用。

¹ “黄色代码”是 Google 的术语,意为“为解决关键问题而举行的紧急黑客马拉松”。受影响的团队预计暂停所有工作,并将 100% 的注意力集中在问题上,直到宣布紧急状态结束。

我们的目标是将我们的目标与其他目标结合起来。这为我们带来了新的策略。例如，我们的数据科学家能够准确衡量延迟对用户参与度的影响。这使得他们能够构建一个指标，将质量驱动的短期用户参与度改进与延迟驱动的长期用户参与度损害进行对比。这种方法使我们能够对产品变更做出更多数据驱动的决策。例如，如果一个小的改变提高了质量但也损害了延迟，我们可以定量地决定这个改变是否值得推出。

我们始终在确定我们的质量、延迟和容量变化是否平衡，并且每个月都会重复我们的决定。

总是要离开

从表面上看，“永远离开”听起来是个糟糕的建议。为什么一个好的领导者会试图离开？事实上，这是前谷歌工程总监 Bharat Mediratta 的一句名言。他的意思是，你的工作不仅仅是解决一个模糊的问题，而是让你的组织自己解决它，而不需要你在场。如果你能做到这一点，你就可以自由地转向新的问题（或新的组织），在你身后留下一串自给自足的成功之路。

当然，这里的反模式是您将自己设置为单点故障 (SPOF) 的情况。正如我们在本书前面提到的，Google 员工对此有一个术语，即巴士因素：您的项目完全失败之前需要被巴士撞到的人数。

当然，这里的“巴士”只是一个比喻。人们生病了；他们换了团队或公司；他们搬走了。作为试金石，想想你的团队正在努力解决的一个难题。现在想象一下，你，作为领导者，消失了。

你的团队是否继续努力？是否继续取得成功？这里有一个更简单的测试：回想一下你上次休假的时间，至少持续一周。你是否一直在查看工作邮件？（大多数领导者都会这样做。）问问自己为什么。如果你不注意，事情会崩溃吗？如果是这样，你很可能让自己成为了 SPOF。你需要解决这个问题。

您的使命：打造一支“自我驱动”团队回到 Bharat 的名言：成为一名

成功的领导者意味着建立一个能够自行解决难题的组织。该组织需要拥有一批强大的领导者、健康的工程流程以及一种持续存在的积极、自我延续的文化。是的，这很难；但它回到了这样一个事实：领导一个团队往往更多的是组织人员，而不是成为一名技术奇才。同样，构建这种自给自足的团队有三个主要部分：划分问题空间、委派子问题和根据需要进行迭代。

划分问题空间具有挑战性的问题

通常由困难的子问题组成。如果你领导着一个团队，一个明显的选择是让一个团队负责每个子问题。

然而，风险在于子问题会随着时间而变化，而僵化的团队界限无法注意到或适应这一事实。如果可以，请考虑采用更宽松的组织结构 子团队可以改变规模，个人可以在子团队之间迁移，分配给子团队的问题可以随着时间而变化。这需要在“过于僵化”和“过于模糊”之间找到一条微妙的界线。一方面，您希望您的子团队对问题、目标和稳定的成就有清晰的认识；另一方面，人们需要自由地改变方向并尝试新事物以应对不断变化的环境。

示例：细分 Google 搜索的“延迟问题”在解决搜索延迟问题时，

我们意识到该问题至少可以细分为两个一般领域：解决延迟症状的工作和解决延迟原因的不同工作。很明显，我们需要招募许多项目人员来优化代码库以提高速度，但只关注速度是不够的。还有成千上万的工程师在增加搜索结果的复杂性和“质量”，速度改进一经实现就立即被抵消，因此我们还需要人员专注于并行问题领域，即首先防止延迟。我们发现我们的指标、延迟分析工具以及开发人员教育和文档中存在差距。通过指派不同的团队同时处理延迟原因和症状，我们能够系统地长期控制延迟。（另外，请注意这些团队负责解决问题，而不是特定的解决方案！）

将子问题委托给领导者管理书籍中

谈论“授权”基本上是陈词滥调，但这是有原因的：授权真的很难学。它违背了我们追求效率和成就的本能。这种困难就是“如果你想把事情做好，就自己动手”这句格言的由来。

话虽如此，如果你认同自己的使命是打造一个自动驾驶的组织，那么主要的教学机制就是通过授权。你必须打造一批能够自给自足的领导者，而授权绝对是培养他们的最有效方式。

你给他们布置任务，让他们失败，然后不断尝试。硅谷有句著名的口头禅，叫“快速失败，不断迭代”。这种哲学不仅适用于工程设计，也适用于人类学习。

作为领导者，你的工作量总是被需要完成的重要任务填满。这些任务大部分都是你相当容易完成的事情。假设你正在努力处理收件箱，回复问题，然后你

决定花 20 分钟来解决一个长期存在的棘手问题。但在执行任务之前,请保持警惕并停下来。问这个关键问题:我真的是唯一一个可以完成这项工作的人吗?

当然,这样做可能最有效,但你却未能培训你的领导者。你没有建立一个自给自足的组织。除非任务确实紧迫且紧迫,否则就咬紧牙关,把工作分配给其他人 可能是你知道可以做但可能需要更长时间才能完成的人。

如果需要,可以指导他们完成工作。你需要为领导者创造成长的机会;他们需要学会“升级”并自己完成这项工作,这样你就不会再处于关键位置。

这里的推论是,作为领导者中的领导者,你需要牢记自己的目标。如果你发现自己深陷困境,那么你就是在损害你的组织。每天上班时,问自己一个不同的关键问题:我能做什么,而我的团队中其他人都做不到?

有很多好的答案。例如,你可以保护你的团队免受组织政治的影响;你可以鼓励他们;你可以确保每个人都善待彼此,创造一种谦逊、信任和尊重的文化。“向上管理”也很重要,确保你的管理链了解你的团队在做什么,并与整个公司保持联系。但通常,这个问题最常见、最重要的答案是:“我能透过树木看到森林。”换句话说,你可以定义一个高级策略。你的策略不仅需要涵盖总体技术方向,还需要涵盖组织策略。你正在为如何解决模糊问题以及你的组织如何随着时间的推移管理问题构建蓝图。你不断地绘制森林地图,然后将砍树的任务分配给其他人。

调整和迭代假设您现

在已经达到了构建自给自足机器的阶段。您不再是 SPOF。恭喜!您现在做什么?

在回答之前,请注意,你实际上已经解放了自己 你现在有“随时离开”的自由。这可能是解决新问题的自由,或者你甚至可以调动自己去一个全新的部门和问题领域,为你培训过的领导者的职业生涯腾出空间。这是避免个人倦怠的好方法。

“现在怎么办？”的简单答案是控制这台机器并保持其健康。但除非有危机，否则你应该采取温和的方式。《调试团队2》一书中有一个关于进行有意识调整的寓言：

有个故事讲的是一位机械方面的大师，他早已退休。他以前的公司遇到了一个没人能解决的问题，所以他们请大师来帮忙找出问题所在。大师检查了机器，听了听机器的声音，最后拿出一根磨损的粉笔，在机器侧面画了一个小 X。他告诉技术人员，那里有一根松动的电线需要修理。技术人员打开机器，拧紧了松动的电线，问题就解决了。当大师收到 10,000 美元的发票时，愤怒的首席执行官回信要求详细说明为什么一个简单的粉笔标记要收取如此高昂的费用！大师回复了另一张发票，显示用粉笔做标记的费用为 1 美元，而知道在哪里画标记则要花费 9,999 美元。

对我们来说，这是一个关于智慧的故事：一个经过深思熟虑的调整可以产生巨大的影响。我们在管理人员时使用这种方法。我们想象我们的团队就像乘坐一艘巨大的飞艇，缓慢而坚定地朝着某个方向飞行。

我们不是进行微观管理，也不是试图不断修正航向，而是花了一周的大部分时间仔细观察和倾听。周末，我们会在飞艇上的一个精确位置用粉笔做个小标记，然后轻轻“敲击”一下，调整航向。

良好的管理就是这样：95% 的观察和倾听，5% 的在恰当的地方做出关键调整。听取领导的意见，不要盲目汇报。

与你的客户交谈，并记住，通常（特别是如果你的团队构建了工程基础设施），你的“客户”不是世界上的最终用户，而是你的同事。客户的幸福需要像你的报告的幸福一样认真倾听。什么有效，什么无效？这艘自动驾驶飞船是否朝着正确的方向前进？你的方向应该是迭代的，但要深思熟虑且尽量精简，做出纠正路线所需的最小调整。如果你退回到微观管理，你就有可能再次成为 SPOF！“永远离开”是对宏观管理的呼唤。

小心地确定团队身份一个常见的错误是

让团队负责某个具体产品，而不是一般问题。产品是问题的解决方案。解决方案的预期寿命可能很短，产品可以被更好的解决方案取代。但是，如果选择得当，问题可以永远存在。将团队身份确定为特定解决方案（“我们是管理 Git 存储库的团队”）可能会随着时间的推移导致各种各样的焦虑。如果你的大部分工程师想要切换到新的版本控制系统，该怎么办？团队可能会“坚持下去”，捍卫自己的解决方案，并抵制

² Brian W. Fitzpatrick 和 Ben Collins-Sussman, 《调试团队：通过协作提高生产力》 (波士顿:O'Reilly, 2016 年)。

改变,即使这不是组织的最佳路径。团队会固守自己的观点,因为解决方案已经成为团队身份和自我价值的一部分。如果团队自己承担了问题(例如,“我们是为公司提供版本控制的团队”),那么团队就可以自由地随着时间的推移尝试不同的解决方案。

始终保持扩张

许多领导力书籍都在学习“最大化影响力”的背景下谈论“扩展”发展团队和影响力的策略。除了我们已经提到的内容外,我们不会在这里讨论这些内容。很明显,建立一个拥有强大领导者的自动驾驶组织已经是实现增长和成功的绝佳秘诀。

相反,我们将从防御和个人的角度而不是进攻的角度来讨论团队扩张。作为领导者,你最宝贵的资源是你有限的时间、注意力和精力。如果你积极地扩大团队的责任和权力,而没有学会在这个过程中保护你的个人理智,那么扩张注定会失败。所以我们将讨论如何通过这个过程有效地自我扩张。

成功循环当一个团队解

决一个难题时,就会出现一个标准模式,即一个特定的循环。它看起来是这样的:

分析首

先,你收到问题并开始努力解决它。你要找出盲点,找到所有的权衡,并就如何管理它们达成共识。

挣扎无论

你的团队是否认为一切准备就绪,你都会开始着手工作。你要为失败、重试和迭代做好准备。此时,你的工作主要是放牧猫。鼓励你的领导和实地专家形成意见,然后仔细聆听并制定总体战略,即使你一开始必须“假装”。³

³这时,冒名顶替综合症很容易发作。克服不知道自己在做什么的感觉的一种方法是假装有专家知道该怎么做,他们只是在度假,你暂时代替他们。这是一种很好的方法,可以消除个人风险,允许自己失败和学习。

牵引力 最

终,您的团队开始弄清楚事情。您做出了更明智的决定,并取得了真正的进展。士气提高了。您正在反复权衡利弊,组织开始围绕问题自我驱动。干得好!

