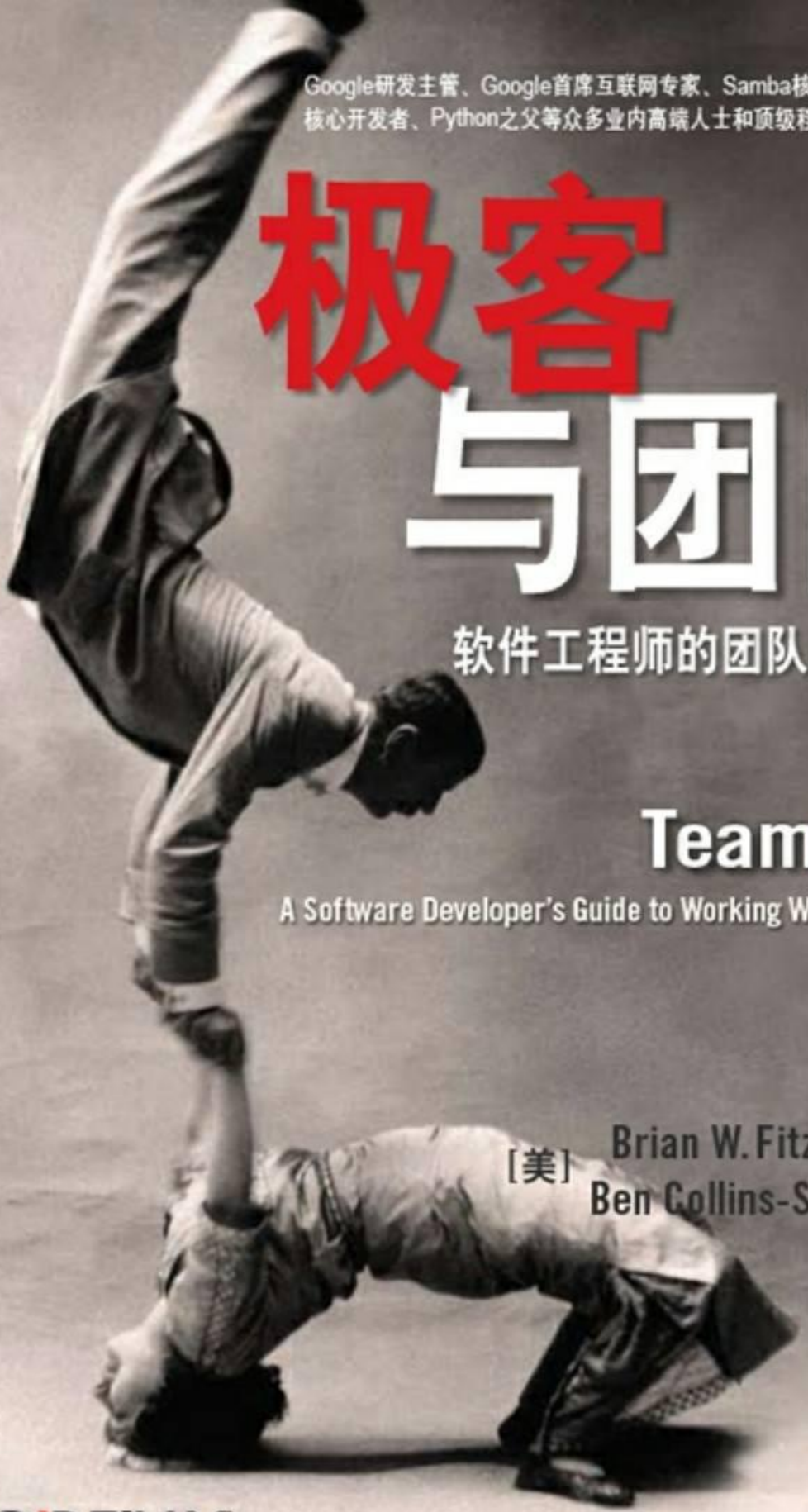
Google研发主管、Google首席互联网专家、Samba核心开发者、Django核心开发者、Python之父等众多业内高端人士和顶级程序员力荐

极客与团队

软件工程师的团队生存秘笈



TeamGeek

A Software Developer's Guide to Working Well with Others

[美] Brian W. Fitzpatrick 著
Ben Collins-Sussman
徐旭铭 译

O'REILLY

人民邮电出版社
POSTS & TELECOM PRESS

极客与团队

[美] Brian W.Fitzpatrick Ben Collins-Sussman 著
徐旭铭 译

人民邮电出版社

北京

内容提要

软件开发是一项团队运动，人的因素对结果的影响完全不亚于技术因素。正如本书作者所说的，一个项目成功的关键不仅仅是写出漂亮的代码：团队中的所有人朝着同一个目标一起合作也是同样重要的。

本书是一本写给程序员看的，教你怎么交朋友，怎么影响团队中的其他人。书中充满了操作性极强的建议和意见，让你在技术团队中过得更开心，变得更有效率，更加如鱼得水。本书旨在帮助程序员改进理解他人，与人沟通，以及与人合作的能力，进而在编写软件的过程中变得更有效率。

本书的两位作者来自Google，具有丰富的项目开发和管理经验，还曾经是Subversion的初创成员。本书得到Google研发主管、Google首席互联网专家、Samba的开发者、Django创始人、Python之父等众多业内高端人士和顶级程序员的好评。

本书适合那些想要更上一层楼并编写出色软件的程序员阅读，也适合软件项目和团队的管理者自己或组织团队成员阅读和参考。

致谢

尽管封面上只有两个人的名字，但是这本书是我们的整个职业生涯以来和成百上千人交谈后的结晶。因此我们想要在此对本书作出贡献的人们表示一下感谢（书中若有任何谬误，都是我俩造成的）。

感谢奥莱利公司的诸位同仁：封面设计师艾迪·弗里德曼，以及我们大无畏的编辑玛丽·特莱斯勒——没有玛丽的鼓励、耐心和时不时的鞭策，就不可能有这本书。

感谢sunnibrown.com的桑尼·布朗和安博·刘易斯用那么愉悦的方式让整本书变得鲜活起来，与你们合作真是太愉快了。

感谢本书的技术审校，你们贡献了无数的建议、想法，还有修改意见，才使得这本书变得更加完整：达斯汀·伯斯维尔、特拉佛·佛切尔、迈克·亨格、乔纳森·勒布朗、蓝俊彪和杰克·韦尔奇。感谢我们的朋友和同事在写书过程中帮忙发现的很多可笑的错误：戴夫·鲍姆、马特·卡兹、威尔·罗宾逊、和比尔·杜安。感谢听我们唠叨，还耐心给出建议的朋友，你们实在是太棒了：傅凯、吉姆·布兰迪、马特·布雷斯维特、丹尼·伯林和克里斯·德波纳。另外，还要感谢以下这些朋友提供的想法和建议：琳达·斯通、德维特·克林顿、布鲁斯·约翰逊、罗兰德·麦克格雷斯和艾米

特·佩特尔。

感谢 Google，特别是 Google 芝加哥的工程师们，感谢你们的支持、想法和建议，只是能和这样一支梦幻团队一起工作就值得令人心怀感激了。

特别感谢我们的老领导和导师：比尔·库格伦、史提夫·温特、阿兰·尤斯德斯、斯图·费尔德曼和埃里克·施密特。我们已经很努力地把从你们那里汲取而来的智慧写到这本书里了。

特别感谢布莱恩·罗宾逊和伊芬·埃里森·桑德拉给我们的悉心指导和照顾。

感谢 Apache 软件基金会接纳我们，也感谢你们对软件社区以及合作所作出的贡献。

感谢所有好朋友，让我们变得更加充实。别那样看着我们——说的就是你。

本书的大部分内容都是在芝加哥（也就是我们居住的城市）的Filter Cafe里构思创作出来的，这是一家舒适宜人的邻家咖啡馆。

傅攀勃的致谢

感谢妻子玛丽的巨大耐心、理解和鼓励——你的洞察力和同情心让我获益良多。感谢我母亲永不停歇的支持和热情。特别感谢我的岳母丽塔·冈穆勒作的那个“人是植物”的比喻。

写给本的话：认识14年，一起换了三次工作，写了三本书，你还没有厌倦我啊？感谢你陪

我经历这段疯狂、奇怪但是了不起的旅程——你是我的良师益友。我已经等不及看看我们接下来能折腾出什么东西来了（当然，先让我休息几个月）。

最后，感谢查理·麦克甘农先生在11年级的英语课上那么努力地教我写作。那时候的我觉得为了写一篇文章打四遍草稿是荒唐可笑的事情，现在我才知道自己当年的想法是多么荒唐可笑——好文章真的是改出来的。在此，向“四角和中间写作法”以及牛津英文字典致敬！

本的致谢

语言已经不足以表达我对妻子佛朗西斯的感激之情，她给了我巨大的空间——我不单指在撰写本书的过程中，还有过去几年来我参与的很多其他创作。没有她默默坚定的支持，我将一事无成。

写给傅攀勃的话：看到我们写给对方的寄语后，我要说，我们真像是老夫老妻。我以前从来没有意识到和人一起做演讲是这么有趣的事情，更别提一起编程和写书了。感谢上苍给了我们机会一起经历那么多事情！感谢你教了我这么多。

最后，感谢所有疯狂的硅谷人和企业：要不是你们引我入门，我是绝对体验不到这些疯狂经历的。

作者简介

布莱恩·傅攀勃现在是Google数据解放阵线和透明度工程两个团队的负责人，之前他还领导过Google的项目托管团队以及电子商务伙伴团队。他帮忙建立了Google芝加哥分部，并且为Google在开放数据上提出了很多想法和建议。

本·科林斯-萨斯曼是Subversion的初创成员之一，领导过Google 的项目托管团队，现在是Google 电子商务伙伴团队的负责人。他帮忙建立了 Google 芝加哥分部，并将 Subversion移植到Google的BigTable平台上。

本书宗旨

本书旨在帮助程序员改进理解他人、与人沟通，以及与人合作的能力，进而使其在编写软件的过程中变得更有效率。

对本书的赞誉

“精彩至极，直指人心！哪怕你不觉得自己是极客，它提供的建议也值得一读。”

——文顿·瑟夫，Google首席互联网专家

“我和工程师已经打了三十几年交道了。经验告诉我，人际关系在工作中的重要性并不亚于科学技术，但很多工程师都不愿意尝试了解如何与人合作。如果你想要更有效地进行创新，那绝不能错过本书。”

——狄恩·卡门，DEKA研发公司创始人

“作者为软件开发团队搜集了一套令人惊艳的模式和反模式。无论你是开发人员本身还是经理，或是任何有一点关系的人，只要你觉得自己难以理解为什么这些东西能让团队更有活力，你就应该读一读这本书。它道出了很多出色的开源项目程序员与生俱来的特质。我当初要是有这么一本书就好了。”

——布莱恩·贝伦多夫，世界经济论坛CTO

“软件开发是一项团队运动。如果你想要扬名立万，市面上有无数本教你如何磨练技术水平的好书，教你当好经理的书也不少。但这本书却另辟蹊径，教你应该如何与团队合作，以及如何当好合作伙伴。这个领域早就需要这样一本书了。”

——彼得·诺维格，Google研发主管

“如果你想要组建一支能专注开发优秀软件的团队，那绝不可错过本书。作者将谦虚、尊重和信任等感性的话题漂亮地转变成极富技巧的建议，哪怕是最挑剔的工程师也会欣然接受的。”

——埃里克·伦特，BrightTag联合创始人兼CTO

“这本书太精彩了。它探讨的是计算机编程里最难的问题，怎样和其他程序员打交道。我打算让Samba团队人手一本。”

——杰瑞米·埃里森，Samba作者之一

“你可能已经听过所谓的‘十倍程序员’传说了吧，它的意思是顶尖程序员的生产力比普通程序员要高一个数量级。但巨大的影响力不仅来自经验和技术，更少不了来自同事和用户的共鸣感，而且无论多少聪明才智都弥补不了后者的缺失。好在这本书可以帮你磨练这项软技能，以期给世界留下更深的烙印。”

——鲍勃·李，Square支付CTO

“作者设定了一个基本的信条——谦虚、尊重和信任——并围绕它们提供了大量的情景案例。这些宝贵的经验和智慧能够帮助绝大多数需要团队合作的工程师（也就是我们）变得更有效率。”

——格雷格·巴罗斯，Facebook产品工程部副总裁

“软件是由人创造出来的。只要能运用本书中所列出的准则，这样的团队无论是在思路、代码品质上，还是在产品发布方面，都可以完胜任何单打独斗的黑客。程序员们，好好学吧！”

——乔纳森·南丁格尔，Mozilla Firefox工程部高级主管

“本书是写给程序员看的，一本教你怎么交朋友、怎么影响他人的书。书中充满了操作性极强的建议和意见，让你在技术团队中过得更开心，变得更有效率，更加如鱼得水。”

——艾德利安·霍罗瓦第，Django创始人之一

“作者说出了我一直在做但是总结不出来的东西。”

——吉多·范·罗苏姆，Python之父

“请把本书送至：

保罗-海宁·坎普

FreeBSD核心团队

必须在1994年3月之前送达。”

——保罗-海宁·坎普，FreeBSD项目程序员

“作者无意为孤独的程序员唱赞歌，相反他们决心要亲手埋葬这个传说。他们撰写了一系列文章来指导那些靠谱的程序员如何对付他们这辈子最复杂的难题：怎么处理好和团队的关系。本书说明了为什么最有人情味的软件往往都是最会合作的人创造出来的，而且它还教你如何同时做

到这两点。”

——约翰·托尔瓦，芝加哥市政府CTO

“这是一本有关软件开发社会学的出色著作，它同时照顾到了开源项目和大公司的需求。对所有新踏入职场的工程师来说，有关管理和应对办公室政治的那个部分绝对是必读的。我的建议是不管你是什么背景的工程师都应该读一读这一章！这是我见过的第一本写给工程师看的、专门有讲到办公室政治的读物，而且可读性非常强。在‘怎么和难以相处的人一起工作’里分享的奇闻异事和实用小贴士都是金玉良言！千金难买哦。”

——蓝俊彪，ArEngineeg’s Guide to Silicon Vauer Startups和St artup Engineering Managerrerg作者

“本书绝对是书中瑰宝，作者分享的理念能让程序员们更好地为团队作出贡献。我们终于有幸可以开诚布公地探讨这个领域的话题，而且还是以这么平和幽默的方式。要是在21岁的时候有这样一本书让我研读领会就好了。”

——布莱恩·奥沙利文，Facebook

“这本书为建立健康的软件开发文化提供了基本的蓝图。它应该成为工程经理和技术领导的必读书目，甚至那些想要了解团队动力学是如何留住顶级人才以及影响软件质量的非技术主管也

不应该错过本书。”

——布鲁斯·约翰逊，Google工程主管

“编程技术能让你混口饭吃，但要是能把它和与人合作的能力结合起来，你就可以改变世界。这本书教你的并不仅仅是当一个更好的程序员，它还要你当个了不起的程序员。”

——克雷·约翰逊，The Information Diet作者

“本书就如何构建成功团队和产品的话题进行了极富洞察力的探索，探讨的是多年来我们程序员在职业生涯里都经历过的痛苦和挣扎。这种轻松愉快的办法不但能同时解决技术团队里的技术问题和人际关系问题，更以一种有趣的方式转化成文字，实在是所有程序员书架上必不可少的书目。”

——乔纳森·勒布朗，X.Commerce首席工程师

“编程现在涉及的已经不仅仅是代码和机器了，它更像是把已有的组件按照新的方式拼装在一起——而这些组件背后的作者都是活生生的人。本书的作者对此了然于胸，无论给出什么样的建议，他们要传达的信息都是非常简单直观的：只要像在代码上那样在人际关系上狠下功夫，你不但可以变成更快乐的程序员，更可以让其他程序员也变得快乐起来。这本书来的正是时候！”

——傅凯，Open Tech Strategies LLC联合创始人

“很少有人会谈及和极客合作时该怎么处理好人际关系这方面的东西，所以多年来我一直通过博客记录本和傅攀勃在各种大会上的演讲。现在我很高兴能方便地在一本书里就读到他们所有演讲里的精华，而不用再追着他们满世界跑了。”

——罗伯特·凯，MusicBrainz首席极客

前言

“工程问题都很简单。人际关系才是最难的。”

——比尔·库格伦，前Google工程部资深副总裁

生活中总是充满了离奇的转折，就好像我们俩从没想过会合作写一本软件工程的书一样。

和大多数电脑狂一样，大学毕业后我们发现自己的兴趣和热情（折腾电脑）居然也是不错的谋生手段。而和那个时代的大多数黑客一样，我们的整个20世纪90年代中期都是在干这些事情，用别人剩下的零件攒机，拿着一大叠软盘安装预览版的 Linux，然后学着操纵 UNIX 机器。我们都是系统管理员出身，然后在互联网泡沫刚起来的时候跑到小公司里去当程序员。泡沫破裂后，我们开始为那些幸存的硅谷企业（比如 Apple）工作，后来又跳槽去了一家创业公司（CollabNet），全心设计开发了一款开源版本控制软件Subversion。

然而2000年到2005年期间，一些意想不到的事情发生了。尽管我们创造了 Subversion，但是我们每天的任务却渐渐发生了变化。我们不再只是天天坐在那里写代码，而是开始领导一个开源项目了。这意味着我们要整天挂在聊天室和一堆程序员志愿者打交道，关注他们都在做些什么。

这还意味着要几乎完全通过邮件列表来协调各种新特性。我们逐渐发现一个项目成功的关键不仅仅是写出漂亮的代码：所有人向着同一个目标一起合作也是同样重要的。

2005 年的时候我们一起创建了 Google 芝加哥分部，以程序员的身份继续着我们的职业生涯。这时我们已经完全融入开源世界了——不仅仅是Subversion，还有Apache软件基金会（ASF）。我们把 Subversion移植到 Google的BigTable架构上，并以Google Code为名发布了一项开源项目托管的服务（类似于 SourceForge）。我们开始参加（后来开始做演讲）各种开发者大会，例如OSCON、ApacheCon、PyCon，还有Google I/O。我们发现同时为企业和开源项目工作的经历让我们无意之间了解到了软件工程团队运作的奥秘。一开始我们做的演讲都是轻松幽默的，大多是批判无用的开发流程（“Subversion最差实践”），后来则开始转向探讨如何保护团队不受害群之马的影响（“开源项目如何应付害群之马”）。越来越多的听众聚集到我们的演讲会场，我们称之为程序员的“集体疗法”。每个人都对我们谈论的问题有着切肤之痛，都想要对大家抱怨一下。

于是，六年来我们做了一大堆反响热烈的和软件开发过程中人际关系有关的演讲。最后奥莱

利公司的编辑玛丽·特莱斯勒建议我们干脆把这些演讲写成一本新书。接下来发生的事情就略过不表了。

想要编写出色的软件？这本书就是为你写的

本书的读者

本书是专门写给那些想要更上一层楼以及编写出色软件的程序员看的。CEO、心理学家、管

理层\计算机理论学家，还有焊电路板的技师等都不算是我们的目标读者（虽然这些朋友或许也能读得津津有味）。在此我们假设本书的读者具备以下两个重要条件。

• 你需要和团队里的其他程序员合作。无论是为公司企业工作，或是某个开源项目或是学校作业的一员都可以。

• 你喜欢软件工程，并且觉得它应该是一件很有成就感、很好玩的事情。如果你只是应付工作混口饭吃的话，那估计你对实现自我价值和获得职业满足感不会有什么兴趣。

在讨论程序员怎么才能“通力合作”的时候，我们总结出了一些表面上看起来和程序员这份工作风马牛不相及的经验和准则。我们会在各章里谈到怎么才能更好地领导一支团队和企业，怎么才能和用户建立起良性的关系等话题。乍看起来这些章好像是专门写给所谓的“管人的人”或者说“产品经理”看的，但我们可以保证，你的职业生涯到了某个阶段的时候一定会有意无意地担当起这样的角色。所以先放下疑虑，继续往下读吧！这本书里讲到的东西是每个软件工程师最终都会碰到的。

警告：本书不是技术手册

首先我们需要澄清一些事情。好学的程序员通常都喜欢去读一些能将某些个特定问题用纯数

学的形式展现出来的书籍，而通常这样的问题都存在完美的流程化的答案。

本书并不属于这种类型。

本书主要关注的是软件开发中有关人的那个方面，而人是很复杂的。正如在我们自己的演讲中说到的那样，“人基本上就是由一大堆间歇性bug组成的”。这里论及的问题和答案都是很凌乱的，很难用完美逻辑来解释。本书读起来像是论文集，而事实上它的确也就是一本论文集。每一章我们都会讨论一系列相关的问题（通常是用案例故事的方式），然后针对整个主题再探讨各种应对方案。你可能需要反复阅读很多页，努力把它们联系起来，甚至思考数日才能融会贯通！

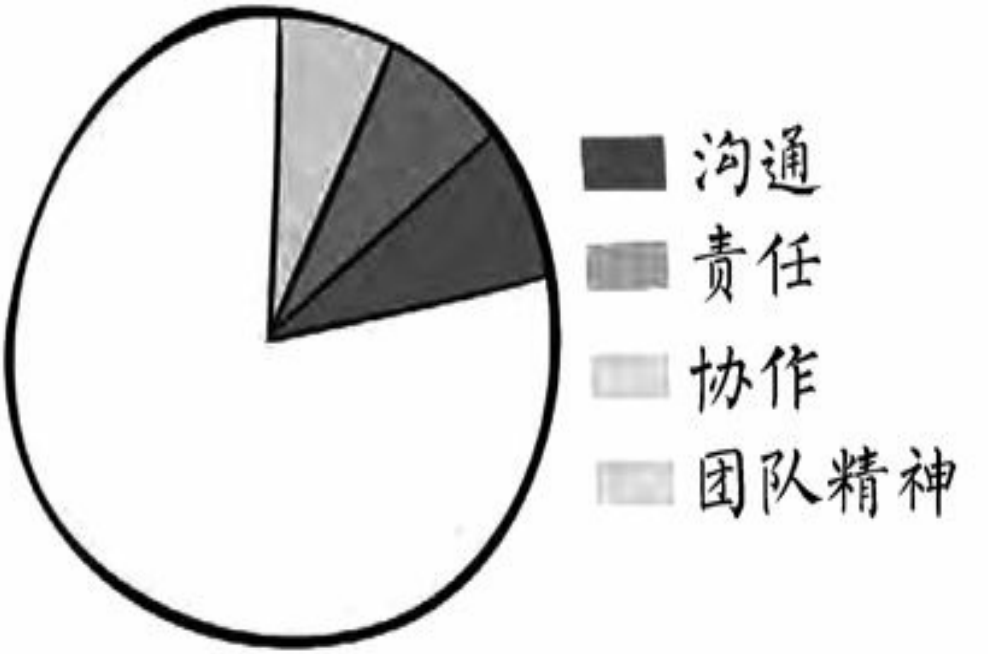
我们在此还要作一些声明。正如我们在演讲里开过的玩笑，“这里的观点完全是基于我们自己的经验得出的，完全属于我们自己。要是你不同意我们的观点，完全可以去自己做演讲啊”。和我们的现场演讲一样，只要和本书主题相关，任何讨论都是值得鼓励的。我们欢迎各种反馈、修正、新观点，以及反驳：你可以在http://www.benandfitz.com/上找到我们。本书的所有内容都来自我们的亲身经历和无数错误中得到的教训。

你还应该注意，我们在例子中提到的都是化名，以保护无辜（或者犯错）的当事人。

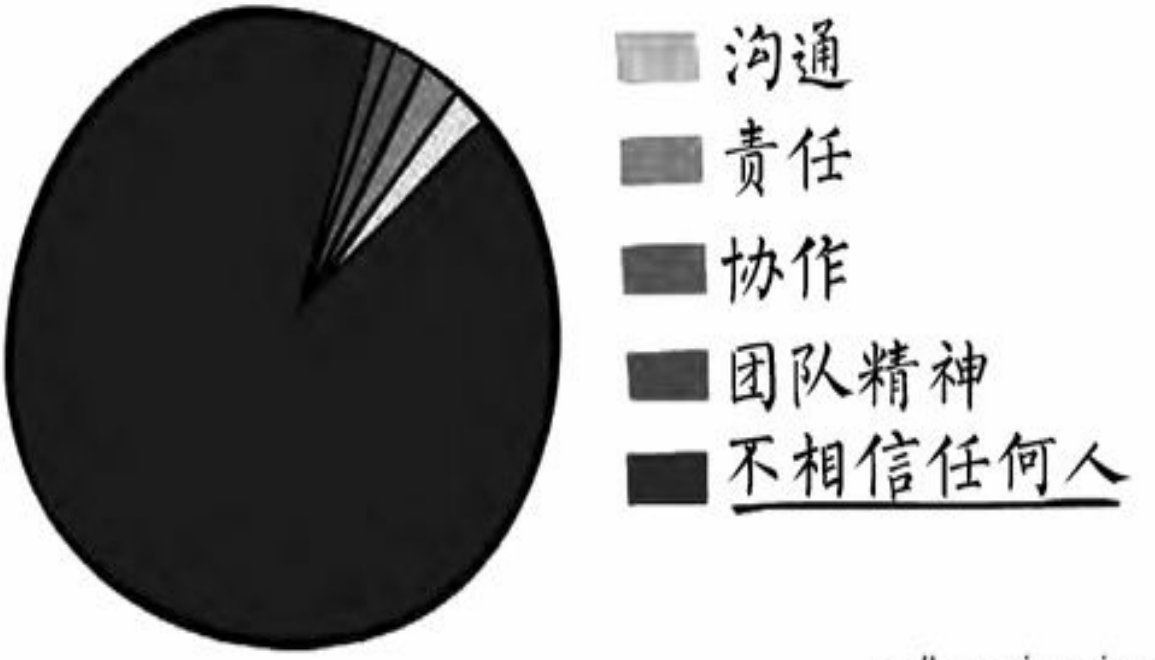
本书的内容都是课本里学不到的

我们认识的软件工程师绝大多数都花了4～10年的时间在学校里学习计算机科学和软件工程。截至本书出版为止，我们不知道有任何一门课程[1]是真正教你怎么在团队和公司里与人合作交流的。好吧，大多数学生都会在学校里被要求参与某个项目，但是教一个人怎么好好地和另一个人协作与把他直接丢到一个迫使他协作的环境里完全是两码事。绝大多数学生最后都不怎么喜欢这种经验。

小组项目应该教会你



小组项目实际上教会了你



小结

所谓成功的程序员不仅仅是追逐最新的语言或是编写最快的代码。职业程序员几乎都是要参与团队合作的，而且事实上团队直接影响个人生产力和幸福感的程度超出很多人的想象。

本书的目标很简单：编写软件是集体项目，而且我们认为人的因素对结果的影响不亚于技术因素。大多数人虽然在编程技术上耕耘数载，但是在人际关系上却从来没有下过功夫，而学习与人合作也是成功路上不可或缺的重要环节。只要你能在软件工程的“软技能”上下点功夫，就能达到事半功倍的效果。

注释：[1] 我们读过汤姆·迪马克的《人件》，的确是一本好书，但它并不是一本专门教工程师怎么与人高效合作的著作，而是一本教管理人员怎么带领团队成功的著作。

目录

既然本书讨论的是软件开发里危险的人际关系，那么把注意力放在有绝对控制权的因素上是显而易见的选择，这个不定因素就是：你自己。

没有人是完美的，但是在给同事挑错之前，你得先知道自己的毛病。我们希望你想想自己的反应、行为和态度——或许你可以从中得到一些心得体会，从而变成一名更成功、更高效的软件工程师。在处理人际关系的问题上花的精力越少，你就有越多的时间编写漂亮的代码。

本章的主旨是要理解软件开发是集体项目（这一实际）。要在团队里获得成功，你必须以谦虚、尊重和信任为核心原则。

不过在此之前，我们先来观察一下程序员的日常行为吧。

帮我把代码藏起来

过去6年来我们俩一直在各种编程大会上做演讲。由于我们是2006年发布Google开源项目托管服务的小组成员，所以我们收到了很多关于这个产品的问题和请求。到了2008年中的时候，我们注意到这些请求里出现了很明显的趋势。

？能让Google Code上的Subversion隐藏某个

分支么？

？能不能实现这样的功能：先把新建项目隐藏起来，等到准备妥当的时候再公开发布？

？我想推倒重来，能不能删掉整个历史记录呢？

你能看出这些请求之间的共同之处么？

这里的要害就是缺乏安全感。人们不喜欢自己做到一半的事情被别人指手画脚。从某种意义上来讲，这是人的本性——没人喜欢被批评，特别是还没完成的工作。这种态度透露出软件开发的某种趋势。缺乏安全感其实意味着背后可能隐藏着更严重的问题。

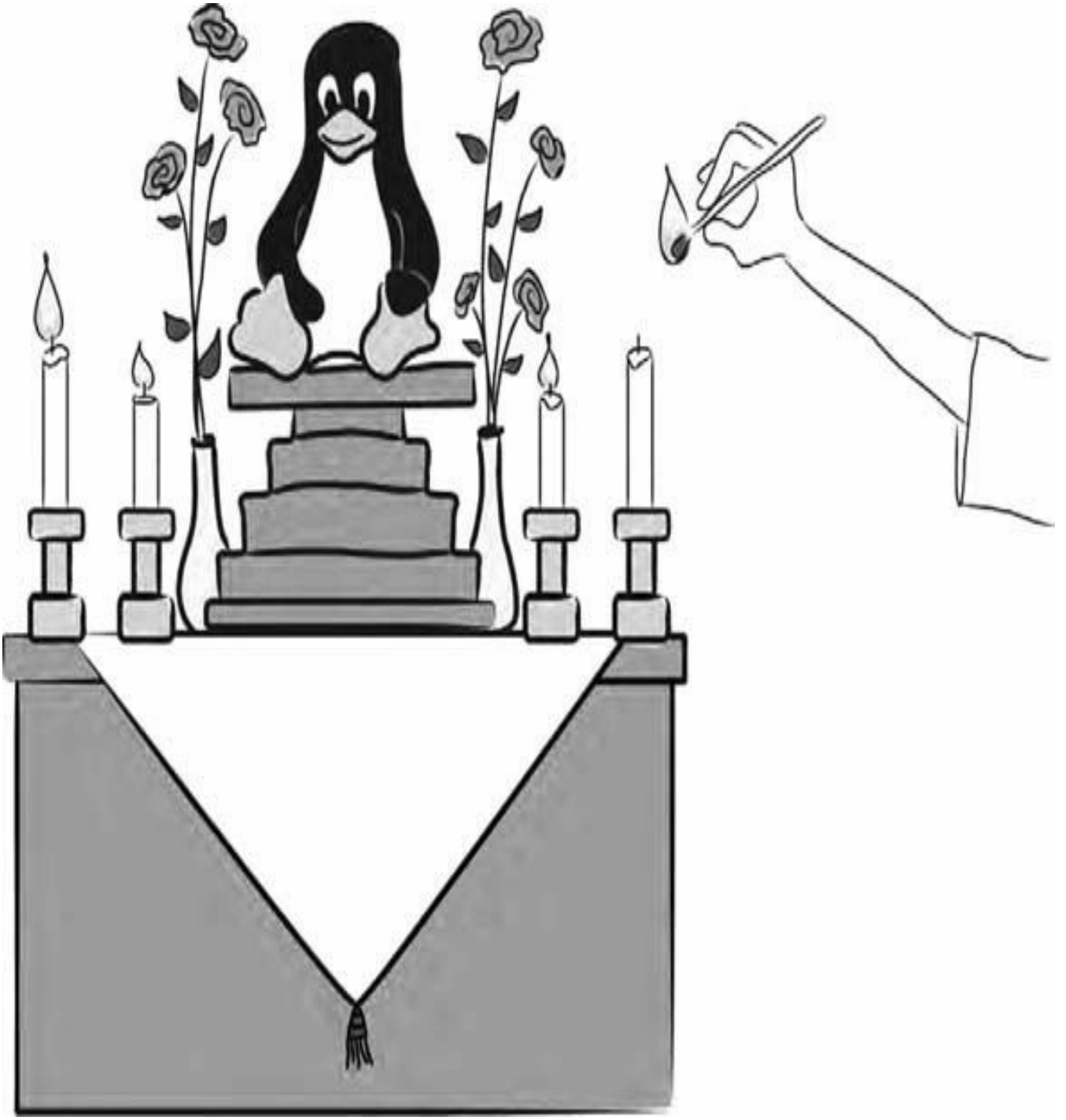
天才的传说

首先我们要澄清一件事情：我们实际上不是体育迷。每次我们的太太们在电视机前为了篮球或者足球比赛欢呼雀跃的时候，我们总会挠挠头皮觉得这有什么好激动的。但不管怎么说，我们毕竟是见证了20世纪90年代初芝加哥公牛队的辉煌（顺便说一句，这是一支篮球队）。我们当时都住在芝加哥，全国媒体在这里聚集了好多年，来报道这支传奇球队。

那么我们在电视和报纸里听到最多的是什么？不是球队本身，而是迈克尔·乔丹，球队的超

级巨星。全世界的球员都想成为乔丹那样的明星。我们可以看到他在其他球员周围跳舞转圈圈，在电视广告里也能看到他。他演了一部很傻的电影，在其中他和一群卡通人物一起打球。他是大明星，每个小孩子都会在球场里偷偷练习打篮球，希望将来有一天也能像他一样。

程序员其实也一样，我们也会有自己崇拜的偶像。莱纳斯·托瓦兹、理查德·斯托曼、比尔·盖茨——这些改变了世界的英雄都作出了了不起的贡献。毕竟莱纳斯靠自己就写出了Linux不是吗？



要小心自己本能地去崇拜事物

其实莱纳斯只是写了一个可以工作的类UNIX内核的初级版本，然后把它贴到了邮件列表上而已。这并不是一项简单的任务，而且它也的确是一项了不起的成就，但是这真的只是冰山一角而

已。Linux 的规模是这个的几百倍，有几百名聪明绝顶的程序员参与了开发。莱纳斯真正的成就是领导并协调他们的工作，Linux 之所以如此耀眼完全是这些人通力合作的结果（另外，UNIX 也是由贝尔实验室里的一小群天才写出来的，并不完全是肯·汤姆森和丹尼斯·里奇的功劳。）

同样的，自由软件基金会的软件都是由斯托曼编写的吗？他编写了第一版Emacs，而bash、GCC，以及所有其他运行在Linux上的软件都是由几百名程序员负责的。史提夫·乔布斯领导的团队开发了麦金塔电脑，还有比尔·盖茨，尽管他为早期的家用电脑编写了 BASIC 解释器，但其实他更大的贡献是围绕MS-DOS创办了一家成功的软件公司。可是这些集体荣誉都被算在了他们这些领袖的头上。

那么迈克尔·乔丹呢？

我们还是一样崇拜他，但事实上他是不可能靠自己一个人就赢得每一场篮球赛的，他真正天才的地方是他和球队一起打球的方式。球队教练菲尔·杰克逊是非常聪明的一个人——他的教练水平是毋庸置疑的：他知道单靠一名球员是无法赢得冠军的，所以他围绕乔丹打造了一支“梦之队”。这支队伍干劲十足，耀眼程度完全不亚于乔丹。

既然如此，我们为什么还是不断地去崇拜这

些故事里的主角呢？人们为什么为明星代言的产品掏钱？为什么我们会想要去买米歇尔·奥巴马[1]的裙子和迈克尔·乔丹的球鞋？

明星的号召力是很大的。人类会本能地去寻找领导者和榜样，崇拜他们，然后模仿。我们都需要榜样的激励，编程世界也不例外，“技术英雄”的现象几乎都要被神化了。我们都想要写出像Linux那样改变世界的东西，或是设计一门了不起的程序语言。

从内心深处来讲我们都默默地希望自己是天才。极客的终极梦想就是得到一个激动人心的灵感，然后闭关数周甚至数月将它完美地实现出来，最后向全世界发布自己的作品，名动天下。同行们会折服于你的聪明才智，人们会排着队来买你的软件，名望和财富更是唾手可得。

不好意思先等一下：醒醒吧，你很可能不是什么天才。

当然我们并无恶意，你肯定是一个很聪明的人，但是你知道这个世界有多少真正的天才吗？的确，你能写代码，拥有这种能力已经算是人群里的聪明人了，但问题在于即便你真的是天才也是不够的。天才也会犯错，好点子和高超的技术并不是软件成功的充分条件，你的职业生涯能否成功完全要看你能不能与人合作。

事实上所谓的天才传说只是我们缺乏安全感

的一种表象罢了。很多程序员都害怕和别人分享他刚刚开始做的东西，因为这意味着同行会看到他们的错误，从而知道这些代码背后的作者并非天才。这里引用本的博客上某位程序员的留言：

“如果别人看到我未完成的作品，我会非常忐忑，觉得他们会因此对我产生质疑，把我当成一个傻瓜。”

这是程序员这个人群里很普遍的看法，所以最自然的反应就是躲起来不断地努力工作。只要没人看到你犯错，你就还有机会最终一鸣惊人。所以产品完善之前还是先藏拙吧。

不愿献丑的另一个原因是害怕别的程序员会偷走你的创意，然后抢先发布。所以只要保持低调，创意就不会被偷走了。

我们知道你现在可能会想：那又怎么样？难道我们不能按照自己的方式工作吗？

事实上还真的不能。我们可以断定你这样做是不对的，而且错得离谱。让我们来告诉你为什么。

隐瞒是有害的

假如你一直都是单打独斗的话，你其实是增加了自己失败的风险，而且浪费了自己成长的可能性。

首先，你怎么知道自己选的路是对的？

假设你是一名狂热的自行车设计师，有一天你想到了一个绝妙的主意，设计出一个具有颠覆性的换挡装置。你订购了零件，然后在车库里泡了好几个星期来制作原型。当你的邻居（他也是自行车爱好者）问你最近在忙什么的时候，你想还是先保密好了。在这个设计完善之前，你不想任何人知道它的存在。几个月以后，你遇到了瓶颈，没有办法令原型正常工作，但由于这个项目是保密的，所以也没办法向你的机械师朋友们寻求帮助。

直到有一天，你的邻居从他的车库里推出一辆自行车，上面有一个全新设计的换挡装置。其实他也一直在建造一个和你的设计类似的东西，只不过他有自行车行的朋友帮他的忙。这时你一定非常窝火吧。你把自己的东西拿给他看，他立刻就指出了你在设计上存在的缺陷——要是你早点拿出来的话，搞不好这些缺陷在一开始就修复了。

单打独斗的结果往往令人失望

这里我们可以得到好几个教训。真正做出产品之前不愿分享好创意实际上是一场很大的赌博，你很容易在一开始就犯下很基本的错误，你有可能是在重新发明轮子[2]。还有你完全丧失了合作的好处。注意到你的邻居和别人合作后进展有多快了吗？这正是为什么人们在跳进泳池前会拿脚趾试试水的原因：你需要确定自己在做的事情

是对的，方法也是对的，而且不是重复劳动。一开始就踏错步的概率总是很高的，越早征求意见和反馈，就越能把风险降低[3]。记住这句久经考验的原则——“确保失败尽早发生，尽快发生，经常发生”——我们会在本书稍后详细讨论失败的重要性。

尽早分享不仅仅可以防止一开始就步入歧途和检验创意，它还可以强化所谓的“公车因子”。

公车因子（名词）：一个项目里，需要有多少人被公车撞到才能令其完全瘫痪。

你的团队的公车因子是多少

你的项目里知识的分散程度是多少？假如只有你一个人理解原型代码是怎么工作的，那么或许对你的职位的安全程度来说是很高，但要是你被车撞了的话，对项目本身来说就是灭顶之灾了。但如果有一个朋友和你合作的话，公车因子就可以翻倍。要是有一个小组一起设计制作原型的话就更好了——项目不会因为某位成员的消失

而完蛋。记住：这里只是比喻，并非真的是被车撞，而是指任何生活中都会发生的意外情况，比如结婚、搬家、离职或是照顾生病的家人等。你若想要确保项目有一个光明的前景，就一定要把公车因子纳入考量。

除了公车因子之外，还有整体进展的速度。人们往往会忘记单独工作通常都是很艰辛的，进展有时会慢得令人难以接受。独自工作能让你学到多少东西？你的进展如何？网络虽然是观点和信息的大熔炉，但是它是替代不了人与人之间真正的交流的。直接和别人一起工作能潜移默化地提升集体智慧。每次遇到事后觉得可笑的障碍时，你浪费了多少时间才走出死胡同？想想如果有同事在旁边帮你一把的话会有什么不同——他们会立刻指出你哪里弄错了，该怎么解决它。这就是软件公司里团队通常都是坐在一起（或是在一起结对编程）的原因：你需要一位旁观者。

再打一个比方。想想你是怎么用编译器的。当你编译一个有一定规模的软件的时候，你会花好几天甚至好几个星期坐在那里写代码，然后在全部写完确保一切完美后才第一次进行编译么？肯定不会啦。你能想象一口气编译50 000行代码会有什么后果么？程序员是最需要不断反馈的：写一个新函数，编译，加了一个测试用例，再编译一下，又重构了一部分代码，再编译一下。这

样就能在生成代码的时候尽快地改正笔误和bug。我们的每一步都离不开编译器，好像僚机一样。有些开发环境甚至能在我们打字的同时进行编译。正是依靠这种方式我们才能保持高质量的代码并确保软件在正确的轨道上演化。

这种快速反馈不仅在代码层面，对整个项目来说也是必不可少的。有抱负的项目都必须快速演化并不断适应环境的变化。项目可能会遇到意料之外的设计瓶颈，或者行政上的阻力，又或者只是发现事与愿违。需求总是在往意想不到的方向变化。你要如何通过反馈来发现自己的计划和设计中需要修改的地方？答案是：团队合作。埃里克·雷蒙说过，“只要有足够多双眼睛，就能发现所有的bug，”而更好的说法是，“足够多双眼睛可以确保你的项目保持正确的方向。”闭门造车的结果往往是当实现最初的创意后，却发现世界已经完全改变，原本的产品已经失去意义了。

工程师和办公室

20 年前的传统看法是工程师需要自己独立的办公室，关上门安静工作才会有生产力。这样才能保证他有大把不受干扰的时间可以专心编程。

而我们却认为把绝大多数工程师[4]关在独立办公室里不但毫无必要而且还很危险。今时今日，软件业已经不是个人英雄的行业了，它需要团队的合作，保持沟通渠道的通畅甚至比随时在线的

