



Google程序员Steve Yegge的呐喊和吐槽
深度探讨编程语言选择、代码中的哲学
和Google工作文化

程序员的呐喊

A PROGRAMMER'S RANTINGS

[美] Steve Yegge 著 徐旭铭 译



人民邮电出版社
POSTS & TELECOM PRESS

程序员的呐喊

A PROGRAMMER'S RANTINGS



本书的作者是业界知名的程序员、来自Google的Steve Yegge。他写过很多颇有争议的文章，其中有不少就收录在这本书中。本书是他的精彩文章的合集。

针对业界的各种现象、技术、趋势等，作者都在本书中表达了自己独特犀利的观点。

本书包括以下精彩话题：

- Java真的是一门优秀的面向对象语言吗？
- 重构真的那么美好吗？
- 强弱类型语言到底哪个更好？
- 敏捷真的靠谱吗？
- 程序员要不要懂数学？
- 亚马逊做平台为什么那么成功？
- Google面试攻略。

本书讨论的都是程序员非常关注的热点话题，内容广泛，观点独到，非常适合广大程序员阅读参考。

人民邮电出版社 - 信息技术分社
<http://weibo.com/ptpitbooks>



ISBN 978-7-115-34909-5



ISBN 978-7-115-34909-5

封面设计：任文杰

分类建议：计算机 / 软件开发

人民邮电出版社网址：www.ptpress.com.cn

定价：45.00 元

程序员的呐喊

A PROGRAMMER'S RANTINGS

[美] Steve Yegge 著 徐旭铭 译



人民邮电出版社
北京

图书在版编目(CIP)数据

程序员的呐喊 / (美) 雅吉 (Yegge, S.) 著 ; 徐旭
铭译. -- 北京 : 人民邮电出版社, 2014.5
ISBN 978-7-115-34909-5

I. ①程… II. ①雅… ②徐… III. ①程序设计
IV. ①TP311.1

中国版本图书馆CIP数据核字(2014)第056155号

版权声明

Simplified Chinese translation copyright ©2014 by Posts and Telecommunications Press Published by arrangement with Hyperink.

ALL RIGHTS RESERVED

A Programmer's Rantings, by Steve Yegge

Copyright © 2013 by Hyperink

本书中文简体版由 Hyperink 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播，

版权所有，侵权必究。

◆ 著 [美] Steve Yegge
译 徐旭铭
责任编辑 陈冀康
责任印制 彭志环 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
大厂聚鑫印刷有限责任公司印刷
◆ 开本：700×1000 1/16
印张：12.75
字数：191千字 2014年5月第1版
印数：1-3500册 2014年5月河北第1次印刷
著作权合同登记号 图字：01-2013-3137号

定价：45.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书的作者是业界知名的程序员——来自 Google 的 Steve Yegge，他写过很多颇富争议的文章，其中有不少就收录在本书中。本书是他的精彩文章的合集。

本书涉及编程语言文化、代码方法学、Google 公司文化等热点话题。对 IT 界的各种现象、技术、趋势等，作者都在本书中表达了自己独特犀利的观点。比如 Java 真的是一门优秀的面向对象语言吗？重构真的那么美好吗？强弱类型语言到底哪个更好？敏捷真的靠谱吗？程序员要不要懂数学等。另外，他还谈到了很多大公司的理念，比如亚马逊做平台为什么那么成功等。最后，本书还收录了他写的 Google 面试攻略，这篇文章可以说为无数应试者点亮了明灯。

本书讨论的都是程序员非常关注的热点话题，内容广泛，观点独到，非常适合广大程序员阅读参考。

作者简介

Steve Yegge 是一名程序员，也是博主，写了很多关于编程语言、生产力和软件文化的文章。他拥有华盛顿大学计算机科学本科学位，20 年的业界经验，开发领域涉及嵌入式操作系统、可扩展的电子商务系统、移动设备应用、提升软件生产力的工具等。他曾就职于亚马逊和 Google 等公司。

译者简介

徐旭铭，从事编程十几年，翻译过几本书，现在在亚马逊当码农。工作和兴趣都是写代码，喜欢看上去很麻烦的问题。他住在西雅图，闲暇时喜欢看美剧。

前言

痛苦是本书的灵感源泉。唔，还有酒精。而当痛苦累积到一定程度的时候，我就会忍不住开始抱怨。再加上酒精作祟，什么刻薄（甚至滑稽）的话我都说得出来。现在我还会时不时地回头翻看这些东西，每次都忍俊不禁。原来我很毒舌嘛。

和绝大多数程序员不同，痛苦在我身上留下了独特的印记，迫使我的世界观发生了极大的变化。我现在能一眼看穿很多老鸟都看不到的东西，因为，老实说，他们都太墨守成规了。

过去 25 年里，经历了很多形态迥异的编程类型，写了太多原本不用写的代码。弯路走多了，反倒让我培养出一种第六感——我能看见死人。这种感觉其实很不好。如果你也这么倒霉，得到了这种第六感，那可以选择的路其实只有两条：要么心怀不满，沮丧不已；要么微笑面对。

于是我试着用乐观的态度去面对它。一开始会很难，不过现在也越来越熟练了。这当中少不了酒精的作用。当然，熟能生巧也是一方面啦。你得习惯自嘲，还要学着嘲笑别人，以及我们所生活的这个疯狂世界。更重要的是要把握好分寸，适可而止。

那就以开车来举例好了。每次看着那些显然没有从车祸中吸取教训的人们，我都会笑得很紧张，眼角不由自主地抽搐起来。我总是能注意到那些跟车跟得太紧，结果把前保险杠撞烂的家伙；还有突然急刹，结果把后保险杠撞得稀巴烂的车；车头太出路口，结果被撞得一塌糊涂的车；以及那些冒冒失失偏离自己的路线，在隔离带上擦出巨大刮痕的车。

2 前言

只要你注意到这些东西，它们可以说随处可见。很多司机似乎都不知道吸取教训。他们总是觉得自己“知道”怎么开车，之前的那次车祸根本不是自己的错！

你周围有多少成年人真的花时间去提高自己的驾驶水平？你认识的人里，是不是有些人的驾驶习惯让你难以忍受，想要帮他们指出来？只不过你知道他们肯定不爱听。

而事实上，每个人都觉得自己会开车。好像开车这件事和用微波炉加热玉米煎饼一样简单似的。任何指责他们驾驶技术不娴熟的人都是不知所谓。

其实这中间是有微妙区别的。大多数司机的问题在于他们觉得“会开车”和“好司机”是一回事。只要考到了驾照，再开上几年车，自己就算是经验丰富的司机了！该知道的自己都知道了。除了一些罕见的情况，比如雪地、沙漠、湿地、烟雾、极冷极热、强风、陡峭山地、人群，等等。那时只要开得慢一点儿不就万事大吉了嘛。而且大多时候是可以避免开车的。如果真的有必要，到时候再专门去学一下就好了。你一定是这么想的吧？

好吧……只不过书到用时方恨少，到了真的需要这些驾驶技巧的时候，通常连佛脚也没处抱了。比如车子正在打滑冲向电线杆的时候。运气好的话或许还能躲过一劫。只不过事故过后，你会想到要去接受培训，提高驾驶水平吗？当然不会啦，除非法官逼你去。毕竟你是“会开车”的人。

如果罗列一下所有的特种驾驶类型——专业赛车手、追捕车辆的警察、在森林公园防火道上的巡逻员、巨型车司机，还有很多闻所未闻的驾驶种类——只要稍微动点儿脑筋就能明白，其实你的驾驶技术根本不算是什么，充其量也就是过得去而已。既然有一半的司机低于平均线，那么很可能你也是其中一员，大家都心知肚明，只有你自己还感觉良好。

这似乎是让所谓的僵尸启示录显得合理的原因之一。因为我们都知道，没人对此有所准备。大多数司机都只是僵尸的猎物罢了，因为他们都觉得自己“会开车”，别人也不会点穿他们。

事实证明，编程和开车非常像。只要写几年代码，行了！自己就算是“会写程

序”了，好像用微波炉加热玉米煎饼一样。大多数程序员这时都会陷入舒适区，再也走不出来，就像司机会尽量避开自己不熟悉的情况一样。要是有人给他们提点建议和意见，那这个人肯定是有毛病，要么就是搞不清楚状况。

但这种狗屁不通的观点和很多父母对待游泳课的态度不是一回事吗？无数的父母不让自己的小孩学游泳，因为他们有可能会溺水？

程序员和司机一样，总是自我安慰说等到需要的时候再去学新技能也来得及。但是在内心深处他们都明白，其实当需求出现的时候就已经太晚了。因此现实情况是这样的，旱鸭子会和水保持距离，司机会绕开泥泞的路段，而程序员会躲在舒适区里，搭建围栏把自己保护起来，然后祈祷世界和平。

可能你觉得这没什么大不了的——只要每个岗位上都有专门的人才不就万事大吉了嘛。但只要稍微深入思考一下编程和开车的比喻，就会发现事情并没有那么简单。开车很复杂，也是一件很自我的事情。我们都不愿承认这一点，因此产生了很多现实生活中的问题。

比如每个地区的驾驶风格就各有不同。尽管西雅图和洛杉矶的交规基本一样，但是两地的司机完全不一样。有一次，有个土木工程师告诉我说，一个地区的车速和咄咄逼人的程度，和这个地区糟糕的交通状况历史成正比。洛杉矶的交通几十年来一直都很差，所以每个人都知道，不凶狠一点是开不了车的。而西雅图的交通变糟只是这十几年来的事情，而且还没糟到洛杉矶的那种程度，所以西雅图的司机开车都像老奶奶。

不过西雅图的司机很不擅长在雨里开车，这一点相当搞笑。就算把车速降到接近零，每年还是会发发生无数次车祸。

为什么会这样呢？这是因为西雅图的驾驶风格正处在转变之中，从休闲变得咄咄逼人。其中一部分原因是因为很多加利福尼亚居民沿着海岸线北上的同时，把原来的驾驶习惯也一起带了过来。另一部分原因是当地人口的快速增长，使得西雅图公路上的汽车数量年年翻新。所以现在西雅图的路上，既有彬彬有礼的司机，也有凶狠粗鲁的司机。

4 前言

驾驶风格发生碰撞的时候，车辆自然也会撞到一起。等到风格统一了，也就是同一个地区的人开车方式“都一样”的时候，车祸数量自然也会减少了——哪怕在外人看来，这些司机都像疯子一样。

编程和开车其实一样错综复杂。编程的世界里也有司机、技工、汽车生产商、交通工程师、地区性的交通法规、交通执法、不同地区的驾驶风格，当然还有大规模的汽车追尾事故等。编程也有自己的亚文化，若硬要把它放在一起，肯定会出现不协调。

这就是我要写这本书的原因，也是怎么写这本书的指引。我选择了一条不寻常的小路，几十年来不断探索大多数程序员从未接触过的领域，了解了人们在不同的风格和文化里是如何开发软件的。我领导过一帮拥有令人难忘的怪癖的专家，用过十几种语言，活跃于各个社区，见证了巨型项目的崛起、辉煌、衰败和消亡。虽然我不敢自称编程界的“百晓生”，但也是见过大世面的。

现在的狭隘可以说随处可见。就好像不同地区的司机，程序员也会在工业界和学术界里组成自己的小圈子，形成自己的术语、惯例、禁忌、学问等文化产物。他们能创造出自己的知识领地，就觉得自己天下第一、独一无二。

不管发生何种重大事故或交通问题，人们都会把责任怪在不合格的司机和程序员头上。虽然有时候的确是他们不好，但很多时候，冲突都是由文化差异导致的理念不同所产生的。

可惜，大多数自认为“会写程序”的程序员都会狭隘地给异己者贴上“错”的标签。这是人的本性，是我们最容易犯的错误之一，我自己也不曾幸免。

刚刚开始发牢骚的时候，我还是个丑陋的美国佬，在帖子里上蹿下跳，除了大喊“你们这帮家伙到底在搞什么”什么都不会。不过在接下来的十年里，我觉得我渐渐变成了一个业余的软件人类学家。现在我非常赞同文化相对论，尽量不去对那些和我意见不同的人下结论。

当然我不会因此就放弃开他们玩笑的机会，同样我也不介意别人来拿我寻开心。我最终希望能说服那些还在摇摆的程序员认同我对编程的看法，因为编程和开

车一样，只有在大家步调一致的时候才是最好的。所以我会继续宣扬自己的观点，也就是所谓的“软件自由主义”，它是完全合理的，甚至可能是很多软件开发都应该采用的方式。

妄图让所有人都更自由肯定是行不通的，这一点毋庸置疑。即便如此，我还是希望能帮助来自不同软件文化的人们更好地理解对方。

我会继续摇旗呐喊，因为这似乎是可以让大众听到我声音的唯一的方法。现在还有人会跟我说我的博客写得太啰嗦了，他们觉得我的观点完全可以在 100 个字里表达清楚。我发现这些评论主要是来自我的反对者，其实他们真正想要的是在反驳我的时候可以少写点字。不过一些赞同我的人也在抱怨，觉得我的博客太长，没办法抓住他们的注意力。在这里我要说，没抓住重点的人是他们，我的博客根本不长，没有足够的“分量”是无法直指人心的。经过这么多年的试错，我发现最容易抓住听众的办法还是讲故事。而不深入其中，享受过程，是讲不好故事的。

这就是本书的大致内容了。它是由一系列故事组成的。它们形式各异，可能是文章、论文、指南、抱怨，也可能是小说。但不论文体是什么，每篇都和你分享了一个故事。就算你不会同意我所有的观点，至少我希望你能喜欢我的故事，若还能让你觉得豁然开朗的话，就再幸运不过了。

Hyperink 的编辑们挑选了要收录的帖子，而且大部分章节段落的组织都是他们负责的。虽然我自己也做了一点修改，但是你现在所看到的这本书基本上是根据他们的想法组织起来的。他们干得非常漂亮。

愿你能和我一样享受这段旅程。

Steve Yegge
2012 年 8 月

目 录

| | |
|----------------------|-----|
| 前言 | 1 |
| 第 1 章 编程语言里的宗教 | 1 |
| 作者手记：巴别塔 | 1 |
| 巴别塔 | 2 |
| 作者手记：名词王国里的执行 | 17 |
| 名词王国里的执行 | 18 |
| 作者手记：神秘机器的笔记 | 28 |
| 神秘机器的笔记 | 29 |
| 作者手记：摩尔定律就是胡扯 | 50 |
| 摩尔定律就是胡扯 | 50 |
| 作者手记：变换 | 57 |
| 变换 | 58 |
| 作者手记：弱类型机制够不够强 | 65 |
| 弱类型机制够不够强 | 66 |
| 第 2 章 代码里的哲学 | 77 |
| 作者手记：软件需要哲学家 | 77 |
| 软件需要哲学家 | 78 |
| 作者手记：代码的天敌 | 85 |
| 作者手记：反对反宣传 | 98 |
| 作者手记：斑比和哥斯拉 | 103 |
| 斑比和哥斯拉 | 104 |
| 作者手记：程序员的数学 | 114 |
| 程序员的数学 | 115 |

| 2 目录

| | |
|---------------------|------------|
| 作者手记：土豪程序员的美食 | 124 |
| 土豪程序员的美食 | 124 |
| 第3章 关于Google | 139 |
| 作者手记：应聘Google | 139 |
| 应聘Google | 140 |
| 作者手记：敏捷好，敏捷坏 | 152 |
| 敏捷好，敏捷坏 | 153 |
| 作者手记：Google能保持领先吗 | 168 |
| Google能保持领先吗 | 169 |
| 作者手记：吐槽Google平台 | 175 |
| 吐槽Google平台 | 178 |
| 总结 | 189 |

第 1 章 Chapter 1

编程语言里的宗教

作者手记：巴别塔

这篇是本书最老的文章，写于 2004 年 9 月，当时我已经在亚马逊干了差不多 6 年了。当时亚马逊正饱受其庞大代码库的困扰，我曾经一度认为它的代码库规模失控是因为语言问题，后来才意识到企业文化是主因。

首当其冲的是，亚马逊的主流语言里有两门非常啰嗦的语言 C++ 和 Java，外加一门精练的语言 Perl。但是 Perl 正受到排挤，渐渐退出主流。我觉得这是因为 Perl 程序员能用更少的人力完成和 Java/C++ 程序员同样的工作量，所以要是比人多的话，他们注定是赢不了的。根据我们的估算，亚马逊的代码量比它的功能数量膨胀得更快。

第二个因素是，亚马逊的很多技术问题完全可以用自定义领域语言（DSL）的方式来解决。比如大规模的查询、分布式计算、产品配置等，他们写了太多不必要的代码了。我后来跳到 Google，发现他们为这些完全一样的问题专门编写了强大的自定义 DSL。这证实了我心中的疑虑，亚马逊的工程师在这些问题上和无头苍蝇没什么两样。我敢说这句话误伤的概率极低。

最后一点就是，和绝大多数公司一样，亚马逊非常抗拒用新语言来解决问题。他们会避免使用表达能力更强的通用语言，比如 Ruby 或 Erlang。他们也几乎从来不会想到自己去写 DSL。

结果就是，我知道他们的问题在哪里，也知道怎么解决这些问题，但是我的主管和经理们完全不买我的账。除了少数例外，大多数人都抱着非礼勿视，非礼勿听的态度：任何问题都可以用 C++ 来解决。要是你喜欢 Java 的话，也同样照此办理。其他的办法都不在考虑之列。他们连听一听的意愿都没有。

终于有一天我忍无可忍，决定彻底发泄一下对这些亚马逊同事的不满。

这大概是我第一篇认认真真写的博客。在这篇文章名为“牢骚”的文章里我承认自己有点失控，花了整整一半的篇幅来发牢骚。当时肯定没人读的啦。但是这些年来，很多人都跑来跟我说，正是因为读了这篇文章，他们才决定花毕生精力去掌握 Emacs 和 Lisp —— 其中不乏高一新生！

巴别塔

我在这篇文章里会大致谈一谈对各种语言的看法——本来是想给这个月的亚马逊开发者期刊投稿的，后来却发现改来改去都难以示人。

一方面是因为，我时不时地就用上一些粗鲁的字眼，说出一些得罪人的话来，实在是不适合在亚马逊的官方刊物上发表。所以我还是把它贴在博客上，反正也没人会看。除了你。没错，就是你。你好！

另一个原因是，我还没写完呢，通篇都是东一榔头西一棒子的片段，完全没有经过润色打磨。这也是把它放在博客里的又一个理由，不用去考虑漂亮和完整性的问题，可以想到哪儿就写到哪儿。爱看不看。

这场旋风之旅涵盖了 C、C++、Lisp、Java、Perl（这些是亚马逊在用的语言）、Ruby（因为我就喜欢这门语言），还有 Python（提到它是因为……唔，我还是先卖个关子吧）。

C

C 是必修课。为什么？因为就一切实际用途来说，这个世界上你遇到的每一台

电脑都是冯·诺伊曼结构的，而 C 以精悍的语法展现了冯·诺伊曼机的能力。

今时今日，冯·诺依曼体系结构就是计算机体系结构的标准：一个 CPU、RAM、一个磁盘、一套总线。多 CPU 其实并不影响其本质。冯诺伊曼机是一个在 20 世纪 50 年代就实现的图灵机（这是一个进行计算的抽象模型，非常有名），它很实用，性价比也很高。

其他类型的机器也是存在的。比如 Lisp 机，它实现了 20 世纪 50 年代的 Lisp。Lisp 是一种基于 lambda 演算的编程语言。而 lambda 演算则是另一种进行计算的模型。与图灵机不同的是，人也可以理解和编写 lambda 演算。不过这两种模型是等价的。它们都精确地描述了计算机的能力。

除了在跳蚤市场，Lisp 机并不常见。相比之下，冯·诺伊曼机的接受程度要高得多。此外还有诸如神经网络、细胞自动机等各种类型的计算机，只不过它们都谈不上流行，至少目前还没有。

所以你是躲不开 C 的。

还有一个原因就是，Unix 是用 C 写的。不仅如此，包括 Windows 等在内的几乎所有的操作系统都是用 C 写成的，因为它们全部属于冯·诺伊曼机操作系统。你觉得自己还有其他选择吗？至少在操作系统领域里，任何与 C 迥异的语言都发挥不出硬件的实际能力——至少这句话放在近一百年里都是对的，这些系统都诞生于这段时期内。

Lisp 是另一门必修课。倒不是说真正干活的时候要用到它，只不过要是了解一点的话，在遇到很多 GNU 的应用程序时会觉得很趁手。特别是应该学一下 Scheme，它是一种短小纯正的 Lisp 方言。对应的 GNU 版本则名叫 Guile。

麻省理工和伯克利的新生都会学一到两个学期的 Scheme，但是没人知道为什么要学这样一门怪异的语言。老实说，它其实并不适合作为入门语言，可能将它作为第二门语言也不是好选择。虽然是一定要学的语言，但是等一等也无妨。

Lisp 不好学，它的门槛很高。用 C 的思维来学习编写 Lisp 程序是不够的，而

且也没什么意义。C 和 Lisp 是两个极端，它们都擅长对方不行的事情。

假如说 C 最擅长的是映射计算机是如何工作的话，那么 Lisp 最擅长的就是映射计算是如何进行的。你真的不需要太深入 Lisp，只要掌握 Scheme 就足够了，它最简单、简洁。其他 Lisp 都已经变成了像 C++ 和 Java 那样复杂的编程环境，附带了一大堆库和工具，那些都是你不需要了解的东西。你要掌握的是用 Scheme 来写程序。如果你能做完《The Little Schemer》和《The Seasoned Schemer》后面的全部习题，那水平在我看来就已经绰绰有余了。

在选择日常的工作语言时，标准应该是它提供的库、文档、工具支持、操作系统集成、各种资源，以及很多其他和计算机怎么工作没什么关系，但是和人怎么工作大有关系的东西。

今时今日，C 的应用依然十分广泛，它是必须掌握的语言！

C++

C++非常冷漠，可以说是地球上最糟糕的语言。

它连自己是谁也不知道，完全缺乏自省的能力。好吧，C 也做不到，但是 C 不是“面向对象”语言，而让程序了解自身对面向对象来说是非常重要的。对象也是参与者。所以作为一门面向对象语言，一定要有运行时反射和获取类型的能力。而 C++不具备这种能力，好吧，应该说不是真的具备这种能力，至少你不会想要去用它。

对于 C 来说，你可以很方便地写一个 C 编译器，然后在 C 之上构建一些工具，让它具备自省的能力。可是 C++基本上是无法解析的，因此，假如你想要写个很聪明的工具，告诉你虚拟函数的签名是什么，或是帮你重构代码的话，就只能去用别人写好的工具集，因为自己解析实在太麻烦了。可是目前所有解析 C++的工具集都不好用。（作者注：现在 clang 还算不错，不过距离我写这篇东西已经 8 年多了。抗战都打完了啊！）

C++非常愚钝，用愚钝的语言是写不出聪明的系统来的。语言能塑造世界观。愚蠢的语言只能创造愚蠢的世界。

一切计算都是以抽象为基础的。高阶必须构建在低阶之上。直接在分子的层面上搭建一个城市是不现实的。采用过于低阶的抽象纯粹是自找麻烦。

麻烦来了。

C 能负担的最大项目应该是操作系统，老实说，操作系统并不是什么庞然大物。它们之所以看起来很恐怖，都是因为系统之上的应用程序，其实内核本身是很小的。

而 C++ 的最大负荷……也是操作系统。好吧，或许还能再大一点。就算 3 倍大好了，10 倍也行。操作系统内核最多有多少行代码？100 万行差不多了吧。那么 C++ 可以驾驭的系统规模大概在 1000 万行代码，超过这个数字后，系统就会开始失控，就好像《异形奇花》里的那棵植物一样。“我要吃东西……”

这还是在假设你能成功编译的情况下。

我们拥有 5000 万行 C++ 代码。不对，这个数字现在肯定更大了。我都不知道有多少了。5000 万是 9 个月之前的数字，它还在以每个季度 800 万的速度增长，而且增长的速度也在加快。天哪……

这里任何事情都进展缓慢。曾经有个亚马逊工程师把我们的代码库比作“像山一样高的排泄物，规模超过你见过的任何山脉。每当你要修复什么东西的时候，非得爬到最中间的地方才行”。

这是 4 年前的事情了，伙计们。现在他已经跳槽到更好的地方了。他是非常出色的程序员，你说可惜不可惜。

这都是 C++ 的错。别跟我吵。就是它的错。我们用的是全世界最愚蠢的语言。你不觉得这简直蠢到家了吗？

尽管如此，漂亮的 C++ 代码还是可以写出来的，这样的代码绝大部分都是 C，外加一些 C++ 特性，而且用得优雅，有节制。可惜这几乎是痴人说梦。C++ 是个巨大无比的坑，越是了解它，优越感就越强，最后一定会忍不住要用上各种特性。可是要用得好真的非常难，这门语言实在是太糟糕了。不管你有多牛，最后肯定会弄得一团糟。

你肯定觉得我在胡说八道吧？我也懒得解释。读大学的时候我曾经非常热爱 C++，那时我也只懂 C++。当我听说我的编程语言教授克雷格·钱伯斯彻底否定 C++ 的时候，我心想：“为什么会这样？我挺喜欢的啊。”STL 的发明人公开宣称他讨厌 OOP 的时候，我还以为他被黑了呢。怎么会有人讨厌 OOP？而且还是 STL 的作者？

计算机编程语言里没有所谓的“亲近生侮谩”，只有在掌握更优秀的语言的前提下，才会懂得怎么批判自己最熟悉的那门语言。

因此，要是你不喜欢我批评 C++，我建议你去了解一下更优秀的语言是什么样子的（如 Lisp），然后你才有资格否定我的话。不过到那时你就不会来否定我了。我忽悠成功了。那时你不会再喜欢 C++，可能会有点生我的气，忽悠你讨厌自己之前最爱的语言。所以你还是别管我说什么了。C++ 很出色，非常优秀。别在意我说的话。它是门很棒的语言。

Lisp

（我打赌这一节会让你大吃一惊，就算你已经了解我的风格了也不例外。）

亚马逊初创的时候，有很多了不起的工程师。虽然我不是每个人都认识，但还是熟悉其中一些的。

比如，谢尔·卡范，大牛。格雷格·林登，大牛。埃里克·本森，加入亚马逊之前就已经声名遐迩了，同样也是大牛。

Web 服务器 Obidos 就是他们写出来的。亚马逊的成功离不开 Obidos。只是后来那些垃圾工程师和 Web 开发人员，绝大多数是前端工程师，为了赶进度，为了让老板满意，不惜以最快的速度制造垃圾代码，最后把 Obidos 给搞坏了。换句话说，就是坏了一锅粥啊。但不管怎么说，Obidos 都是亚马逊创业成功的重要基石。

亚马逊的伟大元老们只用两种语言：C 和 Lisp。

奇怪吧。

显然，他们都是 Emacs 的拥趸。埃里克·本森就是 XEmacs 的作者之一^①。全世界的伟大工程师都用 Emacs。我说的是改变世界的那种工程师。不是坐在你隔壁小隔间里的漂亮女生，也不是楼下那个棒小伙弗莱德。我说的是咱们行业里最出色的软件工程师，他们改变了整个行业的面貌：詹姆斯·高斯林、高德纳、保罗·格雷厄姆^②、杰米·扎温斯基、埃里克·本森。真正的工程师只用 Emacs。要用好它是需要一点智商的，但是掌握以后它却是异常强大的工具。要是你不相信，不妨找机会去观摩一下保罗·诺德斯特龙是怎么工作的。对那些只用过.NET 系 IDE 的人来说绝对大开眼界。

Emacs 是不朽的编辑器。

对于谢尔、埃里克、格雷格，还有很多我未曾有幸与之合作的人来说，C++ 和 Perl 都是禁止出现的（因此 Java 也同属此类）。他们知道什么东西更好。

而现在，我们写的尽是 C++、Java，还有 Perl。老鸟们也早就择木而栖了。

Mailman 是谢尔用 C 写成的，然后客服部门用 Lisp 把它封装起来。没错，就是 Emacs-Lisp。除非待的时间够长，或是遇到必须帮客户解决问题的情况，否则非技术员工通常都不知道 Mailman 是什么东西。比如你写的某个垃圾功能工作不正常（因为你是用 C++ 来写的），客户很生气，你必须马上把问题解决，不然后果很严重。这时你必须直接和他们打交道，我是说那些可爱的、大字不识几个却能说会道、善意的、满怀希望的、搞不清楚状况的、乐于助人的、愤怒并快乐着的客户，他们是真正的在我们这里买东西的客户。这时你就知道 Mailman 是干吗的了。

Mailman 以前是客服部门用来处理客户 E-mail 的应用程序……有四五年了吧？反正已经很久了。它是在 Emacs 里写成的，大家都很喜欢它。

现在大家也还念着它。直到今天，我还在被迫去听非技术同事跟我长篇大论地怀念 Mailman。我可没在唬你。去年圣诞的时候我参加了一个亚马逊的聚会，我压根就不知道自己会跑到这种聚会上来，满眼望去全是商务人士，比我们这些在亚马

① 埃里克告诉我，其实他们在 Lucid 共事的时候，几乎都是杰米·扎温斯基干的活。

② 自从我写了这段话以后，好多人跑来跟我说其实保罗·格雷厄姆用的是 vi。这不科学。

逊锅炉房里上班的人靓丽多啦。有 4 个年轻妹子知道我曾经在客服部门待过后，直接把我拉到一边，聊了整整 15 分钟，说她们有多怀念 Mailman Emacs 和 Arizona（我们花了数年开发，用来替代 Mailman 的 JSP 应用）就是不好用。

当时的情况真的很诡异，我觉得她们大概是酒喝多了吧。

谢尔是天才，Emacs 也是天才之作。即便是非技术人员也喜欢 Emacs。我现在也是在用 Emacs 打字，可以的话我绝不会想在其他环境下打字。Emacs 提供了无与伦比的输入快捷方式和文本编辑功能，能大大提高工作效率。在自由输入的情况下，我能在 Emacs 里每分钟无错误地输入 130~140 个单词。这是用我写的一个打字测试的 Emacs 应用程序来计时的。但 Emacs 的能力远不止如此。

Emacs 具备了那种难以名状的特质。

可是我们却让 Mailman 退休了，因为我们具备了一种名叫愚蠢的品质。我们太差劲了。我们找不到擅长 Emacs-Lisp 的人，所以也就没法让 Mailman 继续服役。这个问题要是放到今天就好解决了。现在的亚马逊里随便一抓就是一把 Emacs-Lisp 高手，但在当时，谁会关心客服应用呢，他们也只能螺蛳壳里做道场了，而且当时也没太多人懂 Emacs-Lisp。曾经有一段时间，他们甚至把鲍勃·格里克斯坦都请来，把他关在一间小办公室里专门给 Mailman 写 Gnu Emacs 扩展。他就是那个写长颈鹿书（奥莱利出版的《Writing Gnu Emacs Extentions》）的家伙。

你知道吗，客服应用是亚马逊第一支“双比萨团队”（译者注：最佳团队）。无论是过去还是现在，他们都是完全自治的。没人搭理他们，也没人出手帮忙，一切都是他们自己搭建起来的。他们没有 Web 开发，也没有支持工程师，更没有什么 squat（一种软件质量保证流程），他们只有水平扎实的工程师，以及传帮带的传统。这就足够了。

可惜当时他们除了让 Mailman 退休外别无选择。真的很可惜。直到今天我还会听到怀念它的声音，包括在聚会上。

我觉得客服应用团队里的 Lisp 高手按人均数量来算的话，可能仍然能在亚马逊里排第一。这并不是说他们用得多，而是像埃里克·雷蒙说的那样，就算不是日

常语言，学习 Lisp 的经历也能让你受益终身。

Java

Java 可以说是过去 10 年来，计算机行业里出现的最好也是最坏的事物。

一方面，Java 把你从 C++ 里无数平凡却又容易出错的细节中解放出来。再也不用担心越界，也不再有 `core dump` 文件。抛出的异常会确切地告诉你错误在哪一行，而且 99% 的情况下都很准。对象能很聪明地按照需要把自己打印出来。诸多优点不一而足。

另一方面，作为一种语言，Java 除了拥有虚拟机、庞大的类库、安全模型和可移植的字节码格式外，还是一种信仰。所以那些太喜欢 Java 的人都不值得信任。要找到优秀的 Java 程序员是很不容易的。

但无论如何，Java 都称得上是软件工程历史上的一大进步。

从 C++ 到 Java，改变的不仅仅是语法。这种编程范式的转换可以说翻天覆地，不需要一点时间浸淫是不行的。这感觉就好像突然拥有自己的行政助理一样。你知道那些身居高位的副总裁们为什么看起来总是有那么多时间开会，对公司运营状况了如指掌，同时还能写出那么多漂亮的文件吗？他们往往会忘记自己其实是两个人：他们自己，还有他们的行政助理。行政助理让你有时间去思考真正需要解决的问题，而不是把宝贵的时间浪费在无聊的琐事上。同理，Java 把你变成了两个程序员——一个专门负责你无须关心的东西，另一个就可以专心解决实际问题。尽管这种差异非常明显，但是很快就能让人适应。

杰米·扎温斯基曾经写过一篇非常有名的文章来批判 Java 有多糟糕，但他还是这样写道：“先说好的地方：Java 没有 `free()`。我必须承认这一点，其他锦上添花而已。光这一点就足以让我忽视其他缺点了，不管它们有多糟糕。有鉴于此，本文接下来的内容都可以说无足轻重。”

杰米的这篇文章写于 1997 年，那时的 Java 还在襁褓之中，如今 Java 早已今非昔比，他当时抱怨的有些东西现在都已经修复了。

但也不是全都改好了。就语言层面，Java 仍然算不上优秀。但正如杰米所言，

它“依然是今天最好的语言，远远比我们在实际工作中用的那些彻头彻尾的垃圾语言要好得多”。

说真的，你应该读一下用 Java 写的程序。

除了作为语言本身不怎么样（杰米的不满主要就集中在这里）外，Java 在每个层面上都表现得异常出色。当然光凭这一点，可以吐槽的地方就很多了。语言本身不给力，类库再牛也是有限的。相信我：你懂的东西可能比我多很多，但是我很清楚，再好的类库也救不了一门垃圾语言。这可是在 Geoworks 的汇编地狱里摸爬滚打 5 年得来的经验。

Java 在语言层面上和 C++ 也就打个平手。唔，好吧，这句话我收回，Java 其实要好得多，至少它有字符串。连字符串都支持不好的语言哪是人用的呀。

不过 Java 也缺了一些 C++ 的优点，比如（在栈上）传引用、typedef、宏，还有重载操作符。这些东西并非必不可少，但是需要的时候就很方便。

对了，还有多重继承，说得我都开始怀念从前了。假如你要用我自己的“固执己见的精灵”来反对多态，那么我还可以举出更多为什么多重继承是必需的例子。至少也要像 Ruby 那样的 mixin 或自动委托。有时间我们可以讨论一下“火焰剑”或者“盗贼披风”的问题，你就会明白接口是多糟糕的东西了。^①

几年前，高斯林自己也承认，要是有机会重来的话，绝对不会考虑接口。

而这正是 Java 的问题所在。詹姆斯的言论让人大吃一惊。我能感觉到那种冲击力，也能猜到 Sun 的营销和法律部门肯定心急火燎地杀过去让他闭嘴，并否认他的观点。

Java 的问题就在于人们被营销攻势牵着鼻子走。C++ 和 Perl 等各种主流语言都有这个问题，当然这也是无可奈何的事情，没有造势的话，这些语言也火不起来。因此，假如语言的设计者故作天真地宣称自己的设计并不完美，你就应该给他打上

^① 译者注：“固执己见的精灵”、“火焰剑”和“盗贼披风”都是作者在其他博客中举的例子
<https://sites.google.com/site/steveyegge2/scheming-is-believing>

一针镇静剂让他闭嘴，然后取消所有的研讨大会。

炒作是不可避免的，我只是希望人们不要盲从罢了。

我自己都曾经中过 OOP 的毒，傻乎乎地为它摇旗呐喊。刚加入亚马逊的时候，我能背诵各种代替智慧和经验出现的咒语、诗篇和巫术，比如因为大家都说多重继承不好，所以它就是不好的，运算符重载也是不好的等。我隐约明白其中的道理，却又没有真正明白。那时我才渐渐意识到其实糟糕的不是多重继承，而是程序员自己。我就曾经是其中一个，当然我现在也很弱，不过每年都会变好一点就是了。

上星期面试的时候我就碰到一个候选人跟我聊为什么多重继承不好的问题，他举的例子是，如果允许多重继承，那么“人”就可以继承“头”、“手臂”、“腿”和“躯干”了。他只说对了一半。在这个例子里，多重继承当然是不好的，但要怪的是他自己。让他通过面试的话就是我傻了。

不管什么语言，不合格的程序员写出来的都是烂代码。可惜整个业界里大多数都是这样的人。

尽管多重继承的麻烦多多，而 mixin 似乎看起来很美，但其实谁也没能彻底解决问题。不过就算没有多重继承，我还是觉得 Java 要更胜 C++一筹，因为我很清楚，无论出发点有多好，我的周围肯定会出现不会写代码的家伙，而 Java 能让他们的危害小一点。

此外，Java 所拥有的并不只是语言核心本身。哪怕它演化的速度很慢，只要它还在进步，就还有希望。这才是亚马逊应该用的语言。

还有一点需要格外小心的就是，任何语言都一样，你很容易碰到那种对语言环境很熟悉，但是对品味以及计算本身等真正重要的东西却一无所知的人。

若是吃不准要找什么样的 Java 程序员，不妨考虑一下这些条件：会玩好几门语言，讨厌 J2EE 和 EJB 那类臃肿框架，还有用 Emacs 编程。这些都是不错的指标。

Perl

Perl。从哪里说起呢？

我用 Perl 好多年了。差不多是 1995 年开始写 Perl 的吧，10 年来我们一直合作愉快。

这就好像一辆骑了将近 20 万公里的老爷自行车，每每想到它的时候心里总有一种特别的感情，哪怕现在的新车只有 2.5 千克重，也不会骑得屁股疼。

Perl 的流行主要有 3 个原因。

1. Perl 代码写起来很快，能出活，这一点其实是很关键的。
2. Perl 的营销是世界一流的，专门为此写本书出来也不为过。Java 是 Sun 用钱堆出来的，而 Perl 则完全依赖拉里·沃尔和他的同僚无与伦比的营销手段，流行度却不遑多让。哈佛商学院的那些人应该好好学学 Perl 是怎么宣传的。绝对惊人。
3. 不谦虚地讲，就算到现在，它也没遇到什么真正的对手。

假如“更好”的定义是“不疯狂”的话，比 Perl 好的语言真的有好多。我随便就能举出二三十种比 Perl “更好”的语言，Lisp、Smalltalk、Python、gosh，它们都不会像夏天中国台湾省的街道上爆炸的抹香鲸一样，器官喷得到处都是，汽车、摩托、行人都无法幸免。而这正是 Perl 的写照，同时也是它真正的魅力所在。^①

不过 Perl 也有很多不可错过的特点，到今天也是独一无二的，这就足以抵消其他令人纠结的缺点了。爆炸的鲸鱼也能做出各种有用的东西来，比如香水，因此它仍然是有价值的。Perl 也一样。

其他语言（主要是 Lisp 和 Smalltalk）全都假装不存在操作系统这种东西，而列表（Lisp）和对象（Smalltalk）才是解决一切问题的银弹，Perl 的理念正好相反。

拉里如是说：“Unix 和字符串处理才是王道。”

而大多数时候，他说得一点也没错。在 Unix 集成和字符串处理方面，Perl 毫无疑问是王者。唯一的例外也只是最近才出现，这里我先卖个关子，后面再分解。

^① 译者注：这里指的是 2004 年在台南发生的抹香鲸爆炸事件，有兴趣的朋友可以自己搜索。主要是抹香鲸死后体内器官由于天热迅速腐烂，产生大量气体，搬运过程中固定用的绳索又勒得比较紧，结果体内压力太大承受不了，就在中途爆炸了。

可惜的是，拉里太过关注 Unix 集成和字符串处理，完全忘了列表和对象的存在，结果错过了正确实现它们的时机。其实他还是犯过一些错误的，特别是 Perl 早期的……唔，我不太想用“设计”这个词，不妨说“版本”好了。这些错误导致 Perl 很难正确实现列表和对象，最后（至少）在这方面，Perl 逐渐演变成一台典型的哥德堡装置。^①

拉里！列表和对象其实也是非常重要的哟！

Perl 没办法处理好列表是因为，拉里一开始就做了一个悲剧到极点的愚蠢决定，自动压扁列表。这样一来，`(1,2,(3,4))` 就会神奇地变成 `(1,2,3,4)`。而这有时并非你的本意。当时拉里肯定是遇到了某些特殊的问题，这种方式又正好适合。结果自此 Perl 的数据结构就彻底变成了爆炸鲸鱼。

时至今日，不花三分之一的时间去研究“参考书”的话，是没有办法理解以 Perl 为主题的书籍、教程，或者 ppt 的，这无疑是一种悲哀，拉里妄图以这种哥德堡式的方法来补救扁平化列表所带来的麻烦，这根本就是越帮越忙。只不过 Perl 的宣传实在是一流，硬是让你觉得参考书好像是圣经一样。什么东西都查得到！多好玩呀！捧在手上还有书香呢！

Perl 没有对象则是因为拉里从来没有真正相信过它。这倒没什么，我都不敢说自己认同它。既然如此，他干嘛还要把它加进来呢？OO 和 Perl 貌合神离，从未被社区接受过。它和字符串处理以及 Unix 集成完全不能相提并论。

当然啦，Perl 还有很多其他怪异的设计。随便举个例子，拉里设计了非常滑稽的多变量命名空间，然后通过所谓的前导符（sigil）来解析，结果弄出“上下文”这种可怕的副作用。他大概是从脚本语言那获得的灵感吧。根据当前“上下文”，Perl 语言里的每个操作符、每个函数，以及每个运算都可以有 6 种不同的行为。一个特定运算在给定上下文里的行为完全没有规则或者直觉可言。除了死记硬背，没有其他办法。

① 译注：用过于复杂的方法来解决简单问题的机械组合。

比如说？比如在标量上下文里，访问哈希会得到一个包含分数的字符串，分子是已分配的键，而分母是桶的总数。不骗你吧？真的很诡异。

不过就像前面说的一直到最近出现了能像 Perl 那样干净利落的语言。

Ruby

差不多每 15 年左右，就会出现更优秀的语言。C++取代了 C，至少对大规模应用程序开发来说是这样的，那些人追求性能却又渴望数据类型。现在 C++逐渐被 Java 所取代，而毫无疑问 Java 会在 7 年之后被更好的语言取代——我说的是它彻底干掉 C++ 的 7 年以后，显然它还没做到这一点，这主要归功于微软拖慢了 Java 占领桌面的进程。不过在服务器端，C++已经基本出局了。

Perl 的命运也差不多。因为有一种叫 Ruby 的新语言终于被翻译成英文了。没错，这是日本人的作品，这让人大跌眼镜，毕竟日本是以硬件制造而非软件开发闻名于世的。所有人都在琢磨为什么（日本不擅长软件），不过我觉得肯定和打字有关。我完全无法想象他们是怎么运用一门有超过 10 万字符的语言做到快速打字的。

不过 Emacs 在几年前终于开始支持多字节字符，所以我估计现在他们打字应该打得很快了吧。[没错，他们的确是用 Emacs——其实 Mule (Emacs 的多字节扩展) 主要就是由日本人开发的，而且完成得非常漂亮。]

总之，Ruby 对 Perl 充分实行了拿来主义。Ruby 的作者 Matz (我没记错的话，他的本名是松本行弘，不过通常都自称“Matz”) 甚至可能有点借鉴过头了，连一些不好的东西也拿了过来。好在不多，只有一点点而已。

基本上 Ruby 照搬了 Perl 的字符串处理和 Unix 集成，语法完全一样，只此一点，Perl 的精华就全都有了。这可以说是开了个去芜存菁的好头。

接着 Matz 从 Lisp 那里吸收了列表处理的精华，从 Smalltalk 那里拿来了 OO，迭代器则是取自 CLU，基本上各个语言里的优点都吸收进来了。

所有的这些东西被完美地糅合在一起，你压根注意不到斧凿的痕迹。我学 Ruby 比之前接触过的三四十种语言都快；我有 8 年的 Perl 经验，可我用了 3 天

Ruby 就觉得比 Perl 更顺手了。Ruby 具备出色的一致性，一眼就能猜到它是怎么工作的，而且基本上能猜个八九不离十。它非常漂亮，充满乐趣，同时又兼具实用性。

假如把编程语言比作自行车的话，那么 Awk 就是粉色小童车，前面有个白色的篮子，把手上还有些穗子，Perl 则是沙滩自行车（记得它们以前有多酷吗？天哪！），Ruby 则是价值 7 500 美元的钛架山地车。Perl 到 Ruby 的飞跃和 C++ 到 Java 是一样的，而且没有附带任何缺点，因为 Ruby 是 Perl 优点的超集，而 Java 去掉了一些人们想要的东西，却又没提供真正的替代方案。

后面我还会聊到 Ruby，只要有灵感。比如读一读 _why 先生的 Ruby 指南^①，那本书让我大开眼界。真的，推荐你也读一下，非常棒。虽然那种思维不是我能理解的，但是那本书实在太好玩、太犀利了，而且完全以 Ruby 为主题。好吧，也不全是，反正你读一下就明白了。

Python

说了半天，Python 这门在幕布后守候多年的龙套语言又怎么样呢？Python 社区一直以来都是避难所，收留那些吃下红色药丸，从 Perl 母体中苏醒过来的人们。^②

其实他们和 Smalltalk 程序员一样，苦等取代 C++ 的机会，结果 Java 一出，就彻底毁掉了他们所有的希望。今天的 Ruby 之于 Python，扮演的正是这样的角色，而且是一夜之间。

Python 本来是有机会一统江湖的，但是它有两个致命的缺陷：一个是空白符，另一个是死脑筋。

所谓空白符的问题就是 Python 的嵌套是通过缩进来完成的。它强迫你用特定的方式来缩进，这样大家的代码看起来就是一样的了。可惜，很多程序员都讨厌这

① 译注：_why 先生是 Ruby 社区的神秘人物，wiki 链接：http://en.wikipedia.org/wiki/Why_the_lucky_stiff。豆瓣上有一篇很详细的介绍：<http://www.douban.com/note/64005602/>。

② 译注：有关红色药丸和母体的寓意，请参考电影《黑客帝国》。

个规定，感觉好像被剥夺了自由一样；他们觉得胡乱排版和编写那种精简到一行，没人看得懂的小程序是自己的权利，而 Python 却侵犯了这一点^①。

Python 之父吉多·范罗苏姆之前也出过几次昏招——虽然不如拉里那么惊世骇俗，但也真的是够小儿科的了。比如，Python 原本是没有词法作用域的。可问题是它连动态作用域也没有，虽然说动态作用域也有自身的问题，但至少还勉强可以用。Python 最早只有全局和局部（函数）作用域，所以虽然它拥有一个“真正的”OO 系统，可是一个类却连自己的实例变量都没法访问。你只能给每个实例方法带上一个“self”参数，然后通过 self 来访问自己的实例数据。所以你在 Python 里看到的都是 self, selfself, selfselfself, selfSELFselfSELF__SELF__，哪怕你忍了空白符，这些 self 也能把你给逼疯了。

这样的问题不胜枚举。

不过在我看来，真正压死 Python 的稻草其实是它的死脑筋，这才是它成为脚本语言，乃至所有语言老大的梦想的绊脚石。你看，Python 可以说在各方面都远胜 Tcl，可就是因为这份固执，在嵌入解释器领域，大家还是选择 Tcl。

你或许要问，所谓的死脑筋到底是怎么回事？本来我有很多坏话要讲的，不过看在 Python 用起来挺顺手的份上（只要你能容忍它的缺点），我觉得还是不要对 Python 信徒太刻薄了。所谓的“死脑筋”其实也就是他们有点……脑子不转弯罢了。为什么这么说？