报酬

意想不到的事情发生了。你的经理把你拉到一边,祝贺你的成功。你发现你的奖励不仅仅是拍拍肩膀,而是一个需要解决的全新问题。没错:成功的奖励是更多的工作……和更多的责任!通常,这个问题与第一个问题类似或相邻,但同样困难。

所以现在你陷入了困境。你遇到了一个新问题,但(通常)没有更多的人手。你现在需要以某种方式解决这两个问题,这可能意味着原来的问题仍然需要用一半的人手和一半的时间来解决。

你需要另一半员工来处理新工作!我们将这最后一步称为压缩阶段:你要把你一直在做的所有事情压缩到原来的一半大小。

所以,成功的循环其实更像是一个螺旋(见图6-2)。经过数月和数年,你的组织通过解决新问题,然后想办法压缩这些问题,以便能够应对新的、平行的斗争,从而不断扩大规模。如果你足够幸运,你就可以雇佣更多的人。然而,通常情况下,你的招聘速度跟不上规模扩大的速度。谷歌创始人之一拉里·佩奇可能会将这种螺旋称为“令人不安的兴奋”。

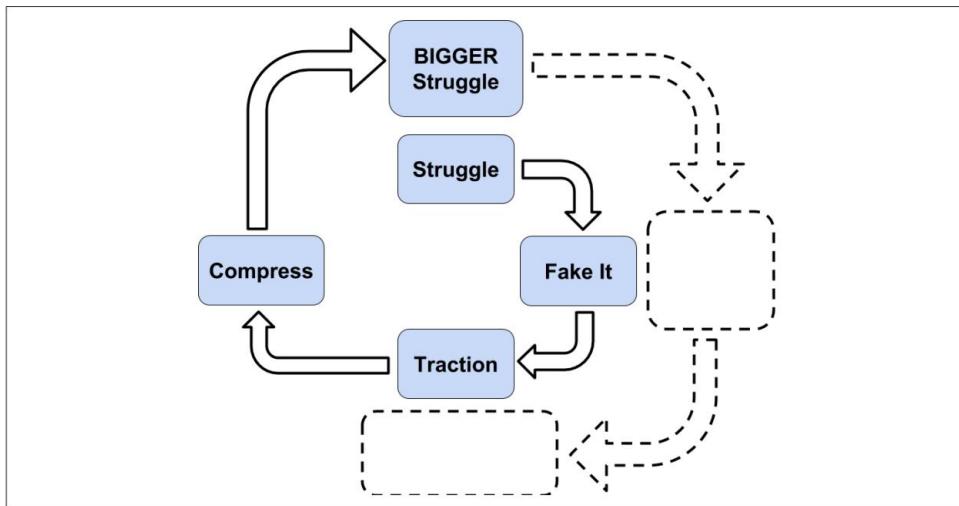


图 6-2 成功螺旋

成功螺旋是一个难题 它很难管理,但它是扩大团队规模的主要范例。压缩问题的行为不仅仅是弄清楚如何最大限度地提高团队的效率,还包括学习如何调整自己的时间和注意力以适应新的责任范围。

重要与紧急回想一下你还不是领导

者,但还是一个无忧无虑的个人贡献者的时候。如果你曾经是一名程序员,你的生活可能更平静,更无所畏惧。你有一长串的工作要做,每天你都会有条不紊地完成清单上的工作,编写代码和调试问题。确定工作优先级、计划和执行很简单。

然而,随着你进入领导层,你可能已经注意到你的主要工作模式变得不那么可预测,更像是救火。也就是说,你的工作变得不那么主动,而更加被动。你在领导层中的职位越高,你收到的升级就越多。你是一长串代码块中的“finally”子句!

您的所有通信方式(电子邮件、聊天室、会议)都开始感觉像是对您的时间和注意力的拒绝服务攻击。事实上,如果您不留心,您最终会将100%的时间花在被动模式上。人们向您扔球,而您疯狂地从一个球跳到另一个球,试图不让任何一个球落地。

很多书都讨论过这个问题。管理学作家史蒂芬·柯维(Stephen Covey)因谈论区分重要的事情和紧急的事情而闻名。事实上,美国总统德怀特·艾森豪威尔(Dwight D. Eisenhower)在1954年的一句名言中推广了这一理念:

我有两种问题,紧急的和重要的。紧急的不重要,重要的永远不紧急。

这种紧张局势是影响您作为领导者效力的最大危险之一。如果您让自己陷入纯粹的被动模式(这几乎是自动发生的),那么您生命中的每一刻都会花在紧急的事情上,但从大局来看,这些事情几乎都不重要。请记住,作为领导者,您的工作是做只有您才能做的事情,例如绘制穿越森林的路径。制定元策略非常重要,但几乎从来都不是紧急的。回复下一封紧急电子邮件总是更容易。

那么,如何才能强迫自己把精力主要放在重要的事情上,而不是紧急的事情上呢?以下是一些关键技巧:

委派任务

很多你看到的紧急任务都可以委派给组织中的其他领导。如果这是一项微不足道的任务,你可能会感到内疚;或者你可能会担心

移交问题效率低,因为其他领导可能需要更长的时间来解决。但这对他们来说是很好的培训,而且可以让你腾出时间去做只有你才能做的重要事情。

安排专门的时间

定期留出两个小时或更长的时间静静地坐下来,只处理重要但不紧急的事情 - 例如团队战略、领导者的
职业道路或计划如何与邻近团队合作。

找到一个有效的跟踪系统 有几十种系统

可以跟踪和确定工作的优先级。有些是基于软件的（例如,特定的“待办事项”工具）,有些是基于笔和
纸的（“Bullet Journal”一些系统是固定的,有些系统是固定的,有些系统与实现无关。在最后一个类
别中,David Allen 的书《搞定一切》在工程经理中非常受欢迎;它是一种抽象算法,用于完成任务并保
持珍贵的“收件箱零”。这里的重点是尝试这些不同的系统并确定哪种适合您。有些会让您满意,有些则
不会,但您绝对需要找到比装饰计算机屏幕的小便签更有效的东西。

学会丢球还有一种管理时间

的关键技巧,表面上听起来很激进。对许多人来说,这与多年的工程直觉相矛盾。作为一名工程师,你注重细节;
你列清单,检查清单上的内容,你一丝不苟,有始有终。这就是为什么在错误跟踪器中关闭错误或将电子邮件减少
到零时感觉如此良好。但作为领导者中的领导者,你的时间和注意力不断受到侵扰。无论你如何努力避免,你
最终都会丢球 有太多球向你扔来。

这让人难以承受,而且你可能一直为此感到内疚。

所以,现在让我们退一步,坦率地看看情况。如果丢掉一些球是不可避免的,那么故意丢掉一些球不是比意外
丢掉更好吗?至少这样你就能有某种控制力。

这有一个很好的方法可以实现这一点。

近藤麻理惠是一位组织顾问,也是非常受欢迎的《怦然心动的人生整理魔法》一书的作者。她的理念是有效地
清理家里的所有垃圾,但这也适用于抽象的杂物。

想象一下你的物质财产被分成三堆。大约 20% 的东西是无用的 你再也不会碰它们,而且很容易扔掉。大约
60% 的东西有些有趣;它们对你的的重要性各不相同,你有时会用它们,有时不会。然后大约 20% 的东西是极其
重要的:这些是你一直使用的东西,

这些物品有着深刻的情感意义,或者用近藤女士的话来说,仅仅是拿着它们就能激发出深深的“喜悦”。她在书中提出的论点是,大多数人对生活的整理方式都是错误的:他们花时间把最底层的 20% 扔进垃圾桶,但剩下的 80% 仍然感觉太杂乱。她认为,整理的真正工作是找出最顶层的 20%,而不是最底层的 20%。如果你只能找出关键的东西,那么你就应该扔掉剩下的 80%。这听起来很极端,但非常有效。如此彻底地整理可以让人感到非常自由。

事实证明,你也可以将这一理念应用到你的收件箱或任务列表上。向你扔来的一堆球。将你的一堆球分成三组:最下面的 20% 可能既不紧急也不重要,很容易被删除或忽略。中间有 60%,可能包含一些紧急或重要的事情,但它是混合包。最上面的 20% 是绝对、至关重要的事情。

所以现在,当你完成你的任务时,不要试图解决最主要的任务。你仍然会不堪重负,并且主要会处理紧急但不重要的任务。

相反,要谨慎地找出那些严格属于前 20% 的球。只有你才能做的关键事情,并严格专注于它们。明确允许自己放弃其他 80%。

一开始这样做可能会感觉很糟糕,但当你故意犯错时,你会发现两件令人惊奇的事情。首先,即使你不委派中间 60% 的任务,你的副领导也经常会注意到并自动接手。其次,如果中间的某些任务确实至关重要,它最终还是会回到你身边,最终上升到前 20%。你只需要相信,低于你前 20% 门槛的事情要么会得到处理,要么会得到适当的发展。同时,因为你只关注至关重要的事情,所以你能够调整时间和注意力,以涵盖团队不断增长的职责。

保护你的精力我们已经讨论

过如何保护你的时间和注意力,但你的个人精力是方程式的另一个部分。所有这些扩展都让人精疲力尽。在这样的环境中,你如何保持精力充沛和乐观?

部分答案是,随着时间的推移,随着年龄的增长,你的整体耐力会增强。

职业生涯初期,每天在办公室工作八小时可能会让你感到震惊;你回家时会感到疲惫和头晕。但就像马拉松训练一样,你的大脑和身体会随着时间的推移积累更多的耐力储备。

答案的另一个关键部分是,领导者逐渐学会更明智地管理自己的精力。这是他们学会不断关注的事情。通常,这意味着要意识到自己在任何特定时刻有多少精力,

并有意识地选择在特定时刻以特定方式“给自己充电”。以下是一些有意识的能量管理的很好的例子：

真正享受假期 周末不是

假期。至少需要三天时间才能“忘记”工作；至少需要一周时间才能真正感到精神焕发。但如果你查看工作邮件或聊天，你就毁了充电的过程。一连串的担忧又涌入你的脑海，心理疏离带来的所有好处都消失了。只有你真正自律地断开连接，假期才能让你恢复精力。⁴当然，这只有当你建立了一个自动驾驶组织时才有可能。

让断开连接变得简单

断开连接时，请将工作笔记本电脑留在办公室。如果您的手机上有工作通讯，请将其删除。例如，如果您的公司使用 G Suite（Gmail、Google 日历等），一个很好的技巧是将这些应用安装在手机的“工作配置文件”中。这会导致第二组带有工作标志的应用出现在您的手机上。例如，您现在有两个 Gmail 应用：一个用于个人电子邮件，一个用于工作电子邮件。在 Android 手机上，您可以按一个按钮一次禁用整个工作配置文件。所有工作应用都会变灰，就像它们已被卸载一样，您无法“意外”查看工作消息，直到您重新启用工作配置文件。

享受真正的周末

周末的效果不如假期，但仍具有一些恢复活力的力量。

再次强调，只有断开工作通信后，此充值才有效。

尝试在周五晚上真正签到，在周末做你喜欢的事情，然后在周一早上回到办公室时再次签到。

白天休息一下 你的大脑以 90 分

钟为一个自然周期运行。⁵利用这个机会站起来在办公室里走走，或者花 10 分钟到外面散步。这样的短暂休息只是短暂的充电，但却可以极大地改善你的压力水平以及你在接下来的两个小时工作中的感受。

允许自己休息一天来保持心理健康 有时，你毫无理由地就过得很快

糟糕。你可能睡得很好，吃得很好，锻炼了，但你的心情仍然很糟糕。如果你是领导者，这是一件很糟糕的事情。你的坏心情会给你周围的每个人定下基调，并可能导致糟糕的决定（你不应该发送的电子邮件、过于严厉的判断等）。如果你发现自己处于这种情况，就转过身来

⁴你需要提前计划，并假设你的工作在假期期间根本无法完成。

在休假之前和之后努力工作（或巧妙工作）可以缓解这个问题。

⁵您可以在https://en.wikipedia.org/wiki/Basic_rest-activity_cycle上阅读有关BRAC的更多信息。

然后回家,请病假。那天什么都不做比主动造成破坏要好。

最后,管理精力与管理时间同样重要。如果你学会了掌握这些事情,你就能准备好应对扩大责任范围和建立自给自足的团队这一更广泛的周期。

结论

成功的领导者自然会在进步过程中承担更多责任(这是一件好事,也是自然而然的事情)。除非他们能有效地想出技巧,以便快速做出正确决策、在需要时授权和管理增加的责任,否则他们最终可能会感到不知所措。成为一名有效的领导者并不意味着你需要做出完美的决定、亲力亲为或加倍努力。相反,要努力始终做出决定、始终离开并始终扩大规模。

TL;DR

- 始终保持决策:模棱两可的问题没有神奇的答案;它们都是找到当下正确的权衡,并进行迭代。
- 随时离开:作为领导者,你的工作是建立一个组织,随着时间的推移,无需你亲自在场,就能自动解决一类模糊问题。
- 不断扩大规模:随着时间的推移,成功会带来更多的责任,您必须主动管理这项工作的规模,以保护您个人的时间、注意力和精力等稀缺资源。

第七章

测量工程生产力

作者:Ciera Jaspen
编辑:Riona Macnamara

Google 是一家数据驱动型公司。我们用硬数据支持大多数产品和设计决策。使用适当指标的数据驱动决策文化存在一些缺点,但总体而言,依靠数据往往会使大多数决策变得客观而非主观,这通常是一件好事。然而,收集和分析人为因素方面的数据有其自身的挑战。具体来说,在软件工程领域,Google 发现,随着公司规模的扩大,拥有一支专注于工程生产力本身的专家团队非常有价值和重要,并且可以利用此类团队的见解。

为什么要衡量工程生产力?

