程序员生存定律

作者：李智勇
V 众投发起人，《完美软件开发：方法与逻辑》作者，
微博：李智勇 SZ，微信：vfacebook

V 众投(vzhongtou.com)是国内最靠谱的生活问答社区，只有验证用户才能回复和投票，一人一号，一人一票。
公众号是 vzhongtou，二维码如下，欢迎来来瞄瞄：

前言：解码程序人生

有这样两个人，他们一个 66 年出生于陕西长安和一个 68 年出生于山西阳泉，而后经过自己的努力两个人又先后以长安县第一名和阳泉市第一名的成绩，在 85 年和 87 年考进北京大学。在近 50 岁的时候两个人一个历经社教、装饰业、卖肉最终在档案馆找到了归宿，一个则在留学、Infoseek 打工、回国创业之后成为知名公司的 CEO。前一个是卖肉才子陆步轩，后一个则是百度 CEO 李彦宏。两个人起点类似，但现状的差异又是如此的巨大！究竟是什么力量造成了这种人生差异，其中又是否有规律可寻？

对上面这个问题，我想至少在程序员的职场里答案是肯定的，在这里人生是有一种内在的支配规律的。

本书中试图用四个可控变量来定义程序人生的规律，它们分别是：自身价值---也就是你能干什么；自身价值上的表达力---也就是别人认为你能干什么；自身价值的稀缺性---也就是在特定时空背景下，市场对某种技能的渴求程度；身处公司的特质和未来---也就是公司提供了怎样的平台给人发挥。本书认为这四个变量一起决定了一个人在职场中的市场价值，个人的一切选择主要是为了在这四个变量上有所收益，并使市场价值最大化。

程序员是一个非常离奇的职业，虽然名字都是程序员但现实中的程序员各方面差异却极大。单纯从物质收益的角度看超级程序员可以驾着游艇度假休闲，而有的程序员却只能怀揣仅剩的 180 元而看不到明天。这是身处职业路径上不同位置所带来的差异。职业路径确实就像分了级的梯子一样，走到那个高度就有那个高度的荣光。而为了在这梯子上走的顺畅，则需要使努力与规律契合，并在关键时刻做对选择，惟其如此人生整体表现出来的效能才会高。

在《黑客帝国》这部电影中曾经出现过这样一种场景，尼奥带着钥匙匠走在一条满是门的走廊里，必须开对门，尼奥才能见到 Matrix 的架构师。人生与此类似，每个人都面对和尼奥一样的问题：未来处于未知之中，每一次选择就是一次对人生未知部分的解码，选择之后人生的确定性增强而不确定性减弱，当所有未知褪去，人生也就瓜熟蒂落。

就像尼奥历尽许多辛苦才来到众多选择之门前面一样，要想主动选择而不是被动承受，首先要争得选择权。这种选择权往往来之不易，大多时候一个人许多年的努力拼搏才能换来一次选择的权利；但选择权的浪费却又无比的容易，这在软件行业尤为突出。看看周围，就可以感觉到有多少本来可以很杰出的人因为选错公司、选错行业、选错领域而默默无闻。

这是由软件独属于自身的特征所决定的。比如说：软件本身并不是一个边界清晰的领域，当软件和数学相结合时，它具有数学的特征，当软件和管理结合时，它就具有管理的特征。这种特别之处使程序员的选择权更容易被误用，因为待选项实在太多。

要想挣得选择权并尽可能避免误用，事实上需要对程序人生中所内蕴的规律洞若观火。为此，我们就需要知道程序员可能的人生出口、需要知道左右这种人生出口的职场定律、需要知道定律之下那些是可控变量、那些是预设前提、需要知道可控变量应该如何去改变。其中可控变量是挣得选择权并避免误用它的关键。这也正是本书的根本目的。通过对规律的解析来为程序员的成长来提供一点参照，让人少走一点弯路。

最后为了避免被人指责为功利心太重，不懂人生价值，补充一点关于人生价值的说明。

一个人变成什么样是有规律可循的，比如：陆步轩最终成为档案馆的职员、李彦宏成为 CEO，这背后是有一种支配规律的；但变成这个样子是否有价值，则只依赖于人的内心世界和自身的选择。陆步轩可以认为自己很成功，李彦宏可以认为自己很失败。这样的想法依赖于人的价值系统，就像只有原点、X 轴、Y 轴定了才有坐标可以标记位置一样，价值系统定了才能判定某种现实是否有价值，比如说：忠诚的神父可能并不会认为成为 CEO 是有价值的事情。这种价值系统与个人关联很大，也只有个人才有定义个人价值系统的权利。本书中基本不谈这种人生价值，而只谈如何取得更大的成绩。

第一章 代码之外的人生

§1.1 程序员的人生出口

很多人非常想知道自己的未来是什么样子的，迫切到一定程度甚至会找算命先生。如果并不是想得到一个精确结果，这事儿其实并没有想的那么难。程序员的人生看起来五花八门，可以是 Windows 系，可以是 Android 系，可以是 iPhone 系等等，但如果为之做点抽象，那就会发现人生出口的类别其实不多，接下来我们就结合例子来看看程序员的人生中最可能的几个出口。这些出口里有大部分人的未来景象。

1.1.1 成就超一流高手

袁峰先生是《Windows 图形编程》一书的作者，也是一位纯粹的程序员。从袁峰先生公开的简历(www.fengyuan.com)来看，袁峰先生从 1993 年加入 HP 一直到现在在微软从事 CLR 的开发始终是在编程第一线。在他的 Blog 上可以看出清晰的技术路线图：Printing →GDI/GDI+ → XPS →WPF →Parallel Computing → Debugging → Performance → CLR。

考虑到袁峰先生是 78 年入学的大学生，今年估计已近 60 岁，我们可以想象在剩余的职业生涯中，袁峰先生应该会一直从事编程工作。这也就意味着袁峰先生的职业生涯以编程开始，以编程结束，是非常典型的纯技术流的程序员：一生中不涉其他，唯有编程。

如果你很仔细的读过《Windows 图形编程》，你就会对袁峰先生在技术上的造诣佩服的五体投地，无论是深度还是广度。可惜的是，随着技术更迭，这本书里讲的很大一部分东西已然失去了现实意义，所以大概是买不到了。

这是程序员比较典型的一个出口，其特征是走比较纯粹的技术路线，其要点是技术上要能达到一定高度，最终成为顶级程序员。常说的 CodeGuru 和架构师都可以归在这一类别下，虽然这两者间还是有差异。走这条路时最怕陷入平庸，那样的话高不成低不就，很容易被新入行者超越。一旦成为技术路线上的低值人群，那就有被提前淘汰的风险，同时几乎必然会转变成失意者。

1.1.2 积累、爆发、开始创业

与袁峰先生成为鲜明对比的是大家很熟悉的求伯君先生，这两个人虽然八竿子打不着，但参照意义其实很强。

袁峰先生 1982 年本科毕业，而求伯君先生 1984 年本科毕业，可以讲他们是同龄人。袁峰先生毕业于南京大学，求伯君先生毕业于国防科技大学，可以讲两个人起点、天分估计也差不多，但两个人走的道路却完全不一样。

求伯君先生在 1994 年前可以说是比较典型的程序员，这段时间里他和袁峰先生非常类似，其中值得写入软件开发史册的事情是几乎一个人完成了 DOS 下的 WPS。

在这个时间段上，求伯君先生无论在技术上还是产品上应该都是超绝的高手。这点可以从小米 CEO 雷军先生的回忆上看出来：

1990 年初，我在一个朋友那用了 WPS 汉卡，当时就被震住了。界面易用美观，更强的是打印结果可以先模拟显示出来。署名是香港金山公司求伯君，觉得这个“香港”软件写

得真好。

1994 年求伯君先生创立珠海金山电脑公司，自此人生路径与袁峰先生开始有绝大分歧。在这之后，求伯君先生更多的是以企业家，而非是程序员的身份出现在大众面前，直至 2011 年求伯君先生宣布退休，退出公众视线。

求伯君先生的履历昭示了另一种程序员的道路，即以技术和热情创造辉煌产品，再以产品为创业基石。

IT 行业本来就是个智力密集型的行业，想想创造一家水泥公司和软件公司的差别可以更好的理解 IT 行业与传统行业间的这种差异。互联网、云计算的兴起使创业的门槛进一步降低，理论上讲只要能做出优秀的产品并找到风险投资，几乎每个人都可以开始创业。虽然创业并非是这本书主要探讨的内容，但在当前的形势下创业确实也是少部分程序员的一个人生出口。

1.1.3 转向管理之路

由程序员而管理者是一条非常常见的道路，很像是程序世界里的学而优则仕。

现在很多软件企业中的中级管理人员（包括部门经理、产品经理、项目经理等）里 70 后、80 后比较多，他们大多在 2000 年之后毕业。

这部分人员中的很大一部分走的是这样一条道路：在毕业后往往会从事 3~5 年的编程工作，接下来由于工作表现不错，同时也具备比较好的表达和沟通能力，于是开始肩负起部分管理工作。一旦开始从事管理工作后，接下来在工作中管理的成分越来越重，和程序的距离越来越远，已经算不得是纯粹的程序员了。从时间开销的角度看，他们同 PPT 打交道的时间稳步上升，而直接同编译器打交道的时间则稳步下降，并有变 0 的趋向。

这条道路是如此普遍，以至于每个程序员只要往四周一看，就都可以看到这条道路上的人。好奇者可以问问身边的中层管理人员，看看是不是这样一个成长故事。

走上这条道路的人需要爬的是另一种梯子，比如从项目经理到技术总监再到 CTO。当然每一层的选拔都会卡死相当一部分人。某些前行无路的人往往需要做一个艰难的选择，要么安全的维持现状，要么冒点险切换公司。

在很多技术路径较短，不以技术为核心的公司里，为了收入的提升这往往是一条必走的道路，所以可以讲这也是一个比较常见的人生出口。但很多人所认为的“程序员 30 岁前需要向管理转行，这碗青春饭没法持续吃下去”，并不永远正确，在很多场合下这类选择反倒相当于自废武功，使自己管理上没什么成绩，技术又荒废了，这点会在后面的章节里进一步提到。

1.1.4 维持原状的老码农

维持原状的含义是加入程序员队伍后，工作内涵并无实质性变化，只是体现为一种简单的重复---这是与第一种程序员不同的关键，第一种程序员要不停的有技术上的深化和提高。

这类程序员所做的工作的表象形式会有所不同，比如可能今天做的是处理服装公司的网页，明天处理的则可能是化工厂的网页，但实质上各种工作本质上差别不大，从技术上看，没有层级差异。

下面走到这条道路上的程序员的常见经历：姑且把这个程序员成为 X，他大学毕业后，加入一家对日外包类公司。接下来他工作的公司和地点就不停的变化。去过苏州、南京、北京、上海，由于总是不太如意，迫于经济压力也曾经到过日本，做过派遣社员，但当无法被派遣出去时，无奈之下就又回到了国内。

在不停的迁移过程中，工作内容变更的比较频繁，横跨多个领域，比如：银行、证券、GPS 等。但从层次来看，基本上是在应用层打转，工作内容也趋于简单化，基本上是在设计好的文档指导下做编码。由于自身能创造的价值没有显著突破，作为结果收入、职位基本上改善不多。挣扎 10 年之后，他很苦恼，因为自己做的事情，毕业生往往也很快就可以做，而生活的压力却在不停的随着年纪的增加而增加。

上述这样的经历应该并不特殊，在很多人的身上应该都可以看到类似的人生轨迹。差别只是 X 可能是公司 A，具体到某个人可能是公司 B；X 可能去的是华东，具体到某个去的可能是华北。这类程序员如果不退场，不升级，那接下来的生命中工作和生活只会以某种模式继续重复。如果运气不好，甚至可能会失业。

更可怕的是到一定年纪后，很大一部分这类程序员会发现自己连退场的资格也没有，因为除了会写程序，并不会做其他的事情，与此同时房贷、家庭却成为一种长久的负担。这个人生出口虽然很让人叹息，也没人愿意选择，但它确实存在，并且很多人可能一不小心就走到了它的面前。

本书更主要的目的真的不是让人成为辉煌伟大的人物，我也不认为任何一本书有这样的能力。本书最主要的目的就是希望能尽可能帮助初入行者不要走到这条道路上来，毕竟一旦走到这条道路上来，人生就过于凄惨了。

1.1.5 提前退场、向代码说再见

对于喜欢闲聊的人，有时候会在闲聊中发现些让人吃惊的事情。很久以前一个房地产中介在听说我在软件公司工作后，告诉我：我也曾经是做软件的。我很惊讶的问他，怎么就不做了，却跑来做房产中介，这不是自废武功么。他回答我：做程序员太累，没前途。

在那之后，接下来这些年里，身边断断续续总会有人退出程序员这个行业。有的去销售电子产品，有的去做公务员，去做纯黑盒的测试诸如此类。2013 年最吸引大家眼球的退场事件是新浪架构师徐佳在奋斗数年后，告别 PHP，告别互联网开始卖水果。

据说徐佳先生因此而焕发了青春，但我个人并不看好这类选择，也不认为这是一种正确的选择。提前退场大多时候是对之前选择的一种否定，从人生整体收益的角度看，大致上是亏了，至少亏掉了做程序员那几年的时间。

这类事情一旦上升到人生价值的层次上，那就没有是非，只有个人选择，所以本书中不从这个视角考虑问题，而主要从功利且现实的角度来考察这类事情。

虽然退场大多时候从投资收益的角度看并不怎么好，往往是无奈下的选择，但这确实也是程序员群体的一个人生出口---“悄悄的我走了，正如我悄悄的来；我挥一挥衣袖，不带走一片云彩。”

§1.2 那个是你的人生出口

对于一个程序员，大致的结局多是上述五种中的一个。其中转管理的程序员和成就高手的程序员算是稳中有升。创业者算是生死未卜。维持原状的老程序员则是失意的。他的选择权会被收的越来越窄，在工作层面，他需要和毕业生竞争，在家庭支出方面他所要承担的比别人一点不少。人生总是这样，向你索取的一端往往是确定的，而你所能获得的却往往是不确定的，人就夹在这个中间，如果无法给自己撑开足够大的空间，那就会很难受。

初入职场的程序员看到这里，最为关心的几乎一定是：这么多出口，我的未来可以是其中那个？

很不幸这并非是一个可以立刻给出答案的简单问题，为了回答这个问题不只要考虑那个是适合自己的出口，也要考虑如何走到这个出口面前，并避免那些代表失意的。

为了回答上述问题，需要对未来做点预测，而为了对未来做点预测，那么需要为人生建立一个大致的模型，接下来把个人的努力等作为输入，这样输出端会因此而有各种相应的变化。但这个模型就像被藏起来的密码，需要费点心思才能找得到---这也正是这本书要做的事情。

这时尝试以写程序的方式建立一种精确的人生模型是非常困难的。一旦试图这样做很容易进入一个误区，即期望为人生建立一个真理式的公式。

比如：如果你努力，那么你一定成功。如果你有责任感，你一定成功。如果你人品好，你一定成功。如果你读书，那么一定成功。如果你注意细节，那么你一定成功。如果你时间管理做好，你一定成功。……

上边的这些逻辑并非凭空想象，而是很多成功学书籍的基本出发点。这种种来自成功学书籍的陈词滥调很像一个拙劣的程序员把分支、选择、循环硬套在人生上的结果，虽然看着有几分道理，但实是不对的。在分析人生规律时，事实上很难找到类似自然科学中的、精确的绝对真理。据说某位行为科学家曾经总结过：上帝把所有容易的问题都留给了物理学家。言下之意是，社会学科的问题都大不易。

其根本原因在于，人生是不精确的。太多偶然因素可以影响最终的结果，而这些因素本身又大多不可量化，试想一下你应该如何量化一份时运，一场姻缘。但这并不意味着不可以对人生的方向进行大致的分解和预测，比如：如果一个人每天除了吃饭就是睡觉，那他人生一定没有成绩，这就很容易预测。

这里的关键是要找到起关键作用的可控因子和权变变量。我们可以说 X 导致 Y，但这只能在 Z 所限定的条件下。而又由于权变变量太多，偶然性无法完全去除，我们也必须在结果中接受一定的模糊性。一个人可以是天才程序员，也很努力，但他跑到了一个外包的公司做简单重复的工作，那么就可能他的才能始终无人发现，人生一样可能没有闪光点。这就是人生的偶然性。

为了帮助程序员走到自己期望的人生出口而避免失意，那么首先要分解的就是都是什么力量左右着一个程序员的未来？这就是下一章中即将展开的内容。

这是成功学么？

上面所表述的内容很容易让人想到成功学。是的，这本书所主要讲述的规律和方法确

实确实可以划入成功学的范畴。这样一来就要连带着回答另一个问题：我们可以相信成功学么？ 如果说自然界发展有规律，社会发展有规律，没道理个人的成长就没有规律。从这个角度看，讲规律的成功学是可信的，但将具体方法，讲怎么怎么做一定成功的成功学就有点问题了。这更像药和包治百病的脑白金的区别。成功学本身没什么太高的壁垒，这就导致口若悬河的人更容易吸引目光，最终结果就是这池子里充满了太多的说客，放眼望去也就全是《细节决定成败》、《有效沟通》这类书籍。也不是说这类书籍没有价值，而是说由于其往往会用片面取代整体，而并不具有太大的现实价值。突然看到李四成功了，也看到了李四比较注重细节，最终就得出一个细节决定成败的结论，这并不怎么有意义。因为人的成功往往同时取决于自身和环境。研究如何成功时事实上需要与编程一样的抽象能力。真要想找到种种与成功的规律，那就要多采集案例，从外部环境、个人努力、工程特征等多个方面进行切入，抽象出起作用的要素，再剃掉偶然性的部分，只保留必然性的部分。这时候往往能够更贴近本质规律。这类成功学是有用的，但通常达不到一看就可以飞黄腾达的地步，只能让人少走一点弯路。

§1.3 小结

人生其实是条曲线，其振幅则随着时间的流逝而逐渐收窄。对婴儿而言，其未来具有无限的可能性。对耄耋老翁而言，其未来则唯一而确定。而一个人最终振幅的高度则同时取决于：机缘、天分和努力。家世，时代种种皆可归为机缘。智商、情商、体质种种皆可归为天分。机缘和天分皆是命数，无从左右的起。也即是说，一个人持有的，可以打破既定命数的砝码也只是努力而已。从人生长短的角度来看，上帝是公平的，每个人可用时间大致相同。不同的则是努力的效能。

努力本身并不只是简单的付出，还牵涉到方向的选择，形势的顺应，环境的驾驭诸如此类。考虑了这些的努力更容易在收入、职位等有形收获上看见结果，忽视这些的则更可能辛苦却颗粒无收。这就是在下一章中将会展开的生存定律。

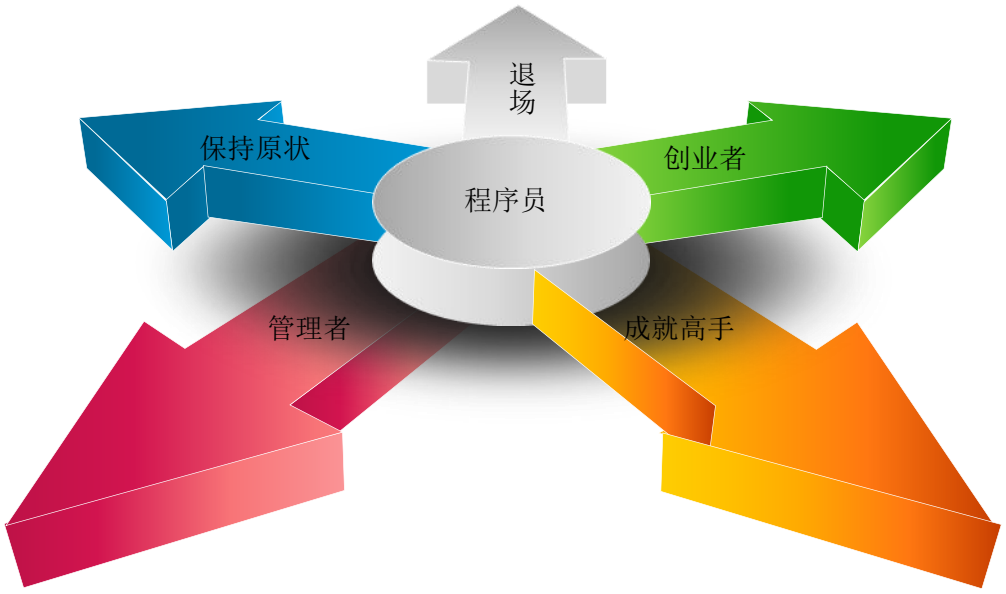


图 1-1：程序员的人生出口

第二章 职场生存定律

§2.1 交换是职场里一切的根本

支撑职场的基本规则是交换，交换的两端分别是你可创造的价值与你的职场位置（包含收入）。交换就像任督二脉间的通道一样，越是通畅，人生也就越顺风顺水；堵得越死，人也就越寸步难行。

这点要刻在脑子里，一旦要忘记了，就赶紧打自己两个耳光。忘了这点的人一旦被炒，就会很委屈的发“不要拿公司当家”这类感慨。

那什么是交换？

在一般人眼里，交换就是你有个东西我要，我有个东西你要，大家互通有无这样一个过程。但在学者眼里，事情却要更复杂一点。何新先生在《反主流经济学》中，对交换进行了深度剖析，他说：

- 社会交换成立之第一前提，就是人类之有欲求与需要，而靠自身不能满足。
  因此，事物是否具有使用价值，决定于其能否满足人类之某种欲求与需要。所以，事物之内在属性形成其使用价值。

- 私有制，占有，占有权利，是社会交换得以发生之第二前提。
  人有需求之物，而先为他人已据有者，若欲取之，只有二途：或强取（掠夺、战争、索要），或巧取——后者即交换。

- 随机之交换，导致对交换品的偶然随机定价，故成交之价格有极大随意性。
  而常态之交换，则必然趋向理性之定价，即均衡定价。买卖双方经协商而接受的均衡定价之尺度，其依据乃参照商品所内涵之真实价值。
  故：（1）商品确有真实价值存在。
  （2）真实价值乃是一种无形的、形而上的虚拟实体。（马克思说：价值是一种社会关系的反映。）

- 因此，商品价值之评估有三重性：
  一为使用价值，二为内涵价值，三为外化的实际交换价值。
  交换价格是内涵价值的外化表现。内涵价值与价格之不同，在于内涵价值是内在的、非外现的。

- 物品买卖成交之交换量，谓之价格或市场价格。
  价格具主观性，是商品内涵固有之价值之外化。内涵价值能否得到实现，取决于需求者之购买评估。内涵价值得到市场实现，意味着商品转换为货币。

- 但是，价值在市场中之二重化，价格对价值之背离，造成等价交换规律之破坏。等价交换意味着等价价值品之交换，并非意味同等价格品之交换。同等价格对同等价格之交换，未必是等价交换。因市场价格之形成，受市场之需求度影响，而非单纯

决定于价值之同等性。

上面这段文字说的是经济，所以很多人会看的云里雾里，但其中对交换的解析入木三分，只要我们还承认工作本身也是一种个人和法人间的交换，那么就逃不出这些约束---虽然有杀鸡用牛刀之嫌疑。

尽管上面的解释已经非常清楚，但为了让其更加的通俗易懂，这里附加一点说明：

交换本身起源于互有所需，比如：某一公司开发打印机驱动程序，那么就需要了解页面描述语言(PDL)、操作系统打印子系统技能的程序员。

交换本身的基本原则是等价交换，但这种原则往往会被市场需求度等因素破坏。比如：TTS 可能很难搞，一个人学习了数十年，本应获得较高的市场价格，但很不幸，如果搞这个的人碰巧很多，或者应用还不广，这时候交换价格也可能很低。想想同样是在科大讯飞，工作实质没什么太大变化，但在移动互联网兴起前和兴起后交换的价格会不会有很大差异。

破坏这一原则的因素还有很多，比如说垄断。如果一个人掌握的技术只有一家公司用，那么这个公司具有破坏等价交换原则的权利。反过来讲，如果某项技术只有某个人掌握，那么这个人具有破坏等价交换的权利。

工资可以表示为一种成交价格，这一价格具有一定的随意性，具体表现为同样工作内容，不同公司，不同时间点，收入差异可能非常巨大。比如：同等技术能力下，表现力好的程序员可能更容易获得较好的薪资。但总体市场行情却在一定范围内趋向于稳定，比如毕业生的起薪大致在 10K/月以下，Google、Facebook 这些公司的平均薪资相差也不大。

这也就意味着，影响最终交换价格（即工资收入的）的主要是两个因素：内含价值和市场因素（稀缺性等）。

在程序员与公司进行交换的过程中，其中最为基础的一点是你要有维护自身权利的能力，即程序员自身要有选择权，在只能被选择时，事情会趋向于另一个极端：必须不停的让步，放弃各种可争得的权利，最直接的表现是收入上没有议价能力。

很少有公司会主动宣传工作首先是一种交换，但这一事实本身却应该没什么太多的争议。但细想下来这个基础支撑点也只能是交换，恰如食物、水与生命间的关联。对外，企业有所产出，与客户交换获取利润；对内，则是员工有所产出，与企业交换获取工资等等。这是经济形态所决定的，在这一前提下，裁员与跳槽都是一种必然出现的现象，反倒是雇佣终身制是反其道而行。

认清交换是第一支撑之后，我们就可以推导出职场生存定律。

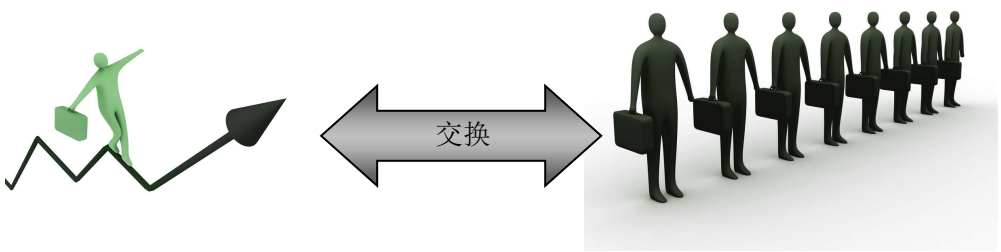


图 2-1：交换支撑职场

交换的衍生品：裁员、跳槽、雇佣终身制

大概是在2004年，联想进行了一次裁员，某一位亲身经历了裁员的员工在网上发表了一篇叫《公司不是家》的文章，引起了非常大的轰动，这轰动甚至大到了柳传志先生亲自为此写了篇回复的地步。

这件事情在有点年纪的IT人脑海里应该还有很深的印象。

如果我们认识到职场的基石首先是一场交换，那么个人跳槽、公司裁员就都不是让人很意外的事情。虽然公司会努力营造归属感，创建企业文化，但市场大潮之下，公司确实无法保证不裁员。这在情感上很难接受，但在交换上却非常自然：即使你大白菜烂在了地里，我不用大白菜，我也没必要购入它，除非是因为慈善。

真的让人意外的事情是“雇佣终身制”这一做法。雇佣终身制未尝没有收益，但潜在的与等价交换规则相冲。

在日本，雇佣终身制是由松下幸之助提出来的，在当时特定的历史环境下这也是成功的，这从松下幸之助被誉为“经营之神”上可见一斑。

但随着经济环境的恶化，松下似乎也顶不住了，在2012年时见松下裁员的消息于各大网站。

另一个试图挑战交换规则的公司是IBM，IBM当年也推行类似的雇佣终身制度，但后来IBM也顶不住了，为了让大象来跳舞，郭士纳大刀阔斧的进行机构精简。

所以，长线来看，职场首先是交换场，在此之上才有其他。

下面是柳传志先生对《公司不是家》的回复，看了之后，读者对交换为职场第一支撑应该会有一个更加直观的认识。

一名联想员工写的《公司不是家》的文章登在网上，委婉哀怨，记载的是他的真实感受。我看了以后，心里很难过。在裁下去的员工中，有一些是我当年直接指挥过的老员工，他们谁都没有打电话给我，在默默地接受一切，这让我心中更加产生歉意。我想这其中的更多员工是元庆的直接下属，他歉意的心情一定更胜过我许多。

我在想，一个企业应该遵循的最根本原则就是发展，只有发展才能做到为股东、为员工、为社会几个方面负责；而从发展的角度出发，企业就必须上进，内部就必须引进竞争机制。员工在联想既要有感到温馨的一面，更要有奋勇争先而感到压力的另一面，因此不能把企业当成一个真正意义上的家是必然的。在家里，子女可以有各种缺点，犯各种错误，父母最终都是包容的。企业则不可能是这样。

在《公司不是家》这篇文章里，提到了战略性调整是由于领导的错，却要普通员工负责任的问题。我看到这个观点时，停下来作了思考。我想，他说的是对的。这次，有一部分员工被裁和领导的决策失误有关，和战略制定的失误是有关的。这是非常沉痛的事。所以，我们应该向因此而被裁的员工诚恳地说“对不起”。但是我，作为董事长，以企业发展为根本追求，我应该要求杨元庆他们怎样工作呢？一种做法是尽量小心谨慎，避免受损失；一种是要求突破、创新，就一定会犯错误。我们是在一种很困难的情况下打仗，在方方面面都和外国企业有相当大差距的情况下做的，但是我们必须去争，去抢；去拼！我们必须在毫无经验的情况下进入新的领域，投入人力、物力去尝试，由于我们看不清方向，由于我们没有掌握好方法，我们会被碰得头破血流，付出惨重代价。

在我直接负责的十几年中，不知犯了多少错误，付出了多大代价，只不过领导宽容，没有跟我算账罢了。在惠阳联想工厂的马路对面，有40万平方米的土地，至今闲置，价值6000万，是1997年前后联想买的。当时我拍板准备发展制造业，由于后来要全力支持别的业务，我把它停了。1998年联想全年的利润也就两三个亿，6 000万是多大的分量啊。负责组织这项工作的同事也都全部调整，也有一些因此就离开了联想。从1988年联想向海外出击起，分分合合多少次，在这些整合中，我已经说不清有多少支队伍离开了联想。说来惭愧，由于我作为主要负责人的失误，可能会改变一些人一生的命运。

当我想起这些事情的时候，我会问自己，我应该怎么办？今天同样遇到了这样的情况，我要回答的问题是我应该怎么样要求杨元庆。我对元庆说的只能是：

一、牢记你的目标，牢记你的责任，进取、创新、突破，此外没有别的出路；

二、要爱惜资源，特别要爱惜员工，牢牢记住以人为本；

三、去打仗，十仗中胜七仗就是优秀的指挥官！

我很抱歉地对《公司不是家》的作者说，我们考虑问题的角度不同。元庆只能从企业发展的角度，从大局的角度看问题，这才是最根本的以人为本，最根本的为员工负责。如果元庆真的用为局部员工负责的方法去考虑问题，企业就会陷入一片儿女情长之中，完全无法发展，中国就会失去联想。因此企业前进的主旋律只能是战鼓，是激昂。

以元庆为代表的联想高层领导承受着巨大的压力。员工们说：“我们把生命中最好的时光交给你们了，带好我们，不要走错，”股东们更是不停地在说：“发展！利润！增长！”同行则是要抓住每一个机会，给我们以致命的打击。在这无休止的竞技场上，稍有不慎就会鼻青脸肿，头破血流。

我是过来人，我知道个中滋味。因此我们——股东们、员工们、对他们——企业管理的直接负责人，要严格要求，要批评，要提意见；同时也要鼓掌，要呐喊，要叫好！我们上下一心，精诚团结，打胜仗，我们不骄不躁；打了败仗，我们互相鼓励，依然战歌嘹亮，这才是我们，这才是真正的斯巴达方阵！

§2.2 生存定律总纲

如果我们承认交换是职场里一切的根本，那么就可以基于交换的特征推导出生存定律的纲要。

假设说一个人的技能所带来的价值是 S，实现程度是 A，那么 S x A 即为一个人可以为公司创造的可见价值，也即是可以从公司交换到的最大价值。

其中技能 S 是指一个人所能做的事，即自身的价值，可以是编程语言、设计知识也可以是管理知识等等。实现程度 A 则是指各种技能被周围的人认知的程度。

这好像有点绕，但实际上一个人的实际能力水平和被认可的能力水平往往存在偏差，极端的情形就是一个人确实很优秀，程序也确实写的很好，但周围的人都认为他不行，并只分配他做周边性的工作，这个情形下这个人的价值实现程度很低，创造的价值也很低。

这就是我想说的生存定律总纲，并不复杂，也不抽象。下面我们来进一步做点分析。

如果收入水平为 I，那么当 S x A > I 时一个人是有选择权的也是安全的，否则一个人对于公司而言是负资产（至少是被认为是负资产），潜在的有被剔除的风险。一旦一个人在多家公司里都处于这样一种情形下时，这个人的选择权会越来收的越窄（只有公司可以选择个人，个人却没可能选择公司），人生也就会越来越被动。

当然无论技能 S 和实现程度 A 都很难清楚量化只能做定性分析，但确实有几个因素会使实现程度 A 急速膨大。这几个因素可以概括为：自身价值表达力，自身价值的稀缺性，公司的特质和未来。

就日常的感受来看，这往往是程序员这个群体不太关注的地方。作为结果很多人的真实价值可能并没有得到体现，或者说被低估了。

- 自身价值的表达力

简单来讲，一个人可以有屠龙的本事，但如果所有周围的人都认为你只能杀鸡，那么现实中，你的能力就是杀鸡而不是屠龙。

一个人的真正自我和别人眼里的你往往是有差距的。这种差距可能来源于不熟悉，也可能来源于误解等，但确实是一个不能忽视的因素。特别是对于毕业生或者刚刚开始工作这类还处在被评判位置上的人。

假如一个人身处任人唯亲的环境之中，那么表达力的作用会被无限放大，这时候个人价值的作用反倒会缩小。比较良好的情形是身处公平竞争的环境中，这时候表达力的作用会有相对清晰的边界。但由于每个人对公平都有属于自己的定义，所谓的公平竞争也只是相对的公平，表达力的作用并不会减少为 0。

- 自身价值的稀缺性

假如开发某一产品时，必须某项技术，而这一技术只有某人才掌握，这时这个人的价值会被高估。这时起作用的影响因素可以称之为稀缺性。稀缺性存在与否即取决于自身高度，也取决于大势，而对于后者而言，个人改变余地很小，大多时候只能顺应。

比如说：在 IT 行业刚开始的时候，产业本身急速膨胀，但程序员的人数却相对较少，这就导致 10 年前程序员的起点工资反倒很高。而 10 年之后，由于介入门槛一再拉低，大批人员涌入程序员队伍，起点的工资反倒下滑。这种行业的时间特征主要是稀缺性所造成的，反倒是很难讲 10 年前的程序员更加努力，水平更高，而今天的程序员就不努力，水平变差了。

- 公司的特质和未来

最后一个影响实现程度 A 的关键因素是公司自身的特性。公司更像是一个平台，往往对平台上的人所能达到的平均高度有所影响。假设两个人同样是很优秀的程序员，一个进了 Google 这样的公司，另一个进了一家外包公司，他们两个加入公司后都表现非常优秀，公司也对其非常认可，但从收入、技术高度等来看，这两个人却会有所差异，时间越长差异可能越大。这种差异并非努力所造成的，而是由于公司在生产链条上所处的位置不同而导致的。

接下来将依次对上面所提及的四个关键要素：自身价值、自身价值的表达力、自身价值的稀缺性和公司的特质和未来进行简要的分析。

§2.3 定律要素之一：自身价值

在金庸先生构建的武侠世界里，最犀利的杀伐武功应该是《独孤九剑》，但学会了独孤九剑却失了内功的令狐冲一样会被一堆无赖按到地上揍个鼻青脸肿。待到学会了吸星大法，内力大进，那就再没这回事了。

根据打铁还要自身硬的道理，修炼确实应该先从内功开始，但不是说招式就不必要了，这很辨证。至于究竟那个更重要这样的问题不在具体的时空背景下是没有答案的。

从最终表现来看，一个人的价值（或者说内功）可以体现在几乎无限多的地方，如：编程语言、分析设计、沟通、管理、估算、流程改善等等。但如果为各种技能稍做一点分类后就会发现，人创造价值的基本途径只有两个：一个是完全依赖于自身的技能，另一个则是假于他人之手。

后者也许不太容易理解，这里举一个简单的例子：假设张三和李四的技术能力都非常的优秀，但两者性格不合，一旦在一起工作就非常容易各自固守己见，寸步不让，每天争吵不断。这个时候王二出现了，王二的技术能力并不优秀，但比较擅长协调各个人的意见，有王二在，张三和李四就可以配合的比较好。在这种情形下，王二创造了价值，虽然很大程度上这种创造依赖于 A 和 B。

具体来讲，编码、设计、测试、估算、需求分析等更类似于直接通过自身的技能创造价值，而管理、流程改善等则类似于后者。专注于设计、编码、测试等实现工作的人在现实中往往被定义为程序员，而专注于管理、流程改善等方面的人在现实里则往往被定义为管理者。

从可创造价值大小的角度看，一般情形下两者似乎都没有一种准确限度：伟大的政治家和伟大的科学家可以同样的伟大。但从现实来看，至少是在国内，很多时候人们更倾向于认为管理类职位拥有更高的价值。现实中很多公司中确实如此，但这并不总是对的。

似乎可以在某一条线上把公司安放在不同的位置上，越过某条界限之后，直接做事的人所创生的价值才可能超过管理人员。这反过来要求岗位的技术附加值要比较高。想象下在制造类企业中，从收入的角度看，生产线上的工人的收入总是会偏低。同理，在软件公司中，越是技术含量低的开发工作，这点体现的越为明显。而一旦到了顶级软件公司中故事就可能会不一样了。

在《微软的秘密》这本书中有这样一段描述：

我们中有些人对开发人员怀有极度的不满，那简直就是嫉妒。达瑞尔•希文斯是 Windows NT 的主要开发人员之一，他有 9 辆保时捷，我当然也希望能够拥有 9 辆保时捷。但我怨恨达瑞尔么？当然不。他绝对受之无愧，他真的棒极了。如果用我的支票来给他开工资，我也愿意。不过从长远来看，过一两年你肯定就能得到应得的报酬。如果由于某些原因，我们引入了达瑞尔，并付给他足够买 9 辆保时捷的薪水，而他一旦干得不够好，就不会在这里工作很久... ...但这些开发人员都是精心挑选出来的人才.....唯一的不利之处在于，总有一部分人感到开发人员才是“主角”，但这正是这一行业的本质。

持上述看法的人是 Windows NT 的高级产品经理理查德•巴斯，从描述来看，这个管理者的收入是要低于被管理的技术人员的。

促成这一结果的三个关键词是：微软、Windows NT、主要开发人员。微软抓住天时雄霸 PC 的操作系统市场，而在操作系统中 Windows NT 正是核心，在前两者基础上达瑞尔•希文斯又是主要的开发人员。为使技术人员的收入超过管理人员的，这三个因素恐怕是缺一不可。这在以销售为核心的公司里很难复制，但在以产品为核心的公司里却始终存在着可能性。

单以方向自身而言，很难单纯讲那条是金光大道，而那条是华容道。关键点首先在于要避免选择自身内部蕴含矛盾，比如你想做技术却选了不以技术为核心支撑的公司，选了管理，却在纯研发环境里。其次在于无论那条路上都要努力走到专业和高端。

而不管技术还是管理，影响增值整个过程的因素都比较多：知识体系上的认知、方向的选择、博与专的平衡、知识的可流动性等都要分别进行考察。对这些点的考察将在第四章中逐渐展开。

自身价值也是公司命运的根本

如果要选过去十年 IT 世界里的大事件，我想有两个事情一定会被选上，一是 Google 的崛起；一是苹果从跌到中爬起来。苹果的价值比较直观，姑且不论，那 Google 为什么能够崛起？

搜索引擎这东西，Google 既不是第一家做也不是最后一家做，那为什么只有 Google 站到了最高点？

对此每个人都会有自己不同的解读，但我个人在读完吴军先生的《浪潮之巅》后最深

的感觉就是 Google 内功了得，说回来还是上面那句老话---打铁还需自身硬。我们来看几个《浪潮之巅》中记录的细节：

- *雄才大略的领袖*

对此吴军先生评价说：我总是对别人讲，比尔•盖茨使我们信息时代的拿破仑。如果要问我谁是盖茨的继承人，我认为不是微软的鲍尔默或其他人，而是 Google 的佩奇和布林。

其认知、行事也确实佐证了吴军先生的评价，这点可以体现在几个具体的事情上。

第一个是在他们的 Google 还没有挣钱甚至还没有成立时，两个人就体会到在互联网时代，让所有人很容易免费上网并方便的找到自己想到的东西的公司，必将成为互联网时代之王。

第二个是他们认识到，只有把互联网的内容送到千家万户就行了，至于互联网的内容是谁的并不重要。

第三个是他们创立了 Google AdWords 的商业模式。这个工作是佩奇亲自领导的。

第四个是他们努力削减中间环节，直接面对最终用户。

这在今天看来，也就普通，但在九几年就认识到这些，并能准确的达成目标，确实可以用雄才大略来形容。

不过末了说一句，再怎么雄才大略，在中国也没玩转，并且是国外的互联网大佬们没一个玩的转的，这才有腾讯系，百度系的崛起，这事情很有挖掘、考察的价值，但由于超出了本书范畴，不再深谈。

- *猛将如云*

根据吴军先生的记述，Adwords 由萨拉主导开发，最让然吃惊的是这套系统只有 10 几个人，在很短的时间就搞定了，而雅虎和微软花了十几倍的人力和几倍的时间也没搞好。

而著名的 GFS 和 MapReduce，据说在 Google 从设计到实现一共就三四个人干了两年。

更夸张的是据说搜索引擎的第一个版本完全是一个叫克雷格•希尔弗斯坦的人搞定的，这个人总是能用最优的实现方式，多少年后读起来其代码仍然是非常优美。

不只是工程师，据说销售队伍也都是精兵强将。Google 的销售副总裁，带领不到 10 人的销售队伍，成功的以刚成立的小公司的身份拿下了雅虎的搜索服务合同。

从这个视角看，修炼先从内功开始不管是在个人还是公司身上似乎是普适的，当然大家功法可能不一样，有人练独孤九剑，有人练葵花宝典，有人练吸星大法。

§2.4 定律要素之二：自身价值上的表达力

很多人咋一看这个标题，也许会有疑问：假如说是一个很牛很牛的大侠，那还需要表达力么？

实际情形是，如果大侠总是猫在山洞里隐居，那么有没有表达力其实一点都不关键，但现在关键的是江湖需要表达力，所以大侠一入江湖，就变的需要表达力了。

一说到表达力，很多人就会想到沟通和说话，但其实说话远不是表达力的核心。敏于事讷于言的人很多，难道他们就没有表达力了么？显然不是的。一个人的过往、行止、习惯、性格等都是表达力的一部分。

我们先来看一个简单的例子。

2012 的 CSDN 上有一篇翻译的文章，叫“编程的技能和做员工的技能--那个更重要？”这篇文章里描述了两个极端的例子：

Rodrigo 毕业于麻省理工，他在业余时间开发编译器。他是 Haskell 语言的核心代码捐助者，他开发了很多非常有名的 Python 程序库。他写出的代码都是非常健壮的代码，可读性好，能够优雅的处理各种程序上的临界计算场景。然而，他通常是拖延几天才回复邮件，你很少见他会接听电话，他看起来并不真正理解按时完成任务的重要性，他按自己的方式做事，你不可能弄清楚他究竟是怎么想的，只感觉他脑子里都是一些漫无边际的想法。

Gabriella 并不是一个非常优秀的程序员，她写的程序看起来显然很业余。15 到 20 行就能完成的程序她写了 30 行。她的程序里有 bug，这让 QA 部门在上面花费了不少时间，她没有真正理解写出的代码应该具有好的性能的道理——“能用就行啦！”。然而，她很热情——她收到邮件几分钟内必给予回复，她从不漏接一个电话，她善于沟通，她能把复杂的技术问题清楚的讲给客户听，她从没有逾期完不成任务，她不断的寻求反馈来改进自己的工作，她是一个很随和的人，同事喜欢跟她说话。

这两个极端的例子很有意思，但如果我们真的二选一的去判断那个更重要，就会失去领会职场中一个本质问题的机会。

文中所描述的做员工的技巧事实上很像拱猪游戏里面的梅花 10（变压器），他并非与所谓的编程技能相对立，而是普遍存在于每一个程序员的身上，任何一个程序员必然同时具备这两方面的能力：编程技能与做员工的技能，而做员工的技能则像一个变压器，最终放大或缩小你的真实能力。这就是表达力的功效，而做员工的技巧正是表达力的一部分。

那表达力为什么会有价值？

我们都知道管中窥豹是不好的，但很不幸即使是在最为公正理智的组织里，大多数人仍然是被管中窥豹的。企业的组织结构基本上呈现为金字塔形状，而位置越往上，权柄越大，也即是说位置在下面的人，其评判权利掌握在其上司手中。

而当上位者对下位者进行认知时，上位者印象中的某个人和真实的某个人往往会有差异。而好的组织和不好的组织的一个区别则是这种偏差究竟是主观造成的，还是客观现实而无法避免，而绝不是这种区别是否存在。

这种差异得以存在的客观原因有很多，比如：

- 信息丢失

  层级一旦产生，信息往往需要中转，总经理要想看到某个人，往往要通过几个层级，这个过程中无疑的信息会被丢失。

- 信息量过大

  一个人能处理的信息是有限的。比如一个 Manager 负责一个 20 个人的团队，那么由于待处理的信息过多，就就很可能在是推卸责任还是陈述困难上产生误判。

- 语言

  即使是信息没丢失，不同的人对同样的信息理解也可能不一致。比如说 V 手势在英国就意味着滚开而不是胜利。

这类因素最终导致认知上的偏差成为一种无法规避的客观现实，是一种必须去适应而

无法彻底改变的东西。像组织行为学这类学科中会把这个问题单独作为一个研究项目：印象管理（impression management)，首因效应等探讨的都是这个事情。

这并不难理解，通过自我推销、赞扬别人、适当的从众、搞好人际关系这类印象管理的手段来管理个人表现面无疑的会让自己产生溢价，提升自己在别人眼中的价值。

毕竟在组织里，别人眼中的你才有现实意义，即使它和真实的你有所差异。从长期的视角来看，影响自身价值表达的几个主要因素是：资历、自身性格特征、借势的程度以及权术的运用等。这几点将在第五章进行展开说明。

最后需要做一点区别的是改善表达力与恶意专营。

两者间本质上并无差别，有差别的是程度。从适用场景来看，在任人唯亲的环境里曲意逢迎是一种生存必备技能，但即使在最公平的组织里也需要改善自己的表达力。

年轻的程序员往往会仇恨上面所说的这点，并用充满负面情绪的词汇去形容这类技能比如：拍马屁，无耻。但其实不是，从人生效能的角度看，忽视这点是危险的，除非你在自身价值上已经达到了众人瞩目的地步，比如：简历上就一句话，我创造了 Python。

欠缺表达力的历史故事：弹铗而歌

《战国策》和《史记》里都讲述了这样一个故事：

冯谖因为太穷而无法生活，就申请成为孟尝君的门客，但当孟尝君问他有什么本事时，冯谖却回答说自己没什么本事。

结果孟尝君虽然吸纳了他，但冯谖却被安排为最下等的门客。

孟尝君的门客有三个等级：一等门客出门有车坐，二等门客有鱼吃，三等门客只能吃粗劣的饭菜。

冯谖并不很满意，就弹自己的剑而做歌，说：长剑啊，我们回去吧！没有鱼吃。

下人把这事儿告诉了孟尝君，孟尝君还是很大度，说：那就给他鱼吃。

过一阵，冯谖又开始弹自己的剑而做歌，说：长剑啊，我们回去吧！出门没车坐。

左右的人取笑他之后，又把这消息告诉了孟尝君，孟尝君又很大度，说：给他车座。

接下来，冯谖继续弹，还是这个调子，说：长剑啊，我们回去吧！没法养家。

这时候大家已经很厌恶他了，但孟尝君还是问了他的困难，并派人给把他母亲也养了起来。

接下来冯谖连续做了几件很体现自己远见卓识的事情。

第一件是当他申请替孟尝君到自己领地上收债的时候，他把债条都给烧了。理由很简单，孟尝君家里啥都不缺，就缺人心归附。这一举措，在孟尝君被罢黜时，给孟尝君提供了东山再起的缓冲。