因为他们实在是听空白符的抱怨听怕了！

我觉得这才是 Python 一直没有 Perl 那么流行的原因，当然这也只是我个人的感觉而已。

尾声

这才是我真正想写给亚马逊开发者期刊的文章，至少是我想表达的东西。不过由于某种原因，我只有在凌晨 3 点到 6 点，因为失眠而攻击性变强的时候，才能表

^① 这里我得声明一下，我个人对这个规定没什么意见。仅仅因为这个理由而讨厌 Python 太可笑。我想说的是“其他”程序员里讨厌它的比例是很惊人的。

达出自己真正的感受。好啦，睡了！睡两个小时还要起来开会呢。

作者手记：名词王国里的执行

这是我第一篇比较出名的牢骚，所以印象很深。2005 年年中的时候，我离开亚马逊加入 Google，顺便把一些内部分享的旧文章转到了公共博客上。12 月的时候，这些文章通过 reddit 和 Hacker News 的推荐慢慢获得了一些关注——一下子我贴的每篇博客都吸引了几千人的阅读量。

与此同时，当时我在 Google 做一个很讨厌的 Java 项目。这块代码我就是看不对眼。之前做过的几个 Java 项目都还可以忍受，但这个实在是设计过头了。在这里我就不点名了。不过代码真的写得很烂，它本身几乎就是个玩笑。它采用的是那种 Java 界很流行的代码风格，追求组合性（composability）和不可变性（immutability）这两种 Java 都不怎么支持的特性。更糟糕的是，这些还是 Java 5 之前的代码。

我感觉那支团队非常爱自己的工作，因为他们不断地制造这样的代码。等读到我的解析，你就明白怎么回事了。

不管这么说，反正我是受够了，每天晚上都和我朋友托德·斯塔姆夫（Todd Stumpf）发牢骚，他也在 Google 上班。我一直想寻求某种方法来描绘这些代码，让团队能像我一样看到他们的代码有多糟糕。

终于我想到了一个办法，有一天晚上，我写了一个讽刺性的寓言故事，啰里啰唆地写了很多不写也不妨的解释。发表后一下就炸锅了，所有人都疯了。当时大多数 Java 程序员都极为震惊，好多人被冲昏了头，连生气都顾不上了。其他人则幸灾乐祸地跟着起哄，完全没意识到自己就是下一个。（我在接下来的几年里，几乎把所有语言都涮了一遍。）

反正在我写这篇东西的时候，都已经过去 6 年了，他们还是没把 lambda 加进

来。我觉得我锲而不舍的奚落多少还是有点用的，他们终于打算在 Java 8 里引入 lambda 了（假设这次不跳票——从历史经验来看并不乐观）。Java 社区非常害怕这个已经存在于其他语言里好几十年的东西，这是对“未知”事物的恐惧，而这份压力也迫使支持者们迟滞不前。

现在看起来 Java 或许终于要有动词了，和我写这篇东西之间隔了整整 7 年。

名词王国里的执行

“他们中有些家伙的脾气可大了——特别是动词，最傲气的就是他们——形容词可以随便捏，但动词就不行——不过他们都得听我调遣！不可捉摸！我就说这么多啦！”——矮胖子^①

大家好！今天要讲的是邪恶的 Java 国王和他满世界消灭动词的故事。（首先要干掉的动词就是“消灭”(to stamp out)，取而代之的是 `VerbEliminatorFactory.createVerbEliminator(currentContext).operate()`。不过现在说这个太早了，先卖个关子……）

警告：本文不是大团圆结局，也不适合心脏脆弱或喜欢无理取闹的人士阅读。若阁下火气比较大，总是忍不住在评论里挑事的话，也敬请离开。

故事开始前，首先让我们厘清一些概念。

垃圾的溢出^②

既然 Java 程序员都喜欢“用例”，我们就先给出一个吧。比如，倒垃圾。再具体一点，就是：“强尼，快去倒垃圾！都快满出来啦！”

只要是正常说英语的普通人，在被要求描述倒垃圾这个行为时，基本思路都是这样的：

① 译注：见《爱丽丝镜中奇遇记》第 6 章。

② 译注：原文是 *garbage overfloweth*，这里是故意这样翻译的，以便强调“溢出”在这里是个名词。

| | |
|--|--------------------|
| ① <i>get the garbage bag from under the sink</i> | “取出” 水槽下的垃圾袋 |
| <i>carry it out to the garage</i> | “拎出” 车库 |
| <i>dump it in the garbage can</i> | “丢进” 垃圾桶 |
| <i>walk back inside</i> | “走回来” |
| <i>wash your hands</i> | “洗手” |
| <i>plop back down on the couch</i> | “坐回” 沙发上 |
| <i>resume playing your video game (or whatever you were doing)</i> | “继续” 打电动（或者之前在做的事） |

就算你不懂英文，思考步骤应该也差不多，只不过用的是你最熟悉的语言罢了。不管用什么语言，也不管具体的步骤，只要你按部就班地去倒垃圾，最终的结果都是垃圾在外面，而你回到了屋里。

我们的脑子里充满了各种勇猛激进、热情洋溢的行为：呼吸，走路，交谈，欢笑，哭泣，希望，恐惧，吃喝，停停走走，倒垃圾。做什么和怎么表现都是我们的自由。假如我们只是太阳底下的顽石，或许生活不算糟糕，但绝不是自由的。自由来自我们自主行事的能力。

当然，我们的思想里也有名词。名词可以是吃的，从店里买的，用来坐的或睡的。“名词”会砸到你脑袋上，砸出一个大“名词”来。名词是事物，若连事物都没了，那我们如何自处？但名词也只是事物，仅此而已：它们能表示通向终点的方法，或者终点本身，或是什么珍宝，又或是周遭常见物体的名字。那是一栋楼。这里有块石头。连小孩子都能说出一大串名词。有趣的是发生在名词身上的“变化”，不是名词本身。

有行动才有变化，生命才会有滋有味。即便是滋味本身也离不开行动！毕竟不亲口“尝一尝”又怎么了解它们的味道呢。名词比比皆是，但让生活不断变化、多姿多彩的，还是动词。

① 译注：注意英文斜体和中文引号中动词的对应。

除了动词和名词，还有形容词、介词、代词、冠词、回避不了的连词、好用的感叹词等其他各种可爱的词类，我们才得以思考和表达有趣的事物。我想没人会反对各种词类都各司其职，且同等重要，一个都不能少的观点。

假如有一天突然规定再也不许用动词了，难道不会让你觉得别扭吗？

接下来我要讲的故事就发生在这样一个地方……

名词王国

从前有个 Java 国，国王 Java 实行铁腕统治，生活在那里的人们是禁止像你我这样思考的。根据国王的法令，名词在 Java 国拥有举足轻重的地位，是王国里的一等公民。他们衣着华丽，招摇过市，这些都是由形容词提供的。尽管形容词远不及名词高贵，但他们也算安分守己，毕竟比起动词来，他们要幸运多了。

因为动词在这个王国里的日子是非常艰难的。

根据 Java 国王的法令，所有在 Java 国内的动词都为名词所有。这表示他们不仅仅是宠物这么简单，动词要负责 Java 国内全部的脏活累活。事实上他们就是王国的奴隶，最多也只是农奴或者奴仆。Java 国的居民总体上还是安于现状的，很少考虑生活能发生什么变化。

动词负责了 Java 国内的全部工作，但由于没人看得起他们，动词是被禁止闲逛的。只有在有名词陪同（escort）的情况下，动词才能出现在公共场合。

当然，“陪同”本身也是一个动词，是不可以单独行动的，所以必须获取（procure）所谓的动词陪护（Verb Escorter）来协助（facilitate）陪同。那“获取”和“协助”怎么办？这时就需要另外两个相当重要的名词，协助者（Facilitator）和获取者（Procurer），分别通过协助和获取的名词形式来监护下等动词“协助”和“获取”。

国王在 Sun 大神的指引下，多次威胁要彻底将动词驱逐出 Java 王国。假如真有那么一天，那他们肯定需要至少一个动词才能行动，而我们颇具黑色幽默的国王

宣布，“执行”（execute）将是最合适的人选。

只要找到合适的执行者（Executioner）来调用 execute()，任何动词都可以被替换为动词“执行”，或是他的各种表亲“运行”（run），“开始”（start），“go”，“justDoIt”，“makeItSo”。等待（wait）可以变成 Waiter.execute()。刷牙（brush）可以变成 ToothBrusher（myTeeth）.go()。倒（take out）垃圾可以变成 TrashDisposalPlanExecutor.doIt()。没有动词可以幸免，全部会被名词取代。

在国内比较狂热的地方，名词已经彻底把动词给驱逐出去了。乍一看似乎动词依然到处都是，比如耕田、倒夜壶之类的。但只要仔细观察，就能看出端倪：名词把自己的 execute() 动词都包装成自己的样子，其实根本换汤不换药。看看 FieldTiller 的 till()，ChamberPotEmptier 的 empty()，或者 RegistrationManager 的 register()，你会发现他们其实全都隶属邪恶国王的执行者大军，只是穿着自己名词主人的马甲罢了。

邻国的动词

在其他编程语言的王国里，倒垃圾是一件简单明了的事情，和英语里的说法没什么区别。和 Java 一样，数据对象是名词，函数则是动词。（变量名都有恰当的名字，属性是形容词，操作符一般作为连词，而不定参数表示的是“你们全部”等。不过这不是故事的重点。）

和 Java 国不同，其他王国的臣民可以自由按照符合自己要求的方式组合名词和动词。

比如，在邻邦 C 国、JavaScript 国、Perl 国和 Ruby 国里，倒垃圾由一系列动作组合而成——也就是所谓的动词，或者说函数。只要对适合的对象，按照正确的顺序执行这些动作（“取”垃圾，“拎”出去，“丢进”垃圾桶等），倒垃圾的任务就算成功完成，不需要什么多余的陪护或监护。

在这些王国，基本不需要硬造名词来束缚动词。他们没有 GarbageDisposalStrategy 和 GarbageDisposalDestinationLocator 这样的名词来找到倒垃圾的地点，也不需要 PostGarbageActionCallback 这样的名词来让自己倒完垃圾后坐回沙发。他们只要用

动词去操作周围的名词，然后用一个归结性的动词 `take_out_garbage()` 就能让这些动作按照正确的顺序执行了。

这些邻邦通常会提供必要的机制，在需要的时候负责创建重要的名词。假如他们发明了一种全新的概念，比如房子、马车，或比人力快百倍的拉犁机械，他们就会为这个概念定义一个类，赋予它名字、描述、状态以及操作指令。

区别在于动词是可以独立存在的，无须发明新名词概念来存放它们。

Java 王国的人蔑视他们的邻居，这也是编程王国里的普遍现象。

若要寻根究底的话……

在世界的另一端是人烟稀少的地区，动词在那里拥有很高的地位。那些是函数式王国，有 Haskell、Ocaml、Schema 等王国。那里的居民很少踏足 Java 周边的国家。由于周围没有什么其他国家，所以函数式王国只好相互鄙视，闲着没事的时候互相打个仗。

在函数式王国，名词和动词通常同属一个阶层。不过，名词就是名词，基本上整天啥事也不干。动词已经相当活跃，包办一切了，所以他们也就没有动力去操劳。反正也没有奇怪的法律规定一定要创造辅助性名词来监护动词，所以名词和王国里事物的数量是相等的。

因此，动词在这些国家可以翻手为云覆手为雨（请原谅我的用词）。在外人看来，很容易就会得出动词（比如函数）是一等公民的印象。这或许是它们叫函数式王国，而非东西的王国的原因吧。

在最遥远的国度，比函数式王国更远的地方，有一个传说中的地带叫做 Lambda the Ultimate。相传那个地方根本没有名词，只有动词！虽然那里有“东西”，但它们皆由动词创造。若传闻不虚，那地方连睡觉前数羊用的数字也不例外，而羊是当地最流行的货币。数字 0 就是 `lambda()`，1 则是 `lambda(lambda())`，2 是 `lambda(lambda(lambda()))`，依次类推。在这个传奇国度，每样东西，不管是名词、动词，还是其他什么，都是由“lambda”这个初始动词开始构建出来的。（据说动词“lambda”的本意就是“去 lambda”。）

老实说，绝大多数 Java 国民很幸运地对世界另一边一无所知。不然你能想象得到那种文化冲击吧？他们会晕眩到必须发明新名词（比如“仇外”，“Xenophobia”）来表达自己的新情绪。

Java 国的人们快乐吗？

你可能会觉得 Java 国的生活说好听点是奇怪，说难听点就是效率低下。不过一个社会的幸福指数可以通过儿歌窥探一二，而 Java 国的儿歌确实有点古怪、诗意图。例如，Java 国的小孩常常会背诵这样一首非常有名的警世寓言：

```
For the lack of a nail,  
    throw new HorseshoeNailNotFoundException("no nails!");  
  
For the lack of horseshoe,  
    EquestrianDoctor.getLocalInstance().getHorseDispatcher().shoot();  
  
For the lack of a horse,  
    RidersGuild.getRiderNotificationSubscriberList().getBroad-  
    caster().run(  
        new BroadcasterMessage(StableFactory.getNullHorseInstance()));  
  
For the lack of a rider,  
    MessageDeliverySubsystem.getLogger().logDeliveryFailure(  
        MessageFactory.getAbstractMessageInstance(  
            new MessageMedium(MessageType.VERBAL),  
            new MessageTransport(MessageTransportType.MOUNTED RIDER),  
            new MessageSessionDestination(BattleManager.getRoutingInfo(  
                BattleLocation.NEAREST))),  
        MessageFailureReasonCode.UNKNOWN RIDER FAILURE);  
  
For the lack of a message,  
    ((BattleNotificationSender)  
        BattleResourceMediator.getMediatorInstance().getResource(  
            BattleParticipant.PROXY_PARTICIPANT,  
            BattleResource.BATTLE_NOTIFICATION_SENDER)).sendNotification(  
        ((BattleNotificationBuilder)  
            (BattleResourceMediator.getMediatorInstance().getResource(  
                BattleOrganizer.getBattleParticipant(Battle.Participant.  
                    GOOD_GUYS),  
                BattleResource.BATTLE_NOTIFICATION_BUILDER))).build-  
        Notification(  
            BattleOrganizer.getBattleState(BattleResult.BATTLE_  
                LOST)),
```



```

Result(),
PriorityMessageDispatcher.getPriorityDispatchInstance())),
waitForService();

All for the lack of a horseshoe nail.

```

时至今日，这也是金玉良言。

虽然 Java 国的版本和本·富兰克林的原版在叙述上略有不同，但他们还是觉得自己的版本别有一番风味。

其中特点之一就是凸显了“架构”。Java 国王授予了架构尊崇的地位，因为架构完全是由名词组成的。我们都知道，名词就是事物，而在 Java 国里，事物的地位远胜一切动作。建筑^①是由看得见摸得着的事物构成的，譬如高耸入云的庞然大物，又如用棍子敲打时发出低沉悦耳声音的东西。Java 国王特别喜欢这种沉闷的声音，每次换新马车的时候，他都特别喜欢从踢轮子中获得快感。不管上述的儿歌有何瑕疵，它就是不想要任何东西。

为躲避风吹雨打，人类会本能地寻找庇护所。庇护所越强大，我们就越有安全感。Java 国有很多这样坚固的东西可以满足国民安全感。他们在惊叹宏伟架构的同时会觉得这样的设计“一定是非常了不起的”。当需要修改结构时，这种感觉尤为强烈。架构上的强度过于惊人，没人相信这样的结构会倾倒。

除了坚固的架构，Java 国的一切都有条不紊：所有名词都各得其所。所有故事都是一个套路：创建对象自然是重中之重，每层抽象离不开一个管家（manager），每个管家都有一个 run() 方法。只要稍微训练一下这种风格的概念建模，Java 国的人们发现不管什么对象都可以这样表达。任何抽象和计算都可以表达出来，这就是所谓的“名词演算”。只要有足够的名词以及名词构造函数、遍历名词图表的访问方法，还有负责执行计划的 execute() 就行了。

Java 国的居民不光幸福——他们的自豪感简直是爆棚！

StateManager.getConsiderationSetter(“Noun Oriented Thinking”, State.

^① 译注：作者这里在玩文字游戏，architecture 既是架构，又是建筑。

HARMFUL).run()

按 Java 国国外的话来说就是：“面向名词的思考方式是有害的。”

面向对象编程将名词的地位捧上了天。可为什么费那么大劲把语言中的一部分请上神坛？为什么一种概念比另一种优异？其实并非 OOP 突然将我们思考方式里动词的地位降低了。它其实是一种扭曲了的观点。我的朋友雅各布·加布里尔森（Jacob Gabrielson）曾经说过，推崇面向对象编程就好像推崇面向裤子穿衣服一样。

Java 的静态类型系统和其他语言一样，有些毛病是共通的。只是过分强调面向名词的思考方式（以及建模方式）确实有点让人讨厌。任何类型系统都会要求你改变一下思维方式来适应它的系统，但是彻底消除单独的动词似乎也太过了。

C++不存在这个问题，因为 C++是 C 的超集，你可以单独定义函数。另外，C++专门提供了命名空间的抽象，而 Java 则重载了类的概念，用来表示命名空间、自定义类型、语法委托机制、一部分可见性和作用域机制，不一而足。

千万别误会——我没有说 C++“好”。但我确实更喜欢其类型系统之灵活性，至少比 Java 好一点。C++的问题在于就算是普通的句子，也会让听众忍不住想要扑上来杀人（比如，意料之外的 segfault 等各种一不小心就会踩进去的陷阱），用 C++表达某个特定想法的时候，有时找到正确方式的难度高得惊人。但是它表达思想的简洁能力却远胜 Java，因为 C++里有动词，谁会想要用没有动词的语言啊？

事实上类是 Java 里唯一的建模工具。每当新主意出现的时候，你就要通过打磨、封装、敲打的办法硬把它变成一个什么东西，哪怕它原来只是一个动作、流程，或是其他什么不是“东西”的概念。

我又想起了八九年前那些玩 Perl 的朋友跟我说的话：“老兄，不是所有的东西都是对象。”

不过奇怪的是，Java（好吧，C#也算一个，谁让它们那么像呢）似乎是主流面

向对象语言里唯一一个表现出狂热的以名词为中心行为的语言。在 Python 和 Ruby 里就几乎完全没有 `AbstractProxyMediator`、`NotificationStrategyFactory` 这种东西的存在。

那为什么 Java 里那么泛滥呢？肯定是动词上的区别造成的。Python、Ruby、JavaScript、Perl 以及所有的函数式语言都可以把函数当做单独的实体来声明和传递，而无须将它们包裹在类里。

在动态类型的语言里这当然更加容易，你只要传递函数的引用（通过函数名获取），调用者自会用正确的参数来调用函数，并正确处理返回值。

可很多静态类型语言里也都把函数作为基本对象。这其中就包含了类型啰唆的 C 和 C++，另外还有类型推导的 Haskell 和 ML。作为语言，只要提供创建、传递，按照恰当的类型签名来调用函数字面量的机制就可以了。

Java 也完全可以引入函数亦对象概念，从而变得成熟，不再扭曲，让大家可以在思维过程里使用动词。事实上有一门名叫 “The Nice programming language” 的 JVM 语言^①，它采用类似 Java 的语法，同时还提供了使用动词的机制：独立函数，Java 会迫使你用 `Callback`、`Runnable`，或其他匿名借口实现类来包裹它，以便调用。

Sun^②甚至不必打破要求所有函数都要属于类的规定，只要匿名函数携带一个隐含的 “this” 指针指向定义它的类就可以了。

我不知道为什么 Sun 坚持把 Java 圈在名词王国里。我不觉得是因为他们低估程序员能力的问题。他们连泛型都加了进来，这可是一个复杂好多倍的概念，所以他们显然也不再打算保持语言简洁。增加复杂度也不一定是坏事，毕竟 Java 现在已经稳定下来了。给程序员提供更多的工具，让他们能按照自己的方式编程才更有意义。

我真心实意地希望他们能修复这个问题，这样我就能倒完垃圾后回来继续打电

① 译注：2006 年后这门语言就不再有人维护。

② 译注：Sun 在 2010 年被 Oracle 收购。

动，或继续刚刚在做的事情了。

作者手记：神秘机器的笔记^①

这篇东西是在本书发表之前写的，所以完全可以说是为了这本书而写的。我觉得没有哪篇博客可以概括我所有的牢骚，连一部分都谈不上。老实说，对那些本就和我意见相左的人来讲，我也影响不到他们。他们很擅长当我是空气——或者说当我们不存在。

可我已经受够了被无视的感觉。

所以苦苦思考了几天后，一挥而就写了这篇东西。当然，那天写了整整 17 个小时，中间只是休息停顿了几次而已。

尽管有点言之过早，但我觉得我还是激起了一点波澜的。无数的人跑来反驳我，有人说核心观点本身就有问题，也有人质疑我界定不同观点的方法。不过我这里所展示的模型是很巧妙的，越反驳我，就越证明了它的正确性。有点像越拉越紧的哲学死结。

我觉得至少要很多年的沉淀，它才能稳定下来，为人广泛接受。但令人意外的是，才几个星期，就已经有很多人（包括原本就支持我，还有以前反对我模型的人）开始承认这个模型所描述的必然性。

它实在是太牛了，让人无法忽略。我的问题也就迎刃而解！

几天后，我又发了一篇续，主要内容是一些无关紧要的说明，所以没有收录在这里。但那篇续里提到了一个观点，让人觉得有必要在这里再强调一下。

在我创建发表这个软件工程师视角的概念模型前，被我称为“自由主义者”的那些人是没有资格参与设计讨论和实施复审的。他们往往被认为是临时手段，

① 译注：“mystery machine bus”其实是《史酷比狗》里的一台小型面包车。

或根本就懒得搭理。其实这就等于赤裸裸地无视了全世界半数的工程师——差不多是 Facebook 的人口总和，加上亚马逊和雅虎的所有前端工程师，reddit 等流行网站的工程师，以及整个 Rails 社区等。那些说自己是例外的人只是在回避现实而已。

在我看来这一切都已经成为历史了，是我将它纠正了过来。虽然花了点工夫，但还是拨乱反正了，他们终于得以登堂入室。

梦想成真了吧！

哦，最后再提一下：文章发表数周以来，我们了解到那些主张政治自由、软件保守的人士对自己被划分为“保守派”非常不满，乃至完全否定我的模型。我苦思冥想了好久，最后的结论是：爱谁谁。既然你的编程倾向保守，那就理应接受这个标签。有种就捍卫自己的信仰！

神秘机器的笔记

过去 8 年来（2004 年 6 月起至今）我一直在写各种各样的牢骚，主要是一些和软件工程有关联的问题。

之所以这样愤青是因为我真的被一些“诡异”的世界观给搞糊涂了，持这些观点的人（在我看来）差不多占了所有我遇到过的程序员的一半，包括网上碰到的和现实里认识的人。

就在上个星期，我终于想明白了这个困扰了我快 10 年的问题，现在我知道它到底是怎么回事了。

今天这篇文章就是要以全新的理念来展示软件工程。这些东西其实一点就通。等你看到的时候肯定想抽自己一嘴巴，怎么早没想到！或者抽旁边的人。要么就是你已经有那个意识了，只不过没有显而易见而已。

以我 30 多年的从业经历来看，就算这种思维不是我首创，它也从未进入过主

流。我敢这么说是因为放狗搜了十几次都一无所获。所以对这一点我相当肯定。

而我现在就要将它带入主流。看好了！

另外，接下来要传授给你的这种思维模式可能会立刻，而且永远成为你手上最有力的工具之一，特别是在和其他程序员交流，或是谈论他们的时候。

经典的 TL;DR^①

好了，不吊胃口了。首先让我阐述一下论题。我多年来从亚马逊到 Google 牢骚不断主要就是因为这个。

（注：我的话和公司无关。这一点无须赘述。假如公司需要发言人，他们自会去雇用索伦之口那种人^②，保证大家知道他们可以代表公司发言。）

我的论题如下。

1. 软件工程有自己的政治轴心，一端是保守派，另一端是自由派。（注：假如你没耐心读长篇大论的话，理论上你已经完成 90% 啦，可以不用往下读了。）
2. 这里提到的“保守派”和“自由派”概念是针对软件工程而言的。不过在一定程度上他们和现实世界里的政治还是非常相似的。

不管有没有意识到，软件业里从事和编程相关工作的每个人都能在这根轴上找到位置。

换句话说，你要么是自由派程序员，要么是保守派程序员。或许偏中庸一点，又或许极端一点，但基本上都分布在轴心两端。

和现实世界里的政治一样，软件里的保守主义和自由主义同样是迥异的世界观。这意味着他们就像针尖对麦芒一样。他们的价值观、看中的事物、核心理念，乃至动机都是完全对立的。这种价值观的冲突充斥设计阶段、是实现阶段、诊断阶

① 译注：TL;DR 指的是 Too Long; Didn't Read，太长了，懒得看。通常是回复他人又臭又长的文章时用的话。这里作者的意思是他要说一点不得不说的废话。

② 译注：《魔戒》中的人物。

段，还有恢复阶段。就好像绿鸡蛋和火腿一样。^①

我认为认识并了解我们行业中保守和自由的区别是非常重要的。或许这无助于我们达成共识，否则也不用做什么区分了。只要是本质上不可调和的观点，任何争论都可以套用这个政治轴心。程序员基本上不会（大概也不可能）改变自己的核心价值观。

不过这套政治轴心的理论让我们得以利用大家都熟悉的观点和术语来界定分歧中本质的部分。这样就能更快地解决矛盾。只要能迅速将问题归结为某个大家都明白的政治问题，那么就可以省下试图说服对方改变想法的时间，直接进入解决问题的阶段，（和政治一样）这常常意味着讨价还价和妥协让步。当然了，还有一种手段，水门，你懂的。^②

真实世界里的政治和软件行业里的政治相关吗？

政治上的保守倾向会让一个人在对待软件上也趋于保守吗？到目前为止，我觉得我们只是在依靠直觉。而任何有科学依据的结论是都离不开研究的。

不过我的直觉告诉我，即便真的存在相关性，也不会很高。我怀疑政治上的左右和软件上的左右所产生的 4 种组合在编程界里大概各占 25%，误差不超过 8%。当然这只是完全没有科学依据的猜测，要是有真实数字的话，我很乐意了解一下。

之所以认为它们没什么相关性，是因为我觉得一个人的软件政治观主要受两股力量支配。

1. 之前的编程经历——具体说就是那些让他们失败或大获成功的决策。
2. 他们的老师、教授、导师、榜样，还有同事所拥有的软件政治观。

不过有些因素的确能增加政治取向和软件取向的相关性。比如地域和区域的影响——比如，自己就读的科技型大学位于红州还是蓝州。^③另外，性格也可以让一

^① 译注：绿鸡蛋和火腿来自美国绘本作家苏斯博士的著名故事。这里意指奇怪的搭配。

^② 译注：参见水门事件。

^③ 译注：红州指倾向保守的共和党，蓝州指倾向自由的民主党。这里只是笼统地介绍红、蓝的意思。有兴趣的读者可自行查阅相关资料。

个人拥有偏向传统的自由派或保守派的价值观。

无论如何，很多程序员在发现自己政治上很保守，软件上却很自由（或者反过来）的时候肯定会大吃一惊。

不是说两党体系是有缺陷的吗？

好吧，是没错啦。硬把政治观点这种具有微妙差别的东西套到一维空间上实在是过分简化了。比如，一些美国人觉得自己在经济上很保守，而在社会问题上观点却比较自由，所以并不存在坚定支持两党之一的情况。尽管研究表明人们很容易相信极端的观点，但这种所谓的“摇摆选票”却往往成为温和派赢得大选的决定因素。

而温和派在软件工程界里的影响力也同样不可忽视，所以后面我们会继续深入探讨。

不过就算在一维空间上，仍然能从左到右得到很分散的分布。和真正的政治一样，就算是稍稍右倾的观点，对极右的人来说也可能是让人绝望的自由派。我在后面举例的时候会给出几个现实世界里有名的编程语言问题。

不管怎么说，即便有点过分简化，我仍然觉得两党模型对政治和软件的教育是个不错的起点。它建立了一个很好的框架，方便以后进一步完善模型，当然这已经超出本书的范畴了。

那么到底什么是软件自由派/保守派呢？

不妨先忽略现实世界里一些特定的问题，把注意力放在现实世界的保守派和自由派所拥有的特质以及价值观上，你会发现其实本质的东西就那么几点罢了。而这两点东西也恰恰是软件政治里的基本。

我们先从比较容易说的保守派开始，然后那些不是保守派的自然就是自由派了。之所以这么说，是因为保守派通常具有清晰统一的价值观，而自由派则比较散漫，仅仅是为了对抗保守主义才联合在一起。无论是现实世界还是软件世界里的政治都是如此。

首先我们来看一下约斯特教授等人给保守主义下的定义：^①

“我们认为政治上的保守主义是一种意识形态上的信仰，主要（但不完全）是和有意识地调节不确定性和恐惧的心理有关。”

加尼教授等人在 2008 年的研究“自由派和保守派的秘密：性格、风格，以及被他们遗弃的东西”里，探讨验证了这个理论。

我希望你也觉得这个定义没什么好争议的。毕竟“保守的”这个形容词基本上和谨慎、厌恶风险就是同义词。金融上的保守主义常常（也是显而易见的）和年龄以及财富联系在一起。公司会随着时间逐渐变得保守起来，因为它们熬过了各种法律诉讼、技术失败、公共危机、金融风暴等危机。连蚂蚁和蚱蜢的寓言故事都告诉我们寒冬将至，要储存食物。

本质上，保守主义就是风险管理。

同样，自由派的观点常常和年轻、理想主义、天真无邪联系在一起。在企业里，创业公司往往是典型的自由派——部分原因是他们本来就是为了（在一定程度上）改变世界而存在的（而自由主义原本就意味着变化），另一部分则是他们必须全力以赴完成投资人设定的目标，所以放弃一点软件安全也就变得合理（不得已）了。

自由主义并不适合作为根本的动机，不过我们不妨把它想象成一种信仰，一切都是为了改变。商业化的说法就是为了改变世界。在软件行业里，自由主义主要是为了以最快的速度开发出新功能，同时又能保证系统的灵活性，这样开发过程就不会被拖慢或是受阻。

当然啦，保守派也是如此为目标的。不过他们的做法……很保守。灵活性和生产力仍然是动机，但是不再是主要动机。安全才是最重要的考量，性能常常也是软件保守主义者眼里非常重要的东西。

① 译注：约斯特是纽约大学教授，论文地址：[http://www.psych.nyu.edu/jost/Carney,%20Jost,%20%26%20Gosling%20\(2008\)%20The%20secret%20lives%20of%20liberals%20.pdf](http://www.psych.nyu.edu/jost/Carney,%20Jost,%20%26%20Gosling%20(2008)%20The%20secret%20lives%20of%20liberals%20.pdf)

软件世界里自由派和保守派的根本分歧就在于：到底要在安全性上下多大功夫？这里说的并不仅仅是编译时的类型安全，还包括更广义的大规模系统上“防止小白误操作”的安全性。

下面我们用一些例子来说明这些本质上的分歧，保守派比自由派更看重这些理念。

- 软件在发布之前应该尽量修复所有 bug。（标语：“不要调试！”）确保为所有的类型和接口建模，编写完整的测试，系统发布之前要有明确的规范。不然的话就有你好看！
- 程序员应该回避错误。很多语言特性从本质上来说都是很容易出错和危险的，所以应该被禁止用在我们的代码里。没有这些特性我们一样可以进行开发，代码也会因此变得更安全。
- 程序员学不会新语法。公司里可以使用的语言数量应该受到限制，这样万一系统在半夜或是圣诞节挂掉的时候，值班的人就不需要去临时抱佛脚学习新语法了。另外，我们也应该禁止那些能让你定义新语法，或许改变现有语法语义的特性。（常见的例子：禁止操作符重载，禁止元编程！）
- 产品代码必须通过编译器的安全检查。不能进行静态检查的代码通常是不可接受的。就算真的有必要，也必须事先得到中央委员会的同意。（例如，eval、动态调用、RTTI）。
- 数据必须遵循事先定义好的格式。关系型数据库必须满足第三范式，必须遵循 UML 或等价的格式定义。所有的 XML 都必须有 DTD。NoSQL 数据库和键值存储（一般来说应该避免使用）必须有单独的格式定义，标明所有允许的键，以及相应的值类型。
- 公共接口必须严格建模。数据绝不可以保存在字符串或是无类型的集合里。所有的输入输出实体都必须完整显式地定义为可以静态检查的模型，最好是面向对象模型。

- 生产系统里绝不允许存在危险或有风险的后门。想要通过调试器, telnet shell, 或是任何接口连接到工作中的生产系统, 去修改运行时的代码或数据应该是不可能的。生产系统里唯一允许开发的接口只能是只读的监控频道。
- 假如一个组件的安全性存在任何疑虑, 那它就不能发布上线——不管团队怎么哀求哭嚎说没它没法继续工作。(我说的就是你, FUSE)。
- 快比慢好。没人喜欢慢的代码, 所以代码的性能一定要好。从一开始你的代码就应该奔着最优速度去写, 否则就有可能不够快。凡是那些据说比较慢的语言、DSL、库, 都应该避免使用。就算能满足现在的需要也不行, 需求(或者说调用者)总是在变化的, 可能突然软件就会变得太慢了!

这些例子只是为了说明我的意思, 实际情况要复杂得多。另外, 也不是所有自认为是保守主义者的人都认同这些例子。不过除了少数例外, 这些都是非常典型的保守派观点。

判断一种理念或技术属于自由派还是保守派, 只要去检验一系列独立的“安全性条款”即可。假如大部分都符合的话, 那么总体来说它就是倾向保守的。符合的越多就越保守。

下面列出了以上 9 个例子的自由派版本, 权作比较。假如你愿意的话, 可以把他们想象成《史酷比狗》里的角色。弗莱德会持上面的观点, 而下面的例子则属于薛吉。

- 有 Bug 没什么大不了的。反正不管多努力, bug 总是无法避免的, 而生活还是会继续下去。优秀的调试器绝对是好东西, 任何方法也不能像分步执行代码那样让你对自己的代码有深刻的认识。调试和诊断是高难度的艺术, 每个程序员都应该去掌握它们。所谓的圣诞节当机这种事情其实永远也不会发生——不然要代码冻结来干什么。**有 Bug 没什么大不了的!**(这一条大概是区分保守派和自由派理念的关键所在。)
- 菜鸟不可能永远是菜鸟。稳健的程序员生涯肯定离不开聪明地工作, 保持

学习的动力，富有创造力，知道怎么获取资源，以及拥有老道的经验。为了避免菜鸟失误（或受到伤害）就立下一堆规矩，完全是头痛医头脚痛医脚的做法。

- 在工作需要的时候，程序员的学习能力是惊人的。很多人都掌握了五线谱、盲文、手语等各种符号系统，甚至连大猩猩都能学会！所以根本没必要“保护”程序员远离新语法。只要有文档，有时间，他们自然能学会。
- **简练就是力量。**代码应短小精悍。静态检查工具无法理解你的代码不是把代码写得又臭又长的理由，而是应该（比如，通过融合运行时数据）去把检查工具做得更聪明。
- **严格的数据定义只会妨碍灵活性，延缓开发进程。**更好的策略是定义轻量级的数据定义，或者只定义一部分，甚至先略过它。因为在收集到大量数据，分析过大量用户案例之前，没人知道数据应该怎么定义。所以数据未动，代码先行才是正确的做法。
- 公共接口应该尽量简单，向前向后都兼容。建模时太过缜密的话，其实只是在猜测接口会怎么演化而已。这样反而顾此失彼，两头都无法兼顾，导致接口频繁修改，客户也不会高兴的。公共接口应该以极简为目标，能完成需求即可，增添新接口必须谨慎。
- 系统的灵活与否能决定一个客户，（或者合同）是归你还是被对手抢走。至于生产系统上的安全隐患，则可以通过日志、监控、审核等手段来缓解控制。事实证明，很多有最高权限后门和 shell 的大型系统（比如，一些数据库和网游服务器）都做到了在控制风险的同时具备世界顶尖水平的运行灵活性。
- 企业要敢于冒险，拥抱进步，警惕僵化。这和企业的规模没有关系：逆水行舟，不进则退。要保持竞争力，唯有不断有意识地去承担风险，接受随之而来的阵痛。灾难是无法避免的，所以你得做好准备，想办法从失败中再站起来。其实就算不会去冒险，这些手段仍然不可或缺。既然伸头缩头

都是一刀，那还不如冒险一试呢！

- 过早优化是一切罪恶之源。让代码先跑起来再说，正确性比性能重要，而原型迭代又比正确性更重要。只有当客户将延时列为首要解决的问题时，你才应该去考虑分析性能，进行优化。

这就是薛吉和弗莱德的不同了。

和现实生活里的政治一样，软件自由主义者在保守派眼里就是一帮邋里邋遢，缺乏纪律，天真无邪，没有原则，彻头彻尾的“坏”工程师。而保守主义者在自由派眼里则是偏执狂，是只会散播恐惧、自欺欺人的官僚主义者。

重申一下，尽管我觉得这两个阵营永远不会达成一致，但是相互理解一下对方的价值观仍然有助于双方达成妥协。

最低限度，这种保守派和自由派的分别也能帮助彼此避开对方。我觉得一支和谐的团队最好是由单一人群组成，要么全是自由派，要么全是保守派，免得双方不停地发生理念上的冲突。这就好像每个地区的驾驶风格都不同一样——霸道一点没关系，大家都一样就行了。

照你这么说，安全工程师肯定是最保守的了，是吗？

错！事实正好相反。我提到它也是想说明在看待工作内容的时候，我们的直觉有多糟糕。

安全工程师非常关心风险评估以及攻击面（attack surface）管理，所以会给人以倾向软件保守主义的印象。

然而实际上安全工程师往往非常清楚谨慎和进步之间的利弊权衡，因为他们整天都在和软件团队打交道，而软件工程师通常都忙得要死，没时间去关心安全性（更别说相关知识的匮乏了）。安全工程师在熟悉实际情况后，能迅速做出切合实际的风险管理决策，而不会为了安全而保守。

因此很多安全工程师其实都是软件自由主义者，至少也是倾向自由派的。

Google 的很多安全工程师也都是 Ruby 的粉丝，Ruby 不但本身就是一门安全

性很高的语言，同时也非常好用，表达能力很强，非常适合编写审核脚本等各种安全分析工具。我在 Google 的第一个项目就是用 Ruby 写的，通过安全认证还是很简单的。可是要得到我们那些非常保守的系统工程师的认可就真的比登天还难。

事实上几乎任何编程领域都可以分出自由和保守两个阵营。Web 程序员、数据库工程师、协议工程师、服务系统工程师等各种编程的细分子类都能分出左翼和右翼来。

我很难想象有什么领域是完全倾向自由派或是保守派的，除了网站可靠性工程，这个从本质上讲就是一个偏保守的职位外。至于其他领域，尽管乍一看似乎清一色（比如数据分析师，我一开始觉得他们肯定全是自由主义者），但后来都有接触到某个团队或细分领域是完全相反的情况。

所以我觉得这大概还是个人喜好的问题吧。肯定是的。行业领域不是决定性的因素，完全是性格使然。

有的人生来就是自由派，而有的人天生就是保守派，没什么道理可讲。

严正声明：在继续之前，我要发布一条重要的声明，我是一个死硬派软件自由主义者，甚至有点（但不完全是）自由极端分子。

这也（主要）是我会发那么多臭名昭著、争议性很强的牢骚，比如“名词王国里的执行”、“菜鸟的肖像”、“犀牛和老虎”、“代码的死敌”、“Scheme 就是信仰”、“巴别塔”、“动态语言反击了”、“变形”、“Haskell 研究员宣布了业界程序员也会关心的发现”等文章的原因。

我当然也会写其他东西。比如我正在写的“程序员的宇宙观”系列就来自不同的灵感，和软件政治学一点关系也没有。

而“通用设计模式”这篇东西则超越了软件政治学，无论是保守派还是自由派都能有效地利用其中提出的技术。另外，我也会写写电子游戏、日本动漫等其他话题。

不过我也清楚心里的自由主义对我的思维有多大的影响。就算在讨论管理的

时候，我也自认为是一个自由派的经理，而非保守派。另外，尽管我年纪不小了，生活也算富裕，但我的政治观点在面对社会问题和经济问题的时候仍然是偏自由的。

不管怎么样，我会尽量在本文里公平准确地将双方观点都表述清楚。我也只能尽力而为，你懂的。最重要的是你要自己看清楚左右的区别，同不同意我对某一边的看法倒是其次。

只要有足够多人的同意软件工程上确实存在自由派和保守派的区别，我对不同技术和理念的左右区分看起来是合理的，那这篇文章就算没白写。哪怕个别细节上有不同意见也没关系，只要大体上同意我的观点就可以了。

那么……静态类型爱好者应该算是保守主义者了？

没错，的确如此。静态类型毫无疑问是软件政治观点的分水岭，更是保守主义世界观的特征。

站在保守派的角度，静态类型（不管是显式的还是推导的）都是现代软件工程里必不可少的组成部分。这一点是不容置疑的。这里完全没有商量的余地：它是公认的工程实践（Acceptable Engineering Practice）的基石。

在自由派的眼里，静态类型就是安全闹剧（Security Theater）的同义词。它存在的唯一意义就是让人“感到”安全。人们（还有机场）已经屡次证明，就算没有它，统计数据显示的安全程度仍然是一样的。不过人们就是需要它才会觉得“够安全”。

这个问题的分歧非常大——不管你怎么看问题本身，这一点你还是能看出来的。

我之前的很多文章都谈到过这个问题，所以这里就不再重复了。要是到现在我还没能说服你，再费口舌也没什么意义。我尊重你的意见。与此同时，也希望你能多了解一点我的观点。

对了，我应该提一下一个无关的（政治立场中立的）观点，两个阵营都同意：

有静态类型的话，工具链的支持会比较好。时至今日，这还是一个无可争议的事实，而我会尽我一生的努力消灭这个差距。

过去 4 年来我一直在 Google 内部主导所谓的“格洛克计划”(Grok Project)，或许有一天它会冲破高墙，走向世界。这个项目的唯一目标就是让所有语言、客户端、构建系统以及平台都具备同等功能的工具链。

(下面的内容包含技术细节，没兴趣的话可以跳到下一节……)

这个项目的目标非常宏伟，甚至有点狂妄自大。它通过规范与语言无关，或者说跨语言定义，进而标准化工具链里几个重要的部分来完成。这些部分包含了：(1) 编译器和解释器的中间表示和元数据；(2) 编辑器客户端到服务器之间的协议；(3) 源代码的索引、分析和查询语言；(4) 在构建系统、源代码文件和代码符号等各个层面上细致的依赖规范。

好吧，这还不是全部，但大致上就是如此了。

Grok 绝不是你想象中的“小”项目。我在上面会花费相当长的时间。这 4 年里，项目已经经历了好几个重要的阶段，从“寻找风投”，到“接受”，再到“谨慎地热情”，以及“天哪，好多内部甚至外部项目现在都离不开我们了”。最近我们的团队也膨胀了一倍，从 6 名工程师增加到 12 名。每年（甚至每季度）我们都能获得更大的动力，我们的代码索引也变得越来越丰富。

Grok 不是保密项目。不过我们也还没有在公开场合下提到过它，至少现在还没有，因为我们不想太早把大家的胃口吊得太高。在此之前还有太多的工作要做，我们自己也要试用过以后才会考虑公开的事宜。

为了让这篇文章更有看点，我大胆假设，10 年之后，实现一款优秀的工具链不再仰赖静态类型。

尽管到那时类型系统的争论依然不会停歇，保守派们也可能永远无法就到底哪种类型系统最能反映程序的行为达成共识，但至少我为之贡献了自己的观点。开发工具的质量应该不再属于争论的范畴。

划分

接下来我会提及一些技术、模式、设计、规范等，将它们划分到以下 6 种类型里去：“无关政治”、“保守派”、“中间派”、“自由派”，以及中间偏左和偏右。每个类型都会提到。事先声明，这可不是什么严肃科学。这样划分只是为下面更详细的例子做铺垫。

无关政治的例子有算法、数据结构、具体数学、复杂度分析、信息理论、类型理论、计算理论等。基本上所有计算机科学的理论都属于这个类别。这些东西偶尔会在学术界激起千层浪，但就算真有什么，也只是小范围里的局部冲突而已，没什么大不了的。总的来说，这些基本的数学理论都是不朽的东西，不管是自由派还是保守派都会全盘接受它。没错，包括类型理论。

保守派的例子有久经考验的类型系统，强制性的静态类型标注，非 `public` 的符号访问修饰符（`private`、`protected`、`bewareofdog` 等）^①，完整严格的数据定义，`all-warnings-are-errors`^②，尽量采用显式操作 DOM 或是手写状态机的方式，编译依赖限制，API 被标记为过时后应该立即生效，数字类型之间禁止判断相等（也就是说没有自动类型转换），强制异常，一趟扫描的编译器，软件事务内存（Software Transactional Memory），基于类型的函数重载，显式的配置优于约定，纯函数式数据结构，任何带有“演算”（Calculus）字样的编程。

中间派（或完全中立的）的例子有单元测试，文档，Lambda，线程，Actor 模型，回调，异常，Continuation 和 CPS（Continuation-Passing Style）^③，字节编译，即时（JIT）编译，只有表达式（没有语句）的语言，多重分派（multimethod），声明式数据结构，文本化的数据结构语法，类型分派。

自由派的例子有 Eval，元编程，动态作用域，`all-errors-are-warnings`，反射以及动态调用，RTTI，C 语言的预编译器，Lisp 宏，（大部分的）领域专用语言，可选参数，可扩展语法，向下转型，自动转型，`reinterpret_cast`，自动字符串化（stringification），不同类型之间的自动转换，Nil/null 是具有语义、可重载的值（即

① 译注：`bewareofdog` 是作者胡诌的，类似家门口挂的“内有恶犬”之类的东西。

② 译注：将编译警告当成错误处理的编译选项。

③ 译注：一种函数式编程风格，常见于 Scheme 这样的语言中。

可以表示空列表、空字符串等其他不存在的值), 调试器, 位段 (bit field), 隐式转型操作符 (如 Scala 的 implicit), 60 趟扫描编译器, 导入整个名字空间, 线程局部变量, 值分派, 以参数数量区分的函数重载, 混合类型的集合, API 兼容模式, Advice (AOP 中的概念) 和 AOP (面向切面编程), 约定优于显式配置。

中间派偏保守的例子有类型建模, 关系建模, 对象建模, 接口建模, 函数式编程 (即函数没有副作用)。

中间派偏自由的例子有动态加载类和代码, 虚拟方法分派, 面向缓冲区编程。

哇, 这么分门别类实在是太好玩了! 虽然如上所列远不完整, 但是我想你应该明白我在说什么了。

自然而然的, 我们就有了下面的一些结论。

- 隐式通常都是自由派所追求的, 而显式则一般属于保守派。
- 首先考虑性能的通常都是保守派, 推迟优化的往往是自由派。
- 编译时绑定属于保守派, 运行时绑定或者推迟绑定属于自由派。
- 并发和并行看起来似乎是有争议性的话题, 但其实和阵营没什么关系。

我很想继续划分下去, 可是我们还有更高层次的东西要讨论, 不妨先到此为止吧。

一些案例和分析

下面我要举出一些例子, 你看了就会明白这种政治现象的影响有多深远。

案例一：语言

下面是一些很粗略的分类。注意, 每种语言阵营下通常还要细分出自由派和保守派。不过总体来讲, 语言的能力和擅长的方面决定了语言的选择, 因此语言特性促成了它的文化。

这份列表仅仅让你有一个具体的概念。我只罗列了通用语言、DSL 和查询语言的性质太过特殊, 往往难以分类。

难以言喻的自由：汇编语言。

极端自由：Perl、Ruby、PHP，脚本。

非常自由：JavaScript、Visual Basic、Lua。

自由：Python、Common Lisp、Smalltalk/Squeak。

温和自由：C、Objective-C、Scheme。

温和保守：C++、Java、C#、D、Go。

保守：Clojure、Erlang、Pascal。

非常保守：Scala、Ada、Ocaml、Eiffel。

极端保守：Haskell、SML。

上面的分类是根据我自己使用语言的体验，加上它们各自社区的特点而得出的。你大可有自己的感观。不过我很难想象你会把任何一门语言上移或者下移超过两个位置。

值得一提的是，静态类型，乃至强类型和一门语言是不是保守其实没什么必然联系。这一点先按下不表。

另一点值得注意的是，自由和温和的语言都很流行，而语言越保守就越不流行。

我觉得这并不难理解：自由的语言上加一点保守的成分不会很难，反过来就不行了。

例如，在 Javascript 里避免反射、eval、自动类型转换、原型继承等动态特性的代码是很容易读懂的。你完全可以用写 Pascal 的方法规规矩矩地写 Javascript，虽然它不具备静态类型标注，不过只用断言、单元测试以及一板一眼的方法来组织代码也能达到同样的效果。