网络连接更重要。虽然得到了不受干扰的环境，但如果方向错了，那也只是在浪费时间罢了。走进任何一家21世纪以来创建的、快速成长的高科技公司，你会发现工程师都是围坐在共享的小格间（也叫“牛棚”）里，或是共用一张桌子，很少见到各自把自己锁在办公室里的情况。

当然啦，你还是需要找到办法来过滤杂音和干扰的，我们发现很多团队因此发展出一套自己的沟通技巧来表达自己现在很忙，有事等会再说。以前和我们合作过的一支团队就有这样一套语音中断协议：假如你有事找玛丽谈，你要说，“breakpoint 玛丽”。如果玛丽正好有空档可以停下来，她就会转过来和你谈。要是她正在忙，她会说“ack”表示知道了，然后你就先去忙其他事情吧，她那边忙完了就会来找你。

有些团队会给工程师发降噪耳机来解决背景噪音的问题——事实上在很多公司里，带着耳机实际上就表示“不是重要的事情就别烦我”。另外一些团队则会用在显示器上放置记号和玩具的办法来表示自己不愿被打扰。

请别误解我们——我们完全认同工程师应该有不受干扰的时间来全情投入编码之中，但是我们认为他们更需要团队之间高效通畅的沟通渠道。

因此用一句话来总结就是：本质上，单打独

斗比合作风险更高。相比担心自己的创意被偷走或是被人笑话，你更应该担心自己是不是在错误的方向上浪费了大量时间。

令人遗憾的是，这种“创意要保密”的想法并非软件行业独有的，所有领域都普遍存在这个问题。就拿学术界来说，科学原本应该是自由开放、信息共享的。但是“不发表即灭亡”的迫切需求，以及对基金拨款的竞争却造成了反效果。学者们不再乐于分享。他们小心翼翼地保护着自己的想法，秘密地进行研究，把犯过的错误都掩盖起来，使最终发表的论文看起来似乎得来全不费功夫一般。其结果往往都是灾难性的：要么不小心重复了前人的工作，要么在一开始就不知不觉犯下了错误，最糟糕的是创造了一些原本有趣，但现在完全过时无用的东西。其中浪费的时间和精力都是无法估量的。

别再把自己变成上述统计结果的一部分。

团队才是王道

现在我们来小结一下。

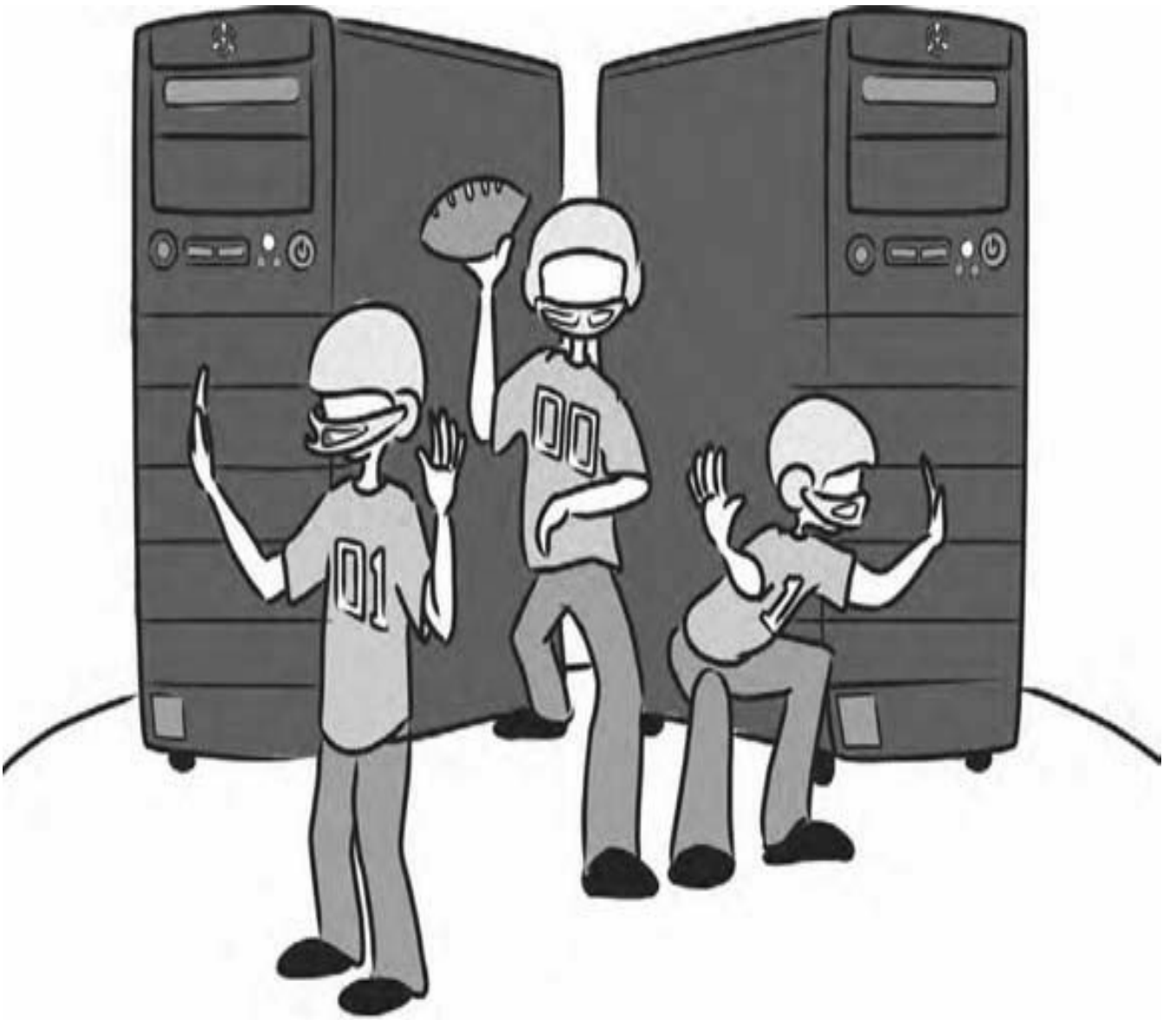
我们到目前为止一直在打磨的观点就是，在编程领域里，真正的独行侠是很罕见的——就算他们真的存在，他们的非凡成就也不是凭空而来的。这些改变世界的成就几乎都是集体智慧努力

得来的结晶。

因此建立一支全明星团队才是真正的目标，不过想达成这个目标，难度高得惊人。最好的团队能充分利用好队里的巨星是没错，但是集体的力量一定是大于个体力量之和的。

用一句话来说就是：软件开发是集体项目。

乍看之下这个理念很难让人接受，毕竟这和我们心里的天才程序员幻想是相抵触的，所以先记下来再慢慢理解吧。



记住，软件开发是集体项目

一个人躲在自己小黑屋里抖聪明是没用的。光靠自己神神秘秘地搞创造发明是不可能改变世界，令千百万用户受益的。你需要合作，告诉别人你的想法，让别人帮你分担劳力，向别人学习，进而打造一支出色的团队。

现在来做个实验：你能想出几个应用广泛的成功软件是真正由一个人完成的？（有些人可能会提到“LaTeX”，但很 难说它是一款“应用广泛”的软件，除非你觉得那些写学术论文的人在所有电脑用户里占有相当大的比例！）

我们会在这本书里不断地重复这个集体项目的概念。高效的团队才是通向成功的关键所在，你应该尽可能地以此为目标，这同样也是本书的使命。

三支柱

到这里我们对于团队的观点已经立论了。既然团队合作才是开发成功软件的捷径，那么如何才能打造出（或者找到）这样优秀的队伍呢？

答案是很难。要达到合作无间的境界，你首先要学习理解所谓的社交技巧“三支柱”。这三项原则不但是人际关系中的润滑剂，更是所有良性

互动与合作的基本。

谦虚

没有人是宇宙中心。谁也不是万能的，谁都会犯错。你必须不断地提高自己。

尊重

你必须真心实意地关心同事。他们都是活生生的人，他们的能力和成绩都需要得到肯定。

信任

要相信别人的能力和判断力，在适当的时候懂得放权[5]。

我们把这些原则简称为HRT。发音是“heart”而不是“hurt”，因为它是要减轻痛苦而是不是伤害别人。事实上我们的论文就是直接基于这些原则的：

基本上所有的社交摩擦最终都是由于缺乏谦虚、尊重，或是信任而造成的。

乍听之下似乎有点难以置信，但请姑且相信我们。你不妨仔细回想一下生活中遇到过的各种令人难受的社交场景。先不说别的，当时大家是不是都有保持有礼有节？是不是真的有尊重对方？是不是有相互信任？

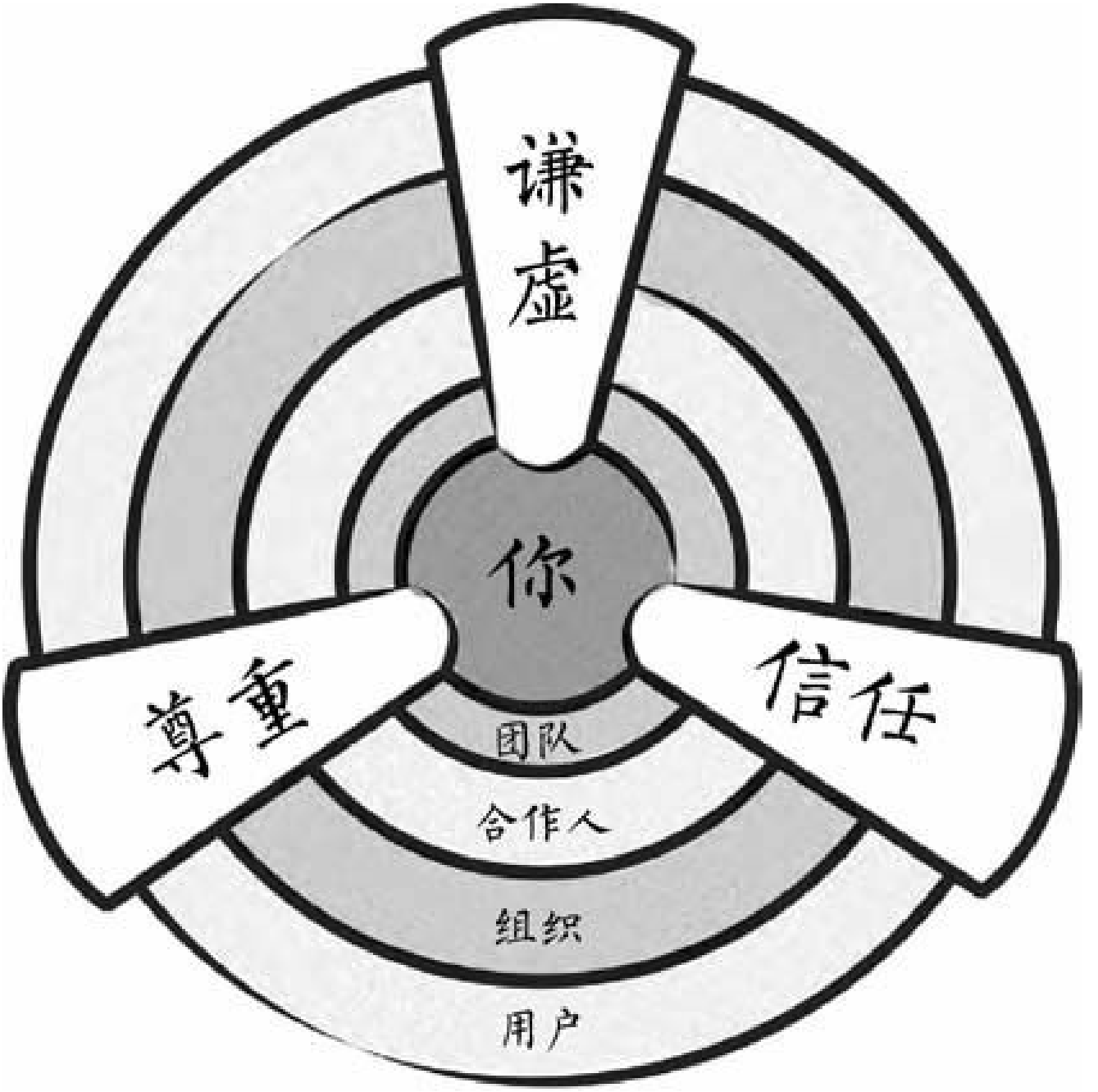
我们相信这些原则至关重要，本书正是围绕它们而展开的。

本书是以你为起点的：让你了解HRT，并真正理解消化以HRT为交际核心的意义。这就是第

一章的内容。然后我们能不断地扩大影响的范围。

第二章讨论的是在三支柱之上打造团队的挑战。建立团队文化是走向成功的必由之路——这就是之前说到的“梦之队”。

然后我们会讨论那些每天会和团队打交道，但是却不属于核心团队文化的人。他们可以是其他团队的同事，或是项目的志愿者。他们中的很大一部分人不但完全无视H RT，更有甚者，完全就是老鼠屎！学习如何抵御这种人对团队的伤害是你的首要任务。不过你的终极目标应该是拔掉他们的爪牙，让他们融入到你的团队文化中来。这是扩充队伍的最佳方式之一。



拥抱HRT才能合作无间

大多数团队都隶属于更大的公司，这种环境常常和害群之马具有同等危害。你的产品能否顺利发布亦或是最终面临被取消的命运，完全取决于你能不能学会如何避开各种行政障碍。

最后，你的用户也是需要考虑的。我们往往

会忘记他们的存在，但就是他们决定了产品的寿命。没人用的软件是没有存在价值的。对团队至关重要的HRT原则同样（也应该）适用于你对用户的态度，你能从中获得的好处是不可言喻的。

说到这儿，让我们先暂停一下。

当你翻开这本书的时候，你大概从没想过要参加什么每周互助小组吧？对此我们表示同情。社交的确是个棘手的难题。人类是混乱复杂的生物，完全无法预料，有些人会让人觉得不愿与之打交道。与其耗费那么多精力去分析各种社交场景，步步为营，还不如干脆完全放弃努力。行为完全可以预判的编译器岂不是容易对付得多？干嘛要跟那些社交的麻烦较劲呢？

这里引用一段理查德·海明的著名演讲[6]：

我花了很多时间给我的秘书们讲笑话，和他们搞好关系，这也让我从他们那里得到了许多好处。比如有一次，不知道什么缘故，梅山[7]所有的复印机都坏掉了。别问我为什么，反正就是坏掉了。而我正好要复印一些文件。于是我的秘书给霍姆德尔的朋友打了个电话，跳上公司的车，开了一整个小时去帮我复印，然后再开车回来。这就是我花了那么多功夫讲笑话逗她开心，表示友善得来的回报。真可谓投之桃李，报之琼瑶。也就是说，通过使用系统并研究如何让系统帮你做事，你就学会了调整系统，让它按照你的意愿工

作。

这个故事告诉我们：不要低估社交的力量。社交不是勾心斗角，或是操纵别人，它是通过建立起人与人之间的关系来把事情做成功，而且这种关系延续的时间肯定比项目本身更长。

HRT实战

说了那么多谦虚、尊敬、信任的东西，听起来都像是纸上谈兵。既然这样，我们现在来看看它们和现实是怎么联系起来的。既然我们追求的是实际可操作的建议，所以下面会给出一系列特定的场景实例。其中很多看起来似乎都是显而易见的，可一旦开始仔细思考，你就会发现自己（和同僚们）犯错的频率有多高了。

放下自负

好吧，说得更直白一点，就是让某些缺乏谦逊的人收敛一点儿。没人喜欢和总是以自我为中心的人合作。哪怕你真的是会议室里最聪明的那个，你也没必要到处炫耀。例如，你是不是总是觉得自己对于每个话题都要抢先发表意见，或是要作总结性发言？你是不是觉得自己需要在一项议案或是讨论中对每个细节发表评论？或是你总是要知道谁在干什么？

注意，“表现得谦逊一点”和人见人踩的懦弱

完全不是一回事：自信并没有错。只是不要过了头，非要弄得自己好像无所不知一样。其实更好的思路是想办法促成“集体”荣誉感。不要去担心你的个人形象是不是高大，而应努力去营造一种团队和集体荣誉感。Apache软件基金会在围绕软件项目经营社区上已经有很长的历史了。这些社区对自己都有很强的认同感，根本不欢迎对那些只关心自己往上爬的人。

自负这个东西有很多种解释，就看你怎么理解，但很多时候它都会妨碍你的工作，让你进步缓慢。这里从海明的演讲里再摘取一段小故事来证明我们的观点：

“约翰·图基几乎总是不修边幅的模样。每次他参加重要会议的时候，对方总是要好一会儿才反应过来原来这是位大人物，应该认真听他说。有好长一段时间约翰都要面对这种麻烦。这实在是太浪费时间了！我不是说你应该屈服，我说的是，‘作出妥协的样子能给你带来很多好处’。如果非要由着自己的性子来，‘我就是我行我素的人’，那么你就在整个职业生涯里不断地付出一些不大但是很讨厌的代价。日积月累就会变成原本没必要的大麻烦。……通过使用系统并研究如何让系统帮你做事，你就学会了调整系统，让它按照你的意愿工作。不然的话，你就得终其一生去和这种潜规则作斗争。”

学会批评和接受批评

乔是一名程序员，他最近找到了一份新工作。在上了一星期的班以后，他开始深入地学习了解团队的代码。出于责任心，他开始以温和的方式对同事的代码提出疑问。他通过E-mail的方式发出代码审查，礼貌地询问设计初衷或是指出逻辑上可以改进的地方。两个星期后他被叫到了主管办公室。“出什么事了？”乔问道，“我是不是什么地方做错了？”主管看起来不太高兴：“乔，最近有很多人投诉你，说你对同事太苛刻了，把他们说得左也不是右也不是。大家都很不爽，你最好低调一点。”乔因此大受打击。若是在一个完全以H RT为基础的环境里，乔的代码审查应该是可以被接受的，同事们也会认同这种做法。但是在这个例子里，乔就应该对团队里弥漫的不安全感多加小心，在引入代码审查的时候应该更谨慎一点。

在职业程序员这个行当里，人身攻击是相当罕见的——批评通常都是为了产品好。其中的诀窍就是要确保你（和你周围的人）认识到对某人的创造性工作提出建设性批评和人身攻击之间的区别。后者是毫无意义的——这种攻击不但显得小气，而且几乎不可能有什么建设性。而前者通常都是有益的，对改进具备指导性。最重要的是，它蕴含着对对方的尊重：只有真正关心对方

的人才会提出建设性意见，希望对方在自身或是工作上有所进步。所以学着尊重对方，礼貌地给出建设性批评吧。如果你真的尊重一个人，那你就会自发地选择有技巧有帮助的措辞——这种技巧需要很多练习才能学会。

作为谈话的另一方，你也应该学会接受批评。这不是说你在技术上要谦虚，而是说你要信任对方，相信他是从心底为你好（当然还有你的项目），完全没有把你当傻瓜的念头。编程和所有技能一样，都需要熟能生巧。如果你的同事提出的做法比你现在的好，你会把它当成是对你的性格或是人生价值观的攻击吗？我们真心希望你不会。同样，你的自尊和你的代码不应该有什么联系。换句话说，你和你写的代码是两回事。你应该不断地告诫自己，你和你写的代码是两回事。不但你自己要相信这个观点，还要让你的同事也认同它。

别把你的自尊和你的代码等同起来

举个例子，假如你有一个比较敏感的同事，那么下面这样的话是万万不能说的：“老兄，你完全把这个方法的控制流程给弄错啦。你应该和大家一样用标准的xyzzy代码模式才对。”这段话全是反模式的东西：首先你告诉他，他“错

了”（好像这个世界非黑即白一样），然后要求他作出修改，最后还指责他写的东西和所有人都不一样（让他觉得自己很傻）。可以预见，这样会让对方变得很抵触，得到的回应也肯定是没有什么理智的成分。

好一点的版本可以是这样“：嗨～我有点看不懂这段代码的控制流程。要是用xyzzy代码模式的话会不会更清楚一点？维护起来也方便？”注意这里，你谦虚地把问题归到自己头上，而不是他。他没有错，只是你理解代码有困难而已。另外提出的建议也只是提供了一种方法让可怜的你能理解代码，或许还能在项目长期维护上有所助益。你也没有提出任何要求——你的同事完全可以谢绝这个建议。讨论的范围被限定在代码上，没有涉及任何人的价值观或是编程技术。

快速失败；学习；迭代

商业界里有一个著名的（也是说烂了的）段子，说是曾经有一个经理因为犯下大错损失了上千万美金。第二天他非常沮丧地回到办公室开始整理桌子，这时他接到了预料之中的通知，“CEO想要见你”，于是他步履沉重地走进CEO的办公室，默默地递上一张纸。

“这是什么？”CEO问道。

“我的辞呈，”这位高管答道，“你叫我来就是要炒我鱿鱼的吧。”

“炒鱿鱼？”CEO一脸狐疑地答道“，我为什么要炒你鱿鱼？我刚刚才花了一千万来培训你！”[8]

当然，这个段子很极端，但是故事里的CEO心里明白，就算炒鱿鱼也不能挽回这一千万的损失，而且还要损失一名很有价值的高管，因为他以后肯定不会再犯这种错误了。

我们俩都在Google工作，而我们最喜欢的Google的格言之一就是“失败是可以接受的”。广泛的认识是如果没有经历过失败的话，就说明你的创新还不够，或者你承担的风险还太小。失败是更上一层楼的绝佳机会。事实上，托马斯·爱迪生曾经说过：“即便我找到了一万种行不通的办法，也不代表我失败了。我并不会因此而泄气，因为每次试错都是在前进。”

Google通常都采用这样一种（我们之前讨论过的）理念，“不要等到完美的时候再出来”：只要产品大致可用，就立刻把它按照原样公开给大众。这就是Google Labs的使命。这种办法使得成功和失败的地方得以立即显现出来，这样编程团队就可以尽快从中学习，迭代，然后发布新版本。它的缺点是Google常常会被揶揄说他家的产品总是在“beta”阶段，比如Gmail就“测试”了四年多。而它的优点在于可以迅速做出调整以适应变化，在很短的时间内就能做出惊艳的产品。所有的这一切都要求谦虚的特质——把不完美的软件

展示给用户是可以接受的，另外还需要一些信任，即用户真的会认同你的努力，并且期望迅速看到改进。

从错误中学习的诀窍是要记住自己摔倒的地方，按商业用语来说就是“事后检讨”。但是要特别小心，千万不能把事后检讨的文件变成一堆无用的道歉和借口——这不是它的目的。真正的事后检讨应该包含有关“学到了什么”以及“怎么改正”等经验教训的详细注解。然后要保证把它放在一个随手可及的地方，并且认认真真地按照上面所写来实施改进。记住，正确地记录错误还能让其他人（不管现在还是将来）方便地了解事情的原委，以避免重复历史。不要抹掉自己的足迹——像跑道一样点亮它们，为后来人指路吧！

一份出色的事后检讨应该包含以下内容：

? 简要

? 事件的时间线，从发现到调查，再到最终结果

? 事件发生的主因

? 影响和损失评估

? 立即修正问题的步骤

? 防止事件再次发生的步骤

? 得到的教训

为学习预留时间

辛蒂曾经是超级巨星——在她的领域里她绝

对是大师级的程序员。被提升为技术指导后，责任随之变大，而她也接受了这个挑战。很快她就能领导周围的每个人并指导他们工作了。她出席各种大会并就自己的领域作演讲，不久之后她已经能掌管好几个团队了，她也非常享受“专家”这个头衔。但是她却开始厌倦这种生活了，不知不觉地她不再学习新东西，成为所有人之中最聪明、经验最丰富的专家的这种新鲜感开始褪去。尽管表面上光鲜亮丽，但是总好像缺了点什么似的。直到有一天她终于意识到自己的领域不再尖端，人们开始转向其他更有兴趣的课题。她到底出了什么问题？

我们来分析一下：成为人群中最睿智的人的确很让人高兴，而且能够指导别人绝对可以带来了不起的成就感。但是问题在于一旦攀至顶峰，人们往往就会停止学习了。而当一个人不再学习的时候，她就会开始觉得厌倦，一不小心还会变得落伍。虽然当领导很过瘾，但是只要能放下一点骄傲，你就能开阔眼界，接触新鲜事物。这说穿了其实还是谦逊的问题和是不是愿意像指导别人一样接受别人的指导。偶尔应该跳出自己的舒适区，在更大的舞台上接受各种挑战。这样你才能长久地保持愉快的心情。

学习保持耐心

傅攀勃在多年前写过一个把CSV仓库转到

Subversion（以及后来的Git）上去的工具，由于RCS和CSV的行为异常古怪，他不断地发现各种诡异的bug，比如 CVS会默默地吃掉明明是非法的RCS文件。而他的老朋友和同事卡尔对CVS和RCS都非常了解，他们决定一起来修复这些bug。

可是就在他们开始结对编程的时候发生了一个问题：傅攀勃喜欢自底向上的工作方式，他会深入项目，通过快速尝试各种情况来梳理细节。而卡尔却是喜欢自顶向下工作方式的工程师，他会先搭好框架，构建好调用栈里每个方法的实现后才会考虑bug的问题。这就引发了两人之间巨大的冲突和分歧，有时甚至会产生激烈的争执。傅攀勃和卡尔尽了极大的努力和专注，还恪守了HRT 的原则才得以完成这项任务。HRT 最终不但拯救了项目，也保住了他们之间的友谊。

对影响保持开放的态度

你越是容易受影响，你就越能影响别人；你越是示弱，你就越强壮。这两句话看起来有点自相矛盾。但是每个人都碰到过某个同事，固执到叫人恼火。无论怎么试图说服他都没用，越劝越不听。最后怎么样？就我们的经验来讲，大家最后就会自然而然地把他当成障碍物“绕道而行”，再也不会有人听取他们的观点或反驳。没人希望这种事情发生在自己身上吧？所以请记住这一点：接受意见改变自己没什么大不了的。不要随

意挑起争斗。不要忘了，要别人听你说话，首先是要学会当一个听众。注意在被影响的时候，如果你要听取意见，一定要在你下定决心或是确定宣布你已经做好决定之前——要是你的主意老是改来改去的，别人会觉得你这个人没什么立场。

说到示弱，初看之下也有点奇怪。如果一个人承认自己对某个东西一无所知，不知道要怎么解决问题，那他还能得到多少团队的信任么？示弱就是暴露弱点，这不是在摧毁自己的自信吗？

其实不是。承认自己犯错或是无知从长远来讲其实能提升你的形象。事实上它蕴含了HRT的全部方面：它对外表示了“谦虚”，这是有责任心、负责的态度，这也是表示“信任”别人意见的态度，同时作为回报，别人也会因为你的诚实和坚强而“尊重”你。所以有时候最好的答案就是：“我不知道。”

诚实和谦虚不是氪气石

想想那些政客吧。这帮人之所以讨厌是因为

无论他们错得多离谱，或是对件事物表现得有多无知，他们也从来不会承认，所以很多人都觉得政客的话是不可信的。这种现象主要是因为政客经常暴露在对手的攻击下。但是写软件的时候就用不着总是抱着防备的心态了——你的同事是合作者，不是竞争者。

下一步

若能读到这里，说明你已经在掌握“与人合作”的艺术的征途上跨出了第一步，你已经检视反省了自己的行为。只要能在日常生活中运用这些策略，你就会发现合作将变得更自然，工作效率也能大大提高。

重要的改变要由自己做起，然后慢慢影响其他人。在下一章里，我们要讨论如何在你隶属的团队中培养HRT文化。

注释：[1] 第一夫人。

注释：[2] 假设你真的是自行车设计师的话。

注释：[3] 当然值得注意的是有时候一开始获得太多反馈也不是好事，我们在后面的章节里再讨论。

注释：[4] 当然我们也承认有些内向的人确实需要更多安静独处的时间，即便没有自己的办公室，他们也需要一个相对安静的环境才能好好工作。

注释：[5] 如果你过去遇到过所托非人的情况，那这一条的确是非常难做到的。

注释：[6] “你和你的研究，”http://www.cs.virginia.edu/~robins/YouAndYou
译注：网上有中文翻译，下面两个都是转载的。
http://bbs.virology.com.cn/archiver/tid-5061.htmlhttp://louisfaye.wordpress.com/2007/11/27/and-your-research-richard- hamming/

注释：[7] 译注：Murray Hill，贝尔实验室所在地。

注释：[8] 这个段子在网络上有无数版本，曾经被套在好多著名经理人的头上。

第二章 培养出色的团队文化

团队和团队之间的文化差异是非常大的，它反映了多种多样的价值观和对不同事物的重视程度。有些能引领团队走向成功，而有些却会导致彻底失败。即便是那些能带领团队成功的文化也有高下之分，有的非常高效，能让团队的注意力完全集中在编写软件上面；有的则会给手头的工作带来很多干扰。这一章我们要探讨的就是团队文化，特别是各种对成功有助益的沟通技巧。我们将会说明这些技巧是如何帮助出色的工程师团队更有效率地编写软件的。

什么是文化

当我们听到“文化”这个词的时候，脑子里浮现的情景往往是某个晚上去歌剧院看演出，或是高中生物课上在培养皿里繁殖细菌的画面。工程师团队的文化其实和后者的差别并不大。

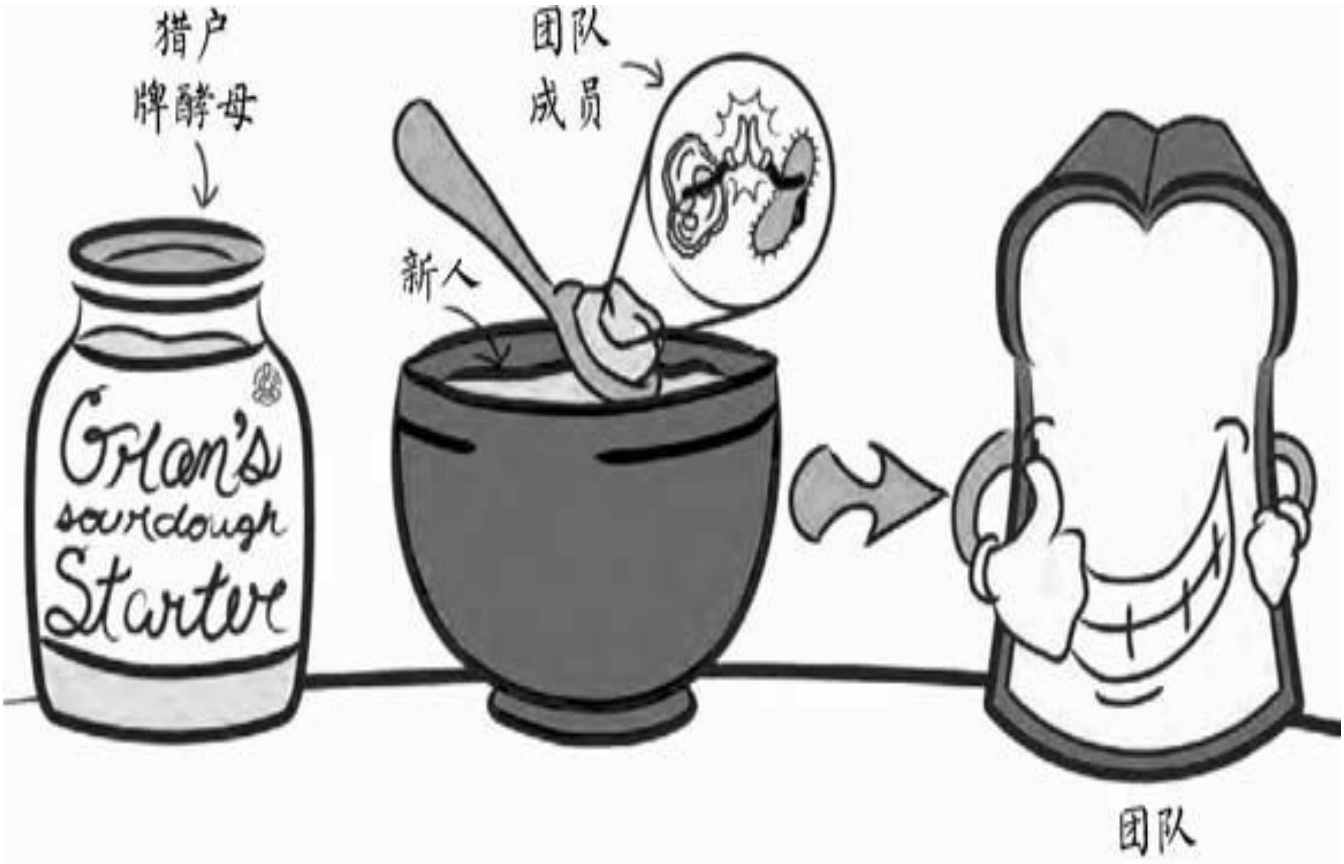
假如你吃过非常美味的发酵面包并且对烘培它的人感到好奇的话，你会发现这面包的关键就在于酵母。酵母是面粉和水里的酵母菌和乳酸菌。酵母菌能让面包膨大，而乳酸菌是让面包具有强烈酸味的秘密。然而并非所有乳酸菌都是一样的，有些乳酸菌产生出来的风味更好吃，所以

当面包师找到味道一流的酵母（即含有恰当酵母菌混合比例的面团）时，她就会小心翼翼地通过加水和面粉来继续培养这种益菌群。然后只要取出一小部分酵母混进面包里，瞧！马上就能得到一条好吃的发酵面包啦！这是因为酵母里的菌群不但能产生她想要的风味，而且还能盖过面包食材本身以及空气中的酵母菌。


Gran's
Starter

你需要一块好酵母

团队文化就像是一块含有酵母的面团：酵母（团队创始人）能将菌群培养物[1]植入生面团（团队新人），从而变出一块好吃的面包（团队）。如果团队本身具有很强的风格，它就能压过新人带来的任何“坏习惯”[2]。如果团队文化不够强势，团队就会被新人带来的风气所影响。由于未知的文化往往伴随着未知的结果，因此一个团队最好拥有自己的熟悉的团队文化。



好的酵母能让新人融入你的文化中

团队文化不仅仅是成员们编写代码的方式或是成员之间的相处之道，它还包含了所有人都认可的经验、价值观、目标。我们工作过或观察过的每个工程团队对此都有不同的见解。团队或者公司的创始人决定了团队大部分的特点，但它还是会随着时间不断变化发展。

组成团队文化的元素非常丰富。有些和代码编写有直接的关联，比如代码审查、测试驱动开发，以及在大规模进行编码前对于良好设计文档所抱的态度等。有些元素则和社交的关系更大一点，比如每个星期四一起去某个餐厅吃午饭，或是星期五的时候一起去大家都喜欢的酒吧喝一杯之类的。有些在外人看来甚至很傻很古怪：比如Google匹兹堡分部曾经就在一条货运火车线旁边，每次有火车经过的时候（顺便提一句，火车可是非常长的哦），大家都会跳起来互射Nerf泡沫枪弹[3]。所有的这些元素组成了一支团队的文化，并且影响着团队的生产力以及吸引和留住优秀成员的能力。

如果你观察一下现在任何一家成功的软件公司——Google、Apple、微软、Oracle，你就会发现每家公司的企业文化都非常不同，但都是由创始人和最早的员工确立起来的。随着公司的成长和成熟，其企业文化也发生了演化和改变，但是它们仍然在方方面面保留着独一无二的特质，比

如产品开发模式、对待员工的方式，以及与其他公司竞争的手法等。

为什么要关心它

简单来说，关心团队文化的原因就在于如果不努力营造它，那么团队最终会因为某个特别强势的人的出现而被注入他个人的文化基因。这种文化或许是生产力强劲的健康文化，能产出大量的优秀代码。但事实往往相反，你会突然发现自己在争执和争斗中浪费了太多精力，没有办法集中精神去设计和编写代码。不仅如此，团队拥有一个共同的价值观并愿意为之奋斗是非常重要的事情。要是团队不在意自身的团队文化，那么不仅构建强烈的团队认同感以及对自身工作的骄傲感会变得十分困难，而且会很容易受新人影响而引入糟粕。

大多数工程师都会犯的第一个错误是假设建设团队文化是负责人的事。这种想法再离谱不过了：尽管创始人和负责人通常会关注团队文化的健康情况，但其实每位成员都是团队文化的一部分，都要为定义、维护和保护它作出贡献。每当有新人加入时，她并不只是从团队负责人那里了解团队文化，而是从一起工作的每个成员身上学习。例如，你在仔细检查新同事的代码的时候，

会向她解释为什么你的团队是按照某种方式写代码的，这样她很快就会明白团队重视的是代码里的哪些部分。她还会通过观察团队的工作、交流，以及解决冲突的方式来学习团队文化。

所谓“强壮的文化”，是指能接受有益的改进，同时又能抵御有害的激进变化的团队文化。最成功的团队文化都把大部分的注意力放在了开发优秀软件上面。如果你的团队把主要精力放在了其他东西上（比如聚会、开会，或是怎么把别人踩下去等），那么也许你们很团结，但是却写不出什么东西来。如果你最高兴的时刻是写代码和发布产品，那么最好还是找一个重视这些东西的团队，然后努力维护这样的氛围吧。并不是说缺乏强壮和生产力的文化就没有办法发布产品，只不过在缺乏这些特质的团队文化里，发布产品需要耗费你更多的时间和精力。强壮的文化能为你提供专注、效率和力量，这些东西都能让团队更快乐。

团队文化有意思的地方就在于如果你清楚地定义好它，它是会进行自我选择的。在开源世界里，那些构建在HRT之上，专注于编写干净、优雅、可维护代码的项目会神奇地吸引拥有相同价值观（即尊重信任他人，并且致力于编写干净、优雅和可维护的代码）的工程师加入。然而，如果你的团队文化是侵略性的、欺凌性的，或者是

感情用事地进行人身攻击的话，那最终吸引过来的也只能是这样的人罢了。

我们在Apache软件基金会已经见过很多这样自我选择的文化了：ASF聚集了很多软件开发团队，它们大多都是社区性质的团队，采用共识决策[4]的方法运营。很多新成员加入邮件列表的时候，或是无意或是有意，会有一些和团队文化冲突的行为。社区成员一般会尝试教育新人（有时会很有礼貌地进行教育；有时候，唔～，就“不是那么客气”啦），假如这位新人对ASF团队的做事方法不感兴趣的话，他就会离开去寻找更适合自己的项目。

在企业里，团队的自我选择是通过招聘来实现的，这可以是潜移默化地筛选潜在候选人重视的技术和优势，也可以是明确地在招聘过程中考察与团队文化的契合度。Google在招聘过程中采用的是明确的方式，因为它在面试的时候十分重视文化上的契合度：如果一个人在参加Google的面试中各方面都像是超级明星工程师一样，却无法和团队合作，或是要求非常结构化的环境的话，面试官还是一样会在反馈里亮起红灯。

如果你在招聘的时候不重视团队文化的契合度，结果招了一个不合适的人，那最后无论是让他融入团队还是请他走人都要耗费你大量的精力。不管结果如何，其代价都是非常高昂的，还

不如在招聘的时候就确认新成员能够和现有团队一起工作。

确认新成员的文化契合度的唯一方法就是在面试的时候注意这方面的东西。很多公司（比如Google）都将文化契合度作为面试官面试候选人时的考察点之一。有些公司为了避免招聘失败甚至采用更激进的做法：他们会在技术面试之前先进行单独的面试来考察文化契合度，对于价值观不同的人根本就不予考虑，哪怕他们在技术上没有问题。这种流程对于建立和保护强壮的文化是至关重要的，而且并非偶然的产物；事实上它通常都是由公司创始人和早期员工有意识地设立的。

文化和人

编写软件和在流水线上简单地组装产品可不一样。有些工作只需要几天培训和一些基本的工具就可以完成，如果有工人退出或离职（或者就是学不会），你只需要替换为另一个工人就可以了。在流水线环境里，员工通常只要机械性地完成简单的任务即可，而不需要什么创造性思维或是解决问题的本领。但在软件行业里，产品工程师则需要大量的创造性思维[5]，这就是说如果你想要出色的产品，那么就离不开出色的工程师。而

且如果你希望这些出色的工程师能做出漂亮的产品（并且留住这些优秀人才的话），你就需要为他们建立起一种团队文化，让他们可以放心地分享创意，并且在决策过程中拥有发言权。

如果你想要优秀的工程师为自己的团队工作，首要的就是雇佣出色的工程师！这听起来有点奇怪，不过事实就是大多数优秀的工程师都喜欢加入那些拥有优秀工程师的团队。我们认识的很多优秀工程师都倾向那些可以向业界巨人学习的团队[6]。那么一开始的时候怎么才能吸引这样的工程师呢？

对于初创人员来说，他们需要的不仅仅是为产品开发贡献力量，更重要的是能参与到产品的决策中来，这通常就意味着某种程度的共识驱动管理。在自顶向下的管理模式中，首席工程师就是团队主管，其他工程师则被雇为团队成员。这是因为这些团队成员的成本比较低，也比较容易指挥。但是你很难在这样的团队里找到优秀的工程师，毕竟，要是能在另一家公司里主导开发的话，她何苦屈尊在这里听指挥呢？但是在共识驱动管理之下，整个团队都能参与到决策中来。

很多人一听到“基于共识决策的团队”时，立刻就会想到一群嬉皮士围着篝火唱着“Kumbaya”[7]，却做不了任何决定或是完不成任何工作的画面，但是这种固有印象其实更符合功

能失调的团队所具有的症状，而非基于共识决策的团队。所谓的“共识”，是指每个人都对产品的成功抱有强烈的主人翁精神和责任感，同时团队的领袖也真的愿意倾听团队的意见（即HRT中“尊重”的部分）。这表示有时候产品成功需要反复的讨论和检讨，而有时候团队又必须搁置争议先行动起来再说。在后面这种情况下，团队成员会决定把日常的决策工作托付给一位或者少数几位领导[8]。为此，团队作为一个整体必须在大方向上达成一致，不管你信不信，要做到这一点，关键就是要为它设立一个章程（本章后面会详细讨论）。

