

计算机科学基础班（第5期，现实接轨班）招生

计算机科学基础班，是一门零基础，目标在于短期内掌握计算机科学精髓的课程。它是我 20 多年的计算机学术和工程实践，加上两年面向社会的教学实验的结晶。课程吸取了世界上主要的计算机入门教学方式的优点，避免了它们阻碍初学者理解的各种问题，以至于完全零基础的学生也可以在短短两个月之内，掌握大学博士阶段才可能学到的精华内容。这些内容足以建立起坚实的知识基础，使得他们对于理解计算机科学的其他方面从容自如。

[第 4 期基础班](#)于 2021 年 5 月进行之后，就再也没有开过基础班。现在经过了[进阶班](#)和[continuation](#)和[并发计算专项班](#)的巨大成功，我决定再次开展从零开始的基础班。欢迎有志人士报名。

这次课程相比之前的几届，我会增加一些“现实接轨”内容。通过几种当今最流行的语言的特性，掌握现实工程实践中能用到的概念。这些“接轨语言”初步计划是包括 Python，Java 和 Rust。通过理解它们，掌握“面向对象”，“静态类型系统”和“内存管理”等重要概念。

所以这一期的基础班，不但会打下“递归”这样的坚实基础，理解“解释器”这样的深入内容，而且会更容易将这些精髓概念与现实工程接轨。

课程方式

- 课程使用 Zoom 视频会议的方式进行教学。
- 每周一次课，大概 2 小时。
- 每周有一次自由讨论会，为同学们统一答疑。
- 每个学生会被加入自己独有的辅导群，会由经验丰富的助教进行辅导，并且由我监督和指点，保证学生不卡在不必要的环节。
- 学生会被加入第 5 期基础班的大群，方便同学之间交流。

适用人群

- 从中学到博士阶段的各专业学生，不管文科还是理科生都能学会。之前的课程已经成功让一个[13 岁少年](#)深入地掌握了这些内容，并且超过了很多经验丰富的成年学生。
- 已经从事工作的人员，包括 IT 从业人员，IT 管理类工作和其它工程类工作人员。
- 其它各类对计算机感兴趣，想把它作为业余爱好的人员。

学费和报名方式

- 学费统一为 12800/人。
- 报名请发送 email 到 yinwang.advising@icloud.com。标题为《计算机科学基础班第 5 期报名》。来信请说明自己的基本信息，附件发送一份简历，提供微信联系方式。请写一段 200 字左右的“个人说明”，说明你的学习动机。对于符合要求的求学者，我会进行简短的微信语音面试。

开始时间

课程计划于 2024 年 1 月开始，具体的时间会根据报名情况而定。课程中间如果遇到春节等节假日，会短暂休息。

教学语言

课程开头会使用 JavaScript 作为教学语言，因为 JavaScript 易用而且应用非常普遍。但课程并不是教 JavaScript 语言本身，课程教的思想不依赖于 JavaScript 的任何特性，它可以应用于任何语言。

为了使学生能够和现实的编程工作接轨，这一期的课程会在最后加入其它三种“现实语言”。目前初步的计划是 Python，Java 和 Rust。我会讲授每种语言特有的功能，比如“面向对象”，“静态类型系统”，“内存管理”等等。当然，这些特性都会建立在之前学会的基础上。

对于每种语言我会增加一节课，一次讨论会，和一周的辅导时间。根据进阶班的成功经验，我觉得这些语言每种增加一节课，应该足够。

课程大纲：

第 1 课：函数。跟一般课程不同，课程不从所谓“Hello World”程序开始，也不会叫学生做一些好像有趣而其实无聊的小游戏。一开头我就讲最核心的内容：函数。关于函数只有很少几个知识点，但它们却是一切的核心。只知道很少的知识点的时候，对它们进行反复的练习，让头脑能够自如地对它们进行思考和变换，这是教学的要点。我为每个知识点设计了恰当的练习。

第 1 课的练习每个都很小，只需要一两行代码，却蕴含了深刻的原理。练习逐渐加大难度，直至超过博士课程的水

平。我把术语都改头换面，要求学生不上网搜索相关内容，为的是他们的思维不受任何已有信息的干扰，独立做出这些练习。练习自成系统，一环扣一环。后面的练习需要从前面的练习获得的灵感，却不需要其它基础。有趣的是，经过正确的引导，好些学生把最难的练习都做出来了，完全零基础的学生也能做出绝大部分，这是我在世界名校的学生里都没有看到过的。具体的内容因为不剧透的原因，我就不继续说了。

第 2 课：递归。递归可以说是计算机科学（或数学）最重要的概念。我从最简单的递归函数开始，引导理解递归的本质，掌握对递归进行系统化思考的思路。

递归是一个很多人自以为理解了的概念，而其实很多人都被错误的教学方式误导了。很多人提到递归，只能想起“汉诺塔”或者“八皇后”问题，却不能拿来解决实际问题。很多编程书籍片面强调递归的“缺点”，教学生如何“消除递归”，却看不到问题的真正所在——某些语言（比如 C 语言）早期的函数调用实现是错误而效率低下的，以至于学生被教导要避免递归。由于对于递归从来没有掌握清晰的思路，在将来的工作中一旦遇到复杂点的递归函数就觉得深不可测。

第 3 课：链表。从零开始，学生不依赖于任何语言的特性，实现最基本的数据结构。第一个数据结构就是链表，学生会在练习中实现许多操作链表的函数。这些函数经过了精心挑选安排，很多是函数式编程语言的基本函数，但通过独立把它们写出来，学生掌握的是递归的系统化思路。这使得他们能自如地对这类数据结构进行思考，解决新的递归问题。

与一般的数据结构课程不同，这个课程实现的大部分都是「函数式数据结构」，它们具有一些特别的，有用的性质。因为它们逻辑结构清晰，比起普通数据结构书籍会更容易理解。与 Haskell 社区的教学方式不同，我不会宗教式的强调纯函数的优点，而是客观地让学生领会到其中的优点，并且发现它们的弱点。学会了这些结构，在将来也容易推广到非函数式的结构，把两种看似不同的风格有机地结合在一起。

第 4 课：树结构。从链表逐渐推广出更复杂的数据结构——树。在后来的内容中，会常常用到这种结构。树可能是计算机科学中最常用，最重要的数据结构了，所以理解树的各种操作是很重要的。我们的树也都是纯函数式的。

第 5 课：计算器。在熟悉了树的基本操作之后，实现一个比较高级的计算器，它可以计算任意嵌套的算术表达式。算术表达式是一种“语法树”，从这个练习学生会理解“表达式是一棵树”这样的原理。

第 6 课：查找结构。理解如何实现 key-value 查找结构，并且亲手实现两种重要的查找数据结构。我们的查找结构也都是函数式数据结构。这些结构会在后来的解释器里派上大的用场，对它们的理解会巩固加深。

第 7 课：解释器。利用之前打好的基础，亲手实现计算机科学中最重要，也是通常认为最难理解的概念——解释器。解释器是理解各种计算机科学概念的关键，比如编程语言，操作系统，数据库，网络协议，Web 框架。计算机最核心的部件 CPU 其实就是一个解释器，所以解释器的认识能帮助你理解「计算机体系构架」，也就是计算机的“硬件”。你会发现这种硬件其实和软件差别不是很大。你可以认为解释器就是「计算」本身，所以它非常值得研究。对解释器的深入理解，也能帮助理解很多其它学科，比如自然语言，逻辑学。

第 8 课：Python。利用 Python 语言，讲述所谓“面向对象”编程方式。之前的课其实已经用到“面向对象”的精华思想，所以现在用 Python 只是为了和“现实编程”对接一下，让同学们理解之前学到的思想能如何对应到现实的工作代码中。这节课也会覆盖其它命令式语言的基本用法，比如数组，赋值，循环等等。

因为“AI 热”，Python 可能是目前最热门的语言。我个人不在乎 AI，但 Python 作为理解“面向对象编程”的工具，我觉得是一个不错的选择。虽然有些人（比如我）可能不在乎“面向对象”，但理解别人心目中的“面向对象”概念，对于实际工作也是有帮助的，否则你会很难理解这样的代码。

第 9 课：Java。利用 Java 语言，引入“静态类型系统”这个重要工具。静态类型系统是创建大型工程，保证基本质量的重要工具。通过 Java 的类型系统，同学们可以掌握基本的静态类型检查规则，泛型等重要概念。这个基础会为其它类似的语言，比如 C++，C#，Rust 等打好基础。

Java 虽然备受很多“函数式程序员”诟病，说它过度复杂，但只要你学会了精髓的编程技巧，一样能用 Java 写出简单而优质的代码。而且 Java 至今仍然是 Web 服务器领域用得最多，相对而言最可靠的语言。

第 10 课：Rust。Rust 是当今最热门的新兴语言，但里面的功能其实大部分都是其它语言早就有的，只不过 Rust 选择了它们最好的一些部分，也当然包含一些自己的设计错误。Rust 独有的特性是“静态内存管理”，也就是不通过“垃圾回收”（GC），而是通过静态类型系统来保证内存的安全性。我们会通过这一节课，掌握内存管理的基本原理，理解 Rust 独有的 ownership，lifetime 等重要概念。如果有时间，我们也会利用 Rust 来理解其它一些重要的类型系统特性。

通过 Rust 理解了内存管理，也会帮助同学们理解 C，C++ 等完全“手动内存管理”的语言，为底层的系统编程打下基础。

continuation 和并发计算专项班

进阶班的内容在短时间的内覆盖面很广。虽然每个主题都讲的很深入浅出，整个课程也需要两个多月，所有的内容都很烧脑，所以再次开课可能不大方便。我在考虑把进阶班的内容分成小块，对每一块进行专项讲座。这样整个过程会大大缩短，时间安排会更加灵活，脑力负担会减轻。专门针对一小块内容，会大大增强认知效果。

所以我考虑进行一些非常短的“专项班”课程（也可以叫讲座），第一个主题可能是「continuation 和并发计算的实现」。它将以基础班为基础开始，自己实现最先进的 continuation 构造，包括 CPS，call/cc，shift/reset 等高级操作。这些主题将会加深对操作系统进程（线程）调度，coroutine（goroutine），node.js，async/await，promise，future 等系统和概念的内部机制的认识。

这个主题我估计总的过程只需要两周，大约三次课完成，随着内容的深入和发展，也可能临时增加课时和讨论。课后的微信群也会提供更多的思考内容，与现实的并发系统接轨。课后会形成一个专项研究群，持续讨论与这个话题相关的最新研究。我希望参加课程的同学能成为这方面的专家。

最后的时间还没有确定，会根据报名情况而定。由于需要基础班的基础知识，并且可能涉及一些具体的练习，所以目前仅限参加过基础班或阅读班的同学，参加过进阶班的同学也可以再次报名以加深理解。有兴趣的同学可以微信跟我联系。

计算机科学阅读班提供疫情特价

由于疫情封控给大家带来的经济和生活困难，我决定从今天起，直到中国完全放开，经济生活完全恢复到以前的状态为止，计算机科学阅读班的学费对国内的同学一律实行半价，也就是 ¥5120。国外的同学如果最近失去了工作，也可以申请这个疫情特价。经济特别困难的同学，经过核实之后可以提供更低的学费。希望大家平安度过难关。

自然视力恢复法

(由于主页从国内访问不方便，如需分享，可以下载[本文的 PDF 版](#)。网页版生成的 PDF 排版不大好，建议下载专门拍版的 PDF。)

防止和逆转近视是一个相当简单的事情，理解其中的原理只需要高中光学知识，具体操作方法只需要另外配一副眼镜，然而我也是最近才发现。

简短版

这篇文章因为增加了很多原理说明，变得有点长。如果你不耐烦看那些原理，这里有一个两句话的版本。想要逆转近视，你只需要做这件事：

1. 配一副比你本来的度数少 100 度的眼镜。戴上这个眼镜的视觉效果，就像是近视 100 度的人，基本不影响生活。以后就戴这副眼镜照常生活，该怎么用眼怎么用，不需要做什么特别的事情。
2. 几个月之后，会发现视力变好，度数降低了。当度数降低了 50 度左右，就去换一副新的镜片，保持比实际度数低 100 度。如此循环，直到视力恢复到 1.0。

不耐烦的人已经可以去操作了，但如果你不相信这事有这么简单，希望了解这个方法的原理，那就继续往下看吧。

眼球外部肌肉的秘密



近视的成因，得从眼球外部肌肉说起。眼球的外面有 6 块精密的肌肉（4 块直肌，2 块斜肌）。很多人以为这些外部肌肉只是用于转动眼球，而 100 年前有个叫 W. H. Bates 的眼科专家通过对动物眼睛做实验发现，这些肌肉也能改变眼球的形状，进行精密的对焦操作。



从图中可以看到，那两块斜肌环绕着眼球侧面，可以一起产生侧向的压力，通过挤压使眼球变长，而那四块直肌一起向后用力，就可以让眼球变短。Bates 的实验（通过给这些肌肉通电），也显示了以上的现象。在这 6 块肌肉的相互作用下，眼球随时都在改变形状。

通常认为人眼只是用睫状肌改变晶状体的形状进行对焦，而其实眼球外部的肌肉也同时参与了这个操作，它们通过改变眼球的形状（改变眼轴长度）来进行对焦。你可以认为眼球是一个底片可以活动的相机，它可以通过改变镜头（晶状体）的焦距进行对焦，也可以通过同时改变底片（视网膜）的位置来进行对焦。



相对于晶状体的对焦范围，改变眼球形状能获得的改变很小。也许可以认为晶状体是进行“粗略对焦”，而外部肌肉改变眼球形状是在进行“精密对焦”。因为外部肌肉能改变眼球的形状，所以它们对于近视的形成和逆转起着关键性作用。

近视产生的原理



看近的时候，如果眼球完全放松，成像就会落到视网膜后面。为了看清物体，晶状体需要被压缩，增大屈光。同时眼球的形状也可能被外部肌肉稍微拉长，这样成像就能正好落在视网膜上。看近的时间太长，眼球就可能持续处于被拉长的状态。长期保持这种形状，眼轴就变长了，看远的时候也回不去，以至于看不清远处物体，就近视了。

所以近视产生的根本原因，是长时间紧张地看近距离的物体（比如书，手机，电脑），而跟光线，遗传什么的都没有直接的关系。长时间看近，眼轴就变长，这样睫状肌不费力就能看清近处的物体，所以近视是动物眼睛的一个 feature，而不是 bug。眼轴变长是为了让人看近时更轻松，是一种对生活环境的适应，谁叫你看近的时候那么多呢？

很多人以为小孩的近视是“假性近视”，以为是睫状肌紧张所致，眼球并没有变形，所以求助于各种治疗假性近视的方法。其实小孩的近视也都是因为眼轴变长，是真性近视，所以各种号称治疗假性近视的方法都是不灵的。我认为“假性近视”这个说法，就是为各种不起作用的智商税产品准备的，可以说是一个伪科学词汇。

不近视的“基因”

有些人说自己“基因好”，似乎无论如何滥用眼睛也不会近视，但我发现这种人似乎都有一种特殊的用眼方式。虽然貌似一直在看近，但眼睛其实处于一种“似看非看”的放松状态，似乎并没有聚焦在近处，而是在看屏幕后面某个地方，

或者聚焦在屏幕上很小的一点上。这种情况下晶状体其实是放松的，眼球也没有被拉长，虽然长时间“看近”，却也不会近视，因为他们并不是真正聚焦在近处的。这种用眼方式在 Bates 的书上有介绍，叫做「Central Fixation」。

所以我感觉这种所谓的“不近视基因”，也许并不是生理的遗传，而是心理和习惯上的“遗传”。父母的教育和生活方式传给了孩子，以至于他们看东西养成那种好的习惯，结果就不会近视。反之那种“学术家庭”，父母看很多书，孩子也跟着很爱看书，结果就近视了，然后人们就以为是遗传的。

近视度数与“清晰范围”的关系

如果你近视了也不戴眼镜，度数是很难超过 300 度的，一般在 200 度左右就不会再发展了。因为 200 度近视的眼睛，完全放松的时候，正好能看清 0.5m 远的物体。这个 0.5m 距离叫做 200 度近视的“清晰范围”，在这个距离以内的物体，200 度近视的人都能看清。为什么我知道是 0.5m 呢？公式是这样：

$$100 / \text{近视度数} = \text{能看清的最大距离 (以 m 计算)}$$

练习：根据这个公式，300 度近视的人能看清多远的物体？

根据这个公式，你也可以估算出要产生某个度数的近视，你需要看多近的物体。比如，眼睛需要产生 200 度的近视，就可以在晶状体完全放松的状态下看清 0.5m 的物体。所以持续看 0.5m (及以外) 的物体，能造成的近视就不会超过 200 度。

看 0.5m 以内的物体，基本是需要弯腰曲背的，自己也会发现不舒服，会调整回去。所以如果一个人不戴眼镜，是比较难超过 300 度近视的。

如何防止近视发展

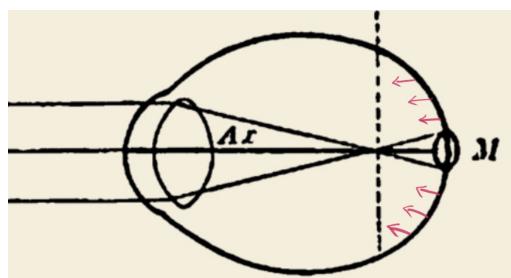
高度近视产生的原因，一般都是戴着全度数（或过度矫正）的眼镜看近。近视的眼睛看近处，本来比正常眼睛更轻松，睫状肌不怎么用力，所以如果看时不戴眼镜，近视就不会发展得很厉害。但戴上全度数眼镜之后，这个让看近变得轻松的 feature 被抵消了，睫状肌仍然需要用力才能获得需要的屈光，外部肌肉继续拉长眼轴，结果眼轴继续加长，近视就不断加深。然后又去验光，配足新的度数，然后就恶性循环了……

所以对于已经近视的人，眼镜最好不要配足 1.0 的度数，平时的眼镜应该至少在 1.0 的度数基础上减掉 25 度。

如果已经近视，度数不超过 200 度。那么注意看近的时候不要戴眼镜，因为戴全度数眼镜看近是近视加深的根源。需要看远的时候可以戴眼镜。另外注意阅读距离不要太近，这样近视就不会发展。

为什么眼科医生和眼镜店都告诉你要配足度数，否则就会加深近视呢？因为你近视度数越高，他们的生意就越好 ;)

近视逆转的原理



近视的自然逆转，关键在于“看远”。注意这里的“看远”是加了引号的。

首先我来解释一下什么叫做“看远”。这里所谓的“远”，并不是指固定的距离，而是指物体在眼球中成像的位置。如果晶状体完全放松之后，成像仍然落在视网膜前面（如上图），对于人的视觉来说，这就叫“远”了。所以对于 1.0 视力的人，6 米以外才能叫“看远”，对于近视 100 度的人， $100/100 = 1$ 米以外就叫“看远”，而对于近视 500 度的人来说， $100/500 = 0.2$ 米以外就可以算“看远”了。

看远的时候，晶状体需要放松，减少屈光。如果晶状体完全放松之后，成像仍然落在视网膜前面，就看不清物体。这时候如果继续注视物体，眼球外部的肌肉就会开始轻轻地压缩眼球，使眼轴变短一点点（可能只有 0.x 毫米），这样模糊的物体就会逐渐清晰一些。如果经常处于这种模糊状态，眼轴反复地小幅度压缩，就会永久性地缩短，近视就会逆转。

所以近视逆转的关键，在于经常处于看远有点模糊的状态。物体不一定需要很远，只要超出了清晰范围就行。比如，你如果已经 200 度近视，那么你不戴眼镜看 0.5m 以外的物体，都能达到逆转近视的效果。

注意这需要是一个长期习惯性的操作，最好是不知不觉，自然而然地进行，成为一种习惯。刻意地每天专门腾出时间来做“练习”，往往难以达到很好的效果。另外注意应该很自然地看物体，不要为了看清物体而忘了眨眼睛，或者盯着看太久，那样对眼睛不好。轻轻地看“远”处的物体，自然用眼就好了。

逆转近视的具体方法

知道了原理，我们来看看具体怎么操作。

最简单粗暴的逆转近视的方法，就是完全不戴眼镜。最近有一个 18 岁的年轻人，本来双眼近视 600 多度，听了我讲的原理之后，开始平时完全不戴眼镜（除了必须看清远处的时候），结果 3 个月之后度数降低了 100 度左右。



高度近视的人完全不戴眼镜生活，当然可能是相当痛苦的，所以我建议稍缓和的方式，也就是戴“低度数眼镜”。方法是这样，如果你近视 600 度，就把眼镜度数降低 100 度，配一副 500 度的眼镜，散光不要减。你戴上它之后看到的景象，就相当于一个只近视 100 度的人，生活几乎不受影响。

100 度近视眼的清晰范围是 1m，所以你只要看 1m 以外的物体，都会达到逆转近视的效果。这副眼镜起到的效果，就是降低了算作“看远”的距离，把戴全度数眼镜算“看近”的地方，很多都变成了“看远”，这样看远的机会就大大增加了。

所以方法就是配一副降 100 度的眼镜，平时都戴它就行了。几个月之后，近视度数应该会下降一些。如果你明显感觉看东西比以前清晰了，估计降了 50 度的时候，就去验光，配一副更低度数的眼镜，保持与实际度数 100 度的差距。如此良性循环……

但降 100 度的眼镜看电脑可能就有点太清晰了，所以如果想见效快一些，可以再配一副降 150 度的眼镜。平时在外面戴降 100 度的眼镜，看电脑的时候换成降 150 度的眼镜，把屏幕放在 $100/150 = 0.66m$ 以外稍有点模糊的地方。这样越是看电脑，视力越是变好。但我不建议把字全都用力看清楚，模糊就让它模糊着，也不要看得太累了，注意休息。

另外需要注意的是，屏幕的字体不要设置得太大。很多人误以为大的字体可以保护视力，其实正好相反。大字体的视角太大，相当于是看更加近的物体。把屏幕字体设置为正常大小，或者稍小一些，这样视角变小，就相当于在看更远的物体。久而久之，视力就改善了。

因为大脑总是选择视力好的那只眼睛提供的图像，所以这个方法总是让视力较好的那一只眼睛变得更好，所以如果两只眼睛视力相差太多，就会更加不平衡。如果两只眼睛视力相差大于 50 度，需要配眼镜让两眼视力比较平均。

这个方法会很缓慢，不要急于求成。想想你的近视是多少年形成的，所以逆转也不会很快。3 个月能逆转 25 度就已经很好了，这样一年就少了 100 度。因为肌肉在悄悄用力压眼球，所以要注意不要太用力看，不要过量。这跟健身不要过量的原理差不多。年龄超过 40 岁的人要特别小心，因为眼球的玻璃体开始老化，如果太过用力地看，可能会导致飞蚊症。如果你太拼，很短的时间之内视力提高了很多，那就应该提高警惕了，小心弄坏眼睛！

我不确定这个方法适合所有人。虽然我认为眼睛应该不像普通认为的那样，成年就定型，但眼球可能确实有一个接

近“定型”的年龄。超过年龄可能就不大容易逆转，或者非常缓慢。如果你尝试了一年还没有一点变化，那可能眼睛已经定型了。

近视度数很高的情况

因为高度近视容易导致视网膜脱落之类的严重问题，所以如果超过了 800 度，最好是先去医院做个眼底检查，没有问题再小心，小剂量地尝试，切不可太用力看东西。

近视度数很低的情况

如果你通过以上的方法，把近视度数减少到了 100 度，或者你的近视度数本来较低，比如低于 100 度，那么光是不戴眼镜，是比较难恢复到 1.0 视力的。这是因为度数太低的时候，外部肌肉产生的眼球形变很小，难以形成持续的改变。

对于近视度数很低的情况，可以采用戴老花镜的办法。比如你近视 100 度，那么可以去买一个 50 度的老花镜。戴上之后就成了近视 150 度的视觉效果，所以你看 0.66m 以外的物体就会开始模糊。你每天戴着这个老花镜看屏幕，近视就会减少。等减少到一定程度，就再去买一个 100 度或者 150 度的老花镜来戴，直到恢复到 1.0 或者更好的视力。

方便的视力评估方法

如果你试验了一段时间，怎么才能方便地知道有没有改善呢？总不能三天两头地去验光，视力表挂家里又太难看。我有个简单的办法，那就是在你经常坐的地方选择一个参照物，最好是上面有字的物体。比如我喜欢坐在沙发上看电视柜上的 Marshall 音箱的 logo，根据它的清晰程度，我就知道我的视力有没有改善。

如果你想精确一些，可以拿一张有字的纸和一个钢卷尺，用卷尺测量每只眼睛能看清的距离。然后根据上面那个公式「近视度数 = 100/清晰距离」，就能得到大概的度数了。或者你光是记住你能看清的距离，用这个作为比较的基础也行。

我不建议频繁地进行这种测量，因为改善的过程非常缓慢，可能起起伏伏，所以频繁测量会带来挫败感。最好是顺其自然，隔长一点时间测，这样也许会有惊喜的感觉。

自然很重要

相比我这种完全“无为”的自然视力恢复法，网络上流传着好些其它方法，比如 EndMyopia 一类的。这些方法一般要求你做一些特定的事情，比如花很多时间做「Active Focus」一类的练习。有些方法让人瞪着眼睛使劲看远，甚至让人不眨眼，看得流泪还忍痛继续看。有些让人用“3D 立体图”来进行“交叉眼”练习，而且严重夸大改善的速度。

这些方法让人以为必须做点什么，必须严格按照他们的说法来，才会有效果，却没搞清楚近视逆转的原理。我也曾经短时间尝试过某些这类练习，但后来我发现，它们都不是自然地在用眼，有可能造成严重的紧张，甚至导致眼睛受伤。后来我发现，其实什么特别的练习都不需要做，只把眼镜度数配低就行了。

这类练习看似“自然”，其实可能是相当危险的。眼睛是有血肉的活物，而且是有弹性的，并不是像面团一样可以随意蹂躏成形的。它不可能在短时间之内就永久改变自己的形状，它可能受伤。疼痛是非常重要的信号，它让你停止伤害自己的行为。要是忽视疼痛的信号，就有可能受伤。所以我不推荐任何会导致眼睛不适的方法，我不相信任何宣扬“忍痛继续，坚持就是胜利”的人。要知道这个世界充满了骗子，他们可能是想害人。

虽然我的做法除了配眼镜之外没有其他特别的练习，最大限度的接近了“自然用眼”，但仍然有可能因为眼镜度数太低，或者看太久，或者因为有一定“野心”而出现异常的疲劳或者疼痛。所以我建议，一旦出现疼痛或者异常的紧张，请立即停止当下的做法，充分休息。如果多次尝试之后仍然不舒服，可以稍提高眼镜的度数，比如只降低 75 度或者 50 度，等眼睛不适感消失，恢复自然之后再继续尝试。

虽然你知道应该“自然”，但这个事情毕竟是有一些目的性在里面，所以有些人可能不自觉的会用眼过度，或者做一些强迫自己的事情。我总结了一下，有一些信号可以提醒自己，出现了不该做的事情：

- 眼眶，太阳穴或眼球出现酸痛
- 视力出现模糊，之前看得清的东西现在看不清了
- 看东西有双影
- 眼睛对焦速度减慢
- 出现飞蚊

遇到以上情况的话，你应该停下来休息。如果经常出现以上现象，就应该检查一下自己的做法了。比如，你是不是经常强迫眼睛去看清太远太模糊的物体？是不是盯着一个地方看太久？是不是忘了眨眼睛？是不是一直在努力看远，而没有看近的时候？等等。

参考资料

1. W. H. Bates - Perfect Sight Without Glasses ([PDF 下载](#))。中文版译名《不戴眼镜的完美视力》已出版。
2. Todd Becker - Myopia: A Modern Yet Reversible Disease ([讲座视频](#))

计算机科学进阶班招生

自从开设计算机科学基础班（集体班，一对一，阅读班）以来，收到了很多学生的好评。我很高兴地看到很多完全零基础的人，真正从零开始，学会了计算机科学最精华的部分。这让我大受鼓舞。从我自己的探索，遇到的各种社会问题，我再次深刻地认识到教育的价值和意义。

在这种前提条件下，我已经有了开设计算机科学进阶班的想法一段时间了。目前已经有 13 岁的小白鼠同学试了一节课，我发现一个孩子真能在短时间内学会一些曾经只有博士生才能学会的东西。我也曾经试过即兴教一个朋友大提琴。之前她从未碰过大提琴，结果只花了一个小时，就能拉出巴赫 Prelude 的第一个小节。当然仍然是不流畅的，但原理已经掌握，自己多练就会不断提高。掌握本质原理，不断提高。师父领进门，修行在个人。我感觉这是我领悟到的教育的真谛。

关于计算机科学进阶班，我还没有非常确切的内容和计划，但最初开设基础班的时候，其实也是一样的情况。除了前两节课的内容，后面的其实都是我根据教学情况，临时想出来的。但我最好还是大概列出一些内容，让同学们心里有个底。

课程大纲

大概的内容范围如下，虽然可能临时根据情况更新或增删：

1. CPS (continuation-passing style) 变换基础。从零开始，理解 CPS 是怎么回事。
2. 使用 CPS 实现程序的并发执行。根据 continuation 这个统一的程序语言概念，理解操作系统的进程，线程，以及 Go 语言和 node.js 等系统的所谓“大并发”。
3. Scheme 语言基础。因为 Scheme 语言的干净设计，我们将切换课程语言，用 Scheme 来学习后面的内容。学生会用 Scheme 实现递归函数，然后用它写一个类似基础班的解释器。
4. 理解和实现 call/cc 操作符。理解 call/cc 是什么，并亲自实现 Scheme 语言的 call/cc 操作符，而不只是会使用它。然后使用 call/cc 做一些有趣的事情，比如实现并发。
5. 理解和实现“自动 CPS 变换”。这个就是人们常说的“王垠 40 行代码”，它的作用是自动把普通程序自动转换成 CPS 形式。学生将理解它的原理，并且自己实现自动 CPS 变换。
6. 使用“自动 CPS 变换”的思想，实现一个微型编译器。这个领域叫做「compiling with continuation」。它使用 CPS 变换，ANF 等理论基础来设计编译器，而不是传统的方式。虽然这个编译器的语言非常简单，但它揭示了编译器的本质，理解它会帮助学生将来写出任意复杂的编译器。如果有兴趣，学生可以自己扩展这个语言。
7. 逻辑编程 (logic programming) 基础。理解并实现一个类似 Prolog 的简单逻辑编程语言。由此从根本上理解逻辑编程。使用逻辑编程的思想，实现 OCaml 和 Haskell 的 Hindley-Milner 类型系统。
8. 程序的形式化证明。使用 Coq，学习形式化证明系统的使用。与程序测试不同，通过形式化证明，我们可以完全保证程序的正确。通过简单的例子，讲我将述形式化证明最根本的思路和原理。在某种程度上，这些思想就是数学证明的本质。

课程初步计划，应该有至少 8 节课。课程会有专门的交流群。每节课后，学生可以在群里讨论自己的理解和发现。我会临时想一些小练习给大家做，但这个课程相当于是大学“研究生班”的级别，所以课后主要是开放的研究方式进行思考和探索，而不是做固定的练习。我也许会建议一些论文给大家看，或者自己实现一些以前没试过的事情，或者探索学生自己感兴趣的方向，或者做一些小型工程实践。

收费标准

由于内容的难度很大，所以我不不是很确定 8 节课能够完全讲清楚这些内容，可能会超出一些。我打算按照一节课 1024 人民币进行收费。课程的单位课时学费比基础班要低。最初的学费是 8192，包括 8 节课，每节课 2 小时。虽然按我的性格会尽量精炼，但如果无法在 8 节课完成，也许会延续 1~2 节课。延续的课时不另外收费。

由于课程内容会用到基础班的很多知识，课程的招生对象主要针对基础班（包括一对一和阅读班）已经顺利毕业的同学，但如果有自学过 Scheme 或者函数式编程（比如 SICP）的其它学习者，我也可以考虑他的加入。

报名方式

课程只招收基础班已经毕业的学生。基础班已经毕业学生，可以发微信给我报名就行。注意，请勿用微信转账。由于进阶班对基础要求比较高，课程不接受没有参加过基础班，或者基础班未能完成的学生。

计算机科学阅读班招生说明

计算机科学阅读班，是一门零基础，目标在于短期内掌握计算机科学精髓的课程。它是我 20 多年的计算机学术和工程实践，加上两年面向社会的教学实验的结晶。课程吸取了世界上主要的计算机入门教学方式的优点，避免了它们阻碍初学者理解的各种问题，以至于完全零基础的学生也可以在短短两个月之内，掌握大学博士阶段才可能学到的精华内容。这些内容足以建立起坚实的知识基础，使得他们对于理解计算机科学的其他方面从容自如。

课程使用特别的授课方式——阅读加练习辅导的方式。阅读内容为我正在写的书《[Ground-Up Computer Science](#)》(GUCS)，练习辅导是一对一的微信辅导方式。虽然课程只有 7 节课，但是由于每个练习会及时得到提示和反馈，实际的学习时间大大超过传统课堂。这种教学方式避免了普通课堂的各种不灵活性，给予了学习者最大的方便和灵活性。这使得工作学习繁忙的人士也能抽空完成学习。一般学生能够在两个月之内完成课程。对于比较繁忙的学生，时间可以放宽到 4 个月。

现在阅读班的形式已经基本成型，我觉得适合长期面向社会招生。

适用人群：

- 从中学到博士阶段的各专业学生，不管文科还是理科生都能学会。虽然课程已经成功让一个 [13 岁少年](#)深入地掌握了这些内容，但对于中学生，希望学生本人对计算机科学感兴趣。如果只是家长的期望，恐怕学生会比较难以用心去阅读。
- 已经从事工作的人员，包括 IT 从业人员，IT 管理类工作和其它工程类工作人员。
- 其它各类对计算机感兴趣，想把它作为业余爱好的人员。

学费和报名方式：

- 学费对于已经工作人员是 10240/人，对于无收入的在校学生（中学，大学本科，但不包括研究生和博士生）是 8192/人。对于经济特别困难，但有求知欲望的学习者，可以来信询问特殊价格。
- 报名请发送 email 到 yinwang.advising@icloud.com。标题为《计算机科学阅读班报名》。来信请说明自己的基本信息，附件发送一个简历，提供微信联系方式。请写一段 200 字左右的“个人说明”，说明你的求学动机。对于符合要求的求学者，我会进行简短的微信语音面试。
- 报名者可以要求一节课的试课。试课费用是 300 人民币。

教学语言。课程目前使用 JavaScript 作为教学语言，但并不是教 JavaScript 语言本身，不会使用 JavaScript 特有的任何功能。课程教的思想不依赖于 JavaScript 的任何特性，它可以应用于任何语言，课程可以在任何时候换成任何语言。学生从零开始，学会的是计算机科学最核心的思想，从无到有创造出各种重要的概念，直到最后实现出自己的编程语言和类型系统。

课程强度。课程的设计是一个逐渐加大难度，比较辛苦，却很安全的山路，它通往很高的山峰。要参加课程，请做好付出努力的准备。在两个月的时间里，你每天需要至少一个小时来做练习，有的练习需要好几个小时才能做对。跟其他的计算机教学不同，学生不会因为缺少基础而放弃，不会误入歧途，也不会掉进陷阱出不来。学生需要付出很多的时间和努力，但没有努力是白费的。

课程大纲：

第一课：函数。跟一般课程不同，课程不从所谓“Hello World”程序开始，也不会叫学生做一些好像有趣而其实无聊的小游戏。一开头我就讲最核心的内容：函数。关于函数只有很少几个知识点，但它们却是一切的核心。只知道很少的知识点的时候，对它们进行反复的练习，让头脑能够自如地对它们进行思考和变换，这是教学的要点。我为每个知识点设计了恰当的练习。

第一课的练习每个都很小，只需要一两行代码，却蕴含了深刻的原理。练习逐渐加大难度，直至超过博士课程的水平。我把术语都改头换面，要求学生不上网搜索相关内容，为的是他们的思维不受任何已有信息的干扰，独立做出这些练习。练习自成系统，一环扣一环。后面的练习需要从前的练习获得的灵感，却不需要其它基础。有趣的是，经过正确的引导，好些学生把最难的练习都做出来了，完全零基础的学生也能做出绝大部分，这是我在世界名校的学生里都没有看到过的。具体的内容因为不剧透的原因，我就不继续说了。

第二课：递归。递归可以说是计算机科学（或数学）最重要的概念。我从最简单的递归函数开始，引导理解递归的本质，掌握对递归进行系统化思考的思路。递归是一个很多人自以为理解了的概念，而其实很多人都被错误的教学方式误导了。很多人提到递归，只能想起“汉诺塔”或者“八皇后”问题，却不能拿来解决实际问题。很多编程书籍片面强调递归的“缺点”，教学生如何“消除递归”，却看不到问题的真正所在——某些语言（比如 C 语言）早期的函数调用实现是错误而效率低下的，以至于学生被教导要避免递归。由于对于递归从来没有掌握清晰的思路，在将来的工作中一旦遇到复杂点的递归函数就觉得深不可测。

第三课：链表。从零开始，学生不依赖于任何语言的特性，实现最基本的数据结构。第一个数据结构就是链表，学生会在练习中实现许多操作链表的函数。这些函数经过了精心挑选安排，很多是函数式编程语言的基本函数，但通过独立把它们写出来，学生掌握的是递归的系统化思路。这使得他们能自如地对这类数据结构进行思考，解决新的递归问题。

与一般的数据结构课程不同，这个课程实现的大部分都是「函数式数据结构」，它们具有一些特别的，有用的性质。因为它们逻辑结构清晰，比起普通数据结构书籍会更容易理解。与 Haskell 社区的教学方式不同，我不会宗教式的强调纯函数的优点，而是客观地让学生领会到其中的优点，并且发现它们的弱点。学会了这些结构，在将来也容易推广到非函数式的结构，把两种看似不同的风格有机地结合在一起。

第四课：树结构。从链表逐渐推广出更复杂的数据结构——树。在后来的内容中，会常常用到这种结构。树可能是计算机科学中最常用，最重要的数据结构了，所以理解树的各种操作是很重要的。我们的树也都是纯函数式的。

第五课：计算器。在熟悉了树的基本操作之后，实现一个比较高级的计算器，它可以计算任意嵌套的算术表达式。算术表达式是一种“语法树”，从这个练习学生会理解“表达式是一棵树”这样的原理。

第六课：查找结构。理解如何实现 key-value 查找结构，并且亲手实现两种重要的查找数据结构。我们的查找结构也都是函数式数据结构。这些结构会在后来的解释器里派上大的用场，对它们的理解会巩固加深。

第七课：解释器。利用之前打好的基础，亲手实现计算机科学中最重要，也是通常认为最难理解的概念——解释器。解释器是理解各种计算机科学概念的关键，比如编程语言，操作系统，数据库，网络协议，Web 框架。计算机最核心的部件 CPU 其实就是一个解释器，所以解释器的认识能帮助你理解「计算机体系构架」，也就是计算机的“硬件”。你会发现这种硬件其实和软件差别不是很大。你可以认为解释器就是「计算」本身，所以它非常值得研究。对解释器的深入理解，也能帮助理解很多其它学科，比如自然语言，逻辑学。

第一位“计算机科学少年班”学生毕业

最近有一件让我很欣慰的事情，经过一段时间的实验教学，我的第一位“少年班”学生（13岁）已经完成计算机科学基础班的全部课程，并且超额完成了成人学生难以完成的某些难度大的“进阶练习”。教学的过程比教成人还轻松，有些课只需要一半的时间就讲完了，还有一节课作为练习直接给他了。这个成功案例让我更加的相信了教学的有效性。

某人说“如果你不能给6岁小孩讲明白，那你不是真的懂”，我的另一个实验却显示这个说法是不切实际的。与另一位小朋友（6岁）的教学实验说明，6岁的小孩由于心理不够成熟，恐怕难以对课程内容感兴趣，因为我没有设计“小兔子”，“小乌龟”这样的故事。

但13岁的少年显然已经成熟到能够兴致盎然地做练习，牢固地理解博士级别的知识。想起我自己13岁的时候也是一个样子，每天放学回家都在和同学谈论宇宙是怎么回事。我开始想象，如果我13岁的时候有我这样的老师，我现在会是什么样子。真希望有“穿越时空”的本事。

“超人类机器视觉”是不存在的

我想再向大众解释一下，人工智能（AI）领域的“历史性突破”——“超人类机器视觉”是怎么来的。之前的[文章](#)解释过，但可能信息被埋在比较长的内容里，很多人没看到，以至于仍然蒙在鼓里。对于这种关键问题，简洁易懂是非常重要的，所以我决定在极短的篇幅之内把它解释清楚。

简言之，AI 领域的所谓“机器视觉”，一个重要问题就是让机器回答“图片上是什么东西？”这个问题。比如图片上是一辆汽车，如果你说是“汽车”，就算对了。所谓“Top-5”标准，就是每张图片给 5 次机会，你说出 5 种东西的名字，只要其中一个对了就算对。比如图片上是一辆汽车，你说“猫，吉他，风扇，汽车，橙子”，这也算你对了。已经察觉到问题了吗？我们继续……

用大量图片做这个测试，统计“识别率”，这就是很多机器视觉专家做的事情。有人用这种方式做了一个“人机对比”实验，发现机器的识别率超过了人，这就是“超人类视觉”（super-human level vision）的由来。“超人类视觉”被认为是 AI 领域的历史性突破，也就是这些年 AI 如此火热的原因。

但很多人没看出来，“Top-5”是非常不科学，不合理的。因为如果是人见过的东西，他只需要一次就能对，毫不含糊，另外 4 次机会完全没必要，而机器经常一次猜不对，需要另外 4 次机会才能蒙混过关。人精确地知道这个东西是什么，而机器只是在猜测“它可能是 A，也可能是 B, C, D, E”，一共 5 种可能。所以对于同一个图片，虽然人和机器按照“Top-5”都算“对了”，他们的准确程度其实大不一样。这就像设计一个考试，每道选择题本来只有一个正确答案，但却给了 5 次机会做对。这样就没法分清优等生和差等生了，甚至差等生有时候表现比优等生还好，因为不确定知道答案，瞎蒙都能做对。

我想一般人都理解这里的问题，然而“Top-5”标准却是 AI 领域所谓“超人类视觉”的来源。其实就算用如此不公平的标准，机器的识别率也没超出很多，几乎可以作为“噪音”忽略。当年参加测试的只有一个人，这个人不是从别处请来的“独立实验者”，而是参与此项目的一个学生。这个人的名字叫 Andrej Karpathy，他后来成为了 Tesla 公司的 AI Director。根据如此偏颇，甚至可能是作弊的方式，他们宣称“机器视觉超越了……人类”。

你可能以为“Top-5”虽然不科学，不合理，也许也没什么太大的害处。你可要小心了，它其实可以致命，而且正在威胁着很多人的生命。人的生存环境里，往往是没有 5 次机会来判断一个东西是什么的，实际上经常只有一次机会，不能有任何含糊。比如，在马路上把“卡车”识别为“白板”，是可以致命的。然而这种识别错误，就是 Tesla 的 Autopilot 多次导致致命车祸的原因。详情可以参考我这篇 2016 年的[文章](#)。

这个测试其它的问题还有很多。比如，测试用的图片都是光照良好情况下的清晰图片，没有自然环境的各种复杂性，比如暗光，夜景，遮挡，阴影，反光，镜面，折射，模糊等。另外，只识别出物体“叫什么名字”，并不等于知道了它的 3D 形状和边界，并不等于可以拿起，操作，或者避开物体，并不等于可以依靠这个技术来做“自动驾驶”。

『Ground-Up Computer Science』样章



经过两个月的阅读班实验，我的计算机科学入门教材已经完成了前面 6 章的草稿，还剩最后两章正在继续写。这本书暂命名为『Ground-Up Computer Science』，就是“从地而起”，不需要其它基础的意思。

完善这本书，到最后发表肯定需要更多时间，所以我先把第一章的草稿发布在这里，提供给大家作为入门学习之用。准备参加计算机科学基础班的同学，也可以通过这个样章预习第一课的内容。

第一章虽然是最基础的内容，它却是这本书里面最长的一章，所以信息含量其实是相当大的。用这一章的内容打好基础之后，后面难度高的章节反而越来越短。

因为草稿会被经常改动，所以里面的插图都是用 iPad 手画的，比较潦草，颜色也没有很好的搭配，请大家见谅。正式出版的时候会把图片重新画一遍。另外，对话的两个人物的名字也没有想好，所以暂且叫 A 和 B。A 是老师，B 是学生。

关于版权。这个文档的内容受版权法保护，我保留一切版权。仅供个人阅读和学习，不能将其作为商业用途。请勿转载或拷贝这个文档到其它网站，也请勿翻译。这个博客上的链接应该作为这个样章唯一的来源，链接是可以分享的。如果发现有违反版权约定或抄袭的现象，请来信告诉我。如果发现有错误或者可以改进的地方，欢迎跟我联系。

样章可以在这里下载：

[第 1 章](#)
[第 2 章](#)
[第 3 章](#)
[第 4 章](#)
[第 5 章](#)
[第 6 章](#)
[第 7 章](#)

因为是样章，所以不提供练习和指导。

计算机科学基础班（第 4 期）报名

经过两个月的教学，计算机科学基础班第 3 期已经顺利结束了。现在可以开始接收第 4 期基础班的报名。详细方式和[第 3 期](#)一样。

由于课程对某些同学负荷较大，所以一般我们会在课程结束后的两周之内仍然指导之前没做完的练习，五一放假不算在内。第 4 期课程预计可以在两周后开始。

第 3 期课程有好几个完全零基础的同学，来自各种不同的背景。第 3 期同学有学作曲出身的音乐人，有全职妈妈，有自学入门的人，也有在校大学生，已经毕业进入工作，或者创业的人。

我有两位出色的助教同学，他们为这次课程付出了很多的辛苦。有时候早上起来，发现前一天晚上零基础同学的辅导群里有几十上百条的耐心辅导，我都很感动助教同学们的认真和耐心。虽然我还是鼓励大家按时睡觉 :)

课程进行的这几个月，我自己也学到很多东西。我再次深刻地体会到作为一个领域初学者的不易。我发现每一个领域本来都应该有一套顺利而有效的学习方法，但总是有很多误导。因为从不同的老师得到的学习方法不同，有些人几十年的努力，可能还不如别人几个月的。因为进行的道路不同，所以效果也不同。

我越来越相信一套正确的学习方法，一套有效的练习的重要性。人类的认知和学习能力是一个非常有趣的领域。我越来越觉得，与其研究所谓“人工智能”，真的不如好好的研究一下如何让人类更好的学会知识。为此我经常把自己的头脑和身体作为实验对象，用不同的方式来测试和对比它们的反应，试图寻找最有效的方法。我不相信那种一条路走到底的思路，那往往是人们花太多时间还学不好的原因。

通过自己亲自参加好几个领域的学习，我体会到了付出那么多学费，最后得到的是什么。在今天的社会，付出很多金钱和时间，却很有可能什么也学不到。我更加地相信我正在做一件有利于社会的好事。只有大多数人都获得真的知识，社会才可能朝着好的方向发展。

计算机科学基础班不会讲述计算机的所有知识，但它却是获得更多知识的一套核心知识和方法，它应该可以帮助人们寻找和理解更多的计算机知识。这套方法肯定还需要改进和完善，我将不断的对它做出调整。

[计算机科学阅读实验班](#)也进行到第 6 课，反馈良好。我正在开始撰写最重要的第 7 章（解释器）内容。阅读班目前人数饱和，所以暂时不能接受新的学生。全部内容完成一遍之后，会开始接收新的申请。

由于撰写文档比直接教学复杂很多，而且和心情很有关系，并不是所有时候都能写出好的内容，所以进行慢了一些。近期我会对第 1 课的内容文档进行整理，公开在这里。希望能对大众有所帮助，也给准备进入基础班的同学提供预习内容。

大提琴为什么这么难学

(本文记录了我开始“破解大提琴计划”一个月以来的阶段性发现。图片比较多，虽然经过压缩，还是需要一定时间，请耐心等待加载。)

一个月之前决定开始学大提琴，找了一个老师上了几节课，并且在 YouTube 看了许多大提琴教学视频，每天琢磨和练习。然后才发现，大提琴原来是世界上最难学的乐器。不仅乐器本身难用，而且教学也很成问题。

其实大提琴的教学和网球等体育运动的教学差不多。反思其中的道理，我觉得应该把它们记录下来，分享给关注教育的人们。如果你是大提琴或者小提琴演奏者或者学生，这篇文章也许含有你需要，却没有任何老师会告诉你的，这类乐器背后的秘密。

机器人的教学方式

这些领域的教育，共通的问题是什么呢？就是特别死板，把人当成机器在调试，而不是作为具有自我调节能力的生命体。比如大提琴老师们会说，拿琴弓的时候这根手指要放这里，那根手指要放那里，这个关节要什么角度，手腕，肘部和肩部分别什么时候动。似乎很科学很精确的样子，可你就是没法照做。

如果你知道工业机器人投产之前是怎么“训练”的，就会发现那些都是给机器的指令，而你不是机器。



实际上，训练工业机器人的做法恐怕都要聪明一些。现在的工业机器人已经有了自动路径规划，自动避障等能力，不需要指定每一个关节的运动。可是我们人类的教育者却似乎假设了我们没有这些基本能力。

当年上网球课也是类似，连机器人都不如的待遇。老师指定了握球拍的每根手指该放哪里，然后告诉你蹬腿，转腰，跟我挥拍，第一步到这个角度，第二步开始转体，到这个角度，定格！好了，第三步……却不告诉你这些细节的位置是要达到什么目的。

多年以后去学泰拳，遇到一样的教法。我永远也记不住拳头要用什么角度挡在下巴那里，只记得老师一直说我的手角度不对，但纠正了无数次都无效，因为他没告诉我为什么拳头一定要是那个角度。如果说“你那个角度挡着，别人的拳头是会打进来的”，并且来几拳让我试试，我就明白了，可是他只是像量角器一样看着我，要纠正到他认为对的地方。最糟的是，手的角度还没对，他又开始纠正脚的角度。等你注意脚那头去了，手的角度又错了。这是一种非常不舒服的感觉，教学效果也非常差，可以说最后什么都没学到，还差点把身体弄出毛病来。

从传奇的法国大提琴家说起

我不相信这类活动的教学应该是这样进行的。可惜这种教学方式，不仅包括了普通音乐培训班的老师，而且包括一些有名的大提琴家。比如 YouTube 上有法国“传奇”的大提琴家 Andre Navarra 的录像《[My Cello Technique](#)》。他有多传奇我不知道，但在多方面采集信息并研究实践之后，我发现他有些方面挺误导人的。比如，他说小指要放在琴弓那个“眼睛”的位置，拇指要弯曲，拇指指尖的一角要放在那个不舒服的顶起来的地方。可是很多人就照做了，并且照这样教别人。



我开头也照做了，结果拇指指尖顶在那个叫“尾库”（frog）的部件顶起来的地方，就开始痛。



很明显，frog 是拿来固定和调整弓毛的，顶起来的那个部位不像是拿来抓握的地方。可是都说要把拇指放那里，而且不能放指肚，只能放指尖的一角。前面一点的位置明明裹着一块皮，看起来是给手指拿的地方，但他们偏不让你放那里，一定要你放在那个最不像是放手指的地方。他们说开头就是会很痛，等那里长了茧和厚皮就好了。结果后来我就长了茧，然后才发现，其实根本没有必要拿在那个地方。果断换了一个姿势，拇指就自然拿在有皮的那个地方，现在不但拿得更稳更准确，茧都快没有了。

你觉得最初设计琴弓的人是傻子吗？裹一块皮在那个地方却不让你用它，真正抓握的地方却没有舒服的垫子。琴弓的设计者不傻，最初使用这些琴弓的人也不傻，傻的是后来误用这些设施的人。

大提琴的领域有很多这种反直觉的教条。你觉得本来应该这样做的事情，他们告诉你不要这样做，让你用一个很别扭的姿势。下文里面还有好几个这样的例子。

另一些大提琴家，跟你说拿弓的手腕要“放松”，可是手腕真的能放松吗？手拿着琴弓那么顶端的位置，由于杠杆作用，会让手有挺大的受力。手腕需要支持琴弓的重量，在这种情况怎么可能放松？其实本来应该说，手腕不要僵硬，应该可以灵活自如地运动，但这不等于它应该放松。如果你真的放松了，弓就掉下去了。至于手腕为什么要灵活，他们也没有从原理去解释。

其实只要多看一些顶级大提琴家的表演视频（比如 Pablo Casals）就会发现，他们很多人都没有遵循这些教条，用什么姿势的都有。Casals 的小指经常不在弓杆上，就算有时候在弓杆上，也不在那个“眼睛”的地方。而且他的琴放得很低，这也违反了一些人的“标准高度”，说琴的那个角要正好在膝盖的高度。



观察其他的大提琴家，小指也不一定是在圆圈处。有的可能会把小指压在弓杆上，这样可以帮助翘起弓杆。有趣的是，并没有任何人教我，把弓放在弦上之前，我往往会不知不觉把小指压在弓杆上面，因为这样特别稳定。稍后小指可能会去别的位置，我也不知道它具体去了哪里，我只知道一件事：琴弓应该如何运动。可是总有人说，小提琴才那样小指放上面，大提琴不应该这样。

右手拇指要弯曲？



很多人说拿弓的右手，拇指第一关节要弯曲，不能直着。可是你仔细看看马友友的[视频](#)，他的拇指就是直着的。放大了看，拇指是直着捏着弓杆的，控制拇指的那块肌肉是紧张的，鼓起来的。也许他的拇指并不总是伸直的，但不像很多人教的，一定要弯着。



这个[视频](#)角度更清晰。



还有这个[视频](#)，能看见不论弓在什么位置，拇指都是伸直的，稳稳地按着琴弓。很明显拇指第一关节处于完全“锁定”的状态，那力道，按得关节都快翻过来了。





这个[视频](#)里可以看见，其实手可以拿在其它位置，他用这个靠前的位置拉出了一整首巴赫大提琴曲。有趣的是，弓还在动的时候他也能自由地调整握弓的姿势。后来我才知道，这种握弓的位置更靠近弓的重心，叫做“巴洛克式握法”(Baroque bow-hold)。



可能因为 Pablo Casals 前无古人后无来者的录音，马友友的巴赫大提琴曲 No.1 Prelude 一直不是我的最爱，但不得不承认他的其他曲目都挺好的，是非常好的大提琴家，也是一个很有启发意义的教育者。所以呢，他的拇指姿势应该是说明一定问题的。

所有人都说拇指应该弯曲，所以我试图按照其他人的说法弯曲拇指，却总是发现它不自觉地就伸直了，我为此还在疑惑。可能因为要稳定地握住一根杆子，拇指自然就会伸直。马友友的握弓姿势似乎证明了，拇指其实不一定要弯着。我觉得又一个困惑得到了解脱，我只需要顺其自然。

当然也不能说拇指一定要伸直，那就成为另一种教条了。我只是说它可以伸直，这没什么错。不只马友友一个人是这样拿弓的。可以观察一下其他大提琴家，应该各种姿势都有，但我发现大部分人的拇指都是直的。





除了 Andre Navarra，我只发现一个人的拇指随时都弯着，一丝不苟按照姿势来，那就是 Mischa Maisky。不过他不是我最喜欢的大提琴演奏者。



关于拇指是否应该弯曲，可以参考一下其它类似活动，比如剑术。我发现琴弓的用法和剑差不多，它们形状类似，都是一根长杆子，单手握住一端，需要非常精确和迅速地运动。如果拿剑的拇指总是弯着，只用指尖的一角接触剑柄，你对剑的控制能力将会如何？





实际上，我发现一个很好的练习握弓姿势的办法，就是把琴弓当成剑一样练一会剑法。当然，小心不要伤到自己或者旁边的人就好。

左手四指要弯曲？

类似“拇指要弯曲”的要求，还针对左手按弦的四根手指。很多大提琴老师告诉你，左手那四根手指都要弯曲成拱形，不要伸直“锁定”，如下图。



仔细看看大提琴家们实际的姿势就会发现，他们经常是伸直了锁定住的，不然有些音是不可能同时按下去的。比如我见过一个很好的大提琴家，出了挺多唱片的，他经常是这样按的。



而且他的左手拇指也违反了另外一个教条“拇指不要捏”，本文后面会讲。马友友左手的手指经常也是伸直的，甚至按得关节反转，不然用不上力。比如这个图片里就是马友友的手，你可以在这个[视频](#)里看到。



“弯曲左手四指”这个教条我也照做过，后来发现因为这样用力不稳定，而且如果手指不能伸直，一根手指是很难同时按下两根弦的，这大大拖延了学习的进度。一旦打破这个教条，就感觉好多了，进步也快了。

还有一种更傻的做法，就是把左手四根手指同时放在第一把位的四个音上面，每根手指对准一个音，就像这样。还有的会说，拇指一定要在后面对准中指，有的说一定要对准食指。其实根本没有这样的标准。这种对准关系根本帮不了什么忙。



他们甚至要你能够把四根手指从天而降，落在四个音上面，而且每一个都要对准。而其实人手的构造是中指和无名指挨得很近，非常难分开，所以四根手指同时对准正确的位置是非常难的，因为这些位置违反了人体的构造。除非你把手关节弄得畸形，否则是做不到这个的。

很多老师，很多视频就是这么教的，但你仔细观察 Casals 之类的大师，他们的指法根本就不是这样，几乎从来没有四根手指同时按下去的时候。实际上四根手指同时对准同一根弦的四个音是没有用处的，先后能按出来就行了。好一点的老师都会告诉你，不要把多余的手指按在琴弦上。

巴洛克时代的大提琴

其实在 18 世纪巴洛克时代，大提琴并不是现在这样用的。当时的大提琴没有下面那根杆子（endpin），而是架在两腿之间。琴弓也跟现在不一样，所以拿琴弓的时候，位置和手势都不一样，像餐刀一样拿着就好了。琴弦是羊肠做的，音色比现在的金属弦要柔和，手感好。调音频率是 415Hz，比现在低了半个音。因为琴弦张力小，对琴的压力小，琴的振动好很多，弦按起来也轻松。

比如这幅画里，是 18 世纪意大利作曲家和大提琴家 Luigi Boccherini。你会发现那个时候使用大提琴的姿势跟现在的很不一样，而且相比现在的姿势有好些优点。



我试过把琴下面的杆子收起来，把琴架在小腿上，而且采用巴洛克式琴弓握法。你猜怎么的？我发现琴的稳定性高了很多，声音似乎也变得更好了，而且这姿势并不像很多人想象的那么累。再把音调到 415Hz，立即发现音色好了许多，简直成了另一个乐器。感觉这琴出了一口长气，轻松了！用 415Hz 试着练了一会，音色如此之美，就再也不想调回 440Hz 了。LinnStrument 的软件也被我调成了 415Hz，练了几遍巴赫的曲子，后来再试 440Hz，就忽然发现不好听了。我打算今后就用 415Hz 了。

很可惜，我听说现在国内学乐器已经普遍调成了 442Hz，为了“穿透力”。这是迫击炮还是火箭筒？

从这幅图里，你还可以看见琴颈的位置不在肩膀上，而是更靠侧面，所以他的眼睛是可以看见左手的位置的。虽然熟悉了可以基本不看手，但偶尔瞟一眼，会大大提高准确性。现在的大提琴都架在肩膀上，大部分位置都看不见左手，操作起来跟盲人一样。我猜这也是大提琴比其它乐器难的原因之一。小提琴，吉他，钢琴，double bass，都没有这种情况。

人们总以为以前的做法是落后的，说 endpin 是后来才“发明”的，而其实新的发明也许并不那么好。巴洛克时代的制琴师能做出那么好的琴，你觉得他们会连一根撑地的杆子也不会设计吗？我觉得 endpin 的设计只是为了长时间放那里不累，特别适合在一个大乐团里滥竽充数，只是有时候动一下，协助演奏那些冗长得让人打瞌睡的交响曲。那种情况，一直放在腿上确实显得多余，但如果演奏巴赫那样精彩的音乐，根本不需要这个东西。

现在已经有少数大提琴家采用完全复古的方式，演奏了巴赫的 Cello Suite。比如一个叫 Ophelie Gaillard 的法国大提琴家，不仅采用了巴洛克式姿势，而且全套装备都是巴洛克式的：巴洛克大提琴，巴洛克琴弓，羊肠弦，415Hz 调音。音色如此之美，现在这个专辑已经和 Pablo Casals 的版本一起，成为了我最喜欢听的大提琴音乐。



看看下面这些历史画作里的大提琴姿势。很多人是拿在弓下面的，跟二胡似的，叫“下握法”。有人做过历史研究，发现在那个年代，几乎一半的人是下握法。为了理解这些姿势，我买了一个巴洛克式琴弓，并且试了这个握法，发现比起“上握法”在某些时候有优势，当然也有缺点。







他们拿大提琴的方式接近吉他，更靠侧面，眼睛看得见左手。这些姿势和氛围是很亲切的，都是一些小型聚会，大家在一起玩。不像现在的大提琴，只出现在音乐厅里，都是正襟危坐，非常严肃地在演奏。



从 Viola da Gamba 想到的

在文艺复兴和巴洛克时代，有一种很流行的乐器叫 viola da gamba（维奥尔琴，简称 viol），它是路易十四最爱的乐器，有一些大提琴没有的优点。在当时，viol 是比提琴地位高的乐器，稍富裕点的家庭会有放 viol 的橱柜，里面有各种大小的 viol。

可是历史变化无常，好的东西经常得不到弘扬。虽然现在不再流行，你仍然可以买到这个乐器，而且仍然有人用它演奏很美的音乐。巴赫的学生 Carl Friedrich Abel 为它作出了重要的乐曲。巴赫自己也为这个乐器作过一些乐曲。

有一部法国电影叫「Tous les Matins du Monde」（中文片名：日出时让悲伤终结，优酷上有），讲一个音乐家 [Marin Marais](#) 和他的老师 [Sainte-Colombe](#) 的故事，他们用的乐器就是 viol。这部电影画面和音乐都极美。可以注意看看他们的姿势，包括大提琴的姿势。跟现在扛着火箭筒上前线一样的大提琴姿势比较起来，要轻松和优雅很多。





Viol 琴颈上有羊肠弦做成的品格 (frets)，所以找音比大小提琴容易很多。弦比较多，角度没那么大，所以能奏出三音和弦。虽然这些不一定是好事，但我们应该知道有这种方式存在，而使用大提琴的困难也许不是必须的。

至于 viol 为什么后来不再流行，按照 [Wikipedia 中文版](#) 的说法是：

由于它产生不出小提琴的辉煌和感染力的效果，因而在十七世纪后期开始失宠，渐被以维奥尔琴为蓝本，但运弓和指法接近小提琴的大提琴和低音提琴取代。

而 [Wikipedia 英文版](#) 的说法是：

Viols fell out of use as concert halls grew larger and the louder and more penetrating tone of the violin family became more popular.

翻译：Viol 不再流行，是因为音乐厅越来越大，使得音量更大，音色更有穿透力的小提琴家族更受欢迎。

对比中英文 Wikipedia 两种说法，你应该会发现中文作者和英文作者对事情理解程度的重大差异。去听听 viol 的[效果](#)吧，你会发现中文 Wikipedia 的说法不能离事实更远了。历史上有多少好东西，都埋没在这类作者的以讹传讹中了。

我比较相信英文版的说法，人们因为音乐厅越做越大，最后因为音量和所谓“穿透力”而选择了大提琴和小提琴。大提琴音量真的很大，大得你会担心吵到邻居，甚至损害自己的听力。这一切是为了美吗，还是功利？我不得不怀疑 viol 的失宠是人类的愚昧所致，就像计算机领域的人们选择了“面向对象语言”一样。

嗯，回到正题。下握法有一个显著的优点，就是中指和无名指都在弓毛上，它们可以感觉到琴弦的振动，可以随时调整弓毛的松紧程度，产生微妙的音色变化。另外，我发现下握法手腕会轻松灵活一些，可以自如地产生微妙的抖动。

人可以选择自己喜欢的握弓法，但很多大提琴学生恐怕从来没试过这些做法，因为老师告诉了他们什么是“正确姿势”，连每个手指要放哪里，拇指要弯曲这些事情都规定了，不照做就别想继续学。通过亲身体验，我感觉音乐老师比起美术老师，更有一种高高在上的味道，不管是国内的还是国外的，都喜欢使用权威，我们搞的是高雅艺术，你非得照我说的做。

左手拇指不要捏？

大提琴教学还有另一个疑似误导的说法。大提琴老师们都跟你说：“按弦的时候左手的拇指不要捏琴颈。应该用手臂重量压下琴弦，而拇指只是轻轻摸着琴颈，应该像拿着一颗草莓一样，不能用力。”我照这个说法试了，发现这使得左手的操作很不稳定。拇指如果一点都不用力，按弦的时候琴就得靠胸口来稳定，而胸口却不是一个稳定的依靠点。琴

身靠在胸口，琴颈其实在半空中，如果拇指不用力，按弦的时候是不会稳定的。而且按弦时其余四指对琴颈的压力比拇指大很多，这很不自然。

这个疑惑也从这些历史画面得到了解答。观察上面影片里的巴洛克大提琴和 viol 姿势，你也许会发现在那种姿势下，因为角度比较竖直，胸口没有使劲抵住琴身，如果把琴弦按紧在指板上，指板只从一个方向受力，是没法保持平衡的，所以拇指肯定需要在后面提供支撑。

从 viol 大师 Jordi Savall 的[演奏视频](#)，你会发现他按弦的时候，拇指稳稳地支持着琴颈。琴颈不停地在前后摇动，显然胸口对琴的支持非常少。虽然看起来拇指不是很用力，但这显然不是“一颗草莓”能经得起的力道。当然，琴也不是完全靠拇指支撑平衡的，不然他就没法用左手扶眼镜了；)



再看看视频里另一位 viol 演奏者 (Philippe Pierlot) 的姿势，简直就是抓着琴颈尽兴地在摇。他的琴完全没有靠在胸口上，只用两腿提供了下面的支撑点。有时仍然按着弦的时候，他会忽然把琴整个往上提，因为琴滑到下面去了。所以他的拇指显然大部分时候是用了力的，而且支撑着琴，要是放手琴就会倒。



还有这么小的 viol，拇指还能不在后面用力吗？:)

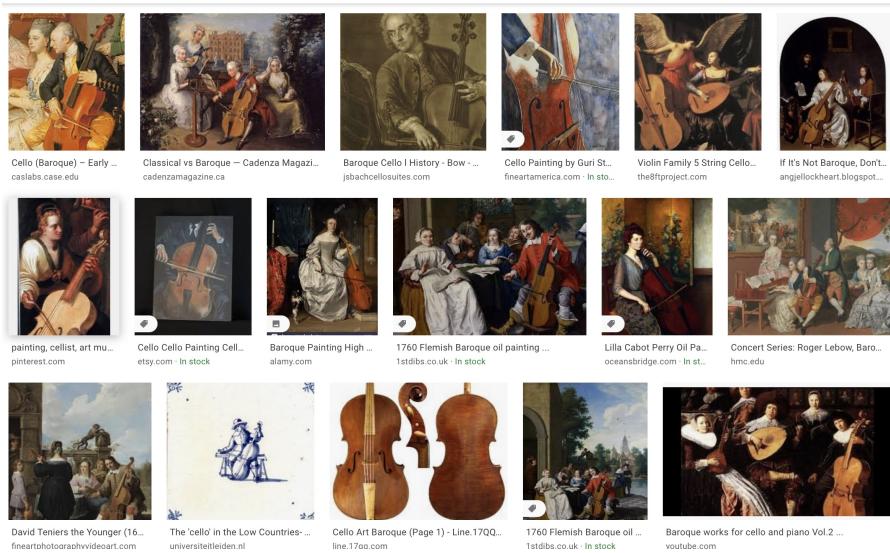


这样的姿势，如何解释它的力学呢？



之前的巴洛克历史画里面的大提琴手，也看到好几个用左手支撑琴颈的。用那种姿势按弦，拇指不可能不捏。你可能会说，画上不是真的音乐家，是模特而已。可是去搜索一下“cello painting baroque”，就会发现几乎都是这样的姿势。如果就一幅画，可能确实是画家美化过，可是这么多幅历史画全都是这样的姿势，不得不使人猜想以前就是这样用大提琴的。





大提琴老师们往往告诉你拇指是大问题，拇指捏琴颈会导致难以实现振音（vibrato，俗称“揉弦”），难换把位。看了巴洛克音乐家们的姿势，我觉得这些说法都是误导。要把弦按下去，最自然最稳定的方式就是拇指在后面挡住，产生反作用力，而且这样和琴一起摇曳着，很舒服很尽兴。至于 vibrato，即使拇指捏住你也应该有办法实现，而且 vibrato 本身并不是可以大量使用的东西。

巴洛克时代的音乐比较少用 vibrato，一般用另一种叫 trill 的方式，也就是用另一根手指在弦上面轻轻摸几下，而不是同一根手指揉弦。近代的大提琴协奏曲（比如 Schumann, Dvorak, Elgar 的 Cello Concerto），动不动就来 vibrato，甚至接二连三都是 vibrato，而且抖动幅度很大，像是蚊子飞来飞去，又像是有人边唱歌边抖腿，让人心神不宁。作曲家莫扎特的父亲 Leopold Mozart 是个小提琴家，他对滥用 vibrato 的做法也是差不多的[看法](#)。他说：“每个音都抖动的演奏者，就像得了麻痹症（palsy）。”

Leopold Mozart's position was similar to Cramer's (and again no distinction between solo and orchestral playing is mentioned):

Now because the tremolo is not purely on one note but sounds undulating, so would it be an error if every note were played with tremolo. Performers there are who tremble consistently on each note as if they had the palsy. The tremolo must only be used at places where nature herself would produce it, [...] on] a closing note or any other sustained note.⁷

这种协奏曲（concerto）和交响曲（symphony）类似，一般又长又无聊，就开头几个音有点新意，但往往缺乏美感，然后掺入大量不知哪里抄来的千篇一律的套路，弄成个大长篇，跟莫扎特的手法如出一辙。经常忽然安静得没有声音，忽然又惊悚一下，害得听众虽然无聊，却又无法睡着。每当有人说这音乐怎么这么长这么无聊，就有人居高临下地说你不懂，这音乐是在讲一个故事，你得细细去品味里面的意境。我也曾经信过他们，以为是自己不懂。现在我明白了，这些音乐确实是故事，就像网络上千篇一律的穿越小说和电视上的肥皂剧，又像是软件行业的“设计模式”（design patterns）。作曲者无法让每一秒钟都是精华，所以想出了这种掺水的手段。

这样的音乐我听都听不下去，就别说演奏了。听了这些近代的大提琴协奏曲，我都开始怀疑学大提琴有没有意义了。实话说，当初要不是因为 Pablo Casals 演奏的巴赫大提琴曲，我会认为大提琴只是在乐团里给人配低音的，根本不会考虑学它。这几个近代大提琴协奏曲演奏得最好的那些大提琴家（比如 Mstislav Rostropovich, Jacqueline du Pre），我听了一下他们演奏的巴赫大提琴曲，那真是没法听，大概是平时节拍器用太多了，毫无表现力，听起来很像我见过的某些小学生拉的。

然而他们演奏这几个协奏曲却是世界顶级的，由此可见这些近代作曲家比起巴赫差距有多大。虽说近代的大提琴协奏曲难以进我的耳朵，Rostropovich 演奏的 Vivaldi 的大提琴协奏曲还是挺好的。Vivaldi 的大提琴协奏曲每一曲都很短，但从 C.P.E Bach 开始，就有点开始单调和缺乏想象力了，而且变得冗长。C.P.E Bach 标志着音乐进入了 Classic 时代，这个时代不管谁作的曲子听起来都差不多的味道。

如果你想了解更多关于 vibrato 的说法，可以参考[这篇论文](#)。我自己用 LinnStrument 演奏巴赫的大提琴曲的时候，开头和中间的音一般是不用 vibrato 的，不过到了一长串音结尾的那一个的时候，有时候会自然想让它轻轻晃一下。那碰巧是多位作者提到的“vibrato 的自然时机”。

下定决心不听也不演奏近代这些大提琴协奏曲，忽然就觉得眼前一片光明，而且姿势也更加自由和舒服。如果不演奏这种有过多 vibrato 的东西，去掉为此而产生的“拇指不能捏”的姿势教条，允许拇指捏琴颈，按弦会更容易和稳定，而且你可以把琴颈像巴洛克式拿法一样放到身体左侧或者其它位置，而不是一定要抵在胸口上。你甚至能看见左手动

作和位置，用左手摇动大提琴。观察影片里 viol 和巴洛克大提琴的演奏者们，他们都把琴颈放在左侧，是可以看见自己左手的，而且按弦就是简单地捏下去。



如果你仔细观察现代的巴洛克大提琴演奏者的视频，也会发现他们不但左手拇指在使劲，而且另外四根手指经常伸直了使劲压弦。因为经常有好几个音要一起压下去，特别是当同一根手指要压住两根弦的时候，这样使劲是很自然的方式，一点问题都没有。所以我的经验是，凡是自己觉得自然的姿势，而别人告诉你不要那样，你都得仔细研究一下他说的到底对不对。

不是每个人都愿意抛弃那种含有大量 vibrato 的音乐，但我们应该明确的是，并不是每个大提琴学生都需要演奏那些音乐。片面因为那种音乐而让学生采用“拇指不准捏”的方式使用大提琴，大大地阻碍了早期的学习进展。如果不用拇指捏，光是靠“手臂自重”，按弦会特别不稳定，所以最初学按弦的时候难度加倍。为了一个不是每个人都想要的目标，使得他们学习难度增加好几倍，这不是合理的教学方式。在我看来，学生应该首先掌握的是找准音在什么地方，练习耳朵的辨音听力，至于最初这个弦要怎么按下去其实不是那么重要。一旦能找准音，这些姿势后来很容易改的，拇指捏一捏并不是什么大不了的事情。

这些巴洛克乐器能够捏琴颈，一个原因也许是因为它们的琴弦张力不大（羊肠弦，415Hz），而且琴弦高度很低，所以要把弦按下去非常轻松。现代的大提琴琴弦都是钢丝做的，440Hz 调音，张力很大，而且为了音量往往把琴桥做得很高，所以弦按起来很吃力，初学者手指都会痛，连大师们经常都得两根手指一起才能把弦按下去。要是拇指用力捏，就会给拇指造成很大的压力，也会影响到 vibrato 和换位的实现。

所以我认为，现代大提琴的各种不合理，急功近利的设计，以及近代缺乏品味的作曲方式和权威崇拜风气，导致了这种“拇指不准捏”的姿势教条。历史画面和其它乐器，往往会给我们新的启示。观察它们，让我看明白了大小提琴领域中的各种误区。各种弦乐器按弦都是可以用拇指捏琴颈提供支撑力的，只有大提琴老师跟你说不要捏，你不觉得是它有问题吗？

我的大提琴教育

本来并没有“正确”的姿势，但随着时间演变，不知道怎么出来那些关于姿势的死板说法。中国这种现象就更严重。我挑选老师已经很小心了，可是仍然难以幸免。上课前的一个星期，我在两个地方试过课，那是我第一次摸到大提琴，拉了几下空弦。第二次试课的时候我再次拉空弦，老师好像惊讶地说：“你真是第一次拉大提琴吗？已经能拉出这么好的声音了。”我说：“这是第二次。”其实当时我心里在想，拉个弦有什么好惊讶的，跟拿刀切肉的动作不是差不多吗。

但这并没有改变我后来的命运。一个月上了四节课，全都是在讲姿势，木偶一样摆来摆去的，相当的精确。第三节课好不容易教了拉空弦，心想下节课该动动左手了吧，结果他来检查空弦。说你下弓到头的时候怎么角度就上去了，没有垂直于弦？我说我看到是垂直于琴弦的啊？他说没有，你的视角看下去是直角，而其实不是，你那个视角要看到钝角，那才是直角。你回家对着镜子看看，拿三角板量一下，就知道是不是垂直的。来，我用手给你比着，你沿着这条直线拉。等等，你的小指怎么又跑到上面去了？我说过要放在圆圈那里的……

这样一节课又过去了，仍然没有动左手。结果回家还真照了镜子，拿三角板量了，之前的姿势其实是垂直的。不管什么视角看过去，我不可能连空间中两条（静止的）直线是否垂直都看不出来吧？连这都会因为视角不同而判断错误的话，我的乒乓球和网球怎么可能打得好？那些都需要判断飞速运动的物体的空间角度关系，可比拉弓难多了。

很多提琴老师没说清楚的事情可能是这样：垂直于弦的方向不止一个，而是有无数个，因为垂直于弦的是一个平面。琴弓需要在这个垂直的平面上，但它不一定要走直线，而是可以在这个平面里选择不同的角度，只要不碰到旁边的弦就行。不同的运弓方向，弓在这个平面里的角度是可以不一样的，甚至可以在中途沿着这个平面调整角度，用以调整压力和摩擦的角度，使琴弦发出最好的声音。



其实整个“垂直”和“走直线”这件事并不是那么的重要。音乐最重要的是声音，不是角度和姿势。很多时候琴弓甚至不需要垂直于琴弦，不同的角度会发出不同的声音，有不同的用处，而且可以利用不同角度拉弓来转移弓与弦的接触点，改变音色。这些我都从各种教学视频学到了。Pablo Casals 的视频里特别明显，他有时候会用很陡的角度拉 C 弦，发出微妙的低音。这我都反复试验过了，那声音就是比垂直拉出来的好。纠结这些角度，非要垂直，本来是垂直的还说你不是垂直的，真是感觉被教傻了似的。



在某种程度上，我觉得 Casals 才是我的老师。我光是旁观他几十年前给别人上课的视频，都比一对一私教课学到得多。他不只是大提琴拉得好，而且很懂音乐的本质。我很喜欢他对学生反复讲的几句话：“不要完全照着乐谱拉，要有表现力！”“乐谱不是那样写的，但它就是那个意思！”语言很权威，语气很可爱。我多想有一个这样老师啊。

不幸的我，一个月的课就是在折腾这种细节，一个练习曲都没有，左手一个音都没有按。第一节课摆身体姿势，第二节课摆握弓姿势，第三节课摆空弦姿势，第四节课还是拉空弦，不断地挑剔角度啊，握弓手指的位置啊之类的。本来一节课就该讲完的东西，硬是拖了一个月。间隔的一个星期时间，如果不是自己看视频学东西，大部分时间无事可做。我自己在家看 YouTube 学的东西，可比老师教的深入多了。我自己用巴赫大提琴曲做练习曲，都学了几个小节了。

后来遇到一个小提琴老师，她告诉我她也是那样教学生的，说国内的大小提琴老师都这样，你非得把手指放对了地方，拉弓必须很直，拉一两个月空弦过了关，才有资格学其它的。而且说他们不大愿意教成人，因为成人往往后来说太忙，就不来上课了。我想我理解这里的原因。具有理解能力和生活经验的成年人，是不容易接受这样教条式的教学的，不愿意在无关紧要的事情上被纠正。

经过这段时间的了解，我还发现国内人学音乐特别重视“节奏”，爱用节拍器，而国外一般只在需要调整的地方才少量使用节拍器。中国小孩姿势和节奏都很“正确”，可就是没有感情和表现力，就像用直尺和圆规作出来的画。这是在制造机器人，不是在教音乐。真正的音乐教育应该让孩子瞬间就爱上这个乐器，而不是只看见吃苦和枯燥。这也许这就是为什么中国一直出不了好的音乐家和艺术家，本来每个小孩都有天赋，却都被这样埋没了。

拉弦的真正原理

你不觉得琴弓很像牙刷吗？只不过它刷的不是牙齿，而是琴弦。不过小朋友要注意了，琴是横着刷的，牙应该是竖着刷。你拿着牙刷的时候，拇指一直是弯着的吗？自然的方式抓握一根杆子，拇指就是要直着才舒服，才拿得稳，但有时候为了让物体到达某个角度，它确实想弯一下，那就让它弯一下。

如果你仔细观察一下，刷牙的复杂程度可比拉琴大多了。拇指有时在这里，有时在别的地方，有时甚至会搭到牙刷上面去。小指也会到处跑，有时在上面，有时候却拐到下面去了。可是没有注意它们的时候，我们全然不知这些事情。你的父母教过你各个手指要放在牙刷的哪个位置吗？他们只告诉你牙刷应该怎么动，告诉你不能横着刷牙，每个角落都要刷到。

用叉子勺子吃饭，有人教过你每根手指要放在哪里，什么角度吗？自己试过之后，自然会找到一个省力又方便的办法。有硬的东西叉不动，那就临时换一个姿势。拿琴弓和拿牙刷，叉子，勺子很不一样吗？你还记得吃饭的时候，拿餐具的拇指什么时候是弯的，什么时候是直的吗？反正我不记得，它好像自己知道怎么办。我得仔细观察自己的拇指，才会明白它是什么状态，它为什么要那样。

类似的，乒乓球拍要怎么握？有好多种握法。有直拍握法，有横拍握法，每一种都有世界级的高手。我国某世界冠军，拍子的握法如此奇怪，还专门为自己的握法定制了球拍，以至于他发来的球很难对付。

我觉得人们都没有理解，这里的关键事情并不是“哪个手指应该放哪里”这样的「方式」。关键的事情应该是“物体的运动”这个「目标」。

对于大提琴，目标其实是这样：拿着琴弓，让它大概在弦的垂直方向走直线，摩擦发出均匀的声音。下弓拉到头的时候，由于琴弓对琴弦力矩变小，会导致声音变小，所以可以稍按一点，也可以就让它自然弱下去。回弓的那一瞬间，如果琴弦振动得厉害，琴弓就不能骤然回头，而是要释放压力，继续往前送出去一点点，否则就会跟琴弦的振动发生冲突，产生杂音。这就是为什么看到拉大提琴的人弓送到头的时候，有时候手腕轻轻抖了一下。这时候手腕不能绷紧了，因为绷紧了琴弓就会骤然停止运动，产生杂音。

这就像打乒乓球，来了一个旋转球，你不能正面就打回去，而是得稍微改变球拍的角度，在接触的一瞬间手腕抖一下，中和球的旋转，才不会把球打飞。更好的做法是不但中和了球的旋转，而且把球朝反方向旋转。比如对方抽来一个上旋球，你就给他抽回去，这就跟拉弦的原理一样了。

如果你仔细观察高速摄像机拍下来的被弓拉动的琴弦的振动，就会发现它不是在平面上来回振动，而是绕着一根中轴在转，形状是很扁的椭球形，像一些旋转球，或者更贴切点，像一根长跳绳。被琴弓拉动的琴弦和被手拨动的琴弦的振动方式是不一样的，前者是椭球形在旋转，后者只在平面上来回。其实你不需要高速摄像机，自己做一些小实验就能验证这个事情。



因为琴弦是被琴弓推动在旋转，所以回弓的时候你需要让琴弦朝反方向旋转，其实就是把球抽回去。如果琴弓突然折返，或者返回时速度不对，琴弦就会像被打飞的乒乓球一样发出杂音。所以我的体会是，大提琴根本就是一项体育运动。拉大提琴有时候像是在抽球，有时候像是在颠球。

有句话说得很好：音乐家就是使用小块肌肉的运动员。

旋转的琴弦这个物理模型，不但可以用来理解回弓的动作，而且可以解释大提琴演奏中出现的许多其它做法。比如，当靠近琴桥（bridge）拉弦的时候，你不能拉很快，而且需要更用力按住。旋转的琴弦模型可以解释这一切。

马友友 7 岁的时候遇到 Pablo Casals，为他演奏了一曲。马友友演奏之后，Casals 说：“你的大提琴拉得很好，但你也应该去打棒球！”你能明白此中的深意吗？因为拉大提琴很像打棒球，旋转的棒球。另外，这类体育运动可以训练人的空间感知能力，提高对乐器的定位和操作能力。这就是为什么很多中国父母逼着小孩周末去学乐器，却不让他们玩耍和参加体育运动，结果到头来什么都做不好，因为缺乏运动的小孩不能自如地控制物体的运动。

如果大提琴老师能告诉学生这些原理，这些动作需要造成的物体运动目标，拉空弦就不再是一件枯燥无味的事情了。就像打乒乓球，每次热身都要来回抽球。踢足球，每次热身都要用各种部位颠球。人的运动系统会依据这些热身运动的反馈进行调整，这样正式运动起来才会流畅。颠足球可以多有趣？我中学的时候经常双脚颠球玩，多的时候可以颠 200 多次不落地。颠球使得运动系统对球的物理性质如此熟悉，以至于传球和射门角度可以随心所欲。

物体的运动才是重点

所有这些「目标」都是关于物体的运动，而不是人的姿势。人所有的姿势都是为了这个目标。知道了目标，经过一些尝试之后，手指自然会拿成需要的形状，每个关节都会按照合适的方向运行。每个人的手臂和手指都是不一样的，长度，角度，关节运动轨迹都不一样。一个人觉得正确的具体位置，对于另一个人就不一定是正确的位置，但为了达成目标，它们会自动调整自己的姿态。这就是人体的神奇之处。

大提琴教学如此，至于学拳，学剑什么的，都有差不多的误区。老师一般都教你摆姿势，却不会告诉你，这些角度啊，位置啊，扭转啊，最终的目的只是把拳头或者剑沿着直线送出去，在击中目标的一瞬间达到最大的速度和强度。学生一开头肯定不会直接做到最佳的姿势，但好的姿势恐怕不是拿尺子量着纠正就能做对的。只有知道了需要达到的目标，自己去调整，才可能达到最好的姿势。

网球和其他运动也不例外。对于网球，本来只需要让学生直接上场先吃点苦头，然后告诉他，球过来的惯性很大，所以你得提前甩动球拍，利用惯性把球打回去。然后你的握拍手势，各种发力角度和时机都会自然调整。可是一般网球老师却只是死板地传授身体的姿态，而忽略了解释球和球拍的运动规律。

如此僵化的教学，就是为什么网球天才王振在清华上了一节网球课之后，就被埋没了。我从小在电视上看网球比赛，看见阿加西之类的网球选手流畅的动作，虽然自己没有条件打网球，心里却在想象，要是有一天我有了网球拍，附近有了网球场，我就这样打球，要是球那样飞过来，我就这样抽回去……结果十几年以后第一次上场，就能成功地使用专业的过顶姿势发球，第二次就能反手超低角度抽杀。同伴很吃惊，就推荐我去上网球课。结果第一节课各种摆姿势，就像被教傻了一样，真不想去上课。后来因为遇不到风格合适的对手，就荒废了。

网球教练 Timothy Gallwey 写的《The Inner Game of Tennis》讲到，网球的教学不能通过给学生一百个细节的指令，批评和纠正他的动作来完成。最好的做法其实很简单，那就是反反复复给学生示范。学生通过观察，他的潜意识会模仿老师的动作，不知不觉在头脑中练习，最后自己就会了。这就是所谓“用脑子练球”。我从小就用脑子练球，我的“教练”就是阿加西。



Timothy Gallwey 的这个说法我很早以前就听说过，所以我经常会发现它的妙用。然而很可惜，世界上大部分的老师都没有明白这个道理。很有趣的是，你到 YouTube 看看 Pablo Casals 当年的 masterclass 视频，会发现他就是通过给学生示范，让学生模仿来完成教学的。Casals 是一个很好的老师。

画画的姿势也一样。某些老师会跟你说铅笔一定要这样拿，手臂一定要是这个角度去画线，然后就学傻了，画出一些缺乏品味的东西，到处都是生硬的线条。

电脑键盘打字也有类似现象。很多人坚持 backspace 键属于右手小指的“分工区域”，所以总是伸长小指去按那个键。而我发现无名指比小指长很多，是最方便够着那个键的手指，为什么不用它呢？所以我一直都用无名指去按 backspace，小指就轻松多了。

所以对于各种活动的姿势，我觉得人们的教学都是误区。理解「物体的运动」这个目标，而不是死扣人的具体姿势，应该才是这类动作教学的关键。人的运动系统有着非常先进的「目标驱动」机制，只要指定了目标，运动系统通过多次试验，自动调整自己，最后达到目标。有意识地指定细节反而会僵掉了，甚至受伤。

音乐与乐器设计

刚开始学大提琴一个月就发现了这些，我是幸运的。我不会再像其他人一样傻傻地努力，一丝不苟地跟老师学习，拿自己的身体跟这个乐器拼命。可能很多人都没有明白，「音乐」和「乐器」其实是两回事。真懂音乐的人，他的乐器不一定需要演奏那么好，他知道各种乐器的设计缺点，所以不会跟它们拼命。乐器演奏得特别好的人，其实不一定真懂音乐。

在视频上看见某些专业大提琴演奏者的左手，我都觉得惨不忍睹。不仅都是茧，很多茧子还是破裂的，像炸成两半的爆竹一样。还有一些人因为使用“德国式指法”，硬要把自然挨在一起的中指和无名指分开，同时放在两个音上面，导致手指关节都变形了。

难道这就叫做“为音乐献身”？就算再喜欢音乐，我的手也不应该为这些所谓“专业演奏级”钢丝琴弦，为“大音量”设计的琴弦高度，被磨损成那个样子。要真正的享受音乐，我必须突破这些教条，抛弃一切不符合人体自然构造，造成畸形的指法设计。毕竟我的目的是享受，而不是像很多学音乐的孩子和家长一样，是为了“出人头地”。我会把它变得容易一些，舒服一些，成为它的主人，它的朋友，而不是仆人。

如果有某些音乐导致我必须使用畸形的动作，我就抛弃那些音乐，或者改造乐器。我发现很多乐器中存在的难度并不是本质性的。大提琴，钢琴，比起它们的巴洛克时代的前辈们，在设计上不一定更加先进，反而可能是倒退的。比如，巴赫的 Goldberg Variations 本来是为两层键盘的 harpsichord 设计的，是写给当时的音乐爱好者做练习曲，自娱自乐用的。如果你没见过，harpsichord 是这样子的：



现代人以为钢琴比 harpsichord 进步，就拿钢琴弹这些曲子，结果难度大了许多。不仅因为钢琴键太宽，而且钢琴没有双层键盘，所以经常出现两手像出了车祸一样交叉飞过，接下来还要回头交换保险信息。因为用钢琴弹这个太难了，就把巴赫的练习曲膜拜为大作，会弹这个的人都被膜拜为天才。

大部分人都以难为荣，用着一些不合理的工具和姿势，朝着“顶峰”奋勇前进。有些人虽然自己不能弹 Goldberg Variations，也要膜拜一下 Glenn Gould。真正聪明的音乐爱好者，为了享受巴洛克时代的音乐，已经去旧货市场买了 harpsichord，在家把 Goldberg Variations 弹得舒舒服服。

巴赫的大提琴曲最初也不是为现代大提琴作的，而是某种有[五根琴弦的乐器](#)，多了一根高音弦，它甚至不一定是大提琴。某些曲目（比如巴赫第 6 套大提琴组曲）用四根弦的现代大提琴拉，就困难了很多，高音老往“拇指位置”上跑，但会拉的人往往因此引以为豪，其他人也觉得他们是天才。



很多大提琴家可能没想过，这些又难又辛苦的“拇指位置”（thumb position），其实只要多加一根琴弦，大部分都没必要了，可以舒舒服服按正常姿势来。

这也许就是为什么巴洛克时代的大提琴指板没有很长，因为与其延长指板，还不如加一根琴弦。实际上在巴赫的年代，演奏大提琴都是不用拇指的，巴赫写音乐的时候从来就没想过后人会傻到开始用拇指，而不是想法改造他们的乐器。

根据一份来自日本的[五弦大提琴研究报告](#)的说法，大提琴家们不使用五根弦的大提琴的原因至今是一个谜。他们好像宁愿用四根弦的琴折腾，可是很少有人成功演奏巴赫第 6 套大提琴组曲。猜想的一个原因，也许是因为如果有五根弦，琴身振动不会有四弦大提琴这么厉害，所以音量会小一些。嗯，音量就是这么重要。

只有巴赫在背后偷着乐：我只管写，你们费力去演奏，管你用什么乐器，最后人们仍然只记得我的名字！不管是作曲还是乐器设计，巴洛克时代的智慧似乎都失传了。可是我们仍然看见各种影片在片面美化各种乐器（特别是钢琴），把它们等同于音乐本身，把它们描述成伟大不可改变的事物，值得人为此忍受各种不便，甚至献出健康和生命。可是乐器并不是音乐，我欣赏音乐，可是我并不觉得需要对任何乐器那么“忠诚”。

计算机科学阅读班（实验）

昨天发出计算机科学基础班（第三期）的信息，已经收到了好些同学的踊跃报名。请大家写申请不要有太大压力，我会在春节之后的一个星期才开始阅读这些申请。请注意我不是按照发送的时间顺序，而是综合考虑的，所以请大家写申请不要有时间压力，好好享受春节的最后几天。

大家的踊跃报名让我感觉欣慰，因为我长期以来改善教育的希望得到了很多的理解和支持。同时也让我有些不安，因为亲自授课付出很多精力，以至于课程的容量很有限，而且价格给很多年轻人造成一定压力。每次集体班都会不得已而拒绝很多人的申请，以至于很多有志于学习的人们得不到帮助。经过「集体班」和「一对一微信教学」的实验，我发现这样的教学效果确实好，所以这段时间我一直在设想一种新的授课方式，可以逐步改善这种情况。

这个新的授课方式就是通过阅读来进行。内容很像一对一的对话教学方式，只不过讲课不是通过实时的对话，而是通过分阶段的阅读来进行。我会根据以往的一对一对话教学积累的经验撰写一本书，这本书将来会正式出版。为了确保这本书真的能看懂，这个阅读班的同学会成为第一批“小白鼠”。

我会把书的内容分批写出来，不断地改进，内容是对话的形式。在我写书的同时，参加阅读班的同学会分多次拿到写出来的内容，进行阅读，然后我会给他们和普通班一样的练习。练习做出来之后，按照跟普通班一样的方式发给我，进行一对一的指点。基础的练习可能会采用助教辅助的方式，难度较高的练习由我和助教一起来指点。助教会从以往参加过教学的同学里面选择表现很好又热心的同学。

这样的方式会减轻我的教学压力，而且根据到目前为止的经验，我预计效果应该也很好。最终我会写出一本真正能让大部分人都能看懂的计算机入门书，帮助改善全世界的计算机教学。

这本书除了内容方面不同，还有一个其它书籍没有的特点，那就是它是可以扩展的。将来我可以根据这个框架设计更高级的练习，可能会收取新的费用之后作为“升级”。这种“升级”方式可能在将来达到很高的水平，囊括计算机领域最混乱，最难的一些领域，甚至扩展到计算机以外的领域。但因为思维是一套系统，恐怕只有通过了基础学习的同学才会很好地吸收这些内容，所以这类升级内容并不提供给没有参加过我的基础教学的人员。

虽然我考虑过设计 app 来进行教学，但 app 并不是永恒的。最近发现以前有人用 flash 做过一些我想看的教学动画，结果因为 flash 不再被支持，我无法看那些内容。iPhone app 也就是最近十几年的事情而已，而且系统升级之后 app 就不一定还能用了。如果想要把知识传承三百年以上，恐怕不能考虑用 app。

书籍（特别是纸质书籍）仍然是人类最宝贵，最可靠的财富，因为它使得人们可以跨越时空的交流，不依赖于任何技术，甚至不依赖于电，永远不会消失。但前提是书籍的作者必须真的用心写了，真的能通过它传递思想。不好意思地说，我发现现代大部分书籍都是沽名钓誉的工具而已，有的压根没想把事情说清楚，或者只是罗列一堆知识点。只有经过真正教学实验的书籍，才可能真的被人看懂。

我曾经开始写过一本《解谜计算机科学》。虽然我以为写出了“精髓”，但那个时候因为没有真正的教学反馈，内容很多时候其实是初学者看不明白的。通过一年以来的教学，我发现了这些问题，并且理解了为什么很多领域的教材是看不懂的。这一次的阅读教学，应该会真的写出一本这样的计算机科学入门书来，传承真正的知识，造福子孙后代。

为了专注于核心内容，避免将来的翻译开销，并且为了广泛的传播，这本书首先会用英语写作。所以阅读班的同学应该具有基本的英语能力。我会使用很容易的英语，不含复杂的语法和单词。这也锻炼参加者的英语能力。

书正式出版之后，我会把它翻译成中文或其它语言。练习和指导会仍然通过中文进行，但其中的一些练习可能会逐渐被翻译成英语，作为书籍的内容或者附件出版。

我觉得现在可以同时开始进行这个阅读班的实验。具体的方式如下：

1. 内容和集体班一样，只是通过阅读，而不是视频授课进行。比起集体班，有一点好处就是学习时间是更自由的，可以根据自己的步调进行阅读。
2. 阅读的时候如果有看不明白的地方，可以随时提问。我会在工作时间内，根据问题进行实时指导，并且根据问题的所在更新书籍的内容，避免再次出现看不懂的情况。
3. 阅读班的内容范围，请查看集体班的内容说明。
4. 因为没有了实时的视频教学负担，学费比集体班要低，但因为仍然有大量的练习指导需要花费很多精力，所以学费也不会过度便宜，暂定为 10000 元每人。
5. 阅读班因为没有班级授课，所以随时可以开始，可以跨时区进行。国外参加者可以用 PayPal 美元支付学费。
6. 因为内容需要临时撰写，所以在每个文档之间会有一个星期左右的间隔。在这个间隙时间，学生可以进行阅读，提问，做练习。
7. 为了避免造成懈怠和拖沓情况，学生应该在三个月之内完成全部学习。在这个时间之内，请保证能用心付出大量的努力。这个期限和以前的一对一教学一样，应该能满足需求。
8. 为了保护阅读材料的著作权，需要参加阅读班的同学签署一个版权保护协议。
9. 阅读班的内容由于还没有写，所以不能提供给第三期集体班的同学作为参考。
10. 我会根据实际的实验情况更新授课的方式。

报名方式

报名请仍然通过 email 发送申请到 yinwang.advising@icloud.com，标题《计算机科学阅读班报名》。信件内容和集体班一样，写明自己的基本信息，学习动机，附带自己的简历。阅读班同样会经过面试。由于具有大量的亲自教学，参加的同学应该具有良好的品质，所以请慎重书写申请。

已经报名参加集体班第三期的同学，在班级确定之前可以申请换成阅读班，但是集体班开课之后就不能再换，所以请想清楚自己的需求和情况再决定。

计算机科学基础班（第三期）报名

（第三期课程已经报名结束，并且开始教学。有意报名第四期的同学可以发送申请，我可以提前进行面试工作，不过需要等两个月以后才能开始了。也可以考虑最近开始研制的[阅读班](#)。）

计算机科学基础班（第二期）已经成功结束两个月了。每次的教学都让我发现以前没有注意到的细节，以至于每一次都在改进。现在休息了两个月之后，我觉得大概可以召集第三期的报名工作了。

第二期课程总结

第二期课程调低了学费，增加了课程规模。虽然减轻了学生的经济负担，让更多的人能够参加，总体的效果也很好，但这使得我和助教的工作都比较辛苦。

我的教学跟普通学校有很大的不同，不仅在于内容，讲课方式，还在于精心设计的练习。练习是教学中很重要的部分，学生自身能力的提高，其实主要是由循序渐进的练习来完成的。就像设计良好的健身练习一样，它们会逐渐让学生的头脑变得强壮，而不是让他们半途放弃或者受伤。

每一个练习都是单独提交，而不是都做完才一次性提交。这样学生会得到准确而及时的反馈，避免重复犯错误。对于思路不清而卡住的练习，也会收到量身定制的准确提示，让学生意识到错误所在，思维走上正路，却不“剧透”最终结果。这样的设计，使得学生的思维得到最大限度的锻炼，逐渐获得独立思考的能力。

对于学生的练习，我的要求不仅是要正确，而且要极度简单，没有任何多余的东西，逻辑严密清晰，就算空白和排版都要符合最高的标准。每一个练习都可能被要求多次修改之后才能通过。从最小的程序出发，学生从一开头就养成逻辑严密的习惯，直到复杂的代码都保持这种习惯，所以程序极少出现错误。从这个课程，学生会自然地掌握我在《编程的智慧》一文里指出的各种方面。几乎没有任何大学会纠正和引导学生这方面的风格，而这其实是很重要的。这就是为什么这么多人博士毕业了，却做出那么复杂而容易出错的设计来。

我为每个学生都建立了一个“辅导群”，里面有四个人：学生，老师和两个助教。这样的设计使得学生能最大限度的得到反馈。即使老师在忙其他事，助教有时间也可以回复。助教不清楚的地方，都由老师亲自来看。所以虽然是集体班，但其实每个人都得到了近似于一对一的教学。

相比大学里的情况，教授和助教是只有上课时或 office hours 才见得到的。作业都是一次性提交批改，不可能来回的提示和指引。这种特别的教学方式，使得对时间的利用效率大大提高。课后的每个练习本身也都变成了教学，所以虽然只有 8 节课，实际的教学时间却是大大高于讲课时间的。大部分时间是学生在用功，老师只需要在关键时刻点一下就行。

做这样的练习是如此有趣，以至于到后面几节课时，学生们都希望我少讲，少“剧透”，从而可以把更多的知识点作为练习给他们自己思考，以至于中间有一两节课几乎没讲什么内容，大部分作为练习发放了。这样的教学方式和效果是我的生涯中前所未见的，可能是世界上独一无二的。我在中国和美国待过四个大学，没有任何一门课程，任何书籍像我讲的这么少，学生却学到这么多。

虽然效果很好，练习的回复也增加了老师和助教的工作。第二期课程中的许多天，我和助教们直到晚上 12 点都还在回复学生的练习，给他们提示和指点。由于很多学生平时要工作，所以直到周末才开始做练习，以至于周末的时候涌来大量的练习信息。我有两个非常认真热心的助教，有时候看到他们深夜和周末还在忙着回复学生，我都叫他们快去休息。助教的反映是，虽然微信聊天都设置为了“免打扰”，学生也知道晚上不期待我们回复，但看到学生提交的练习，总是忍不住会去看，去回复。我可以说，在当今的世界，可能很少有人会这么在乎教学这个事了。

第三期课程报名

因为第二期的学费和学生数量，使得教学进行比较累，所以第三期课程不会再采用第二期的学费价格和班级大小。第三期课程的具体计划如下：

1. 学费调整回原先的 12000 人民币，可以接受相应的美元付款。
2. 班级大小限制为 15 人以内。
3. 讲课课时仍然是 8 节课，每节课大概 2 小时。但因为我发现其中有一两节课内容很少，时间其实没有很好利用，所以也可能把其中的一两节课换成练习发布。
4. 因为讨论会效果不是很明显，而且可能占用学生的工作时间，不再设置每星期一次的讨论会。
5. 由于第二期的两位助教付出了很多辛苦，所以这次课程不再让他们做助教。这次可能没有助教，不过如果有以前表现出色的学生自愿做助教，也可以考虑。
6. 上课时间一般会定在周中某一天晚上 8 点到 10 点。因为涉及到比较多人，上课时间一旦定下之后就不再因为个人的工作时间变动而改动。课程不提供录播，所以加班较多的学生请考虑好自己的工作时间，错过一节课的代价可能会很大。
7. 课程会在春节之后，经过对申请进行筛选，班级容量达到之后开始。

报名方式

报名请发送 email 到 yinwang.advising@icloud.com (请勿向这个 email 发送其它主题的信件)。标题为《计算机科学基础班 (第三期) 报名》。来信请说明自己的基本信息，附件发送一个简历。由于班级人数有限，而且为了课程的顺利，会对申请人进行选择。像申请国外大学一样，请写一段“personal statement”说明自己的学习动机。因为你的态度决定了是否提供课程，所以对于申请请慎重，不要着急和轻率，学生的选择不是按照申请的时间顺序的。如果觉得合适，我会通知你进行下一步的面试。

课程大纲

根据第二期课程的经验，我想对课程的内容做一个比以前详细的说明。之前一直对课程内容没有很多说明，一方面的是留下自由发挥的空间，一方面是让学生有一定的神秘感，引发好奇心。但这么简单的说明似乎会让不知情的人误以为“已经学过这些东西”，有时候会发现一些人看了说明之后，自以为我教的内容他都会了。我只为他们感到可惜。

下面简要说一下课程的内容：

教学语言。课程目前使用 JavaScript 作为教学语言，但并不是教 JavaScript 语言本身，不会使用 JavaScript 特有的任何功能。课程教的思想不依赖于 JavaScript 的任何特性，它可以应用于任何语言，课程可以在任何时候换成任何语言。学生从零开始，学会的是计算机科学最核心的思想，从无到有创造出各种重要的概念，直到最后实现出自己的编程语言和类型系统。

课程强度。课程的设计是一个逐渐加大难度，比较辛苦，却很安全的山路，它通往很高的山峰。要参加课程，请做好付出努力的准备。在两个月的时间里，你每天需要至少一个小时来做练习，有的练习需要好几个小时才能做对。跟其他的计算机教学不同，学生不会因为缺少基础而放弃，不会误入歧途，也不会掉进陷阱出不来。学生需要付出很多的时间和努力，但没有努力是白费的。

曾经有一两个学生因为低估了学习的强度，同时又有其他重要任务，结果发现忙不过来，所以请合理的安排，不要在有其他重要任务的同时参加学习。

第一课：函数。跟一般课程不同，我不从所谓“Hello World”程序开始，也不会叫学生做一些好像有趣而其实无聊的小游戏。一开头我就讲最核心的内容：函数。关于函数只有很少几个知识点，但它们却是一切的核心。只知道很少的知识点的时候，对它们进行反复的练习，让头脑能够自如地对它们进行思考和变换，这是教学的要点。我为每个知识点设计了恰当的练习。

第一课的练习每个都很小，只需要一两行代码，却蕴含了深刻的原理。练习逐渐加大难度，直至超过博士课程的水平。我把术语都改头换面，要求学生不上网搜索相关内容，为的是他们的思维不受任何已有信息的干扰，独立做出这些练习。练习自成系统，一环扣一环。后面的练习需要从前面的练习获得的灵感，却不需要其它基础。有趣的是，经过正确的引导，好些学生把最难的练习都做出来了，完全零基础的学生也能做出绝大部分，这是我在我世界名校的学生里都没有看到过的。具体的内容因为不剧透的原因，我就不继续说了。

第二课：递归。递归可以说是计算机科学（或数学）最重要的概念。我从最简单的递归函数开始，引导理解递归的本质，掌握对递归进行系统化思考的思路。递归是一个很多人自以为理解了的概念，而其实很多人都被错误的教学方式误导了。很多人提到递归，只能想起“汉诺塔”或者“八皇后”问题，却不能拿来解决实际问题。很多编程书籍片面强调递归的“缺点”，教学生如何“消除递归”，却看不到问题的真正所在——某些语言（比如 C 语言）早期的函数调用实现是错误而效率低下的，以至于学生被教导要避免递归。由于对于递归从来没有掌握清晰的思路，在将来的工作中一旦遇到复杂点的递归函数就觉得深不可测。

第三课：链表。从零开始，学生不依赖于任何语言的特性，实现最基本的数据结构。第一个数据结构就是链表，学生会在练习中实现许多操作链表的函数。这些函数经过了精心挑选安排，很多是函数式编程语言的基本函数，但通过独立把它们写出来，学生掌握的是递归的系统化思路。这使得他们能自如地对这类数据结构进行思考，解决新的递归问题。

与一般的数据结构课程不同，这个课程实现的大部分都是「函数式数据结构」，它们具有一些特别的，有用的性质。因为它们逻辑结构清晰，比起普通数据结构书籍会更容易理解。与 Haskell 社区的教学方式不同，我不会宗教式的强调纯函数的优点，而是客观地让学生领会到其中的优点，并且发现它们的弱点。学会了这些结构，在将来也容易推广到非函数式的结构，把两种看似不同的风格有机地结合在一起。

第四课：树结构。从链表逐渐推广出更复杂的数据结构——树。在后来的内容中，会常常用到这种结构。树可能是计算机科学中最常用，最重要的数据结构了，所以理解树的各种操作是很重要的。我们的树也都是纯函数式的。

第五课：计算器。在熟悉了树的基本操作之后，实现一个比较高级的计算器，它可以计算任意嵌套的算术表达式。算术表达式是一种“语法树”，从这个练习学生会理解“表达式是一棵树”这样的原理。

第六课：查找结构。理解如何实现 key-value 查找结构，并且亲手实现两种重要的查找数据结构。我们的查找结构也都是函数式数据结构。这些结构会在后来的解释器里派上大的用场，对它们的理解会巩固加深。

第七课：解释器。利用之前打好的基础，亲手实现计算机科学中最重要，也是通常认为最难理解的概念——解释器。解释器是理解各种计算机科学概念的关键，比如编程语言，操作系统，数据库，网络协议，Web 框架。计算机最核心的部件 CPU 其实就是一个解释器，所以解释器的认识能帮助你理解「计算机体系构架」，也就是计算机的“硬件”。

你会发现这种硬件其实和软件差别不是很大。你可以认为解释器就是「计算」本身，所以它非常值得研究。对解释器的深入理解，也能帮助理解很多其它学科，比如自然语言，逻辑学。

第八课：类型系统。在解释器的基础上，学生会理解并实现一个相当高级的类型系统（type system）和类型检查器（typechecker）。这相当于实现一个类似 Java 的静态类型语言，但比 Java 在某些方面还要高级和灵活。我们的类型系统包含了对于类型最关键的要素，而不只是照本宣科地讲解某一种类型系统。当你对现有的语言里的类型系统不满意的时候，这些思路可以帮助你设计出自己的类型系统。学生会用动手的方式去理解静态类型系统的原理，其中的规则，却不含有任何公式。

类型的规则和实现，一般只会在博士级别的研究中才会出现，可以写成一本厚书（比如 TAPL 那样的），其中有各种神秘的逻辑公式。而我的学生从零开始，一节课就可以掌握这门技术的关键部分，实现出正确的类型系统，并且推导出正确的公式。有些类型规则是如此的微妙，以至于微软这么大的公司在 21 世纪做一个新的语言

(TypeScript)，仍然会在初期犯下类型专家们早已熟知的基本错误。上过这个课程的很多同学，可以说对这些基础原理的理解已经超过了 TypeScript 的设计者，但由于接受的方式如此自然，他们有一些人还没有意识到自己的强大。

关于面向对象。虽然课程不会专门讲“面向对象”的思想，但面向对象思想的本质（去掉糟粕）会从一开始就融入到练习里。上过课的同学到后来发现，虽然我从来没直接教过面向对象，而其实他们已经理解了面向对象的本质是什么。在将来的实践中，他们可以用这个思路去看破面向对象思想的本质，并且合理地应用它。

奖励练习。途中我会通过“奖励练习”的方式补充其它内容。比如第二期的课程途中，我临时设计了一个 parser 的练习，做完了其它练习的同学通过这个练习，理解了 parser 的原理，写出了一个简单但逻辑严密的 parser。奖励练习之所以叫“奖励”，因为并不是所有学生都能得到这个练习，只有那些付出了努力，在其他练习中做到融会贯通，学有余力的学生才会给这个练习。这样会鼓励学生更加努力地学习。

如果理解了以上内容蕴含了什么，你可能就不会再问“这些我都学过了，我可不可以参加高级班”了，因为极少有人真的理解了以上内容。就算世界上最高级职位的一些程序员，大学里的教授，对于这些也有很多含糊不清的地方。我自己也通过讲授这些内容得到了启发。

一个朋友看了我的课程内容说，这不叫“基础班”，只能叫“大师班”。他不相信零基础的学生能跟上，但事实却是可行的。为什么不能即是“基础班”又是“大师班”呢？有句话说得好，大师只不过是把基础的东西理解得很透彻的人而已。我希望这个基础班能帮助人们获得本质的原理，帮助他们看透很多其它内容。所以上了“基础班”，可能在很长时间之内都不需要“高级班”了，因为他们已经获得了很强的自学能力，能够自己去探索未知的世界，攀登更高的山峰。

计算机科学集体班（第二期）报名

从四月开始进行计算机科学基础教学，到现在已经快半年了。不论是集体班还是一对一教学，都取得了让人欣慰的效果。挺多人学到了真实的知识，我自己也更加清晰地理解了所教的内容和教学方法。经过反复的迭代，思考和改进，教学的步骤和方式已经趋于成熟。

通过自己最近对一些新领域的学习，我深入地体会到了中国教育长期存在的问题。我体会到的现实是，中国在几乎任何领域，都没有掌握切实的，深入的知识。从国外学来的东西，几乎都是不彻底，不地道。不求甚解，以讹传讹的思维方式盛行。所以在中国，在几乎任何领域，都很难找到好的老师。我自己感兴趣的几个领域，目前都只能靠自己去琢磨，或者直接从国外资源学习。

我难以看见这么大的一个国家如此对待知识，继续落后下去。是改变的时候了。

一对一的问答教学是成功的，它让我对人的认知规律有了一些理解。但由于一对一教学具有大量的重复，时间久了对于老师来说相当枯燥，虽然每个学生都觉得很新颖。最开头有利于改进教学方式，但前一天才讲了的东西，第二天就给另一个人讲，并不是有趣的事情。所以我决定限制一对一教学的数量，提高了价格，并且对申请人更有选择性。

在将来的一段时间里，我会投入更多精力在集体教学上。集体教学对于个人，虽然没有一对一那么灵活，但由于教学方式和内容安排的先进性，效果还是大大高于传统大学计算机科学专业的教学。

我不想在这里详细说明我的教学哪里先进，因为我发现有盗用“关键词”的人，拿过去就成为自己的主打广告词，却做不到实质。我提出的“问答式教学”这个做法，显然已经有了一些“山寨品”。似乎再好的理念到了中国，都会被很多人抄袭，成为空洞的口号。另一方面，过多宣传会引来很多不合适的人，所以我不想对自己的做法进行宣传。

现在我决定召集第二期的计算机科学基础班，但具体的很多事宜和特点，我不想在这里说，而只说明对于报名最关键的几点：

1. 学费调整为 6000 元每人。
2. 方式为 Zoom 音视频教学。
3. 时间跨度还是两个月左右（法定节假日除外）。国庆节之后开始上课。
4. 一周一次课，总共还是 8 次课。每周安排一次讨论会，方便大家交流。
5. 课后练习仍然是一对一回复指点，但可能会安排有经验的助教帮忙。为了确保质量，助教拿不准的地方都会由老师直接指导。
6. 内容包含了计算机科学最根源，最本质的一些思想，它们可以作为坚实的基础，用以理解和掌握更广泛的知识。
7. 由于时区问题，可能无法照顾时区相差太多的国外报名者。
8. 人数限制会放宽一些。由于集体教学比较少有人提问，所以不会是一问一答的方式，但可能有积极的同学参与问答，目的在于启发大家。
9. “基础”并不等于“容易”或者“初级”，课程的内容虽然从零开始，最后却会很深入。这不是一般程序员知道的东西，所以不再要求申请者完全是初学者。但有基础的人士需要理解，这个教学仍然是从零开始，需要重新认识最基本的知识。

人员要求

心理健康，有礼貌，能尊重老师和其他同学。谦虚好学，能平等讨论。

申请方式

发送 email。标题：申请计算机科学基础课程（第二期）。内容包括：

1. 你的真名，简单自我介绍，包括教育经历，工作单位，工作内容，上课的动机等。
2. 你所在的地区，用于安排时间。
3. 附上简历（PDF 格式）。
4. 你的微信号，方便联系。

由于申请人数可能较多，请耐心等待回复。我会安排简单的电话面试。

一对一教学学生可以免费听课

已经参加一对一教学，还没有结束，也没有超过时间跨度的学生，可以免费旁听集体课程，作为一种补充或者复习。需要者可以给我联系，到时我会告知课程的 Zoom 会议号码。

知识星球成员优惠

由于之前建立知识星球的时候，没有考虑到提问的方式（而不是教学）很难真的帮助到人们，自我感觉知识星球的效果并不好。好的方式应该是老师提问，而不是学生提问。没有老师的输入和指引，学生会很难提出好的问题。提问越

越来越少，越来越艰深，所以来知识星球就不再接受新的成员。有些人的提问进入一些误区，却因为理解那些需要补充比较多基础，而无法帮助他们。有时候看到问题进入自己探索过的误区，但因为别人正热心，也不好意思扫兴。由于教学的效果好很多，我已经挺长时间没有在知识星球活跃。

为了回馈知识星球的成员，感谢他们的支持，知识星球成员如果申请参加计算机科学基础班，会给予 2000 元的减免。这相当于退回我实际的知识星球收入（除去平台服务费）。也就是说知识星球成员报名基础班，只需要 4000 元学费。来信请注明在知识星球里的用户名。

几个需要避免的美国英语习惯

这篇文章汇集了最近我在微博提到的一些美国英语的“病毒”。

加州英语的语调问题

为什么我推荐机器学习入门，可以看 cs231n 2016 年冬季的[视频](#)，而不是同一门课的新版本呢？主要原因当然是 Andrej Karpathy 讲得比其它两个美国小孩好。很明显，换了另外两个人之后，发现很多地方听不懂。感觉他们只是拿着 Karpathy 写的 ppt 照本宣科，而不得要领。感觉他们在给别人讲，而自己心里都在发抖。

另一个我没说的事情，其实是我很受不了那两个美国小孩的英语，其中一个是白人，一个是 ABC。很典型的加州音，每个句子都以升调结尾，中间也是一路上飘语调，一直降不下来。像是一直在提问，征求意见，而其实是一个陈述句。

这样的语调在加州很常见，似乎他们想显得可爱，以为一路升调，可以让别人更喜欢他们。可是这种语调会让听者的心一直悬着，不知道什么时候结束。而且太刻意了，听久了就觉得特别假，很不自然。加州人不管男的女的，很多人这样。

看了 Karpathy 的视频，很明显发现他不是那样的，陈述句的结尾很明显就降下来了，给人一种踏实的感觉。他不是加州人，他恐怕就不是美国人。注意我不是说我喜欢伦敦音。看 Andrej 这个名字，是东欧某个靠近俄罗斯的国家来的。

我在 IU 的时候，好几个教授都是欧洲小国来的，他们说英语明显清晰很多，不是美国音，也不是伦敦音。Dan Friedman, Kent Dybvig, ... 都没有明显的美国音。我的“正式导师”Amr Sabry 是从埃及来的，当然也不会美国音。

应该去掉的语气词

加州人的英语还有一个问题，那就是他们会附带很多没用的“语气词”。我最讨厌的语气词是 “like”。听加州人说英语，你会经常听见“..... like ,”，句子说到一半，接一个 like。意思不是“喜欢”，而是“看起来”。比如：“He is like, ...”本来要说他怎么样，结果加一个 like，接着一个很长的停顿。甚至 like 后面就没有下文了，尽在不言中。还有“It's like...”，本来要说一件事，非得在前面加上一个“It's like...”。现在每次听到句子里夹 like 的人就很受不了。

还有一个很讨厌的语气词，是“you know”。句子说到一半，加一个“you know...”（你知道的.....），然后接着说。有时候一个句子里面可以加进七八个 you know，真叫人着急。我都 know 了你还说什么呢？而且我真的 NOT know 你要说什么。毫无意义，自相矛盾，严重影响语言表达，就像口吃一样。

另外几个加州人喜欢用的词：cool，awesome。开头他们都喜欢说 cool，这是一句毫无意义的套话。不管一个事情喜不喜欢，听到马上说 cool！很兴奋的样子，刻意让你对他产生“好感”。后来发现 cool 用得太多了，套路被识破了，有些人就开始说 awesome。不管听别人说什么，管自己心里怎么认为呢，先来一个 awesome！逐渐的，awesome 也失去了意义。

我不知道他们下一个选择是什么。总之，他们总有一句口头禅，用于掩饰他们对你说的内容的不理解，不知所云，假装很喜欢。

看 Karpathy 的视频，你会发现他没有任何这样的语气词。如果一句话说到一半想不起来，他会停顿一下，没有任何声音，没有 you know, like, 甚至没有“嗯”，想起来了就继续说，所以就感觉很清晰。

语言的垃圾真是很有传染性。听到身边朋友这样说话，不知不觉就学了。一般的心理是：这样听起来更像美国人，所以别人才看得起我。内心的自卑，导致了在美华人小孩很容易学会这些不好的东西。真希望住在加州的中国人能别让孩子学会这些。

So...

既 ‘like’，‘you know’，‘cool’，‘awesome’ 之后，我又想起一个大家不应该学习的美国口头禅，那就是 “So, ...”

一般说来，so 应该用在一个句子中间，前面应该有一个分句，然后接着一个 so 开头的分句，前面的东西表示 so 分句的条件或者起因。比如，“..., so we can...”（.....，这样我们就能.....）你必须有一个起因，然后你才能说 so...

可是很多人喜欢在一句话开头说“So, ...”前面的句子也没有相关的起因，不知道这个 So 接着什么在说。而且说出这个 So 之后，有时会有很长的停顿。往椅子上一靠，So... 意味深长的样子。有时候一个 So 还不够，接着又来几个 so。你就听到 so so so... 个不停。

我发现这个现象已经传染到中国人，而且翻译成了中文。有些人喜欢以“那么.....”开始一个句子，而“那么”前面的内容却没有任何相关内容，就是这种“英语病毒”的国内版本。

以 ‘right?’ 结尾的句子

最近有人跟我说话，忽然冒出一句英语，而且句末加了一个 ‘right?’。我心里忽然哽了一下。不仅是因为中国人说话忽然冒出一句英语，而且因为这个句末的 ‘right?’

在美国的时候，经常听到有人这样说话，似乎习以为常。离开美国几年之后，再听到这样的句子，却忽然注意到它隐含的让人不快。这真是有意思。

因为最近都在琢磨美国英语的问题，今天想起这件事，仔细分析，并且上网搜索调研，才发现原来这种 ‘right?’ 在末尾表示疑问的句型，也是一种隐藏的“语言病毒”。

很多人经常听身边人用 ‘right?’ 结束一个句子，自己也开始这样说，却没发现这是一种居高临下，不礼貌，不尊重的语言。久而久之，这种语言就像病毒一样传播开去，我们就又离文明远了一些。

一个陈述句，加一个 ‘right?’ 在末尾，是什么意思呢？这不是问你问题，或者征求意见，而是陈述一个自己相信的事情，并且想确认你也认可。他只是在要求你认可他说的话，而不是在征求你的意见。“你听明白了吗？明白了就说‘对！’”他期望你的反应是口头认可，或者点头。

简言之，说 ‘right?’ 就是在迫使你点头。不停地问 ‘right?’ 就是不停地迫使你点头。这是一种居高临下，潜移默化奴化其他人的语言。什么人会不停对别人点头呢？奴才。

这样一种居高临下的语言，却被很多美国人用于日常对话。在美国的时候，很多人这样说话。本来应该是一个疑问句，比如本来应该说：

Do you like football? 你喜欢足球吗？

结果说出来一个陈述句，最后接一个 ‘right?’

You like football, right? 你喜欢足球，对吧？

为什么第二种句型显示出居高临下和不尊重呢？因为它表面上在问你问题，却强制给你安排了一个答案。比如这个例子，对方好像假定了你喜欢足球，而你可能并不喜欢，他却等着你说 ‘right!’。可能因为他自己喜欢足球，所以他认为所有人都应该喜欢足球。

普通的疑问句，相当于在末尾有一个 ‘yes or no?’ 可是这种句子，结尾是 ‘right?’，并没给你 ‘no’ 的选择。

普通的疑问句，你可以回答 ‘yes’ (right) 或者 ‘no’。可是这种 ‘right?’ 结尾的，前面是一个陈述句，而不是疑问句。接一个 ‘right?’，隐含的意思是：我已经决定了，肯定就是这样，我期望的回答是 ‘right’，你最好说 ‘right’，不要说 ‘no’，我等着你说 ‘right’，你说了 ‘right’ 我就可以继续了。

很多时候有人说出 ‘right?’，却并不等你回答就继续，默认了你的回答就是 ‘right’。也就是说，说这句话的人并不在乎你的回答。他已经单方面决定了答案，只是象征性的问你“对吧？”

这就像一个知道考试答案的老师，在问一个学生。“这个是这样，对吧？” “你知道这个，对吧？” 他不是在问你，不是在讨论，而只是想确保你明白了他认为一定是正确的东西。所以显示出不尊重，居高临下。

警察录口供的时候，陈述一个他所了解的事情，然后说“Am I correct?” “Yes or no?” 这都比 ‘right?’ 好点，因为他给了你否认的机会。

“4月27号晚上 11 点，你去了这个地方。Am I correct?”

所以 ‘right?’ 给人是怎样的感觉，你们体会一下。实际上，我觉得受过专业训练的警探，不可能结尾说 ‘right?’。因为法律给了你说 ‘no’ 的权利，他这样说就是在强迫你接受，是要被举报的。

如果 ‘right?’ 前面这个句子是事实，那只是显示出不尊重和自我中心。可要是前面这个句子是事实性的错误，你根本没法说 ‘right’，那么就显示出说话人的愚蠢，自以为是。

而且有些美国人会把这个 ‘right?’ 里面的 ai “哎” 这个音发音比较扁，张口较小，听起来就像 ei “唉”。听起来有点像 ‘rate?’，有种乡土味。再加上这个句型隐藏的强迫含义，经常这样说，就显得缺乏教育，素质低。

我给大家的建议是：中国人说的日常语言，除非是学术术语，就不要冒英语出来了。另外，必须说英语的时候，应该停止使用这种末尾的 ‘right?’

你还可以参考 Quora 上对于这个问题的[讨论](#)。

使用干净的语言

当然美国英语的常见毛病不止这些，我只是发现了一些典型问题。这些英语语言现象，应该像中文的“[网络用语](#)”一样，主动避免。不管用什么语言，我们都应该讲究良好的风格，不要因为身边人都在那样说，就跟风学一些不好的习

惯。只有这样，我们才会成为文明人，我们的社会才会更加舒适。

一对一教学计划

(更新：一对一教学的时间跨度，原先设定为不超过两个月。根据实际情况，现在调整一下，改为不超过三个月。)

(更新：由于一对一教学需要的时间和精力较多，重复讲课对于老师来说比较枯燥，所以我决定减少一对一教学的数量，将价格调整为 18000，对学生也会更有选择性。请大家理解这个改变，欢迎大家积极报名集体课程。)

开展[计算机科学基础课程](#)两个星期了，我终于可以相信，最初的目标“零基础教育”是一个很好的出发点。不仅因为它能检验和改进我的学识和教学方法，而且因为零基础的人似乎更容易学会正确而干净的知识。

这一期课程的成员有各种各样的背景。有些同学没有理科背景，没上过大学，有些完全是出于兴趣，有些已经博士毕业几年。他们短短两个星期以来的表现，思考问题的角度和深度，学习的态度和动力，让我惊讶又欣慰。每当看到他们的进步，就觉得这一切都是值得的。我正在改变一些人的人生。

我教学的内容和方式，跟学校的计算机教育有着大幅度差别，而且根据学生的反应，会尝试更好的做法。我发现对计算机一无所知的人，比知道一些东西的人，很可能更容易理解精华的思想，更愿意敞开心扉去尝试，而不会产生怀疑，骄傲，急躁和抵触情绪。就像一台干净的新电脑，里面没有陈旧的内容混杂在里面，是更容易安装新型系统的。在我的班上，无知可以是一种福利。

我最近在试验一种新的教学方法，那就是完全的一对一教学。小班授课的效果已经很好了。课程中的每个同学跟我都有私信交流，在课后接受大量个人指点。上了课，每个人有不同的理解深度，不一样的作业进度，犯不一样的错误。这些我都针对每个人进行不同的指点。

有些同学上课不好意思说话，怕别人笑自己，以为别人更聪明一些。这些我无法控制的心理现象，稍微阻碍了原来设想的“苏格拉底式”教学的实现。但就算稍有缺点，这样的教学，比起大学的计算机基础教育，都是强很多的。不仅是在于教学内容，而且在于方法和方式。

常规的班级当然还是欢迎大家继续报名，但我同时也接受少量一对一的学生。实际上我已经有一些这样的学生，而且效果很好。现在我描述一下这个新做法：

1. 没有固定的上课时间。课程基本通过微信消息进行，这样可以实现真正的“苏格拉底式问答”。我循序渐进提出一些问题，学生回答。对于不好解释的概念，也可能通过短时间 Zoom 会议方式进行。当然为了问答的有效，还是需要有一段时间的持续互动，而不是一条消息隔很多个小时。
2. 由于微信消息的灵活性，这种教学是跨国界，跨时区，非常灵活的。你甚至可以在茶余饭后接受少量教学。我的消息包含了有益的问题和提示，学生可以思考一段时间之后再回复。没有人盯着你，等着你，没有同学的压力，更有利于自在思考。
3. 由于大部分知识是学生通过回答问题，独立思考得到，所以比起传统课堂传输的知识，可能会更加可靠，容易灵活运用。
4. 内容范围还是跟普通班的范围相同，只是教学方式不同。请明确自己的期望值，不要认为“一对一教学”等于你想知道什么，我就教你什么。内容跟普通课程一样，是事先规划好了，只是一对一方式可以根据你的个人时间，反应，调整方式来让你学会这些内容。请有较多经验的人士不要利用这个“零基础课程”来获得超出范围的信息，以免失望。
5. 课程没有指定的教科书，不会按任何一本书来进行教学。提前阅读某些书籍（比如 SICP）不会对教学有好处。正好相反，看了其它内容并深陷其中，也许会对教学产生相反的作用和误导。我不想花太多工夫解释由于已有知识而产生的困惑，最好的方式是试图先忘记一切。
6. 时间可以随时开始，时长在 3 个月以内。如果你进展特别快，可能最后没有新的内容了，你就学习成功了。如果你特别慢，那么仍然会在两个月的时候结束。这样是为了避免有人不努力，结果无限拖下去。
7. 这个课是重视实践的，不适合不愿意动脑动手的人。请不要以为躺在那里，随便说几句话就可以学会。就像健身一样，你不能躺着不动就长肌肉。你需要按照我的指导，对自己的头脑进行循序渐进的练习，让它变得强大而精密。自己动脑，动手是必要的，而且练习会很多。我会要求你给出非常具体，精确，可以运行，优雅的代码。我会以很高的标准要求你，稍有不够好的地方就会要求你修改，经常我会要你简化，简化，再简化…… 你会写出一些艺术品级别的代码。我不喜欢自己不努力的人，不会回答未经思考问题，未经执行的代码，或者重复说过的要点。
8. 一对一教学费用是 18000（新价格）。

因为每个人都是单独进行，容量会有一定限制。目前看来，因为对于这些内容是轻车熟路，对我个人的压力是极小的。我很有信心用这种方式，从零开始培养出更多的高级技术工作者和计算机科学家。

另外请注意，这个一对一教学仍然属于“零基础教学”，不适合有多年经验的工程师。因为我有设定好的内容界限，所以如果你已经会了很多内容，恐怕会浪费我们双方的时间。我一说什么你都会了，就不好进行问答了。

对这个方式感兴趣的人，可以使用[普通课程](#)同样的方式报名，来信请注明《申请一对一教学》。请考虑清楚，对我有足够的信任，决定参加再联系我。我没有时间和精力来回答各种顾虑，不想说服任何人参加。请不要报名沟通之后再讲价或者要求分期，浪费大家的时间。

计算机科学入门班报名

(2020.05.29 更新：根据实际授课情况，更改了课程的内容简介。去掉了一些可有可无的内容，增加了解释器等关键的内容。)

(2020.04.28 更新：半个月过去了，却似乎过了一年。零基础的同学们得到了让我有点吃惊的进步，我为此感到欣慰。同学们也反应一周两节课已经忙不过来，因为我会给他们很多练习，引导他们进行独立思考。所以考虑之后，决定把课程的间隔改为一周一次，这样课后有充足的时间来练习，消化，吸收。)

(2020.04.16 更新：经过一段时间的准备，第一期的计算机科学基础课程已于 4 月 13 日星期一顺利启动。由于人数和时区限制，后来报名的同学会被安排到下一期参加。另外由于北美时区关系，可能会有一个专门的“北美班”，等待合适的时间启动。欢迎大家继续报名。描述请说明希望报名参加“第二期中国班”还是“第一期北美班”。)

我需要更多的“小白”来上我的入门课。自从上次准备开课一来，我这里有挺多申请者，但他们很多都已经有一些基础，有工作多年的工程师，团队带头人。不过我想先讲“零基础”课程，而让那些需要更进一步的人等等。

为什么重视“零基础”教育

有些人可能不大明白我为什么喜欢讲“零基础”课程。一方面，真正好的教育应该是能让完全无基础的人顺利掌握的。就像爱因斯坦说：“如果你不能给一个六岁小孩解释清楚，那你并不真的懂。”所以“零基础”的学生能够检验我是否达到了这个“真懂”的目标。

实际上，我的很多深刻理解，都是通过反复琢磨非常基础的概念获得的，而不是通过很“高级”，很复杂的概念。我最常用的“心理模型”，其实跟初学者第一节学的内容差不多。

在我心里并没有“初学者”和“资深者”的差别。我发现很多工作了几十年的工程师，很多连最基本的概念都是一知半解的，这也许就是为什么他们在工作中无法找准正确的方向，经常瞎撞。

经验告诉我，有一些基础，特别是从错误的地方学过一点东西的人，容易在教学中产生各种麻烦的心理。加上急功近利的目的，甚至会表现出“你讲的这些对我有什么用？”的态度。

我不希望跟这种人对话，反而觉得教完全不会的人更开心一些，我甚至愿意教小孩子。看着他们从一无所知，到逐渐领悟，甚至在某些方面超越资深工程师，这比起教其他人更有成就感。

从社会的角度，一无所知的人是最需要帮助的。因为他们的思想不受已有知识的牵绊，也是最容易吸收干净思想的人。改善他们的生活和思维能力，会让我觉得更有意义。

对 PL 表现出极大兴趣的人，我都会比较谨慎。因为我经常提到 PL，已经有太多功利人士，试图通过这个方向取得地位上的“优势”。有时在别人转发的知乎帖子看到有“PL 人”发一堆让人不明觉厉的术语，让我担心我的教学会助长这些人的气焰。

有人告诉我，都是因为我总是谈 PL，知乎上才有这么多人鹦鹉学舌，用一堆术语打压其他人。我之前毫不知情，但我觉得不能再不帮助他们了。有些人已经得到太多，却想进一步取得压制其他人的优势，我不喜欢这种贪婪的人。

我对 PL 和编译器人的一些看法，已经在之前一些博文里说得比较清楚了。我希望避免“培养”出太多这类人。从我学到东西，到头来成为我和其他人的灾害。所以我教学初期肯定不教 PL 专业的内容。

这对于真心想了解 PL 的人来说可能是一种不幸，但也许有某种方式调和这种矛盾。总的说来，传授甚至指点关于 PL 的方向，我都会很谨慎，而且会很贵。

我最近的课程，恐怕要针对完全的小白。同时我会根据从中获得的教学经验写一本人人都能看懂的书，然后进行进一步的教学，写更深入一些的书，如此循环…… 最后我希望破解很多计算机的领域（比如操作系统，数据库，AI），把它们化繁为简。

由于目前小白数量太少，我欢迎不会编程，或者知道很少的人来报名。

课程内容

课程计划涵盖计算机科学的主要思想。当然因为时间和学生实际的吸收速度，到时候可能会有调整。大概会包括以下内容：

1. 基础语言构造，包含最常用几种语言的主要特性。
2. 递归思想，递归数据结构的处理。
3. 基本数据结构，少量基础算法。
4. 函数式编程基本思想。
5. 抽象的思维方式。

6. 基础的解释器原理。

如果从书籍的覆盖面来看，我试图包括以下书籍的精华内容：

1. SICP (前 4 章)
2. The Little Schemer
3. A Little Java, A Few Patterns

你将受到的训练

1. 掌握系统化的思维方法，严密的推理技巧
2. 写出简单，优雅，容易理解，可靠的代码
3. 从无到有，不依赖于任何语言的特性，解决各种计算问题的思路

授课方式和理念

1. 采用网络授课方式。
2. 小班，人数不超过 12 人。
3. 有聊天室讨论课程内容，有合适数量的作业和思考题。
4. 无教条主义，无死知识。

课时和收费

因为是第一次授课，所以时间只是初步估计，到时候要以实际情况为准。

1. 估计两个月完成课程。如果中途有事耽搁，或者感觉太紧凑无法消化，可能按需少许延长。
2. 每周 1 次课，总共 8 次。
3. 上课时间安排在国内时间下班后，晚上 8 点的样子。
4. 为了讨论充分放松，每堂课 2 小时左右。
5. 收费暂定每人 12000，可能根据实际人数调整。

人员要求

1. 没有很多已有计算机知识。
2. 心理健康。谦虚好学，能平等讨论。

申请方式

发送 [email](#)。标题：申请计算机科学基础课程。内容包括：

1. 简单自我介绍。教育经历，工作单位，工作内容，上课的动机等。
2. 你的时区，因为可能按时区分班。
3. 附上简历。

开始时间

我打算在一两周之内安排面试申请者，准备一两次课的内容，然后开始教学。

一对一课程

经过一段时间的实验，我推出了新的一对一课程方式。如果考虑一对一的教学，你可以参考[这篇文章](#)。班级授课的模式仍然会继续进行，只是你可以有两种选择。

对智商的怀疑

想写一篇「新手上路指南」有一段时间了。一方面，因为顾问工作和知识星球上存在很多类似的问题。另一方面，因为这些问题引起了我内心的共鸣。

在清华读博的时候，我曾经跟这些“新手”一样，问过同样的问题。甚至到了 Cornell 的时候，这些问题仍然存在。无人解答这些疑问，每个人好像都很聪明，学得很轻松的样子…… 现在这些问题再次回到了身边，我才发现有那么多人需要帮助。

从哪里开始呢？先从“智商”和思维能力开始吧。

对智商的怀疑

写下「对智商的怀疑」这个标题，我就发现它有双重含义。普通的含义，就像一些同学提到的，他们怀疑自己的智商，害怕自己生下来智商就如某些“学霸”，觉得自己天生就不能思考有点难度的问题。

不过在我这里，「对智商的怀疑」有着完全反过来的含义：我怀疑「智商」这个概念，怀疑它的科学性，怀疑用来测智商的方式，怀疑所有拿智商说事的人。

我发现怀疑自己智商有问题的人，问题往往不在于他们的「智商」，而在于他们的「怀疑」。人脑的局限性往往产生于自我怀疑。本来一般人都可以有条不紊思索出规律，找到答案，结果有人怀疑自己的能力，就做不到了。怀疑的心理内耗了人的思维能力，造成了思维线路的瘫痪。

我听说一个故事。有个心理医生曾经给一个举重大力士催眠，让他相信自己瘫痪了，抬不起胳膊。结果你猜怎么的，在催眠之下他居然相信了，所以就真的抬不起胳膊来。信念的力量是巨大的，如果你不相信自己能做到，那你很可能就真的做不到。

很多人都希望你有这种错误信念，这样他们就能显得比你聪明，比你智商高，然后就可以收你的智商税了。

实话说，我也怀疑过自己的智商。高考失利之后，我进了一所不理想的大学。本来到了这种地方好像应该很轻松才对，结果却觉得学好多东西都很费劲。旁边的同学总是在我耳边吹风，说：“你看人家某同学，也是北大没考上落到这里的，人家小学就开始学编程了。你虽然成绩好，可是你上大学之前都没摸过真正的电脑……” 如此各种比较，有些人只要一有机会就会打击我的自信。

在军训的时候，听别人滔滔不绝谈论“Win 95”，问“Win 95 是什么？” 答曰：“Win 95 就是 DOS 的升级版。” 其实我心里还在问“DOS 是什么？” 但是没好意思再说出来丢脸…… 高中还存在的自信，就这样被消灭了。逐渐的，我发现看书怎么每个字都认得，却看不懂它在说什么。我开始怀疑自己的智商。我是不是真的难以理解数学，难以理解算法，难以理解计算机相关的各种问题？我高中那些聪明，恐怕都是假的吧？我就是个低能儿也说不定，只不过因为高中科目都太容易，所以我才好像挺厉害？

这种情况一直持续到我大学毕业，直到我进入清华，甚至延续到 Cornell 时代。在 Cornell 的时候，情况更加严重一些，因为身边充满了各种雀跃的“天才”。Cornell 当时有几个被大家吹上了天的教授。我开头还真以为他们是天才，在别人眉飞色舞的鼓吹之下，去上他们的课。结果每次选这种课，都发现教授背对学生在那里写黑板，自言自语，学生拼命在下面抄笔记。当然，最后我都以听不懂，跟不上，退课而告终。

我以为我是唯一不称职，智商低的学生，直到我遇到另一个打算退课的学生，他是一个印度人。我那时候已经受不了这个课了，却没人能表达我的感受。正巧那天我看他走出教室面无表情，就打探了一下他的口气：“你觉得这个课如何？” 出乎我意料，他并没有像其他人一样说“好得不得了”，而是摇摇头：“他这叫讲课吗？” 我才终于发现我不是独自一人。

后来在各种打击之下，我寻求了一些启发，才发现原来他们并不是真正的天才，而是被吹捧起来的。天才教授们当年那么红，是因为他们研究的是“社交网络”，而当时正是 Facebook 红极天下的时候。所以随便拿点社交网络数据，用一点点图论知识，写点 Matlab 代码画个图，就能发一些顶级刊物的 paper。言必称“六度分离”，虽然“六度分离”不是自己提出来的，但你可以说“是我第一个推广它”的呀。还说 Google 的 pagerank 算法是受了自己 paper 的启发……

逐渐的，我察觉到死钻这样的“算法”没有很大意义，而且并不是我真正喜欢的事情。同时我也对“P vs NP”之类问题的重要性产生了怀疑（Cornell 是“P vs NP”问题的故乡）。后来我转向了编程语言（PL）。其实自己对编程语言一直存在好奇心和疑惑，干嘛不就拿 PL 作为研究方向呢？

找准了方向，忽然就有动力了。一旦有了动力，似乎就不存在很难的事情了。虽然我的思考起初可以很慢，我遇到一个新概念，需要很多时间才能弄明白它是怎么回事。可是我发现慢归慢，却终究是可以理解和把握的。

我在书桌旁看了几个小时可能看不懂，可是我吃饭时还可以想，我吃完了饭去散步时还可以想，我坐公车去买菜的时候也可以想。不是绞尽脑汁的想，而是慢条斯理，优哉游哉的想…… 头脑的工作是不受时间和地点限制的，所以我头脑虽然好像很慢，但可以有很多时间用来思考。

哦，忘了告诉你是谁启发了我——费曼。正好是在那段时间，一个朋友跟我提到费曼的自传《Surely You're Joking, Mr. Feynman》（中文名：别闹了，费曼先生）。我去图书馆没找到那本书，却找到费曼的另一本自传《What Do You Care What Other People Think》。就是这本书告诉了我，世界上有许多假装很聪明的人。其实每个人都不笨的，只是他们被这些自称为“天才”的人骗了。

费曼讲到他的女朋友被医生误诊的故事。医生说她得了某种绝症，活不过一年。结果费曼自己查阅了医书，发现根本就不是那个病，而是另外一种。虽然也是绝症，但没那么厉害，至少还可以活五年。费曼毅然跟她结了婚，到一个安静的地方买了一座房子，陪了她几年，一直到她去世。

这个故事告诉我们，别太相信权威了，即使是你领域之外的权威，也可以错得离谱。你完全可以靠自己的力量，获得很多领域的正确知识。费曼这本书里还有很多其他例子，他作弄过各种自以为是的专家。

这本书开启了我为自己破除权威迷信的路，让我找回了自信，找回了自己的思考。我不再相信智商这种东西。我其实很怀疑「智商」这概念的提出者自己的智商，他有什么资格来评价其他人的智商？人的能力是多种多样的。用仅仅一个数字就可以评价别人头脑的能力，似乎只是满足了某些人不切实际的妄想。

思维快慢无所谓

除了怀疑自己的智商，很多人存在的一个误区，是把「聪明」等价于「思考很快」。如果他们发现一个问题好像很复杂，首先产生了恐惧心理。如果发现看了好一会还没弄懂，他们会觉得自己头脑“太慢”，继而开始怀疑自己的智商。越是怀疑就越是心浮气躁，无心思考。无心思考，当然就得不到结果。

所以破除智商迷信的一个要点，就是不要盲目追求“想得快”。

世界上存在许多传说，用来夸赞一些公认的天才头脑有多快。比如冯诺依曼就有这样一个故事。有个年轻人问冯诺依曼一道[数学题](#)：

两列火车相隔 200 公里，各以每小时 50 公里的速度相向而行。一只苍蝇从其中一列前端出发，以每小时 75 公里的速度，在两列车之间来来回回飞个不停，问：直到两车相撞，苍蝇飞过的总距离是多少？

冯诺依曼沉吟几秒钟答道：“应该是 150 公里。”年轻人惊叹，拍冯诺依曼马屁道：“您真厉害。普通数学家总是忽略简单方法，而采用无穷级数求和的复杂方法。”冯诺依曼说：“我用的就是无穷级数求和的方法啊。”

很多人喜欢拿这故事来说明冯诺依曼的头脑速度有多快，可以几秒钟算出无穷级数求和，听了的人还都信了。我起初也信了。后来受到费曼的启发之后，再次遇到这个故事，我就开始怀疑了。有任何证据证明这件事发生过吗？它完全可以是冯诺依曼的崇拜者编造出来，用来吹嘘他有多聪明的。

真正聪明的人是不会在乎自己头脑有多快的，只要它思考一段时间，外加查阅各种资料，能解决问题就行。这就跟真正聪明的程序员不会炫耀自己打字有多快是一个道理。打字快，可是打出来太多垃圾，又有什么意义呢？

我不相信冯诺依曼自己听了这个故事，会认为是在赞美他。如此低级的问题，如此低级的标准，这不是辱没一代大师的威名吗？

我能想清楚很多事情，可是我的头脑真的不快。跟我讨论过问题的人，我的同事，都知道我的这个特点。我不但想得慢，而且理解速度也慢，经常需要他们讲好一会儿，还得画个图，才能把我讲明白。

一旦把我讲明白了，我就会明白得很深入，我还能帮助他们自己理解这个事。我养成了刨根问底的习惯，很多问题我一直会追问，知道能用「直觉」解释清楚，用「心之眼」看得见，才会善罢甘休。

我现在工作中，经常带领大家走弯路，走错方向的人，往往是那些喜欢炫耀自己“想得快”的人，而且他们说话也快，还大声。这使得他们无法深入思考，无法听到别人在说什么。

许多想法都可以来自跟人讨论，可是喜欢显示自己头脑快，很聪明的人，是听不到其它人的想法的。他们把每一次讨论都当成了显示自己聪明的机会，当成了抢答比赛。所以他们就成了“单核处理器”，虽然主频很快，也终究无法跟主频低很多的“多核处理器”抗衡。

所以对自己的头脑没有信心的人，首先应该破除对速度的崇拜。不要一味求快，但求想得深入。

(待续.....)