假设你的生意蒸蒸日上（例如,你经营着一家在线搜索引擎）,你想扩大业务范围（进入企业应用市场、云市场或移动市场）。想必,为了扩大业务范围,你还需要扩大工程组织的规模。然而,随着组织规模的线性增长,沟通成本会呈二次方增长。¹增加更多人员是扩大业务范围的必要条件,但沟通开销成本不会随着人员增加而线性增长。因此,你无法将业务范围线性扩展到工程组织的规模。

¹ Frederick P. Brooks,《人月神话:软件工程论文集》(纽约:Addison-Wesley,1995年)。

不过,还有另一种方法可以解决我们的扩展问题:我们可以提高每个人的生产力。如果我们可以提高组织中每个工程师的生产力,我们就可以扩大业务范围,而无需相应增加沟通开销。

Google 必须快速发展新业务,这意味着我们必须学习如何提高工程师的工作效率。为此,我们需要了解是什么让他们的工作效率更高,找出工程流程中的低效之处,并解决已发现的问题。然后,我们会根据需要重复这一循环,形成持续改进的循环。通过这样做,我们将能够随着需求的增加而扩大工程组织的规模。

然而,这一改进周期也需要人力资源。如果每年需要 50 名工程师来了解和修复生产力阻碍因素,那么每年将工程组织生产力提高 10 名工程师的水平就不值得了。因此,我们的目标不仅是提高软件工程生产力,而且要高效地提高生产力。

在 Google,我们通过组建一支致力于了解工程生产力的研究团队来解决这些权衡问题。我们的研究团队包括来自软件工程研究领域的人员和通才软件工程师,但我们还包括来自认知心理学和行为经济学等各个领域的社会科学家。社会科学人员的加入使我们不仅可以研究工程师制作的软件工作,还可以了解软件开发的人性化一面,包括个人动机、激励结构和管理复杂任务的策略。该团队的目标是采用数据驱动的方法来衡量和提高工程生产力。

在本章中,我们将介绍我们的研究团队如何实现这一目标。首先是分类过程:我们可以测量软件开发的许多部分,但我们应该测量什么?选定项目后,我们将介绍研究团队如何确定有意义的指标,以确定过程中存在问题的部分。最后,我们将介绍 Google 如何使用这些指标来跟踪生产率的提高。

在本章中,我们将遵循 Google 的 C++ 和 Java 语言团队提出的一个具体示例:可读性。在 Google 存在的大部分时间里,这些团队一直管理着 Google 的可读性流程。(有关可读性的更多信息,请参阅第 3 章。)可读性流程是在 Google 早期就已实施的,当时自动 format (第 8 章)和阻止提交的 linter 还未普及 (第 9 章)。该流程本身的运行成本很高,因为它需要数百名工程师为其他工程师执行可读性审查,才能授予他们可读性。一些工程师认为这是一个过时的、不再有用的欺辱过程,它是午餐桌上争论的热门话题。具体

语言团队的问题是:花在可读性过程上的时间是否值得?

分类:是否值得衡量?

在决定如何衡量工程师的生产力之前,我们需要知道什么时候一个指标是值得衡量的。衡量本身是昂贵的:它需要人们衡量过程、分析结果,并将其传播给公司的其他部门。此外,衡量过程本身可能很繁重,并拖慢工程组织其他部门的速度。即使它并不慢,跟踪进度也可能会改变工程师的行为,可能以掩盖潜在问题的方式。我们需要明智地衡量和估计;虽然我们不想猜测,但我们不应该浪费时间和资源进行不必要的衡量。

在 Google,我们提出了一系列问题来帮助团队确定是否值得衡量生产力。我们首先要求人们以具体问题的形式描述他们想要衡量的内容;我们发现,人们提出的问题越具体,他们就越有可能从这一过程中获益。当可读性团队找到我们时,他们的问题很简单:让一名工程师经历可读性过程的成本是否值得他们为公司带来的利益?

然后我们要求他们考虑问题的以下方面:

您期望什么结果?为什么?

尽管我们可能想假装自己是中立的调查员,但事实并非如此。我们确实对应该发生的事情有先入为主的观念。通过从一开始就承认这一点,我们可以尝试解决这些偏见并防止事后对结果进行解释。

当这个问题被提交给可读性团队时,他们表示并不确定。人们确信在某个时间点,成本值得收益,但随着自动格式化程序和静态分析工具的出现,没有人完全确定。越来越多的人认为这个过程现在变成了一种欺辱仪式。虽然它可能仍然为工程师带来好处(他们有调查数据显示人们确实声称这些好处),但不清楚它是否值得代码作者或审阅者投入时间。

如果数据支持你预期的结果,会采取什么行动?

我们提出这个问题是因为如果不采取行动,测量就没有意义。

请注意,如果我们没有得到这个结果,那么如果有计划的改变就会发生,那么行动实际上可能是“维持现状”。

当被问及这个问题时,可读性团队的回答很直接:如果好处足以证明该过程的成本是合理的,他们会链接到有关可读性的研究和常见问题解答中的数据,并进行宣传以设定期望。

如果我们得到负面结果,我们会采取适当的行动吗?

我们之所以问这个问题,是因为在很多情况下,我们发现负面结果不会改变决策。决策中可能还有其他因素会推翻任何负面结果。如果是这样的话,那么一开始就不值得衡量。这个问题阻止了我们研究团队开展的大多数项目;我们了解到,决策者有兴趣了解结果,但由于其他原因,他们不会选择改变方向。

然而,在可读性方面,我们得到了团队的有力行动声明。它承诺,如果我们的分析表明成本超过收益或收益微不足道,团队将终止该流程。由于不同的编程语言在格式化程序和静态分析方面的成熟度不同,因此此评估将根据每种语言进行。

谁将决定根据结果采取行动,以及他们何时采取行动?

我们提出这一要求是为了确保请求测量的人是真正有权采取行动的人(或直接代表他们采取行动)。最终,测量我们软件流程的目标是帮助人们做出业务决策。了解这个人是谁,包括什么形式的数据能说服他们,这一点很重要。尽管最好的研究包括多种方法(从结构化访谈到日志的统计分析),但为决策者提供所需数据的时间可能有限。在这种情况下,最好迎合决策者。他们是否倾向于通过从访谈中检索到的故事来产生同理心,从而做出决策?²他们信任调查结果还是日志数据?他们对复杂的统计分析感到满意吗?如果决策者原则上不相信结果的形式,那么测量该流程也没有意义。

在可读性方面,我们对每种编程语言都有明确的决策者。Java 和 C++ 两个语言团队积极向我们寻求帮助,其他团队则在观望这些语言的发展情况

²值得一提的是,我们这个行业目前对“轶事数据”持鄙视态度,每个人都有一个目标

是“数据驱动的”。然而,轶事仍然存在,因为它们很有影响力。轶事可以提供原始数字无法提供的背景和叙述;它可以提供与他人产生共鸣的深刻解释,因为它反映了个人经历。虽然我们的研究人员不会根据轶事做出决定,但我们确实使用并鼓励使用结构化访谈和案例研究等技术来深入了解现象并为定量数据提供背景。

首先。3决策者相信工程师们自我报告的经验能够理解幸福感和学习能力,但决策者希望看到基于日志数据的速度和代码质量的“硬数字”。这意味着我们需要对这些指标进行定性和定量分析。

这项工作没有严格的截止日期,但如果变化,会有一个内部会议,这是一个很好的宣布时间。这个截止日期给了我们几个月的时间来完成这项工作。

通过提出这些问题,我们发现,在很多情况下,测量根本就不值得……这没关系!有很多很好的理由不去衡量工具或流程对生产力的影响。以下是我们看到的一些例子:

您现在无法承担更改流程/工具的费用 可能存在时间限制

或财务限制,导致无法进行更改。例如,您可能认为,只要切换到更快的构建工具,每周就可以节省数小时的时间。但是,切换意味着在每个人都转换时暂停开发,并且即将面临一个重要的资金截止日期,您无法承受中断。工程权衡不是在真空中评估的 在这种情况下,重要的是要意识到更广泛的背景完全可以证明推迟采取行动是合理的。

任何结果很快就会因其他因素而失效。这里的例子可能包

括在计划重组之前测量组织的软件流程。或者测量已弃用的系统的技术债务量。

决策者有强烈的意愿,而你不太可能提供足够多的、正确类型的证据来改变他们的信念。

这归结于了解你的受众。即使在谷歌,我们有时也会发现人们由于过去的经历而对某个主题有坚定不移的信念。我们发现利益相关者从不相信调查数据,因为他们不相信自我报告。我们还发现利益相关者最容易被通过少量访谈获得的令人信服的叙述所左右。当然,也有一些利益相关者只被日志分析所左右。在所有情况下,我们都试图使用混合方法来对真相进行三角测量,但如果利益相关者仅限于相信不适合问题的方法,那么这项工作就没有意义了。

3 Java 和 C++ 拥有更多的工具支持。两者都有成熟的格式化程序和静态分析

发现常见错误的工具。两者都得到了内部的大力资助。尽管其他语言团队(如 Python)对结果感兴趣,但如果无法证明 Java 或 C++ 具有同样的优势,那么 Python 消除可读性显然不会带来任何好处。

结果将仅用作支持你要做的事情的虚荣指标

反正

这也许是我们告诉 Google 人们不要衡量软件流程的最常见原因。许多时候，人们出于多种原因计划做出决策，而改进软件开发流程只是众多好处之一。例如，Google 的发布工具团队曾经要求对发布工作流系统的计划更改进行衡量。由于更改的性质，很明显更改不会比当前状态更糟，但他们不知道这是一个小改进还是一个大改进。我们问团队：如果结果只是一个改进，你是否愿意花费资源来实现该功能，即使它看起来不值得投资？答案是肯定的！该功能确实提高了生产率，但这只是一个副作用：它的性能也更高，并降低了发布工具团队的维护负担。

唯一可用的指标不足以衡量问题，而且可能会受到其他因素的干扰

在某些情况下，所需的指标（请参阅后面关于如何确定指标的部分）根本无法获得。在这些情况下，使用其他不太精确的指标（例如编写的代码行数）进行衡量可能很有吸引力。

但是，这些指标的任何结果都是无法解释的。如果指标证实了利益相关者先前的信念，他们可能会继续执行他们的计划，而不会考虑该指标不是一个准确的衡量标准。如果它不能证实他们的想法，那么指标本身的不精确性就很容易解释，利益相关者可能会再次继续执行他们的计划。

当你成功测量软件流程时，你并不是要证明某个假设是正确的还是错误的；成功意味着向利益相关者提供他们做出决策所需的数据。如果利益相关者不使用这些数据，那么项目就注定会失败。我们只应该在根据结果做出具体决策时测量软件流程。对于可读性团队来说，需要做出一个明确的决定。如果指标显示该流程是有益的，他们就会公布结果。如果不是，该流程将被废除。最重要的是，可读性团队有权做出这个决定。

选择具有目标和信号的有意义的指标

决定测量软件流程后,我们需要确定使用什么指标。显然,代码行数 (LOC) 不能满足要求,⁴但我们如何才能真正测量工程效率呢?