第二件是他去忽悠魏王，说齐国强盛都是因为孟尝君，现在他被罢黜了，如果能为魏国效力，那么魏国富国强兵指日可待。魏王听了后，就派使臣携重礼，三次延请孟尝君到魏国为相。齐王一看，这可不得了，孟尝君确实是人才，要不然魏国怎么会这么劳师动众来请他。孟尝君因此而得以恢复相位。

第三件是孟尝君恢复相位后，感叹说：以前那堆门客，我一落难就都跑了，现在我恢复了相位，他们有什么脸面来见我，谁要让我见到了，我一定呸他一脸。冯谖当即跪倒进行劝谏，说：富贵多士，贫贱寡友是自然规律，希望孟尝君能够遇客如故，潜台词是：你要这么干了，树敌不说，那还能有可用之人。孟尝君又听取了他的建议。

也就是说孟尝君很辉煌的一生和这个没事谈剑要东西的冯谖是分不开的。

也许有的程序员会感觉冯谖这样不挺好么，但在现代冯谖的做事方法实际上是取死之道，几乎百分百会被现存规则轰成灰灰。

从后来行事来看，冯谖无疑是有才华的，但他得以体现才华的机会完全依赖于时势而非是自己争取来的。

他所做的所有事情都是在看不到回报的时候多索取：要鱼、要车、要养家、从孟尝君

的角度看，这些可能连长线投资都算不上。因为在他要东西这个时间点，这个人本身有没有价值则完全没人知道---等价于无价值。在古代还有孟尝君，但在现代企业里，这么做落在周围人的眼里就是眼高手低，几乎一定会被开除掉。指望沧海横流方显英雄本色是不太行的，万一一辈子沧海也不横流呢。

但偏偏冯谖和很多程序员的行事风格还真的有点类似，很多程序员擅长做事但不擅长表达，再加上很多时候程序员收入不低，所以人生境地没准就真和冯谖早时有点类似。

§2.5 定律要素之三：自身价值的稀缺性

现实里，体现稀缺性的故事也很多。

在东北曾经发生过一场非常惨烈的战争，这场战争之所以惨烈，倒不是因为战斗，反倒是因为其中所使用的围困战术。当一城居民都处于饥饿状态时，馒头和黄金的比价就不断下跌，故老相传，即使还没到最后阶段，一个馒头已经可以换一个金戒指了。与之相对比，在今天假设一个黄金戒指是 4 克，那么其价值大概在 1500 元左右，大致等价于 3000 个馒头。

抛开人文关怀不论，这背后其实体现的是稀缺性对价格的巨大影响。

我们常说物以稀为贵，但其实在以交换为支撑的职场中人亦如此。某种技能的稀缺程度往往构成一种大的环境，进而使这个环境中的所有人产生溢价/折价。

有的程序员往往对如何用某个程序不太感兴趣，而认为把程序开发出来更体现价值。所以很多人在了解到 SAP 顾问这类职位的收入远高于一般程序员时，往往会感到震惊。单纯从技术难度看，成为编程高手似乎总是会比成为 SAP 顾问更花时间。这件事情找不到数据支撑，暂时还只是一种判断，但现实确实可能是你花了更多的时间，学了更难的技术，收入上却比某些做着看着相对简单工作的人低。

这大多时候是稀缺性所掀起的波澜。许多很让人纠结的问题与此有关，比如：为什么 Java 语言的程序员就比 C#语言的程序员收入高？

稀缺性本身取决于需求与供给，这样获得稀缺性就有两个主要的手段：

一是站在需求相对恒定，供给比较稀少的位置上；一是加入需求急速膨胀，而供给有限的场景下。前者很好理解，爱因斯坦总是稀缺且有价值的。这主要是因为社会总是需要优秀的科学家而达到爱因斯坦的高度又总是非常艰难。而后者则需要更多的一点说明。

如果在 90 年代加入程序员这个行业的话，那天生就会处于比较稀缺的位置，因为那个时候这个行业在国内刚刚兴起，需求极大，但程序员本身非常稀少。而随着教育机构开足马力，加大供给，再加上普通软件开发介入壁垒较低，在 10 年之后，单从量上看恐怕程序员已经处在供大于求的情势之下了。在我印象之中，2001 年 211 学校软件开发行业的毕业生的薪资水平基本达到税后 4000 元/月，而这一水平即使不考虑通货膨胀，直到今天也没有彻底恢复。

虽然没有权威机构进行具体测算，但在这 10 年里程序员增加 10 倍（比如：从 60 万增加到 600 万），那是毫不稀奇。因此可以讲在 10 年前，程序员是普遍稀缺的，而在 10 年后，这种情形就只在特定领域里了。

稀缺性的客观状态是一种大势，作为个人基本上不可能改变，只能做选择以对应将来。所以其背后主要隐藏的是在特定时间点做出恰当判断的问题。

我的一个同学曾经和我感叹，如果 2001 年毕业加入某家国内知名通信设备厂商，从收益的角度看，恐怕比读研要好的多。这样的例子可以提醒我们，稀缺性是有时效的。

对稀缺性的进一步考察将在第六章里展开。

稀缺性与选择权

在前言中曾经简略提到如果非要选一个指标来描述一个人的成功的话，选择权可能远比其他的指标更有用。当一个人买房子的时候，可以在多种房型中选择的人往往就比只能选择偏远小区的人更成功。

不考虑理想和自我实现这些与人生价值相关的角度的话，人生争的往往也就是选择权。自我增值、表达力的增强、稀缺性的营造骨子里都是在扩大一个人的选择权，让一个人可以有能力去选择企业而不总是被选择。而选择合适的公司则讲的是不要浪费自己好不容易争取来的选择权，并为下一步的选择权打下更好的基础。

选择权的争取远比想的残酷，当一个很聪明的人在很好的大学里因为挂科而无法准时毕业时，他的选择权将大幅缩水，很可能必须退回到三线城市谋生；当一个技术很有天分的毕业生错位的进入了一家体力密集型企业时，他的选择权也将大幅缩水，5 年后他可能一点竞争力也没有。

§2.6 定律要素之四：身处公司的特质和未来

当令狐冲以华山派大弟子的身份来到林平之的外公家时，虽然衣衫褴褛，神情萎靡可大家也不敢瞧不起他，给了 40 两银子做见面礼不算，喝酒还有主家关键人物相陪。一旦这些人发现令狐冲在华山派里不太受待见时，虽然令狐冲还是那个令狐冲，挑衅就来了。这体现的是帮派的力量。

在程序员的世界里帮派一样有力量，不过这时候帮派都不叫帮派，而改名叫公司了。

两个不熟悉的程序员一见面往往会互报家门，一听是微软、Google、阿里、腾讯等大公司出来的心里先高看三分；一听是个完全陌生的公司，倒不一定低看，但高看却是不可能的了。而公司作为一种平台，其力量则远不止体现在虚名上。

我认识的程序员中有这样一个人：

他喜欢钻研程序，不喜欢和人说话，做事特别认真，并细致耐心，你如果有一个任务，交到他手里那可以放一百八十个心了。当然，勤劳肯干的同时，他也有一些不知道算是优点还是缺点的特征，比如：并不是很有野心，不善表达，并不会主动去改变身边什么。

他所从事的行业偏向于驱动程序，供需相对比较稳定，稀缺性上并不会有很大变化。

他加入第一家公司后，在那里工作了 4 年多，但即使表现比较好，公司却处在下滑期，这导致他的年收入始终突不破 10 万。

虽然在换工作上他的惰性比较强，可最后迫于无奈终于他还是动了。换工作之后，工作内容并没有太大变更，但职位和收入提升总算踏上了正轨，2 年内的涨幅比过去 4 年还多。

对于上述这个程序员，他工作内容和个人价值本身并没有太大的改变，但收入本身却起了比较大的变化。

这里面的根本原因是因为公司提供的平台高度不同。对个人而言，公司是一个平台。平台如果天生高一点，那么一个人的起点也就会高一点。法律面前公司是平等的，但在实际上公司和公司间的差别可能比人与猴子的差别还要大。富士康和苹果都是公司，还都和 iPhone 有关，但即使只是看公开报道，我们也知道这两个公司内的生态一定差别巨大。富士康必须极度关注成本节约，而苹果则需要更多的创新能力

如果要给公司进行归类，那么不同视角会得出不同的结果：

如果把生产链条比作食物链，那么可以区分不同公司在食物链上的不同位置。

如果从公司的存在历史来看，则可以把公司分类为新创立的公司，成长中的公司，成熟的公司和衰落中的公司。

如果从公司的文化特征来看，则可以把公司分为相对公平的公司和不公平的公司。

如果从地域来看，则可以把公司分为大都市里的公司和发达二级城市的公司以及其它的公司。

这类分类有助于我们认识公司的特质，并进行更为理智的选择，当然你要有选择权才行。

对公司特质的考察将在第七章来具体展开。

组织是利益分配的基本单位

单只把一群人圈在一起并不足以形成组织，一群人有类似的价值观，有特定的行为规则，有共同的努力目标，那才能够形成组织。组织往往表现为一种个人之外的约束，所以很多人并不喜欢它。但确实是有组织才有力量，个人在组织勉强力量往往非常渺小，这进一步导致组织是利益分配的单位。

那为什么说有组织才有力量？

这起源于人类的两个个基本特质：一是精神差异无限大，但肉体的差异则相对有限；一是欲望的无边界特质。第一个基本特质内含着一种矛盾，即是一个精神强大的人其事实可以驱动的力量远大于其自身可以拥有的。这时候在第二个基本特质的驱动下，人们就必须结合起来才能达成自己的目标。结合之后，在统一意志之下，组织中的个人才能互补，形成远超个体的力量。虽然组织内部耗散往往非常严重，但相对于个人往往已是类似超人的存在。霍布斯论文中曾经把国家描画为一个有无数人组成的持剑巨人，这是非常形象的。

那为什么组织是利益分配的单位？

想想一个人去和微软，Google，美国、日本对抗是什么结果。这些大型组织虽然可能决策缓慢、行动迟缓，但远不是个人所能对抗和竞争的。而利益格局必然与参与利益分配的个人和组织的强弱有着关联。作为结果，不管喜欢不喜欢，组织是利益分配的基本单位，在这之后，才是组织中的个人如何进行利益分配。在分配 iPhone 所带来的利润时，首先是苹果和富士康间的利润分配，接下来才是苹果内部和富士康内部。

§2.7 小结

在《史记•货殖列传》里太史公写过这样一段话：是以无财作力，少有斗智，既饶争时，此其大经也。

这段话里描述的三个层次则与我们上述描述的层次类似：谋求自身增值是无财作力的层次，增强表达力是少有斗智的层次，而注意稀缺性和公司特质则大致是既饶争时的层次。不同因素在人生的不同时期，这些因素的影响力不同，对于毕业生增值最重要，而对于一个 30 岁的技术上已经前行无路的程序员则很可能表达力更重要。但不管对于谁，最终这些因素是同时在起作用，它们最终会汇聚成一种无差别的量，这种无差别的量标识了一个人的高度。而这一高度决定了一个人在职场中的位置，而位置进一步决定了一个人的收入，自我实现程度等等。

那这种规律以及相应的影响因素究竟可以可以用来干什么？

用途其实很简单，这种种要素可以用来当镜子使，进而照见一个人的过去、现在和未来。

从过去照出失误，从现在照出短板，为未来照见道路。当人生陷入瓶颈并不愿意安于

现状时，就可以用这面镜子照照自己。当然为了使这面镜子更加的强大和光亮，后续章节里会需要为之补充更多的细节。

我们可以看一个简单且常见的例子，并用上面四个要素分析一下利弊得失：

工作中总是存在简单且重复的工作，极端的就是生产线，你可能只是需要伸缩一下手臂，只是每 3~4 秒就要来一次。在程序员的工作中不太会有生产线那么夸张，但确实也有简单重复性工作，比如单纯的拖控件、关联数据库或者例行设置大量配置信息等。

对这样一份工作应该如何进行分析？

平心而论，想彻底避免简单重复性工作是不可能的，但如果一项工作的主体部分体现为简单重复，并且不可能自动化，那么这种工作从增值的角度看是没有助益的，可归为技术路径短，不增加自身价值的工作。

稀缺性则依赖于所做工作的具体含义。比如：如果你所处理的大量配置信息深嵌在一个公司内部的各个角落（收集困难），各个配置项的含义并不是很好理解，这个工作对某家公司又特别有意义，那么稀缺性反倒可能很好，但可流动性就几乎没有，个人利益和公司利益将结合的无比紧密。这时候公司平台就变得很关键。

由于选择权已然变小（只能在特定的公司或领域），这个时候表达力就非常的重要，如果不能在表达力这样的活动上有所突破，那个人的未来就很难明朗。

这个四个要素总是同时起作用，但不同时期、不同情景下，权重会有比较大的变化，这点在后面会逐渐展开。微观来看，每个人的职场经历可能都很不相同，但骨子里起作用的始终都是这几个因素。

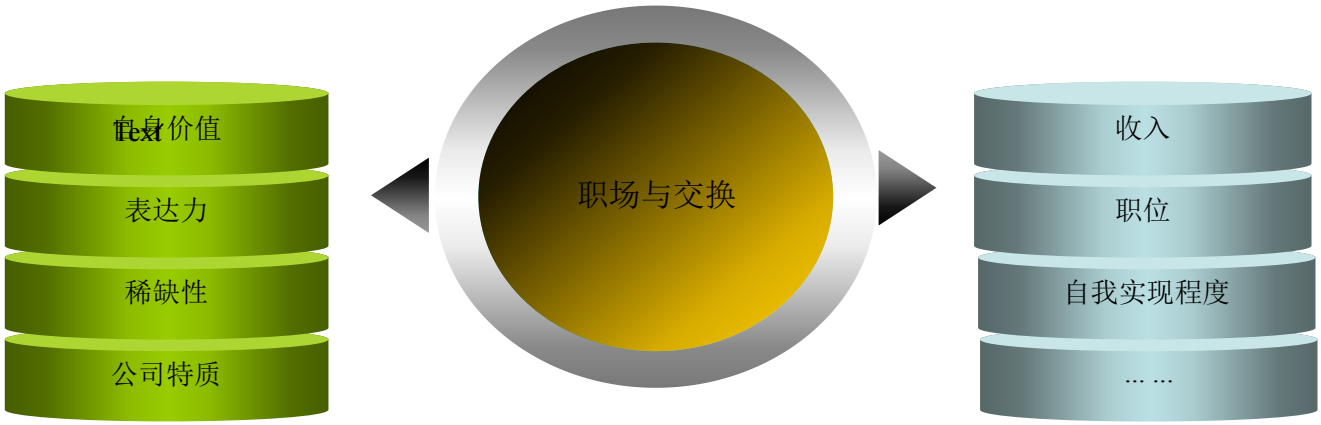


图 2-2：职场生存定律

时运也是一种力量

在这一章中，我们谈到了自身价值、表达力、稀缺性与公司，并认为这四个元素是左右人生轨迹的根本性力量，但实际上在此之外还有时运。时运绝对是左右人生的一种关键力量，一个无比优秀的人，如果 35 岁就因为意外死了，无疑人生高度会比较有限。但这太飘渺了，所以本书中不会主要考察这个维度上的力量。与时运相关的，主观上可以把我的事情主要是稀缺性，这点会在第六章考察。在此之外，时运往往还需要人能够耐心等待，这里对这点做点说明。

在唐诗里面有两首《唐诗三百首》《诗境浅说》这类诗集不太会选，但却很为大政治家喜欢的诗：一首是温庭筠的《经五丈原》，一首是罗隐的《筹笔驿》：

经五丈原
温庭筠
铁马云雕共绝尘，柳营高压汉宫春。
天清杀气屯关右，夜半妖星照渭滨。
下国卧龙空寤主，中原得鹿不由人。
象床宝帐无言语，从此谯周是老臣。

筹笔驿
罗隐
抛掷南阳为主忧，北征东讨尽良筹。
时来天地皆同力，运去英雄不自由。
千里山河轻孺子，两朝冠剑恨谯周。
唯余岩下多情水，犹解年年傍驿流。

据说某位震古烁今的大政治家在晚年常观“时来天地皆同力，运去英雄不自由”及“下国卧龙空寤主，中原得鹿不由人”两句而落泪。这两首诗都隐含着这样一种“时运不在大英雄亦是壮志难酬”的意味。通俗讲就是运道没来，怎么折腾也没用，位置越高，越受限于时运。

程序员的命运也受时运所左右，你可能能力优秀、思维活跃，性格鲜明，表达、技术也很优秀，但如果几次换工作却总是碰到呆板，喜欢老实人的领导，碰到平庸的公司，就可能一直不被欣赏。这种事情既不会是今天才有，也不会在明天就立刻消失。面对这类较差的时运，最需要的旺盛的斗志和耐性，干别的都没用，求神拜佛也不行。马化腾、马云这些大佬当年一样很折磨，如果马化腾在最痛苦的时候把腾讯卖了，那就不会有今天的每年 400 多亿收入，程序员也一样，这时候要有点但行好事莫问前程的心态。但这里有个陷阱，很多人会把个人的失误推诿到时运身上，麻醉自己，让自己心里好过点，这个是假的时运。

第三章 软件的世界是怎样的

规律是必须顺应而不能改变的，但除此之外现实中还有一些事实也是无法改变的，这两者都很像程序中的常量，想提高人生的高度则需要同时驾驭这两者，而不能试图为两者赋值。下面我们就一起来看一下，软件世界中只能顺应，而不能试图改变的特质有那些。

§3.1 技术更迭偏快

在学校里，动力机械类专业往往会学习一门叫工程热力学的课程，如果耐心翻阅就会发现虽然封皮换了，但这门课程现在的教科书和五几年的教科书其实差别不大，热力学第一定律还是那个热力学第一定律。

与之相对应《C#高级编程》这本书在 2005 年还是第三版，但到 2011 年已经出到了第七版，页数则从 1027 页增加到了 1473 页。

这看着是一个很小的不同，但实际上已经折射出了软件行业的一个根本特质：技术更迭、增加速度较快。

技术更迭较快说的是这样一种现象：今天有价值的，明天可能会贬值为 0。

在软件行业里，你所依赖的某一平台或语言很容易产生更迭。单以 Windows 平台而论，10 几年前很多人只有 Win32 API 好用，但一个人如果只停留在 Win32 API 里，是不太能适应今天的软件开发的---虽然没有官方统计，但感受上在今天 Web 开发、手机终端开发明显比 Windows 开发要火热。

这也许源自于这样一种现实，很多传统行业的技能直接依赖于某种自然规律，如：热力学、流体力学、材料力学等等。这些东西自身只会深化或细化，比如从牛顿定律到相对论，但很少会有颠覆性变化。但软件开发所需的东西(API 等)往往依赖于某一个公司或组织，比如微软、苹果等，进而是一种人造系统。一旦社会基本需求发生变化，这些公司或组织就必需不断的抛弃并更新自己的系统，比如：GDI -->GDI+ -->WPF。

同时一旦公司因为某种原因倒闭，这一公司所支撑的技术也会变得淹没无闻。

1995 前后开始从事这个行业的人很多都会知道 Delphi，但我估计 2005 后加入这个行业的人就会对这个东西感觉陌生了。我们很难去深究原因，但至少现象上来看，Delphi 这样的开发平台随着 Borland 一起远去了。当然，与之一起远去的还有 Delphi 世界里的很多牛人。

极端来讲，如果 Windows 彻底打输了当前移动终端这场战争，那么靠 Windows 吃饭的人（包括研究 Native API 的和研究.net framework 的）无疑的都有贬值的风险。

可以打一个比方来使这种差异更形象一点：

好比说两个不同的人，一个在传统行业一个在软件行业，两个人都很勤奋，不停的往自己脚下垫东西，努力使自己达到更高的位置。传统行业中的人比较自然的会越垫越高，而软件行业中的人则会垫到一定时候，突然间某几块砖就会消失了。

这倒并不意味着软件行业中并非没有具有较长生命价值的东西，但这些东西往往集中在一些特定的领域里，牵涉的从业人员比较少因此不太具有代表性。

具有长久价值的东西里面最典型的东西是通用数据结构和算法，今天的排序算法在 10 年后必然同样具有价值，但专门从事算法优化改良的毕竟是少数。可以讲大部分人群还是处在技术更迭的大潮之中。此外，图形算法、分析设计方法等也具有稳定且长久的价值。形象

来讲似乎越抽象、越偏向于研究的东西其价值越长久，而越具体、越立刻可用的东西其时效性就越强。

这一基本特质的影响非常深远，甚至引出了学习可能会产生较大负效应这类比较特别的问题，这点将在后续章节中陆续有所陈述。

技术更迭的一个实例

为了让读者对技术更迭有一个更直观的印象，我们来看一下袁峰先生所著的《Windows图形编程》的目录，并看一下这本书里那些东西在过去的 10 年里被更迭掉了，而那些没有？

其中第一章，第四章牵涉的是一些基础知识，比如 Windows 基本结构、如何 Hook API 等，因此虽然部分内容有点过时，主体上仍然是有现实意义的。

第十章、第十一章、第十二章主要和位图格式相关，而位图格式变化不大，所以这几章的主体部分仍然是有现实意义的。

第十四章主要讲的是字体，而 Truetype 字体即使在今天也是字体的主流，因此也还是

有现实意义的。

其他的章节则因为主要是和 GDI 相关联大致上是过时了（不意味着完全没用），也就是说 18 个章节里只有 6 个章节还有较大的现实意义。

这本书在国内的出版时间是 2002 年，到 2012 年正好是间隔 10 年，10 年时间淘汰了某一类技术差不多 80%的内容。不知道还有那个行业会有这种淘汰率。

如果任何人以为书里被淘汰的那 80%的内容容易学，那就错了，在当年即使是有 Windows 基础编程知识的人（知道线程、消息机制等）把这部分知识搞通至少也需要 1 年（工作后）。

§3.2 介入门槛偏低

某一次喝酒的时候和几个朋友闲聊，谈到了自己的专业：

有的说我是学物理的，和核能有关系。

有的说我是学涡轮机的，这是主要动力机械，发电厂常用。

有的说我是学变压器的，负责把电送出去。

听了之后，其中一人大笑，说：你们几个拼起来就是一个发电站，纯属打入软件队伍的杂牌。

虽然看不到具体数字，但就日常感受来看软件行业中来自其他专业的人似乎确实偏多。这反过来就不成立，你很少听说学软件的跑去做数学了。

这背后隐含的是这样一个事实：软件行业介入的门槛相对比较低。虽然做到高处，很难讲软件就好做，机械就难做，但从介入壁垒来看，确实是软件行业偏低。

如果去做动力机械，那么要学习工程热力学、传热学、材料力学这样的课程，但如果要做软件开发，那么学好一门编程语言以及对应的 IDE 已经可以开始工作了。

当然，后劲不足可能会把不思进取的软件开发人员限制在某个范围内，比如说只能做应用级的开发，最终让他们等待淘汰。

软件的这个特质，也导致了软件开发人员所特有的一些问题，比如：如果自身没有突破，那么很容易就会被海量的后来者赶上。这点的影响也将在后续章节里陆续提到。

门槛可以低到什么程度？

以著名的北大青鸟为例，其公开数据是累计培养了 50 万 IT 人才，均摊到 10 年里，这么一家培训机构每年就可以提供大概 5 万人。当然这其中不都是程序员，但从北大青鸟的角色来看，其中的主体部分是程序员。而国内的培训机构则远不止北大青鸟一家。这是量的视角。

如果你再细心去关注北大青鸟公开出来的故事，你就会发现，高考落榜者、酒店保安、流水线工人都在介入这个行业。这里并没有一点歧视任何人出身的意思，而只是想说，这个行业的介入门槛相对比较低。

而同时我们也很难讲，只有做编译器的、文件系统、MapReduce 的才是程序员。也许有的人做的工作更难，而有的人做的工作则相对容易，但不管怎么样，大家确实是属于同一个行业，都叫程序员。

§3.3 软件和软件差别可以很大

我在《完美软件开发：方法与逻辑》这本书里曾经写过一段有点抽象的话：

从特质上来看，既然软件是固化的思维，那就必然同时具备**思维**以及**思维所承载之物**之特质。

- 思维的特质是指：思维的澄清通常是渐进的，思维自身是不可度量的，思维的主体一定是人，思维通常由概念和逻辑组成，思维的无边界化（灵活易变）这样的特质。这部分特质是共通部分，同时属于所有软件。

- 思维承载之物之特质是指：当思维的对象是数学的时候，思维本身也就具备了数学的特质；当思维的对象是商业逻辑的时候，思维自身也就具备了商业逻辑的特质。

既然思维自身的特质是复合的，那么作为固化思维的软件，其特质必然也是复合的：

既有属于所有软件的共同特质，也有特属于某类软件，甚至同其他类软件完全相反的独有特质。

上述文字主要想强调的是虽然都是软件但软件 A 和软件 B 可以有相似部分，但差异可能更大。一个人可能研究 OCR 算法好几年最终只写几百行代码，完全不需要用什么面相对象和设计模式，但在信息管理系统中一个人一两天内可能就需要写几百行代码。这两者虽然有巨大差异，但实际上都会被称作软件。

这种特质导致了软件开发所需要的知识日益的分化，最终结果就是不同软件领域差别很大。想用唯一的知识体系覆盖所有的软件类别变的非常困难。

对方法论而言，基于这一点最关键的一个引申结论是：任何一种方法论不只要陈述自己的方法，还要陈述自己方法的适用边界。

对个人发展而言，那就意味着要关注知识的可流动性这类问题。可流动性是说，你在 A 类软件中可能达到了一定高度，但如果穿越到了 B 类软件的领域中，可能江湖地位会一下子下降很多。

通俗的说法是：男怕入错行，不同的软件种类也勉强可以被看做是不同的行业，虽然他们都用一个词：软件来概括。

这一特质也带来很多非常典型的问题，比如：学习必须聚焦。这点的影响也将在后续章节里陆续提到。

多内部分野可以多到什么程度？

要想对多内部分野这一点有个直观感觉，最直接的方法是去看招聘广告。

有以语言来区分职位的：.net 开发工程师、C++软件开发工程师、PHP 开发工程师、Java 工程师等。

有以平台来区分职位的：Android 开发工程师、iPhone 游戏软件开发工程师等。

有以领域来区分职位的：GIS 数据工程师、金融项目软件工程师、电子商务软件工程师等等。

接下来还会有各种交叉，比如：Java 软件工程师（金融）等。

这里面未必没有重叠，但大致上来讲很难在彼此间穿越，年头越多穿越越难。

如果觉得这个分类不是很系统，那么可以参照软件工程中对软件的分类，再乘上平台

和编程语言就可以切分出大致不同的领域：

- 航空电子
- 应用系统
- 命令与控制
- 嵌入式系统
- 微代码
- Web 应用
- 科学研究和工程研究
- 实时系统
- 驱动程序
- 电信软件
- ... ...

最极端的情形是也不用分什么软件种类，一个项目的整个生命周期就能耗尽一个人一生中大部的能量。

§3.4 小结

软件的世界里还有许多其它的特征，但和个人发展相关且无法改变主要是这里提到的三点：技术更迭较快、介入门槛相对较低、不同软件差别极大。这三者类似于全局常量，而个人努力则类似于变量，它们共同在生存定律之下起作用，影响人生的最终高度。从自我增值的角度看，最关键的事情是不能与这三者所带来的效应相违背。

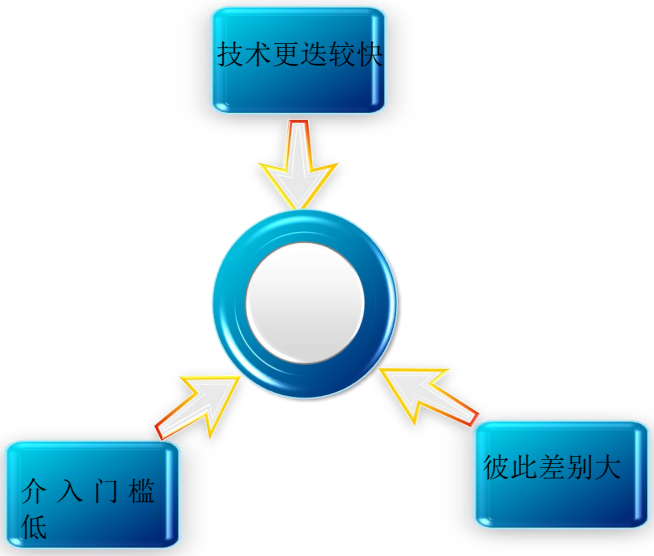


图 3-1：软件世界的根本特征

要不要转行到软件行业？

前面曾经提到过软件行业介入门槛相对较低，其他专业的人相对比较容易转行到软件行业，那其他专业的人究竟要不要转行到软件行业？

答案是原则上不要，除非有特别的理由，比如：自己的专业实在找不到工作，实在是做不了自己原来的工作等等。

很多人转行做软件主要的原因是在 2000 年左右软件行业突然爆发，平均收入水平远高于其他行业。但这是由于行业新兴起，注入资本较多，程序员需求大于供给而导致的。这些年来，各大学校和培训机构开足马力进行软件工程师的培训，大环境已经有了很大的变化。单纯的转行会意味着自己需要在并没有积累的行业中脱颖而出，这并不划算。优秀的自动化工程师和优秀的软件工程师同样是稀缺且有价值的，在这样的一种前提下，抛弃自己的原有专业会导致价值降低，是不划算的。

但有一种特别的情形，转行到软件行业比较有优势。假设一个人并不想始终打工，而希望开创自己的一番事业，这个时候转行到软件行业就比较值得。个人创业的话大多时候并不适合切入到链条长、资本密集或者劳力密集的行业，而适合进入链条短，智力密集型的行业。想象一下，重型机械生产、物流、纺织显然不适合一般人创业，而软件行业尤其是互联网则不同，只要产品合适坐在家里也可以产生收入。Instagram 靠 10 几个人做出了 10 亿美元的市值，这在传统行业中是不可想象的。

第四章 程序员的增值之路

在知道职场的基本定律以及软件世界的根本特征之后，就可以比较具体的去考察究竟应该怎么样具体的设计并实现自己的进阶路径。

在上面所谈到的四个要素中，对程序员而言最根本的始终是自己的价值，忽视自身价值提升而单纯沉迷于成功学这类方法论，早晚会吃到苦果。这点要一直记在心中。

在下面我们将把增值之路分解为三个根本步骤并分别进行讨论：如何选定自己的方向、如何开始自己的学习、如何持续的进阶并成为高手。

§4.1 方向的选择：技术还是管理

一个程序员在考虑增值时无法回避的一个根本问题是到底是做技术还是做管理。当然也有些职位会介于两者之间比如架构师，但我们暂时不去做细分，而是用简单的二分法。

这种基本方向上的选择对后续很多细节上的取舍有关键影响，所以在考虑其他之前，最好先回答一下这个问题。这就和修炼时要选择少林、武当、华山还是魔教一样，一旦选择，基本上是回不了头。

当然选择管理不意味着不需要掌握编程技能，毕竟当下大多公司还是信奉“宰相拔于州郡，将军起于行伍”的。但当技术达到一定水平后，管理还是技术这种方向性的选择将对下一步做什么有比较大的影响。在考虑那个方向前，则要先弄清楚管理和技术的关键差异。

4.1.1 技术与管理的关键差异

到了 30 几岁后，转为管理人员的程序员经常会调侃自己的技术能力：当年解决这种有时出、有时不出的 Bug 时，我常常在其前后都加几条调试输出，这招很管用很可能立刻就把它搞定了。结果多年后维护这代码的人困惑了，还来问我，这句为啥不能去掉，看着也没用啊，其实我也不知道，只能说运气和人品在程序里也是很有影响力的。

这是管理人员的一种真实写照，大家都知道，一旦走上管理岗位，那就和 ppt 越走越近，和代码越走越远了。虽然他仍然要跟踪最新技术的动向，但他很可能已经无法深究很多技术细节了。

据说微软这样的公司推崇一个人要想走上管理岗位，那要先把自己的代码用远少于别人的时间写好，省下来的时间才用来做管理工作。这很好，也不是完全不可能，但大多时候很难，需要很强大的天分，大多数人是做不到的。

主要原因是管理和技术所要处理的问题有根本上的差异。

管理者往往需要处理许多与人相关的事情，这导致要处理的事情是碎片化的，如果坚持编码，那么每天的打断往往会大幅降低写代码的效能，大家都知道编码是需要专注的。

管理工作总是需要面对大量的琐碎工作的，比如：老板对项目不满要赶紧去说明，免得发酵成大问题；人力缺了要赶紧协调，一是要能要到人，关键还得能要到合适的人；工具缺了，要赶紧购买；兄弟们有情绪了，要赶紧安抚；PPQA 了有抱怨了，要赶紧改正。如果工作进一步泛化，还要涉及到预算、评估、职业路径规划等。

我们很难让这些事情按照自己的节奏发生，如果管理人员做编程，最终这些都会变成

一种对编程工作的随机性干扰。所以一般来讲很难把它们很好的与编码结合在一起。想象一下，一个管理人员负责某个项目中影响关键路径的某个模块，接下来上面所列的意外发生了，那这个管理者怎么办？

唱歌的时候常说到 Key 或者调门这个词。同样是《花心》这首歌，周华健的用的 Key 和原本的冲绳民谣《花》的就不同，这导致两首歌听起来差别就很大，完全不一个感觉。也许可以说管理也是一种技术，但管理和设计编码这种技术的 Key 不一样。做技术需要面对的是程序，程序是讲道理的，Stack Overflow 时它一定会崩溃；而做管理时需要考虑技术因素，但更需要面对的是各种人，人则只在一定程度上讲道理，所以管理不只是一种技术。因此基本上可以认为管理和技术时完全不同的两个方向。

如果大家细心观察周围，就会发现，做技术（编码）的往往可以转去做管理，但做管理的再转回做技术（编码）就难了。这意味着技术背景对做管理往是很有帮助的，而管理背景对做技术则几乎没用。

了解到这种差异后，要想做出自己的那份选择，还需要考虑三件事情：一是既定环境下技术路径究竟有多长，也就是说做技术有前途么；一是个人的性格适不适合做管理工作；一是做管理工作可能会有什么负面影响。这三点将在接下来的三个小节中分别进行探讨。

4.1.2 技术路径长短对前途的影响

程序员往往自嘲自己是“码农”，不知道这词是那里出来的，但听起来“码农”和“农民工”已经有点近似了。而“农民工”往往是收入低，工作时间长的代名词。这就折射出了一个很尴尬的事实，在很多公司中，单纯从收入的角度来看管理职位是要高于纯粹的技术岗位的。

这并非是一个绝对规则，前文就曾经提到早在 20 年前，微软的超级程序员就可以拥有比管理人员更高的工资，可以拥有多辆保时捷。但在技术路径短的公司里，管理人员收入偏高这事情却具有必然性。

当一个公司的核心技术并没有创生多大价值，而是需要靠人力规模、商业模式等来支撑业务的时候，那么我们可以称之为技术路径短的公司。想象一下，如果一家公司专门承接本地化工作，那么也许也会需要程序员编制某些工具，但对程序员而言技术路径无疑是短的。

如果暂时把眼光从程序的世界移开，那么事情就可以看得更清楚。

在盖楼的时候，只要达到基本的质量，一个人每天砌 200 块砖，固然比砌 100 块要好的多，但相对于大楼而言，多砌 100 块砖，所多带来的价值有限。再进一步由于砌每块砖的价值是固定的，同时一个人每天所能砌的砖也是有限度的，这就会导致砌砖工人，不管多么努力，其收入水平必然会被限制到某一个较低的水平，只要他的工作还只是砌砖。这种限度是由这一工作的内涵所决定的，倒不是谁遭到了歧视。

再类比到软件行业里，单纯的在既定接口下实现已定义的业务逻辑就是技术路径比较短的工作，是体力密集型的；而分析业务逻辑，控制整体架构或者去研究 TTS 的算法则是智力密集型的，技术路径较长。

在选择方向时关键要避免的是选择了技术方向，但身处的现实中技术方向却路径较短，或者喜欢管理但跑到了纯粹技术流的公司里，这种选择其内部所蕴含的矛盾会给当事人的人生造成极大的困扰。比如说开发小型信息管理系统时，其所需要的技术含量并不高，公司的

主营如果是这个，单纯的做技术可能会直接影响收入。这是一个需要考虑的很现实的事情。

4.1.3 什么样的程序员适合转管理

《黑客帝国》的动画片中有一集叫做“Matriculated”，在这一集里有个机器人被逮住后，人类通过各种场景让他相信自己是个人类，计划看似成功了，但实际却不是。这个动画的启示意义在于，先天带来的很多东西，比如性格等实在很难改变，更多时候选择顺应自己的天性比选择对抗更加明智。

从先天性格来看，确实有的人天生适合做管理多一点，有的人天生适合做技术多一点。

比如说：
有的程序员天生有点被动，不喜欢主动学习很多东西，不喜欢与人沟通，但对工作所直接关联的领域研究较深，做事情兢兢业业，一丝不苟。

有的程序员非常聪明，理解东西很快，但不愿意搭理别人，总感觉别人水平比较差，脾气也比较暴躁。
有的程序员精力充沛，对技术狂热，但并不仅局限于技术本身，有大局观，有理想，能坚持。
单从性格而论前两者都不太适合做管理工作的，一旦做了管理工作，接触各种性格的人，容易造成人际关系紧张，反倒对自己形成一定的压力，极端情形下就会精神失常。

单纯的因为收入而选择管理工作，并不总是明智的，你可能无法适应，反倒导致事业出现起伏---不要低估这点的影响，现实中非常多的人因为这种错位而使人生走入低谷，甚至生病。

在大五模型里用五个因素来考察人格特质：

外倾性（extroversion):

外倾者者倾向于喜欢群居，善于社交和自我决断。内倾者则比较内向，胆小害羞，安静少语。

随和性（agreeableness）:

高随和性的人是合作的，热情的和信赖他人的，低随和性的人是冷淡的，敌对的和不受欢迎的。

责任心(conscientiousness):

高责任心的人是负责的，有条不紊的，值得信赖的，持之以恒的。低责任心的人则容易精力分散，缺乏规划性，且不可信赖。

情绪稳定性(emotional stability):

积极的情绪稳定性者倾向于平和，自信；而消极情绪稳定性者（神经质的人）倾向于紧张，焦虑，失望和缺乏安全感。

经验开放性(Openness to experience):

开放性高的人富有创造性，凡事好奇，具有艺术的敏感性；开放性低的人则保守对熟悉的事物感到舒适和满足。

总的来看，外倾性和经验开放性好的人更适合走上管理岗位。

千万不要忽视这种错位的力量。金山的求伯君先生就直承自己不擅长做管理。他认为人的一生之中最关键的是对自己能够有所了解，不是说自己什么都能干，是万能的。在雷军走后的 4 年里，做 CEO 有些力不从心，快 50 岁的他精神压力太大，多次想退休，请雷军出山。最终求伯君先生在不到 50 岁的时候退出江湖，不知道是不是和这个有关。

当然很多人可能远走不到求伯君先生的高度，但终究类似，可以打个比方形容错位的中层管理者。上司和下属员工像两块板子，管理这门功夫没练好的话，中层管理者就被搓球了：上司说，你做的这叫什么事儿，脑子大大的坏了。下属说：你瞎答应什么，这事儿怎么做，我不干，要干你自己干，爱咋咋地。

管理这功夫练好了，情形就变了：上司尊重你的意见，下属把你视为旗帜。一处天堂，一处地狱，核心差别其实不大，根本还在天生的人格特质。待管理人群的特质也很有影响，但这是运气所管理的范畴。

是不是适合做管理者的简明判断方法

假设说团队里两个兄弟吵起来来了，你愿不愿意去调解？

假如有一个人脾气很坏你愿不愿意和他沟通，即使你不喜欢？

假如有一个人问题很多，你愿不愿意面对面批评他？

假如有一个人屡教不改，你愿不愿意采取直接的惩罚措施，那怕关系紧张？

这个列表还可以增长。一旦做管理工作，这类需要抛开个人视角，而从组织的视角去看待问题并行动的地方很多。

如果对这类问题的回答是否定的，那么最好是不要往管理的方向上走。

上面这几个问题，纯走技术道路的还可以作壁上观，但如果是发生在自己团队里，管理者却保持逃避的态度，那么管理者就失职了。

由于人的世界很复杂，所以期望坏的事情一件也不发生，那是不现实的。我个人感觉管理者面对这类事情的几率是 100%，区别是遇到多少件，而不是遇不遇得到。

其实故事到这里还没完，如果往深了考察，就会发现，即使一个人愿意去搞定吵架中的两个人，那还有你怎么去搞定，搞不搞得定的问题。

捣糨糊、各打五十大板这类简单粗暴的方法往往只能有效于一时，等价于埋下定时炸弹，长线来看不是什么高明方法。但把这个展开就需要另外一本书，这里就不进行展开了。

4.1.4 管理工作的负效应

从日常很多人发表的言论来看，管理工作似乎被无限美化了，很多人都认为管理工作似乎是一条彻底金光大道，但这并不完全正确。为了让事情回归本来面目，这里说一点管理方所可能带来的负效应。

同纯技术工作相比，管理工作（特别是中层管理）的可流动性可能会非常低，形象来讲很多公司并不会愿意请外来的中层管理者来管理已有的员工，而更愿意请技术上有专长的人来解决具体的问题。这是由管理工作的几个特质所决定的：

管理工作和人打交道比较多，所以对人员的特质有很强的依赖性。如果一个团队的人都非常像机器人，那么在不同公司间管理技能是完全通用的---只要有 PMP，CMMI 这类东西就够了。但关键问题是人员的特性是多样的，这导致管理人员和被管理人员需要较多的磨合和适应。形象点讲就是，如果无法搞定特定人群，你考 5 个 PMP 证书，该不管用还是不管用。

同时长时间在管理岗位的话，即使是做技术出身，技术能力也会退化，沟通技能、与上级的信任程度反倒会提高。而这些东西，到一家新公司后，一定会被归零，，其价值并不明显。反倒不如擅长算法，擅长某类业务的技术人员可流动性好。

这也就意味着，管理人员往往与公司的利益绑定的更紧。尤其是中层管理人员，达到一定年纪后（比如：40 岁），很可能会失去流动的可能性，一旦所处的公司出现问题，那就可能会面临非常尴尬的局面---直接讲就是，如果你选择了管理方向，却缺乏相应的人脉，35 岁之后基本不具备可流动性，换工作会很难，至少比纯技术的高端人员难。

这点的一个旁证是各个初创期公司的人员构成。如果你用心观察就会发现对于初创期的公司而言，它需要创始人把握方向和寻找资金，也需要工程师来完成具体事务，但不太需要中层管理人员。比如：Pinterest 曾经公开了自己的数据，在 2010 年是 2 个创始人，1 个工程师；2011 是 3 个工程师；2012 年是 6 个工程师；2013 年是 40 个工程师。这种情况下，只有到 2013 年后中层管理人员才有存在价值，而一般情形而言这种情况并不会社招，而是会从现有人员中选拔。这最终导致纯管理人员的可流动性并没有想的那么好。

当然什么事情都有例外，如果你是成功运作几个产品的产品经理，那么也可不在流动性上受到限制。因为那些产品就是你最好的名片，他们使你在江湖里有了一席之地。

4.1.5 小结

考虑上述三个方面，大多时候可以判明自己是应该做技术还是做管理。比如说：如果一个人日常很容易和人产生冲突，但脑子很好使，也能静下心来钻研技术。这种情形大致上应该努力找一家技术路径长的公司做技术，否则可能会人际关系紧张。而与此相反，一个人如果技术能做的还不错，也愿意与人沟通，同时已经身处一家技术路径不是很长的公司，并不太能够换工作，那么就很可能需要尽早转向管理方向。

图 4-1：管理方向还是技术方向的选择

§4.2 增值之路的起点

从大环境来看，想不写程序直接去做管理工作是很难的。大多时候都要在开发上做出一定成绩，接下来才有选择技术还是管理的机会。因此即使是希望选择管理方向，技术上的基本功还是需要的。所以下面这节的内容和选择做管理方向还是技术方向关联不大，只要是想做程序员，大致上都有必要一读。

4.2.1 从那里开始编程生涯

大学里经常会开设软件工程专业，在这门课程里面大多时候会讲解需求工程、开发模型、设计方法、项目管理等。但很多同学会感觉这课程让人摸不着头脑，认真学也学不到什么。从这种普遍现象可以发掘出一些本质问题。

这里的本质问题是指软件开发是实践性非常强的科目，因此不适合先从概念开始。这不意味着软件工程、设计思想不重要，而只是说不适合从纯粹的概念开始学习编程。

在软件这个行业里，很多比较资深的人员对如何学习编程是有统一认识的：学一点，实践一点，再学一点，再实践一点。但困难的是每个人对每次迭代的“一点”究竟是多少认知不同，对“一点”是什么的定义也是不同。

我个人的观点是以 3000 代码行为界算第一个一点，也就是说一个人学会某个语言后小练习不算，先完成一个 3000 代码行左右的，没有 UI 的独立程序。为解释什么叫独立的程序，举一个具体的例子。

在《敏捷软件开发：原则、模式和实践》一书中有一个薪水支付的例子程序，正好是这个规模，很适合帮助达成这一目的。这个程序的基本规格说明是：

（下文引自《敏捷软件开发：原则，模式与实践》）

- 有些雇员是钟点工。会按照他们雇员记录中每小时报酬字段的值对他们进行支付。他们每天会提交工作时间卡，其中记录了日期及工作小时数。如果他们每天工作超过 8 小时，那么超过的部分会按照正常报酬的 1.5.倍进行支付。每周五对他们进行支付。
- 有些雇员完全以月薪进行支付。每个月的最后一个工作日对他们进行支付。在他们的雇员记录中有一个月薪字段。
- 同时对于一些带薪(Salaried）雇员，会根据销售情况，支付给他们一定数量的酬金。他们会提交销售凭条其中记录了销售的日期和数量。在他们的雇员记录中有一个酬金字段。每隔一周的周五对他们进行支付。
- 雇员可以选择支付方式。可以选择把支付支票邮寄到他们指定的邮政地址；也可以把支票保存在出纳人员那里随时支取；或者要求将薪水直接存入他们指定的银行账户。
- 一些雇员会加入协会。在他们的雇员记录中有一个每周应付款项字段。这些应付款必须要从他们的薪水中扣除。协会有时也会针对单个协会成员征收服务费用。协会每周会提交这些服务费用。服务费用必须要从相应雇员的下个月的薪水总额中扣除。
- 薪水支付程序每个工作日运行一次，并在当天为相应的雇员进行支付。系统会被告知雇员的支付日期。这样它会计算从雇员上次支付日期到规定的本次支付日期间应付的数额。

在学完编程语言、面向对象、UML 之后可以先参照这份规格说明，什么例子程序都不看，自己完整的做一份实现，实现中要包含 UML 图和代码，接下来可以去把 Robert C.Martin

的例子程序下载回来，同自己的实现在设计上和实现细节上做详细的比较，找出那点自己好，那点 Robert C.Martin 的好。这样对编程语言、对面向对象、对设计原则就可以有比较踏实的一些理解。此外，这个程序的一个额外的好处是它可以完全独立于平台，只依赖于语言和标准库即可。

假设说一个新手已经熟练掌握了一门语言，那么完成上述的任务估计需要 2~4 人周，当然有经验的人 1 个人周左右已经足够了。

在此之后，可以精读一个上点规模的（1~5 万行最佳）独立性比较强的应用程序，由于已经上了规模，做到完全的与平台相隔离就有点难了，没法提供统一的例子作参考。但选择标准主要有两个，一个是尽可能和自己未来期望的方向相吻合，一个是尽可能比较独立和经典。独立的目的是方便调试，经典的目的是确保代码质量。比如：如果是 Windows 本地应用就可以考虑 Notepad++类的开源应用，Web 应用就可以考虑 PetShop 等。这个时间点上需要避免好高骛远，Linux 内核与 Chrome 当然很好，但它们并不适合初学者的。