可要是你想在 Haskell 代码里玩出一点动态的感觉，那要做的工作可就多得多了。Haskell 的狂热爱好者已经实现了动态代码加载等很多表面上看起来很动态的特性，但是其中所付出的努力可谓艰巨。

更有甚者，如果你用保守的方法来用自由的语言，人们最多会说：“好吧，这代码看起来真无趣，其实用一点动态特性的话，你可以少写很多代码呢。算了，反正这么写也没错。LGTM。”^①

可要是在保守的语言里玩一点野路子，就有被大家鄙视的危险了，因为……你干嘛要用这么危险的动态特性呢？我下面聊 Clojure 的时候还会进一步聊聊这种文化现象。

最后一个有趣的地方是很多非常流行的语言只是温和保守——和那些非常动态的兄弟姐妹比起来，甚至有些还自认为是相当保守的。

这里我要再次强调 C++、C#还有 Java 这样的语言之所以能在市场上取得巨大的成功，完全是因为它们知道怎么左右逢源，这一点和真正的政客是一样的。

向往自由的程序员可以把 C++当成 C 来用，而偏爱保守的程序员则可以有各种静态类型建模的手段可以用，用到什么程度完全取决于他们需要多少安全感。Java 呢？基本上也半斤八两。

同时具备“随心所欲”和“随手关门”两种思想已经被证明是获得市场认同的关键所在。营销也是一样，它的作用是要让自由和保守两大阵营都觉得它契合自己的理念。

现在市面上还有一种新型语言（如 Google 的 Dart，还有新版的 EcmaScript），专门针对中立人群（而且为了刻意讨好自由派和保守派）提供了可选的静态类型。理论上这是个不错的想法，不过操作起来我觉得还是要看营销做得到不到位。而这往往都很让人失望。

语言的设计者似乎都低估了营销的重要性！

案例二：科技公司

娱乐一下，我们来比较一下 4 家相似的科技公司的软件政治观。

(1) Facebook——诊断：极端自由。Facebook 的规模已经很大了，可他们的行为处事仍然像是一家创业公司，而且到目前为止似乎也活得挺好。他们用得主要是 C++ 和 PHP，而且很喜欢炫耀自己的代码怎么在 PHP 和 C++ 之间调来调去，而这

^① 译注：Looks good to me 的意思。常常表示 code review 通过了。

大概是最糟糕的地方了。他们的数据都放在 memcached 里：只有键值对，没有数据库结构。他们把数据导出来放到一个后台 Hive 数据仓库里，然后用 Hadoop 来进行离线数据分析。每两个星期左右他们仍然会举办通宵黑客马拉松，反正他们的程序员大多都是单身男青年（至少我上次去参观的时候还是如此），股票的估值也还很高（我上次查价格的时候好像已经没那么好了）。作为一家公司，Facebook 是非常紧密的，具有很强的执行力，十分注重程序员在网站上发布新功能的单兵能力，没有什么官僚主义。这对一家规模这么大、用户那么多的公司来讲是难能可贵的。保守派毫无疑问会厌恶蔑视他们。但是 Facebook 证明了不管具有什么世界观的程序员，只要联合起来，就能解决很多问题。

(2) Amazon.com——诊断：自由。以它的年龄、年收入、成熟的运营部门，以及财务方面的保守性来讲，这算是很令人意外的了。但事实上和早年相比，说它“自由”已经算客气的了。1998—1999 年的时候它和 Facebook 几乎一模一样，唯一的区别是当时他们用的是关系型数据库，并且事先做了大量的关系数据的建模工作。哦，除了客服软件，那里用的是键值对的存储方式，否则无法灵活应对混乱不堪的更新发布。这是我几十年来身为自由主义者接受教化的结果。不管怎么说，即便公司为了工作生活的平衡做出了很多变化（特别是股价多次下跌和好多年工程师周转率都保持在很高的两位数之后），亚马逊的工程师依然保留了自由，像创业公司那样的核心价值。每个团队都自己管理数据，自己做决策，基本上像是独立运作的商业个体。亚马逊的发布和执行速度依旧比任何人都快，因为他们真的敢冒风险（可能导致大规模瘫痪的那种风险），为了及早发布，经常发布，敢于做出取舍。亚马逊证明了自己在成立 15 年之后，仍然能保持无人能及的创新力，灵感仍在。

(3) Google——诊断：保守。Google 一开始是属于稍微有点自由的那种，然后就变得越来越保守了。Google 只有在刚刚开始的时候才是软件自由的，那时候的搜索引擎是用 Python 写的。随着公司不断壮大，他们很快就转向了软件保守主义，而这完全是由工程师自己主导的。他们写了很多宣言警告太多语言所带来的危险，而仅有的几门语言里，也有严格的风格指南，限制使用那些“危险”或者“难以阅读”的语言特性。Google 的 JavaScript 代码风格极端保守，

充斥大量静态类型标注，`eval` 更是被完全禁止。Python 的风格指南禁止元编程等各种动态特性，搞得像是没有类型的 Java 一样。他们还严格限制很多 C++ 的特性，而与此同时 C++ 11 每几周就会支持一个新特性。（C++ 11 里有超过 500 个的新特性。）内部调查显示，Google 工程师觉得妨碍特性升级和快速发布的主要障碍是官僚主义，周转率高，人事复杂。Google 曾多次努力试图削减这种官僚主义，然而他们却一次又一次地被工程师自己给抵制回来（没想到吧），因为这些人已经变成了死硬的保守派，会主动（当然更多的是被动）抵制更具灵活性的方案和技术。过去 5 年里 Google 内部的技术转向绝大多数都是保守的。对于我这样的自由派人士来说，目睹这一切实在是太让人扼腕了。好在我为自己找到了一个双方阵营都觉得有价值的位置，在我自己的团队里，还能继续保持自由的风格而不受外界干扰。

(4) 微软——诊断：难以言喻的保守。微软有两只下金蛋的鹅：`Office` 和 `Windows`。微软已经彻底退化成一个农民，只会保护它的鹅不受侵害。由于重新培训团队实在是划不来，因此它的客户根本别无选择，金蛋自然是有它的价值。可是正因为如此，微软也不再在 `Office` 或 `Windows` 上有什么创新。他们的贴牌厂商被压得利润非常薄。苹果占领了手持设备市场，而微软却在扼杀自家 `Windows Phone` 的最后一点创新，因为他们害怕这会吃掉 `Windows` 的核心业务。微软已经有 15~20 年没有在产品层面上有成功的创新了，所有的成功产品都是从竞争对手那里抄来的：`IE`、`Xbox`、`C#`、`.NET`、`Bing`、`Windows Phone` 等不胜枚举。这些都是很好的东西，可惜都是别人的创意。微软的策略是拥抱，扩展，然后利用品牌压垮竞争——至少曾经如此，直到政府在 2002 年左右出手制止为止。现在这家公司自己都不知道要干什么了，更糟糕的是，他们失去了比尔·盖茨，换了一个疯子来当家。员工不断地离开，所有人都觉得自己内心有“存在感危机”，另外，部门之间从竞争变成互害也让人无法忍受。微软已经变成了一个右翼社团主义的辛辣讽刺：坐在门廊前端着枪诅咒路人，期待向政府行贿能让他们在等死期间再多混几年补助。过去 7 年间，我私下接触过不下 400 个现任和前微软雇员。我肚子里的八卦多得是啊……或许有一天我会说出来。

(5) 奖励：苹果。诊断：未知，但是他们的营销太牛了，所以也没所谓了。不过我很有兴趣想了解它内部的软件文化。有人想爆料吗？穿马甲也行？AMA？^① 好啦，这也是个好玩的游戏。不过我们还得继续往下聊！就快完了。

特别案例：Clojure 语言

我关注 Clojure 已经好久了，至少有一年吧。但是一直都不知道怎么组织语言，把想说的东西解释清楚。

现在我终于知道这么说了！

Clojure 是一种新型的 Lisp 方言，能在 JVM 和.NET 上跑，我曾有幸为《The Joy of Clojure》^②一书作序。我曾经很兴奋地学了好几年 Clojure，刚上手的时候用起来还是很顺手的。

可后来我发现 Clojure 社区可谓极端保守，对一门 Lisp 方言来讲这有点让人出乎意料。在广泛的认知里，Lisp 一直都是自由语言家族的代表，而 Clojure 表面上似乎也是一门随心所欲的语言。它的表达能力非常强，拥有（咳咳）一堆自由度很大的新语法。和 Scheme 以及 Python 不同，它没有静态类型标注，也没有强类型建模，取而代之的是一系列很规范、易于组合的数据类型和操作。

可惜狐狸尾巴到这里就藏不住了。Clojure 的社区里全是来自纯函数界的程序员，保守得要死：基本上都是 Haskell 或者 ML 这种类型（放地图炮了），正好发现 Lisp 的 tree 语言很对胃口罢了。因此在强大的表达能力后面，Clojure 却是一个彻头彻尾的保守语言，其核心是要防范程序员犯错。

另外，这个社区非常循规蹈矩。在去年（还是前年？时光飞逝啊……）的 Clojure 大会上，就有一个重量级的发言人在那里讲述宏的害处，为什么要在现代 Clojure 代码里回避它。

我坚信只要你是懂一点 Lisp 的人，看到这里肯定会血气上涌。至少我当时的

① 译注：Ask Me Anything 的意思，常见于 reddit。

② 中文版《Clojure 编程乐趣》已由人民邮电出版社出版。

感觉是这样的。

当然从经典的软件保守主义角度来看，他的理由合情合理。有了宏就可以发明抽象出自己的领域特定语言，然后用户就必须依赖文档来理解这些抽象的作用和含义。这就意味着就算会 Clojure 也没用，你还是要依赖文档才能真正读懂别人的 Clojure 代码。那这和什么都不懂的菜鸟又有什么区别？这就让人心里打鼓了。因此保守派真正害怕的是由于自己不懂宏，突然老鸟变菜鸟。（作者注：很不巧这其实是误读了他们的立场，所以这个例子并不恰当。不过没关系，我仍然坚持自己的观点，那就是 Clojure 社区真的很保守，后来他们自己也在私底下承认了这一点。）

把个人名誉押在垃圾上面可不是件好玩的事情。现实世界里的政治保守派可是出了名的“顽固”，他们是死也不肯在自己的世界观上妥协的。因为他们把自己的名誉都押了上去。

所以虽然我挺喜欢 Clojure 的，然而内心深处坚定的自由基因最终还是让我和它分道扬镳。这对我和 Clojure 都未尝不是件好事。反正也没必要妥协嘛。

我的理念就是不要对这种事情有什么情感上的寄托。把问题归结为自由主义和保守主义之争，大家各自保留意见就行了。

这也有助于语言的设计者和社区更有针对性地推广营销。现在每门语言都宣称“大家都应该来用我”，其实我们都明白这是不可能的——就算真的是，至少我们也可以确定他们走的是中间路线，这其实是要冒风险的，冒同时被两派鄙视的风险。

而在保守派和自由派的世界里，语言设计师就可以更准确地描述自己，不用老是玩先画个饼把人忽悠进来再说的把戏。例如：“任何极端保守的程序员都应该用 Haskell！”

措辞方面我们可以再斟酌，你明白意思就好。

收官

原本我还想多举几个例子的，只不过我自己已经在喝第 3 杯（编辑注：第

4 杯) 酒了, 说不定随时会倒。

所以还是赶快收官吧!

有一点很关键, 我想提但是一直找不到机会。那就是: 要是我叫你(软件)保守派, 千万别太敏感。

我一直很担心那些政治上左倾的程序员只要一听到“保守”这个词就会立刻把它和……呃, 美国当今政坛上各种右翼保守主义所代表的东西联系起来。就是种族主义、性别歧视、宗教原教旨主义、反对同性恋、鼓吹战争, 还有猎熊那种东西^①, 你懂的。

我没说这些都是“坏的”, 至少没有在这篇文章里说。我想说的只是任何正常人都不会想和那些极翼分子扯上哪怕一点点关系。看出 3 (编辑注: 4) 杯酒能带来多少细微差别了吧? 但是我可没指责他们的观点。真没有, 至少没在这儿。我只是注意到这些观点被赋予了浓重的政治色彩, 而且最近在美国政坛上和“保守主义”这个词紧紧地联系在了一起。

所以请千万不要把现实世界里的政治观点和通常意义上的“保守”联系起来, 在这里它的意思仅仅是“厌恶风险”而已。

在编程上趋于保守是一件很平常也很正常的事, 用不着通过猎熊来证明自己。我其实还挺期待在编程界里看到人们自称自由或保守的。我的意思是, 人应该维护自己的信仰, 而且我相信我们已经在这么做了, 那么给信仰一个方便的称呼应该也不是什么难事吧。

更委婉的说法肯定会出现的, 拭目以待吧。

不管怎么说, 请告诉我你的想法! 任何观点和评论都欢迎, 包括和熊有关的!

特别感谢 Writer's Block Syrah 为这篇自断职业生涯的文章所做的贡献。

(特别给我的 Google 同事: 没错, 这篇东西的确是打算公开发表的。再次申明,

^① 译注: 有关猎熊是怎么和保守主义联系在一起的, 请搜索共和党人莎拉·佩林。

我不是索伦之口。)

作者手记：摩尔定律就是胡扯

这篇文章剖析语言演化的角度略有不同，主要关注的是并行计算，因此它有很强的个人色彩，而且这也是我最爱的话题之一。这里有太多八卦可以聊了。

我注意到这篇文章发表的时间是“名词王国里的执行”发表的前 6 天。看来那个月我的心情肯定不太好。

Google 一直在慢慢解决困扰我的问题，进展不快但一步一个脚印。C++还是用得太多，更不用说 Java 了。不过这些年来，他们越来越擅长用更多的机器和更少的代码来工作。按照这个速度，或许不远的将来真的有一天会出现一些好东西。

摩尔定律就是胡扯

经常有人会问我怎么有时间去学新东西。我在这里统一回答一下：时间是挤出来的。

这个答案没人会喜欢，但你我都知道这是真理。

我有个兄弟大卫，高中毕业以后体重暴增。他之前整天都在踢球，进了大学后就不再玩了，还兼了两份工。可惜这对减肥帮助不大，因为一份是送披萨，另一份是当服务生。所以他像吹气球一样一下儿就变成了一个大胖子，从 81 公斤涨到了 115 公斤。

有一天他瞧见一辆卡车的防撞栏上贴着一条小广告：“减肥找我！”于是他在下个红灯赶上去，里面坐着两个牛仔，他问：“要怎样才能减肥？”他们嘲笑道：“就直接减啊，你个肥猪！哈哈哈哈！”接着扬长而去。

大卫为此消沉了好一会儿，不过最终还是释怀了，因为他知道要这么做。果不

其然，两年后他就是足球冠军校队的成员了。其实压根就没有什么神奇的法术，他只不过跑去买了辆山地车，然后拼命骑，定期跑健身房，控制饮食，一年之内就减掉了 85 磅。

我自己通过节食和锻炼，也在两年前从 115 公斤减到了 66 公斤。可是随后就懈怠了（这种事情时有发生），结果现在又重了 50 磅。很讨厌，但也没办法。两个月前我终于又开始去健身房，一周 7 天，我的腿每天都酸得要死，体重还没有下降的迹象。但很快就会了，只要坚持就会有成果。

不过这些道理其实你早就懂了，是吗？

哦，对了，我写的可不是什么励志博客，那不是我会干的事儿，而且我也不是真的懂，所以没办法帮到你们。我写的东西都是牢骚，有时候还会丢几个疑问出来。我闲得无聊的时候就会干这个。

我也不知道自己为什么会写博客，这大概是强迫症吧，是我控制不了的。另外别把这些东西太当真，我的观点时不时就会变，我唯一可以肯定的就是，首先我懂的东西并不多，还有就是那个鲸鱼爆炸的新闻大概是我这辈子听过最好笑的事情了。当然你一定要我说为什么的话，我也说不出来。

今天的这篇全是牢骚，我有一种不吐不快的感觉，这样下星期我的健身搭档托德才能听我吐槽别的事情。

重大抉择

一个人的时间怎么分配是由自己决定的。

要是你不在意，那么时间就会从你身边溜走，这一点我心里很清楚。不久之前我去了一趟拉斯维加斯，途中经过一间赌场，我跑进去上厕所，洗手的时候旁边站着一个老头，也在洗手。一时兴起，我就随口问道：“嗨，老兄，你多大啦？”

他答道：“72 岁啦！我有个儿子。他出生的日子就像昨天一样！我像这样把他抱在怀里。现在嘛，他上个星期已经 40 岁了！时间过得太快啦！等你反应过来的时候，就会发现自己已经变成这样了！”

他指指自己的老脸，总结道：“哈哈！咳咳！该死，哈哈哈！”然后就出去了。我觉得他今天遇到我应该会很高兴，不过我就不敢肯定自己的心情和他一样了。

通常大学毕业（或者高中毕业）的时候，你会有两个选择：要么继续深造，要么停止学习。

决定是否继续学业其实有个简单到发指的判断标准，那就是：不劳无获。学习是一件很困难的事情。要是你觉得很简单，那你肯定是在摸鱼，不去挑战自己做一些过去做不到的新事物，你是不可能提高自己的。

或者说，这和运动是差不多的，你得把它们放在一起看。要是每天都练到腰酸背痛，那说明你真的有练到好（假设你知道区分“好的”酸痛和受伤导致的酸痛，要是你分不出来，最好还是去找专业人士问清楚）。

学习的时候也一样，研究已经知道的东西会很有吸引力，因为这样不会那么痛苦。另外，想要在任何一门领域里有所建树，都绝非一日之功。交叉训练在运动学里是有理论基础的，只打篮球是不可能成为篮球大师的，你还要训练强化自己的力量、敏捷和耐力。

交叉训练能全面提高你的身体素质，这也包括了编程。要是你写的代码都不怎么费脑子，那么就算你真的在进步，那提高的速度也会慢到无法令人注意到。只有努力（哪怕不是天天）去做一些自己不懂的东西，才会拓展自己的眼界。

待在学校里是一件非常奢侈的事情，不过真的这样的时候很少有人意识到这一点，因为学习是很痛苦的事情。但是你可以相信我：荒废时间只会让人更痛苦。

通常在学校的时候，你会时不时地奋发努力一下，一口气学到好多东西。然后学习的速度会逐渐放慢，这是无法避免的。很多和学习无关的事情会消耗掉你大部分的时间。

而毕业后所面临的选择是：你可以学一点，或不再学习。

假如你属于“不再学习”那类，我也会表示尊重，毕竟这是你自己的选择。说不定你过得比我乐呵多了。身为程序员，我非常讨厌做起事来束手束脚，而罪魁祸

首正是编程语言。没错，所有的语言都脱不了干系。编程界就像是一家巨型修车厂^①，我们用百万人/年的辛勤劳动去打造宏伟的金字塔。可这些东西根本不用那么复杂，更让我生气的是很多人安于现状，不思进取。

在我看来，主流语言的问题在于本质上它们仍然是顺序执行，属于早期的冯诺依曼模型，无法脱离单机。它们争来争去只是为了一个“最不烂”的头衔，当前领先的（不管是 Python、Ruby、Lisp，还是你的最爱）也只配得上这个称号罢了：最不烂的语言。因为它们关注的只是如何在垃圾电脑上更优雅地表达顺序计算，要不就是用线程来假装并行（假设语言支持线程的话）。

当然啦，并行语言方面也有一些有趣的尝试。比如 Erlang 就是其中比较有名的一个，说起来其实它还是挺不错的。可惜 Erlang 也不算成功，因为你都没怎么听说过它。

编程最大的问题

咱们行业的现状其实很糟糕，就是现在说话这当口。大多数硬件设计师眼里只有摩尔定律，因为赚钱全都靠它。他们想要每 18 个月就翻番。其实若能达成共识，把注意力放在并行上（大规模分布式计算），现在我们完全可以做到每 18 个月 10 倍的增长。

可惜程序员喜欢的是 XML，甚至到了病态的地步。C++也是一样。还有 Java。因为这些技术确实有用，另外就是他们不想学习，因为学习很难。没忘了这一点吧？或许你觉得我跑题了，但深入下去，其实本质是一样的。

我们还是坦率一点吧：并行语言若要摆脱冯·诺依曼图灵机的阴影，它必须与众不同。假如我们采用细胞自动机，或者任何一种无惧单点失败的并行计算模型，那就必须要有新语言，因为现有的顺序语言不是性能太差，就是难以掌控，甚至两者兼具。

细胞或者网格（或者随便什么）并行计算肯定会衍生出一整套完全不同的体系。

① 译注：意指人多，又脏又乱，每个人都有自己的车要修。

比如全新的数据结构、算法、指令集，所有的一切都要重新来过。希望你没忘记约翰·冯·诺依曼曾经是个经济学家。他的身份很多，其中一项就是经济学家，他在设计第一台电脑的时候肯定会受到影响，也就是你桌上摆的那台电脑。

冯·诺依曼创造计算机的环境和电影《阿波罗 13 号》里的情节非常相似，在电影里，休斯顿中心必须要想办法利用火箭上仅有的资源来制作二氧化碳过滤器^①，然后再把制作过程解释给太空人听，好让他们自己动手。

约翰抓来一帮工程师，其中有材料工程师、电机工程师、机械工程师，每人手里都有一些闲置物品。利用当时仅有的材料（如真空管、磁鼓、电缆、牛皮胶带），他们设计出一种勉强能用的计算机。

约翰主要关心的是这台机器中所谓的“内部经济”。访问二级存储在当时是很慢的，而访问内存就快很多，寄存器访问则非常快，依次类推。所以他设计的各种表示方法、数据结构乃至算法，都是专为这台从闲置部件中诞生的机器定制的。

假如他的机器是用巴西雨林里的行军蚁，或是机械装置，或是提线木偶制作的，那么他的数据结构和算法就会完全是另一种样子。有些东西还是共通的，比如数字、算术、函数、数据存储、排序等。但是行军蚁计算机上效率最高的东西搬到真空管计算机上就未必好用了。

你知道任何东西都可以拿来做计算机的，是吧？而且未必都要遵循图灵机模型。图灵不但是 20 世纪最伟大的天才之一，他还是第一个告诉我们拥有同等计算能力的机器可以有无数种设计的人。他自己的模型只是其中之一，而他的导师的模型（Lisp 的前身）则是另外一种。不过谁能说他们的模型就是最好的呢？

不同的计算机有自己擅长的计算类型。另外有些比较好建造，有些速度比较快，有些天生比较健壮，有些则是为并行而生。

你知道自己的大脑其实就是这样一台机器，对吧？就目前而言，它在模式匹配方面比任何电脑都要快 10 万倍，这是因为冯·诺依曼机是顺序执行的，而你的每

^① 译注：不太准确，其实是由于接口不一致所以要想办法做一个对接装置出来，有兴趣的读者可以去看看这部经典好片。好多商学院都喜欢拿它来作管理学的案例分析。

个神经元都可以独立工作。

小提示：人脑的操作系统可不是 C++写的。

那么，我们行业还有希望打破一潭死水，跳出这个让我等程序员像行军蚁一样团团转的死胡同吗？还有各种 SOAP 调用、UML 图表，这些玩意儿真的是计算机行业的要素吗？

假如真的有那么一天，上帝保佑我能活着看到，那么计算机、语言、数据结构还有算法肯定都会同时发生变化。因为为顺序型电脑而优化的语言在某些性能指标上的表现完全就是垃圾，这样的指标可不止一个。但我们也不能直接改用并行语言，因为它们在现在的电脑上根本发挥不出来：这里我要再次强调，有些“性能”的价值不是这里要讨论的内容，不妨暂且笼统地归为计算机性能，或者人的绩效好了。

身为程序员不沉迷于性能的话，还叫什么程序员呢？是不是有点讽刺？

之所以那么说的部分原因是，远比我们现在用的语言生产力更高的语言是真的存在的。可惜它们的威力大都没办法在我们的硬件上发挥出来，因为这些语言是为理论上的虚拟机而设计的，而这些机器通常又是（不正式地）由语言本身的能力“定义”的。假如无法匹配，硬件自然会拖语言的后腿。

大多数 Java 以外的 JVM 语言都有这个问题：它们需要硬件（想想蚂蚁！硬件是抽象的概念，任何东西都可以是硬件）来支持非本地跳转（long-jump）和尾递归（tail-call）优化，可是 JVM 没有在它的抽象机器定义里支持这些功能。

Lisp 也是一样，它跑的机器都不是 Lisp 机，所以压根发挥不出威力来。要是有这样的机器，我跟你打保票，C++在上面跑起来会慢到无以复加。不过可惜，程序员关心的不光是性能。他们还很不愿意学习。

这正是另一半讽刺的地方。程序员非常在意性能，他们愿意为此花费无数时间去摆弄算法和数据结构，压榨程序里的每个指令周期和字节，但却不愿意用这些时间去学习在新硬件上的新语言。哪怕这门新语言能让他在相同的时间里写出快 1000 倍的程序，或是只要千分之一的时间就能写出性能相等的程序。

因为他们就是不想努力去学任何东西。“无获则不用劳”，就是这么简单。

由此引出了本文的中心思想：摩尔定理是垃圾。别说 1000 倍了，哪怕只是追求 10 倍的生产力和计算效率，应该改变的也是我们的计算模型。除此之外，别无他法。渐进式的方法是行不通的，非得革命性的变化不可。

不过要是所有的东西都在一夕之间改变，对那些不具备学习能力的人来说岂不是个大问题？别紧张，也别给我发信，因为我现在对此表示悲观，至少在这个问题上，我很怀疑我们能摆脱泥潭。因为我们都无视了挖坑第一定律^①。

你知道吗？约翰·冯·诺依曼在生命的最后 10 年单枪匹马建立起一套基于细胞自动机的计算理论。你现在用来读我博客的计算机只不过是他该死的原型机！他原本是打算抛弃它去造一个更好的！可是后来他死于癌症，就像我的兄弟大卫，就像千千万万原本可以活得更久，做出更多贡献的人一样。我们在攻克癌症上也没什么进展，因为我们的电脑和编程语言都是可悲的垃圾。

你无法想象我编程时的痛苦。那种感觉就好像走在尖刀和碎玻璃上。你无法想象我有多看不起 C++、J2EE、你最爱的 XML 解析器，以及那些用来进行计算的可怜垃圾。未经打磨的宝石不是没有，但绝大多数都只能让我觉得恶心。

现在你开始明白我为什么喜欢和那些毕业以后仍然保持学习动力的程序员共事了吧？因为即便深陷泥潭（我们只不过是一群目不识丁的洞穴人，身处编程的石器时代），至少这些天天向上的程序员让我看到了一点希望。那种假如有更好的技术出现，他们会愿意去尝试，认真、努力去尝试的希望。我甚至希望他们会愿意和我一起来创造一些“更好的东西”。

现实很残酷，只有希望能支持我们前行。

一步一个脚印

话说回来，这些东西其实也挺有意思的。碎玻璃和尖刀也不算太糟糕，要是我生在另一个时代或地方的话，搞不好会更惨呢，这一点你我都很清楚。

① 译注：“First Law of Holes”的原文是“If you find yourself in a hole, stop digging”，引申含义是“当发现自己处于不利地位或者没有前途的时候，应该立刻转换方向，不要死撑”。

我现在工作的地方待遇相当不错，周围都是大牛，比我聪明的多了去了，大家做的事情也非常了不起。所以请不要误解：我博客里的吐槽诉苦完全都是针对我臆想中的未来，不过我觉得它成真的几率很高。等到那一天来临，我的博客里肯定会有更多关于“新”未来的抱怨。抱怨是人的本性。但我现在真的很满足。

我花很多时间在找乐子和陪家人上面，努力让自己的生活不会留下遗憾。所以我的计划中享受生活是第一位的。

如果你想要上进的话，唯一要做的就是持之以恒。不管你是想要提高自己的编程水平，还是数学水平，或是想要健身，玩风筝，甚至克服人类比害怕死亡更甚的第一恐惧：公开演讲。只要脚踏实地，就能循序渐进。

我无法许诺你能从学习中获得任何快感。你会擅长更多东西，会对很多事情有独到的见解。甚至可能会有更好的工作，或是写出让你扬名立万的软件，或是让你的工作变得更有乐趣。但是你会没时间看电视。有舍才有得嘛。我们都必须选择怎么分配自己的时间，这是零和博弈。

假如你和我一样对现状感到不满，相信我，你可以在书里找到安慰，任何书都行，数学、机器学习、编译原理或任何宣称自己在某方面有所助益的书。

当然你肯定要学着忍受学习的痛苦，每天努力一点，习惯成自然。

每个人的情况都不一样，你需要找到适合自己的方法。可能你一个星期只能抽出一个小时，但这就好像单元测试、健身或者刷牙一样，做了总比不做好。

好好享受生活吧，其他的顺其自然就好。

作者手记：变换

这篇是我为奥莱利的 Ruby 博客而写的文章，我原来是那里的客座博主，后来觉得厌烦就退出了。当然我还是写了三四篇东西的。

事实上回想起来，我退出的主要原因是稍后不久我就不怎么写 Ruby 了。不是因为我讨厌 Ruby（正相反，我非常喜欢它），而是因为它在 Google 流行不起来。所以日常工作中除了写点自己的小脚本，用到它的机会真的很少。超过这个规模就会招致怀疑的目光。

（尽管我在《巴别塔》那篇文章里说 Google 比亚马逊在语言方面更擅长，比如在适当的时候采用强大的 DSL 来解决问题，但 Google 总体上仍然很抗拒新的通用语言。现实如此，争也没用。）

这篇文章里，我会拿 Java 和 Ruby 来对峙，Java 是我原本很喜欢，后来越来越讨厌的语言，而 Ruby 在我眼里是现代语言里设计最精美的语言之一。Ruby 并非完美，是语言就有缺点，但瑕不掩瑜。在世界上所有非 Lisp 系的语言里，Ruby 应该是我的最爱。

不管怎么说，我自认为这篇东西写得相当不错。它有一种结构上的美感，我很少能写到这种境界。一般我写的东西神散形也散，可是这一篇确实脉络清晰（至少也差不太远），尽管我是打算夹带一点私货的。

可以说我成功了一半吧。好吧，Java 界根本没人注意到这篇东西。

变换

我经常听到有人说 Ruby 没有自动化的重构工具，所以根本没法用。虽然 Ruby 将来会具备一些重构功能，但是总会有一些是 Java 已经自动化，而 Ruby 没有的。他们说，这是必不可少的功能啊。

对此我表示怀疑。

到底什么是重构？字典里根本就没有这个词。

福勒告诉我们所谓重构，就是通过迭代，将恶心的代码变成优质代码的艺术和科学，是能妆点代码却不会在操作过程中产生破坏的算法，而且正确性都是能证明的。

他的剖析非常出色。他专为 Java 程序员呈现了一些很不错的技术，有些是早已为大家熟知和习以为常的，而另一些则是全新的技术。

有些“重构”技术是可以自动化的，其中很多也可以运用在 Java 之外的语言上。看起来似乎福勒和他的朋友们误打误撞，碰对了几乎和 OOP 一样大的概念。至少他们也是第一个将这个概念包装营销成功的人。无论如何，我们现在都知道这个概念了。谢谢你，福勒和你的朋友们！

《重构》是我见过的编程书里第一本讲述编程中神秘细节的著作。它将整个过程拆解开来，一步一步带着你检验代码中每一个会影响品质的小决策。这些都是原本几乎没人会在意的事情，大家都熟视无睹了，大多数人更关心的是“架构”。重构讨论的是日常工作中代码里的惯例，是真正的代码，而非纸上谈兵。

这是了不起的成就，真的，以前从来没人想到过这些，大家都觉得所谓的风格是程序员自己的选择。《重构》却在字里行间告诉我们哪里错了。很漂亮。

发现重构

我注意到《重构》这本书是在 2002 年的时候，距离它出版已经好多年了。之前一直没有读是因为出版它的是那帮搞 UML 的蠢货。我从来都不是它的粉丝。UML 在数据库建模上还（可能）有点用，但是在类建模上根本一无是处。我也从来没有在意过布池（Booch）、雅各布森（Jacobsen）等人的书。

《重构》偏偏就和这样一堆烂书挤在一起，每次看到它我都直接扫过，从不多停留一秒。别在烂书上浪费时间！

直到 2002 年冬，我在一家书店里拿起了它。没什么特别的理由，纯粹出于好奇而已。或许我在其他什么地方听到过“重构”这个词吧。它到底是什么意思？字典里根本查不到嘛。

“分解”^①倒是字典里有的词。你可以分解数字，也可以分解多项式。这个我懂，但是干嘛要再做一次呢？什么叫“再分解”？（英文中的前缀 re-有重新、再次的

① 译注：Factoring，和 Refactoring 拼写相近。

意思。)

翻开书，局部变量是万恶之源。这或许不是原话，但引入我眼帘的第一段话就在讨论这个。局部变量！？我抓来椅子一屁股坐下来，非常愤怒地往下读。我要看看这个家伙到底是脑子不正常，还是傻瓜一个。

接着一股恐惧袭上心头：他居然说得没错，有理有据。我最自豪的编程习惯（把中间值保存在局部变量里，作为简单的性能优化）显然是个坏习惯，书中明明白白地展示了一点。它解释了我代码里的某些方法为什么会不断膨胀，那就是因为这些方法无法分割，这一点我之前从未想到过。

而这些巨大的方法正是邪恶的温床。那些代码我碰都不想碰。每次不得不去修改它们的时候，黑暗洞穴就会变得愈加邪恶。不管多么不愿意，我还是必须要为之添加新功能，可是那些局部变量已经深入骨髓，像蜘蛛网一样将我困住。

书里解释了为什么它们无法分割，然后提供了斩断它们的利斧——尖锐精准的工具。这些都是很棒的技术，有些甚至还可以自动化，太了不起了。

我越读越快，完全入了迷。

这本书接着告诉我：不要写注释。又是疯话！可是它说得的确有道理。我从此也不再写单行注释，开始编写更直白的函数和参数名。

我掏钱买它回家，反复研读。彻底震惊，太天才了。时至今日我也依然这么觉得，虽然那种震撼没有当天那么巨大。但这本书实乃惊世之作，好似醍醐灌顶，令人茅塞顿开。这种事情可不常见。

突然我心里冒出一个尴尬念头：我怎么没在 1998 年的时候读这本书呢？这股寒意一下子将我淹没，好像我多年来一直都脱了裤子上班一样。其他人是不是早就读过这本书了？我会不会是唯一不知道的人？

第二天我就四处打探。我假装冷静，好像随口提起一样。你读过《重构》吧？没有。每个人都回答没有，大多数人连听都没听过。我问了 20 个程序员，只有一个读过这本书。不过他什么书都读，所以没什么好意外的。你问他的读后感，他说：

“我读过，那是本好书！”

我顿时松了一口气。既然大多数人都不知道这本书，那我的处境还算安全。我可以好好研读运用，不用担心别人发现我的代码有多烂。它们只是在某几个方面比较糟糕而已，大部分还是经过精心设计的。各种常见的软件工程规范、设计模式、单元测试、版本控制等我都有用到。它只是在有些地方不太漂亮而已，而我现在已经知道怎么改正了。

重构的现状

今天，所有人都知道《重构》这本书，因为很多 IDE 包含了书中描述的全部自动化重构技术，甚至还有自己的扩展。

尽管一夜爆红，我却怀疑到底有多少程序员真的读过福勒的这本书，恐怕连部分章节都没读过吧。我觉得大多数程序员到今天也不知道，其实还有很多重构技术是无法自动化的，就连 Java 也无能为力。甚至可以说大部分都是如此。当然这说起来就话长了。

我到今天也不知道他们到底为什么要起“重构”这个名字。我猜可能是比较吸引眼球吧。站在数学的角度上来看，这个词和“分解”勉强有点联系，而重组（reorganization）则太宽泛了，所以“重构”似乎是个好名字。

而有时候能不能取个好名字就能决定一个想法会不会为大众接受。

到今天，重构已经衍生出一整个产业了，它就是一面旗帜。Java IDE 的粉丝们都在为之摇旗呐喊。重构工具就好像摆在瓶子里的产品一样。翻开菜单从里面挑一个出来，连地球都能撬起来。

为什么自动化重构在 Java 阵营里那么流行，在其他语言里却没掀起什么浪花呢？

Java 说那是因为只有 Java 才能将代码变换自动化做到这种程度。

重奏

我经常听到有人说 Ruby 没有自动化的重构工具，所以根本没法用。虽然 Ruby 将来会具备一些重构功能，但是总会有一些是 Java 已经自动化，而 Ruby 没有的。

他们说，这是必不可少的功能啊。

对此我表示怀疑。

现在我已经读过福勒的书了，他说的东西我也懂了。这本书的确是艺术和科学的完美结合，它教你如何通过可以证明的步骤，将糟糕的代码变成优质代码。

但它教给我们的应该不止这些，不是吗？

当然，你得先读一下这本书才知道那是什么。那么，你到底读了没有？从头到尾一字不漏？得了吧，别不承认，你肯定跳读了。

所以事情其实是这样的：要了解代码从烂到好的过程，首先要知道什么叫烂代码。书中首先给出了一些例子，然后解释了为什么这些代码很烂。福勒列出了一些值得警惕的地方，甚至还给它们起了个名字：“代码异味”。从营销的角度来讲是不是很高明？也许吧。不过他们的观点还是正确的。

那么这些代码一开始是怎么变烂的呢？

首先当然是由于过早优化造成的，为了避免重复计算而保存了太多的中间变量。因为害怕方法调用会造成虚幻的负担，而刻意回避编写短小的函数。我们还弄出一大堆类的继承关系，仅仅是为了想象中可能存在的复用，为了避免分配容器对象而弄出一个巨大的参数列表。滥用 null，把它当成具有语义的符号。放任简单的布尔逻辑表达式变成错综复杂、无法阅读的浆糊。不用访问方法来封装数据和数据结构。还有其他很多乱七八糟的问题。

正是因为各种各样的小错误累积起来造成了代码品质下降。这本书将它们分门别类，加以命名，并归类成严重错误。

如此一来，我们是不是就不会再犯错了呢？大概读过这本书的人就不会了吧。亡羊补牢，读晚了总比没读过好。至少读过这本书后，你会了解烂代码是什么样子的，也会知道它是怎么沦落到那个样子的，你也会学到避免它的办法。

那么自动化重构工具是什么时候开始受到关注的？这本书一开始关心的是设计，以及修复代码品质的工具。然而现在的焦点却落到了修复上，特别是修复技术

里可以自动化的那部分上面。

其实，这其中隐含的意思就是默认烂代码是无法避免的，就算我们了解了它的方方面面，就算我们能一眼发现它也没用。假如你好好读过这本书，就会发现它其实不只是一份列出 100 多种重构技术的列表，事实上它为我们呈现的是一种思维方式。只要领会了其中的精神，你完全可以发明自己的重构技术，发现新的代码异味。

现在你知道怎么一次就把代码写对了吧。

什么？你不同意？因为代码是活的，需求会不断变化？没错，代码的确需要改。但是重构并非修改代码的全部，它其实只是其中一小部分罢了。此外还有数据建模、架构、设计模式等其他高阶的玩意儿。这还不包括那些敝帚自珍的代码技巧，这些模式很趁手，但是不够通用，没有响亮的名字。修改它们的技巧可不是能在 IDE 的重构目录下找到的选项。这一点我还是可以肯定的。

重构是微观的。你关心的这个类怎么写，那个方法怎么实现，细节到局部变量，控制流等微设计的决策。

你现在知道的是怎么在那个层面上避免犯错。

当然这是在你读过书，而且是仔细读过以后。另外就是你得有足够的经验才能有被当头棒喝的感觉，否则这些经验教训是无法深入骨髓的。

《重构》让我们变得懒惰了吗？可能吧，特别是假如我们只是快速浏览了一下，只看了讲解怎么做，略过为什么的时候。这会让我们觉得它就是教你怎么将代码从无法避免的腐烂中救回来的办法。就算你真的读了，那些鼓吹自动化重构的言论也会让你忘记这本书真正的主旨。那就是改善你的代码，然后再也不再写这么菜鸟的代码。

那这就是全部了？不是。有时候是不是还是需要重构？没错。有没有什么我还不了解的微妙之处？当然有，单纯地讨论重构太虚了。它和其他现代发展理念是互通的，比如“不要重复自己”，“一次且仅一次”，单元测试等。到时候我还会再提到重构。

不过我今天感兴趣的只是为什么 Java 程序员认为点几个按钮就能“编程”的功能那么重要，重要到其他语言都不愿考虑。即使这门语言在世界范围内也在迅速得到认可，生产力的提升相较于 Java 对 C++ 只会多不会少，而且还没有 Perl 和 Python 那些前辈的缺点和难用的地方。

我的意思是，这些人根本连试一下 Ruby 都不愿意？天哪，点按钮一定……非常好玩。

我只得闭上眼睛，努力去想象那种力量……

点按钮的生产力

啊哈，自动化重构。只要点点鼠标就能看到成果；点选菜单就能编程；选择自动攻击，就能撬动地球；不费吹灰之力就能移山。是不是感觉自己像超人？编程似乎简单得像玩电子游戏一样。

这种感觉就像操作巨型机械：约翰迪尔、小松公司、卡特彼勒——那些拥有世界上最大轮胎的黄色机械怪物，我们小时候每次看到都会惊叹不已。司机只要一拉操纵杆，就能铲起一大堆土。原本需要上百人劳动一整天的工作，只在弹指之间就完成了。一台完全在你控制之下的黄色巨兽。

这才叫生产力。假如我的工作是搬运泥土，那我表示同意：没有巨型机械、拖拉机、土方车、大吊车或者挖土机的话，工作起来是完全没有效率可言的。这是我重构地球表面的私人工具。我是绝对不会让你染指这些几百吨的庞然大物的。

卡特彼勒（毛毛虫），对机械巨人来说这真是个奇怪的名字。是不是？其实这个名字来自一种小型的肢节小虫。这种小虫每一节都差不多，每段都有两条一样的小腿，所有的腿以波浪的方式推动小虫向前。你看看，光是在泥泞中爬行就需要进行那么多计算！

好了，现在我懂了。毛毛虫就好像长长的类似机器的昆虫，而巨型机械就是巨大的类似昆虫的机器。开始明白其中的联系了吧。

自动化代码重构工具很适合对付卡特彼勒那样的代码。你面对的是一大堆实

体——对象、方法、名称，以及任何有迹可循的东西。它们几乎一模一样。你要以一致的方式修改它们，就像毛毛虫爬行一样，同时移动所有的腿。

那我们的代码是怎么变成那样的呢？因为写得烂。这时重构就能救命。再优秀的设计也会出纰漏，但我们仍然可以补救，反正有自动化的奴仆来帮我们修复这些小问题。它们不知疲倦，我们只要点个按钮就行了。

既然如此，谁能离得开自动化重构工具？还有谁能协调 Java 那些数以百计的小腿，让它们像毛毛虫一样统一行动呢？

让我来告诉你答案：Ruby 是蝴蝶。^①

作者手记：弱类型机制够不够强

这是本书最老的文章之一。我在亚马逊时代写了差不多 50 篇博客，其中只有两篇被收录到本书中，这是一篇，另一篇是《巴别塔》。

这篇博客写于我离职亚马逊的前一个月。我在尽量保证公平公正的情况下，准确地总结了亚马逊在核心理念上的不同。我观察了这些理念上的差异在团队里的 Perl 和 Java 阵营之间，不同的数据建模风格，以及其他很多地方所产生的冲突。

当时我关注的焦点主要是和（编程语言的）类型安全以及（关系型数据库设计的）结构安全。后来我将这个问题进一步提炼，又包含进来了很多其他两极分化很严重的设计问题——我最近的另一篇博客“神秘机器的笔记”就是我在不同的团队和整个公司中，经过多年观察很多项目后，对这种分歧的一个总结。

我在亚马逊的时候还不知道哪一种“更好”。现在我觉得从本质上讲，其实哪个都称不上更好，只能说萝卜青菜各有所爱罢了。

尽管这篇博客和“神秘机器的笔记”有点重复，它还是有点干货的。它详细描

① 译注：意指 Ruby 是完全不同的物种，Java 中自动化重构工具所要解决的问题在 Ruby 中根本不存在。

述了我亲身观察的一个项目的第一手资料，Perl 和 Java 程序员在这个项目上合作了一年多，就为了解决同一个问题。而那帮大多是自学成才的 Perl 程序员，可以说完爆 Java 程序员。

那么，请欣赏。

弱类型机制够不够强

你觉得……一个动态类型系统能有多大？静态类型系统到底重不重要？我非常想知道答案。

亚马逊有很多很大的系统，绝大多数采用了静态强类型，至少我知道的情况如此。那么有没有这样一种需求，或者说我们能不能用 Perl、Ruby、Lisp、Smalltalk 完成同样的工作？

其实，我感兴趣的是更宽泛一点的话题，也就是编程语言里的类型系统。例如，强类型之于关系数据建模和编程语言是一样的。你可以为每种可能性建模，也可以用键值对的方法快速建立原型。XML 数据建模也一样。

静态类型的优点

下面列出了静态类型的主要优点。

- (1) 静态类型可以在程序运行之前，依赖其与生俱来的限制来及早发现一些类型错误。（或是在插入/更新记录，解析 XML 文档等情况下进行检测。）
- (2) 静态类型有更多机会（或者说更容易）优化性能。

例如，只要数据模型完整丰富，那么实现智能化的数据库索引就会更容易一些。编译器在拥有更精确的变量和表达式类型信息的情况下，可以做出更优的决策。

- (3) 在 C++ 和 Java 这样拥有复杂类型系统的语言里，你可以直接通过查看代码来确定变量、表达式、操作符和函数的静态类型。

这种优势或许在 ML 和 Haskell 这样的类型推导语言里并不明显，他们显然认为到哪里都要带着类型标签是缺点。不过你还是可以在有助阅读理解的情况下标明类型——而这些在绝大多数动态语言里是根本做不到的。

(4) 静态类型标注可以简化特定类型的代码自动化处理。

比如说自动化文档生成、语法高亮和对齐、依赖分析、风格检查等各种“让代码去解读代码”的工作。换句话说，静态类型标签让那些类似编译器的工具更容易施展拳脚：词法工具会有更多明确的语法元素，语义分析时也比较少要用猜的。

(5) 只要看到 API 或是数据库结构（而不用去看代码实现或数据库表）就能大致把握到它的结构和用法。

还有其他要补充的吗？

静态类型的缺点

静态类型的缺点如下。

(1) 它们人为地限制了你的表达能力。

比如，Java 的类型系统里没有操作符重载、多重继承、mix-in、引用参数、函数也不是一等公民。原本利用这些技术可以做出很自然的设计，现在却不得不去迁就 Java 的类型系统。

无论是 Ada 还是 C++，或是 OCaml 等任何一种静态类型系统都有这样的问题。差不多半数的设计模式（不光是 GoF 的那些）都是扭曲原本自然直观的设计，好将它们塞进某种静态类型系统：这根本就是方枘圆凿嘛。

(2) 它们会拖慢开发进度。

事先要创建很多静态模型（自顶向下的设计），然后还要依据需求变化不断修改。这些类型标注还会让源代码规模膨胀，导致代码难以理解，维护成本上升。（这个问题只在 Java 里比较严重，因为它不支持给类型取别名。）还有就是我上面已经提到过的，你得花更多的时间来调整设计，以适应静态类型系统。

(3) 学习曲线比较陡。

动态类型语言比较好学。静态类型系统则相对挑剔，你必须花很多时间去学习它们建模的方式，外加静态类型的语法规则。

另外，静态类型错误（也可以叫编译器错误）对于初学者来说很难懂，因为那时程序根本还没跑起来呢。你连用 `printf` 来调试的机会都没有，只能撞大运似的调整代码，祈求能让编译器满意。

因此学习 C++ 比 C 和 Smalltalk 难，OCaml 比 Lisp 难，Nice 语言比 Java 难。而 Perl 所具备的一系列静态复杂性——各种诡异的规则，怎么用，什么时候用等——让它的难度比 Ruby 和 Python 都要高。我从来没见过有哪门静态类型语言是很好学的。

