

开放式协作

开源软件的生产与维护

[美]娜迪亚·埃格巴尔 著

X-lab 开放实验室 译

开源社 审校

华东师范大学出版社
• 上海 •

Introduction of the Author

作者介绍

娜迪亚·埃格巴尔

是一位探讨互联网如何赋能于个人创作者的作家和学者。2015 至 2019 年间,她专注于开源软件的生产,在独立工作与 GitHub 工作之间切换,以改善开源开发者的体验。她也是《路与桥:数字基础设施背后的隐形劳动》的作者,在书中她认为,开源代码是一种需要维护的公共基础设施。



Recommendation

推荐语

1. 娜迪亚从开源、经济学和诗意的独特的交汇视角写作。这是关于在线创作社区动向的权威书籍。

—— 纳特·弗里德曼 (Nat Friedman), 前 GitHub CEO

2. 娜迪亚是当今对在线社区的深度和潜力观察得最细致入微的思想家之一。这本书的出版恰逢其时,因为互联网对我们彼此之间的连接方式的影响已经变得更加强烈。

—— 德文·祖格尔 (Devon Zuegel), 前 GitHub 社区产品总监

3. 在信息丰富的时代,我们都是维护者。《开放式协作》从人类学的视角,对真实的开发者故事进行了深入研究,为我们提供了一种通过开源“镜头”提出新问题的方法。娜迪亚所关注的不仅是围绕着金钱、许可证和代码的问题,还有作为各种创造者的我们所有人。

—— 亨利·朱 (Henry Zhu), Babel 维护者

4. 《开放式协作》值得一读。现代软件开发者都会很感兴趣,无论是对编程新手还是使用了数百个开源软件包来支持私有代码的专业人士,本书中都有一些见解和示例可供借鉴。即使是对书中所说的对各类问题非常熟悉的开源维护者,也能获得更广泛的社区意识,以及如何使开源可持续的新想法。

—— 威廉·文森特 (William Vincent), Django 软件基金会董事会成员

5. 《开放式协作》是一本引人入胜的书……我们需要重新思考众包的能力——并

认识到它可能比承诺的更有限。开源革命是在一些非常疲倦的人身上进行的。

——Wired 网站

6. 娜迪亚帮我们拓展了认知创作者、开源软件和经济的方式。任何好奇开源如何改变我们的工作方式、如何改变创作者的生活的人，都应该读这本书，并加入这场讨论。

——斯托米·彼得斯（Stormy Peters），GitHub 副总裁

7. 开源开放是大势所趋，没有开源就没有互联网的发展，没有开源就没有数字化转型。开源是手段，也是趋势，更是一种文化、一种创新的氛围。热烈祝贺本书的正式出版！

——周傲英，华东师范大学副校长

8. 这本书以广阔又深入的视角，呈现了开源世界正在发生的变迁。我们认为开源的本质是协作，娜迪亚说，“在过去的 20 年中，开源的发展莫名其妙地从依靠社区协作转变为依靠个人努力”。我们认为开源社区崇尚个人价值，娜迪亚说，“GitHub 在现代开发者中如此流行体现了便利战胜个人价值的经典技术故事”。文中许多有趣的洞见，是愉悦的阅读。

——周明辉，北京大学教授

9. 开源是一个新世界！这个世界新奇又有趣，现实又理想。与传统的以技术视角来看待开源不同，这本由娜迪亚所写、X-lab 开放实验室所翻译的书，以开源创作者的视角，来研究与探讨形成社区链接的各个角色相互之间的关系以及他们所共同形成的协作者模型。了解这个模型，对于大部分有志于参与开源社区贡献的开发者而言，是有益的。

——堵俊平，开放原子开源基金会 TOC 主席，华为计算开源业务总经理

10. 开源给我们打开了一扇门，在网络空间中链接起一群有共同目标的人，以一种不可思议的方式激发着每个人的创造力，创造出令人叹为观止的成就。这本书使我们在实现人生自我价值的道路上有很多启发。

——王永雷，新思科技开源治理专家，开源社成员

11. 开源让大家能够在一个公开的场所跨越时空和组织的边界进行协作。是怎样的一个“魔法”让大家在公开平台上进行有效的协同？开源项目是如何运作的？相关的角色与激励关系又是怎样的？开源项目的核心开发人员处于怎么样的工作状态？娜迪亚的这本《开放式协作》通过大量的访谈和细致入微的分析向我们展示了她对这一神奇的开源世界的观察与洞见。无论你是初识开源的爱好者，还是开源项目的亲历者，都可以从这本书中受益匪浅。

——姜宁，华为开源软件管理中心技术专家，Apache 软件基金会现任董事，ALC-Beijing 发起人

12. 人类曾经有过各种各样的协作形式，随着自由/开源软件的蓬勃发展，开放式协作也正在快速演进之中。在这样的演进过程中，创新与困惑总是相伴相生，乐观与悲观也会交替出现。本书就是这个领域的最新探索与全新洞见，特别推荐给所有对人类协作感兴趣的朋友。

——庄表伟，开源社理事，华为开源管理中心开源专家

13. 开源就是开放式协作，但是支撑这种开放式协作背后的原因是什么，又是如何组织的？本书是知名作家娜迪亚·埃格巴尔在完成业内非常有名的一篇报告《路与桥：数字基础设施背后的隐形劳动》（可能是开源软件经济学最具代表性的研究）之后，完成的又一部大作。该书聚焦于开源软件的开发和维护工作，介绍和分析了开源软件社区工作中的角色、激励、组织等。她说：“开源构成了数字基础设施的支撑，理解基础设施的维护成本是了解如何关照我们的数字未来的关键。”这本书出版于 2020 年 7 月，王伟老师团队慧眼识珠，把这本书翻译成中文带给国内开源爱好者，有助于国内开源相关工作者了解开源软件社区背后运作的机制、角色、工具和平台等，是非常有价值的工作。

——谭中意，开放原子开源基金会 TOC 副主席，Apache 软件基金会成员

14. 开源软件经历近四十年的发展，很多深层次问题开始显现，理想的开源模式与现实的开源项目之间出现巨大的脱节——很多开源项目背后根本没有社

区,只有个人的努力,孤军奋战的维护者们倍受经济回报和质量保障的压力;开源项目面临着大量的长尾贡献,有近一半的贡献者仅贡献了一次,这又给维护者们带来大量的协调管理工作;等等。如今,开源模式已进入呼唤新思想、新变革、新机制、新发展的新阶段,而这本书正是难得的探讨这些开源模式深层次问题的好书,从引言开始便充满思想性和启发性,值得一读。

——包云岗,中科院计算所副所长

15. 近几年,开源在国内的关注度得到了前所未有的提升,从开发者群体进入到了更广阔的视线范围之内。也有越来越多的人在关注,开源到底是什么,我们应该怎样去发展开源。本书通过软件开发的视角,以自身实践开源的所思所想为主线,深入浅出地抛出了很多观点。有一点非常认同:开源和开放式协作有紧密的联系,是个人创作、行为模式和文化思想的综合体,“就像橡皮泥一样将它们揉在一起”。《开放式协作》是一本非常值得阅读的开源书籍。感谢 X-lab 开放实验室将本书翻译成中文以飨读者。

——杨丽蕴,中国电子技术标准化研究院研究室主任

Translator's Preface

译者序

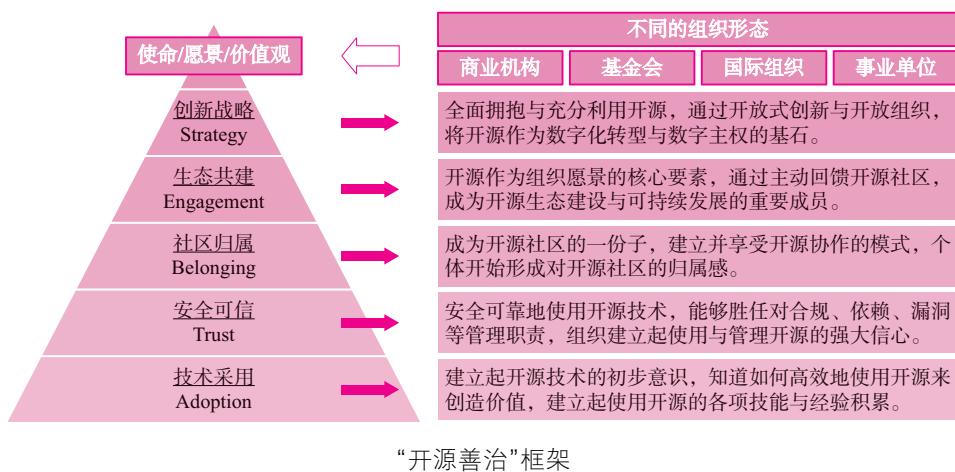
第一次接触 *Working in Public* 这本书是在一个国内的开源共读群里。当时有开源“布道师”热烈推荐，而书名也一下子就吸引了我们。于是开始组织实验室的老师和同学们于 2021 年的暑假，在 B 站上开展了一轮公开共读与分享的活动，吸引了不少国内的开源爱好者围观。

在经过了解、越发体会到这本书的价值后，我们萌生了翻译这本书的想法。几番周折，最终由华东师范大学出版社获得了本书的中文版权，我们的翻译之旅也随之启程。开源作为我们实验室的一个主要研究方向，书中的话题深深地吸引着我们，而整本书的翻译过程也是十分新颖的，我们采取了一种异步协作的模式，也在中西表达模式的不同中学习到了不少翻译小技巧。

在我们组织翻译的同时，2022 年初，X-lab 开放实验室联合开源社共同发起了 ONES Group 工作组，继续实践“开源治理、国际接轨、社区发展”的宗旨。随着全球数字化的不断深入，开源也开始走向大众化。提高全球对开源的认识、促进各方（包括企业、政府、高校等）对开源的结构化和专业化管理、帮助公司和公共机构发现和理解开源、帮助所有人在整个开源生态中受益等，是我们通过开源连接到一起的初心：Openness(开放)，Networking(连接)，Equality(对等)，Sharing(分享)。

著名的“马斯洛的需求层次理论”是亚伯拉罕·马斯洛 (Abraham Maslow) 在其 1954 年出版的著作《动机与人格》(Motivation and Personality) 当中第一次完整阐述的。这个框架很容易通过跨学科的方式（如社会学、心理学、管理学等）引起人们的共识，凡是涉及人和组织的地方亦是如此，开源也不例外。在 ONES Group 的推动下，我们参照 Linux 基金会旗下的 TODO Group 及欧盟的 OW2 组织，设计了“开源善

治”的总体框架,希望能够给在国内所有企业、社区、高校、基金会等组织机构中与开源事务相关的人士作参考,共同推动全球化的开源生态的建设。



“开源善治”框架

娜迪亚的这部作品也同样反映了上述理念。全书主要从开源项目的生产与维护两大视角进行阐述,特别关注了开源项目和社区背后的贡献者。通过非常多样性的实例与调研,讲述了开源背后的快乐以及辛苦。正如审校者之一的赵生宇博士在他的一篇博客标题中所主张的那样:开放协作的世界里,每一份贡献都值得回报。

这是一本难得的从社会学、经济学、心理学等不同层面分析开源项目与社区背后的现象,进而总结相关原理的作品。实际上,在这本书之前,娜迪亚的另外一部作品《路与桥:数字基础设施背后的隐形劳动》也很早被圈内的开源人士所熟知。正如娜迪亚所主张的,数字基础设施应被视为一种必须的公共品,自由而免费的公共源代码,不仅使企业与组织构建软件的成本呈指数级降低,还使技术在全球范围内更容易获取。然而,这些伟大工作的背后,则主要是由开源社区的志愿者所共同创建与维护的,他们这样做是为了建立自己的声誉,出于责任感,或者仅仅是出于爱心与兴趣。

两部作品中,娜迪亚都在不断地强调,开发者与社区志愿者的长期激励和补偿方案没有得到足够的重视,那些获得价值回报的开发者因此认为自己是“幸运的”。当前数字基础设施的商业模式包括来自公司或个人的“赏金”、赞助或捐

赠。尽管基金会、学术界和企业等机构在支持数字基础设施方面做出了一些努力,但还远远不够,尤其是公司还没有主动投身进来,参与这场全球性的公共基础设施开发协作。

由于开源依赖于人力而非财力资源,因此仅靠金钱并不能解决所有问题。我们需要的是对开源文化有细致入微的理解,以及管理(治理)而不是操控数字基础设施的方法。娜迪亚建议的资助和支持数字基础设施的措施包括:去中心化、主动与现有开源社区合作并提供长期和全面的支持、提高对数字基础设施挑战的认识、让组织机构更容易贡献时间和金钱、扩大并多样化开源贡献者群体、制定跨基础设施项目的最佳实践和政策等,这些都将大大有助于建立一个健康和可持续的开源生态系统。

本书的初稿翻译工作由来自华东师范大学数据科学与工程学院的在读博士生夏小雅牵头,翻译团队成员包括承担主要翻译工作的顾业鸣、杨鸣、朱香宁,以及参与部分翻译工作的何莹、徐焕、赵景元、郝斯尘、李为公、陆长权。在初稿完成后,我们自觉专业程度与水平还不够,于是邀请开源社的一众拥有不同企业背景的开源专业人士帮我们进行审校,并很快得到答复。开源社是 X-lab 开放实验室的长期战略合作伙伴,他们中参与整个审校工作的专业人士包括:庄表伟(第一章、第五章)、姜宁(第二章、第三章)、陈阳(第四章、第五章)、赵生宇(第一章、第四章)、王永雷(第二章、第三章)。

在经历过 2022 年新一轮的疫情之后,我们终于将这本科普式的专业书籍译稿交付给了出版社,如释重负,需要感谢的人太多。因为作者的语言习惯,翻译团队在 2021 年开始分工的时候,普遍感觉翻译难度较大。其间我们组织了多次研讨会,统一了整体翻译风格与相关术语。进入到专家审校阶段后,由于时间和地理位置分布的原因,我们采取了全线上异步协作的模式,将开源协同的那套工作方式应用到书籍翻译的过程中,颇为成功。开源共创的文化与精神再次得到了体现。进入到出版阶段,华东师范大学出版社同样也给予了我们巨大的支持,在此表示衷心的感谢。

谨以此书献给我们热爱的开源事业!