在 Google,我们使用目标/信号/指标 (GSM) 框架来指导指标的创建。

- 目标是期望的最终结果。它以您希望从高层次理解的内容来表述,不应包含对具体衡量方法的引用。
- 信号是让您知道您已实现最终结果的方式。信号是我们想要测量的东西,但它们本身可能无法测量。
- 指标是信号的代理。它是实际上可以测量的东西。它可能不是理想的测量方法,但我们认为它足够接近理想测量方法。

GSM 框架在创建指标时鼓励几个理想的属性。

首先,通过先创建目标,然后创建信号,最后创建指标,可以防止路灯效应。该术语来自短语“在路灯下寻找钥匙”:如果你只看你能看到的地方,那么你可能没有在正确的地方寻找。

对于指标,这种情况发生在我们使用那些我们可以轻松获得且易于衡量的指标时,无论这些指标是否符合我们的需求。相反,GSM 迫使我们思考哪些指标实际上有助于我们实现目标,而不仅仅是现有的指标。

其次,GSM 鼓励我们在实际测量结果之前,使用原则性方法提出适当的指标集,从而有助于防止指标蔓延和指标偏差。考虑一下这种情况,如果我们选择指标时没有使用原则性方法,结果就不符合利益相关者的期望。这时,我们就面临着利益相关者建议我们使用他们认为会产生预期结果的不同指标的风险。因为我们一开始并没有根据原则性方法进行选择,所以没有理由说他们错了!相反,GSM 鼓励我们根据指标衡量原始目标的能力来选择指标。利益相关者可以很容易地看到,这些指标与他们的期望相符。

⁴ “从这里开始,用‘每月编写的代码行数’来衡量‘程序员生产力’仅需迈出一小步。这是一个非常昂贵的测量单位,因为它鼓励编写乏味的代码,但今天我不再关心这个单位有多愚蠢,即使从纯粹的商业角度来看也是如此。我今天的观点是,如果我们想计算代码行数,我们不应该将它们视为‘编写的代码行数’,而应该视为‘使用的代码行数’:当前的传统观点非常愚蠢,以至于将这个计数记在错误的一边。” Edsger Dijkstra,关于真正教授计算机科学的残酷性, [EWD 手稿 1036](#)。

最初的目标，并提前同意这是衡量结果的最佳指标。

最后，GSM 可以向我们展示我们在哪里有测量覆盖，在哪里没有。当我们执行 GSM 流程时，我们会列出所有目标并为每个目标创建信号。正如我们将在示例中看到的，并非所有信号都是可测量的 - 没关系！使用 GSM，至少我们已经确定了哪些是不可测量的。通过识别这些缺失的指标，我们可以评估是否值得创建新的指标，甚至是否值得测量。

重要的是保持可追溯性。对于每个指标，我们应该能够追溯到它所代表的信号以及它试图测量的目标。这确保我们知道我们正在测量哪些指标以及为什么要测量它们。

目标

目标应该以期望的属性来写，而不是参考任何指标。这些目标本身是不可衡量的，但一组好的目标是每个人都能达成共识，然后才能继续讨论信号和指标。

为了实现这一目标，我们首先需要确定要衡量的正确目标。这似乎很简单：团队肯定知道他们工作的目标！然而，我们的研究团队发现，在许多情况下，人们忘记在生产力中考虑所有可能的权衡，这可能导致

测量错误。

以可读性为例，我们假设团队过于专注于使可读性过程变得快速而简单，以至于忘记了代码质量的目标。

团队设置了跟踪测量，以了解完成审核流程所需的时间以及工程师对流程的满意度。我们的一名队友提出了以下建议：

我可以让你的审查速度非常快：完全删除代码审查。

虽然这显然是一个极端的例子，但团队在测量时总是忘记核心的权衡：他们过于专注于提高速度而忘记了测量质量（反之亦然）。为了解决这个问题，我们的研究团队将生产力分为五个核心组成部分。这五个组成部分相互权衡，我们鼓励团队考虑每个组成部分的目标，以确保他们不会在无意中提高一个组成部分的同时降低其他组成部分的水平。

为了帮助人们记住所有五个组成部分，我们使用助记符“QUANTS”：

代码质量 代码质量

如何?测试用例是否足够好,可以防止回归?架构在降低风险和变更方面有多好?

工程师的注意力 工程师

多久会进入心流状态?他们被通知分散了多少注意力?工具是否鼓励工程师切换上下文?

智力复杂性 完成一项

任务需要多少认知负荷? 正在解决的问题的内在复杂性是什么? 工程师是否需要处理不必要的复杂性?

节奏和速度 工程师

能多快完成任务?他们能多快发布产品?他们在给定的时间内完成了多少任务?

满意度 工

程序员对他们的工具有多满意?工具能多好地满足工程师的需求?他们对自己的工作和最终产品有多满意?工程师是否感到精疲力竭?

回到可读性示例,我们的研究团队与可读性团队合作,确定了可读性过程的几个生产力目标:

代码质量通过可读

性过程,工程师可以编写出更高质量的代码;通过可读性过程,他们可以编写出更一致的代码;并且通过可读性过程,他们为代码健康文化做出了贡献。

工程师的注意力 我们没

有为可读性设定任何注意力目标。这没关系!并非所有关于工程生产力的问题都涉及所有五个领域的权衡。

智力复杂性 工程师通过可读性过

程了解 Google 代码库和最佳编码实践,并在可读性过程中接受指导。

节奏和速度 可读性

过程让工程师能够更快、更高效地完成工作任务。

满意度工程

师看到了可读性过程的好处并对参与其中抱有积极的感觉。

信号

信号是让我们知道我们已经实现目标的方式。并非所有信号都是可衡量的，但目前这是可以接受的。信号和目标之间并不是 1:1 的关系。每个目标至少应该有一个信号，但可能会有更多信号。一些目标也可能共享一个信号。[表 7-1](#) 显示了可读性过程测量目标的一些示例信号。

表 7-1. 信号和目标

目标	
可读性过程的结果是工程师可以编写出更高质量的代码。	读性授予的 工程师认为他们的代码质量比未获得可读性的工程师更高。 可读性过程对代码质量有积极的影响。
工程师通过可读性过程了解了 Google 代码库和最佳编码实践。	工程师报告了从可读性过程中学到的东西。
工程师在可读性过程中会接受指导。	工程师们报告称，他们与在可读性过程中担任审阅者的经验丰富的 Google 工程师进行了积极的互动。
可读性流程的实施使得工程师能够更快、更高效地完成工作任务。	被授予可读性的工程师认为自己比未获得可读性的工程师更有效率。 被授予可读性的工程师编写的更改比未获得可读性的工程师编写的更改的审核速度更快。
工程师看到了可读性过程的好处并对参与其中抱有积极的感觉。	工程师认为可读性过程是有价值的。

指标

指标是我们最终决定如何测量信号的地方。指标不是信号本身；它们是信号的可测量代理。因为它们是代理，所以它们可能不是完美的测量。因此，当我们尝试对底层信号进行三角测量时，某些信号可能会有多个指标。

例如，为了衡量工程师的代码在可读性之后是否能更快地被审查，我们可能会结合使用调查数据和日志数据。这两种指标都不能真正提供根本的真相。（人类的感知是会出错的，日志指标可能无法衡量工程师花在审查一段代码上的全部时间，或者可能会受到当时未知的因素（如代码更改的大小或难度）的干扰。）但是，如果这些指标显示不同的结果，则表明其中一个可能是不正确的，我们需要进一步探索。如果它们相同，我们更有信心我们已经找到了某种真相。

此外,某些信号可能没有任何相关指标,因为此时信号可能根本无法测量。例如,考虑测量代码质量。尽管学术文献提出了许多代码质量代理,但没有一个真正捕捉到它。为了便于阅读,我们决定要么使用较差的代理并可能根据它做出决定,要么简单地承认这是目前无法测量的一点。最终,我们决定不将其作为定量指标,尽管我们确实要求工程师自我评估他们的代码质量。

遵循 GSM 框架是明确衡量软件流程的目标以及实际衡量方法的好方法。但是,所选指标仍有可能没有说明全部情况,因为它们没有捕捉到所需的信号。在 Google,我们使用定性数据来验证我们的指标并确保它们捕捉到预期的信号。

使用数据验证指标

举例来说,我们曾经创建了一个指标来衡量每个工程师的平均构建延迟;目标是捕捉工程师构建延迟的“典型体验”。

然后,我们进行了一项经验抽样研究。在这种研究方式中,工程师在执行感兴趣的任務时会被打断,以回答几个問題。在工程师开始构建后,我们会自动向他们发送一份关于他们的经验和对构建延迟的预期的小调查。然而,在少数情况下,工程师回答说他们还没有开始构建!事实证明,自动化工具正在启动构建,但工程师没有被这些结果所阻碍,因此它不“计入”他们的“典型经验”。然后我们调整了指标以排除此类构建。

定量指标很有用,因为它们可以赋予你权力和规模。你可以衡量整个公司工程师在很长一段时间内的经验,并对结果充满信心。但是,它们不提供任何背景或叙述。定量指标无法解释工程师为何选择使用过时的工具来完成任务,为何采用不寻常的工作流程,为何绕过标准流程。只有定性研究才能提供这些信息,也只有定性研究才能提供有关改进流程的后续步骤的见解。

5根据我们在谷歌的经验,当定量和定性指标不一致时,
是因为定量指标没有捕捉到预期的结果。

现在考虑表 7-2 中列出的信号。您可以创建哪些指标来衡量每个指标？其中一些信号可能可以通过分析工具和代码日志来衡量。其他信号只能通过直接询问工程师来衡量。还有一些信号可能无法完全衡量 例如，我们如何真正衡量代码质量？

最终，在评估可读性对生产力的影响时，我们得到了来自三个来源的指标组合。首先，我们进行了一项专门针对可读性流程的调查。这项调查是在人们完成该流程后进行的；这使我们能够立即获得他们对流程的反馈。这有望避免回忆偏差⁶，但它确实引入了近期偏差⁷和抽样偏差⁸。其次，我们使用了一项大规模的季度调查来跟踪与可读性无关的项目；相反，它们纯粹是关于我们预计可读性应该影响的指标。最后，我们使用了来自开发人员工具的细粒度日志指标来确定日志声称工程师完成特定任务需要多少时间。⁹表 7-2 列出了完整的指标列表及其对应的信号和目标。

表 7-2. 目标、信号和指标

量化分析师	目标	公制
代码质量	可读性过程的结果是工程师可以编写出更高质量的代码。	读性的工程师认为他们的代码质量比未获得可读性的工程师更高。
	可读性过程对代码质量有积极的影响。	季度调查：对自己代码质量感到满意的工程师比例 可读性调查：报告可读性评审对代码质量没有影响或产生负面影响的工程师比例

⁶回忆偏差是记忆偏差。人们更有可能回忆起特别有趣或令人沮丧。

⁷近因偏差是记忆偏差的另一种形式，即人们对最近的经历有偏见。在这种情况下，由于他们刚刚成功完成了这个过程，他们可能会对此感觉特别好。

⁸因为我们只询问了那些完成了该过程的人，所以我们没有收集那些没有完成该过程的人的意见。

⁹人们很容易用这些指标来评估个别工程师，甚至可能用来识别表现好的和表现差的工程师。但这样做会适得其反。如果将生产力指标用于绩效评估，工程师很快就会玩弄这些指标，这些指标将不再有助于衡量和提高整个组织的生产力。让这些测量发挥作用的唯一方法是放弃衡量个人的想法，转而接受衡量总体效果。

量化分析师	目标	信号	公制
			可读性调查:报告称参与可读性过程提高了团队代码质量的工程师比例
	可读性过程的结果是工程师可以编写更加一致的代码。	作为可读性过程的一部分,可读性审阅者会在代码审查中为工程师提供一致的反馈和指导。	可读性调查:报告可读性审阅者评论不一致的工程师比例和
	工程师通过可读性促进了代码健康的文化过程。	获得可读性的工程师会定期在代码审查中对风格和/或可读性问题进行评论。	可读性标准。
工程师的关注 n/a	无	无	无
知识分子	工程师通过可读性过程了解了 Google 代码库和最佳编码实践。	工程师报告了从可读性过程中学到的东西。	可读性调查:报告称了解四个相关主题的工程师比例
			可读性调查:报告称学习或获得专业知识是可读性流程的优势的工程师比例
	工程师在可读性过程中会接受指导。	工程师们报告称,他们与在可读性过程中担任审阅者的经验丰富的 Google 工程师进行了积极的互动。	可读性调查:报告称与可读性审阅者合作是可读性流程优势的工程师比例
节奏/速度	可读性流程的实施使得工程师的工作效率更高。	被授予可读性的工程师认为自己比未获得可读性的工程师更有效率。	季度调查:表示自己工作效率高的工程师比例
			可读性调查:有比例的工程师表示,缺乏可读性会降低团队的工程速度
	工程师们报告说,完成可读性过程对他们的工程速度产生了积极的影响。		
	获得可读性的工程师编写的变更列表 (CL) 比未获得可读性的工程师编写的 CL 审查速度更快。		日志数据:具有可读性和不具有可读性的作者的 CL 的平均审查时间

量化分析师	目标	信号	公制
		获得可读性的工程师编写的 CL 比由其他工程师编写的 CL 更容易通过代码审查。未被授予可读性的工程师。	日志数据:具有以下特征的作者的 CL 的平均引导时间 可读性和不可读性
		获得可读性的工程师编写的 CL 比未获得可读性的工程师编写的 CL 更快通过代码审查。	日志数据:作者提交 CL 的平均时间 具有可读性与不具有可读性
		可读性过程不会对工程速度产生负面影响。	可读性调查:报告称可读性流程对其速度产生负面影响的工程师比例
			可读性调查:报告可读性审阅者及时做出回应的工程师比例
			可读性调查:报告称评审及时性是
			可读性过程的强度
满意	工程师看到了可读性过程的好处并对参与其中抱有积极的感觉。	工程师认为可读性过程总体来说是一种积极的体验。	可读性调查:报告称其对可读性流程的体验总体上是积极的工程师比例
		工程师认为可读性过程是值得的	可读性调查:报告可读性过程有价值的工程师比例
			可读性调查:报告称可读性评审质量是流程优势的工程师比例
			可读性调查:报告称全面性是流程优势的工程师比例
		工程师并不认为可读性过程令人沮丧。	可读性调查:报告可读性过程不确定、不清楚、缓慢或令人沮丧的工程师比例