(4) 它们会带来虚幻的安全感。

静态类型系统确实能减少运行时的错误，提升数据的完整性，所以很容易误导人们觉得只要能通过编译让程序跑起来，那它基本上就没什么 bug 了。人们在用强静态类型系统的语言写程序时似乎很少依赖单元测试，当然这也可能只是我的想象罢了。

(5) 它们会导致文档质量下滑。

很多人觉得自动生成的 `javadoc` 就足够了，哪怕不注释代码也没关系。SourceForge 上充斥着这样的项目，甚至连 Sun JDK 也常常有这个问题。（比如，Sun 很多时候都没有给 `static final` 常量添加 `javadoc` 注释。）

(6) 很难用它们写出兼具高度动态和反射特点的系统。

绝大多数静态类型语言（大概）都出于追求性能的目的，在运行时丢弃了几乎所有编译器生成的元数据。可是这样一来，这些系统通常也就很难在运行时作出修改（甚至连内省都做不到）。比如，若要想给模块加一个新函数，或是在类里加一个方法，除了重新编译，关闭程序然后重启之外别无他法。

受此影响的不单是开发流程，整个设计理念也难逃波及。你可能需要搭建一个复杂的架构来支持动态功能，而这些东西会无可避免地和你的业务代码混在一起。

我还漏掉了其他什么缺点吗？

只要把上面的列表对调一下，你基本上就可以列出动态类型语言的优缺点了。动态语言的表达能力更强，设计灵活度也更大；易学易用，开发速度快；通常运行时的灵活性也更高。相对地，动态语言无法及时给出类型错误（至少编译器做不到），性能调优的难度也比较高，很难做自动化静态分析，另外，变量和表达式的类型在代码里很不直观，没办法一眼看出来。

静态语言最终会向用户屈服，开始添加一些动态特性，而动态语言常常也会尝试引入一下可选的静态类型系统（或是静态分析工具），此外它们还会设法改善性能，增加错误检测，以便及早发现问题。很遗憾，除非一开始设计语言的时候就考虑到可选的静态类型，否则强扭的瓜怎么也不会甜的。

到底正确的方法是什么？

强弱类型之争真的会让人肾上腺素飙升。选择任何一种都会影响到项目周期、架构以及开发实践。

假设你（暂时）只能选一个的话，下面哪个更符合贵公司的实际情况？

1. 总的来说，稳定性更重要。公司规模巨大，业务的复杂度也很高，所以只有为代码和数据建立严格的模型才能拨乱反正。若不能在一开始就正确地建好模型和架构，将来必定被反咬一口，所以最好在前期设计阶段多下点工夫。健壮的接口肯定是少不了的——这基本上就是静态类型的意思了，否则用户怎么知道如何使用它们。另外还必须追求性能最优，这也离不开静态类型和细致的数据模型。我们在业务上最重要的优势是系统和接口能保持稳定、可靠、预期可控以及性能卓越。可选技术有 SOAP（或者 CORBA）、UML 和严格的 ERD，所有 XML 都要有的 DTD 或者 schema 定义，以及 C++、Java、C#、Ocaml、Haskell 和 Ada。

2. 总的来说，灵活性更重要。我们的业务需求会不断产生不可预期的变化，死板的数据模型几乎不可能合理地预见这些变化。小团队需要迅速完成自己的目标，同时还要适应业务的快速变化。因此我们需要灵活性大、表达能力强的语言和数据模型，哪怕它增加了所需的性能成本也在所不惜。可靠性可以通过严格的单元

测试以及敏捷开发实践来保证。我们在业务上最大的优势就是可以快速发布新特性。可选技术有 XML/RPC 和 HTTP，必不可少的敏捷编程，XML 和关系数据都采用不那么严格的键值对模型，以及 Python、Ruby、Lisp、Smalltalk 和 Erlang。

下面我会以稍微有点戏谑的方式解释这两种理念的工作流程，尽可能将它们的本质区别展现出来。

强类型阵营基本是这样工作的：首先是按照当前的需求进行设计；制定出文档，哪怕只是初稿也没关系；然后定义接口和数据模型。假设系统要承受巨大流量，因此每个地方都要考虑性能。避免采用垃圾收集和正则表达式这类抽象。（注意：即便是 Java 程序员，通常也会努力避免触发垃圾收集，他们总是在开始写程序之前讨论对象池的问题。）

他们只有在无计可施的情况下才会考虑动态类型。例如，一支采用 CORBA 的团队只有在极端情况下才会在每个接口调用上添加一个 XML 字符串参数，这样他们就能绕开当初选择的死板的类型系统了。

第二个阵营基本是这样工作的：先搭建原型。只要你写代码的速度比写同等详细程度的文档快，你就可以更早地从用户那里获得反馈。按照当下的需求定义合理的接口和数据模型，但是别在上面浪费太多时间。一切以能跑起来为准，怎么方便怎么来。假设自己肯定要面对大量的需求变化，所以每个地方首先考虑的是尽快让系统运行起来。能用抽象的地方就尽量用（比如每次都去收集数据而先不考虑缓冲，能用正则的地方就先不用字符串比较），就算明知是牛刀也没关系，因为你换回的是更大的灵活性。代码量比较少，通常 bug 的数量也会更少。

他们只有在被逼无奈的情况下才会进行性能调优以及禁止修改接口和数据定义。例如，一支 Perl 团队可能会将一些关键的核心模块用 C 重写，然后创建 XS 绑定。时间一长，这些抽象就渐渐变成了既定标准，它们被包裹在数据定义和细致的 OO 接口里，再也无法修改。（就算是 Perl 程序员也常常会忍不住祭出银弹，为常用的抽象编写 OO 接口。）

那你觉得最终采用这些策略的结果会怎么样？

案例大分析

多年来我目睹了亚马逊客服应用的（各种形式的）强弱之争。一开始我是这样被归类的：

- 语言上我属于“强类型”阵营，最爱的语言是 Java；
- 协议（如在 SOAP 上的 XML/RPC）和 XML 建模（完全不要 DTD 和 schema）上我倾向的是“弱类型”阵营；
- 关系建模上则不属于任何一方（谦虚低调向专家学习）。

我观察下来的一个结论就是喜欢 Perl 的那些家伙总是能很快把东西做出来，速度快得吓人，连经验丰富的 Java 程序员也比不过。而且活儿干得又好又快，绝不是 Java 程序员想象中的那种 hack 一大堆的东西。他们的代码通常整洁有序，就算一开始有点乱，他们也会定期保养修复。有时候他们会顺手写一点 hack 味道很重的脚本，然而事实一再证明，这种能力往往是非常关键的。不过总的来说，Perl 和 Java 绝对可以分庭抗礼。就算性能出现瓶颈，他们也能发挥聪明才智想办法解决问题。

客服应用曾经为客户联络方式建立过一个关系数据模型，不过由于采用的是无类型的属性系统（说白了就是键值对），它并不完善。关系模型变化的频率很低，其中一个原因是当时流行的是集中控制的数据库，就算我们的软件工程师里有不少数据建模天才，修改数据结构仍然很麻烦。另一个原因是就算可以改，也赶不上需求变化的速度。所以灵活的联络属性就是我们的救命稻草了，就算静态语言阵营也得不同意这一点。

好吧，数据完备性的确是个问题。键值对模型则更容易发生名字错误（笔误）、无效值、无效依赖等，因为没有办法依赖数据库来保证约束。不过也正是因为这样，我们变得更加谨慎小心。每次只要发现数据不完备，我们就会设法写程序把丢失的数据填回去。这活儿有时候并不简单。但要知道就算是强类型的表格，也不可能彻底消灭数据完备性错误。所以不管是不是强类型，从错误中恢复过来的能力都是必不可少的。

好吧，有时候我们也会受到性能的困扰——可是 Java 也要面对这个问题，C++ 代码同样不能幸免（没办法，客服应用是个大杂烩，我们差不多要和公司里所有的系统打交道）。语言和性能其实没有必然联系，你要做的就是找到并且修复“瓶颈”。

不管怎么样，多年来我们都同时在 Java 和 Perl 下进行开发，这纯粹是当初相互博弈妥协后的结果。我们在讨论怎么实现 Arizona（客服应用的内部 Web 版）的时候，倾向 Perl 和 Java 的意见基本上是对半开。

不过真正开始开发的时候，Perl 的那部分完成任务的速度可谓惊人。有一段时间，Arizona 的 Perl 代码要比 Java 来得多，就是因为 Perl 程序员干完自己的活儿以后，把 Java 的那部分也拿过来顺手实现了。后来公司的“风向”逐渐倾向 Java。这个说来话长，这里先按下不表，总之几年以后，Arizona 里大部分的 Perl 代码都用 Java 重写了。（我听说我离开以后，风向又再次改变，Java 又渐渐被 C++ 和 Perl 赶了上来，不过那个基本上和语言本身没什么关系。）

不管这么说，多年来我见证了 Perl 和 Java 程序员一起开发同一个系统的情况，有时候甚至要分别实现同一份逻辑。我承认这种 Perl 和 Java 协同开发的方式确实效率不高，但当时选择那么做是有正当理由的，所以，我有幸亲眼目睹了一场强弱阵营之间长达数年的较量，而且是在同一个生产环境里，可谓针尖对麦芒。

总的来说，感觉很震撼。我当时是 Java 的死忠，不过不得不承认 Perl 确实比 Java 更精悍、更简单。Perl 的代码谈不上“干净”，反正它本来在这方面就名声不佳，不过模块化做得还可以。它的架构很好，能出活，表现稳定。

尽管我读 Java 已经没有任何障碍，但相比之下它（对我来说）确实复杂得多。我觉得 Java 程序员总是有一种过度工程化的倾向，我自己都不能例外。我想大概很多 Java 程序员会觉得我们的 Perl 代码写得太粗糙吧。可是如果它真的那么粗糙的话，早就应该出问题了。其实 Perl 代码里的大多数问题都是和外部服务（或者数据库，假如没有提供服务的话）交互的问题。大多数服务都没有为接口提供 Perl 绑定，结果我们的 Perl 程序员只好自己动手想办法绕开限制。

在数据建模方面，我学到了在关系模型下创建灵活属性系统的技巧。DBA 和

(特别是) 数据建模工程师通常不喜欢这种技巧，但是软件工程师却正好相反，因为它绕过了“真正”的对象关系映射里的限制，在出现新需求的时候也不用去修改数据模型。我很怀疑要是没有这套灵活的系统，我们还能不能跟上变化的速度。

我为什么支持弱类型

我现在觉得要解决亚马逊所面对的大多数问题，其实方法二（具备一定限制的弱类型系统）比方法一（没那么严格的静态强类型）更好。我们的业务一直在快速变化，所以要不断地调整接口和数据模型。要做的事情永远都比我们料想得多。当然啦，我们可以仰赖摩尔定律……除了它已经失效了 3 年之外。不过谁知道呢……

尽管我觉得客服应用的例子已经相当能说明问题了（这种长期大规模在同一个系统下比较两种不同理念的类型系统的事情可不常见），但它并不是我的唯一论据。

我更喜欢弱类型的另一个主要原因是我见过很多采用强类型的团队最终都放弃了。我见过一支团队彻底放弃“硬性规定”的接口，给一个 CORBA 接口加上一个 XML 字符串参数，我觉得这个决定实在是太英明了。他们被 CORBA 的类型系统折腾得死去活来：一个客户的细微改动就会影响所有客户。这个“后门”让他们能多喘一口气。

这种事情我见得太多了。Sun 的 JMX 接口看起来是强类型的，其实它通过一个字符串参数（“**ObjectID**”）嵌了一个弱类型的查询语言进去，编译器、接口生成器什么的对这个参数都一无所知。强类型的好日子可谓到头了。

有一次我们给某个大型体育厂商做了一个网站，他们希望让客户能在运动衫等商品上印上他们的名字缩写（或者客户号码等随便什么东西）。可是我们的订单模型（以及相应的强类型接口）都是定死的，根本没办法把自定义的缩写字段包含在发货请求里传给后台。（这个其实还不算发货请求，不过这不是重点……）我到现在都还记得当时那种好几个星期对不知道要怎么解决这个问题的焦虑。

而客户名缩写这种事情和无线设备（手机）所带来的模型变化相比就是小巫见大巫了。那种焦虑持续了好几个月。而无线设备和我们最近的项目比起来（至少从数据模型和接口设计的角度相比）又更加微不足道。

我还能举出更多第一手的资料。基本上，静态强类型一而再再而三地给我们带来麻烦，而弱类型却不会造成更多“糟糕透顶”的结果。毕竟不管用什么方法，糟糕透顶的事情总是避免不了的。

完全以我个人的观点来看，设计优秀的弱类型系统比同样优秀的强类型系统更有竞争力（用起来也让人心情更愉悦）。Emacs 就是一个例子。虽然我用 Java 比 Lisp 用得更好，但是我还是宁可给 Emacs 写插件，也不想给 Eclipse 写。简单程度至少差一个数量级。好吧，前提是你会 Lisp，而 Lisp 并不容易学。但是从长远来看，我还是宁可先多花点时间在学习上，这样的效率提升是永久性的。

其实 Emacs 的设计算不上一流，至少以现代的标准来看如此。要是当初写的时候用了新版本的 Lisp 的话，那就可以用到 OO 接口、名字空间、多线程的好处，少依赖一点动态作用域这些东西了。可即便是 Emacs 这样的庞然大物，为它写插件仍然要简单一个量级。要是你把 Emacs 和 Eclipse 的比较范围严格限定在用户扩展机制上，那么这个强弱类型的比较是极具说服力的，弱类型再次获胜。

回过头来看 Java：就算是 Java 的拥护者也认为 Java 的反射、动态代理、动态类加载、可变参数等无法进行静态检查的特性至关重要。他们虽然会警告你可能存在的性能缺陷，也会告诫你不要太过依赖这些动态特性——但我很怀疑 Java 社区会愿意放弃这些东西。这些特性被认为是采用 Java 的主要优势。

我曾经是强类型的大粉丝，不过随着阅历增长，我开始觉得其实我错了，至少对于我们搭建的那些系统来说如此。假如你在银行工作，强类型当然可以胜任。假如你的业务几乎不怎么变化，稳定的数据模型自然没什么问题。假如你所在的行业对性能和安全有异常严格的要求，那么强类型系统是自然的选择。

上周末我看了新版的《银河系漫游指南》，里面有一幅非常好笑的讽刺沃根人英国政府式的官僚的漫画。看完后我告诉自己：我讨厌官僚主义。而静态类型系统基本上就是官僚主义。我不希望被它们束缚住手脚，不用填一大堆表格什么的就能把工作做完。假如非要和一个笨到死的编译器和严格的类型系统打交道才能获取所谓的静态类型安全的话，那还是谢谢你吧，我自己就能处理好类型错误这种事情。

弱类型的能力够不够强？

要是说疑惑，我心里还是有的。弱类型系统是不是天生就扩展性不足？是不是真的像静态类型阵营宣称的那样，达到一定规模后，弱类型系统就是一个巨大的坑？运行时类型错误率会不会失控，就算有详尽的单元测试和软件工程规范也无济于事？

另外，性能的代价真的昂贵到不可接受吗？举个例子，你知不知道有哪些弱类型的大型系统最后用静态类型语言重写，来达到所需的性能目标的？（要求是同一个团队，这样才能表明重写并非是因为语言喜好。）我对那种分布式系统的失败案例特别感兴趣，不是嵌入式系统或是给终端用户使用的桌面程序。

我觉得这会是个问题，理论上，我觉得在亚马逊，用 Ruby 或 Lisp（或 Smalltalk、Python 等其他支持代码模块化和面向对象抽象机制的动态语言）来构建直接面对客户的大型服务是完全可以做到的。不过我还是希望先看到一些成功的先例。

时至此刻，我觉得强制性的静态类型（例如，Java、C++、Ocaml、Ada 这种语言）是妨碍进步和灵活性的。当然我也觉得完全没有（比如 Ruby 和今天的 Python）也是有问题的，因为等到系统的运用模式稳定下来以后，就没有办法有选择地收紧了。我觉得 Lisp 的方法还蛮接近理想状态的，即静态类型可按需添加。

由于 Ruby 的性能以及缺乏原生线程的支持，我仍然对于用 Ruby（我选择的弱类型语言）编写任何大型应用有所犹豫。对于 Common Lisp 我也一样有所保留，主要是因为 Clikli 上的软件包实在是不够看，语言本身的趋势也没有好到让我有信心选择它。对所有其他选项的态度也都差不多（比如 Python、Erlang、Scheme、Lua）。

可是我也不想再写 Java 和 C++ 了。

这个问题有点棘手。

注：出于两个原因，这篇文章里我故意误用了“强”和“弱”这两个词。首先是为了强调我要表达的东西不仅仅是编程语言，数据和接口建模里也有这个问题。其次是这样诗意一点。

在谈到编程语言的时候，我其实是在说静态类型和动态类型。我知道其实最好应该用二维表格来表达静态/动态和强/弱的组合，比如本杰明·皮尔斯在《类型和程序设计语言》的第1章里所采用的方法。

之所以专门强调一下，是因为有一些Python的粉丝实在是不可理喻，他们选择性地无视了我文章里后Python（或者Smalltalk等）时代的观点，因为他们拒绝承认Python的动态类型属于“弱类型”。显然Python和我的诗不怎么合拍。

第 2 章 Chapter 2

代码里的哲学

作者手记：软件需要哲学家

写这篇文章的时候我已经失眠好多天了，因为之前写的另一篇文章（没有收录在这本书里）“Lisp 不是合格的 LISP”。那篇东西引起了不小的骚动，这是我原本没有料到的，我的 Lisp 经验不算很丰富，所以其实我是没有资格去品评 Lisp 的。不管怎么说，很多人都不喜欢它。

我在文章里犯了不少技术错误，结果被人抓住小辫子，试图将焦点集中在细节上，让人忽略整体。他们之所以这么做是因为我指出了大多数 Lisp 拥趸不愿面对的黑暗面：流行度。

好在一些广受尊敬的资深 Lisp 黑客对我表示了理解，同意我的立论还是正确的，即 Lisp 应该更受欢迎，但不应该像现在这么时髦（作为标准来说），而 20 多年来社区在这方面似乎也没怎么努力过。

我觉得最让我震撼的是那种情绪上被激发出来的暴力倾向。而我一直以为 Lisp 众应该有足够的自信来面对有建设性的批评。至少 Perl 众在这方面就自我感觉特别好。随便你怎么骂，Perl 众都可以很淡定地耸耸肩，点点头，微笑一下继续工作。

可是我发现 Lisp 社区里却有相当一部分人非常缺乏安全感，面对批评非常容易受伤。很多语言社区里都有这种情况，但是我从来没想到 Lisp 会有这个问题。

于是我失眠了，翻来覆去，结果以创纪录的速度写下了这篇文章，前后只花了两个小时。写完之后如释重负。虽然这篇东西仍然很具争议性，但爱怎么样就怎么样吧。

警告：我在这篇文章里对有组织的宗教表现得非常严厉。我对这个不感冒。我知道其实在统计学上，你很有可能属于某个有组织的宗教，所以在这里我先说声对不住了（特别是天主教）。

不过真要是碰上了，我是非常抗拒各种组织的。我觉得大型组织几乎总是会表现出一些很典型的失败信号，而创始人是预见不到这一点的。它们会渐渐累积起各种“糟糕的”组织类型的特征——拉帮结派、狂热的崇拜、元老会、激进的种族主义小团体。你肯定还能想到其他的。

基本上我觉得大型组织（特别是那些有些年头的）都不怎么样。我甚至觉得罪魁祸首是代码本身。我们就是不擅长代码而已。而在大型组织里又特别容易出现这种问题。

所以在这里再次声明：我并不反对唯心论，也不反对任何一种信仰，只要它们不要为了自己的利益过分膨胀就好了。其实我觉得信仰和任何狂热行为就是唯心论的两面。没有信仰是不可能的，在编程和设计领域里也不例外，但是一旦出现狂热的苗头，那就大事不妙了。

我希望你能懂我想说什么。

软件需要哲学家

软件需要哲学家。

这个想法一年来一直在我脑海里盘桓不去，最近更是像肿瘤一样愈发不可收拾。网上很多人大概都会幸灾乐祸地看我被折腾死吧。

现在的哲学家已经不怎么受人待见了。大众对哲学的印象停留在它只是用来争

论那些永远也没有正确答案的雄辩之术。“哲学？饶了我吧。咱们还是说点实在的！”

滑稽的是，正是哲学家们教会我们理性思考，坚持事实的能力。要没有无数哲学家们的努力，揭露和分享事实仍然是一件会让人受伤害，甚至掉脑袋的事情。

下面这个事实不知道有没有震撼到你一点呢？哲学在兴盛了几百年后（差不多是公元前 400 年到公元后 100 年），我们在罗马天主教廷的思想奴役下生活了整整 1500 多年，任何胆敢有不同意见的人都会被杀死。

而更奇怪的是我们似乎假装这一切压根就没发生过一样。我们总是在脑海里跳过那上百代人所经历的宗教统治，把今天的宗教想象成帮我们照顾小孩的慈祥老爷爷。大家相安无事。过去的事情就让它去吧。当年在上帝的名义下进行过大屠杀、十字军远征、种族灭绝，在宗教裁判所里发生过各种惨不忍睹的酷刑等。但大家当时都还年轻，不是吗？现在没人会再做这种事情了，至少在“文明的”国度里不会发生这种事情。

我们还是不要去想那些未开化的国家好了。

正是哲学家带领我们走出黑暗时代，相当一部分为此献出了生命。而今天，哲学专业是大多数笑话的主角，因为我们在启蒙之后，就忘了自己当初为什么需要哲学家了。

要是真的去想一下这个问题，我们都会说那些以信仰的名义（不管是上帝、政党，还是任何一种暴力形式的洗脑）犯下的恶行，其实都是别人干的，那些人跟我们完全不是同一类人。

一般大家都觉得自己生活在启蒙时代，其实并非如此。人类其实压根就没怎么变过。我们依旧相互折磨，杀伐不断。决定要杀人然后付诸行动简单得像呼吸一样。全世界每时每刻都在发生这样的事情。折磨也是一样。

可他们也是人。如果他们和你生在同一个城市，或许他们会和你一起上学，交朋友、学编程、写博客、写评论，绝不会为了一个想法就去杀人或折磨别人。他们原本和你没什么两样。你只不过走运，生对了地方而已。

我博客的上一篇文章里有一条咒我快点去死的评论。当然啦，他们只会在网上大放厥词而已。可要是他们真的觉得，仅仅为了一个滑稽好笑、完全无足轻重的话题（Lisp 到底有没有一个过得去的实现）就可以威胁他人生命的话，你会有什么感觉？

所有在博客下面放狠话的都上钩了。你们（不管有没有匿名）都是宗教狂。唯一例外的批评人士是保罗·科斯坦萨（自称很固执，讽刺吧），还说自己提出的技术修正其实是有点迂腐了。可对于这样的评论我是相当欢迎的。任何技术上的概念错误我都乐于改正。但这不是重点，就算我在那篇文章里的所有设计技术的地方都说错了，下面的评论里也没有一条能挑战我的观点本身（现在的 Lisp 不是合格的 Lisp）。

这些人其实根本没看懂我在说什么，这倒没什么，反正也无伤大雅。要是你有很多年的 Lisp 经验，写过文章出过书，还写过无数行 Lisp 代码的话，你大概已经不记得它最初的模样了。这些人已经不再有怀疑精神，变得宗教徒一般狂热。

其实在任何语言之争中，相当一部分人完全明白对方在说什么，可他们还是想要彻底封杀对方：去死吧！毕竟死人是不会说话的。你还得抹掉他们的所有言论，现在要做到这一点已经不太容易了，所以网上的言论也就越来越杀气腾腾。

我博客的老读者都知道我想表达什么。我是故意站到对立面去的，而且是旗帜鲜明地反对。铁了心要和技术宗教对着干，我再也受不了脱离事实了。

事实：函数在 Java 里不是头等公民，Java 也没有宏。结果在解决问题的时候只好想方设法地扭曲代码，而随着代码量的增加，它会变得越来越难看，这个是 Java 独有的缺陷。Lisp 程序员对这一点看得很通透。Python 程序员和 Ruby 程序员也一样。结果 Java 程序员就受不了了，叫嚣着“宏的威力大得过分了”，“你怎么知道我不懂”，“去你的吧，白痴，Lisp 永远也赢不了”。

你觉得这些玩意儿我每天收到的还不够多吗？

我可不想和这种 Java 狂同处一室，天知道他们会干出什么事情来。

要是把矛头指向 Python 会怎么样？（我觉得很滑稽的是）这时 Java 程序员会给我发邮件说：“我一直都很讨厌 Python，你把我的心里话都说出来了。谢谢！”而这时 Python 程序员会怒从中起，恨不得在键盘上按一个键就能“干掉那个混蛋”。

而昨天我把枪口对准了 Lisp。反正写什么是我的自由，而且老婆还在家等着呢，所以我写得很潦草，几乎没怎么斟酌。结果 Lisp 程序员只看了一眼，就像复仇的女人一样怒火中烧起来。

其实我是不是一挥而就压根不是重点。相反我还挺高兴自己写得很快。就算真的去花时间花精力把那些没用的细节“弄对”（比如怎么通过奇怪的设定来重写不具多态的 `length` 方法，而这种东西本来就应该基本配备），效果也是一样的：Lisp 的狂热信徒总会有办法把技术讨论变成对喷。与其这样还不如省下这两三个小时呢。

好吧，就算我写的东西是找骂好了，毕竟我事先确实没怎么做功课。这篇文章只不过是我去年以来，尝试 Common Lisp，然后又花了一年时间尝试各种 Scheme 方言后累积起来的怨气罢了。至少对我来讲，目前的 Lisp 都不及格。这只是个人感觉，但我打算坚持自己的观点。

当然我离不开 Lisp。学到一定程度后，它会变成你的一部分。我自己写了很多 elisp 小程序。不过给我留言的人说得也没错，我确实没有用 Common Lisp 写过什么真正的大程序，可那是因为每次我想认真地去尝试的时候，总会遇到太多障碍。风险太高，收益却不够高。相信我，我是真的想要好好用它的。

可是多次努力失败后，我彻底放弃了 Common Lisp。他们不让我在工作中用，要知道在我工作的地方，Lisp 程序员的人均占有量应该是全世界大公司里最高的了，其中更不乏颇有名气的程序员。可要是在工作中也不让用，那这门语言又怎么称得上合格呢？我觉得光这一条就很有说服力了。

我现在更感兴趣的是我的帖子被认定为乱喷后所产生的情况。在一个充斥着有各种宗教诉求的狂热信徒的世界里，会出现什么样的社会现象（而且是在哲学家备受误解，现实中的宗教逐渐式微的情况下），我对此非常好奇。

让我们再来仔细看看我们生活的这个世界，首先我觉得非常有趣的一段话，来自彼得·西贝尔：

我一直努力希望找到我们会花那么多时间去编写我们完全不喜欢的东西的理由。直到我学到了 HCGS 这个词^①，太感谢了。

他的第一句话谈的是社会学。他的观点并不新鲜：要是没有建设性的观点，最好是闭上嘴。人们有选择的权利，质疑别人的信仰是不对的，特别是当他们生活困难的时候。

这时就该哲学家登场了。他们会将你信仰的东西大卸八块，然后告诉你信仰的归宿在哪里，包你永生难忘。我不是哲学家，对这玩意儿一窍不通，不过我非常希望有这样出色的哲学家来解析一下我们的技术社会。

我当然明白这样思考问题会带来什么后果。首当其冲的就是对语言本身毫无益处，尤其是 Lisp 和 Scheme，它们本来应该在创新的道路上相互竞争的，结果却不思进取，自取灭亡。

编程语言都是宗教。我曾经还有点犹豫，觉得“宗教”这个词有点过，可是现在不再有这种疑虑了。两者之间区别真不大。语言宗教里至高无上的自然就是语言本身，它是神一样的存在，是顶礼膜拜的对象。

任何宗教组织都会有一位教皇（或是在那种政府不择手段将自己塑造成神一样的国家里的政治局主席）：他是精神领袖，神的代言人。而这通常都是设计语言的那个人。Lisp 的情况比较复杂，因为麦卡锡、萨斯曼，还有斯蒂尔作为各自语言的精神领袖并不怎么活跃。

有一定规模的宗教组织肯定会分权力层级，编程语言也不例外。它也会有红衣主教、主教、牧师、信徒之分：越是接近圣火，离权力中心就越近，等级也就越高。这是刷存在感的好方法：基本上就是一个排行榜。这对提升自信很有好处，不过也会拉低你的辩论水平，因为你在自身形象上投入了太多情绪化的东西。

^① 译注：这个词是 Helpful Critical Guy Syndrome，来自呆伯特的漫画。直译的话是帮助性批评男性综合症。专门指那些只会给出没营养却正确的建议的男人。

你觉得自己可以通过贡献技术和文档来升级，但实际上这完全只取决于你有多少信徒和粉丝，你的言论有多少人听到而已。

这就是保罗·格雷厄姆成不了 Lisp 的教皇的原因。他其实是有资格的，但可惜他是个异端分子。不知道你们注意到没有，在我上篇博客里的评论中，几乎没人提到我对他的评价。唯一一个（到目前为止）也只是拿他为 Common Lisp 背书而已。

面对现实吧，让这种异端分子曝光太多并不明智，这样会有越来越多的人听到他们！彼得，这下你明白我为什么要写那么多我明明不喜欢的东西了吧？我真的想喜欢它的，但是它实在是烂泥扶不上墙，无论在技术层面还是文化层面上都有致命缺陷。这就好像就算我想去信仰罗马天主教却做不到一样，因为这帮人在历史上的所作所为实在令人无法正视。

我出生在一个信仰罗马天主教的家庭，13岁的时候决定不再信仰它。我的叔叔弗兰克（虔诚的天主教恐怖分子，假设真的有这样的人存在的话）让我别在读“圣经”了，因为那“真的会毁掉一个人”的。一定要有人阻止你那么做。当然这不是我放弃信仰的唯一理由，不过也是一个很充分的理由了。

技术上我天生就是一个汇编语言程序员，至少那是我第一份正式的工作，大学毕业后一干就是5年。不过写汇编太折磨人了，所以之后整整7年我都皈依了Java教。然后我实在受不了Java的缺点，就在这时，保罗·格雷厄姆和他早期的文章出现了，他给我带来了Lisp，太棒了！

可问题在于，每次改变信仰，下一个对你的影响力就越低。俗话说，一入佛门，终身弟子。我不知道这句话在我身上能应验多少，因为我是汇编语言的狼孩子，但这似乎意味着我再也不会对其他编程语言那样着迷了。那既然现在我选择了红色药丸，我有什么选择呢？至少我要努力让人们知道外面的世界是怎么样的。

有趣的是，扼杀我继续学习Common Lisp的罪魁祸首，也正是扮演了弗兰克叔叔角色的那本书《Practical Common Lisp》，这是彼得·赛贝尔的名著。正是彼得让我看清了这只野兽的真面目。市面上的其他书籍都把Lisp伪装得纯洁美丽、一

尘不染，把所有丑陋的部分都归结为“取决于具体实现是怎么定义的”。可当我发现要编译一个勉强可移植的 Lisp 代码库有多麻烦的时候，我就直接投降了（还是在亲自尝试过几次以后）。

所以相比 Lisp 的实现，我更喜欢 Lisp 这个概念本身。

老实说，这篇文章我酝酿了至少一年。开了十几个头，尝试过从不同的角度切入。我非常想要表达软件需要哲学家这个观点。我们需要伟大的思想家——菲奥多尔·陀思妥耶夫斯基、大卫·休谟、亚里士多德、让·保罗·萨特、本·富兰克林、伽利略·伽利莱、伯特兰·罗素和阿尔伯特·爱因斯坦这样的人来指引我们走出软件的黑暗世纪：这个每一寸都深陷黑暗和无知的时代一定会像中世纪一样被人铭记。

但是我失败了。这不是我想要写的东西，我既不是伟大的思想家，文章写得也一般般。但是这不表示我将来没机会，就算现在不是，或许将来有一天也说不定。所以与其藏着掖着希望将来有一天一鸣惊人，还不如现在就把想法分享出来。

要是你觉得很奇怪为什么 Lisp 社区被我的小小抱怨所激起的反弹又猛又烈的话，我希望能帮你解惑，为什么这是不可避免的。基本上他们都是经验丰富的老手。Lisp 是最古老的技术宗教之一，他们都经历过类似的宗教追求。

但这不是重点。这里的关键是他们其实就是你。每当遇到一些事物你觉得很好很强大，别人却批评抱怨的时候，你总会忍不住希望他们赶紧离开，好继续坐井观天，这时千万别忘了这一点：之所以我们今天能冲破黑暗时代，生活在一个相对光明的启蒙时代，正是因为我们敢于挑战自己最宝贵的信仰。

所以质疑是好事。

注：

1. 没错，R6RS 我读完了。这本书里的内容基本是在各种重要的问题上做一些不温不火的妥协。它无法让 Scheme 更上一层楼，这在我看来是无法接受的，因为本来它是可以有所突破的。不过无所谓了。你就把这条注脚当成胡说八道好了。要是这就触到了你的逆鳞，那说明你是技术宗教的脑残粉，希望我能在有生之年找

到办法解救你。回复之前好好想想我的话。

2. 在这里声明一下，那些说问题其实出在 IDE 上的回复我还是很赞同的。哪怕是 SLIME 这样的神器也无能为力。比如，它在 Windows 上就用不了。不过那只是 Lisp IDE 问题中极小的一个罢了。而且在博客上要把它们全部讨论清楚也是没有意义的。或许这些讨论本身就没有意义，因为 Lisp 压根就没有出色的 IDE，讨论只会让我更加难过 (Elisp 除外，Emacs 和 Elisp 是绝配)。这也是我现在只玩 Elisp 的原因。

作者手记：代码的天敌

这又是一篇 Java 的吐槽文。不过这次要说的不是类型系统。它主要感叹的是 Java 本质上缺乏组合以及压缩的能力。另外，我想和那些对于攻击 Java 的内容感到不适的人说，这是本书中最后一篇涉及这个话题的文章了。放轻松！

不过核心问题并不只是 Java 本身，而是更深层次的东西。我现在越来越觉得（即便是 5 年之后再来看这篇文章）其实所有的代码从某种程度上来说都不是好东西。代码是负债。所有的问题，延展性、维护性、工具链、分析性、构建等问题，都是由代码造成的。代码量越大，问题也就越严重。

我知道，我们都热爱代码。我喜欢写程序的程度不输任何人。写程序很好玩，也很重要。但是，代码也是数据。这一点大家都知道吧。而且我们也都知道，数据量变大会产生问题。就算（可能是）全世界最擅长处理大数据的 Google，也很容易制造出大到无法控制的数据集。

这里的道理很浅显：人类能处理的代码量是有限的。这一点无可否认。代码给人的感觉通常很轻柔，但实际上代码属于高度结构化的文本数据，具有异常复杂的处理要求。上至一定规模后，任何公司都会觉得很头疼。

所以归根结底，就是代码压缩到什么程度才是可以接受的，你究竟愿意放弃什

么来保证代码规模足够小。

在这篇文章里，我抛出了一个我觉得相当有说服力（至少也是能激发思考）的理由，假如你是 Java 程序员，那么你要做出“终极牺牲”，抛弃 Java。显然他们是不可能听我劝的。反正听不听在他们，遭罪的人又不是我。

代码的天敌

我是个程序员，今天我休息。猜猜我在干嘛？虽然我很想告诉你我在巴哈马享受鸡尾酒，但实际上我休假的时候还是在写代码。

因此所谓的“休假”只是对 HR 来说的——我只是放下工作，好专心让我的游戏重新上线。这个游戏我写了快 10 年，开发的时间差不多有 7 年。这个游戏下线已经好一段时间了，现在重新上线，部分原因是为了摆脱那些一直追在我屁股后面的玩家。这至少要花一个星期，所以我只好休一个星期的假来搞定它。

那为什么这个游戏下线了呢？主要原因是为了避免它流行。对一个基本上只靠一个人，完全用业余时间做出来的游戏来说，它算是相当成功的了。注册玩家超过 25 万（至少创建角色的人次达到了这个数字），几年下来积累了好几万活跃账号。得过奖，也在杂志上有过专题报道。很多游戏门户都关注过它，甚至还吸引到不少投资人，当然还有好多孩子。

你没看错，我写的就是孩子。原本这个游戏的目标客户群是大学生，结果意外的是也吸引了好多小朋友，我原来一直以为那个年龄段的小孩只喜欢打游戏机呢。这个游戏是为我自己写的，显然很多人也和我一样喜欢这样类型的游戏，大家一起建立起一个相当稳定的小社会。

游戏下线的原因其实很简单——游戏要升级，而我的工作又很忙，晚上实在没时间等。但这些原因归结起来其实暴露出一个更深层次的问题：我一个人已经掌控不了这样的代码规模了。

我花了近 10 年的时间，结果造出一个失控的庞然大物。

这让我陷入了沉思。过去四五年来，这个问题占据了我在技术上的思考的很大

一部分，我写的博客和代码都深受影响。

我接下来假设你是一个年轻聪明的大学生或是高中生，希望将来成为一个更好，甚至更伟大的程序员。

（请不要据此推断我这是在暗示自己是个伟大的程序员。我还差得远呢。十几年来我的代码都没法看，然而在这过程中，我学到了很多很多，在此分享给你，希望能帮到你。）

在这里我要假设你是年轻读者，因为如果面对“资深”程序员，我的血压会直线上升，根本没办法专心写东西。待会你就知道为什么了。

好在你是天天向上的好孩子，我可以告诉你真相到底是怎么样的。睁大眼睛好好看着，几年后就明白我说得对不对了。

少数派观点

我对代码的观点是属于比较难得和另类的。我坚信代码最大的敌人就是体格。

“体格”在这里只是一个代词，因为我再怎么苦思冥想也找不到更好的词汇了。后面我会尽力解释，到时候你就明白我想表达什么了，说不定你还能找到更好的名字。“膨胀”或许更准确一点，没人喜欢“膨胀”，但可惜很多所谓的资深程序员也不知道怎么界定膨胀，那些明显肥胖过度的代码在他们眼里也像铁轨一样苗条。

还好我们不用和这样的家伙打交道，你说是吧？

之所以我的观点比较难得是因为很少有人会注意代码的大小，没人觉得这有什么问题。相反，大多数人觉得这根本不算是什么。这就是说和我持相同观点的人在别人眼里和神经病差不多，否则正常人怎么会无病呻吟？

业界对那些表面上能帮你管理大规模代码的玩意儿往往表现得异常激动，比如能让你像操作“代数结构”那样操作代码的 IDE、搜索索引等。代码之于他们就像水泥之于建筑工人，他们需要重型机械来搬动它们。保存水泥是有专门的方法的：因为水泥无法压缩（就算可以也不会太多），所以只能用各种办法把它们铲来铲去。甚至连编程面试都会问到（当然是比喻）如何搬动山一样高的水泥，答案是一次一

卡车。

业界程序员非常着迷于解决这种不是问题的大问题。那只不过是一堆水泥而已，只要有足够大的工具就能移动它了。水泥很无聊，但工具不是。

而我的少数派观点是，对任何人、团队或者公司来说，没有比一大坨代码更糟糕的东西了。代码量太大会压垮项目和公司，而且代码到了一定规模后除了重写别无他法，任何聪明的团队都会尽一切努力避免代码成山。不管有没有工具。这就是我的理念。

事实上人都是不见棺材不掉泪的。比如我曾经用一门非常恶心的语言写了一个很棒的游戏，外表看起来光鲜亮丽，内部实现却能吓死人。普通程序员大概会对此不以为然，甚至还觉得怎么没有单元测试（对此我也有点遗憾），这会让已经高达 50 万行的代码量再翻一倍。所以从某种程度上来说，他们可能还会觉得不够大吧。如果我真的按照今天的标准来做的话，结果肯定更糟糕。

为避免误读，这里专门声明一下：单元测试绝对是好东西。事实上我觉得它的作用至关重要，对于当初做游戏的时候没写单元测试，我一直都耿耿于怀。我想说的是，我写这个游戏的方法和任何有经验的程序员都会告诉你的方法是一样的，然而它还是彻底失控了。而我要是依照“正确的”方法去写单元测试的话，它的体格还会变得更吓人！这种两难的悖论正是我要坚持精简代码规模的关键因素。

大多数程序员都从来没有经历过这种悲剧。就算真的发生了，他们也常常不觉得那算什么问题，就像建筑工人不会觉得灰尘有什么好大惊小怪的一样。什么工地没有灰啊，根本谈不上好坏，至多是一点小障碍罢了。

很多公司都要面对数百万行的代码，不过在他们看来这只是工具的问题，仅此而已：尘土多了没关系，移走不就好了嘛。

绝大多数人一辈子都没有单独维护过几十万行的代码库，所以我的眼界和他们的是不一样的。希望年轻好学的你能意识到，在这个问题上，只有那些真正接触过那种规模代码的人才有资格发表意见。

你肯定会听到很多人很鄙视我，觉得我说的东西不着重点，“不学无术的家伙”，他们会摇摇头如是说。但我敢断言那些说这些话的家伙肯定从来没有收拾过自己留下的烂摊子。

如果那几十万行代码全是你自己写的，那问责的也只能是你自己。要怪也是怪自己，正因为如此，我才会成为少数派。

当然我的游戏并不是唯一的因素。光是它还不足以让我得到教训。从业 20 年来，我推倒重写过很多你想象不到的巨大代码，也学到了一些很多人穷尽职业生涯也体会不到的东西。我不奢望你现在就作出判断。只希望你不要对自己的代码着急下结论，保持开放的心态，过几年再回头来看看。

看不见的膨胀

首先要定义什么叫剧烈膨胀。我知道自己没这个本事，但还是尽力而为吧，希望能通过勾勒问题，让你对它有个大致的概念。

代码里能出现的问题就那么几种，对大多数人来说并不难界定。

首先是复杂度。没人喜欢复杂的代码。衡量复杂度的常见方法是所谓的“循环复杂度”，也就是通过简单的静态分析代码结构来估算所有可能的执行路径的函数。

我百分之百确定我自己是不喜欢复杂代码的，但我也觉得衡量循环复杂度有什么用。因为只要把代码拆成小函数，就可以提高循环复杂度的分数了。10 年以来，把代码拆成小函数的做法一直都被标榜为“良好的设计”，这完全要归功于马丁·福勒的《重构》。

但重构之于 Java 这样的语言，问题就在于，它会令代码量膨胀，这也是我今天要说的重点。我估计现在 IDE 支持的标准重构技术中，大概只有不到 5% 有精简代码的作用。重构就像是在清理衣柜的同时保证不会丢掉任何东西一样。只要换一个大一点的衣柜，把所有的东西分门别类地摆放好，衣柜自然就会变得很整齐。但是程序员往往忽略了一点，那就是不愿意丢掉不要的东西怎么能叫大扫除。

这就引出了第 2 个明显的代码问题：粘贴复制。很多程序员都在这个问题上吃

过亏。这种事情根本不用专门去记，因为是个人就会在这个问题上栽跟头，不管你喜不喜欢。粘贴复制在电脑上太容易了，常在河边走，哪能不湿鞋？记住，代码永远都在变，永远永远永远都在变，一旦你要在 N 个地方修改相同的代码，而 N 大于 1 的时候，你就知道什么叫痛苦了。

然而粘贴复制的伤害非常隐蔽，很多经验丰富的程序员也会被打得措手不及。其实它的本质就是重复（*duplication*），可惜的是有些类型的重复在 Java 里是无法避免的。这些重复遍布 Java，无所不在，结果 Java 程序员很快就失去了辨别它们的能力。

Java 程序员常常会疑问为什么马丁·福勒弃 Java 而去，投入 Ruby 的怀抱。虽然我不认识马丁，不过我想应该是他在用 Java 的过程中遇到了什么“让人崩溃的事情”吧。有趣的是（除了马丁自己），我觉得这个“让人崩溃的事情”说不定就是撰写《重构》，这本书告诉 Java 程序员要怎么扩大自己的衣柜，把东西放得更整齐，同时也让马丁自己意识到，他想要的其实是在一个更好更漂亮、正常大小的衣柜里放下更多的东西。

马丁，我猜对了没有？

正如我开头预料的那样，除了一些语焉不详的词汇外，我还是没能定义到底什么叫膨胀。为什么我的游戏会有好几十万的代码？这些代码到底都干了些什么？

设计模式不是特性

软件设计方面另一本意义深远的大作是《设计模式》，它狠狠地扇了世界上每个程序员一记响亮的耳光——假设这个世界上只有 Java 和 C++ 程序员。

《设计模式》写于 20 世纪 90 年代中叶，它设计了 23 个漂亮的盒子来帮你整理衣柜，外加一系列扩展机制，好让你自己定义新盒子。对我们这些衣柜像狗窝一样，几乎没有盒子、袋子、隔板或者抽屉的人来说，这本书真是大救星。我们只要改建一下屋子，把衣柜扩大 4 倍，眨眼之间它们就能变得和 Nordstrom 百货商店的货架一样干净了。

滑稽的是，销售们对《设计模式》没什么兴趣。项目经理、营销部门，乃至工程主管都对它兴致缺缺。只有程序员在说到《设计模式》的时候才会兴奋异常，特别是那些使用特定语言的程序员。Perl 程序员基本上就对《设计模式》没什么反应。不过 Java 程序员却很令人费解地得出这样的结论：Perl 程序员不喜欢设计模式是因为他们太邋遢，只有屌丝才会把衣柜里的衣服堆到天花板上。

不过现在大家都清醒过来了，不是吗？设计模式不是特性。工厂不是特性，委托、代理、桥接也都不是。它们只是提供了漂亮的盒子，以松散的方式来装载特性。但是别忘了，盒子、袋子和隔板自己也是要占用空间的。设计模式也不例外（至少在“四人帮”的书里所介绍的大多数模式都是这样）。更悲剧的是，“四人帮”模式里唯一能精简代码的解释器（Interpreter）模式却被那些恨不得把设计模式纹在身上的程序员忽略了。

依赖注入是另一个新型的 Java 设计模式，Ruby、Python、Perl，还有 JavaScript 程序员大概听都没听过吧。就算他们听过，他们也能（正确地）得出他们根本不需要这种玩意儿的结论。依赖注入是一种惊人的描述式架构，让 Java 能在某些方面和更高级的语言一样，变得更动态一点。你猜得没错，依赖注入会让 Java 代码变得更大。

变大是 Java 中无法回避的东西。成长是生活的一部分。Java 就像是俄罗斯方块，不过积木和积木之间的空隙都填不满，结果只能越堆越高。

数百万行的代码

最近我有幸听了一位自称 Java 程序员的人做的演讲，其中一页列出了（当前 Java 系统的）问题，下一页则是（新雾件系统的）需求。他列出的第一个问题就是代码的体格：他的系统有数百万行代码。

哇哦！这种事情我也经历过，所以感同身受。Geoworks 拥有上千万行汇编代码，在我看来这是他们最终破产的帮凶之一（当然似乎没什么人这么觉得——那些程序员永远不懂得吸取教训）！我在亚马逊干了 7 年，他们拥有的代码量超过 1 亿行，以各种语言写成，“复杂度”常常在内部被认为是最糟糕的技术问题。

所以我很高兴看到这家伙把代码量列为他的头号问题。

但接下来的事情让我大跌眼镜。当他翻到了需求那一页的时候，我发现他把“必须能支持百万级的代码量”列为需求之一。会场里所有人（除了我）都点头表示同意这个观点。于是我就崩溃了。

为什么会有把头号问题列为新系统的需求？我的意思是，你在列出需求的时候，不是应该想办法解决问题，而不是假设问题会再次出现吗？所以我站起来打断他，要求他解释他到底在想什么。

他答道：因为他的系统有很多特性，特性多了，代码量自然也会增加，既然如此，百万级别的代码量就是必然的结果了。“这不代表 Java 是一门啰嗦的语言！”他补充道——不管从哪方面来说，当时的情景都是非常滑稽的，我压根就没提到过任何和 Java 或者啰嗦有关的话题。

假如你读完这则小故事后感到震惊，心想“这个家伙莫非是瞎了不成”，那么恭喜你，你已经是编程界里的少数派了。或者说，不受欢迎的那一派。

大多数人可以用对待代码量大小的态度来划分。Java 程序员的情况特别严重，但并不是唯一有这个毛病的。一方面，他们知道代码量太大是不好的。这一点只要具备小学算术水平就能明白。比方说有 100 万行代码，每页 50 行，那就等于 2 万页的代码。光读一遍要多久？视情况不同，想要看明白整体结构恐怕就要几个星期乃至数月。更不用说架构上的改动了，那是要按月和年来算的。