在精读过程中可能需要几类书籍：一是平台框架相关的（线程机制等），一类是模式相关的，一类是工具型的书（如何调试）。这个时候是要耐下心来读几本比较经典的书的。

精读之后，就要再找到一个项目来实践。这里的关键是真的项目，至少要有真的用户，并且用户数目越多越好。最好是能够向知名开源项目提交代码。

各种基础知识中比较例外的是计算机体系结构、数据结构和算法这类理论性比较强的东西，这种学习曲线比较陡的东西需要结合大学的课程把它学会，接下来再在实践中逐渐应用，而不能一边做事一边学习。原因是学习曲线越陡的东西越需要大块时间，毕业之后再学效率会差。

总的来看，上述几个步骤，应该在大学毕业后 2~3 年内完成，最好在大学里完成，这样可能会有点优势。这些完成之后，打基础阶段可以算是基本结束。

如何参与开源项目？

蔡俊杰主编的《开源软件之道》对开源软件的各个方面进行了比较系统的介绍，也介绍了参加开源项目的方法。总的来看参与开源项目并没有很高的门槛，需要的是用心和投入。

为了参与开源项目第一步选定项目并弄清目标开源社区的运作方法。Apache 和 Mozilla 都是开源社区，但运作方式还是有差异的。

接下来要熟悉特定的开源项目，包括产品的功能和特点、代码规范和架构等。为了做出具体的贡献，那么需要订阅邮件列表或者新闻组来了解社区当前状况。接下来可以尝试提交补丁或实现新功能来解决社区中的特定问题。

一般来讲开源社区会把参与人员分为几个层次，比如：在 Apache 的运作模式中共有这样几种角色：开发者（Developer），技术专家(Committer)，项目管理委员会成员（Project Management Committees)，Apache 软件基金会理事会(Board of Directors)。当一个开发者付出足够努力，获得足够影响力后，项目管理委员会会授予这个人 Committer 称号，只有 Committer 才拥有对代码库的写权限。如果 Committer 的贡献足够大，就有可能被选举进入项目管理委员会，而项目管理委员会的主席则都是 Apache 软件基金会的 VP。新参与者在最开始阶段必然只能是 Developer 的层次，不会具有写权限。只有通过积极参与讨论，为社区做出了具体贡献之后才可能获得 Committer 称号。这是一个持续投入的过程。

当然参与社区工作，不一定非是提交代码，也可以帮助完善文档、运行测试用例、报告缺陷维护网站等等。

4.2.2 打牢根基 VS 速成道路

有很多不同的方法可以学会编程，比如说：一个人既可以先打牢基础，接下来再逐步学会如何进行各种开发工作；也可以不管三七二十一，先借助各种 IDE 把程序做出来再说。上一节主要介绍的是先打根基的方法。

相对于打根基的方法，后一种学习方法更容易在短时间内看到效果，所以很多人都是这么入的行，比如：先从 IDE 开始，接下来再从表面往本质去学，逐步去了解控件拖放背后所隐含的东西。这不能说完全不好，但我认为这并非是一种上佳的成长方法。

在刚开始编程的时候，如果形成对 IDE 的过度依赖，那就会导致根基浅薄，能做的事很可能被限制在某个有限的范围内。当下的大多 IDE 功能已经非常强劲，这对提高产品的生产率无疑是非常有必要的，但在学习阶段，则要尽可能避免过度依赖于 IDE，避免用各种控件来快速完成任务。

比如说：微软在 Visual Studio 2012 里面内置了一种名为 LightSwitch 的技术，基于这项技术，一个人可以在基本不编码的情况下，完成信息系统的开发，并支持相对比较丰富的功能（增删改查，搜索，排序等）。无疑的基于这样的工具做开发速度会快上很多，但在学习阶段过度使用这类工具，却会毁了一个人的根基。

想象一下，在使用 LightSwitch 的过程中一个人会学到什么？他所能学到的主要是这种工具的使用方法，既不会学到 SQL 语句，也不会学到数据库表格的设计方法，也不会学到 ASP.net 的基本架构（虽然 ASP.net 已经封装了非常多的东西）。这样一来，这个人虽然能够快速的完成某个工作，但却给自己埋下了很深的隐患---他很容易的被束缚于某个工具，并且无法应对新领域。

善用 IDE 集成好的功能进行快速开发不是学习阶段应该做的事情，学习阶段最根本的目的是打基础，把一门语言学精，把一种设计思想学精，把一种算法学精等等，这种基础可能不直接表现为生产力，即使把算法学精了，可能还是无法立刻写出来比较炫的程序，但这有助于面对不停变化的世界，这与单纯的达成某个目的，完成某个程序不同。这可以类比为打地基与盖楼，地基部分显然不能单独进行销售，也不能住人，但没有地基也就没有其上几十层可以高价售出的住宅，楼越高，地基也就越深。当然，只盖地基或者让五层楼和五十层楼使用一样的地基也没必要，这不用多说。

那具体来讲，那些东西可以被认为是编程的根基，需要在学习阶段扎实的掌握？下面将通过推荐几本书（或者说几类书）来描述一个共通于所有程序员的最小集合。

- 计算机体系结构

这一类别下最具代表性的书籍是《深入理解计算机系统》，作者是 Randal E.Bryant 和 David O'Hallaron。读这本书的目的是了解计算机到底是怎么个东西，软件到底运行在什么样的基础之上。

- 算法和数据结构

这一类别下最具代表性的书籍是《算法导论》，作者是 Thomas H.Cormen，Charles E.Leiserson，Ronald L.Rivest，Clifford Stein。读这本书的目的是了解软件到底可以用什么样的手段干些什么事情。软件是一种工具，可以帮助解决人类面临的许多问题，而主要手段则只有两种，一种是这本书里所介绍的算法和数据结构；另一种则是下一类别中所涉及到的分析和设计方法。它们像小刀子一样，可以把各种领域中的各种问题进行分割，并映射到程序的世界里来。

- 设计原则和模式

这一类别中比较有代表性的书籍是《敏捷软件开发：原则、模式与实践》，作者是 Robert

C.Martin。读这本书的目的是了解数据结构和算法之外另一种对现实问题进行抽象的方法如面向对象以及进行这种抽象时所要遵守的原则。

这类书籍经典的还有很多比如：GoF 的《设计模式》，而之所以选择上述这本是因为这本书里提供一些比较完整的例子，更适合初学者一些。

- 软件工程

这一类别中最具代表性的书籍是《代码大全》，作者是 Steve McConnell。读这本书的目的是建立对软件开发的全局视图。知道一个软件从无到有所要经历的一系列过程。

软工的书还有很多，比如很有名的《人月神话》，但《人月神话》类书其实对很多人是不适合的，对初学者就更不适合。

上面四本书是一个最小集合，针对不同场景需要进行不同的增加，比如可能需要进一步了解某种框架的机制，那就需要读《Hadoop 技术内幕》这类书。但即使是读这几本书也不适合只读书而不动手，最好是穿插在上一节中提到的实践中来读，否则的话对后两本的理解会有所欠缺。

如果想走厚积薄发这条路的上面几本书是一定要预先读通的。阅读过程中，如果发现有些细节问题彻底无法理解，那就要在实践过程中进一步琢磨，找到自己的答案。读者类书时，有一件事情一定要有心理准备：虽然这些书读通并不容易，很花时间，但想读了这些书后立刻写出来一个能卖点钱的工具是不太可能的，这真和大楼的地基一样，没什么立竿见影的效果。如果想尽快达成后者这样的目的，那很可能就要走速成的道路，去读些介绍 IDE 怎么用，某个框架怎么用的书，比如：《C#高级编程》这类。

为避免误解，有一点需要额外进行一点说明。前面强调的避免过度 IDE 依赖主要是指不要用封装良好的模块来取代对基础知识的学习，不是说不需要建立自己的工具箱。查找工具、调试工具（进程线程查看等）、二进制数据查看工具、正则表达式工具、持续集成工具、文档生成工具（JavaDoc)、正则表达式工具等对一个程序员的生产力是非常有帮助的，应该在学习过程中逐个掌握。软件开发工作发展到现在，任何一个类别下面都有相当多的比较成熟好用工具了，关键选定一个把它用熟。

如果条件实在不允许厚积薄发，非要速成，我觉得可以尝试用这样一种方法：死磕一个流行的开源程序。比如国内用的比较多的是 WordPress，那如果狠狠心花半年一年把 WordPress 搞透了，技能能提升不说，对找份合适的工作也还是很有帮助的。

4.2.3 掌握读代码的方法和技巧

不管最终想成为什么，刚入行之后，一定离不开的是读代码和写代码。这里将介绍一些读代码的方法和技巧。

读代码这事，先要分是精读还是泛读。从学习的目的来看，一定要精读一定量的经典代码。而精读是指每行都读懂，不看代码脑子里就能勾画出程序的基本结构。

要想判断是不是精读了有个很形象的判断方法：精读代码时会满脑子都是代码，放不下，甚至睡觉前脑子里也是代码。达到这个程度就是精读了，否则应该就还不是。精读代码要控制规模（初始阶段一万行以下即可）并用心，不太需要什么特别的方法。

这节里主要关注的是如何泛读较大规模代码，不是精读。

现存的很多系统往往很大，几十万行的可能也只算普通。这时候一旦加入了这样一个

项目，应该如何去读代码？

读规模较大的程序前，先得把规格说明书大致弄清楚，而不能上来就读。比如：对于应用程序，要先大致整清楚它的使用方法、使用场景；对于库则要弄清楚它对外接口的定义。

如果其中有涉及到某些专门的领域知识，比如：流程、财会等，那也最好预先有些认识。这类东西彻底的从代码里反推回来是不太可能的。如果弄不清这类东西，很多时候对读程序是个很大的障碍。你不知道代码做的是什么，却去读对应的程序，那就只能看到调用来调用去，最终会云里雾里。

接下来从大往小，从面到点来看。

一旦开始接触代码，那要先弄清楚代码的基本静态结构。如：包构成、类构成等。这里几乎一定会涉及一个层次问题。一下子把层次探的太深，就容易盯在细节上出不来。把层次拔得太高，又容易流于表面。从数目上看，一个层次最好不要超过 10 个关键概念，超过了真记不住。在静态结构这步，要弄清楚每个部分的核心职责，可以是很简单的概括，最好能记住。

接下来选择出最常用的典型场景，然后在典型场景下考察上面的静态结构是如何发挥作用的。典型场景下用到的接口往往就是关键的接口，要弄清楚他们的定义和作用。也要整清楚典型场景下数据流的变迁。

通过这两个步骤等价于脑子里可以生成一份比较高层次的静态和动态结构图，很像 UML 里的 Sequence 图和类图。牵涉到数据库的时候，一般需要对相应的数据规格有所了解。

接下来要关注进程、线程的结构。比如：都是什么时候开始、什么时候结束的，在上述典型场景下都负责干什么。

上述四步（规格、静态结构、典型场景、进程线程）完成后，对程序的第一次泛读完成。检验是否达成目标的方法可以很简单，如果真的基本读懂了，这时应该能够单靠纸笔描述出程序典型场景的 Sequence 图。

做第一次泛读的时候，要抑制自己的求知欲，因为总是很想在调试器里通过 call stack 把一个功能的实现细节整清楚。至少在第一个次泛读里，可以先不要这样。

第一次泛读后，就要进入深掘的过程，一般来讲需要针对自己会负责的部分进行深入挖掘。这部分功能往往会隐藏在某个接口之下。

这时候一般来讲可以把功能型的模块优先级降低，比如：XML 解析的模块等。其他部分可以认为是需要把之前所说的四个步骤再重复一下。但这时候要关注细节和调用堆栈了。

不管是在那个读代码的层次，有两个基本技巧总是需要的，一个是要掌握具体程序里内嵌的 Log 机制，要能看 Log，必要时可能还得加 Log；一个是基本调试方法。同时一个合适的代码阅读工具会对提升代码阅读速度有所帮助，比如：一款名叫 SourceInsight 的小工具中可以把窗口分拆为几个部分，点击任何方法的时候，这个方法的实现以及 Calls Graph 都可以被自动展开，这样的小功能无疑的对阅读代码是有帮助的。

4.2.4 从那门编程语言开始学习？

学习编程至少要掌握一门编程语言，但从那门编程语言开始是一个极其容易引起争议的问题。为使结论经得起推敲，这里需要做一点系统的分析。

纯从未来应用的角度看，结果是不确定的，在学习的时候，其实没人能够知道未来会主要使用那门语言。因为最终工作中使用那门编程语言往往取决于一些很偶然的因素，比如现有产品的开发语言，待解决问题的领域等。比如说如果命运安排你去做和 Hadoop 相关的工作，那很可能会用到 Java，如果安排你去做驱动开发，那就很可能会用到 C/C++。

如果上述这点成立，并且被预设为前提，那么在学习阶段应该学什么就可以有个相对确定的答案：学习阶段学习语言的目的是为了掌握编程的基础概念并能更快速的学好另一门语言。显然这仍然是打基础的范畴。

从这个角度看，只有一门语言是必须学的，那就是 C。因为不了解这门语言会造成一定视野上的限制，使基础薄弱，比如不掌握 C 语言的人，很可能无法了解《深入理解计算机系统》这样的书，进一步也就不理解什么是指针，什么 Stack，什么是 Stack Overflow，什么是写超界，做性能优化的时候可能也就想不到一些系统级的手段。Joel 在《软件随想录》里专门有一章叫“学校只教 Java 的危险性”，其中所表达的观点与这里的观点类似。

作为结果，尽管很可能在工作中用不上 C 语言，在学习的时候还是要把它掌握，除非在最初阶段就已经下定决心只把技术当做敲门砖，而不想走的更远。要不然根基就过于薄弱了。

至于其他一些比较主流的语言比如 C++，Java，C#等可以完全按照兴趣来进行选择，唯一关键的是不管选择那个都要累积一定代码量并把它学透。这样依此扩展到将来要用的编程语言，学习曲线往往就会很平，大致 2~3 周就可以用新的语言做一些基本的开发工作。

选择编程语言的另一种思路是从脚本语言入手，比如 PHP，Python，Javascript 等。这就和赵匡胤当年要下决策是先搞定弱的南唐还是先搞定强的辽国一样，是个两难的话题。从入手容易，培养兴趣的角度看，显然脚本更好些，并且脚本语言也是互联网的显学，未来用到的机会很高；但如果想多积累，厚积薄发那么就还是从 C 入手会好些。我个人的建议是如果在大学里那就先难后易好些，因为人生里不总是有这么大块的时间；但如果是后想转入这个行业，那就直接找脚本开始吧。

4.2.5 小结

写程序、读程序、学好学习曲线陡的知识、避免 IDE 依赖这些事情的根本目的都是为了打好基础。这个环节里最忌讳的是急功近利，比如：学习一堆 IDE 的操作方法、每个编程语言都掌握一点。很多人可能误以为这对找工作有帮助，所以把但凡接触过的技术都列到简历里是很常见的做法。但其实这个认识是不对的，但凡有点规模的公司招聘毕业生或者刚毕业不久的开发人员的时候都更看重他的基础和发展潜力。而基础和潜力这两样东西很难精确度量，但并不难判断，通过简单的面试既可以判断出来。只关注当下这个人能干什么的公司很可能是看不到明天的公司。

§4.3 如何顺利的成为高手

一旦度过了初始阶段，那么一个人算是基本入行了，接下来的目标就非常简单，要在选定方向上成为高手。高手意味着专业，而在分工无限细化的年代里，专业则是生存、发展好最为重要的一个前提。

4.3.1 高手的定义和养成关键

我估计如果问 100 个人“什么样的程序员是高手？”，那答案会有 100 多个。因为同一个人还可能给高手下不同的定义。

在这里我们认为，在特定领域里能搞定大部分人搞不定事情的就是高手。从这样一个定义出发，我们会发现在技术人员和销售人员眼里，高手的内涵是有很大差异的。

纯技术人员更多的关注性能能不能提到极致，并发能不能处理的很好，内存溢出 Bug 能不能很快搞定，类库的机理熟悉不熟悉等等。而在销售人员的眼里，则在技术外还多看了些东西，比如业务流程熟不熟悉、使用性好不好、能否迅速对应变化、能否在限定工期和预算下搞定任务等。

考虑到职场和产品销售有着非常紧密的关系，我们这里使用后一个视角，而非是单纯的技术视角。

有几类本质上很不同的人都会被视为高手，比如说：

- 能写出很牛的病毒的

  这个不举例子，但当年读过 CIH 的代码，我是被其精巧给震住了。此外也许搞加密解密的也应该放在这个类别里。
- 能把一堆 3D 图形放到 64K 的

  以前专门有个比赛是干这个的，64K 大小的 EXE 能给你放 10 几分钟很酷的 3D 动画，第一次见绝对会很震惊。
- 能迅速调试出问题所在的

  内存泄露、多线程同步这类问题往往让人纠缠很久也搞不定，但就是有人能很快的解决这类问题。
- 能仅靠几个人就架起高并发网站的

  新兴 Web2.0 网站如：Flickr，甚至还可以包括 Google，在初期往往是几个人搞起来的，这些人名声不显，但绝对是高手。
- 能主导开发出很牛的产品的

  这个上可以想想 Unix 和 Linux 的作者等。
- 能主持大规模软件设计的

  这个往往更有商业价值，我们常说的 Martin Fowler 应该可以算在这个类别。
- 能把一种语言研究的特别牛的

  想想各个编程语言的创建者，想想 C++的大牛们。当然创建某一门语言的也可以归到这个类别里。
- 能开辟自己方法论的

比如搞 CMMI 的 Watts S. Humphrey

- 能写出很牛的书的

比如：Windows 平台下写了 Windows 核心编程的 Jeffry Richard

- 能写出很牛的算法的

比如：Donald Knuth

这个表应该还可以加长很多，单以大家认可这个角度来看确实高手可以从各个方面冒出来。

不管在那一方面，要想成为上面所描述的高手总是需要学习、思考、实践这些环节，这没什么可说的。但和软件相关的知识其实多如牛毛，完全不像小说里武功秘籍那么稀缺，几乎可以讲满地都是。这就使选择和集中成为难题。

软件的三个基本特征（技术更迭快、低介入门槛、多内部分野）就像铡刀一样，一旦选择出错，就会把个人的努力切的粉碎，一点价值也留不下来。而与此相对的，则是人的黄金学习时间其实并不多---不过是毕业后的 10 年左右的时间。

曾经有人希望自己能够从事嵌入式软件的开发，因此给自己买了 ARM 板，自己在家里花了很多时间来学习并实践相关知识，最终却因为其他的原因进入了一家做网络的公司。这个人等价于被软件的内部分野较多，而彼此间技能流动性较差这样的一个特质斩了一刀，被斩掉的倒不是 ARM 板，而是自己一年多的辛苦投入。这种情况下强调学的知识将来有用是没有太大意义的，因为还有两刀在等着：如果你三年都不做这个，你今天学到的知识可能会被更迭掉了，同时由于你年纪增长了，可能也不太适合与大批新介入这个行业的人员进行竞争。

这类事情使软件行业中的成为高手这事变得复杂了。

为了在成为高手这条路上走的顺畅，事实上有三个关键点：一是要有一张全局性的地图，以便选好方向；二是要知道都有那些坑，好绕开它，免得掉进去。三是要有足够的热情和动力，能坚持走下去。下面将分别从这三个方面来说明成为高手途径和方法，而这种途径和方法会因为具体目标不同而有所微调。

4.3.2 全局性的地图

清代著名学者曾对知识地图的必要性做过非常精确的表述：

凡读书最切要者，目录之学也。目录明，方可读书，不明，终是乱读。
---王鸣盛，《十七史商榷》

目录即是地图。

对于软件开发的知识，我更愿意使用下面的的“地图”，这不一定是最合理的，但确实对归纳各种软件开发知识有所帮助。

- 通用的领域知识
  - ✧ 编程语言（C/C++，Java，C#，Python，Perl，PHP 等）
  - ✧ 框架和类库(Struts，Spring，OSGi 的某个具体实现，MFC，Boost 等)

- 平台(Windows API，POSIX，.Net Framework※1，Java API，C/C++ Runtime Library 等）。恰如 Jeffry Richter 所说，大多时候可以从内存机制、线程机制、错误处理、异常处理、组件构建、组件组合等方面来进一步考察一个平台。
- 计算机体系结构(CPU 指令，虚拟存储等)
- 数据库
- 实用技巧(调试方法，代码生成器等)
- ... ...

※1 有的时候子类别间的界限并不是很容易界定，其中一个主要原因就是存在着像.Net Framework 这样涵盖了过多内容的概念。

- 概念和逻辑创建和优化
  - 面向对象分析和设计/结构化分析和设计
  - 设计模式
  - 重构
  - 契约式编程
  - UML ※2
  - ... ...

※2 从形式上来看 UML 更近似于一种编程语言，但从其目的上来看也许归在这里是更合适的一种选择。

- 专业领域知识
  - 图形图像算法
  - 网络协议
  - 人工智能
  - 数值/非数值类算法
  - 财务知识
  - 负载均衡
  - ... ...

关于软件的间接知识：

- 需求开发和描述
  - ... ...
- 估算
  - 估算法。比如，COCOMO, FP 等。

- ✧ 估算术。比如，使用计数等原始办法。

- 测试
  - ✧ 略
- 软件工程和方法论
  - ✧ 轻量型方法论。比如敏捷。
  - ✧ 大方法论。比如 CMMI
  - ✧ 综合分析。比如，《人月神话》，《人件》所做的工作。

随着待解决问题越来越复杂，通用的领域知识中，几种技术往往会组成一种技术 Stack，他们更需要被看做一组必须一起掌握的知识，比如：LAMP（Linux+Apache+MySQL+Python/PHP）。

当然上面罗列的远不是全部，这种罗列更多的是展示一种分类的方法。通过对这种分类方法的补充和完善，大多可接触到知识都可以被归入特定的类别，比如说：WinRT 可以看做一种新的平台，HTML5 则可以看做是一种语言等。

每个人可以根据自己的情形，参照上面的分类建立属于自己的地图，有点问题没关系，有就比没有要好很多。接下来依据这样的地图就可以选一条自己的线路，持续累积，寻求实践机会，最终就很可能会成为真正的高手。

而关于增值所需的动力，所要避开的陷阱，将下面的章节里陆续提到。

增值、读书与大局观

单纯从达成某一目的而言，读书往往非是绝对必要条件。

秦始皇把书一把火烧了，刘邦项羽一样造反并取得胜利。但读书无疑的可以加速一个人增值的过程，记不得是谁说过：实践无疑是人类最好的老师，但只靠实践来认知世界无疑也是愚蠢的。这是非常精辟的。除此之外，要想培养大局观，那就非读书不可。

每个人的亲身经历，在大的时空背景中往往只是一个简单的截面，这一截面中绝不会包含可以归纳出所有真理的事实，因此只依赖于自身的实践也就必然限定了一个人的视野。

这一点随着一个人的责任范围变大往往会体现为一种制约和限制。所以培根讲：有实际经验的人虽能够处理个别性的事务，但若要综观整体，运筹全局，却唯有学识方能办到。

即使从实践来看也是如此，要想培养出一种大局观，那就非读书不可。而大局观往往是成为将帅之才的必要条件。

具体到软件而言，有一本很有名的书对培养技术的大局观有帮助：《代码大全》。至于专业性较强的书，反倒是可以根据自己的情景比较容易的选择，这里就不提了。

基于上面这样的一张地图，我们就可以具体的去考虑几条进阶路径。

1. 路径一：由程序员而架构师

架构师是一个很火的职位名字，但你很难给它下精确的定义。

下面所陈述的一切是我个人的理解和体会，我无法保证它和其他人的解释完全吻合。因为架构师，乃至架构设计实在是一种非常模糊的概念，如果你用心去找，可以找到各种定义（甚至 IEEE 和 SEI 的定义也不一致），这就导致你只能参照别人，相信自己。

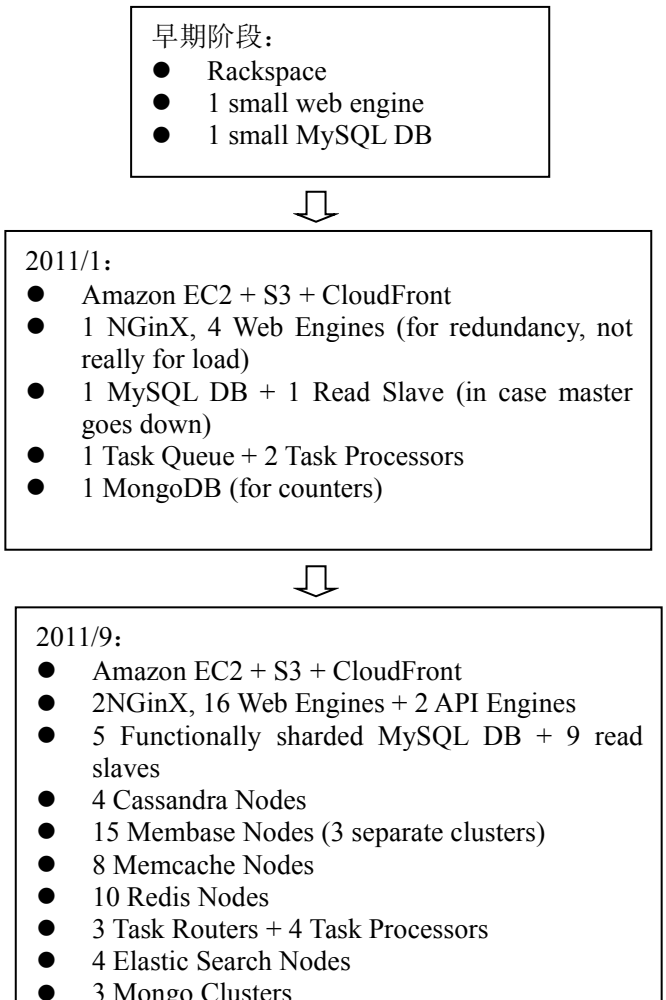
本质来讲架构设计也是设计，所以凡是做设计的都可以称自己为架构师。

当一个系统的规模变大的时候，设计上的决策就具有了特别的价值，并且也越来越需要专门的人来做，架构设计也就越来越像是一种特别的设计。比如说：考虑架构设计的时候，可能需要考虑选用什么样的数据库、选用那个开源框架、选用什么样的硬件平台，这些东西在小规模程序中往往是居于次要地位的。

假设说一个人已经掌握了一门或几门编程语言、面向对象、设计模式、能够很熟练的写出质量较高的代码，接下来他想成为架构师，这个时候他需要做什么？

我个人认为，这时候这个人首先要有一个“专业”。这个专业可以是“金融”，“财务”，“电商”，“管理”等等。这是一种属于某一专业的领域知识，而不是编程技术。如果把需求和最终的代码，看成描述同一事物的一体两面，那么设计始终是要架起这两者间的桥梁。而架桥的时候，怎么可能只知道一端而不知道另一端。

接下来是深化设计所需要的各种通用领域知识（UML、面向对象、性能确保等）。这时和一般所说的设计的一个关键区别是，那就是架构设计要分心思去考虑那些东西用别人的就好了，而那些东西要自己开发。而一般所说的设计技术中，比较侧重自己应该怎么干（面向对象、测试驱动等）。为达成这一目的，就需要对现有技术的优劣有相对比较清晰的认识，比如要能分清楚那些是成熟稳定的技术，那些是处在实验阶段的技术。Pinterest 网站就曾经进行过下列这样的架构改进，在这样的改进过程中，不知道各种技术的优劣是代价很大的：



2012/1：

- Amazon EC2 + S3 + Akamai, ELB
- 90 Web Engines + 50 API Engines
- 66 MySQL DBs (m1.xlarge) + 1 slave each
- 59 Redis Instances
- 51 Memcache Instances
- 1 Redis Task Manager + 25 Task Processors
- Sharded Solr

2012/12：

- Amazon EC2 + S3 + Edge Cast,Akamai, Level 3
- 180 Web Engines + 240 API Engines
- 88 MySQL DBs (cc2.8xlarge) + 1 slave each
- 110 Redis Instances
- 200 Memcache Instances
- 4 Redis Task Manager + 80 Task Processors
- Sharded Solr

摘自：

http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.html

最后一点是，做架构设计已经相对于在做技术管理工作，至少要适当涉猎估算并能做出合适的任务分解，这样一旦日程紧张，则可以通过增加人手等手段来在质量、成本和进度之间进行均衡。

由于知识面已经扩的比较大，架构师在具体某个专业领域上的深度可能会有所欠缺，比如：在做一款电子消费产品时用到了 TTS，但架构师不一定能很好的了解 TTS 的算法---这是 CodeGuru 的领域。

架构师所需要达成的最终目标可以形象的描述为：产品经理考虑用户和市场建立了一个模型，那么架构师要能把这东西映射到技术的世界里来。如果是在互联网行业，那么在你的主导设计下要可以做出高并发的网站。换到其他行业也与此类似，从产品的的角度往回看，架构师要能解决和技术相关的所有问题，主导完成商业上有价值的产品或项目的开发工作。实现手段上倒并无限制，可以是购买，可以是组织人员进行开发，只要能平衡短期和长期利益，解决特定的问题即可。

2. 路径二：由程序员而 CodeGuru

与架构师相对应，在某些智力密集型的程序中，也需要技能高超的程序员，这种程序员往往被称为 Guru。

这条路线里，程序员并不把自己擅长的领域扩的太宽，但在指定领域上会挖掘的很深。驱动、字库、图形库、算法库、OCR 等都偏向于这一领域。

假如说，一个程序员在掌握基本语言之后，想往这个方向发展，那么需要的技能和架构师差别很大。比如：一个人如果想往驱动方向发展，那么就需要了解 CPU 的基本结构、内核调试方式、操作系统中与相应驱动所对应的机制、硬件侧的规格、通讯协议等。

这时候很可能由于程序规模并不十分庞大，面向对象、设计模式这类东西没有太大发挥空间，而是需要处理的是大量或麻烦或艰深的细节。

2013 年 1 月，ITeye 发布了一条信息说：Intel 面向学生免费提供 C++ 开发工具，并简单的介绍了一下是那些工具免费向学生提供：

- Intel C++ Composer XE，其中包括：
  - Intel C++编译器（高度优化的编译器）
  - Intel 数学核心函数库（高性能数学库）
  - Intel 线程构建模块（C++任务模型、最流行的 C++并行方法）
  - Intel 集成性能基元（多媒体基元库）
- Intel Advisor XE（推荐的并行开发建模方法）
- Intel VTune Amplifier XE（非侵入性的性能分析工具）
- Intel Inspector XE（先进的线程和内存调试工具）

看到这张列表，我们可以思考下要开发这类工具需要什么样的程序员。上面这些工具的开发都属于高难度的工作，和开发大规模的 MIS 系统完全不同，如果不是某方面的专家，基本不太可能负担起相应的责任。而这类领域则正是 Guru 的天下。

成为 Guru 的最简单方法？

有一条成为 Guru 的捷径，并且可能可以比较好的实现自己的价值：选定一个应用比较广的，有深度的开源项目，读通它的代码，参与进去，在社区中累积自己的声誉。这样的项目越来越多，比如 Hadoop，比如 Webkit。选择这种项目时，有两点非常关键，一个是项目自身要有一定难度，否则无法借助它成为 Guru；一是这个项目要有非常广的应用范围，否则可能 Guru 的价值无法变现。这是国内来自现状的一种约束，大多数人还无法单纯的按照喜好的来选择项目。

这么做之所以有价值在于，开源项目因为即可以获得特定软件的全部功能，又可以进行灵活定制，正逐渐成为知名互联网公司的一种必然选择。看看国内的浏览器市场可以对这点有更深刻的认识。为了争夺入口，腾讯、搜狐、360 等纷纷推出了自己的浏览器，但为了推出这种浏览器是自己从头开发一个内核，还是使用现有的相对比较成熟的内核 Webkit 等似乎是个不需要考虑的问题。这就是开源的价值。在这样一种前提下，成为特定领域中开源项目项目的 Guru 就不单有技术修炼上的价值，也有很大的现实意义。

这个方法虽然简单，但真要操作有两点要事先了解：一是虽然操作简单，但不意味着不需要付出很多努力，需要投入的心力一般来讲会远比事先设想的多；一是一旦走这类道路，等于给自己选择了个专业，很可能只能面对特定的几个公司。比如上面提到的 Webkit，掌握这类知识只有加入想做浏览器的公司才会实现自己的价值，而真正能做的起浏览器的公司其实并不多。

3. 路径三：由程序员而纯管理

纯管理工作和技术管理工作可以用是否接触乃至编写代码来区分。纯管理工作往往需要把精力放在预算编制、人员职业路径、考评、度量、流程改善这些工作上。一定程度上讲，这等价于和编程工作说拜拜，当然前提是你得有编程经验，有一些通用领域知识和概念创建乃至逻辑优化的知识，否则的话和程序员没法沟通，进而给工作造成障碍。

从需要读的书来看，这时候可能要看过 PMBOK，《项目管理修炼之道》，《管理的实践》，《基业长青》等等。

但如果一个人认为想做管理要从 PMP 开始，那大概是还没太明白管理这项工作的本质。管理本身是一种借势，虽然有技术性的一面，比如要理解挣值曲线这类，但这方面知识其实并没有想的那么复杂---至少没有 C++11 复杂，只要有时间正常智商的人都可以在不太长的时间内掌握。所以如果你想做管理，并使用了和学习 C++语言一样的方法，那基本上是偏离了方向。

抛开机缘这类东西不论，做好管理工作有两点很关键：

一是要把技术工作做的相对比较好。这好像有点学而优则仕的意味，但大多时候人们更愿意相信“将军起于行伍，宰相拔于州郡”，而不愿意相信单只会耍嘴皮子的人。过度务实的人容易迷失于道路，过度务虚的人则容易飘的太高而丧失根基。管理者正应该身处在这两者之间的一个平衡点。

二是要能够借势。要情商比较好，能把很多人组织在一起。这个时候要知道那些东西需要规则化，那些东西需要灵活把握。过度偏向规则是教条，过度偏向灵活则是人治，平衡点始终要根据具体人员的状况，工作特质这些不可改变的事情来把握。这有点微妙。但即使程序员这个群体相对简单，但并不能推翻先人后事这类规则。这不知道是不是东方特色，当你想做管理并想推进事情的时候，终究要理清人际上的关系，否则就和可能会欲速则不达。这点会在第五章进一步展开来谈。

下面我们来看一个具体点的例子，这个例子出自郭致星老师的博客，是一个学员的真实疑问（文字上有修饰），X 入职后的现状如下：

开发部现在 26 个人。基本组织架构是由开发组、需求组、测试组、运维部四个部门组成。需求组的人收集需求，再通过系统指派给开发组，进行开发。开发组的文档严重缺乏。

现在公司内有技术总监和开发部经理的岗位。眼下全公司就 X 一个项目经理，目前是跟着开发部经理，在熟悉一个核心系统，由于没有任何文档的遗留，所以现在是相当于一个开发人员在开发一些实际的功能，一边开发一边熟悉现有系统。

现在 X 并没有项目的实际权力，而且整个项目组也只有开发部经理，加上 2 个开发，再加上 1 个测试和 X 共 5 个人。其中 X 和其中一个开发都是新来的。

与此同时，业务部门在搞一些流程，但还没做完，没有在开发部进行实施。工作气氛比较沉闷。技术总监和开发部经理，都是技术型的人，比较偏向技术，平时工作也多偏向于具体执行。

在使用一个 Redmine 系统进行项目管理和 BUG 跟踪。

这里的系统大部分都类似于产品线制，属于需要长期开发维护的，需求源源不断的来，相应人员也就需要不停的做，没有版本制度，也没有计划和规划。

系统现在主要的问题是性能问题。

问题就是在这样的一种环境下，X 应该如何开始自己的管理工作？

这类问题通常并没有唯一答案，但确实有些通行的手段可供参考，最终做不做的好和

个人能力乃至环境关联很紧：

1.了解现有系统的状况，包括规格、代码规模、代码质量、代码内部结构、工作流程、问题所在等。比如说：很可能这类系统缺乏一种整体设计，是靠单纯的增加代码的量堆积出来的，代码冗余非常厉害，数据库的表也创建的比较随意。

2.了解人员。包括人员的能力水平、工作意愿状况、性格。

3.了解公司。尤其是公司的运作风格，有的公司偏人治有的公司偏于规则。短期对这类现行秩序要考虑如何顺应，而不是如何改变。

4.对当前系统的状况和人的状况有所把握后，要对愿景进行描画，比如在功能上做那些改善，对速度做如何改善，目标的高低要适度，要能获得上司和下属的支持。这时候还要能平衡短期和长期目标，既不能长时间投入没有产出，也不能有产出但进步不可见。在这一步骤里最典型的忌讳是急功近利的做超出自己影响力范围的事情。比如：目标与现有人员的能力完全不匹配或者完全不顾及对销售可能产生的影响而单纯的做系统的优化。最理想的情形是，连续达成几个目标，提升自己的影响力。

5.搞清楚团队成员和公司的的基本诉求，在取得成绩的同时尽可能双赢的扩大自己的影响力，目标是确保团队的执行力。

6.逐步导入基本流程，使项目上轨道。但流程不能成为成绩的借口。

7.接下来进一步的规划愿景，看能否取得更大的成绩，比如：挑战是否能做出真正有特色比较优异的产品。

在不同类型的公司里，对应手段上会有不同。比如在规范性比较强的大公司，第 4,5 两步的权重就会比较低。在上述这样的场景下，PMP 这类书籍中所提到的种种技术手段诚然是必要的，但和人打交道的部分（老板、直属上司、下属）往往会对最终的结果产生更大的影响，这是管理工作与纯粹技术工作不同的地方。

管理者需要有产品的视角

对于公司而言，产品往往因为和现金流更近而有更直接的价值，而技术之所以有价值则是因为他往往是产品构建中最关键的一环。所以说大多时候是技术服务于产品，而非是产品服务于技术。而过于关注于技术的程序员则往往倒置这一关系，试图让产品服务于技术。但事实上单只技术上可行，却对现金流没帮助的东西大多时候本质上毫无意义（开源例外，开源世界的规则与商业世界的规则似乎是正好掉过来的）。这反过来要求不管是上面三条路线中的那一条，都需要有一些产品的视角。从程度上来讲，对管理者要求最高，架构师稍弱，专家型程序员最低。当然按照具体职位划分，细节上差别会很大。管理人员中的产品经理，其视角需要与产品完全重合，而程序员中负责 OCR 算法的可能完全不需要关注产品的视角。产品视角是个很含糊的词，也很难被清晰的定义，它可能涵盖用户、需求、设计、气质和 UI 等等，理解上未必很有难度，有难度的则是在实践中做种种折中，并做出优秀的产品。

4.3.3 避开增值路上常见的“坑”

上面讲到了程序员成为高手需要有一张地图，借助这样一张地图，程序员可以尝试成为架构师、Guru 或者纯管理者。但这条路总是不会那么顺畅。

升级练功流的网文中总会给修仙的主人公设置下几大关卡，比如：金丹难成、元神难成、成了元神后还有天人五衰等等。没有这些关口，情节很难推动，所以仙侠类的网文几乎无一例外依赖于这类设定。这一传统甚至可以追溯到《西游记》和《封神演义》。

拿这个来对照程序员的增值道路，就会猛地发现，升级练功流也不完全是扯淡。不管走那条道路，程序员的修炼路上同样的也有三灾九难，要想成就高手，还是得一个个跨过去才行。如果一个程序员达到一定高度后再回头观望，那就会发现自己的同学、曾经的同事总是会因为这样那样的原因倒下去。这并不是一个简单的天道酬勤就可以敷衍的行当，下面就

让我们来具体看看，究竟增值过程中可能遇到那些陷阱，掉到坑里又需要付出多大代价才爬的出来。

1. 学习失去焦点

软件行业里有几个经典的题目，每过一段时间就会被翻出来 PK 一下，比如关于编程语言优劣的比较等。其中一个经典题目是软件和数学的关系。

从结果来看，一派人认为数学是软件的基础，而另一派人认为数学和软件没什么太大关系，除非是在某几个特定领域里。如果你用心观察过这事情，你就会发现这事情特别有意思。

比如说：

2011 年，CSDN 转了一篇，叫：“数学是成就卓越开发人员的必备技能”的文章，在文末作者说：

那么，数学对所有事都有利么？这事情很难说，我对我现在的处境十分满意，或许你也如此，但这都和潜能有关系。如果你是协作世界的一名开发人员，你真的不需要数学。如果你乐于你的整个职业生涯是这样的：在工作时间中做企业 CRUD 应用，或在闲暇时间滑翔跳伞或极限水上滑板（或其他各种时髦的极客运动），也分配较多时间在 Spring、Hibernate、Visual Studio 或其它东西上。那些特殊的职位并没有真正限制你的潜力，你能变得极具价值，甚至可深入追求。但是如果你想为多样化的职业生涯而奋斗，想要有能力尝试几乎所有涉及代码的事，从信息检索到 Linux 内核。总之，如果你想成为一个开发人员、程序员和计算机科学家的完美组合，你必须确保你的数学技能达到标准。长话短说，如果你在数学方面有一定天赋，那在软件开发领域中所有的大门都是向你敞开的，如果没有，那你就安安心心地做 CRUD 型工作吧！

总的来看，这个作者认为数学很关键，得学。

而在 2012 年 CSDN 又转了篇文章，叫：编程需要知道多少数学知识？在文章里作者说：

数学和编程有一种容易让人误解的联系。许多人认为在开始学习编程之前必须对数学很在行或者数学分数很高。但一个人为了编程的话，需要学习多少数学呢？

实际上不需要很多。这篇文章中我会深入探讨编程中所需要的数学知识。你可能已经都知道了。

对于基本的编程，你需要知道下面的：

加减乘除—实际上，电脑会帮你作加减乘除运算。你仅需要知道什么时候运用它们。

模运算—模运算是用来计算余数，它的符号通常用%百分号来表示。所以 23 除以 7 等于 3，余数是 2。23 mod 7 = 2。

判断是奇数还是偶数的模运算—如果你想知道一个数是奇数还是偶数，用它 mod 2 来作模运算。如果结果是 0，它就是偶数。如果结果是 1，就是奇数。23 mod 2 等于 1，所以 23 是奇数，24 mod 2 等于 0，24 是偶数。

*对一个数作百分数运算，就是用这个数来乘以一个百分数。譬如你要得到 279 的 54%，就是用 0.54*279。这就意味着为什么 1.0 等于 100%，0.0 等于 0%。*

知道负数是什么。负数乘以负数等于正数。负数乘以正数等于负数。就这么简单。

知道迪卡尔坐标系统。在编程中，（0,0）代表屏幕左上角，Y 坐标的正轴往下。

知道勾股定律，因为它是用来计算笛卡尔坐标中两点之间的距离的。勾股定律

a^2+^2=c^2。(x1,y1)和(x2,y2)两点之间的距离等于((x1 -x2)^2+(y1 -y2)^2)。

知道十进制、二进制、十六进制。十进制就是我们通常用的十个数：0-9。通常认为这个十进制系统是人类发明的，因为我们有十个手指。

总的来看，这个作者认为数学不太关键，大多时候可以不学。

你如果持续关注这事儿，各种极端对立的观点绝对会吵爆你的头。但这事情其实不可能有结论，因为被对立起来的两极都是太大的概念。

比如说：如果把软件开发缩减为应用软件的开发，那争议性就会降低很多。

这里的关键问题是，假如一个人的目标是应用软件开发，却花了 10 年来学习数学，接下来在实际应用程序开发过程中，每天面对的是 UI 布局、IDE 使用、圈复杂度控制、面向对象使用、设计模式的使用和类库的使用这类问题，那么这个人就会发现数学其实没啥用，他等价于因为失去焦点而失去了 10 年。

这里想表明的是，一旦误读了知识与目的间的因果关系，那么学习就会失去焦点，进而造成负效应，毕竟相对于人的可承受负荷而言，这个世界上的知识不是太少，而是太多。

一般的认识是只要学习就必然有所得，所以对人生的影响一定的是正面的。但实际上这个想法是偏颇的，尤其是在软件行业里。

在软件行业中，这种风险之所以异常突出，就在于前面提到过的软件的两个特质：更迭速度快和子领域众多。这两个因素导致软件相关的知识是爆炸性增长。

而避免“失去焦点”这一陷阱的第一关键则是分类：对软件开发进行分类，对软件所关联的知识也进行分类，形成自己的大局观和整体视图。

前文曾经提过集中对知识进行分类，对软件进行分类的方法，这里再补充一个对绕过这一陷阱有帮助的分类方法，它来自《软件成本估算：COCOMOII 模型方法》这本书。

书里面把软件分成了下面几个类别：

<table>
<tr><td colspan="3">终端用户编程（一般的应用程序）</td></tr>
<tr><td>应用程序生成器(开发工具等)</td><td>应用组装</td><td>系统集成</td></tr>
<tr><td colspan="3">基础结构（OS，DB 等）</td></tr>
</table>

在进行分类的同时，书里还给出了一组数据，即 95%的人从事的是终端用户编程。

这个分类的意义在于，通过它我们可以认识到每一类别背后隐含的知识需求其实不一样，程序员则要根据自己的目的设定焦点。再来看一个具体的例子：

算法领域中，最经典的书籍恐怕是 Donald Knuth 的《计算机程序设计艺术》。对此书的评价已经不止是高那么简单了，如：

这部多卷专著是公认的对经典计算机科学最权威的描述。几十年来，无论是学生、研究人员还是编程从业人员，本套书的前三卷都是他们学习编程理论、进行编程实践的宝贵资源。

这是一套集所有基础算法之大成的经典之作，当今软件开发人员所掌握的绝大多数计算机程序设计的知识都来源于此。 ---Byte

那么是不是每个人都应该把这书读一读？这书的前三卷大致有 2000 页，上班的人都读通估计要 1~2 年这个样子。

至少在我来看，答案是否定的。具体情况要看你做的是那类软件，你人生的下一步在那里而定。

假设说一个人做的是终端用户编程（比如：财务类软件），并且目标也是在这个方向

上继续深造，那么你读这书其实是在浪费时间。

这是因为在终端用户编程上，自己去写算法的机会非常的少，甚至可能没有。反倒是业务领域知识（会计知识）、OO、设计模式、甚至于估算这些知识都比研究算法更有价值。

既然如此，那么从务实的观点上看，为什么去学习不能创生价值的东西，而不是去学习立刻可以变现的东西。走极端的人会找出需要写算法的例子来反驳上述立论，但点不足以表征面，即使偶尔需要自己了解下算法，真就值得花那么多时间去学习么，为什么不尽可能借鉴现成的。

而在基础结构性的软件开发中，情形就不同，这时算法无疑是非常核心的东西。所以 Google 这类公司的面试中往往非常强调算法。

这里最关键的就是聚焦，聚焦的根本则是要在限定时间范围内创生价值。

从方法上讲，聚焦就是找到一个适合自己大小的区域，然后做深。才华横溢的不论，一般来讲，横向穿越和纵向穿越都不太行，除非已经基本穷尽当前的领域。纵向穿越是指从底层穿到上层（想想开发网络协议的和用网络协议的），横向穿越则指横跨太多的领域（想想从内核驱动跨越到信息管理软件）。

其实如果一个人真的拥有无限的时间资源，那么什么时间点学习什么就变得不太关键。但这是不可能的，同时一个人的学习时间远不像想的那么充沛。这点会在后面探讨。

在清楚自己的目标后，再配合前面提到的知识分类地图，那么避开这一陷阱的几率就大了很多。

2. 学习与实践相分离

喜欢看网文的很少有人会不知道起点，不过估计很少有人注意过起点给小说分类的标签。在网文小说里有很多个流派，其中一个叫做“扮猪吃虎”。这个流派的基本特征是很厉害的一个人要假装很菜，最后在关键时刻力挽狂澜。这个流派十分有人气，大概仅次于“升级练功”，而远高于“重生”、“转世”这类的流派。

网文是绝对的市场导向，所以这个现象可以从侧面说明很多人喜欢“扮猪吃虎”或者说潜意识里有着“扮猪吃虎”的情节。“扮猪吃虎”这事儿小说里看着很爽，但挪到现实里来很容易让人掉到学习陷阱里。更可怕的是，现实里有这种心思的人其实也还很多。

有时候会看到这样一种现象：很多人自学的东西和工作中用的东西完全没关系。比如：一边用着 C#做 Web 开发，一边自己学习着 C/C++做嵌入式。

这事并不一定不对，只能说非常危险，很可能会导致那样都没有高度。我们得承认当人生被错位的时候，往往只能这样来改变命运，这是没办法，也是正确的。但首先要认识到这样做是相当低效的，低效到一定程度后对的事情也并不一定有结果。