量化分析师

目标

信号

公制

季度调查:表示对自己的工程速度感到满意的工程师比例

采取行动并追踪结果

回想一下本章的最初目标:我们想要采取行动,提高生产力。在对某个主题进行研究后,Google 团队总会准备一份建议清单,说明如何继续改进。我们可能会建议为某个工具添加新功能、改善某个工具的延迟、改进文档、删除过时的流程,甚至改变工程师的激励结构。理想情况下,这些建议是“工具驱动的”:如果工具不支持工程师这样做,那么告诉他们改变流程或思维方式是没有用的。

相反,我们总是假设,如果工程师拥有适当的数据和合适的工具,他们就会做出适当的权衡。

对于可读性,我们的研究表明,它总体上是值得的:实现了可读性的工程师对流程感到满意,并觉得他们从中学到了很多东西。我们的日志显示,他们的代码审查速度也更快,提交速度也更快,甚至不需要那么多审阅者。我们的研究还发现了流程有待改进的地方:工程师确定了可以使流程更快或更愉快的痛点。语言团队采纳了这些建议,改进了工具和流程,使其更快、更透明,以便工程师获得更愉快的体验。

结论

在 Google,我们发现,组建一支工程生产力专家团队对软件工程具有广泛的好处;一个集中式团队可以专注于复杂问题的广泛解决方案,而不是依靠每个团队制定自己的路线来提高生产力。众所周知,这种“以人为本”的因素很难衡量,专家必须了解正在分析的数据,因为改变工程流程所涉及的许多权衡很难准确衡量,而且往往会产生意想不到的后果。这样的团队必须保持数据驱动,并致力于消除主观偏见。

TL;DR

- 在衡量生产力之前,无论结果是正面的还是负面的,都要问一问结果是否可行。如果你无法对结果采取任何行动,那么它可能不值得衡量。

- 使用 GSM 框架选择有意义的指标。好的指标是您要测量的信号的合理代理，并且可以追溯到您的原始目标。
- 选择涵盖生产力所有部分的指标（QUANTS）。通过这样做，您可以确保不会以牺牲生产力某一方面（如开发人员速度）为代价来提高生产力的另一方面（如代码质量）。
- 定性指标也是指标！考虑建立一种调查机制，用于跟踪有关工程师信念的纵向指标。定性指标也应该与定量指标保持一致；如果不一致，则可能是定量指标不正确。
- 旨在创建融入开发人员工作流程和激励结构的建议。尽管有时需要推荐额外的培训或文档，但如果融入开发人员的日常习惯中，则更有可能发生变化。

第三部分

流程

第八章

风格指南和规则

作者:Shaindel Schwartz
编辑:Tom Mansreck

大多数工程组织都有管理其代码库的规则。关于源文件存储位置的规则、关于代码格式的规则、关于命名和模式以及异常和线程的规则。大多数软件工程师都在控制其操作方式的一组政策范围内工作。在 Google,为了管理我们的代码库,我们维护了一组定义规则的样式指南。

规则就是法律。它们不仅仅是建议或推荐,而是严格的强制性法律。因此,它们具有普遍的可执行性。除非在需要使用的基础上获得批准,否则规则不得被忽视。与规则相比,指南提供了建议和最佳实践。这些内容值得遵循,甚至强烈建议遵循,但与规则不同,它们通常有一定的变化空间。

我们将自己定义的规则、编写代码时必须遵循的注意事项都收集在我们的编程风格指南中,这些指南被视为经典。“风格”在这里可能有点用词不当,它意味着仅限于格式化实践的集合。

我们的风格指南远不止这些;它们是管理我们代码的全套惯例。这并不是说我们的风格指南是严格规定性的;风格指南规则可能需要判断,例如使用“**在合理范围内尽可能描述性**”的名称的规则。相反,我们的风格指南是要求我们的工程师遵守的规则的权威来源。

我们为 Google 使用的每种编程语言维护单独的代码风格指南。¹从总体上看,所有指南都有类似的目标,旨在指导代码

¹我们的许多风格指南都有外部版本,您可以在<https://google.github.io/styleguide>找到。我们本章中引用了这些指南中的大量示例。

着眼于可持续性的开发。同时，它们在范围、长度和内容方面也存在很大差异。编程语言具有不同的优势、不同的特性、不同的优先级，并且在 Google 不断发展的代码库中采用的历史路径也不同。因此，独立定制每种语言的指南更为实用。我们的一些风格指南简洁明了，侧重于命名和格式等一些总体原则，如 Dart、R 和 Shell 指南中所示。其他风格指南则包括更多细节，深入研究特定的语言特性并延伸到更长的文档 - 尤其是我们的 C++、Python 和 Java 指南。有些风格指南非常重视典型的非 Google 语言用法 - 我们的 Go 风格指南非常简短，只在摘要指令中添加了几条规则，以遵守外部 [认可的惯例中概述的做法](#)。其他包括与外部规范根本不同的规则；我们的 C++ 规则不允许使用异常，这是 Google 代码之外广泛使用的语言特性。

甚至我们自己的风格指南之间也存在很大差异，因此很难确定风格指南应涵盖哪些内容。指导 Google 风格指南开发的决策源于保持代码库可持续性的需要。其他组织的代码库对可持续性的要求各不相同，因此需要一套不同的定制规则。本章讨论了指导我们规则和指南开发的原则和流程，主要从 Google 的 C++、Python 和 Java 风格指南中选取示例。

为什么要有规则？

那么我们为什么要制定规则呢？制定规则的目的是鼓励“好”行为并阻止“坏”行为。对“好”和“坏”的解释因组织而异，取决于组织关心的是什么。这样的指定不是普遍的偏好；好与坏是主观的，并根据需要量身定制。对于某些组织，“好”可能会促进支持较小内存占用的使用模式或优先考虑潜在的运行时优化。在其他组织中，“好”可能会促进使用新语言功能的选择。

有时，组织最关心的一致性，因此任何与现有模式不一致的事情都是“坏的”。我们必须首先认识到特定组织重视什么；我们使用规则和指导来鼓励和阻止相应的行为。

随着组织的发展，既定的规则和指导方针塑造了编码的通用词汇。通用词汇使工程师能够专注于他们的代码需要表达什么，而不是他们如何表达。通过塑造这种词汇，工程师将倾向于默认甚至潜意识地做“好”的事情。因此，规则为我们提供了广泛的杠杆作用，可以将常见的开发模式推向期望的方向。

创建规则

在定义一套规则时,关键问题不是“我们应该制定什么规则?”

需要问的问题是:“我们试图推进什么目标?”当我们专注于规则将服务的目标时,确定哪些规则支持该目标可以更轻松地提炼出一组有用的规则。在Google,风格指南是编码实践的法律,我们不会问“风格指南中包含什么?”而是问“为什么风格指南中包含某些内容?”通过制定一套规范代码编写的规则,我们的组织获得了什么好处?

指导原则让我们把事情放在背景

中:Google 的工程组织由 30,000 多名工程师组成。这些工程师的技能和背景千差万别。每天,大约有 60,000 次提交,提交到包含超过 20 亿行代码的代码库中,这些代码库可能会存在数十年。我们正在优化一组与大多数其他组织所需的价值观不同的价值观,但在某种程度上,这些问题无处不在 我们需要维持一个既能适应规模又能适应时间的工程环境。

在此背景下,我们制定规则的目的是管理开发环境的复杂性,保持代码库易于管理,同时仍允许工程师高效工作。我们在这里做出权衡:帮助我们实现这一目标的大量规则确实意味着我们在限制选择。我们失去了一些灵活性,甚至可能得罪一些人,但权威标准带来的一致性和减少冲突的好处胜出。

基于此观点,我们认识到指导我们制定规则的一些总体原则,这些原则必须:

- 发挥自己的作用
- 为读者进行优化
- 保持一致
- 避免容易出错和令人惊讶的结构 · 必要时向实际

情况让步

规则必须发挥作用并非所有内容

都应该纳入样式指南。要求组织中的所有工程师学习和适应任何新制定的规则，成本不为零。规则太多，²不仅工程师在编写代码时很难记住所有相关规则，而且新工程师也更难学习。规则越多，维护规则集就越困难，成本也越高。

为此，我们特意选择不包括那些不言而喻的规则。

Google 的风格指南并非旨在以律师的方式进行解释；仅仅因为某件事没有被明确禁止并不意味着它是合法的。例如，C++ 风格指南没有禁止使用 goto 的规则。C ++ 程序员已经倾向于避免使用它，因此包含一条禁止它的明确规则会带来不必要的开销。如果只有一两个工程师犯了错误，通过创建新规则来增加每个人的心理负担是不可持续的。

为读者优化我们规则的另一

个原则是为代码的读者而不是作者进行优化。随着时间的推移，我们的代码被阅读的频率将远远高于被编写的频率。我们宁愿代码输入起来乏味，也不愿代码难以阅读。在我们的 Python 风格指南中，当讨论条件表达式时，我们认识到它们比 if 语句更短，因此对代码作者来说更方便。但是，由于它们往往比更冗长的 if 语句更难让读者理解，因此我们限制了它们的使用。³我们更看重“易读”而不是“易写”。我们在这里做了一个权衡：当工程师必须反复输入可能更长的描述性变量和类型名称时，前期成本可能会更高。我们选择支付这笔费用，以换取它为所有未来读者提供的可读性。

作为优先考虑的一部分，我们还要求工程师在代码中留下预期行为的明确证据。我们希望读者在阅读代码时能够清楚地了解代码的作用。例如，我们的 Java、JavaScript 和 C++ 风格指南要求在方法重写超类方法时使用 override 注释或关键字。即使没有明确的设计证据，读者也很可能弄清楚这种意图，尽管每个读者在阅读代码时都需要多花点功夫。

² 工具在这里很重要。衡量“太多”的标准不是实际使用的规则数量，而是工程师需要记住多少规则。例如，在 clang-format 出现之前的糟糕时代，我们需要记住大量的格式化规则。这些规则并没有消失，但有了我们目前的工具，遵守这些规则的成本已经大幅下降。我们已经达到了这样的地步：有人可以添加任意数量的格式化规则，没有人会在意，因为工具会帮你完成这些工作。

当预期行为可能令人惊讶时,其证据就变得更加重要。在 C++ 中,有时很难仅通过阅读一段代码来跟踪指针的所有权。如果将指针传递给函数,而我们又不熟悉该函数的行为,我们就无法确定会发生什么。调用者是否仍然拥有该指针?该函数是否取得了所有权?函数返回后我还能继续使用该指针吗?或者它可能已被删除?为了避免这个问题,我们的C++ 风格指南更喜欢使用`std::unique_ptr`当打算转让所有权时。`unique_ptr`是一种管理指针所有权的构造,可确保只存在一个指针副本。当函数以`unique_ptr`作为参数并打算取得指针所有权时,调用者必须明确调用

移动语义:

```
// 接受 Foo* 的函数可能会或可能不会承担传递的指针的所有权。void TakeFoo (Foo* arg);
```

```
// 对函数的调用不会告诉读者有关函数返回后所有权方面的任何信息。
```

```
Foo* my_foo(NewFoo());
TakeFoo(my_foo);
```

将此与以下内容进行比较:

```
// 接受 std::unique_ptr<Foo> 的函数。void TakeFoo
(std::unique_ptr<Foo> arg);
```

```
// 对该函数的任何调用都明确表明所有权已被 // 放弃,并且函数返回后不能使用 unique_ptr。
std ::unique_ptr<Foo> my_foo(FooFactory()); TakeFoo(std::move(my_foo));
```

根据样式指南规则,我们保证所有调用点在适用时都会包含所有权转移的明确证据。有了这个信号,代码的读者就不需要了解每个函数调用的行为。我们在 API 中提供了足够的信息来推断其交互。这种对调用点行为的清晰记录可确保代码片段保持可读性和可理解性。我们的目标是局部推理,目标是清楚地了解调用点发生的事情,而无需查找和引用其他代码 (包括函数的实现)。

大多数涵盖注释的风格指南规则也旨在支持为读者提供现场证据这一目标。文档注释 (添加到给定文件、类或函数前面的块注释)描述了后面代码的设计或意图。实现注释 (散布在代码本身中的注释)证明或突出显示不明显的选择,解释棘手的部分,并强调代码的重要部分。我们有涵盖两种类型的风格指南规则