而另一方面，他们又觉得有了 IDE 就可以不用担心代码量的问题了。我们马上会谈到这个问题。

说实话，100 万行代码确实是很小的量。很多公司要是只有 100 万行代码估计都要偷笑了。通常一支小团队就能在两年内搞出那么多代码来。今天的大公司手上至少也有好几亿行的代码。

长话短说吧，如果把我从编写那个 Java 游戏中学到的教训总结成一句话，那

就是：假如你下定决心要精简代码量，那最终的结果就是被迫放弃 Java。相反，如果一定要用 Java，那么最后肯定会搞出几百万行代码来。

那么这个交易值不值得？Java 程序员肯定会说值得。这样一来，其实他们就默认了代码量太大不是好事，所以至少这方面你赢了。

但是“Java 程序员”说的东西是不能尽信的，因为“X 语言程序员”的话可信度并不高，不管这个 X 语言是什么。现在想成为优秀的运动员都要进行交叉训练。程序员也要掌握各种特点不同的语言后，才能真正做出正确的设计。

而我最近发现 Java 是一个尤其糟糕的 X 语言。如果你一定要雇用一个 X 语言程序员，不妨找一个 Y 语言的程序员。

其实我刚开始写这篇东西的时候并没有打算针对 Java（或是 Java 的拷贝 C#，它虽然比 Java 好一点，但是基本特点还是一样的，所以最多就是险胜）。我的这种少数派的观点适用于任何语言、任何代码。体积膨胀都是不好的。

无论如何，我现在只能死马当活马医，先把注意力放在 Java 上，尽力挽救这只巨象。别怪我，这不是我的错。如果有人和我一样也有一只 C++ 大象的话，一定能理解我的处境。不过现在，关于 Java 造成代码膨胀的话题就先到此为止吧。

IDE 是救星？

Java 社区坚信（几乎是百分之百）先进的 IDE 可以化代码量过大的问题于无形。

这种观点有几个问题。首先是简单的算术：只要代码量足够大，最终会为了管理代码而耗尽全部的机器资源。想象一下用 Eclipse 或者 IntelliJ 打开一个 1 亿行代码的项目时会发生什么。CPU、内存、硬盘，还有网络，这台机器会直接死掉。这是因为即使是现在最先进的 IDE 和机器，2000 万行代码也已经超过极限了。

事实上，我试了好几个星期都没能把自己那 50 万行代码导入 Eclipse，完成索引。一动它就假死，然后就再也动不了了（就算留过夜也没用）。处理 2000 万行代码更是妄想。

把代码从本机搬到服务器集群上或许能缓解一些。但说到底这其实是理念，而

不是技术上的问题：只要 IDE 用户继续否认问题的存在，它就不可能得以解决。

这就好比前面提到的俄罗斯方块，想象一下你可以通过某种工具管理巨大的屏幕，方块已经叠了几百层。在这种情况下，堆砌方块并非问题所在，相对的，重点也不在消除方块上面。这是理念的问题：他们没有意识到自己玩的方法压根就是错的。

第2点问题是 Java 系的 IDE 本质上是在搬石头砸自己的脚。这是语言本身的问题：每块“方块”的形状是由语言的静态类型系统决定的。Java 的方块不支持代码消除，因为 Java 的静态类型系统没有压缩的功能——没有宏，没有 eval，没有声明式数据结构，没有模板，没有任何可以消灭复制粘贴代码的机制，在 Java 程序员的眼里，这些东西是“回避不了”的，但事实上动态语言压根就不存在这种问题。

再说砸脚的问题，动态特性使得 IDE 难以施展管理静态代码的威力。IDE 和动态代码特性很难契合，所以 IDE 又反过来对那些鼓励使用需要……IDE 的语言起到了推波助澜的作用。

Java 程序员从某种程度上来说并不是不理解这个问题。比如，在 Java 里很流行的反射机制（让你可以凭空创造方法名，然后通过它来调用那些方法）会让 IDE 连重命名方法这样基本的重构都做不了。可是由于静态和动态的阵营不同，Java 众人会指着动态语言哇哇叫，抱怨（有些）自动化重构没办法做，而事实上 Java 自己也不过半斤八两罢了——也就是说，它们都只能做到一部分而已。在使用动态机制的时候，重构在一定程度上必定是有“缺失”的，这个和用 Java 还是其他语言并没有关系。重构不是万金油，特别是在发布公开 API 的情况下：这就是 Java 提供了废除（depreciation）机制的原因。要修改每个人机器上的方法名是不现实的。可是 Java 中却还是不断地鼓吹错误的理念，说什么自动化重构适用于他们“全部”的代码类的鬼话。

我打赌你现在肯定和我一样高兴不用和 Java 程序员打交道了！因为你已经看到了他们极端荒谬的一面，所以显然他们的反应也不太可能有多理性。

合理的代码量

所谓理性的反应应该是后退一步，放下手上的工作，仔细思考这样一个问题：“如果不`Java`，我该用什么？”

我4年前就这样问过自己。我暂停游戏的开发，将游戏转成维护模式。我希望把它重写一遍，在保持功能不变的情况下，把规模缩减到10万到15万行代码。

结果半年后我意识到要达到这个目标，用`Java`是不行的。即便算上`Java 5`里新添加的特性也不够，哪怕再加上`Java 7`承诺的东西也一样于事无补（他们还是加了不少很酷的东西，比如一个真正好用的闭包，`Java`社区一直非常抗拒这个特性）。

`Java`看来是指望不上了。但是我在`Java`虚拟机上已经投入了太多，就好像微软在`.NET`上的投资一样。我的意思是，这些东西当初在营销传单上看起来很美，别问我原因，我也说不出个所以然来。在我玩过各种解释器和原生代码编译器后，它们就更让人觉得有道理了。不过在那方面我也有不少牢骚要发，咱们改天再聊。

那么既然今天大家都认同虚拟机是个好东西，而且我也承认我的游戏已经和JVM紧紧地绑在了一起——不光是各种类库、监控工具，还有很多微妙的架构设计，如线程模型和内存模型等，因此要解决代码量膨胀的唯一合理的方案就是换用另一种JVM语言。

JVM语言的一大优势就是对`Java`程序员来说，它们的学习曲线比较平缓，因为所有的类库、监控工具、架构设计的决策都是现成的。而缺点是大多数`Java`程序员都是X语言程序员，我前面也说过，X语言程序员在团队里是不受欢迎的。

既然你不是那种活500年也要誓死穿喇叭裤的人，那么你肯定也愿意尝试不同的语言。非常好！

3年前，我开始研究有希望（精简代码）继承`Java`衣钵的JVM语言。过程并不顺利，结果也远远不能令人满意。就算在3年后的今天，我仍然觉得差了一两年的火候。（6年后，作者注解：仍然差得远！）

不过喧闹过后，我现在已经变得很有耐心了，因为我知道自己比那些还在折腾

几百万行代码的 Java 死忠领先了差不多两年。等到他们不得不在问题和需求的 ppt 里再加一页的时候，我说不定能为他们找到一条出路。

同时，我还希望自己有时间用这门语言重写我的游戏，在保证不削弱功能的前提下，把代码量从 50 万压缩到 15 万左右（当然至少还要再加上 5 万行左右的单元测试）。

下一个 Java

那么究竟哪门 JVM 语言有望成为下一个 Java 呢？

假如你的标准是纯粹的精简代码，那么最有希望的是一个 Lisp 方言：Common Lisp 或者 Scheme。

现在已经有一些非常不错的 JVM 实现了。我都用过。不过可惜的是，要作为 Java 的替代品，这门 JVM 语言必须做直接替换（否则移植的工作量会非常大），而这些 Lisp/Scheme 实现似乎都没有把这一条看得很重要。

此外，你会遭人唾弃的。哪怕只是建议在公司里采用 Lisp 或者 Scheme 方言的可能性，平时没有那种习惯的人都会用口水淹死你，就像是动物园里的骆驼那样。

所以 Lisp 和 Scheme 只能排除。我们必须牺牲一点简洁性，换取比较能为大众主流接受的语法形式。

理论上 Perl 6 也是候选人之一，只要 Parrot 那帮人真的能把东西搞出来。可惜这帮家伙比我还慢性子，你懂的。Perl 6 的设计其实相当不俗——2001 年的时候确实很惊艳。只不过这种感觉在 5 年后被消磨殆尽。而且 Perl 6 也不太可能出现在 JVM 上。它非常依赖一些强大的 Parrot 特性，JVM 永远也不会支持它们。（我本来想说甚至连 Parrot 也不太可能会支持它们，后来觉得这么说太寒碜人了。）

最有可能成为新一代 Java 的语言应该是一门已经有一定用户群，且在 JVM 上实现得不错的语言。它背后要有专门的开发团队以及出色的营销部门。

这样我们的选择范围一下就从 200 多门语言缩小到了 3 到 4 门：JRuby、Groovy、Rhino（JavaScript），Jython 勉强也算一个，要是它还能活过来的话。

这些语言（包括 Perl 6）都有能力将一个组织良好的 50 万行 Java 代码压缩 50%~75%。实际效果究竟如何还有待检验，所以我打算自己动手试试看。

首先是 Groovy，用下来的感觉是想法不错，但是语言本身很丑陋。它想当 Ruby，但是没有 Ruby 的优雅（或者说 Python）。它出现也有一段时间了，看起来也不怎么流行，所以我把它排除了。（这是永久性的排除——我不会再多看它一眼。而且 Groovy 的实现有 bug，让我失望透顶。）

我很喜欢 Ruby 和 Python，但是 3 年前评估的时候，它们在 JVM 上的版本都还不及格。JRuby 最近的进步很大。要不是和我一起工作的那些家伙拼命反对 Ruby 的话，或许就会选它了，我真心希望它的实现有一天能“接近”Java 的速度。

不过在这之前，我选择了 Rhino。我打算和 Rhino 的开发团队合作，帮助他们尽快实现 EcmaScript 第 4 版的标准。我相信 ES4 版的 JavaScript 在表达能力和组织管理大规模代码方面已经基本上不输 Ruby 和 Python 了。任何有欠缺的地方，只要考虑到它有可选的类型标注，就什么都无所谓了。而且我觉得对喜欢花括号（就是那些现在在用 C++、Java、C#、JavaScript，或者 Perl）的人来说，JavaScript（特别是具备 ES4 以后）比 Ruby 和 Python 更容易令人接受。这些人的数量可是非常庞大的，这可是大实话。（作者注：ES4 被标准委员会给残忍地扼杀了。实在令人惋惜。）

我不奢望今天这篇小小的牢骚能说服任何人同意我对代码量的看法。我知道有相当一些大牛（例如，比尔·盖兹、大卫·托马斯、马丁·福勒、詹姆斯·邓肯·戴维森）都各自得出了相同的结论：那就是，代码最怕的就是体积剧烈膨胀。这完全都是血的教训。

我也不会期望这样痛苦的事情发生在 Java 程序员身上，其实，它已经发生了。他们只是学会了麻痹自己，假装这对他们无害而已。

但是对于你，青春热情的高中生或者大学生，如果你还憧憬将来有一天成为优秀程序员的话，希望我的文章能帮你在未来几年的编程摸索中开拓视野。

如果你准备换语言，很好，Mozilla Rhino 绝对是你的首选。它现在已经很出色

了，未来一定会更好。另外，我真心希望 JRuby、Jython 等语言也能迎头赶上。你现在不妨试试这些语言，关注一下它们的进展。

你的代码也会感谢你的。

作者手记：反对反宣传

延续我一贯拿各种编程语言开涮的传统，这次我吐槽的对象是 Python。吐槽的焦点不是语言本身，虽然在某种程度上来说它的内部实现有点恐怖，但是从外面看起来还是相当漂亮的。

不过我还是好好嘲笑了一番 Python 社区，因为我觉得这帮人当时简直就是在给自己挖坑，而且很快就会因为挖得太深结果把自己埋了。

他们后来变得越来越好。对此我不敢居功……但是发表文章几周以后，Python 论坛上的的确确展开了一场有关 Python 营销问题的讨论，而且有人引用了我的文章。随后他们就对那个破破烂烂的官网进行了大革新（我文章里好多链接都因此失效了，不过那不是重点）。

只要天时地利人和，就算是小小牢骚也能激起千层浪来。例如，这篇文章就有点刺激到他们了。管他的，反正能引起注意我就达到目的了。

现在的 Python 已今非昔比。尽管一些 Python 众会想要否认这一点，声称自己对新人非常热情。但是自从有了网页快照这种东西后，妄图重写历史就是很难的，只要有心就可以轻易戳破他们的谎言。不过无论如何，至少他们端正了自己的态度，这一点还是值得鼓励的。我希望 Lisp 众也能正视这一点。

反对反宣传

声明：若本文激怒到你，请翻看下一篇“斑比遇上哥斯拉”。我会以更温和详尽的方式对我的一些过激评论做出解释。希望能让你明白，我不是故意找 Python 的茬。

大家都在谈论布鲁斯·埃克尔的那篇“反宣传”的文章^①。我希望他能继续保持清醒。

埃克尔的文章的重点是那种大肆宣扬的激情其实削弱了 Ruby 阵营的声音，搅浑了真正的讨论。那些讨厌的脑残粉只知道无休止地把焦点放在 Ruby 有多酷以及对它的喜爱上面，却不知别人需要的其实是在 Python 和 Ruby 之间客观中立，有礼有节，一切以标准为基础，深入细致的学术讨论，这样我们才能做出理性的判断。如果说最后是最优秀的语言胜出，那么它一定是 Python。

但是 Python 众从来没有真正理解营销是怎么回事。

我很吃惊地发现居然还有人对这段历史那么陌生，毕竟我们之前已经谈论过很多次啦。既然如此，我们就再复习一下语言演化基础论。

首先，再垃圾的语言和技术也一样有机会赢。甚至赢面可能还会大一点，因为改正起来会更快。Java 击败了 Smalltalk, C++ 击败了 Objective-C, Perl 击败了 Python, VHS 击败了 Beta，诸如此类。并不是说一项技术（特别是编程语言）比较优秀，它就一定会胜出。营销才是关键。追求公平竞争只会导致你的语言无人问津。

在宣传机器上砸钱不失为一种好方法，Sun 和 IBM 就是这样宣传 Java 的，Borland 以前也是这样宣传 Turbo Pascal 的。非常烧钱，但是很有效。更常见的手法是通过一小群有影响力的推手，通过口耳相传的方式逐渐传播到工业界，那里有很多被压榨到不行的程序员，总是在想方设法找乐子，他们会开始密谋在工作中使用这门“禁忌的”新语言。很快，人事经理就会开始在简历上寻找这门语言的关键字了，这又会进一步推动相关图书的销售，于是不知不觉之间，连锁反应形成了。

Perl 就是一个典型的例子——它是怎么打败 Python 的？它们出现的时间差不多。可能 Perl 稍长几岁，但是差距也不至于那么大。Perl 的用户群差不多是 Python 的 10 倍，并且这一优势保持了 10 年之久。它是怎么做到的？这完全要归功于拉里·沃尔的天才营销，以及得到了奥莱利这样大的出版社的支持。

① 译注：<http://www.artima.com/weblogs/viewpost.jsp?thread=141312>。

《Perl 语言编程》可以说是语言书里的经典之作：它很口语化，让你觉得很好读，同时也很风趣，似乎 Perl 已经有很长的历史，而你在读的只是最新版。它的营销手段是双重的：首先是把自己塑造成一个成熟稳重，值得信赖的品牌（就好像巴诺书店明明是不知道从哪里冒出来的，结果非要宣称自己 1897 年就创立了一样），然后再辅以新奇和与众不同的感觉。拉里坚持了这个策略好多年。Perl 各种丑陋的缺点和难懂的复杂性都被包装成“高级”的技巧。Perl 的世界里充满了口号、嬉皮式的段子，各种牛，不过最重要的是，它很好玩。没错，Perl 想要给人的印象就是很好玩。

那么 Python 呢？它嬉皮吗？滑稽吗？好玩吗？都说不上。Python 社区非常严肃认真，很稳重，也很专业，不过他们不怎么关心好不好玩这个问题，而这对推广一门语言来说是非常重要的。（作者注：原本这句话是这么写的，“但是他们就和税务员一样乏味”。结果他们就发飙了，气得要命。而其他人都觉得这句话很好笑，而且很准确。）

这个问题三天三夜也说不完，随便举个例子好了，比如在命令行下面输入“python”就可以启动一个互交的解释器。如果你在提示符下输入“quit”，它会显示“Use Ctrl-D (i.e., EOF) to exit.”的字样。（译者注：就是输入退出，解释器会提示你用 Ctrl-D 来退出。）

这实在不是什么很好的体验，你觉得呢？它明明知道你想要退出，却还是要求你输入 EOF，天知道那是什么意思。（好吧，你我都知道 EOF 的意思，但是初学者怎么办？）它为什么不直接退出就好了呢？

10 年来，只要你把这个疑问拿到 Python 新闻组里去问，就会有人要你去读 FAQ。或者，他们会解释给你听，告诉你用“quit”来退出是违反 REPL 语义的做法，这在英语里不是先验的（译者注：就是说事先是不知道的，得用过的人才知道可以这样），而 Ctrl-D 是绝大多数终端模拟器上都广泛接受的 EOF 字符，当然，win32 和 VAX 平台除外。交互 shell 设计得非常漂亮，解释器可以认为一切输入都是来自文件或者类似的流，等等，综上所述，这是正确的行为，证明完毕。

他们完全无视了“quit”就应该退出 shell 这么一个显而易见的事实，语义在这里应该让位。可是他们不在乎，因为“正确的事情”高于一切，甚至可以牺牲用户体验。有一句谚语说得好：只见树木，不见森林。

当然啦，从 Perl shell 里退出来也不见得简单多少。但如果你把同样的问题拿到邮件列表或是新闻组里去问，Perl 众会非常热情地伸出援助之手，不但会告诉你怎么做，还会花上一个小时的时间告诉你一键退出是一件多么酷的事情，这样在其他 Unix 命令下它也一样有效，他们甚至可能还会告诉你怎么直接修改 Perl 的二进制文件，好让“quit”退出 shell。两者高下立判：尽管两个 shell 都有这个烂问题，然而 Python 众只会说教，而 Perl 众会出手相助。

迂腐就是 Python 界的代名词。你必须全盘接受它的那一套。要是你不喜欢，那么错的人是你。想要寻求改变，你可以去提交 PEP，就是 Java 那套冷若冰山的 JSR 流程的 Python 版。Python 的 FAQ 花了巨大的篇幅来为一系列明明就是垃圾的语言特性正名。显然如果这些特性不垃圾的话，也不会经常被问到了，但是他们不仅不大大方承认“我们正在计划修复方案”，他们居然反过来指责别人没有正确理解问题所在。每当一些特性真的被修复的时候（比如词法的作用域），他们又会宣称是因为大家搞浑了，所以才顺应大流作出修改。注意，骂谁也不能骂 Python。

相反，骂 Ruby 骂得最凶的人可能就是 Matz 自己了。他在自己的演讲“为什么 Ruby 很烂”里，自陈了 Ruby 的各种问题，当时看得我汗都下来了。（译注：见 <http://www.rubyist.net/~matz/slides/rc2003/mgp00001.html>。）但是不可否认，任何语言都有缺点。相比之下，我更喜欢 Ruby 众的坦诚，Python 那种一味指责别人，回避问题，过分地自我标榜的行为令人感到恶心。（作者注：这些链接现在都失效了，你觉得是巧合吗？）（译注：带上版本号的话，还是可以找到的，见 <http://docs.python.org/2/faq/design.html>。）

在语言特性方面，Perl 和 Python 的理念可谓截然不同：只要有人提出要求，拉里就会把它添加进来。时间一长，Perl 就慢慢从一个厨房水槽变成了各种语言的垃圾填埋场。但是他们从来不会说：“不好意思，这个 Perl 做不了。”毕竟那不利于语言推广。

老实说，现在的 Perl 非常丑陋，糟粕太多，多到他们自己也承认无法收拾，Perl 6 是彻彻底底的重写。假如 Perl 当初能像 Ruby 一样打好根基，它就不用为了迎合那些广告语而过分扭曲自己了，或许今天仍然能保持竞争力。可惜一切都为时已晚。拉里的营销魔法已经失效了，人们终于反应过来，原来那些没用的玩具（引用、上下文、typeglob、绑定等）只有在用 Perl 最快的前提下才有意思。回想一下，有趣的部分是什么？还不就是在把活干完的同时，还能跟朋友炫耀一下你写的软件有多酷嘛。Perl 的那些稀奇古怪的特性里只有一半有这种效果。

因此我们现在一无所有了。Perl 已经无以为继，身上背负了太多不必要的特性。Java 也是强弩之末，太官僚了。大家都开始慢慢觉得它们对编写优秀软件来说其实是个障碍。这种空虚将会由……没错，营销推广所弥补。所有人（包括招聘主管）都会迅速察觉到风向的变化，不知不觉之间就会出现麦尔坎·葛拉威尔所描绘的引爆趋势。

我们现在正处在这么一个引爆趋势之下。其实说不定趋势已经形成了，Ruby 正在逐渐成为领头羊，这种编程语言出现在各种简历和书架上，就像今天的 Java 一样。布鲁斯·忒特的书说的就是这个。你可以选择左右逢源，比如像埃克儿那样（译注：布鲁斯·埃克儿，《Think in Java》的作者）。或者选一种你觉得更有希望赢的语言，为之摇旗呐喊，反正不要去抱怨其他语言的支持者（在谈论语言的时候）不公平就行了。

那么 Python 有希望成为下一个超级语言吗？不是没有可能。它的确是一门很棒的语言（当然这并非真正关键的因素）。一门语言成功与否，完全取决于今天的执行力。不是一年或是几个月，就是今天——而且它还必须能认识到自己的落后和不足的地方。Ruby 谈不上有多出色，但它现在手上有杀手级应用。Rails 对推动 Ruby 起到了巨大的作用。在 Web 框架方面，Python 可谓输得一败涂地。号称要和 Rails 竞争的 Python 框架至少有 5 个：Pylons、Django、TurboGears、Zope，还有 Subway。其实 3 个（甚至 4 个）都嫌多啊。从营销的角度来讲，到底哪个比较优秀其实根本不重要，重要的是 Python 社区应该选中其中一个，然后全力鼓吹；否则每个框架都只能分到 20%，结果谁都没有实力跟上 Rails 创新

的步伐。

Web 框架之争或许已经结束，但是战争才刚刚开始。Python 如果不想输，就必须认真面对推广的问题，它得赶紧抓一些有影响力的写手弄出几本鼓吹的书来。他们必须放弃现在这种反宣传的策略，这一点毋庸置疑。不好意思，布鲁斯。学术讨论是无法吸引新用户的。宗教式的言论才是关键。你得让人觉得你很有趣，让别人羡慕你。

我估计 Python 和 Java 的拥趸这次还是会捡了芝麻丢了西瓜。他们会逐条反驳我的观点，然后宣称证明了我是错的：营销推广其实不那么重要。或者他们会说：“得了吧，哪有什么战争啊。我们自然有我们的生存空间。”然后他们会继续琢磨为什么巴诺书店里全是 Ruby 的书。

不过我才懒得关心呢，因为 Ruby 实在是太酷了。我有提到过在 Ruby 的 shell 里输入“quit”就可以退出吗？它真的可以，而且用 Ctrl-D 也行。Ruby 就是最棒的。那么 Rails 呢？说真的，你要是错过的话就太可惜了。绝对是一流的。Ruby 的老爹绝对胜过 Python 的老爹。要是你不同意的话，看看 why 先生的《Why’s Poignant Guide》就明白了。（译注：著名的漫画教程，似乎俚语太多翻译困难，所以没有见到过中文版。）Ruby 实在是太好玩了——基本上现在这是我唯一想要用的语言了。而且学起来也非常方便。这可不是在吹嘘。真正酷的东西是装不出来的。

作者手记：斑比和哥斯拉

这是一篇后续，我的“反对反宣传”发表几天后，激起了 Python 社区滔天的反击，所以我又写了一篇。

这篇文章的语调要委婉得多，感觉还是相当有理有节的。亲眼目睹语言消亡是一件很沉重的事情，我觉得每个人都能从中学到点什么。

在这篇文章发表之后，Python 社区基本上就不再和我计较了，对此我一直心怀感激。

斑比和哥斯拉

葛雷乔伊写道：

“史蒂夫，我觉得还是不要无端为 Ruby 程序员挑起一些他们根本无意挑起的战火比较好。”

“两边都不需要狂热的脑残粉。”

“这篇文章实在有失偏颇，打压 Python 的意图太过明显。很不厚道。”

“Python 程序员并不都是税务员。”

* * * * *

好吧，好吧，我明白那篇“反对反宣传”写得有问题。假如我是一条狗的话，今年已经有 600 岁了，有时候写起文章来会忘记交代背景。所以我打算重新解释一下昨天的思路。如果还是说不清楚的话也没关系——反正我写的是技术博客。

我会顺便聊聊其他语言，不过最后还是会回到 Ruby 上来。别走开，广告之后马上回来。

我发现我们不太谈论一些对程序员产生直接影响的问题。比如政治，如果不是狂热疯子，这个话题其实挺敏感的。不过今天我就打算聊一聊这个话题。为什么？

首先，这里是我每天的日程安排。和你们一样，我是一个程序员，我喜欢写程序。而且最好是用我最喜欢的语言来写，那就是 Ruby——不过它的优势并不大，因为我还喜欢很多其他语言，如 Python、Scheme、Lisp、D 语言、C 语言（但 C++ 除外）等。此外，还很很多我觉得还不错的语言，以及一小部分我不太喜欢的语言。

巧合的是，Python 在我最爱的语言榜上排名第二（好吧，和 Scheme 并列第二）。

没错，你没看错。我非常喜欢 Python，Guido 非常了不起。Matz 也是。他们

都是大牛。

我每天的安排都非常简单：我希望不管在家里还是在公司，都能用优秀的编程语言来写代码。

但是出于一些复杂而又痛苦的原因，其实说起来简单做起来难。下面我们就来稍微聊一下，希望能化解一点我的那篇不怎么厚道的文章所引起的怨念。

美丽语言之死

我亲眼目睹了 Smalltalk 的消亡。

其实当时我在 Smalltalk 上并没有投入太多精力，只是用它写过一点程序而已。Smalltalk 曾经（现在依然）是一门非常出色的语言。可惜我学会之后它就不行了。

或许有些 Smalltalk 的铁杆会指着 Squeak 等 Smalltalk 的变体，宣称 Smalltalk 还没死。但这才是重点。我打赌你就没怎么关心过 Smalltalk 吧。它根本就不在你的视野范围内。没人觉得还有学习它的必要。所以前面我说“我亲眼目睹了 Smalltalk 的消亡”时，你只是觉得我在和你聊古代史罢了。

可是对很多真正喜爱 Smalltalk，以及仰赖它在商业上有所建树的人来说，Smalltalk 的陨落却是一个非常沉重的话题。绝对不是什么无聊的历史。至今这些伤痕仍未抚平，他们依然在幻想有一天能看到 Smalltalk 恢复往日荣光。

虽然你可能感受不到他们的这种痛苦，但是这和我们接下来要讨论的核心却非常接近。正是因为会伤害到一部分人的感情，所以这种话题尤其敏感。谈论 Smalltalk 的时候怎么说都可以，特别是你不了解这门语言的时候。可能你只是随便瞄了一眼，然后评论道“看起来有点笨重”，结果祸从口出，得罪人了。

其实这个话题很难讨论，因为有的人光是提到 Smalltalk 就会发飙。不管是哪方面，只要一涉及这个话题，他们就会上蹿下跳。

不论 Smalltalk 是真死了，还是只是受伤的熊在冬眠，我想至少 Smalltalk 对当今的编程界不存在直接影响这一点是毋庸置疑的。

当然，Smalltalk 的间接影响还是很多的——比如，Ruby 就从 Smalltalk 那里借

鉴了很多东西；所有的面向对象语言都在一定程度上借鉴了它，Ruby 尤其突出。但是随便哪家电脑书店里大概都不会有超过两本的 Smalltalk 的书了吧。假如你要找一份 Smalltalk 程序员的工作，恐怕腿都要跑断了，就算找得到，选择的余地也非常小。Smalltalk 的活动领域已经十分有限了，至少目前如此。

那么凶手是谁？我读了很多分析，也和很多关键人物深入讨论过这个问题。似乎大家都一致认为是 Java 干掉了 Smalltalk，而且干得相当漂亮。不知道你有没有看过那个著名的短片《斑比遇上哥斯拉》？基本上那种感觉是一样的，当然啦，事实上整个过程持续了好多年。

老实说，Smalltalk 和 Java 的区别真不大。它采用了一种颇不寻常的多合一的模型，它就好像是你的操作系统，语言本身，IDE，以及应用程序环境全部都在同一个程序里。虽然现在没人觉得这是个好主意，但滑稽的是，JVM 其实不也半斤八两嘛。Smalltalk 不是免费的，用户要为之付钱。Java 也一样，不过最终用户可以免费使用。它们曾经非常相似，结果一个却远比另一个流行。

或许有人会争辩就算没有 Java，Smalltalk 也一样会死，不过我想大多数人都同意 Java 在其中扮演了关键角色，而且动作还不小。这可是价值数百万美元的赌注。Smalltalk 的供应商主要有两家，外加一大堆马上要失业的 Smalltalk 程序员，愤怒地在那里抵制 Java，这门“显而易见的”垃圾语言，无耻地偷走了原本属于 Smalltalk 的市场。

最终一切归于平静，对大多数人来说，Smalltalk 大概只是一门小众的教学语言，从来也没流行过吧。

我觉得 Java 崛起，进而挤掉 Smalltalk 的大部分功劳要归功于成功的营销。虽然这不是唯一的因素。从很多方面来讲，天时也是一个重要的因素。语法和静态类型也是，Java 为了吸引那些失望的 C++ 程序员显然是下了功夫的，说实话这是相当聪明的策略。Java 本身的一些天才创新当然也起了不少作用。

但正是出色的营销将这一切联系在一起，帮助 Sun 建立起一个拥有数百万 Java 程序员的全球社区。

Java 并没有提出什么新鲜的东西，它有的 Smalltalk 早就有了。（这种论点好像在哪里听过？）

爱与利益

在我看来，要激起语言之争，通常有两个要素。

首先，绝大多数程序员都不喜欢学习新语言。这一点很奇怪，但是像我这样真正掌握多种语言的人真的很少见，大概只占整个程序员人群的 5%~10% 吧。大多数人都选择掌握一种语言，然后和它厮守一辈子。

其次就是经济因素，最终一切还是向钱看。公司要赚钱赢利，程序员要养家糊口。俗话说，时间就是金钱，商场如战场，世界是由利益（或者爱）来驱动的，而在恋爱和战争中的人，是什么事情都干得出来的。

程序员会爱上自己使用的语言，所以你面对的是这个世界上最强大的两股力量：爱和利益，外加一点恐惧和懒惰。这样解释人们为什么会为语言起争执是不是就容易理解一点？

好吧……勉强是吧。说起来，似乎人们可以选择任何让他们称心如意的语言。但其实你只能从 C++、Java、Perl，或是任何你的雇主指定的“官方”语言里选一门自己比较喜欢的。

大多数科技公司可供选择的官方语言并不多，并且禁止你使用其他语言。于是这就引发了一些奇怪的现象。

事实上大多数公司根本不在乎你用什么编程语言——只要能出活就行。这种语言限制其实是工程师自己折腾出来的。当然他们自己并不会限制自己，而是公司里比较资深的工程师做出的决定。虽然我听过这种理由，但这依然让我觉得很莫名其妙。

另一件怪事就是绝大多数公司会用到 15~40 种编程语言，可是公开承认的只有 2~3 种。他们会声称自己只用 Java 或者 C++，但实际上各种内部工具、数据库、构建系统里大量充斥着脚本、awk、PHP、Perl、JavaScript、Tcl、emacs-lisp、vim

脚本、excel 宏、PL/SQL 等语言。或许唯一例外的就是那些基于 Windows 的科技公司吧。

最后一件怪事就是程序员新入职以后通常都不得不(至少)再去学一门新语言,似乎也没见过学不下去的情况。程序员往往觉得学习新语言是很麻烦的事情。可碰到换工作这种情况,他们学习的速度却又快得惊人。毕竟程序员一般都不是笨蛋。

当你想要将自己最喜欢的语言引入公司时,那些“老古董”们抵制的力度会让你大吃一惊。这里的“老”指的是“先来的”的意思,和年龄无关,那些刚刚成功创业的 23 岁计算机专业的毕业生也一样可以是个“老古董”。

那些家伙的论调我听了都快 20 年了。别用 C++, 它太慢了(我的第一家公司)。别用 Java, 它太慢了(我的第二和第三家公司)。别用 Python, 它不但速度慢,而且那个空白符的问题很讨厌(所有公司,特别是最近)。别用 Ruby, 它很奇葩(90%的公司)。语言多样性不是什么好事。要是别人半夜三更要调试你的程序,结果发现他们不懂那门语言怎么办?这种散播恐怖的事情每家公司里都有,包括那些从来都不会在半夜三更上班的公司。不要用其他语言。我们不需要那些技术。我们不信任那些语言。我们已经在 Fortran、Cobol、C++、Java, 或者别的什么语言上投入很多了。不行,免谈,想也别想。

说“不行”的那些人全是工程师。只要你能搞出漂亮受欢迎的东西来,CEO 都会欣赏。哪怕你用 Intercal^①来写她也不会在意的,只要行之有效,外加你的团队能让它一直运行正常就好了。那么为什么工程师会那么在意呢?天知道。我猜大概是自尊在作祟吧——他们自觉是顶级 Java 程序员、Perl 大牛,或是什么 Python 名流,所以要利用这种自以为是的影响力来影响别人的技术决策。不管怎么样吧,这在工作中是一股很强的力量,我们在互联网上看到的语言之争中往往都能见到它的身影。

毕竟就算你最爱的语言受众够大,问题也依然存在啊——只不过你感受不到而已!

① 译注,一种冷门的语言。

哥斯拉和斑比回归

我在 Geoworks 干了 5 年，基本上用的都是汇编，这大概就是我对任何语言都不抗拒的原因吧。之后又用了一段时间的 C/C++，然后接下来的 7 年时间一直都在用 Java。

在接触 Java 两三年后，我偶然发现了 Jython，Python 在 JVM 上的移植非常精致小巧。在这之前我都是用 Perl 来写脚本和各种辅助工作的。尽管 Jython 还很新，但是之前我还从没想过或许真的存在这样一门能适用于大多数编程工作的语言。Java 和 Perl 各有优缺点，所以两种语言我都离不开。

在《反对反宣传》那篇文章里我小议了一下 Perl 的营销手段。说实话的确是一流的，搞得我曾经一度真的觉得自己很喜欢 Perl。那种宣传攻势实在是太有影响力了，当时的我完全就被洗脑了。

Jython 则是一股清风，我甚至开始考虑是不是要整个迁移过去，放弃 Java 和 Perl。但是后来它的开发工作停顿了下来，所以很自然地我就捡起了 Python。我迷恋了 Python 至少有 3 年的时间，不过由于之前在 JVM 上投入得太多，所以大多数时候我只能与 Jython 为伍。尽管它只是一门老旧的，基本上处于无人维护状态的 Python 语言，但我还是用得很开心。

这些年来我一直在琢磨，Python 到底为什么不像 Perl 那么流行呢？毕竟乍一看 Python 比 Perl 强多了。当然这是我个人的看法，而且我还挺怀念 Perl 里面的一些东西，所以我并不是说 Python 的语言设计就是完美的，只不过它确实看起来很美。

那为什么它就是火不起来呢？我觉得主要还是败在营销上面——那些杀红了眼的 Perl 铁杆到处为自己的最爱摇旗呐喊，把用户一个一个拉到自己这边来。Perl 就像是病毒一样迅速传播开来，而与此同时的 Python 却后知后觉，行动迟缓，错过了成长壮大的良机。理查德·加布里尔更是早在他那篇著名的短文《论“差一点才更好”的兴起》^①里就已经指出，C 和 Unix 都是这样发家的。

在这里我想明确一点：尽管很多人跳出来说 Python “击败” 了 Perl，我却认为

① 译注：原文 <http://www.jwz.org/doc/worse-is-better.html>。

Python 已经一败涂地了。虽然 Python 确实在品质方面较优，但是对我来说，在商业领域的流行度才是最直观的指标，毕竟（还记得我每天的日程安排吗？）我的愿望是用最好的语言来写代码。刚入职就宣称自己只写 Python 的想法是不现实的，他们会直接灭了我。从这种意义上说，Python 确实输了。我是真的打心底希望它没有。因为和 Smalltalk 不同，我在 Python 上是花了很多心血的。

Python 曾经是我的最爱，我读过所有关于 Python 的书，写了大量的 Python 代码，长期在各个 Python 新闻组里潜水，接受熏陶。几年下来，我慢慢整理出一套为什么 Python 在商业上（暂时）不成功的理论。

文化

我明白没人喜欢这种话题，不过我还是打算稍微谈一点文化。文化是实实在在的东西。其重要性不亚于爱情和金钱。

法国巴黎的服务生的表现通常和日本东京的服务生的表现截然不同。这一点毋庸置疑，而且差异惊人。这是我在两座城市流连多年，吃过无数餐馆后得出的结论。

有一次我和朋友一起吃饭，他悄悄探过头来和我说：“我只是想要点盐，怎么服务生的眼神好像要杀人一样？”你猜我当时在哪个国家？

法国的服务生并无好坏之分，日本的也一样。他们的行为在各自文化里都是合理的。只不过两种文化截然不同罢了。每个国家的餐厅都有各自的潜规则，所以这里比较的仅仅是法国巴黎的服务生和日本东京的服务生而已。

说句冒天下之大不韪的话，我觉得法国巴黎的服务生太吓人了。朋友和我只是礼貌地坐下，点菜的时候犹豫了一点而已，结果他们说话就粗声粗气的，压迫感实在太强，吹胡子瞪眼地甩下盘子就走，粗鲁得让人觉得简直就是故意的。（作者注：最近我又去了一次，这次他们都非常热情。）

相反，我却亲眼见证了日本服务生为了满足那些在商务旅行中的醉汉所作出的努力，他们的敬业程度让我这个美国人都感到羞愧。如果要问世界级的服务水平是什么样的，来日本看看就明白了。

到这里我想大家已经清楚文化之间是有差异的了，而且这种差异会在与人交往

的过程中产生巨大的影响。

我想你也应该知道编程语言同样也有自己的次文化。Perl 的文化就和 Python 截然不同，而它们又都和 Ruby 风格迥异。

Python 的文化其实是有优势的。当初我是一见倾心，可是时间久了以后，我开始注意到 Python 社区里一些行为背后更深层次的东西，让我怀疑 Python 之所以没有一夜爆红是不是与之有关。当然，这只是我个人的看法，和他人无关。

在《反对反宣传》那篇文章里我已经讨论了一点那些现象，于是有些人（Ruby 程序员、Python 程序员、不明真相的群众）觉得我是在黑 Python，其实那是因为他们没有见到 Smalltalk 消亡，也没有我现在给你解释的语境。

事实上，我只是对 Python 无法企及 Perl 那样的市场成功感到很失望而已——如果你知道我在说什么的话，你就会明白这真的会有经济后果的，比如说它会影响到 Python 在业界的普及程度。让人沮丧的是，这似乎是个原本可以避免的问题：我认为 Python 界的一些所谓的被广泛接受的做法事实上伤害了市场对 Python 的接受程度。

我大可不厌其烦地一条一条地去证明我在《反对反宣传》里的论点，但是我们不妨只关注一点：喜欢给人扣“不正确”的帽子。这个习惯实在很讨人嫌，新人很容易就被吓跑了。这属于文化上的习惯，和巴黎拉丁区里的小饭馆服务生之间很普遍的那种吹胡子瞪眼，大声嗤之以鼻的习惯是一回事。

这样的例子很容易就能在网上搜到。比如，《Python Cookbook》范例 1.7 的最后讨论了一下 `attribute` 和 `item`，声称很多 Python 初学者，特别是那些有 JavaScript 背景的人，喜欢“自找麻烦”（幻想以统一的方式来访问）。^①

这种说辞实在是有点过分。程序员看不懂你的东西并不是什么大不了的事情，没必要揪着不放吧。如果只是因为他们希望有一种他们习惯的方式来解决这些问题，就要给他们打上“自找麻烦”的标签，不会让人觉得有点过于严厉

① 译注：这本书出到第 3 版了，这里指的是第 1 版。

了吗？

再举一个例子，《Jython Essentials》的第5章里有一段关于Python的类系统的讨论，它这样写道：“有时候人们会错误地觉得，在参数列表里必须显式写出实例是Python生搬硬套面向对象编程的证据。”

“错误地”？天哪，这个似乎是纯主观的看法，而不是什么可以分出对错的事实吧？观点看法怎么能是错误的呢？或许这又是所谓的文化吧。当一种文化喜欢给不同意见打上错误的标签的时候，那么任何意见都可能是错的。

Python FAQ里曾经有过这么一个条目（一两年前被删掉了），标题好像是“我可以对语言提出修改建议吗？”，下面的回答非常简单：“不可以。”我记不清它具体是怎么措辞的了，但是我记得很清楚当时那种震惊的感觉。那个条目挂在上面好多年，最近才被清理掉。

这些细节随处可见。阅读Python讨论或者文档的时候你未必注意得到。我注意到了是因为我刻意在寻找那些对Python感到厌倦最后放弃的人所写的文章和观点。我发现它们常常感到漠视，社区不欢迎它们，或是引用一些似乎原本无足轻重的小问题。但是有一点很重要：它们背后所显现的趋势。

这是全体Python程序员的错吗？当然不是。绝大多数Python程序员还是非常友好、热情、真诚、诚实的聪明人的。

但是文化这种东西不管你喜不喜欢，它潜移默化的影响是巨大的。如果初体验之下导致太频繁的文化冲击，那很容易就把潜在用户给吓跑了。

为什么你可以在很多公司里用Perl写程序，但是Python就相对很少？你当然可以有自己的见解。我已经给出了我的理由，哪怕它不是全部，至少也是因素之一。营销推广并不是有光鲜亮丽的横幅和扭来扭去的卡通吉祥物就够了的，一对一的交流也是它的一部分。

我真的不希望在10年后写下这样的话：“我是看着Python消亡的。”这个市场容纳5~8门主流语言是完全绰绰有余的。我想Ruby应该很快就能占据一席之地，老实说我也很希望看到Python的身影。

当然，我也必须承认改变文化是不可能的事情，文化形成后就不再改变了。我希望这个假设是错的，不过这是我今年夏天最终决定转向 Ruby 的关键因素之一，在可见的未来（大概 5~10 年）这会是我的主力语言。

Ruby

Ruby 的文化在全球范围内是我这么多年编程经验里最热情、最友善的。Ruby 本身也是一门非常棒的语言。这也是其他很多人的看法，一传十，十传百，结果却被酸葡萄们打上了“狂热脑残粉”的标签。对我来说，Ruby 实现了我对 Python 的期望，这促使我决心学习 Ruby，并最终把大部分的工作都搬了过来。

当然了，这两门语言和 Java 相比，在很多方面都还有所欠缺，比如各种工具、IDE、书籍、性能、多线程的稳定性等。在充分评估之后，我决定理性地赌一把，挑选一门我觉得更有希望的语言，这样不但我用得顺手，而且在工作中用它也不会遇到什么障碍。

Ruby 很容易学。我只花了几分钟时间，就可以和使用多年的语言一样熟练了。个人感觉非常好用。它并不完美，但什么语言完美呢？缺点并不是致命伤，而且 Matz 似乎打算在 Rite 里修正那些缺点。

我不敢说自己喜欢 Ruby 超过喜欢 Python 和 Scheme，但可以说至少不亚于其他语言。不过目前 Ruby 是我的最爱，因为我能看到它的发展轨迹，特别是有了 Rails 以后，更令我坚信 Ruby 会像 Java 那样获得成功。就像布鲁斯·忒特写《超越 Java》的时候预言的那样。詹姆斯·邓肯·戴维森、大卫·托马斯、马丁·福勒等很多比我聪明得多的人也都是这么认为的。他们看好的东西肯定有点价值吧？反正我信。

像 Java 那样在全球范围内掀起热潮是关键。如果 Ruby 被接纳的程度达不到那种市场规模的话，它是得不到它所需要的工具、稳定性，以及 CPAN 那样丰富的库的，那么和 Java、Perl 的竞争也就无从谈起。这是所有语言都要面临的问题，先有蛋还是先有鸡？而 Ruby 却有机会做到 Smalltalk、Python 等很多优秀语言（目前为止）都没有做到的事情。

我发现有些 Ruby 程序员担心被 Rails 抢走了风头。天哪，你们是怎么想的？抢就抢呗。这可是借东风的好机会啊。Java Applet 让 Java 有机会将自己呈现在上

百万程序员的面前，最终让 Java 平台占领了那些它做梦也没想到过的领域，一切都亏了这个所谓的“杀手级应用”Applet。

我们生活的世界充满了文化、经济、流行风潮的影响。它们对我们程序员的生活质量都能产生实质的作用。忽视它们是要付出代价的，想想那些被 Java 和 Perl 踩在脚下，几乎没人记得的语言吧。

我曾经非常看好 Python，就算到现在也仍然希望它能好好活下去，但是我觉得它真的很忽视营销。我也非常希望 Ruby 成功，所以当我听到像布鲁斯·埃克儿这样的名人在没有充分调查研究的情况下就对 Ruby 和努力推广 Ruby 的朋友妄下断论的时候，感到相当恼火。我希望 Python 社区也把注意力放在怎么推动 Python 成功上。这需要在社区文化上做一点调整，他们必须接受这样一个事实，宣传是生活中必不可少的一部分，也是在纷扰中杀出一条血路的必要条件。我相信只要愿意，他们是能做到的。

说了那么多，现在你们是不是觉得我的那篇“反对反宣传”稍微好理解一点了呢？不妨请再读一下吧。

如果还是不行，那我真的是无计可施了。既然无法让所有人满意，我也只好随他去了。

作者手记：程序员的数学

这又是一篇在 Hacker News 上掀起波澜的文章。这篇东西很刺激，它向所有人传递了一种正能量——除了老师们，他们对这篇文章还是颇有微词的。

这篇文章发表有好些年了，但我至今仍然觉得他们应该考虑改革 K-12 的数学教育。离散数学自有它的一席之地，忽视它的后果也是显而易见的。

此外，我写的所有文章里收到的最好的评论之一就来自杰米·扎温斯基对这篇文章的评论。他显然很喜欢它，还把链接贴在自己的博客 www.jwz.org/blog 上，这个博客可是世界闻名的。后来他在更新博客的时候这样写道：

更新：我贴这个链接的原因是我觉得他对数学教育，以及数学教育和编程的关系有非常有趣的见解，而你们这些家伙却硬把它歪曲成某种无知当有趣的公投（dick-waving referendum），“我每天都会手写算除法”！好吧，算你狠。非常感谢你和我们分享。感谢你喜欢这篇文章，没有在鸡蛋里挑骨头。

我讨厌你们，都给我闭嘴。

4年后的今天，杰米·扎温斯基仍然在“dick-waving referendum”的搜索结果里排名第一。回首我人生的各种成就，没有一条能厉害到接近那种程度。

程序员的数学

15个月之前我读了一本约翰·冯·诺伊曼的传记，之后我就一直在努力把过去丢掉的数学再捡起来。我读了一大堆数学书，还有一堆更高的没读。渐渐地，我的思路开始清晰起来。

听我慢慢讲。

传统观念没用了

首先，程序员觉得数学没用。我经常听到这种观点，似乎也没什么人持反对意见。连那些数学专业出身的程序员都跟我说，他们很少用到数学！他们觉得设计模式、面向对象方法论、软件工具、接口设计这些东西更有用。

你知道吗？他们说得一点也没错。合格的专业程序员其实并不需要掌握太多的数学知识。

但是别忘了，你连编程都不一定要懂。很多职业程序员其实很清楚自己不擅长编程，然而他们还是能贡献自己的力量。

如果你突然觉得自己知识不够用了，周围的人都比你强，这时候你怎么办？可能你发现自己擅长项目管理，或是很会和人打交道，亦或是会UI设计、技术写作、系统管理等很多其他“程序员”未必擅长的重要工作。你可以着手填补这些空白（毕

