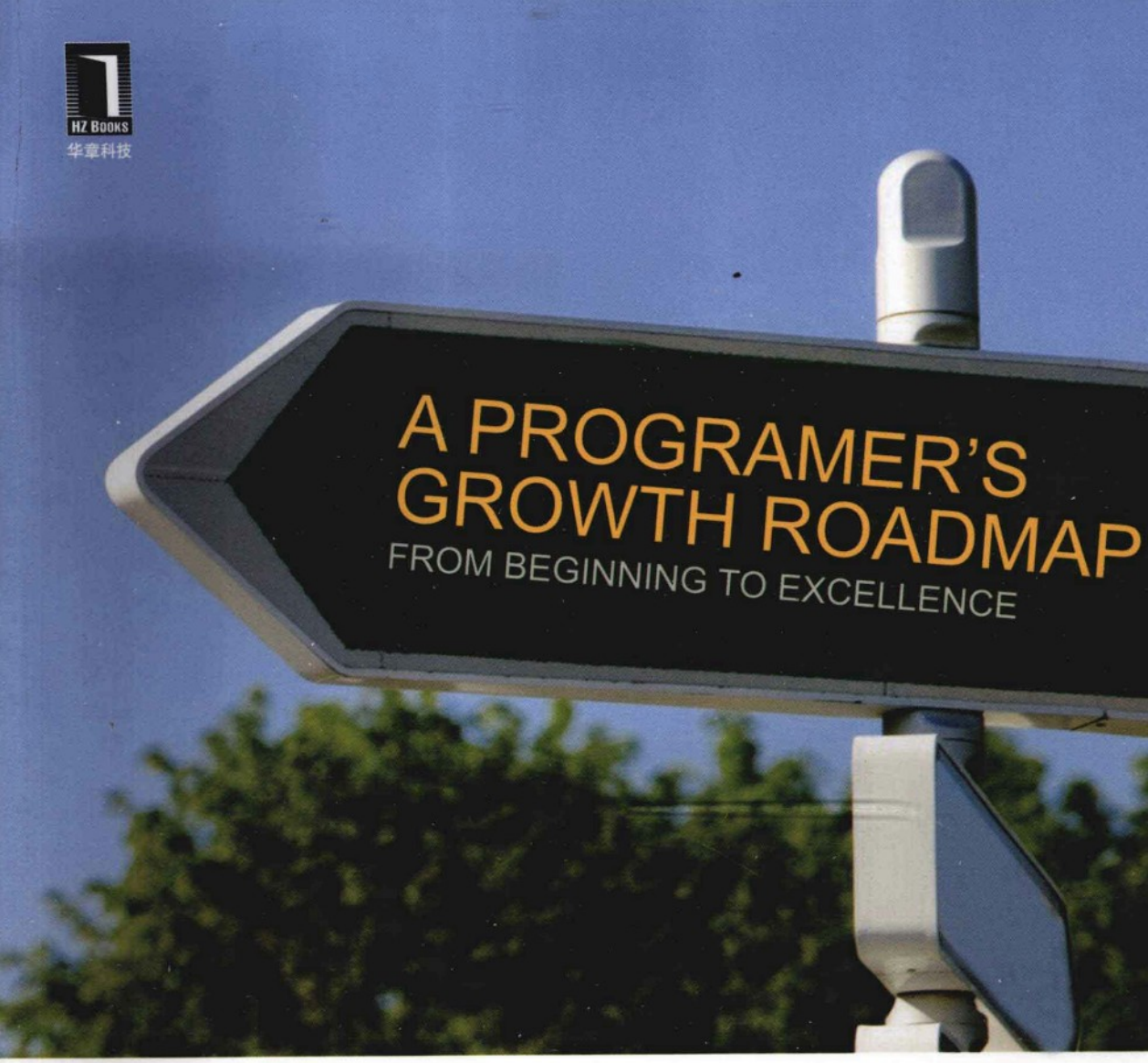
HZ BOOKS
华章科技



程序员成长路线图
从入门到优秀

揭示程序员成长中的热点、重点、难点问题

N216 张磊 吉阳 著

机械工业出版社
China Machine Press

作者N216于1979年上大学期间开始编程序，目前仍坚持在编程工作岗位。几十年的程序员成长经历使其对程序员成长深有感悟，他清晰地划分了程序员成长的阶段，并且揭示了各阶段需要注意的热点、重点、难点问题，以帮助程序员清楚认识优秀程序员的标准以及成长路线图。

入门篇

程序员的梦想——中国的比尔·盖茨
语言选择与就业方向
新手如何学习一门新的语言
选择大公司还是小公司
新手应当具备的基本素质

成长篇

加班，加班，加班
为什么程序员不愿写文档
为什么程序员的社会地位在下降
如何快速确定自身水平
程序员应该掌握的实用编程技能

成熟篇

有关程序员的性别、年龄、个性、编程方法的话题
跳槽还是留守
你能当老板吗
谈谈程序的参数化
程序员成熟的标志

优秀篇

成熟到优秀的瓶颈问题
制作有价值的软件才是程序员内在的目标
从程序制作到构架制作
从定制软件到通用软件
优秀程序员应该具备的素质

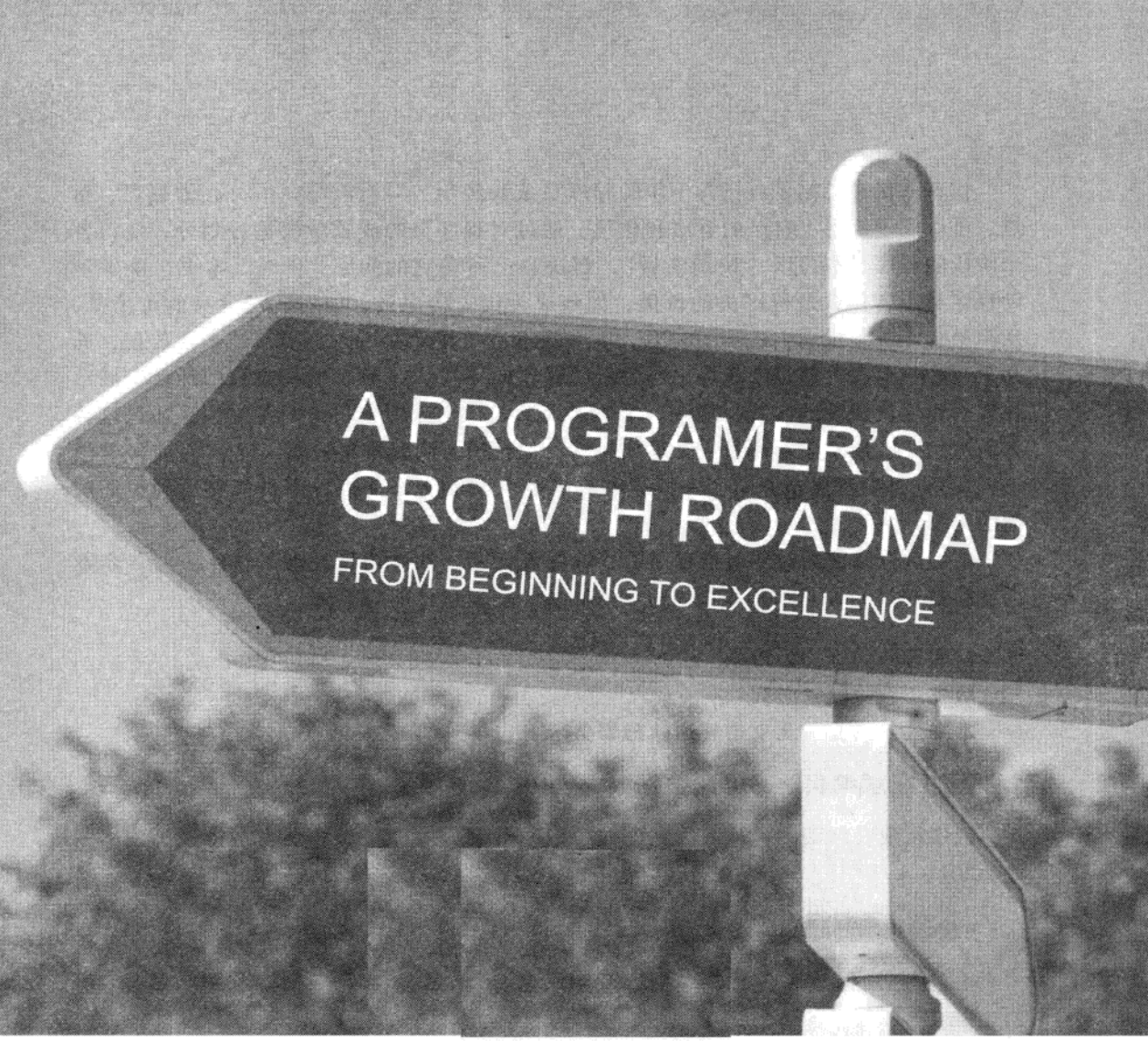
本书既不是纯技术文章，也不是纯个人传记，而是采用随笔形式，以促膝交谈的方式系统地讲解程序员成长过程，希望与读者产生共鸣，无论是新手还是成熟的程序员都可以从中受益。阅读本书，轻松中不乏经验之谈，随意中不乏中肯之言。本书以程序员的技术成长和心理成长为叙述的两条主线，希望成为程序员成长的良师益友。

A PROGRAMER'S GROWTH ROADMAP
FROM BEGINNING TO EXCELLENCE

程序员成长路线图
从入门到优秀

上架指导：IT文化
ISBN 978-7-111-33913-7
9 787111 339137
定价：39.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com



程序员成长路线图
从入门到

N216 张磊 吉阳 著

机械工业出版社
China Machine Press

作者回忆和总结了自己几十年的程序员成长经历，对当前程序员关心的热点、重点、难点问题给出了自己的看法和建议。通过对程序员的成长阶段进行划分，使得各个阶段的程序员都可以“按图索骥”，解决自己所遇到的问题。同时，本书也能够帮助程序员了解什么是程序员的价值，如何成为优秀程序员，如何实现自身的价值等，从而给程序员提供向上进步的动力。本书既不是纯技术文章，也不是纯个人传记，而是采用随笔形式，通过问题提出、分析、解答的形式，并辅以个人成长的经历，展示作者对程序员成长的实践与理解。本书以技术成长和心理成长为两条主线，通过这两方面的结合，展示了程序员应如何实现自己的价值。同时本书还涉及了“企业经营模型”，据此针对程序员介绍了一些企业经营模型的知识，使得程序员能够看到更高层面的未来。

本书适合阅读的对象包括程序员、软件设计师、软件项目经理、软件公司（企业内部科技部门）职员、国家政府机关等相关企业信息化部门职员。

图书在版编目（CIP）数据

程序员成长路线图：从入门到优秀/N216，张磊，吉阳著．—北京：机械工业出版社，2011.4

ISBN 978-7-111-33913-7

Ⅰ.程…　Ⅱ.①N…　②张…　③吉…　Ⅲ.程序设计－工程技术人员－自学参考资料　Ⅳ.TP311.1

中国版本图书馆 CIP 数据核字（2011）第 051152 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：秦　健
北京诚信伟业印刷有限公司印刷
2011年5月第1版第1次印刷
170mm×242mm · 16.25 印张
标准书号：ISBN 978-7-111-33913-7
定价：39.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

前　言

只要在编程序就应该称为程序员。若以这个标准来衡量，我可能是国内从事编程工作时间最长的程序员之一。几十年前我是一名程序员，几十年后的今天我依然是一名程序员，未来几十年我还会是一名程序员。

2008 年我提出了 EOM（Enterprise Operating Model，企业经营模型）理论，并把 EOM 系列文章发表到博客园网站上。也许是 EOM 对普通的程序员来说显得过于抽象，若没有丰富的工作经历和一定的编程技术，一般人会很难理解，因此网上反应平平。2009 年年末，为了让普通程序员能够了解 EOM，于是我从程序员关心的热门话题开始，由浅入深地发表了几十篇博文，最终让大家看到了 EOM 是程序员成长中的一个重要结果，它与程序员的成长密切相关。没想到这些文章一发布到网上就引起网友热议，无论是点击率、回帖数，还是作者与网友之间的辩论，激烈程度都很高，而且转载甚多。我的朋友、同事、家人甚至出版社见此状况都极力建议我将这些写成书，让更多的程序员能够分享这种成长。