X-lab 开放实验室

2022 年 10 月

Author's recommendation Preface

作者推荐序

当我在 GitHub 工作时,还记得那是 2018 年,我们正在为即将发布的 Octoverse 报告(GitHub 对其平台上新兴趋势的年度报告)整理文案。我和数据团队的人坐在会议室里,看着同事在白板上画图。那是一张粗略的草图,展示了当年亚洲与其他地区的用户增长情况,各种颜色的线条代表世界各地的非亚洲地区,然后我的同事为亚洲画了一条线,那条标记线贯穿了整个白板。

那一年,亚洲创建的开源项目比包括美国在内的世界其他任何地方都多。中国成为全球第二大 GitHub 最受开源贡献者欢迎的国家。^①自 2014 年以来,GitHub 的亚洲开源贡献者数量创历史新高,2017—2018 年贡献者增长超过所有其他地区。截至 2019 年,GitHub 上近三分之一的亚洲开发者来自中国。^②

众所周知,开源文化以西方为中心,但亚洲的区域增长表明开源的中心正在迅速转移。在《开放式协作》的开篇中,我有提到,我的观察主要针对居住在美国、欧洲和澳大利亚的开发者。我提到这一点是为了使本书中谈到的现象不至于过时,因为在写作和研究过程中,我意识到我们现在所认识的开源会在未来几年发生变化。

正如开源项目不再像 2000 年代初那样,它们也不一定像我们在西方国家看到的那样。当我在 2018 年研究其中一些趋势时发现,中国的开源开发者的行为与西方国家明显不同。中国的开发者在 GitHub 上显著地**创建和使用**开源项目,

^① The State of Octoverse 2018. [EB/OL]. [2022-04-13]. <https://octoverse.github.com/2018/people/#location>.

^② The State of Octoverse 2019. [EB/OL]. [2022-04-13]. <https://octoverse.github.com/2019/#regions>.

但他们似乎更踟蹰于为他人的代码仓库做贡献。

开源文化历来强调**活跃贡献者**或深度参与开源项目的开发者的作用。但正如第3章所述，开源项目中有多种角色——包括维护者、活跃贡献者和用户。随着越来越多的开发者在亚洲发起、发展和维护开源项目，其中一些项目可能活跃贡献者会很少，而用户更多——正如我在第2章中描述的“体育场”模型——这些差异不仅受到技术决策和领导风格的影响，还会受到地域的影响。

软件发展的速度很快。这种感觉很奇妙，即使我刚刚才写完《开放式协作》，但开源历史的全新篇章才刚刚展开。我十分荣幸能够与世界上发展最快地区之一的开发者分享我所学到的知识。随着本书的读者扩展到全球，我希望本书中的思想和概念能帮助我们描述开源新时代的篇章。

娜迪亚·埃格巴尔(Nadia Eghbal)

2021.10.10

Contents

目 录

引言 / 1

第一部分 人们怎样生产 / 11

01 GitHub 开源平台 / 13

02 开源项目的结构 / 33

03 角色，激励和关系 / 55

第二部分 人们怎样维护 / 91

04 软件所需的工作 / 93

05 管理生产的成本 / 125

结论 / 169

致谢 / 183

注释 / 185

后记 / 221

引言

直到最近,人们仍认为,信息是很有用的,并且信息越多越好。如果思想的自由交流构成了繁荣社会的基础,那么我们在道德上有义务使更多的人彼此联系。

这种开放的精神持续了超过 200 年。我们倡导识字和教育的价值。我们修建了道路、桥梁和公路,将从前各自独立的社区汇聚在一起。我们对新的千禧年充满了向往。

因此,在 20 世纪末,当互联网开始从萌芽转变为肆意生长,它携带了人们和谐一致地迷恋于永无止境地传播知识的所有特质。万维网的发明者蒂姆·伯纳斯-李(Tim Berners-Lee)在给孵化他项目的欧洲粒子物理实验室——欧洲核子研究中心(CERN)最初的提议中,设想了一个“可以繁衍和生长的信息池”^[1]。他写道:“为了使之成为可能,存储的方式一定不能对信息加以限制。”将他的提议作为蓝图,技术人员建立了一个规模超过了我们在物理领域中可以想象的亚历山大图书馆,将世界各地的人们及其思想紧密地联系在一起。

然后我们遇到了障碍。生活中突然之间充斥了大量的信息,太多的通知使我们希望减少查看它们的次数;太多的社交互动使我们希望减少在线发帖的频率;太多的电子邮件使我们不想回复。实际上,我们之间正在互相 DDoS: 这是一个全称为“分布式拒绝服务攻击”的术语,指恶意行为者通过巨大的流量来淹没被攻击目标,从而使受害者丧失服务能力。我们的在线公共生活变得难以应付,导致我们许多人缩回到了私人领域。

在开源软件的世界中,也发生着类似的故事。“开源”作为一个几乎与“公共协作”同义的术语,其开发者(编写和发布任何人都可以使用的代码的人)却经常对入站请求数量感到不堪重负。

在采访了数百名开源开发者并研究了他们的项目之后,我在福特基金会 2016 年的一份报告中总结了这个问题,报告题为“路与桥: 数字基础设施背后的隐形劳动”^[2]。然后我开始尝试着手解决这个问题。

我的大部分压力测试都发生在 2016 年至 2018 年,当时我正在努力改善 GitHub 的开源开发者体验。GitHub 是一家为开源项目提供托管服务的公司。作为一个大多数开源软件在其之上构建的平台,GitHub 是理想的学习场所。这期间我遇到了更多开发者,也参与了更多项目,让我不得不切实考虑解决方案。

在这过程中,不乏一些言辞与现实相悖的情况出现,这往往令我感到羞愧。

毫无疑问,许多开源开发者都缺乏支持。我的收件箱中充斥着渴望分享他们的故事的、来自开源领域的人们的电子邮件。但是困难的是如何确定他们的真实需求。

最初,我的探索重点是资金的缺乏。尽管产生了数万亿美元的经济价值,但许多开源开发者并不直接从开源工作中获得报酬。在缺乏额外的声誉或经济利益的情况下,维护供公众使用的代码很快就变成了他们难以推辞的无薪工作。

但随着这些年来对开发者故事的追踪,我注意到金钱只是问题的一部分。一些开发者在没有金钱补偿的情况下,巧妙地处理了用户的需求。而一些有偿开源开发者似乎也在经历着同样奇怪的行为过程,但这种行为过程在为雇主编写“专有”或私有代码的人员中似乎不那么普遍。

这个过程看起来是这样的:开源开发者公开编写和发布他们的代码。他们在聚光灯下度过了几个月甚至几年的时光。但是,最终人气下降导致收益递减。如果维护代码的价值未能超过回报,那么这些开发者中的许多人就会悄悄地退回到阴影中。

受雇于私人公司的开发人员主要与同事合作。在公开平台上编写代码的开发人员必须切切实实与成千上万的陌生人一起工作,因为任何可以接入互联网并关心这个项目的人都可以对其代码发表评论。缺乏奖励也许是激励错位的症状,但是上述不可避免的开发者流失的过程似乎有更深层次的暗示。

当今默认的假设是,面对不断增长的需求,一个软件项目的开源“维护者”(用于指代软件项目的主要开发者)需要找到更多的贡献者。人们通常认为开源软件是由社区构建的,这意味着任何人都可以参与其中,从而分散工作负担。从表面上看,这似乎是正确的解决方案,尤其是因为它看起来很容易实现。如果一个独立的维护者对他的工作量感到筋疲力尽,那么他应该让更多的开发者参与进来。

如今,有无数旨在帮助更多开发者为开源项目做出贡献的计划。这些尝试被广泛拥护为“对开源有益”,并且通常是通过利用公众的善意来实现的。

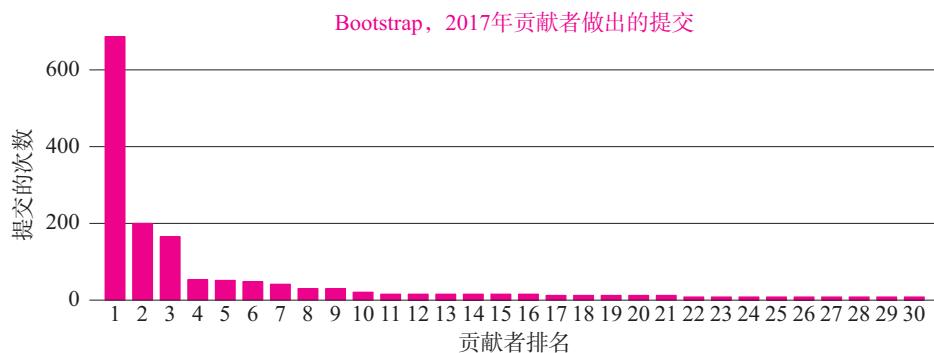
但是,在与维护者私下交谈时,我发现这些举措经常使他们焦虑不安,因为此类举措通常会吸引低质量的贡献。而这给维护者带来了更多的工作。毕竟,

所有贡献都必须经过审核后才能被接受。维护人员经常缺乏将这些贡献者带入“贡献者社区”的基础设施；在许多情况下，项目背后根本没有社区，只有个人的努力。

在与维护者的对话中，我听到他们表达了一种真正意义上的矛盾：他们既希望鼓励新人参与开源工作，又感到无法亲自承担这项工作。维护者根本就没有精力去使每一个表现出一时的兴趣的人都参与其中。许多人告诉我，他们常常为那些对开源项目摇摆不定而最终没有留下来成为贡献者的人感到沮丧。维护者们细数了那些表达了兴趣的人，其中很多甚至在提交第一次贡献之前就消失了。

我开始发现问题不是没有人愿意为开源项目做贡献，而是有太多的贡献者，或者说，他们是错误的那一类贡献者。开源是公开的，但它不一定需要人人参与：维护者可能在过度需求下屈服。

一项研究发现，在使用各种编程语言的 275 个流行的 GitHub 项目中，有近一半的贡献者仅贡献了一次。这些贡献者占据了总提交或总贡献的不到 2%。^[3]使“贡献者社区”这一概念受到质疑的，并不是只有一小部分开发者对项目做出了有意义的贡献，而是通常情况下成百上千的贡献者只能对项目做出微小的实质性贡献。



Bootstrap 在 2017 年的贡献者，根据提交次数进行绘制

我意识到，我们所认为的开源工作方式与开源工作的实际开展之间存在巨大的脱节。开源项目中存在一个极长且还在不断增长的长尾，这些项目并不符合典型的协作模型。这样的项目包括 Bootstrap，它是一种流行的设计框架，大

约有 20% 的网站基于它而运行,^[4]此项目中的三位开发人员承担了 73% 的代码提交。^{*}^[5]另一个例子是 Godot, 一个用于制作游戏的软件框架, 此项目中的两名开发人员每周对项目中出现的 120 多个问题做出回应。^{†^[6]}

这种一个或几个开发者完成大部分工作, 其后是大量的临时贡献者以及更多的被动用户的社区分布在开源中并不是例外, 现在已经成为一种常态。从维护者的角度来看, 在未解决的杂乱无章的问题中, 他们悄悄地守护自己的代码, 这个世界看起来不像是早期互联网先驱拥护的乌托邦理想, 即陌生人之间的大规模合作。如果有有什么不同的话, 它看起来与这些早期倡导者的预测完全相反。一项研究发现, 在研究人员所调查的 GitHub 上 85% 的开源项目中, 不到 5% 的开发人员负责超过 95% 的代码编写工作和社区互动。^[7]在他们的报告中, 研究者们注意到了一种令人困惑的“已建立的理论与广泛观察到的经验效应之间的不匹配”。

这些数字看起来似乎敲响了警铃, 但这仅仅是因为它们不符合大众的期望。我们假设开源项目需要不断壮大的贡献者社区才能生存。这里有一个术语叫“巴士系数”, 指项目的健康程度可以通过度量“有多少个开发者被公交车撞了会使得项目陷入困境”来表示。

但这种表达不再适用于描述有多少个开源项目在运作(译者注: 有多少个开源组件停滞会导致一个项目陷入困境)。鉴于如今软件开发者通常会依靠数百个开源项目来编写代码, 因此不可避免地, 他们只能被动依赖于这些项目。

对开源的期望本来应该是一种协作努力, 因此最终孤军奋战的维护者们感到不知所措, 甚至担心没有人露面是否是因为他们做错了什么。但是, 如果我们从“一切本应如此”的前提开始呢? 我决定重新审视诊断的方式, 把这些症状作为一个起点, 而不是使用我们拥有的唯一处方——更多的参与。

就像当今其他类型的在线内容一样, 代码也趋向于模块化: 它是由一个个小库组成的千层蛋糕, 而不是一个笨拙的大型果冻模具。如今, 开发者可以轻松地在线发布一些代码以供公众使用, 就像其他创作者发现并使用他们的代码一样容易。但是, 就像推文很容易阅读和转发, 而不需要标明作者信息一样, 代码也很容易被复制粘贴, 同时无需知道它们的来源。

npm 生态系统(据其母公司估计, 该系统构成了当今现代网络应用程序中

97%的代码)为未来提供了一些线索。^[8] npm 全称为“Node 包管理器”(Node Package Manager),它是 JavaScript 开发者通常用于安装和管理软件包或库的平台(库是其他开发者可以使用的预先编写的代码包,而不必从头开始编写相同的代码。合法使用其他人的代码的能力使现代软件开发者可以更快、更高效地工作)。

与传统的大型整体软件项目和围绕它们所兴起的繁荣社区相反,npm 软件包的设计是小型且模块化的,这导致每个项目只需要更少的维护者,而且维护者与他们编写的代码形成暂时的关系。从这个角度来看,当今贡献者缺乏反映的是对不断变化的环境状况的适应,其中维护者、贡献者和用户之间的关系更轻松,更易处理。

当与新手和临时贡献者讨论他们的经历时,我发现他们的故事与维护者的一样具有启发性。临时贡献者经常意识到他们对项目背后发生的事情了解甚少。但更重要的是,他们也不想花时间熟悉项目的目标、规划和贡献过程。这些开发者主要将自己视为项目的用户。他们不认为自己是“贡献者社区”的一部分。

维护者的角色正在演变。维护者不再仅是对一组开发者进行协调,而是承担一种内容筛选工作,需要从那些争夺其注意力的频繁交互——如用户问题、错误报告和功能请求——的噪音中筛选出有价值的信息。

在有限的时间和精力下,单个维护者需要在被动任务(社区交互)与主动任务(编写代码)之间取得平衡。维护者还依赖于平台(即 GitHub)和工具(例如可以帮助管理问题、通知和代码质量的 bot)来跟上他们的工作。

如今,维护者面临的问题不是如何吸引更多的贡献者,而是如何管理大量的频繁、低接触的交互。这些开发者并不是在构建社区,而是在指挥空中交通。

开源参与度的大幅提升并没有限制访问代码的途径。使用开源代码的开发者数量呈指数增长。2001 年,SourceForge 是占主导地位的代码托管平台,那时候只有 200000 个用户。^[9] 之后,作为继任者的 GitHub 拥有超过 4 千万的注册用户;而在过去两年中,这个数字几乎翻了一番。^[10]

信息量大,并且唾手可得,这是数字时代最重要的胜利之一。更重要的是,不是因为过多的代码消费,而是因为用户的过多参与在抢占维护者的注意力,才

使得当今的维护工作难以进行。

虽然这个开发者孤军奋战来为用户做事情的世界似乎与开源的故事有明显的偏差,但它跟更广泛的在线世界并没有什么不同。在线世界越来越多地由个人建立,而不仅仅是社区。[‡]就像一位 Reddit 用户观察到的:

如果你在亚马逊上阅读评论,则主要在阅读由格雷迪·哈普(Grady Harp)等人撰写的评论。如果你阅读维基百科,则主要是阅读贾斯汀·纳普(Justin Knapp)等人撰写的文章……还有,如果你阅读 YouTube 评论,则主要是阅读贾斯汀·杨(Justin Young)之类的人发表的评论。你在互联网上消费任何内容,其实主要是在消费由于某些原因而花费大量时间和精力在互联网上的人创建的内容。这些人显然在某些重要的方面和一般人群有所不同。^[1]

作为开发者也是同理,如果你使用命令行工具 cURL,则用的是丹尼尔·斯坦伯格(Daniel Stenberg)编写的代码;如果你使用命令行界面 bash,则用的是切特·拉米(Chet Ramey)维护的代码;如果你使用 npm,则使用的是辛德勒·索尔许斯(Sindre Sorhus)和 Substack 公司编写的软件包;如果你使用 Python 的包管理工具,则使用的是唐纳德·斯塔夫特(Donald Stufft)维护的代码。

与任何其他创作者一样,这些开发者创作的作品与用户交织在一起,同时受到用户的影响。但他们的协作方式和我们通常认为的在线社区的协作方式有所不同。

相比于论坛或 Facebook 群组的用户,GitHub 的开源开发者更类似于 Twitter、Instagram、YouTube 或 Twitch 上的个体创作者。这些创作者必须找到某种方法来管理与广泛且快速增长的受众群体的互动交流。正如流行文化评论家马克·费雪(Mark Fisher)所说的那样,取代了传统的面对面交流,创作者的受众如今面对的是一个舞台。

和其他创作者一样,开源开发者也在创作可以给公众消费的制品。他们也需要处理拥挤的收件箱,管理他们有限的注意力并依赖于热情的支持者。与其

他创作者一样，开源开发者也非常依赖平台来分发其作品。作为封闭的经济体，这些平台还肩负着帮助创作者提高声誉，并且捕捉他们创作内容价值的责任。

早期互联网活动的特点是大规模、分散的在线社区：邮件列表，在线论坛，会员群组。这些社区如同一个个村庄集群，每个村庄都有自己的文化、历史和规范。

社交平台将所有这些社区聚集到了一起，就像橡皮泥一样将它们揉在一起。在这个过程中，我们制作和消费内容的方式被潜移默化地改变了。创作者如今可以接触到更多的潜在观众，但是这些关系是短暂的、片面的，而且常常是压倒性的。

克里斯汀·鲁佩尼安(Kristen Roupenian)在《纽约客》(*New Yorker*)发表的短篇小说《猫人》(*Cat Person*)广为传播之后，回顾了自己的经历，并描述了当时的心情：

我渐渐接近一种状态，描述这个状态的词语可能听起来很戏剧性，它叫做“摧毁”(annihilating)。我和所有思考和谈论我的人面对面，这种感觉就像独自站在体育场中心，而成千上万的人正在对着我放声尖叫。不是为了我尖叫，而是瞄准了我。有些人可能会觉得这样令人振奋。但我不
会。^[13]

在创作者的世界中，开源开发者是一个非常有趣的子集。从经济角度看，代码类似于其他形式的内容。就像书籍或视频一样，代码只是一堆信息，打包起来可以分发。当然它的角色更接近于实用主义。

尽管社交媒体、新闻和娱乐都在我们的生活中扮演着至关重要的公共角色，但我们直接依靠开源代码来保持电话、笔记本电脑、汽车、银行和医院的平稳运行。如果 YouTube 视频出现故障，我们可能仅仅为失去有价值的信息而叹息，但是如果一个开源项目出现故障，它甚至会破坏整个互联网(我们将在第 4 章中看到)。

正因为此，与其他类型的创作者相比，我们审视开源开发者的行会牵引出

一系列基本的经济问题。更方便的是，开源开发者在完全公开的视野下工作，从而使他们的故事更易于被研究。

开源一直是其他在线行为的先锋。在 20 世纪 90 年代后期，开源种下了大规模协作的希望火种，被称作为“同行生产”(peer production)。开源软件实际上也开始超过了商业软件，因此经济学家认为这些开发者已经实现了不可想象的目标。随着互联网从萌芽状态继续向前发展，世界似乎确实有可能最终由自组织社区驱动。

但是在过去的 20 年中，开源的发展莫名其妙地从依靠社区协作转变为依靠个人努力。尽管使用开源代码的人比以往任何时候都多，但其开发者却无法捕捉到他们创造的经济价值——重新审视这个悖论同样是件十分有趣的事情。

通过研究开源从“小型互联网”到“大型互联网”的过渡，我们可以更好地、更广泛地了解在线创作者的情况。我们仍在试图将个人创作者的崛起与报纸、书籍出版商和人才中介的衰落联系在一起——公司不再是变革的主要推动主体。作为一个案例研究，开源可以帮助我们理解为什么我们的网络世界没有像早期学者所预测的那样发展，以及我们的经济如何围绕个人创作者及其建立的平台进行自我调整。

如果创作者（而不是社区）准备成为我们在线社交系统的中心，那么我们需要对他们的工作方式有更好的了解。在当今有 45 亿人在线的世界中，一个人的作用是什么？这些创作者如何塑造我们的品味，以及当热度逐渐消减后，我们如何保护、鼓励和奖励这种工作？平台又是如何帮助或阻碍这种工作的？

* 对于 2017 年的贡献来说确实如此。

[†]该数据收集于 2017 年 8 月 29 日至 2018 年 8 月 29 日之间。

[‡]甚至 Wikipedia（维基百科），世界上最大的在线百科全书、大规模合作的典范例子也是如此。史蒂文·普鲁特（Steven Pruitt）自愿编辑了该网站上所有英语文章的三分之一。^[12]

第一部分 人们怎样生产

第一部分

人们怎样生产

-
- 01 GitHub 开源平台
 - 02 开源项目的结构
 - 03 角色，激励和关系

Part I

01

GitHub 开源平台

本书通过讲述开源的故事,试图厘清并扩展“在线”在当今的真实内涵。通过开源,个体开发者编写的代码可以被数百万人使用。这些个体开发者们的工作并不是要最大程度地参与,与之相反,他们的工作可以被定义为:需要过滤和策划大量的互动。我试图解释为什么会出现这种情况,以及平台是如何将焦点从在线社区转移到独立创造者身上的。

当我探讨这个主题时,我将主要关注使用 GitHub 的开发者。^{*}这并不是在贬低 GitHub 之外的平台的开源价值,也不是在暗示开源只能在单一平台上开发,或者不应该存在迁移到其他平台的机会。我主要写的是个人经历,而不是机构或公司参与者在开源中的角色——这是一个不同的、复杂的主题,值得单独写一本书。

开源在保持独立于任何专有软件、工具或平台方面有着悠久而丰富的历史。开源的故事比 GitHub 早了 20 多年。在很大程度上,开源缺乏标准工具是人为之的:开发者喜欢选择,他们希望可以选择自己最喜欢的工具来工作。

GitHub 对开源产生了巨大的影响。它冲破了自由和开源软件所在建筑的屋顶,落在建筑内的长凳上,压碎了下面的一切。

虽然没有要求开发者必须使用 GitHub 来编写开源软件,但 GitHub 是目前占主导地位的代码托管平台。事实上,大多数开源开发者现在使用 GitHub 不仅仅代表了个人品味的转变,也意味着开发者文化的转变。

平台和它们的创造者之间的关系,对于讨论我们的网络世界如何改变是至关重要的。而且,因为 GitHub 是开源领域的主流平台,如果不讨论平台上所提供的服务,以及在平台上使用并受益于这些服务的开发者,就无法理解平台与创造者之间的关系。

代码自由

在实际托管代码的平台出现之前,大多数开放源代码都以“tarball”(以.tar 文件命名,它将文件打包在一起)的形式发布在一些自托管的独立网站上。开发者使用邮件列表进行沟通和协作。每个项目对其贡献的管理都略有不同。如果开发者想要做出贡献,他们就像来到另一个国家一样,必须首先了解每个平台的

风俗习惯。

开发者使用版本控制来跟踪和同步他们的更改,当有多个人(其中许多人是陌生人)在世界各地不同的时区编写相同的代码时,这一点就变得尤其重要。因此,如果小明和小红对同一行代码进行了更改,版本控制将帮助开发者协调这些差异,并在不破坏任何代码的情况下保持井井有条。

但是现在最流行的版本控制系统 Git 直到 2005 年才发布。在此之前,开发者主要使用集中式版本控制系统,如 Subversion 或 CVS(如果他们使用版本控制的话)。这些系统不是为去中心化的大规模协作而设计的。

在集中式版本控制下,开发者必须将代码提交回相同的服务器,但在分布式版本控制下,每个人都可以分别处理自己的代码副本,然后再将更改的内容与其他人同步。Git(以及几乎同时发布的竞争性系统 Mercurial)是第一个成为主流的分布式版本控制系统,这在技术层面上使得开发者可以独立地工作。

然而,即使在 Git 发布之后,仍然没有一个标准化的开发工作流程。在某种程度上,早期的自由软件和开源软件的开发者喜欢这种工具、习惯和过程之间的不和谐,因为这意味着没有一个工具主宰这个领域,没有人能够完全抓住开发者的注意力。

理查德·斯托曼(Richard Stallman)是发起自由软件运动的 MIT 黑客。他受到鼓舞去发起 GNU 项目——一个自由软件操作系统。缘由于 1983 年,他在麻省理工学院人工智能实验室试图定制施乐打印机时,发现无法访问或修改其源代码。

斯托曼想把代码从私有使用中解放出来。术语“free”指的是能够使用代码做你想做的事情,而不是指代码是免费的。(因此,人们经常重复斯托曼所说的话“Free 是指自由,而不是指免费的啤酒”,以及偶尔使用西班牙语单词 *libre*,而不是 *gratis*,来指自由软件,以区分这两种含义)^[15]

描述出代码自由对于自由软件开发者的重要性是件艰难的事情。非营利性组织软件自由保护协会(Software Freedom Conservancy, SFC)的负责人布拉德利·库恩(Bradley Kuhn)将他的生活方式比作素食主义者。就像素食者不吃肉一样,布拉德利也不使用专有软件。^[16]这意味着不使用 Twitter、Medium、YouTube 或 GitHub 等网站。代码在此处好比是家畜,需要从人类手中解放出

来,即使是以牺牲个人便利为代价。

因此,编写自由软件就是要摆脱通常困扰商业软件环境的种种限制。自由软件是反主流文化,正好与那个时代蓬勃发展的黑客文化相一致。

“黑客”这个词是由作家史蒂文·利维(Steven Levy)普及的,他在《黑客:计算机革命的英雄》(*Hackers: Heroes of the Computer Revolution*)一书中描绘了20世纪80年代的黑客一代,令人印象深刻。在《黑客》一书中,利维介绍了当时许多著名的程序员,包括比尔·盖茨(Bill Gates)、史蒂夫·乔布斯(Steve Jobs)、史蒂夫·沃兹尼亚克(Steve Wozniak)和理查德·斯托曼。他认为黑客相信共享、开放和去中心化,他称之为“黑客伦理”。^[17]根据利维的描述,黑客关心改善世界,但不相信遵循规则就能达到目的。

黑客的特点是虚张声势、表现欲强、爱恶作剧和对权威的极度不信任。黑客文化今天依然存在,就像垮掉的一代、嬉皮士仍然存在一样,但黑客不再像过去那样抓住软件文化的时代精神。如今,黑客一代的接班人可能是密码破译人员和涉猎信息安全的人。

尽管利维在他的书中并没有只关注自由和开源的开发者,但是20世纪80年代和90年代的黑客文化与早期的自由和开源软件紧密相连,正如三位领导者:理查德·斯托曼、埃里克·S.雷蒙德(Eric S. Raymond)和莱纳斯·托瓦兹(Linus Torvalds)所证实的那样。

理查德·斯托曼(也被称为RMS)是20世纪80年代在麻省理工学院发起自由软件运动的黑客。埃里克·S.雷蒙德(也被称为ESR)在20世纪90年代将自由软件重塑为“开源”,他被视为早期开源的非官方人类学家。莱纳斯·托瓦兹则是一个在1991年创建了Linux,一个驱动当今许多操作系统的开源内核,以及在2005年创造了Git的程序员。前一个项目成为早期大规模合作的典范,而后一个项目无意中(也许让托瓦兹懊恼)催生了GitHub。[†]

如果自由软件是一种意识形态,那么斯托曼就是它的忠实信徒、衣衫不齐的传道者。我曾经听人这样描述他:“那个穿着夏威夷长袍、拎着塑料购物袋去上课的人。”他以坚持区分“自由软件”和“开源软件”而闻名;他对植物有一种奇怪的恐惧;他还不请自来地出席演讲,就自由软件的一些小问题向不情愿回答他问题的讲师提问。

埃里克·S.雷蒙德是拥护“开源”这个术语的一群程序员中的一员，他们希望与自由软件的意识形态定位保持距离，并使开源这个想法看起来更有利于商业。他在1997年发表的一篇文章《大教堂和集市：一个偶然的革命者对Linux和开源的思考》(The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary)中对开源软件的好处给出了令人信服的解释。这篇文章认为，比起被限制在一个更小的开发团队中(就像一个“大教堂”)，当更多开发者参与到软件的开发过程中(就像一个“集市”)时，开发者会发现更多的软件bug。第二年，雷蒙德和一群志趣相投的开发者组成了开放源代码促进会(Open Source Initiative)，开始宣传他们的想法。[‡]

除了写关于开源软件的内容，雷蒙德还写一些个人文章，并形成了一个极具辨识度的ESR品牌。他因出版《极客性爱小贴士》(Sex Tips for Geeks)而臭名昭著，这是一本关于泡妞和床上功夫的文章集。^[19]自称“枪支狂人”和自由主义者，雷蒙德还写了一些关于枪支所有权的文章，这些文章被收集在“埃里克的枪支狂人页面”上。^[20]

这三人中的最后一个是莱纳斯·托瓦兹，他在很多方面都很出名，但最臭名昭著的可能是他鲁莽的交流风格，以及他对开源项目的铁腕控制。托瓦兹的邮件列表中充满了咒骂，有时全用大写。2012年，他在阿尔托大学做了一次演讲。当被问及英伟达缺乏对Linux的支持时[英伟达是图形处理单元(GPUs)的制造商]，他转向摄像头，竖起中指，咆哮道：“英伟达，去你的！”^[21]