竟工作是永远做不完的），只要找到真正适合自己的事情，你就会全心投入。

说难听点，只要你能活下去，什么都不懂又怎么样。

所以我才会同意他们的观点：就算不懂数学，你的生活也会好好的。

但是最近我却了解到一些令人惊奇的事情。

1. 懂编程的人学数学会更容易。如果你是个还算不错的程序员的话，数学简直太好学了。

2. 学校里教数学的方法都是错的。错得离谱。只要方法得当，自学的速度会快得多，学会以后也不容易忘。懂数学对程序员来说是如虎添翼。

3. 哪怕只了解一点点相关领域里的数学，就能让你写出非常有趣的程序来，反之，这些程序会很难写。换句话说，数学知识是可以在空闲时间里慢慢累积的。

4. 就算是最伟大的数学家也不可能通晓数学。这是一个不断发展的学科，人们总是会发明新的形式化方法来解决问题。和编程一样，任何数学问题总是有多个解的。你大可以挑选自己最喜欢的那个。

5. 数学……呃，请不要跟别人说这是我说的，不然就再也不会有人找我出去玩了。但是数学……我还是悄悄说吧：数学其实还是很好玩的。

学了（又忘）的数学

这里列出了我在学校里学过的数学，大致上想得起来的如下。

- 小学：数字、计数、算术、初等代数（伪装成小故事的数学题）。
- 中学：代数、几何、高等代数、三角函数、初等微积分（圆锥曲线和极限）。
- 大学：微分和积分、微分方程、线性代数、概率论和统计、离散数学。

这里的中学课程到底是谁决定的？基本上全美的所有高中都会教这些。我估计其他国家也差不多，只不过他们的学生在 9 岁的时候就已经学完这些内容了。（不过美国人在怪物卡车大赛上处于绝对领先地位，所以也算平衡了一点。）

代数？毫无疑问肯定是要掌握的。解析几何也要稍微懂一点。这些知识都很有

用，而且只要几个月就能掌握了。那么剩下的呢？我觉得稍微介绍一下基本就可以了，花一个学期甚至一年去学习就有点过分了。

这份列表在我看来是为那些打算从事科学和工程专业的学生准备的。高中里教的数学对立志要当程序员的人来说根本没用，而且程序员的需求量早就已经甩开所有其他工程类行业了。

就算你真的立志要成为科学家或是工程师，在真正理解什么是数学以后（起源、发展、作用），学习几何和三角函数的难度也会下降，你也会更容易欣赏数学。你完全不需要去死记硬背几何证明和三角恒等式。可惜这正是高中硬塞给你的东西。

由此可见，这份列表已经过时了。学校不但教错了科目，教学方法也是错的。难怪程序员都觉得自己不需要数学：因为学校教的数学绝大多数都在工作中用不到。

学校里不教的数学

实际生活中，计算机科学家常用的数学和上面那个列表几乎没有重叠。其一，小学和中学里教的绝大部分数学都是连续的，也就是实数上的数学。而对计算机科学家来说，95%有趣的数学都是离散的，也就是整数上的数学。

我打算单独再写一篇博客，专门讲计算机科学、软件工程、编程、黑客等很多经常被混淆的概念之间的区别。这些不成熟的想法主要来自理查德·加布里尔的《软件模式》，所以你要是等不了，也可以去读读那本书。那是本好书。（作者注：我后来没写。因为没必要了，大家都知道所谓的“计算机科学”其实就是用词不当。^①）

不过我们暂时先不去纠结“计算机科学家”这个词。乍听起来很唬人，但数学并不是计算机科学家的专属领域。你完全可以像地下黑客一样自学，而且可以不输甚至超过他们。程序员的背景有助于你把注意力放在更务实的方面。

用来解决可计算问题的数学主要都是离散整数上的数学。这么说有点太笼统了。不过假如你读了我今天的博客后同意我的观点，进而去有意识地多学一点数学，你就会发现我的这个说法在很多地方是不对的。但到了那个时候，我相信不会很久，

^① 译注：见 <http://cacm.acm.org/magazines/2012/10/155530-where-is-the-science-in-computer-science/abstract>。

你会更加自信，完全可以无视我这里写的东西，以自己的方式来学习数学。

对程序员来讲，离散数学里最有用的部分就是概率论。这是小学生在学完算术后最应该学的东西。那么到底什么是概率论？为什么要先学它？因为它就是计数。扑克里组成葫芦（满堂红，三带二）的方式有多少种？大同花顺呢？只要你遇到的问题以“什么什么有多少种”或者“什么什么的概率是多少”这种形式出现，那就是一个概率问题。只要事件发生（概率是多少），那么它就转变成一个“简单的”计数问题。通常这种问题都是由抛硬币开始的。这绝对是学会怎么用计算器以后首先应该学的东西。（作者注：有人告诉我说我把组合数学和概率论混为一谈了，可在我看来，它们只是一枚硬币的两面而已。孔乙己什么的最讨厌了。）

我到现在还保留着大学里的离散数学教科书。对小学三年级生来讲（或许）重了一点，但是它包含了很多计算机科学和计算机工程里“每天”都会用到的知识。

不过很奇怪，我的教授当初根本没告诉过我离散数学是干嘛的。也可能是我走神了或是别的什么原因。所以当时听课很不专心：反正这门课和编程没关系，我只要及格就好了，然后赶快忘掉它，再也不用和它有什么交集。我在大学里差不多有四分之一的计算机专业课都是这么混过去的。现在想想太惨了！后来我只能靠自己一点点摸索出哪些东西是重要的。

要是每门数学课都花一个星期来介绍自己到底是干嘛的就好了，方式要尽可能地吸引人，这样你才知道自己到底为什么要学它。咳，其实每门课都应该这样。

除了概率论和离散数学，其他数学分支也是有助于程序员的，可惜除非你去辅修数学，否则学校是不会教你的。它们包含了：

- 统计，我的离散数学书里讲到了一点，但是统计是一门完整的学科，而且是非常重要的学科，重要到根本不需要额外介绍。
- 代数和线性代数（比如矩阵）。线性代数应该紧跟在代数后面教。它不是很难，而且在很多领域都非常非常有用，比如机器学习。
- 数理逻辑。关于这门学科，我有一本非常牛，但是完全读不懂的书。作者是斯蒂芬·克莱尼。他发明了克莱尼闭包，以及（据我所知）Kleenex。别

费劲去读了。我发誓我至少尝试了 20 次，每次最多到第 2 章就放弃了。不过毫无疑问，这是非常重要的知识。（作者注：没想到有人认真了，还跑过来告诉我其实 Kleene 没有发明 Kleenex。数学家，哈！）

- 信息论和柯氏复杂性。是不是很奇怪？我打赌中学里是绝对不会教这个的。这两个都是比较新兴的学科。信息论（粗略地讲）主要是关于数据压缩的，而柯氏复杂性（同样很粗略地讲）则是关于算法的复杂度（比如最小空间是多少，需要多长时间，程序或者数据结构有多优雅等）的。它们都是好玩，有趣，有用的学科。

当然还有其他的分支，而且有些学科互有重叠。但重点在于：对你有用的数学和学校觉得有用的数学是非常不同的。

那么微积分呢？大家都教，所以应该很重要吧？

其实微积分一点也不难。微积分在我学习之前给我的印象好像是全宇宙最难的东西，和量子力学是一个等级的。量子力学现在仍然超出我的理解范围，但是微积分完全是小菜一碟。在发现程序员学数学有多简单之后，我重新捡起微积分的教科书，每个晚上读一个小时，一个月就读完了。

微积分的本质就是连续——变化的速度，曲线下的面积，固体的体积。很有用，但是需要大量的记忆和很多烦琐的步骤，程序员通常不需要这些东西。知道大致概念和技巧就可以了，细节方面等到需要的时候再查也来得及。

几何、三角函数、微分、积分、圆锥截面、微分方程，以及多维和多元的对应版本——这些都是非常重要的应用。只不过没必要全部记在脑子里。所以花上好多年的时间反复去证明和做练习似乎没什么意义，你说呢？就算你真的要花那么多时间去学习数学，也应该是去研究那些和你的生活息息相关的东西。

学习数学的正确方法

广度优先才是学习数学的正确方法，深度优先是不行的。先打开眼界，了解各种名称，搞清楚什么是什么。

举个例子吧，手算除法。你现在还能在纸上手算除法吗？就现在。有谁可以？

我看没什么人可以做到。

回头看看小学里教的手算除法，其实它真的不算很复杂。整个步骤非常有条理，但是你真的没必要用手算啊，哪怕被困在没有电的沙漠荒岛上，你也能找到计算器吧。比如说手表上，或者牙齿填充物等其他任何东西里。

那么学这个到底有什么用？为什么我们会对忘记怎么手算除法隐隐感到不安？毕竟我们根本用不着它啊。而且，只要你在线，你就可以对任意大的数字做除法。假设你被关在某个阴暗潮湿的地牢里，除非你能算出 $219308862/103503391$ ，否则不会释放你。这时候该怎么办？太简单了。你只要用一个计数器，然后不断地从被除数上减去除数，直到无法再减就可以了，剩下的数字就是余数。如果还不满意，你可以继续做减法，估算出余数的小数形式（这里的余数差不多是 0.1185678219，至少这是我的 Emacs M-x 计算器给我的答案。够接近了吧！）

你最终能算出结果是因为你知道除法其实就是不断进行的减法。只要你知道了这个，除法的直观概念就深深地印在你脑子里了。

学习数学的正确方法应该是忘记那些算法和证明（绝大多数情况下），去了解那些技术的方方面面：名称，作用，大概是怎么计算的，历史有多悠久，（有时候还可以了解一下）是谁发明的，局限性在哪里，相互之间的关联是什么。这就好像是数学里的素质教育。

你问为什么？因为运用数学的第一步就是要界定问题。当一个问题在手却不知道如何下手的时候，是最花时间的。但假如你看出来这是一个微分问题，或是凸优化问题，亦或是布尔逻辑问题的话，至少你立刻就知道从哪儿着手寻找答案了。

现在的数学技术和分支学科的规模已经非常庞大了。如果你连组合数学是什么都不知道的话，那么你也不太可能识别那些属于组合数学的问题了，对吧？

说回来，这其实是好事，因为了解这些领域和各种名称与掌握真正的算法、建模的方法，以及计算出结果比起来，实在是简单太多了。比如链式法则，学校里都会教，你把它背下来去应付考试，然后呢？有多少学生真正了解它的“含义”？结

果等到了真正遇到一个链式法则的问题时，没人知道要去用那个公式。知道它是什么比死记硬背记住怎么用公式要简单得多，可惜学校是不会这么教的，可笑吧？所谓的链式法则其实就是对“串在一起”的函数求导——也就是说，函数 $x()$ 调用函数 $g()$ ，而你想要得到 $x(g())$ 的导数。程序员对函数是再熟悉不过了。我们每天都在和函数打交道，所以现在理解起来比在学校里的时候要容易多啦。

这就是我为什么说他们的教学方法是错的。好多地方都错了。他们把注意力放在那些经验证明对高中毕业生来说没用的学科上，而且教授学科的次序也弄反了。在学会求导和积分之前，应该先去学怎么计数和编程。

我觉得学习数学的最佳方式是每天花上 15~30 分钟浏览一下维基百科。上面有各种关于数学分支的文章，数量多达好几千。你可以自行挑选觉得有趣的文章来读（比如，弦理论，或者傅立叶变换，或者张量，任何让你觉得很牛的东西）。读去吧。遇到任何不懂的东西，只要点一下链接跳过去读就可以了。一直读，读到你觉得乏了累了。

几个月下来，就能让你眼界大开。你会渐渐摸出一些门道，比如，每一种数学分支都会涉及一个对应复杂多元版本的单变量，而这个多元版本几乎一定是用矩阵或者一次方程来表示的。至少在应用数学里都是如此。所以线性代数的重要性就会慢慢上升，直到你发现不弄懂它到底怎么回事就不行的时候，就会跑去下载教程或者去买本书来读，皆大欢喜。

只要坚持这个方法，很快你就会找到数学基础这个链接 (http://en.wikipedia.org/wiki/Foundations_of_mathematics)，所谓条条大路通罗马，这篇文章就是罗马。基本上所谓的数学就是将人们在某些领域里的“常识”形式化，进而 在那个领域里推断或者证明出新的事物。数学研究之所以那么迷人就是因为它唯一的极限就是数学本身：形式化模型本身固有的能力、证明、公理系统，以及法则、信息、计算的表示等。

符号是很容易让人崩溃的东西。对外行来讲，数学符号更是其中之最。就算你对总和、积分、多项式、幂等概念都异常熟悉，可看到它们成群结队地出现时，你

仍然会下意识地把它们当做一个原子操作，略过去拉倒。

但是浏览数学则完全不同，尝试去了解人们到底在努力解决什么问题（或许有一天这会对你非常有用），你就能渐渐地从那堆符号里看出端倪来，它们也不会再像天书一样。比方说，就算你有一定的基础，总和的符号（大写的 Σ ）和总积（大写的 Π ）乍看之下还是很吓人。但是如果你是程序员，你就能马上反应过来，这其实只是一个循环罢了：一个把值累加起来，而另一个就是将它们相乘。积分只不过是连续区间里曲线的和而已，所以习惯以后也就不怎么可怕了。

只要了解了数学的各个分支以及各种不同形式的符号，你就离掌握大量数学知识又进了一步。因为这时的数学已经不再那么可怕了，下次再遇到数学问题的时候，你马上就能做出反应。“嘿！”你会想，“我见过这个。这是一个乘号！”

接着只要掏出计算器就行了。这个计算器可以很炫，比如 R、Matlab、Mathematica，甚至是能支持向量机的 C 库。反正几乎所有实用的数学在很大程度上都可以自动化，所以你只需要一些能自动帮你干活仆人就好了。

做习题有用吗？

经过一年业余时间的自学数学，你已经在脑袋里塞了很多东西了，只不过你还从没动过笔。比如，你肯定经常见到多项式，久而久之自然潜移默化。对数、求根、超越数等各种基本的数学概念都会变得熟悉起来。

这么做同样可以培养出和动手做题一样的感觉来。我发现我喜欢用所谓的“可行性测试”来看自己是不是能跟得上解释（证明）的思路。举例来说，假如我看到有人在做因式分解的时候，心里大概会知道结果是什么样子的，要是最后结果看起来差不多是对的，那我就不会深究。但要是中间步骤比较诡异，又或者看起来不太对或是不太可能，我就会仔细检查。

这和读代码是不是差不多？你用不着在纸上将整个程序跑一遍。只要知道计算的大致过程是怎么样的，那么只要看看结果是不是合理就行了。比如说，假如返回结果应该是个列表，但却返回了一个标量，那说明肯定什么地方有问题了。不过通常你扫过代码的速度和读英文的速度是差不多的（有时候可以做到一样快），而且

读完以后对程序的大致流程有相当的自信，或许还能立即指出一些明显的错误来。

我认为很多泡在数学里的人（数学家和业务爱好者）都是这么读数学论文，或是各种包含大量数学的经典论文的。和你读代码的时候一样，他们也只会凭感觉大致判断其正确性，若非真的想要鸡蛋里挑骨头，就不会太过深究。

不过我有时候仍然会动手做习题。有些东西如果反复出现的话（比如代数和线性代数），我还是会去做练习，以保证自己真正弄懂。

我想要强调的是：不要被习题抹杀了你对数学的兴趣。要是某个题目（甚至某篇文章或章节）让你觉得无聊，不妨大胆跳过去。再频繁也没关系。跟着感觉走。这样你的学习速度会快很多，也有助于培养自信。

这对我有什么用？

或许没有，至少没有立竿见影的效果，但这会提升你逻辑推理的能力，这就好像健身，只要每天坚持，整个精气神都会得到改善。

对我来说，我一直都感兴趣的领域（比如人工智能、机器学习、自然语言处理，以及模式识别）都需要大量的数学。而当我深入研究后发现，其实那些数学并不比高中数学难多少，大多数情况下只不过是不同类型的数学，不会更难。学习数学让我可以写出神经网络、遗传算法、贝叶斯分类、集群算法、图像匹配等各种很有技术性的代码，或是在自己的代码里运用这些东西作出很酷的应用程序来，好让我炫耀给朋友看。

现在再看到那种有一堆数学概念的文章，我也不会发怵了：不管是 n 选取 k （译者按：就是二项式系数），微分，矩阵，行列式，还是无穷级数，等等。概念的作用就是为了易于理解，只不过一开始有点费脑子，让人有点畏惧罢了（好似编程语言的语法一样）。不过今时今日我已经能跟得上了，就算遇到不懂的东西，我也不觉得自己好像白痴一样。因为我知道我一定会弄明白的。

这难道不是好事吗？

而且我只会越来越得心应手。我还有的是时间，有很多书和文章可以读。有时候我会花整个周末去读一本数学书，有时候又会好几个星期想都不去想。这就和其他兴趣、

爱好一样，只要有兴趣，它就会越来越简单，不管你在上面花多少时间，总会有收获的。

每天学一点数学。我当初的这个主意实在是太棒了！

作者手记：土豪程序员的美食

我在招人的时候有一个诀窍。就是在寻找优秀的软件工程师“通才”的时候，“编译器”是我唯一感兴趣的词。通常在简历上你可以看到各种让你觉得不行的关键字和词。只有很少的一些词能让你眼前一亮，我的意思是“在一场比赛中胜出的可能性比较高”。而“编译器”就是其中最惹眼的词之一。

别想太多，我不会把这些词都说出来的。秘密都说了，我还怎么混啊！但它确实是很重要的一个词。玩编译器的人通常在面试里的表现都会相当出色，不管有没有被问到和编译器有关的问题。相反，对编译器一无所知的人往往在计算机科学教育和对系统架构的整体把握上有缺陷。这样的人就算能过关也只是侥幸而已。

我第一次注意到这种现象（编译原理知识和面试表现之间的线性关系）应该是在亚马逊工作的时候。不过回过头来看，我之前工作过的公司也是一样，那些最牛的工程师都是玩过编译器和解释器的家伙。

所以加入 Google 以后，这就成了我的秘密武器。我招进来的候选人都非常出色，别人一直好奇我是怎么筛选的，现在你们明白了吧。

如果你是招聘人员或是雇佣经理，那么单是这篇文章就已经值回票价了。

如果你是程序员，而且还不知道这些东西，那么希望我能说服你开始学习。

土豪程序员的美食

“橄榄花园：穷人享受富人美食的地方。”——大卫·耶格

这又是一篇我犹豫很久，希望找到一种比较礼貌的方式来表达一些基本上不礼貌的观点的文章。可惜我没找到。所以，继续往下读的话，你很有可能觉得自己中

枪了。（嘿！别说我没警告过你。）

我顺便关了评论功能，因为总是有一些聪明的坏蛋用一些非算法的办法绕过captcha，我实在是疲于应付。所以只好对不起大家了。

我不喜欢搞突然袭击，所以我打算先说一个婉转但依然坚持观点的总结。要是你读完以后没有心跳加速感到不适，那么就可以继续。否则最好还是先灌两口酒，就像老电影里演的那样，要锯腿之前先喝酒壮胆。每次遇到这种情况我都会这么做（我是说喝酒，不是锯腿）。

婉转但依然坚持观点的总结：不懂编译器原理的人，也不懂计算机的原理。如果你吃不准自己是不是真的懂，那么你就是不懂。

知知为知知，不知为不知，这个道理你懂不懂？

事实上，以我个人卑微，而且可能错得离谱的观点来看，编译原理是计算机科学本科里第二重要的课程。

因为过去一年来，每次尝试深入这个话题都失败了，我一直想要说服自己，等酒醒以后，我要好好聊聊这个话题，起承转合，错落有致。好了，废话不多说……

算了，这好像太麻烦了。我还是发两句牢骚就可以了。反正你付那么多钱不就是来听我发牢骚的嘛。嬉笑怒骂，保证精彩。

单人床和大胡子

我在学校就学过编译原理了，当时的老师是华盛顿大学的大卫·诺金教授，差不多是1991年的时候吧。

你猜我得了几分？0分，折合绩点0.0。我成绩单上的最终成绩就是这个。在华盛顿大学，假如你的课业是“未完成”，然后又不去补救的话，就会得鸭蛋（顺便插一句，我一直都没搞明白到底要怎么补救）。反正不管怎么说，期限一到，未完成就会变成0分。

得到“未完成”的途径有好几种，包括我的那种，就是当一个脑袋进水的大傻瓜，选课以后一直到过了退课期限才反应过来自己不喜欢，不想学了。所以这个“未

完成”是我“应得”的。

几年后我又选修了一次编译原理。我在大学里待了很久，因为毕业前一年，我在 Geoworks 找了份全职工作（当然还有其他原因），结果拖了好多年才毕业。

千万别学我。工作以后想要再静下心来读书就很难了，先拿学位再工作。假如你是快毕业的博士生的话就更不应该放弃。当 ABD 只会让你后悔终生。就算你自己不这么觉得，我们也会为你感到惋惜。（译者注：ABD 就是 All But Dissertation，就是除了论文其他都准备好了的意思，引申为那些临门一脚放弃博士学位的人。）

这回我拿到了一个很高的分数。其实第一次读的时候我就已经在一定程度上理解编译原理了，所以第二次算是轻车熟路吧。只不过这么多年来我一直没有领会到编译原理为什么这么重要，我现在和你分享一下我的经验，省得你和我一样走弯路。

1991 年我第一次选修的时候我是这么想的，你可以看看是不是似曾相识。当时我想：编译器只是一个工具，它会对我的程序这里不行那里不对地抱怨一通，最后把它转换成计算机能理解的语言。写程序需要各种工具，编译器只是其中之一。比如你得有一台电脑吧，然后还需要键盘，或许还要有个账户、编译器、文本编辑器，再加个可有可无的调试器就可以开始工作了。会写程序就是程序员。现在只要学一下怎么用 API 就好了。

每次学习编译原理的念头浮现的时候，我都会告诉自己：只有在两种情况下我才需要了解编译器到底是怎么工作的。第一种是去微软上班，而且还是在 Visual C++组。那么很自然我需要了解编译器的工作原理。第二种就是我突然迸发出一股热情，开始蓄长胡子，不洗澡，然后去 MIT 朝圣，求理查德·斯托曼让我在走廊里支张单人床，然后像个云游四方的基督徒一样和他一起写 GCC。

在当时，这两种情况似乎都不太可能发生在我身上，不过就算真的有那么一天，和去微软上班比起来，单人床和大胡子好像也不是不能接受吧。

对了，很久以前，我哥们大卫曾经去参加过一个聚会，结果全是微软的人，而且还有个不知所谓的家伙在那里大声吹嘘自己手下的 Visual C++组里有 15 个全世界最好的编译器程序员（要知道这是在聚会啊）。我跟大卫说：“哇哦，没想到理查

德·斯托曼在微软上班。”大卫表示非常懊恼，怎么当时没这种急智呛他一下。我说过都过去了，想也没用。

我说这个故事的原因是我发现自己有时候也会忍不住想要吹嘘一下在 Google 同我一起工作的那些顶尖编译器工程师。所以我在这里有个不情之请：要是你碰到我在聚会上吹这种牛的话，千万别客气，直接让我闭嘴。必要的时候直接抄起台灯把我砸晕了也行。

无论如何，你现在了解我在 1991 年那会儿对编译器的态度了吧。当初选课的时候可能是脑子进水了。反正我是没修完。第二次尽管花了足够的时间去理解，但也只是为了得到一个漂亮的成绩去完成课程罢了（我决定再次选修并不是因为上次得了 0 分所以心里不爽，而是不想让大卫·诺金失望）。

其实我和普通人没什么两样。如果你是计算机科学专业的学生而且还喜欢编译器（基本上在全世界范围里的计算机科学专业里，你也可以排到前 5% 了），那我真的很敬佩你。不过我 Nethack 肯定比你玩得好！事实上绝大多数程序员都和我一样，所以我也没资格批评别人。

在结束这个愚蠢的故事之前，我觉得有必要指出这其实有一部分是学术界的错。除开类型系统的研究不说（基本上大家对这个亚瑟王的圣杯任务非常谨慎克制），编译器也在学术界失宠好多年了。所以学校里也不怎么强调编译器的重要性。很多学校甚至都不要求选修编译原理就可以得到计算机科学的学位，这实在是令人唏嘘不已。

唉。

那你说怎么办……

你是程序员吧？那好，我现在给你几个编程场景，你告诉我你会怎么解决它们。

场景一：你的日常工作语言是 Java，公司对代码的格式有严格的规定，详细到每一个能想到的细节。你该怎么设置编辑器，让它能根据规定帮你自动格式化代码？

场景二：公司拥有很多 Ajax 代码，这些 JavaScript 代码的增长速度完全不亚于其他代码。你打算用 jsdoc（JavaScript 版的 javadoc）来注释函数，这样就能自动生成

成文档了。可是你发现 jsdoc 只是一堆写得很烂的 Perl 脚本，整个代码库扫描到一半的时候就会崩溃。而且（请原谅我这么说）你已经发誓这辈子都不再碰 Perl 了，因为，没有为什么。随便什么理由都行。那么只好自己动手写一个 jsdoc 出来，你会怎么写？记住它需要能够扫描解析 JavaScript 代码哦。

场景三：你的公司有一个巨大的 C++ 代码库，是多年来数十名乃至数百名工程师努力工作的成果。你发现需要对这些代码进行大规模的重构，比如从 32 位升级到 64 位，或是修改使用数据库事务的方式，或是（上天保佑）因为需要升级 C++ 编译器，语法和语义全都（又）变了。你的任务就是要把代码调通了。你会怎么做？

场景四：公司同事新开发了一个给予浏览器的代码审查工具，非常牛。大家都很喜欢。可用了一段时间以后，你发现要是它能高亮语法就好了。可是你没有很多时间，最多就是抽出一个礼拜的业余时间来实现这个功能。你会怎么做？（假设你公司的代码 99% 由 5~8 种语言构成。）

场景五：手头上的项目里出现了一个意料之外，有点怪异的新需求：需要支持某种新型号的硬件路由。可能是你那些 Web 2.0 的代码把边界路由器或是流量监视器都弄坏了吧，谁知道呢。反正系统维护和网络工程师说你要直接去和那些新的路由器打交道。这些路由器有一个 IP 地址、telnet 接口，以及一套专用的命令语言。你发送命令，它们就会返回结果。每个命令都有自己的语法和参数，而且你需要解析返回结果（没有文档，不过反正可以反向工程嘛），搜寻特定的模式，然后把那些诡异的上传下载调节到正确的状态。你会用什么工具？

场景六：公司的项目最近有点失控。工程师都是聪明人，他们所采用的语言和遵循的敏捷，面向对象工程理念全都是业界最新、最了不起的。所以这绝对不是他们的错。但是不知道怎么回事，代码开始变得越来越复杂，估算项目的时候越来越离谱。很简单的任务也要好久才能完成。大家都开始讨论要怎么重新设计。这已经是过去 5 年来第 n 次重新设计了，而且“这次”肯定能达到翻天覆地的效果，解决一切问题。你会给他们什么颜色的纸呢？啊哦，不好意思，我是说你怎么确保这次他们能成功？

场景七：你拥有一家小规模的创业公司，公司里全是各种奇形怪状的年轻人，

染成蓝色的长发，戴鼻环、舌钉、穿嬉皮式的黑色衣服，玩 iPhone 等各种年轻人的东西。你的网站用的是 Ruby on Rails，它完全能够应付当前的访客数。（你也从来没打算去衡量一下网站的延迟对访客数的影响，因为压根就没想到过这个问题。）有一天你读到了 Rails 最新的安全隐患，用户只要精心构造一个 GET 请求，就能以你公司的名义向 SEC 提交任意文件。你赶紧去下载最新版本，仔细阅读单元测试代码，想要搞清楚那个隐患到底是怎么回事，因为他们没有明说，所以你决定大刀阔斧地修改代码，把某种写法清除出去（不知道为什么没办法用 grep 来找到这些代码），代之以另一种等价的写法。你会怎么做？

场景八：有个喝多了不知所谓的博主，给你列出了几种莫名其妙的场景，要求你找出它们之间的共同点。你知道答案了吗？

答案在此。什么？你以为我是说笑的吗？

场景一：说服公司采用 Eclipse 默认的风格。

场景二：到 jsdoc 邮件列表上去问问看别人是不是也遇到了同样的问题。有几个人回复说他们也遇到了，然后这个问题基本上就不了了之了。

场景三：辞职。这还用我教你啊？读到第一个逗号的时候你应该就知道答案是什么了。

场景四：管它的。菜鸟才需要代码着色，或者用 GNU Source Highlight，它支持从 Fortran 到 Ada 等各种语言，将就着用吧。

场景五：Perl。那可是瑞士军刀一样的万能工具。等你把自己开膛破肚的时候，就没关心你是不是解决问题了。

场景六：粉色。（译者注：通知员工终止雇佣合约的通知书通常都是用粉色的纸，工资单里要是夹着一张粉色的纸就表示要失业了。）

场景七：自己动手。整个网站也才不过 1 万行代码而已，有什么大不了的。你用的是 Rails，别哭哭啼啼的，认真你就输了。

场景八：找到了。你读到最后才知道计算机科学里最重要的是哪门课。史蒂夫

最擅长讲这种冷笑话了。

好了，现在各种可能出现的编程场景你都知道要怎么应付了。你还需要了解编译器干什么？

编译器的工作原理

当被问及编译器的工作原理时，下面是一些真实的，来自拥有计算机科学博士学位的应聘者的答案。

真实的应聘者1号：“哦！它们啊……嗯……会逐行扫描你的程序，然后把每一行都转成汇编语言。”

真实的应聘者2号：“编译器会检查程序里的错误，而且……嗯……还会指出语法不对的地方。我就记得那么多啦。”

真实的应聘者3号：“我……（卡住3分钟）……我不知道。”

真实的应聘者4号：“它们会预处理你的程序，把#define语句变成代码，然后……嗯……输出机器码。”

差不多75%的应聘者都只能做到这样，因为，嘿，没人想要在MIT的走廊里工作呀。所以你也不能怪他们吧？

只有差不多3%~5%的应聘者（这还是乐观的估计）能说出编译器工作的所有细节。剩下的或许能手舞足蹈地抛出lex、yacc和代码生成的那么几个词。

我刚刚说过会让你心跳加速的，是吧？

放轻松，深呼吸。

为什么编译器很重要，第一部分

编译原理是一门重要的计算机科学课程的首要原因就是，它非常切实地将你之前学过的几乎所有东西都捏合在了一起。

不了解计算机架构的人，是不可能完全理解编译器是怎么工作的，因为编译器会输出机器码。这不光是一堆指令，比编译器需要理解底层的机器到底是怎么运作

的，然后才能高效地翻译源代码。

另外，所谓的机器只不过是任何可以进行计算的一种表达而已。Perl 是机器，操作系统也是机器，Emacs 同样可以是机器。要是你能证明家里的洗衣机是图灵完备的，给它写一个编译器来跑 C 代码也是可以的。

这些你都知道的吧。

不了解操作系统的人也无法理解现代编译器的工作原理，因为现在任何像样的机器都离不开操作系统。操作系统接口是目标机器的组成部分之一。你当然可以知道那些在大型机项目上耕耘数年的人，可那些机器最终也快不过 Costco 里卖的电脑。他们或许是因为时间限制而放弃了操作系统，另外整个全球市场大概也就那么一个客户。反正对我们大多数人来说，操作系统就是机器的一部分。

不先学一点计算理论的课程，你是无法理解编译器的工作原理的。计算理论就像是编译原理课本上的第 1 章第 1 节。一定要全部弄懂。

不先学一点编程课，你会发现很难掌握编译器（可能连输入输出都搞不清楚）。你必须了解编程语言的极限在哪里，至少也要在实现程序之前有个大致概念吧。除非你精通多门语言，否则用 A 语言去写个程序把 B 语言转换成 C 语言是毫无意义的。

其实你的周围都是编译问题。每天都会遇到它们。前面我列出的那 7 种场景只不过是冰山一角。（第 8 个才是整个冰山，别敷衍！）

编译器会接收一串符号流，根据预先定义号的规则，分析出这串符号的结构，然后把它转换成另一串符号流。

是不是很笼统？的确是。

一幅图片能不能被当成是符号流？当然可以。它可以是每一行像素所组成的流。每个像素就是一个数字。每个数字就是一个符号。编译器当然可以转换图片。

英语可以被当做符号流吗？当然可以。规则或许会很复杂，但是自然语言处理的确可以被看成是某种很炫的编译（不过一板一眼的确定性的方法是行不通的，它已经被概率方法取代了）。

那么普通代码呢？我是说，图像处理和自然语言处理毕竟只是一小部分而已。剩下的人呢？我们只是写代码而已，这样的话编译器对我们来说还重要吗？

我想问你，你难道从来没有碰到需要写程序处理自己代码库的时候？如果你要写一个语法高亮的功能怎么办？如果你的编程语言添加了一些新特性，但是你的编辑器还不支持怎么办？你只能干坐在那里等“别人”来帮你更新编辑器吗？要是好多年都不更新怎么办？作为另一个优秀的程序员，你是不是抄起袖子自己动手会比较快？

难道你从不在自己的代码库里搜寻某种特定的东西？难道你从来不用自己提取文档？

你有没有遇到过无论怎么努力维护模块性和面向对象，自己的代码还是膨胀得不像话的情况？肯定有。你是怎么解决这个问题的？

要么去学习编译器，然后自己动手写 DSL，要么就只有换一种更好的语言了。

顺便说一句，我要推荐 NBL (<http://steve-yegge.blogspot.com/2007/02/next-big-language.html>)。这是我个人的最爱：邪恶张量场里的极大值，地狱里的制高点。不过现在我还不打算告诉你 NBL 是什么。耐心点！我为它写的 Emacs 模式只写到一半而已。

假如你不学编译原理……

很多程序员不学编译原理的原因之一是这门课听起来很难。通常是计算机科学专业里的巅峰课程（另一门是操作系统），就是说这门课算是一种“可选的通过仪式”，让你成为一名真正的程序员，长大成人。^①

如果你打算在钱花光之前按时毕业，而且不想成绩太难看，导致未来雇主直接叫看门狗哄你走的话，谁也不能责怪你选择避开“可选的通过仪式”吧？

我当然不是在说其他计算机科学的课程就不重要。操作系统、机器学习、分布式计算、算法设计等其实都和编译原理一样重要。只不过学完那些课程后，你还不

^① 译注：通过仪式一般是指在遇到人生大事时必须经过的某种仪式的洗礼，从而蜕变成一个新的自我，比如出生、结婚等。

知道计算机是怎么工作的话，这就让我觉得编译原理真的应该是一门 300 级的必修课了。可惜要学好它需要太多的基础知识，要求大多数学校这么做有点不太现实。

设计一套行之有效的计算机科学本科课程并非易事。也难怪那么多常春藤大学都基本上放弃努力，转而变成了一家家 Java 认证培训机构。

如果你是认真好学的计算机科学专业学生，选修操作系统和人工智能是最低要求。这样你最后或许不懂编译原理（很可惜），但至少可以在很多其他领域里和别人一样体现自己的价值。这就很有成就感了，至少不输其他人。

去吧，年轻人。

很遗憾，今天大多数程序员眼里只有学位。他们根本不在乎自己学了什么。他们只要能拿到学位，找到工作，付得起账单就行了。

大多数程序员喜欢的那些课程可以归结为计算机科学里的橄榄花园：笨蛋程序员向聪明程序员学习的地方。

我实在不想列出这些课程的名单。我不够敏捷，躲不开那些动画化的，项目管理型的，面向对象的工程师针对我用 Java 和 Web 2.0 技术所写出来的血腥游戏，他们会利用那些通过团队和现代软件方法论合理设计出来的用户界面把我架在火上烤。我会变成软件伦理及其对我们的文化冲击中的案例分析。

不过我不说你大概也能猜到这些课程是什么吧。

如果你不选修编译原理，那就要冒着永远混迹于二流程序员行列的风险：就是那种从热情的青年架构师最后混成阴郁的古董架构师，终其一生都在构建某种大型系统，还引以为傲。

大型系统都很烂

这条定律具有 100% 的传递性。假如你做出这么个系统来，那么你也很烂。

编译器阵营

事实上很多所谓的编译器“专家”其实也没有很懂编译器，因为编译器在逻辑上可以分成 3 个阶段，之间的差异之大，可以组成完全不同，几乎没有重叠的研究

领域。

编译过程中第一个大阶段就是解析，即把输入的内容变成一棵树。中间要经历预处理，词法分析（也叫单词化），然后是语法分析和中间代码生成这几个步骤。词法分析通常是由正则表达式来完成的。语法分析则是根据语法完成。你可以采用递归向下（最常见），或是解析器生成器（在小语言中比较常见），或是更炫的算法来实现，只不过相应的执行速度也会慢一点。无论如何，最后的结果通常都是某种解析树。

职业程序员光是了解这些就能比其他人走得更远了。就算你完全不懂接下来的编译是怎么进行的，你也可以利用工具或算法来产生解析树。事实上，完成解析就可以帮助你解决之前提到的场景一到四了。

如果你不懂解析，那么就只能用正则表达式去勉强实现，要是连正则表达式也不懂，那么最后手写出来的状态机肯定是千疮百孔，没人看得懂也跑不起来的垃圾。

我是说真的。

我在面试中必问的一个问题就是，怎么在一堆 HTML 文件里查找电话号码？很多人（差不多 30%）都会给我写一个 2 500 行的 C++ 程序。

后来，应聘的人开始告诉我说他们在我的博客上读到这个问题了，仔细想想，还挺奇怪的。所以现在我都不再问这个问题了。（作者注：5 年后我在一次电话面试里又试了一次，结果那个人立刻说：“我在你的博客上见过这个问题了！”唉，太可惜了，这真的是个很不错的面试题啊。）

我有时候会问它的变形题，一样可以问住他们：你要么能看穿这是一个很简单的问题，要么就只好掏出瑞士军刀找机会自裁，省得给家里丢人。

C++ 就经常有这种令人吃惊的效果。

第二个大阶段是类型检查。这是一群狂热的学术分子（包括他们的组织以及/或者手下的研究生），他们自信可以写出非常聪明的程序，能分析出你的程序想要干什么，并且在你出错的时候帮你指出。不过奇怪的是，他们并不觉得自己是在研究人工智能，毕竟人工智能界已经（明智地）放弃确定性的方法了。

这些人已经基本上找出了确定性检查的极限，然后他们就宣称这就是计算本身的边界，边界之外都是弱肉强食的荒蛮之地，或者叫“通过软件赚钱的污秽之地”。

他们只有在狂欢派对上喝醉的时候说的话还可以听听。

我的一个拥有语言博士学位的朋友最近跟我说，当发现多年来追求数学上的美丽和纯粹在现实世界中一文不值时，那种感觉是“非常痛苦”的。

这其中的问题（之一）正是其背后的前提假设，那就是如果没有辛德勒-米尔纳类型系统，或是只有 Java 那样的垃圾类型系统的话，绝对写不出有用的代码。它最终只会不堪负重：对粗心大意的探险者来说，它就像是一个巨大无比的无类型陷阱。

显然他们都是在闭门造车啊。

另一个问题是他们觉得任何类型“错误”，不管在当下对你的程序的影响有多么微不足道，都应该像华尔街日报头版一样严肃对待。所有人都应该立即放下手头的一切，直到问题修复为止。类型“警告”这种东西压根就不应该存在。

还记得模糊逻辑么？哦，哦，等等——记得冯·诺依曼和斯坦·乌拉姆引入蒙特卡洛方法的时候吗？哦，对了，我忘了：你是 90 后，现在也就 24 岁，而我已经 90 多岁的老古董了。

算了，总有一天他们会反应过来，严格的确定性是行不通的，所有存在维度灾难的领域，最终都转向了概率型的方法。

“可选的静态类型”才是具有光明前景的雏形。谁有机会成为 NBL？

第三个阵营是代码生成，他们通常都被边缘化了。只要你对递归有足够的了解，知道自己的祖先不是亚当和夏娃，那么代码生成还是挺直观的。这里要讲的其实是优化，就是那种生成足够正确的代码，让绝大多数用户都意识不到有问题的艺术。等等，不好意思，这是亚马逊化。优化是指根据你那些昂贵的菜鸟程序员写出来的垃圾代码，生成“正确”代码的艺术。

我觉得编译器优化尽管很好玩，但它其实是一个黑暗永恒的深渊。所以也可以

说是一个很好玩的黑暗深渊吧。不过你可以将它推向几乎不可能的极限，这是一个开放活跃的研究领域，等到他们“完成”的时候，他们和类型检查领域的人都会被称为人工智能专家。

我想说的其实是机器学习，“人工智能”这玩意儿不但是确定性的方法，而且现在也没人肯投钱了。

不管怎么说，这3个领域相互之间其实没什么交集，却都有资格在聚会上自称是“编译器专家”。

编译器的阴暗面

之所以这篇莫名其妙的博客花了我这么多时间，原因之一就是我希望自己在吹嘘之前，先自己动手写一个编译器出来。

这个任务已经完成了！（见 <http://steve-yegge.blogspot.com/2008/11/ejacs-javascript-interpreter-for-emacs.html>。）

好吧，勉勉强强。其实“未完成”或许更准确一点，不过我发现，所有号称稳定的编译器其实都是这个状态。

我不想在时间未成熟的情况下说太多细节，反正我试着用一门很有用的语言，为另一门很有用的语言写了一个解释器，在一个很有用的平台上输出很有用的字节码。

我觉得很好玩。尽管15年前我重修过一次编译原理，编译器和编程语言也只是在过去5年里自学的，但我还是写得很快，也学到了很多。

光是自己动手，我就已经学到很多了。

只不过写编译器等于创造了一个生命。这一点是我没有料到的。我没想过要小孩啊。在业界摸爬滚打20年后，我真的是没料到写一个简单的解释器居然会演变成一生的工作。

你说怪不怪。

鲍勃·杰维斯最担得起“一生的工作”这个说法，他是我的好朋友，同时也是Turbo C的原作者（我就是靠它来学编程的），他是个了不起的编译器作者，可以说

是世界级的。

他最近做了一场技术演讲（Google 有很多这样的演讲），他指出哪怕只是听众要求的一小部分特性都需要耗费一生去完成。

这种说法让我产生了极大的共鸣。18 个月前，我发现自己必须很小心地筛选自己拖拉了 5 年的项目，数量这么有限的项目就已经让我无法兼顾了，在写了“生产解释器”之后，我更加意识到要做的工作根本看不到头。

不开玩笑，真的是看不到头。

因此从某方面来看，其实我应该将现有的成果公开出去进行推广，这样就能让别人来帮忙一起做。但是另一方面，我业余做这个项目并不是要给自己找事做（根本没那么想过），我只不过是想确认自己真的了解编译器，这样才够胆量（在几碗黄汤下肚后）对着几十万读者吐槽。

所以我至少也要完成字节编译器的部分。

我总有一天会做完的，最后的代码保证漂亮。我只和很小一部分朋友提过这个疯狂的小项目，几乎所有人都异口同声地大喊：“你在搞什么？？？？”就好像发现全世界最正常的人突然往嘴里塞炸药时那般震惊。

好了，编译器的话题就聊到这儿吧。改变世界是自己动手写编译器的必由之路。

这就是你要学习它的原理的原因。这也是你要自己动手去写的原因，没错，就是你。

其实它没你想得那么难，只不过它最后会变成一生的工作。这也没什么。只要你愿意，随时都可以停手。不过这样的人很少很少。通常最后都是因为时间、精力有限，而不得不放弃。不过就算这样，你的编程水平也已经远远甩开很多人了。

你会知道怎么去修复语法高亮的问题。

你会知道怎么写从代码里生成文档的工具。

你会知道怎么去修复 Eclipse 代码对齐的问题。

你甚至不会再去吹嘘自己的工具有多聪明，能理解代码等——至少对我来说，这些东西根本不值得大声宣扬。

你会知道怎么去修复你最爱的语言里的各种毛病。别跟我说你的最爱是完美的，没有毛病这种话。

当这个世界上最聪明的一些人（比如詹姆斯·高斯林和盖伊·斯提尔这样级别的）想要为 Java 社区注入真正的闭包功能以及实打实的扩展时，你会很有自信地去反对那些不知所谓的大多数。我很同情那些人。真的。

说不定你还会开始享用土豪程序员的美食。写编译器只是个开始，我可没说过那是终点。你会最终超越那些无关紧要的服务 API 和 JavaScript 小程序，开始编写可以治疗癌症，对付世界上各种病毒以及生老病死的程序。甚至是（我开始胡说八道了啊）幻想用确定性的方法解决静态类型的灵丹妙药。

还有，不管你的编译器是编译什么语言，你会真正了解掌握这门语言。没有其他捷径。不好意思！

好了，写到这里就差不多了。我明早还要出发去参加某个聚会，尽管我不清楚主题是什么，不过猜也知道和编译器应该没什么关系，除了 GVR（吉多·范罗苏姆）有一场关于 Python 3000 的演讲。那场或许还有点意思。

要是你不懂编译原理，那也不必担心。我觉得你仍然可以是一个不错的程序员。只不过挑战一下极限总是好的吧！

那么计算机科学专业里最重要的到底是哪门课？

当然是打字课啦。这还用问嘛。

我得赶紧走了。

第 3 章 Chapter 3

关于 Google

作者手记：应聘 Google

和其他大多数文章一样，这篇文章也是一篇郁闷之作。不过这次之所以觉得郁闷，是因为留不住人才。在 Geoworks 是这样，在亚马逊也是这样，现在在 Google 还是这样。公司其实也清楚它们会错过人才。当然这是另一个话题了（这个话题很大，改天一定好好写一写，保证精彩），不过这里我们先说一个小一点的，那就是应聘者常常准备得不充分。

仔细想想这真的很让人难过。一个绝对有能力有水平的应聘者，只是因为欠缺了一点准备，结果就把面试给搞砸了。

这些年我参加了很多场面试——超过 1000 场吧，1500 多场说不定都有，外加这个数量级的电话面试。在亚马逊的时候，有时候参加校园招聘，一天要面 10~15 场。

20 年来面过那么多人后，你肯定能看出点门道来。其中很大的一个就是，毫无准备的应聘者只要一进门，你就能察觉出来。他们没有做过练习，热身得也不充分，学校里学过的东西都忘得差不多了，很容易在陌生的面试环境里被击溃，然后表现失常，令人失望。

面试很难。当好面试官很难，做一个好的应聘者就更是难上加难。

我们真心希望他们准备充分，在现场发挥出最佳水平。Google 在柯克兰（西雅图分部）会在应聘者参加面试前，先对他们进行一个小时左右的指导，总结之前

的电话面试。还会教授一些面试的技巧。

所以尽管当初这篇文章是在完全未经授权和同意的情况下写出来的，但全球各地的 Google 面试官仍然把它转发给应聘者来准备电话面试和现场面试。数以千计的人因此受益。

我还收集了好几百人的反馈。他们很多人虽然没能通过面试，但还是花时间给我写信，告诉我这篇文章很有帮助，否则结果可能会更糟糕。

Google 的雇佣团队曾经很纠结要不要润色一下“面试拒人团”那一节。我想这大家都能理解，毕竟他们还是希望让 Google 给人一个比较好的形象，这却很明显是我们的流程中一个比较明显的污点。不过最终他们还是觉得原封不动地发表才是最好的，因为它确实有用。很多应聘者告诉我们，正是读了那一节以后，才让他们有自信和勇气去应聘。

这篇文章显然会有一点争议性。很多程序员对于 Google 这样的公司故意问一些没人知道答案的问题都有点恼羞成怒。很多自学成才的程序员在读了文章以后甚至感到很伤自尊。有一名读者还指责我太自以为是，把其他 Google 面试官会问图论这种事情也公布出来。

不过我明白，面试这个话题总是会伤害到一些人的自尊心的。这也是没办法的事情。我在文章里讲到的东西对于绝大多数技术公司（微软、Google、亚马逊、苹果、Facebook）来讲都是比较准确的。其实这些都是很基本的东西——大多数都是计算机科学大二或大三本科的内容。真的不是什么高深的东西。

我还花了好长时间为这篇文章准备了一个续篇，现在已经接近完工。（真的，还差几个星期。）不过那篇只是各种读后感而已。总之，哪怕是稍微准备一下，都能让你在这个大日子里有完全不同的表现。

应聘 Google

我想写点关于应聘 Google 的东西已经很久了。但是我犹豫了很久，因为你们

一定不喜欢。大概吧。这里说的是统计意义上的“你”，很可能你看了以后会生气。

为什么？因为……好吧，我写了一首打油诗：

天哪，史蒂夫讲的那些东西我都不懂

要是我老板觉得这些玩意儿都很重要

那我铁定会被炒鱿鱼的啊啊啊啊

哎呀呀，我的妈呀……

刚开始写有关面试的文章时（很久以前，在其他公司的时候），我完全没想到过这种反应居然是很常见的。如此看来我还真的是很迟钝的一个人啊。

通常的对话都是像这样的。

我：你看啊，我喜欢在面试的时候问 X 的问题，……

你：X 的问题？天哪，毕业以后我就再没接触过 X 了！工作中从来都没碰到过！他在面试中间这种问题？这说明那是很重要的知识啊，可是，可是……我不懂！要是被人发现我这么无知，我不但会因为不够资格而被扫地出门，而且还会因为面试的时候被问 X 的问题而找不到工作啊！要是大家都听了史蒂夫的话，那所有人都会这么做的！我会无家可归，一贫如洗！只是因为我不懂一些我根本不需要的知识！太可怕了！我应该攻击 X 的，但是我不想读书，要弄明白它怎么回事才去诋毁它太麻烦了，还是直接大吼史蒂夫是笨蛋比较好，这样就没人听他的了！

我：所以说，……等等，你刚刚说什么“扫地出门”？“一贫如洗”？你在说什么啊？

你：啊啊啊啊啊啊啊！！！（准备要拔刀子捅人了。）

我：到此为止。我不会再谈任何面试的话题了。

X 是什么并非重点。它可以是任何东西。就算我说“我喜欢在面试的时候问（应聘者的名字）”，他们也一样会发狂，这些人要么对面试本身缺乏安全感，要么就是连自己的名字都忘了，我希望是前者。

铁打的面试，流水的应聘者。我们会想说：“天哪，要是那个家伙准备得再充分一点就好了，他看起来就是个聪明人啊。我们有没有什么办法帮一下他们，让后来人好有所准备？”

可是没有人付诸行动，因为谁都不想被那些不知道 X 的人捅一刀。

我本想就着这个 X 说点心得体会，不要涉及具体的东西，但是又觉得这样太空泛，最后反而搞得大家都很失望。如果要是能匿名的话，我肯定就那么干了。

正所谓良药苦口，再难听的话也会有人听的。与其说点不痛不痒的车轱辘话，我打算解开 X 的面纱，分享一点准备面试的干货。

警告和免责声明

这篇博客并未得到 Google 的支持。Google 不知道我会发表这些心得。这纯粹只是你我之间的交流，希望你们能明白。不要告诉他们是我给了你们提示。只要你能在面试中表现出色，我们就算扯平了。

我只讨论一般软件工程职位，以及和这些职位相关的面试。

这些心得都是通用的。并没有拿 Google 和其他公司比较的意思。这些东西就算放到 20 年前，我的第一份程序员工作上面也一样适用。所以至少对我们的职业生涯来说，它们并没有什么时效性。

光靠这些心得就想谋得一份工作显然是不现实的。我只希望你看了它们以后能在面试中做到最好。

为什么要应聘 Google?