和团队的决策过程同样重要的是同事之间的相处方式，因为这是最具备自我选择性的方面。如果你的团队习惯骄傲自大，对同事大吼大叫的话，你能吸引（和挽留）的也只能是这种攻击性极强、极端自我的人（事实上，我们认识的绝大多数女性对这种环境都非常厌恶）。如果你能建立起HRT的氛围，同事之间能友善相处，提出的都是经过考量的建设性批评，那么你不但能吸引更多的员工，而且大部分精力都可以分配在编写软件上。培养强烈的团队自豪感是好事[9]。若团队被个人的自负盖过，那就是灾难的源泉。我们会在第四章里讨论防止这种情况发生的办法。

建设性批评是任何工程团队成长发展的基

石。接受批评是需要一定的自信心的，而我们觉得建设性批评是最容易接受的一种。另一方面，给出建设性批评要比直接批评嘲笑对方困难得多。而且我们也清楚，请别人批评指正是非常困难的事情，大多数人只会觉得你想听的是赞许和褒扬罢了。如果你身边有朋友和同事能在你需要的时候真正提出建设性批评，千万不要疏远他们，诤友难得。

如果你想要在工作上有所进步，或者改掉自身的坏毛病的话，这样的朋友和同事可以帮助你认识到阻碍自己成为优秀工程师的缺点。除非自我意识和自我反省的能力异常突出，在没有别人批评的情况下，你只会不断重复同样的错误——这就好像有人告诉你牙齿上有菜渣一样。例如，在本书出版的过程中，我们请了十几个人来帮忙审稿，告诉我们写作上存在的问题，这些建设性批评大都非常详细，是十分宝贵的意见。不管你是不是喜欢本书，如果我们忽略甚至不敢去请求这些珍贵的反馈的话，它只会比现在更糟。

攻击性性格的人（通常）也能够适应比较平和安静的环境，但是比较内向的人却很少能在激烈的环境里生存（或者开心地工作）——这样的环境下，他们的声音不但很容易被杂音盖过，而且会逐渐影响他们参与的积极性[10]。如果你想找一个能让大多数人高效工作的环境，那还不如自

已去建立一个谦虚、尊重和信任的文化氛围呢！

建立在尊重之上，气氛随和的团队文化更容易受到性格激进的人的影响，其程度要大于随和的人对于激进的团队文化所产生的影响。因此随和的团队文化要特别小心这一点，不要被激进的新人牵着鼻子走，特别是要避免和这样的人发生激烈的冲突。有时候，团队的资深成员要挺身而出，正面迎击这样的新人，防止他们对团队的随和氛围产生破坏。我们在第四章里还会更进一步讨论对付这类“害群之马”的办法。

优秀团队文化中的沟通模式

沟通一般都不是工程师的强项，他们宁可花一个下午和（可理喻，有逻辑的）编译器搏斗，也不想和（不可理喻，情绪化的）人打交道。大多数时候，工程师都视沟通为编写代码的障碍，但是如果你的团队没有事先达成共识，那么是没有办法知道你的代码写得对不对的。

工程师通常更喜欢和可理喻的、有逻辑性的人待在一起

只要检视一下任何优秀、有效率的工程师文化，你就会发现它们对各种沟通渠道的重视，例如邮件列表、设计文档、任务宗旨、代码注释、产品说明等。让所有人认同团队的方向并完全了

解团队要做什么是很花精力的，但是这些努力的回报是生产力的提高和更快乐的团队。

沟通的指导原则之一就是在同步沟通的时候（比如开会），人越少越好。而在异步沟通的时候（比如E-mail），涉及的听众越多越好。更重要的是，你必须确保项目文档里的信息要尽可能地让所有人都看到。接下来我们要讨论软件开发过程中团队里主要会用到的沟通方式。其中有些看起来似乎是无需赘言的，但是其中还是存在一些细微差别，值得再好好审视一下。有一件事是肯定的：如果你不花精力好好沟通，最终一定会浪费更多的精力去做一些没必要的工作，或是团队里别人已经做过的工作。

高层面同步

在最高层面上，整个团队必须目标一致，在报告进度上遵守最佳实践。

任务宗旨——不，我是认真的

每当听到“任务宗旨”这个词，很多人的第一反应都是大公司颠来倒去说的那些空洞无物、营销意味浓重的公司宗旨之类的东西。下面这个例子就是一段某家不具名的大型电信公司的公司宗旨。

我们立志成为全球最受尊敬、最有价值的公

司。我们的目标是通过令市场瞩目的优质服务来丰富客户的个人生活，帮助企业获得成功，同时为股东创造价值。

说来奇怪，我不觉得有任何人尊敬这家公司！这里是另一家大型企业的公司宗旨：

提供实时的解决方案以满足客户需求。

这句话到底是什么意思？它可以代表任何事情——如果我们在这家公司工作的话，我们根本就不知道到底是编写软件更重要，还是修漏水管或者送披萨更重要。正是这种含糊不清的东西让公司宗旨在世人眼里显得那么空洞。

对工程团队来讲，撰写任务宗旨就是要准确地定义产品的方向和范围。好的任务宗旨不是轻易写成的，但是它或许能为你节省数年的时间来澄清哪些是团队应该做的，哪些是不应该做的[11]。

几年前，当Google决定要把Google Web Toolkit（GWT）开源的时候，我们曾经担任过团队的指导。我们检视了开源和闭源开发之间很多不同的地方，特别注意了在一个谁都可以来发表意见、贡献补丁、对产品最微小的某个方面提出批评的环境里，设计、讨论和编写软件所要面对的困难[12]。在解释了这些困难之后，我们告诉团队，他们需要写一份任务宗旨，向公众描述哪些是产品的目标（还有哪些不是）。

出于上述各种原因，有些工程师对此颇为犹豫，但是也有一些人表示有兴趣，团队负责人似乎也觉得这是个不错的主意。但是等到真正坐下来开始写的时候，大家对于这份宗旨的内容、基本要素和风格又产生了巨大的争论。在一系列讨论（和会议）之后，团队不但交出一份漂亮精练的任务宗旨，甚至还准备了一整篇“让GWT更出色”的文档[13]来逐条解释它。他们还专门开辟一节来解释哪些不是项目的目标。下面就是这份任务宗旨。

GWT的任务是要通过让程序员利用现有的Java工具，为任何现代浏览器构建全功能的AJAX，从而彻底改善用户的网络体验。

这句话包含了很多基本要素，我们认为它是任务宗旨的绝佳范例：它不但包含了方向（通过让程序员……改善网络体验），同时还限制了范围（Java 工具）。几年后有一次我们和那位团队负责人吃饭的时候，傅攀勃告诉他说我们非常感谢他当年坚定地支持我们让团队撰写任务宗旨。他回答说，其实当初我们提出的时候他觉得这完全是浪费时间，但是当他开始和团队争论怎么写的时候，他居然发现了一些之前都不知道的事情：他的首席工程师居然不同意产品的方向！

撰写任务宗旨强迫他们在产品的走向上求同存异，否则到时候一定会拖缓（甚至停滞）开发

进度。在网上公开任务宗旨后，不但整个团队自己可以完全专注在那些必须完成的事情上面，而且也不需要再浪费口舌去和潜在的贡献者争论产品方向——他们只需要让新人去读一读“让GWT更出色”这篇文章就可以了，很多问题都能迎刃而解。

任务宗旨可以帮助团队求同存异

任务宗旨可以保证项目不会随着进展而偏离目标。但它也不是一成不变的。如若所处大环境

或是商业计划发生巨变（对创业公司来说很常见），团队成员绝不能顽固不化，他们应该重新评估原来的任务宗旨是否还有价值。像宪法一样，这种改变肯定是异常困难的，因为它原本就不应该是可以随意更改的东西。但是当这个时刻来临之际，至少它是可以改变的，人们也应该予以认真考虑。任务宗旨应该与时俱进，及时反映公司或者产品的变化。

开会要有效率

绝大多数工程师都把会议归类为必不可少的恶魔。如果利用得当，开会是很有用的，但是人们经常会滥开会，组织不当，而且几乎一定会把会议拉得很长。我们期望的会议应该像污水处理厂一样：数量要少，位置要远，还要在下风口。所以这一节也非常精练，只讨论小组会议。

首先是所有会议中最可怕的：常务会议。这种会议一般一周开一次，应该只能用于发布简单的通告或者介绍——要求所有与会人员一个一个报告进展情况（不管他们是不是真的有什么重要的事情要宣布）完全是浪费时间，只会让人觉得不耐烦，恨不得立刻打晕自己好不用再受折磨。任何值得深入讨论的内容都应该在会议结束后进行，而且应该只涉及相关人员。这种会议应该允许人们在重要议程结束时离开，而且要是没什么大事，或者可以通过 E-mail 来宣布消息的话，完

全可以取消会议。我们曾经见过一些把能够参加会议和身份地位等同起来的公司文化，结果没人愿意被落下。说得直白一点，这种做法实在是太愚蠢了。

有些工程师非常推崇像敏捷这样的开发方法所倡导的每日站会，如果能保持短小精悍，这种形式是可以接受的。这种会议一开始的时候都很短（15分钟左右），大家真的是围站在一起，轮流报告自己正在做什么，但是如果不保持警惕，很快这些会议就会被拉长到 30 分钟，人们会坐下来东拉西扯，好像是参加集体治疗课程一样。所以，如果你的团队打算开每日站会的话，一定要事先委任会议主持人，而且他要看着时间，防止会议变得太长。

如果你正打算做一些新的设计，那么尽量把会议人数控制在五个人以下——除非只有一个人可以拍板，否则在超过五个人的会议室里是做不出任何新设计或者决策的。如果你不信，可以去找五个朋友一起跑到市中心，然后试试看你们六个人能不能设定一条覆盖六个旅游景点的观光路线。我打赌除非你们让其中一个担任领头羊，然后他怎么走你们就怎么走，否则你们会站在路口争论一整天。

没用的会议和折磨没什么两样

会议对所谓的“工作时间”[14]来说通常是一种干扰，这个词的灵感来自保罗·格雷厄姆的一篇文章“工匠和老板的时间安排”[15]。如果工程师总是要中断手头的工作去开会的话，他们是很难完全

进入工作状态的。你应该在日程表上留出三到四个小时的大段时间，把它们标记为“忙碌”或者“工作时间”，然后把事情做完。如果一定要开会，那么尽量把会议安排在休息时间，比如午饭时间，或是下班之前。Google一直都有（可惜也经常被无视）“周四不开会”[16]的传统，就是为了让大家有时间把工作做完。好的开始就是成功的一半，以后甚至可以空出20～30个小时的大段时间来专注工作。

有关开会的五条小贴士：

1．只邀请一定要参加的人；

2．开会前要决定好议程，而且要事先通知所有人；

3．达成目的后应提早散会；

4．注意别跑题；

5．尽量把会议安排在休息时间前后（比如午饭时间，下班前等）。

开会前一定要事先准备好会议议程并且至少提前一天通知所有与会人员，让大家知道要开的是什么会。把会议人数降到最低（别忘了同步交流的成本）。我们认识很多人会无视没有议程的会议邀请，包括工程师、工程经理，甚至主管和副总裁。

为了能达成会议的目的，只邀请那些真的需要与会的人。有些人在发现与会者实际上是在看

E-mail 而没有认真开会的时候，会规定不准带笔记本电脑去开会。可惜这只是治标不治本——人们在开会的时候看 E-mail 很有可能是因为他们一开始就不需要参加这个会议。

主持会议的人应该拿出权威来毫不犹豫地（有礼貌地）打断那些跑题乃至妄图独揽话语权的人。要做到这一点并不容易，但却是值得的。最后，也是最重要的一点就是，如果时间尚早而议程上的议题已经讨论完毕，千万别犹豫，立刻散会吧。

地理上分散的团队

如果你隶属一个分布各地的团队，或者工作的地方离同事们很远，那么不但需要在沟通渠道另寻出路，更要在沟通上面多下功夫。如果团队里有异地的同事，那么决策的记录和分享往往都是书面的形式，比如E-mail。在线聊天、即时通信、走廊上的随性交谈都可以是沟通讨论的地点，但是我们还是需要把这些讨论的相关内容广播给所有人知道，保证大家的信息对等性以及参与度（另外一个好处是，存档后的邮件列表就是文档）。当需要快速建立对话的时候，视频对话也是非常有用的工具，而且现在为整个团队配备摄像头的成本也是相当低廉的。

我们在做Subversion项目的时候有一句座右铭：“邮件列表上查不到的讨论等于没发生

过。”在聊天室里说话可以百无禁忌，什么想法都可以提出来，但是到了真正确定方案的时候，我们千万不能忘了那些没参与讨论的人。即便团队分散各地，只要将这些谈话内容重新发表到邮件列表上，每个人就都有机会看到决策是如何形成的（如果觉得有必要，他们还可以发表意见）。如果你想要培养基于共识决策的团队文化，那么这一点是至关重要的。

和一个远在异地的同事交谈应该和直接去他的办公桌交谈一样做到无摩擦。如果你在异地工作，那你应该利用所有的媒介（比如在线聊天、即时通信、E-mail、视频对话、打电话等）和团队沟通，保证大家不但知道你的存在，也清楚你在干什么。最重要的是，千万不要低估了面对面交流的力量。无论你发了多少E-mail，在网上聊了多久，打了多少个电话，还是要大胆地经常跳上飞机去见见同事们。这也适用于海外的雇员、团队和分部雇员——安排好时间去总部和同事们见个面吧。

设计文档

开发新项目的时候，往往很难抑制住那种想要立刻开始写代码的冲动，但是这么做的结果往往是很糟糕的（除非你只是打算很快地做一个粗糙的原型出来）。同样，很多工程师都不做设计就直接开始编码，后果往往是惨不忍睹的。

设计文档一般由一个人负责，两到三个人撰写，审核的人数则更多一些。它不但勾勒出整个项目的前景，也直白地告诉整个团队你想做什么以及打算怎么做。而且这时你还没有花费几周（甚至数月）编写代码，比较容易接受意见和建议，帮助你改进产品和优化实现。另外，当设计文档定型之后，它能帮助你安排划分项目的工作量。而且，当编码工作开始以后，设计文档绝不是一成不变的：随着项目的进展而发生的变化应当及时反映在设计文档里，而不是等到了产品要发布的时候才去更新，不过这一点是说起来容易做起来难。大多数团队根本没有文档，而有些团队只在很短的一段时间里有很漂亮的文档，剩下的很长时间里只有过期的文档。

说到这里，你也不要走到另一个极端“唯设计文档论”里去。我们曾经遇到过为了一百行的小程序写了整整四页设计论文的控制狂。如果写一份设计文档的时间足够把整个项目推翻重写好几遍的话，那就不要浪费时间写设计文档了。用经验和判断来作出适当的权衡吧。

每日进行的讨论

假设大方向已经确定，接下来需要确定的就是每天团队用来协调的工具。这些工具很有用，

但是可能会限制沟通的效果，因为它们常常缺乏面部表情以及身体语言这种辅助的沟通渠道。结果它们可能会导致沟通产生误解，从本质上对HRT造成威胁。不管怎么说，这些工具对绝大多数团队来说仍然是不可替代的，（只需要一点点努力）就可以大大提高生产力。

邮件列表

我们还没见过写软件不用邮件列表的人，不过这些技巧可以让你更好地利用邮件列表。

很多非常成功的项目都有好几个邮件列表，把开发讨论、代码审查、用户讨论、公告发布、调度邮件，以及各种管理琐事区分开来。有时候一些比较小的项目一开始的时候就会试着依葫芦画瓢，在只有三个工程师和两名用户的时候就建了一大堆邮件列表。这相当于五个人为了讨论一件事情却准备了六间会议室——最终只会导致讨论缺乏连贯性，大量的重复，还有很多空置的房间。最好的做法其实是从一个列表开始，当信息量太大无法管理（通常是列表成员开始抱怨求饶的时候）的时候再逐渐增加数量。好好花点时间培养邮件讨论的礼仪——文明讨论，不要被那些“嘈杂的少数人”[17]所阻挠。

虽然当整个团队坐在一个办公室里的时候，邮件列表并非是进行讨论的最佳选择，不过用它来发布会议议程、会议记录、决策、设计文档，

以及任何相关的文字信息再好不过了，它是一个非常方便的集中记录点。通过这些列表将所有帖子存档，并为之建立可搜索的索引，如果是开源项目，可以把它公布在网上；如果是闭源项目，则可以把它放在公司的内网上。这样你的项目就拥有了一份完整的历史记录，当新人对过去作出的某项决策心存怀疑的时候，就可以很方便地回溯查看当时那么做的原因。如果不存档这些讨论的话，你会发现自己不得不一次又一次地重复讨论它们。

在线聊天

在线聊天对于团队来说是非常方便的沟通方式，特别是因为它能在不打断同事的情况下快速发送请求（当然啦，她的聊天工具一定要设置成可以不受打扰的）。团队如果需要在晚上或是周末做一点简单工作，或是某位成员休息一两天的时候，这个工具对团队来说是很方便的。一对一聊天是非常有用的工具，在团队交流里绝对占有一席之地，但是我们强烈推荐采用群聊的方式[18]。

多年前在即时通信还不流行的时候，团队通常都会挂在互联网中继聊天（就是所谓的IRC）频道里，大多数讨论都是以群聊的方式进行的。有时候这会显得有点吵闹，但是这样一来，讨论就是“当着整个团队的面进行”的了，而且如果谈

话内容和其他人没什么关系的时候，转成私下讨论也是非常方便的。这样其他人就可以选择随时加入讨论，或者潜在一边只看不说，甚至可以对错过的讨论补充意见。它的便捷不但体现在即兴讨论可以随时进行上，更表现为能帮助团队培养社区感，即使他们相隔万里也没关系。新成员不一定要参与，光是看大家讨论（或是阅读过去的谈话记录）就能学到很多东西，效果往往能叫人惊叹不已。

当即时通信出现之后，很多原本在集体聊天室里进行的谈话都变成了私下交谈，因为即时通信默认就是如此。害怕自己会问出傻问题的不安全感会让人更倾向于一对一的讨论，以避免当众出丑的风险。可惜这么做只会给团队增加负担，因为这样一来知识就无法共享，同事之间可能会重复不断地问同一个问题。所以无论用什么软件来沟通，我们都强烈建议要有一套方便可用的群聊机制。VPN和安全限制的确会造成一点障碍，但是这层沟通渠道对于团队来说是很有价值的，再麻烦也值得。

现在很多人第一次听到IRC的时候，都会嘲笑它简陋的文本环境，最新版的 IRC 客户端看起来都要比 iChat 或者Google Talk的旧版本差劲。但是千万不要被 IRC其貌不扬的外表和风格所迷惑——专门设计的异步群聊才是它的杀手级特

性，绝大多数客户端都支持无限向上翻页，所以你可以回过头去看之前错过的讨论。漂亮的视频会议软件包、共享白板系统等或许很有诱惑性，但是它们只会打扰工程师的工作，完全抹杀了IRC异步的优势。IRC并不是你唯一的选择，但是如果你打算用别的软件的话，一定要选真正为群聊而设计的产品，不要选那种只是硬是加上群聊功能的即时通信系统。

有时候人们更喜欢在网上聊天。还记得那是我们头一次参加编程马拉松，就是很多开源贡献者互相见个面（很多人都是头一次见），然后一起做项目。我们走进一间几乎沉默的房间，里面摆了十几张桌子，每张桌子都坐了六到八个人，在笔记本上疯狂地打字。因为我们迟到了，所以看到这个情景的时候还以为大家都在专心写程序呢，所以我们也坐了下来，打开笔记本，启动编辑器，登入项目的 IRC 频道，看看那些没办法参加马拉松的人是不是在线，结果发现 IRC 频道里正讨论得热火朝天。我们打了个招呼，说刚刚才到马拉松现场，立刻就有好几个人在 IRC 频道里和我们说哈啰了，而当时他们就坐在我们三米开外啊！你可以想象我们吃惊的程度了吧。习惯了在网上聊天而产生的惯性其实只是一部分原因，更多的原因是这种和团队的沟通方式对有些人来说是最自然的。可在坐了四个多小时的飞机后，

我们非常想要和人说说话，于是我们还是站了起来，越过桌子走过去和他们面对面的交流了。

使用bug跟踪系统

如果你打算使用问题和bug跟踪系统的话（事实上你也应该用），很重要的一点是你要准备好一套流程来处理和分流bug，这样才能鼓励人们提交并且按时修复重要的bug。如果bug跟踪系统不被重视或者没有轻重缓急的安排，也就不会再有人提交 bug，人们的不满也会无处发泄。当团队最终开始在 bug跟踪系统里乱挖的时候，他们很可能会去修复那些不重要的bug，却把重要的bug丢在一边。

记住，bug跟踪系统的本质只是一个稍微专业化一点的“网络论坛”或者“电子公告栏”。所以它也拥有很多和邮件列表相同的特点，因此很多最佳实践都是可以照搬过来的。和bug有关的随心交谈都应该作为更新记录在bug跟踪系统里，所有的想法和决策都要“正式”发布给所有人看到。语气措辞要文明，绝不容忍任何挑衅行为。如果讨论变得过于冗长或是琐碎，就应该把它暂时移到主邮件列表上去——E-mail客户端更擅长处理这种复杂的讨论串。

沟通也是工程的一部分

市面上软件开发流程方面的书可以说是数不胜数。我们无意在此做过多深入的讨论，只想提几点和沟通特别有关的，无论遵循何种开发理念，这些要点都值得关注。

代码注释

代码注释风格是一样很主观的东西。原作者常常通过详细的注释解释自己的意图和理由，这些对理解代码都很有帮助，但是却要付出后续维护的代价：过时甚至不准确的注释反而会极大地妨碍对代码的理解。同样，过于扼要的注释，或者干脆没有注释也会浪费将来维护或者API用户的时间。注释一般是用来说明代码里缺失的部分，以及起得不好的名字，然后把代码的功能再解释一遍。注释应该尽量解释为什么代码要那么写，而不是去解释代码做了什么。

注释在函数（或者方法）层面是最有用的，特别是用作API文档的时候。注释不应该涉及过多细节，用一句著名的希腊谚语来总结就是“μηδε?ν α?γαν”，即“过犹不及”。此外还应该好好为团队准备一套注释风格，然后每个人都要好好遵守——我们认为风格一致比风格本身更重要[19]。风格指南还应该自陈存在的理由及其目的——例如，这里是Google C++风格指南的介绍部

分[20]。

C++是很多Google开源项目的主力开发语言。每个C++程序员都知道这门语言拥有多少强大特性，然而伴随这种强大特性的是它的复杂性，这也使得代码更容易出现 bug，难以阅读和维护。

本文的目标是通过讲解C++编程中的各种要点来驾驭这种复杂度。这些规则在保证代码可控性的前提下，能让程序员高效地使用C++的语言特性。

所谓的风格（也就是可读性），就是管理C++代码的约定惯例。风格这个词本身有点不恰当，因为这些约定所涵盖的内容远远不止源文件格式化那么简单。

强制统一代码的风格是保持代码可控的一种方法。让别人快速看懂、理解代码是非常重要的。保持一致的风格并遵守约定意味着可以简单地通过“模式匹配”的方式来推断各种符号的含义以及它们具有哪些不变的特性等信息。通用和强制要求的习惯和模式可以帮助理解代码。有时候可能有很好的理由去改变某些风格，但是为了保证一致性，我们仍然会予以拒绝。

这份指南要解决的另一个问题是 C++的特性膨胀。C++博大精深，拥有很多高级特性。有时候我们需要限制（甚至禁止）使用某些特性，这

是为了保持代码简洁，避免这些特性带来的各种常见错误和问题。本指南会列出这些特性，并一一解释为什么要限制它们。

Google的开源项目都遵守这份指南的规定。

注意本指南并非C++教程，我们假定读者对这门语言已经有了相当的了解。

在源文件里署名（也就是“作者标签栏”问题）

每个人都希望自己的工作得到认可，艺术家会在自己的画作上签名，文字工作者会把自己的名字放在书脊上或是博客顶部。无论以何种方式，渴望得到认可是人的本性，但是在我们看来，在源文件里留下名字绝对是弊大于利。你一定在各种源文件的顶部见过这样的东西，它们往往和版权声明挤在一块儿。

--

Created: October 1998 by Brian W. Fitzpatrick <fitz@red-bean.com>

--

在源代码顶部署名的传统已经是老黄历了（我们两个都这么干过，唉！），在程序大多由个人而非团队编写的年代这样做或许还说得过去。但是今天，很多人只是接触到一小块代码而已，文件里的作者标签栏只会导致争吵不休，浪费时间，甚至伤害感情。因此，我们强烈反对在

源文件里署名的做法（最多也就是署上审核人的名字，告诉大家在修改这个文件后首先应该拿去给谁看，但是要小心不要有文件归属的暗示）。

想象一下，假如你在团队项目里新建了一个文件——编写了几百行代码，然后在文件顶部打上自己的名字和适当的版权信息，把它送去代码审查，然后提交给代码仓库。到目前为止一切正常。现在假设你的同事阿德里安跑过来修改了一下文件，那么他要改多少才有资格把自己的名字写上去呢？一定要修复一个bug以后？还是至少要五个bug？一定要实现一个函数？还是需要两个函数？要写多少行代码才够呢？如果他写了一个函数，然后在文件上加上自己的名字，结果这个函数又被后来的人重写了呢？这个人能不能也把她的名字加上去？是不是要把阿德里安的名字拿下来？和其他联合创作（比如剧作、小说、电影等）不同，软件即使在“完成”后依然会不断发生变化。因此电影可以在结尾放心地列出创作人员，可在源文件上这样添加、删除名字的做法却是一件永无止境的疯狂举动。

你当然可以通过大量文档来解决这些问题，但是维护、跟踪、防止意外发生都要浪费很多时间——这些时间原本都可以用来写代码的。因此我们推荐在项目层面而不是代码上认可大家的工作。如果需要更多的细节，可以在版本控制系统

里找到答案。一切都会随时间散去，就像雨中的泪滴[21]。

追求个人“地盘”是很容易犯的错

每个提交都必须经过代码审查

如果打算要实行某种编程标准，那么就必须要有监控哪些代码可以成为产品一部分的方法。无论是在提交之前还是之后进行代码审查，都应该保证每一行进入仓库的代码都要有作者之外的

人检查过，检查的内容有风格、质量，当然还有粗心大意的错误。代码改动应该尽量短小以保证审查的质量——若改动涉及几千行代码，那么除了挑挑格式的毛病外，基本是没办法进行审查的。做到这一点不但能提高整个代码库的品质，更能从长远上培养代码质量的集体荣誉感。

真正的测试和发布流程

无论你是采用全套的测试驱动开发模式还是只有一些简单的回归测试，自动化程度越高，你在修复bug和添加新特性的时候就越自信。决定好测试所要扮演的角色后，它在编程和审查阶段都应该占有一席之地。发布流程也是一样重要，它应该方便到可以频繁发布的程度（比如每周一次），同时也要具备相当的完备性，这样才能在用户之前发现问题。

说到底真正重要的还是代码本身

虽然这些文化和沟通的习惯看起来可能只是代表了笔者自己所偏好的工作方式，但其实它们没有你想象得那么主观。我们发现，只要在组建团队时为它培养强大高效的团队文化，并且在团队沟通上花点时间精力，这样的团队就会有更多的时间编写和发布产品，而不用老是去争论要写什么代码的问题。

强大的团队不是自发形成的，它们都是由团队的领袖和创始人培育起来的，他们对领导废柴团队编写软件所需的代价都有切身体会。所以从一开始就着手培养对创建自我选择的文化是大有裨益的，这样团队才有更多的时间设计和编写代码，而不用去定义或是维护自己的文化。这些努力（沟通和流程）还有一个很大的好处，即它能极大地降低新人融入团队的门槛。不然的话，新人要么吃力不讨好地去学习团队的工作方式，要么就干脆放弃学习，然后试图让团队采用自己在之前团队采用的工作方式（结果可能有好有坏）。

尽管为团队招募到合适的人才和为团队注入正确的价值观都是非常重要的事情，但最后绝大部分能真正成为文化一部分的努力其实都是来自沟通。任务宗旨、会议、邮件列表、在线聊天、代码注释、文档，乃至决策过程都是团队自己以及和外部沟通的不同方式。很多人都想不到只是为了写代码就要在沟通上花那么多时间和精力（包括感情上的交流），但这却是事实。代码最终是要和人沟通，而不是机器。

无论你的团队文化如何，也不管你的团队沟通有多顺畅，我们见过的每一支高效团队都少不了一个领袖。在下一章，我们要谈谈怎样才算是强力的团队领袖，为什么他的角色可能和你想象

的不同，以及为什么工程师懂一点带领团队的基础知识是非常重要的等话题。

注释：[1] 译注：culture这个词有“文化”的意思，也有“培养物”的意思。这里作者玩了个文字游戏。表示文化是需要潜移默化的意思。文中在作比喻时将相关的词打乱在一起混用，为了行文方便，译文里全部改成直白的话。

注释：[2] 当然啦，再强势的团队文化也应该能够接纳新人带来的“好习惯”。

注释：[3] 傅攀勃头一次拜访Google匹兹堡分部的时候被吓得不轻。译注：这是孩之宝的一种枪械玩具。

注释：[4] 译注：不是简单地少数服从多数，具体请参见维基百科。

注释：[5] 有的人认为只要雇佣一个超级架构师，再配几个普通程序员就可以做出好产品了。我们承认这的确是可行的，但是和一群能激发你的灵感、挑战你、教导你的优秀同事一起工作比起来，这种方式实在是太无聊、太无趣了。

注释：[6] 优秀工程师还需要有优秀团队负责

人的领导，因为糟糕的领队不但会在和优秀工程师打交道的时候显得不自信，通常还是那种喜欢瞎指挥人的家伙。

注释：[7] 译注：指盲目乐观。

注释：[8] 在无法达成共识的情况下，有些团队会将决定权交给负责人，而有些团队则会采取投票的方式。采用什么方式并不重要，重要的是要事先决定好在意见不同时采用什么方式来解决它，并贯彻之。

注释：[9] 也就是集体荣誉感。

注释：[10] 参见苏珊·凯恩在TED上的出色演讲——“自省的力量”（http://www.youtube.com/watch?v=c0KYU2j0TM4）和她的著作《安静：自省的力量》。

注释：[11] 这一点我们要一再强调——保持专注的方法就是要拒绝各种诱惑。

注释：[12] 我们经常把编写开源软件比喻成在弹跳床上用扑克牌搭建房屋。这需要稳健的双手、大量的耐心，还有能大声制止那些看也不看就想往下跳的人的决心。

注释：[13] “让GWT更出色”可以在这里找到，http://code.google.com/webtoolkit/making-gwtbetter.html。它绝对是任务宗旨的写作典范。

注释：[14] 译注：原文是“make time”，即工匠（maker）专注工作的时间。这里的工匠一词泛指那些从事创造性活动的人，比如程序员、作家等。

注释：[15] http://www.paulgraham.com/makersschedule.html

注释：[16] 这个传统是Google工程部的副总裁韦恩·罗辛在2001年建立的，旨在改善工程师的生活品质。傅攀勃多年来一直都坚持不在周四开会，效果相当不错，但是这需要非常严格的执行力，如果有人不识相的话，有时候还可以写E-mail去大骂一通。

注释：[17] 所谓“嘈杂的少数人”是指少数的那一两个人，他们会回复每个帖子，反驳每个他们不同意的观点。这些帖子乍看之下似乎讨论得热火朝天，其实一共就那么一两个牢骚满腹的人在上窜下跳而已。你需要及时地小心处理这种情况（第四章会详细讨论对付这种人的办法）。

注释：[18] 当然，在需要不受打扰，或是没办法切换工作内容的情况下，忽略聊天也是完全可以接受的。

注释：[19] 达斯汀·伯斯维尔和特拉佛·佛切尔在《编写可读代码的艺术》一书中对注释有一节精彩的论述。

注释：[20] 本文以及其他风格指南可以在这里找到：http://code.google.com/p/google-style-guide/。

注释：[21] 这是1982年的电影《银翼杀手》里罗伊的台词。

第三章 大海航行靠船长

就算你曾经对天发誓自己永远不会去当“经理”，可到了职业生涯的某个阶段，你一定会有意无意地担当一些领导职务。本章的目的就是要帮助你理解当这个时候来临时该怎么办。

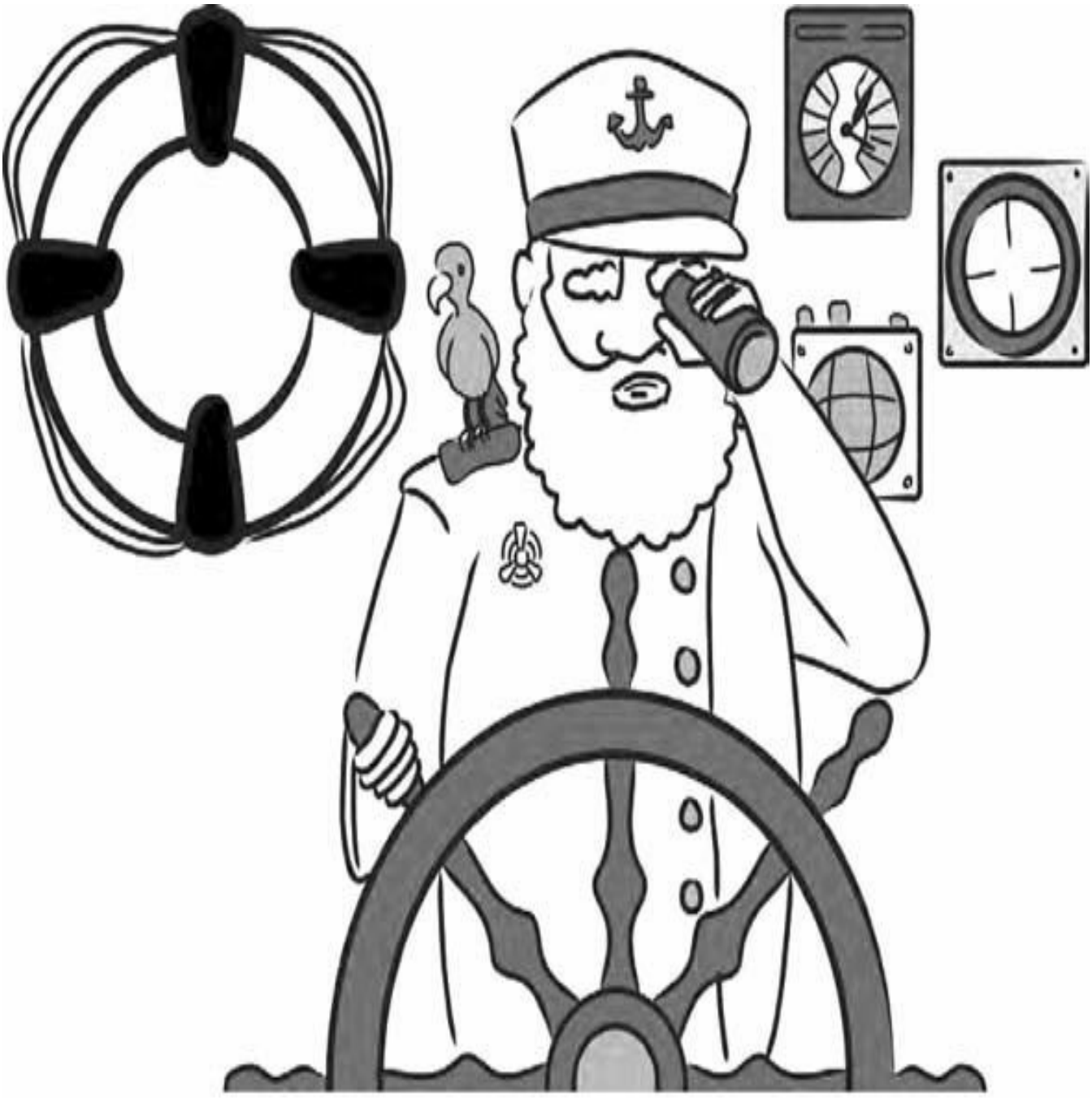
市面上专门写给经理们看的管理书有很多，而本章却是专门写给那些客串管理的工程师看的。出于各种理由，很多工程师都对当经理这件事感到害怕，但是群龙不可无首啊。我们并不是要说服你去当经理（虽然我们两个现在都是工程主管了），而是要告诉你为什么团队需要领袖，为什么工程师常常都惧怕当经理，以及为什么领导最好以谦虚、尊重和信任的原则为基础来服务团队。此外，我们还会探讨一下领导过程中的模式、反模式，以及激励模式。

对于影响软件开发的方向来说，理解工程领导学的方方面面是至关重要的技术。如果你想要主导自己的产品而不只是当一名随行者的话，（那你）就必须学会掌舵，否则只会令自己（还有项目）搁浅。

自然界没有真空地带

没有船长的船和一间浮在海上的等候室没什

么区别——如果没人站出来执掌轮舵启动引擎的话，它只会漫无目的地随波逐流罢了。软件项目就和船一样：如果没人领航，那群极客只会闲坐在那里无所事事。



大海航行离不开船长，每个团队都需要领袖

若想要项目有所进展，就必须要有人站出来领导大家，不管是不是有正式任命。如果你有目标有野心，这个人说不定就是你。你会突然发现自己总是被叫去帮忙解决矛盾、制定决策，或者协调人手，而且都是不经意之间发生的。你从来没想过要当一个“领袖”，但它就是发生了。有些人将这种苦恼称之为“管理炎症”[1]。

@Deprecated Manager

今天的那种尖发老板[2]的形象部分是源自军队阶级，后来又在工业革命时期[3]被传承下来——历时百年哦！当时工厂如雨后春笋一般四处兴起，它们需要大量（通常是没有技术的）工人来维持流水线的运作，因此就需要有监督人员来管理这些工人。由于要替换这样的工人成本很低，加上劳动力过剩，经理往往不会在意雇员的感受，也无意为他们改善什么条件。先不论这是否人道，反正这在机械重复劳动的年代里的确是行之有效的管理模式，而且也延续了好多年。

经理对待员工的办法通常和赶骡车的马夫用的差不多：前面吊着一根萝卜以示激励，如果行不通，那么就用棒子抽。这种“萝卜加大棒”的管理策略从工厂一直延续到了现代办公室，20世纪中叶的时候这种典型的强硬派马夫经理到处都

是，那时候的员工会在同一岗位上工作好多年（当然他们也指望着自己的退休金）。

尽管无数研究表明这种过时的萝卜加大棒理念效率很低[4]，甚至会伤害工程师的生产力，但是这种现象到今天在某些行业里还是依然存在——包括那些需要创造性思维和解决问题的行业（比如工程）。以前的流水线工人只要几天培训就可以上岗，人员替换也很随意，但是工程师却需要数月的时间才能适应新团队的节奏。流水线工人可以熟能生巧，工程师却需要培养，需要有足够的时间和空间来进行思考创造。

主管才是新的经理

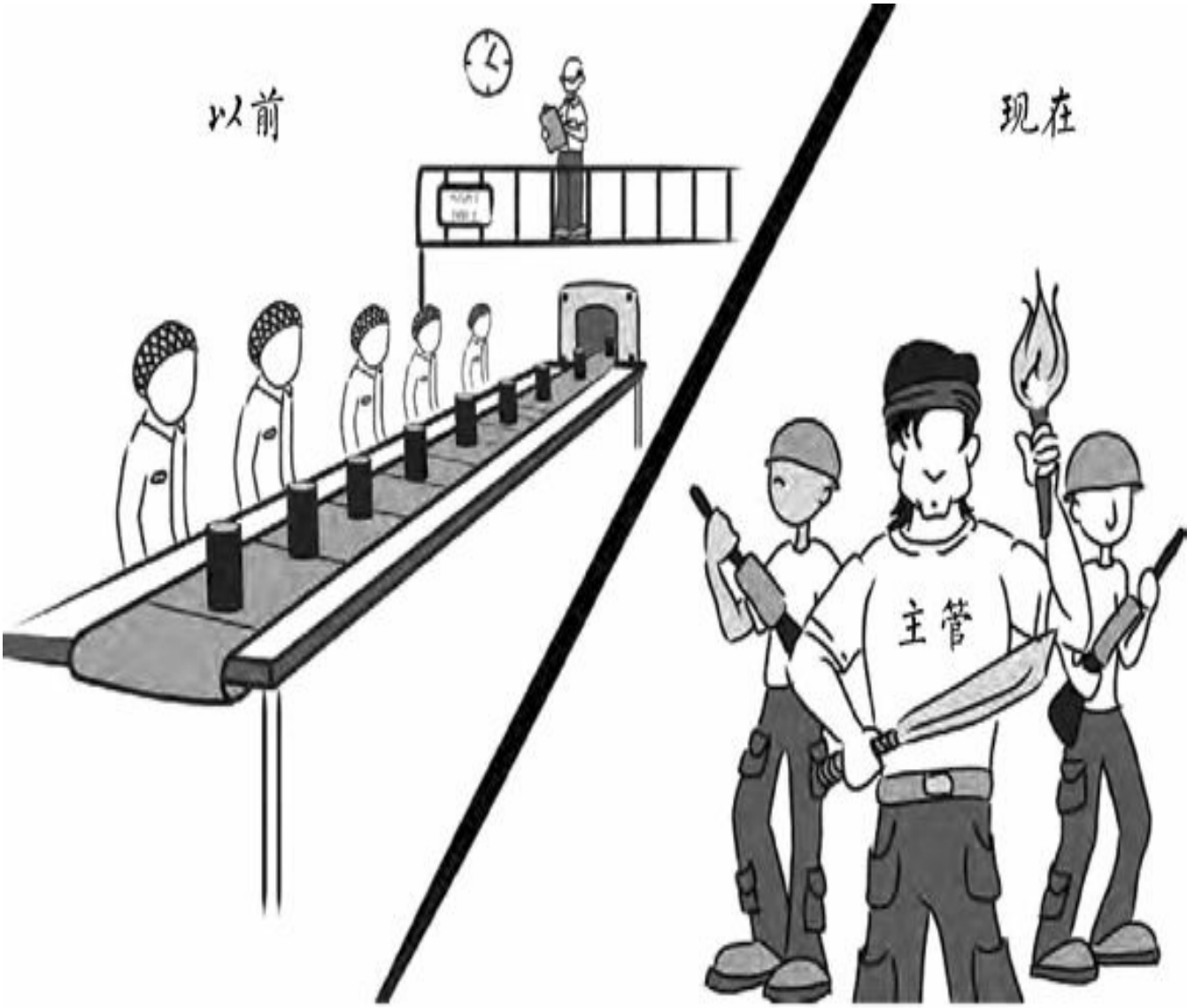
在工程行业里，尽管经理已经是一个不合潮流的头衔，但是还是有很多人在用。而我们认为应该废弃经理（manager）这个词，用主管（leader）取而代之比较好。