于是我在 2010 年 5 月正式和出版社签约，开始进行本书的撰写工作。在写作的过程中，我和我的合作者对网上发表过的文章进行了整理，并根据本书的编写大纲增加了很多新文章，把这些文章按照程序员的成长阶段划分为入门篇、成长篇、成熟篇和优秀篇，从而形成了程序员成长的一个完整的“路线图”。在这个过程中我重新回忆了我成长的各个阶段，每个阶段的故事依然让我感到鲜活、感动和难忘。我努力去发现程序员最关注的问题，努力去分析这个问题后面的原因，努力去给出解决问题的建议。我想我的努力会给读者带来不一般的体会。

本书适用于各层次的程序员：有刚出校门寻找工作的新手，有在工作岗位年限不长的程序员，有工作多年的项目经理、技术高手，有在编程事业中表现突出的优秀人物。各层次的程序员都可以从中找到自己想要的看点，例如：新手想知道如何找工作、如何面试、工资待遇、掌握什么语

言、编程的前途如何等一些经验和观点；成长期中的程序员关注的是如何面对加班、如何提高自己的编程能力、如何确定自己的编程水平等；优秀的程序员可能关注优秀程序员应该具备哪些素质、哪些技术水平，如何证实其优秀，软件发展方向是什么，自己事业的未来发展方向是什么，如何实现自己的价值等。

每个程序员的成长之路各不相同，但都会经历大大小小的成功和坎坷，很多人在这个过程中会迷茫、会不知所措。希望本书能成为程序员成长的“路标”，打破那种程序员只看技术类书籍就能提高、就能成长的狭隘想法。希望程序员能多了解其他程序员成长的经历，这些经历不仅指技术上的，而且是指在心理上的、职场上的、事业上的和梦想上的。程序员可以从中学到很多成功的经验，避免常见的问题，使得他们能够更快、更全面地成长起来，更好地在这个职业中有所发展。

在此，我要感谢合作者张磊、吉阳两位同事，感谢我的朋友、同事、编辑、家人，谢谢他们给我的每一次鼓励。他们中有很多人都是我的文章的首批读者，在我写作的过程中给予了许多有益的建议。没有他们的鼓励和支持我可能无法完成这项工作。我还要特别感谢远在英国的女儿倪好，她的鼓励如同我给予她的鼓励一样，本书将是我送给她的一件礼物，希望这份礼物伴随她快乐成长。

除了本书之外，我还打算在近期编写有关软件设计师、项目经理、EOM、NSS 等方面的系列书籍。我想通过这些系列书籍与那些有志促进我国软件业发展的各位同仁分享自己的经验和观点。

由于篇幅有限，话题广度和深度也不可能拓展太多，请有兴趣的读者访问我的博客，参与讨论程序员以及软件业相关话题。

N216（倪燕农）

博客：http：//n216. cnblogs. com/

邮箱：eom_ n216@ hotmail. com

目　　录

引　言

我的程序员成长之路

程序员的成长经历往往很相似，大部分的人走过了最前面相同的一段路，而有的人则走得更远。总结自己这些年来的历程，这也许能让年轻的程序员少走一些弯路，成长得更快；或许更好一些，能让大家从中得到一些启发，早日进入优秀程序员的阶段，实现梦想，释放激情。

第一阶段，最初是在学校里学习计算机基础知识，学习经典的程序设计语言，编写测试用的小程序。这个过程可以说是对计算机和程序设计的入门阶段。这个阶段主要是培养了自己对计算机软件的兴趣，打下了良好的计算机基础知识。

第二阶段，而后参加工作，从事计算机软件开发工作。按照工作要求，一边学习，一边编程，终于可以让自己的程序投入运行了。在这个阶段我突然感觉到了自己的价值，感觉到了软件的神奇，并且自己编写的软件成为了实用产品。这个阶段实现了学习到生产的过渡。

第三阶段，随着工作的增加，开始编写各种程序，开发各种系统，这时候忙于编程知识的积累和应用。应该说在这个阶段自我感觉很充实，好像有做不完的事，程序设计水平还处在语言级阶段。

第四阶段，随着积累了一定编程技巧之后，我开始想这样的问题：我是不是最好的程序员？我能否编写出最好的程序？这个过程是一个反思的阶段。我对自己的要求是：不但要会编程序，而且要编好程序，从关注程序数量开始转向关注程序质量。

第五阶段，开始在提高自己的软件开发水平上做文章。经过各种系统开发，尤其是大型系统的开发，发现了软件中有许多功能是重复的。因此，有一段时间把精力花在编制各种库函数上，通过不同系统调用相同的函数，以便减少重复开发，实现功能共享。当时比较得意的是库函数不是我一个人在调用，而是整个项目小组都在调用，甚至不同的系统也能调用，从而体会到编写库函数特别有价值。这个阶段的标志是库函数，程序员水平上升到库函数那一级。

第六阶段，到了库函数那一级后，很快就发现，单单实现程序函数级的调用是远远不够的。当你做了很多项目，包括大项目和小项目，尤其是做过跨行业的项目之后，你就会把库函数的共享思想用于项目开发。你就会想这样一个问题：为什么不同项目不能有相同的架构？如果有相同的架构，那么开发就有了相对的标准，我们就有可能通过配置的方法实现相同架构的系统。于是我提出了 IASG（交互式软件自动生成器）思想，并在 C 语言和其他一些语言中实现了 IASG 实例。记得最快的一次是编写一个系统（公安部门的自行车信息管理系统，主要用于丢失自行车信息登记）只用了 3 个小时（从需求到安装盘）。这个事情对我影响很大。我在这个阶段上升了一个很大的台阶，从程序上升到软件。核心思想就从库函数共享上升到软件共享。具体过程是建立一个通用的系统架构，架构中有许多共同的功能，例如，参数设置、用户权限管理、库表管理等。另外还提供信息建立查询开发模板，通过配置和特殊功能的编制就能很快完成了一个系统的开发。现在想起来 IASG 距离我已经有 20 年了。

第七阶段，到了 IASG 阶段后，我发现无论技术如何提高，都无法改变开发落后于需求的现实。通俗地说就是：程序员水平再高，仅仅是拉车水平高，但是，应该在什么路上拉车程序员并不知道。如果这条路是一条光明的路，则程序员越拉越有劲，有前途；如果这是一条死胡同，则程序员白费工夫；如果这是一条漫长的路，前途不明，则程序员可能要累倒在路上。现实中程序员水平低、收入低；系统需求不明确，系统开发周期一拖再拖；系统重复开发多，信息甚至不能在一个企业内实现共享，更不用说在企业之间、行业之间实现共享了；各种企业级的软件 ERP、CRM、BI

层出不穷，也没有哪个能满足中国的市场；各种新技术、新概念不断出现，却没有哪种技术或概念能真正发挥其内在价值，最终还是处于被学习、被运用的阶段。

这个过程是程序员脱离技术本身，开始思索、开始求源的阶段。在这个阶段的程序员的思想有了质的飞跃。以前光拉车不看路，现在要抬头看路了。

第八阶段，有了抬头看路的想法，于是我踏上寻路征程。我首先弄明白了我们脚下的路是什么样的，为什么这条路那么不平坦、不宽广。从软件生命周期来看，软件主要由用户需求发起，用户需求是软件生存的根本理由。由于企业、用户的不同而导致不同的需求——大量的无序的需求，这种需求驱动方式必然造成了我前面介绍的各种现象。这个阶段是寻找根源的阶段。只要我们找到了根源，就可以有机会解决问题。这个过程相对来说比较困难，这不仅需要编程技术，还需要很多方面的知识。若要了解这个根源，就迫使你学习和积累更多程序以外的知识。

第九阶段，当我找到软件是需求驱动方式之后，就开始考虑什么是用户需求？用户为什么要提出这些需求？我们可以更深入地分析用户需求产生的根源，我们能否让无序需求变成有序需求呢？当然针对这些问题我们都进行了深入分析，其过程也很难在这里展开说明。我只能说，最后结论是用户的需求来源于企业的经营。很多人思考问题还是就需求而论，并没有站在企业经营角度去考虑问题。千万不要小看这个变化，这个变化最终会产生一个理论。于是我们尽可能地站在企业经营角度看待企业经营方式、企业管理、企业信息化等。但是，我们最终要解决企业经营这个概念问题，如果我们都不能明确企业经营这个概念，或者我们不能科学地定义企业经营这个概念，那一切基于企业经营的各种具体现象就如同无本之源一样无序泛滥。就像 ERP、CRM 等所谓企业信息化产品一样，由于没有一个企业经营定义的支撑，只能就企业经营的某个方面提出解决方案。这些产品不缺乏需求的支持，缺乏的是最基本的企业经营定义的支持。而这个概念就是 EOM。

EOM 是从定义企业经营角度入手，把我们今后要开展的各种研究和开发活动都放在一个理论可支持的基础上。只有定义了企业经营之后，我们才有可能分析我们需要什么软件，我们的软件采用什么技术才能实现企业经营的目标。而程序员则通过 EOM 了解到企业经营需要什么样的软件，这个软件有多大的价值，这个软件采用什么技术才能实现，自己要提高哪方面的技术水平才能获得更大的价值。

这个过程就是 EOM 阶段，通过 EOM 了解软件的根源和有价值的软件所在，进而选择自己未来的方向。

第十阶段，当我建立了 EOM 之后，便开始了 EOM 实现阶段。这个实现阶段分为两部分，通过这两部分的结合，我们就可以逐步看到 EOM 软件产品的实例，看到 EOM 的真正价值。

第一部分是 EOM 的业务实现。当我们明确了 EOM 之后，就可以根据 EOM 来重新规划企业信息化的整体架构，可以细分这个架构中的各种平台产品、通用产品、专业产品，可以细分出这个架构实现的各种技术架构和实现手段，可以细分出这个架构中的各种标准功能和标准信息。通过这样的分析，我们的程序员就可以根据自己的特长和爱好以及价值的判断来选择其中的软件产品和技术。在明确目标和方向的情形下，通过自己的努力，不断提高自己的各种技能水平，让自己的价值和企业经营价值有机地结合在一起，从而实现自己的理想。

第二部分是 EOM 的技术实现。有了 EOM 并根据 EOM 理论构建企业信息化的架构后，我们就必须从技术上实现这个架构，否则这个架构将永远停留在理论阶段，不具有可行性。我们可以采用现有的各种技术来实现这个架构，但是，现有的技术都是基于原有的业务需求而建立和发展的，它适用于原来的应用对象。目前的 EOM 是一个全新的企业经营理念，因此，我们必须建立一种新的软件架构来适应和最好地实现这个理念。幸运的是，我们找到了称作 NSS（New Software Structure）软件新架构的技术，该技术体现了适应企业经营发展方向，将软件合理分层，用最新的软件技术按照架构的方式规范软件开发的模式，可以实现最大范围的功能共享，

实现软件的可扩展性。

这个阶段可以让程序员在软件产品业务设计或软件产品技术实现上等多个方面进行深入钻研，并且成为领域专家。这和我们平时涉及的简单的需求分析和简单的技术实现有着本质区别。

从我的程序员经历可以看出，程序员的成长是无止境的，只要有的放矢地努力，就会一步步登高向上。我认为程序员成长经历主要有三大阶段，即通用技术阶段、市场阶段、专业技术阶段。

1）通用技术阶段是程序员专注编程水平提高的阶段，也就是说“只拉车不看路”阶段。这个程序员能做的事情那个程序员也能做，程序员的替代性很强，程序员市场价值相对较低，程序员只关注编程技术本身。

2）市场阶段是程序员跳离技术层面开始考虑为什么要开发这个软件，这个软件有什么价值的阶段，通过求软件之源来重新认知自己的方向。

3）专用技术阶段是程序员认知了这个软件和技术有很大的市场价值，全身心投入到这个领域中去，并在这个领域成为专家的阶段。程序员不但要懂技术，更要懂得客户业务，不同的程序员的技术和业务变得没有可比性，这种稀缺性造就了程序员极大的价值。