如果你喜欢这篇文章，可以扫描这篇文章的专用支付宝二维码付费。



英语学习的一些经验

最近很多人问到关于英语学习的问题，所以我想稍微总结一下自己的经验。

技术人员的英语学习，我的经验是，首先肯定要专门学习英语，然后可以读英文技术书籍和文档。技术书籍的难度一般比小说等文学作品小很多，因为他们得照顾非英语国家的人，都是很简单的单词和语法。

很多人（包括我）看英语技术资料完全没问题，阅读小说一般还是有点慢的。小说往往有各种花哨的语法和单词，是技术书籍没有的，所以先看小说可能会牛刀杀鸡。

如果你觉得技术书籍对你还比较难，也许可以试试 Don Norman 的『The Design of Everyday Things』。这本讲设计理念的书，没有什么技术难度，但英文浅显易懂，可以拿来一边学习英语，一边学习设计。

读书我一般都是默读。如果你觉得必须出声，那你阅读能力还没有练到位。默读是直接的“光 => 视觉系统 => 语言系统 => 语义”的转换。如果你出声了，或者在头脑里“默念出声”，那你就多了一些步骤：“光 => 视觉系统 => 听觉系统 => 语言系统 => 语义”。

需要跟外国人交流的时候，肯定听说读写都要会。但是“说”可以另外练，不需要在看书的时候出声。

任何语言的听说读写，理解语法都是很重要的。但注意：“理解语法”不等于看语法书，死记硬背语法规则。我在「解谜英语语法」里面已经说过，大部分英语语法书都是不能实用的教条，甚至根本就是错的。你只需要掌握最精髓，可以实用的部分。

英语或者任何自然语言，最精髓的部分都很像编程语言。句子是最关键的结构。每个句子都是一个「函数调用」。动词（谓语）是函数名，其它内容（主语，宾语等）都是参数。每个部分又可以有修饰语，就像对象的 property 一样。理解这一点可以帮助你快速分析句子结构，而不是停留在字面上。

如果你理解了句子是一个函数调用，那么你就会懂得何时该使用句号。很多中国人对句子没有清晰的概念和边界。本该是句号的地方他们却打逗号，所以你不知道他的句子到哪里结束。如果你不能清楚的分辨出句子，那你就不能很好的理解里面的逻辑。

人脑处理语言有一个隐含的「parse」过程，就像编程语言的 parser。你需要训练自己的 parse 能力。利用上面对句子结构的理解，你可以快速分析出句子里最关键的“骨架”，然后再填充修饰部分。这样你就能理解长句。

最好花一些时间专门练习这种 parse 能力。反复读一些你感兴趣的英语技术文档。可以是书籍，也可以是网络上的 blog。比如你可以拿两段话来练习，把每句话按照我提出的“函数调用”结构分解开，画出函数名，参数，修饰语，构造一棵「语法树」。反复琢磨语法树的结构，然后再回去读这两段话，直到你可以迅速把每句话都看出「语法树」来。

很多人练习英语阅读，喜欢读很长的文章，其实效果不大好。因为每句话都没有分析清楚就掠过了，结果读到最后还是没有提高 parse 能力。反复读同一段话，仔细分析自己的失误原因，纠正反复发生的错误。就像深度学习一样进行 back propagation，训练头脑里的神经网络 parser，提高 parse 准确度。

我不是说你得一直这么留意语法树。等头脑里的 parser 被训练到纯熟的地步之后，你就不需要刻意去想这件事了。英语 parser 一直悄悄工作着。你以为它不存在，其实它只是熟练了，自动化了。

语法和句子结构是关键，其次才是词汇量。如果你的词汇量足够阅读技术文档，那就可以开始看了。偶尔有不认识的词，临时查一下字典，拿小本子（不要用手机）写下来。下次遇到同样的词，还不认识，就看看你的小本子。多几次就记住了。

如果真记不住，你可以在单词旁边画一幅简笔画来表达单词的意思。画夸张一些，发挥你的创意。这幅画不需要完美，只需要努力去画。画的内容不需要符合逻辑，你甚至可以把单词拆成几个毫不相干，或者只是长得有点像的词，然后对它们画画。不要去拿别人的“助记法”来用，因为你拿了别人的，就不会努力。

手写单词这种「努力」，可以训练你的大脑，把单词“刻进”你的记忆里。号称“轻松”的记单词方法，效果可能都不会很好，因为努力是产生记忆的一个重要原因。就像努力是健身练出肌肉的原因一样，你不可能躺在那里就长肌肉。

不拿手机记单词的一个原因是，手机不是随时都能看的。它可能锁屏了，你得多好几个步骤才能翻到你的单词表，就很慢。手机上不时弹出消息，很分心。拿小本子记，摆在旁边立即就能看。

另外，字典的选择也是有讲究的。对此我最强烈的建议是：一定要用英英字典，不要用英汉字典。无数的经验告诉我，就算用最好的英汉字典，也会严重延缓甚至误导对英文单词的理解。

英汉字典的释义一般是几个意思相近的中文单词。很多人觉得这样看起来轻松，以为看了那些中文单词就能理解，这就是很多人一直用英汉字典，无法真正理解英文单词的原因。他们没有明白一个道理，只有英文才能准确解释英文。英英字典里面的释义，一般是一句话，外加很多上下文信息，而不只是短短几个近义词。这种方式才能准确描述单词的意思。

英文和中文之间有巨大的鸿沟，英文里的很多意思，中文里根本就没有。所以再好的英汉字典，释义也不能很准确。中文根本没有那个意思，那怎么解释呢？只能用近似的词，可是这样一来，词义偏差就很大了。所以用英汉字典只能勉强知道是什么意思，却无法真正理解英文单词。

很多人担心英英字典看不懂，而其实英英字典都是用最基本的词汇解释，所以不用担心看不懂。只要你会了基础的单词，就应该开始用英英字典。

我推荐的字典是牛津英英字典。很多人崇拜美国，因为美国很发达，所以觉得学英语必须学美式英语，用美国人编撰的韦氏字典。可是经验告诉我，韦氏字典比牛津的差很多。不仅释义没有牛津字典准确，不容易理解，而且有些词的词性都可能标错。韦氏字典的例句，基本是片段，而不是完整的句子。例子数量比较少，质量不高。排版也没有牛津的美观。

虽然美国经济科技发达，而最正宗而优美的英语，仍然是英国的。

我看到评论里有人说读技术资料，其实中文就可以了，但我觉得中文资料非常不容易理解，容易被误导。很多中文内容是翻译过来的，作者自己也不怎么懂，经常翻译不到位。

中文对于描述技术细节有先天的弱点，很多术语用中文写出来，夹在一段文字里，很不容易分辨出来。因为汉字之间是没有空格的，中文术语跟普通词汇长得差不多样子，所以眼睛很难看出中间哪些是术语，哪些是普通词汇。

这就是为什么我写中文技术文章，都喜欢给“术语”加引号。后来觉得引号有点不明显，容易跟说话的引号混淆，所以开始给「术语」加直角引号。但很少有中文技术文档这样写，所以看起来很辛苦。我不喜欢把一些英文术语写成中文，也是这个原因。比如如果你写卫生宏，我会很难理解它。你写 hygienic macro，就很明显那是一个术语。

我觉得日本人很聪明，他们的文字分两种，平假名和片假名。平假名拿来写平常用语，片假名专门拿来表示外来词。这是很合理的做法。我觉得写中文技术文档也可以借鉴这个做法，干脆把术语都用原来的英文表示。

扯远了…… 我本来要说英语学习，但这对你们写出好的中文也会有帮助。

而且英文的技术资料质量和数量都比中文多很多，还有很多非常有价值的视频在网络上，所以学好英语是至关重要的。希望大家都能提高英语水平，多看英语文献。

完。

如果你喜欢这篇文章，可以扫描这篇文章的专用支付宝二维码付费。



计算机科学课程

经过一段时间的顾问工作和聊天沟通，比较深入的了解到国内计算机教育的现状和大家对于学习的困惑，我觉得是可以“系统授课”的时候了。只是这次的授课，恐怕和我原来设想的方式有很大不同。

每当需要“系统”的准备任何事情，我都会犯严重的拖延症。一方面是我其实不想对别人负责，而且“系统”这个词的含义，对我来说也很模糊。心里有了“教好这门课”，“系统化”……这些目标，反而很难真正开始准备课程。我要教他们什么内容呢？从哪里开始？要是我没理解他们，他们听不懂怪我呢？做幻灯片好麻烦啊，精益求精……一系列的紧张。

最后经过思考，我发现自己最有效的工作方式，其实不是系统而周密的准备，而是即兴的发挥，边做边想。不管是参加我的顾问服务，还是加入我的聊天室的人，都发现学到了很多。他们是从对话中去领悟，而不是传统的课堂。我不是作为一个“真理传播者”，而只是一个“启发者”。我甚至可以有些时候头脑不清楚，概念也稀里糊涂，而且很多的“新概念”我其实没仔细看过，我甚至可以完全是错的。

但通过对话和提问，我们产生了头脑里从来没有过的知识，而且深刻理解了它的来龙去脉。这种认识可能超越文档或者书籍。

这就是对话的力量，这让我再一次想起“Little 系列”书籍。为什么《The Little Schemer》是对话的形式，为什么它用那么短的篇幅，可以教会我比普通书籍多很多的东西？我一直很好奇这个现象。

有一次我问 Friedman，你的书怎么都是这种对话风格啊？这叫什么 style？他笑了：“这叫 little style！”后来我发现，这种对话式的教学方式几千年前就出现了，叫做「苏格拉底方法」。

苏格拉底承认他自己本来没有知识，而他又要教授别人知识。这个矛盾，他是这样解决的：这些知识并不是由他灌输给别人的，而是人们原来已经具有的；人们已在心上怀了“胎”，不过自己还不知道，苏格拉底像一个“助产婆”，帮助别人产生知识。

孔子似乎也用了不少对话式的教学。不过孔子的方式是学生提问，老师只是回答，像顾问似的；而苏格拉底的方式是老师提问，学生回答。方向是相反的。

我个人觉得苏格拉底的方式要好些，因为苏格拉底方法中的老师并不是“布道者”或者“圣人”的角色，而只是一个“启发者”。对于某些话题，老师可以本来没有知识，但是老师很会提问，通过追问和对话，最后大家都得到了新的知识。另外，没有基础的学生经常不知道该问什么问题，所以老师提问可以启动学生思考。

没想到两千多年前，有人认识到比我们深刻很多的教学原理，而且它在我的实践中被印证了。每一次有意义的对话，都是一场头脑的 SPA，让人身心愉悦。当初我是作为学生，现在我是作为“老师”。

我发现“老师”和“学生”的界限越来越模糊。到底是谁教会了谁，也许其实并没有谁教会了谁？是的，作为一个“老师”，很多时候我并没有知识，可是通过对话式的教学，却产生了知识。

Friedman 的情况也类似。他问你的有些问题，可能他自己都还没想清楚，但他经常问很好的问题。跟他对话的时候，他不总是对的，有些时候甚至稀里糊涂的。可是他会指引你朝着破除谜团的方向前进，最后他和你一起把问题想得清清楚楚。

其实有好几本新的 Little 书，都是 Friedman 开头不熟悉的主题：《The Reasoned Schemer》，《The Little Prover》，《The Little Typer》……可是通过与合作者讨论，从他们身上学习，找该领域最高明的专家咨询，他把这个领域给嚼烂消化掉，然后把精华的营养都写进了书里。

鉴于如此成功而快乐的试验，我决定不再试图准备一堂“系统”的课程，而是像爵士乐手一样即兴发挥。我授课的内容，授课的方式，都可能即兴改变。

一道 Java 面试题

关于程序员对 Java 类型系统的理解，比较高级的一个面试问题是这样：

```
public static void f() {  
    String[] a = new String[2];  
    Object[] b = a;  
    a[0] = "hi";  
    b[1] = Integer.valueOf(42);  
}
```

这段代码里面到底哪一行错了？为什么？如果某个 Java 版本能顺利运行这段代码，那么如何让这个错误暴露得更致命一些？

注意这里所谓的「错了」是本质上，原理上的，而不一定是 Java 编译器，IDE 或者运行时报给你的。也就是说，你用的 Java 实现，IDE 都可能是错的，没找对真正错误的地方，或者没告诉你真正的原因。

如果你知道哪里错了，并且知道「为什么」错了，可以联系我。

如何阅读别人的代码

挺多人问我「如何阅读已有代码」这个问题，希望我能有一个好方法。有些人希望通过阅读「优质项目」（比如 Linux 内核）得到提高，改进自己的代码质量。对于这个问题，我一般都不好回答，因为我很少从阅读别人的代码得到提升。我对自己阅读的代码有很高的标准，因为世界上存在太多风格差劲的代码，阅读它们会损害自己的思维。同样的道理，我也不太会阅读风格差劲的文章。

但这并不等于我无法跟其它程序员交流和共享，我有别的办法。比起阅读代码，我更喜欢别人给我讲解他们的代码，用简单的语言或者图形来解释他们的思想。有了思想，我自然知道如何把它变成代码，而且是优雅的代码。很多人的代码我不会去看，但如果他们给我讲，我是可以接受的。

如果有同事请我帮他改进代码，我不会拿起代码埋头就看，因为我知道看代码往往是事倍功半，甚至完全没用。我会让他们先在白板上给我解释那些代码是什么意思。我的同事们都会发现，把我讲明白是需要费一番工夫的。因为我的要求非常高，只要有一点不明白，我就会让他们重新讲。还得画图，我会让他们反复改进画出来的图，直到我能一眼看明白为止。如果图形是 3D 的，我会让他们给我压缩成 2D 的，理解了之后再推广到 3D。我无法理解复杂的，高维度的概念，他们必须把它给我变得很简单。

所以跟我讲代码可能需要费很多时间，但这是值得的。我明白了之后，往往能挖出其他人都难以看清楚的要点。给我讲解事情，也能提升他们自己的思维和语言能力，帮助他们简化思想。很多时候我根本没看代码，通过给我讲解，后来他们自己就把代码给简化了。节省了我的脑力和视力，他们也得到了提高。

我最近一次看别人的代码是在 Intel，我们改了 PyTorch 的代码。那不是一次愉悦的经历，因为虽然很多人觉得 PyTorch 好用，它内部的代码却是晦涩而难以理解的。PyTorch 不是 Intel 自己的东西，所以没有人可以给我讲。修改 PyTorch 代码，增加新功能的时候，我发现很难从代码本身看明白应该改哪里。后来我发现，原因在于 PyTorch 的编译构架里自动生成了很多代码，导致你无法理解一些代码是怎么来的。

比如他们有好几个自己设计的文件格式，里面有一些特殊的文本，决定了如何在编译时生成代码。你得理解这些文件在说什么，而那不是任何已知的语言。这些文件被一些 Python 脚本读进去，吐出来一些奇怪的 C++，CUDA，或者 Python 代码。这其实是一种 DSL，我已经在之前的[文章](#)中解释过 DSL 带来的问题。要往 PyTorch 里面加功能，你就得理解这些脚本是如何处理这些 DSL，生成代码。而这些脚本写得也比较混乱和草率，所以就是头痛加头痛。

最后我发现，没有办法完全依靠这些代码本身来理解它。那么怎么解决这个问题呢？幸好，网络上有 PyTorch 的内部工程师写了篇 [blog](#)，解释 PyTorch 如何组织代码。Blog 的作者 E. Z. Yang 我见过一面，是在一次 PL 学术会议上。他当时在 MIT 读书，一个挺聪明的小伙子。不过看了这 blog 也只能初步知道它做了什么，应该碰大概哪些文件，而这些每天都可能变化。

这篇 blog 还提到，某几个目录里面是历史遗留代码，如果你不知道那是什么，那么请不要碰！看看那几个目录，里面都是一些利用 C 语言的宏处理生成代码的模板，而它使用 C 语言宏的方式还跟普通的用法不一样。在我看来，所谓「宏」（macro）和「元编程」（metaprogramming）本身就是巨大的误区，而 PyTorch 对宏的用法还如此奇怪，自作聪明。

你以为看了这篇 blog 就能理解 PyTorch 代码了吗？不，仍然是每天各种碰壁。大量的经验都来自折腾和碰壁。多人同时在进行这些事情，然后分享自己的经验。讨论会内容经常是：「我发现要做这个，得在这个文件里加这个，然后在那个文件里加那个…… 然后好像就行了。」下次开会又有人说：「我发现不是像你说的那样，还得改这里和那里，而那里不是关键……」许多的知其然不知其所以然，盲人摸象，因为「所以然」已经被 PyTorch 的作者们掩盖在一堆堆混乱的 DSL 下面了。

所以我从 PyTorch 的代码里面学到了什么呢？什么都没有。我只看到各种软件开发的误区在反复上演。如果他们在早期得到我的建议，根本不可能把代码组织成这种样子，不可能有这么多的宏处理，代码生成，DSL。PyTorch 之类的深度学习框架，本质上是某种简单编程语言的解释器，只不过这些语言写出来的函数可以求导而已。

很多人都不知道，有一天我用不到一百行 Scheme 代码就写出了一个「深度学习框架」，它其实是一个小的编程语言。虽然没有性能可言，没有 GPU 加速，功能也不完善，但它抓住了 PyTorch 等大型框架的本质——用这个语言写出来的函数能自动求导。这种洞察力才是最关键的东西，只要抓住了关键，细节都可以在需要的时候琢磨出来。几十行代码反复琢磨，往往能帮助你看透上百万行的项目里隐藏的秘密。

很多人以为看大型项目可以提升自己，而没有看到大型项目不过是几十行核心代码的扩展，很多部分是低水平重复。几十行平庸甚至晦涩的代码，重复一万次，就成了几十万行。看那些低水平重复的部分，是得不到什么提升的。造就我今天的编程能力和洞察力的，不是几百万行的大型项目，而是小到几行，几十行之短的练习。不要小看了这些短小的代码，它们就是编程最精髓的东西。反反复复琢磨这些短小的代码，不断改进和提炼里面的结构，磨砺自己的思维。逐渐的，你的认识水平就超越了这些几百万行，让人头痛的项目。

所以我如何阅读别人的代码呢？Don't。如果有条件，我就让代码的作者给我讲，而不是去阅读它。如果作者不合作，而我真的要使用那个项目的代码，我才会去折腾它。那么如何折腾别人的代码呢？我有另外一套办法。

我不是编译器专家

工作多年以来，我深刻体会到一个现象，那就是做过“编译器”工作的人，哪怕只做了点皮毛，都容易产生高人一等的心理，以至于在与人合作中出现各种问题。由于他们往往也存在偏执心理和理想主义，所以在恶化人际关系的同时，也可能设计出非常不合理的软件构架，浪费大量的人力物力。

我曾经提到的 [DSL](#) 例子，就是这样的两个人。他们都自称做过编译器，成天在我面前高谈阔论，甚至在最基础的概念上摆弄斧，显示出一副“教育”其他人的姿态。其实他们只有一个人做过 [parser](#)，还不算是真正的编译器工作，却总显示出高深莫测的模样。哲人一样捋捋胡子，摇摇脑袋，慢条斯理，嗯…… 另外一个完全就是外行，只是知道一些术语，成天挂在嘴边。每次他一开口，我都发现这个人并不知道自己在说什么，却仍然洋洋得意的样子。

我是被他们作为专家请来这个公司的，来了之后却发现他们最喜欢的事情，是在我面前显示他们才是“专家”。他们也问过我问题，可是每一次我都发现他们并不想知道答案，因为我说话的时候他们并没有在听。不管说什么问什么，他们似乎只想别人觉得他们是最聪明的人。

虽然对其他人趾高气昂，全懂了的样子，对于 Brendan Eich (JavaScript 语言的创造者) 这样有权势的人物，却是各种跪舔，显示出各种“贱”来。我虽然尊重 Brendan Eich 个人和他的语言，然而很明显他是半路出家，对语言设计并没有很深的造诣。对语言稍微有点研究的人，都不会对这种人物显示出谄媚的态度。

“Yin，你知道 X 吗？”当然他期望的是你说不知道，这样他就能像大师一样，把这个刚学到的术语给你讲半天。每当这个时候，我就想起一个前同事喜欢说的一句话：“你问我，是因为你不知道，还是因为你知道？”其实他问的这个概念 X，常常是我很多年前热心过，试验过，到最后发现严重问题，抛弃了的概念。

更糟的事情是，这其中一人还是 Haskell 语言的忠实粉丝，他总是有这样的雄心壮志，要用“[纯函数式编程](#)”改写全公司的代码……

遇到这样的人是非常闹心的，到了什么程度？他们经常雄心勃勃用一种新的语言（Scala，Go 之类）试图改写全公司的代码，一个月之后开始唾骂这语言，两个月之后他们的项目不了了之，代码也不知道哪里去了。然后换一种语言，如此反复……

后来实在没做出什么有用的东西，这两个人又突发奇想，开始做 [DSL](#)，闹得团队不得安宁，有点资历的工程师（包括我和一位早期 Netscape 的资深工程师）都极力反对，向大家指出更容易，更省力的解决方案。然而由于管理层根本不懂，所以任凭这两个人拍胸脯，没有困难制造困难也要上。因为烦于他们在我面前高谈阔论，而且对这个 DSL 的事情实在看不下去了，我干脆换了一个部门，不再做跟语言和编译器相关的事情。

现在这个 DSL 做了好几年了，仍然很垃圾，然而公司人傻钱多，居然请到了 Java 界的资深人物来给这 DSL 写 specification。这两人也分别升职为 Principal Engineer 和 Distinguished Engineer。当看到“Distinguished Engineer”这个 title，我觉得太好笑了。当然，我相信有资历的 PL 人都会明白这 DSL 的问题，我想象着这位 Java 人跟这两人将会发生的冲突。如果他对此没意见的话，那他的水平还真是值得怀疑了。

在 Coverity 和其它公司遇到的编译器人，基本是差不多的问题。他们下意识里把自己看成是最高档次的程序员，所以对其他人显示高高在上的气势。

Coverity 有一个 ABC 工程师，因为自己写过完整一点的静态分析，比较会折腾 C++，总是趾高气昂的对待其他人，甚至直接对别人说：“你写的这是什么代码啊？我绝对不会写出这么烂的代码！”还有一个从斯坦福编译器教授 Alex Aiken 那里毕业的 PhD，在 Coverity 做构架师，平时一行代码不写，也不看其他人写的，说不出见解深刻点的话，因为与实际工程脱节，尽在瞎指挥。地位最高的 Distinguished Engineer，成天优哉游哉，看一些关于 [parser](#) 的话题，似乎 parser 是他终身的研究方向，也不做什么实事。

我所在的每一家公司，只要工作跟编译器沾边，总是不免遇到这样的人。其它的我就不细讲了。

有些美国公司在招人的时候表示，对简历里提到“做过编译器”的求职者有戒备心理，甚至直接说“我们不招编译器专业的人”。以至于我也曾经被过滤掉，因为我做过编译器相关工作。编译器专业的人本来可以做普通的程序员工作，为什么有公司如此明确不要他们呢？我现在明白为什么了，因为自认为是“编译器专业”的人，有大概率是性格很差的团队合作者，喜欢显示出高高在上，拯救世界的姿态，无法平等而尊重的对待其他人。

有些人也把我叫做“编译器专家”，喜欢在我面前提“编译器”这个词。我一直听着别扭，却没有正式拒绝这个称呼。每遇到“真正”的编译器专家，我总觉得自己不是那个圈子的。不是我不能做编译器的工作，而是编译器领域人士的认识水平，理念和态度和我格格不入。

所以我应该明确表个态：我不是编译器专家，而且我看不起编译器这个领域。我一般不会居高临下看低其它人，然而对于认识肤浅却又自视很高的人，我确实会表示出藐视的态度。现在我的态度是针对编译器这整个领域。真的，我看这些人不顺眼很多年了。

就最后研究的领域，我是一个编程语言（PL）研究者，从更广的角度来看，我是一个计算机科学家。有人听了“科学家”一词总是误以为我在抬高自己，而在我心目中“科学家”仅仅是一个职业，就像“厨师”一样，并不说明一个人的水平和地位。PL 研究者被叫做“计算机科学家”是很恰当的，因为 PL 领域研究的其实不只是语言，而是计算的本质。通常

人公认的计算机科学鼻祖 [Alan Turing](#) 也可以算是一个 PL 研究者，虽然他认识水平比较一般。

IT 业人士经常混淆编程语言 (PL) 和编译器两个领域，而其实 PL 和编译器是很不一样的。真懂 PL 的人去做编译器也会比较顺手，而编译器专业的却不一定懂 PL。为什么呢？因为 PL 研究涵盖了计算最本质的原理，它不但能解释语言的语义，而且能解释处理器的构架和工作原理。当然它也能解释编译器是怎么回事，因为编译器只不过是把一种语言的语义，利用另外一种语言表达出来，也就是翻译一下。PL 研究所用的编程范式和技巧，很多可以用到编译器的构造中去，但却比编译器的范畴广阔很多。

深入研究过 PL 的人，能从本质上明白编译器里在做什么。所以编译器算是 PL 思想的一种应用，然而 PL 的应用却远远不止做编译器。每次有人说我是做编译器的，我都觉得是一种贬低。我只不过拿精髓的理念稍作转换和适应，做了点编译器的事情，就被人叫做“编译器专家”，而我根本不是局限在这个方向。

专门做编译器的人，一般是专注于“实现”别人已经设计好的语言，比如 C, C++。他们必须按照语言设计者写好的语言规范 (specification) 来写编译器，所以在语言方面并没有发挥的空间，没有机会去理解语言设计的微妙之处。

许多做编译器的人并不是从零开始写的，而是拿现成的编译器来修改，所以他们往往被已经存在的，具体的构架限制了想象力。极少有编译器人完整实现过一个语言，都是在已有的基础上小改一下，优化一些局部的操作。这大大限制了他们可以获得的全局洞察力。

很多编译器工程师并没有接受过系统的 PL 理论教育，有些甚至是半路出家，在学校里根本没碰过编译器，也没研究过 PL。比如我的第一个公司 Coverity，招进去的很多人从来没碰过编译器，也不懂 PL。我进去不久，Coverity 的 VP 满口牛气向新人宣布：“我们会教会你们一切！”然而很可惜，PL 的精华根本不是一个公司在短期能够传授的。Coverity 没有这个能力，Google, Facebook, Intel, 微软…… 都没有这个能力。

很多半路出家的编译器工作者以为在公司跟着做项目，折腾下 LLVM 之类，就会明白所有的原理。然而事实是很多人这样做了十几年，仍然不明白最基础的原理，因为他们被具体的实现限制了想象力。PL 理论联系着计算的本质，不明白这些原理就只能看到肤浅的表面，死记硬背，遇到新的现象就没法理解了。跟 LLVM 专家聊天，我很多时候发现他们的知识是死的，僵化在 LLVM 具体的实现里了。

由于缺乏对 PL 理论的深入研究，编译器人往往用井底之蛙的眼光来看待语言，总以为他们实现过的语言（比如 C++）就是一切。一个语言为什么那样设计？不知道。它还可以如何改进？不知道。“它就是那个样子！”这是我常听编译器人说的话。当然很多编译器人连 C++ 都没法完整实现，只是在已有基础上做了很小的改动。

许多编译器人把 C++ 的创造者 Bjarne Stroustrup 奉为神圣，却不知道 Stroustrup 在 PL 领域并不是闪耀的明星。Stroustrup 曾经在 2011 年 11 月 11 日来到 IU 进行关于 C++11 的演讲，IU 的资深 PL 教授们都有到场。Stroustrup 谦卑的说：“我需要向你们学习很多东西来改进 C++。”看似“谦虚”，其实他说的是实话，因为 IU 的教授们在语言设计上确实比他强很多。

Stroustrup 的整场演讲，我没有看到任何新颖的突破，全都是几十年早已出现，我天天都在用的东西。然而这些 C++ 的改进被编译器人看作是重大的历史性的突破，因为他们很多人根本没用过其它语言，甚至不知道它们的存在。

后来我的一个能力比较弱的 PL 同学进入了 C++ 委员会，为改进 C++ 做一些事情。从她的描述和表现，我感觉 C++ 委员会气氛十分的官僚，古板和愚钝。她进了 C++ 委员会之后，感觉整个人都傻了一样，很肤浅的小事也说得眉飞色舞，好像什么重大的突破一样。真懂 PL 的一些同学，很少有混进 C++ 委员会的，因为那意味着要利用另外的关系网，让一些自己根本看不起的人骑在自己头上，必须先帮他们做一些瞎扯淡的事情。

编译器人所膜拜的大师，在真正的 PL 研究者眼里其实不算什么。编译器人与 PL 研究者在见识上的差距是非常明显的。PL 人因为看透了很多东西，比较谦虚，往往不想揭穿编译器人的差距。但编译器人却因为在“工业界”有地位，趾高气昂以为自己懂了一切一样，结果遇到深刻点的 PL 问题就各种稀里糊涂。

实际上做编译器是很无聊的工作，大部分时候只是把别人设计的语言，翻译成另外的人设计的硬件指令。所以编译器领域处于编程语言 (PL) 和计算机体系构架 (computer architecture) 两个领域的夹缝中，上面的语言不能改，下面的指令也不能改，并没有很大的创造空间。

编译器领域几十年来翻来覆去都是那几个编程模式和技巧，玩来玩去也真够无聊的。起初觉得新鲜，熟悉了之后也就那个样了。很多程序员都懂得避免“低水平重复”，可是由于没有系统的学习过编译器，他们往往误以为做编译器是更高级，更有趣的工作，而其实编译器领域是更加容易出现低水平重复的地方，因为它的创造空间非常有限。

同样的编译优化技巧，在 A 公司拿来做 A 语言的编译器，到了 B 公司拿来做 B 语言的编译器…… 大同小异，如此反复。运气好点，你可能遇到 C, C++, Java。运气不好，你可能遇到 JavaScript, PHP, Ruby, Go 之类的怪胎，甚至某种垃圾 DSL。但公司有要求，无论语言设计如何整脚，硬件指令设计如何繁琐，你编译出来的指令必须能正确运行所有这语言写出来的代码。你说这活是不是很苦逼？

我在 Cornell 的时候，有一个很有权势的编译器教授，从未发表有理论价值的 paper，却老在 Java 上面做文章。他和他的博士生们总是把一些其它语言几十年前已经有的“新特性”搬到 Java 上面，老酒换新瓶，发 paper 拉 funding。由于拉了很多钱，所以在系里很受宠，他的学生们在其它人面前都趾高气昂的样子。

后来这教授的一个学生去了 Facebook，帮他们做 HipHop，一个从 PHP 到 C++ 的“编译器”。其实这种“源到源”编译器做起来不算难，但给 PHP 这样劣质的语言做编译器，实在是狗血的工作，繁琐而头痛。没有任何理论价值

不说，在工业界有什么价值也难说。我的一个前同事曾经对 Facebook 的这个项目发表了一个尖锐而幽默的评价：“Facebook 现在不但给母猪涂上了口红，而且真的开始 f.. 它了！”

后继的还有 PHP VM 一类的东西，越来越离谱。后来这位同学可能也受不了，换组去做其它跟语言无关的事情了。在 PL 研究者看来，VM 也并没有什么稀奇。PL 领域有各种各样的“抽象机”(abstract machine)，比如 CEK machine，它们揭示了计算的方方面面。我自己都设计实现过好几个“可逆抽象机”，它们可以进行所谓“可逆计算”。所以一个 PL 研究者很容易就能设计出一个 VM 来，它们只不过是一种经过部分优化的解释器。

每每看到编译器说到“VM”这个词的时候那种荣耀而敬畏的神情，好像只有他们明白 VM 是什么，我就觉得好笑，外加一种说不出的滋味。编译器人虽然知道一个具体的 VM 怎么实现，知道一些死板的细节和术语，却不知道 VM 的本质是什么，不知道一个全新的，具有新特性的 VM 要怎么设计出来。

在《[Chez Scheme 的传说](#)》一文中，我提到在 Cornell 的时候选过一门[编译器课程](#)，后来在半学期的时候 drop 掉了。现在回想起这段历史，发现它对“教育理念”这件事挺有启发意义。教育是什么，是为了什么？Cornell 的这门课给了我一个很好的反面教材。

这个编译器课程那一年的教授是 Tim Teitelbaum，他也是 GrammaTech 公司的创始人。GrammaTech 是与 Coverity 类似的静态分析工具，不过 GrammaTech 还能分析二进制代码。Tim Teitelbaum 是 Donald Knuth 的崇拜者，他经常提到 Knuth 提出的一些“伟大概念”，比如 attribute grammar。总是把 Knuth 那些东西说成是最伟大的发明。

这门课不知道最初是谁设计的。Andrew Myers 和 Tim Teitelbaum 以前交替着讲这个课。

那么我为什么会 drop 这门课，而且是在学校允许 drop 课程的 deadline 之后呢？因为它的教育理念非常的落后和不合理，可以说就是坑人的。

从课程的大纲你可以看出来，它是很传统的编译器课程，一开头花很多时间精力去折腾 parser。源语言是一种类似 Java 的语言，parser 是使用类似 lex, yacc 的工具生成的。这种盲目重视 parser 的误区，我已经在另外一篇[文章](#)批评过，但还这不是我鄙视的重点。

这门课最让人受不了的事情，发生在我成功完成 parser，开始编译代码的第一个 pass 之后。当得到那次作业分数的时候，我惊呆了。我从来没有得过这么差的分数！仔细看原因，说我的代码没通过好些“测试”。我到那个时候才明白，原来提交后的代码，会被助教拿来跑一些我毫不知情的测试 (test)，然后他简单的根据这些测试的结果给出分数。

作业本身的要求是用大段大段的英语写下来的。你需要按照这些英语描述从零实现编译器。真的是从零开始，没有任何的框架或者示例代码，完全从白纸开始。经过许多努力，你写出了编译器，还自己写了一些小测试，你觉得完全满足了作业的要求。可是提交之后，你的编译器代码却要被一整套你手里没有的“测试”进行检验。所以最后你惊讶的发现，自己以为做对了，而助教那里的测试有那么多没通过！

最让人无语的事情是，学生手里是没有这套测试的，而且他们不给你。也就是说，你提交作业的时候，无法用最后给你评分用的那些测试来跑你的编译器，所以你无法知道提交之后会有多少测试失败。

当我向助教和教授抗议，说这样不合理，要求得到那些测试的时候，我受到粗暴的拒绝和鄙视。那种语气，好像是在说我是一个不合格的学生，提一些无理要求。用来打分的测试怎么可能给你，你是太笨了吧？

很多其它 Cornell 学生被这样对待，可能都以为没什么，按照他们的要求做就行了，然而这是完全不合理的。按照合理的教学理念，学生应该有权得到自己学习状态的反馈。如果学生做这种编程作业，就应该能从实际的测试中得到反馈，知道自己的编译器是否符合要求。要知道，大段大段的英语描述，是很容易漏看或者误解的。只有大量的测试才能正确的抓住“要求”本身。所以不给测试，就相当于不给你准确的要求，到后来却要拿这套测试来给你打分。

课程本来应该把测试连同英语描述一起给学生，他们实现之后，自己跑通所有测试，再提交代码。这样学生就能准确的把握作业的“要求”，而不是看着那些混淆不堪的英语段落自己在那里猜。

因为这个原因，而且由于教授和助教的傲慢态度。我最终决定在课程都快进行到一半的时候 drop 这门课程。当然，要进行这个操作是需要系主任签字特许的，为此我还在系主任那里留下一笔“污点”。

在我看来，Cornell 教授们的这种做法，根本就不是合格的教育者，可以说就是在坑人，整人，害人。在他们的理念里，教育是单方面的，学生必须通过作业和考试，而教授却不需要为教学方法负责，可以随便怎么教，作业和考试想怎么整都行。

很多 Cornell 教授有类似的现象，教学不用心，光是各种拉 funding，耀武扬威，完全不顾学生死活。也许这就是为什么 Cornell 总是有学生自杀。我走了之后有一年，在一个星期之内有三个学生从学校里瀑布旁边的吊桥跳下去自杀，新闻轰动了全美国。

后来在网上看到有人骂 Cornell，说：“Cornell 想教你游泳，于是把你推进池塘里，等你扑腾上岸。等你快上来的时候，他又朝你扔一块大石头，然后继续等你游上来。等你又快上岸了，他又拿起一个榔头往你头上猛砸。这样你就可以死了，可是 Cornell 仍然在那里等着你游上岸来……”

这段话恰到好处的描述了我的在 Cornell 的经历。

转学到 IU 之后，我参加了 Kent Dybvig 的编译器课程，发现我所设想的编译器课程原来早已被他实现了，而且实现的如此友好。编译器的每一个 pass，都会把所有的“官方测试”发给学生。学生按照要求实现每个编译器 pass，在自己电脑上跑通所有测试，充分检查，然后才提交作业。而且作业的网站会自动测试你提交的代码，在提交的当时就给你反馈：“你有 N 个测试没通过，请修改后重新提交。”

这才是正确的教育方法，因为它给予学生合理的反馈，让他们清晰的知道自己的表现是否符合预期，主动进步，而不是拿一些学生事先不知道的标准在那里瞎坑人，光是给人打分。

Cornell 没有明白教育的目的是培养人，而不仅是给人发文凭。Dybvig 教授不但技术和学术水平远高于传统的编译器人，而且他的课程也设计得如此科学和友好。这才是真正的教育者。

虽然苦逼，编译器人往往自高自大，高估自己在整个 IT 领域里的地位，看低其它程序员。编译器人很多认为自己懂了编程语言的一切，而其实他们只是一知半解。

编译器领域最重要的教材，龙书和虎书，在我看来也有很多一知半解，作者自己都稀里糊涂的内容。而且花了大量篇幅讲 [parser](#) 这种看似高深，实则肤浅的话题，浪费读者太多时间，误导他们认为 parser 是至关重要的技术。以至于很多人上完编译器课程，只学会了写 parser，对真正关键的部分没能理解。龙书很难啃，为什么呢，因为作者自己都不怎么懂。虎书号称改进了龙书，结果还是很难啃，感觉只是换了一个封面而已。

我曾经跟虎书作者 Andrew Appel 的一个门徒合作过，当时这人在 IU 做助理教授。借着一次我跟她做 independent study 的机会，逼我写毫无意义的论文，而且对人非常的 push 和虚伪。作为普林斯顿大学毕业的 PhD，学识水平跟 IU 的其他教授格格不入，却在待人接物方面显示出各种“贱”，对编译器领域的“牛人”各种跪舔，随时都在显示自己以前在某某人身边工作过。那是我在 IU 度过的最难受的一个学期，这使我对“编译器人”的偏见又加深一层。

编译器领域的顶级人物如此，其它声称做过编译器的人也可想而知了。大部分自称做过编译器的人，恐怕连最基本的编译器都没法从头写出来。利用 LLVM 已有的框架做点小打小闹的优化，就号称自己做过编译器了。许多编译器人死啃书本，肤浅的记忆各种术语（比如 SSA），死记硬背具体实现细节（比如 LLVM 的 IR），看不透，无法灵活变通。

所以我常说，编译器是计算机界死知识最多，教条主义最严重的领域。经常是某人想出一个做法，起个名字，其他人就照做，死记硬背，而且把这名字叫得特别响亮。你要是一时想不起这名字是什么意思，立马被认为是法国人不知道拿破仑，中国人不知道毛泽东。你不是做编译器的！

现在因为 AI 的泡沫，很多人转向所谓“AI 框架”，“AI 编译器”。这类职位如此之多，以至于很多人根本没碰过编译器，也摇身一变成为了“深度学习编译器工程师”。

半路出家的“AI 框架工程师”和“AI 编译器工程师”们，在别人写出来的框架上小打小闹优化一下，就以为自己做的是世界上最前沿的工作，却不知道深入研究过 PL 的人其实很容易就看破了那些东西。很多 AI 框架工程师嘴里各种奇怪的术语，却看不透所谓“AI 框架”只不过是“可求导编程语言”，完全不能从高级语言和逻辑的角度去看问题。

AI 框架和编译器里面的原理和本质很容易被 PL 理论解释，PL 研究者能够为这些项目指出正确的方向，避免不必要的弯路，然而这些自诩为“编译器人”的 AI 框架工程师们完全意识不到这一点。自高自大，膜拜权威，完全没有去听 PL 研究者在说什么，甚至觉得能“教育”比自己看得透的人。

每一个大公司都要趁着 AI 这个热度做自己的“AI 框架”，“AI 编译器”，唯恐不做自己的框架，就会在业界丢面子，所以一窝蜂而上。一定要聘用名声很大的 AI 框架专家来公司站台，虽然也不知道他最后能做出什么来。所有 AI 框架和编译器都大同小异，属于无谓的重复劳动。有些人捣鼓一下这个框架，然后用同样的技巧去捣鼓另外一个，中间都是一些工程性的脏活。这种事情真是非常无聊。

AI 的热潮正在褪去，大部分 AI 公司会在一年之内失败。“AI 编译器”的工作也会濒临灭绝。所以任凭他们自己瞎蒙乱撞吧，反正坚持不了多久了。

这就是为什么虽然有多次编译器的工作机会，包括 Apple 的 LLVM 部门，我最后都没去。进入 Intel 的时候，本来编译器部门也欢迎我，可是再三考虑之后还是选择了其它方向。因为我很清楚的记得，每一次做编译器相关工作都是非常压抑的，需要面对一些沉闷古板而自以为是的人，而且内容真的是重复，无聊和枯燥。

我唯一敬佩的编译器作者是 [Kent Dybvig](#)，但我不想跟他一起做编译器。最近很多芯片公司的“AI 编译器”部门找我，我全都拒绝了。我不喜欢身边围绕着这些人，做着这些事。我宁愿去卖烧饼也不想做编译器。

由于编译器人的性格特征，除非一个公司专门要做编译器，否则对于曾经做过编译器，想换个方向的求职者，在面试的时候最好深刻了解他们的性格，态度和做事方式，看他们是否能看淡这些，能否平等对待其他人，能否理性而实在的对待工程。否则自视很高的“编译器人”进了公司，很可能对团队成为一种灾难。

我写这篇文章是为了警醒广大 IT 公司，也是为了在精神上支持其它程序员。我希望他们不要被编译器的“难度”迷惑了，不要被编译器人吓唬和打压。你们做的并不是更低级，更无聊的工作。正好相反，真正可以发挥创造力的空间并不在底层的编译器一类的东西，而在更接近应用和现实的地方。

每当有人向我表示编译器高深莫测，向往却又高攀不上，我都会给他打一个比方：做编译器就像做菜刀。你可以做出非常好的菜刀，然而你终究只是一个铁匠。铁匠不知道如何用这菜刀做出五花八门，让人心旷神怡，米其林级别的菜肴，因为那是大厨的工作。要做菜还是要打铁，那是你自己的选择，并没有贵贱之分。

自动驾驶的责任和风险分析

说到“自动驾驶”，人们最熟悉的名字恐怕是 Tesla 的 Elon Musk 先生了。他总是对 Tesla 的 Autopilot 进行各种夸大宣传，让人误解 Autopilot 的能力。Autopilot 引起车祸死了人之后，Musk 先生总是在网上发话扭曲人们的逻辑，抓住“车主没有及时接管”等各种借口，逃避对事故的责任。

很多人对他的言论感到荒谬和愤怒，却又难以说清楚他到底哪里错了，甚至政府监管机构都对各自动驾驶公司的歪理无能为力。我发现对于自动驾驶车的责任和风险问题，人们仍然缺乏一个精确的，使人信服的说法，所以我一直在思索这些问题。

在本文里，我试图使用逻辑和概率的工具来分析自动驾驶车的责任和风险问题。虽然我用的数学可能不是那么的细节到位，但是你可以从中获得分析这类问题的思路。逻辑推理和概率分析不仅可以用于科学研究，而且可以用于法律和各种社会现象。

根据对 Elon Musk 的采访，你可以看出他的言论大体包含以下内容：

- 全自动驾驶很快就要实现了，Autopilot 的视觉识别能力成指数增长，所以实现全自动驾驶就在眼前。
- 现在出产的 Tesla 车已经安装了具有“全自动驾驶能力”（FSD）的硬件。只要我们在不久的将来更新车里的软件，你就能拥有全自动驾驶的车，所以现在买 Tesla 的车是一种升值的财富，而不是贬值的物品。
- 统计数字显示，Autopilot 的事故率远远低于人类驾驶员。Autopilot 比人类驾驶员安全很多，这是不可争辩的事实。如果你否认这个事实，你就是危害公共安全。

From: Elon Musk [REDACTED]
Subject: Re: Musk Private Foundation Inquiry
Date: August 7, 2019 at 5:46 PM
To: Aaron Greenspan [REDACTED]

The data is unequivocal that Autopilot is safer than human driving by a significant margin. It is unethical and false of you to claim otherwise. In doing so, you are endangering the public.

As for the people you mention below, they have actively harassed and, in the case of Hothi, almost killed Tesla employees. What was a sideswipe when Hothi hit one of our people could easily have been a death with 6 inches of difference.

How can you possibly endorse this? You obviously couldn't care less about the truth.

虽然各种证据都说明 Autopilot 几乎没有自动驾驶能力，Elon Musk 却仍然在宣扬这些歪理。很多书呆子极客会听信他的“事故率”，为他所谓的“高科技”欢呼，甚至有人跟风说“Autopilot 比人类驾驶员安全 6 倍”。这些人都不明白，统计数字对于事故责任分析，对于 Autopilot 的风险评估都是没用的，而且他们的统计方法，解释统计数字的方式都是错误的。

在本文中我想说明以下几点：

- 大范围的统计数字对于“责任”和“风险”的分析是没有任何关系的。
- Autopilot 导致的任何一次车祸，Tesla 公司在法律上都是负有责任的。
- 要求驾驶员“随时接管”是推脱责任的手段，根本不合法理。
- 自动驾驶行业对于车祸死亡率的数据解释是片面而错误的。由车祸引起的死亡，相对其它死亡因素并不是特别严重的问题。自动驾驶技术并不能降低车祸死亡率。

责任

Elon Musk 和其他很多自动驾驶公司都喜欢拿“事故率”说事，总是说自动驾驶比人类驾驶员安全，因为统计数字显示它们的事故率低，其实那相当于在说：“我活了这么久，为这么多客户服务，没杀过其中任何一个人，我杀人的概率非常低，低于全国的谋杀犯罪率，所以我现在杀了你没什么大不了的。”

先不说 Autopilot 的事故率是否真的那么低。即使它事故率是很低，难道弄死了人就可以不负责，甚至不受谴责吗？

到底 Tesla 有没有责任，我们可以使用逻辑学的因果关系“反事实分析”（counterfactual analysis）。假设驾驶员没有使用 Autopilot 而是自己开车，那么这次事故还会不会发生？如果不会发生，那么我们得到因果关系：Autopilot 导致了事故。不管其他人用 Autopilot 有没有出事故，事故占多大比例，面对这里的因果关系都是无关紧要的。因果关系等于责任。

如果是 Autopilot 导致了事故，即使总共只发生了一次事故，都该它的设计者 Tesla 公司负责。很多人都是混淆了“责任”和“事故率”，所以才会继续支持 Elon Musk 和 Tesla 的谬论。有些人以为“自动驾驶可能会降低全国的车祸率”，从而认为 Autopilot 引起少数几次车祸问题不大，而不明白“事故率”跟“责任”和“事故再次发生的风险”，完全是两码事。

另外，如果你看透了这些吹嘘得神乎其神的“[机器的视觉能力](#)”有多假，就会知道“自动驾驶会降低车祸率”这个说法根

本就不可能实现。

为什么我强调“责任”呢？因为人如果自己开车，不小心出了车祸伤到自己，他自己是可以接受的，因为是自己的责任。然而要是 Autopilot 判断错误引起车祸，撞伤了自己，对于车主来说这就是不可接受的，必然要追究 Tesla 公司的责任。

任何人都明白这个道理吧？这就跟自己开车不小心受了伤，和出租车司机不小心导致你受伤的差别一样。你会告那个出租车司机，你却不会上法庭告自己。简单吧？

每一次 Autopilot 相关的事故，Tesla 公司都会在事后散布新闻说是驾驶员开车不认真，手没有在方向盘上准备“随时接管”，所以不是 Autopilot 的责任。驾驶员是否认真在开车，人死了无所对证，但这些全都成为了 Tesla 公司推脱责任的借口。

如果发现 Autopilot 判断失误，你真来得及接管吗，你能在那么短的时间内做出正确的反应吗？就算你双手都在方向盘上，车到了离障碍物多近的地方不减速，你才会意识到它出错了，决定接管呢？恐怕到了自己接管的时候就已经晚了。所以要求车主随时接管，根本就不是一个合理的要求，不应该作为 Tesla 免责的理由。

Autopilot 的个人风险分析

为什么每年几万起其它车祸没什么人关心，而 Autopilot 引起一两次车祸就这么新闻舆论呢？因为要是车祸是由于 Autopilot 引起的，那么同样的车祸就可能发生在所有使用 Autopilot 的 Tesla 车主身上，“Autopilot 再次发生车祸”的后验概率就会大大提高。Autopilot 导致自己伤亡的风险就很高了。

这里的核心问题就在于，到底是人开车还是 Autopilot 开车。人和软件不仅在技术能力上有很大差别，对于概率风险分析，人和软件的效果也是很不一样的。简言之，人是“独立随机变量”，而 Autopilot 不是独立变量。

每个人都是不一样的，是独立的个体。有的人开车很稳，有的人开车一般，而少数人很鲁莽。这些人之间没有必然的联系，是“独立随机变量”。什么叫“独立”呢？意思是某个人自己开车不小心出车祸，其他人并不一定会出同样的车祸，因为每个人的开车方式都不一样。在概率论里面，这些人是否出现车祸完全是独立的事件。

而 Autopilot 是一个软件系统，所有安装 Autopilot 的车都有一模一样的行为方式，所以使用 Autopilot 的许多 Tesla 车不是独立变量，而是“相关变量”，它们通过 Autopilot 系统的设计关联在了一起。如果 Autopilot 因为判断错误导致一次车祸，那么所有使用 Autopilot 的车都很可能发生同样的车祸。

相应的随机变量是否“独立”，导致了人类驾驶员与 Autopilot 出现一次事故的风险分析完全不一样。

如果你学过概率论，那么 Autopilot 车主出事的“后验概率”（[posterior probability](#)）会因为“Autopilot 引起一次车祸”的发生而大幅度提高，而如果是人开的汽车，那么它的后验概率基本不会因为另外一辆同型号车出事而提高。写成数学公式就是：

$$P(\text{其它 Autopilot 出车祸} \mid \text{Autopilot 引起一次车祸})$$

远大于

$$P(\text{其它非自动驾驶出车祸} \mid \text{一辆非自动驾驶出车祸})$$

事故起因的随机性不同，后验概率也就随之不同。

面对“Autopilot 有一定概率会要了你的命”这一事实，不管 Autopilot 的总体事故率有多低，甚至像 Elon Musk 说的低于全国车祸率，对于 Tesla 车主来说都是毫无意义的。一是因为“责任”：车主可以允许自己要了自己的命，却不允许 Autopilot 或者其他人要了自己的命，更不允许是因为别人（Autopilot）的愚蠢而要了自己的命。二是因为“个人风险”：不管全国的事故率是多少，自己开车的风险一般只跟自己开车的小心程度有关，也就是说自己开车出事的概率基本是独立于全国事故率的。而使用 Autopilot，自己的风险就受到 Autopilot 能力的影响，跟 Autopilot 的平均事故率差不多了。

仔细看看统计数字

Autopilot 的事故率真的低吗？你可以自己研究一下。如果你算对了数学，恐怕它的事故率并不低。举一个例子，普通人只计算了事故的数目与 Autopilot 导航的总里程的比例，却忽视了那些由于驾驶员及时接管而避免了的事故的数目。

Autopilot 能不受打断的连续驾驶多少里程呢？按照现有的视觉技术，恐怕不会很远。聪明点的人都不会让 Autopilot 进入稍微复杂的局面，只用它进行“高速车道控制”，所以 Autopilot 事故率比较低的原因，很可能是因为大部分用户根本不在复杂的情况下使用它。所以虽然 Autopilot 统计数据看起来是“几十亿英里”，恐怕它从来没有在复杂的情况下做出过正确的反应。

另外 Tesla 属于比较贵的车，买车的人属于对自己比较负责的人，所以事故率不应该跟所有车比，而应该跟同样年代的奔驰，保时捷之类的车比。

我们来仔细看看汽车业的总体统计数字吧。美国 [2017 年车祸死亡人数](#)是 3.7 万人。看上去很多，可是按里程数的死亡率，每一亿英里平均只有 1.16 人。从 1975 年到 2017 年，每一亿英里死亡人数从 3.35 人降低到了 1.16 人，所以即使没有 Autopilot，开车也是越来越安全了。



对比一下[其它死因](#)吧。美国 2017 年总共死亡 281 万人，其中因心脏病死亡 64.7 万人，癌症 59.9 万，呼吸道疾病 16 万，中风 14.6 万，意外伤害死亡 16.9 万（包括车祸），糖尿病 8.3 万，流感 5.5 万，自杀 4.7 万。

意外伤害死亡的 16.9 万里面包括了车祸的 3.7 万，所以另外 13.2 万人死于其它的意外。连自杀都有 4.7 万人。所以你可能意识到了，车祸死亡 3.7 万人并不是一个那么可怕的数字，而是相对来说最安全的领域之一了。

Deaths and Mortality

Data are for the U.S.

- Number of deaths: 2,813,503
- Death rate: 863.8 deaths per 100,000 population
- Life expectancy: 78.6 years
- Infant Mortality rate: 5.79 deaths per 1,000 live births

Source: [Deaths: Final Data for 2017, tables 1, 3, 13](#) [PDF – 2 MB]

Number of deaths for leading causes of death:

- Heart disease: 647,457
- Cancer: 599,108
- Accidents (unintentional injuries): 169,936
- Chronic lower respiratory diseases: 160,201
- Stroke (cerebrovascular diseases): 146,383
- Alzheimer's disease: 121,404
- Diabetes: 83,564
- Influenza and Pneumonia: 55,672
- Nephritis, nephrotic syndrome and nephrosis: 50,633
- Intentional self-harm (suicide): 47,173

Source: [Deaths: Final Data for 2017, table B](#) [PDF – 2 MB]

你知道车祸死掉的都是什么人吗？他们是怎么开的车？只要自己小心开车，我不觉得自己的风险会有那么高，可能比自杀的概率都要小。

Elon Musk 在[采访](#)中把汽车叫做“two-ton death machine”（两吨重的死亡机器），甚至说“难以置信我们居然允许人开车”，根本就是危言耸听。盲目的强调车祸死亡人数，号称可以降低事故率，就是自动驾驶领域常见的幌子。他们解决的并不是一个那么重要的问题，而且解决的方法根本就是[不切实际的忽悠](#)。



所以 Tesla 不但在技术上无法实现自动驾驶，而且人品和诚信都很成问题。我还没有见过一个汽车公司如此急于推脱责任的，一般都是积极配合调查，勇于承担责任，及时整改，这样才可能得到公众的信任。

机器与人类视觉能力的差距 (3)

本文属于个人观点，跟本人在职公司的立场无关。由于最近 GitHub 服务器在国内访问速度严重变慢，虽然经过大幅度压缩尺寸，文中的图片仍然可能需要比较长时间才能加载。这篇文章揭示了 AI 领域重要的谬误和不实宣传，为了阻止愚昧的蔓延，我鼓励大家转发这篇文章和它的后续，转发时只需要注明作者和出处就行。

这是这个系列文章的第三集，在这一集中，我想讲讲 AI 领域所谓的“超人类识别率”是怎么来的，以及由于对机器视觉的盲目信任所导致的灾难性后果。

“超人类准确率”的迷雾

我发现神经网络在测试数据的可靠性，准确率的计算方法上，都有严重的问题。

神经网络进行图像识别，所谓“准确率”并不是通过实际数据测出来的，而是早就存在那里的，专用的测试数据。比如 ImageNet 里面有 120 万张图片，是从 Flickr 等照片网站下载过来的。反反复复都是那些，所以实际的准确率和识别效果值得怀疑。数据全都是网络上的照片，但网络上数据肯定是不全面的，拍照的角度和光线都无法概括现实的多样性。而且不管是训练还是测试的数据，他们选择的都是在理想环境下的照片，没有考虑各种自然现象：反光，折射，阴影等。

比如下图就是图像识别常用的 ImageNet 和其它几个数据集的一小部分。你可以看到它们几乎全都是光线充足情况下拍的照片，训练和测试用的都是这样的照片，所以遇到现实的场景，光线不充足或者有阴影，准确率很可能就没有 paper 上那么高了。



如此衡量“准确率”，有点像你做个编译器，却只针对很小一个 benchmark 进行优化跑分。一旦遇到实际的代码，别人可能就发现性能不行。但神经网络训练需要的硬件等条件比较昂贵，一般人可能也很少有机会进行完整的模型训练和实际的测试，所以大家只有任凭业内人士说“超人类准确率”，却无法验证它的实际效果。

“Top-5 准确率”的骗局

不但测试数据的“通用性”值得怀疑，所谓“准确率”的计算标准也来的蹊跷。AI 领域向公众宣扬神经网络准确率的时候，总喜欢暗地里使用所谓“top-5 准确率”，也就是说每张图片给 5 次机会分类，只要其中一个对了就算正确，然后计算准确率。依据 top-5 准确率，他们得出的结论是，某些神经网络模型识别图像的准确率已经“超越了人类”。

ResNet(2015)

At last, at the ILSVRC 2015, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced a novel architecture with "skip connections" and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs. Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.

如果他们提到“top-5”还算好的了，大部分时候他们只说“准确率”，而不提“top-5”几个字。在跟人比较的时候，总是

说“超越了人类”，而绝口不提“top-5”，不解释是按照什么标准。我为什么对 top-5 有如此强烈的异议呢？现在我来解释一下。

具体一点，“top-5”是什么意思呢？也就是说对于一张图片，你可以给出 5 个可能的分类，只要其中一个对了就算分类正确。比如图片上本来是汽车，我看到图片，说：

1. “那是苹果？”
2. “哦不对，是杯子？”
3. “还是不对，那是马？”
4. “还是不对，所以是手机？”
5. “居然还是不对，那我最后猜它是汽车！”

五次机会，我说出 5 个风马不及的词，其中一个对了，所以算我分类正确。荒谬吧？这样继续，给很多图片分类，然后统计你的“正确率”。

为什么要给 5 次机会呢？ImageNet 比赛 ([ILSVRC](#)) 对两种不同的比赛给出了两种不太一样的说法。一种说是为了让机器可以识别出图片上的多个物体，而不因为其中某个识别出的物体不是正确标签 (ground truth) 而被算作错误。另外一种说是为了避免输出意义相同的近义词，却不能完全匹配标签而被算作错误。

两个说法的理由不同，但数学定义基本是一样的。总之就是有五次机会，只要对了一个就算你对。

II: Object localization

The data for the classification and localization tasks will remain unchanged from ILSVRC 2012. The validation and test data will consist of 150,000 photographs, collected from flickr and other search engines, hand labeled with the presence or absence of 1000 object categories. The 1000 object categories contain both internal nodes and leaf nodes of ImageNet, but do not overlap with each other. A random subset of 50,000 of the images with labels will be released as validation data included in the development kit along with a list of the 1000 categories. The remaining images will be used for evaluation and will be released without labels at test time. The training data, the subset of ImageNet containing the 1000 categories and 1.2 million images, will be packaged for easy downloading. The validation and test data for this competition are not contained in the ImageNet training data.

In this task, given an image an algorithm will produce 5 class labels $c_i, i = 1, \dots, 5$ in decreasing order of confidence and 5 bounding boxes $b_i, i = 1, \dots, 5$, one for each class label. The quality of a localization labeling will be evaluated based on the label that best matches the ground truth label for the image and also the bounding box that overlaps with the ground truth. The idea is to allow an algorithm to identify multiple objects in an image and not be penalized if one of the objects identified was in fact present, but not included in the ground truth.

The ground truth labels for the image are $C_k, k = 1, \dots, n$ with n class labels. For each ground truth class label C_k , the ground truth bounding boxes are $B_{km}, m = 1 \dots M_k$, where M_k is the number of instances of the k^{th} object in the current image.

Let $d(c_i, C_k) = 0$ if $c_i = C_k$ and 1 otherwise. Let $f(b_i, B_k) = 0$ if b_i and B_k have more than 50% overlap, and 1 otherwise. The error of the algorithm on an individual image will be computed using:

$$e = \frac{1}{n} \cdot \sum_k \min_i \min_m \max\{d(c_i, C_k), f(b_i, B_{km})\}$$

The winner of the object localization challenge will be the team which achieves the minimum average error across all test images.

For each image, algorithms will produce a list of at most 5 scene categories in descending order of confidence. The quality of a labeling will be evaluated based on the label that best matches the ground truth label for the image. The idea is to allow an algorithm to identify multiple scene categories in an image given that many environments have multi-labels (e.g. a bar can also be a restaurant) and that humans often describe a place using different words (e.g. forest path, forest, woods).

For each image, an algorithm will produce 5 labels $l_j, j = 1, \dots, 5$. The ground truth labels for the image are $g_k, k = 1, \dots, n$ with n classes of scenes labeled. The error of the algorithm for that image would be

$$e = \frac{1}{n} \cdot \sum_k \min_j d(l_j, g_k).$$

$d(x, y) = 0$ if $x = y$ and 1 otherwise. The overall error score for an algorithm is the average error over all test images. Note that for this version of the competition, $n=1$, that is, one ground truth label per image.

看似合理？然而这却是模糊而错误的标准。这使得神经网络可以给出像上面那样风马不及的 5 个标签（苹果，杯子，马，手机，汽车），其中前四个都不是图片上的物体，却仍然被判为正确。

你可能觉得我的例子太夸张了，但是准确率计算标准不应该含有这样的漏洞。只要标准有漏洞，肯定会有错误的情况会被放过。现在我们来看一个实际点的例子。

ILSVRC ImageNet Challenge

ILSVRC top-5 Example

- ILSVRC Object localization challenge (top-5) example

- Top-5 selections for each image listed with probability histograms

lens cap	abacus	slug	hen
reflex camera			hen
polaroid camera	abacus	slug	cock
pencil sharpener	typewriter keyboard	zucchini	cocker spaniel
switch	space bar	ground beetle	Partridge
combination lock	computer keyboard	common newt	English setter
	accordion	water snake	
tiger	chambered nautilus	tape player	planetarium
tiger	lampshade	cellular telephone	planetarium
tiger cat	throne	slot	dome
tabby	goblet	reflex camera	mosque
boxer	table lamp	dial telephone	radio telescope
Saint Bernard	hamper	iPod	steel arch bridge

上图是一个 Coursera 的[机器学习课程](#)给出的 top-5 实际输出结果的例子。你可以从中发现，纵然有一些 top-5 输出标签是近义词，可是也有很多并不是近义词，而是根本错误的标签。比如“算盘”图片的 top-5 里面包含了 computer keyboard (电脑键盘) 和 accordion (手风琴)。“老虎”图片的 top-5 里面包含了两种狗的品种名字 (boxer , Saint Bernard) 。

另外你还可以看到，测试图片是经过精心挑选和裁剪的，里面很少有多于一个物体。所以第一种说法，“可能输出某个图片上存在的物体但却不是正确答案”，恐怕是很少见的。

所以 ILSVRC 对使用 top-5 给出的两个理由是站不住脚的。它想要解决的问题并不是那么突出地存在，但是它却开了一道后门，可能放过很多的错误情况。比如上面的“算盘”图片，如果排名第一的不是 abacus，而是 computer keyboard (电脑键盘) 或者 accordion (手风琴) ，只要 abacus 出现在 top-5 列表里，这个图也算识别正确。所以 top-5 根本就是错误的标准。

其实要解决图片上有多个物体的问题，或者输出是近义词的问题，都有更好的办法，而不会让错误的结果被算成正确的。每一个学过基础数据结构和算法的本科生都应该能想出更好的解决方案。比如你可以用一个近义词词典，只要输出的标签和“正确标签”是近义词就算正确。对于有多个物体的图片，你可以在标注时给它多个标签，算法给出的标签如果在这个“正确标签集合”里面就算正确。

但 ILSVRC 并没有采用这些解决方案，而是采用了 top-5。这么基础而重要的问题，AI 业界的解决方案如此幼稚，却被全世界研究者广泛接受。你们不觉得蹊跷吗？我觉得他们有自己的目的：top-5 使得神经网络的准确率显得很高，只有使用这个标准，神经网络才会看起来“超越了人类”。

Top-5 准确率总是比 top-1 高很多。高多少呢？比如 ResNet-152 的 top-1 错误率是 19.38%，而 top-5 错误率却只有 4.49%。Top-1 准确率只能算“勉强能用”，换成 top-5 之后，忽然就可以宣称“超越人类”了，因为据说人类的 top-5 错误率大概是 5.1%。

Table 3. Error rates (%), **10-crop** testing) on ImageNet validation.
VGG-16 is based on our test. ResNet-50/101/152 are of option B
that only uses projections for increasing dimensions.

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43 [†]
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	19.38	4.49

Top-5 准确率对人是不公平的

可能很多人还没意识到，top-5 比较方法对人是不公平的。图片上要是人见过的物体，几乎总是一次就能做对，根本不需要 5 次机会。使用“top-5 准确率”，就像考试的时候给差等生和优等生各自 5 次机会来做对题目。当然，这样你就分不清谁是差等生，谁是优等生了。“top-5 准确率”大大的模糊了好与坏之间的界线，最后看起来都差不多了，甚至差等生显得比优等生还要好。

具体一点。假设一个人识别那些图片的时候，他的 top-5 错误率是 5.1%（就像他们给出的数字那样），那么他的 top-1 错误率大概也是 5.1%。因为人要是一次机会做不对，那他可能根本就没见过图片上的物体。如果他一次做不对，你给他 5 次机会，他也做不对，因为他根本就不知道那东西叫什么名字。

现在某个神经网络（ResNet-152）的 top-5 错误率是 4.49%，它的 top-1 错误率是 19.38%。你却只根据 top-5 得出结论，说神经网络超越了人类。是不是很荒谬？

退一万步讲，就算你可以用 top-5，像这种 4.49% 与 5.1% 的差别，只相差 0.61%，也应该是忽略不计的。因为实验都是有误差，有随机性的，根据测试数据的不同也有差异，像这样的实验，1% 以内的差别根本不能说明问题。如果你仔细观察各个文献列出来识别率，就会发现它们列出的数字都不大一样。同样的模型，准确率差距可以有 3% 以上。但他们拿神经网络跟人比，却总是拿神经网络最好的那个数，跟人死扣那百分之零点几的“优势”，然后欢天喜地宣称已经“超人类”了。

而且他们真的拿人做过公平的实验吗？为什么从来没有发布过“神经网络 vs 人类 top-1 对比结果”呢？5.1% 的“人类 top-5 准确率”数字是哪里来的呢？哪些人参加了这个测试，他们都是什么人？我唯一看到对人类表现的描述，是在 Andrej Karpathy 的主页上。他拿 ImageNet 测试了自己的识别准确率，发现好多东西根本没见过，不认识，所以他又看 ImageNet 的图片“训练”自己，再次进行测试，结果准确率大大提高。

就那么一个人得出的“准确率”，就能代表全人类吗？而且你们知道 Andrej Karpathy 是谁吧。他是李飞飞的学生，目前是 Tesla 的 AI 主管，而李飞飞是 ImageNet 的发起者和创造者。让一个“内幕人士”拿自己来测试，这不像是公正和科学的实验方法。你见过有医学家，心理学家拿自己做个实验，就发表结果的吗？第一，人数太少，至少应该有几十个智商正常的人来做这个，然后数据平均一下吧？第二，这个人是个内幕人士，他的表现恐怕不具有客观性。

别误会了，我并不否认 Andrej Karpathy 是个很聪明，说话挺耿直的人。我很欣赏他讲的斯坦福 cs231n 课程，通过他的讲述我第一次明白了神经网络到底是什么，明白了 back-propagation 到底如何工作。我也感谢李飞飞准备了这门课，并且把它无私地放在网上。但是这么一个领域，这么多人，要提出“超越了人类视觉”这么一个口号，居然只有研究者自己一个人挺身而出做了实验，你不觉得这有点不负责任吗？

AI 领域对神经网络训练进行各种优化，甚至专门针对 top-5 进行优化，把机器的每一点性能每一点精度都想榨干了去，对于如何让人准确显示自己的识别能力，却漫不经心，没有组织过可靠的实验，准确率数字都不知道是怎么来的。对比一下生物，神经科学，医学，这些领域是如何拿人做实验，如何向大家汇报结果，AI 领域的做法像是科学的吗？

这就是“AI 图像识别超越人类”这种说法来的来源。AI 业界所谓“超人类的识别率”，“90+ % 的准确率”，全都是用“top-5 准确率”为标准的，而且用来比较的人类识别率的数字没有可靠的来源。等你用“top-1 准确率”或者更加公平的标准，使用客观公正抽选的人类实验者的时候，恐怕就会发现机器的准确率远远不如人类。

尴尬的 top-1 准确率

Table 3. Error rates (%, **10-crop** testing) on ImageNet validation.
 VGG-16 is based on our test. ResNet-50/101/152 are of option B
 that only uses projections for increasing dimensions.

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43 [†]
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	19.38	4.49

Table 1. Single crop validation error on ImageNet-1k (center 224x224 crop from resized image with shorter side = 256).
 The SENet-154 is one of our superior models used in [ILSVRC 2017 Image Classification Challenge](#) where we won the 1st place (Team name: [WMW](#)).

Model	Top-1	Top-5	Size	Caffe Model	Caffe Model
SE-BN-Inception	23.62	7.04	46 M	GoogleDrive	BaiduYun
SE-ResNet-50	22.37	6.36	107 M	GoogleDrive	BaiduYun
SE-ResNet-101	21.75	5.72	189 M	GoogleDrive	BaiduYun
SE-ResNet-152	21.34	5.54	256 M	GoogleDrive	BaiduYun
SE-ResNeXt-50 (32 x 4d)	20.97	5.54	105 M	GoogleDrive	BaiduYun
SE-ResNeXt-101 (32 x 4d)	19.81	4.96	187 M	GoogleDrive	BaiduYun
SENet-154	18.68	4.47	440 M	GoogleDrive	BaiduYun

我们来看看 top-1 准确率吧。业界最先进的模型之一 ResNet-152 的 top-1 错误率是 19.38%。2017 年的 ImageNet 分类冠军 [SENet-154](#)，top-1 错误率是 18.68%。当然这也没有考虑过任何实际的光线，阴影和扭曲问题，只是拿标准的，理想情况的 ImageNet “测试图片”来进行。遇到实际的情况，准确率肯定会更低。

神经网络要想提高 top-1 准确率已经非常困难了，都在 80% 左右徘徊。有些算法工程师告诉我，识别率好像已经到了瓶颈，扩大模型的规模才能提高一点点。可是更大的模型具有更多的参数，也就需要更大规模的计算能力来训练。比如 SENet-154 尺寸是 ResNet-152 的 1.7 倍，ResNet-152 尺寸又是 ResNet-50 的 2.4 倍，top-1 准确率才提高一点点。

我还有一个有趣的发现。如果你算一下 ResNet-50 和 ResNet-152 的差距，就会发现 ResNet-152 虽然模型大小是 ResNet-50 的 2.4 倍，它的 top-1 错误率绝对值却只降低了 1.03%。从 22.37% 降低到 21.34%，相对降低了 $(22.37-21.24)/22.37 = 4.6\%$ ，很少。可是如果你看它的 top-5 错误率，就会觉得它好了不少，因为它从 6.36% 降低到了 5.54%，虽然绝对值只少了 0.82%，比 top-1 错误率的改进还小，可是相对值却降低了 $(6.36-5.54)/6.36 = 12.9\%$ ，就显得改进了挺多。

这也许就是为什么 AI 业界用 top-5 的第二个原因。因为它的错误率基数很小，所以你减小一点点，相对的“改进”就显得很多了。然后你看历年对 top-5 的改进，就觉得神经网络识别率取得了长足的进步！



而如果你看 top-1 准确率，就会觉得几乎没有变化。模型虽然大了几倍，计算量大了那么多，top-1 准确率却几乎没有变。所以神经网络的 top-1 准确率似乎确实到了一个瓶颈，如果没有本质的突破，恐怕再大的模型也难以超越人类。

AI 业界的诚信问题和自动驾驶的闹剧

准确率不够高，不如人类其实问题不大，只要你承认它的局限性，把它用到能用的地方就行了。可是最严重的问题是人的诚信，AI 人士总是夸大图像识别的效果，把它推向超出自己能力的应用。

AI 业界从来没有向公众说清楚他们所谓的“超人类识别率”是基于什么标准，反而在各种媒体宣称“AI 已经超越了人类视觉”。这完全是在欺骗和误导公众。上面 Geoffrey Hinton 的[采访视频](#)中，主持人也提到“神经网络视觉超越了人类”，这位深度学习的先驱者对此没有任何说明，而是欣然接受，继续自豪地夸夸其谈。

你可以给自动驾驶车 5 次机会来判断前面出现的是什么物体吗？你有几条命可以给它试验呢？Tesla 的 Autopilot 系统可能 top-5 正确率很高吧：“那是个白板…… 哟不对，那是辆[卡车](#)！” “那是块面包…… 哟不对，那是高速公路的[隔离带](#)！”

我不是开玩笑，你点击上面的“卡车”和“隔离带”两个链接，它们指向的是 Tesla Autopilot 引起的两次致命车祸。第一次车祸，Autopilot 把卡车识别为白板，直接从侧面撞上去，导致车主立即死亡。另一次，它开出车道，没能识别出高速公路中间的隔离带，完全没有减速，反而加速撞上去，导致车主死亡，并且着火爆炸。



神经网络能把卡车识别为白板还算“top-5 分类正确”，Autopilot 根本没有视觉理解能力，这就是为什么会引起这样

可怕的故事。



你可以在这里看到一个 [Autopilot 导致的事故列表](#)。

出了挺多人命，可是“自动驾驶”的研究仍然在混沌中进行。2018 年 3 月，Uber 的自动驾驶车在亚利桑那州撞死一名推自行车过马路的女性。事故发生时的[车载录像](#)已经被公布到了网上。

报告显示，Uber 的自动驾驶系统在出事前 6 秒钟检测到了这位女士，起初把她分类为“不明物体”，然后分类为“汽车”，最后分类为“自行车”，完全没有刹车，以每小时 40 英里的速度直接撞了上去…… [【新闻链接】](#)

在此之前，Uber 被加州政府吊销了自动驾驶实验执照，后来他们转向了亚利桑那州，因为亚利桑那州长热情地给放宽政策，“拥抱高科技创新”。结果呢，搞出人命来了。美国人看到 Uber 自动车撞死人，都在评论说，要实验自动驾驶车就去亚利桑那州吧，因为那里的人命不值钱，撞死不用负责！

据 2018 年 12 月[消息](#)，Uber 想要重新开始自动驾驶实验，这次是在宾夕法尼亚州的匹兹堡。他们想要在匹兹堡的闹市区进行自动驾驶实验，因为那里有狭窄的街道，列车铁轨，许多的行人…… 我觉得要是他们真去那里实验，可能有更好的戏看了。

自动驾驶领域使用的视觉技术是根本不可靠的，给其它驾驶者和行人造成生命威胁，各个自动驾驶公司却吵着想让政府交通部门给他们大开绿灯。某些公司被美国政府拒绝批准牌照之后大吵大闹，骂政府监管部门不懂他们的“高科技”，太保守，跟不上时代。有的公司更是异想天开，想要政府批准他们的自动车上[不安装方向盘](#)，油门和刹车，号称自己的车已经不需要人类驾驶员，甚至说“只有完全去掉了人类的控制，自动车才能安全运行。”

The letter is in response to a request for
public comment by the NHTSA to a proposal
it made last May to [amend the Federal Motor
Vehicle Safety Standards](#), a list of 75 rules that automakers must follow before selling cars to
customers. Currently, those rules state that cars need to have controls such as a steering
wheel and pedals.

BE QUICK ABOUT IT

But self-driving cars may not need these controls, proponents say, and the rules could be a
hindrance to the technology being widely released at scale. Waymo and others like Cruise,
the self-driving division of GM, and Ford hope to inevitably release tens of thousands of
driverless cars without any human controls. Only by cutting the human completely out of the
equation can an autonomous vehicle operate safely, these companies argue. And the
NHTSA is considering rewriting the rules so self-driving car companies like Waymo can
release cars without those features.

Waymo's letter is full of language like "promptly," "should move rapidly," and "urges NHTSA
not to await" the completion of other third-party research into autonomous technology. The
message it sends is one of urgency: the government needs to drop everything and change
the damn rules already.

一出的闹剧上演，演得好像自动驾驶就快实现了，大家都在拼命抢夺这个市场似的，催促政府放宽政策。很是有些我们当年大炼钢铁，超英赶美的架势。这些公司就跟小孩子耍脾气要买玩具一样，全都吵着要爸妈让他玩自动驾驶，各种蛮横要求，马上给我，不然你就是不懂高科技，你就是“反智”，“反 AI”，你就是阻碍历史进步！给监管机构扣各种帽子，却完全不理解里面的难度，伦理和责任。玩死了人，却又给出各种借口，不想负责任。

虽然 Tesla 和 Uber 是应该被谴责的，但这里面的视觉问题不只是这两家公司的的问题，整个自动驾驶的领域都建立在虚浮的基础上。我们应该清楚地认识到，现有的所谓 AI 根本没有像人类一样的视觉理解能力，它们只是非常粗糙的图像识别，识别率还远远达不到人类的水平，所以根本就不可能实现自动驾驶。

什么 L1~L4 的自动驾驶分级，都是瞎扯。根本没法实现的东西，分了级又有什么用呢？只是拿给这些公司用来忽悠大家的口号，外加推脱责任的借口而已。出事故前拿来做宣传：“我们已经实现 L2 自动驾驶，目前在研究 L3 自动驾驶，成功之后我们向 L4 进军！”出事故后拿来推脱责任：“我们只是 L2 自动驾驶，所以这次事故是理所当然，不可避免的！”

如果没有视觉理解，依赖于图像识别技术的“自动驾驶车”，是不可能在复杂的情况下做出正确操作，保障人们安全的。机器人等一系列技术，也只能停留在固定场景，精确定位的“工业机器人”阶段，而不能在复杂的自然环境中行

动。

识别技术还是有意义的

要实现真正的语言理解和视觉理解是非常困难的，可以说是毫无头绪。一代又一代的神经学家，认知科学家，哲学家，为了弄明白人类“认知”和“理解”到底是怎么回事，已经付出了许多的努力。可是直到现在，对于人类认知和理解的认识都不足以让机器具有真正的理解能力。

真正的 AI 其实没有起步，很多跟 AI 沾点边的人都忙着忽悠和布道，没人关心其中的本质，又何谈实现呢？除非真正有人关心到问题所在，去研究本质的问题，否则实现真的理解能力就只是空中楼阁。我只是提醒大家不要盲目乐观，不要被忽悠了。与其夸大其词，欺骗大众，说人工智能快要实现了，不如拿已有的识别技术来做一些有用的事情，诚实地面对这些严重的局限性。

我并不是一味否定识别技术，我只是反对把“识别”夸大为“理解”，把它等同于“智能”，进行不实宣传，用于超出它能力的领域。诚实地使用识别技术还是有用的，而且蛮有趣。我们可以用这些东西来做一些很有用的工具，辅助我们进行一些事情。从语音识别，语音合成，图片搜索，内容推荐，商业金融数据分析，反洗钱，公安侦查，医学图像分析，疾病预测，网络攻击监测，各种娱乐性质的 app…… 它确实可以给我们带来挺多好处，实现我们以前做不到的一些事情。

另外虽然各公司都在对他们的“AI 对话系统”进行夸大和不实宣传，可是如果我们放弃“真正的对话”，坦诚地承认它们并不是真正的在对话，并没有智能，那它们确实可以给人带来一些便利。现有的所谓对话系统，比如 Siri，Alexa，基本可以被看作是语音控制的命令行工具。你说一句话，机器就挑出其中的关键字，执行一条命令。这虽然不是有意义的对话，却可以提供一些方便。特别是在开车不方便看屏幕的时候，语音控制“下一首歌”，“空调风量小一点”，“导航到最近的加油站”之类的命令，还是有用的。

但不要忘记，识别技术不是真的智能，它没有理解能力，不能用在自动驾驶，自动客服，送外卖，保洁阿姨，厨师，发型师，运动员等需要真正“视觉理解”或者“语言理解”能力的领域，更不能期望它们取代教师，程序员，科学家等需要高级知识的工作。机器也没有感情和创造力，不能取代艺术家，作家，电影导演。所有跟你说机器也能有“感情”或者“创造力”的都是忽悠，就像现在的对话系统一样，只是让人以为它们有那些功能，而其实根本就没有。

你也许会发现，机器学习很适合用来做那些不直观，人看不透，或者看起来很累的领域，比如各种数据分析。实际上那些就是统计学一直以来想解决的问题。可是视觉这种人类和高等动物的日常功能，机器的确非常难以超越。如果机器学习领域放弃对“人类级别智能”的盲目追求，停止拿“超人类视觉”一类的幌子来愚弄大众，各种夸大，那么他们应该能在很多方向做出积极的贡献。

(全文完)

机器与人类视觉能力的差距 (2)

本文属于个人观点，跟本人在职公司的立场无关。由于最近 GitHub 服务器在国内访问速度严重变慢，虽然经过大幅度压缩尺寸，文中的图片仍然可能需要比较长时间才能加载。这篇文章揭示了 AI 领域重要的谬误和不实宣传，为了阻止愚昧的蔓延，我鼓励大家转发这篇文章和它的后续，转发时只需要注明作者和出处就行。

这是这个系列文章的第二集，在这一集中，我想详细分析一下 AI 领域到底理解多少人类神经系统的构造。

神经网络为什么容易被欺骗

“神经网络”与人类神经系统的关系是很肤浅的。等你理解了所谓“神经网络”，就会明白它跟神经系统几乎没有一点关系。“神经网络”只是一个误导性质的 marketing 名词，它出现的目的只是为了让外行产生不明觉厉的效果，以为它跟人类神经系统有相似之处，从而对所谓的“人工智能”信以为真。

其实所谓“神经网络”应该被叫做“可求导编程”。说穿了，所谓“神经网络”，“机器学习”，“深度学习”，就是利用微积分，梯度下降法，用大量数据拟合出一个函数，所以它只能做拟合函数能做的那些事情。

用了千万张图片和几个星期的计算，拟合出来的函数也不是那么可靠。人们已经发现用一些办法生成奇怪的图片，能让最先进的深度神经网络输出完全错误的结果。



(图片来源：<http://www.evolvingai.org/fooling>)

神经网络为什么会有这种缺陷呢？因为它只是拟合了一个“像素=>名字”的函数。这函数碰巧能区分训练集里的图片，却不能抓住物体的结构和本质。它只是像素级别的拟合，所以这里面有很多空子可以钻。

深度神经网络经常因为一些像素，颜色，纹理匹配了物体的一部分，就认为图片上有这个物体。它无法像人类一样理解物体的结构和拓扑关系，所以才会被像素级别的肤浅假象所欺骗。

比如下面两个奇怪的图片，被认为是一个菠萝蜜和一个遥控器，仅仅因为它们中间出现了相似的纹理。



另外，神经网络还无法区分位置关系，所以它会把一些位置错乱的图片也识别成某种物体。比如下面这个，被认为是一张人脸，却没发现五官都错位了。



神经网络为什么会犯这种错误呢？因为它的目标只是把训练集里的图片正确分类，提高“识别率”。至于怎么分类，它可以是毫无原则的，它完全不理解物体的结构。它并没有看到“叶子”，“果皮”，“方盒子”，“按钮”，它看到的只是一堆像素纹理。因为训练集里面的图片，出现了类似纹理的都被标记为“菠萝蜜”和“遥控器”，没有出现这纹理的都被标记为其它物品。所以神经网络找到了区分它们的“分界点”，认为看到这样的纹理，就一定是菠萝蜜和遥控器。

我试图从神经网络的本质，从统计学来解释这个问题。神经网络其实是拟合一个函数，试图把标签不同的样本分开。拟合出来的函数试图接近一个“真实分界线”。所谓“真实分界线”，是一个完全不会错的函数，也就是“现实”。

数据量小的时候，函数特别粗糙。数据量大了，就逐渐逼近真实分界线。但不管数据量如何大，它都不可能得到完全准确的“解析解”，不可能正好抓住“现实”。



除非现实函数特别简单，运气特别好，否则用数据拟合出来的函数，都会有很多小“缝隙”。以上的像素攻击方法，就是找到真实分界线附近，“缝隙”里面的样本，它们正好让拟合函数出现分类错误。

人的视觉系统是完全不同的，人直接就看到了事物是什么，看到了“解析解”，看到了“现实”，而没有那个用数据逼近的过程，所以除非他累得头脑发麻或者喝了酒，你几乎不可能让他判断错误。

退一步来看，图像识别所谓的“正确分类”都是人定义的。是人给了那些东西名字，是许多人一起标注了训练用的图

片。所以这里所谓的“解析解”，“现实”，全都是人定义的。一定是某人看到了某个事物，他理解了它的结构和性质，然后给了它一个名字。所以别的人也可以通过理解同一个事物的结构，来知道它是什么。

神经网络不能看到事物的结构，所以它们也就难以得到精确的分类，所以机器在图像识别方面是几乎不可能超越人类的。现在所谓的“超人类视觉”的深度学习模型，大部分都是欺骗和愚弄大众。使用没有普遍性的数据集，使用不公平的准确率标准来对比，所以才显得机器好像比人还厉害了。这是一个严重的问题，在后面我会详细分析。

神经网络训练很像应试教育

神经网络就像应试教育训练出来的学生，他们的目标函数是“考高分”，为此他们不择手段。等毕业工作遇到现实的问题，他们就傻眼了，发现自己没学会什么东西。因为他们学习的时候只是在训练自己“从 ABCD 里区分出正确答案”。等到现实中没有 ABCD 的时候，他们就不知道怎么办了。

深度学习训练出来的那些“参数”是不可解释的，因为它们存在的目的只是把数据拟合出来，把不同种类的图片分离开，而没有什么意义。AI 人士喜欢给这种“不可解释性”找借口，甚至有人说：“神经网络学到的数据虽然不可解释，但它却出人意料的有效。这些学习得到的模型参数，其实就是知识！”

这些模型真的那么有效吗？那为什么能够被如此离谱的图片所欺骗呢？说“那就是知识”，这说法简直荒谬至极，严重玷污了“知识”这个词的意义。这些“学习”得到的参数根本就不是本质的东西，不是知识，真的就是一堆毫无道理可言的数字，只为了降低“误差”，能够把特征空间的图片区分开来，所以神经网络才能被这样钻空子。

说这些参数是知识，就像在说考试猜答案的技巧是知识一样可笑。“另外几套题的第十题都是 B，所以这套题的第十题也选 B”……深度学习拟合函数，就像拿历年高考题和它们的答案来拟合函数一样，想要不上课，不理解科目知识就做出答案来。有些时候它确实可以蒙对答案，但遇到前所未见的题目，或者题目被换了一下顺序，就傻眼了。

人为什么可以不受这种欺骗呢？因为人提取了高级的拓扑结构，不是瞎蒙的，所以人的判断不受像素的影响。因为提取了结构信息，人的观察是具有可解释性的。如果你问一个小孩，为什么你说这是一只猫而不是一只狗呢？她会告诉你：“因为它的耳朵是这样的，它的牙是那样的，它走路的姿势是那样的，它常常磨爪子，它用舌头舔自己……”

做个实验好了，你可以问问你家孩子这是猫还是狗。如果是猫，为什么他们认为这是一只猫而不是一只狗？



神经网络看到一堆像素，很多层处理之后也不知道是什么结构，分不清“眼睛”，“耳朵”和“嘴”，更不要说“走路”之类的动态概念了，所以它也就无法告诉你它认为这是猫的原因了。拟合的函数碰巧把这归成了猫，如果你要追究原因，很可能是肤浅的：图片上有一块像素匹配了图片库里某只猫的毛色纹理。

有一些研究者把深度神经网络的各层参数拆出来，找到它们对应的图片中的像素和纹理，以此来证明神经网络里的参数是有意义的。乍一看好像有点道理，原来“学习”就能得到这么多好像设计过的滤镜啊！可是仔细一看，里面其实没有多少有意义的内容，因为它们学到的参数只是能把那些图片类别分离开。

所以人的视觉系统很可能是跟深度神经网络原理完全不同的，或者只有最低级的部分有相似之处。

“神经网络”与人类神经元的关系是肤浅的

为什么 AI 人士总是认为视觉系统的高级功能都能通过“学习”得到呢？非常可能的事情是，人和动物视觉系统的“结构理解”，“3D建模”功能不是学来的，而是早就固化在基因里了。想一想你生下来之后，有任何时候看到世界是平面的，毫无关联的像素吗？

所以我觉得，人和动物生下来就跟现有的机器不一样，结构理解所需的硬件在胚胎里就已经有了，只等发育和激活。人是有学习能力，可是人的学习是建立在结构理解之上，而不是无结构的像素。另外人的“学习”很可能处于比较高的层面，而不是神经元那么“底层”的。人的神经系统里面并没有机器学习那种 back-propagation。

纵使你有再多的数据，再多的计算力，你能超越为期几十亿年的，地球规模的自然进化和选择吗？与其自己去“训练”或者“学习”，不如直接从人身上抄过来！但问题是，我们真的知道人的视觉系统是如何工作的吗？

神经科学家们其实并没有完全搞明白人类视觉系统是如何工作的。就像所有的生物学领域一样，人们的理解仍然是很粗浅的。神经网络与人类视觉系统的关系是肤浅的。每当你质疑神经网络与人类视觉系统的关系，AI 研究者就会抬出 Hubel & Wiesel 在 1959 年拿猫做的那个实验：“有人已经证明了人类视觉系统就是那样工作的！”如此的自信，不容置疑的样子。

我问你啊，如果我们在 1959 年就已经知道人类视觉系统的工作原理细节，为什么现在还各种模型改来改去，训练来训练去呢？直接模仿过来不就行了？所以这些人的说法是自相矛盾的。

你想过没有，为什么到了 2019 年，AI 人士还拿一个 60 年前的实验来说明问题？这 60 年来就没有新的发现了吗？而且从 H&W 的实验你可以看出来，它只说明了猫的视觉神经有什么样的底层功能（能够做“线检测”），却没有说那就是全部的构造，没说上层的功能都是那样够构造的。

H&W 的实验只发现了最底层的“线检测”，却沒有揭示这些底层神经元的信号到了上层是如何组合在一起的。“线检测”是图像处理的基础操作。一个能够识别拓扑结构的动物视觉系统，理所当然应该能做“线检测”，但它应该不止有这种低级功能。

视觉系统应该还有更高级的结构，H&W 的实验并没能回答这个问题，它仍然是一个黑盒子。AI 研究者们却拿着 H&W 的结果大做文章，自信满满的声称已经破解了动物视觉系统的一切奥秘。

那些说“我们已经完全搞明白了人类视觉是如何工作”的 AI 人士，应该来看看这个 2005 年的分析 Herman grid 幻觉现象的幻灯片。这些研究来自 Schiller Lab, MIT 的脑科学和认知科学实验室。通过一系列对 Herman grid 幻觉图案的改动实验，他们发现长久以来（从 1960 年代开始）对产生这种现象的理解是错误的：那些暗点不是来自视网膜的“边缘强化”功能。他们猜想，这是来自大脑的 V1 视觉皮层的 S1 “方向选择”细胞。接着，另一篇 2008 年的 paper 又说，Schiller 的结果是不对的，这种幻觉跟那些线条是直的有关系，因为你如果把那些白线弄弯，幻觉就消失了。然后他们提出了他们自己的，新的“猜想”。

Stopping the Hermann grid illusion by sine distortion



从这种研究的方式我们可以看出，即使是 MIT 这样高级的研究所，对视觉系统的研究还处于“猜”的阶段，把人脑作为黑盒子，拿一些图片来做“行为”级别的实验。他们并没有完全破解视觉系统，看到它的“线路”和“算法”具体如何工作，而是给它一些输入，测试它的输出。这就是“黑盒子”实验法。以至于很多关于人类视觉的理论都不是切实而确定的，很可能是错误的猜想。

脑科学发展到今天也还是如此，AI 领域相对于脑科学的研究方式，又要低一个级别。2019 年了，仍然抬出神经科学家 1959 年的结果来说事。闭门造车，对人家的最新成果一点都不关心。现在的深度神经网络模型基本是瞎蒙出来的。把一堆像素操作叠在一起，然后对大量数据进行“训练”，以为这样就能得到所有的视觉功能。

动物视觉系统里面真有“反向传导”（back-propagation）这东西吗？H&W 的实验里面并没有发现 back-propagation。实际上神经科学家们至今也没有发现神经系统里面有 back-propagation，因为神经元的信号传递机制不能进行“反向”的通信。很多神经科学家的结论是，人脑里面进行 back-propagation 不大可能。

所以神经网络的各种做法恐怕没有受到 H&W 实验的多大启发。只是靠这么一个肤浅的相似之处来显得自己接近了“人类神经系统”。现在的所谓“神经网络”，其实只是一个普通的数学函数的表达式，里面唯一起作用的东西其实是微积分，所谓 back-propagation，就是微积分的求导操作。神经网络的“训练”，就是反复求导数，用梯度下降方法进行误差最小化，拟合一个函数。这一切都跟神经元的工作原理没什么关系，完全就是数学。

为了消除无知带来的困惑，你可以像我一样，自己去了解一下人类神经系统的工作原理。我推荐你看看这个叫《Interactive Biology》的 YouTube 视频系列。你可以从中轻松地理解人类神经系统一些细节：神经元的工作原理，视觉系统的原理，眼睛，视网膜的结构，听觉系统的工作原理，等等。神经学家们对此研究到了如此细节的地步，神经传导信息过程的每一个细节都展示了出来。



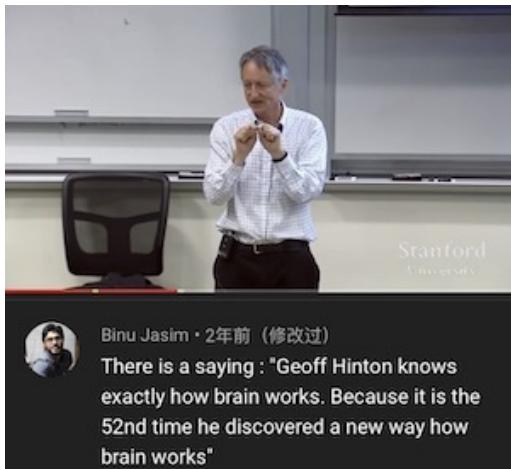
AI 研究者并不知道人脑如何工作

AI 领域真的理解人脑如何工作吗？你可以参考一下这个演讲：“[Can the brain do back-propagation?](#)”（人脑能做 back-propagation 吗？）。演讲人是深度学习的鼻祖级人物 Geoffrey Hinton。他和其它两位研究者（Yoshua Bengio 和 Yann LeCun），因为对深度学习做出的贡献，获得了 2018 年的图灵奖。演讲一开头 Hinton 说，神经科学家们说人脑做 back-propagation 是不可能的，然后他开始证明这是可能的，依据神经元的工作原理，back-propagation 如何能用人脑神经元来实现。

是的，如果你有能力让人脑按你的“算法”工作的话，神经元组成的系统也许真能做 back-propagation，可是人脑是你设计的吗？很可惜我们无法改变人脑，而只能去“发现”它到底是如何工作。这不是人脑“能不能”的问题，而是“做不做”的问题。研究人脑是一个科学发现工作，而不是一个工程设计工作。

看了这个演讲，我觉得 AI 人士已经进入了一种“上了天”的状态。他们坚定的认为自己的模型（所谓的“神经网络”）就是终极答案，甚至试图把人脑也塞进这个模型，设想人脑神经元如何能实现他们所谓的“神经网络”。可是他们没有发现，人脑的方式也许比他们的做法巧妙很多，根本跟他们的“神经网络”不一样。

从这个视频我们也可以看出，神经科学界并不支持 AI 领域的说法。AI 领域是自己在那里瞎猜。视频下面有一条评论我很欣赏，他用讽刺的口气说：“Geoff Hinton 确切地知道人脑是如何工作的，因为这是他第 52 次发现人脑工作的新方式。”



AI 人的盲目信仰

AI 人士似乎总是有一种不切实际的“信仰”或者“信念”，他们坚信机器一定可以具有人类一样的智能，总有一天能够在所有方面战胜人类。总是显示出一副“人类没什么了不起”的心态，张口闭口拿“人类”说事，好像他们自己是另外一个物种，已经知道人类的一切能力，有资格评判所有人的智力似的。

我不知道是什么导致了这种“AI 宗教”。有句话说得好：“我所有的自负都来自我的自卑，所有的英雄气概都来自于我内心的软弱，所有的振振有词都因为心中满是怀疑。”似乎是某种隐藏很深的自卑和怨恨，导致了他们如此的坚定和自负。一定要搞出个超越所有人的机器才善罢甘休，却没发现人类智能的博大精深已经从日常生活的各种不起眼的小事透露出来。

他们似乎看不到世界上有各种各样，五花八门的人类活动，每一种都显示出奇迹般的智能。连端茶倒水这么简单的事情，都包含了机器望尘莫及的智能，更不要说各种体育运动，音乐演奏，各种研究和创造活动了。就连比人类“低级”一点的动物，各种宠物，家畜家禽，飞鸟走兽，甚至昆虫，全都显示出足以让人敬畏的智能。他们对所有这些奇迹般的事物视而不见，不是去欣赏他们的精巧设计和卓越表现，而是坐井观天，念叨着“机器一定会超越人类”。

他们似乎已经像科幻电影似的把机器当成了一个物种，像是保护“弱势群体”一样，要维护机器的“权益”和“尊严”。他

们不允许其他人质疑这些机器，不允许你说它们恐怕没法实现人类一样的智能。总之机器在他们心理已经不再是工具，而是活的生命，甚至是比人还高级的生命。

对此你可以参考另一个 Geoffrey Hinton 的[采访视频](#)，录制于今年 5 月份的 Google 开发者大会（Google I/O '19）。

从这个视频里面我看到了许多 AI 人士盲目信仰和各种没有根据的说法的来源，因为这些说法全都集中而强烈的体现在了 Hinton 的谈话中。他如此的坚信一些没有根据的说法，不容置疑地把它们像真理一样说出来，却没有任何证据。有时候主持人都不得不采用了有点怀疑的语气。

Hinton 在采访中有以下说法：

1. “神经网络被设计为像人脑的工作原理。”
2. “等神经网络能够跟人对话，我们就能用它来进行教育工作了。”
3. “神经网络终究会在所有事情上战胜人类。”
4. “我们不都是神经网络吗？”（先后强调了两次）
5. “…… 所以神经网络能够实现人类智能的一切功能。这包括感情，意识等。”
6. “人们曾经认为生命是一种特殊的力量，现在生物学解释了生命的一切。人们现在仍然认为意识是特殊的，可是神经网络将会说明，意识并没有什么特别。”



他的这些说法都是不准确，不科学，没有根据的。

我发现每当主持人用稍微怀疑的语气回答：“这真的可以实现吗？” Hinton 就会回答：“当然能。我们不都是神经网络吗？”这里有一个严重的问题，那就是他所谓的“神经网络”，其实并不是人脑里面的神经元连成的网络。AI 领域的“神经网络”只是他们自己的数学模型，是他们自己给它起名叫“神经网络”而已。所以他的这种“证明”其实是在玩文字游戏：“因为我们都是神经网络，所以神经网络能够实现一切人类智能，感情，甚至意识本身！”

前面的“神经网络”和后面的“神经网络”完全是两回事。我们是“神经网络”吗？我们的脑子里是有神经元，神经元貌似连成了一个网络，可是它的结构却跟 AI 领域所谓的“神经网络”是两回事，工作原理也非常不一样。Hinton 面对问题作出这样的回答，是非常不科学，不负责任的。

最后关于生命，感情和意识的说法，我也很不认同。虽然生物学解释了生命体的各种构造和原理，可是人们为什么仍然没能从无生命的物质制造出有生命的事物呢？虽然人们懂得那么多生物学，生物化学，有机化学，甚至能合成出各种蛋白质，可是为什么没能把这些东西组装在一起，让它“活”起来呢？这就像你能造出一些机器零件，可是组装起来之后，发现这机器不转。你不觉得是因为少了点什么吗？生物学发展了这么久，我们连一个最简单的，可以说是“活”的东西都没造出来过，你还能说“生命没什么特别的”吗？

这说明生物学家们虽然知道生命体的一些工作原理，却没有从根本上搞明白生命到底是什么。也就是说人们解决了一部分“how”问题（生命体如何工作），却不理解“what”和“why”（生命是什么，为什么会出现生命）。

实际上生物学对生命体如何工作（how）的理解都还远远不够彻底，这就是为什么我们还有那么多病无法医治，甚至连一些小毛病都无法准确的根治，一直拖着，只是不会马上致命而已。“生命是什么”的 what 问题仍然是一个未解之谜，而不像 Hinton 说的，全都搞明白了，没什么特别的。

也许生命就是一种特别的东西呢？也许只有从有生命的事物，才能产生有生命的事物呢？也许生命就是从外星球来的，也许就是由某种更高级的智慧设计出来的呢？这些都是有可能的。真正的科学家应该保持开放的心态，不应该有类似“人定胜天”这样的信仰。我们的一切结论都应该有证据，如果没有我们就应该说“一定”或者“必然”，说得好像

所有秘密全都解开了一样。

对于智能和意识，我也是一样的态度。在我们没有从普通的物质制造出真正的智能和意识之前，不应该妄言理解了关于它们的一切。生命，智能和意识，比有些人想象的要奇妙得多。想要“人造”出这些东西，比 AI 人士的说法要困难许多。

有心人仔细观察一下身边的小孩子，小动物，甚至观察一下自己，就会发现它们的“设计”是如此的精巧，简直不像是随机进化出来的，而是由某个伟大的设计者创造的。46 亿年的时间，真的够进化和自然选择出这样聪明的事物吗？

别误会了，我是不信宗教的。我觉得宗教的圣经都是小人书，都是某些人吓编的。可是如果你坚定的相信人类和动物的这些精巧的结构都是“进化”来的，你坚定的相信它们不是什么更高级的智慧创造出来的，那不也是另外一种宗教吗？你没有证据。没有证据的东西都只是猜想，而不能坚信。

好像扯远了……

总之，深度学习的鼻祖级人物说出这样多信念性质的，没有根据的话，由此可见这个领域有多么混沌。另外你还可以从他的谈话中看出，他所谓的“AI”都是各种相对容易的识别问题（语音识别，图像识别）。他并没有看清楚机器要想达成“理解”有多困难。而“识别”与“理解”的区别，就是我的这篇文章想澄清的问题。

炼丹师的工作方式



设计神经网络的“算法工程师”，“数据科学家”，他们工作性质其实很像“炼丹师”(alchemist)。拿个模型这改改那改改，拿海量的图片来训练，“准确率”提高了，就发 paper。至于为什么效果会好一些，其中揭示了什么原理，模型里的某个节点是用来达到什么效果的，如果没有它会不会其实也行？不知道，不理解。甚至很多 paper 里的结果无法被别的研究者复现，存在作假的可能性。

我很怀疑这样的研究方式能够带来什么质的突破，这不是科学的方法。如果你跟我一样，把神经网络看成是用“可求导编程语言”写出来的代码，那么现在这种设计模型的方法就很像“一千万只猴子敲键盘”，总有一只能敲出“Hello World！”

许多数学家和统计学家都不认同 AI 领域的研究方式，对里面的很多做法表示不解和怀疑。为此斯坦福大学的统计学系还专门开了一堂课 [Stats 385](#)，专门讨论这个问题。课堂上请来了一些老一辈的数学家，一起来分析深度学习模型里面的各种操作是用来达到什么目的。有一些操作很容易理解，可是另外一些没人知道是怎么回事，这些数学家都看不明白，连设计这些模型的炼丹师们自己都不明白。

所以你也许看到了，AI 研究者并没能理解人类视觉系统的工作原理，许多的机器视觉研究都是在瞎猜。在接下来的续集中，我们会看到他们所谓的“超人类识别率”是如何来的。

请看下一篇：[机器与人类视觉能力的差距 \(3\)](#)

机器与人类视觉能力的差距 (1)

本文属于个人观点，跟本人在职公司的立场无关。由于最近 GitHub 服务器在国内访问速度严重变慢，虽然经过大幅度压缩尺寸，文中的图片仍然可能需要比较长时间才能加载。这篇文章揭示了 AI 领域重要的谬误和不实宣传，为了阻止愚昧的蔓延，我鼓励大家转发这篇文章和它的后续，转发时只需要注明作者和出处就行。

很多人以为人工智能就快实现了，往往是因为他们混淆了“识别”和“理解”。现在所谓的“人工智能”都是在做识别：语音识别，图像识别，而真正的智能是需要理解能力的。我们离理解有多远呢？恐怕真正的工作根本就没开始。

很长时间以来，我都在思索理解与识别的差别。理解与识别是很不一样的，却总是被人混为一谈。我深刻的明白理解的重要性，可是我发现很少有其他人知道“理解”是什么。AI 领域因为混淆了识别和理解，一直以来处于混沌之中。

最近因为图像识别等领域有了比较大的进展，人们对 AI 产生了很多科幻似的，盲目的信心，出现了自 1980 年代以来最大的一次“AI 热”。很多人以为 AI 真的要实现了，被各大公司鼓吹的“黑科技”冲昏了头脑，却看不到现有的 AI 方法与人类智能之间的巨大鸿沟。所以下面我想介绍一下我所领悟到的机器和人类在视觉能力方面的差距，希望一些人看到之后，能够再次拥有冷静的头脑。

在之前一篇文章《[人工智能的局限性](#)》中，我已经阐述了对自然语言处理领域误区的看法。当时因为对计算机视觉方面了解不多，所以没有包含视觉方面的内容。熟悉了机器视觉的各种做法之后，我想在这篇文章里详述一下视觉方面的内容。这两篇文章加在一起，可以说概括了我对 AI 语言和视觉两个方面的领悟。

“图像识别”和“视觉理解”的差别

对于视觉，AI 领域混淆了“图像识别”和“视觉理解”。现在热门的所谓“AI”都是“图像识别”，而动物的视觉系统具有强大的“视觉理解”。视觉理解和图像识别有着本质的不同。

深度学习视觉模型 (CNN一类的) 只是从大量数据拟合出从“像素=>名字”的函数。它也许能从一堆像素猜出图中物体的“名字”，但它却不知道那个物体“是什么”，无法对物体进行操作。注意我是特意使用了“猜”这个字，因为它真的在猜，而不像人一样准确地知道。

“图像识别”跟“语音识别”处于同样的级别，停留在语法（字面）层面，而没有接触到“语义”。语音识别是“语音=>文字”的转换，而图像识别则是“图像=>文字”的转换。两者都输出文字，而“文字”跟“理解”处于两个不同的层面。文字是表面的符号，你得理解了它才会有意义。

怎样才算是“理解了物体”呢？至少，你得知道它是什么形状的，有哪些组成部分，各部分的位置和边界在哪里，大概是什么材料做成的，有什么性质。这样你才能有效的对它采取行动，达到需要的效果。否则这个物体只是一个方框上面加个标签，不能精确地进行判断和操作。



想想面对各种日常事物的时候，你的脑子里出现的是它们的名字吗？比如你拿起刀准备切水果，旁边没有人跟你说话，你的脑子里出现了“刀”这个字吗？一般是没有的。你的脑子里出现的不是名字，而是“常识”。常识不是文字，而是一种抽象而具体的数据。

你知道这是一把刀，可是你的头脑提取的不是“刀”这个字，而是刀“是什么”。你的视觉系统告诉你它的结构是什么样的。你知道它是金属做的，你看到刀尖，刀刃，刀把，它也许是折叠的。经验告诉你，刀刃是锋利的可以切东西的部分，碰到可能会受伤，刀把是可以拿的地方。如果刀是折起来的，你得先把它翻开，那么你从哪一头动手才能把它翻开，它的轴在哪里？

你顺利拿起刀，开始切水果。可是你的头脑里仍然没有出现“刀”这个字，也没有“刀刃”，“刀把”之类的词。在切水果的同时，你大脑的“语言中心”可能在哼一首最近喜欢的歌词，它跟刀没有任何关系。语言只是与其他人沟通的时候需要的工具，自己做事的时候我们并不需要语言。完成切水果的动作，你需要的是由视觉产生的对物体结构的理解，而不是语言。

你不需要知道一个物品叫什么名字就能正确使用它。同样的，光是知道一个物品的名字，并不能帮助你使用它。看到一个物体，如果脑子里首先出现的是它的名字，那么你肯定是很愚钝的人，无法料理自己的生活。现在的“机器视觉”基本就是那样的。机器也许能得出图片上物体的名字，却不知道它是什么，无法操作它。

试想一下，一个不能理解物体结构的机器人，它只会使用图像识别技术，在你的头上识别出一个个的区域，标注为“额头”，“头发”，“耳朵”……你敢让它给你理发吗？

这就是我所谓的“视觉理解”与“图像识别”的差别。你会意识到，这种差别是巨大的。

视觉识别不能缺少理解

如果我们降低标准，只要求识别出物体的名字，那么以像素为基础的图像识别，比如卷积神经网络（CNN），也是没法像人一样准确识别物体的。人识别物体并不是像神经网络那样的“拍照，识别”两节拍动作，而是一个动态的，连续的过程：观察，理解，观察，理解，观察，理解……

感官接受信息，中间穿插着理解，理解反过来又控制着观察的方向和顺序。理解穿插在了识别物体的过程中，“观察/理解”成为不可分割的整体。人看到物体的一部分，理解了那是什么，然后继续观察它周围是什么，反复这个过程，最后才判断出物体是什么。机器在识别的过程中没有理解的成分存在，这就是为什么机器在图像识别能力上无法与人类匹敌。

这个“观察/理解”的过程发生的如此之快，眨眼间就完成了，以至于很多人都没察觉到其中“理解成分”的存在。所以我们现在放慢这个过程，来一个慢镜头特写，看看到底发生了什么。假设你从来没见过下面这个东西，你知道它是什么吗？



一个从没见过这东西的人，也会知道这是个“车”。为什么呢？因为它有轮子。为什么你知道那是轮子呢？仔细一想，因为它是圆的，中间有轴，所以好像能在地面上滚动。为什么你知道那是“轴”呢？我就不继续折腾你了，自己想一下吧。所有这些分析都是“视觉理解”所产生的，而这些理解依赖于你一生积累的经验，也就是我所谓的“常识”。

其实为了识别这个东西，你并不需要分析这么多。你之所以做这些分析，是因为另一个人问你“你怎么知道的？”人识别物体靠的是所谓“直觉”。一看到这个图片，你的脑子里自然产生了一个3D模型。一瞬间之后，你意识到这个模型符合“车”的机械运动原理，因为你以前看见过汽车，火车，拖拉机……你的脑子里浮现出这东西可能的运动镜头，你仿佛看到它随着轮子在动。你甚至看到其中一个轮子压到岩石，随着连杆抬了起来，而整个车仍然保持平衡，没有反倒，所以这车也许能对付崎岖的野外环境。

这里有一个容易忽视的要点，那就是轮子的轴必须和车体连在一起。如果轮子跟车体没有连接，或者位置不对，看起来无法带着车体一起运动，人都是知道的。这种轮轴与车身的连接关系，属于一种叫“拓扑”（topology）的概念。

拓扑学是一门难度挺高的数学分支，但人似乎天生就理解某些浅显的拓扑概念。实际上似乎高等动物都或多或少理解一些拓扑概念，它们一看就知道哪些东西是连在一起的，哪些是分开的。捕猎的动物都知道，猎物的尾巴是跟它们身体连在一起的，所以咬住它们的尾巴就能抓住它们。

拓扑学还有一个重要的概念，那就是“洞”。聪明一点的动物基本上都理解“洞”的概念。很显然老鼠，兔子等穴居动物必须理解洞是什么。它们的天敌，猫科动物等，也理解洞是什么。如果我拿一个纸箱给我的猫玩，我在上面挖一个洞，等他钻进去，他是不会进去的。我必须在上面挖两个洞，他才会进去。为什么呢？因为他知道，要是箱子上面只有一个洞，要是他进去之后洞被堵上，他就出不来了！

机器如何才能理解洞这个概念呢？它如何理解“连续”？

总之，人看到物体，他看到的是一个3D模型，他理解其中的拓扑关系和几何性质，所以一个人遇到前所未见的物体，他也能知道它大概是什么，推断出如何使用它。理解使得人可以非常准确地识别物体。没有理解能力的机器是做

不到这一点的。

人的视觉系统与机器的差别

人的眼睛与摄像头有着本质的差异。眼睛的视网膜中央非常小的一块区域叫做“fovea”，里面有密度非常高的感光细胞，而其它部分感光细胞少很多，是模糊的。可是眼睛是会转动的，它被脑神经控制，敏捷地跟踪着感兴趣的部位：线条，平面，立体结构…… 人的视觉系统能够精确地理解物体的形状，理解拓扑，而且这些都是 3D 的。人脑看到的不是像素，而是一个 3D 拓扑模型。

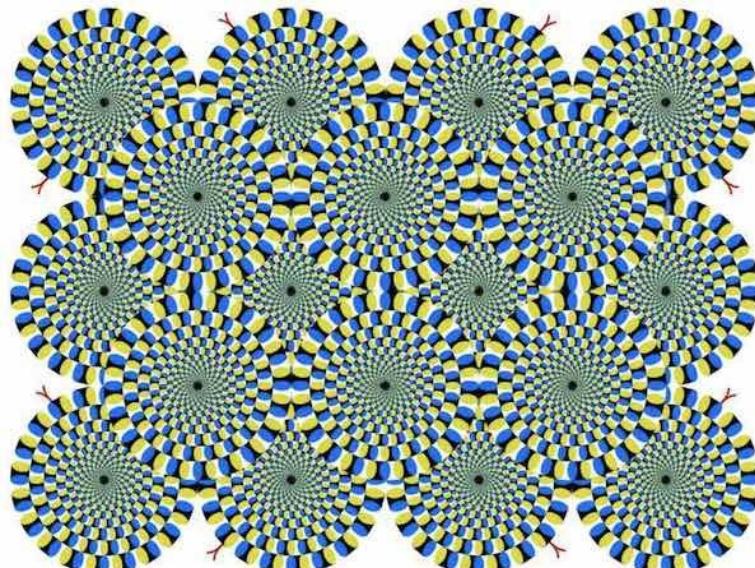
眼睛观察的顺序，不是一行一行从上往下把每个“像素”都记下来，做成 6000x4000 像素的图片，而是聚焦在重点上。它可以沿着直线，也可以沿着弧线观察，可以转着圈，也可以跳来跳去的。人脑通过自己的理解能力，控制着眼睛的运动，让它去观察所需要的重点。由于视网膜中央分辨率极高，所以人脑可以得到精度非常高的信息。然而由于不是每个地方都看的那么仔细，所以眼睛采集的信息量可能不大，人脑需要处理的信息也不会很多。

人的视觉系统能理解点，线，面的概念，理解物体的表面是连续的还是有洞，是凹陷的还是凸起的，分得清里和外，远和近，上下左右…… 他能理解物体的表面是什么质地，如果用手去拿会有什么样的反应。他能想象出物体的背面大概是什么样子，他能在头脑中旋转或者扭曲物体的模型。如果物体中间有缺损，他甚至能猜出那位置之前什么样子。

人的视觉系统比摄像头有趣的多。很多人都看过“光学幻觉”（optical illusion）的图片，它们从一个角度揭示了人的视觉系统背后在做什么。比如下图本来是一个静态的图片，可是你会感觉有很多暗点在白线的交叉处，但如果你仔细看某一个交叉处，暗点却又不见了。这个幻觉很经典，被叫做 Herman grid，在神经科学界被广泛研究。稍后我还会提到这个东西。



本来是静态图片，你却感觉它在转。



本来上下两块东西是一样的颜色，可是看起来下面的颜色却要浅一些。如果你用手指挡住中间的高亮部分，就会发现上下两块的颜色其实是一样的。



另一个类似的幻觉，是著名的“Abelson 棋盘幻觉”。图中 A 和 B 两个棋盘格子的颜色是一样的，你却觉得 A 是黑色，而 B 是白色。不信的话你可以用软件把这两块格子从图片上切下来，挨在一起对比一下。如果你好奇这是为什么，可以参考[这篇文章](#)。



在下图里，你会觉得看见了一个黑色的倒三角形，可是其实它并不存在。



很多的光学幻觉都说明人的视觉系统不是简单的摄像头一样的东西，它具有某些特殊功能。这些特殊功能和机制导致了这些幻觉。这使得人类视觉不同于机器，使得人能够提取出物体的结构信息，而不是只看到像素。

提取物体的拓扑结构特征，这就是为什么人可以理解抽象画，漫画，玩具。虽然世界上没有猫和老鼠长那个样子，一个从来没看过《猫和老鼠》动画片的小孩，却知道这是一只猫和一只老鼠，后面有个房子。你试试让一个没有拿《猫和老鼠》剧照训练过的深度学习模型来识别这幅图？



更加抽象的玩具，人也能识别出它们是哪些人物。头和四肢都变成了方的，居然还是觉得很“像”。你不觉得这很神奇吗？



人脑理解“拓扑”的概念，这使得人能够不受具体像素干扰而正确处理各种物体。对拓扑结构的理解使得人对物体的识别非常准确，甚至可以在信息不完整，模糊，扭曲的情况下工作，在恶劣的天气环境下，有反光，有影子的情况下也能识别物体。

说到反光，你有想过机器要如何才能识别出场景里有一面镜子或者玻璃吗？如果场景中有反光的物体，比如镜子，平静的水面，镀铬的物品，神经网络（CNN）那种依靠像素滤镜训练出来的函数还会有用吗？要知道它们看到的像素，可能有一大片是通过镜面反射形成的，所以无法通过局部的纹理识别出这种情况来。



这是个现实的问题。汽车或者机器人要如何知道前面的路面上有积水或者结冰了？它们要如何知道从水面反射过来的镜像不是真实的物体？比如，它们如何知道下图里路面上的倒影不是真正的树呢？要知道，倒影的像素纹理，跟真实的场景可能是非常相似的。



人是通过对光的理解，各种常识来识别镜子，玻璃，地上的水和冰的存在。一个不理解光和水的性质的机器，它能察觉这些东西的存在吗？靠像素分析能知道这些？要知道，这些东西在某些地方出现，可以是致命的危险。

很有趣的事情，理解光线的反射和折射，似乎已经固化到了每个动物的视觉系统里面。我观察到这一点，是因为我的卧室和客厅之间的橱柜门上有两面大镜子。我的猫在卧室里，能够从镜子里看见我在客厅拿着逗猫绳。他冲过来的时候却不会撞到镜子上面，而是出了卧室门立马转一个角度，冲向我的方向。我每次看到他敏捷的动作都会思考，他是如何知道镜子的存在呢？他是如何知道镜子里的猫就是他自己，而不是另一只猫？



人脑会构造事物的 3D 模型

说了光，再来说影吧。画过素描的人都知道，开头勾勒出的轮廓是没有立体感的，然后你往恰当的位置加一些阴影，就有了立体感。所以动物的视觉系统里存在对影子的分析处理，而且这种功能我们似乎从来没需要学习，生下来就有。“立体视觉”是如此强烈的固化到了我们的头脑里，一旦产生了立体感，你就很难再看见平面的像素。



靠着光和影的组合，人和动物能得到很多信息。比如上图，我们不但看得出这是一个立体的鸡蛋，而且能推断出鸡蛋下面是一个平面，可能是一张桌子，因为有阴影投在了上面。

神经网络知道什么是影子吗？它如何知道影子不是实际存在的物体呢？它能从影子得到有用的信息吗？

神经网络根本不知道影子是什么。早就有人发现，Tesla 基于图像识别的 Autopilot 系统会被阴影所迷惑，以为路面上的树影是一个障碍物，试图避开它，却差点撞上迎面来的车。我在很早的一篇[文章](#)已经谈过这个问题。

再来一个关于绘画的话题。学画的初期，很多人都发现画“透视”特别困难。所谓透视就是“近大远小”。本来房子的几堵墙都是长方形，是一样高的，可是你得把远的那一边画短一些，而且相关部分的比例都要画对，就像照片上那样，所以墙就成了梯形的。房顶，窗户等，也全都得做相应的调整。你得这样画，看画的人才会感觉是对的，不然就会感觉哪里不对劲，不真实。



这件事真的很难，大部分人（包括我）一辈子都没学会画透视。虽然拿起笔来量一下，我确实看到远的那一边要短一

些，可是我的脑子似乎会“自动纠错”，让我认为它们都是一样长的。所以要是光靠眼睛徒手作画，我会把那些边都画成一样长。我似乎永远学不会画画！

画透视是如此困难的事情，以至于 16 世纪的德国画家丢勒为此设计了一种专门的设备。



你可能没有想到，这个使得我们学画困难的罪魁祸首，其实是人类视觉系统的一项重要功能，它帮助我们理解身边的环境。虽然眼睛看到的物体是近大远小，可是人脑会自动调整它们在你“头脑里的长度”，所以你知道它们是一样长的。

这也许就是为什么人能从近大远小的光学成像还原出正确的 3D 模型。在你头脑中的模型里面，房子的几堵墙是一样高的，就像它们在现实中的情况一样。有了准确的 3D 模型，人才能正确地控制自己在房子周围的运动。

这种导致我们学画困难的“3D 自动纠错”功能，似乎固化到了每个人，每个高等动物的视觉系统里。我们并不需要学习就有这种能力，它一直都在起作用。反倒是我们要想“关掉”这个功能的时候，需要付出非常多的努力！

为什么人想要画出透视效果那么困难呢？因为一般人画画，都不是在画他们头上那两只眼睛看到的东西，而是在画他们的“心之眼”（mind's eye）看到的东西——他们头脑中的那个 3D 模型。这个 3D 模型是跟现实“同构”的，模型里房子的墙壁都是一样高的，他们画出来也是一样高的，所以就画错了。只有经过专业训练的画家，才有能力关闭“心之眼”，直接画出眼睛看到的东西。

我猜想，每一种高等动物的视觉系统都有类似的机制，使得它们从光学成像“重构”出与现实同构的 3D 模型。缺乏 3D 建模能力的机器，是无法准确理解看到的物体的。

现在很多自动驾驶车用激光雷达构造 3D 模型，可是相对于人类视觉形成的模型，真是太粗糙了。激光雷达靠主动发射激光，产生一个扫描后的“点云”，分辨率很低，只能形成一个粗糙的 3D 轮廓，无法识别物体，也无法理解它的结构。我们应该好好思考一下，为什么人仅靠被动接收光线就能构造出如此精密的 3D 模型，理解物体的结构，而且能精确地控制自己的动作来操作这些物体。

现在的深度学习模型都是基于像素的，没有抽象能力，不能构造 3D 拓扑模型，甚至连位置关系都分不清楚。缺乏人类视觉系统的这种“结构理解”能力，可能就是为什么深度学习模型需要那么多的数据，那么多的计算，才勉强能得出物体的名字。而小孩子识别物体根本不需要那么多数据和计算，看一两次就知道这东西是什么了。

人脑提取了物体的要素，所以很多信息都可以忽略了，所以人需要处理的数据量，可能比深度学习模型小很多。深度学习领域盲目地强调提高算力，制造出越来越大规模的计算芯片，GPU，TPU…… 可是大家想过人脑到底有多大计算能力吗？它可能并不需要很多计算。

从上面的各种现象，我们也许已经看明白了，人类视觉系统是很神奇的。现有的机器视觉研究并没有理解人类视觉的这些能力是怎么实现的。在接下来的续集中我们会详细的看清楚，AI 领域到底理解多少人类神经系统的构造。

请看下一篇：[机器与人类视觉能力的差距 \(2\)](#)

(这个系列的文章包含了很多独到的见解。如果你觉得有帮助可以[付费支持](#)。)

Talk is not cheap

(本文描述的是我长久的经历中形成的看法，跟我现在身边的人和事没有直接联系，请勿对号入座。)

长久以来，我发现挺多 IT 人士学会了一句口头禅，无论你表达什么观点，他们会拿出一副小学老师要检查作业的口气，说：“Talk is cheap. Show me the code!” 或者“给我看看你做出了什么！”

这类说法包含了两种可能的含义：

1. 说话是没用的，你要做出来我才信。
2. 你要已经有了重要的成果，才有资格发言。

“Talk is cheap. Show me the code.” 这句话出自 Linus Torvalds 在 linux-kernel mailing list 的一个[回帖](#)。如果你看了我对微内核和线程的[分析](#)，也许会明白 Jamie Lokier 的话其实是有意义的。如果保持开放的心态，继续的探讨也许会给 Linux 内核带来突破性的改进，然而这种可能性却被 Linus 一句“Talk is cheap. Show me the code.” 给扼杀了。

```
Date      Fri, 25 Aug 2000 11:09:12 -0700 (PDT)
From     Linus Torvalds <>
Subject   Re: SCO: "thread creation is about a thousand times faster than onnative

On Fri, 25 Aug 2000, Jamie Lokier wrote:
>
> Well well. I think it's possible to over the best of user-space "fake"
> threads plus the advantages of "true" kernel threads in one blindingly
> fast combination, in less than 8kB per thread.

Talk is cheap. Show me the code.

Linus
```

Linus 可能当时不耐烦了，你知道这家伙的性格……我相信他不是每次都说这样的话，但因为 Linus 形象太高大，这话就被人记下来，作为可以反复拿出来压制言论的手段。管你表达什么，他们都有一句万能的台词：“Talk is cheap. Show me the code.”

“苦干，用代码说话，忽视想法”，是很多程序员的误区。人的思想不一定需要代码来证明，甚至很多的想法无法简单的用代码表示，只有靠人的头脑才能想得清楚。思想是首要的，代码只是对思想的一种实现。我们先得要有思想（算法），才可能有代码。有些人动不动就“show me the code”，却忽视了思考和探讨的重要性。如果你没有好的想法，弄一堆代码出来又有什么用呢？只是死钻进一堆无谓的细节，掩盖了本质。

代码不能代替思想交流和讨论。代码不能清晰的表达一个人的想法，也不能显示一个人的思维深度。任何程序员都可以写出复杂冗长的代码，你有时间去看吗？就算水平很高的程序员，他的代码组织方式你不熟悉，也会看不出来本来的想法。实际的代码里面往往充斥着因为编程语言，硬件，系统，历史遗留问题导致的各种复杂性。如果每个想法真要“show me the code”才被考虑的话，那效率实在太低了。

我跟同事讨论代码的时候，一般都会先请他们在白板上画个图，用简单的语言解释他们的算法，这比直接看代码要容易很多。很多时候几句话就能说清楚，我在脑子里就能看到它是怎么工作的，根本用不着看代码。我从来不会说“show me the code！” 想法应该在实现之前被讨论清楚，对不可行的想法应该停止于摇篮之中，对可行的想法应该看到各种可能的发展方向…… 这些都应该在实现代码之前沟通弄明白，这会节省我们大量的时间和精力。

这就是 Talk 的价值。

另外，代码和成果不应该成为一个人是否可以表达看法的条件。只要这个人的见解有它的依据，就是有价值的。他不需要有什么重大的成就也应该可以表达自己的看法。这个我在之前一篇[文章](#)说过。以代码和所谓“成果”来压制人的言论自由是不合理的，而且代码和“成果”其实很难说明一个人的看法是否有价值。

很多人面试程序员都有类似的经验，他们给你看已经写好的代码，根本无法用来鉴别他们的水平。因为代码是可以拷贝的，所以你无法知道这代码是否他自己写出来的。代码可以是冗长晦涩的，所以就算是他自己写出来的，你也不会想花时间去看懂它。

代码是死的，它是对已有问题的解决方案。而你想要知道的是这个人在面对新的问题的时候，他会怎样去解决它。所以你必须知道这个人的思维方式，看清楚他是否真的知道他声称“精通”的那些东西。

一个人说他之前的工作做出了什么样的成果，很多时候也是不可靠的。因为成果是可以虚构或者窃取的，他可以把别人的成果说成是自己的。如果是管理岗位，“成果”就更加难以鉴定。这人也许只是瞎指挥，对很多人各种发号施令，对不同的人指出 N 种不同的方向，然后瞎蒙对了一个。其中一个方向做出了点东西，当然工作都是手下人做的，具体的想法都是手下人的。然后领导者挂个名字，就成了大家追捧的“技术大牛”。

很多博导都是用这种方法出成果的。招 N 个博士生来，分别给他们每人一个课题。管它有没有可能做出来，有没有价值，都跟你说这个课题很好。只要 N 个博士生有一两个做出东西，他就可以发 paper 升职了。被分配到那些做不出来的方向的学生，他才不管你的死活呢。

有见识的人跟他们对话，就会发现这些人一知半解，还仍然牛逼轰轰的样子。这就是我多次的经历。管他说做过什么代码项目，写了什么书，得了什么奖，一旦当面对话就能显示出真实的水平。代码和书都可以抄来，成果可以盗窃，你甚至可以因此得诺贝尔奖，可是对话没法偷来。对话可以显示出一个人是否有真知灼见。

一个小故事。我以前就职的某公司，有次招了一个 VP，他的 github 上有上百万行的代码，项目有上万的“star”，在领域里很是有点名声。这算是成果了吧？结果一进公司就各种瞎指挥，搞得大家没法工作了。还招进来很多自己圈子里的亲信，也是一群只会吹牛不做事的人，各种打压其他人，浪费大量的人力物力。我都感觉公司快要被搞垮了，最后创始人终于醒悟，费了好大功夫才把这些人赶走或者架空。

所以一个人有再多的代码，成果，都可能是没用的。我不排除它们是真实的情况，但你需要懂得如何去鉴别，而不只是依据这些表面的标准。

对于人的水平，我一般会观察他们说的话，最好是当面的对话。我不会盲目相信他们所谓的“成果”，我不看他们的代码。有真知灼见的人可以毫不犹豫地说出自己的想法和观点，而不需要时间去背诵和计算。我很容易看出一个人是否在说真话，因为说真话的人不需要时间去“计算”他们要说什么，不需要演戏。

可惜，“Talk is cheap”已经成为了很多人用来压制言论的手段。它误导了很多人，让他们无法正确鉴别技术人员的水平，犯下严重的人事错误。招进来一个错误的人，可以毁掉整个公司。

那些被“Talk is cheap”压制的人，变得不敢表达自己的观点，总想默默无闻“做”点什么给大家看。可是对方有什么资格要求这些呢？他们自己做出了什么呢？等你真做了给他们看，他们又会说你的东西不好，不如别人 xx 的。想当年我做的那什么，比你厉害多了…… 其他人也云里雾里，没有鉴别能力，只能随机倒向一边。

受到“show me the code”影响的人，可能还会让别人去看他的代码，而不解释自己的想法。大家工作中也许遇到过有人拒绝解释自己的想法，说：“你看代码就明白我做了什么。”这其实是不大尊重人的行为。没有人应该被迫去阅读其它人的代码，写出代码的人有义务讲清楚自己代码背后的思想。

由于我对某些领域很在行，不止一次有人 email 联系我：“我做了这个东西，我想知道你对它的评价。”连基本的想法都不说，甚至称呼和自我介绍都没有，接下来就是一个 github 的代码链接，或者粘贴一大段代码在 email 里面。这种代码我是不看的。我可能 email 都不会回，因为这显示出他们缺乏基本的礼貌和对他人时间的尊重。

代码不是有效的沟通工具，也不应该用来决定一个人是否有发言权。“Talk is cheap”只不过是封嘴的手段。说别人“Talk is cheap”的那些人，他们自己却不断地 talk。

人们应该可以平等自由的表达自己，不受这种无谓的教条压制。每当有人一针见血，指出我迷惑已久的问题的要点，豁然开朗的时候，我会很清楚的记得这个人。我会尊敬他，在合适的时候给予他回报。就算没有给我新的想法，要是他说出一些我曾经琢磨很久才想清楚的点，或者给了我另一个角度的观点，我也会记得他。这些给我指出正确方向的人，我不需要看他们的代码，我不以最终的“成果”来衡量他们的价值。想法和观点在我这里是高于代码的。

Talk 一点都不 cheap，而可以是有很大价值的。中国有句古话，“听君一席话胜读十年书”，说的就是这个道理。当然我们还是应该避免无意义的对话，但不应该笼统的说“Talk is cheap”。

可是我发现并不是每个人都像我这样。有些人，他迷惑的时候你给他指出要点或者方向，最后他却说那是他自己想出来的，甚至说你的话没有价值，我只看结果…… 遇到这种情况，你就知道遇到了错误的人。你不需要向他证明什么，不应该再给他任何有价值的信息。

很多人被“成果”或者代码所蒙蔽，而忽略了那些能够看透问题，用简单的几句话指出正确方向的人。我希望以这篇文章纠正很多业内人士的思维方式。

Talk is not cheap. Talk can be powerful.

关于微内核的对话

不知怎么的，最近“微内核 vs 宏内核”又成了热门话题。这场争论从 1992 年开始.....

FLAMEWAR

- In 1992, I said microkernels were better than monolithic designs
- Big flamewar with Linus Torvalds ensues
- 24 years later I still get lots of mail about this
- **Lesson: The Internet is like an elephant; it never forgets**



19

前言

说实话我很久没有关心操作系统了，因为通常所谓的“操作系统”在我心里不过是一个 C 语言的运行时系统（run-time system），就像 JVM 是 Java 的运行时系统一样。由于 C 语言的设计缺陷，这些系统引入了各种无须有的概念（进程，线程，虚拟内存……），以及它们带来的复杂性和开销。

微内核与宏内核之争当然也包括在内，在我看来这些都是无须有的概念和争论。操作系统相关的领域有很多的“宗教斗争”，比如“Linux vs Windows”，“自由软件 vs 不自由软件”，“RISC vs CISC”，甚至“VIM vs Emacs”..... 人们为了证明自己用的系统或者工具是世界上“最好”的，吵得昏天黑地。遇到有人指出自己用的工具的缺点，随时可能拿枪毙了对方。这些被叫做“flame war”。

我曾经是某些宗教斗争中活跃的一员，不知道这事的人可以去查一下我的历史。等你经历了很多才发现，原来这些宗教情绪和斗争都是那么幼稚无知。

这种“技术宗教情绪”往往显示出参与者心理地位的低下。因为他们缺乏自信，所以他们的心理需要靠山。这个靠山可能是某种操作系统（比如 Linux），某种编程语言（比如 C++），或者某种工具（比如 VIM）。这些人以自己能熟练使用这些工具为豪，居高临下，看不起“异教徒”。

具有技术宗教情绪的人看似是为了“技术”，“理想”，而其实跟那些以为开着豪车，穿着名牌就是“上流社会”的人是一样低级的，因为他们依靠于这些物品，所以他们的地位在这些物品之下。

一个人需要彻底的把这些东西看成是“东西”，不带有任何崇拜或者鄙视的情绪，他的心理才算是成熟了。

在我的理念里，一个操作系统本应该大概是[这个样子](#)。简单得很，根本不存在那么多问题。我可以利用这些思想来看透现有操作系统的绝大部分思想，管它是微内核还是宏内核。我可以把现有的操作系统看成是这个系统的“退化版”。

操作系统是一个死知识横行的领域。很多人发现操作系统课难学，难理解。里面有些内容，比如各种同步机制，很多人上完课毕了业，工作很多年以后都还弄不明白是怎么回事，它们为什么在那里。类似的东西包括虚拟内存，进程与线程的区别，等等。

经过了很多的经验和思考，加上其他领域给我的启发，我终于明白了。原来很多这些概念都是无须有的，死掉的知识。

操作系统课程里面的概念经常是这样形成的：

1. 很久以前，有人为了解决了一个特定的问题，提出了一个概念（比如 semaphore）。这个概念本来只有一个用途，就是解决他遇到的那个特定的问题。
2. 因为这人太有名，这概念就被写进了教科书里。有时候连他当时的具体实现细节都给写进去了。比如 semaphore 的两个操作被叫做 P 和 V，连这两个名字都给作为“典故”写进去了。
3. 教授们照本宣科，吹毛求疵，要你用这概念解决很多其它问题。很多根本就是人为造出来的变态问题，现实中遇不到的，或者是一些不该用这个概念解决的问题。

这就是为什么操作系统课学起来那么难——很多都是没道理的难。

再加上 Unix 系统里面一堆设计恶劣，无法有效组合使用的工具软件，操作系统就在学生心中产生了威慑力。死记硬背，喜欢折腾，喜欢发现奇技淫巧的人，在这个领域里茁壮成长。逐渐的，他们产生了莫名的自信。他们并不理解里面的很多概念是怎么来的，只是记住了它们，他们写的代码很难看懂。然后他们开始从心理上打压那些记不住这些概念，看不懂他们代码的人。

久而久之，这些人就成为了大家所崇拜的“神”。

跟有些人聊操作系统是件闹心的事，因为我往往会抛弃一些术语和概念，从零开始讨论。我试图从“计算本质”的出发点来理解这类事物，理解它们的起因，发展，现状和可能的改进。我所关心的往往是“这个事物应该是什么样子”，“它还可以是什么（也许更好的）样子”，而不只是“它现在是什么样子”。不明白我的这一特性，又自恃懂点东西的人，往往会觉得我连基本的术语都不明白。于是天就这样被他们聊死了。

幸运的是我有几个聊得来的朋友，他们不会那么教条主义。于是今天我跟一个朋友在微信上聊起了“微内核 vs 宏内核”这件事。其实这个问题在我脑子里已经比较清楚了，可是通过这些对话，我学到了新的东西。这些东西是我们在对话之前可能都没有完全理解的，也许很多其他人也没理解。所以我觉得可以把这些有价值的对话记录下来。

我不想从头解释这个事，因为你可以从网络上找到“微内核”和“宏内核”的设计原理。我想展示在这里的只是我们的对话，里面有对也有错，翻来覆去的思想斗争。对话是一个很有意思的东西，我觉得比平铺直叙的文章还要有效一些。

对话

好了，现在开始。对话人物“WY”是我，“LD”是我的一个朋友。

（8月19日，开始）

WY：好多年没折腾 OS，现在再折腾应该有新的发现。这篇 [paper](#) 说 Minix 3 比 Linux 要慢 510%。

WY：通常的定义，说微内核只需要 send 和 receive 两个系统调用。你不觉得有问题吗？其实函数调用的本质就是 send（参数）和 receive（返回值），但只有这两个系统调用，这种做法是过度的复用（multiplex）。

（下载 Minix 3 源代码看了一会儿。上网搜索关于微内核的资料……）

LD：是。

LD：一个外设产生了中断，中断管理进程接收到到中断，发一个消息给相应的设备驱动进程，这个进程处理中断请求，如果设备驱动有 bug，挂了，也不会干扰 OS。这就是微内核逻辑。

WY：微内核似乎一直没解决性能问题。后面的 L4, QNX... 把 sever 隔离在不同的地址空间似乎是个最大的问题。

LD：导致通讯成本特别大。本来传递个地址就可以的事。现在要整个复制过去。

WY：地址空间不应该分开。或者也许可以在 MMU 上面做文章，传递时把那片内存给 map 过去。这样上下文切换又是一个开销……函数调用被搞的这么麻烦，微内核似乎确实是不行。对了，微内核服务调用时会产生进程切换吗？

LD：会，按照微内核的定义，每一个基本单元都是一个进程。

WY：完蛋了。

LD：内存管理是一个进程，IO 管理是一个进程，每个设备驱动是一个进程，中断管理是一个进程。

WY：进程切换的开销……

LD：为了降低进程间通信开销，所以定义了 L4。我也不太懂这个有啥用。

WY：改善的是通信开销，但仍然有进程切换开销。我刚看了一下 L4，它是从寄存器传值，但是进程切换会把寄存器都放到内存吧。

LD：对呀，所以 L4 意义似乎不大。

LD：带“微”的除了微软和微信，没一个成功的。

LD：最近流行的所谓微服务。

WY：驱动的 bug 应该有其他办法。

LD：现在的 OS 的问题，就是内核微小的错误，都是让整个系统挂掉。这和我们写软件应该用多进程还是多线程，同样的问题。

WY：应该从硬件底层彻底抛弃现在的进程切换方式。保存的上下文太多。

LD：现在 OS 不是分成 user 和 kernel 保护级别么。我觉得再增加一个两个保护级别，专门针对设备驱动程序似乎是更好的选择。

WY：我以前设想一个办法可以完全不需要保护级别，而且不需要虚拟内存。

LD：怎么办？编译器静态分析搞定？Rust？

WY：完全使用实地址，但是代码无法访问对象外面的内存。

LD：靠编译器保证？

WY：不需要多先进的编译器。语言里面没有指针这东西就行，这样你没法访问不是给你的对象。嗯，需要抛弃 C 语言……

LD：Rust！

WY：还用不着 Rust。其实 JVM 早就是那样了。只不过通常不认为 JVM 是一个操作系统，但操作系统完全可以做成那样。

LD：所谓对象，就是每次地址访问，除了地址还有一个 size？超过 size 不允许？还是编译器确保一定不会超过 size？

WY：你在 Java 或者其它高级语言比如 Python... 都没法访问对象外面的内存啊。只有 C 可以，因为 C 有指针，可以随便指到哪。

LD：是的。C 这种方式，就是天天在没有护栏的桥上走来走去。除了越界访问，还有一个问题，就是多个 task 同时改一块内存。

WY：然后为了防止越界，有了“进程”，“虚拟地址”这种概念。

LD：虚拟地址，还是为了用虚拟内存。

WY：虚拟地址，虚拟内存就是为了隔离。每个进程都以为地址从0开始，然后本来很容易的函数调用被隔离开了。如果改变了这个，微内核就真的可以很快了。实际上内核就不存在了…… 哦，还是有。就只剩下调度器，内存管理。

IPC 没了，被函数调用所取代。

LD：换个思路。其实 OS 最容易出问题的是硬件驱动，所以尽量让硬件标准化，别每个硬件都搞一套自己的驱动。让一套驱动支持多种硬件，问题就解决了。比如 usb 驱动。完全可以做到一类硬件都用一个设备驱动。

WY：我还是觉得驱动程序 bug 其实可以不导致当机。用内核线程行不行？共享地址空间，但是异步执行。

LD：Linux 似乎就是这样。tasklet，可以被调度的。

WY：所以驱动程序要是当掉，可以不死对吗？我回去查一下。

LD：看啥错误了。不小心修改了其它模块的内存就完蛋了。其它错误最多硬件本身不能用了。

WY：所以就是为什么你说再多一个保护级别。

LD：嗯，别碰了内核关键的代码。但是驱动之间还是可以互相干扰的。

WY：是个不错的折中方案。所以微内核解决了一个不是那么关键的问题。

LD：是的。这个问题不重要。哦，对了，Windows 是微内核的。好像从 2000 开始。

WY：只是号称吧。Mac OS X 不是号称 Mach 微内核加 BSD 吗？

LD：对。MacOS 也是微内核。

WY：那他们怎么解决的性能问题呢？

LD：不知道。Windows 蓝屏可不少，显然没做到完全隔离。至于 Mac，不清楚为啥那么稳定。

WY：根据我们之前的讨论，Mac 微内核可能是假的。Mac 稳定是因为它的 driver 就没几个吧，硬件都是固定选好的。

LD：嗯，也是稳定的主要原因。

WY：这个英明了…… 而且看来微内核在集群方面也没什么用处。

LD：集群，每个计算机是一个 node。挂了也不怕。

(8月20日继续讨论)

WY：我发现这个 [paper](#).....

WY：这东西叫 L4Linux，就是 Linux 跑在 L4 微内核上。比起纯 Linux，开销只有 5%

WY：代码在这里：<http://os.inf.tu-dresden.de/L4/LinuxOnL4>

WY：L4 的做法是 1) 小参数用寄存器传递，不切换某些寄存器。2) 大型参数把内存映射到接收进程，跟我之前设想的一样。这样避免了拷贝。然后采用了“direct process switch”，“lazy scheduling”降低了调度开销。现代处理器的 tagged TLB 之类也大大降低了进程切换开销。

L3 therefore uses a new method based on temporary mapping: the kernel determines the target region of the destination address space, maps it temporarily into a *communication window* in the source address space, and then copies the message directly from the sender's user space into its communication window. Due to aliasing the message appears at the right place in the receiver's address space.



Figure 4: Direct Message Copy

WY：上图是 direct message copy。先把接收进程的目的地址映射到发送进程的地址空间，然后发送进程往里拷贝。所以其实仍然有一次拷贝，并不像我理想的 OS 那样直接就能传递对象引用，完全不用拷贝。Pass-by-value vs pass-by-reference。但这比起 Linux 似乎开销是一样的。

LD：微内核好处真的很大么？

WY：好处就是微内核的好处，隔离。可能看各人需求了。一个 99.99% 可靠的系统和一个 99.999999% 可靠的系统的差别？

WY：不过似乎高可靠需求都去用 vxworks 之类的了

(上网查询 vxworks.....)

WY : 原来 vxworks 也是微内核。

WY : 5% 的开销还可以接受..... 进程切换开销貌似没有提，用的地址映射方法。

LD : cross address space

WY : 刚买了个tplink 路由器，里面跑的 vxworks。

LD : tplink 不是 Linux ?

WY : 新的tplink AC1900，改成了 vxworks。Airport Extreme 也是 vxworks。

LD : why ?

WY : 实时，可靠性高吧。

LD : 可靠性应该是最高的之一。卫星、武器都用。

WY : 波音 787 也用这个，各种火星车..... 貌似还是说明一些问题。

WY : 还有个 [GreenHills Integrity DO-178B](#) 实时操作系统。F35 用的。

WY : Much of the F-35's software is written in C and C++ because of programmer availability; Ada83 code also is reused from the F-22. The Integrity DO-178B real-time operating system (RTOS) from Green Hills Software runs on COTS Freescale PowerPC processors.

WY : Freescale PowerPC...

LD : 我们的一个mcu 就是 Freescale 的 PowerPC

LD : 有个叫“rtems”的os，我一直很关注。

WY : 摘自 Integrity DO-178B RTOS :

Safe and secure by design

- RTOS designed for use in reliable, mission critical, safety critical and secure (MILS & MLS) applications
- Based on modern microkernel RTOS design
- Fast, deterministic behavior with absolute minimum interrupt latencies

WY : Integrity 也是微内核。看来微内核是可靠一些，属于在 C 语言框架下的一个不错的折中方案。

.....

(如果你有什么不同意见，欢迎联系我。如果觉得有帮助，可以考虑[付费](#))

再谈“P vs NP”问题

谨以此文献给“最伟大的计算机科学家”

好几年前曾经写过一篇文章表达对计算机科学里著名的“[P vs NP](#)”问题的看法。当时正值我人生中第 N 次研究那些东西，由于看透了却不在乎，所以写得特别简略。没想到有人看到后，还以为我没仔细学过复杂度理论，说我信口开河。我一般懒得谈论这种太理论的问题，身边也很少有人关心，所以后来干脆把文章撤了。不是我说的有什么不对，而是我懒得跟人争论。

没想到最近又遇到有人抓住我删掉的文章，乘机拿出来贬损我，尽其羞辱之能力。说王垠你太自以为是了，你成天写那些博客，有什么价值吗？你居然连“P vs NP”都敢批。你知道“P vs NP”要是解决了，世界将有天翻地覆的变化，多少的计算难题会被解决，机器学习都没必要了，非对称加密全都被破解……跟上课似的头头是道滔滔不绝，几乎把他本科算法课本上的内容给我背了一遍，以为别人不知道一样，却没有显示出任何他自己的思想。

呃，我真是服了某些人背书冒术语的能力，难怪能做国内某大厂的 P10（注：不是我的在职公司）。鉴于很多人对此类问题的一知半解，反倒嘲笑别人不懂，牛逼轰轰打压其他人，我决定事后把这个问题再详细讲一下，免得以后还要为它费口舌。

对于初学者这篇文章有点门槛，需要学习一些东西。“[P vs NP](#)”问题属于计算理论（Theory of Computation）的一部分——复杂度理论。计算理论不止包括复杂度理论（Complexity），还包括可计算性（Computability），也就是“停机问题”一类的内容。

国内大学的计算机教学一般在算法课上对复杂度理论有初级的讲授，但很少人能够真的理解。如果你没有系统的学习过复杂度理论，我建议你研读一下计算理论的专著（而不是普通的算法教材），比如 Michael Sipser 的『[Introduction to the Theory of Computation](#)』。

我当年在 Indiana 做研究生计算理论课助教的时候，可算是把这书给看透了……被逼的。其中“可计算性理论”在我将来的 PL 研究中起了比较大的启发作用，而复杂度理论的用处一般。我觉得 Sipser 的书写的不够清晰透彻，但很多学校拿它做教材，好像也没有其它特别好的替代品。

计算理论如此晦涩难懂，我认为图灵机是祸首。如果你能理解 lambda calculus，将会大大简化理解计算理论的过程。如果你想用更深刻更容易的方法理解计算理论，可以参考[这篇文章](#)的“Lambda 演算与计算理论”一节，里面会提到另一本参考书。从这篇文章你也可以看出来，我丝毫不崇拜图灵。

“P vs NP”真的重要吗？

“P vs NP”这个问题有它的理论价值，它是有趣的问题，里面的有些思路有启发意义，值得花些时间来了解。但计算机科学界长久以来都严重夸大它的重要性，把一个很普通的问题捧上了天，吹得神乎其神。

再加上图灵机模型在计算理论界的广泛使用，使得这门学问显得异常艰深。很多人看到图灵机就晕了，在课程上蒙混过关，考试完了就全忘了，根本无法理解里面的实质内容。正是因为很多人的不明觉厉，使得“P vs NP”登上了它在 CS 界的宝座。

很多人做了一辈子计算机工作，做了很多巧妙的设计构架，写了许许多多的代码，解决了很多性能难题。提到“P vs NP”，虽然一辈子都没用上这个理论，仍然顶礼膜拜。由此可见“不明觉厉”对于人们心理的威力。

很多人认为“P vs NP”是计算机科学最重要的问题。Clay 数学研究所甚至悬赏一百万美元解决这个问题，把它叫做数学界的 7 个千年难题之一，跟黎曼猜想并列其中。

好几次有人声称解决了“P vs NP”，上了新闻，闹得舆论沸沸扬扬，小编们吹得好像世界要天翻地覆了一样，把他们追捧为天才苦行僧，后来却又发现他们的结果是错的……

如果你真的理解了“P vs NP”的内涵，就会发现这一切都是闹剧。这个问题即使得到解决，也不能给世界带来很大变化。解决这个问题对于现实的计算，作用是微乎其微的。不管 P 是否等价于 NP，我们遇到的计算问题的难度不会因此有重大改变。

甚至有些数学家认为“P vs NP”根本没有资格跟黎曼猜想一起并列于“千禧年问题”。我倒是希望有人真的解决了它，这样我们就可以切实的看到这有什么意义。

“P vs NP”也许不是愚蠢的问题，但计算机科学界几十年来夸它的重要性的做法，是非常愚蠢的，让整个领域蒙羞。

真正重要的数学问题被解决，应该对现实世界具有强大的作用。这种作用可以是“潜在的”，它的应用可以发生在很久以后的将来，但这必须能够被预见到。数学家们把这叫做“applicable result”（注意不叫 applied 或者 practical）。否则这个数学问题就只能被叫做“有理论价值”，“有趣”，而不能叫做“重要”。即使所谓“纯数学”，也应该有可以预见的效果。

很多数学家都明白黎曼猜想（Riemann hypothesis）的重要性。大数学家希尔伯特说过：“如果我沉睡了三千年醒过来，我的第一句话会是‘黎曼猜想被解决了吗？’”假设希尔伯特还在世，他会对解决“P vs NP”有同样的渴望吗？我觉得不会。实际上，很多数学家都觉得“P vs NP”的重要性根本没法和黎曼猜想相提并论，因为我们预见不到它会产生任何重要的效果。

什么是多项式时间？

很多人提到“P vs NP”就会跟你吹嘘，P 如果等于 NP，世界将有天翻地覆的变化。许许多多我们以前没法办到的事情，都将成为现实。非对称加密技术会被破解，生物化学将得到飞跃，机器学习将不再有必要……

这些人都忽略了一个重要的问题：什么是多项式时间。盲目的把“多项式”等同于“容易”和“高效”，导致了对“P vs NP”重要性的严重夸大。

n^{100} 是不是多项式？是的。 $n^{1000000}$ 也是多项式。 $n^{100^{100}}$ 也是多项式， $n^{100^{100^{100}}}$ 也是多项式……实际上，只要 n 的指数是常数，它就是一个多项式，而 n 的指数可以是任意大的常数！n 的指数可以是任意大的常数！n 的指数可以是任意大的常数！重要的事情说三遍。

时间复杂度 $n^{100^{100^{100}}}$ 的算法，能用吗？所以即使 P=NP，你需要的计算时间仍然可以是宇宙毁灭 N 次，其中 N 是任意的常数。

说到这里，又会有人跟我说你不懂，当 n 趋近于无穷的时候，非多项式总会在某个时候超越多项式，所以当 n “足够大”的时候，多项式时间的算法总是会更好。很可惜，“无穷”对于现实的问题是没有意义的。任何被叫做“重要”的问题，都应该在合理的时间内得到结果。

我们关心的要点不应该是“足够大”，而是“具体要多大”。精确的量化，找到实际可以用的区间，这才是合格的科学家该有的思路。计算机科学里，大 O 表示法泛滥成灾，只看最高次幂，忽略系数和常数项，也是常见的误区。我也曾经沉迷于如何把 $O(n^3)$ 的算法降低到 $O(n^{2.9})$ ，现在回头才发现当年是多么的幼稚。

“多项式时间”这个概念太宽泛太笼统。以如此笼统的概念为基础的理论，不可能对现实的计算问题产生意义。我们关心的不应该仅仅是“是否多项式”，而是“具体是什么样的多项式”。 $6n^{20} + 26n^7 + 200$, $1000n^3 + 8n^2 + 9$,每一个多项式的曲线都是很不一样的，在各个区间它们的差别也是不一样的。多项式的幂，系数，常数项，它们的不同都会产生重大的差异。

这就是为什么“P=NP”没有很大意义，因为 P 本身太笼统，其内部的差异可以是天壤之别。与其试图笼统的证明 P 等价于 NP，还不如为具体的问题想出实质意义上高效的算法，精确到幂，系数，常数项。

更进一步看，这些“复杂度”的数学公式，不管是多项式还是指数，不管你的幂，系数常数项有多精确，终究难以描述现实系统的特性。物理的机器有各种分级的 cache，并行能力，同步开销，传输开销，各种瓶颈.....最后你发现性能根本无法用数学公式来表达，它根本不是一个数学问题，而是一个物理问题，工程问题。这就像汽车引擎的功率一样，只有放到测试设备（Dyno）上面，通过系统的测量过程才能得到。

有些理论家喜欢小看“工程”，自以为会分析复杂度就高高在上的样子，而其实呢，工程和物理才是真实的。数学只是粗略描述物理和工程的工具。

P!=NP 有意义吗？

“P vs NP”问题有两种可能性：P=NP（等价），或者 P!=NP（不等价）。以上我说明了 P=NP 的意义不大，那么要是 P!=NP 呢？

很多人会跟你说，要是一个问题是 NP-Hard，然后又有 P!=NP，那么我们就知道这个问题没有多项式时间的算法存在，就避免了为多项式时间算法浪费时间了。这不也有一些价值吗？

我并没有否认 P!=NP 是有那么一点价值：在某些时候它也许避免了浪费时间。但这种价值比较小，而且它具有误导性。

一个常见的 NP-Hard 问题是 SAT。如果 P!=NP，那么大家就应该放弃为它找到高效的算法吗？如果大家都这样想，那么现在的各种高效的 SAT solver 就不存在了。实际上，利用随机算法，我们在大多数时候都能比较快的解决 SAT 问题。

问题在于，“P vs NP”关心的只是“最坏情况”，而最坏情况也许非常罕见。有些问题大部分实际的情况都可以高效的解决，只有少数变态的情况会出现非常高的复杂度。为了这少数情况放弃大多数，这就是“P vs NP”的误导。

如果因为 P!=NP，你认为 NP-Hard 的问题就没有高效的算法，那你也许会误以为你可以利用这些“难题”来做非对称加密。然而 NP-Hard 并不等于没法快速解决，所以要是你因此被误导，也许会设计出有漏洞的加密算法。

即使 P!=NP，我们仍然不能放弃寻找重要的 NP-Hard 问题的高效算法，所以确切的证明 P!=NP 的价值也不是那么重要了。其实你只要知道 P=NP “大概不可能”，就已经能起到“节省时间”的目的了。你没必要证明它。

什么是 NP？

这一节我来讲讲“P vs NP”里的“NP”到底是什么。内容比较深，看不懂的人可以跳过。

很多人都没搞明白 NP 是什么就开始夸夸其谈“P vs NP”的价值。经常出现的错误，是把 NP 等同于“指数时间”。实际上 NP 代表的是“Nondeterministic Polynomial time”，也就是“非确定性图灵机”(nondeterministic Turing machine)能在多项式时间解决的那些问题。

什么是“非确定性图灵机”？如果你把课本上那堆图灵机的定义看明白看透了，然后又理解了程序语言理论，你会发现所谓“非确定性图灵机”可以被很简单的解释。

你可以把我们通常用到的程序看作是“确定性图灵机”(deterministic Turing machine)。它们遇到条件分支，在同一个时刻只能走其中一条路，不能两边同时探索。

那么“非确定性图灵机”呢？你可以把“非确定性图灵机”想象成一个具有“超能力”的计算机，它遇到分支语句的时候，可以同时执行 True 和 False 两个分支。它能够同时遍历任意多的程序分支，这是一台具有超能力的机器！

所以“P vs NP”的含义大概就是这样：请问那些需要非确定性图灵机（超能力计算机）在多项式时间才能解决的问题，能够用确定性图灵机（普通计算机）在多项式时间解决吗？

现在问题来了，具有如此超能力的机器存在吗？答案当然是“No！”就算是量子计算机做成功了，也不可能具有这样的计算能力。没有人知道如何造出非确定性图灵机，人们没有任何头绪它如何能够存在。

所以“P vs NP”这个 问题的定义，是基于一个完全假想的机器——非确定性图灵机。既然是假象的机器，为什么一定要是“非确定性图灵机”呢？为什么不可以是其它具有超能力的东西？

仔细想想吧，“非确定性图灵机”对于现实的意义，就跟 Hogwarts 魔法学校和哈利波特对于现实的意义一样。我们为什么不研究“P vs HP”呢，其中 H 代表 Harry Potter。HP 定义为：哈利波特能够在多项式时间解决的问题。

“P vs NP”问题：请问那些需要非确定性图灵机（超能力计算机）在多项式时间才能解决的问题，能够用确定性图灵机（普通计算机）在多项式时间解决吗？

“P vs HP”问题：请问那些需要哈利波特在多项式时间才能解决的问题，能够用确定性图灵机（普通计算机）在多项式时间解决吗？

我不是开玩笑，仔细回味一下“P vs NP”和“P vs HP”的相似性吧。也许你会跟我一样意识到 NP 这个概念本身就是虚无的。我不明白“一个不存在的机器能在多项式时间解决的问题”，这样的说法有何意义，基于它的理论又有什么科学价值。

非确定性图灵机存在的意义，也许只是因为它可以被证明等价于其它一些常见的问题，比如 SAT。计算理论书籍一般在证明 SAT 与 非确定性图灵机等价性之后，就完全抛掉了非确定性图灵机，之后的等价性证明都是通过 SAT 来进行。

我觉得 NP 这个概念其实是在故弄玄虚。我们完全可以从 SAT 本身出发去发展这个理论，而不需要设想一个具有超能力的机器。我们可以有一个问题叫做“P vs SAT”，而不出现 NP 这个概念。

(有点扯远了)

其它质疑 P vs NP 价值的人

有人认为我质疑 P vs NP 的价值是一知半解信口开河，然而我并不是第一个质疑它的人。很多人对 P vs NP 都有类似的疑惑，但因为这个问题的地位如此之高，没人敢站出来。只要你开口，一群人就会居高临下指责你基础课程没学好，说你眼界太窄……再加上那一堆纷繁复杂基于图灵机的证明，让你有苦说不出。

由于这个原因，我从来没敢公开表达我的观点，直到我发现 Doron Zeilberger 的这篇[文章](#)。Zeilberger 是个数学家，Rutgers 大学的数学系教授。在那之前他开了个玩笑，戏称自己证明了 $P=NP$ ，还写了篇像模像样的论文。在文章里他告诫大家：不要爱上你的模型 (Don't Fall In Love With Your Model)。他这句话说到了我心里。

你还能在网络上找到其它人对“P vs NP”的质疑，比如这篇来自于一位专门研究计算理论的学者：

[Is P=NP an Ill Posed Problem?](#)

我觉得他讲的也很在理。正是在这些人的鼓舞之下，我随手写出了之前对“P vs NP”的质疑。只言片语里面，融入了我多年的深入学习，研究和思考。

总结

看这篇文章很累吧？我写着也累。对于我来说这一切都已经那么明了，真的不想费口舌。但是既然之前已经说出来了，为了避免误解，我仍然决定把这些东西写下来摆在这里。如果你暂时看不懂可以先放在一边，等到了需要深入研

究计算理论，想得头痛的时候再来看。你也许会感谢我。

我希望严谨的计算机科学工作者能够理解我在说什么，反思一下对“P vs NP”的理解。计算机专业的学生应该理解“P vs NP”理论，但不必沉迷其中。这并不是一个值得付出毕生精力去解决的问题。计算机科学里面还有其它许多有趣而重要的问题需要你们去探索。如果你觉得计算机科学都不过瘾，你可以去证明黎曼猜想啊 :)

当然所有这些都是我的个人观点，我没有强求任何人接受它们。强迫别人接受自己的观点是不可以的，但想阻止别人表达对此类问题的质疑，也是不可以的，因为我们生活在自由的世界。

没人想抢走你们的玩具，但不要忘了，它只是玩具。

学习的智慧

有些人很爱学习，兢兢业业把书一个字一个字从头看到尾。好不容易学完一本书，却不知道自己学到了什么。

另外一些人聪明一点，他们嘴里喜欢冒出各种术语，听得别人头都冒汗。等遇到实际问题的时候，你就发现他们虽然胸有成竹的样子，做事动作快，却把握不准方向。

而垠神呢，更奇葩。垠神身边的人常发现他问一些很傻的“初学者”问题，简直让人不屑。遇到术语名词丈二和尚张冠李戴，好像不知道那些是什么。垠神居然什么都不会！

每次到了需要作出关键决策的时候，垠神默默听完大家正儿八经滔滔不绝之后，有时会不经意抖出一句：“那看起来是xx……那样那样弄一下，就可以了。”你起初不信他，跟他争论，说这样不能满足我们的宏伟目标。他又轻描淡写跟你说一些，然后回头玩他的去了。

他的话被你当耳边风，你坚信自己是对的。几个月之后，经过实现N种方案，各种教训之后，你发现自己最后选择了垠神最初指出的方向。如果你开头就试图理解他在说什么，可能几天就完工了。

在 Indiana 的时候，垠神经常享受的一件事情，就是静静看着同学们喊着各种口号和术语，眼睁睁看着他们误入歧途，重蹈自己几年前犯过的错误。甚至有些人弄了一两年都没发现是死路一条，还继续在垠神面前手舞足蹈。

不是垠神自私，而是很多人根本没有在意过他的看法，甚至没给他发言的机会。如果有人滔滔不绝，垠神就懒得去插嘴。如果有人如此急切的证明自己是对的，垠神总不至于热心到想打断他，讲述自己在同一路线的失败经历吧？

死知识，活知识

很多人坚信“知识就是力量”，可是他们不知道，知识和知识是不一样的。

大部分人从学校，从书籍，从文献学知识，结果学到一堆“死知识”。要检验知识是不是死的，很简单。如果你遇到前所未见的问题，却不能把这些知识运用出来解决问题，那么这些知识就很可能是死的。

死知识可能来源于真正聪明的人，但普通人往往是间接得到它。从知识的创造者到你之间，经过了多次的转手倒卖。就算你直接跟知识的鼻祖学习都不容易得到真传，普通人还得经过多次转手。每一次转手都损失里面的信息含量，增加“噪音”，甚至完全被误传。所以到你这里的时候，里面的“信噪比”就很低了。这就是为什么你学了东西，到时候却没法用出来。

追根溯源之后，你会发现这知识最初的创造者经过了成百上千的错误。这就像爱迪生发明灯泡，经过了数千次失败的实验。知识的创造者把最后的成功记录在文献里发表，然后你去读它。你以为得到了最宝贵的财富，然而最宝贵的财富却是看不见的。作者从那成百上千的失败中得到的经验教训，才是最宝贵的。而从来没有人把失败写下来发表。

没有这些失败的经验，你就少了所谓“思路”，那你是不大可能从一个知识发展出新的知识的。就像你读了别人的重要 paper，你是不大可能由此发展出重大想法的。你的 paper 会比别人低一个档次，往往只能修修补补，弄出一个小点的想法。而原来的作者以及他的学生们，却可以很容易的变出新的花样，因为他们知道这些路是怎么走过来的，知道许许多多没有写下来的东西。“失败是成功之母”，在我脑子里就是这个意思。

垠神从很早的时候就知道了这个道理，所以他很多时候不看书，不看 paper。或者只看个开头，知道问题是什么。他看到一个问题，喜欢自己想出解决方案。他不是每次都成功，实际上他为此经历了许许多多的失败。运气好的时候，他得到跟已有成果一样的结果。运气再好一点的时候，他得到更好的结果。但他关心的不只是成功，中间的许多失败对他也是价值重大的。

然后他会去找有经验的人讨论，这些人也许很厉害，早就做过深入的研究。也许是初学者，刚刚接触到同样的问题。但很奇特的是，不管跟什么样的人交流，垠神几乎总是能得到启发。即使这个人什么都不懂，现教给他也一样。通过向不懂的人解释这个问题，他经常意外的发现问题的答案。

死知识是脆弱的。面对现实的问题，死知识的拥有者往往不知所措，他们的内心充满了恐惧。他们急于证明自己的能力，忙于维护各种术语和教条。因为这不是他们自己的思想，他们只能抬出权威来镇压大家：这个理论是某某大牛提出的，所以肯定能解决问题！

为死知识引以为豪的人往往满口的术语，对“初级问题”不屑一顾。懂得活知识的人，却知道每一个初级甚至傻问题的价值。世界上最重大的发现，往往产生于对非常基础的问题的思考，比如“时间是什么？”如果你觉得理所当然每个人都该知道这个问题的答案，只有白痴才会问出这种问题，那你就失去了很多产生生活知识的机会。这就是为什么垠神常问一些基础问题，因为他想知道它们背后还隐藏着什么他不知道的内涵。

这就是垠神获取活知识的秘密。活知识必须靠自己创造出来，要经过许许多多的失败。如果没有经过失败，是不可能得到活知识的。

由于活知识是自己创造的，其中包含的概念，垠神是不知道它们在文献中的术语——垠神平时都懒得看文献。这就是为什么很多人跟垠神交流，发现他连基本的术语都不知道是什么。经过进一步交流，你也许会发现虽然垠神不知道

一个东西的名字，他却知道这个东西是什么——以他自己的理解方式；）

知识的来源

所以呢，知识的来源最好是自己的头脑，但也不尽然。有些东西成本太高，没条件做实验就没法得到，所以还是得先获取现成的死知识。

有些人说到“学习”，总是喜欢认认真真上课，抄笔记，看书。有些人喜欢勾书，把书上整整齐齐画满了横杠。兢兢业业不辞辛苦，最后却发现没学会什么。

为什么会这样呢？首先因为他们没有理智的选择知识的来源。其次，他们不明白如何有效的“提取”知识。这第一点属于“品位”问题，第二点则属于“方法”问题。

很多人没有意识到，对于同一个问题有很多不同的书，不同的作者对于问题的见解深度是不一样的。如果你拿着一本书从头看到尾，而不参考其他人的，往往会误入歧途。你手上的书的作者，也许自己没把这问题研究很透。只是他发表的早，占了先机，所以这书成了学校指定的，大家推崇的“经典教材”。

在学校的时候，我不止一次的发现经典教材很难懂。经过努力，让自己的思维爬到一定高度之后我才发现，原来这经典教材作者很多地方没有看透彻。写书的时候他也把一些可有可无的内容写进去，引经据典的罗列出各种 paper，却忽视了最重要的思想和直觉。看这种书，你当然头痛了。

所以我喜欢在网上搜索对应一个主题的内容，往往能发现一些名不见经传人的作品，反而比写书的“大牛”来的深刻。当然网上内容鱼龙混杂，你也不要死钻进去出不来了。

看书的时候不要老想从头看到尾。如果一个主题你看得头大，最好的办法是放下这书，去寻找对同一主题的更简单的解释。这些东西可以来源于网络，也可以来自其它书籍，也可以来自身边的人。同时保留多个这样的资源，你就可以对任何主题采用同样的“广度优先”搜索，获得深入理解的机会就会增加。

都说书籍是人类的朋友，我却发现看书是很闷的事情，我很不喜欢看技术方面的书。我最喜欢的是直接跟人学东西。找到懂一点的人，跟他聊。别管他懂多少，懂多深，我发现真人几乎总是比书好。至少，你聊天的时候不会打瞌睡；而且很多时候他没告诉你答案，但通过聊天，你自己把它给想出来了。

参加学术会议的时候，我会事先把会议的 paper 浏览一下，然后发现根本看不懂。带着好奇心来到会议，听了演讲还是不懂。接下来我使出绝招……等演讲者下台之后的休息时间，我会走到他面前说：“你好，我比较笨看不懂你的 paper。请问你能在三句话之内把里面的要点概括一下吗？”接下来奇迹发生了，作者说出了他从未发表的直觉，仔仔细细教会了我，甚至跟我成了朋友。当然对于这样的人，我也会告诉他一些我知道的东西作为回报。

英语的重要性

关于学习，我最后想提醒大家的是英语的重要性。很多人英文不够好，对看英文材料有畏惧心理，只看中文内容，这使得他们很难得到准确的信息，经常被人误导，被收智商税。

我从大学年代开始就很少看中文内容了。专业书籍，技术文档，全部都看英文的。现在没那么排斥中文了，然而看中文网站的时候仍然发现很多误导。国产电视剧也大部分是各种脑残剧情，误导人们的三观。

不是我崇洋媚外，可是实话说，这几年中文内容虽然改进了很多，可是很多方向上的专业程度还是比英文的低很多，很多不准确甚至根本就是错的。所以虽然我平时说话用中文，写东西用中文，却很少看中文的东西。我看的中文内容大部分是人文的，小说一类的。

中文信息经常包含各种误导，危言耸听，造成了人们生活中不必要的麻烦。手机放枕边说有辐射，充电器用完不拔说会爆炸，被鱼刺扎了不敢自己弄下去，医院的椅子不敢坐说会传染皮肤病，不要喝“阴阳水”，不要吃这不要吃那全都有害，快点贷款买房快点结婚生孩子……各种事实上观念上文化上的误导，导致了许多国人生活方式的困窘。

中国小孩子从小就学英语，到了关键时候却从来不用。我不排斥看中文内容，但我建议不要片面的只看中文内容。事无巨细都应该同时参考英文信息，多方面分析之后再做决定。生活的决策如此，专业知识的学习当然也一样。对于同一个知识点，看到中文的时候你最好搜索它的英文，对比各种资料，这样你就更容易得到准确的信息。

解谜英语语法

我发现很多人仍然在为语法的枯燥繁琐而头痛。市面上好像不存在一本深入本质的语法教材。语法对于我来说已经早就不是问题，所以我萌生了写这样一篇文章的念头，帮助那些正在为学习语法而痛苦挣扎的人们。

这篇文章里包含了一些我自己保留多年的关于英语学习的秘密。我曾经想过把这写成一本完整的语法书，可是后来发现似乎一篇文章足矣。

句子的核心地位

直到几百年前，各个不同大陆上的人还从来没见过面，他们的语言里却不约而同出现了同样的结构：句子。这似乎说明句子的出现是一种自然规律，必然结果，而不只是巧合。

句子是人类语言最核心的构造。为什么呢？因为人和人说话终究是为了一个目的：描述一件事。

这件事也许只有一个字：吃！

也许可以很长：昨天晚上在上海某路边餐厅吃的鹅肝，是我吃遍全世界最好的。

一个句子表达的就是一件事，或者叫一个“事件”。人与人交流，无非就是讲述一个个的事件。

许多人都学英语，一来就背单词，背了很多单词，仍然写不出像样的句子来。只见树木不见森林，因为他们没有意识到句子才是最关键的部分。我们应该一开头就理解句子是什么，如何造出句子，而不是背单词。单词是树木，句子才是森林。

你需要的能力

所以掌握一门语言，基本就是要掌握句子。有了句子就有了一切。

掌握句子包括两种能力：

1. 能够迅速地造出正确的句子，准确地表达自己的意思。
2. 能够迅速地分析别人的句子，准确地理解别人的意思。

这两件事，一个是表达（发送），一个是理解（接收）。因为语言是沟通（或者叫“通讯”）的工具，所以它就只包含这两件事。

句子的本质

假设我们是原始人，还没有语言。我想告诉同伴“我吃苹果”这件事，该怎么表达呢？没有语言，那我可以先画个图嘛：



画图是很麻烦的，笔画太多不说，还可能有歧义。到后来，部落里的人聪明了一点，发明了“符号”这种东西，只需要几笔就能表示一个概念。他们给事物起了简单的符号名字，不再需要画图了。于是我们有了 I, apple 这样的词用来指代事物。有了 eat 这样的词，用来代表动作。所以画面变成这个样子：



后来干脆连框也不画了，直接写出这些符号来，这就是我们现在看到的“句子”：

I eat apples.

注意，虽然没有了上面的框图，这句话其实隐含了这幅图。写这个句子的人假设阅读者能够从一串符号还原出一个画面（或者叫结构）来。

有些人不能理解别人的话，看书看不懂，就是没能从符号还原出结构来。很多语法书列举出千奇百怪的“组合情况”，为的只是帮助你从这串符号还原出结构来。在现代语言学和计算机科学里面，这个过程就叫做“语法分析”（parsing）。

动词是句子的核心

那么，你觉得“我吃苹果”这个事，里面最关键的部分是什么呢？是“我”，“苹果”，还是“吃”呢？

稍微想一下，你也许会发现，关键在于“吃”这个动作。因为那是我和苹果之间发生的事件。这句话是说“吃”这件事，而“我”或者“苹果”，只是“吃”的组成部分。

用 eat 这个词，你不但可以表达“我吃苹果”，还可以表达“他吃面条”，“猫吃老鼠”之类的很多事情。于是，聪明一点的人就把 eat 这个词提取出来，做成一个“模板”：



这个模板就是所谓“动词”。eat 这个动词给你留下两个空，填进去之后，左边的东西吃右边的。

句子是语言的核心，而动词就是句子的核心。动词是事件的关键，比如 eat。

A eat B.

我们可以选择空格里的 A 或者 B 是什么。但不管怎么换，事情仍然是“吃”。为了描述方便，我们把 A 和 B 这两个空格叫做参数（parameter）。

这跟数学函数的参数（ $f(x)$ 里面那个 x ）类似，也跟程序函数的参数类似。用数学或者程序的方式来表示这个句子，就是这样：

eat(A, B)

其中 A 和 B，是动作 eat 的参数。我只是打个比方帮助你理解，当然我们不会这样写英语。如果你完全不懂数学或者编程，可以忽略这个比方。

动词决定了它可以有几个参数，它们可以在什么位置，参数可以是什么种类的成分。比如 eat，它可以有两个参数。这两个参数只能是某种“物体”。你不能放另一个动作（比如 walk）进去，也不能放一个形容词（比如 red）进去。这种动词对参数的约束，叫做参数的“类型”。

在这个例子里，eat 可以接受两个“名词”（noun），所以它的两个参数，类型都是 noun。

你可能注意到了，I eat apples 里面的“I”并不是名词，而是“代词”。我解释一下。我这里所说的“名词”，是泛指一切物体以及指代物体的名字。所以我叫做“名词”的东西，也包括了代词，比如 I, you, he, she, it。如果你回想一下代词的英文是 pronoun，就会意识到它和名词（noun）之间的关系。

pronoun

/'prəʊnən/ ⓘ

noun

noun: pronoun; plural noun: pronouns

a word that can function as a noun phrase used by itself and that refers either to the participants in the discourse (e.g. *I, you*) or to someone or something mentioned elsewhere in the discourse (e.g. *she, it, this*).

你会发现这种扩展的“名词”，会大大方便我们的理解。在本书中除非特别指明，所谓“名词”包括了代词，以及一切可以被作为名词使用的结构（比如从句，动名词）。

一个句子除了动词，好像就只剩下动词的参数了。动词对它的参数具有决定性的作用，动词就是句子的核心。准确理解一个动词“想要什么参数”，什么样的结构可以出现在参数的位置，就是造出正确句子的关键。

使用不同的动词可以造出不同的句子。所以要理解语法，你应该把大部分精力放在各种各样的动词身上，而不是花几个月时间去背名词和形容词。我并不是说名词和形容词不重要，只是它们并不是核心或者骨架。

没有人会怪你不认识某种恐龙的名字，但如果你不能理解“*I am not used to eating garbage food.*”是什么意思，那你可能就有麻烦了。

具有三个参数的动词

现在举个复杂点的例子：

Coffee **makes** me happy. (咖啡使我快乐)

这里的动词是 make。跟 eat 不大一样，make 可以接受三个参数：coffee, me, happy。它的模板可以表示为：

A make B C

意思是：A 使得 B 具有性质 C。

比如 Coffee makes me happy，其中 A 是 coffee，B 是 me，C 是 happy。

再来一个例子：

I told you everything. (我告诉了你一切)

这里动词 tell 也有三个参数，它的模板是这样：

A tell B C.

意思是：A 告诉 B 一件事 C。

比如 I told you everything，其中 A 是 I，B 是 you，C 是 everything。

扯个淡：什么是宾补

说到这里我想扯个淡。初学者不知道什么是“宾补”的，可以跳过这一节，你不会损失什么。

在传统语法里，上面一节的 A make B C 和 A tell B C 被看做是不同的语法现象，前者被称为含有“宾语补足语”，后者含有“双宾语”。可是在我们的框架下，这两者都不过是“接受三个参数的动词”。你只需要熟悉 A make B C 和 A tell B C 是什么意思就可以了。

A make B C 里的 C 参数，其实就是传统语法叫做“宾语补足语”（宾补）的东西。然而跟传统语法不同，我不把它叫做“宾补”。这个成分没有任何特殊的名字和地位，而只是动词 make 的第三个参数。

有的动词可以有三个参数，有的动词只能有两个参数，有的动词只有一个参数。有的动词有时有两个参数，有时只有一个参数……就是这么简单，没有什么道理好讲，因为人们就是那么说话的。

人们约定俗成的说话方式，决定了 make 可以有三个参数，决定了这三者之间的关系：A 使得 B 变得 C。这就像数学的“定义”一样，是没有道理可讲的。你只需要多多练习，按照这个模板造句，知道它具体的意思就可以了。

模板“A make B C”，精确地决定了动词 make 可以产生的句型，定义了参数 A, B 和 C 之间的关系。你不需要把 C 叫做“宾补”就能明白这个句子在说什么。实际上，我认为“宾语补足语”，“补足语”这些术语，基本是子虚乌有的。它们来源于一种古板的观念，认为句子只有主谓宾三种成分，所以多出来一个东西，就只能叫做“补足语”了。他们没有意识到，有的动词可以有三个参数，就是这么简单。

如何造出正确的句子

我已经提到，对于人的语言能力，“造句”能力占了一半。很多人不知道复杂的长句是怎么造出来的，所以他们也很难看懂别人写的长句。

我并不是说一味追求长句是好事，正好相反。如果你能用短句表达出你的意思，就最好不要用长句。虽说如此，拥有造长句的“能力”是很重要的。这就像拥有制造核武器的能力是重要的，虽然我们可能永远不会用到核武器。

当然，长句不可能有核武器的难度。造长句其实挺容易。你先造出一个正确的短句，然后按照规则，一步步往上面添加成分，就可以逐渐“生成”一个长句。

这就像造一个房子，你首先打稳地基，用钢板造一个架子，然后往上面添砖加瓦。你可以自由地选择你想要的窗户的样式，瓦片的颜色，墙壁的材质，浴缸的形状……好像有点抽象了，我举个例子吧。

首先，我造一个最简单的句子。最简单的句子是什么呢？我们已经知道动词是句子的核心，有些动词自己就可以是一个句子。所以我们的第一个句子就是：

eat.

它适用于这样的场景：你在碗里放上狗粮，然后对狗儿说：“吃。”当然，你体会到了，这句话缺乏一些爱意，或者你只是早上起来还比较迷糊，不想多说一个字，但它至少是一个正确的句子。

接下来，我们知道 eat 可以加上两个参数，所以我就给它两个参数：I 和 apples。

I eat **apples**. (我吃苹果)

这个句子适用于这样的场景：别人问我：“你一般吃什么水果呢？”我说：“我吃苹果。”

有点单调，所以我再加点东西上去。

I eat **Fuji apples**. (我吃富士苹果)

Fuji 被我加在了 apples 前面，它给 apples 增加了一个“修饰”或者“限定”。它只能是富士苹果，而不是其它种类的苹果。

但我并不总是吃富士苹果，我有时不吃苹果。我想表达我只是“有时”吃富士苹果，所以句子又被我扩充了：

I **sometimes** eat Fuji apples. (我有时吃富士苹果)

你觉得这个 sometimes 是在修饰（限制）句子的哪个部分呢？它在修饰“我”，“苹果”，还是“吃”？实际上，它是在限制“吃”这个动作发生的频率，所以它跟 eat 的关系紧密一些，也就是说它是在修饰 eat，而不是 I 或者 apples。

以此类推，我们可以把它发展得很长：

I sometimes **eat** fresh Fuji **apples** from a nearby grocery store.

我有时候吃从附近杂货店买来的新鲜富士苹果。注意，虽然这句子挺长，但它的“骨架”仍然是 I eat apples.

我已经演示了一个长句是怎么“生成”的。先造一个短句，然后往上面添砖加瓦。正确的短句，按照规则加上一些成分，就成为正确的长句。从正确走向正确，这样你的语法就会一直是正确的。

当然，扩展句子的时候，你不能随意往上加东西，它们必须满足一定的规则才能正确的衔接。比如，你只能把 Fuji 放在 apple 前面，而不是后面，from 之类的词不可少。这就像造房子，你不能在该放窗户的地方放一道门，你不能用错配件，漏掉胶水。所谓语法，很多时候就是在告诉你这些部件要怎么样才能接的上，就跟做木工活一样。

如何理解句子

人与人交流的另一个部分就是“接收”。如果书上有很长一句话，你要怎么才能理解它呢？许多人看到长句就头痛，不知道该怎么办。这是因为他们不明白长句都是从短句扩展出来的，是有结构的。许多人理解长句失败的原因，在于他们总是从左到右，一个个的扫描单词。开头几个词感觉还认识，再多看几个词，就不知道是怎么回事了。

其实理解长句的方法，都隐含在了上一节介绍的造长句的方法里面。造句的时候我们先勾画出一个框架，然后往里面填修饰的成分。理解的时候如果有困难，我们可以用类似的办法。我们首先分析出句子的主干，把这个框架理解了，然后再把其它成分放回去，逐步把握整个句子的含义。

这个分析主干的过程，往往是“跳跃式”的，而不是“顺序式”的扫描单词。

比如之前的那个例子：

I sometimes **eat** fresh Fuji **apples** from a local grocery store.

你需要跳过修饰的成分，分析出句子的主干是短句“I eat apples”。如果你觉得一下子找不到主干，那么你可以挨个找到“修饰成分”，把它们逐个删掉，最后留下来的就是主干了。

注意，主干“I eat apples”本身就是一个语法正确的句子，它满足所有的语法规则。于是你理解了它在说“我吃苹果”。然后你返回去再看几遍，逐渐加上细节，知道是什么样的苹果，从哪里买来的，什么时候吃。

漏掉或者误解了细节，你可能会误解一部分意思，但抓住了主干，你就不会完全不理解这个句子在说什么。

再次强调，每一个复杂的长句，里面都藏着一个非常短的，语法正确的短句。理解长句的关键，就在于找到这个核心的短句。

如何获得识别修饰成分，找到主干短句的能力，也在于你对具体的语法规则的理解。

句子的树状结构

之前，我们的原始人画了这样一个图：



它表示这样一个英语句子：

I eat apples.

很多人觉得后者是更简洁，更先进的方法。然而他们没有意识到，原始人的图片里，其实包含了关键而本质的东西。被转换成一串符号之后，里面的结构看不到了，反而需要费一些脑筋才能理解。这个简单的情况也许不能说明问题，等句子复杂起来之后，你就能体会到这一点。

从现代语言学，计算机自然语言处理（NLP）的观点看来，句子并不是一串符号，而是一个“树状”的结构。我们把这种树叫做“语法树”。

比如 I eat apples，其实表示的是下图这样的结构：



你可以把这个图看成是一棵倒着长的树。你把屏幕旋转 180 度，就会看到一棵树。树干 eat 发出两个“分支”，连接着它的两个参数：I 和 apples。为了表达清晰，我用红色圆圈来表示动词，而用蓝色方形表示名词。

动词 eat 需要两个名词参数，我们给它 I 和 apples，就成了一个完整的句子。再次声明，我这里的“名词”，包括了像“I”这样的“代词”。

扩展一棵树

之前我们通过扩充 I eat apples 这句话，得到了一个逐渐变长的句子。现在有了“语法树”的概念，我们来重新演示一下这个扩充句子的过程，看看它对应的语法树是怎么变化的。

首先，我们给苹果加上“富士”（Fuji）的修饰：

I eat **Fuji** apples.

Fuji 是对 apples 的修饰，或者说是它的“属性”，所以我们在树上把它和 apples 连在一起。



对于这种“修饰”成分，我们用绿色方框来表示。它们通过灰色箭头指向它们所修饰的部分。

接着，我们加上一个时间修饰 sometimes：

I **sometimes** eat Fuji apples.

由于 sometimes 是修饰 eat 动作的频率，我们把它指向 eat 动词节点。



最后那个复杂点的句子：

I sometimes eat fresh Fuji apples from a nearby grocery store.