尽管很难说我们是共和党人[5]，和政治正确也搭不上边，但是对我们来说经理这个词已经是骂人的话了——这种职位除了写报告啥都不干。经理就像家长[6]，工程师则会像孩子一样回应。若将之代入HRT，那么就是：经理若能表明对手下员工的信任，员工就会感受到正面压力，并努力让自己对得起这份信任。事实就这么简单。主管为

团队铺好道路，为他们解决各种后顾之忧，同时还要设法满足员工的需要。如果要说读完这一章你学到了什么，那么记住下面这句话：

传统型经理关心的是怎么完成任务，而主管只关心完成了什么任务……（并且相信团队能自己想出解决问题的办法）。

几年前傅攀勃的团队里曾经新来了一名工程师。杰瑞的前任经理（另一家公司的）是一个非常刻板的人，要求杰瑞每天必须朝九晚五准时上班，否则就认为他没有努力工作（这当然是非常可笑的啦）。上班第一天，差不多下午四点四十分的时候，杰瑞找到傅攀勃，他显得非常为难，连声抱歉说要提早15分钟离开，因为他有一个约会没办法改时间。傅攀勃微笑地看着他，非常坦率地说："只要你每个星期能全心投入工作75个小时[7]，我才不在乎你几点下班呢。"杰瑞愣了一下，盯着傅攀勃看了几秒，然后咧嘴笑了出来，他答道："太棒了——比起上一份工作来，我会有更多自由时间啦！"傅攀勃把杰瑞当作成年人来对待，而杰瑞也总是能按时完成任务，从来不需傅攀勃担心他的工作，因为杰瑞"不需要"保姆盯着他。



经理要关心怎么干，而主管只负责设定大方向。

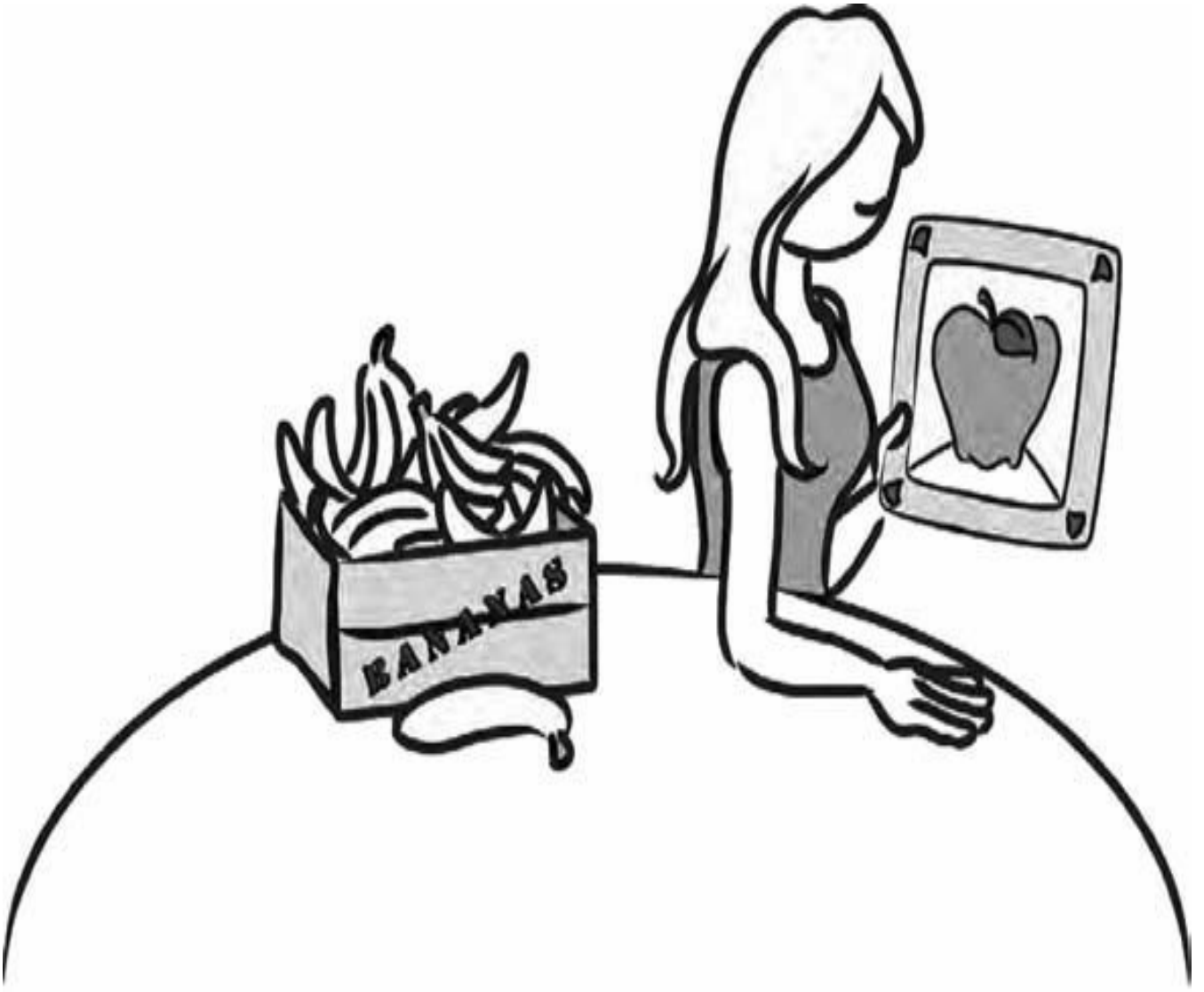
成为“主管”并不意味着你要为所有的事情负最重要的责任。领导的方法有很多，有的偏技术，有的偏情感。Google为领导团队的人准备了两种截然不同的职位（和头衔）：TL（技术主管，techlead）和TLM（技术主管经理，techleadmanager[8]）。TL通常负责产品整体（或者部分）的技术走向，而TLM除了负责产品的整体（或者部分）技术走向外，还要关心手下工程师

的职业发展和愉快程度。这样就可以把那些不想涉及主管工作中人员管理那部分琐事的工程师解放出来，让他们可以专心带队做产品。

唯一要担心的就是……好吧，所有的事情

除了大多数工程师在听到“经理”这个词时感到的不适外，还有其他一些原因致使大多数工程师都不想当经理。其中最常听到的原因就是写代码的时间变少了，这确实不假，不管是技术主管还是人员主管都是这样。不过这个问题稍后再讨论，首先我们来看看其他一些让人不想当经理的原因。

假设你的职业生涯到目前为止都是在写程序，那么不管是程序还是设计文档，或是刚刚修复的一堆 bug，反正每天总能交出一点东西来说，“我今天做了这些”。可根据这种标准，在忙了一天的“管理”之后，你会觉得，“我今天好像什么事也没干”。其实这就等于说是多年来你每天都会数一数今天摘了多少苹果，有一天换了份摘香蕉的工作，结果一天下来跟自己说“我今天一个苹果也没摘”，完全无视旁边那一大堆香蕉一样。量化管理工作比数数要困难得多，而你也不用把团队的成果据为己有，相反，让他们开心有动力才是你的主要工作。



作为主管，不要忘记你创造的是不同的价值

另一个让人对当经理这件事避之不及的原因是人人讳言、但又很著名的“彼得原理”，这条原理说道，“在等级制度中，员工会趋于提升到他无法胜任的地位上去”。大多数人都遇到过不能胜任自己工作，或是真的不会管人[9]的经理，我们知道有些工程师从来就没有遇到过好经理。要是你一直都在坏经理手下做事，那么你怎么会有要当经理的念头呢？为什么你会想要升职去担任自

己干不了的职位？

当然也有很多很好的理由让你去考虑当一名经理。首先，这是拓展自己的一个方式。就算你的程序写得非常好，你能写的代码量仍然是有限的。想象一下，如果你能领导一支由优秀工程师组成的团队，那代码的产量能达到多少！还有就是说不定你很有领导天分呢——很多被赶鸭子上架的工程师发现自己其实很擅长提供指导帮助，或是为团队打掩护。

仆人式领导

新晋经理似乎往往会忘记当初自己遭受过的待遇，突然开始用同样的手段来“管理”自己的手下。其表现包括但不限于微管理，无视表现不好的人，以及只雇佣那些听话的人。如果不及时治疗，这种病能毁掉整个团队。我们刚刚当上Google工程经理的时候，工程主管史提夫·温特给了我们一条非常棒的建议。他说：“千万别新官上任三把火。”新晋经理会非常积极地“管理”自己的手下，毕竟这就是经理的工作嘛，不是吗？可这样做的后果往往都是灾难性的。

要治好这种“管理”病，就应该采用宽松的方式，我们称之为“仆人式领导”，这个名字很好地表达了经理最重要的工作就是为团队服务，如同

管家之于一个运转良好的家庭一样。作为仆人式领导，你应该努力去营造一种谦虚、尊重和信任（HRT）的氛围。这意味着消除工程师自己无法消除的像官僚作风这类障碍，帮助团队达成一致，甚至还包括加班的时候帮大家买晚饭等。仆人式领导要为团队填补前进道路上的裂缝，并在必要的时候给予建议，同时还要勇于冲到第一线。仆人式领导唯一要做的管理工作就是对团队的技术和人事健康状况负责。尽管完全把注意力放到团队的技术层面上是个很有诱惑力的想法，但是人事方面的状况也是同样重要的（但管理起来往往要困难无数倍）。

反模式

在讨论成功领袖所采用的一系列“设计模式”之前，我们先来看看那些阻碍你成功的模式。这些具有毁灭性的模式都是我们从自己的职业生涯中遇到过的那些糟糕领袖身上总结出来的，还有不少傻事是我们自己干过的[10]。

反模式：雇佣听话的人

如果你是经理，却对自己的职位缺乏安全感（不管是什么原因），那么有一种办法可以保证没人会质疑你的权威或者威胁到你的工作，即雇佣你能掌控的人。只要在招聘的时候专门雇佣那

些不如你聪明，没什么野心，或者比你更缺乏安全感的人就可以了。不过这样做虽然可以确保你作为领导和决策者的地位无忧，却也意味着你自己的工作量增加了。这样的团队如果没有你带领，是没有办法做成任何事的，就像拴着皮带的狗一样。所以如果你搭建起这样一支听话的团队，那么你大概就没办法休假了。因为只要你一离开办公室，生产力就会直线下降。但这毕竟可以让你在位子上坐得安稳一些，不是吗？

其实，你应该努力去雇佣那些比你聪明、可以替代你的人。这对很多人来说难以接受，因为这些人会经常挑战你（还会在你出错的时候毫不留情地指出来）。但同样是这些工程师，他们还会不断地给你惊喜，交出漂亮的工作。他们能引导自己更上一层楼，有些人还会主动想要去带领团队。你不应该视此为意图篡夺权位，而应该把它看作是让你多领导一支团队的机会，让你可以去探索新的机会，甚至可以放心去度假，而不必每天盯着团队的进度。

反模式：无视表现不佳的人

傅攀勃刚刚在Google当上团队主管的时，每次要给员工发奖金的时候他都会笑哈哈地对他的经理说：“我非常喜欢工程经理这份工作。”傅攀勃的经理是业界的资深老鸟，他总是坚定地答道：“当牙仙当然好，但是有时候也要扮演一下

牙医的角色。”

拔牙总是很让人痛苦的。我们见过很多团队主管辛辛苦苦建立起非常强大的团队，结果就是因为一两个表现不佳的人，使得团队无法更进一步（最终分崩瓦解）。我们都明白编写软件的时候和别人打交道是其中最难的部分，而和人打交道的时候最难的是怎么处理好那些无法达成目标的人。有时候这些人完不成指标只是因为他们花的时间不够或者本身不够努力，但是最棘手的是碰到能力不够的人，这种情况和工作时间够不够长或者是不是努力已经没有关系了。

在Google，负责所有服务正常运行的那支团队有这样一句座右铭：“希望可不是一种策略。”而在处理表现差的人的时候，希望却被当作策略用到滥。大多数团队主管都会咬着牙根、闭着眼睛希望那些表现不佳的人要么自己神奇地产生进步，要么自动离开。可惜事实上这种人很有可能两条路都不会选。

当领导在盲目地奢望，可表现不佳的人却又毫无进步（也不肯离开）的时候，那些表现优秀的成员却要浪费宝贵的时间拽着落后的人前行，团队的士气也因此受到打击。可以肯定的是，就算你无视那些表现不佳的人，大家心里也都跟明镜儿一样——团队里的其他人都清楚地知道谁是短板，因为大家必须拖着他走。

无视表现不佳的人还会妨碍优秀新人的加入，导致牛人离开。结果整支团队里变得只剩下表现差的人，因为他们都是一些没什么意志力的家伙。最终，保住他们的席位其实并不是在帮他们，这种无法在你的团队里发挥作用的人往往会在其他方面产生影响。

尽快处理表现不佳的人的好处就在于你还有机会帮他的忙。假如你立刻出手，常常会发现他只是需要一些鼓励，或是点拨一下就能大大提高生产力。但是如果耽搁久了，他和团队的关系就会变得棘手，你也会因为没办法再帮忙而感到失望沮丧。

那么怎么才能有效地调教表现不好的人呢？我们两个（不幸）在这方面有不少经验，都是从经验教训里摔打出来的。有个比喻再恰当不过，就是帮助一个腿脚不便的人学着再走路，然后慢跑，最后赶上大家的步伐。这肯定要用上一些暂时性的微管理手段——同时还不能忘了大量的HRT，特别是尊重。规定好一个期限（比如两到三个月），然后设定一些你希望他达成的目标。目标要设得小一点，循序渐进，这样才有机会累积很多小成功。每周都要跟他碰个头检查一下进展情况，每个里程碑的目标都要清楚无误，这样才能轻易衡量成功和失败。如果他跟不上，双方也都能在一开始就意识到。这样，他也能认识到

不足，自觉退出。反之，他会鼓起雄心不辜负期望。无论如何，只要手把手地帮助表现不佳的员工，你都可以借此激发必要的重大改变。

反模式：无视人际关系

我们前面说过，团队主管主要关心团队的两个层面：人际关系层面和技术层面。主管在技术方面通常会强一点，因为大多数主管都是从技术职位上提拔上来的（他们的主要工作就是解决技术问题），所以他们会倾向于无视人际关系。把全部精力都放在技术层面上是非常有诱惑力的想法，因为作为个人，能作的贡献就是把大部分时间花在解决技术问题上。在学生时代，你在课堂上学到的都是工程学里技术方面的东西。但是你现在是主管了，无视团队里人的因素只会让自己承担相当大的风险。

就此，我们先来举个例子。傅攀勃的一位好朋友（我们都叫他杰克）的第一个孩子出生了，杰克和傅攀勃共事多年，虽然常常远程工作，但他们同属一个办公室。所以孩子出生后的那个星期，杰克都是在家工作。杰克和妻子觉得这样很好，傅攀勃对此也没什么意见，毕竟他早就习惯和杰克远程合作了。他们的工作效率也一如往常，直到他们的经理帕布罗（他在另一间办公室）发现原来杰克一周来大多数时候都在家工作。杰克没有到办公室来和傅攀勃一起上班让帕

布罗很不高兴，尽管事实上杰克的生产力完全没有下降，而且傅攀勃也完全可以接受这种状态。杰克试着和帕布罗解释，表示自己不在办公室效率也不会降低，而且在家工作几个星期对自己和妻子都好。帕布罗答道："伙计，谁没有孩子啊（别来这套）。你一定要按时来办公室上班。"不用说，杰克（平时的脾气很温和）被激怒了，从此不再尊敬帕布罗。

其实帕布罗原本是有很多更好的方式来处理这件事的：他原本可以对杰克希望在家多陪陪妻子的愿望表示理解，如果他的生产力的确没有降低，团队也没有因此受到影响的话，不妨让他这样继续工作一段时间。他原本可以和杰克商量一下，让他每周到办公室来一两天，直到宝宝和家里安顿下来为止。无论结果如何，只要稍微表示出一点同情心，杰克就会很开心了，长远来讲对大家都有好处。

反模式：和谁都是朋友

大多数工程师在进入领导行列后遇到的第一个难关就是他从团队的成员变成了团队的领袖。大多数主管都不想失去和团队之间的友谊，因此他们在成为主管后有时会格外努力地去维护它。这其实是灾难的源泉，很多友谊破裂都由此开始。千万不要把友谊和怀柔搞混了：当你掌握了别人的生计时，他可能会感受到压力，主动表现

出友好的姿态。

记住，在领导团队以及达成共识的时候并不需要站到团队的对立面去（或是搞一言堂）。同样，你可以在成为强悍的领袖的同时保住和大家的友谊。我们发现和大家一起吃午饭是拉近距离的好办法，还不会让大家不舒服——这样你就有机会在正式的工作环境之外聊一些随性的东西。

有时候变成好朋友的老板是一件很微妙的事情。假如他管不好自己，不努力工作的话，大家都会觉得有压力。我们建议你最好还是尽可能地避免这种情况。

反模式：降低招聘标准

史提夫·乔布斯曾经说过："顶尖的人会雇佣和自己一样优秀的人才，而差一点的人只雇得到更差的人。"在招聘过程中一不小心就会在这句格言上栽跟头，特别是求贤若渴的时候很容易饥不择食。我们见过很多团队在招人的时候会采用这种办法，比如他们需要五名工程师，因此就从一大堆申请里挑出 40 到50人来面试，然后留下五个最好的，而不管他们是不是达到了招聘的标准。这可以说是建立平庸团队最快的方法之一了。

找到对的那个人的成本（不管是付给面试官的钱，还是花在广告上的钱，还是寻求推荐的费用）比起招到一个不应该招的员工的代价来绝对

是微不足道的。后者包括了团队丧失生产力，导致团队压力，耗费时间去管理这名员工，以及解雇员工过程中涉及的各种书面工作和压力等。当然，我们假设你在遇到这种事情的时候，不想出现把他丢在一边任他侵占团队的成本的情况。假如你在自己管理的团队里，在招聘上没有发言权，而你又不喜欢招进来的人，那么你一定要尽一切力量抢到最好的工程师。要是招进来的仍然是水平不怎么样的人，那你还是想办法另谋高就吧。没有人才是打造不出顶尖团队的。

反模式：把团队当小孩子

把团队当小孩子，就能清楚地传达你不信任他们这样一个信息——通常你怎么待人，他们就会怎么处事。所以如果你把他们当成孩子或者犯人，那么别吃惊，他们的行为真的会像孩子和犯人一样。比如过多地干预他们的工作，或者完全不尊重他们的能力，在工作上不给他们任何机会去承担责任的话，其结果必然如此。如果因为无法信任对方，而只能不断地去干预他们的工作，那说明你在招聘的时候就已经犯错了。好吧，除非你的目标就是一辈子手把手搀着自己打造出来的团队。只要你雇佣的人值得信任，并且表现出你对他们的信任，那么他们通常会付出额外努力来对得起这份信任（正如前面提到的，雇佣优秀人才是要坚持的基本前提）。

傅攀勃在芝加哥的一个本地机构租了个场地来办研讨会。他头一次去的时候，负责接待的经理带傅攀勃转了一圈，好让他熟悉一下环境，随后他把钥匙交给了傅攀勃，约定一周后取回。整个过程中没有提到任何有关“什么可以做”和“什么不能做”的话题，也没有什么干预。结果傅攀勃和他的团队把场地当成自己的地方一样，发挥主人翁精神来保持整洁有序，大大超出了预期。

这种信任完全可以从大楼钥匙延伸到办公室和电脑设备。再举一个例子：Google提供了各种各样的办公用品（诸如笔、笔记本等“传统”物品），都陈列在柜子里任需取用。我们的IT 部门有好多“技术站点”，就是好像迷你电器商店一样的自助服务区域，那里有各种电脑配件（比如电源线、网线、鼠标、USB 等），同样可以随便拿，但是我们因为得到了信任，所以感到有责任要“做正确的事”。很多来自传统公司的人在听到我们的做法时都反应激烈，直呼我们肯定会因此大出血，因为人们肯定会“偷拿”这些东西的。当然这也是有可能发生的情况，但是和拥有一群像小孩子一样的员工的成本相比呢？肯定要比几支笔和几个USB贵多了吧。

领袖的处事之道

下面是一系列成功领袖所遵循的行为模式，它们来自我们的经验总结，偷师其他成功领袖的心得，还有很大一部分是我们自己的导师教导的。这些模式不但我们自己成功采用过，它们也是那些我们一直都非常尊敬的精神领袖所采用的模式。

放下自负

我们在第一章讲到HRT的时候就已经讨论过“放下自负”这个话题了，身为仆人式领导，这一点尤为重要。人们常常会对这种事情产生误解，觉得这样一来领导就会变得无足轻重，最后被团队骑到头上来，事实上根本就不是那么回事。我们承认表现得谦虚一点和被人欺负之间的界线有点模糊，但是谦逊绝对不是缺乏信心的表现。你不需要表现得像个以自我为中心的疯子，同样可以自信、有主见。对任何团队（特别是团队主管）来说，搞个人主义都是没有市场的。努力培养团队的骄傲感和认同感才是正确的方向。

“放下自负”里有一部分内容是我们已经讨论过的，即你应该信任自己的团队。这意味着尊重团队成员的能力，以及他们之前的成就，哪怕他们是刚刚加入团队也不例外。

假设你没有在团队里实行微管理，那么在一线工作的同事们肯定比你更了解他们在做的事情。这就是说虽然你要负责让团队达成共识，帮

助他们设定方向，但是具体怎么完成任务则最好是由这些做产品的人来决定。这样不但能激发他们的主人翁精神，更能让他们对产品成功（甚至失败）有更大的责任感。如果你有一支优秀的团队，并且放手让他们去设定质量以及评价自己工作的标准，那么他们的表现一定会超过胡萝卜加大棒下的表现。

很多刚刚担任领导职务的工程师总是迫不及待地想要立刻做好所有的事情，了解所有的事情，回答所有的问题。我们敢保证，你不可能把每件事都做对，也不可能知道所有问题的答案，而且假如你硬要如此表现的话，很快团队就不会再尊重你了。很多时候其实这都和职位的安全感有关。回想一下你自己还是工程师的时候，对这种事情肯定是很敏感的吧。所以不妨尝试一下多问问题：当别人对你的决定或言论产生疑问时，别忘了对方往往只是想要更多地了解你罢了。如果你能鼓励这种提问，你才有更多的机会听到诤言，这能让你成为一个更好的领袖，团队也会变得更成功。诤友难寻，而要从下级那里听到这种批评就更是难得。所以从大局上考虑一下你希望打造什么样子的团队，以开明的态度对待反馈和批评，千万要克制住搞一言堂的冲动。

放下自负的最后一个方面非常简单，然而很多工程师宁可下油锅也不肯那么做，（这最后一

个方面）就是为自己的错误道歉。这并不是说你只要像在爆米花上撒盐那样在交谈中点缀一点“对不起”就可以了——道歉必须是真心实意的。犯错肯定是无法避免的，而且不管你承不承认，你的团队都会知道你犯错了。不管有没有告诉你，他们肯定是知道的（而且有一件事是肯定的，那就是他们会跟别人聊你犯的错）。道歉又不要钱。人们对有勇气道歉的领袖反而会倍加尊敬，和广泛的认知相反，道歉并不会降低你的威信。事实上，道歉往往会赢得别人的尊重，因为大家会因此觉得你是个头脑清醒的人，善于对事物作出评估，而且（回到HRT上来）还很谦虚。

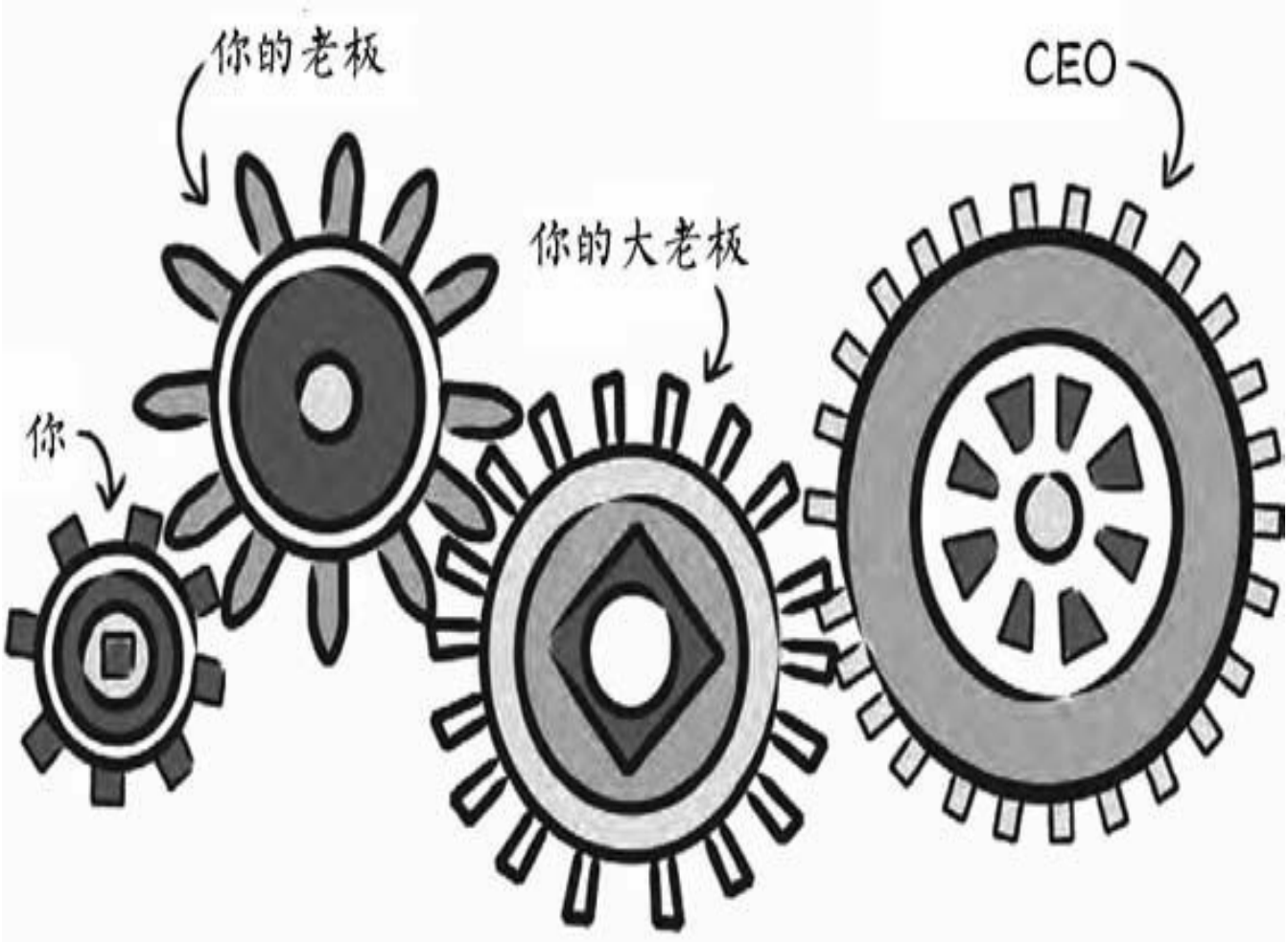
做一个禅师

身为工程师，你可能职业性地变得多疑和愤世嫉俗，但这对于带领团队来说却是不利的。这不是说盲目乐观是对的，只不过在告知团队你已经意识到工作中要面对的纷乱和障碍时，如果能在言语上少一点疑惑就好了。领导的人越多，保持淡定和冷静就越是重要，因为众人都会（不管是不是有意识地）看着你，看你在面对事物时的态度和反应是怎么样的。

口母~~

只要把你公司的组织结构图想象成一系列齿轮，就可以轻松地看到它的效果。写代码的工程师是一边那个小小的齿轮，只有几个轮齿，他上面每一级的经理都是一个更大一点的齿轮，而最

末端的大齿轮是CEO，拥有几百个轮齿。这样，工程师之上的“经理齿轮”（可能有十几个轮齿）转一圈，“工程师齿轮”就要转两到三圈。而CEO只要动一点点，就能让六七层之外的工程师齿轮飞速旋转！你转动的齿轮离底端越远，就能让下面的齿轮转得越快，不管是不是本意如此。



理解公司里的齿轮比

傅攀勃之前有个经理名叫比尔[11]，他的绝活就是任何时候都能保持冷静。不管发生了什么，也不管事情变得多糟糕，哪怕火烧眉毛，比尔也

没有慌过神。大多数时候他都会一手插在怀里，另一手托着下巴，看着一个已经完全不知所措的工程师问他到底发生了什么问题。这样一来那名工程师也会渐渐冷静下来，帮助他把思路集中在解决问题上，而不是像无头苍蝇一样不知所措。傅攀勃曾经开玩笑说，如果有一天有人跑来告诉比尔说我们有19个数据中心被外星人攻击了的话，他的回答也只会是："你知道为什么他们不干脆凑个整数，攻击20个吗？"

这引出了禅式管理的另一个秘诀：提问。当队员来向你寻求建议的时候，通常都是很让人兴奋的，终于有机会来修复点什么东西了！在担任领导职务前，你已经在这行干了好多年了，所以很多时候你会直接跳到解题模式，但是其实最不应该的就是这么做。工程师来问你建议通常不是要你去解决他的问题，而是要你帮助他解决问题，所以最简单的方法应该是问问题。这不是说要你把自己变成魔法八号球[12]，这反而会适得其反。正确的做法应该是在HRT的原则下，帮助他解析分析问题，从而达到让他自己解决问题的目的。这通常能引导工程师得出答案[13]，最重要的是，这是他自己想出来的答案，因此也就回到了本章开头所讲的主人翁精神和责任感。不管你是不是知道答案，这种办法几乎总是能让工程师觉得其实你早就知道答案了。是不是很难相信？苏

格拉底会为你骄傲的。

成为催化剂

化学中的催化剂是一种能加速化学反应的物质，其本身并不会在化学反应中被消耗。催化剂的原理之一（比如，酶）就是将反应物相互拉近：而不是让它们在溶液里随机地弹来弹去，当催化剂将反应物之间的距离缩短的时候，反应物就能更顺利地相互作用了。身为领袖，你经常也要扮演这样的角色，而且方法有好几种。

团队主管最经常要做的事情之一就是引导大家达成共识。这表示你可能需要从头到底主导整个过程，也可能只要在正确的方向上稍稍推一把就行了。设法帮助团队达成共识是非正式领导经常会用到的一种领导技能，因为这样就可以在没有正式认命的情况下带领团队。假如手握大权，你可以采用强硬的手段来主导，但是总体来说这样做比不上引导大家达成共识的方法。在团队需要灵活机动的情况下，有时候他们也会自愿听从指挥，接受领导。这看起来可能有点像独裁或是寡头政治，但是在自愿的情况下，也算是达成共识的一种。

有时候你的团队已经就下一步的行动达成了共识，结果却碰到了障碍。这可以是技术上的障碍，也可以是组织性的障碍，不管是哪种，出手帮助团队越过障碍继续前进是必不可少的领导技

能。有些障碍对于你的队员来说几乎无法逾越，对你来说却是小菜一碟，让你的团队了解到你帮助他们解决障碍的意愿和能力是非常有价值的事情。

曾经有一次，傅攀勃的团队想要解决一个和公司的法律部门之间的纠纷，为此耗费了好几个星期的时间。最后黔驴技穷只能去找傅攀勃，他在两个小时之内就摆平这件事了，因为他知道要去找谁。还有一次，本的团队需要一些服务器，可就是弄不到。幸好本当时和公司的其他团队一直都保持联络，结果当天下午就帮他们搞定了。再举个例子，傅攀勃团队里有个工程师在一段晦涩的Java代码上遇到了麻烦，虽然傅攀勃自己不是Java专家，但是他认识的人里有高手可以解决问题。在帮忙扫除障碍的时候，你用不着通晓一切，往往认识能解决问题的人就足够了。很多时候认识正确的人比知道正确答案要有价值得多。

催化团队的另一种方法是给予他们安全感，这样他们就愿意接受更高的风险。风险是一种很迷人的东西——大多数人都很不会判断风险，而大多数公司都会不惜一切代价规避风险。因此常规的运营方式都是非常保守的，专注于小富即安，哪怕高风险可能意味着指数级的巨大成功。在Google我们常说的一句话就是，若明知不可为而为之，那么成功的机会肯定是渺茫的，但即使

失败了，你的收获也比只尝试那些你知道自己肯定能完成的事情所得到的要多得多。如果要培养起敢于冒风险的氛围，就一定要让团队明白，失败没什么了不起的。

我们首先来看看这句话：失败没什么了不起的。事实上我们倾向于把失败当成是一种快速学习的方法，只要你不重复自己的错误就行。此外，将失败视为学习的机会是非常重要的，失败不是让人相互推卸责任，相互指责的。快速失败其实是好事，因为这样代价就不会太大[14]。虽然慢慢失败也能得到很深刻的教训，但是也会痛苦很多，因为风险担得越多，损失也就越大（通常是工程上的时间）。会影响到用户的失败应该是我们最不希望遇到的失败，同时也是我们从失败中学习最好的时机。就像前面提到的，在Google，每次在生产系统上出现问题的时候，我们都会进行事后检讨。这个流程会记录下究竟什么事情导致了这一结果，并发展出一系列步骤，防止未来再次发生这样的错误。这绝不是给大家相互指责的机会，也无意引入不必要的官僚程序，而是要把注意力全都放在问题的核心上，一劳永逸地解决它。很难，但是很有效（还非常解恨）。

个人成功和失败的情况则稍有不同。称赞一个人的成功是一回事，但是在遭受失败的时候指

责个人则不利团结，而且会从根本上阻碍承担风险的意愿。失败不要紧，就算失败也要像一个团队一样，并且从中吸取教训。个人的成功可以在众人面前表彰，但个人的失败最好还是私下检讨吧[15]。无论如何，本着HRT的原则，抓住机会帮助团队在失败中成长吧。

当一个导师

身为团队主管，最难受的就是看着一个新人在一项任务上花了三个小时，而你自己出手的话可能只要二十分钟就够了。刚开始的时候，指导你的手下，给他们自学的机会是非常困难的事情，但这却是优秀领导必不可少的素质。这对于新人来说尤为重要，他们不但要学习团队的技术和代码，还要熟悉团队的文化以及承担责任的界线。

大多数工程师不会主动要求担任导师的角色——他们常常是在新成员加入的时候，受主管委托来指导他们。成为导师不需要接受什么正式的培训，也不用做太多的准备工作。事实上，你只要具备三个条件就基本可以胜任了：熟悉团队的流程和系统，向他人解释事物的能力，以及估计被指导的人到底需要多少帮助的能力。最后一个条件或许是最重要的——你要做的就是给你的学生提供足够的信息，但是假如你解释得太多，或者无休止地东拉西扯，那么你的学生也未必会领

情，他可能会直接忽略你，而不是礼貌地告诉你他已经明白了。

设置明确的目标

这一点似乎有点显而易见无需赘言的感觉，但结果无数的主管、领导、负责人都忽略了它。团队如果想要在一个方向快速前进，你就要让所有人同心协力。我们不妨把产品想象成一辆大卡车（不是一堆轮胎）。团队里每个成员手上都有一根绳子连到卡车前面，好像大家都在做同一个项目一样，他们会在自己的方向上拉动卡车。如果你打算把卡车（或者说项目）尽可能快地向北拉，那么就不能让团队成员想怎么拉就怎么拉——你要让他们都往北拉动卡车。

每个人都在同一个方向上拉吗

设置明确的目标，让团队同心协力的最佳方法就是为他们写一份简明扼要的任务宗旨（参见第二章里“任务宗旨——不，我是认真的”，这一

节详细讨论了任务宗旨的内容）。只要帮助团队设定好方向和目标，你就可以放手给他们更多的自主性，只要定期检查看他们有没有偏离方向就可以了。这样你不但可以有更多的时间来处理其他管理事务，还能大幅提升团队的效率。虽然在缺乏清晰目标的情况下团队也能够取得成功，但是这样通常会浪费很多精力，因为每个人努力的方向会略有差异。这会让人觉得很沮丧，导致团队进展缓慢，还会迫使你耗费越来越多的精力来纠正这些偏差。

坦诚

这并不是说我们认为你在向自己的团队撒谎，这里提到它的原因是有时候不可避免地会出现一些不能告诉团队的事情，或者是你必须要告诉他们一些他们不愿意听到的事情（这种情况更糟糕）。傅攀勃以前有一个经理是这样对他的新成员说的：“我不会对你撒谎，但是我会告诉你有些事情我不能说，或者我确实不知道。”

如果你的下属跑来问一些你不方便说的东西，你完全可以告诉他说虽然你知道答案，但是不能告诉他。更普遍的情况是你也不知道他想知道的东西，那么你不妨直言。这又是一个看似明显的例子，但是很多工程师在当上经理后，担心自己要是答不上来的话，会显得能力不足或是落伍了。但事实上，这只是说明经理也是人罢了。

批评人是非常困难的。当你不得不告诉你手下的工程师他犯了个错误，或者没有达到你的期望时，压力是非常大的，特别是你头一次干这种事的时候。大多数管理学教材的建议是采用所谓的“三明治赞美法”来缓和批评的冲击力。这里是一段三明治赞美法的例子：

“你是团队骨干，是我们最聪明的工程师之一。不过你的代码写得实在是太难懂了，团队里大概没人能看明白。但是你有很大的潜力，而且你的络腮胡太帅了。”

这样气氛就不会那么紧张，不过大多数人都听不懂这种躲躲闪闪的说话方式，他们在离开的时候会想：“太好了！我的络腮胡真的很帅！”我们强烈反对这种三明治赞美法，这不是说你可以无端地欺凌或者苛责别人，而是因为大多数人听不到谈话里关键的部分，也就是那些有关需要改进的部分他们是听不进去的。另外不要忽略HRT：只要表现出亲和力和同情心，无需诉诸三明治赞美法，同样可以提出建设性的批评。事实上，亲和力和同情心是让你的批评对象不会立即表现出防御心态的法宝。

警惕三明治赞美法

傅攀勃在好多年前从另一个经理那里挑走过一个工程师，名叫提姆。当时那名经理对提姆的评价是没办法和他一起工作。他对傅攀勃说，提姆从来都无视对他的反馈和批评，就算告诉过他不要那么做，他依然我行我素。于是傅攀勃就跑去参加这名经理和提姆的会议，看看他们是怎么交流的。他发现这名经理不断地用三明治赞美法来避免伤害提姆的感情。傅攀勃把他调到自己手

下后，找了个机会坐下来跟他讲清楚，他必须作出调整才能更有效地在团队里发挥作用。傅攀勃没有说什么恭维的话，也没有美化问题，当然他也不会无端苛责提姆——他只是根据提姆在之前团队里的表现，把事实陈述出来。果不其然，短短几个星期之内（再加上一些“提醒”性质的会议），提姆的表现就和之前判若两人。他只是需要有个人提点一下，帮他清楚地指出方向而已。

在直截了当反馈或者批评的时候，表达的方式是确保别人听得进你的意思，不会发生偏差的关键。如果对方被你弄得很防备，他是不会思考自己怎么改进的，他只会想着怎么和你争辩，证明你错了。本的手下曾经有个工程师，我们暂且叫他狄恩。狄恩是个很强势的人，无论什么事情，他都能和团队里的其他人争个面红耳赤。大到整个团队的任务方向，小到控件在网页上的位置，狄恩都会固执己见，寸步不让。这种情况持续了好几个月，本只好找他谈话，告诉他说他太争强好胜了。那么，假如本这样说的话：

“狄恩，你不能再这么嚣张了。”

我敢打赌狄恩肯定听不进去。本捉摸了很久，该怎么和狄恩说他才会明白自己的行为已经伤害到了团队呢，最后他用了一个下面这样的比喻：

每次要做决策的时候，我们都会觉得好像要

开火车穿过小镇一样——你会跳到火车前面试图阻止它前进，这样不但减缓了火车的速度，而且可能还会影响到火车司机的心情。假如火车是每15分钟一班，而且每列火车你都不放过的话，不但你自己要浪费大量时间去拦火车，而且最后肯定会有某个火车司机忍受不了，直接从你身上压过去的。所以虽然跳到火车前面不是不行，但还是要慎重一点，把机会留给真正有必要拦的火车吧。

这个故事在荒诞之中穿插了一点幽默，本和狄恩在讨论时，不但指出他浪费了精力，也更能说明他的这种“拦火车”的行为对团队所产生的影响。

记录快乐程度

身为领导，从长远上提高团队生产力（同时减少减员率）的办法就是在评估团队快乐的程度上多加注意。我们合作过的最棒的主管都是业余心理学家，他们会不定时地关心手下员工的福利，确保他们的工作得到认可，并且努力保证他们喜欢自己做的事情。我们认识一个主管，他自制了一张表格，列出了所有没人想干但是一定要完成的脏活累活，然后平均分配给大家。另一个主管则会关注团队在工作上花费的时间，然后穿插一些休息时间和好玩的团队出游，以免搞得大家精疲力竭。还有的主管会采用一对一的方式作

为切入点和手下一起解决技术问题，然后努力确保每个人都有足够的资源来完成工作。等到大家进入状态后，他又会找他们谈话，看他们是不是喜欢手上的工作，下一步想要干什么等。

想要知道团队快乐程度最有用的一招就是每次在一对一会议结束的时候问问你的队员“你还有什么要求吗”。这个问题很简单，非常适合用在会议结束的时候，它能帮助发现每个人保持生产力和心情愉快的需求，当然，你还需要小心探索才能获得更多的细节。只要你坚持在一对一的时候问，最后你的团队一定会记在心里，有事甚至还会主动来找你，告诉你他们需要什么才能把工作做到更好。