不仅仅是沟通技巧，托瓦兹的管理风格也使他声名狼藉。雷蒙德在他其中一篇文章中称这种风格为“仁慈的独裁者”^[23]，后来被Python编程语言的作者吉多·范罗苏姆(Guido van Rossum)改编为更著名的短语“一生仁慈的独裁者”，用来描述即使在项目发展过程中仍然保持控制权的开源项目作者。尽管Linux基金会报告说，自2005年以来，Linux内核的贡献者已经超过了14 000人，^[24]但托瓦兹仍然是唯一一个被允许将这些贡献合并到主项目中的人。^[25]

尽管不乏令人难忘的黑客人物，但自由和早期的开源精神也被一些不受关注的人们所定义。在我最早与自由软件运动的一位杰出成员(我不愿透露其姓名)的一次谈话中，他对开源与开发它的开发者有任何关系的观点都会愤怒地吐槽。他告诉我，代码是“无政府主义的”和“不可触及的”，它必须能够在个人欲望

或需要支付租金以外独立生存。他说：“它需要成为一种任何人都无法夺走的东西。”“系统才是重要的。如果一个开发者离开了，另一个开发者就会介入并维护它。”代码的自由也延伸到编写代码的人的自由。

卡尔·福格尔(Karl Fogel)是流行版本控制系统 Subversion(Git 的集中式前身)的合著者，[§]他对这一观点表示赞同：

当我们说“那是我的自行车”或者“那是我的鞋子”时，我们的意思是我们拥有它们。我们对我们的自行车有最终决定权。但是当我们说“那是我的父亲”或“我的妹妹”时，我们的意思是与他们有联系。很明显我们并没有拥有他们。在开源中，你只能在关联意义上说“我的”。在开源中没有所有格意义上的“我的”。^[26]

因此，可以理解为什么许多开发者对将代码绑定到特定平台的想法感到愤怒。鉴于开源的意识形态根源在于将代码从私有控制中“解放”出来，许多开发者关心的是保持在互联网上任何地方发布和协作编写代码的自由。转移到其他地方的能力在开源设计中无处不在，无论是 *forking*——制作代码副本，还是能够 *clone*——将一个项目下载到本地计算机上。

令这些开发者苦恼的是，GitHub 本身并不是开源的，这意味着，就像斯托曼和他的施乐打印机一样，用户修改平台以满足自己需求的能力是有限的。Git 是用来管理代码的版本控制系统，独立于 GitHub 或任何其他托管平台，因此任何人都可以在其他地方复制和托管项目代码。但是 GitHub 的社区功能——所有的对话都是在平台上进行的——很难导出。(GitHub 紧随其后的竞争对手 GitLab 于 2011 年推出，它大力宣传自己的平台是开源的。GitHub 目前仍是市场的主导者：虽然很难找到采用 GitLab 的用户的公开数字，但它的网站声称有超过 10 万个组织使用它的产品，^[27]而 GitHub 声称有超过 290 万个组织使用其产品^[28])

许多自由软件开发者拒绝使用 GitHub。埃里克·黄(Eric Wong)是开源网络服务器 Unicorn 的维护者，他在回应一位用户“请移到 GitHub 上”的请求时写道：

不。绝对不会。GitHub 是一个专有的通信工具,它要求用户登陆和接受服务条款。这给了一个单一实体(以及一个盈利组织)权力和影响力。我向自由软件贡献的原因是我反对任何形式的被绑定或专有特性。当我遇到没有专用通信工具就不能使用 Git 的用户时,我感到非常难受。^[29]

不管是出于意识形态还是商业原因,自由和早期的开源倡导者都专注于传播开源的理念。但是今天的开发者几乎不再注意到“开源”这个概念。他们只是想要编写并发布自己的代码,而 GitHub 正是让这一切变得容易的平台。

便利的胜利

GitHub 由四名 Ruby 开发者于 2008 年创立,他们将其宣传为一个“社交编程”(social coding)的平台,当时 Web 2.0——Facebook、Twitter、YouTube 和 Tumblr 刚刚起步。通过将所有开源开发者带到一个平台上,GitHub 帮助规范了他们的工作方式。

GitHub 并不是第一个代码托管平台。它的前身是成立于 1999 年的 SourceForge。如果说 GitHub 就像 Facebook,那么 SourceForge 就是代码托管平台中的 MySpace: 它是同类产品中的第一个重要产品,尽管至今仍然存在,但最终都只是作为一幅蓝图被记住。和 GitHub 一样,SourceForge 给开发者提供了一个托管和下载代码的地方,但它更像是一个文件共享网站,而不是一个协作的地方。SourceForge 专注于代码分发;GitHub 专注于改善开发者的工作流程。对于 SourceForge 失败的原因,每个人都有自己的看法。许多人将其归咎于对广告的依赖(这给用户体验带来了负面影响)以及总体上较差的产品开发。其他人则指出 SourceForge 一直不愿意支持 Git 作为版本控制系统。

顾名思义,GitHub 的创始人把它的声誉押在了只有几年历史的 Git 上,将其作为软件协作的未来。在相对较新的技术上加倍投入,帮助 GitHub 吸引了一批刚刚意识到 Git 好处的开发者。(现有的代码托管平台 SourceForge 当时并不支持 Git)GitHub 友好的用户体验帮助提高了人们对 Git 的兴趣,而 Git 的

学习曲线是出了名的陡峭。很难说是 Git 还是 GitHub 促成了对方的成功,但它们结合在一起成为了大规模分布式软件协作的主导工具集。^{¶[30]}

GitHub 将开发者工作流程的每个部分都整合到它的产品中,包括问题跟踪器、开发者个人资料,以及它所谓的“PR”:一种提交、评审和合并贡献的方式。与在繁杂的互联网上搜索 tarball 和邮件列表相比,GitHub 的用户界面简单直观。你可以注册,选择一个用户名,并使用搜索栏来查找项目。打开一个 issue 或 PR 就像点击一个大的绿色按钮一样简单。

到 2017 年,GitHub 托管了 2500 万个公共项目,而且该服务的需求仍在增长。GitHub 称,其 2018 年的新用户数量超过了成立后头六年的总和。^[31](译者注:据维基百科数据显示,截至 2020 年 9 月份,GitHub 宣布已有超过 5600 万的注册用户)

通过吸引所有人到 GitHub 的平台,GitHub 让人们更容易发现新项目,以及每个开发者的历史和声誉。网站创建者还增加了“star”项目的能力,这样就可以知道哪些项目更受欢迎或不受欢迎。不管这个指标有多大的缺陷,现在 star 已经成为开发者们成功的标志,让他们可以将自己的项目与他人的项目进行排名。

GitHub 还帮助普及了宽松许可证的广泛使用,这极大地改善了开放源代码的覆盖范围和分发方式。自由软件开发者经常拥护 copyleft 授权,比如 GNU 通用公共许可证(GPL)。Copyleft 许可会像病毒一样附着在任何使用它们的代码上。也就是说任何包含 GPL 许可代码的代码也必须在 GPL 下进行授权。所以,如果像 Facebook 这样的公司在其移动应用中使用 GPL 授权的代码,该应用也必须按照 GPL 授权其他人使用。可以想象,copyleft 许可在商业上并不友好,因为公司必须以相同的条款许可他们自己的软件。所以早期的开源倡导者开始强调宽松的许可证,比如 BSD 许可证(the Berkeley Software Distribution)和 MIT 许可证(the Massachusetts Institute of Technology),这些许可证允许开发者在不改变他们自己项目条款的情况下,对代码做任何他们想做的事情。

到目前为止,MIT 许可证是 GitHub 项目中最广泛使用的许可证。一篇 2015 年的公司博客文章称,45% 的开源项目都在使用它。虽然 MIT 许可证使开源代码的发布变得顺利,但它“设置后忘记”的风格也被批评为在开源代码和

它的意识形态起源之间制造了裂痕。奇怪的是,从长远来看,GPL 可能对开源开发者更友好,因为它让开发者对其他人如何使用他们的代码有更多的控制。但是很难想象 GPL 如何成为 GitHub 这样的平台上的主导许可证,因为它与平台的一个最大的优势不兼容:自由、不受限制的发布。^[32]

GitHub 在现代开发者中如此流行体现了便利战胜个人价值的经典技术故事。对早期的自由和开源开发者来说,由 GitHub(2018 年被微软收购)这样一家公司引领的向标准工具和工作流的转变,代表着他们自 20 世纪 80 年代以来一直为之奋斗的一切的倒退。代码协作不应该属于任何人,尤其不应该属于一家价值数十亿美元、制作专有软件的公司。

但是 GitHub 一代的开源开发者并不这么认为,因为他们优先考虑便利性,不像自由软件倡导者优先考虑自由,或早期的开源倡导者优先考虑开放。这一代的人不知道,也不真正关心自由和开源软件之间的区别;他们也不热衷于宣传开源理念本身。他们只是在 GitHub 上发布他们的代码,因为就像现在任何其他形式的在线内容一样,共享是默认的。

开发者布雷特·坎农(Brett Cannon)对于将 Python 项目转移到 GitHub 的决定,解释说,他和他的合作者选择 GitHub 而不是 GitLab,部分原因是前者是开发者现在最满意的平台,另一部分原因是它有更好的自动化开发工具。但他引用的第三个原因最能说明早期和现代开源的区别:

GitLab 没有杀手锏。现在有些人会说,GitLab 是开源的,这是它的杀手锏。但对我来说,开发过程比担心基于云计算的服务是否发布源代码更重要。^[33]

GitHub 为开发者提供了完成工作所需的工具。与 20 世纪 80 年代到 21 世纪初的早期工作流相比,GitHub 易于使用、可靠,并且能够处理大规模交互。如果有人知道如何使用 GitHub,他们就很容易进入一个不熟悉的项目并做出贡献。GitHub 用户的个人资料中有照片、简介和他们过去活动的公共链接,所有这些都是自动生成的,这让他们的声誉对于每一个项目来说都是可见的。

GNU,一个自由软件的旗帜项目,并不需要在 GitHub 上获得成功。鉴于它是基于“代码解放”的思想而建立的,我甚至可以说 GNU 只有在无视 GitHub 作为一个平台的情况下才能真正流行起来。GNU 之所以是 GNU,是因为它并不托管在 GitHub 上。

相比之下,今天的这一代开源开发者需要 GitHub 来把他们的工作做到最好。就如同有 Instagram 影响力者和 Twitch 主播一样,这里也有 GitHub 的开发者。这些活动也可以在平台之外进行——你仍然可以把你夏威夷度假的照片或视频上传到一个自托管的网站上——但你为什么要这样做呢?对于那些希望获得受众的人来说,平台和创造者已经变得密不可分。

平台通常会被描述为与创造者对立。《应用:人类故事》(*App: The Human Story*)是一部关于苹果应用商店开发者们如何与平台局限性作斗争的纪录片。^[34]与此同时,Facebook 经常被指责“用一种算法取代了大量出版商的受众”。^[35]但是,除了平台可能造成的所有问题,它们也带来了不可估量的价值。今天的开源开发者似乎真心喜欢 GitHub,认为它是一个编写、分享和发现代码的地方。

本·汤普森(Ben Thompson)在他的博客 Stratechery 上写了一些关于商业和技术的文章,他甚至认为传达这种价值本身就是平台的定义,而不仅仅是作为一个聚合器。平台将价值传递给基于平台建立的第三方,而聚合只是纯粹的中介作用。汤普森引用了比尔·盖茨的一句话,后者对平台的定义是“当每个使用它的人的经济价值超过创造它的公司的价值时”。^[36]基于这个定义,汤普森认为 Facebook 实际上不是一个平台,而是一个聚合器,因为“除非是出于好心,Facebook 没有理由为(媒体)出版商做任何事情”。

就像人才经纪公司一样,平台首先会改善创作者的分销渠道,让他们接触潜在的数百万人,从而为他们增加价值。这种发现主要有利于那些仍在建立受众的创作者。这种反馈是正向循环的,更多创作者会被鼓励加入。只要越来越多的人继续使用这个平台,就不存在某一个创作者会吸走这个房间里所有的氧气的情况。

不同于人才经纪公司,平台没有合同来阻止创作者把观众带到别的地方去
[有些创作者有足够的影响力,如果他们这么做的话,会对平台造成影响:顶级

主播 Ninja 离开 Twitch 去了微软的独家竞争平台, Mixer。碧昂斯(Beyoncé)在她的专辑 Lemonade 发布后的头三年,都只在 Tidal,一个新兴的订阅音乐服务上独家放送^[37]]。所以平台必须让自身变得不可或缺,来最大限度地鼓励创作者留下来。这个措施通常与降低创作者的成本划上等号。正如布雷特·坎农在他关于 Python 迁移的文章中解释的那样,GitHub 帮助他的团队“尽可能地自动化开发过程,同时降低 Python 开发者必须维护的基础设施成本”。^[37] Python 不再需要 GitHub 才能够被发现,但一个成熟的编程语言项目尤其受益于 GitHub 所提供的能够降低成本的基础设施。

因此,平台与创造者的关系必然类似于一种共生的、“俄罗斯套娃”式的混合生产模式,在这种模式中,创作性人才广泛分布,但却被嵌在一个集中平台的茧中。(在 Instagram,或 Spotify,或 Medium 上……任何人都可以成为创作者)尽管创作者来自世界各地,但他们需要这些平台来生存和成长。

事实上,今天的开发者不仅仅是使用,而是积极地偏爱 GitHub,这表明了一套与前几代不同的价值观。使用 GitHub 的简单行为已经将这些开发者与那些拒绝使用任何平台的教条主义的自由软件倡导者区分开来。而现代的开发者不加思考就给他们的项目贴上 MIT 许可证的事实——如果他们愿意为他们的项目申请许可证的话——将他们与早期的开源倡导者区分开来,后者是在“开放”的教条上大肆宣扬与传播的,这种教条通过许可来体现。

GitHub 一代的开源开发者对这些问题并没有特别强烈的感觉。他们只是想创造东西,而分享是对他们努力成果的自然衍生。流行前端框架 Bootstrap 的联合创始人雅克布·桑顿(Jacob Thornton)曾在一次会议上承认,尽管他从事高可见性的开源项目多年,但他“真的不知道开源是什么”。^[39]

史蒂夫·克拉布尼克(Steve Klabnik),一个因他在两个流行的编程语言社区 Rust 和 Ruby 中的工作而出名的开发者,指出过时的词汇限制了我们谈论开源是如何产生的:

为什么自由软件和开源的概念本质上与许可证捆绑在一起是一个问题? 因为这两个运动的目的和目标都是关于分配和消费的,但今天人们最

关心的是软件的生产。软件许可证协议规定分发,但不能规范生产……这也是开源之后的主要挑战。^[40]

通过关注开发者体验,GitHub 让开源更多地关注人而不是项目,这就是开发者迈克尔·罗杰斯(Mikeal Rogers)所说的“开源的业余化”,“push 代码几乎变成了和发 Twitter一样常规”:

我已经为开源项目贡献了 10 多年,但现在不同的是,我不是这些项目的“成员”——我只是一个“用户”,贡献一点点就是作为一个用户的一部分。^[41]