评论,要求工程师提供另一位工程师在阅读代码时可能需要的解释。

保持一致

我们对代码库一致性的看法与我们在 Google 办公室中采用的理念类似。由于工程人员数量庞大且分布广泛,团队经常分散在不同的办公室,Google 员工经常需要前往其他地点。尽管每个办公室都保持着独特的个性,融合了当地的风情和风格,但对于完成工作所需的一切,我们刻意保持了相同。来访的 Google 员工的徽章可与所有本地徽章读取器配合使用;任何 Google 设备都将始终获得 WiFi;任何会议室中的视频会议设置都将具有相同的界面。Google 员工无需花时间学习如何设置这一切;他们知道无论身在何处,一切都是一样的。在办公室之间移动很容易,而且仍然可以完成工作。

这正是我们努力追求的源代码。一致性使得任何工程师都能快速进入代码库中不熟悉的部分并开始工作。本地项目可以有其独特的个性,但其工具是相同的,其技术是相同的,其库是相同的,并且一切都正常工作。

一致性的优势尽管办公室

被禁止定制徽章阅读器或视频会议界面可能会让人感到受限,但一致性带来的好处远远超过我们失去的创作自由。代码也是如此:保持一致性有时可能会让人感到受限,但这意味着更多的工程师可以用更少的努力完成更多的工作:³

- 当代码库在风格和规范上具有内部一致性时,编写代码的工程师和阅读代码的其他人就可以专注于正在完成的工作,而不是代码的呈现方式。在很大程度上,这种一致性允许专家进行分块。⁴当我们使用相同的接口解决问题并以一致的方式格式化代码时,专家可以更轻松地浏览一些代码,集中精力于重要内容并了解其作用。这也使模块化代码和发现重复变得更容易。出于这些原因,我们将大量注意力集中在一致的命名约定、一致使用通用模式以及一致的格式和结构上。还有许多规则对看似很小的问题做出决定,仅仅是为了保证事情只以一种方式完成。

例如

³感谢 H. Wright 对现实世界的比较,他在访问了大约 15 个不同的谷歌办公室。

⁴“组块化”是一种认知过程,它将信息碎片组合成有意义的“组块”,而不是单独记录它们。例如,国际象棋高手会考虑棋子的配置,而不是单个棋子的位置。

例如,选择用于缩进的空格数或设置行长的限制。⁵这里有价值的部分是答案的一致性,而不是答案本身。

- 一致性可以实现扩展。工具是组织扩展的关键,一致的代码使构建能够理解、编辑和生成代码的工具变得更加容易。如果每个人的代码都存在差异,那么依赖于统一性的工具的全部优势就无法得到充分发挥。如果某个工具可以通过添加缺失的导入或删除未使用的包含来保持源文件更新,如果不同的项目为其导入列表选择不同的排序策略,那么该工具可能无法在任何地方工作。当每个人都使用相同的组件,并且每个人的代码都遵循相同的结构和组织规则时,我们可以投资于可以在任何地方工作的工具,为我们的许多维护任务构建自动化。如果每个团队都需要单独投资于同一工具的定制版本,以适应他们独特的环境,我们就会失去这一优势。
- 一致性也有助于扩展组织的人力部分。随着组织的发展,在代码库上工作的工程师数量也会增加。

尽可能保持每个人编写的代码一致,可以实现更好的跨项目移动性,最大限度地减少工程师更换团队的准备时间,并使组织能够随着员工人数需求的波动而灵活调整。不断发展的组织还意味着其他角色的人员也会与代码进行交互,例如 SRE、库工程师和代码管理员。在 Google,这些角色通常跨越多个项目,这意味着不熟悉某个团队项目的工程师可能会加入进来处理该项目的代码。整个代码库的一致体验使这一切变得高效。

· 一致性还能确保适应时间。随着时间的推移,工程师离开项目,新人加入,所有权转移,项目合并或分裂。努力实现一致的代码库可确保这些转换的成本低廉,并允许代码和编写代码的工程师几乎不受约束地流动,从而简化长期维护所需的流程。

⁵参见4.2 块缩进:+2 个空格,空格与制表符, 4.4 列数限制:100和线长。

规模化

几年前,我们的 C++ 风格指南承诺几乎永远不会改变导致旧代码不一致的风格指南规则:“在某些情况下,改变某些风格规则可能会有充分的理由,但为了保持一致性,我们仍然会保持原样。”

当代码库变小,陈旧、积灰的角落变少时,
感觉。

当代码库变得越来越大、越来越老时,这不再是需要优先考虑的事情了。
这是(至少对于我们 C++ 风格指南背后的仲裁者来说)一次有意识的改变:当删除这一点时,我们明确指出 C++ 代码库将永远不会再完全一致,我们甚至也没有以此为目标。

不仅要将规则更新为当前的最佳实践,而且还要将这些规则应用于曾经编写的所有内容,这实在是太过沉重了。我们的大规模变更工具和流程使我们能够更新几乎所有的代码,以遵循几乎所有的新模式或语法,以便大多数旧代码都展现出最新批准的样式(参见第 22 章)。然而,这种机制并不完美;当代码库变得如此之大时,我们无法确保每一段旧代码都能符合新的最佳实践。要求完美的一致性已经到了代价太高的地步。

制定标准。当我们提倡一致性时,我们倾向于关注内部一致性。有时,本地惯例会在采用全球惯例之前出现,调整所有内容以适应全球惯例是不合理的。在这种情况下,我们提倡一致性的层次结构:“保持一致”从本地开始,其中给定文件中的规范优先于给定团队的规范,给定团队的规范优先于大型项目的规范,大型项目的规范优先于整个代码库的规范。事实上,风格指南包含许多明确遵循本地惯例的规则,⁶重视这种本地一致性而不是科学的技术选择。

然而,组织仅仅制定并坚持一套内部惯例是不够的。有时,还应考虑外部社区采用的标准。

⁶ `const` 的使用,例如。

计算空格Google 的

Python 风格指南最初要求所有 Python 代码都使用两个空格的缩进。外部 Python 社区使用的标准 Python 风格指南则使用四个空格的缩进。我们早期的 Python 开发大部分是直接支持 C++ 项目,而不是实际的 Python 应用程序。因此,我们选择使用两个空格的缩进,以与已经采用这种格式的 C++ 代码保持一致。随着时间的推移,我们发现这种理由并不成立。编写 Python 代码的工程师读写其他 Python 代码的频率远远高于读写 C++ 代码。我们的工程师每次需要查找某些内容或引用外部代码片段时,都要花费额外的精力。每次尝试将代码片段导出到开源社区时,我们也要经历很多痛苦,花时间调和内部代码和我们想要加入的外部世界之间的差异。

当Starlark⁷的时机到来时 (谷歌设计的一种基于 Python 的语言,用作构建描述语言)拥有自己的风格指南,我们选择改
为使用四个空格的缩进,以与外界保持一致。⁷

如果已经存在惯例,那么组织与外界保持一致通常是一个好主意。对于小型、独立和短期的努力,它可能不会产生任何影响;内部一致性比项目有限范围之外发生的任何事情都重要。一旦时间的流逝和潜在的扩展成为因素,您的代码与外部项目交互甚至最终进入外部世界的可能性就会增加。从长远来看,遵守广泛接受的标准可能会带来回报。

避免容易出错和令人意外的构造我们的风格

指南限制使用我们所用语言中一些更令人意外、不寻常或棘手的构造。复杂的功能通常存在乍一看并不明显的微妙陷阱。在不彻底了解其复杂性的情况下使用这些功能很容易误用它们并引入错误。即使项目工程师很好地理解了构造,也不能保证未来的项目成员和维护者也有相同的理解。

这就是我们的 Python 风格指南规定避免使用强大功能的原因例如反射。反射 Python 函数 `hasattr()` 和 `getattr()` 允许用户使用字符串访问对象的属性:

⁷ 使用 Starlark 实现的 BUILD 文件的样式格式由 buildifier 应用。请参阅<https://github.com/bazelbuild/buildtools>。

```
如果hasattr(my_object,   foo  ):
    some_var = getattr(my_object,   foo  )
```

现在,有了这个例子,一切似乎都很好。但请考虑一下:some_le.py:

```
A_CONSTANT =
[  foo  ,
  bar  ,
  baz  ,]
```

其他_le.py:

```
values = []对于
some_file.A_CONSTANT中的字段:
values.append(getattr(my_object, field))
```

在搜索代码时,您如何知道字段foo、bar和baz正在被访问?没有为读者留下明确的证据。您无法轻易看到,因此无法轻易验证哪些字符串用于访问对象的属性。如果我们不是从A_CONSTANT读取这些值,而是从远程过程调用(RPC)请求消息或数据存储中读取它们,会怎么样?这种混淆的代码可能会导致重大的安全漏洞,只需错误地验证消息,就很难注意到这个漏洞。测试和验证这样的代码也很困难。

Python的动态特性允许这种行为,并且在非常有限的情况下,使用hasattr()和getattr()是有效的。然而,在大多数情况下,它们只会导致混淆并引入错误。

虽然这些高级语言特性可能完美地解决了知道如何利用它们的专家的问题,但强大的特性通常更难理解,而且使用得并不广泛。我们需要所有工程师都能在代码库中操作,而不仅仅是专家。这不仅对新手软件工程师提供支持,而且对SRE来说也是一个更好的环境。如果SRE正在调试生产中断,他们会跳到任何可疑代码中,即使是用他们不熟悉的语言编写的代码。我们更看重简化、直接的代码,这些代码更容易理解和维护。

向实用主义让步正如拉尔夫·沃

尔多·爱默生所说：“愚蠢的一致性是小心眼的人的恶魔。”在追求一致、简化的代码库时,我们不想盲目地忽略其他一切。我们知道,我们的风格指南中的某些规则会遇到需要例外的情况,这没关系。必要时,我们会允许对可能与我们的规则相冲突的优化和实用性做出让步。

性能很重要。有时,即使牺牲一致性或可读性,进行性能优化也是有意义的。例如,虽然我们的 C++ 风格指南禁止使用异常,但它包含一条允许使用`noexcept` 的规则,可以触发编译器优化的异常相关语言说明符。

互操作性也很重要。如果针对目标进行量身定制,设计用于与特定非 Google 组件配合使用的代码可能会表现更好。例如,我们的 C++ 风格指南对通用的 CamelCase 命名指南进行了例外处理,允许对模仿标准库功能的实体使用标准库的 `snake_case` 风格。⁸ C++ 风格指南还允许对 Windows 编程进行豁免,与平台特性的兼容性需要多重继承,而其他所有 C++ 代码都明确禁止这样做。我们的 Java 和 JavaScript 风格指南都明确指出,生成的代码经常与项目所有权之外的组件交互或依赖这些组件,因此超出了指南规则的范围。⁹一致性至关重要;适应性是关键。

风格指南那么,语言风格

指南包括哪些内容?所有风格指南规则大致分为三类:

- 避免危险的规则 · 执行最佳
- 实践的规则 · 确保一致性的规则

避免危险首先,我们

的风格指南包括关于语言特性的规则,这些特性出于技术原因必须或不能实现。我们有关于如何使用静态成员和变量;有关使用 lambda 表达式的规则;有关处理异常的规则;有关线程构建、访问控制和类继承的规则。

我们介绍了应使用哪些语言特性以及应避免使用哪些结构。我们列出了可以使用的标准词汇类型以及用途。我们特别包括了难以使用和难以正确使用的裁定 某些语言特性具有细微的使用模式,可能不直观或不易应用

⁸请参阅[命名规则的例外情况](https://github.com/abseil/abseil-cpp/blob/master absl/utility/utility.h)。例如,我们的开源 Abseil 库使用 `snake_case` 命名来替代标准类型。请参阅[https://github.com/abseil/abseil-cpp/blob/master absl/utility/utility.h 中定义的类型](https://github.com/abseil/abseil-cpp/blob/master absl/utility/utility.h)。这些是 C++14 标准类型的 C++11 实现,因此使用标准青睐的 `snake_case` 样式,而不是 Google 首选的 CamelCase 形式。

⁹参见[生成的代码:大部分免除](#)。

不正确,导致细微的错误潜入。对于指南中的每一项裁决,我们都旨在包括权衡利弊,并对所做出的决定作出解释。这些决定大多基于对时间弹性的需求,支持和鼓励可维护的语言使用。