这三个阶段其实就是三个过程，每一个过程都是一次飞跃。程序员知道自己可以飞多高，依靠的是程序员的学习和眼界；而程序员能飞到哪里，那就要靠程序员自身的努力。一个程序员可以没有能力，但是不可以没有眼界。

第一部分
Part 1

入 门 篇

1.1 程序员的梦想——中国的比尔·盖茨

作为一个IT行业职员，我经历过一个普通程序员成长的过程，同时也接触过许多不同层次的程序员。他们或在我身边匆匆而过，或与我共同工作，或在我可以关注的范围内成长着。他们的喜怒和哀乐、挫折和成功、幻想和现实、希望和失望，无不与我心共振。我知道这个行业从业人员的梦想，也知道这个行业的残酷。无数人怀着希望而来，却抱着无奈离去。我早就有和他们共语的愿望，希望通过这个主题和他们交流程序员所关注的各种问题，希望我的经验有助于他们的成长，同时我也想谈谈EOM对程序员的真正价值的影响，以及如何实现“成为比尔·盖茨”这个程序员的最高梦想。

什么是程序员？什么人能称得上是程序员？会编程序的人都是程序员嘛！这个问题看似简单，但仔细想一下，也很难回答。其实在中国，关于程序员的称呼有很多种近似的叫法，例如“开发人员”、“编程人员”、“计算机人员”等。只是现在分工越来越细、专业化程度不断提高的情况下，程序员这个词才逐渐地流行起来。

那么什么人才算是程序员呢？现在看来凡是从计算机专业或相近专业毕业的、以编写程序为职业的人都可算得上是程序员。但是在20世纪80年代到90年代，由于计算机还是新生事物，整个社会对其有种神秘的、高贵的、不可触及的印象。加之当时计算机人才少之又少，除了计算机专

业从事开发工作之外，很多非计算机专业的学生，甚至初、高中生也加入到计算机开发队伍之中。他们充满激情，敢于学习，勇于探索，其中有许多人很快就成为开发队伍中的主力军，成为编程人员中的佼佼者。有的时候，专业的程序员还不如业余的程序员，这种情况比比皆是。那个时候，开发环境、学习环境要比现在差得多，程序设计语言比较单调，技术书籍更是少之又少。记得当时只能把单位印制的 8086、Z80 等汇编程序设计资料当做教材，用 debug 把操作系统中的代码打印成厚厚书籍来阅读。由于当时我的单位是生产（组装）计算机的，因此，使用计算机还是比较方便的，但是，不像现在，绝不可能在家里使用计算机的。

由于那时程序员可以触及计算机，可以看明白别人不懂的代码，可以让计算机执行自己的指令，这让许多外行甚至内行人很羡慕。

程序员一般只掌握单一的程序设计语言，比如编写汇编程序的程序人员，编写 C 语言程序的程序人员，编写 Unix、XENIX、AIX、SCO、HP-unix 等 Unix 类的 shell 程序的程序人员，编写与数据库打交道的 proc c 程序人员，编写 C ++ 程序的程序人员，编写面向对象的 VFP、VB、Delphi、PB 的程序人员，编写 Web 程序的 HTML、CGI、ASP、PHP 的程序人员，编写 C#、Java 的程序人员。这些程序语言有些保留了下来，有的则被时代无情地淘汰了。同样是程序员，如果不能适应语言的发展也避免不了被淘汰的命运。

我自己认为的程序员与其他职业人员之间的区别：

1）因为比尔·盖茨是编程序的，所以似乎每个程序设计人员都有一个“比尔·盖茨”梦想：比尔·盖茨能做的，我也许能做到，即使做不到，做到一半也是不错的。这个潜在的意识是程序员最大的财富，许多程序人成了有理想、有抱负的人。我想很多人选择 IT 大都与此有关吧。

2）工作成果完全由自己把握，随时编随时运行随时出结果。这种自我感觉是很多职业都不具备的。因此，程序员有很强的自信心。而且这种自信心往往可以使得程序员产生自己开公司的念头。

3）有很高的预期价值。软件通过使用创造了价值，程序员通过制作产生了软件。因此，程序员往往把软件的价值看做自己的价值，例如一个软件卖了5万元，程序员就会把自己的劳动价值估算在5万元以上；如果这个软件有100个潜在的市场，那么程序员就会把这个价值升值到5×100=500万以上。所以在程序员这个群体中，很多人都相信自己未来能够获得更多的收入。

当然，程序员也有其他一些特点，例如：有的程序员喜欢晚上干活，白天睡觉；有的喜欢钻研，连续加班；有的头脑灵活，动手能力强；有的喜欢追逐最新技术，变成别人公司的代言人等。

我注意到了有这样一点，那就是现在的程序员已经失去了神秘感，以往给人以仰目而视的形象正逐步走下神坛。

这是程序员职业发展的必由之路，抑或是再正常不过的社会现象？

1.2 谈谈程序员的基础知识

对于程序员需要具备哪些最基础的知识和技能这个问题，不少刚从大专院校毕业出来的新入职员工，甚至是从事过一段时间编程工作的程序员，都是比较模糊的。只有认真掌握一些基础的知识和技能，才能走上程序员这条大道。

说句实在话，我在从事编程工作很长一段时间内都没有关注这个问题，基本上是边编边学，边学边编，从学习中积累，从编程中积累。除了和同事进行工作上交流之外，学习材料很少，基本上是一个人在战斗，也不知道自己是不是成为了合格的程序员。也许是受那个年代所限，当时的程序员人数很少，因此同行间没有什么竞争，有了位置就不怕失去。要是放在现在，真的很后怕。

回到正题，我认为程序员在最初阶段要从流程、语法、调用三个层次要求自己。

1. 流程

这是对程序员最基本的要求，这个层次就是要求程序员能够把一个最简单的程序编辑、编译、运行成功，强调的是掌握编程的环境和流程。

在这个层次上又分三个方面的要求：

（1）计算机基础知识

我见过许多程序员新手，他们都自称学过计算机基础知识。但实际情况是，学而不致用，学的概念太多，自己却理不出头绪，和实际工作对不上号。有的甚至认为自己忘了，什么都不知道。但是我要提醒新手的是，尽管基础知识十分丰富，但是有关编程的一些基础知识和概念是必须掌握的。

1）操作系统

什么是操作系统？你所要编写的程序在什么操作系统上运行？目前主要有 Windows 类、Unix 类、Linux 类操作系统。每种操作系统对编程的影响是不同的。

2）计算机、内存、硬盘

这些概念对编程来说也是最基础的，例如计算机分为 PC 机、小型机、大型机。在 PC 机上编程和小型机上编程是有差别的。程序设计语言安装时也要注意内存大小和硬盘大小。

3）目录、文件

这些是最基础的概念了！一定要掌握和理解。因为你编写的程序就是一种文件，而且要放置在指定目录下。

4）程序设计语言、程序、编辑、源程序、编译、可执行程序、运行

这些概念也是最基础的。不同的程序设计语言对编程具有很大的影响。

目前主流的程序设计语言有 Java、C#、C 语言等。

（2）编程环境

作为程序员一定要知道自己的编程环境是什么：

1）了解所要使用的计算机

2）操作系统安装

3）程序设计语言安装要求环境

4）程序设计语言安装盘

5）安装程序设计语言

PC 机和小型机有很大的不同，使用 PC 机的程序员一定对 5 个部分要全部掌握，使用小型机的程序员只需要了解程序设计语言是否安装好了。

（3）编程流程

PC 机开发程序的一般流程：

运行开发环境→建立项目→建立源程序→编写源程序→编译项目（源程序）执行可执行程序→查看运行结果。

小型机开发程序的一般流程（以 C 语言为例）

编写源程序→编译→运行→查看结果。

程序员要验证自己是否掌握了这个层次，建议编写一个最简单的显示“hello world!”的程序。如果程序员能白手起家，能运行自己编写的程序并能显示“hello world!”则可以通过了。

2. 语法

这是对程序员的基础要求。这个层次建立在上一层的基础上，应该掌握程序中最基本的语法、运算、基本功能。

主要掌握：进入程序参数、退出程序命令、赋值语句、条件语句、循环语句、引用、字符串操作、算术运算、文件操作等。

对于人机交互程序主要掌握：窗口、标示控件、文本框控件、命令按钮控件、列表控件、下拉框控件、表格控件等。掌握对控件属性赋值、对属性的读取、增加事件、对事件的调用等。

3. 调用

这是对程序员的一般要求。这个层次建立在上一层次的基础上，应该掌握程序中对数据库、库函数、动态链接库等外部环境的调用。相应的概念也要掌握。

另外，程序员还要具备查询语法和寻求帮助的技能。

这里还谈不上程序员水平问题，需要强调的是，以上是程序员必须掌握的，是最低的要求。由于开发环境不断在变化，程序设计语言也不断在升级，作为程序员就应该扎实地掌握这些方法，做到心中有数，以不变应万变。

1.3 语言选择与就业方向

程序设计语言与就业之间的关系现在变得非常强烈和敏感，这是我始料未及的。记得当年自己在学校里学习程序设计语言，自己从没有选择程序语言的概念。基本上是学校规定学什么，自己就学什么，从未考虑到这些学习将对今后分配工作产生的影响。这可能是由于在当时的社会环境下，不愁就业的状况所造成的。没有了就业的压力，学生可以安心学习一些基础性的课程，可以学习很多经典的程序设计语言，看到语言发展的轨迹，从而对一种程序语言打下良好基础。当然，很多学生并没有把程序语言学好的主观意识，只是想做学习的“奴隶”，而不是为自己真正掌握生存技能服务的。若跟这样的学生交流好好学习，那就是浪费时间。

而今，社会已经发生翻天覆地的变化。程序员从高高在上的“稀缺人才”地位开始下降。这期间的变化令人感慨和无奈。不得不承认，我们已

经进入了市场经济社会，而且是一个充满竞争的市场经济社会。面对市场和竞争，我们必须拿出满足市场需求的商品：我们的编程能力。编程能力有两个方面的含义，一是编程语言，二是编程本身能力。所以，如何选择我们的编程语言是首要问题。而编程技术的提高则需要时间积累和项目积累。

当我们能够树立市场观念和就业意识时，我们对程序设计语言的选择就变得相对简单了。就这个问题我想提出自己的一些建议供各位参考。

1. 就业方向

程序员就业范围应该很广。有的是软件工厂式的编程工作，有的是项目开发的编程工作，有的是用户单位维护类的开发工作，有的是后台编程工作，有的是前端编程工作，有的是编制网站工作，有的是软件培训工作，有的是与硬件相关的汇编级编程工作，有的是数据库类编程和管理工作。随着软件专业化分工的加快，分工会越来越细，就业种类也会越来越多。面对各种就业种类如何选择，这是一个仁者见仁智者见智的问题。只有先定下就业方向，然后再根据就业方向所要求的必须掌握的语言来确定自己要学的语言。

例如，你想去做后台的编程工作，你就可能会选择 C 语言、数据库等。

如果你想去编写网站，你就会在 C#、Java、PHP 中进行选择。

如果你想去做底层与硬件有关的编程，你就可以学习汇编和专业的单片机语言。

2. 市场状况

当你自己不知如何选择的时候，可以从“唯利是图”角度出发，看看在程序员这个行业当中，做什么职业收入最高。你可以通过百度搜索的方式反复比较，获知收入相对较高的职业，然后根据这个职业要求选择所需

的编程语言。

你当然可以从“技术至上”的角度出发，你可以通过百度搜索的方式了解当前哪种语言最流行，因为流行的程序往往体现出这种语言未来可持续发展，当你选择了流行的语言的时候，这就意味着软件市场对这种语言的程序员需求是比较大的，因此，就业的概率相对较高。但是，正是需求增加，在供过于求的情况下，竞争就会加剧，如果你没有表现出更多的能力的话，就很难在竞争中胜出。