编写代码就像以多种方式发推文一样。包管理器的出现为开发者提供了一种简单的方法来安装和管理与其所选择的编程语言相关联的软件库。包管理器最初出现在 20 世纪 90 年代,以支持 Linux 生态系统,但最终它们成为大多数编程语言的标准工具。例如,编程语言 Ruby 有一个名为 RubyGems 的包管理器,而 Python 有 PyPI。

包管理器使得为软件开发创建、发布和共享可重用组件变得更加容易,这些组件通过一个注册中心进行管理:你可以想象为一个公共图书馆,其中包含数千本书,任何人都可以使用他们的借书证访问这些书。这些管理器极大地加快了开发者从命令行完成的工作。

通过一个安装命令,开发者现在可以导入数百个包(由其他开发者编写的代码块),并在自己的代码中使用它们。结果就是,构成“开源项目”的概念也变得更小了,就像从博客文章到推文的转变一样。

开发者拉斯·考克斯(Russ Cox)在描述这种转变时解释道:

在依赖管理器出现之前,发布一个只有 8 行的代码库是不可想象的:开销太大而收益太少。但是 npm(JavaScript 的包管理器)已经把开销几乎降到了零,其结果是几乎微不足道的功能都可以打包和重用。^[42]

因此,这些新出现的行为成为 JavaScript 的标志性特征之一就不足为奇了。JavaScript 是在 GitHub 上发展起来的语言生态系统。

从黑客到哈士奇

JavaScript 诞生于 1995 年,但最近的技术发展赋予了它新的当代意义。它最初是由布伦丹·艾希(Brendan Eich)为 Netscape Navigator 编写的。因为 JavaScript 是唯一支持在现代浏览器中内置引擎的语言,所以从那时起它就一直与开发者息息相关。

虽然 JavaScript 已经出现了 20 多年,但在它的大部分历史中,主要被用作网页的脚本语言。它与标记语言 HTML 和样式表语言 CSS 一起构成了前端开发的三位一体。HTML 为网页提供了脚手架,CSS 样式化了这些元素,JavaScript 使它们成为动态的。如果网页是一座建筑,则 HTML 构成了骨架,CSS 构成了油漆和干墙,JavaScript 构成了电力和管道。

JavaScript 很容易编写和修改,因为开发者可以直接从浏览器中完成。每个主流浏览器都有一个“查看源代码”选项,这使得检查构成任何网页的 HTML、CSS 和 JavaScript 很容易。对于许多软件开发者来说,JavaScript 为他们提供了一个可访问的编程第一个切入口。

在 2000 年代中期,开发者开始寻找创造性的方法在他们的应用程序后端(对用户不可见的底层逻辑)使用 JavaScript。2009 年,瑞恩·达尔(Ryan Dahl)发布了 Node.js,这是一个在客户端和服务器端都可以轻松运行 JavaScript 的平台。现在,开发者不必为应用程序的后端和前端各学习一种语言,而只需要学习 JavaScript 一种语言,然后在任何地方都能使用它。

现在的 GitHub 项目是用各种各样的编程语言开发的,包括 Java、Ruby 和 PHP,但 JavaScript 比其他所有语言都更占优势。在 GitHub 上,JavaScript 的受欢迎程度是排名第二的 Python 的两倍多。^[43]根据 Stack Overflow 网站的说法,它已经迅速成长为开发者中最常用的编程语言。^[44]

JavaScript 的普遍吸引力使它具有极高的可触达性并且十分强大。从文化的角度来看,它让前端和后端开发者成为了奇怪的伙伴。当代 JavaScript 是在

后 Web 2.0 时代发展起来的。它既友好又精致,但也带有政治色彩。它吸引了那些喜欢解释事物的开发者,他们用表情符号和色彩鲜艳的标识来解释事物。

JavaScript 开发者欣然接受“开源的业余化”,尽管他们的前辈偶尔会把他们翻白眼。npm, JavaScript 的包管理器,通过让同时发布和依赖多个包变得更容易,从而鼓励模块化设计。而且由于 JavaScript 必须保持跨浏览器的兼容性,它的官方核心库比大多数其他语言都要小,开发者不得不定制化编写自己的包来填补空白。因此,每个项目都趋向于更小,更易于使用,就像组装在一起的乐高积木,而不是用石头雕刻的城堡。

自由软件黑客将自己定义为独立于平台的人,但如果我没有 GitHub 这样的平台提供的优势,JavaScript 可能不会像现在这样流行,更不用说 Twitter 和 Slack 这样的现代通信渠道了。与那些强调代码自由高于一切的人不同,JavaScript 社区散发出相反的吸引力:当代码很小时,突出的是人。

JavaScript 最知名的开发者以他们的演讲、他们录制的视频、他们写的推文和博客帖子而闻名。他们拥有大量的追随者,以一种 PHP 开发者做不到的方式吸引着热切的观众。例如,肯特·C. 多兹(Kent C. Dodds)花了大量时间制作关于 React 的内容,以至于他辞去了 PayPal 的 JavaScript 工程师工作,成为了一名全职教育家。^[45]

JavaScript 开发者不再总是因为特定的项目而出名,就像斯托曼因为 GNU 而出名,托瓦兹因为 Git 和 Linux 而出名一样。与其只和一个或几个项目相关联,杰出的 JavaScript 开发者通常创建数百个小型但广泛使用的项目。人们知道他们是谁,而不是他们参与了什么项目。

安东尼·凯宾斯基(Antoni Kepinski)是一名十几岁的开发者,他开发了一个开源的披萨跟踪应用程序。当一位采访者问他是如何进入编程行业的时候,他回答说:

事实上,我开始用 C# 编程,但这不是我真正喜欢的东西,我偶然发现了一个 GitHub 用户辛德勒·索尔许斯的个人资料。当我第一次看到 JavaScript 代码时,我就知道这是我将来想要学习的东西。事情就是这样开始的。^[46]

在随后的采访中,他开玩笑说,他使用了索尔许斯的一个项目,而不是另一个 JavaScript 开发者费罗斯·阿布哈迪耶(Feross Aboukhadijeh)的项目:

凯宾斯基: 对不起,我没有使用你的 linter(一种捕获代码中格式错误的工具)。

阿布哈迪耶: 是的,我注意到了。你用的是辛德勒·索尔许斯的,因为我觉得你更喜欢他。这很好。(笑声)

凯宾斯基: 不,不……

罗杰斯: 就是这样。你跟随你最喜欢的开发者,“我想要他们的风格指南”。就这么简单。

对个性的崇拜在 JavaScript 中盛行,即使最著名的开发者不愿承认这一点,也许是因为这种态度与公开宣称的开源“社区构建”的理想相冲突。与早期著名的黑客托瓦兹、雷蒙德和斯托曼相比,今天的许多 JavaScript 开发者都非常谦逊。

举几个例子: 流行前端框架 React 的维护者丹·阿布拉莫夫(Dan Abramov)写了一篇博文,列出了所有“人们常常错误地认为我知道”的编程主题。^[47] Babel 是一个帮助 JavaScript 在不同版本间进行转换的库,它的维护者亨利·朱(Henry Zhu)说,开源帮助他“在我的耐心和谦逊中接受测试,并学会更好地专注和沟通”。^[48] 另一个流行的前端框架 Vue 的维护者莎拉·德拉斯纳(Sarah Drasner)和肯特·C. 道兹(Kent C. Dodds)共同编写了一个“开源礼仪”,在其中,他们强调贡献者应该“礼貌、尊重和善良”。^[49] 辛德勒·索尔许斯曾经在 Patreon 网页(一款付费创作社区)上为他的创作募捐,主页显示他抱着三只快乐的哈士奇;^[50] 费罗斯·阿布哈迪耶的 Patreon 页面显示了他和一只柔软毛茸茸的小猫依偎在一起的画面。^[51] 这些开发者中的每一个都拥有大量的用户,这些用户都追随他们个人;他们拥有成千上万开发者的关注,他们选择利用这种影响力来展示善意和尊重。

每个 JavaScript 开发者都是圣人吗?当然不是。和其他开源社区一样,



莱纳斯·托瓦兹和辛德勒·索尔许斯
(经阿尔托大学和辛德勒·索尔许斯许可使用的图像)

JavaScript 在 Twitter 和 GitHub 上也经历了激烈的争论和个人攻击。但是值得注意的是，那些不参与这些事情的杰出开发者，似乎都在回避自己的名人身份。

2019 年，JavaScript 社区的两个子集 Vue 和 React 之间爆发了戏剧性事件。丹·阿布拉莫夫暂时关闭了自己的 Twitter 账户，后来解释说：“这个周末我变得越来越焦虑。在这种状态下我对社区毫无用处。我现在感觉好多了，我来这里是为了听取大家的意见的。”^[52]

开源文化的重心已经从推崇强权专制转向寻求深思熟虑和管理。就连莱纳斯·托瓦兹也在 2018 年发布了一份道歉声明，他离开 Linux 项目一个月，是为了反思多年来“我们社区的人们面对我这一辈子不理解情感的经历”，并称之为“照镜子”的时刻。^[53] ESR 被禁止进入 OSI 的邮件列表，因为他的言辞好斗。^[54] RMS 在麻省理工学院计算机科学和人工智能实验室 (MIT's Computer Science and Artificial Intelligence Laboratory, CSAIL) 的邮件列表上发表了有争议的言论后，辞去了他在麻省理工学院和自由软件基金会的职务。^[55] 在这个问题上，双方的许多人都注意到斯托曼的评论——他在评论中用他剖析“自由软件”这个词的同样迂烂的方式剖析了“强奸”的定义——从他过去的行为来看并没有什么异常，但这是个在变化的时代。

这种善良的倾向也会导致平台的分销能力产生紧张。由于它的流行, JavaScript 吸引了没有经验的开发者,他们第一次学习如何编写代码,也不熟悉如何与开放源码项目交互。因此,在 GitHub 上,JavaScript 维护者和用户之间的交互可能很难管理。因为 GitHub 很容易使用,所以打开一个问题或提出问题的障碍很低,用户向维护者寻求一切帮助:如何解决合并冲突,如何编写测试。这些技能并不是特定于任何一个开源项目;它们都是关于“我如何开源”的问题。

负责维护数百个 JavaScript 项目的辛德勒·索尔许斯曾在 Twitter 上写道:“在 GitHub 上审核了 1 万多个 PR 之后……大约 80% 的贡献者不知道如何解决合并冲突。”^[56]几年后,他又进行了更多的观察:

- 几乎没人能写出一个好的 PR 的标题
- 有一半的人不知道‘Fixes #112’的语法
- 约 30% 的人在提交 PR 之前不会本地跑测试
- 约 40% 的人不会包含文档/测试^[57]

我还注意到,在过去几年里,整体 PR 质量大幅下降。我猜这是 GitHub 越来越受欢迎的结果。^[58]

这不是新开发者的错。他们不知道开源是如何工作的,他们被告知提出问题是做正确的事情。“但不要让这成为你贡献的阻碍,”索尔许斯自己补充说,^[59]“你要记住,我的时间是有限的,如果我不得不在你的 PR 中来回奔波,你可能会发现自己会再看一遍,这就占用了你从其他 PR 中抽出的时间。”^[60]

开源维护者实际上已经成为了学习如何贡献的开发者的老师。在过去,当新开发者试图加入项目社区时,这么做是很有意义的。时至今日,仍向陌生人不断重温这些基本知识只会令人疲惫不堪:被纸划伤一千次也会让人丧命。开发者诺兰·劳森(Nolan Lawson)将自己的经历描述为“一种反常效应,即你越成功,你就越会受到 GitHub 通知的‘惩罚’”。^[61]

GitHub 的一代

开源是建立在开发者不应受制于特定平台的理念之上的。与其他类型的开

发者相比,开源开发者似乎更应该不受平台效应的影响。毕竟,在 GitHub 之前有整整两代自由和开源的开发者,他们使用的工具在没有 GitHub 的情况下也能正常工作,而且有技术娴熟的用户,他们对不方便但符合价值观的解决方案有很高的容忍能力。

然而,GitHub 继续占据主导地位。就连 1999 年成立的保护伞型组织 Apache 软件基金会(Apache Software Foundation)也在 2019 年宣布,它将把基于 Git 的项目迁移到 GitHub。^[62]人们一度认为,Apache 软件基金会不愿采用现代开源工具。^[63]

人们可以哀叹真正的“开放”软件开发的死亡,就像一些反对者仍然在做的那样,并批评 GitHub“破坏”了开源的发展轨迹。但 GitHub 克服了最初的宗教狂热,成为了大多数开发者的首选平台,这表明平台为开发者所做的事情可能比我们意识到的要多。我们不应该将平台视为对手,而应该将平台视为创造者的强大盟友,并尝试着去理解它们之间所形成的奇怪的共生关系。这种紧密的联系不可避免地会滋生强烈的情感和冲突,但也许这是它们变得不可或缺的标志。

GitHub 的优点——易于使用、易于共享和发现他人的代码——也是当今开源领域面临挑战的根源。对于在容易吸引新手开发者的 JavaScript 中,这些挑战显得尤为突出。

考虑一下这样一个事实:JavaScript 项目是特意设计成小型和模块化的,而不是大型的社区项目,有长期成员,可以接纳新人。然后再考虑一下 JavaScript 的领导者们不愿表现得粗鲁或不受欢迎,他们对斯托曼-托瓦兹-雷蒙德的散伙黑客中那些挥舞着鸟、拿着枪的幽灵们置之不理。

总而言之,在一个鼓励新手参与并且在社交规范中明确抵制排斥新人的平台上,即便大部分参与是出于好意的,但这些“路过”的用户确实淹没了项目维护者的所有注意力。

就像现在的其他社交平台一样,GitHub 专为大规模分解的用例而设计。每个平台都必须找出如何适应一套尚未被很好地理解的新兴社会行为。

* 我的范围一般也仅限于美国、欧洲和澳大利亚。开源开发正在这些地区之外发展,特别是在中国和印度,但由于他们的行为在某些方面不同,我将我的重点限制在我可以更自信地与之交谈的地理区域。^[14]



[†]在过去，托瓦兹拒绝接受从 GitHub 上拉到 Linux 内核的请求，称它“很适合托管，但是 PR 和在线提交编辑，都是纯粹的垃圾”。^[18]

[‡]尽管“自由”和“开源”软件的定义，甚至根据斯托曼自己的说法，本质上是相同的，但这两个概念在文化上是不同的。有些人，特别是自由软件开发者，不喜欢混合使用这些术语。自由软件的提倡者因意识形态而团结在一起，这是一场“自由和正义的运动”。他们认为代码应该从私有控制中解放出来。另一方面，早期的开源倡导者专注于实用的目标，比如标准许可，使代码更容易被任何人自由发布和使用，包括商业实体。^[22]