你问为什么要应聘 Google？好吧，我们就把这段对话放在最前面吧。

你：为什么我要去 Google？它是不是真的有传说中的那么好？我真的会快乐吗？我应该马上投简历吗？

我：是的。

你：你回答的是哪个问题？“是的”是什么意思？我还没有自我介绍呢！

我：伙计，我的答案就是“是的”。（你可能是女性，不过我还是叫你伙计好了。）

你：可是……可是，我懒得挪窝啊！现在的公司我做得很开心，或者说我已经多多少少习惯现在这种不舒服了。我一个 Google 的人都不认识啊！如果跳槽的话，要重新去学 Google 的构建系统等各种技术！我在那里没名气，也没人信任我——基本上一切都要重头来过！我不想再熬那么久，跳槽对我没好处啊！太可怕了！

我：伙计！我已经说了答案是“是的”，明白了没有？不管你说什么，我的答案都不会变的。每个新入职的员工和你的起点都一样，当然确实有几个大牛，胡子长得连甘道夫都要甘拜下风，但那真的只是极少数。所有人在应聘时都会有你提到的那些担忧。结果来了以后，每个人都说：“天哪，还好我来了！”所以别犹豫了，赶紧投简历。不过在这之前，先要好好准备一番。

你：要是我悲剧了怎么办？也许我很聪明，水平也够，但偏偏那天表现不好，面试失败了！那太打击自信心了！我宁可错过机会，也不要冒这种风险！

我：没错，你说得也不算错。老实说，我当初第一次也失败了，不过我像条哈巴狗一样哀求他们给我第二次机会。他们心一软就答应了。这次我有备而来，表现比第一次出色多啦。

Google 其实在这方面的失误是有名的，就是说，我们有时候会拒掉合格的候选人，因为就算错过，也比招一个不合格的人进来要好。这在业界也算是共识吧，只是各个公司的处理方式略有不同而已。Google 的漏报率还是很高的。我没有具体的数字，但我认识很多绝对有资格的聪明人最后都没能通过面试。老实说这挺可惜的。

不管怎么样，我想说的是：就算你这次没成功，也不代表你水平不够。所以没必要觉得沮丧。

就我的观察来看，这种情况完全是随机的，和技术经验什么的都没有关系。涉及的因素多种多样，随便举几个例子：

1. 你那天心情不好。

2. 某个面试官心情不好。
3. 你和面试官之间的沟通可能有些大家都没意识到的问题。
4. 运气不好撞上了面试拒人团。

天哪，为什么我会撞上面试拒人团！

好吧，我就知道你会担心这个。

这到底是个什么玩意儿？话说我还在亚马逊的时候，我们就一直在思考这个问题（我相信到现在他们还在思考）。最后我们发现，对每一个亚马逊的员工 E 来说，至少会对应这么一个“面试拒人团”：这些员工 S 是永远不会雇用 E 的。在走进一场面试的时候，理解这种现象对你来说是至关重要的，所以让我来解释一下这些年来的经验总结。

首先，除非面试官来向你寻求建议，否则你没办法告诉他们哪些东西是重要的。至少不是每家公司都能这么做。你只有很短的一段时间来向工程师灌输面试的艺术，差不多是他们大学毕业以后的一年之内吧，之后他们就会觉得自己已经是一个“出色的面试官”了，不会再修改自己的面试题，问问题的技巧、面试的风格、写反馈的风格等都不会再有变化。

这个问题其实蛮严重的。但是我为此吃过好多次亏，所以现在已经放弃提醒别人的念头了。

第二个问题：每个“资深”面试官心里都会有一套主题，甚至是特定的题目，让他们觉得可以准确地判断应聘者的水平。每个面试官的题目都可能非常不一样，甚至完全不重叠。

举个随处可见的典型例子：面试官 A 喜欢问 C++的各种细节，文件系统，网络协议，具体数学。而面试官 B 喜欢问 Java 的各种细节，设计模式，单元测试，Web 框架，项目管理。任何应聘者如果同时遇到 A 和 B，他得到的结果就很可能非常不同。说难听点，A 和 B 可能互相都看不上眼呢。只不过正好他们都是在面试官 C 手上通过的。C 最喜欢问数据结构，Unix 命令，进程和线程的区别等，

A 和 B 正好都能答上来。

基本上你能通过一家技术公司的面试都是这种情况。你正好答上来了那些问题罢了。这是面试本质上的缺陷，就算你是阿兰·图灵，也一样会碰到看不上你的面试官。老实说，如果你真的是阿兰·图灵，你肯定过不了 C++ 那一关。

所以不管你去哪家软件公司面试，都要做好心理准备，可能会很倒霉，撞上那么一两个肯定会拒掉你的面试官。面试过程不但会很艰难，最后还会被告知不适合那家公司，这种感觉确实不好受。但是千万不要被这种感觉打败，其实根本没什么大不了的。而且如果你有这种感觉，那其实还不错啦，至少说明你是个正常人。

然后你应该潜心准备 6~12 个月，重新应聘。对待这种错漏的案例，我们（至少我周围认识的人里）也没有更好的办法了。忘记过去，一切重新来过。很多人都是尝试两三次以后才进来的，这些可都是很牛的人哦。

所以你也一样做得到。

好吧，应聘失败好像也不是那么可怕了

非常好！下面我们来聊聊面试的技巧吧。

假如你读得很仔细的话，应该会发现其实我和上面提到的那些面试官都不一样，姑且叫我面试官 D 吧。意思就是我也有我自己偏爱的问题和主题，和别人差不多。尽管我很想告诉你这些问题是什么，但是我真的不能，因为这样一来真的会得罪所有那些和我问的问题差不多的面试官。

我打算帮你准备一些常见的主题，很多类似 Google 的公司里的面试官都会问到这些东西。这些公司会自己编写软件，自己搭建分布式计算系统。其他类型的技
术公司也是有的，另一个极端就是把所有的東西都外包出去，尽量采用第三方软件。我的建议只对那些类似 Google 的公司有效。

这样或许你也有机会来 Google 呢？

首先，我们先来说一下非技术的部分。

热身

拳击赛场上不热身就开打是不可能的。所以你也一样，去面试的时候千万别忘了带拳击手套。等等，不对，不好意思，我是说：事先要热身！

热身要热多久？一般来说热身可以分成长期的和短期的，这两样都不能忽略。

长期热身的意思是：在面试之前要复习一到两个星期。要让自己适应进入在白板前解决问题的“模式”。要是在白板前也不发怵，那其他媒介（笔记本、多人协作文档这些）就更不在话下了。这是必备技能。

短期热身的意思是：面试前一天晚上要休息好，然后当天早上要做一些密集快速的热身练习。

我知道的最佳的长期热身方式有以下两种。

(1) 好好读一本讲数据结构和算法的书。为什么？因为这能强化你分辨问题的能力。很多面试官都很喜欢那些不需要费太多口舌解释就能明白他们在问什么的应聘者。例如，假如他们问你怎么给美国地图上不同的州着上不同颜色的时候，你能立刻反应出这是一个图着色问题的话，肯定会加分不少，哪怕其实你记不清楚图着色到底是怎么回事也没关系。

要是你记得它的算法，那更是能很快给出答案了。所以从准备面试的角度来讲，能够识别出问题属于哪个类型，最好用什么算法和数据结构来解是至关重要的。

史蒂芬·斯基纳的《算法设计手册》就是我准备面试的最爱。他让我认识到图着色问题是多么常见（同时又很重要）——每个程序员都应该弄懂它。这是其他任何一本书都没有做到的。这本书还涵盖了各种基础数据结构和排序算法，这也是很重要的补充。不过这本书下半部分才是精髓所在，它列出了一大串非常有用的问题和解决它们的各种方法，每个问题只用一页，没有解释太多细节。每一页都配有简单的插图，便于记忆。这应该是学习怎么识别问题类型的最佳方式了。

还有一些面试官则比较推崇《算法导论》。这的确是一本经典著作，价值不言而喻，但是恐怕两个星期是读不完的。要是你真的想要“好好准备”面试的话，不

妨考虑暂缓应聘，等读完这本书再说。

(2) 找个朋友来面你。请朋友来随机给你出面试题，你应该要能够立刻在白板上写出来。不要被惰性打倒，坚持写完才能结束。咬牙坚持才是王道。

我第一次面 Google 的时候就没有准备这两方面，结果连我都被自己糟糕的白板编程水平给震到了，毕竟上次参加面试已经是 7 年前的事情了。真的很难！很多熟悉的算法和数据结构都记不起来了，更不要说那些只是耳闻过的东西了。

第二次面试前我按照这套练习训练了一个星期，结果当然要好得多了。个中区别是非常大的。

短期准备就是要尽可能使自己保持警醒，充分热身。不要身体没热就上赛场。读读笔记，做两道题练练手。喝点咖啡：不管你信不信，咖啡确实能让你的思路快一点。这样的热身至少要在开始面试前一个小时进行。把它当作是一场体育比赛、公开演奏，甚至考试来对待：充分热身才会有最出色的表现。

心理准备

不管你是多么出色的程序员，过去的成就有多大，现在都要忘记那些东西，把注意力放在通过面试上。

保持谦逊、开明、专注的态度。

如果你表现得很高傲，别人会犹豫会不会想要和你共事。质疑面试官问题的合理性就是傲慢的典型表现——这点我刚刚已经提到过了，真的是很让人恼火的一件事。还记得我说过你不能教面试官怎么面试吗？如果你是应聘者的话，就更是如此啦。

所以千万不要反问说：“天哪，算法真的那么重要吗？你在工作中真的有用到过算法吗？我从来没碰到过。”说这种话纯粹属于作死。就算你不知道答案，感觉很郁闷、很沮丧，也要认真对待每一个问题。

要是觉得卡住了，不要害怕问问题。有些面试官会因此减分，但是通常也能帮你清除一些障碍，让你的总体表现好看一点，这比沉默僵持半小时要好多了。

在“思考”的时候，嘴里不要发出各种叽哩咕噜的象声词。

不要试图转移话题去问答不同的问题。不要妄图通过长篇大论地讲故事来避免面试官问问题。不要唬弄面试官。你应该把注意力放在问题本身，尽力去完整地解答。

有的面试官不会要求你写代码，但是他们会希望你在回答问题的时候，在白板上写一点代码。他们会用暗示的方式，而不是直接说：“现在你可以去白板上写代码了。”如果你吃不准的话，不妨直接问他们是不是要你写代码。

面试官对代码的要求可以说千差万别。我个人不怎么在意语法的问题（除非你写的东西明显不可能在任何语言里跑得起来，这时我会追问细节，确认你不是来搞笑的，否则那就真的搞笑了）。但是有的面试官对语法是很严格的，哪怕是漏掉一个分号或者大括号都会默默扣你分，他们是不会提醒你的。我认为这种面试官太吹毛求疵，但是他们却觉得自己是很优秀的技术考官，偏偏这种事情还没法讨论，你说头疼不头疼。

所以最好还是要问清楚。问面试官语法是不是很重要，如果面试官很在乎语法，那就尽量写对。要各个角度和距离审视自己的代码。假装这是别人的代码，而你是来挑毛病的。你会惊奇地发现，站在两尺以外看白板，背后还有一个面试官盯着你的时候，自己会犯多少错。

有不清楚的地方是可以问的（而且是鼓励你问的），有时候还可以去和面试官确认自己是不是在正确的方向上。什么都不问，直接就冲到白板前开始写代码有时候是会减分的，就算你的代码是对的也没用。他们会觉得你没有仔细思考，是那种“设计无用”的冲动型选手。所以就算你觉得自己知道问题的答案，也应该问一点问题，在写代码之前说说自己是怎么想的。

不过话说回来，也不要说得太多，免得让面试官觉得你在拖延时间。很多面试官会希望能多问点问题，所以手脚尽量快一点，不然等解答完第一个问题，时间也差不多到了。面试官会因此不给你高分，因为他们没能充分考察你的技能。无罪推定这种事情在面试里可不常见。

最后一个非技术方面的提示：自带可擦马克笔。办公用品店里有那种和铅笔一样细的，而大多数公司（包括 Google）里的都是那种粗头的。用细头的笔等于把白板从一台 480i 分辨率的老式电视机瞬间升级成 58 吋的 1080p 高清等离子屏幕。面试中要利用一切可利用的资源，而空闲的白板绝对是一大助益。

另一个要练习的技能就是白板上的空间管理，例如，不要从右上角开始写代码，一直写到右下角，字还小得看不清。面试官可不会喜欢这种代码。不过尽管看到别人这么做的时候很火大，偏偏我自己在面试的时候也是这么干的，好笑吧。只要尽量留意就好了。

哦，对了，小心别在指指点点的时候让马克笔干掉。别说我没警告过你：在面试的时候最怕的就是被打岔，马克笔出问题的概率那是出奇的高。

好了，非技术的部分就说到这里。接下来我要说 X 的部分了！别捅我！

准备技术面试的建议

最好的建议是：去拿一个计算机科学的学位。越了解这方面的知识，对你的好处就越大。尽管学位不是必须的，但是它确实有用。再高的学位也不是必须的，但对你都是有帮助的。

不过我猜你大概等不了 2~8 年后再应聘 Google 吧，所以这里给出的都是一些短期的建议。

算法复杂度：大 O 是一定要懂的。如果连基本的大 O 复杂度分析都磕磕绊绊，那基本上是肯定过不了面试的。它最多就是计算理论里开头的一章内容，所以一定要去读。你能读懂的。

排序：知道排序是怎么回事。不要写冒泡排序。至少要了解一个复杂度为 $n * \log(n)$ 的排序算法的细节，能记住两个更好（比如快速排序和归并排序）。归并排序在那些快速排序不好用的情况下通常都很有用，所以也应该了解一下。

无论如何，也不要在面试的时候去排序一个链表。

哈希表：哈希表可以说是人类已知的最重要的数据结构。一定要弄明白它的原

理是什么。它也就是数据结构书里的一个章节，去读吧。你要能够做到在一场比赛里，用自己最熟悉的语言，只用数组来实现它。

树：必须弄懂。这是基础，我真的不想说，但是有些人连最基本的怎么构造一棵树，怎么遍历，怎么操作的算法都没搞明白就来面试，太丢人了。你至少也应该熟悉二叉树、n叉树、trie树。树应该是长期热身里最好的练习题资源了。

至少要熟悉一种平衡二叉树，可以是红黑树、伸展树、AVL树。而且必须掌握实现细节。

树的遍历算法也要知道，BFS和DFS，还要了解前序、中序、后序遍历的区别。

可能你树用得很少，但那是因为你在回避它。只要你弄懂了树，就不用再害怕啦。好好学吧！

图：图真的非常非常重要。远超你的想象。就算你已经觉得它很重要，很可能它的重要性还是超过了你的认知。

在内存里表示图的方法基本上有3种（对象和指针、矩阵、邻接表），每种表示都要熟悉，并且知道它们的优缺点。

你应该了解基本的图遍历算法：广度优先搜索和深度优先搜索。知道它们的计算复杂度、优缺点，以及具体的实现代码。

然后还应该根据情况了解一些更高级的算法，如Dijkstra算法和A*算法。无论是在游戏编程中还是在分布式计算中，这些都是非常了不起的算法，所以应该了解一下。

每次遇到问题，首先应当考虑的就是图算法。它们是任何关系最基本、最灵活的表示方法，任何有点意思的设计问题可以说有一半的机会会涉及图算法。只有在你百分之百确定没办法用图算法来解的时候，才能去考虑其他方案。这条建议一定要牢记在心！

其他数据结构

只要脑袋里装得下，就尽可能多地掌握一点各种数据结构和算法。特别是那些

著名的 NP 问题，如旅行商问题和背包问题。在面试官问到这类问题的时候，你要能认出它们来。

你应该知道 NP 完全问题是什么意思。

一定要好好读数据结构书，能记多少就记多少，相信我，你不会失望的。

数学：有些面试官会问一点基本的离散数学。在 Google 似乎这种现象比其他地方都要流行，虽然我自己也不是很擅长离散数学，不过我觉得这算是好事吧。我们身边有各种计数问题、概率问题，以及其他基本的离散数学，而我们周围的数学盲却在用拙劣的方法绕过它们，完全不知道自己在干嘛。

要是面试官真的问你数学问题也不要生气。尽力就好。要是能在面试之前复习（或者自学）一下基本的组合学和概率学，你的表现会出色很多。至少也要熟悉 n 选 k 这类问题——多多益善。

可是时间不够用啊，我懂的。但是是否听从这些建议的区别就是“不好说啊”和“就是她了”。其实离散数学也不是那么难——很多高中里学过又忘了的数学都派不上用场。只要懂小学数学就可以了，所以花个两天时间好好学一下就能捡起来。

操作系统：你只要知道进程、线程、并发这些概念就够了。很多面试官都会问到这些基础内容，所以你应该了解它们。掌握锁的概念，互斥锁，信号量，管程机制的工作方式。知道什么是死锁，什么是活锁，怎么避免它们。知道进程需要哪些资源，线程需要哪些资源，上下文切换是怎么进行的，操作系统和底层硬件是怎么触发上下文切换的。知道一点调度是怎么回事。现在多核正在变得越来越流行，要是你连基本的“现代”（或者说“有时工作有时失灵的”）并发结构也不理解的话，那就真的是史前动物了。

这方面我读过的最实用的书是道格·李的《Java 并发编程》。每一页都是真知灼见。讲并发的书其实很多。不过一般我不会去读学术著作，而是喜欢关注更实用的方面，毕竟这些才是在面试中比较容易碰到的。

编程：至少要熟练掌握一种编程语言，最好是 C++ 或者 Java。C# 也行，反正

它和 Java 很像。一轮面试下来，至少有几场会要写代码。所以一定要对那门语言的细节有相当程度的了解。

其他

尽管我列出了各种规则，但遇到面试官 A 的可能性依然存在，这时再怎么根据我的建议来学习都是没用的（热身的部分除外）。尽力而为吧。最坏的情况也不过就是再等 6~12 个月，对吧？看起来时间很长，其实一眨眼就过去了，我保证。

我上面提到的这些东西其实都是必备的知识：要是这些都不懂，基本过不了面试。离散数学的要求或许没那么严格，但是一点也不懂的话好像也说不过去。剩下的内容都要融会贯通，这样就算是达到面试及格线了。根据不同面试官的风格，面试的难度可能会很高，也可能简单很多。

这真的是要看你的运气如何。你觉得自己是不是有运气的人？来试试看吧！

给我发简历

我一般每周都会把收到的简历整理起来，一口气提交上去。你应该趁着这个时候努力复习！好好热身。实际工作只会让你反应迟钝。

希望这篇文章对你有用。想骂的话就来吧。我先去睡觉了。

作者手记：敏捷好，敏捷坏

这篇文章在当时是我写过的文章里最毒辣的。“敏捷”当时正在渗透 Google，而它也不怎么受欢迎。所以我只用一只手就扼杀它了。

写这篇文章我花了好几个星期的时间来斟酌。问过很多人的意见和措辞，甚至包括（当时还是）敏捷的狂热支持者。所以我真的是很用心在写这篇文章。

在我的文章发表之前，敏捷几乎已经快要成为“主流”了，我的意思是要是你不喜欢它的话，你就会被归为异类。而这种唯我独尊的状态也是很危险的。它的危害可能要好多年才会过去。

正因为我看到了这一点，所以主动发起攻击，以期一击毙命。为此也得罪了一些人。但我不在乎。你要敏捷是你的事情，但你没资格逼我同流合污。

最后我赢了。“我们”赢了。有一些博主也在差不多时候加入我的行列——这也算是一种趋同演化吧，掀起了一场反革命。我们成功传达了一条信息，那就是：你可以拒绝敏捷。这句话可以刻在我的墓碑上。事实上，连“敏捷”那两个字都可以去掉。你永远都有权利说不。

对我们这一代搞软件工程的人来说这是一大胜利。你根本想不到它的意义有多重大。现在，敏捷已经被戳穿得差不多了，咨询师们基本上已经放弃了这个词，开始用更时髦的“精益”等词汇，妄图救回这棵摇钱树。但敏捷的热潮确实是从我的文章发表以后开始消退的。是我给了它致命一击。百足之虫，死而不僵，在投入了那么多人力物力去推广行销之后，真的要看它死掉还是要花点时间的。但那已经无可挽回，只是时间的问题罢了。

正所谓笔端可挽千钧力，要是真的有激情，就为它写文章吧！

敏捷好，敏捷坏

“争球（Scrums）是橄榄球里最危险的词，因为摔倒或是动作不当可能会导致前排运动员受伤，甚至可能会扭坏脖子。”——维基百科

在我小的时候，胆固醇还是个坏东西。大家都知道。脂肪，不好。胆固醇，不好。盐，不好。所有的东西都不好。但是现在胆固醇里也分好坏，好像我们能区分它们似的。这种转变非常奇怪，就好像食品药物监督管理局突然发布声明，宣布老鼠药其实有两种：一种是好的，另一种是坏的，你应该吃很多好的老鼠药，不要吃坏的老鼠药，而且绝对不要混在一起吃。

差不多在一年前，我对所谓的“敏捷”编程还只是抱着相当单一的看法，觉得这玩意儿基本上又是某种愚蠢的营销骗局，是专门忽悠那些从来没读过《没有银弹》

的菜鸟程序员的科技病毒。那种会去延长保险期，会买自助教科书，相信老板真的关心他们，把他们当人看的程序员，那种专门去各种研讨会认识朋友，不知道怎么在机场躲开和挥舞小本本的狂热分子眼神接触的程序员，还有那些真的相信在小卡片上乱写一通就能瞬间简化软件开发的家伙。

这些人都是傻瓜。只有这个词可以形容他们。我体内坏的那部分胆固醇告诉我，敏捷方法论就是忽悠那些傻瓜用的。

不过最近我有了很多机会观察不同风格的敏捷主义，现在我觉得这种看法只有九成是对的。其实还是有好的敏捷的，只不过我花了很长时间，拨开各种对scrum等敏捷流程狂热顶礼膜拜的迷雾后，才看出这一点。现在我应该看得很清楚了。

欢迎你们来参加我的研讨班，只要 499.95 美元，绝对低价！哈哈哈，傻瓜！

好啦，开个玩笑。研讨班里只有坏的敏捷。要是将来有一天你发现我摇身一变，打着敏捷咨询师的头衔到处招摇撞骗，让无知群众来交钱听我对敏捷开发的深层思考和理念的话，你有权来戳穿我的西洋镜。要是我借口说我只是在说笑，告诉我说我告诉过你我会那么说。要是我辩解说自己其实是泰勒·德顿，命令你不得对我不利，告诉我说我肯定说过自己会说这句话，然后毫不留情地干掉我吧。^①

现在我会告诉你好的敏捷是什么样子的，免费哦。

要把好的敏捷和坏的敏捷孤立起来讲是很难的，所以可能会把它们放在一起说。不过放心，好的那种我会用一只快乐的小老鼠来标注，而坏的那种则会用一只悲伤的死老鼠来标注，这样你就能分清楚啦。

坏的那种

(死老鼠)

远古时代，大多数公司都会采用这样的软件开发流程：

- 雇一堆工程师，然后再雇用更多的工程师。

^① 译注：参见《拳击俱乐部》。

- 幻想一个项目出来。
- 定一个打算发布的日期。
- 指派一些工程师开始干活。
- 不断催进度，要么最后项目发布，要么全体阵亡，要么干完的同时阵亡。
- 然后或许可以搞一个廉价可怜的小聚会。这一步不是必须的。
- 然后重新来过。

谢天谢地，你的公司应该不是这样的吧？好险好险！

有趣的是，这也是非技术公司（比如克莱斯勒）开发软件的方式。只不过他们没有雇工程师，而是临时招募了一堆软件咨询师，然后丢给他们一份为期两年的项目文档，要求他们不但要按时完成，还要在签署合同后接受客户拍脑袋想出来的各种变化和修改。结果最后做得一塌糊涂，没人愿意付钱，弄得所有人都不开心。

于是有些咨询师开始琢磨：“嘿，要是这些公司坚持这么幼稚的话，那我们就应该把他们当小孩子！”他们也的确是这么做的。当甲方说“我们要一个 A 到 Z 的特性”时，咨询师就会在这些大大的索引卡片上写下“A”，然后再在第二张上写下“B”，依次类推，每张卡片上还有一个时间估算，最后把它们都贴在墙上。每当客户要加什么东西的时候，咨询师就会指指墙壁说：“没问题，伙计。你打算换掉上面哪张卡？伙计？”

难道没有人质疑过为什么克莱斯勒最后取消了项目吗？（见：<http://zh.wikipedia.org/wiki/%E6%9E%81%E9%99%90%E7%BC%96%E7%A8%8B#.E5.8E.86.E5.8F.B2。>）

有一天，这帮丢了大客户的咨询师们正在酒吧里消磨时间，其中一个（名叫罗恩·贺伯特）^①说道：“这种按代码数量收钱的办法太烂了。你们知道怎么才能赚大钱吗？自己开宗立派才是王道。”于是极限编程和科学教诞生了。

很快，人们就发现所谓的极限编程完全是胡说八道。就拿结对编程来说吧，这

① 译注：罗恩·贺伯特是科学教派的创始人。

算得上是极限编程里最夸张的一项了。敏捷大师都不喜欢谈论它，现实是：根本没人这么工作。它背后的理念是：“如果一个程序员坐在屏幕前能写出好代码，那么10个程序员肯定能做得更好，因为越多越好嘛！可惜大多数屏幕前最多就能挤两个程序员，不然坐着不舒服，所以我们就叫它结对编程吧！”

这你得原谅他们。这么多年来，和他们打交道的那些公司基本上都是学龄前的智商，时间一长真的会拉低一个人的水平。

可棘手的地方在于，病毒很难杀干净，特别是那种渗入基因里的东西。当所有人都接受敏捷那套东西后（因为大家都希望工作更有效率），要承认失败的代价是很大的，太丢人了。于是就出现了一些新的敏捷“方法论”，他们宣称尽管那些方法论行不通，但是他们的理论是可行的！

你自己去看看他们的网站吧 (<http://www.controlchaos.com/>)。告诉我这些玩意儿不是电视导购。来，试试看。光是看一眼就够丢人了。（作者注：6年了，现在看还是很丢人。）

由于巴纳姆效应的存在，这帮人还是迅速骗到了不少钱，就像科学教一样。这不是他们的错。有的人就是不把自己的钱当回事，更不用提什么尊严了。

只要观察一下常见营销法则，我们就能明白敏捷方法论有多么不靠谱：

- 任何自称“方法论”的东西基本上都是愚蠢的。
- 任何需要“传道士”或是喜欢办研讨班的东西，都是专门骗钱的。
- 从来不提竞争对手或是替换方案的东西很可能都是自私自利的。
- 一般来说，缺乏数学细节的图表都是愚蠢的。

这里所说的“愚蠢”，是指“专门针对笨蛋的那些了不起的营销手段”。

无论如何，咨询师们得以继续在各种路演上挥舞光鲜亮丽的小册子。我觉得他们一开始的目标只是大公司，反正只要签订灵活的合约，让他们可以不断地在“两个星期”内发布“不管什么东西”，直到客户破产就好了。不过我也觉得应该不会有那么多笨蛋客户会答应签这种合约。

这就是那些咨询师开始跑来忽悠你的原因。为什么不打入公司内部，直接向开发人员推销呢？很多公司都采用了我上面提到过的那种催进度的开发流程，要是能说服中层经理和技术主管相信这种低成本的方式能救他们于水火，那肯定会有买账的。

我的朋友啊，这时开始，他们这些人就从“无害的小丑”变成了“潜在的危险”，因为在这之前，他们只不过是忽悠一下那些连自己开发软件都不会的土豪公司，而现在却可能给我们身边的经理洗脑。大多数情况下，我们对这种尴尬的状况束手无策：一个本来好好的经理现在却深受其害，挥舞着极限编程的书和索引卡，滔滔不绝地鼓吹这种新形式的官僚主义对提升团队生产力有多大帮助。

我们怎么知道它没有改善生产力呢？这个问题不好答。因为要是很明显的话，它的谎言早就被揭穿了。但是软件工程师的生产力本来就很难衡量，个中原因大家都明白。要科学验证软件开发就更是天方夜谭了。你不能让同一支团队把一个项目做两遍，因为第二次的时候，很多因素都会发生变化。你也不能让两支团队去做同一个项目，因为有太多不可控的因素在里面，而且尝试的代价太高，谁也负担不起。让同一支团队接连去做两个不同项目也不能算是有效的实验。

最好的办法可能是从很多团队做的很多项目里收集一些统计数据出来，然后看看能不能发现一些相似的地方，然后做一点回归测试，以期发现一些有意义的相关性。但是这些数据要从哪里来？就算公司有保留这种资料，它们也不会给你看的。况且大多数公司也不会保存，它们只会掩盖自己计划的失败，然后乐观地开始下一个项目。（作者注：很多敏捷脑残粉都试图说服我说，有人曾经证明敏捷在某个真正科学的实验里是“行之有效”的。他们在说话的时候，眼睛都是望向别处的。）

既然没法进行实验和证明，那么它的科学性就值得质疑了。这就是这个问题很难回答的原因。食物盲从现象如此大行其道也是这个原因。人们从心里希望那些减肥食品真的有效，老实说，连我都这么期望。比如隔壁乔家的吃了这个玩意儿以后，成功减掉 35 磅这种压根毫无统计意义的个案，会让想要减肥的人听说以后想：“反正试试看也没坏处啊。”

每次我听到别人想要在自己的团队里尝试敏捷方法论的时候，用的也是这个理由。这可不是巧合。

但是只写坏的敏捷肯定效果不好。就好像不管你 how 吐槽科学教，怎么驳斥食物盲从，谁知道对方有没有听进去呢。消灭这种文化上的东西比戒烟要难多了。我知道是因为我都经历过。想要产生正确的影响，一定要同时另指一条明路，我以前吃过亏就是因为当时没方向，表达不出来。

坏的敏捷里，最大的问题之一就是它把非敏捷的开发流程武断地分成两类：瀑布式和牛仔式。大家都知道瀑布式不好，这基本上已经是公认的了。那么牛仔式编程又如何呢？敏捷大师们将之定义为“团队里的每个成员都按照自己认为的最佳方式来行事”。

难道开发流程就只有这些了吗？牛仔式编程一定是不好的吗？敏捷大师们的口气听起来好像是显而易见的，但是除了默认它是“混乱的”外，又说不出具体怎么不好，为什么不好。

去年我有幸同时观察到了好的敏捷和坏的敏捷，我问了这些团队以及各自的技术主管（同时采用好的和坏的敏捷）很多问题：效果如何，感觉如何，具体流程是怎么样的。我确实很好奇，一方面是因为我去年圣诞节的时候就答应说要尝试一下敏捷（“嘿，反正也没坏处”），结果因为和一个组员争论到底什么数据可以放到索引卡上而闹得不欢而散，最后不了了之。另一方面，其他组的朋友似乎被这种玩命似的冲进度搞得精疲力竭，这种事情在 Google 可不常见。

所以我花了整整一年的时间去深入地观察学习。

好的那种

（快乐的小老鼠）

我先来聊一聊 Google 的开发流程。我说得肯定不完整，但是对今天的讨论来说应该足够了。我在 Google 待了有一年半了，虽然当初花了点时间，不过我想我已经基本弄明白了。我还在学习。所以这里分享的都是我目前了解到的东西。

从大局上来看，Google 的流程在那些有着比较传统的软件公司出身的人眼里的确很混乱。新人加入的时候，首先引起注意的就是：

- 经理也至少有一半时间在写代码，所以他们更像是技术主管。
- 工程师可以在任何时候换组，或者换项目，不会有人质疑你。只要一句话，第二天就会有人来帮你把东西都搬到新的组那边去。
- Google 的理念是不去告诉工程师要做什么，他们也是这样严格执行的。
- 鼓励工程师花 20% 的时间（这里指的是周一到周五，早上 8 点到下午 5 点，不包括周末或是私人时间）去做任何想做的事情，而不是日常工作。
- 很少开会。我估计一个工程师平均一周大概开 3 次会吧，这还包括了和主管的一对一沟通。
- 安静的环境。工程师或单独，或三五一组，都非常安静地专注于自己的工作。
- 没有甘特图，或是日期—任务—负责人的表格，或是任何看得到的项目管理工具，反正我没见过。
- 就算真的碰到项目吃紧的时候（其实很少），大家还是会去吃午饭和晚饭，都是免费的哦，味道也很好（这点还是很出名的），除非自愿，没人会长时间拼命工作。

当然了，这些说得还是很笼统的。资深一点的员工肯定会有不同的看法，就好像我对亚马逊的看法也不是完全客观一样，毕竟我从 1998 年开始就在那儿了，遥想当年还是很了不起的。不过我估计大多数 Google 员工还是会基本同意我的总结的。

那么这样的形式怎么行得通呢？

很多人都这么问我。老实说，连我自己都这么问过自己。那么到底是什么促使工程师去解决那些麻烦的项目，去对付那些满是 bug 的运维噩梦的呢？要是工程师可以任意选择自己想做的事情，他们又是怎么和公司的目标保持一致的？最重要的那些项目是怎么招募到充足合适的人才的？为什么工程师不会肥到堵塞防火通

道，逼得救火员要破墙救人？

我先回答最后一个问题，然后再解释其他的。简单来说，我们有一种 Noogler Fifteen 的说法，这个名字取自 Frosh Fifteen：就是说很多大学新生初来乍到，踏上这片压力和比萨之地后，体重都会增加 15 磅左右。Google 的解决办法就是在防火通道里加润滑剂。

至于剩下的问题，我想大多数的答案都是类似的。

首先，应该也是最重要的，Google 通过利益来驱动行为。通常做比较重要的项目所得到的奖励比做那些不那么重要的项目要来得高。你当然可以选择去做一个研究性质的，看起来很遥远的项目，或许永远也无法实用化，但是它本身必须是有价值的。要是最后证明你对了，其他人都错了（创业公司的梦想），你的小项目确实产生了巨大的影响，那么你肯定会得到奖励。

具体的奖励和激励形式多种多样，这里是说不完的，经济上的奖励小至礼品卡和按摩券，大到巨额奖金和股票奖励，这里我不想给“巨额”下一个精确的定义，不过以 Google 的规模，再加上一点想象力，你也能猜个大概吧。

另外还有其他的奖励。其中一个就是 Google 以同行评审为导向的文化，赢得同事们的尊重可谓意义重大。我个人认为比其他地方都更重要。一部分原因是因为文化就是这样的，一开始是人为，后来渐渐就变成了习以为常的东西。另外，你的同事都是很聪明的人，能赢得他们的尊重可不是件容易的事。你的年终总结也是以此为依据，所以对个人收入也算有间接的影响。

还有一个就是每个季度，公司都会雷打不动地集会，向大家展示每一个发布的项目，并且打上名字和头像（总是很小），大家都会鼓掌欢呼。光是想想也会让人兴奋不已。Google 对产品发布非常认真，我觉得做出了不起的东西并且得到认可是这个公司里最强大的驱动力。至少对我来说是这样的。

激励并不止如此，我还可以列出很多。对外人来说，各种奖励、自豪感、归属感等梦幻般的一切都太不真实，让人觉得招聘官完全是在忽悠，这个世界上怎么可能有公司对所有的员工都那么慷慨，真的是所有员工哦，甚至连清理厨房的阿姨们

都会得到“Google 厨房员工”的 T 恤和套头衫。

地球上应该找不到第二家这样的公司了。在 Google 工作有多爽，我三天三夜也说不完。因为每周都有新花样，新福利，新变化，新的调研，来问我们怎么才能让大家在 Google 过得更舒服。

当然我也可能是错的。每个季度在大屏幕上看到自己的名字和头像或许并不是最大的动力。驱使大家去做正确的事的动因是感激，这超过了所有的因素，甚至比所有的因素加在一起还要多。你会忍不住想要做到最好，因为 Google 对你的照顾无微不至，让人觉得好像欠了它似的。

动力这个话题就说到这儿吧。你应该了解了吧。我想，至少大致概念有了。每次有朋友（所有的朋友，不光是我之前工作的地方的朋友）问我在 Google 工作的感觉如何时，我的感觉就像是，你刚刚出狱，而你的狱友（假设全是幼年被判刑，没见过什么市面）写信问你“外面的世界怎么样”时的那种感觉。你会怎么说？

我会说，还可以吧，马马虎虎，总的来说还不错。

虽然以奖励为本的文化确实是一个很重要的因素，但它只能驱动工程师去做“正确的”事，却无法做到高效。所以下面我打算说说他们是怎么做项目的。

新生特质 vs. 鞭子

项目管理的基本理念就是要带领项目走向完成。这是一个公开显式的过程，需要全程呵护：这离不开有力的领导，组织性，纯粹的意志力，你要做到原本不可能自己发生的事情。

项目管理的风格多种多样，从轻量级到重量级，但是它们都有相同的特质，那就是对团队施加外力。

可 Google 却是完全相反，项目启动是因为它处在系统的基态。

在继续之前，我必须承认，我要说的东西很强大，而且并不完全属实。我们并不是没有项目经理、产品经理、人事经理、技术主管等职位，但是他们在系统上所需的精力比其他业界都要少得多。说起来，它更像是时不时地轻推一下，而不是持

续不断地推动。当团队需要更大的推动力的时候，管理高层就会介入，这一点和其他地方是一样的。但是这并不是持续的推动。

不过 Google 是一家有礼貌的公司，所以不会有人大喊大叫，也不会有哀嚎、咬牙切齿这种事，更不会有事态升级，相互推诿指责，或是其他公司的高层经常大吼的各种繁文缛节。霍布斯告诉我们，组织是领导人的影像，这一点大家都知道。Google 的高层都是温文尔雅的人，所以大家自然也都是如此。

我之所以说项目启动是 Google 内部生态系统自然而然所产生的状态，是因为他们花了大量的精力把人引向那个方向。我刚刚也说过，你需要的一切都已经帮你准备好了，所以你可以安下心来，全心投入那些 Google 喜欢的事情。

所以项目启动自然也就成了系统里的新生特质。

这样一来就不需要那一大堆项目管理里常见的理念和方法了：那些对付偷懒的人，夸大估算，迫使大家在设计上达成一致的方法等。开会不用像打仗一样，也不再需要进度报告。因为大家已经有动力去做正确的事，也会一起合作。

Google 所采用的项目管理的技术更像是润滑剂而非汽油：他们是为了让项目进行得更顺利，而不是迫使项目向前进。这里有很多会议室，很多开放空间让人们交谈。团队总是围坐在一起，所以结对编程自然会在需要的时候发生（差不多有 5% 的时间），而非刻意进行。

Google 知道就算是在安静的公司里，一天的中午是最容易被打扰的，所以很多工程师都会选择很早来上班，或是待到很晚，方便自己有时间专心编程。所以会议都安排在中午，把会议安排在上午 10 点之前或是下午 4 点半以后都是很罕见的。在这个时间段外安排会议等于占用了工程师真正干活的时间，所以没人会这么做。

Google 并不是唯一这样管理项目的公司。仔细想一想，其实还有两个地方也是这样的：创业公司和研究生院。Google 就像是创业公司和研究生院的结合体：一方面，它好像我们要动作快，先把东西做出来，越简单越好，然后在慢慢完善的创业风格。另一方面，它又很放松，很低调，我们面临的问题很难，从来没人解

决过，这不是百米冲刺，而是一场马拉松，需要全心全意地专注，拼命开会不顶用。两者的交汇，创业公司和研究生院都是创新的沃土，参与者对结果都负有巨大的责任感。

这并非原创，但令人惊艳的是 Google 在那种规模下做成功了。

能做到这种规模绝非偶然。Google 对待这个问题非常认真，他们知道在这种规模下什么情况都有可能发生，所以他们时刻都保持着警惕。正是这种居安思危的态度，才让他们能在壮大的过程中保持生命力和生产力不至衰退（甚至有所改进）。

从软件工程的角度来讲，Google 是一家非常有纪律性的公司。他们对待单元测试、设计文档、代码审查的认真态度，超过我所知的任何一家公司。他们努力保持井然有序，有严格的规章制度，防止某些工程师和团队肆意妄为。如此一来，整个代码库看起来整齐划一，如出自一人之手，所以换组和代码共享也要比其他地方容易得多。

工程师需要称手的工具，所以 Google 很自然地会聘请最好的人才来打造自己的工具，而且只要工程师有意向，就会（通过各种激励）鼓励他们去做这方面的工作。如此一来，Google 的工具堪称世界级水准，而且还会越来越好。

这样的例子还有很多。Google 的软件工程学实在是太了不起了，我几天几夜都说不完。总之最重要的就是做到那种规模（不光是技术层面，还有组织层面）绝非偶然。只要你适应了 Google 的风格，一切就会变得异常简单——当然啦，这是和一般意义上大多数公司的软件开发相比。

日程表的暴政

还有一点。最后我想说的是日期。传统的软件开发几乎毫无例外都是以日期为导向的编程。

创业公司受制于投资人和预算，日期一到必须有个交代。大客户则能给咨询师们设置目标日期。销售和产品经理会根据对市场情况的评估来设置日期。工程师会根据过去的经验来估算日期。所有的估算都是带着过于乐观的有色眼镜进行的，好

了伤疤，谁还记得痛。

所有人挑日期的时候都是随便说的。“这个看起来要3个星期。”“要是能在第4季度开始的时候上市就好了。”“大家加把劲，明天之前把它赶出来吧。”

我们行业里大多数人都是被日期催着走的。永远有下一个里程碑，永远有一个最后期限，永远有一个限定期日的目标。

我能想到的例外大概只有：

1. 开源软件项目。
2. 研究生院的项目。
3. Google。

大多数人都默认你会定下一个日期。就算是我最欣赏的软件项目管理著作《人月神话》，也都假设你要定下估算。

如果你习惯了事先宣布自己的软件，那么大众通常就会想要知道一个时间表，这就隐含了日期。我想这就是Google通常不事先公开自己的产品的原因吧。他们是真的明白软件开发这种事情就和烹饪、生孩子一样是急不来的。

如果上面列出的3个例外不受日期所限，那么又是什么在驱使它们呢？从某种意义上来说应该是对创新和想要做东西的渴望，所有出色的工程师都有这种欲望。（我们行业里有很多人只是想要“混口饭吃”，这种人回家以后就不会再想工作的事了。开源软件得以存在，正是因为这个世界上有比这种人更有激情的人。）

不过这还不是全部：光有创新的欲望是不够的，因为它的方向感和激励不一定够。Google肯定也是受制于时间的，它也想要尽可能快地做成一件事。外面不但有很多虎视眈眈的竞争对手，同时还要满足饥渴的投资人不断增长的需求，就连我们自己，也会有一些在有生之年希望能够完成的长远计划和目标。

只不过区别在于，Google不蠢，也没有自以为是到可以宣称知道一件事应该要多久可以完成。所以在公司里，我唯一知道的日期就是每个季度末，因为大家都想挤上那块大大的产品发布的屏幕，希望得到掌声、礼物、奖金、团队旅游等

各种只要发布能对 Google 产生重大影响的产品就能得到的好东西。

而除此之外的日子就像流水一样，所有人都在以自己的最佳状态工作，当然人跟人是不同的。每个人对工作生活的平衡点定义都不一样，Google 则可以让你选择任何合理的平衡点，并作出成绩来。最优的生产力离不开培训，Google 为你提供了大量的培训，每周都会有来自内部和外部的讲师做技术演讲，这些演讲都会永久存档，任何时候都可以去看。Google 会提供给你任何完成工作所需要的资源，或是学习完成工作需要的知识。最优的生产力和工作用的机器环境也有关系：代码的质量，工具，文档，计算平台，团队，甚至是每天的时间（有东西吃，尽量避免打扰等）。

你要做的就是给工作排排序。什么？数学？我这里有大把：把软件开发映射到排队理论上去。这可不是什么八杆子打不到一起的东西。我们行业里很多人都发现公司组织的模型其实和软件模型区别不大。

有了这样一个工作队列（当然这是一个优先队列），你瞬间就获得了大部分敏捷方法论所宣称的那些神奇的好处。而且说实话，这比写在一堆索引卡片上要实在多了。要是你还不信，那我只好把你的索引卡片都偷偷藏起来了。

这个优先队列就好像是一个垃圾堆放场，随着项目不断深入展开，你可以把大家的各种想法（包括 bug）排列进来。除非这个队列里没有任务了（表示项目可以发布了），否则工程师不会没事做。只要将一个任务放在队列里，附上合适的注解和文档，就知道它是被暂停了还是恢复了。项目还有多少工作量一目了然，愿意的话，还可以根据剩下的任务进行估算。你可以检验已经完成的任务，推断出回归率、个人生产力（如果有必要的话）等属性。你可以看到哪些任务经常被忽略，以此找出组织内部的问题根源。所有人都看得到工作队列，所以重复劳动的可能性很小。

它的好处还有很多很多。

但可惜，工作队列对研讨班和峰会来说不够华丽，缺乏魅力。听上去就像是一堆工作，因为它本来就是啊。

再议坏的敏捷

前面我大致勾勒了公司开发软件的方式，它既不是敏捷方法论，也不是瀑布模型，更不是牛仔式编程。它就是普通的“敏捷”：Google速度很快，反应也很快。

那么要是在这样一个优秀的软件开发流程之上再套一层敏捷方法论的话会怎么样？你可能会想：“反正也没什么坏处吧！”我去年也曾经这么大胆设想过一次。

简单来说：其实是有害处的。首当其冲的就是，选择敏捷的技术主管和经理往往对现实缺乏足够的认识。坏的敏捷会在各个层面上伤害团队。

首先，坏的敏捷会以最糟糕的方式关注日期：短周期，快速发布，不断的估算和重新估算。周期长至以月为单位（这大概还勉强可以接受），短至以天为单位，这是最糟糕的。这完全是理想化的世界。

而在现实世界里，项目的每个参与者都是人。人的状态是有起伏的。有时候你会精力充沛，一口气写 18 个小时的代码也没问题。有时候却心不在焉，不想写代码。有时候则会感到疲惫不堪。每个人的生物钟和生物节律都不一样，几乎无法控制。要是团队按照天或者半周为步调，我们很可能无法与之协调。

此外还有私人物：有些和工作无关的事情会在工作时间突然冒出来，需要你去处理。

这些因素都不在坏的敏捷的考虑之列。就算完成一个大任务后仍然处于兴奋状态，你也不能继续拼命写代码。因为你要为下一次冲刺保存精力，所以只能放慢节奏。于是这种不一致硬生生将出色的工程师逼成了平庸之辈。

此外还有所谓的业余时钟：就是那些在主要工作之外你想完成的事情。这通常是一些重要的清理工作，或是其他一些最终能改善整个团队生产力的事情。坏的敏捷对这种事情特别不友好，常常会在一个大的里程碑之后留出一大块时间，让大家去做这种工作，完全不考虑他们当时的状态如何。坏的敏捷只把注意力放在眼前的目标上，这对创新是有害的。虽然他们为自己留出了清理代码的时间，但是却不会无私地去帮助公司里的其他人。毕竟要是你只是机械地照章办事的话，怎么会想到去做别的事情呢？

不知道为什么，早起的人似乎都很喜欢坏的敏捷。我觉得“天没亮就醒”，“喜欢静态类型，讨厌类型推导”，“连上厕所的时间都要安排好”，“喜欢开会”这些性格特点和“喜欢坏的敏捷”之间肯定存在某种神秘的联系。我说不清楚，但是却发现这种情况很常见。

大多数工程师都不喜欢早起。我知道有一个组每周至少有一次 8 点的早会（可能不止一次）。结果他们像僵尸一样坐在那里，对着 E-mail 发愣，一直到中午。然后他们都回家睡觉去了。到了晚上他们会再回来工作，但个个都是熊猫眼，好像永远醒不过来似的。我和他们说话的时候，他们也不会心情不好，只是很少会说整句。

我私下问他们觉得敏捷怎么样，他们的回答是“好像有点用，但是我觉得好像违反了某种工作守恒定律……”，还有“我不知道，我想我们在尝试吧，不过老实说我看不到价值”等。他们都是新人，所以不太敢说话，也没人确定到底是敏捷的问题，还是公司就是这样的。

朋友，那不是“敏捷”，那只是一堆垃圾。只要你的老板是个容易上当的笨蛋，你就会落入这种处境。

好的敏捷应该放弃这个名字

时刻警惕下面这两种主张：

- “他描绘的所有好东西都是真正的敏捷。”
- “他描绘的所有不好的东西都是执行上的问题。”

你会不断地听到这种论调。我读了很多关于敏捷的书（所以才能看穿这玩意儿的真身：病毒），也读过很多对敏捷的批评。敏捷借助上面两条那样的诡计来躲避批评：功劳都是我的，坏的都是你们做错了。

如果 90%（甚至更高）的情况下，本意良好的聪明人还是做砸了，那么这个流程就是行不通的。推卸责任的次数是有限的。

我担心现在“敏捷”这个词已经变得太沉重，任何优秀的程序员都应该彻底避开它。我已经解释了“敏捷编程”的两种形式，其实还有第 3 种（高端、大气、上

档次)试图通过技术来提升生产力(比如灵活性)的风格。这种书的名字一般都是《Ruby on Rails 敏捷开发》、《敏捷 AJAX》、《敏捷 C++》之类的。在我看来,这些名字本身没什么太大的问题,但是它们实在是有点滥用“敏捷”这个词了。

坦白说,市面上大多数的敏捷其实都是坏的敏捷。

如果我是你,我就会去掉自己简历里的敏捷字样。我会默默合上讲 SCRUM 和 XP 的书,锁进柜子。我会把要干的活都放到 bug 数据库或者其他工作队列的软件上,丢掉那些索引卡片。我会尽快把敏捷从公司里消灭干净。

然后我会把精力放在真正的敏捷上(而不是什么方法论)。

这只是我个人的观点,而且现在都已经早上 4 点了。你可以有你自己的结论。无论如何,我都不觉得我明天会很早起床。

哦,对了,差点忘了免责声明:我的观点不代表 Google。这些只是我自己的观点,他们看到这篇博客的时候会和你一样惊讶。只不过我希望这更像是“生日惊喜”,而不是“在野外惊到一只犀牛”的那种。走着瞧呗!