当然，你也可以从“反向思维”选择相对冷门的职业，由于是相对冷门的职业，竞争就会有所减弱，就业的概率同样会大大增加。

3. 社会资源

就已经参加工作的人来说，就业还取决于个人的社会资源，假如你有足够的社会资源，你就可能想到哪里就到哪里。这种个别的现象不在我讨论的范围之内。其他的个人的社会资源如学校、家庭、朋友、朋友的朋友，其他求职中介和求职渠道，都是你求职的重要因素。例如，你的朋友正好在一家软件公司，而这家软件公司正好要招聘你这种程序员，求职往往会很快成功。因此，将目光放在你的社会资源上，看看他们能够触及的单位，然后去了解这些单位对程序设计语言的要求，再进行语言的准备，我感到这样做还是非常有必要的。

4. 个人基础

当然，你通过各方面考量最终选择了就业方向，也就确定了你的程序设计语言。你可能在学校学过这个语言，也可能没有学过。对于你来说都要有一个重新学习的阶段。这个学习阶段和大学里无忧无虑的学习有着本质区别，这可是关系到你能否被用人单位录用，关系到你今后的生存，关系到你今后的职业生涯规划。因此，你不但要把用人单位所需要的程序设计语言学好，而且要把相关的其他语言学好，以及相关的计算机基础知识学好。这样才能把你的个人编程基础夯实。可以说基础是必需的。

另外要有的放矢，了解用人单位所采用的语言、开发项目等情况，只

有这样，自己在应聘的时候才会更有把握。

我们可能因为语言而获得就业，我们也可能因为语言失去工作。关键是我们每时每刻要注重语言的发展趋势，注重用人单位的语言发展趋势。语言的学习可以伴随程序员终身。从我的经验来看，要想进入程序员这个行业，主流的程序语言都是应该了解和掌握的，至于掌握的深度可以根据就业的要求深浅不一。因为语言的掌握是无止境的，人们不可能为掌握而花费太多的时间和精力。就目前而言，学习 .NET、Java，学习任何可用于网络应用软件开发的程序语言都是非常有前途的，因为基于网络、互联网、手机（包括移动设备）的软件将是未来软件的主流。

现在我们因为语言而生存，未来能否让语言因为我们而骄傲？

1.4 新手如何学习一门新的语言

学习程序设计语言是程序员的看家功夫。许多程序员边编边学，没有止境，可以说是“活到老学到老”。在语言面前，程序员永远是学生。然而，程序设计语言是一项不断发展的技术，从机器语言到汇编语言，从低级语言到高级语言，从面向过程语言到面向对象语言，从 C/S 语言到 B/S 语言，从非跨平台语言到跨平台跨系统语言，从纯语言到开发平台工具，真的是目不暇接。这些革命性的技术浪潮推动着程序员不断学习新的语言以适应其发展，否则程序员将被淘汰。

学习新的语言有两类人员，一类是从来没有程序设计语言基础的、没有编过程序的人；另一类是已经掌握了一门或一门以上语言，正想要学习更高版本语言或新的语言的人。对于第一类的人我且称为新手。

新手要学习一门程序语言，若是将新手比作一张白纸，可以画出最新最美的图画，同样也可以画得乱七八糟没有美感。新手的第一门语言对其今后的影响是巨大的，如果学得好，则自信心大增，对今后的发展非常有利；如果学得不好，学不下去，则对其职业生涯打击很大。新手面对众多的语言往往无从下手，第一，不知道学哪种语言为好；第二，不知道如何学习；第三，不知道能不能学好！

我认为，就目前而言，先学习 Java、C#，今后再想学习应用于后台的 C 语言都是不错的选择。

针对第三点，我认为只要想学，有职业压力，没有学不好的，只有学不到最好的。

关于第二点，我的建议应特别注重两个要点：一是基础，二是动手能力。学语言首先要看书，然后要动手。那些认为光看书就能学会语言的想法是很幼稚的，而那种光注重编程而不注重读书的人，将来一定是动手能力强而编程水平低的。

1）新手不要急于求成，要把时间放长一点，先把基础知识学好，基础越扎实，今后编程水平就越有可能提高。看书至少要花三个月时间。

主要选择计算机原理、程序设计原理以及所学语言相关（最好是入门类，不要刻意地选择哪本，对初学者来说，任意一本都是好的）的三类书籍。

2）由于看书过程中肯定会遇到很多不懂的概念，而且各种概念之间的关系也不容易理解和掌握，所以许多新手望而生畏，坚持不下去。这个时候一定要学会坚持，坚持读下去，反复读下去，对实在不懂的概念要注意收集，将其牢记在心。这个时候最好不要找人去解答，而是把所有的书籍反复看完 3 遍之后，再去找人解答，这样就可以加深对这个问题的理解，而且解答者也愿意回答。如果一有问题就去问，一是解答者容易失去耐心，二是自己对问题没有印象，容易产生依赖性。

3）有了一定基础知识之后，就要自己想办法安装编程环境。安装编程环境的要点可参照 1.2 节中有关“流程”的阐述，这里就不再赘述。

4）开发环境安装完成后，新手就可以编写显示“hello world!”程序了。

编写这个程序目的主要是学习主程序的作用、主程序的参数入口、简单的赋值语句、显示功能调用和退出程序语句。通过完成这个程序，新手就可以对编写程序流程有一个切身体会。

5）接下来可以学习算术运算编程。试一试算术表达式编写，可以简单地编写一个计算器示例；也可以编写一个显示日历的复杂程序（若输入年份，则显示这个年份的日历）。在这个阶段主要是学习函数以及函数的调用、算术运算、条件语句、循环语句、显示功能等，这些都是编程的基础。

6）完成算术运算的学习后，可以编写更复杂的完整的程序了。例如可以编写一个学生信息管理程序。其功能为：接受一个学生信息（例如，学号、学生姓名、班级、年龄）并把它保存在计算机中，同时提供增加、删除、修改、查询功能。信息保存形式可以是文本文件，也可以是数据库。

这个阶段主要是学习变量、数据存放、文件操作、数据库操作、程序完整性等，这也是编程的基础。

7）完成上述学习之后，程序员要学会回头梳理自己编写的程序，梳理自己已经学过的概念。可以对自己以前的程序进行修改，培养不断提高自己编程水平的意识。

8）在这些过程中若遇到问题，先看帮助。帮助不行，最好是找懂行的人询问，不要自己钻牛角尖，浪费时间。上网查询也可以，但是不如询问别人直接。看书是不能解决问题的，切记！

9）在这些都完成之后，程序员可以有目的地针对自己将要开发的内容进行相应的技术学习和准备了。

10）编写程序的时间估计需要 2 ~ 3 个月。一般而言，一个新手学习一门语言需要半年左右的时间。有的可能要少一点，有的可能会多一点。但是，无论如何，最终的结果是程序员对语言有了初步的了解，可以用语言编写简单的程序了。

1.5 理性看待考证热

在当今这个社会，经常会出现各种所谓的热门现象。这说明目前的社

会相对浮躁，而IT行业也同样不能避免。IT证书可谓名目繁多，应该有几十种。IT证书（认证证书）大体可分两类：**一类是各大企业的认证**，如微软、IBM、SUN、CISCO等认证考试，这类考试主要是考核考生的专业技能和特殊技术的水平；**另一类是国家认可的考试认证**：如全国计算机等级考试（NCRE）和全国计算机应用技术等级考试（NIT），考核的是考生综合的或某一领域的标准要求达到的程度。其实几乎所有的程序员都知道，程序员的能力是一种综合能力，其动手能力的成效是其能力的主要方面。若认为单靠考试就可以反映出考生的实际能力，那简直就是一个笑话。

证书的出现说明社会对任职资格的追求，反映出社会对无序现象的一种规范要求，反映出对“持证上岗”的认同。社会资格的本质是对人们之间竞争的一种限制。为了突破这种限制，获得在竞争中的优势，人们便千方百计地获取这种社会资格。于是社会上便出现了三类群体，一类是发放证书的，一类是获取证书的，一类是招聘中认可证书的。由于这三类群体都能从中获得直接的或间接的证书经济利益，于是，证书不火也难。当证书火到一定程度的时候，其原本内在的职能就慢慢地被异化了。人们不是为证明证书持有人的能力而设立考试，而是为了获取考试带来的经济利益而设立各种证书，于是证书的价值就贬值了。

但是，我们很多考生并没有深入地看到证书背后真正的内容，他们认为用人单位需要证书，没有证书就不能被录取。社会上大量流传着未经证实的证书和高薪之间的关系，更使某些证书神乎其神。因此，他们在证书方面花了大量的时间和金钱，为了证书而证书，能考多少证书就考多少证书。为的是在应聘时递交的简历上可以附上厚厚的各种证书。

就我本人而言，我好像没有考过什么证书，工作依然很顺利。这虽然和我成长所处的年代有关，但更重要的是我可能更关注自己能力的提高。如果自己的能力水平能够达到用人单位满意的要求，我相信即使你没有那些证书，用人单位也会考虑录取你的。我也相信有些用人单位的人力资源部门会按规定办事，没有证书不予录取状况也是存在的。这个时候，你自己就要权衡了，是进入按能力看人的企业求得发展呢，还是进入凭证书看

人的企业求得发展呢？

我对证书方面的建议：

1）如果不占用自己大量的时间和金钱，能考多少证书就考多少证书。把考证书当做消遣和对知识的复习。

2）如果需要对国家证书和企业证书进行选择时，在没有针对性的情况下，选择国家证书。

3）如果自己对某类公司或某类岗位有意向的时候，最好打听其是否有刚性的证书要求，若有，则要不惜时间和金钱去获得这个证书。若没有刚性的要求，仅仅是一种参考，那就要根据个人情况，例如时间和金钱方面的条件，进行取舍。

4）在已经有足够证书的情况下，不要见一个证书就去考一个证书。尽量把时间放在自己的学习上和能力提升上。把时间和金钱节约下来做更重要的事。

5）不要和别人攀比证书多与少。因为每个人就业方向和机会都是不相同的。

6）没事的时候，可花点时间跟踪流行证书的情况，以便自己及时掌握证书最新情况，早作打算。

7）工作之后，也有可能因为你考了相关证书，可能会对你的薪水、岗位产生有利的影响；也可能对你跳槽有所帮助。这些在现实生活中也是可能出现的情况。

总之，我们要理性地对待考证热，我们不赞成为考证而考证，我们也不赞成能考证而不考证。我们赞成那种实用主义的态度：当我们刚性地需要考证的时候，我们就准备考试，争取获得证书。当证书仅作参考的时候，我们能考就考，不考则把时间和精力花在自己的能力提升上面。

在国内，证书的泛滥早已使证书失去了原有的价值。但是，我们应学会内外兼修，也不妨在提升自己内在能力的同时学会打扮自己，让自己更

加漂亮一些，让别人更欣赏自己。

1.6 选择大公司还是小公司

很多人在第一次求职的时候几乎都会遇到“到大公司还是到小公司”的问题。他们认为大公司的薪水高、工作稳定、技术水平高、升迁机会多，但是，大公司要求高，竞争激烈，自己怕进不了；而小公司薪水相对低一些，工作稳定性较差，技术水平参差不齐，升迁机会不多，但是，录取率相对较高。这可能是各种求职中的一个常见话题。这个选择应该是因人而异，几乎没有什么正确答案。所以，我们看到过选择大公司的人获得了成功，也看到过选择小公司的人获得了成功，相反的情况我们同样也见到过。

作为程序员的求职，我想还是具有职业特殊性的。程序员职业和一般的职业有很大的不同，程序员职业有其鲜明的特点：1）个人劳动；2）产品可复制。这两个特点注定了程序员有很大个人发展的空间。

1. 个人劳动

表明其工作主要和劳动者自身有关。无论是程序员独自承担一个编程项目，还是在一个项目中承担部分编程工作，都可以归结为个人劳动。不像有的职业需要昂贵的劳动设备，依赖昂贵的设备，例如：炼钢工人必须要有炼钢厂的炼钢炉；也不像有的职业需要其他人联合劳动，例如，流水线上的装配工；也不像其他职业有严格操作规范和工作流程，例如，制药厂必须按照制药的配方生产药品。