[§]福格尔是一个有思想的，在我看来被低估了的早期开源文化的代言人。他写了迄今为止唯一一本关于开源产品的著名著作《生产开源软件：如何运行一个成功的自由软件项目》(*Producing Open Source Software: How to Run a Successful Free Software Project*)，该书于 2005 年首次出版。你可以在 <https://producingoss.com/> 上找到它的全部内容。

[¶]根据 Stack Overflow 2018 年的开发者调查，87% 的受访者使用 Git。

^{**}来自碧昂斯，她在歌曲《Nice》中说道：“耐心等待我的死亡/因为我的成功无法被量化/如果我对流媒体流量有多在乎/就会把《Lemonade》放到 Spotify 上。”^[38]

02

开源项目的结构

在保护生物学中，“有魅力的巨型动物”这一术语是指北极熊比起软体动物或昆虫更能推动环境保护事业。越可爱越好。在创作者的世界里，充满了名副其实的魅力巨型动物，如 YouTube 喜剧演员和 Instagram 模特，而我选择淡水贻贝。^{*}（译者注：对作者来说，淡水贻贝更能激起环境保护欲望）

开源是复杂的，因为它包含了技术和社会规范的混乱组合，其中大部分是在公开场合发生的。它被广泛地记录下来（几乎任何一条讨论都在某个地方被记录下来），但又不明确（你需要从经年累月的存档中发掘你所需要的）。它的宝藏隐藏在荆棘丛生的环境中。

社会规范是通过试验和错误传下来的，这意味着如果做错了什么，就有可能在同行面前遭遇尴尬和嘲弄。开发者并不是因为缺乏技术能力，而是害怕犯错，才不对开源项目做出贡献。

以代码和文字著称的基础设施开发者朱莉娅·埃文斯（Julia Evans）描述了她在开源贡献方面的经验：

我有时对开源感到有点畏惧，因为在开源项目中，我需要把代码发给完全陌生的人评审。在工作中，我一般把代码评审发给同样的 10 个人左右，他们中的大多数人已经和我一起工作了一年或更久，而且他们往往已经清楚地知道我在做什么。^[64]

更具挑战性的是，这些技术和社会规范在不同的语言生态系统、不同的项目类型，甚至不同的开发者之间都是不同的。作为一个 Python 开发者，知道如何参与开源，并不意味着你可以大摇大摆地进入 Haskell 的世界。一个 C++ 的开发者可能会在一群 Electron 的开发者中感到格格不入。

开源的部落性质经常被忽视和误解。正如我们所看到的，从“黑客”到更中立的“程序员”或“工程师”，再到今天听起来很光鲜的“开发者”，其实没有什么“开源社区”，就像没有“城市社区”一样。当然，一个城市居民可以在许多方面同情另一个城市居民，特别是当他们与郊区或农村的同行进行比较时。但了解旧金山的街道并不能说明你在香港会有多好。“城市”是一个形容词，不是一个

终点。

把这个比喻进一步延伸：有这种城市，也有那种城市。[†] 雷克雅未克（Reykjavik）是冰岛最大的城市，大约有 20 万居民。^[66] 但在中国，它几乎不可能成为一个城市，中国最大的城市上海，有超过 2400 万居民。^[67] 同样地，有一些开源项目，像 chalk——一种用颜色和文本格式为自己的代码定型的工具，它只包含几行代码，执行一个小而有用的功能。这种项目包含很少的代码行，并执行一个小但有用的功能。也有一些开源项目，如 OpenStack——一个拥有数百万行代码的云计算软件平台，被分解成多个大型子项目，每一个子项目都使一些整个独立模块的代码库相形见绌。

术语“开源”只描述了代码的分发和消费方式。它没有说到代码是如何产生的。“开源项目”与“公司”一样，彼此之间没有更多的共同点。根据定义，所有的公司都会生产一些有价值的东西来换取金钱，但我们不认为每个公司都有相同的商业模式。[‡]

本书并不假定你是一个开发者，但它确实假定你不畏惧学习。对于那些不太熟悉开源软件的人，我将简要介绍一下这些项目的结构，这将有助于解码其中的人际关系动态。

贡献是怎么完成的

开源软件经常被描述为参与性的，这意味着任何人都可以修改其代码。虽然这在理论上是正确的，但在实践中，开源项目并不是被盲目地开放给每一个想改变它的人。（相比之下，例如维基，通常可由公众编辑，不需要额外的权限）

任何人都可以以“补丁”的形式向开源项目提出修改建议，或者用 GitHub 的术语来说，就是提一个 PR——但这些修改都要经过评审，并要得到事先信任的贡献者的批准。这相当于在一个共享文件上有评论权限：任何人都可以提出修改建议，但不是每个人都能真正批准它。

有些开发者有权限将修改合并到主干（或者 master 分支）上，这是项目的基线版本。这种权限通常被称为提交（commit）权限，这就像一部分人有权限编辑一个共享文件。

提交权限是一种技术许可,但也有社会考量。即使是那些拥有提交权限的人,也不能单方面地行使他们的权力。在他们合并一个改动之前,他们还必须考虑其他贡献者和用户会如何接受它。

更大的项目通常使用正式的“征求意见”(request for comments, RFC)过程,以允许社区在合并之前讨论这些变化。例如,在 Python 中,这些请求被称为 Python 增强提案(Python Enhancement Proposals, PEPs),^[68]而在另一种编程语言 Go 中,正式的提案被称为“设计文件”(design document)。^[69]在较小的项目中,RFC 过程可能看起来就像一个关于 PR 的非正式讨论过程。

相反,也有一些维护者在社会关系上被视为领导者,对项目有影响力,但同样没有提交权限。在一些项目中,无论项目规模有多大,除了作者,没有人有提交权限。例如,亚历克斯·米勒(Alex Miller)是编程语言 Clojure 的长期维护者,但他并不合并补丁,而是对来自社区的补丁进行分流和升级,然后由几个有提交权限的维护者进行评审和合并,主要是里奇·希基(Rich Hickey)(Clojure 的作者和主要开发者)和另一个共同维护者斯图尔特·哈洛威(Stuart Halloway)。

布雷特·坎农回忆他获得 Python 的提交权限的经历:

当我说只要有人提交补丁,我就很乐意写一些文档时,我已经定期提交补丁几个月了。吉多(Python 的作者)回答说:“你有一个 SF 的用户 ID 吗?这样我们就可以给你提交权限了!”我获得提交权限的方式和今天人们需要对 Python 项目做的是相同的:不断地贡献出好的补丁,最终一个核心开发者注意到了我,并问我是否愿意加入这个团队。^[70]

开发者获得 commit 权限的过程在不同的项目中差异很大,并受制于预先存在的社会规范。一些项目的理念模式是你需要预先证明你是值得信任的,而另一些项目则喜欢在低信任的基础上运作。

例如,基于 Linux 的操作系统 Debian,要求开发者遵循一个规范的进阶过程,在这个过程中,他们需要阅读手册,找到导师,并与项目维护者接触。^[71]另一方面,在 JavaScript 的开发者中,普遍存在着更自由的代码提交权限的情况。这

样做的目的是通过让其他人更容易做出贡献来分散维护的负担,而且在没有得到证明之前,人们会提前假定陌生人是值得信任的。

这些社会规范的差异往往与技术设计紧密相联。Clojure 的核心开发者高度重视稳定的代码,这意味着他们更不愿意接受修改。^[72]而 Debian 有一个单体的、紧密耦合的代码库,对错误的维护者开绿灯确实会带来可怕的后果。但包括 Node.js 在内的 JavaScript 的设计是模块化的,每个维护者影响生态系统其他组件的能力有限,所以 JavaScript 的开发者更有可能优先考虑快速行动和接受贡献。

在加密货币领域,这些理念在比特币和以太坊项目的管理方式上表现出明显的差异。比特币的社区,就像 Clojure 的社区一样,优先考虑稳定性和安全性,倾向于缓慢而谨慎地前进,即使这意味着引入较少的功能和贡献者。以太坊更像 Node.js: 它是一个供他人开发的平台,覆盖的范围很广。它就如同一个洛杉矶般蔓延的城市,由许多社区和亚文化组成。尽管有慷慨激昂的咆哮与反对(我学到的一件事情就是开发者是有个人想法的),但没有一个绝对正确的处理事情的方式,只有不同的社区,每个社区都有自己的文化规范。

获得变更批准的过程,一般来说,取决于变更的复杂性(和项目的复杂性),以及一个人在有能力批准变更的人中的声誉。

如果开发者提出的变更与项目的目标相一致会更有帮助,比如,开发者可能实现了以前公开讨论过的愿望清单上的功能,或者修复了项目上的一个已知的错误。这些变化往往比全新的想法更容易得到批准,因为这些是显式的需求。

dkubb commented on Apr 18, 2016

Collaborator + ⚡ ...

@backus this is awesome! Feel free to ping me for reviews about this anytime; this is one of the features I've wanted most in mutant and I'd love to help our however I can.

I would say something like this should be merged early once you've got one or two basic mutations working reliable and @mbj has approved it. Your work provides an outline, and we can gradually fill in various other mutations over time.

一个 Ruby 测试工具的维护者对一个新的贡献者的回复^[73]

开发者的声誉会严重地影响到一个 PR 是否能被合并。声望并不总是局限于特定的项目,也包括更广泛的生态系统。(反之同理:在其他地方的坏名声可以带到一个项目中,使贡献更难被合并,即使这个人以前从未为这个特定的项目做过贡献)

例如,洛伦佐·西德拉(Lorenzo Sciandra)首先为 React Navigation(一个 React Native 相关的库)做了贡献,从而成为 React Native(一个用 React 构建移动应用的框架)的维护者。^[74]当时,他觉得自己还“不够格”为 React Native 做贡献,但当 React Navigation 的维护者离开项目后,剩下的开发者就积极招募更多的贡献者。

西德拉在写代码上不够自信,所以他专注于解决 issue,在四个月内关闭了 900 多个 issue。他的活动引起了在 Facebook 工作的 React Native 维护者赫克托(Héctor)的注意,他问西德拉,基于他的 React Navigation 经验,他是否愿意为该项目做出贡献。

如果有 commit 权限的人不能保证评审和接受开发者的 PR,这可能会导致管理纠纷。例如,一家公司可能会发布开源代码,但主要依靠自己的员工来维护它。虽然该项目是开源的,任何人都可以使用、检查、分叉和修改代码,但作为非雇员,可能很难做出实质性的贡献。任何人都可以提交修改,但并不意味着它会被批准。

虽然当企业开源者参与其中时,治理纠纷可能看起来更明显,但这些问题也发生在单一维护者的项目中。Setuptools 是一个广泛使用的 Python 软件包,主要由其作者维护。在许多用户抱怨他们的贡献很难被合并到 Setuptools 中后,一群开发者将该项目分叉成他们自己的版本,命名为 Distribute,几年后,该版本最终被合并回 Setuptools 中。

如果一个开发者在项目中不为人知,他们必须努力争取维护者的注意。礼貌的做法是将自己的互动信息包含在具体的 PR 中。也可以@(即通知)相关人员要求评审——就像对一周(而不是一天)没有被回复的沉寂邮件进行跟进。通过个人渠道(如私人电子邮件)进行联系,通常被认为是不礼貌的,就像在推特上要求别人回复你的电子邮件被认为是不礼貌的一样,除非另有规定。

例如,马吉特·拉马努贾姆(Amjith Ramanujam)在他的项目 pgcli(一个开源数据库系统 PostgreSQL 的命令行接口)的 README 中列出了他的电子邮件

和 Twitter 账号，并补充说贡献者应该“随时与我联系，如果需要帮助的话”。^[75]

相比之下，WP-CLI，一个流行的内容管理系统 WordPress 的命令行接口，在其 README 中的要求完全相反：

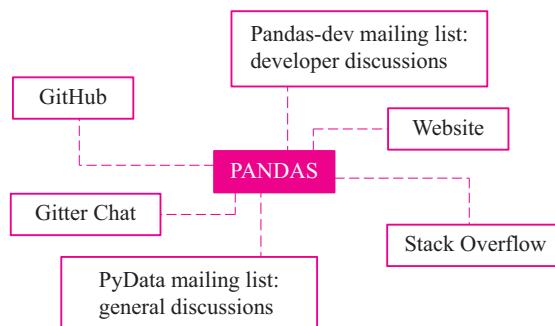
请不要在 Twitter 上提出支持问题。Twitter 不是一个可接受的支持场所，因为：(a) 很难在 280 个字符的限制下进行对话；(b) Twitter 不是一个可以让和有你同样问题的人在之前的对话中搜索到答案的地方。^[76]

根据项目的情况、复杂程度和受欢迎程度，以及一个人提出的改变和人们的关注程度，PR 可能需要几分钟到几个月的时间才能得到回应。有时，在一小时内就被合并了；而有时，根本就没有被评审。

互动发生的地方

开源项目被称为“项目”，而不仅仅是代码，这是有原因的。虽然代码是一个项目的最终产出，但“项目”一词指的是整个社区、代码以及支持其基本生产的通信和开发者工具。

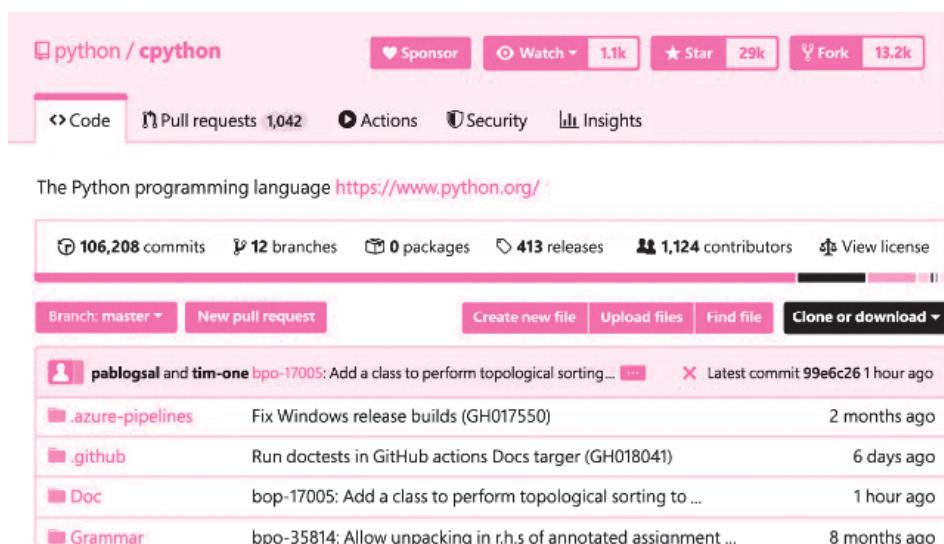
术语“项目”和“仓库”有时可以互换使用，而且仓库经常作为项目的主要命名空间。但仓库特指包含代码的文件目录，而项目则意味着支持代码生产的全套工具和交流渠道（例如，邮件列表、聊天、文档和 Q&A 网站）。