CSDN 曾经做过一份薪酬统计（http://www.programmer.com.cn/5877/），这里面有三个与上述身在曹营心在汉现象有关联的结论：

一是 73%的程序员对自己的薪资并不满意。

二是各个主流开发语言上的差异并没有想的那么大。虽然 C#开发者中月收入小于 5000 元的比例最高，但 5000~10000 这个群体在主流开发语言上相差并不大。

三是平均来看，收入增长和工作年限正相关。当然认为到那个岁数工资自然就高了是很危险的。

表 2　薪资／工龄交叉分析表（百分比）

		您目前每月的工资属于以下哪一个范围?				合计
		不到2,000元	2000元至5000元	5000元至10000元	10000元及以上	
您从事软件开发工作多少年了？	1年或一年以下	19%	66%	13%	2%	100%
	2年	7%	61%	30%	3%	100%
	3年	2%	40%	51%	7%	100%
	4年	0.6%	29%	58%	12.4%	100%
	5年	1%	24%	56%	19%	100%
	6年	0.7%	14.8%	54.5%	30%	100%
	7年	0.5%	19.5%	46.%	34%	100%
	8年	0.5%	14%	46.5%	39%	100%
	9年	0%	13%	50%	37%	100%
	10年及10年以上	2%	13%	38%	47%	100%

数据来源：CSDN网站2011年初程序员薪资大调查

图 4-2：薪资与年龄的关系

对薪资不满意应该是程序员希望跨界的一个原动力，但收入和年限正相关，与语言非正相关却说明单纯从功利角度看跨界并不明智。因为假设一个人 Java 语言用过三年，C#用过三年，总的来看收入水平更可能处在三年的水平上，而不是六年的水平上。

软件是一种固化的思维，这就软件开发更多是一种实践而非是一种理论。软件开发内的很多领域，总体上看体现的是复杂而不是艰难，不论是前端开发还是驱动开发。

讲到到这里就有必要简单区分一下“复杂”和“艰难”。考试出题可以有两种方法：一是每道题都不是很难，但题量很大；一是题量很少，但每道很难。从结果来看，两类考试

方法下，得高分都并不容易，但其难度的来源却并不相同，前者更多的体现为复杂，而后者却体现为艰难。

在软件行业里，除了一些专门的领域，比如图像算法等，软件开发则更类似于前者，所以经过培训后大部分人都可以做软件开发，进入门槛并不高。

解决艰难问题时，天分很重要；解决复杂问题时，练习很重要。所以软件开发的学习过程中，实践很重要，纯理论知识的权重较低，当然基本的算法复杂度还是要明白的。

这也就意味着脱离项目实践的学习投入产出比往往会差。比如说：编程中常见的多线程问题。如果单纯从学习的角度看，创建线程本身并不复杂，掌握各种线程同步方法（事件、信号灯、互斥量等）也并不复杂。写简单例子的时候，也很少会出错。但一旦落到具体的场景下，虽然多线程的本质没变，但没经验的人几乎一定会在涉及多线程的代码上导致一会儿出，一会儿不出的问题。

再比如说：你可能看了很多设计的书，但从来没有从头到尾写过什么程序，总是在既有代码上修修改改，或者只是完成几千行代码的小工具，那么你的设计知识是很难融汇贯通的，也还是无法很好的承担大系统的设计工作。

这点上有一个旁证，根据统计最多的 Bug 是由新手导致的。这从侧面说明，能做和能做好之间的鸿沟需要大量的实践来填平的。

在这样一种前提下，期望先选个工作，再自己学习，努力转行这样的想法是损失很大的。单纯从增值效能上看，解决这点很简单，除非必须放弃当前的工作领域，否则要以当前参加的项目为根基展开学习，这样才能比较好的调和学习和实践。

而除非一个工作领域过于偏狭，大多时候在编程语言（C#→C/C++）、不同领域间（图形处理→地理信息系统等）穿越损失可能更大。

至于如何在“博”与“专”间平衡，如何选中更适合自己的工作领域，将在后续章节里陆续谈到。

3. “博”与“专”上的迷失

假设说一个人的学习已经聚焦，并且学习的内容和自己实际参与的项目也相吻合，那么是不是就没有问题了？很不幸，答案仍然是否定的，在任何一个子领域里，仍然需要进一步去考虑“博”与“专”的均衡。

对于软件开发而言，设计是再常见不过，再简单不过的一个词了。可如果把视角拔高一点就会发现，单以设计而论仍然是一个不可穷尽的领域，我们可以快速扫描一下和设计相关的部分概念：

- 面向对象分析与设计
- 结构化分析与设计
- 模型驱动开发
- 契约式编程
- 面向方面的开发
- 基于组件的开发
- 元编程

有些时候方法论也会和设计牵扯到一起：

- 测试驱动开发
- 敏捷软件开发

如果感觉这个还不够多，那可以去 Wiki 上查编程的范式(paradigms)这个条目，那里列

了 47 种范式，每个都和设计多少有点关系。

上述这些还只是说了设计，如果横向展开，那么在特定领域中必然还会牵涉到框架的选用、辅助工具的使用等等。这也就意味着，从博的角度来看，即使是在设计这样一个看似狭小的领域中仍然是没边界的。

与此同时，把一个 API 研究的再透，也是低值人群，因为这种深入理解和单纯会用某个 API 相比，从创造价值的角度看，差别不大。

这也就意味着对于大多数软件开发人员而言，要去寻找广博与精专间的均衡点：既不能闭上眼睛，也不能就用显微镜来看世界。而这一均衡点的价值则可用反木桶原理来说明：木桶原理说的是桶里的水是由最短的一块板决定的，但考量人的价值时却是适用于反木桶原理，即人的价值往往由最长的一块板决定。

考虑博和专的问题不能离开产品开发进行考虑，前面曾经提到过，产品开发往往和公司的现金流绑定的更紧，能为现金流贡献力量的技术才是有价值的技术。而产品开发本身事实上对博和专的程度提出了最基本的要求，这种要求往往具有迭代的特质。为了形象的说明这一点，这里举一个通用的例子来进行一点说明：

在第一次跌代里，往往需要达到两个最基本的目标。第一个目标是可以为产品贡献自己力量，但代码质量普通。这个目标如果达不到，一个人会失去自己的存在价值。

这时候最少需要了解某种语言（比如：C++)、某个平台（比如：Windows)、某个 IDE（比如:Visual Studio)和某些业务相关的知识（比如：打印体系）。这个范围可以尽可能圈的小点，但用到的则要学透。比如：不管接触到那个框架，都要去了解它的内存机制、线程机制、异常处理组件构建和国际化处理这些全局性的机制，而不能只是了解某个接口怎么用。

这并非是很高的要求，没有这些就变成了“靠运气编程”，写完程序后还要祈祷他能跑起来。了解这些之后就可以负担起部分开发工作，否则的话只能做旁观者，没法参与到实际工作中来。

第二个目标是把事情做好，并能负担些层次更高的工作。这时候要比较深入的了解面向对象、结构化方法、设计模式、理解设计原则，并能把它们用好。至少要能判定，这个程序写的好，那个程序写的不好，同时面对需求能把工作进行下去。

前两个目标是基础，一般来讲学校中基础打的越好，这个阶段越短。达成这两个基本目标之后就可以结合情境来做进一步的选择，可以认为这是博与专选择上的第二次迭代。当然这时候也要谨记不要和实践分开。

完成上述两个层次后，可以有两个方向可供选择。

- 可以进一步考虑专的问题，比如在特定领域里把知识深化下去。做驱动就要理解操作系统的核心机制，做打印的就要了解页面描述语言等，但这个时候要适当警惕边际效应。

  边际效应是说，你让一亩地从亩产 500 斤增加到 1000 斤可能只需要投入 100 块；让亩产从 1000 增加到 1500 可能就需要 200 块；让亩产从 1500 增加到 2000 则需要 400 块了。

  一个典型的例子是对 C++的学习，C++是公认的复杂，如果想做 C++的律师，那么估计搞个 10 年可能够资格了，但问题是把时间都投在这个上，投入产出比可能不好。而停在那里合适则是个尺度问题，大致来讲是可以靠时间弥补的细节问题，并不适合专到最底层。比如对于 100 万行的程序，预先花时间去了解每一处细节，就有点过了。

- 可以把博再推进一步，比如：熟悉专门领域的专业知识、熟悉多种既存框架的特性、熟悉提高用户体验的关键点。熟悉多种既存框架的特性的具体含义是：

  设计某一种解决方案时，首先要考虑的就是是自己开发还是使用现有的模块。一旦决

定使用现有的模块（包，框架等），那就要进一步考虑究竟用那个。

做这类工作时，如果没有一定广博的知识，做选择的时候就会特别的艰难。

假使说现在公司内部要导入一套项目管理系统，那么做决定的负责人必须至少考虑所有下面这些事情：

- ✧ 自己从头造，还是用现成的做二次开发？
- ✧ 用现成的，是用开源产品，微软的还是其他公司的？
- ✧ 用微软的话，是用 MS Project 还是基于 SharePoint，还是混合？考虑 License 费用的话真的划算么？
- ✧ 用开源产品，有这么多选项究竟导入那一个?
- ✧ 如果自己从头造，那么是基于微软的技术，还是基于 LAMP 这样的技术？
- ✧ 使用什么框架？
- ✧ 如果要做，用什么语言？

一个人很难精通上面所有的领域，但当做选择时，完全没有概念也是灾难性的。

此外，考虑博与专平衡点时似乎有一种特例，钻研特定算法的人，从一开始就只往专的方向发展，并不会考虑其他。比如：钻研 TTS 的人，可能几十年如一日只要专注于 TTS 就完了。

至于具体选择那个方向，则要根据自身情形来定。总的原则是要以当下工作为根基，以实用为目的甄选各种知识，并追求平衡点。

大致上讲，期望做技术专家的更适合前一个方向，而期望做技术管理的则更适合后一类方向。

学习软件工程的时机与必要性

简单来讲越是没实践经验的人越不适合学习软件工程，越需要规划整体把握全局的时候越需要学习软件工程。

软件工程中覆盖的元素非常繁杂，可以有管理、流程、开发模型、估算、分析设计方法等。这无疑会把知识面扩展的很宽，一旦没有根底，就很容易变成纸上谈兵，夸夸其谈。

在众多软件相关的知识中，软件工程绝对是很特别的一个。很多人很鄙视软件工程，说：我一看到软件工程的书就直接略过；与之相对应，很多人很推崇软件工程，会花很大的心思去研究敏捷、CMMI 等。

刚入职场的程序员大致上是讨厌软件工程的，因为这东西离自己的实践有点远，并且主要是添加束缚。但既然更加复杂纷繁的历史都可以总结出规律，忽视软件开发的内在规律无疑的对有志于成为管理者的人是不利的。

真要学习软件工程，不太适合从抽象层次很高的教科书开始，而适合从《代码大全》这样与实际关联比较紧密的书籍开始。

在国内软件工程的落地似乎始终困难，软件工程相关名词始终在不停的变换（ISO，CMMI，敏捷等），但实际能落地起作用的却不多，这最终导致了一种吊诡的局面：刚对一个绝望，就开始对新的一个报以希望，并在这两个简单的步骤上做无限循环。这种状况也许有其更深层次的原因，比如生存压力过于强大导致工程力量的长远价值被漠视，进而使方法论并不为解决现实问题而存在，而是为了证书而存在。很难据此就说软件工程毫无价值。

4. 错过人生中的好时机

没毕业的程序员或者刚毕业的程序员往往感觉空余时间比较充沛，还很苦恼不知道如何打发时间，但实际上一个人一生中可以用于充电的时间远比想的少。一旦错过时机，往往悔之莫及。

对于大多数人而言，人生就像个模板，小处还有偏差，大处却基本相同。

20~30 岁这个阶段可以讲是黄金时期，这个阶段里，家庭负担较小，可以自由支配的时间较多。当然撞到了很特别的、需要疯狂加班的公司只能另算。

30 岁之后因为娃娃出生等，家庭上的时间开销增加，个人可支配时间变少。其中很大一部分人还有很大可能会面对电视剧里常说的婆媳矛盾，让你每天心绪不宁。

40 岁之后，家庭琐事会进一步增加，典型的上有老下有小。实在运气不好的自己也会生点病---颈椎病、腰间盘突出、胃病大概可以入选程序员的三大职业病。

50 岁之后，时间上会再次解脱，但可惜的是自己也老了，时机不在。

如果把人生按照年龄画一条抛物线的话，40 岁左右一个人可以达到的人生的顶点，未来再突破的几率则变小。从历史人物来看，大器晚成的不是没有，但真的很少。

用心观察就会发现，招聘启示里经常会注明年龄要在 35 周岁以下或者 40 周岁以下，除非是招聘高层。这反过来意味着如果没有到高层，人生会在 40 之前定型，之后有下滑危险（如遭遇不景气、公司倒闭等）。对程序员而言，这种风险尤其的大，因为很可能你辛苦掌握的知识体系被更迭掉了。

学习本身无疑的是需要顺应这种自然规律的。

很多人很大的一个错误在于，在黄金时期，没做什么积累，就顾得享受生活了，而一旦意识到积累的必要性时，却又受困于诸多琐事而欲振乏力，最终人生高度有限，并迅速走低。这就是现代程序员版的“少壮不努力，老大徒伤悲”。

基本上讲，35 岁以前要把需要花大量时间，比较硬的技能，学习曲线陡的技能掌握，具备工作所需要的所有主要技能，而 35 岁之后则主要关注知识的更新和某些软技能。

学习时添水战术效率真的很差，每次点一根火柴烧水，一亿年水也烧不开一壶。同时，比较硬的技能（比如：Donald Knuth 的《计算机程序设计艺术》)往往是需要大块时间投入的，但年纪越大时间越呈现为碎片化，越难搞定硬的知识---先天就容易造就添水战术。比较软的技能，则可以用碎片时间来学习，比如：提高 PPT 的制作水平，提高表达能力。

如果能够安排好自己的时间和软硬知识的关系，那么就可以在特定基础上做积累，小步前进，使自己的价值越来越高。从这个角度看，年轻绝对是一种债务，大多数人必须在他没完全结束前，还掉所欠的东西。

那么具体来讲那些东西是比较硬的，要在 35 岁前搞定呢？这因目标而异，但下面这些项目应该具有非常高的通用性：

- 精通一门最常用的语言
- 了解一个最常用平台的基本机制，比如：内存管理、线程机制等
- UML 图和面向对象分析设计方法
- 设计原则，如：职责单一等
- 设计模式
- 《代码大全》里讲的一切
- 精读一个知名的，但有点规模的程序。这点上要感谢开源项目给我们提供了这么多优秀程序。但要谨防好高骛远，动辄挑战 Linux 内核，精读是关键。
- 累积一定的代码量，比如：独立的完整做过一个数万代码行的东西。这里的关键是完全自己打造，一定不要拷贝粘贴。
- 掌握基本算法和数据结构（可以不自己写，但至少要知道其复杂度和区别）

- 养成一种清晰的编码风格
- 有自己的专业（金融、高并发网站，图像处理，TTS 等）

学习英语的时机和必要性

总的来看，程序员学习英语是一项投资回报率相对比较好的投入。从目标上来看，程序员未必一定要口语流利，但最低要达到阅读英文资料没有障碍的程度。这里面有一个微妙的事情，一旦英语阅读问题较大，查找问题会习惯用百度，这天然会限制一个人的视野。不是说百度自身有多不好，而是说英语的世界里有着更多更精彩的内容。不管喜欢不喜欢，我们必须承认一种现实，在 IT 的世界里英语是一种世界语，一方面是由于美国公司的强大，一方面则是由于开源选择了英语。这最终导致 IT 世界里的新动向、解决问题的小技巧、网站的架构等等都要到英语的世界里去找。在 StackOverlow 很容易找到各种小问题的答案，在 Quora 则很容易找到各种网站的架构。

从学习时机来看，这件事情特别应该在大学里面搞定，如果不行至少也要在毕业 1~2 年内达到阅读无障碍的程度，当然希望加入外企还需要额外的付出。从学习方法来看，学习外语真没什么特别的窍门，坚持并投入时间即可。

5. 停止知识更新

对程序员的增值而言，人生里最大的陷阱也许是为安全的假象所欺骗而彻底的放松自己。这种状况在生存环境比较恶劣的情形下不太会发生，但在垄断企业或某一领域中绝对领先的企业里则容易滋生。发现自己是否停止知识更新了并不困难，比如：一年一本书没看，一年一点新知识没接触，一年中工作负荷基本不满等都可以成为一种信号。

这真的是温水煮青蛙，一旦到了三十几岁，并在这种环境中呆习惯了，那么再想跳出来，基本没可能。唯一能做的事情是，祈祷公司不要挂掉，公司也不要来场运动，进行人员的大换血。孔夫子说：日当三省吾身，这是很有必要的，至于认识危险后能否做点什么，那就是事在人为了。

- 技术人员的知识更新

接触一个新的岗位后，大致要经历一个学习并逐渐胜任的过程，这个时间段里大多数人的学习热情是很高的。一旦基本胜任之后，事情就有了变化。

很大一部分人可能会感觉，反正工作也就用到这么些知识，学习其他的也用不上，因此开始把自己封闭起来，不太看书，不太看技术新闻。

这其实很危险，因为这种做法等于把自己绑死在当前这份工作上。而任何一个产品都有自己的生命周期，一旦一个产品的生命周期结束时，碰巧其所用的技术也已经过时，那么当事人就会很尴尬。因为产品可以结束，生活却还得继续。

这里面一个非常经典的例子是 MFC。微软的这款产品的历史非常悠久，从 1992 年发布到 2012 年几近存在了 20 年时间。随着 90 后程序员的逐渐出现，马上这款技术就要变得比程序员的年纪还要大了。

即使到今天，很多桌面应用仍然是基于 MFC 开发的，这可以通过查看程序包的 dll 依赖来很容易的进行验证。MFC 是一个很大的池子，有深度、有历史。想把 MFC 的类继承关系、消息机制、框架结构、RTTI、序列化都搞清楚还是要很花一点时间的。

现在我们假设一款庞大的企业应用是基于 MFC 开发的，一个程序员也通过几年的努力了解了 MFC，了解了应用本身，并可以负担起 Bug 修正，新功能追加等任务了。

接下来这个程序员似乎没什么好学的了。因为 MFC 的更新几乎已经停滞，因此对 MFC 的学习几乎不需要花太多的时间了。现有代码也理清楚了，也不需要再花很多时间学习了。

现有程序也比较好的满足了企业的需求，推倒重来的可能性几乎没有。

那这个时候这个程序员不需要学习了么？答案一定是否定的。

这里面蕴藏着一个天大的矛盾。

从企业的角度看，一定是需要一个团队来维持这个程序的开发的。但从个人的角度看，如果把所有的青春都耗费在老技术上，那么一旦老技术退出历史舞台，个人该何去何从？

还是上面的例子，假设说一个人持续投入在这类开发上，当他 45 岁的时候，当前产品生命周期结束，世界变的只有移动开发和云端开发，那么只擅长 MFC 的他该何去何从？

如果真的如此，这个人就被逼到了死角里，人生很可能产生巨大滑落。所以一定不能认为所学足够而停止技能的更新与学习。

从具体应对措施来看，一是要参照知识的地图，横向扩展知识的广度，比如不只要盯着代码，也要了解业务；不只关注开发也关注一点估算；二是提升可流动性比较好的东西的掌握程度，比如：面向对象分析与设计，这样跨越到其他技术时就能够比较平缓的进行过渡。三是要争取轮换岗位，争取多种实践机会。

- 管理者的知识更新

到现在为止大部分人认同，管理者是需要懂技术的。从逻辑上看“懂”基本上是不瞎指挥的前提，所以这可以称为中国版的“现场主义”，估计争议不大。

那关键问题就是究竟要“懂”到什么程度？

如果说两个人，一个选择了管理方向，一个选了技术方向。接下来要求管理方向上的人技术水平要和技术方向的一样，那么除非这个人特别天才，否则不太可能。正像前面所说，这是由于这两个方向的“Key”不同所造成的。

如果把目标设定为确保最终产品的成功，同时假设管理者有更高的决策权，那么管理者必须在下面这些方面有技术感觉。

从做产品来看，要想成功，有两个关键维度需要同时进行把握，一是产品的概念完整性的把握；一是用合适的手段去实现这个产品。

前一个话题很老，《人月神话》就有提及，但实践中却总是被人忘记。好的产品必须贯彻某一种统一意志，iPhone、微信又重新验证了这一个老的原则。 机械拼凑的产品虽然融合了很多人的想法，但往往是平庸的，并且在项目执行过程中，往往是出错的根源。很像是虽然有法律，但每个人有自己的理解，各行其是这样一个状态。这种概念完整性是管理者第一个需要有所把握的事情，其次就是解决如何去构建产品这个问题。为达成这一目标在下面这几个方面上，管理者要有自己的理解，至少要有自己的原则：

下面简单列举几个比较关键的考量，这和前面论及的如何往博的方向发展有点重叠：

- 使用现有产品还是自己开发

  比如：那些模块适合自己搞定而那些购入就可以了。购入的时候要遵循怎么样的标准去选择。
- 使用那种平台技术

  比如：是使用微软的技术，还是开源的技术。
- 现行架构是否可以达成产品目标

  比如：在硬件加软件可以同时支撑的并发数目。
- 代码可维护性如何约束

  这要求必须熟练掌握一些原则性的东西，比如：什么信息隐藏、正交分解、抽象是否充分等。以及一些无歧义指标，比如：圈复杂度，单元测试的收益平衡。
- 那些环节必须固化为流程，那些一定要团队自由决定

  比如文档化要到什么程度才合适，不同阶段间什么是必须的输入输出。
- ……

假设说有人不这么认为，而是在做了管理后，表现出足够的惰性，不再持续更新自己的知识体系了，那么会发生什么事情？

这时候会很可能会管理倒置。即管理者是名义上的上级，但基本失去对现场的把握，所有的决策完全依赖于下属。得力下属不在，各种决定就只能靠瞎蒙，最终变成只会沟通的管理者---即使被食人族吃了也不会有人注意到，因为存在价值已经被无限稀释，变成了一个象征性的符号。也可能会和下属爆发激烈冲突。因为这类管理者没有自己的立场，上面有任务只能下压。结果同实际情况偏离万里，不具有可实现性，这类管理者无法对自己的上司陈述，也就只能向下转移压力。

不管是那种，一旦到这种地步，其实是趋于失败，只能祈祷食人族不要来。

为什么中层管理者也要坚持知识更新？

在 IT 行业流传着一个很有名的关于食人族的笑话，这个笑话说的是：

两个食人族的人应聘进了某家大公司，公司人事主管知道这两个这伙每天都要吃人，于是警告他们：“如果你们胆敢在公司吃一个人，你们就会立即被炒掉！”两个食人族唯唯喏喏地答应，表示绝不会在公司吃人。两个月过去了，公司平安无事。

突然有一天，公司发现负责打扫公司卫生的清洁工不见了。于是人事主管非常气愤，找来两个食人族怒斥，并当场炒掉了他们。出了公司大门，一个食人族马上对另一个抱怨起来：“我一直警告你不要吃有在做事的人，你就是不听！我们两个月来每天吃一个经理，没人发现。你看现在吃了清洁工，他们马上就发现了！你真是个猪！”

这个笑话嘲讽的是某些大公司大企业病发作，人浮于事。大企业病的成因很难一下子说的清楚，但结果却比较明显，一定会导致较多人成为中层管理者。如果说成功的企业天然有感染大企业病的趋势，那无疑的中层管理者也天然有着膨胀趋势。从个人角度看，成为被食人魔吃掉也没有人在意的经历并非是什么好事，因为这意味着存在价值减弱，也不需要什么知识更新。一旦面临裁员这类事情，这个人很可能已经失去了面对残酷竞争的能力。

4.3.4 给自己找一个驱动力

在国内有一个很特别的现象：高中打了鸡血一样使劲学习的人，到了大学往往会放松下来，跑去享受生活了，大学里打游戏、打牌、临考冲刺的大有人在。

这倒也在情理之中，高中不学真考不上大学，考不上大学真就完蛋了；大学不学，只要能毕业，大致还能找到工作。但人生是场长跑，稍一停顿，后面的人就嗖嗖的窜到你前面了，虽然你不一定能看得到到底是谁。这很像龟兔赛跑，但为了避免不成为那只兔子，单靠口号层次的主观意愿是不行的。

停下来的人其实很多，有的人停在了大学里，有的人停在了小有成绩之后，而为了持续前行而成为高手，那首先要给自己找个动力源，而要想找到动力源，那么首先要拷问自己的内心，知道自己为什么上山学艺，为什么要成为高手。

1. 纯物质上的驱动力

如果回看历史就会发现历史记下名字的人（孔子、商鞅、汉武帝、李白、王安石等等），往往具有一个共通的特征：他们具有澎湃的生命力，绝对不是每天混混日子的人，虽然生命力的表现往往不同。

这似乎和好人坏人，理想高尚与否无关，必须的只是这个人要有所执著，并持续的运

用你的脑力、体力去做某件事情。而一个人要想有所执著，那就必须有一种持续存在的驱动力。

在这个时代里，对于大多数人而言第一驱动力无疑是物质及金钱，但这似乎只可以做起点而不能做终点。这就和梁山好汉一样，最开始没准只是为了大碗喝酒，大块吃肉，可聚了义之后，则要考虑替天行道，除暴安良这类比较高端一点的事情了。

组织行为学中的研究佐证了这一结论，美国人的研究显示，当年薪超过 40000 美元后，薪资将不再是工作满意度的核心支撑因素。

在中国估计相差不大，20 万以上年薪的人，往往需要寻找其他的驱动力。
否则这个时候程序员会处在一种瓶颈状态：基本物质需求得到满足（比如：住房、汽车等），但更高一层的物质需求又很不现实（比如：别墅、游泳池等）。

这个时候单纯的物质支撑会让生命力变的萎靡，人也就会变得无可无不可，工作中即不会很高兴，也不会很不高兴，每天也只是例行公事。接下来人生就会开始下滑，倒霉的话还会失业。等到困境真的来了，再想振作，却没那份儿心力了。

在 2012 年 11 月，新闻报导百度总裁李彦宏先生在公司内呼唤狼性，排斥小资。具体来讲，呼唤的是敏锐的嗅觉、不屈不挠奋不顾身的进攻精神，群体奋斗，排斥的是信奉工作只是人生的一部分，不思进取，追求个人生活的舒适的态度。这事情的是非很复杂，这里略过不提，反倒是百度这一行动本身说明公司里确实容易存在一种生命力萎靡的状态，这点甚至让 CEO 忍无可忍。

被激发的“狼性”是否可以成为一种长久的驱动力？

狼性似乎是一种很中国化的提法，就公开资料来看并没有见到 Google，Apple，微软这类的公司提倡狼性，反倒是本土的华为、百度先后走到这个方向上来。那么狼性是否可以成为人生的一种有效支撑，成为一种持续的动力源泉？

我个人认为这只是阶段性的企业精神，而并非一种长久之策。

什么是狼性，狼性也许表现为敏锐的嗅觉、不屈不挠奋不顾身的进攻精神，群体奋斗，但其根本驱动则是生存所面临的巨大压力。

生存威胁是所有狼性表象的动力之源。而从马斯洛等心理学家的归类来看对水，对食物，对基本安全的需要处在需求的最低层次。

想用它来驱动一个已经处在较高物质生活水平的人，那么意愿和现实间就会产生巨大的矛盾的。很多高科技公司里的高层，乃至中层是相对富裕，远离生存压力的人群，已经失去了拥有狼性的根本，不太可能仍靠狼性来做驱动。

简单来讲，它也许更适用于苦大仇深时期，适用于水深火热中的人，在这个前提下，它也许可以激发出人们高昂的斗志。从这个角度看，处在食物链高端的公司，不应该进一步提倡狼性，而是应该和它渐去渐远，并寻找其他的有效支撑，但这确实很复杂。

2. 兴趣的力量

如果说纯物质上的驱动有一定局限性，那么兴趣又如何？
当问及一个人为什么从事软件开发时，很多人会回答是：兴趣。但人们这么表达的时候可能并没有意识到常说的这种兴趣是一个不怎么靠得住的驱动力。这种兴趣往往是一种一时的好奇心，而不是与自己性格特质相契合，可以用来给生命解闷的那种兴趣。

总的来看，兴趣可以分为两个层次：一个是浅层次的。比如看到一个游戏比较酷可能

想玩玩，看别人写博客，自己也写几篇；另一种则是深层次的。比如：爱因斯坦你不让他思考，他可能感觉活着就没什么意思了。

找到后一种兴趣的人是幸运的，但大多数人并没有这么幸运，所以一般所说的兴趣都是前一种，尤其是即将毕业和刚毕业不久的人。

真正做软件产品的时候，牵涉的往往是大量比较繁琐的细节，大多时候脏活、累活、苦活远比看上去风光的活，有趣的活多。具体一点讲就是，可能需要面对比较垃圾的既存代码，可能要面对不怎么负责的同事，无理的需求变更，十分紧张的日程等。除非你做的是很高端的研究。

这些东西本身并非是很有趣的，它们很快会让初始时的一点夹杂着好奇心的，浅层次的兴趣消耗殆尽。最初印象和实际的偏差很可能是造成某些程序员提前退场的一个主要原因。

这种原生态的兴趣消失后，又没有其他支撑点的话，人就越来越会像机器人，并可能会厌恶自己的生活。因为每天大部分时间都花在工作上，而工作本身又是如此的无聊。所以故事到兴趣这里并没有结束，还需要继续找寻驱动力。

当然很可能以兴趣为出发点而找寻其他，最终反倒又回到起点，这是人生常态，倒不值得惊讶。

兴趣、体力与智力

人和人之间的体力往往相差不大，但智力的差异却往往不可以道里计。所以一个经验丰富的老农半天种一亩地，一个不太熟练的忙和一天大致也可以搞定一亩地。

纯体力上不太拉得开人的差距。而智力则不一样，人的智力容易有数量级上的差异。前苏联著名物理学家、诺贝尔获得者朗道将世界上的物理学家从一级到五级分为 5 个层次，每个层次的贡献差一个数量级。朗道认为自己是 2.5 级，获得诺贝尔奖后，才把自己生为 1.5 级。就不用再考虑 5 级的物理学家和普通人的差异了。

我个人感觉，越是靠近体力一端的工作越不可能兴趣驱动，而越靠近智力一端的工作越可能是纯兴趣驱动。恰如我很难相信干重体力活的人是因为兴趣一样，我也很难相信爱因斯坦不去做以色列总统而选择继续研究物理不是因为兴趣。

程序员的工作在这两个极端间，所占跨度较大，但产品开发也许是中间偏左，完全的兴趣驱动估计很难。即使是兴趣驱动，也不是我喜欢游泳，我喜欢吃小笼包这个层面的兴趣。

3. 使人生永动的势能

如果说物质和单纯的兴趣不足以成为一种长久的驱动力，那么无疑的我们需要继续去寻找一种可以使人生永动的势能。

很多一部分程序员其实是认识到了技术更迭这样的特征时刻在淘空自己立足的根基的，但很多的人在这种时候并不是采取积极的态度去面对，而是会试图欺骗自己，给自己一种安全的假象。比如：可能会告诉自己，反正公司短时间没问题，把手里的事做好就行了。让自己那么辛苦干什么？这在某些有点年纪生活相对安稳的程序员身上比较常见，这很可怕，有时候会把自己逼到死角里。摆脱这种状况当然需要一定的危机意识，但更关键的是要找到一种使人生永动的势能。

据说美军的麦克阿瑟将军非常喜欢一篇名为《青春》的散文，在占领日本期间日本人在美军总部发现了这篇散文，于是这篇文章很快变得很流行，在商界大佬（如松下幸之助等）间流传甚广。那《青春》这篇散文说的究竟是什么呢？我们来一起看一下：

青春不是年华，而是心境；青春不是桃面、丹唇、柔膝，而是深沉的意志，恢宏的想象，炙热的恋情；青春是生命的深泉在涌流。

青春气贯长虹，勇锐盖过怯弱，进取压倒苟安。如此锐气，二十后生而有之，六旬男子则更多见。年岁有加，并非垂老，理想丢弃，方堕暮年。

岁月悠悠，衰微只及肌肤；热忱抛却，颓废必致灵魂。忧烦，惶恐，丧失自信，定使心灵扭曲，意气如灰。

无论年届花甲，拟或二八芳龄，心中皆有生命之欢乐，奇迹之诱惑，孩童般天真久盛不衰。人人心中皆有一台天线，只要你从天上人间接受美好、希望、欢乐、勇气和力量的信号，你就青春永驻，风华常存。

一旦天线下降，锐气便被冰雪覆盖，玩世不恭、自暴自弃油然而生，即使年方二十，实已垂垂老矣；然则只要树起天线，捕捉乐观信号，你就有望在八十高龄告别尘寰时仍觉年轻。

---塞缪尔-厄尔曼德

文字非常优美，但意思并不深奥。青春是一种进取的精神，是一种远离颓废追逐理想的状态。我个人非常认同这种观点。

人的思维和欲望具有无边界特质，只要在未来和现在之间制造一种差距，那么就会产生无尽的势能，人也就会不断的前行。而制造这种差距的最佳素材往往只能是理想。

大多数人可能误解了理想和道德间的关系，但实际上理想并非是一个高尚的词汇，与善恶无关，更与宏大与渺小无关。

鲁智深说：平生不修善果，专爱杀人放火，这也可以是种理想，只要你可以承受它所带来的负能量并不感到痛苦。物质需要、成就需要、权利需要、归属需要这些动机理论中经常提到的东西，乃至前文提到的兴趣都可以成为理想的素材。所以反过来讲，理想是什么似乎并不关键，关键是要有，并且你真的可以很狂热的投入去做，这就可以产生一定的势能。

对与天生对代码狂热的程序员而言，这并不是什么问题；但对大多数人这种理想往往并不在程序之内，而在程序之外。这时候很可能需要叠加几类东西才能给自己蓄积足够的势能：物质的需求、成就的渴望、不安全感的驱离、技术上的追求等都是不错的素材。

当然这是个人的私有领域，最终只能由个人做出选择。

这里最后想说的是有理想、有斗志不一定会成功，但无理想、无斗志几乎一定会铸就平庸和失败，因为细致想来世界本身归根到底是理想主义者的。

这可以通过一个简单的逻辑游戏来做点证明：

意识决定行动：个人意志决定个人行止，组织意志决定组织行为→理想主义者个人意志更为鲜明，自我意识强烈，而无理想的个人意志薄弱，愿意随波逐流→意识强者上位→处于组织核心地位的是理想主义者 →无理想的人不论在个人层面还是组织层面都沦为追随者→ 从资源掌控的角度看理想主义者掌控世界。

动机理论与人生势能

如果把我们前面提到的“势能”做细致分解，那么你可以得到各种各样的动机理论。动机理论却真的可以解释现代人种种行为，因此在这里对其做一点介绍。

- 马斯洛的需求层次理论

  动机理论里最有名的可能是马斯洛的需求层次理论，即：

生理需要：包括觅食，饮水，栖身，性和其它身体需要。
安全需要：保护自己免受生理和情绪伤害的需要。
社会需要：爱、归属、接纳和友谊。
尊重的需要：自尊、自主和成就感；地位、认可和关注等。
自我实现的需要。

- ERG 理论：

存在需要(Existence)：与马斯洛的生理、安全需要相似。
关系需要(Relatedness)：与马斯洛的社会及地位需要相似。
成长需要(Growth)：与马斯洛的自尊及自我实现需要相似。

- X 理论和 Y 理论：

X 理论持下面四种假设：
员工生来不喜欢工作，只要有可能，他们就逃避工作。
由于员工不喜欢工作，因此必须采取强制和控制措施，或采用惩罚威胁他们从而实现目标。
只要有可能，员工就会逃避承担责任，并寻求正式的指令。
大多数员工把安全感视为高于其他所有工作相关因素，并且没有雄心壮志。
与之相反，Y 理论持四种积极的人性假设：
员工视工作如同休息、娱乐那样自然。
如果员工承诺完成某个目标，他会进行自我引导和自我控制。
通常人们都能学会承担责任，甚至会主动寻求责任。
人们普遍具有做出创造决策的能力，并不仅仅是管理者才具备这种能力。

如果把 X 理论和 Y 理论对应到 IT 行业，那么提倡人件和敏捷的基本是基于 Y 理论，而提倡 CMMI 和大棒加胡萝卜的基本是基于 X 理论。

- 双因素理论(two-factor theory)

也被称为激励-保健理论(motivation-hygiene theory)。这个理论认为激励和让人不烦（保健）是两码事情。满意与不满意不是对立的两极，满意对立的是没有满意，而不满意对立的则是没有不满意。简单来讲管理质量、薪金水平、公司政策、工作环境这些被认为是保健因素，他们不产生满意，只产生不满意或没有不满意两种状态。要想激励员工必须在晋升、个体成长上有所作为。

- 麦克莱兰的需要理论

这个理论把人的内在需要归为三类：成就需要、权利需要和归属需要。成就需要是一种追求卓越、获取成功的需要。权利需要则是一种控制他人行为的需要。归属需要则是建立良好人际关系的需要。

- 认知评价理论

这个理论比较神奇。他认为工作本身的乐趣是已经是一种内在的奖励，而外部奖励反倒会降低动机的水平。这听着比较抽象，但意思是如果总是用奖金配合绩效考核来激励员工，那么个人由于兴趣而从事某项工作的动力就会降低。这很难理解，但确实也有证据来支持这个理论。

- 目标设置理论

由于很多公司会做目标管理，所以很多程序员对这一理论反倒不陌生。这一理论大意是说：明确而具体的目标有助于提高个人效能，目标的挑战性、针对目标的反馈都在此基础上进一步提高目标的效果。

- 自我效能理论

这个理论走到极端就是常说的“人有多大胆，地有多大产”。大意是说一个人对自己有能力完成某项任务的信念越强，那效能越高。或者简单说就是自信很重要，而促成自信的方式可以是：过去的成功经验、从榜样吸取力量、受到言语激励等。

- 公平理论

公平理论是说一个人总是会拿自己的产出收入比和另一个人比较，如果比率相似，那么人们会认为环境是公平的。

这落到个人头上，就是我们要重视这种差异么？都是同样的学校、同样的智商、同样的家庭背景，如果一个人收入等比你多很多，你在意么？

- 期望理论

这种理论把人的动机分解成几个部分：一是如果我努力了，那么这种努力能否在绩效评估中表现出来？二是如果我获得了良好的绩效评估，那么能否得到奖励？三是即使我被奖励了，这东西是不是我想要的。

这些动机理论从公司的角度看是激励员工的方法，从自己的角度看则是努力前行的动力。我还是上面的观点，这些理论知道就可，但如果无法找到对自己内心进行触动的点，理论在这里并没有很大的价值。如果想进一步了解，则需要阅读组织行为学相关的书籍了。上面的内容则主要参照了人民大学出版的《组织行为学》一书。

最后想说的是单纯从境界上来说似乎有一种精神远远超出动机理论所能涵盖的范围，这种澎湃的生命力和永不屈服的精神，虽然年代久远，乃至失去传承，但这种精神却正是上古华夏民族所身体力行的精神。

何新先生为自己国学经典系列写了一个总序，对此进行了很好的描述，他说：

华夏民族的先史中有一个神话时代。这个时代实际就是华夏民族肇始和文明滥觞的英雄时代。

女娲是敢于蹈火补天的英雄。伏羲、神农、黄帝、炎帝、蚩尤、大禹，或创世纪，或创文明，或拓大荒，或开民智，或奋身为天下法。

鲧与大禹父死子继，以身济世，拯黎民于水火；蚩尤共工九死不悔，虽失败而壮志不屈，天地为之崩陷。

夸父逐日，体现了对神灵的藐视。而精卫填海，则表现了对宿命的不驯。

由此观之，中华民族的神话先古时代，实在是一个群星灿烂的时代，慷慨悲壮的时代，奋进刚毅的时代；是献身者的时代，殉道者的时代，创造英雄和产生英雄的时代。

传说中华民族是龙和凤的传人，而龙凤精神，我以为就是健与美的精神。

我个人总是感觉很难用动机理论去解释这种宏大且壮阔的精神。

§4.4 小结

IT 这个行业确实是年轻人更容易出头的行业，如果不信，去看看腾讯和银行高层的年龄吧。

一个企业兴起的时候往往会带着一批人走上成长的快车道。在这类企业里没有太多的历史包袱所以占据高位的往往是些年轻人，这和有个几十年历史的行业非常不同。

而为了达成这一目的，程序员首先必须有效的持续增值并成为某一方的高手（架构师、Guru 或者优秀的管理者）。单纯从技能上讲，你一定不能和金字塔最底层的百万大军竞争，想象一下一个刚学会某门语言的人和一个每天思考虚拟机原理与设计、设计跨平台系统的人有多大的差距。而从公司的角度看，高手则意味着可以为公司在现在或将来创生较大价值。

想成为高手就需要选定一个大致的方向，持续深化，并避开学习路上的诸多陷阱，如：失去焦点、分离学习与实践、过散而不专等等。

这也正是前文所重点阐述的东西。

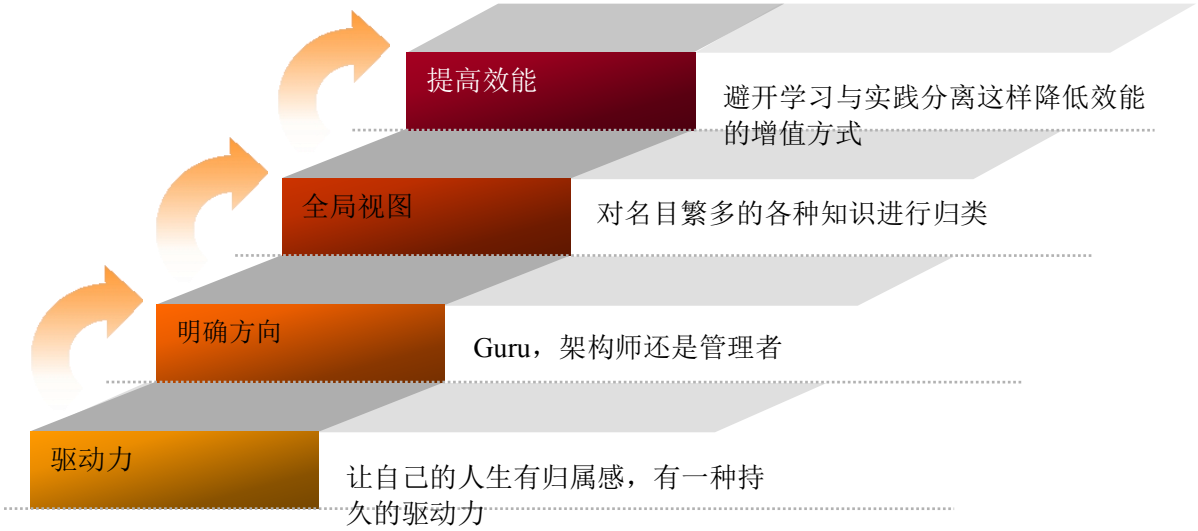


图 4-3：成为高手的关键

IT 行业的魅力

取得巨大成绩始终是个低概率的事情，但只要有理想，愿意持续付出，在 IT 行业中取得成绩确实几率相对会高一点。一个原因是这确实仍然是一个仍在告诉发展的行业，这导致资本会较多的流入到这个行业里来；另一个原因是这是个智力密集型的行业，智力密集型行业里就会有更多的事在人为。

下面我们来看一下 10 亿美元公司的列表。据统计在美国每 3 个月就出现一家 10 亿美元市值的公司，在中国新崛起的公司也大多是 IT 公司。

下面这张表取自 CSDN 的一篇报道以及关联的 Google Docs 上的数据（为缩减篇幅，有删节）。

公司	*创立时间*	*市值/股价*)收购价^(十亿)*
VMware	*1998*	*37 B*
Google	*1998*	*210 B*
PayPal	*1998*	*1.5 B^*
SalesForce	*1999*	*19 B*
LinkedIn	*2002*	*10 B*
Skype	*2003*	*8.5 B*

Facebook	2004	41 B
Evernote	2004	1 B*
YouTube	2005	1.1 B^
Groupon	2008	1.8 B

与我们关联更近的公司状况如下：

公司名称	创立时间	市值（十亿美元）
腾讯	1998年	64.5 B
百度	1999年	36.5 B
网易	1997年	6.9 B
新浪	1998年	3.6 B
优酷土豆	2012年	3.1 B
奇虎360	2005年	2.8 B
携程	1999年	2.8 B
前程无忧	2009年	2.8 B
人人	1999年	1.3 B
搜狐	1998年	1.5 B
巨人网络	2004年	1.2 B
搜房	1999年	1.5 B

*在 Google Docs 上的文档名是：A billion dollar software tech company is founded every 2 months in U.S，在 CSDN 上的转载名称是：在美国，每三个月就有一家 10 亿美元的企业被创立，感兴趣的人可以自己查找并看完整数据。

这些例子已经可以说明，从创业的角度看，IT 行业有得天独厚的条件，尤其是互联网企业。

正是因为这是个智力密集型行业，并且很容易直接面对终端客户，因此一个人或几个人往往可以用很少的资本加上自己的努力撬动很大的事业。这在重型机器、银行这类牵涉大量资本、复杂营销的行业里是不可能的。

所以总的来看，IT 行业还是一个充满朝气的行业，对个人而言要想去的成绩更关键的还是有理想、能坚持，不混日子，持续提高自己的价值。

第五章 程序员的表达力磨砺

前一章讲的主要是提升一个人自身的价值，对于程序员而言，自身价值几乎与技术能力相等价。对喜欢编程的程序员而言，最简单的生存方式也许是最幸福的，比如说：我只把精力放在提高编程技术上，并且工作中也只负责把程序写好，凭借这个也可以让自己物质上比较富足。如果真的可以实现，这无疑是非常让人幸福的工作生活方式，但现实往往并没有那么纯粹，所以这里还是会讲些有用但可能并不讨人喜欢的内容。

前面曾经提到过一个人的表现出来价值始终由两部分组成，一是自身的价值或者说能力；一是表达力也就是自身能力的表现程度。一旦说到表达力、说到沟通、说到公司政治，纯粹的程序员往往是没耐心听下去的，但这确实是一种现实，与喜欢与否关联不大。不同公司间不同的是对表达力的需求程度，而不是表达力是否有其存在价值。

§5.1 表达力的类别和作用

如果一件事情真的是非你莫属，那么作为需求方的企业会尽一切努力来从茫茫人海里把你挖出来，这时候表达力反倒不太关键。这种可以叫卖方市场。想象一下，你的简历中如果只有一句话，如：我创建了 C 语言或者我开发了 Linux 的第一个版本，那么这时候是不需要说别的说明的。

但在更多时候，具体到某个程序员，他往往只是众多可选项之一，总是有着 A、B、C 可以替代你做同样的工作。这时候他就会面临竞争。这种叫买方市场。越有吸引力的职位，越是买方市场。

前者总是很难达到，毕竟杰出的天才程序员并不像想的那么多，这样的话，表达力就有了价值。

在公司里，如果你稍微用点心进行观察，就会发现，CEO、CTO 只有一个，总监级的人数要少于部门经理，部门经理则要少于项目经理，项目经理要少于程序员。当然这只是管理路线的情形，纯技术路线在不同公司间差别较大，不太好比较。但样一种进阶路径已经足以说明竞争之惨烈，高职位总是稀缺的。在竞争高级职位时，往往并不会有唯一的决定性因素，一旦有缺乏表达力这样的明显短板是很难一层层脱颖而出的---因为待选者很多，你有技术，他也有技术，你没有表达力但他有表达力，这种情况下显然他的机会更大。

前面曾经提到过，表达力和技术技能不是一种对立关系，而是一种叠加的关系。从纯量的观点来看，叠加后的高度，才是一个人表现出来的高度。超级程序员单只技术一项就已经足够高了，他们真不用理会这事儿，但大多数人不行，大多数人在技术达到一定高度后可能会碰到瓶颈，因此需要从其他方面做弥补，这样才能确保最终表现出来的高度。

一提表达很多人会想到沟通，尤其是言语沟通，但这是一个非常巨大的误解，表达力并非只限于语言，现实亦可为表达作证。当面对一个陌生人的时候，更多人会采用察其言观其行的方法。这时候往往行重于言。也就是说，但凡影响一个人在他人、在组织里的形象的东西皆可视为一种表达。