程序员只要有台计算机（其价格可以忽略不计）就可以工作了，而且在工作时是一个人在工作（工作前后以及项目的衔接可以忽略不计）。其编程过程完全取决于程序员个人的技术水平发挥。

2. 产品可复制

绝大部分企业生产出的是实物产品，每个产品都需要投入一定的原材料，价值很大限度上与其原材料的成本有关。劳动者生产一件产品，其最大的价值就限制在这个产品的售价之内了。

而程序员生产的产品是软件，软件最大的特点是可复制性，而且可复制得不计其数。因此，程序员生产软件的价值就是软件拷贝数。当拷贝数不断增加的时候，程序员生产的软件价值就在不断增大。一个软件最大的价值等于其单价乘上可能的用户数。如果这个用户数是成千上万的时候，程序员的一个劳动价值可达到一个天文数字。所以，世界上成功的软件企业，正是这种可复制性的受益者。

程序员的这两个职业特点表明程序员的发展空间很大。程序员不管是在大公司还是在小公司都能获得成功。只要这个公司能生产和销售出可复制的软件，程序员都能够获得比职业高得多的收益。关键是程序员是否具备编制这些软件的技术，这些软件能否被大量复制（定制软件价格高也行）。

比尔·盖茨正是成功通过编制软件而将微软发展成为一个软件帝国的。在现实生活中，也有一些成功的程序员自己当上了软件公司的老板。不成功的也有很多，但是，其中的原因不在此，我会在后面的内容中加以分析。

通过以上分析，当程序员面对“选择大公司还是小公司”的时候，我建议：

1）在程序员趋向于安稳工作的情况下，而且具有大公司所要求的学历、经验、证书、能力的时候，应该首选大公司。这些大公司指的是国内外著名企业，有的甚至是地区内著名企业。进入大公司后，可以保证收入高，岗位稳定。

2）在程序员技术水平一般的情况下，想进大公司也可能是一个梦想，那只好选择小公司了。小公司的好处是起点低、机会多，缺点是收入低、开发不规范。

3）对于有理想和抱负的程序员，我建议先进小公司，然后再进大公司。小公司专业化分工比较粗糙，有的甚至一个人就会负责一个项目，对人的锻炼机会很多，程序员既可以学到编程，又可以学到设计和项目管

理，往往会成为一个“全能型”的程序员，这对程序员以后的发展有很大的好处。但是，程序员在完成编程积累之后，应该转向到大公司发展，学习大公司的软件开发流程、团队意识、大项目的开发经验、规范和管理、企业间的合作以及技术交流和运用等。

4）我们知道软件能否复用是软件小公司发展的关键。因此，我们在面对软件小公司的招聘的时候，我们应当对公司在软件行业应用的范围，以及软件的客户数有所了解，以判断小公司的成长性。当小公司成长性很高的时候，你的选择应该是正确的，当小公司成长性很低的时候，你若选择，就有可能不正确。

5）那些缺乏远大理想的程序员，只是把软件当做普通工作，求得平均工资和正常收入的程序员，最好能找到一些工作较为稳定的部门，以保证有一个稳定收入。这些人即使进入大公司、小公司，都可能面临解雇的危险。最好的情况就是处在一个岗位多年不动，工资多年不涨的局面。

对于有理想的程序员来说，无论进入大公司还是小公司，都是一个积累过程，都是一个锻炼自己的天地。对于普通的程序员来说，进入大公司则实现梦想更容易些，进入小公司则可能存在诸多困难。只是前者取决于自己，而后者取决于别人。而关注公司成长性则是两者都要重视的，毕竟这和自己的当前和未来收入密切相关。

总之，我想说程序员是一个很特殊的职业，它给每个程序员造就了一飞冲天的可能。关键是程序员自己是否有这个想法，能不能在进入各种公司之后，积累和提高自己的技术水平，为一飞冲天做好准备。

1.7 新手面试常见问题和对策

招聘面试是一个用人单位（面试官）与招聘者之间的博弈，不同的应聘者、不同的用人单位、不同的面试官会产生不同的面试过程和不同的面试结果。因此，如何应对面试是很难有什么正确答案的。但是，大量面试沉淀下来的各种成功的经验和失败的教训却是我们可以参考的。

在日常生活当中，常常有朋友和同事找到我，向我询问他们学计算机专业的孩子在面试时要注意些什么。也有一些软件公司因为和我比较熟悉，请我在业余的时候为他们招聘人员把把关，提提意见。在单位的时候，有时候也会带上实习生，也会面临培养新员工的问题。因此，我见到新手的机会相对比较多一些，加之自己也是从新手走过来的，所以对新手相对了解一些。可以说我身兼了被招聘者和招聘者两种角色。客观了解和分析这两者的面试心理可以让我们面试时心态更从容和平和一点。

作为新手，主要是一些刚毕业的大学生，他们很少有编程经历，很少有人能把学的东西与现实对上号。但是他们求职心切，急需通过录用来证明自己的社会价值，证明自己的独立生活能力，因此，他们在应聘中一直处于弱势地位。

作为面试官，肩负着企业招聘员工的重任，知道企业招人的急迫性和招人的标准，当应聘者众多的时候，其招聘要求应更加严格；当应聘者不多的时候，要求就可能放宽。他们拥有招与不招的大权。在招聘中处于强势地位。

新手在面试时常见有以下几个问题：

1. 简历灌水

新手在求职时，往往会投寄和递交简历。这些简历对用人单位了解应聘者的意义重大，这是用人单位对其第一印象。现实中有很多简历言过其实，例如：“精通 C#、Java 语言”，其实远远达不到精通的水平。“开发过某某项目”，实际上仅仅是参与过这个项目，在项目中作用微乎其微。这样言过其实的部分往往会在“掌握语言”、“编程水平”、“英语水平”、“项目能力”、“团队精神”、“学习经历”、“学习成绩”有所体现。有的人明明没有学过 C#，但是他敢于写上自己精通 C#。

灌水的直接结果就是应聘者在面试时，显得很尴尬，显得很不诚实。例如用人单位急聘 C#程序员，看到简历上精通 C#之后，必然会在面试时问及 C#编程情况，在某些情况下，还会出一些 C#编程试题。如果这个新

手并不精通或根本不会 C#，其结果就可想而知了。

很多新手认为，你不写夸大一点，你可能连面试的机会都得不到。但是，到了面试，一切都会暴露的。我认为，在简历上诚实地写清自己的各种情况和水平，这可能更容易得到用人单位的认可。如果自己掌握的东西不够的话，还是应该把应该掌握的东西在应聘之前多掌握一些，把真实的自己展示在用人单位面前。

大家知道，在招聘程序员的时候，面试官中一定会有一个懂程序的高手（用人单位的高手），而这些人自信心很强，具有好斗的潜意识。你说强，他就要让你把强说出来，好让他表现自己。所以这个职业特点注定你不能太夸大自己。

2. 恐惧心理

由于新手处于弱势地位，其恐惧心理很严重，有的大学生社会实践比较少，连起码的见面礼节和打招呼都忘了。有的人在介绍自己的时候话不成句，声音极小，连面试官都听不清楚，尤其是被问到程序方面问题的时候，更是紧张得词不达意。几乎所有新人都有恐惧心理，只是恐惧程度有高有低罢了。恐惧心理使应聘者留给面试官的印象不好，他会认为你社会经验比较少，沟通力差，还是一个学生。

可以通过时间和经历来克服恐惧心理，新手平时要注意这方面的改进，讲话声音尽量大一些，说话要尽可能多，话与话之间尽量连贯，在说话时，眼光要尽量看着听者。新手可以找一些同学、老师、家人当做面试官练练兵，一次不行二次，二次不行三次，多练几次就会好得多。

3. 表现欲强

程序员中也有一些表现欲强的人，这类人动手能力相对强一些，对新技术有追逐的爱好，有的也做过一些项目，自己也赚过一些钱，有的认为自己已经会编程了，有的认为自己已经是高手了，他们在面试时表现出了那种过于自信的谈吐。有的甚至会问面试官知道不知道某个东西，这往往让面试官感到不快。因为在企业工作的程序员或项目经理都知道技术水平

的提高是没有止境的，否则，他们也不会来招聘新人。他们既要看到应聘者的能力，又要看到应聘者表现能力的方式。他们认为，那些过高看待自己的人往往在团队合作上容易出现问题。而且夸夸其谈的人往往和那些没有真实能力的人画上等号。

面试时，缺乏能力的时候，要表现出能力；缺乏风度的时候，要表现出风度。应聘者，切记不要班门弄斧，弄斧可能伤到自己的脚。如果想要表现自己的话，可以在同学、家人、陌生人面前尽情表现，虽然这些表现可能不会影响你的应聘。

不过，有的企业不把表现欲很强的人录用为程序员，而是把其录用为软件销售人员，这种情况在现实生活中也真的出现过。

4. 准备不足

很多企业和单位在招聘的时候对招聘岗位和人员要求都是很明确的。但是，有的应聘者往往对此没有针对性、重点性的准备。例如，某用人单位要招聘有金融软件开发经验、熟练使用 C#语言的程序员，很显然用人单位是要做金融方面程序的，而且缺少 C#程序员。应聘者应该针对金融方面的知识和 C#语言两大方面多做准备，如果你有金融软件开发经历，那就要在面试时突出这个方面的内容，如果没有，你最好多准备金融方面的知识，谈谈对金融业务的理解和自己的看法，表明你对金融软件的了解程度。同理，在 C#方面，也要针对 C#在金融软件方面常用到的技术做些准备。这样有的放矢地回答面试官的问题，总比摆出一副叫我做什么我就做什么的架势要好得多。

许多新手对用人单位的性质、员工人数、工资状况、同业中排名、产品方向、单位特点、发展趋势、可能笔试、面试的内容都无准备，便匆匆应聘，其结果是可想而知的。机会总是留给那些有准备的人。

5. 性格内向

由于编程是一种个人劳动，很多学程序的学生很自然地沉浸于个人世界里，与外界交往和交流并不主动和积极。表现在与人打交道上很内向，

少言寡语，给人看上去很“老实”的感觉。这些人往往对自己很自信，却又埋怨别人不理解自己的能力。在面试时，往往和面试官形成一问一答那种很机械的场景，严重缺乏主动性，从而给面试官留下一种很不自信的印象。除了女人找对象要找“老实人”之外，很少有软件企业要招那种性格极端内向的程序员的。

在当今项目规模越来越大，项目内部越来越需要协调、交流和合作，软件开发更需要团队精神来支持。无论是交流意识、交流方式都是优秀程序员必须具备的。所以在面试的时候，要避免一问一答，要主动地在重点问题上谈谈自己的看法和想法，主动和面试官进行沟通。这样面试官自然会对你另眼相看。我们不排除有的用人单位喜欢雇用性格内向的程序员，但那毕竟是少数。

6. 逻辑不清

程序员最重要特征之一就是逻辑性，凡事要讲逻辑、讲条理，有条理才能成为程序。但是，面试时也常常遇到一些应聘者答非所问。如果你要问他多高，他绝不会回答他1米70，而是可能回答“今天上午我才赶来面试的”。你要问他常用的排序方法有几个，他绝不会回答有3个，第1个是什么，第2个是什么，第3个是什么，而是回答让你不知道他在说些什么，也许其中有那么个排序的名词。对于那些没有什么正确答案的问题，那更是能回答得云里雾里，听不出其中的头绪。可以说这种人说话和思维缺乏逻辑，能把一个简单事情复杂化，能把一个真实事情虚无化。

这种人在理解别人的话的能力方面，在处理问题的能力方面，在逻辑方面都很欠缺。如果不加以注意和改进，那么进入了程序员这个行业后，自己痛苦，同时别人也会跟着痛苦。了解这些后，应聘者在面试时，一定要集中精力，要充分理解面试官提问的真正意图，回答简明而有条理，不懂也不要胡乱说。只有这样面试官才会认可你是做程序员的材料。