Pandas，一个用于数据分析的 Python 的框架，扩展到多个交流渠道

仓库是一个单一的衡量单位,不同仓库在颗粒度方面有很大的不同。Isorted,一个检查数组是否被排序的模块,包含少于 30 行的代码,是一个仓库。^[77]CPython 包含 Python 编程语言的全部代码库,也是一个仓库。^[78]

就像其他事情一样,版本库的大小和结构受到技术决策和个人喜好的影响。有些项目习惯把他们的代码分成许多库,每个库都包含少量的代码,这些仓库都在一个 GitHub 组织下。其他项目则采用单库理念,在一个库里管理大量的代码。



CPython 的 Github 仓库^[79]

托管在 GitHub 上的开源项目一般可以分成三个部分:代码(项目的最终输出)、问题追踪器(讨论修改的方式)和 PR(进行修改的方式)。

代码通常使用版本控制系统来管理,其中 Git 是最流行的。这个系统直接与代码捆绑在一起,使用一个.git 文件目录存储代码修改记录,这意味着无论代码文件实际托管在哪里,无论是在 GitHub 还是其他地方,都会进行变更追踪。

另一方面,issue 和 PR 则绑定在 GitHub 上。尽管 issue(别称 ticket)和 PR(别称 patch)的概念比 GitHub 要早得多,但 issue 和 pull request 是 GitHub 对这些功能的扩展,因此在平台间迁移起来并不那么容易。问题追踪器是用来进行对话的,比如讨论新功能或报告错误。PR 是提议的修改,如果被批准和合并,

将修改最终的代码库。

issues 往往分为三类：错误报告、功能请求和问题。从维护者的角度来看，错误报告是最优先的，因为它们意味着某些东西目前没有按预期工作。功能请求是用户希望得到的东西。问题本质上是帮助请求，例如，“我怎么做 X”。有些维护者不使用问题追踪器来处理问题，而是倾向于将用户从 GitHub 引向其他支持渠道。

一些开源项目，尤其是较大的项目，可能使用 GitHub 来托管和管理他们的代码，但使用不同的工具来管理项目的其他方面。例如，Django 在 GitHub 上托管代码，但使用 Trac 作为其问题跟踪器。React 也将其代码托管在 GitHub 上，但将其文档、教程和社区放在一个单独的网站上。^[80]

项目经常使用同步通信工具（如 IRC、Slack 或 Discord）和异步工具（如邮件列表、Reddit 或 GitHub Issue）来与用户和其他贡献者交谈。这些工具被用来进行各种对话，包括：

- 在用户和贡献者之间提出和回答问题（“我如何使用项目来做 X”以及“我如何做出 Y 贡献”）。
- 协调维护者之间的工作。
- 提供一个社区空间。
- 召开办公会议。
- 教育用户（例如，教授或演示某种东西）。

Stack Overflow，一个面向开发者的 Q&A 网站，成为了 GitHub 的一个重要补充工具（尽管其有用性因编程语言或框架而异），因为它是用户经常提出问题并获得答案的地方；该网站有自己的社交动态和奖励制度。用户通过回答问题获得积分，从而建立起他们的公共声誉。Stack Overflow 上的顶级回答者可能永远不会与 GitHub 上的项目核心开发者互动，尽管他们为用户提供了大量支持；这两个平台是与同一个项目相关的独立生态系统。（与用户-用户之间的支持系统相关的话题将在后面更深入地介绍）

项目如何随着时间变化

一个项目的维护者和他们的社区之间的关系随项目的成熟度而变化。在一个较高的水平上,开源项目倾向于从封闭到开放、再到封闭的过程,或根据规模进行分布式开发。

创建

在一个项目的最初阶段,可能有一个或几个开发者在相当封闭的开发状态下编写代码。虽然有些开源开发者从一开始就公开写代码,但许多人更喜欢在私下里做最初的创造性工作,这样他们就可以在开放项目获得反馈之前适当地阐述他们的想法。即使开发者在早期就公布了代码,也可能不会广泛宣传,直到有了可以发布的东西。

创建了流行的 JavaScript 平台 Node.js 的瑞恩·达尔(Ryan Dahl)回忆他早期独自工作六个月的经历,以及最终在 JSConf EU 会议上做了第一个 demo 演示时说:

我辞去了工作,在 Node 上工作了六个月,基本上投入了所有时间。我坚信这将是一件伟大的工作。我给 JSConf 的人写了一封非常友好的信,恳求他们给我一个名额,让我在 JSConf EU 上展示它……我当时非常害怕,要把这个我已经研究了六个月的东西展示出来。^[81]

布道

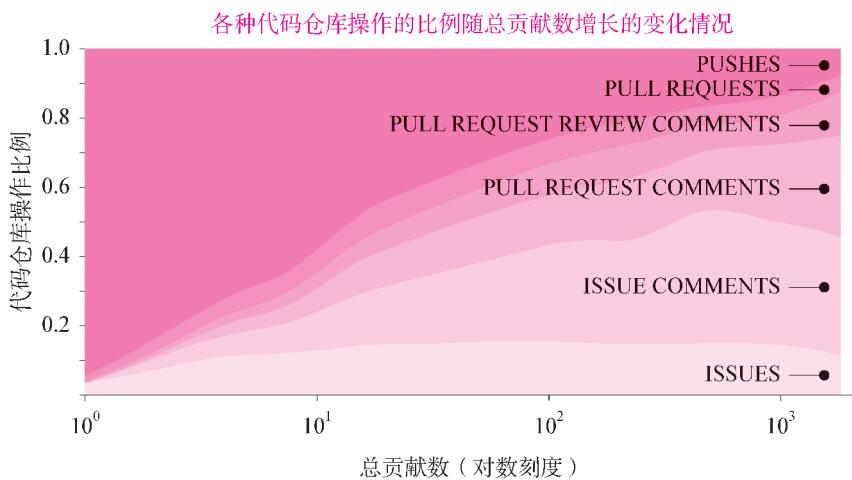
一旦项目发布,作者通常渴望得到反馈、错误报告、问题和 PR。他们通常会像创始人推广创业公司一样推广该项目:在网上的相关渠道分享该项目,在会议和聚会上发表演讲,并鼓励其他人撰写和谈论该项目。

编写数据计算系统 Apache Storm 的内森·马兹(Nathan Marz)在一个名

为“怪圈”(Strange Loop)的软件会议上发表演讲时，在讲台上公开了这个项目。这个消息引起了很多注意，他建立了一个邮件列表，让开发者提供反馈，而他会“每天花一到两个小时来回答问题”。^[82]在接下来的一年里，他继续在会议、聚会上和公司里发表关于 Storm 的演讲。他指出“所有这些演讲使 Storm 积累了越来越多的人气”。

在这个阶段，目标是“分发”——让其他开发者掌握新的代码——这样项目就会过渡到一个更开放的发展状态。就像在网上发表博客文章或视频一样，开源开发者希望他人有实质性的参与，所以他们会鼓励贡献，同时仍然保留他们自己对项目愿景的控制。如果其他开发者正在提交 issues，给项目加星，或是提出问题，这就表明人们对它感兴趣。

那些确实吸引了其他开发者注意的项目可能会稳定在这里，有一个即便不是快速增长的，但是却坚实的、会继续下载和使用该项目的用户群。然而，在一个较小的项目子集中，用户和贡献者的互动将继续上升，直到把项目推向成长阶段。



摘自《开源的形态》(*The Shape of Open Source*)，作者阿方·史密斯(Arfon Smith)。随着贡献者数量的增加，仓库的活动类型会发生变化，特别是评论的形式。^[83]

成长

当一个项目被广泛使用时，更多的开发者会与它互动。判断这种转变何时

发生的一个启发式方法是，维护者开始在项目上处理更多的非代码工作，而不是代码工作，例如分流问题和评审他人的 PR。

“增长”意味着有更多的开发者在使用这个项目，但这并不一定意味着有更多的开发者做出了有意义的贡献，也不意味着有更多的维护者来满足这一水平的需求（就像人们期望一个成功的公司会增加员工人数一样）。在这个阶段，社区互动变得更加嘈杂——评论、功能请求、来自陌生人的 PR——这使得维护者很难对每个人做出回应。

为移动开发者编写 fastlane 工具的费利克斯·克劳斯（Felix Krause）解释说：

项目规模越大，保持项目初期所提出的创新就越难，因为你可能需要在短时间里同时考虑上百种不同的用例……一旦你的产品获得了数千名活跃用户后，你会发现，帮助用户所花的时间比在项目本身上花费的时间更多。人们会提交各种各样的问题，其中大部分都不能称之为“问题”，只是与功能相关的要求和疑问。^[84]

维护者如何应对这些吸引注意力的需求，取决于贡献者的基数类型（以下两种方案将在第 5 章中作详细讨论）。如果一个项目没有很多贡献者，维护者就应当学会过滤杂音，进入到一个更封闭、更专注的开发状态之中。在封闭状态下，维护者对外部提交的代码采取更严格的筛选，以便能够专注于自己的工作。

如果一个项目的贡献者规模正在快速增长，并且有足够的工作需要分工完成，维护者就能更广泛地分发工作。在分布式开发状态下，维护者会积极招募更多的贡献者参与进来，目的是将他们留在项目中。在这种情况下，维护者会投入更多的时间来增加开发者的数量，以满足用户和贡献者的需求。

区分项目类型

随着项目的发展，它会获得更多的用户和贡献者。从“开源是重在参与”的

假设出发,我们可以想象“最佳”的用户-贡献者比例是 1 : 1,也就是每位用户同时又是贡献者。

事实上,一个项目的用户和贡献者规模常常以不同的速度增长,两者有时甚至彼此独立。一些项目似乎拥有大量用户却只有少量贡献者。另一些项目的贡献者社区十分活跃,但并没有得到广泛应用。是什么因素导致了这些差异?

项目贡献者的增长取决于技术范围、所需支持、参与门槛以及用户采用情况。

技术范围指项目代码库的规模和复杂度。换句话说,就是剩余多少事情要做。一个功能完备的项目不会像一个可扩展和可定制的项目那样吸引那么多的贡献者,这样的项目也许会被广泛应用,但这并不意味着会有大量的开发者为其做出贡献。

例如,React 是一个用于前端开发的 JavaScript 库,它可以被视为一个平台,因为它是许多其他应用程序的基础。Webpack,一个用于打包 JavaScript 文件的工具,就是这个“平台”的一个组件:它经常被 React 使用,但是它的适用范围更小。我们很容易想到,比起专门贡献 Webpack,开发者更倾向于定期为 React 生态做贡献。Webpack 由四位开发者共同维护[(其中仅有一位核心开发者托拜厄斯·科珀斯(Tobias Koppers)],^[85]而 React 由一个更大的团队维护,其中许多人在 Facebook 工作。

所需支持不单指为项目编写代码,还包含一些其他任务,如回答他人的提问或者评审别人提交的 PR。前者会被潜在的贡献者认为是“有趣”的工作并参与其中;后者虽然也是必要的工作,但往往只落在维护者身上。吸引一个新贡献者去实现一个有趣的新功能要比让他们去分流问题容易得多。

此外,如问题分流这样的任务需要你比普通的贡献者更熟悉这个项目。在一个项目中工作多年的人能够比一个新贡献者更快地识别重复的问题或常见的问题。

一个项目可能代码量并不大,但是需要大量的用户支持,反之亦然。比如Youtube-dl——一个能够从 YouTube 或其他视频网站上下载视频的程序,就是一个在技术层面十分“小”的项目,但它是 GitHub 上支持量最高的项目之一。^[86]著名的图标工具包 Font Awesome 也是一个从代码角度来看非技术密集的项

目,但它收到了许多来自用户的新图标需求和贡献。^[87]

参与门槛指对项目贡献的难易程度。一个项目是否容易做出贡献在很大程度上取决于它的技术范围,但还有一些其他因素:如文档的质量、维护者的积极程度、社交的友善程度,以及参与项目是否需要预先掌握某些工具或技能。最重要的问题是该项目是否在 GitHub 上。

对于 GitHub 和非 GitHub 的项目,对新贡献者的吸引力有着天壤之别。一位曾经在不同代码仓库工具上工作了数年的基础架构开发者告诉我,他现在已经过于习惯使用 GitHub,以至于当他在某个使用其他问题追踪器的项目中发现 bug 时,他甚至都不愿去提交问题——因为工作量太大了。

2015 年,Babel 的作者塞巴斯蒂安 · 麦肯齐 (Sebastian McKenzie) 在 Twitter 上抱怨称:“GitHub 在管理大型开源软件的表现太糟糕了,我经常感到不知所措,UI(用户界面)根本不能很好地放缩。”^[88]之后,他尝试将 Babel 转移到了 Phabricator。不到一年后,开发者们又将他们的问题追踪记录迁移回了 GitHub。也许 Phabricator 的某些工具更加强大,但是大部分人并不熟悉它。比起花功夫去熟悉一个新的问题追踪软件,Babel 的用户更倾向于在 Twitter、其他不相关的代码库或者评论区下提交他们的问题。当时,有 Babel 的维护者提到,“我们只有很少的长期贡献者,并且像其他大部分项目一样,维护者的人数也不多。如果打算扩大贡献者团队,我们至少得降低门槛,使用 GitHub 来开展工作”。^[89]

用户采用情况指的是一个项目的影响范围:贡献者的目标市场是什么?有多少人有潜在的可能来为项目做出贡献?开源项目的大多数贡献者都是从一名用户开始,因此分析用户采用情况是评估一个项目的潜在贡献者规模的启发式方法。

例如,尽管 Go 语言拥有一个强大而活跃的开发者社区,但它并不像 Python 那么应用广泛。Stack Overflow 在 2019 年的开发者调研中发现,41.7% 的受访者使用 Python,而使用 Go 的开发者仅占 8.2%。^[90]因此,我们会认为与 Go 相关项目的潜在贡献者总数比 Python 要少。

总的来说,这些因素可以帮助我们识别“大”或“小”的开源项目到底指什么。Bootstrap 在用户采用率上是一个“大”项目,而在技术范围上是一个“小”项目,

因此只有两个开发者(mdo 和 XhmiosR)负责其绝大部分的提交就不足为奇了。^[91]Bootstrap 确实在一段时间里拥有较大的提问量(或者需要的支持),所以另一个开发者 Johann-S 在回复 Bootstrap 的开放问题(open issues)时更加积极,而提交的代码量相对较少。

这些因素也会以非直观的方式影响项目。一个项目如何吸引新的贡献者和如何保留现有的贡献者是有区别的。例如,较高的准入门槛可能会使通过考验的开发者产生成就感并激发更大的兴趣。

奇怪的是,一些代码凌乱的老项目也能吸引小批十分敬业的贡献者。对于开发工具来说也一样:多位 Go 语言开发者曾向我承认比起 GitHub 他们更喜欢将代码提交至 Gerrit,因为这样能减少很多杂音。另一方面,用户采用情况较高的项目可能产生某种旁观者效应:没人愿意为项目做出贡献,因为他们都认为会有其他人来做这件事。虽然这些因素如何影响贡献者数量的增长取决于具体的项目内容,但关键在于它们确实会以某种方式影响项目的发展。

