Joel Spolsky曾经感叹：招聘难，难于上青天（此处笔者稍加演绎:)）。他有两个辛辣但不乏洞察力的断言：真正的牛人也许一辈子就投大概4次简历，这些家伙一毕业就被好公司抢走了，并且他们的雇主会给他们不赖的待遇，所以他们也不想挪窝。（刚刚去世的Dennis Ritchie就是这样一个人）而“人才”市场上能找到的大多都不是什么人才。招到这帮人轻则费钱重则把你公司搞挂。

（当我把这篇文章给邹欣老师review的时候，他说了另外两点：1. 最好的人也许不投简历，就决定去哪里了。所以要在他们做决定前找到他们。2. 比较差的会投很多次简历，找不到工作的时间越多，投的简历越多，给整个pool 带来很多噪音，top10%的简历也许根本不算全部人的top10%。）

诚然，也许没有哪个行业像IT行业这样，无形资产占据公司的绝大多数资产。拒坊间传言比尔·盖茨就曾经说过类似这样的话：只要允许我带走100个人我可以再造一个微软。这话没搜到原版出处，但是从一个侧面反映了IT公司当中智力资产所占的比例之重。

所以一个自然的推论就是，**招聘也许是一个公司决策当中最最重要的一个环节**。Joel Spolsky把他在这方面的观察，体会和洞见集结成了一本小册子《Smart and Gets Things Done》，开篇就挑战“产品是公司成败的关键”这个传统观念，他认为创造最适合工程师生活的环境，留下最优秀的人才才是最先最重要的一步，接下来好的产品是水到渠成的事情。国内iapp4me.com创始人郝培强正是这个理念，所以他在微博上说：

> 我们是小公司，工资开的不高，也不招太多的人，但是电脑都是iMac27，iMac21，Macbook pro15，基本上比很多大公司都好多了。软件没盗版，刚才photoshop的正版我也收了。中午管饭，公司备伞。哈哈。节日假正常放，从不加班，早晨11点上班，下午6点下班。我是有资格说某些大公司的员工苦逼的。

事实上，米国找个人尚且难成这样，搞得Joel还费心费力写本书语重心长地劝企业们要善待好工程师，国内找个人更是难上加难，国内高质量问答社区知乎创始人周源就曾经在知乎上分享他呕心沥血的招人历程，看完真是让人慨叹这年头找个靠谱的人多不容易（这条知乎问答还有很多精彩的跟帖）：

> 其实从 08 年到现在，我一直想这事能不能有点窍门，或者是实用的方法，结论是几乎没有。我用过的大家都用的方法：
>
> - 在水木上发贴子（有点效果）
> - 在蓝色理想上发贴子（无效）
> - 在技术邮件组里发贴子（无效）

- 买 51job/智联 最便宜的服务（有点效果）
- 给所有可以想到的人打电话，请他们推荐（无效）
- 给所有和你讨论过创业，喝过点小酒的人打电话（无效）
- 约前同事私下谈（有效）

我用过的大家可能没有用的方法：

- 上 twitter，看 XXX 的 follower，一个一个看，看他们的 twitter，博客，Google Reader 分享，想办法搞到邮件，联系，半夜电话骚扰。
- 上豆瓣，前端后端挑几本重量级的书，去找想看，看过，正在看这本书的人，一个一个看，看他们的活动，博客，Google Reader 分享，想办法搞到邮件，联系，半夜电话骚扰。
- 找同事，问他们都看什么技术博客，想办法搞到邮件，联系，半夜电话骚扰。

正是这样的不容易，才有不少公司走内部培养的办法，这里的逻辑是：一上来就招到靠谱的人太难了，但找一块靠谱的璞玉然后雕琢雕琢相对就简单很多。这倒是个办法，但这样做的人难免就陷入了纠结：培养好了，人跑了怎么办。这也不能怪招聘的公司，的确是人之常情。其实解决的办法也很简单，培养的时候进行适当引导，让员工发挥自己的主动学习能力，这样不但人得到更多成长，公司也不会觉得投入太多患得患失。所谓师傅领进门修行在个人。

但是，这仍然还是没有解决根本的问题，就是**招聘真的很困难**。应聘者固然觉得自己是在“海投”，大海捞针一般。而招聘者何尝不也是这种大海捞针的感觉。这就好比两个人谈恋爱，都想和对方好上，但是偏偏就聊不到一块去。