它的语法树大概是这个样子：

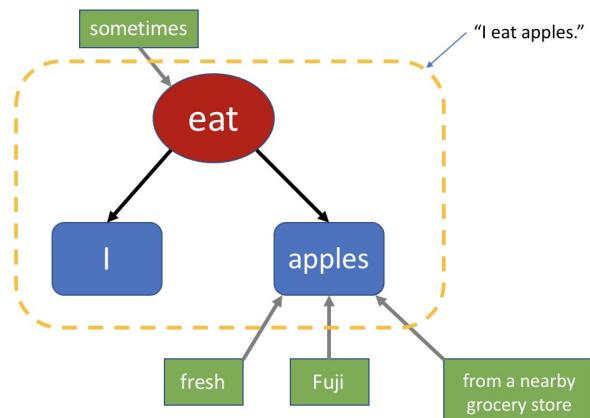


之所以说“大概”，是因为我没有把“from a nearby grocery store”完全表示成一棵树结构。当我们觉得暂时没必要深入理解一个部分的时候，我们可以把它合在一起。所以“from a nearby grocery store”一起放在了一个节点里，表示对 apples 的另一个修饰成分。

树的作用

从上面的扩展过程，你也许发现了语法树在造句时用处。它帮助你快速的“定位”需要扩展的部分。如果你的句子只是一串字符，那么你得先用眼睛找到你需要的部分，把它和旁边的文字分离开来。

在理解句子的时候，它的用处就更加明显了。树结构把句子之间相关的部分都直接连在了一起，所以你能清晰地看到它的结构。哪个词在修饰哪一部分，都一目了然。看看上面最复杂的那个句子，你可以一眼就能看出它的主干是什么：



对比一下原来短句的语法树，你发现虽然句子变长了，然而它的主干其实一点都没有变，仍然是 I eat apples。如果把句子写成一行，你就需要通过一阵子分析才能知道主干是什么。

这就是为什么我跟你讲语法树这个概念，因为它可以简化你对句子结构的理解。帮助你造句，帮助你理解复杂的句子。如果有长句看不懂，你可以使用语法树对其进行分解。

如何培养真正的语言能力

这一章我只是介绍了你需要的两种能力，可是如何培养这两种能力呢？其实它们两者是相辅相成的。造句的能力可以帮助你理解别人的句子，而阅读别人的句子，分析其结构，可以帮助你获得造出类似句子的能力。

所以我给你开的处方是这样：

1. 练习造句。每学一个动词，要先看例句，然后用它造出多个句子来。这样你就获得了灵活运用的能力。
2. 分析句子。看到一个复杂的句子，觉得理解有难度，你就把它抄下来。按照我介绍的“造句方法”，把它分解成主干和修饰成分。不久，你就会发现理解能力和造句能力都提高了。

要注意的是，分析句子的时候，没必要去纠结一个句子成分“叫什么”，对应什么术语。比如它是表语还是宾语，还是宾补……这没有意义。

你可以理解任何英语句子，你可以成为很好的记者或者作家，却仍然不知道什么叫做“宾补”。你只需要造句的能力和理解句子的能力，而你不需要术语就能做到这两点。

另外，你分析的句子来源，最好是真正的，有良好风格的英文书籍，而不是来自中国人写的语法书。比如，你可以选一本通俗易懂的英文小说，比如《哈利波特》的第一部。或者你可以用英文杂志（比如《TIME》）上的文章。很有趣的是，中国人写的语法书里面，为了演示各种语法规则，经常是“没有困难，制造困难也要上”，造出一些外国人根本不会用的，容易让人误解的句子。这种句子，就算你分析清楚了，反而是有害的。这种丑陋的句子会破坏人的语感，而且让你觉得语法无比困难，打击你的信心。你受到影响之后，就会写出类似的，让外国人看了翻白眼的丑陋句子。

最后可能有人问，你这是提高实际的英语能力，可是我需要应付标准化考试，这样学能行吗？当然行，而且你做语法题的速度会非常快。托福，雅思，GRE 之类的考试，不可能变态到要你“找出句子里的宾补成分来”。实际上，题目里根本不可能出现“宾补”这类词。他们只会在某个位置留一个空，让你选择合适的内容填进去。也就是说，你不需要知道那个成分叫“宾补”，就能做对题。

实际上，做题的时候，你的头脑里根本不应该出现“宾补”这样的术语。具有了真正的英语能力，做语法选择题的时候，你会一眼就选对正确的答案，却说不出这道题在考你哪方面的能力。是时态呢，还是某种句子成分？我不知道，因为那毫无意义。我就是感觉其它答案都不“顺口”，我根本不会写那样的句子，而正确的选项一眼看起来就是“通的”。

所以不管是实际的交流还是做题，死抠语法术语都没有什么意义。你去问问每一个英国人，美国人，他们是怎么做对语法题的，你会得到同样的答案。你应该努力得到这种母语级别的能力，而不是记住一些纸上谈兵的术语。

（如果你觉得这篇文章有启发，可以点击[这里付费](#)）

解谜计算机科学

要掌握一个学科的精髓，不能从细枝末节开始。人脑的能力很大程度上受限于信念。一个人不相信自己的时候，他就做不到本来可能的事。信心是很重要的，信心却容易被挫败。如果只见树木不见森林，人会失去信心，以为要到猴年马月才能掌握一个学科。

所以我们不从“树木”开始，而是引导读者一起来探索这背后的“森林”，把计算机科学最根本的概念用浅显的例子解释，让读者领会到它们的本质。把这些概念稍作发展，你就得到逐渐完整的把握。你一开头就掌握着整个学科，而且一直掌握着它，只不过增添更多细节而已。这就像画画，先勾勒出轮廓，一遍遍的增加细节，日臻完善，却不会失去对大局的把握。

一般计算机专业的学生学了很多课程，可是直到毕业都没能回答一个基础问题：什么是计算？这一章会引导你去发现这个问题的答案。不要小看这基础的问题，它经常是解决现实问题的重要线索。世界上有太多不理解它的人，他们走了很多的弯路，掉进很多的坑，制造出过度复杂或者有漏洞的理论和技术。

接下来，我们就来理解几个关键的概念，由此接触到计算的本质。

手指算术

每个人都做过计算，只是大部分人都没有理解自己在做什么。回想一下幼儿园（大概四岁）的时候，妈妈问你：“帮我算一下， $4+3$ 等于几？”你掰了一会手指，回答：7。当你掰手指的时候，你自己就是一台简单的计算机。

不要小看了这手指算术，它蕴含着深刻的原理。计算机科学植根于这类非常简单的过程，而不是复杂的高等数学。

现在我们来回忆一下这个过程。这里应该有一段动画，但现阶段还没有。请你对每一步发挥一下想象力，增加点“画面感”。

1. 当妈妈问你“ $4+3$ 等于几”的时候，她是一个程序员，你是一台计算机。计算机得到程序员的输入： 4 ， $+$ ， 3 。
2. 听到妈妈的问题之后，你拿出两只手，左手伸出四个指头，右手伸出三个指头。
3. 接着你开始自己的计算过程。一根一根地数那些竖起来的手指，每数一根你就把它弯下去，表示它已经被数过了。你念道：“1，2，3，4，5，6，7。”
4. 现在已经没有手指伸着，所以你把最后数到的那个数作为答案：7！整个计算过程就结束了。

符号和模型

这里的幼儿园手指算术包含着深刻的哲学问题，现在我们来初步体会一下这个问题。

当妈妈说“帮我算 $4+3$ ”的时候， 4 ， $+$ ， 3 ，三个字符传到你耳朵里，它们都是符号（symbol）。符号是“表面”的东西：光是盯着“4”和“3”这两个阿拉伯数字的曲线，一个像旗子，一个像耳朵，你是不能做什么的。你需要先用脑子把它们转换成对应的“模型”（model）。这就是为什么你伸出两只手，一只手表示 4，另一只手表示 3。

这两只手的手势是“可操作”的。比如，你把左手再多弯曲一个手指，它就变成“3”。你再伸开一根手指，它就变成“5”。所以手指是一个相当好的机械模型，它是可以动，可操作的。把符号“4”和“3”转换成手指模型之后，你就可以开始计算了。

你怎么知道“4”和“3”对应什么样的手指模型呢？因为妈妈以前教过你。十根手指，对应着 1 到 10 十个数。这就是为什么人都用十进制数做算术。

我们现在没必要深究这个问题。我只是提示你，分清“符号”和“模型”是重要的。

计算图

在计算机领域，我们经常用一些抽象的图示来表达计算的过程，这样就能直观地看到信息的流动和转换。这种图示看起来是一些形状用箭头连接起来。我在这里把它叫做“计算图”。

对于以上的手指算术 $4 + 3$ ，我们可以用下图来表示它：



图中的箭头表示信息的流动方向。说到“流动”，你可以想象一下水的流动。首先我们看到数字 4 和 3 流进了一个圆圈，圆圈里有一个“+”号。这个圆圈就是你，一个会做手指加法的小孩。妈妈给你两个数 4 和 3，你现在把它们加起来，得到 7 作为结果。

注意圆圈的输入和输出方向是由箭头决定的，我们可以根据需要调整那些箭头的位置，只要箭头的连接关系和方向不变就行。它们不一定都是从左到右，也可能从右到左或者从上到下，但“出入关系”都一样：4 和 3 进去，结果 7 出来。比如它还可以是这样：



我们用带加号的圆圈表示一个“加法器”。顾名思义，加法器可以帮我们完成加法。在上个例子里，你就是一个加法器。我们也可以用其他装置作为加法器，比如一堆石头，一个算盘，某种电子线路…… 只要它能做加法就行。

具体要怎么做加法，就像你具体如何掰手指，很多时候我们是不关心的，我们只需要知道这个东西能做加法就行。圆圈把具体的加法操作给“抽象化”了，这个蓝色的圆圈可以代表很多种东西。抽象 (abstraction) 是计算机科学至关重要的思维方法，它帮助我们进行高层面的思考，而不为细节所累。

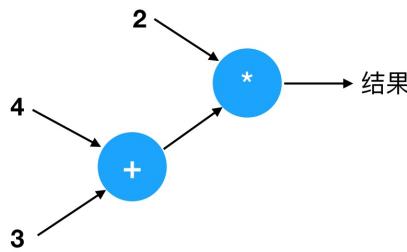
表达式

计算机科学当然不止 $4 + 3$ 这么简单，但它的基本元素确实是如此简单。我们可以创造出很复杂的系统，然而归根结底，它们只是在按某种顺序计算像 $4 + 3$ 这样的东西。

$4 + 3$ 是一个很简单的表达式 (expression)。你也许没听说过“表达式”这个词，但我们先不去定义它。我们先来看一个稍微复杂一些的表达式：

$2 * (4 + 3)$

这个表达式比 $4 + 3$ 多了一个运算，我们把它叫做“复合表达式”。这个表达式也可以用计算图来表示：



你知道它为什么是这个样子吗？它表示的意思是，先计算 $4 + 3$ ，然后把结果 (7) 传送到一个“乘法器”，跟 2 相乘，得到最后的结果。那正好就是 $2 * (4 + 3)$ 这个表达式的含义，它的结果应该是 14。

为什么要先计算 $4 + 3$ 呢？因为当我们看到乘法器 $2 * \dots$ 的时候，其中一个输入 (2) 是已知的，而另外一个输入必须通过加法器的输出得到。加法器的结果是由 4 和 3 相加得到的，所以我们必须先计算 $4 + 3$ ，然后才能与 2 相乘。

小学的时候，你也许学过：“括号内的内容要先计算”。其实括号只是“符号层”的东西，它并不存在于计算图里面。我这里讲的“计算图”，其实才是本质的东西。数学的括号一类的东西，都只是表象，它们是符号或者叫“语法”。从某种意义上讲，计算图才是表达式的本质或者“模型”，而“ $2 * (4 + 3)$ ”这串符号，只是对计算图的一种表示或者“编码” (coding)。

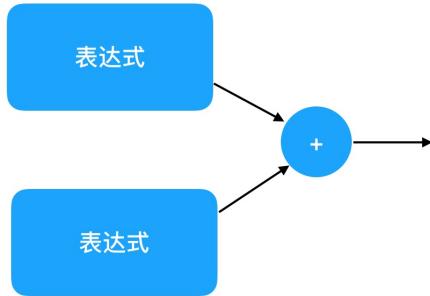
这里我们再次体会到了“符号”和“模型”的差别。符号是对模型的“表示”或者“编码”。我们必须从符号得到模型，才能进行操作。这种从符号到模型的转换过程，在计算机科学里叫做“语法分析” (parsing)。我们会在后面的章节理解这个过程。

我们现在来给表达式做一个初步的定义。这并不是完整的定义，但你应该试着理解这种定义的方式。稍后我们会逐渐补充这个定义，逐渐完善。

定义（表达式）：表达式可以是如下几种东西。

1. 数字是一个表达式。比如 1, 2, 4, 15,
2. 表达式 + 表达式。两个表达式相加，也是表达式。
3. 表达式 - 表达式。两个表达式相减，也是表达式。
4. 表达式 * 表达式。两个表达式相乘，也是表达式。
5. 表达式 / 表达式。两个表达式相除，也是表达式。

注意，由于我们之前讲过的符号和模型的差别，为了完全忠于我们的本质认识，这里的“表达式 + 表达式”虽然看起来是一串符号，它必须被想象成它所对应的模型。当你看到“表达式”的时候，你的脑子里应该浮现出它对应的计算图，而不是一串符号。这个计算图的画面大概是这个样子，其中左边的大方框里可以是任意两个表达式。



是不是感觉这个定义有点奇怪？因为在“表达式”的定义里，我们用到了“表达式”自己。这种定义叫做“递归定义”。所谓递归（recursion），就是在一个东西的定义里引用这个东西自己。看上去很奇怪，好像绕回去了一样。递归是一个重要的概念，我们会在将来深入理解它。

现在我们可以来验证一下，根据我们的定义， $2 * (4 + 3)$ 确实是一个表达式：

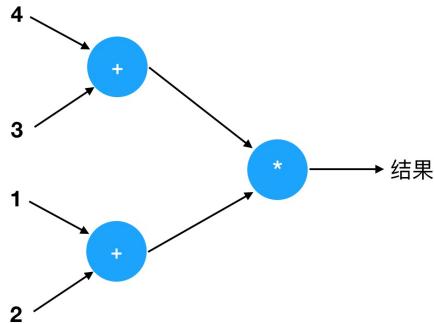
- 首先根据第一种形式，我们知道 4 是表达式，因为它是一个数字。3 也是表达式，因为它是一个数字。
- 所以 $4 + 3$ 是表达式，因为 + 的左右都是表达式，它满足表达式定义的第二种形式。
- 所以 $2 * (4 + 3)$ 是表达式，因为 * 的左右都是表达式，它满足表达式定义的第四种形式。

并行计算

考虑这样一个表达式：

$(4 + 3) * (1 + 2)$

它对应一个什么样的计算图呢？大概是这样：



如果妈妈只有你一个小孩，你应该如何用手指算出它的结果呢？你大概有两种办法。

第一种办法：先算出 $4+3$ ，结果是 7。然后算出 $1+2$ ，结果是 3。然后算 $7*3$ ，结果是 21。

第二种办法：先算出 $1+2$ ，结果是 3。然后算出 $4+3$ ，结果是 7。然后算 $7*3$ ，结果是 21。

注意到没有，你要么先算 $4+3$ ，要么先算 $1+2$ ，你不能同时算 $4+3$ 和 $1+2$ 。为什么呢？因为你只有两只手，所以算 $4+3$ 的时候你就没法算 $1+2$ ，反之也是这样。总之，你妈妈只有你一个加法器，所以一次只能做一个加法。

现在假设你还有一个妹妹，她跟你差不多年纪，她也会手指算术。妈妈现在就多了一些办法来计算这个表达式。她可以这样做：让你算 $4+3$ ，不等你算完，马上让妹妹算 $1+2$ 。等到你们的结果（7 和 3）都出来之后，让你或者妹妹算

7*3。

发现没有，在某一段时间之内，你和妹妹同时在做加法计算。这种时间上重叠的计算，叫做并行计算（parallel computing）。

你和妹妹同时计算，得到结果的速度可能会比你一个人算更快。如果你妈妈还有其它几个孩子，计算复杂的式子就可能快很多，这就是并行计算潜在的好处。所谓“潜在”的意思是，这种好处不一定会实现。比如，如果你的妹妹做手指算数的速度比你慢很多，你做完了 $4+3$ ，只好等着她慢慢的算 $1+2$ 。这也许比你自己依次算 $4+3$ 和 $1+2$ 还要慢。

即使妹妹做算术跟你一样快，这里还有个问题。你和妹妹算出结果7和3之后，得把结果传递给下一个计算 $7*3$ 的那个人（也许是她，也许是你的妹妹）。这种“通信”会带来时间的延迟，叫做“通信开销”。如果你们其中一个说话慢，这比起一个人来做计算可能还要慢。

如何根据计算单元能力的不同和通信开销的差异，来最大化计算的效率，降低需要的时间，就成为了并行计算领域研究的内容。并行计算虽然看起来是一个“博大精深”的领域，可是你如果理解了我这里说的那点东西，就很容易理解其余的内容。

变量和赋值

如果你有一个复杂的表达式，比如

```
(5 - 3) * (4 + (2 * 3 - 5) * 6)
```

由于它有比较多的嵌套，人的眼睛是难以看清楚的，它要表达的意义也会难懂。这时候，你希望可以用一些“名字”来代表中间结果，这样表达式就更容易理解。

打个比方，这就像你有一个亲戚，他是你妈妈的表姐的女儿的丈夫。你不想每次都称他“我妈妈的表姐的女儿的丈夫”，所以你就用他的名字“叮当”来指代他，一下子就简单了。

我们来看一个例子。之前的复合表达式

```
2 * (4 + 3)
```

其实可以被转换为等价的，含有变量的代码：

```
{
    a = 4 + 3      // 变量 a 得到 4+3 的值
    2 * a          // 代码块的值
}
```

其中a是一个名字。 $a = 4 + 3$ 是一个“赋值语句”，它的意思是：用a来代表 $4 + 3$ 的值。这种名字，计算机术语叫做变量（variable）。

这段代码的意思可以简单地描述为：计算 $4 + 3$ ，把它的结果表示为a，然后计算 $2 * a$ 作为最后的结果。

有些东西可能扰乱了你的视线。两根斜杠//后面一直到行末的文字叫做“注释”，是给人看的说明文字。它们对代码的逻辑不产生作用，执行的时候可以忽略。许多语言都有类似这种注释，它们可以帮助阅读的人，但是会被机器忽略。

这段代码执行过程会是这样：先计算 $4 + 3$ 得到7，用a记住这个中间结果7。接着计算 $2 * a$ ，也就是计算 $2 * 7$ ，所以最后结果是14。很显然，这跟 $2 * (4 + 3)$ 的结果是一样的。

a叫做一个变量，它是一个符号，可以用来代表任意的值。除了a，你还有许多的选择，比如b, c, d, x, y, foo, bar, u21...只要它不会被误解成其它东西就行。

如果你觉得这里面的“神奇”成分太多，那我们现在来做更深一层的理解.....

再看一遍上面的代码。这整片代码叫做一个“代码块”（block），或者叫一个“序列”（sequence）。这个代码块包括两条语句，分别是 $a = 4 + 3$ 和 $2 * a$ 。代码块里的语句会从上到下依次执行。所以我们先执行 $a = 4 + 3$ ，然后执行 $2 * a$ 。

最后一条语句 $2 * a$ 比较特别，它是这个代码块的“值”，也就是最后结果。之前的语句都是在为生成这个最后的值做准备。换句话说，这整个代码块的值就是 $2 * a$ 的值。不光这个例子是这样，这是一个通用的原理：代码块的最后一条语句，总是这个代码块的值。

我们在代码块的前后加上花括号{}进行标注，这样里面的语句就不会跟外面的代码混在一起。这两个花括号叫做“边界符”。我们今后会经常遇到代码块，它存在于几乎所有的程序语言里，只是语法稍有不同。比如有些语言可能用括号(...)或者BEGIN...END来表示边界，而不是用花括号。

这片代码已经有点像常用的编程语言了，但我们暂时不把它具体化到某一种语言。我不想固化你的思维方式。在稍后的章节，我们会把这种抽象的表达法对应到几种常见的语言，这样一来你就能理解几乎所有的程序语言。

另外还有一点需要注意，同一个变量可以被多次赋值。它的值会随着赋值语句而改变。举个例子：

```
{  
    a = 4 + 3  
    b = a  
    a = 2 * 5  
    c = a  
}
```

这段代码执行之后，`b` 的值是 7，而 `c` 的值是 10。你知道为什么吗？因为 `a = 4 + 3` 之后，`a` 的值是 7。`b = a` 使得 `b` 得到值 7。然后 `a = 2 * 5` 把 `a` 的值改变了，它现在是 10。所以 `c = a` 使得 `c` 得到 10。

对同一个变量多次赋值虽然是可以的，但通常来说这不是一种好的写法，它可能引起程序的混淆，应该尽量避免。只有当变量表示的“意义”相同的时候，你才应该对它重复赋值。

编译

一旦引入了变量，我们就可以不用复合表达式。因为你可以把任意复杂的复合表达式拆开成“单操作算术表达式”（像 `4 + 3` 这样的），使用一些变量记住中间结果，一步一步算下去，得到最后的结果。

举一个复杂点的例子，也就是这一节最开头的那个表达式：

```
(5 - 3) * (4 + (2 * 3 - 5) * 6)
```

它可以被转化为一串语句：

```
{  
    a = 2 * 3  
    b = a - 5  
    c = b * 6  
    d = 4 + c  
    e = 5 - 3  
    e * d  
}
```

最后的表达式 `e * d`，算出来就是原来的表达式的值。你观察一下，是不是每个操作都非常简单，不包含嵌套的复合表达式？你可以自己验算一下，它确实算出跟原表达式一样的结果。

在这里，我们自己动手做了“编译器”（compiler）的工作。通常来说，编译器是一种程序，它的任务是把一片代码“翻译”成另外一种等价形式。这里我们没有写编译器，可是我们自己做了编译器的工作。我们手动地把一个嵌套的复合表达式，编译成了一系列的简单算术语句。

这些语句的结果与原来的表达式完全一致。这种保留原来语义的翻译过程，叫做编译（compile）。

我们为什么需要编译呢？原因有好几种。我不想在这里做完整的解释，但从这个例子我们可以看到，编译之后我们就不再需要复杂的嵌套表达式了。我们只需要设计很简单的，只会做单操作算术的机器，就可以算出复杂的嵌套的表达式。实际上最后这段代码已经非常接近现代处理器（CPU）的汇编代码（assembly）。我们只需要多加一些转换，它就可以变成机器指令。

我们暂时不写编译器，因为你还缺少一些必要的知识。这当然也不是编译技术的所有内容，它还包含另外一些东西。但从这一开头，你就已经初步理解了编译器是什么，你只需要在将来加深这种理解。

函数

到目前为止，我们做的计算都是在已知的数字之上，而在现实的计算中我们往往有一些未知数。比如我们想要表达一个“风扇控制器”，有了它之后，风扇的转速总是当前气温的两倍。这个“当前气温”就是一个未知数。

我们的“风扇控制器”必须要有一个“输入”（input），用于得到当前的温度 `t`，它是一个温度传感器的读数。它还要有一个输出，就是温度的两倍。

那么我们可以用这样的方式来表达我们的风扇控制器：

```
t -> t*2
```

不要把这想成任何一种程序语言，这只是我们自己的表达法。箭头 `->` 的左边表示输入，右边表示输出，够简单吧。

你可以把 `t` 想象成从温度传感器出来的一根电线，它连接到风扇控制器上，风扇控制器会把它的输入（`t`）乘以 2。这个画面像这个样子：



我们谈论风扇控制器的时候，其实不关心它的输入是哪里来的，输出到哪里去。如果我们把温度传感器和风扇从画面里拿掉，就变成这个样子：



这幅图才是你需要认真理解的函数的计算图。你发现了吗，这幅图画正好对应了之前的风扇控制器的符号表示： $t \rightarrow t^2$ 。看到符号就想象出画面，你就得到了符号背后的模型。

像 $t \rightarrow t^2$ 这样具有未知数作为输入的构造，我们把它叫做函数（function）。其中 t 这个符号，叫做这个函数的参数。

参数，变量和电线

你可能发现了，函数的参数和我们之前了解的“变量”是很类似的，它们都是一个符号。之前我们用了 a, b, c, d, e 现在我们有一个 t ，这些名字我们都是随便起的，只要它们不要重复就好。如果名字重复的话，可能会带来混淆和干扰。

其实参数和变量这两种概念不只是相似，它们的本质就是一样的。如果你深刻理解它们的相同本质，你的脑子就可以少记忆很多东西，而且它可能帮助你对代码做出一些有趣而有益的转化。在上一节你已经看到，我用“电线”作为比方来帮助你理解参数。你也可以用同样的方法来理解变量。

比如我们之前的变量 a ：

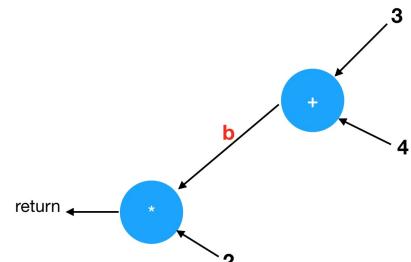
```
{
    a = 4 + 3
    2 * a
}
```

它可以被想象成什么样的画面呢？



我故意把箭头方向画成从右往左，这样它就更像上面的代码。从这个图画里，你也许可以看到变量 a 和风扇控制器图里的参数 t ，其实没有任何本质差别。它们都表示一根电线，那根电线进入乘法器，将会被乘以 2，然后输出。如果你把这些都看成是电路，那么变量 a 和参数 t 都代表一根电线而已。

然后你还发现一个现象，那就是你可以把 a 这个名字换成任何其它名字（比如 b ），而这幅图不会产生实质的改变。



这说明什么问题呢？这说明以下的代码（把 `a` 换成了 `b`）跟之前的是等价的：

```
{  
    b = 4 + 3  
    2 * b  
}
```

根据几乎一样的电线命名变化，你也可以对之前的函数得到一样的结论：`t -> t*2` 和 `u -> u*2`，和 `x -> x*2` 都是一回事。

名字是很重要的东西，但它们具体叫什么，对于机器并没有实质的意义，只要它们不要相互混淆就可以。但名字对于人是很重要的，因为人脑没有机器那么精确。不好的变量和参数名会导致代码难以理解，引起程序员的混乱和错误。所以通常说来，你需要给变量和参数起好的名字。

什么样的名字好呢？我会在后面集中讲解。

有名字的函数

既然变量可以代表“值”，那么一个自然的想法，就是让变量代表函数。所以就像我们可以写

```
a = 4 + 3
```

我们似乎也应该可以写

```
f = t -> t*2
```

对的，你可以这么做。`f = t->t*2` 还有一个更加传统的写法，就像数学里的函数写法：

```
f(t) = t*2
```

请仔细观察 `t` 的位置变化。我们在函数名字的右边写一对括号，在里面放上参数的名字。

注意，你不可以只写

```
f = t*2
```

你必须明确指出函数的参数是什么，否则你就不会明白函数定义里的 `t` 是什么东西。明确指出 `t` 是一个“输入”，你才会知道它是函数的输入，是一个未知数，而不是在函数外面定义的其它变量。

这个看似简单的道理，很多数学家都不明白，所以他们经常这样写书：

有一个函数 $y = x^2$

这是错误的，因为他没有明确指出“`x` 是函数 `y` 的参数”。如果这句话之前他们又定义过 `x`，你就会疑惑这是不是之前那个 `x`。很多人就是因为这些糊里糊涂的写法而看不懂数学书。这不怪他们，只怪数学家自己对于语言不严谨。

函数调用

有了函数，我们可以给它起名字，可是我们怎么使用它的值呢？

由于函数里面有未知数（参数），所以你必须告诉它这些未知数，它里面的代码才会执行，给你结果。比如之前的风扇控制器函数

```
f(t) = t*2
```

它需要一个温度作为输入，才会给你一个输出。于是你就这样给它一个输入：

```
f(2)
```

你把输入写在函数名字后面的括号里。那么你就会得到输出：4。也就是说 `f(2)` 的值是 4。

如果你没有调用一个函数，函数体是不会被执行的。因为它不知道未知数是什么，所以什么事也做不了。那么我们定义函数的时候，比如

```
f(t) = t*2
```

当看到这个定义的时候，机器应该做什么呢？它只是记录下：有这么一个函数，它的参数是 `t`，它需要计算 `t*2`，它的名字叫 `f`。但是机器不会立即计算 `t*2`，因为它不知道 `t` 是多少。

分支

直到现在，我们的代码都是从头到尾，闷头闷脑地执行，不问任何问题。我们缺少一种“问问题”的方法。比如，如果我想表达这样一个“食物选择器”：如果气温低于 22 度，就返回 “hotpot” 表示今天吃火锅，否则返回 “ice cream” 表示今天吃冰激凌。

我们可以把它图示如下：



中间这种判断结构叫做“分支”（branching），它一般用菱形表示。为什么叫分支呢？你想象一下，代码就像一条小溪，平时它沿着一条路线流淌。当它遇到一个棱角分明的大石头，就分成两个支流，分开流淌。

我们的判断条件 $t < 22$ 就像一块大石头，我们的“代码流”碰到它就会分开成两支，分别做不同的事情。跟溪流不同的是，这种分支不是随机的，而是根据条件来决定，而且分支之后只有一支继续执行，而另外一边不会被执行。

我们现在看到的都是图形化表示的模型，为了书写方便，现在我们要从符号的层面来表示这个模型。我们需要一种符号表示法来表达分支，我们把它叫做 if（如果）。我们的饮料选择器代码可以这样写：

```
t -> if (t < 22)
{
    "hotpot"
}
else
{
    "ice cream"
}
```

它是一个函数，输入是一个温度。if 后面的括号里放我们的判断条件。后面接着条件成立时执行的代码块，然后是一个 else，然后是条件不成立时执行的代码。它说：如果温度低于 22 度，我们就吃火锅，否则就吃冰激凌。

其中的 else 是一个特殊的符号，它表示“否则”。看起来不知道为什么 else 要在那里？对的，它只是一个装饰品。我们已经有足够的表达力来分辨两个分支，不过有了 else 似乎更加好看一些。很多语言里面都有 else 这个标记词在那里，所以我也把它放在那里。

这只是一个最简单的例子，其实那两个代码块里面不止可以写一条语句。你可以有任意多的语句，就像这样：

```
t ->
if (t < 22)
{
    a = 4 + 3
    b = a * 2
    "hotpot"
}
else
{
    x = "ice cream"
    x
}
```

这段代码和之前是等价的，你知道为什么吗？

字符串

上面一节出现了一种我们之前没见过的东西，我为了简洁而没有介绍它。这两个分支的结果，也就是加上引号的 “hotpot” 和 “ice cream”，它们并不是数字，也不是其它语言构造，而是一种跟数字处于几乎同等地位的“数据类型”，叫做字符串（string）。字符串是我们在计算机里面表示人类语言的基本数据类型。

关于字符串，在这里我不想讲述更加细节的内容，我把对它的各种操作留到以后再讲，因为虽然字符串对于应用程序很重要，它却并不是计算机科学最关键最本质的内容。

很多计算机书籍一开头就讲很多对字符串的操作，导致初学者费很大功夫去做很多打印字符串的练习，结果几个星期之后还没学到“函数”之类最根本的概念。这是非常可惜的。

布尔值

我们之前的 `if` 语句的条件 `t < 22` 其实也是一个表达式，它叫做“布尔表达式”。你可以把小于号 `<` 看成是跟加法一类的“操作符”。它的输入是两个数值，输出是一个“布尔值”。什么是布尔值呢？布尔值只有两个：`true` 和 `false`，也就是“真”和“假”。

举个例子，如果 `t` 的值是 15，那么 `t < 22` 是成立的，那么它的值就是 `true`。如果 `t` 的值是 23，那么 `t < 22` 就不成立，那么它的值就是 `false`。是不是很好理解呢？

我们为什么需要“布尔值”这种东西呢？因为它的存在可以简化我们的思维。对于布尔值也有一些操作，这个我也不在这一章赘述，放到以后细讲。

计算的要素

好了，现在你已经掌握了计算机科学的几乎所有基本要素。每一个编程语言都包括这些构造：

1. 基础的数值。比如整数，字符串，布尔值等。
2. 表达式。包括基本的算术表达式，嵌套的表达式。
3. 变量和赋值语句。
4. 分支语句。
5. 函数和函数调用。

你也许可以感觉到，我是把这些构造按照“从小到大”的顺序排列的。这也许可以帮助你的理解。

现在你可以回想一下你对它们的印象。每当学习一种新的语言或者系统，你只需要在里面找到对应的构造，而不需要从头学习。这就是掌握所有程序语言的秘诀。这就像学开车一样，一旦你掌握了油门，刹车，换挡器，方向盘，速度表的功能和用法，你就学会了开所有的汽车，不管它是什么型号的汽车。

我们在这一章不仅理解了这些要素，而且为它们定义了一种我们自己的“语言”。显然这个语言只能在我们的头脑里运行，因为我们没有实现这个语言的系统。在后面的章节，我会逐渐的把我们这种语言映射到现有的多种语言里面，然后你就能掌握这些语言了。

但是请不要以为掌握了语言就学会了编程或者学会了计算机科学。掌握语言就像学会了各种汽车部件的工作原理。几分钟之内，初学者就能让车子移动，转弯，停止。可是完了之后你还需要学习交通规则，你需要许许多多的实战练习和经验，掌握各种复杂情况下的策略，才能成为一个合格的驾驶员。如果你想成为赛车手，那就还需要很多倍的努力。

但是请不要被我这些话吓到了，你没有那么多的竞争者。现在的情况是，世界上就没有很多合格的计算机科学驾驶员，更不要说把车开得流畅的赛车手。绝大部分的“程序员”连最基本的引擎，油门，刹车，方向盘的工作原理都不明白，思维方式就不对，所以根本没法独自上路，一上路就出车祸。很多人把过错归结在自己的车身上，以为换一辆车马上就能成为好的驾驶员。这是一种世界范围的计算机教育的失败。

在后面的章节，我会引导你成为一个合格的驾驶员，随便拿一辆车就能开好。

什么是计算

现在你掌握了计算所需要的基本元素，可是什么是计算呢？我好像仍然没有告诉你。这是一个很哲学的问题，不同的人可能会告诉你不同的结果。我试图从最广义的角度来告诉你这个问题的答案。

当你小时候用手指算 $4+3$ ，那是计算。如果后来你学会了打算盘，你用算盘算 $4+3$ ，那也是计算。后来你从我这里学到了表达式，变量，函数，调用，分支语句…… 在每一新的构造加入的过程中，你都在了解不同的计算。

所以从最广义来讲，计算就是“机械化的信息处理”。所谓机械化，你可以用手指算，可以用算盘，可以用计算器，或者计算机。这些机器里面可以有代码，也可以没有代码，全是电子线路，甚至可以是生物活动或者化学反应。不同的机器也可以有不同的计算功能，不同的速度和性能……

有这么多种计算的事实不免让人困惑，总害怕少了点什么，其实你可以安心。如果你掌握了上一节的“计算要素”，那么你就掌握了几乎所有类型的计算系统所需要的东西。你在后面所要做的只是加深这种理解，并且把它“对应”到现实世界遇到的各种计算机器里面。

为什么你可以相信计算机科学的精华就只有这些呢？因为计算就是处理信息，信息有它诞生的位置（输入设备，固定数值），它传输的方式（赋值，函数调用，返回值），它被查看的地方（分支）。你想不出对于信息还有什么其它的操作，所以你就很安心的相信了，这就是计算机科学这种“棋类游戏”的全部规则。

(如果你觉得这篇文章有启发，可以点击[这里付费](#))

智能合约和形式验证

在之前一篇[关于人工智能的文章](#)里，我指出了“自动编程”的不可能性。今天我想来谈谈一个相关的话题：以太坊式的智能合约的形式验证。有些人声称要实现基于“深度学习”的，自动的智能合约形式验证（formal verification），用以确保合约的正确性。然而今天我要告诉你的是，跟自动编程一样，完全自动的合约验证也是不可能实现的。

随着区块链技术的愈演愈烈，很多人开始在以太坊（Ethereum）的“智能合约语言”上做文章。其中一部分是搞 PL 的人，他们试图对 Solidity 之类语言写的智能合约进行形式验证，号称要用严密的数理逻辑方法，自动的验证智能合约的正确性。其中一种方法是用“深度学习”，经过训练之后，自动生成 Hoare Logic 的“前条件”和“后条件”。

Hoare Logic

我好像已经把你搞糊涂了…… 我们先来科普一下 Hoare Logic。

[Hoare Logic](#) 是一种形式验证的方法，用于验证程序的正确性。它的做法是，先给代码标注一些“前条件”和“后条件”（pre-condition 和 post-condition），然后就可以进行逻辑推理，验证代码的某些基本属性，比如转账之后余额是正确的。

举一个很简单的 Hoare Logic 例子：

{ $x=0$ } $x:=x+1$ { $x>0$ }

它的意思是，如果开头 x 等于 0，那么 $x:=x+1$ 执行之后， x 应该大于 0。这里的前条件（pre-condition）是 $x=0$ ，后条件（post-condition）是 $x > 0$ 。如果 x 开头是零，执行 $x:=x+1$ 之后， x 就会大于 0，所以这句代码就验证通过了。

Hoare Logic 的系统把所有这些前后条件和代码串接起来，经过逻辑推导验证，就可以作出这样的保证：在前条件满足的情况下，执行代码之后，后条件一定是成立的。如果所有这些条件都满足，系统就认为这是“正确的程序”。注意这里的所谓“正确”，完全是由人来决定的，系统并不知道“正确”是什么意思。

Hoare Logic 对于程序的安全性，确实可以起到一定的效果，它已经被应用到了一些实际的项目。比如微软 Windows 的驱动程序代码里面，有一种“安全标注语言”，叫做 SAL，其实就是 Hoare Logic 的一个实现。然而前条件和后条件是什么，你必须自己给代码加上标注，否则系统就不能工作。

比如上面的例子，系统如何知道我想要“ $x>0$ ”这个性质呢？只有我自己把它写出来。所以要使用 Hoare Logic，必须在代码上标注很多 pre-condition 和 post-condition。这些条件要如何写，必须要深入理解程序语言和形式逻辑的原理。这个工作需要经过严格训练的专家来完成，而且需要很多的时间。

自动生成标注是不可能的

所以即使有了 Hoare Logic，程序验证也不是轻松的事情。于是呢就有人乘火打劫，提出一个类似减肥药的想法，声称他们要用“深度学习”，通过对已有标注的代码进行学习，最后让机器自动标注这些前后条件。还在“空想”阶段呢，却已经把“自动标注”作为自己的“优势”写进了白皮书：“我们的方法是自动的，其他的项目都是手动的……”

很可惜的是，“自动标注”其实跟“自动编程”是一样的空想。自动编程的难点在于机器没法知道你想要做什么。同理，自动标注的难点在于机器没法知道你想要代码满足什么样的性质（property）。这些信息只存在于你的心里，如果你不表达出来，任何其它人和机器都没有办法知道。

除非你把它写出来，机器永远无法知道函数的参数应该满足什么样的条件（前条件），它也无法知道函数的返回值应该满足什么样的条件（后条件）。比如上面的那个例子，机器怎么知道你想要程序执行之后 x 大于零呢？除非你告诉它，它是不可能知道的。

你也许会问，深度学习难道帮不上忙吗？想想吧…… 你可以给深度学习系统上千万行已经标注好的代码。你可以把整个 Windows 系统，整个 Linux 系统，FireFox 的代码全都标注好，再加上一些战斗机，宇宙飞船的代码，输入深度学习系统进行“学习”。现在请问系统，我下面要写一个新的函数，你知道我想要做什么吗？你知道我希望它满足什么性质吗？它仍然不知道啊！只有我自己才知道：它是用来给我的猫铲屎的 :p

所以，利用深度学习自动标注 Hoare Logic 的前后条件，跟“自动编程”一样，是在试图实现“读心术”，那显然是不可能的。作为资深的 PL 和形式验证专家，这些人应该知道这是不可能自动实现的。他们提出这样的想法，并且把它作为相对于其他智能合约项目的优势，当然只是为了忽悠外行，为了发币圈钱；)

如果真能用深度学习生成前后条件，从而完全自动的验证程序的正确性，那么这种办法应该早就在形式验证领域炸锅了。每一个形式验证专家都希望能够完全自动的证明程序的正确性，然而他们早就知道那是不可能的。

设计语言来告诉机器我们想要什么，什么叫做“正确”，这本身就是 PL 专家和形式验证专家的工作。设计出了语言，我们还得依靠优秀的程序员来写这些代码，告诉机器我们想要做什么。我们得依靠优秀安全专家，给代码加上前后

条件标注，告诉机器什么叫做“正确安全的代码”…… 这一切都必须是人工完成的，无法靠机器自动完成。

当然，我并没有排除对智能合约手动加上 Hoare Logic 标记这种做法的可行性，它是有一定价值的。我只是想提醒大家，这些标记必须是人工来写的，不可能自动产生。另外，虽然工具可以有一定的辅助作用，但如果写代码的人自己不小心，是无法保证程序完全正确的。

如何保证智能合约的正确呢？这跟保证程序的正确性是一样的问题。只有懂得如何写出干净简单的代码，进行严密的思考，才能写出正确的智能合约。关于如何写出干净，简单，严密可靠的代码，你可以参考我之前的一些文章。

做智能合约验证的工作也许能圈到钱，然而却是非常枯燥而没有成就感的。为此我拒绝了好几个有关区块链的合作项目。虽然我对区块链的其它一些想法（比如去中心化的共识机制）是感兴趣的，我对智能合约的正确性验证一点都不看好。

智能合约是一个误区

实际上，我认为智能合约这整个概念就不靠谱，是一个比较大的误区。比特币和以太坊的系统里面，根本就不应该，而且没必要存在脚本语言。

比特币的解锁脚本执行方式，一开头就有个低级错误，导致 injection 安全漏洞。用户可以写出恶意代码，导致脚本的运行系统出错。比特币最初的解锁方式，是把两段代码（解锁脚本+加锁脚本）以文本方式拼接在一起，然后执行。以文本方式处理程序，是程序语言实现方法的大忌。稍微有点经验的黑客都知道，这里很可能有安全漏洞。

以太坊的 Solidity 语言一开头就有低级错误，导致价值五千万美元的以太币被盗。以太坊的智能合约系统消耗大量的计算资源，还导致了严重的性能问题。

虽然比特币和以太坊的作者大概在密码学和分布式系统领域都是高手，然而我不得不坦言，他们都是 PL 外行。然而如果是内行来做这些语言，难道就会更好吗？我并不这么认为。

首先的问题，是 PL 这个领域充满了各种宗教，和许多的传教士。一般的 PL 内行都会把问题复杂化，他们会试图设计一个自己的“信仰”中完美的语言，而不顾其他人的死活。如果你运气不好，就会遇到那种满嘴“纯函数”，monad，dependent type，linear logic 的极客…… 然后设计出来的语言就没人会用了。

有责任感的 PL 科学家，都是首先试图避免制造新的语言。能不用新语言解决问题，就不要设计新的语言，而且尽量不在系统里采用嵌入式语言。所以，如果换做是我设计了比特币，我根本不会为它设计一种语言。

让用户可以编程是很危险的。极少有用户能够写出正确而可靠的代码，而且语言系统的开发过程中极少可以不出现 bug。语言系统的设计错误会给黑客可乘之机，写出恶意脚本来进行破坏。从来没有任何语言和他们的编译器，运行时系统是一开头就正确的，都需要很多年才能稳定下来。

另外你还要考虑性能问题。对于去中心的分布式系统，这种问题就更加棘手。这对于普通的语言问题不大，你不要用它来控制飞机就可以。然而数字货币系统的语言，几乎不允许出现这方面的问题。

所以与其提心吊胆的设计这些智能合约语言，还不如干脆不要这种功能。

我们真的需要那些脚本的功能吗？比特币虽然有脚本语言，可是常用的脚本其实只有不超过 5 个，直接 hard code 进去就可以了。以太坊的白皮书虽然做了那么多的应用展望，EVM 上出现过什么有价值的应用吗？我并不觉得我们需要这些智能合约。数字货币只要做好一件事，能被安全高效的当成钱用，就已经不错了。

美元，人民币，黄金…… 它们有合约的功能吗？没有。为什么数字货币一定要捆绑这种功能呢？我觉得这违反了模块化设计的原则：一个事物只做一点事，把它做到最好。数字货币就应该像货币一样，能够实现转账交换的简单功能就可以了。合约应该是另外独立的系统，不应该跟货币捆绑在一起。

那合约怎么办呢？交给律师和会计去办，或者使用另外独立的系统。你有没有想过，为什么世界上的法律系统不是程序控制自动执行的呢？为什么我们需要律师和法官，而不只是机器人？为什么有些国家的法庭还需要有陪审团，而不光是按照法律条款判案？这不只是历史遗留问题。你需要理解法律的本质属性才会明白，完全不通过人来进行的机械化执法是不可行的。智能合约就是要把人完全从这个系统里剔除出去，那是会出问题的。

奢望过多的功能其实是一种过度工程（over-engineering）。花费精力去折腾智能合约系统，可能会大大的延缓数字货币真正被世界接受。实话说嘛，试用了多种数字货币，了解了它们所用的技术之后，我发现这些技术相当的有趣，然而这些数字货币仍然处于试验阶段，离真正作为货币使用还有一定距离。集中精力改进它们作为货币的功能，将会加速它们被接受，而耗费精力去研究智能合约，我觉得是误入歧途。

在这一点上，我觉得比特币比它的后继者们（比如以太坊）都要做的地道一些。比特币虽然也有脚本语言，然而它并不过分强调这种脚本的作用。比特币的脚本语言非常简单，而且不是图灵完备的。这迫使用户只能写出功能简单，不伤害系统性能，容易验证的脚本。相比之下，以太坊花了太多精力去折腾智能合约，弄得过度复杂，搞成了图灵完备的，最终带来各种问题，影响了大家接受最重要的货币功能。

成都的雾霾

今天是大年初一，本来不想写东西的。可是一大早起来，就看到这样的天空，空气质量又是“重度污染”。连大年初一都不放过我，出不了门，所以我这个年也不想过了，决定写点什么，在我被这雾霾灭口之前 :)



三个月以来，成都的天空一直是这个样子，每一天都像是世界末日。当你还能看见太阳的时候就更诡异了，大白天的看到太阳，是一个朦胧的火球，空气都是红色橙色的，红得可怕。这只有在好莱坞的外星人入侵地球大片里才能看到。

别搞错了，这不是普通的四川盆地的雾气，而是严重污染的雾霾。成都的这三个月，几乎每一天都有严重的雾霾。查看空气质量历史数据，成都的冬天就没几天空气是优良的，这比北京上海都要差很多。北京和上海每个月还是会有一天空气差的时候，然而大部分时候的空气，还是基本可以接受的。而成都呢？三个月来空气好的时候，掰着指头都数的出来。



很多人以为成都只有冬天有空气污染，还不以为然的样子。冬天有三个月之久，每天这样的雾霾，已经是非常难受的。而且成都也许不只是冬天才有污染。你想想，北方冬天雾霾严重，是因为北方需要烧煤产生暖气。成都冬天根本没有集体供暖，不需要烧煤，那么这雾霾是怎么来的？

夏天的时候好点，然而那是因为成都夏天下很多的雨，雨水会部分的洗掉空气中的颗粒物。成都夏天的雨多得让人讨厌，每天都得带伞。如果下这么多的雨，在其它城市空气早就变优了，然而成都夏天的空气，也只能算勉强能活下去而已。



所以这污染应该是不分季节常年进行的，并不是冬天才有。

不要误解了，空气污染绝对不是自然现象，也不是汽车多了造成的。这绝对是人祸。问自己几个问题，稍微上网搜索一下，你就能明白这些雾霾是怎么来的。

1. 纽约，东京那么大的城市，那么多汽车在跑来跑去，或者堵在那里。为什么纽约和东京每天空气质量都超级好？pm2.5 基本都是 20 以下。所以看来雾霾并不是汽车尾气造成的。



另外我还观察到一个现象，那就是成都的污染会在晚上加重。也许白天是轻度污染，到了晚上就是重度污染，然后天亮了又开始好一点。如果污染是汽车产生的，为何会在晚上三更半夜的加重？要知道半夜的时候路上几乎没有车的！所以这又说明污染不是汽车产生的。

汽车尾气里面基本都是气体，是没有固体“颗粒物”的。除了原来的空气成分，大部分尾气都是二氧化碳（CO₂）和水蒸气。二氧化碳不是污染物。汽车尾气的污染物主要是二氧化硫（SO₂），一氧化碳（CO），二氧化氮（NO₂）之类的气体，它们是不会导致 pm2.5 的值升高的。

看看成都空气质量的详细数据你就会发现，NO₂，CO，SO₂，O₃ 四项指标，几乎一直是优的。如果汽车造成污染，那么这几种气体应该会超标吧？结果没有。说明污染不是汽车造成的。

如果你还不信，就再来想想汽车引擎的工作原理。汽车的动力来源于汽油在气缸内点火爆炸，推动活塞运动。如果汽油爆炸产生颗粒物的话，那是会损坏气缸和活塞的。所以质量好点的汽油应该是没有颗粒物杂质的，它一挥发就干干净净的，它燃烧也不会产生颗粒物排放。中国的汽油应该是达到了这个标准的。

古老的手扶拖拉机一类的车子，烧劣质的燃料，经常冒黑烟，还是可能有一些颗粒物排放。然而你有多少次在成都这样的城市看到手扶拖拉机或者冒黑烟的汽车？我小的时候住在一个县城，好多冒黑烟的手扶拖拉机跑来跑去的，可是为何没有产生雾霾？;)

所以颗粒物污染，基本只能是烧煤，或者烧别的什么固体燃料（垃圾？）

2. 查看空气质量历史，为什么成都到了晚上，空气质量就变得很差，而白天就稍微好点？请你想想，成都的晚上和白天有什么不同？成都不像北京，它没有烧煤的集体供暖设施，所以晚上并不会比白天需要烧更多的煤。为什么污染会在晚上更严重？

你只需要上网搜一下，就基本有了线索：原来那些颗粒物是附近的工厂排出来的。工厂为了掩盖排烟的真相，都选择在晚上偷排，这样就不那么明显了。近一点的人可以看见，远一点的，混在夜幕中就看不见了。

为什么成都多雾霾天气？空气质量那么差？

kenji chueng + 关注

坐几次晚上12点以后到成都的飞机看看窗外
你就知道了 成都周边全是工厂在偷排浓烟完全
看不到地面 成都市里面都要相对清楚一些
另外我觉得就是成都车辆太多

发布于 2016-12-04
著作权归作者所有

▲ 赞同 14 ▼ 感谢作者 收藏 评论 1

为什么成都多雾霾天气？空气质量那么差... 38个回答 > ...

jin + 关注

我觉得最重要的是 成都的各个工地，几百万的汽车排放，各个工厂（彭州石化）的晚上偷排，以前在双流读书时，半夜睡不着出来在窗边抽烟时就看见不远处的烟窗在冒着黑黑的大烟，白天却重来没看见排放过，是白天不敢偷排，成都并非一线城市，是因为他还没有发展起来的，但现在的雾霾与几年前相比却严重了许多，可以肯定的是再等十几年，等成都有十几条地铁线，可以与一线城市媲美的时候，雾霾更甚，那时它就再也不是外人看来特别适合休闲生活的地方了，而把这个雾霾原因归结于熏腊肉，烧秸秆的，那是完全是扯淡，烟熏是会造成一些影响，但也仅仅是可忽略的，为什么，熏腊肉，烧秸秆从古至今一直都有，古时那时怎么没有雾霾。

▲ 赞同 5 ▼ 感谢作者 收藏 评论

3. 再研究一下雾霾数据地图，就会发现他们说的那些位置比较准确。污染最严重的地区，是成都的郊区：双流，郫县，崇州，彭州。如果你按时间观察，会发现那些地方晚上就开始污染加重。



再来印证一下。退一万步说，如果汽车能造成颗粒物污染，那种乡下郊区晚上会有那么多汽车在跑，造成那么严重的污染吗？显然不可能有那么多车在跑。那么是什么造成污染呢？那不很明显了吗。

下图是我的一个朋友来成都时，晚上到达双流机场给我发来的图片。我自己出去玩，回来的时候在双流机场也有类似的遭遇：空气无比之差，一下飞机就感觉到窒息！



4. 另外，去年 6 月我刚到成都的时候，空气还挺好。出租车司机告诉我，这段时间中央来了专家组调查雾霾，所以工厂都收工躲起来了。等到冬天的时候你看看，那雾霾可比北京厉害！现在你们明白了？

喂，中央专家组！快来看这里！:)

为什么成都多雾霾天气？空气质量那么差？

chris lin 做比目鱼，把自己埋在泥沙里

+ 关注

每逢开重要会议，空气就好了，你说雾霾的原因在哪

发布于 2016-11-15

著作权归作者所有

▲ 赞同 13 ▾ 感谢作者 ★ 收藏 □ 评论 2

5. (大年初七补充) 从大年初二下午开始，成都和附近地区的空气突然变好。一直处于优良的状态。全国的空气质量也有大幅度改观，绝大部分都是绿的。不要觉得这可喜可贺。这正好说明了雾霾不是自然现象，也不是因为汽车，而是人为造成的。想想为什么春节空气会突然变好呢？因为工厂放假了！等春节假期结束，你再看看 :)





成都人对于雾霾，几乎是集体的无知。大部分人不知道雾霾的危害，严重污染的时候，大街上也没有几个戴口罩的。有一次看我戴口罩，出租车司机对我说：“成都空气常年是这样的。戴口罩干嘛？适应了就好了……”

建议大家研究一下 pm2.5 颗粒物（大小在2.5微米的颗粒物）的危害。这种细微的颗粒物可以穿过肺部细胞，进入到其它器官里面。由于它们很小，所以经常是进去就出不来了，一辈子就留在里面。另外，pm2.5 颗粒物的主要成分，肯定也不是什么好东西。你们有没有想过，自己的慢性咽炎是哪里来的？为什么自己没抽烟还会得那种呼吸道疾病？

老实告诉你吧，我做过实验的。我本来没有咽喉问题。每逢空气污染比较严重的时候，我带着口罩出门就没事，带着口罩说再多话也没事。但是一旦脱下口罩，跟人说一会儿话，很快就感觉到咽喉不舒服。像腻住了堵住了一样，不断地想清嗓子……再想想吧，你们的慢性咽炎是怎么来的；）

当然，我猜雾霾的危害不止于慢性咽炎之类的疾病，虽然慢性咽炎已经够痛苦的了。上网搜一下，就会发现 pm2.5 颗粒物还会导致心脏病，肺癌等。患上这些病而死亡的概率是以 $10\mu\text{g}/\text{m}^3$ 的浓度变化递增的，而且不管是短期或者长期接触 pm2.5 都会起作用。这让我想起了 9-11 的大楼倒塌之后，附近人员吸入粉尘之后导致的各种奇怪的肿瘤和癌症……

2002 FOLLOW-UP STUDY RESULTS

- Each 10- $\mu\text{g}/\text{m}^3$ increase in PM2.5 was associated with increased risk of death:
 - 4% for all natural cause mortality
 - 6% for cardiopulmonary mortality
 - 8% for lung cancer mortality
- Positive associations with sulfur-containing air pollution, not other gases
- No consistent associations for coarse PM



(资料来源)

为什么 pm2.5 危害那么大呢？不仅因为它会穿过肺壁进入其它器官里面，而且因为它的化学成分。你想一下嘛，从化工厂的烟囱出来的颗粒物，里面可能有些什么东西？我还听说附近可能有烧垃圾的发电厂…… 我们吸进去的都是些什么啊，还进去就进不来了……

人们对于雾霾的误解，也有可能是因为成都的气候让人模拟两可。成都一直以来都以多雾出名，然而雾并不是雾霾。雾是水汽，是对人体无害的。我还清楚的记得小时候生活在成都，吸入雾气的感觉，是那么的清爽，沁人心脾！现在有了雾霾，很多人还以为是像以前一样干净的雾气，所以严重污染的时候还不戴口罩。

另外很多人还有一种误解，他们以为在室内就没有污染了。我也曾经以为那些颗粒物在室内就会沉降下来，从而没有室外厉害。我错了…… 使用 pm2.5 测试仪，你可以确认室内的污染其实是和室外差不多的，就算你把门窗关起来也不会有太大区别。下图就是我在一个商场里的饭店吃东西时的测试结果。在外面测也是差不多的结果。



想来也是啊，2.5 微米那么小的颗粒物，是很轻的，是很不容易沉降下来的。所以呢，不要以为你在外面有地方可以躲！除了在自己家里密封好门窗，打开空气净化机，否则你的室内空气质量跟室外不会有很大差别。

成都号称是“天府之国”，是宜居休闲的城市，还被联合国评为“适宜人类居住的城市”。成都市还想在人才和技术领域有更大的发展，建造了大规模的软件园等设施，引进各种技术公司。可是成都这几年的污染如此严重，比北京上海都要严重很多。连续不断的雾霾天，几乎没法出门，出门也不舒服，使得人的生活无比的沉重和郁闷，就跟蹲监狱差不多。

成都本来就属于经济相对落后的地区。由于其它的一些原因，成都的人才本来一般就是出去了就不再想回来，人才流失严重。外地人才大部分也不愿意来成都发展。现在雾霾让这种情况更加严重。我本来是考虑在成都有所作为的，然而为了自己的身体和心理健康考虑，我只好暂时放弃在成都发展的想法。我不会在成都买房，不会在成都创业，不会在成都工作，也不会推荐本领域的人才到成都工作。连现在成都市轻易可得的户口，我都不想去折腾了。我会在短期内离开成都，到空气好的地方生活。

敬告成都市政府：雾霾严重的伤害了市民的身心健康，给成都抹了黑，严重的损害了成都市的形象。如果不及时采取措施，它将会给成都的发展带来巨大的困扰。政府应该调查，追究和整顿违法排放污染物的工厂。成都市民：为了自己和家人的健康，你们也应该有所觉悟，有所行动了。

参考资料：

1. [世界卫生组织 关于颗粒物，臭氧，二氧化氮和二氧化硫的空气质量准则](#)

天才是不存在的

几年前我写过一篇文章，叫『[怎样成为一个天才](#)』。当时我煞有介事的告诉人们，如何才能成为一个天才，并且开玩笑似的把自己叫做“天才”。今天是双11，我想送给大家一份礼物。我要告诉大家一个秘密：我撒了一个谎。其实世界上不存在天才这种东西，“天才”这个概念根本就没有定义。

我写那篇文章，只是讲了一下怎样做一个合格的科学家，怎样更好的解决问题。那些所谓“成为天才的方法”，其实跟天才没有任何关系。至于我？我其实从来就没认为自己是天才，只不过我比很多人先意识到“天才的不存在性”而已。我曾经想成为天才，但后来我发现，“天才”这个词毫无意义啊，只是愚弄人的工具而已，所以就忙别的事情去了 :p

我知道有人会反驳说，爱因斯坦，费曼，图灵，或者邱奇 (Alonzo Church) 这些人不都是天才吗？我问你啊，你见过爱因斯坦，费曼，图灵，跟他们聊过吗？没有吧，那你怎么知道他们是天才？仔细想想，你头脑里关于他们的信息，都是哪里来的？

答案是，都是从别人嘴里来的！从老师嘴里，从本领域的专家们嘴里，从影视节目里，网络上，书上（传记）..... 总之，你没有亲自跟他们谈话。就算你亲自见过他们，你又怎么知道他们是天才？你是以什么标准判断的？

所以，说一个人是天才，其实都是道听途说来的。总有人喜欢吹捧一些人，把他们吹成“天才”。最后一传十传百，就真有人以为他们是天才了。最喜欢把别人吹成天才的人，莫过于传记作者了。

在『[图灵的光环](#)』一文里，我指出了『图灵传』作者的片面性，对图灵的各种吹嘘，对其他人的肆意贬低和歪曲。基本每一个“天才传记”的作者，都是这种德行。多年以前，我看 John Nash 的传记『美丽心灵』，它的作者口气跟『图灵传』有异曲同工之妙。他们都会强调一件事：这个人做的事情，其他人都做不了，想都别想！

为什么传记作者喜欢这样夸耀别人呢？因为他们就是靠那个吃饭的。他们很多根本不懂那些理论是什么，就跟猪头对于技术的理解深度差不多。这样的人又有什么资格来评判别人是不是天才呢？当然了，把一个人吹得天花乱坠，就会有很多人想买他们的书！大家都仰慕这位天才，想知道他为何那么聪明，所以就买了书。

还有一种喜欢把别人吹嘘成天才的人，就是他的同伴们：同学，同事，校友之类的。我在 IU 的时候，每次提出一个什么无关紧要的点子，总是有人想把我捧上去，鼓动我发表一些其实没啥价值的东西，把我吹得很神，弄得我都不好意思了。后来我发现，这种人并不是真的欣赏我，而是有他们自己的目的。

如果你把一个同伴吹成天才，你不就成了天才的同伴了吗？一传十，十传百，到时候大家都认为他是天才的时候，你就是天才的同学了。你就可以“不经意”地提起“当年我坐在他旁边那个办公桌”，“他的办公室就在我头顶上”之类的事情了；-) 当然你吹嘘的这个人，应该要比你强一些，不然你干嘛不吹自己呢。可是呢，这个人有多聪明，就不一定了。

图灵的天才名号，很大部分就是他的同僚或者接班人们搞的政治。计算理论领域的研究者们，几乎都是图灵的“利用者”，图灵是他们的工具。本来很简单就可以说清楚的原理，硬是要用图灵机搞得无比复杂，乌七八糟的。他们当然很愿意告诉你，图灵是一个天才，图灵机是无比伟大的发明，没有图灵就没有计算机科学！而实际上呢，没有任何一台早期的通用计算机接受过图灵的启发 (Enigma 机器不是通用的计算机)。

你一直被图灵机蒙在鼓里，直到有一天你听说一个叫 Neil Jones 的人。他用一种比图灵机简单很多的模型，解释和证明了完全一样的定理，甚至更加严格的定理，证明的方式也严格很多..... Neil Jones 是天才吗？相对于图灵，他更接近一个负责的科学家，然而他当然也不是天才。

不仅天才的名号是如此，很多业界的“牛人”，也都是这样吹起来的，比如 Google 很有名的那几个牛人。我见过有些人很喜欢说这样的话：“当年我在 Google 的时候，Jeff Dean 就在我头顶上那个办公室.....” :p

“天才”还有一种用途，就是从心理上打击你的对手，让他们紧张。我上大学的时候，有个同学知道我学习好，又自知超不过我。后来，我就发现他很喜欢在我耳边说这样的话：“你看人家某某做了什么什么，可比你聪明多了，上高中就已经是少年编程天才了！你现在才在学.....” 明白了吧？有人无法从能力上战胜你，他就想出一个心理战术，类似于激将法。他把别人叫做“天才”，用以激发你的嫉妒心或者心理不平衡，导致你最后失误或者崩溃，这样来战胜你。

很聪明就是天才吗？到底什么是天才？其实“天才”这个词，根本就是没有意义的。天才是怎么产生的？为什么会产生？等你有一天醒悟过来，才发现这一切都是政治！总有人喜欢利用“天才”这个称号，来达到自己不可告人的目的。或者让自己沾光，或者利用“天才”的名号来打压或者利用其它人。

这也就是为什么喜欢跟人“比聪明”的人，其实是很傻的。他们没有意识到，把自己的智力或者能力跟其他人比，其实让自己落入了居心叵测的人的圈套里，以至于被他们利用或者奴役。名义上叫你是“天才”，而其实呢，你只是一个工具或者奴隶。

我并不是说人一定不能被别人利用，暂时被人利用也是一种谋生方式。我只是在告诉你，历史上和你身边的很多所谓“天才”身上正在发生的事情。我并没有觉得这种事相对于人类社会的其它龌龊事情有什么特别不好的地方，我也没说你得为此做什么，改变什么。我只是觉得这是很多人不知道的信息，知道了是有好处的。

当然我并不否认爱因斯坦，费曼，达芬奇之类的人物是很聪明的。我很尊敬他们，然而他们是天才吗？..... 说到这里，这样的问题还有意义吗？世界上根本就不存在天才这种东西。所有喜欢给你提起“天才”，对他们各种夸赞和崇拜

的人，要么是有政治目的，要不然就是还被蒙在鼓里。

所以呢，不但天才出于勤奋，而且天才出于政治。“天才”这个词，根本就不应该在人们心目中继续占据重要的地位。世界上有远比聪明更重要的东西。去除“天才”在你心中的地位，你的心灵才能得到解脱，获得自由。这可能就是我最后一次跟大家提起“天才”这个词。

网络用语

不知道有人注意到没有，凡是在跟我的对话中使用过“吐槽”，“喷”，“low”，……这类词汇的人，都会被我自动在心理上进行隔离。也许他们对我用了这些词，也许对其他人用了，也许对他们自己用了。不管怎样，他们被我自动划为“另一类人”。

使用了这类词的商家或者 app，也会被我划到“低级”的行列。比如有的商家在请求评价的菜单里给出这些选项：“1. 给个好评。2. 我要吐槽。3. 残忍拒绝……”我只好对这种商家翻个白眼。

我也不知道怎么会这样，这本来不是我的理智做出的决定，而是一种自然反应。但现在我想来分析一下，我的潜意识为什么不喜欢这样的“网络用语”。

首先我对一个词的感觉，大部分都来源于它的字面或者发音。我的潜意识不会去想这个词有什么渊源，是音译过来的还是什么，它只知道“好听”还是“不好听”。“吐槽”的两个字，不管是发音还是含义，都非常的不雅，甚至是恶心。“吐”你知道的，很恶心。“槽”是什么？猪的食槽吗？听到这个词我首先想到的，就是一头很脏的猪，吃东西时吐了，吐在它的食槽里。不管网络字典如何解释这个词的渊源，它给我的第一印象就是这样的，非常难听，无法改变。

“喷”这个词也是差不多，感觉很恶心，不卫生，而且带有强烈的情绪特征。

那么“low”是什么问题呢？有些人喜欢说：“好 low。”虽然他是在说其他人“低级”，他自己却被我放到了低级的类别里。你发现没有，我用了“低级”这个词，这跟“low”不是一个意思吗？虽然词义相同，它们表达的说话人的态度，却是很不一样的。如果说“低级”，“低俗”，是完全没有问题的。要是他说“好 low”，就有问题了，自动降低了自己的身份。

“low”的问题就在于，作为一个中国人，明明有一个大家熟知的中文词汇“低级”，他却故意要用英文单词“low”。这说明这人的脑子里存在某种毛病，也许是自卑，也许是傲慢，也许是崇洋媚外，或者在显示自己会英文？总之，他不是他自己。他需要冒出英文来，才能在气势上压倒别人，这不正好说明他的地位其实很低吗？低不是问题，low 是问题。

我提到“地位”或者“身份”，可能会引起一些异议，难道我是在煽动社会等级和歧视吗？不是的。在我的心目中，人的“地位”不是指他的官职，衣着或者财富，而是他的文明程度。这类似于 Emily Post 的『[Etiquette](#)』（礼仪）一书中，对“best society”（最好社会，或者上等社会）的定义。一个人是否属于上等社会，不是看他有多少钱，他的官爵，而是他的礼仪和素质。更精确的说法，是文明程度。

之前说到的这些用语，都是在短时间内流行起来的，属于“网络用语”。为什么在中国人的社会里，网络用语层出不穷呢？我觉得其原因就在于很多国人“人云亦云”的心理，再加上很多人沉迷于网络，成天被各种信息洗脑。某富豪，某明星说了什么，他们就跟着转。朋友圈看到别人说了什么，他们也不管是否难听，就跟着用。在知乎，BBS 之类的地方“交流”太多，深受毒害。这就是为什么某些从没见过的词汇，忽然红遍网络，像“给力”，“也是醉了”，“不明觉厉”，“吐槽”，“喷”，“low”……

喜欢使用这类词汇的人，一般都属于“网民”，也就是大部分时间生活在网络空间里，在各种论坛，知乎，MITBBS 这类地方进行“交流”的人。他们在网络上交流那么多，在现实世界却找不到人说话，所以他们沉迷于虚幻的世界。这种人在网络上看似幽默，能说会道，在现实生活中就显示出很严重的自闭现象。他们往往不修边幅，不懂得基本的礼仪，言谈举止看上去俗气，粗鲁，不近人情，总是关心“网络上发生的事情”。

这也许就是我对使用这类网络词汇的人产生无意识反感的原因。当今中国泛滥的网络用语，其实是文化垃圾，是对中华民族文字和文化的侮辱。它们已经泛滥到广告里，电视里，书籍里，成为反复出现，乏味又恶心的口头禅。我们应该从自己做起，避免使用这类网络词汇。

另外对于“吐槽”，“喷”这类词，还有一个“类比心理”的误解问题。当我在指出一些事物的问题，说了一些实话的时候，有些人就产生了类比心理，说我在“吐槽”，“喷”或者抱怨。有些甚至开始跟我讲道，说：“无力改变的就接受……”这种人可能就是自己平时抱怨太多了，或者上 MITBBS 之类看别人抱怨或者对骂太多了，所以不知道还有一种态度叫做“调侃”。当他们看到有人调侃一件事，就觉得别人像他平时的态度一样，在抱怨。

抱怨属于弱者，属于那种无奈，没能力改变事物的人。而调侃属于强者，属于独立于事物之外，不受其影响，甚至有能力改变现实的人。

大部分时候我看到一些不合理的事物，我喜欢调侃。比如我经常调侃一些设计有问题，社区质量低还自以为是的程序语言。我看到 Tesla 或者 Google 又在吹牛，误导群众，我会嘲笑一下，调侃一下。我看到某些地方的市政设施做工拙劣，严重影响市容市貌，或者设计错误，花钱办傻事，我可能会拍个照片，调侃一下。

我在乎这些东西吗？它们对我有什么影响吗？没有。我都不用 Tesla 的产品，我不住的那个城市。我对这些不合理的事物并没有强烈的情绪。我只是从一个设计者和创造者的角度，提点一下更好的办法是什么，这样某些人可能会提高一下认识。当然，知乎和 MITBBS 上那些一本正经爱抱怨的人们，就习惯性地认为这是抱怨了。

对于程序语言就更是如此了。有些人说我“又在喷 Python”，就是完全没有明白抱怨和调侃的差别。我经常都是在调侃 Python，Ruby，JS 之类语言，而不是在抱怨它们。如果一个普通程序员，用 Python 时觉得很痛苦，却无法

改变它，他确实可能是在抱怨。而对于我来说，这些语言的设计者根本就是业余爱好者，还自以为是。我只能嘲笑或者调侃他们，而不可能抱怨。明白了吧。

所以有时候我只是幽默地嘲笑一下这些公司，这些设计者，同时利用信息的威力，间接地帮助每天都生活在这些地方，受到其影响的人。我不但独立于这些事物，不受它们影响，而且我有力量改变它们。利用我的洞察力和影响力，把正确的信息传播到很多人的头脑里，让他们认识到更好的设计和办事方式，逐渐引起社会的改变，这就是知识的力量。

AlphaGo Zero 和强人工智能

前段时间比较热门的是 AlphaGo (阿法狗) 的升级版 :AlphaGo Zero (阿法狗零)。跟阿法狗不同，阿法狗零不依赖于任何人类对弈记录，完全从围棋的规则出发，“自学成才”，推导出所有的战略，在与阿法狗的对战中完胜。有人问我，阿法狗零是否改变了我对人工智能的看法。我的回答是：没有。

我必须承认阿法狗零是个重要的成就。在以往的文章中我没能表达这种欣赏，有些人可能误解我，以为我对它的态度是不屑。不是那样的。我尊重 DeepMind 为此做出的努力，就像我尊重任何为我们做出重大贡献的人一样。阿法狗和阿法狗零的成功，对于我来说就像是一种新药研制成功，它能治好一种以前我们无法抵御的顽疾，而且可能开辟更加广阔的领域。显然它的意义是重大的。

但这是是否能改变我对人工智能的看法呢？还是不能。问出这种问题的人，应该只是道听途说，却没真正看过我的文章，或者怀着一颗找茬的心来，以至于没有理解其中的论述。仔细读过并且理解了我的『[我为什么不在乎人工智能](#)』（以下简称『智能』）一文的人，可能已经发现了，『智能』一文里面的论述，可以原封不动用到阿法狗零的身上，不需要做任何调整。阿法狗零实现了一个机械化系统所擅长的目标，这并没有什么意外。这个活动里面并没有理解，也没有智能，就像计算乘法一样。

实际上阿法狗零的出现和它的成功，都是意料当中的。要制造出战胜人类棋手的机器，本来就不应该需要依赖所谓“人类对弈记录”。这就像我们要制造出计算器，不应该需要很多人类手算账本的记录一样。完全根据加减乘除的规则，我们本来就应该可以制造出超越一切人类会计师的机器，不是吗？如果既阿法狗之后很多年，却没有人做出阿法狗零这样的东西，我还真有点意外了。对我来说，这东西一开头就应该做成阿法狗零那个样子。

正因为阿法狗零可以完全不依赖于人类棋谱，它更加强劲地印证了我的一个说法。那就是：围棋本来就是一种机械的活动，就像手算乘法一样，它并不是“人类智能”最高级别的标志。在很早以前我就是这个想法，那也是我不喜欢玩棋类游戏的原因。对于我来说，这种枯燥的机械化活动，伤脑子，不利于身体健康，还容易出错，本来就该造个机器去干。当然，我最讨厌的还是国际象棋。因为它不但横竖可以吃子，两个斜线方向也可以。一想到八面都可以受敌，我眼睛和脑子都涨了 :p

现在有人造出了阿法狗和阿法狗零，藐视了这个无聊却又被很多人膜拜的领域，我显然是拍手称快的。我应该感谢阿法狗的团队！

阿法狗零解决了围棋的难题，然而围棋并不能代表人类智能。按照之前“药”的比方，虽然这种伟大的新药治愈了一种顽疾，我们却不能因此就说它就是包治百病的神药。

每当说到这些，就有人很坚定地说：“虽然如此，我认为强人工智能一定能实现。也许不是几十年，几百年，也许是一千年，但终究是可以实现的。”他们所谓的“强人工智能”，就是用机器完全的实现“人类智能”。

对于这种“强人工智能信仰”，我是什么看法呢？一般说来，我不大喜欢具有“XX 一定能实现”这样强烈信念的人，这种信念使他们失去了理性。你说“一定”，你说“终究”，那只是一种信仰，而不是事实。具有科学精神的人，应该坦然接受“某些事情不可能实现”的可能性，不然他们可能因为信仰而继续制造永动机。借用一句广告词：一切皆有可能，所以也有可能“不可能” :)

别误会了，我可没说强人工智能是“永动机”…… 实际上它连永动机都还不是 :p

为什么这么说呢？因为“永动机”是有确切的定义的，而“强人工智能”没有。“强人工智能”这个概念，除了造出来让人工智能专家可以像哲学家一样扯淡，本身就是没有意义的。没有人切实的知道“人类智能”是什么，所以我们根本就没有资格谈“强人工智能”，用机器来实现“人类智能”。说强人工智能一定会实现，就像在说：“有个东西，我们不知道它是什么，但它一定会实现！”这不是无稽之谈吗？

所以想造永动机的工程师虽然终究会失败，他们至少知道自己在造什么。相比之下，想实现强人工智能的工程师，连自己在造什么都不知道自己在造什么都不知道……

我再打个比方你就知道这有多可笑了。如果我们把一辆汽车通过时间机器送到石器时代，原始人看到这辆汽车，知道它可以跑，力气还很大，但是他们不能拆开它。他们每天都开着它运输各种东西，观察它的工作，但是他们看不到车子里面是怎么运转的。他们不知道里面有引擎这种东西，更不知道它的构造，他们不知道汽油的存在……

总之，这辆汽车的内部构造是一个黑匣子。请问，这些原始人什么时候能造出他们自己的汽车呢？几十年，几百年，几千年，几百万年？

我的答案是：我们根本没有资格讨论这个问题。一些原始人碰巧捡到一辆汽车，就像『[上帝也疯狂](#)』里的故事，天上掉下来一个可乐瓶子。结果有人就开始问“原始人什么时候能造出自己的汽车”。想点别的问题行不行？有那么多有趣的故事可以演绎，为什么一定要是原始人造出自己的汽车呢？有可能他们在造出汽车之前，就因为自己的贪欲而灭绝了呢？:p

对于“强人工智能”，我的看法也是类似的。在我们真正理解人类智能到底是什么之前，“强人工智能”这个词并没有确切的定义。所以谈论强人工智能是否可以实现，什么时候实现，是没有意义的。我们还没有资格讨论这个问题。

(如果你喜欢这篇文章，可以考虑[付款购买](#)。)

理性的力量

曾经有一个显得自己地位挺高的人给我来信，谈论我的博客和一些他对工程的看法。这两天回忆起过去的一些经历，想把这段故事讲一下。当然我不会点名这个人是谁，他只是有类似想法的人其中一个。

他说，你应该等自己有了地位再说那些话，那时候听你说话的人会多一些。我当时给了他一个礼貌而中肯的回复。我告诉他：“我不需要到达所谓的‘地位’才说话。我说的话，价值就在这些话自己身上，它们不需要依附于任何人的地位或者名气来起作用。实际上，许多人都在听我说的话，因为他们理解了其中的内容。”

这段回复，其实道穿了我所看到的一些社会现象和误区。现在很多人判断一个说法对不对，只看说话人的“地位”。比如，他们会提起某富豪，或者某大公司总裁说：“人工智能将会颠覆所有人的工作。”但是证据呢？没有证据来支持这种说法，光是有“地位”和“财富”在那里。在有些人的意识里，有钱有权的人说的都对，这是很悲哀的。

我的话有什么不同呢？因为我不但告诉你“是什么”，而且我告诉你“为什么”。所以一个人要相信我说的话，他不需要知道我是谁，我也不需要什么地位。这句话有它自己的力量，这就是理性的力量。就像一个数学定理和它的证明，我不需要告诉你这个定理是什么大数学家或者天才提出来的。我只需要告诉你这定理说了什么，然后按部就班把它证明出来。

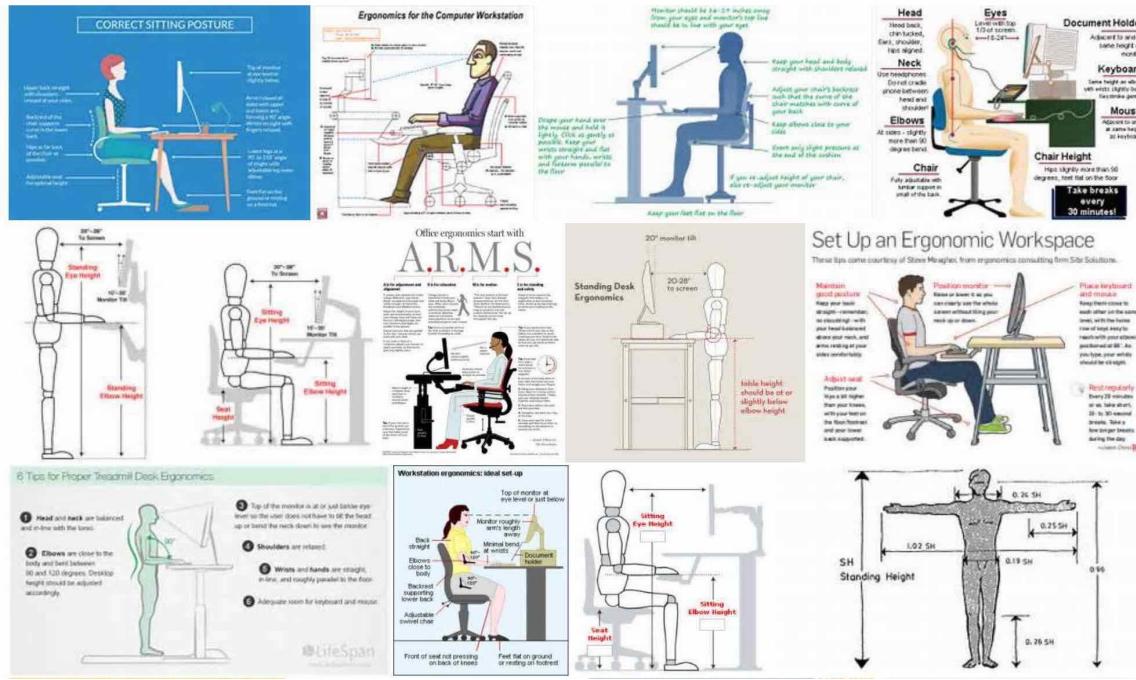
这就是为什么那位来信的人，没能理解我为什么要写博客，为什么可以告诉人们一些事情，为什么许许多多的人在看这些博文，甚至付钱购买它们。我完全可以用一个化名来干这些事，效果是一样的。为了说这些话，我不需要达到其它的目标，不需要有什么惊天动地的“成就”，不需要坐到大公司的高层位置上。

这些话有它们自己的地位，它们不属于我，不依赖于我，它们本身就是一种成就。如果有人想用我的话去说服另一个人，他不需要告诉他：“王垠说……”他只需要告诉他是什么，为什么，就像转述一个定理一样。这就是为什么王垠打倒了很多人心中的权威，而他自己并没有成为新的权威。他说了很多有价值的内容，却经常被一些刚学会 Python 的初学者鄙视。这就是所谓“上善若水”吧。

如果按照那个人的说法，等到我有了“地位”再来说话，恐怕到了那个时候，我就不再能说出这些有价值的话来了。要想混到大公司的高层，在那尔虞我诈，勾心斗角之后还保留一颗能够说出这些话的心，几乎是不可能的。即使它存活下来了，处于自己的地位，也必然会冠冕堂皇。所以“等到有了地位再来说话”，这种说法其实是想封我的嘴，让这些信息被埋没，让事实无人知晓。

我很明白这些想法的价值，所以我不会听信此类言论而停止我的表达。理性的力量会发扬光大。

人体工学



长时间用电脑工作的人，都会开始关注自己的健康。很多人出现腰肌劳损，颈椎病，肌腱炎之类的烦人问题。其他人呢，因为看到身边的同事朋友得了那些疾患，也开始小心起来。通常人们的解决方案，是求助于“人体工学”(ergonomics)。很多公司请来专业的人体工学专家，为员工咨询和调整他们的座位，姿势和工作方式，为他们提供各种昂贵的人体工学产品，比如各种人体工学键盘，鼠标，椅子，桌子，等等。

作为一个长期大量使用电脑工作的计算机专业“前辈”，我从来没有因为用电脑而出现这些毛病（近视除外）。我却惊讶的发现，小我十几岁的非计算机专业人士，因为朝九晚五用电脑而产生的毛病，居然比我这个下班还趴在电脑前面的人还多。所以我今天想抽空讲讲我对这些问题的看法。

当我在[微博](#)上提到我打算谈谈人体工学的问题，很多人就开始猜测我要说什么。有的说我可能会提到某种昂贵的椅子，有的请求我推荐键盘和鼠标，有的问我是否应该采用“站立式办公”，有的问我颈椎病的治疗和康复……

我确实会讲其中一些问题，不过很显然，我不是一个医学专家。我从来没得过颈椎病，腰肌劳损或者肌腱炎。所以如果你已经患了这些毛病，我恐怕无法告诉你如何康复。我没有资格对这些问题负责，所以关于这些问题，你最好去请教医生。

我在这里所能做的，只是描述我的经验：一个本来就健康的人，如何在使用电脑的时候继续保持健康。另外，我还想解释一下关于人体工学的误区。

当然所有这些都会因人而异，这里只是我的个人经验。虽然我希望它们有所帮助，但请勿把它们作为教条，生硬的照搬。我不对你们盲从我的话而产生的健康问题负责 :p

所谓“人体工学”

先开门见山谈谈我对“人体工学”的看法吧。你也许发现了，似乎任何产品都可以打上“人体工学”(ergonomic)的名头，而不需要任何官方的认证。食品和药物需要 FDA 认证，有机食物需要 USDA 认证，可是“人体工学”呢，没有任何机构来认证它。我并不是说 FDA 之类的机构值得信任，有个认证机构，至少不至于完全胡来吧。

这意味着什么呢？这意味着打着“人体工学”名头的产品，并不一定是经过真正的研究产生的。它们并不一定符合人的生理构造，有些反而会损害人的健康。“人体工学”并不是一门经过实验检验的科学，它甚至不是一门正经的学科。它其实跟减肥药有很多相似之处。

为什么卖减肥药那么赚钱？因为它是一个模糊的领域，没有人知道什么是对的。没有政府监管，没有科学实验，没有临床对比，难以取证，难以追寻其因果关系。大部分的减肥药都是不起任何作用的“安慰剂”(placebo)，它们的作用都是使用者自己想像出来的。吃减肥药的同时也控制了饮食，最后少了几斤，就归功到减肥药身上。这就是所谓“信则灵” :p

要是吃了不起作用，很多人都会以为是因为自己的吃法不够“虔诚”。要是一种减肥产品不起作用，你没办法起诉减肥药公司。比如某种减肥产品，号称吃三个疗程就能减掉50斤，无效退款。你吃完三个疗程，发现体重根本没变，去找

他们算账，你能拿回你的钱吗？不能。

这是因为，减肥药公司总是可以挑出你的岔子来。他们可以说因为你没有严格按照他们的规定吃药，或者质疑你到底有没有吃够计量和疗程，或者说你的体质不符合这样那样的条件，或者你有其他的身体问题，或者你吃了不该吃的东西，或者说你谎报了吃药之前的体重…… 总之，你没能成功减肥肯定有其他原因，而不是因为这个减肥药不起作用！

懂了吗？人体工学的市场也是一样的原理。如果你控告人体工学设备不但没能帮助你，反而损害了你的健康，或者你控告人体工学专家误导了你的办公姿势，导致你得了颈椎病，这跟你控告减肥药公司是一样的结果。

你说用了我们的人体工学键盘，结果导致了肌腱炎，你有什么证据呢？注意，你必须证明“使用我们的键盘”和“患肌腱炎”这两件事之间有“因果关系”，而不只是“同时发生”。也许你有其他不好的生活方式，导致了肌腱炎呢？没人能证明你平时在做什么，没人能证明你没有其它损害健康的生活方式，所以也没人能证明是这个键盘“导致”你得了肌腱炎……

几乎所有关于人体工学产品的好处，都是道听途说，没法用科学实验来验证它。有人跟你说这个键盘好，可以防止RSI。可是他从来就没得过RSI，又何以证明可以防止RSI呢？要是他不用这个键盘，也许同样不会得RSI。也许他不得RSI只是因为他姿势健康，注意休息，也许因为他打字慢，或者也许因为他根本就不怎么用电脑呢？

你听信了他的话，买了这键盘，却发现多年以后自己得了RSI。你能怪谁呢？推荐键盘给你的人，制造键盘的公司，都不会承认他们有责任，因为你找不出确凿的证据来，你没法证明“因果关系”。

所以对于所有的“人体工学产品”，你只有靠自己去感觉，去判断它到底能不能起到效果。很多时候这种感觉相当的模糊，因为你没法知道一个键盘或者鼠标用久了会怎么样。也许它开头用起来不顺手，后来适应了发现更好。也许它开头感觉很高效，但时间久了就出现健康问题。

这就是为什么我想在这里分享一下我对“人体工学产品”的经验。我希望引起人们对“人体工学这个词”的警惕，多长一个心眼。不要盲目相信人体工学设备厂商，甚至人体工学专家的建议。我发现很多被公众广为接受的习惯和姿势，很多流行的“保健”工作方式，都是没有经过实践检验的。

桌子，椅子和坐姿

用电脑的元素包括了环境，姿势和设备。环境包括了房间，桌子，椅子等。设备包括了键盘，鼠标等。我想首先从环境和姿势来讲这个问题，因为环境和姿势的差别，决定了对设备的不同使用方式。

常见的“健康坐姿”示意图，基本是这个样子。这也是公司请来的“人体工学专家”会推荐给你的姿势。



很多年前，为了实现这样的坐姿，人们基本都用这样的电脑桌：



这坐姿的要点可能大家都知道，我就不在这里废话了，总结下来就是“正襟危坐”。因为这种坐姿是人体工学专家推荐的，是书上写的，当年很多人就照办了。现在很多都不再使用电脑桌，不再遵循这些姿势，可是从这些你至少可以看出，所谓“人体工学”可以荒谬到什么样的程度。

十多年前，我也曾经用过两三个这样的电脑桌。我也看过很多遍类似的“健康姿势图”，我也试图采用这样的姿势。可是十多年后的今天，我再看这幅图，这种电脑桌，都觉得是陈年的笑话。为什么会有人相信那是最“健康”的姿势？为什么有人会把键盘放在那样的托盘里，打键盘时手肘悬空，键盘的前面是抬高的，还要两眼直视前方的显示器……

这些对我来说，都是不舒服，不健康的姿势和做法。我很确信，画出这种姿势图的所谓“人体工学专家”，其实不懂什么东西。这些所谓“健康姿势”，其实没有经过科学实验的检验。把这姿势发扬光大的电脑桌设计，也没有经过实践的考验。

这些所谓“健康姿势”，其实是某些人凭空想出来的。就跟现在很多所谓“健康专家”跟老年人胡扯一样，今天说吃这个不吃那个，明天说吃那个不吃这个。今天说这个有毒，明天说那个有害，不要用微波炉，不要用空调，不要开暖气，小心手机辐射！……而其实呢，这帮人自己都不懂科学，不知道是怎么回事。

你知道在十多年前，我用台式机，用这样的电脑桌的时候，我的键盘放在哪里的吗？不是在它专用的托盘里，而是在桌面上：p 因为在十多年前，我就注意到了一个问题：把键盘放在托盘里，用起来会很不舒服，手会累得比较快。所以后来我就把键盘直接放在桌面上。我把显示器尽量往后面退，这样桌面上就有足够的空间，可以放下键盘和鼠标。

然后我发现，打键盘的时候，手肘不应该像“标准姿势图”里那样位于身体侧面，而且悬空，那样会很累。可能很多人都发现了这个问题，所以手肘必须靠在某个地方。很多人把键盘放在托盘里，所以他们只有把手肘靠在椅子的扶手上。可是扶手不是那么稳定的东西，所以他们仍然会很累。由于我把键盘放在桌面上，尽量把显示器和键盘往前靠，这样桌面上就留下足够的空间，可以把我的两只胳膊和肘都放上去。

很多人出现肩膀和肩胛骨后面很酸的情况，我觉得就是因为他们按照“健康姿势”的结果。如果你的小臂不靠在桌上，肩膀上和背部的肌肉就得用力支撑起小臂的重量，始终处于一种僵硬的状态。久而久之，哪些部位就开始酸痛，劳损……

因为显示器，键盘，鼠标，我的手和胳膊，全都在同一个桌面上，所以这样的姿势稳定又舒服。所以，我就发明了自己的用电脑姿势。我没有照片，你们可以自己想象一下：

1. 不要用专用的电脑桌。用普通的，宽大的，稳重的办公桌，没有键盘托盘那种。电脑桌一般都是廉价刨花板拼成的，它带来的不稳定感，也会导致疲劳。桌子不要用玻璃面的，会很冰很滑，木头桌子舒服很多。如果没有地方放办公桌，用饭桌也可以。
2. 键盘和鼠标放在桌面上（而不是托盘里）。如果是笔记本电脑，就直接放在桌面上。
3. 显示器放在键盘和鼠标前面（不要垫高）。如果是笔记本电脑，就直接用它的显示器，尽量不用外接显示器。
4. 眼睛不要平行直视显示器，因为那样一直抬着头很累。显示器可以处于比眼睛低一点的位置。稍微俯视一点，有一点埋头，才是自然的姿势。但也不要放得太低，否则脖子会弯得很累。
5. 两只胳膊，连同肘关节在内，全部趴在桌面上。如果趴不下，就把键盘往前推，直到你趴下为止。如果还趴不下，也可以把肘关节放在外面。想象一下，你要趴在桌面上打个盹是什么姿势？基本就像那种感觉：）
6. 桌面最好是可以通过手摇或者电动升降的。尽量把它升高，这样可以支撑上身的重量。如果太高了感觉不舒服，可以往下降一些。不过总的说来，高的比低的好。
7. 如果肘关节在桌面靠久了痛的话，稍微降低桌面的高度，或者垫个软的垫子在那里，或者把肘关节的那块骨头挪到桌面外面去。
8. 椅子的高度足够让膝盖不要比髋关节高就可以了。
9. 椅背最好高一点，休息的时候可以把头靠在上面最好。椅背是什么曲线无关紧要，因为用电脑的时候背不靠在上面。它只是休息或者思考的时候用来靠着。
10. 椅子最好有扶手。这样休息或者思考的时候可以靠手，但打键盘的时候不要把肘靠在扶手上，趴在桌上就行了。

你可能注意到了，我没有推荐一万块钱的“人体工学椅”。我觉得买那些昂贵的椅子完全是多此一举，坐很普通的椅子

照样能保持健康。

站立办公

最近有很多人流行站立办公。我也试过站立办公，可是实话说吧，我觉得站久了太容易累，其实并不是好事。所以我大部分时候还是坐着，有时候坐疲倦了也许把桌子升起来站站。可是一般都不会站太久，因为站着真的比坐着累！

我还试过坐在健身球上面，站着办公时踩在一个“平衡板”上。最后那些东西都被我丢在一边不用了。站着不如坐着，坐着不如躺着，这是千古不变的真理！

变换姿势，休息，伸展筋骨

这只是一个大体的姿势，绝大部分打电脑的时间，我是用这个姿势。我并不是说就这样连续工作很多小时不变姿势。保持健康很重要的一件事，就是不时的换换姿势，伸伸懒腰，觉得怎么舒服就怎么样。思考的时候，当然就往后靠一下，头往后仰一仰。如果在办公桌做久了，你可以拿起笔记本电脑，走到公司的小厨房，咖啡厅，或者到沙发上坐坐。

关于姿势，我推荐大家都观察一下猫的姿势，因为我受到小莫奈的很大启发。



跟猫在一起久了，你就会发现这个家伙怎么有那么多种姿势睡觉。怎么舒服怎么睡，有时候扭成你从来没想象到的姿势。然后你去学一下它的姿势，就会发现那还真的挺舒服。原来世界上除了正襟危坐，还有那么多种舒服的姿势！

这就是我要告诉你的：不时的换换姿势，试试一些奇怪的伸展姿势，伸懒腰！

另外，你还可以试试休息的时候做一些“瑜伽姿势”。很多人想到瑜伽，就想到去上瑜伽课，花个一小时，其实你不需要花那么多时间。我觉得你只需要学会两种最基本的瑜伽动作：downward dog 和 child pose，就能有很多好处了。把这两个动作保持一分钟，你就会看到效果。



其实这些姿势都是跟动物学来的。瑜伽的 downward dog 姿势，其实就是猫伸懒腰的动作（应该叫 downward

cat)。小莫奈还经常做出类似 child pose 的动作。



如果你想了解更多的姿势，可以去上点瑜伽课，或者养一只猫，学它的姿势。

健康指法

现在我想谈一下打键盘的指法问题。指法习惯的改善，也许可以改善肌肉的紧张程度，避免出现一些毛病。上句话说“也许”，是因为我从来没有出现过那些毛病。或许我的指法更健康，但也许有其它的原因，也许是多方面的因素。所以是否采用我这里的建议，还请你斟酌。

改良盲打指法



很多人恐怕都是严格按照上图那种“标准盲打指法”打字的，而我发现那并不是最科学，最健康的指法。观察这个指法图，它只是生硬的把按键按直线划分成几个部分，把它们分配给各个手指。

起初你也许没有发现问题，但是时间久了你就会发现，小指负责的键实在太多了！中指和无名指这么长的指头，才管 4 个键，右边小指却要管 16 个键，是中指和无名指的四倍之多！左边的小指也要负责 11 个键。这是很不公平的。

食指，中指，无名指都比较长，可以轻松地摸到它们负责的键。而小指由于相对短小，除非你移动手掌，否则就一定困难。我发现小指不容易摸到的几个键是：第一排的 0, -, =, delete，最下面的 ctrl 和 alt。

当然，你可以强迫自己用小指去打那几个键，但这也许并不是最舒服，最健康的方式。我在比较早的时候就观察到了这个问题，由于小指相对比较短，我必须移动整个手，或者比较吃力的伸出小指，导致手心变形扭曲，才摸得着这几个键。虽然当时不是很明显，但长久这样，也许会引起一些健康问题。

解决的办法很简单：

1. 用无名指去打第一排的 0, -, =, delete。
2. 不要用右边的 ctrl 和 alt 键，只用左边的。
3. 使用左手拇指去按左边的 ctrl, alt 和 command。

现在我们来做一个实验，你就知道为什么这样更合理。把右手放在“盲打位”，稍微伸直手指。以手掌为中心，向右旋转右手，扫过键盘，注意小指和无名指扫过的键。你会发现小指能扫到的键，只有第二排的 P, {, }, \。这说明什么呢？这说明小指根本不适合去按第一排的几个键，它处于自然状态的时候，根本够不着那些键。

稍微伸直无名指，不要挪动手掌，你会发现这根的手指，完全没有问题摸到第一排的按键：0, -, =, delete。实际上，我刚才就是用的无名指一一打出了这几个字符。完全不需要移动手掌，也不需要用力伸，就能用无名指轻松按到 0, -, =, delete。我甚至能摸到 delete 键的最右边。

发现这个秘密之后，我开始练习用无名指去打这几个键，很快就适应了。这样右小指的负担减轻了很多，我完全不需要移动手掌，而无名指对于这点事情游刃有余，没有任何怨言。

另外，小指去按 ctrl 和 alt，也不是最舒服的位置，这需要把小指弯成勾状才能够得着它们。我用的是 Mac，所以按理我的小指还该负责 command 键。这对左手小指也是非常不公平的。

所以我想出了另外一种安排：让左手的拇指去按 ctrl, alt (option) 和 command。因为这几个键都在空格键旁边，而空格键是拇指管的，所以拇指去按这几个键似乎也顺理成章。另外我自己从来不用右边的 command 和 option，所以右小指的负担又减轻了。

另外，很多 Emacs 痴迷者仍然痴迷于用 ctrl-f, ctrl-b 之类的键来移动光标，这也是很不健康的。我对此的解决方案是：用方向键，或者 command-left, option-left 一类的“方向组合键”。当然，我不用小指去按方向键。我直接把手移到方向键上面，用食指，中指，无名指去按它们。

于是乎，我按组合键的方式总结下来是这样的：

1. 需要按组合键的时候，跳出通常的“高速打字模式”。
2. 如果组合键在左边（比如 command-c, command-s），那么用左手拇指（不是右手）按住 command，同时用左手食指按另一个键。也就是左手单手完成组合键，就像“捏”了一下键盘。
3. 如果组合键在右边（比如 command-i, command-p），那么用左手拇指按住 command，同时用右手相应的“盲打指”按另一个键。这种组合键同时用两只手完成。
4. 不用 ctrl-f, ctrl-b 一类的 Emacs 组合键来移动光标。使用方向键。用食指，中指，无名指去按它们。
5. 有些时候完全不用固定的指法按组合键，想怎么按怎么按。比如，用右手的拇指和食指去按左边的 command-s，用右手的食指和中指去按左边的 command-r（刷新网页）……

避免三键组合

也许有人问，如果遇到三键组合，或者更多的，那怎么办呢？首先，我都是尽量避免三键及以上组合的。三键组合是反人类的设计，不但难记还不健康。这种组合只应该用在很极端，不常发生的情况，比如重启电脑 (ctrl-alt-delete)，或者强制杀死 GUI 程序。可惜的是，很多程序员用的 IDE 和编辑器，都把一些常用的功能放到三键组合上面。

比如下图是 IntelliJ 的一些常用组合键：

Editing	Navigation
Basic code completion	Ctrl + Space
Smart code completion	Ctrl + Shift + Space
Complete statement	Ctrl + Shift + Enter
Parameter info (within method call arguments)	Ctrl + P
Quick documentation lookup	Ctrl + Q
External Doc	Shift + F1
Brief Info	Ctrl + mouse
Show descriptions of error at caret	Ctrl + F1
Generate code...	Alt + Insert
Override methods	Ctrl + O
Implement methods	Ctrl + I
Surround with...	Ctrl + Alt + T
	Go to class
	Ctrl + N
	Go to file
	Ctrl + Shift + N
	Go to symbol
	Ctrl + Alt + Shift + N
	Go to next / previous editor tab
	Alt + Right/Left
	Go back to previous tool window
	F12
	Go to editor (from tool window)
	Esc
	Hide active or last active window
	Shift + Esc
	Go to line
	Ctrl + G
	Recent files popup
	Ctrl + E
	Navigate back / forward
	Ctrl + Alt + Left/Right
	Navigate to last edit location
	Ctrl + Shift + Backspace

像“Go to symbol”（跳转到符号）这么常用的功能，居然绑定到一个四键组合上面。另外，“往回跳转”（Navigate back）也很常用，却绑定在“ctrl-alt-left”这么别扭的三键组合上。

Emacs 的组合键更加反人类。不但有大量的三键组合，还有“组合键序列”。比如 ‘C-x C-s’ 是保存文件，它表示按了 ctrl-x 之后，继续按 ctrl-s。你还记得小指按 ctrl 是多么的别扭吗？这表示每保存一次文件，某些人需要连续别扭两次 ;)

更加反人类的设计是 ‘C-x b’ 这样的 Emacs 组合键。它要你按 ctrl-x，接下来放开 ctrl，然后按 b。如果是 ‘C-x C-b’ 还稍微好一点，至少我不需要记得在按完 ctrl-x 之后必须放开 ctrl 才按 b，我可以一直按住 ctrl，然后另外一只手依次按 x，然后 b。按这种键的时候经常会不自觉的继续按住 ctrl，结果按成 ‘C-x C-b’，那是另一个功能！

所以如果发现经常要用这样的组合键，我就想法把它换成两键的组合键。如果这个功能用得很频繁，我会干脆给它绑定一个“Fn键”（其中 Fn 表示 F1, F2, ...），这样“一指禅”就能触发它。

有些人会觉得用 Fn 键很不“专业”，像是非计算机专业人士的做法，还想“一指禅”……他们觉得，记得住那些复杂的组合键，甚至方向键都不用，全用 ctrl-f, ctrl-b 一类的来移动光标，才像是个专业的“黑客”。我不这样认为！垠神的存在，不是直接否定了这种说法吗？;)

事实上，由于世界上许多黑客对这种组合键的迷恋或者装逼心理，导致了他们全都不如垠神。对复杂组合键的迷恋，说明他们没有理性的思维，所以他们会在更多其它方面走火入魔，写出复杂难懂的代码，设计出很不人性的产品。

我喜欢把 IDE 最常用的功能放在某个 Fn 键上面，然后把这种功能的“变种”放在“command-Fn”或者“option-Fn”上面。比如，我把 F9 设定为“在这个位置放书签”，把 command-F9 设定为“跳转到下一个书签”，把 option-F9 设定为“显示书签列表”。这有什么好处呢？好处就是，关于“书签”的功能都在 F9 上，或者含有 F9 的组合键上。我不需要去其它地方找它们。

如果遇到那种不能改键绑定的软件，我按三键组合的方法很简单：用两只手，完全不按指法，怎么舒服怎么按！

关于“打字效率”的思考

也许你觉得这样按组合键导致左手离开盲打位，会降低输入的速度，然而我觉得它并不会减慢很多速度。原因是按组合键的时候，我们一般都处于一种思维的“间隙”。比如 command-s 存盘，一般都是等你打完一句话，或者一个函数。再比如 command-c, command-v 复制粘贴，一般都是当你用鼠标选中了文字，用鼠标选中了插入点之后。

在这种“思维间隙”的时候，打字的“流水线”本来就已经中断了，导致了速度降低。所以你的手离开打字的“盲打位”，其实并不会带来很多的不便。

另外我必须提醒大家，不要一味的追求打字的速度。除非你的工作是专业打字员，不动脑子，专门帮别人输入已经写好的文档，否则你其实不需要非常快的打字速度，一刻都不停。不管是写程序，写文章还是做设计，思考的优先级都是高于打字的。思考是需要时间的，而我思考的速度，几乎总是比我能打字的最高速度慢。

所以为了思考，我打字总是需要停下来，等我的脑子跟上。很多人不但打字快，而且他的手不等他的脑子，所以这种人经常写出很多垃圾代码，垃圾文章 :p 如果你稍微对代码或者文章负责一些，就会发现为了按组合键而损失的那点时间，比起平时为思维停顿的时间，那真的不算什么。

现在我已经讲完了用电脑的环境是姿势，接下来的几节，我想谈谈我对各种电脑输入设备的看法。这些设备包括，键盘，鼠标，触摸板，轨迹球，小红点等工具。

键盘和鼠标

有些人请求我推荐人体工学键盘和鼠标，所以我想从这些电脑最重要的输入工具谈起。很显然，不好的键盘或者鼠标，会增加你患各种毛病的几率，比如肌腱炎。然而是不是花一万块钱买最好的人体工学键盘，最好的鼠标，就可以解决这个问题呢？我觉得不是这样的。

先老实告诉你，我用的是什么键盘和鼠标吧。

- 绝大部分时间，我用笔记本电脑自带的键盘。我不用任何外接键盘，包括苹果的蓝牙键盘。打键盘的时候，我的两只胳膊成“自然角度”，而不是垂直于键盘。
- 绝大部分时间，我用笔记本电脑自带的，键盘下面那个触摸板。我一般不用力把它按下去，我打开了它的“轻触点击”功能。我不用外置的苹果蓝牙触摸板。我不用鼠标，包括苹果的鼠标在内。
- 少数使用台式机的时候，我喜欢用戴尔电脑自带的普通键盘。我不使用任何人体工学键盘。用戴尔键盘的时候，我把它放平，而不用背后的支架把它撑起来。我使用戴尔电脑自带的鼠标，或者简单的无线三键鼠标。

总而言之，我所有的工具基本就是一台 MacBook Pro，2013 年产的。我没有任何其它人体工学电脑产品。在换用 MacBook 之前，我用一台 ThinkPad T60，长达 6 年之久。在微软工作的时候，我用一台 ThinkPad X1 Yoga，昏天黑地的写代码也没得病。我只用它自带的键盘和触摸板，不用外接键盘或鼠标，不用 ThinkPad 的“小红点”。

是不是很惊讶？我不用任何人体工学键盘和鼠标，成天趴在一一台笔记本电脑上面写代码，写文章，做研究，上网……经常时间相当长，却没有患上任何毛病？

我并不是没有试过其它的键盘和鼠标。较早的时候我买过微软自然键盘，好几种其它人体工学键盘。我还买过比较贵的机械键盘。我试过各种人体工学鼠标，轨迹球，苹果的大号蓝牙触摸板。我考虑过买很贵的，形状特别的 Kinesis 键盘，Data Hand，……

最后，我放弃了所有这些号称可以拯救健康的人体工学工具，只留下一台 MacBook。为什么呢？我现在来讲一下，我发现的这些人体工学输入设备的问题。

台式机键盘



我总是尽一切可能使用笔记本电脑，不用台式机。在迫不得已要用台式机的时候，我也不用那些奇形怪状的“人体工学键盘”。那么我用什么键盘呢？我就用戴尔台式机自带的键盘：）

当然我不是图便宜舍不得买键盘，而是我真的觉得那个键盘比别的键盘舒服。我试过挺多其它键盘，却发现这个默默无闻的便宜键盘最舒服，为什么呢？

分析其原因，我觉得有几点很重要：

1. 这个键盘几乎没有边框，不占多余的空间，所以我的手掌可以直接靠在外面的桌上。
2. 它非常轻巧，所以我可以随时把它挪到桌上任何我觉得舒服的位置。
3. 键程和力度都比较舒服。
4. 它很简单，没有多余的按键和功能。

对于第一点，我想再解释一下。很多人喜欢那种连着一块垫子的键盘，以为把手掌垫起来更符合“人体工学”。如果用不带垫子的键盘，他们也喜欢买个垫子放在下面，就像这样：



我也买过垫子，可是经过很多次的试验之后，我发现那并不舒服。其实最舒服的设置，是把手掌直接放在键盘外面的桌上。由于戴尔键盘几乎没有边沿，空格键下面直接就是桌面，所以你可以很方便的把手掌放到外面去。

为什么我不喜欢垫子呢？因为我觉得“人体工学”很重要的一个事情，是能够随时根据自己的坐姿来挪动键盘的位置。让键盘来顺从自己的位置，而不是让自己去顺从键盘的位置。很显然，有了这种垫子，键盘的位置就被固定在那里了。要想移动键盘，你必须移动两个东西，就不是那么方便了。而且垫子下面一般都有胶皮，是没法在桌上滑动的。

另外，这些垫子远远没有桌面那么稳定。手掌靠在上面如果没有稳定感，人就会轻微的紧张，久而久之就容易累了。

很多人还喜欢把键盘前面的支架立起来，就像这样：



我觉得这是错误的作法。因为键盘前面抬高，会迫使打字的手指往上抬，导致小臂肌肉和肌腱更加紧张。最放松的作法应该是把键盘放平，或者让前面稍微低一些（而不是高一些）。由于我没有找到好的办法可以让键盘前面低下去而不损失其它方面的要求，所以我一般就把它的支架收起来，把键盘放平。

微软自然键盘



最常见的人体工学键盘，恐怕就是微软自然键盘了。这种键盘把两只手控制的键，从中间分开成两半，中间留一个间隙。这样两只手分别放在一边，成一个角度。注意，我并不推荐这个键盘。我只是想通过它的一个特征，介绍一种健康的打键盘姿势。

很多人认为这样两只手成一定角度是更健康的，确实如此。很显然，如果你的两只手端端正正，垂直于键的方向打字，由于肩膀比两手的间距宽，所以手腕会呈现一种不自然的扭曲状态，久而久之肯定会出现问题。

但你其实不需要微软自然键盘，就可以防止那种别扭的姿势。虽然笔记本电脑键盘的按键都是端正排列的，我用它们的时候，两只手却都是成“自然角度”打字。你可能会说，键盘是正的，你的手要是放成那种角度，怎么能准确地打字呢？事实是，你确实可以把手放成自然的角度，却仍然准确地使用任何普通键盘，你并不需要把键盘的按键转一个角度。这是因为人的手指是很灵活的，它们可以方便地移动到任何能够触及的位置，所以你只需要稍微适应一下，就可以准确地打字了。

那么为什么我不用微软的自然键盘呢，它的角度不是很好吗？我不用它的原因，是因为它用起来其实不舒服，造成了紧张和疲劳。这种键盘把按键从中间分开，食指控制的键旁边留有一段空白，所以那些键旁边就缺少一种物理和心理的“边界”。

具体一点吧，如果是普通的键盘，你用右手食指按 Y 键。如果你手指伸过了头，你会碰到 T 键的右边。这种触觉反馈信息，可以帮助你的食指退回到 Y 键上面，整个过程不需要用眼睛看。可是自然键盘不一样，食指要是按 Y 键伸的太远，就伸到空隙里去了。你不能很有效的靠着 T 键的边沿，“摸回” Y 键本来的位置。

所以呢打字的时候，食指很容易按进那个空挡里面。为了防止不小心敲到那个空隙里面，你的头脑必须给食指多一点的控制力。虽然这个力道不多，但它造成了脑子和肌肉的轻度紧张，久而久之就会积累起来，形成疲劳。

另外，这个键盘某些键的大小和形状，也因为这种“分裂设计”变得奇怪，按起来别扭。由于空格键跨越了两只手，所以必须做成弧形的，拐个弯，按起来感觉有点不平衡，不太平滑。整个键盘有一定的坡度，说是为了“自然”，我却发现为了适应这个坡度，造成了我手部肌肉的紧张。

整个键盘拱起来比较高，而我发现键盘如果离桌面太高，手会处于一种紧张的状态，即使有那个托手的垫板，也不会太舒服。因为我的手肘是放在桌面上的，垫板虽然抬高了手掌，却不能改变从手到肘的别扭角度。

（为什么我的手肘是放在桌面上，而不是像人体工学专家推荐的放在椅子扶手上或者悬空，我已经在之前的一节解释过了。）

还有，这个键盘很大，很宽，又重，所以它不像普通的键盘，可以随意地被挪动到你喜欢的位置。这个键盘一旦放下，它就几乎固定在那里了。你必须去迎合这个键盘的位置，而不能轻松地推动它来迎合你自己的姿势。这种不能移动的设备，强迫你用某种不适合自己的姿势工作，可能是引起疲劳和劳损的原因。

所以虽然微软自然键盘让手处于自然的角度，我却不用这种键盘。使用普通的笔记本电脑键盘，我的手同样可以使“自然角度”，这样防止了产生手腕的畸形角度和过度疲劳。

机械键盘



很多编程或者游戏发烧友，都热衷于机械键盘。这种键盘采用了老式的机械触发装置，有各种“颜色”的机关，可以发出各种不同的咔哒声。很多人觉得键盘发出这种声音很酷，看起来很专业，很古典，所以不惜重金去买机械键盘。

我也试过机械键盘，我曾经买了一个不错的机械键盘，花了两百多美元。我还买的是那种键程比较短的，力道比较小的。可是一到手试了几分钟，我就发现比起 MacBook 的键盘，这种机械键盘明显费力一些。

不要小看了这键盘多出来的那一点点键程和力度，就那么一点点距离和力度，作用到成千上万的击键次数上面，累积下来就等于劳损。一个小时下来，我明显的感觉小臂上的肌肉开始有疲劳和紧张的现象。

久而久之，这种疲劳可能会引起肌腱炎或者 RSI。所以在第一天之后，我果断的停止使用这个键盘，接着就把它退货了。

用了这么多种键盘，最后我发现 MacBook 的键盘（旧版的），键程和力道都恰到好处。打起来很轻松，声音也很小。其实打字最好不好发出“咔哒”的声音，这样对自己和身边的人都更好。

Kinesis 键盘和 DataHand



Kinesis 和 DataHand 是两种形状奇特的人体工学键盘，价格相当的贵。我从来没有买过它们，但我看别人用过这两种键盘。

Kinesis 把按键放在两个凹下去的“井”里面，传说这样让手指尽量伸直，可以防止某些问题的出现。

DataHand 是一种几乎不移动手指的“键盘”。每根手指放在一个凹陷里，旁边有几个可以触发的机关。所以手指几乎不用移动就可以进行输入。

我确实考虑过购买它们，毕竟为了健康，这些价钱算得了什么呢。然而我最后还是放弃了这个想法。为什么呢？因为：

1. 我从来没有因为使用笔记本电脑的键盘而出过问题。用了十几年都没问题，这说明也许键盘并不能导致受伤。
2. 这两种奇特的键盘跟普通的键盘太不一样了，所以肯定需要大量的适应过程。由于笔记本键盘十多年来没有给我带来什么严重后果，我并不确定是否值得花时间和精力去适应这些很不一样的东西。我至今不会五笔输入法，也是一样的原因，因为拼音输入法已经很不错了 :p

3. 一旦熟悉了这些键盘，我恐怕没法再用其他人的电脑。走到哪里我都得背着这键盘，这给生活增加了很多复杂性。
4. 我不是很确信这些键盘的“人体工学”是否经过了科学实验的检验。一个键盘是否引起手部受伤，是需要很长时间的使用才知道。也许这些造型奇怪的键盘非但不能避免受伤，反而会导致受伤呢？我没有足够的证据可以证明它们是符合“人体工学”。如果真有人因为它们而受伤，我是不会知道的。这些痛苦的人没有话语权，你不会听到他们的悲惨故事。
5. 向我推荐这些键盘的人，几乎都显示出“发烧友”的心理。这些人跟我的工作方式和态度非常的不一样，在学术研究中也显示出盲从的心理。所以我不是很相信这些键盘真的能带来什么好处。

因为这些原因，我继续使用 MacBook 自带的键盘。一点问题没有，我为什么要没事找事呢？

人体工学鼠标

接下来谈谈鼠标。我已经很多年不用鼠标了，因为我发现笔记本电脑自带的触摸板更加舒服，而且准确性和工作效率比鼠标还高。不过我以前还是试过好些“人体工学鼠标”，最后把它们全部放弃了。现在我来说说，为什么我不使用鼠标，特别是不喜欢所谓“人体工学鼠标”。



所谓人体工学鼠标，总是喜欢做成各种奇怪的弧线形状，就像上图里面的这些。你看到它们形状特别，曲线似乎符合手的形状，就以为设计者肯定懂得“人体工学”，所以用起来应该舒服，不会引起问题。可是等你把它买来，往往发现还不如普通的鼠标。



人体工学鼠标一个很常见的问题，就是把鼠标做成一种下面宽，上面窄的流线型。那弧线，让你握着好像很符合手的曲线，有什么不好呢？问题在于，由于它下面宽，上面窄，然后抓握的部位又很光滑，所以你很难把鼠标从桌面上拿起来。

用鼠标的人都知道，你不能一直在桌上推动鼠标。很多时候你需要把鼠标拿起来，离开桌面，挪动一定的距离。这样桌面上朝某个方向才会有足够的“跑道”，让鼠标指针可以移动较长的距离。现在这些鼠标的造型让它们很难被拿起来，所以你的麻烦就来了。你需要用比普通鼠标大很多的力，才能让它们离开桌面。显然这是很累人，非常不“人体工学”的。

人体工学鼠标设计的另外一个问题，他们似乎很喜欢在鼠标上安放几个“特殊功能按钮”，而这些按钮又放在你拿鼠标时很容易碰到的地方。所以你经常不经意的碰到这些按钮，导致意想不到的事情发生。即使你在操作系统里关掉这些无关紧要的功能，这些按钮仍然会在碰到之后发出讨厌的咔嚓声，让你神经紧张会不会发生什么后果。

所以我发现，最好的鼠标应该只有最多三个按钮：左键，右键，中间一个滚轮。



最近还流行“垂直鼠标”，也就是你握住它的时候，手是侧起来的，而不是手掌向下。传说这样侧着，才是手的“自然位置”，所以很多人相信它能保护手。可是等你买来一试，就发现一个严重的问题。因为要把手侧立起来，这种鼠标必须立在拇指和其它四个指头之间。所以如果你要把手拿到键盘那去打字，就发现被鼠标挡住了，得翻过一座山才过得去。而且这种鼠标也有不容易从桌面拿起来的问题，还更加严重。你使劲一抓想把它拿起来挪一下，就会不小心按到鼠标的按键。所以我真是不明白为什么有人花大价钱去买它。



另外我也不用苹果的鼠标。这鼠标看起来好看，有“设计感”，但用起来就很不舒服。最严重的问题是，这个鼠标太扁了，所以鼠标上部不能支撑到手心。而且侧面只有一条棱给你握住，连个“面”都没有。所以这个鼠标拿起来很费劲，不舒服。由于它的“无按钮”设计，按起来感觉也不稳定，到底按没按到，反馈也不好。

所以很多年前，在试过多种所谓“人体工学鼠标”之后，我选择了最简单，最便宜的鼠标。它下面比上面稍微窄一点，或者中间有一个凹槽，侧面有胶皮，所以我可以很容易的把它从桌面上拿起来。它只有两个按钮，一个滚轮，侧面没有多余的按钮，所以我不会不小心按到那些花俏的按钮。

另外，鼠标的按键需要的力道一定要轻。不要小看了这个力道，太硬的鼠标按钮，按久了之后你的食指会很痛，痛得没法使用鼠标，没法拿东西！我的食指痛过，过了好久才恢复。所以我再也不敢使用按键很重的鼠标。

举个例子，像这样的鼠标就还可以：



再后来，我发现笔记本电脑自带的触摸板其实才是最舒服，最高效的鼠标，所以来我就再也没有用过鼠标。

苹果蓝牙触摸板



虽然说我觉得 MacBook 自带的触摸板是最舒服的，同样是苹果的产品，我却不用苹果的外置蓝牙触摸板。实际上我买过一个，两天之后就把它退了。为什么呢？因为它其实不如 MacBook 自带的那个。

跟鼠标一样的问题，这个触摸板必须放在键盘侧面，而不像 MacBook 的触摸板，被固定在键盘的正下方。所以你要

用这个外置触摸板，必须把手从键盘挪开，这样手就得左右挪来挪去的。这比 MacBook 键盘和触摸板之间的距离要大很多，而且不顺手。

然后最严重的问题，就是这个触摸板太大了，以至于手一旦处于它的上空，就没有地方可以暂时歇一下。MacBook 的触摸板没有这么大，而且它旁边的机壳基本跟它一个高度，所以触摸板的旁边有挺多可以靠手的地方。可是这个外接触摸板如此之大，放在身体靠右的位置，而且它的平面比桌面要高，所以你的手在旁边是找不到舒服的支撑点的。

跟鼠标不一样，如果手需要停留在触摸板的上空，休息或者思考，你的手指必须抬起来足够高，否则就会碰到触摸板，导致指针移动或者点击。再加上这触摸板前面是抬高的，所以手指的抬高角度又更大了。抬起手指，是需要小臂上方的肌肉用力的。而那块肌肉，正好是 RSI 经常会发病的那块肌肉。这种把触摸板前面抬高的作法，跟很多人用键盘喜欢把前面支起来的做法类似，我觉得都是错误的。

所以使用这个触摸板没多久，我就发现小臂肌肉开始有紧张状况。然后我就注意到原因是我停留在触摸板上方时，肌肉和肌腱必须用力抬起手指。回忆起一个同学得了 RSI，每天都得在那块肌肉上带上夹板，我果断的停止了使用这个触摸板，回到 MacBook 的触摸板和键盘。

购买这种触摸板的人，一般是因为工作的时候使用外接显示器，外接键盘，所以他们也必须使用外接鼠标，外接触摸板。由于我不用鼠标，而外接触摸板又有这个问题，所以我也停止了使用外接键盘。使用 MacBook 自己的键盘和触摸板，位置和大小都正好合适。

轨迹球



在学校的时候，有个同学强烈向我推荐轨迹球，所以我也买了一个来试试。推荐轨迹球的同学，同时也炫耀了下自己的 DataHand 键盘。轨迹球比起鼠标有一个优点，那就是轨迹球放在桌面上不需要移动，你不需要把它拿起来。可是拿到轨迹球之后没几天，我就把它退掉了。它有什么问题呢？

你有没有发现，轨迹球就是一个翻转过来的老式鼠标？很久以前，鼠标的下面不是光学的，而是一个可以滚动的球。移动鼠标的时候，球就开始滚动，带动几个滚轴，软件就根据滚轴的转动算出光标的位置。

滚球的鼠标有个问题，那就是用过一段时间之后，球就会脏，脏东西跑到轴上去，后来就不利索了，所以你得定期清理这个球。如果鼠标做得不够精密，球或者滚轴就会变形，无论你怎么清理，它都不会再顺畅。

鼠标的球只需要接触桌面，还算相对干净的环境。一个翻转过来的鼠标，问题就更严重了。你的手可能刚吃过薯片…… 所以脏东西更容易粘到球上。本来把这样大一个球做得够圆滑，能够自如的滚动，就已经不容易了。再加上手上的脏东西，就更加困难。

这也许就是为什么我买到的第一个轨迹球，滚起来就不是很顺畅，总感觉有轻微的卡顿。也许你说，再多花点钱，买更好的轨迹球不就行了。实话说，我买的轨迹球已经够贵了。但是你发现没有，这不是钱或者做工的问题。要做出一个顺滑的轨迹球，比做出一个顺滑的鼠标，要困难很多。所以贵的也不一定就能做好，贵的也不一定能一直好用。做出一个好用的轨迹球，这个问题是不必要的困难。

轨迹球还有个问题，就是那个球一般都顶出来一定的高度，所以你的手放上去，就成了一种前面抬高的，不健康的角度。轨迹球一般都比较大，旁边的座子上就是按钮，所以你的手在旁边不容易找到可以歇一下的位置。所以轨迹球有跟苹果蓝牙触摸板类似的问题：找不到歇手的地方。

鼠标从滚球鼠标进化到了光学鼠标，轨迹球却停留在滚球的时代。要是轨迹球进化到“光学时代”，它会变成什么样呢？稍微一想你就会发现，它会进化成触摸板！触摸板的操作方式，难道不是跟轨迹球一模一样吗？只不过那个滚动的球，变成了一个能够感知电容的平板。

所以触摸板兼具了光学鼠标和轨迹球的优点，这也许就是为什么触摸板成为了我最终的选择。

ThinkPad 的小红点



很多人都认为 ThinkPad 的小红点（也就是键盘中间那个红色“摇杆”，可当鼠标用），是个好东西。确实，它比起触摸板有某种优势。因为在键盘的中间，所以你的手不需要离开键盘就能移动光标。

我也曾经喜欢用小红点，我觉得它效率很高。因为用小红点上了瘾，我很长一段时间都没用下面的触摸板。直到有一天，我的食指开始痛……我无法再使用小红点，甚至没法用食指拿东西。于是我明白了，长期大量的使用小红点会引起手指受伤。还好，过了好几个月，手指就恢复了健康。于是我再也不敢用小红点，而转向了触摸板。

使用了十多年的触摸板，我发现小红点引起的伤害，似乎没有办法在触摸板上重现。因为使用触摸板的时候，我的手指是几乎完全不用力按的。我把触摸板全都调成了“轻触点击”模式，也就是轻轻点它一下，就作为鼠标点击。所以我一般不把 MacBook 的触摸板按下去，也不用 ThinkPad 触摸板的按钮。所以我的手与触摸板之间只有非常小的力学接触，它不可能造成小红点的那种伤害。

我并不是说完全不可以使用小红点，但你要知道它对于手指尖的细胞组织是有一定的压力的。所以用小红点的时候不要太用力，不要太着急。经常按得太重都是因为你想让光标快速的移动，所以得使劲推小红点，久而久之就会受伤。

总结

到这里，我已经总结了一些自己对于人体工学的经验。这些是经过十几年的使用电脑总结出来的，跟常见的“人体工学”所宣扬的作法恐怕很不一样。我希望它们对人有所帮助。

（写这样一篇文章花费了我很多的精力。如果你从中受益，请考虑[付款购买](#)。）

旅行的智慧

每一次旅行都是一场修行。通过每一次旅行，我都会总结出一些简化生活，让它变得更舒适的方法。有了『编程的智慧』和『生活的智慧』，我现在在机场发呆，观察到自己和其他人的一些旅行用品和方式，所以想试试写个『旅行的智慧』。当然有一些绝密的技巧我不方便公开，但还是有很多可以分享的东西，也许可以帮助人们。

不用硬壳旅行箱



这次回国发现硬壳的旅行箱非常流行，似乎占了绝大多数。有些甚至是金属外壳加铆钉材料，就像电影里的那种炸弹箱。下面有四个轮子，开口是从一半的位置而不是最上面。我很纳闷为什么大家用这种箱子。

首先，旅行的时候其实没有什么值钱的物品，需要那么硬的箱子来保护的。大不了把一些东西夹在衣服下面，就差不多起到保护的作用了，撞一下根本伤不到它们。如果你真有很贵又脆弱的物品，那么你完全可以拿一个盒子先把它保护起来，然后放进箱子里。

据我观察，很多这种硬壳箱子新的时候好看，可是一旦被硬物（地板，台阶，火车飞机上的各种设施）划伤，就会留下难看而锋利的划痕或者凹陷。这样碰到自己身上或者其他人都不是很舒服的，摸起来也很不爽。很多硬壳箱子外壳都是光滑的，这样上面粘了灰之后，手一摸灰就到手上了。想一想吧，你需要用手摸着到处是划痕，沾满灰尘的硬壳把它扛起来，放到行李架上……

所以尽管硬壳箱子如此流行，我也只用软质面料的箱子。软面料箱子的好处是，撞了之后不会留下凹陷，也很难留下划痕，粘了灰之后，灰不会转移到手上。碰在自己身上不会伤到自己，摸起来也舒服。软的面料给人一种舒适的感觉，要知道箱子很多时候是会靠你很近的。而且软面料箱子外侧一般都有放小件物品的小袋子，可以放一些随时可能用的东西在里面。

硬箱子的开口一般都在一半的地方，貌似更加容易整理物品，但我以前就用过一个在一半位置开口的箱子，发现其实不能达到那个效果。旅行的时候，东西稍微有些规律塞进去就可以了，再买一个卫生用品小包放牙刷一类的物品，就很容易整理了。没必要从一半的地方开口，这样盖上打开的时候都感觉沉重，而且害怕上面的东西掉下来。

行李箱上的密码锁基本都是废物。三位数字的密码锁，你以为别人需要试1000次才能打开它吗？由于转到正确数字的时候发出的声音不一样，这种密码锁的组合数其实只有30。我可以在半分钟之内打开这种锁。所以买箱子的时候就别拿密码锁当回事了。

下面有四个轮子还是好的，因为飞机和火车中间的过道都比较窄，横着不容易过去。在过道里把箱子提起来很费力，而且如果被前面的人挡住去路你就得又放下来等……

所以我的建议呢？使用下面有四个轮子的软面料箱子。

带自己的牙刷和拖鞋

不要指望酒店会有你满意的牙刷或者拖鞋。任何酒店提供的拖鞋基本都是一次性的那种，穿上去有一种被黏住的感觉。一两天的话很多人都那样就凑合了，可是要是多几天，还是准备自己的拖鞋为好。

我从来不用酒店提供的免费牙刷。其实美国的酒店就从来不提供牙刷和牙膏，因为这种用品如果有问题，有可能威胁到人的健康。美国酒店怕出事你告他们，所以就干脆不免费提供这些东西，我觉得这还是有道理的。牙刷这么小的东西，最好自己带着。

不带 Crocs 塑料鞋出门



说到拖鞋.....由于一些历史原因，我喜欢用 Crocs 大头塑料鞋当拖鞋用。旅行的时候也带着它。但是后来我发现 Crocs 的鞋在箱子里很占空间，而且很不实用。

也许你以为这种塑料鞋在雨天或者海边沙滩上很舒服，那你就错了。原因是因为这种鞋子前面不是完全开口的，一旦有沙子或者小石子进了这鞋就很难出去。所以那些沙子和小石子就会留在鞋子里折磨你的脚。

这次出门我又把跟随我多年的 Crocs 鞋放在箱子里，占了一大块地方。当意识到它一直以来的无能之后，我在地摊上买了一双轻巧的草编拖鞋，然后 Crocs 鞋就被我扔在酒店的垃圾桶里面了 :P

提前多一点去机场

我发现很多人把去机场的时间算的太精确，比如提前两个小时动身，结果路上遇到堵车，或者到了机场发现安检的队很长，搞得神经紧张跌跌撞撞的。

我作法是避免买早上的机票，买下午的机票，提前多一点去机场，到了机场还会有一个小时休息时间的样子。因为机场一般有比较好的设施，像咖啡店一类的，所以早点去坐在那里发呆，跟在城里发呆，区别其实不大。

这样做的好处是你的神经一直处于放松的状态，从容不迫的。

避免登机口排队



我总也不明白，还没有广播叫大家开始登机呢，为什么总有很多人喜欢在登机口排好队，等着上飞机。一架飞机上的人，不管你先上还是后上，都是大家一起起飞，一起着陆。你以为排在前面就可以早点到达目的地吗？据我唯一一次试验，这种队可以排你 40 分钟的样子，那次我觉得自己真是疯了傻了。

所以聪明人都是坐在登机口旁边座位上或者咖啡店里。等到广播说开始登机了，才慢悠悠的走到登机口。一般说来，因为前面的人要在机舱过道上停下来放行李，所以这个长队要一二十分钟才能上完。你如果听到广播就走过去，就得站一二十分钟。所以就算听到广播，你仍然可以继续坐着，看到队变得很短了才走过去，就直接可以上飞机了。

需要注意的一点，有些地方的机场很狭长，所以你最好一开头就找准登机口的位置，坐在附近。免得到时候通知登机了，发现还有十分钟的路要走！

别急着下飞机

绝大部分人会在飞机停靠之后，立即走出座位，拿下行李，站在那里等着下飞机。其实飞机停稳之后，还是需要几分钟时间，等通道对接好，还有一些不知道什么手续，才会打开舱门的。所以你早早出去了还是只有站在过道里，跟其

他人挤在一起。

所以聪明人都是坐在位置上，看到队伍开始动了，才走出来准备下飞机。一般礼节是机舱后面乘客应该让前面的乘客先下飞机，所以你随时有权利站到过道上，挡住后面的人，取行李，然后下飞机，而不会有妨碍后面乘客的礼貌问题。如果你的行李放在了座位后面的行李架上，前进的队伍让你没法拿到行李，那你可以等所有人都下了再下，也不会迟多少。

在路上招出租车

我发现很多人爱用滴滴叫车，不管眼前有没有空的出租车，都用滴滴叫“快车”。其实这种完全靠滴滴的做法并不是最好的。

用滴滴之类的软件叫车，有一种“找人”的开销。司机经常不知道你在哪里，或者知道你在某个路口，却是一个很大的路口，不知道你在哪一个角上。解释老半天终于找到，才能上车，这个过程经常需要好几分钟甚至十分钟以上。

其实招路边来的出租车要高效准确很多，因为你看到那个车了，你招手，它停下来你就直接坐上去了。就没有了这找人的麻烦事，整个过程只需要几秒钟时间。

坐出租车后座

我发现国内仍然有人（虽然已经不多了）喜欢坐出租车或者滴滴车的前排座位，坐在司机旁边。这似乎来源于一个古老的意识，认为前排座比后排的更舒服，或者更安全。可能因为很早的时候汽车都不大好，所以前排的座位确实更舒服，而后面的座位可能很颠，容易晕车。所以那种古老的观念，一直流传到现在，即使现在的汽车前后座位已经一样舒服。

前排座的问题是司机就在你旁边，你必须跟他分享前面的空间。你的一举一动，司机都能看见，包括你手机上发的信息。如果遇到脾气或者态度不好的司机，你会更容易受到他的影响。有些城市的出租车为了保护司机，会在司机座位旁边装一个有机玻璃的隔离板。加上这个板之后，右边的座位就没有多大的地方了，而且由于那个玻璃挡在那里，感觉很不舒服。

其实出租车最好的座位是后排。不管是一个人还是跟朋友一起，我都会坐后排。如果是两个人一起坐，我们两个都会坐后排，而把前面座位空着。如果是三个人，那么前排不得不坐一个人。

这里有一个礼节问题。如果是坐朋友开的车，那么他右边的座位必须有一个人陪。因为朋友不是服务人员，所以你不可以让他孤零零的在前面开车，好像你的司机一样。反过来，出租车司机确实就是为你服务的，就像餐厅的服务员，所以你坐后面，把前排座位留空，并不是不礼貌的行为，而是理所应当的。

（未完待续。。。）

生活的智慧

我曾经写过一篇文章，叫『[编程的智慧](#)』。再加上我总写一些技术性的文章，很多人可能还以为我是个技术狂人，只知道谈论技术，不会生活。其实跟我接触紧密的人都知道，我是一个很会生活的人。

我花了很多心思来研究各种物品，它们的用法和组织方式。我用一种理性而智慧的方式对待生活中的方方面面，我的“生活智慧”并不亚于我的“编程智慧”。实际上这两种智慧的起源是相同的，它们都来自于一个人的内心，来自于他的本性。

一个生活邋遢，不会收拾和安排，匆匆忙忙，不会花钱的人，他写出来的代码大概也差不多蹩脚。反之，如果一个人生活里很有条理，他编程的时候肯定也无法原谅自己写出混乱复杂的代码。这种人还有一个特点，他们能把钱花在最恰当的地方，而不是一味的省钱，给自己造成不方便。

实际上如果你开始琢磨生活里的智慧，就会发现很多编程的算法，几乎可以直接对应到生活里面。反之亦然，生活里的琐碎智慧，也可以启发编程的想法。

整理房间

说到算法在生活里的应用，我首先想到的就是整理房间。你如何组织和摆放房间里的各种物品，使得它们美观温馨，不占太多地方，可以方便的拿到，用完之后又可以方便的放回去？你可能想不到，物品的管理，跟一种常见的编程概念有关系，那就是[哈夫曼编码](#)。

整理房间这看似简单的问题，耗费了很多人一辈子的时间还没琢磨清楚。很多人的房子虽然还算干净，东西却是任意摆放的，缺乏规律和组织，而且他们舍不得花钱买恰当的工具，于是他们就遇到这些问题：

- 东西要用的时候就找不到了。
- 很多东西摆在蹩脚的地方，你每次用它都得先挪开另外一些东西。用完放回去又得再次挪开那些东西。
- 家里东西太多，空间不够摆放了，却全都舍不得扔。
- 家里的储物空间其实挺多，柜子抽屉很多，却由于没有合理的组织，所以放不下很多东西。
- 有些物品由于大小和形状特殊，不知道放在哪里好，放在哪里似乎都别扭。

很多人每天都遇到这样的问题，却因为是“生活中的小事”，一直没有认真思考过，甚至认为自己是大忙人，赚大钱的，所以不屑于解决这种问题。而我则不一样，每当遇到这种问题，我都会像解决编程问题一样，认真而理性的思考，所以很多时候我会想出很合理的解决方案。

对于物品的管理，我有一种通用的“智慧”，类似于哈夫曼编码。在哈夫曼编码里面，最常用的字符用最短的编码来表示，不常用的字符用长一点的编码来表示。这种概念应用到生活里面，那就是把最常用的物品摆在最容易拿到的地方，把不常用的物品摆在不容易拿到的地方。

举个例子，牙刷牙膏是每天都用的，所以我就把它们直接摆在浴室的案台上面。浴盐不是每天都用，我就把它放在下面的柜子里。越是不常用的东西，它在柜子里的位置也越靠里面。

购置物品，抛弃物品

我发现那种家里东西太多，似乎没有空间存放的人，他们并没有仔细的思考过自己“需要”什么。他们舍不得扔掉旧的东西，即使自己永远也不会再用它们。他们会跟自己说：“这些东西当年可是花了多少多少钱买来的呀。舍不得扔！”或者说：“虽然这个东西不值钱，可是勤俭节约是我们的光荣传统，积少成多。这也扔那也扔，买房子的钱从哪里来呀！”

从编程和设计的角度来看这种人，这种人就是没有明白所谓“极简主义”，而且他们并没有对物品的价值做过理性的计算和分析，就盲目的崇尚所谓“节约”。

做技术和设计的人也许都看过乔布斯设计房间的故事，还有那副标志性的照片。乔布斯坐在房间中间，旁边只有一盏灯，其它什么东西都没有…… 你也许没有想到，每一次搬家到遥远的地方，我也是这样开始生活的。我会从自己的物品里面挑选出自己最想留下来的，而把其它的都送人，卖掉或者扔掉。

到了新的住处，我会购置一盏灯，一张气垫床，这样让我晚上看得见，在一个星期之内有睡的地方。然后，我会很快去商店挑选一张非常舒服的床，不惜血本把它买下来。床垫，床单，枕头之类的贴身物品，直接关系到人的休息和健康，所以对于它们我会很舍得花钱。你也许不知道，不同质量的这些东西，感觉真的相差很多。

然后我会开始细心思考自己需要其它什么东西。我不会到商场里看到什么就买，每购置一件新的用品，我都会问自己这个问题：“我需要这个物品吗？它是否跟已有的物品的功能有冲突？它比起同样功能的物品有什么优势吗？它质量更好，更方便，更美观，更温馨？它会不会带来什么麻烦？会不会占用太多空间？它带来的价值相对于这些弊端，值得吗？”

我会权衡所有这些需求。如果我还没有类似功能的物品，那么我会上网搜索类似作用的东西，看看有没有更好的，然

后根据自己的判断力和其他人的评价，从中挑选一个最好的。我经常使用美国商店和 Amazon 的无条件退货功能。对同样功能的物品，我有时候会买好几个不同的过来，经过几天的试用，留下其中最好的一个，把其它的都退了。有时候有的东西过了退货期或者退不掉了，就送人或者扔掉。

这样看上去似乎很残酷，很浪费，然而一旦我选好了一个物品，我以后就会一直记得它，我会记得我当初为什么选择了它，我会记得其它相似的产品有什么缺点。下一次再买类似的东西，我就会利用这些积累的智慧，只跟以前没见过的物品比较，而不会重复同样的过程。这样就不会给商家造成过多麻烦和过多的浪费。

由于这种极简主义的方法，我的房子里很少有多余的物品。

什么是浪费

很多人盲目的反对“浪费”，认为我们应该勤俭节约，扔了本来还可以用，或者可以吃的东西就是浪费。所以这种人买了东西，总是会一直把它们留着，自己做了菜或者在饭店点了菜，总是要把它吃完。他们没有理性的意识到，使用不好的劣质的产品，让自己的身体或者心理受损，吃过食物损害自己的身体，影响自己的睡眠，那才是真正的浪费。

使用高质量的，安全的产品，扔掉不好的产品，不吃过多的食物，表面上浪费了钱，实际上却是节约了钱。很多人，特别是老一辈的中国人，远远地低估了自己身体和健康的价值。年轻一点的时候拼命地省钱，克扣自己的身体。由于舍不得扔掉旧东西换新的，给自己的生活造成各种不方便，不愉快。

年纪大一点的时候，发现身体有点不好了，很多地方开始出问题，又因为想省钱而不去找好的医生，因为省钱而不使用最好的医疗产品和设备，所以问题不能在早期得到防治。到后来，问题就越来越严重，最后到了不可收拾的地步。最后，他们把以前“节省”下来所有的钱都赔进去了，甚至还要多几倍，却还是难以很好的保全自己的身体……

所以关于浪费这个事，我觉得每个人都应该做一些理性的分析，把账算清楚。你现在花掉几千块钱解决好生活上的一个问题，也许会在将来帮你节省几十万也说不定。有些人几毛钱都斤斤计较，你计较个几十年，总共加起来能节省多少？

有些人几块钱一大卷的专用垃圾袋都舍不得买，用超市购物留下来的袋子装垃圾。这种袋子经常是拎东西的时候被物品穿了孔的，所以他们就发现厨房的垃圾漏出来，臭气熏天，难以打扫干净。理解到这个道理，我从来就不用超市的购物袋来装垃圾。专用的垃圾袋很靠谱不会漏，价格微乎其微，我省那点钱干什么？

理性一点的人都会“优化”节约这件事，这就是算法的思想。我的技巧是优先在贵的东西上面节约，而在不那么贵的东西上面就放松一些。比如我在车子上花几天工夫研究，买一个好用又不贵，不容易出毛病的，一下子比别人多省下来几万块钱。然后我就可以在贴身用品上面多花钱，买最安全最舒服的。我就可以不吝惜钱去找好的医生，用最先进的医疗设施，保持身体的最佳状态。虽然为此花了好几千，但比起买车省下来的几万，只是一个零头而已。

相比之下，我认识一些人开着保时捷，自己家里的用品质量和安排方式却档次很低，该有的东西没有，自己的孩子病了还舍不得钱看医生。省下几乎所有其它开支去买保时捷，我也是醉了 :P

价格和价值

还有一项智慧，那就是物品的好坏很多时候不是价钱决定的。很多人以为最贵的东西一定是最好的，那是因为他们根本不会识货，所以就让价格来帮他们选择。经过许许多多的比较，我发现好的商品往往都不是最贵的，当然也不排除它们确实就是最贵的。

举一个例子，很多人都以为 Beats 的耳机是最好的，因为它卖的很贵。可是经过研究网上的评价，自己的试听，我发现这种流行的大牌子，其实都不咋滴。实际上 Beats 耳机的音色属于最差的之一，声音特别模糊特别钝，重音太重，以至于你没法听清楚音乐的旋律。

后来我发现，有一种耳机叫做“专业录音棚监听耳机”。这种耳机由于要给专业录音师使用，所以很重视对音乐原封不动的还原，而不是盲目的加强低音，让你感觉很有节奏感。于是我发现了 Audio-Technica ATH-M50，还有 AKG K-712。这两款耳机价格都不是特别贵，音质却跟 Beats 最贵的型号都天上地下，舒适程度也好很多。后来我发现凡是识货的人，几乎人手一个 ATH-M50。

如果你想要蓝牙耳机，可以试试 AKG Y50BT。

需求和品质

之前说到我买了两个音质超级好的耳机，然而后来从“需求”的角度出发，我发现其实我是不需要这种头戴耳机的。一来它们比较大，不方便携带，二来它们罩在耳朵上，还是容易引起耳朵感觉热。

我需要那么高的音质吗？其实有些耳塞的音质已经能满足我的需求，小巧又不会闷着耳朵。所以虽然我花了几百美元买了两个大耳机，平时在办公室用得最多的，却是轻便又便宜的耳塞。ATH-M50 放在办公桌上，只用过一两次。后来这两个大耳机就被我送人了 :P

我最近很喜欢用的耳塞是 1More ibfree，是非常轻便的蓝牙耳塞。

这个故事告诉我，每个人的需求都不一样，你必须从自己的需求出发，而不能盲目的追求某种最好的品质，比如音质。耳机对于我的价值，不但在于音质，还在于它是否便于携带，耳朵是否舒适。在这么多方面权衡之后，我发现我自己其实可以稍微降低对音质的“发烧”追求。

价格不贵的好产品的例子

同样的原理，我发现了好些价格不贵，质量却比贵的还好的产品。比如：

- 家里的充电吸尘器，我选择了 Hoover，而不是贼贵，吸力还不怎么好的 Dyson。
- 机器人吸尘器，我选择了一个老版本的 Neato，而不是最贵的 iRobot，也没有买最贵型号的 Neato，可以远程 WiFi 遥控的那种。在国内的时候，我用小米的扫地机器人，我发现小米的机器人比 Neato 的还聪明，能远程控制，还便宜。
- 电视机，我随便买了个非 4K 的 55 寸 Sony，而没有买昂贵的 4K 电视。实话说，看起来很爽，真没感觉少了什么。4K 电视完全没必要，片源稀少，而且真有人在乎那么高的清晰度吗？
- Amazon Basics 的苹果充电线，比苹果原厂的还结实耐用，摸起来感觉也好一些（塑料而不是橡胶感觉），价格却便宜很多。所以我买了好多根，每个房间两三根，车上两根，办公室一根，背包里还带一根跟充电宝一起用。这样就不用把一根充电线到处带了。
- 汽车，我买了个本田雅阁 V6，而不是奔驰宝马之类的。实话说，这个本田就是比我女票的奔驰好开。引擎代码被我 hack 了之后就更好开了 :)

功能覆盖不重复

从需求出发，我精挑细选了很多的物品，它们的功能处于一种“覆盖而不重复”的状态。覆盖是说我的每一种需求，都有物品来满足它，不重复是说同一种需求没有重复的物品。这个不重复的特点，使得我在做一件事的时候，不需要每次都从不同的物品里面挑选。

举一个例子，我之前有很多的手机充电器。每买一个苹果的产品就有一个充电器，iPhone 充电器，iPad 充电器，…… 然后你就发现 iPhone 自带的充电器速度很慢，因为它的输出电流只有 1A。iPad 充电器输出有 2.2A，却只有一个输出端口。后来我从 Amazon 买了一些 Anker 的快速充电器，有好几个输出端口那种，每个输出都可以动态变化，最高可以超过 2A，可以用于所有的 USB 设备充电。

我在每个房间里都放了一个这种充电器，给它们配上足够多的充电线。这样我就可以在任何房间充电，而不用把手机放在固定的地方。这下子我就可以把 iPhone 的充电器都扔掉了，因为理智告诉我，有了 Anker 的充电器，我再也不会想用 iPhone 自带的充电器了。至于 iPad 的充电器，我拿到公司去用。因为在公司我只需要给一个设备充电，即我的手机，所以 iPad 的充电器正好满足这个需求，还很快。

同样的道理，每买一个 iPhone 我都会直接把它的耳机扔掉，因为它远远不如我的 Symphonized 耳塞，而且很容易从耳朵里滑落出来，基本没法用。

由于这种功能不重复的删除法，我的物品数量一直处于合理可控制的范围。还有一个好处就是要做一件事的时候我不需要进行选择，因为能做这件事的产品只有一种。

对于家具和衣物也是差不多的原理。如果你发现有东西两年都没有用过，那很有可能你永远都不会再用它。由于你不再需要它，不管你花了多少钱买来的，你都可以把它扔掉，送人，或者二手卖掉了。

为多个位置购置同样的物品

有些物品是如此的常用，以至于你在家也需要它，在公司（学校）也需要它。这样的物品包括耳机，手机充电线，电脑充电器，茶壶，等等。如果这样的物品你只买了一个，你却需要在两个地方待（比如家里，公司里），那么你很可能需要把它两边带。

有些物品每次带走其实都需要花一点工夫，比如充电线需要拔下来折叠好，耳机需要把线收好，电脑充电器需要拔下来，收线。到了目的地，你又得把这些东西拿出来，把缠起来的线整理开。有些东西，比如电脑充电器，其实有一定的重量，背来背去的加重身体的负担，在大热天更加不舒服。还有茶壶，这东西每天都带来带去，那就很麻烦了。

如果你发现一个东西每次都需要带着走，那么你就可以考虑再买一个一模一样的物品，把它们分别放置在多个位置，这样你就不需要把它到处带了。

这就是为什么我的耳机，手机充电线，茶壶，电脑充电器之类的东西，总喜欢买两个。一个放在家里，一个放在公司。有些东西，比如充电宝和手机线，我还会在车上购置一个。因为很多时候开车出去玩都会忘了带充电宝和充电线，下车出去玩就开始担心手机没电，特别是在某些荒郊野外爬山的地方。所以在车上常备一套这种东西，出去的时候就总是有备用的。

我工作过的某公司曾经来了一个实习生，他经常跟我借手机充电线，理由是：“我的线今天又忘了带，忘在家里了。”借人家东西一两次还好，后来他就开始天天借我的，因为我有一根线总是留在公司的。这个实习生就是没有明白我这里说的道理。一根充电线几块钱都懒得买，天天借人家的，给自己和他人都造成不方便。

收纳物品

精挑细选了物品，然后你还需要考虑如何整理和放置它们，这就叫做所谓“收纳”。收纳东西其实有很多的智慧，跟编程里面的概念非常类似。不会收纳物品的人，往往代码也会写的很乱，不容易理解，因为他们同样不明白代码应该如何“放置”。

有些人除了室内装好的柜子，抽屉，架子，从来不自己购置收纳产品，所以他们总是发现东西收拾不好，不知道该放哪里，或者放进去就找不到了。

这其实就是我在『编程的智慧』一文里面指出的问题，如果你的函数里面有太多行代码，就会很难理解。如果你把所有物品都放进一个大抽屉，那么这些东西就会杂乱无章，而且会随着抽屉的移动到处乱跑。

所以稍微懂得收纳一点的人，都会买一些小盒子或者筐子放在抽屉里，把东西分门别类的放进去，这样它们就不会乱动，而且很容易找到。

同样的原理也适用于柜子。很多人因为有了衣柜，就完全依赖于它里面的隔板。可是衣柜里的空间往往是很大块的，所以有时候你只放一点东西进去，它就会在逻辑上占据整个空间。你为了分门别类，只好把其它东西放进其它隔间。后来你就发现隔间不够用了，后来你就发现自己开始到处乱塞东西，然后就完全乱了。还有人喜欢把很多衣服叠在一起，放进衣柜的隔间里。但是这隔间挺大，所以衣服垒起来很高。这种垒起来的衣物是很容易像山崩一样垮下来的，造成很多的麻烦。

聪明一点的人都会使用一种好东西，叫做筐子。筐子是如此的有用，以至于我总是在考虑某个地方是不是该有一个筐子。我的衣橱里有各种大小的筐子，它们分门别类的容纳我的衣物：袜子，内裤，T恤，毛巾，……还有一个脏衣服筐，一个小的脏袜子筐。还有一个筐子里面铺上一张很舒服的毛巾，给我的猫睡觉用，这样他就不会睡我的其他筐子，搞得全是猫毛:P



我的客厅里和厨房里也有各种筐子。分门别类的放置各种物品：书籍，遥控器，背包，食物，调料，…… 我还特别在出门的地方放几个筐子，这样我出门需要的东西都可以放在那里。比如上班用的电脑包，雨伞，帽子，钥匙等等。

另外，不但需要考虑大小，我还会考虑这些筐子的温馨程度。显然我不会用塑料盒子来装所有这些东西。对于卧室衣橱里的筐子，一般都是藤条编制，里面衬有印花美观的布料。客厅里也有很多这种筐子。大门口放背包的筐子是一个粗糙的藤条筐，没有布在里面，因为那些东西可能有一点脏，有时候在外面会放在地上的。

电线也是一个讨厌的东西，有些人的电线很长，就让它到处乱跑。其实好的办法是买一些尼龙搭扣的电线收纳带子，就像 ThinkPad 充电器上面带的那种，把过长的电线都折叠起来绑好。这样你的电视柜附近才会井然有序，容易打扫。

容器过剩

说到收纳物品，有可能走向另外一个极端，那就是家里的空容器太多。有人喜欢把物品的包装盒，食物的瓶子之类都留着，心想以后可以用作收纳容器。这种“以后用于收纳”的想法，就是很多人家里的储藏空间不够用的原因，因为这些空盒子，空瓶子占用了太多空间，而它们几乎永远是空着的，你永远也不会用它们装任何东西！

还有些人不丢盒子，原因是为了准备以后搬家的时候，可以有一个合适的盒子来保护那件物品，特别是一些电器，比如电饭煲，电视机之类的。所以他们的储物空间里堆满了各种电器的包装盒，以及里面的泡沫。我也曾经这样做，我买了一个很好的电饭煲，所以我希望搬家的时候它得到很好的保护。

可是后来我就发现家里的储物空间被这些盒子塞满了，于是我开始理性的思考这个问题。搬家是一件经常做的事情吗？不是。生活才是经常做的事情。我为了一件偶尔才做一次的事，损失了我每天都需要的储物空间，让我的生活变得蹩脚，不舒服。

而且电饭煲之类的东西，不管是多好的，花了多少钱买的，搬家时真的需要很好的保护吗？不是的。这些电器的包装盒之所以如此严密地保护它们，是为了防止商家在长途大量运输它们的时候撞坏，以至于无法销售。然而普通人搬家，其实是一件从容不迫，小心轻放，不大可能撞坏东西的事情。后来我发现，像电饭煲一类的东西，最多拿个厨房毛巾裹一下，跟其它厨具餐具一起放进一个大箱子里，就能在搬家时达到很好的保护效果了，根本不需要它原来的包装盒。

很多人留着包装盒是为了防止需要退货的情况，我也有这种需求。可是很多人把盒子放起来，然后就忘了它们在那里占据着空间，不知不觉中空间越来越少。我一般把包装盒保留几天，我会把它们放在很显眼的地方，比如厨房的入口处，这样我每次进入厨房都会看到它，这样我就不会忘记扔掉它。过了几天之后，我确信我喜欢这东西，不想退货，我就把这盒子及时扔掉。这样一来，我家里的储物空间全都装着有用的物品，而不是空的容器。

购置很多一模一样的袜子

我从别人那里学过来的一个小窍门，那就是买很多一模一样的袜子。以前我的袜子都是这里买一双那里买一双，到了后来它们的颜色，样式，长度都不一样。每次洗了袜子你都需要给它们配对，卷起来收好。

但是如果你买很多一模一样的袜子，每天穿一双就扔进一个专门放“穿过袜子”的筐子。由于每双袜子只穿一次，它们就不会臭。然后你就可以两个星期甚至一个月才洗一次袜子。洗了袜子之后你不需要把它们配对折叠，直接拿出来扔进一个“干净袜子”的筐子就行了：）

对于内裤和T恤，我也有类似的办法。每次洗了衣服，折那么多T恤都是一件讨厌的事情。我的T恤是不会皱的，所以后来我发现干脆不要叠T恤。买一个专门放T恤的筐子，洗了之后就把它们扔到里面，要用的时候抓出来就可以了。

由于使用了筐子来收纳这些小衣物，虽然筐子里面乱乱的，可是整个房间看起来却井井有条。

对清洁用品的研究

另外，清洁用品和清洁方式也是需要讲究的。很多人可能以为像我这样的“大师”肯定不屑于这种琐事，而其实我对家居清洁的研究，可以跟专业的清洁大妈相比。实际上，我曾经向专业的清洁大妈请教过，学会了挺多技巧:P 下面我就稍微提点一下。因为清洁窍门内容太多，都可以单独写成一本书了，我这里肯定不可能面面俱到。

首先我想指出一种中国人很常见的误区，那就是他们的家里分“脏的地方”和“干净的地方”。很多人认为卫生间是脏的，湿的，马桶是脏的，阳台是脏的，而卧室是干净的。然后他们就把问题复杂化了，从客厅进卧室需要换鞋，从浴室出来需要换鞋，上阳台需要换鞋。他们在浴室里洗拖把，所以浴室里就有好几个盆子，分别有标签：洗脸的，洗内衣的，洗袜子的，给我洗脚的，洗拖把的，给老爸洗臭脚的…… 这么多盆子放在哪，是一个严重的问题。

很多家庭的某些物品被认为是“应该有灰”的地方，比如冰箱上面，马桶下面的座子，所以他们把那些地方置之不理。家里约定的规矩是，不准去碰那些地方！或者他们为了“方便打扫”，就盖一层塑料纸，甚至改一些报纸或者超市的促销广告在那些地方。然后你就发现整个屋子里有好几处地方盖着这种垃圾一样的纸，真个屋子就变得不温馨，不舒服了。他们心里想的是要是那些塑料纸粘了灰，就扔掉换了，但实际情况是那塑料纸上一直有一层很厚的灰尘，根本没人管它，跟没有那塑料纸没什么区别。

与其把家里搞得这么复杂，设立各种条款要家人遵守，你还不如想个办法，把所有区域都变得一样干净，这样你在家里只需要一双拖鞋就可以到处走，随地都可以坐，也不怕不小心碰一手的灰，“市容市貌”也好很多。你只需要买一个充电的手持吸尘器，就完全不需要担心这些灰尘的问题。

另外，家里其实根本不应该有拖把这种东西，还需要一个桶或者盆子来洗它。平时合理打扫的房子根本不应该有很多脏东西，所以不需要拖把这样重量级的清洁设备。你应该考虑买一个 Swiffer 之类的地板擦，它用的是一次性的擦布。地板上沾了一点灰尘渣滓，轻轻擦一下就干净，擦布脏了扔掉就是了，根本不需要清洗。

然后我想谈谈灰尘的处理。很多人不管什么东西都喜欢拿湿抹布擦，然后就发现本来漂浮在表面的灰尘，变成泥浆糊在了物体表面，后来就很难清洗干净了。很多人认为湿抹布可以防止灰尘飞扬，对健康好。可是他们没有发现，有更好的作法可以防止灰尘飞扬，却不需要把灰尘变成泥浆。

我很早发现了这个问题，发现真不能用湿抹布擦，后来我就开始用手持吸尘器去吸那些灰。这种吸尘器吸角落缝隙效果还可以，但是对于像茶几一样的有很大面的家具，覆盖就不是很好，总是留下很多灰尘。可是像茶几这样的东西，你又不能拿吸地板的大吸尘器去吸。怎么办呢？

后来，一位清洁大妈教会了我一个窍门，这个窍门就是 Swiffer。Swiffer 是一种地板擦，它配有两种一次性的擦纸，一种是干的，一种是湿的。干的那种擦纸可以吸附灰尘，把它用静电吸起来，而不会把灰尘推着到处跑，飞起来。这位清洁大妈的重大发现就是，Swiffer 的这种擦纸，不但可以安装在地拖上用来擦地，而且可以直接当成抹布拿在手上擦东西。

所以 Swiffer 擦纸就成为了新一代的鸡毛掸子，但跟鸡毛掸子不一样，它不会让灰尘飞起来。后来家具和物品上面的灰尘，基本就是用 Swiffer 擦纸吸掉的，然后再用湿的厨房纸巾一擦，就很干净了。Swiffer 的擦纸擦了之后就可以直接扔掉，省了不少事。

在美国，厨房纸巾是很常见的东西，擦脏了就扔掉。很多中国人还在用毛巾擦东西，擦脏了还得去用水淘它。一般你

得有好几条抹布，有些抹布拿去擦了厕所，你就不能用来擦厨房了，你得记住哪条是“脏抹布”，哪条是“干净抹布”…… 麻不麻烦？由于厨房纸巾结实又不掉渣，甚至可以用水冲了再用，基本可以当毛巾来用，我很多时候擦东西都只用厨房纸巾，把一个地方擦干净了就扔掉。

但有一个例外，厨房纸巾用来擦玻璃或者镜子，无论如何都会留下一些微小的残渣，这在普通家具上面是看不到的。所以擦玻璃或者镜子，我一般不用纸巾，而是用微纤维抹布。这种微纤维抹布不会留下任何痕迹，弄脏了很容易清洗干净。另外，擦玻璃有专用的玻璃清洁剂（比如 Invisible Glass），不会留下任何痕迹的那种。

有些人擦厨房里的案台和灶具，喜欢在纸巾或者毛巾上面弄点餐具洗洁精，以为这样擦得干净。可是餐具洗洁精很不容易清干净，一旦放上去你就需要很多遍的清水才能去掉，这实在是太麻烦了。

其实最好的办法不是用餐具洗洁精，而是用像 Lysol, Clorox 之类的生活清洁剂或者擦巾。Lysol 之类的产品里面含有可挥发的氨，它去污力很强，无毒，不怎么伤手，而且擦了之后可以不用清水漂洗，自己就挥发掉了。

不过除非表面有清水擦不掉的脏东西，我一般也不用 Lysol，毕竟它还是有一点味道，而且很多时候你没必要使用它的杀菌能力。所以我一般就用清水，偶尔遇到清水擦不干净的，就用 Lysol。

总之，清洁用品和技巧有很多门类，我这里只是抛砖引玉，提示大家这里面有可以研究的学问。我就不多啰嗦了，具体的问题还得靠你自己去分析和处理。

中国人的洁癖

很多中国人有一种非理性的洁癖，或者“健康癖”。他们总觉得外面是脏的，所以如果出去坐过的裤子，就不能再坐在家里的床上或者沙发上。如果你去医院坐过，那就得一回家就把裤子脱下来洗了，完全不可以碰家里的东西，万一你做过的椅子是皮肤病人或者性病病人坐过的呢！

还有人觉得洗了外衣的洗衣机就不可以再用来洗内衣，因为外衣被认为是脏的，而内衣接触皮肤，应该很干净才对。甚至有人专门买了“小小神童洗衣机”，专门拿来洗内衣内裤。如果没有这种洗衣机，他们的家里会分别有洗各种东西的盆子：洗袜子的，洗内衣的，……

所有这些都是没有经过科学分析得出的结论。这些人似乎并不理解病菌是如何生存，如何传播的，就盲目的认为在外面坐过的裤子上会有对身体有害的病菌。

这里我不得不称赞一下美国人的理性思维。我曾经问过好几个美国人，你们怎么洗鞋子啊？他们都觉得这个问题很奇怪：“什么？不就是丢进洗衣机吗？”这显然是大部分中国人觉得不可思议的事情，他们会跟你说那是因为美国很干净，所以才可以这样子。

然而并非如此，美国人可以穿着那鞋子到处走，那鞋子可能去过中国，去过非洲，到很脏的厕所里踩过…… 回到美国，忘了这些事，把鞋子丢进洗衣机…… 嘿嘿，恶心吧？

然而事实就是从来没有人因此生病，因为病菌不可能通过那样的方式传播。粘在鞋子上的病菌，必须要有营养和水才能生存，而鞋子上就算粘了它们可以生存的粘液，过段时间也会干掉。无论如何，鞋子进了洗衣机，病菌都会被洗衣液给消灭掉。所有的灰尘之类，会随着漂洗冲到下水道里去。

我亲自试验过用洗衣机洗鞋子，鞋子当然要单独洗了。你最好丢一些破布跟鞋子一起洗，这样破布可以帮助擦洗鞋子。倒上洗衣液，启动洗衣机就完了。洗了之后再看洗衣机的桶，没有发现什么残留的脏东西。之后再拿来衣服，内衣，也没有发现任何问题。

如果你不放心，可以买专用的洗衣机清洁剂，在这之后丢进洗衣机，启动自清洗程序。总之，过度的担心卫生和健康，似乎是中国人的通病，也是他们的生活过度复杂的原因。世界上并没有那么多致病的细菌，你得理解它们的生存和传播方式，而不是盲目的认为它们可以粘在任何地方。

清洁经常用手接触的物体

虽说很多中国人有洁癖，觉得这也脏那也脏，把生活搞得不必要的复杂，然而他们却往往忽视有些“隐蔽位置”的清洁。这些位置经常用手触摸，却没有得到重视。有趣的是，洁癖人士们对待它们的策略是：尽量不碰它们，而不是把它们弄干净。对于这些位置的疏忽，带来了他们反反复复的不爽，却一直被忽视。

下面我就列举一些这样的物体：

1. 电灯开关，电源插座。我发现很多中国家庭的电灯开关和电源插座都糊着厚厚的一层污垢或者油烟。他们打扫卫生，把其他地方都擦干净了，就是不擦这两个东西。可是经过简单的观察，你就会发现电灯开关和电源插座，其实是你的手接触最频繁的两个设备。每天你都要摸电灯开关好多次，每一次都是那种黏糊糊的感觉，你觉得舒服吗？这种不舒服的感觉，反反复复，不知不觉中引起了细微的情绪变化，所以你就不会安心和舒服。我发现这个问题，所以每当我发现电灯开关比较脏了，我就会用消毒巾仔细把它擦干净。这样不但对手是一种享受，而且对眼睛也是一种保养。
2. 电线，充电线。电器的电源线和手机充电线一类的东西，如果上面有很多污垢，拿起来就会很不爽。特别是手机充电线，经常要用手拿，所以要注意清洁干净。由于 iPhone 自带的充电线外面是橡胶，特别容易粘脏东西上

去，就算是新的拿起来也黏糊糊的感觉，所以我一般都不用 iPhone 自带的充电线。Amazon Basics 的充电线感觉好很多，另外你也可以试试 Belkin 之类的牌子。

3. 键盘，鼠标。很多人的电脑键盘和鼠标上面有很多垢，平时太忙了忽视了清洁。但如果你静下心来，就会发现肮脏的键盘和鼠标，跟手接触非常频繁。虽说没有什么健康威胁，却带来了很多的不爽。就像人需要修剪指甲一样，稍微花点时间把键盘和鼠标擦一下，平时心情会好很多。

过时的洗脸盆

最近十多年的生活，我发现有一件东西从我的生活中消失了，那就是盆子：洗脸盆，洗脚盆，搓衣盆……可是我发现很多中国人的家里仍然有盆子这东西，所以我很纳闷为什么我曾经需要它们，而为什么现在不再需要。

后来我发现盆子从我生活中消失的原因，是跟热水器的出现密不可分的。在很早的时候，人们没有燃气热水器，所以要洗脸，只有用脸盆来混合冷水和开水。先把脸盆装点冷水，然后用暖水瓶加一些开水在里面，然后放一条洗脸毛巾进去弄湿了，拿出来洗脸。

燃气热水器的出现改变了这个局面。因为水龙头放出来的水直接可以调温，所以很多人开始直接用手从水龙头捧热水来洗脸，而不再需要一个容器来存放这些热水。使用流水洗脸不但更加卫生，而且消除了对洗脸盆这种东西的需求。

可是很多年长的人不理解，看到现代家居里面的“洗脸池”，还以为那是用来放热水在里面，然后才放毛巾进去洗脸。这种不理解的人，往往觉得洗脸池很脏，不知道该怎么用它，所以仍然去买塑料洗脸盆来接水洗脸。由于洗脸池的形状构造，他们发现这非常不方便，不顺手，而其实只是他们不理解它的用法。

如果你理解了热水器和流水洗脸的原理，就会发现这种所谓“洗脸池”，其实并不是用来盛放洗脸水的。它只是用来接你洗过脸，刷过牙的脏水的。所以它根本不需要特别干净，多人共用这种池子，一点问题也没有。

同样的原理，刷牙其实也可以不用杯子。因为水龙头就在面前，你完全可以用手捧一点水到嘴里，那就足够用来刷牙了。所以我选了一个漂亮的杯子用来放牙刷，而从来不用它来装水。另外我发现有些人喜欢把牙膏也放进杯子里，由于牙膏很胖很重，那会使得取用变得很蹩脚，有时候杯子还会因为重心不稳被弄翻。其实牙膏放在案台上就可以了，只有牙刷需要放进杯子里，因为它上面沾了水。

我家里只有一种盆子，那就是“足疗盆”。我发现洗脸池和浴缸都不能达到这个功效，所以我买了一个。放上一盆热水，加上舒缓筋骨的浴盐，放在漂亮的地毯和毛巾上面，把脚放进去，猫咪在旁边陪伴，真是一种惬意的享受 :)



洗碗

洗碗是世界上最讨厌的活了。很多中国家庭里面常见的现象就是，吃了饭马上就必须有一个人去刷碗。这样很不好，本来享受了美食，应该惬意地喝点茶，聊聊天，家人之间交流感情。结果吃了饭，大家想的是该谁去洗碗，然后一个人去厨房洗碗，没法交流了。大家都想洗就更麻烦，甚至伤感情。

对于这个问题，我的方案是，吃了饭大家都休息，不要去洗碗。如果需要腾出桌子喝茶，那可以把碗都扔到洗碗池里面，然后开始休息。想一下，你们那么急着洗碗到底有什么好处？没有。

还有一个更好的办法，那就是使用洗碗机。我觉得洗碗机是世界上最伟大的发明之一。洗碗机是美国房子的标配，很多在美国的中国人却把它用作碗柜，从来不启动它。他们觉得洗碗机会很费水费电，所以舍不得用。这是一个误区，如果你观察洗碗机的构造，就会发现它不会用很多水。它的原理是用一个很大的涡轮把少量的水高速喷出，水里面含有烈性的洗涤剂，这样冲刷餐具，达到清洁的目的。所以终究它不会用掉很多水，肯定比洗澡水要少很多，而且看起来也不怎么费电。

这个我是做过实验的，发现用洗碗机之后我的水电费并没有比不用它的时候增加多少。洗碗机的专用洗涤剂也不贵。我见过有人为了“节约”，想把洗涤剂的小包切开分成两次用，结果里面的液体流出来，很伤手。几毛钱一包的东西，就不必这么省了吧:P

那么单身汉平时在家就吃一点东西，没有几个碗，放进洗碗机不是大材小用吗？我再告诉你一个诀窍吧。我单身的时候也有这种想法，但是后来发现一个窍门。那就是去买很多碗，盘子，叉子，筷子，就像你有一大家子人一样。我之前一个人的时候，给自己买了至少十套餐具，目的就是利用这个洗碗机。你每次吃饭拿出干净的碗和餐具，吃完就丢

到洗碗机里，但不启动洗碗机，等它积累起来。等洗碗机装满了，或者过了两三天还没有装满，你就启动洗碗机……明白了吧？

开车

说了这么多家务事，来讲讲出门的事情。很多中国人开车的时候很着急，喜欢紧跟在别人屁股后面，生怕旁边有车插进来。这种人到了美国还是一样的想法，要是别人老从旁边插进我的车道，那我不就慢下来了吗？所以我就紧跟前面车的屁股，这样别人就插不进来了。

这种人就是没有仔细分析过问题：

- 有多大的概率，有人会从旁边车道插到你的前面？
- 对于每辆插进来的车，它会延迟你多少时间到达？
- 你是否在乎这点时间？

等你仔细分析和观察之后，就发现就算你跟前车保持很长的距离，任凭别人插到你前面，也不会比原来的时间慢多少。因为汽车本来就跑得很快，而并不会有多少人会想插到你前面，不是每个人都像你这么急！

而且跟前车屁股太近有一个很不好的问题，那就是如果前车踩刹车，你就得踩刹车。到后来你的神经就完全被前面的司机控制了，别人刹车你得很快做出反应，不然就可能撞车。你当然可以随时保持警惕，避免撞车，但如果你保持比较大的车距，那么就不需要这么紧张了。你可以有比较长的反应时间，优哉游哉的开你的车，不受别人的左右。

各种贴膜

经常看到有人拿着一个贴膜的手机，那膜都磨花得不行了，还有些人的膜很硬，导致触屏响应严重受影响。经过一番道理，我说服了他们其中一些人把膜扔了。扔掉之后，他们才发现原来的屏幕是那么的好看那么的亮，响应是如此的流畅！然后他们才开始遗憾，为什么自己一直在用一个比现在差很多的手机。

我对人们对手机屏幕的爱护真是深感震撼。要是他们有那么爱护自己的身体，那该多好啊。

我是怎么说服他们揭掉手机膜的呢？我告诉他们，我的第一台智能手机也贴了膜。等到换新手机的时候，我把它的膜揭开，然后用钥匙和刀子在上面使劲的划了好半天。结果，屏幕完好无损，连一点细微的划痕都没有！

经过很简单的实验，你就可以让自己相信手机屏幕的玻璃是如此之硬，它是完全不怕钥匙之类的硬物摩擦的。我拿来做试验的手机是 2008 年的产物了，现在的手机屏玻璃又更新换代，肯定更不怕划伤了。所以手机贴膜真是多此一举，害得自己不能更好的享受重金买来的物品。

还有一种常见的“膜”，就是木头桌子上面放个胶皮或者玻璃板。很多人买了很好的实木桌子，觉得价格很贵，所以又买了一张胶皮来保护它。后来又发现这胶皮价格也不便宜，所以又买了几张小胶皮来保护这张大胶皮。所有的碗都必须放在小胶皮上面…… 最后，你就发现你用的不是豪华的实木餐桌了，而是一张胶皮桌子。你花了几千块钱，结果买了一张胶皮桌子！

实际上实木桌子根本就不需要保护，它非常的结实耐用。木头给人的感觉是温馨舒服的，却用一张胶皮盖起来，完全抹杀了这种感觉。如果没有胶皮，正常的磨损不可避免，然而桌子的魅力不会改变。这些微小的划痕，正好增加了这种木头桌子的魅力，不是吗。我就见过商店里卖这种有划痕和烫伤的实木桌子的，中间还有树洞。卖的很贵，因为它很有感觉。

不过这种桌子上面，倒可以在吃饭的位置放上漂亮的餐垫，这样吃饭的时候餐具不会在上面发出磕碰声，感觉更加舒服温馨。

同样的道理，你也不应该把各种遥控器放进保护套。它影响了你对遥控器的使用感觉，有时候模糊得看不清楚按钮上的字，却并不能让遥控器用得更久。实际情况是电器的换代速度比遥控器的寿命要短很多，电器过时换代的时候，遥控器往往还是好好的。而且像遥控器那么便宜的东西，真有必要保护它吗？上网搜一下“万能遥控器”卖多少钱就知道了。

我买车的时候，车行向我推荐一种“皮革保护涂料”，说喷上去之后它可以在皮革表面形成一层保护膜，可以使座椅不怕水，不怕脏东西，这样车再卖出去的时候可以保值。我拒绝了这个推销，调侃道：“我希望坐在皮革上面，而不是坐在不知道什么化学材料做成的涂料上面 :）”

这是一样的道理：我买了皮革座椅的车子，我希望享受它，而不是损害自己的舒适程度来保护它。买来的物品是为人服务的，而不是用来保护起来保值的。后来这车子开了几年，座椅还像新的一样，这说明这些什么皮座椅保护涂料根本就没有必要。

那么同样的原理，是不是说我们不需要给 iPhone 买保护壳呢？不是的，iPhone 是一个不幸的个例。因为 iPhone 本来的壳子太滑太圆了，你不给它装个壳子是注定要滑落到地上的！所以不得已，你是得给 iPhone 买个壳子。

铲屎官的秘诀

我养了一只猫，对于养猫我也有很多的智慧，把他管理的井井有条，家里一点味道都没有。这里我只透露一个秘密，那就是我用了 [Litter Genie](#)。

这是一个不起眼的塑料盒子，专门用来装猫屎的。它的特点是非常好的隔离效果，猫屎进去之后就完全闻不到味道了。你可以每天铲猫屎放进去，等几个星期才一起拿去扔掉也不会有问题。

起初朋友介绍给我的时候，我还不以为然的说，我每天铲屎放进一个垃圾袋，马上拿去扔了不就行了？等到真正用了 Litter Genie 几个月之后，才发现没有它是多么的不方便，因为我已经习惯了简单的：铲屎，盖上，完事。

我养猫的经验也可以写成一本书了，所以这里限于篇幅就不多说了:P

总结

嗯，写了这么多，我自己都有点糊涂了，有点没条理了。生活的智慧当然不止这一点，它是博大精深的，所以我这里讲的一点东西只是抛砖引玉，希望帮助大家追求有品质的生活。如果你有什么生活小窍门很多人不知道的，欢迎来信跟我交流。如果你知道文中提到的美国产品（比如 Lysol）的中国等价物，也请告诉我。

（如果你觉得这篇文章有所帮助，可以[付款购买](#)，建议价格 ¥ 30。）

如何掌握所有的程序语言

对的，我这里要讲的不是如何掌握一种程序语言，而是所有的.....