7. 潜力不明

用人单位招收程序员一般有两种类型，一类是招来即用的，一类是培

训后使用的。对于前者，用人单位招的是和自己项目开发最接近的程序员，比如之前做过此类项目最好（挖同行的人员），招人的标准很明确，能干则要，不能干就不要。针对第二类，主要是看应聘者的各方面素质，看看是否通过培训后能逐步成为单位的有用之才，所以标准比较宽泛，无论是应聘者还是招聘者都有很大的选择范围。很多应聘者并不了解这些，在面试的时候，没有主动地表现自己各个方面的素质，表明自己是一个有素质和潜质的员工，以获得面试官的认可。

有一点要特别说明，应聘者千万不要过度地表现自己的学习欲望，说一些“到单位之后，我会好好学习的”之类的话，认为用人单位是一个学校，只要当一个好学生就可以了。用人单位要的不是一个学生，不会提供一个免费培训基地，而是让你为它创造财富的。面试官特别不喜欢那些已经走出校门而思想还没有走出校门的应聘者。你不如说“我到了单位后，将多做工作，为单位创造财富贡献自己微薄之力”之类的话，这反而更能让面试官认同。

应聘者的语音语调、着装打扮、递交材料方式、等候面试、介绍自己的分寸、对用人单位的关注度、对自己未来的企盼、对自己可成长性阐述、对自己不懂问题的回答等面试中的细节，都可能是面试官对你产生印象的一个因素。这些因素的综合形成了面试官对你的素质的判断。

面试官可以理解你现在编程技术达不到所要求的水平，但是不能容忍今后你达不到所要求的水平。关键是让面试官看到你的潜质。

总之，新手在面试时，要准备充分，要端正自己的心态，即不要过分地表现自己，也不要恐惧犹豫，说话要条理清楚，注意面试中的各个细节，尽可能地向面试官展示自己良好的职业素质，展示自己未来的发展潜力，这样面试官就可能对你有一个良好的印象，有助于获得这个就业机会。

面试只是应聘者素质的集中反映。素质的形成可以是在面试之前，也可以是在面试之后。只要我们注重培养和提高自己的素质，我们就可以在任何时候面对任何的面试。

1.8 薪水的苦恼

上班了！拿工资了！从学生转变成社会的劳动者，这是人生阶段的一个重要转折标志。当拿到自己的薪水之后，是喜悦、是满足、是苦恼、是无奈，每个人的感觉都是不一样的。我记得自己第一次领到上班的工资不到 10 元钱，兴奋异常，忙着请客庆贺。而今一个新手拿着 2000 多元的薪水估计也不会欣喜到哪里去。

程序员注重自己的收入与其他所有劳动者一样，注重自己的收入是天经地义的事情。只要是为了谋生而工作的（为了其他目的，如爱好、兴趣、自我实现等而工作的，不在我们谈论之列），都会注重自己的收入。几乎没有一个人会认为自己的收入高，所有的人都希望自己的收入越高越好。这个原因主要是人对金钱的追求是无止境的。

现实中程序员对自己的薪水还是有很多苦恼的。

1. 从宏观上看收入

从宏观上看，由于程序员所处的行业或企业不同，其收入的差距确实存在。所以，当程序员看到在其他行业工作的同学所拿到的工资时，就会感叹自己拿的工资太少。我初步估算了一下，新进软件公司的大学生的工资收入一般是当地最低工资的 2 倍左右。

2. 从企业内部看收入

在一个企业内部，由于各个部门或职位分工不同，其工资也存在较大差异，而且在不同的行业中这种差距也比较明显。作为软件公司的程序员一般是公司收入最低的阶层，而作为用户单位的程序员的收入一般处于单位平均工资的中等水平。总之程序员的平均收入水平是不高的。而程序员本身也因工龄、技术能力、项目的不同而收入不同，有的程序员升任到了设计师、项目经理，其收入也能是普通程序员的 5 ~ 10 倍。这种眼前工资的差异也会让新进公司的程序员心中有所不平，尤其是看到和自己一样工作，甚至能力不如自己的同事，拿着比自己高的工资，这种心态尤其

强烈。

3. 从话语权看收入

程序员除了感到收入低之外，还会对自己的薪水制定没有话语权感到沮丧，而且也会为奖金有无，分配问题激动，还会为收入不能满足各种生活支出而感到无奈。大凡与收入挂钩的事都会让人苦恼不已。

现实，现实，还是现实。程序员不但要学习技术，还要学习适应社会，增加自己的社会知识和经验。我们无法掌控薪水的多少，薪水不是靠我们想出来的，就像发财不是靠做梦就能实现的道理一样。所以我们要学会“自己不能左右的事不要多想”，多做些自己能把控的事，例如，有想象的时间不如把自己的工作做好，把自己的学习忙好，把自己周围的人际关系搞好。

从另一个角度来看，新进公司的程序员至少在 3 个月到半年时间内还是处于学习和适应阶段，还没有能力为公司创造利润。从市场经济的“等价交换”原则来看，程序员此时对公司来说是负效益的。而那种只要上班就必须给我工资的想法是一种很天真的想法。在这段时期，如果程序员能想通这个道理，就不会为薪水烦恼了。

建议程序员换一种阿 Q 的心态，毕竟自己要比那些没找到工作的强呀。

有了好的心态，我们可能更加现实，可能更加有利于我们在这个社会的成长。今天的薪水也许很低，但是必须承认我们的水平也很低；我们眼前虽然有收入比我们高得多的程序员，但是总有一天我们也会达到他们的收入水平。如果我们能认识到这点，抓住一切时间去学习、工作、提高自己的工作能力和技术能力，提高自己的社会生存能力，我们可能就会缩短自己的成熟时间，我们也可能缩短新手的低薪水的时间，而增加自己的高薪水时间。

程序员的能力是最重要的，而能力需要得到程序员的理想支持。在程序员这个职业中，心有多高，薪水就会有多高。

1.9 求书、求网还是求人

我们已经知道了新手如何学习一门新的语言，那么对于已经掌握一种或一种以上的程序员如何学习一门新的语言呢？由于程序员已经有一定语言基础，形成了自己的一套学习方法和思维定式，所以这个问题还是和新手学语言有些差别的。

在现实生活中，我发现不少程序员因为工作的需要而学习新的编程语言。他们心怀恐惧，不知道自己能否学好，而且学习方法非常传统，像新手一样：首先去买本这方面的书（或者借本书，或者网上阅读），看完以后再动手。但是，我不知道他们是怎么看书的，是从头到尾地看？还是挑选着看？他们关注书中的什么内容？第二种情况是程序员动手能力特别强，先把语言安装起来，先编起来再说，但是他们不知道这个过程有多长，何时是终点。无论哪一类程序员，当你向他讨教如何学习一门新语言时，大都是仁者见仁，智者见智，而且几乎没有人能从头到尾说清楚。若不信，读者可以自己回答这个问题。

其实，程序员学新语言是有讲究的。学习与自己现在使用的不同类型的语言难度最大，相同类型的语言难度就小些。当掌握过程语言（如 C 语言）的人去新学面向对象的语言（如 C ++）时，难度就非常大，因为这两种语言的思想是完全不同的，用过去的定势去思考新的语言，那种痛苦是难以言表的。例如 VB 程序员去学 VFP、PB 等同是 C/S 类开发程序相对就容易得多。又例如，VB 程序员去学 ASP 就很困难，因为一个是 C/S，一个是 B/S，虽然语法上相差不大，但是架构差距很大，很难马上适应。所以，学习新的语言，第一个要分析这个语言的类型自己是否已经掌握，如果已经掌握，那么学习的时间会很短，一般在一个星期到半个月就行了；如果没有学过，则学习时间会很长，一般要 2 ~ 3 个月或者更长，而且非常痛苦。

对程序员而言，学习新的语言，第一，建议不买什么资料，买了资料也不要花时间一字一句地去看。主要的学习手段就是动手编程序，通过在编写范例程序的过程中学习！第二，一定要找到一个懂这种语言的人，如

果能够找到人问的话，一定要找人去问。这个时候和新手找人问是完全不同的，因为新手会听不懂别人所说的一些基本概念，这会引起高手的不耐烦。第三，程序员千万不要自己去“刻苦钻研”，因为，程序员有一定的语言基础，心中只要记住“流程”、“语法”、“调用”（见1.2节）就可以了。

程序员在掌握编程的流程之后，可以通过编写以下三个程序：

1）显示“hello world!”程序。
2）打印日历程序。
3）“学生学籍信息”处理程序。

来掌握新语言中的语法和调用方法。当然程序员自己也可以选择编写自己的程序以达到学习目的。

与新手不同的是：

1）要特别注意语言的语法差别。一般来说，语言的语法差别很小，但是很令人讨厌。程序员学过之后往往会混淆不同语言的语法。例如，有的语句是以分号为结束符的，有的语言中的语句是以回车符为结束符的。如果混淆了，你就会一会儿加分号，一会儿不加分号的。尤其当程序员对原先语言特别精通时，这种操作惯性是很大的，因此也特别烦人。

2）要特别注重程序的调试方法。因为除语言本身之外，如何调试也可能不太相同。调试对于程序员来说太重要了，必须加以关注。

3）要特别注重程序运行环境。程序编出来了，生成exe可执行文件了，但这并不意味着程序就能执行了。例如，用C语言编写的程序一般是不需要额外环境的，直接运行就可以了；用VFP编写的程序，则需要系统中安装VFP系统DLL，因此，VFP程序要做安装盘；C#则需要更多的环境才能执行等。

4）要特别注重程序中的各种调用。由于新学的语言往往功能更强，涉及调用功能更多，程序员要关注新语言如何调用的问题，调用也是今后程序编写的最重要的内容。

程序员按照这种方法去学习新的语言，刚开始一定会感到寸步难行，可能会遇到很多困难。但是，我相信只要程序员知道整个学习的流程，知道下一步工作的目标，他的进步会越来越快。我最不希望看到程序员在学习上表现出自发、盲目、恐惧的精神状态。

1.10 新手看高手

入门之前，很多程序员心里有一个高手情结。通过书籍、媒体、传说渲染，他们认为 IT 行业是一个高手林立的行业，好像这些高手创造了这个行业的奇迹。这些高手可能是国外的，也可能是中国的。这个高手可能是一个具体的人，也可能抽象于某些著名软件背后看不见的程序员。只知其名，不闻其声，这个时候的高手是一种无所不能的神，一种虚幻，是令程序员崇拜的偶像。

到了工作岗位之后，这种高手情结更加严重，由于新手发现自身技术水平有限，而内心想尽快摆脱这种状况，使得很多新手对高手感觉更加恐惧和渴望。我发现在 IT 行业中，中国程序员认为外国的程序员是高手；网下的程序员认为网上的是高手；搞硬件的人认为搞软件的是高手，搞软件的认为搞硬件是高手；搞应用程序的认为搞系统的是高手；搞界面设计的认为程序员是高手，程序员认为搞界面设计的是高手；年轻的人认为年长的是高手，年长的认为年轻的是高手；企业内的程序员认为企业的外部程序员是高手；客户单位内部的程序员认为软件公司是高手，软件公司的程序员认为客户单位内部的程序员是高手等。所有这些高手都是建立在程序员本身对某个领域无知或掌握不精的基础上。因此，高手的多少是和程序员的水平成反比的，如果程序员水平高，那他眼中的高手就会少；如果水平低，那他的眼中的高手就会多。所以，不同程序员对待高手的标准应该是不同的。

高手情结是新手的正常心理现象，它反映了程序员对未知领域的向往，说明程序员还有远大的理想，还有激情，还有奋斗的目标。关于高手，我想新手应该关注以下几个方面。

1. 向身边的高手学习

身边的高手，一般是新手的第一个师傅，也就是第一个培养他的那个人。这个人对新手来说是一个决定其技术命运的人。程序员很多人可以忘记，但是第一份工作，第一次带他的人是很难忘记的。假定这个师傅带他的时间有一年以上，我想从这个程序员身上一定能够看到其师傅的某些影子。比如说，这个师傅喜欢的语言，这个师傅喜欢的编程格式等，都能在新手身上找到。

作为新手来说，除了要向他的师傅学习之外，也要把身边的同事看做高手，要向他们学习，只要单位或团队范围内有比自己强的，不管这个人是早于自己进入这个单位一天，哪怕只是一个函数、一个语句、一个调用、一个算法都值得去讨教和学习。这个时候学习是很实在的，是能解决具体问题的。