招聘真的很困难。以至于招聘者每年需要绞尽脑汁出新笔试题，以免往年的笔试题早就被人背熟了。出题很费脑子，要出的不太简单也不太难，能够滤掉绝大多数滥竽充数的但又要保证不因题目不公平而滤掉真正有能力的，要考虑审题人的时间成本就只能大多数用选择题，而选择题又是可以猜答案的（极少有人会在选了答案之后还敢在空白的地方写为什么选某答案的原因的）。更悲催的是，有些题目出的连公司的员工们自己都会做错（真的是员工们做错了吗？还是题目本身就出错了？）

笔试完了之后如果还没有被鄙视就要进入面试环节，姑且不说笔试题的种种弊端，就说面试环节，短短几个小时的面试（大多数公司也许连几个小时的面试时间都没有），既需要全面考察基本知识，又要考察编程素养，还要考察（也许最重要的）性格心态。再然后还有一项根本没法考察但却占据程序员相当一部分工作时间的：debug能力。面试官不但得找准问题，不因对方一题答对而妄下结论，也不因一题打错而就扼杀机会，还要以管窥豹，从一朵花看到整个世界，从面试人的举止言谈，分析问题的方式，甚至写程序的笔迹来观察这个人的性格，做事的方式和心

态，简直是要面试官具备心理分析师的水准才行。

这厢要招人的雇主苦不堪言，那边找工作的人也是一团乱麻。绝大多数应届生直到毕业也不清楚他们想要去的公司到底需要什么样的能力，或者说，他们到底需要具备什么样的能力才能在应聘季节拥有自己的选择权。中国虽然本科教育环境差，但是同样有很多的人在本科希望整点东西出来，他们有一腔的激情和抱负，有强大的动力，但就是不知道自己需要掌握哪些技能才能满足雇主的要求，求告无门，整年整年苦闷的像没头苍蝇一样乱撞（我就收到过很多次这样的来信，他们往往很想学点东西，但又不知道哪些重要哪些不重要，到底该学到什么程度，**不知道导致不确定，不确定导致决策瘫痪**，干脆嘛也不动，荒废时间）。

什么叫熟练？什么又叫精通？那么扎实呢？两年的YY经验又意味着什么？能这么简单的量化吗？同样是两年的“实践”有的人能真的学到点东西，有的人也许近似一无所得。那么实习呢？很多人都一定要在简历上弄个实习经验，这个又能说明多少问题呢？大作业呢？得奖呢？有一次我面试一位同学，据简历说编译原理课的大作业得了一等奖，可我一问什么是递归下降，就傻眼了。

这个现实的结果就是，**现在绝大多数应届简历而言，也许最具信息量的部分不是“精通XXX，熟悉YYY，掌握ZZZ”，不是“在UUU实习过”，也不是这个项目那个作业，反倒是越来越被认为不重要的一项：毕业学校**。毕业学校本不应该是最具信息量的，它之所以最具信息量只是源于一个悲剧的事实：简历上其他条目实在信息量太少了。所以靠谱的面试者往往学会了无视简历上华而不实的内容，只相信面试的时候亲眼所见，扫两眼简历也就罢了，最后还得自己捋起袖子慢慢面。而应聘者也许也知道招聘的也不会细细纠简历上的条目，所以什么词也都敢往上捅，反正先过了HR筛简历这关再说。从经济学角度来讲，应聘者的这种策略是正确的，没有代价（因为目前似乎没有公司会去给已经申请过的人做一个诚信数据库），但至少有可能会带来巨大的收益。应聘成了博彩。而博彩式的应聘给招聘公司带来了巨大的筛选压力。简历成了摆设。

那么招聘这个关系里面的第三者——学校——所处的位置呢？学校更关心的是毕业率和就业率，这似乎是件好事，有这个为目标，那么老师们似乎应该努力让自己的学生多学点东西。可惜就业的质量似乎不是最重要的指标，此其一。其二老师本身大多数没有丰富的业界经验，根本不知道企业整整需要的人才是什么样的，可能花了精力，但却培养不出雇主真正需要的人。另一方面，老师所起的作用很多时候甚至是一个负面的作用，例如布置大作业表面上看上去是培养学生的能力，我们姑且不说抄袭，假设每个人都做了，那么大作业本身能够衡量多少东西呢？能否衡量代码质量，能否衡量团队协作能力？能否衡量交流能力？考虑到大作业用到的东西往往都是书里面现成的，**大作业甚至不能衡量学习能力。而学习能力简直算是这个行**