很多编程初学者至今还在给我写信请教，问我该学习什么程序语言，怎么学习。由于我知道如何掌握“所有”的程序语言，总是感觉这种该学“一种”什么语言的问题比较低级，所以一直没来得及回复他们:P 可是逐渐的，我发现原来不只是小白们有这个问题，就连美国大公司的很多资深工程师，其实也没搞明白。

今天我有动力了，想来统一回答一下这个搁置已久的“初级问题”。类似的话题貌似曾经写过，然而现在我想把它重新写一遍。因为在跟很多人交流之后，我对自己头脑中的（未转化为语言的）想法，有了更精准的表达。

如果你存在以下的种种困惑，那么这篇文章也许会对你有所帮助：

1. 你是编程初学者，不知道该选择什么程序语言来入门。
2. 你是资深的程序员或者团队领导，对新出现的种种语言感到困惑，不知道该“投资”哪种语言。
3. 你的团队为使用哪种程序语言争论不休，发生各种宗教斗争。
4. 你追逐潮流采用了某种时髦的语言，结果两个月之后发现深陷泥潭，痛苦不堪.....

虽然我已经不再过问这些世事，然而无可置疑的现实是，程序语言仍然是很重要的话题，这个情况短时间内不会改变。程序员的岗位往往要求熟悉某些语言，甚至某些奇葩的公司要求你“深入理解 OOP 或者 FP 设计模式”。对于在职的程序员，程序语言至今仍然是可以争得面红耳赤的宗教话题。它的宗教性之强，以至于我在批评和调侃某些语言（比如 Go 语言）的时候，有些人会本能地以为我是另外一种语言（比如 Java）的粉丝。

显然我不可能是任何一种语言的粉丝，我甚至不是 Yin 语言的粉丝;) 对于任何从没见过的语言，我都是直接拿起来就用，而不需要经过学习的过程。看了这篇文章，也许你会明白我为什么可以达到这个效果。理解了这里面的东西，每个程序员都应该可以做到这一点。嗯，但愿吧。

重视语言特性，而不是语言

很多人在乎自己或者别人是否“会”某种语言，对“发明”了某种语言的人倍加崇拜，为各种语言的孰优孰劣争得面红耳赤。这些问题对于我来说都是不存在的。虽然我写文章批评过不少语言的缺陷，在实际工作中我却很少跟人争论这些。如果有其它人在我身边争论，我甚至会戴上耳机，都懒得听他们说什么;) 为什么呢？我发现归根结底的原因，是因为我重视的是“语言特性”，而不是整个的“语言”。我能用任何语言写出不错的代码，就算再糟糕的语言也差不了多少。

任何一种“语言”，都是各种“语言特性”的组合。打个比方吧，一个程序语言就像一台电脑。它的牌子可能叫“联想”，或者“IBM”，或者“Dell”，或者“苹果”。那么，你可以说苹果一定比 IBM 好吗？你不能。你得看看它里面装的是什么型号的处理器，有多少个核，主频多少，有多少 L1 cache，L2 cache.....，有多少内存和硬盘，显示器分辨率有多大，显卡是什么 GPU，网卡速度，等等各种“配置”。有时候你还得看各个组件之间的兼容性。

这些配置对应到程序语言里面，就是所谓“语言特性”。举一些语言特性的例子：

- 变量定义
- 算术运算
- for 循环语句，while 循环语句
- 函数定义，函数调用
- 递归
- 静态类型系统
- 类型推导
- lambda 函数
- 面向对象
- 垃圾回收
- 指针算术
- goto 语句

这些语言特性，就像你在选择一台电脑的时候，看它里面是什么配置。选电脑的时候，没有人会说 Dell 一定是最好的，他们只会说这个型号里面装的是 Intel 的 i7 处理器，这个比 i5 的好，DDR3 的内存比 DDR2 的快这么多，SSD 比磁盘快很多，ATI 的显卡是垃圾..... 如此等等。

程序语言也是一样的道理。对于初学者来说，其实没必要纠结到底要先学哪一种语言，再学哪一种。曾经有人给我发信问这种问题，纠结了好几个星期，结果一个语言都还没开始学。有这纠结的时间，其实都可以把他纠结过的语言全部掌握了。

初学者往往不理解，每一种语言里面必然有一套“通用”的特性。比如变量，函数，整数和浮点数运算，等等。这些是每个通用程序语言里面都必须有的，一个都不能少。你只要通过“某种语言”学会了这些特性，掌握这些特性的根本概念，就能随时把这些知识应用到任何其它语言。你为此投入的时间基本不会浪费。所以初学者纠结要“先学哪种语

言”，这种时间花的很不值得，还不如随便挑一个语言，跳进去。

如果你不能用一种语言里面的基本特性写出好的代码，那你换成另外一种语言也无济于事。你会写出一样差的代码。我经常看到有些人 Java 代码写得相当乱，相当糟糕，却骂 Java 不好，雄心勃勃要换用 Go 语言。这些人没有明白，是否能写出好的代码在于人，而不在于语言。如果你的心中没有清晰简单的思维模型，你用任何语言表述出来都是一堆乱麻。如果你 Java 代码写得很糟糕，那么你写 Go 语言代码也会一样糟糕，甚至更差。

很多初学者不了解，一个高明的程序员如果开始用一种新的程序语言，他往往不是去看这个语言的大部头手册或者书籍，而是先有一个需要解决的问题。手头有了问题，他可以用两分钟浏览一下这语言的手册，看看这语言大概长什么样。然后，他直接拿起一段例子代码来开始修改捣鼓，想法把这代码改成自己正想解决的问题。在这个简短的过程中，他很快的掌握了这个语言，并用它表达出心里的想法。

在这个过程中，随着需求的出现，他可能会问这样的问题：

- 这个语言的“变量定义”是什么语法，需要“声明类型”吗，还是可以用“类型推导”？
- 它的“类型”是什么语法？是否支持“泛型”？泛型的“variance”如何表达？
- 这个语言的“函数”是什么语法，“函数调用”是什么语法，可否使用“缺省参数”？
-

注意到了吗？上面每一个引号里面的内容，都是一种语言特性（或者叫概念）。这些概念可以存在于任何的语言里面，虽然语法可能不一样，它们的本质都是一样的。比如，有些语言的参数类型写在变量前面，有些写在后面，有些中间隔了一个冒号，有些没有。

这些实际问题都是随着写实际的代码，解决手头的问题，自然而然带出来的，而不是一开头就抱着语言手册看得仔仔细细。因为掌握了语言特性的人都知道，自己需要的特性，在任何语言里面一定有对应的表达方式。如果没有直接的方式表达，那么一定有某种“绕过方式”。如果有直接的表达方式，那么它只是语法稍微有所不同而已。所以，他是带着问题找特性，就像查字典一样，而不是被淹没于大部头的手册里面，昏昏欲睡一个月才开始写代码。

掌握了通用的语言特性，剩下的就只剩某些语言“特有”的特性了。研究语言的人都知道，要设计出新的，好的，无害的特性，是非常困难的。所以一般说来，一种好的语言，它所特有的新特性，终究不会超过一两种。如果有个语言号称自己有超过 5 种新特性，那你就得小心了，因为它们带来的和可能不是优势，而是灾难！

同样的道理，最好的语言研究者，往往不是某种语言的设计者，而是某种关键语言特性的设计者（或者支持者）。举个例子，著名的计算机科学家 Dijkstra 就是“递归”的强烈支持者。现在的语言里面都有递归，然而你可能不知道，早期的程序语言是不支持递归的。直到 Dijkstra 强烈要求 Algol 60 委员会加入对递归的支持，这个局面才改变了。Tony Hoare 也是语言特性设计者。他设计了几个重要的语言特性，却没有设计过任何语言。另外大家不要忘了，有个语言专家叫王垠，他是早期 union type 的支持者和实现者，也是 checked exception 特性的支持者，他在自己的[博文中](#)指出了 checked exception 和 union type 之间的关系 :P

很多人盲目的崇拜语言设计者，只要听到有人设计（或者美其名曰“发明”）了一个语言，就热血沸腾，佩服的五体投地。他们却没有理解，其实所有的程序语言，不过是像 Dell，联想一样的“组装机”。语言特性的设计者，才是像 Intel，AMD，ARM，Qualcomm 那样核心技术的创造者。

合理的入门语言

所以初学者要想事半功倍，就应该从一种“合理的”，没有明显严重问题的语言出发，掌握最关键的语言特性，然后由此把这些概念应用到其它语言。哪些是合理的入门语言呢？我个人觉得这些语言都可以用来入门：

- Scheme
- C
- Java
- Python
- JavaScript

那么相比之下，我不推荐用哪些语言入门呢？

- Shell
- PowerShell
- AWK
- Perl
- PHP
- Basic
- Go
- Rust

总的说来，你不应该使用所谓“[脚本语言](#)”作为入门语言，特别是那些源于早期 Unix 系统的脚本语言工具。PowerShell 虽然比 Unix 的 Shell 有所进步，然而它仍然没有摆脱脚本语言的根本问题——他们的设计者不知道他们自己在干什么 :P

采用脚本语言学编程，一个很严重的问题就是使得学习者抓不住关键。脚本语言往往把一些系统工具性质的东西（比如正则表达式，Web 概念）加入到语法里面，导致初学者为它们浪费太多时间，却没有理解编程最关键的概念：变量，函数，递归，类型……

不推荐 Go 语言的原因类似，虽然 Go 语言不算脚本语言，然而他的设计者显然不明白自己在干什么。所以使用 Go 语言来学编程，你不能专注于最关键，最好的语言特性。关于 Go 语言的各种毛病，你可以参考这篇[文章](#)。

同样的，我不觉得 Rust 适合作为入门语言。Rust 花了太大精力来夸耀它的“新特性”，而这些新特性不但不是最关键的部分，而且很多是有问题的。初学者过早的关注这些特性，不仅学不会最关键的编程思想，而且可能误入歧途。关于 Rust 的一些问题，你可以参考这篇[文章](#)。

掌握关键语言特性，忽略次要特性

为了达到我之前提到的融会贯通，一通百通的效果，初学者应该专注于语言里面最关键的特点，而不是被次要的特性分心。

举个夸张点的例子。我发现很多编程培训班和野鸡大学的编程入门课，往往一来就教学生如何使用 printf 打印“Hello World！”，进而要他们记忆 printf 的各种“格式字符”的意义，要他们实现各种复杂格式的打印输出，甚至要求打印到文本文件里，然后再读出来……

可是殊不知，这种输出输入操作其实根本不算语言的一部分，而且对于掌握编程的核心概念来说，都是次要的。有些人的 Java 课程进行了好几个星期，居然还在布置各种 printf 的作业。学生写出几百行的 printf，却不懂得变量和函数是什么，甚至连算术语句和循环语句都不知道怎么用！这就是为什么很多初学者感觉编程很难，我连 %d, %f, %.2f 的含义都记不住，还怎么学编程！

然而这些野鸡大学的“教授”头衔是如此的洗脑，以至于被他们教过的学生（比如我女朋友）到我这里请教，居然骂我净教一些没用的东西，学了连 printf 的作业都没法完成：P 你别跟我讲 for 循环，函数什么的了…… 可不可以等几个月，等我背熟了 printf 的用法再学那些啊？

所以你就发现一旦被差劲的老师教过，这个程序员基本就毁了。就算遇到好的老师，他们也很难纠正过来。

当然这是一个夸张的例子，因为 printf 根本不算语言特性，但这个例子从同样的角度说明了次要肤浅的语言特性带来的问题。

这里举一些次要语言特性的例子：

- C 语言的语句块，如果里面只有一条语句，可以不打花括号。
- Go 语言的函数参数类型如果一样可以合并在一起写，比如 `func foo(s string, x, y, z int, c bool) { ... }`
- Perl 把正则表达式作为语言的一种特殊语法
- JavaScript 语句可以在某些时候省略句尾的分号
- Haskell 和 ML 等语言的 [currying](#)

自己动手实现语言特性

在基本学会了各种语言特性，能用它们来写代码之后，下一步的进阶就是去实现它们。只有实现了各种语言特性，你才能完全地拥有它们，成为它们的主人。否则你就只是它们的使用者，你会被语言的设计者牵着鼻子走。

有个大师说得好，完全理解一种语言最好的方法就是自己动手实现它，也就是自己写一个解释器来实现它的语义。但我觉得这句话应该稍微修改一下：完全理解一种“语言特性”最好的方法就是自己亲自实现它。

注意我在这里把“语言”改为了“语言特性”。你并不需要实现整个语言来达到这个目的，因为我们最终使用的是语言特性。只要你自己实现了一种语言特性，你就能理解这个特性在任何语言里的实现方式和用法。

举个例子，学习 SICP 的时候，大家都会亲自用 Scheme 实现一个面向对象系统。用 Scheme 实现的面向对象系统，跟 Java, C++, Python 之类的语言语法相去甚远，然而它却能帮助你理解任何这些 OOP 语言里面的“面向对象”这一概念，它甚至能帮助你理解各种面向对象实现的差异。

这种效果是你直接学习 OOP 语言得不到的，因为在学习 Java, C++, Python 之类语言的时候，你只是一个用户，而用 Scheme 自己动手实现了 OO 系统之后，你成了一个创造者。

类似的特性还包括类型推导，类型检查，惰性求值，如此等等。我实现过几乎所有的语言特性，所以任何语言在我的面前，都是可以被任意拆卸组装的玩具，而不再是凌驾于我之上的神圣。

总结

写了这么多，重要的话重复三遍：语言特性，语言特性，语言特性，语言特性！不管是初学者还是资深程序员，应该专注于语言特性，而不是纠结于整个的“语言品牌”。只有这样才能达到融会贯通，拿起任何语言几乎立即就会用，并且写出高质量的代码。

(如果你觉得这篇文章有所帮助，可以[付款](#)购买，价格随意。)

带猫回国经历



经过一番折腾，我和小莫奈已经顺利到达成都家里。在此我感谢各位朋友给我的信息和指点。第一次带宠物回国的人，不免面临困惑和压力，所以我想把我的经验总结一下。

带宠物回国，这个事情说起来轻松，做起来压力其实蛮大的。我为这个事情，处心积虑至少有两个月，每一个环节弄得不好都可能出问题。网络上的信息有些过时了，有些啰嗦太多。毕竟带了宠物回国的人如释重负之后，不免喜欢附加各种“爱心废话”：) 在这里我尽量避免啰嗦废话，以便需要信息的人能够直接得到“指示”。

找一个 USDA 认证的兽医

直接找一个 USDA (美国农业部) 认证的兽医，你的问题就解决了一大半了。兽医会告诉你该做些什么。你可能就不需要看网上的各种攻略了，也不需要看我这篇文章。

注意一定要是 USDA 认证的兽医，因为美国有很多兽医都不是 USDA 认证过的。

由于每个人的出发地，目的地和其它情况都可能不同，没有任何其他人的攻略可以代替本地的兽医给你专门的指点。所以我觉得任何攻略类文章，都比不上找兽医来的靠谱。

看了网上的多篇文档之后，我直接带上猫去了一个 USDA 认证的兽医那里，跟他说我要带猫去中国，该怎么办。他给了我几乎所有的信息，包括疫苗，健康证明，各种时间调度问题，最近的 USDA 办公室的位置，等等。

所以如果你看了很多文章还不清楚，那最好直接找本地的兽医问。兽医应该清楚如何带宠物进入中国。

权威文档

从美国带猫狗到中国的权威流程请见 USDA 的网站：

<https://www.aphis.usda.gov/aphis/pet-travel/by-country/petravel-china>

下面我只介绍一下我带猫到成都的故事。由于我选择了最简单的方式，也许你的情况还得自己多考虑。

狂犬疫苗（最先考虑的问题）

如果距上次狂犬疫苗注射已经超过一年，临行至少一个月之前（注意不是一个月之内），到兽医那打狂犬疫苗，开疫苗证明。这是因为中国海关需要狂犬疫苗注射时间在一年以内，一个月以外。也就是说，一个月以内打的疫苗，或者一年以外，就会有问题。

注意，疫苗证明上面要有狂犬疫苗的序列号（serial number），如果没有，请当时就要求他们打印出来。由于我的兽医没把序列号印在疫苗证明上面，导致我到 USDA 盖章的时候费了很多时间和口舌，而且可能在中国海关那里遇到麻烦。

这是你应该最早考虑的问题。我的疫苗是在两个月之前打的，你提前三，四个月都没问题，但不能超过一年，否则就失效了。

选择航班，买机票

你必须先选好航班，定好出发时间，买好机票。因为之后的健康证明有很狭窄的时间限制（10天），而且可能还需要提前预约。建议订机票最好提前一个月。

尽量选择美国航空公司（UA，Delta，……）的直达航班回国。因为只有美国公司的航班可以带宠物进入机舱。如果需要在国内转机，国内的那个飞机肯定是不准你带宠物进机舱的，那你就需要考虑托运。

总之为了简便，我选择了唯一能直达成都的航班，硬着头皮坐了美联航的飞机 (UA9) ☺ 你要坐中国公司的飞机，或者坐需要转机的飞机，我这里就帮不了你了，可以参考一下其他人的攻略。

因为一架飞机上面最多可以带进机舱的宠物数目有限（貌似 4 只？），不然宠物们联合起来可能劫持飞机 :P，所以买机票之前，打电话给航空公司确认那天的飞机还可以带宠物。貌似一般都可以，因为难得那么多人带宠物坐同一架飞机。买了自己的机票之后，打电话去航空公司，要求加一只舱内宠物 (In-cabin Pet)，花费 \$125。

因为宠物需要占据座位前面放脚的空间，我买了美联航的 Economy Plus，这种座位之间的距离大一点，更舒服一些。

国际健康证明

临行前 10 天之内，带上猫，去 USDA 认证 (USDA accredited) 的兽医那里进行体检，开“国际健康证明” (International Health Certificate)。因为兽医有可能很忙，这个你最好在订好机票之后，马上预约时间。

注意这里有一个糊涂的地方：USDA 会告诉你健康证明是 30 天有效，然而航空公司却告诉你是 10 天。为了保险起见，你取最小值，也就是 10 天之内。否则航空公司有可能不让你上飞机。所以为了万无一失，一定要在 10 天以内。

!!!!!! 关于 UA 办票人员业务水平的警告 !!!!!

有前人说在美国机场带宠物上飞机的时候根本没人看这些健康证明，只有中国海关要看这些，但我遇到的实际情况是，美联航办登机牌的大妈听说我要带猫去中国，差点当场晕过去，因为她从来没办过这个事情，也不知道该怎么办！UA 你是怎么培训办票人员的？

我看她一脸惶恐，旁边的同事帮她拿出一个电话本来，临时抱佛脚打电话找人问。后来又跑到别的柜台去找人帮忙，足足在这证明上面费了半个小时，差点害我误了飞机。我过了安检就收到登机口的电话催我，结果最后一个上飞机。几个星期连续准点的航班，因此延误了 20 多分钟才起飞 ☺

健康证明上面需要有狂犬疫苗的各种信息，包括型号和序列号。我找的是同一个兽医，所以他已经有我的狂犬疫苗信息，直接就填进去了。如果你的疫苗是在其它兽医那里打的，就得带上那个疫苗证明。为了简单可靠，我建议你找同一个兽医办这些事情，记得一定要问清楚他是否 USDA 认证的兽医，因为有很多兽医都不是 USDA 认证的！

健康证明的格式，必须是 USDA 网站上针对中国的文档格式：

<https://www.aphis.usda.gov/pet-travel/health-certificates/non-eu/china-cat.pdf>

注意更新：USDA 已经不再接受老攻略上的 APHIS 7001 国际健康证明卡。不过最好叫兽医把 Aphis 7001 也开上，两个文档都带上。我的兽医说航空公司可能要看 APHIS 7001。由于美联航的傻瓜糊里糊涂折腾了半天，我没能搞明白他们到底在看哪个文件（晕）。总之，我把两个证明都给了他们，折腾了半个小时，我才拿到登机牌！

所以对不起，这里我不能提供更精确的信息，你让兽医把两个都准备好就是了。USDA 认证的兽医一般都知道怎么办这个事，所以你可以多咨询他们。如果你有更准确的信息，请来信告诉我。

最后再鄙视一下 UA :P

USDA 盖章

有了兽医开的健康证明还不够，你得把健康证明拿到 USDA 的办公室去盖章。盖章收费 \$38，你必须同时带上狂犬疫苗证明，USDA 要核对健康证明上的疫苗信息跟你的狂犬疫苗一样，包括序列号。

由于我的兽医似乎业务水平有问题，所以他没把狂犬疫苗的序列号打印在疫苗证明上。结果在 USDA 遇到了麻烦。他们跟我说那个疫苗证明不对，他们需要一个疫苗的序列号，要跟健康证明上那个一致，因为中国海关好像需要那个序列号。好像…… ☺ 我再三强调了那就是兽医给我的疫苗证明，我没有别的文件了，她这才同意打电话给兽医确认一下。在电话上折腾了好一阵子，让兽医把有序列号信息的文件发 email 给她，才给我盖了章。

湾区童鞋参考：我是到 SFO 机场附近那个 USDA 办公室盖的章。地址是：

USDA APHIS Veterinary Services SFO Port

389 Oyster Point Blvd. Suite 2B; South San Francisco, CA 94080

这个办公室必须提前预约才给你办这些事，所以一定要提前准备。预约方式是发 email 到这个地址：

sfo.port.services@aphis.usda.gov

告诉他们你的航班时间，要去哪里。他们会直接给你预约时间。

其它地方的人，得根据自己的情况灵活处理。不方便直接去的话，USDA 也接受邮件处理，貌似把材料寄过去，加上回邮信封和支票就可以。这个你得参考其他人的攻略了，或者电话 USDA 问清楚。

这个办公室付款方式可以接受信用卡，不像前人经验说只能收支票，不过我建议还是把支票带上。

宠物包

另外，记得提前买好宠物包，因为你得给猫咪一定的时间来熟悉这个包，这样他才会对它有安全感。宠物包挺有讲究，所以我详细介绍一下。

我买的是[这个](#)侧面可以伸出来一块扩展空间的宠物包。事实证明，它的大小差不多合适，伸出来的那一块在飞机上也起了点作用。



这个猫包上面和前面各有一道门，我感觉上面这道门对于飞机旅行是很重要的。因为过安检的时候，你得把猫抱出来跟你一起过，而不能让他经过 X 光机。由于猫害怕外面的环境，如果你从前面那道小门可能很难把他拽出来。

我把这包放在家里，敞开门，把小莫奈最喜欢的那个毛巾垫在里面。把一个玩具隧道对着侧面门口，用绳子逗他穿过隧道进猫包。后来我就发现它有时候晚上会在里面睡觉。另外我还做了一些演习，用这个包把他带上年，开车到不远的公园里，下车拎着转了一圈，然后回家。传说这样他会更加信任这个包，知道进了这个包不会去兽医那里，也不会去其它什么可怕的地方 :P



我还以为经过这一番训练，把小莫奈放进去应该很顺利了，结果出发当天却花了差不多半个小时才把他骗进去！我从来没有花这么长时间才把他放进包里，看来他意识到将要发生什么了…… :P

加上 UA 那傻瓜耽误了好些时间，所以经验之谈，请比普通国际航班的准备时间提前再多至少一个小时！

我选的是靠窗的座位，事实证明这是最好的选择。这包横着放在座位前面，会挡住中间座位乘客大约 10 厘米的放脚空间。



(请忽略右边的蓝色外套，它并没有占空间，猫包的右侧已经抵住前面座位的支撑架了。)

在起飞之前我很客气的跟邻座的人表示抱歉，并告诉她起飞之后我会把这个包竖着放，这样就不占她放脚的地方了。起飞之后，我把包竖过来，两只脚分在两边，就这样凑合了 14 个小时。



过安检

带着宠物如何过安检呢？很奇葩的经历。美国机场似乎怕你在宠物里面塞了什么东西，所以你不能简单的抱着他通过安检门了事。

1. 你把其他东西都放到传送带上，然后把猫从猫包上面那个门抱出来，让机场工作人员帮忙把猫包送上传送带。注意这个时候抱猫不能像平时那样“摇篮抱法”，得用拎着他两只前腿背对着你的“安全抱法”。否则我不知道他可能会做出什么事情来:P
2. 猫身上不需要有 harness 或者绳子。有些网站叫你买那种不带金属的 TSA 专用 harness，事实证明，机场没有人要求猫身上有 harness 和绳子。这多不符合猫的权益，对吧？
3. 安检人员会叫你到一个专用的小门，而不是那个一般人通过的门。
4. 走过那道门，你会遇到特殊的安检过程。安检人员叫我抱着猫，举起左手，然后他拿一个纸条在我手上画了一个“丰”字。然后要我举起右手，在我右手上也画个“丰”字。把纸条塞进一个机器，等了一会，然后就放我过了。

在举起一只手的过程中，小莫奈忽然惊恐地往我肩膀上爬，害得我不得不弯腰把它放下去，按在地上重新抱好，才举起右手。结果他又往我肩膀上爬……幸好他没用爪子爬，不然我就惨了。非常尴尬的经历 😅

过了安检门，请安检人员帮我把猫包放在地上。她主动把上面的门给我关好，打开了前面的门。我还担心小莫奈死活不进去呢，结果他溜溜就窜进去了。这证明我平时的“训练”是有效的，猫包是他在机场唯一可以信任的地方，所以直接就进去了，毫无阻力 :P

这时候登机口的人来电话了，问我在哪里。我说刚过了安检，她叫我赶快，就要起飞了！收拾好自己的东西，拎着猫包往登机口冲刺，气喘吁吁上了飞机……

食物，水和镇定剂

看了不少攻略说起飞前 4 小时要给断粮断水，还有人给猫吃了安眠药。当我问我的兽医，该如何处理食物和水的问题，他对我说：“哦对也，你得先让他吃饱喝足了。因为航空公司可能不让你带食物和水上飞机，或者你也许可以悄悄带一些猫粮上去。不过他可能会很紧张，不会吃任何东西也不会喝水……”

我说：“我的意思是……我看网上的文章都说要提前 4 小时断粮断水，免得他在飞机上随地大小便……”

兽医说：“那不应该是你关心的问题。你应该关心的是动物的健康。你的飞机 14 个小时，如果你提前 4 小时断粮断水，一只宠物那么长时间不吃东西也许可以，不喝水是很不好的！另外，他是一只猫，他在飞机上会紧张，就算想要大小便也会尽量憋着的。”

我还问了关于安眠药和镇定剂的事情。兽医也建议什么药都不要给他，因为无法预料这些药物在长时间的飞行途中会引起什么不好的后果。

经过考虑，我最后按照兽医的建议做了。我没有提前给小莫奈断水断粮，反而带了一些猫粮和猫零食上飞机。乘务员送水的时候，我还放了一个矿泉水瓶盖的水，捎进他的猫包里。因为紧张，他只吃了一点点猫粮，也许喝了一点点水吧。

不过整个飞行途中，小莫奈都没有大小便，憋得稳妥妥的 :)

飞机上的故事

跟小莫奈上了飞机，安顿好之后，他在猫包里很淡定。旁边的美国大叔还在夸奖他呢，说真是一只安静的好猫咪 :) 我隔一段时间还会把上面的门打开一个小口，伸手进去抚摸安慰他。就这样过了差不多十个小时。

可是过了十个小时之后，他忽然七窍八供起来，想从里面出来，而且开始不停地叫唤。我还以为他想大小便了，憋不住了，叫来乘务员问可不可以带他去洗手间。乘务员说：“猫不喜欢长时间待在那里的，所以他应该只是不耐烦了。你可以带他去洗手间，但你不可以把它放出来，否则你就再也没法让他进去了！”她还说这架飞机上曾经有一只猫从包里出来，就在机舱里到处跑，再也没法把它抓进去……

又出现几次不满之后，我把他带去卫生间看了看。没事，没有大小便，只是把我放进去的水给拱翻了。用卫生纸擦了，回座位，把猫包放在膝盖上抱了一会，他貌似安静了下来，后来就睡着了……

后来又不安分了几次…… 又过了四个小时，我们终于到了成都 :)

DSL 的误区

DSL 时不时地会成为一个话题，所以今天想专门说一下。

DSL 也就是 Domain Specific Language 的简称，是指为特定领域（domain）设计的专用语言。举个例子，Linux 系统下有很多配置文件，每个配置文件格式都不大一样，它们可以被看成是多种 DSL。IP Tables 的规则是一种 DSL，FVWM 窗口管理器的配置文件是一种 DSL，VIM 和 Emacs 的配置文件，当然也是 DSL。Makefile 是 DSL。CSS 是 DSL。JSON 是 DSL。SQL 也可以被看成是数据库领域的 DSL。也有很多人在自己的工作中创造 DSL，用它们来解决一些实际问题。

由于自己的原则，我个人从来没有设计过 DSL，但我用过别人设计的 DSL，并且对此深有感受。我觉得人们对于 DSL 有挺多的误解，所以我今天想分享一下自己对 DSL 的看法和亲身经历。

DSL 与库代码（Library）

开门见山说说对 DSL 的看法吧。我觉得大部分 DSL 都是不应该存在的，我们应该尽量避免创造 DSL。这一论点不但适用于只有少量用户的产品内部 DSL，也适用于像 SQL 这样具有大量从业者的 DSL。

DSL 这名字本身就是一种误导，它让人误以为不同“领域”（domain）的人就该用不同的语言，而其实不是那样的。这不过是在制造领域壁垒，为引入不必要的 DSL 找借口。绝大部分所谓不同“领域”，它们对语言的基本需求都是一样的。很多时候人们误以为需要新的 DSL，是因为他们没有分清“库代码”（library）和“新语言”（language）的差别。

不同领域需要的，绝大部分时候只是针对该领域写出的“库代码”，而不是完全不同的“新语言”。分析大部分所谓 DSL，你会发现它们不过提取了通用程序语言里的一部分，比如结构定义，算术表达式，逻辑表达式，条件语句，等等。极少有 DSL 是不能用通用的程序语言构造表示的。绝大部分时候你都可以用一种通用的语言，写出满足领域需求的库代码，然后领域里的人就可以调用库函数来完成他们的任务。

绝大部分 DSL 的存在，都是因为设计它的人没有理解问题的本质，没有意识到这问题并不需要通过设计新的语言来解决。很多人设计 DSL，是因为看到同类产品里面有 DSL，所以就抄袭照搬。或者因为听说 DSL 很酷，设计出 DSL 会显得自己很厉害，很有价值。同时，设计 DSL 还可以让同事和公司对自己产生依赖性。因为有人用我的 DSL，所以公司需要我，离不开我，那么 job security 就有所保证；）

然而如果你仔细分析手头的问题，就会发现它们绝大部分都可以用库代码，利用已有的语言来解决。就算类似的产品里面实现了 DSL，你会发现它们绝大部分也可以用库代码来代替。在自己的工作中，我一般都首先考虑写库代码来解决问题，实在解决不了才会考虑创造 DSL。

因为遵循这一原则，加上对问题透彻的理解，我发现自己几乎每次都能用库代码解决问题，所以我从来没有在自己的职业生涯中创造过 DSL。

“新语言问题”（The New Language Problem）

现在我来讲一下，盲目创造 DSL 带来的问题。很多人不明白 DSL 跟库代码的区别，拿到一个问题也不想清楚，就一意孤行开始设计 DSL，后来却发现 DSL 带来了严重的问题。由于 DSL 是一种新的语言，而不只是用已有语言写出来新函数，所以 DSL 必须经过一个学习和理解的过程，才能被其他人使用。

举个例子。如果你看到 `foo(x, y + z)` 这样的库代码，很显然这是一个函数调用，所以你知道它会先计算 $y+z$ ，得到结果之后，把它传递给 `foo` 函数作为参数，最后得到 `foo` 函数算出来的结果。注意到了吗，你并不需要学习新的语言。虽然你不知道 `foo` 函数的定义，然而你很清楚函数调用会做什么：把参数算好放进去，返回一个结果。也就是说，你对函数调用已经有一个“心理模型”。

可是一个 DSL 就很不一样，对于一个新的 DSL 构造，你也许没有任何心理模型存在。同样看到 `foo(x, y + z)`，它的含义也许根本不是一个函数调用。也许 `foo` 在这个 DSL 里就表示 `foreach` 循环语句，那么 `foo(x, y + z)` 表示类似 Java 的 `foreach (x : y + z)`，其中 `y` 和 `z` 都是链表，`+` 号表示连接两个链表。

这样一来，为了理解 `foo(x, y + z)` 是什么意义，你不能直接通过已有的，关于函数的心理模型，而必须阅读 DSL 设计者给你的文档，重新学习。如果 DSL 设计者是有素养的语言专家，那也许还好说。然而我发现绝大部分 DSL 设计者，都没有受到过专业的训练，所以他们设计出来的语言，从一开始就存在各种让人头痛的问题。

有些 DSL 表达力太弱，所以很多时候用户发现没法表达自己的意思。每当需要用这 DSL 写代码，他们就得去请教这个语言的设计者。很多时候你必须往这个 DSL 添加新的特性，才能解决自己的问题。到后来，你就发现有人设计了个 DSL，到头来他自己是唯一会用这 DSL 的人。每当有人需要用一个语言，就得去麻烦它的作者，那么这个语言的存在还有什么意义？

当然，很多 DSL 还会犯下程序语言设计的一些常见问题。很多人把设计语言想得太容易，喜欢耍新花样，到后来就因此出现各种麻烦事。容易出错，产生歧义，语法丑陋繁琐，难学难用，缺乏编辑器 IDE 支持，出错信息难以理解，无

法用 debugger 调试，等等。最后你发现还不如不要设计新的语言，使用已有的语言来解决问题就可以了。

最强大的 DSL 实现语言

有些人很崇拜 Haskell 或者 Scala，说这两个语言有着非常强大的“DSL 实现能力”，也就是说你可以用它们来实现自己想要的 DSL。这是一种误解。虽然我已经指出创造 DSL 并不是什么好事，我觉得还是应该把这个问题说清楚。如果你跟我一样看透了各种语言，就会发现世界上最强大的 DSL 实现语言，并不是 Haskell 或者 Scala，而是 Scheme。

2012 年的时候，我参加了 POPL 会议（Principles of Programming Languages），这是程序语言界的顶级会议。虽然名字里面含有 principle（原理）这个词，明眼人都看得出来，这个会议已经不是那么重视根本性的“原理”，它已经带有随波逐流的商业气息。那时候 Scala 正如日中天，所以在那次会议上，Scala 的 paper 简直是铺天盖地，“Scala 帮”的人趾高气昂。当然，各种 JavaScript 的东西也是如火如荼。

很多 Scala 人宣讲的主题，都是在鼓吹它的 DSL 实现能力。听了几个这样的报告之后，我发现 Scala 的 DSL 机制跟 Haskell 的挺像，它们不过是实现了类似 C++ 的“操作符重载”，利用特殊的操作符来表达对一些特殊对象的操作，然后把这些操作符美其名曰为“DSL”。

如果你还没看明白 Haskell 的把戏，我就提醒你一下。Haskell 的所谓 type class，其实跟 Java 或者 C++ 的函数重载（overloading）本质上是一回事。只不过因为 Haskell 采用了 Hindley-Milner 类型系统，这个重载问题被复杂化，模糊化了，所以一般人看不出来。等你看透了就会发现，Haskell 实现 DSL 的方式，不过是通过 type class 重载一些特殊操作符而已。这跟 C++ 的 operator+(...) 并没有什么本质区别。

操作符重载定义出来的 DSL，是非常有局限性的。实际上，通过重载操作符定义出来的语言，并不能叫做 DSL，而只能叫做“库代码”。为什么呢？因为一个语言之所以成为“语言”，它必须有自己独特的语义，而不只是定义了新的函数。重载操作符本质上只是定义了新的函数，而没有扩展语言的能力。就像你在 C++ 里重载了 + 操作符，你仍然是在使用 C++，而不是扩展了 C++ 的语义。

我用过 Haskell 实现的一个用于 GPU 计算的“DSL”，名叫 Accelerate。这个“语言”用起来相当的蹩脚，它要求用户在代码的特定位置写上一些特殊符号，因为只有这样操作符重载才能起作用。可是写上这些莫名其妙的符号之后，你就发现代码的可读性变得很差。但由于操作符重载的局限性，你必须这样做。你必须记住在什么时候必须写这些符号，在什么时候不能写它们。这种要求对于程序员的头脑，是一个严重的负担，没有人愿意去记住这些不知所以的东西。

由于操作符重载的局限性，Haskell 和 Scala 实现的 DSL，虽然吹得很厉害，发了不少 paper，却很少有人拿来实用。

世界上最强大的 DSL 实现语言，其实非 Scheme 莫属。Scheme 的宏系统（hygienic macro）超越了早期 Lisp 语言的宏系统，它本来就是被设计来改变和扩展 Scheme 的语义的。Scheme 的宏实质上是在对“语法树”进行任意变换，扩展编译器的功能，所以你可以利用宏把 Scheme 转变成几乎任何你想要的语言。这种宏系统不但可以实现 Haskell 和 Scala 的“重载型 DSL”，还能实现那些不能用重载实现的语言特性（比如能绑定变量的语句）。

miniKanren 就是一个用 Scheme 宏实现的语言，它是一个类似 Prolog 的逻辑式语言。如果你用 Haskell 或者 Scala 来实现 miniKanren，就会发现异常的困难。就算实现了，你的 DSL 语法也会很难看难用，不可能跟 miniKanren 一样优雅。

我并不是在这里鼓吹 Scheme，搞宣传。正好相反，对 Scheme 的宏系统有了深入理解之后，我发现了它带来的严重问题。内行人把这个问题称为“新语言问题”（The New Language Problem）。

因为在 Scheme 里实现一个新语言如此的容易，几行代码就可以写出新的语言构造，改变语言本来的语义，所以这带来了严重的问题。这个问题就是，一旦你改变了语言的语义，或者设计出新的语言构造，人们之间的交流就增加了一道障碍。使用你改造后的 Scheme 的人，必须学习一种新的语言，才能看懂你的代码，才能跟你交流。

由于这个原因，你很难看懂另一个人的 Scheme 代码，因为很多 Scheme 程序员觉得宏是个好东西，所以很喜欢用它。他们设计出稀奇古怪的宏，扩展语言的能力，然后使用扩展后的，你完全不理解的语言来写他的代码。本来语言是用来方便人与人交流的，结果由于每个人都可以改变这语言，导致他们鸡同鸭讲，没法交流！

再次声明，我不是在这里称赞或者宣扬 Scheme，我真的认为宏系统的存在是 Scheme 的一个严重的缺点。我那热爱 Scheme 的教授们知道了，一定会反对我这种说法，甚至鄙视我。但我确实就是这么想的，这么多年过去了，仍然没有改变过这一看法。

Scheme 宏系统的这个问题，引发了我对 DSL 的思考。后来我发现所谓 DSL 跟 Scheme 宏系统，存在几乎一模一样的问题。这个问题有一个名字，叫做“新语言问题”（The New Language Problem）。下面我详细解释一下这个问题。

NaCl 的故事

现在我来讲一个有趣的故事，是我自己跟 DSL 有关的经历。

在我曾经工作过的某公司，有两个很喜欢捣鼓 PL，却没有受过正规 PL 教育的人。说得不好听一点，他们就是“PL 民科”。然而正是这种民科，最喜欢显示自己牛逼，喜欢显示自己有能力实现新的语言，以至于真正的专家只好在旁边静静地看着他们装逼 :P

他们其中一个人知道我是研究 PL 的，开头觉得我是同类，所以总喜欢走到桌前对我说：“咱们一起设计一个通用程序语言吧！然后用它来解决我们公司现在遇到的难题！”每当他这样说，我都安静的摇摇头：“我们公司真的需要一个新的语言吗？你有多少时间来设计和实现这个语言？”

当时这两个人在公司里，总是喜欢试用各种新语言，Go 语言，Scala，Rust，…… 他们都试过了。每当拿到一个新的项目，他们总是想方设法要用某种新语言来做。于是乎，这样的历史就在我眼前反复的上演：

1. 为一种新语言兴奋，开始用它来做新项目
2. 两个月之后，开始骂这语言，各种不爽
3. 最后项目不了了之，代码全部丢进垃圾堆
4. Goto 1

这两个家伙每天就为这些事情忙得不亦乐乎，真正留下来的产出却很少。之前他们还设计了一种 DSL，专门用于对 HTML 进行匹配和转换。这个 DSL 被他们起了一个很有科学味道的名字，叫做 NaCl（氯化钠，食盐的化学分子式）。

我进公司的时候，NaCl 已经存在了挺长一段时间，然而很少有人真正理解它的用法，大部分人对它的态度都是“能不碰就不碰”。终于有一天，我遇到了需要修改 NaCl 代码的时候。也就一行代码，看了半天 NaCl 的“官方文档”，却不知道如何才能用它提供的语法，来表达我所需要的改动。其实我需要的不过是一个很容易的匹配替换，完全可以用正则表达式来完成，可是已有的代码是用 NaCl 写的，再加上好几层的框架，让你绕都绕不过，所以我不知道怎么办了。

问了挺多人，包括公司里最顶级的“NaCl 专家”，都没能得到结果。最后，我不得不硬着头皮去打扰两位日理万机的“NaCl 之父”。叽里呱啦跟我解释说教了一通之后，眨眼之间噼里啪啦帮我改了代码，搞定了！其实我根本没听明白他在说什么，为什么那样改，也不知道背后的原理。总之，我一个字都没打，目的就达到了，所以我就回去做自己的事情了。

后来跟其他同事聊，发现我的直觉是很准的。他们告诉我，公司里所有 NaCl 代码可以表达的东西，都可以很容易的用正则表达式替换来解决，甚至可以用硬邦邦的，不带 regexp 的字符串替换来解决。同事们都很不理解，为什么非得设计个 DSL 来做这么简单的事情，本来调用 Java 的 String.replace 就可以完成。

后来“NaCl 专家”告诉我，虽然他很了解 NaCl，却根本不喜欢它。在那两个家伙提出要设计 NaCl 的时候，他就已经表示了强烈的反对，他觉得不应该创造 DSL 来解决这样的问题。当时他就给大家解释了什么是“新语言问题”，警告大家新语言会带来的麻烦。可是领导显然跟这两个家伙有某种政治上的联盟关系，所以根本没听他在说什么。

在领导的放任和支持下，这两个家伙一意孤行创造了 NaCl，然后强行在全公司推广。到后来，每次需要用 NaCl 写点什么，就发现需要给它增加新的功能，就得去求那两个家伙帮忙。所以我能用上今天的 NaCl，基本能表达我想要的东西，还多亏了这位“NaCl 专家”以前栽的跟头，他把各种坑基本给我填起来了 ;)

我有一句格言：如果一个语言，每当用户需要用它表达任何东西，都得去麻烦它的设计者，甚至需要给这个语言增加新的功能，那这个语言就不应该存在。NaCl 这个 DSL 正好符合了我的断言。

当然 NaCl 只是一个例子，我知道很多其它 DSL 的背后都有类似的故事。几个月之后，这两个民科又开始创造另一个 DSL，名叫 Dex，于是历史又开始重演……

动态逻辑加载

Dex 的故事跟 NaCl 有所不同，但最后的结果差不多。NaCl 是一个完全不应该存在的语言，而 Dex 的情况有点不一样。我们确实需要某种“嵌入式语言”，只不过它不应该是 Dex 那个样子，不应该是一个 DSL。由于 Dex 要解决的需求有一定的代表性，很多人在遇到这类需求的时候，就开始盲目的创造 DSL，所以这是一个很大的坑！我想把这个故事详细讲一下，免得越来越多的人掉进去。

原来的需求是这样：产品需要一种配置方式，配置文件里面可以包含一定的“逻辑”。通过在不更换代码的情况下动态加载配置文件，它可以动态的改变系统的逻辑和行为。这东西有点像“防火墙”的规则，比如：

1. 如果尺寸大于 1000，那么不通过，否则通过。
2. 如果标题含有“猪头”这个词，不通过，否则通过……

这些规则从本质上讲，就是一些逻辑表达式“size > 1000”，加上一些分支语句“if ... then ...”。在 Dex 出现之前，有人用 XML 定义这样的规则，后来发现 XML 非常不好理解，像是这个样子：

```
<rule>
  <condition>
    <operator>gt</operator>
    <first>size</first>
```

```
<second>1000</second>
</condition>
<action>block</action>
</rule>
```

看明白了吗？这个看得人眼睛发涨的 XML，表达的不过是普通语言里面的 `if (size > 1000) block()`。为了理解这一点，你可以把这个 XML 所表示的“数据结构”，想象成编译器里面的“抽象语法树”（AST）。所以写这个 XML，其实是在用手写 AST，那当然是相当痛苦的。

那我们为什么不把 `if (size > 1000) block()` 这条语句直接写到系统的 Java 代码里面呢？因为 Java 代码是编译之后放进系统里面的，一旦放进去就不能随时换掉了。然而我们需要可以随时的，“动态”的替换掉这块逻辑，而不更新系统代码。所以你不能把这条 Java 语句“写死”到系统代码里面，而必须作为“配置”。

想清楚了这一点，你就自然找到了解决方案：把 `if (x > 1000) block()` 这样的 Java 代码片段写到一个“配置文件”里，然后使用 JVM 读取，组合，并且编译这个文件，动态加载生成的 class，这样系统的行为就可以改变了。实际上这也就是公司里另外一个团队做过的事情，让用户编辑一个基于 Java 的“规则文件”，然后加载它。

我觉得这不失为一个可行的解决方案。为了实现动态逻辑加载，你完全可以用像 Java 或者 JavaScript 那样已有的语言，利用已有的编译器来达到这个目的，而不需要设计新的语言。然而当 PL 民科们遇到这样的问题，他们首先想到的是：设计一个新的 DSL！于是 Dex 就诞生了。

Dex 要表达的东西，本质上就是这些逻辑表达式和条件语句，然而 Dex 被设计为一个完全独立的语言。它的语法被设计得其它语言很不一样，结合了 Haskell，Go 语言，Python 等各种语言语法的缺陷。后来团队里又进来一个研究过 Prolog 逻辑式语言的人，所以他试图在里面加入“逻辑式编程”的元素。总之他们有很宏伟的目标：让这个 DSL “可靠”，“可验证”，成为“描述式语言”……

他们向团队宣布这个雄心勃勃的计划之后，一个有趣的插曲发生了。听说又要创造一个 DSL，“NaCl 专家”再次怒发冲冠，开始反对这个计划。这一次他拿出了实际行动，自己动手拿 Java 内嵌的 [JavaScript 解释器](#)，做了一个基于 JavaScript 的动态配置系统，只开发了一个星期就可以用了。

我觉得用 JavaScript 也不失为一个解决方案，毕竟浏览器的 PAC 文件就是用 JavaScript 定义代理规则的，而这种代理规则跟我们的应用非常类似。我虽然没有特别喜欢 JavaScript，但它其中的一些简单构造用在这种场景，是没什么大问题的。

其实在此之前我也看不下去了，所以自己悄悄做了一个类似的配置系统，拿已有的 JavaScript parser，提取 JavaScript 的相关构造，做了一个解释器，嵌入到系统里，只花了一天时间。但我心里很清楚，一切技术上的努力在政治斗争的面前，都是无足轻重的。我早已经伤不起了，在好心人的帮助下，我离开了这个团队，但暗地里我仍然从精神上支持着“NaCl 专家”继续他的抗争。

争吵的最后结果，当然是由于领导偏心庇护，否决了“外人”的作法，让两个民科和一个 Prolog 狂人继续开发 Dex。几个月之后，公司的第二个奇葩 DSL 诞生了。它用混淆难读的方式，表达了普通语言里的条件语句和逻辑表达式。他们为它写了 parser，写了解释器，写了文档，开始在公司强行推广。“可靠”，“可验证”，“描述式”的目标，早已被抛到九霄云外。用的人都苦不堪言，好多东西没法表达或者不知道如何表达，出错了也没有足够的反馈信息，每次要写东西就得去找“Dex 之父”们。

嗯，历史就这样重演了……

结论

所以，我对于 DSL 的结论是什么呢？

1. 尽一切可能避免创造 DSL，因为它会带来严重的理解，交流和学习曲线问题，可能会严重的降低团队的工作效率。如果这个 DSL 是给用户使用，会严重影响用户体验，降低产品的可用性。
2. 大部分时候写库代码，把需要的功能做成函数，其实就可以解决问题。
3. 如果真的到了必须创造 DSL 的时候，非 DSL 不能解决问题，才可以动手设计 DSL。但 DSL 必须由程序语言专家来完成，否则它还是可能给产品和团队带来严重的后果。
4. 大部分 DSL 要解决的问题，不过是“动态逻辑加载”。为了这个目的，你完全可以利用已有的语言（比如 JavaScript），或者取其中一部分构造，通过动态调用它的解释器（编译器）来达到这个目的，而不需要创造新的 DSL。

（本文建议零售价 ¥30，如果它让你的团队或者公司幸免落坑，请付款 ¥1000000：）

Kotlin 和 Checked Exception

最近 JetBrains 的 Kotlin 语言忽然成了热门话题。国内小编们传言说，Kotlin 取代了 Java，成为了 Android 的“钦定语言”，很多人听了之后热血沸腾。初学者们也开始注意到 Kotlin，问出各种“傻问题”，很“功利”的问题，比如“现在学 Kotlin 是不是太早了一点？”结果引起一些 Kotlin 老鸟们的鄙视。当然也有人来信，请求我评价 Kotlin。

对于这种评价语言的请求，我一般都不予理睬的。作为一个专业的语言研究者，我的职责不应该是去评价别人设计的语言。然而浏览了 Kotlin 的文档之后，我发现 Kotlin 的设计者误解了一个重要的问题——关于是否需要 checked exception。对于这个话题我已经思考了很久，觉得有必要分享一下我对此的看法，避免误解的传播，所以我还是决定写一篇文章。

可以说我这篇文章针对的是 checked exception，而不是 Kotlin，因为同样的问题也存在于 C# 和其它一些语言。

冷静一下

在进入主题之前，我想先纠正一些人的误解，让他们冷静下来。我们首先应该搞清楚的是，Kotlin 并不是像有些国内媒体传言的那样，要“取代 Java 成为 Android 的官方语言”。准确的说，Kotlin 只是得到了 Android 的“官方支持”，所以你可以用 Kotlin 开发 Android 程序，而不需要绕过很多限制。可以说 Kotlin 跟 Java 一样，都是 Android 的官方语言，但 Kotlin 不会取代 Java，它们是一种并存关系。

这里我不得不批评一下有些国内技术媒体，他们似乎很喜欢片面报道和歪曲夸大事实，把一个平常的事情吹得天翻地覆。如果你看看国外媒体对 Kotlin 的[报道](#)，就会发现他们用词的迥然不同：

Google's Java-centric Android mobile development platform is adding the Kotlin language as an officially supported development language, and will include it in the Android Studio 3.0 IDE.

译文：Google 的以 Java 为核心的 Android 移动开发平台，加入了 Kotlin 作为官方支持的开发语言。它会被包含到 Android Studio 3.0 IDE 里面。

看明白了吗？不是“取代了 Java”，而只是给了大家另一个“选择”。我发现国内的技术小编们似乎很喜欢把“选择”歪曲成“取代”。前段时间这些小编们也有类似的谣传，说斯坦福大学把入门编程课的语言“换成了 JavaScript”，而其实别人只是另外“增加”了一门课，使用 JavaScript 作为主要编程语言，原来以 Java 为主的入门课并没有被去掉。我希望大家在看到此类报道的时候多长个心眼，要分清楚“选择”和“取代”，不要盲目的相信一个事物会立即取代另一个。

Android 显然不可能抛弃 Java 而拥抱 Kotlin。毕竟现有的 Android 代码绝大部分都是 Java 写的，绝大部分程序员都在用 Java。很多人都知道 Java 的好处，所以他们不会愿意换用一个新的，未经时间考验的语言。所以虽然 Kotlin 在 Android 上得到了和 Java 平起平坐的地位，想要程序员们从 Java 转到 Kotlin，却不是一件容易的事情。

我不明白为什么每当出现一个 JVM 的语言，就有人欢呼雀跃的，希望它会取代 Java，似乎这些人跟 Java 有什么深仇大恨。他们已经为很多新语言热血沸腾过了，不是吗？Scala，Clojure……一个个都像中国古代的农民起义一样，煽动一批人起来造反，而其实自己都不知道在干什么。Kotlin 的主页也把“drastically reduce the amount of boilerplate code”作为自己的一大特色，仿佛是在暗示大家 Java 有很多“boilerplate code”。

如果你经过理性的分析，就会发现 Java 并不是那么的讨厌。正好相反，Java 的有些设计看起来“繁复多余”，实际上却是经过深思熟虑的决定。Java 的设计者知道有些地方可以省略，却故意把它做成多余的。不理解语言“可用性”的人，往往盲目地以为简短就是好，多写几个字就是丑陋不优雅，其实不是那样的。关于 Java 的良好设计，你可以参考我之前的[文章](#)《[为 Java 说句公道话](#)》。另外在《[对 Rust 语言的分析](#)》里面，我也提到一些容易被误解的语言可用性问题。我希望这些文章对人们有所帮助，避免他们因为偏执而扔掉好的东西。

实际上我很早以前就发现了 Kotlin，看过它的文档，当时并没有引起我很大的兴趣。现在它忽然火了起来，我再次浏览它的新版文档，却发现自己还是会继续使用 Java 或者 C++。虽然我觉得 Kotlin 比起 Java 在某些小地方设计相对优雅，一致性稍好一些，然而我并没有发现它可以让我兴奋到愿意丢掉 Java 的地步。实际上 Kotlin 的好些小改进，我在设计自己语言的时候都已经想到了，然而我并不觉得它们可以成为人们换用一个新语言的理由。

Checked Exception (CE) 的重要性

有几个我觉得很重要的，具有突破性的语言特性，Kotlin 并没有实现。另外我还发现一个很重要的 Java 特性，被 Kotlin 的设计者给盲目抛弃了。这就是我今天要讲的主题：checked exception。我不知道这个术语有什么标准的中文翻译，为了避免引起定义混乱，下文我就把它简称为“CE”好了。

先来科普一下 CE 到底是什么吧。Java 要求你必须在函数的类型里面声明它可能抛出的异常。比如，你的函数如果是这样：

```

void foo(string filename) throws FileNotFoundException
{
    if (...)

    {
        throw new FileNotFoundException();
    }

    ...
}

```

Java 要求你必须在函数头部写上“throws FileNotFoundException”，否则它就不能编译。这个声明表示函数在某些情况下，会抛出 FileNotFoundException 这个异常。由于编译器看到了这个声明，它会严格检查你对 foo 函数的用法。在调用 foo 的时候，你必须使用 try-catch 处理这个异常，或者在调用的函数头部也声明“throws FileNotFoundException”，把这个异常传递给上一层调用者。

```

try
{
    foo("blah");
}
catch (FileNotFoundException e)
{
    ...
}

```

这种对异常的声明和检查，叫做“checked exception”。很多语言（包括 C++，C#，JavaScript，Python……）都有异常机制，但它们不要求你在函数的类型里面声明可能出现的异常类型，也不使用静态类型系统对异常的处理进行检查和验证。我们说这些语言里面有“exception”，却没有“checked exception”。

理解了 CE 这个概念，下面我们来谈正事：Kotlin 和 C# 对 CE 的误解。

[Kotlin 的文档](#)明确的说明，它不支持类似 Java 的 checked exception (CE)，指出 CE 的缺点是“繁琐”，并且列举了几个普通程序员心目中“大牛”的文章，想以此来证明为什么 Java 的 CE 是一个错误，为什么它不解决问题，却带来了麻烦。这些人包括了 Bruce Eckel 和 C# 的设计者 [Anders Hejlsberg](#)。

很早的时候我就看过 Hejlsberg 的这些言论。他的话看似有道理，然而通过自己编程和设计语言的实际经验，我发现他并没有抓住问题的关键。他的论述里有好几处逻辑错误，一些自相矛盾，还有一些盲目的臆断，所以这些言论并不能说服我。正好相反，实在的项目经验告诉我，CE 是 C# 缺少的一项重要特性，没有了 CE 会带来相当麻烦的后果。在微软写 C# 的时候，我已经深刻体会到了缺少 CE 所带来的困扰。现在我就来讲一下，CE 为什么是很重要的语言特性，然后讲一下为什么 Hejlsberg 对它的批评是站不住脚的。

首先，写 C# 代码时最让我头痛的事情之一，就是 C# 没有 CE。每调用一个函数（不管是标准库函数，第三方库函数，还是队友写的函数，甚至我自己写的函数），我都会疑惑这个函数是否会抛出异常。由于 C# 的函数类型上不需要标记它可能抛出的异常，为了确保一个函数不会抛出异常，你就需要检查这个函数的源代码，以及它调用的那些函数的源代码……

也就是说，你必须检查这个函数的整个“调用树”的代码，才能确信这个函数不会抛出异常。这样的调用树可以是非常大的。说白了，这就是在用人工对代码进行“全局静态分析”，遍历整个调用树。这不但费时费力，看得你眼花缭乱，还容易漏掉出错。显然让人做这种事情是不现实的，所以绝大部分时候，程序员都不能确信这个函数调用不会出现异常。

在这种疑虑的情况下，你就不得不做最坏的打算，你就得把代码写成：

```

try
{
    foo();
}
catch (Exception)
{
    ...
}

```

注意到了吗，这也就是你写 Java 代码时，能写出的最糟糕的异常处理代码！因为不知道 foo 函数里面会有什么异常出现，所以你的 catch 语句里面也不知道该做什么。大部分人只能在里面放一条 log，记录异常的发生。这是一种非常糟糕的写法，不但繁复，而且可能掩盖运行时错误。有时候你发现有些语句莫名其妙没有执行，折腾好久才发现是因为某个地方抛出了异常，所以跳到了这种 catch 的地方，然后被忽略了。如果你忘了写 catch (Exception)，那么你的代码可能运行了一段时间之后当掉，因为忽然出现一个测试时没出现过的异常……

所以对于 C# 这样没有 CE 的语言，很多时候你必须莫名其妙这样写，这种做法也就是我在微软的 C# 代码里经常看到的。问原作者为什么那里要包一层 try-catch，答曰：“因为之前这地方出现了某种异常，所以加了个 try-catch，然后就忘了当时出现的是什么异常，具体是哪一条语句会出现异常，总之那一块代码会出现异常……”如此写代码，自己心虚，看的人也糊涂，软件质量又如何保证？

那么 Java 呢？因为 Java 有 CE，所以当你看到一个函数没有声明异常，就可以放心的省掉 try-catch。所以这个 C# 的问题，自然而然就被避免了，你不需要在很多地方疑惑是否需要写 try-catch。Java 编译器的静态类型检查会告诉你，在什么地方必须写 try-catch，或者加上 throws 声明。如果你用 IntelliJ，把光标放到 catch 语句上面，可能抛出那种异常的语句就会被加亮。C# 代码就不可能得到这样的帮助。

```
OutputStream symOut = null, refOut = null, docOut = null;
try {
    docOut = new BufferedOutputStream(new FileOutputStream(docFilename));
    symOut = new BufferedOutputStream(new FileOutputStream(symFilename));
    refOut = new BufferedOutputStream(new FileOutputStream(refFilename));
    $.msg( m: "graphing: " + srcpath);
    graph(srcpath, inclpaths, symOut, refOut, docOut);
    docOut.flush();
    symOut.flush();
    refOut.flush();
} catch (FileNotFoundException e) {
    System.err.println("Could not find file: " + e);
    return;
} finally {
    if (docOut != null) {
        docOut.close();
    }
}
```

CE 看起来有点费事，似乎只是为了“让编译器开心”，然而这其实是每个程序员必须理解的事情。出错处理并不是 Java 所特有的东西，就算你用 C 语言，也会遇到本质一样的问题。使用任何语言都无法逃脱这个问题，所以必须把它想清楚。在《[编程的智慧](#)》一文中，我已经讲述了如何正确的进行出错处理。如果你滥用 CE，当然会有不好的后果，然而如果你使用得当，就会起到事半功倍，提高代码可靠性的效果。

Java 的 CE 其实对应着一种强大的逻辑概念，一种根本性的语言特性，它叫做“union type”。这个特性只存在于 Typed Racket 等一两个不怎么流行的语言里。Union type 也存在于 PySonar 类型推导和 Yin 语言里面。你可以把 Java 的 CE 看成是对 union type 的一种不完美的，丑陋的实现。虽然实现丑陋，写法麻烦，CE 却仍然有着 union type 的基本功能。如果使用得当，union type 不但会让代码的出错处理无懈可击，还可以完美的解决 null 指针等头痛的问题。通过实际使用 Java 的 CE 和 Typed Racket 的 union type 来构建复杂项目，我很确信 CE 的可行性和它带来的好处。

现在我来讲一下为什么 Hejlsberg 对于 CE 的[批评](#)是站不住脚的。他的第一个错误，俗话说就是“人笨怪刀钝”。他把程序员对于出错处理的无知，不谨慎和误用，怪罪在 CE 这个无辜的语言特性身上。他的话翻译过来就是：“因为大部分程序员都很傻，没有经过严格的训练，不小心又懒惰，所以没法正确使用 CE。所以这个特性不好，是没用的！”

他的论据里面充满了这样的语言：

- “大部分程序员不会处理这些 throws 声明的异常，所以他们就给自己的每个函数都加上 throws Exception。这使得 Java 的 CE 完全失效。”
- “大部分程序员根本不在乎这异常是什么，所以他们在程序的最上层加上 catch (Exception)，捕获所有的异常。”
- “有些人的函数最后抛出 80 多种不同的异常，以至于使用者不知道该怎么办。”……

注意到了吗，这种给每个函数加上 throws Exception 或者 catch (Exception) 的做法，也就是我在《[编程的智慧](#)》里面指出的经典错误做法。要让 CE 可以起到良好的作用，你必须避免这样的用法，你必须知道自己在干什么，必须知道被调用的函数抛出的 exception 是什么含义，必须思考如何正确的处理它们。

另外 CE 就像 union type 一样，如果你不小心分析，不假思索就抛出异常，就会遇到他提到的“抛出 80 多种异常”的情况。出现这种情况往往是因为程序员没有仔细思考，没有处理本来该自己处理的异常，而只是简单的把下层的异常加到自己函数类型里面。在多层次调用之后，你就会发现最上面的函数累积起很多种异常，让调用者不知所措，只好传递这些异常，造成恶性循环。终于有人烦得不行，把它改成了“throws Exception”。

我在使用 Typed Racket 的 union type 时也遇到了类似的问题，但只要你严格检查被调用函数的异常，尽量不让它们传播，严格限制自己抛出的异常数目，缩小可能出现的异常范围，这种情况是可以避免的。CE 和 union type 强迫你仔细的思考，理顺这些东西之后，你就会发现代码变得非常缜密而优雅。其实就算你写 C 代码或者 JavaScript，这些问题同样是存在的，只不过这些语言没有强迫你去思考，所以很多时候问题被稀里糊涂掩盖了起来，直到很长一段时间之后才暴露出来，不可救药。

所以可以说，这些问题来自于程序员自己，而不是 CE 本身。CE 只提供了一种机制，至于程序员怎么使用它，是他们自己的职责。再好的特性被滥用，也会产生糟糕的结果。Hejlsberg 对这些问题使用了站不住脚的理论。如果你假设程序员都是糊里糊涂写代码，那么你可以得出无比惊人的结论：所有用于防止错误的语言特性都是没用的！因为总有人可以懒到不理解这些特性的用法，所以他总是可以滥用它们，绕过它们，写出错误百出的代码，所以静态类型没用，CE 没用，…… 有这些特性的语言都是垃圾，大家都写 PHP 就行了；）

Hejlsberg 把这些不理解 CE 用法，懒惰，滥用它的人作为依据，以至于得出 CE 是没用的特性，以至于不把它放到 C# 里面。由于某些人会误用 CE，结果就让真正理解它的人也不能用它。最后所有人都退化到最笨的情况，大家都只好写 catch (Exception)。在 Java 里，至少有少数人知道应该怎么做，在 C# 里，所有人都被迫退化成最差的 Java 程序员；）

另外，Hejlsberg 还指出 C# 代码里没有被 catch 的异常，应该可以用“静态分析”检查出来。可以看出来，他并不理解这种静态检查是什么规模的问题。要能用静态分析发现 C# 代码里被忽略的异常，你必须进行“全局分析”，也就是说为了知道一个函数是否会抛出异常，你不能只看这个函数。你必须分析这个函数的代码，它调用的代码，它调用

的代码调用的代码…… 所以你需要分析超乎想象的代码量，而且很多时候你没有源代码。所以对于大型的项目，这显然是不现实的。

相比之下，Java 要求你对异常进行 throws 显式声明，实质上把这个全局分析问题分解成了一个个模块化 (modular) 的小问题。每个函数作者完成其中的一部分，调用它的人完成另外一部分。大家合力帮助编译器，高效的完成静态检查，防止漏掉异常处理，避免不必要的 try-catch。实际上，像 [Exceptional](#) 一类的 C# 静态检查工具，会要求你在注释里写出可能抛出的异常，这样它才能发现被忽略的异常。所以 Exceptional 其实重新发明了 Java 的 CE，只不过 throws 声明被写成了一个注释而已。

说到 C#，其实它还有另外一个特别讨厌的设计错误，引起了很多不必要的麻烦。感兴趣的人可以看看我这篇文章：[《可恶的 C# IDisposable 接口》](#)。这个问题浪费了整个团队两个月之久的时间。所以我觉得作为 C# 的设计者，Hejlsberg 的思维局限性相当大。我们应该小心的分析和论证这些人的言论，不应该把他们作为权威而盲目接受，以至于让一个优秀的语言特性被误解，不能进入到新的语言里。

结论？

所以我对 Kotlin 是什么“结论”呢？我没有结论，这篇文章就像我所有的看法一样，仅供参考。显然 Kotlin 有的地方做得比 Java 好，所以它不会因为没有 CE 而完全失去意义。我不想打击人们对新事物的兴趣，我甚至鼓励有时间的人去试试看。

我知道很多人希望我给他们一个结论，到底是用一个语言，还是不用它，这样他们就不用纠结了，然而我并不想给出一个结论。一来是因为我不想让人感觉我在“控制”他们，如何看待一个东西是他们的自由，是否采用一个东西是他们自己的决定。二来是因为我还没有时间和机会，去用 Kotlin 来做实际的项目。另外，我早就厌倦了试用新的语言，如果一个大众化的语言没有特别讨厌，不可原谅的设计失误，我是不会轻易换用新语言的。我宁愿让其他人做我的小白鼠，去试用这些新语言。到后来我有空了，再去看看他们的成功或者失败经历 :P

所以对我个人而言，我至少现在不会去用 Kotlin，但我并不想让其他人也跟我一样。因为 Java，C++ 和 C 已经能满足我的需求，它们相当稳定，而且我对它们已经很熟悉，所以我为什么要花精力去学一个新的语言，去折腾不成熟的工具，放下我真正感兴趣的算法和数据结构等问题呢？实际上不管我用什么语言写代码，我的头脑里都在用同一个语言构造程序。我写代码的过程，只不过是在为我脑子里的“万能语言”找到对应的表达方式而已。

(本文建议零售价 ￥15)

什么是现实理想主义者

曾经有人看了我的文章，以为我是一个“理想主义者”，来找我聊天。他说：“你知道吗，我跟你一样喜欢简单优雅的代码。上次我在某公司工作，看到他们的代码乱得不成样子，二话没说给他们重写了，结果有几个小地方跟原来的代码不大一样，后来系统因此当掉了。老板对我说，明天你不用再来上班了！你说我是不是好心没好报啊？”

虽然我同情他丢了工作，然而我并不认同这种不经同意就推翻重写别人代码的作法。实际上我曾经跟一个老喜欢重写别人代码的人合作，后来整个团队（包括我）都差点被他给弄疯了。所以我对他说：“你不可以这样改别人的代码的！如果我是你老板，可能不会开掉你，却也会给你一个严重警告的。”

从我们的对话你也许已经发现了，我并不是一个通常人所谓的“理想主义者”。虽然我有很多新颖而美好的想法，然而它们全都深深植根于现实中。我反对不以现实为基础的“理想”，实际上那不叫理想，而只能叫做“空想”。我的直觉和理性会很快的告诉我，哪些事情是可能的，哪些是不大可能的。我往往在早期就能察觉和避免那些最终会失败的“理想主义作法”。

从我对各种“新语言”，“新理论”和“新技术”的看法，你也许已经发现了我的这个特点，我不再是十年前那个“热爱新奇事物”的王垠。不理解的人甚至会觉得我“守旧”，然而我只是通过理性分析，预见了某些“新技术”的失败。在我的心里，事物和技术并没有新旧之分，只有合理与不合理的差别。

如何对待别人的代码

那么我是如何对待别人的“垃圾代码”的呢？你也许会很惊讶我的做法：我尽量不动它们！

虽然我喜欢简单优雅的代码，然而对于别人写的代码，就算它再丑再乱，我也不不会乱动它。我就像一个外科专家，多次对已有代码进行“换心手术”。这种手术成功的要诀，是制造尽量小的“切口”，刚好可以换掉心脏，而不动其他部位。就算那些地方血管乱绕，堆满各种垃圾，也不要动它们。

这是为什么呢？因为代码首要的目标应该是“解决问题”（包括“没有 bug”），其次的目标才是“简单优雅”。如果不能解决问题，再优雅又有什么用呢，只不过是玩具而已。对于已经可以解决问题的代码，就算它再乱再复杂，我也是高度尊重的，绝对不敢像这个朋友一样，不假思索就删掉重写。这就像你给别人做换心手术，看到大腿上有些血管是乱的，就把大腿也切开倒腾，你的病人不死才怪呢。

我自己写代码的时候，“解决问题”和“简单优雅”往往是紧密结合，交织在一起的。如果我写不出简单优雅的代码，我就不能又快又正确的解决问题。所以我的代码往往从一开头就是简单优雅，模块化的。我从很小的函数开始写起，每个函数只解决很小的问题，最终我把它们组合在一起，解决掉整个问题。

对于别人的代码，情况就很不一样了。很多人写的代码很乱，很复杂，不易理解，看得我头痛，但由于他们在上面花了很多的时间，而且这些代码经过了很长时间的使用，大量现实情况的考验，所以它们已经算是解决了问题。对于这样的代码，我的经验是这样：如果把它删掉完全重写，是很难不犯原作者已经犯过的错误的。就算你自认为水平世界一流，写的代码极其简单和优雅，也不能避免犯错。

这不是一个智力的问题，而是一个智慧的问题。喜欢删掉别人代码重写的人，也许有很高的智力，却缺乏智慧。代码是用来解决现实问题的，而现实有许许多多的细节，代码需要覆盖现实世界各种不完美的地方。这些不完美也许来自库代码，也许来自操作系统，也许来自网络协议，也许来自用户习惯，也许来自自然界。我们必须承认，很多这些东西我们是没有能力，没有时间，也没有必要去改变的。

别人已经写好，用了几年的代码，很有可能已经遇到各种现实问题，各种边角情况，原来的作者虽然不像你一样思路清晰，却也为此付出了时间和精力。这些复杂混乱的代码逻辑里面，已经针对现实世界的不完美，做出了基本可行的解决方案。一个有智慧的人，必须能利用这些前人留下来的混乱代码，因为它包含了时间积累下来的财富。

那么我一般是如何利用别人遗留下来的代码的呢？我的策略包含好几个要点。

首先，我尽量保持别人的代码原封不动。因为别人的代码解决的问题，很可能不是我当前需要解决的问题。因为看不顺眼而去改别人的代码，不但分散自己的精力，而且有可能制造新的 bug，导致新老代码中同时多处出现 bug，难以追踪和修复。为了保持别人的代码原封不动，却又让自己写的新代码简单优雅，我必须理解原有代码的接口（interface），以及它原有的各种特征，我力求保持它们不变。这就像外科大夫做换心手术，他必须保证已有的血管都连接到正确的地方。

我喜欢把自己的代码做成一个可替换的，模块化的元件，可以随时在系统里插入或者移除。一旦发现了问题，我可以随时切换到原来的代码，重新测试，这样我就可以知道问题出在原来的代码，还是出在我的新代码里面。另外，我还会注意避免对已有函数进行换名，这样我可以把自己的修改局限在一个或者少数几个文件里面，避免 Git 的历史里面出现不必要的，让人分心的修改。就算要换名也应该单独作为 commit，而不应该跟逻辑的修改混在一起。

如果经过多次试验，我发现别人的代码的确需要改，不然我没法继续写新的代码，那么我只好对它进行修改。由于已有的代码复杂混乱，我一般会极其小心的对待它。我不会删掉大片的代码，从头开始写，那几乎注定是要失败的。通常我会先“隔离”出很小的一块代码，对它进行重写。随之立即进行大量的测试和试验，找原作者来帮我检查是否有问

题，如此反复……

那么这块改掉的代码需要小到什么程度呢？我也许就只改写一个 `for` 循环，把几行代码提出去做成帮助函数，简化一个表达式，把一个类成员变成一个局部变量，改几个局部变量的名字之类的。你可以参考我在《[编程的智慧](#)》里提到的各种改进代码的方式。每一个这样的小改动都有可能出错，所以在此之后必须进行严格的验证，确保修改后的代码和原来的代码语义相同。这样反反复复很多次之后，你才能正确的替换掉原来的代码。

从我对待别人代码的方式，你也许已经发现了，我不是一个通常意义上的理想主义者。我不会为了自己简单优雅的理想，而完全推翻重写别人的代码，因为我知道现实世界的复杂性，我知道这样做注定是要失败的。我对待别人代码的态度，是深深地植根于现实的。通过极其严密的措施，我确保改进后的代码跟原来的代码语义完全相同，尽最大可能避免重复前人的错误，避免制造新的 bug。

由于我的理想植根于现实，我把自己称为“现实理想主义者”（practical idealist），而不是“理想主义者”（idealist）。我曾经跟纯粹的理想主义者共事，这种人总是嫌别人的代码丑，不经商量就大幅度的删除重写大量代码，结果给团队的开发带来灾难性的后果。我在将来会避免跟这样的人共事。

通过这个例子，你可能已经发现为什么“现实理想主义”是优于“理想主义”的。下面我来讲一下，为什么“现实理想主义”也超越了完全的“现实主义”。

超越现实主义

既然我不是一个完全的理想主义者，那么是不是说，我就是一个完全的“现实主义者”呢？在我的职业生涯中，我已经多次证明了，我不是一个完全的现实主义者，我能做到现实主义者做不到的事情。我心中的“理想”成分，让我能够看到现实主义者看不到的可能性，而我的“现实”成分，又帮助我为这种可能性找到切实可行的路线。理想和现实的结合，指引我达到现实主义者认为是不可能的目标。

说到这一点，第一个跳进我脑海里的例子，是我当年在 Google 完成的项目。Google 需要一个可以像 IDE 一样索引 Python 代码的工具，可以支持准确的“跳转到定义”功能。作为现实主义者的团队领导（Steve）对我说，你去拿一个开源的 Python 工具，比如 PyDev，修改之后插入到我们的构架里就可以了。

当我调研了十多个开源 Python 工具和 IDE 之后，发现它们都不能准确地实现“跳转到定义”。它们的实现方式基本都是字符串匹配而已，所以找出来的“定义”完全不着边际，甚至把字符串里出现的名字都给加亮了。这时候，我的理想成分告诉我，准确的定义查找应该是可能的，只不过现有的工具都不知道怎么实现它而已。为了给 Python 这样的动态语言实现精确的定义索引，就必须实现类型推导，而这是我很在行的事情。于是我决定做一个新的 Python 类型推导器，这样就可以利用它实现精确的跳转功能。

我把这个想法告诉了 Steve 和其它团队成员，结果作为现实主义者的他们，非常的担心这个项目无法在三个月的实习期内完成。Steve 说：“你知道吗，光是写一个 Python 的 parser 就够写三个月了。我很担心你不能完成任务！”这时候，我的现实成分开始起作用。我说：“你知道吗，我并不觉得写 Python 的 parser 是一件很难的事情，但我也不觉得它是一件很有意义的事情，所以我会拿一个开源的 parser 来，利用它生成的语法树，然后在上面完成我们需要的功能。”

结果，我拿了 Jython 里面的 Python parser，然后在上面实现了 PySonar。整个对付 parser 的过程只花了我两天时间，剩下的时间我都在研究和实现最关键，最有趣的部分。我拿了别人已经做好的，自己不想做的东西来，然后加上自己的核心思想，达到了最终的目的。最后，我不但在三个月的时间里完成了 PySonar，而且把它集成到了 Grok 项目里面。

在这个例子里，现实理想主义者帮助了现实主义者，完成了他们以为不可能的事情。本来 Grok 项目在 Google 处于濒临灭亡的境地，由于 PySonar 的成功实现增大了项目的影响力，团队在 Google 存活了下来，并且开始受到公司的重视，相关人员也获得了提拔。今天 PySonar 仍然在为 Google 的 Python 程序员提供高质量的索引服务，它生成的数据在背后默默支持着 CodeSearch 等内部代码搜索服务。

个人兴趣与企业兴趣

最后，我想再讲一个跟这个话题相关的故事，它说明现实理想主义者不但是一种个人技术财富，而且是企业的财富。他不但与“企业的兴趣”一点矛盾都没有，反而在很多时候可以帮助甚至拯救公司和团队。这个故事很有趣，但中间部分技术性有点强，看不懂的人可以跳过。

我曾经在职的某公司，邀请了某位“大牛”来做 VP。经过一段时间的接触，我发现这个人不懂很多东西，尽在瞎指挥。很明显，他并没有把公司的利益放在心上。在多次的瞎指挥之后，有一天他又提出一个“新想法”。他说，我们团队的代码应该实现“模块化管理”。如何实现模块化管理呢？我们把代码按目录结构切分开，分成 30 个“模块”。把每个模块做成一个 Git 代码库（repository），代码库之间通过 Maven 的版本号依赖关系进行连接。每个人负责一两个模块，使用“语义版本号”（semver）标注模块的版本。如果修改了代码，就更新对应的版本号，这样依赖于这个模块的代码库就必须做出相应的修改，才能连接到新的模块代码，不然它们就可以继续使用旧的模块代码……

这个新想法没有经过团队的集体讨论研究，就被 VP 的一个亲信动手实现了。一夜醒来，我们发现代码库被他分成了 30 多个，制定了一系列规章条款，要我们遵守。接下来的事情，我发现自己没法工作了。一天当中有超过半天的时间，我自己在为那些 semver 伤脑经。你刚刚更新了所有的代码，才工作了个把小时，正要提交的时候，却发现

另外几个模块的版本号更新了！你得手动去看是哪些代码库发生了改变，更新自己 maven 文件里的依赖关系，然后才能进行测试，提交自己的代码。有时候当你提交之前，忽然又有其它的模块版本号发生了改变，所以你前功尽弃，又得去查到底是谁改了他的模块版本号。有很多次，有人没有把版本号完全搞对就提交了代码，结果导致项目 build 失败。

后来我发现，这种所谓的“模块化”，根本就不是真正的模块化，而 semver 版本号，在这里也并不比 Git 的 hash 更好。模块不应该是按目录结构划分的，而应该是按代码的逻辑结构，而且模块之间不应该有“循环依赖关系”，否则这些模块就不应该被分成模块，而应该合并在一起。另外，semver 根本不是用来干这个事情的，它根本不应该被用于连接同一个项目里的多个模块，它只能被用来引用库代码。每一个 Git commit 的 hash，本身就是一个“全宇宙唯一”的版本号，它包含了代码所处的独一无二的状态。所以 Git 其实自然而然的解决了这种“模块”间版本依赖的问题。所以把代码拆分成 30 多个 Git 代码库，使用 semvar 连接它们，完全是多此一举，而且严重的损害了开发效率。

观察到这个问题之后，我向团队群发了邮件，告诉他们我觉得这样的做法已经造成了我工作效率严重打折，并且指出了问题的要害。一个来自法国的资深工程师深有同感，也开始抱怨，说自己花了超过一半的时间来折腾这些版本号。然而 VP 听了这些意见，却坚持认为自己的“创新”是有价值的，对我们说：“任何一项伟大的创新，都会受到不理解它的旧势力的阻碍。同志们，困难是暂时的，适应是必须的！”为了这个问题，我们在 email 里面吵了两个星期之久。任凭我们据理力争，拿出具体的证据证明这种做法不可行，严重的伤害了团队的开发效率，VP 凭着自己的名气和地位，毫不退缩。

最后无奈之下，我决定采取实际的行动。我写了一个 Python 脚本，它调用 Git 的一些罕见命令，可以自动把多个 Git 代码库合并成一个，并且保留所有的历史 commit 信息。有了这个脚本之后，我可以随时制造出一个合并的代码库。我把这个脚本分享给了团队，告诉他们我随时可以把代码库合并在一起，而且给了他们一个合并后的代码库，作为试验用。我告诉他们，可以试用这个代码库，看它是否解决了 30 个代码库带来的问题。最后法国同事和其它几个人采用了我的代码库，发现不再有之前的头痛问题。

我们用理论和切实的证据证明了所谓的“模块化代码管理”的不可行。通过对其它公司代码的观察，我们发现 Google 的 Chrome 项目有三千多万行代码，却全都存放在同一个 Git 代码库里。这说明一个 Git 代码库足以支持管理 Chrome 那么大的项目。我们的团队总共才 20 多人，代码不超过十万行，却被强行切分成 30 多个代码库，这是非常荒唐滑稽的。

最后在工程师们的一致同意下，再加上团队 director 委婉的支持，我用脚本将 30 个代码库合并在了一起，结束了大家的痛苦……在此之后，VP 的亲信们还不死心，在合并后的代码库里又做了一些手脚，故意加大工作的复杂性，让我们依赖于他们的“工具”，这些我不细说了。总之你看到了，这位 VP 的瞎指挥，导致团队浪费很多的时间和精力。如果这种情况不受控制继续下去，整个团队甚至整个公司，都有可能因此走向灭亡。

我发现很多所谓管理人物，他们到一个新的公司出任要职，其实并没把公司的利益放在心上。他们不是为了公司的发展和成功做出决定，而是为了自己的“仕途”。这些管理者明白，公司就像一艘船，自己表面上在为公司服务，而其实是在利用公司的资源达成自己的目标。由于自己挥霍公司的资源，而不作出实质的贡献，甚至瞎指挥帮倒忙，这艘船在将来很可能会沉没。但作为管理者，自己总是可以在沉船之前跳到另外一艘船上，靠着自己的关系网，不断找到高薪的职位……

像这样的例子我还有很多。为了团队，为了公司能够达成自己的目标，我多次顶着压力，帮助团队和公司避免不必要的浪费，甚至悬崖勒马。当然很多时候团队在错误的道路上走得太远，看清真相的我却受到压制，没有话语权，所以也爱莫能助，只能听之任之。注意我在这里谈“企业利益”，并不是说我喜欢为资本家卖命。这里的“公司”和“企业”，只是代表一个集体，它包括了公司里所有的员工和股东。

从这样一个例子，你也可以看到我作为一个“现实理想主义者”的特征。这个 VP 可算是“理想主义”了，他一拍脑袋提出了“新颖”的，其它公司都没想到的工作方式，结果却给大家带来了灾难。我从现实和理性的角度，分析得知这种做法的荒谬，论证了“传统做法”的和理性，与他据理力争，维护公司和团队的利益，再加上团结大多数有职业素养的工程师，最终我们合力战胜了 VP 的瞎指挥，逆转了他给团队和公司带来的伤害，避免了灾难性的后果。

当我离开这个公司的时候，我收到了这样一封来自团队成员的感谢 email：

to me ▾

Thanks for helping us move forward while preserving common sense and sanity. You'll be missed!

...

他说：“谢谢你帮助我们保持了常理和理智，把事业推向前进。我们会怀念你的！”

这样的现实理想主义者，不管是作为员工，作为团队的领导，还是作为公司的统帅，都会身体力行，给他们带来帮助，避免不必要的浪费和弯路，引导企业走上正轨，走向兴旺繁荣。我希望广大 IT 工作者能理解我这里说的东西，把自己的“伟大理想”植根于现实，避免因为自己的轻狂而走向歧途。

如果你觉得这篇文章对你有帮助，可以自愿[付费购买](#)，建议零售价：¥ 30。

人工智能的局限性

有人听说我想创业，给我提出了一些“忽悠”的办法。他们说，既然你是程序语言专家，而现在人工智能（AI）又非常热，那你其实可以搞一个“自动编程系统”，号称可以自动生成程序，取代程序员的工作，节省许许多多的人力支出，这样就可以趁着“AI 热”拉到投资。

有人甚至把名字都给我想好了，叫“深度程序员”（DeepCoder = Deep Learning + Coder）。口号是：“有了 DeepCoder，不用 Top Coder！”还有人给我指出了这方向最新的，吹得神乎其神的研究，比如微软的 [Robust Fill](#)……

我谢谢这些人的关心，然而其实人工智能的能力被严重的夸大了。现在我简单的讲一下我的看法。

识别系统和语言理解

纵观历史上机器学习能够做到的事情，都是一些字符识别（OCR），语音识别，人脸识别一类的，我把这些统称为“识别系统”。当然，识别系统是有价值的，OCR 是有用的，我经常用手机上的语音输入法，人脸识别对于公安机关显然意义很大。然而很多人因此夸口，说我们可以用同样的方法（机器学习，深度学习），实现“人类级别的智能”，取代大量的人类工作（比如客服，保洁，送外卖，司机……），这就是神话了。这些人完全没有理解这些看似“简单枯燥”的人类工作背后所隐含的，难以逾越的难度。

识别系统跟真正理解语言的“人类智能”，其实相去非常远。说白了，这些识别系统，也就是统计学的拟合函数能做的事情。比如 OCR 和语音识别，就是输入像素或者音频，输出单词文本。很多人分不清“文字识别”和“语言理解”的区别。OCR 和语音识别系统，虽然能依靠统计的方法，“识别”出你说的是哪些字，它却不能真正“理解”你在说什么。

聊一点深入的话题，看不懂的人可以跳过这一段。“识别”和“理解”的差别，就像程序语言里面“语法”和“语义”的差别。程序语言的文本，首先要经过词法分析器（lexer），语法分析器（parser），才能送进[解释器](#)

（interpreter），只有解释器才能实现程序的语义。类比一下，自然语言的语音识别系统，其实只相当于程序语言的词法分析器（lexer）。我在之前的文章里已经指出，词法分析和语法分析，只不过是实现一个语言的万里长征的“第0步”。

大部分的 AI 系统里面连语法分析器（parser）都没有，所以主谓宾，句子结构都分析不清楚，更不要说理解其中的含义了。IBM 的语音识别专家 [Frederick Jelinek](#) 曾经开玩笑说：“每当我开掉一个语言学家，识别率就上升了。”其原因就是语音识别仅相当于一个 lexer，而语言学家研究的是 parser 以及 interpreter。当然了，你们干的事情太初级了，所以语言学家帮不了你们，但这并不等于语言学家是没有价值的。

很多人语音识别专家以为语法分析（parser）是没用的，因为人好像从来没有 parse 过句子，就理解了它的意义。然而他们没有察觉到，人其实必须要不知不觉地 parse 有些句子，才能理解它的含义。

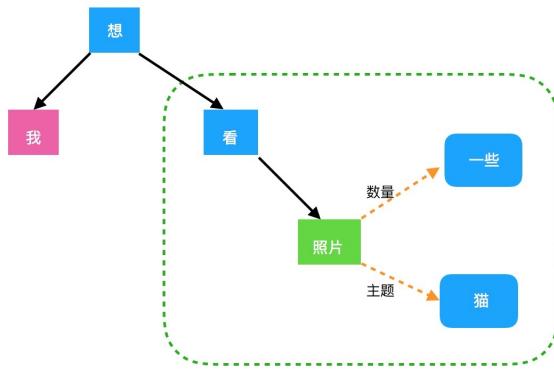
举一个很简单的例子。如果我对 Siri 说：“我想看一些猫的照片。”它会给我下图的回答：“我在网上没有找到与‘一些猫’有关的资料。”



这说明了什么呢？很多人可能都发现了，这说明了 Siri 无法理解这个句子，所以它到网上去搜一些关键字。可是这还说明一个更深层次的问题，那就是 Siri 里面并没有 parser，甚至连一个好的分词系统都没有，所以它连该搜什么关键字都不知道。

为什么 Siri 去网上找关于“一些猫”的信息，而不是关于“猫”的信息呢？如果搜索“猫”和“照片”，它至少能找到一些东西。这是因为 Siri 其实没有 parser，它里面根本没有语法树。它只是利用一些普通的 NLP 方法（比如 n-gram），把句子拆成了“我...想...看...一些猫...的...照片”，而不是语法树对应的“我...想...看...一些...猫...的...照片”。

这个句子的语法树，按照我之前做过的一种自然语言 parser 的方式，分析出来大概是这个样子。



具体细节太过技术性，我就不在这里解释了。不过有兴趣的人可能发现了，根据语法树，这句话可以简化为：“我想看照片。”其中“看照片”是一个从句，它是“我想...”的宾语，也就是所谓宾语从句。多少照片呢？一些。看什么样的照片呢？主题是猫的照片。

- 我想看照片
- 我想看一些照片
- 我想看猫的照片
- 我想看一些猫的照片

是不是挺有意思？

Siri 里面没有这种语法树，而且它的 n-gram 居然连“一些”和“猫”都没分开，这就是为什么它去找“一些猫”，而不是“猫”。它甚至把“照片”这么重要的词都忽略了。所以 Siri 虽然正确的进行了“语音识别”，知道我说了那些字。但由于没有 parser，没有语法树，它不可能正确的理解我到底在说什么，它甚至不知道我在说“关于什么”。

制造自然语言的 parser 有多难？很多人可能没有试过。我做过这事。在 Indiana 的时候，我为了凑足学分，修了一门 NLP 课程，跟几个同学一起实现了一个英语语法的 parser。它分析出来的语法树形式，就像上面的那样。

你可能想不到有多困难，你不仅要深刻理解编程语言的 parser 理论 (LL, LR, GLR.....)，还得依靠大量的例子和数据，才能解开人类语言里的各种歧义。我的合作伙伴是专门研究 NLP 的，把什么 Haskell，类型系统，category theory，什么 GLR parsing 之类..... 都弄得很溜。然而就算如此，我们的英语 parser 也只能处理最简单的句子，还错误百出，最后蒙混过关 :P

经过了语法分析，得到一棵“语法树”，你才能传给人脑里语言的理解中心（类似程序语言的“解释器”）。解释器“执行”这个句子，为相关的名字找到对应的“值”，进行计算，才能得到句子的含义。至于人脑如何为句子里的词汇赋予“意义”，如何把这些意义组合在一起，形成“思维”，这个问题似乎没有人很明白。

至少，这需要大量的实际经验，这些经验是一个人从生下来就开始积累的。机器完全不具备这些经验，我们也不知道如何才能让它获得经验。我们甚至不知道这些经验在人脑里面是什么样的结构，如何组织的。所以机器要真的理解一个句子，真是跟登天一样难。

这就是为什么 Hofstadter 说：“一个机器要能理解人说的话，它必须要有腿，能够走路，去观察世界，获得它需要的经验。它必须能够跟人一起生活，体验他们的生活和故事.....” 最后你发现，制造这样一个机器，比养个小孩困难太多了，这不是吃饱了没事干什么。

机器对话系统和人类客服

各大公司最近叫得最响亮的“AI 技术”，就是 Siri，Cortana，Google Assistant，Amazon Echo 一类含有语音识别功能的工具，叫做“个人助手”。这些东西里面，到底有多少可以叫做“智能”的东西，我想用过的人都应该明白。我每一次试用 Siri 都被它的愚蠢所折服，可以让你着急得砸了水果手机。那另外几个同类，也没有好到哪里去。

很多人被“微软小冰”忽悠过，乍一看这家伙真能理解你说的话呢！然而聊一会你就发现，小冰不过是一个“网络句子搜索引擎”。它只是按照你句子里的关键字，随机搜出网上已有的句子。大部分这类句子出自问答类网站，比如百度知道，知乎。

一个很简单的实验，就是反复发送同一个词给小冰，比如“王垠”，看它返回什么内容，然后拿这个内容到 Google 或者百度搜索，你就会找到那个句子真正的出处。人都喜欢自欺欺人，看到几个句子回答得挺“俏皮”，就以为它有智能，而其实它是随机搜出一个句子，牛头不对马嘴，所以你才感觉“俏皮”。比如，你跟小冰说：“王垠是谁？”，她可能回答：“王垠这是要变段子手么。”



心想多可爱的妹子，不正面回答你的问题，有幽默感！然后你在百度一搜，发现这句话是某论坛里面黑我的人说的。

下面是一个确切的例子，它显示了小冰是如何工作的。图片是 2016 年 10 月底抓的，那时候我试了一下跟小冰对话。现在的情况可能稍微有所不同。



这说明小冰的答复，基本是百度问答，知乎一类的地方来的，它只是对那上面的数据做了一个搜索匹配。随机搜索出这句子作为回答，至于幽默感，完全是你自己想象出来的。很多人跟小冰对话，喜欢只把其中“符合逻辑”或者“有趣”的部分截图下来，然后惊呼：“哇，小冰好聪明好有趣！”他们没有告诉你的是，没贴出来的对话，很多都是鸡同鸭讲，枯燥无味，把人都聊走了。

IBM 的 Watson 系统在 Jeopardy 游戏中战胜了人，很多人就以为 Watson 能理解人类语言，具有人类级别的智

能。这些人甚至都不知道 Jeopardy 是怎么玩的，就盲目做出判断，以为 Jeopardy 是一种需要理解人类语言才可以玩的游戏。等你细看，发现 Jeopardy 就是很简单的“猜谜”游戏，题目是一句话，答案是一个名词。比如：“有个歌手去年得了十项格莱美奖，请问他是谁？”

如果你理解了我之前对“识别系统”的分析，就会发现 Watson 也是一种识别系统，它的输入是一个句子，输出是一个名词。一个可以玩 Jeopardy 的识别系统，可以完全不理解句子的意思，而是依靠句子里出现的关键字，依据分析大量语料得到的拟合函数，输出一个单词。世界上那么多的名词，到哪里去找这样的语料呢？这里我给你一个 Jeopardy 谜题作为提示：“什么样的网站，你给它一个名词，它输出一些段落和句子，给你解释这个东西是什么，并且提供给你各种相关信息？”

很容易猜吧？就是 Wikipedia 那样的百科全书！你只需要把这种网站的内容掉一个头，制造一个“倒索引”搜索引擎。你输入一个句子，它就根据里面的关键字，搜索到最相关的名词。这就是一台可以玩 Jeopardy 的机器，而且它很容易超越人类玩家，就像 Google，Yahoo 之类的搜索引擎很容易超越人查找网页的能力一样。可是这里面基本没有理解和智能可言。

其实为了验证 Watson 是否理解人类语言，我早些时候去 Watson 的网站玩过它的“客服 demo”，结果完全是鸡同鸭讲，大部分时候 Watson 回答：“我不清楚你在说什么。你是想要……”然后列出一堆选项，1, 2, 3……

你指望拿这样的东西代替你公司的人类客服吗？那你的公司就等着倒闭吧。

当然，我并不是说这些产品完全没有价值。我用过 Siri 和 Google Assistant，我发现它们还是有点用处的，特别是在开车的时候。因为开车时操作手机容易出事故，所以我可以利用语音控制。比如我可以对手机说：“导航到最近的加油站。”然而实现这种语音控制，根本不需要理解语言，你只需要用语音识别输入一个函数调用：导航（加油站）。

个人助手在其它时候用处都不大。我不想在家里和公共场所使用它们，原因很简单：我懒得说话，或者不方便说话。点击几下屏幕，我就可以精确地做到我想要的事情，这比说话省力很多，也精确很多。个人助手完全不理解你在说什么，这种局限性本来无可厚非，可以用就行了，然而各大公司最近却拿个人助手这类东西来煽风点火，夸大其中的“智能”成分，闭口不提他们的局限性，让外行们以为人工智能就快实现了，这就是为什么我必须鄙视一下这种做法。

举个例子，由于有了这些“个人助手”，有人就号称类似的技术可以用来制造“机器客服”，使用机器代替人作为客服。他们没有想清楚的是，客服看似“简单工作”，跟这些语音控制的玩意比起来，难度却是天壤之别。客服必须理解公司的业务，必须能够精确地理解客户在说什么，必须形成真正的对话，要能够为客户解决真正的问题，而不能只抓住一些关键字进行随机回复。

另外，客服必须能够从对话信息，引发现实世界的改变，比如呼叫配送中心停止发货，向上级请求满足客户的特殊要求，拿出退货政策跟客户辩论，拒绝他们的退货要求，抓住客户心理，向他们推销新服务等等，各种需要“人类经验”才能处理的事情。所以机器不但要能够形成真正的对话，理解客户的话，它们还需要现实世界的大量经验，需要改变现实世界的能力，才可能做客服的工作。由于这些个人助手全都是在忽悠，所以我看不到有任何希望，能够利用现有的技术实现机器客服。

连客服这么按部就班的工作，机器都无法取代，就不用说更加复杂的工作了。很多人看到 AlphaGo 的胜利，以为所谓 Deep Learning 终究有一天能够实现人类级别的智能。在之前的一篇[文章](#)里，我已经指出了这是一个误区。很多人以为人觉得困难的事情（比如围棋），就是体现真正人类智能的地方，其实不是那样的。我问你，心算除法（23423451345 / 729）难不难？这对于人是很难的，然而任何一个傻电脑，都可以在 0.1 秒之内把它算出来。围棋，国际象棋之类也是一样的原理。这些机械化的问题，其实不能反应真正的人类智能，它们体现的只是大量的蛮力。

纵观人工智能领域发明过的吓人术语，从 Artificial Intelligence 到 Artificial General Intelligence，从 Machine Learning 到 Deep Learning，…… 我总结出这样一个规律：人工智能的研究者们似乎很喜欢制造吓人的名词，当人们对一个名词失去信心，他们就会提出一个不大一样的，新的名词，免得人们把对这个名词的失望，转移到新的研究上面。然而这些名词之间，终究是换汤不换药。因为没有人真的知道人的智能是什么，所以也就没有办法实现“人工智能”。

生活中的每一天，我这个“前 AI 狂热者”都在为“人类智能”显示出来的超凡能力而感到折服。甚至不需要是人，任何高等动物（比如猫）的能力，都让我感到敬畏。我发自内心的尊重人和动物。我不再有资格拿“人类”来说事，因为面对这个词汇，任何机器都是如此的渺小。

纪念我的聊天机器人 helloooo

乘着这个热门话题，现在我来讲一下，十多年前我自己做聊天机器人的故事……

如果你看过 PAIP 或者其它的经典人工智能教材，就会发现这些机器对话系统，最初的思想来自一个叫“[ELIZA](#)”的 AI 程序。Eliza 被设计为一个心理医生，跟你对话排忧解难，而它内部其实就是一个类似小冰的句子搜索引擎，实现方式完全用正则表达式匹配搞定。比如，Eliza 的某个规则可以说，当用户说：“我(.)”，那么你就回答：“我也\$1……”其中 \$1 代替原句子里的一部分，造成一种“理解”的效果。比如用户也许会说：“我好无聊。” Eliza 就可以说：“我也好无聊……”然后这两个无聊的人就惺惺相惜，有伴了。

有些清华的老朋友也许还记得，十多年前在清华的时候，我做了一个聊天机器人放在水木清华 BBS，红极一时，所以

我也可以算是网络聊天机器人的鼻祖了 :) 我的聊天机器人，水木账号叫 helloooo。helooooo 的性格像蜡笔小新，是一个调皮又好色的小男孩。

它内部采用的就是类似 Eliza 的做法，根本不理解句子，甚至连语料库都没有，神经网络也没有，里面就是一堆我事先写好的正则表达式“句型”而已。你输入一个句子，它匹配之后，从几种回复之中随机挑一个，所以你反复说同样的话，helooooo 的回答不会重复，如果你故意反复说同样的话，最后 helooooo 会对你说：“你怎么这么无聊啊？”或者“你有病啊？”或者转移话题，或者暂时不理你……这样对方就不会明显感觉它是一个傻机器。

就这么简单个东西。出乎我意料的是，helooooo 一上网就吸引了很多人。一传十十传百，每天都不停地有人发信息跟他聊。由于我给他设置的正则表达式和回复方式考虑到了人的心理，所以 helooooo 显得很“俏皮”，有时候还可能装傻，捣蛋，延迟回复，转移话题，还可能主动找你聊天，使用超过两句的小段子，…… 各种花样都有。最后，这个小色鬼赢得了好多妹子们的喜爱，甚至差点约了几个出去 :)

在这点上，helooooo 可比小冰强很多。小冰的技术含量虽然多一些，数据多很多，然而 helooooo 感觉更像一个有感觉的人，更受欢迎。这说明，我们其实不需要很高深的技术，不需要理解自然语言，只要你设计巧妙，抓住人的心理，就能做出人们喜爱的聊天机器。

后来，helooooo 终于引起了清华大学人智组研究生的兴趣，来问我：“你这里面使用的什么语料库做分析啊？”我：“&%& ¥@#@#%……”

自动编程是不可能的

现在回到有些人最开头的提议，实现自动编程系统。我现在可以很简单的告诉你，那是不可能实现的。微软的 [Robust Fill](#) 之类，全都是在扯淡。我对微软最近乘着 AI 热，各种煽风点火的做法，表示少许鄙视。不过微软的研究员也许知道这些东西的局限，只是国内小编在夸大它的功效吧。

你仔细看看他们举出的例子，就知道那是一个玩具问题。人给出少量例子，想要电脑完全正确的猜出他想做什么，那显然是不可能的。很简单的原因，例子不可能包含足够的信息，精确地表达人想要什么。最最简单的变换也许可以，然而只要多出那么一点点例外情况，你就完全没法猜出来他想干什么。就连人看到这些例子，都不知道另一个人想干什么，机器又如何知道？这根本就是想实现“读心术”。甚至人自己都可以是糊涂的，他根本不知道自己想干什么，机器又怎么猜得出来？所以这比读心术还要难！

对于如此弱智的问题，都不能 100% 正确的解决，遇到稍微有点逻辑的事情，就更没有希望了。论文最后还“高瞻远瞩”一下，提到要把这作法扩展到有“控制流”的情况，完全就是瞎扯。所以 RobustFill 所能做的，也就是让这种极其弱智的玩具问题，达到“接近 92% 的准确率”而已了。另外，这个 92% 是用什么标准算出来的，也很值得怀疑。

任何一个负责的程序语言专家都会告诉你，自动生成程序是根本不可能的事情。因为“读心术”是不可能实现的，所以要机器做事，人必须至少告诉机器自己“想要什么”，然而表达这个“想要什么”的难度，其实跟编程几乎是一样的。实际上程序员工作的本质，不就是在告诉电脑自己想要它干什么吗？最困难的工作（数据结构，算法，数据库系统）已经被固化到了库代码里面，然而表达“想要干什么”这个任务，是永远无法自动完成的，因为只有程序员自己才知道他想要什么，甚至他自己都要想很久，才知道自己想要什么……

有句话说得好：编程不过是一门失传的艺术的别名，这门艺术的名字叫做“思考”。没有任何机器可以代替人的思考，所以程序员是一种不可被机器取代的工作。虽然好的编程工具可以让程序员工作更加舒心和高效，任何试图取代程序员工作，节省编程劳力开销，克扣程序员待遇，试图把他们变成“可替换原件”的做法（比如 Agile，TDD），最终都会倒戈，使得雇主收到适得其反的后果。同样的原理也适用于其它的创造性工作：厨师，发型师，画家，……

所以别妄想自动编程了。节省程序员开销唯一的办法，是邀请优秀的程序员，尊重他们，给他们好的待遇，让他们开心安逸的生活和工作。同时，开掉那些满口“Agile”，“Scrum”，“TDD”，“[软件工程](#)”，光说不做的扯淡管理者，他们才是真正浪费公司资源，降低开发效率和软件质量的祸根。

我的人工智能梦

回过头来，很多人可能不知道，我也曾经是一个“AI 狂热者”。我也曾经为人工智能疯狂，把它作为自己的“伟大理想”。我也曾经张口闭口拿“人类”说事，仿佛机器是可以跟人类相提并论，甚至高于人类的。当深蓝电脑战胜卡斯帕罗夫，我也曾经感叹：“啊，我们人类完蛋了！”我也曾经以为，有了“逻辑”和“学习”这两个法（kou）宝（hao），机器总有一天会超越人类的智能。可是我没有想清楚这具体要怎么实现，也没有想清楚实现了它到底有什么意义。

故事要从十多年前讲起，那时候人工智能正处于它的冬天。在清华大学的图书馆，我偶然地发现了一本尘封已久的『[Paradigms of Artificial Intelligence Programming](#)』（PAIP），作者是 Peter Norvig。像个考古学家一样，我开始逐一地琢磨和实现其中的各种经典 AI 算法。PAIP 的算法侧重于逻辑和推理，因为在它的年代，很多 AI 研究者都以为人类的智能，归根结底就是逻辑推理。

他们天真地以为，有了谓词逻辑，一阶逻辑这些东西，可以表达“因为所以不但而且存在所有”，机器就可以拥有智能。于是他们设计了各种基于逻辑的算法，专家系统（expert system），甚至设计了基于逻辑的程序语言 Prolog，把它叫做“第五代程序语言”。最后，他们遇到了无法逾越的障碍，众多的 AI 公司无法实现他们夸口的目标，各种基于“神经元”的机器无法解决实际的问题，巨额的政府和民间投资化为泡影，人工智能进入了冬天。

我就是在那样一个冬天遇到了 PAIP。它虽然没能让我投身于人工智能领域，却让我迷上了 Lisp 和程序语言。也是因为这本书，我第一次轻松而有章法的实现了 A* 等算法。我第一次理解到了程序的“模块化”是什么，在代码例子的引导下，我开始在自己的程序里使用小的“工具函数”，而不再忧心忡忡于“函数调用开销”。PAIP 和 SICP 这两本书，最后导致了我投身于更加“基础”的程序语言领域，而不是人工智能。

在 PAIP 之后，我又迷了一阵子机器学习（machine learning），因为有人告诉我，机器学习是人工智能的新篇章。然而我逐渐意识到，所谓的人工智能和机器学习，跟真正的人类智能，关系其实不大。相对于实际的问题，PAIP 里面的经典算法要么相当幼稚，要么复杂度很高，不能解决实际的问题。最重要的问题是，我看不出 PAIP 里面的算法跟“智能”有什么关系。而“机器学习”这个名字，基本是一个幌子。很多人都看出来了，机器学习说白了就是统计学里面的“拟合函数”，换了一个具有迷惑性的名字而已。

人工智能的研究者们总是喜欢抬出“神经元”一类的名词来吓人，跟你说他们的算法是受了人脑神经元工作原理的启发。注意了，“[启发](#)”是一个非常模棱两可的词，由一个东西启发得来的结果，可以跟这个东西毫不相干。比如我也可以说，Yin 语言的设计是受了九 yin 真经的启发 :P

世界上这么多 AI 研究者，有几个真的研究过人脑，解剖过人脑，拿它做过实验，或者读过脑科学的研究成果？最后你发现，几乎没有 AI 研究者真正做过人脑或者认知科学的研究。著名的认知科学家 Douglas Hofstadter 早就在接受采访时指出，这帮所谓“AI 专家”，对人脑和意识（mind）是怎么工作的，其实完全不感兴趣，也从来没有深入研究过，却号称要实现“通用人工智能”（Artificial General Intelligence, AGI），这就是为什么 AI 直到今天都只是一个虚无的梦想。

傻机器的价值

我不反对继续投资研究那些有实用价值的人工智能（比如人脸识别一类的），然而我觉得不应该过度夸大它的用处，把注意力过分集中在它上面，仿佛那是唯一可以做的事情，仿佛那是一个划时代的革命，仿佛它将取代一切人类劳动。

我的个人兴趣其实不在人工智能上面。那我要怎么创业呢？很简单，我觉得大部分人不需要很“智能”的机器，“傻机器”才是对人最有价值的，我们其实远远没有开发完傻机器的潜力。所以设计新的，可靠的，造福于人的傻机器，应该是我创业的目标。当然我这里所谓的“机器”，包括了硬件和软件，甚至可以包括云计算，大数据等内容。

只举一个例子，有些 AI 公司想研制“机器佣人”，可以自动打扫卫生做家务。我觉得这问题几乎不可能解决，还不如直接请真正智能的——阿姨来帮忙。我可以做一个阿姨服务平台，方便需要服务的家庭和阿姨进行牵线搭桥。给阿姨配备更好的工具，通信，日程，支付设施，让她工作不累收钱又方便。另外给家庭提供关于阿姨工作的反馈信息，让家庭也省心放心，那岂不是两全其美？哪里需要什么智能机器人，难度又高，又贵又不好用。显然这样的阿姨服务平台，结合真正的人的智能，轻而易举就可以让那些机器佣人公司死在萌芽之中。

当然我可能不会真去做个阿姨服务平台，这种东西可能已经有了。我只是举个例子，说明许许多多对人有用傻机器，还在等着我们去发明。这些机器设计起来虽然需要灵机一动，然而实现起来难度却不高，给人带来便利，经济上见效也快。利用人的智慧，加上机器的蛮力，让人们又省力又能挣钱，才是最合理的发展方向。

（如果你喜欢这篇文章，欢迎[付款](#)。）

经验和洞察力

很多人很在乎“经验”，比如号称自己在某领域有 30 年的经验，会用这样那样的技术。我觉得经验是有价值的，我也有经验，各个领域的都有点。然而我并不把经验放在很重要的位置，因为我拥有大部分人都缺乏而且忽视的一种东西：洞察力（insight）。

每进入一个新的公司，我进入的几乎都是不同的领域。所以最开头的时候，我有可能对那个领域所知甚少。甚至有人觉得我没有经验，所以可以“教育”我。然而每一次他们都没有想到的是，我很快就掌握了他们的经验，并且经过提炼，抛弃其中的垃圾，很快的超越了他们，完成他们根本无法达到的目标。这就是洞察力的威力。

举个亲身例子，很多人都有用线程的经验，可是有多少人知道线程的本质是什么？有多少人在头脑里有一幅画面，显示出多线程程序的各种动态特征？其实很少有人知道。这就是为什么很多人过度的使用线程并发，结果产生各种同步问题，竞争状态（race condition），死锁等现象。某公司的一片多线程代码，号称是“有非常多并发程序经验”的程序员写的。结果没多久我就发现里面其实含有非常微妙的竞争情况，会在非常小的概率随机发作。发现之后没过几天，已经卖出去用了两年多的产品，由于这个竞争情况，终于引发了严重的后果。有那么多并发编程经验的程序员，两年多都没有察觉这个竞争情况，而很少写多线程程序的我，不但发现了这个竞争，而且很快的想出了修复它的办法，这是为什么呢？靠的就是洞察力。我知道线程的本质，而这是经验不会告诉你的。

什么是洞察力？洞察力就是透过现象看到本质的能力。有洞察力的人很容易得到经验，然而有经验的人却不一定有洞察力。再愚钝的人，总是可以通过大量的时间获取经验，然而就算你花再多的时间和精力，也难以得到洞察力。所以洞察力是比经验宝贵很多的东西。很难说清楚如何才能有洞察力，也很少有人会告诉你如何去得到它。当然，我也不告诉你会告诉你。

看别人简历，经常会列出各种各样的技术经验，我看一眼就会的东西，也会在上面占个位置。由于这个原因，我把自己 LinkedIn 上面曾经列出的“工作经验”全都删掉了。这些东西列在那里，对于我本身的价值，实在是一种贬低。我是一个身上不贴任何标签的，不能被任何头衔所局限的，真正有价值的人。

经验虽然不是最重要的，然而还是有必要的。很多技术你不能完全不碰它，然而一碰就明白了。但如果沒有实际的问题，你又会没有动力去接触那些技术。所以我一直在做的一件事情，就是接触各种技术，然后利用洞察力来获得越来越多的经验。回国之后的初期，我打算着手做自己的产品。同时，我想跟国内的各种公司或者个人做这样的交易。我利用洞察力帮助解决他们最棘手的，已有经验无法解决的难题，从而让我获得经验。当然，我不是作为公司的职工，而只是作为独立的顾问。对公司我会象征性的收取一定的费用，换句话，就是作为“职业杀手”。对于个人，他的问题必须对我也有启发意义。对此感兴趣的公司或者个人，可以跟我联系。

C# 的 IDisposable 接口

我在微软的团队快被微软 C# 里面的各种 [IDisposable](#) 对象给折腾疯了……

故事比较长，先来科普一下。如果你没有用过 C#，IDisposable 是 C# 针对“资源管理”设计的一个接口，它类似于 Java 的 [Closeable](#) 接口。这类接口一般提供一个“方法”（比如叫 Dispose 或者 Close），你的资源（比如文件流）实现这个接口。使用资源的人先“打开资源”，用完之后调用这个方法，表示“关闭资源”。比如，文件打开，读写完了之后，调用 close 关掉。看似很简单？;-)

相比于 Java，C# 大部分时候是更好的语言，然而它并没有全面超越 Java。一个显著的不足之处就是 C# 的 IDisposable 接口引起的头痛，要比 Java 的 Closeable 大很多。经过我分析，这一方面是因为 .NET 库代码里面实现了很多没必要的 IDisposable，以至于你经常需要思考如何处理它们。另一方面是由于微软的编码规范和 Roslyn 静态分析引起的误导，使得用户对于 IDisposable 接口的“正确使用”过度在乎，导致代码无端变得复杂，导致 IDisposable 在用户代码里面传染。

IDisposable 的问题

回来说说我们的代码，本来没那么多问题的，结果把 [Roslyn 静态分析](#) 一打开，立马给出几百个警告，说“你应该调用 Disposable 成员的 Dispose 方法”（[CA2213](#)），或者说“类型含有 disposable 成员，却没有实现 IDisposable 接口”（[CA1001](#)）。

奇葩的是，C# 里面有些很小却很常用的对象，包括 [ManualResetEvent](#), [Semaphore](#), [ReaderWriterLockSlim](#) 都实现了 IDisposable 接口，所以经常搞得你不知所措。按官方的“规矩”，你得显式的调用所有这些对象的 Dispose 方法进行“释放”，而不能依赖 GC 进行回收。所以你的代码经常看起来就像这个样子：

```
void foo()
{
    var event = new ManualResetEvent(false);
    // 使用 _event ...
    event.Dispose();
}
```

貌似没什么困难嘛，我们把每个对象的 Dispose 方法都调用一下，不就得了？然而问题远远不是这么简单。很多时候你根本搞不清楚什么时候该释放一个对象，因为它存在于一个复杂，动态变化的数据结构里面。除非你使用引用计数，否则你没有办法确定调用 Dispose 的时机。如果你过早调用了 Dispose 方法，而其实还有人在用它，就会出现严重的错误。

这问题就像 C 语言里面的 free，很多时候你不知道该不该 free 一块内存。如果你过早的 free 了内存，就会出现非常严重而蹊跷的内存错误，比泄漏内存还要严重很多。举一个 C 语言的例子：

```
void main()
{
    int *a = malloc(sizeof(int));
    *a = 1;

    int *b = malloc(sizeof(int));
    *b = 2;

    free(a);

    int *c = malloc(sizeof(int));
    *c = 3;

    printf("%d, %d, %d\n", *a, *b, *c);
}
```

你知道这个程序最后是什么结果吗？自己运行一下看看吧。所以对于复杂的数据结构，比如图节点，你就只好给对象加上引用计数。相信我，使用引用计数很痛苦。或者如果你的内存够用，也不需要分配释放很多中间结果，那你就干脆把这些对象都放进一个“池子”，到算法结束以后再一并释放它们……

是的 C# 有垃圾回收（GC），所以你以为不用再考虑这些低级问题了。不幸的是，IDisposable 接口以及对于它兢兢业业的态度，把这麻烦事给带回来了。以前在 Java 里用此类对象，从来没遇到过这么麻烦的事情，最多就是打开文件的时候要记得关掉（关于文件，我之后会细讲一下）。

我不记得 Java 的等价物（[Closeable](#) 接口）引起过这么多的麻烦，Java 的 [Semaphore](#) 根本就没有实现 Closeable 接口，也不需要在用完之后调用什么 Close 或者 Dispose 之类的方法。作为一个眼睛雪亮的旁观者，我开始怀疑 C# 里的那些像 Semaphore 之类的小东西是否真的需要显式的“释放资源”。

.NET 库代码实现不必要的 **IDisposable** 接口

为了搞明白 C# 库代码里面为什么这么多 **IDisposable** 对象，我用 JetBrains 出品的反编译器 [dotPeek](#)（好东西呀）反编译了 .NET 的库代码。结果发现好些库代码实现了完全没必要的 **IDisposable** 接口。这说明有些 .NET 库代码的作者其实没有弄明白什么时候该实现 **IDisposable**，以及如何有意义地实现它。

这些有问题的类，包括常用的 **HashAlgorithm**（各种 SHA 算法的父类）和 **MemoryStream**。其中 **HashAlgorithm** 的 **Dispose** 方法完全没必要，这个类的源代码看起来是这个样子：

```
public abstract class HashAlgorithm : IDisposable, ICryptoTransform {
    ...
    protected internal byte[] HashValue;
    ...
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (HashValue != null)
                Array.Clear(HashValue, 0, HashValue.Length);
            HashValue = null;
            m_bDisposed = true;
        }
    }
}
```

看明白了吗？它不过是在把内部数组 **HashValue** 的每个元素清零，然后把指针设为 **null**。这个库代码作者没有搞明白的是，如果你的 **Dispose** 方法只是在把一些成员设为 **null**，那么你根本就不需要实现 **IDisposable**。为什么呢？因为把引用设为 **null** 并不等于 C 语言里面的 **free**，它并不能立即回收那份内存，就算你的对象里面有一个很大的数组也一样。我发现有些 C# 程序员喜欢在使用对象之后把引用赋值为 **null**，就像这样写代码：

```
void foo()
{
    BigObject x = new BigObject();
    // ...
    // 使用 x 指向的对象 ...
    // ...
    x = null;
}
```

x = null 是毫无意义的。写出这样的代码，说明他们不明白 GC 是如何工作的，以为把引用设为 **null** 就可以释放内存，以为不把引用设为 **null**，内存就不会被回收！再进一步，如果你仔细看 **HashAlgorithm** 的源代码，就会发现 **HashValue** 这个成员数组其实没有必要存在，因为它保存的只是上一次调用 **ComputeHash()** 的结果而已。这种保存结果的事情，本来应该交给使用者去做，而不是包揽到自己身上。这个数组的存在，还导致你没法重用同一个 **HashAlgorithm** 对象，因为有共享的成员 **HashValue**，所以不再是 **thread safe** 的。

其实在 C# 里面，你没有办法可以手动回收内存，因为内存是由 GC 统一管理的。就算你实现 **Dispose**，在里面把成员设置为 **null**，内存也只有等下次 GC 执行的时候才可能被回收。举一个例子：

```
class Foo : IDisposable
{
    private byte[] _data = new byte[1000000000];

    public void Dispose()
    {
        _data = null;      // 没用的
    }
}
```

在这个例子里面，**Foo** 类型的 **Dispose** 只是在把 **_data** 设为 **null**，这是毫无意义的。如果你想释放掉这块数组，那么你只需要等不再有人使用 **Foo** 对象。比如：

```
void UseFoo()
{
    Foo foo = new Foo();
    // 使用 f...
    foo.Dispose();   // 没必要
    foo = null;      // 没必要
}
```

这里的 **foo.Dispose()** 是完全不必要的。你甚至没必要写 **foo = null**，因为 **foo** 是一个局部变量，它一般很快就会离开作用域的。当函数执行完毕，或者编译器推断 **foo** 不会再次被使用的时候，GC 会回收整个 **Foo** 对象，包括里面的巨大数组。

所以正确的做法应该是完全不要 Dispose，不实现 IDisposable 接口。有些人问，要是 Foo 对象被放进一个全局哈希表之类的数据结构，GC 没法释放它，就需要 Dispose 了吧？这也是一种常见的误解。如果你真要回收全局哈希表里的 Foo 对象，你只需要把 Foo 对象从哈希表里面删掉就可以了。一旦哈希表对 Foo 对象的引用没有了，GC 运行的时候就会发现它成了垃圾，里面的 _data 数组自然也是垃圾，所以一起就回收掉了。

所以简言之，Dispose 不是用来给你回收内存用的。在 Dispose 方法里把成员设为 null，并不会导致更快的内存释放。有人可能以为 HashAlgorithm 是为了“安全”考虑，所以在 Dispose 方法里对数组清零。然而 IDisposable 是用于释放“资源”的接口，把安全清零这种事情放在这个接口里面，反而会让人误解，造成疏忽。

而且从源代码里的注释看来，HashAlgorithm 的这个方法确实是为了解放资源，而不是为了什么安全考虑。这些库代码实现 IDisposable，意味着这个接口会通过这些库代码不必要的传递到用户代码里面去，导致很多不知情用户的代码被迫实现 IDisposable，造成“传染”。

作为练习，你可以分析一下 MemoryStream 的 Dispose 方法，为什么是没必要的：

```
protected override void Dispose(bool disposing)
{
    try
    {
        if (disposing)
        {
            _isOpen = false;
            _writable = false;
            _expandable = false;
#if FEATURE_ASYNC_IO
            _lastReadTask = null;
#endif
        }
    }
    finally
    {
        // Call base.Close() to cleanup async IO resources
        base.Dispose(disposing);
    }
}
```

另外，我发现 AutoResetEvent，ManualResetEvent，ReaderWriterLockSlim，Semaphore 这些 IDisposable 对象，里面的所谓“资源”，归根结底都是一些很小的 Windows event 对象，而且它们都继承了 SafeHandle。SafeHandle 本身有一个“析构函数”（finalizer），它看起来是这个样子：

```
~SafeHandle()
{
    Dispose(false);
}
```

当 SafeHandle 被 GC 回收的时候，GC 会自动调用这个析构函数，进而调用 Dispose。也就是说，你其实并不需要手动调用这些对象（例如 ManualResetEvent，Semaphore 之类）的 Dispose 方法，因为 GC 会调用它们。这些对象占用资源不多，系统里也不会有很多这种对象，所以 GC 完全应该有能力释放它们占用的系统资源。

文件的特殊性质

很多人谈到这个问题，就会举文件的例子来反驳你，说：“你不应该依靠 GC 来释放 IDisposable 对象。你应该及时关闭文件，所以对于其它 IDisposable 资源，也应该及时关闭，不应该等 GC 来释放它。”这些人没有抓住问题的关键，所以他们把文件和其它 IDisposable 资源一概而论。

文件是一种很特殊的资源，它和其它 IDisposable 对象是很不一样的。你之所以需要在用完一个文件之后立即关掉它，而不能等 GC 来做这事，是因为文件是一种隐性的“全局资源”。这种“全局”，是从程序语言语义的角度来看的。文件很像程序里的全局变量，无论从什么地方都可以访问。

使用文件的时候，你使用文件名来读写它，任何知道这个名字的进程都可以访问这个文件。（我们这里忽略权限之类的问题，那跟语义是不相关的。）这使得文件成为一种“全局资源”，也就是说它不是 thread safe 的。在并发系统里面，在任何一个时刻，只能有一个进程打开文件进行写操作。然后这个文件就被它“锁住”了，其它进程不能打开，否则就会出现混乱。所以如果这个进程不及时关掉文件，其它人就没法用它。

写文件其实是给它加了锁，当然你必须及时进行解锁，而不能等 GC 这种非实时的方式来帮你解锁。否则即使你不再引用这个文件，其他人仍然没法立即进入锁定的区域，这就造成了不必要的等待。所以文件的所谓“打开”和“关闭”操作，本质上隐含了加锁和解锁操作。

文件是很特殊的资源。系统里的大部分其它资源，都不像文件这样是共享的，而是分配给进程“私人使用”的。系统里面可以有任意多个这样的资源，你用任何一个都可以，它们的使用互不干扰，不需要加锁，所以你并不需要非常及时的关闭它们。这种资源的性质，跟内存的性质几乎完全一样。

像 C# 里的 ManualResetEvent, Semaphore, ReaderWriterLockSlim 就属于这种非共享资源，它们的性质跟内存非常相似。就算它们实现了 IDisposable 接口，关闭它们的重要性也跟关闭文件相差非常大。我通过测试发现，就算你把它们完全交给 GC 处理，也不会有任何问题。无论你是否调用它们的 Dispose 方法，系统性能都一模一样。只不过如果你调用 Dispose，计算花的时间还要稍微多一些。

官方文档和 Roslyn 静态分析不可靠

微软官方文档和 Roslyn 静态分析说一定要调用 Dispose，其实是把不是问题的问题拿出来，让没有深入理解的人心惊胆战。结果把代码给搞复杂了，进而引发更严重的问题。很多人把 Roslyn 静态分析的结果很当回事，而其实看了 Roslyn 静态分析的源代码之后，我发现他们关于 Dispose 的静态分析实现，是相当幼稚的作法。基本的流分析 (flow analysis) 都没有，靠肤浅的表象猜测，所以结果是非常不准确的，导致很多 false positive。回忆一下我的 PySonar 全局流分析，以及我在 Coverity 是干什么的，你就知道我为什么知道这些 ;-)

另外 Roslyn 分析给出的警告信息，还有严重的误导性质，会导致一知半解的人过度紧张。比如编号为 [CA1001](#) 的警告对你说：“Types that own disposable fields should be disposable。”如果你严格遵循这一“条款”，让所有含有 IDisposable 的成员的类都去实现 IDisposable，那么 IDisposable 接口就会从一些很小的对象（比如常见的 ManualResetEvent），很快扩散到其它用户对象里去。许多对象都实现 IDisposable 接口，却没有任何对象真正的调用 Dispose 方法。最终结果跟你什么都不做是一样的，只不过代码变复杂了，还浪费了时间和精力。

C 编译器优化的 Bug

一个朋友向我指出一个最近他们发现的 GCC 编译器优化过程（加上 -O3 选项）里的 bug，导致他们的产品出现诡异的行为。这使我想起以前见过的一个 GCC 的 bug。当时很多编译器专家认为那种做法是正确的，跟他们说不清楚。简言之，这种有问题的优化，喜欢利用 C 语言的“未定义行为”（undefined behavior）进行推断，最后得到奇怪的优化结果。

这类优化过程的推理方式都很类似，他们使用一种看似严密而巧妙的推理，例如：“现在有一个整数 x ，不知道是多少。但 x 出现在一个条件语句里面，如果 $x > 1$ ，那么程序会进入未定义行为，所以我们可以断定 x 的值必然小于或者等于 1，所以现在我们利用 $x \leq 1$ 这个事实来对相关代码进行优化……”

看似合理，却是不正确的。举一个具体的例子吧。这篇 Chris Lattner 写于 2011 年的[文章](#)里面就含有这样一个优化。文中指出，编译器利用“未定义行为”进行优化，是合理的，对于性能是很重要的，并且举出这样一个例子：

```
void contains_null_check(int *P) {
    int dead = *P;
    if (P == 0)
        return;
    *P = 4;
}
```

这例子跟我之前看到的 GCC bug 不大一样，但大致是类似的推理方式：这个函数依次经过这样两个优化步骤（RNCE 和 DCE），之后得出“等价”的代码：

```
void contains_null_check_after_RNCE(int *P) {
    int dead = *P;
    if (false) // P 在上一行被访问，所以这里 P 不可能是 null
        return;
    *P = 4;
}

void contains_null_check_after_RNCE_and_DCE(int *P) {
    // int dead = *P; // 死代码
    // if (false) // 死代码
    //     return; // 死代码
    *P = 4;
}
```

他的推理方式是这样：

1. 首先，因为在 `int dead = *P` 里面，指针 P 的地址被访问，如果程序顺利通过了这一行而没有出现未定义行为（比如当掉），那么之后 P 就不可能是 `null`，所以我们可以把 $P == 0$ 优化为 `false`。
2. 因为条件是 `false`，所以整个 `if` 语句都是死代码，被删掉。
3. `dead` 变量赋值之后，没有被任何其它代码使用，所以对 `dead` 的赋值是死代码，可以消去。

最后函数就只剩下一行代码 `*P = 4`。然而经我分析，发现这个转换是错误的，而不只是像他说的“存在安全隐患”。现在我来考考你，你知道这为什么是错的吗？庆幸的是，现在如果你把这代码输入到 Clang，就算加上 -O3 选项，它也不会给你进行这个优化。这也许说明他们也许已经意识到了这个错误。

我写这篇文章的目的其实是想告诉你，不要盲目相信编译器的优化变换都是正确的。无论它看起来多么的合理，只要打开优化之后你的程序出现不合理的行为，你就不能排除编译器进行了错误优化的可能性。Lattner 指出这样的优化完全符合 C 语言的标准，但就算你符合国际标准，也有可能是错的。有时候你得相信自己的直觉……

对 Rust 语言的分析

Rust 是一门最近比较热的语言，有很多人问过我对 Rust 的看法。由于我本人是一个语言专家，实现过几乎所有的语言特性，所以我不认为任何一种语言是新的。任何“新语言”对我来说，不过是把早已存在的语言特性（或者毛病），挑一些出来放在一起。所以一般情况下我都不会去评论别人设计的语言，甚至懒得看一眼，除非它历史悠久（比如像 C 或者 C++），或者它在工作中惹恼了我（像 Go 和 JavaScript 那样）。这就是为什么这些人问我 Rust 的问题，我一般都没有回复，或者一笔带过。

不过最近有点闲，我想既然有人这么热衷于这种新语言，那我还是稍微凑下热闹，顺便分享一下我对某些常见的设计思路的看法。所以这篇文章虽然是在评论 Rust 的设计，它却不只是针对 Rust。它是针对某些语言特性，而不只是针对某一种语言。

由于我这人性格很难闭门造车，所以现在我只是把这篇文章的开头发布出来，边写边更新。所以你要明白，这只是一个开端，我会按自己理解的进度对这篇文章进行更新。你看了之后，可以隔一段时间再回来看新的内容。如果有特别疑惑的问题，也可以发信来问，我会汇总之后把看法发布在这里。

变量声明语法

Rust 的[变量声明](#)跟 Scala 和 Swift 的很像。你用

```
let x = 8;
```

这样的构造来声明一个新的变量。大部分时候 Rust 可以推导出变量的类型，所以你不一定需要写明它的类型。如果你真的要指明变量类型，需要这样写：

```
let x: i32 = 8;
```

在我看来这是丑陋的语法。本来语义是把变量 x 绑定到值 8，可是 x 和 8 之间却隔着一个“i32”，看起来像是把 8 赋值给了 i32.....

变量缺省都是不可变的，也就是不可赋值。你必须用一种特殊的构造

```
let mut x = 8;
```

来声明可变变量。这跟 Swift/Scala 的 let 和 var 的区别是一样的，只是形式不大一样。

变量可以重复绑定

Rust 的变量定义有一个比其它语言更奇怪的地方，它可以让你在同一个作用域里面“重复绑定”同一个名字，甚至可以把它绑定到另外一个类型：

```
let mut x: i32 = 1;
x = 7;
let x = x; // 这两个 x 是两个不同的变量

let y = 4;
// 30 lines of code ...
let y = "I can also be bound to text!";
// 30 lines of code ...
println!("y is {}", y); // 定义在第二个 let y 的地方
```

在 Yin 语言最初的设计里面，我也是允许这样的重复绑定的。第一个 y 和 第二个 y 是两个不同的变量，只不过它们碰巧叫同一个名字而已。你甚至可以在同一行出现两个 x，而它们其实是不同的变量！这难道不是一个很酷，很灵活，其他语言都没有的设计吗？后来我发现，虽然这实现起来没什么难度，可是这样做不但没有带来更大的方便性，反而可能引起程序的混淆不清。在同一个作用域里面，给两个不同的变量起同一个名字，这有什么用处呢？自找麻烦而已。

比如上面的例子，在下面我们看到一个对变量 y 的引用，它是在哪里定义的呢？你需要在头脑中对程序进行“数据流分析”，才能找到它定义的位置。从上面读起，我们看到 let y = 4，然而这不一定是正确的定义，因为 y 可以被重新绑定，所以我们必须继续往下看。30 行代码之后，我们看到了第二个对 y 的绑定，可是我们仍然不能确定。继续往下扫，30 行代码之后我们到了引用 y 的地方，没有再看到其它对 y 的绑定，所以我们才能确信第二个 let 是 y 的定义位置，它是一个字符串。

这难道不是很费事吗？更糟的是，这种人工扫描不是一次性的工作，每次看到这个变量，你都要疑惑一下它是什么东西，因为它可以被重新绑定，你必须重新确定一下它的定义。如果语言不允许在同一个作用域里面重复绑定同一个名字，你就根本不需要担心这个事情了。你只需要在作用域里面找到唯一的一个 let y = ...，那就是它的定义。

也许你会说，只有当有人滥用这个特性的时侯，才会导致问题。然而语言设计的问题往往就在于，一旦你允许某种奇葩的用法，就一定会有人自作聪明去用。因为你无法确信别人是否会那样做，所以你随时都得提高警惕，而不能放下心情来。

类型推导

另外一个很多人误解的地方是类型推导。在 Rust 和 C# 之类 的语言里面，你不需要像 Java 那样写

```
int x = 8;
```

这样显式的指出变量的类型，而是可以让编译器把类型推导出来。比如你写：

```
let x = 8; // x 的类型推导为 i32
```

编译器的类型推导就可以知道 x 的类型是 i32，而不需要你把“i32”写在那里。这似乎是一个很方便的东西。然而看过很多 C# 代码之后你发现，这看似方便，却让程序变得不好读。在看 C# 代码的时候，我经常看到一堆的变量定义，每一个的前面都是 var。我没法一眼就看出它们表示什么，是整数，bool，还是字符串，还是某个用户定义的类？

```
var correct = ...;
var id = ...;
var slot = ...;
var user = ...;
var passwd = ...;
```

我要把鼠标移到变量上面，让 Visual Studio 显示出它推导出来的类型，可是鼠标移开之后，我可能又忘了它是什
么。有时候发现看同一片代码，都需要反复的做这件事，鼠标移来移去的。而且要是没有 Visual Studio，用其它编辑器，或者在 github 上看代码或者 code review 的时候，你就得不到这种信息了。很多 C# 程序员为了避免这个问题，开始用很长的变量名，把类型的名字加在变量名字里面去，这样一来反而更复杂了，却没有想到直接把类型写出来。所以这种形式的类型推导，看似先进或者方便，其实还不如直接在声明处写下变量的类型，就像 Java 那样。

所以，虽然 Rust 在变量声明上似乎有更灵活的设计，然而我觉得 C 和 Java 之类 的语言那样看似死板的方式其实更好。我建议不要使用 Rust 变量的重复绑定，避免使用类型推导，尽量明确的写出类型，以方便读者。如果你真的在乎代码的质量，就会发现大部分时候你的代码的读者是你自己，而不是别人，因为你需要反复的阅读和提炼你的代码。

动作的“返回值”

Rust 的文档说它是一种“[大部分基于表达式](#)”的语言，并且给出这样一个例子：

```
let mut y = 5;
let x = (y = 6); // x has the value `()`，not `6`
```

奇怪的是，这里变量 x 会得到一个值，空的 tuple，()。这种思路不大对，它是从像 OCaml 那样的语言照搬过来的，而 OCaml 本身就有问题。在 OCaml 里面，如果你使用 print_string，那你会得到如下的结果：

```
print_string "hello world!\n";;

hello world!
- : unit = ()
```

这里，print_string 是一个“动作”，它对应程式语言里面的“statement”。就像 C 语言的 printf。动作通常只产生“副作用”，而不返回值。在 OCaml 里面，为了“理论的优雅”，动作也会返回一个值，这个值叫做 ()。其实 () 相当于 C 语言的 void。C 语言里面有 void 类型，然而它却不允许你声明一个 void 类型的变量。比如你写

```
int main()
{
    void x;
}
```

程序是没法编译通过的（试一试？）。让人惊讶的是，古老的 C 的做法其实是正确的，这里有比较深入的原因。如果你把一个类型看成是一个集合（比如 int 是机器整数的集合），那么 void 所表示的集合是个空集，它里面是不含有任何元素的。声明一个 void 类型的变量是没有任何意义的，因为它不可能有一个值。如果一个函数返回 void，你是没法把它赋值给一个变量的。

可是在 Rust 里面，不但动作（比如 y = 6）会返回一个值 ()，你居然可以把这个值赋给一个变量。其实这是错误的作法。原因在于 y = 6 只是一个“动作”，它只是把 6 放进变量 y 里面，这个动作发生了就发生了，它根本不应该返回一个值，它不应该出现在 let x = (y = 6); 的右边。就算你牵强附会说 y = 6 的返回值是 ()，这个值是没有任何用处的。更不要说使用空的 tuple 来表示这个值，会引起更大的类型混淆，因为 () 本身有另外的，更有用的含义。

你根本就不应该可以写 `let x = (y = 6);` 这样的代码。只有当你犯错误或者逻辑不清晰的时候，才有可能把 `y = 6` 当成一个值来用。Rust 允许你把这种毫无意义的返回值赋给一个变量，这种错误就没有被及时发现，反而能够通过变量传播到另外一个地方去。有时候这种错误会传播挺远，然后导致问题（运行时错误或者类型检查错误），可是当它出问题的时候，你就不大容易找到错误的起源了。

这是很多语言的通病，特别是像 JavaScript 或者 PHP 之类的语言。它们把毫无意义或者牵强附会的结果（比如 `undefined`）到处传播，结果使错误很难被发现和追踪。

return 语句

Rust 的设计者似乎很推崇“面向表达式”的语言，所以在 Rust 里面你不需要直接写“return”这个语句。比如，这个例子里面，你可以直接这样写：

```
fn add_one(x: i32) -> i32 {  
    x + 1  
}
```

返回函数里的最后一个表达式，而不需要写 `return` 语句，这是函数式语言共有的特征。然而其实我觉得直接写 `return` 其实是更好的作法，像这个样子：

```
fn foo(x: i32) -> i32 {  
    return x + 1;  
}
```

编程有一个容易引起问题的作法，叫做“不够明确”，总想让编译器自动去处理一些问题，在这里也是一样的问题。如果你隐性的返回函数里最后一个表达式，那么每一次看见这个函数，你都必须去搞清楚最后一个表达式是什么，这并不是每次都那么明显的。比如下面这段代码：

```
fn main() {  
    println!("{}", add_one(7));  
}  
  
fn add_one(x: i32) -> i32 {  
    if (x < 5) {  
        if (x < 10) {  
            // 做很多事...  
            x * 2  
        } else {  
            // 做很多事...  
            x + 1  
        }  
    } else {  
        // 做很多事...  
        x / 2  
    }  
}
```

由于 `if` 语句里面有嵌套，每个分支又有好些代码，而且 `if` 语句又是最后一个语句，所以这个嵌套 `if` 的三个出口的最后一个表达式都是返回值。如果你写了“`return`”，那么你可以直接看有几个“`return`”，或者拿编辑器加亮一下，就知道这个函数有几个出口。然而现在没有了“`return`”这个关键字，你就必须把最后那个 `if` 语句自己看清楚了，找到每一个分支的“最后表达式”。很多时候这不是那么明显，你总需要找一下，而且这件事在读代码的时候总是反复做。

所以对于返回值，我的建议是总是明确的写上“`return`”，就像第二个例子那样。Rust 的文档说这是“poor style”，那不是真的。有一个例外，那就是当函数体里面只有一条语句的时候，那个时候没有任何歧义哪一个是返回表达式。

这个问题类似于重复绑定变量和类型推导的问题，属于一种“用户体验设计”问题。无论如何，编译器都很容易实现，然而不同样式的代码，对于人类阅读的工作量，是很不一样的。很多时候最省人力的做法并不是那种看来最聪明，最酷，打字量最少的办法，而是写得最明确，让读者省事的办法。人们常说，代码读的时候比写的时候多得多，所以要想语言好用省事，我们应该更加重视读的时候，而不是写的时候。

数组的可变性

Rust 的数组可变性标记，跟 Swift 犯了一样的错误。Swift 的问题，我已经在之前的[文章](#)有详细叙述，所以这里就不多说了。简言之，同一个标记能表示的可变性，要么针对数组指针，要么针对数组元素，应该只能选择其一。而在 Rust 里面，你只有一个地方可以放“`mut`”进去，所以要么数组指针和元素全部都可变，要么数组指针和元素都不可变。你没有办法制定一个不可变的数组指针，而它指向的数组的元素却是可变的。

请对比下面两个例子：

```
fn main() {  
    let m = [1, 2, 3];      // 指针和元素都不可变
```

```

m[0] = 10;           // 出错
m = [4, 5, 6];       // 也出错
}

fn main() {
    let mut m = [1, 2, 3]; // 指针和元素都可变
    m[0] = 10;             // 不出错
    m = [4, 5, 6];         // 也不出错
}

```

内存管理

Rust 号称实现了非常先进的内存管理机制，不需要垃圾回收 (GC) 或者引用计数 (RC) 就可以“静态”的管理内存的分配和释放。然而仔细思考之后你就会发现，这很可能是不切实际的梦想（或者广告）。内存的分配和释放（如果要及时释放的话），本身是一个动态的过程，无法用静态分析来实现。现在你说可以通过一些特殊的构造，特殊的指针和传值方式，静态的决定内存的回收时间，真的有可能吗？

实际上我有一个类似的梦。我曾经向我的教授们提出过 N 多种不需 GC 和 RC 就能静态管理内存的办法，结果每一次都被他们给我的小例子给打败了，以至于我很难相信有任何人可以想到比 GC 和 RC 更好的方法。

Rust 那些炫酷的 move semantics, borrowing, lifetime 之类的概念加在一起，不但让语言变得复杂不堪，我感觉并不能从根本上解决内存管理问题。很多人在 blog 里面为这些概念热情洋溢地做宣传，显得自己很懂一样，拿一些玩具代码来演示，可是从没看到任何人说清楚这些东西为什么可以从根本上解决问题，能用到复杂一点的代码里面去。所以我觉得这些东西有“皇帝的新装”之嫌。

连 Rust 自己的[文档](#)都说，你可能需要“fight with the borrow checker”。为了通过这些检查，你必须用很怪异的方式来写程序，随着问题复杂度的增加，就要求有更怪异的写法。如果用了 lifetime，很简单一个代码看起来就会是这种样子。真够烦的，我感觉我的眼睛都没法 parse 这段代码了。

```

fn foo<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
}

```

上一次我看 Rust 文档的时候，没发现有 lifetime 这概念。文档对此的介绍非常粗略，仔细看了也不知道他们在说些什么，更不要说相信这办法真的管用了。对不起，我根本不想去理解这些尖括号里的 '[a](#) 和 '[b](#) 是什么，除非你先向我证明这些东西真的能解决内存管理的问题。实际上这个 lifetime 我感觉像是跨过程静态分析时产生的一些标记，要知道静态分析是无法解决内存管理的问题的，我猜想这种 lifetime 在有递归函数的情况下就会遇到麻烦。

实际上我最开头看 Rust 的时候，它号称只用 move semantics 和好几种不同的指针，就可以解决内存管理的问题。可是一旦有了那几种不同的指针，就已经复杂不堪了，比 C 语言还要麻烦，而且显然不能解决问题。Lifetime 恐怕是后来发现有新的问题解决不了才加进去的，可是我不知道他们这次是不是又少考虑了某些情况。

Rust 的设计者显然受了 [Linear Logic](#) 一类看似很酷的逻辑的启发和熏陶，想用类似的方式奇迹般的解决内存和资源的回收问题。然而研究过一阵子 Linear Logic 之后我发现，这个逻辑自己都没有解决任何问题，只不过给对象的引用方式施加了一些无端的限制，这样使得对象的引用计数是一个固定的值 (1)。内存管理当然容易了，可是这样导致有很多程序你没法表达。

开头让你感觉很有意思，似乎能解决一些小问题。到后来遇到大一点的实际问题的时候，你就发现需要引入越来越复杂的概念，使用越来越奇葩的写法，才能达到目的，而且你总是会在将来某个时候发现它没法解决的问题。因为这个问题很可能从根本上是无法解决的，所以每当遇到有超越现有能力的事情，你就得增加新的“绕过方法” (workaround)。缝缝补补，破败不堪。最后你发现，除了垃圾回收 (GC) 和引用计数 (RC)，内存管理还是没有其它更好更简单的办法。

当然我的意见也许不是完全准确，可我真是没有时间去琢磨这么多乱七八糟，不知道管不管用的概念（特别是 lifetime），更不要说真的用它来构建大型的系统程序了。有用来理解这些概念，把程序改成奇葩样子的时间，我可能已经用 C 语言写出很好的手动内存管理代码了。如果你真的看进去理解了，发现这些东西可以用的话，告诉我一声！不过你必须说明原因，不要只告诉我“皇帝是穿了衣服的” :P

完

本来想写一个更详细的评价的，可是到了这个地方，我感觉已经失去兴趣了，困就一个字啊…… Rust 比 C 语言复杂太多，我很难想象用这样的语言来构造大型的操作系统。而构造系统程序，是 Rust 设计的初衷。说真的，写操作系统那样的程序，C 语言真的不算讨厌。用户空间的程序，Java，C# 和 Swift 完全可以胜任。所以我觉得 Rust 的市场空间恐怕非常狭小……

(如果你喜欢这些内容，请付费5美元或者30人民币，谢谢！)

测试的道理

在长期的程序语言研究和实际工作中，我摸索出了一些关于测试的道理。然而在我工作过的每一个公司，我发现绝大多数人都不明白这些道理，很多团队集体性的采用错误的做法而不自知。很多人把测试当成一种主义和教条，进行过度的测试，不必要的测试，不可靠的测试，并且把这些错误的做法传授给新手，造成恶性循环。本来目的是提高代码质量，结果不但没能达到目的，反而降低了代码质量，增大了工作量，大幅度延缓工程进度。

我也写测试，但我的测试方式比“测试教条主义者”们的方式聪明很多。在我心目中，代码本身的地位大大的高于测试。我不忽视测试，但我不会本末倒置，过分强调测试，我并不推崇测试驱动开发（TDD）。我知道该测试什么，不该测试什么，什么时候该写测试，什么时候不该写，什么时候应该推迟测试，什么时候完全不需要测试。因为这个原因，再加上高强的编程能力，我多次完成别人认为在短时间不可能完成的任务，并且制造出质量非常高的代码。

测试的道理

现在我就把这些自己领悟到的关于测试的道理总结一下，其中有一些是鲜为人知或者被误解的。

1. 不要以为你处处显示出“重视代码质量”的态度，就能提高代码质量。总有些人，以为自己知道“单元测试”（unit test），“集成测试”（integration test）这样的名词，就很懂编程，就可以教育其他人。可惜，光有态度和口号是不解决问题的，你还必须有实战的技巧，深入的见解和智慧，必须切实地知道应该怎么做。代码的质量不会因为你重视它就得到提升，也不会因为你采取了措施（比如测试，静态分析）就一定会得到改善。你必须知道什么时候该写测试，什么时候不该写测试，需要写测试的时候，要写什么样的测试。其实，提高代码质量唯一可行的手段不是写测试，而是反复的提炼自己的思维，写简单清晰的代码。如果你想真的提高代码质量，我的文章『[编程的智慧](#)』是一个不错的出发点。
2. 真正的编程高手不会被测试捆住手脚。是的，你身边那个你认为“不很在乎测试”的家伙，也许是个比你更好的程序员。我喜欢把编程比喻成开赛车，而测试就是放在路边用来防撞的轮胎护栏……



护栏有时候是很有用，可以救命的，然而一个合格的车手，绝对不会一心想着有护栏保护，测试在编程活动中的地位也应该就是这样。优秀的车手会很快看见优雅而简单的路径，恰到好处地掌握速度和时机，直奔终点而去。护栏只是放在最危险的地段，让你出了意外不要死得太惨。护栏并不能让你成为好的车手，不能让你取得冠军。绝大多数时候，你的安全只有靠自己的技术，而不是护栏，你永远有办法可以撞死自己。测试的作用也是一样，即使有了很多的测试，代码的安全仍然只掌握在你的手里。你永远可以制造出新的 bug，而没有测试可以检测到它……

通常情况下，一个合格的车手是根本碰不到这些护栏的，他们心里想的是更高的目标：快点到达终点。相比之下，一个不合格的车手，他经常撞到赛道外面去，所以在他的心里，护栏有着至高无上的地位，所以他总是跟别人宣扬护栏的重要性。他开车的时候为了防止犯错，要在在他经过的路径两边密密麻麻摆上护栏，甚至把护栏摆到赛道中间，以确保自己的转弯幅度正确。他在护栏之间跌跌撞撞，最后只能算是勉强到达终点。鼓吹测试驱动开发的人，就是这种三流车手，这种人写再多的测试也不可能倒腾出可靠的代码来。

3. 在程序和算法定型之前，不要写测试。TDD 的教条者喜欢跟你说，在写程序之前就应该先写测试。为什么写代码之前要写测试呢？这只不过是一种教条。这些人其实没有用自己的脑子思考过这个问题，而只是人云亦云，觉得这样“很酷”，符合潮流，或者以为这样做了别人就会认为自己是高手。实际上在程序框架完成，算法定型之前，你

都不需要写测试。如果你想知道代码是否正确，用人工方式运行代码，看看结果足以。

如果你发现编程初期需要保证的性质纷繁复杂，如此之多，不写测试你就没信心的话，那你还是想办法先提高下基本的编程技术吧：多做练习，简化代码，让代码更加模块化，看看我的『编程的智慧』或者『SICP』一类的东西。写测试并不能提高你的水平，正好相反，过早的写测试会捆住你的手脚，让你无法自由的修改代码和算法。如果你不能很快的修改代码，不能用直觉感觉到它的变化和结构，而是因为测试而处处卡顿，你的头脑里就不能产生所谓“flow”，就不能写出优雅的代码来，结果到最后你什么也没学会。只有在程序不再需要大幅度的改动之后，才是逐渐加入测试的时候。

4. 不要为了写测试而改变本来清晰的编程方式。很多人为了满足“覆盖”（coverage）的要求，为了可以测试到某些模块，或者为了使用 mock，而把本来简单清晰地代码改成更加复杂而混淆的形式，甚至采用大量 reflection。这样一来其实降低了代码的质量。本来很简单的代码，一眼看去就知道是否正确，可是现在你一眼看过去，到处都是为了方便测试而加进去的各种转接插头，再也无法感觉到代码。这些用来辅助测试的代码，阻碍了你对代码进行直觉思维，而如果你不能把代码的逻辑完全映射在头脑里（进而产生直觉），你是很难写出真正可靠的代码的。

有些 C# 程序员，为了测试而加入大量的 interface 和 reflection，因为这样可以在测试的时候很方便的把一片代码替换成 mock。结果你就发现这程序里每个类都有一个配套的 interface，还需要写另外一个 mock 类，去实现这个 interface。这样一来，不但代码变得复杂难以理解，而且还损失了 Visual Studio 的协助功能：你不能再按一个键（F12）就直接跳转到方法的定义，而需要先跳到对应的 interface 方法，然后再找到正确的实现。所以你不再能够在代码里面快速的跳转浏览。这种方便性的损失，会大幅度降低头脑产生整体理解的机会。而且为了 mock，每一个构造函数调用都得换成一个含有 reflection 的构造，使得编译器的静态类型检查无法确保类型正确，增加运行时出错的可能性，出错信息还难以理解，得不偿失的后果。

5. 不要测试“实现细节”，因为那等同于把代码写两遍。测试应该只描述程序需要满足的“基本性质”（比如 sqrt(4) 应该等于 2），而不是去描述“实现细节”（比如具体的开平方算法的步骤）。有些人的测试过于详细，甚至把代码的每个实现步骤都兢兢业业的进行测试：第一步必须做A，第二步必须做B，第三步必须做C…… 还有些人喜欢给 UI 写测试，他们的测试里经常这样写：如果你浏览到这个页面，那么你应该在标题栏看见这行字……

仔细想一下就会发现，这种作法本质上不过是把代码（或者UI）写了两遍而已。本来代码里面明白写着：先做 A，再做B，再做C。UI 描述文件里面明白写着：标题栏里面是这些内容。你有什么必要在测试里把它们全都再检查一遍呢？这根本没有增加任何可靠性：你在代码里会犯错，你把同样的逻辑换种形式再写一遍，难道就不会错了吗？

这就像某些脑子秀逗的人，他出门时总是担心门没锁好，关门之后要推推拉拉好几次，确认门是锁上了的。还没走几步，他仍然在怀疑门没锁好，又走回去推推拉拉好几次，却始终不能放心:P 这种做法非但不能保证代码的正确，反而给修改代码制造了障碍。理所当然，你把同一段代码写了两遍，每当要修改代码，你就得修改两次！这样的测试就像紧箍咒一样，把代码压得密不透风。每一次修改代码，都会导致很多测试失败，以至于这些测试都不得不重写。本质上就是把代码修改了两遍，只不过更加痛苦一些。

6. 并不是每修复一个 bug 都需要写测试。很多公司都流传一个常见的教条，就是认为每修复一个 bug，都需要为它写测试，用于确保这个 bug 不再发生。甚至有人要求你这样修复一个 bug：先写一个测试，重现这个 bug，然后修复它，确保测试通过。这种思维其实是一种生搬硬套的教条主义，它会严重的减慢工程的进度，而代码的质量却不会得到提高。写测试之前，你应该仔细的思考一个问题：这个 bug 有多大可能会在同一个地方再次发生？很多低级错误一旦被看出来之后，它就不大可能在同一个地方再次出现。在这种情况下，你只需手工验证一下 bug 消失了就可以。

为不可能再出现的 bug 大费周折，写 reproducer，构造各种数据结构去验证它，保证它下次不会再出现，其实是多此一举。同样的低级错误就算再出现，也很可能不在同一个地方。写测试不但不能保证它不再发生，而且浪费你很多时间。这测试在每次 build 的时候都会消耗时间，每次编译都因为这些测试多花几分钟，累积起来之后，你就发现工程进度明显减慢。只有当发现已有的测试没有抓住程序必须满足的重要性质时，你才应该写新的测试。你不应该是为这个 bug 而写测试，而是为代码的性质而写测试。这个测试的内容不应该只是防止这个 bug 再次发生，而是要确保 bug 所反映出来的，之前缺失的“性质”得到保证。

7. 避免使用 mock，特别是多层的 mock。很多人写测试都喜欢用很多 mock，堆积很多层，以为只有这样才能测试到路径比较深的模块。其实这样不但非常繁琐费事，而且多层的 mock 往往不能产生足够多样化的输入，不能覆盖各种边界情况。如果你发现测试需要进行多层的 mock，那你应该考虑一下，也许你需要的不是 mock，而是改写代码，让它更加模块化。如果你的代码足够模块化，你不应该需要多层的 mock 来测试它。你只需要为每一个模块准备一些输入（包括边界情况），确保它们的输出符合要求。然后你把这些模块像管道一样连接起来，形成一个更大的模块，测试它也符合输入输出要求，以此类推。
8. 不要过分重视“测试自动化”，人工测试也是测试。写测试，这个词往往隐含了“自动运行”的含义，也就是假设了要不经人工操作，完全自动的测试。打一个命令，它过一会就会告诉你哪些地方有问题。然而，人们往往忽略了“人工测试”。他们没有意识到，人工去试验，去观察，也是一种测试。所以你就发现这样的情况，由于自动测试很多时候非常难以构造（比如，如果你要测试一段复杂的交互式GUI代码的响应），很多人花了很多时间，利用各种测试框架和工具，甚至遥控 WEB 浏览器去做一些自动操作，花太多时间却发现各种不可靠，没法测到很多东西。

其实换一个思路，他们只需要花几分钟的时间，就可以用人工的方式观察到很多深入的问题。过分的重视测试自动化的原因，往往在于一个不切实际的假设，他们假设错误会频繁的再次发生，所以自动化了可以省下人的力气。但是其实，一旦一个 bug 被修好，它反复出现的机会不会很大的。过分的要求测试自动化，不但延缓了工程进度，让程序员恼火，效率低下，而且失去了人工测试的精确性。

9. 避免写太长，太耗时的测试。很多人写测试，叽里呱啦很长一串，到后来再看的时候，他已经不记得自己当时想测什么了。有些人本来用很小的输入就可以测试到需要的性质，他却总喜欢给一个很大的输入，下意识的以为这样更加靠谱，结果这测试每次都会消耗大量的 build 时间，而其实达到的效果跟很小的输入没有任何区别。
10. 一个测试只测试一个方面，避免重复测试。有些人一个测试测很多内容，结果每次那个测试失败，都搞不清楚到底是哪个部件出了问题。有些人为了“放心”，喜欢在多个测试里面“附带”测某些他认为相关的部件，结果每次那个部件出问题，就发现好多个测试失败。如果一个测试只测一个方面，不重复测同一个部件，那么你就可以很快的根据失败的测试，发现出问题的部件和位置。
11. 避免通过比较字符串来进行测试。很多人写测试的时候，喜欢通过打印出一些东西，然后使用字符串比较的方式来决定输出是否符合要求。一个常见的做法是把输出打印成格式化的 JSON，然后对比两个文本。甚至有人 JSON 都不用，直接就比较 printf 输出的结果。这种测试是非常脆弱的。因为字符串输出的格式往往会发生微小的变化，比如有人在里面加了一个空格之类的。把这种字符串作为标准输出，进行字符串比较，很容易因为微小的改动而使大量测试失败，导致很多的测试需要做不必要的修改。正确的做法，应该是进行结构化的比较，如果你要把标准结果存成 JSON，那么你应该先 parse 出 JSON 所表示的对象，然后再进行结构化的对比。PySonar2 的测试就是这样的做法，所以相当的稳定。
12. “测试能帮助后来人”的误区。每当指出测试教条主义的错误，就会有人说：“测试不是为了你自己，而是为了你走了以后，以后进来的人不犯错误。”首先，这种人根本没有看清楚我在说什么，因为我从来没有反对过合理的测试。其次，这种“测试能帮助后来人”，其实是没有经过实践检验，站不住脚的说法。如果你的代码写得很乱，就算你测试再多，后来人也无法理解，反倒被莫名其妙的测试失败给弄得更糊涂，不知道是自己错了还是测试错了。我已经说过了，测试不能完全保证代码不被改错，实际上它们防止代码被改错的作用是非常弱的。无论如何，后来人都必须理解原来的代码的逻辑，知道它在做什么，否则他们不可能做出正确的修改，就算你有再严密的测试也一样。

举一个亲身的例子。我在 Google 做出 PySonar 之后，最后一个测试都没写。第二次我回到 Google，我的上司 Steve Yegge 对我说：“你走了之后，我改了一些你的代码，真是太清晰，太好把握了，修改你的代码是一种快乐！”这说明什么问题呢？我并不是说你可以不写测试，但这个例子说明，测试对于后来人的作用，并不是你有些人想象的那么大。创造清晰的代码才是解决这个问题的关键。

这种怕人突然走了，代码无法维护的想法，导致了一些人对测试过分的重视，但测试却不能解决这种问题。相反，如果测试太繁琐，做不必要的测试，反而容易让员工不满，容易走人，去加入在这方面更加有见地的公司。有些公司以为有了测试，就可以随便打发人走，这种想法是大错特错的。你需要明白的一个事情是，代码永远是属于写出它的那个人的，就算有测试也一样。如果核心人物真的走了，就算你有再多的测试也没用的，所以解决的方法就是把他们留住！一个有远见的公司总是通过其他的手段解决这个问题，比如优待和尊重员工，创造良好的氛围，使得他们没那么快想走。另外，公司必须注意知识的传承，防止某些代码只有一个人理解。

案例分析

有人会疑问，我凭什么可以给别人讲这些经验，我自己为此有什么成功的案例呢？所以现在来讲讲我做过的几个东西，以及我亲眼目睹的测试教条主义者们的失败案例。

Google

很多人可能听说过我在 [Google](#) 做的 PySonar。当时 Google 的队友们战战兢兢，说这么高难复杂的东西要从头做起，几乎是不可能的。特别是某位队友，一开头就吵着要我写测试，一直吵到最后，烦死我了。他们为什么这么担心呢？因为对 Python 做类型推导是非常高难度的代码，需要相当复杂的数据结构和算法，需要精通 Python 的语义实现。

作为一个训练有素的专家，我没有在乎他们的咋呼，没有信他们的教条。我按照自己的方式组织代码，进行精密的思考，设计和推理，最终在三个月之内做出了非常优雅，正确，高性能，而又容易维护的代码。PySonar 到现在仍然是世界上最先进的 Python 类型推导和索引系统，被多家公司采用，用于处理数以百万计的 Python 代码。 ，

如果我当时按照 Google 队友的要求，采用已有的开源代码，或者过早的写了测试，别说无法在三个月的实习时间之内完成这个东西，就算折腾好几年也没有可能。

Shape Security

这种思维方式最近的成功实例，是给 Shape Security 做的一个先进的 JavaScript 混淆器（obfuscator）和对集群（cluster）管理系统的改进。不要小看了这个 JS 混淆器，它的混淆能力要比 uglify 之类的开源工具强很多，也快很多。它不但包含了 uglify 的变量换名等基本功能，而且含有专门针对人类和编译器的复杂化，使得没人能看出一点线索这个程序到底要干什么，让最先进的 JS 编译器也无法把它简化。

其实这个混淆器也是一种编译器，只不过它把 JavaScript 翻译成不可读的形式。在这个项目中，由于失之毫厘就可以差之千里，我采用了从 Chez Scheme 编译器学过来的，非常严密的测试方法。对每一个编译器的步骤（pass），我都给它设计一些正好可以测到这个步骤的输入代码（比如，具有函数定义的，for 循环，try-catch 的，等等）。Pass 输出的代码，经过 JavaScript 解释器执行，把结果跟原来程序的执行结果对比。每一个测试程序，经过每一个 pass，输出的中间结果都跟标准结果进行对比，如果错了就表明那个 pass 有问题，出错的小程序会指出大概是哪一个部分出了问题。遵循小巧，不冗余，不重复的原则，我总共只写了 40 多个非常小的 JavaScript 程序。由于这些测试涵盖了 JavaScript 的所有构造而且几乎不重复，它们能够准确的定位到错误的改动。最后，这个 JS 混淆器能够正确的转换像 AngularJS 那么大的项目，确保语义的正确，让人完全无法读懂，而且能有效地防止被优化器（比如 Closure Compiler）简化掉。

相比之下，过度鼓吹测试和可靠性的人，并没能制造出这么高质量的混淆器。其实在我进入团队之前，里面的两三位高手已经做了一个混淆器，项目延续了好几个月。这片代码一直没能发布给客户用，因为它的换名部件总是在某些情况下输出错误的代码，修改了好多次仍然会出错。不是 100% 的正确，这对于程序语言的转换器来说，是不可接受的。换名只是我的混淆器里的一个步骤，它还包含大概十个类似的步骤，可以把代码进行各种转换。

在实现换名器的时候，队友们让我直接拿他们以前写的换名代码过来，把 bug 修好就可以。然而看了代码之后，我发现这代码没法修，因为它采用了错误的思路，缝缝补补也不可能达到 100% 的正确，而且明显效率低下，所以我决定自己重写一个。由于轻车熟路，我只花了一下午的时间，就完成了一个正确的换名器，它完全符合 JavaScript 的语义，各种奇葩的作用域规则，而且结构非常简单。说白了，这个换名器也是一种 [解释器](#)。对解释器的深刻理解，让我可以很容易的写出任何语言的换名器。

不幸的是，历史再次重演了；）队友们听说我花一下午重写了一个换名器，非常紧张，咋呼地跟我说：“你知道我们的换名器是花了多少个月的时间做出来的吗？你知道我们写了多少测试来保证它的正确性吗？你现在一下午做出来一个新的，你如何能保证它的正确！”我不知道他们怎么好意思说出这样的话来，因为事实是，他们花了这么多个月，耗费这么多人力，写了这么多的测试，做出来的换名器却仍然有 bug，没法用。当我把我写的测试和几个大点的 open source 项目（AngularJS, Backbone 等）放进他们的换名器之后，就发现有些地方出问题了，而所有的测试和 open source 项目通过我的换名器，却得到完全正确的代码。另外经过性能测试，我的换名器速度要快四倍的样子。所以就像 [Dijkstra](#) 所说：“最优雅的程序往往也是最高效的。”

结束这个项目之后，我换了一个团队（cluster 团队），这个团队的人要好很多，低调而且幽默。Shape Security 的产品（Shape Shifter）里面包含一个高可靠（HA）集群管理系统，它可以通过网络，选举 leader，构建一个高容错的并行处理集群。这个集群管理系统一直以来都是公司里很复杂，却是可靠性要求最高的一个部件，一旦出问题就可能有灾难性的后果。确实，它当时可靠性非常高，从来没出过问题。但由于历史原因，它的代码过度复杂而缺乏模块化，以至于很难扩展来应付新的客户需求。我进入这个新团队的任务，就是对它进行大规模的简化，模块化和扩展，让它满足新的需求。

在这个项目中，由于代码的改动幅度很大，在同事和部门领导的理解，信任和支持下，我们决定直接抛弃已有的测试，完全靠严格而及时的 code review，逻辑推理，推敲讨论，手工试验来保证代码的正确。在我修改代码的同时，一位更熟悉已有代码的队友一直通过 git 默默监视着我的每一次改动，根据他自己的经验来判断我的改动是否偏离了原来的语义，及时与我交流和讨论。由于这种灵活而严格的方式，工程不到两个月就完成了。改进后的代码不但更加模块化，更可扩展，适应了新的需求，而且仍然非常可靠。假设部门领导是“测试教条主义者”，不允许抛弃已有的测试，这样的项目是绝对不可能如期完成的。然而在当今世界遇到这样领导的机会，恐怕十个人里面不到一个吧。

Coverity

最后，我举一个由于测试方式不当而非常失败的案例，那就是 Coverity 的 Java 静态分析产品。我承认 Coverity 的 C 和 C++ 分析器也许是非常好的，然而 Java 的分析器，很难说。当我进入 Coverity 的时候，同事们已经忍受了整整一年的管理层的威逼和高压，超时过劳工作，写出了基本的新产品和很多的测试。可是由于技术债太多，再多的测试也没能保证产品的可靠性。

我的任务就是利用我深入的 PL 知识，不停的修补前人留下的各种蹊跷 bug。有些 bug 需要运行 20 多分钟之后才出现，一次还看不出是怎么回事，所以修起来非常耗时。有时候我只好趴在电脑前面养神，时不时的睁眼看看结果。Coverity 是如此的在乎测试，他们要求每修复一个 bug 你就必须写出新的测试。测试必须能够如实的重现 bug 的现象，修复之后测试必须能够通过。这看似一个很在乎代码质量的做法，然而它不但没能保证产品的稳定可靠，而且大幅度的减慢了工程进度，并且造成员工的疲惫和不满。

有一次他们分配给我一个 bug：在分析一个中型项目的时候，分析器似乎进入了死循环，好几个小时都不能完成。因为 Coverity 的全局静态分析，其实就是某种图遍历算法。当这个图里面有回路的时候，你就必须小心，如果不问青红皂白就递归进去，就可能进入死循环。避免死循环的办法很简单，你构造一个图节点的集合（Set），然后把它传递到函数里面作为参数。每当访问一个节点，你先检查这个节点是否已经在这个集合里，如果在你就直接返回，否则你就把这个节点加入到集合里，然后递归处理这个节点的子节点。它的 C++ 代码大概就像这个样子：

```
void traverse(Node node, Set<Node> &visited)
{
    if (visited.contains(node)) {
        return;
    } else {
        visited.add(node);
    }
}
```

```
    process_node(node, visited); // 里面会递归调用 traverse
}
}
```

查看代码之后我发现，代码其实没有进入“死循环”，而是进入了指数复杂度的计算，所以很久都不能完成。这是因为写这函数的人不小心，或者没有理解 C++ 的函数参数缺省是传值（做拷贝）而不是传引用，所以他忘了打那个“&”，所以函数被递归调用的时候不是传递原来的集合，而是做了一个拷贝。每一次递归调用 `traverse`，`visited` 都得到一个新的拷贝，所以返回之后，`visited` 的值就恢复到之前的状态，就像 `node` 被自动 `remove` 了一样。所以这个函数仍然会在某种情况下再次访问这个节点。这样的代码不会进入死循环，然而在某种特殊的图结构下，这会造成指数级的时间复杂度（请想一下这是什么样的一种图）。

本来很明显的一个图论算法问题，加一个“&”就修好了，手工试验也发现问题消失了。然而 Coverity 的测试教条主义者们（包括写出这 bug 的那人自己），吵着闹着，严肃命令我必须写出测试，构造出可以造成这种后果的数据结构，确保这个 bug 不会再重新出现。

为一个我根本不会犯的错误写测试，而且它不可能再次发生，这不是很搞笑吗？就算你写了测试，也不能保证同样的事情不再发生。如果你不小心漏掉“&”，下次同样的问题还会发生，并且发生在另外的地方，而你却没有给那块代码写测试，所以给这个 bug 写测试，并不能防止同样的问题再次发生。这就像一个技术不过关的赛车手，他在别人不大可能撞车的地方撞了车，然后就要求赛场在那个地方装上轮胎护栏。可是下一次，这个车手又会在另一个其他人都不会撞车地方撞车……

稍微有点图论常识，熟悉 C++ 基本概念的人，都不会犯这种错误。防止这种问题，只有靠个人的技术和经验，而不能靠测试。防止它再次发生的最好办法，恐怕是开个会把这个问题讲清楚，让大家理解，下次不要再犯。所以给这个 bug 写测试，完全是多此一举。跟队友们讲解了这个原理，他们听了之后，仿佛什么都没有听到一样，仍然强硬的要求：“可是你还是得写这个测试，因为这是我们的规定！你知道要是出了 bug，送一个销售工程师去客户那里，要花多少钱吗……”无语了。

Coverity 的 Java 分析，就是经常因为这种测试教条主义，使得项目进展及其痛苦和缓慢，却仍然 bug 百出。Coverity 的其他的问题，还包括我上面指出的，写重复的测试，一个测试测太多东西，使用字符串比较来做测试，等等。你恐怕很难想象，一个制造旨在提高代码质量的产品的公司，自己代码的质量是这样维护的 :P

完

由于绝大多数人对测试的误解如此之深，测试教条主义的流毒如此之广，导致许许多多优秀的程序员沉沦在繁琐的测试驱动开发中，无法舒展自己的长处。为了大家有一个轻松，顺利又可靠的工作环境，我希望大家多多转发这篇文章，改变这个行业的陋习。我希望大家在工程中理性的对待测试，而不是盲目的写测试，只有这样才能更好更快的完成项目。

（由于这篇文章包含了我很多年的经验和深入的见解，希望你觉得有收获的话为此付费。建议价格是5美元，或者30人民币。[【付费方式】](#)）

Tesla autopilot 引起致命车祸

好一段时间没关心 Tesla 了，今天才发现他们的 autopilot 终于引起了[致命的车祸](#)。这场 Model S 撞上18轮大卡车的车祸，发生于5月7号，距今已经两个月了。Tesla 把这事隐瞒了两个月之久，直到现在美国国家公路交通安全管理局（NHTSA）开始调查此事，才迫不得已公之于众。由于 Tesla 没有及时向政府监管部门报告事实，政府正在考虑对 Tesla 公司采取法律行动。

本来都懒得再提 Tesla 这公司的名字，但是由于 Tesla 对于这起车祸态度极不端正，不但隐瞒事实，而且继续找各种借口为 autopilot 开脱罪名，让这玩具级别的技术继续危害无辜开车人的安全，很多人（包括新闻机构）对此的分析很多都抓不住关键，所以我不得不再出来说几句。

死者名叫 [Joshua Brown](#)，40岁，曾作为炸弹专家，服役美国海军11年之久。退役以后成立了自己的技术公司，近段时间热衷于 Tesla 的电动车技术，还建立了一个 YouTube 频道，用于演示自己的 Tesla 车子。所以可以说，Joshua 对 Tesla 的 autopilot 使用方法已经很熟悉了。然而这不幸的事件，恰恰就发生在这个专家用户和热心人身上。

Tesla 方面称，那天 Joshua 行驶在佛罗里达州一条中间有隔离带的公路上，符合规定的启用了 autopilot。行车途中，前方有一辆18轮卡车左转，由于卡车车厢是白色的，后面的天空也是白色，所以 autopilot 没发现这个卡车，没有进行刹车，最后 Model S 撞上卡车，车主身亡。白色卡车衬托在白色天空上，所以 autopilot 就把卡车当成空气，这是个什么情况……

先不说这技术有什么问题，出了这种事情，Tesla 对此[反应](#)让人非常的失望。不但没有基本的自我检查，反而各种狡辩，把责任全都推到用户身上。首先，他们从统计的角度，说明 Tesla 车引起死亡的比例，比其它车子小很多。然后旁敲侧击地想说明，就算是那人自己开车，也不能避免这种车祸。最后他们再三的强调，autopilot 的说明书已经声明，功能还不成熟，如果看到要出事而没有及时接管，你们自己负责！

这些都是 Tesla 老一套的诡辩方法。首先，Tesla 的死亡比例比其它车要小，并不能掩盖 autopilot 存在严重问题的事实。死亡比例小可能跟 Tesla 的技术没有很大关系，Tesla 是新公司，车都很新所以不容易出机械故障，而且买 Tesla 的都是有钱人，受过良好的教育，懂技术，所以一般不会乱开。那这种死亡比例，跟老牌子的车比是不公平的。其他牌子的车总数比 Tesla 多太多了，很多车子都十几二十年老掉牙，开车的各种人都有，酒鬼也有，老汉也有，罪犯也有，当然事故比例就上去了。如果你只看其它牌子最近几年的新车和豪华车，死亡比例拿来算一下，就很小。

如果你光看 autopilot 导航的总里程数，事故比例恐怕就上去了，因为很多 Tesla 用户可能没有启用 autopilot，或者用的很少。Autopilot 不是第一次引起车祸了，之前我的[另一篇文章](#)已经提到，由于它的视觉技术不成熟，引发了许多险些发生车祸的情况，而且最近引起了好多次真正的车祸。要知道微小的比例落在一个人头上，就等于100%的不幸。等你因为 autopilot 而受害，才会发现 Tesla 摆出来的那些统计数字，对你其实毫无意义。也许，它确实造福了全人类，可惜死伤的人是你或者你的家人，而且那是因为 autopilot 极其弱智的判断错误…… 你会因为统计数字很安全而饶了 Tesla 吗？

另外 Tesla 喜欢旁敲侧击的指出 autopilot 的驾驶能力高于人类，而事实并不是那样。你怎么能证明人开车不能避免这车祸？Tesla 说：“驾驶员和 autopilot 都没有看到卡车。”你们怎么知道驾驶员没有看见卡车？那可是18轮的大卡车！说白色的侧面车厢映在白色的天空，所以人看不见它，这不是搞笑吗。

一个东西是白色的，不等于它是看不见的，一个不透明的东西会挡住后面的景物，这一点人是很清楚的。白色的物体也会有反光，纹理会跟天空不一样，人可以通过这种反光感知它的存在。卡车不止有白色的侧面，还有黑色的轮子，车头上有烟囱，车窗，油箱，…… 各种其它颜色的附件。为了让其他人在夜间能看到车厢的大小，大卡车必须在车厢的八个角上都安装红色的警示灯，这些灯在白天不亮的时候也看得见的。就算天空是白色，人也是不可能看不见它，把卡车当成空气的。所以我猜真实情况是，驾驶员发现 autopilot 判断错误，想接管过来，但已经来不及了。要知道这个反应时间也许不到一秒！人死了，当然死无对证。

从多次的事故现象中，我分析出这样一个规律，虽然 Tesla 声称 Model S 上装备了雷达和声呐，但是 autopilot 的操作却似乎仅靠摄像头的“像素”，通过神经网络进行图像分析，所以它才会连18轮大卡车这么巨型的东西都没有发现，在路上看到个树影还以为是障碍物…… 这些都是人根本不会犯的奇葩错误。我请大家不要对自动驾驶技术过于乐观，急于求成。机器视觉在某些地方是很有用的技术，然而它要能被用于自动驾驶，还有非常长的路要走。

Tesla 确实警告过人们，说这个技术还不成熟，你必须把手一直放在方向盘上，准备随时接管，然而这并不能免除 Tesla 的责任。首先，Tesla 根本就不应该把不成熟的技术发布出来，而且大肆宣传，搞得大家以为它很先进很可靠的，争相试用。其次，说明书上的警告，在法律上也许是没效力的。你要求别人随时接管，那么你必须在可能判断错误的时候给出警示，而且给人足够的响应时间，才能算是合理。

Autopilot 的设计是有严重问题的。它操纵着车子，却不给人解释自己看见了什么，准备进行什么操作，在道路情况超越了自己能力的时候，也不给人提示，以至于人根本不知道它出了问题，不能及时接管。要知道，车在直走的时候，autopilot 是否判断正确，人往往是看不出来的。一辆没有 autopilot（只有普通 cruise control）的车子，跟一辆启用了 autopilot 的车子，在匀速直线运动的时候，人是无法察觉出任何区别的。可是人知道 autopilot 会自动刹车，而普通的 cruise control 不能，所以人就会期望有 autopilot 的车子会刹车。等你发现它一声不吭，前面有

障碍物却没有刹车，才会知道它有判断错误，可是那个时候就已经晚了。

所以在这种情况下，Tesla 虽然事先有“免责声明”，把责任全都推在用户头上，在法庭上其实仍然可以败诉，因为他们对用户提出的要求是不切实际的，没有人能够在上述 autopilot 判断错误情况下及时的接管过来。我建议这起车祸死者的家属把 Tesla 告上法庭，要求巨额赔偿。我也建议所有 Tesla 的车主，为了对自己和他人的生命负责，请关闭 autopilot 这个功能！Tesla 根本就不懂如何设计自动驾驶系统，技术不过硬，设计有缺陷，基本就是个玩具。生命很宝贵，用自己的生命来给所谓的“新技术”做试验品，是不值得的。

珍爱生命，远离 autopilot !