这不意味着言语沟通没有价值，而是说言语需要有现实进行佐证才更有力量，表达力正是言行合一后所形成的力量。大多的程序员不善言辞，但很少有人对此反感；反倒言行分离者容易遭到一致的排斥。其实大家都知道说很容易，而做很难，从时间轴上来看，“说”赢的是短线，而“做”赢的是长线。其实倒没什么可以偏废的。

在下面的章节里里我们所主要关注的并非是如何改善自己的沟通方法和言辞，而是如何提高自己言行合一后的表达力。这种表达力对任何人都是必须的，但对管理方向上的人帮助会更加明显。

证书是表达力的来源之一么？

诚然证书也是一种表达自己的方式，但我总怀疑软件开发，项目管理的证书是各种证书中含金量最低的。至少我是想不出那个认证能有注册会计师在会计行业里同等的地位。事实是因为注会证书找到高薪职位的常常听说，但因为某个软件考试认证找到特别好职位的就不太听说了。

很可能下面这段文字是偏颇的，但我还是想对此表达一下自己的看法。

证书只在职业的初级阶段会略有作用，比如：一个人如果不是学计算机专业的，但想做软件开发，这时候持有软件设计师这样的证书会有一定的帮助。但长线来看，各种证书似乎只能起一个敲门砖的作用。这也许是因为，一个人程序写的好不好实在太容易检验了。在代码面前基本没什么秘密可以隐藏起来。而对于管理来讲，实践又远比理论重要，所以持有 PMP 证书和能管理好项目的重叠度并不高。这大概就是 IT 行业里证书不太有用的两个主要原因吧。

§5.2 改善表达力的途径

在本书中，我们认为四个因素对一个人的表达力有关键影响，它们分别是：资历、性格与习惯、借势与公司政治。下面将分别针对这四点做一点说明。

5.2.1 给自己一点资历

资历也是一种表达方式。

如果你翻看创业公司高管的简历，往往会发现，有些人曾经披荆斩棘的一起奋斗过，具有着某种特别的资历。如果 CEO 是掌门，那么这些人很像真传弟子，在公司里有着很特别的地位。

马云先生身上曾经发生过一个很有意思的转变。大概是在 2001，马云先生对自己一起创业的部下说：你们只能做小组经理，而所有副总裁都得从外面聘请。 而后马云先生承认这是个错误，他说：公司初创，融了钱就开始犯错，尽招有著名大公司经历的人。寒流起，这群人大都犹如蝗虫，只留下一堆概念、 PPT 和不屑。多年来总有一批"人才"在各公司甚至竞争对手间奔走，加薪升职！其实错在自己。看到十年换过七八份工作的人，就像十年内结过七八次婚，漂亮但无忠诚，您敢娶和该娶吗？

与此同时，柳传志先生曾经强调联想要做不是家族的家族企业。

这两位泰山北斗级人物的表述虽然不同，但其核心其实偏差不大，皆是强调一种对公司的忠诚度和归属感。这应该引起职场中人的注意：在跳槽的同时，似乎有些损失被忽略了。

这个被忽略的东西以往一直被称为资历。在很多人眼里，资历代表着一种腐朽的力量，是应该被后浪打在沙滩上或者被扔进垃圾堆的。但资历本身的力量在过去数百年里却一直起

着不容忽视的力量。这甚至可以扩展到国外，史蒂夫·鲍尔默之所以成为微软的首席执行官，应该不单只是因为能力。

资历本身并不直接增加一个人自身的能力，但他增加一个人可以左右的力量，影响个人力量的表达。程序员很容易就对着程序使劲，而忽略这个维度上的力量。

这可以举个简单例子来做一点说明：

一个人在一个公司工作了 10 年，那么这 10 年中他就会在他所接触过的人中间，建立某种声誉，培养一种信任。这种声誉和信任都会转化为做事时的润滑剂，很多人可能更愿意去配合他。但一旦他换了个环境，可能原本只需要打个招呼的小事情，就需要一个上午的解释。

资历这东西位置越高，背后隐藏的力量越大，但即使是对于没成为管理者的程序员，资历中也还蕴含着力量，比如：

总是很投入的去考虑多线程、数据结构、出错处理的人，确实有些方面会疏忽，比如思考的太投入，就忘记和其他人联络进度了。这个时候从管理者的角度看，可能表现为一个被分配的任务，既没有被完成也没有进行预先联络，这就很可能被曲解为对自己的不尊重，接下来就会出矛盾了。这类事情往往可以被资历化解无形。

用心来想，资历之中主要包含了三个方面的力量：忠诚、信任与对既有规则的熟悉和对既有人员的熟悉：

- 形象一点来讲，假设说一个人白手起家开创了一番事业，有一批人风里雨里跟随十几年，甚至几十年，同时另外有一批人，假设能力上比老部下略微优秀一点，那作为领头的人会因为这种优秀把老部下都换掉么？这固然是因为人与人之间终究还有一种情分在，但往往这并非主因。一个人最终表现出来的能力事实上有两个关键支撑，一是能做什么所对应的能力，一是工作意愿，两者相乘才是一个人表现出来的能力。当归属感强的时候，工作意愿上会有绝对优势。一个私心过重的人，能力越优秀，位置越高，对公司的损害可能就越大。

- 生人和熟人之间的主要一个差别是信任程度不同，一个人在生命中其实是在不断的累积自己的声誉，但这种声誉的作用范围往往是有限的。而资历正是反映了这个影响范围的一个东西。当一个人很好的理解另一个人的时候，他就会知道这个人会用心做事，那个人马马虎虎，合作的时候心里也就更有底，因此大多时候人都喜欢和熟人来往，背后其实是一个道理。

- 没有那个公司会把事情事无巨细都写在纸上，资历可以帮助人们很好的理解这部分事情。这也不难理解，A 和 B 可能性格不合并小有恩怨，不了解的项目经理可能就会把这两个人安排在一起做沟通很多的工作。

上面说了很多，但结论其实很简单：资历有助于体现个人能力，所以要给自己一点资历，只在彻底没有希望的时候换工作。否则会损失掉资历背后的力量。

那究竟在什么情形下，应该舍弃资历背后的力量，而另谋出路？总的来看，一个人应该在自己价值无法增长且自己不愿意混日子的时候换公司。

造成价值无法增长的原因有很多：

- 公司开始衰落，看不见自己的未来
- 做的是非主营业务，遭到明显歧视，看不到希望
- 公司内斗太厉害，显失公平，完全没人敢说真话了，都在搞政治
- 无所事事或做的事情完全的简单重复
- 公司需求和个人意愿完全对立
- 你虽然水平一般，但在公司里已经是绝顶高手了
- 感觉每天活得特别虚伪，很累

出现了上述场景大致上是该换工作了，反之就要斟酌。出现了上述情形，不换工作是一种误判。但误判却不只是“该换不换”这一种。反过来“不该换换了”也是一种误判。后者这类误判比较常见的是：

- 因为薪资、福利的小幅增加而换工作，比如上浮 10%。
- 因为工作无趣而换工作

  这点要补充一点说明，大部分工作都会包含一定无趣的部分，期望工作和打游戏一样有趣是不可能的。所以这里的关键是判断无趣的尺度。而树立这种尺度的关键是看工作和思考的关联程度。

  如果一项工作已经是需要很多思考才能做好，那么如果对这类工作感到无趣的时候，要适当做些自省，因为很可能下一个工作还是同样的无趣。比如说：如果你不喜欢用 C#，那么很可能你也不喜欢用 Java。因为这类问题，往往并不是出在工作有没有趣，而是出在了对有趣无趣的定义上。

  如果一个工作真的只是简单的重复，不需要动脑，那无趣就真是工作本身的问题。
- 因为某件事情受点委屈而换工作。比如遭受了批评，感觉很没面子。

最后想说的是，不要低估资历的影响而频繁换工作。换工作是可以的，但想通过频繁换工作而提高获取较高职位的几率则是缘木求鱼。想象一下，要什么样的跳槽才可能获得华为前 100 号员工的职位和收入？一位曾经前 100 号的员工，在较早的时期因为误判而离开了华为，在当时无疑的这种跳槽会带来一定的收益，但这种收益往往是有限度的。这位员工在几十年后，非常感慨的说：人这一生，跟对人是非常重要的。这种事情和观点的是非很难评论，但无疑的它是值得职场中人思考的。

资历是东方文化特有的产物么？

我们经常听到很多国外大公司会空降 CEO，因此可能会感觉资历这东西独属于东方文化。但这其实不对，即使在美国很多的公司也愿意选择长时间在自己公司工作的人做 CEO，而非是空降。空降只是没办法时候的办法。

我们熟知的大公司往往如此，微软的史蒂夫鲍尔默是微软的最老员工，苹果的吉姆库克在 1998 年加入苹果，GE 更是历来如此，而新秀们（Google 等）的 CEO 都还太年轻，没到这个时候。

对此 Intel 即将退认的 CEO 保罗·欧德宁的观点很有代表性，他说：“我认为英特尔将继续坚持从公司内部选聘新任首席执行官的策略，董事会对这一策略很满意。只要我们从公司内部选聘新任首席执行官，我们的管理团队会相当优秀。我认为这是最可能的结果。我对公司内部的新任首席执行官候选人非常满意。芯片产业以往的经验表明，可能的情况下应当尽量从公司内部选聘新任首席执行官。如果从公司外部选聘，新任首席执行官需要 2 年时间了解公司文化、同事和运作方式，我们为什么要浪费这 2 年时间，冒这样的风险？我猜测新任首席执行官将来自公司内部。”

上述的两年时间大致可以等价于量化了的资历力量。

当然在没有做一份美国企业 CEO 甄选规则的列表前，得出美国公司也重视资历这一观点具有片面性。但可以换个视角来让论述更充分一点。并进一步查看 CEO 级别上资历的力量。

在管理的书籍里有一本是当之无愧大腕：基业长青。

这书在对大量公司进行调查之后得出了这样一个结论：要想成为基业长青的公司，那么最关键的是要恪守核心价值观、核心目标。而要想维护核心价值观，那么首先要有认同这一价值观的当家人。空降的 CEO 在这一点上是有绝大风险的。

对此，基业长青一书中还提供了一些很有力也有意思的数据:

高瞻远瞩公司从内部人才中培养、提升和慎重选择管理人才的程度，远远超过对照公

司，他们把这件事当成保存核心要素的关键步骤。从 1806 年到 1992 年间，我们发现，只有两家高瞻远瞩公司曾经直接到公司外面聘请 CEO，比率是 11.1%，却有 13 家对照公司直接聘用外人当 CEO，比率是 72.2%。我们持有资料的高瞻远瞩公司的 113 为 CEO 当中，只有 3.5%来自公司外，对照公司的 140 位 CEO 中，却有 22.1%来自公司外。

上文里高瞻远瞩的公司是指可以基业长青的公司，而对照公司则是指无法基业长青的公司。

我想这样一种统计至少可以说明，资历在很多美国公司中也有着举足轻重的作用，而并非只属于东方文化。

5.2.2 去除性格和习惯中的致命缺陷

性格决定人缘，而人缘影响沟通成效，最终影响一个人的表达力。想成为一个道德完美的人是非常困难的，但只要稍微注意，去除一些谁都厌烦的性格缺陷还是可能的。

1. 人情练达

在《红楼梦》第八十二回里有一小段对话很有意思：

袭人道："你还提香菱呢，这才苦呢，撞着这位太岁奶奶，难为她怎么过！"把手伸着两个指头道："说起来，比他还利害，连外头的脸面都不顾了。"黛玉接着道："他也够受了，尤二姑娘怎么死了。"袭人道："可不是。想来都是一个人，不过名分里头差些，何苦这样毒？外面名声也不好听。"黛玉从不闻袭人背地里说人，今听此话有因，便说道："这也难说。但凡家庭之事，不是东风压了西风，就是西风压了东风。"

最末一句鲜明的体现了林黛玉和薛宝钗的性格差异。如果是薛宝钗估计会讲，姐妹们需要互相扶持。从《红楼梦》的故事里也可以看到这两种人格会导致的不同结局。

这对我们有一定的启示意义，我们可以抽象出一个极其绝对的场景：

两个人作为一个团队而存在的时候，如果张三无求于李四，或者张三具有绝对的控制权，那么张三不需要和李四做沟通，只要保持沉默或者命令也就足够。否则的话，两者就需要协作，进行更多的交流，今天是你帮助我一点，明天是我帮助你一点，这样彼此工作上都可以有比较好的进境。

以程序员的工作状况来看，期望东风压倒西风式的绝对控制基本上是不可能的。把自己封闭在某个独立的领域里（比如算法），达到绝对高度，做孤狼型的人倒是可能，但终究罕见。而与人协作，从他人那里获得更多的支持并取得成绩这一事情则需要人情练达。

走纯粹技术路线的程序员之间不需要很多这方面的考量，但程序员也是人，一点情商也不要也是不可能的。

想象一个很常见的场景：

张三和李四同时加入公司，张三的水平高一点，因此李四在遇到程序问题时，总是会问到张三，而张三也总是很热心的给予帮助。有一天，张三有事，下午要请假半天，但有一个功能还没有对应掉。这时张三找到李四请他帮忙。但李四头也不回地说：没空，你自己解决吧。

这种情境下，一般来讲张三会愤怒，在可做可不做的时候会拒绝向李四提供帮助。李四不是不能拒绝，但他应该认识到自己欠人人情，拒绝的时候需要诚恳的表示歉意，解释一下自己的困难。

李四如果持续自己的做事风格，可以想见他会越来越被孤立，也许他的技术能力不断提高，但对他的评价则会下浮。除非有一天他达到了一种别人只能仰望的地步，事情也许会有变化。如果李四想往管理方向发展，那么影响就更为致命，这种行事方式几乎堵死了自己取得成绩的可能性。

对于程序员而言，在这个上面需要注意的点并不多，但有几条传统的智慧还是要的：

- 欠人的要记清楚，别人欠自己的可以含糊。不要认为任何对自己的帮助都是理所应当的。
- 不要为无谓的事情争吵，乃至口出恶言。人与人的关系坏起来容易，修复起来难。
- 要言而有信，确实无法信守承诺时，要主动道歉。
- 不要通过贬低别人来证明自己，也不要因为言辞不当让人以为是在贬低别人。
- 不要恶意欺骗他人。想想当你被恶意欺骗了，你会什么感觉，就知道恶意欺骗别人能造成多大的伤害。

傲气与狂妄的差别

程序员是需要有点傲气的，一点傲气都没有的程序员往往就会失去对技术的追求并失去对自己的信任。这在技术上是很致命的。

但狂妄则是走向灭亡的前兆，要引起警觉。傲气的人会坚持自己的看法，在没有事实和逻辑支撑时绝不轻易认输；但狂妄的人则会在坚持自己的同时贬低甚至羞辱他人。傲气的人大致知道自己骄傲的边界，能够在工作中找出自己的位置；狂妄的人则眼里只有自己，认为公司的规则、所有的同事都得围着自己转。

一旦一个人由傲气转向狂妄，那必然会人嫌狗不爱，这样一来这个人能创造的价值往往会降低，但他的索取却会因为狂妄而不断增加，这就为未来可能的悲剧打下了伏笔。

2. 有条件的顺应环境

中国古代的钱币外形是圆的，但中间则是一个方空，这可以是一种很有含义的隐喻。全无个性的人往往是平庸的，但在那里都张扬个性的人往往是痛苦的。因为公司必然有自己的规则和文化，而这种规则和文化并不会因为某个人而突然发生变化。

我们可以强调职业精神，说拿了钱必须干活，个性完全不关键，但只有这个是不够的。这里面必须把握一种限度，在这种限度下，不只要拿了钱干活，还要努力适应选定的公司。而一旦超过这种限度，那则意味着需要尽快离开，而不是继续的抱怨。这时有两个关键点需要被认识到：一是天下间没有完美的公司；一是要知道那类事情需要顺应。

❖ 天下没有完美的公司

2012 年 CSDN 转过一篇 Facebook 员工对公司的抱怨，其中的几条非常特别：

- Zuck 的过于关注。既然都成为上市公司了，作为公司的 CEO，你主要接头的应该是：投资者、分析师、博学者等。但你仍然和我们这些工程师谈产品的规划和战略！这是彻彻底底的侵吞时间。你忘了你主要的责任是提高公司的股价而不是原材料的加工。
- 太多的决策由工程师给出。有些决定甚至是一个工程师单独下的，更甚至在午饭中就做出了决定。让缺少公司运作经验的工程师去做这些决定是不是太草率了？！
- 对于内部员工的过度信任。

我们有理由相信，完全相反的抱怨也绝对存在：

- CEO 完全不关注技术。

- 工程师没有决策权。
- 员工完全不被信任。

这充分说明，只要你想抱怨，那就总会有可抱怨的东西。这点起源于人思维的善变以及欲望的无边界特质，实属正常。其实事情并没有那么麻烦。喜欢和不喜欢就像天平的两端，临界点就一个：走还是留。

想走的可以尽情抱怨，自不必说，想留的就要适应某些自己并不喜欢的东西。而选择留下来却使劲抱怨则是最不明智的选择。

这似乎很消极，但以人生而论，无法改变的，无法抛弃的，就要考虑如何去适应。想象一下，不管你如何生气，地球也不会围着你转。

选择了留下来，却去抱怨完全不可能改变的东西，进而总是认为自己受到了不公正的待遇，总是满腹怨气，这不可能不影响到工作，也不可能不影响到别人眼中的你，所以说这也是一种表达。

这里其实有个陷阱：越是认为自己怀才不遇的，那就越真的会怀才不遇。当然也可能其实才华也只是自己认为的。这点在容易在有些知名学校的毕业生身上，体现出来。假设说对应某一个学校有一个大致的就业水平，这似乎会对这个学校的学生产生一种心理暗示，他们就应该在某个水平以上的公司里。一旦进入了低于这个水平的公司，心理先天就会有优越感，可能会想：这个人怎么能来领导我？这么多这么差水平的人每天干的都是什么事？逐渐下来就很容易眼高手低，评价也会走低，反倒是越来越沉底。

认不清这点会很麻烦，但凡是多人聚集在一起地方几乎必然是名利场，而名利场中几乎一定有不堪的地方，公司也不例外。总是期望公司百分百与自己的期望相符会导致所有的公司都可以抱怨，进一步导致工作状态变坏，并对自己造成损伤。当然，接受某些自己不如意的东西也是有底线的，这是下一节的话题。

❖ 知道那类事情必须顺应

受到委屈的时候，首先要判定的是环境是否公正，不要因为升职的不是自己而郁闷，更可怕的是升职的人不具备对应的能力---后者说明整体环境有问题，这是更应该引起警觉的事情。

坦诚的讲，大部分人并不具备改变周围环境的能力，而更像行业或者公司历史中的一片尘埃。当一个企业的基因确定，其中所蕴含的力量是无比宏大的，当这个企业并没有突破基本公正的底线时，最优的选择只能是在大多地方进行顺应，而非是消极对抗。

最不应该顺应的东西主要有两个：一个是公司中处处显失公平；一是个人在公司里面完全看不到发挥的机会和未来。这两点对个人未来是致命的，弄不清楚还不只是适应不适应的问题，而是糊涂不糊涂的问题。

其他的东西则大多是要适应的。不要看很多大人物今天站在台上无限风光，但在取得成绩的路上，几乎每个人都调整过自己来适应周围的环境。

据说杨元庆先生曾经在事业挫折时流泪过，而柳传志先生曾经对杨元庆先生讲：当你真像鸵鸟那么大时，小鸡才会心服。只有赢得这种“心服”，才具备了在同代人中做核心的条件。

这里隐含的一层意思是，两个公鸡可能一个尾巴长，一个冠子亮，但这时候人们往往无法区分究竟那个更好。选尾巴长的，冠子亮的可能会抱怨；选冠子亮的，尾巴长的可能会

抱怨。但这种愤怒是格局不够的一种体现，与其抱怨，不如考虑怎么让自己成为鸵鸟。但恰如前面所说，环境要相对公正。

具体来讲，人不能老等着上司变的开明，变得更英明，这些事很多时候，你改变不了。

你想干个什么事，你得自己做准备，把脏活累活都干了，当然大家看不见这些的，能看见的只有成绩。不能老指望自己动动嘴巴，事情搞定，功劳到手。你得去了解，公司里可能不太好的流程，利害关系人可能有些奇怪的想法，这些都得去理解和摆平。因为换个公司它更可能还是这样子。

你可以讲这太烦了，那也 OK，关键是要能接受平凡的结果。做点事情其实远比想的麻烦，即使是在开明的公司里面，唯有抱怨最容易，但抱怨什么也换不来。

适应环境里有一个极端的情形，也很危险：

很多人可能会认为反正我就赚这么多钱，混混日子也没什么，这也算是彻底适应环境了。但这时候，可能没认识到只要这个状态持续五年，诚然你可能赚到几十万，但失去的却是人生最为黄金时期的五年，一生中所有剩下的时间都可能需要为此而背负债务。在相对公平的环境里主动就是人家跟着你跑，被动就是你跟着别人跑。从长期视角来看，主动去做，错了也是对的；被动做事，对了也是错的。所以被动混日子是危险的，做事的时候要尽可能主动。

这点之所以需要针对程序员群体专门一提是因为程序的世界里是非比较分明，但公司里不是的，再怎么优秀的公司里，也需要一些模糊区域。如果用看待程序的眼光来看待公司，那就 Bug 太多了，并且很多时候很多 Bug 你还不能修，还得假设它是对的，并顺应它，简直是岂有此理，但这也确实是一种现实的规则，无论喜欢不喜欢，都要学会给予它一定的尊重。如果你真的很长情，很有理想，那不妨耐心等待，直到有足够力量把你不喜欢的击个粉碎。当然这不意味着，有意见不能提，而是说提了意见没被采纳，大多时候实属正常。

遭遇可怕的上司怎么办？

很久以前看过慕名看过杰克韦尔奇写的《赢》，可能是自己记忆力不太好，书里说过什么大多是很快忘记了，但其中记录的一件小事却记得特别清楚。

杰克韦尔奇在书里说，2004 年在中国的时候，听众中一个年轻女性流着泪问到，“在只有老板才有发言权”的情况下，又有那个商业人士能够实践坦诚精神和推行区别考评制度呢？我们这些在基层工作的人们有非常多的想法。但很多人甚至想都不敢想能把它们讲出来，除非自己成为老板。”

我之所以记得这个片段，倒不是因为问题本身，而是因为一个人会在公众场合哭着发言---这必然是因为心里累积太多的压力。这也让我私下猜测，想必是我们的商业环境里有很多特别之处。

这种特别之处往往会让我们以更大的频度遭遇一个麻烦的问题：真遇到一个可怕的上司，程序员该怎么办？

在细说这个问题前，首先还是要再强调一下选择权。选择权是博弈的基础，而上一章里提到的自身价值则是选择权的基础。这点虽然在后续章节里不会总强调，但他明显比其他因素有更高的权重。

如果真的遭遇了可怕的上司，并感觉遭遇了不公正的待遇。首先倒不是去和他吵一架，而后辞职，而是要先反省下，看看问题是不是出在自己身上，或者说自己究竟有多大责任。

培根说：聪明者反省自身，愚蠢者欺惑大众，还是很智慧的。

这里可能的原因就太多了。

可能是价值观的冲突，你的上司并非只是针对你而是有自己的是非标准和行事原则，找你麻烦只是因为你的价值观和他的不一样。这种时候如果工作本身没问题，可能需要考虑适应，因为你换个工作可能还有问题。

可能是你年少轻狂做人失误，在很多场合对上司过于藐视。要是这种，要看看能不能修补。毕竟如果当前工作很适合自己，并不适合因为意气之争而换工作。

也可能真是上司纯属个人瞧你不顺眼（因为内斗等）或者他自己过于古怪，这种大致没办法，要考虑尽快换个地方。

3. 去除致命的坏习惯

谈习惯的书很多，但基本上是在告诉你，什么样的习惯更好。但在考虑改善表达力时，却要做逆向思维，在这里认清什么样的习惯更差是更加关键问题。很少有人会期望程序员八面玲珑，因此很多程序员的习惯都是可接受的，那么不可容忍的到底是什么？

我们来看一个每天都会发生的例子：

A 是一名程序员，每当他宣称自己的工作完成时，你总是能在他的代码或者文档中发现缺陷。比如：代码中不遵守大家约定好的编码规范，使用文件时可能会使用绝对路径并导致基本测试无法通过，文档中记录错操作系统的名字等。

想象一下，长久下来 A 身上会发生么？很简单，他会逐渐失去周围人的信任，也许 A 的能力并不差，能解决比较复杂的问题，但是做程序的时候，有这样的队友也还是很可怕的。

这类问题并不涉及高深的知识，基本上是因为习惯不好而导致的。这类习惯里充满了负能量，会让周围的人倾向于看低你。会导致下面两种结果的习惯等价于职场上的核弹，如果你有，没准那天会被他们炸的粉身碎骨。

一是忽视细节，这会导致别人认为你不具备做事能力。一是负不起责任，这会导致别人认为你不用心做事。“能力不足”和“态度不好”这两顶帽子只要带上一个，个人前景立刻会变的非常暗淡。

- 关于忽视细节。

  有的人工作习惯比较好，做的时候稳扎稳打，自己做完会做双重检查，表现出来的结果就是工作的一次成型能力强。与之相反，有的人则做事的时候分心，做完之后不做自我检查，表现出来的结果就是小错误很多，在文档上可能就表现为拼写错误，版本号不对，字体混乱等等。总之，让人感觉就是个半成品。能够一次成型其实是一种很关键，也很被看重的能力，而要想保证这个，只能在小习惯上下功夫。

- 关于推卸责任。

  写程序的时候给自己的问题找借口特别容易，因为纯粹的像 int add(int i, int j){ return i+j;}这类代码特别少，总是要用一点别人的东西，因此总是可以在别人的身上找到借口。可以抱怨开源的文档少，可以抱怨微软代码不公开，诸如此类。但其实这一点意义也没有，只会让人认为对工作负不起责任。

上述这两类不良习惯中蕴含着巨大的负能量，是每个人要用心规避的。如果说一个人的天分、才华、知识、能力都像水一样，那么上述这两个坏习惯就像漏勺，不知不觉中就拉低了你可以达到的高度。

5.2.3 善用借势

取他人、他物所长，为我所用的这一面，始终有着不可忽视的价值。在大约 2300 年前，

荀子对此进行了很好的说明：

吾尝终日而思矣，不如须臾之所学也。吾尝跂而望矣，不如登高之博见也。登高而招，臂非加长也，而见者远；顺风而呼，声非加疾也，而闻者彰。假舆马者，非利足也，而致千里；假舟楫者，非能水也，而绝江河。君子生非异也，善假于物也。

——引自《荀子·劝学》

借势这事儿其实很微妙，既可以认为是自我增值的一个子项，也可以认为是表达力的一个子项，但本书里把它归结为表达力的子项，因为它直接影响一个人在他人眼里的高度。

1. 借势的价值

从易中天老师在百家讲坛开讲《品三国》以来，三国又重新热了起来。而提到三国，最不能忽略的环节大概就是三顾茅庐和隆中对。

隆中对里讲：

自董卓以来，豪杰并起，跨州连郡者不可胜数。曹操比于袁绍，则名微而众寡，然操遂能克绍，以弱为强者，非惟天时，抑亦人谋也。今操已拥百万之众，挟天子而令诸侯，此诚不可与争锋。孙权据有江东，已历三世，国险而民附，贤能为之用，此可以为援而不可图也。荆州北据汉、沔，利尽南海，东连吴会，西通巴、蜀，此用武之国，而其主不能守，此殆天所以资将军，将军岂有意乎？益州险塞，沃野千里，天府之土，高祖因之以成帝业。刘璋暗弱，张鲁在北，民殷国富而不知存恤，智能之士思得明君。将军既帝室之胄，信义著于四海，总揽英雄，思贤如渴，若跨有荆、益，保其岩阻，西和诸戎，南抚夷越，外结好孙权，内修政理；天下有变，则命一上将将荆州之军以向宛、洛，将军身率益州之众出于秦川，百姓孰敢不箪食壶浆以迎将军者乎？诚如是，则霸业可成，汉室可兴矣。

这是非常经典的天时、地利、人和分析法。

曹操得天时，气候已成无法与之争锋；孙权是国险而民附，可以说是即得地利也得人和，因此可以为援而不可图；那么刘备自己就要充分利用这三者为自己谋得优势，出路即是把握时机，谋得荆州、益州，而后通过帝室之胄，信义著于四海扩大人和优势，蓄积力量并进一步等待时机，最终成就霸业。

从这里我们可以清晰的看到，天时、地利、人和皆是借势，是巩固或达成目的的手段，其中差异不过是借的对象不同。

从时运借来的势即是天时。所谓天下大势浩浩汤汤，顺之者盛逆之则亡；时来天地皆同力，运去英雄不自由，说的都是天时的力量。曹操把握时机趁着诸侯混战而坐大，那么对刘备而言最大的天时其实已经没有了。

从地理格局借来的势即是地利。《过秦论》提到的崤函之固和上文提到的国险而民附等皆是此类。现代的地缘政治仍是和这一维度紧密相关。

从人借来的势即是人和。隆中对里的内修政理，总揽英雄，乃至成语众志成城等描述的则是人和的力量。所有这些借来的力量加诸到一个人的身上后，就形成了一种力量，表征着一个人的实力。单只是曹操、孙权一个人再厉害又能厉害到那里去。

借势之所以有价值，其根本原因在于人的能力更大程度上体现为均一性。抽象来看，不同的人能力差异一定是有，但远达不到孙悟空和人类这种地步。这样的话，一个人借到东西的多少，往往对一个人表现出来的力量有根本性影响。

孙悟空看到小妖时，可以一棒子打死，碰到猪悟能也可以几个回合搞定，但碰到金翅大鹏，单靠自己就搞不定了，借不借得到其他力量就成为能不能过关的关键。

能借势无论对管理方向的程序员和技术方向上的程序员都是非常关键的一种表达力，

但不同方向上借势的对象不一样。

对于技术方向上的程序员，借势体现为善用各种框架和工具；对于管理方向上的程序员，借势则体现为团队建设和团队协同作战的能力。

借势的一个小技巧

由于古装剧的兴盛，我们经常会看到皇帝顶着满是珠子的帽子出现在电视里，心里俏皮的人估计会想，带那么个东西能舒服么，看人还看不清楚。

这还真就猜对了，给帽子上装个帘子的目的就是提醒皇帝看人不要看太清楚。人天生就是有问题的，你总看人毛病，周围那还有可用之人。偶尔糊涂一点，叫“明有所不见，聪有所不闻”，是一种政治智慧。

这对现实的启示意义是，你要想借势，那就要在很多时候糊涂一点，包容一点，总是心思太细，眼睛太亮，那就很容易与所有人疏离。

但这反过来什么事情都糊里糊涂也还是不行。郑板桥写难得糊涂，那是风雅，因为他一辈子聪明。一般人写难得糊涂那就是真糊涂，因为大多数本来就糊涂，一点也不难得。

2. 借势的具体方法

落到具体的个人成长上面，谋求加入处在高速发展期的公司是择时，比如：创业初期加入公司和稳定发展期加入公司，后面的成长一定会不同。

考察工作地点对个人发展的影响则是导入地利，比如：二级城市和北上广的可能发展路径明显不同。

上述两点将在第七章进行展开，这里关注的主要是在人和以及技术层面如何借势。管理方向的程序员主要要考察前者，技术方向的则两者都要考虑。

- 人和层面的借势

人和层面的借势可以分为两类：一类是处理与自己直接接触的人的关系；另一类是处理不直接接触，但通过社交网络可以接触到的人的关系。

（1）直接接触人群中的借势

程序员这个群体并不复杂，人们更愿意按照朴素的规则进行做事，比如：你很牛，我就很尊重你，我也就更愿意和你一起做事。所以对同级和平级的人员而言，借势的基本前提是要让自己有一定高度。但这里有个小陷阱，需要引起注意。程序员大多性格比较简单直接，也不厚黑，但软件本身不确定项很多，也估不太准，因此大多时候讨论很多，争议很多，这个时候不能表现的像个刺猬，一旦有不同意见，容易激动或者争吵。这很容易把四周的人推开，即使你能力很强，一旦如此就成了孤家寡人，什么也借不到了。

对上级而言，要形成的印象也很简单，最基础的是这个人靠谱。优秀与否往往只能靠事实说话，而是不是靠谱，这完全是可以掌控的。比如：言而无信要远比坦承搞不定负面影响大。

一旦正式拥有自己的团队了，那就要时刻提醒自己：有借有还，再借不难。产品和项目可以是第一目标，但除此之外就要尽一切可能为团队成员谋求利益。其中最关键的则是为他人谋求发展和进步的机会。这点上也许有的时候能做的有限，但你有没有用心去做，其实还是很容易看出来的。判断标准很简单，如果同等条件下或者收入略有增幅很多人愿意和你一起换个公司，那么在这点上就非常成功了。

最后说一点有点玄的东西。在特定的生态中，人的行为往往具有必然性，除非特别异常的个体。想象一下，假如说一个公司以 Bug 率为考核指标，那么要么开发与 QA 人员十之八九会激烈对抗。而一旦官僚，那么必然对客户反映迟缓。因此一个人如果能掌握这种规律，那在借势中将会更加顺风顺水。

为了达成这一点，需要看一点组织行为学的书，但只有组织行为学也是不够的，还需要冷静的做些分析才行。

（2）社交媒体中的借势

在曹操生活的年代里，有一项非常有人气的活动：找品评家给自己下批语。著名的“治世之能臣，乱世之奸雄。”之评就是这么来的。当年曹操找到了当时的著名品评家许劭，希望许劭给他下评语。

许劭最初鄙视他比较奸诈，并不愿意评价他，可是曹操找到可乘之机对许劭进行威胁，最终无奈之下，许劭才给出了：君清平之奸贼，乱世之英雄。 据说曹操听完这个之后是大悦而去。而这一评价不知道怎么就变成了“治世之能臣，乱世之奸雄。”。

这个风气至少持续到了唐朝，李白为求赏识，还曾经做了一篇流传千古的《与韩荆州书》，说：生不愿封万户侯，但愿一识韩荆州。

上述几件事情虽然跨越千年，但折射的东西却本质相同，都是风评对一个人的影响。在现实之中这种力量同样非常巨大，但因其引而不发，往往会被忽视。

庞果网 CEO 李炯明在接受 CSDN 专访时谈到了这样一种现象：

中国知名的 IT 企业，例如腾讯、百度、阿里巴巴、盛大等，每年社会招聘中有 30%~45%来自“内部员工推荐”；而根据我们对 2000 名专业人士的调查，46%的被调查对象说他们“最近这份工作”是通过朋友或者相关人士引荐获得的。

李炯明先生说的这个数据与下面这样的一般调查的结果并不吻合，我私下猜测可能待获取职位所需经验与能力越高，引荐起的作用越大。但不管从那个数据看，引荐自身已经成为一种比较关键的换工作途径是一种没什么争议的事实。

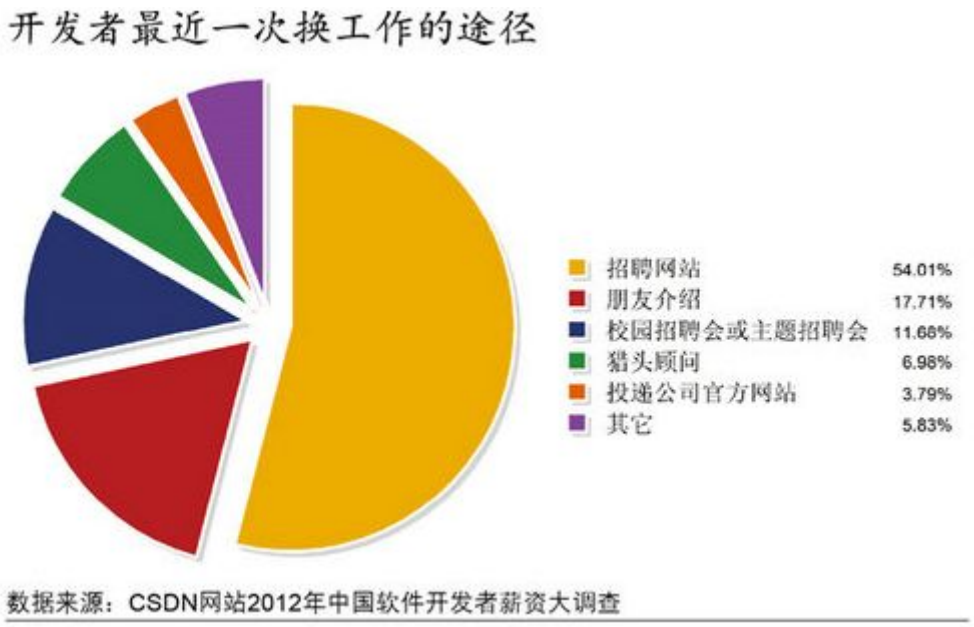


图 5-1：开发者换工作的途径

http://www.csdn.net/article/2013-04-16/2814909

这应该可以从侧面说明风评对一个人表达自己能力的巨大影响：同样能力的人，一个在社交网络中获得了较大的认可，一个处于封闭的环境中，那么无疑的前者会有更好的成长空间。

这也是一种借势并表达自己的方法。特别是当一个人年纪逐渐变大的时候，如果没有办法从社交网络中借势，那么被埋没的可能性就非常的大，除非你牛到了所有人都需要仰视的地步。而为了从社交网络中借势，博客、写书、参加各种线上线下的活动、参加定期活动的协会都是比较有效的手段。

当然这事上不适合急功近利，而适合逐步去做，否则就容易成为为求成名不择手段的怪人。种种行动的根本目的还是为了比较顺畅的把自己的实力展示出来让更多人知道，而非是营造原本就不存在的东西，所以最终能否取得关注和自身实力还是有本质关联的。

- 技术上的借势

2012 年的时候 Facebook 花 10 亿美元买下了照片共享应用服务商 Instagram 公司。这事在当时引起了非常大的轰动。这事之所以引起极大关注，其关键点在于后者只有 10 几个员工。10 几个员工支撑起 10 亿美元的价值，这绝对是让人震惊的事情。

做个简单的比较就可以知道这个差距有多大。

假设一个公司是做外包的，非常厉害，人月单价有 4 万 RMB。同时这个公司包括支持人员在内有 500 人，公司里每个人月都可以换成收入，那么这个公司一年的收入大概是 4000 万美元，而 Instagram 用 10 几个人达成 10 亿美元价值只用了差不多 2 年。两家公司中，单人一年创造价值的比率是 500 万倍。

关于 Instagram 如何成功想必有很多细节，但从公开报道来看，技术上的完美借势是很关键的一个原因。这从其开发团队所遵循的原则就可以看出来：

- Keep it very simple（极简主义）；
- Don’t re-invent the wheel （不重复发明轮子）；
- With proven and solid technologies when you can （能用就用靠谱的技术）；

两个技术水平差不多的人，一个信奉什么都从头造，一个信奉尽可能用现有靠得住的技术，那么无疑的从“表现力”的角度看，后者占优。

在各种平台和框架不断涌现出来的今天，善在技术上借势，那么确实可以用很少的人开创出比较大的事业。想象一下，架设网站可以用云平台而不用自己购入机器，分析大数据可以用 Hadoop 这样成熟框架而不用从头来写，只要专注在商业模式，就可以开创自己的事业，这相比以前什么都要自己做的年代确实是不知进步了多少。当然这样也有坏处，就是技术壁垒不可能太高，你能做他也能做，这反过来要求盈利模式、专利等上面要能架起壁垒来保护自己。

借势的一个关键前提

当一个人被鄙视的时候，常见的一个负评是：这个人格局不太行。

那么什么是格局？

老盯着公司有没有提供咖啡，就是没格局。考虑公司成长中自己可以扮演那种角色，并放弃某些眼前利益，就有点格局。

非要定义下的话，格局大致上就是心胸、气量、眼界、理想的一种复合体。借不借到势与当事人的格局有非常紧密的关联。

林则徐写过一幅对联叫：海纳百川有容乃大，壁立千仞无欲则刚。毛主席写了首词说：小小寰球，有几个苍蝇碰壁。嗡嗡叫，几声凄厉，几声抽泣。

这背后都可以看到格局的影子。

当一个人关注的概念（公司成败，行业兴衰）足够大时，往往他就会看到更多的东西，心胸也就随之开阔，格局变大。

当一个人关注的概念足够小（柴米油盐）时，无形中也就缩小了自己的世界，格局也就随着变小。

这里倒不是鼓励好高骛远，眼高手低，而只是说有的时候要改变既定的生命轨迹，往往需要突破现有的格局。

5.2.4 了解一点“政治”

武林里往往会有很多利益纠葛，常常是你想成为五岳剑派的盟主，我也想成为五岳剑派的盟主。一旦如此，那么就得各凭道行，做过一场一分高下。有利益纠葛就需要合纵连横，恒山派支持了华山派，华山派就实力大进；泰山派支持了嵩山派，嵩山派就实力大进。

有合纵连横就需要政治与权术，许你一个副盟主，许他某块地盘，对某个小帮派进行威胁利诱诸如此类的活动就纷纷出现了。但凡是江湖就必有政治，有政治就必有权术，程序员的江湖里也躲不过这游戏规则，只不过程度上有所差别罢了。从结果来看，这必将影响一个人体现在别人眼里的价值，所以这也是改善表达力的一种手段。

这点对于想在管理方向发展的人更有意义，对于技术方向发展的人则意义较弱。

1. 程序员躲不开“政治”

很多程序员可能会说，我一个做程序的，要多看代码，多写代码，政治有就有了，那有功夫去考虑它，躲开还不行么？

具体答案依赖于具体环境。但一般来讲，如果想往管理方向发展，那基本躲不开。如果想往技术方向发展，那躲开的可能性依赖于公司的传统，同时即使能基本躲开，想一点都不遇到也不可能。

只要程序员一天还需要和人打交道，那么程序员就无法完全脱离政治。消极的对应方法是让自己成为有技术的怪人或者不求上进的工程师，但这两者在职场里都会有不测之祸，不是什么好主意。

在人大版的《组织行为学》里给组织里的政治行为下了一个很精辟的定义：

政治行为定义为一些在组织正式角色中并不要求的活动，但它们会影响到或可能影响到组织内部的利益分配。

在对政治活动现实进行描述时，这本书里又写道：

政治活动是组织生活中的一个事实。忽视这一生活事实的人会把自己置于危险的境地。

在分析是否可能避免政治活动时，这本书上写道：

为什么政治活动一定要存在呢？难道作为组织不能避免政治行为吗？虽然这是可能的，但大多数情况下，不会如此。

组织由具有不同价值观、目标和兴趣的个体和群体组成，这就形成了对于资源的潜在冲突。部门预算、工作空间的分配、项目的责任以及薪资的调整等等，这些例子都表明组织成员可能会在资源分配上产生冲突。

组织中的资源又是有限的，这常常使潜在冲突转化为现实的冲突。如果资源充足，那么组织中的各个组成部分都可以实现他们的目标。当资源有限时，不可能满足所有人的利益。而且，不管正确与否，人们常常觉得一个个体或群体所获得的利益是以牺牲其他人利益为代价的。这种压力导致群体成员为了争取组织中有限的资源而展开激烈竞争。

导致组织内部产生政治行为的一个重要因素可能来自这样一种认识：大多数用来分配

有限资源的“事实”可以有各种解释。比如，什么是优良的绩效？什么是恰当的改进？什么因素构成了令人不满意的工作？某个人认为一种活动是“为了组织利益而进行的无私努力”，另一个人却会认为是“为了进一步获得个人利益而进行的积极尝试。”

长期在职场工作的人看了上面的文字往往会有会于心。

结合事实来看，在任何组织中，彻底避开政治行为确实很难。一般来讲在新兴行业，公司历史较短，生存时刻受到挑战的公司，政治氛围会偏淡；而在传统行业，公司历史较长，生存没有问题或者生存依赖于特定组织的公司里，政治氛围会重。

即使现实如此，纯粹的工程师可能仍然会厌恶这东西。其实这也不是不可以，要想尽可能的忽略组织政治因素，那么你必须在某一方面一骑绝尘，让稀缺性达到一种可怕的程度。好比说搜索引擎有你在，不管多少并发都可以达到在 0.1 秒内返回结果，你不在就不行。但这很难，不只是要求个人的才情，还要求特定的环境。如果是去种地，那么不管一个人体力多好，他也达不到一骑绝尘的地步。

也许仍然有程序员对此心存疑虑，说你言过其实了，IT 这种新兴行业，那会如此。很难去直接证明 IT 行业（包括新生的互联网）他就是如此，但可以从日常某些新闻报道中进行管窥。下面是一篇互联网上公开的报道，题目是：为什么大公司的钱不能随便要？，作者是 21 世纪经济报道资深记者，移动互联网观察者，著有《一只 iPhone 的全球之旅》一书的曾航。不知道每个人从其中可以看到多少公司与公司、公司间部门与部门间的政治的气息。又由于事情比较繁杂，这里只做一点转载，但不做过多解释，但请仔细阅读标记出来的部分：

为什么大公司的钱不能随便要？

当腾讯、360、百度、阿里巴巴、盛大这些大公司的投资部门，拿着钱想投资你的创业公司的时候，你是应该高兴呢，还是应该警惕呢？

我想大部分创业者在得到这个消息的时候会满怀憧憬，高兴的晚上觉都睡不着。生逢乱世，有个有实力的大哥说要罩着你，以后的道路，立马变的平坦起来，谁不高兴呢？

接受了腾讯投资的人在憧憬着，自己的产品能够率先接入微信，坐拥 3 亿用户的巨大流量；接受了百度投资的人在憧憬着，百度随便倒一点流量过来，自己也吃喝不愁了；接受了阿里巴巴投资的人在憧憬着，自己能够早日用上淘宝那非常值钱的数据资源……

不过，我们从近期的一系列教训来看，拿巨头的钱，要格外谨慎。用一位资深投资人的话来形容，就是你叫他一声干爹，他不一定答应，但是却给你添了一大堆敌人。

首先我们要明白一个道理，投资你的多半是大公司的投资部门，腾讯有腾讯的投资部，360 有 360 的投资部。投资部门，往往是指挥不动业务部门的。而你想要对接的巨头资源，往往掌握在业务部门手上。

一些互联网巨头内部的复杂程度，远超过人们的想象。举个例子，腾讯负责游戏的任宇昕，去找分管微信的张小龙办点事情，都不一定能办得成的。亲儿子尚且如此，一个外面投资来，没有控股的干儿子，想要对接大公司资源，又谈何容易？

你叫巨头一声干爹，他不答应，可巨头的竞争对手们，却在你拿了投资后把你当成了它的干儿子，和你老死不相往来。

举个例子，你要了 360 的投资，就等于得罪了腾讯、百度、雷军系，互联网的半壁江山，都不会再和你合作了。

某刷机公司不久前接受了 360 的投资，想去找小米、百度合作，结果被人直接拒

绝，使得这家公司丧失了诸多发展的宝贵时机。

最糟糕的情况不止于此，如果有的巨头投资了你，还对你指手画脚，那就更加糟糕了。

当时盛大在力推盛大账号的开放平台，就想让盛大投资的一些公司率先接入做示范，盛大投资的在线电影票公司格瓦拉花了半年时间做系统对接，结果第一个月通过盛大的账户只卖出 2 张电影票——这说明盛大的内部业务整合对于其所投资的创业公司往往并没有太大的帮助，此事一度在圈内引为笑谈。

不过，对于许多独立的创业公司来说，卖给巨头是他们的唯一选择。因为一些创业方向，纯粹依靠自己的力量做大太难了，必须要对接巨头的流量、用户。

... ...(后文略）

这里面的规律可以简单描述为一旦一个公司发展起来后，内部必然伴随着某种势力格局，这种势力格局就会衍生出政治。只有在公司初创，高速成长期，这类氛围才可能比较淡薄。

组织行为学中的定义比较精确，但有点抽象，这里可以对政治这事做一个更简单的说明：

马化腾先生可以 30 岁就执掌企鹅帝国，因为他创建了这家公司，并伴随其成长，所以他是 CEO 是很自然的事情。我们假设企鹅帝国一直存在，那么终究要面对谁是下一个掌门这个问题？掌门只有一个，品格、能力、贡献、资历都够的人却一定有好几个，那么究竟是 A，是 B，还是 C？这种问题的存在始终是政治无法被彻底避免的一个根源。真想完全靠能力胜出，最好是选择在高速扩张，处在新兴领域的行业。道理很简单，打仗的时候首先考量的是一个将军是不是能征善战，和平时期则不只要考虑一个人是不是能征善战。