业最重要的能力没有之一了。

所以，简而言之，**如果把人才培养/招聘这件事情本身类比做一个项目，那么这整个项目迄今为止就是一个巨大的失败**。为什么这么说呢：

- 和需求严重脱节：作为人才需求方的雇主的需求到底是什么？绝大多数应聘者都没搞清。更严重的是，这却一点都不是应聘者的错。因为雇主是stakeholder，是雇主自己的责任得去说清楚需求是什么。结果应聘者实现的不是雇主想要的，雇主想要的应聘者没有实现。
- 应聘者雇来培训自己的人根本不管事：学生交了学费，就相当于雇老师来培训自己，可培训者根本也不了解（或不关心）他的客户们的需求。这里，学生是需求方，老师则是实现方。弄清需求的职责在后者，可后者也弄不清。
- 学生自己也弄不清：学生自己既是需求方（需要特定技能），也是实现方。可他们自己也弄不清需求到底是什么。

以上三点还不是最严重的，最严重的在下面：

- 明白需求是什么的也不知道怎么实现：怎么去培养现代IT企业真正需要的人才？特别地，实战能力怎么培养？代码素养怎么培养？协作沟通能力怎么培养？学习能力怎么培养？就算这些都知道怎么培养，又怎么给在象牙塔里头，离催命之日还遥遥无期的学生提供足够的动力呢？而学生自己就算知道该学哪些技能，又怎么知道具体怎么着手？什么是最有效率的学习方法？又如何让自己保持学习的热情？

以上这些问题，就是当下人才培养/招聘的惨淡现状。简而言之，在雇主和学生之间，横梗着一条巨大的鸿沟，两头都很着急，两头都有动力，但就是没有方法，君住长江头妾住长江尾。像微软谷歌这样的，干脆和高校合作，直接插手本科或硕士的教育，从而保证到时有足够强的候选，某种程度上，这的确是根本解决之道，可一来这代价太大了，非一般企业承受得起，二来这影响面也太小了。

这一切，也许将在未来的5年发生根本的变化。

《Switch: How to Change Things When Change Is Hard》（中译《瞬变》）里面指出，表面上看来非常困难的改变，也许是因为根本就没有抓住要害。在书中作者通过大量案例分析和心理学研究，雄辩地指出以下几点促成改变的关键之处：

- 触动内心的大象：要改变的人必须要有情感层面的动力。有一些特定的方法能够比另一些方法更能对人的情感产生触动。
- 给出清晰、明确的目标：目标一定不能含糊，模棱两口的目标让人无所适从，导致决策瘫痪。例如最近我们组在招实习生，我在微博上发了一条招聘信息，其中提到“扎实”的系统底层知识，有同学就写信来问，怎么叫“扎

实”。我傻眼了。比尔·盖茨就以目标清晰明确著称，不仅在战略制定上，“每个人桌面上都有一台PC”，而且居然还体现在招聘上——“如果你读完了TAOCP，那么就给我投简历吧”。多么清晰，明确的目标啊——虽然高了点，也许这就是比尔·盖茨至今还没被应聘邮件淹没的原因:)

- 给前进的道路扫清障碍：人是懒惰的，只要有借口就会不想往前。如果既有明确的目标，同时道路又直直指向目标，一览无余，只等你开始往前走，那么便没有借口，一往无前。

那么让我们对照上面看看，可以做什么?

首先，内心的大象不需要触动，中国有足够多的人足够早就开始焦虑就业的事情，只是不知道往哪使劲，这部分人如果把劲头用到正确的事情上面也许足以满足现在的IT企业人才饥渴了。至于其他人，好吧，也许身边的人开始动起来他们也会被触动。

然后是清晰、明确的目标。这一点上目前雇主们的做法可谓好坏参半，好的一点是大家都强调要有实践经验，要有团队协作精神，坏的一点就在基础知识和技能的要求方面，可谓再含糊不过了：“精通XX语言”，“扎实的XX功底”，“熟悉XX技术”，甚至看上去最具量化感的描述“X年YY经验”其实都根本说明不了多少东西，在信息量方面还不如我家门口菜市场上一家卖酥油饼的店门口挂的横幅——“**三天不硬、至少六层！**”。