在加入Google后不久，傅攀勃和当时的CEO埃里克·施密特开了一次会。会议快结束的时候，埃里克问道：“你还需要什么吗？”傅攀勃自以为准备充分，却被这个问题问了个措手不及。他坐在那里干瞪眼，愣了半天没说话。不过如果还有下次的话，他肯定不会再被问住了！

另外，多关心一下团队在工作之外的情况也是很有必要的。绝对不要以为人人都是工作狂——对别人在工作上能投入的时间不要有不切实际的期望，否则很容易失去别人对你的尊敬，甚至彻底绝望。这绝不是在鼓励你刺探手下的私生活，我们想表达的是，如果知道一些队员私底下

的情况，你就更能了解他们在某段时间里表现出色或者不够专注的原因。如果某人家里有点变故的话，不妨在工作上多给他点时间，这样将来团队要是碰到项目很紧的情况时，他也会更愿意付出以来回报你。

此外还有一个重要的部分就是要关心他们的职业生涯。如果你问你的队员五年后对自己有什么展望，绝大多数时候他们都只会耸耸肩表示没想过。其实这种情况下大多数工程师都不会说太多，但是每个工程师心里总有一些想要在未来五年之内完成的东西，比方说升职、学一点新东西、发布重要的产品，或者是和牛人一起工作等。不管有没有说出来，大多数工程师心里都会有抱负的。要是你想当一个好领导，就应该考虑一下怎么帮忙才能够实现这些愿望，让你的团队知道你真的放在心上了。最重要的就是了解这些没有说出来的目标，让它们显现出来，这样才能在给出职业建议的时候有可以依赖的标准来评估不同情况和机会。

记录快乐程度不但是关注（队员的）职业生涯，更是让你的队员有机会得以成长，让他们的工作可以得到认可，同时还能给整个过程多添几分乐趣。

其他建议和窍门

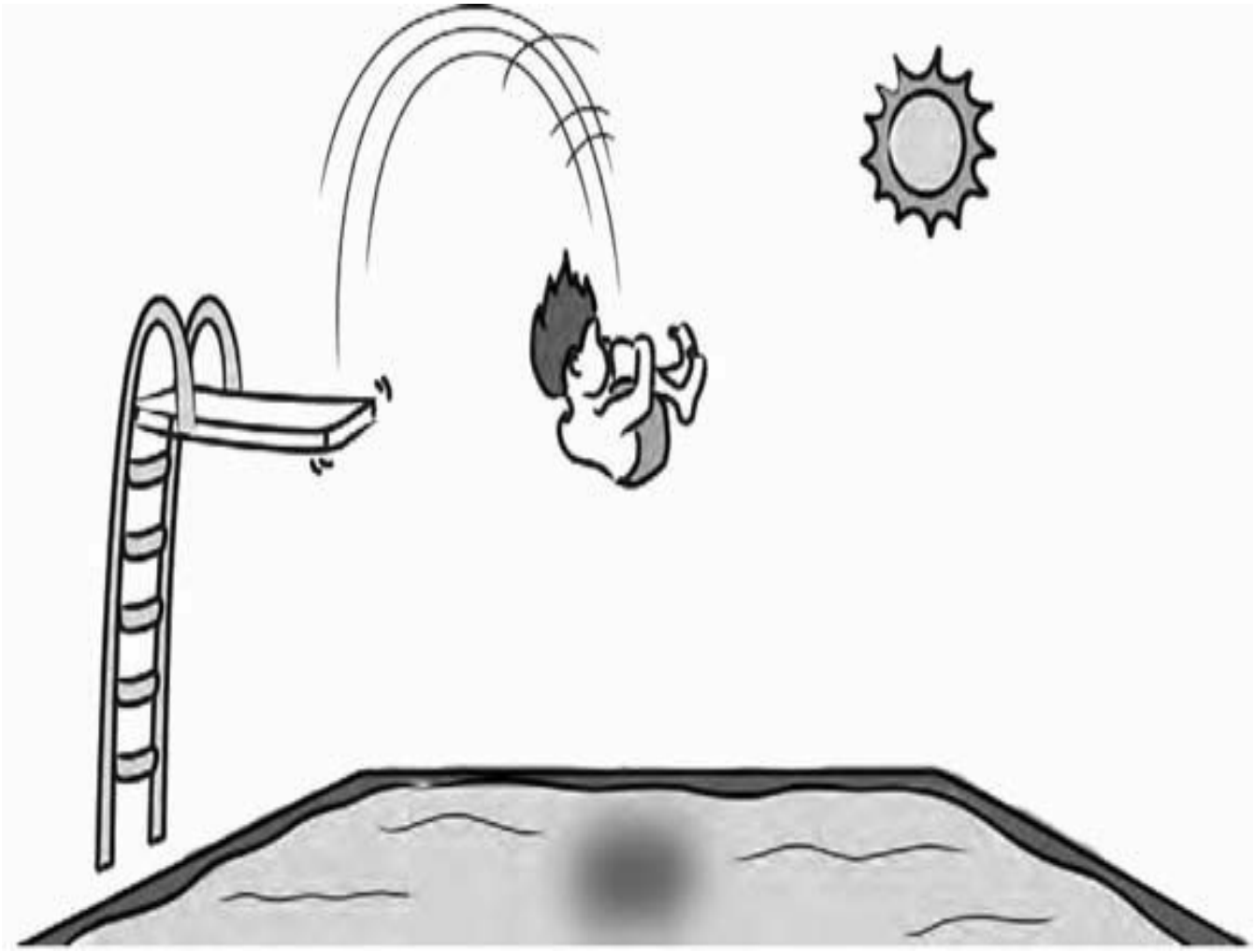
不必事事躬亲，但也不能当甩手掌柜。在

从个人贡献的角色转变成领导角色的过程中，要做到平衡是最困难的：一开始的时候，你会倾向于自己包揽一切，但是在担任领导职务一段时间后，又很容易养成凡事自己不动手的习惯。在刚刚担任领导职务的时候，你往往需要努力克制自己，把工作交给其他人去做，即使他们比你要花更多时间才能完成。这样你不但自己能保持清醒的头脑，还能让团队里的其他人有机会学习。如果你已经有了领导的经验，然后去领导一支新的团队，那么获得团队尊敬，并且跟上大家步伐最简单的办法就是卷起袖管亲自动手——最好是那些没人愿意做的脏活累活。你的简历上或许有一长串的成就，但是没有什么能比得上你参与进来，真正地去攻克一些难题更能让团队了解你的技术和决心（当然还有谦虚的态度）。

寻找接班人。除非这辈子都不打算再换工作，否则你就应该为自己寻找接班人。之前也提到过，这件事要从招聘的那一刻开始做起：如果你打算在团队里挑选接班人，那么在招聘的时候就应该雇佣有能力接替你的人，这就是我们常常挂在嘴边的那句话，你应该“雇佣比自己聪明的人”。而找到这样的工程师之后，还要给他们机会来承担更多的责任，有时候还可以暂代一下领导职务。这样一来，很容易就能在他们中间找到最有潜力，并且有意愿要带领团队的人——记

住，有些出色的工程师只想在自己的岗位上发光发热，没有当领导的兴趣，这种选择并没有错。我们总是会吃惊地发现一些公司不顾员工意愿，把最优秀的工程师放到管理职位上去。其实这么做往往只会让你的团队失去一名优秀的工程师，平添一名蹩脚的经理罢了。

知道什么时候要做恶人。总会有一些（不可避免，同时又经常发生的）棘手的情况，让你束手无策，只想逃避。或许是某个工程师的技术达不到平均水平；或许是某个工程师不管什么事情都要发表反对意见；又或许是某个工程师每周迟到早退，只干30个小时。“再等一下吧，会好的。”你对自己说。“说不定自己就好了。”你会自己给自己找理由。千万不要掉进这种陷阱——在遇到这种情况的时候，你就应该立刻站出来有所行动。这些问题是不会自动消失的，你拖得越久，对团队的伤害就越大。俗话说得好，夜长梦多。等待只会延缓问题的爆发，只会导致无法言喻的损失，所以还不如坐言起行。



即便心里不愿意，但是有时候做恶人是有必要的

16

保护团队不受混乱干扰。担任领导职务后，通常第一件让你注意到的事情就是团队之外的世界混乱不堪，充满了不确定性（甚至还有点疯狂），以前还是工程师的时候是看不到这些的。傅攀勃第一次当经理是20世纪90年代（之后他又回到工程师岗位上去过），他完全被公司里大量的不确定因素以及各种组织上的混乱给震惊了。于是他跑去问另一个经理，公司里到底发生

了什么事情，怎么会突然一下子变得这样天崩地裂的。那经理听了大笑傅攀勃的幼稚：这些混乱情况并不是突然出现的，只是傅攀勃之前的经理保护了他和他的团队不受这些东西的干扰罢了。

帮团队遮风挡雨。保护团队不受各种不确定性的干扰，帮他们挡下团队之外各种琐碎无意义的需求是非常重要的；同样，保持信息的透明度，让他们知道公司的“高层”有什么新闻的重要性也完全不亚于前者。在条件允许的情况下，你应该尽可能地和团队分享信息，但是也不要把那些不太可能会直接影响到他们的事情告诉他们，这种组织性的混乱只会让他们分心。

告诉团队他们干得很好。很多新上任的团队主管都不怎么会处理队员的缺点，结果忘记了要经常表扬他们。别人把事情办砸了的时候你要说，把事情干好了也一样要说出来，还要让他（和团队里所有人）知道他干得有多出色。

最后，如果遇到有冒险精神、常常想要尝试新东西的队员，最好的领导常常会用这样的招数：只要回转的余地足够大，就可以批准。比如说有个队员想要花一两天时间试用一套新工具或者类库，说不定可以提升产品的性能（而且项目也不紧），那不妨说：“没问题，试试看。”但要是情况相反，比如他要发布一个未来十年都需要提供支持的产品，那你就要好好权衡一下了。真

正优秀的主管一般对事物有没有回旋余地都有很准的直觉。

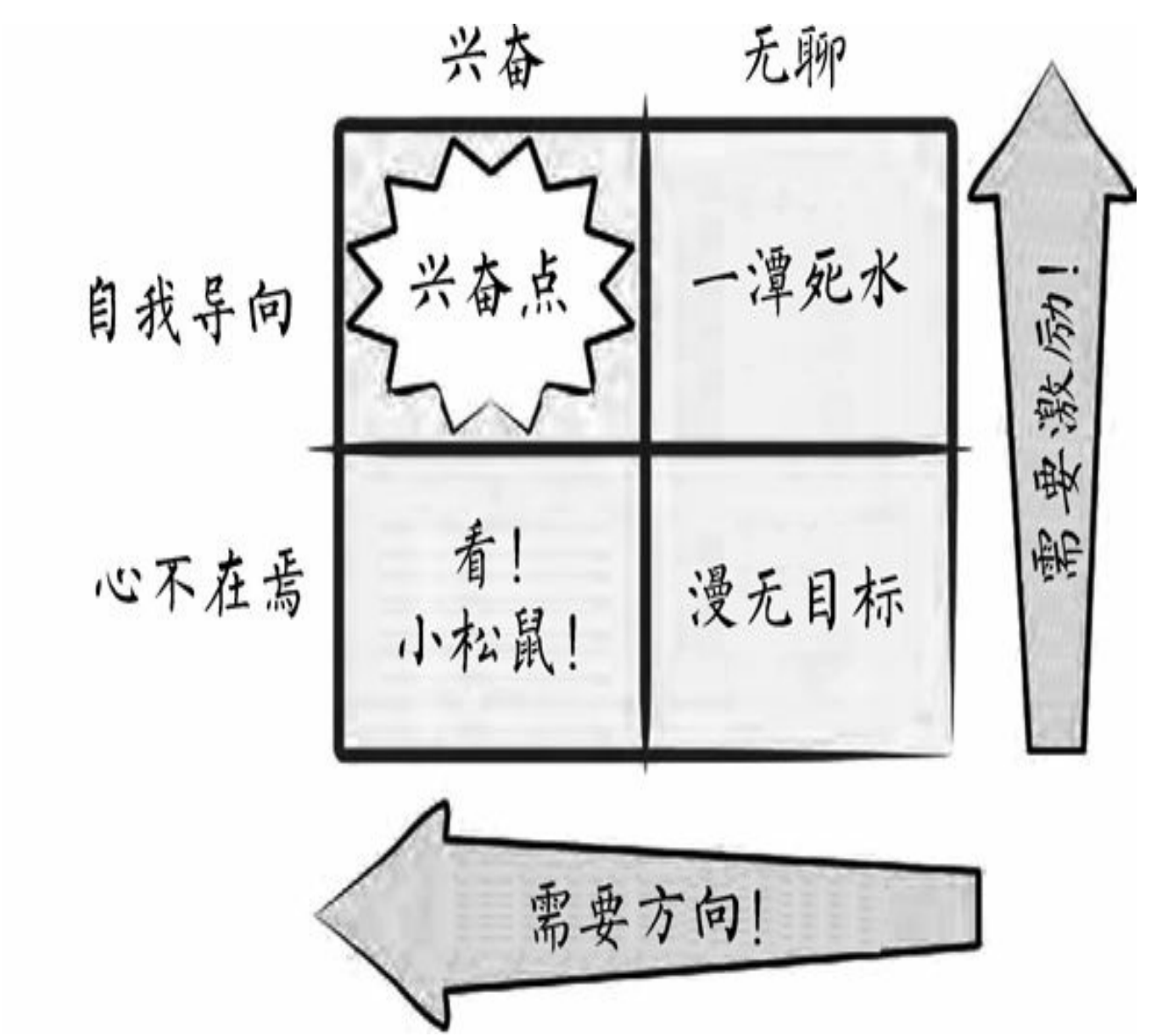
人是植物

傅攀勃的太太兄弟姐妹六个，她是老幺。她的母亲曾经要面对的麻烦事就是怎么把六个性格各异的孩子拉扯大，每个人都有不同的需要。傅攀勃拿着这个疑问去问丈母娘，她到底是怎么做到的，她答道，孩子就像是植物一样：有些是仙人掌，几乎不用浇水，多晒太阳就可以了，而有些则像非洲紫罗兰，应避免暴晒，保持土壤湿润，还有一些像番茄，只要一点肥料就能茁壮成长了。如果六个孩子都用相同分量的水、阳光和肥料，尽管待遇相同，但是很有可能谁都得不到自己真正需要的关怀。

每个工程师都需要不同的养分才能成长

因此工程师也像是植物一样：有些需要更多的光照，有些则需要更多的水分（还有一些则需要更多的牛粪[17]，哦不对，是肥料）。而你身为领导的任务就是要弄清楚每个人的需求，然后满足他们。

下面是一张表格：



你的员工属于哪一类

要让所有人都落在“兴奋点”那一格里，你就需要鼓励那些属于“一潭死水”的工程师，给属于“看！小松鼠！”那一格里的工程师提供明确的方向。当然，那些属于“漫无目标”的工程师则两者都需要。因此，水和阳光在这里是没有用的，激励和方向的组合才是你需要提供给工程师的东西，这样他们才能开心起来，进而提高生产力。

但是也不要给得太多——假如某个工程师不需要激励和方向，你再去激励他、指导他的话，只会造成反效果。

指导方向的意思还是比较直观的——你对究竟要做什么需要有一点基本的了解，还需要具备一点组织能力和协调能力来将它分解成可以掌控的具体任务。有了这些工具，你就能引导工程师往正确的方向上前进（好吧，其实没那么简单，我们在本章开始的地方已经讨论得很多了）。而激励则稍微复杂一点，下面我们来多讲讲这方面的东西。

内部激励和外部激励

激励分为两种，外部激励指的是来自外部的力量（比如金钱上的报酬），而内部激励则是来自内在。丹·平克在他的畅销书《驱动力[18]》里说道，最能让人开心、充满活力的并不是来自外部的激励（比如砸一堆钱给他），而是设法从内心激励他们。丹宣称，只要能给予以下三样东西就可以达到内部激励的目的：自主、精通、目标[19]。

工程师拥有自主权的意思就是他可以独立工作，不需要别人盯着才能干活[20]。对有自主意识的工程师，你只要告诉他们产品的大致走向就行

了，至于具体怎么操作，完全可以留给他们自己决定。这是很有激励性的，因为他们更了解产品（比你更清楚怎么打造产品），而且这样更能激发他们的主人翁精神。产品的成功和他们关系越大，他们就越希望看到它成功。

从本质上来讲，所谓的精通就是说你要让工程师有机会学习新技能，在现有的基础上继续磨炼提高。大量地提供这样的机会不但有助于激励工程师，还有助于工程师进步，进而令团队变得更强[21]。工程师的水平就和刀刃一样：你可能要花好多钱才能为团队找到最高水平的工程师，可要是你只“用”刀却不磨刀的话，用不了几年，这把刀就钝了，运气差点干脆就废了。只有给工程师提供大量的机会让他们去学习和掌握技艺，才能保持锐利、高效实用。

说到底，要是一个人不知道为什么工作的话，不管有多大的自主权，不管精通多少技艺都是没有意义的。因此你需要给他一个工作的目标。很多工程师虽然做的产品是非常了不起的，但是这些产品对公司、客户，甚至世界所产生的正面影响和他们都没有关系。其实就算产品的影响力没有那么大，你也可以让他们知道自己的努力是有价值的，以此来激励团队。只要你能让他们看到工作的目标，他们的动力和生产力就会成倍增加[22]。我们有个经理朋友一直都会关注公司

里有关产品（某个“没什么影响力”的产品）的E-mail反馈，只要他看到有客户说到这个产品是怎么帮到他或者他的业务的，他立即就会把它转发给团队。这不但激励了整个队伍，还能不断地激发他们的灵感，想方设法把产品做得更好。

结语

不管你有没有领导团队的意愿，我们都希望这一章能帮助你理解什么才是优秀的团队领袖，并且解开一些有关主管究竟要为团队做什么的谜团。就算你决心永远不当领导，了解一下本章说到的概念也是好的，因为它们有助于你理解为什么你的主管要做那些事，不管他是不是胜任自己的工作。想想自己团队的日常运作，你的主管采用了哪些模式和反模式，效果如何，你会对自己的团队为什么是现在这个样子有一个更具体直观的了解。

不过理解天天在一起工作的团队和领导只是与人合作的一个方面——和团队之外的人的交流更具挑战性，特别是如果这个人会拖累团队的话。我们把这种人叫作“害群之马”，下一章的主角就是他们。

注释：[1] 译注：其实中英文里都没有这个

词，是硬造出来的，由“manager”和“itis”组合而成，“itis”作为词根表示炎症。这里指的是当了经理就不用干活（或者没时间干具体工作）的情况。

注释：[2] 译注：即Pointy-haired Boss。著名漫画《呆伯特》里的人物。

注释：[3] 工业革命在欧洲始于18世纪，在美国则始于19世纪。

注释：[4] http://www.ted.com/talks/dan_pink_on_motivation.htr

注释：[5] 译注：这里指倾向中产阶级偏上阶层。

注释：[6] 假如你有小孩的话，一定记得自己第一次制止孩子做什么事情的时候说的某句话让自己大吃一惊（甚至惊呼）：“天哪，我简直变成了我妈（那样的人）。”

注释：[7] 正如福亨·来亨说的：“我在开玩笑呢，孩子。”译注：福亨（Foghorn Leghorn）是华纳兄弟的著名动画《乐一通》（Looney Tunes）里的著名角色，形象是一只白色公鸡。这句话是

他的著名台词。

注释：[8] Google在这里用经理这个词只是表示“有人要向他汇报”，仅此而已，绝无“他必须发号施令”的意思。

注释：[9] 这是另一个公司不应该强迫人们转入管理的理由：如果一名工程师非常擅长写程序，却一点都不想管人或是领导团队的话，逼她去做管理或是技术主管只会让你失去一名优秀工程师，得到一名糟糕的经理。这种做法不但糟糕，而且相当有害。

注释：[10] 参见第二章里有关失败的一节。

注释：[11] 这是真名。

注释：[12] 译注：一种玩具，可以用来占卜。

注释：[13] 参见《橡皮鸭调试法》，http://en.wikipedia.org/wiki/Rubber_duck_debugging

注释：[14] 参见阿尔伯特·萨夫亚的演讲，“快速构建原型宣言”。http://www.youtube.com/watch?v = t4AqxNekecY。

注释：[15] 需要公开批评某一个人的情况几乎是不存在的，绝大多数时候这样做都是很过分，很残忍的。团队里的其他人肯定早就知道这个人把事情办砸了，所以完全没必要重复讲。

注释：[16] 译注：这里的原文是make waves，所以才会配这样一张图。它指的是为了谋求事物的改变不得不去做一些会产生矛盾，或者让别人不爽的事情。在这里我把它翻译成“做恶人”。为了避免读者看到这张图觉得莫名其妙，特此解释一下。

注释：[17] 译注：这里作者开了个小玩笑。Bullshit也有胡说八道的意思。

注释：[18] 本章之前也提过，另请参见丹在TED上就这个主题所作的精彩演讲。

注释：[19] 这里的前提是，工程师的收入不成问题，不会因为钱的问题而焦虑。

注释：[20] 当然，这个的前提是你手下的工程师不需要微管理。

注释：[21] 当然，这也意味着他们的本领和身价变得更高，因此要是他们不喜欢自己的工作，

随时都有可能离开。参见本章前面讨论记录员工幸福指数的模式。

注释：[22] http://www.management.wharton.upenn.edu/Grant_JAnificance.pdf。

第四章 对付害群之马

在引言里的一开始我们就提到，软件开发中最难的是跟人打交道。

到目前为止，我们都是在审视自我。我们会先检验你自己的行为，以及它们是如何反映到谦虚、尊重和信任（HRT）三大原则上的。然后，我们探讨了怎么围绕这些概念来培养没有沟通障碍的团队文化。而在上一章里，我们讲解了在必须站出来领导这样的团队的时候，你应该怎么做。

在本书的下半部分，我们要换个方式，把目光转向外面。和团队之外的人应该怎么交流？总会有人希望加入或者和你的团队合作吧。在大公司里，办公室政治总是无法避免的。当然，你一定要想清楚怎么和团队之外最重要的人沟通：他们就是软件的用户！

本章我们要讨论的是一个非常重要的话题，防止那些捣乱的家伙破坏你的团队辛辛苦苦建立起来的合作氛围。更重要的是，我们要探讨一下怎么对付那些已经在团队里的害群之马。

什么是“害群”

我们已经讨论了培养稳定的、沟通无碍的团

队文化有多重要。我们一直在强调，好的文化氛围应该包括基于共识决策的开发模式、高质量的代码、代码审查，以及能让人放心尝试新事物或者快速失败的环境。

同样重要的是还要了解哪些东西不应该包括在文化里面。如果你打算打造一支高效敏捷的团队，那么知道自己不要什么也是非常重要的。虽然优秀的工程师能让团队更快更有效率，但是有些不好的习惯和做事风格会拖累团队，让公司变得不再那么让人舒服——最终这些都会侵蚀队伍的团结。

我们刚开始在研讨会上说到软件开发中的社交难题的时候，用的演讲标题是“怎么对付坏蛋”，大会主席建议我们把它改成“项目如何才能在害群之马的铁蹄下幸存”，希望这种八卦风格的标题能吸引到更多听众。事实上他是有道理的，这个演讲在很多研讨会上大受欢迎，听众多得连站的地方都没有。吸引听众的不单是“害群”这样非常负面的词汇，还有一个原因就是很多人都有这样不得不和那些叫人恼火的人打交道的经历。演讲到最后几乎肯定会变成一场诉苦大会，听众们会交换自己的故事，商量对策。

不过这样其实是很危险的。一般来说，一个人总是让自己沉浸在负面情绪里是不健康的行为——长远来讲，它会侵蚀你的一切，制造更多麻

烦[1]。“害群之马”是很大的帽子，一下子就把“我们”（也就是好人）和“他们”（捣乱的坏蛋）对立了起来。其实完全可以换一个思路来看这个问题。在带领团队的时候，不要把自己想成是一帮精英，众志成城地要把所有的烂人都轰走，而是要培养一种拒绝容忍负面行为的文化氛围，这才是正确的态度。要剔走的是行为本身，而不是人，单纯地区分好人和坏人是很幼稚的想法。规定好哪些是不可容忍的行为，然后予以惩戒，才是更有建设性的务实态度。

简单起见，我们暂时继续采用“害群之马”这种修辞手法来指代那种行为不端的人，但是在日常对话里千万不要轻易使用这个词哦！

保护团队

还记得酵母的比喻吗？团队文化和创始人的气质是紧密联系的。对团队文化影响最深的就是创始人，要是创始团队的文化不够强势，那么后来的文化就会压过它。如果创始团队很清楚哪些行为是可以接受的，哪些行为是不可以接受的，那么这些东西就能传承很多年。

我们两个在开源项目圈子里混了好多年，多年的经验也充分印证了这一点。

Subversion 是我们接触最多的项目，刚开始

的时候它只有很少的成员。大家都非常谦虚，相互之间有一种自然而然的信任感和尊重。十一年来，项目的参与者来来去去至少三四拨（大部分创始人早就离开了），但是传统还是保留了下来——大家都和和气气的，很有礼貌，相互尊敬。这不但是因为它坚持了高标准，还因为文化总是会进行自我选择。物以类聚，人以群分。

自我选择也很容易往坏的方向发展。如果团队一开始就聚集了一帮愤青，那么它就会有越来越多的同类。有些项目我们实在是不愿意提，比如Linux内核社区就是典型的例子——无止境的斗嘴，狂妄的发言，还有各种出口伤人。这样的团队或许能完成很多工作，但是总体上的运营效率却叫人怀疑。要是在人身攻击上没有浪费这么多精力，那么能多做多少事情？如果没有把那些礼貌的人拒之门外，那么他们的潜在贡献有多大？

我们再次提到这个话题是为了让你明白其中的代价，害群之马会直接危害到你的高效团队。如果容忍了不良行为的存在，不但你的生产力会受到影响，还会渐渐侵蚀团队的文化。而对抗它的最好办法就是通过一系列强有力的最佳实践和流程来提高团队的抵抗力。这些内容在第二章里都已经讲过了，这里再简单回顾一下。

保护团队，抵御不良行为

？写一份明明白白的任务宗旨。这样可以随时保持专注，知道哪些是目标，哪些不是。

？E-mail 讨论要有礼仪。保留归档，要求新人研读，防范那些“嘈杂的少数人”。

？所有历史都要有记录。这不单指代码历

史，还有设计决策、重要的bug修复，以及过去犯下的错误。

? 有效地进行协作。利用版本控制，代码改动要尽可能的小，方便进行审查，扩大“公车因子”，避免出现领地感[2]。

? 修复bug，测试，发布软件要有清晰的政策和流程。

? 降低新人加入时的壁垒。

? 依赖基于共识决策，在无法达成共识的时候也要准备好化解矛盾的方法。

最重要的是，这些最佳实践越根深蒂固，社区就越不能容忍各种有害行为。等捣乱的人真的出现的时候，你也就做好准备了。

发现威胁

要帮助团队抵御害群之马，首先要明白的就是到底什么才算是威胁，什么时候要引起注意。

团队的注意力和专注力是最容易受到威胁的。

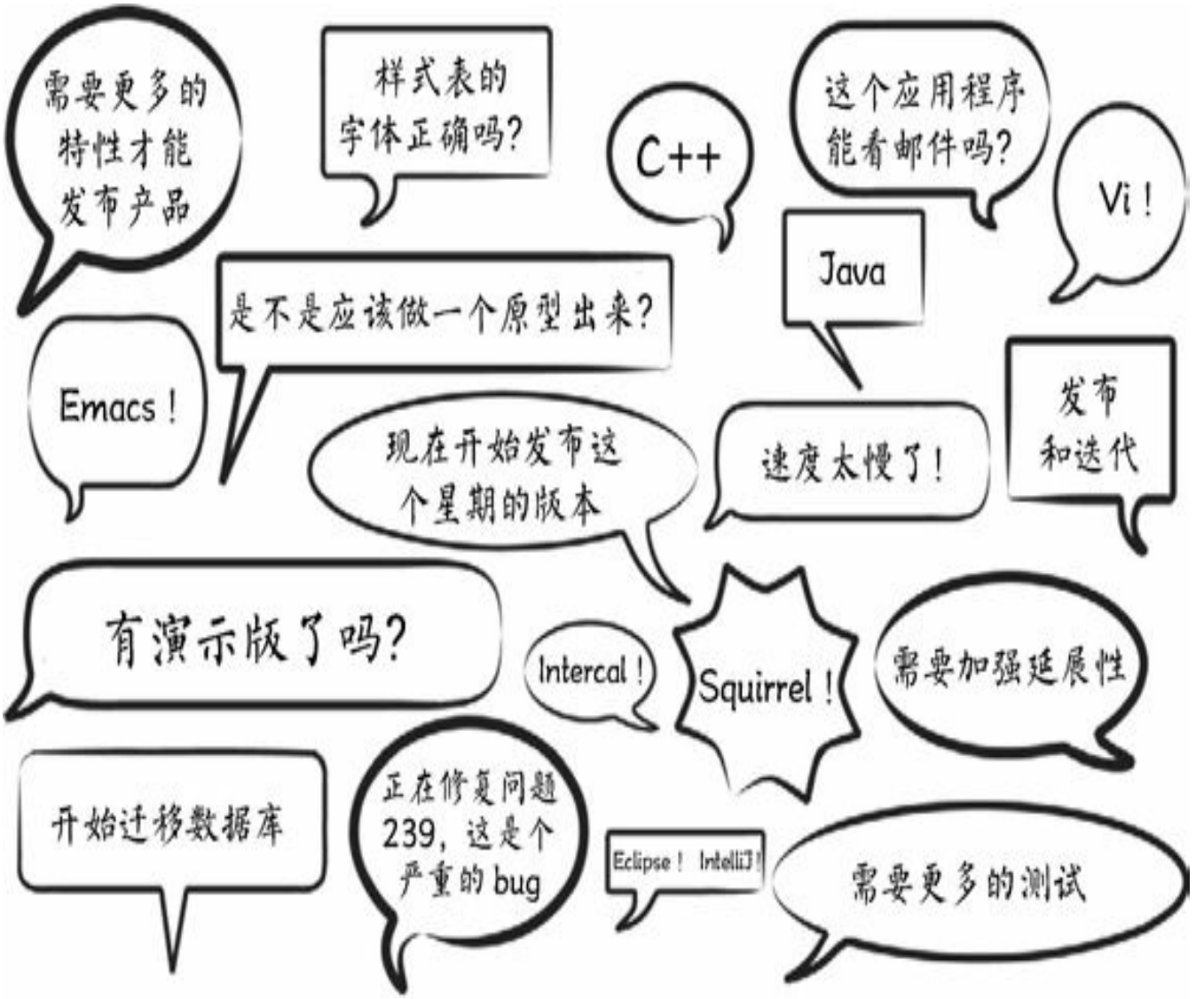
注意力和专注力是最宝贵的资源。团队规模越大，编写软件和解决有趣问题的能力就越强——不过这种能力毕竟是有极限的。要是你不去主动保护它们，很容易就会被害群之马引入歧途。团队最终会争论不休，变得心烦意乱、身心

疲惫。所有人都会把注意力和专注力放到那些编写优秀软件以外的事情上去。

这时你大概会问：害群之马到底是指什么样子的人？正所谓有则改之，无则加勉嘛。

根据我们的经验，很少会有人故意干坏事（也就是存心捣乱的那种）。我们管这种行为叫作“钓鱼”，通常无视这种人就可以了。而大多数人在行为出格的时候，要么是没有意识到自己过分了，要么就是根本不在乎别人的感受。无知和冷漠其实比蓄意更严重。绝大多数出格的行为都可以归结为缺乏基本的HRT。

这里是一些特别值得注意的信号和模式。只要这些东西一冒头，我们就会对这个人亮出黄牌——也就是说，我们会在心里作个记号，记住这个总是做出危害大家行为的家伙，以后和他打交道的时候就会格外小心。



你必须保护好团队的注意力和专注力

不尊重别人的时间

总会有一些人搞不清楚项目的状况，他们的危害通常是浪费团队的时间。他们宁可不断地拿那些很容易就能找到答案的问题去骚扰整个团队，也不愿意自己花点时间去读一读最基本的项目文档、任务宗旨、FAQ，或是最近的邮件讨论。

我们在Subversion项目里就曾经碰到过这样

一个人，他把开发主论坛当成了自己每天报流水账的地方。查理实际上没有贡献什么代码，但他每隔两三个小时就会发布自己最新的异想天开。这样就无可避免地产生了很多回复，去解释为什么他的想法是不正确的，不可能的，已经在开发中了，之前已经讨论过了，或者是已经有文档记录了等。更糟糕的是，查理甚至开始回答那些临时用户的问题，而且都答错了。这样，我们的核心成员只好不断地去更正他的回复。过了好久我们才反应过来，这位和蔼可亲的热心人其实是好心办坏事，大家被他牵扯了太多的精力。本章稍后我们会讨论遇到这种情况的时候该怎么办。

自负

这里“自负”可能不是最恰当的词，我们想要表达的是那种无法接受多数人决议，无法倾听和尊重其他观点，以及不愿作出妥协的人。这种人常常会重新挑起些早就已经结束（并且保留在邮件存档里）的讨论，仅仅是因为当时她不在场。这种人不肯去读存档，也压根不想去思考，她只会要求为了自己重启争论。她常常会就项目的前途作出极端的评价，声称除非按照她的思路走，否则失败就在眼前。

Subversion 就有过这么一段经历，当时有一名非常聪明的程序员出现在邮件列表里，声称产品的整体设计存在严重缺陷，而自己已经成竹在

胸，有一些大刀阔斧的办法来纠正错误，并且坚持项目应该整个推倒重来。他甚至还毛遂自荐希望能亲自领导重建工作，他宣称要是没有他的领导，项目随时都会有覆巢之险。

项目的创始人浪费了整个星期的时间，和这个家伙无休止地争论，誓要捍卫自己最初的设计目标。所有的注意力和专注力都涣散了。这个人显然无意作出任何妥协，也不想把自己的想法融入到现在的产品里，而项目（已经在公测阶段，拥有大量用户）也不可能重新来过。所以我们只能选择不再争论，回到自己的步调上来。讽刺的是，多年以后，事实表明他的预言在很多方面都是对的，但这并不妨碍 Subversion 的巨大成功——至少在企业级的软件开发上Subversion做得很好。这里关键的地方不在于谁对谁错，而是能否和而不同，以及争论是否有继续的必要。一定要提醒自己注意这些问题，有时候你必须作出决定，舍弃一些东西，继续向前。

过分索求

每当有陌生人跟你要求做什么的时候，一定要提高警惕。这样的人把所有的精力都用来抱怨软件功能不足，却不愿意自己动手作点贡献。

有时候等天上掉馅饼的心态会演变成过激行为。在运营Google的项目托管服务时，我们就遇到过这样的例子，当时有一个项目作者要求我们

封掉一个用户，因为他的所作所为实在是太讨厌了。这是一个开源的电视游戏模拟器项目，而这个用户最喜欢的游戏却无法在上面正常运行，于是他在问题跟踪系统里提交了一个口气相当粗鲁的bug报告。开发人员礼貌地解释了那个游戏跑不起来的原因，还告诉他相当一段时间里可能都没办法修复那个问题，结果那个人接受不了，每天都来骚扰开发人员。他不断地提交同样的bug报告，里面充斥着各种不满，还在其他bug报告里评论说拒绝修复他的问题的程序员是个“蠢货”。尽管项目人员和Google管理员屡次警告，他的用词却反而越来越不堪。不管我们怎么努力去消除他的这种破坏性行为，他就是冥顽不灵，万般无奈之下，我们只好祭出最后一招——彻底把他封掉了。

幼稚或是莫名其妙的交流

这样的人不会用真名。他们常常会用一些幼稚的昵称，比如“SuperCamel”、“jubjub89”，或是“SirHacksalot”之类。更糟糕的是，这样的人往往会在不同的地方用不同的昵称——E-mail一个，即时消息里又是另外一个，可能提交代码的时候还有一个。更有甚者，你会看到他们用火星文、黑客语、全部大写，甚至含有大量标点符号的沟通方式！[3]

偏执妄想

在上面的例子里我们看到，有时候不切实际要求会直接转变成对项目的恶意。我们无数次看到它彻底演变成偏执。当团队和访客的意见不一时，这种心怀恶意的人就会抛出某种阴谋论。要是太把他当真，去花精力和时间反驳的话就实在是太滑稽了。而且如果你已经建立起一条开放透明的沟通渠道的话（正如我们在第二章里所推崇的），这种指控只会显得更加可笑，因为所有的谈话内容都是有公开记录的。我们的建议是根本就不用去理会这种指控。当这种人真的做到这一步的时候，你说什么都是没用的，既然这样干嘛还费这劲呢？还不如把时间用来写代码。

完美主义

乍看之下，完美主义者根本就是无害的。尽管时不时地会有一些奇怪的强迫症类型的行为出现，但是总体上这样的人都是谦虚有礼貌的，而且愿意倾听别人的意见，看起来满是快乐的HRT和良好的本意。那么问题出在哪里呢？答案就是太追求完美会变得瞻前顾后、犹豫不决。

就拿我们以前的同事来举例吧。帕特里克是一名非常出色的工程师。他做的设计非常出色，代码和测试的质量也很高，人也非常容易相处。但是每当要设计新软件的时候，他就会无休止地调整、改进自己的设计。他从不满足，好像永远也不会开始写代码一样。尽管他对我们所面临的

问题有非常好的见解和洞察力，但是团队里的其他成员最后都被折腾到不行。这样下去就没法工作了，我们几个考虑了很久要怎么办。一方面，对团队来说帕特里克是巨大的财富；另一方面，他也妨碍了团队前进的步伐。每次我们打算开始编写代码的时候，他就会很有礼貌地否定我们的方案，指出其中只在理论上成立的潜在问题，而且都是一时半会不会产生什么影响的问题，不知不觉中他让我们整个陷于瘫痪状态。在下一节里，我们会讨论该如何应对这种情况。

对抗有害行为

我们不鼓励仅仅因为别人有点反社会或是不太礼貌就把他们踢走。我们之前已经提过，拉帮结派地把我们（所谓的好人）和他们（所谓的坏人）对立起来不是什么好主意。注意在之前的例子里，我们关注的不是批判人，而是批判“行为”。恶意的行为是不可容忍的，如果重复警告之下仍然没有改观，那么只有考虑正式拒绝了。

把精力都放在消除恶意行为上常常足以将一个聪明人（虽然可能不太擅长和人打交道）变成团队里的得力干将。几年前我们手下有一个非常出色的工程师，但是他有一个致命的缺点，就是有时候会惹到团队里的其他人。我们没有直接轰走他，而是把他拉到一边问他有没有意识到自己的言论会让别人对他敬而远之。他表现得很吃

惊，完全不能理解为什么自己说的话会造成这种效果，但是答应说会调整自己以适应团队。事实证明效果非常好，随着他的转变，问题也就自然消失了。并不是所有的故事都是悲剧收场的！

好吧，你已经找到了害群之马，可能现在就有人在干扰或是牵扯团队的精力。怎么才能有效地对付这种情况呢？下面是一些行之有效的策略。

转移完美主义者的注意力

一旦找到解决问题的办法，哪怕不是最佳方案，但是只要够用，就应该把完美主义者的注意力放到其他问题上去。

这个办法非常好用，Subversion 就是这样解决完美主义者的难题的。当时实在是没办法了，我们只好把帕特里克叫到一边，告诉他说：“我们决定以这个设计为起点开始工作，看看效果如何。希望你能帮助我们继续解决在这个过程中出现的各种问题。”出人意料的是，帕特里克对此完全没有异议，转身就继续研究其他难题去了。气氛相当融洽，帕特里克也一直在贡献自己的力量。

俗话说，过犹不及。在打造高效团队的时候，一定要时时警惕不要过于追求完美，否则只会适得其反。

这种转移注意力的办法用在那些花太多时间

抱怨和批评的家伙身上也同样有效。有时候对这种人会忍不住回一句“随时欢迎提交补丁”——开源社区里让人闭嘴的委婉说法，但其实不如引导他去正式参与软件的测试，说不定就能发现一些以前修复过的问题又重新出现。这样，虽然他还在发牢骚，但是会比较有建设性。

别去搭理那些挑衅的家伙

这是Usenet上流传的一句格言[4]。这对任何蓄意捣乱的家伙来说是最好的手段——这种人会处心积虑地向你和你的团队挑衅。你回复得越多，他就越来劲，结果你就会浪费越来越多的精力和时间。其实无视就是最好的办法。无论心里有多少警言妙语可以一句话就噎死他，都要克制这种冲动。当他发现没人理他的时候，自然就会意兴阑珊地离开。忍住、不回应往往是需要很大的意志力的。坚持！

别太感情用事

人往往会很容易表现出防备的心态，不管别人是不是蓄意挑衅。当你被人骂说设计做得太烂，或是谋划什么阴谋，又或者浪费了太多时间去回答再简单不过的问题的时候，的确是很让人郁闷的。但是别忘了，你的任务是写出漂亮的软件，没有义务讨好所有人，也不需要一再去证明自己存在的价值。你越是情绪化，就越容易浪费宝贵的时间去写一些激昂的回帖，而那些都是不值得你关注的人。应战之前应该谨慎一点，时刻

保持冷静，知道哪些人是值得回应的，哪些人是可以无视的。

抓住重点

继续刚才的话题，保持理智的更深一层含义就是要学会抓住重点。一个人在抱怨、发泄情绪的时候，一定要认真听他说。虽然会夹杂一些愤怒和粗俗的话，但是要相信对方本质上是没有恶意的。他说的到底有没有道理呢？我们是不是可以从他的经验里学到什么？他的想法是不是值得回应？很多时候答案都是肯定的——那就是虽然他语言上有点刻薄，但背后其实是有亮点的，所以应该尽量把争执再次引向技术讨论。[5]

我们团队在这一点上做得非常叫人自豪。只要保持绝对冷静，一切以事实说话，发帖人就自然会随着交谈的深入而显得疯狂可笑。到最后没人会相信他说的话，他脸皮再厚也混不下去了。