2. 可参考的“政治”手段

为了面对组织政治中的种种纷扰，至少要了解其基本的表现方法、权术手段。

你可以不用，但不知道则是对自己不负责任的。就我个人感觉程序员大多不喜欢这个，但这确实是必要的，尤其是对管理方向上的人，虽然这并非根本。

- 印象管理

印象管理讲的是如何控制自己的表现面。一个人说谎来夸大自己的优点无疑是错误的，但这不意味着一个人要积极的表现自己的缺点，也不意味着一个人不需要宣传自己。

当进行这类印象管理时往往需要一定程度的自我监控。比如一个人可能天生性情比较急躁，这时如果他的上司或同事比较满斯条理，他必然比较倾向于打断上司或同事的发言，自己迅速总结出结论，但这样做基本上没有任何好处。这个时候如果没有一定的自我监控，那么很容易会变成讨人厌的人，最终导致很容易被负面解读，事业心强可能就会被解读成有野心，展示自我可能就会变成班门弄斧。

再比如人可以适当的推销自己，但如果缺乏自我监控，那么就很容易超过一定限度而试图通过贬低别人达到目的，这就容易有负面效果。一个人说：通过我这一年的努力，Bug 率下降了 5%，这是推销自己。而如果他说：通过我这一年的努力，Bug 率下降了 5%，而李四做同样的工作时则 Bug 率上升了 4%，这就是贬低别人。

落到具体工作环境下，这类场景很多，需要用心考虑的事情也很多，但总的来看原则却并不复杂：诚实的宣传自己，更多的获得认可。

这样的行为之所以有意义在于，在相当多的地方优秀本身是一种相对值。如果两个人之间的差别是数量级上，比如 1 厘米和 1 米，那么印象管理是没有用的。但如果这种差异是 1 厘米和 1.1 厘米，那么决定谁更优秀的往往就不是绝对值而是印象管理。

- 常见的权术手段

权术手段这东西似乎是能理解的就理解了，不理解的就很难解释。因此，在这里只是罗列一下组织行为学中总结出的各种权术（注：下文的权术项目及描述来自人大版《组织行为学》一书，但说明则是本书补充的），以及不愿意了解组织政治的人常犯的错误，并不会对此进行过多的解释：

- 合法性。强调自己的请求与组织的政策规则一致。也即是说先让自己占据“名分”，占据道义的制高点。不理解这点的人，往往会把任何请求都变为个人请求，而个人请求往往师出无名，导致冲突。

- 理性说服。提出逻辑论据和事实依据来证明请求的合理性。这点借用的是逻辑的力量，潜在前提是“人是讲道理”的。不理解这点的人，往往会把事情变成 0 或 1 的游戏：要么同意，要么拒绝。其实这有中间态，即游说后同意。

- 鼓舞式诉求。通过所选人物的价值观、需求等来开发情绪承诺。陈胜吴广造反时说：王侯将相，宁有种乎？ 伐无道，诛暴秦。就是这种。这类事情不能不做，也不能长做。不做的话，团队容易死气沉沉，上班干活，下班走人，毫无生气。做多了，容易被认为是专职画大饼的。

- 商议。通过让他人参与决策如何执行计划和变革来提高对目标的激励，获得更多的支持。比如说探讨开发日程时如果全员参与了，那么即使估算和实际有偏差，团队也会愿意努力保持自己的承诺。与之相对的则是独裁，这会把自己放到和整个团队对立的地步上，大多时候不可取，应该只保留用来维护核心价值。

- 交换。通过奖励目标人物一定的利益或好处来交换接下来的请求。比如说：去阿富汗工作可能比较危险，那么就需要预先承诺工作三年后回来可以得到什么。

- 个人式诉求。使用友谊或忠诚获得同意。当一个人日常多为人提供帮助时，那么一旦他需要帮忙时，道义上受过他援助的人很难拒绝。

- 逢迎。提出请求前，先吹捧、赞扬或使用友好的行为。这就是老子说的：预先取之，必先予之。当一个国家被评选为负责任的大国时，这个国家做事的时候往往就会先考虑是不是对得起这个称号。

- 施压。使用警告和威胁，反复重复你的要求。这个某些领导常用，如：如果你不能搞定这个事情，那么就不要回来见我了。

- 联盟。寻找他人帮助说服目标人物或利用别人的支持作为他人同意你目标的理由。这个在山头林立的组织常见，三人成虎也可以成为一种手段。

在我个人看来，过分钻研权术手段的使用有点舍本逐末，但忽视他们的作用则容易造成人与人之间尖锐对立，没有回旋余地，因此还是多少知道一点比较好。

为避免矫枉过正，还是要补充一点，程序员这个群体里厚黑是不行的，程序上没什么秘密，一看就知，藏不住过黑的东西。

从代码里你可以看到政治么？

纯粹的程序员从代码里只看到技术，所以大多时候会抱怨：谁写了这么垃圾的代码？但懂点微观经济学的程序员则会在技术之外从代码中看到利益纠葛，看到人心世道。

为什么世界上会有这么多垃圾代码，这绝对不只是因为技术不行。如果世界只由技术因素主宰，那么按理说只要一个软件存在的时间足够长，投入的人力足够多，代码一定会变的足够好。但事实恰恰与这相反，存在时间越长的代码往往越垃圾。

这可以简单理解，既存的代码表征着一种市场价值，如果改了它那么一旦造成的损失谁来负责？程序员来负责？总经理来负责？

没人来负责，那么只能破坏逻辑清晰性来保证妥当性，代码自然就会变得越来越垃圾。所以说这里首先是利益纠葛问题。

这是非常有意思的一个课题，因为改好代码长期有收益，短期必然有风险---再牛的人也没办法保证自己的修改毫无偏差。宏观来看，保证好代码真的很简单：找一帮有责任心的很牛的人，让他们不考虑市场因素的持续进行重构，那代码必然越变越好。而关键则是，如果你是CEO，你愿意这么干么？

所以说在市场经济里面，往往并没有纯粹的技术。这样一来，政治的存在也就不是什么值得奇怪的事情了。当然开源很可能是例外的。

§5.3 检查自己的表达力

讲了这么多关于表达力，也许有人很好奇自己的表达力到底怎么样。如果想简单快速检验自己的表达力，可以用这样一个方法：

假设有一个你很想做的项目要开始了，你的技术能力是足够的，你想在这个项目里承担比当前责任范围更大的工作，这时候在有竞争者的前提下你有多大把握获得参与这个项目的机会？这个时候如果答案是绝对没问题，那么表达力上大致是没什么问题的。如果是没可能，那么表达力上是很有问题的，需要做点反省，看看到底是上述那个环节出了问题。

如果想系统的做个检查，那就要按照上述所说逐项进行检查：

- 自己换工作的频度是不是太高了？

  既然从资历的角度看，换工作次数不宜过多，那么一生换几次工作合适？

  单纯从功利的角度来考虑，那大概是 2~3 次。注意这里不是说一定要 2~3 次，而是说最多这样。如果人生顺利，最好跳的次数越少越好。

  不管怎么说毕业 2~3 年左右要考虑一次，因为毕业的时候终究是对这个行业了解的并不多，这个时候累积了一些技能，可能需要重现审视一下自己的人生方向。再选的时候要尽可能选值得自己长期工作的公司。或者值得信赖的公司或者创业题目十分吸引自己的公司都可以。这个时候要尽可能忽视短期利益驱动，比如单纯因为增加了几百块钱而换个工作。

  接下来的一次很可能在毕业后 5~10 年间，这时候也许做到了比较关键的职位（中层左右），但感觉公司所提供的平台无法让自己彻底的施展，为了谋求更大的空间，可能需要换工作。

  当然在此之前为了提升到中层左右的职位，有可能也需要换一次工作。上面并没有考虑为了实现人生价值而重新寻找适合自己的公司这类情形。

- 自己是不是个惹人厌烦的人？

  这点上要反省有没有什么事情自己也觉得非常不好，但却在自己的工作中重复出现？有没有多个不同的人都用类似的言辞抱怨自己？真心问问自己有没有糊弄手里的工作？

- 自己有没有一点影响力？

  会有人找自己解决些有困难的问题么？同事之外还有人知道自己么？自己知道在行业里谁在做和自己类似的工作么？又知道别人在怎么做和自己类似的工作么？

上面所列的这些检查项目，任何一个上有问题都会导致表达力有问题，可以以其为契机进行一定的反省。

§5.4 小结

表达力和价值这两者进行组合可以出三条人生道路：纯表达力道路、纯提升自我价值的道路、表达力+提升价值的道路。我们很难说那条道路一定不可行，马云先生据说是不懂技术的，但一样有很大的成就。

不同道路间的优缺点却是比较清楚的。纯表达力道路容易找不到起点及实现自己的机会。除非你自己白手起家，毕竟很少有 IT 公司会招不太懂程序的程序员。纯技术道路容易被埋没，并达不到高处。纯技术道路如果达不到一定高度，那就容易有付出却不被关注，除非你达到没你不行的高度。大多时候，两者应该是需要混合着来的，程序员要补的是自己的短板。当技术道路碰到天花板，那就回头要看看表达力；当表达力足够，那就要回头看看技术根基。

至于怎么判定自己是缺表达力，还是缺技术力，则并不是很难的事情。大多数人只要在夜深人静的时候仔细想想过往，大致就可以找出答案。

如果答案仍然模糊，那么可以做张表，给自己的技术力在公司里排排位置，如果已经排的很靠前，并且掌握前面所说的知识地图中大部分知识，但职业路径却不顺畅，那基本上是缺表达力了。这时候需要参照前面章节，寻找自己的短板，并开始改善自己。

这里面很可怕的一件事情是，技术因为天分、机会的限制没达到特定的高度（比如：能一个人搞定百万并发的网站），但却自我感觉良好，认准了自己是程序员中稀缺的、比较靠前的 5%，而不注重表达力的改善。这时候选择的道路和现实的需要间就会有矛盾，会影响人生高度。

最后想强调的是，在这一章里，我们对表达力进行了重定义，把它定义为资历、性格、借势、政治应对综合后所表现出来的一种力量，而不是单纯的言语表达。

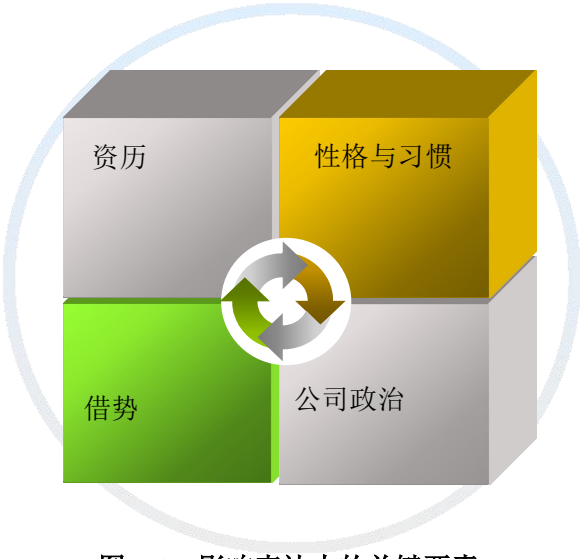


图 5-1：影响表达力的关键要素

第六章 程序员的稀缺性营造

假设说你想在江湖里谋求一定的地位，那么你可以练习独孤九剑成为超一流高手，也可以练习医术，成为绝世神医。这两者在江湖里都是有地位的，也都是稀缺的，一者是因为杀伤力，二者是因为人都有山高水长。

程序员也一样，增值也好，改善表达力也好，最终都要在某种环境下达成一定的稀缺性，这样一个人才有价值。稀缺性同时受两个维度上的力量影响：一个是自身的努力，比如前文所提到的增值和表达力；一个是大环境的变化以及对这种变化的适应。在这一章里主要关注的是后者。

§6.1 稀缺性可带给你什么

既然稀缺性对个人有如此大的影响，那稀缺性到底可以带给一个人什么样的影响，我们来看一个简单的例子：

在日本曾经有这样一个故事。一个人在某电信公司负责一个大型系统的维护，收入虽然不菲，但时间一长，这个人就对薪资发展不太满意，因此最终选择了离开。结果他一离开，这大型系统立时跑的磕磕绊绊，无奈之下，这家电信公司只得以高职厚薪把这个人请了回来。可以想见为了达到这一目的，这家电信公司，无论在收入还是职位上必然都开出了让这个人无法拒绝的条件。

这是稀缺性起作用的一个典型例子。大型系统因为关联到庞大的用户群体而必须要用，同时这一系统的维护没有这个人又不行，这就使这个人的稀缺性变得非常突出。

这事其实很有意思，因为在这里事实上是不好的软件成就了一个人的价值和稀缺性。这虽然不是很好，但其实这类情形并不罕见。从市场的角度来看，它并不关注一个程序的内部逻辑是否清晰，是否有足够的注释，它只关注这东西能不能运作好。所以使用中的垃圾代码一样有巨大的价值，也就是说商业上的考量对稀缺性的影响更大。

为防止上述文字被曲解，这里补充一点说明。上述道路并非是一条非常值得模仿的道路。因为对上述那个人而言，事实上他的价值绑定于特定的一套系统，这会导致可流动性几乎没有，这就会限制住一个人的成就，并使未来存在很大风险。

§6.2 改善稀缺性的途径

为了改善自己的稀缺性，通常需要同时做两个方面的工作：一是提升自己；一是顺应时势。提升自己可以让自己稀缺这点很好理解，但如果没有顺应时势相配合，就很容易让这种稀缺性无法很好的实现。在 2013 年精通 DOS 编程的人无疑是稀缺的，可这不一定能产生价值。下面我们将从上述两个方面对稀缺性做一点说明。

6.2.1 奔向程序世界里的价值高地

投资大师巴菲特先生说过一句流传很广的话：有的企业有高耸的护城河，河里头还有

凶猛的鳄鱼、海盗与鲨鱼守护着，这才是你应该投资的企业。这句话非常传神的描述了价值高地的外在形象。

对于企业而言，护城河可以是很多东西：高难的技术（波音飞机）、难以攻破的用户粘度（QQ）、独占的资源（中石油）、独特的企业文化（苹果）等等。

护城河使企业拥有一种无可取代的价值，从供给上看这就是营造企业自身价值的稀缺性：缺了它不行，你又没有更多选择。这就是价值高地，当企业在这上面时，他相对安全。也正因此，大公司最终都会试图主导一种秩序与生态系统，只有如此大公司才能掌控稀缺性。

这道理同样适用于个人。稀缺本身可以有很多来源，可以来源于时机，也可以来源于高度。来源于时机的稀缺性更像一种偶然，很容易被打破，往往并不具备长久的价值，相对于人的一生而言，这并非是一种有力支撑。比如：Erlang 可能比较稀少，但单纯的语言壁垒并没有想的那么高，如果真的有巨大需求，这个世界上可以在一个月间多出几百万 Erlang 程序员。

当一个人经营自己的稀缺性时，确实要找到一个有鳄鱼、海盗和鲨鱼守护的地方，这才是价值高地。当然鳄鱼之类很难是你放的，这与企业不同。在这点上管理方向上和技术方向上的程序员所面临的选择和所需要采取的措施不同。

对于技术方向上的程序员而言，走向上述这类价值高地本身可以有两种方法：

一是达到一定高度横向展开。比如：编程语言，（金融）业务逻辑，外语，网络知识等组合在一起就可以成为一个高地，这里面编程语言上一个人可能不如天才程序员，业务逻辑上可能不如银行员工，外语可能不如专职翻译，但每多一重过滤，就会导致高地的海拔拔高一分，最终转换为稀缺性。

一是彻底的专家型道路。有的岗位可能不需要把面扩的很宽，比如做 TTS，OCR 的算法，有些人甚至编程语言都可能不是了解的很熟，但确实可以是某一方面的专家。这同样是一种价值高地。在这个方向上，一旦真的达到一定高度，那就不是单纯的累积数量可以超越的。比如：认为 100 个或多少个平庸的科学家等价于一个爱因斯坦无疑的是愚蠢的。

不管是那种方向，最终都要达成这样一种效果：你可以完整的搞定一件很有商业价值的事情，而这件事情大多数人搞不定。比如说：

- 我可以主导开发一款手机，因为我即懂软件又懂硬件，也还知道如果开发一款良好的产品。
- 我可以把 OCR 的识别率提高 1%。
- 我可以主导架起百万级并发的网站。
- 我可以带领队伍搞定这个银行的整个系统。

... ...

这个时候最好不要用单纯的技术观点来衡量自己，比如我擅长 Java，我会用 PHP，我知道 TCP/IP 协议等等。不是说这没有价值，而是说这种视角有点低端。只有能完整搞定一件事情才会与商业利益直接挂钩，才可能有真正的稀缺性。

对于管理方向上的程序员，走向上述这类价值高地似乎只有一种途径：

要努力做出让人记得住的成绩，这个成绩可以是一个产品，也可以是某种业绩。今时今日，提到微信相信大家都会想到张小龙。这是因为微信本身在不到两年的时间里吸引了 2 亿用户，并且口碑很好，实在是个奇迹。

关于价值高地，有一个典型的陷阱：不含复杂度的，特属于某个公司的经验，往往让人误以为是价值高地，但其实不是，因为只要环境相对的公开，这类东西往往可以在短时间内被攻破。比如：一个公司可能定义了自己的流程，其中很多东西较为模糊，新人一做就处处碰壁。这很容易让然误解为掌握流程本身有较高的价值，但其实这是由于流程不完善所造成的，是特定场景下的一种偶然。这确实导致稀缺性，但基本不具备可流动性，大多时候未必是好的选择。

需求开发算价值高地么？

在偏敏捷的组织里程序员往往离需求很近，但在比较传统的开发方法中，做需求的和程序员往往是有段距离的。做需求开发的可能不太会写程序，写程序的不太会写需求。

那需求开发算价值高地么？

很多纯粹的程序员可能觉得单纯的文档工作没什么技术含量，似乎谁都能写，因此可能认为这算不上什么价值高地。但从商业价值来看，当一个人摸透某个行业的业务（懂技术更好），那么这还真是价值高地。

这可以来做个类比，天猫只做平台，各个商家卖东西，那么天猫有价值么？当然有价值，天猫 11/11 的销售额 100 多亿比美国的黑色星期五还高，怎么可能没有价值。

那为什么天猫有价值？因为终端客户的眼里是先有天猫，再有各个商家，天猫垄断了入口，所以天猫更有价值。

需求与开发的关系与此类似。当一个人做某个产品的需求时，在外人的眼里，这个人做的需求才表征着这个产品，透过产品才能看到程序员的贡献。外部人员思考的思路是先需求开发人员再程序员。

其中比较极端的一种实践是需求开发人员主导整个项目，所有其他人员在需求开发人员的领导下工作。

这个时候钻牛角尖是没意义的，比如：有的人可能认为没程序员那有产品，这就和争论没店家那来天猫一样，毫无意义。在现实中当然两者都有存在价值，这里讨论的只是说这是否算是一块价值高地。

6.2.2 走在技术大潮的前面或里面

IT 世界里，城头变幻大王旗来的特别的快，而每一次变幻时事实上都将导致某种技术的兴起或者某种技术的衰落。

当年 WPS97 的开发时间非常长，对此百度百科上对此的描述是：Windows 有很多新东西，我们还没有熟悉过来，微软又升级了。很多技术资料，也很难找到。微软掌握着 Windows，而我们什么都要靠自己从头做起，这导致了 WPS97 难产。如果 WPS97 能在 1995 年推出，直接和 Word6．0 竞争，Word6．0 肯定没戏。

这很生动的记述了一门新技术兴起时所造成的稀缺性，从侧面也可看出来，在 95 年的时候企业对高端 Windows 开发人员是何等的渴望。这种稀缺性是行业周期背后的技术更迭所造成的。而在今天，借助搜索引擎，初入行的程序员也可以解决大部分 Windows 编程的问题。

面对这种技术潮流，比较合适的办法是基于现实勇敢拥抱新技术。

基于现实是指考虑技能的可流动性，考虑实践和学习的不可以分离特质，选择自己认为前景好的新技术，并投入时间。但这里面有个陷阱，一提到新技术很多人可能会联想到新编程语言，但编程语言太基础了，壁垒太低，并不是一个足够大的考量区域。视角如果限在这个尺度上，看到的东西就会太多，而不容易聚焦，这时候需要把自己考量的单位适当放大一点，英文中常用 Tech Stack 这个词来描述这一组技术。

比如说：LAMP(Linux+Apache +MySQL+Perl/PHP/Python)可以是一种考量单位，Windows 编程+ASP.NET 也可以是一种考量单位，大数据处理相关种种也可以是一种考量单位。

如果回望十年，我们就会发现，先有 PC 客户端程序的鼎盛，接下来是互联网的兴起，再接下来则是移动客户端的兴旺。以当下而论，无疑的移动客户端和互联网要比传统的 PC

客户端来的更有吸引力。而在云的时代里，壁垒比较分明的两套 Tech Stack 则是基于闭源的一系列技术（主要是由微软提供）和基于开源的一系列技术。在这里面如果那个 Tech Stack 的技术逐渐取得优势，那么无疑的在相应的 Tech Stack 中有积累的人会有比较好的稀缺性。

虽然眼下看来，两者似乎没有明显差别，但在这点上，我个人认为未来开源 Tech Stack 会逐渐取得优势。在 Quora(quora.com)和 High Scalability(highscalability.com)上，我们可以查找到国外大部分新兴的、市值超过 10 亿美元 Web2.0 网站的技术架构，如：Flickr，Pinterest，Instagram 等。如果用心来读这些技术架构，就会发现他们一个根本的共同点：他们都是基于开源技术构建的。

这种不约而同的选择背后有一定的必然性。当希望一定的定制性并且不愿意支付高额成本时开源 Tech Stack 几乎是一种唯一的选择，尤其是当开源的技术有越来越多成功实例的时候，这种优势就越来越明显。

如果非要在客户端（iOS，Android，WinRT)和互联网中选择，我个人认为互联网比客户端更有优势。

技术落潮所伴随的风险

很多人会讲微软在 2002 到 2012 这 10 年里几乎无所作为，也会谈论从股票上来看如果 10 年前买入的是微软股票那么现在只能赚 30~40%，而如果是买的苹果股票那就要赚 3 倍多。

我个人偶尔思维发散，想到的却不只是这个，而是如果微软再失去 10 年，那挂掉的不只是微软，还有同微软绑在一起的各种公司和个人，包括很多资深的 Windows 程序员。

在 PC 的世界里微软是无疑的霸主，但如果 PC 的时代过去了，那么这个霸主如果无法转型成功，那么无疑也要随之殉葬。而那个时候无数在微软平台上花了半生心血的人却还都在，他们又该何去何从？

技术大潮的兴起会使潮头的很多人称为耀眼的明星，而某波潮水的退去，同样会带走与之相伴的一些人的光环。所不同的是前者轰轰烈烈，而后者寂寂无声。

在这种情境下，还真就只能与时俱进。

§6.3 检查自己的稀缺性

从社会需要的角度检查自己的稀缺性非常困难，因为相关的各种数据总是非常缺乏。但有个简单的方法可以很快的让一个人认清自己的稀缺性：假设一个毕业生很努力的学，那么多久他可以取代你的工作？比如一个毕业生只要努力，那么可以在一两年取代你，而你的年纪已经接近 30 岁，那么稀缺性必然非常不好。

而与这个相反，如果一个毕业生即使很努力，也要五年才有你的技术水平，同时如果没有特定的机缘，怎么也无法取代你，那么即使你已经 30 岁，你的稀缺性也会非常好。这里的机缘可以是指某些特别的实践机会。

如果想比较系统的评估自己的稀缺性，那么需要依次考虑如下问题：

- 自己所掌握的技术是即将过时的技术么？

  技术大潮总是会定时的淘汰各种技术，不同的时间点淘汰的对象也不太相同。有的虽然不是完全淘汰，但至少他们不再像当年那么辉煌了，如果以 2013 为界限而回看 10 年，那这样的技术有：Flash，MFC，Delphi 等。

  为保持对技术动向的敏感度，定期阅读别人的架构非常关键。

  当然可能过时的技术不单指通用的技术，还指老旧的可能会为新解决方案所替代的系统。比如说：曾经很多公司使用 Lotus Notes 来做知识管理的，但很少人使用这样的系统了。

- 自己所掌握的技能究竟有多少人会？

  考察这点时要像前文所描述的，更多的从公司的视角去考虑，而不是个人的视角。单纯的会使用某个语言或者框架这种程度，稀缺性一定没有。比如：单纯的会用 ASP.net 开发网页几乎没有较高的技术壁垒，但对数据库的设计有相当程度的掌握、能够较好的通过负载均衡、缓存等手段保证系统的性能就可以使自己的稀缺性上个台阶。

§6.4 小结

从技术角度看，稀缺性就是选定一个技术路径长的，不处在衰落期的领域，不停的打磨，不停的前行。

软件相关的知识实在是太庞杂，没法设计一条唯一的道路，但稀缺性好不好却可以大致评定：考虑一下一个新毕业生达到自己这个高度，需要多久。如果和新毕业生三年后所能达到的高度相似，那就怎么也不是稀缺性好的位置。

至于大环境里，做与自己类似工作的人是多是少这点，则只能靠运气了。

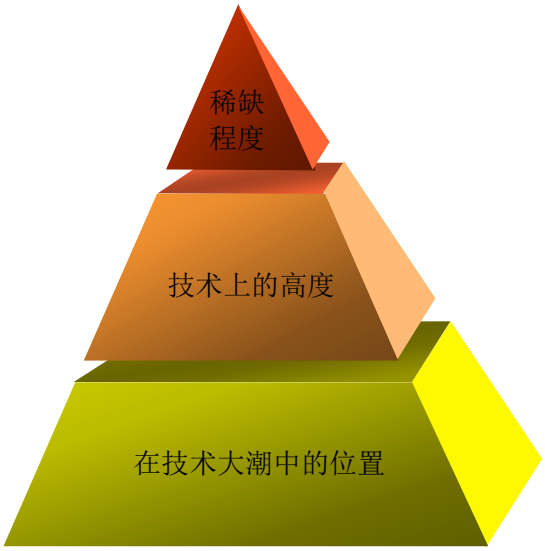


图 6-1：稀缺性的关联要素

第七章 程序员的公司选择

前面几章里面讲到了自身价值、自身价值上的表达力和稀缺性，这三项更多的讲的的是个人，在职场中无疑的与个人直接关联的是公司。这一章将具体说明与公司相关的、影响个人发展的要素。

在武侠的世界里，帮派本身借助了个人的力量而成其威名，但反过来个人却又因为帮派的力量而被烘托的更高。如果我们把公司类比为帮派，把程序员类比为一众江湖人士，那么帮派和成员间这种异化、同化的过程就几乎在每一个程序员的身上都可以看到。

牛顿说过一句广为流传的话：如果说我比别人看得更远些，那是因为我站在了巨人的肩上。选择在什么样的公司里工作，几乎等价于选择站在什么人的肩膀上。再形象点讲就是选错了公司等价于输在起跑线上。

公司提供的岗位决定了你所接触的技术、人物，进一步决定了你的眼界、能力和人脉，也一定程度上决定了你履历的价值。这些都是对一个人至关重要的东西。

假设说一个人在大学里非常的优秀，以较高的薪资加入了一家做生僻技术的公司，5年后，这家公司倒闭了，这时候这个人可能会陷入困境，甚至变的不如很多大学里成绩比他差很多的人。这类事情往往并不是因为个人不努力，往往是因为选择不慎而把自己给憋到死角里了。

每个人要想争得选择权往往并不容易，大学四年的努力挣来的选择权却可能在一时不慎下毁于一旦。这种选择往往很难很精确，但有几个基本项目还是可以考察的。这些项目将在下面各节里逐步展开。

§7.1 给公司分类

公司是一个含混的词，要想弄清楚公司对人生的影响，那么先要给公司分类。一说到分类大多数人可能首先会想到好公司或者坏公司，但其实这是分类的结果，在好和坏背后往往都有着一打的因素来支撑一个公司是好还是坏。不了解这些背后的因素，单纯根据一个笼统的好或坏实际上是很难判定一个公司是不是适合自己的。

7.1.1 分类的方法

一个人要想做对选择，一要有选择权，二则要了解待选项。在这里待选项就是公司的类别。可以有非常多的视角对公司进行分类，这里主要列举和职业发展紧密相关的几个。

- 分工所处位置的视角

帮派间是有食物链的，比如东方不败的魔教就凌驾在黄河帮之上。还不用东方不败或任我行这种大头目出来，只是盈盈就可以让这些帮派鞍前马后了。

类似的关系也存在于公司与公司之间，但这时这种关系则是有产业分工时公司所处的位置所决定的。即使我们不用心去做非常详细的调查，也可以看到 iPhone 生产背后的三层结构：苹果 + 富士康 + 元器件供应商。

这种链条的一个根本特征是上游的企业人少但分享的利润更高，当然其承担的创新风险也更大。

在这里无意对这种分工模式是否合理进行更多的探讨，想强调的则是，这是影响个人工作和发展的一个很重要的维度，不考察是对自己不负责任的。

商业社会中的公司所处的位置可以做简单理解：筹码多的，控制权、发言权大的在上端。比如：付钱的在上端，靠别人给钱的在下端。垄断的或接近于垄断的在上端，被垄断的在下端。比如说：公司 A 把业务分包了给公司 B，那么公司 A 在食物链的上端，公司 B 则在下端。比如说：公司 G 直接面向客户，占有了市场份额 60%，其他公司分享其他的 40%，那么 G 在上端，其他公司在下端。在这个视角下可以把公司分为发包的公司，接包的公司。从分工的角度看，一个公司所处的位置越靠后，其利润空间可能也就越微薄。

与这类分工相关的场景还有很多，比如：母公司与子公司、发包的和接包的、做操作系统的和必配指定操作系统的硬件厂商、做电商平台的和各个产品制造商等等。

从选择公司的角度，应该尽可能考虑上游的公司，因为这种分工上的位置对成长空间和收入会有比较大的影响。

- 行业的视角

在这个视角下，公司往往被分为电信、金融、互联网、ERP、游戏、嵌入式等。很难讲那个行业相对另一个行业具有绝对优势，但有两点需要额外注意：

一是软件是不是公司的主营业务对发展空间往往会形成一定限制。制造业的工厂里也做软件，但不太可能选几个做软件的去做厂长，金融公司里程序员也很难发展成为总裁。

另一点是有的子行业处在成熟期，发展相对平缓，而有的子行业则处在告诉发展期，发展很迅速。后者机会更多，年轻人尤其是容易在全新的领域出头。这是因为在全新的领域中没有既定的利益格局，所有人都在同一起跑线上。

从选择公司的角度看，无疑的要尽可能选选主营是软件并在高速发展子行业的公司。而之所以选择行业非常重要，根本原因是一旦选择了那个行业，那很可能一辈子就会干这个了。做金融的很难去做互联网，做互联网的也很难去做嵌入式。

- 时间轴的视角

从生命周期来看，公司可以分为发展期的公司，成熟期的公司和衰落期的公司。

从选择公司的角度看，极力要避免的是衰落期的公司。识别衰落期的公司简单的办法是财务数据和市场地位。已经赤字的或市场地位已经排位靠后的都是相对比较危险的公司。

而选择发展期的公司还是选择成熟期的公司上，要依赖于个人选择。通常来讲选择成熟度较高，市场地位较高的大公司会比较好，便于做技术积累。

- 核心竞争力的视角

有的公司是核心竞争力偏向于销售渠道，有的则是技术，有的则是两者并重。

比如对于信息管理系统销售渠道就比较重要。从这个视角看，有核心竞争力的公司总是比没有的更好。而在有核心竞争力的公司里，技术导向的公司无疑更适合程序员。前面曾经提到过，非技术导向的公司里，技术路径通常会短。比如说：只做信息管理系统的公司技术路径通常不会长。

- 地域的视角

从这个视角可以把公司分为一线城市的公司，二线城市的公司。地域主要影响可流动范围。

在 2012 年曾经发生了一起很有点影响的事情：摩托罗拉移动因为经营的原因，关闭南京摩托罗拉移动，裁掉了整个团队，这个时候南摩员工联合起来，一起拒绝在离职协议上签

字。员工和公司前后经历多轮谈判，员工方坚持不让步，其中的一个关键理由就是：500 多人一起在南京找工作非常困难。与此相对应的则是，同样是裁员，但北京方面所有员工皆较快签字走人。

这一事请本身的是非很难去评论，但南京和北京员工的不同反应却可以一定程度上折射出城市（或者说地域)所带来的影响。

南京已经是非常发达的城市，但 500 人的队伍已经会担心很难被吸收掉，在北京恐怕这 500 人就掀不起什么波澜。这很好理解，一升水倒在矿泉水瓶子里，会溢出来，但如果倒在小河里，就几乎等价于不存在。但这种城市对人员的容量对可流动性、对发展空间却会形成一种刚性的制约。

当一个人选择了某个公司的城市，也就选择了其背后隐含的一切，对于大多数人而言与城市的关联度将会随着时间的流逝而越来越高（这体现在学校、房子等方面），所以适当对此进行考察是有必要的。

总的来看越是期望快速发展，越是希望开创一番事业，对自己越是有信心，越适合到一线城市发展；而相应的越是希望安稳，越是侧重生活和工作的平衡，二线城市就更适合一点。

- 公司文化的视角

如果去翻阅管理相关的书籍比如《基业长青》，《赢》等，你就会发现公司的文化被放在一个非常高的位置上。极度优秀并可存续的组织无疑是需要一种共同的价值取向来镇压气运的，但这和现实其实有点远。现实中的很多公司更可能是规则和个人双重驱动，而没有什么明显的文化特征。所以这种分类视角被放在了最后。

很难讲那种企业文化一定会成功，但如果从负面来解读文化的话，有一种文化却几乎一定会使公司挂掉。这种文化可以叫做蠹虫文化。典型特征是：一盘散沙，所有人都一起糊弄工作，工作上的事情能推就推。这类公司几乎一定会比较快的走向毁灭，除非是有某些垄断性的因素可以确保其生存。

人生中的关键选择与公司分类

程序员的一生中会有几次非常关键的选择。

第一次关键选择往往是你毕业后加入了什么样的一个公司。这个公司背后往往隐含着你所加入的领域，加入搜索引擎公司和加入外包公司意义绝对不一样。男怕入错行在这里体现的特别明显，在软件的不同子行业中穿越代价太大，很多人往往是这次选择了什么，就持续做什么。同时这个公司所处的生命周期往往也对一个人的发展影响很大，在 2000 年左右，同样是做通讯设备的公司有所谓“巨大中华”。今时今日看来，在那个时间点加入大唐和加入华为，10 年之后差别还是很大的。

第二次关键选择是毕业后的 3~5 年。这次选择很可能是关于换工作换行业，也可能是关于选择持续做技术还是转到管理方向，或者两者都有。3~5 年间，不同人的累积可以差到很悬殊的地步，上面的两类选择只对其中表现比较优秀的才有现实意义。没有积累的，往往既没有选择管理的机会，也没有换工作换行业的机会，或者说即使换工作也只是表面的变化，而工作的内涵则很难变化。因为这段时间一定程度上还是在打基础，所以多少还有转换子行业的机会，在此之后就非常困难了。

第三次选择是毕业后 10 年左右。这次选择是关于保持生活现状，还是打破现状，进行冒险。比较优秀的人在这个时候，往往会有一定成绩，很可能算是比较中层的位置。但接下来的生活很可能变得重复性比较强。这时一个人可以选择换一种活法，还是持续自己现有的

工作和生活风格。

当然就像武侠的世界中有没有品级的高手一样，这世界上也总是有例外和英雄豪杰，大学生也可以创业，开创自己的商业帝国。上面只是正常的情形罢了。

7.1.2 具体类别的点评：外包行业

很难笼统的讲外包行业好或不好，有前途没有前途，因为外包也有很多层次。

假设说一家公司把一款手机从设计到制造的所有环节都外包给一家公司，只是最后自己进行贴牌，那么对于承接这种外包业务的公司而言，单纯从技术角度看与自己进行产品开发并无差别。

假设说一家银行把需求之外所有的业务都外包给一家公司，那么对于承接这种业务的外包公司而言，就更像是银行的 IT 部门。

上面这类外包与特定企业的核心业务相关联层次是比较高的。与此相对应，外包业务的层次可以逐级降低，比如：同样是银行业务，在外包过程中也可以把基础设计都完成之后，把量较大，但没有技术难度的编码工作外包给其他公司。这类工作往往是由成本控制而触发，所以在分工链条上所处的位置越低，工作越容易辛苦，且收入较低。

这点上可以看一点数据，当前国内最大的外包公司是文思海辉，根据公开数据，文思海辉在 2012 年总计有员工 23270 名，总收入是 3.59 亿美元，大致是 22.7 亿人民币，平均每人年创造不到 10 万元的收入。与之相对应，腾讯 2012 年总计有员工 24160 名，总收入 438.9 亿人民币，平均每人年大约创造 181.6 万元。这种数据大致可以说明从分工链条的角度看，国内的外包公司仍然有很大的提升空间。

那究竟加入外包公司是怎么样的一种选择？

这不能说是很差的选择，但如果让我来排个顺序则首先是知名的独立软件开发商(ISV)，其次是前景比较好的独立软件供应商，接下来才是做高端外包业务的公司，做低端外包业务的公司大多时候不是什么好的选择，因为对个人成长不利。

从杰克•韦尔奇的发言分析外包的内在驱动力

外包这事情在今年来一直是一个热点，很多人坚信在软件领域中，中国和印度一样在外包上具有优势，那外包这事情自身内在的驱动力是什么？这里根据知名 CEO 的发言来做一点分析。

杰克•韦尔奇在《赢》里面这么说：

我们的战略其实是有更加明确的方向，即 GE 将要逐渐放弃那些已经成为大众化产业的领域，而更多转向创造高价值的技术性产品，或者是转向销售服务而不是实物的产业。作为战略行动的一部分，我们需要大规模的提升自己的人力资源--人才---空前的关注培训和发展。

这样的战略抉择是在 20 世纪 70 年代做出的，当时我们遭受到了来自日本企业的沉重打击。在电视机、空调机等产业上，我们在传统上保持着合理的利润率，可是日本人迅速的使其走向了大众化。

……

然而，看到GE 资本公司在70 年代末的表现之后，我感到震惊（与欣慰）。在金融服务产业赚钱是多么容易，特别是与GE 报表中的其他产业相比而言。

……

而我的建议是，当你思考战略的时候，要考虑反大众化的方向。要尽量创造与众不同的产品和服务，让顾客离不开你。把精力放在创新、技术、内部流程、附加服务等任何能使你与众不同的因数上面。

这里的核心词汇是大众化与反大众化、利润率。大众化的东西不可能就不要了，但关键是作为公司必须力争走到反大众化的道路上，即高附加值的领域中，否则企业将无法持续。

在谈到外包问题时，杰克•韦尔奇在《赢的答案》里写道：

当然，外包的道路也不会一帆风顺，工人失业就是一个非常棘手的问题。但是，我们要从长远的角度来看待失业这个问题，外包不仅仅是世界经济不可缺少的一部分，对于我们也至关重要。

“不可缺少”是因为“经济”的定义就是满足消费者的需求。人们都期望用最低的价格买到质量最高的商品。如果企业不在全世界范围内寻找成本优势和创新人才，我们就无法满足人们的这种期望。……难怪目前，绝大多数政治家都在大力鼓吹全球一体化的整体优势。

这里的关键词是外包、成本优势、全球一体化的整体优势。从上面两段描述中应该可以比较清晰的看出外包的根本目的就是节约成本，不是为了创新等高附加值工作，也就是说通过把附加值低的工作转移到人力成本相对便宜的地区来降低成本是外包的根本目的。这是一种无可避免的潮流，A 公司做了，A 就可以大幅拉低成本，B 和C 也就必须得跟进。

7.1.3 具体类别的点评：互联网行业

如果说IT 行业是朝阳行业，那互联网绝对是朝阳中的朝阳。在这个领域中我们看到了门户网站，看到了搜索引擎，看到了社交网络，看到了各种云平台和大数据，但故事应该远未终结。未来究竟出现什么样的产品很难预测，但未来的产品会和互联网紧密相关这点错的可能性很低。

所以在行业选择上加入一家以互联网业务为核心的公司是不错的选择，从发展的角度看很同等条件下要比加入通信、金融这样的行业更有优势一点。

假如一个人想创业，那这就更是一种必然选择，互联网是智力密集型的工作，需要一定资本，但对资本的要求并不高，几个人拉到风险投资后，假设网站开始创业怎么也比研发路由器开始创业更适应于一般人一点。

这里面在分类上有一个微妙的地方需要注意，搜索引擎、大数据、社交网络可以认为是互联网，但挪到互联网上的ERP 也还是ERP，不能算是互联网行业，这个差别很大，考察公司的时候要注意。

7.1.4 具体类别的点评：外企

按国别来区分企业，误杀的几率很大。但毕竟同一国家的企业还是有些共通的特征，因此这里择取两个视角做一个简单的分析。一个视角是透明天花板的高低；一个则是规范程度。

什么叫透明天花板？职业路径被封死在某个高度以下，但这种封锁在可见制度层面又不存在，这就是透明天花板，往往是做到一定高度的人对此感受的更为清晰。平心而论，透明天花板在那里都有，本身倒不一定就具有贬义。但不同企业中其高度确实不同。如果要排个次序，那就是是国内企业高于欧美企业，欧美企业高于日韩企业。

理论上讲一个人在国内企业里的发展是不受高度限制的，可以想见联想，阿里巴巴未来的接班人还是个中国人，空降个黑人或白人 CEO 的可能性不大。而欧美企业相对开放，至少在大中华区，高层人士里还主要是华人，但想再往上走则很艰难。这可以从李开复老师的自传里读出些端倪，李老师无疑非常优秀，但从自传里绝对看不到成为微软或者 Google CEO 的可能性。高晓松老师在优酷上做了个很有意思的节目叫《晓说》，里面更是很直接的谈到这一点：在好莱坞就是犹太人升迁快，华人虽然收入高，但就是有天花板，要很辛苦工作来维持自己的收入。

在日韩企业这点上就收的比较紧一点，各个子公司的负责人基本上会选本国人。

所以假如说你认为自己非常优秀，并且期望无限的发展空间，那么优选本土公司（也许注册地不是中国）是对的，虽然它们的 CEO 没准也是美国华裔，虽然它们可能会有一些负面的报道，虽然不管怎样成为 CEO 都是一种可能性非常低期望，但至少存在着这种可能性。

顺道说一句，从这个角度上看，我们真有理由希望这个国家和对应的企业强大，只有如此，大多数和本土的优秀人才会有更加光明的未来。

从规范程度来讲，越是本土企业，反而可能会越差一点，这也符合知己知彼百战不殆原则，虽然有点黑色幽默。而跨国企业往往是些大公司，在规范性这点上都会非常的注意。我个人倒是感觉，只要没滑落到无赖公司的地步，稍微有点不规范也是可以接受的，犯不上刀兵相见。所以说单纯的希望成为优秀的工程师，那么优秀的外企仍然是个不错的选择。

7.1.5 具体类别的点评：受非市场因素影响大的公司

非市场因素影响大的公司是个诡异的类别，完全是我杜撰出来的，这个类别的公司特征是可以因为特别的机会（某个大单或某项特别资助）在短时间蓬勃发展，但接下来又很可能会在短时间轰然倒下。在 IT 业中最典型的例子托普，有点年纪的 IT 人应该还记得这家公司。

我个人倾向于认为应该规避这类公司，因为这类公司通常并不会把精力放在产品的开发上，而会放在非市场因素的经营上，这对个人成长并不是很有利，尤其是走技术方向的人。一旦公司出现危机，那么个人反倒可能会因此陷入困境。

上面的文字有点含糊，为了把事情说的更清楚一点，我们来看一个历史上的故事。

中国人里面不知道胡雪岩的恐怕不多。胡雪岩出身不高，几近于白手起家，先后得王有龄、左宗棠之助迅速成为显赫一时的红顶商人。百度百科上对这段往事的记载是：

他从扫地、倒尿壶等杂役干起，三年师满后，就因勤劳、踏实成了钱庄正式的伙计。正是在这一时期，胡雪岩靠患难知交王有龄的帮助，一跃而成为杭州一富。

王有龄，字英九，号雪轩，福建侯官人。在道光年间，王有龄就已捐了浙江盐运使，但无钱进京。后胡雪岩慧眼识珠，认定其前途不凡，便资助了王五百两银子，叫王有龄速速进京混个官职。后王有龄在天津遇到故交侍郎何桂清，经其推荐到浙江巡抚门下，当了粮台总办。王有龄发迹后并未忘记当年胡雪岩知遇之恩，于是资助胡雪岩自开钱庄，号为阜康。之后，随着王有龄的不断高升，胡雪岩的生意也越做越大，除钱庄外，还开起了许多的店铺。

庚申之变成为胡雪岩大发展的起点。在庚申之变中，胡雪岩处变不惊，暗中与军界搭上了钩，大量的募兵经费存于胡的钱庄中，后又被王有龄委以办粮械、综理槽运等重任，几乎掌握了浙江一半以上的战时财经，为今后的发展奠定了良好的基础。

胡雪岩之所以可以迅速倔起，除了得益于王有龄之外，另一个人也起到了重要的作用，

这个人就是左宗棠。1862 年，王有龄因丧失城池而自缢身亡。经曾国藩保荐，左宗棠继任浙江巡抚一职。左宗棠所部在安徽时晌项已欠近五个月，饿死及战死者众多。此番进兵浙江，粮饱短缺等问题依然困扰着左宗棠，令他苦恼无比。急于寻找到新靠山的胡雪岩又紧紧地抓住了这次机会：他雪中送炭，在战争环境下，出色地完成了在三天之内筹齐十万石粮食的几乎不可能完成的任务，在左宗棠面前一展自己的才能，得到了左的赏识并被委以重任。在深得左宗棠信任后，胡雪岩常以亦官亦商的身份往来于宁波、上海等洋人聚集的通商口岸间。他在经办粮台转运、接济军需物资之余，还紧紧抓住与外国人交往的机会，勾结外国军官，为左宗棠训练了约千余人、全部用洋枪洋炮装备的常捷军。这支军队曾经与清军联合进攻过宁波、奉代、绍兴等地。 胡雪岩是一位商人，商人自然把利益放在第一位。在左宗棠任职期间，胡雪岩管理赈抚局事务。他设立粥厂、善堂、义垫，修复名寺古刹，收硷了数十万具暴骸；恢复了因战乱而一度终止的牛车，方便了百姓；向官绅大户劝捐，以解决战后财政危机等事务。胡雪岩因此名声大振，信誉度也大大提高。这样，财源滚滚来也就不在话下了。自清军攻取浙江后，大小将官将所掠之物不论大小，全数存在胡雪岩的钱庄中。胡以此为资本，从事贸易活动，在各市镇设立商号，利润颇丰，短短几年，家产已超过千万。

没仔细考证过，也许上面的描述与历史有所偏差，但其崛起与左宗棠有紧密关联这一点，大致不错。胡雪岩成功的快，败落的也很快。败落的原因很多其中关键的一条就是在李鸿章与左宗棠的内斗。从发迹到败落前后大概 20 年。但在现代这种变化节奏因为某些原因变快了，往往不要 20 年，托普走完这条路用了 10 年左右，未来就可能会更短。从这个角度看，这类公司是不适合纯粹的工程师的。

§7.2 选择公司的方法

在弄清楚公司的分类方法，以及某些类别公司的长处和短处后，就可以开始考虑如何去选择公司。

经常会看到这样一些问题，比如：是去大公司好呢，还是去小公司好呢？是去用 ASP.net 做 ERP 的公司好呢，还是去做 Mobile 应用的公司好呢？这些问题的当事人大多是希望别人给个具体答案的，但实际上上非当事人是很难给具体答案的，而只能提供原则。因为最终的判断同时依赖于公司和个人的偏好。

做这类选择的基本过程是这样：

- 先给公司分类，并弄清楚特定类别公司的利弊得失。
  这正是上一节中已经提到东西。
- 考虑自身的状况进行选择。
  这时候主要需考虑两个事情：一个是工作要和自己的根基契合，要扬长避短，使工作成为发挥自己长处的场所，而非相反；一个是在发展、赚钱、安稳和兴趣之间取找到平衡点。前者是很理智的一种判断，后者则是主观的一种选择。下面对这两点做一点详细说明。