很多朋友也许注意到一个现象，**现在企业对招聘者简历的要求也在变得越来越灵活变通**，例如ThoughtWorks在招聘的时候就希望招聘者能给出自己的博客地址，博客对IT行业的意义也许胜过其他所有行业，一个积累多年的技术博客比任何简历都更能说明问题。台湾的郭安定也说“为什么写技术博客对新人如此重要”。可惜这个做法也有一个弊端：并不是所有技术牛人都写博客，有人就是只干不说型的，而就算写博客，乃至动手写过一阵子的，写一个常年的博客，也远比你想象的更为困难，因为很多时候，写（说）得靠谱比做得靠谱更难。所以这个过滤器很多时候用不上。

但是这的确表明了一个思考的方向，就是寻找更具鉴别力的过滤器，Stackoverflow Careers 2.0之所以强大，是因为Joel Spolsky和Jeff Atwood这两位常年混社区的资深博主创造性地将一个人在社区的活动历史浓缩成为一系列的量化数值，由于这个历史很长期，所以鉴别力非常高。但它同样也有问题，就是对于应聘者来讲相当花费时间，而且并不是花时间（在Stackoverflow上回答问题）就一定能花到点子上。

到底什么特征才是既通用，又能够有效地鉴别高低应聘者的特征呢？**这个特征必须**

不像博客那样难以实现，同时又必须有足够的区分度。

有的地方在要求填写简历的时候必须填上平时都访问哪些技术网站。恩，很不错的尝试，可区分度仍然还是不够，因为上网站上查东西毕竟只占现阶段大多数应届生的少数信息来源，特别是当我们看重得更多的是应届应聘者的系统性的知识基础的时候，网上的东西虽然丰富，但属于提高班，也更为琐碎，什么是更系统的知识来源呢？答案其实大家都知道——

书。

我一向认为，**很多时候，是否好好看完一本好书，对一个人的提升往往能达到质的区别**。就算不好好看完一本好书，马马虎虎看完，只要书是真的好书，也肯定会有很大的提高。我在面试的时候就经常询问对方看过哪些技术书籍，经常上哪些网站，订哪些博客。这里头尤其数书籍这一项的区分度最高。此外，**好书和坏书的差别，从本质上，就是学习效率和大方向的差别**。一本烂书可以浪费你半年的时间，但一本好书却可以为你带来真正扎实的基础和开阔的视野。人们常常用“内功”来形容扎实的基础，认为学好了内功以后学什么都快，其实一点没错，好的“内功”书不仅讲清楚深刻的原理，而且指明技术的本质，刻画领域的地图。好的书抓住不变量，让人能够触类旁通。好的书不仅介绍知识，而且阐释原则，介绍那些万变不离其宗的东西。**读烂书浪费时间，但读好书却节省时间**。

象牙塔内的学生受到视野的限制，往往择书不慎，事倍功半，烂书不仅浪费时间，还会打击人的积极性，让人对知识心生恐惧，认为很难掌握，殊不知只是作者没有讲好（或者没有翻译好）。因此，为招聘头疼的公司完全可以给出“**应聘俺们公司前必读的十本书**”，也不一定要每个公司都不一样，在某个技术子领域有影响力的人，或者创始人们，可以来定义具有代表性的书单。

我们姑且把这个计划叫做“书单计划”，容易看到“书单计划”具备以下几个卓越的优点：

1. 清晰、明确。完全可度量。
2. 防伪：读没读过，随便一问便知。而正因为应聘者也知道这事不像实习经验可以忽悠，所以也不敢乱往简历上捅词。
3. 不在乎是否“泄题”：书单完全公开的，无所谓，本来就是要你去读的。想背题？背书吧。真能背下来说明认真看了。
4. 管你用心不用心读，只要读了，读完了，就有区别。真正的好书，你想不被吸引都难。据我观察很多人就是不知道该去读什么书。
5. 不存在“怎么做”的障碍：所有人都知道怎么读书——一页一页读。
6. 不需要招聘者投入精力：书单在此，就这么简单，您看着办。

7　评估的负担很大程度转移到了应聘者的身上：是不是认真看完了，有没有心得体会，您自己掂量。没看完别来找我们。

“书单计划”能很大程度上起到**强鉴别器**的作用，看了就是看了，必然能学到东西，没看就是没看。**知道和不知道，区别是本质的。其实很多企业内部培训，根本上其实还不就是叫员工去看之前没看过的书或者资料嘛**。最后，除了鉴别作用之外，它还是一个清晰促进的目标，是**完全不花精力的培养**。