对付挑衅要不卑不亢

除了上面提到的办法（就是保持冷静，摆事实，讲道理），有时候连礼貌也可以拿来当武器吓走敌人！这里是一段Subversion在IRC频道里的真实记录。

哈里：Subversion太烂了，完全用不了。

萨斯曼：有问题可以问我啊。

哈里：我想要cvs一下某个人的文件。不对，我要抗议一下。

那家伙用的是一种叫Subversion的东西，他用svn，不是cvs。

萨斯曼：你可以装一个svn客户端，然后检出他的代码。

哈里：所以我就去下了一份Subversion啊……结果你以为它可以像 cvs那样 configure make make install吗？当然不行。比起subversion的人，我怪他比较多。

萨斯曼：（你）没办法 ./configure; make; make install不代表软件有严重bug。每天都有无数人从svn源码包编译编译软件。

哈里：我可没说有bug。

萨斯曼：你觉得我们会发布一个像这样的有严重问题的源码包吗？

哈里：我只是要骂一下这个垃圾（软件）。居然还要我装expat和libxml。（叹气）

萨斯曼：这些软件在大多数系统上都是预装的。

萨斯曼：那个人用的是apache服务器吗？说不定你只要下载一份编译好的版本就可以了。

哈里：我不知道，他就说了svn而已……

萨斯曼：你用的是什么发行版？

哈里：FreeBSD

萨斯曼：其实你直接到ports里去make一份就可以了。

哈里：碰到你我什么脾气都没了……我上来是想要吵架发泄的……你也太和蔼可亲、热心助人了吧。

萨斯曼：:-)

哈里：什么时候开始IRC频道变成一个这么和谐的地方了？算了不说了。

哈里退出了频道。

知道什么时候应该放弃

有时候，当无论怎么努力都是徒劳的时候，你就应该果断放弃，继续向前。就算你已经花费了大量的精力去纠正有害行为，也要学着去承认失败。

让我们回到查理的故事，这位友善的哲学家在 Subversion的邮件列表里发了太多的帖子。我们对所有的讨论做了一个统计，发现他在两个月之内挤进了发帖量的前三名。前两名都是项目的核心，而且他们70%的帖子都是在回复查理！尽管查理本身没有恶意，但显然我们的精力和注意力都被吸走了。最后我们只好私底下给他发了一封E-mail，（礼貌地）请求他不要再这样频繁地发帖。这段谈话进行得很艰难，主要是因为他没有意识到自己造成了多大的干扰。可是几个星期之后情况仍然没有改观，我们的一个同事只好给他打电话，好好地和他谈了一次（这样的谈话难度更大），要求他彻底停止发帖。最终他还是接

受了，虽然有一点难过和不解，但仍然尊重了团队的意愿。大家都觉得有点内疚，因为他一直都没能理解自己到底做了什么，但是大家也觉得这么做是正确的。要解决好这种事情是很微妙的，我们只有谨守HRT的原则才能处理得当。

关注长远

通向成熟软件产品的道路上有无数的干扰。要是说在对付害群之马所带来的干扰时最常见的情景是什么，那肯定就是太容易被当时的情况所吸引了。如果你发现了所谓的有害行为，一定要问自己两个关键的问题：

？虽然短期之内会损失一些注意力和专注力，长远来讲你真的相信项目会因此受益吗？

？你相信这些冲突最终会以有益的方式解决吗？

把注意力放在重要的地方，不要被眼前的东西迷惑

只要任何一个回答是“不”，你就应该果断介入，中止那种行为。虽然人们常常会自我安慰说暂时忍耐一下换取短期利益是值得的，但其实事实并非如此：例如，有人或许在技术上贡献良多，可仍然会做出一些有害团队的行为，这时往往就会为了那点技术上的优势而选择对他睁一只眼闭一只眼。但是这时候千万要三思！HRT的氛

围是无可替代的，而再强的技术也是可以替代的。我们以前有一位同事这样说道：

我有不少朋友都认识他。其中有一个是这样说的，“他常常游走在天才和疯子之间。可问题是，现在天才已经不稀奇了，没人会因此接受这样举止古怪的人了”。

——格雷格·哈德森

这里格雷格说的不是通常意义上的“天才”，他的意思是这个世界上有的是合格的程序员。如果你发现有人在长远来讲会威胁到团队的文化，那么不妨再等另一个出现。

我们在Subversion项目里就曾经遇到过这样的情况。团队有严格规定，不准把名字写到源文件里（这一条我们在第二章里就讨论过了），我们觉得做只会产生领地感。如果代码里有别人的名字，修改的时候会觉得有点担忧，而且这么做还会人为地降低公车因子。所以在版本控制的记录里认可大家的贡献才是比较恰当的做法，我们在项目的根目录里放了一个这样的文件，里面有所有贡献者的名字。

有一天，来了一个非常聪明的程序员，主动写了一个所有人都想要的新特性，工作量还不小。在他提交代码审查的时候，我们只是要求他删掉文件开头的名字——我们会在和别人一样的地方感谢他的贡献。可是他拒绝了，谈判陷入了

僵局。最后，大家决定拒绝接受他的代码，他带着自己的玩具离开了。虽然大家都很失望，但是我们不愿意仅仅是为了能快点得到新特性，就打破自己的规矩（进而放弃自己的传统）。几个月之后，就有其他人重新实现了这个特性。

在这里要再次明确强调：为了短期利益而打破规矩不值得——特别是对于那些不懂得尊重HRT重要性的家伙来说，再大的天才也没用。

小结

这一章我们讨论了很多场景，说到最后似乎会产生一种偏执的感觉。但是别忘了，这个世界里混蛋其实并不多。正如罗伯特·J·翰龙所说的：

不要把用愚蠢可以解释的行为归结为恶意的。

这里我们更倾向用“无知”而不是“愚蠢”，但是基本思想还是一样的。就像我们在一开始提到的，把人简单地分成好和坏是很幼稚的。没有什么坏人处心积虑地想要毁掉你的文化——大多数人只是被误导了而已；又或者只是想要得到认可，同时又不太擅长与人交际罢了。不管怎么样，你的任务不是要培养傲慢的态度，把那些没有那么聪明的普通人赶出项目；你的任务是拒绝容忍毁灭性的行为，明确自己对 HRT的期望。有智慧的人才能体会其中的差别，而有能力的人才能真正予以执行。

注释：[1] 尤达在这种时候大概会建议远离黑暗面。译注：参见星球大战。

注释：[2] 译注：第二章里解释过了，就是指某个东西只有极少数人，甚至只有一个人懂。

注释：[3] 译注：这里的火星文是指lol-speak，或者叫lolcat，指的是那种故意用错语法、时态、量词等的说话方式。而黑客语则是指1337speak，也叫leet语言，就是那种将字母变成数字和符号的书写方式。

注释：[4] 这句话本身则是来自初代《星际旅行》里的一集，“和平日”（Day of the Dove），里面有一种以负面情绪为能量的生物。柯克（Kirk）和他的对手克林贡人刚（Kang）都命令手下不要再以负面情绪给这种能量生物喂食，最终迫使它离开进取号。看到了吧，《星际旅行》是最棒的。

注释：[5] 更多关于这个话题的内容可以看一下诺曼·柯西在《项目回顾》（Dorset House）一书中的“回顾里的主要目标”。

第五章 操纵组织的艺术

到目前为止，我们已经讲过了如何处理好你和团队之间有关人的部分。我们提到了领导工程师团队所必需的一些基本的人际上的能力，还提到了应对害群之马的威胁时所要面对的风险。然而除此之外，你同样还需要理解怎么在好公司和坏公司里生存。大多数软件工程师都服务于效率低下的官僚机构（公司），因此要采取一些非常的手段才能达到目的。有些人称之为办公室政治，你也可以把它叫作社会工程。

而我们给它起的名字是组织操纵。

优点、缺点和策略

大公司是一个复杂无比的有机体系，无论你怎么神通广大，也需要GPS、手电筒，以及一大堆指示牌才能搞清楚东南西北。

我们首先来看看在一家理想的公司里团队是如何运作的，然后再来讨论运营不良的公司是怎么给团队制造障碍的。我们会讨论在这两种公司里做事情的策略，最后如果这些策略都失败了的话，我们还准备了一个后备计划。

在公司里找方向有时候很难

理想的情况：团队在公司里应该是怎么运作的

运营良好的公司通常指的是两个层面：首先是你的顶头上司，也就是大多数时间里和你打交道的人；还有就是除他以外的人，这包括了工程师、中层经理、公司高层、销售人员、律师等。

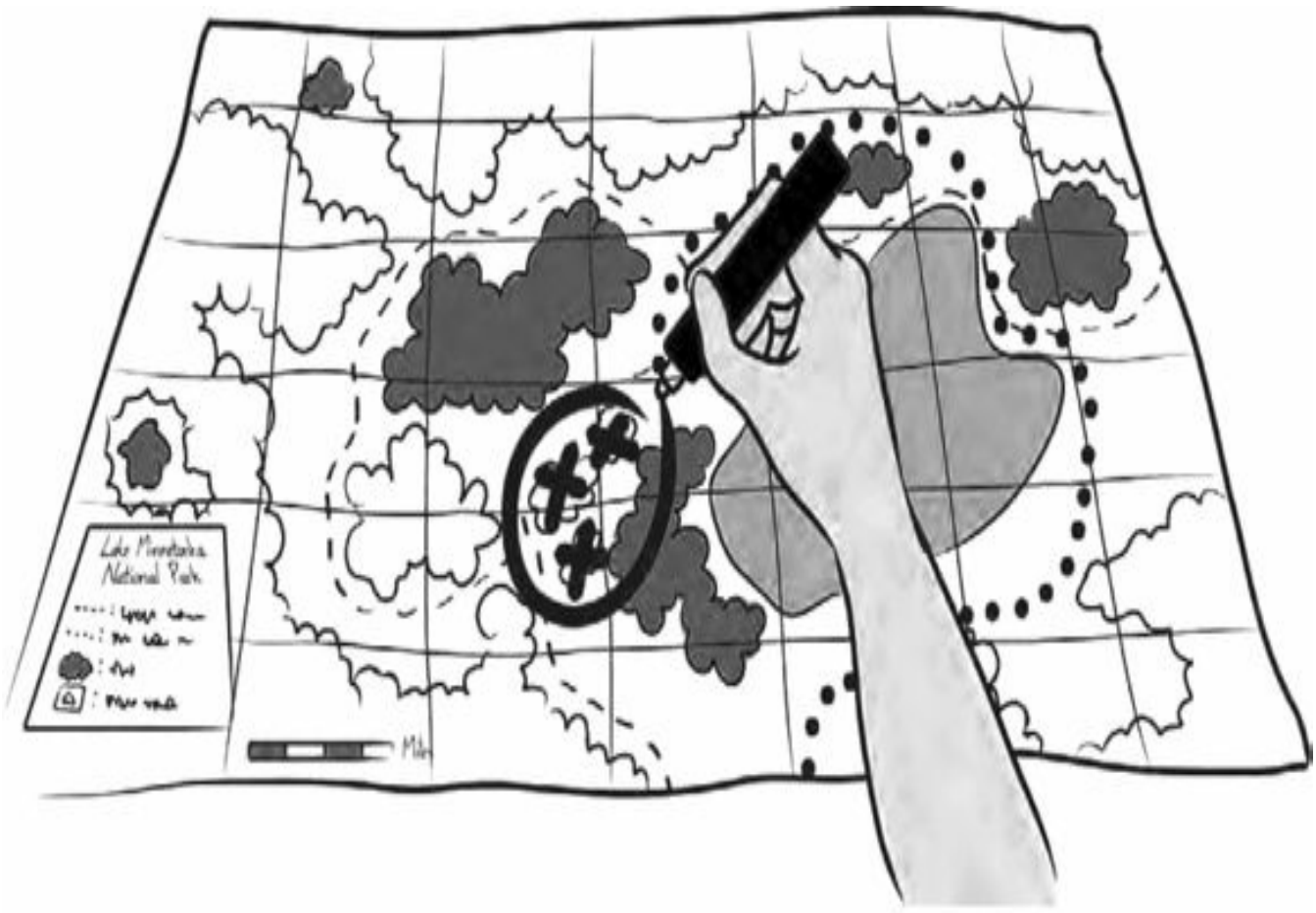
不单要专注手头的任务……

在完美经理手下干活

如果你的经理是典型的仆人式领导，遵循HRT的理念，发自内心地想要助你成功（参加第三章）的话，你只要做到下面几点就可以减轻他的负担，进而也让自己变得更有效率和价值。

在完成自己工作的前提下，要求更多的责任。我们最爱用的比喻就是把你比作一个刚入行的守林人，受命于一个资深守林人去砍掉一棵生病受损的树。如果你只是抱着完成任务的心态，

那么一定会径直走进森林，砍倒那棵病树，然后就洋洋得意地回来交差了。但如果你能把眼光放大一点，就应该把路上看到的其他病树也都记录下来，再准备好一个能迅速把它们全部解决的方案，让资深守林人看一下是不是可行。这样，下次守林人再碰到这种任务的时候，她就会首先想到把这份责任交给你。这不单是因为她了解你的能力，更因为比较轻松——她的工作量减小了。



……更要起出预期

这种主动承担责任的行为能减轻你经理的负

担，她不再需要担心这件事情，而且也展示出你超越当前工作难度的能力。不过这也意味着你要离开自己的舒适区域，去尝试新的事物。如果团队的氛围是鼓励冒险和快速失败的话，这些其实都没什么大不了的。

勇于冒险，不怕失败。在第三章和第四章里我们讨论了很多有关冒险和快速失败的重要性。只要经理够开明，失败其实是快速成长的最佳手段，认识自己的极限，然后逐步提升它们。我们有一个经常出差的朋友，叫作史蒂夫·黑曼，他常常这样说，“如果你从来没有误过飞机，那说明你去机场太早了。”这句话放到软件开发里也同样适用：如果你从来没有失败过，那说明你太保守了。和追求更多责任一样，勇于冒险也是展现自己有能力做大事的一种方法。

尽管在工作中不冒险可以降低失败发生的概率，但也表示大获成功的机会随之变少。好的经理会鼓励团队去尝试，看看他们可以做到什么程度（这个过程本身就是很好的学习经历），万一失败，她也会准备好软着陆的方法。如果真的失败了，也要站出来承担责任，不要推卸，总结经验吸取教训，避免再次犯错，然后重复整个过程。

表现得像成年人一样。另一个建议似乎看起来有点再明显不过了：你有责任引导别人的行为

和对待你的态度。糟糕的经理常常会把团队当成小孩子，压制任何想法、责任，以及打破常规的念头。要是你曾经碰上过这么一位，你肯定也会以为所有的经理都是这样的。

对不确定的事情提出疑问。如果经理的决定让你觉得难以认同，千万不要害怕和她争论，或是对她决定的出发点提出质疑。尽管这不是说你就可以肆意制造障碍，但是盲从对于领导来说也是毫无帮助的，特别是当你掌握了一些她不具备的知识和经验的时候。

你的经理不是千里眼，公司里几乎不存在沟通过多的情况，所以别犹豫，一定要在经理问你进展之前，就向她汇报自己在干什么。在遇到困难、完成任务、需要帮助，或是希望看到什么事情发生的时候，不妨给她写张条子。这样不但你的经理很清楚你在干嘛，同时也是对付微管理的绝佳办法：假如你的经理不断催你进度，那么主动以同样的频率给她发E-mail汇报就是让她收手的好办法。

这些技巧在理想的企业里都是行之有效的手段，但要是自己所在的公司和“理想”两个字完全不沾边的话该怎么办？

[现实的情况：当环境成为成功路上的绊脚石]

幸福的家庭都是相似的，而不幸的家庭则各有各的不幸。——列夫·托尔斯泰，《安娜·卡列尼娜》。[1]

公司环境里阻碍你成功的事情有很多，但是基本上可以分成两大类：糟糕的人，还有糟糕的公司。

在糟糕的经理手下当值

要描述什么样子才叫糟糕的经理都不知道要从何说起——有无数的电影和电视剧专门去讽刺那些不称职的经理人。我们大多数人在各自的职业生涯里也都至少碰到过一位不合格的经理，这样的经理人能把最优秀的团队带入深渊，所以我们在这里打算讨论一些坏经理的典型特征，特别是会影响到工程师的那些特征。

害怕失败是糟糕的经理身上最常见的特征。缺乏安全感会把他们变得非常保守，这和一般工程师的工作风格南辕北辙。假如你的经理不希望你冒险，那么你就几乎没什么机会把自己的想法加入到产品中去，最后只能（死板地）实现别人设计好的产品[2]。

通常没有安全感的经理会坚持要参与你和团队之外任何人之间的交流中去，进而防止你说出一些“没有经过上级批准”的东西来。这种经理认为任何你和团队之外的工程师所作的直接交流都属于背叛，更不要说和其他经理接触了。如果你

需要从团队或者公司之外寻求任何帮助，她会希望你先去找她，这样才能凸显自己的重要性以及和你的上下级关系，这样也就让她觉得自己更有力量。

我们大都在学校里再三听过“知识就是力量”的说法。坏经理也非常了解这一点，但是她的切入点却有点不太一样：她希望独享这份力量，不愿意和你分享，不管这对你的工作是不是有帮助。她会把得到的信息藏匿起来不让你知道，这样才能保证自己不会被排除在任何和这个信息有关的事物之外，这样做不但让你无法完成工作，还能让她保住自己的地位和权利，开发进程被拖得再慢也没关系。

有的人喜欢隐藏消息

坏经理通过藏匿消息，硬是成为信息和沟通的渠道的办法，来达到窃取胜利果实[3]以及让你背黑锅（有时候，甚至是她们自己的错误）的目

的。很多时候，这种坏经理只是把你当作她往上爬的垫脚石，根本不关心你的职业发展，更不用说团队的幸福指数了。

我们的朋友苏珊好多年前就碰到过一个可怕的经理，这家伙常常会丢给苏珊一个新项目，却不告诉她任何相关信息，也不说怎么才能把项目做完——就算苏珊对这个新任务一无所知的情况下也是如此。苏珊的经理并不一定是希望苏珊把事情办砸，但要是他能告诉苏珊他知道的所有细节的话，不但苏珊的日子会好过一点，苏珊也会觉得和经理比较好打交道。如果可以直接和相关的团队联系，对方就会知道是苏珊（而不是她的经理）在负责这个项目。苏珊有好多次都想要让新项目的进展速度快一点，想要快点把事情做完，可最后都崩溃地从各种渠道发现，所有的努力成果都被经理抢了功劳。

这和我们在第三章里提到的仆人式经理完全相反，坏经理只想要知道你为她做了什么。而那些团队里表现不佳的人呢？根本没人关心，只要他们不闯大祸就可以了——对坏经理来说，和那些人打交道实在是太麻烦了。

办公室政治高手

尽管我们坚信信任是非常重要的，或者说至少也要先假定对方是无辜的，但是如果这份信任被那些办公室政治高手利用的话，却是对职业生

涯十分不利的。

刚接触一个人的时候很难界定他是不是所谓的办公室政治高手，因为这种人往往左右逢源，很擅长和人打交道——所以一开始的时候他一定让人觉得很好相处。一般这种人很会利用同僚或下属，踩着他们往上爬。他会抓住各种机会推卸责任，甚至比抢别人功劳还迅速。通常这种人都是两面三刀，见人说人话，见鬼说鬼话，就是希望给你留个好印象。但要是发现没办法利用你或是操纵你，他就会当你不存在，或是把你当作是威胁，想方设法地在暗地里中伤你。所以只要相处一段时间，这种人就藏不住狐狸尾巴了：他大多数时间都在想办法让自己表现得有影响力，而不是真正地让自己变得有影响力。

我们的建议是对这种办公室政治高手最好敬而远之：没什么事的话就躲远点，但不要口没遮拦地在公司里跟人说他的坏话，谁知道对方是不是清楚他的为人呢，或许他也被蒙蔽了。另外，如果你是那种喜欢保持低调、专心干活的人，可偏偏身边就有那么一位高手的话，你最好改变一下这种工作方式。你或许会觉得不想“玩这种勾心斗角的把戏”，懒得去争权夺位，可最后却发现那些小丑变成了你的上司，结果你身边不但有玩弄办公室政治的家伙，又多了一个坏经理。详细的讨论请参见后面“操纵你的组织”一节。

糟糕的组织

随着公司的成长壮大，往往会滋生出各种官僚政治和流程，以便管理利润、降低风险、提高预期，以及支撑公司庞大的自重，而这种官僚机构到了一定时候会逐渐变成公司成功道路上的阻碍。和糟糕的经理一样，市面上有关糟糕组织的文章也是数不胜数，所以这里我们只谈一些经常会影响到工程师的组织性问题。

首先最简单的一个现实就是，大多数公司都不是以工程师为核心的。这就是说，工程师只是实现商业目标的工具，而商业目标本身通常不是技术性的。这就意味着经营公司的那些人通常理解不了支撑他们系统的技术，他们只明白在这之上的商业需求，因此他们往往会提出一些技术上不太现实的要求。就算一个懂技术的高管在这样的公司里试图保护自己的组织，最后也常常会发现自己被那些愿意为了商业需求而牺牲员工利益的人所替代。通常这些都会以不切实际的时间期限，或是缺乏人手却要求项目按期完成等形式出现。你会发现自己很难得到足够的硬件，产品没办法有效地跑起来，或是团队花了好几个星期重写的东西其实只要花几百块钱买点硬件就能解决问题了。不幸的是，在那些不尊重工程师，只把他们当奴隶的公司里，这种情况是非常典型的，在这种公司里，工程师对公司如何运作是完全没

有发言权的。

而其中最糟糕的，就是那种有非常僵化的指挥控制结构，搞得和封建领地一样的公司。我们有个朋友特伦斯在很多年前曾经服务过一家公司，那里对团队之间传递bug有非常严格的规定。有一次另一个团队产生了一个bug，会导致特伦斯的产品在几个小时之内耗尽内存，特伦斯花了整个晚上重现bug，收集数据，写出案例分析，然后把这份E-mail发给他的经理。他的经理再把E-mail转发给再上一层的主管，这名主管再把E-mail转发给对方团队的主管，对方团队的主管再把E-mail转发给对方团队的经理，这名经理再搞清楚谁是负责出问题的那块软件。而其实他只要把E-mail直接发给负责那块代码的人，让他们去查日志和代码就行了。十多天后，特伦斯发现要和两个经理、两个主管，还有三个工程师讨论这个bug能不能在下一次发布之前修复。听起来很好笑吧？但这种事情事实上每天都在发生[4]。

有的公司里净是一些对名头和阶级很狂热的家伙，这只会导致无尽的权力倾轧，经理常常会阻碍工程师跳到别的团队以避免自己的团队人才流失——哪怕这种人员流动对公司和工程师本身（来说）都是好事。

你的公司有没有把你当成过淘气包一样对待？比如公司有一道神经质一样的防火墙，让你

连一些无关紧要的外部网站都上不了？你是不是每天都要填写详细的表格，记录每时每刻都做了什么？有些公司甚至会通过一些毫无意义（而且是非常不准确）的方式来衡量你的生产力，比如每星期写了多少行代码[5]。

你有没有被当成淘气的小孩

还有一些公司培养的员工衡量成功的标准不是发布了多少优秀的产品，而是受邀参加了多少

会议。

最后，你的公司可能欠缺诸如专注力、大视野，以及方向感一类重要的东西。这往往是山头林立，或者所谓的“由委员会设计[6]”所造成的，结果底下干活的小兵往往得到相互矛盾的指令，最后只能被卷进不断收紧的恶性循环，而不是明确的方向。

很多坏公司身上都具备这样的特征，有的甚至更糟糕。但就算是这样的公司也还是由人组成的，这里有一些小技巧或许可以帮助你和这些人走出困境。

操纵你的组织

这是一个演示程序，和Matrix里编好的现实差不多。它也有一些基本的规则，比如说重力。你要知道的是这些规则和计算机系统的规则没有什么本质区别。有些可以变通，有些还可以打破。明白了吗？现在来打我吧。——墨菲斯

和演示程序一样，公司也是由规则组成的，有些可以变通，有些还可以打破。如果你老是把注意力放在公司“应该”怎么做事上的话，最后除了挫败和失望什么都得不到。相反，若能接受现实，然后从中找到利用它的方法，就能完成工作，还能在公司里找到一片立足之地。不管你身

处好公司还是坏公司，下面的这些策略总能帮助我们有效地完成工作。

“请求谅解比请求许可要容易得多”[7]

首先，你要知道公司的底线在哪里——虽然请求许可让你有机会把责任推给别人，但同时也给了别人拒绝你的机会，所以你一定要弄清楚在没有明确得到上级许可的情况下你能得到多少资源。我们建议你的是，无论如何，你都要确信自己要做的事情是符合公司利益的。

就算你准备好要请求谅解了，也一定要作出明智的选择——每次你申诉什么事情，或是要反驳公司里的其他人时，你都在消耗自己的政治资本。如果把这些资本用在赢得一堆无关紧要的事情上，那么当碰到真正重要的事情的时候你会发现自己一无所有。所以只有在事关重大，或是确信自己有把握赢的时候才去争取。把所有的资本投在一场没有胜算的战斗上毫无意义，只会给自己带来压力，限制自己的职业前景罢了。

如果你决定选择“请求谅解”，那么最好在公司里找到一些同事和朋友来支持你——特别是那些风险比较高的想法。这些人应该对你在公司里的影响力有所了解，也应该知道哪些想法是无论如何都不会成功的。

当初营销部的人建议傅攀勃应该把他的数据解放团队在 Google的高层那里好好宣传一下的时

候，傅攀勃咨询了周围的同事们，大家都觉得数据解放的确是非常酷的玩意，应该让高层看一看。于是傅攀勃决定试一下，结果大获成功。几年后，傅攀勃又盘算着一些，怎么说呢，不怎么样的想法，同样还是这帮同事，这次他们却表示计划实在太冒险了，于是傅攀勃决定放弃。如果你打算先斩后奏的话，相信自己的直觉虽然是好事，但听一下你信任的人怎么说也是很有价值的。

路是人走出来的

在公司里，另一个寻求改变的办法是拉拢群众。当你找到足够的人支持你的创意或是采用某个特定产品的时候，官僚机构常常会发现已经民意难违了。这时管理层只能被迫将注意力转向你，要么选择接受；要么采取措施去反对它（你猜对了，这会消耗他们的政治资本！）。这是多年来很多工程师都一直在用的办法，例如，在每天的工作中悄悄塞进一些开源的工具，以方便自己的工作。

假如你想要说服一个人，如果你能找到几个认同你的人在和他聊天的时候提到你的创意（或是提案，或是请求），那你成功的概率就能大大增加。就算你的目标对这一切都心知肚明，人类的基本心理还是会有暗示作用的。当他从多个渠道（而不只是你一个人）听到同样的想法时，自

然就会对其另眼相看了。

创意是非常吸引人的东西，如果你不在意是不是一定要让别人知道是你原创的话，创意往往能变得非常有活力！有时候你会发现人们谈论创意只是想要让人觉得这是他想出来的，这时你就要作出选择，到底是要原创还是要让创意传播。尽管听到自己的想法从别人（可能这个人还有点让你看不起）的口中说出来会叫人很痛苦，但很多时候这是唯一让创意传播的办法。无论在大公司还是在小公司，这种事情我们已经见怪不怪了：公司高层口中的那些高深玄妙的概念和创意其实都是来自公司里的其他人。但请想一下，这些原本无人喝彩的创意在这种情况下能得到多少听众吧！

路是自己走出来的

公司也和人一样，江山易改，本性难移。坏习惯总是很难改的。本小时候有一个老师曾经这样说道：“坏习惯是停不下来的，你只有用一个好习惯去替换掉它。”任何尝试过戒烟的人肯定都非常熟悉这种情况。公司也是一样——如果你想要消灭一个坏习惯，最好是用一个好习惯去代替它。比如你不喜欢现在的周会，那把它换成另一种不同形式的会议或是其他什么（更有效的）活动。不喜欢现在毫无意义的汇报流程？别抱

怨，干嘛不自己动手写一份更漂亮的、好到没人可以忽略的流程？

学习向上管理

不管你是经理还是负责具体工作的人，都有必要花点时间在向上管理上。这就是说，你要尽可能确保你的经理以及团队之外的人不但知道你在干嘛，还要知道你干得很棒。有些工程师不太喜欢这种“自卖自夸”的东西，事实也的确如此，但这样做的好处却是大大的。

之后在第六章里我们会提到，在做承诺的时候要谨慎，而干工作的时候要尽最大努力。这并不是说你可以故意夸大工作量，拖延期限，而是说你应该尽可能地避免答应自己做不到的事情，哪怕不喜欢，也要拒绝对方。因为如果你一直无法按期完成，或是因此而放弃新功能，公司同事对你就会变得越来越不信任，当他们需要人手完成工作的时候，会越来越忽略你的存在。

身为工程师，应该把精力放在发布产品上，而不是其他的事情上面。发布产品比任何事情都能在公司内部提升你的信誉、声望，以及政治资本。发布产品是曝光率极高的大事，能充分展示你的成果。尽管在清理代码和重构上面花大力气是非常有诱惑力的想法，但是经验教训告诉我们，不要花太多时间在这种防御性的工作上，很少有人看重这些，到时候你会发现自己的处境很

尴尬，因为花了那么多时间，你却拿不出什么（政治上）看起来很重要的成果[8]。这样你不但得不到别人的认可，还很容易导致自己的项目被取消。

“进取性”和“防御性”的工作

本在头一次晋升为经理的时候，他的团队正被堆积如山的技术债务压得透不过气来。他决定团队首要任务就是清偿这些债务，其他的事情以后再说，他的上级当时没想太多就答应了。结果事情进展得不是很顺利。虽然本的经理事先答应了他的计划，但是几个月下来还是有点失去耐心了——为什么整个团队看起来“什么都没干”？其实本的团队相当有效率，他试图告诉大家他们偿还了多少技术债，但事实上这种工作是没办法给人留下什么深刻印象的，从情感上来讲甚至还有点烦人。

有了这次糟糕的经验后，本开始把所有的工作分成“进取性”和“防御性”两大类。进取性的工作通常是指用户看得见的新功能——在外人眼里这些都是很炫、很令人兴奋的东西，或是能展现产品优势的地方（比如，界面改进、速度提升，或是互操作性的增强等）。而防御性的工作主要是着重产品长期的健康状况（比如，代码重构、特性重写、修改数据库模式、数据迁移，或是改进紧急监控等）。这些防御工作能让产品更稳定

可靠，可维护性更强。然而尽管这些工作至关重要，却得不到任何政治上的好处。所以你要是把时间都花在这上面，在外人看来，你的项目就好像停滞了一样。套用一句成语就是“先入为主”啊！[9]

如此我们不妨作这样的总结：不管技术债务有多少，团队也永远不应该花超过三分之一甚至一半的时间和精力去做防御性的工作，否则就等于政治自杀。

运气和互惠的经济学

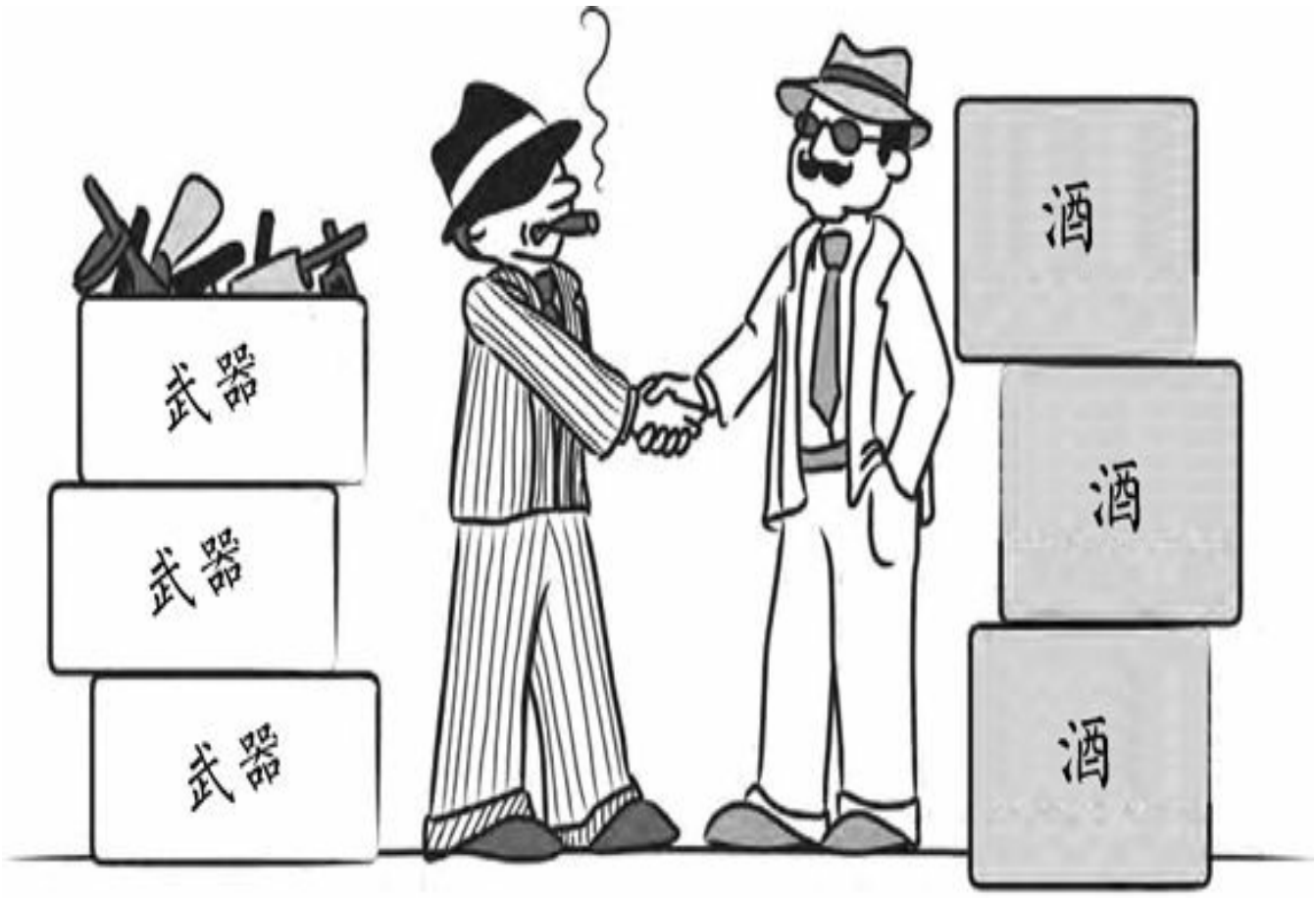
不管你在什么样的公司工作，给自己带来一点运气其实不是很困难的事情。《幸运要素》[10]的作者理查德·怀斯曼曾经做过这样一个实验，来测试人们发现机会的能力[11]。

我分别给了幸运的人和不幸运的人一张报纸，要求他们看看里面有多少张照片。一般来说，不幸运的人要花两分钟来数照片，而幸运的人只要几秒钟就够了。为什么？因为报纸的第二页上有这么一句话：“别数了。这张报纸有43张照片。”这句话占了一半的版面，字体差不多2英寸多高。直接呈现在人们面前，可是不幸运的人就是看不到，而幸运的人很容易就能看到它。

他继续写道：“幸运的人非常善于创造和注意到机会的存在。”我们觉得这也同样适用于在公司里创造机会的情况：如果你只是逐字逐句地

完成任务，除了自己的工作对其他事情都漠不关心的话，是很难看到什么机会的。如果有机会有条件帮助别人完成工作，即便那不是你份内的事情，而且他们也不一定会回报你（你也不应该期望对方投桃报李），但是很多人还是会很乐意在将来有机会的时候回报你的。

每家公司都有这种存在于桌子底下的互惠经济。你常常会碰到这种情况，虽然这件对公司有利的事情对你来说轻而易举，但它并不是你份内的工作，如果你能找到机会去做这些事情（很多时候，会有人突然跑过来请你帮忙），就能在这个互惠经济里得到好处。不妨把这些好处想象成一系列的小赌博：有些人永远也不会回报你，而有些人会回报你同等的好处，另外还有一些人会涌泉相报。虽然事先很难知道对方是否会回报你，但是有一件事情是肯定的，那就是人们会记住你曾经在他们遇到困难的时候出手相助，而不是甩下一句“这不关我的事”。如果你将来遇到麻烦要他们帮忙的话，他们就非常有可能（甚至迫切地）想要帮你的忙。就算真的没人“回报”你，在帮助别人的过程中你也能学到一点新东西，而且助人为乐除了牺牲一点时间和精力外，你能有什么损失呢？



“将来……我会找你帮忙的。”

互惠经济最有意思的地方就在于，哪怕离职了，你的账户也不会被清空——你还是可以在需要的时候找他们帮忙。这也是为什么在离职的时候千万不要自断后路的原因，不管当时有多冲动[12]。

晋升到一个安全的位置上

大多数工程师都觉得，只有工作成绩优秀才是符合逻辑的晋升条件，然而这种事情只有在最牛的公司里才会出现。其他大多数公司通常都需要你（在出色地完成工作的基础之上，）再耍一点手段才能如愿。

假如你对自己的工作、薪水和团队都感到非常满意，可能会不愿意卷入那些勾心斗角的事情，倾向安于自己现在的头衔和等级，但这会从好几个方面伤害到你——例如当公司结构发生变化，你被调整到新团队里的时候，有可能会碰到一个坏经理，或者会被分配到一个办公室政治专家的手下等。

在公司里的位置越高（不管是作为负责具体工作的人还是担任领导职务），你就越能掌控自己在公司里的命运。在对自己的职位感到满意的前提下，稍微投资一点精力在获得晋升上面是保护自己的好办法，特别是当公司或者团队发生问题的时候。你应该时刻关注自己完成了些什么，并以此来检验自己的工作。定时更新自己的简历，让你的经理和有权提拔你的人看到。仔细研读公司的晋升流程，和你的经理聊一聊，看看哪些事情是有助于晋升的，然后有条不紊地完成它们。即便晋升是个很主观和不确定性很大的事情，你还是可以去做很多事情来提高概率的。

和有能量的人交朋友

每家公司里都有一个“影子”组织，它没有任何文献记录，却有权有势，能产生极大的影响力，而只有少数几类人能在其中占有一席之地。

联络人是指那种认识公司里方方面面人物的人，就算他们不直接认识某个团队里的某个人，

他们也能设法帮你找对人。有时候要办成一件事，只要托对人就行了，而联络人就是那种可以帮你找对人的人。

老臣子不一定位高权重，但是他们知道很多别人不知道的事情，而且凭借资历也能发挥出相当大的影响力。当你想要了解公司这样处事的原因的时候，或者需要一个大家都尊敬的人来支持你的时候，去问这种人准错不了。

或许很多人都对此不以为然，但事实上高层的助理在公司里也是有极大的权力和影响力的，因为他们从某种意义上等于是管理层的代言人。更重要的是，他们常常要做大量的工作才能保持事情进展顺畅，因此对他们态度不好的话，吃亏的肯定是你自己（还有你的工作），所以千万不要错过讨好他们的机会哦。

最后要说的那个人是大家常常会忽略的，那就是你自己。职位高一点，对公司的掌控能力就越大。因此就算你对自己现在的薪水和工作很满意，就算你在一家完美的公司里工作，也要继续努力为下一次晋升做准备。记住：万一发生什么坏事，你的位置越高，全身而退的希望就越大。

如何通过**E-mail**向忙碌的管理层求助

只要在大公司里待过一段时间，肯定就会碰到需要给公司高层（或是某个你不认识的大忙人）发E-mail，要求对方帮你忙的情况。可能是

你要为自己的产品和团队争取什么东西，又或者是你打算要纠正一些错误之类的。无论如何，总之你从来都没和这个人打过交道，大姑娘上轿头一回。这种时候，几乎每个人都会犯同样的低级错误：啰啰嗦嗦，不着重点。

十几年前，傅攀勃（当时还在Apple上班）给他母亲买了一台非常糟糕的iMac，然后在同事的建议下给史蒂夫·乔布斯写了一封“简短的”E-mail[13]。这封E-mail基本上可以当作一篇如何向管理层求助的范文。

日期：2001年2月1日，星期四

收件人：sjobs@apple.com

主题：硬件体验实在太糟糕了——我该怎么办？

我非常希望你能给出建议，帮我解决这个问题，谢谢。这对Apple和我自己来说都是非常难堪的。

今年母亲节的时候，我给我母亲买了一台iMac——她是新奥尔良一家蒙台梭利[14]学校的副校长，之前一直在用学校里的一台旧麦金塔。新的iMac让她非常兴奋，甚至还为学校准备了资金，想要为实验室添置iMac。

然而我买给她的这台iMac却让人非常失望。

首先七月的时候，它进入休眠状态后就再也无法唤醒。她把电脑拿到Apple的授权经销商那

里去，他们诊断后认为是一块逻辑电路板有问题，于是换了个新的上去。

然后她把电脑带回家，插上电源后启动，可是却得到了一个启动失败的声音。无奈，只能再把电脑拿回经销商那里。他们诊断后认为是一块模拟电路板有问题，于是再次更换。

到了九月的时候，我终于说服她再试试看那个睡眠功能（而不是关机再启动），可是这台 iMac 就是醒不过来。拔掉电源再插回去后终于可以启动了。我们只好彻底关闭睡眠功能。

十二月，显示器开始出现色闪，黄的、绿的、蓝的一直不停地跳动。她昨天把电脑拿到经销商那里去了，现在还留在那边修。

所以我今天决定给你写信。我的母亲以为我在故意恶作剧，还跟所有人说她的那台iMac非常垃圾，我在Apple的同事也都不知道该怎么办。

有什么办法能让我还她一台工作正常的 iMac 吗（除了再买一台新的以外）？

谦逊的，

傅攀勃

20 小时不到，傅攀勃就接到了一通电话，是某个为史蒂夫工作的人打来的，两星期后他的母亲收到了一台新的（完全没问题的）iMac。

其中的秘密在于：当有机会改正错误的时候，那些在权位上的人都是非常乐意那么做的。

但不幸的是，这种人的E-mail 邮箱里往往看起来像是被无尽的拒绝服务攻击侵袭过一样，要是他们看到一份陌生的邮件，洋洋洒洒3000多字，连个分段都没有的话，他们大概只会读15个字，然后就删掉去读下一封了。