7.2.1 使工作和自己的根基契合

什么叫根基和工作不契合？

我们来想象一下，一个人算法、数据结构、数学的基础很好，特别喜欢计算性工作，但跑去做纯粹的应用开发，这就是根基和工作的不契合。

不是说应用开发就简单，而是说应用开发的难度往往不来源于算法、数据结构这类东西。如果真的很擅长这方面，那更适合去开发基础库、乃至编译器这类东西。

软件是无限宽广的一个概念，恰如前面所说，里面流派众多，不同流派都有自己的难

度，但这种难度的来源往往不同。

假设说一个人完全是自学成才，那他能不能成为软件开发高手？当然可以。但这种情况更适合去挑战内核设备驱动这类有难度的工作，而不适合去挑战编译器这类有难度的工作。当然后者也不是完全没可能，只是说挑战后者更为艰难。设备内核驱动的难处来源于对设备、对操作系统、对协议的理解，成为这方面高手可以逐渐累积。但编译器这类工作需要事先累积很多东西，接下来一下子爆发出来，这对自学成才者往往就很难---当然到现在为止我也不知道国内那家公司是自己开发编译器的。

不契合首先造成一种浪费，其次会对自己的职业路径形成障碍。这种选择错位，大致可以有两种表现。

把自己升级使用，这种情形下公司大多时候不干，所以情况不多，但把自己置身在前行无路的状况下却很可能。比如：本科这样的程度即使很精通各种编程技术，也很难去做纯粹的研发工作，在纯粹的研发工作中（图像算法等），只能扮演一种助手性的角色。这也就意味着，如果学历不够却选了一种偏研发的道路，那么就容易卡死在某个位置上。

与上述例子相比，把自己降级使用因为种种原因反倒更常见，所以个人主要力争避免的是别把自己降级使用。比如：很优秀的人每天主要使用拷贝粘贴而不用动脑子就可以搞定手里的工作，这样做个几年这个人也就不优秀了。

从具体方法上来讲，在对公司进行分类后，先看一下公司的核心竞争力和技术路径。不管目标是做技术还是做管理，一旦其技术路径并不是自己能走下去的，那就不太适合自己。比如说：算法、数学基础不好的人就不太适合加入做 TTS 的公司。

7.2.2 当前可得到什么、将来可得到什么

抛开价值观与自己契合与否的考量，选公司最关键的事情是弄清楚在公司里当前可以得到什么，未来可以得到什么？当前可以得到的一般是收入和成长机会，未来可以得到的是一种能力与履历。其中能力贡献于自己的价值，而履历则贡献于自己的表达力，显然的 Google 工作过的人与飞鸟公司（杜撰的公司名）工作过的人在一般衡量标准中其价值有巨大差异，即使这两个人技能水平相近。

对短期目标和长期目标的平衡可以从四个维度上进行考察：收入、发展、安稳和兴趣。

收入、发展、安稳和兴趣这四个维度所导致的可能选择往往是矛盾的。但如果非要给这四个因素按权重排个优先级的话，更合理的结果是：兴趣>发展>赚钱和安稳。后两者纯属是个人选择，很难清晰区分。但其它几个因素的权重却大致是确定的。

在进行其他说明前，需要补充的一点是这里说的兴趣不是那种很快会来很快会走的兴趣，而是说能够伴随自己一生的偏好。

兴趣之所以大于发展是因为人生要想有高度，必须持续的投入，而要想在一生中持续的投入，没有一个关键支撑，那是不可能的。而没有一种持续的投入，最终可达到的高度就会被限定在一定尺度以下。比如说：一个人非常喜欢写程序，有钻研精神。这时候从行业分类上看，就不太适合介入金融、ERP 这类行业，因为这类行业中的软件往往是一种业务流，同现实世界关联很紧，并不是一种纯粹的程序。与之相对应，在互联网公司开发浏览器或者开发软硬件结合的产品就更像一种纯粹的技术。这个人一旦加入了前一种类型的公司，可能做做就会失去兴趣，会失去前进的动力，后者则会与这个类型的人的兴趣点契合的好一点。而这里面很可怕的一个陷阱是一旦你选择了某个领域比如金融，再想跨越到其他的领域比如

说游戏，那成功几率极低。因此兴趣是大于发展、安稳与赚钱的权重。

但这里有个极容易被误解的东西，那就是虽然兴趣非常关键，但实际上大多数人其实并没有真的兴趣，而会把一时的喜好误认为兴趣，这种兴趣因为会在时间轴上频繁变化而毫无价值，过多考虑这种兴趣会让自己走入误区，为自己造成损伤。这时候要对自己有个清醒的认识，如果自己不是一个对兴趣非常执着，而是做什么都还可以接受的人，那么就可以把兴趣的优先级放在较低的位置上。

发展之所以大于安稳和赚钱，道理也很简单。年纪越小，未来越重要，没有个人价值的提高，未来既不会赚到钱，也不会安稳。

平衡这四点之后，就可以找到一些大多时候都适用的选择方法：

- 知名软件公司在大多时候在这四个方面可以达成很好的均衡，是首先考虑的对象。

  显然的国外的 Google、微软等，国内的阿里等即可以提供与兴趣相吻合的工作，又可以提供合适的发展机会以及安稳的，不错的收入。所以如果有机会加入这类公司是不太需要考虑太多的。

  有一种情形需要稍微注意。比如某家公司设立的某个子公司主要目的是节约成本，并只做限定范围内的业务的，这种可能会造成流动性上的问题。

- 软件不是主营的公司在一个或多个方面都有负面影响应该尽力回避。

  在某些公司里，软件是给其他业务配套的，也就是说是非核心业务。这种工作岗位通常同时对兴趣、发展有负面影响。大多时候应该回避。除非你真的非常认可这类工作岗位的安稳，并且也能接受相应的收入水平。

  假设说你真的在软件不是主营的公司里做软件，那么你大概会碰见下面这些事情：

  首先是永远要用最简单最直接的技术做基本能用的软件。比如用 Visual Studio 开发一个界面和数据库进行连接，直到他能用。注意，能用往往是这类软件的终点。太好的代码是不需要的。接下来就可以开始做下一个软件了。而你一旦这么干了，你的技术水平就得不到提高，可替换性也就始终很强。

  其次是你可能不会被重视，会和周围的人格格不入。你的领导可能只会给你非常泛泛的指示，但你必须一丝不苟的按照这些安排去做。

  很多事情在中国和美国都会不一样，但这事儿上实在是相似极了。在《软件随想录》里 Joel 说的更直接：千万不要去干什么内部软件，那会把你榨干。

- 衰落期的公司、分工上处在下游的公司、无核心竞争力的公司并非优选。

  这些公司可能能满足兴趣需要，但公司自身在满足发展需要、赚钱和安稳上会相对较差。没选择的情形下这些公司也是一种选择，但不是优选。这里没法具体去列举那个公司处在衰落期，那个公司在分工上处在下游，那个公司没有核心竞争力，只能列几条判断的原则。

  判断是否处在衰落期，最直接的办法是财务数据，对于上市公司而言获取这种数据并不困难。销售额和利润在处连续 2 年以上下降的公司，大概是有问题。

  对于小公司财务数据大致是不公开的，这时候则只能判定领域。一些存在时间比较长且没形成自己品牌、规模或者技术优势的公司，更可能是生存艰难的。比如：做 MIS 系统的公司。

  在新兴领域中小公司有可能以小博大，高速发展，成为大公司，这种领域往往是蓝海，这点可以回头考虑下 2000 年的阿里，百度，腾讯。而在成熟的领域中，势力地图已经划分完毕，留给后来人的的工作要么是利润空间比较低，要么需要大额资本进行驱动。淘宝天猫可以逐步发展，苏宁想启动类似的业务，那就必须先投入足够的钱，再有小公司想模仿淘宝天猫，已经变得很艰难，并且成功几率极低。

- 越有梦想，越想快速成长，就越适合在一线城市工作。越想安稳的平缓进步就越适合在二线城市。但在二线城市的选择上要尽可能选择那种有软件园，政府提供相应

配套设施，软件园中集中了很多软件公司的二线城市。选择这种城市要比选择软件非主体，只有很少几家软件公司的城市在可流动性上有优势。

感受大公司中主营业务的威力？

李开复先生的《世界因你不同》里面提到了.NET 开发在微软内部扭曲变形的过程:

在 2000 年的时候，微软决定启动.NET 计划，那时候规划中的.NET 和我们现在看到的.NET 并不相同，那时的目标是基于浏览器可以运行所有的应用软件，和今天的云计算概念非常相似（注意这是 2000 年），电子邮件、即时通讯、登录、Office、高质量的打印、机器翻译、人工翻译等都会被整合到网络之中。

但一旦开始行动时，李开复先生遭遇了“人的问题”（即是政治问题），待整合的团队有三个：IE、MSN Explore、NetDocs，其中 MSN Explore 和 IE 是死对头，MSN Explore 和 NetDocs 因为使用不同技术有过节，与此同时，庞然大物 Office 团队则因为 NetDocs 意图取代 Office 而恨之入骨。

这种复杂的关系导致从各个团队抽调人员非常困难，最终只有 MSN Explore 被整合到了李开复先生这边，但这个团队只有 100 多人，根本无法开始.NET 的宏伟计划。

正在这时微软的一个一个强势人物：吉姆•阿尔钦回到了公司。吉姆•阿尔钦是 Windows 团队的负责人，1995 年开始一直负责 Windows98 和 XP 的开发，他是 Windows 的狂热拥护者，一回来就把.NET 计划批评的一无是处。他说：你们知不知道是谁在付你们薪水？是 Windows！我的血液里流的是视窗的四色血液。你们呢！难道是冷血的？“他又到盖茨的面前威胁如果公司执意而行，他就辞职。在强势威胁后，阿尔钦通过 Windows 的远景说服了比尔盖茨。

最终结局是取消原来的.NET 计划，重点还是 Windows，接下来就启动了著名的 Vista 的开发。

---上面的案例参照《世界因你不同》缩略而成。

上述这样的一个案例落在不同的人眼里，看到的东西应该不同，可能是政治的力量，也可能是其他。这里我想强调的则是强势部门的力量。

吉姆•阿尔钦这个人为什么有这么大的影响力？原因可能有很多，但其中一定绕不开的一条是他所负责的 Windows 是微软的根基所在，是主营业务。

在公司之中往往可以有很多产品线，但这些产品线往往并非均摊权重，而是要有君臣佐使。这种差异往往会成为内部资源分配、职位晋升的影响因素。在这种情形下，无疑的主营业务相关人等会被重点关注。

§7.3 小结

让我们再回到最开始时提到的问题：是去大公司好呢，还是去小公司好呢？是去用 ASP.net 做 ERP 的公司好呢，还是去做 Mobile 应用的公司好呢？

这时答案已经比较清楚了。对于前者，显然是去知名的大公司比较好。对于后者，同等条件下，首先考虑下那个与自己的兴趣契合的比较好，也就是说自己期望在那个行业里做长期发展。接下来要考虑回避掉无核心竞争力的公司。

图 7-1：选择公司的过程

从起薪看公司的差异

假如说一个人要毕业了，那么他的起薪大概会是多少？又有那些因素影响了起薪的高低？

在国内起薪差距很大，以本科而论从 2K 到 10 几 K 不等。

从影响因素来看，学校和公司的影响最大。简单来讲牛的公司往往就到几个特定的学校招人，所以没考到好学校的人在起跑线上略输一筹，当然这可以在以后搬回来。

如果到非常有名的企业里，在北上广这样的城市，那么起薪可能到 10K，如果到二线城市的小公司，那么起薪则可能跌到 2~3K。这个差距已经很大了。

更关键的是，加入不同的公司，损益还不止限于起薪，在今后的人生中，工作经历的含金量也是不同。

从这个看结论很简单，毕业前好好学习天天向上是不二法门，如何避开学习陷阱则要看第四章。

值得一提的是，在日本大家起薪一样，不管你是东京大学的还是早稻田的，全都在 20 万日元（15K RMB）左右。这在中国人看来非常不可思议，但日本人的逻辑是你想获得高收入，要先证明你的价值，在没证明前，所有人全一样。也不是完全没道理，只能说是不同文化背景的差异。

第八章 六个程序员的故事

此前的几章基本上是在分析并试图建立一种成长的模型，为免这种分析和模型离开现实，下面将考查一些具体的，程序员的故事。

很多人更喜欢看名人的故事，比如李开复老师的故事。这必然是有帮助的，但可参照意义往往并没有想象的那么高。因为大部分人往往即走不进哥伦比亚大学，也走不进卡内基梅隆，这就导致了名人和一般人的起点非常不同。此外美国处在整个 IT 产业链的高端，各种游戏规则也与国内不同，这就导致了李老师的人生路径很难被一般人所复制。

在国内这个情形也同样存在，排名前五的大学往往会在起点上就与大多数不同。很多公司的招聘，大致是按照某种学校的排名来设定招聘范围的。

我曾经很客观的评估了下自己，结论是即使再给我一次机会，进很好的高中，自己也努力学习，我也考不进清华，但即使如此我也要做 IT，我也要谋求发展。相信大部分人的经历更类似于我，而非是李开复老师，因此下面收集了几个普通程序员的成长故事来给大家做些参考。有的留下了名字，有的则隐去了真实姓名，但不管怎样，这里的故事很草根，它其实就在你我身边，相信参照意义也会更强一些。

§8.1 一个 40 岁程序员的无奈

8.1.1 佚名老程序员的故事

老 A 有点莫名其妙的就混进了程序员的队伍，稀里糊涂做了几年之后，猛一抬头发现自己快 40 岁了。老 A 出身名门，是中国顶级学校的毕业生，毕业后跑到英国留学七年，得了两个硕士学位，按理讲奋斗个英国绿卡应该并不是很难，不过老 A 还是选择了回国。

老 A 自身智商是不缺的，否则也考不进顶级学校，但却不太考虑自己的事情，反倒对如何解决台湾问题，如何解决和印度的关系思考颇多，让大家印象非常深刻的一个观点是：国家应该使劲鼓励往澳大利亚这类地方移民，改变它的选民构成，这样他自然就变成亲中的。

这类小事糊涂体现在，他归国后并没太多的考虑自己的发展，而是随便找了一个苏州的做打印机驱动的公司。

公司当时刚刚开始起步，需要同不同文化背景的人做大量沟通，并处理与设备相关的极多的各种细节。

老 A 的背景和性格都不太适合做这类工作，沟通的时候太以自我为中心，但实际上食物链下端的公司是处在弱势地位上的，更要扮演倾听者的角色；再加上人又有点疏阔，细节问题不大处理的来，因此虽然一开始被给予厚望，但成绩却一直平平。老 A 在这家公司坚持三年之后在看不到任何前景的情况下选择了离开。

离开后老 A 加入了一家纯粹做外包的公司，并被派去日本。这时候，工作范围一下扩的很开，什么野村证券的金融系统，某个网站的前端开发等等都一一做过。每当换一家公司，都要从头学点东西。

这些工作本质上和上一家公司差别并不太大，但总是很繁琐，和以前一样老 A 完成的比较一般。这样再加上日本的经济实在不景气，很快老 A 派不出去了，无奈之下，老 A 又回到了国内。可是时间不等人，一来二去老 A 已经差不多 40 岁了。

从日本回国之后，由于前面几年积累不多，老 A 只好回到离家乡比较近的一个城市，继续在对日外包行业里摸爬滚打。

8.1.2 感悟程序人生

老 A 的路其实走错了。考入等级学校并能出国留学可以讲是给自己挣得了不错的选择权。但就像再多的遗产也经不起子女挥霍一样，基础再好也经不起错误的选择。几次相对比较不好的选择之后，选择权就消失殆尽，而时间不等人，价值没提升年龄却提升的同时，人生也就开始有点困顿。

我们可以用这本书里提到的各个维度来观察一下老 A 的人生。

价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

老 A 在几乎所有项目上做错了选择。

从增值的角度看，首先是大方向上过于糊涂。以老 A 的履历，如果走纯技术路线，那就要到一家技术含量高的公司，否则就要迅速向管理职转型。老 A 却选择在一家技术路径不长的公司里做技术。

即使单纯是从做技术的角度看，老 A 也还是做错了。我们在前面曾经提到，要持续增值，不要失去焦点，你不能学的是嵌入式底层开发，回头做前端开发。老 A 又恰巧选择了一个自己完全陌生的领域---打印机驱动程序。这类工作对他以及对其他毕业生是同样的陌生，难度又不高，在这类工作面前，他的学历背景等等完全没有优势，等于主动的为自己做了重置，也等于人生的增值失去了焦点。

既然做错了选择，本来就应该赶紧在“表达力”这项上调整自己，迅速适应当前情境，这样依赖于在公司起步阶段进入公司这一优势，也可以谋求一定发展。但老 A 又错了，他看到了许多自己不适应，但不涉及是非的地方，但并没有积极改变自己，最终导致了自己的出走。

当老 A 第一次从公司出走的时候，他的价值、表达力、稀缺性并没有因为近三年的工作经历有任何提升，仍然吃的是老本（名校，留学经历）。

接下来，在公司的选择上老 A 再一次错了，第一次的选择本来就不太正确，但第二次选择就几乎把自己逼到了死胡同里面，转到纯粹的外包领域这又是一次重置，让之前三年的经验归零。

去日本一事更是让事情雪上加霜，短期来看也许可以多赚点钱，可几乎一直处在食物链的末端，没有任何技术积累，这直接决定了回国后的选择空间。回国后再一次进入外包公司，以老 A 的年纪和性格，未来已经很那看到亮色。

老 A 的内心想必是痛苦的。老 A 的情形并非个案，如果我们用心观察就会发现很多很多类似的故事。

有的可能是已经失业，再就业艰难；有的可能是只能维持现状，并等待人生下滑。也许是巧合，这点在 40 岁左右的程序员身上发生的特别惨烈。

这种情形有一个根本的特点，就是自身价值没有质的突破，这样不管你曾经多么风光，多么好运，在 10 年时光面前，往往会被打回本色。

陷入这种困境之后，要想挣脱已经很难，而如何避免陷入这种困境，也许可以从下面李云的故事中获得启示。李云的故事与这里的老 A 可以形成非常鲜明的对比，老 A 是通过高中的努力达到人生高点，但浪费了自己的选择权，使选择权逐步收窄，而李云则是起点很差，但始终坚持增值，最终累积出了高度，逐渐使选择权越来越大。

§8.2 一个普通码农的退场过程

8.2.1 码农退场的的故事

声明：下面这篇文字，非本人所做，而是 CSDN 上流传比较广的一个 2007 年的老帖，原作者已经联系不到，但这个故事真的很有代表性，因此把它录在这里，与广大程序员再次分享。为了保持原滋原味，基本未对原文做任何修改，而只是调整某些敏感词汇和小的笔误。感谢原作者。

再过几天，我就正式告别程序员生涯了，这也是我最后一次以职业程序员身份在 CSDN 发表文章。小弟谈谈入行几年来的感受，做一个人生阶段的自我总结，同时希望能给后来者带来点参考意见，能在这段路上走的更好。

本人 2002 年下半年正式入行，至今 2007 年 4 月一直从事软件开发工作。

上大学选择这个专业是阴差阳错，但接触之后对计算机产生了强烈的兴趣，对写软件有一种强烈的冲动。

软件成型后，那种成就感和自豪感能给我难以名状的满足。

那时，喜欢看侯捷的书，对核心技术和核心技术人员由衷的崇拜，对技术的追求和水平的提高有一种莫名的狂热，当时我想只要能从事软件开发的工作，起初的薪金待遇可以不计，等我技术方面成熟后，自己就有更高更好的选择了，程序员→系统分析员→项目经理→自己的软件公司，这是当时我一个朦胧的程序人生规划。

现在想来，不禁一番唏嘘。

我记得我在培训的时候，一个培训的老师当时是本地一家有名的高科技企业的 CTO，确实是专业人才，就是不太会讲课。我问他做程序员的感受，他说经常写程序经常写到凌晨 2 点钟，很累不过很有意思，因为他喜欢这份职业（当然喜欢啊，他月薪 7000-8000，2001 年，济南），但也干不长啊，他已经做好了转行的准备，去做和计算机相关的行业。那年他 28 岁，我 22 岁。其实他那番话让我和我的同学已经很羡慕，我们羡慕他的技术深度和现在的岗位层次。金领啊，他就是我们眼里的金领啊。

而另一个技术水平很高的老师（在外企写单片机的，30 岁，月薪 8000-1 万），告诉我的是：迟早要转行，就像他现在来当培训教师一样，原因只有一个：太累。

我没在乎他们的感慨，因为我年轻啊，加班到夜里 2 点很轻松啊，何况写写自己喜欢的软件，很高兴啊。那时对我来说 30 岁只是一个遥远的数字。工作后，感觉完全不一样了。首先很惭愧自己的机遇和能力都不是太好，一直从事基于数据库的信息管理系统的开发（我认为是软件开发里最简单和最基础的方面），换了三家公司，从基础程序员作到了系统分析的层次，现在开始往对外和管理方面发展。可以说粗略的沿着我以前设计的程序人生轨迹走了走。

其间也有过失业的落魄，吃不上饭的紧张，我记的最难的时候到 CSDN 上来发表文章，得到了很多兄弟姐妹的祝福和支持，给了我很大的鼓励，真的谢谢。

（*此处省略 64 字，因为可能会产生不必要的争议。*）

我今年 28 岁，未婚，彻底烦了。为什么？累，没有希望。先说说我的一点感悟：
软件行业分析：
1、开发出售行业适用的单机版软件。
2、开发行业适用的网络版(B/S)软件，一般是大单，几十万到几百万。
3、和行业的政府主管部门合作，推行一些行业方面的应用软件。
补充：做软件一定要做行业软件，才有前途。
这是本人几年来对这个行业的一点分析。

其实第一种情况是软件公司最通常的盈利模式，这种模式软件价格不高，但只要质量站得住脚，可以细水长流，保证公司的成本没问题，做的好还可以盈利不少，但想做大公司很难。

第二种情况，是真正挣钱的情况，接一个大单，什么钱都挣出来了。可以锻炼开发队伍，建立完整的大的开发框架，而且在这个行业里可以造成很大影响，在一个地方实施成功后可以低成本的再推广，占领一片市场。总之一句话可以让一个小公司真正的成长起来。

第三种情况纯粹就是敛财了，和主管部门合作，强行推广软件，绝对的只赚不赔，我想各个地区都有这样的案例，如税务方面的。缺点是这样的公司都受地域性限制，老板钱拿的太舒服，没什么上进心，公司很难做大，不过也成了地方的行业一霸了，也不错。

累，大家都知道就不说了。为什么没有希望呢？因为我发现一个公司真正勤勤恳垦的实干是挣不到大钱的。

真正能挣到大钱的公司完全都是靠老板的个人关系到什么程度，要想在某个行业里成为软件老大，要看你和这个行业里的政府主管部门的关系如何。我看到了太多软件和他们公司的产品，总结起来就是一个字“烂”。可那赚钱的速度... ...。

其实赚多少钱，都是老板的，我们打工的不就是拿个死工资吗，我们更多的人不是连个受剥削的机会都找不到吗?每次面试刚从大学出来的计算机的本科生，我真想对他说：你何必要选择这一行？每次面试那些工作经历比我长，年龄比我大程序员，看着他唯唯诺诺的目光，我就想：曾几何时我也像他这样，被人横眉冷对的面试多少次，以后我是不是还会像他这样，再去看人家的脸色啊？心寒啊！

看着同期毕业的同学，都转了行，在自己的行业里都混的不错，大部分都比自己挣钱多，有干头；就是挣钱少的，他也干的轻松啊，最简单的例子就是和自己项目接洽的企业或政府的信息部主管或网管，懂的不多，轻轻松松，钱比我们的多，有问题老找我们，面对他犯的低级错误，我们还得笑呵呵。我心里确实不平衡啊。

一句话，不当程序员后悔，当了程序员更后悔。

出路在哪？我在找…

1、从程序员，到系统分析，到项目经理。条件：必须是大公司，工资高，福利好,有完整的发展曲线；个人对软件开发有持续的热情。

2、转行到大型企业，事业单位，政府做信息化方面的工作（可以说是网管）。生活有保障，不必太辛苦。条件：一定的能力，一定的人际关系。

3、考研，考博再深造，出国或留校搞教学，培养一代不如一代的本科生，闲时打着大学的名义做做项目，赚个房钱。条件：高学历，一定的经济基础和家庭背景。

4、创业：这个谈起来大发了。这里只说条件：很好的项目，创业精神，一定的经济基础。

5、共享软件：很多程序员的梦想，自己写个软件全世界的卖，光注册费够一家人生活的了。成功少数，但只要有的都发达了。如 ACDsee，优化大师，超级兔子，千千静听。但我告诉你，这方面基本是没法干了，写个小东西挣钱玩玩可以，要靠他吃饭，饿死吧。条件：过硬的专门的软件技术，富有创意的头脑。

6、网站：基本情况和共享软件差不太多，只是比共享软件更好干点。但奇迹照样有，可看看 hao123 的神话和现在很牛的 80 后的富翁。关键你有没有这个本事和这个命了。

7、行业信息化咨询顾问：随着各个行业信息化的普及，企业对这方面人才需求很大。真正实现信息化的企业都需要这样一个既懂软件，又懂行业知识的人员，他和网管还是有区

别的，他的要求更高些，更像一个自由职业者，专家类型的，这样的人放在企业里小的是个主管，大的是个副总。条件：很深的行业内部的技术或管理经验，较强的软件开发或实施经验；通常 35 岁以上才是成熟人才，因为经验是要经过历练的。

8、转行，彻底的转行。干不下去，精力不够了，脑子不灵了，钱太少。只要你够理由，你就走。重新开始另一段新的生活，有什么了不起的，哪里也饿不死我这个干软件的。

我是哪种人，我说我是第 8 种人，看看能不能兼第 5，6 种人。

我大学由于种种原因没毕业，最高学历是高中。就学历而讲，能干到我目前这个水平我觉的可以了，是时候激流勇退了。

就职业规划和财富而言，我这几年走的路并不成功，最起码无奈的转行本身就是一种失败。就我的人生而言，我觉的很成功。我了却了自己的一个人生梦想，在短时间内品尝了一个“高科技”行业的酸甜苦辣，技术出身也使我比别人多了一份淡定和从容。现在我可以放下这段旅程，再来一个新的开始。

程序员兄弟们别自卑，说到优势我们有很多：

1、聪明的头脑，较高的智商。有人说程序员呆，不会为人处事，只会和机器打交道，没前途没希望。我告诉你，程序员愿意和机器打交道是因为他专注于技术，是职业特点，如果我们程序员的头脑用到一般行业，企业，政府单位里去耍耍阴谋诡计，骗骗人，卖卖产品或套套别人的话，我敢说，他们十个人也玩不过我们一个人。俗一点：就他们那点智商，也就骗骗鬼啊。

2、创新精神，学习能力和频繁的知识更新速度。做软件的都知道，干一行的软件，就得学一行的知识，这一行的知识掌握的越丰富，软件才可能写的越好。我们都具备很好的学习能力，学习新知识，新技术的能力。不敢说每个做过的行业我们多么了解，最起码我们总是站在风头浪尖上，高屋建瓴，问题看得远，想的长（要不你怎么去设计数据库啊）。我们能以非专业人士的角度，系统的分析出一个行业某方面的流程，那当我们就做这个行业时，我们对我们的能力还没有信心吗？

3、扎实的工作态度，未雨绸缪的危机意识。扎实的工作态度是每个合格的程序员都应该具备的，因为我们要对代码负责；谈到危机意识，我想大多数程序员都和我一样吃着碗里的，看着锅里的吧，也是被社会逼的没办法。其实这都成了我们的优点了，以后从事哪个行业，都需要这两点精神。

我要走了，去干个和软件根本不搭边的行业，我去干是因为我是老板之一，而且钱绝对比现在好赚。过年的时候，我有个外甥刚大学毕业，非要做软件，我给他了以下建议，算是为后来者留一点东西：

1、能进大公司就别去小公司，在大公司里你能接受真正正统软件开发教育，比到小公司当个什么啥都干，啥都不精的主管强。

2、不断的学习，注意技术积累和更新，那是你唯一的资本。

3、做软硬件结合方面的开发，单片机的开发，嵌入式系统的开发，比较有前途而且门槛高。但凡基于数据库的开发，不管是.NET 平台的，J2EE 平台的，VC,DELPHI,PB,VB 都是扯淡，其核心价值是开发人员的经验而不是技术本身。因为真正的核心技术都在国外，中国没有，我发现不管那种语言，最好用的类库或组件都是老外写的。

4、要有个好点的学历，别像我一样。毕竟是个高学历的行业，学历低人家都瞧不起你，你的发展也很有限 。30 岁之前，可考虑弄个高程，CCNA，数据库管理员之类比较有含金量的证书打扮打扮自己，过了 35 岁其实意义就不大了。

何去何从，我们都有自己的路要走。我转行了，我就不再是程序员了吗，不！我只是不在做为别人打工的职业程序员了，我要做自己的终身程序员。闲来时我会为自己写程序，写我愿意写的。当写程序不再是一种职业而是一种兴趣和热情时，他才会陪伴我一辈子。我还会再来 CSDN，做为一个非专业人士，一个轻松的真正的程序员而来。未来的一天，当我老的时候，不管那时我有什么成就，或不名一文，如果别人问起我以前是干什么的，我希望仍能自豪的回答：“我曾经是一名软件工程师”。

8.2.2 感悟程序人生

上面这段文字里，有比较大的一部分是个人的心路历程，而非是一种经历，但正是从这些心路历程里，我们可以很真切的感受到一个要退场的程序员真实的内心世界。

很多人也许看的很感动，感动之后且让我们回到理性的世界。
我们还是用这本书里提到的各个维度来观察一下上述这段人生。

价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

决定这段故事主旋律的其实是故事主人公所选择公司（以及行业）在整个社会分工链中的位置。这点直接决定了主人公内涵价值增加不易，最终看不到出路。
当一个人选择沙漠的时候，就必然缺水；当一个人选择做信息管理系统，那就必然会面对激烈的竞争以及纯技术路径短的现实。

在一个特定的领域里，如果几个公司做不好，那可能是公司经营的问题；但如果所有公司都做不好，那就更可能是大环境问题。一旦大环境有问题，那就不管你怎么选择公司，都不太会有起色，这等价于前面谈到的关于公司的各个因素中，只有一个分工位置因素有影响。这种情形下你基本没机会去使用你的表达力，好比说大政治家如果去种地，那多高的政治手腕也影响不了亩产，这是一个道理。

在这种情境下任何一个人大致只能有三个选择：

一是均衡发展，成为公司里最最关键的人。均衡发展意味着，有业务、有技术、有管理、有人脉。因为技术路径较短，所以一定不能只是停留在技术层面。这样在有一定的稀缺性后，可以选择一个利润来源比较稳定，较有潜力的公司。故事的主人公也认识到了这点，列举了几种类型的软件，颇得其中精髓。现实点讲就是，很可能做个三到四年后要把精力转到业务、管理和人脉的构筑上，当然构筑成不成功因人而异。这是通过横向展开，扩展自己的价值，并选择留在这个领域里的办法。

一是短时间累积技术基础，在分工链上升级。信息管理系统所面对的问题和网络应用的开发是共通的，其差异往往是用户规模、数据规模所导致的。极端的讲，12306、淘宝也是一种信息管理系统，但明显其开发需要较高的技能。所以说如果一个人真的很喜欢技术，那不妨去看看关联公司的招聘广告，归纳一下必备的技能。深化算法基础、参加特定的开源项目等都应该都对技术水平提高有所帮助，接下来可以考虑换到故事主人公所说的一些大公司或者需要较高个人技能的公司去，这样就可以站到一个新的起点上。当然这个过程不能太长，否则一旦七老八十，那工作就换不成了。这是利用技能的可流动性，在分工链上爬格子的办法。

最后就只能是转行了，如果有合适的机会，这也许是一个不错的选择，但这样一来，确实会浪费一些之前的积累。

同上一个故事的 A 相比，这段故事的主人公起点要低了一点，因为某种原因，大学没有读完，所以当初并非有特多的选择权，从信息管理系统切入也许有着自己的无奈。从此可以得出一个废话一样的结论：大学的时候给自己一个方向，开始积累，那么接下来的路会宽很多。

§8.3 一个关于项目经理的故事

8.3.1 项目经理的养成日记

L 在 2001 年毕业之后加入到了福建实达公司。

在今天这个公司几乎是很少有人听说了，但在当年实达还是在 IT 这片江湖里有些地位的。当年实达的产品线非常全，有网络、有电脑还有外设。外设里面就包含了终端、打印机和 POS 机。当然也还做过 VCD，不过即使在 2001 年 VCD 这一笔也是作为失败案例来提的。

当年 L 选择了到外部设备公司去做激光打印机驱动程序。那时候实达外设试图开发一款自己的中端激光打印机，因此需要全线配备软硬件人员，正是借助这个机会，L 加入了激光打印机这个团队。

现在想来这个决定有点狂妄，不管是自己还是实达。

随着 iPad 这类平板的兴起，人们的打印需求越来越少，所以激光打印机这类东西越来越不受关注，似乎要被被强塞到旧纸堆里了。但不管它火不火，单纯从技术难度上看，这东西绝对比手机难做，虽然世界上所有做打印机的厂商加起来市值也不一定有苹果高。

激光打印机包括现在的多功能一体机属于是精密机械，机械、光学仪器、硬件、软件、甚至图形字体都搅在一起十分难搞，没有绝大的投资，绝对啃不下来。激光打印机等利润最丰厚的部分是面向企业的各种机型，眼下这块市场始终在富士施乐、佳能、理光等少数几家厂商手中，国内并没有厂商介入这一领域很可能是和介入壁垒过高有关。在 2012 年，很多日本有名的大公司都巨亏，但做激光打印机的还能支持，我想这也和这一领域壁垒过高，竞争对手不多有一定关系。

当时做这个项目的时候，团队里的人员都很痛苦，L 这个做驱动的尤其痛苦。微软为打印机提供了标准的驱动程序叫 Unidrv，如果基于这个来做，虽然也麻烦，但基本不用编程，主要工作是调整配置文件。之后这个驱动负责帮你生成打印机能认识的用专门语言描述的页面数据，那时最主流的页面描述语言有 HP 的 PCL 和著名的 PostScript。

但用 Unidrv 坏处是这样一来你能定制的东西就非常少，很简单的六合一功能都没有。所以如果真的自己开发产品，那驱动程序最好要自己从头写。但 L 当时没认识到自己写其实是不太可能的。

L 当时的水平大概是这么个状况：C++基本会用，但达不到很高的水准，熟读《Windows 核心编程》，但大部分书中讲的内容没有用过，反倒是 MFC 用的比较熟练，但很可惜的是做驱动的时候 MFC 用不上。通过了高级程序员考试，所以各种通用算法和数据结构没什么太大问题。

而从头做驱动需要什么呢，你要了解 Windows 提供的 DDI 接口，要了解图形图像、字体、页面描述语言、色彩的知识。打印机驱动中最好做的是 UI，基本上用 Win32 就行了，最难的是做页面渲染，也就是把 GDI 描述的页面转换成相应页面描述语言(PCL)描述的页面。这一过程非常繁杂，根本不是初级程序员能搞定的。其中不说别的难点，一个图像二值化就能憋死很多初级程序员。彩色页面打到黑白打印机上，要把彩色图转为用黑白两色表示的灰度图，这东西那那么好弄的。

简单来讲是，L 当时是两眼一抹黑，差距太大。

可以举个最简单的例子来形象说明这种差距大到什么程度：L 当时基本的调试也不会，只能用 OutputDebugString()输出用 DbgView 抓 Log，来看程序那里有问题。有人可能很奇怪

说 2001 年时，VC6 的调试器不很好用么。秘密在于，Win98 的打印机驱动是 16 位的，VC6 完全不好使。直到后来找到 SoftICE 才解决了这个问题。

一边做驱动的开发，L 一边把 Unidrv 搞定了，这样基本上不耽误其他硬件开发工作。同时 L 疯狂补各种知识，单只为了把 C++搞通就啃了数本书，其中最难啃的反倒是《C++程序设计语言》，这书即厚又不好懂，那时候那明白什么叫不充分的抽象。但回头想来，读这些书其实对工作帮助不大，打根基的东西总是见效慢。这就和高烧 40 度要赶紧挂水一样，吃中药慢慢熬，就是没有立竿见影的效果。当你需要搞定矢量图形如何转换时，设计原则、面向对象这些东西对你能不能做出来一点帮助也没有，只有当你能做出来了，这些东西可以帮你把事情做好倒是真的。

做了一年多后，大家都发现这活实在不是一个人能干的，团队中就又加了 2 个人，所以 L 勉强算是个小头目了。但即使如此，整体进展仍然不太好，这和公司的策略有关，这家公司的核心产品，其实是针式打印机，并不是针式打印机，实达有着自主研发的整套针式打印机技术。而激光打印机实际上处在摸石头过河的状况，但偏偏这是个资本密集，技术密集的领域，这样一来，进展不顺也就在情理之中了。现在想来最好的解决方法其实是买套代码，在上面定制。等到 Win2000 成为操作系统的主流，借助 DDK 的例子，这个问题一定程度上得到了解决。

说到这里要抛开技术，说下大环境，要不然无法说明 L 的幸运。当时实达集团有三家子公司：实达网络专攻网络设备，如 Modem、路由器等，实达设备专攻外设，如终端、打印机和 Pos，实达电脑则主攻 PC。

当时这三家公司是冰火三重天的状况：实达网络日子很好过，发展也很快；实达外设则稳步发展；实达电脑则活的很不容易。整体来看实达实际上是处在下滑期，顶着 ST 的帽子让公司很难受，新的利润增长点又没找到，尤其是电脑部分，利润是必然是越来越薄。

在实达的三年里 L 的工资竟然没有一点调整，不过 L 当时没太注意这些，还是在研究驱动。借 Win2000 DDK 的启发，发现在打印处理器那个环节可以做很多事情，这样针对页面的各种操作就都可以自己来做了。

正当 L 把这个工作作出点进展的时候，家里出了点事情，于是跳槽，到了苏州一家也是做打印机驱动的公司。这纯粹是种幸运而不是种安排，如果不是家里有事，L 未必会换工作，而在开始衰落的公司里做非主营项目真未必是什么好事情。

在此前整整三年里，L 理清了打印机驱动的体系结构，打下了语言、平台的各种基础，确定了基本开发方法，找到了适合的二值化算法，但真没做出什么太大的贡献。

换到苏州这家公司后，借助过去的经验，职位有所提升，是以 Leader 的 Title 入职的，实际上是项目经理。不过一进公司，L 很吃惊，当年从头开发驱动不过也就三个人，这里维护现有驱动搞了快 20 号人，主要做的事情就是修改现有的驱动程序，每次代码变更量不大。绝对的八旗子弟。

不过麻烦事也出现了，以前那有那么多报告，现在需求的确认，日程的确认，合同的确认，记录的跟踪，问题的总结全都要做。一天到晚大事没有小事不断，会议数目直线上升。再加上外语这一层障碍，还经常出现说半天对方还没明白的状况。

L 信奉适者生存，开始积极转型，把 PPT 写好，把 Excel 写好，文档里不能有小错误，读项目管理书籍，读估算书籍，把口语练好。过往的技术经验和基础还是很有帮助的，这让 L 可以比较快的把握各种需求的规模、难度等。

直到有一天，原来的老大离职了，L 被提升成了部门经理，开始带自己的队伍。

公司由于处在成长期，团队的规模也就在不断扩大，而 L 的责任范围也就随之逐渐扩张。这时候 L 的工作又发生了变化，以前是关注一个项目，现在要关注多个项目，也要关心兄弟们的士气。

L 这时候技术基础还行，也试图坚持写代码，可发现挺难的只能负责那种时限不是很严的，独立性比较强的模块，因为你不知道接下来会发生什么，出差、来客人、兄弟们吵架、流程出问题、临时分配的其它工作等等。

在经历了最初的几次失败之后，L 的队伍逐渐成熟，项目成功的几率逐步提高。现在 L 比较自信，认为自己是一个比较合格的经理了。虽然不能讲任何一个项目到手里之后，都能保证它绝对成功，但至少可以尽可能保证它成功的几率较高。

可惜的是，L 发现自己累积的技术基础一点点荒废，基本程序虽然看的懂，但写起程序来变的很慢，每一行代码调用每一个方法都要去仔细查找帮助文档。

8.3.2 感悟程序人生

我们还是回到之前一直提到的几个维度：
价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

从 L 的经历可以看出来，技术是根本，即使你想做管理，即使你学了可能用不上那么那怕是当敲门砖用，你也要在恰当的领域里有一定的技术基础，这是价值的根本。要不然你可能没有做管理的机会。

任何一个企业招一个人都希望这个人能尽快为公司创造价值，这点和大学绝对不一样，企业不负责再培养你四年。因此，一个人很难对企业说：我什么都不会，我就会做项目管理，你招我吧。一般人很难遇到这么疯狂的企业。

第二点是你加入处在那个阶段的公司非常关键，如果 L 一直坚守在第一家公司，L 是不可能有所提升的。原因很多，但其中最关键的一点是在整个产品线中，驱动程序是在一个配套的地位上，而在第二家公司驱动程序则是公司最核心的业务，这点影响非常大。

第三点是做可流动区域小的工作时，要特别当心。短期来看 L 运气还好，大致解决了自己的生存发展问题，但其实如果以 20 年为尺度来看，问题仍然存在，对打印机驱动从业人员的需求永远不可能像对网站开发人员那么强烈，其划定的区域也就非常有限。实际上这点在 L 的同事身上有了一定体现：你做的事情相对比较生僻，而你在这一领域上又沉淀了很久，一旦走出这个领域，又只有年纪大的劣势，而没有优势，这会导致一个人非常艰难。但以 L 而言，确实又因此而受益，正因为此领域有经验的人员稀少，L 才有机会获得较快的提升。

第四点是一定要避免加入处于衰落期的公司。假设说，领域想对比较生僻，但公司能够为此支付一定的溢价，那也是有得有失。只要没有欺骗，当事人因为合适的薪资而选择了一个流动性不好的工作，这合情合理，风险理应由自己承担。但关键是 L 的第一家公司处在衰落期，三年不调整工资基本等价于降薪。如果 L 不是碰巧离开了，那么纯经济的角度看损失还是要大，一旦正好赶到房价暴涨后买房，那对人生的影响就不是一点半点。

第五点则是关于管理技能的可流动性。虽然说管理技能的大部分是共通的，但由于 L 的技术背景是驱动开发，L 将很难成为电商类项目的经理人。

L 的经历里还潜藏着一个冷幽默，当 L 不会做驱动的时候，他被分配去做驱动；当 L 不会做项目经理的时候，他被分配去做项目经理；当 L 不会做部门经理时，他被分配去做部门经理。大致上是在做自己能力不足以匹配的事情，L 的人生仍然再继续，不知道此后的人生是否仍然会符合这条规律。

§8.4 一个技术牛人的成长经历

8.4.1 杭州李云的技术牛人之路

李云是《专业嵌入式软件开发》一书的作者，为了为后来人提供参考，把自己的经历非常详细的写了下来，在我看来，李云的经历非常的有价值，因此在这里与大家做一点分享。

因篇幅太长，引入的时候去掉了部分内容，下面是李云的故事。

故事的开始得从大学以前开始。从小受“学好数理化，走遍天下都不怕”观念的影响，我认为只要学好数理化就行了，所以偏科很严重，高二时英语还考过 29 分。那时也不爱读书，高三时，别的同学在复习，我却在看《晶体管技术》这类电子技术书。这种状态，直接的结果就是第一次高考落榜了。

落榜的那个暑假，父母为我的出路没少操心。在一天早晨刷牙时，当我妈对我说希望我去复读时，我当时脑海里想“能象表哥那样考上大学那该多好啊！”，在这个念头驱使下，我答应了去复读。从那天开始，我顿悟了，真正知道自己要什么了。在复读的一年里，我学到的一种重要能力是自学，这为以后大学乃至职场学习打下了很好的基础。正因如此，我想给出我的职场第一感悟：自学能力是竞争力之本。

经过复读，高考总成绩提高了 100 多分，但也只够专科线。最终，我被南昌水利水电高等专科学校录取，专业是“供用电技术”。这个专业相信很多人不知其所以然，其实就是电力自动化的变种专业，其专业内容主要是电站、发电厂高电压的继电保护技术。

大学读书期间，我开始有与人在成绩上一争高下的念头了，加上复读一年所获得的自学能力，以及自己的努力，学习相当轻松，尤其是只要与电子技术沾边的课程，都能轻松地胜出。三年共六个学期的学习，我拿了五个一等奖学金，一个二等奖学金。毕业时，我是系里唯一的一名优秀毕业生。期间通过了大学英语四级考试和计算机二级考试，获得了江西省电子技能比赛一等奖。需要提及的是，在大学期间所学的与计算机相关的课程只有：《电子技术基础》、《计算机组成原理》、《计算机软件基础》、《单片机技术》和《Basic 编程语言》。

在大学期间，我完成了人生很重要的一件事 --- 找好了现在的妻子。由于她是浙江人，所以毕业时工作地点毫不犹豫地选择了杭州。那时很多同学的工作还是包分配的，而我来到了杭州的人才市场进行双向选择，那时找一份工作还是相对轻松的（注：我们大学录取那年的招生人数是 90 多万），投出一份简历就找好了工作。第一个工作单位是一家不到 100 人、地处杭州花港观鱼对面（三台山）的电力设备制造民企。

尽管选择去这家民企后立马到公司去做了实地调查，但由于没有社会经验，加上被问的人没如实反应，所以进入这家民企后所了解的情况让人大跌眼镜。另外也了解到单位会通过一些不入流的做法控制我们的户口，不让我们跳槽（那会儿的户口还是相当重要的，结婚要户口证明，有同事就因为户口被控制而登记不了）。而我们在进入这家单位时签订了六年的劳动合同。在这样的小企业干上六年意味着什么？！当时与家人打电话告知这一状况时，我都哭出来了（就在现在杨公堤与虎跑路交叉的、现早已不存在的一个电话亭里，记忆犹新呀！）。

尽管前途是那样的渺茫，但带有“优秀毕业生光环”的我仍坚信自己能做得比别人更好，因为有我的职场第二感悟：自信能让你与众不同，尽管有时的自信有点莫名其妙。在这个企业一开始的工作职责是电站设备的电气设计工程师，需要用 AutoCAD（到单位后学的）

设计电气图纸，并指导工人最终完成电气设备装配及调试。期间，企业经营范围扩大，需要从事电子设备的生产，因此我开始有机会接触电子技术方面的设计工作。在兄弟单位一同事的帮助下，在一个星期内我掌握了如何用 Tango（后来更名为 Protel，现在的名称是 Altium Designer）进行原理图和 PCB 线路板设计。而且，这一个星期的设计结果最终成为了电气产品的一个部件。对于一个毕业不到一年的我来说，这是不小的进步。那时知道了什么是网络表、过孔、焊盘等，掌握了很多电子原件的工作原理（有的还自己用面包板做实验），明白了做电路板的大致业务流程，还能动手焊接电路板，熟练运用示波器和万用表进行调试。那段时间，我对电子技术的兴趣帮上了大忙，学习起来远比别人快。当我精通电路原理，能自如运用示波器和万用表调试电子产品时，别人却还不明白我的调试动机。我的职场第三感悟：兴趣是学习效率的催化剂，培养自己的职业兴趣。

第一次真正对编程感兴趣是从知道 PLC（Programming Logic Controller）开始的。当时的电站设备采用了三菱的 PLC，为了配合这一电气产品的需要，企业社招了一名懂 PLC 编程的工程师。由于老板担心我们相互学技术而“翅膀变硬”，所以明确提出工程师所掌握的技能不能互通有无。当时看到这位兄弟能通过“梯形图”改变 PLC 的行为，真是觉得他太神奇了，仰慕不已。后来通过这位兄弟的私下帮助，我晚上偷偷地在厂房里面学习 PLC 编程。为了获得良好的学习效果，我设定了对电气产品的 PLC 程序进行重写的目标，且最终达成了这一目标（当然，由于这个目标不能让老板知道，所以我的 PLC 程序不能用于商用）。我的职场第四感悟：学习应给自己设置虚拟的项目目标，以做项目的形式提升学习效果，只有这样学到的内容才会深入而实用，切忌无目标地学到哪算哪。