执行最佳实践我们的风格指南

还包括执行一些编写源代码的最佳实践的规则。这些规则有助于保持代码库的健康和可维护性。例如,我们指定代码作者必须在何处以及如何添加注释。¹⁰我们的注释规则涵盖了注释的一般惯例,并扩展到必须包含代码内文档的特定情况 - 这些情况的意图并不总是很明显,例如 switch 语句中的 fallthrough、空的异常捕获块和模板元编程。我们还有详细说明源文件结构的规则,概述了预期内容的组织方式。我们有关于命名的规则:包的命名、类的命名、函数的命名、变量的命名。所有这些规则旨在指导工程师支持更健康、更可持续的代码的实践。

我们的风格指南强制执行的一些最佳实践旨在使源代码更具可读性。许多格式规则都属于此类别。我们的风格指南指定了何时以及如何使用垂直和水平空格来提高可读性。它们还涵盖了行长限制和括号对齐。对于某些语言,我们通过遵循自动格式化工具来满足格式要求 - Go 的 gofmt, Dart 的 dartfmt。列出格式要求的详细列表或命名必须应用的工具,目标是相同的:我们有一套一致的格式规则,旨在提高可读性,我们将其应用于所有代码。

我们的风格指南还对新的和尚未被充分理解的语言特性进行了限制。我们的目标是在我们学习的过程中,预先为特性的潜在缺陷安装安全围栏。同时,在每个人都开始使用之前,限制使用使我们有机会观察使用模式的发展,并从我们观察到的例子中提取最佳实践。对于这些新功能,一开始,我们有时不确定应该给出什么正确的指导。随着采用的扩大,想要以不同方式使用新功能的工程师会与风格指南所有者讨论他们的例子,要求允许在初始限制所涵盖的范围之外使用更多用例。通过观察收到的豁免请求,我们可以了解该特性的使用情况,并最终收集到足够多的例子来从坏的做法中总结出好的做法。在我们

¹⁰请参阅<https://google.github.io/styleguide/cppguide.html#Comments>, <http://google.github.io/styleguide/pyguide#38-评论和文档字符串>,以及<https://google.github.io/styleguide/javaguide.html#s7-javadoc>,其中多种语言定义了一般的评论规则。

有了这些信息,我们就可以重新回到限制性裁决并进行修改,以允许更广泛的使用。

案例研究:引入 std::unique_ptr当 C++11 引入

`std::unique_ptr` (一种智能指针类型,用于表示对动态分配对象的独占所有权,并在`unique_ptr`超出范围时删除该对象)时,我们的风格指南最初不允许使用。`unique_ptr` 的行为对大多数工程师来说并不熟悉,并且该语言引入的相关移动语义非常新颖,对大多数工程师来说非常令人困惑。

防止在代码库中引入`std::unique_ptr`似乎是更安全的选择。我们更新了工具以捕获对不允许类型的引用,并保留了现有指南,建议使用其他类型的现有智能指针。

随着时间流逝。工程师们有机会适应移动语义的含义,我们越来越相信使用`std::unique_ptr`直接符合我们的风格指导目标。`std ::unique_ptr`在函数调用站点提供的有关对象所有权的信息使读者更容易理解该代码。引入这种新类型以及随之而来的新移动语义所带来的额外复杂性仍然是一个强烈的担忧,但代码库长期整体状态的显著改善使得采用`std::unique_ptr`成为一种值得的权衡。

建立一致性我们的风格指

南还包含涵盖许多较小内容的规则。对于这些规则,我们制定和记录决策主要是为了制定和记录决策。此类别中的许多规则并没有重大的技术影响。诸如命名约定、缩进间距、导入顺序之类的事情:一种形式相对于另一种形式通常没有明确的、可衡量的技术优势,这可能是技术社区倾向于继续争论它们的原因。11通过选择一种,我们就退出了无休止的争论循环,可以继续前进。我们的工程师不再花时间讨论两个空格与四个空格。对于这一类规则来说,重要的不是我们为给定规则选择了什么,而是我们选择了它。

对于其他一切...

尽管如此,我们的风格指南中还有很多内容没有提及。我们试图专注于对代码库健康影响最大的事物。这些文档中确实有一些最佳实践没有明确说明,包括许多基本的良好工程建议:不要耍小聪明,不要分叉代码库,不要重新发明

11这样的讨论其实只是在开玩笑,帕金森琐碎定律的例证。

wheel 等。我们的风格指南等文档无法帮助完全的新手达到对软件工程的大师级理解 有些事情我们假设 ,这是有意为之。

改变规则

我们的风格指南并不是一成不变的。与大多数事物一样 ,随着时间的推移 ,制定风格指南决策的环境和指导特定裁决的因素可能会发生变化。有时 ,条件变化足以需要重新评估。如果发布了新的语言版本 ,我们可能需要更新规则以允许或排除新特性和习语。如果某条规则导致工程师投入精力来规避它 ,我们可能需要重新审视该规则本应提供的好处。如果我们用来执行规则的工具变得过于复杂且难以维护 ,那么规则本身可能已经衰落 ,需要重新审视。注意何时需要重新审视一条规则是保持规则集相关性和最新性的重要部分。

我们的风格指南中捕捉的规则背后的决策都有证据支持。

在添加规则时 ,我们会花时间讨论和分析相关的利弊以及潜在后果 ,试图验证给定的更改是否适合 Google 的运营规模。Google 风格指南中的大多数条目都包括这些考虑因素 ,列出在此过程中权衡的利弊 ,并给出最终裁决的理由。理想情况下 ,我们会优先考虑这些详细的理由 ,并将其包含在每条规则中。

记录决策背后的原因可以让我们认识到何时需要改变。随着时间的流逝和条件的变化 ,之前做出的好决定可能不是当前最好的决定。

通过清楚地记录影响因素 ,我们能够确定何时与这些因素中的一个或多个相关的变化需要重新评估规则。

案例研究 :CamelCase 命名在 Google,当

我们为 Python 代码定义初始样式指导时 ,我们选择使用 CamelCase 命名样式而不是 snake_case 命名样式作为方法名称。

尽管公共 Python 代码指南(PEP 8) 并且大多数 Python 社区都使用蛇形命名法 ,当时 Google 的 Python 大部分用于 C++ 开发人员 ,他们将 Python 用作 C++ 代码库之上的脚本层。许多定义的 Python 类型都是相应 C++ 类型的包装器 ,而且由于 Google 的 C++ 命名约定遵循 CamelCase 风格 ,因此跨语言一致性被视为关键。

后来 ,我们发展到了构建和支持独立 Python 应用程序的阶段。最常使用 Python 的工程师是 Python 工程师

开发人员需要开发 Python 项目,而不是 C++ 工程师来编写快速脚本。

我们给 Python 工程师带来了一定程度的尴尬和可读性问题,要求他们为我们的内部代码维护一个标准,但每次引用外部代码时都要不断调整为另一个标准。我们也让有 Python 经验的新员工更难适应我们的代码库规范。

随着 Python 项目的发展,我们的代码与外部 Python 项目的交互也越来越频繁。我们为一些项目整合了第三方 Python 库,导致我们的代码库中混合使用了我们自己的 CamelCase 格式和外部首选的 snake_case 样式。当我们开始开源一些 Python 项目时,在一个不符合惯例的外部世界中维护它们,这不仅增加了我们的复杂性,也增加了社区的警惕性,社区认为我们的风格令人惊讶且有些奇怪。

面对这些争论,在讨论了成本(与其他 Google 代码失去一致性、对习惯了我们的 Python 风格的 Google 员工进行再教育)和好处(与大多数其他 Python 代码保持一致、允许第三方库中已经泄漏的内容)之后,Python 风格指南的风格仲裁者决定更改规则。由于限制将其作为文件范围的选择应用、现有代码的豁免以及项目自行决定最适合自己的方式,Google Python 风格指南已更新为允许使用蛇形命名法。

流程

考虑到我们想要的长寿命和可扩展性,我们认识到事情需要改变,因此我们创建了一个更新规则的流程。更改样式指南的流程是基于解决方案的。样式指南更新的提案以此观点为框架,确定现有问题并提出建议的更改作为修复方法。在这个过程中,“问题”不是可能出错的假设例子;问题是通过现有 Google 代码中发现的模式来证明的。给出一个已证明的问题,因为我们有现有样式指南决策背后的详细理由,我们可以重新评估,检查现在不同的结论是否更有意义。

按照风格指南编写代码的工程师社区通常最能注意到何时需要更改规则。事实上,在 Google,我们对风格指南的大多数更改都是从社区讨论开始的。任何工程师都可以提出问题或提出更改建议,通常是从专门用于风格指南讨论的特定语言邮件列表开始。

风格指南变更提案可能已完整形成,并建议具体、更新的措辞,也可能以关于特定规则适用性的模糊问题开始。社区会讨论收到的想法,并收到其他语言用户的反馈。一些提案被社区共识否决,并经过评估

没有必要、太模糊或没有益处。其他提案则收到积极反馈，被评估为有价值，无论是原样还是经过一些改进。这些提案，即通过社区审查的提案，需要经过最终决策批准。

风格仲裁者在 Google，对

于每种语言的风格指南，最终的决定和批准均由风格指南的所有者（我们的风格仲裁者）做出。对于每种编程语言，一组长期从事该领域的语言专家是风格指南的所有者和指定的决策者。特定语言的风格仲裁者通常是在该语言库团队的高级成员和其他具有相关语言经验的长期 Google 员工。

任何风格指南变更的实际决策都是对拟议修改的工程权衡的讨论。仲裁者在风格指南优化的商定目标背景下做出决策。变更不是根据个人喜好进行的；而是权衡判断。事实上，C++ 风格仲裁小组目前由四名成员组成。这可能看起来很奇怪：委员会成员人数为奇数可以防止在出现分歧决策时出现平局。然而，由于决策方法的性质，没有什么是“因为我认为应该这样”，一切都是权衡的评估，所以决策是通过协商一致而不是投票做出的。这个四人小组很高兴能按原样运作。

例外是的，我们

的规则是法律，但有些规则确实需要例外。我们的规则通常是为更广泛、更普遍的情况而设计的。有时，特定情况会受益于对特定规则的豁免。当出现这种情况时，将咨询风格仲裁员以确定是否有理由豁免特定规则。

豁免并非轻易就能获得。在 C++ 代码中，如果引入了宏 API，则样式指南要求使用项目特定的前缀来命名它。由于 C++ 处理宏的方式，将它们视为全局命名空间的成员，因此从头文件导出的所有宏都必须具有全局唯一的名称，以防止冲突。关于宏命名的样式指南规则确实允许仲裁者授予某些真正全局的实用程序宏豁免。但是，当要求排除项目特定前缀的豁免请求背后的原因归结为由于宏名称长度或项目一致性而产生的偏好时，豁免将被拒绝。

这里代码库的完整性比项目的一致性更重要。

如果认为允许违反规则比避免违反规则更有利，则允许例外。C++ 风格指南不允许隐式类型 con

版本,包括单参数构造函数。但是,对于设计为透明地包装其他类型的类型,其中底层数据仍然准确且精确地表示,允许隐式转换是完全合理的。

在这种情况下,可以豁免无隐式转换规则。有如此明确的有效豁免案例可能表明需要澄清或修改相关规则。但是,对于此特定规则,收到的豁免请求足够多,这些请求看似符合豁免的有效案例,但实际上并不符合 因为所讨论的特定类型实际上不是透明包装器类型,或者因为该类型是包装器但实际上不需要 因此保持规则原样仍然是值得的。

指导

除了规则之外,我们还以各种形式提供编程指导,从对复杂主题的长期深入讨论到我们认为的最佳实践的简短而尖锐的建议。

指导代表了我们工程经验的集体智慧,记录了我们从一路上获得的经验教训中提取的最佳实践。

指导往往侧重于我们观察到的人们经常犯错的事情或不熟悉且容易引起混淆的新事物。如果规则是“必须的”,那么我们的指导就是“应该的”。

我们培养的指导池的一个例子是针对我们使用的一些主要语言的一套入门书。虽然我们的风格指南是规范性的,规定了哪些语言特性是允许的,哪些是不允许的,但这些入门书是描述性的,解释了指南所认可的特性。它们的覆盖范围相当广泛,几乎涉及到谷歌新接触该语言的工程师需要参考的所有主题。它们不会深入研究给定主题的每个细节,但它们提供了解释和推荐用法。当工程师需要弄清楚如何应用他们想要使用的特性时,这些入门书旨在作为首选的指导参考。