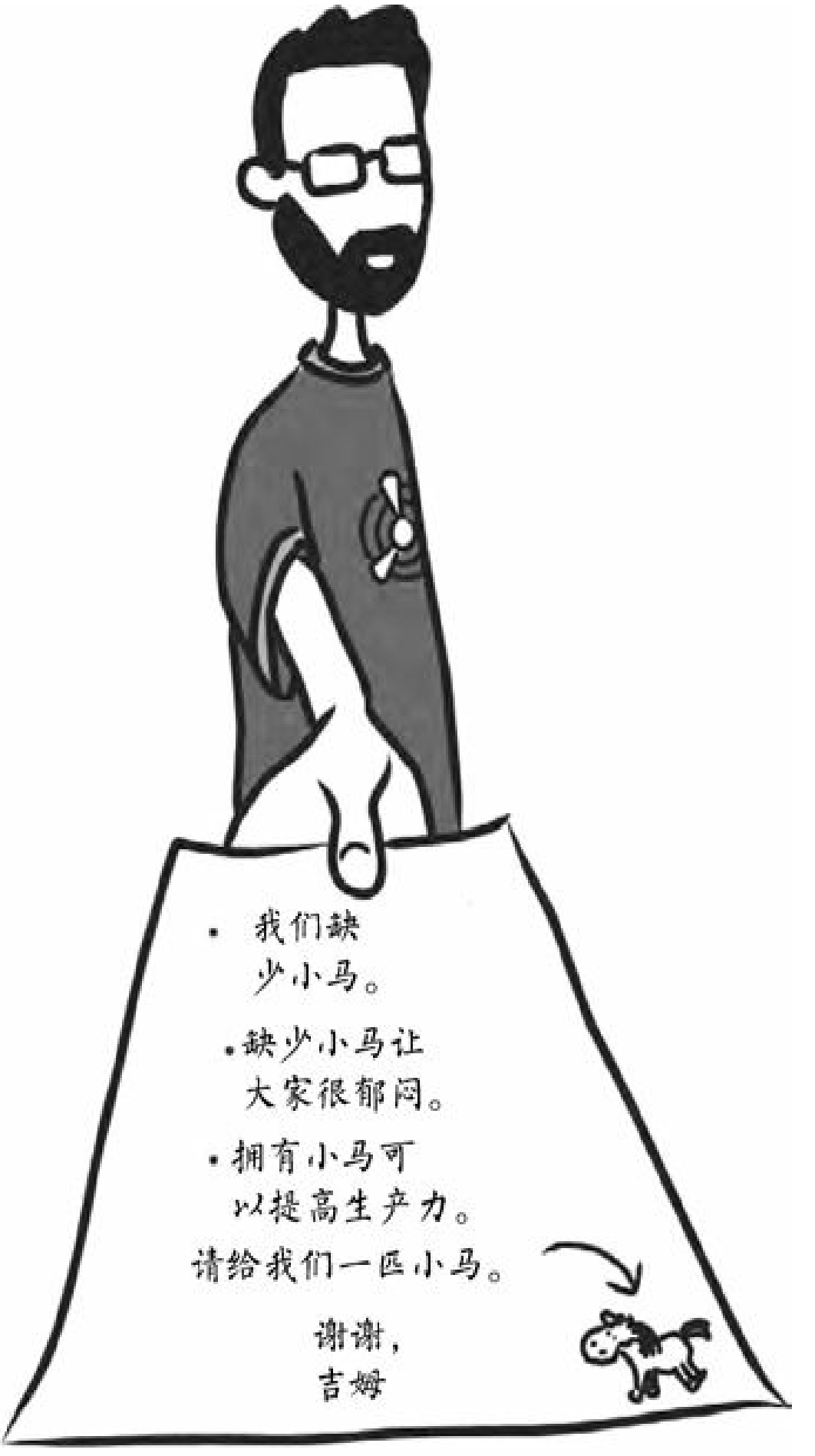
但是如果他们能在10秒之内读完E-mail，然后挥挥魔棒（比如给手下发个邮件让他们去帮忙处理一下）解决问题的话，他们就会很乐意去做。事实上，经过多年的试错后，我们发现越短的邮件就越有机会得到回复。

我们管这种技巧叫“三个论点和一个行动”，无论你向谁求助（不只是高管），无论你求助什么，这都能大大增加你成功的概率——至少也能得到一个答复[15]。

写得好的三个论点和一个行动的邮件（最多）包含三个点，让你解释问题的细节，然后一个（只能有一个）行动请求，绝不能有其他的内容。这份E-mail应该可以被轻易转发，要是你啰哩啰嗦，或者在邮件里写了四件完全不相干的事情，那么就会给对方造成很大的心理负担，其结果就是你的邮件会被丢弃掉。罗列要点一定要用简短的句子（每句话最长不能超过一行），行动请求也应该越短越好[16]。你的E-mail应该谨守HRT的原则：有礼有节，避免出现语法错误和拼写错误。如果你实在忍不住或者真地需要在E-mail里

描述背景或者包含更多信息，也应该把它们放在E-mail的最后（哪怕放到你的签名后面也无妨），然后清楚地注明这些是“详细信息”或者“背景信息”。

虽然有点马后炮，但是我们觉得傅攀勃的那封教科书E-mail还是太啰嗦了——要是让我们现在再写一遍的话，应该是这样的。


• 我们缺
少小马。
• 缺少小马让
大家很郁闷。
• 拥有小马可
以提高生产力。
请给我们一匹小马。
谢谢，
吉姆

申请小马的正确步骤

日期：2001年2月1日，星期四

收件人：sjobs@apple.com

主题：糟糕的用户体验——你能帮忙看一下吗？

我给我妈买了一台iMac，她是学校的管理员。她原本对这台 iMac 寄予厚望，甚至为学校准备了资金，打算为实验室添置更多的iMac。

结果7 月份的时候，Apple 给它换了一块有问题的逻辑电路板，一个月以后又换了一块模拟电路板。

9月份的时候，休眠功能无法正常工作，12月的时候，显示器开始出现问题。现在电脑还在经销商那边。

我妈逢人就说她的iMac是垃圾，我在Apple认识的人都已经束手无策了。

还有没有办法能让我还她一台正常的iMac？

谦逊的，

傅攀勃

E-mail重写后去掉了很多评论性的东西，但是再忙的高管也能在10秒钟之内读完它了。

我们工作到现在，已经通过这些技巧做成功了很多事，但是有时候有些事情是人力不可为的。

B计划：走为上

多年来我们一直在讨论怎么在糟糕的企业里和糟糕的人相处，怎么在各种情况下工作，每次演讲结束后，总会有人跑到我们面前，恼火地跟我们说他们已经尽了最大的努力了，但依然看不到情况有任何改善，什么事情都做不成功，他们该怎么办呢？这里有一个很遗憾，但是很简单的答案：你可能真的无能为力了，赶紧撤退，别被波及了。

如果你改变不了整个体系，继续投入精力去改变它是没有意义的。你应该把精力放到怎么离开它上面去：更新自己的简历，去问问周围的好朋友，看看有没有机会到其他公司就职；自学一点新知识。今时今日，身为工程师的最大好处就是优秀的工程师永远是紧缺的，因此你完全有能力掌握自己的未来。

认识到这一点能让你释放出极大的潜力。当你四处寻找机会发现有很多工作机会的时候，你会发现自己做事情的效率突然变高了（而且也不再感到那么大的压力），因为你根本不怕老板炒你鱿鱼！我们在Google的资深工程师陈一鸣的博客上找到了这篇文章，从此对我们的工作方式产生了深远的影响。

只做正确的事，随时准备被炒

Google 的新员工（我们称之为“Nooglers”）

常常会问我，为什么我的工作这么有效率。我总是半开玩笑地答道，很简单，我只做对 Google和这个世界来说正确的事，做完之后我就等着被炒鱿鱼。要是没有被炒，那说明我做的事情对大家都有好处；要是真地被炒了，那说明这不是我想要服务的雇主。无论如何，我都不吃亏。这就是我的职业理念。

直到最近我才发现这种反叛的性格都是遗传自我的父亲。其实这是很奇怪的事情，因为在我成长的过程中，我对父亲有一种既定的印象，而这种印象正是我所反抗的，所以把父亲想象成叛逆的人在我的认知上是非常不协调的，但他就是一个离经叛道的人。

我爸以前是童工（没错，就是你在国家地理杂志上时不时能看到的那些发展中国家里，几百万无名无姓的小孩子中的一个），但是后来却做到了新加坡最资深的军官之一。我最近才了解到，他之所以这么成功，是因为他敢于当面对长官说出令人不快的事实，就算是面对国防部长和总理也不含糊。在他的军队生涯临近结束之前，他的长官之一问他为什么那么有效率。我的父亲答道：“很简单，我每天回家的路上都会开车经过 HDB 公寓（新加坡的政府廉租房），我总会多看那里一眼。为什么？因为要是你炒我鱿鱼，我就要搬到那里去住了。”

将来有一天，等你到了我这个年纪的时候，不妨花点时间和你自己的父亲聊聊他的职业生涯，你就会发现自己和父亲有多像了，保证又惊又喜。

不要放弃

我们说了这么多选择辞职或是等着被炒的话题，不是要让你觉得如果你不喜欢现在的工作就应该马上准备简历想办法跳槽。相反，你的首要任务应该是做出必要的改变，让自己变得高兴起来，努力完成工作目标，而这一章讨论了好多可以帮助你做到这一点的方法。如果你不努力去学习、了解引导公司的方法，那就等于是拿自己的命运去赌博。

注释：[1] 参见“安娜·卡列尼娜原则”，http://en.wikipedia.org/wiki/Anna_Karenina_principl

注释：[2] 我们要强调的是，这种开发软件的方式本身并没有错，我们只是觉得对优秀工程师来说这样有点太无趣了。

注释：[3] 这样就加倍让人郁闷了，因为你在她的干扰之下依然做成功的事情，居然还要被他

抢功劳。

注释：[4] 与此相对的，傅攀勃在Google上班的头一个星期的时候，在Gmail里发现了一个打错字的地方。他直接找到源码，把错误改掉，然后把补丁发给了Gmail团队，对方也对他表示了诚挚的谢意。大公司也不一定都是官僚主义盛行的！

注释：[5] 删代码不才是更应该得到嘉奖吗？

注释：[6] 译注：即 design by committee，指的是生活中常见的一种情况，即由于各方意见不同，最后只得作出妥协，由委员会制定标准。结果往往粗劣不堪。

注释：[7] 这句话来自格蕾丝·穆雷·霍普将军，COBOL语言之母，一个非常诙谐幽默的计算机科学家。

注释：[8] 这不是说这些预防性的工作无足轻重，不过是这些工作在团队之外的人眼里是如此罢了。

注释：[9] 译注：原文为Perception is nine-tenths of the law。它借用的那句谚语是，Possession is

nine-tenths of the law。意思是如果你确实拥有一件东西，那么如果别人说这件东西是他的，就算打官司，你不但不会吃亏，而且会有很大的胜算。这里作者用perception这个词无非是想表达人们产生固有印象后是很难转变观点的，中文里最贴切的应该就是“先入为主”这个成语了。

注释：[10] 由Miramax出版发行（ISBN：978-1401359416）。

注释：[11] htp:/www.telegraph.co.uk/technology/2204496/Be-lucky-its-an-easy-skil -to-learn.html。

注释：[12] 技术行业的圈子比你想象得要小得多，人们谈论的东西也比你想得要多，所以今天被你棒打落水狗的那个人很可能也就是十年后毁掉你工作机会的人。除非你打算以后搬到沙漠里去编竹篮，否则绝对不要把事情做绝了。朋友有来有去，而敌人却是有增无减的。

注释：[13] 原本傅攀勃写了一大篇语无伦次的抱怨给史蒂夫，要是把这份东西发出去，除了被炒鱿鱼以外肯定什么都得不到。他的同事建议说应该尽量简单，只说重点，然后给出切实可行的建议才有希望。

注释：[14] 译注：Montessori是指一种教育方法。

注释：[15] 警告：如果你是一个异想天开的人的话，我们要明确一下，这种技巧不可能帮你得到和美国总统会面的机会，也不可能让雪佛兰向你订购你最新发明的激光雨刷，更不可能让你有机会和Whole Foods的销售主管共进午餐。这种技巧只适用于切合实际的情况。

注释：[16] 如果你期望对方回复，那么就应该尽量简化，让他能直接在问题下面作答。不要一口气问五六个问题——尽量一个段落里只放一个问题，最好是一封E-mail只问一个问题。

第六章 用户也是人

到这里，我们已经讨论了一长串成功软件开发的关键要素。

首先是要有一组聪明的程序员，然后遵循谦虚、信任和尊重的原则，为团队培养一种强大的文化氛围。以为团队服务为宗旨来实施领导，鼓励他们合作，引导他们作出正确的决定。根据需要给予足够的水分、阳光、指引，还有来自内心的激励。保护他们远离负面影响——比如威胁到团队文化和生产力的各种破坏性行为（或者环境）。把温度调到163℃，持续烘焙六个月，你就能生产出优秀的软件了。任务完成，简单吧？

很多程序员到这里就不再往下想了，他们写软件是为了自己，所以对这个结果很满意，自然也就可以胜利凯旋啦。

但现实世界并不是这样运作的，把“好的软件”当作成功的标准太过狭窄。如果你要养家糊口（即便只是想让简历漂亮一点），你也应该让更多的人来使用，并且喜欢你的软件。软件开发流程并不是到软件发布就终止了，事实上它永远不会结束。人们在使用你的软件，而你也需要作出反馈，然后逐渐改善自己的产品。如果你不去学着掌握这个反馈机制，软件就会消亡。

在这一章里，我们会讨论通常用户会参与的三个阶段。首先你要让用户注意到你的软件——

他们意识到它的存在了吗？在使用软件之前，他们的第一印象是怎么样的？然后你要考虑的是，用户开始使用软件后的体验如何，是否能够满足他们的期望？好不好用？有没有帮到他们？最后，我们要讨论的是，当用户和你的作品紧密联系起来后，怎么能够有效地和他们进行沟通。这些互动都是软件开发里自然循环的组成部分。

要是你不关心这些事情，那么就算软件再漂亮也是不会有用户的。如果真是如此的话，或许你应该考虑一下换个行当了。

管理大众的印象

你听到营销这个词的时候，脑子里反映出来的第一件事是什么？

和大多数人一样，这个词大概会让你想到老奸巨猾的销售员，脸上堆满了假笑，梳着油光闪亮的大背头，他的任务就是要为客户或者产品打造形象。如果说你的软件是一块待售的“生肉”，那么营销的任务就是故意把牛排煎得嗞嗞作响，吸引更多人上前。

为什么这种想法这么令人讨厌？为什么我们对营销人员的想法感到这么不安？

这是因为，对程序员来讲，营销代表的正是工程师文化的对立面。我们追求的是真实，到底

代码能不能编译，到底这个软件有没有某个功能，它到底有没有解决某个问题。我们不会堆砌华丽的辞藻，我们只会陈述事实，然后设法改变它们。营销在我们的眼里除了谎言还是谎言，而我们最讨厌被骗了。我们希望在作决定的时候，看到的是有序、可预期及准确的陈述。

正因为我们对营销的印象是专门歪曲事实的东西，（所以）它违背了工程师对完美的本能追求。我们相信最好的产品一定会赢得一切。我们所谓的“最好”指的是产品从客观上拥有最优异的品质，功能是最有效的，而不是华而不实的电视广告里演的那些东西。但我们一而再、再而三地看到优秀的技术被打败：很多人都觉得 Betamax 要胜过 VHS，Laserdisc也要比DVD更好，或者 Lisp仍然是最好的编程语言（我们只是乘机把这个事实告诉大家而已）。就算是在版本控制工具圈子里，Subversion 也统治了企业市场，尽管很多新系统在技术上具备更多的优越性。

更糟糕的是，我们觉得营销人员总是对客户作出不切实际的承诺，结果弄得软件工程师好像干活不卖力一样，实在是叫人火冒三丈、七窍生烟。

这里我们要告诉你一个坏消息和一个好消息。

坏消息是，你绝不能忽略营销。事实上营销

非常重要，你一定要和它打好交道。好消息是，积极地和营销一起合作是完全可行的，只要你掌握窍门就行了，不需要搞什么乱七八糟的花样。

程序员通常逻辑思维比较发达，而普通人则是理性和感性并驾齐驱。做营销的大多都是操控情感的高手，这也是他们做事情特别有效率的原因：他们把感情掺到事实里来引起他人注意。如果你想吸引别人来使用你的软件，你就必须去关心他们对你的软件的情感诉求，而试图改变人们作决定的方式则是不现实的。


闪亮的
头发
亮白的牙齿
熏死人
的香水
难看的领带
总是西
装笔挺
喜欢对你
比手枪
手机一
直在响
蛇皮的
真皮皮鞋

千万别当这种人

Apple毫无争议地是这方面的霸主，它非常善于让不懂技术的人对科技产生强烈的诉求。我们在2012年的时候曾经这样问过自己：iPhone真地比Android电话好那么多吗？两者几乎具有相同的特性。但假如一个不懂技术的用户觉得iPhone是神奇的，那么它对这名用户来说的确就是神奇的。印象即真实[1]。正如我们先前提到过的“先入为主”的力量。

有时候这会让人觉得既然如此，我不玩了总可以吧，可这是一个你无法回避的东西。如果没有基本的营销策略，你的软件会无人问津。这里是一些你可以掌控的基本概念，而且都是以HRT为基础的。

注意第一印象

当你饥饿难耐地在马路上找饭馆的时候，饭馆的外观是非常重要的。要是看起来很恶心，或者没什么吸引力的话，你根本不会想要走进去。如果看起来很温暖、很友好，门口迎宾的小妹又很客气的话，或许你就会想要试试看那一家。所以千万不要低估了为产品精心设计的初体验，对情感所产生的影响——如果你拆过iPad或者Nest恒温计的包装，你就明白我在说什么了。

对新手来说，你的软件好用吗？界面是否友

善，是不是方便尝试不同的选择？或者反过来，找个专家坐下来用上几分钟你的软件，有没有一种熟悉的感觉？对于刚上手的人来说，你的软件是立刻能让人投入呢？还是要面对一个陡峭地学习曲线，让人欲哭无泪呢？说得再具体一点，即用户在软件启动后30 秒能体验到了什么？光有一个技术上的答案（“她会看到一个选择菜单，还有一个登录框”）是不够的，还要有一个情感上的答案。新用户在一分钟之内的感受是怎么样的？是感到有用呢，还是觉得很迷惑？你要怎么做才能改善这种体验？回过头来再看看产品的网站，它是不是够专业、够友好，就像是一个漂亮的店面一样？这些东西都需要你非常认真地去对待，这样用户才会把你的软件当回事儿。

承诺的时候要谨言，做产品的时候要超出预期

这就是说，不能让营销部门的人口无遮拦地放卫星。要是用户问起新特性或是发表时间，一定要抓住机会，给出最最保守的估计。如果你放任营销部门的人说大话，你就会陷入《永远的毁灭公爵》的那种境地——这款软件跳票了15年，彻底变成了一个笑话。但只要你能把第一手的（也是更准确的）消息发布出来，用户一定会感到非常兴奋的。Google就精于此道，它有一个深思熟虑后的政策，即任何产品都不预先发布它会

拥有什么新特性，所以当人们看到新特性突然出现的时候，往往都会觉得很惊喜。这也防止出现公司内部拼命赶进度，试图追上不现实但是已经发布出去的日期的这种事情。只有在真正准备妥当，可以使用的时候，才能发布软件。

尊重业界分析师

很多程序员都不喜欢媒体行业——他们就是穿着马甲的营销嘛。每当有销售杂志或者研究机构上门的时候，很多公司宁可丢下手头的一切事情，也要满足他们的需求。因为大多数公司相信一篇好的（或者坏的）评论会让大众对产品的印象发生相应的改变。而工程师却往往非常厌恶这种力量，以及公司所表现出来的顺从。

例如，Apache软件基金会（ASF）的成员曾经有一段时间就和分析师有点合不来。每次有分析师希望ASF能提供一份符合工业标准的白皮书来描述一下 Apache HTTPD 服务器的时候，通常都会得到这类不屑的回答：“自己去读网站上的文档，所有人都是这么做的。”虽然这满足了开源程序员内心深处的精英情结，但是对公众印象来说这却是非常不利的——特别是对企业用户来说。不过最终ASF的公关部门还是努力说服了那些社区成员，让他们以更加积极地态度去应对分析师。消极抵抗整个大环境（不管它有多么讨厌）是没有意义的。对待这些人要给予特别照顾

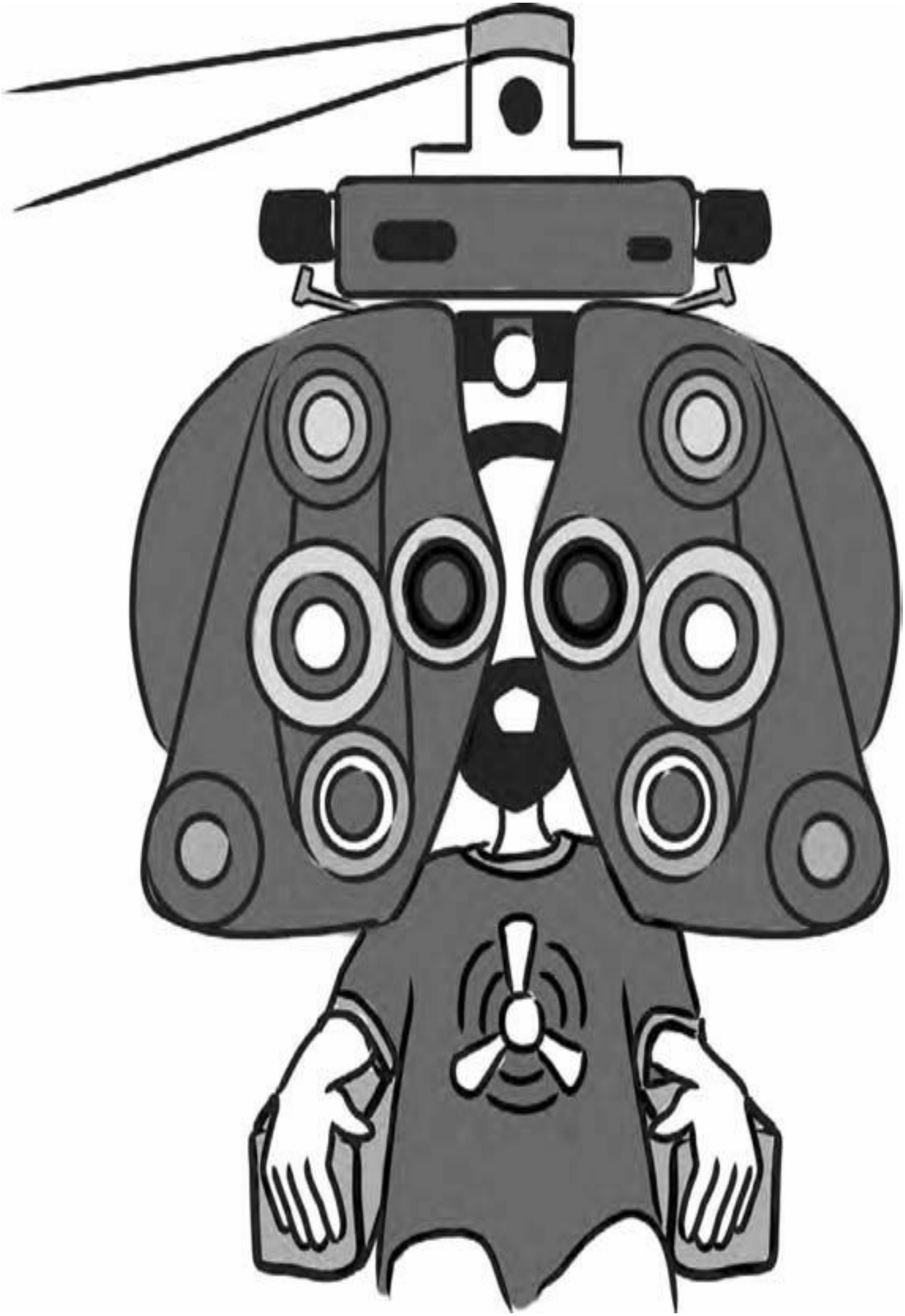
么？可能没什么必要。但有必要仅仅因为个人喜好就故意刁难他们么？估计也没什么必要吧。这么做伤害的只是你自己罢了。树敌的时候要小心。

软件好不好用？

这里是一个铁的事实：除非你开发的是软件工具，否则工程师并非你的用户。因此，身为一个工程师，你绝对不是评估软件可用性的好人选。对你来说再平常不过的界面，却可能让你家隔壁不懂技术的邻居抓狂不已。

如果我们将“成功的软件”定义为“有很多人用，而且他们都喜欢你的软件”，那么你一定要把注意力放在用户身上。Google有一句非常著名的格言：

关注用户，其他的东西自会随之而来。



用户才应该是你关注的焦点

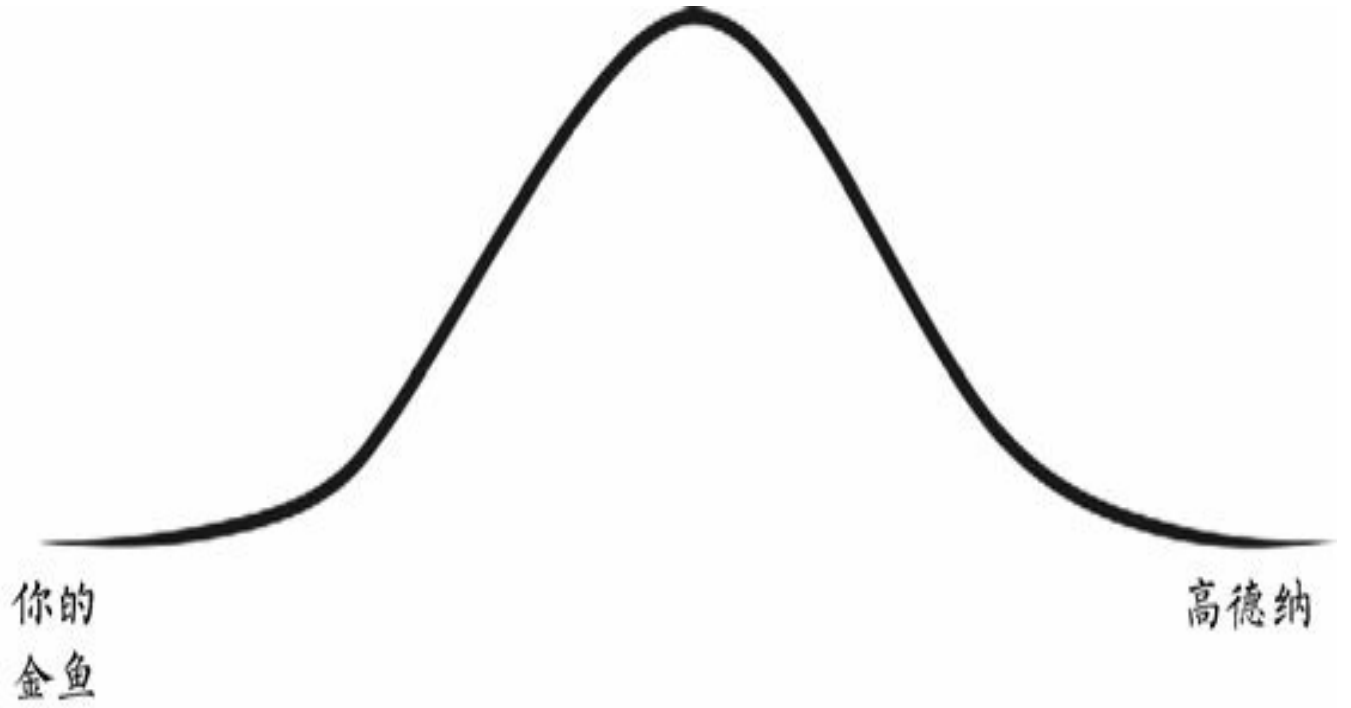
听起来是不是很做作？但我们作为Google的员工，已经亲眼目睹了这条原则在很多项目里发扬光大。我们见证了这条真理决定了项目的成功与失败。

Google最大的突破之一就是开始评估搜索广告的有效性。用户如果去点某个广告，那一定是因为这个广告对他有用。要是这个广告没人点，那么它一定很烦人，或者完全没用。因此坏的广告会从系统里移走，并且反馈给广告商让他们改进。从短期来看，这么做似乎会造成反效果：Google主动拒绝了收入来源。但是由于将关注的焦点放在了搜索用户（而不是广告商）的身上，结果从长远来讲，Google搜索广告系统的效果和利用程度都大大提高了。

下面我们来讨论几个能把注意力直接放在用户身上的方法。

选择你的用户

首先，以技术水平为标准，你的用户应该落在下面这张图的范围里。

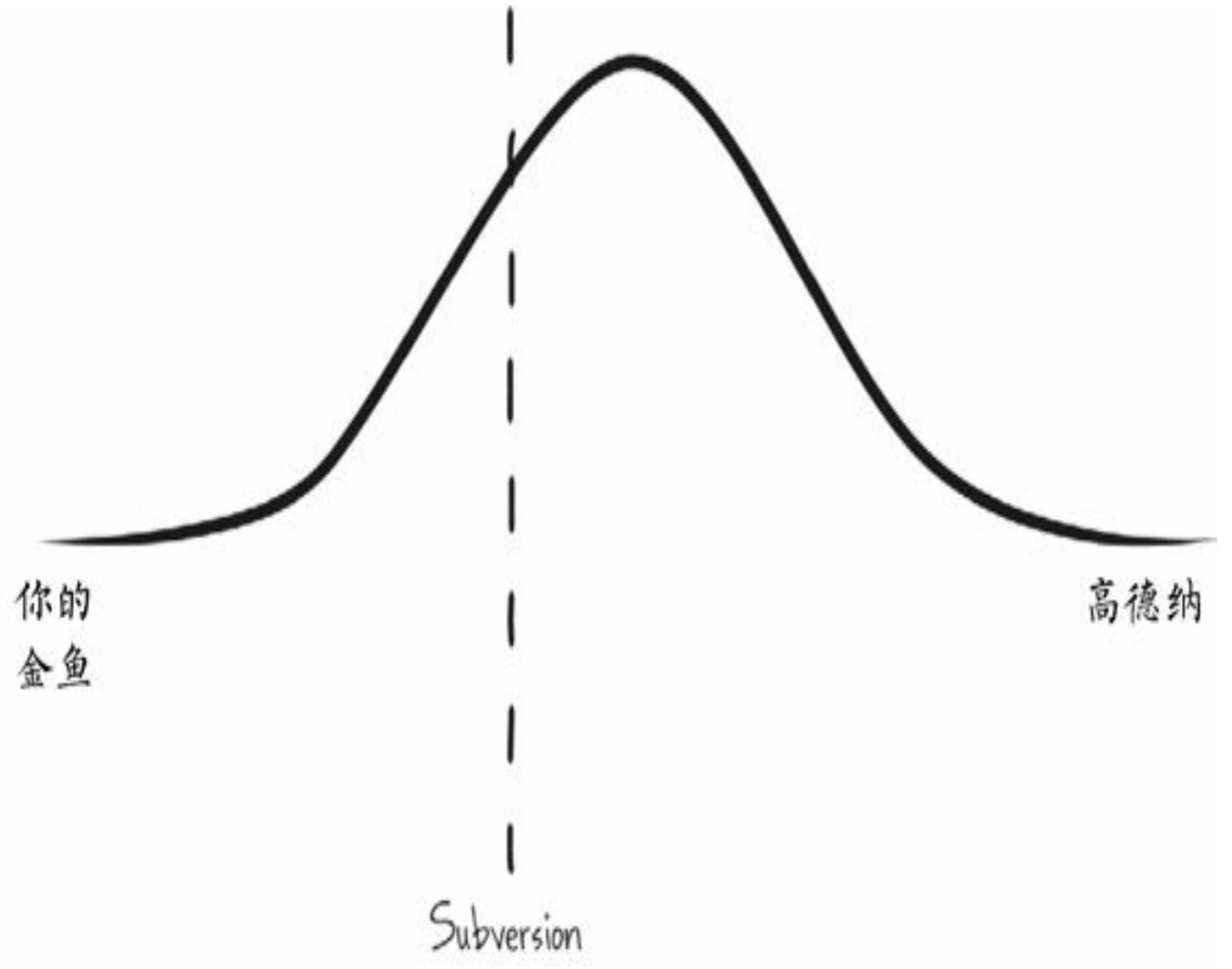


软件的潜在用户

假如你要在图上画一条竖线来表示哪部分用户是你的软件的目标群，你会把它画在哪里？如果画在中间，那么表示有一半的计算机用户会喜欢你的软件（比如，在线右边的人群）。

我们对版本控制软件非常熟悉，所以这里就用它来展示Subversion 在我们眼里的情况。它的界面非常简单，就算不懂技术，也有相当一部分人能学会，而分布式版本控制系统就要复杂得多。Git和Mercurial都是非常成熟的产品，拥有的特性也几乎相同，但是（在本书撰写之时），Git在主流极客程序员中间得到的关注程度要高得多，我们自己则比较倾向于Mercurial，因为它的界面要简单得多。Mercurial和Subversion非常类

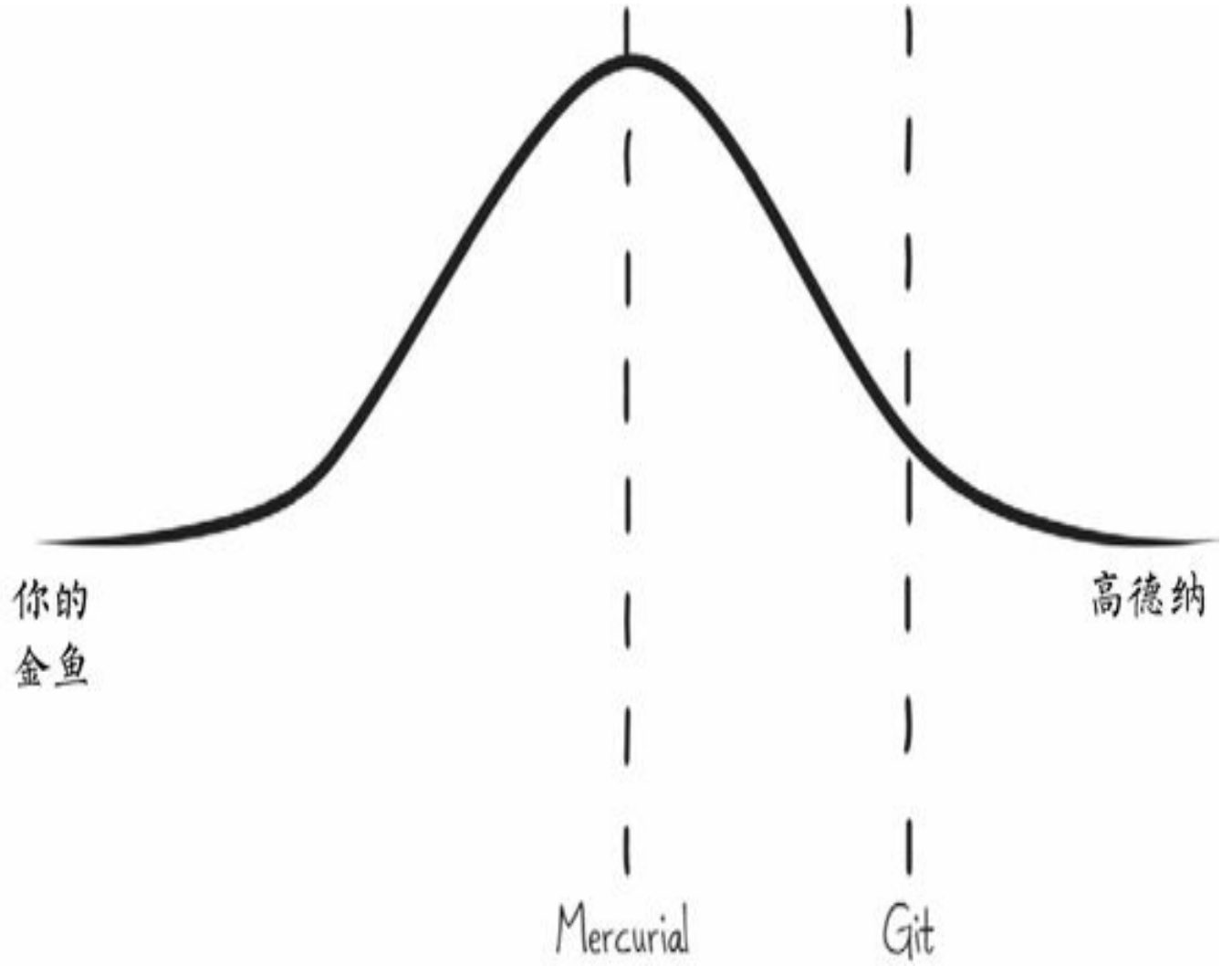
似，一致性保持得非常好，命令和选项的名字也都很直观，它把实现细节都隐藏了起来。相反，Git 的界面则更像是暴露在外面的回路和电线，绝大多数用户最终都要去了解它的内部架构，否则就没办法有效地运用它！我们知道很多Git 用户都要去记忆一些“魔法”命令，虽然当下可以工作，但是总是要抱有一丝担心，害怕万一不小心行差踏错自己做的东西就会被软件吃掉。



Subversion有多易用

这就是说，Git 之所以能在主流极客之间流行的一大部分原因是和类UNIX操作系统一样：难学，但同时提供了强大的功能。很多主流极客都愿意为此牺牲一点易用性。

但是不要忘了，UNIX 和 Git 可以说是罕见的例外。这种软件针对的是技术能力高超的用户，而大多数软件开发的目标是要让竖线尽量往左靠。这样的流行软件有Gmail、Facebook，以及Apple的iOS。通常来说，用户越多越成功（公司赚的钱也就越多），所以在考虑用户的问题的时候，一定要搞清楚到底什么人才是你的用户。软件是不是能让尽可能多的人觉得好用？这就是简单的、照顾人的界面——还有其他一些东西，比如漂亮的文档和方便的入门教程等很重要的原因。



Git的易用程度如何

考虑入门的门槛

现在我们来考虑一下软件的新用户。第一次运行软件的难度有多高？如果你的用户无法轻松让它跑起来，那你是吸引不到他们的。新用户通常不会考虑你的软件和竞争对手孰强孰弱，她只是想要用它来完成工作，而且要快捷。

为了说明这一点，不妨来看看流行的脚本语言。大多数程序员都觉得无论是Perl、Python，还是 Ruby，都要比PHP来得好。他们的观点是，

Perl、Python、Ruby 的代码比较好读，长期来讲维护也方便，拥有更多成熟的软件库，本质上更安全一点，因此在公共Web上也比较放心。但是PHP的流行程度却远超它们——至少在Web开发上是这样的。为什么？因为任何中学生都能依葫芦画瓢，抄一点朋友的网站代码，而完全不用读书、看大量的教程，或是去学习一些严谨的编程模式。它就是靠这种修修补补的工作来传播的：只要不断地摆弄自己的网站，然后从朋友那里学一下各种PHP技巧就可以上手了。

另一个案例是文本编辑器。程序员应该用Emacs还是vi？这其中的差别有关系么？实际上没什么关系，那为什么有人会偏好其中一个呢？这里我们讲一段真实的故事。本在刚刚学UNIX的时候（1990年实习的时候）想要打开一个文本编辑器。他用vi打开了一个文件，这是他第一次启动vi，然后挣扎了将近 20 秒——他可以在文件里上下移动，却没办法输入任何东西！vi用户自然知道要修改文件必须先进入“编辑”模式，但是这对于新手来说却是无比糟糕的体验。而当本启动Emacs后，他却可以立即开始编辑文件，就像在他熟悉的 DOS 文字处理器上一样。由于Emacs的默认行为和之前的经验重合，本立即决定从此成为一名Emacs用户。以此就作为选择一样产品的理由似乎有点傻，但这种事情真的每天都在发

生！使用产品的第一分钟是至关重要的。

当然，摧毁第一印象的方式还有很多，比如在初次启动软件的时候让用户去填一张巨大的表格，或是强迫他们设置一大堆偏好。强迫用户去创建新账户也很让人讨厌，因为这暗示了用户在还没有做任何事情之前，就要被迫签长期协议一样。这些细节都会迫使用户远离你的产品。

如果你的产品是网络应用，那么它加载的速度一定要快！我们已经习惯了速度快的网页了。每次有人叫傅攀勃去看看某个新网站的时候，如果那个网站在三到四秒内加载不完，他通常就失去兴趣退出了。没有什么理由好讲的，程序员迫使用户在门口等待是非常讨厌的障碍。浏览器让我们非常容易把注意力转到其他的东西上去，相对于等待加载一个页面，我们有更重要的事情要做。

再举个几乎没有门槛的好例子，TripIt 的网络服务，这是一个专门用来管理行程的服务。要使用这项服务非常简单，只要把你的行程确认 E-mail（机票、酒店、租车等）转发给plans@tripit.com就行了，不再需要额外的动作。它会自动为你创建一个临时账户，解析你的E-mail，然后创建出一个非常漂亮的行程单，完成这一切后，再发E-mail通知你。这就像是一名随时随地待命的私人助理，你要做的只是转发一些

信息而已！只要动一动手指，你就被吸引进来，可以像用户一样浏览网站了。这时，创建真正的账户也变得顺理成章了。

如果你想要知道自己的产品入门门槛高不高，不妨做一点简单的试验。随便找一些人（懂技术的和不懂技术的都要）来用你的软件，观察一下他们在头一两分钟内的反应，保证有惊喜。

衡量使用数量，而不是用户数量

除了用户规模和易用程度之外，你还应该考虑如何衡量使用数量。注意，我们说的是实际使用的情况，不是安装或者注册的用户数量——你要的是大量用户来使用你的产品，而不是很多人来下载你的产品。你可能经常会听到有人说；“嘿，我的产品有三百万的下载量——那可是有三百万的满意用户哦！”等等，别着急下结论。这三百万里究竟有多少人真的在用你的软件？这才是我们所说的“使用数量”。

举个极端的例子，有多少机器上装有 UNIX 的备份工具“ar”？答案：基本上每台类UNIX系统都可以算上，这包含了各种版本的Linux、Mac OS X、BSD等。那么有多少人在用这个程序？甚至，有多少人知道它的存在？这可以说是一个拥有百万装机量，但是接近零使用量的软件。