一年多的功夫，我成为了某电气产品的技术负责人，对整个产品的所有技术细节都了如指掌，我带领了其他几个工程师实现了该产品的“自主研发”。有趣的一件事是，老板当时并不知道我已经“翅膀硬了”，想抵赖答应过的 8000 元项目奖金，年轻气盛的我在与之拍完桌子之后对其他工程师下令：“没有我的允许，谁也不能将电气图纸和电路原理图用于生产”。对抗的结果以老板兑现承诺而告终。这时我隐约地有了我的职场第五感悟：话语权首先来自能力，而不是职位权力。

我那时还学会了 CRC 算法并将之运用于 PLC 的串口通讯中，由于对计算机如何通过串口与 PLC 通讯获得采集数据存在很大的好奇心，所以想到了学习编程语言，并计划做一个能在计算机上实时显示 PLC 所采集数据的软件。在向负责 PLC 编程的兄弟表达了这一想法后，他给我的建议是：学习 C 语言比较难，Basic 语言则更容易。于是，我毫不犹豫地选择了自学 C 语言，因为我深信我的职场第六感悟：难学的技能一旦掌握更具竞争优势。

也正是从那时开始，我真正开始了成为软件工程师的自学之路。那时比较幸运的是，单位专为我配备了工作电脑，所以具备了自学的硬件条件。由于那时 Internet 还不普及，学习书籍都来自浙江大学的科海书店（后来眼见着它的店面越来越小，这也是进入电子商务时代的一个缩影），那时隔三叉五地到科海去找书，生活最大的花费就在于购书（那时这方面的书不少是质次价高）。当然，学习的过程或多或少还得瞒着老板。那段时间，别人午休我就编程，除了看书和做书后的习题，还一直朝实现自己的计算机监控软件这个目标迈进（参见我的职场第四感悟）。终于有一天，我用 Turbo C 在 DOS 环境下实现了具有串口通讯功能的、基于图形界面的监控软件（如果你用现在的眼光看那个软件，一定会说“很土”）。当我乐此不疲地向他人演示时，你可以想象我那时有多高兴和自豪！这种小小的成功助长了我的信心，也让我得到了我的职场第七感悟：用阶段性成果不断增强自己的自信，但最终支持自信的是能力，而不是自大。尝到了成功甜头的我随后拓展了自己软件开发方面的学习内容。那时的我已经下定决心要向软件开发方向发展，这种选择是因为我的职场第八感悟：做自己喜欢的事，如果那是自己的兴趣最好。

1999 年的某月，在企业拖欠了一个月工资的情形下，“蓄谋”逃离企业束缚的我们（共 19 个工程师）经过几个月的劳动仲裁后，与企业解除了劳动合同。在离开这家民企的第二天，1999 年 11 月的某天，我在浙江大立机电技术开发公司（即现在的大立科技。后面都简

称为大立公司）找到了第一份专职的软件开发工作。我逃离束缚后能很快地找到新的支点，完全得感谢我的职场第九感悟：不论身处多么困难的环境，即使觉得前途渺茫，也不要放弃学习，否则就是“自断筋脉”。

在大立公司所参与的第一个软件项目，是使用 Visual C++从事 Windows 某变电站图像监控桌面软件的开发。尽管我之前自学过 C++语言，但那时并未完全掌握面向对象编程，尤其是其中的多态。我在该桌面软件中借鉴微软的示例软件 DrawCli，独立地实现了电子地图功能。正是通过掌握这个示例软件的设计与实现，我真正领悟到了面向对象设计的好处。也通过该图像监控桌面软件的开发经历，掌握了 Windows VxD 驱动开发、socket 通讯、多线程编程、图像处理（锐化、伪彩处理、图像字符识别和图像对比等）、ODBC 数据库编程（用的是 SQL Server）等。

在妻子进入大立公司不久，由我担纲了新版图像监控软件的重新开发，这是我第一次担任软件项目负责人。在这个项目上，我可以尽情发挥，将我在老版本软件上所看到的设计不足完全克服。也正是通过这个软件项目，我的面向对象编程能力有了很大的提高，而且完整地做过了一个软件产品。用我现在的眼光来看：那时的开发工作除了引入了版本控制软件外，是不折不扣的作坊式软件开发；至于管理技能的提高，也可以说是微乎其微。

2000 年底，大立公司因为业务拓展的需要，需开发嵌入式图像监控系统（系统中的前端产品是后来数字硬盘录象机的前身）。为此，公司社招了一位比我年长十岁的资深硬件开发工程师，他在进公司时已经有基于 AMD 的 Elan SC520 x86 嵌入式微控制器的硬件开发经验。他在进公司之初与章总交谈时指出：“做这类嵌入式产品，需要软件功底非常强的人”，章总的回答是：“你放心好了，我一定找一个最好的人与你搭档”（这是章总后来告诉我的）。是的，所找的那个人就是我！而其实那时我只有用 Visual C++从事 Windows 桌面软件的开发经验，可见公司领导对我能力之信任！我的职场第十一感悟：机遇很重要，但你得有能力才能抓住它。

我当时所面临的技术挑战，读者可以想象。要知道，在 2000 年时基于 x86 微控制器的嵌入式系统的开发人员国内还很少。我的自学能力、电子爱好的兴趣在这种挑战面前又帮了大忙。其实，做嵌入式系统开发最主要的是参考各种资料以便掌握各类技术细节，这得通过大量地阅读芯片手册、用户手册，以及研究 AMD 在其官网上所提供的示例程序。在这个过程中，就技术困惑坚持探究和养成各种好的工作习惯（思考习惯、笔记习惯、总结习惯、阅读习惯）非常重要。我的职场第十二感悟：职场首先比拼的不是智商，而是坚持与好习惯。

我独自完成了该嵌入式前端产品上的软件开发工作。其中包含的大致技术内容有：从编程的角度精通 x86 处理器架构； PCI、IDE 硬盘、网卡、串口、闪存等总线或外设的驱动；实时操作系统内核的移植工作；MINUX 操作系统的文件系统的移植； XINU 操作系统的 TCP/IP 协议栈的移植工作。移植工作往往会碰到各种技术细节问题，等移植工作完成，对被移植模块的实现和背后的原理也已了如指掌。正应如此，这一时期的工作让我对操作系统的实现原理有了很深的理解。

除了软件方面的进步，我在大立公司时硬件知识也得到了很强扩充。不仅能轻松地阅读数字电路原理图，还自学了 VHDL 语言，使得拿到逻辑器件 CPLD 的 VHDL 程序就能调试软件（通过 VHDL 程序，可以了解编程所需的译码端口、相关信号的操作时序等）。还学会了如何使用逻辑分析仪辅助软件调试工作。前面提到的这位兄长式硬件工程师调侃我说：“你让我看到了中国软件的希望！”，而我将这话当成了对自己的鼓励。另外，这期间还考入了浙江大学专升本的通讯工程专业，给自己充电（2001 年入学，2004 年毕业，获多学期“优秀学生”和“优秀毕业设计”）。

由于大立公司是浙江省测试技术研究所的子公司，它或多或少带有事业单位的气息。

加上公司的技术舞台有限，以及妻子也在同一家公司工作，我于 2003 年 4 月份左右离开了大立公司。在我离开之前，浙江省科委已批复了公司的申请，分配给我一套福利房。在我离开之时，房子仍在建，不少同事对于我的离职很是不解，也劝我拿到房再走。但我有我的职场第十三感悟：当短期利益与长远利益无法得兼时，选择长远利益。

在大立公司工作期间，很希望自己能入职 UTStarcom 这样的通讯企业（那时的 UTStarcom 是多么地辉煌！）。计划离开大立公司之际，我向 UTStarcom 提交了求职简历。这次求职开始好像很顺利，但我真正入职 UTStarcom 的过程却很是曲折。

一开始当我收到 UTStartcom 的面试通知时，可能太希望能进入这个公司了，在没有很深入了解这个岗位的前提下，就去面试了，且马上拿到了 Offer。但后来才了解到，我拿到的是生产部测试开发岗位，与实际研发部门是有区别的。 当时很纠结 — 这是我想进的公司，但却不是我想要的岗位。如果拒绝生产部的 Offer，我很有可能与 UTStarcom 无缘。考虑再三，我还是选择了拒绝（参见我的职场第十三感悟），并重新向研发部门投了简历。

经过度日如年的一个多月等待(那会儿刚好发生了 SARS 疫情),在觉得入职 UTStarcom 研发部门无望的情况下，我入职了另外一家小公司。令人意外的是，在入职那家公司的第二天，我收到了 UTStarcom 研发部门的面试通知。在 HR 面试的那一轮中，HR 对我说：“你是我所面试的人中最有工作激情的”。那时的技术面试官中，其中一位是我日后入职后的上司 — 夏青（现在是恒生电子通讯事业部的总经理），他是我的伯乐。由于我的学历问题，在技术面试通过后，别人只要一位 VP 面试通过就行，我却需要两位。我的职场第十四感悟：学历是很重要的敲门砖，即便你的能力很强；学历尽管很重要，但能力才是最终的通行证。

2003 年 6 月份左右，我正式入职 UTStarcom 研发部，从事小灵通基站控制器（后面简称为基站控制器）的软件开发工作，也从此踏入通讯行业。在入职之初，由于自认为对于操作系统的原理很精通，又完整地做过软件项目，有点飘飘然，觉得自己是个“小牛牛”。然而，入职后一接触工作就发现，内容没有想象的那么简单！

首先，基站控制器的软件规模比我以前主导开发的项目要大很多，而且需要熟悉通讯行业的相关信令。其次，尽管我那时精通 x86 处理器，基站控制器用的却是 PowerPC 8250，这意味着我得重新掌握它。再次，实时操作系统用的是前美国军方的、开源的 RTEMS，那是我第一次接触这个系统。最后，UTStarcom 的工作语言是英语，写文档和邮件都得用英语。尽管我那时能无障碍地阅读 MSDN 和各类芯片手册，但要着手写，却是一大挑战（口语不作要求，因为不需直接接触老外）。

一入职所分配的工作是网元网管部分告警抑制软件模块的开发。尽管 PowerPC 处理器和 RTEMS 操作系统技术细节的掌握与否并不影响日常开发工作，但我仍将掌握它们作为自己的努力目标，这是我的职场第十五感悟：技术细节掌握得越深，解决问题时就越能游刃有余。

那时工作时间应付日常开发工作，业余时间则先将精力集中放在熟读 PowerPC 8250 处理器相关的技术手册上（晚上还得上夜大）。加起来超过 2000 页的英文资料，我读了不少于 3 遍。随着时间的推移，当我对 PowerPC 8250 处理器很有感觉之后，我将工作重点转移到了熟悉 RTEMS 操作系统的实现细节上。先处理器后操作系统的学习安排，是基于我以往在 x86 处理器上的工作经验而得出的，也是因为我的职场第十六感悟：技能的发展应采取深度先于广度且交替进行的方式，只有这样，面对大量的新知识才能更淡定。

RTEMS 是一个类 UNIX 的实时操作系统，也正因为接触这个操作系统我才意识到了自己在软件设计能力上存在很大的提升空间。尽管我对操作系统的实现原理胸有成竹，但却无力于构建一个象 RTEMS 那样的操作系统，也真切地体会到了 RTEMS 的设计之美。那时基站控制器上运行的 RTEMS 操作系统是由美国的新泽西研发中心移植好的，杭州研发中心只需在之上做应用开发。为了就 RTEMS 操作系统获得更好的学习效果，我又一次运用了我的

职场第四感悟，设定了自己完成 RTEMS 新版本移植这一目标。

RTEMS 新版本的移植工作虽不在公司的日常工作范围内，但却得到了上司的支持。由于那时 RTEMS 还在开发新的功能，并不是很稳定，在移植过程中碰到各种奇怪的问题，有些问题还与 GNU 的 binutils 工具集有关（binutils 中包括 nm、ld、objdump 等工具。RTEMS 是用 GCC 编译的）。在无法确认是 GNU 工具集的问题之前，我甚至还向 Wind River 公司（其知名产品是 VxWorks 实时操作系统）寻求过帮助，因为那时用的是它的 JTAG 仿真器。移植工作虽曲折，但最终还是成功了（我所移植的版本并没有运用到产品中，后来的同事又做过了 RTEMS4.6.0pre4 的移植，且运用于产品中）。这一移植经历，让我对 GNU 的 binutils、RTEMS 操作系统的实现有了更为深入地掌握。

在 UTStarcom 工作的前期，我大多从事的是 RTEMS 操作系统相关的代码维护工作，工作内容除了 OS 内核，还包括 FTP、Telnet 等协议。直到中期转为做 E-Box 产品的互联网接入模块的开发工作。

E-Box 是一个企业级电话交换产品，其中还存在一块基于 ADSL 的互联网接入数据板（与现在的 ADSL 猫功能一样），用于实现企业网对互联网的数据接入功能，这一数据板使用的是 VxWorks5.5.0 实时操作系统（PNE 2.0），处理器是 Intel 的 XScale IXP425。那时 VxWorks 的 IP 协议栈还是基于 BSD 的，但 Wind River 对之做了一定增强。这段时期我的工作重点全在 IP 协议栈上（《TCP/IP 详解》这套书帮上了大忙）。这一时期的开发经历，让我对 PNE 的 Bridge、FastPath、MUX、PPPoE 协议、Radix 路由算法和 VLAN 协议很熟悉，也学会了用 SmartBit 仪器和 Chariot 软件做网络性能测试。总之，让我在 IP（v4）协议栈方面的知识上和软件实现上有了长足的进步。

E-Box 产品数据板上的开发工作进行了半年后，管理层决定放弃，于是我被调到了 E-Box 产品的软件平台组。那时平台组刚好面临一个比较麻烦的问题 --- 在命令行上运行 reboot 命令后，有时会出现整个系统挂起，而不是期望的重启。平台组的同事花了一个多星期的时间仍没有解决这一问题。

进入平台组之际，同样是在没有任何人安排的情况下，我自己主动承担解决 reboot 命令功能异常的工作。在我的职业生涯中，我一直热衷于去解决别人难以解决的技术问题，这是因为我的职场第十七感悟：越难的技术问题，其所蕴藏的知识越丰富，也越具学习价值。经过一天半的时间，问题被解决了。其根源在于，reboot 之前没有禁用 CPM 协处理器。我能那么快地解决这一问题，完全是因为之前熟读过 PowerPC8250 处理器的资料。

我在 UTStarcom 工作的后期，致力于 ACE 在 E-Box 产品中的一些应用，借助 ACE 的网络通信功能帮助实现在 Windows 平台上通过 Visual Studio 调试 E-Box 产品。我在《专业嵌入式软件开发》一书的《可开发性设计，一种高效且经济的开发模式》一章中所阐述的内容其实就是这一工作经历的总结与延伸。

另外，我还在 E-Box 产品上做过难度比较大的一个特性是，利用 PowerPC 8250 的 MMU 功能在 VxWorks 操作系统上实现了对任务栈的保护 --- 当一个任务被调度而处于运行状态时，它的栈就处于可读写状态，而其他任务的栈全处于只读状态（VxWorks5.5.0 内核中，还没有 RealTime Process 的概念，这一概念是从 6.0 开始有的，所以那时我所做的这一特性很具实用性）。通过这一特性，可以有效地防止任务栈被意外篡改（比如野指针操作），即便出现篡改也能尽早发现根源。这个功能的实现过程需要调试 VxWorks 内核，那时 VxWorks 的源码虽对公司提供，但 Wind River 公司对所提供的 GNU 的 binutils 做了特殊处理，使得无法为内核代码生成调试所需的信息，结果是无法对内核进行源码级程序调试。由于我之前的 RTEMS 操作系统移植经历让我对 binutils 非常熟悉，通过使用一定的方法（说来话长了）绕过了 Wind River 公司所设置的障碍，成功地实现了对 VxWorks 的源码级程序调试。

在职场中，我不时能成功解决复杂问题和克服技术障碍。这与我的职场第十八感悟是

分不开的：每次积累的点滴知识，一定会在将来不知不觉地发挥效能。

2006 年 4 月份左右，我离开了 UTStarcom。在 UTStarcom 所学到的，不只是前面所介绍的那些技术知识，更让我知道了软件开发的“正规军”是怎样的，与小公司相比，UTStarcom 的软件开发流程要正规得多；也经历了英文写作的“挤牙膏”时期过渡到轻松时期（好友周海东在我的英语学习中帮了不少忙）；看到了好友于善成如何通过大量阅读成为一个知识渊博的人（他的阅读量现在仍是我的学习榜样）；还有上司夏青的技术敏感度到现在仍让我为之称道，是我职场至今所见过的二位具有良好技术敏感度的技术管理者之一（另一位是我在 Motorola 工作期间认识的，后面会谈到他）；团队实力之强使得开发出的 E-Box 产品在我离开 UTStarcom 后不时能听到正面的评价。

说到这里有补充一点，我在大立公司工作时期，就很注重软件设计文档的编写，而且在我离开之时，不仅完善了所有文档，还为后继同事做了全面的培训。我始终坚守我的职场第十九感悟：通过文档化的方式传承知识给后继者是你的基本责任，因为你作为后继者时也希望如此，这也是对自己负责的一种表现。在 UTStarcom 工作期间，我进一步形成了将自己的技术想法写成文章与大家分享的习惯（那时同事贺旭东称我为“作家”，而我则称他为“点评家”），也因为自己在嵌入式软件开发技术上的长期点滴积累，开始有了写书的想法。

离开 UTStarcom 后，我入职了杭州华数集团旗下的雷科通技术（杭州）有限公司。公司当时的意向是安排我负责某宽带接入产品的软件开发工作。在这个公司，尽管只有两个月的时间但也做了些事。除了一个月内完成了宽带接入产品以太网交换芯片在 VxWorks 操作系统上的驱动开发，并使得产品支持 VLAN 功能外，还解决了好几个影响整个产品系统稳定性的严重遗留缺陷。这两个月的工作不光让我在技术团队中很快地树立了自己的威望，也使得公司高层管理者真切地看到了我的能力而在我提出离开时极力地挽留。这短暂两个月的工作经历带给我职场第二十感悟：别人对你价值的认可，其实不是简单地根据你的自身能力，而是根据你对他人和团队的贡献。

入职 2006 年初在杭州成立的 Motorola 研发中心的故事得从面试开始。在入职雷科通不久，我收到了猎头的电话，虽然那时并没有换工作的想法，但也没有拒绝猎头投简历。随后我收到了 Motorola 的面试电话。那次面试过程记得很清楚，因为那是我所经历的第一次英语口语技术面试。虽然工作中从没有锻炼过英语口语，好在对于自己做过的技术知识很熟悉，也经常需要查阅英文资料，所以对于所做过的内容还能用英语勉强解释清楚。在面试的最后，我对印裔技术面试官说，“现在我的英语口语不好，但我相信只要有合适的环境，能很快地提高”。印裔技术面试官最后将我领到 HR 那，说了一声“Yes” — 我的技术面试通过了！

面试结束的第二天，收到了 Motorola HR 的电话，告知 Offer 的相关信息（我的入职级别是 E09，E09 及以上的人在整个 Motorola 杭州研发中心占比大约为 10%）。那时由于并没有换工作的想法，所以拒绝了 Offer。想法很简单，因为曾在 UTStarcom 这样的公司呆过了，所以对外企的工作并不是很向往，反而认为在雷科通这种小公司更能施展。在我拒绝了 Motorola 的 Offer 后，我将这件事告诉了身边的同事，他们的反馈几乎都是“你应当去 Motorola”。

幸运的是，另一名 HR 再一次致电给我，试图说服我加入 Motorola。她当时说“你一旦加入 Motorola，以后离开时所看到的就是 HP 或 IBM 这样的大公司”，也正是这句话打动了我。之后的经历证明，加入 Motorola 是很正确的一个选择！

2006 年 7 月 6 日，我正式入职 Motorola 杭州研发中心。加入的初期是大量的内部培训，培训内容包括技术方面的、流程方面的和英语。Motorola 有着成熟的企业文化，通过培训可以让工程师很快地融入企业，使人行事象是 Motorolan（摩托罗拉人）。在经历了约半年的培训和学习后，2006 年底，我开始参与 WiMAX 产品线上的 CLA 中间件软件项目。

尽管我在 CLA 项目上没有具体的工作（比如，没有缺陷修复工作会分配给我，也没有新的特性开发工作会挂在我的名下），但对整个团队所从事的技术工作都得负责。我的日常工作主要是设计方案评审、代码审查、帮助或带领团队解决技术难题等。

在 CLA 项目上工作了一个月左右，2007 年春节之后，我被第一位派到 Motorola 的芝加哥研发中心做为期二个月的现场技术支持。之前尽管在公司有过英语培训，但要很好地听与说还是存在很大的障碍，加上芝加哥那边一起工作的是口音较重的印度人和巴基斯坦人，挑战可以想象。在芝加哥研发中心除了做现场技术支持，还得为后续人员的到来做铺垫。比如，租好房子、车子，准备好生活所需的一些家当（当时因为预算有限，我们住的是公寓，还得自己烧饭）。那段时间虽然因为语言的问题倍感压力，但在全英文的环境中，我的听说能力进步也明显。之后差不多每年一次的出国，见到以前认识的外国同事，总会有人对我说“Your English is getting better”。对于自认为英语听说能力不行的同仁，请记住我的职场第二十一感悟：英语的听说能力只要有合适的环境，并勇于张嘴练习的情况下能快速地提高，不必担心。

CLA 软件在技术上属于运行于 Linux 操作系统上的一个中间件，它存在多个进程用于帮助通讯设备网元（包括 WiMAX 基站和接入网关）实现网管功能。由于软件架构的特点，使得 CLA 团队不时会碰到由于其他团队没有用好 CLA 而产生的技术问题，这类问题开始大多难以定位是属于 CLA 的、还是不属于 CLA 的，因而查错过程很低效。在 CLA 项目的后期，我希望通过引入新的软件设计方案帮助团队提高软件的查错能力，并改善软件质量。引入新设计需要增加很多代码，如何让管理层不担心由此而引入更多的缺陷是我着力这事时首先要考虑和解决的问题。

在这种背景下，我在 CLA 项目引入了单元测试，寄希望于通过单元测试提高新增代码的质量，以使管理层更具信心而获得他们强有力的支持。最终结果表明，在新增了近一万行代码的情况下，代码在最终发布后总共只发现了一个软件缺陷。这个项目上的工作经历让我第一次真正尝到了单元测试的甜头，在《专业嵌入式软件开发》一书中，单元测试方面的内容很多源于我在这一项目上的成功经验。我在 CLA 上新增设计中的 AED（Abnormal Exiting Detection）功能，在我离开 CLA 项目之后，还帮助团队发现了很隐蔽的多线程问题。当通过 AED 功能发现这一问题的同事高兴地跑过来对我说这个功能管用时，我的高兴劲写满了整张脸。这个项目的经历，也让我更加坚信我的职场第二十二感悟：在软件开发活动中，应设法通过有效的技术途径去解决工程困境。

2009 年初，Motorola 杭州研发中心迎来了一个重量级项目 --- WiMAX 产品线的接入网关 ASN-GW，我被安排到该项目，角色是软件开发架构师。初期我的架构师一职只是杭州研发中心单方面的角色安排，而非全球性的（当时该产品由美国、印度和中国三个研发中心共同参与）。

在 ASN-GW 项目上与我一同共事的经理，是曾在 Motorola 美国研发中心呆了近十年、后来临时转到国内来工作的华人李亮（后面简称亮，习惯了）。他之前在美国工作时做过架构师、软件发布经理（Release Manager）等职，是一个对技术很有敏感度的管理者（我前面提到过的两位有技术敏感度的管理者之一）。我在此之后的成长，完全离不开他的支持与信任，以及他为我所创造的职场发展环境，能与他共事让我倍感荣幸和感激。

我从亮身上学到的第一个内容是如何与美国管理层打交道。总体说来，Motorola 在软件开发管理方面很是四平八稳，其管理存在两大特色，一是争夺项目的所有权（Ownership），另一个是质疑（Challenge）。前者使得各团队职责清晰，不容易出现突发问题或状况找不到负责人；后者使得团队在工作中有所作为，不至于让人浑水摸鱼。在面对美国团队的质疑时，我以前看到的大多管理者都很紧张，总想一味地达到美国方面的要求，但亮在这方面的表现却明显不同。他告诉我们（包括 Team Leader）：“如果美国提的要求不合理，直接与他们‘掰’”。后来我认识到，美国方面做事其实很讲逻辑，只要我们对于他们所质疑的问题能

给出合理的解释，很多异常事件根本就没什么大不了。我的职场第二十三感悟：不要用沉默的方式一味地迎合别人的要求，据理力争或许才是作为的表现。

参与 ASN-GW 的呼叫处理子系统的开发工作后，整个团队经历了大约半年的成长痛苦。痛苦有几个根源，一是对 WiMAX 无线接入技术相关的国际标准不熟悉，另外则是对 ASN-GW 产品的现有实现不了解，而且产品的复杂度的确很大（其中一个技术指标是：必须达到 99.999%的容错能力）。在半年的痛苦期中，我很重要的一个工作职责是帮助团队成长，作为亮这类管理层与基层工程师间的桥梁。比如，为团队起草《开发者指南》和《测试指南》这样的文档，且要求和引导工程师通过文档化的形式沉淀经验与教训，以便提高工作效率（虽然文档化方法的实施过程需要我不断地提醒，但这一方法被证明在这种时期很有效）；我也会在例会上毫不留情地指出工程师的哪些行为影响了工作效率。我的职场第二十四感悟：流程、文档的作用，不只是引导我们做完事，更能规范我们的行为和帮助培养工作习惯。

亮在项目进展的过程中，一直向美国方面主张杭州团队必须设置架构师一职，也正是由于亮的一再争取，美国方面最终努力地帮助我向这个方向发展，不断为我分派属于架构师工作的任务（如更新产品架构模型、参与需求管理、参与系统设计文档的评审、完成新特性开发工作评估等）。亮那时告诉我，我应是杭州研发中心第一个真正从事架构师工作的人。

刚接触架构师方面的工作时，其实还是不大自信的，尽管我那时掌握了软件架构师所需的基础技术技能（比如，我的软件设计能力很强、UML 从 1998 年开始接触加上之后的持续学习所以功底也很好），但对于软件研发管理方面的内容，以及 WiMAX 无线接入技术知识的系统性认识还是相对单薄的。那时与美国同事接触下来的感觉是，他们的综合能力都很强，似乎随便一个人都知道如何做架构师，不少人有做 GSM、iDen 和 CDMA 产品的经验，而且长期工作于无线接入技术领域。随着更多地参与架构师方面的工作，不仅逐渐建立了自信，对 Motorola 的软件研发管理也有了更为深入地认识与理解。所看到的不仅仅是产品技术本身的复杂度，更有开发活动运作管理方面的复杂度。最终，我成为了整个 ASN-GW 产品的架构师。

在 2009 年，我考入了浙江大学的 MBA，同时还开始着手写自己的处女作《专业嵌入式软件开发》。在之后长达近两年的工作、学习和写作的三重压力下，我在时间管理上有很大的进步，抗压能力也得到了很好的锻炼，这时我的职场第十二感悟（指其中的坚持）又让我最终渡过了这段最为艰难的时期。（注：《专业嵌入式软件开发》一书其实不只专注于嵌入式，其中绝大部分内容是 C/C++开发人员应当掌握的。当时书名中采用“嵌入式”三个字完全是因为给书定位的需要，害怕书名不具体而使人难以选书。当然，也正因为“嵌入式”三个字，使人觉得面太窄了。有利有弊吧！该书在各大网上书店都归类于“软件工程及软件方法学”，而非“嵌入式系统”）

2010 年中期，NSN 宣布收购我所在的 Motorola 网络部门，收购活动直到 2011 年的 4 月份才结束。同时由于 WiMAX 市场的不景气，美国不少系统架构师转到了 FDD-LTE 产品线上，我也因为这一缘故担任了大约半年的系统架构师，主要负责 WiMAX 技术的移动性与网络安全方面的工作。

2012 年 7 月份，因为 WiMAX 产品线裁员，我转到了 NSN 的 WCDMA 产品线。也从此开始离开了 Motorola 的研发管理环境，而真正步入了 NSN 的研发管理环境。

真感谢你花时间读到这！尽管我们常将“职业规划”挂在嘴边，实际上职场发展真的是一种“布朗运动”。你不知道下一站会是哪、也不知道后面将要从事什么工作、更不清楚后面会碰到怎样的老板。在众多不确定因素面前，或许参照我一路走来所总结出的职场感悟能让你不断地朝好的方向发展。

8.4.2 感悟程序人生

上面的文字比较长，其中记录了李云成长过程中的各种细节。其中非常让人感动的是李云的这种永不放弃的精神，从文字中我们可以看到李云从来没有停止自己前进的脚步。这是非常可贵的，李云的成绩中几乎没有运气成分，完全的靠自己的双手把握自己的人生。

我们还是回到之前一直提到的几个维度：

价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

在这一公式下，你会发现李云道路的主旋律就是在一个技术路径长的领域里增值、增值再增值。不论是最初的 PLC 还是后来的 WiMax 都可以比较笼统的归到一个叫软硬结合的软件领域，这个领域技术路径比较长，和基于 ASP.net 做信息管理系统完全不是一个难度。从事这类工作其知识必须贯通软硬件，否则寸步难行。一个一直做纯软件的人，在这类领域中听都听不懂相关工程师在说什么。今天之所以很多程序员可以缺乏硬件知识也能做软件开发，关键在于 Window 送，Java 这类平台屏蔽了大量细节，而做上述这类工作时等价于需要撕掉这种屏蔽，重新面对软件最本来的面目。所以这天生是个技术路径长的领域。

在这样的领域中达到一定高度后，稀缺性会很自然的呈现出来。，与此同时，公司平台的切换也使李云所站的位置越来越高，稀缺性越来越好。

李云的经历同样也说明了人生确实需要一种永动的势能，否则一旦他安于现状比如停留在大立公司，停止了自己在技术上的追求，那么也不会有现在的高度，而是会在某个环境下上过着相对比较稳定的生活。

在李云身上表达力起到了一定的作用，但作用暂时并不明显，技术的不断提高和积极的精神使李云并没有碰到这方面上的瓶颈。

如果与第一个故事里的主人公老 A 相对比就可以发现更多的事情。两者年纪相差不大，但却走出了截然相反的人生道路。同北大、清华、浙大这类顶级学校相比，李云的学校实在是差的一塌糊涂，如果在九几年两个人刚毕业的时候让企业进行选择，李云绝对是一点机会都没有，夸张点讲李云根本就不会有和老 A 并列在一起的机会。

但在 2011 年，经过十几年的奋斗之后，形式完全逆转，老 A 不再有和李云并列在一起的机会了。李云可以在各大公司间选择比较适合自己的工作，而老 A 只能被外包公司被动选择。

形象点讲，老 A 出身名门，但人生一路下滑。李云毕业于专科学校，但靠自己的努力创造了命运。

也许有刁钻的人会说，李云这样有什么好，马云什么技术都不懂，阿里系一样千亿市值。这么想就危险了，所谓一念天堂，一念地狱即是如此。恰如人不能寄希望于中彩票一样，对于人生而言，理想可以远大，但手里要有看得见的适合自己的路径。崇拜马云先生可以，但马云先生自有其自身的艰辛和机缘，而大多时候这类成功完全不可复制。不具体了解这些，而单纯因为虚无缥缈的远大，而荒了现在，那就是好高骛远。

§8.5 一个创业者的十年

我一直很犹豫要不要在这样一本书里去讲创业的故事，因为我一直相信创业是成功率极低的一项活动，只不过是成功者的光环太盛，才使这项活动吸引了过多的关注。

不是创业不好，而是说输不起的人不适合创业，而很不幸大部分人其实是输不起。

我们必须承认即使在物质基础非常薄弱的时候，人也可以有很高尚的理想，但大多时

候理想往往植根于现实，也需要植根于现实。

好比说，一群人从悬崖上飞跃而下去体验飞翔的快感，这诚然没错，但有的人可以有很好的防护措施来保证不会摔死，但有的人却很可能会摔的脑浆崩裂。

后者过于惨烈了，我不想为之摇旗呐喊。

但最终还是决定放一个创业的故事在这里，但我会对其进行“负面”的解读，更多的去强调什么情形下不要去创业。

8.5.1 戴志康和他的 Discuz!

很多人都知道戴志康的故事，下面的文字是根据公开报道整理而成，比如：CCTV 创业故事的专访，2006 年戴志康做客新浪时的自述等。也正是由于公开报道很多，下面只是对这个故事的最关键点做一点陈述，并不会展开细节。

戴志康开发 Discuz!的出发点并非是为了创业，而是找工作。

2000 年，戴志康考入哈尔滨工程大学，从 2001 年开始近 2 年多的时间，戴志康一直沉醉在 Discuz!的开发之中，与此同时，戴志康的成绩变得极为糟糕，挂科 14 门。

2003 年，戴志康准备开始销售 Discuz!，而不是再免费向用户提供。最开始的时候，销量非常惨淡。这其中有一个小插曲是，在毕业前 1 年，就有企业出 30 万年薪来招聘戴志康，从这个角度看，戴志康的原始目的达成了，但他放弃了这样的机会。

2004 年，通过海外销售，戴志康赚到了 30 万。也是在这一年，戴志康创立北京康盛世纪科技有限公司（Comsenz），并被认定为高新技术企业。

2005 年，Discuz!开源免费，转向以服务赚钱。营业额达到 500 万元。

2006~2009 年，公司获得红杉资本等公司的持续注资，用户量持续增加。

2010 年，康盛创想（Comsenz）被腾讯收购。这个时候应该是商业模式上遭遇了一定困境。

8.5.2 感悟程序人生

创业的初期，技术、资金、人脉、主意等等，都可以看成是一个个筹码，创业者所主要要做的就是要把你的筹码转换为持续不断的现金流。随着现金流的不断放大，这个人的创业故事也就越来越成功。

从这个角度看，戴志康的最大筹码就是他的 Discuz!，已经很难看到沉浸了戴志康数年心血的最初的版本到底是什么样子，但恰如戴志康自己所陈述的这个产品一定是比较优势的（比如：速度、安全、负载能力等）否则不可能获得这么大的成功。

但恰恰也是在这一点上，导致事情很难被复制。在起点上戴志康和很多做共享软件的人很相似，唯一的差别只是用户的接受程度。而开发出一款广为接受的软件，不只是需要技术能力，也需要一定的时机和运气。你可能付出的比戴志康还多，但你开发的软件更可能是默默无闻，而不是被很多人接受。未必很多人做不出 WinZIP，但一旦时机不在，做出来也不会有那种辉煌。在创业故事中，技术只是一个较大的筹码，远不是全部。

从初衷和手段来看，戴志康的选择并不正确，单纯为了找工作，写一个软件出来远不

是最有效的手段---很多大公司并不会因为你写过一个软件而招聘你，关注的还是某些基本功是否扎实。而一旦一个人因为某个软件而无法毕业，同时软件又默默无闻的话，那人生可能就会困顿。但从创业的角度看，有一款获得广泛认同的产品绝对是上佳起点，在那个时代里，太多的人重复了这个故事，求伯君的 WPS，王江民的 KV300，鲍岳桥的 UCDOS 等等。

创业的话题一直很火，戴志康这类的故事更是可以激励一大批年青的程序员，使他们有创业的梦想。很多人也在很多场合鼓励年轻人开创自己的事业，但对此我的观点则与主流相反，我认为大多程序员并不适合创业，而更适合走李云这类道路。道理很简单，赌博可以，但要输得起。在考虑创业前，在考虑巨大收益前，要考虑自己是不是付得起创业的基本代价。很大一部分人的经济基础其实是薄弱的，从家庭整体来看个可能是像在负债经营，在这样一种前提下走戴志康这类道路，冒的风险太大了。因为不管怎么说，创业成功始终是低概率的事情。

不是说创业没价值或不对，有基础的人在年轻的时候追逐自己的梦想不管怎样都是对的。关键是对于青春是唯一资本的人，如果把这个资本投资于高风险的事业，那么人生的风险就太大了。其实再别的场合也一样，风险与利益的比例值似乎总是个常数，风险高的事情，收益就大。买创业板的股票，升值空间大但也可能让你赔的一毛钱也不剩下；蓝筹股则与之相反。

我们总提到的公式：

价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

更适合用于评定职场中人，而不适合评定制定规则的人。以戴志康为例，在开发 Discuz! 时他基本上处在持续增值的状态，一旦开始创业，则处在制定规则的状态，上述规则开始不适用，但在并入腾讯后，上述公式则又开始发挥作用。

§8.6 一个女程序员的编程之旅

虽然上面的故事覆盖了程序员的主要出口，但如果缺一个女程序员的故事总感觉像人生没有婚礼一样，缺了点什么。女程序员之所以特别，实在是因为相对稀少。

好多女生似乎是真的不喜欢这个行业，但确实也有战斗力非常彪悍的女码农奋斗在编程第一线。单纯从气氛的角度看，女程序员真的是必要的，即使战斗力不是很彪悍。

8.6.1 女程发贴引起的讨论

假如你在 CSDN 发了这样一个帖子，那会引起什么样的讨论？

三年前毕业于重点大学计算机专业，毕业的时候倒不是因为喜欢，而是当时 IT 环境较好，又是学这个专业的，毕业后就顺理成章的做了程序员。

毕业后，虽然工作的成绩还不错，薪水也还行，但我总觉得女的做这行特别不适合，忙起来披头散发，整天对着电脑、书本，整个一黄脸婆，特别没有女人味，就象没有水的花一样，枯燥、干渴。

没有尽头的工作，让我对未来的担忧也很大，真的不知道自己三，四十岁的时候是怎样的生活？

去年 7 月，我一狠心辞职了，想重新定位一下，找其它性质的工作，但因为没有干过其它的 工作，也不知道自己适合什么，加上没有工作经验，所以，半年后，弹尽粮绝，只有老本行能找到工作，没办法，只好又回到了原来的行业。

同行们的姐妹们，你们目前的状态如何？又有什么打算呢？

其实这是真的，这是 2003 年的一个老帖，各种回复也特别有意思，下面选几个有代表性的：

回复： 趁早改行罢！

回复： 呵呵，我准备再干两年，攒点资本然后开个小店（能挣钱养活自己就行了），让自己闲下来做一些相做的事，比如说旅游，比如说养个 BB。

回复： 常玩电脑的美眉要多吃维生素 A，多喝水，不然皮肤很容易变怀啊。

回复： 我觉得各位女同胞应象软件配置管理、培训方面发展，个人意见。

回复：是啊，这是大家的心病。这世界上出名的人毕竟不多，要么干得很出色，要么就做一个平凡的女生，但我知道这世界上鱼和熊掌不可监得。事业和家庭难道永远是个矛盾，理想和现实亦是一对矛盾。男人全心全意扑在事业，老婆为家庭奉献这种模式倒可理解。如果倒过来，女孩子事业心太强，男人也不可无事业心，这个家庭如何能建设？

回复：我工作快三年了，从入行就想着以后肯定得转行。具体出路在何方还是很渺茫考研 成功的概率不大，现在是漫长的等待。考不上就去考 CPA 希望转向财务方面。

回复：其实，我真的很想说“编程让女人走开！”尽管知道说出来肯定会被骂。好象 CSDN 里的阿婆阿妈阿姐阿妹都到了这里。

我可能入行时间比较短，感觉程序员这个职业好象是一次巨大的赌博，把你的所有身家、青春和精力都放到桌子上。

男人好赌，往往可以不顾一切，但女人有太多割舍不掉的。在赌掉了自己所有的青春后，可能换回来的却太少了。

其实程序员没有性别的分别，没有人会因为你是女的，而放过你的 BUG。

其实，我很希望开发团队里有几个女程序员，女性承受压力的能力比男性要强，而且平时可以让男性们不至于太过邋遢。不过呢，我同意 MYAN 的话，女性最好不要当团队的领导，除了 MYAN 说的对下太苛刻外，还有一点，大局观不好。其实女性比较适合 QA 的工作。

http://bbs.csdn.net/topics/20377173

8.6.2 感悟程序人生

从讨论我们可以看出来，很少有人支持女生在程序员的岗位上好好学习，天天向上奋斗出自己的天空。

我个人从来不认为女生会在天分、才华上输给男生，包括写程序。

区别的产生似乎来源于上帝的安排，娃娃的孕育和抚养往往会牵扯女生太多的精力。娃娃一哭，当爸爸的还会没心没肺的不当回事，当妈妈的往往就会立刻放下手里的事情。人生里时间是有限资源，你这里用多了，那里就少了。而当平均智商相差不大的时候，时间是关键筹码。

职场里一定会有文化，但在成绩贡献面前是不能讲感情的，你错过两年，不管你是因为什么，始终还要和其他人站在一样的起跑线上。

这时候女生往往会输，至少在程序员这个群体里是这样，因为生活里可能更有价值的东西牵扯了她们太多的精力。

人生的价值其实是一种选择，能判定选择对错的往往也只有自己。

假如说一个女生真的选择过不同流俗的程序生涯，我深信她有和男生一样的机会，唯一的关键就是值不值得。

工作与生活的关系

如果把女程序员的话题稍微引申一下，就会发现这背后是无比宏大的一个主题：工作与生活究竟应该是怎么样的一种关系？这又是一个没有也不应该有唯一答案的问题。

人生很多事其实是个可伸缩的模型。

理想重要么？当然很重要，但当饿到一定程度时，对绝大多数人，理想就没有馒头重要。

工作比生活重要么？生活比工作重要么？一这么考虑问题，脑子必然永无宁日。

如果把活着定义为生活，那么工作其实是生活的一部分。一个人最好的时光往往是在工作中度过的。但工作确实和家庭生活有对立的一面。人的时间就那么多，一边花的时间多，一边的时间必然就少。

如果我们相信吃饭是人生最关键的事情，也相信贫贱夫妻百事哀这样的俗语，在没有彻底的财务自由之前，家庭里至少要有一个人要工作优先。

这很功利，但没办法。

当代中国，家庭的基本模式几乎是固定的：两个独生子女组成家庭，一旦有了孩子之后，要么一人不工作，要么从父母双方获得帮助。而随着上一代人的老去，自身的责任必然加重。

在这事儿上大部分人选择权真的不大，必须工作优先，这是一种现实约束。在这个步骤里，家庭要争取自己的选择权，无疑的争取到了财务自由的家庭是幸运的，他们可以重新规划自己的生活模式。这个时候仍然可以选择继续工作，但意义已经不同，并非是因为外部的什么压力才这么做，更主要的是因为自己想这么做才这么做。这是一种值得期望的状态。

从这个角度看，在人生这个舞台上，年轻的时候一个人所扮演的角色是带着面具的，只有财务自由后的演出才是本色出演。

§8.7 观罢六段人生，体验是非成败

《三国演义》开篇的那首《临江仙·滚滚长江东逝水》随着杨洪基先生的演唱是越来越有名了：

滚滚长江东逝水，
浪花淘尽英雄。
是非成败转头空。
青山依旧在，几度夕阳红。
白发渔樵江渚上，
惯看秋月春风。
一壶浊酒喜相逢。
古今多少事，都付笑谈中。

大多数人的人生虽然远不如历史那般波澜壮阔，但对个人而言其间悲欢离合往往也是触目惊心的，是非成败转头空是一点不差。恰如前文一直强调的，在谋求安身立命的时候，我们可以不很成功，但关键是不能很失败。而在一个高频变化的 IT 世界中，对任何人而言，其人生基本趋势往往不进则退，中立状态是不太存在的。

外部不断变化增长的需求和自然规律所带来的精力衰退两者之间蕴含着非常大的矛盾，这就要求在速度放缓之前一个人必须走的足够高，足够远。

前面故事中，L、李云、戴志康的未来尚还可看，老 A 和退场者在 IT 这片世界里却是默默无闻了。

在年轻的时候，手里最大的筹码就是时间，时间用尽却没转换成价值，那么在高频变化下往往会被打回原形。而一旦如此，一个人在晚年怕也是没什么心情去吟唱“青山依旧在，几度夕阳红”的。

足够长的时间尺度下，往往偶然因素的作用会被压缩的很小，更多的体现的则是人生的必然性。而人生的必然性恰恰就在下面这个简单的公式里面：

价值 x 实现程度（表达力，稀缺性，公司平台）= 职场成就

结束语

我个人其实有点宅，喜欢闷在屋子里看死了好几百年人写的书，看的多了就想自己写写。那时候一共构思了两本书，一本是想写给做了很多年软件的人看的，一本是写给要入行或者刚入行的人看的。写第一本的时候是比较狂妄的，那时候我觉得《人月神话》有些过时了，有的地方也挖掘的不够深，应该有本书来取代它的位置，所以使了很大力气来尝试把书写的很有价值，写到我认为能够超越《人月神话》的程度，但等书出来我才明白，能不能超越《人月神话》并不只是书的问题。

写完《完美软件开发：方法与逻辑》，我就着手开始写第二本书，也就是现在大家看到的《程序员生存定律》。写到这程度其实可以讲，这书基本写完了，但这书其实有个小问题，博文的侠少曾经和我提过这事，他说：大家有压力想成长时，那有耐心读你这类有点硬的书呢？我想了下，觉得他说的有道理，但却不太想做什么修改，主要原因是我认为这类书从质量上讲，有道理无疑比容易读更关键，毕竟这东西影响一个人的一生，而如果一个人都不愿意花点时间看看一些有道理的建议，那其实你也不太可能能够改变他，还提什么建议呢，毕竟我本意不是卖书，而是提有价值的建议。这就叫真的 2B 青年其实都是狂妄的，一旦认定自己有道理，特别容易瞎坚持。

《完美软件开发：方法与逻辑》出版后我就想，我也别总宅在家里应该出去和人接触下，于是就去了 InfoQ 的 QCon，南京那边的 TalenCamp 等活动，一圈走下来我发现世界处在巨变之中，做传统软件的如果不睁开眼睛看看这变化，很容易就被淘汰了。我觉得我自己应该尝试做点什么，来实际体验下，结果就是 V 众投（公众号：vzhongtou），出发点也超级简单，我想买东西的时候一般会看看网上怎么说，可是百度哪怕知乎一旦牵涉到利益，它里面的信息就不太可信，无它，水军太厉害了。大家都学小米做社会化营销，这问题其实愈演愈烈。于是就想尝试能不能做个大家愿意用，但产生靠谱信息的地方。这念头一出来，就特别有实现它的冲动，最后把事情简化成做一个只有验证用户才能回复投票，一人一号，一人一票的东西。从头做很麻烦，就选了 WeCenter 做基础，可即使这样也还是挺牵涉时间的，《程序员生存定律》的完善出版也就耽搁了下来，后来再想想既然没空完善，那还不如就那么放出来让大家看看，用心读读的总该能体会到些东西，于是就一边做小修改，一边在博客上发这系列文章。从大家的反馈来看，我感觉有些同学还是学到一些东西的。

《程序员生存定律》这书一个比较特别的地方是如果你不琢磨其实他没啥用，我把影响人生成长的要素乃至他们的关系都列了出来，但这些东西只有在一个人愿意思考并结合自己实际情况的时候才会有用，有的书会写要先做 A，再做 B，再做 C，这本书不是的，他主要是讲决定 A，B，C 是啥的方法，当然有时候为了让这方法容易理解一点，也会写在一定条件下 A，B，C 具体可以是什么。

我比较真诚的希望要入行或者入行不久的人用点心思读读这书，绝对会赚回你读它的时间，我自己一边写也常一边感叹，这事要早点明白就好了。

对于程序员而言，过去这二十年其实是好几代人，求伯君他们是一代，任正非的华为等是一代，马云、马化腾、李彦宏的 BAT 是一代，移动互联网又是一代。之所以说他们是一代是因为他们都分别的经历了自己的高速成长期，对程序员而言赶上没赶上这个高速成长期，结果完全不同。而我碰到的人里面有华为前一百号员工，但早早离开的，有莫名其妙从很牛的外企出来加入了一个叫百度的鸟公司，接下来百度上市，一下子；拉开和辛苦奋斗的群众的距离的。这人生确实还是很奇妙的，而增加抓住这类机会的几率，无疑的要解决好内功和选择的问题，谁能说这问题不重要呢？