当然，“书单计划”的背后是另一个悲剧的现实，如果不是因为这个现实，这个计划也完全没有必要，那就是，中国IT大学教育当中要求要学的书，和企业真正需要你去读的书相比，不是完全不够用，就是写的不够好，或者更悲剧的就是根本用不上，所以在这个大背景下出来的牛人都是自己淘书自己学的。微软高级开发测试工程师，《Windows用户态程序高效排错》作者熊力就在微博上说过：“我当年毕业的时候总结了一个公式：第一份工作的月薪=大学四年买过的技术书籍价格的总和。”

但是光有“书单计划”还不够，因为**书籍只能管基础知识这一块，一些更难以量化衡量的实战“能力”又怎么办呢？**至少目前为止，除了“练”之外好像还没有特别好的办法。可是在象牙塔里面做的项目，或大作业，真的能起到练的作用吗？前面说了，学生会知道自己最终要交差的不是雇主，而是老师，于是就以老师能够评判的标准来默认要求自己了，老师能够评判编码素养？代码风格？文档？设计？协作？甚至连著名的Joel 12条的第一条“是否用源代码管理系统”都没法通过。所以大多数时候，大作业能起到的作用近乎0。

但是如果这一切是由雇主来评判的，这个“作业”是由雇主来给出的，就完全不一样了。一想到作业是要作为简历的一部分的，能不紧张嘛。能不好好做嘛。能不学到点东西嘛？

可是这事儿能实现吗？雇主能给学生出大作业吗？也许一两个关系好的高校可以，可是中国那么多学生呢？

为什么不能呢？如果像书单那样，列出各个技术领域“推荐在学校期间尝试的项目”，至于动不动手做，那是学生自己的问题。做的，自然能够得到锻炼，面试的时候自然能得到更大的优势。

可问题是，面试的人又怎么来评估呢？这不又回到了没法有效评估的怪圈了吗？答案很简单，但这个答案，直到最近几年，才真正成为现实——

GitHub

GitHub诞生于08年春天，第一年便产生了4万6千个公共项目，大约一年半之后用

户就已经达到10万用户之巨。而到今年九月份，GitHub已经迎来了百万级用户。Host超过两百万个项目。

增长的太快了！就像Twitter一样。这样疯了一般的增长只能说明一个事实——**人们等待这个产品太久了**。

Social Coding。

真实的项目，真实的流程，真实的人名，一切代码review, check-in, test, build, document, 甚至讨论，计划，brianstorming，流程，一切的一切，都是项目历史的一部分，都可以像棋局那样复盘。有经验的面试者只要稍稍扫两眼一个人的GitHub历史，挑出几个check-in历史看一看，便完全能够迅速判断这个人是否满足他的要求。不再需要费劲心机地去想题目，去观察，去揣测，去花费大量的时间的同时还只能采样到几个极为有限的点。

不像象牙塔里面大作业，这里有源代码管理系统，自动化build，有check-in，有review，有分工，有合作，最重要的是——这是一个集市，一个超出象牙塔的集市，牛人相互吸引，**你可以在互联网上找到和自己拥有共同兴趣的一帮人，真正做起一点事情，而不是交差，不需要受限于几十个人的一个小班级。Here Comes Everybody**。

为什么我这么有信心？因为这事儿已经发生了。**这个想法也完全不是我原创的**。

正如很多事情一样，现在在国内发生的事情，往往是美国那头的历史。今年7月中旬，纽约一家公司的工程师老大发了一篇博客文章：Github is Your New Resume。指出一个惊人但再合理不过的事实：越来越多的IT公司在招聘的时候要求应聘者给出GitHub账号。甚至已经有人为GitHub写了根据GitHub上的历史自动生成简历的工具。

仔细想想，这是必然的趋势，没有比这个再合理的事情了，既然StackOverflow的历史能够作为简历，GitHub的历史不本该就是更好的简历吗：你想要具有实战经验，懂check-in懂review懂test和代码质量的重要性，懂交流和沟通的重要性，你本就应该在一个真实的项目当中去锻炼这些东西，而这些在目前已经完全可以办到。正如邹欣老师所说，你的工作就是最好的面试。

这件事情放在早几年，是完全没法做到的，因为我们那时候还没有GitHub。正如没有Twitter，没有微博之前，很多事情都不会成为可能一样，你有千钧之力，缺乏一个合适的支点，也没法撬动一整个社群。无组织中的组织，具有强大的杠杆效应。