向身边的高手学习，就是要充分利用身边的优势，可以面对面地接受别人的技术传授。新手千万不要顾及面子，或不好意思，或怕麻烦别人，能请教的就赶快请教，学习是硬道理。向别人请教不仅仅是去弄懂某个问题或解决某个问题，更重要的是培养与人沟通和建立良好人际关系的习惯。

新手千万不要有那种有人会主动帮助自己的幻想，一方面是大家工作都很忙，另一方面很多程序员不习惯于主动帮助新手。

要注意身边的高手并不是所有问题都能解答的，遇到这种情况，新手不要吃惊，很多“伪高手”就是在这个时候被揭开其伪装的。对于他们，新手要怀有宽容的心态，应坦然面对，不要深究，这个人不能解答，那就去请教另一个人。新手要记住，你请教的不是高手，而是解决问题。因此，向其请教的人是否是高手并不重要。

新手千万不要太迷信不可触及的高手，因为迷信了也没有用，而且也不会对你的实际工作有什么帮助，而应该把对那些高手崇拜的时间和精力用在身边的工作上，这可能更有成效。只有当自己有了一定进步之后，确

定了自己发展方向之后，我们才有本钱和资格向外部高手学习。

2. 学习高手的技术

（1）软件制作流程

我记得很多新手并不会安装操作系统、不会安装开发环境、不会配置网络、不会新建项目、不会新建程序、不会编写程序、不会运行程序、不会调试程序、不会制作安装盘等。即使会也只是知其然，而不知其所以然。而一般的程序员对此驾轻就熟，而且没有太多变化，相对机械，新手死记硬背就行了。因此，新手首先要把编程的流程搞清楚。把流程搞清楚之后，我们就可以把精力放在编写程序上了。

（2）语言的掌握

在程序的编写方面，新手常常遇到的就是语法问题，如语句使用不当、变量问题、格式问题、备注问题、命名问题、函数调用问题、参数问题等，这些问题就不那么简单了，不同的程序员会给出不同的解决方案。当解决这些问题之后，只能说是新手刚刚学会编程的工具而已，接下来就要学习如何运用这些工具来开发项目了。

（3）开发项目

这个时候新手就会遇到需求问题、功能问题、处理流程问题、数据结构问题、算法问题、可靠性问题、边界处理问题等，针对这些问题，不同的程序员更是有不同的观点和看法，所以，新手要多听不同的解决方案，通过比较来加深对这些问题的认识。

（4）高级技术

以上就是满足开发的基本要求了，但是，能运行的程序并不是一个好程序，只是站在一个项目的角度上来看技术，也不是最好的技术。因此，新手要更进一步提高自己的技术水平，还要在程序的架构、接口、参数、共享、安全、效率、交互等与程序密切相关的问题上进行探讨和研究，而在这方面有所建树的程序员已经不多见了。所以，新手要多留意，自己也

要多积累，在这些方面多下工夫。

3. 学习高手的方法

具体的问题是永远解决不完的，尤其是编程。遇到的问题可以说是千奇百怪，无所不有，因此，新手不能把全部的精力放在学习解决问题上。新手要在学习具体问题的解决方法前提下，更要学习高手们是如何解决这类问题的。通过方法的学习，可以使新手有很大的进步。例如，高手向新手讲解一个具体程序的调试方法。新手除了掌握这个程序的调试方法之外，还要学会跳出具体程序，掌握好调试基本流程、主要调试命令和注意事项。只有这样，当新手再次遇到调试问题的时候，就可以不用再找高手指导了，可以用这种方法自己来处理了。

高手的另一特点就是站得高，看得远，一般新手的问题，他们都很轻松摆平，所以，新手也要在平时看问题时，尽量地跳到具体问题之外，站在更高的层次上看待具体问题。

4. 学习高手的激情

高手除了技术水平高、工作方法好之外，还有一点是新手要学习的，那就是他们的激情、坚持、专注。高手之所以是高手，一定付出比常人更多的劳动和辛苦。而这种付出并不像工作那样是强制的，往往是他们心甘情愿的，而且乐在其中。新手可能对老程序员没日没夜地连续工作感到不可思议，但是一旦他们也这样做了，他们同样也会感受到那种付出后获得成功的喜悦。

在遇到困难的时候，新手要向高手学习那种永不放弃的精神，不战胜困难就绝不罢休。高手之所以是一个高手，那是在某个领域，他能比别人有更大发言权，因为他比别人更了解这个领域的客观规律，这个规律是他长期专注和研究的结果。因此，新手遇到各种问题时，要沉下心来，注意观察问题中的细节，不放过任何疑点。只有这样，新手对问题的理解就会比别人更加深入。

5. 不迷信高手

其实，现实中并不存在完美的高手，高手总会存在这样或那样的缺陷。新手不要对高手一味盲从，编程好，不一定界面设计好；编程好，不一定设计好。所以，新手要有自己判断是非的能力。例如，高手让新手按照某种要求编写程序，而这种要求本身就可能存在问题。当新手发现这个问题时，就应该停止编程，找出问题的原因所在，最好能找出解决方法，向高手说明情况，争取高手的理解和支持。这样高手就能看到新手的进步。而新手也可以在自己发现的新方法的过程中找到自我价值。

虽然我们是新手，但是，面对高手我们无须恐惧，我们会发现，随着时间推移，自己看到的高手会越来越少。这说明自己的技术水平在不断提高，当有一天有人开始请教你问题的时候，尽管你不处在高手的最前列，但是，你已经步入了高手行列。其实，面对高手就是面对未来的自己。

1.11 新手应该具备的基本素质

新手走过的路，我们都走过了，别人走过的路，我们也看过了。回首往事，我们有过很多经验和教训，但是，几乎没有人会刻意告知我们应该怎么成长，更没有人专门对我们进行素质上的要求和引导。直到今天我们才知道我们的成长是自发的。它完全取决于我们的工作环境和内容，取决于我们遇到的人和开发的项目。假如我们能够回到从前，我们一定会在年轻的时候更加注重培养自己的素质，让我们成长得更好更快。我将与程序员有关的素质方面的要求进行了一个系统性的归纳，希望这些会对程序员有所启发。

1. 激情

激情是程序员的职业标志，在很多情况下，激情具有递减规律。也就是说新手的激情最高，越到最后激情越低，甚至麻木和消失。而成功的程序员几乎都是充满激情的，他们能将激情始终保留在身边。我们要对年轻的程序员说，请保持你们的激情，无论你们在工作中遇到多大的不满、委

屈、挫折、失望都不要丧失你们的激情，只要你们有了激情，你们才能东山再起，才能勇往直前，才能达到事业的顶峰。

2. 学习

无论是新手还是优秀程序员，无论是程序员还是其他职业人员都应该注重学习，人只有在学习中才能增加自己的知识，才能将更多知识用于自己的工作。针对程序员这个职业，由于其涉及软件技术、项目管理、用户的业务知识等方方面面，而且这些方面的知识还在不停地变化和更新，所以只有学习才跟得上职业发展的要求。刚开始的时候，我们发现若不学习，那就什么都不会，我们只好学习。后来，我们发现即使学习了，还是有很多东西不会。当我们把学习看做一种常态的时候，我们就会不断获得新知识，这样才会适应职业要求。

3. 基础

“万丈高楼平地起”，这充分说明了基础的重要性。程序员在开始的时候，并没有感觉到基础的重要性，但是随着程序员不断成长，这种基础的制约现象就会很明显。很多程序员总是感觉自己的进步不大，进步不快，不知道自己的问题出在什么地方。其实，很多最根本的问题是大学生的基础问题。因为很多大学生在学校的时候，并不知道所学的基础知识用在什么地方，有什么看得见的重要性。很少有人会对基础知识有真正的理解。到了工作单位之后，一些程序员在编程的时候，复制网上的程序，还是知其然不知其所以然，基础越来越差。不单是知识基础，其他基础也很重要，例如，一些新手开始不愿意学习盲打，不肯在开始学打字的时候练习盲打，基础没打好，结果若干年后，只会一个手指头按键盘，被人们笑谈为“一指残”。程序员要有基础意识，要把现有的取得的成绩当做基础，只有把现有的基础打牢，这样才能向上更好地发展。

4. 好胜

没有一个程序员没有好胜之心的。但是，好胜有强有弱，有的程序员大有舍我其谁的气概，有的程序员则不露声色暗下工夫比高低，有的则看

不出来是否好胜，一副好坏与己无关的样子。有的因能力而好胜，有的因面子而好胜。作为程序员好胜是必需的，因为程序员是一种智慧劳动，要比就是比智慧。只有通过智慧的竞争，智慧才能精彩，程序才能更加漂亮，软件才能向前发展。好胜意味着要比同伴做得更好，好胜意味着遇到困难必须克服，好胜意味着自己要比自己原先做得更好。我们要的是好胜的结果，我们不需要的是好胜的表现。其实，不要刻意表现你的能力，你的成果足以表现你的能力了。

5. 动脑

软件本质上是一种智力的产物，这个智力并非天生就有的，它和程序员的技术能力、理论基础、思维方式、知识范围、周围影响等因素密切相关。只有通过对这些因素进行收集、存储、加工、处理，进行各种各样排列组合，形成各种解决方案，然后在这些方案中进行择优判断，才能得出最后的解决问题的办法。程序员除了动脑还要通过打键盘编写程序，所以程序员还有一个动手的工作。很多程序员习惯于边想边编，久而久之就养成了动手习惯。从成长的眼光来看，程序员应更加注重开动脑筋，而且要把其和动手编程分离开。这样养成动脑的习惯之后，对编写程序有很大的帮助。一般而言，动脑的时间越长，动手的时间越短，程序员技术水平就可能越高。

6. 外向

程序员性格对成长有很大的影响，无论是原来外向还是内向性格的程序员，只要从事这个职业，只要和计算机打交道，其性格都会有点向内向方面转变的趋势。我们看到不少程序员不善于和别人交谈，怕见人，甚至从内心里不愿意和别人交谈。这样的自我封闭其实对自己并无益处。我认为程序员性格要外向一些，要乐于和别人交流，要主动和别人交流。和别人交流并不一定要限于技术，用户的需求、公司成长情况甚至对方个人生活等方方面面都可以进行交流。这种外向的性格可以让程序员见识更多的人，见识更多领导，见识更多的高手，见识过去不敢见识的人。这样可以积累更多的社会关系。

7. 技能

程序员的技术能力是程序员生存下来的基础，而技能从某些方面来说是首要的。可以说程序员就是技术的代名词。有了好的技能你就能更好地开展自己的工作，有了好的技能你就有了和别人交流的内容，有了好的技能就可能让自己获得更高一层的技能。因此，程序员要重视自己的技能学习和提高，要在技术上体现自己的能力，要通过技术能力去影响或帮助自己的同伴。只有这样，程序员的最基本的价值才能真正体现出来。很多程序员到了一定阶段放松了对技术的追求，技术平平，只能满足现有工作，这就愧对程序员这个称号了。

8. 团队

程序员可以一个人编程序，但是，一个人只能编写一些程序或小规模的程序。有一些程序员一个人完成了整个系统开发，其技术水平固然值得称贺，但是，如果他认为软件可以一个人摆平，这就会影响到其向更高水平发展。随着软件业发展，软件的规模也在变大，软件制作的专业化程度变得越来越高，一个人即使有这个能力开发一个完整的系统，最好也不要一个人去完成，只有通过团队的分工协作，软件制作才能走向正道。如果程序员一开始就能注重团队意识，一开始就认为自己仅仅是团队的一分子，一开始就注重与其他团员的沟通和协作，这样程序员一定能融入团队中，而团队的巨大作用是每个程序员个人作用所不可比拟的。程序员不要只是享受自己独自编程的快乐，而是要享受整个团队编程的快乐。