养生节目的危害

国内总是流行各种各样的“养生节目”，深受中老年人的欢迎。比如我爸妈，有时无聊了，就会转发给我一些养生节目，比如这个：『[多喝白开水带来的危害](#)』。这节目说，有人得了过敏性鼻炎，喷嚏鼻涕不断，严重脱发，头都半秃了，虚弱无力，性能力衰退…… 最后专家得出结论，是因为他每天早上喝一杯凉水导致的！

据我观察，这些养生节目里面的理论，基本可以归为两种：好的和新的。可惜好的理论都不新，新的理论都不好。

好的理论

第一类理论，就是把人们早已熟悉的，久经考验的常识，比如早晚要刷牙之类的，拿来包装成“新理论”。这些理论当然错不到哪里去，然而却是每个人从小都已熟知的。现在挂着“专家”头衔的人出来一宣传，这些斯通见惯的常识，忽然间被老人家们当成了最新的研究成果，惊天动地的发现。

我遇到的这种例子挺多的。有时候父母给我发个信息，说你要注意这个那个习惯啊，不然会得什么什么样的病…… 这本来就是我从小就己经知道并且照办的事情，而且我还记得当年这东西就是爸妈教给我的。现在让这帮“养生专家”一忽悠，倒像个新鲜事，又拿出来讲一遍，好像别人不知道一样，忘了自己几十年前就已经知道……

“专家”的威力就是这么强大！

新的理论

第二类理论，就是胡编乱造出一些“新理论”，却没有经过科学实验证实。『[多喝白开水带来的危害](#)』就属于这一种情况。通常这种理论把问题的原因归结为某一个很小的生活习惯（比如早上喝一杯凉水），而忽略所有其它引起问题的因素。这些理论的问题在于，它们通过臆断，得出错误的“因果关系”。

据我了解，过敏性鼻炎，脱发等问题，很多都是因为基因遗传，跟人平时的生活习惯几乎没有关系。现在有人出现了过敏性鼻炎和脱发，而且碰巧这人早上起来喜欢喝一杯凉水，于是专家就得出一个可笑的结论：一定是喝凉水引起了过敏性鼻炎和脱发。

养生专家们很喜欢把健康问题跟某些不起眼的习惯挂钩，这样就可以创造一些惊人的理论，却无法验证其真实性。他们可以说，就是因为你这个小小的习惯，导致了如此严重的健康问题。这样一来，他们就可以告诉你吃什么，不吃什么，做什么，不做什么。每过一段时间，这些人都会换一套不同的说法，让你感觉有新东西出来，却没发现这些其实跟之前的说法自相矛盾。老年人记性不好，看不出破绽，有些人为了健康不惜一切，仿佛活着就是为了不停地研究如何才能继续活着…… 这就是这帮养生专家和养生节目得以生存的关键。

得出喝凉水引起了鼻炎，脱发，身体虚弱这样的结论，且不说它看起来有没有可能，你必须先经过科学实验。你不能只看一个人，因为数据量太小，很可能是偶然巧合，没法建立因果关系。所以实验必须要有两组人进行对照，就是所谓“对照实验”。一组人早上喝一杯凉水，另外一组人不喝。过一段时间，分析这两组人里面出现上述问题的人的比例，如果喝凉水的人大部分出现了问题，而不喝凉水的人大部分没有出现问题，你才可以得出“喝凉水可能导致鼻炎和脱发”这样的结论。

显然，这个节目里的专家并没有经过实验，而是引用（滥用）『黄帝内经』里面的各种阴阳理论。说早上阳气上升，喝一杯凉水把阳气给浇灭了，怎么能不得病哪！这显然是完全不科学，不负责任的说法。实际上这套阴阳理论是如此的模棱两可，跟占星学如出一辙，你可以利用这些说法来解释世界上的几乎任何现象！不管遇到好事还是坏事，同样的一句话，可以同时支持两种完全相反的结果。为什么它可以这样呢？因为这些说法本来模棱两可，所以不同的人从不同的角度去解释它，发现都是说得通的。

这些养生节目，经常把严重身体问题的起因，归结为某些生活上的小习惯，吃什么，不吃什么之类，很容易让人忽略真正的起因，更加严重的因素，这属于一种误导。中老年人看了这些节目，往往盲目的认为坚持或者改变生活上的一些小习惯，就可以避免或者修正一些严重的身体疾患，结果耽误了真正科学研究出来的补救办法。在这种意义上，养生节目是有危害性的。我建议中老年人少看这种节目，多跟真正的医生了解科学的医疗知识。

Java 有值类型吗？

有人看了我之前的文章『[Swift 语言的设计错误](#)』，问我：“你说 Java 只有引用类型（reference type），但是根据 Java 的[官方文档](#)，Java 也有值类型（value type）和引用类型的区别的。比如 int，boolean 等原始类型就是值类型。”现在我来解释一下这个问题。

Java 有值类型，原始类型 int，boolean 等是值类型，其实是长久以来的一种误解，它混淆了实现和语义的区别。不要以为 Java 的官方文档那样写就是权威定论，就可以说“王垠不懂”：) 当你认为王垠不懂一个初级问题的时候，都需要三思，因为他可能是大智若愚…… 看了我下面的论述，也许你会发现自己应该怀疑的是，Java 的设计者到底有没有搞明白这个问题 :P

胡扯结束，现在来说正事。Java，Scheme 等语言的原始类型，比如 char，int，boolean，double 等，在“实现”上确实是通过值（而不是引用，或者叫指针）直接传递的，然而这完全是一种为了效率的优化（叫做 inlining）。这种优化对于程序员应该是不可见的。Java 继承了 Scheme/Lisp 的衣钵，它们在“语义”上其实是没有值类型的。

这不是天方夜谭，为了理解这一点，你可以做一个很有意思的思维实验。现在你把 Java 里面所有的原始类型都“想象”成引用类型，也就是说，所有的 int，boolean 等原始类型的变量都不包含实际的数据，而是引用（或者叫指针），指向堆上分配的数据。然后你会发现这样“改造后”的 Java，仍然符合现有 Java 代码里能看到的一切现象。也就是说，原始类型被作为值类型还是引用类型，对于程序员完全没有区别。

举个简单的例子，如果我们把 int 的实现变成完全的引用，然后来看这段代码：

```
int x = 1;      // x指向内存地址A, 内容是整数1
int y = x;      // y指向同样的内存地址A, 内容是整数1
x = 2;          // x指向另一个内存地址B, 内容是整数2。y仍然指向地址A, 内容是1。
```

由于我们改造后的 Java 里面 int 全部是引用，所以第一行定义的 x 并不包含一个整数，而是一个引用，它指向堆里分配的一块内存，这个空间的内容是整数 1。在第二行，我们定 int 变量 y，当然它也是一个引用，它的值跟 x 一样，所以 y 也指向同一个地址，里面的内容是同一个整数：1。在第三行，我们对 x 这个引用赋值。你会发现一个很有意思的现象，虽然 x 指向了 2，y 却仍然指向 1。对 x 赋值并没能改变 y 指向的内容，这种情况就跟 int 是值类型的时候一模一样！所以现在虽然 int 变量全部是引用，你却不能实现共享地址的引用能做的事情：对 x 进行某种操作，导致 y 指向的内容也发生改变。

出现这个现象的原因是，虽然现在 int 成了引用类型，你却并不能对它进行引用类型所特有（而值类型没有）的操作。这样的操作包括：

1. deref。就像 C 语言里的 * 操作符。
2. 成员赋值。就像对 C struct 成员的 x.foo = 2。

在 Java 里，你没法写像 C 语言的 *x = 2 这样的代码，因为 Java 没有提供 deref 操作符 *。你也没法通过 x.foo = 2 这样的语句改变 x 所指向的内存数据（内容是1）的一部分，因为 int 是一个原始类型。最后你发现，你只能写 x = 2，也就是改变 x 这个引用本身的指向。x = 2 执行之后，原来数字 1 在的内存空间并没有变成 2，只不过 x 指向了新的地址，那里装着数字 2 而已。指向 1 的其它引用变量比如 y，不会因为你进行了 x = 2 这个操作而看到 2，它们仍然看到原来那个 1……

在这种 int 是引用的 Java 里，你对 int 变量 x 能做的事情只有两种：

1. 读出它的值。
2. 对它进行赋值，使它指向另一个地方。

这两种事情，就跟你对值类型能做的两件事情没有区别。这就是为什么你没法通过对 x 的操作而改变 y 表示的值。所以不管 int 在实现上是传递值还是传递引用，它们在语义上都是等价的。也就是说，原始类型是值类型还是引用类型，对于程序员来说完全没有区别。你完全可以把 Java 所有的原始类型都当成引用类型，之后你能对它们做的事情，你的编程思路和方式，都不会因此有任何的改变。

从这个角度来看，Java 在语义上是没有值类型的。值类型和引用类型如果同时并存，程序员必须能够在语义上感觉到它们的不同，然而不管原始类型是值类型还是引用类型，作为程序员，你无法感觉到任何的不同。所以你完全可以认为 Java 只有引用类型，把原始类型全都当成引用类型来用，虽然它们确实是用值实现的。

一个在语义上有值类型的语言（比如 C#，Go 和 Swift）必须具有以下两种特性之一（或者两者都有），程序员才能感觉到值类型的存在：

1. deref 操作。这使得你可以用 *x = 2 这样的语句来改变引用指向的内容，导致共享地址的其它引用看到新的值。你没法通过 x = 2 让其他值变量得到新的值，所以你感觉到值类型的存在。
2. 像 struct 这样的“值组合类型”。你可以通过 x.foo = 2 这样的成员赋值改变引用数据（比如 class object）的一部分，使得共享地址的其它引用看到新的值。你没法通过成员赋值让另一个 struct 变量得到新的值，所以你感觉到值类型的存在。

实际上，所有的数据都是引用类型就是 Scheme 和 Java 最初的设计原理。原始类型用值来传递数据只是一种性能优化（叫做 inlining），它对于程序员应该是透明（看不见）的。那些在面试时喜欢问“Java 是否所有数据都是引用”，然后当你回答“是”的时候纠正你说“int，boolean 是值类型”的人，都是本本主义者。

思考题

有人指出，Java 的引用类型可以是 null，而原始类型不行，所以引用类型和值类型还是有区别的。但是其实这并不能否认本文指出的观点，你可以想想这是为什么吗？

Swift 语言的设计错误

在『[编程的智慧](#)』一文中，我分析和肯定了 Swift 语言的 optional type 设计，但这并不等于 Swift 语言的整体设计是完美没有问题的。其实 Swift 1.0 刚出来的时候，我就发现它的 array 可变性设计存在严重的错误。Swift 2.0 修正了这个问题，然而他们的修正方法却没有击中要害，所以导致了其它的问题。这个错误一直延续到今天。

Swift 1.0 试图利用 var 和 let 的区别来指定 array 成员的可变性，然而其实 var 和 let 只能指定 array reference 的可变性，而不能指定 array 成员的可变性。举个例子，Swift 1.0 试图实现这样的语义：

```
var shoppingList = ["Eggs", "Milk"]

// 可以对 array 成员赋值
shoppingList[0] = "Salad"

let shoppingList = ["Eggs", "Milk"]

// 不能对 array 成员赋值，报错
shoppingList[0] = "Salad"
```

这是错误的。在 Swift 1.0 里面，array 像其它的 object 一样，是一种“reference type”。为了理解这个问题，你应该清晰地区分 array reference 和 array 成员的区别。在这个例子里，shoppingList 是一个 array reference，而 shoppingList[0] 是访问一个 array 成员，这两者有着非常大的不同。

var 和 let 本来是用于指定 shoppingList 这个 reference 是否可变，也就是决定 shoppingList 是否可以指向另一个 array 对象。正确的用法应该是这样：

```
var shoppingList = ["Eggs", "Milk"]

// 可以对 array reference 赋值
shoppingList = ["Salad", "Noodles"]

// 可以对 array 成员赋值
shoppingList[0] = "Salad"

let shoppingList = ["Eggs", "Milk"]

// 不能对 array reference 赋值，报错
shoppingList = ["Salad", "Noodles"]

// let 不能限制对 array 成员赋值，不报错
shoppingList[0] = "Salad"
```

也就是说你可以用 var 和 let 来限制 shoppingList 这个 reference 的可变性，而不能用来限制 shoppingList[0] 这样的成员访问的可变性。

var 和 let 一旦被用于指定 array reference 的可变性，就不再能用于指定 array 成员的可变性。实际上 var 和 let 用于局部变量定义的时候，只能指定栈上数据的可变性。如果你理解 reference 是放在栈 (stack) 上的，而 Swift 1.0 的 array 是放在堆 (heap) 上的，就会明白 array 成员 (一种堆数据) 可变性，必须用另外的方式来指定，而不能用 var 和 let。

很多古老的语言都已经看清楚了这个问题，它们明确的用两种不同的方式来指定栈和堆数据的可变性。C++ 程序员都知道 int const * 和 int * const 的区别。Objective C 程序员都知道 NSArray 和 NSMutableArray 的区别。我不知道为什么 Swift 的设计者看不到这个问题，试图用同样的关键字 (var 和 let) 来指定栈和堆两种不同位置数据的可变性。实际上，不可变数组和可变数组，应该使用两种不同的类型来表示，就像 Objective C 的 NSArray 和 NSMutableArray 那样，而不应该使用 var 和 let 来区分。

Swift 2.0 修正了这个问题，然而可惜的是，它的修正方式是错误的。Swift 2.0 做出了一个离谱的改动，它把 array 从 reference type 变成了所谓 value type，也就是说把整个 array 放在栈上，而不是堆上。这貌似解决了以上的问题，由于 array 成了 value type，那么 shoppingList 就不是 reference，而代表整个 array 本身。所以在 array 是 value type 的情况下，你确实可以用 var 和 let 来决定它的成员是否可变。

```
let shoppingList = ["Eggs", "Milk"]

// 不能对 array 成员赋值，因为 shoppingList 是 value type
// 它表示整个 array 而不是一个指针
// 这个 array 的任何一部分都不可变
shoppingList[0] = "Salad"
```

这看似一个可行的解决方案，然而它却没有击中要害。这是一种削足适履的做法，它带来了另外的问题。把 array 作为 value type，使得每一次对 array 变量的赋值或者参数传递，都必须进行拷贝。你没法让两个变量指向同一个

array，也就是说 array 不再能被共享。比如：

```
var a = [1, 2, 3]  
  
// a 的内容被拷贝给 b  
// a 和 b 是两个不同的 array, 有相同的内容  
var b = a
```

这违反了程序员对于数组这种大型结构的心理模型，他们不再能清晰方便的对 array 进行思考。由于 array 会被不经意的自动拷贝，很容易犯错误。数组拷贝需要大量时间，就算接收者不修改它也必须拷贝，所以效率上有很大影响。不能共享同一个 array，在里面读写数据，是一个很大的功能缺失。由于这个原因，没有任何其它现代语言（Java，C#，……）把 array 作为 value type。

如果你看透了 value type 的实质，就会发现这整个概念的存在，在具有垃圾回收（GC）的现代语言里，几乎是没有意义的。有些新语言比如 Swift 和 Rust，试图利用 value type 来解决内存管理的效率问题，然而它带来的性能提升其实是微乎其微的，给程序员带来的麻烦和困扰却是有目共睹的。完全使用 reference type 的语言（比如 Java，Scheme，Python），程序员不需要思考 value type 和 reference type 的区别，大大简化和加速了编程的思维过程。Java 不但有非常高效的 GC，还可以利用 escape analysis 自动把某些堆数据放在栈上，程序员不需要思考就可以达到 value type 带来的那么一点点性能提升。相比之下，Swift，Rust 和 C# 的 value type 制造的更多是麻烦，而没有带来实在的性能优势。

Swift 1.0 犯下这种我一眼就看出来的低级错误，你也许从中发现了一个道理：编译器专家并不等于程序语言专家。很多经验老到的程序语言专家一看到 Swift 最初的 array 设计，就知道那是错的。只要团队里有一个语言专家指出了这个问题，就不需要这样反复的修改折腾。为什么 Swift 直到 1.0 发布都没有发现这个问题，到了 2.0 修正却仍然是错的？我猜这是因为 Apple 并没有聘请到合格的程序语言专家来进行 Swift 的设计，或者有合格的人，然而他们的建议却没有被领导采纳。Swift 的首席设计师是 Chris Lattner，也就是 LLVM 的设计者。他是不错的编译器专家，然而在程序语言设计方面，恐怕只能算业余水平。编译器和程序语言，真的是两个非常不同的领域。Apple 的领导们以为好的编译器作者就能设计出好的程序语言，以至于让 Chris Lattner 做了总设计师。

Swift 团队不像 Go 语言团队完全是一知半解的外行，他们在语言方面确实有一定的基础，所以 Swift 在大体上不会有特别严重的问题。然而可以看出来这些人功力还不够深厚，略带年轻人的自负，浮躁，盲目的创新和借鉴精神。有些设计并不是出自自己深入的见解，而只是“借鉴”其它语言的做法，所以可能犯下经验丰富的语言专家根本不会犯的错误。第一次就应该做对的事情，却需要经过多次返工。以至于每出一个新的版本，就出现一些“不兼容改动”，导致老版本语言写出来的代码不再能用。这个趋势在 Swift 3.0 还要继续。由于 Apple 的统治地位，这种情况对于 Swift 语言也许不是世界末日，然而它确实犯了语言设计的大忌。一个好的语言可以缺少一些特性，但它绝不应该加入错误的设计，导致日后出现不兼容的改变。我希望 Apple 能够早日招募到资深一些的语言设计专家，虚心采纳他们的建议。BTW，如果 Apple 支付足够的费用，我倒可以考虑兼职做他们的语言设计顾问 ;-)

Java 有 value type 吗？

有人看了以上的内容，问我：“你说 Java 只有 reference type，但是根据 Java 的[官方文档](#)，Java 也有 value type 和 reference type 的区别的。”由于这个问题相当的有趣，我另外写了一篇[文章](#)来回答这个问题。

正面思维的误区

有些人喜欢宣扬所谓“正面思维”（positive thinking），而不顾事实真相。每当你批评一些事情，他们就会拿出正面思维这个万能法宝来压制你，说：“你这人怎么这么 negative？要 positive，要看到事物好的方面才对！”

比如这次有人说：“你把之前每个东家都喷了一遍。这里面难道就没有你自己的问题吗？”我只能说，如果它们真的就是那么恶劣，那我有什么办法呢？由于没来得及选择，连续进入好几家问题公司，其实很正常。我不是一个完美的人，然而在公司的人际关系上，我可以说是仁至义尽了。我没架子，容易相处，这点很多同事都知道，甚至厨师和扫地大妈都知道。然而我绝对不是好欺负的。

像 Coverity, Sourcegraph 这类极品，欺压员工，无耻利用，行为极其恶劣，难道我还能说它们好话不成？我的心理不知道要扭曲到什么程度，才能发掘出他们好的地方来。这些公司的恶劣行径，严重损害了员工的身心健康，伤害了他们的事业发展，在某种程度上可以说是犯罪行为，没有把这些告上法庭就已经不错了。关于这些公司，有很多骇人听闻的细节我还没有说出来，我保留对这些进行进一步揭露的权利。

然而这不是今天的主题，我今天想谈的是所谓“正面思维”。很多人没有意识到，盲目的正面思维，其实是一个很严重的问题。正面并没有什么问题，快乐是好事，然而它们应该是结果，而不应该是目的。如果一个社会需要刻意去提倡“正面”和“快乐”，去宣扬它们，通过舆论压力或者暴力，迫使每个人都“正面思维”，那就有了严重问题了。文化大革命的时候，人们的思维可真是很正面啊，各种歌颂…… 你要是敢说任何不好听的话，立即被打成反革命右派。可是今天，我发现这种文革似的“正面思潮”，又有抬头之势。其实，它在美国已经泛滥成灾，以至于有人专门写了一本书来批判这种“正面思维”：



当你遇到困难的时候，美国人喜欢说：“别担心，一切都会好起来的……”，“要专注于事物好的方面……”，“只要你努力，就会有好结果……”，“困难是临时的，面商会有的，Go 语言会改进的……”，“危机会过去的，经济会持续增长的……”，“美国是世界上最伟大的国家，上帝保佑美利坚……”看看这本书，你就知道这些说法有多大的欺骗性。整个美国，其实都沉浸在人们不切实际的“正面幻想”之中。

“正面思维”跟美国的剥削制度和资本主义，是密不可分的。美国总是宣称自己是民主自由的国家。听到这个，比美国民主和自由很多的国家，都笑了。一个真正民主自由的国家，有什么必要反复的宣称自己是民主和自由的呢？事实上，美国是一个剥削和压迫非常严重的国家，美国人民并不幸福。实际上，正面思维就是剥削者想出来，用于安抚人民，让人安心做廉价劳动力的工具。一些所谓“成功人士”，总是鼓励大家要上进，要看到事物好的方面，说失业是一种福分，要安于现状，一步一步奋斗，往上爬！然后呢，自己却在背后玩弄权术，利用人们的正面不设防的心理，招摇撞骗，投机取巧，贬低人的价值，压低雇员工资，让别人加班加点，动作慢了随时开掉。自己却不劳而获，靠着一口官腔（所谓“领导才能”）飞黄腾达。

在美国，正面思维是一个产业。号称“快乐民族”的美国人，每年消耗掉世界上三分之二的抗抑郁症药物。美国出产层出不穷的正面思维和“成功学”书籍，DVD，以及其他产品：『[人性的弱点](#)』，『[心灵的鸡汤](#)』，『[谁动了我的奶酪](#)』，『[秘密](#)』…… 出产成千上万的所谓人生导师，职场教练，宗教领袖，知心大妈，心理医生，鸡汤和蛇油贩

子…… 他们的谋生方式，就是训练你如何正面思维，抑制负面情绪。这些人不能给你任何切实可行改善生活的办法，而只是告诉你，如何才能把生活的挫折，社会的不合理，不公平，都想成自己的思想有问题，或者自己不够努力，不够好。不论遇到什么样的不幸或者不公正待遇，你都不能抱怨抗议，反而还得“心存感激”，因为你活着就是上帝最好的恩赐。这也就是为什么美国有个节日叫“感恩节”，除了美国及其附庸加拿大，世界上没有其它国家庆祝感恩节。

美国的正面思维产业是如此的发达，甚至产生了一门学科，叫做“正面心理学”。哈佛大学还开设了红极一时的『[正面心理学](#)』课程（所谓“幸福课”）。我当年看了一阵子这课的[视频](#)，发现它真的很不寻常。课程进行到将近一半，教授仍然在做一般课程第一堂课的那种“动员工作”。没有传授任何切实可行的方法，只是反反复复地试图说服你，为什么你应该学正面心理学…… 老师啊，我坐在你课堂上半学期了，你还在告诉我为什么应该上你的课？！后来我发现，这个学科很像传销。它并不能让人快乐起来，然而它确实能教会你如何说服别人来上这门课，能把你训练成跟老师一样的“幸福课推销员”，然后你又可以去训练下一代的推销员…… 最后大家都成了推销员，然而推销员自己并不快乐，因为他们没有真正的产品和客户。

你知道为什么自从小布什做总统以来，美国的正面思维产业越来越红火了吗？因为小布什本来就是拉拉队长（cheerleader）出生，他以前的工作就是给大家加油鼓气的。小布什要求美国人民，一定要正面，一定要认为美国是世界上最伟大的国家，一定要认为美国人民是上帝的宠儿！;-)



在 Cornell 和 Google 的时候，我饱尝了盲目的正面思维所带来的危害。Cornell 这学校有个奇怪的现象，跟同学聊天时，如果你想打听某个教授的学术或者为人，得到的回应必然是：“他好牛！”“好厉害！”“非常聪明！”之类的话语。你听不到任何人说不好的方面，比如：“他讲课像是背书”，“他的研究没有实质意义”，“他的学生都很累”之类的负面信息。所以在 Cornell，你无法从同学那里得到任何信息，每个人都饱尝了与某些教授打交道的辛酸，可是每个人都把那些秘密藏在心底。他们对你说：“嗯，他很厉害，他的研究很伟大……”

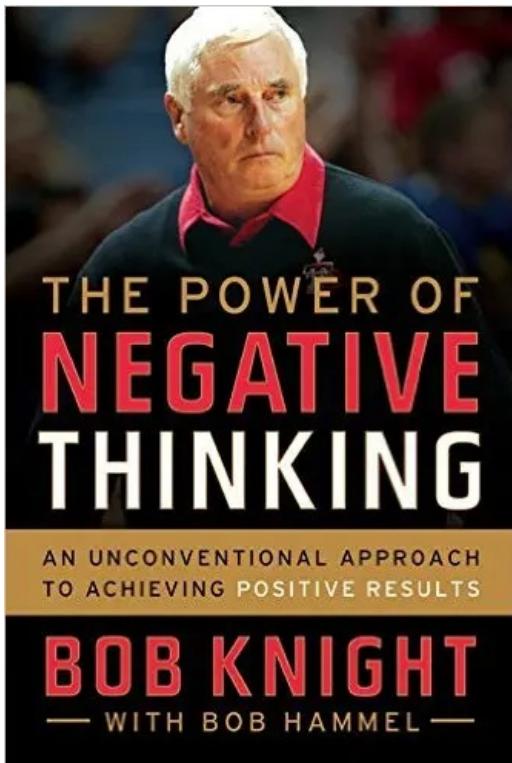
这种铺天盖地的正面信息，是无益甚至有害的。如果你只听到正面的声音，那你就无法做出正确的决定。这就像你在网上买东西，如果只看正面的评价，那你很可能买到有问题的商品。正确的作法，应该是正面负面的信息都看。特别是负面的信息，必须仔细看。它们可以告诉你，这个产品有哪些烦扰其他人的缺陷，会不会影响到你的使用。一般我在网上如果被一个产品吸引，我首先看的是一颗星的评价，因为给一颗星的人，一般是恨透了这个产品。当然里面有些无知或者不知好歹的人，你可以忽略，但是大部分人都会告诉你，他们不喜欢这个产品的具体原因。我很会分析这些评价，这就是为什么我家里的很多产品，都是非常好用的。

Cornell 这个学校，就是缺乏这种有益的负面评价。你总是听说每个教授都很牛，人都很好，…… 然而当你真正跟他们接触，就发现事实并非如此。你一次次的跳入火坑，然后才开始希望，要是开头的时候听到一些负面的信息，该多好。可是每个人表面上都是那么的 positive，每个人都认为 negative 是错误的心理，每个人都在强装笑容。这是一个多么可怕的地方！

Google 的气氛非常类似于 Cornell。Google 员工吃饭时，谈论每个项目或者团队，都带着玫瑰色的光环，仿佛 Google 做的一切都是美好的，先进的，有前途的。在每个星期的 TGIF (Tell Googlers It's Friday) 大会上，founder 们都在大讲台上宣布各种好消息，而对坏消息闭口不提或者一笔带过。下面的 Google 员工们群情激昂，对一些小不点的事情各种欢呼鼓掌尖叫，跟传销大会似的。事实上，Google 内部有许多穷途末路的项目。表面看上去很厉害的样子，等你进去才发现是死路一条，垂死挣扎。项目领导平时紧紧张张，生怕上面来人调查，把自己的项目杀掉。在公司内部搞各种政治，东拉西扯建立各种依赖关系，这样自己的项目才得以生存。

这种虚伪的正面氛围，存在于很多的美国公司，员工每个星期都被领导打各种鸡血针，保持激昂向上的状态。我曾经跟英国，法国，德国，意大利，瑞典，波兰等国家的同事聊天，他们都暗自嘲笑美国人，说过度正面，传销式的群情激昂，吃错药了一样，确实是美国文化的一大特色。欧洲人比较务实，不搞这套，好的就说好，坏的就批评或者嘲笑，直率坦荡。当然，我不能说所有美国公司都有这种问题，所以我仍然存在希望，找到稍微实在点的公司。

盲目的正面思维，忽略问题，并不能解决问题。你必须看到负面的事实，才有可能避免困难，得到好的结果。正面思维和浮夸的气氛，正在侵蚀 Google 和很多其它美国公司。为了看清楚正面思维的危害性，我推荐你看看这本书，名叫『负面思维的威力』：



博文的自愿付费方式

喜欢我的文章的人，可以自愿付费。你可以使用以下方式付款。为了防止有人滥用这些付款方式，我给它们设置了固定的金额。

1. 支付宝 (32 元) :



2. PayPal : 请点击[\[付款链接\]](#)

为什么自动车完全不可以犯错误

有人跟我讲，我对Google的自动车要求太苛刻了。人无完人，所以Google的产品也不需要是完美的，只要“够好用”就有市场。世界上有那么多糟糕的司机，酒后驾车的，开车时发短信的，打瞌睡的，判断失误的……导致了那么多的车祸，可比Google的自动车差多了。所以自动车不需要完美，只要99.9%的情况下可以正确工作，能大幅度减少车祸率，就是人类的福气了。

首先，现在的情况是，Google自动车现在只能在非常局限的情况下出来：白天，天气好，交通简单，而且就算是这样理想的条件下，一年之中仍然会发生270多起需要“[人工干预](#)”的事件，所以自动车的“驾驶技术”最后能不能超过最低级别的驾驶员，其实还很值得怀疑。其次，就算我们抛开这个问题不谈，假设自动车能够超过绝大部分人类驾驶员，能在99.9%的情况下判断正确，那么它也是不可行的。其实自动车必须能在100%的情况下做出正确的判断，不能犯任何错误，才有可能被人接受。这是为什么呢？

这其实是因为伦理和法律的原则。法律上的责任，并不是从宏观角度出发的。也就是说，法律不会因为自动车在99.9%的情况下判断正确，就免除那0.1%的情况下，Google对车祸的责任。法律的原则很简单，谁犯错误导致了车祸，谁就得负责，不管它是人还是机器都一样。是的，自动车也许不需要完美就可以用，但如果它犯错误引起了事故，责任就必须完全由Google，而不是车主来承担。因为如果车主是驾驶员，他开车引起车祸，那么车主就得负责。现在车主不是驾驶员，Google的软件才是驾驶员，所以如果自动车引起车祸，Google就得负完全的责任。

如果你还没有明白，我们来设想一个实例好了。假设Google自动车在99.9%的情况下，判断都是正确的，可就那么0.1%的情况下，它会判断失误而导致车祸。现在你就是这些不幸的人其中之一，你乘坐的Google自动车由于软件判断失误，导致车祸，让你双腿截肢，终生残疾。你把Google告上法庭。Google对法官讲，因为我们的自动车在99.9%的情况下都是可靠的，大幅度降低了社会的总体车祸率，对人类做出了巨大贡献。这个人很不幸，遇上了这0.1%判断失误的情况，所以Google对此不负责任。你觉得这可以接受吗？;)

0.1%的出错概率，落到一个人的头上，就等于100%的不幸。如果你本来是一个安全的驾驶员，那就更加不幸，因为如果是你自己开车，其实完全不会犯那样的错误。在这种情况下，就算自动车使得社会的总体车祸率急剧降低，对你来说其实毫无意义，因为残废的人是你。这就是为什么从伦理上讲，对机器和人，我们必须有两种不同的标准。自动车的判断力，并不是超越了大部分的驾驶员就可以的，它必须超过所有人！有些人开车时会犯的那些错误，自动车却完全不可以犯。因为坐了这辆犯错的自动车，导致身体残疾的人，他可以说：“如果是我自己开车，根本就不可能犯这样的错误。诚然，其它人在这种情况下可能会犯错，但我不会！所以Google的自动车对此负有严重的责任。”

明白了吗？只是能从宏观上减少车祸是不够的。自动车的驾驶技术，必须超越世界上最安全的驾驶员，它完全不可以犯错误。现在世界上虽然有许多的车祸，可是因为人是驾驶员，所以责任分摊在很多当事人的头上，谁犯错误谁负责。可是如果Google的自动车进入市场，代替了大部分的驾驶员，以后自动车引起的车祸的责任，全都会落到Google的头上。所以这样的生意，是非常困难而不切实际的。

AlphaGo与人工智能



在之前的一篇[文章](#)中我指出，自动驾驶所需要的“视觉识别能力”和“常识判断能力”，对于机器来说是非常困难的问题。至今没有任何机器可以在视觉方面达到人的水平，更不要说和人比。可是最近Google的[AlphaGo](#)战胜了围棋世界冠军，挺闹腾的，以至于对AI的误解又加深了。

本来玩个游戏而已，非要吹成是“历史性的人机大战”，说得好像是机器挑战了人类的智能，伤了人类的自尊似的。这整个项目打着一个相当高大上的招牌，叫做“[Deep Mind](#)”。当然，其中的技术也有一些吓人的名字，什么“神经网络”啊，“深度学习”啊……

听到这些，总有一知半解的人，根据科幻电影的情节开始展望，这样厉害的技术，应该可以用来做更加“智能”的事情，然后就开始对“人类的未来”作出一些猜想，比如自动驾驶就要实现，人们的工作很快都要被机器取代，甚至[Skynet](#)就要控制人类，云云。

我只想在这里给这些人提个醒：还是别做科幻梦了，回到现实吧。

棋类是相对容易的AI问题

一个常见的外行想法，是以为AlphaGo真的具有“人类智能”，所以Google利用同样的技术，应该可以实现自动驾驶。这些人不但大大的高估了所谓“AI”的能力，而且他们不明白，不同的“AI问题”的难度，其实有着天壤之别。

围棋是简单的，世界是复杂的。机器视觉和自动驾驶，难度比围棋要大许多倍，根本不在一个量级。要达到准确的视觉判断能力，机器必须拥有真正的认知能力和常识，这并不是AlphaGo所用的树搜索和神经网络，就可以解决的。由于需要以极高的速度处理“模拟信号”，这根本就不是人们常用的“数字计算机”可以解决的问题。也就是说，不是写代码就可以搞定的。

很早以前，人工智能专家们就发现一个很有趣的现象，是这样：

- 对于人来说很难，很烦的事情（复杂的计算，下棋，推理……），对于计算机来说，其实算是相对容易的事情。
- 对于人来说很容易的事情（认人，走路，开车，打球……），对于计算机来说，却非常困难。
- 计算机不能应付复杂的环境，只能在相对完美的环境下工作，需要精确的，离散的输入。
- 人对环境的适应能力很高，擅长于处理模糊的，连续的，不完美的数据。

从以上几点你可以看出，棋类活动正好符合了计算机的特点，因为它总是处于一种隔离的，完美的环境，具有离散的，精确的，有限的输入。棋盘上就那么几十，几百个点，不是随便放在哪里都可以的。一人走一步，轮流着走，不能乱来。整个棋盘的信息是完全可见的，没有隐藏和缺损的信息。棋局的“解空间”虽然很大，却非常规整，有规律可循。如果完全不靠经验和技巧的话，围棋的第一步可以有361种情况，第二步有360种情况，……

这对机器是非常有利的情况，因为计算机可以有计划有步骤，兢兢业业的把各种可能出现的情况算出来，一直到许多步以后，然后从中选择最有优势的走法。所以下棋归根结底，就是一个“树搜索”问题，只不过因为规模太大，需要加入一些优化。围棋的解空间虽然大，却是一个已知数，它最多有 250^{150} 种情况。AlphaGo使用所谓“神经网络”，就是为了在搜索的时候进行优化，尽早的排除不大可能取胜的情况，免得浪费计算的时间。

这种精确而死板的活动，就跟计算一个比较大的乘法算式（比如 2463757×65389 ）的性质类似，只不过规模大很多。显然，人做这类事情很繁，很累，容易出错，计算机对此却任劳任怨，因为它本来就是个机器。当年“深蓝”战胜国际象棋世界冠军的时候，我就已经预测到，计算机成为围棋世界冠军是迟早的事，所以没必要玩这些虐待自己脑子的游戏了。可惜的是，挺多人仍然把精通棋艺作为一种荣耀（因为“琴棋书画剑”嘛）。很多中国人认为，中国人下围棋总是输给韩国人，是一种耻辱。现在看来这是多么可笑的事情，这就像心算乘法不如韩国人快，就觉得是耻辱一样。）

认知是真正困难的AI问题

现在来对比一下人们生活中的琐事，就说倒水端茶吧。



让一个机器来给你倒水，有多难呢？意想不到的难！看看这个场景，如果你的电脑配备有摄像头，那么它怎么知道茶壶在哪里呢？要知道，茶壶的材料，颜色，形状，和角度，可以有几乎无穷多的变化。甚至有些茶壶跟哈哈镜一样，会把旁边的物体的形状都扭曲反射出来。桌上的物品附近都有各种反光和阴影，不同材料的反光特性还不一样，这些都会大幅度的影响机器对物品的识别。

为了识别物体，机器需要常识，它的头脑里必须有概念，必须知道什么样的东西才能叫做“茶壶”和“茶杯”。不要小看这一步的难度，这意味着机器必须理解基本的“拓扑结构”，什么叫做“连续的平面”，什么叫做“洞”，什么是“凹”和“凸”，什么是“里”和“外”……另外，这机器必须能够分辨物体和阴影。它必须知道水是什么，水有什么样的运动特性，什么叫做“流动”。它必须知道“水往低处流”，然后它又必须知道什么叫“低”和“高”……它必须知道茶杯为什么可以盛水，茶壶的嘴在哪里，把手在哪里，怎样才能拿起茶壶。如果一眼没有看见茶壶的把手，那它在哪里？茶壶的哪一面是“上面”，要怎样才可以把水从茶壶的嘴里倒出来，而不是从盖子上面泼出来？什么是裂掉的茶杯，它为什么会漏水，什么是缺口的茶杯，它为什么仍然可以盛水而不漏？干净的茶杯是什么样子的，什么是脏的茶杯，什么是茶垢，为什么茶垢不算是脏东西？如何控制水的流速和落点，什么叫做“水溅出来了”，要怎么倒水才不会溅出来？……

你也许没有想到，倒茶这么简单的事情，需要用到如此多的常识。所有这些变数加在一起，其实远远的大于围棋棋局的数量，人却可以不费力的完成。这能力，真是应该让人自己都吓一跳，然而人却对此不以为然，称之为“琐事”！因为其他人都可以做这样的事情，甚至猴子都可以，怎么能显得出我很了不起呢？人的自尊和虚荣，再一次的蒙蔽了他自己。他没有意识到，这其实是非常宝贵，让机器难以匹敌的能力。他说：“机器经过大量的学习，总有一天会做到的。看我们有神经网络呢，还有深度学习！”

机器学习是什么

有些人喜欢拿“机器学习”或者“深度学习”来吓唬人，以为出现了“学习”两个字，就可以化腐朽为神奇。而其实所谓机器学习，跟人类的学习，完全是两回事。机器的“学习能力”，并没有比石头高出很多，因为机器学习说白了，只不过是通过大量的数据，[统计拟合](#)出某些函数的参数。



比如，你采集到一些二维数据点。你猜测它们符合一个简单的函数 $y = ax^3 + bx^2 + cx + d$ ，但不知道 a, b, c 和 d 是多少。于是你就利用所谓“机器学习”（也就是数学统计），推断出参数 a, b, c 和 d 的值，使得采集到的数据尽可能的靠近这函数的曲线。可是这函数是怎么来的呢？终究还是人想出来的。机器无论如何也跳不出 $y = ax^3 + bx^2 + cx + d$ 这个框子。如果数据不符合这个范式，还是只有靠人，才能找到更加符合数据特性的函数。

所谓神经网络，其实也是一个函数，它在本质上跟 $y = ax^3 + bx^2 + cx + d$ 并没有不同，只不过输入的参数多一些，逻辑复杂一些。“神经网络”跟神经，其实完全没有关系，却偏喜欢说是受到了神经元的启发而来的。神经网络是一个非常聪明的广告词，它不知道迷惑了多少人。因为有“神经”两个字在里面，很多人以为它会让机器具有智能，而其实这些就是统计学家们斯通见惯的事情：拟合一个函数。你可以拟合出很好的函数，然而这跟智能没什么关系。

AlphaGo并不是人工智能历史性的突破

这次 AlphaGo 战胜了围棋冠军，跟之前 IBM 的“深蓝”电脑战胜国际象棋世界冠军，意义其实差不多。能够写出程序，在这些事情上打败世界冠军，的确是一个进步，它肯定会对某些特定的应用带来改善。然而，这并不说明 AI 取得了革命性的进步，更不能表明电脑具有了真正的、通用的智能。恰恰相反，电脑能够在棋类游戏中战胜人类，正好说明下棋这种活动，其实并不需要很多的智能。从事棋类活动的能力，并不足以衡量人的智力。

著名的认知科学家 Douglas Hofstadter (《GEB》的作者)，早就指出 AI 领域的那些热门话题，比如电脑下棋，跟真正意义上的人类智能，几乎完全不搭边。绝大部分人其实不明白思考和智能到底是什么。大部分所谓 AI 专家，对人脑的工作原理所知甚少，甚至完全不关心。

AlphaGo 所用的技术，也许能够用于其它同类的游戏，然而它并不能作为解决现实问题的通用方法。特别是，这种技术不可能对自动驾驶的发展带来突破。自动驾驶如果只比开车技术很差的人强一点，是不可接受的。它必须要近乎完美的工作，才有可能被人接受，然而这就要求它必须具有人类级别的视觉认知能力。比如，它必须能够察觉到前面车上绑了个家具，没绑稳，快要掉下来了，赶快换车道，超过它。可惜的是，自动驾驶的“眼睛”里看到的，只是一个个的立方块，它几乎完全不理解身边到底发生着什么，它只是在跟随和避让一些线条和方块…… 我们多希望马路都是游戏一样简单，清晰，完美，没有意外的，可惜它不是那样的。每一个细节都可能关系到人的生死，这就是现实世界。



为AlphaGo热血沸腾的人们，别再沉迷于自动驾驶和Skynet之类的幻想了。看清AI和“神经网络”的实质，用它们来做点有用的东西就可以，没必要对实现“人类智能”抱太大的希望。

我看自动驾驶技术

这段时间，Google的自动车，Tesla的autopilot，经常出现在新闻头条。人们热烈的讨论自动驾驶技术，对这“科幻般”的技术充满了憧憬，好奇，甚至恐惧。Google说：“自动车很安全。人类是糟糕的驾驶员。”很多人不假思索就接受了这种观点，以为自己不久以后就会被自动车所代替，所以我今天想谈谈对这些“自动车”的看法。

从我的另一篇文章，你应该已经看到，Tesla的autopilot其实根本不算“自动驾驶”，它完全不能和Google的自动车相比。Tesla把这种不成熟的软件推送到用户的车里，为的是跟Google抢风头，塑造自己的高大形象。看，我们先出了自动车！可是呢，Tesla那东西顶多算一个“adaptive cruise control”，离真正的自动驾驶还很遥远。可惜的是，Tesla为了自己的名声，拿用户的性命当儿戏，还有些人为它叫好。

然而就算是Google的自动车，离能够投入使用，其实还差得很远。我这里说的“很远”，不是像某些人预测的10年，20年，而是至少100年，1000年……甚至永远无法实现。这是为什么呢？Google不是声称，每天都要让它的自动车“学习”上百万mile的行驶记录吗？难道学习了如此的“大数据”，不能让这车子变得跟人一样聪明吗？

如果你这么想，那你可能根本不了解人工智能（AI）。需要“学习上百万mile”，并不能说明自动车很聪明。恰恰相反，这说明它们很笨。只需要问自己一个问题：一个人要学会开车，需要开多少里程？普通人从完全不会，到能安全上路，一般只需要12节课，每节课1小时。就算这一个小时你都在高速公路上开，也就80 mile的样子。12个小时就960 mile。也就是说，普通人只需要小于1000 mile的驾驶，就能成为比较可靠的司机。

对比一下Google的自动车，它们每天“分析”和“学习”一百万mile的“虚拟里程”，而且经常在外面采集数据，累计上百万的mile。然而这些自动车，仍然只能在白天，天气好的时候，在道路环境非常简单的Mountain View行驶。Mountain View就是一个小镇子，总共就没几条路，路上几乎没有行人。我从未在时速超过50mph的公路上，或者交通复杂的大城市，见到过Google的自动车。

另外据最近的报道，Google的自动车在过去一年时间里，发生了272起需要“人工干预”的错误情况。如果人不及时抢过控制权，不少情况会出现车祸。在此如此简单的条件下，还需要如此多的人工干预。如果环境稍微复杂一些，自动车恐怕就完全不知所措了。

这里还有一个“特殊关照”的问题，由于Google的自动车身上有着明显的标志，行人和其它驾驶员看到它，其实都有点提心吊胆的，不敢轻举妄动，怕它犯傻撞了自己，这也变相的降低了自动车的环境复杂度。一旦Google把车身上的标志去掉，大家看不出来谁是自动车，不对它们进行特殊的关照，我行我素，事故率恐怕就上去了。

所以Google的自动车，离能够投入真正的使用，差距还非常远。在这种情况下就妄言“自动车很安全”，“人类是糟糕的驾驶员”，……未免也太早了些吧？自动车跟人类差距到底有多远呢？天壤之别。普通人只需要开1000 mile就能学会开车，而这些自动车学习了几百万，几千万，几亿mile，仍然门都没有摸到。这说明自动车跟人类的运动神经，有着根本的区别。

人在运动的时候看见一个物体，他的头脑里会立即闪现与之相应的“概念”，然后很快浮现出这种东西的运动特点，以及相应的对策。相比之下，自动车看到物体，它并不能准确的判断它是什么东西：它是一个车，一个人，一棵树，一个施工路障，一个大坑，还是前面的车掉下来的床垫呢？所以自动车就像一个智障儿童，学了这么久连什么都不知道，却有人指望它们在十年之内能开车穿越美国。

对的，自动车配备了GPS，激光，雷达，……它的“感官”接收到很多的数据，有些是人类无法感觉到的。然而自动车的“头脑”（电脑），是没有认知能力的，所以就算收集到了大量的数据，它仍然不知道那东西是什么，它们之间是什么关系。电脑没有这些“常识”，所以它无法为人做出正确的判断。在危急的关头，它很可能会做出危及乘客安全的决定。“认知”是一个根本性的问题，AI领域至今没有解决它，甚至根本没有动手去研究它。

自动车使用的所谓“机器学习”的技术，跟人类的“学习”，完全是两回事。举个例子，一个小孩从来没见过猫，你只需要给她一只猫，告诉她这是“猫咪”。下一次，当她见到不管什么颜色的猫，不管它摆出什么姿势，都知道这是“猫咪”。现在的电脑，认知能力其实比小孩子，甚至其它动物都差很多。你先让电脑分析上百万张猫的照片，各种颜色，各种姿势，各种角度，拿一只猫摆在它的摄像头面前，让它看整整一年……最后它仍然不理解猫是什么，不能准确的判断一个东西是否是猫。如果说电脑有智商，那么它的级别就像一个蠕虫，甚至连蠕虫都不如。电脑没有认识和适应环境的能力，所以就算它再用功，“学习”再多的数据，都是白费劲。

很多人听说“人工智能”（AI），或者“机器学习”（machine learning），“深度学习”（deep learning）这类很酷的名词，就想起科幻小说里的智能机器人，就以为科幻就要成为现实。等你真的进入“机器学习”这领域，才发现一堆堆莫名其妙，稀里糊涂的做法，最后其实不怎么管用。这些大口号，包括所谓“深度学习”，其实跟人的思维方式，几乎完全不搭边。所谓“机器学习”，不过是一些普通的统计方法，拟合一些函数参数。吹得神乎其神，倒让统计专业的人士笑话。

人工智能在80年代出现过一次热潮。当时人们乐观的相信，电脑在不久就会拥有人类的智能。日本还号称要动员全国的力量，制造所谓“第五代计算机”，发展智能的编程语言（比如Prolog）。结果最后呢？人们意识到，超越人类（动物）的智能，比他们想象的困难太多太多。浮夸的许诺没能实现，AI领域进入了冬天。最近因为“大数据”，“自动车”和“Internet of Things”等热门话题的出现，“AI热”又死灰复燃。然而当今的AI，其实并没有比80年代的进步很多。人们对于自己的脑子以及感官的工作原理，仍然所知甚少，却盲目的认为那些从统计学偷来的概念，改名换姓

叫“机器学习”，就能造出跟自己的头脑媲美的机器。这些人其实大大的低估了自己身体的神奇程度。

视觉和认知能力，是动物（包括人类）特有的，卓越的能力。它们让动物能够准确的感知身边复杂的世界，对此作出适合自己生存的计划。一辆能够穿越整个国家的自动车，它必须适应各种复杂的环境：天气，路况，交通，意外情况…… 所以它需要动物的认知能力。我并不是说机器永远不可能具有这种能力，然而如果你根本不去欣赏，研究和理解这种能力，倒以为所谓“机器学习”就能办到这些事情，张口闭口拿“人类”说事，你又怎么可能用机器实现它呢？我的预测是，直到人类能够完全的理解动物的脑子和感官如何工作，才有可能制造出能够接近人类能力的自动车。

诚然，有少数人开车不小心，甚至酒后驾车，导致了很多的车祸。然而因此就声称“人类是糟糕的驾驶员”，那就是以偏概全了。大部分的人还是遵纪守法，注意安全的。很多人开车几十年，从没出过车祸。另外，我们必须把“态度”和“能力”区分开来看。酒后驾车的人，不是技术不够好，而是态度有问题。电脑当然没有态度问题，然而它的技术确实难以达到人的水平。就算那些酒后驾车的人，他们的能力其实也远远在电脑之上。我无法想象当今的电脑技术，要如何才能超越驾驶技术好的人，以及职业赛车手。

如果你还没明白，也许下面这个图片可以把你拉回到现实世界：



一个机器，如何能知道旁边的车上正在发生什么，即将可能发生什么样的危险情况呢？它如何知道，需要赶快避开这辆车呢？它不能。一个没有认知能力的机器，是难以应付复杂多变的现实世界的。

现在人们对于自动车技术的关注，热情，盲目乐观和浮夸，感觉跟文化大革命，“大跃进”年代的思维方式类似。只不过现在“毛泽东”换成了Google或者Tesla，“每亩产量十万”换成了“两年之内自动驾驶穿越美国”…… 我觉得与其瞎折腾自动驾驶技术，不如做点脚踏实地，在短期内能够见效，改善人们生活的东西。

给Java说句公道话

有些人问我，在现有的语言里面，有什么好的推荐？我说：“Java。”他们很惊讶：“什么？Java！”所以我现在来解释一下。

Java超越了所有咒骂它的“动态语言”

也许是因为年轻人的逆反心理，人们都不把自己的入门语言当回事。很早的时候，计算机系的学生用Scheme或者Pascal入门，现在大部分学校用Java。这也许就是为什么很多人恨Java，瞧不起用Java的人。提到Java，感觉就像是爷爷那辈人用的东西。大家都会用Java，怎么能显得我优秀出众呢？于是他们说：“Java老气，庞大，复杂，臃肿。我更愿意探索新的语言……”

某些Python程序员，在论坛里跟初学者讲解Python有什么好，其中一个原因竟然是：“因为Python不是Java！”他们喜欢这样宣传：“看Python多简单清晰啊，都不需要写类型……”对于Java的无缘无故的恨，盲目的否认，导致了他们看不到它很重要的优点，以至于迷失自己的方向。虽然气势上占上风，然而其实Python作为一个编程语言，是完全无法和Java抗衡的。

在性能上，Python比Java慢几十倍。由于缺乏静态类型等重要设施，Python代码有bug不容易发现，发现了也不容易debug，所以Python无法用于构造大规模的，复杂的系统。你也许发现某些startup公司的主要代码是Python写的，然而这些公司的软件，质量其实相当的低。在成熟的公司里，Python最多只用来写工具性质的东西，或者小型的，不会影响系统可靠性的脚本。

静态类型的缺乏，也导致了Python不可能有很好的IDE支持，你不能完全可靠地“跳转到定义”，不可能完全可靠地重构(refactor) Python代码。PyCharm对于早期的Python编程环境，是一个很大的改进，然而理论决定了，它不可能完全可靠地进行“变量换名”等基本的重构操作。就算是比PyCharm强大很多的PySonar，对此也无能为力。由于Python的设计过度的“动态”，没有类型标记，使得完全准确的定义查找，成为了不可判定(undecidable)的问题。

在设计上，Python，Ruby比起Java，其实复杂很多。缺少了很多重要的特性，有毛病的“强大特性”倒是多了一堆。由于盲目的推崇所谓“正宗的面向对象”方式，所谓“[late binding](#)”，这些语言里面有太多可以“重载”语义的地方，不管什么都可以被重定义，这导致代码具有很大的不确定性和复杂性，很多bug就是被隐藏在这些被重载的语言结构里面了。因此，Python和Ruby代码很容易被滥用，不容易理解，容易写得很乱，容易出问题。

很多JavaScript程序员也盲目地鄙视Java，而其实JavaScript比Python和Ruby还要差。不但具有它们的几乎所有缺点，而且缺乏一些必要的设施。JavaScript的各种“WEB框架”，层出不穷，似乎一直在推陈出新，而其实呢，全都是在黑暗里瞎蒙乱撞。JavaScript的社区以幼稚和愚昧著称。你经常发现一些非常基本的常识，被JavaScript“专家”们当成了不起的发现似的，在大会上宣讲。我看不出来JavaScript社区开那些会议，到底有什么意义，仿佛只是为了拉关系找工作。

Python凑合可以用在不重要的地方，Ruby是垃圾，JavaScript是垃圾中的垃圾。原因很简单，因为Ruby和JavaScript的设计者，其实都是一知半解的民科。然而世界就是这么奇怪，一个彻底的垃圾语言，仍然可以宣称是“程序员最好的朋友”，从而得到某些人的爱戴……

Java的“继承人”没能超越它

最近一段时间，很多人热衷于Scala，Clojure，Go等新兴的语言，他们以为这些是比Java更现代，更先进的语言，以为它们最终会取代Java。然而这些狂热分子们逐渐发现，Scala，Clojure和Go其实并没有解决它们声称能解决的问题，反而带来了它们自己的毛病，而这些毛病很多是Java没有的。然后他们才意识到，Java离寿终正寝的时候，还远得很……

Go语言

关于Go，我已经评论过很多了，有兴趣的人可以看[这里](#)。总之，Go是民科加自大狂的产物，奇葩得不得了。这里我就不多说它了，只谈谈Scala和Clojure。

Scala

我认识一些人，开头很推崇Scala，仿佛什么救星似的。我建议他们别去折腾了，老老实实用Java。没听我的，结果到后来，成天都在骂Scala的各种毛病。但是没办法啊，项目上了贼船，不得不继续用下去。我不喜欢进行人身攻击，然而我发现一个语言的好坏，往往取决于它的设计者的背景，觉悟，人品和动机。很多时候我看人的直觉是异常的准，以至于依据对语言设计者的第一印象，我就能预测到这个语言将来会怎么发展。在这里，我想谈一下对Scala和Clojure的设计者的看法。

Scala的设计者Martin Odersky，在PL领域有所建树，发表了不少学术论文（包括著名的《[The Call-by-Need Lambda Calculus](#)》），而且还是大名鼎鼎的[Niklaus Wirth](#)的门徒，我因此以为他还比较靠谱。可是开始接触Scala没多久，我就很惊讶的发现，有些非常基本的东西，Scala都设计错了。这就是为什么我几度试图采用Scala，最后都不了了之。因为我一边看，一边发现让人跌眼镜的设计失误，而这些问题都是Java没有的。这样几次之后，我

就对Odersky失去了信心，对Scala失去了兴趣。

回头看看Odersky那些论文的本质，我发现虽然理论性貌似很强，其实很多是在故弄玄虚（包括那所谓的“call-by-need lambda calculus”）。他虽然对某些特定的问题有一定深度，然而知识面其实不是很广，眼光比较片面。对于语言的整体设计，把握不够好。感觉他是把各种语言里的特性，强行拼凑在一起，并没有考虑过它们是否能够“和谐”的共存，也很少考虑“可用性”。

由于Odersky是大学教授，名声在外，很多人想找他拿个PhD，所以东拉西扯，喜欢往Scala里面加入一些不明不白，有潜在问题的“特性”，其目的就是发paper，混毕业。这导致Scala不加选择的加入过多的特性，过度繁复。加入的特性很多后来被证明没有多大用处，反而带来了问题。学生把代码实现加入到Scala的编译器，毕业就走人不管了，所以Scala编译器里，就留下一堆堆的历史遗留垃圾和bug。这也许不是Odersky一个人的错，然而至少说明他把关不严，或者品位确实有问题。

最有名的采用Scala的公司，无非是Twitter。其实像Twitter那样的系统，用Java照样写得出来。Twitter后来怎么样了呢？CEO都跑了:P 新CEO上台就裁员300多人，包括工程师在内。我估计Twitter裁员的一个原因是，有太多的Scala程序员，扯着各种高大上不实用的口号，比如“函数式编程”，进行过度工程，浪费公司的资源。花着公司的钱，开着各种会议，组织各种meetup和hackathon，提高自己在open source领域的威望，其实没有为公司创造很多价值.....

Clojure

再来说一下Clojure。当Clojure最初“横空面世”的时候，有些人热血沸腾地向我推荐。于是我看了一下它的设计者Rich Hickey做的宣传讲座视频。当时我就对他一知半解拍胸脯的本事，印象非常的深刻。Rich Hickey真的是半路出家，连个CS学位都没有。可他那种气势，仿佛其他的语言设计者什么都不懂，只有他看到了真理似的。不过也只有这样的人，才能创造出“宗教”吧？

满口热门的名词，什么lazy啊，pure啊，STM啊，号称能解决“大规模并发”的问题，..... 这就很容易让人上钩。其实他这些词儿，都是从别的语言道听途说来，却又没能深刻理解其精髓。有些“函数式语言”的特性，本来就是有问题的，却为了主义正确，为了显得高大上，抄过来。所以最后你发现这语言是挂着羊头卖狗肉，狗皮膏药一样说得头头是道，用起来怎么就那么蹩脚。

Clojure的社区，一直忙着从Scheme和Racket的项目里抄袭思想，却又想标榜是自己的发明。比如Typed Clojure，就是原封不动抄袭Typed Racket。有些一模一样的基本概念，在Scheme里面都几十年了，恁是要改个不一样的名字，免得你们发现那是Scheme先有的。甚至有人把SICP，The Little Schemer等名著里的代码，全都用Clojure改写一遍，结果完全失去了原作的简单和清晰。最后你发现，Clojure里面好的地方，全都是Scheme已经有的，Clojure里面新的特性，几乎全都有问题。我参加过一些Clojure的meetup，可是后来发现，里面竟是各种喊着大口号的小白，各种趾高气昂的民科，愚昧之至。

如果现在要做一个系统，真的宁可用Java，也不要浪费时间去折腾什么Scala或者Clojure。错误的人设计了错误的语言，拿出来浪费大家的时间。

Java没有特别讨厌的地方

我至今不明白，很多人对Java的仇恨和鄙视，从何而来。它也许缺少一些方便的特性，然而长久以来用Java进行教学，用Java工作，用Java开发PySonar，RubySonar，Yin语言，..... 我发现Java其实并不像很多人传说的那么可恶。我发现自己想要的95%以上的功能，在Java里面都能找到比较直接的用法。剩下的5%，用稍微笨一点的办法，一样可以解决问题。

盲目推崇Scala和Clojure的人们，很多最后都发现，这些语言里面的“新特性”，几乎都有毛病，里面最重要最有用的特性，其实早就已经在Java里了。有些人跟我说：“你看，Java做不了这件事情！”后来经我分析，发现他们在潜意识里早已死板的认定，非得用某种最新最酷的语言特性，才能达到目的。Java没有这些特性，他们就以为非得用另外的语言。其实，如果你换一个角度来看问题，不要钻牛角尖，专注于解决问题，而不是去追求最新最酷的“写法”，你就能用Java解决它，而且解决得干净利落。

很多人说Java复杂臃肿，其实是因为早期的[Design Patterns](#)，试图提出千篇一律的模板，给程序带来了不必要的复杂性。然而Java语言本身跟Design Patterns并不是等价的。Java的设计者，跟Design Pattern的设计者，完全是不同的人。你完全可以使用Java写出非常简单的代码，而不使用Design Patterns。

Java只是一个语言。语言只提供给你基本的机制，至于代码写的复杂还是简单，取决于人。把对一些滥用Design Patterns的Java程序员的恨，转移到Java语言本身，从而完全抛弃它的一切，是不明智的。

结论

我平时用着Java偷着乐，本来懒得评论其它语言的。可是实在不忍心看着有些人被Scala和Clojure忽悠，所以在里说几句。如果没有超级高的性能和资源需求（可能要用C这样的低级语言），目前我建议就老老实实用Java吧。虽然不如一些新的语言炫酷，然而实际的系统，还真没有什么是Java写不出来的。少数地方可能需要绕过一些限制，或者放宽一些要求，然而这样的情况不是很多。

编程使用什么工具是重要的，然而工具终究不如自己的技术重要。很多人花了太多时间，折腾各种新的语言，希望它们会奇迹一般的改善代码质量，结果最后什么都没做出来。选择语言最重要的条件，应该是“够好用”就可以，因为项目的成功最终是靠人，而不是靠语言。既然Java没有特别大的问题，不会让你没法做好项目，为什么要去试一些不靠谱的新语言呢？

Tesla Autopilot

以下内容是《[Tesla Model S的设计失误](#)》一文中新加入的小节。由于写作时间相距太远，而且由于它的时效性，现在也把它单独提出来，独立成文。

两个月前，Tesla 通过“软件更新”，使 Model S 具有了初级的“自动驾驶”（autopilot）功能。这个功能可以让 Model S 自动地，沿着有“清晰边界线”的车道行驶，根据前后车辆的速度相应的加速和减速。

这貌似一个很新很酷的功能，乍一看跟 Google 的自动车有的一拼（其实差得天远）。然而在推出后不久，YouTube 上出现了一些视频（[视频1](#)，[视频2](#)，[视频3](#)，[视频4](#)，[视频5](#)）。它们显示，autopilot 在某些情况下有可能进行错误的判断和操作，有些险些造成严重的迎面车祸。



特别是[视频1](#)显示，在路面线条清晰，天气很好的路上，autopilot 忽然向左，试图转向反方向的车道，差点导致严重的对撞车祸。仔细观察 autopilot 转向之前的情况，是由于路面上有阳光投下来的树影。Autopilot 误以为那是一个障碍物，所以试图把车转上反方向的车道！

从这个简单的视频我们可以看出：

1. Autopilot 没有对图像进行基本的“阴影消除”，它不能区分阴影和障碍物。阳光强烈，阴影明显的时候，autopilot 可能把阴影当成障碍物。阴影消除在计算机视觉已经研究挺多了，这说明Tesla有可能没有进行基础的计算机视觉研究。缺乏分辨阴影和障碍物的能力，这样的自动驾驶系统是完全不可接受的。
2. 道路中间有明显的，表示“禁止超车”的双黄线，对面有来车。Autopilot 为了避开“障碍”，冒着对撞的危险，左转跨越双黄线。这表示 autopilot 连基本的交通规则，紧急情况下的正确操作方式都搞不清楚。或者也许这软件里面连双黄线都没有识别，甚至连这个概念都没有。

对于一个有经验的驾驶员来说，如果发现前方有障碍物，正确的作法不应该是猛烈地转弯避开，而应该是紧急刹车。从视频上我们看出，车子没有刹车减速（保持在 37~38），而是猛烈地左转。而且是等树影到了面前，才忽然进行操作，没有计算提前量。这说明设计 autopilot 的人，连基本的开车常识都不明白。

让我感到悲哀的是，这些视频的很多评论，大部分都在谩骂车主是傻逼：“这是车主自己的责任！”，“Autopilot 只能在高速公路上使用”，“只能在车道上有明确的边界线的时候使用！”，“不能在有很多弯道的地方”，“只能在能够看见前方300米道路的地方使用”，“谁叫你不看说明书的！”…… Elon Musk 也在一次[采访](#)中明确的告诉记者：“如果用户因为使用 autopilot 而导致了车祸，是用户自己的责任！”他反复地声明：“autopilot 还处于 beta 版本……”意思是，你们小心着用！

我对这些说法持不同的观点。首先，Tesla 根本就不应该把一个处于“beta 状态”的功能，自动推送到所有 Model S 的系统里面。实际上，像 autopilot 这种功能，关系到人的生命安全，根本就不应该有“beta版本”或者“测试版本”之说。Tesla 把这样不成熟的系统，强制推送给用户，然后又说如果出了事故，用户负所有责任，这是一种推卸责任的做法。要知道，没有任何人愿意拿自己的生命给 Tesla 做“beta 测试”。

另外，就算是用户没有仔细阅读 autopilot 的使用说明，在“不该”用它的地方（比如路面线条不清晰的地方）使用了 autopilot，如果出了车祸，Tesla 也应该负完全的责任。理由如下：

1. 作为用户，他们没有义务阅读并且深刻的理解 autopilot 的局限性。在软件行业，存在一种习惯性的“责备用户”的不良风气。如果软件的设计有问题，用户没记住它的毛病，没能有效地绕过，那么如果出了问题，一般被

认为是用户的错。Tesla 想把软件行业的这种不正之风，引入到人命关天的汽车行业，那显然是行不通的。

2. Tesla 的 autopilot 实现方式幼稚，局限性实在太多。天气不好的时候不行，路面上的边界线不清晰也不行，光线不好或者有阴影不行，路上有施工的路桩不行，高速出口不行，…… 实际上，在如此苛刻的限定条件下，任何一个汽车厂商都可以做出 Tesla 那种 autopilot。

我自己的便宜 Honda 车，就有偏离车道时发出警告的功能（Lane Drift Warning，LDW）。装个摄像头，来点最简单的图像处理就搞定。在Indiana大学的时候，我们有一门本科级别的课程，就是写代码控制一辆高尔夫球车（也是电动车呢），沿着路面上的线条自动行驶。这根本没什么难度，因为它能正确行驶的条件，实在是太苛刻了。

其它汽车厂商很清楚这种功能的局限性，所以他们没有大肆吹嘘这种“线检测”的技术，或者把它做成 autopilot。他们只是把它作为辅助的，提示性的功能。这些汽车厂商理解，作为一个用户，他们不可能，也不应该记住 autopilot 能正确工作的种种前提条件。

3. 用户没有足够的能力来“判断”autopilot 正常工作的条件是否满足。比如，路上的线还在，但是被磨损了，颜色很浅，那么 autopilot 到底能不能用呢？谁也不知道。把判断这些条件是否满足的任务推给用户，就像是在要求用户帮 Tesla 的工程师 debug 代码。这显然是不可行的。如果 autopilot 能够在检测到道路条件不满足的情况下，自动警告用户，并且退出自动驾驶模式，那还稍微合理一些。
4. 用户也许没有足够的时间来响应条件的改变。Autopilot 自动驾驶的时候，车子有可能最初行驶在较好的条件下（天气好，路面线条清晰），然而随着高速行驶，路面条件有可能急速的变化。有可能上一秒还好好的，下一秒路面线条就不再清晰（[视频5](#) 貌似这种情况）。路面条件的变化突如其来，驾驶员没有料到。等他们反应过来，想关闭 autopilot 的时候，车祸已经发生了。这种情况如果上诉到法庭，稍微明理一点的法官，都应该判 Tesla 败诉。
5. Autopilot 显摆出的“高科技”形象，容易使人产生盲目的信任，以至于疏忽而出现车祸。既然叫做“autopilot”，这意味着它能够不需要人干预，自动驾驶一段时间。既然用户觉得它能自动驾驶，那么他们完全有理由在到达高速路口之前（比如 GPS 显示还有一个小时才到出口），做一些自己的事情：比如看看手机啊，看看书啊，甚至刷刷牙…… 不然，谁让你叫它是“autopilot”的呢？我坐飞机时，就见过飞行员打开 autopilot，上厕所去了。如果启用了 autopilot 还得一秒钟不停地集中注意力，那恐怕比自己开车还累。自己开车只需要看路，现在有了 autopilot，不但要看路，还要盯着方向盘，防止 autopilot 犯傻出错……
6. Tesla 把“beta 版”的 autopilot 推送给所有的 Model S，是对社会安全不负责任的做法。你要明白 Murphy's Law：如果一个东西可能出问题，那么就一定会有人让它出问题。Autopilot 的功能不成熟，限制条件很多，不容易被正确使用，这不但对 Model S 的车主自己，而且对其他人也是一种威胁。汽车不是玩具，随便做个新功能，beta 版，让人来试用，是会玩出人命的。我觉得 Tesla 的 autopilot，跟无照驾驶的人一样，应该被法律禁止。由于 autopilot 的复杂性和潜在的危险性，使用 autopilot 的用户，应该经过 DMV 考核，在驾照上注明“能正确使用 Tesla autopilot”，才准上路。
7. 关系到人的生命安全的“免责声明”和“用户协议”，在法律上是无效的。在美国，到处都存在“免责声明”之说。比如你去参加学校组织的春游活动，都要叫你签一个“waiver”，说如果出了安全事故或者意外，你不能把学校告上法庭。这种免责声明，一般在法律上都是无效的。如果由于学校的过错而致使你的身体受了损伤，就算你签了这种 waiver，照样可以把学校告上法庭。我估计 Tesla 的 autopilot 在启动时，也有这样的免责声明，说如果使用 autopilot 而出现车祸，Tesla 不负责任。由于 autopilot 直接操控了你的车子，如果真的出了车祸，这跟其它的 waiver 一样，都是无效的。你照样可以上法庭告他们。

由于意识到这个问题，知道出了问题自己是逃不掉责任的，Tesla 最近又通过强制的软件更新，对 autopilot 的功能进行了一些限制，说是为了防止用户“滥用” autopilot 做一些“疯狂”的事情。Tesla 很疯狂，反倒指责用户“滥用”和“疯狂”。这让人很愤慨。

对 autopilot 进行限制的同时，Tesla 又推出了 beta 版的“[自动趴车](#)”和“[召唤](#)”（summon）功能。这些功能貌似很酷，然而它们也附带了许多的限制条件。你只能在某些地方，满足某种特定条件，才能用这些功能。如果你违反这些条件，出了事故，Tesla 声称不负责。

这些能够让车子自己移动的功能，跟 autopilot 一样，同样会给社会带来安全隐患。比如，有人在不该使用自动趴车和 summon 功能的地方用了它，就可能会导致车祸。这不是用户的问题，而是 Tesla 根本不应该发布这些不成熟的技术来哗众取巧。

Tesla Model X的车门设计问题

Tesla即将推出的SUV (Model X) , 不但继承了以上提到的Model S的各种问题 (触摸屏, 门把, ...) , 而且还制造了新的问题。Model X具有一个别出心裁的车门设计, 这车子看起来像一只展翅的鸟:



这样开的车门貌似更省空间, 方便在狭窄的地方开门, 而且看起来更酷, 有点像McLaren或者Lamborghini。然而这样的设计, 在我看来有以下几个问题:

安全性问题

由于后门完全由电机控制, 在车子失去电力的时候要打开后门, 过程复杂得离谱。首先在失去电力的时候, 无论如何是不可能在车外把门打开的。这意味着, 如果出车祸着火了, 消防队员可能没法很快帮你打开车门。

如果你运气好, 没有受伤, 头脑还清醒, 气囊和安全带也没挡住你, 那么你必须完成三件复杂的操作, 才能逃离Model X :

1. 揭开车门上扬声器的盖子
2. 扳动一个隐蔽在那里的机关
3. 然后自己把车门举起来

我比较无语..... 如何揭开扬声器的盖子? 你需要随时在车里准备好榔头和改锥吗? 螺丝藏在哪里的? 要知道, 用户可不是Tesla雇佣的机械师。这对于车祸逃生是非常不利的设计。如果撞坏了电源, 后面的人恐怕无法在合理时间之内逃离。

OPENING THE FALCON WING DOORS WITHOUT POWER

Without 12 volt power, the falcon wing doors can only be opened from the inside of the vehicle. Remove the speaker grill from the door and pull the mechanical release cable down and towards the front seat, as shown below. After the latch has released, manually lift up the doors.



雨雪天气的麻烦



- 车门升起时，车的后部上方有很大的空挡。下雨或者下雪的时候，雨雪会乘着车门张开的时候，大量飘进车里。
- 当车门升起来的时候，夹在门缝里的灰尘和渣滓会掉进车里。车顶上如果有积雪，也会掉进去。设想一下，下雪天开了一会车，打开后门，结果车顶上的雪都掉在后面的人头上了……
- 下大雪的时候，后面的车门可能被大雪压住打不开，或者导致电动机超负荷损坏。
- 由于门上的缝隙太长，这种设计更加容易出现由于密封圈老化而漏水的问题。

趴车的麻烦

在顶棚很低的车库里可能会碰到天花板。这是一个很现实的问题，因为很多车库旁边的空间很多，顶棚却很低。比如，这次我到洛杉矶和拉斯维加斯旅游，经常遇到这样的车库：



总的说来，在非常狭窄的地方开门，其实并不是什么很需要解决的事情。有钱买Model X的人，难道会经常把车停在狭窄的夹缝里吗？为了这种不常见的应用，用得着花这么大功夫设计个车门吗？就算你能开门，人出去之后挤不挤得

出去，是另外一回事。如果地方实在太窄，你完全可以让后面的人先下车，然后再进车位。

另外，滑动式的车门同样可以解决这个问题，根本用不着花大成本来实现升起来的车门。



如果你知道，Model X的车身宽度为81.6英寸，比Hummer H2, Cadillac Escalade和Ford F-150这样的庞然大物还要宽，你就会发现真正的问题不在于空间不够，而是在于这车实在太宽了。有多少人愿意开这么宽的车，是一个问题。

实用性问题

- 顶棚不再能安装货架。不知道滑雪板和kayak之类该绑在哪里。这降低了Model X作为一个SUV (Sport Utility Vehicle) 的使用价值。



- 后门上不再能放随身物品。这样后面的乘客会不是很方便。



制造和维修的问题

- 这种车门机械非常复杂，容易出问题，维修起来很麻烦。制造起来也很麻烦，以至于一家很有经验的，为奔驰和通用提供配件的德国设备厂商，都没法满足Tesla的要求。请参考这篇[新闻](#)。



编程的智慧

编程是一种创造性的工作，是一门艺术。精通任何一门艺术，都需要很多的练习和领悟，所以这里提出的“智慧”，并不是号称一天瘦十斤的减肥药，它并不能代替你自己的勤奋。然而由于软件行业喜欢标新立异，喜欢把简单的事情搞复杂，我希望这些文字能给迷惑中的人们指出一些正确的方向，让他们少走一些弯路，基本做到一分耕耘一分收获。

反复推敲代码

有些人喜欢炫耀自己写了多少多少万行的代码，仿佛代码的数量是衡量编程水平的标准。然而，如果你总是匆匆写出代码，却从来不回去推敲，修改和提炼，其实是不可能提高编程水平的。你会制造出越来越多平庸甚至糟糕的代码。在这种意义上，很多人所谓的“工作经验”，跟他代码的质量其实不一定成正比。如果有几十年的工作经验，却从来不回去推敲和反思自己的代码，那么他也许还不如一个只有一两年经验，却喜欢反复推敲，仔细领悟的人。

有位文豪说得好：“看一个作家的水平，不是看他发表了多少文字，而要看他的废纸篓里扔掉了多少。”我觉得同样的理论适用于编程。好的程序员，他们删掉的代码，比留下来的还要多很多。如果你看见一个人写了很多代码，却没有删掉多少，那他的代码一定有很多垃圾。

就像文学作品一样，代码是不可能一蹴而就的。灵感似乎总是零零星星，陆陆续续到来的。任何人都不可能一笔呵成，就算再厉害的程序员，也需要经过一段时间，才能发现最简单优雅的写法。有时候你反复提炼一段代码，觉得到了顶峰，没法再改进了，可是过了几个月再回头看，又发现好多可以改进和简化的地方。这跟写文章一模一样，回头看几个月或者几年前写的东西，你总能发现一些改进。

所以如果反复提炼代码已经不再有进展，那么你可以暂时把它放下。过几个星期或者几个月再回头来看，也许就有焕然一新的灵感。这样反反复复很多次之后，你就积累了灵感和智慧，从而能够在遇到新问题的时候直接朝正确，或者接近正确的方向前进。

写优雅的代码

人们都讨厌“面条代码”（spaghetti code），因为它就像面条一样绕来绕去，没法理清头绪。那么优雅的代码一般是什么形状的呢？经过多年的观察，我发现优雅的代码，在形状上有一些明显的特征。

如果我们忽略具体的内容，从大体结构上来看，优雅的代码看起来就像是一些整整齐齐，套在一起的盒子。如果跟整理房间做一个类比，就很容易理解。如果你把所有物品都丢在一个很大的抽屉里，那么它们就会全都混在一起。你就很难整理，很难迅速的找到需要的东西。但是如果你在抽屉里再放几个小盒子，把物品分门别类放进去，那么它们就不会到处乱跑，你就可以比较容易的找到和管理它们。

优雅的代码的另一个特征是，它的逻辑大体上看起来，是枝丫分明的树状结构（tree）。这是因为程序所做的几乎一切事情，都是信息的传递和分支。你可以把代码看成是一个电路，电流经过导线，分流或者汇合。如果你是这样思考的，你的代码里就会比较少出现只有一个分支的if语句，它看起来就会像这个样子：

```
if (...) {  
    if (...) {  
        ...  
    } else {  
        ...  
    }  
} else if (...) {  
    ...  
} else {  
    ...  
}
```

注意到了吗？在我的代码里面，if语句几乎总是有两个分支。它们有可能嵌套，有多层的缩进，而且else分支里面有可能出现少量重复的代码。然而这样的结构，逻辑却非常严密和清晰。在后面我会告诉你为什么if语句最好有两个分支。

写模块化的代码

有些人吵着闹着要让程序“模块化”，结果他们的做法是把代码分部到多个文件和目录里面，然后把这些目录或者文件叫做“module”。他们甚至把这些目录分放在不同的VCS repo里面。结果这样的作法并没有带来合作的流畅，而是带来了许多的麻烦。这是因为他们其实并不理解什么叫做“模块”，肤浅的把代码切割开来，分放在不同的位置，其实非但不能达到模块化的目的，而且制造了不必要的麻烦。

真正的模块化，并不是文本意义上的，而是逻辑意义上的。一个模块应该像一个电路芯片，它有定义良好的输入和输出。实际上一种很好的模块化方法早已经存在，它的名字叫做“函数”。每一个函数都有明确的输入（参数）和输出（返回值），同一个文件里可以包含多个函数，所以你其实根本不需要把代码分开在多个文件或者目录里面，同样可以完成代码的模块化。我可以把代码全都写在同一个文件里，却仍然是非常模块化的代码。

想要达到很好的模块化，你需要做到以下几点：

- 避免写太长的函数。如果发现函数太大了，就应该把它拆分成几个更小的。通常我写的函数长度都不超过40行。对比一下，一般笔记本电脑屏幕所能容纳的代码行数是50行。我可以一目了然的看见一个40行的函数，而不需要滚屏。只有40行而不是50行的原因是，我的眼球不转的话，最大的视角只看得到40行代码。

如果我看代码不转眼珠的话，我就能把整片代码完整的映射到我的视觉神经里，这样就算忽然闭上眼睛，我也能看得见这段代码。我发现闭上眼睛的时候，大脑能够更加有效地处理代码，你能想象这段代码可以变成什么其它的形状。40行并不是一个很大的限制，因为函数里面比较复杂的部分，往往早就被我提取出去，做成了更小的函数，然后从原来的函数里面调用。

- 制造小的工具函数。如果你仔细观察代码，就会发现其实里面有很多的重复。这些常用的代码，不管它有多短，提取出去做成函数，都可能是会有好处的。有些帮助函数也许就只有两行，然而它们却能大大简化主要函数里面的逻辑。

有些人不喜欢使用小的函数，因为他们想避免函数调用的开销，结果他们写出几百行之大的函数。这是一种过时的观念。现代的编译器都能自动的把小的函数内联（inline）到调用它的地方，所以根本不产生函数调用，也就不会产生任何多余的开销。

同样的一些人，也爱使用宏（macro）来代替小函数，这也是一种过时的观念。在早期的C语言编译器里，只有宏是静态“内联”的，所以他们使用宏，其实是为了达到内联的目的。然而能否内联，其实并不是宏与函数的根本区别。宏与函数有着巨大的区别（这个我以后再讲），应该尽量避免使用宏。为了内联而使用宏，其实是滥用了宏，这会引起各种各样的麻烦，比如使程序难以理解，难以调试，容易出错等等。

- 每个函数只做一件简单的事情。有些人喜欢制造一些“通用”的函数，既可以做这个又可以做那个，它的内部依据某些变量和条件，来“选择”这个函数所要做的事情。比如，你也许写出这样的函数：

```
void foo() {
    if (getOS().equals("MacOS")) {
        a();
    } else {
        b();
    }
    c();
    if (getOS().equals("MacOS")) {
        d();
    } else {
        e();
    }
}
```

写这个函数的人，根据系统是否为“MacOS”来做不同的事情。你可以看出这个函数里，其实只有c()是两种系统共有的，而其它的a(), b(), d(), e()都属于不同的分支。

这种“复用”其实是有害的。如果一个函数可能做两种事情，它们之间共同点少于它们的不同点，那你最好就写两个不同的函数，否则这个函数的逻辑就不会很清晰，容易出现错误。其实，上面这个函数可以改写成两个函数：

```
void fooMacOS() {
    a();
    c();
    d();
}
```

和

```
void fooOther() {
    b();
    c();
    e();
}
```

如果你发现两件事情大部分内容相同，只有少数不同，多半时候你可以把相同的部分提取出去，做成一个辅助函数。比如，如果你有个函数是这样：

```
void foo() {
    a();
    b()
    c();
    if (getOS().equals("MacOS")) {
        d();
    } else {
        e();
    }
}
```

```
    }
```

其中a()，b()，c()都是一样的，只有d()和e()根据系统有所不同。那么你可以把a()，b()，c()提取出去：

```
void preFoo() {
    a();
    b();
    c();
```

然后制造两个函数：

```
void fooMacOS() {
    preFoo();
    d();
}
```

和

```
void fooOther() {
    preFoo();
    e();
}
```

这样一来，我们既共享了代码，又做到了每个函数只做一件简单的事情。这样的代码，逻辑就更加清晰。

- 避免使用全局变量和类成员（class member）来传递信息，尽量使用局部变量和参数。有些人写代码，经常用类成员来传递信息，就像这样：

```
class A {
    String x;

    void findX() {
        ...
        x = ...;
    }

    void foo() {
        findX();
        ...
        print(x);
    }
}
```

首先，他使用findX()，把一个值写入成员x。然后，使用x的值。这样，x就变成了findX和print之间的数据通道。由于x属于class A，这样程序就失去了模块化的结构。由于这两个函数依赖于成员x，它们不再有明确的输入和输出，而是依赖全局的数据。findX和foo不再能够离开class A而存在，而且由于类成员还有可能被其他代码改变，代码变得难以理解，难以确保正确性。

如果你使用局部变量而不是类成员来传递信息，那么这两个函数就不需要依赖于某一个class，而且更容易理解，不易出错：

```
String findX() {
    ...
    x = ...;
    return x;
}

void foo() {
    String x = findX();
    print(x);
}
```

写可读的代码

有些人以为写很多注释就可以让代码更加可读，然而却发现事与愿违。注释不但没能让代码变得可读，反而由于大量的注释充斥在代码中间，让程序变得障眼难读。而且代码的逻辑一旦修改，就会有很多的注释变得过时，需要更新。修改注释是相当大的负担，所以大量的注释，反而成为了妨碍改进代码的绊脚石。

实际上，真正优雅可读的代码，是几乎不需要注释的。如果你发现需要写很多注释，那么你的代码肯定是含混晦涩，逻辑不清晰的。其实，程序语言相比自然语言，是更加强大而严谨的，它其实具有自然语言最主要的元素：主语，谓语，宾语，名词，动词，如果，那么，否则，是，不是，…… 所以如果你充分利用了程序语言的表达能力，你完全可以用程序本身来表达它到底在干什么，而不需要自然语言的辅助。

有少数的时候，你也许会为了绕过其他一些代码的设计问题，采用一些违反直觉的作法。这时候你可以使用很短注释，说明为什么要写成那奇怪的样子。这样的情况应该少出现，否则这意味着整个代码的设计都有问题。

如果没能合理利用程序语言提供的优势，你会发现程序还是很难懂，以至于需要写注释。所以我现在告诉你一些要点，也许可以帮助你大大减少写注释的必要：

1. 使用有意义的函数和变量名字。如果你的函数和变量的名字，能够切实的描述它们的逻辑，那么你就不需要写注释来解释它在干什么。比如：

```
// put elephant1 into fridge2
put(elephant1, fridge2);
```

由于我的函数名put，加上两个有意义的变量名elephant1和fridge2，已经说明了这是在干什么（把大象放进冰箱），所以上面那句注释完全没有必要。

2. 局部变量应该尽量接近使用它的地方。有些人喜欢在函数最开头定义很多局部变量，然后在下面很远的地方使用它，就像这个样子：

```
void foo() {
    int index = ...;
    ...
    ...
    bar(index);
    ...
}
```

由于这中间都没有使用过index，也没有改变过它所依赖的数据，所以这个变量定义，其实可以挪到接近使用它的地方：

```
void foo() {
    ...
    ...
    int index = ...;
    bar(index);
    ...
}
```

这样读者看到bar(index)，不需要向上看很远就能发现index是如何算出来的。而且这种短距离，可以加强读者对于这里的“计算顺序”的理解。否则如果index在顶上，读者可能会怀疑，它其实保存了某种会变化的数据，或者它后来又被修改过。如果index放在下面，读者就清楚的知道，index并不是保存了什么可变的值，而且它算出来之后就没变过。

如果你看透了局部变量的本质——它们就是电路里的导线，那你就能更好的理解近距离的好处。变量定义离用的地方越近，导线的长度就越短。你不需要摸着一根导线，绕来绕去找很远，就能发现接收它的端口，这样的电路就更容易理解。

3. 局部变量名字应该简短。这貌似跟第一点相冲突，简短的变量名怎么可能有意义呢？注意我这里说的是局部变量，因为它们处于局部，再加上第2点已经把它放到离使用位置尽量近的地方，所以根据上下文你就会容易知道它的意思：

比如，你有一个局部变量，表示一个操作是否成功：

```
boolean successInDeleteFile = deleteFile("foo.txt");
if (successInDeleteFile) {
    ...
} else {
    ...
}
```

这个局部变量successInDeleteFile大可不必这么啰嗦。因为它只用过一次，而且用它的地方就在下面一行，所以读者可以轻松发现它是deleteFile返回的结果。如果你把它改名为success，其实读者根据一点上下文，也知道它表示“success in deleteFile”。所以你可以把它改成这样：

```
boolean success = deleteFile("foo.txt");
if (success) {
    ...
} else {
    ...
}
```

这样的写法不但没漏掉任何有用的语义信息，而且更加易读。successInDeleteFile这种“[camelCase](#)”，如果超过了三个单词连在一起，其实是很碍眼的东西。所以如果你能用一个单词表示同样的意义，那当然更好。

4. 不要重用局部变量。很多人写代码不喜欢定义新的局部变量，而喜欢“重用”同一个局部变量，通过反复对它们进行赋值，来表示完全不同意思。比如这样写：

```
String msg;
if (...) {
    msg = "succeed";
    log.info(msg);
} else {
    msg = "failed";
    log.info(msg);
}
```

虽然这样在逻辑上是没有问题的，然而却不易理解，容易混淆。变量msg两次被赋值，表示完全不同的两个值。它们立即被log.info使用，没有传递到其它地方去。这种赋值的做法，把局部变量的作用域不必要的增大，让人以为它可能在将来改变，也许会在其它地方被使用。更好的做法，其实是定义两个变量：

```
if (...) {
    String msg = "succeed";
    log.info(msg);
} else {
    String msg = "failed";
    log.info(msg);
}
```

由于这两个msg变量的作用域仅限于它们所处的if语句分支，你可以很清楚的看到这两个msg被使用的范围，而且知道它们之间没有任何关系。

5. 把复杂的逻辑提取出去，做成“帮助函数”。有些人写的函数很长，以至于看不清楚里面的语句在干什么，所以他们误以为需要写注释。如果你仔细观察这些代码，就会发现不清晰的那片代码，往往可以被提取出去，做成一个函数，然后在原来的地方调用。由于函数有一个名字，这样你就可以使用有意义的函数名来代替注释。举一个例子：

```
...
// put elephant1 into fridge2
openDoor(fridge2);
if (elephant1.alive()) {
    ...
} else {
    ...
}
closeDoor(fridge2);
...
```

如果你把这片代码提出去定义成一个函数：

```
void put(Elephant elephant, Fridge fridge) {
    openDoor(fridge);
    if (elephant.alive()) {
        ...
    } else {
        ...
    }
    closeDoor(fridge);
}
```

这样原来的代码就可以改成：

```
...
put(elephant1, fridge2);
...
```

更加清晰，而且注释也没必要了。

6. 把复杂的表达式提取出去，做成中间变量。有些人听说“函数式编程”是个好东西，也不理解它的真正含义，就在代码里大量使用嵌套的函数。像这样：

```
Pizza pizza = makePizza(crust(salt(), butter()),
    topping(onion(), tomato(), sausage()));
```

这样的代码一行太长，而且嵌套太多，不容易看清楚。其实训练有素的函数式程序员，都知道中间变量的好处，不会盲目的使用嵌套的函数。他们会把这代码变成这样：

```
Crust crust = crust(salt(), butter());
```

```
Topping topping = topping(onion(), tomato(), sausage());
Pizza pizza = makePizza(crust, topping);
```

这样写，不但有效地控制了单行代码的长度，而且由于引入的中间变量具有“意义”，步骤清晰，变得很容易理解。

7. 在合理的地方换行。对于绝大部分的程序语言，代码的逻辑是和空白字符无关的，所以你可以在几乎任何地方换行，你也可以不换行。这样的语言设计是个好东西，因为它给了程序员自由控制自己代码格式的能力。然而，它也引起了一些问题，因为很多人不知道如何合理的换行。

有些人喜欢利用IDE的自动换行机制，编辑之后用一个热键把整个代码重新格式化一遍，IDE就会把超过行宽限制的代码自动折行。可是这种自动这行，往往没有根据代码的逻辑来进行，不能帮助理解代码。自动换行之后可能产生这样的代码：

```
if (someLongCondition1() && someLongCondition2() && someLongCondition3() &&
    someLongCondition4()) {
    ...
}
```

由于someLongCondition4()超过了行宽限制，被编辑器自动换到了下面一行。虽然满足了行宽限制，换行的位置却是相当任意的，它并不能帮助人理解这代码的逻辑。这几个boolean表达式，全都用&&连接，所以它们其实处于平等的地位。为了表达这一点，当需要折行的时候，你应该把每一个表达式都放到新的一页，就像这个样子：

```
if (someLongCondition1() &&
    someLongCondition2() &&
    someLongCondition3() &&
    someLongCondition4()) {
    ...
}
```

这样每一个条件都对齐，里面的逻辑就很清楚了。再举个例子：

```
log.info("failed to find file {} for command {}, with exception {}", file, command,
         exception);
```

这行因为太长，被自动折行成这个样子。file，command和exception本来是同一类东西，却有两个留在了第一行，最后一个被折到第二行。它就不如手动换行成这个样子：

```
log.info("failed to find file {} for command {}, with exception {}", file, command, exception);
```

把格式字符串单独放在一行，而把它的参数一并放在另外一行，这样逻辑就更加清晰。

为了避免IDE把这些手动调整好的换行弄乱，很多IDE（比如IntelliJ）的自动格式化设定里都有“保留原来的换行符”的设定。如果你发现IDE的换行不符合逻辑，你可以修改这些设定，然后在某些地方保留你自己的手动换行。

说到这里，我必须警告你，这里所说的“不需注释，让代码自己解释自己”，并不是说要让代码看起来像某种自然语言。有个叫Chai的JavaScript测试工具，可以让你这样写代码：

```
expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(tea).to.have.property('flavors').with.length(3);
```

这种做法是极其错误的。程序语言本来就比自然语言简单清晰，这种写法让它看起来像自然语言的样子，反而变得复杂难懂了。

写简单的代码

程序语言都喜欢标新立异，提供这样那样的“特性”，然而有些特性其实并不是什么好东西。很多特性都经不起时间的考验，最后带来的麻烦，比解决的问题还多。很多人盲目的追求“短小”和“精悍”，或者为了显示自己头脑聪明，学得快，所以喜欢利用语言里的一些特殊构造，写出过于“聪明”，难以理解的代码。

并不是语言提供什么，你就一定要把它用上的。实际上你只需要其中很小的一部分功能，就能写出优秀的代码。我一向反对“充分利用”程序语言里的所有特性。实际上，我心目中有一套最好的构造。不管语言提供了多么“神奇”的，“新”的特性，我基本都只用经过千锤百炼，我觉得值得信赖的那一套。

现在针对一些有问题的语言特性，我介绍一些我自己使用的代码规范，并且讲解一下为什么它们能让代码更简单。

- 避免使用自增减表达式（`i++`, `++i`, `i-`, `-i`）。这种自增减操作表达式其实是历史遗留的设计失误。它们含义蹊跷，非常容易弄错。它们把读和写这两种完全不同的操作，混淆缠绕在一起，把语义搞得乌七八糟。含有它们

的表达式，结果可能取决于求值顺序，所以它可能在某种编译器下能正确运行，换一个编译器就出现离奇的错误。

其实这两个表达式完全可以分解成两步，把读和写分开：一步更新i的值，另外一步使用i的值。比如，如果你想写`foo(i++)`，你完全可以把它拆成`int t = i; i += 1; foo(t);`。如果你想写`foo(++i)`，可以拆成`i += 1; foo(i);`。拆开之后的代码，含义完全一致，却清晰很多。到底更新是在取值之前还是之后，一目了然。

有人也许以为`i++`或者`++i`的效率比拆开之后要高，这只是一种错觉。这些代码经过基本的编译器优化之后，生成的机器代码是完全没有区别的。自增减表达式只有在两种情况下才可以安全的使用。一种是在for循环的update部分，比如`for(int i = 0; i < 5; i++)`。另一种情况是写成单独的一行，比如`i++;`。这两种情况是完全没有歧义的。你需要避免其它的情况，比如用在复杂的表达式里面，比如`foo(i++)`, `foo(++i) + foo(i)`,没有人应该知道，或者去追究这些是什么意思。

- 永远不要省略花括号。很多语言允许你在某种情况下省略掉花括号，比如C, Java都允许你在if语句里面只有一句话的时候省略掉花括号：

```
if (...)  
    action1();
```

乍一看少打了两个字，多好。可是这其实经常引起奇怪的问题。比如，你后来想要加一句话`action2()`到这个if里面，于是你就把代码改成：

```
if (...)  
    action1();  
    action2();
```

为了美观，你很小心的使用了`action1()`的缩进。乍一看它们是在一起的，所以你下意识里以为它们只会在if的条件为真的时候执行，然而`action2()`却其实在if外面，它会被无条件的执行。我把这种现象叫做“光学幻觉”(optical illusion)，理论上每个程序员都应该发现这个错误，然而实际上却容易被忽视。

那么你问，谁会这么傻，我在加入`action2()`的时候加上花括号不就行了？可是从设计的角度来看，这样其实并不是合理的作法。首先，也许你以后又想把`action2()`去掉，这样你为了样式一致，又得把花括号拿掉，烦不烦啊？其次，这使得代码样式不一致，有的if有花括号，有的又没有。况且，你为什么需要记住这个规则？如果你不问三七二十一，只要是if-else语句，把花括号全都打上，就可以想都不用想了，就当C和Java没提供给你这个特殊写法。这样就可以保持完全的一致性，减少不必要的思考。

有人可能会说，全都打上花括号，只有一句话也打上，多碍眼啊？然而经过实行这种编码规范几年之后，我并没有发现这种写法更加碍眼，反而由于花括号的存在，使得代码界限明确，让我的眼睛负担更小了。

- 合理使用括号，不要盲目依赖操作符优先级。利用操作符的优先级来减少括号，对于`1 + 2 * 3`这样常见的算数表达式，是没问题的。然而有些人如此的仇恨括号，以至于他们会写出`2 << 7 - 2 * 3`这样的表达式，而完全不用括号。

这里的问题，在于移位操作`<<`的优先级，是很多人不熟悉，而且是违反常理的。由于`x << 1`相当于把x乘以2，很多人误以为这个表达式相当于`(2 << 7) - (2 * 3)`，所以等于250。然而实际上`<<`的优先级比加法+还要低，所以这表达式其实相当于`2 << (7 - 2 * 3)`，所以等于4！

解决这个问题的办法，不是要每个人去把操作符优先级表给硬背下来，而是合理的加入括号。比如上面的例子，最好直接加上括号写成`2 << (7 - 2 * 3)`。虽然没有括号也表示同样的意思，但是加上括号就更加清晰，读者不再需要死记`<<`的优先级就能理解代码。

- 避免使用`continue`和`break`。循环语句(for, while) 里面出现`return`是没问题的，然而如果你使用了`continue`或者`break`，就会让循环的逻辑和终止条件变得复杂，难以确保正确。

出现`continue`或者`break`的原因，往往是对循环的逻辑没有想清楚。如果你考虑周全了，应该是几乎不需要`continue`或者`break`的。如果你的循环里出现了`continue`或者`break`，你就应该考虑改写这个循环。改写循环的办法有多种：

- 如果出现了`continue`，你往往只需要把`continue`的条件反向，就可以消除`continue`。
- 如果出现了`break`，你往往可以把`break`的条件，合并到循环头部的终止条件里，从而去掉`break`。
- 有时候你可以把`break`替换成`return`，从而去掉`break`。
- 如果以上都失败了，你也许可以把循环里面复杂的部分提取出来，做成函数调用，之后`continue`或者`break`就可以去掉了。

下面我对这些情况举一些例子。

情况1：下面这段代码里面有一个`continue`：

```
List<String> goodNames = new ArrayList<>();  
for (String name: names) {
```

```

if (name.contains("bad")) {
    continue;
}
goodNames.add(name);
...
}

```

它说：“如果name含有‘bad’这个词，跳过后面的循环代码……”注意，这是一种“负面”的描述，它不是在告诉你什么时候“做”一件事，而是在告诉你什么时候“不做”一件事。为了知道它到底在干什么，你必须搞清楚continue会导致哪些语句被跳过了，然后脑子里把逻辑反个向，你才能知道它到底想做什么。这就是为什么含有continue和break的循环不容易理解，它们依靠“控制流”来描述“不做什么”，“跳过什么”，结果到最后你也搞不清楚它到底“要做什么”。

其实，我们只需要把continue的条件反向，这段代码就可以很容易的被转换成等价的，不含continue的代码：

```

List<String> goodNames = new ArrayList<>();
for (String name: names) {
    if (!name.contains("bad")) {
        goodNames.add(name);
    ...
}

```

goodNames.add(name);和它之后的代码全部被放到了if里面，多了一层缩进，然而continue却没有了。你再读这段代码，就会发现更加清晰。因为它是一种更加“正面”地描述。它说：“在name不含有‘bad’这个词的时候，把它加到goodNames的链表里面……”

情况2：for和while头部都有一个循环的“终止条件”，那本来应该是这个循环唯一的退出条件。如果你在循环中间有break，它其实给这个循环增加了一个退出条件。你往往只需要把这个条件合并到循环头部，就可以去掉break。

比如下面这段代码：

```

while (condition1) {
    ...
    if (condition2) {
        break;
    }
}

```

当condition成立的时候，break会退出循环。其实你只需要把condition2反转之后，放到while头部的终止条件，就可以去掉这种break语句。改写后的代码如下：

```

while (condition1 && !condition2) {
    ...
}

```

这种情况表面上貌似只适用于break出现在循环开头或者末尾的时候，然而其实大部分时候，break都可以通过某种方式，移动到循环的开头或者末尾。具体的例子我暂时没有，等出现的时候再加进来。

情况3：很多break退出循环之后，其实接下来就是一个return。这种break往往可以直接换成return。比如下面这个例子：

```

public boolean hasBadName(List<String> names) {
    boolean result = false;

    for (String name: names) {
        if (name.contains("bad")) {
            result = true;
            break;
        }
    }
    return result;
}

```

这个函数检查names链表里是否存在一个名字，包含“bad”这个词。它的循环里包含一个break语句。这个函数可以被改写成：

```

public boolean hasBadName(List<String> names) {
    for (String name: names) {
        if (name.contains("bad")) {
            return true;
        }
    }
}

```

```
    }
    return false;
}
```

改进后的代码，在name里面含有“bad”的时候，直接用return true返回，而不是对result变量赋值，break出去，最后才返回。如果循环结束了还没有return，那就返回false，表示没有找到这样的名字。使用return来代替break，这样break语句和result这个变量，都一并被消除掉了。

我曾经见过很多其他使用continue和break的例子，几乎无一例外的可以被消除掉，变换后的代码变得清晰很多。我的经验是，99%的break和continue，都可以通过替换成return语句，或者翻转if条件的方式来消除掉。剩下的1%含有复杂的逻辑，但也可以通过提取一个帮助函数来消除掉。修改之后的代码变得容易理解，容易确保正确。

写直观的代码

我写代码有一条重要的原则：如果有更加直接，更加清晰的写法，就选择它，即使它看起来更长，更笨，也一样选择它。比如，Unix命令行有一种“巧妙”的写法是这样：

```
command1 && command2 && command3
```

由于 Shell 语言的逻辑操作a && b具有“短路”的特性，如果a等于false，那么b就没必要执行了。这就是为什么当 command1 成功，才会执行 command2，当 command2 成功，才会执行 command3。同样，

```
command1 || command2 || command3
```

操作符||也有类似的特性。上面这个命令行，如果command1成功，那么command2和command3都不会被执行。如果command1失败，command2成功，那么command3就不会被执行。

这比起用if语句来判断失败，似乎更加巧妙和简洁，所以有人就借鉴了这种方式，在程序的代码里也使用这种方式。比如他们可能会写这样的代码：

```
if (action1() || action2() && action3()) {
...
}
```

你看得出来这代码是想干什么吗？action2和action3什么条件下执行，什么条件下不执行？也许稍微想一下，你知道它在干什么：“如果action1失败了，执行action2，如果action2成功了，执行action3”。然而那种语义，并不是直接的“映射”在这代码上面的。比如“失败”这个词，对应了代码里的哪一个字呢？你找不出来，因为它包含在了||的语义里面，你需要知道||的短路特性，以及逻辑或的语义才能知道这里面在说“如果action1失败……”。每一次看到这行代码，你都需要思考一下，这样积累起来的负荷，就会让人很累。

其实，这种写法是滥用了逻辑操作&&和||的短路特性。这两个操作符可能不执行右边的表达式，原因是为了机器的执行效率，而不是为了给人提供这种“巧妙”的用法。这两个操作符的本意，只是作为逻辑操作，它们并不是拿来给你代替if语句的。也就是说，它们只是碰巧可以达到某些if语句的效果，但你不应该因此就用它来代替if语句。如果你这样做了，就会让代码晦涩难懂。

上面的代码写成笨一点的办法，就会清晰很多：

```
if (!action1()) {
    if (action2()) {
        action3();
    }
}
```

这里我很明显的看出这代码在说什么，想都不用想：如果action1()失败了，那么执行action2()，如果action2()成功了，执行action3()。你发现这里面的一一对应关系吗？if=如果，!=失败，…… 你不需要利用逻辑学知识，就知道它在说什么。

写无懈可击的代码

在之前一节里，我提到了自己写的代码里面很少出现只有一个分支的if语句。我写出的if语句，大部分都有两个分支，所以我的代码很多看起来是这个样子：

```
if (...) {
    if (...) {
        ...
        return false;
    } else {
        return true;
    }
} else if (...) {
```

```
...
    return false;
} else {
    return true;
}
```

使用这种方式，其实是为了无懈可击的处理所有可能出现的情况，避免漏掉corner case。每个if语句都有两个分支的理由是：如果if的条件成立，你做某件事情；但是如果if的条件不成立，你应该知道要做什么另外的事情。不管你的if有没有else，你终究是逃不掉，必须得思考这个问题的。

很多人写if语句喜欢省略else的分支，因为他们觉得有些else分支的代码重复了。比如我的代码里，两个else分支都是return true。为了避免重复，他们省略掉那两个else分支，只在最后使用一个return true。这样，缺了else分支的if语句，控制流自动“掉下去”，到达最后的return true。他们的代码看起来像这个样子：

```
if (...) {
    if (...) {
        ...
        return false;
    }
} else if (...) {
    ...
    return false;
}
return true;
```

这种写法看似更加简洁，避免了重复，然而却很容易出现疏忽和漏洞。嵌套的if语句省略了一些else，依靠语句的“控制流”来处理else的情况，是很难正确的分析和推理的。如果你的if条件里使用了&&和||之类的逻辑运算，就更难看出是否涵盖了所有的情况。

由于疏忽而漏掉的分支，全都会自动“掉下去”，最后返回意想不到的结果。即使你看一遍之后确信是正确的，每次读这段代码，你都不能确信它照顾了所有的情况，又得重新推理一遍。这简洁的写法，带来的是反复的，沉重的头脑开销。这就是所谓“面条代码”，因为程序的逻辑分支，不是像一棵枝叶分明的树，而是像面条一样绕来绕去。

另外一种省略else分支的情况是这样：

```
String s = "";
if (x < 5) {
    s = "ok";
}
```

写这段代码的人，脑子里喜欢使用一种“缺省值”的做法。s缺省为null，如果x<5，那么把它改变(mutate)成“ok”。这种写法的缺点是，当x<5不成立的时候，你需要往上面看，才能知道s的值是什么。这还是你运气好的时候，因为s就在上面不远。很多人写这种代码的时候，s的初始值离判断语句有一定的距离，中间还有可能插入一些其它的逻辑和赋值操作。这样的代码，把变量改来改去的，看得人眼花，就容易出错。

现在比较一下我的写法：

```
String s;
if (x < 5) {
    s = "ok";
} else {
    s = "";
}
```

这种写法貌似多打了一两个字，然而它却更加清晰。这是因为我们明确的指出了x<5不成立的时候，s的值是什么。它就摆在那里，它是""(空字符串)。注意，虽然我也使用了赋值操作，然而我并没有“改变”s的值。s一开始的时候没有值，被赋值之后就再也没有变过。我的这种写法，通常被叫做更加“函数式”，因为我只赋值一次。

如果我漏写了else分支，Java编译器是不会放过我的。它会抱怨：“在某个分支，s没有被初始化。”这就强迫我清清楚楚的设定各种条件下s的值，不漏掉任何一种情况。

当然，由于这个情况比较简单，你还可以把它写成这样：

```
String s = x < 5 ? "ok" : "";
```

对于更加复杂的情况，我建议还是写成if语句为好。

正确处理错误

使用有两个分支的if语句，只是我的代码可以达到无懈可击的其中一个原因。这样写if语句的思路，其实包含了使代码可靠的一种通用思想：穷举所有的情况，不漏掉任何一个。

程序的绝大部分功能，是进行信息处理。从一堆纷繁复杂，模棱两可的信息中，排除掉绝大部分“干扰信息”，找到自己需要的那一个。正确地对所有的“可能性”进行推理，就是写出无懈可击代码的核心思想。这一节我来讲一讲，如何把这种思想用在错误处理上。

错误处理是一个古老的问题，可是经过了几十年，还是很多人没搞明白。Unix的系统API手册，一般都会告诉你可能出现的返回值和错误信息。比如，Linux的[read](#)系统调用手册里面有如下内容：

```
RETURN VALUE  
On success, the number of bytes read is returned...  
  
On error, -1 is returned, and errno is set appropriately.
```

ERRORS

```
EAGAIN, EBADF, EFAULT, EINTR, EINVAL, ...
```

很多初学者，都会忘记检查read的返回值是否为-1，觉得每次调用read都得检查返回值真繁琐，不检查貌似也相安无事。这种想法其实是很危险的。如果函数的返回值告诉你，要么返回一个正数，表示读到的数据长度，要么返回-1，那么你就必须要对这个-1作出相应的，有意义的处理。千万不要以为你可以忽视这个特殊的返回值，因为它是一种“可能性”。代码漏掉任何一种可能出现的情况，都可能产生意想不到的灾难性结果。

对于Java来说，这相对方便一些。Java的函数如果出现问题，一般通过异常（exception）来表示。你可以把异常加上函数本来的返回值，看成是一个“union类型”。比如：

```
String foo() throws MyException {  
    ...  
}
```

这里MyException是一个错误返回。你可以认为这个函数返回一个union类型：{String, MyException}。任何调用foo的代码，必须对MyException作出合理的处理，才有可能确保程序的正确运行。Union类型是一种相当先进的类型，目前只有极少数语言（比如Typed Racket）具有这种类型，我在这里提到它，只是为了方便解释概念。掌握了概念之后，你其实可以在头脑里实现一个union类型系统，这样使用普通的语言也能写出可靠的代码。

由于Java的类型系统强制要求函数在类型里面声明可能出现的异常，而且强制调用者处理可能出现的异常，所以基本上不可能出现由于疏忽而漏掉的情况。但有些Java程序员有一种恶习，使得这种安全机制几乎完全失效。每当编译器报错，说“你没有catch这个foo函数可能出现的异常”时，有些人想都不想，直接把代码改成这样：

```
try {  
    foo();  
} catch (Exception e) {}
```

或者最多在里面放个log，或者干脆把自己的函数类型上加上throws Exception，这样编译器就不再抱怨。这些做法貌似很省事，然而都是错误的，你终究会为此付出代价。

如果你把异常catch了，忽略掉，那么你就不知道foo其实失败了。这就像开车时看到路口写着“前方施工，道路关闭”，还继续往前开。这当然迟早会出问题，因为你根本不知道自己在干什么。

catch异常的时候，你不应该使用Exception这么宽泛的类型。你应该正好catch可能发生的那种异常A。使用宽泛的异常类型有很大的问题，因为它会不经意的catch住另外的异常（比如B）。你的代码逻辑是基于判断A是否出现，可你却catch所有的异常（Exception类），所以当其它的异常B出现的时候，你的代码就会出现莫名其妙的问题，因为你以为A出现了，而其实它没有。这种bug，有时候甚至使用debugger都难以发现。

如果你在自己函数的类型加上throws Exception，那么你就不可避免的需要在调用它的地方处理这个异常，如果调用它的函数也写着throws Exception，这毛病就传得更远。我的经验是，尽量在异常出现的当时就作出处理。否则如果你把它返回给你的调用者，它也许根本不知道该怎么办了。

另外，try { ... } catch里面，应该包含尽量少的代码。比如，如果foo和bar都可能产生异常A，你的代码应该尽可能写成：

```
try {  
    foo();  
} catch (A e) {...}  
  
try {  
    bar();  
} catch (A e) {...}
```

而不是

```
try {  
    foo();  
    bar();  
}
```

```
} catch (A e) {...}
```

第一种写法能明确的分辨是哪一个函数出了问题，而第二种写法全都混在一起。明确的分辨是哪一个函数出了问题，有很多的好处。比如，如果你的catch代码里面包含log，它可以提供给你更加精确的错误信息，这样会大大地加速你的调试过程。

正确处理null指针

穷举的思想是如此的有用，依据这个原理，我们可以推出一些基本原则，它们可以让你无懈可击的处理null指针。

首先你应该知道，许多语言（C，C++，Java，C#，……）的类型系统对于null的处理，其实是完全错误的。这个错误源自于[Tony Hoare](#)最早的设计，Hoare把这个错误称为自己的“[billion dollar mistake](#)”，因为由于它所产生的财产和人力损失，远远超过十亿美元。

这些语言的类型系统允许null出现在任何对象（指针）类型可以出现的地方，然而null其实根本不是一个合法的对象。它不是一个String，不是一个Integer，也不是一个自定义的类。null的类型本来应该是NULL，也就是null自己。根据这个基本观点，我们推导出以下原则：

- 尽量不要产生null指针。尽量不要用null来初始化变量，函数尽量不要返回null。如果你的函数要返回“没有”，“出错了”之类的结果，尽量使用Java的异常机制。虽然写法上有点别扭，然而Java的异常，和函数的返回值合并在一起，基本上可以当成union类型来用。比如，如果你有一个函数find，可以帮你找到一个String，也有可能什么也找不到，你可以这样写：

```
public String find() throws NotFoundException {  
    if (...) {  
        return ...;  
    } else {  
        throw new NotFoundException();  
    }  
}
```

Java的类型系统会强制你catch这个NotFoundException，所以你不可能像漏掉检查null一样，漏掉这种情况。Java的异常也是一个比较容易滥用的东西，不过我已经在上一节告诉你如何正确的使用异常。

Java的try...catch语法相当的繁琐和蹩脚，所以如果你足够小心的话，像find这类函数，也可以返回null来表示“没找到”。这样稍微好看一些，因为你调用的时候不必用try...catch。很多人写的函数，返回null来表示“出错了”，这其实是对null的误用。“出错了”和“没有”，其实完全是两码事。“没有”是一种很常见，正常的情况，比如查哈希表没找到，很正常。“出错了”则表示罕见的情况，本来正常情况下都应该存在有意义的值，偶然出了问题。如果你的函数要表示“出错了”，应该使用异常，而不是null。

- 不要catch NullPointerException。有些人写代码很nice，他们喜欢“容错”。首先他们写一些函数，这些函数里面不大小心，没检查null指针：

```
void foo() {  
    String found = find();  
    int len = found.length();  
    ...  
}
```

当foo调用产生了异常，他们不管三七二十一，就把调用的地方改成这样：

```
try {  
    foo();  
} catch (Exception e) {  
    ...  
}
```

这样当found是null的时候，NullPointerException就会被捕获并且得到处理。这其实是很错误的作法。首先，上一节已经提到了，catch (Exception e)这种写法是要绝对避免的，因为它捕获所有的异常，包括NullPointerException。这会让你意外地捕获try语句里面出现的NullPointerException，从而把代码的逻辑搅得一塌糊涂。

另外就算你写成catch (NullPointerException e)也是不可以的。由于foo的内部缺少了null检查，才出现了NullPointerException。现在你不对症下药，倒把每个调用它的地方加上catch，以后你的生活就会越来越苦。正确的做法应该是改动foo，而不改调用它的代码。foo应该被改成这样：

```
void foo() {  
    String found = find();  
    if (found != null) {  
        int len = found.length();  
        ...  
    }  
}
```

```
    } else {  
        ...  
    }  
}
```

在null可能出现的当时就检查它是否是null，然后进行相应的处理。

- 不要把null放进“容器数据结构”里面。所谓容器（collection），是指一些对象以某种方式集合在一起，所以null不应该被放进Array，List，Set等结构，不应该出现在Map的key或者value里面。把null放进容器里面，是一些莫名其妙错误的来源。因为对象在容器里的位置一般是动态决定的，所以一旦null从某个入口跑进去了，你就很难再搞明白它去了哪里，你就得被迫在所有从这个容器里取值的位置检查null。你也很难知道到底是谁把它放进去的，代码多了就导致调试极其困难。

解决方案是：如果你真要表示“没有”，那你就干脆不要把它放进去（Array，List，Set没有元素，Map根本没那个entry），或者你可以指定一个特殊的，真正合法的对象，用来表示“没有”。

需要指出的是，类对象并不属于容器。所以null在必要的时候，可以作为对象成员的值，表示它不存在。比如：

```
class A {  
    String name = null;  
    ...  
}
```

之所以可以这样，是因为null只可能在A对象的name成员里出现，你不用怀疑其它的成员因此成为null。所以你每次访问name成员时，检查它是否是null就可以了，不需要对其他成员也做同样的检查。

- 函数调用者：明确理解null所表示的意义，尽早检查和处理null返回值，减少它的传播。null很讨厌的一个地方，在于它在不同的地方可能表示不同的意义。有时候它表示“没有”，“没找到”。有时候它表示“出错了”，“失败了”。有时候它甚至可以表示“成功了”，……这其中有很多误用之处，不过无论如何，你必须理解每一个null的意义，不能给混淆起来。

如果你调用的函数有可能返回null，那么你应该在第一时间对null做出“有意义”的处理。比如，上述的函数find，返回null表示“没找到”，那么调用find的代码就应该在它返回的第一时间，检查返回值是否是null，并且对“没找到”这种情况，作出有意义的处理。

“有意义”是什么意思呢？我的意思是，使用这函数的人，应该明确的知道在拿到null的情况下该怎么做，承担起责任来。他不应该只是“向上级汇报”，把责任踢给自己的调用者。如果你违反了这一点，就有可能采用一种不负责任，危险的写法：

```
public String foo() {  
    String found = find();  
    if (found == null) {  
        return null;  
    }  
}
```

当看到find()返回了null，foo自己也返回null。这样null就从一个地方，游走到了另一个地方，而且它表示另外一个意思。如果你不假思索就写出这样的代码，最后的结果就是代码里面随时随地都可能出现null。到后来为了保护自己，你的每个函数都会写成这样：

```
public void foo(A a, B b, C c) {  
    if (a == null) { ... }  
    if (b == null) { ... }  
    if (c == null) { ... }  
    ...  
}
```

- 函数作者：明确声明不接受null参数，当参数是null时立即崩溃。不要试图对null进行“容错”，不要让程序继续往下执行。如果调用者使用了null作为参数，那么调用者（而不是函数作者）应该对程序的崩溃负全责。

上面的例子之所以成为问题，就在于人们对于null的“容忍态度”。这种“保护式”的写法，试图“容错”，试图“优雅的处理null”，其结果是让调用者更加肆无忌惮的传递null给你的函数。到后来，你的代码里出现一堆堆nonsense的情况，null可以在任何地方出现，都不知道到底是哪里产生的。谁也不知道出现了null是什么意思，该做什么，所有人都把null踢给其他人。最后这null像瘟疫一样蔓延开来，到处都是，成为一场噩梦。

正确的做法，其实是强硬的态度。你要告诉函数的使用者，我的参数全都不能是null，如果你给我null，程序崩溃了该你自己负责。至于调用者代码里有null怎么办，他自己该知道怎么处理（参考以上几条），不应该由函数作者来操心。

采用强硬态度一个很简单的方法是使用Objects.requireNonNull()。它的定义很简单：

```

public static <T> T requireNonNull(T obj) {
    if (obj == null) {
        throw new NullPointerException();
    } else {
        return obj;
    }
}

```

你可以用这个函数来检查不想接受null的每一个参数，只要传进来的参数是null，就会立即触发`NullPointerException`崩溃掉，这样你就可以有效地防止null指针不知不觉传递到其它地方去。

- 使用`@NotNull`和`@Nullable`标记。IntelliJ提供了`@NotNull`和`@Nullable`两种标记，加在类型前面，这样可以比较简洁可靠地防止null指针的出现。IntelliJ本身会对含有这种标记的代码进行静态分析，指出运行时可能出现`NullPointerException`的地方。在运行时，会在null指针不该出现的地方产生`IllegalArgumentException`，即使那个null指针你从来没有dereference。这样你可以在尽量早期发现并且防止null指针的出现。
- 使用Optional类型。Java 8和Swift之类语言，提供了一种叫Optional的类型。正确的使用这种类型，可以在很大程度上避免null的问题。null指针的问题之所以存在，是因为你可以在没有“检查”null的情况下，“访问”对象的成员。

Optional类型的设计原理，就是把“检查”和“访问”这两个操作合二为一，成为一个“原子操作”。这样你没法只访问，而不进行检查。这种做法其实是ML，Haskell等语言里的模式匹配（pattern matching）的一个特例。模式匹配使得类型判断和访问成员这两种操作合二为一，所以你没法犯错。

比如，在Swift里面，你可以这样写：

```

let found = find()
if let content = found {
    print("found: " + content)
}

```

你从`find()`函数得到一个Optional类型的值`found`。假设它的类型是`String?`，那个问号表示它可能包含一个`String`，也可能是`nil`。然后你就可以用一种特殊的if语句，同时进行null检查和访问其中的内容。这个if语句跟普通的if语句不一样，它的条件不是一个Bool，而是一个变量绑定`let content = found`。

我不是很喜欢这语法，不过这整个语句的含义是：如果`found`是`nil`，那么整个if语句被略过。如果它不是`nil`，那么变量`content`被绑定到`found`里面的值（unwrap操作），然后执行`print("found: " + content)`。由于这种写法把检查和访问合并在了一起，你没法只进行访问而不检查。

Java 8的做法比较蹩脚一些。如果你得到一个`Optional<String>`类型的值`found`，你必须使用“函数式编程”的方式，来写之后的代码：

```

Optional<String> found = find();
found.ifPresent(content -> System.out.println("found: " + content));

```

这段Java代码跟上面的Swift代码等价，它包含一个“判断”和一个“取值”操作。`ifPresent`先判断`found`是否有值（相当于判断是不是`null`）。如果有，那么将其内容“绑定”到lambda表达式的`content`参数（unwrap操作），然后执行lambda里面的内容，否则如果`found`没有内容，那么`ifPresent`里面的lambda不执行。

Java的这种设计有个问题。判断null之后分支里的内容，全都得写在lambda里面。在函数式编程里，这个lambda叫做“continuation”，Java把它叫做“Consumer”，它表示“如果`found`不是`null`，拿到它的值，然后应该做什么”。由于lambda是个函数，你不能在里面写`return`语句返回出外层的函数。比如，如果你要改写下面这个函数（含有`null`）：

```

public static String foo() {
    String found = find();
    if (found != null) {
        return found;
    } else {
        return "";
    }
}

```

就会比较麻烦。因为如果你写成这样：

```

public static String foo() {
    Optional<String> found = find();
    found.ifPresent(content -> {
        return content; // can't return from foo here
    });
    return "";
}

```

里面的return a，并不能从函数foo返回出去。它只会从lambda返回，而且由于那个lambda ([Consumer.accept](#)) 的返回类型必须是void，编译器会报错，说你返回了String。由于Java里closure的自由变量是只读的，你没法对lambda外面的变量进行赋值，所以你也不能采用这种写法：

```
public static String foo() {  
    Optional<String> found = find();  
    String result = "";  
    found.ifPresent(content -> {  
        result = content; // can't assign to result  
    });  
    return result;  
}
```

所以，虽然你在lambda里面得到了found的内容，如何使用这个值，如何返回一个值，却让人摸不着头脑。你平时的那些Java编程手法，在这里几乎完全废掉了。实际上，判断null之后，你必须使用Java 8提供的一系列古怪的[函数式编程操作](#)：map, flatMap, orElse之类，想法把它们组合起来，才能表达出原来代码的意思。比如之前的代码，只能改写成这样：

```
public static String foo() {  
    Optional<String> found = find();  
    return found.orElse("");  
}
```

这简单的情况还好。复杂一点的代码，我还真不知道怎么表达，我怀疑Java 8的Optional类型的方法，到底有没有提供足够的表达力。那里面少数几个东西表达能力不咋的，论工作原理，却可以扯到functor，continuation，甚至monad等高深的理论……仿佛用了Optional之后，这语言就不再是Java了一样。

所以Java虽然提供了Optional，但我觉得可用性其实比较低，难以被人接受。相比之下，Swift的设计更加简单直观，接近普通的过程式编程。你只需要记住一个特殊的语法if let content = found {...}，里面的代码写法，跟普通的过程式语言没有任何差别。

总之你只要记住，使用Optional类型，要点在于“原子操作”，使得null检查与取值合二为一。这要求你必须使用我刚才介绍的特殊写法。如果你违反了这一原则，把检查和取值分成两步做，还是有可能犯错误。比如在Java 8里面，你可以使用found.get()这样的方式直接访问found里面的内容。在Swift里你也可以使用found!来直接访问而不进行检查。

你可以写这样的Java代码来使用Optional类型：

```
Option<String> found = find();  
if (found.isPresent()) {  
    System.out.println("found: " + found.get());  
}
```

如果你使用这种方式，把检查和取值分成两步做，就可能会出现运行时错误。if (found.isPresent())本质上跟普通的null检查，其实没什么两样。如果你忘记判断found.isPresent()，直接进行found.get()，就会出现NoSuchElementException。这跟NullPointerException本质上是一回事。所以这种写法，比起普通的null的用法，其实换汤不换药。如果你要用Optional类型而得到它的益处，请务必遵循我之前介绍的“原子操作”写法。

防止过度工程

人的脑子真是奇妙的东西。虽然大家都知道过度工程（over-engineering）不好，在实际的工程中却经常不由自主的出现过度工程。我自己也犯过好多次这种错误，所以觉得有必要分析一下，过度工程出现的信号和兆头，这样可以在初期的时候就及时发现并且避免。

过度工程即将出现的一个重要信号，就是当你过度的思考“将来”，考虑一些还没有发生的事情，还没有出现的需求。比如，“如果我们将来有了上百万行代码，有了几千号人，这样的工具就支持不了了”，“将来我可能需要这个功能，所以我现在就把代码写来放在那里”，“将来很多人要扩充这片代码，所以现在我们就让它变得可重用”……

这就是为什么很多软件项目如此复杂。实际上没做多少事情，却为了所谓的“将来”，加入了很多不必要的复杂性。眼前的问题还没解决呢，就被“将来”给拖垮了。人们都不喜欢目光短浅的人，然而在现实的工程中，有时候你就是得看近一点，把手头的问题先搞定了，再谈以后扩展的问题。

另外一种过度工程的来源，是过度的关心“代码重用”。很多人“可用”的代码还没写出来呢，就在关心“重用”。为了让代码可以重用，最后被自己搞出来的各种框架捆住手脚，最后连可用的代码就没写好。如果可用的代码都写不好，又何谈重用呢？很多一开头就考虑太多重用的工程，到后来被人完全抛弃，没人用了，因为别人发现这些代码太难懂了，自己从头开始写一个，反而省好多事。

过度地关心“测试”，也会引起过度工程。有些人为了测试，把本来很简单的代码改成“方便测试”的形式，结果引入很多复杂性，以至于本来一下就能写对的代码，最后复杂不堪，出现很多bug。

世界上有两种“没有bug”的代码。一种是“没有明显的bug的代码”，另一种是“明显没有bug的代码”。第一种情况，由于代码复杂不堪，加上很多测试，各种coverage，貌似测试都通过了，所以就认为代码是正确的。第二种情况，由于代码简单直接，就算没写很多测试，你一眼看去就知道它不可能有bug。你喜欢哪一种“没有bug”的代码呢？

根据这些，我总结出来的防止过度工程的原则如下：

1. 先把眼前的问题解决掉，解决好，再考虑将来的扩展问题。
2. 先写出可用的代码，反复推敲，再考虑是否需要重用的问题。
3. 先写出可用，简单，明显没有bug的代码，再考虑测试的问题。

完。

(这不是一篇免费的文章，如果你想把这些信息留在脑子里，请去[这里付费](#)。不然就请看[这里](#)：



图灵的光环

仿佛全世界的人都知道，[图灵](#) (Alan Turing) 是个天才，是他创造了计算机科学，是他破解了德国纳粹的 Enigma 密码。他被叫做“计算机之父”。由于他的杰出贡献，计算机科学的最高荣誉，被叫做“图灵奖”。然而根据自己的看法，我发现图灵本人的实际成就，相对于他所受到的崇拜，其实相差甚远。

由于二战以来各国政府对于当时谍报工作的保密措施造成的事情，再加上图灵的不幸生世所引来的同情，图灵这个名字似乎拥有了一种扑朔迷离的光环。人们把很多本来不是图灵作出的贡献归结在他身上，把本来很平常的贡献过分地夸大。图灵的光环，掩盖了许多对这些领域做出过更加重要贡献的人。

图灵传

2012年，在图灵诞辰[一百周年](#)的时候，人们风风火火的召开各种大会，纪念这位“计算机之父”，很多媒体也添油加醋地宣传他的丰功伟绩。还有个叫 Andrew Hodges 的人抓住这个时机，推销自己写的一本传记《[Alan Turing: The Enigma](#)》。这本书红极一时，后来还被改编成了电影。

这本传记看似客观，引经据典，字里行间却可以感受到作者对图灵个人的膜拜和偏袒，他在倾心打造一个“天才”。作者片面地使用对图灵有利的证据，对不利的方面只字不提。仿佛图灵做的一切都是有理的，他做的不好的地方都是因为别人的问题，或者风水不好。提到别人做的东西，尽是各种缺陷和局限性，不是缺陷也要说成是缺陷；提到图灵的工作，总是史无前例，开天辟地的发明。别人先做出来的东西，生拉硬拽，硬要说成是受了图灵的“启发”，还怪别人没有引用图灵的论文。这让你感觉仿佛别人都在抄袭图灵伟大的研究成果，都在利用他，欺负他似的。如果你不想花钱买书，可以看看同一作者写的一个[图灵简要生平](#)，足以从中感受到这种倾向。

我写这篇文章的很大一部分原因，就是因为这本传记。作者对图灵贡献的片面夸大，对其他一些学者的变相贬低，让我感到不平。图灵在计算机界的名声，本来就已经被严重的夸大和美化，被很多人盲目的崇拜。现在出了这本传记和电影，又在人们心中加重了这层误解。所以我觉得有必要澄清一些事实。

密码学

很多人提到二战 Enigma 密码的故事，就会把功劳一股脑地归到图灵头上，只字不提其他人。其实破解 Enigma 密码是很多人共同努力的结果，图灵只是其中的一员。这些人缺少了任何一个，都可能是灾难性的后果。其中好些人的想法早于图灵，启发过图灵，设计的东西比图灵的先进，却很少有人听说过他们的名字。图灵有自己的贡献，但最后说起来倒好像是他单枪匹马拯救了大家，这是不公平的。

最初破掉 Enigma 密码的其实不是英国人，而是波兰人。波兰人不但截获并且仿造了德国人的 Enigma 机器，而且发现了其中微妙的漏洞，发明了一种用于解密的机器叫做 [BOMBA](#)，发明了一种手工破解的方法叫做 [Zygalski sheets](#)。BOMBA 可以在两个小时之内破解 Enigma 密码。波兰人一声不吭地窃听了德国人的通信长达六年半，最后在二战爆发前夕把这技术送给了英法盟友。

BOMBA 的工作原理就是同时（并发）模拟好几个 Enigma 机器，这样可以加速猜出秘钥。最开头这样还行，但后来德国人改进了 Enigma 机器，把可选的齿轮从 3 个增加到了 5 个，使得秘钥的空间增大了 60 倍。理论上 BOMBA 只要运转 60 倍多的 Enigma 机器，就可以破解这增大的解空间，然而那已经超出了波兰的物资和人力。再加上德国人就要打过去，所以波兰只好请英法盟友帮忙。

图灵最重要的贡献，就是改进波兰人的 [BOMBA](#)，设计了一个更好的机器叫 [BOMBE](#)。BOMBE 比起 BOMBA 并没有质的飞跃，只不过 BOMBE 同时模拟的 Enigma 机器更多，转得更快。另外它加入了一些“优化”措施，尽早排除不可行的路径，所以速度快很多。图灵最初的设计，要求必须能够事先猜出很长的文本，所以基本不能用。后来 [Gordon Welchman](#) 发明了一种电路，叫做 diagonal board，才使 Bombe 能够投入实用。关于 Gordon Welchman 的故事，你可以参考这个 [BBC 纪录片](#)。

在 Bombe 能够投入使用之前，有一个叫 [John Herivel](#) 的人，发现了一种特殊的技巧，叫做 Herivel tip，这种技术在 Bombe 投入使用之前几个月就已经投入使用，破解掉很多德军的消息，立下汗马功劳。如果 Herivel tip 没有被发明，盟军可能在 1940 年 5 月就已经战败，BOMBE 也就根本没机会派上用场。

同时在 Bletchley Park，还诞生了一台大型可编程电子计算机 [Colossus](#)，它是由一个叫 [Tommy Flowers](#) 的工程师设计的。Colossus 不是用来破解 Enigma 密码的，而是用于破解 [Lorenz SZ-40](#)。那是一种比 Enigma 更先进的密码机器，用于发送希特勒的最高指令。

德国人后来又改进了他们的通信方式，使用了一种具有四个齿轮的 Enigma 机器。这大大的增加了破解的难度，普通的 Bombe 机器也破不了它了。后来是 [Harold Keen](#) 设计了一个叫做 Mammoth 的机器，后来加上美国海军的帮助，制造了更快的 Bombe，才得以破解。

所以你看到了，所有这些人的工作加起来，才改善了二战的局面。波兰人的 BOMBA，已经包含了最重要的思想。图灵的工作其实更多是量的改进，而不是质的飞跃。现在很多人喜欢跟风，片面的夸大图灵在其中的作用，这是不对

的。如果你对 Enigma 机器的技术细节感兴趣，可以参考这两个视频：[\[视频1\]](#)[\[视频2\]](#)。

理论计算机科学

图灵被称为“计算机之父”，计算机科学界的最高荣誉叫做“图灵奖”（Turing Award）。然而如果你深入的理解了计算理论和程序语言理论就会发现，图灵对于理论计算机科学，并没产生长远而有益的影响。在某种程度上说，他其实帮了一个倒忙。图灵的理论复杂不堪，给人们造成很大的误导，阻碍了计算机科学的发展。而且他对于发表论文，对待研究的态度让我怀疑，我觉得图灵本人其实就是当今计算机学术界的一些不正之风的鼻祖。

图灵机和 lambda 演算

绝大部分计算机专业的人提到图灵，就会想起图灵机（Turing Machine）。稍微有点研究的人，可能知道图灵机与 lambda 演算（lambda calculus）在计算能力上的等价性。然而在“计算能力”上等价，并不等于说它们具有同样的价值，随便用哪个都无所谓。科学研究有一条通用的原则：如果多个理论可以解释同样的现象，取最简单的一个。虽然 lambda 演算和图灵机能表达同样的理论，却比图灵机简单，优雅，实用很多。

计算理论（Theory of Computation）这个领域，其实是被图灵机给复杂化了。图灵机的设计是复杂而缺乏原则的。它的读写头，纸带，状态，操作，把本来很简单的语义搞得异常复杂。图灵机的读写两种操作同时发生，这恰好是编程上最忌讳的一种错误，类似于C语言的 `i++`。图灵机是如此的复杂和混淆，以至于你很难看出它到底要干什么，也很难用它清晰地表达自己的意思。这就是为什么每个人上“计算理论”课程，都会因为图灵机而头痛。如果你挖掘一点历史，也许会发现图灵机的原型，其实是图灵母亲使用的打字机。用一台打字机来建模所有的计算，这当然是可行的，然而却复杂不堪。

相比之下，lambda 演算更加简单，优雅，实用。它是一个非常有原则的设计。Lambda 演算不但能清晰地显示出你想要表达的意思，而且有直接的“物理实现”。你可以自然的把一个 lambda 演算表达式看成是一个电子线路模块。对于现实的编程语言设计，系统设计，lambda 演算有着巨大的指导和启发意义。以至于很多[理解 lambda 演算的人](#)都搞不明白，图灵机除了让一些理论显得高深莫测，还有什么存在的意义。

历史的倒退

图灵机比起 lambda 演算来说，其实是一个[历史](#)的倒退。1928年，Alonzo Church 发明了 lambda 演算（当时他25岁）。Lambda 演算被设计为一个通用的计算模型，并不是为了解决某个特定的问题而诞生的。1929年，Church 成为普林斯顿大学教授。1932年，Church 在 Annals of Mathematics 发表了一篇[论文](#)，纠正逻辑领域里几个常见的问题，他的论述中用了 lambda 演算。1935年，Church 发表[论文](#)，使用 lambda 演算证明基本数论中存在不可解决的问题。1936年4月，Church 发表了一篇两页纸的“[note](#)”，指出自己 1935 年那篇论文可以推论得出，著名的 Hilbert “[可判定性问题](#)”是不可解决的。

1936年5月，当时还在剑桥读硕士的图灵，也写了一篇论文，使用自己设计的一种“计算机器”（后来被叫做图灵机）来证明同一个问题。图灵的论文投稿，比 Church 最早的结论发表，晚了整整一年。编辑从来没见过图灵机这样的东西，而且它纷繁复杂，远没有 lambda 演算来得优雅。就像所有人对图灵机的第一印象一样，编辑很难相信这打字机一样的操作方式，能够容纳“所有的计算”。编辑无法确定图灵的论述是否正确，只好找人帮忙。Church 恐怕是当时世界上唯一能够验证图灵的论文正确性的人，所以一番好心之下，编辑写了封信给 Church，说：“这个叫图灵的年轻人很聪明，他写了一篇论文，使用一种机器来证明跟你一样的结果。他会把论文寄给你。如果你发现他的结果是正确的而且有用，希望你帮助他拿到奖学金，进入普林斯顿大学跟你学习。”

图灵就是这样成为了 Church 的学生，然而图灵心高气傲，恐怕从来没把 Church 当成过老师，反倒总觉得 Church 抢先一步，破坏了自己名垂青史的机会。跟 Church 的其它学生不一样，图灵没能理解 lambda 演算的精髓，却认为自己的机器才是最伟大的发明。进入 Princeton 之后，图灵不虚心请教，只是一心想发表自己的论文，想让大家对自己的“机器”产生兴趣，结果遭到很大的挫折。当然了，一个名不见经传的人，做了个怪模怪样的机器，说它可以囊括宇宙里所有的计算，不被当成民科才怪呢；）

1937年，在 Church 的帮助下，图灵的那篇[论文](#)（起名为《Computable Numbers》）终于发表了。Church 还是很器重图灵的，他把图灵的机器叫做“图灵机”。不幸的是，论文发表之后，学术界对此几乎没有任何反响，只有两人向图灵索取这篇论文。图灵当然不爽了，于是后来就到处推销自己的图灵机，想让大家承认那是伟大的发明。有了一个锤子，看什么都是钉子。后来每到一个地方，每做一个项目（见下一节），他都想把问题往自己那篇论文和图灵机上靠，东拉西扯的想证明它的价值，甚至称别人发明的东西全都是受到了图灵机的启发…… 经过人们很长的时间的以讹传讹之后，他终于成功了。

图灵当年的作法，其实跟当今计算机学术界的普遍现象差不多。我想发表自己的想法 A，结果别人已经发表了 B，解决了 A 要解决的问题，而且还比 A 简单和清晰。怎么办呢？首先，我声明自己从没看过 B 的论文，这样就可以被称为“独立的发现”。然后，我证明 A 和 B 在“本质”上是等价的。最后，我东拉西扯，挖掘一下 B 的局限性，A 相对于 B 在某些边沿领域的优势…… 这样反复折腾，寻找 A 的优势，总有一天会成功发表的。一旦发表成功，就会有人给我唱高调，没用的东西也要说成是有用的。他们会在 A 的基础上发展他们自己的东西，最后把我推崇为大师。那发表更早，更简单优美的 B，也就无人问津了。胜利！

现在不得不说一下《图灵传》对此的歪曲。Church 的论文发表，比图灵的论文投稿还早一年，而且 Church 使用了比图灵机更简单优雅的计算模型。Church 的成果本来天经地义应该受到更多的尊重，到头来作者却说：“... and

Turing was robbed of the full reward for his originality”（见第 3 节“[The Turing machine](#)”）。让人感觉貌似是 Church 用不正当的手段“抢走”了图灵的“原创性”一样。本来没有什么原创性，还丑陋复杂，所以何谈抢走呢？我怎么觉得恰恰相反，其实是图灵抢走了 Church 的原创性。现在提起 Hilbert 可判定性问题，可计算性理论，人们都想起图灵，有谁还想得起 Church，有谁知道他是第一个解决了这问题的人，有谁知道他用了更优美的办法？

Lambda 演算与计算理论

由于图灵到处推销自己的理论，把不好的东西说成是好的，把别人发明的机器硬往自己的理论上面靠，说他们受到了图灵机的“启发”，以至于很多人被蛊惑，以为它比起 lambda 演算确实有优势。再加上很多人为了自己的利益而以讹传讹，充当传教士，这就是为什么图灵机现在被人们普遍接受作为计算模型。然而这并不能改变它丑陋和混淆的本质。图灵机的设计，其实是专门为了证明 Hilbert 的可判定性问题不可解决，它并不是一个用途广泛的计算模型。图灵机之所以被人接受，很大部分原因在于人的无知。很多人（包括很多所谓“理论计算机科学家”）根本没好好理解过 lambda 演算，他们望文生义，以为图灵机是“物理的”，实际可用的“机器”，而 lambda 演算只是一个理论模型。

事实恰恰相反：lambda 演算其实非常的实用，它的本质跟电子线路没什么两样。几乎所有现实可用的程序语言，其中的语义全都可以用 lambda 演算来解释。而图灵机却没有很多现实的意义，用起来非常蹩脚，所以只能在计算理论中作为模型。另外一个更加鲜为人知的事实是：lambda 演算其实在计算理论方面也可以完全取代图灵机，它不但可以表达所有图灵机能表达的理论，而且能够更加简洁和精确地表达它们。

很多理论计算机科学家喜欢用图灵机，仿佛是因为用它作为模型，能让自己的理论显得高深莫测，晦涩难懂。普通的计算理论课本，往往用图灵机作为它的计算模型，使用苦逼的办法推导各种可计算性（computability）和复杂性（complexity）理论。特别是像 Michael Sipser 那本经典的[计算理论教材](#)，晦涩难懂，混淆不堪，有时候让我都怀疑作者自己有没有搞懂那些东西。

后来我发现，其实图灵机所能表达的理论，全都可以用更加简单的 lambda 演算（或者任何一种现在流行的程序语言）来表示。图灵机的每一个状态，不过对应了 lambda 演算（或者某种程序语言）里面的一个“AST 节点”，然而用 lambda 演算来表示那些计算理论，却可以比图灵机清晰和容易很多。在 Indiana 大学做计算理论课程助教的时候，我把这种思维方式悄悄地讲述给了上课的学生们，他们普遍表示我的这种思维方式更易理解，而且更加贴近实际的编程。

举一个很简单的例子。我可以用一行 lambda 演算表达式，来显示 Hilbert 的“可判定性问题”是无解的：

```
Halting(λm.not(Halting(m,m)), λm.not(Halting(m,m)))
```

完整的证明不到一页纸，请看我的另外一篇[文章](#)（英文）。这也就是图灵在他的[论文](#)里，折腾了十多页纸证明的东西。

我曾经以为自己是唯一知道这个秘密的人，直到有一天我把这个秘密告诉了我的博士导师 Amr Sabry。他对我说：“哈哈！其实我早就知道这个，你可以参考一下 Neil Jones 写的一本书，叫做《Computability and Complexity: From a Programming Perspective》。这本书现在已经可以[免费下载](#)。

此书作者用一种很简单的程序语言，阐述了一般人用图灵机来描述的那些理论（可计算性理论，复杂性理论）。他发现用程序语言来描述计算理论，不但简单直接，清晰明了，而且在某些方面可以更加精确地描述图灵机无法描述的定理。得到这本书，让我觉得如获至宝，原来世界上有跟我看法如此相似的人。

在一次会议上，我有幸地遇到了 Neil Jones，跟他切磋思想。当提到这本书的模型与图灵理论的关系，老教授谦虚地说：“图灵的模型还是有它的价值的……”然而到最后，他也没能说清楚这价值何在。我心里很清楚，他只是为了避免引起宗教冲突，或者避免显得狂妄自大，而委婉其词。眼前的这位教授，虽然从来没有得过图灵奖，很少有人听说过他的名字，然而他对于计算本质的理解，却比图灵本人还要高出很多。

总的说来，图灵机也不是一文不值，然而由于 lambda 演算可以更加清晰地解释图灵机能表示的所有理论，图灵机的价值相对来说几乎为零。Church 在 1937 年给图灵论文写的[Review](#) 指出，图灵机的优势，在于它可以让不懂很多数学，不理解 lambda 演算之类理论的人也可以看得懂。我怎么觉得图灵机对于不懂很多数学的人，理解起来其实更加痛苦呢？而且就算它真的对“外行”或者“笨人”的理解有好处，这价值貌似也不大吧？:P

电子计算机

很多“理论计算机科学家”喜欢说，大家现在用的计算机，只不过是一个“Universal Turing Machine”。就算你根本不知道图灵是谁，自己辛苦设计出一个机器或者语言，他们总喜欢说：“是图灵启发了你，因为你那东西是跟图灵机等价的，是图灵完备的……”

那么现在让我们来看看，图灵本人和他的理论，真正对电子计算机的发展起过多大的作用吧。如果一个人对一个行业起过重大的作用，那我们可以说“没有他不行”。然而事实却是，即使没有图灵，计算机技术会照样像今天一样发展，丝毫不受影响。看一看历史，你也许会惊讶的发现，图灵的理论不但没能启发任何计算机的设计，而且图灵亲自设计的唯一一个计算机（ACE），最后也以悲惨的失败告终。

什么是 Universal Turing Machine (UTM)

ACE 失败的一个重要原因，是因为图灵过度的看重他自己发明的 Universal Turing Machine (UTM)。所以我想首先来解密一下，这个被很多人吹得神乎其神的，似乎什么都可以往上面扯的 UTM，到底是什么东西。

说白了，UTM 就是一个解释器，就像 Python 或者 JavaScript 的解释器一样。计算机的处理器 (CPU) 也是一个解释器，它是用来解释机器指令的。那这样说来，任何可编程，具有指令集的机器都是 UTM 了，所以图灵的理论启发了所有这些机器？你尽管跟我扯吧 :)

你应该知道，在图灵的 UTM 出现以前，Church 的 lambda 演算里面早就有解释器的概念了，所以 UTM 不是什么新东西，而且它比起 lambda 演算的解释器，真是丑陋又复杂。而 Church 其实也不是第一个提出解释器这概念的人，像这类通用的概念，已经很难追溯是谁“发明”的了。也许并不是某一个人发明了它，而是历史上的很多人。

解释器这个概念的涵义实在是包罗万象，几乎无处不在。只要是“可编程”的机器，它本质上必然包含一个解释器。一个工程师在不知道解释器这概念的情况下，照样很有可能“不小心”设计出一个可编程的机器，所以如果你把这些全都归结成图灵或者 Church 的功劳，就太牵强了。

图灵与 ACE 的故事

事实上，最早的电子计算机，并不是图灵设计的，而是电子工程师跟其他一些数学家合作的结果。根据老一辈工程师的叙述，图灵的工作和理论，对于现实的电子计算机设计，几乎没有丝毫的正面作用。很多工程师其实根本不知道图灵是谁，图灵机是什么。他们只是根据实际的需求，设计和制造了那些电路。这就是为什么我们今天看到的电子计算机，跟图灵机或者图灵的其他理论几乎完全不搭边。

世界上最早的两台电子计算机，ENIAC 和 EDVAC，都是美国人设计制造的。其中 EDVAC 的设计报告，是冯诺依曼 (von Neumann) 参与并签署的。提到 EDVAC 的设计，《图灵传》有一段有趣的介绍，它基本是这样说的：“冯诺依曼在 Princeton 的时候，很了解图灵开天辟地的发明—UTM。UTM 只有一根纸带，而 EDVAC 把指令和数据放在同一个存储空间，所以 EDVAC 的设计肯定是受了 UTM 的启发。然而 EDVAC 的设计报告，却只字不提图灵和 UTM 的名字，更没有引用图灵划时代的论文《Computable Numbers》……”

这其实是在含沙射影的说，冯诺依曼和 EDVAC 团队抄袭了图灵的研究成果。照这种歪理，我洗衣服的时候，袜子和内裤放在同一个桶里洗，也是受了图灵的启发了，就因为 UTM 只有一条纸带？这世界上的事物，还有什么不是受了 UTM 启发的？这让我想起某些全靠打专利官司赚钱的公司 ([patent troll](#)) …… 冯诺依曼作为一代数学大师，比 UTM 重大的研究成果多得是了，他会在乎抄袭图灵的东西吗？其实人家恐怕是根本没把图灵和他的论文当回事。而且其他人（比如 Church）早就有跟 UTM 等价的想法，而且还更好，更简单。之前抢了 Church 的风头，现在居然欺到冯诺依曼头上了。哎，真受不了这种一辈子只想出过一个点子的人 ;)

所以听说美国人造出了 EDVAC，图灵开始各种羡慕嫉妒恨，感叹自己英才无用武之地。终于有一天，他的机会来了。在 EDVAC 诞生几个月之后，英国国家物理实验室 (NPL) 联系了图灵。他们想赶上美国的计算机技术发展，所以想招募图灵，让他帮忙山寨一个 EDVAC 的“英国特色版本”。图灵设计的机器叫做 ACE (Automatic Computing Engine)。最初，图灵给 NPL 一个很宏伟的蓝图：ACE 可以如此的强大，以至于整个英国只需要这样一台计算机就够了，我们可以把它叫做“英国国家计算机”…… 然而再大的口号，也难逃脱现实的检验，ACE 项目最终以失败告终。

《图灵传》把 ACE 失败的责任，推托到 NPL 和其它人的“近视”和“官僚”，然而 ACE 失败的主要责任，其实在于图灵自己：他没有设计一台现实的计算机的基本技能，却设立高大空的目标。图灵的设计跟当时（包括现在的）所有实用的计算机都有巨大的差别。不出你所料，他最初的设计思路，是根据自己的 UTM，不过从中去掉了一些不实际的设计，比如用一根纸带来存储数据。这一点改进貌似做对了，可是呢，他又加入了一些让工程师们无语的设计，美其名曰“极简设计” (minimalism)。比如，ACE 的硬件只提供 AND, OR, NOT 之类的逻辑运算作为“基本操作”，其它的算数操作，包括加减乘除，全部用代码来实现。图灵大师啊，你知不知道有一种重要的指标，叫做“效率”？