作者手记: Google 能保持领先吗

这篇文章从来没有在 Google 以外公开过。原本是专门写给 Google 内部员工看的。这算是那篇吐槽平台的“前传”吧。这是 11 篇连载的第 1 篇,我从 9 个完全不同的角度解释了 Google 今天的策略为什么是错的。吐槽平台的那篇是第 2 篇,剩下的我会慢慢写。

结果吐槽平台的那篇文章不小心发到了外网,引起了轩然大波。我差一点就丢了工作——不是因为吐槽,而是因为那篇文章很容易泄露公司内部的保密资料,毕竟原本我只打算发在内网里的。那次真是险过头了。不过大众的反应确实有点过了(老实说我也没料到),所以连载还是先等一等再说吧。

刚刚说过,这是连载的第 1 篇。我要再次声明,文章只代表我个人的观点——

因为我并不认同 Google 的其中一条关键的策略，所以这显然是我自己的观点，和 Google 无关。公司内部是欢迎不同意见的。他们也会选择无视你，我觉得这样也挺好。我只是希望在他们最终意识到问题的时候，我有一点“早就告诉过你了”的证据。

我没有申请公开这篇文章的许可（把它放在这本书里）。所以我有可能会因此被炒鱿鱼！希望这样会让你们有多点期待。也可能不会吧。这些东西对大家来说应该都是显而易见的。

我很喜欢在手记里引用杰米·扎温斯基的话。他总是精辟地说出我想要说的东西。他在自己的博客上这样写道：

Google 的声明显然非常可笑，我来告诉你为什么。“支持”化名的方法应该是：

1. 放弃仅仅因为怀疑人们没有使用真名就删除他们账号的做法。
2. 没了。

唉。真希望有一天我能像 JWZ 那么牛。我一直这么告诉自己。有目标总是好的。

希望有一天我能鼓起勇气，有动力去完成整个连载系列，外加两篇总结。总结里不会有新的内容，但是要确保整个系列完整——希望能在职业生涯里完成。

总有一天。

要是写不出来，我就自己动手了！走着瞧吧！

Google 能保持领先吗

在 Google，我从来都不怀疑我们保持领先的能力。

我们确实犯过错，但是这些小插曲从来都没什么好担心的，因为我觉得 Google 一直牢牢掌握着世界的脉搏。Google 从未和社会脱节，也从未变得曲高和寡。

每当人们谈及 Google 的时候，不管是在超市、机场，还是足球场，言语之间

总是透露出对 Google 的信任感，即便他们不见得明白或赞同 Google 在做的东西。

之所以 Google 能得到这种尊重，是因为尽管时不时地会犯错，但 Google “知道自己在做什么”。大家都明白，Google 做的东西和我们是息息相关的。

然而微软却非如此。他们在 20 世纪 80 年代曾经非常了不起，但是自 20 世纪 90 年代起就渐渐离开了大众视野，2003 年左右更是完全不见踪迹了。唯一的例外就是 XBox，在这方面，微软和索尼的关系就像是百事可乐和可口可乐一样，除此之外，微软已经完全和大众脱离了关系。哪怕是不懂技术的人，只要看一看微软的产品和服务，从 Bob 到 Zune，再从 Bing 到 Kin，就能马上得出微软的领导层“完全不知道自己在干嘛”的结论。

微软今天能做的就是去复制别人的东西，然后拼命追赶。可是他们对自己杀进去的领域根本就是一知半解。而在略懂皮毛的情况下盲目追赶是很困难的，所以他们的东西通常都做得很烂。结果呢？他们不但失去了触觉，连自己的老本也不断被苹果、Google、亚马逊、Facebook 等其他知道自己在干嘛的公司抢走。

我曾经为这样沉沦的公司工作过。这种事情难免会发生，比如我曾经供职的 Geoworks。15~20 年前，我们有 20 多人在那儿上过班，现在都跑来 Google 了。Geoworks 一度是家游戏公司，后来转型做操作系统，后来又转做智能手机。Geoworks 和诺基亚、夏普、松下，还有惠普这样的公司合作过，曾经在智能手机市场上小有成绩。

然而到了 1993 年，网景公司和互联网的出现一下子创造了自工业革命以来最大的市场契机，Geoworks 却生生错过了它。当时公司里有上百名工程师看到了这个巨大的机会，还在员工大会上问当时的 CEO 高登·R·艾利文说为什么我们什么都没有做。他嘟哝着说了一些“不是我们的核心业务”的话。高登完全不“懂”互联网。在接下来的短短几年间，Geoworks 失去了大量重要的合作伙伴，被股票市场除名，最终在 2001 年前后被彻底拆分。

我知道这听起来像是不着边际的历史课。但是在 Geoworks 的经历告诉我，再强的公司也很容易迷失方向，并一蹶不振。DEC，惠普康柏，IBM，甚至微软都逃不出这个命运。

我在亚马逊待了 7 年，它也曾经差点迷失方向。杰夫·贝索斯有一次在员工大会上提到，各种办公用品、书籍、影音制品都可以数字化，所以也意味着很容易被盗版。利润越来越低，很快这些东西就不再产生任何收入了，贝索斯如是说。然而轻工业制品（比如服饰和电子消费品）的周期都很短：连烤炉这种东西，也没人想要去年的型号。而且轻工业制品往往意味着复杂的供销问题。在贝索斯和我们分享这些的时候，亚马逊的竞争对手 eBay 已经快不行了，现在看起来连亚马逊自己也岌岌可危。

贝索斯用尽了一切力量来防止自己失去敏锐。过去 10 年来，我们看到他们发布了很多产品，有 Kindle、Mechanical Turk、AWS 和 EC2/S3。但这只是开始。贝索斯不愿意失去这种敏锐的地位，到目前为止，他已经成功地将亚马逊的触角伸到了很多全新的领域：很多专家都认同这种业绩可以说是前无古人的。说实在的，我感觉只要贝索斯能保持这种势头，最后亚马逊这个品牌会彻底翻新。他从来都不擅长运营或是保留员工，但是他对市场的理解确实很有一套。

有时候人们称这种寻求新的生存之道为“存在危机”。微软员工来我们这儿面试的时候，经常会用到这个词，频率之高令人有点意外。

正因为我目睹了 Geoworks 和微软失去了触觉，从此一蹶不振，也眼见了亚马逊随着利润空间的消失而苦苦挣扎，所以我对这种事情变得特别敏感。为一家深陷存在危机的公司工作一点也不好玩。他们往往会立即转入追赶模式，拼命压迫员工去完成这个或者那个新产品，因为这将是他们卷土重来的大好机会。

由于对这种事情的超极敏感度，我在朋友中间特别以知微见著闻名。往往水只升到脚踝，船长跟我拍胸脯说没事的时候，我就知道该撤了。

所以老实讲：我对实名制这个东西非常有顾虑。尽管现在水只到脚趾，但是我曾经在海军担任过核反应堆的操作员，对任何微小的泄漏都会严肃对待。

在这之前我从来没有担心过 Google。一次也没有。我见过我们盲目地跳进一些莫名其妙的大坑，比如柯克兰那个现在已经取消的 Google Next (GN)。我婉拒了这个项目，因为在看过原型以后，我发现看不懂。它无法让我产生共鸣。我不知

道为什么要做这个东西。但是我知道的是（这种信念从未动摇过）最终理智会占据上风，因为 Google 只会支持那些真的让人有感觉的项目。

Wave 也是一样。我理解不了这个东西，所以也从来没用过它。有一次在一个集会上，演讲人在做完产品预览后响起了巨大的欢呼声，当时我坐在后排，觉得你们这些人干嘛拼命拍手。我是错过了什么吗？后来想想，这些掌声大概只是技术人给技术人的吧，但是我并不是这样的人。我在评估一个产品的时候，依然是从最原始的角度出发：从一个需要裤子和牙刷这些东西的普通人的角度，而非一个技术人的角度。所以我理解不了 Wave。

事实证明大众并不喜欢 Wave，所以我们决定终止项目。发布 Wave，让它运营一段时间并不是什么大不了的事情。真正了不起的是我们能及时关掉它，没有假装它其实还不错，只是有点小问题，然后浪费好多年的时间去修正。微软就做不到这样，因为这家公司似乎完全失去了理智，假如他们曾经有过的话。

可是我们的实名制政策，以及它背后所考虑的那些东西，让我开始对我们的未来产生了疑虑。尽管有 GN、Wave 等失败的产品，我却从来没有担心过 Google 本身。但是这次不同。这是我第一次看到了“那件事”真的会发生的可能：就是那件将 DEC、IBM、惠普、Geoworks、微软等无数公司送入坟墓，一去不回的事情。我不知道要给它起个什么名字，但是可以保证它的真实性绝不亚于它的致命性。这个怪物以公司为食粮，看不见摸不着。

我之所以会担心是因为这个实名制政策和日常生活并不一样。我这里说的不是餐馆订餐。我说的是那个从 Usenet、MUD 和 BBS 系统，经过 40 年演变而来的网上世界，今天的它是由非常精美、延展性极佳的 Usenet、MUD 和 BBS 系统组成的。

它今天有这个样子可以说是非常惊人的。每隔两代人就会对网络有非常不同的看法。但是说到底，它其实还是那些东西，换汤不换药，只是披了一件高科技的外衣而已：讨论组，聊天室，实时通讯（还记得 Unix 的“talk”命令吗），离线通讯，媒体和内容分享，多人游戏，文档创建，求售邮件列表，新闻 feed，知识数据库，网上购物，垃圾邮件，色情网站。

差不多就是这样了。人性如此啊。

即便是再简单的调查都显示，在现实世界里，匿名为王。看看维基百科和 IMDb 这样的知识分享网站上的账户吧。或是 Slashdot、Reddit、Digg、Y-Combinator 这样的新闻聚集社区。或是各种网络游戏——全部哦。或是 PSN、XBox Live、PopCap 等网络游戏社区。看看 eHarmony 和 OkCupid 这样的交友网站。还有 eBay 和 craigslist 等私人交易网站等，更不用说各种爱好者交易和分享的网站了。再看看网络聊天室、IRC 客户端、讨论版、论坛、博客。

一圈转下来你就会发现，没人在网上用真名。事实上很多社区根本就不建议用真名。谁想和一个名叫道格·史密斯的精灵法师一起去冒险啊。

放眼世界，你看到的全是虚拟形象。这才是人们在网上交流的方式。你大可以质疑这种现象的正当性，或是其背后的原因，但是你不能无视现象本身。所有人都是通过虚拟形象来交流的——除非主动亮身份，否则没人知道你是谁。

Facebook 是一个著名的反例，因为他的创始人（我懒得去记他的名字，好像是叫杰瑞·布洛克海默）说过这样一句著名的话“匿名是懦夫的行为”，他的意思其实就是匿名的话，他不好赚你钱。Facebook 也的确在一定程度上很成功。

不过对一个不用 Facebook，不喜欢也不理解它的普通人来说，我相信他们的成功并非源自其政策，而是尽管有这种政策，他们还是成功了。作为一个每天都泡在社交网络里的普通人，不管是网络游戏、知识数据库、讨论版、聊天室、新闻聚集网站，还是爱好者网站，我觉得由于 Facebook 的实名政策，让它失去了一个巨大的全球市场。

所以当我听说 Google+这么个玩意儿的时候还是非常兴奋的，因为我知道我们是唯一能将真正安全的匿名带给这个已经习惯匿名 40 年的世界的公司，而且我们能做到和实名制一样赚钱，一样温和。

当然，其中的秘密其实一点也不神秘：只要将 Google 账户和信用卡连起来，你就可以拥有任意数量的虚拟形象了，只有 Google 才知道谁是谁。这对匿名网络来说简直是天上掉馅饼一样完美的方案。让 Google 这样的机构来管理真正的身份，

不用将他们泄露给参与者的好处在于，可以在幕后进行各种防范和监管工作（包括人工干预和算法干预），同时人们又可以自由地扮演自己想要扮演的角色。

我当初的确是非常兴奋。我甚至还一大堆腹稿，打算好好写一写这个全新的社交网站。我曾经觉得（现在也还这么觉得）我们有着近乎无与伦比能力来彻底革新，甚至可以将各种网络属性上发展起来的社交网络商品化，而且不用沦为 Facebook。

所以最后发现原来我们的实名政策及其背后的理念是这么回事的时候，我是非常沮丧的——老实说我没有怎么关心相关的讨论，所有的信息都是二手听来的。

我感到失望是因为，我们的政策和网络上的现实以及正常人的直觉完全脱节了。更糟糕的是，我们还拒绝去反省这个政策是对还是错。我们没有说：“我们会进行评估，然后决定这样做对不对。”而是机械地复述餐馆用餐是穿什么衣服这种莫名其妙的话，好像一个不相干的线下比喻就能胜过 40 年的网络经验一样，同时只顾埋头前进，不管不顾地实现选定的政策，这实在是太鲁莽了，完全不像 Google 的作风。

最不可思议的是，我们似乎刻意阻止别人在我们的 API 上搭建任何应用（至少这是我看过 API 之后的感觉），人们会在 eBay、reddit、eHarmony、魔兽世界、IMDb 这样的网站上通过自己的虚拟形象和其他人一起互动，而我们原本也是可以营造出相似的体验的。

另一个让我担心的是我们似乎觉得这只是关于昵称的争论而已。昵称的确很重要，但是光有它们是远远不够的。我们反对昵称，然后又极不情愿地考虑把它加进来，这就让人觉得有点讨厌了。不过至少这算是对市场压力的一点回应吧。真正让人反感的是我们似乎觉得昵称可以“解决”世界人民声讨的问题。昵称和虚拟形象不是一回事。只有那些失去网络社交常识的公司才不会这么想。

平台和应用对做二次开发的人来说具备相同的重要性。可惜 Google+ 平台完全不能引起我的兴趣。我曾经有过希望，但是它不但无法实现我自己的想法，连我朋友和家人的想法也都无法实现。对于任何想要搭建一个优秀虚拟互交系统，或是想要扩展现有系统的人来说，Google+ 平台完全是莫名其妙的。

我不想争论 Facebook 的问题：显然他们找到了自己的市场，我和我老婆通常直白地称之为“跟踪狂和炫耀的地方”。Facebook 在那方面确实做得很好。这里要点个赞。至少他们找到了一个可以依托的次文化，这总比微软强吧。

同样，一般情况下我也不会去争论 Google+的 API，因为我压根就对它没兴趣。

但这是我的经验里 Google 第一次这样对市场视而不见，这就很严重了。我们没有说“我们暂时不想碰那块市场”，或者说了，是我自己没听到。相反，我们说的是：“网络互动就是这样的。”对普通人来说，这就像是在说：“这是你希望人们在网上互动的方式。”

这实在是太奇怪了。就我所知，我们过去从来没有试过告诉别人他们想要什么。我们不是营销公司。我们一直都是发现人们的需求，然后把它做得简单好用。

但是整个 Google+系统（从政策、API，到公开场合的言论）都是在说：“你不要这个。”或许它说的是：“如果你要这样，那我们就需要你了。”反正从整体来看，这两句的意思是一样的。

这在我看来，我们就是在没有真正理解一个领域的情况下，盲目地发起追赶。这种现象我只在那些饱受存在危机的公司里看到过。

我从未对公司的远景和领导层失去过信心。或许现在说动摇还早了点。我只是有时候会想（之前从来没有过这种念头），那件事情是不是又来了。

我真心希望不是。

作者手记：吐槽 Google 平台

这篇文章很重要。应该是我写过的东西里最有名的一篇。至于它会不会是我这辈子最著名的文章嘛，那就不好说了，至少我暂时还不会挂，而这种吐槽企业文化的文章我还可以写出好多来，保证篇篇精彩，让你大开眼界。

说白了，首先我得豁出去。或许有一天我真的会！

其实写这篇文章本身的原因就已经很好玩了。有一次我去 Google 总部出差一周。我太太第二天跑来看我。我花了一整天时间和同事争论平台的价值。我们听说高层有意要做基础设施服务了，我等这一天已经很久了，所以我非常兴奋。结果，我们还是遇到了各种阻力。

我不是在责备同事。在 Google 做产品还是很安全的。从来没人会因为做了某个产品而被炒鱿鱼。只不过有时候你很难量化平台所带来的好处，相对的成本却很容易看到：基本上打造平台要难 3~5 倍。

只是我们差一点就做成了。自从离开亚马逊以后，我就饱受这种文化差异的困扰好多年。我非常喜欢平台。这是我想做的事情。我从 20 年前大学毕业起就一直在做这个。对我来说，平台的好处是显而易见的，我经历过太多成功的例子了。而我的老板们突然想要搭建一些服务，让其他组同时可以开发他们自己的工具，而不是让他们等我们去帮他们开发工具。（我的组就是干这个的——帮助工程师开发工具。）

我们真的只差一点点，这篇文章从内部分析了失败的原因。我实在是不吐不快。所以回到宾馆后，我抓起一瓶红酒，一口气写下我 11 篇系列连载的第 2 部分。虽然我热爱 Google，但是这件事 Google 真的做得不怎么样。我不但要挑战工程师基础设施团队的同事，更打算要挑战所有 Google 员工。这家拥有 2 万名员工的公司里的每一个人。就算我只是个无名小卒。

所以我的态度必须要谦卑，语言要准确、完整，措辞要谨慎，同时又要言之有物，让大家看到我想说的东西。

结果我只坚持了一半，后面就变成说故事了。虽然喝多了一点，但是我仍然非常尊敬我的同事们（每天上班都觉得自己有种中奖了一样的心情），所以我的文章是很真诚的，毫无保留地关心。我想很多公司外的读者也这么觉得吧。

虽然醉酒对写文章影响不大，但是它真的降低了我使用 UI 的能力，我连按钮都找不到了。我只记得自己用 Emacs 写了两个小时，然后又花了 90 分钟修改（读

到我自己都咯咯笑了好久），然后又花了整整 30 分钟研究怎么在 Google+ 上发表。它和 Blogger 真的很不一样，我原本以为肯定不是那个写着“分享新信息”的小输入框。

Google 员工有一个内部版本的 Google+，看起来和外部版本一模一样。分辨它们的唯一方法就是看右上角的 E-mail 地址，或是头像（假如不一样的话）。我记得自己检查了好几遍确保自己不会把文章贴到外网上去，不过更多的精力都用来琢磨怎么发表文章了。

等我终于搞清楚怎么发文章后，我记得自己按下了“分享”按钮（不记得具体叫什么了），告诉自己“这样你们就知道我怎么想的了”，然后在 20 秒内睡着——当时应该已经是半夜了。

差不多是凌晨一点半的时候，我被电话吵醒。我住的酒店是随便挑的，我也没有告诉别人酒店的名字——只有在报销单据的时候你才会告诉别人你住在什么地方，有时候出差回来已经好几周了。我已经和老婆说过晚安了，她在柯克兰，应该还没睡吧。谁会在凌晨一点半给我打电话？

打电话的是我们苏黎世办公室的一位同事，他是个不错的家伙，他代表公共关系部门（我想应该是在伦敦的那个）问我知不知道我的文章发到了外网上。我的第一反应是：“什么？！不会吧！绝对不可能！”虽然我喝高了，而且累得要死，外加凌晨一点半，我心底还是有一个微弱的声音告诉我，要不是我的文章捅了篓子，这个时间不太可能有个身在苏黎世的陌生人能找到我住的酒店，还给我打电话的。

他非常客气，PR 部门也很客气，接下来发生的事情我在后续的文章和评论里已经解释得很清楚了。等事情过去后，我们每次说起来都会哈哈大笑，所有的烦恼只是让我们多添了几根白头发罢了。

直到现在我也不知道他们是怎么知道我住哪家酒店的。我也不想知道。

这篇文章发表已经有 11 个月了，与此同时，我们也在某个小角落里慢慢搭建平台。这种感觉非常好。此外，我的生活并没有什么太大的变化，除了我收到了很多很多工作邀约。包括亚马逊，滑稽吧。我问他们：“你确定公司高层知道你在邀

请我吗？”他们答道：“当然，我们很清楚你的，嗯，来吧，别担心。”每次我都努力地克制住自己不去问：“那么，嗯……最高你问到了哪个级别？”因为这实在是，怎么说呢……太诡异了吧，我觉得。总觉得像是陷阱。

不过 Google 依然是这个世界上最棒的公司，所以我想我还会继续待着吧。

吐槽 Google 平台

我在亚马逊待了差不多 6 年半，在 Google 也差不多有这么长时间了。要说这两家公司最大的不同（这种感觉每天都在加深），那就是亚马逊所做的一切都是错的，Google 所做的一切都是对的。好吧，这样说未免有点太以偏概全，但却出乎意料的准确。很疯狂吧。比较两家公司的方法有几百种，Google 几乎在每个点都毫无疑问地胜出，除了 3 个方面，要是我没记错的话。我曾经还做过一张表格，但是公司的法律部门不让我公开，不过招聘部门倒是很爱这份表格。

我就大致说一下好了：亚马逊的招聘流程是让每个团队自己招人，这种方式有本质上的缺陷，尽管做了很多努力，但是每个团队的标准依然大相径庭。他们的运维一团糟，他们没有 SRE (Site Reliability Engineer, 网站可靠性工程师) 这种岗位，工程师必须自己包揽一切，进而导致他们没时间写代码——当然这也分什么组，看运气了。他们对慈善漠不关心，也不会对社区的需求伸出援手。除了不痛不痒的调笑，他们从来没想到过这个问题。办公区域是布满灰尘的小隔间，他们不愿意在装修办公室和会议区上花一毛钱。他们的薪金和福利制度很差，不过近来由于 Google 和 Facebook 的竞争，情况已经好很多了。但是他们没有任何令人骄傲的东西——他们只会去匹配聘书上的数字，仅此而已。他们的代码库可以说是灾难现场，除了团队自己遵循的标准以外，毫无任何工程标准可言。（作者注：听说这方面现在总算有所改进了。哈哈。）

平心而论，他们那个按照版本组织类库的系统的确很棒，我们应该模仿一下的，另外，他们那个发布—订阅的系统也很出色，我们也没有对应的东西。但是大多数

时候，他们用的那堆工具只是在关系数据库上读写状态机信息的垃圾而已。就算免费我们也看不上。

我觉得那个发布订阅系统和类库系统是亚马逊胜过 Google 的两个方面。（作者注：现在我觉得 Google 的发布订阅系统更出色。不过亚马逊在组件方面依然做得比较好。）

我估计很多人会争辩说，他们的发布和迭代的速度也是一绝，但我觉得这种事情得两说。他们把尽早发布看得比一切都重要，包括保留员工和工程准则等一系列长远来讲很重要的东西。所以尽管这让他们在市场上占得了一点先机，但同时也制造了很多其他麻烦。

但是有一件事，他们做得非常非常出色，甚至能弥补他们在政治、理念和技术方面留下的烂摊子。

杰夫·贝索斯出了名地喜欢越级管理。亚马逊网站上的每个细节他都会插手。他曾经雇用了拉里·特斯拉（苹果的首席科学家，可能也是全世界最著名、最受尊敬的人机交互专家），结果 3 年来没有听过一次他的建议，最后拉里只得（明智地）选择辞职。拉里原本可以通过各种可用性研究来无可辩驳地证明没人会用那个该死的网站，可是贝索斯就是不愿放手，首页上这些数以百万计的像素就好像他的宝贝孩子一样。所以它们还在网页上，而拉里只能离开。

顺便说一句，微管理可不是亚马逊胜过我们的第 3 个方面。当然我不是说他们管理得不好，但是我不会把这一条列为优点。我说那么多只是想介绍一点背景知识，方便你了解到底是怎么回事。我们说的那个家伙在很多公开场合都严肃地宣称，他在亚马逊工作应该要领薪水的。当别人不同意他的话时，他会递出黄色的小纸条，提醒别人“别忘了是谁在运营这家公司”。我猜这个人就是个平庸的……嗯，史蒂夫·乔布斯。只不过他没有时尚和设计的品味。贝索斯是个非常聪明的人，这一点我承认。他只是把再普通不过的控制狂演得好像喝高了的嬉皮士一样。

有一天杰夫·贝索斯发布了一条指令。当然，他经常发命令，然后人们就会像被橡皮锤子乱砸一通的蚂蚁一样乱成一团。不过这次（差不多是 2002 年前后吧，

误差不超过一年），他的指令令实在是太超前，太瞩目，太震撼，让其他指令显得完全微不足道。

他的这项指令大致含义是这样的：

1. 从今天起，所有的团队都要以服务接口的方式提供数据和各种功能。
2. 团队之间必须通过接口来通信。
3. 不允许任何其他形式的互操作：不允许直接链接，不允许直接读其他团队的数据，不允许共享内存，不允许任何形式的后门。唯一许可的通信方式就是通过网络调用服务。
4. 至于具体的技术则不做规定。HTTP、Corba、Pubsub、自定义协议都可以。贝索斯不关心这个。
5. 所有的服务接口，必须从一开始就要以可以公开为设计导向，没有例外。这就是说，团队必须在设计的时候就计划好，接口要可以对外面的开发人员开放。没有讨价还价的余地。
6. 不听话的人会被炒鱿鱼。
7. 谢谢！祝你今天过得开心！

哈哈！你们这些 150 多名前亚马逊员工肯定能立刻看出第 7 条是我开的玩笑，贝索斯才懒得关心你今天过得怎么样呢。

不过第 6 条却是如假包换的，所以大家都忙不迭地开始干活。贝索斯指派了两个大监工以确保进度顺利，他们由超级大监工李克·戴尔泽尔负责。李克是退役军人，西点军校毕业，打过拳击，前沃尔玛首席刑侦专家兼 CIO，是一个很壮的家伙，既和蔼可亲，又有点吓人，老喜欢把“强化接口”挂在嘴上。李克自己就是一个能说会走的强化接口，所以毫无疑问，大家都做了很多工作，而且确保李克看到他们的成果。

于是在接下来的两年里，亚马逊内部转向了面向服务的架构。在这过程中他们学到了大量的宝贵经验。市面上关于 SOA 的文献资料有很多，但是对亚马逊这种

规模的公司来说，这些资料根本就是纸上谈兵。亚马逊的技术人员在这个过程中总结了很多经验教训。这里仅列出一小部分：

- 出问题的时候定位更麻烦，因为一张传票可能要经过 20 次服务器调用才能找到问题的真正所在。要是每个小组需要 15 分钟来定位，那么等到正确的小组接手的时候，时间已经过去几个小时了，除非你为之搭建了很多外围监控和报警措施。
- 其他小组突然就变成了潜在的 DOS 攻击者。除非每个服务上都设有严格的用量和限量措施，否则谁也动不了。
- 监控和 QA 是一回事。这只有在你真正开始做 SOA 的时候才能体会到。当你的服务说“是的，一切正常”的时候，很有可能整个服务唯一还正常工作的部分就是那个回应“一切正常，收到，完毕”的小模块。只有真的去调用服务，你才能确定它是正常的。所以直到监控措施可以对所有的服务和数据进行完整的语意检查，否则问题依然存在，而如果能做到这一点，其实本质上就是自动化 QA 了。所以他们是统一的。
- 在面对成百上千的服务时，如果你的代码必须和其他组的代码通过这些服务来通信，那么没有服务发现机制是不可想象的。这又离不开服务注册机制，而它本身也算是一个服务。所以亚马逊有一套统一的服务注册机制，你可以通过反射（编程的方式）找到所有服务，它有那些 API，目前是不是运行正常，在什么位置等。
- 在设计他人代码的情况下查找问题的难度要高很多，除非有统一的方式在沙盒里运行所有服务，否则几乎不可能进行任何调试。

这只是很小的几个例子。亚马逊还掌握了几十乃至几百条这样的经验。其中有不少很滑稽的故事是关于公开服务的，不过也没有你想象得那么多啦。将团队组织成服务让大家明白，基本上他们不能信任其他团队，正如不能信任第三方工程师一样。

在 2005 年中，我离职去 Google 的时候，这一转变仍在进行中，不过已经取得

很大的成果了。从贝索斯发布命令到我离开的这段时间里，亚马逊已经从理念上彻底转型为一家一切以服务为优先考量的公司了。基本上现在所有的设计都是以此为基础，包括那些可能永远也不会公开的内部设计。

现在他们不再是因为害怕被炒鱿鱼才这么做。我是说，炒鱿鱼还是挺吓人的，只不过那已经是每日生活的一部分了，就像是给恐怖海盗贝索斯打工一样。他们做服务是因为他们明白那是正确的。SOA 毫无疑问有优点也有缺点，有些缺点甚至还挺严重。但是总体来说思路是正确的，因为只有 SOA 驱动的设计才能产生平台。

这才是贝索斯的命令背后真正的意图。他才（包括现在）不关心团队好不好，用什么技术，除非手底下的人把事情办砸了，否则他也不会关心任何具体细节。但是贝索斯高瞻远瞩地认识到亚马逊需要成为平台。

你不会真的觉得一家网上书店真的有必要变成可扩展、可编程的平台吧？

好吧，贝索斯首先意识到他们搭建的卖书送书的基础设施其实可以变成一个非常出色、可定制的计算平台，于是就有了 Amazon Elastic Compute Cloud, Amazon Elastic MapReduce，以及 Amazon Relational Database Service 等一系列服务，你可以在 aws.amazon.com 上自由浏览。很多成功的公司的后台都搭建在这些服务上，其中 reddit 是我的最爱。

另一点就是他自己不可能永远不犯错。我觉得当拉里·特斯拉说他的妈妈根本不知道要怎么用这个网站的时候还是有触动到贝索斯的。虽然我们不清楚他说这话的时候指的是谁的妈妈，但是这又有什么关系呢，反正不管是谁的妈妈，都不知道要怎么用这个网站。连我自己都觉得他的网站太难用了，要知道我在那儿可是工作了好多年啊。我也是到现在才学会模糊自己的视线，把注意力放到页面中心那几百万像素上去的。

我不太清楚贝索斯是怎么意识到这一点的——打造一个所有人都满意的产品是不可能的。不过无所谓了，反正他明白就好。这种现象其实是有正式名称的。那就是所谓的可用性，在 IT 界没有比这更重要的了。

再说一遍，最重要的事。

要是你听到这个词首先想到的是：“什么？你是说盲人和聋哑人的那种可用性？”那么就对了，像你这么想的人很多很多：对可用性缺乏正确的理解，你只是还没明白而已。这不是你的错，也不是那些眼盲目聋，或是任何行为不便的人的错。如果软件（或者任何想法）对某个人群不怎么好用的话，不管是什么原因，那都是软件或是那个想法所要传达的信息的错。即不可用。

和生活中的其他重大事物一样，可用性有一个邪恶的孪生兄弟，由于幼年时父母的偏爱，逐渐变成了死对头（对了，可用性的对头可不止一个），那就是安全性。他们简直可以说是势不两立。

不过我觉得可用性实际上还是要比安全性来得更重要一些，因为可用性为 0 等于没有产品，而安全性为 0 并不等于不能有一个相对成功的产品，比如说 Playstation Network。

这个话题我可以写一本书出来，厚厚一大本，里面全是我工作过的那家蚂蚁和橡皮锤子的公司里传出来的段子。不过我不会发表它的，你也不会读到它，除非我不想混了。

Google 做得不好的最后一个方面就是平台。我们完全不理解平台。我们不明白平台“意味着”什么。有些人懂，但是他们人微言轻。过去 6 年来我已经清楚地认识到了这一点。我曾经以为来自微软、亚马逊，甚至新晋的 Facebook 的同行压力会让我们觉醒过来，着手去做通用服务。不是那种不经考虑的半成品，而是像亚马逊做的那样：实实在在，真刀真枪，从当下开始把它当作头等大事来做。

但是没有，一点迹象也没有。它在日程表上最多也就是第 10 名，11 名的样子。甚至可能只有 15 名，天知道呢。反正是无足轻重。虽然有几个团队对此非常认真，但是大多数团队要么完全没想到，要么只有很小一部分人在很肤浅的层面上思考这个问题。

大多数团队甚至要求他们提供一个简单的服务，以可编程的方式去访问他们的数据和计算，都很困难。大多数人都觉得自己做的是产品。而简单的服务其实也是很可怜的服务。回头看看上面那份亚马逊经验教训的列表，你告诉我哪一条是简单

的。至少在我看来是没有。简单是很好，但是那就像我明明要辆车，你却丢给我一个零件一样。

没有平台的产品是没用的，说得更精准一点，缺少平台支持的产品肯定会被有平台支持的同等产品取代。

Google+就是一个从公司最高层（嗨，拉里、谢尔盖、埃里克、维克，你们好）一直到最基层的员工（说的就是你）都完全没有理解平台的典型例子。没人看懂。平台的金科玉律就是要吃自己的狗粮。Google+平台却很遗憾，是个马后炮。我们在发布的时候根本没有任何 API，上次我看的时候，我们才有一个可怜巴巴的 API。他们发布的时候有个组员跑来通知我，我问道：“所以这个是偷窥 API 了？”她一下被我问蒙了，只得说：“算是吧。”天哪，我只是在说笑而已，但是没有……我们提供的唯一一个 API 就是获取其他人的信息流。所以我觉得这个笑话应该算是我发明的。

微软了解狗粮定律至少有 20 年了。这成为他们的企业文化已经整整一代人了。你不能自己吃人吃的东西，却让开发者吃狗吃的东西。这样只是拿长期的平台价值来换取短期的成功而已。平台是需要长远眼光的。

Google+就是这样一种膝跳反射，短期思考的产物，错误地以为 Facebook 之所以成功是因为人家做了一个好产品。但那并不是他们成功的原因。Facebook 之所以那么成功是因为他们通过让其他人参与进来的方式，搭建起一整套各异的产品。所以 Facebook 才会与众不同。有的人会整天泡在黑帮战争里，而有的人则把时间都花在开心农场上。Facebook 上有成百上千种消磨时间的好去处，所以才能满足所有人。

而我们的 Google+团队看到这个后说：“天哪，看来我们也需要游戏。我们去找人来帮我们写游戏吧。”现在你能看出这错得有多离谱了吗？问题就在于我们在试图预测人们想要什么，然后再实现它。

可是这样是行不通的，很不靠谱。纵观 IT 界，这个世界上有能力做到这一点的人屈指可数。史蒂夫·乔布斯就是其中之一。可我们没有史蒂夫·乔布斯。很遗憾，但这是事实。

拉里·特斯拉或许让贝索斯意识到自己不是史蒂夫·乔布斯，但是贝索斯也意识到自己用不着成为史蒂夫·乔布斯也一样可以做出好产品：大家都喜欢好用的接口和工作流。他只需要让第三方开发者参与进来，一切就顺理成章了。

要是你觉得我说的东西都是秃子头上的虱子，那我要向你道歉，因为你的感觉是对的。这真的是明摆着的道理。可是我们并没有那么做呀。我们不懂平台，也不懂可用性。这两者其实是一回事，因为平台可以解决可用性的问题。平台就是可用性。

而且，微软也懂这个道理。你一定和我一样吃惊吧，毕竟他们很少“懂”什么东西。但是他们理解平台，因为无巧不巧，他们一开始就是做平台的。他们在这个领域里已经耕耘 30 多年了。要是你从来没去过 msdn.com，不妨上去瞧一瞧，花点时间浏览一下，你会感到惊艳的。它真的包罗万象。他们有成千上万的 API。他们有巨大的平台。大到过分了，因为他们不懂怎么精简设计，但至少他们做的是平台。

亚马逊也懂，亚马逊的 AWS 非常了不起。去看看吧，随便浏览一下。太丢人了，我们根本没有这样的东西。

苹果显然也懂。他们的有些东西并不开放，特别是移动平台。但是他们很懂可用性，知道第三方开发的威力，他们吃自己的狗粮。你知道吗？他们的狗粮味道相当不错。他们的 API 比微软的要干净很多，从一开始就是如此。

Facebook 也懂。这才是真正让我担心的，担心到让我挪动懒惰的屁股来写这篇文章。我讨厌写博客，我讨厌……加一，管它叫什么呢，反正 Google+ 不是个吐槽的好地方，但是你还是要说，因为我们真的希望 Google 最后能成功。我真心希望如此。Facebook 邀请过我，跳过去只是分分钟的事情。但是 Google 才是我的家，所以我才坚持大家像一家人一样坐下来说说真心话，哪怕忠言逆耳。

在看过微软、亚马逊，还有 Facebook（我没有去看，因为我不想太伤心）的平台后，再去 developers.google.com 看看。区别很大，是不是？那种感觉就好像小学五年级的外甥的作业，要是一家公司只有一个五年级学生的话，会怎么搭建一个

强大的平台。

请不要误解我——我知道开发团队已经很努力才能做到今天这样的地步。他们已经很棒了，因为他们真的懂平台，在这样一个说客气了对平台漠不关心，说难听点对平台抱有敌意的环境里，像英雄一样挣扎着想要做出一个平台来。

我只是诚实地以一个外人的角度来评价 developers.google.com。它真的太小儿科了。天哪，地图 API 在哪里？有些东西还只是实验室项目。可以点的那些 API 都是……无足轻重的东西。它们的确是狗粮。可惜品质实在太差。和我们的内部 API 比起来，这些玩意儿只是边角料。

也请不要误会我在批评 Google+。他们不是唯一应该承受批评的人。这是文化的问题。我们在内部经历的就是战争，一场几乎必输的战斗，一边是支持平台，人微言轻的少数派，另一边是强大，有钱有自信的制作人。

任何成功在内部植入从一开始就应该开发外部编程平台概念的团队都是由小角色发起的——比如 Google Maps 和 Google Docs，我知道 GMail 也在朝那个方向发展。但是搞到资金可不容易，因为这不符合我们的文化。而且和微软 Office 庞大的编程平台比起来，我们那点点资金根本微不足道：就好像毛绒绒的小兔子和霸王龙那样的差距。Docs 团队知道除非能实现 Office 那种程度的脚本，否则他们永远也无法和它竞争，可是他们得不到任何资源上的支持。这是我猜的，因为现在的 Apps Script 只支持 Spreadsheet，而且它的 API 连键盘快捷键都不支持。在我看来，这支团队就是姥姥不疼舅舅不爱啊。

讽刺的是，Wave 倒是一个很棒的平台，愿它安息吧。不过创造一个平台并不能让你立刻成功。平台需要杀手级应用。Facebook（就是那套朋友墙的东西）就是 Facebook 平台上的杀手级应用。但要是以此断论 Facebook App 离开了 Facebook 平台也一样能在任何地方成功就很愚蠢了。

你知道人们为什么老是说 Google 很傲慢吗？我自己就是 Google 的员工，所以听到别人这么说的时候就特别反感。不管从哪个方面来说，我们都不傲慢。我们基本上可以说是 99% 免疫傲慢的。我在文章开头确实有说过（要是你还记得的话）

Google “做的每件事都是对的”。我们的本意是好的，大多数被指责傲慢的时候，是因为我们没有雇用他们，他们不喜欢我们的政策等。这样的指责能让他们觉得心里舒服一点。

但当我们当仁不让地说我们知道怎么为所有的用户设计最完美的产品的时候，相信我，这种话我听过很多次了，我们的确是傻瓜。你可以说这是傲慢，或者不知天高地厚，或者随便什么——什么都行，因为这的确很蠢。因为根本就不存在适合所有人的完美产品。

所以我们搞出一个不允许设置默认字体大小的浏览器。这算是对可用性的侮辱了吧。我的意思是，等我老了以后肯定会眼神不好，真的。我这辈子基本上都是近视，等过了 40 岁以后估计再近也看不清楚东西了。所以选择字体变成了一件生死攸关的大事：它基本上能把你彻底排除在一个产品之外。但是 Chrome 团队在这一点上却傲慢得惊人：他们想要做出一款零配置的产品，而且非常固执，要是你瞎了或者聋了或者有其他什么不方便，那你就死去吧。接下来你生命里访问的每一个网页都要不停地按 Ctrl+ 或 Ctrl-。

不光是他们哦，所有人都是如此。问题在于我们是一家彻头彻尾的产品公司。我们做出了一个需求广泛的成功产品——搜索引擎，它的巨大成功让我们有点飘飘然了。

亚马逊也曾经是一家产品公司，所以要依靠不同寻常的外力才能让贝索斯了解平台的重要性。这种力量就是他们不断消失的利润空间，他很担忧，所以必须找到其他出口。可他手上只有一堆工程师和一堆计算机……怎么才能变现呢……你可以看到他是怎么想到 AWS 的了吧，当然我这是事后诸葛亮了。

微软从一开始就是一家平台公司，所以他们自然经验丰富。

不过 Facebook 却让我很担心。我不是专家，但我知道他们也是以产品起家，也是靠产品大获成功的。所以我不知道他们到底是怎么转型成一家平台公司的。这已经有一段时间了，因为他们必须要先转变成平台，然后黑帮战争这样的东西才有可能出现。

或许他们只是看着我们问自己：“我们才能击败 Google？他们缺了什么？”

我们面临的问题很严重，因为这要求我们从文化上做出巨大的转变后才有可能迎头赶上。我们在内部也不做面向服务的平台，更遑论外部的了。这表示“不懂平台”是整个公司的问题：产品经理不懂，工程师也不懂，产品团队不懂，没人懂。就算某些人懂，比如说你好了，那也无法撼动大局，除非我们能把这个当成生死存亡的大事来对待。我们不能不断地发布产品，然后假装将来有一天会把它们神奇地变成可扩展的优美平台。我们已经尝试过了，是行不通的。

平台的金科玉律“吃自己的狗粮”换句话说就是“建一个平台，然后所有的东西都搭在它上面”。事后再迁移是不现实的。至少不会如你所愿——问问那些平台化微软 Office 的人，或者问问那些平台化亚马逊的人你就知道了。拖拉一下，难度就可能比一开始就做要高 10 倍。讨不了巧。你不能给内部应用留下后门以获取特殊权限，不管是什么理由。必须从一开始就从困难的问题着手。

我并不是说现在已经太晚了，只不过越等下去，我们就离“太晚”越近。

我真的不希望事情会到那一步。我要说的已经基本上说完了。这篇文章我酝酿了 6 年。要是有什么冒犯地方，或者我歪曲了某个产品、团队、个人，或者其实我们又在做很多平台的东西的话，我感到很抱歉，因为我和我聊过的人从来都没听说过。非常抱歉。

但是我们现在必须开始做正确的事了。

总结

没想到你真的读完了整本书，我实在是有点受宠若惊。要是你有兴趣，我还有更多的东西可以分享。我们只包含了所有素材的 10%~15%，保证书的主旨清晰、范围可控。你可以在网上搜到我其他的牢骚，2005 年 6 月之前的博客是“[Stevey's Blog Rants](#)”，2005 年 6 月以后的博客是“[Stevey's Drunken Blog Rants](#)”。

要是你只是跳到最后看看我有没有写 TL;DR 的话，好吧，满足你了。

太长；懒得读 (Too Long; Didn't Read)

- 软件开发的方式多种多样，不存在谁好谁坏。但是它们互相都看不起。
- 好的程序员之所以出色是因为熟能生巧。
- 只要你愿意，随时都可以学习新语言。
- 如果你想要当经理，那你很可能不会是个好经理。
- Lisp 很难掌握，但它是唯一能让我继续快乐的语言。
- Emacs 很难掌握，但却是受益终身的投资。
- 离开舒适区。时不时地学点新东西。
- 为自己写点东西。只有这样你才知道那是不对的。
- 多笑一点。这很健康，也让人感觉良好。
- 记得自嘲，不过不要在公开场合大声自嘲，也不要忘了解释。

请告诉我你的想法。我的 E-mail 是 steve.yegge@gmail.com。有时候我会很忙，也不一定每封 E-mail 都回。但是我一定会读的，大部分都会回，而且每一封都能让我学到点什么。