几年前,我们开始发布一系列 C++ 技巧,其中既有通用的语言建议,也有 Google 特定的技巧。我们涵盖了难点 对象生命周期、复制和移动语义、依赖参数的查找;新内容 代码库中采用的 C++ 11 特性、预先采用的 C++17 类型,如`string_view`、`optional`和`variant`;以及需要温和纠正的内容 提醒不要使用`using`指令,警告要记住注意隐式`bool`转换。这些技巧源于遇到的实际问题,解决了风格指南未涵盖的实际编程问题。与风格指南中的规则不同,它们的建议不是真正的经典;它们仍然属于建议而非规则的范畴。然而,鉴于它们是从观察到的模式而非抽象理想中发展而来的,它们的广泛和直接的适用性使它们与众不同

与大多数其他建议不同,它是一种“通用准则”。提示重点明确且相对较短,每条提示只需几分钟即可阅读。这个“每周提示”系列在内部取得了巨大成功,在代码审查和技术讨论中经常被引用。¹²

软件工程师在加入新项目或代码库时,会了解他们将要使用的编程语言,但缺乏 Google 内部如何使用该编程语言的知识。为了弥补这一差距,我们为每种主要编程语言维护了一系列“<语言>@Google 101”课程。这些全天课程重点介绍在我们的代码库中使用该语言进行开发的不同之处。它们涵盖了最常用的库和习语、内部首选项和自定义工具使用。对于刚刚成为 Google C++ 工程师的 C++ 工程师来说,该课程填补了缺失的部分,使他们不仅成为一名优秀的工程师,而且成为一名优秀的 Google 代码库工程师。

除了教授旨在让完全不熟悉我们设置的人快速上手的课程外,我们还为深入代码库的工程师准备了现成的参考资料,以便他们随时找到可以帮助他们的信息。这些参考资料形式各异,涵盖了我们使用的各种语言。我们内部维护的一些有用参考资料包括:

- 针对通常更难纠正的领域(如并发性和散列)提供特定于语言的建议。 · 详细分析语言更新中引入的新功能以及如何在代码库中使用它们的建议。
- 列出我们库提供的关键抽象和数据结构。这可以防止我们重新发明已经存在的结构,并可以解决“我需要一个东西,但我不知道它在我们的库中叫什么”的问题。

应用规则

规则本质上是可执行的,因此其价值更大。规则可以通过社会、教学和培训来执行,也可以通过技术、工具来执行。我们在 Google 开设了各种正式培训课程,涵盖了我们的规则所需的许多最佳实践。我们还投入资源来保持文档的更新,以确保参考资料保持准确和最新。在提高对规则的认识和理解方面,我们整体培训方法的一个关键部分是代码审查所起的作用。我们在 Google 运行的可读性流程(通过代码审查指导刚接触 Google 特定语言开发环境的工程师)在很大程度上是为了培养习惯

¹² <https://abseil.io/tips> 精选了一些我们最受欢迎的提示。

以及我们的风格指南所要求的模式（请参阅第 3 章中有关可读性过程的详细信息）。该过程是我们确保跨项目边界学习和应用这些实践的重要组成部分。

尽管某种程度的培训总是必要的（毕竟，工程师必须学习规则，这样他们才能编写遵循规则的代码），但在检查合规性时，我们强烈倾向于使用工具来自动执行，而不是完全依赖基于工程师的验证。

自动规则执行可确保规则不会随着时间的推移或组织规模扩大而被丢弃或遗忘。新人加入；他们可能还不了解所有规则。规则会随着时间而变化；即使沟通良好，也不是每个人都会记住所有事情的当前状态。项目发展并添加新功能；以前不相关的规则突然变得适用。工程师检查规则合规性取决于记忆或文档，这两者都可能失败。只要我们的工具保持最新，与规则变化同步，我们就知道所有工程师都在将我们的规则应用于所有项目。

自动执行的另一个优势是将规则解释和应用方式的差异降至最低。当我们编写脚本或使用工具检查合规性时，我们会根据单一、不变的规则定义验证所有输入。

我们不会将解释权交给每个工程师。人类工程师看待一切事物的视角都带有偏见。无论是否无意识，偏见可能很微妙，甚至可能是无害的，但仍然会改变人们看待事物的方式。将执行权交给工程师可能会导致对规则的解释和应用不一致，并可能导致对责任的期望不一致。我们越是委托工具，留给人类偏见的切入点就越少。

工具还使执行具有可扩展性。随着组织的发展，一个专家团队就可以编写公司其他成员可以使用的工具。如果公司规模扩大一倍，在整个组织内执行所有规则的努力不会翻倍，成本与以前大致相同。

即使我们通过整合工具获得了优势，也可能无法自动执行所有规则。一些技术规则明确要求人工判断。例如，在 C++ 风格指南中：“避免复杂的模板元编程。”“使用auto避免嘈杂、明显或不重要的类型名称 - 类型无助于读者清晰理解的情况。”“组合通常比继承更合适。”在 Java 风格指南中：“对于如何 [对类的成员和初始化器进行排序]，没有单一的正确方法；不同的类可能以不同的方式对其内容进行排序。”“对捕获的异常不做任何响应很少是正确的。”“重写Object.finalize 的情况极其罕见。”

对于所有这些规则，都需要判断，而工具还不能（暂时！）取代这一点。

其他规则是社会性的而非技术性的,用技术手段解决社会问题往往是不明智的。对于许多属于这一类别的规则,细节往往定义得不太明确,工具也会变得复杂且昂贵。将这些规则的执行留给人来执行往往更好。例如,当谈到给定代码更改的规模(即受影响的文件数和修改的行数)时,我们建议工程师倾向于进行较小的更改。小的更改易于工程师审核,因此审核往往更快、更彻底。它们也不太可能引入 bug,因为更容易推断出较小更改的潜在影响和效果。然而,小的定义有些模糊。将相同的一行更新传播到数百个文件的更改实际上可能易于审核。相比之下,较小的 20 行更改可能会引入复杂的逻辑和难以评估的副作用。

我们认识到,对代码规模的衡量有很多种方法,其中有些方法可能带有主观性——尤其是在考虑变更的复杂性时。这就是为什么我们没有任何工具可以自动拒绝超出任意行数限制的变更提议。如果审阅者认为变更过大,他们可以(并且确实)拒绝。对于此规则和类似规则,执行由编写和审阅代码的工程师自行决定。但是,当涉及技术规则时,只要可行,我们倾向于技术执行。

错误检查器

许多涵盖语言使用的规则都可以通过静态分析工具来执行。事实上,我们的一些 C++ 库管理员在 2018 年年中对 C++ 风格指南进行了一次非正式调查,估计其中大约 90% 的规则可以自动验证。错误检查工具采用一组规则或模式,并验证给定的代码示例是否完全符合要求。自动验证消除了代码作者记住所有适用规则的负担。如果工程师只需要查找违规警告(其中许多都附带建议的修复),这些警告是在代码审查期间由紧密集成到开发工作流程中的分析器发现的,那么我们就可以最大限度地减少遵守规则所需的工作量。当我们开始使用工具根据源标记标记已弃用的函数,并就地显示警告和建议的修复时,弃用 API 的新用法问题几乎在一夜间消失了。降低合规成本使工程师更有可能愉快地遵守规则。

我们使用像[clang-tidy](#)这样的工具(对于 C++)且[容易出错](#)(适用于 Java)自动执行规则的过程。有关我们的方法的深入讨论,请参阅[第 20 章](#)。

我们使用的工具是经过设计和定制的,以支持我们定义的规则。大多数支持规则的工具都是绝对的;每个人都必须遵守规则,所以每个人都使用检查规则的工具。有时,当工具支持最佳实践时

在遵守惯例方面有更多灵活性,存在选择退出机制以允许项目根据其需求进行调整。

代码格式化程序

在 Google,我们通常使用自动样式检查器和格式化程序来强制代码中的格式一致。行长问题已不再令人感兴趣。¹³工程师只需运行样式检查器并继续前进。如果每次都以相同的方式进行格式化,则在代码审查期间就不再是问题,从而消除了原本用于查找、标记和修复细微样式缺陷的审查周期。

在管理有史以来最大的代码库时,我们有机会观察人工格式化与自动化工具格式化的结果。

机器人的平均水平比人类高出很多。有些地方需要领域专业知识 例如,格式化矩阵通常比通用格式化程序做得更好。除此之外,使用自动样式检查器格式化代码很少会出错。

我们通过提交前检查来强制使用这些格式化程序:在提交代码之前,服务会检查在代码上运行格式化程序是否会产生任何差异。如果会产生差异,则拒绝提交,并提供有关如何运行格式化程序来修复代码的说明。Google 的大多数代码都经过这样的提交前检查。对于我们的代码,我们使用`clang-format`适用于 C++; `yapf`的内部包装器对于 Python; `gofmt`用于 Go;`dartfmt`用于 Dart;以及`buildifier`用于我们的BUILD文件。

案例研究:gofmt

萨米尔·阿吉马尼

2009 年 11 月 10 日,Google 将 Go 编程语言开源。从那时起,Go 就发展成为一种开发服务、工具、云基础设施和开源软件的语言。¹⁴我们从第一天起就知道我们需要一种标准的 Go 代码格式。我们还知道,开源发布后几乎不可能再改进标准

格式。因此,最初的 Go 版本包含了 `gofmt`,这是 Go 的标准格式化工具。

¹³当你考虑到至少需要两名工程师进行讨论,并且将这个数字乘以在 30,000 多名工程师的集合中可能发生的次数时,就会发现“多少个字符”可能是一个非常昂贵的问题。

¹⁴ 2018 年 12 月,根据拉取请求衡量,Go 是 GitHub 上排名第四的语言。

动机

代码审查是软件工程的最佳实践,然而在审查中花费了太多时间争论格式问题。虽然标准格式可能不是每个人都喜欢的,但它足以消除这种浪费的时间。¹⁵通过标准化格式,我们可以自动更新 Go 代码而不会产生虚假差异的工具奠定了基础:机器编辑的代码与人工编辑的代码将无法区分。¹⁶例如,在 2012 年 Go 1.0 发布前的几个月里,Go 团队使用了一

个名为 gofix 的工具将 1.0 之前的 Go 代码自动更新为语言和库的稳定版本。多亏了 gofmt,gofix 生成的差异只包括重要的部分:对语言和 API 使用的更改。这使得程序员可以更轻松地查看更改并从工具所做的更改中学习。

影响

Go 程序员希望所有 Go 代码都使用 gofmt 格式化。gofmt 没有配置旋钮,其行为很少改变。所有主流编辑器和 IDE 都使用 gofmt 或模拟其行为,因此几乎所有现存的 Go 代码都采用相同的格式。起初,Go 用户抱怨强制执行标准;现在,用户经常将 gofmt 作为他们喜欢 Go 的众多原因之一。即使在阅读不熟悉的 Go 代码时,格式也很熟悉。

数以千计的开源软件包读取和写入 Go 代码。¹⁷由于所有编辑器和 IDE 都同意 Go 格式,因此 Go 工具是可移植的,并且可以通过命令行轻松集成到新的开发环境和工作流程中。

回火

2012 年,我们决定使用一种新的标准格式化程序: buildifier 自动格式化 Google 的所有 BUILD 文件。BUILD 文件包含使用 Google 的构建系统 Blaze 构建 Google 软件的规则。标准 BUILD 格式使我们能够创建自动编辑 BUILD 文件而不破坏其格式的工具,就像 Go 工具处理 Go 文件一样。

一位工程师花了六周时间才让谷歌 20 万个 BUILD 文件的重新格式化被各代码所有者接受,在此期间,超过一千名新

¹⁵ Robert Griesemer 于 2015 年发表演讲,“gofmt 的文化进化”详细介绍了 gofmt 的动机、设计以及对 Go 和其他语言的影响。

¹⁶ Russ Cox 于 2009 年解释 gofmt 是关于自动更改的:“所以我们拥有一个程序操作工具的所有困难部分,只等着被使用。同意接受 ‘gofmt 风格’是让它在有限的代码量中可行的关键。”

¹⁷ Go AST 和 格式包 每个国家都有数千家进口商。

每周都会添加BUILD文件。Google 的用于进行大规模更改的新兴基础设施大大加速了这一进程。
(参见第22章。)

结论

对于任何组织,尤其是像 Google 工程团队这样庞大的组织,规则可以帮助我们管理复杂性并构建可维护的代码库。一套共享的规则构成了工程流程的框架,以便它们可以扩大规模并不断发展,从而确保代码库和组织的长期可持续发展。

TL;DR

- 规则和指导应旨在支持对时间和规模的适应性。 · 了解数据,以便调整规则。
- 并非所有事情都应该是规则。 · 一致性是关键。 · 尽可能自动执行。