这还不算…… 后来他更加异想天开，终于扯上了“思考机器” (thinking machines) —他想让 ACE 成为可以像人一样思考的机器，还想让这机器能够自己写自己的代码。按照图灵的原话：“在 ACE 的工作中，我对人脑建模的兴趣，比实际的计算应用更感兴趣。” 他显然已经把 ACE 当成了自己一个人的玩具，而不再是解决人们实际需求的工具。只要有人反对这想法，他就会嘲笑说，你是怕我的机器太聪明了，抢了你的饭碗吧？其实图灵对于实际的人脑工作原理所知甚少，基本处于初中生理卫生课本水平，然而他总喜欢对人说，人脑不过就是一个 UTM。看吧，它有输入，输出，状态转换，就跟 UTM 一样…… 所谓“图灵测试” (Turing Test)，就是那时候提出来的。当然了，因为他扯到了“thinking machine”，就有后人把他称为人工智能 (AI) 的鼻祖。其实呢，图灵测试根本就不能说明一个机器具有了人的智能，它只是在测试一些肤浅的表象。后来，“[thinking machines](#)”成为了一种通用的幌子，用于筹集大笔科研经费，最后全都血本无归。

图灵设计了这机器，NPL 当时却没有能力制造它。于是他们求助于另外两位实现过计算机的工程师：[F. C. Williams](#) 和 [Maurice Wilkes](#)（后来 EDSAC 计算机的设计者），请他们帮忙实现图灵的设计。可想而知，Williams 和 Wilkes 都表示不喜欢 ACE 的设计，而且指出图灵的性格与自己的研究风格不匹配，不愿跟他合作，所以双双拒绝了 NPL 的邀请。最后，NPL 新成立了一个电子部门，ACE 的工程终于可以开始。然而，根据资深工程师们的讨论，觉得图灵提出的制造一个“电子人脑”和“智能机器”，并不是实际可行，或者在短期之内能派上用场的项目，所以决定做一些实际点的事情。图灵对此非常恼火，各种抱怨，说别人官僚啊，近视啊，没想象力啊之类的，然后开始公开的抵制 NPL 的决定。

最后工程师们和管理层都受不了他了，鉴于他名声在外，又不好意思开掉他，只好提出一个破天荒的提议：由 NPL

资助，让图灵回到剑桥大学去度年假（sabbatical），做一些纯数学的研究。于是 ACE 在图灵不在的情况下，终于可以开工了…… 1950 年，ACE 运行了它的第一个程序。然而工程师们实现的 ACE，完全偏离了图灵的设计，以至于实际的机器和图灵的设计之间，几乎没有任何相似性。一年之后，图灵还想回到 NPL，继续影响 ACE 的设计，然而 NPL 的领导们却建议他继续留在大学里做纯理论的研究，并且让曼彻斯特大学给他一个职位。最后图灵接受了这个建议，这下大家伙儿都松了一口气…… :P

图灵设计的唯一一个计算机 ACE，终究以图灵完全退出整个项目而告终。今天回头看来，如果当时图灵留下来了，NPL 真的按照图灵的意思来做，ACE 恐怕直到今天都造不出来。由于图灵不切实际的设计和高傲的性格，NPL 失去了最优秀的人的帮助。1949 年，Maurice Wilkes 按照 EDVAC 的思路，成功制造了 [EDSAC](#)，速度是 ACE 的两倍以上，而且更加实用。

如果对 ACE 和其它早期计算机感兴趣，你可以参考一下更详细的[资料](#)。你也可以看一看《图灵传》，虽然它观点荒唐，对图灵各种偏袒，然而图灵和其他人的通信，基本的史实，他应该不好意思篡改。

总结

我说这些是为了什么呢？我当然不是想否认图灵所做出的贡献。像许多的计算机工作者一样，他的工作当然是有意义的。然而那种意义并不像很多人所吹嘘的那么伟大，它们甚至不包含很多的创新。

我觉得很多后人给图灵带上的光环，掩盖了太多其它值得我们学习和尊敬的人，给人们对于计算机科学的概念造成了误导。计算机科学不是图灵一个人造出来的，图灵并不是计算机科学的鼻祖，他甚至不是在破解 Enigma 密码和电子计算机诞生过程中起最重要作用的人。

许许多多的计算机科学家和电子工程师们，是他们造就了今天的计算科学。他们的聪明才智和贡献，不应该被图灵的光环所掩盖，他们应该受到像跟图灵一样的尊敬。希望大家不要再神化图灵，不要再神化任何人。不要因为膜拜某些人，而失去向另一些人学习的机会。

对 Parser 的误解

一直很了解人们对于 parser 的误解，可是一直都提不起兴趣来阐述对它的观点。然而我觉得是有必要解释一下这个问题的时候了。我感觉得到大部分人对于 parser 的误解之深，再不澄清一下，恐怕这些谬误就要写进歪曲的历史教科书，到时候就没有人知道真相了。

什么是 Parser

首先来科普一下。所谓 parser，一般是指把某种格式的文本（字符串）转换成某种数据结构的过程。最常见的 parser，是把程序文本转换成编译器内部的一种叫做“抽象语法树”（AST）的数据结构。也有简单一些的 parser，用于处理 CSV, JSON, XML 之类的格式。

举个例子，一个处理算数表达式的 parser，可以把“ $1+2$ ”这样的，含有 1 , $+$, 2 三个字符的字符串，转换成一个对象（object）。这个对象就像 `new BinaryExpression(ADD, new Number(1), new Number(2))` 这样的 Java 构造函数调用生成出来的那样。

之所以需要做这种从字符串到数据结构的转换，是因为编译器是无法直接操作“ $1+2$ ”这样的字符串的。实际上，代码的本质根本就不是字符串，它本来就是一个具有复杂拓扑的数据结构，就像电路一样。“ $1+2$ ”这个字符串只是对这种数据结构的一种“编码”，就像 ZIP 或者 JPEG 只是对它们压缩的数据的编码一样。

这种编码可以方便你把代码存到磁盘上，方便你用文本编辑器来修改它们，然而你必须知道，文本并不是代码本身。所以从磁盘读取了文本之后，你必须先“解码”，才能方便地操作代码的数据结构。比如，如果上面的 Java 代码生成的 AST 节点叫 `node`，你就可以用 `node.operator` 来访问 `ADD`，用 `node.left` 来访问 `1`, `node.right` 来访问 `2`。这是很方便的。

对于程序语言，这种解码的动作就叫做 parsing，用于解码的那段代码就叫做 parser。

Parser 在编译器中的地位

那么貌似这样说来，parser 是编译器里面很关键的一个部分了？显然，parser 是必不可少的，然而它并不像很多人想象的那么重要。Parser 的重要性和技术难度，被很多人严重的夸大了。一些人提到“编译器”，就跟你提 LEX, YACC, ANTLR 等用于构造 parser 的工具，仿佛编译器跟 parser 是等价的似的。还有些人，只要听说别人写了个 parser，就觉得这人编程水平很高，开始膜拜了。这些都是肤浅的表现。

我喜欢把 parser 称为“万里长征的第一步”，因为等你 parse 完毕得到了 AST，真正的编译技术才算开始。一个编译器包含许多的步骤：语义分析，类型检查/推导，代码优化，机器代码生成，……。这每个步骤都是在对某种中间数据结构（比如 AST）进行分析或者转化，它们完全不需要知道代码的字符串形式。也就是说，一旦代码通过了 parser，在后面的编译过程里，你就可以完全忘记 parser 的存在。所以 parser 对于编译器的地位，就像 ZIP 之于 JVM，就像 JPEG 之于 PhotoShop。Parser 虽然必不可少，然而它比起编译器里面最重要的过程，是处于一种辅助性的地位。

鉴于这个原因，好一点的大学里的程序语言（PL）课程，都完全没有关于 parser 的内容。学生们往往直接用 Scheme 这样代码数据同形的语言，或者直接使用 AST 数据结构来构造程序。在 Kent Dybvig 这样编译器大师的课程上，学生直接跳过 parser 的构造，开始学习最精华的语义转换和优化技术。实际上，Kent Dybvig 根本不认为 parser 算是编译器的一部分。因为 AST 数据结构才是程序本身，而程序的文本只是这种数据结构的一种编码形式。

Parser 技术发展的误区

既然 parser 在编译器中处于次要的地位，可是为什么还有人花那么大功夫研究各种炫酷的 parser 技术呢。LL, LR, GLR, LEX, YACC, Bison, parser combinator, ANTLR, PEG, …… 制造 parser 的工具似乎层出不穷，每出现一个新的工具都号称可以处理更加复杂的语法。

很多人盲目地设计复杂的语法，然后用越来越复杂的 parser 技术去 parse 它们，这就是 parser 技术仍然在发展的原因。向往复杂的语法，是程序语言领域流传非常广，危害非常大的错误倾向。在人类历史的长河中，留下了许多难以磨灭的历史性糟粕，它们固化了人类对于语言设计的理念。很多人设计语言似乎不是为了拿来好用的，而是为了让用它的人迷惑或者害怕。

有些人假定了数学是美好的语言，所以他们盲目的希望程序语言看起来更加像数学。于是他们模仿数学，制造了各种奇怪的操作符，制定它们的优先级，这样你就可以写出 $2 << 7 - 2 * 3$ 这样的代码，而不需要给子表达式加上括号。还有很多人喜欢让语法变得“简练”，就为了少打几个括号，分号，花括号，……。可是由此带来的结果是复杂，不一致，有多义性，难扩展的语法，以及障眼难读，模棱两可的代码。

更有甚者，对数学的愚蠢做法执迷不悟的人，设计了像 Haskell 和 Coq 那样的语言。在 Haskell 里面，你可以在代码里定义新的操作符，指定它的“结合律”（associativity）和“优先级”（precedence）。这样的语法设计，要求 parser 必须能够在 parse 过程中读入并且加入新的 parse 规则。Coq 试图更加“强大”一些，它让你可以定

义“mixfix 操作符”，也就是说你的操作符可以连接超过两个表达式。这样你就可以定义像 `if...then...else...` 这样的“操作符”。

制造这样复杂难懂的语法，没有什么真正的好处。不但给程序员的学习造成了不必要的困难，让代码难以理解，而且也给 parser 的作者带来了严重的挑战。可是有些人就是喜欢制造问题，就像一句玩笑话说的：有困难要上，没有困难，制造困难也要上！

如果你的语言语法很简单（像 Scheme 那样），你是不需要任何高深的 parser 理论的。说白了，你只需要知道如何 parse 匹配的括号。最多一个小时，几百行 Java 代码，我就能写出一个类似 Scheme 语言的 parser。

可是很多人总是嫌问题不够有难度，于是他们不停地制造更加复杂的语法，甚至会故意让自己的语言看起来跟其它的不一样，以示“创新”。当然了，这样的语言就得用更加复杂的 parser 技术，这正好让那些喜欢折腾复杂 parser 技术的人洋洋得意。

编译原理课程的误导

程序员们对于 parser 的误解，很大程度上来自于大学编译原理课程照本宣科的教育。很多老师自己都不理解编译器的精髓，所以就只有按部就班的讲一些“死知识”，灌输“业界做法”。一般大学里上编译原理课，都是捧着一本大部头的“[龙书](#)”或者“[虎书](#)”，花掉一个学期 1/3 甚至 2/3 的时间来学写 parser。由于 parser 占据了大量时间，以至于很多真正精华的内容都被一笔带过：语义分析，代码优化，类型推导，静态检查，机器代码生成，…… 以至于很多人上完了编译原理课程，记忆中只留下写 parser 的痛苦回忆。

“龙书”之类的教材在很多人心目中地位是如此之高，被誉为“经典”，然而除了开头很大篇幅来讲 parser 理论，这本书其它部分的水准一般般。大部分学生的反映是“看不懂”，然而由于一直以来没有更好的选择，它经典的地位是难以动摇。“龙书”后来的新版我浏览过一下，新加入了类型检查/推导的部分，可是我看得出来，作者们自己对于类型理论都是一知半解，所以也就没法写清楚，关键部分几乎是一笔带过。

虎书作者 Appel 水平稍高些，但还是免不了 parser 这个坑。我在 Cornell 的时候上过一门用虎书做教材的编译器课，也是痛苦不堪，一个月都在折腾 parser，我不知道自己为什么要做那些无聊的事情。再加上极其不合理的评分制度，导致我最后不得不退掉这门课程。后来跟虎书作者一个学生合作搞研究，也有一段奇葩的经历，扯远了 ;)

所以我从来就不认为自己是“编译器”专业的，我认为自己是“PL 专业”。编译器领域照本宣科成分更多一些，PL 专业更加注重本质的东西。

如果你想真的深入理解编译理论，最好是从 PL 课程的读物，比如 [EOPL](#) 开始。我可以说 PL 这个领域，真的和编译器的领域很不一样。请不要指望编译器的作者（比如 LLVM 的作者）能够设计出好的语言，因为他们可能根本不理解很多语言设计的东西，他们只是会实现某些别人设计的语言。可是反过来，理解了 PL 的理论，编译器的东西只不过是把一种语言转换成另外一种语言（机器语言）而已。工程的细枝末节很麻烦，可是当你掌握了精髓的原理，那些都容易摸索出来。

我写 parser 的心得和秘诀

虽然我已经告诉你，给过度复杂的语言写 parser 是很苦逼，没有意思的工作，然而有些历史性的错误已经造成了深远的影响，所以很多时候虽然心知肚明，你也不得不妥协一下。由于像 C++，Java，JavaScript，Python 之类语言的流行，有时候你是被迫要给它们写 parser。在这一节，我告诉你一些秘诀，也许可以帮助你更加容易的写出这些语言的 parser。

很多人都觉得写 parser 很难，一方面是由于语言设计的错误思想导致了复杂的语法，另外一方面是由于人们对于 parser 构造过程的思维误区。很多人不理解 parser 的本质和真正的用途，所以他们总是试图让 parser 干一些它们本来不应该干的事情，或者对 parser 有一些不切实际的标准。当然，他们就会觉得 parser 非常难写，非常容易出错。

1. 尽量拿别人写的 parser 来用。维护一个 parser 是相当繁琐耗时，回报很低的事情。一旦语言有所改动，你的 parser 就得跟着改。所以如果你能找到免费的 parser，那就最好不要自己写。现在的趋势是越来越多的语言在标准库里提供可以 parse 它自己的 parser，比如 Python 和 Ruby。这样你就可以用那语言写一小段代码调用标准的 parser，然后把它转换成一种常用的数据交换格式，比如 JSON。然后你就可以用通用的 JSON parser 解析出你想要的数据结构了。

如果你直接使用别人的 parser，最好不要使用它原来的数据结构。因为一旦 parser 的作者在新版本改变了他的数据结构，你所有的代码都会需要修改。我的秘诀是做一个“AST 转换器”，先把别人的 AST 结构转换成自己的 AST 结构，然后在自己的 AST 结构之上写其它的代码，这样如果别人的 parser 修改了，你可以只改动 AST 转换器，其它的代码基本不需要修改。

用别人的 parser 也会有一些小麻烦。比如 Python 之类语言自带的 parser，丢掉了很多我需要的信息，比如函数名的位置，等等。我需要进行一些 hack，找回我需要的数据。相对来说，这样小的修补还是比从头写一个 parser 要划得来。但是如果你实在找不到一个好的 parser，那就只好自己写一个。

2. 很多人写 parser，很在乎所谓的“one-pass parser”。他们试图扫描一遍代码文本就构造出最终的 AST 结

构。可是如果你放松这个条件，允许用多 pass 的 parser，就会容易很多。你可以在第一遍用很容易的办法构造一个粗略的树结构，然后再写一个递归树遍历过程，把某些在第一遍的时候没法确定的结构进行小规模的转换，最后得到正确的 AST。

想要一遍就 parse 出最终的 AST，可以说是一种过早优化（premature optimization）。有些人盲目地认为只扫描一遍代码，会比扫描两遍要快一些。然而由于你必须在这一遍扫描里进行多度复杂的操作，最终的性能也许还不如很快的扫完第一遍，然后再很快的遍历转换由此生成的树结构。

3. 另外一些人试图在 parse 的过程中做一些本来不属于 parser 职责的事情，比如进行一些基本的语义检查。有些人会让 parser 检查“使用未定义的变量”等语义错误，一旦发现就在当时报错，终止。这种做法混淆了 parser 的作用，造成了不必要的复杂性。

就像我说的，parser 只是一个解码器。parser 要做的事情，应该是从无结构的字符串里面，解码产生有结构的数据结构。而像“使用未定义的变量”这样的语义检查，应该是在生成了 AST 之后，使用单独的树遍历来进行的。人们常常混淆“解码”，“语法”和“语义”三者的不同，导致他们写出过度复杂，效率低下，难以维护的 parser。

4. 另一种常见的误区是盲目的相信 YACC，ANTLR 之类所谓“parser generator”。实际上 parser generator 的概念看起来虽然美好，可是实际用起来几乎全都是噩梦。事实上最好的 parser，比如 EDG C++ parser，几乎全都是直接用普通的程序语言手写而成的，而不是自动生成的。

这是因为 parser generator 都要求你使用某种特殊的描述语言来表示出语法，然后自动把它们转换成 parser 的程序代码。在这个转换过程中，这种特殊的描述语言和生成的 parser 代码之间，并没有很强的语义连接关系。如果生成的 parser 有 bug，你很难从生成的 parser 代码回溯到语法描述，找到错误的位置和原因。你没法对语法描述进行 debug，因为它只是一个文本文件，根本不能运行。

所以如果你真的要写 parser，我建议你直接用某种程序语言手写代码，使用普通的递归下降（recursive descent）写法，或者 parser combinator 的写法。只有手写的 parser 才可以方便的 debug，而且可以输出清晰，人类可理解的出错信息。

5. 有些人喜欢死扣 BNF 范式，盲目的相信“LL”，“LR”等语法的区别，所以他们经常落入误区，说“哎呀，这个语法不是 LL 的”，于是采用一些像 YACC 那样的 LR parser generator，结果落入非常大的麻烦。虽然有些语法看起来不是 LL 的，它们的 parser 却仍然可以用普通的 recursive descent 的方式来写。

这里的秘诀在于，语言规范里给出的 BNF 范式，并不是唯一的可以写出 parser 的做法。BNF 只是一个基本的参照物，它让你可以对语法有个清晰的概念，可是实际的 parser 却不一定非得按照 BNF 的格式来写。有时候你可以把语法的格式稍微改一改，变通一下，却照样可以正确地 parse 原来的语言。由于很多语言的语法都类似于 C，所以很多时候你写 parser 只需要看一些样例程序，然后根据自己的经验来写，而不需要依据 BNF。

Recursive descent 和 parser combinator 写出来的 parser 可以非常强大，甚至可以超越所谓“上下文无关文法”，因为在递归函数里面你可以做几乎任意的事情，所以你甚至可以把上下文传递到递归函数里，然后根据上下文来决定对当前的节点做什么事情。而且由于代码可以得到很多的上下文信息，如果输入的代码有语法错误，你可以根据这些信息生成非常人性化的出错信息。

总结

所以你看到了，parser 并不是编译器，它甚至不属于编译里很重要的东西。程序语言和编译器里面有比 parser 重要很多，有趣很多的东西。Parser 的研究其实是在解决一些根本不存在或者人为制造的问题。复杂的语法导致了复杂的 parser 技术，它们仍然在给计算机世界带来不必要的困扰和麻烦。对 parser 写法的很多误解，过度工程和过早优化，造成了很多人错误的高估写 parser 的难度。

能写 parser 并不是什么了不起的事情，它是个苦差事。所以如果你会写 parser，请不要以为是什么了不起的事情，如果你看到有人写了某种语言的 parser，也不要表现出让人哭笑不得的膜拜之情。

数学和编程

好些人来信问我，要成为一个好的程序员，数学基础要达到什么样的程度？十八年前，当我成为大学计算机系新生的时候，也为同样的问题所困扰。面对学数学，物理等学科的同学，我感到自卑。经常有人说那些专业的知识更加精华一些，难度更高一些，那些专业的人毕业之后如果做编程工作，水平其实比计算机系毕业的还要高。直到深入研究程序语言之后，对这个问题我才得到了答案和解脱。由于好多编程新手遇到同样的困扰，所以我想在这里把这个问题详细的阐述一下。

数学并不是计算机科学的基础

很多人都盲目的认为，计算机科学是数学的一个分支，数学是计算机科学的基础，数学是更加博大精深的科学。这些人以为只要学会了数学，编程的事情全都不在话下，然而事实却并非如此。

事实其实是这样的：

- 计算机科学根本不是数学，它只不过借用了非常少，非常基础的数学，比高中数学还要容易。
- 所谓“高等数学”，并不是研究计算机科学必须的。你可以用计算机来做微积分计算，可是这时候你其实是在做数学工作，用计算机作为工具。你研究的并不是计算机科学。这就像你可以用计算机来设计建筑，但建筑学却不是计算机科学的基础。
- 计算机是比数学更加基础的工具，就像纸和笔一样。计算机可以用来解决数学的问题，也可以用来解决不是数学的问题，比如工程的问题，艺术的问题，经济的问题，社会的问题等等。
- 计算机科学是完全独立的学科。学习了数学和物理，并不能代替对计算机科学的学习。你必须针对计算机科学进行学习，才有可能成为好的程序员。
- 数学家所用的语言，比起常见的程序语言（比如C++，Java）来说，其实是非常落后而蹩脚的设计。所谓“数学的美感”，其实大部分是夜郎自大。
- 99% 的数学家都写不出像样的代码。

数学是异常糟糕的语言

这并不是危言耸听。如果你深入研究过程序语言的理论，就会发现其实数学家们使用的那些符号，其实是一种非常糟糕的程序语言。数学的理论很多是有用的，然而数学家们用于描述这些理论所用的语言，却是纷繁复杂，缺乏一致性，可组合性（composability），简单性，可用性。这也就是为什么大部分人看到数学就头痛。这不是他们不够聪明，而是数学语言的“[设计](#)”有问题。人们学习数学的时候，其实只有少部分时间在思考它的精髓，而大部分时间是在折腾它的语法。

举一个非常简单的例子。如果说 $\cos^2\theta$ 表示 $(\cos \theta)^2$ ，那么理所当然， $\cos^{-1}\theta$ 就应该表示 $1/(\cos \theta)$ 了？可它偏偏不是！别被数学老师的教条和借口欺骗啦，他们总是告诉你：“你应该记住这些！”可是你想过吗：凭什么？ $\cos^2\theta$ 表示 $(\cos \theta)^2$ ，而 $\cos^{-1}\theta$ ，明明是一模一样的形式，表示的却是 $\arccos \theta$ 。一个是求幂，一个是调用反函数，风马不及，却写成一个样子。这样的语言设计混淆不堪，却喜欢以“约定俗成”作为借口。

如果你再多看一些数学书，就会发现这只是数学语言几百年累积下来的糟粕的冰山一角。数学书里尽是各种上标下标，带括号的上标下标， $x, y, z, a, b, c, f, g, h$ ，各种扭来扭去的希腊字母，希伯来字母……斜体，黑体，花体，双影体，……用不同的字体来表示不同的“类型”。很多符号的含义，在不同的子领域里面都不一样。有些人上一门数学课，到最后还没明白那些符号是什么意思。

直到今天，数学家们写书仍然非常不严谨。他们常犯的一个错误是把 x^2 这样的东西叫做“函数”（function）。其实 x^2 不是一个函数，它只是一个表达式。你必须同时指明“x 是参数”，加上 x^2 ，才会成为一个函数。所以正确的函数写法其实看起来像这样： $f(x) = x^2$ 。或者如果你不想给它一个名字，可以借用 lambda calculus 的写法，写成： $\lambda x.x^2$ 。

可是数学家们非常的喜欢“约定俗成”。他们定了一些不成文的规矩是这样：凡是叫“x”的，都是函数的参数，凡是叫“y”的，都可能是一个函数……所以你写 x^2 就可以表示 $\lambda x.x^2$ ，而不需要显式的写出“ λx ”。殊不知这些约定俗成，看起来貌似可以让你少写几个字，却造成了许许多多的混淆和麻烦。比如，你在 Mathematica 里面可以对 [\$x^2 + y\$](#) 求关于x的导数，而且会得到 $y'(x) + 2x$ 这样蹊跷的结果，因为它认为 y 可能是一个函数。更奇怪的是，如果你在后面多加一个 a ，也就是对 [\$x^2 + y + a\$](#) 求导，你会得到 $2x$ ！那么 $y'(x)$ 到哪里去了？莫名其妙……

相对而言，程序语言就严谨很多，所有的程序语言都要求你必须指出函数的参数叫什么名字。像 x^2 这样的东西，在程序语言里面不是一个函数（function），而只是一个表达式（expression）。即使 JavaScript 这样毛病众多的语言都是这样。比如，你必须写：

```
function (x) { return x * x }
```

那个括号里的(x)，显式的声明了变量的名字，避免了可能出现的混淆。我不是第一个指出这些问题的人。其实现代逻

辑学的鼻祖 Gottlob Frege 在一百多年以前就在他的论文“[Function and Concept](#)”里批评了数学家们的这种做法。可是数学界的表达方式直到今天还是一样的混乱。

很多人学习微积分都觉得困难，其实问题不在他们，而在于莱布尼兹 (Leibniz)。莱布尼兹设计来描述微积分的语言 (\int , dx , dy , ...)，从现代语言设计的角度来看，其实非常之糟糕，可以说是一塌糊涂。我不能怪莱布尼兹，他毕竟是几百年前的人了，他不知道我们现在知道的很多东西。然而古人的设计，现在还不考虑改进，反而当成教条灌输给学生，那就是不思进取了。

数学的语言不像程序语言，它的历史太久，没有经过系统的，考虑周全的，统一的设计。各种数学符号的出现，往往是历史上某个数学家有天在黑板上随手画出一些古怪的符号，说这代表什么，那代表什么，……然后就定下来了。很多数学家只关心自己那块狭窄的子领域，为自己的理论随便设计出一套符号，完全不管这些是否跟其它子领域的符号相冲突。这就是为什么不同的数学子领域里写出同样的符号，却可以表示完全不同的涵义。在这种意义上，数学的语言跟 Perl (一种非常糟糕的程序语言) 有些类似。Perl 把各种人需要的各种功能，不加选择地加进了语言里面，造成语言繁复不堪，甚至连 Perl 的创造者自己都不能理解它所有的功能。

数学的证明，使用的其实也是极其不严格的语言——古怪的符号，加上含糊不清，容易误解的人类语言。如果你知道什么是 [Curry-Howard Correspondence](#) 就会明白，其实每一个数学证明都不过是一段代码。同样的定理，可以有许多不同版本的证明（代码）。这些证明有的简短优雅，有的却冗长繁复，像面条一样绕来绕去，没法看懂。你经常在数学证明里面看到“未定义的变量”，证明的逻辑也包含着各种隐含知识，思维跳跃，非常难以理解。很多数学证明，从程序的观点来看，连编译都不会通过，就别提运行了。

数学家们往往不在乎证明的优雅性。他们认为只要能证明出定理，你管我的证明简不简单，容不容易看懂呢。你越是看不懂，就越是觉得我高深莫测，越是感觉你自己笨！这种思潮到了编程的时候就显出弊端了。数学家写代码，往往忽视代码的优雅性，简单性，模块化，可读性，性能，数据结构等重要因素，认为代码只要能算出结果就行。他们把代码当成跟证明一样，一次性的东西，所以他们的代码往往不能满足实际工程的严格要求。

数学里最在乎语言设计的分支，莫过于逻辑学了。很多人（包括很多程序语言专家）都盲目的崇拜逻辑学家，盲目的相信数理逻辑是优雅美好的语言。在程序语言界，数理逻辑已经成为一种灾害，明明很容易就能解释清楚的语义，非得写成一堆稀奇古怪，含义混淆的逻辑公式。殊不知其实数理逻辑也是有很大的历史遗留问题和误区的。研究逻辑学的人经常遇到各种“不可判定” (undecidable) 问题和所谓“悖论” (paradox)，研究几十年也没搞清楚，而其实那些问题都是他们自己造出来的。你只需要把语言改一下，去掉一些不必要的功能，问题就没了。但逻辑学家们总喜欢跟你说，那是某天才老祖宗想出来的，多么多么的了不起啊，不能改！

用一阶逻辑 (first-order logic) 这样的东西，你可以写出一些毫无意义的语句。逻辑老师们会告诉你，记住啦，这些是没有意义的，如果写出来这些东西，是你的问题！他们没有意识到，如果一个人可以用一个语言写出毫无意义的东西，那么这问题在于这个语言，而不在于这个人。一阶逻辑号称可以“表达所有数学”，结果事实却是，没有几个数学家真的可以用它表达很有用的知识。到后来，稍微明智一点的逻辑学家们开始研究这些老古董语言到底出了什么毛病，于是他们创造了 Model Theory 这样的理论。写出一些长篇大部头，用于“验证”这些逻辑语言的合理性。这些问题在我看来都是显而易见的，因为很多逻辑的语言根本就不是很好很有用的东西。去研究它们“为什么有毛病”，其实是白费力气。自己另外设计一个更好语言就完事了。

在我看来，除了现代逻辑学的鼻祖 [Gottlob Frege](#) 理解了逻辑的精髓，其它逻辑学家基本都是照本宣科，一知半解。他们喜欢把简单的问题搞复杂，制造一些新名词，说得玄乎其玄灵丹妙药似的。如果你想了解逻辑学的精华，建议你看看 [Frege 的文集](#)。看了之后你也许会发现，Frege 思想的精华，其实已经融入在几乎所有的程序语言里了。

编程是一门艺术

从上面你也许已经明白了，普通程序员使用的编程语言，就算是 C++ 这样毛病众多的语言，其实也已经比数学家使用的语言好很多。用数学的语言可以写出含糊复杂的证明，在期刊或者学术会议上蒙混过关，用程序语言写出来的代码却无法混过计算机这道严格的关卡。因为计算机不是人，它不会迷迷糊糊的点点头让你混过去，或者因为你是大师就不懂装懂。代码是需要经过现实的检验的。如果你的代码有问题，它迟早会导致出问题。

计算机科学并不是数学的一个分支，它在很大程度上是优于数学，高于数学的。有些数学的基本理论可以被计算机科学所用，然而计算机科学并不是数学的一部分。数学在语言方面带有太多的历史遗留糟粕，它其实是泥菩萨过河，自身难保，它根本解决不了编程中遇到的实际问题。

编程真的是一门艺术，因为它符合艺术的各种特征。艺术可以利用科学提供的工具，然而它却不是科学的一部分，它的地位也并不低于科学。和所有的艺术一样，编程能解决科学没法解决的问题，满足人们新的需求，开拓新的世界。所以亲爱的程序员们，别再为自己不懂很多数学而烦恼了。数学并不能帮助你写出好的程序，然而能写出好程序的人，却能更好的理解数学。我建议你们先学编程，再去看数学。

如果你想了解更多关于数学语言的弊病以及程序语言对它们的改进，我建议你看看这个 Gerald Susman 的[讲座](#)。

谈程序的正确性

不管在学术圈还是在工业界，总有很多人过度的关心所谓“程序的正确性”，有些甚至到了战战兢兢，舍本逐末的地步。下面举几个例子：

- 很多人把测试（test）看得过于重要。代码八字还没一撇呢，就吵着要怎么怎么严格的测试，防止“将来”有人把代码改错了。这些人到后来往往被测试捆住了手脚，寸步难行。不但代码bug百出，连测试里面也很多bug。
- 有些人对于“使用什么语言”这个问题过度的在乎，仿佛只有用最新最酷，功能最多的语言，他们才能完成一些很基本的任务。这种人一次又一次的视一些新语言为“灵丹妙药”，然后一次又一次的幻灭，最后他们什么有用的代码也没写出来。
- 有些人过度的重视所谓“类型安全”（type safety），经常抱怨手头的语言缺少一些炫酷的类型系统功能，甚至因此说没法写代码了！他们没有看到，即使缺少一些由编译器静态保障的类型安全，代码其实一点问题都没有，而且也许更加简单。
- 有些人走上极端，认为所有的代码都必须使用所谓“形式化方法”（formal methods），用机器定理证明的方式来确保它100%的没有错误。这种人对于证明玩具大小的代码乐此不疲，结果一辈子也没写出过能解决实际问题的代码。

100%可靠的代码，这是多么完美的理想！可是到最后你发现，天天念叨着要“正确性”，“可靠性”的人，几乎总是眼高手低，说的比做的多。自己没写出什么解决实际问题的代码，倒是很喜欢对别人的“代码质量”评头论足。这些人自己的代码往往复杂不堪，喜欢使用各种看似高深的奇技淫巧，用以保证所谓“正确”。他们的代码被很多所谓“测试工具”和“类型系统”捆住手脚，却仍然bug百出。到后来你逐渐发现，对“正确性”的战战兢兢，其实是这些人不解决手头问题的借口。

衡量程序最重要的标准

这些人其实不明白一个重要的道理：你得先写出程序，才能开始谈它的正确性。看一个程序好不好，最重要的标准，是看它能否有效地解决问题，而不是它是否正确。如果你的程序没有解决问题，或者解决了错误的问题，或者虽然解决问题但却非常难用，那么这程序再怎么正确，再怎么可靠，都不是好的程序。

正确不等于简单，不等于优雅，不等于高效。一个不简单，不优雅，效率低的程序，就算你费尽周折证明了它的正确，它仍然不会很好的工作。这就像你得先有了房子，才能开始要求房子是安全的。想想吧，如果一个没有房子的流浪汉，路过一座没有人住的房子，他会因为这房子“不是100%安全”，而继续在野外风餐露宿吗？写出代码就像有了房子，而代码的正确性，就像房子的安全性。写出可以解决问题的程序，永远是第一位的。而这个程序的正确性，不管它如何的重要，永远是第二位的。对程序的正确性的强调，永远不应该高于写出程序本身。

每当谈起这个问题，我就喜欢打一个比方：如果“黎曼猜想”被王垠证明出来了，它会改名叫“王垠定理”吗？当然不会。它会被叫做“黎曼定理”！这是因为，无论一个人多么聪明多么厉害，就算他能够证明出黎曼猜想，但这个猜想并不是他最先想出来的。如果黎曼没有提出这个猜想，你根本不会想到它，又何谈证明呢？所以我喜欢说，一流的数学家提出猜想，二流的数学家证明别人的猜想。同样的道理，写出解决问题的代码的人，比起那些去证明（测试）他的代码正确性的人，永远是更重要的。因为如果他没写出这段代码，你连要证明（测试）什么都不知道！

如何提高程序的正确性

话说回来，虽然程序的正确性相对于解决问题，处于相对次要的地位，然而它确实是不可忽视的。但这并不等于天天鼓吹要“测试”，要“形式化证明”，就可以提高程序的正确性。

如果你深入研究过程序的逻辑推导就会知道，测试和形式化证明的能力都是非常有限的。测试只能测试到最常用的情况，而无法覆盖所有的情况。别被所谓“测试覆盖”（test coverage）给欺骗了。一行代码被测试覆盖而没有出错，并不等于在那里不会出错。一行代码是否出错，取决于在它运行之前所经过的所有条件。这些条件的数量是组合爆炸关系，基本上没有测试能够覆盖所有这些前提条件。

形式化方法对于非常简单直接的程序是有效的，然而一旦程序稍微大点，形式化方法就寸步难行。你也许没有想到，你可以用非常少的代码，写出[Collatz Conjecture](#)这样至今没人证明出来的数学猜想。实际使用中的代码，比这种数学猜想要复杂不知道多少倍。你要用形式化方法去证明所有的代码，基本上等于你永远也没法完成项目。

那么提高程序正确性最有效的方法是什么呢？在我看来，最有效的方法莫过于对代码反复琢磨推敲，让它变得简单，直观，直到你一眼就可以看得出它不可能有问题。

DRY原则的误区

很多编程的人，喜欢鼓吹各种各样的“原则”，比如KISS原则，DRY原则…… 总有人把这些所谓原则奉为教条或者秘方，以为兢兢业业地遵循这些，空喊几个口号，就可以写出好的代码。同时，他们对违反这些原则的人嗤之以鼻——你不知道，不遵循或者藐视这些原则，那么你就是菜鸟。所谓“[DRY原则](#)”（Don't Repeat Yourself，不要重复你自己）就是这些教条其中之一。盲目的迷信DRY原则，在实际的工程中带来了各种各样的问题，却经常被忽视。

简言之，DRY原则鼓励对代码进行抽象，但是鼓励得过了头。DRY原则说，如果你发现重复的代码，就把它们提取出来做成一个“模板”或者“框架”。对于抽象我非常的在行，实际上程序语言专家做的许多研究，就是如何设计更好的抽象。然而我并不奉行所谓DRY原则，并不是尽一切可能避免“重复”。“避免重复”并不等于“抽象”。有时候适当的重复代码是有好处的，所以我有时候会故意的进行重复。

抽象与可读性的矛盾

代码的“抽象”和它的“可读性”（直观性），其实是一对矛盾的关系。适度的抽象和避免重复是有好处的，它甚至可以提高代码的可读性，然而如果你尽“一切可能”从代码里提取模板，甚至把一些微不足道的“共同点”也提出来进行“共享”，它就开始有害了。这是因为，模板并不直接显示在“调用”它们的位置。提取出模板，往往会使阅读代码时不能一目了然。如果由此带来的直观性损失超过了模板所带来的好处时，你就应该考虑避免抽象了。要知道，代码读的次数要比写的次数多很多。很多人为了暂时的“写的快感”，过早的提取出不必要的模板，其实损失了读代码时的直观性。如果自己的代码连自己都不能一目了然，你就不能写出优雅的代码。

举一个实际的例子。奉行DRY原则的人，往往喜欢提取类里面的“共同field”，把它们放进一个父类，然后让原来的类继承这个父类。比如，本来的代码可能是：

```
class A {  
    int a;  
    int x;  
    int y;  
}  
  
class B {  
    int a;  
    int u;  
    int v;  
}
```

奉行DRY原则的人喜欢把它改成这样：

```
class C {  
    int a;  
}  
  
class A extends C {  
    int x;  
    int y;  
}  
  
class B extends C {  
    int u;  
    int v;  
}
```

后面这段代码有什么害处呢？它的问题是，当你看到class A和class B的定义时，你不再能一目了然的看到int a这个field。“可见性”，对于程序员能够产生直觉，是非常重要的。这种无关紧要的field，其实大部分时候都没必要提出来，造出一个新的父类。很多时候，不同类里面虽然有同样的int a这样的field，然而它们的含义却是完全不同的。有些人不管三七二十一就来个“DRY”，结果不但没带来好处，反而让程序难以理解。

抽象的时机问题

奉行DRY原则的人还有一个问题，就是他们随时都在试图发现“将来可能重用”的代码，而不是等到真的出现重复的时候再去做抽象。很多时候他们提取出一个貌似“经典模板”，结果最后过了几个月发现，这个模板在所有代码里其实只用过一次。这就是因为他们过早的进行了抽象。

抽象的思想，关键在于“发现两个东西是一样的”。然而很多时候，你开头觉得两个东西是一回事，结果最后发现，它们其实只是肤浅的相似，而本质完全不同。同一个int a，其实可以表示很多种风马牛不及的性质。你看到都是int a就提出来做个父类，其实反而让程序的概念变得混乱。有的时候，有些东西开头貌似同类，后来你增添了新的逻辑之后，发现它们的用途开始特殊化，后来就分道扬镳了。过早的提取模板，反而捆住了你的手脚，使得你为了所谓“一

致性”而重复一些没用的东西。这样的一致性，其实还不如针对每种情况分别做特殊处理。

防止过早抽象的方法其实很简单，它的名字叫做“等待”。其实就算你不重用代码，真的不会死人的。时间能够告诉你一切。如果你发现自己仿佛正在重复以前写过代码，请先不要停下来，请坚持把这段重复的代码写完。如果你不把它写出来，你是不可能准确的发现重复的代码的，因为它们很有可能到最后其实是不一样的。

你还应该避免没有实际效果的抽象。如果代码才重复了两次，你就开始提取模板，也许到最后你会发现，这个模板总共也就只用了两次！只重复了两次的代码，大部分时候是不值得为它提取模板的。因为模板本身也是代码，而且抽象思考本身是需要一定代价的。所以最后总的开销，也许还不如就让那两段重复的代码待在里面。

这就是为什么我喜欢一种懒懒的，笨笨的感觉。因为我懒，所以我不会过早的思考代码的重用。我会等到事实证明重用一定会带来好处的时候，才会开始提取模板，进行抽象。经验告诉我，每一次积极地寻找抽象，最后的结果都是制造一些不必要的模板，搞得自己的代码自己都看不懂。很多人过度强调DRY，强调代码的“重用”，随时随地想着抽象，结果被这些抽象搅混了头脑，bug百出，寸步难行。如果你不能写出“可用”(usable)的代码，又何谈“可重用”(reusable)的代码呢？

谨慎的对待所谓原则

说了这么多，我是在支持DRY，还是反对DRY呢？其实不管是支持还是反对它，都会表示我在乎它，而其实呢，我完全不在乎这类原则，因为它们非常的肤浅。这就像你告诉我说你有一个重大的发现，那就是“ $1+1=2$ ”，我该支持你还是反对你呢？我才懒得跟你说话。人们写程序，本来自然而然就会在合适的时候进行抽象，避免重复，怎么过了几十年后，某个菜鸟给我们的做法起了个名字叫DRY，反而他成了“大师”一样的人物，我倒要用“DRY”这个词来描述我一直在干的事情呢？所以我根本不愿意提起“DRY”这个名字。

所以我觉得这个DRY原则根本就不应该存在，它是一个根本没有资格提出“原则”的人提出来的。看看他鼓吹的其它低劣东西（比如Agile，Ruby），你就会发现，他是一个兜售减肥药的“软件工程专家”。世界上有太多这样的肤浅的所谓原则，我不想对它们一一进行评价，这是在浪费我的时间。世界上有比这些喜欢提出“原则”的软件工程专家深邃很多的人，他们懂得真正根本的原理。

所谓软件工程

很多编程的人包括我，头衔叫做“软件工程师”（software engineer），然而我却不喜欢这个名字。我喜欢把自己叫做“程序员”（programmer）或者“计算机科学家”（computer scientist）。这是为什么呢？这需要从“软件工程”（software engineering）在现实中的涵义谈起。

有人把软件工程领域的本质总结为：“How to program if you cannot？”（如果你不会编程，那么你如何编程？）我觉得这句话说得很好，因为我发现软件工程这整个领域，基本就是吹牛扯淡卖“减肥药”的。软件行业的大部分莫名其妙的愚昧行为，很多是由所谓“软件工程专家”发明的。

总有人提出一套套的所谓“方法论”或者“原则”，比如 Extreme Programming，Design Patterns，Agile，Pair Programming，Test Driven Development (TDD)，DRY principle…… 他们把这些所谓方法论兜售给各个软件公司，鼓吹它们的各种好处，说使用这些方法，就可以用一些平庸的“软件工程师”，制造出高质量低成本的软件。这就跟减肥药的广告一样：不用运动，不用节食，一个星期瘦 20 斤。

你开头还不以为然，觉得这些肤浅的说法能造成什么影响。结果久而久之，这些所谓“方法论”和“原则”成为了整个行业的教条，造成了文化大革命一样的风气。违反这些教条的人，必然被当成菜鸟一样鄙视，当成小学生一样教育，当成反革命一样批斗。就算你技术比这些教条的提出者高不知道多少倍，也无济于事，因为他们已经靠着一张嘴占据了自己的地位。

打破软件工程幻觉的一个办法，就是实地去看看“专家”们用自己的方法论做出了什么好东西。你会惊奇的发现，这些提出各种新名词的所谓“专家”，几乎都是从不知道什么旮旯里冒出来的民科。他们跟真正的计算机科学家或者高明的程序员没有任何关系，也没有做出过什么有技术含量的东西，他们根本没有资格对别人编程的方式做出指导。这些人做出来少数有点用的东西（比如 JUnit），其实非常容易，以至于每个初学编程的人都应该做得出来。一个程序员见识需要低到什么程度，才会在乎这种人说的话？

可世界上就是有这样划算的行当，虽然写不出好的代码，对计算的理解非常肤浅，却可以通过嘴里说说，得到评价别人“代码质量”的权力，占据软件公司的管理层位置。久而久之，别人还以为他们是什么泰斗。你仔细看过提出 Design Pattern 的“四人帮”（GoF），做出过什么有实质价值的东西吗？提出“DRY Principle”的作者，做出过什么吗？再看看 Agile，Pair Programming，TDD 的提出者？他们其实不懂很多编程，写出文章和书来也是极其肤浅。

所谓“软件工程”，并不像土木工程，机械工程，电机工程，是建立在实际的，科学的基础上的。跟这些“硬工程”不一样，软件弄得不好不会出人命，也不会像芯片公司那样，出一个 bug 立即导致几十上百亿的损失。

所以研究软件工程，似乎特别容易钻空子，失败了之后也容易找借口和替罪羊。如果说我的方法不好，你有什么证据吗？口说无凭，我浪费了你多少时间呢？你的具体执行是不是完全照我说的来的呢？你肯定有什么细节没按我说的做，所以才会失败。总之，如果你用了我的办法不管用，那是你自己的问题！

想起这些借口我就想起一个笑话：两夫妻发现床上有跳蚤，身上被咬了好多包。去买了号称“杀伤率 100 %”的跳蚤药，撒了好多在床上。第二天早上起来，发现仍然被咬了好多新的包。妻子责怪丈夫，说他没看说明书就乱撒。结果丈夫打开说明书一看，内容如下：

本跳蚤药使用方法：

1. 抓住跳蚤
2. 煞开跳蚤的嘴
3. 把药塞进跳蚤嘴里
4. 合上跳蚤的嘴

我发现很多软件工程的所谓方法论失败之后的借口，跟这跳蚤药的说明书很像。

人都想省钱，雇用高质量的程序员不容易，所以很多公司还是上钩了。他们请这些“软件工程专家”来到公司，推行各种各样的方法论。推行所谓的 agile，煞有介事的搞一些 stand-up meeting，scrum 之类形式主义东西，以为这些过家家似的做法就能提高开发质量和效率，结果最后都失败了。这是为什么呢？因为再高明的方法论，也无法代替真正的，精华的计算机科学教育。

编程的宗派

总是有人喜欢争论这类问题，到底是“函数式编程”（FP）好，还是“面向对象编程”（OOP）好。既然出了两个帮派，就有人积极地做它们的帮众，互相唾骂和鄙视。然后呢又出了一个“好好先生帮”，这个帮的人喜欢说，管它什么范式呢，能解决问题的工具就是好工具！我个人其实不属于自己这三帮人中的任何一个。

面向对象编程（Object-Oriented Programming）

如果你看透了表面现象就会发现，其实“面向对象编程”本身没有引入很多新东西。所谓“面向对象语言”，就是经典的“过程式语言”（比如 Pascal），加上一点抽象能力。所谓“类”和“对象”，基本是过程式语言里面的记录（record，或者叫结构，structure），它本质其实是一个从名字到数据的“映射表”（map）。

你可以用名字从这个表里面提取相应的数据。比如 point.x，就是用名字 x 从记录 point 里面提取相应的数据。这比起数组来是一件更方便的事情，因为你不需要记住存放数据的下标。即使你插入了新的数据成员，仍然可以用原来的名字来访问已有的数据，而不用担心下标错位的问题。

所谓“对象思想”（区别于“面向对象”），实际上就是对这种数据访问方式的进一步抽象。一个经典的例子就是平面点的数据结构。如果你把一个点存储为：

```
struct Point {  
    double x;  
    double y;  
}
```

那么你用 point.x 和 point.y 可以直接访问它的 X 和 Y 坐标。你也可以把它存储为“极坐标”方式：

```
struct Point {  
    double r;  
    double angle;  
}
```

这样你可以用 point.r 和 point.angle 访问它的模和角度。现在问题来了，如果你的代码开头把 Point 定义为第一种 XY 的方式，使用 point.x, point.y 访问 X 和 Y 坐标，后来你又决定改变 Point 的存储方式，用极坐标，你却不想修改已有的含有 point.x 和 point.y 的代码，怎么办呢？

这就是“对象思想”的价值，它让你可以通过“间接”（indirection，或者叫做“抽象”）来改变 point.x 和 point.y 的语义，从而让使用者的代码完全不用修改。虽然你的实际数据结构里面可能没有 x 和 y 这两个成员，但由于 .x 和 .y 可以被重新定义，所以你可以通过改变 .x 和 .y 的定义来“模拟”它们。在你使用 point.x 和 point.y 的时候，系统内部其实在运行两片代码（所谓 getter），它们的作用是从 r 和 angle 计算出 x 和 y 的值。这样你的代码就感觉 x 和 y 是实际存在的成员一样，而其实它们是被临时算出来的。

在 Python 之类 的语言里面，你可以通过定义 “[property](#)” 来直接改变 point.x 和 point.y 的语义。在 Java 里稍微麻烦一些，你需要使用 point.getX() 和 point.getY() 这样的写法。然而它们最后的目的其实都是一样的——它们为数据访问提供了一层“间接”（抽象）。

这种抽象有时候是个好主意，它甚至可以跟量子力学的所谓“不可观测性”扯上关系。你觉得这个原子里面有 10 个电子？也许它们只是像 point.x 给你的幻觉一样，也许宇宙里根本就没有电子这种东西，也许你每次看到所谓的电子，它都是临时生成出来逗你玩的呢？然而，对象思想的价值也就到此为止了。你见过的所谓“面向对象思想”，几乎无一例外可以从这个想法推广出来。面向对象语言的绝大部分特性，其实是过程式语言早就提供的。因此我觉得，其实没有语言可以叫做“面向对象语言”。就像一个人为一个公司贡献了一点点代码，并不足以让公司以他的名字命名一样。

“对象思想”作为数据访问的方式，是有一定好处的。然而“面向对象”（多了“面向”两个字），就是把这种本来良好的思想东拉西扯，牵强附会，发挥过了头。很多面向对象语言号称“所有东西都是对象”（Everything is an Object），把所有函数都放进所谓对象里面，叫做“方法”（method），把普通的函数叫做“静态方法”（static method）。实际上呢，就像我之前的例子，只有极少需要抽象的时候，你需要使用内嵌于对象之内，跟数据紧密结合的“方法”。其他的时候，你其实只是想表达数据之间的变换操作，这些完全可以用普通的函数表达，而且这样做更加简单和直接。

这种把所有函数放进方法的做法是本末倒置的，因为函数并不属于对象。绝大部分函数是独立于对象的，它们不能被叫做“方法”。强制把所有函数放进它们本来不属于的对象里面，把它们全都作为“方法”，导致了面向对象代码逻辑过度复杂。很简单的想法，非得绕好多道弯子才能表达清楚。很多时候这就像把自己的头塞进屁股里面。

这就是为什么我喜欢开玩笑说，面向对象编程就像“[地平说](#)”（Flat Earth Theory）。当然你可以说地球是一个平面。对于局部的，小规模的现象，它没有问题。然而对于通用的，大规模的情况，它却不是自然，简单和直接的。直到[今天](#)，你仍然可以无止境的寻找证据，扭曲各种物理定律，自圆其说地平说的幻觉，然而这会让你的理论非常复杂，经常需要缝缝补补还难以理解。

面向对象语言不仅有自身的根本性错误，而且由于面向对象语言的设计者们常常是半路出家，没有受到过严格的语言理论和设计训练却又自命不凡，所以经常搞出另外一些奇葩的东西。比如在 JavaScript 里面，每个函数同时又可以作为构造函数（constructor），所以每个函数里面都隐含了一个 this 变量，你嵌套多层对象和函数的时候就发现没法访问外层的 this，非得“bind”一下。Python 的变量定义和赋值不分，所以你需要访问全局变量的时候得用 global 关键字，后来又发现如果要访问“中间层”的变量，没有办法了，所以又加了个 nonlocal 关键字。Ruby 先后出现过四种类似 lambda 的东西，每个都有自己的怪癖……有些人问我为什么有些语言设计成那个样子，我只能说，很多语言设计者其实根本不知道自己在干什么。

软件领域就是喜欢制造宗派。“面向对象”当年就是乘火打劫，扯着各种幌子，成为了一种宗派，给很多人洗了脑。到底什么样的语言才算是“面向对象语言”？这样基本的问题至今没有确切的答案，足以说明所谓面向对象，基本都是扯淡。每当你指出某个 OO 语言 X 的弊端，就会有人跟你说，其实 X 不是“地道的” OO 语言，你应该去看看另外一个 OO 语言 Y。等你发现 Y 也有差不多的问题，有人又会让你去看 Z……直到最后，他们告诉你，只有 Smalltalk 才是地道的 OO 语言。这不是很搞笑吗，说一个根本没人用的语言才是地道的 OO 语言，这就像在说只有死人的话才是对的。这就像是一群政客在踢皮球，推卸责任。

等你真正看看 Smalltalk 才发现，其实面向对象语言的根本毛病就是由它而来的，Smalltalk 并不是很好的语言。很多人至今不知道自己所用的“面向对象语言”里面的很多优点，都是从过程式语言继承来的。每当发生函数式与面向对象式语言的口水战，都会有面向对象的帮众拿出这些过程式语言早就有的优点来进行反驳：“你说面向对象不好，看它能做这个……”拿别人的优点撑起自己的门面，却看不到事物实质的优点，这样的辩论纯粹是鸡同鸭讲。

函数式编程 (Functional Programming)

函数式语言一直以来比较低调，直到最近由于并发计算编程瓶颈的出现，以及 Haskell，Scala 之类语言社区的大力鼓吹，它忽然变成了一种宗派。有人盲目的相信函数式编程能够奇迹般的解决并发计算的难题，而看不到实质存在的，独立于语言的问题。被函数式语言洗脑的帮众，喜欢否定其它语言的一切，看低其它程序员。特别是有些初学编程的人，俨然把函数式编程当成了一天瘦二十斤的减肥神药，以为自己从函数式语言入手，就可以对经验超过他十年以上的老程序员说三道四，仿佛别人不用函数式语言就什么都不懂一样。

函数式编程的优点

函数式编程当然提供了它自己的价值。函数式编程相对于面向对象最大的价值，莫过于对于函数的正确理解。在函数式语言里面，函数是“一类公民”（first-class）。它们可以像 1, 2, “hello”, true, 对象……之类的“值”一样，在任意位置诞生，通过变量，参数和数据结构传递到其它地方，可以在任何位置被调用。这些是很多过程式语言和面向对象语言做不到的事情。很多所谓“面向对象设计模式”（design pattern），都是因为面向对象语言没有 first-class function，所以导致了每个函数必须被包在一个对象里面才能传递到其它地方。

函数式编程的另一个贡献，是它们的类型系统。函数式语言对于类型的思维，往往非常的严密。函数式语言的类型系统，往往比面向对象语言来得严密和简单很多，它们可以帮助你对程序进行严密的逻辑推理。然而类型系统一是把双刃剑，如果你对它看得太重，它反而会带来不必要的复杂性和过度工程。这个我在下面讲讲。

各种“白象”（white elephant）

所谓白象，“white elephant”，是指被人奉为神圣，价格昂贵，却没有实际用处的东西。函数式语言里面有很好的东西，然而它们里面有很多多余的特性，这些特性跟白象的性质类似。

函数式语言的“拥护者”们，往往认为这个世界本来应该是“纯”（pure）的，不应该有任何“副作用”。他们把一切的“赋值操作”看成低级弱智的作法。他们很在乎所谓尾递归，类型推导，fold，currying，maybe type 等等。他们以自己能写出使用这些特性的代码为豪。可是殊不知，那些东西其实除了能自我安慰，制造高人一等的幻觉，并不一定能带来真正优秀可靠的代码。

纯函数

半壶水都喜欢响叮当。很多喜欢自吹为“函数式程序员”的人，往往并不真的理解函数式语言的本质。他们一旦看到过程式语言的写法就嗤之以鼻。比如以下这个 C 函数：

```
int f(int x) {
    int y = 0;
    int z = 0;
    y = 2 * x;
    z = y + 1;
    return z / 3;
}
```

很多函数式程序员可能看到那几个赋值操作就皱起眉头，然而他们看不到的是，这是一个真正意义上的“纯函数”，它在本质上跟 Haskell 之类语言的函数是一样的，也许还更加优雅一些。

盲目鄙视赋值操作的人，也不理解“数据流”的概念。其实不管是对局部变量赋值还是把它们作为参数传递，其实本质

上都像是把一个东西放进一个管道，或者把一个电信号放在一根导线上，只不过这个管道或者导线，在不同的语言范式里放置的方向和样式有一点不同而已！

对数据结构的忽视

函数式语言的帮众没有看清楚的另一个重要的，致命的东西，是数据结构的根本性和重要性。数据结构的有些问题是“物理”和“本质”地存在的，不是换个语言或者换个风格就可以奇迹般消失掉的。函数式语言的拥护者们喜欢盲目的相信和使用列表（list），而没有看清楚它的本质以及它所带来的复杂度。列表带来的问题，不仅仅是编程的复杂性。不管你怎么聪明的使用它，很多性能问题是根本没法解决的，因为列表的拓扑结构根本就不适合用来干有些事情！

从数据结构的角度看，Lisp 所谓的 list 就是一个单向链表。你必须从上一个节点才能访问下一个，而这每一次“间接寻址”，都是需要时间的。在这种数据结构下，很简单的像 length 或者 append 之类函数，时间复杂度都是 $O(n)$ ！为了绕过这数据结构的不足，所谓的“Lisp 风格”告诉你，不要反复 append，因为那样复杂度是 $O(n^2)$ 。如果需要反复把元素加到列表末尾，那么应该先反复 cons，然后再 reverse 一下。

很可惜的是，当你同时有递归调用，就会发现 cons + reverse 的做法颠来倒去的，非常容易出错。有时候列表是正的，有时候是反的，有时候一部分是反的……这种方式用一次还可以，多几层递归之后，自己都把自己搞糊涂了。好不容易做对了，下次修改可能又会出错。然而就是有人喜欢显示自己聪明，喜欢自虐，迎着这类人为制造的“困难”勇往直前。

富有讽刺意味的是，半壶水的 Lisp 程序员都喜欢用 list，真正的 Lisp 大师级人物，却知道什么时候应该使用记录（结构）或者数组。在 Indiana 大学，我曾经上过一门 Scheme（一种现代 Lisp 方言）编译器的课程，授课的老师是 R. Kent Dybvig，他是世界上最先进的 Scheme 编译器 Chez Scheme 的作者。我们的课程编译器的数据结构（包括 AST）都是用 list 表示的。到了期末的时候，Kent 对我们说：“你们的编译器已经可以生成跟我的 Chez Scheme 媲美的代码，然而 Chez Scheme 不止生成高效的目标代码，它的编译速度是你们的 700 倍以上。它可以在 5 秒钟之内编译它自己。”然后他透露了一点 Chez Scheme 速度快的原因。其中一个原因，就是因为 Chez Scheme 的内部数据结构不是 list。在编译一开头的时候，Chez Scheme 就已经把输入代码转换成了数组一样的，固定长度的结构。后来在工业界的经验教训也告诉了我，数组比起链表，确实在某些时候有大幅度的性能提升。在什么时候该用链表，什么时候该用数组，是一门艺术。

副作用的根本价值

对数据结构的忽视，跟纯函数式语言盲目排斥副作用的“教义”有很大关系。过度的使用副作用当然是有害的，然而副作用这种东西，其实是根本的，有用的。对于这一点，我喜欢跟人这样讲：在计算机和电子线路最开头发明的时候，所有的线路都是“纯”的，因为逻辑门和导线没有任何记忆数据的能力。后来有人发明了触发器（flip-flop），才有了所谓“副作用”。是副作用让我们可以存储中间数据，从而不需要把所有数据都通过不同的导线传输到需要的地方。没有副作用的语言，就像一个没有无线电，没有光的世界，所有的数据都必须通过实在的导线传递，这许多纷繁的电缆，必须被正确的连接和组织，才能达到需要的效果。我们为什么喜欢 WiFi，4G 网，Bluetooth，这也就是为什么一个语言不应该是“纯”的。

副作用也是某些重要的数据结构的重要组成元素。其中一个例子是哈希表。纯函数语言的拥护者喜欢盲目的排斥哈希表的价值，说自己可以用纯的树结构来达到一样的效果。然而事实却是，这些纯的数据结构是不可能达到有副作用的数据结构的性能的。所谓纯函数数据结构，因为在每一次“修改”时都需要保留旧的结构，所以往往需要大量的拷贝数据，然后依赖垃圾回收（GC）去消灭这些旧的数据。要知道，内存的分配和释放都是需要时间和能量的。盲目的依赖 GC，导致了纯函数数据结构内存分配和释放过于频繁，无法达到有副作用数据结构的性能。要知道，副作用是电子线路和物理支持的高级功能。盲目的相信和使用纯函数写法，其实是在浪费已有的物理支持的操作。

fold以及其他

大量使用 fold 和 currying 的代码，写起来貌似很酷，读起来却不必要的痛苦。很多人根本不明白 fold 的本质，却老喜欢用它，因为他们觉得那是函数式编程的“精华”，可以显示自己的聪明。然而他们没有看到的是，fold 包含的精髓，只不过是在列表（list）上做递归的“通用模板”，这个模板需要你填进去三个参数，就可以生成一个新的递归函数调用。所以每一个 fold 的调用，本质上都包含了一个在列表上的递归函数定义。

Fold 的问题在于，它定义了一个递归函数，却没有给它一个一目了然的名字。使用 fold 的结果是，每次看到一个 fold 调用，你都需要重新读懂它的定义，琢磨它到底是干什么的。而且 fold 调用只显示了递归模板需要的部分，而把递归的主体隐藏在了 fold 本身的“框架”里。比起直接写出整个递归定义，这种遮遮掩掩的做法，其实是更难理解的。比如，当你看到这句 Haskell 代码：

```
foldr (+) 0 [1,2,3]
```

你知道它是做什么的吗？也许你一秒钟之后就凭经验琢磨出，它是在对 [1,2,3] 里的数字进行求和，本质上相当于 sum [1,2,3]。虽然只花了一秒钟，可你仍然需要琢磨。如果 fold 里面带有更复杂的函数，而不是 +，那么你可能一分钟都琢磨不透。写起来倒没有费很大力气，可为什么我每次读这段代码，都需要看到 + 和 0 这两个跟自己的意图毫无关系的东西？万一有人不小心写错了，那里其实不是 + 和 0 怎么办？为什么我需要搞清楚 +, 0, [1,2,3] 的相对位置以及它们的含义？

这样的写法其实还不如老老实实写一个递归函数，给它一个有意义名字（比如 `sum`），这样以后看到这个名字被调用，比如 `sum [1,2,3]`，你想都不用想就知道它要干什么。定义 `sum` 这样的名字虽然稍微增加了写代码时的工作，却给读代码的时候带来了方便。为了写的时候简洁或者很酷而用 `fold`，其实增加了读代码时的脑力开销。要知道代码被读的次数，要比被写的次数多很多，所以使用 `fold` 往往是得不偿失的。然而，被函数式编程洗脑的人，却看不到这一点。他们太在乎显示给别人看，我也会用 `fold`！

与 `fold` 类似的白象，还有 [currying](#)，Hindley-Milner 类型推导等特性。看似很酷，但等你仔细推敲才发现，它们带来的麻烦，比它们解决的问题其实还要多。有些特性声称解决的问题，其实根本就不存在。现在我把一些函数式语言的特性，以及它们包含的陷阱简要列举一下：

1. `fold`。`fold` 等“递归模板”，相当于把递归函数定义插入到调用的地方，而不给它们名字。这样导致每次读代码都需要理解几乎整个递归函数的定义。
2. [currying](#)。貌似很酷，可是被部分调用的参数只能从左到右，依次进行。如何安排参数的顺序成了问题。大部分时候还不如直接制造一个新的 `lambda`，在内部调用旧的函数，这样可以任意的安排参数顺序。
3. Hindley-Milner 类型推导（HM）。为了避免写参数和返回值的类型，结果给程序员写代码增加了很多的限制。为了让类型推导引擎开心，导致了很多完全合法合理优雅的代码无法写出来。其实还不如直接要程序员写出参数和返回值的类型，这工作量真的不多，而且可以准确的帮助阅读者理解参数的范围。HM 类型推导的根本问题其实在于它使用 unification 算法。Unification 其实只能表示数学里的“等价关系”（equivalence relation），而程序语言最重要的关系，subtyping，并不是一个等价关系，因为它不具有对称性（symmetry）。
4. 代数数据类型（algebraic data type）。所谓“代数数据类型”，其实并不如普通的类型系统（比如 Java 的）通用。很多代数数据类型系统具有所谓 sum type，这种类型其实带来过多的类型嵌套，不如通用的 union type。盲目崇拜代数数据类型的人，往往是因为盲目的相信“数学是优美的语言”。而其实事实是，数学是一种历史遗留的，毛病很多的语言。数学的语言根本没有经过系统的，全球协作的设计。往往是数学家在黑板上随便写个符号，说这个表示某概念，然后就定下来了。
5. Tuple。有代数数据类型的的语言里面经常有一种构造叫做 tuple，比如 Haskell 里面可以写 `(1, "hello")`，表示一个类型为 `(Int, String)` 的结构。这种构造经常被人看得过于高尚，以至于用在超越它能力的地方。其实 tuple 的本质就是一个没有名字的结构（类似 C 的 structure）。临时使用 tuple 貌似很方便，因为不需要定义一个结构类型。然而因为 tuple 没有名字，里面的成员没法用名字访问，一旦多加几个成员就发现很麻烦了。Tuple 往往只能通过模式匹配来获得里面的域，一旦你增加了新的域进去，所有对这个 tuple 的模式匹配代码都需要修改。所以 tuple 一般只能用在大小不超过 2 的情况下，而且必须确信以后不会增加新的域进去。
6. [惰性求值](#)（lazy evaluation）。貌似数学上很优雅，但其实有严重的逻辑漏洞。因为 bottom（死循环）成为了任何类型的一个元素，所以取每一个值，都可能导致死循环。同时导致代码性能难以预测，因为求值太懒，所以可能临时抱佛脚做太多工作，而平时浪费 CPU 的时间。由于到需要的时候才求值，所以在有多个处理器的时候无法有效地利用它们的计算能力。
7. 尾递归。大部分尾递归都相当于循环语句，然而却不像循环语句一样，能够一目了然看明白它们的意图。你需要仔细看每个分支的返回位置，判断它是不是递归，然后才能判断这代码其实是个循环。而循环语句从关键字（`for`, `while`）就知道这是一个循环，不需要去看里面是什么结构。所以等价于循环的尾递归，最好还是写成专门的循环语句。当然，尾递归在另一些情况下是有用的，这些情况不等价于循环，而是“树递归”。在这种情况下使用循环，经常需要复杂的 `break` 或者 `continue` 条件，导致循环不易理解。所以循环和尾递归都是有必要的，不要总想着用一个代替另外一个，要分情况选择合适的方式。

好好先生

很多人避免“函数式 vs 面向对象”的辩论，于是他们成为了“好好先生”。这种人没有原则的认为，任何能够解决当前问题的工具就是好工具。也就是这种人，喜欢使用 shell script，喜欢折腾各种 Unix 工具，因为显然，它们能解决他“手头的问题”。

然而这种思潮是有害的，它的害处其实更胜于投靠函数式或者面向对象。没有原则的好好先生们忙着“解决问题”，却不能清晰地看到这些问题为什么存在。他们所谓的问题，往往是由于现有工具的设计失误。由于他们的“随和”，他们从来不去思考，如何从根源上消灭这些问题。他们在一堆历史遗留的垃圾上缝缝补补，妄图使用设计恶劣的工具建造可靠地软件系统。当然，这代价是非常大的。不但劳神费力，而且也许根本不能解决问题。

所以每当有人让我谈谈“函数式 vs 面向对象”，我都避免说“各有各的好处”，因为那样的话我会很容易被当成这种毫无原则的好好先生。

符号必须简单的对世界建模

从上面你已经看出，我既不是一个铁杆“函数式程序员”，也不是一个铁杆“面向对象程序员”，我也不是一个爱说“各有各的好处”的好好先生。我是一个有原则的批判性思维者。我不但看透了各种语言的本质，而且看透了它们之间的统一关系。我编程的时候看到的不是表面的语言和程序，而是一个类似电路的东西。我看到数据的流动和交换，我看到效率的瓶颈，而这些都是跟具体的语言和范式无关的。

在我的心目中其实只有一个概念，它叫做“编程”（programming），它不带有任何附加的限定词（比如“函数式”或者“面向对象”）。我研究的领域称叫做“Programming Languages”，它研究的内容不局限于某一个语言，也不局限于某一类语言，而是所有的语言。在我的眼里，所有的语言都不过是各个特性的组合。所以最近出现的所谓“新语言”，其实不大可能再有什么真正意义上的创新。我不喜欢说“发明一个程序语言”，不喜欢使用“发明”这个词，因为不管你怎么设计一个语言，所有的特性几乎都早已存在于现有的语言里面了。我更喜欢使用“设计”这个词，因为虽然一个语言没有任何新的特性，它却有可能在细节上更加优雅。

编程最重要的事情，其实是让写出来的符号，能够简单地对实际或者想象出来的“世界”进行建模。一个程序员最重要的能力，是直觉地看见符号和现实物体之间的对应关系。不管看起来多么酷的语言或者范式，如果必须绕着弯子才能表达程序员心目中的模型，那么它就不是一个很好的语言或者范式。有些东西本来就是有随时间变化的“状态”的，如果你偏要用“纯函数式”语言去描述它，当然你就进入了那些 monad 之类的死胡同。最后你不但没能高效的表达这种副作用，而且让代码变得比过程式语言还要难以理解。如果你进入另一个极端，一定要用对象来表达本来很纯的数学函数，那么你一样会把简单的问题搞复杂。Java 的所谓 design pattern，很多就是制造这种问题的，而没有解决任何问题。

关于建模的另外一个问题是，你心里想的模型，并不一定是最好的，也不一定非得设计成那个样子。有些人心里没有一个清晰简单的模型，觉得某些语言“好用”，就因为它们能够对他那种扭曲纷繁的模型进行建模。所以你就跟这种人说不清楚，为什么这个语言不好，因为显然这个语言对他是有用的！如何简化模型，已经超越了语言的范畴，在这里我就不细讲了。

英语口音

我目前生活在一个说英语的国家，然而我对英语的用法和态度却跟很多人不一样。我并不认为英语说得“地道”是一种好事。英语说得太地道，或者试图说得地道的中国人，总是让我有一种异样的感觉。有些人一眼看去不错，可是说了几句话之后，我就失去了很多的好感。很多时候出现这种情况，都是因为这个人说话用的语言显示出了他内心的自卑。

首先，我认为中国人和中国人说话，不管身在哪个国家，都应该尽量使用中文。这就像其它任何国家的人之间对话，都应该用他们自己的语言一样。除非个别的单词和术语没法很好的翻译，才使用英语。或者说话的时候有外国友人参与，在这种情况下可以为了照顾外籍人士而使用英文。

如果你不是在美国土生土长的美籍华人，说一口很地道的美式英语，其实并不能获得人的好感。我觉得中国人说英语本来就应该有点口音，就像法国人，德国人，日本人，俄国人，说英语都有自己特有的口音一样。有一定的口音，发音不“地道”，其实是自尊和自信的表现，这说明你知道自己的价值，没有为了让别人看得起你而刻意模仿美国人说话。中国人使用美国俚语，或者发信息用“lol”之类的简写，看起来很“融入文化”，其实是不自信的表现。有人喜欢模仿 Friends 电视剧里的那种夸张的语气，特别是 Joey 的语气，其实让人厌烦。

我自己说英语的时候，都不会刻意去强化那个“er 音”。我觉得美国英语的卷舌音“er”特别不好听，舌头放平一点，接近英式发音，会好很多。但我也不会刻意去模仿“伦敦音”，因为无论模仿哪种标准发音，都会显得很刻意，假惺惺的。从小我就说不标准任何地方的方言。我总是把我觉得不好听的音去掉。所以刚认识我的人都纳闷我是哪里人，因为我说的不是任何地方的方言，也不是标准的普通话。

总之，中国人之间使用英语甚至地道的美国口音，被我认为是不自信的表现。我喜欢周杰伦说过的一句话：“我是中国人，所以我的英文不好！”我希望每个中国人都有这样的志气。

智商的圈套

上次买了个任天堂3DS游戏机，觉得里面的游戏很无聊，所以第二天就把游戏机连同游戏一起，转手倒卖给了别人。从那天之后，我开始琢磨一个问题——到底是什么让我觉得一个游戏好玩或者不好玩。我似乎对事物有一种很特别的品味，很多别人说“好玩”，“有趣”的游戏或者电影，我一看就觉得很无趣，或者很自虐。我一生中玩过最好玩的游戏，其实没有几个，可能掰着手指头都数得出来：[Braid](#), [Limbo](#), [Klonoa](#) (風のクロノア door to phantomile), 《纪念碑谷》, [Metal Gear Solid](#),



如果你觉得我智商太高，所以才觉得很多游戏没有挑战性，不好玩，那么你其实并不了解我。我并不是一个“智商达人”，我不追求挑战性。我觉得很多游戏缺乏的不是挑战性和“难度”，而是设计的巧妙。很多游戏我根本没法玩过关，却只是觉得呆板，繁琐，老套，公式化。我并不会因为游戏玩不过关，作业做不出来，或者书看不懂而沮丧。恰恰相反，我认为我的智力根本就不应该是用来干这些事情的。如果有事情让我觉得沮丧，我一般都认为是这个事情有问题，而不是我有问题。如果说我也有错的话，那么我的错误就在于选择了参与这项活动，我根本不应该做这件事情。这就是为什么我大部分时候都比一般人开心。

我觉得很多人有一种奇怪的倾向，他们喜欢挑战或者彰显自己的智商。每当我向人推荐类似Braid的游戏，他们就会认为我喜欢“解谜题”，于是他们给我推荐类似Zelda或者Antichamber之类的游戏，告诉我它们很考智力。可是这样的游戏，我一般玩不到几分钟就开始觉得无聊。这说明我并不是喜欢“解谜题”，而是因为另外一些特征而喜欢某些游戏。喜欢玩Zelda, Antichamber, 或者《生化危机》一类游戏的人，往往有一种自虐倾向。这种人似乎很在乎自己的智商，所以游戏玩了不久之后，就会被“套牢”。他们会认为能够把某个游戏打通关，是对自己智商的认可。如果你跟他说这游戏太难太麻烦，他就会开始鄙视你的智力，吹嘘自己只花了多么短的时间就玩通关了。

然而如果你退后一步，就会发现这些游戏，其实都存在某种“[设计公式](#)”。一旦掌握了这些公式，你就可以轻而易举地制造出这样的游戏。然后你就会发现，热衷于这些游戏的人，其实并不聪明，因为他们被游戏的设计者玩弄于鼓掌之中，而没能发现其中的设计公式。这些人为了得到别人的认可，检验或者训练所谓的“智力”，甚至为了“合群”，选择了这类只能叫做“自虐型”的游戏。

这种游戏玩到后来，你就会发现这不是在娱乐，而是在完成任务，不是你在玩游戏，而是游戏在玩你。你盼望它早点结束，但却无法立即罢手，因为你对自己说：“如果现在半途而废，我就是一个懦夫，一个笨蛋，就不再是一个天才……”你在虚拟的空间中来回的游走，摸索和寻找那些能打开机关的“钥匙”，而它们被游戏的设计者故意放在一些让人恼火的地方。你感觉到的不是快乐，而是繁琐，沮丧和空虚。

我发现容易落入这种圈套的人，他们在日常生活和工作中也容易出现类似的倾向。总的说来，这种人正如卓别林的《大独裁者》最后的[演讲](#)所描述的，“想得太多，感觉太少” (think too much, feel too little)。这种人如果沿着这条道路发展下去，就会变成像机器一样思考的人。正是这种人，给世界带来了灾难。希特勒就是这样一种人的典型代表，他太在乎自己是否优秀和聪明，却感觉不到人间的爱和痛苦，所以他对自己认为是劣等民族的人进行残酷的屠杀。

所以，我其实并不是因为智力上的挑战性而喜欢Braid, Limbo, Klonoa等游戏。我喜欢它们，是因为它们充满了创意和想象力，却又不让人觉得繁琐和累赘。在这样的游戏里，你能做一些你从前根本没想到过的事情，它们的设计可以用“妙不可言”来形容。这种游戏的逻辑很连贯流畅，你不需要到处瞎撞，来回跑动，而是一气呵成，行云流水，却又不乏波澜起伏和机智巧妙之处。这就像自己在演出一场出神入化的电影。你感觉到的不是沮丧，迷茫，不是对自己智力的考验和评价，而是真正的愉悦和解脱。

当我推荐Klonoa给一个朋友的时候，我说：“玩这个游戏就感觉是在梦里……”结果他对我说：“你知道另外一个叫什么什么的游戏里面，也有个四维空间吗？……”其实我根本不是在跟他讨论“梦是什么”这种学术问题，而是在说“梦幻的感觉”。这位朋友就属于我前面提到的，“想得太多”的类型。我说像是在梦里，说的是一种感觉，只有心才看得见；而他所理解的“梦”，是一种很理论的东西，就像数学里的多维空间，需要用脑子分析得出来。由于过度理性，他总是忙于分析一些“深层次”的理论，而看不见我能轻松感觉到的乐趣。我对他的建议是：少想一点，少分析一点，多用心感觉。只有用心去体会，你才会理解，Klonoa这样的游戏的价值，其实不在于智力和难度，而在于它让你感觉到的梦幻，创意，自由，想象力，和艺术。

设计的重要性

我曾经在一篇[文章](#)里谈过关于设计的问题，然而那篇文章由于标题不够醒目，可能很多人没有注意看。我觉得现在有必要把里面的内容专门提出来讲一下，因为设计在我的心目中具有至关重要的地位，却被很多计算机科学家和程序员所轻视。

我觉得自己不但是一個计算机科学家和程序员，在很大程度上我还是一个设计师。我不但是一个程序语言的设计师，而且是其它很多东西的设计师。我设计的东西不但常常比别人的简洁好用，而且我经常直接看出其他人的设计里面的问题。我写的代码不仅自己容易看懂，而且别人也容易理解。我有时候受命修补前人的BUG，结果没法看懂他们的代码。在这种情况下，我的解决方案是推翻重写。经我重写之后的代码，不仅没有BUG，而且简洁很多。

很多人自己的设计有问题，太复杂不易用，到头来却把责任推在用户身上，使用类似“皇帝的新装”的技巧，让用户有口难言。之前一篇[文章](#)提到的严重交通事故，就是一个设计问题，却被很多人归结为“人为错误”。这种出人命的事情都这么难引起人们对设计的关注，就更不要说软件行业那些无关性命的恼人之处了。有些人写的代码过度复杂，BUG众多，却仿佛觉得自己可以评估其他人的智商，打心眼里觉得自己是专家，看不懂他代码的人都是笨蛋。

很多程序员有意把“用户”和自己区别开来，好像程序员应该高人一等，不能以用户的标准。所以他们觉得程序员就是应该会用各种难用的工具，难用的操作系统，程序语言，编辑器，…… 他们觉得只要你追求这些东西的“易用性”或者“直观性”，就说明你智商有问题。只要你说某个东西太复杂，另一个东西好用些，他们就会跟你说：“专家才用这个，你那个是菜鸟用的。”这些人不明白，程序员其实也是用户，而且他们是自己的代码的用户，每一次调用自己写的函数，自己都是自己的用户。可是这种鄙视用户的风气之盛行，带来了整个行业不但设计过度复杂，而且以复杂为豪的局面。

经常有人自豪的声称自己的项目有多少万行代码，仿佛代码的行数是衡量一个软件质量的标准，行数越多质量越好，然而事实却恰恰相反。就像[《小王子》](#)作者说的：“一个设计师知道他达到了完美，并不是当他不能再加进任何东西，而是当没有任何东西可以被去掉。”

如果你跟我一样关心设计，却发现身边的人喜欢显示自己能搞懂复杂的东西，跟你说容易的东西都是菜鸟用的，那么你需要一个朋友。书籍是人类最好的朋友，因为它的作者可以跨越时间和空间的限制，给你最需要的支持和鼓励。这就是当我阅读这本1988年出版的[《The Design of Everyday Things》](#)（简称DOET）时的感觉。我觉得，终于有人懂我了！有趣的是，它的作者 Don Norman 曾经是 Apple Fellow，也是[《The Unix-Haters Handbook》](#)一书序言的作者。



DOET 不但包含并且支持了我的博文[《黑客文化的精髓》](#)以及[《程序语言与.....》](#)里的基本观点，而且提出了比[《什么是“对用户友好”》](#)更精辟可行的解决方案。

我觉得这应该是每个程序员必读的书籍。为什么每个程序员必读呢？因为虽然这本书是设计类专业的必读书籍，而计算机及其编程语言和工具，其实才是作者指出的缺乏设计思想的“重灾区”。看了它，你会发现很多所谓的“人为错误”，其实是工具的设计不合理造成的。一个设计良好的工具，应该只需要很少量的文档甚至不需要文档。这本书将提供给你改进一切事物的原则和灵感。你会恢复你的人性。

值得一提的是，虽然 Don Norman 曾经是 Apple Fellow，但我觉得 Apple 产品设计的人性化程度与 Norman 大叔的思维高度还是有一定的差距的。因为我看了这书之后，立马发现了iPhone的一些设计问题。

如果你跟我一样不想用眼睛看书，可以到 Audible 买本[有声书](#)来听。

关于Git的礼节

(这里的内容本来是《[怎样尊重一个程序员](#)》的一小节，但由于Git的使用引起了很普遍的不尊重程序员的现象，现在特别将这一节提出来单独成文。)

Git是现在最流行的代码版本控制工具。用外行话说，Git就是一个代码的“仓库”或者“保管”，这样很多人修改了代码之后，可以知道是谁改了哪一块。其实不管什么工具，不管是编辑器，程序语言，还是版本控制工具，比起程序员的核心思想来，都是次要的东西，都是起辅助作用的。可是Git这工具似乎特别惹人恼火。

Git并不像很多人吹嘘的那么好用，其中有明显的蹩脚设计。跟Unix的传统一脉相承，Git没有一个良好的包装，设计者把自己的内部实现细节无情地泄露给了用户，让用户需要琢磨者设计者内部到底怎么实现的，否则很多时候不知道该怎么办。用户被迫需要记住挺多稀奇古怪的命令，而且命令行的设计也不怎么合理，有时候你需要加-f之类的参数，各个参数的位置可能不一致，而且加了还不一定能起到你期望的效果。各种奇怪的现象，比如“head detached”，都强迫用户去了解它内部是怎么设计的。随着Git版本的更新，新的功能和命令不断地增加，后来你终于看到命令行里出现了foreach，才发现它的命令行就快变成一个（劣质的）程序语言。如果你了解[ydiff](#)的设计思想，就会发现Git之类基于文本的版本控制工具，其实属于古代的东西。然而很多人把Git奉为神圣，就因为它是Linus Torvalds设计的。

Git最让人恼火的地方并不是它用起来麻烦，而是它的“资深用户”们居高临下的态度给你造成的心灵阴影。好些人因为自己“精通Git”就以为高人一等，摆出一副专家的态度。随着用户的增加，Git最初的设计越来越被发现不够用，所以一些约定俗成的规则似乎越来越多，可以写成一本书！跟Unix的传统一脉相承，Git给你很多可以把自己套牢的“机制”，到时候出了问题就怪你自己不知道。所以你就经常听有人煞有介事的说：“并不是Git允许你这么做，你就可以这么做的！Unix的哲学是不阻止傻人做傻事……”如果你提交代码时不知道Git用户一些约定俗成的规则，就会有人嚷嚷：“rebase了再提交！”“不要push到master！”“不要merge！”“squash commits！”如果你不会用git submodule之类的东西，有人可能还会鄙视你，说：“你应该知道这些！”

打个比方，这样的嚷嚷给人的感觉是，你得了奥运会金牌之后，把练习用的器材还回到器材保管科，结果管理员对你大吼：“这个放这边！那个放那边！懂不懂规矩啊你？”看出来问题了吗？程序员提交了有高价值的代码（奥运金牌），结果被一些自认为Git用的很熟的人（器材保管员）厉声呵斥。

一个尊重程序员的公司文化，就应该把程序员作为运动健将，把程序员的代码放在尊贵的地位。其它的工具，都应该像器材保管科一样。我们尊重这些器材保管员，然而如果运动员们不懂你制定的器材摆放规矩，也应该表示出尊重和理解，说话应该和气有礼貌，不应该骑到他们头上。所以，对于Git的一些命令和用法，我建议大家向新手介绍时，这样开场：“你本来不该知道这些的，可是现在我们没有更好的工具，所以得这样弄一下……”

怎样尊重一个程序员

得知一位久违的同学来到了旧金山湾区，然而我见到他时，这人正处于一生中最痛苦的时期。他告诉我，自己任职的公司在他加入之前和之后，判若两人。录取的时候公司对他说，我们对你在实习期间的表现和学术背景非常满意，你不用面试，甚至不用毕业拿学位，直接就可以加入我们公司成为正式员工。然而短短一年后的今天，这位同学已经完全感觉不到公司对自己技能的尊重。Manager 让他做一些乱七八糟没技术含量的事情，还抱怨说他做事太慢，并且在他的 evaluation 上很是写了一笔。在人格尊严和工作安全感的双重打击之下，这位同学压力非常大，周末经常偷偷地加班，仍然无法让 manager 满意。

我很了解这位同学的能力，在任何一流公司任职，肯定是绰绰有余了。他的名字我当然保密，然而他所任职的公司因为太过嚣张，我不得不直接指出来——这就是被很多人向往得像天堂一样的地方，Google。这位同学所描述的遭遇，跟我几年前在 Google 的实习经历如出一辙。我仍然记得，Google 的队友在旁边看着我用 Emacs，用小学老师似的口气对我说：“按 ctrl-k！”我仍然记得，在提交队友完全无法写出来的高质量代码时，被指责和嘲笑不会用 Perforce。我仍然记得，吃饭时同事们对所谓“Google 牛人”眉飞色舞的艳羡。我仍然记得，最后我一个人做出整个团队做梦都做不出来的项目的时候，有人发出沉闷的咆哮：“快——写——测——试！”……

我的这位同学也算得上本领域顶尖的专家了。如此的践踏一个专家的价值，用肤浅的标准来评判和对待他们，Google 并不是唯一一个这样的公司。我之前任职的好几个公司，或多或少都存在类似的问题。很多时候也不一定是公司管理层无端施加压力，而是程序员之间互斗的厉害，互相评判，伤害自尊。从最近 [Linus Torvalds](#) 在演讲现场公然对观众无理，你可以看出这种只关心技术，不尊重人的思潮，在程序员的社区里是非常普遍的。

后来我发现，并不是程序员故意想要藐视对方或者互相攻击，而是他们真的不明白什么叫做“尊重”，他们不知道如何说话才可以不刺伤别人。尊重他人其实是一个“技术问题”，并不是有心就可以做到的。由于这个原因，我想从心理和技术角度出发，指出这类不尊重人现象的起源，同时提出几点建议，告诉人们如何真正的尊重一个程序员。我希望这些建议对公司的管理层有借鉴意义，也希望它们能给与正在经受同样痛苦的程序员们一些精神上的鼓励。

为了建设一个互相尊重的公司文化，我认为应该注意以下几个要点。

认识和承认技术领域的历史遗留糟粕

很多不尊重人现象的起源，都是因为某些人偏执的相信某种技术就是世界上最好的，每个人都必须知道这些东西，否则他就不是一个合格的程序员。

这种现象在 Unix (Linux) 的世界尤为普遍。Unix 系统的鼓吹者们（我曾经是其中之一）喜欢到处布道，告诉你其它系统的设计有多蠢，你应该遵从 Unix 的“哲学”。他们认为 Unix 就是终极的操作系统，然而事实却是，Unix 是一个设计非常糟糕的系统。它似乎故意被设计为难学难用，容易犯错，却美其名曰“强大”，“灵活”。

眼界开阔一点的程序员都知道，Unix 的设计者其实基本不懂设计，他们并不是世界上最好的程序员，却有一点做得很成功，那就是他们很会制造宗教，煽动人们的盲从心理。Unix 设计者把自己的设计失误推在用户身上，让用户觉得学不会或者搞错了都是自己的错。

如果你对计算机科学理解到一定程度，就会发现我们其实仍然生活在计算机的石器时代。特别是软件系统，建立在一堆历史遗留的糟糕设计之上。各种蹩脚脑残的操作系统（比如 Unix，Linux），程序语言（比如 C++，JavaScript，PHP，Go），数据库，编辑器，版本控制工具，……时常困扰着我们，这就是为什么你需要那么多的所谓“经验”和“知识”。然而，很多 IT 公司不喜欢承认这一点，他们一向以来的作风是“一切都是程序员的错！”，“作为程序员，你应该知道这些！”这就造成了一种“皇帝的新装现象”——大家都不喜欢用一些设计恶劣的工具，却都怕别人嘲笑或者怀疑自己的能力，所以总是喜欢显示自己“会用”，“能学”，而没有人敢说它难用，敢指出设计者的失误。

我这个人呢，就是这种“[黑客文化](#)”的一个反例。我所受到的多元化教育，让我从这些偏激盲从，教条主义的心理里面跳了出来。每当有人因为不会某种工具或者语言来请教我时，我总是很轻松的调侃这工具的设计者，然后告诉他，你没理由知道这些破玩意儿，但其实它就是这么回事。然后我一针见血的告诉他这东西怎么回事，怎么用，是哪些设计缺陷导致了我们现在的诡异用法…… 我觉得所有的 IT 从业人员对于这些工具，都应该是这样的调侃态度。只有这样，软件行业才会得到实质性的进步，而不是被一些自虐的设计所困扰，造成思维枷锁。

总之，这是一个非常重要的“态度问题”。虽然在现阶段，我们有必要知道如何绕过一些蹩脚的工具，利用它们来完成自己的任务。然而在此同时，我们必须正视和承认这些工具的恶劣本质，而不能拿它们当教条，把什么事都怪罪于程序员。只有分清工具设计者的失误和程序员自己的失误，不把工具的设计失误怪罪于程序员，我们才能有效地尊重程序员们的智商，鼓励他们做出简单，优雅，完善的产品。

分清精髓知识和表面知识，不要太拿经验当回事

在任何领域，都只有少数知识是精髓的，另外大部分都是表面的，肤浅的，是从精髓知识衍生出来的。精髓知识和表面知识都是有用的，然而它们的分量和重要性却是不一样的。所以必须区分精髓知识和表面知识，不能混为一谈，对待它们的态度应该是不一样的。由于表面知识基本是死的，而且很容易从精髓知识推导衍生出来。我们不应该因为自己知道很多表面知识，就自以为比掌握了精髓知识的人还要强。不应该因为别人不知道某些表面知识，就以为自己高

人一等。

IT公司经常有这样的人，以为精通一些看似复杂的命令行，或者某些难用的程序语言就很了不起似的。他们如果听说你不知道某个命令的用法，那简直就像法国人不知道拿破仑，美国人不知道华盛顿一样。这些人没有发现，自己身边有些同事其实掌握着精髓的知识，他们完全有能力从自己已有的知识，衍生制造出所有这些工具，而不只是使用它们，甚至设计得更加完善和方便易用。这种能够设计制造出更好工具的人，往往身负更加重要的任务，所以他们往往会在被现有工具的用法迷惑的时候，非常谦虚的请同事帮助解决，大胆的承认自己的糊涂。

如果你是这个精通工具用法的人，切不可以把同事的谦虚请求当成可以显摆自己“资历”的时候。这同事往往真的是在“不耻下问”。他并不是搞不懂，而是根本不屑于，也没有时间去考虑这种低级问题。他的迷惑，往往来源于工具设计者的失误。他很清楚这一点，他也知道自己的技术水平其实是高于这工具的设计者的。然而为了礼貌，他经常不直接批评这工具的设计，而是谦虚的责怪自己。所以同事向你“虚心请教”，完全是为了制造一种友好融洽的气氛，这样可以节省下时间来干真正重要的事情。这种虚心并不等于他在膜拜你，承认自己的技术能力不如你。

所以正确的对待方式应该是诚恳的表示对这种迷惑的理解，并且坦率的承认工具设计上的不合理，蹩脚之处。如果你能够以这种谦和的态度，而不是自以为专家的态度，同事会高兴地从你这里“学到”他需要的，肤浅的死知识，并且记住它，避免下次再为这种无聊事来打扰你。如果你做出一副“天下只有我知道这奇技淫巧”的态度，同事往往会对你，连同这工具一起产生鄙视的情绪。他下次会照样记不住这东西的用法，然而他却再也不会来找你帮忙，而是一拖再拖。

不要自以为聪明，不要评判别人的智商和能力

在IT公司里，总是有很多人觉得自己聪明，想显示自己比别人聪明。这种人似乎随时都在评判（judge）别人，你说的任何话，不管认真的还是开玩笑的，都会被他们拿去作为评估你智商和能力的依据。

有时候你写了一些代码，自己知道时间不够，可是当时有更重要的事情要做，所以打算以后再改进。如果你提交代码时被这种人看到了，他们就会坚定地认为你一辈子只能写出那样的代码。这就是所谓“*wishful thinking*”，人只能看到他希望看到的东西。这种人随时都在希望自己比别人聪明，所以他们随时都在监听别人显得不如他聪明的时候，而对别人比他高明的时候视而不见。他们只能看到别人疏忽的时候，因为那是可以证明他们高人一等的有利证据。

当然，谁会喜欢这样的人呢，可是他们在IT公司里相当的普遍。你不敢跟他们说话，特别是不敢开玩笑，因为他们会把你稀里糊涂的玩笑话全部作为你智商低下或者经验不足的证据。你不敢问他们问题，因为他们会认为你问问题，说明你不懂！我发现具有这种心理的人，一般潜意识里都存在着自卑。他们有某些方面（包括智力在内）不如别人，所以总是找机会显得高人一等。我还没有想出可以纠正这种心理问题的有效方法，但如我上节所说，意识到整个行业，包括你仰慕的鼻祖们，其实都不懂很多东西，都是混饭吃的，是一个有效的放松这种心理的手段。

有时候我喜欢自嘲，对人说：“我们这行业的祖先做了这么多BUG来让我们修补。现在你做了一坨屎，我也做了一坨屎，我的屎貌似比你的屎香一点。”这样一来，不但显示出心理的平等和尊重，而且避免了因为谦虚而让对方产生高人一等的情绪。说真的，做这行根本不需要很高的智力，所以最好是完全放弃对人智力的判断。你比任何人更聪明，也不比他们笨。

解释高级意图，不要使用低级命令

随时都要记住，同事和下属是跟你智力相当的人。他们是通情达理的人，然而却不会简单地服从你的低级命令。像我在Google的队友的做法，就是一个很好的反面教材。其实这位Googler只是想告诉我：“删掉这行文本，然后改成这样……”就是如此一个简单的事情，然而她却故弄玄虚，不直接告诉我这个“高级意图”，而是使用非常低级的指令：“按Ctrl-k！……”语气像是在对一个不懂事的小学生说话，好像自己懂很多，别人什么都不知道似的。

有哪个Emacs用户不知道Ctrl-k是删掉一行字呢，况且你现在面对的其实是一个资深Emacs用户。我想大家都看出来这里的问题了吧。这样的低级命令不但逻辑不清楚，而且是对另一个人的智力的严重侮辱。你当我是什啊？猴子？如果这位Googler表明自己的高级意图，就会很容易在心理上和逻辑上让人接受，比如她可以说：“配置文件的这行应该删掉，改成……”

在项目管理的时候也需要注意。在让人做某一件事之前，应该先解释为什么要做这件事，以及它的重要性。这样才能让人理解，才能尊重程序员的智商。

不要期望新人向自己学习

很多IT公司喜欢把新人当初学者，期望他们“从新的起跑线出发”，向自己“学习”。比如，Google把新员工叫做“*Noogler*”（*Newbie Googler*的意思），甚至给他们发一种特殊的螺旋桨帽子，其寓意在于告诉他们，小屁孩要谦虚，要向伟大的Google学习，将来才可以飞黄腾达。



这其实是非常错误的作法，因为它完全不尊重新员工早已具备的背景知识，把自己的地位强加于他们头上。并不是你说“新的起跑线”就真的可以把人的过去都抹杀了吗。新人不了解你们的代码结构和工程方式，并不等于你们的方式就会先进一些。Google里面真的有很多值得学习的东西吗？学校的教育真的一文不值吗？其实恰恰相反。我可以坦然的说，我从自己的教授身上学会了最精髓的知识，而从Google得到的，只是一些很肤浅的，死记硬背就可以掌握的技能，而且其中有挺多其实是糟粕。我在Google做出的所有创新成果，全都是从学校获得的精髓知识的衍生物。很多PhD学生鄙视Google，就是因为Google不但自己技术平庸，反倒喜欢把自己包装成最先进的，超越其它公司和学校的，并且嚣张的期望别人向他们“学习”。

一个真正尊重人才的公司会去了解，尊重和发挥新人从外界带来的特殊技能，施展他们特有的长处，而不是一味期望他们向自己“学习”。只有这样，我们才能保持这些锐利武器的棱角，在激烈的竞争中让自己立于不败之地。如果你一味的让新人“学习”，而无视他们特有的长处，最后就不免沦为平庸。

不要以老师自居，分清“学习”和“了解”

如上文所说，IT行业的很多所谓“知识”，只不过是一些奇技淫巧，用以绕过前人设计上的失误。所以遇到别人不知道一些东西的时候，请不要以为你“教会”了别人什么东西，不要以为自己可以当老师了。以老师自居，使用一些像“跟我学”一类的语言，其实是一种居高临下，不尊重人的行为。

人们很喜欢在获得了信息的时候用“学习”这个词，然而我觉得这个词被滥用了。我们应该分清两种情况：“学习”和“了解”。前者指你通过别人的指点和自己的理解，获得了精髓的，不能轻易制造出来的知识。后者只是指你“了解”了原来不知道的一些事情。举个例子，如果有人把一件物品放在了某个你不知道的地方，你找不到，问他，然后他告诉你了。这种信息的获取，显然不叫“学习”，这种信息也不叫做“知识”。

然而，IT行业很多时候所谓的“学习”，就是类似这种情况。比如，有人写了一些代码，设计了一些框架模块。有人不知道怎么用，然后有人告诉他了。很多人把这种情况称为“学习”，这其实是对人的不尊重。这跟有人告诉你他把东西放在哪里了，是同样性质的。这样的代码和设计，我也可以做，甚至做得更好，凭什么你说我在向你学习呢？我只是了解了一下而已。

所谓学习，必须是更加高级的知识和技能，必须有一种“有收获”，“有提高”的感觉。简单的信息获取不能叫做“学习”，只能叫做“了解”。分清“了解”和“学习”，不以老师自居，是尊重人的一个重要表现。

明确自己的要求，不要使用指责的语气

有些人很怪异，他根本没告诉过你他想要什么，有什么特别的要求，可他潜意识里假设已经告诉你了。到了后来，他发现你的作法不符合要求，于是严厉指责你没有按照他“心目中的要求”办事。这种现象不止限于程序员，而且包括日常生活中的普通人。举个例子，我妈就是这种人的典型，所以我以前在家生活经常很辛苦。她心目中有一套“正确”的做事方式，如果你没猜出来就会挨骂。你为了避免挨骂，干脆什么事都不要做，然后她又会说你懒，所以你就左右不是人：)

IT公司里面也有挺多这样的人，他们假设有些信息他已经告诉你了，而其实根本没告诉你。到了后来，他们开始指责你没有按照要求做事。有些极其奇葩的公司，里面的程序员不但喜欢以老师自居，而且他们“传授”你“知识”的主要方式是指责。他们事先不告诉你任何规则，然后只在你违反的时候来责备你。我曾经在这样一个公司待过，名字就不提了。

现在举一个具体的场景例子：

A: 你push到master了？

B: 是啊？怎么了？

A: 不准push到master！只能用pull request！

B: 可是你们之前没告诉过我啊……

A: 现在你知道了？！

注意到了吗？这不是一个技术问题，而是一个礼节（etiquette）问题。你没有事先告诉别人一些规则，就不该用怪罪的语气来对人说话，况且你的规则还不一定总是对的。所以我现在提醒各位IT公司，在技术上的某些特殊要求必须事先提出来，确保程序员知道并且理解。如果没有事先提出，就不要怪别人没按要求做，因为这是非常伤害人自尊的作法。其实，在任何时候都不应该使用指责的语气，它不但对解决问题没有任何正面作用，而且会恶化人际关系，最终导致更加严重的后果。

程序员的工作量不可用时间衡量

很多IT公司管理层不懂得如何估算程序员的工作量，所以用他们坐在自己位置上工作的时间来估算。如果你能力很强，在很短的时间内把最困难的问题解决了，接下来他们不会让你闲着，而会让你做另外一些很低级的活。这是很不合理的作法。打个比方，能力强的员工就像一辆F1赛车，马力和速度是其他人的几十倍。当然，普通人需要很长时间才能解决，甚至根本没法解决的问题，到他手里很快就化解掉了。这就像一辆F1赛车，眨眼工夫就跑完了别人需要很久的路程。如果你用时间来衡量工作量，那么这辆赛车跑完全程只需要很短时间，所以你算出来的工时量就比普通车子小很多。你能因此说赛车工作不够努力，要他快马再加鞭吗？这显然是不对的。

物理定律是这样：能量 = 功率 x 时间。工作量也应该是同样的计算方法。英明的，真正理解程序员的公司，就不会指望高水平的程序员不停地工作。高水平程序员由于经常能够另辟蹊径，一个就可以抵好几个甚至几十个普通程序员。他们处理的问题比常人的困难很多，费脑力多很多，当然他们需要更好的休息，保养，娱乐，……如果你让高水平的程序员太忙了，一刻都不停着，有趣有挑战性的事情做完了就让他们做一些低级无聊的事情，他们悟出这个道理之后，就会故意放慢速度，有时候明明很快做完了也会说没做完。与其这样，不如只期望他们工作短一点的时间，把事情做完就可以。

当然这并不是说初级的程序员就应该过量工作。编程是一项艰苦的脑力活动，超时超量的工作再加上压力，只会带来效率的低下，质量的降低。

不要让其他人修补自己的BUG

这个我已经在一篇专门的[文章](#)里讨论过。让一个程序员修补另一个程序员的BUG，不但是效率低下，而且是不尊重程序员个人价值的作法，应该尽量避免。

在软件行业，经常看到有的公司管理让一个人修补另一个人代码里的BUG。有时候有人写了一段代码，扔出来不管了，然后公司管理让其他工程师来修复它。我想告诉你们，这种方法会很失败。

首先，让一个人修复另一个人的BUG，是不尊重工程师个人技术的表现。久而久之会降低工程师的工作积极性，以至于失去有价值的员工。代码是人用心写出来的作品，就像艺术家的作品一样，它的质量牵挂着一个人的人格和尊严。如果一个人A写了代码，自己都不想修复里面的BUG，那说明A自己都认为他自己的代码是垃圾，不可救药。如果让另一个人B来修复A代码里的BUG，就相当于让B来收拾其他人丢下的垃圾。可想而知，B在公司的眼里是什么样的地位，受到什么样的尊重。

其次，让一个人修复另一个人的BUG，是效率非常低下的作法。每个人都有自己写代码的风格和技巧，代码里面包含了一个个人的思维方式。人很难不经解释理解别人的思想，所以不管这两人的编程技术高下，都会比较难理解。不能理解别人的代码，不能说明这人编程技术的任何方面。所以让一个人修补另一个人的BUG，无论这人技术多么高明，都会导致效率低下。有时候技术越是高的人，修补别人的BUG效率越是低，因为这人根本就写不出来如此糟糕的代码，所以他无法理解，觉得还不如推翻重写一遍。

当我在大学里做程序设计课程助教的时候，我发现如果学生的代码出了问题，你基本是没法简单的帮他们修复的。我的水平显然比学生的高出许多，然而我却经常根本看不懂，也不想看他们的代码，更不要说修复里面的BUG。就像上面提到的，有些人自己根本不知道自己在写什么，做出一堆垃圾来。看这样的代码跟吃屎的感觉差不多。对于这样的代码，你只能跟他们说这是不正确的。至于为什么不正确，你只能让他们自己去改，或者建议他们推翻重写。也许你能指出大致的方向和思路，然而深入到具体的细节却是不可能的，而且不应该是你的职责。这就是我的教授告诉我的做法：如果代码不能运行，直接打一个叉，不用解释，不用推敲，等他们自己把程序改好，或者实在没办法，来office hours找你，向你解释他们的思想。

如果你明白我在说什么，从今天起就对自己的代码负起责任来，不要再让其它人修补自己的BUG，不要再修补其他人的BUG。如果有人离开公司，必须要有人修补他遗留下来的BUG，那么说话应该特别特别的小心。你必须指出需要他帮忙的特殊原因，强调这件事本来不是他的错，本来是不应该他来做的，但是有人走了，没有办法，并且诚恳的为此类事情的发生表示歉意。只有这样，程序员才会心甘情愿的在这种特殊关头，修补另外一个人的BUG。

不要嚷着要别人写测试

在很多程序员的脑子里，所谓的“流程”和“测试”，比真正解决问题的代码还重要。他们跟你说起这些，那真的叫正儿八经，义正言辞啊！所以有时候你很迷惑，这些人除了遵守这些按部就班的规矩，还知道些什么。大概没有能力的人

都喜欢追究各种规矩吧，这样可以显得自己“没有功劳有苦劳”。这些人自己写的代码很平庸，不知道如何简单有效地解决困难的问题，却喜欢在别人提交代码让他review的时候叫喊：“测试很重要！覆盖很重要！你要再加一些测试才能通过我的review！”

本来code review是让他们帮忙发现可能存在的问题，有些人却仿佛把它作为评判（judge）其他人能力，经验，甚至智商的机会。他们根本不明白别人代码的实质价值，就知道以一些表面现象来判断。我在Google实习，最后提交了质量和难度都非常高的代码，然而一些完全没能力写出这样代码的人，不但没表示出最基本的肯定，反而发出沉闷的咆哮：“快——写——测——试！”你觉得我会高兴吗？

我并不否认测试的用处，然而很多人提起这些事情时候，语气和态度是非常不尊重，让人反感的。这些人不但没有为解决问题作出任何实质贡献，当有人提交解决方案的时候，他们没有表达对真正做出贡献的人的尊重和肯定，反而指责别人没写测试。好像比他高明的人解决了问题，他反倒才是那个有发言权的，可以评判你的代码质量似的：“我管你代码写得多好，我完全没能力写出来，但你没写测试就是不够专业。你懂不懂测试的重要性啊，还做程序员！”

人际交往的问题经常不在于你说了什么，而在于你是怎么说的。所以我的意思并不是说你不该建议写测试，然而建议就该有建议的语气和态度。因为你没有做实际的工作，所以一些礼貌用语，比如“请”，“可不可以”……是必须的。经常有人说话不注意语气和态度，让人反感，却以自己是工程师，不善于跟人说话为借口。永远要记住，你没有做事，说话就应该委婉，切不可使用光秃秃的祈使句，说得好像这事别人非做不可，不做就是不懂规矩一样。

礼貌的语言，跟人的职业完全没有关系。身为工程师，完全不能作为说话不礼貌的借口。

关于Git的礼节

Git是现在最流行的代码版本控制工具。用外行话说，Git就是一个代码的“仓库”或者“保管”，这样很多人修改了代码之后，可以知道是谁改了哪一块。其实不管什么工具，不管是编辑器，程序语言，还是版本控制工具，比起程序员的核心思想来，都是次要的东西，都是起辅助作用的。可是Git这工具似乎特别惹人恼火。

Git并不像很多人吹嘘的那么好用，其中有明显的蹩脚设计。跟Unix的传统一脉相承，Git没有一个良好的包装，设计者把自己的内部实现细节无情地泄露给了用户，让用户需要琢磨者设计者内部到底怎么实现的，否则很多时候不知道该怎么办。用户被迫需要记住挺多稀奇古怪的命令，而且命令行的设计也不怎么合理，有时候你需要加-f之类的参数，各个参数的位置可能不一致，而且加了还不一定能起到你期望的效果。各种奇怪的现象，比如“head detached”，都强迫用户去了解它内部是怎么设计的。随着Git版本的更新，新的功能和命令不断地增加，后来你终于看到命令行里出现了foreach，才发现它的命令行就快变成一个（劣质的）程序语言。如果你了解[ydiff](#)的设计思想，就会发现Git之类基于文本的版本控制工具，其实属于古代的东西。然而很多人把Git奉为神圣，就因为它是Linus Torvalds设计的。

Git最让人恼火的地方并不是它用起来麻烦，而是它的“资深用户”们居高临下的态度给你造成的心灵阴影。好些人因为自己“精通Git”就以为高人一等，摆出一副专家的态度。随着用户的增加，Git最初的设计越来越被发现不够用，所以一些约定俗成的规则似乎越来越多，可以写成一本书！跟Unix的传统一脉相承，Git给你很多可以把自己套牢的“机制”，到时候出了问题就怪你自己不知道。所以你就经常听有人煞有介事的说：“并不是Git允许你这么做，你就可以这么做的！Unix的哲学是不阻止傻人做傻事……”如果你提交代码时不知道Git用户一些约定俗成的规则，就会有人嚷嚷：“rebase了再提交！”“不要push到master！”“不要merge！”“squash commits！”如果你不会用git submodule之类的东西，有人可能还会鄙视你，说：“你应该知道这些！”

打个比方，这样的嚷嚷给人的感觉是，你得了奥运会金牌之后，把练习用的器材还回到器材保管科，结果管理员对你大吼：“这个放这边！那个放那边！懂不懂规矩啊你？”看出来问题了吗？程序员提交了有高价值的代码（奥运金牌），结果被一些自认为Git用的很熟的人（器材保管员）厉声呵斥。

一个尊重程序员的公司文化，就应该把程序员作为运动健将，把程序员的代码放在尊贵的地位。其它的工具，都应该像器材保管科一样。我们尊重这些器材保管员，然而如果运动员们不懂你制定的器材摆放规矩，也应该表示出尊重和理解，说话应该和气有礼貌，不应该骑到他们头上。所以，对于Git的一些命令和用法，我建议大家向新手介绍时，这样开场：“你本来不该知道这些的，可是现在我们没有更好的工具，所以得这样弄一下……”

所谓“人为错误”

昨天是一个让人悲哀的日子。旧金山湾区主要的上下班交通工具Caltrain，在24小时之内发生三次事故，撞死三人。其中一次事故发生在Menlo Park，一辆汽车被困在铁轨上，因为被前后的车辆堵塞而无法逃避，终于被飞驰而来的列车撞成一堆废铁。开车人被消防队员从残骸里切割出来，送往医院后不久死亡。（[新闻视频](#)）



我为生命的殒灭而悲哀，然而让我更加悲哀的是，每当这样的事故发生，总有人指责说是“人为错误”。比如，Twitter上有人说这事故是因为死者没有遵守交通规则，才导致自己的汽车被困在铁轨之上，所以她死的活该。

 **Joe Burgoon** @Keekers16 · 2h
@KTVU why people should never sit on the tracks. People trap themselves by putting themselves in that situation. Just wait. What a shame

◀ ▶ ★ 2 ⋮

真的是因为她不遵守交通规则吗？真的有人愿意把车停在铁轨上等死吗？也许是这规则没法遵守，或者设计得让人很容易“违反”呢？

首先，规则必须要让人理解，切实可行，才能叫做规则。



但是请看看铁轨交叉路口上的指令：“不要停在铁轨上（DO NOT STOP ON TRACKS）”，“保持路口畅通（KEEP CLEAR）”。我也不想停在铁轨上啊，可是我刚开到铁轨上，前面的车就停下来了，过不去怎么办？另外什么叫做clear？一定要等到路口里面完全没有车才可以进去吗？如果路口里面虽然有车，然而它们都以每小时30英里的速度行驶？这时我还该停下来吗？如果前面车的速度不到每小时5英里呢？如果前面车辆貌似很快，结果我一进路口它就慢下来了怎么办？

如果“不要停在铁轨上”的指令我想遵守都不可能，如果连clear这个单词都定义不清楚，这还叫什么“交通规则”呢？既然规则都不清楚，又怎么能责怪别人不遵守？我要有多么高的预知未来的能力，才能猜得到前面的车会不会正好在我开到铁轨上的时候停下来，把我堵在铁轨上呢？也许你已经看出来了，这其实不是开车人的错误，它最多算一个“判断失误”。每个人都可能在那种模棱两可的情况下发生判断失误，因为你没法知道前面的车会怎样运动。记者在现场采访的几个开车人都说：“过那个路口要极度小心，因为你不知道前面的车会怎么样走。”

如果你仔细看看卫星图，就会发现铁轨前方的道路狭窄，而且不远处有一个红绿灯。如果这个红绿灯变红，那么就有可能把直到铁轨处的车辆全都叫停。如果你熟悉湾区的道路，就知道红灯处是82号公路（El Camino Real），上那条路的红灯经常等很久。也就是说，可能有很多车在那里等红灯，一直到铁轨的地方！



如果你再仔细一点，用Google Map的street view去实地看一下那个路口，就会发现，地面上的“KEEP CLEAR”字样，其实是用来给被堵在铁轨上的车预留后路的。然后你就发现，如果后面的车不遵守KEEP CLEAR的指令，那么它们就会断掉铁轨上的车的退路。所以，其实不是铁轨上的车自己等死，而是后面那些不遵守KEEP CLEAR指令的车，把它逼上了绝路。然而就像我之前提到的，想要遵守KEEP CLEAR又是很模棱两可的事情，后面的车有可能以为你过得去，所以才跟上的。所以你死了，不能怪火车，不能怪你自己，不能怪前面的车，还不能怪后面的车！怪谁呢？只能怪路口的“设计”！

这种路口交通规则还有一个致命的特征，那就是后果的严重性不明显，人不会敏锐的感觉到犯了错误的结果是车毁人亡。一般人都不闯红灯，因为很显然，如果你红灯不停就会被另一个方向的车撞上。可是违反这铁道路口的规则，后果不是立显的，有可能什么事也没有，也有可能呆在那里几分钟之后才出事，到时候想逃都逃不了。这就像把活青蛙放进冷水里，然后慢火加热一样，它不会立即被烫得跳出来，而会死在里面！等你慢慢的开到铁轨上，才发现前面的车不走了（因为更前面路口亮了红灯），后面的车又抵上来。过一会儿，当当当，栏杆放下来，火车来了…… 你这是在设陷阱诱捕野生动物吗？

如此容易出现的失误（甚至不叫做自己的失误），真的值得一个人用生命来偿还吗？按照这样的逻辑，我就可以把地雷埋在大街上，插上标志牌说：“下面有地雷，不要踩！”如果你踩了，那我就可以怪你没遵守规则，自己找死！

如果你回头看看历史就会发现，Caltrain几乎每个月都会撞上至少一辆汽车，所以这次的事故绝不是偶然，它有更深层的原因。上一个月，我乘坐的一列Caltrain，就因为前面一趟列车撞上了汽车而延误了好几个小时。当时我就在Twitter上看到有人责备开车的人，说他脑子秀逗了，不该把车停在铁轨上。当时我就在Twitter上警告@Caltrain，说你们应该仔细分析一下这个交叉路口的设计，也许是因为设计有问题。没有人回应我。这次出了三条人命，交叉路口的设计问题才终于受到了重视。

出了人命的大事故，也许能唤醒人们一点理智，认识到所谓的“人为错误”，其实在很多时候是设计错误。在这个例子中，交叉路口的设计是不合理的。一旦你因为判断失误把车开进去了，就有可能出现无路可逃，车毁人亡的局面。然而很多生活中的设计失误所引发的“人为错误”都是不致命的，有点像慢性毒药。这种貌似无关痛痒的设计错误，更容易被忽视，它们就潜伏在我们的身边。

在我所在的软件行业里，就有很多这样的设计错误。在我看来，整个软件行业基本就是建立在一堆堆的设计失误之上。做程序员如此困难和辛苦，大部分原因就是因为软件系统里面积累了大量前人的设计失误，所以我们需要做大量的工作来弥补或者绕过。举个例子，Unix/Linux操作系统就是一个重大的设计失误。Unix系统的命令行，系统API，

各种工具程序，编辑器，程序语言（C，C++等），设计其实都很糟糕。很多工具程序似乎故意设计得晦涩难用，让人摸不着头脑，需要大量时间学习，而且容易出错。出错之后难以发现，难以弥补。

然而一般程序员都没有意识到这里面的设计错误，知道了也不敢指出来，他们反而喜欢显示自己死记硬背得住这些稀奇古怪的规则。这就导致了软件行业的“皇帝的新装现象”——没有人敢说工具的设计有毛病，因为如果说出来，别人就会认为你在抱怨，那你不是经验不足，就是能力不行。这就像你不敢说皇帝没穿衣服，否则别人就会认为你就是白痴或者不称职的人！Unix系统的同盟者和后裔们（Linux，C语言，Go语言），俨然形成了这样一种霸权，他们鄙视觉得它们难用，质疑它们的设计的人。他们嘲笑这些用户为失败者，即使其实有些“用户”水平比Unix的设计者还要高。久而久之，他们封住了人们的嘴，让人误以为难用的东西就是好的。

我体会很深的一个例子就是Git版本控制工具。有人很把这种东西当回事，引以为豪记得住如何用一些稀奇古怪的Git命令（比如git rebase, git submodule之类）。好像自己知道了这些就真的是某种专家一样，每当遇到不会用这些命令的人，都在心底默默地鄙视他们。作为一个比Git的作者还要高明的程序员，我却发现自己永远无法记住那些命令。在我看来，这些命令晦涩难懂，很有可能是因为没设计好造成的。因为如果一个东西设计好了，以我的能力是不可能不理解的。可是Linus Torvalds的名气之大，威望之高，有谁敢说：“我就是不会用你设计的破玩意儿！你把我怎么着？”

其他人的BUG

在软件行业，经常看到有的公司管理让一个人修补另一个人代码里的BUG。有时候有人写了一段代码，扔出来不管了，然后公司管理让其他工程师来修复它。我想告诉你们，这种方法会很失败。

首先，让一个人修复另一个人的BUG，是不尊重工程师个人技术的表现。久而久之会降低工程师的工作积极性，以至于失去有价值的员工。代码是人用心写出来的作品，就像艺术家的作品一样，它的质量牵挂着一个人的人格和尊严。如果一个人A写了代码，自己都不想修复里面的BUG，那说明A自己都认为他自己的代码是垃圾，不可救药。如果让另一个人B来修复A代码里的BUG，就相当于是让B来收拾其他人丢下的垃圾。可想而知，B在公司的眼里是什么样的地位，受到什么样的尊重。

其次，让一个人修复另一个人的BUG，是效率非常低下的作法。每个人都有自己写代码的风格和技巧，代码里面包含了一个个人的思维方式。人很难不经解释理解别人的思想，所以不管这两人的编程技术高下，都会比较难理解。不能理解别人的代码，不能说明这人编程技术的任何方面。所以让一个人修补另一个人的BUG，无论这人技术多么高明，都会导致效率低下。有时候技术越是高的人，修补别人的BUG效率越是低，因为这人根本就写不出来如此糟糕的代码，所以他无法理解，觉得还不如推翻重写一遍。

当我在大学里做程序设计课程助教的时候，我发现如果学生的代码出了问题，你基本是没法简单的帮他们修复的。我的水平显然比学生的高出许多，然而我却经常根本看不懂，也不想看他们的代码，更不要说修复里面的BUG。就像上面提到的，有些人自己根本不知道自己在写什么，做出一堆垃圾来。看这样的代码跟吃屎的感觉差不多。对于这样的代码，你只能跟他们说这是不正确的。至于为什么不正确，你只能让他们自己去改，或者建议他们推翻重写。也许你能指出大致的方向和思路，然而深入到具体的细节却是不可能的，而且不应该是你的职责。这就是我的教授告诉我的做法：如果代码不能运行，直接打一个叉，不用解释，不用推敲，等他们自己把程序改好，或者实在没办法，来office hours找你，向你解释他们的思想。

如果你明白我在说什么，从今天起就对自己的代码负起责任来，不要再让其它人修补自己的BUG，不要再修补其他人的BUG。如果有人离开公司，必须要有人修补他遗留下来的BUG，那么说话应该特别特别的小心。你必须指出需要他帮忙的特殊原因，强调这件事本来不是他的错，本来是不应该他来做的，但是有人走了，没有办法，并且诚恳的为此类事情的发生表示歉意。只有这样，程序员才会心甘情愿的在这种特殊关头，修补另外一个人的BUG。

创造者的思维方式

我不知道人们是怎么回事，缺乏想象力还是怎么的，所以我跟其他人对话常常遇到类似的问题。

我：A其实不怎么好。

其他人：你说A不好，难道你要我用B？

(对于政治爱好者，如果A是资本主义，B就是社会主义；如果A是美国，B就是中国，等等。对于IT人员，如果A是Unix，B就是Windows；如果A是Vim，B就是Emacs；如果A是关系式数据库，B就是NoSQL数据库，等等……)

然后呢，这人就会深信我是B的拥趸，进而产生敌意。这种对话越说越糊涂，越说越尴尬，越说我越觉得降低我的身份。

仔细分析之后，我发现了这问题的起因，其实是因为我跟其他人的思维方式是完全不同的。我总是从一个“创造者”的角度说话，而对方却站在“使用者”的角度。站的高度不同，当然就没法沟通，鸡同鸭讲。

创造者说“A其实不怎么好”，他的意思往往不是说你应该去“用”别的什么东西。他的意思其实是，A不怎么好，我可以把它的缺点去掉，“创造”一个更好的东西。这里的区别就在于“用”和“创造”的不同。使用者说“A不好”，是无可奈何的抱怨；创造者说“A不好”，却是对改进机遇的欣喜。可惜的是，使用者永远无法理解创造者的心，创造者的喜悦在使用者的头脑里，直接被“翻译”成了抱怨。

创造者拥有使用者没有的能力，他能够随心所欲的制造出新的事物，而不带有现存事物设计的思维枷锁。创造者因此具有比使用者更高的安全感，更深的远见，更豁达的胸襟。他不容易陷入非此即彼的“宗教冲突”，他不需要选择任何一个“阵营”，因为他对这种冲突的解决方案很简单：创造一个全新的宗教，消灭掉冲突的双方 :)

小费和中国人的尊严

小费，一个尴尬的话题，一般是中国来到美国之后第一个不习惯的文化现象。在中国，吃饭理发等活动是不需要付小费的，而在美国，付小费貌似服务行业一种必须的事情。在美国的饭店吃饭，一般人会付至少 15% 的小费给服务员。

初到美国的中国人一般比较小气，想方设法省钱，当然也想在小费上省钱。有些人去饭店吃饭，不管别人服务如何周到如何有礼貌，都只付不到 15% 甚至少于 10% 的小费，吃饭之后总是有做贼一样逃之夭夭的感觉。

后来这些人发现了，苛扣小费的做法是损害中国人在美的社会形象的，所以很多人变得“大方”起来。有些人去饭店吃饭，不管服务如何都给 20% 甚至以上的小费，就算服务员不够尊重，甚至态度恶劣也一样，还生怕得罪了他们似的。殊不知，给小费太多，或者小费比例大于服务质量，不仅损害了中国人的尊严和形象，而且对社会有危害。下面我就谈谈自己的看法。

其实从理论上讲，付小费这习惯是美国社会的一种历史遗留陋习。美国餐馆雇佣服务员，只付给他们非常低的工资，以至于服务人员必须用客人付的小费来维持生活。餐馆本来应该给服务员足够的工资却没有给，到头来这负担却摊到顾客身上，不付小费还觉得自己有罪似的，这就是美国的歪理。小费的习俗使得美国服务员变得势利，以貌取人。服务员的态度，经常取决于“他认为”你吃完之后会给多少小费，即使你平时给小费很大方也无济于事。顾客与服务员之间这种不清不楚的猜测关系，搞得跟中国人吃完饭讨论该谁付账时一样难受。

在社会文明的某些方面更加先进的[英国](#)和欧洲其它国家，基本不用给小费，而且给小费被视为对服务人员的侮辱（人家的工资不低）。所以就跟美国人仍然使用落后的英制单位（英里，英寸，加伦，盎司，华氏度...）一样，小费并不是什么“高大上”的社会现象。相反，这是社会制度落后，社会福利不好，贫富分化剧烈的表现。所以作为一个具有更先进的文明来的人，付小费真的只是为了施舍，为了怜惜这些美国社会底层的服务业工作者。不仅是坐下来吃饭的餐厅，美国的很多快餐店咖啡店会在收款员柜台上放一个小筐子用来收小费，搞得就像是街上乞讨的一样。很多英国人在美国不付小费，甚至对此强烈反感，也许就是这个原因。

给大额的小费在美国有时候成为了显摆或者炫富的做法。比如有些球星会在饭店留下巨额的小费，把小票拍照，然后在网络上大肆炒作。殊不知在有修养的人看来，是非常荒谬和可鄙的做法。



哎，入乡随俗嘛。既然服务员依赖小费为生，就应该为小费做出相应质量的服务。这里的原则应该是：小费的比例应该符合所接受的服务质量。一个态度恶劣的服务员，本来就不应该继续待在他的岗位上，更不要说拿到 15% 甚至更多的小费。如果你不管服务如何都给一样的小费，其实就是在纵容这些不适合做服务工作的人，让他们继续用恶劣的态度给顾客的心灵带来阴影。同时，这也是对那些彬彬有礼，服务周到的人员的不公平。久而久之，那些向顾客显示出会心微笑的人，就会越来越少。

而且如果你给一个对你态度粗鲁甚至恶劣的服务员 15% 甚至以上的小费，实际上就是在损害自己的社会尊严。因为你心里想的其实是，“可不要得罪了服务员”，“不想有一种做贼的感觉”，“不要给中国人丢脸”…… 这些都是卑微的想法。想想美国人到中国，到秀水街买件盗版名牌衣服都要讨价还价。中国人最喜欢在外国显示自己“懂规矩”，有钱，大方。这其实是自卑的表现。你认为服务员收着高额的小费就会尊敬你吗？人家在背地里笑你呢！

之所以想写这些，是想矫正一下一些在美中国人的心理。有一天我和朋友去一个中餐馆吃饭，服务员的态度真的很不好，各种交接礼节都很粗鲁。最后付账的时候20块钱的样子，我把信用卡放在盘子里，服务员过来，很不客气的跟我说只收现金。等他走开之后我淡淡一笑，拿出22块钱放进盘子里。显然，我不会为了一两块钱斤斤计较。只给 10%

的小费，在我的意识里表示“差评”。可是朋友看到挺紧张，说：“你给的太少啦！待会儿别人会追出来要小费的……”

你可以由此可见朋友和我在尊严和社会地位上差别。在我的心里，我维护了自己（以及中国）的尊严，小小惩罚了一下态度不好的服务人员，做了一件正义的事情，然而在朋友的心里，也许我就是一个小气鬼。我可不在乎别人怎么想:-)

总结一下：

1. 美国小费的普遍性来自于落后的社会关系和严重的贫富分化。中国和英国很少有需要付小费的地方，这是社会制度在某些方面先进的表现。所以没必要在美国餐厅战战兢兢，担心自己“不懂规矩”，或者以此来审视别人。
2. 在心理上，小费应该是一种施舍（不幸的事实），本来是随便给不给都可以的。高兴就多给，不高兴就少给，没必要因为自己或者别人给了多少小费而惊诧。
3. 没有人有权利向顾客索取小费，或者直接把小费加到账单上（英国人遇到这种情况会公开的表示鄙视，中国人也应该显示出一点态度）。
4. 根据著名的 Emily Post [Etiquette](#)。美国餐厅服务员的小费一般是“税前”金额的 15-20%。很多人计算小费用“税后”的总金额，那是不对的，因为税是给政府的，不属于给餐馆的开销。
5. 对于服务态度不好的餐厅，应该适当减少小费，到10%左右，给2块钱小费，或者干脆不给。
6. 对于快餐店和咖啡店收款处的“小费筒”，大可不必放钱进去。我觉得往里面丢钱是对店员的侮辱。不过如果我用现金，一般会把找回来的硬币顺手丢进去，因为我不喜欢带硬币在身上。

恶评《星际穿越》



(Spoiler 警告，本文含有大量具体情节！)

上周末受朋友之邀，去看了红极一时的《星际穿越》（Interstellar）。因为是在首映的第二天，人多不说，票价也贵一些。可惜开演没多久，我就发现这片子简直跟新闻联播一般味同嚼蜡，好不容易熬过那漫长的三个小时，到后面几乎是睁一只眼闭一只眼睡过了。

回来之后却发现人们对它好评如潮，IMDB 评价居然达到 9.0，真是让我匪夷所思。当我正庆幸自己的欣赏水平还没被好莱坞颠覆的时候，惊闻公司小头目决定组织一次 team building 活动，其内容为“八个男人集体观看 Interstellar”。我哭笑不得，遂决定请假一天 :P

我为什么认为《星际穿越》是烂片呢？原因如下：

1. Geeky，呆板。导演似乎不认为自己是个导演，而是史蒂芬·霍金，所以通篇都在炫耀自己懂得多少宇宙学原理。他忘记了电影的主要作用是娱乐，电影最重要的价值在于它的故事性和戏剧性。物理学，宇宙学，飞船术语，看上去貌似很酷，但其实让这电影成为了让人犯困的课堂。自称喜欢这部电影的人，往往在于他们可以在事后炫耀自己懂得多少黑洞的原理。
2. 说教和煽情。“爱是可以穿越星际的力量，你感觉到了吗？”这屁话前不沾村后不着店的冒出来，唐僧了一遍又一遍，恐怕杨过和小龙女听了都会吐吧;) 爱是美好而重要的，然而它的力量和作用范围是有限的，不必如此牵强和夸张吧？而且这种感情的东西最好是从情节的点滴表现出来（比较一下好片《Life Is Beautiful》或者《西雅图不眠夜》），这样直白的平铺直叙，就索然无味，起不到效果了。这体现了好莱坞一贯以来的作风，跟我党一样，喜欢对民众进行“思想品德教育”，把观众当低龄甚至弱智儿童对待。这种说教倾向在 Disney 的片子里面最为明显，然而它也贯穿了许多本来旨在给成人看的好莱坞电影。
3. 主题肤浅，片面追求特效。一部电影应该有一个主题，所有的情节，特效，都应该是为主题服务的。这部电影的主题貌似是关于“爱”，然而它用于支持主题的情节非常牵强扯淡，以至于完全没有力量支持它所期望的主题。“5, 4, 3, 2, 1, 点火！”，炫酷的飞船和宇航服造型，主人公在船舱中的各种装模作样的操作，…… 好像很酷的样子？可是这一切都没法掩盖剧情的苍白无味。相反，剧情的肤浅让这一切的高科技模型，都显得像小孩子玩过家家的道具一样可笑。我看着那些飞船和宇航服，越看越像是泡沫塑料做的。“人间大炮，一级准备……”好莱坞，多拍点给成人看的片子好不好？
4. 故作深沉，过于安静，感觉成本很低。片中有大量独白，有好几次就两个人在那里“说真心话”。这种对白如果有好的铺垫，少量出现的话会有效果。然而本片没有做好铺垫，所以这些对白突然出现的时候，让人感觉不自然，不真实，故作深沉，煽情，冗长。搞得整个放映厅里鸦雀无声，观众面面相觑，大气都不敢出，感觉不舒服不自在。片中很多外太空画面是完全的寂静，没有背景音效，让人感觉是不是拍片时资金不足，请不起人来做音乐。整个片子感觉非常“低成本”，低到了吝啬的地步，也许几个人在一个房间里聊聊天，然后用电脑做做特效，就可以拍摄完成。
5. 严重的“常理漏洞”。有物理民科朋友看了此片之后兴奋的叫好：片子里的物理学，黑洞原理，居然没有破绽！可是这位朋友虽然了解高深的物理学原理，却缺乏一种重要的，普通人都明白的道理，叫做“常理”。常理决定了特定的人在特定的时候该说什么话，该有什么行为。这片中的人物有一些非常严重的，违反常理的漏洞，让人哭笑不得。其中一个就是，快到穿越 wormhole 的时候，副驾驶拿起一张纸，开始循循善诱的给机长（主角）讲解

wormhole 是什么，以及它的工作原理，仿佛机长在这次目的为“穿越 wormhole”的任务发射升空之前，完全不明白 wormhole 是什么似的。这就像是开着战斗机到了航母面前，才开始了解它的跑道长度一样！

我才不管你的 wormhole 理论讲解得有多么透彻多么通俗易懂，因为这样的对话根本就不应该在那个时刻，那个地点，在那些人物之间出现。能犯这样的错误，其最终原因还是因为好莱坞把观众当小孩，喜欢说教。这一番科普，俨然是导演安排讲给观众听的，而不是讲给机长听的。银幕前的小朋友们来看那，wormhole 就是这样的，就像一张纸，被折起来了哦，咔嚓…… 某些人看电影总是很“理性”的跟踪其中所描述的“高深理论”，如果发现没有破绽，学到了东西，就觉得是好片。只有当你跳出“学物理，理解宇宙原理”这一思维圈套，才会明白这些人物的行为是多么的荒唐，不合情理。剧中人物还有很多类似的诡异对话和行为，有待大家进一步发掘。

6. 又臭又长。故事很烂就算了，如果只有一两个小时还可以忍，可是此片居然有三个小时！所以真是忍无可忍，必须喷了！

最后，我对观看此片之后深入探讨第一个星球上的大水如何可以弄死人以及最后女宇航员去了哪个星球之类“学术问题”的朋友表示崇高的敬意和深切的慰问！我对此类问题统一的终极答案为：导演的安排！

关系式模型的实质

每当我调侃关系式数据库，就会有人说，SQL 和关系式数据库的设计偏离了 E. F. Codd 最初的关系式理论，关系式理论和关系式模型本身还是很先进的，只不过实现的时候被人搞砸了。

我很悲哀，因为如果你看透了关系式理论（模型/代数）本身，就会发现关系式数据库的问题是根源性的：关系式理论本身就是空洞而虚浮的，它是一个披着“数学”外衣的噱头，是潜伏在大学计算机系课程里几十年之久的无稽之谈。

人们总是喜欢制造这些概念上的壁垒，用以防止自己的理论受到攻击。把过错推到 SQL 身上，说 SQL 没有忠实的实现关系式理论的精髓，是关系式数据库领域常见的托词，用以掩盖其本质上的空洞。在下面的讨论里为了方便，我会使用少量 SQL 来表示关系式模型里面对应的概念，但这并不削弱我对关系式模型的批评，因为它们表示的是关系式模型里面等价的核心概念。

关系式模型与数据结构

很多人把关系式理论和数据结构（data structure）独立开来，认为它们是完全不同的领域。而其实数据结构的理论可以很容易的解释所有关系式数据库里面的操作。

关系式模型的每一个“关系”或者“行”（row），表示的不过是一个普通语言里的“结构”，就像 C 语言的 struct，或者 Java 的 class。一个表（table），不过就是某种结构的数组（比如 Student[]）。举个例子，以下 SQL 语句构造的表：

```
CREATE TABLE Students ( sid CHAR(20),
                        name CHAR(20),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL )
```

其实相当于以下 C 语言的结构数组：

```
struct student {
    char* sid;
    char* name;
    char* login;
    int age;
    double gpa;
}
```

每一个数据库的“key”，本质和 C 语言的指针是一回事，就像 char* p。所谓“join”操作，就是对指针的“访问”（dereference），得到指针指向的对象，就像 C 语言里写 *p。在实现上，join 跟指针访问有一定差别，因为 join 需要用软件查“索引”（index），所以它比指针访问要慢很多。

数据库所谓的查询（query），本质上就是函数式语言里面的 filter, map 等操作。只不过关系式代数更加笨拙，组合能力很弱。比如，以下的 SQL 语句

```
SELECT Book.title
  FROM Book
 WHERE price > 100
```

表达的东西相当于以下的 Lisp 代码：

```
(map book-title
      (filter (lambda (b) (> (book-price b) 100)) Book))
```

但 SQL 的嵌套组合能力和一致性都要比 Lisp 差很多。很多你认为应该自然可以表达的查询，SQL 表达不了，折腾很久才发现得用很蹩脚的方式表达。嵌套的查询经常是个问题，需要扩展 SQL 的语法才能实现，而 Lisp 天生可以优雅地表达任意的嵌套和组合。

不可否认，某些 SQL 底层实现对基本查询的实现或许更加高效，然而其实 Lisp 的底层运行系统也可以采用类似的高效实现。我们不应该把“底层实现”和“上层概念”混淆起来。

一个糟糕的概念可以被实现得很快，然而概念本身仍然是糟糕的，用起来痛苦，莫名其妙。一个优雅的设计也许被实现得很低效很慢，但聪明人看到了它概念上的优势，可以改变底层实现，做出很高效的系统。实际上已经有人实现了这样的数据库系统，它用类似这里的 Lisp 方式来表达查询。

关系式模型的局限性

所以关系式模型所能表达的东西，不会超过普通数据结构，然而关系式模型却有比数据结构更多的局限。由于“行”只

能有固定的宽度，所以导致了你没法在里面放进任何“变长”的对象。比如，如果你有一个动态长度的数组，那你是不能把它放在一个行里的。你需要把数组拿出来，旋转 90 度，做成另一个表 B。从原来的表 A，用一个“foreign key”指向 B。更傻的是，在表 B 的每一行，这个 key 都要被重复一次。数组有多长，这个 key 就需要重复多少次，占用大量不必要的空间。这种从数据结构角度看来极其愚蠢的做法，在数据库领域却被起了一个高深莫测的名字，叫做“normalization”；）

类似这样的操作，组合在一起，导致了关系式数据库的繁琐。说白了，normalization 就是在手动做一些比 C 语言的手动内存管理还要低级的工作。连 C 这么低级的语言，都允许你在结构里面嵌套数组，而在关系式模型里面你却不能。许多宝贵的人力，耗费在构造，释放，连接这些“中间表格”的工作中。

另外有一些人（比如这篇[文章](#)）采用五十步笑一百步的做法，通过关系式模型与其它数据模型（Data Model，比如网状模型之类）的对比，以支持关系式模型存在的必要性。你说我关系式模型不好，看哪，还有更差的！如果你理解了这小节的所有细节就会发现，你完全可以使用基本的数据结构，表示关系式模型以及被它所“超越”的那些数据模型。这些所谓“数据模型”，其实全都是故弄玄虚，无中生有。

数据模型可以完全被普通的数据结构所表示，然而它们却不可能简单而完整的表达数据结构带有的信息。这些数据模型之所以流行，是因为它们让人误以为知道了所谓的“一对一”，“一对多”等冠冕堂皇的概念，就可以取代设计数据结构所需要的技能。所以我认为数据模型本身就属于技术上的“减肥药”，告诉你要吃好几个疗程才会见效，最后还是不见效，那肯定是你自己什么地方操作错了；）

与其寄希望于这些贴着精美“数学”标签的减肥药，你不如去隔壁二流大学旁听一堂基础的数据结构课程；）

NoSQL

所以 E. F. Codd 的关系式理论（关系式模型，关系式代数）是这一切麻烦的祸根，而 SQL 只是它的一个小喽啰。人们用数据库遇到麻烦，一般都拿小喽啰开刀，骂 SQL，却给关系式理论制造各种托词。他们畏惧“代数”这样的术语。一个概念被冠以“关系式代数”这样的称呼，你是不敢骂它的，否则别人会说你不懂，学识太浅，理解不了“数学”；）

关系式理论和它的小喽啰 SQL 所引起的一系列无须有的问题，终究引发了所谓“NoSQL 运动”。很多人认为 NoSQL 是划时代的革命，然而在我看来，它最多可以被称为“不再愚蠢”。大多数 NoSQL 数据库的设计者并没有看到上述的问题，或者他们其实也想故弄玄虚，所以 NoSQL 数据库的设计，并没有完全摆脱关系式模型以及 SQL 所带来的思维枷锁。

最早试图冲破关系式模型和 SQL 限制的一种技术，叫做“列存储数据库”（column-based database），比如 Vertica, HBase 等。这种数据库其实就是针对了我刚刚提到的，关系式模型无法保存变长结构的问题。它们所谓的“列压缩”，其实不过是在“行结构”里面增加了对“数组”的表示和实现。很显然，一个数组放在存储设备里，需要一个字段来表示它的长度 N，剩下的空间依次保存每个元素。这样你只需要一个 key 就可以找到数组里所有的元素，而不需要进行 normalization，把 key 重复 N 遍。

这是每个初学编程的人存储数组的时候都会想到的做法，却被关系式模型排除在外。列存储数据库只不过是纠正了一个历史遗留的愚蠢错误，却把自己说成是重大的突破。甚至很多列存储数据库也没有看到这一实质。它们经常存在一些无端的限制，比如给变长数组的嵌套层数作出限制，等等。所以，列存储数据库其实也没能完全逃脱关系式数据库的思想枷锁。如此明显的事情，数据库专家们最开头恁是看不到。到后来改来改去，改得六成对，还美其名曰“优化”和“压缩”。

最新的一些 NoSQL 数据库，比如 Neo4j, MongoDB 等，部分的改善了 SQL 的表达力问题。Neo4j 设计了个古怪的查询语言叫 Cypher，不但语法古怪，表达力弱，而且效率出奇的低，以至于几乎任何实际的操作，你都必须使用 Java 写“扩展”（extension）来完成。MongoDB 等使用 JSON 来表示查询，本质就是手写编译器里的语法树（AST），不直观又容易出错。

现在看来，数据库的主要问题，其实是语言设计的问题。NoSQL 数据库的领域，由于缺乏负责的程序语言专家，而且由于利益驱使，急功近利，所以会在很长一段时间之内处于混沌之中，给使用者造成痛苦。

其实数据库的问题哪有那么困难，它跟“远过程调用”（RPC）没什么两样。只要你有一个程序语言，几乎任何程序语言，你就可以发送这语言的代码到一个“数据服务器”。服务器接受并执行这代码，对数据进行索引，查询和重构，最后返回结果给客户端。如果你看清了 SQL 的实质，就会发现这样的“过程式设计”并不会损失 SQL 的“描述”能力。反而由于过程式语言的简单，直接和普遍，使得开发效率大大提高。NoSQL 数据库比起 SQL 和关系式数据库存在优势，也就是因为它们在朦胧中朝着这个“RPC”的方向发展。

有些人说你这样直接编程不好，因为外存的管理，索引数据结构，都是很容易出错的代码，还是不如用数据库。可是谁告诉你一定要自己写外存管理和索引代码呢？你完全可以使用经过千锤百炼的代码库，把它们放在服务器上面做一个“存储索引系统”，你的“查询代码”只需要发送过去调用这些代码库就可以了。

所以到现在，我的脑子里早已不存在“数据库”，“关系式”，“NoSQL”这样的概念，因为它们带来的更多是困扰，它们把本来简单的问题复杂化。在我的脑子里，只有更通用而简单的数据结构，以及针对它们的高效存储处理方式。

对 Go 语言的综合评价

以前写过一些对 Go 语言的负面评价。现在看来，虽然那些评价大部分属实，然而却由于言辞激烈，没有点明具体问题，难以让某些人信服。在经过几个月实际使用 Go 来构造网站之后，我觉得现在是时候对它作一些更加“客观”的评价了。

定位和优点

Go 比起 C 和 C++ 确实有它的优点，这是很显然的事情。它比起 Java 也有少数优点，然而相对而言更多是不足之处。所以我对 Go 的偏好在比 Java 稍低一点的位置。

Go 语言比起 C, C++ 的强项，当然是它的简单性和垃圾回收。由于 C 和 C++ 的设计有很多历史遗留问题，所以 Go 看起来确实更加优雅和简单。比起那些大量使用设计模式的 Java 代码，Go 语言的代码也似乎更简单一些。另外，Go 的垃圾回收机制比起 C 和 C++ 的全手动内存管理来说，大大降低了程序员的头脑负担。

但是请注意，这里的所谓“优点”都是相对于 C 之类语言而言的。如果比起另外的一些语言，Go 的这种优点也许就很微不足道，甚至是历史的倒退了。

语法

Go 的简单性体现在它的语法和语义的某些方面。Go 的语法比 C 要稍好一些，有少数比 Java 更加方便的设计，然而却也有“倒退”的地方。而且这些倒退还不被很多人认为是倒退，反而认为是进步。我现在举出暂时能想得起来的几个方面：

- **进步**：Go 有语法支持一种类似 struct literal 的构造，比如你可以写这样的代码来构造一个 S struct：

```
S { x: 1, y: 2, }
```

这比起 Java 只能用构造函数来创建对象是一个不错的方便性上的改进。这些东西可能借鉴于 JavaScript 等语言的设计。

- **倒退**：类型放在变量后面，却没有分隔符。如果变量和它的类型写成像 Pascal 那样的，比如 x : int，那也许还好。然而 Go 的写法却是 x int，没有那个冒号，而且允许使用 x, y int 这样的写法。这种语法跟 var，函数参数组合在一起之后，就产生了扰乱视线的效果。比如你可以写一个函数是这样开头的：

```
func foo(s string, x, y, z int, c bool) {  
    ...  
}
```

注意 x, y, z 那个位置，其实是很混淆的。因为看见 x 的时候我不能立即从后面那个符号 (, y) 看到它是什么类型。所以在 Go 里面我推荐的写法是把 x 和 y 完全分开，就像 C 和 Java 那样，不过类型写在后面：

```
func foo(s string, x int, y int, z int, c bool) {  
    ...  
}
```

这样一来就比较清晰了，虽然我愿意再多写一些冒号。每一个参数都是“名字 类型”的格式，所以我一眼就看到 x 是 int。虽然多打几个字，然而节省的是“眼球 parse 代码”的开销。

- **倒退**：类型语法。Go 使用像 []string 这样的语法来表示类型。很多人说这种语法非常“一致”，但经过一段时间我却没有发现他们所谓的一致性在哪里。其实这样的语法很难读，因为类型的各部分之间没有明确的分隔标识符，如果和其他一些符号，比如 * 搭配在一起，你就需要知道一些优先级规则，然后费比较大的功夫去做“眼球 parse”。比如，在 Go 代码里你经常看到 []*Struct 这样的类型，注意 *Struct 要先结合在一起，再作为 [] 的“类型参数”。这种语法缺乏足够的分隔符作为阅读的“边界信号”，一旦后面的类型变得复杂，就很难阅读了。比如，你可以有 *[]*Struct 或者 *[]*pkg.Struct 这样的类型。所以这其实还不如像 C++ 的 vector<struct*> 这样的写法，也就更不如 Java 或者 Typed Racket 的类型写法来得清晰和简单。
- **倒退**：过度地“语法重载”，比如 switch, for 等关键字。Go 的 switch 关键字其实包含了两种不同的东西。它可以是 C 里面的普通的 switch (Scheme 的 case)，也可以是像 Scheme 的 cond 那样的嵌套分支语句。这两种语句其实是语义完全不同的，然而 Go 的设计者为了显得简单，把它们合二为一，而其实引起了更大的混淆。这是因为，就算你把它们合二为一，它们仍然是两种不同的语义结构。把它们合并的结果是，每次看到 switch 你都需要从它们“头部”的不同点把这两种不同的结构区分开来，增加了人脑的开销。正确的作法是把它们分开，就像 Scheme 那样。其实我设计语言的时候有时候也犯同样的错误，以为两个东西“本质”上是一样的，所以合二为一，结果经过一段时间，发现其实是不一样的。所以不要小看了 Scheme，很多你认为是“新想法”的东西，其实早就被它那非常严谨的委员会给抛弃在了历史的长河中。

Go 语言里面还有其他一些语法设计问题，比如强制把 { 放在一行之后而且不能换行，if 语句的判断开头可以嵌套赋

值操作等等。这些试图让程序显得短小的作法，其实反而降低了程序理解的流畅度。

所以总而言之，Go 的语法很难被叫做“简单”或者“优雅”，它的简单性其实在 Java 之下。

工具链

Go 提供了一些比较方便的工具。比如 `gofmt`, `godef` 等，使得 Go 代码的编程比起单用 Emacs 或者 VIM 来编辑 C 和 C++ 来说是一个进步。使用 Emacs 编辑 Go 就已经能实现某些 IDE 才有的功能，比如精确的定义跳转等等。

这些工具虽然好用，但比起像 Eclipse, IntelliJ 和 Visual Studio 这样的 IDE，差距还是相当大的。比起 IDE，Go 的工具链缺乏各种最基本的功能，比如列出引用了某个变量的所有位置，重命名等 refactor 功能，好用的 debugger (GDB 不算好用) 等等。

Go 的各种工具感觉都不太成熟，有时候你发现有好几个不同的 package 用于解决同一个问题，搞不清楚哪一个好些。而且这些东西配置起来不是那么的可靠和简单，都需要折腾。每一个小功能你都得从各处去寻找 package 来配置。有些时候一个工具配置了之后其实没有起作用，要等你摸索好半天才发现问题出现在哪里。这种没有组织，没有计划的工具设计，是很难超过专业 IDE 厂商的连贯性的。

Go 提供了方便的 package 机制，可以直接 import 某个 GitHub repository 里的 Go 代码。不过我发现很多时候这种 package 机制带来的更多是麻烦事和依赖关系。所以 Go 的推崇者们又设计了一些像 `godep` 的工具，用来绕过这些问题，结果 `godep` 自己也引起一些稀奇古怪的问题，导致有时候新的代码其实没有被编译，产生莫名其妙的错误信息（可能是由于 `godep` 的 bug）。

我发现很多人看到这些工具之后总是很狂热的认为它们就能让 Go 语言一统天下，其实还差得非常之远。而且如此年轻的语言就已经出现这么多的问题，我觉得所有这些麻烦事累积下来，多年以后恐怕够呛。

内存管理

比起 C 和 C++ 完全手动的内存管理方式，Go 有垃圾回收 (GC) 机制。这种机制大大减轻了程序员的头脑负担和程序出错的机会，所以 Go 对于 C/C++ 是一个进步。

然而进步也是相对的。Go 的垃圾回收器是一个非常原始的 mark-and-sweep，这比起像 Java, OCaml 和 Chez Scheme 之类的语言实现，其实还处于起步阶段。

当然如果真的遇到 GC 性能问题，通过大量的 tuning，你可以部分的改善内存回收的效率。我也看到有人写过一些文章介绍他们如何做这些事情，然而这种文章的存在说明了 Go 的垃圾回收还非常不成熟。GC 这种事情我觉得大部分时候不应该是让程序员来操心的，否则就失去了 GC 比起手动管理的很多优势。所以 Go 代码想要在实时性比较高的场合，还是有很长的路要走的。

由于缺乏先进的 GC，却又带有高级的抽象，所以 Go 其实没法取代 C 和 C++ 来构造底层系统。Go 语言的定位对我来说越来越模糊。

没有“generics”

比起 C++ 和 Java 来说，Go 缺乏 generics。虽然有人讨厌 Java 的 generics，然而它本身却不是个坏东西。Generics 其实就是 Haskell 等函数式语言里面所谓的 parametric polymorphism，是一种非常有用的东西，不过被 Java 抄去之后有时候没有做得全对。因为 generics 可以让你用同一块代码来处理多种不同的数据类型，它为避免重复，方便替换复杂数据结构等提供了方便。

由于 Go 没有 generics，所以你不得不重复写很多函数，每一个只有类型不同。或者你可以用空 interface {}，然而这个东西其实就相当于 C 的 void* 指针。使用它之后，代码的类型无法被静态的检查，所以其实它并没有 generics 来的严谨。

比起 Java，Go 的很多数据结构都是“hard code”进了语言里面，甚至创造了特殊的关键字和语法来构造它们（比如哈希表）。一旦遇到用户需要自己定义类似的数据结构，就需要把大量代码重写一遍。而且由于没有类似 Java collections 的东西，无法方便的换掉复杂的数据结构。这对于构造像 PySonar 那样需要大量实验才能选择正确的数据结构，需要实现特殊的哈希表等数据结构的程序来说，Go 语言的这些缺失会是一个非常大的障碍。

缺少 generics 是一个问题，然而更严重的问题是 Go 的设计者及其社区对于这类语言特性的盲目排斥。当你提到这些，Go 支持者就会以一种蔑视的态度告诉你：“我看不到 generics 有什么用！”这种态度比起语言本身的缺点来说更加有害。在经过了很长一段时间之后 Go 语言的设计者们开始考虑加入 generics，然后由于 Go 的语法设计偷工减料，再加上由于缺乏 generics 而产生的特例（比如 Go 的 map 的语法设计）已经被大量使用，我觉得要加入 generics 的难度已经非常大。

Go 和 Unix 系统一样，在出现的早期就已经因为不吸取前人的教训，背上了沉重的历史包袱。

多返回值

很多人都觉得 Go 的多返回值设计是一个进步，然而这里面却有很多蹊跷的东西。且不说这根本不是什么新东西（Scheme 很早就有了多返回值 let-values），Go 的多返回值却被大量的用在了错误的地方—Go 利用多返回值来表示出错信息。比如 Go 代码里最常见的结构就是：

```
ret, err := foo(x, y, z)
if err != nil {
    return err
}
```

如果 foo 的调用产生了错误，那么 err 就不是 nil。Go 要求你在定义了变量之后必须使用它，否则报错。这样它“碰巧”避免了出现错误 err 而不检查的情况。否则如果你想忽略错误，就必须写成

```
ret, _ := foo(x, y, z)
```

这样当 foo 出错的时候，程序就会自动在那个位置当掉。

不得不说，这种“歪打正着”的做法虽然貌似可行，从类型系统角度看，却是非常不严谨的。因为它根本不是为了这个目的而设计的，所以你可以比较容易的想出各种办法让它失效。而且由于编译器只检查 err 是否被“使用”，却不检查你是否检查了“所有”可能出现的错误类型。比如，如果 foo 可能返回两种错误 Error1 和 Error2，你没法保证调用者完全排除了这两种错误的可能性之后才使用数据。所以这种错误检查机制其实还不如 Java 的 exception 来的严谨。

另外，ret 和 err 同时被定义，而每次只有其中一个不是 nil，这种“或”的关系并不是靠编译器来保障，而是靠程序员的“约定俗成”。这样当 err 不是 nil 的时候，ret 其实也可以不是 nil。这些组合带来了挺多的混淆，让你每次看到 return 的地方都不确信它到底想返回一个错误还是一个有效值。如果你意识到这种“或”关系其实意味着你只应该用一个返回值来表示它们，你就知道其实 Go 误用了多返回值来表示可能的错误。

其实如果一个语言有了像 [Typed Racket](#) 和 [PySonar](#) 所支持的“union type”类型系统，这种多返回值就没有意义了。因为如果有 union type，你就可以只用一个返回值来表示有效数据或者错误。比如你可以写一个类型叫做 {String, FileNotFoundError}，用于表示一个值要么是 String，要么是 FileNotFoundError 错误。如果一个函数有可能返回错误，编译器就强制程序员检查所有可能出现的错误之后才能使用数据，从而可以完全避免以上的各种混淆情况。对 union type 有兴趣的人可以看看 Typed Racket，它拥有我迄今为止见过最强大的类型系统（超越了 Haskell）。

所以可以说，Go 的这种多返回值，其实是“歪打”打着了一半，然后换着法子继续歪打，而不是瞄准靶心。

接口

Go 采用了基于接口（interface）的面向对象设计，你可以使用接口来表达一些想要进行抽象的概念。

然而这种接口设计却不是没有问题的。首先跟 Java 不同，实现一个 Go 的接口不需要显式的声明（implements），所以你有可能“碰巧”实现了某个接口。这种不确定性对于理解程序来说是有反作用的。有时候你修改了一个函数之后就发现编译不通过，抱怨某个位置传递的不是某个需要的接口，然而出错信息却不能告诉你准确的原因。要经过一番摸索你才发现你的 struct 为什么不再实现之前定义的一个接口。

另外，有些人使用接口，很多时候不过是为了传递一些函数作为参数。我有时候不明白，这种对于函数式语言再简单不过的事情，在 Go 语言里面为什么要另外定义一个接口来实现。这使得程序不如函数式语言那么清晰明了，而且修改起来也很不方便。有很多冗余的名字要定义，冗余的工作要做。

举一个相关的例子就是 Go 的 [Sort](#) 函数。每一次需要对某种类型 T 的数组排序，比如 []string，你都需要

1. 定义另外一个类型，通常叫做 TSorter，比如 StringSorter
2. 为这个 StringSorter 类型定义三个方法，分别叫做 Len, Swap, Less
3. 把你的类型比如 []string cast 成 StringSorter
4. 调用 sort.Sort 对这个数组排序

想想 sort 在函数式语言里有多简单吧？比如，Scheme 和 OCaml 都可以直接这样写：

```
(sort '(3 4 1 2) <)
```

这里 Scheme 把函数 < 直接作为参数传给 sort 函数，而没有包装在什么接口里面。你发现了吗，Go 的那个 interface 里面的三个方法，其实本来应该作为三个参数直接传递给 Sort，但由于受到 design pattern 等思想的局限，Go 的设计者把它们“打包”作为接口来传递。而且由于 Go 没有 generics，你无法像函数式语言一样写这三个函数，接受比较的“元素”作为参数，而必须使用它们的“下标”。由于这些方法只接受下标作为参数，所以 Sort 只能对数组进行排序。另外由于 Go 的设计比较“底层”，所以你需要另外两个参数：len 和 swap。

其实这种基于接口的设计其实比起函数式语言，差距是很大的。比起 Java 的接口设计，也可以说是一个倒退。

goroutine

Goroutine 可以说是 Go 的最重要的特色。很多人使用 Go 就是听说 goroutine 能支持所谓的“大并发”。

首先这种大并发并不是什么新鲜东西。每个理解程序语言理论的人都知道 goroutine 其实就是一些用户级的“continuation”。系统级的 continuation 通常被叫做“进程”或者“线程”。Continuation 是函数式语言专家们再了解不过的东西了，比如我的前导师 Amr Sabry 就是关于 continuation 的顶级专家之一。

Node.js 那种“callback hell”，其实就是函数式语言里面常用的一种手法，叫做 continuation passing style (CPS)。由于 Scheme 有 call/cc，所以从理论上讲，它可以不通过 CPS 样式的代码而实现大并发。所以函数式语言只要支持 continuation，就会很容易的实现大并发，也许还会更高效，更好用一些。比如 Scheme 的一个实现 Gambit-C 就可以被用来实现大并发的东西。Chez Scheme 也许也可以，不过还有待确认。

当然具体实现上的效率也许有区别，然而我只是说，goroutine 其实并不是像很多人想象的那样全新的，革命性的，独一无二的东西。只要有足够的动力，其它语言都能添加这个东西。

defer

Go 实现了 defer 函数，用于避免在函数出错后忘了收拾残局 (cleanup)。然而我发现这种 defer 函数有被滥用的趋势。比如，有些人把那种不是 cleanup 的动作也做成 defer，到后来累积几个 defer 之后，你就不再能一眼看得清楚到底哪块代码先运行哪块后运行了。位置处于前面的代码居然可以在后来运行，违反了代码的自然位置顺序关系。

当然这可以怪程序员不明白 defer 的真正用途，然而一旦你有了这种东西就会有人想滥用它。那种急于试图利用一个语言的每种 feature 的人，特别喜欢干这种事情。这种问题恐怕需要很多年的经验之后，才会有人写成书来教育大家。在形成统一的“代码规范”以前，我预测 defer 仍然会被大量的滥用。

所以我们应该想一下，为了避免可能出现的资源泄漏，defer 带来的到底是利多还是弊多。

库代码

Go 的标准库的设计里面带有浓郁的 Unix 气息。比起 Java 之类语言，它的库代码有很多不方便的地方。有时候引入了一些函数式语言的方式，但却由于 Unix 思维的限制，不但没能发挥函数式语言的优点，而且导致了很多理解的复杂性。

一个例子就是 Go 处理字符串的方式。在 Java 里每个字符串里包含的字符，缺省都是 Unicode 的“code point”。然而在 Go 里面 string 类型里面每个元素都是一个 byte，所以每次你都得把它 cast 成“rune”类型才能正确的遍历每个字符，然后 cast 回去。这种把任何东西都看成 byte 的方式，就是 Unix 的思维方式，它引起过度底层和复杂的代码。

HTML template 库

我使用过 Go 的 template library 来生成一些网页。这是一种“基本可用”的模板方式，然而比起很多其他成熟的技术，却是相当的不足的。让我比较惊讶的是，Go 的 template 里面夹带的代码，居然不是 Go 语言自己，而是一种表达能力相当弱的语言，有点像一种退化的 Lisp，只不过把括号换成了 { ... } 这样的东西。

比如你可以写这样的网页模板：

```
{ define "Contents" }
{ if .Paragraph.Length }
<p>{ .Paragraph.Content } </p>
{ end }
{ end }
```

由于每个模板接受一个 struct 作为填充的数据，你可以使用 .Paragraph.Content 这样的代码，然而这不但很丑陋，而且让模板不灵活，不好理解。你需要把需要的数据全都放进同一个结构才能从模板里面访问它们。

任何超过一行的代码，虽然也许这语言可以表达，一般人为了避免这语言的弱点，还是在 .go 文件里面写一些“帮助函数”。用它们产生数据放进结构，然后传给模板，才能够表达模板需要的一些信息。而这每个帮助函数又需要一定的“注册”信息才能被模板库找到。所以这些复杂性加起来，使得 Go 的 HTML 模板代码相当的麻烦和混乱。

听说有人在做一个新的 HTML 模板系统，可以支持直接的 Go 代码嵌入。这些工作刚刚起步，而且难说最后会做成什么样子。所以要做网站，恐怕还是最好使用其他语言比较成熟的框架。

总结

优雅和简单性都是相对而言的。虽然 Go 语言在很多方面超过了 C 和 C++，也在某些方面好于 Java，然而它其实也没法和 Python 的优雅性相比的，而 Python 在很多方面却又不如 Scheme 和 Haskell。所以总而言之，Go 的简单性和优雅程度属于中等偏下。

由于没有明显的优势，却又有各种其它语言里没有的问题，所以在实际工程中，我目前更倾向于使用 Java 这样的语言。我不觉得 Go 语言和它的工具链能够帮助我迅速的写出 PySonar 那样精密的代码。另外我还听说有人使用 Java 来实现大并发，并没发现比起 Go 有什么明显的不足。

Alan Perlis 说，语言设计不应该是把功能堆积起来，而应该努力地减少弱点。从这种角度来看，Go 语言引入了一两个新的功能，同时又引入了相当多的弱点。

Go 也许暂时在某些个别的情况下有特殊的强项，可以单独用于优化系统的某些部分，但我不推荐使用 Go 来实现复杂的算法和整个的系统。

一个对 Dijkstra 的采访视频



(也可以访问 [YouTube](#) 或者从源地址下载 [MPEG1](#) , 300M)

之前在微博上推荐了一个对 Dijkstra 的采访视频，看了两遍之后觉得实在很好，所以再正式推荐一下。大部分人可能都知道他对图论算法和操作系统的贡献，而其实 Dijkstra 在程序语言上的造诣也很深厚。我们常用的程序语言里面司通见惯的“递归函数”，其实当年就是 Dijkstra 和另一个人不顾委员会里众人的反对和怀疑，坚持要放进 Algol 60，所以来才进入了 Pascal, C, Java 这样的语言的。那个时候 John McCarthy 缺席，不然的话就会有三个人支持了。

现在看来，任何一个语言里面没有递归函数都是不可思议的事情，然而在1950-60年代的时候，居然很少有人知道它有什么用！所以你就发现，所谓的“主流”和“大多数人”一直都是比较愚蠢的。现在，同样的故事发生在 lambda 身上。多年以后，没有 lambda 的语言将是不可接受的。

在这里只摘录他提到的几个要点。某些观点也许不是最好的办法，但我确信其中有非常值得学习的地方。

1. 软件的版本号 2.6, 2.7, ... 都是胡扯。本来第1版就应该是最终的产品，可是软件公司总是先弄出来一个不完整的版本，骗大家买了，以后再慢慢“升级”。每次升级都要用户再次付钱。
2. 编程有多种流派，我喜欢把它们归类成“莫扎特 vs 贝多芬”。当莫扎特开始写乐谱时，作品就已经完成了。他的手稿一气呵成，书法也很好。贝多芬不一样，他总是在怀疑和挣扎。他的作品一般是还没有想好就开始写，然后就往上面贴纸条修改。有一次贝多芬改了9遍才把手稿完成，后来有人把这手稿一层层的撕开，发现第一版和最后一版是一摸一样的。这种改来改去的做法是 Anglo-Saxon 民族的传统，它贯穿了英国式的教育。
3. 作曲家的工作不是写乐谱，而是构思音乐。最早的时候人们编程都是用汇编语言的，就跟写乐谱差不多。后来他们发明了高级语言，就以为这些语言把编程的问题解决了。但是你仔细一瞧，发现它们只是把编程最微不足道的问题解决了，但是困难的问题仍然困难。这些高级语言与越来越大的野心加在一起，反而让程序员头脑的负担更重了。
4. 称职的程序员都知道自己头颅的尺寸是有限的，所以他们以谦逊的态度来对待工作，像回避瘟疫一样地回避小聪明。
5. 当我1970年在法国巴黎讲学如何编程的时候很成功，听众都非常积极。回家的路上我又在比利时布鲁塞尔的一个大软件公司进行了同样的演讲，结果非常失败。那恐怕是我一生中最失败的演讲。后来我发现了为什么：他们的管理层不喜欢无懈可击的程序，因为这公司是靠“维护软件”的合同来维持生存的。程序员对此也不感兴趣，因为最让他们兴奋的事情在于不知道自己在干什么。他们觉得如果清楚地知道自己在干什么，那就没有挑战性了，就是无聊的工作。
6. 研究物理的人如果遇到不理解的事情，总是可以责怪上帝，世界这么复杂不是你的错。但是如果你的程序有问题，那就找不到替罪羊了。0就是0, 1就是1，就是你把它搞砸了。
7. 1969年，在阿波罗号登月之后不久，我在罗马的北约软件工程会议遇到了 Joel Aron，阿波罗计划的软件负责人。我知道每个阿波罗飞船上面的代码都会比前一个多4万行。我不知道“行”对于代码是个什么单位，但4万行肯定是很多了。我很惊讶他们能把这么多代码做对，所以我问 Joel：你们是怎么做到的？他说：做什么？我说：把那么多代码写正确。Joel 说：“正确？！其实在发射前仅仅五天，我从登月器计算轨道的代码里发现一个错误，这代码把月球的重力方向算反了。本来该吸引的，结果写成了排斥。是一个偶然的机会让我发现了这个错误。”我的脸都白了，说：这些家伙运气真好？Joel 说：“是的。”
8. 软件测试可以确定软件里有 bug，但却不可能用来确定它们没有 bug。