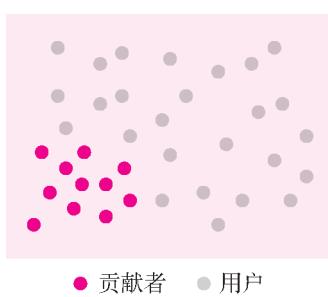
在关注贡献者与用户之间的关系时,我们可以根据贡献者增长和用户增长将所有项目分为四类生产模式:联邦模式、俱乐部模式、玩具模式和体育馆模式。

	高用户增长	低用户增长
高贡献者增长	联邦模式(如: Rust)	俱乐部模式(如: Astropy)
低贡献者增长	体育馆模式(如: Babel)	玩具模式(如: ssh-chat)

按用户和贡献者增长分类的各种类型的开源项目

联邦模式

联邦模式适用于具有高贡献者增长率和高用户增长率的项目。在想象一个具有该特点的开源项目时,我们通常会想到由埃里克·S. 雷蒙德首先提出的“集市”概念。这样的项目虽然不多,但极具影响力:正像大多数创业公司最终不会成为



Facebook 一样,大多数开源项目最终也不会成为 Linux。尽管联邦项目仅占开源项目的一小部分(根据一项调查表明不到 3%),但由于单个项目规模庞大,它们吸引了开源社区最多的注意力。^[92] Rust、Node.js 及 Linux 均是联邦模式的例子。

联邦模式类似于公司或者非政府组织。从治理的角度来看,它们的管理更加复杂,会倾向于通过设立投票、领导职位、基金会、工作组和技术委员会等方法来解决贡献者社区的协调问题。这些被选出的贡献者再去为更广大的被动用户群体做决策。

随着贡献者社区的发展,联邦通常会将贡献者拆分为更小的工作组,在这些工作组中,维护者专注于项目的某些领域,如基础建设或社区建设。联邦模式还会经常采用 RFC(评论请求),类似于投票倡议,来管理项目的主要变更提议。

然而,这些工作组可能仍旧会遇到与拆分之前同样的瓶颈问题,这些问题会被归咎于负责该部分的维护者领袖身上。因此一些联邦项目,例如 Node.js 也在尝试更自由的贡献策略,^[93] 开源开发者彼得·欣廷斯(Pieter Hintjens)曾称其为“乐观合并”策略。^[94] 这些维护者尝试更广泛地分配管理权限,而不是独自把关所有新提交的贡献,这么做能鼓励更多的人成为活跃的贡献者。比起复杂的管理制度对开发者热情的遏制,自由贡献策略给了开源项目更多拓展和成长的空间——如同一簇星星之火。

俱乐部模式

俱乐部模式指具有高贡献者增长率和低用户增长率的项目,这种发展趋势使得贡献者和用户群体大致重叠。虽然用户的总体数量较少,但他们更有可能



作为贡献者参与进来。例如,Astropy 是一个为天文学和天体物理学研究者提供核心功能和辅助的 Python 包。虽然大多数开发者不会用到 Astropy,但其相对狭窄的关注点使它更容易招募到贡献者,并使得那些依赖 Astropy 进行工作的人之间保持紧密联系。^[95]

Clojure、Haskell 和 Erlang 之类的编程语言,它们没有像 Java、C++ 和 Python 那样被广泛使用,但是这些语言在某些领域有其特长,无论是数学还是电信。(一位开发者曾经半开玩笑地告诉我,只要学会了 Haskell 就意味着你拥有了铁饭碗,因为对于招聘 Haskell 开发者的公司来说,能找到写 Haskell 的程序员就很不容易了)

俱乐部模式类似于聚会或者兴趣小组:它们只吸引一小部分用户,这些用户后来也成为了贡献者,因为它们对项目背景有较深的理解并且能在团队中感受到亲切感。俱乐部模式的受众可能不广,但它们是被一群爱好者所喜爱并建立的。

只要还有一小批开发者继续使用和贡献一个项目,这个项目就能无限期地存在下去,无论它能拥有多少用户。[§] 就像一些晦涩难懂的网络留言板,或是那些已经过气了的在线社区依旧能保持运营一样,俱乐部模式往往是稳定的,只要在老成员退出的同时还能有足够的新成员加入。

山下和弘(Kazuhiro Yamashita)等人用“磁性”和“粘性”来描述贡献者留存现象,这两个词语最早由 Pew 研究中心提出用于描述人口迁移趋势。^[96] 具有“磁性”的项目是指那些能够吸引大量新贡献者的项目,具有“粘性”的项目是指大部分贡献者正在持续做出贡献的项目。

成功的俱乐部项目具有高粘性,即使它们无法吸引到那么多的新贡献者,但它们依旧能够留住大量已经加入的贡献者。只要这些贡献者能留下来,并且俱乐部能够持续吸引足以保持项目活跃的新贡献者加入,这些项目就能持续地、自给自足地存在。

和联邦模式一样,俱乐部模式也需要吸引新成员,但它们在寻找新鲜血液的时候往往更有选择性,对参与者的综合素质往往有更高的期望。在俱乐部模式中,大部分用户都同时是贡献者:你要么加入,要么退出。因为俱乐部模式的社区规模要小很多,跟大多数用户并非开发者且工作量充足的联邦模式相比,每一位参与者对项目的影响要更大。这就和生活在小城镇和大城市的区别一样:在城市里,社区很容易分成更小的群体;但在小城镇里,大家都彼此了解,会更热衷于关心人与人之间的家长里短。

玩具模式



● 贡献者 ● 用户

玩具模式指贡献者增长率和用户增长率均较低的项目。就本书而言,它们可能是分析起来最无趣的开发模型,因为它们实际上就是个人项目。玩具项目可能就是一个业余项目或一份周末作业。也许在未来它们会被更多人使用,但在当前阶段,它们只是单个开发者为了好玩而随意摆弄的东西。例如,开发者安德烈·彼得罗夫(Andrey Petrov)做了一个名为 ssh-chat 的项目,这是一个允许用户通过 Secure Shell(SSH)协议聊天的客户端。尽管该项目在 GitHub 上获得了几千个星,但它只是一个有趣的实验,不需要太多的维护工作,也并不需要扩展它的用户量。^[98]

GitHub 上获得的星数小于 10 的项目也属于玩具项目(虽然星数不是衡量项目受欢迎程度的最佳标准,但这一指标能告诉我们至少有多少人看过这个项目)。这些项目可能有一个开源许可,但在当前阶段它们的作者并不期望能够获得许多贡献,他们也不认为有人在关注他们正在做什么。

体育馆模式

体育馆模式指具有低贡献者增长率和高用户增长率的项目。虽然他们可能偶尔会收到一些非长期开发者的贡献,但他们的长期贡献者数目并不随着用户数目同比例增长。因此,这样的项目往往只有一个或几个主力开发者。许多被广泛使用的包和库都适合这个模式,包括 webpack、Babel、Bundler 和 RSpec。如今,体育馆模式变得越来越普遍。

在体育馆模式中,一个或少数几个维护者代表更广泛的用户群体做出决策。不同于联邦模式或者俱乐部模式的社区是无中心的,体育馆模式的社区采用了



● 贡献者 ● 用户

一种中心化的结构,以其维护者为中心。无中心的社区以多对多的交流为主,而中心化的社区采用的是一对多的社交结构。

人类学家斯宾塞·希思·麦卡勒姆(Spencer Heath MacCallum)称这些为专有社区,在社区中由公认的所有者“将整个社区的每位成员串联起来”。^[99](文中提到的“所有者”并非是指其对整个社区拥有商业性质的所有权,而是指项目的所有者或者项目中大家公认的开发者,社区中的交流都由他/她来主持)酒店、商用飞机和家庭露营公园都是现实世界中专有社区的例子。这些社区中的客人、游客和居住者通过一位中心所有者——酒店拥有者、航空公司或者公园管理者来相互联系。

同样地,在网络世界,中心化的社区也由独立的创造者建立,这些创造者“为社区提供空间来‘促进’社区中的每一项活动进行”,因此他们的任务是维持整个社区的运作。^[100]社区之所以存在,是因为其创造者为人们提供了一个能够聚在一起的地方。一位电竞玩家是这样解释这种现象是如何在 Twitch 上发生的:“每位主播都有一个小社区,看他们的人互相喜欢,因为他们基本上都是被同一个人所吸引。”^[101]

与去中心化的社区相比,中心化的社区并不是特别依赖制定好的社区治理规章,因为社区中的互动行为主要发生在社区创建者和他们的用户之间,而不是长期贡献者之间。由创建者,而非他们的用户,来“对社区的基本经济结构负责”。^[102]



Rust 与 Clojure 前 100 位贡献者的代码提交次数分布^[103]

体育馆模式中,工作分配的不平衡一定程度上可以用供给侧的规模经济来解释。软件就像现实中的基础设施(道路、公用事业、电信)一样,其初始的固定生产成本高,而持续投入的边际成本低。换句话说,项目起步时成本很高,但每增加一个用户所需的成本相对较低。

电力公司的前期成本很高,但当基础设施架设完成后,每增加一个客户的服务成本就越低。因此,受规模经济的影响,这些类型的行业往往会展开垄断市场。集中承担高昂的固定成本的方法更经济,而且新入行者又难以达到这一门槛。

类似地,当我们在为一个开源项目提供劳动力时,增加新的维护者是“昂贵”的,因为维护者通常要具备一些需要一定学习成本的知识。所以新加入者倾向于做一些日常性质的贡献,而不是参与到更复杂的项目管理任务之中。

考虑到加入维护者团队所需的高学习成本,维护项目所需的知识往往集中在一个或几个人身上。如果不向他人传授这些知识,新人就难以参与进来。

一个开源项目在不同时期可能会在上述四种模式之间转换。随着用户增长率的变化,早期俱乐部模式的项目可能发展为联邦模式的项目;同样,玩具模式也有可能发展为体育馆模式。一些项目可能用户增长率较高,但由于不合理的贡献规则等因素,人为地降低了贡献者增长率。维护者可以通过简化贡献要求,或者缩小技术范围的方法来使他们的项目由体育馆模式转化为联邦模式。

那些去中心化的社区,如俱乐部模式和联邦模式的社区,具有高贡献者增长的潜力。这些项目的未来取决于新贡献者的招募,以及能否做到尽量减少不同贡献之间的冲突。早期为 Node.js 工作的迈克·罗杰斯在他的博客文章《健康的开源》中描述了这种模式:

制定(Node.js 的贡献)规则的目标是为了吸引新贡献者并尽可能地留住他们,然后再让这些贡献者去管理涌入的新入和对应的贡献,如此循环往复……

在制定决策时,要避免复杂的等级制度。相反,应当建立一个扁平化的、不断壮大的、拥有自主决策权的贡献者团队,这样才能在无干涉的情况下推动项目的发展。[\[104\]](#)



相比之下,中心化社区的运作则是基于有限的关注。作为社区的所有者,创造者必须亲自管理用户需求。因此,他们更倾向于依赖自动化、分布式的点对点支持,并且会更加激进地消除噪音。虽然所有的热门项目都能够使用这些方法,但它们是体育馆模式的必需品。

与俱乐部模式和联邦模式相比,体育馆模式的这一特点凸显了开源平台在实现中心化社区中的关键作用。去中心化的社区,无论是大型的在线论坛还是小型的群组聊天,总是难以明确自己的一亩三分地何在。曾经协助知名在线论坛“吓人玩意”(Something Awful, SA)运营的乔恩·亨德伦(Jon Hendren)回忆称,当他们网站的流量逐渐减少时,“我们中的大多数人都去了推特或其他网站”。^[105]但是,他还补充道:“某种意义上我们本身就是社区……大部分在 DM 组的人都是和我一起共事了 12 年的同仁,我们的社区并没有消亡,只是换了个地方。”

类似地,一个面向 40 岁以上女性的社区“What Would Virginia Woolf Do?”在转移到自己的平台之前,它的 Facebook 群组成员已经超过了 3 万。对于一个如此规模的社区来说,Facebook 更像是一个麻烦而不是一个赖以生存的平台。该组织的创始人妮娜·洛雷斯·柯林斯(Nina Lorez Collins)向记者解释说:“Facebook 是一个扁平化的平台,在 Facebook 上同时管理 38 个(子)群组(包括不同主题、地域和管理员)是一件十分棘手的事情,而 Facebook 官方没有给我们提供丝毫帮助。”^[106]柯林斯认为,“唯一的出路就是采取订阅模式并搭建一个自有 App,在那我们才终于能够……掌控属于我们自己的天地”。另一方面,创造者被平台所提供的支持所束缚,因为平台能大幅降低他们的成本,使他们比起单打独斗能够完成更多的事情。

这些项目模型之间的差异并不存在精确的定义。虽然在任何一个极端,我们都可以看到明显的差异:我们可以说 Linux——一个具有多个子项目和工作组的项目,看起来不像 tslib——一个用于编程语言 TypeScript 的助手库,但是中间的界限就有些模糊。有些项目有四五个维护者;另一些项目有很多的贡献者,但他们并没有深入参与到项目中。

不同的项目类型在如何有效地管理贡献者社区上有不同的特点。作家凯文·西姆勒(Kevin Simler)在研究开放社区和封闭社区之间的差异时提到:“如

果你正在筹建一个跳蚤市场,那么要求对所有买家和卖家进行背景调查或者需要推荐信才能加入,显然是无益的(更不用说这很荒唐)。但如果你经营的是钻石经销商,也许这些措施就是必要的。”^[107]在明确了这些极端情况后,我们才能更好地分析那些中间情况。

* 根据维基百科,“Ptychobranchus subtentum,又称槽形肾壳,是淡水贻贝的一种,属于河蚌科水生双壳类软体动物”。^[65]

[†]我的一位受访者把 Drupal 项目比作巴斯克人,以此来描述 Drupal 的文化和历史。像巴斯克的自治社会一样,Drupal 处于自己的领域中,与外部隔绝;由于项目不在 GitHub 上,Drupal 的开发者制定了他们自己的规范,这些规范受到主流开源实践经验的影响,但与其又有所差异。

[‡]感谢德文·祖格尔(Devon Zuegel)提的这一类比。

[§] 2019 年在 Gizmodo 发表的《Twitch 的温柔面》一文中,妮可·卡彭特(Nicole Carpenter)提到了一些 Twitch 主播,他们的观众规模较小,但观众的参与度很高。她采访的其中一位主播詹妮弗·钱伯斯(Jennifer Chambers),她给“虽然不多,但很忠实的观众”直播她的编织过程,“对于钱伯斯而言,在 Twitch 上直播并不像一个音乐家在体育场中为座无虚席的粉丝表演,而是像在一个编织同好会”。^[97]