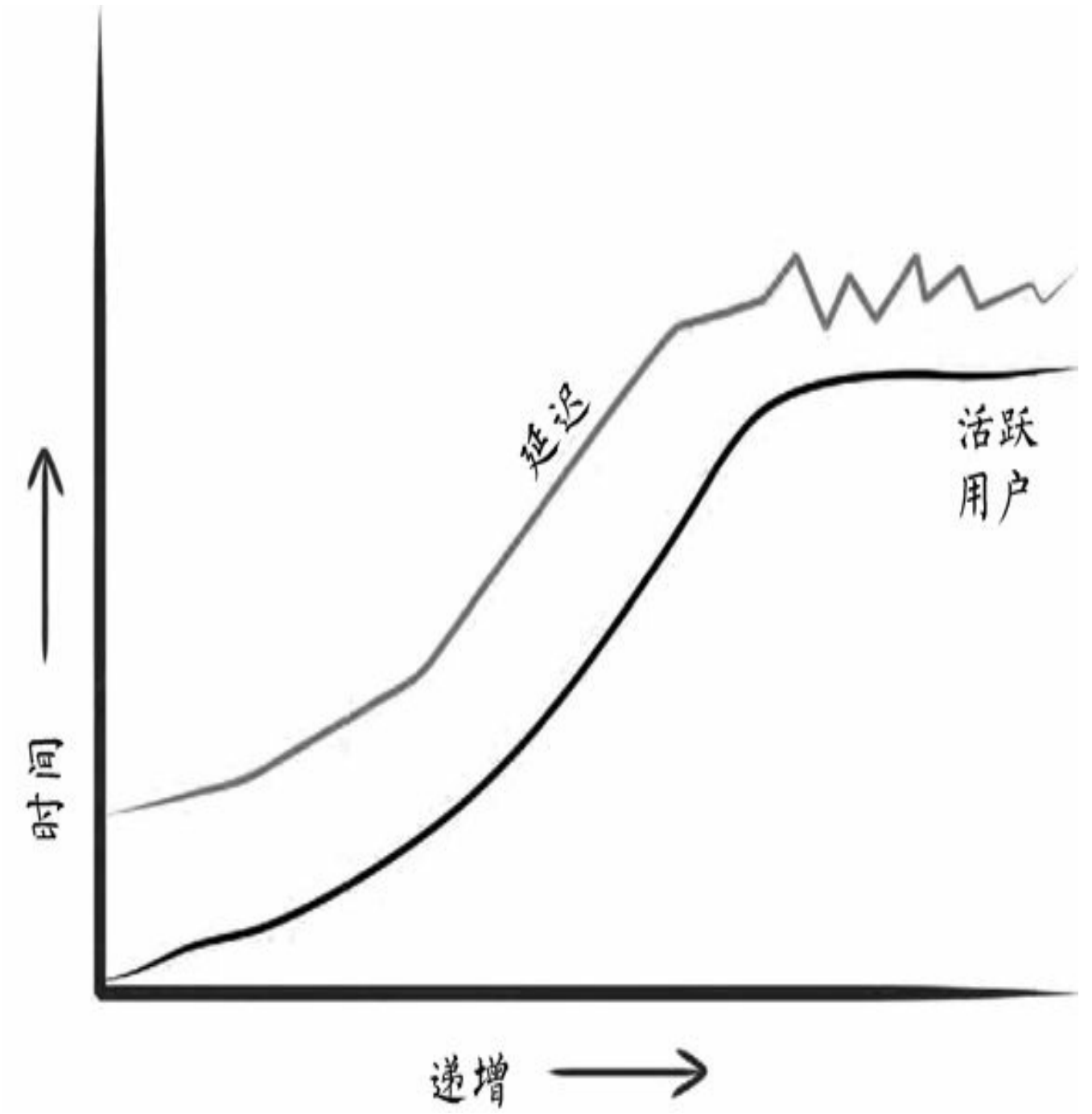
很多公司（包括Google）都花费了大量时间去衡量使用数量，常见的指标有“7天活跃度”以

及“30天活跃度”——即，有多少用户在过去一周或者一个月内使用了软件。这些重要的数字能告诉你软件的表现情况到底如何。不要在意下载数量，找到衡量活跃程度的方法才是正道，这才是帮助理解软件的真正指标。

速度很重要

回到网页速度的问题：大多数程序员都严重低估了应用程序速度（或者说得更科学一点，延迟）的重要性。程序响应速度快的时候，会对用户产生深层次的阈下刺激效果[2]。由于感觉流畅，他们使用软件的频率会上升，这样不知不觉间软件就变成了日常生活中的一部分。相反，反应缓慢的应用程序会随着时间变得越来越让人郁闷，用户会渐渐疏远这样的软件，有时候连他们自己也没有意识到这一点。

产品在刚发布的时候，使用数量的增长总是令人激动的，但是一段时间后这个数值就会进入平台期——在图上看就是一条横线。这时营销部门的人常常就会介入进来，大声嚷嚷着要增加更多特性、更漂亮的颜色、更好看的字体，还有各种“流行”的动画元素等。但有时候（遇到瓶颈的）罪魁祸首往往就是延迟，程序变得很卡，用起来叫人绝望。下面的这幅图表明，用户的参与程度随着延迟的上升而下降。



观察随时间变化用户参与程序的变化

这是发生在Google的真实故事。有一支工程团队在某一天发布了一些对Google Maps的改进，大大改善了它的延迟。没有任何公告，也没有在任何博客提到，整个发布都是默默地秘密进行，

但是活跃图表却显示在两天之内用户的使用数量跳升了一个台阶（而且没有回落）。这就充分说明了心理对用户所产生的影响！

在提供Web服务的时候，再小的延迟改进也是很重要的。假设你的主应用程序页面需要750毫秒载入。看起来还蛮快的吧？不会给任何用户带来困扰。但是如果你能把载入时间下降到250毫秒，那省下来的半秒时间加起来其实是很惊人的。比如说你有一百万用户，每人每天会发送20次请求，这就等于帮用户省下了116年的时间——别再谋杀你的用户了！改善延迟是提升使用数量、让用户满意的最佳途径之一。正如 Google 的创始人喜欢说的：“速度也是特性。”

不要大而全

你的软件是不是做太多事情了？乍一听这个问题好像有点蠢，但是市面上有些糟糕至极的软件之所以糟糕就是因为它们太野心勃勃了。它想要做到大而全，以适应所有人的需求。而最好的软件之所以成功是因为它解决了一个很特定的问题，而且解决得非常漂亮。与其拙劣地解决所有问题，还不如解决大多数用户都真正会遇到的问题，然后做到最好。

我们常常会嘲笑一些在杂志广告上看到的工具：嘿，快看！这是野营灯，它内置了天气预报收音机！……而且，呃，还有一个内置电视，

呃，还有秒表、闹钟，还有……嗯？真是一团糟，完全搞不明白这是个什么东西。你应该把自己的软件想象成一台简单的吐司烤箱。它能烹饪任何食物吗？绝对不可能，但是它可以做出最常见的食物，几乎任何人都用得上它……而且不会被弄得晕头转向。当好一台吐司烤箱吧。少即是多。

这到底是个什么玩意

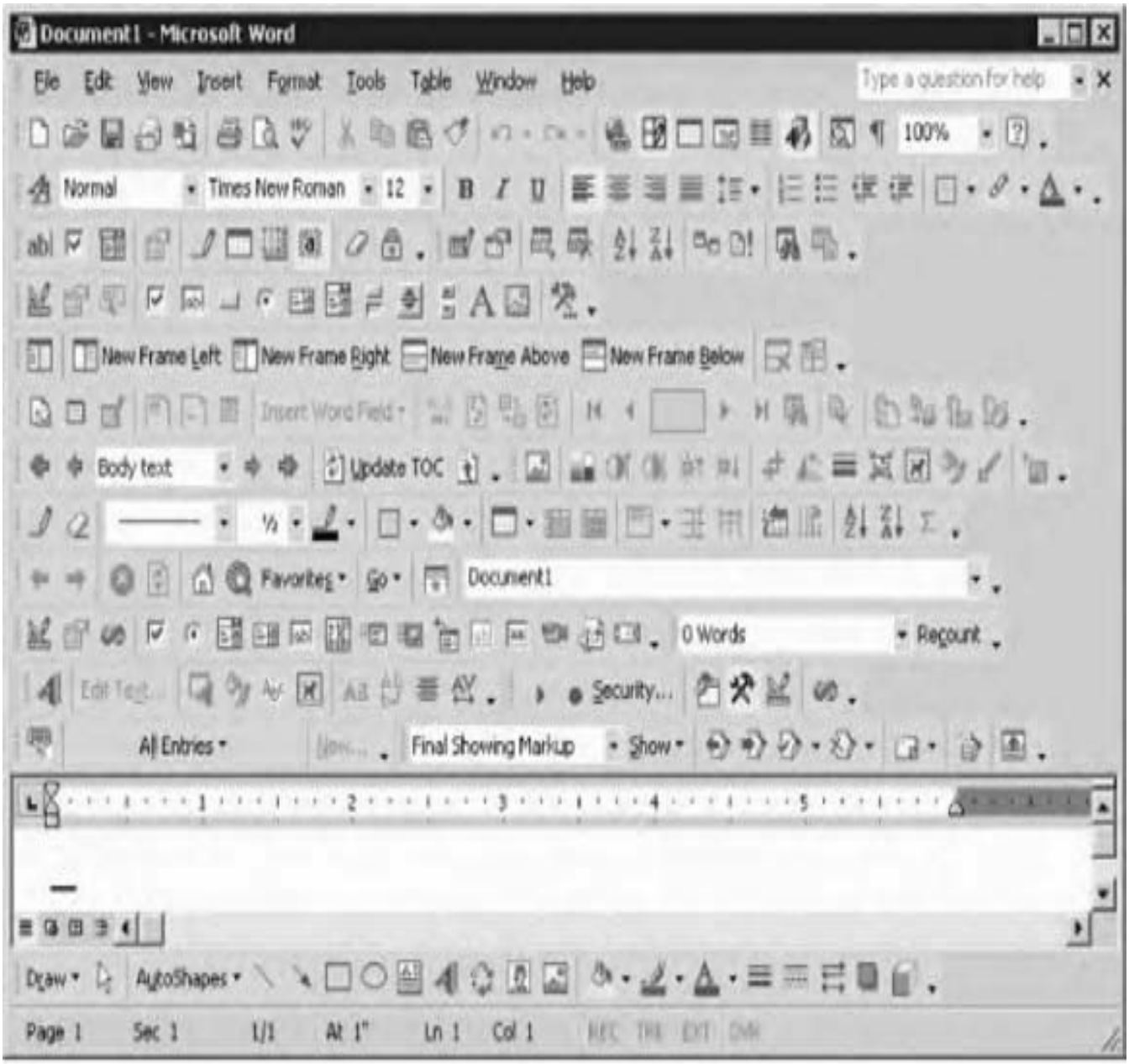
别偷懒

懒惰是千万要小心的陷阱。有人或许会争辩说懒惰是程序员的美德，因为程序员会因此去实

现自动化来提高效率。但是同时，懒惰也会导致程序员写出一些很糟糕的程序，给用户造成巨大的痛苦。写出方便用户的软件对程序员来说往往是很麻烦的事情。但是你关注的应该是用户，而不是自己写代码方便，再麻烦你也得忍着。

懒惰的典型反面教材就是展示太多的选择给用户。大家都喜欢嘲笑 90 年代末微软的 Office 产品：按钮条！它们一下子就让你可以选择任何菜单项……太方便了！用户界面设计师非常喜欢拿这个案例讲笑话，特别是下面这种极端情况。

太多选项只会让人无所适从。看起来不但很吓人，而且也很讨厌。市面上有很多书籍都讨论了为什么太多选择会让人焦虑和痛苦[3]，甚至在软件的偏好对话框里也要非常小心。（你知道Eudora吗？这是一个非常流行的email客户端，它有 30 个不同的偏好面板。）如果一定要填表格，你应该多为用户考虑一点：自己处理多余的空格、标点符号等。不要强迫用户去做文字解析的工作！这也是尊重用户的时间。每当看到程序员完全可以把东西做得更友好、更易用却懒得那么做的时候，是非常让人恼火的。



不要偷懒，一下子就把所有的选择展示出来

隐藏复杂性

“但是我的软件本身就很复杂啊”，你可能会想，“它解决的就是一个复杂的问题。既然如此我为什么要把它隐藏起来？”这的确是一个合理的质疑，但这也是优秀软件设计里最大的挑战之一。优雅的设计应该保持简洁，化困难为可能。

即便你的软件在处理复杂的逻辑，它也应该让人觉得流畅简洁（我们关注的是用户的情绪）。

这就是所谓的“隐藏复杂性”。将复杂的问题分解开来，然后隐藏住复杂性，或者至少也要设法让软件看起来很简单。

我们还是拿 Apple 来举例。Apple 的产品设计可谓登峰造极，它最聪明的就是巧妙地解决了管理MP3专辑的问题。iPod出现之前，一些蹩脚的产品都试图在移动设备上直接管理音乐。Apple的天才之处就是意识到在那么小的屏幕上管理MP3太困难了，还是干脆把它移到电脑上吧，于是iTune 应运而生。用户在电脑上（有大屏幕、键盘和鼠标）来管理音乐专辑，然后在iPod上听音乐。iPod因此变得简洁优雅，而管理音乐也不再让人头疼。

Google则是另一个经典案例。Google的搜索界面（以及入门门槛）可以说几乎不存在：它就是一个魔法一般的输入框。而在那个小框框后面，却有成千上万、遍布世界各地的电脑同时响应你的每一次击键，为你提供自动完成的提示。当你按下回车的时候，搜索结果早就已经在后台整理渲染好了，随时可以展示在你的屏幕上。在那个文本框背后的技术含量是惊人的，但是所有的复杂性都被隐藏了起来。

Apple和Google的界面都神奇得像魔法一

般[4]。这是非常了不起的目标，是所有软件工程师追求的标杆，这基本上就代表了软件的易用性。

最后，我们还要提一点关于复杂性的警告。尽管掩盖复杂性是值得赞赏的行为，但是它却不应该是软件封闭、束缚用户的理由。掩盖复杂性通常涉及创造一些漂亮的抽象，作为程序员，你应该准备好这些抽象会发生“泄漏”。浏览器显示404错误就是抽象泄漏的一种，因为（浏览器编制的）幻象破裂了。但不要慌张——你完全可以假设抽象一定会泄漏，然后准备好应对之策就可以了，比如为其他程序员提供API就是一个好办法。或者为真正的高级用户提供一个“专家模式”，让这些想要绕过抽象的用户可以有更多的选项和选择。

保持界面灵活，有办法绕过抽象，这些很重要，而访问用户数据的能力也同样重要。傅攀勃为此投入了极大的热情，让Google的产品能够“解放数据”——即，用户可以轻易地从应用程序里导出自己的数据，然后带着它们离开。不管界面有多优雅，软件都不应该绑架用户。由于用户可以取得自己的数据，做任何想做的事情，这就迫使你老老实实地竞争：人们选择你的软件是因为他们想要那么做，不是因为被（软件）绑架。这样建立起来的信任，是你最神圣的资源。（下节还会提到。）

管理和用户之间的关系

OK，你的软件第一眼就让人惊艳，非常容易上手，使用起来也非常舒服、非常好用，那么接下来的几个月呢？你怎么数年如一日地和每天使用软件的用户进行交流？

或许你不相信，其实很多用户都想和你的公司或者团队建立起某种联系。满意的用户会想要知道你的软件的演化的情况，愤怒的用户则需要一个地方发泄不满。程序员犯的最大的一个错误就是把做完的软件丢在一边，不再去听取用户反馈。

客服和营销一样，也是一个通常让人产生负面联想的词汇。每当提到“客服”行业，总会让人想到在咖啡店里的咖啡师，或是满满一屋子的人，机械地接听支持电话这样的画面。但是实际上客服并非这种令人讨厌的、枯燥乏味的工作，也不是什么低等的（缺乏内涵的）工作。它是一种处世哲学——一种和用户之间建立长久联系的方式。软件团队应该主动参与进来，而不只是被动地对来自外界的抱怨作出反应。

软件工程师通常惧怕直接和用户打交道，他们觉得：“用户什么都不懂。他们很讨厌，完全无法沟通。”但是尽管没人强迫你爱上每一个用户，但其实用户是很希望有人听到他们的想法的。就算他们的建议空洞无物，或是抱怨的东西

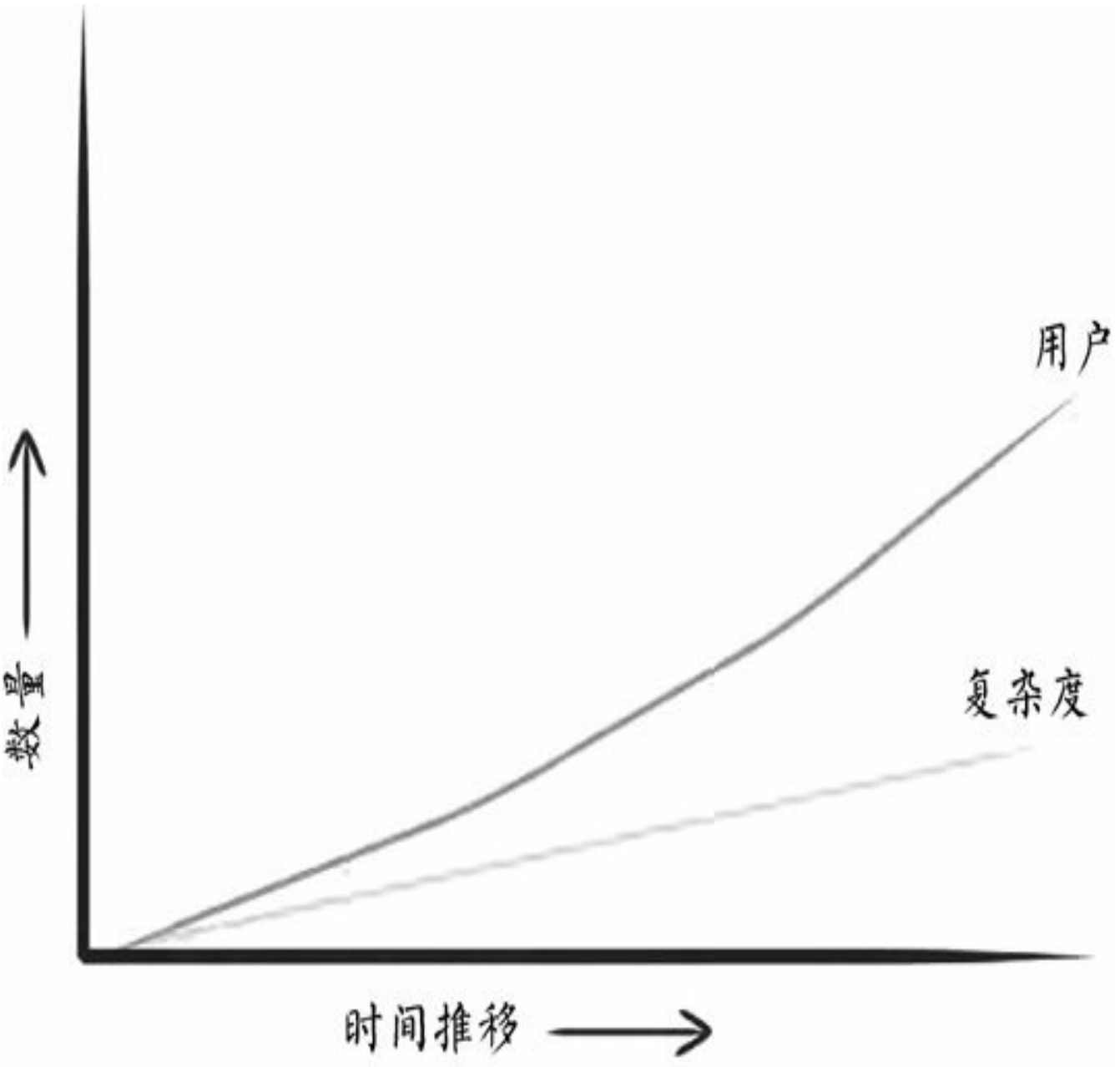
完全不知所谓，这些都不重要，最重要的是你要让他们知道，他们说的你都听到了。越是让他们参与到讨论和开发过程中来，他们就越忠实、越开心。你不一定要同意他们说的东西，但是你一定要听。在这个社交媒体的时代，很多公司都已经迅速意识到了这一点——摆出一副冷冰冰的大公司的派头是没用的，但只要以普通人的姿态出现，就足以缓和用户的焦虑。人们非常乐见公司开明地展现出HRT的姿态。

倾听用户比什么都重要

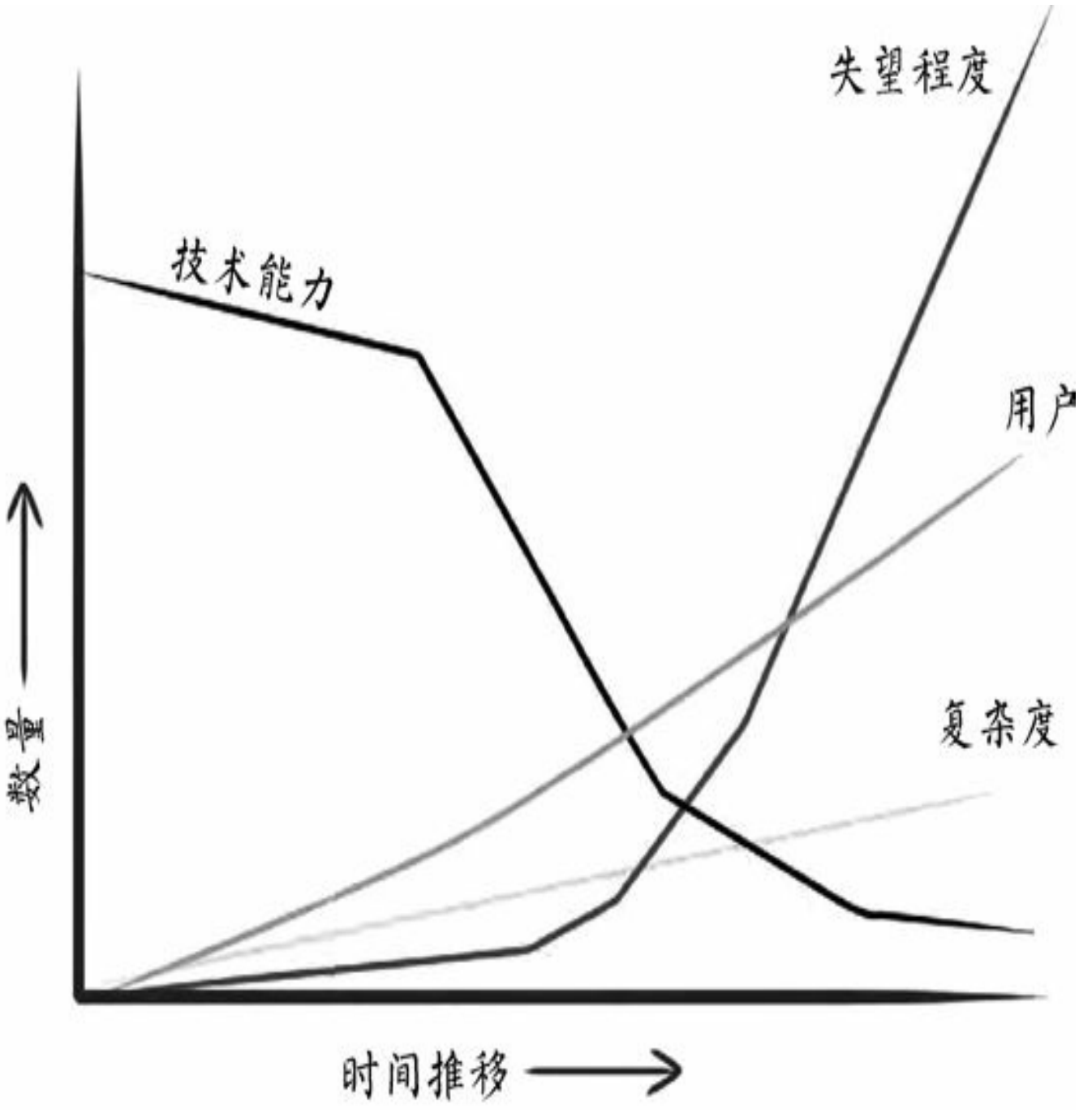
这里，我们再画一张简单（但是不太科学）的图来说明坚持用户管理的重要性。随着时间的推移，软件会收获越来越多的用户；同时，随着产品的“改进”，它也会变得越来越复杂。

问题在于，当用户数量上升时，他们的平均技术水平会递减，因为有越来越多的普通大众变

成了你的用户。再加上不断增加的复杂度，用户失望的程度会直线上升。



衡量产品的用户数量



衡量用户满意度的趋势

失望越多，抱怨就越多，用户就会更不满，希望和软件工程师公开交流的意愿就越强烈！

很多公司都会尽一切努力在程序员和用户之间设置各种行政障碍，他们用自动语音应答的方

式来引导和记录用户的投诉，然后安排了很多层没有接触过软件代码的人来跟踪这些“求助记录”。信息只能间接地在这些层级中间穿梭，弄得好像如果直接和这些危险的暴民接触的话，会危及到程序员（或是会莫名其妙地影响他们写代码）一样。这就是用户会觉得自己不受重视，而同时程序员却被完全孤立起来的原因。

其实更好的沟通方法是直接接触用户。让他们通过公开的bug跟踪系统来投诉，同时也通过这个系统来直接回复他们。为他们建一个邮件列表，让他们互相帮助。如果你的产品可以开源，那也能有很大的助益。你表现得越有人情味，用户就会越信任你的产品，失望的情绪也会随之减少。你应该注意用户有没有在以某种意外（同时非常惊艳）的方式使用你的产品。而只有通过真正的对话，你才能了解他们到底是怎么使用你的软件的，或许他们的用法非常聪明、完全出人意料呢。

不要有优越感

一个常见的误解就是用户都是笨蛋，这加深了我们和用户直接交流的恐惧。毕竟写软件的又不是他们，他们“懂什么”呀，对吧？可当你最终有机会和用户交流的时候，千万要记住不要有优越感。要求普通大众懂电脑是不公平的，很多聪明人只是把电脑当作工具，仅此而已。他们对调

试没有兴趣，也不想用科学的方法诊断问题。别忘了，我们大多数人也都不知道怎么拆开并修好自己的车；把你的用户当作笨蛋就好像汽车机械师觉得你是笨蛋一样，毕竟你也不懂怎么自己造一个变速箱，甚至懒得去想怎么诊断变速箱的毛病。汽车对你来说就是一个黑盒——只要能开就行了。对大多数人来说，电脑（包括你的软件）同样是一个黑盒，用户不想参与你的分析过程，他们只想要完成自己的工作罢了。这和智力没有任何关系！他们应该得到尊重。

保持耐心

由此我们必须学会保持巨大的耐心。大多数用户不具备足够的专业术语把问题表达清楚，光是学习、理解他们想说什么就需要很多年的经验，问问那些曾经在电话里帮爸妈修过电脑的人（基本上你们大多数人都中枪了）吧，至少有一半的对话都是在试图让对方明白自己在说什么。很多人不懂什么是浏览器，他们觉得浏览器就是电脑的一部分。他们把应用程序描述成行为，或是把屏幕上的图标说成是某种神秘的工作流的名字。就算是最聪明的家伙，在这种时候也只能设法创造出一套符合他们自己逻辑的词汇来解释电脑是怎么运行的，他们会用自己想象出来的类别和规则来诊断问题。

父母：“我觉得你的电脑速度慢是因为硬盘

满了。”

你：“你怎么知道硬盘满了？你检查过了吗？”

父母：“对呀，你看，屏幕上全是图标，所以很可能这就是我没办法下载 E-mail 的原因。要不我试试删掉一些 cookies，看能不能腾出一点空间来？上次这么做就解决问题了。”

你：（扶额）

这里的关键之处就是要学会理解他们到底想说什么，而不是硬把他们说的东西按照字面翻译过来。这不但需要语言翻译的能力，还需要一些情商。哦，还有读心术。

傅攀勃就有过这样的一次经历，他的祖母（在电话里）问他说：“布莱恩，你爷爷的那台旧电脑还有用么？”傅攀勃回答说没有了，那台Mac Classic实在是太老了，连网络都没法连接——最好安全回收了。结果她答道：“好吧，那我只会在削铅笔的时候才打开它。”

傅攀勃听得一头雾水，好一会才反应过来要好好问清楚到底她想要说什么！

结果发现原来那台Mac和他祖母的电动削铅笔机接在同一块排插上，他的祖母每周都会跑去那间房间，打开排插电源来削铅笔。这时Mac就会滴一声地开始启动，削完铅笔后，祖母就会切断电源离开房间，这等于强行关机，Mac还来不

及启动就被关掉了[5]。这是一个非常经典的例子，不懂技术的人会设法用有限的词汇和各种当下和电脑有关的现象来解释发生了什么。

很多人都还对Google的搜索服务抱有某种神奇的观念，很多人都把它当作是自己电脑的一部分。2005年的时候，当我们告诉别人我们是Google工程师的时候，他们总会露出迷惑的表情："喔！我根本不知道有人在那里工作？！"还有一次，傅攀勃祖母的一个朋友听到全公司都要去度假滑雪（那时候公司规模还很小）的时候，感到非常生气。"太可怕了！那怎么行，怎么可以都去滑雪？"她问道。"这样谁帮我搜索呀？"显然，那时候的Google太玩忽职守，没有留下足够的接线员来保持正常运营。

请记住这台饱受折磨的可怜的Mac

营造信任和愉悦的氛围

此外，在和用户交流的时候还有两点需要格外注意：信任和愉悦。

信任是很微妙的东西。我们在提到HRT的时候已经讨论过信任这个话题了——如何让同事感受到你对他们的信任。而这里我们要谈谈怎么从用户那里获取信任。所谓用户信任你的团队（或

者公司）基本上是很感性的，很少会有人说：“我信任某个产品是因为它具备什么什么优点，所以我可以放心使用它。”他们相信你是因为之前在和你的交流过程中所积累起来的感觉达到了一个良性的状态。

看看你周围的朋友和家里人吧。他们中间有几个人真正信任修车师傅？今时今日，这个数字几乎可以说为零。因为我们多年来已经饱受这种“mailboxing”的困扰：每次去定期保养（比如说换机油）的时候，都会莫名其妙地多做一堆其他的保养，所以大家都不再相信修车师傅，他们只知道抓住一切机会赚钱。而信任这种东西，一旦失去，就再也变不回来了。

这可以说是为了眼前利益牺牲长久关系的典型例子。哪怕只是偶尔让客户小失望一下，随着时间累积，最后他们还是会觉得受到了极大蔑视。如果每次你的团队都能帮到他们，或者至少能作出一点反应，就能在客户的脑海里一点点建立起信任的感觉，就好像蛋糕店的店员在你买了一打甜甜圈的时候再额外送你一个（在新奥尔良，我们称之为“lagniappe”，就是法语的“小赠品”）的话，你肯定会很高兴吧。如果能多年如一日地坚持下来，这种信任的感觉就会不断累积，最后用户只要一想到你的产品，心里就会涌起一种温暖、难以言喻的感情。

但是信任也是很脆弱的东西，你可能在一夕之间就失去它——就像是冲动地买了一件名不符实的东西，结果一下子就让荷包见底一样。假如你的公司做出任何不尊重用户的举动（即便只是无意为之），也可能一夜之间失去用户的信任。

最近的（2011年末，就是撰写本书的时候）的一个例子就是Netflix曾经差点毁掉和用户的关系。Netflix是一种在线播放电影的流媒体服务，同时也可以通过邮件租借DVD，由于简单、便利和新颖，十多年来它越来越受欢迎，而且价格也很便宜。到了 2011年初的时候，它已经拥有超过2 300万用户。

然而公司的某些决策层觉得他们的 DVD 和流媒体服务其实是两种不同盈利模式的业务，经营需求等都不尽相同，于是他们决定将这两个业务分开收费，有些用户的月费将提高60%，用户彻底愤怒了。Netflix进而再宣布，为了更清晰方便地展开业务，它将拆分公司。可对用户来说，这只是意味着“原本只要付一份账单，现在要付两份了”。最终他们意识到这是一场公关灾难，于是立即撤销拆分公司的决定，但那时已经太晚了。伤害已经造成，无可逆转。尽管总体仍然是增长的趋势，但是他们在三个月之内失去了80万用户。只是几个看似简单必要，但其实和用户毫无关系的商业决策，一下子就毁掉了十年来建立

起来的信任。

信任是最神圣的资源，必须悉心呵护、步步为营。任何举动都要三思而行。决策的时候，眼光要长远，不要只注重眼前利益。

同样，愉悦是另一种能大大改善用户关系的感情。它能提升温暖、无法言语的感觉，让团队更加人性化。

首先是别太把自己当回事。Google有一个传统，就是在愚人节的时候宣布一些古怪的产品，比如有一年，YouTube首页上的视频都会被转向一个搞怪视频。你还可以看看www.woot.com，这是一个打折网站，每天都有不同的折扣，它的广告词写得非常自嘲，充满了古怪的幽默。

不时地给用户送一些小礼物

尽量给用户带来一些惊叹奇妙的惊喜。（毕竟这就是愉悦的定义，不是吗？）虽然Google是严肃的计算机科学机构，但每到节假日或者纪念

日的时候，还有什么能比时不时地“装疯卖傻”一下更让用户兴奋呢。这些融入人们每天生活中的小情趣，却能激发无数的反响和茶余饭后的谈资。

当然，有时小小惊吓也能起到激发用户的作用，只是手法上必须幽默一点。某家做社交网络的公司曾经想要鼓励新用户上传自己的照片做头像，可效果不佳，最后他们决定把所有没有上传照片的用户头像都改成咆哮的迪克·切尼[6]——结果照片上传的数量瞬间就飙升了！

长远来讲，有技巧地增加一点愉悦和幽默，能让用户体会到你是真地关心他们，在乎和他们的关系。

记住你的用户

在这一章里，我们讨论了很多内容，归结起来，你要记住的就是以下三个简单的概念。

营销

了解用户对你的软件的印象是怎么样的，这决定了他们是否愿意尝试你的产品。

易用性

如果你的软件不好用，速度不够快，界面不友好，访问不方便的话，用户最后一定会离你而去。

客服

主动和老用户保持联系能影响软件的演化，

决定你是不是能留住他们。

作为程序员，我们每天都有各种分散注意力的事情——代码审查，设计审查，摆弄工具，产品出问题了要去救火，还要给bug分门别类——这让我们有时候会忘了自己为什么要编写软件。不是为了自己，不是为了团队，也不是为了公司，而是为了给用户带来方便。关注用户在想什么，如何评论你的产品，以及长期使用的感受是至关重要的，用户才是你的软件成功的源泉。一分耕耘才有一分收获。

注释：[1] 译注：Perception is reality。其实这句话是故意反过来说，以引起读者注意的。英文里传统的俗语应该是Perception is deception，告诫人们不要被表象所迷惑。

注释：[2] 译注：subliminal effect指的是经过反复潜意识的刺激，让受众产生熟悉和好感等。

注释：[3] 巴里·施华兹，《选择的悖论：为什么多即是少》（Ecco）。

注释：[4] 参见亚瑟·克拉克的第三定律。译注：亚瑟·克拉克是英国著名科幻作家，他提出了克拉克基本定律。第三条是任何技术只要够先进，看起来都和魔术无异。其他两条如下。一，

如果一名资深科学家说某件事是可能的，那么他基本上不会错；如果他说某件事不可能，那么他很有可能错了。二，要知道可能和不可能之间的界线在哪里，唯一的办法就是越过那条线。

注释：[5] 要是你担心的话，那次以后那台Mac就不用再受此折磨了。

注释：[6] 迪克·切尼是小布什的副总统。

我们在这本书里讨论了很多话题，读完后或许你会觉得不知道哪些东西可以吸收到日常生活中去。可要是不能给自己的工作带来一些变化，这本书读来还有什么意义呢？接下来要怎么办？

我们先不要把事情想得太复杂。假如你要从我们的故事里学到点什么的话，只要记住HRT就好了：谦虚、尊重、信任。

我们在第一章里说道，你在社会上的一举一动和所有的人际关系都离不开这三个核心价值观。只要仔细观察就不难发现，几乎所有的人际关系问题都是由于在这些价值观上出现了缺失。

牢记HRT会影响到你的方方面面。它首先会影响到你：这些价值观会影响到你的待人接物。它还会影响你的团队：拥有谦虚、尊重、信任的文化氛围能让你的团队把大部分时间用在写代码上，而不是相互争斗上。它会影响领导团队的风格：有经验的领导会为团队服务，而不是让团队为自己服务。HRT还会影响你和团队之外的人交流合作的方式，不管对方是不是容易合作，或是臃肿瘫痪的官僚机构。最后，这些原则还会直接影响你和最重要的人群（也就是软件的用户）交流的方式。

如果能在工作中一切以HRT为导向，就可以事半功倍地发挥影响力。我们认为只有把大部分

时间花在自己喜欢做的事情上（生产优秀的软件），把处理社交摩擦、官僚主义等其他人际关系琐事的时间降到最低，才是最好的结果。

最后一点想法

现在是时候爆点料了。要是你还没发现的话，我们现在可以告诉你，这本书里大部分的内容其实和软件开发没有必然的联系。

我们的故事其实都是在讲怎么维护一个健康有效率的社区（任何社区）。你完全可以隐去我们故事中和软件开发有关的部分，将它代入任何场景中去。它可以是某个社区俱乐部、某个团契、某个联谊会，或是某个建筑团队，那里也有同样的社交问题，也可以用同样的办法去解决它们。有人的地方就有江湖，软件开发圈子里所面临的问题和其他行业没有什么不同。

因此，在努力将HRT融入程序员生活中的时候，也不要忘了它们同样适用于生活的其他部分。

谁也不知道自己的真正使命是什么，或许是在教堂里写布道词也未可知。不过现在我们还是先专注在编写软件，尽量与人合作上面吧。而且，现在你也有能力做好它们了。

这本书不光是根据我们和无数团队和人编写软件的经验写成的，我们还查阅了很多书籍和文章，来帮助自己在下笔的时候整理思路。这里的一份列表，我们或多或少地都受到了它们的影响。

- 《人件：高效项目和团队》第二版，汤姆·迪马克（DorsetHouse出版）
- 《驱动力：激励的真相》，丹尼尔·平克（Riverhead出版）
- 《你和你的研究》，理查德·海明（http://www.cs. virginia.edu/~robin/YouAndYourResearch.pdf）
- 《可预测的非理性：影响我们决定的神秘力量》，丹·艾瑞尔（HarperCollins出版）
- 《人月神话：软件工程论文集》第二版，弗利德里克·布鲁克斯（Addison-Wesley Professional出版）
- 《创业公司工程管理》，蓝俊彪（自行出版）
- 《软件开发者路线图》，大卫·胡佛和阿德威尔·奥辛奈（O’Reilly 出版）http://shop.oreilly.com/product/9780596518387.do
- 《安静：自省的力量》，苏珊·凯恩（Crown出版）

- 《勇敢改变：引入新创意的方法》，玛丽·琳·曼恩斯（Addison-Wesley出版）
- 《养蜂的艺术和冒险》，奥蒙德·艾比（Rodale Press出版）
- 《工匠和老板的时间安排》，保罗·格雷厄姆（http://www.paulgraham.com/makersschedule.html）
- 《阅读代码的艺术》，达斯汀·伯斯维尔和特雷佛·佛切尔（ O'Reilly 出 版 ）http://shop.oreilly.com/product/9780596802301.do
- 《登峰造极的秘密》，乔治·莱昂纳德（Plume出版）
- 《任务重要性的关键：工作表现效应、关系机制，以及边际条件》（2008），亚当·格兰特（应用心理学期刊 93:1， pp108-124）
- 《项目回顾：团队小结实用手册》，诺曼·柯西（Dorset House出版）
- 《幸运要素》，理查德·怀斯曼（Miramax出版）
- 《发现自我》，陈一鸣（HarperOne出版）
- 《你就是极客》，迈克· 洛浦（ O'Reilly 出版）http://shop.orelly.com/product/9780596155414.do
- 《选择的悖论：为什么多即是少》，巴里·施华兹（Ecco出版）

- 《必要一环》，艾利亚胡·金莱特（North River Press出版）
- 《三双鞋》，谢家华（Hachette Book Group出版）

图书在版编目（**CIP**）数据

极客与团队/（美）菲茨帕特里克，（美）萨斯曼著；徐旭铭译.--北京：人民邮电出版社，2013.2

ISBN　978-7-115-30844-3

Ⅰ.①极…　Ⅱ.①菲…②萨…③徐…　Ⅲ.①心里交往—通俗读物　Ⅳ.①C912.1-49

中国版本图书馆CIP数据核字（2013）第009951号

极客与团队

♦著　[美] Brian W.Fitzpatrick Ben Collins-

Sussman
译　徐旭铭
责任编辑　陈冀康
封面设计　Edie Freeclman　张健
♦人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷
♦开本：880×1230　1/32
印张：12.5
字数：154千字　　2012年2月第1版
印数：1–0000册　　2012年2月北京第1次印刷
著作权合同登记号　图字：01-2012-5791号
ISBN　978-7-115-30844-3
定价：　元
读者服务热线：**(010)67132705**　印装质量热线：**(010)67129223**
反盗版热线：**(010)67171154**
广告经营许可证：京崇工商广字第**0021**号

“幸福的味道”已提供120个不同类型的书单

1、 25岁前一定要读的25本书

2、 20世纪最优秀的100部中文小说

3、 10部豆瓣高评分的温情治愈系小说

4、 有生之年，你一定要看的25部外国纯文学名著

5、 有生之年，你一定要看的20部中国现当代名著

6、 美国亚马逊编辑推荐的一生必读书单100本

7、 30个领域30本不容错过的入门书

8、 这20本书，是各领域的巅峰之作

9、 这7本书，教你如何高效读书

10、 80万书虫力荐的“给五星都不够”的30本书

……

关注“幸福的味道”微信公众号，即可查看对应书单

如果你不知道读什么书，就关注这个微信号。