9. 程序的优雅性不是可以或缺的奢侈品，而是决定成功还是失败的一个要素。优雅并不是一个美学的问题，也不是一个时尚品味的问题，优雅能够被翻译成可行的技术。牛津字典对 *elegant* 的解释是：[pleasingly ingenious and simple](#)。如果你的程序真的优雅，那么它就会容易管理。第一是因为它比其它的方案都要短，第二是因为它的组件都可以被换成另外的方案而不会影响其它的部分。很奇怪的是，最优雅的程序往往也是最高效的。
10. 当没有计算机的时候，编程不是问题。当有了比较弱的计算机时，编程成了中等程度的问题。现在我们有了巨大的计算机，编程就成了巨大的问题。
11. 我最开头编程的日子跟现在很不一样，因为我是给一个还没有造出来的计算机写程序。造那台机器的人还没有完工，我在同样的时间给它做程序，所以没有办法测试我的代码。于是我发现自己的东西必须要能放进自己的脑子里。
12. 我的母亲是一个优秀的数学家。有一次我问她几何难不难，她说一点也不难，只要你用“心”来理解所有的公式。如果你需要超过5行公式，那么你就走错路了。
13. 为什么这么少的人追求优雅？这就是现实。如果说优雅也有缺点的话，那就是你需要艰巨的工作才能得到它，需要良好的教育才能欣赏它。

程序员的心理疾病

由于程序员工作的性质，他们长期以来受到的所谓“黑客”式的“熏陶”，形成了一种行业性的心理疾病。患了这种病的人对于很多新入行的人，甚至一些外行人士造成了持续的伤害。慢慢的，这些不幸的受害者也形成了“条件反射”，进而成为了这个心理变态的系统的一部分，导致越来越多的人，越来越快的变成“怪胎”。这是一件可怕的事情，所以我觉得有必要警醒一下。

这里我就简单的把我所观察到的一些症状总结一下，希望作为对于 IT 业界人士的警示，有则改之，无则加勉。也希望为遇到类似问题的新手和外行人士提供一些精神上的支持，以免他们也成为这个系统的一部分。

无自知之明

由于程序员的工作最近几年比较容易找，工资还不错，所以很多程序员往往只看到自己的肚脐眼，看不到自己在整个社会里的位置其实并不是那么的关键和重要。很多程序员除了自己会的那点东西，几乎对其它领域和事情完全不感兴趣，看不起其他人。这就是为什么我的前同事 TJ 作为一个资深的天体物理学家，在一个软件公司里面那么卑微。貌似会写点 node.js, iOS 软件的人都可以对他趾高气昂的样子，而其实这些东西的价值哪里可能跟 TJ 知道的物理知识相提并论。很多科学家其实都可以轻而易举的掌握程序员知道的那点东西，有人却认定了他们不是这个专业的，不懂我们的东西，或者故意把问题搞复杂，让他们弄不明白。

其实对于一个物理学家，他心目中知识的价值是这样排序的：



COBOL 在那么靠前的位置我觉得是用来搞笑的，不过你大致看到了很多 IT 技术在真正的科学家眼里的价值和它们的有效期。

如果力学工程师犯了错误，飞机会坠毁；如果结构工程师犯了错误，大桥会垮塌；可是如果软件工程师犯了错误，大不了网站挂掉一小时，重启一下貌似又好了。所以所谓“软件工程师”，由于门槛太低，他们的工作严谨程度，其实是没法和力学工程，结构工程等真正的工程师相提并论的。实际上“软件工程”这个名词根本就是扯淡的，软件工程师也不能被叫做“工程师”。跟其他的工程不一样，软件工程并不是建立在科学的基础上的一计算机科学其实不是科学。

垃圾当宝贝

按照 Dijkstra 的说法，“软件工程”是穷途末路的领域，因为它的目标是：如果我不会写程序的话，怎么样才能写出程序？

为了达到这个愚蠢的目的，很多人开始兜售各种像减肥药一样的东西。面向对象方法，软件“重用”，设计模式，关系

式数据库，NoSQL，大数据…… 没完没了。只要是有钱人发布的东西，神马垃圾都能被吹捧上天。Facebook 给 PHP 做了个编译器，可以编译成 C++，还做了个 VM，多么了不起啊！其实那种东西就是我们在 Indiana 第一堂课就写过的，只不过我们是把比 PHP 好很多的语言翻译成 C。我们根本不想给 PHP 那么垃圾的语言做什么编译器，让垃圾继续存活下去并不能证明我们的价值。

其实软件里面有少数永恒的珍宝，可惜很少有人理解和尊重它们的价值。这在其它的工程领域看来是不可思议的，然而这却是事实。由于没有科学作为理论的基础，没有实验作为检验它们的标准，软件行业的很多东西就像现代艺术一样，丑陋无比的垃圾还能摆在外表堂皇的“现代艺术博物馆”里面，被人当成传世大作一样膜拜。

为了凸显自己根本不存在的价值，又提出一些新的“理念”，就像有些现代艺术家一样，说“艺术的目的不是为了美，而是为了自由。”哦，这就是为什么你们可以自由地把那些让人反胃的东西放在博物馆里，还要买门票才能参观？

宗教斗争

当然了因为没有实质的技术，为了争夺市场和利益，各种软件的理念就开始互相倾轧。一会儿说软件危机啦，面向对象方法来拯救你们！一会儿又提出设计模式。过了一会儿又有人说这些设计模式里面有些模式是“反模式”，然后又有人把函数式编程包装起来，说是面向对象编程的克星，一会儿是关系式数据库，一会儿是 NoSQL，一会儿是 web，一会儿是 cloud，一会儿又是 mobile…… 每个东西都喜欢把自己说成是未来的希望。

这就是为什么有人说在软件行业里需要不停地“学习”，因为不断地有人为了制造新的理念而制造新的理念。在这样一个行业里，你会很难找到一个只把程序语言或者技术当成是工具的人。如果有人问你对某个语言或者技术的评价，是非常尴尬甚至危险的事情，所以最可靠的办法就是不做评论，什么都不要说。



引难为豪

在 IT 行业里批评一个技术难用，是一件非常容易伤自尊的事情，因为立马会有人噼里啪啦打出一些稀奇古怪的命令或者一大篇代码，说：就是这么简单！然后你就发现，这些人完全不明白什么叫做设计，他们以自己能用最快的速度绕过各种前人的设计失误为豪，很多程序员甚至以自己打字快为豪。

往往也就是这些自诩打字快的人喜欢使用过度复杂的方法来解决问题。我可以告诉你，我打字的速度是相当之慢的。我大量的使用鼠标，方向键，而且把 Emacs 里最常用的功能都尽量绑到 F 功能键上，这样我就可以用一个指头启动一个功能。Dan Friedman 的打字速度就更慢，而且他经常故意使用“一指禅”。为什么呢？因为我们写出来的代码非常精辟，几乎不带多余的垃圾，所以根本不需要打很快。

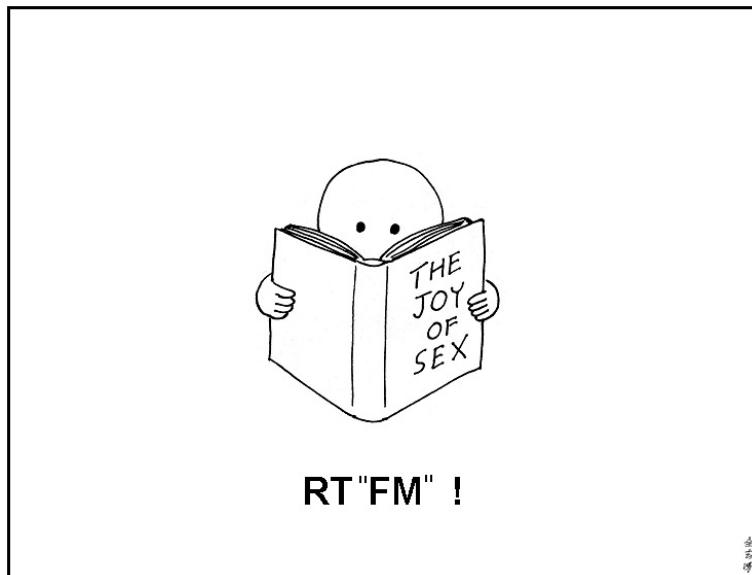
当遇到这样引难为豪的人，我的经验是，千万不要恭维他们。你必须嘲笑这些东西的设计，并且指出它们的失误之处，否则你不但助长了这些人的气焰，让这种风气继续延续下去，而且将来自己的自尊也难保了。很可惜，并不是每个人都有这种勇气把这些话说出来，这就造成了今天的局面，纷繁复杂的垃圾充斥着世界。

爱因斯坦说，你需要很多的天才和非常大的勇气，才能追求到简单。非常大的勇气……也许就是这个意思。

去读文档！

不知从什么时候开始，人们开始引用 Eric Raymond 的一篇叫做《提问的艺术》的文章，这篇文章后来就成为了对提问者没礼貌的借口。由于这篇文章的误导，当你希望同事能给你一个手把手的演示的时候，他们往往会丢给你一篇不知道什么时候写的文档，让你自己去读，仿佛文档就可以代替人之间的直接互动。况且不说这文档可能已经过时，里面有很多地方已经不符合最新的设计，而这意味着在潜意识里，他们觉得高你一等。

对于这种现象有一个专门的词汇，叫做 RTFM (Read The Fucking Manual) :



在 IRC 的聊天室里，由于隔着网络的屏障，这种对提问者没礼貌的现象就更加嚣张。我曾经有几次去 Java 的聊天室问一些貌似基础，而其实很深入的语言设计问题，结果没有一次不是以收到像“去读 API！”这样的回答而结束。API 谁不会读，然而我需要的是一个有血有肉的人对此的理解。所以来我根本不去 IRC 这种地方了，因为那里面对你打字基本上已经不是人类了。他们觉得你问问题浪费了他们的时间，好像他们一天到晚泡在 IRC 里面就是在做什么正事似的。不想回答问题，不开口还不行吗。后来你发现，原来在 IRC 里面训斥新手就是这些人唯一的乐趣，所以其实他们是非开口说话不可的。然而这次他们遇到的却不是个新手，而是一个可以把 Java 整个造出来的人。

像 Haskell 之类的聊天室貌似稍微友好一点，然而后来你发现他们显得友好是有所企图的。因为当时 Haskell 还没有很多人用，他们需要吸引新手，所以竭尽所能的诱导他们。而一旦它用户稍微多了一点，有声势了，就有人开始居高临下，成为专家一样的人物。他们就开始写书，然后就开始牛气哄哄的了。然后你就会发现当对 Haskell 的设计提出异议的时候，这些“id”们是多么的不友好，有理也说不清。所以最后你发现，其实所有语言的所谓“社区”都一个德行。如果 Haskell 有一天像 Java 一样如日中天（当然不大可能），肯定对大部分问题的答案也就是“去读API！”其实它已经在向这一步发展了。

不得不指出，《提问的艺术》等介绍“黑客文化”的文章对于这种现象的出现有着极大的责任。说穿了，写这些文章的人一般都是 Unix 的跟屁虫。这种文章试图抹去人类文明几千年来传承的文化，而重新给“礼貌”做出定义。其结果是，人类的文明因为这些文章，在程序员的世界里倒退了几十甚至几百年。很多外行人不喜欢跟程序员说话，叫他们是 nerd，就是这个原因。



不要提问，不要谦虚，不要恭维

跟上面的症状相似，程序员世界里的一条重要的潜规则是：只有菜鸟才会问问题。所以如果你有任何机会可以自己得到答案，就不要试图向人“请教”，尤其不要显得好奇，否则你就会被认为是菜鸟。我有几次不耻下问的经历，最后导致了我被人当成菜鸟。我只是觉得那问题有趣，也许能够启发我设计自己的东西，所以吃饭时觉得是个话题可以说一下，结果呢就有人忙着鄙视你，那么小的问题都没搞清楚。正确的态度应该是诚实，直接，见惯不惊，那有什么大不了的，我什么没见过，我很怀疑。

随之而来的引论就是：不要谦虚！那些“职场经验”之类的文章告诉你的进入新的公司工作，要谦虚好问，对IT公司是不管用的。有的大IT公司有所谓的“文化”，比如叫你要“humble”，其实只是用来贬低你价值的借口。他们只是想让你安于“本分”，做一些微不足道，不能发挥你才能的工作。看看那些叫你要 humble 的人，他们 humble 吗？所以跟江湖一样，在IT公司里面一件很重要的事情是，亮出自己的宝剑和绝招，给人下马威。介绍自己的东西一定要自豪，这就是世界上最好的，无敌的，没有其他人能做到！不能有任何保留。不要像科学家一样介绍自己技术的局限性，否则随之而来的就是有些人对你价值的怀疑和对你自信心的打击。

另外要注意的是对于别人介绍的东西，不要轻易地表扬或者点头，否则有人就更有气势了。你要问这样的问题：这里面有什么新的东西吗？这个事情，另外一种技术早就能做了啊，没觉得有什么了不起。

以语言取人

你的软件是什么语言写的，告诉别人的时候是千万要小心的，不到万不得已最好不要说。因为十有八九，对方会立即在心里对你的软件的价值做出判断，光凭你用的是什么语言。

很多程序员都以自己会用最近流行的一些新语言为豪，以为有了它们自己就成了更好的程序员。他们看不到，用新的语言并不能让他们成为更好的程序员。其实最厉害的程序员无论用什么语言都能写出很好的代码。在他们的头脑里其实只有一种很简单的语言，他们首先用这种语言把问题建模出来，然后根据实际需要“翻译”成最后的代码。这种在头脑里的建模过程的价值，是很难用他最后用语言的优劣来衡量的。

有时候高明的程序员用一个语言并不是因为他只会用那种语言，而是其他的原因。他们的头脑里有着万变不离其宗的理念，可以让他们立即掌握几乎任何语言或者工具，所以他们对所谓的“新语言”都不以为然。可是很多人误以为他们不愿意学习“新东西”，从而从心里鄙视他们。其实计算机的世界里哪里有很多新的东西，只不过是有人给同样的东西起了很多不同的名字而已。如果连这样的程序员都不能理解你的技术，就说明你的技术设计有问题，而不是他们有问题。就像 Seymour Cray 说的，我只能理解简单的东西，如果它太复杂了，我是不能理解的。

早些年的时候，大家都认为招募某种特定语言的程序员是一种浮浅的做法，很多公司看重的都是解决问题的能力。可是近些年我发现这些浮浅的做法越来越普遍。可以说现在像 Google 这样的公司面试员工的方式和态度，其实还不如八年前我的第一份国内工作。而这种现象在使用 Python，Ruby，JavaScript 等“流行语言”的公司里就更为普遍。

跟屁虫

有些程序员对新手和同事是那么的不友好，然而对大牛们拍马屁的功夫可真是出类拔萃。我刚到旧金山的几个月有时候参加一些程序语言的“meetup”，后来我发现这种 meetup 都是宗教气氛非常浓厚的地方，跟传销大会差不多。Scala 的 meetup 里面的人几乎全都对 Scala 和 Martin Odersky 顶礼膜拜，甚至把 Rod Johnson 请来说一堆胡话。Clojure 的，当然基本上把 Rich Hickey 当成神，甚至称他为“二十一世纪最重要的思想家之一”。各种 talk 总是宣扬，哇，我们用 Scala/Clojure 做出了多么了不起的东西云云，其实只不过是在向你兜售减肥药。

很多人喜欢做这些新的语言和技术的“evangelist”，尽显各种马屁神功，然后就开始写书，写 blog，…… 目的就是成为这个“领域”的第一批专家。这就难怪了，再垃圾的语言也有一大批人来鼓吹。因为这些没真本事的人，随便把一个东西捧上天都有自己的好处。

由于受到这些“先知”的影响，有些人开始在他们自己的公司里“布道”。比如有人在 Python 的 meetup 集会时告诉我，他试图在自己的小组里推 Python，可是一些老顽固一定要用 Java，认为 Java 才是王道。很鄙夷不高兴的样子。我并不认为 Java 是很好的语言，然而 Python 也好不到哪去。它们在我眼里只不过是临时拿来用一下的工具，可是我仍然能用它们写出一流的代码。

看到这些宗教性质的聚会，我终于理解了一些地区是如何被从一个国家分裂出去，最后沦落为另外一个国家殖民地的。最早的时候，一般是派传教士过去“传经”，然后就煽动一小部分人起来造反。到后来就可以名正言顺的以“保护传教士”，“保护宗教自由”，“维持和平”等理由把军舰开到别人家门口……

程序语言与它们的工具

谈论了这么多程序语言的事情，说得好像语言的好坏就是选择它们的决定性因素。然而我一直没有提到的一个问题是，“程序语言”和“程序语言工具”的设计，其实完全是两码事。一个优秀的程序语言，有可能由于设计者的忽视或者时间短缺，没有提供良好的辅助工具。而一个不怎么好的程序语言，由于用的人多了，往往就会有人花大力气给它设计工具，结果大大的提高了易用性和程序员的生产力。我曾经提到，程序语言其实不是工具，它们是像木头，钉子，胶水一样的材料。如果有公司做出非常好的胶水，粘性极强，但它的包装不好，一打开就到处乱跑，弄得一团糟。你是愿意买这样的胶水还是稍微差一点但粘性足够，包装设计合理，容易涂抹，容易存储的呢？我想大部分人会选择后者，除非后者的粘性实在太弱，那样的话包装再好都白搭。

这就是为什么虽然我这么欣赏 Scheme，却没有用 Scheme 或者 Racket 来构造 PySonar 和 RubySonar，甚至没有选择 Scala 和 Clojure，而是“臭名昭著”的 Java。这不只是因为 PySonar 最初的代码由于项目原因是用 Java 写的，而且因为 Java 正好有足够的表达能力，可以实现这样的系统，但是最重要的其实是，Java 的工具非常成熟和迅捷。很难想象如果缺少了 Eclipse 我还能在三个月内做出像 PySonar 那样的东西。而现在我只用了一个月就做出了 RubySonar，其中很大的功劳在于 IntelliJ。这些 IDE 的跳转功能，让我可以在代码中自由穿梭。而它们的 refactor 功能，让我不必再为变量的命名而烦恼，因为只要临时起个不重复的名字就行，以后改起来小菜一碟。另外我还经常使用这些 IDE 里面的 debugger，利用它们我可以很方便的找到 bug 的起因。PySonar2 在有一段时间变得很慢，看不出是哪里出了问题。最后我下载了一个 JProfiler 试用版，很快就发现了问题的所在。如果这问题出现在 Scheme 代码里面，恐怕就要费很多功夫才能找到，因为 Scheme 没有像 JProfiler 那样的工具。

但这并不等于说学习 Scheme 是没有用处的。恰恰相反，Scheme 的知识在任何时候都是非常有用的。一个只学过 Java 的程序员基本上是不可能写出我那样的 Java 代码的。虽然那看起来是 Java，但是其实 Scheme 的灵魂已经融入到其中了。我从 Scheme 学到的知识不但让我知道 Java 可以怎么用，而且让我知道 Java 本身是如何被造出来的。我知道 Java 哪些地方是好的，哪些地方是不好的，从而能够择其善而避其不善。我的代码没有用任何的“Java 设计模式”，也没有转弯抹角的重载。

其实我有空的时候在设计和实现自己的语言（由于缺乏想象力，暂命名为 Yin），它的实现语言也在最近换成了 Java。Yin 的语法接近于 Scheme，好像理所当然应该用 Scheme 或者 Racket 来实现。有些人可能已经看到了我 GitHub 上面的第一个 prototype 实现（项目已经进入私密状态）用的是 Typed Racket。Racket 在很大程度上是比 Java 好的语言，然而它却有一个让我非常恼火的问题，以至于最后我怀疑自己能否用它顺利实现自己的语言。

这个问题就是，当运行出现错误的时候，Racket 不告诉我出错代码的具体行号，甚至出错的原因都不说清楚。我经常看到这样一些出错信息：

“函数调用参数个数错误”
“变量 a 没有定义，位于 loop 处”

只说是函数调用，函数叫什么名字不说。只说是 loop，文件里那么多 loop，到底是哪一个不知道。出错信息里面往往有很多别的垃圾信息，把你指向 Racket 系统里面的某一个文件。有时候把代码拷贝进 DrRacket 才能找到位置，可是很多时候甚至 DrRacket 都不行。每当遇到这些就让我思路被打断很长时间，导致代码质量的下降。

其它的 Scheme 实现也有类似的问题，像 Petite Chez 这样的就更加严重，只有商业版的 Chez Scheme 会好一些，所以这里不只是小小的批评一下。这种对工具设计的不在意心理，在 Lisp 和 Scheme 等函数式语言的社区里非常普遍。每当有人抱怨它们出错信息混乱，没有 debugger，没有基本的静态检查，铁杆 Schemer 们就会鄙视你说：“Aziz 说得好，我从来不 debug，因为我从来不写 bug。”“函数式语言编程跟普通语言不一样。你要先把小块的代码调试好了，问题都找到了，再组合起来。”“当程序有问题却找不到在哪里的时候，说明我思路混乱，我就把它重写一遍……”我很无语，天才就是这样被传说出来的：)

除了由于高傲，Scheme 不提供出错位置的另外一个重要原因，其实是因为它的宏系统。由于 Scheme 的核心非常小，被设计为可以扩展成各种不同的语言，所以绝大部分的代码其实是由宏展开而成的。而由于 Scheme 的宏可以包含非常复杂的代码变换（比 C 语言的宏要强大许多），如果被展开的代码出了问题，是很难回溯找到程序员自己写的那块代码的。即使找到了也很难说清楚那块代码本来是什么东西，因为编译器看到的只是经过宏展开后的代码。如果实现者为了图简单没有把原来的位置信息存起来，那就完全没有办法找到了。这问题有点像有些 C++ 编译器给模板代码的出错信息。

所以出现这样的问题，不仅仅是语言设计者的心态问题，而且是语言自己的设计问题。我觉得 Lisp 的宏系统其实是一个多余的东西，带来的麻烦多于好处。一个语言应该是拿来用的，而不是拿来扩展的。如果连最基本的报错信息都因此不能准确定位，扩展能力再强又有什么意义呢？所以强调一个语言可以扩展甚至变成另外一种语言，其实是过度抽象。一个设计良好的语言应该基本上不需要宏系统，所以 Yin 语言的语法虽然像 Lisp，但我不会提供任何宏的能力。而且由于以上的经历，Yin 语言从一开始就有方便工具的设计做出了努力。

RubySonar：一个 Ruby 静态分析器

在过去一个多月时间里，我大部分时间都在做一个 Ruby 的静态分析叫做 [RubySonar](#)。它使用与 PySonar2 类似的技术，不过针对 Ruby 的语义进行了很多调整。现在这个分析器已经能够支持 [Sourcegraph](#) 的 Ruby 代码搜索和浏览。这比起之前的效果是一个很大的进步。



The screenshot shows a code editor window titled "Definition" containing Ruby code. The code defines a class method `globbs` that changes the current directory to `source`, selects all files, and then performs some file operations. It also defines a class method `process_site` that runs the site's process and handles fatal exceptions by logging an error message and exiting with code 1. The code is syntax-highlighted with colors for different language constructs.

```
def self.globbs(source, destination)
  Dir.chdir(source) do
    dirs = Dir['*'].select { |x| File.directory?(x) }
    dirs -= [destination, File.expand_path(destination), File.basename(destination)]
    dirs = dirs.map { |x| "#{x}/**/*" }
    dirs += ['**']
  end
end

# Static: Run Site#process and catch errors
#
# site - the Jekyll::Site object
#
# Returns nothing
def self.process_site(site)
  site.process
rescue Jekyll::FatalException => e
  puts
  Jekyll.logger.error "ERROR:", "YOUR SITE COULD NOT BE BUILT:"
  Jekyll.logger.error "", "-----"
  Jekyll.logger.error "", e.message
  exit(1)
end
```

在 RubySonar 的帮助下，对于很多 repo，Sourcegraph 可以搜索到比以前多几十倍甚至上百倍的符号，当然代码的使用范例也随之增加了。代码定位的准确性有很大提高，基本不会出现错位的情况了，另外还支持了局部变量的加亮，所以看起来有点像个“静态 IDE”的味道。

由于 RubySonar 比起 Sourcegraph 之前用的基于 [YARD](#) 的分析在速度上有上百倍的提高，我们现在可以处理整个 [Ruby 标准库](#)（而不只是以前的一小部分）。[Ruby on Rails](#) 的结果也有比较大的改善。另外，以前不支持的像 [Homebrew](#) 之类的独立应用，现在也可以分析了。

RubySonar 的静态分析使用跟 PySonar2 相同的跨过程，数据流+控制流分析，而且采用同样的类型推导系统，所以分析的精度是很高的。我还没有跟 Ruby 的 IDE 比较过，不过因为构架的先进性，它应该已经能处理一些现在最好的 Ruby IDE 也搞不定的事情，当然由于时间短，在细节上比起它们肯定也有不足之处。

虽然 Ruby 和 Python 看起来是差不多的语言，为了把 PySonar2 改到 Ruby 上，还是做了不少的工作的。最开头我试图让它们“重用”大部分代码，只是在不一样的地方做一些条件分支进行特殊处理。可是后来发现这样越来越复杂，越来越危险。为了照顾一个语言的特性，很容易破坏掉为另一个语言已经调试好的代码。结果最后决定把它们完全分开，其中共享的代码通过手工拷贝修改。事实证明这个决定是正确的，否则到现在我可能还在为一些莫名其妙的错误伤脑筋。这个经验告诉我，所谓的 DRY (Don't Repeat Yourself) 原则其实有它的局限性。有时候真的是宁愿拷贝粘贴代码也不要共享。

目前 RubySonar 还缺少对 native 库代码的支持，但是由于代码始终保持了简单的原则（RubySonar 只有 7000 多行代码），那些东西会比较容易加进去。感兴趣的 Ruby 用户可以看看自己的 repo 是否已经得到处理，如果没有的话可以来信告诉我，也欢迎给我指出其中存在的问题。

我和权威的故事

每个人小时候心里都是没有权威的，就像每个人小时候也都不相信广告一样。可是权威就像广告，它埋伏在你的潜意识里。听一遍不信，听两遍不信，……，直到一千遍的时候，它忽然开始起作用了，而且这作用越来越强。

消灭广告所造成的幻觉，最好的办法就是去尝试，去实地的考察它。有些虚幻的东西只要你第一次尝试就会像肥皂泡一样破灭掉。可是如果你不主动去接触它，它就会一直在你脑海里造成一种美好神圣的假象。越是得不到的越是觉得美好。很神奇的一个现象就是，权威对人思想的作用其实也跟广告一样。

上大学以前的人因为没有专业，所以还不怎么崇拜权威，大不了追追歌星，影星，球星啥的。而进了大学之后，就会开始对本领域的权威耳濡目染。一遍，两遍，一千遍的听到同学们仰慕某“牛人”或者“大师”的名字，虽然从来没亲身见过，不知不觉就对这人产生了崇拜心理，然后自愧不如。不知不觉的，自己也开始附和这些说法，不自觉地提到这些大师的名字，引用他们说的话作为自己的行动指南。

Donald Knuth, Dennis Ritchie, Ken Thompson, Rob Pike, … 就是通过这些途径成为了很多计算机学生的权威。以至于几十年以后，他们的一些历史遗留下来的糟糕设计和错误思想还被很多人奉为神圣。

Donald Knuth

很多人（包括我）都曾经对 Knuth 和他的 The Art of Computer Programming (TAOCP) 极度崇拜。在我大学和研究生的时候，有些同学花了不少钱买回精装的 TAOCP 全三卷，说是大概不会看，但要供在书架上，镇场子。当时我本着“书非借不能读也”的原则，再加上搬家的时候书是最费力气的东西，所以坚决不买书。我就从图书馆把 TAOCP 借了来。说实话我哪里看的下去啊？那里面的程序都是用一个叫 MIX 的处理器的汇编语言写的。一个字节只有6位，每位里面可以放一个十进制数（不是二进制）！还没开始写程序呢，就开始讲数学，然后就是几十页的公式推导，证明……接着我就睡着了。但我总是听说有人真的看完过 TAOCP，然后就成为了大师。比尔盖茨也宣称：“要是谁看完了 TAOCP，请把简历投给我！”在这一系列的号召和鼓吹之下，我好几次的把 TAOCP 借回来，下定决心这次一定看完这旷世奇书。每次都是雄心勃勃的开始，可从来就没看过开头那段 MIX 机器语言和数学公式。

看不懂 TAOCP 总是感觉很失败，因为看不懂 TAOCP 就成不了“大师”，可我仍然认为 Knuth 就是计算机科学的神，总能从他那学点什么吧，所以又开始折腾他的其他作品。这就是为什么我开始用 TeX，并且成为中国 TeX 界的主要“传教士”之一。为了 TeX，我把 Knuth 的 TeXbook 借回来，从头到尾看了两遍，做完所有的习题，包括最难最刁钻的那种“double bend”习题。接着又开始看 MetaFont Book，开始使用 MetaPost 进行绘图。开头还挺有成就感，可是不多久就发现学会的那些 TeX 技巧到了临场的时候就不知道该怎么用，然后就全都忘记了。这就是为什么我把 TeXbook 看了两遍，可是看完第二遍之后不久还是忘记得一干二净。

师兄师姐看到我用 TeX，说怎么折腾这么过时的玩意儿。我很气愤他们以及国内学术界居然都用 Word 排版论文，就开始针锋相对，写出一系列煽动文章鼓吹 TeX 的种种好处，打击“所见即所得”这种低智商玩意儿。这还不够，又开始折腾 Knuth 设计的 MMIX 处理器，并且认为 MMIX 的寄存器环就是世界上最先进的设计。发现一些无关紧要的小错，就给 Knuth 发 email，居然拿到两张传说中的“Knuth 支票”，并且一度引以为豪。当然像所有拿到 Knuth 支票的人一样，你是不会去兑现它的，甚至有人把它们像奖状一样放在相框里。我还没那么疯狂，那两张支票一直在它们原来的信封里。多年以后我到美国想兑现那支票的时候，发现它们已经过期了。



当你心里有了这样的权威，其他人的话你是不可能听得进去的，就算他们其实比你心目中的权威更具智慧也一样。在清华的时候我有时候去姚期智的小组听串讲。有一次请来了美国某大学一个教授讲算法，不知道怎么的我就跟他聊起 TAOCP，大概是想请教他如何学习算法。他跟我说 Knuth 的书已经比较过时了，你可以看看 MIT 的那本《算法导论》。可是这位教授的名气怎能和 Knuth 相比，这话我倒是没有听进去，仍然认为 TAOCP 隐藏了算法界最高的机要，永恒的珍宝。

在清华的时候我很喜欢一门叫做“计算几何”的课，就经常跟那门课的老师交流思想。有一次我在 email 里面提到 Donald Knuth 是我的偶像，那位老师很委婉的回复道：“有偶像很好啊，Knuth 也曾经是我的偶像。”我对“曾经”这两个字感到惊讶：难道这意味着 Knuth 现在不是他的偶像了？在我执意的询问之下他才告诉我，其实世界上还有很多更聪明的人，Knuth 并不是计算机科学的一切。你应该多看看其他人的作品，特别是一些数学家的。然后他给了我几个他觉得不错的人的名字。

现在回想起来，这些话对我是有深远作用的。那位老师虽然在系里的“牛人”们眼里是个研究能力（也就是发 paper 能力）不强的人，但是他却对我的人生转折有着强有力的作用。他引导了我去追寻自己真正的兴趣，而不是去追寻虚无的名气。我发现很多人都在为着名气而进行一些自己其实不感兴趣的事情，去做一些别人觉得“牛气”的事情。我真希望他们遇到跟我一样的好老师。

在现在看来，Knuth 的 TAOCP 就是所谓的“神圣的白象”(white elephant)。大家都把它供起来，其实很少有人真的看过，却要显得好像看过一样，并且看得津津有味。这就让试图看懂它的人更加自卑和着急，甚至觉得自己智商有问题。别人都看过了，我怎么就看不懂呢？其实 TAOCP 里面的大部分算法都不是 Knuth 自己设计的，而且他对别人算法的解释经常把简单的问题搞得很复杂。再加上他执意要用汇编语言，又让程序的理解难度加倍。

有一句话说得好：“跟真正的大师学习，而不是跟他们的徒弟。”如果你真的要学一个算法，就应该直接去读那算法的发明者的论文，而不是转述过来的“二手知识”。二手的知识往往把发明者原来的动机和思路都给去掉了，只留下苍白无味，没有什么启发意义的“最后结果”。确实是这样的，多年以后当我看见 Knuth 计划中的几卷新的 TAOCP 的目录时，发现其中大部分的东西我已经通过更容易的方式学到了，因为我找到了这些知识的源头。

所以之前的那位访问清华的教授说的其实是实话，Knuth 真的落伍了，可是就算在美国也少有人知道或者承认这个情况。有一次看一个对世界上公认最厉害的一些程序员的采访，包括总所周知的一些大牛，以及 ML 的设计者 Robin Milner, Haskell 的设计者之一 Simon Peyton Jones 等人。也不知道采访者是什么心理，在对每个人的采访中他都问，你看过 TAOCP 吗？大部分人都说看过，真是了不起的巨著，很重要啊云云。只有 Robin Milner (如果我没记错的话) 比较搞笑，他说我希望我看过去，但是可惜实在没时间。我一直把 TAOCP 垫在我的显示器下面，这样我工作时就可以一直看着它们 :)

Knuth 说“premature optimization is the root of all evil”，然而他自己却是非常喜欢用 premature optimization 的人。他的代码里到处是莫名其妙的小聪明，小技巧。把代码弄得难懂，实际上却并没有得到很多性能

的提高。有一次看 MMIX 处理器的模拟程序，发现他用来计算一个寄存器里的“1”的个数的代码非常奇怪。本来写个循环，或者用那种从末位减 1 的做法就可以了，结果他的代码用了 Programming Pearls 里面一个古怪的技巧，费了我半天时间才看懂，后来我发现这个技巧其实还不如最简单的方法。就是这些细小却又蹊跷的做法，使得 Knuth 的代码用细节掩盖了全局，所以到最后我其实也没从大体上搞懂一个处理器的模拟器应该如何工作。直到后来到 Indiana 学习了程序语言的理论之后我才发现，其实处理器模拟器（以至于处理器本身）的工作原理很简单，因为它就是一个机器代码的解释器。使用跟高级语言解释器同样的结构，你可以比较容易的写出像 MMIX 模拟器那样的东西。

Knuth 最重要的一个贡献恐怕是程序语言的 parsing（语法分析），比如 [LR parsing](#)，然而 parsing 其实是一个基本不存在的人造问题。它的存在是因为人们的误解，以为程序语言需要有跟人类语言一样的语法，所以把程序语言搞得无端的复杂和困难。如果你把语法简化一下，其实根本用不着什么 LR, LALR。我最近给我自己设计的语言写了一个 parser，从头到尾只花了两个小时，500 行 Java 代码，包括了从 lexer 一直到 AST 数据结构的一切。完全手写的代码，根本没用任何复杂的 parsing 技术和 YACC 之类的工具，甚至正则表达式都没有用。之所以可以这样，因为我的语法设计让 parsing 极其容易，比 Lisp 还要容易。Knuth 过度的强调了 parsing。他的误导使得很多人花了几十年时间来研究 parser，到现在还在不时地提出新的技术，用于设计更加复杂的语法。何必呢？这只会让程序员和编译器都更加痛苦。如果这些人把时间都花在真正的问题上，那今天的计算机科学不知道要美好多少。

几乎每一本编译器教材都花大量篇幅来讲述 DFA, NFA, lexing, LL, LR, LALR…… 几乎每个学校的编译器课程都会花至少 30% 的时间来做 parser，折腾 LEX, YACC 等工具，而对于编译器真正重要的东西却没有得到很多的训练。这就是为什么 [Kent Dybvig 的编译器课程](#)如此有效，因为 Scheme 的语法非常简单，我们根本没有花时间来做 parser。我们的时间用在了思考真正的问题：做优化，实现尾递归，高阶函数…… 很多语言梦寐以求却又做不好的东西。这样的课程给了我可以发挥自己潜力的余地，我的课程编译器里面具有大量的独创写法，我的 X64 机器代码生成器生成极其短小的代码，让 Kent Dybvig 都在背地里琢磨是怎么回事。这些东西到现在也许仍然是世界上最先进的技术。

一个人的思维方式似乎决定了他设计的所有东西。Knuth 的另一个最重要的发明，文学编程（Literate Programming）其实也是多此一举，制造麻烦。文学编程的错误在于认为程序语言应该像人类语言，应该适应所谓的“人类思维”。然而程序语言却是在很多方面高于人类语言的，它不应该受到人类语言里的糟粕的影响。把程序按照 Knuth 的方式分开在不同的文章段落里，造成了代码之间的关系很难搞清楚，而且极其容易出错。这个错误与“Unix 哲学”的错误类似，把程序作为一行一行的文本，而不是一个像电路图一样的数据结构。我不想在这里细说这个问题，对此我专门写了一篇[文章](#)，讲述为什么文学编程不是一个好主意。

TeX 其实也是异常糟糕的设计。它过度的复杂，很少有人搞得懂怎么配置。经常为了一个简单的效果折腾很久，然后不久就忘了当时怎么做的，回头来又得重新折腾。原因就是因为 TeX 的设计缺乏一致性，特殊情况太多，而且组合（compose）能力很差。所以你需要学太多东西，而不是跟象棋一样只需要学习几个非常简单的规则，然后把它们组合起来形成无穷的变化。

在程序语言设计者看来，TeX 的语言是世界上最恶劣的设计之一，但如果没用这个语言，它也许会更加糟糕。其实 TeX 之所以有一个“扩展语言”，有一个鲜为人知的小故事。在最早的时候 Knuth 的 TeX 设计里并没有一个语言。它之所以有一个语言是因为 Scheme 的发明者 Guy Steele。Knuth 设计 TeX 的那个时候 Steele 碰巧在斯坦福实习。他听说 Knuth 要设计一个排版系统，就建议他设计一个语言，以应付以后的扩展问题。在 Steele 的强烈建议和游说之下，Knuth 采纳了这个建议。可惜的是 Steele 并没能直接参加语言的设计，在短短的一个夏天之后就离开了斯坦福。

Knuth 的作品里面有他的贡献和价值，TeX 的排版算法（而不是语言）也许仍然是不错的东西。可是如果因为这些好东西爱屋及乌，而把他所推崇的那些乱七八糟的设计当成神圣的话，那你自己的设计就逃脱不出同样的思维模式，让简单的事情变得复杂。仍然对 TeX 顶礼膜拜的人应该看一下 [TeXmacs](#)，看看它的作者是如何默默无闻的，彻彻底底的超越了 TeX 和 Knuth。

在我看来，Knuth 是个典型的精英主义者，他觉得自己做的都是最好，最有“格调”的。他利用自己的权威和特立独行来让用户屈服于自己繁复的设计，而不是想法设计出更加易用的工具。TeX 的版本号每次更新都趋近于圆周率π，意思是完美，没有 bug。他奖励大额的支票给发现 TeX 代码里 bug 的人，用于显示自己对这些代码的自信，然而他却“冰封”了 TeX 的代码，不再填加任何新东西进去，也不再简化它的设计。当然了，如果不改进代码，自然就不会出现新的 bug，然而它的设计也就因此固步自封，停留在了几十年以前。更奇怪的是，“TeX”这个词居然不按照正常的英语发音逻辑读成“teks”。每当有人把它“读错”，就有“高手”打心眼里认为你是菜鸟，然后纠正：“那个词不读 teks，而要读‘特喝’，就像希腊语里的 chi，又像是苏格兰语的 loch，德语的 ach，西班牙语的 j 和俄语的 kh。”也许这就叫做附庸风雅吧，我是纯种的欧洲人！;-) 当一个软件连名字的发音都这么别扭，这么难掌握，那这个软件用起来会怎样？每当你提到 TeX 太不直观，就有人跟你说：“TeX 是所想即所得，比你的所见即所得好多了！”可事实是这样吗？看看 TeXmacs 吧，理解一下什么是“所见即所得+所想即所得”二位一体。

我跟 Knuth 的最后一次“联系”是在我就要离开清华的时候。我从 email 告诉他我觉得中国的研究环境太浮躁了，不是做学问的好地方，想求点建议。结果他回纸信说：“可我为什么看到中国学者做出那么多杰出的研究？计算机科学不是每个人都可以做的。如果你试了这么久还不行，那说明你注定不是干这行的料。”还好，我从来没有相信他的这段话，我下定了决心要证明这是错的。多年的努力还真没有白费，今天我可以放心的说，Knuth 你错了，因为我已经在你引以为豪的多个方面超过了你。

Unix

Unix 的创造者们是跟 Knuth 非常类似的权威，他们在我心目中也曾经占据了重要的位置，以至于十年前我写了一篇文章叫《完全用 Linux 工作》，大力鼓吹 Unix 的“哲学”，甚至指出 Linux 不能做的事情就是不需要做的，并且介绍了一堆难用的 Unix 工具，引得很多人去折腾。可如果你知道我现在对 Unix 的态度，肯定会大吃一惊，因为在经过努力之后，我成功的“忘记”了 Unix 的几乎一切，以至于本科刚毕业的学生都会以为我是脑盲，并且以为可以在我的面前炫耀自己知道的 Linux 技巧。他们不会明白，在我心里 Unix/Linux 的设计是计算机软件界目前面临的大部分问题的罪魁祸首，而他们显示给我看的，只不过是 Unix 的思想和精英主义给程序员造成的精神枷锁。其实我并不会忘记 Unix 的设计，但我已经下意识的以熟悉 Linux 的奇技淫巧为耻，所以很多时候我即使知道也要装作不知道。因为我是机器的主宰，而不是它的奴隶，所以我总是想办法让机器去帮我做更多的事，帮我记住那些无聊的细节，而不是去顺从它的设计者所谓的“哲学”。

评论 Unix 和它的后裔们总是一件尴尬的事情，因为你提到它们的任何一个缺点，都会被很多人认为是优点。GNU 的含义是“GNU is Not Unix”，但很可惜的是 GNU 和 Linux 的设计从来没有摆脱过 Unix 思想的束缚。Unix 的内存管理，进程，线程，shell，进程间通信，文件系统，数据库……几乎都是很蹩脚的设计。所谓的“Unix 哲学”，也就是进程间通信主要依靠无结构字符串，造成了一大批过度复杂，毛病众多的工具和语言的产生：AWK，sed，Perl，…… Unix 的内存管理是按“页”而不是按“结构”分配，相当于把内存分配的任务完全推给应用程序。而且允许任意的指针操作，这就像给每个老百姓一把爱走火的枪。可是又想要“安全”，自相矛盾。没办法，不得不强制进程数据空间完全隔离，使得进程间无法直接传递数据结构。进程和线程上下文切换开销过大，造成了使用大规模并发或者分布式计算的瓶颈，导致了 goroutine 和 node.js 等“变通方法”的产生。把数据无结构的存储在文件里，无法有效的查找数据，造成了关系式数据库等过度复杂的数据解决方案的产生。再加上后来 WEB 的设计，现在的网站基本上就是补丁加补丁，一堆堆的 hack。

“Unix 哲学”貌似也有好的部分，比如“每个程序只做一件事，多个程序互相合作。”然而，这个所谓的哲学其实就是程序语言（比如 Lisp）里面的模块化设计。它当然是好东西，然而这些思想被 Unix 偷来之后，有其名而无其实。很少有 Unix 程序真正只做一件事的，而且由于字符串这种通信机制的不可靠，它们之间其实不能有效地合作。有时候你换了一个版本的 make 或者 sed 之类的工具，你的 build 就莫名其妙的出问题。这就是为什么有的公司请了专门的所谓“build engineer”，因为高级别的程序员不想为这些事情操心。Lisp 程序员早就明白这个道理，所以他们尽一切可能避免使用字符串。他们设计了 S 表达式，用于结构化的传输数据。实际上 S 表达式不是“设计”出来的，它是每个人都应该首先想到的，最简单的可以表示树结构的编码方法。Lisp 的设计原则里面有一条就是：Do not encode。它的意思是，尽量不要把有用的数据编码放进字符串。Unix 的世界折腾来折腾去，XML，CORBA，…… 最后才搞出个 JSON，然而其实 JSON 完全不如 S 表达式简单和强大。Unix 就像一个肿瘤，它让人们放着最好的解决方案几十年不用，不断地设计乌七八糟的东西用来取代乌七八糟的东西。这些垃圾对人有很大的洗脑作用。前段时间我说 S 表达式比 JSON 简单，有人居然跟我说 JSON 好些，因为它结构的 field 是“无顺序”的。这让我相当无语，因为一个编码方式有没有顺序完全取决于你如何解释它。从这个意义来讲，S 表达式可以是有顺序，也可以是没有顺序的。

Unix 喜欢打着“自由”和“开源”的旗号，可是它的历史却充满了政治，宗教，利益冲突和对“历史教科书”的串改。几乎所有操作系统课本的前言都会提到 Unix 的前身 Multics，而提到 Multics 的目的，都是为了衬托 Unix 的“简单”和伟大，接下去基本上就是按部就班的讲 Unix 的设计，仿佛 Unix 就是世界上唯一的操作系统一样。课本会告诉你，Multics 由于设计太复杂，试图包罗万象，最后败在了 Unix 手下。可是如果你仔细了解一下 Multics 的历史，就会发现最后一台 Multics 机器直到 2000 年还在运行，拥有 Unix/Linux 到现在还没有的先进而友好的特性，并且被它的用户所爱戴。Multics 的设计并不是没有问题（对比一下 Lisp Machine 和 Oberon），但是相比之下，Unix 的设计一点都不简单。Unix 抄了 Multics 最好的一些思想，有些没有抄得像，然后又引入了很多自以为聪明的糟粕。可是 Unix 靠着自己病毒一样的特征，迅速占领了市场。Unix 最开头是开源和免费的，但是后来 AT&T 发现这里面有利可图，所以就收回了使用权，并且开始跟很多人打官司。AT&T 的邪恶比起微软来，真是有过之而无不及。

Unix 的很多设计是如此龌龊，很多人却又由于官僚的原因不得不用它。以至于 Unix 出现的早期怨声载道，有人甚至组织了一个 mailing list 叫“[Unix 痛恨者](#)”(Unix Haters)。你很有可能把这些人当成菜鸟，可是这些人其实都用过更好的操作系统，有的甚至设计实现过更好的操作系统甚至程序语言。最后他们的叫骂声被整理为一本书，叫做 [Unix Hater's Handbook](#)。让人惊讶的是，这本书有一个“反序言”(anti-forward)，作者正是 Unix 和 C 语言的设计者之一，Dennis Ritchie。这个反序言说，Unix 这座设计缺乏一致性的监狱会继续囚禁你们，聪明的囚犯会从它里面找到破绽，可惜的是自由软件基金会 (FSF) 会建造跟它完全兼容的监狱，只不过功能多一些。拥有三个 MIT 学位的记者，微软的研究员，Apple 的高级科学家可能还会对这座监狱的“规矩”贡献一些文字。从这些文字里，我看到了一个炫耀武力的暴君，看到了赤裸裸的权威主义和教条主义。

可惜的是在软件的世界里任何糟糕的设计都可以流行，只要你的广告做得好，只要你的传教士够多。一知半解的人（比如十年前的我）最喜欢到处寻找“新奇”的东西，然后开始吹嘘它们的种种好处，进而成为它们的布道者。再加上大学计算机系的“紧跟市场”的传统，不幸的事情发生了：Unix 和它的后裔们几乎垄断了服务器操作系统的市场。由于 Unix 的垄断，现在的软件世界基本上建立在一堆堆的变通之上，并且固化之后成为了“[珍珠](#)”。公司里，学校里，充满了因为知道一些 Unix 的“巧妙用法”而引以为豪的人，殊不知他们知道的只是回避一些蹩脚设计的小计俩。程序员有太多的特例和细节需要记忆，不但不抱怨，还引以为豪。很少有人想过如何从根本上解决问题，历史的教训很少有人吸取，以至于几十年前犯过的错误还在重现。Unix 的最大贡献，恐怕就是制造了大量的工作岗位——因为问题太多太麻烦，所以需要大量的人力来维护它的运行。

现在看来，Unix 当初就是依靠《皇帝的新装》里织布工的办法封住了大家的嘴。皇帝的织布工们说：“愚蠢或者不称职的人都看不见这件衣服。”Dennis Ritchie 说：“Unix 是简单的，但只有天才才能理解这种简单。”看出来了了吗？你不敢说 Unix 的设计太乱太复杂，因为这话一出口，立马会有人引用 Dennis 的话说，是你自己不够天才，所以不理解。当然了，这就意味着他比你聪明，因为只有天才才能理解这种简单嘛。哎，这种喜欢显示自己会用某种难用工具的人实在太多了。你不敢批评这些工具对用户不友好，因为你立即会被鄙视为菜鸟。

Dennis Ritchie 去世了。死者长已矣，可是有些他的崇拜者在那个时候还要煽风点火，拿他的死与 Steve Jobs 的死来做对比，把像这样的照片四处转帖，好像 Steve 死错了时间，抢了 Dennis 的风头似的。然后就有人写一些这样的文章，把世界上的所有系统，所有语言都归功到 Dennis 和 Unix 身上。看到这些我明白了，所谓的“天才”就是这样被造出来的。在我看来这些是很滑稽的谬论，就像是在说有人拿一把很钝的剪刀做出了一件精美的衣服，所以这剪刀立下了汗马功劳。其实这人一边裁布一边在骂这剪刀，心想妈的这么难用，快点做出这衣服，卖了钱买把好点的！

用了这么久 Apple 的产品，平心而论，虽然它们并不完美，然而它们并不是 Unix 的翻版，它们做出了摆脱 Unix 思想束缚的努力。它们本着机器为人服务的原则，而不是把人作为机器的奴隶。Mac 的很多内部设计跟 Unix 有着本质的不同。然而就是这样的系统，被 Dennis Ritchie 在他的[反序言](#)里面蔑称为“以 Sonic the Hedgehog 作为智力主题和交互设计基础的系统”。

有谁知道，在那同样一段时间里，Lisp 的发明者 John McCarthy，ML 的发明者 Robin Milner，都相继去世了呢？那个时候我只是在 mailing list 看到有人发来简短的消息，然后默默地思念他们给我带来的启迪。我们没有觉得 Steve Jobs 的死抢了他们的风头，因为他们不需要风头。死就是要安安静静的，让知己者默哀已经足矣。出现这种事情恐怕不能怪 Dennis Ritchie 自己，然而这些 Unix 的崇拜者们，真的应该反省一下自己的做法了。

Unix 的设计者们曾经在我的心里占据了一席之地，可是现在觉得他们其实代表了反动的力量，他们利用自己的影响力让这些糟糕的设计继续流传，利用人们的虚荣心，封住大部分人的嘴，形成教条主义，让你认为 Unix 的设计是必须学习的东西。很多人成为了 Unix 的传教士和跟屁虫，没有什么真实水平，就会跟着瞎起哄，把 Unix 设计者的话当成教条写进书里。可是他们的权威和名气是如此之大，让我在很多人面前只能无语。

Go 语言

现在，同样这帮 Unix “大牛”们设计了 Go 语言，并且依仗自己的权威和 Google 的名气大力推广。同样的这帮跟屁虫开始使用它，吹捧它，那气势就像以为 Go 可以一统天下的样子。真正的程序语言专家们都知道，Go 的设计者其实连语言设计的门都没摸到。这不是专家们高傲，他们绝不会鄙视和嘲笑一个孩子经过自己的努力做出一个丑陋的小板凳。他们鄙视，他们嘲笑，因为做出这丑陋小板凳的不是一个天真的小孩，而是一些目空一切的人，依仗着一个目空一切的公司。他们高举着广告牌，试图让全人类都坐这样丑陋的板凳。

跟当年设计 Unix 时一个德行，不虚心向其它语言和系统学习经验教训，就知道瞎猜瞎撞。自己想个什么就是什么，但其实根本就不知道自己在干什么。把很多语言都有的无关紧要的功能（比如自动格式化代码）都吹嘘成是重大的发明，真正重要的东西却被忽略。Go 语言的设计在很多方面都是历史的倒退，甚至犯下几乎所有其他语言都没有的[低级错误](#)。在语法上大做花样，却又搞得异常丑陋，连 C 和 Java 都不如。自己不理解或者实现难度大点的东西就说是不是需要的，所以连很多语言支持的 parametric type（类似 Java generics）都没有，以至于没法让程序员自定义通用数据结构，只好搞出一堆特例（比如 map，make，range）来让程序员去记。这些做法都跟 Unix 如出一辙。

Go 语言最鲜明的特征就是 goroutine，然而这个东西其实每个程序语言专家都知道是什么。有些语言比如 Scheme 和 ML 提供了 first-class continuation (call/cc)，可以让你很容易实现像 goroutine 这样的东西，甚至实现硬件中断的“超轻量线程”。至于 Go 那种“基于接口”的类型系统设计，我在很多年前就已经试验过，并且寄予了很大的希望。结果最后经过很多的研究和思索后发现有问题，于是放弃了这个想法。很显然，我不是第一个在这个问题上失败的人，很多语言专家在使用 parametric type 以前都试图过做这种基于接口的设计，结果最后发现不是什么好东西，放弃了。然而 Go 的设计者却没有学到这些失败教训，反而把它当成宝贝。一个很显然的问题是，在 Go 里面你经常会使用“空接口”(interface{})，用来表示所有类型。这就像使用 C 的 void 指针一样，有着静态类型系统麻烦，却失去了静态类型系统的好处。

每当你提到 Go 没有 parametric type，Go 的拥护者们就说“我看不到这有什么用处”，就像一些非洲土著跟你说“我看不到鞋子有什么用处”一样。他们利用人们对 Java 的繁复和设计模式的仇恨，让你抛弃了它里面的少数好东西。其实 Java generics 不是 Java 首先有的。它的主要设计者其实包括 Haskell 的设计者之一 Philip Wadler。这种 parametric type 很早就出现在 ML，Haskell 等语言里面，是非常有用的东西。

每当受到批评，Go 的拥护者们就托词说，Go 是“系统语言”(systems language)。这里潜在的前提就是，认为 Unix 就是唯一的“系统”，而 C 就是在 Go 以前唯一的“系统语言”，好像其他语言就写不出所谓的“系统”似的。而事实是，在 C 诞生十年以前，人们就已经在用 Algol 60 这样的高级语言来写操作系统了。由于先天不足却又大力推广，所以 Go 的很多缺陷基本已经没法修补了。这样的语言一旦流行起来就会像 Unix 一样，成为一个无休止的补丁堆。如果像 Java 或者 Haskell 这样的语言还值得批评的话，对 Go 语言的设计者我只能说，去补补课吧。

Cornell

可是权威和名气的威力还是很大的。虽然 Knuth 在我心目中的位置不再处于“垄断地位”，世界上可以占据我心里那个位置的人和事物还很多。在离开清华之后我申请了美国的大学。也许是天意也许是巧合，只有两所大学给了我 offer：Cornell 和 Indiana，而我竟然先后到了这两所大学就读。

说实话，Indiana 给了我比 Cornell 更好的 offer。Cornell 给我的是一个 TA 的半工读职位，而 Indiana 给我的是一个不需要工作白拿钱的 fellowship。说实话我从来没有搞明白 Cornell 这样的“牛校”怎么会给我这样的人 offer，GPA 一般，paper 很菜，而 Indiana 却是真正在乎我的。Indiana 的 fellowship 来自 GEB 的作者 Doug Hofstadter。他从 email 了解到我的处境和我渴求真知的愿望之后，毅然决定给我，一个素不相识的人写推荐信。后来我才发现那 fellowship 的资金也是他提供的。

可是 Indiana 和 Hofstadter 的名气哪里能跟 Cornell 的号称“CS前五”相比啊？Indiana 的 offer 晚来了几天。当收到 Indiana 的 offer 时，我已经接受了 Cornell。Hofstadter 很惊讶也很失望，因为他以为我一定会做他的学生，可是听说我接受了 Cornell 的 offer，他也不知道该怎么办。我只隐约的记得他告诉我，学校的排名并不是最重要的东西……

名气和权威的力量是如此之大，它让我不去选择真正欣赏我并且能给我真知的人。有时候回想起来，我当时真的是在寻找真知吗？我明白什么叫做真知吗？

Cornell 给了我什么呢？到现在想起来，它给我的东西恐怕只有教训，很多的教训。TA 的工作可不是那么好做的，基本就是苦力，你甚至会怀疑他们录取你就是为了利用你的廉价劳动力。我第一次做 TA 就是一个 200 多人在阶梯教室上的大课，教最基本的 Java 编程。虽然有好几个 TA，但任务还是很繁重。讲课的人不是教授，而是专职的讲师。这种讲师一般得靠本科生的好评来谋生，所以虽然在学术上没什么真本事，对学生真可谓是点头哈腰，服务周到。这就苦了各位 TA 了，作业要你设计，还要设计得巧妙，要准备好标准答案，之后还要批作业，批得你头脑麻木，考试要监考，之后还要批试卷。每周还得抽好几个小时来做 office hour，给学生答疑。然后你还有自己要上的课，自己的作业，自己的考试。每当考试的时候都很紧张，因为你得准备自己的考试，还要为学生的考试多做很多工作。

如果真的学到了东西，这么辛苦也许还值得，可是那些教授真的是想教会你吗？有人打了个比方，说 Cornell 说要教你游泳，就把你推到水池里，任你自己扑腾。当你就要扑腾上岸时，他在你头上用榔头一砸，然后继续等你上岸。当你再次快要扑腾上岸时，他又举起一块大石头扔到你头上，这样你就可以死了，可是 Cornell 仍然等着你游上岸……这就是对我在 Cornell 的经历的非常确切的比喻。

我在一篇老的博文里面提到过，Cornell 的学生，包括博士生，一上课就抄笔记，一天到晚都在赶作业。可其实 Cornell 不只是爱抄笔记的学生的天堂，而且是崇拜权威者的天堂。即使你不是那么的崇拜权威，你不可避免的会被一群像朝圣者一样的人围在中间，在你耳边谈论某某人多么多么的牛。不管你向同学打听哪一个教授，得到的回答总是：“哇，他很牛的！”然后你就去上了他的几节课，觉得不咋的嘛，可是人家就说那是因为你不理解他的价值。这种气氛我好像在另一个地方感觉到过呢？啊对了，那是在 Google。这样的气氛也许并不是偶然，Cornell 的大部分 PhD 同学当时的最大愿望，就是毕业后能去 Google 工作。当然，后来 Facebook 上升成为了他们的首选。值得一提的是，Indiana 其实是更有个性的地方。我在 Indiana 的同学们一般都把去 Google 工作作为最后的选择之一。有一次一个刚来不久的学生问，如何才能进入 Google 工作？有个老教授说，那个容易，Google 招收任何能做出他们项目的人！



Cornell 的研究可以用“与时俱进”来形容，什么热门搞什么。当时 Facebook 和社交网络正在“崛起”，所以系里最热门的一个教授就是研究社交网络的。我去听过他几堂课，他用最容易的图论算法分析一些社交网络数据，然后得出一些“理论”。其中好些结论实在太显然了，我觉得街上的卖菜大妈都能猜到，还不如研究星际争霸来得有意思点。可是 Facebook 名气之大，跟着这位教授必然有出路啦，再加上有人在耳边煽风点火，所以好多的学生为做他的 PhD 挤破了头皮，被刷下来的就只好另投门路了。每次新来一个教授都会被吹捧上天，说是多么多么的聪明，甚至称为天才。然后就有一群的人去上他的课，试图做他的学生。结果人家每节课都是背对学生面朝黑板，喃喃自语，写下一堆堆的公式和证明，一堂课总共就没回过几次头。下面的人当然是狂抄笔记，有的人甚至带着录音笔，生怕漏掉一句话。上这样的课还不如干脆把板书打印出来让大家自己回家看。人多了竞争也就难免了。上课的同学们就开始勾心斗角，三国演义的战术都拿出来了。作业做不出来就来找你讨论，等你想讨论了就说自己也没做出来。没听懂偏要故作点头状，显得听懂了，让你觉得有压力。自己越是喜欢的教授就越是说他不咋的，扯淡，然后就自己去跟他。自己不喜欢的教授就告诉你他真是厉害啊，只可惜人家不要我。直到两年后我离开 Cornell 之前，还有好些同学因为没找到

教授而焦头烂额。因为两年内没有找到导师的 PhD 学生，基本上等于必须退学。

当我离开 Cornell 之后，有一位国内的学生给我发 email 套磁（从系里主页上找到我的地址），问我 Cornell 情况如何。我告诉他我都已经走人了，并且告诉了他我的感觉，一天到晚抄笔记赶作业之类的。然后又问我一个刚毕业的 PhD 的情况，我说他水平不咋的，博士论文我看过了，很扯淡，解决一个根本不存在的问题。他对我说的话有点惊讶，但还是将信将疑。为了确保万无一失，他在 visiting day 的时候专程去 Cornell 考察了一下。回去又给我 email，说见到好多牛人啊，大开眼界，哪里像你说的那么不堪。还说跟那位 PhD 的导师谈过话，真是世界级的牛人那，他的博士论文也是世界一流的。我就无话可说了，仁者见仁，智者见智，随他去吧，哎。

结果两年之后，我又收到这位同学的 email，说他在 Cornell 还没找到导师，走投无路了，问我有没有办法转学。

图灵奖

说到这里应该有人会问这个问题，我是不是也属于那种没找到导师走投无路的人。答案是，对的，我确实没有在 Cornell 找到可以做我导师的人。然后我就猜到有人会说，就知道王垠水平不行嘛，没搞定导师，被迫退学，哈哈！可是事情其实没他们想象的那么简单。作为一个 PhD 学生，不仅必须精通学术，而且要懂得政治和行情。哦错了，其实不精通学术也行的，但是一定要懂得政治和行情！可是由于学生之间的窝里斗，他们之间的信息互通程度，是没法和教授之间的信息互通程度相比的。这就造成了“学生阶级”在这场信息战上的劣势，总是被动的被教授挑选，而不能有效地挑选适合自己的教授。

进入 Cornell 之后我上了一门程序语言的课，就开始对这些东西入迷。可是由于“与时俱进”，Cornell 的研究方向并不是那么平衡的发展的，其实是很畸形的发展。程序语言领域的专家们早已因为受到忽视而转移阵地，剩下一群用纸和笔做扯淡理论的。说实话，在历史上程序语言方向曾经是 Cornell 的强项，出现了一些很厉害的成果。可是当我在 Cornell 的时候，只剩下两个名不见经传的教员，一个助理教授，一个副教授。其实 Robert Constable 也在那里，可惜的是他做了 dean 之后已经没空理学生了，以至于我两年之后都不知道这个人的存在。我当时也不知道 Cornell 有过这段历史，看不到它的研究重心的移动趋势。

我不喜欢那个副教授搞的项目，大部分是在 Java 上面加上一些函数式语言早就有的功能。可是人家做的是热门语言，所以拉得到资金，备受系里亲睐，他的学生们也比较趾高气昂。初次见面的时候，我跟他的一个学生说了我的一个想法，他说：“你那也能叫研究吗？待会儿我给你看看什么是真正的研究！”其实那只是我的一个微不足道的想法，我也没说那是研究啊。只是随便聊一下而已就这么激动——何况你们那些 Java 的东西能算是研究？我是不可能跟那样的人合作的，所以我就跟那个助理教授做了一点静态分析的项目。当然我们分析的也不是什么好东西，是用 Fortran 写的 MPI 程序。不过说实话，那个助理教授其实挺有点真知灼见，他有几句话现在仍然在指引我，防止我误入歧途。其中一句话是针对我对 pi-calculus 的盲目崇拜说的：“那些理论其实不管用的。最好是针对自己的问题，自己动脑筋想。”他也是很谦虚很善良的人，可是好人不一定有好报的。后来他没有拿到 tenure 职位，不得不离开 Cornell 加入了工业界，而我就失去了最后一个有可能在程序语言方向做我的导师的人。

没办法，我就开始探索其它相关领域的教授，比如做数据库的，做系统的，看他们对相关的语言设计是否感兴趣。可惜他们都不感兴趣，而且告诉我程序语言领域太狭窄了。我当时还将信将疑，甚至附和他们的说法，可是现在我断定他们都是一知半解胡说八道。如果这些人虚心向程序语言专家请教，现在数据库和操作系统的设计也不会那么垃圾，关系式，SQL，NoSQL，……一个比一个扯淡。没有办法，我就开始探索其他的方向，开始了解图形学和数值分析等东西，进展很不错。可是终究我还是发现，我不喜欢图形学和数值分析所用的语言。我想制造出更好的程序语言来解决这些问题。可是跟教授们谈这些想法的时候就感觉是在对牛弹琴，他们完全不能理解。后来我发现，教授们貌似不喜欢有自己想法的学生，他们更希望找到愿意“打下手”的学生，帮助实现他们自己的想法。

这就让我走到了跟那位向我打听 Cornell 情况的同学差不多的局面，真是心里有许多的苦却没有人可以理解。这时候我想到了系里的一些德高望重的教授，比如得过图灵奖的人，也许这些顶级的大牛会给我指出方向。于是我就联系到一位图灵奖得主，说想找他聊聊。我说我感兴趣的东西 Cornell 貌似并不重视和发展。Cornell 的校训是“any person, any study”，而我想 study 的东西却得不到支持。最后我谈了一下我对 Cornell 的总体感受。我说我觉得大家上课死记硬背，不是很 intellectual，我不是很确定学术界是否还保留有它原来的对智慧和真知的向往。

我很诚恳的告诉了他这些，只是希望得到一些建议。结果他不但没有理解任何一点，而且立马开始用质问的语气问我，你成绩怎么样？考试都通过了没有？哎，说白了就是想搞清楚你是不是成绩不好没人要。怎么就跟高中教导主任一样。于是乎那次谈话就这样不了了之。可是没有想到，这次谈话就造成了我最后的离别。在学生们互相之间勾心斗角，不通信息的同时，系里的教授们其实背后都是“通气”的。他们根本不懂得如何教学，就知道拿作业和考试往学生头上砸，幸存下来的就各自挑去做徒弟，挨不住的就打发掉。这算盘打得真是妙啊。我也不知道他们是什么机制，每个学生对哪些教授感兴趣，表现如何，他们貌似都了如指掌，貌似背后有个什么情报网。然后系里的教授们不知道怎么的，仿佛就都知道有这样一个不知趣的学生，居然敢说学术界的坏话！

大地震前夕的天空总是异常的美。我竟然在过道里看到那位图灵奖教授对我点头致意并且微笑，以前做 TA 时把我呼来唤去还横竖不满意的教授也对我笑脸相迎。我仿佛觉得那一席话打动了那位德高望重的教授，再加上在图形学和数值计算的扎实进展，也许我的学术生涯有了转机。可是，我那一次真正的领悟了什么叫做所谓的“笑里藏刀”。

由于那个学期上的图形学还有矩阵计算的课成绩都不错，我心想应该能找这两门课的授课教授的其中一个做导师吧。再加上那些貌似友好的笑容……所以没想很多，居然过了一个非常快乐的寒假。没有任何前兆，没有任何直接的通知（email，电话），一封纸信不知道是什么时候默默地到了我在系里的“信箱”——一个我基本上从来不看的，系里用来塞广告信息的信夹子里，直到下一个学期开始的时候（2月份）我才发现。信是系主任写的，大概就是说，由于你的表现，我们觉得 Cornell 不是适合你的地方……

说得对，我也觉得 Cornell 不适合我。我本来就有想走的意思，可我一般呆在一个地方就懒得动。如果你们早一点告诉我这个，比如12月以前，我还可以申请转学到其它学校。可是都 2 月份了才收到这样的东西，Cornell 啊 Cornell，你让我现在怎么办？我想我可以说你不仁不义吧？

在这个万分窘迫的时候，我想起了曾经关心过我却又很失望的 Hofstadter。我告诉他我在 Cornell 很不开心，我很想研究程序语言，可是 Cornell 不理解也不在乎这个领域。他回信说，没有关系，你能找到自己喜欢的东西就应该去追寻它。Indiana 的 Dan Friedman 正好是做程序语言的，你可以联系他，就说是介绍你去的。

于是给 Friedman 发了 email，很快得到了回信说：“Yin，两年前我们都看过你的材料，我们觉得你是非常出众的学生，可惜你最后没有选择我们。你要明白，人生最重要的事情不是名利，而是找到你愿意合作的人。你的材料都还在我们这里。现在招生已经快结束了，但是我会把你的材料提交给招生委员会，让他们破例再次考虑你的申请。”我和 [Dan Friedman 的故事](#) 就从这里开始了。

我在 Cornell 的遭遇貌似不可告人的耻辱和秘密，然而我今天却可以把它公之于世，因为 Cornell 不再有任何资格来评价我。依靠自己的努力和 Indiana 的老师的培养，我的水平已经超越了 Cornell 计算机系的大部分教授。现在我觉得自己就像那个到 Cornell 学“游泳精髓”人，本来就是会游泳的，可是每到岸边 Cornell 就搬起大石头来砸我，还说我不会游。于是我钻到水底下钻了一个洞，把水放干。

由于曾经与多位图灵奖得主发生不大愉快的遭遇，再加上在自己的研究中多次受到其它图灵奖得主的理论的误导，而且许多位图灵奖得主最主要的贡献仍然在给软件行业带来混乱，图灵奖这个被许多计算机学生膜拜的神物，其实在我心里已经没有任何效力了。很多人可能对此难以想象，可是对图灵奖是这种态度的不只是我一个人。我认识的几乎所有程序语言专家几乎都不拿图灵奖当回事，而且其中很多人甚至不拿图灵本人当回事，觉得他设计了一些非常丑陋的东西。虽然我现在觉得图灵的研究成果确实有一定价值，但由于上面的原因，拿图灵奖来开玩笑还是成为了我的家常便饭。我甚至觉得 ACM 应该停发这个奖，因为它是一种非常虚幻和政治的东西。每当人们谈起这些“大奖”煞有介事的时候，就让我看到了他们的愚昧。

常青藤联盟和“世界一流大学”

我在 Cornell 的经历应该不是偶然，不是因为我比较特殊。跟我同时进入 Cornell 的博士生有好几个没有拿学位就离开了。其中有一个是非常聪明的少年班，18岁就读 PhD 了，我根本听不懂的理论课他还能拿A。可是四年后他退学去了 Facebook，说真是太难毕业了，神马都是扯淡。有些本科生也告诉我类似的经历，说被一个叫做“笑面虎”的教授“整了”。Cornell 的自杀率居美国大学前列。离开以后的有一天，忽然看到[新闻报道](#)说一周之内有三个 Cornell 学生从瀑布旁边的那座桥跳下去，结果派了警察在桥上日夜巡逻。我觉得自己在 Cornell 所感受到的压力确实超乎想象，是有可能把人逼上绝路的。现在回想起来真是可笑，因为下意识里在乎权威和名气，我给予了一群根本没有资格来教育我的人莫大的权力，让他们可以向我施加无端的压力。

应该指出，这种现象应该不是 Cornell 所特有的。我对清华，还有 Princeton，Harvard，MIT，Stanford，Berkeley，CMU 等学校的学生都有了解。这些所谓的“世界一流大学”或者“世界一流大学 wannabee”差不多都是类似的气氛。你冲着它们的名气和“关系网”挤破了头皮进去，然后就每天有人在你耳边对其他人感叹：哇，他好牛啊！发了好多 paper，还得了XX奖。跟参加传销大会似的，让你怀疑这些人还有没有自尊。然后就是填鸭式的教育，无止境的作业和考试，让你感觉他们不是在“教育”你，而是在“筛选”你。这种筛选总是筛掉最差的，但也筛掉最好的。因为最好的学生能意识到你在干什么，他们不给你筛选他们的机会。一旦发现其实没学到东西，中途就辍学出去创业了。所以剩下的就是最一般的，循规蹈矩听话的。在这样的环境里，你感觉不到真正的智慧和真知的存在。GRE 考试所鼓吹的什么“批判性思维”（critical thinking）在美国大学里其实是相当缺乏的。学生们只不过是在被培训成为某些其他人的工具，他们具有固定的思维定势，像是一个模子倒出来的。他们不是真正的创造者和开拓者。

人们在这些大学里的时候都是差不多感受的，可是一旦他们出来了，就会对此绝口不提。自己身上挂着这些学校的镀金牌子，怎么能砸了自己的品牌，长别人的威风？所以每当我批判 Cornell 就有些以前的同学一脸的着急相，好像自己没有吃过那苦头一样。

程序语言专家

虽然我在 Indiana 得到了思想的自由，但这种自由其实是以孤独为代价的。我不是一个自高自大不合群的人，但是我不喜欢跟一群像追星族一样的人在一起。应该说在 Indiana 的日子里，权威主义的影子也是经常出现的。Indiana 学生们的权威比较特殊一点，不然就是 Dan Friedman，不然就是 Kent Dybvig。Friedman 的身边总是围绕着一群自认为是天才的本科生，喜欢拍他的马屁，喜欢在人面前炫耀。博士生们开始时貌似还比较酷，可是后来发现其实也有很多类似现象，急于表现自己，越是研究能力弱的人越是爱表现。所以你就发现有人开头为了混进这个圈子拍你的马屁，过了两年就开始自高自大，而且经常想这样来压倒你：“Kent 说过……”我很尊敬 Dan 和 Kent，但我其实现在很多方面已经超越了他们。我看到他们的一些思维方式并不是那么的正确，我也从来不引用他们的话作为理论依据。对权威的崇拜其实显示了一个人心理的弱小。如果你对自己有信心，有自己的想法和判断力，又何必抬出个名人来压制别人呢？

在我自己心里毫无疑问的是，我是 Indiana 最厉害的程序语言（PL）学生。由于我不断地动手尝试新的想法，所以几乎没有其他人的研究逃脱过我的探索。我从来不记录自己的半成品和失败（因为太多了），而且我对自己的标准异常的高，所以我经常看到有人做演讲或者写论文，里面其实是我很久以前尝试过又抛弃了的想法。有时候我去听别人的演讲，就会立即看出破绽，问一些演讲者答不出来的问题。其实很多时候我只是怀疑自己，我试图给那些想法再一次的机会来证明它们的价值，而且问得相当委婉，但那样的问题仍然是不受欢迎的，所以同学们甚至一些助理教授

看到我在场都是心惊胆战的。吃饭的时候我也不喜欢旁边的人讨论问题，因为他们经常显示出对理论提出者的膜拜心理，而且煞有介事，可惜那些经常是我早就知道不管用的理论。他们有时候其实也知道那些是扯淡的，但却又怕我捅破这窗户纸，所以就像鸵鸟一样把头埋在沙子下面。

我也想合群一点，但是屡试不爽，所以来我就基本是孤立的做自己的研究了。最开头是不得已，但后来就越来越喜欢独自一人。这是不可避免的，因为创造力和孤独几乎是双胞胎。因为免去了跟人讨论的时间，我有了大把的时间来做自己的探索。然后我才发现当年期望的那种 common room 其实没什么用，因为那里根本不会有人理解你在说什么。现在即使有这样的地方我也不去。

我从一开始进入 Indiana 就没想过要拿博士学位，我只是在玩弄这个系统以达到我求知的目的。所以除非危及到我的存在，我把学校对学位的各种要求都抛到了九霄云外。给教授做 RA 几乎总是被要求研究各种毫无前途的东西，与我自己的思考相冲突，所以我后来干脆都做 TA 了。虽然累点，但不怎么费脑力。其结果是，在短短的一两年时间之内，我利用自己抠出来的时间，独自摸索出了这个领域大部分的理论。我经常不看书不看论文，在一个星期之内解决别人十多年才完成的研究。让人惊讶的应该不是我有多么聪明，而是这些研究者们十年来到底在干什么。我从来不认为自己比别人聪明，我只是觉得很多人的脑子被禁锢了而已。我有非常简单的头脑，我看不懂复杂的公式，听不懂高深的术语。可正是因为这一点，让我脱离了已有理论的困扰。

可以说，这个领域在过去一个多世纪的研究，很少有逃脱过我的洞察力和直觉的。这些研究最早可以追溯到 1870 年代。我一般很少看论文，因为自己想清楚一个问题其实花不了那么多时间的。看别人的论文一般都枯燥乏味，所以与其花那么多时间读论文还不如自己思考。当我看论文的时候，一般是想搞清楚自己琢磨出来的问题有没有人已经研究过了，所以很多论文只需要扫一下就够了。我看到一个东西一般很快就会知道它到底会不会管用。我经常发现一些被认为很艰深的理论其实是在解决根本不存在的问题，甚至是在制造问题，而真正的问题却没有得到有效的解决。很多问题其实是权威的阴影造成的，它让人们不敢否认这些大牛思想的价值，不敢揭穿它们，抛弃它们，甚至想让自己寄生在它们上面，所以很多的时间花在了解决一些历史遗留问题，而不是真正的问题。这就是为什么我的英文 blog 标题叫做“[Surely I Am Joking](#)”，因为它记录了一些我认为根本不存在，或者是人为造成的问题。

逻辑学家

批评 PL 领域的问题并不意味着其它领域就好一些。恰恰相反，我认为做系统和数据库的领域有更大的权威崇拜和扯淡的成分。有时候程序语言专家看起来很明显的问题，做数据库和操作系统的却看不到，扯来扯去扯不清楚，还自以为是的认为 PL 的东西他们都懂。

程序语言的理论是计算机科学的精髓所在，可是程序语言专家有他们自己的问题：他们膜拜逻辑学家。几乎每一篇 PL 领域的论文，至少有一页纸里面排列着天罡北斗阵一样的稀奇古怪的逻辑符号，而它们表示的其实不过是一些可以用程序语言轻松做出来的解释器和数据结构。有人（比如 Kent Dybvig）早就发现了这个规律，所以写了一些工具，可以把程序语言自动转换成 LaTeX 格式的逻辑公式，用以对付论文的写作。

有人觉得那些公式有“数学的美感”，可是它们其实是挺有毛病的设计。如果你看看现代逻辑学鼻祖 [Gottlob Frege](#) 的原著，就会发现其实最早的时候逻辑学不是用公式表示的。Frege 那篇开创性的论文 *Begriffsschrift* 里面全都是像电路图一样的图片，只有 20 多页，而且非常容易读懂。不知道是哪一个后辈把电路图改成了一些稀奇古怪的符号。其实他的目的是用符号来表示那些电路图，结果到后来徒孙们以为那些符号就是祖传秘籍的精髓，忘记了那些符号背后的电路图，所以导致了今天的混乱局面。看完了 Frege 的论文，我再一次领悟到了之前那句话：跟真正的大师学习，而不是跟他们的徒弟。

ACM SIGPLAN 的主席 Philip Wadler 有一次写了一篇论文介绍 [Curry-Howard correspondence](#)，里面提到，好的点子逻辑学家总是比我们先想到。可是他却没有发现，其实程序语言的能力已经大大超越了数理逻辑，数理逻辑那些公式里面的 bug 其实不少。因为逻辑学家们不用机器帮助进行推理，有些问题搞了一百多年都搞不清楚是怎么回事，然后就弄出一些特殊情况和补丁来。有了一堆逻辑“定理”，却又不能确信它们是正确的，而且存在悖论一类无厘头的东西，所以又掰出一些 model theory 之类的东西来验证它们的正确性。逻辑学家们折腾了一百多年都是在折腾类似的事情，却没怀疑过老祖宗的设计。我之前提到的 [Hindley-Milner 系统](#) 的问题，很大部分原因就在于它所使用的逻辑里面其实有一个根本性的误解。简言之，就是把全称量词 \forall 随意乱放，导致输入与输出关系混乱。这也就是我为什么不喜欢 Haskell 和 OCaml 的主要原因。

现在最热门的逻辑学家莫过于 [Per Martin-Löf](#)。他的类型理论 Martin-Löf Type Theory 被很多 PL 人奉为神圣。我一直没有搞清楚这个类型理论有什么特别，直到有一天我把 Martin-Löf 1980 年的那篇论文（其实是演讲稿）拿出来看了一遍。然后我发现他通篇本质上就是在讲一个 partial evaluator 要怎么写，而我早就自己写过 partial evaluator。其实并不是特别神奇的东西，只需要在普通解释器里面改一两行代码就行，可是有人（比如 Neil Jones）却为此写出了 [400 多页的书](#) 和大量的论文。

之前提到的 Curry-Howard correspondence 也被很多人奉为神圣，它来自数学家 Haskell Curry 和逻辑学家 W.A. Howard 的一些早期发现。他们发现有些程序和定理的证明之间有对应的关系。然后就有 PL 专家开始走火入魔，说“程序就是证明，程序的类型就是定理”。可是他们却没有发现这个说法没法解释操作系统这种程序，因为它被设计为永远不停地运行，所以不能满足一个证明所具有的基本特征。而且很多程序被设计出来根本就不是要证明什么定理，它们是被设计来帮人做事情的。所以我觉得“程序就是证明”是很牵强附会的说法，你不能因为有的程序可以用来证明数学定理，就认为所有的程序都是某个定理的证明啊！把那句话反过来，说成“证明就是程序”还差不多。

但从以上的发现，我很高兴的看到了自己作为一个程序员的价值。很多人瞧不起程序员，把他们蔑称为“码农”，可是程序如果写好了，其实比起那些高深的逻辑学家和哲学家还要强，因为程序语言其实比数理逻辑还要强。有一位 [数学](#)

[家](#)说得好：为了真正深入的理解一个东西，你应该把它写成程序。还有人说，编程只是一门失传的艺术的别名，这门艺术叫做“思考”。我觉得很在理。

再见了，权威们

几经颠簸的求学生涯，让我获得了异常强大的力量。我的力量不仅来自于老师们的教诲，而且在于我自己不懈的追求，因为机会只青睐有准备的头脑。

曾经 Knuth 是我心中唯一的权威，后来我又屈服于 Cornell 和常青藤联盟的权威和名气。在一而再再而三的上当受骗之后，我终于把所有的权威们从我的脑子里轰了下去。也许有时候轰得太猛烈了一些，但总的说来是有好处的。不再是我心目中的权威并不等于我鄙视他们或者不尊敬他们。我只是获得了不用膜拜他们，不用跟一群人瞎起哄的自由。我不尊敬的人都是一些自视过高的或者他们的跟屁虫。一般来说，权威们在我的脑子里失去的只是他们在很多其他人脑子里的那种被膜拜的地位，那种你可以用“XX人说过……”来压倒理性分析的地位。现在他们在我心目中是一群普通的，由蛋白质形成的生物，有好心肠或者坏心眼的，高傲，谦虚或者虚伪的人。我不会自讨苦吃，他们的想法如果真的好，我当然要拿来用，但是没有任何人的东西我是不加批判全盘接受的。我深深地知道接受错误想法的危害性，所以我也希望大家都具有批判的思维，不要盲目的接受我说的话。我不喜欢“大神”或者“牛人”这种称呼，我也反感那种自称膜拜我的人，因为正是这种人让权威主义现在横行于世。

美国的权威主义胜于欧洲，但也不是每个人都那么的崇拜权威，而中国才是权威主义的重灾区。像“图灵奖得主XX”这样的称呼，恐怕只有在中国才见得到。所以我希望国内的同学们，不要盲目的崇拜国外的所谓“大师”，“牛校”或者“牛公司”。祝你们早日消灭掉心里的各种权威以及对他们的畏惧心理，认识到自己的价值和力量。

后记 (关于 IU)

有些人看了我的文章介绍在 IU 的经历，告诉我他们申请了 IU。我觉得有必要免责声明一下：我没想到，也不希望有人因为我的文章而去 IU，[YMMV](#) (your mileage may vary)。由于我有所准备，所以对于 Friedman 的教学如鱼得水。很多同学也说学到很多，可是有一些其他人告诉我他们觉得 Friedman 的课他们听起来很吃力，只能说是勉强过关。而且我只介绍了 IU 好的方面，却把不大好的地方一笔带过了。我在 IU 也有很艰难的时候。现在的情况是 Kent Dybvig 已经离开了 IU，加入了 Cisco。他的公司 Cadence Research Systems 和 Chez Scheme 也并入了 Cisco。Dan Friedman 由于年纪原因说不准还带不带学生。最近引进了一些貌似不错的助理教授，但是我跟他们都不熟。我的经验是助理教授一般都会为了研究资金，为了升为正教授而做一些身不由己的事情。其他的 CS 方向我都说不准 IU 是什么水平，所以还请同学们自己斟酌。我可以毫无疑问的一点是，IU 有非常美丽的校园，大大的超过清华，北大，Cornell，Stanford，MIT。

PySonar2 与 Sourcegraph 集成完毕

来到 Sourcegraph 两个星期了，我可以说这里的每一天都是激动人心的，这是一个有真正创造活力的 startup。我们的发展速度相当之快，每一天都出现新的点子，或者发现以前做法的一些大幅度简化。不得不承认 Quinn 和 Beyang 是比我有魄力的人。我虽然做出了 PySonar，却让它的代码束之高阁多年之久，没有发挥出应有的作用。而 Quinn 和 Beyang 看准了这种东西的价值，坚持不懈地做出了 [Sourcegraph.com](#) 这个网站，才使 PySonar 可以发挥出这么强劲的效果，用以搜索全世界的 Python 代码，为广大程序员造福。当然，我们的目标不只限于 Python。Sourcegraph 目前支持 Go, JavaScript, Python 和 Ruby。其中 Ruby 的支持还处于初步阶段，需要改善，更多其它的语言正在开发中。

经过两个星期的勤奋却又不知疲倦的工作，PySonar2 的代码得到了巨大的改善。现在我可以很自信的说，它包含了世界上最强大的静态分析技术。今天 PySonar2 已经正式与 [Sourcegraph.com](#) 集成完毕。现在只要登录 Sourcegraph 主网站就可以看到开源 Python 代码的 PySonar2 分析结果。

PySonar2 的类型推导系统能够不依赖类型标记却精确地分析出 Python 函数的参数类型。比如下图所示的 Flask 框架的最常用的五个函数的参数，都是用通常的方法很难确定类型的，PySonar2 却能得知它们的正确用法。

The screenshot shows the Sourcegraph code browser interface. At the top, there are three tabs: 'Top Functions', 'Code Browser' (which is selected), and 'README'. Below the tabs, there is a search bar and a table of function definitions. The table has five rows, each representing a different function from the Flask framework:

	method	Flask.route(self, rule, **options)	(Flask, str) -> () -> str -> () -> str	A decorator that is used to register a view function for a given URL rule.
func	templating.render_template(template_name_or_list, **context)	str -> ? [str] -> ?	Renders a template from the template folder with the given context.	
func	helpers.url_for(endpoint, **values)	str -> ?	Generates a URL to the given endpoint with the method provided.	
func	helpers.flash(message, category)	(str, str) -> None (Markup, str) -> None	Flashes a message to the next request.	
method	Blueprint.route(self, rule, **options)	(Blueprint, str) -> ? -> ?	Like :meth:`Flask.route` but for a blueprint.	

最有意思的是那个 `render_template`。PySonar2 为它推导出来的类型是一个 intersection type :

```
templating.render_template(template_name_or_list, **context)
str -> ?
| [str] -> ?
```

这是说，第一个参数 `template_name_or_list` 的类型或者是 `str` 或者是 `[str]`（含有 `str` 的 `list`）。如果你给它 `str` 它就会输出 `?` (PySonar2 不知道它会输出什么)，而如果你给它 `[str]`，它输出 `?`。

如果你注意一下这个参数的英文含义 “`template name or list`”，就觉得仿佛 PySonar2 能读懂英文一样。然而 PySonar2 其实不会英文，它只会 Python，它通过代码之间的调用关系和异常强大的类型推导，找到了这个参数的类型。

Sourcegraph 的一些使用诀窍

Sourcegraph 有一些不为人知的巧妙设计，但是由于 Quinn 和 Beyang 太谦虚而且太忙了，所以都没来的及宣传。我现在偷闲在这里透露两招小窍门。

启动分析你需要的 GitHub 代码库

目前这个功能只限于 GitHub。如果在 Sourcegraph 网站上面没有找到你需要的 GitHub 代码库，这不等于你需要等我们来启动分析。你可以自己动手！

方法很简单：把你的 GitHub 地址去掉 `http://` 之后放到 `http://sourcegraph.com/` 后面，然后 Sourcegraph 就会显示一个等待页面，同时自动开始分析这个 repo。



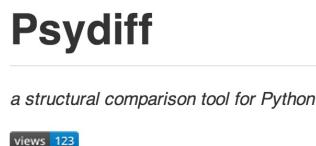
举个例子，如果你想分析 `http://github.com/myname/myrepo` 的代码，就在浏览器输入地址：

`http://sourcegraph.com/github.com/myname/myrepo`

如果 Sourcegraph 还没有分析过这个 repo 它就会把它加入到工作队列里，然后你可以做其他的事情或者浏览其他的代码。分析完毕之后浏览器就会自动跳转到你所需要的代码库。一般大小的代码库几分钟到半个小时就会处理完毕。

在你的 GitHub README 里面使用 Sourcegraph 徽章

你也许发现有些人在自己的 GitHub 里有 Sourcegraph 徽章，这样一来别人就能得知你的代码库的一些统计信息。比如我的 psydiff 的 README 里面有这样一个：



它表示 psydiff 的代码被看过的次数。你也可以使用其他的一些徽章，比如最常用的函数，交叉引用数，用户数，等等：



要得到这些徽章很简单，只要在你的 repo 的 Sourcegraph 主页里点击如图所示的扳手状小图标，然后把“Image URL”拷贝到你的网页里就行：



Badges

Add these badges to the README file:

Badge	Image URL	Markdown
xrefs 0	https://sourcegraph.com/api/repos/git	<code>![xrefs](https://sourcegraph.com/api/r...</code>
funcs 45	https://sourcegraph.com/api/repos/git	<code>![funcs](https://sourcegraph.com/api/r...</code>
most used func <code>_init_</code>	https://sourcegraph.com/api/repos/git	<code>![top func](https://sourcegraph.com/a...</code>
library users 0	https://sourcegraph.com/api/repos/git	<code>![library users](https://sourcegraph.co...</code>
graph ok	https://sourcegraph.com/api/repos/git	<code>![status](https://sourcegraph.com/api/...</code>

Sourcegraph 的功能虽然非常强劲，但是很多设计的工作还处于起步阶段。如果你有什么建议或者发现问题，请联系我们：hi@sourcegraph.com.

丘奇和图灵

丘奇 (Alonzo Church) 和图灵 (Alan Turing) 是两位对计算机科学具有最大影响力的人物，然而他们却具有非常对立的观点和相差很多的名气。在我长达16年的计算机科学生涯中，总是感觉到自己的思想反反复复的徘徊于这两个“阵营”之间。丘奇代表了“逻辑”和“语言”，而图灵代表着“物理”和“机器”。在前面的8年中，我对丘奇一无所知，而在后面的8年中，我却很少再听到图灵的名字。他们的观点谁对谁错，是一个无法回答的问题。完全投靠丘奇，或者完全投靠图灵，貌似都是错误的做法。这是一种非常难说清楚的，矛盾的感觉，但是今天我试图把自己的感悟简要的介绍一下。

丘奇与图灵之争

想必世界上所有的计算机学生都知道图灵的大名和事迹，因为美国计算机器学会 (ACM) 每年都会颁发“图灵奖”，它被誉为计算机科学的最高荣誉。大部分的计算机学生都会在某门课程（比如“计算理论”）学习“图灵机”的原理。然而，有多少人知道丘奇是什么人，他做出了什么贡献，他与图灵是什么样的关系呢？我想恐怕不到一半的人吧。

如果你查一下[数学家谱图](#)，就会发现丘奇其实是图灵的博士导师。然而从 Andrew Hodges 所著的《[图灵传](#)》，你却可以看到图灵的心目中仿佛并没有这个导师，仿佛自己的“全新发明”应得的名气，被丘奇抢走了一样（注意作者的用词：robbed）。事实到底是怎样的，恐怕谁也说不清楚。我只能说，貌似计算机科学从诞生之日起就开始充满了各种“宗教斗争”。

虽然现在图灵更加有名，然而在现实的程序设计中，却是丘奇的理论在起着绝大部分的作用。据我的经验，丘奇的理论让很多事情变得简单，而图灵的机器却过度的复杂。丘奇所发明的 lambda calculus 以及后续的工作，是几乎一切程序语言的理论基础。而根据老一辈的计算机工程师们的描述，最早的计算机构架也没有受到图灵的启发，那是一些电机工程师完全独立的工作。然而有趣的是，继承了丘奇衣钵的计算机科学家们拿到的那个大奖，仍然被叫做“图灵奖”。我粗略的算了一下，在迄今所有的图灵奖之中，程序语言的研究者占了近三分之一。

从图灵机到 lambda calculus

图灵机永远的停留在了理论的领域，绝大多数被用在“计算理论” (Theory of Computation) 中。计算理论其实包括两个主要概念：“可计算性理论” (computability) 和“复杂度理论”(complexity)。这两个概念在通常的计算理论书籍（比如 Sipser 的经典教材）里，都是用图灵机来叙述的。在学习计算理论的时候，绝大多数的计算机学生恐怕都会为图灵机头痛好一阵子。

然而在做了研究生“计算理论”课程一个学期的 TA 之后我发现，其实几乎所有计算理论的原理，都可以用 lambda calculus，或者程序语言和解释器的原理来描述。所谓“通用图灵机” (Universal Turing Machine)，其实就是一个可以解释自己的解释器，叫做“元解释器” (meta-circular interpreter)。在 IU 的程序语言理论课程中，我最后的项目就是一个 meta-circular interpreter。这个解释器能够完全的解释它自己，而且可以任意的嵌套（也就是说用它自己来解释它自己，再来解释它自己……）。然而我的“元解释器”却是基于 lambda calculus 的，所以我后来发现了一种方法，可以完全的用 lambda calculus 来解释计算理论里面几乎所有的定理。

我为这个发现写了两篇博文：[《A Reformulation of Reducibility》](#) 和 [《Undecidability Proof of Halting Problem without Diagonalization》](#)。我把 Sipser 的[计算理论课本](#)里面的几乎整个一章的证明都用我自己的这种方式改写了一遍，然后讲给上课的学生。因为这种表示方法比起通常的“图灵机+自然语言”的方式简单和精确，所以收到了相当好的效果，好些学生对我说有一种恍然大悟的感觉。

我把这一发现告诉了我的导师 Amr Sabry。他笑了，说这个他早就知道了。他推荐我去看一本书，叫做[《Computability and Complexity from a Programming Perspective》](#)，作者是大名鼎鼎的 Neil Jones (他也是[Partial Evaluation](#)这一重要概念的提出者)。这本书不是用图灵机，而是一种近似于 Pascal，却又带有 lambda calculus 的一些特征的语言（叫做“WHILE 语言”）来描述计算理论。用这种语言，Jones 不但轻松的证明了所有经典的计算理论定理，而且能够证明一些使用图灵机不能证明的定理。

我曾经一直不明白，为什么可以如此简单的解释清楚的事情，计算理论需要使用图灵机，而且叙述也非常的繁复和含糊。由于这些证明都出于资深的计算理论家们之手，让我不得不怀疑自己的想法里面是不是缺了点什么。可是在看到了 Jones 教授的这本书之后，我倍感欣慰。原来一切本来就是这么的简单！

后来从 CMU 的教授 Robert Harper 的一篇博文 [《Languages and Machines》](#) 中，我也发现 Harper 跟我具有类似的观点，甚至更加极端一些。他强烈的支持使用 lambda calculus，反对图灵机和其他一切机器作为计算理论的基础。

从 lambda calculus 到电子线路

当我在 2012 年的 POPL 第一次见到 Neil Jones 的时候，他跟我解释了许多的问题。当我问到他这本书的时候，他对我说：“我不推荐我的书给你，因为大部分的人都觉得 lambda calculus 难以理解。”Lambda calculus 难以理解？我怎么不觉得呢？我觉得图灵机麻烦多了。然后我才发现，由于经过了这么多年的研究之后，自己对 lambda calculus 的理解程度已经到了深入骨髓的地步，所以我已经全然不知新手对它是什么样的感觉。原来“简单”这个词，

在具有不同经历的人头脑里，有着完全不同的含义。

所以其实 Jones 教授说的没错，lambda calculus 也许对于大部分人来说不合适，因为对于它没有一个好的入门指南。Lambda calculus 出自逻辑学家之手，而逻辑学家们最在行的，就是把很简单的“程序”用天书一样的公式表示出来。这难怪老一辈的逻辑学家们，因为他们创造那些概念的时候，计算机还不存在。但是如果现在还用那一堆符号，恐怕就有点落伍了。大部分人在看到 beta-reduction, alpha-conversion, eta-conversion, ... 这大堆的公式的时候，就已经头痛难忍了，怎么还有可能利用它来理解计算理论呢？

其实那一堆符号所表示的东西，终究超越不了现实里的物体和变化，最多不过再幻想一下“多种未来”或者“时间机器”。有了计算机之后，这些符号公式，其实都可以用数据结构和程序语言来表示。所以 lambda calculus 在我的头脑里真的很简单。每一个 lambda 其实就像是一个电路模块。它从电线端子得到输入，然后输出一个结果。你把那些电线叫什么名字根本不重要，重要的是同一根电线的名字必须“一致”，这就是所谓的“alpha-conversion”的原理……不在这里多说了，如果你想深入的了解我心目中的 lambda calculus，也许可以看看我的另一篇博文《[怎样写一个解释器](#)》，看看这个关于类型推导的[幻灯片](#)的开头，或者进一步，看看如何推导出 [Y combinator](#)，或者看看《[What is a program?](#)》。你也可以看看 Matthias Felleisen 和 Matthew Flatt 的《[Programming Languages and Lambda Calculi](#)》。

所以，也许你看到了在我的头脑里面并存着丘奇和图灵的影子。我觉得丘奇的 lambda calculus 是比图灵机简单而强大的描述工具，然而我却又感染到了图灵对于“物理”和“机器”的执着。我觉得逻辑学家们对 lambda calculus 的解释过于复杂，而通过把它理解为物理的“电路元件”，让我对 lambda calculus 做出了更加简单的解释，把它与“现实世界”联系在了一起。



所以到最后，丘奇和图灵这两种看似矛盾的思想，在我的脑海里得到了和谐的统一。这些精髓的思想帮助我解决了许多的问题。感谢你们，计算机科学的两位鼻祖。

原因与证明

证明

我在 Cornell 的时候经常遇到这样的问题，那就是教授们一上课就在黑板上写长篇的“定理证明”，全体同学认认真真在下面抄笔记，就连只有十来个人的小课也是那样。有些写字速度慢的人就不得不带上小型录音机，把教授的课全都录下来，要不就是之后去借别人的笔记来抄。

有一次某知名教授照着讲义，背对着学生，在黑板上写了大半节课，写下好几板的证明，证明的是 simply typed lambda calculus (STLC) 的 strong normalization 特性 (SN)。刚写完就到下课时间了，他回过头来喘了一口气，说：“Any questions？”没有人应声，于是他说：“很好！下课！”

几天后我问他，你证明了 STLC 有这个特性，然而你却没有告诉我它“为什么”有这个特性。他神气的看了我一眼：“你不懂吗？”我说：“你的证明我看懂了大部分，可是一个东西具有如此的性质，并不是因为你证明了它。这性质是它天生就有的，不管你是否能证明它。我想知道的是什么让 STLC 具有这个性质，而不只是证明它。”他说：“你问这样的问题有什么意义吗？你需要非常聪明，并且需要经过大量的努力才能想出这样的证明。”

原因

两年之后，我在 Indiana 上了另外一堂程序语言理论课。教授是我之前的导师 Amr Sabry。他上课从来不带讲义，貌似也没有准备，漫不经心的，却每次都能讲清楚问题的关键。于是有一天他也开始讲 STLC 的 SN 特性。他说，我不想写下这个证明让你们抄，我只告诉你们大概怎么去想。SN 的意思就是程序肯定会“终止”。所有会终止的程序，都会有一个“特征值”会随着程序的运行而减小。你需要做的就是找到 STLC 的“特征值”是什么。接着他就开始在黑板上画一个图……

过了一段时间，我不仅学会了这个“证明”，而且知道了 STLC 具有如此特性的“原因”。

证明与原因的区别

从以上的故事，以及你的亲身经历中，你也许注意到了大部分的教育过分的重视了“证明”，却忽略了比证明更重要的东西——“原因”。

原因往往比证明来得更加简单，更加深刻，但却更难发现。对于一个事实往往有多种多样的证明，然而导致这个事实的原因却往往只有一个。如果你只知道证明却不知道原因，那你往往就被囚禁于别人制造的理论里面，无法自拔。你能证明一个事物具有某种特性，然而你却没有能力改变它。你无法对它加入新的，好的特性，也无法去掉一个不好的特性。你也无法发明新的理论。有能力发明新的事物和理论的人，他们往往不仅知道“证明”，而且知道“原因”。

打个比方。证明与原因的区别，就像是犯罪的证据与它的原因的区别。证据并不是导致犯罪的原因。有了证据可以帮助你把罪犯绳之以法，可是如果你找不到他犯罪的原因，你就没法防止同样的犯罪现象再次发生。

古人的“知其然”与“知其所以然”的区别，也就是同样的道理吧。

编辑器与IDE

无谓的编辑器战争

很多人都喜欢争论哪个编辑器是最好的。其中最大的争论莫过于 Emacs 与 vi 之争。vi 的支持者喜欢说：“看 vi 打起来多快，手指完全不离键盘，连方向键都可以不用。”Emacs 的支持者往往对此不屑一顾，说：“打字再快又有什么用。我在 Emacs 里面按一个键，等于你在 vi 里面按几十个键。”

其实还有另外一帮人，这些人喜欢说：“对于 Emacs 与 vi 之争，我的答案是 {jEdit, Geany, TextMate, Sublime...}”这些人厌倦了 Emacs 的无休止的配置和 bug，也厌倦了 vi 的盲目求快和麻烦的模式切换，所以他们选择了另外的更加简单的解决方案。

临时解决方案 - IDE

那么我对此的答案是什么呢？在目前的情况下，我对程序编辑的临时答案是：IDE。

写程序的时候，我通常根据语言来选择最能“理解”那种语言的“IDE”（比如 Visual Studio, Eclipse, IntelliJ IDEA 等），而不是一种通用的“文本编辑器”（比如 Emacs, vi, jEdit, ...）。这是因为“文本编辑器”这种东西一般都不真正的理解程序语言。很多 Emacs 和 vi 的用户以为用 etags 和 ctags 这样的工具就能让他们“跳转到定义”，然而这些 tags 工具其实只是对程序的“文本”做一些愚蠢的正则表达式匹配。它们根本没有对程序进行 parse，所以其实只是在进行一些“瞎猜”。简单的函数定义它们也许能猜对位置，但是对于有重名的定义，或者局部变量的时候，它们就力不从心了。

很多人对 IDE 有偏见，因为他们认为这些工具让编程变得“傻瓜化”了，他们觉得写程序就是应该“困难”，所以他们眼看着免费的 IDE 也不试一下。有些人写 Java 都用 Emacs 或者 vi，而不是 Eclipse 或者 IntelliJ。可是这些人错了。他们没有意识到 IDE 里面其实蕴含了比普通文本编辑器高级很多的技术。这些 IDE 会对程序文本进行真正的 parse，之后才开始分析里面的结构。它们的“跳转到定义”一般都是很精确的跳转，而不是像文本编辑器那样瞎猜。

这种针对程序语言的操作可以大大提高人们的思维效率，它让程序员的头脑从琐碎的细节里面解脱出来，所以他们能够更加专注于程序本身的语义和算法，这样他们能写出更加优美和可靠的程序。这就是我用 Eclipse 写 Java 程序的时候相对于 Emacs 的感觉。我感觉到自己的“心灵之眼”能够“看见”程序背后所表现的“模型”，而不只是看到程序的文本和细节。所以，我经常发现自己的头脑里面能够同时看到整个程序，而不只是它的一部分。我的代码比很多人的都要短很多也很有很大一部分是这个原因，因为我使用的工具可以让我在相同的时间之内，对代码进行比别人多很多次的结构转换，所以我往往能够把程序变成其他人想象不到的样子。

对于 Lisp 和 Scheme，Emacs 可以算是一个 IDE。Emacs 对于 elisp 当然是最友好的了，它的 Slime 模式用来编辑 Common Lisp 也相当不错。然而对于任何其它语言，Emacs 基本上都是门外汉。我大部分时间在 Emacs 里面是在写一些超级短小的 Scheme 代码，我有自己的一个简单的[配置方案](#)。虽然谈不上是 IDE，Emacs 编辑 Scheme 确实比其它编辑器方便。R. Kent Dybvig 写 Chez Scheme 居然用的是 vi，但是我并不觉得他的编程效率比我高。我的代码很多时候比他的还要干净利落，一部分原因就是因为我使用的 ParEdit mode 能让我非常高效的转换代码的“形状”。

当要写 Java 的时候，我一般都用 Eclipse。最近写 C++ 比较多，C++ 的最好的 IDE 当然是 Visual Studio。可惜的是 VS 没有 Linux 的版本，所以就拿 Eclipse 凑合用着，感觉还比较顺手。个别情况 Eclipse “跳转定义”到一些完全不相关的地方，对于 C++ 的 refactor 实现也很差，除了最简单的一些情况（比如局部变量重命名），其它时候几乎完全不可用。当然 Eclipse 遇到的这些困难，其实都来自于 C++ 语言本身的糟糕设计。

终极解决方案 - 结构化编辑器

想要设计一个 IDE，可以支持所有的程序语言，这貌似一个不大可能的事情，但是其实没有那么难。有一种叫做“结构化编辑器”的东西，我觉得它可能就是未来编程的终极解决方案。

跟普通的 IDE 不同，这种编辑器可以让你直接编辑程序的 AST 结构，而不是停留于文本。每一个界面上的“操作”，对应的是一个对 AST 结构的转换，而不是对文本字符的“编辑”。这种 AST 的变化，随之引起屏幕上显示的变化，就像是变化后的 AST 被“pretty print”出来一样。这些编辑器能够直接把程序语言保存为结构化的数据（比如 S 表达式，XML 或者 JSON），到时候直接通过对 S 表达式，XML 或者 JSON 的简单的“解码”，而不需要针对不同的程序语言进行不同的 parse。这样的编辑器，可以很容易的扩展到任何语言，并且提供很多人都想象不到的强大功能。这对于编程工具来说将是一个革命性的变化。

- 已经有人设计了这样一种编辑器的模型，并且设计的相当不错。你可以参考一下这个[结构化编辑器](#)，它包含一些 Visual Studio 和 Eclipse 都没有的强大功能，却比它们两者都要更加容易实现。你可以在这个网页上下载这个编辑器模型来试用一下。
- 我之前推荐过的 [TeXmacs](#) 其实在本质上就是一个“超豪华”的结构化编辑器。你可能不知道，TeXmacs 不但能排版出 TeX 的效果，而且能够运行 Scheme 代码。

- IntelliJ IDEA 的制造者 JetBrains 做了一个结构化编辑系统，叫做 [MPS](#)。它是开源软件，并且可以免费下载。
- 另外，Microsoft Word 的创造者 Charles Simonyi 开了一家叫做 [Intentional Software](#) 的公司，也做类似的软件。

程序语言的常见设计错误(2) - 试图容纳世界

之前的一篇文章里，我谈到了程序语言设计的一个常见错误倾向：[片面追求短小](#)，它导致了一系列的历史性的设计错误。今天我来谈一下另外一种错误的倾向，这种倾向也导致了很多错误，并且继续在导致错误的产生。

今天我要说的错误倾向叫做“试图容纳世界”。这个错误导致了 Python, Ruby 和 JavaScript 等“动态语言”里面的一系列问题。我给 Python 写过一个静态分析器，所以我基本上实现了整个 Python 的语义，可以说是对 Python 了解的相当清楚了。在设计这个静态分析的时候，我发现 Python 的设计让静态分析异常的困难，Python 的程序出了问题很难找到错误的所在，Python 程序的执行速度比大部分程序语言都要慢，这其实是源自 Python 本身的设计问题。这些设计问题，其实大部分出自同一个设计倾向，也就是“试图容纳世界”。

在 Python 里面，每个“对象”都有一个“字典”（dictionary）。这个 dict 里面含有这个对象的 field 到它们的值之间的映射关系，其实就是一个哈希表。一般的语言都要求你事先定义这些名字，并且指定它们的类型。而 Python 不是这样，在 Python 里面你可以定义一个人，这个人的 field 包括“名字”，“头”，“手”，“脚”，……

但是 Python 觉得，程序应该可以随时创建或者删除这些 field。所以，你可以给一个特定的人增加一个 field，比如叫做“第三只手”。你也可以删除它的某个 field，比如“头”。Python 认为这更加符合这个世界的工作原理，有些人就是可以没有头，有些人又多长了一只手。

好吧，这真是太方便了。然后你就遇到这样的问题，你要给这世界上的每个人戴一顶帽子。当你写这段代码的时候，你意识中每个人都有头，所以你写了一个函数叫做 putOnHat，它的输入参数是任意一个人，然后它会给他（她）的头上戴上帽子。然后你想把这个函数 map 到一个国家的所有人的集合。

然而你没有想到的是，由于 Python 提供的这种“描述世界的能力”，其它写代码的人制造出各种你想都没想到的怪人。比如，无头人，或者有三只手，六只眼的人，……。然后你就发现，无论你的 putOnHat 怎么写，总是会出意外。你惊讶的发现居然有人没有头！最悲惨的事情是，当你费了几个月时间和相当多的能源，给好几亿人戴上了帽子之后，才忽然遇到一个无头人，所以程序当掉了。然而即使你知道程序有 bug，你却很难找出这些无头人是从哪里来的，因为他们来到这个国家的道路相当曲折，绕了好多道弯。为了重现这个 bug，你得等好几个月，它还不一定会出现…… 这就是所谓 [Higgs-Bugson](#) 吧。

怎么办呢？所以你想出了一个办法，把“正常人”单独放在一个列表里，其它的怪人另外处理。于是你就希望有一个办法，让别人无法把那些怪人放进这个列表里。你想要的其实就是 Java 里的“类型”，像这样：

```
List<有一个头和两只手的正常人> normalPeople;
```

很可惜，Python 不提供给你这种机制，因为这种机制按照 Python 的“哲学”，不足以容纳这个世界的博大精深的万千变化。让程序员手工给参数和变量写上类型，被认为是“过多的劳动”。

这个问题也存在于 JavaScript 和 Ruby。

语言的设计者们都应该明白，程序语言不是用来“构造世界”的，而只是对它进行简单的模拟。试图容纳世界的倾向，没带来很多好处，没有节省程序员很多精力，却使得代码完全没有规则可言。这就像生活在一个没有规则，没有制度，没有法律的世界，经常发生无法预料的事情，到处跑着没有头，三只手，六只眼的怪人。这是无穷无尽的烦恼和时间精力的浪费。

关于语言的思考

之前写了那么多 Haskell 的不好的地方，却没有提到它好的地方，其实我必须承认我从 Haskell 身上学到了非常重要的东西，那就是对于“类型”的思考。虽然 Haskell 的类型系统有过于强烈的约束性，从一种“哲学”的角度（不是数学的角度）来看非常“不自然”，但如果一个程序员从来没学过 Haskell，那么他的脑子里就会缺少一种重要的东西。这种东西很难从除 Haskell，ML，Clean，Coq，Agda 以外的其它语言身上学到。

Haskell 给我的启发

一个没有学过 Haskell 的 Scheme 程序员最容易犯的一个错误就是，把除 #f (Scheme 的逻辑“假”）以外的任何值都作为 #t (Scheme 的逻辑“真”）。很多人认为这是 Scheme 的一个“特性”，可是殊不知这其实是 Scheme 的极少数缺点之一。如果你了解 Lisp 的历史，就会发现在最早的时候，Lisp 把 nil (空链表) 这个值作为“假”来使用，而把 nil 以外的其它值都当成“真”。这带来了逻辑思维的混乱。

Scheme 对 Lisp 的这种混乱做法采取了一定的改进，所以在 Scheme 里面，空链表 '() 和逻辑“假”值 #f 被划分开来。这是很显然的事情，一个是链表，一个是 bool，怎么能混为一谈。Lisp 的这个错误影响到了很多其它的语言，比如 C 语言。C 语言把 0 作为“假”，而把不是 0 的值全都作为“真”。所以你就看到有些自作聪明的 C 程序员写出这样的代码：

```
int i = 0;  
...  
...  
if (i++) { ... }
```

Scheme 停止把 nil 作为“假”，却仍然把不是 #f 的值全都作为“真”。Scheme 的崇拜者一般都告诉你，这样做的好处是，你可以使用

```
(or x y z)
```

这样的表达式，如果其中有一个不是 #f，那么这个表达式会直接返回它实际的值，而不只是 #t。然后你就可以写这样的代码：

```
(cond  
[(or x y z)  
 => (lambda (found)  
 (do-something-with found))])
```

而不是：

```
(let ([found (first-non-false x y z)])  
(cond  
[(not (eq? found #f))  
 (do-something-with found)]))
```

第一段代码使用了 Scheme 的一个特殊“语法”，=> 后面的 (lambda (found) ...) 会把 (or x y z) 返回的值作为它的参数 found，然后返回函数计算出的结果。第二段代码没有假设任何不是 #f 的值都是“真”，所以它不把 (or x y z) 放进 cond 的条件里，而是首先把它返回的值绑定到 found，然后再把这个值放进 cond 的条件。

这第二段代码比第一段代码多了一个 let，增加了一层缩进，貌似更加复杂了，所以很多人觉得把不是 #f 的值全都作为“真”这一做法是合理的。其实 Scheme 为了达到这个目的，恰好犯了“片面追求短小”的语言设计的小聪明（参考这篇[博文](#)）。为了让这种情况变得短小而损失类型的准确，这种代价是非常不值得的。

Haskell 的类型系统就是帮助你严密的思考类似关于类型的问题的。如果你从来没学过 Haskell，你就不会发现这里其实有个类型错误。可是 Haskell 做得过分了一点，由于对类型推导，一阶逻辑和 category theory 等理论的盲目崇拜，Haskell 里面引入了很多不必要的复杂性。

各种各样的类型推导我设计过不下十个，其中有一些比 Haskell 强大很多。category theory 其实也不是什么特别有用的东西。很多数学家把它叫做“abstract nonsense”，就是说它太“通用”了，以至于相当于什么都没说。我曾经在一个晚上看完了整本的 category theory 教材，发现里面的内容我其实通过自己的动手操作（实现编译器，设计类型系统和静态分析等等），早就明白了。这里面的理论并不能带来对程序语言的简化。恰恰相反，它让程序语言变得复杂。

我对 Haskell 程序员的“天才态度”也感到厌倦，所以我不想再使用 Haskell，然而我的脑子里却留下了它“启发”我的东西。对 Haskell 的理解，让我成为了一个更好的 Scheme 程序员，更好的 Java 程序员，更好的 C++ 程序员，甚至更好的 shell 脚本程序员。我能够在任何语言里再现 Haskell 的编程方式的精髓。然而让我继续用 Haskell，却就像是让我坐牢一样。本来很简单的事情，到 Haskell 里面就变成一些莫名其妙的新术语。Haskell 的设计者们的论文我大部分都看过，几分钟之内我就知道他们那一套东西怎么变出来的，其实里面很少有新的东西。大部分是因为

Haskell 引入的那些“新概念”（比如 monad）而产生的无须有的问题。世界上有比他们更聪明的人，更简单却更强大的理论。所以不要以为 Haskell 就是世界之巅。

怎么说呢，我觉得每个程序员的生命中都至少应该有几个月在静心学习 Haskell。学会 Haskell 就像吃几天素食一样。每天吃素食显然会缺乏全面的营养，但是每天都吃荤的话，你恐怕就永远意识不到身体里的毒素有多严重。

专攻一门语言的害处

我曾经对人说 C++ 里面其实有一些好东西，但是我没有说的是，C++ 里面的坏东西实在太多了。C++ 是一门“毒素”很多的语言，就像猪肉一样。

有些人从小写 C++，一辈子都在写 C++，就像每天每顿吃猪肉一样。结果是他们对 C++ 里面的“[珍珠](#)”掌握的非常牢靠，以至于出现了一种“脑残”的现象——他们没法再写出逻辑清晰的程序。（这里“珍珠”是一个特殊的术语，它并不含有赞美的意思。请参考这篇[博文](#)。）

比如，很多 C++ 程序员很精通 functor 的写法，可是其实 functor 只是由于 C++ 没有 first-class function 而造成的“变通”。C++ 的 functor 永远也不可能像 Scheme 的 lambda 函数一样好用。因为每次需要一个 functor 你都得定义一个新的 class，然后制造这个 class 的对象。如果函数里面有自由变量，那么这些自由变量必须通过构造函数放进 functor 的 field 里面，这样当 functor 内部的“主方法”被调用的时候，它才能知道自由变量的值。所以为此，你又得定义一些 field。麻烦了这么久，你得到的其实不过是 Scheme 程序员用起来就像呼吸空气一样的 lambda。

很多精通 functor 的 C++ 程序员认为会用 functor 就说明自己水平高。殊不知 functor 这东西不但是一个“变通”，而且是从函数式语言里面“学”过来的。在最早的时候，C++ 程序员其实是不知道 functor 这东西的。如果你考一下古就会发现，C++ 诞生于 1983 年，而 Scheme 诞生于 1975 年，Lisp 诞生于 1958 年。C++ 的诞生比 Scheme 整整晚了 8 年，然而 Scheme 一开始就有 lexical scoping 的 lambda。functor 只不过是 lambda 的一种绕着弯的模仿。实际上 C++ 后来加进去的一些东西（包括 boost 库），基本上都是东施效颦。

记得 2011 年 11 月 11 日的良辰吉日，C++ 的创造者 Bjarne Stroustrup 在 Indiana 大学做了一个演讲，主题是关于 C++11 的新特性。当时我也在场，主持人 Andrew 是 boost 库的首席设计师之一（他后来有段时间当过我的导师）。他连夸 Stroustrup 会选日子，只遗憾演讲时间没有定在 11 点。

虽然我对 Stroustrup 的幽默感和谦虚的态度感到敬佩，但我也看出来 C++11 相对于像 Scheme 这样的语言，其实没有什么真正的“新东西”。大部分时候它是在改掉自己的一些坏毛病，然后向其它语言学习一些东西，然后把这些学习的痕迹掩盖起来。可是到最后，它仍然不可能达到其他语言那么原汁原味的效果。然而，由于 C++ 的普及程度高，现成的代码又多，它的地位和重要性还是一时难以动摇的。所以这些“[先辈的罪](#)”，我们恐怕要用好几代人的工作才能弥补。

那么 C++ 有什么其他语言没有的好东西呢？其实非常少。我还是有空再讲吧。

多学几种语言

我今天想说其实就是，没有任何一种语言值得你用毕生的精力去“精通”它。“精通”其实代表着“脑残”——你成为了一个高效的机器，而不是一个有自己头脑的人。你必须对每种语言都带有一定的怀疑态度，而不是完全的拥抱它。每个人都应该学习多种语言，这样才不至于让自己的思想受到单一语言的约束，而没法接受新的，更加先进的思想。这就像每个人都应该学会至少一门外语一样，否则你就深陷于自己民族的思维方式。有时候这种民族传统的思想会让你深陷无须有的痛苦却无法自拔。

Yoda 表示法错在哪里

在上一篇[博文](#)里，我提到了 Yoda 表示法。

Yoda Notation (Yoda 表示法)



它的含义是，在 C/C++ 里面使用这样的表达式顺序：

```
if ("blue" == theSky) ...
```

这是为了避免意外的写成：

```
if (theSky = "blue") ...
```

“Yoda 表示法”的名字来源于《星球大战》的 Yoda 大师。他说话的单词顺序相当奇特，比如：“Backwards it is, yes!”

一般认为

使用这个表示法是为了“变通”（workaround）C/C++ 的一个设计抉择：使用 = 来表示赋值，而使用 == 来表示比较。这个设计充分的展现了“先辈的罪”（Sins of our Forefathers）这一词汇的精髓。

我认为

使用 = 来表示赋值其实并不是真正的错误所在。真正的错误在于 C/C++ 的赋值语句不应该返回一个值。

也就是说，theSky = "blue" 的所有功能应该只是“赋值”这种“副作用”，副作用不应该具有“值”。即使你牵强附会说它有一个值，它的“值”也应该是 void（随之这个 void 会被类型检查所拒绝，因为它不是 if 所期望的 bool）。所以，一个良好的语言不应该允许你把 theSky = "blue" 放进 if (...) 的“条件”里面。如果你真的要赋值又要判断，它会迫使你把这拆开成两行：

```
theSky = "blue";
if (theSky) ...
```

更进一步。if (theSky) 这个写法其实也是一个先辈的罪。theSky 的类型是 string，它不应该可以直接被作为 bool 使用。if (...) 的条件应该必须是一个 bool。所以这里其实应该写成：

```
theSky = "blue";
if (theSky != NULL) ...
```

因为赋值语句永远不可能出现在条件的位置，所以之前的那种错误，即使我们使用 = 作为赋值操作符，也完全不可能出现。这样我们也就完全没必要用 Yoda 表示法了。

相反，如果我们只是把 = 换成像 Pascal 的 := 这样的赋值操作符，而保留其它的“特性”（赋值操作会返回值）的话，我们其实还是会遇到同样的问题：

```
if (theSky := "blue") ...
```

这里假设你想打 =，却不小心打成了 :=。机会虽然小，但是仍然有可能。而我推荐的解决方案，会让你故意想犯错误都不可能，编译器会拒绝接受你的程序。

所以你看到了，问题的根源其实不在于赋值操作的名字，而是有更深的原因。

几个超炫的专业词汇

从同事的[博客](#)上学会了解了几个超炫的专业词汇，激动不已。觉得这些词汇可以言简意赅的概括我的好几篇博文，自己的文章水准真是自愧不如。现在来见识一下真正大师级的英语词汇：

- Yoda Notation (Yoda 表示法)



在 C/C++ 里面使用这样的表达式顺序：

```
if ("blue" == theSky) ...
```

这是为了避免意外的写成：

```
if (theSky = "blue") ...
```

“Yoda 表示法”的名字来源于《星球大战》的 Yoda 大师。他说话的单词顺序相当奇特，比如：“Backwards it is, yes!”

同事认为：使用这个表示法是为了“变通”(wordaround) C/C++ 的一个设计抉择：使用 = 来表示赋值，而使用 == 来表示比较。这个设计充分的展现了“先辈的罪”(Sins of our Forefathers) 这一词汇的精髓。

关于 Yoda 表示法我有不同的见解，请参考[《Yoda 表示法错在哪里》](#)。

- Mental Speedbump (头脑减速杠)



由于设计的不协调性造成的用户的注意力分散。比如，很多软件喜欢弹出一个窗口问你“是否继续？”

- Pearl Effect (珍珠效应)



珍珠是怎么形成的？是由于异物掉进了蛤蚌的外套膜和贝壳之间的夹层里面，没法排出来。异物不断的刺激该处的外套膜，又痒又痛，于是外套膜分泌珍珠质把异物包围起来，包了一层又一层。久而久之，就形成了珍珠。

在软件里面也有很多这样的“珍珠”。由于早期的挠人的设计错误，用户不得不采用一些“变通方

案”（workaround）或者“附加过程”，这些就像珍珠质一样。久而久之，这些变通方案凝结起来，变成了“软件珍珠”，不了解它们来源的人都视之为宝贝。虽然产生于同样的原理，“软件珍珠”远远没有真正的珍珠那么好看。

（请比较：Sins of our Forefathers）

- Sins of our Forefathers（先辈的罪）



当时看起来合乎逻辑并且合情合理最后回顾起来却很傻b的历史遗留设计。

与“珍珠”相比，这些是有意识的加进去的，而不是不小心造成的，虽然这两者都会造成“变通”（workaround）。

- Katrina Effect（卡特里娜飓风效应）

这个词描述的是一种飓风过后完全重头来过的悲惨景象。这种现象现在经常出现在重装或者升级软件之后，或者Windows安装完软件之后要你重启机器（关掉所有窗口）。

- Workaround（变通）



因为开发过程的失败而让用户必须进行的一些操作。这些通常是设计失误。

- Jenga Code



当你加上一小块代码之后，就整个垮掉的那种代码。

Jenga是一种非常流行的party玩具，如图。它的工作原理是，先把那些小木条堆成一个规则的塔。然后，参

加游戏的人轮流从下面抽出一块（只能用一只手）来放在最上面。谁放上之后木塔垮掉了，谁就“胜利”了。之后这个人就要做其他人想出来的一些“惩罚”，跟真心话大冒险那些事情差不多。

- [Higgs-Bugson](#)



一种假想中的 bug。它一般是根据运行日志的少数记录和零星含糊的用户报告推测出来，但是在开发员的机器上很难重现。

- [Heisenbug](#)

$$\sigma_x \sigma_p \geq \frac{\hbar}{2},$$

当你试图观察它的时候就突然消失或者改变行为特征的 bug。

标准化试卷标记语言

(写另外一篇[博文](#)的时候发现跑题太多，所以现在把它独立出来另外写一篇。)

我本科的时候给我爸设计了一种“标准化试卷标记语言”（他是中学英语老师）。当时我写了一个1000来行的 Perl 脚本，可以把这种简单的标记语言转换成美观的 LaTeX 格式文档，并且带有友好的 Tk 图形界面。题目可以包括选择题，填空题，改错题，…… 这种语言的特点是，题目和答案都放在一起，所以出题的时候很直观。经过处理之后，答案与题目分离，并且被放到答案表里正确的位置上。这样出题的人就不会把答案放错位置，也不用担心如何排版。

比如，选择题的格式是这样：

```
# Our teacher told us that there _____ no end to learning.  
A. was      B. is *      C. has      D. had
```

每道题开头都是一个 #，正确答案后面加一个 *。在处理的时候这些 # 都会被变成题目编号，而有 * 标记的答案选项，会被放到答案表里面。

最有意思的是改错题。因为改错题是一个英语段落，某些行有错，但每行最多只能有一个错。所以我的设计是，在普通的段落里插入这样的记号：

```
This is an |extraordinary|extrordinary| sentence...
```

用以“引入错误”。左边是正确的方式，右边是错误的。这里使用 | 是因为这个符号不会在普通的英语文章里出现。| 的左右两边都可以是空白，用以表示“插入”与“删除”。比如：

```
I don't know how to play |the|| piano.
```

这样的题目显示出来之后是这样：

```
I don't know how to play piano.
```

然后答案里会显示：

在 play 和 piano 之间加 the

如果两个都不是空白，那就是“修改”，就像上面的例子。最后我会根据这些标记的位置把段落排版，让每行上面最多只有一个错。为了让段落的行看起来均匀，我使用了一种类似 TeX 的动态规划断行算法。它先算出多种断行方案的“难看程度”，然后从中选出最好看的一个。

现在回想起来，我那时候的设计其实是相当先进的。怎么就没想过把它做成产品卖给教育部呢，也许因为觉得这种技术会制造出更大量的题海，祸害更多的中学生 :-P 跟我的语言相比，现在一些 blog 系统用的 markup 语言（比如 markdown）真是小巫见大巫了。

一种新的操作系统设计

我一直在试图利用程序语言的设计原理，设计一种超越“Unix 哲学”的操作系统。这里是我的设想：

- 这种系统里面的程序间通信不使用无结构的字符串，而是使用带有类型和结构的数据。在这样的系统里面，Unix 和其它类似操作系统（比如 Windows）里的所谓“应用程序”的概念基本上完全消失。系统由一个个很小的“函数”组成，每个函数都可以调用另外一个函数，通过参数传递数据。每个函数都可以手动或者自动并发执行。用现在的系统术语打个比方，这就像是所有代码都是“库”代码，而不存在独立的“可执行文件”。
- 由于参数是数据结构而不是字符串，这避免了程序间通信繁琐的编码和解码过程。使得“进程间通信”变得轻而易举。任何函数都可以调用另一个函数来处理特定类型的数据，这使得像“OLE 嵌入”这样的机制变得极其简单。
- 所有函数由同一种先进的高级程序语言写成，所以函数间的调用完全不需要“翻译”。不存在 SQL injection 之类由于把程序当成字符串而产生的错误。
- 由于这种语言不允许应用程序使用“指针运算”，应用程序不可能产生 segfault 一类的错误。为了防止不良用户手动在机器码里面加入指针运算，系统的执行的代码不是完全的机器代码，而必须通过进一步的验证和转换之后才会被硬件执行。这有点像 JVM，但它直接运行在硬件之上，所以必须有一些 JVM 没有的功能，比如把内存里的数据结构自动换出到硬盘上，需要的时候再换进内存。
- 由于没有指针运算，系统可以直接使用“实地址”模式进行内存管理，从而不再需要现代处理器提供的内存映射机制以及 TLB。内存的管理粒度是数据结构，而不是页面。这使得内存访问和管理效率大幅提高，而且简化了处理器的设计。据 Kent Dybvig 的经验，这样的系统的内存使用效率要比 Unix 类的系统高一个数量级。
- 系统使用与应用程序相同的高级语言写成，至于“系统调用”，不过是调用另外一个函数。由于只有这些“系统驱动函数”才有对设备的“引用”，又因为系统没有指针运算，所以用户函数不可能绕过系统函数而非法访问硬件。
- 系统没有 Unix 式的“命令行”，它的“shell”其实就是这种高级语言的 REPL。用户可以在终端用可视化的结构编辑方式输入各种函数调用，从而启动进程的运行。所以你不需要像 Unix 一样另外设计一种毛病语言来“粘接”应用程序。
- 所有的数据都作为“结构”，保存在一个分布式的数据共享空间。同样的那个系统语言可以被轻松地发送到远程机器，调用远程机器上的库代码，执行任意复杂的查询索引等动作，取回结果。这种方式可以高效的完成数据库的功能，然而却比数据库简单很多。所谓的“查询语言”（比如 SQL, Datalog, Gremlin, Cypher）其实是多此一举，它们远远不如普通的程序语言强大。说是可以让用户“不需要编程，只提出问题”，然而它们所谓的“优化”是非常局限甚至不可能实现的，带来的麻烦远比直接编程还要多。逻辑式编程语言（比如 Prolog）其实跟 SQL 是一样的问题，一旦遇到复杂点的查询就效率低下。所以系统不使用关系式数据库，不需要 SQL，不需要 NoSQL，不需要 Datalog。
- 由于数据全都是结构化的，所以没有普通操作系统的无结构“文件系统”。数据结构可能通过路径来访问，然而路径不是一个字符串或者字符串模式。系统不使用正则表达式，而是一种类似 NFA 的数据结构，对它们的拆分和组合操作不会出现像字符串那样的问题，比如把 /a/b/ 和 /c/d 串接在一起就变成错误的 /a/b//c/d。
- 所有的数据在合适的时候被自动同步到磁盘，并且进行容错处理，所以即使在机器掉电的情况下，绝大部分的数据和进程能够在电源恢复后继续运行。
- 程序员和用户几乎完全不需要知道“数据库”或者“文件系统”的存在。程序假设自己拥有无穷大的空间，可以任意的构造数据。根据硬件的能力，一些手动的存盘操作也可能是必要的。
- 为了减少数据的移动，系统或者用户可以根据数据的位置，选择：1) 迁移数据，或者 2) 迁移处理数据的“进程”。程序员不需要使用 MapReduce, Hadoop 等就能进行大规模并行计算，然而表达能力却比它们强大很多，因为它们全都使用同一种程序语言写成。

我曾经以为我是第一个想到这个做法的人。可是调查之后发现，很多人早就已经做出了类似的系统。Lisp Machine 似乎是最接近的一个。[Oberon](#) 是另外一个。IBM System/38 是类似系统里面最老的一个。最近这些年出现的还有微软的 [Singularity](#)，另外还有人试图把 JVM 和 Erlang VM 直接放到硬件上执行。

所以这篇文章的标题其实是错的，这不是一种“新的操作系统设计”。它看起来是新的，只不过因为我们现在用的操作系统忘记了它们本该是什么样子。我也不该说它“超越了 Unix 哲学”，而应该说，所谓的 Unix 哲学其实是历史的倒退。

Markdown 的一些问题

把我之前的博文基本上转换成了 markdown 格式。我发现 markdown 虽然在编辑器里看起来比 HTML 清晰一些，但也有一些不足。

这些 markup 语言的格式都有点像我本科的时候给我爸做的一种“[标准化试卷标记语言](#)”（因为他是中学英语老师）。当时我写了一个1000来行的 Perl 脚本，可以把这种简单的标记语言转换成美观的 LaTeX 格式文档，并且带有友好的 Tk 图形界面。现在回想起来，我那时候的设计就已经相当先进了。跟我的语言相比，这些 blog 用的 markup 语言真是小巫见大巫了，而且问题多多。有点跑题了，还是回头来看看 markdown 的问题吧。

- Markdown 实际上采用的是类似 Python 和 Haskell 的 layout 语法。

我已经在一篇[英文博文](#)里提到了 layout 语法的多种问题。因为空格的数量决定了文档的结构，这种文档格式相当的“脆弱”。稍微少打一两个空格，就会出现不可预测的结果。这种现象在“itemize”内部的代码块最容易出现。因为每个 item 带来了缩进，所以内部的代码必须比 item 的缩进多4个空格，才能被排到正确的位置。比如我转换博文的时候多次出现以下的情况：

3. Renaming. We seldom choose the best names on the first shot, and good names make programs self-explanatory, so renaming is a very important and commonplace action. But in the following simple Haskell program, if we change the name from "helloworld" to "hello" and don't re-indent the rest of the lines, we will get a parse error.

helloworld z = let x = 1

```
y = 2 in  
x+y+z
```

Because the code becomes the following after the renaming, and the second line will no longer be aligned to "x = ...", and that confuses the parser.

这里的问题是，代码里的第一行 helloworld z = let x = 1 因为缩进不够，被放到了代码块外面。但是为了准确的缩进所耗费的精力，其实比直接打 `<pre>` 这样的 tag 还要多。

- 特殊字符的选择不合理

markdown 对特殊字符的使用不大合理。我多次发现文档段落整段的变成斜体，就是因为原来的文档里出现了 `x*y` 这样的表达式。在程序员的世界里，“乘法”显然比“强调”更加频繁。把 * 用于标记“强调”，实际上把一个非常有用的字符用在了很不频繁的用途。

- 表达力相当有限

在很多细节上，markdown 并不能表达我想要的格式。比如它不能正确的插入断行 `
`。如果你有两块紧接在一起的代码，但你不想把它们连在一起，markdown 非要给你连在一起…… 于是我就发现自己加入了越来越多的 HTML。

这在图片的语法上就更加明显，markdown 引入了 `![alt](image url)` 这样的格式，其实比起 HTML 还要难看和不一致。比如现在它仍然无法表达图片的大小，这是相当重要的信息。所以我觉得 markdown 的语法已经显示出了它的弱点，如果它要表达更复杂的信息，就会变得比 HTML 还要难记，难看。所以对于图片，我觉得还不如直接用 HTML 的 ``。

所以总的感觉是 markdown 引入了太多的“语法”，以至于稍微复杂一点的信息表达起来还不如 HTML 来的直接。现在就这样先凑合着吧。也许过段时间自己设计一个格式。

谈程序的“通用性”

在现实的软件工程中，我经常发现这样的一种现象。本来用很简单的代码就可以解决的问题，却因为设计者过分的关注了“通用性”，“可维护性”和“可扩展性”，被搞得绕了几道弯，让人琢磨不透。

这些人的思维方式是这样的：“将来这段代码可能会被用到更多的场合，所以我现在就考虑到扩展问题。”于是乎，他们在代码中加入了各种各样的“框架结构”，目的是为了在将来有新的需要的时候，代码能够“不加修改”就被用到新的地方。

我并不否认“通用性”的价值，实际上我的某些程序通用性非常之强。可是很多人所谓的“通用性”，其实达到的是适得其反的效果。这种现象通常被称为“过度工程”(over-engineer)。关于过度工程，有一个有趣的故事：

<http://www.snopes.com/business/genius/spacepen.asp>

传说 1960 年代美俄“太空竞赛”的时候，NASA 遇到一个严重的技术问题：宇航员需要一支可以在外太空的真空中写字的钢笔。最后 NASA 耗资 150 万美元研制出了这样的钢笔。可惜这种钢笔在市场上并不行销。

俄国人也遇到同样的问题。他们使用了铅笔。

这个故事虽然是假的，但是却具有伊索寓言的威力。现在再来看我们的软件行业，你也许会发现：

1. 代码需要被“重用”的场合，实际上比你想象的要少

我发现很多人写程序的时候连“眼前特例”都没做好，就在开始“展望将来”。他们总是设想别人会重用这段代码。而实际上，由于他们的设计过于复杂，理解这设计所需的脑力开销已经高于从头开始的代价，所以大部分人其实根本不会去用他们的代码，自己重新写一个就是了。也有人到后来发现，之前写的那段代码，连自己都看不下去了，恨不得删了重来，就不要谈什么重用了。

2. 修改代码所需要的工作实际上比你想象的要少

还有一种情况是，这些被设计来“共享”的代码，其实根本没有被用在很多的地方，所以即使你完全手动的修改它们也花不了很多时间。现在再加上 IDE 技术的发展和各种先进的 refactor 工具，批量的修改代码已经不是特别麻烦的事情。曾经需要在逻辑层面上进行的可维护性设计，现在有可能只需要在 IDE 里面点几下鼠标就轻松完成。所以在考虑设计一个框架之前，你应该同时考虑到这些因素。

3. “考虑”到了通用性，并不等于你就准确地“把握”住了通用性

很多人考虑到了通用性，却没有准确的看到，到底是哪一个部分将来可能需要修改，所以他们的设计经常抓不住关键。当有新的需要出现的时候，才发现原来设想的可能变化的部分，其实根本没有变，而原来以为不会变的地方却变了。

能够准确的预测将来的需要，能够从代码中抽象出真正通用的框架，是一件非常困难的事情。它不止需要有编程的能力，而且需要对真实世界里的事物有强大的观察能力。很多人设计出来的框架，其实只是照搬别人的经验，却不能适应实际的需要。在 Java 世界里的很多 design pattern，就是这些一知半解的人设计出来的。

4. 初期设计的复杂性

如果在第一次的设计中就过早的考虑到将来，由此带来的多余的复杂性，有可能让初期的设计就出现问题。所以这种对于将来的变化的考虑，实际上帮了倒忙。本来如果专注于解决现在的问题，能够得到非常好的结果。但是由于“通用性”带来的复杂度，设计者的头脑每次都要多转几道弯，所以它无法设计出优雅的程序。

5. 理解和维护框架性代码的开销

如果你设计了框架性的代码，每个程序员为了在这个框架下编写代码，都需要理解这种框架的构造，这带来了学习的开销。一旦发现这框架有设计问题，依赖于它的代码很有可能需要修改，这又带来了修改的开销。所以加入“通用性”之后，其实带来了更多的工作。这种开销能不能得到回报，依赖于以上的多种因素。

所以在设计程序的时候，我们最好是先把手上的问题解决好。如果发现这段代码还可以被用在很多别的地方，到时候再把框架从中抽象出来也不迟。

什么是启发

我喜欢用“启发”这个词。比如我经常会对有人说：“你启发了我。”然而听到这话的人有时候不明白我的意思，自以为高我一筹，于是顿显傲气。其实我用“启发”这个词，是有深刻含义的。“启发”的意思并不等于“我没有你懂得多”或者“你比我聪明”，而是一个很含糊的词。

如果 A 受到了 B 启发，有几种可能性：

1. B 做了一件很聪明的事情，所以从正面启发了 A
2. B 做了一件很笨的事情，所以从反面启发了 A
3. B 做了一件不好也不坏的事情，但是这个事情正好触发了 A 事先想的一个问题的答案

这就是为什么“美丽心灵”里的 John Nash 在酒吧看到一个美女之后，解决了一个重要的问题，然后对她说“谢谢”，让大家都莫名其妙。

Richard Feynman 也提到：“在你的头脑里随时准备好12个问题。每当发生一件有趣的事情，就检查一下其中是否有问题可以由此获得线索。久而久之，人们就会称你为天才。”

孔夫子所谓的“三人行必有我师”，也就是这个意思吧。

Scheme 编程环境的设置



```
;; Iterate diff-list on the list of changes
(define find-moves
  (lambda (changes)
    (set! *moving* #t)
    (set! *diff-hash* (make-hasheq))
    (let loop ([workset changes]
              [finished '()])
      (diff-progress "I")
      (letv ([dels (filter (predand del? big-change?) workset)])
        [adds (filter (predand ins? big-change?) workset)]
        [rest (set- workset (append dels adds))])
        [ls1 (sort (map Change-old dels) node-sort-fn)]
        [ls2 (sort (map Change-new adds) node-sort-fn)]
        [(m c) (diff-list ls1 ls2)])
        [new-moves (filter mov? m)])
      (cond
        [(null? new-moves)
         (let ([all-changes (append workset finished)])
           (apply append (map deframe-change all-changes)))]
        [else
         (let ([new-changes (filter (negate mov?) m)])
```

介绍了这么久的 Scheme，却没有讲过如何配置一个高效的 Scheme 的编程环境。有些人开始学习 Scheme 的时候感觉无从下手，所以今天讲一下它的配置。

Scheme 的配置有很多种方式，我不想介绍太多东西，免得有人看花了眼，所以这里只介绍一下我自己的配置。我不大喜欢像 [Quack](#) 一类的复杂的环境，因为它们经常有很多多余的功能，却缺少我想要的功能。一旦我想修改它们，又到处出问题。我的配置很简约，我用它写了几千行的超高难度的代码，翻来覆去的改，感觉效率非常高，也没有觉得缺少什么特别重要的东西。

现在我就一步一步的介绍我的配置。

安装 Scheme

Chez Scheme

世界上最快，最成熟可靠的 Scheme 实现是 R. Kent Dybvig 所作的 Chez Scheme。它可以把 Scheme 编译成机器代码，运行速度非常高。Chez Scheme 曾经是商业软件，价格昂贵，然而现在却开源了，并且可以免费使用。你可以在这里下载 Chez Scheme 的源代码：

<https://github.com/cisco/ChezScheme>

编译安装很快很方便，在 Linux 和 Mac 系统基本就是这样：

```
./configure
make
sudo make install
```

整个编译安装过程只需要30秒。这是世界上最快编译自己全套系统的编译器。

Racket

如果你对性能没有特别高的需求，主要用于学习，也可以用 Racket。它可以在[这里](#)下载：

<http://racket-lang.org>

安装应该很容易。Ubuntu 也自带了 Racket，所以可以直接让系统安装它。

设置 Paredit mode

我编辑 Scheme 的时候都用 Emacs。我使用一个叫做 Paredit mode 的插件。它可以让你“半结构化”式的编辑 Scheme 和其它的 Lisp 文件。开头你可能会有点不习惯，可是一旦习惯了，你就再也离不开它。

Paredit mode 可以在[这里](#)下载：

<http://mumble.net/~campbell/emacs/paredit.el>

下载之后，把它放到一个目录里，比如 `~/.emacs.d`，然后打开 `~/.emacs` 配置文件，加入如下设置：

```
(add-to-list 'load-path "~/.emacs.d")
(autoload 'paredit-mode "paredit"
  "Minor mode for pseudo-structurally editing Lisp code."
  t)
```

这样，只要你使用 `M-x paredit-mode` 就可以自动载入这个模式。具体的操作方式可以看它的说明（按 `C-h m` 查看“模式帮助”），我下面也会简单说一下。

设置 scheme mode

我一般就用系统自带的 Scheme 模式，叫 cmuscheme。但是为了方便，我自己写了几个函数，用于在执行 Scheme 代码的时候自动启动解释器，并且打开解释器窗口。你基本只需要把下面的代码拷贝到你的 .emacs 文件里就行：

```
;;;;;;
;; Scheme
;;;;;;

(require 'cmuscheme)

;; push scheme interpreter path to exec-path
(push "/Applications/Racket/bin" exec-path)

;; scheme interpreter name
(setq scheme-program-name "racket")

;; bypass the interactive question and start the default interpreter
(defun scheme-proc ()
  "Return the current Scheme process, starting one if necessary."
  (unless (and scheme-buffer
               (get-buffer scheme-buffer)
               (comint-check-proc scheme-buffer))
    (save-window-excursion
      (run-scheme scheme-program-name)))
  (or (scheme-get-process)
      (error "No current process. See variable `scheme-buffer'")))

(defun switch-other-window-to-buffer (name)
  (other-window 1)
  (switch-to-buffer name)
  (other-window 1))

(defun scheme-split-window ()
  (cond
    ((= 1 (count-windows))
     (split-window-vertically (floor (* 0.68 (window-height))))
     ;; (split-window-horizontally (floor (* 0.5 (window-width))))
     (switch-other-window-to-buffer "*scheme*")
     ((not (member "*scheme*"
                   (mapcar (lambda (w) (buffer-name (window-buffer w)))
                           (window-list))))
      (switch-other-window-to-buffer "*scheme*"))))
    (defun scheme-send-last-sexp-split-window ()
      (interactive)
      (scheme-split-window)
      (scheme-send-last-sexp))

    (defun scheme-send-definition-split-window ()
      (interactive)
      (scheme-split-window)
      (scheme-send-definition))

    (add-hook 'scheme-mode-hook
              (lambda ()
                (paredit-mode 1)
                (define-key scheme-mode-map (kbd "<f5>") 'scheme-send-last-sexp-split-window)
                (define-key scheme-mode-map (kbd "<f6>") 'scheme-send-definition-split-window)))
```

我的配置会在加载 Scheme 文件的时候自动载入 Paredit mode，并且把 F5 键绑定到“执行前面的S表达式”。这样设置的目的是，我只要把光标移动到一个S表达式之后，然后用一根手指头按 F5，就可以执行程序。够懒吧。

Paredit mode 的简单使用方法

Paredit mode 是一个很特殊的模式。它起作用的时候，你不能直接修改括号。这样所有的括号都保持完整的匹配，不可能出现语法错误。但是这样一个问题，如果你要把一块代码放进另一块代码，或者从里面拿出来，就不是很方便了。

为此，Paredit mode 提供了几个非常高效的编辑方式。我平时只使用两个：

1. C-right: 也就是按住 Ctrl 再按右箭头。它的作用是让光标右边的括号，“吞掉”下一个S表达式。

比如，`(a b c) (d e)`。你把光标放在 `(a b c)` 里面，然后按 `C-right`。结果就是 `(a b c (d e))`。也就是把 `(d e)` 被整个“吞进”了 `(a b c)` 里面。

2. M-r: 去掉外层代码。

这在你需要去掉外层的 let 等结构的时候非常有用。比如，如果你的代码看起来是这样：

```
(let ([x 10]
      (* x 2))
```

当你把光标放在 `(* x 2)` 的最左边，然后按 M-r，结果就变成了

```
(* x 2)
```

也就是把外面的 `(let ([x 10]) ...)` 给“掀掉”了。

其它的一些按键虽然也有用，不过我觉得这两个是最有用的，甚至不可缺少的。有些其他的自动匹配括号的模式，没有提供这种按键，所以用起来很别扭。

设置括号颜色

很多人看见 Lisp 就怕了，就是因为它看起来括号太多。可是这样的语法，却是有很大的好处的（参考这篇博文《谈语法》）。如果你真的觉得括号碍眼，你可以稍微调整一下括号的颜色，比如淡灰色。这样括号看起来就没有那么显眼了。

你只需要下载这个 el，放到你的 .emacs.d:

<https://www.dropbox.com/s/v0ejctd1agrt95x/parenface.el>

然后在 .emacs 里面加入两行：

```
(require 'parenface)
(set-face-foreground 'paren-face "DimGray")
```

然后再打开 Scheme 代码的时候，你就会看到是这个样子：

```
; call-by-name compiler to S, K, I
(define compile
  (lambda (exp)
    (pmatch exp
      [ ,x (guard (symbol? x)) x]
      [( ,M ,N) ` ( ,(compile M) ,(compile N)) ]
      [(lambda ( ,x) ( ,M ,y))
        (guard (eq? x y) (not (occur-free? x M))) (compile M)]
      [(lambda ( ,x) ,y) (guard (eq? x y)) 'I]
      [(lambda ( ,x) ( ,M ,N)) (guard (or (occur-free? x M) (occur-free? x N)))
        ` ((S ,(compile `(lambda ( ,x) ,M))),(compile `(lambda ( ,x) ,N)))]
      [(lambda ( ,x) ,M) (guard (not (occur-free? x M))) ` (K ,(compile M))]
      [(lambda ( ,x) ,M) (guard (occur-free? x M))
        (compile `(lambda ( ,x) ,(compile M))))]))))
```

好了，这就是我写 Scheme 的所有配置了。希望这些有所帮助。

我为什么离开 Cornell

很多人都知道，我曾经在 Cornell 博士就读，两年之后转学到了 Indiana 大学。几乎所有人，包括 Indiana 大学的人都感觉奇怪，为什么会有人从 Cornell 这样的“牛校”转学到 Indiana。我曾经在之前的博文里提到 Cornell 的情况，比如学生一上课就忙着抄笔记，作业压得喘不过气，等等。那些都是实际的情况，所以我没什么必要为我的“母校”说好话。

离开 Cornell 之后，看到有人在 facebook 上成立了一个“Cornell 痛恨者协会”。其中一个人写到：

“Cornell 说要教你游泳，就把你推进池塘里，任凭你扑腾挣扎。等你快扑腾到岸边的时候，它忽然拿起一块大石头砸在你头上，然后继续等着你上岸。当你再次接近岸边的时候，它又拿起一个榔头敲在你头上，这样你就可以死了，可是 Cornell 仍然继续等着你游上岸边……”

这就是一个非常形象的，对我在 Cornell 的两年的总结。现在看看我在 Indiana 学到了什么，而 Cornell 教会了我什么，感觉简直一个天上一个地下。Dan Friedman 和 R. Kent Dybvig，他们的教育真的像是爱因斯坦所说的，像是珍贵的礼物，而不是沉重的负担。他们教会我的东西，让我不再在乎任何“牛校”的博士学位甚至教授职位，不管是 Cornell, Stanford, Berkeley, MIT 还是 Harvard,

所谓的“牛校”，恐怕都是这样吧。学生对于它们只是一种成为“牛校”的工具。你拼着命要进来，好我让你进来。但是我不教你，我让你拼死的做作业。如果你做出来了，我就拿最偏最扯淡的试卷来考你。如果你通过了所有这些，那我就给你一个学位。你得到了这样的“荣誉”，自然就会说“我的学校很牛”。你不敢说它不牛，因为那样就是说你也不牛了。所以这样的学校其实什么也不用干，你能学会东西能毕业，全都是靠你自己，到时候你却要把功劳都归到学校头上。天底下就是有这样好的生意。

曾经有一个 Cornell 的校友跟我是朋友。当我提到 Cornell 的一些事，他总是像个老师一样，上气不接下气地“教育”我，也就是说类似家丑不可外扬的意思吧。“牛校”就是一种传染病，在你还没进去之前就已经埋下病种，当你进去之后它就开始蔓延，等你毕业很多年，它仍然与你同在。

测试驱动开发

现在的很多公司，包括 Google 和我现在的公司 Coverity，都喜欢一种“测试驱动的开发”(test-driven development)。它的原理是，在写程序的时候同时写上自动化的“单元测试”(unit test)。在代码修改之后，这些测试可以批量的被运行，这样就可以避免不应该出现的错误。

这不是一个坏主意。我在 Kent 的编译器课程上也使用了很多测试。它们在编译器的开发中是不可缺少的。编译器是一种极其精密的程序，微小的改动都可能带来重大的错误。所以编译器的项目一般都含有大量的测试。

然而测试的构建，应该是在程序主体已经成形的情况下才能进行。如果程序属于创造性设计，主体并未成形，过早的加入测试反而会大幅度的降低开发效率。所以当我给 Google 开发 Python 静态分析的时候，我几乎没有使用任何测试。虽然组里的成员催我写测试，但是我却知道那只会降低我的开发效率，因为这个程序在几个星期的过程中，被我推翻重来了好几次。要是我一开头就写上测试，这些测试就会碍手碍脚，阻碍我大幅度的修改代码。

测试的另一个副作用是，它让很多人对测试有一种盲目的依赖心理。改了程序之后，把测试跑一遍没出错，就以为自己的代码是正确的。可是测试其实并不能保证代码的正确，即使完全“覆盖”了也是一样。覆盖只是说你的代码被测试碰到过了，可是它在什么条件下碰到的却没法判断。如果实际的条件跟测试时的条件不同，那么实际运行中仍然会出现问题。测试的条件往往是“组合爆炸”的数量级，所以你不可能测试所有的情况。唯一能可靠的方法是使用严密的“逻辑推理”，证明它的正确。

当然我并不是让你用 ACL2 或者 Coq 这样的定理证明软件。虽然它们的逻辑非常严密，但是用它们来证明复杂的软件系统，需要顶尖的程序员和大量的时间。即使如此，由于理论的限制，程序的正确性有可能根本无法证明。所以我这里说的“逻辑推理”，只是局部的，人力的，基本的逻辑推理。

很多人写程序只是凭现象来判断，而不能精密的分析程序的逻辑，所以他们修改程序经常“治标不治本”。如果程序出了问题了，他们的办法是看看哪里错了，也不怎么理解，就改一下让它不再出错，最多再把所有测试跑一遍。或者再加上一些新的测试，以保证这个地方下次不再出问题。

这种做法的结果是，程序里出现大量的“特殊情况”和“创可贴”。把一个“虫子”按下去，另一个虫子又冒出来。忙活来忙活去，最后仍然不能让程序满足“所有情况”。其实能够“满足所有情况”的程序，往往比能够“满足特殊情况”的程序简单很多。这是一个很奇怪的事情：能做的事越多，代码量却越少。也许这就叫做程序的“美”，它跟数学的“美”其实是一回事。

美的程序不可能从修修补补中来。它必须完美的把握住事物的本质，否则就会有许许多多无法修补的特例。其实程序员跟画家差不多，画家如果一天到头蹲在家里，肯定什么好东西也画不出来。程序员也一样，蹲在家里面对电脑，其实很难写出什么好的代码。你必须出去观察事物，寻找“灵感”，而不只是写代码。在修改代码的时候，你必须用“心灵之眼”看见代码背后所表达的事物。这也是为什么很多高明的程序员不怎么用调试器(debugger)的原因。他们只是用眼睛看着代码，然后闭上眼，脑海里浮现出其中信息的流动，所以他们经常一动手就能改到正确的地方。

Currying 的局限性

很多基于 lambda calculus 的程序语言，比如 ML 和 Haskell，都习惯用一种叫做 currying 的手法来表示函数。比如，如果你在 Haskell 里面这样写一个函数：

```
f x y = x + y
```

然后你就可以这样把链表里的每个元素加上 2：

```
map (f 2) [1, 2, 3]
```

它会输出 [3, 4, 5]。

注意本来 f 需要两个参数才能算出结果，可是这里的 (f 2) 只给了 f 一个参数。这是因为 Haskell 的函数定义的缺省方式是“currying”。Currying 其实就是用“单参数”的函数，来模拟多参数的函数。比如，上面的 f 的定义在 Scheme 里面相当于：

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

它是说，函数 f，接受一个参数 x，返回另一个函数（没有名字）。这个匿名函数，如果再接受一个参数 y，就会返回 $x + y$ 。所以上面的例子里面，(f 2) 返回的是一个匿名函数，它会把 2 加到自己的参数上面返回。所以把它 map 到 [1, 2, 3]，我们就得到了 [3, 4, 5]。

在这个例子里面，currying 貌似一个挺有用的东西，它让程序变得“简短”。如果不使用 currying，你就需要制造另一个函数，写成这个样子：

```
map (\y->f 2 y) [1, 2, 3]
```

这就是为什么 Haskell 和 ML 的程序员那么喜欢 currying。这个做法其实来源于最早的 lambda calculus 的设计。因为 lambda calculus 的函数都只有一个参数，所以为了能够表示多参数的函数，有一个叫 Haskell Curry 的数学家和逻辑学家，发明了这个方法。

当然，Haskell Curry 是我很尊敬的人。不过我今天想指出的是，currying 在程序设计的实践中，其实并不是想象中的那么好。大量使用 currying，其实会带来程序难以理解，复杂性增加，并且还可能因此引起意想不到的错误。

不用 currying 的写法 (\y->f 2 y) 虽然比起 currying 的写法 (f 2) 长了那么一点，但是它有一点好。那就是你作为一个人（而不是机器），可以很清楚的从 “\y->f 2 y” 这个表达式，看到它的“用意”是什么。你会很清楚的看到：

“f 本来是一个需要两个参数的函数。我们只给了它第一个参数 2。我们想要把 [1, 2, 3] 这个链表里的每一个元素，放进 f 的第二个参数 y，然后把 f 返回的结果一个一个的放进返回值的链表里。”

仔细看看上面这段话说了什么吧，再来看看 (f 2) 是否表达了同样的意思？注意，我们现在的“重点”在于你，一个人，而不在于计算机。你仔细想，不要让思维的定势来影响你的判断。

你发现了吗？(f 2) 并不完全的含有 \y->f 2 y 所表达的内容。因为单从 (f 2) 这个表达式（不看它的定义），你看不到“f 总共需要几个参数”这一信息，你也看不到 (f 2) 会返回什么东西。f 有可能需要 2 个参数，也有可能需要 3 个，4 个，5 个…… 比如，如果它需要 3 个参数的话，map (f 2) [1, 2, 3] 就不会返回一个整数的链表，而会返回一个函数的链表，它看起来是这样：[(\z->f 2 1 z), (\z->f 2 2 z), (\z->f 2 3 z)]。这三个函数分别还需要一个参数，才会输出结果。

这样一来，表达式 (f 2) 含有的对“人”有用的信息，就比较少了。你不能很可靠地知道这个函数接受了一个参数之后会变成什么样子。当然，你可以去看 f 的定义，然后再回来，但是这里有一种“直觉”上的开销。如果你不能同时看见这些信息，你的脑子就需要多转一道弯，你就会缺少一些重要的直觉。这种直觉能帮助你写出更好的程序。

然而，currying 的问题不止在于这种“认知”的方面，有时候使用 curry 会直接带来代码复杂性的增加。比如，如果你的 f 定义不是加法，而是除法：

```
f x y = x / y
```

然后，我们现在需要把链表 [1, 2, 3] 里的每一个数都除以 2。你会怎么做呢？

```
map (f 2) [1, 2, 3]
```

 肯定不行，因为 2 是除数，而不是被除数。熟悉 Haskell 的人都知道，可以这样做：

```
map (flip f 2) [1, 2, 3]
```

flip 的作用是“交换”两个参数的位置。它可以被定义为：

```
flip f x y = f y x
```

但是，如果 f 有 3 个参数，而我们需要把它的第 2 个参数 map 到一个链表，怎么办呢？比如，如果 f 被定义为：

```
f x y z = (x - y) / z
```

稍微动一下脑筋，你可能会想出这样的代码：

```
map (flip (f 1) 2) [1, 2, 3]
```

能想出这段代码说明你挺聪明，可是如果你这样写代码，那就是缺乏一些“智慧”。有时候，好的程序其实不在于显示你有多“聪明”，而在于显示你有多“笨”。现在我们就来看看笨一点的代码：

```
map (\y -> f 1 y 2) [1, 2, 3]
```

现在比较一下，你仍然觉得之前那段代码很聪明吗？如果你注意观察，就会发现 $(flip (f 1) 2)$ 这个表达式，是多么的晦涩，多么的复杂。

从 $(flip (f 1) 2)$ 里面，你几乎看不到自己想要干什么。而 $\lambda y -> f 1 y 2$ 却很明确的显示出，你想用 1 和 2 填充掉 f 的第一，三号参数，把第二个参数留下来，然后把得到的函数 map 到链表 $[1, 2, 3]$ 。仔细看看，是不是这样的？

所以你花费了挺多的脑力才把那使用 currying 的代码写出来，然后你每次看到它，还需要耗费同样多的脑力，才能明白你当时写它来干嘛。你是不是吃饱了没事干呢？

练习题：如果你还不相信，就请你用 currying 的方法（加上 $flip$ ）表达下面这个语句，也就是把 f 的第一个参数 map 到链表 $[1, 2, 3]$ ：

```
map (\y -> f y 1 2) [1, 2, 3]
```

得到结果之后再跟上面这个语句对比，看谁更加简单？

到现在你也许注意到了，以上的“笨办法”对于我们想要 map 的每一个参数，都是差不多的形式；而使用 currying 的代码，对于每个参数，形式有很大的差别。所以我们的“笨办法”其实才是以不变应万变的良策。

才三个参数，currying 就显示出了它的弱点，如果超过三个参数，那就更麻烦了。所以很多人为了写 currying 的函数，特意把参数调整到方便 currying 的顺序。可是程序的设计总是有意想不到的变化。有时候你需要增加一个参数，有时候你又想减少一个参数，有时候你又会有别的用法，导致你需要调整参数的顺序……事先安排好的那些参数顺序，很有可能不能满足你后来的需要。即使它能满足你后来的需要，你的函数也会因为 currying 而难以看懂。

这就是为什么我从来不在我的 ML 和 Haskell 程序里使用 currying 的原因。古老而美丽的理论，也许能够给我带来思想的启迪，可是未必就能带来工程中理想的效果。

惰性求值

从之前的几篇博文里面你也许已经看到了，Haskell 其实是问题相当严重的语言，然而这些问题却没有引起足够的重视。我能看到的 Haskell 的问题在于：

- 复杂的基于缩进的语法，使得任何编辑器都不能高效的编辑 Haskell 程序，并且使得语法分析难度加倍。对这个观点，请参考我的博文《谈语法》以及我的英文博文《Layout Syntax Considered Harmful》。
- “纯函数式”的语义以及 monad 其实不是好东西。对此请参考博文《对函数式语言的误解》。
- Haskell 所用的 Hindley-Milner 类型系统，其实含有一个根本性的错误。对此请参考《Hindley-Milner 类型系统的根本性错误》。
- Haskell 所用的 type class，其实跟一般语言（比如 Java）里面的重载（overloading）并没有本质区别。你看到的区别都是因为 Hindley-Milner 系统和重载混合在一起产生的效果。type class 并不能比其它语言里的重载做更多的事。

这样一来，好像 Haskell 的“特征”，要么是错误的，要么就不是自己的。可是现在我再给它加上一棵稻草：Haskell 的惰性求值（lazy evaluation）方式，其实大大的限制了它的运行效率，并且使得它跟并行计算的目标相矛盾。

这是一个对我已经非常明显的问题，所以我只简要的说明一下。惰性求值的方式，使得我们在“需要”一个变量的时候，总是有两种可能性：1) 这个变量在这之前已经被求值，所以可以直接取值 2) 这个变量还没有被求值，也就是说它还是一个 thunk，我们必须启动对它的求值。

可能你已经发现了，这其实带来了类型系统的混乱。任何类型，不管是 Int, Bool, List, ... 或者自定义数据类型，都多出了这么一个东西：thunk。它表示的是“还没有求值的计算”。Haskell 程序员一般把它叫做“bottom”，写作 \perp 。它的意思是：死循环。因为任何 thunk 都有可能 1) 返回一个预定的类型的值，或者 2) 导致死循环。

这有点像 C++ 和 Java 里的 null 指针，因为 null 可以被作为任何其他类型使用，却又不具有那种类型的特征，所以会产生意想不到的问题。 \perp 给 Haskell 带来的问题没那么严重，但却一样的不可预料，难以分析和调试。对于 Haskell 来说，有可能出现这样的事情：明明写了一个很小的函数，觉得应该不会花很多时间。结果呢，因为它对某个变量取值，间接的触发了一段很耗时的代码，所以等了老半天还没返回。想知道是哪里出了问题，却难以发现线索，因为这函数并没有直接或者间接的调用那段耗时的代码，而是这个变量的 thunk 启动了那段代码。这就导致了程序的效率难以分析：被“惰性”搁在那里的计算，有可能在出乎你意料的地方爆发。这就是所谓“平时不烧香，临时抱佛脚。”

这种不确定性，并没有带来总体计算开销的增加。然而“惰性”却在另外一方面带来了巨大的开销，这就是“问问题”的开销。每当看到一个变量，Haskell 都会问它一个问题：“你被求值了没有？”即使这变量已经被求值，而且已经被取值一千万次，Haskell 仍然会问这个问题：“你被求值了没有？”问一个变量这问题可能不要紧，可是 Haskell 会问几乎所有的变量这个问题，反复的问这个问题。这就累积成了巨大的开销。跟我在另一篇博文里谈到的“解释开销”差不多，这种问题是“运行时”的，所以没法被编译器“优化”掉。

具有讽刺意味的是，Haskell 这种“纯函数式语言”的惰性求值所需要的 thunk，全都需要“副作用”才可以更新，所以它们必须被放在内存里面，而不是寄存器里面。如果你理解了我写的《对函数式语言的误解》，你就会发现连 C 程序里面的“副作用”也没有 Haskell 这么多。这样一来，处理器的寄存器其实得不到有效的利用，从而大大增加了内存的访问。我为什么可以很确信的告诉你这个呢？因为我曾经设计了一个寄存器分配算法，于是开会的时候我问 GHC 的实现者们，你们会不会对一个新的寄存器分配算法感兴趣，我可以帮你们加到 GHC 里面。结果他们说，我们不需要，因为 Haskell 到处都是 thunk，根本就没什么机会用寄存器。

所以，问太多问题，没法充分利用寄存器，这使得 Haskell 在效率上大打折扣。

然后我们来看看，为什么惰性求值会跟并行计算的目标相冲突。这其实很明显，它的原因就在于“惰性求值”的定义。惰性求值说：“到需要我的时候再来计算我。”而并行计算说：“到需要你的时候，你最好已经被某个处理器算出来了。”所以你看到了，并行计算要求你“勤奋”，要求你事先做好准备。而惰性求值本来就是很“懒”，怎么可能没事找事，先把自己算出来呢？由于这个问题来自于“惰性求值”的定义，所以这是不可调和的矛盾。

所以，惰性求值不管是在串行处理还是在并行处理的时候，都会带来效率上的大打折扣，它是一个很鸡肋的语言特征。

虽然惰性求值不能给我们带来直接的益处，但它背后的理论思想却可以启发另外的设计。如果你想真的了解惰性求值的原理，可以先看一下我写的一个惰性求值的解释器。看看如何在不到 40 行代码之内，实现 Haskell 语义的精髓：

<https://github.com/yinwang0/lightsabers/blob/master/interp-lazy.rkt>

Hindley-Milner 类型系统的根本性错误

之前的一个时间，我曾经公开过这样一段[幻灯片](#)，它是2012年10月的时候，我在 Indiana 大学做的最后一次演讲。由于当时的委婉，我并没有直接说出这些结论的重要性。其实它揭示了 ML 和 Haskell 这类基于 Hindley-Milner 类型系统的语言的一个根本性的错误。

这个错误来源于对一阶逻辑的“全称量词”（universal quantifier，通常写作 \forall ）与程序函数之间关系的误解。在 HM 系统里面，多态（polymorphic）的函数能够被推导为含有全称量词的类型，比如 $\lambda x \rightarrow x$ 的类型被推导为 $\forall a. a \rightarrow a$ ，但 HM 系统决定这个全称量词的位置的方式，却是没有原则的。这就导致了类型变量（type variable）的作用域（scope）的偏差。

我的研究显示，这个错误来源于 HM 系统最初的一项重要的设计，叫做 let-polymorphism。如果右边的函数是一个多态的函数，比如：

```
let f = \x->x in  
...
```

let-polymorphism 总是会把全称量词的位置确定在 let 的“=”右边。然而这是一个非常错误的做法，它的错误程度近似于早期的 Lisp 所采用的 dynamic scoping。这样做的结果是，全称量词的位置会随着程序的“格式”，而不是程序的“结构”，而变化。至于什么是 dynamic scoping，你可以参考我的这篇博文。