这个事情里面，我唯一提出的东西就是：在目前国内这个现状下，苦闷的招聘者应该主动行动，给出一些建议项目，正如前面提到的书单计划一样，招聘者需要给出的只是引导和**清晰明确的目标**，剩下的事情，应聘者自然会去完成，这些项目可以是实验项目，也可以是完全能做出点卖钱的东西的项目（如果好好做的话），唯一的不可或缺的前提是，项目不能太小，单人就能完成的项目不理想，一两个月就能完成的项目不理想，最好足够大到能够锻炼到方方面面，偏大一点倒是无所谓的，因为一个尚未完成的项目完全可以作为简历。当然，可以想见的是，真到了那个时候，学生们肯定又是不会满足于仅去做那些已经有许多人做过的项目了。所以这里企业们一开始所建议的项目只是一个《Nudge》，是滚雪球之前需要的一点初始动能。后面的事情，他们自己会完成。

“GitHub计划”同样有一些明显的、甚至不可替代的优点：

1. 清晰、明确，完全可度量。
2. 防伪：同样不担心“泄题”。你伪造不了GitHub历史，伪造不了check-in历史，review comments，文档，交流记录…
3. 它不但是招聘，也是不花精力的培养。善哉善哉。
4. 评估的责任很大程度上交给了应聘者自己。

从你的GitHub旅程开始，你就已经一脚踏进了真正的企业，而企业的面试也已经开始。

书单+GitHub，就相当于一个两年左右的面试。

没有什么面试比持续两年的面试更具有信息量。

书单，加上项目，已经基本上覆盖了所需的全部技能。最妙的是，有太多的人在焦急的等待着他们未来的雇主给出明确的信号，他们想投入精力，去学习和实践，去成为企业需要的人，但是他们就是不知道往什么方向走，所谓**有动力没方向**。所以，雇主给出了清晰明确的要求，相信对于很多人来说反倒是一个解脱：“终于知道该干什么了”。《编程之美》为什么常居畅销榜？因为它透露了雇主眼中的需求，明确、清晰的需求，可以实现，并且知道怎么去实现的需求。

你提前两年就开始面试和培养未来的候选者，而且还不需要你花出一分精力，而且人家还很乐意，没有比这更完美的面试了。

想一想，以后那些没见过世面的公司看见你拿出GitHub账号给他看，该是多么惊讶同时又觉得多么合理。

而这一切，只是因为两个小小的改变：

1. 由需求方（雇主）给出了清晰、明确的目标。
2. GitHub这样的平台。

那么，学校/老师在这个事情当中的位置呢？说实话我不知道。没有哪个行业像IT行业这样特殊：没有什么东西不能够（应该）在互联网上学到的。自组织的力量完全大过传统的教育方式。而且，既然雇主都当了领路人了，我不知道还有中间开发商什么事儿。（注：这里说的是软件开发，并非计算机科学研究，后者另当别论）

那么，这个改变会发生吗？多久会发生呢？当然，它在国外已经发生了，所以问这个问题多少有点无趣。但我还是预计很快就会在国内发生，毕竟，不是已经有人要求出示博客，和经常浏览的网站了吗？也许5年左右（4年本科和6年硕士的中间值？））就会深刻改变整个人才培养/招聘的格局。当然，我并不是预言家，所以不要把我的时间估计当真，我能肯定的是，这种方式是必然的大势所趋。

刚才我就收到一位同学邀请我上知乎回答一个问题“找工作的首要原则是什么？”，当然，这个问题的答案是：“**弄清雇主的需求到底是什么**”。

列一下我所认为的，你面试微软前必须要读的十本书：

1. Code: The Hidden Language of Computer Hardware and Software （《编码的奥秘》）
2. Computer System: A Programmer’s Perspective （《深入理解计算机系统》） / Windows via C/C++ （《Windows核心编程》 / 《程序员的自我修养》
3. Code Complete 2（《代码大全》） / The Pragmatic Programmer （《程序员修炼之道》，我也把这本书称为《代码小全》）
4. Programming Pearls （《编程珠玑》） / Algorithms / Algorithm Design / 《编程之美》
5. The C Programming Language
6. The C++ Programming Language / Programming: Principles and Practice Using C++ / Accelerated C++
7. The Structure and Interpretation of Computer Programs （《计算机程序的构造和解释》）
8. Clean Code / Implementation Patterns
9. Design Patterns （《设计模式》） / Agile Software Development, Principles, Patterns, and Practices
10. Refactoring （《重构》）