为了弥补这个错误，30多年来，许多的人发表了许多的论文，提出了很多的“改进措施”，比如 value restriction，MLF，等等。但是我的研究却显示，所有这些“改进措施”都是丑陋的 hack。因为他们没有看到问题的根源，所以他们的方案只对一些特殊情况起作用，而不能通用。为此，我可以轻而易举的写出正确的程序，而让它不能通过这些类型系统的检查，比如像我这篇英文博文所示。如果你看到了问题的根源，就会发现 HM 系统的这个错误是无法弥补的，因为它触及了 HM 系统的根基。为了根治这个问题，let-polymorphism 必须被去除掉。

我为此提出了自己的解决方案：在 lambda 的位置进行“generalization”，也就是说把 \forall 放在 lambda 的位置，而不是 let。这样一来 let-polymorphism 就不存在了。但是这样一来，HM 系统就不再是 HM 系统，因为它的“模块化类型推导”的性质，就会名存实亡。由于类型里面含有程序的“控制结构”，这个类型系统表面上看起来是在进行“模块化类型检查”，而本质上是在做一个“跨过程静态检查”（interprocedural static analysis）。也就是说，模块化的类型推导，在 HM 这样的没有“类型标记”的体系下，其实是不可能实现的。

为了达到完全通用的模块化类型检查，却又允许多态函数的存在，我们终究会需要在函数的参数位置手工写上类型，这样我们就完全的丧失了 HM 系统设计的初衷。

函数式语言的宗教

很早的时候，“函数式语言”对于我来说就是 Lisp，因为 Lisp 可以在程序的几乎任意位置定义函数，并且把它们作为值来传递（这叫做 first-class function）。可是到后来有人告诉我，Lisp 其实不算“函数式语言”，因为 Lisp 的函数不“纯”（pure）。

所谓“纯函数”，就是像数学函数一样，你给它同样的输入，它就给你同样的输出。然后你就发现在这种定义下，几乎所有程序语言里面的随机数函数（random），都不是“纯函数”。因为每一次调用 random()，输入都是一样的（没有参数），但每次会输出不同的随机数。他们告诉我，不纯的函数容易出错，没法验证它的正确性。

在这种害怕自己用的语言“不纯”，不安全的恐慌之下，我开始接触 Haskell，一种号称“纯函数式”，安全的语言。我深信 Haskell 的纯函数教条长达几年之久，对其它语言里的“副作用”（side-effect）嗤之以鼻。我认为宇宙的本质是纯的，数学就是宇宙的终极语言，所以程序语言也应该像数学一样纯粹……

你有没有意识到，所有的邪教头子最初都是利用了人们的恐惧心理，进而让他们深信不疑，以为遇到了救世主的？当我多次碰壁，猛然醒悟的时候，我发现 Haskell 就是这样一种邪教 :p

每一种宗教都有一个神秘的，体现自己“精神”的徽标。Haskell 的是这个样子：



当一个初学者进入 Haskell 的 IRC 聊天室的时候，老手们会极其耐心地问答他的问题。他们告诉他，Haskell 社区是最友好，最不宗教，最讲科学和理性的社区。可是久而久之，你发现其实不是那个样子。他们对你友好，当且仅当你没有指出 Haskell 的致命缺陷。如果你知道了那些缺陷，要跟他们讨论的时候，就会发现一切都变了。他们会说，你懂不起！我们是科学家！如果你不承认这一点，我们就杀了你！:p

Haskell 的社区喜欢在他们的概念里省掉“纯”这个字，把 Haskell 叫做“函数式语言”。他们喜欢“纠正”别人的概念。他们告诉人们，“不纯”的函数式语言，其实都不配叫做“函数式语言”。在他们的这种定义下，Lisp 这么老牌的函数式语言，居然都不能叫“函数式语言”了。但是看完这篇文章你就会发现，其实他们的这种定义是狭隘和错误的。

在 Haskell 里面，你不能使用通常语言里面都有的赋值语句，比如 Pascal 里的 `x:=1`，C 和 Java 里的 `x=1`，或者 Scheme 里的 `(set! x 1)`，Common Lisp 里的 `(setq x 1)`。这样一来，你就不可能保留“状态”（state）。所谓“状态”，就是指“随机数种子”那样的东西，其实本质上就是“全局变量”。比如，在 C 语言里定义 `random()` 函数，你可以这么做：

```
int random()
{
    static int seed = 0;
    seed = next_random(seed);
    return seed;
}
```

这里的 `seed` 是一个“static 变量”，其本质就是一个全局变量，只不过这个全局变量只能被 `random` 这一个函数访问。每次调用 `random()`，它都会使用 `next_random(seed)` 生成下一个随机数，并且把 `seed` 的值更新为这个新的随机数。在 `random()` 的执行结束之后，`seed` 会一直保存这个值。下一次调用 `random()`，它就会根据 `seed` 保存的值，算出下一个随机数，然后再次更新 `seed`，如此继续。这就是为什么每一次调用 `random()`，你都会得到不同的随机数。

可是在 Haskell 里面情况就很不一样了。由于 Haskell 不能保留状态，所以同一个“变量”在它作用域的任何位置都具有相同的值。每一个函数只要输入相同，就会输出同样的结果。所以在 Haskell 里面，你不能轻松的表达 `random` 这样的“不纯函数”。为了让 `random` 在每次调用得到不同的输出，你必须给它“不同的输入”。那怎么才能给它不同的输入呢？Haskell 采用的办法，就是把“种子”作为输入，然后返回两个值：新的随机数和新的种子，然后想办法把这个新的种子传递给下一次的 `random` 调用。所以 Haskell 的 `random` 的“线路”看起来像这个样子：

（旧种子）---> （新随机数， 新种子）

现在问题来了。得到的新种子，必须被准确无误的传递到下一个使用 `random` 的地方，否则你就没法生成下一个随机数。因为没有地方可以让你“暂存”这个种子，所以为了把种子传递到下一个使用它的地方，你经常需要让种子“穿过”一系列的函数，才能到达目的地。种子经过的“路径”上的所有函数，必须增加一个参数（旧种子），并且增加一个返回值（新种子）。这就像是用一根吸管扎穿这个函数，两头通风，这样种子就可以不受干扰的通过。

所以你看到了，为了达到“纯函数”的目标，我们需要做很多“管道工”的工作，这增加了程序的复杂性和工作量。如果我们可以把种子存放在一个全局变量里，到需要的时候才去取，那就根本不需要把它传来传去的。除 `random()` 之外的代码，都不需要知道种子的存在。

为了减轻视觉负担和维护这些进进出出的“状态”，Haskell 引入了一种叫 monad 的概念。它的本质是使用类型系统的“重载”（overloading），把这些多出来的参数和返回值，掩盖在类型里面。这就像把乱七八糟的电线塞进了接线盒似的，虽然表面上看起来清爽了一些，底下的复杂性却是不可能消除的。有时候我很纳闷，在其它语言里易如反掌的事情，为什么到 Haskell 里面就变成了“研究性问题”，很多时候就是 monad 这东西在捣鬼。特别是当你有多个“状态”的时候，你就需要使用像 monad transformer 这样的东西。而 monad transformer 在本质上其实是一个丑陋的 hack，它并不能从根本上解决问题，却可以让你伤透脑筋也写不出来。有些人以为会用 monad 和 monad transformer 就说明他水平高，其实这根本就是自己跟自己过不去而已。

当谈到 monad 的时候，我喜欢打这样一个比方：

使用含有 monad 的“纯函数式语言”，就像生活在一个没有电磁波的世界。

在这个世界里面没有收音机，没有手机，没有卫星电视，没有无线网，甚至没有光！这个世界里的所有东西都是“有线”的。你需要绞尽脑汁，把这些电线准确无误的通过特殊的“接线器”（monad）连接起来，才能让你的各种信息处理设备能够正常工作，才能让你自己能够看见东西。如果你想生活在这样的世界里的話，那就请继续使用 Haskell。

其实要达到纯函数式语言的这种“纯”的效果，你根本不需要使用像 Haskell 这样完全排斥“赋值语句”的语言。你甚至不需要使用 Lisp 这样的“非纯”函数式语言。你完全可以用 C 语言，甚至汇编语言，达到同样的效果。

我只举一个非常简单的例子，在 C 语言里面定义如下的函数。虽然函数体里面含有赋值语句，它却是一个真正意义上的“纯函数”：

```
int f(int x) {
    int y = 0;
    int z = 0;
    y = 2 * x;
    z = y + 1;
    return z / 3;
}
```

这是为什么呢？因为它计算的是数学函数 $f(x) = (2x+1)/3$ 。你给它同样的输入，肯定会得到同样的输出。函数里虽然对 y 和 z 进行了赋值，但这种赋值都是“局部”的，它们不会留下“状态”。所以这个函数虽然使用了被“纯函数程序员”们唾弃的赋值语句，却仍然完全的符合“纯函数”的定义。

如果你研究过编译器，就会理解其中的道理。因为这个函数里的 y 和 z ，不过是函数的“数据流”里的一些“中间节点”，它们的用途是用来暂存一些“中间结果”。这些局部的赋值操作，跟函数调用时的“参数传递”没有本质的区别，它们不过都是把信息传送到指定的节点而已。如果你不相信的话，我现在就可以把这些赋值语句全都改写成函数调用：

```
int f(int x) {
    return g(2 * x);
}

int g(int y) {
    return h(y + 1);
}

int h(int z) {
    return z/3;
}
```

很显然，这两个 `f` 的定义是完全等价的，然而第二个定义却没有任何赋值语句。第一个函数里对 y 和 z 的“赋值语句”，被转换成了等价的“参数传递”。这两个程序如果经过我写的编译器，会生成一模一样的机器代码。所以如果说赋值语句是错误的话，那么函数调用也应该是错误的了。那我们还要不要写程序了？

盲目的排斥赋值语句，来自于对“纯函数”这个概念的片面理解。很多研究像 Haskell，ML 一类语言的专家，其实并不明白我上面讲的道理。他们仿佛觉得如果使用了赋值，函数就肯定不“纯”了似的。CMU 的教授 Robert Harper 就是这样一个极端。他在一篇博文里指出，人们不应该把程序里的“变量”叫做“变量”，因为它跟数学和逻辑学里所谓的“变量”不是一回事，它可以被赋值。然而，其果真如他所说的那样吗？如果你理解了我对上面的例子的分析，你就会发现其实程序里的“变量”，跟数学和逻辑学里面的“变量”相比，其实并没有本质的不同。

程序里的变量甚至更加严格一些。如果你把数学看作一种程序语言的话，恐怕没有一本数学书可以编译通过。因为它们里面充满了变量名冲突，未定义变量，类型错误等程序设计的低级错误。你只需要注意概率论里表示随机数的大写变量（比如 `X`），就会发现数学所谓的“变量”其实是多么的不严谨。这变量 `X` 根本不需要被赋值，它自己身上就带“副作用”！实际上，90%以上的数学家都写不出像样的程序来。所以拿数学的“变量”来衡量程序语言的“变量”，其实是颠倒了。我们应该用程序的“变量”来衡量数学的“变量”，这样数学的语言才会有所改善。

逻辑学家虽然有他们的价值，但他们并不是先知，并不总是对的。由于沉迷于对符号的热爱，他们经常看不到事物的本质。虽然他们理解很多符号公式和推理规则，但他们却经常不明白这些符号和推理规则，到底代表着自然界中的什么物体，所以有时候他们连最基本的问题都会搞错（比如他们有时候会混淆“全称量词” \forall 的作用域）。逻辑学家们的教条主义和崇古作风，也许就是图灵当年在 Church 手下做学生那么孤立，那么痛苦的原因。也就是这个图灵，在某种程度上超越了 Church，把一部分人从逻辑学的死板思维模式下解放了出来，变成了“计算机科学家”。当然其中某些计算机科学家堕入了另外一种极端，他们对逻辑学已有的精华一无所知，所以搞出一些完全没有原则的设计，然而这不是这篇文章的主题。

所以综上所述，我们完全没有必要追求什么“纯函数式语言”，因为我们在不引起混淆的前提下使用赋值语句，而写出真正的“纯函数”来。可以自由的对变量进行赋值的语言，其实超越了通常的数理逻辑的表达能力。如果你不相信这一点，就请想一想，数理逻辑的公式有没有能力推断出明天的天气？为什么天气预报都是用程序算出来的，而不是用逻辑公式推出来的？所以我认为，程序其实在某种程度上已经成为比数理逻辑更加强大的逻辑。完全用数理逻辑的思维方式来对程序语言做出评价，其实是很片面的。

说了这么多，对于“函数式语言”这一概念的误解，应该消除得差不多了。其实“函数式语言”唯一的要求，应该是能够在任意位置定义函数，并且能够把函数作为值传递，不管这函数是“纯”的还是“不纯”的。所以像 Lisp 和 ML 这样的语言，其实完全符合“函数式语言”这一称号。

什么是“脚本语言”

很多人都会用一些“脚本语言”(scripting language)，却很少有人真正的知道到底什么是脚本语言。很多人用 shell 写一些“脚本”来完成日常的任务，用 Perl 或者 sed 来处理一些文本文件，很多公司用“脚本”来跑它们的“build”(叫做 build script)。那么，到底什么是“脚本语言”与“非脚本语言”的区别呢？

其实“脚本语言”与“非脚本语言”并没有语义上，或者执行方式上的区别。它们的区别只在于它们设计的初衷：脚本语言的设计，往往是作为一种临时的“补丁”。它的设计者并没有考虑把它作为一种“通用程序语言”，没有考虑用它构建大型的软件。这些设计者往往没有经过系统的训练，有些甚至连最基本的程序语言概念都没搞清楚。相反，“非脚本”的通用程序语言，往往由经过严格训练的专家甚至一个小组的专家设计，它们从一开头就考虑到了“通用性”，以及在大型工程中的可靠性和可扩展性。

首先我们来看看“脚本”这个概念是如何产生的。使用 Unix 系统的人都会敲入一些命令，而命令貌似都是“一次性”或者“可抛弃”的。然而不久，人们就发现这些命令其实并不是那么的“一次性”，自己其实一直在重复的敲入类似的命令，所以有人就发明了“脚本”这东西。它的设计初衷是“批量式”的执行命令，你在一个文件里把命令都写进去，然后执行这个文件。可是不久人们就发现，这些命令行其实可以用更加聪明的方法构造，比如定义一些变量，或者根据系统类型的不同执行不同的命令。于是，人们为这脚本语言加入了变量，条件语句，数组，等等构造。“脚本语言”就这样产生了。

然而人们却没有发现，其实他们根本就不需要脚本语言。因为脚本语言里面的这些结构，在任何一种“严肃”的程序语言(比如 Java, Scheme)里面，早就已经存在了，而且设计得更加完善。所以脚本语言往往是在重新发明轮子，甚至连轮子都设计不好。早期脚本语言的“优势”，也许只在于它不需要事先“编译”，它“调用程序”的时候，貌似可以少打几个字。脚本语言对于 C 这样的语言，也许有一定的价值。然而，如果跟 Scheme 或者 Java 这样的语言来比，这个优势就非常不明显了。比如，你完全可以想一个自动的办法，写了 Java 代码之后，先调用 Java 编译器，然后调用 JVM，最后删掉 class 文件。或者你可以选择一种有解释执行方式的“严肃语言”，比如 Scheme。

很多人把 Scheme 误称为“脚本语言”，就是因为它像脚本语言一样可以解释执行，然而 Scheme 其实是比 C 和 Java 还要“严肃”的语言。Scheme 从一开始就被设计为一种“通用程序语言”，而不是用来进行某种单一简单的任务。Scheme 的设计者比 Java 的设计者造诣更加深厚，所以他们对 Java 的一些设计错误看得非常清楚。像 Chez Scheme 这样的编译器，其实早就可以把 Scheme 编译成高效的机器代码。实际上，很多 Scheme 解释器也会进行一定程度的“编译”，有些编译为字节码，有些编译为机器代码，然后再执行。所以在这种情况下，通常人们所谓的“编译性语言”与“解释性语言”，几乎没有本质上的区别，因为你看到的“解释器”，不过是自动的先编译再执行。

跟 Java 或者 Scheme 这样的语言截然不同，“脚本语言”往往意味着异常拙劣的设计，它的设计初衷往往是目光短浅的。这些语言里面充满了历史遗留下来的各种临时的 hack，几乎没有“原则”可言。Unix 的 shell(比如 bash, csh, ...)，一般都是这样的语言。Java 的设计也有很多问题，但也跟“脚本语言”有天壤之别。然而，在当今现实的工程项目中，脚本语言却占据了它们不该占有的地位。例如很多公司使用 shell 脚本来处理整个软件的“build”过程或者测试过程，其实是相当错误的决定。因为一旦这种 shell 脚本日益扩展，就变得非常难以控制。经常出现一些莫名其妙的问题，却很难找到问题的所在。Linux 使用 shell 脚本来管理很多启动项目，系统配置等等，其实也是一个历史遗留错误。所以，不要因为看到 Linux 用那么多 shell 脚本就认为 shell 语言是什么好东西。

如果你在 shell 脚本里使用通常的程序设计技巧，比如函数等，那么写几百行的脚本还不至于到达不可收拾的地步。可是我发现，很多人头脑里清晰的程序设计原则，一遇到“写脚本”这样的任务就完全崩溃了似的，他们仿佛认为写脚本就是应该“松散”一些。很多平时写非常聪明的程序的人，到了需要处理“系统管理”任务的时候，就开始写一些 shell 脚本，或者 Perl 脚本。他们写这些脚本的时候，往往完全的忘记了程序设计的基本原则，例如“模块化”，“抽象”等等。他们大量的使用“环境变量”一类的东西来传递信息，他们忘记了使用函数，他们到处打一些临时性的补丁，只求当时不出问题就好。到后来，他们开始耗费大量的时间来处理脚本带来的麻烦，却始终没有发现问题的罪魁祸首，其实是他们错误的认为自己需要“脚本语言”，然后认为写脚本的时候就是应该随便一点。

所以我认为脚本语言是一个祸害，它几乎永远是错误的决定。我们应该尽一切可能避免使用脚本语言。在没有办法的情况下(比如老板要求)，也应该在脚本里面尽可能的使用通常的程序设计原则。

Chez Scheme 的传说

在上一篇博文的最后，我提到了 Lisp 编译器的问题。由于早期的 Lisp 编译器生成的代码效率普遍低下，成为了 Lisp 失败的主要原因之一。而现在的高性能 Lisp 编译器（比如 Chez Scheme），其实已经可以生成非常高效的代码，甚至可以匹敌 C 程序的速度。如果你看得到我脑子里的东西，就会明白这完全不是吹牛，而是科学的结论。我在这里介绍一下我写 Scheme 编译器的经历，也许你就会从根本上明白为什么我会对此这么自信。这里的介绍其实不止针对函数式语言，而且针对所有语言的编译器。

编译器是一种神秘，有趣，又无聊的程序。说它神秘，是因为只有非常少的人知道如何写出优秀的编译器。这些会写编译器的人，就像身怀绝技的武林高手一样神出鬼没。说它有趣，是因为编译器的技术里面含有大量的“哲学问题”和深刻的理论（比如 partial evaluation）。但为什么又说它无聊呢？因为你一旦掌握了编译器技术里面最精华的原理，就会发现其实说来说去就那么点东西。编译器代码里面的“创造性含量”其实非常低。里面有些固定的“模式”，几十年都不变。这是因为编译器只是一种“工具”，而不是最终的“目的”。它就像做菜的锅一样，只有屈指可数的那几种形状。设计应用程序才是程序员的最终目的。只有应用程序才能有无穷无尽的创造性。这就像厨师用同样的锅，却能做出无穷变化的菜肴来。然而，我并不是说普通程序员不应该学习写编译器。相反，编译器的原理是非常重要的知识。不理解编译原理的应用程序设计者，就像不理解菜锅组成原理的厨师。

先来说一说为什么早期的 Lisp 编译器生成的代码效率低下吧。在函数式语言的早期，由于它比普通的语言多了一些表达力强大的构造（比如函数作为值传递），人们其实都不知道如何实现它的编译器。很多 Scheme 的编译器其实只是把 Scheme 编译成 C，然后再调用 C 语言的编译器。Haskell 的编译器 GHC 在早期也是这样的。而且由于 C 编译器生成的汇编代码不完全符合 Haskell 的需求，GHC 里面含有一个 Perl 脚本，专门用于调整这汇编代码的结构。这个 Perl 脚本，由于它的工作方式毫无原则，被叫做 evil mangler。现在这个东西已经被去掉了，但从它曾经的存在你可以看出，其实函数式编译器的技术在早期是相当混沌的。

在我看来，早期 Lisp 编译器出现的主要问题，其实在于对编译的本质的理解，以及编译器与解释器的根本区别。解释器之所以大部分时候比编译器慢，是因为解释器“问太多的问题”。每当看到一个构造，解释器就会问：“这是一个整数吗？”“这是一个字符串吗？”“这是一个函数吗？”……然后根据问题的结果进行不同的处理。这些问题，在编译器的理论里面叫做“解释开销”（interpretive overhead）。编译的本质，其实就是在程序运行之前进行“静态分析”，试图一劳永逸的回答这些问题。于是编译后的代码根本不问这种问题，它直接就知道那个位置肯定会出现什么构造，应该做什么事，于是它就直接去做了。早期的 Lisp 编译器，以及现在的很多 Scheme 编译器出现的问题其实在于，它们并没有干净的消除这些问题，甚至根本没有消除这些问题。

当我最早学习 Scheme 语言的时候，我发现 Scheme 有太多的“实现”：PLT Scheme（现在叫 Racket），MIT Scheme，Scheme 48，Bigloo，Chicken，Gambit，Guile，… 让人搞不清楚哪一个更好。有些 Scheme 实现显得高级一些，但实际用起来总是感觉不放心，因为你心里总想着，这代码编译出来到底能不能跟 C 语言代码比？这也是我后来开始使用 Common Lisp 的原因，因为 Common Lisp 似乎有挺多高效的编译器（CMUCL，Lispworks，Allegro 等等）。

直到有一天，我发现了 Chez Scheme，它改变了我对 Scheme 编译器，以至于整个编译器概念的理解。当时我只下载了 Chez Scheme 的免费版本，叫做 Petite。Petite 与正式版 Chez Scheme 的区别是，它不输出二进制代码，所以你不能把编译后的代码拿去销售。另外出于商业目的，Petite 的出错信息非常的“简约”，以至于有时候你不得不用其它的 Scheme 实现，才能找到 bug 的位置。但是一运行就见分晓，Petite 被作为一个“解释器”直接运行 Scheme 代码，比其他的 Scheme 实现编译后的代码还要快很多倍。

Chez Scheme 导致了我命运的改变，我怎么也没有想到，自己最终会见到它的作者 R. Kent Dybvig，并且成为他的学生。我只能说也许一切都是天意吧。第一次见到 Kent 的时候，他安静的对我说，你应该拥有自己的代码，将来有一天，你会发现它的价值。

也就是这个 Kent，单枪匹马的创造了 Chez Scheme，世界上唯一的商业 Scheme 编译器，并且为此成立了自己的公司（Cadence Research Systems）。Chez Scheme 价格不菲，而且不明码实价，它的价格跟项目的大小和公司的规模成正比。有些大公司花重金购买 Chez Scheme 用于一些核心的项目。其中有些公司为了保证这编译器的安全，又花了好几倍的价钱买下了它的源代码。Kent 的公司只有他一个人，不用操心管理，也不用操心销售。所以他过的非常舒服，基本是一个不愁吃穿，不问世事的人。

Kent 是我一生中见过的最神秘，最酷的人。他几乎从来不表扬任何人，但也不贬低任何人。从冷漠的言语之中，你仿佛感觉他并不是这个世界上的人。任何人的喜怒与哀乐，傲慢与偏见，蔑视与奉承，全都不能引起他情绪的变化。他的心里有许许多多的秘密，你需要一些技巧才能套出他的真言。他很少发表论文，却把别人的论文全都看得很透。没有人知道他的核心技术，他也从来不在乎别人是否了解他的水平。最让人惊奇的是，没有人知道他叫什么名字！他的全名叫 R. Kent Dybvig，那么 R. 就应该是他的 first name。然而，却从来没有知道那个 R. 是哪一个名字的简写，所以大家只好叫他的 middle name，Kent。

Chez Scheme 生成的“目标代码”效率之高，我还没有见到任何其它 Scheme 编译器可以与之匹敌。而它的“编译速度”之快，没有任何语言的任何编译器可以相提并论（注意我去掉了“Scheme”这个限定词）。Chez Scheme 可以在 5 秒钟之内完成从头到尾的自我编译。想想编译 GCC 或者 GHC 需要多少时间，你就明白差距了。

另外值得一提的是，Chez Scheme 从头到尾都是 Kent 一个人的作品。它的工作原理是从 Scheme 源程序一直编

译到机器代码，而不依赖任何其他语言的编译器。它甚至不依赖第三方的汇编器，所有三种体系构架（Intel, ARM, SPARC）的汇编器，都是 Kent 自己写的。为什么这样做呢？因为几乎没有其它人的编译器代码能够达到他的标准。连 Intel 自己给自己的处理器写的汇编器，都不能满足他的要求。

如果你上了 Kent 的课，再来看看普通的编译器书籍（比如有名的 Dragon Book），或者 LLVM 的代码，你就会发现 Kent 的水平其实远在这些知名的大牛之上。我为什么可以这么说呢？因为如果你的水平不如这些人的话，你自己都会对这种判断产生怀疑。而如果你超过了别人，他们的一言一行，他们的每一个错误，都像是处于你的显微镜底下，看得一清二楚。这就是为什么有一天我拿起 Dragon Book，感觉它变得那么的幼稚。而其实并不是它变幼稚了，而是我变成熟了。实话实说吧，在编译器这个领域，我觉得 Kent 很有可能就是世界的 No.1。

如果你不了解 Scheme 的编译器里面有什么东西，也许就会轻视它的难度。Scheme 是比 C 语言高级很多的语言，所以它的编译器需要做比 C 语言的编译器多很多事情。在 Kent 的编译器课程的前半段，我们其实本质上是在实现一个 C 语言的编译器，把一种基于“S表达式”的中间语言，编译为 X64 汇编代码。在后半学期的课程中，我们才加入了各种 Scheme 的先进功能，比如函数作为值（需要进行 closure conversion 以及 closure 优化），尾递归优化（tail-call optimization），等等。另外，我还自己为它加入了一种非常漂亮的技术，叫做 online partial evaluation。这种技术可以在一个 pass 就完成普通编译器需要好几个 pass 才能完成的优化。

在这些先进的优化技术之下，几乎所有的冗余代码都会被编译器消除掉。这些优化的智能程度，在很多方面拥有人类思维没法达到的准确性和深度。如果你的程序没有使用到 Scheme 特有的功能，那么生成的目标代码就会跟 C 语言编译后的代码没有什么两样。比如，如果你的代码没有把函数作为值传递，或者你的函数里面没有“自由变量”，或者你的函数里虽然有自由变量，但是你却没法在函数外部改变它的值，那么生成的代码里面就不会含有“闭包”，也就不会产生多余的内存数据交换。你有时甚至会得到比 C 程序编译之后更好的代码，因为我们的“后端”编译器其实比 GCC, LLVM 之类的 C 编译器先进。

Kent 的课程编译器有很好的结构，它被叫做“nanopass 编译器构架”。它的每一个 pass 只做很小的一件事情，然后这些 pass 被串联起来，形成一个完整的编译器。编译的过程，就是将输入程序经过一系列的变换之后，转化为机器代码。你也许发现了，这在本质上跟 LLVM 的构架是一样的。但是我可以告诉你，我们的课程编译器比 LLVM 干净利落许多，处于远远领先地位。每一节课，我们都学会一个 pass。每一个讲义，都非常精确的告诉你需要干什么。每一次的作业，提交的时候都会经过上百个测试（当然 Kent 不可能把 Chez Scheme 的测试都给我们），如果没有通过就会被拒绝接受。这些测试也可以下载，用于自己的调试。有趣的是，每一次作业都需要提交一些自己写的新测试，目的是用于“破坏”别人的编译器。所以我们每次都会想出很刁钻的输入代码，让同学的日子不好过。当然是开玩笑的，这种做法其实大大的提高了我们对编译器测试的理解和兴趣，以及同学之间的友谊。这比起我曾经在 Cornell 选过（然后 drop 掉）的编译器课程，真是天壤之别。

在课程的最后，我们做出了一个完整的编译器，它可以把 Scheme 最关键的子集编译到 X64 汇编代码，然后通过 GNU 的汇编器转化成机器代码。在最后一节课，Kent 对我们的学期做了一个令人难忘的总结。他说：“你们现在写出的这个编译器里面含有很多先进的技术。也许过一段时间再回头看这段代码，你们才会发现它的价值。如果你们觉得自己已经成为了编译器的专家，那我就告诉你们，你们提交的最快的编译器，编译速度比 Chez Scheme 慢了 700 倍。但是不要灰心，我告诉你们哪些地方可以改进……”

只有极少数的人见到过 Chez Scheme 的源代码，我也没有看见过。但是见到过它的人告诉我，Chez Scheme 里面其实只有很少几个 pass，而不是像我们的课程编译器有 50 个左右的 pass，这节省了很多用于“遍历”代码树所需要的时间。Chez Scheme 只使用了一些非常简单的算法，没有使用论文里很炫很复杂的方法，这也是它速度快的原因之一。比如它的寄存器分配，没有使用通常的“图着色”（graph coloring）方法，而是使用非常简单的一种类似 linear scan 的算法，生成的代码效率却更高。另外，Scheme 使用“S表达式”作为它的语法，使得“语法分析”的速度非常之快。其它语言由于使用了复杂的语法，挺大一部分编译时间其实花在了语法分析上面。

所以实际上 Chez Scheme 早就有了超越世人的技术，Kent 却很少为它们发表论文。这是因为他自私吗？应该不是。他已经通过他的课程给予了我们那么宝贵的礼物，我们又怎能要求更多？所以对于更深入的内容，我都是自己摸索出解决方案，再去套他的口气，看他有没有更好的想法。于是有时候我会很惊讶的发现他的一些非常透彻的见解。比如有一天我问他，为什么编译器需要进行寄存器分配？为什么需要寄存器？我觉得 Knuth 设计的 MMIX 处理器里的“寄存器环”，也许能够从根本上避免“寄存器分配”这问题。他听了之后不动声色的说，MMIX 的寄存器环（以及 SPARC 的寄存器窗口）其实是有问题的，当函数递归调用达到一定的深度之后，寄存器环里有再多寄存器都会被用光，到时候就会出现大量的寄存器与内存之间的数据交换，而被“压栈”之后的寄存器，并不会得到有效地“再利用”。于是我发现，他不但早已了解 MMIX 的设计，而且看透了它的本质。

有趣的是在课程进行之中的时候，我发现有些突发灵感的做法，其实已经超越了 Chez Scheme，以至于在某些 pass 会生成比它还要高效的代码，然而我的编译器代码却比它的还要短小（当然绝大部分时间我的代码不如 Chez Scheme）。于是我就隐约的发现，Kent 有时候会悄悄的花时间看我的作业，想搞明白我是怎么做的，但却不想让我知道。有一天开会的时候 Kent 没有来，他的编译器课程助教 Andy 对我说：“Kent 还在对你写的代码进行一些侦探工作……”从任何人那里得到启发，吸收并且融入到自己的能力里面，也许就是 Kent 练就如此盖世神功的秘诀吧。

我想，这篇文章就该到此结束了。写这些东西的目的，其实只是树立人们对于函数式语言编译器的信心。它们有些其实比 C 和 C++ 之类语言的编译器高明很多。我没有时间也没有精力去讲述这编译器里面的细节，因为它实在是非常困难，却又非常优雅的程序。如果你有兴趣的话，可以看看我最后的[代码](#)。由于版权原因，有些辅助部件我不能放在网上，所以你并不能运行它，只能看一个大概的形状。如果你需要一个 Scheme 版本用于学习的话，Chez Scheme 有一个免费的版本叫做 Petite Chez Scheme，可以免费下载。因为 Petite 的出错信息非常不友好，所以我也推荐 Racket 作为替补。不过你需要注意的是，Racket 的速度比起 Chez Scheme 是天壤之别。

Lisp 已死，Lisp 万岁！

有一句古话，叫做“国王已死，国王万岁！”它的意思是，老国王已经死去，国王的儿子现在继位。这句话的幽默，就在于这两个“国王”其实指的不是同一个人，而你乍一看还以为它自相矛盾。今天我的话题仿效了这句话，叫做“Lisp 已死，Lisp 万岁！”希望到最后你会明白这是什么意思。

首先，我想总结一下 Lisp 的优点。你也许已经知道，Lisp 身上最重要的一些优点，其实已经“遗传”到了几乎每种流行的语言身上 (Java, C#, JavaScript, Python, Ruby, Haskell,)。由于我已经在其他博文里详细的叙述过其中一些，所以现在只把这些 Lisp 的优点简单列出来 (关键部分加了链接)：

- Lisp 的语法是世界上最精炼，最美观，也是语法分析起来最高效的语法。这是 Lisp 独一无二的，其他语言都没有的优点。有些人喜欢设计看起来很炫的语法，其实都是自找麻烦。为什么这么说呢，请参考这篇《[谈语法](#)》。
- Lisp 是第一个可以在程序的任何位置定义函数，并且可以把函数作为值传递的语言。这样的设计使得它的表达能力非常强大。这种理念被 Python, JavaScript, Ruby 等语言所借鉴。
- Lisp 有世界上最强大的宏系统 (macro system)。这种宏系统的表达力几乎达到了理论所允许的极限。如果你只见过 C 语言的“宏”，那我可以告诉你它是完全没法跟 Lisp 的宏系统相提并论的。
- Lisp 是世界上第一个使用垃圾回收 (garbage collection) 的语言。这种超前的理念，后来被 Java, C# 等语言借鉴。

想不到吧，现代语言的很多优点，其实都是来自于 Lisp — 世界上第二古老的程序语言。所以有人才会说，每一种现代语言都在朝着 Lisp 的方向“进化”。如果你相信了这话，也许就会疑惑，为什么 Lisp 今天没有成为主流，为什么 Lisp Machine 会被 Unix 打败。其实除了商业原因之外，还有技术上的问题。

早期的 Lisp 其实普遍存在一个非常严重的问题：它使用 dynamic scoping。所谓 dynamic scoping 就是说，如果你的函数定义里面有“自由变量”，那么这个自由变量的值，会随着函数的“调用位置”的不同而发生变化。

比如下面我定义一个函数 f，它接受一个参数 y，然后返回 x 和 y 的积。

```
(setq f
      (let ((x 1))
        (lambda (y) (* x y))))
```

这里 x 对于函数 (lambda (y) (* x y)) 来说是个“自由变量” (free variable)，因为它不是它的参数。

看着这段代码，你会很自然的认为，因为 x 的值是 1，那么 f 被调用的时候，结果应该等于 (* 1 y)，也就是说应该等于 y 的值。可是这在 dynamic scoping 的语言里结果如何呢？我们来看看吧。

(你可以在 emacs 里面试验以下的结果，因为 Emacs Lisp 使用的就是 dynamic scoping。)

如果我们在函数调用的外层定义一个 x，值为 2：

```
(let ((x 2))
  (funcall f 2))
```

因为这个 x 跟 f 定义处的 x 的作用域不同，所以它们不应该互相干扰。所以我们应该得到 2。可是，这段代码返回的结果却为 4。

再来。我们另外定义一个 x，值为 3：

```
(let ((x 3))
  (funcall f 2))
```

我们的期望值还是 2，可是结果却是 6。

再来。如果我们直接调用：

```
(funcall f 2)
```

你想这次总该得到 2 了吧？结果，出错了：

```
Debugger entered--Lisp error: (void-variable x)
(* x y)
(lambda (y) (* x y))(2)
funcall((lambda (y) (* x y)) 2)
eval_r((funcall f 2) nil)
eval-last-sexp-1(nil)
```

```
eval-last-sexp(nil)
call-interactively(eval-last-sexp nil nil)
```

看到问题了吗？`f` 的行为，随着调用位置的一个“名叫 `x`”的变量的值而发生变化。而这个 `x`，跟 `f` 定义处的 `x` 其实根本就不是同一个变量，它们只不过名字相同而已。这会导致非常难以发现的错误，也就是早期的 Lisp 最令人头痛的地方。好在现在的大部分语言其实已经吸取了这个教训，所以你不再会遇到这种让人发疯的痛苦。不管是 Scheme, Common Lisp, Haskell, OCaml, Python, JavaScript…… 都不使用 dynamic scoping。

那现在也许你了解了，什么是让人深恶痛绝的 dynamic scoping。如果我告诉你，Lisp Machine 所使用的语言 Lisp Machine Lisp 使用的也是 dynamic scoping，你也许就明白了为什么 Lisp Machine 会失败。因为它跟现在的 Common Lisp 和 Scheme，真的是天壤之别。我宁愿写 C++，Java 或者 Python，也不愿意写 Lisp Machine Lisp 或者 Emacs Lisp。

话说回来，为什么早期的 Lisp 会使用 dynamic scoping 呢？其实这根本就不是一个有意的“设计”，而是一个无意的“巧合”。你几乎什么都不用做，它就成那个样子了。这不是开玩笑，如果你在 emacs 里面显示 `f` 的值，它会打印出：

```
'(lambda (y) (* x y))
```

这说明 `f` 的值其实是一个 S 表达式，而不是像 Scheme 一样的“闭包”（closure）。原来，Emacs Lisp 直接把函数定义处的 S 表达式 `'(lambda (y) (* x y))` 作为了函数的“值”，这是一种很幼稚的做法。如果你是第一次实现函数式语言的新手，很有可能就会这样做。Lisp 的设计者当年也是这样的情况。

简单倒是简单，麻烦事接着就来了。调用 `f` 的时候，比如 `(funcall f 2)`，`y` 的值当然来自参数 2，可是 `x` 的值是多少呢？答案是：不知道！不知道怎么办？到“外层环境”去找呗，看到哪个就用哪个，看不到就报错。所以你就看到了之前出现的现象，函数的行为随着一个完全无关的变量而变化。如果你单独调用 `(funcall f 2)` 就会因为找不到 `x` 的值而出错。

那么正确的实现函数的做法是什么呢？是制造“闭包”（closure）。这也就是 Scheme, Common Lisp 以及 Python, C# 的做法。在函数定义被解释或者编译的时候，当时的自由变量（比如 `x`）的值，会跟函数的代码绑在一起，被放进一种叫做“闭包”的结构里。比如上面的函数，就可以表示成这个样子：`(Closure '(lambda (y) (* x y)) '((x . 1)))`。

在这里我用 `(Closure ...)` 表示一个“结构”（就像 C 语言的 struct）。它的第一个部分，是这个函数的定义。第二个部分是 `'((x . 1))`，它是一个“环境”，其实就是一个从变量到值的映射（map）。利用这个映射，我们记住函数定义处的那个 `x` 的值，而不是在调用的时候才去瞎找。

我不想在这里深入细节。如果你对实现语言感兴趣的话，可以参考我的另一篇博文《怎样写一个解释器》。它教你如何实现一个正确的，没有以上毛病的解释器。

与 dynamic scoping 相对的就是“lexical scoping”。我刚才告诉你的闭包，就是 lexical scoping 的实现方法。第一个实现 lexical scoping 的语言，其实不是 Lisp 家族的，而是 Algol 60。“Algol”之所以叫这个名字，是因为它的设计初衷是用来实现算法（algorithm）。其实 Algol 比起 Lisp 有很多不足，但在 lexical scoping 这一点上它却做对了。Scheme 从 Algol 60 身上学到了 lexical scoping，成为了第一个使用 lexical scoping 的“Lisp 方言”。9 年之后，Lisp 家族的“集大成者”Common Lisp 诞生了，它也采用了 lexical scoping。看来英雄所见略同。

你也许发现了，Lisp 其实不是一种语言，而是很多种语言。这些被人叫做“Lisp 家族”的语言，其实共同点只是它们的“语法”：它们都是基于 S 表达式。如果你因此对它们同样赞美的话，那么你赞美的其实只是 S 表达式，而不是这些语言本身。因为一个语言的本质应该是由它的语义决定的，而跟语法没有很大关系。你甚至可以给同一种语言设计多种不同的语法，而不改变这语言的本质。比如，我曾经给 TeX 设计了 Lisp 的语法，我把它叫做 SchTeX (Scheme + TeX)。SchTeX 的文件看起来是这个样子：

```
(documentclass article (11pt))
(document
  (abstract (...))
  (section (First Section)
    ...
  )
  (section (Second Section)
    ...
  )
)
```

很明显，虽然这看起来像是 Scheme，本质却仍然是 TeX。

所以，因为 Scheme 的语法使用 S 表达式，就把 Scheme 叫做 Lisp 的“方言”，其实是不大准确的做法。Scheme 和 Emacs Lisp, Common Lisp 其实是三种不同的语言。Racket 曾经叫做 PLT Scheme，但是它跟 Scheme 的区别日益增加，以至于现在 PLT 把它改名叫 Racket。这是有他们的道理的。

所以，你也许明白了为什么这篇文章的标题叫做“Lisp 已死，Lisp 万岁！”因为这句话里面的两个“Lisp”其实是完全不同的语言。“Lisp 已死”，其实是说 Lisp Machine Lisp 这样的 Lisp，由于严重的设计问题，已经死去。而“Lisp 万岁”，是说像 Scheme, Common Lisp 这样的 Lisp，还会继续存在。它们先进于其它语言的地方，也会更多的被

借鉴，被发扬广大。

(其实老 Lisp 的死去还有另外一个重要原因，那就是因为早期的 Lisp 编译器生成的代码效率非常低下。这个问题我留到下一篇博文再讲。)

论对东西的崇拜

在之前的几篇博文里面，我多次提到了 Lisp，它相对于其它语言的优势，以及 Lisp Machine 相对于 Unix 的优点。于是有人来信请教我如何学习 Lisp，也有人问我为什么 Lisp Machine 没有“流行”起来。我感觉到了他们言语中对 Lisp 的敬畏和好奇心，但也感觉到了一些隐含的怀疑。

这是一种复杂的感觉，仿佛我在原始人的部落兜售一些原子能小玩具，却被人当成了来自天外的传教士。敬畏和奉承，并不能引起我的好感。怀疑和嘲讽，也不能引起我的不平。当我看到有人说“别听他误导群众，学那些语言是找不到工作的”的时候，我心里完全没有愤怒，也没有鄙视，我也没必要说服他。我只是微笑着摇摇头，对自己说：可怜而可笑的人。

不明白为什么，当我提到某个东西相对于另一个东西的优点的时候，我总是被人认为是在“推崇”某个东西，或者被称为是它的“狂热分子”。现在显然已经有人认为我在推崇 Lisp 了，甚至在某个地方看到有人称我为“国内三大 Lisp 狂人之一”。他们仿佛觉得我推荐一个东西，就是想让他们完全的拥抱这个东西，而丢弃自己已经有的东西。而“支持”这另一个东西的人，也往往会产生敌视情绪。

很多人都不明白，每个东西都有它好的方面，也有它不好的方面。我推荐的只是 Lisp 好的方面，不好的方面我心里清楚，但是还没有机会讲。这些人显然已经在下意识里把“东西”当成了人。有人说“爱一个人就要爱她（他）的全部”，这是一种很无奈的说法，因为你没有能力把一个人分解成你喜欢的和不喜欢的两部分，然后重新组装成你的梦中情人。可是东西却不一样。因为东西是人造出来的，所以你可以把它们大卸八块，然后挑出你喜欢的部分。

所以我可以很清楚的告诉你，我并不推崇 Lisp，我不是 Lisp 狂人，它只是我的小玩意儿之一。这个非常精巧的小玩意儿，包含了很多其它东西身上没有的优点。人们都说忘记历史就等于毁灭未来。如果 Java 没有从 Lisp 身上学会“垃圾回收”，C# 没有从 Lisp 身上学会 lambda，那么我们今天也许还在为 segfault 而烦恼，也许会继续使用没必要的 design patterns。如果你了解一点历史就会发现，今天非常流行的 JavaScript，其实不过是一个“没能正确实现的 Scheme”。所以 Lisp 的精髓，其实正在越来越多的渗透到常用的语言里面。

很多人没有设计程序语言的能力，所以他们把程序语言，操作系统一类的东西当成是不可改变的，凌驾于自己之上的。相比之下，我受到的训练却给了我设计和实现几乎任何语言的能力。我知道它们的优点和弱点，我有能力把它们大卸八十块，再组装还原。我有能力改变其中我不喜欢的地方，或者增加我觉得必要的功能。当我谈论某个东西比另一个好的地方，总有人以为我在“抱怨”，说：“既然如此，那为什么你说的这个好东西被打败了？”他不明白，其实我只是在“分析”。我希望从各个东西里面提取出好的部分，然后想办法把它们都注入到一个新的东西里面。我也希望吸取前人教训，免得重犯这些东西里面的设计错误。

所以，我其实并不是那么热心的希望有更多的人用 Lisp，Haskell 或者其它什么语言。我不会，也没那工夫去分享自己的秘诀。我没有责任，也没有能力去拯救世界。这是一种找到巨大宝藏的感觉，我蹲在一堆堆的财宝上休养生息。我知道世界上即使没有了我，太阳明天照样会升起。我为什么要那么热心的让别人也知道如何进入这个宝藏？我不是一个特别自私的人，但我也不需要推销什么。这就像我介绍了我的“减肥成功经验”，你觉得太辛苦，偏要去买那些吹得神乎其神的减肥药。我有什么动机来说服你呢？又不是我身上的肥肉。

推崇一个东西，为一个东西狂热，这些感情都在我身上存在过。也许它们确实给我带来了一些益处，让我很快的学会了一些东西。但是这些感情的存在，其实也显示了一个人的弱小。当一个人没有办法控制一个东西的时候，他就会对它产生“崇拜”的心理，这就像所有的宗教和迷信一样。当人们处于自然灾害的凌威之下，没有能力掌握自己命运的时候，他们就对神和超自然的力量产生了崇拜。这是一种心灵的慰藉，至少有上帝或者观音菩萨，可以聆听他们的心声，可以给予他们度过灾难的勇气，但它同时也显示出人的无助和自卑。这种无助和自卑，也引发了偏激的宗教心理，因为他们害怕自己的“保护神”被别人的“保护神”所压倒，以至于让自己受制于他人。这是一种愚昧和卑劣的感情。

可是当你拥有了强大的力量，可以不再畏惧的时候，这种崇拜，以及由于崇拜所带来的偏激心理，就渐渐的消亡了。这就像一个身怀绝世武功的人，他完全没必要让别人都相信他是高手。因为他知道，自己在谈笑之间，就可以让樯橹灰飞烟灭。于是，他自得其乐，对别人表现出的任何感情，都变得淡漠和无动于衷。

“解决问题”与“消灭问题”

一直以来，人们都重视“解决问题”的能力，却忽视了另一种重要的能力：“消灭问题”的能力。各种各样的竞赛，分数和排名，让很多人从小就片面的认为，能“解决问题”的人，就是最厉害的人。拿到一个问题就埋头求解，很少考虑这问题到底有什么意义。这种呆板的思维方式，不仅存在于低级的“应试”和“解题”过程，而且蔓延到了很多艰深的研究领域。

如果你仔细观察就会发现，很多“难题”，其实是“人造”出来的，而不是“必然”的。它们的存在，往往是由于一些早期的“设计错误”。人造的东西里面往往有设计上的错误，如果你把这些东西看成是不可改变的东西，那你就会遇到很多不必要的问题。打个比方，如果当初轮子被设计成方形的，而没有人质疑这样做的“必要性”，那么也许人类早就因为“能源问题”而灭绝了。有点夸张，但它却形象的说明了，为什么错误的设计会导致不必要的难题。

其实如果我们转换一下思路，或者改变一下“设计”，很多问题就可以不自消。这就是我所谓的“消灭问题”的能力。这种“消灭问题”的能力，表面上容易其实难，有点像脑经急转弯，所以经常受到人们的忽视。看到一个问题轻而易举的消失了，总有人满不在乎的说：“这个容易。我也能做到。”可问题就在于，你怎么没想到？说这种话的人，完全没有意识到，他们的思维里面其实缺少了非常重要的东西。由于喜欢炫耀自己的“头脑暴力”，他们经常解决（甚至制造）错误的问题。

所以，在解决问题之前，我们应该先问自己三个问题：

1. 这问题是否真的“存在”？

也许你已经看出来了，很多问题，即使众人都认为它存在，其实也可能是不存在的。在这一点上不能相信任何人或者机构，不管他有多么的“权威”。就像小马过河的道理，只有靠自己的实践。

2. 如果解决了这个问题，会给我和他人带来什么实际的好处？

世界上不存在“永远”，也不存在“无穷”。如果一个“科学算命家”花100年才能算出我的未来，那我还不如坐等“未来”的到来。所有的人，都不过是来这世界上做短暂的旅行。所以，问题的答案，应该能在合理的时间之内给人带来实际的好处。

3. 这问题是否可以在简单的改变某些“设计”或者“思路”之后，不复存在？

很多问题的“存在”，其实是因为人们的“思维定势”。他们看不到问题的“根源”和因果关系，而是经常在下意识里假定某种“先决条件”（A）的存在，然后坚定不移的相信由此“导致”的问题（B）的存在，如下图：

A -----> B

然后，他们开始呆头呆脑的解决 B，完全忘记了质疑 A 存在的必要性。他们从来没有想过，如何消除 A，或者切断 A 与 B 之间的关系。他们没有发现，一旦这前提 A 不复存在，问题 B 就可以不自消。

对这一点，我想起一个有趣的故事。有人在饭桌上给大家出了一道“难题”，要他们把自己盘子里的鸡蛋立起来，最后只有一个人做到了。这个人把蛋壳打破了。所有其他人都没有想到这个做法，却说他“犯规”。可是应该检讨的其实应该是他们自己，因为出题的人根本没有说不能打破蛋壳，他们却对此做出了错误的假设。

我经常发现计算机科学界存在这样的问题。研究了几十年，结果到最后才发现，辛辛苦苦解决的问题，其实包含了错误的假设。如果换一个角度来看，或者稍微改一改设计，这问题就基本不存在了。其中一个例子，就是编译器里面的“语法分析”（parsing）问题。

语法分析成为一个问题的原因，就在于很多人错误的以为程序语言应该有复杂的语法。正是这些复杂的语法，造成了这个问题研究了很多年，仍然没有一个很好的解决方案。可是一旦语法设计被简化（比如像 Lisp 那样），语法分析就变成一个非常容易的问题。实际上计算机系统（比如 Unix）里的很多问题都是由此引发的，想要利用字符串来进行数据交换，却又设计了一些非常不方便的“数据格式”。简单的语法设计，会让这些问题一并消失掉。关于这个问题，我不想重复发文，细节请见另一篇博文《[谈语法](#)》。

爱因斯坦说“想象力比知识更重要”，也许就是这个道理。没有想象力的人经常钻牛角尖，走死胡同，忘记了自己其实还有另外的路可走。

程序语言的常见设计错误(1) - 片面追求短小

我经常以自己写“非常短小”的代码为豪。有一些人听了之后很赞赏，然后说他也很喜欢写短小的代码，接着就开始说C语言其实有很多巧妙的设计，可以让代码变得非常短小。然后我才发现，这些人所谓的“短小”跟我所说的“短小”完全不是一回事。

我的程序的“短小”是建立在语义明确，概念清晰的基础上的。在此基础上，我力求去掉冗余的，绕弯子的，混淆的代码，让程序更加直接，更加高效的表达我心中设想的“模型”。这是一种在概念级别的优化，而程序的短小精悍只是它的一种“表象”。就像是整理一团电线，并不是把它们揉成一团然后塞进一个盒子里就好。这样的做法只会给你以后的工作带来更大的麻烦，而且还有安全隐患。

所以我的这种短小往往是在语义和逻辑层面的，而不是在语法上死抠几行代码。我绝不会为了程序显得短小而让它变得难以理解或者容易出错。相反，很多其它人所追求的短小，却是盲目的而没有原则的。很多时候这些小伎俩都只是在语法层面，比如想办法把两行代码“搓”成一行。可以说，这种“片面追求短小”的错误倾向，造就了一批语言设计上的错误，以及一批“擅长于”使用这些错误的程序员。

现在我举几个简单的“片面追求短小”的语言设计。

自增减操作

很多语言里都有`i++`和`++i`这两个“自增”操作和`i--`和`--i`这两个“自减”操作（下文合称“自增减操作”）。很多人喜欢在代码里使用自增减操作，因为这样可以“节省一行代码”。殊不知，节省掉的那区区几行代码比起由此带来的混淆和错误，其实是九牛之一毛。

从理论上讲，自增减操作本身就是错误的设计。因为它们把对变量的“读”和“写”两种根本不同的操作，毫无原则的合并在一起。这种对读写操作的混淆不清，带来了非常难以发现的错误。相反，一种等价的，“笨”一点的写法，`i = i + 1`，不但更易理解，而且在逻辑上更加清晰。

有些人很在乎`i++`与`++i`的区别，去追究`(i++) + (++i)`这类表达式的含义，追究`i++`与`++i`谁的效率更高。这些其实都是徒劳的。比如，`i++`与`++i`的效率差别，其实来自于早期C编译器的愚蠢。因为`i++`需要在增加之后返回`i`原来的值，所以它其实被编译为：

```
(tmp = i, i = i + 1, tmp)
```

但是在

```
for (int i = 0; i < max; i++)
```

这样的语句中，其实你并不需要在`i++`之后得到它自增前的值。所以有人说，在这里应该用`++i`而不是`i++`，否则你就会浪费一次对中间变量`tmp`的赋值。而其实呢，一个良好设计的编译器应该在两种情况下都生成相同的代码。这是因为在`i++`的情况下，代码其实先被转化为：

```
for (int i = 0; i < max; (tmp = i, i = i + 1, tmp))
```

由于`tmp`这个临时变量从来没被用过，所以它会被编译器的“dead code elimination”消去。所以编译器最后实际上得到了：

```
for (int i = 0; i < max; i = i + 1)
```

所以，“精通”这些细微的问题，并不能让你成为一个好的程序员。很多人所认为的高明的技巧，经常都是因为早期系统设计的缺陷所致。一旦这些系统被改进，这些技巧就没什么用处了。

真正正确的做法其实是：完全不使用自增减操作，因为它们本来就是错误的设计。

好了，一个小小的例子，也许已经让你意识到了片面追求短小程序所带来的认知上，时间上的代价。很可惜的是，程序语言的设计者们仍然在继续为此犯下类似的错误。一些新的语言加入了很多类似的旨在“缩短代码”，“减少打字量”的雕虫小技。也许有一天你会发现，这些雕虫小技所带来的，除了短暂的兴奋，其实都是在浪费你的时间。

赋值语句返回值

在几乎所有像C，C++，Java的语言里，赋值语句都可以被作为值。之所以设计成这样，是因为你就可以写这样的代码：

```
if (y = 0) { ... }
```

而不是

```
y = 0;
if (y) { ... }
```

程序好像缩短了一行，然而，这种写法经常引起一种常见的错误，那就是为了写 `if (y == 0) { ... }` 而把 `==` 比较操作少打了一个 `=`，变成了 `if (y = 0) { ... }`。很多人犯这个错误，是因为数学里的 `=` 就是比较两个值是否相等的意思。

不小心打错一个字，就让程序出现一个 bug。不管 `y` 原来的值是多少，经过这个“条件”之后，`y` 的值都会变成 0。所以这个判断语句会一直都为“假”，而且一声不吭的改变了 `y` 的值。这种 bug 相当难以发现。这就是另一个例子，说明片面追求短小带来的不应有的问题。

正确的做法是什么呢？在一个类型完备的语言里面，像 `y=0` 这样的赋值语句，其实是不应该可以返回一个值的，所以它不允许你写：

```
x = y = 0
```

或者

```
if (y = 0) { ... }
```

这样的代码。

`x = y = 0` 的工作原理其实是这样：经过 parser 它其实变成了 `x = (y = 0)`（因为 `=` 操作符是“右结合”的）。`x = (y = 0)` 这个表达式也就是说 `x` 被赋值为 `(y = 0)` 的值。注意，我说的是 `(y = 0)` 这整个表达式的值，而不是 `y` 的值。所以这里的 `(y = 0)` 既有副作用又是值，它返回 `y` 的“新值”。

正确的做法其实是：`y = 0` 不应该具有一个值。它的作用应该是“赋值”这种“动作”，而不应该具有任何“值”。即使牵强一点硬说它有值，它的值也应该是 `void`。这样一来 `x = y = 0` 和 `if (y = 0)` 就会因为“类型不匹配”而被编译器拒绝接受，从而避免了可能出现的错误。

仔细想一想，其实 `x = y = 0` 和 `if (y = 0)` 带来了非常少的好处，但它们带来的问题却耗费了不知道多少人多少时间。这就是我为什么把它们叫做“小聪明”。

思考题：

1. Google 公司的代码规范里面规定，在任何情况下 `for` 语句和 `if` 语句之后必须写花括号，即使 C 和 Java 允许你在其只包含一行代码的时候省略它们。比如，你不能这样写

```
for (int i=0; i < n; i++)
    some_function(i);
```

而必须写成

```
for (int i=0; i < n; i++) {
    some_function(i);
}
```

请分析：这样多写两个花括号，是好还是不好？

（提示，Google 的代码规范在这一点上是正确的。为什么？）

2. 当我第二次到 Google 实习的时候，发现我一年前给他们写的代码，很多被调整了结构。几乎所有如下结构的代码：

```
if (condition) {
    return x;
} else {
    return y;
}
```

都被人改成了：

```
if (condition) {
    return x;
}
return y;
```

请问这里省略了一个 `else` 和两个花括号，会带来什么好处或者坏处？

（提示，改过之后的代码不如原来的好。为什么？）

3. 根据本文对于自增减操作的看法，再参考传统的图灵机的设计，你是否发现图灵机的设计存在类似的问题？你如

何改造图灵机，使得它不再存在这种问题？

(提示，注意图灵机的“读写头”。)

4. 参考这个《[Go 语言入门指南](#)》，看看你是否能从中发现由于“片面追求短小”而产生的，别的语言里都没有的设计错误？

谈语法



使用和研究过这么多程序语言之后，我觉得几乎不包含多余功能的语言，只有一个：Scheme。所以我觉得它是学习程序设计最好的入手点和进阶工具。当然 Scheme 也有少数的问题，而且缺少一些我想要的功能，但这些都瑕不掩瑜。在用了很多其它的语言之后，我觉得 Scheme 真的是非常优美的语言。

要想指出 Scheme 所有的优点，并且跟其它语言比较，恐怕要写一本书才讲的清楚。所以在这篇文章里，我只提其中一个最简单，却又几乎被所有人忽视的方面：语法。

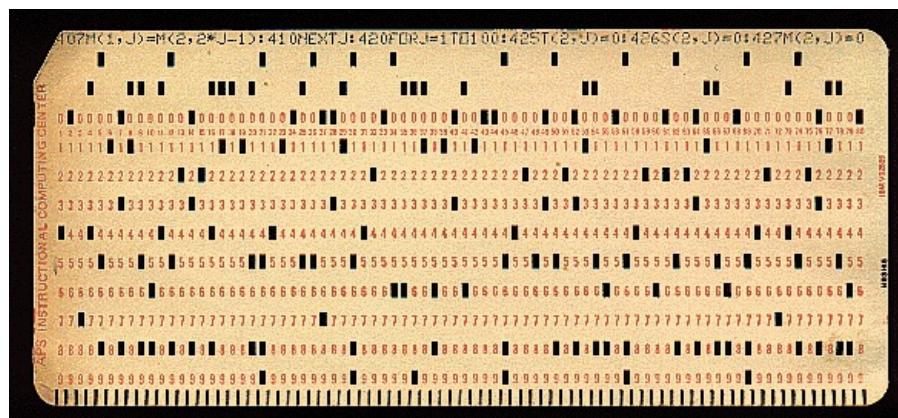
其它的 Lisp “方言”也有跟 Scheme 类似的语法（都是基于“S表达式”），所以在这篇（仅限这篇）文章里我所指出的“Scheme 的优点”，其实也可以作用于其它的 Lisp 方言。从现在开始，“Scheme”和“Lisp”这两个词基本上含义相同。

我觉得 Scheme (Lisp) 的基于“S表达式”(S-expression) 的语法，是世界上最完美的设计。其实我希望它能更简单一点，但是在现存的语言中，我没有找到第二种能与它媲美。也许在读过这篇文章之后，你会发现这种语法设计的合理性，已经接近理论允许的最大值。

为什么我喜欢这样一个“全是括号，前缀表达式”的语言呢？这是出于对语言结构本质的考虑。其实，我觉得语法是完全不应该存在的东西。即使存在，也应该非常的简单。因为语法其实只是对语言的本质结构，“抽象语法树”(abstract syntax tree, AST)，的一种编码。一个良好的编码，应该极度简单，不引起歧义，而且应该容易解码。在程序语言里，这个“解码”的过程叫做“语法分析”(parse)。

为什么我们却又需要语法呢？因为受到现有工具（操作系统，文本编辑器）的限制，到目前为止，几乎所有语言的程序都是用字符串的形式存放在文件里的。为了让字符串能够表示“树”这种结构，人们才给程序语言设计了“语法”这种东西。但是人们喜欢耍小聪明，在有了基本的语法之后，他们开始在这上面大做文章，使得简单的问题变得复杂。

Lisp (Scheme 的前身) 是世界上第二老的程序语言。最老的是 Fortran。Fortran 的程序，最早的时候都是用打孔机打在卡片上的，所以它其实是几乎没有语法可言的。



显然，这样写程序很痛苦。但是它却比现代的很多语言有一个优点：它没有歧义，没有复杂的 parse 过程。

在 Lisp 诞生的时候，它的设计者们一下子没能想出一种好的语法，所以他们决定干脆先用括号把这语法树的结构全都括起来，一个不漏。等想到更好的语法再换。

自己想一下，如果要表达一颗“树”，最简单的编码方式是什么？就是用括号把每个节点的“数据”和“子节点”都括起来放在一起。Lisp 的设计者们就是这样想的。他们把这种完全用括号括起来的表达式，叫做“S 表达式”（S 代表

“symbolic”）。这貌似很“粗糙”的设计，甚至根本谈不上“设计”。奇怪的是，在用过一段时间之后，他们发现自己已经爱上了这个东西，再也不想设计更加复杂的语法。于是S表达式就沿用至今。

在使用过 Scheme , Haskell , ML , 和常见的 Java , C , C++ , Python , Perl , 之后，我也惊讶的发现，Scheme 的语法，不但是最简单，而且是最好看的一个。这不是我情人眼里出西施，而是有一定理论依据的。

首先，把所有的结构都用括号括起来，轻松地避免了别的语言里面可能发生的“歧义”。程序员不再需要记忆任何“运算符优先级”。

其次，把“操作符”全都放在表达式的最前面，使得基本算术操作和函数调用，在语法上发生完美的统一，而且使得程序员可以使用几乎任何符号作为函数名。

在其他的语言里，函数调用看起来像这个样子：`f(1)`，而算术操作看起来是这样：`1+2`。在 Lisp 里面，函数调用看起来是这样(`f 1`)，而算术操作看起来也是这样(`+ 1 2`)。你发现有什么共同点吗？那就是 `f` 和 `+` 在位置上的对应。实际上，加法在本质也是一个函数。这样做的好处，不但是突出了加法的这一本质，而且它让人可以用跟定义函数一模一样的方式，来定义“运算符”！这比起 C++ 的“运算符重载”强大很多，却又极其简单。

关于“前缀表达式”与“中缀表达式”，我有一个很独到的见解：我觉得“中缀表达式”其实是一种过时的，来源于传统数学的历史遗留产物。几百年以来，人们都在用 $x+y$ 这样的符号来表示加法。之所以这样写，而不是 $(+ x y)$ ，是因为在没有计算机以前，数学公式都得写在纸上，写 $x+y$ 显然比 $(+ x y)$ 方便简洁。但是，中缀表达式却是容易出现歧义的。如果你有多个操作符，比如 $1+2*3$ 。那么它表示的是 $(+ 1 (* 2 3))$ 呢，还是 $(* (+ 1 2) 3)$ ？所以才出现了“运算符优先级”这种东西。看见没有，S表达式已经在这里显示出它没有歧义的优点。你不需要知道 `+` 和 `*` 的优先级，就能明白 $(+ 1 (* 2 3))$ 和 $(* (+ 1 2) 3)$ 的区别。第一个先乘后加，而第二个先加后乘。

对于四则运算，这些优先级还算简单。可是一旦有了更多的操作，就容易出现混淆。这就是为什么数学（以及逻辑学）的书籍难以看懂。实际上，那些看似复杂的公式，符号，不过是在表示一些程序里的“数据结构”，“对象”以及“函数”。大部分读数学书的时间，其实是浪费在琢磨这些公式：它们到底要表达的什么样一个“数据结构”或者“操作”！这个“琢磨”的过程，其实就是程序语言里所谓的“语法分析”（parse）。

这种问题在微积分里面就更加明显。微积分难学，很大部分原因，就是因为微积分的那些传统的运算符，其实不是很好的设计。如果你想了解更好的设计，可以参考一下 Mathematica 的公式设计。试试在 Mathematica 里面输入“单行”的微积分运算（而不使用它传统的“2D语法”）。

其实 Lisp 已经可以轻松地表示这种公式，比如对 x^2 进行微分，可以表示成

```
(D '(^ x 2) 'x)
```

看到了吗？微分不过是一个用于处理符号的函数 `D`，输入一个表达式和另一个符号，输出一个新的表达式。

同样的公式，传统的数学符号是这个样子：

$$\frac{d}{dx}(x^2)$$

这是什么玩意啊？`d` 除以 `dx`，然后乘以 `x` 的平方？

在 Lisp 里，你其实可以比较轻松地实现符号微分的计算。SICP里貌似有一节就是教你写个符号微分程序。做微积分这种无聊的事情，就是应该交给电脑去做。总之，这从一方面显示了，Lisp 的语法其实超越了传统的数学。

其实我一直都在想，如果把数学看成是一种程序语言，它也许就是世界上语法最糟糕的语言。数学里的“变量”，几乎总是没有明确定义的作用域（scope）。也就是说他们只有“全局变量”。上一段话的 `x`，跟下一段话的 `x`，经常指的不是同一个东西。所以训练有素的数学家，总是避免使用同一个符号来表示两种不同的东西。很快他们就发现所有的拉丁字母都用光了，于是乎开始用希腊字母。大写的，小写的，粗体的，斜体的，花体的，..... 而其实，他们只不过是想实现 C++ 里的“namespace”。

可惜的是，很多程序语言的设计者没能摆脱数学的思想束缚，对数学和逻辑有盲目崇拜的倾向。所以他们继续在新的语言里使用中缀表达法。Haskell , ML , Coq , Agda , 这些“超高级”的语言设计，其实都中了这个圈套。在 Coq 和 Agda 里面，你不但可以使用中缀表达式，还可以定义所谓的“mixfix”表达式。这样其实是把简单的问题复杂化。想让自己看起来像“数学”，很神秘的样子，其实是学会了数学的糟粕，自讨苦吃。

另外，由于 Lisp 的表达能力和灵活性比其他语言要大很多，所以类似 C 或者 Pascal 那样的语法其实不能满足 Lisp 的需要。在 Lisp 里，你可以写 `(+ 10 (if test 1 2))` 这样的代码，然而如果你使用 C 那样的无括号语法，就会发现没法很有效的嵌入里面的那个条件语句而不出现歧义。这就是为什么 C 必须使用 `test? 1 : 2` 这样的语法来表示 Lisp 的 if 能表示的东西。然而即使如此，你仍然会经常被迫加上一对括号，结果让程序非常难看，最后的效果其实还不如用 Lisp 的语法。在 C 这样的语言里，由于结构上有很多限制，所以才觉得那样的语法还可以。可是一旦加入 Lisp 的那些表达能力强的结构，就发现越来越难看。JavaScript (node.js) 就是对此最好的一个证据。

最后，从美学的角度上讲，S表达式是很美观的设计。所有的符号都用括号括起来，这形成一种“流线型”的轮廓。而且由于可以自由的换行排版，你可以轻松地对齐相关的部分。在 Haskell 里，你经常会发现一些很蹩脚，很难看的地方。这是因为中缀表达式的“操作符”，经常不能对在一起。比如，如果你有像这样一个 case 表达式：

```
case x
  Short _ -> 1
  VeryLooooooooooooooog _ -> 2
```

为了美观，很多 Haskell 程序员喜欢把那两个箭头对齐。结果就成了这样：

```
case x
  Short _           -> 1
  VeryLooooooooooooooog _ -> 2
```

作为一个菜鸟级摄影师，你不觉得第一行中间太“空”了一点吗？

再来看看S表达式如何表达这东西：

```
(case x
  (-> (Short _) 1)
  (-> (VeryLooooooooooooooog _) 2))
```

发现“操作符总在最前”的好处了吗？不但容易看清楚，而且容易对齐，而且没有多余的间隙。

其实我们还可以更进一步。因为箭头的两边全都用括号括起来了，所以其实我们并不需要那两个箭头就能区分“左”和“右”。所以我们可以把它简化为：

```
(case x
  ((Short _) 1)
  ((VeryLooooooooooooooog _) 2))
```

最后我们发现，这个表达式“进化”成了 Lisp 的 case 表达式。

Lisp 的很多其它的设计，比如“垃圾回收”，后来被很多现代语言（比如 Java）所借鉴。可是人们遗漏了一个很重要的东西：Lisp 的语法，其实才是世界上最好的语法。

Oberon 操作系统：被忽略的珍宝

推荐一篇很久以前看的文章：[Oberon - The Overlooked Jewel](#)

它介绍的是 Niklaus Wirth 设计的一种操作系统，叫做 Oberon。Niklaus Wirth 就是大家熟知的 Pascal 语言的设计者。绝大部分人都没听说过有 Oberon 这个东西存在，更难以把它跟 Niklaus Wirth 的大名挂上钩。所以作者说：“Wirth 因为 Pascal 而闻名于世，可是接下来几年，他成为了 Pascal 的受害者。”确实是这样。Wirth 一直都不觉得 Pascal 是他的杰作。我想他应该会更喜欢以 Oberon 闻名于世。

Oberon 比起 Unix，有很大的不同，在于它的数据都是结构化的。进程间不通过字符串交换数据，而是直接使用数据结构。很奇特的一点是，Oberon 操作系统是用一种同名的程序语言（Oberon 语言）写成。令人惊讶的是，在那个年代，ETH 计算机系的所有教职员，学生，包括办公室的大妈，都是用的这种操作系统。

操作系统的功能，真是天外有天。

谈 Linux , Windows 和 Mac

这段时间受到很多人的来信。他们看了我很早以前写的推崇 Linux 的文章，想知道如何“抛弃 Windows，学习 Linux”。天知道他们在哪找到那么老的文章，真是好事不出门…… 我觉得我有责任消除我以前的文章对人的误导，洗清我这个“Linux 狂热分子”的恶名。我觉得我已经写过一些澄清的文章了，可是怎么还是有人来信问 Linux 的问题。也许因为感觉到“舆论压力”，我把文章都删了。

简言之，我想对那些觉得 Linux 永远也学不会的“菜鸟”们说：

1. Linux 和 Unix 里面包含了一些非常糟糕的设计。不要被 Unix 的教条主义者吓倒。学不会有些东西很多时候不是你的错，而是 Linux 的错，是“Unix 思想”的错。不要浪费时间去学习太多工具的用法，钻研稀奇古怪的命令行。那些貌似难的，复杂的东西，特别要小心分析。
2. Windows 避免了 Unix , Linux 和 Mac OS X 的很多问题。微软是值得尊敬的公司，是真正在乎程序开发工具的公司。我收回曾经对微软的鄙视态度。请菜鸟们吸收 Windows 设计里面好的东西。另外 Visual Studio 是非常好的工具，会带来编程效率的大幅度提升。请不要歧视 IDE。要正视 Emacs , VIM 等文本编辑器的局限性。当然，这些正面评价不等于说你应该为微软工作。就像我喜欢 iPhone，但是却不一定想给 Apple 工作一样。
3. 学习操作系统最好的办法是学会（真正的）程序设计思想，而不是去“学习”各种古怪的工具。所有操作系统，数据库，Internet，以至于 WEB 的设计思想（和缺陷），几乎都能用程序语言的思想简单的解释。

先说说我现在对 Linux 和相关工具（比如 TeX）的看法吧。我每天上班都用 Linux，可是回家才不想用它呢。上班的时候，我基本上只是尽我所能的改善它，让它不要给我惹麻烦。Unix 有许许多多的设计错误，却被当成了教条，传给了新一代又一代的程序员，恶性循环。Unix 的 shell，命令，配置方式，图形界面，都是相当糟糕的。每一个新版本的 Ubuntu 都会在图形界面的设计上出现新的错误，让你感觉历史怎么会倒退。其实这只是表面现象。Linux 所用的图形界面（X Window）在本质上几乎是没救的。我不想在这里细说 Unix 的缺点，在它出现的早期，已经有人写了一本书，名叫 Unix Hater's Handbook，里面专门有一章叫做 The X-Windows Disaster。它分析后指出，X Window 貌似高明的 client-server 设计，其实并不像说的那么好。

这本书汇集了 Unix 出现的年代，很多人对它的咒骂。有趣的是，这本书有一个“反序言”，是 Unix 的创造者之一 Dennis Ritchie 写的。我曾经以为这些骂 Unix 的人都是一些菜鸟。他们肯定是智商太低，或者被 Windows 洗脑了，不能理解 Unix 的高明设计才在那里骂街。现在理解了程序语言的设计原理之后，才发现他们说的那些话里面居然大部分是实话！其实他们里面有些人在当年就是世界顶尖的编程高手，自己写过操作系统和编译器，功底不亚于 Unix 的创造者。在当年他们就已经使用过设计更加合理的系统，比如 Multics , Lisp Machine 等。

可惜的是，在现在的操作系统书籍里面，Multics 往往只是被用来衬托 Unix 的“简单”和伟大。Unix 的书籍喜欢在第一章讲述这样的历史：“Multics 由于设计过于复杂，试图包罗万象，而且价格昂贵，最后失败了。”可是 Multics 失败了吗？Multics , Oberon , IBM System/38 , Lisp Machine , …… 在几十年前就拥有了 Linux 现在都还没有的好东西。Unix 里面的东西，什么虚拟内存，文件系统，…… 基本上都是从 Multics 学来的。Multics 的机器，一直到 2000 年都还在运行。Unix 不但“窜改”了历史教科书，而且似乎永远不吸取教训，到现在还没有实现那些早期系统早就有的好东西。Unix 的设计几乎完全没有一致性和原则。各种工具程序功能重复，冗余，没法有效地交换数据。可是最后 Unix 靠着自己的“廉价”，“宗教”和“哲学”，战胜了别的系统在设计上的先进，统治了程序员的世界。

如果你想知道这些“失败的”操作系统里面有哪些我们现在都还没有的先进技术，可以参考这篇文章：Oberon - The Overlooked Jewel。它介绍的是 Niklaus Wirth (也就是 Pascal 语言的设计者) 的 Oberon 操作系统。

胜者为王，可是 Unix 其实是一个暴君，它不允许你批评它的错误。它利用其它程序员的舆论压力，让每一个系统设计上的错误，都被说成是用户自己的失误。你不敢说一个工具设计有毛病，因为如果别人听到了，就会以为你自己不够聪明，说你“人笨怪刀钝”。这就像是“皇帝的新装”里的人们，明明知道皇帝没穿衣服，还要说“这衣服这漂亮”！总而言之，“对用户友好”这个概念，在 Unix 的世界里是被歧视，被曲解的。Unix 的狂热分子很多都带有一种变态的“精英主义”。他们以用难用的工具为豪，鄙视那些使用“对用户友好”的工具的人。

我曾经强烈的推崇 FVWM , TeX 等工具，可是现在擦亮眼睛看来，它们给用户的界面，其实也是非常糟糕的设计，跟 Unix 一脉相承。他们把程序设计的许多没必要的细节和自己的设计失误，无情的暴露给用户。让用户感觉有那么多东西要记，仿佛永远也没法掌握它。实话说吧，当年我把 TeXbook 看了两遍，做完了所有的习题（包括最难的“double bend”习题）。几个月之后，几乎全部忘记干净。为什么呢？因为 TeX 的语言是非常糟糕的设计，它没有遵循程序语言设计的基本原则。

这里有一个鲜为人知的小故事。TeX 之所以有一个“扩展语言”，是 Scheme 的发明者 Guy Steele 的建议。那年夏天，Steele 在 Stanford 实习。他听说 Knuth 在设计一个排版系统，就强烈建议他使用一种扩展语言。后来 Knuth 采纳了他的建议。不幸的是 Steele 几个月后就离开了，没能帮助 Knuth 完成语言的设计。Knuth 老爹显然有我所说的那种“精英主义”，他咋总是设计一些难用的东西，写一些难懂的书？

一个好的工具，应该只有少数几条需要记忆的规则，就像象棋一样。而这些源于 Unix 的工具却像是“魔鬼棋”或者“三国杀”，有太多的，无聊的，人造的规则。有些人鄙视图形界面，鄙视 IDE，鄙视含有垃圾回收的语言（比如

Java)，鄙视一切“容易”的东西。他们却不知道，把自己沉浸在别人设计的繁复的规则中，是始终无法成为大师的。就像一个人，他有能力学会各种“魔鬼棋”的规则，却始终无法达到象棋大师的高度。所以，容易的东西不一定是坏的，而困难的东西也不一定是好的。学习计算机（或者任何其它工具），应该“只选对的，不选难的”。记忆一堆的命令，乱七八糟的工具用法，最后脑子里什么也不会留下。学习“原理性”的东西，才是永远不会过时的。

Windows 技术设计上的很多细节，也许在早期是同样糟糕的。但是它却向着更加结构化，更加简单的方向发展。Windows 的技术从 OLE，COM，发展到 .NET，再加上 Visual Studio 这样高效的编程工具，这些带来了程序员和用户效率的大幅度提高，避免了 Unix 和 C 语言的很多不必存在的问题。Windows 程序从很早的时候就能比较方便的交换数据。比如，OLE 让你可以把 Excel 表格嵌入到 Word 文档里面。不得不指出，这些是非常好的想法，是超越“Unix 哲学”的。相反，由于受到“Unix 哲学”的误导，Unix 的程序间交换数据一直以来都是用字符串，而且格式得不到统一，以至于很多程序连拷贝粘贴都没法正确进行。Windows 的“配置”，全都记录在一个中央数据库（注册表）里面，这样程序的配置得到大大的简化。虽然在 Win95 的年代，注册表貌似老是惹麻烦，但现在基本上没有什么问题了。相反，Unix 的配置，全都记录在各种稀奇古怪的配置文件里面，分布在系统的各个地方。你搞不清楚哪个配置文件记录了你想要的信息。每个配置文件连语法都不一样！这就是为什么用 Unix 的公司总是需要一个“系统管理员”，因为软件工程师们才懒得记这些麻烦的东西。

再来比较一下 Windows 和 Mac 吧。我认识一个 Adobe 的高级设计师。他告诉我说，当年他们把 Photoshop 移植到 Intel 构架的 Mac，花了两年时间。只不过换了个处理器，移植个应用程序就花了两年时间，为什么呢？因为 Xcode 比起 Visual Studio 真是差太多了。而 Mac OS X 的一些设计原因，让他们的移植很痛苦。不过他很自豪的说，当年很多人等了两年也没有买 Intel 构架的 Mac，就是因为他们在等待 Photoshop。最后他直言不讳的说，微软其实才是真正在乎程序员工具的公司。相比之下，Apple 虽然对用户显得友好，但是对程序员的界面却差很多。Apple 尚且如此，Linux 对程序员就更差了。可是有啥办法呢，有些人就是受虐狂。自己痛过之后，还想让别人也痛苦。就像当年的我。

我当然不是人云亦云。微软在程序语言上的造诣和投入，我看得很清楚。我只是通过别人的经历，来验证我已经早已存在的看法。所以一再宣扬别的系统都是向自己学习的 Apple 受到这样的评价，我也一点不惊讶。Mac OS X 毕竟是从 Unix 改造而来的，还没有到脱胎换骨的地步。我有一个 Macbook Air，一个 iPhone 5，和一个退役的，装着 Windows 7 的 T60。我不得不承认，虽然我很喜欢 Macbook 和 iPhone 的硬件，但我发现 Windows 在软件上的很多设计其实更加合理。

我为什么当年会鄙视微软？这很简单。我就是跟着一群人瞎起哄而已！他们说 Linux 能拯救我们，给我们自由。他们说微软是邪恶的公司……到现在我身边还有人无缘无故的鄙视微软，却不知道理由。可是 Unix 是谁制造的呢？是 AT&T。微软和 AT&T 哪个更邪恶呢？我不知道。但是你应该了解一下 Unix 的历史。AT&T 当年发现 Unix 有利可图，找多少人打了多少年官司？说微软搞垄断，其实 AT&T 早就搞过垄断了，还被拆散成了好几个公司。想想世界上还有哪一家公司，独立自主的设计出这从底至上全套家什：程序语言，编译器，IDE，操作系统，数据库，办公软件，游戏机，手机……我不得不承认，微软是值得尊敬的公司。

公司还不都一样，都是以利益为本的。我们程序员就不要被他们利用，作为利益斗争的炮灰啦。见到什么好就用什么，就学什么。自己学到的东西，又不属于那些垄断企业。我们都有自由的头脑。

当然我不是在这里打击 Linux 和 Mac 而鼓吹 Windows。这些系统的纷争基本上已经不关我什么事。我只是想告诉新人们，去除头脑里的宗教，偏激，仇恨和鄙视。每次仇恨一个东西，你就失去了向它学习的机会。

后记：“对用户友好”是一个值得研究，却又研究得非常不够的东西。很多 UI 的设计者，把东西设计的很漂亮，但是却不方便，不顺手。如果你想了解我认为怎样的设计才是“对用户友好的”，可以参考这篇博客《[什么是“对用户友好”](#)》

解密“设计模式”

有些人问我，你说学习操作系统的最好办法是学习程序设计。那我们是不是应该学习一些“设计模式”（design patterns）。这是一个我很早就有定论，而且经过实践检验的问题，所以想在这里做一个总结。

总的来说，如果光从字面上讲，程序里确实是一些“模式”可以发掘的。因为你总是可以借鉴以前的经验，用来构造新的程序。你可以把这种经验叫做“模式”。可是自从《设计模式》（通常叫做 GoF，“Gang of Four”，“四人帮”）这本书在 1994 年发表以来，“设计模式”这个词有了新的，扭曲的含义。它变成了一种教条，带来了公司里程序的严重复杂化以及效率低下。



GoF 借鉴的是一个叫 Christopher Alexander 的建筑师的做法。Alexander 给一些建筑学里的“设计模式”起了名字，试图让建筑师们有一些“共同语言”。可惜的是，Alexander 后来自己都承认，他的实验失败了。因为这些固定的模式，并没能有效地传递精髓的知识，没能让新手成长为出色的建筑师。

照搬模式东拼西凑，而不能抓住事物的本质，没有“灵感”，其实是设计不出好东西的。这就像照搬“模版”把作文写得再好，也成不了作家一样。

我孤陋寡闻，当听说这本书的时候，我已经学会了函数式编程，正在 Cornell 读 PhD，专攻程序语言设计。有一天由于好奇这书为什么名气这么大，我从图书馆借了一本回来看。我很快的发现，其实这本书的作者只是给早已经存在的编程方法起了一些新的名字而已。当时我就拿起一张纸，把所有的 20 来个设计模式跟我常用的编程概念做了一个映射。这个映射居然是“多对一”（many-to-one）的。也就是说，多个 GoF 设计模式，居然只对应同一个我每天都用的概念。有些概念是如此的不值一提，以至于我根本不需要一个名字来描述它，更不要说多个名字！

其中极少数值得一提的“模式”，也许是 visitor 和 interpreter。很可惜的是，只有很少的人明白如何使用它们。所谓的 visitor，本质上就是函数式语言里的含有“模式匹配”（pattern matching）的递归函数。在函数式语言里，这是多么轻松的事情。可是因为 Java 没有模式匹配，所以很多需要类似功能的人就得使用 visitor pattern。为了所谓的“通用性”，他们往往把 visitor pattern 搞出多层继承关系，让你转几道弯也搞不清楚到底哪个 visitor 才是干实事的。

其实，函数式语言的研究者们早就知道 visitor pattern 是怎么得来的。如果你想知道如何从无到有，一步一步“发明”出 Java 的 visitor pattern，可以参考《A Little Java, A Few Patterns》（发表于 1997 年）。



而 interpreter（解释器）模式呢？看了作者们写的例子程序之后，我发现他们其实并不会写解释器，或者说他们不知道如何写出优雅的，正确的解释器。如果你想知道如何写出好的解释器，可以参考我的博文《怎样写一个解释器》。

你说我在贬低这本书的真正价值，因为 GoF 说了：“我们的贡献，就是给这些编程方式起名字。这样让广大程序员有共同的语言。”如果这也叫贡献的话，我就可以写本书，给“空气”，“水”，“猪肉”这些东西全都起个新名字，让大家有“共同的语言”。这不是搞笑吗。

这不是我的一家之言，Peter Norvig 在 1998 年就做了一个演讲，指出在“动态语言”里面，GoF 的 20 几个模式，其中绝大部分都“透明”了。也就是说，你根本感觉不到它们的存在。这就像我刚才告诉你的。

Design Patterns in Dynamic Programming

Peter Norvig

Chief Designer, Adaptive Systems
Harlequin Inc.

Peter Norvig, Harlequin, Inc.

1

Object World, May 3, 1996

在这里 Norvig 的观点是正确的，不过需要小心一个概念错误。Norvig 对“静态语言”的概念是有局限性的。有的静态语言其实也能传递函数作为参数，而且不像 Java 那样什么都得放进 class 里。这样的静态语言，其实也可以避免大部分 GoF 设计模式。而“动态语言”这个概念，在程序语言的理论里面，其实是没有明确的定义的。“动态语言”其实也能进行某些“静态类型检查”。不过在 1998 年，我还是个啥都不懂的屁孩，所以这里就不跟 Norvig 大叔计较了。

既然老人们都有历史局限性，那么为啥我还跟 GoF 找茬？本来这本书很老了，如果没有人再被它误导的话，这篇博文也就不必存在了。可是当我在 Google 实习的时候，我发现几乎每个程序员的书架上都有一本 GoF！我在 Google 实习了两次，第一次的时候代码全都是我一个人写的，所以没有使用任何 GoF 设计模式。代码直接，精巧而简单。当我第二次回到 Google，发现我的代码里已经被加入了各种 factory，visitor，…… 其实啥好事也没做，只不过让我的代码弯了几道弯，让人难以理解。

可见一本坏书，毁掉的不只是一代程序员。鉴于如此，特发此文。各位新手，希望你们敲响警钟，不要再走上这条老路，写出代码来让大家痛苦。

Braid - 一个发人深思的游戏



我已经很久很久没有打游戏了（如果不算是 Angry Birds 之类用来打发时间的游戏的话）。我的最后一个真正意义上的游戏机是 PlayStation 1。在那上面，我真正欣赏的最后一个游戏，是 Metal Gear Solid (1)。

我曾经是一个游戏迷，可是进入了计算机专业的学习之后，我就开始失去对游戏的兴趣，基本上每玩一个都让我失望一次，不管别人把它吹的多么“经典”。不知道为什么，别人玩得津津有味的游戏，我玩一会儿就把它里面的“公式”都看透了。我清楚地知道这游戏的设计者是怎么在“耍我”，在如何想方设法浪费我的时间。

同样的，别人看得津津有味的小说和电影，我经常一看开头就能猜到它要怎么发展，知道这编剧是怎么在胡编滥造，索然无味。所以我基本上不去影院看最新的电影，宁愿在网上看一些几十年前的老电影。我貌似只喜欢那些能让我“猜不透”的东西。

Braid，就是这样一个让我没猜得透的游戏。

这是一个同事推荐的。本来已经对电玩完全失望的我，破例的从 App Store 买了来。玩过之后觉得真的很不错，有一种所谓的“mind blowing”的感觉。以至于我花了两整天时间，废寝忘食，把它给打通关了。

Braid 的主体结构，和最古老的“超级玛丽”没什么两样。一个小人，可以跑，可以跳。一些小怪物，跑来跑去的。你可以跳起来踩它们。

最终的目标，是收集到所有的拼图，然后把它们组合成图片。组合图片是很容易的事情。游戏的难度其实在于如何拿到这些拼图。它们有可能被挂在很高的地方，或者被门挡住。

可是这有什么值得一提的呢？这游戏很不一样的地方是，它给你提供了几种绝无仅有的“超能力”，而且把它们与谜题结合得几乎天衣无缝。

你有三种超能力：

逆转时间的能力

在任何时候按下 Shift 键，游戏的时间就会逆转，“undo”之前的所有动作。即使你死了，都是可以复活的。死去的小怪物们也会复活。可是就算这样，有些拼图还是很难拿到。

值得一提的是，时间逆转的时候，画面是流畅无缺损的，连爆炸场面都会“收缩”。更令人赞叹的是，游戏的背景音乐也会同步逆转。如果在时间逆转的时候按“上”，“下”键，就可以调整时间“快退”和“快进”的速度。当然，此时的场景就像录像机在快退或者快进。

产生“多重现实”的能力



在某些章节，你可以实现“多重现实”。做一个动作，然后按 Shift 键让时间逆转，当你停止逆转的时候，你的影子就会开始“redo”刚才的那段“历史”。而这个时候你可以做一些不同于以前的事情。这就好像有两个世界，一新一旧，从“历史的分叉点”开始，同步交汇。

你必须掌握好时间才能跟影子合作，因为影子的行动速度是不受你的“现场控制”的，它只是按部就班的重演你 undo 掉的历史。

扭曲时间的指环



在某些章节，你会有机会使用一个魔法指环。把这个指环放在地上之后，它会在附近的球状空间中形成时间的“扭曲”。这有点像黑洞的原理。越是靠近指环的位置，时间流动越慢。而当你远离指环，时间就逐渐恢复正常。指环的巧妙使用，是解决这些章节谜题的关键。

同样的，音乐与指环的特异功能是完美配合的。当你靠近指环的时候，背景音乐就会出现相应程度的扭曲。有点像录音机卡带的感觉 :)

在解决了所有的谜题之后，我回味了一下，自己为什么欣赏 Braid。这也许是因为它符合一个优秀的，非低级趣味的游戏设计：屈指可数的简单规则，却可以组合起来，制造出许许多多的变化。

你只有3种超能力，但是如何利用和“组合”这些超能力，却形成了解决谜题的关键。有些题目很有点难度，以至于你会希望有第4种超能力出现，或者希望捡到别的什么“法宝”。可是它们是不存在的。你必须使用那仅有的3种能力，加上巧妙的思索，细心的观察，才能达到目的。在解决了一个很难的谜题之后，你往往会一拍脑袋：哇，我怎么一开头没想到！

TeXmacs : 一个真正“所见即所得”的排版系统



The screenshot shows the TeXmacs interface with a document titled "decidability.tm". The menu bar includes "File", "Edit", "View", "Tools", "Help", and "TeXmacs". The toolbar contains icons for mathematical symbols like Σ , $\{\}$, \sim , \otimes , \prec , \rightarrow , \neq , Γ , B , C , \mathfrak{F} , \mathbb{B} , op , Σ , \mathcal{E} , T , and various document-related icons.

The main window displays the following text:

1.4 Formal proof with mapping reduction

Suppose we have D_{HALT} , a decider for HALT. We define the mapping reduction as:

$$f(\langle M, w \rangle) = \langle M', w \rangle$$

where the TM M' is constructed as computing the function:

$M'(x)$	=	if eval($\langle M, x \rangle$) then accept else loop
---------	---	---

This means exactly (but concisely) what the above informal description says:

Construct the following machine M' .
“On input x :

1. Run M on x .
2. If M accepts, *accept*.
3. If M rejects, enter a loop.”

Notice although M' takes input named “ x ”, the actual input is w when it is simulated. This is because when $\langle M', w \rangle$ is passed as input to a decider of HALT, it uses the w part as the *actual argument* for M' . This is like the difference between a formal parameter and the actual argument in a function call when using a programming language such as Java or Python.

We can see the behavior of M , M' and D_{HALT} in the following table:

M accepts w	M' accepts w	D_{HALT} accepts $\langle M', w \rangle$
M does not accept w	M' loops	D_{HALT} rejects $\langle M', w \rangle$

Table 1. Behavior table for M , M' and D_{HALT}

好久没有推荐过自己喜欢的软件了，现在推荐一款我在美国做数学作业的私家法宝：TeXmacs。我恐怕不可能跟以前那么有闲心写个长篇的 TeXmacs 说明文档了，不过这东西如此的简单好用，所以基本上不用我写什么文档了。鉴于知道的人很少，不理解它的人很多，这里只是帮它打个广告，吊一下胃口。

TeXmacs 的主要特点是：

- 跟 Lyx 等不同，它不是一个 TeX 的“前端”，而是一个完全独立的，超越 TeX 的系统。TeXmacs 拥有跟 TeX 相同，甚至更好的排版美观程度。这是因为它采用跟 TeX 一样的排版算法，并且用 C++ 重新实现。据说分页的算法比 TeX 的还要好些。
- 拥有超越 Word (或者任何一款字处理软件) 的，真正的“所见即所得”(WYSIWYG)。Word 所谓的“所见即所得”其实是假的。所见即所得的含义应该是，屏幕上显示的内容，跟打印下来的完全一样。可是 Word 能做到吗？打印一个文档出来你就发现跟屏幕上显示的有很大区别，一般来说屏幕上显示的要粗糙一些。一些 TeX 的前端，比如 Lyx, Scientific Workspace 等也是类似的，它们都不能达到真正的所见即所得。
- 直接可在屏幕文档里绘图。完全可视化的表格，公式编辑环境。这些都是比 TeX 方便高效很多的方式。需要当心的是，用过 TeXmacs 一段时间之后，你会发现回到 TeX 的公式编辑方式简直就像回到原始社会。
- 非常人性化的按键设计。比如，在数学公式环境下，你按任意一个字符，然后就可以用多次 TAB 键相继选择“拓扑相同”的字符。举个例子，如果你按 @，然后再按几下 TAB，就会发现这个字符变成各种各样的圆圈形的字符。如果你按 >，再按 =，就会出现大于等于号，之后再按 TAB，就会相继出现大于等于号的各种变体。
- 在直观的同时不失对底层结构的控制。比如，(见下图) 窗口右下角的状态栏，显示出当前光标位置的“上下文”是“proof eqnarry* (1,1) start”，这表示的是这是在一个 proof 环境里的 eqnarry 的坐标 (1,1) 的开始处。当你使用 Ctrl-Backspace，最靠近光标的那层“环境”会被删除。比如，如果你现在的字体是斜体，那么在 Ctrl-Backspace 之后，字体就立即还原成正体。

3. Prove by mathematical induction that

$$\forall n \geq 2 : 1 + 2^n < 3^n$$

Proof. Base case. For $n = 2$, we have $1 + 2^2 = 5 < 9 = 3^2$. So the base case holds.

Inductive step. Suppose the proposition holds for $n - 1$, that is, $1 + 2^{n-1} < 3^{n-1}$ (IH), we prove that it holds for n . We have

$\begin{aligned} 1 + 2^n &= 1 + 2(1 + 2^{n-1} - 1) \\ &\leq 1 + 2(3^{n-1} - 1) \\ &\leq 1 + 3(3^{n-1} - 1) \\ &= 3^n - 2 \\ &\leq 3^n. \end{aligned}$

The inductive step holds, thus the proposition holds for all $n \geq 2$. □

math roman 10 proof eqnarray* (1,1) start ...

- 结构化的浏览功能。比如，按 Ctrl-PgUp, Ctrl-PgDn 就可以在“相同类型”的结构里上下跳转。比如，如果你在小节标题里按这个键，就可以迅速的浏览所有的小节标题。如果你在数学公式里按这个键，就可以迅速浏览所有的数学公式。
- 与交互式程序接口。支持很多种计算机代数系统，和交互式软件，比如 MAXIMA，Octave，…… 这些系统返回的数学公式会直接被 TeXmacs 显示为“TeX 效果”。使用 Scheme 作为嵌入式语言，并且可以使用它来扩展系统。这比起 TeX 的语言是非常大的进步。

目前由于 TeX 的垄断地位，以及由于 TeXmacs 是法国人做的，这个系统在美国还不是很流行，很多人都没听说过有这种东西存在。学术圈的很多人由于受到某种错误思想的“洗脑”，都不理解这种图形化编辑软件的价值。希望中国人民和法国人民一样后来居上，超越美国。

想要迅速的掌握 TeXmacs 的基本用法，可以参考我绘制的 [TeXmacs 思维导图](#)：



怎样写一个解释器

写一个解释器，通常是设计和实现程序语言的第一步。解释器是简单却又深奥的东西，以至于好多人都不会写，所以我决定写一篇这方面的入门读物。

虽然我试图从最基本的原理讲起，尽量不依赖于其它知识，但这并不是一本编程入门教材。我假设你已经理解 Scheme 语言，以及基本的编程技巧（比如递归）。如果你完全不了解这些，那我建议你读一下 [SICP](#) 的第一，二章，或者 [HtDP](#) 的前几章，习题可以不做。注意不要读太多书，否则你就回不来了 :-)

当然你也可以直接读这篇文章，有不懂的地方再去查资料。

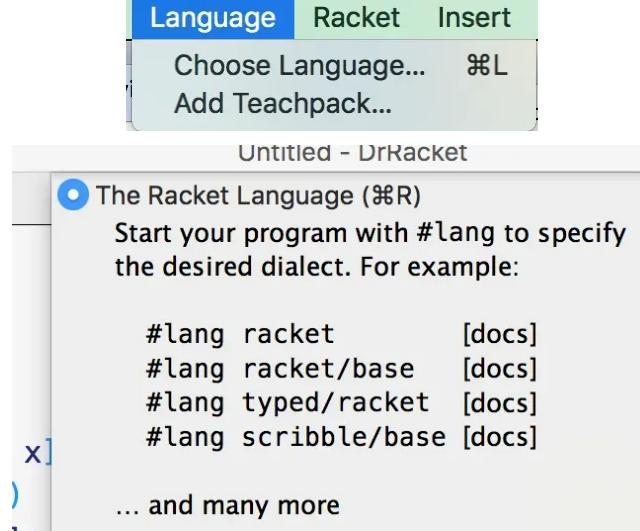
实现语言容易犯的一个错误，就是一开头就试图去实现很复杂的语言（比如 JavaScript 或者 Python）。这样你很快就会因为这些语言的复杂性，以及各种历史遗留的设计问题而受到挫折，最后不了了之。学习实现语言，最好是从最简单，最干净的语言开始，迅速写出一个可用的解释器。之后再逐步往里面添加特性，同时保持正确。这样你才能有条不紊地构造出复杂的解释器。

因为这个原因，这篇文章只针对一个很简单的语言，名叫“R2”。它可以作为一个简单的计算器用，还具有变量定义，函数定义和调用等功能。

我们的工具：Racket

本文的解释器是用 Scheme 语言实现的。Scheme 有很多的“实现”，这里我用的实现叫做 Racket，它可以在[这里免费下载](#)。为了让程序简洁，我用了一点点 Racket 的模式匹配（pattern matching）功能。我对 Scheme 的实现没有特别的偏好，但 Racket 方便易用，适合教学。如果你用其它的 Scheme 实现，可能得自己做一些调整。

Racket 具有宏（macro），所以它其实可以变成很多种语言。如果你之前用过 DrRacket，那它的“语言设置”可能被你改成了 R5RS 之类的。所以如果下面的程序不能运行，你可能需要检查一下 DrRacket 的“语言设置”，把 Language 设置成“Racket”。



Racket 允许使用方括号不只是圆括号，所以你可以写这样的代码：

```
(let ([x 1]
      [y 2])
  (+ x y))
```

方括号跟圆括号可以互换，唯一的要求是方括号必须和方括号匹配。通常我喜欢用方括号来表示“无动作”的数据（比如上面的 [x 1], [y 2]），这样可以跟函数调用和其它具有“动作”的代码，产生“视觉差”。这对于代码的可读性是一个改善，因为到处都是圆括号的话，确实有点太单调，容易打瞌睡。

另外，Racket 程序的最上面都需要加上像 `#lang racket` 这样的语言选择标记，这样 Racket 才可以知道你想用哪个语言变种。

解释器是什么

准备工作就到这里。现在我来谈一下，解释器到底是什么。说白了，解释器跟计算器差不多。解释器是一个函数，你输入一个“表达式”，它就输出一个“值”，像这样：



比如，你输入表达式 `'(+ 1 2)`，它就输出值，整数3。表达式是一种“表象”或者“符号”，而值却更加接近“本质”或者“意义”。我们“解释”了符号，得到它的意义，这也许就是为什么它叫做“解释器”。

需要注意的是，表达式是一个数据结构，而不是一个字符串。我们用一种叫“S 表达式”（S-expression）的结构来存储表达式。比如表达式 `'(+ 1 2)` 其实是一个链表（list），它里面的内容是三个符号（symbol）：`+`, `1` 和 `2`，而不是字符串“`(+ 1 2)`”。

从 S 表达式这样的“结构化数据”里提取信息，方便又可靠，而从字符串里提取信息，麻烦而且容易出错。Scheme（Lisp）语言里面大量使用结构化数据，少用字符串，这是 Lisp 系统比 Unix 系统先进的地方之一。

从计算理论的角度讲，每个程序都是一台机器的“描述”，而解释器就是在“模拟”这台机器的运转，也就是在进行“计算”。所以从某种意义上讲，解释器就是计算的本质。当然，不同的解释器就会带来不同的计算。

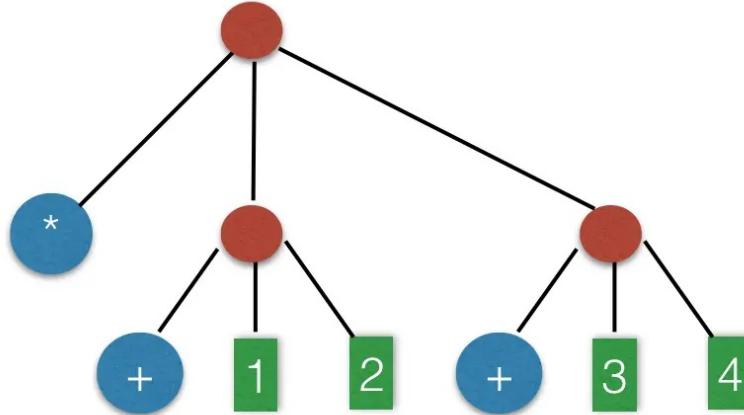
CPU 也是一个解释器，它专门解释执行机器语言。如果你深刻理解了解释器，就可以从本质上看出各种 CPU 的设计为什么是那个样子，它们有什么优缺点，而不只是被动的作为它们的使用者。

抽象语法树（Abstract Syntax Tree）

用 S 表达式所表示的代码，本质上是一种叫做“树”（tree）的数据结构。更具体一点，这叫做“抽象语法树”（Abstract Syntax Tree，简称 AST）。下文为了简洁，我们省略掉“抽象”两个字，就叫它“语法树”。

跟普通的树结构一样，语法树里的节点，要么是一个“叶节点”，要么是一颗“子树”。叶节点是不能再细分的“原子”，比如数字，字符串，操作符，变量名。而子树是可以再细分的“结构”，比如算术表达式，函数定义，函数调用，等等。

举个简单的例子，表达式 `'(* (+ 1 2) (+ 3 4))`，就对应如下的语法树结构：



其中，`*`，两个`+`，`1`，`2`，`3`，`4`都是叶节点，而那三个红色节点，都表示子树结构：`'(+ 1 2)`，`'(+ 3 4)`，`'(* (+ 1 2) (+ 3 4))`。

树遍历算法

在基础的数据结构课程里，我们都学过二叉树的遍历操作，也就是所谓先序遍历，中序遍历和后序遍历。语法树跟二叉树，其实没有很大区别，所以你也可以在它上面进行遍历。解释器的算法，就是在语法树上的一种遍历操作。由于这个渊源关系，我们先来做一个遍历二叉树的练习。做好了之后，我们就可以把这段代码扩展成一个解释器。

这个练习是这样：写出一个函数，名叫`tree-sum`，它对二叉树进行“求和”，把所有节点里的数加在一起，返回它们的和。举个例子，`(tree-sum '((1 2) (3 4)))`，执行后应该返回 10。注意：这是一颗二叉树，所以不会含有长度超过 2 的子树，你不需要考虑像 `((1 2) (3 4 5))` 这类情况。需要考虑的例子是像这样：`(1 2)`，`(1 (2 3))`，`((1 2) 3)`，`((1 2) (3 4))`，……

（为了达到最好的学习效果，你最好试一下写出这个函数再继续往下看。）

好了，希望你得到了跟我差不多的结果。我的代码是这个样子：

```
#lang racket

(define tree-sum
  (lambda (exp)
    (match exp
      [(? number? x) x] ; 对输入exp进行模式匹配
      [(_ ,e1 ,e2) ; exp是一个数x吗？如果是，那么返回这个数x
       (let ([v1 (tree-sum e1)] ; exp是一个含有两棵子树的中间节点吗？
            [v2 (tree-sum e2)]) ; 递归调用tree-sum自己，对左子树e1求值
        (+ v1 v2))])) ; 递归调用tree-sum自己，对右子树e2求值
      ; 返回左右子树结果v1和v2的和
      ))
```

你可以通过以下的例子来测试它的正确性：

```
(tree-sum '(1 2))
;; => 3
(tree-sum '(1 (2 3)))
;; => 6
(tree-sum '((1 2) 3))
;; => 6
(tree-sum '((1 2) (3 4)))
;; => 10
```

（完整的代码示例，可以在[这里下载](#)。）

这个算法很简单，我们可以把它用文字描述如下：

1. 如果输入 `exp` 是一个数，那就返回这个数。
2. 否则如果 `exp` 是像 `(,e1 ,e2)` 这样的子树，那么分别对 `e1` 和 `e2` 递归调用 `tree-sum`，进行求和，得到 `v1` 和 `v2`，然后返回 `v1 + v2` 的和。

你自己写出来的代码，也许用了 `if` 或者 `cond` 语句来进行分支，而我的代码里面使用的是 Racket 的模式匹配 (`match`)。这个例子用 `if` 或者 `cond` 其实也可以，但我之后要把这代码扩展成一个解释器，所以提前使用了 `match`。这样跟后面的代码对比的时候，就更容易看出规律来。接下来，我就简单讲一下这个 `match` 表达式的工作原理。

模式匹配

现在不得不插入一点 Racket 的技术细节，如果你已经学会使用 Racket 的模式匹配，可以跳过这一节。你也可以通过阅读 Racket 模式匹配的[文档](#)来代替这一节。但我建议你不要读太多文档，因为我接下去只用到很少的模式匹配功能，我把它们都解释如下。

模式匹配的形式一般是这样：

```
(match x
  [模式 结果]
  [模式 结果]
  ...
  )
```

它先对 `x` 求值，然后根据值的结构来进行分支。每个分支由两部分组成，左边是一个模式，右边是一个结果。整个 `match` 语句的语义是这样：从上到下依次考虑，找到第一个可以匹配 `x` 的值的模式，返回它右边的结果。左边的模式在匹配之后，可能会绑定一些变量，这些变量可以在右边的表达式里使用。

模式匹配是一种分支语句，它在逻辑上就是 Scheme (Lisp) 的 `cond` 表达式，或者 Java 的嵌套条件语句 `if ... else if ... else ...`。然而跟条件语句里的“条件”不同，每条 `match` 语句左边的模式，可以准确而形象地描述数据结构的形状，而且可以在匹配的同时，对结构里的成员进行“绑定”。这样我们可以在右边方便的访问结构成员，而不需要使用访问函数 (`accessor`) 或者 `foo.x` 这样的属性语法 (`attribute`)。而且模式可以有嵌套的子结构，所以它能够一次性的表示复杂的数据结构。

举个实在点的例子。我的代码里用了这样一个 `match` 表达式：

```
(match exp
  [(? number? x) x]
  [(_ ,e1 ,e2)
   (let ([v1 (tree-sum e1)]
        [v2 (tree-sum e2)])
     (+ v1 v2))])
```

第二行里面的 `'(,e1 ,e2)` 是一个模式 (pattern) , 它被用来匹配 `exp` 的值。如果 `exp` 是 `'(1 2)`, 那么它与 `'(,e1 ,e2)` 匹配的时候, 就会把 `e1` 绑定到 `'1`, 把 `e2` 绑定到 `'2`。这是因为它们结构相同:

```
'(,e1 ,e2)
'('1 '2)
```

说白了, 模式就是一个可以含有“名字” (像 `e1` 和 `e2`) 的结构, 像 `'(,e1 ,e2)`。我们拿这个带有名字的结构, 去匹配实际数据, 像 `'(1 2)`。当它们一一对应之后, 这些名字就被绑定到数据里对应位置的值。

第一行的“模式”比较特殊, `(? number? x)` 表示的, 其实是一个普通的条件判断, 相当于 `(number? exp)`, 如果这个条件成立, 那么它把 `exp` 的值绑定到 `x`, 这样右边就可以用 `x` 来指代 `exp`。对于无法细分的结构 (比如数字, 布尔值), 你只能用这种方式来“匹配”。看起来有点奇怪, 不过习惯了就好了。

模式匹配对解释器和编译器的书写相当有用, 因为程序的语法树往往具有嵌套的结构。不用模式匹配的话, 往往要冗长, 复杂, 不直观的代码, 才能描述出期望的结构。而且由于结构的嵌套比较深, 很容易漏掉边界情况, 造成错误。模式匹配可以直观的描述期望的结构, 避免漏掉边界情况, 而且可以方便的访问结构成员。

由于这个原因, 很多源于 ML 的语言 (比如 OCaml, Haskell) 都有模式匹配的功能。因为 ML (Meta-Language) 原来设计的用途, 就是用来实现程序语言的。Racket 的模式匹配也是部分受了 ML 的启发, 实际上它们的原理是一模一样的。

好了, 树遍历的练习就做到这里。然而这跟解释器有什么关系呢? 下面我们只把它改一下, 就可以得到一个简单的解释器。

一个计算器

计算器也是一种解释器, 只不过它只能处理算术表达式。我们的下一个目标, 就是写出一个计算器。如果你给它 `'(* (+ 1 2) (+ 3 4))`, 它就输出 `21`。可不要小看这个计算器, 稍后我们把它稍加改造, 就可以得到一个更多功能的解释器。

上面的代码里, 我们利用递归遍历, 对树里的数字求和。那段代码里, 其实已经隐藏了一个解释器的框架。你观察一下, 一个算术表达式 `'(* (+ 1 2) (+ 3 4))`, 跟二叉树 `'((1 2) (3 4))` 有什么不同? 发现没有, 这个算术表达式比起二叉树, 只不过在每个子树结构里多出了一个操作符: 一个 `*` 和两个 `+`。它不再是一棵二叉树, 而是一种更通用的树结构。

这点区别, 也就带来了二叉树求和与解释器算法的区别。对二叉树进行求和的时候, 在每个子树节点, 我们都做加法。而对表达式进行解释的时候, 在每一个子树节点, 我们不一定进行加法。根据子树的“操作符”不同, 我们可能会选择加, 减, 乘, 除四种操作。

好了, 下面就是这个计算器的代码。它接受一个表达式, 输出一个数字作为结果。

```
#lang racket ; 声明用 Racket 语言

(define calc
  (lambda (exp)
    (match exp
      [(? number? x) ; 分支匹配: 表达式的两种情况
       ; 是数字, 直接返回
       x]
      [(_ ,op ,e1 ,e2) ; 匹配提取操作符op和两个操作数e1,e2
       (let ([v1 (calc e1)] ; 递归调用 calc 自己, 得到 e1 的值
             [v2 (calc e2)]) ; 递归调用 calc 自己, 得到 e2 的值
        (match op
          ['+ (+ v1 v2)] ; 分支匹配: 操作符 op 的 4 种情况
          ['-' (- v1 v2)]
          ['*' (* v1 v2)]
          ['/' (/ v1 v2)]))))]))
```

你可以得到如下的结果:

```
(calc '+ 1 2)
;; => 3
(calc '* 2 3)
;; => 6
(calc '* (+ 1 2) (+ 3 4))
;; => 21
```

(完整的代码和示例, 可以在[这里下载](#)。)

跟之前的二叉树求和代码比较一下, 你会发现它们惊人的相似, 因为解释器本来就是一个树遍历算法。不过你发现它们有什么不同吗? 它们的不同点在于:

1. 算术表达式的模式里面, 多出了一个“操作符” (`op`) 叶节点: `(,op ,e1 ,e2)`
2. 对子树 `e1` 和 `e2` 分别求值之后, 我们不是返回 `(+ v1 v2)`, 而是根据 `op` 的不同, 返回不同的结果:

```
(match op
  ['+ (+ v1 v2)]
  ['- (- v1 v2)]
  ['*' (* v1 v2)]
  ['/' (/ v1 v2)])
```

最后你发现, 一个算术表达式的解释器, 不过是一个稍加扩展的树遍历算法。

R2 : 一个很小的程序语言

实现了一个计算器, 现在让我们过渡到一种更强大的语言。为了方便称呼, 我给它起了一个萌萌哒名字, 叫 R2。R2 比起之前的计算器, 只多出四个元素, 它们分别是: 变量, 函数, 绑定, 调用。再加上之前介绍的算术操作, 我们就得到一个很简单的程序语言, 它只有5种不同的构造。用 Scheme 的语法, 这5种构造看起来就像这样:

- 变量: `x`
- 函数: `(lambda (x) e)`
- 绑定: `(let ([x e1]) e2)`
- 调用: `(e1 e2)`
- 算术: `(* e2 e2)`

(其中, • 是一个算术操作符, 可以选择 `+, -, *, /` 其中之一)

一般程序语言还有很多其它构造, 可是一开头就试图去实现所有那些, 只会让人糊涂。最好是把这少数几个东西搞清楚, 确保它们正确之后, 才慢慢加入其它元素。

这些构造的语义, 跟 Scheme 里面的同名构造几乎一模一样。如果你不清楚什么是“绑定”, 那你可以把它看成是普通语言里的“变量声明”。

需要注意的是, 跟一般语言不同, 我们的函数只接受一个参数。这不是一个严重的限制, 因为在我们的语言里, 函数可以被作为值传递, 也就是所谓“first-class function”。所以你可以用嵌套的函数定义来表示有两个以上参数的函数。

举个例子, `((lambda (x) (lambda (y) (+ x y)))` 是个嵌套的函数定义, 它也可以被看成是两个参数 (`x` 和 `y`) 的函数, 这个函数返回 `x` 和 `y` 的和。当这样的函数被调用的时候, 需要两层调用, 就像这样:

```
((((lambda (x)
         (lambda (y) (+ x y)))
        1)
      2)
;; => 3
```

这种做法在PL术语里面, 叫做咖喱 (currying)。看起来啰嗦, 但这样我们的解释器可以很简单。等我们理解了基本的解释器, 再实现真正的多参数函数也不迟。

另外, 我们的绑定语法 `(let ([x e1]) e2)`, 比起 Scheme 的绑定也有一些局限。我们的 `let` 只能绑定一个变量, 而 Scheme 可以绑定多个, 像这样 `(let ([x 1] [y 2]) (+ x y))`。这也不是一个严重的限制, 因为我们可以啰嗦一点, 用嵌套的 `let` 绑定:

```
(let ([x 1])
  (let ([y 2])
    (+ x y)))
```

R2 的解释器

下面是我们今天要完成的解释器，它可以运行一个 R2 程序。你可以先留意一下各部分的注释。

```
#lang racket

;; 以下三个定义 env0, ext-env, lookup 是对环境 (environment) 的基本操作：

;; 空环境
(define env0 '())

;; 扩展。对环境 env 进行扩展，把 x 映射到 v，得到一个新的环境
(define ext-env
  (lambda (x v env)
    (cons `(,x . ,v) env)))

;; 查找。在环境中 env 中查找 x 的值。如果没找到就返回 #f
(define lookup
  (lambda (x env)
    (let ((p (assq x env)))
      (cond
        [(not p) #f]
        [else (cdr p)])))

;; 闭包的数据结构定义，包含一个函数定义 f 和它定义时所在的环境
(struct Closure (f env))

;; 解释器的递归定义（接受两个参数，表达式 exp 和环境 env）
;; 共 5 种情况（变量，函数，绑定，调用，数字，算术表达式）
(define interp
  (lambda (exp env)
    (match exp
      [(symbol? x)
       (let ([v (lookup x env)])
         (cond
           [(not v)
            (error "undefined variable" x)]
           [else v]))]
      [(? number? x)
       `,(lambda (,x) ,e)]
      [(Closure exp env)]
      [`(let ([,x ,e1]) ,e2)
       (let ([v1 (interp e1 env)])
         (interp e2 (ext-env x v1 env)))]
      [`(,e1 ,e2)
       (let ([v1 (interp e1 env)]
             [v2 (interp e2 env)])
         (match v1
           [(Closure `(lambda (,x) ,e) env-save)
            (interp e (ext-env x v2 env-save))])
           [(`,op ,e1 ,e2)
            (let ([v1 (interp e1 env)]
                  [v2 (interp e2 env)])
              (match op
                [`(+ (+ ,v1 ,v2))
                 `(- (- ,v1 ,v2))
                 `(* (* ,v1 ,v2))
                 `/ (/ ,v1 ,v2))]))])
      ;; 解释器的“用户界面”函数。它把 interp 包装起来，掩盖第二个参数，初始值为 env0
      (define r2
        (lambda (exp)
          (interp exp env0))))
```

这里有一些测试例子：

```
(r2 '(+ 1 2))
;; => 3

(r2 '(* 2 3))
;; => 6

(r2 '(* 2 (+ 3 4)))
;; => 14

(r2 '(* (+ 1 2) (+ 3 4)))
;; => 21

(r2 `((lambda (x) (* 2 x)) 3))
;; => 6

(r2
`(let ([,x 2])
  (let ([,f (lambda (y) (* ,x y))])
    (,f 3)))
;; => 6
```

（完整的代码和示例，可以在[这里下载](#)。）

在接下来的几节，我们来仔细看看这个解释器的各个部分。

对基本算术操作的解释

算术操作一般都是程序里最基本的构造，它们不能再被细分为多个步骤，所以我们先来看看对算术操作的处理。以下就是 R2 解释器处理算术的部分，它是 interp 的最后一个分支。

```
(match exp
  ... ...
  [`(,op ,e1 ,e2)
   (let ([v1 (interp e1 env)])
     (interp e2 env))
   (match op
     [`(+ (+ ,v1 ,v2))
      `(- (- ,v1 ,v2))
      `(* (* ,v1 ,v2))
      `/ (/ ,v1 ,v2))])]
```

你可以看到它几乎跟刚才写的计算器一模一样，不过现在 interp 的调用多了一个参数 env 而已。这个 env 是所谓“环境”，我们下面很快就讲。

对数字的解释

对数字的解释很简单，把它们原封不动返回就可以了。

```
[(? number? x) x]
```

变量和函数

变量和函数是解释器里最麻烦的部分，所以我们来仔细看看。

变量（variable）的产生，是数学史上的最大突破之一。因为变量可以被绑定到不同的值，从而使函数的实现成为可能。比如数学函数 $f(x) = x * 2$ ，其中 x 是一个变量，它把输入的值传递到函数体 $x * 2$ 里面。如果没有变量，函数就不可能实现。

对变量最基本的操作，是对它的“绑定”（binding）和“取值”（evaluate）。什么是绑定呢？拿上面的函数 $f(x)$ 作为例子。当我们调用 $f(1)$ 时，函数体里面的 x 等于 1，所以 $x * 2$ 的值是 2，而当我们调用 $f(2)$ 时，函数体里面的 x 等于 2，所以 $x * 2$ 的值是 4。这里，两次对 f 的调用，分别对 x 进行了两次绑定。第一次 x 被绑定到了 1，第二次被绑定到了 2。

你可以把“绑定”理解成这样一个动作，就像当你把插头插进电源插座的那一瞬间。插头的插脚就是 $f(x)$ 里面的那个 x ，而 $x * 2$ 里面的 x ，则是电线的另外一端。所以当你把插头插进插座，电流就通过这根电线到达另外一端。如果电线导电性能良好，两头的电压应该相等。

环境

我们的解释器只能一步一步的做事情。比如，当它需要求 $f(1)$ 的值的时候，它分成两步操作：

1. 把 x 绑定到 1，这样函数体内才能看见这个绑定。
2. 进入 f 的函数体，对 $x * 2$ 进行求值。

这就像一个人做出这两个动作：

1. 把插头插进插座。
2. 到电线的另外一头，测量它的电压，并且把结果乘以 2。

在第一步和第二步之间，我们如何记住 x 的值呢？通过所谓“环境”！我们用环境记录变量的值，并且把它们传递到变量的“可见区域”。变量的可见区域，用术语说叫做“作用域”（scope）。

在我们的解释器里，用于处理环境的代码如下：

```
;; 空环境
(define env0 '())

;; 对环境 env 进行扩展，把 x 映射到 v
(define ext-env
  (lambda (x v env)
    (cons `(,x . ,v) env)))

;; 取值。在环境中 env 中查找 x 的值
(define lookup
  (lambda (x env)
    (let ((p (assq x env)))
      (cond
        [(not p) #f]
        [else (cdr p)])))
```

这里我们用一种最简单的数据结构，Scheme 的 association list，来表示环境。Association list 看起来像这个样子： $((x . 1) (y . 2) (z . 5))$ 。它是一个两元组（pair）的链表，左边的元素是 key，右边的元素是 value。写得直观一点就是：

```
((x . 1)
 (y . 2)
 (z . 5))
```

查表操作就是从头到尾搜索，如果左边的 key 是要找的变量，就返回整个 pair。简单吧？效率很低，但是足够完成我们现在的任务。

ext-env 函数扩展一个环境。比如，如果原来的环境 env1 是 $((y . 2) (x . 1))$ 那么 $(ext-env x 3 env1)$ ，就会返回 $((x . 3) (y . 2) (x . 1))$ 。也就是把 $(x . 3)$ 加到 env1 的最前面去。

那我们什么时候需要扩展环境呢？当我们进行绑定的时候。绑定可能出现在函数调用时，也可能出现在 let 绑定时。我们选择的数据结构，使得环境自然而然的具有了作用域（scope）的特性。

环境其实是一个堆栈（stack）。内层的绑定，会出现在环境的最上面，这就是在“压栈”。这样我们查找变量的时候，会优先找到最内层定义的变量。

举个例子：

```
(let ([x 1]) ; env='(). 绑定x到1。
  (let ([y 2]) ; env='((x . 1)). 绑定y到2。
    (let ([x 3]) ; env='((y . 2) (x . 1)). 绑定x到3。
      (+ x y))) ; env='((x . 3) (y . 2) (x . 1)). 查找x, 得到3; 查找y, 得到2。
;; => 5
```

这段代码会返回 5。这是因为最内层的绑定，把 $(x . 3)$ 放到了环境的最前面，这样查找 x 的时候，我们首先看到 $(x . 3)$ ，然后就返回值 3。之前放进去的 $(x . 1)$ 仍然存在，但是我们先看到了最上面的那个 $(x . 3)$ ，所以它被忽略了。

这并不等于说 $(x . 1)$ 就可以被改写或者丢弃，因为它仍然是有用的。你只需要看一个稍微不同的例子，就知道这是怎么回事：

```
(let ([x 1]) ; env='(). 绑定x到1。
  (+ (let ([x 2]) ; env='((x . 1)). 绑定x到2。
       x) ; env='((x . 2) (x . 1)). 查找x, 得到2。
   x)) ; env='((x . 1)). 查找x, 得到1。
;; => 3 ; 两个不同的x的和, 1+2等于3。
```

这个例子会返回 3。它是第 3 行和第 4 行里面两个 x 的和。由于第 3 行的 x 处于内层 let 里面，那里的环境是 $((x . 2) (x . 1))$ ，所以查找 x 的值得到 2。第 4 行的 x 在内层 let 外面，但是在外层 let 里面，那里的环境是 $((x . 1))$ ，所以查找 x 的值得到 1。这很符合直觉，因为 x 总是找到最内层的定义。

值得注意的是，环境被扩展以后，形成了一个新的环境，而原来的环境并没有被改变。比如，上面的 $((y . 2) (x . 1))$ 并没有删除或者修改，只不过是被“引用”到一个更大的列表里去了。

这样不对已有数据进行修改（mutation）的数据结构，叫做“函数式数据结构”。函数式数据结构只生成新的数据，而不改变或者删除老的。它可能引用老的结构，然而却不改变老的结构。这种“不修改”（immutable）的性质，在我们的解释器里是很重要的，因为当我们扩展一个环境，进入递归，返回之后，外层的代码必须仍然可以访问原来外层的环境。

当然，我们也可以用另外的，更高效的数据结构（比如平衡树，串接起来的哈希表）来表示环境。如果你学究一点，甚至可以用函数来表示环境。这里为了代码简单，我们选择了最笨，然而正确，容易理解的数据结构。

对变量的解释

了解了变量，函数和环境，我们来看看解释器对变量的“取值”操作，也就是 match 的第一种情况。

```
[(? symbol? x) (lookup x env)]
```

这就是在环境中，沿着从内向外的“作用域顺序”，查找变量的值。

这里的 $(? symbol? x)$ 是一种特殊的模式，它使用 Scheme 函数 `symbol?` 来判断输入是否是一个符号，如果是，就把它绑定到 x ，然后你就可以在右边用 x 来指称这个输入。

对绑定的解释

现在我们来看看对 let 绑定的解释：

```
[`(let ([,x ,e1]),e2)
(let ([v1 (interp e1 env)])
  (interp e2 (ext-env x v1 env)))]
```

; 解释右边表达式e1，得到值v1
; 把(x . v1)扩充到环境顶部，对e2求值

通过代码里的注释，你也许已经可以理解它在做什么。我们先对表达式 e1 求值，得到 v1。然后我们把 (x . v1) 扩充到环境里，这样 (let ([x e1]) ...) 内部都可以看到 x 的值。然后我们使用这个扩充后的环境，递归调用解释器本身，对 let 的主体 e2 求值。它的返回值就是这个 let 绑定的值。

Lexical Scoping 和 Dynamic Scoping

下面我们准备谈谈函数定义和调用。对函数的解释是一个微妙的问题，很容易弄错，这是由于函数体内也许会含有外层的变量，叫做“自由变量”。所以在分析函数的代码之前，我们来了解一下不同的“作用域”（scoping）规则。

我们举个例子来解释这个问题。下面这段代码，它的值应该是多少呢？

```
(let ([x 2])
  (let ([f (lambda (y) (* x y))])
    (let ([x 4])
      (f 3))))
```

在这里，f 函数体 (lambda (y) (* x y)) 里的那个 x，就是一个“自由变量”。x 并不是这个函数的参数，也不是在这个函数里面定义的，所以我们必须到函数外面去找 x 的值。

我们的代码里面，有两个地方对 x 进行了绑定，一个等于 2，一个等于 4，那么 x 到底应该是指向哪一个绑定呢？这似乎无关痛痒，然而当我们调用 (f 3) 的时候，严重的问题来了。f 的函数体是 (* x y)，我们知道 y 的值来自参数 3，可是 x 的值是多少呢？它应该是 2，还是 4 呢？

在历史上，这段代码可能有两种不同的结果，这种区别一直延续到今天。如果你在 Scheme（Racket）里面写以上的代码，它的结果是 6。

```
; ; Scheme
(let ([x 2])
  (let ([f (lambda (y) (* x y))])
    (let ([x 4])
      (f 3))))
```

; ; => 6

现在我们来看看，在 Emacs Lisp 里面输入等价的代码，得到什么结果。如果你不熟悉 Emacs Lisp 的用法，那你可以跟我做：把代码输入 Emacs 的那个叫 *scratch* 的 buffer，把光标放在代码最后，然后按 C-x C-e，这样 Emacs 会执行这段代码，然后在 minibuffer 里显示结果：

The screenshot shows the Emacs minibuffer with the following text:
(let ((x 2))
 (let ((f (lambda (y) (* x y))))
 (let ((x 4))
 (funcall f 3))))
U:**- *scratch* All (5,0)
12

结果是 12！如果你把代码最内层的 x 绑定修成其它的值，输出会随之改变。

奇怪吧？Scheme 和 Emacs Lisp，到底有什么不一样呢？实际上，这两种看似差不多的“Lisp 方言”，采用了两种完全不同的作用域方式。Scheme 的方式叫做 lexical scoping（或者 static scoping），而 Emacs 的方式叫做 dynamic scoping。

那么哪一种方式更好呢？或者用哪一种都无所谓？答案是，dynamic scoping 是非常错误的做法。历史的教训告诉我们，它会带来许许多多莫名其妙的 bug，导致 dynamic scoping 的语言几乎完全没法用。这是为什么呢？

原因在于，像 (let ((x 4)) ...) 这样的变量绑定，只应该影响它内部“看得见”的 x 的值。当我们看见 (let ((x 4)) (f 3)) 的时候，并没有在 let 的内部看见任何叫“x”的变量，所以我们“直觉”的认为，(let ((x 4)) ...) 对 x 的绑定，不应该引起 (f 3) 的结果变化。

然而对于 dynamic scoping，我们的直觉却是错误的。因为 f 的函数体里面有一个 x，虽然我们没有在 (f 3) 这个调用里面看见它，然而它却存在于 f 定义的地方。要知道，f 定义的地方也许隔着几百行代码，甚至在另外一个文件里面。而且调用函数的人凭什么应该知道，f 的定义里面有一个自由变量，它的名字叫做 x？所以 dynamic scoping 在设计学的角度来看，是一个反人类的设计！

相反，lexical scoping 却是符合人们直觉的。虽然在 (let ((x 4)) (f 3)) 里面，我们把 x 绑定到了 4，然而 f 的函数体并不是在那里定义的，我们也没在那里看见任何 x，所以 f 的函数体里面的 x，仍然指向我们定义它的时候看得见的那个 x，也就是最上面的那个 (let ([x 2]) ...)，它的值是 2。所以 (f 3) 的值应该等于 6，而不是 12。

对函数的解释

为了实现 lexical scoping，我们必须把函数做成“闭包”（closure）。闭包是一种特殊的数据结构，它由两个元素组成：函数的定义和当前的环境。我们把闭包定义为一个 Racket 的 struct 结构：

```
(struct Closure (f env))
```

有了这个数据结构，我们对 (lambda (x) e) 的解释就可以写成这样：

```
[`(lambda (,x) ,e)
(Closure exp env)]
```

注意这里的 exp 就是 ``(lambda (,x) ,e)` 自己。

有意思的是，我们的解释器遇到 (lambda (x) e)，几乎没有做任何计算。它只是把这个函数包装了一下，把它与当前的环境一起，打包放到一个数据结构（Closure）里面。这个闭包结构，记录了我们在函数定义的位置“看得见”的那个环境。稍候在调用的时候，我们就能从这个闭包的环境里面，得到函数体内的自由变量的值。

对调用的解释

好了，我们终于到了最后的关头，函数调用。为了直观，我们把函数调用的代码拷贝如下：

```
[`(,e1 ,e2)
(let ([v1 (interp e1 env)]) ; 计算函数 e1 的值
  [v2 (interp e2 env)]) ; 计算参数 e2 的值
  (match v1
    [(Closure `(lambda (,x) ,e) env-save) ; 用模式匹配的方式取出闭包里的各个子结构
     (interp e (ext-env x v2 env-save))]))] ; 在闭包的环境env-save中把x绑定到v2，解释函数体
```

函数调用都是 (e1 e2) 这样的形式，e1 表示函数，e2 是它的参数。我们需要先分别求出函数 e1 和参数 e2 的值。

函数调用就像把一个电器的插头插进插座，使它开始运转。比如，当 (lambda (x) (* x 2)) 被作用于 1 时，我们把 x 绑定到 1，然后解释它的函数体 (* x 2)。但是这里有一个问题，函数体内的自由变量应该取什么值呢？从上面闭包的讨论，你已经知道了，自由变量的值，应该从闭包的环境查询。

操作数 e1 的值 v1 是一个闭包，它里面包含一个函数定义时保存的环境 env-save。我们把这个环境 env-save 取出来，那我们就可以查询它，得到函数体内自由变量的值。然而函数体内不仅有自由变量，还有对函数参数的使用，所以我们必须扩展这个 env-save 环境，把参数的值加进去。这就是为什么我们使用 (ext-env x v2 env-save)，而不只是 env-save。

你可能会奇怪，那么解释器的环境 env 难道这里就不用了吗？是的。我们通过 env 来计算 e1 和 e2 的值，是因为 e1 和 e2 里面的变量，在“当前环境”(env) 里面看得见。可是函数体的定义，在当前环境下是看不见的。它的代码在别的地方，而那个地方看得见的环境，被我们存在闭包里了，它就是 env-save。所以我们把 v1 里面的闭包环境 env-save 取出来，用于计算函数体的值。

有意思的是，如果我们用 env，而不是 env-save 来解释函数体，那我们的语言就变成了 dynamic scoping。现在来实验一下：你可以把 (interp e (ext-env x v2 env-save)) 里面的 env-save 改成 env，再试试我们之前讨论过的代码，它的输出就会变成 12。那就是我们之前讲过的，dynamic scoping 的结果。

```
(r2
'(let ((x 2))
  (let ([f (lambda (y) (* x y))])
    (let ([x 4])
      (f 3))))
;; => 12
```

你也许发现了，如果我们的语言是 dynamic scoping，那就没必要使用闭包了，因为我们根本不需要闭包里面保存的环境。这样一来，dynamic scoping 的解释器就可以写成这样：

```
(define interp
  (lambda (exp env)
    (match exp
      [ ... ]
      [ (lambda (,x) ,e) ; 函数：直接返回自己的表达式
        exp]
      [ (,e1 ,e2)
        (let ([v1 (interp e1 env)])
          [v2 (interp e2 env)])
        (match v1
          [ (lambda (,x) ,e)
            (interp e (ext-env x v2 env))]))]
      [ ... ]
      )))
....
```

注意到这个解释器的函数有多容易实现吗？它就是这个函数的表达式自己，原封不动。用函数的表达式本身来表示它的值，是很直接很简单的做法，也是大部分人一开头就会想到的。然而这样实现出来的语言，就不自觉地采用了 dynamic scoping。

这就是为什么很多早期的 Lisp 语言，比如 Emacs Lisp，都使用 dynamic scoping。这并不是因为它们的设计者在 dynamic scoping 和 lexical scoping 两者之中做出了选择，而是因为使用函数的表达式本身来作为它的值，是最直接，一般人都会首先想到的做法。

另外，在这里我们也看到环境用“函数式数据结构”表示的好处。闭包被调用时它的环境被扩展，但是这并不会影响原来的那个环境，我们得到的是一个新的环境。所以当函数调用返回之后，函数的参数绑定就自动“注销”了。

如果你用一个非函数式的数据结构，在绑定参数时不生成新的环境，而是对已有环境进行赋值，那么这个赋值操作就会永久性的改变原来环境的内容。所以你在函数返回之后必须删除参数的绑定。这样不但麻烦，而且在复杂的情况下很容易出错。

思考题：可能有些人看过 lambda calculus，这些人可能知道 (let ((x e1)) e2) 其实等价于一个函数调用：((lambda (x) e2) e1)。现在问题来了，我们在讨论函数和调用的时候，很深入的讨论了关于 lexical scoping 和 dynamic scoping 的差别。既然 let 绑定等价于一个函数定义和调用，为什么之前我们讨论对绑定的时候，没有讨论过 lexical scoping 和 dynamic scoping 的问题，也没有制造过闭包呢？

不足之处

现在你已经学会了如何写出一个简单的解释器，它可以处理一个相当强大的函数式语言。出于教学的考虑，这个解释器并没有考虑实用的需求，所以它并不能作为工业应用。在这里，我指出它的一些不足之处。

1. 缺少必要的语言构造。我们的语言里缺少好些实用语言必须的构造：递归，数组，赋值操作，字符串，自定义数据结构，…… 作为一篇基础性的读物，我不能把这些都加进来。如果你对这些有兴趣，可以看看其它书籍，或者等待我的后续作品。
2. 不合法代码的检测和报告。你也许发现了，这个解释器的 match 表达式，全都假定了输入都是合法的程序，它并没有检查不合法的情况。如果你给它一个不合法的程序，它不会马上报错，而是会真去算它，以至于导致奇怪的后果。一个实用的解释器，必须加入对代码格式进行全面检测，在运行之前就报告不合法的代码结构。
3. 低效率的数据结构。在 association list 里面查找变量，是线性的复杂度。当程序有很多变量的时候就有性能问题。一个实用的解释器，需要更高效的数据结构。这种数据结构不一定非得是函数式的。你也可以用非函数式的数据结构（比如哈希表），经过一定的改造，达到同样的性质，却具有更高的效率。另外，你还可以把环境转化成一个数组。给环境里的每个变量分配一个下标（index），在这个数组里就可以找到它的值。如果你用数组表示环境，那么这个解释器就向编译器迈进了一步。
4. S 表达式的歧义问题。为了教学需要，我们的解释器直接使用 S 表达式来表达语法树，用模式匹配来进行分支遍历。在实际的语言里，这种方式会带来比较大的问题。因为 S 表达式是一种通用的数据结构，用它表示的东西，看起来都差不多的样子。一旦程序的语法构造多起来，直接对 S 表达式进行模式匹配，会造成歧义。

比如 (,op ,e1 ,e2)，你以为它只匹配二元算术操作，比如 (+ 1 2)。但它其实也可以匹配一个 let 绑定：(let ([x 1]) (* x 2))。这是因为它们顶层元素的数目是一样的。为了消除歧义，你得小心的安排模式的顺序，比如你必须把 (let ([,x ,e1]) ,e2) 的模式放在 (,op ,e1 ,e2) 前面。所以最好的办法，是不要直接在 S 表达式上写解释器，而是先写一个“parser”，这个 parser 把 S 表达式转换成 Racket 的 struct 结构。然后解释器再在 struct 上面进行分支匹配。这样解释器不用担心歧义问题，而且会带来效率的提升。

付费方式

如果你喜欢这篇文章，可以到[这里付费购买](#)。

什么是语义学



很多人问我如何在掌握基本的程序语言技能之后进入“语义学”的学习。现在我就简单介绍一下什么是“语义”，然后推荐一本入门的书。这里我说的“语义”主要是针对程序语言，不过自然语言里的语义，其实本质上也是一样的。

一个程序的“语义”通常是由另一个程序决定的，这另一个程序叫做“解释器”(interpreter)。程序只是一个数据结构，通常表示为语法树(abstract syntax tree)或者指令序列。这个数据结构本身其实没有意义，是解释器让它产生了意义。对同一个程序可以有不同的解释，就像上面这幅图，对画面元素的不同解释，可以看到不同的内容（少女或者老妇）。

解释器接受一个“程序”(program)，输出一个“值”(value)。用图形的方法表示，解释器看起来就像一个箭头：程序 \Rightarrow 值。这个所谓的“值”可以具有非常广泛的含义。它可能是一个整数，一个字符串，也有可能是更加奇妙的东西。

其实解释器不止存在于计算机中，它是一个很广泛的概念。其中好些你可能还没有意识到。写 Python 程序，需要 Python 解释器，它的输入是 Python 代码，输出是一个 Python 里面的数据，比如 42 或者“foo”。CPU 其实也是一个解释器，它的输入是以二进制表示的机器指令，输出是一些电信号。人脑也是一个解释器，它的输入是图像或者声音，输出是神经元之间产生的“概念”。如果你了解类型推导系统 (type inference)，就会发现类型推导的过程也是一个解释器，它的输入是一个程序，输出是一个“类型”。类型也是一种值，不过它是一种抽象的值。比如，42 对应的类型是 int，我们说 42 被抽象为 int。

所以“语义学”，基本上就是研究各种解释器。解释器的原理其实很简单，但是结构非常精巧微妙，如果你从复杂的语言入手，恐怕永远也学不会。最好的起步方式是写一个基本的 lambda calculus 的解释器。lambda calculus 只有三种元素，却可以表达所有程序语言的复杂结构。

专门讲语义的书很少，现在推荐一本我觉得深入浅出的：[《Programming Languages and Lambda Calculi》](#)。只需要看完前半部分 (Part I 和 II，100来页) 就可以了。这书好在什么地方呢？它是从非常简单的布尔表达式 (而不是 lambda calculus) 开始讲解什么是递归定义，什么是解释，什么是 Church-Rosser，什么是上下文 (evaluation context)。在让你理解了这种简单语言的语义，有了足够的信心之后，才告诉你更多的东西。比如 lambda calculus 和 CEK，SECD 等抽象机 (abstract machine)。理解了这些概念之后，你就会发现所有的程序语言都可以比较容易的理解了。

GTF - Great Teacher Friedman

写小人书的老顽童

Dan Friedman 是 Indiana 大学的教授，程序语言领域的创始人之一。他主要的著作《The Little Schemer》（前身叫《The Little Lisper》）是程序语言界最具影响力的书籍之一。现在很多程序语言界的元老级人物，当年都是看这本“小人书”学会了 Lisp/Scheme，才决心进入这一领域。



Friedman 对程序语言的理解可以说是世界的最高标准，很可惜的是，由于他个人的低调，他受到很多人的误解。很多人以为他只懂得 Scheme 这种“类型系统落后的语言”。有些人觉得他只顾自己玩，不求“上进”，觉得他的研究闭门造车，不“前沿”。我也误解过他，甚至在见面之前，根据这些书的封面，我断定他肯定是个年轻小伙。结果呢，第一次见到他的时候，他已经过了 60 岁大寿。

程序语言的研究者们往往追逐一些“新概念”，却未能想到很多这些新概念早在几十年前就被 Friedman 想到了。举个例子，Haskell 所用的 lazy evaluation 模型，最早就是他在 1976 年在与 David Wise 合写的论文“CONS should not Evaluate its Arguments”中提出来的。

虽然写了 The Little Schemer，但 Friedman 的学识并不限于 Scheme。他不断地实验各种其它的语言设计，包括像 ML 一类的含有静态类型系统的函数式语言，逻辑式语言，面向对象语言，用于定理证明的语言等等。在每次的试验之后，他几乎都会写一本书，揭示这些语言最精要的部分。

觉得 ML 比 Scheme 先进很多的人们应该看看 Friedman 这本书：The Little MLer：



想要真正理解 Java 设计模式的人可以看看这本：A Little Java, A Few Patterns:



这些东西的优点和弱点仿佛在他心里都有数，他几乎总是指向正确的前进方向。

你知道些什么？

可惜的是，由于个人原因，Friedman 始终没有成为我正式的导师（他“超然物外”，几乎完全不关心自己的学生什么时候能毕业），但他确实是这一生中教会我最多东西的人。所以我想写一些关于他的小故事。也许你能从中看出一个世界级的教育者是什么样子的。我来 IU 之前一位师兄告诉我，Dan Friedman 就像指环王里的甘道夫 (Gandalf)，来了之后发现确实很像。

第一次在办公室见到 Friedman 的时候，他对我说：“来，给我讲讲你知道些什么？”我自豪地说：“我在 Cornell 上过研究生的程序语言课，会用 ML 和 Haskell，看过 Paul Graham 的 On Lisp，Peter Norvig 的 Paradigms of Artificial Intelligence Programming, Richard Gabriel 的一些文章……”他看着我平静地笑了：“不错，你已经有一定基础……”

这么几年以后，我才发现他善良的微笑里面其实隐藏着难以启齿的秘密：当时的我是多么的幼稚。在他的这种循循善诱之下，我才逐渐的明白了，知识的深度是无止境的。他的水平其实远在以上这些人之上，可是出于谦虚，他不能自己把话说出来。

反向运行

Dan Friedman 已经远远超过了退休年龄，却仍然坚持教学。他的本科生程序语言课程 C311 是 IU 的“星级课程”。我最敬佩的，其实是他那孩子般的好奇心和探索精神。几乎每一年的 C311，他都会发明不同的东西来充实课程内容。有时候是一种新的逻辑编程语言（叫 miniKanren），有时候是些小技巧（比如把 Scheme 编译成 C 却不会堆栈溢出），等等。

Friedman 研究一个东西的时候总是全身心的投入，执着的热爱。自从开始设计一个叫 miniKanren 的逻辑编程语言，Friedman 多了一句口头禅：“Does it run backwards?”（能反向运行吗？）因为逻辑式语言（像 Prolog）的程序都是能“反向运行”的。普通程序语言写的程序，如果你给它一个输入，它会给你一个输出。但是逻辑式语言很特别，如果你给它一个输出，它可以反过来运行，给你所有可能的输入。但是 Friedman 真的走火入魔了。不管别人在讲什么，经常最后都会被他问一句：“Does it run backwards?”让你哭笑不得。

Friedman 有一个本领域的人都知道的“弱点”——他不喜欢静态类型系统 (static type system)。其实 Scheme 专家们大部分都不喜欢静态类型系统。为此，他深受“类型专家”们的误解甚至鄙视，可是他都从容对待之。

有一次在他的进阶课程 B621 上，他给我们出了一道题：用 Scheme 实现 ML 和 Haskell 所用的 Hindley-Milner 类型系统。这种类型系统的工作原理一般是，输入一个程序，它经过对程序进行类型推导 (type inference)，输出一个类型。如果程序里有类型错误，它就会报错。由于之前在 Cornell 用 ML 实现过这东西，再加上来到 IU 之后对抽象解释 (abstract interpretation) 的进一步理解，我很快做出了这个东西，而且比在 Cornell 的时候做的还要优雅。

他知道我做出来了，很高兴的样子，让我给全班同学（也就8, 9个人）讲我的做法。当我自豪的讲完，他问：“Does it run backwards? 如果我给它一个类型，它能自动生成出符合这个类型的程序来吗？”我愣了，欲哭无泪啊，“不能……”他往沙发靠背上一躺，得意的笑了：“我的系统可以！哈哈！我当年写的那个类型系统比你这个还要短呢。我早就知道这些类型系统怎么做，可我就是不喜欢。哈哈哈哈……”

我后来对类型系统的进一步研究显示，Hindley-Milner 类型系统确实有很多不必要的问题，才导致了他不喜欢。他就是这样一个老顽童。他喜欢先把你捧上天，再把你打下来，让你知道天外有天 :-)

miniCoq

你永远想象不到 Dan Friedman 的思维的极限在哪里。当你认为他是一个函数式语言专家的时候，他设计了 miniKanren，一种逻辑式编程语言 (logic programming language)，并且写出《The Reasoned Schemer》这样的书，用于教授逻辑编程。当你认为他不懂类型系统的时候，他开始捣鼓当时最热门的 Martin-Löf 类型理论，并且开始设计机器证明系统。而他做这些，完全是出于自己的兴趣。他从来不在乎别人在这个方向已经做到了什么程度，却经常能出乎意料的简化别人的设计。

有一次系里举办教授们的“闪电式演讲”(lightening talk)，每位教授只有5分钟时间上去介绍自己的研究。轮到 Friedman 的时候，他慢条斯理的走上去，说：“我不着急。我只有几句话要说。我不知道我能不能拖够5分钟……”大家都笑了。他接着说：“我现在最喜欢的东西是 Curry-Howard correspondence 和定理证明。我觉得现在的机器证明系统太复杂了，比如 Coq 有 nnnnn 行代码。我想在 x 年之内，简化 Coq，得到一个 miniCoq……”

miniCoq... 听到这个词全场都笑翻了。为什么呢？自己去联想吧。从此，“Dan Friedman 的 miniCoq”成为了 IU 的程序语言学生茶余饭后的笑话。

但是 Firedman 没有吹牛。他总是说到做到。他已经写出一个简单的定理证明工具叫 JBob (迫于社会舆论压力，不能叫 miniCoq) ，而且正在写一本书叫《The Little Prover》，用来教授最重要的定理证明思想。他开始在 C311 上给本科生教授这些内容。我看了那本书的初稿，获益至深，那是很多 Coq 的教材都不涉及的最精华的道理。它不仅教会我如何使用定理证明系统，而且教会了我如何设计一个定理证明系统。我对他说：“你总是有新的东西教给我们。每隔两年，我们就得重新上一次你的课！”

C311

当我刚从 Cornell 转学到 IU 的时候，Dan Friedman 叫我去上他的研究生程序语言课 B521。我当时以自己在 Cornell 上过程序语言课程为由，想不去上他的课。Friedman 把我叫到他的办公室，和蔼的对我说：“王垠，我知道你在 Cornell 上过这种课。我也知道 Cornell 是比 IU 好很多的学校。可是每个老师的教学方法都是不一样的，你应该来上我的课。我和我的朋友们在这里做教授，不是因为喜欢这个学校，而是因为我们的家人和朋友都在这里。”后来由于跟 Amr Sabry (我现在的导师) 的课程 B522 时间重合，他特别安排我坐在本科生的 C311 的课堂上，却拿研究生课程的学分。后来发现，这两门课的内容基本没有区别，只不过研究生的作业要多一些。

在第一堂课上，他说了一句让我记忆至今的话：“《The Little Schemer》和《Essentials of Programming Languages》是这门课的参考教材，但是我上课从来不讲我的书里的内容。”刚一开始，我就发现这门课跟我在 Cornell 学到的东西很不一样。虽然有些概念，比如 closure，CPS，我在 Cornell 都学过，在他的课堂上，我却看到这些概念完全不同的一面，以至于我觉得其实我之前完全不懂这些概念！这是因为在 Cornell 学到这些东西的时候只是用来应付作业，而在 Friedman 的课上，我利用它们来完成有实际意义的目标，所以才真正的体会到这些概念的内涵和价值。

一个例子就是课程进入到没几个星期的时候，我们开始写解释器来执行简单的 Scheme 程序。然后我们把这个解释器进行 CPS 变换，引入全局变量作为“寄存器”(register)，把 CPS 产生的 continuation 转换成数据结构（也就是堆栈）。最后我们得到的是一个抽象机 (abstract machine)，而这在本质上相当于一个真实机器里的中央处理器 (CPU) 或者虚拟机 (比如 JVM)。所以我们其实从无到有，“发明”了 CPU！从这里，我才真正的理解到寄存器，堆栈等的本质，以及我们为什么需要它们。我才真正的明白了，冯诺依曼体系构架为什么要设计成这个样子。后来他让我们去看一篇他的好朋友 Olivier Danvy 的论文，讲述如何从各种不同的解释器经过 CPS 变换得出不同种类的抽象机模型。这是我第一次感觉到程序语言的理论对于现实世界的巨大威力，也让我理解到，机器并不是计算的本质。机器可以用任何可行的技术实现，比如集成电路，激光，分子，DNA…… 但是无论用什么作为机器的材料，我们所要表达的语义，也就是计算的本质，却是不变的。

而这些还不是我那届 C311 全部的内容。后半学期，我们开始学习 miniKanren，一种他自己设计的用于教学的逻辑式语言 (logic programming language)。这个语言类似 Prolog，但是它把 Prolog 的很多缺点给去掉了，而且变得更加容易理解。教材是免费送给我们的《The Reasoned Schemer》。在书的最后，两页纸的篇幅，就是整个 miniKanren 语言的实现！我学得比较快，后来就开始捣鼓这个实现，把有些部分重新设计了一下，然后加入了一些我想要的功能。这样的教学，给了我设计逻辑式语言的能力，而不只是停留在一个使用者。这是学习 Prolog 不可能做到的事情，因为 Prolog 的复杂性会让初学者无从下手，只能停留在使用者的阶段。

我很幸运当初听了他的话去上了这门课，否则我就不会是今天的我。

独立思维

Dan Friedman 是一个不随波逐流，有独立思想的人。他的眼里容不下过于复杂的东西，他喜欢把一个东西简化到容得进几行程序，把相关的问题理解得非常清楚。他的书是一种独特的“问答式”的结构，很像孔夫子或者苏格拉底的讲学方式。他的教学方式也非常独特。这在本科生课程 C311 里已经有一些表现，但是在研究生的课程 B621 里，才全部的显示出来。

我写过的最满意的一个程序，自动 CPS 变换，就是在 C311 产生的。在 C311 的作业里，Friedman 经常加入一些“智力题”(brain teaser)，做出来了可以加分。因为我已经有了一定基础，所以我有精力来做那些智力题。开头那些题还不是很难，直到开始学 CPS 的时候，出现了这么一道：“请写出一个叫 CPSer 的程序，它的作用是自动的把 Scheme 程序转换成 CPS 形式。”那次作业的其它题目都是要求“手动”把程序变成 CPS 形式，这道智力题却要求一个“自动”的——用一个程序来转换另一个程序。

我觉得很有意思。如果能写出一个自动的 CPS 转换程序，那我岂不是可以用它完成所有其它的题目了！所以我就开始捣鼓这个东西，最初的想法其实就是“模拟”一个手动转换的过程。然后我发现这真是个怪物，就那么几十行程序，不是这里不对劲，就是那里不对劲。这里按下去一个 bug，那里又冒出来一个，从来没见过这么麻烦的东西。我就跟它死磕了，废寝忘食几乎一星期。经常走进死胡同，就只有重新开始，不知道推翻重来了多少次。到快要交作业的时候，我把它给弄出来了。最后我用它生成了所有其它的答案，产生的 CPS 代码跟手工转换出来的看不出任何区别。当

然我这次我又得了满分（因为每次都做智力题，我的分数总是在100以上）。

作业发下来那天下课后，我跟 Friedman 一起走回 Lindley Hall (IU 计算机系的楼)。半路上他问我：“这次的 brain teaser 做了没有。”我说：“做了。这是个好东西啊，帮我把其它作业都做出来了。”他有点吃惊，又有点将信将疑的样子：“你确信你做对了？”我说：“不确定它是完全正确，但是转换后的作业程序全都跟手工做的一样。”走回办公室之后，他给了我一篇30多页的论文 “Representing control: a study of the CPS transformation”，作者是他的好朋友 Olivier Danvy 和 Andrzej Filinski。然后我才了解到，这是这个方向最厉害的两个人。正是这篇论文，解决了这个悬而不决十多年的难题。其实自动的 CPS 转换，可以被用于实现高效的函数式语言编译器。Princeton 大学的著名教授 Andrew Appel 写了一本书叫《Compiling with Continuations》，就是专门讲这个问题的。Amr Sabry (我现在的导师) 当年的博士论文就是一个比 CPS 还要简单的变换 (叫做 ANF)。凭这个东西，他几乎灭掉了这整个 CPS 领域，并且拿到了终身教授职位。在他的论文发表10年之内也没有 CPS 的论文出现。

Friedman 啊，把这样一个问题作为“智力题”，真有你的！我开玩笑地对他说：“我保证，我不会把这个程序开源，不然以后你的 C311 学生们就可以拿来作弊了。”回到家，我开始看那篇 Danvy 和 Filinski 的论文。这篇 1991 年的论文的想法，是从 1975 年一篇 Gordon Plotkin 的论文的基础上经过一系列繁琐的推导得出来的，而它最后的结果几乎跟我的程序一模一样，只不过我的程序可以处理更加复杂的 Scheme，而不只是 lambda calculus。我之前完全不知道 Plotkin 的做法，从而完全没有收到他的思想的影响，直接就得到了最好的结果。这是我第一次认识到自己头脑的威力。

第二个学期，当我去上 Friedman 的进阶课程 B621 的时候，他给我们出了同样的题目。两个星期下来，没有其它人真正的做对了。最后他对全班同学说：“现在请王垠给大家讲一下他的做法。你们要听仔细了哦。这个程序价值100 美元！”

下面就是我的程序对于 lambda calculus 的缩减版本。我怎么也没想到，这短短 30 行代码耗费了很多人 10 年的时间才琢磨出来。

```
(define cps
  (lambda (exp)
    (letrec
      ([trivs '(zero? add1 sub1)]
       [id (lambda (v) v)]
       [C~ (lambda (v) `(k ,v))]
       [fv (let ((n -1))
             (lambda ()
               (set! n (+ 1 n))
               (string->symbol (string-append "v" (number->string n))))))
       [cps1
         (lambda (exp C)
           (pmatch exp
             [,x (guard (not (pair? x))) (C x)]
             [(_lambda (,x) ,body)
              (C `(_lambda (,x k) ,(cps1 body C~)))]
             [(_rator ,rand)
              (cps1 rator
                (lambda (r)
                  (cps1 rand
                    (lambda (d)
                      (cond
                        [(memq r trivs)
                         (C `(_ ,r ,d))]
                        [(eq? C C~) ; tail call
                         `(_ ,r ,d k)]
                        [else
                         (let ([v* (fv)])
                           `(_ ,r ,d (lambda (,v*) ,(C v*))))))))))])
             (cps1 exp id))))
```

而这还不是 B621 的全部，每一个星期 Friedman 会在黑板上写下一道很难的题目。他不让你看书或者看论文。他有时甚至不告诉你题目里相关概念的名字，或者故意给它们起个新名字，让你想查都查不到。他要求你完全靠自己把这些难题解出来，下一个星期的时候在黑板上给其它同学讲解。他没有明确的评分标准，让你感觉完全没有成绩的压力。

这些题目包括一些很难的问题，比如 church numeral 的前驱 (predecessor)。这个问题，当年是 Stephen Kleene (图灵的学长) 花了三个月冥思苦想才做出来的。不幸的是我在 Cornell 就学到了 Kleene 的做法，造成了思维的定势，所以这个训练当时对我来说失去了意义。而我们班上却有一个数学系的同学，出人意料的在一个星期之内做出了一个比 Kleene 还要简单的方法。他的完整的代码 (用传统的 lambda calculus 语法表示) 如下：

```
 $\lambda n w z. ((n \lambda l h. h (l w)) (\lambda d.z)) (\lambda x.x)$ 
```

其它的问题包括从 lambda calculus 到 SKI combinator 的编译器，逻辑式 (可逆) CPS 变换，实现 Hindley-Milner 类型系统，等等。我发现，就算自认为明白了的东西，经过一番思索，认识居然还可以更进一步。

当然，重新发明东西并不会给我带来论文发表，但是它却给我带来了更重要的东西，这就是独立的思考能力。一旦一个东西被你“想”出来，而不是从别人那里“学”过来，那么你就知道这个想法是如何产生的。这比起直接学会这个想法要有用很多，因为你知道这里面所有的细节和犯过的错误。而最重要的，其实是由此得到的直觉。如果直接去看别人的书或者论文，你就很难得到这种直觉，因为一般人写论文都会把直觉埋藏在一堆符号公式之下，让你看不到背后的真实想法。如果得到了直觉，下一次遇到类似的问题，你就有可能很快的利用已有的直觉来解决新的问题。

而这一切都已经发生在我身上。比如在听说 ANF 之后，我没有看 Amr Sabry 的论文，只把我原来的 CPSer 程序改了一点点，就得到了 ANF 变换，整个过程只花了十几分钟。而在 R. Kent Dybvig 的编译器课程上，我利用 CPS 变换里面的直觉，改造和合并了 Dybvig 提供的编译器框架的好几个 pass，使得它们变得比原来短小好几倍，却生成更好的代码。

现在我仍然是这样，喜欢故意重新发明一些东西，探索不止一个领域。这让我获得了直觉，不再受别人思想的限制，节省了看论文的时间，而且多了一些乐趣。一个问题，当我相信自己能想得出来，一般都能解决。虽然我经常不把我埋头做出来的东西放在心上，把它们叫做“重新发明”(reinvention)，但是出乎意料的是，最近我发现这里面其实隐藏了一些真正的发明。我准备慢慢把其中一些想法发掘整理出来，发表成论文或者做成产品。

俗话说“给人以鱼，不如授人以渔。”就是这个道理吧。Dan Friedman，谢谢你教会我钓鱼。

什么是“对用户友好”



当我提到一个工具“对用户不友好”(user-unfriendly)的时候，我总是被人“鄙视”。难道这就叫“以其人之道还治其人之身”？想当年有人对我抱怨 Linux 或者 TeX 对用户不友好的时候，我貌似也差不多的态度吧。现在当我指出 TeX 的各种缺点，提出新的解决方案的时候，往往会有美国同学眼角一抬，说：“菜鸟们抱怨工具不好用，那是因为他们不会用。LaTeX 是‘所想即所得’，所以不像 Word 之类的上手。”

殊不知他面前这个“菜鸟”，其实早已把 TeX 的配置搞得滚瓜烂熟，把 TeXbook 翻来覆去看了两遍，“double bend”的习题都全部完成，可以用 TeX 的语言来写宏包。而他被叫做“菜鸟”，这是一个非常有趣的问题。所以现在抛开个人感情不谈，我们来探讨一下这种“鄙视”现象产生的原因，以及什么叫做“对用户友好”。

首先我们从心理的角度来分析一下为什么有人对这种“对用户不友好”的事实视而不见，而称抱怨的用户为“菜鸟”。这个似乎很明显，答案是“优越感”。如果每个人都会做一件事情，如何能体现出我的超群智力？所以我就是要专门选择那种最难用，最晦涩，最显得高深的东西，把它折腾会。这样我就可以被称为“高手”，就可以傲视群雄。我不得不承认，我以前也有类似的思想。从上本科以来我就一直在想，同样都会写程序，是什么让计算机系的学生与非计算机系的学生有所不同？经过多年之后的今天，我终于得到了答案（以后再告诉你）。可是在多年以前，我犯了跟很多人一样的错误：把“难度”与“智力”或者“专业程度”相等同。但是其实，一个人会用难用的工具，并不等于他智力超群或者更加专业。

可惜的是，我发现世界上有非常少的人明白这个道理。在大学里，公司里，彰显自己对难用的工具的掌握程度的人比比皆是。这不只是对于计算机系统，这也针对数学以及逻辑等抽象的学科。经常听人很自豪的说：“我准备用XX逻辑设计一个公理化的系统……”可是这些人其实只知道这个逻辑的皮毛，他们会用这个逻辑，却不知道它里面所有含混晦涩的规则都可以用更简单更直观的方法推导出来。

爱因斯坦说：“Any intelligent fool can make things bigger and more complex... It takes a touch of genius - and a lot of courage to move in the opposite direction.”我现在深深的体会到这句话的道理。想要简化一个东西，让它更“好用”，你确实需要很大的勇气。而且你必须故意的忽略这个东西的一些细节。但是由于你的身边都是不理解这个道理的人，他们会把你当成菜鸟或者白痴。即使你成功了，可能也很难说服他们去尝试这个简化后的东西。

那么现在我们来谈一下什么是“对用户友好”。如何定义“对用户友好”？如何精确的判断一个东西是否对用户友好？我觉得这是一个现在仍然非常模糊的概念，但是程序语言的设计思想，特别是其中的类型理论(type theory)可以比较好的解释它。我们可以把机器和人看作同一个系统：

1. 这个系统有多个模块，包括机器模块和人类模块。
2. 机器模块之间的界面使用通常的程序接口。
3. 人机交互的界面就是机器模块和人类模块之间的接口。
4. 每个界面必须提供一定的抽象，用于防止使用者得到它不该知道的细节。这个使用者可能是机器模块，也可能是人类模块。
5. 抽象使得系统具有可扩展性。因为只要界面不变，模块改动之后，它的使用者完全不用修改。

在机器的各个模块间，抽象表现为函数或者方法的类型(type)，程序的模块(module)定义，操作系统的系统调用(system call)，等等。但是它们的本质都是一样的：他们告诉使用者“你能用我来干什么”。很多程序员都会注意到这些机器界面的抽象，让使用者尽量少的接触到实现细节。可是他们却往往忽视了人和机器之间的界面。也许他们没有忽视它，但是他们却用非常不一样的设计思想来考虑这个问题。他们没有真正把人当成这个系统的一部分，没有像对待其它机器模块一样，提供具有良好抽象的界面给人。他们貌似觉得人应该可以多做一些事情，所以把纷繁复杂的程

序内部结构暴露给人（包括他们自己）。所以人对“我能用这个程序干什么”这个问题总是很糊涂。当程序被修改之后，还经常需要让人的操作发生改变，所以这个系统对于人的可扩展性就差。通常程序员都考虑到机器各界面之间的扩展性，却没有考虑到机器与人之间界面的可扩展性。

举个例子。很多 Unix 程序都有配置文件，它们也设置环境变量，它们还有命令行参数。这样每个用户都得知道配置文件的名字，位置和格式，环境变量的名字以及意义，命令行参数的意义。一个程序还好，如果有很多程序，每个程序都在不同的位置放置不同名字的配置文件，每个配置文件的格式都不一样，这些配置会把人给搞糊涂。经常出现程序说找不到配置文件，看手册吧，手册说配置文件的位置是某某环境变量 FOO 决定的。改了环境变量却发现没有解决问题。没办法，只好上论坛问，终于发现配置文件起作用当且仅当在同一个目录里没有一个叫 “.bar” 的文件。好不容易记住了这条规则，这个程序升级之后，又把规则给改了，所以这个用户又继续琢磨，继续上论坛，如此反复。也许这就叫做“折腾”？他何时才能干自己的事情？

TeX 系统的配置就更为麻烦。成千上万个小文件，很少有人理解 kpathsea 的结构和用法，折腾好久才会明白。但是其实它只是解决一个非常微不足道的问题。TeX 的语言也有很大问题，使得扩展起来非常困难。这个以后再讲。

一个良好的界面不应该是这样的。它给予用户的界面，应该只有一些简单的设定。用户应该用同样的方法来设置所有程序的所有参数，因为它们只不过是一个从变量到值的映射 (map)。至于系统要在什么地方存储这些设定，如何找到它们，具体的格式，用户根本不应该知道。这跟高级语言的运行时系统(runtime system)的内存管理是一个道理。程序请求建立一个对象，系统收到指令后分配一块内存，进行初始化，然后把对象的引用(reference)返回给程序。程序并不知道对象存在于内存的哪个位置，而且不应该知道。程序不应该使用对象的地址来进行运算。

所以我们看到，“对用户不友好”的背后，其实是程序设计的不合理使得它们缺少抽象，而不是用户的问题。这种对用户不友好的现象在 Windows, Mac, iPhone, Android 里也普遍存在。比如几乎所有 iPhone 用户都被洗脑的一个错误是“iPhone 只需要一个按钮”。一个按钮其实是不够的。还有就是像 Photoshop, Illustrator, Flash 之类的软件的菜单界面，其实把用户需要的功能和设置给掩藏了起来，分类也经常出现不合理现象，让他们很难找到这些功能。

如何对用户更加友好，是一两句话说不清楚的事情。所以这里只粗略说一下我想到过的要点：

1. 统一：随时注意，人是一个统一的系统的一部分，而不是什么古怪的神物。基本上可以把人想象成一个程序模块。
2. 抽象：最大限度的掩盖程序内部的实现，尽量不让人知道他不必要的东西。不愿意暴露给其它程序模块的细节，也不要暴露给人。“机所不欲，勿施于人”。
3. 充要：提供给人充分而必要（不多于）的机制来完成人想完成的任务。
4. 正交：机制之间应该尽量减少冗余和重叠，保持正交(orthogonal)。
5. 组合：机制之间应该可以组合(compose)，尽量使得干同一件事情只有一种组合。
6. 理性：并不是所有人想要的功能都是应该有的，他们经常欺骗自己，要搞清楚那些是他们真正需要的功能。
7. 信道：人的输入输出包括5种感官，虽然通常电脑只与人通过视觉和听觉交互。
8. 直觉：人是靠直觉和模型(model)思考的，给人的信息不管是符号还是图形，应该容易在人脑中建立起直观的模型，这样人才能高效的操作它们。
9. 上下文：人脑的“高速缓存”的容量是很小的。试试你能同时想起7个人的名字吗？所以在任一特定时刻，应该只提供与当前被关注对象相关的操作，而不是提供所有情况下的所有操作供人选择。上下文菜单和依据上下文的键盘操作提示，貌似不错的主意。