9. 兴趣

程序员的工作内容看起来似乎只是一行行代码。代码的编写来自于程序员头脑的指令，而程序员头脑中的指令并不是来源于程序设计语言，而是来源于各种需求、各种问题的分析方法和处理方法。因此，程序员要编好程序，不仅要对程序语言感兴趣，而且要对所有和编程序相关的事物感兴趣，甚至对看上去与编程无关的东西也要感兴趣。我们成长后才知道各种事物都是相通的，有些东西会潜移默化地影响到你的分析方法和处理方

法中。所以，程序员不要只关心编程，这样只会成为“书呆子”。我们日常生活中的看电视、听新闻、上网聊天、社交活动等都应放在兴趣之中。只有广泛的兴趣才能体会到你最爱的兴趣。

10. 谦和

我喜欢程序员有一种谦和的精神，尤其是那些有本领骄傲的程序员的谦和。我们常常看到一些得意于自己的程序、自己的项目、自己的收入、自己的职位而忘形的程序员，他们无视别人的意见和建议，有的甚至自傲，看不起其他程序员。但是，要知道艺无止境，假定软件领域拥有顶峰，而谦和正是通向这个顶峰的一个阶梯。更何况软件领域没有顶峰，更需要我们的谦和来表明自己不拘泥现有的成就，我们还有更高更远的理想。

11. 求新

我们常说软件技术发展太快，跟不上就会被淘汰。回顾几十年来的软件技术发展就可以明白这一点。求新本质就是注重时代的变化、跟上时代的变化。因此，对程序员来说，要有求新的意识，不要排斥新生事物。要通过网络和各种媒体注意各种新技术的产生，对于与自己工作相关或感兴趣的技术要花点时间进行跟踪，要了解这些新东西的新方面，要学会新旧对比，对自己必须掌握的新技术要毫不犹豫地花时间拿下。求新会让程序员时刻感到压力，但是求新又让程序员能看到自己未来成长的方向。

12. 主动

主动绝对是程序员的一个良好素质。我和许多年轻的程序员打过交道，有主动型的与被动型之分，大凡学习上主动，工作上主动，甚至是劳动上主动的人，大都会得到同事的喜爱，都会得到更多的回报。而那些虽然听话，但是“不说不动”的人，因为缺乏主动意识，只能被动听从别人的安排。别人给什么，自己就吃什么，就像一只填鸭，当哪天没有人给你填食的时候，结果就可想而知了。“会哭的孩子有奶吃。”这是很有道理的一句话。

13. 吃苦

现在的程序员和我们那个年代的程序员有很大的不同，他们很多都是独生子女，家庭娇惯比我们那个年代多了很多，而且绝对的自我。因此，他们眼高手低，怕吃苦，不能吃苦。对于程序员来说，连续工作是一种很常态的事。有的吃不了程序员的苦，因苦而选择离开。我们不能说吃苦是一件好事，但是，一个人能吃苦说明这个人不怕困难，有坚韧不拔的意志。这对程序员成长是很有帮助的。想想当年，自己要是怕吃苦，可能也不会有今天的成果。当一个人没有苦吃的时候，说明这个人真的没有发展机会了。而且吃苦的人更能感到甜的味道。

一个人的素质是其成长的基础，良好的素质一定是后天培养的，是后天自我约束和完善形成的，这种约束和完善的内在要求使得他在做任何事的时候趋于合情、合理，容易获得进步和成功。同时，一个人良好的素质可以给外界一个良好的形象，而外界因此会给这个人更多的关注和鼓励，外界的影响反过来又促使这个人进一步提高素质。相同的时间和相同的环境，不同的人因其素质的不同会有很大的不同。因此，如果我们在入门阶段能够重视自己的素质的培养，知道自己在什么方面需要改进和提高，程序员一定会少走弯路，成长更加顺利，而其中的良好素质将影响程序员的一生。

很多程序员对以上方面也有所了解，也知道应该怎么做。但是，涉及自身时，说归说，做归做。如果是这样的话，缺乏自己约束的程序员也只能“自发”地成长了。

第二部分
Part 2

成 长 篇

2.1 加班，加班，加班

在从事程序员工作之前，虽然工作时偶然有过加班之说，但那仅仅是偶尔而为之，并不会太在意。后来，无论是从自己的亲身经历，还是从周围同事的经历，或者是从亲眼所见的各个软件公司的开发过程，更或者是从可听、可读关于程序员加班的传言，一切的一切都证实了：对程序员这个职业来说，加班是一种常态。如果你去问程序员有哪个没有加过班的，举手的一定是第一天参加工作的程序员。

度过了入门阶段的程序员，由于他们具备了编程能力（尽管这个能力并不是太高，技术并非很强），大量的工作或者说是超负荷的工作，就成了这种能力提高的“体能”训练。有的程序员加班实属不情愿，很无奈。因此，他们对加班叫苦连天，埋怨很多。所以，很多人认为程序员是吃青春饭的职业，年龄一大就承受不了加班的压力了。

加班有两种类型：一种是单位因工作而要求的加班，是强制性的，称为被动加班；另一种是程序员自身习惯性延续工作所产生的加班，称为主动加班。

程序员加班发展趋势是：开始阶段是主动加班时间不断增加；第二阶段是单位强制性的被动加班时间不断增加，主动加班时间减少；第三阶段是被动加班时间逐步减少，主动加班时间逐步增加；第四阶段是自己和单

位都不要求加班。

第一阶段：程序员刚进入程序员这个行业，需要大量的自学时间，以提高自己的编程能力。无论是看书还是编程的热情和压力都很大，而且程序员的思维惯性很强，不容易突然中断，加之程序员是个人劳动，没有外人的约束，所以很多程序员喜欢在无人干扰的夜深人静的时候开始工作，这就造成了晚睡晚起，工作没有规律，养出来职业“加班”习惯。这个时候的加班特点是主动自愿的。

第二阶段：当程序员具备了一定的工作能力后，单位完成开发任务的时间一般要求都比较急，所以，一般在正常工作时间是完不成的，程序员因而被要求加班。不同的单位因开发项目的多少，项目要求时间的松紧，会要求不同程度的加班。这种加班一般会伴随着程序员成长的大部分时间。许多人因加班而抱怨程序员工作太苦而转行。在这个阶段中，由于工作的疲惫，个人主动加班的时间会大大减少，但是，当单位要求某个工作在某个时间点完成的时候，尽管不明确要求加班，个人还是会为完成工作而加班。

要特别指出的是，这个时候的程序员已经很少有精力和兴趣像刚开始时那样去学习和研究自己感兴趣的编程问题了。

第三阶段：当程序员经过成熟期之后，其资历和能力有了提高，很多人转向了软件设计和项目经理角色，“被加班”情况大大减少，往往是要求其他程序员加班，大部分时间是“陪加班”，加班的工作强度大大弱于以前了。

所以，这个阶段有的程序员会由于自身能力提高的需要，又开始增加自身加班的时间了，但是，其程度和开始阶段相比要弱了很多。

第四阶段：程序员发展到最后，要么转行，要么跳槽，要么升职，自己学习的激情消失殆尽，无论是工作还是自身都不需要加班了。

当我们严肃地面对加班问题的时候，我们也会思考加班带来的东西。

1. 加班很自我、很享受

当主动加班的时候，是程序员最自我的时候。夜深人静，没人干扰，思绪和编程效率很高，尤其是把一个问题看懂的时候，把一段程序编写出来的时候，把一个出错原因查出来的时候，程序员会很有成就感。此时加班令人兴奋和享受，令人忘却了加班带来的疲劳。

2. 加班环境很宽松

当整个项目组一起加班的时候，因管理要比白日相对宽松，项目组成员会在加班的时候交流、吃夜餐、听音乐、闲聊、等待、观看、学习、小睡等，大家更加随意，比平时人与人之间的关系更容易相处。很多时候，人与人之间的信任就是这个时候产生的，而且这时更容易感觉到团队的精神和力量。当一个项目完成的时候，最能让人难忘的是那些加班时发生的故事。

3. 程序员水平低

某些时候，加班并非都是项目经理和项目本身造成的。有的时候是由于程序员自身水平造成的。有些程序员水平不高，一个高手几分钟就能解决的问题，他们可能要一天或是几天才能完成。所以，从宏观层面来看，如果程序员的水平能提高的话，其加班时间也会相应缩短。

4. 鞭打快牛

说句笑话，也许你能力越强加班就会越多。在任何单位，“鞭打快牛”都是很普遍的。但是这种打其实是很舒服的，只是当时有点疼罢了。当过了一段时间之后，没有人鞭策你的时候，你就可能享受不了那种“痛感”了。因为你的“价值”没有了，人家不“利用”了。在这个商品社会，当你有价值的时候，别人才会和你进行“等价交换”。所以，让自己价值最大化才是员工职场的最大目标。

5. 加班费

加班中最让人关心的事情就是加班费的问题。有些单位在加班费问题

上总是不尽如人意。程序员能反映一定要反映，能争取一定要去争取，不能争取到钱也要争取到假期。说起来容易，做起来不易。程序员对自己无力改变的事，只能放宽心态，努力提高自己的技术水平，只有这样我们才有可能避免“免费加班”。

总之，加班是程序员这个职业的一个普遍现象，它主要由编程工作的不可中断性所决定。加班说明我们有软件需要开发，加班越多说明我们的软件市场越大，加班越多说明我们的软件开发竞争激烈程度越高。当有一天我们不加班的时候，我们就要怀疑我们的饭碗还能否保住，程序员这个职业是否还会存在下去。而我们主动加班却代表了我们对程序员职业的一种兴趣、爱好和追求。我们在加班中付出了劳动，我们就要在劳动中寻找我们的快乐，去寻找和延续我们的梦想。

2.2 大量编程带来的快乐和烦恼

程序员成长阶段面临的工作就是编程，而且是大量编程。这和以往自娱自乐式的编程是完全不同的。这个时候的编程是职业，是和薪水以及被用户认可的软件联系在一起的。编程少了则无法深入体会各种快乐和烦恼。

程序是程序员创造出来的产品。当然程序越多，说明程序员生产能力越强。据统计，1～3 年之间，程序员一般可能要编写 5 万～20 万行代码（包括编写后删除的）。很多程序员都不太清楚自己从业后到底编写了多少行代码，如果真的有心统计，自己绝对会被吓一跳。从代码行数来说明程序的工作量还只是机械的统计方法，很多程序代码很少，但是花的时间却很多，主要是含金量很高。

程序员马不停蹄、加班加点编制了大量的程序，他们快乐吗？如果没有快乐，难道他们真的只是为了薪水而这样不快乐地工作吗？如果有快乐，难道他们真的以乐为苦，故意喊着工作劳累以博取别人的同情吗？

以我几十年的编程经验来看，编程是快乐和苦恼相互交织的一种工作，而且是一种挑战性的、超越自己的工作。对于我来说，其快乐要远大

于苦恼，否则，我早就弃它而去了。

编程会给程序员带来快乐。这种快乐有时只能意会，是无法用言语、文字表达的。那么编程会给程序员带来怎样的快乐呢？

1. 成就感

我问过很多程序员同样的话题："编程的最大快乐是什么？"得到的回答几乎都是"成就感"。成就感意味着觉得自己做了一件了不起的事，做了一件非常有用的事，做了一件有价值的事，做了一件别人做不了的事。程序编多了，无论是编程的结果还是编程的过程，都会让人产生这种感觉。尽管有的软件项目是分拆给好几个程序员的，但是，就其工作性质而言还是属于个人劳动的范畴。程序员的很多工作在某种程度上都是个人劳动。编写一段代码、一个函数、一个模块、一个软件都是程序员自我实现的过程。每当程序员完成这种过程后都感到如释重负，有一种"我终于成功了"的感觉。

2. 被认同感

程序员原来对程序具有无知、恐惧的心理，而通过大量的编程将逐渐克服这些问题。程序员的自信心也会逐步强大起来，而周围的同事往往比他自己先一步看到这种进步，从而率先对他进行认同。尤其是自己原本初来乍到，水平、能力不能充分展示，自己内心也很着急，但是同事并不当回事，对自己不温不火的。随着工作的开展，自己的能力逐渐显现出来，同事也开始转变对自己的看法，从各个方面或明或暗地表现出对自己的认同，这种认同往往会让程序员内心涌出一种满足感。尤其当程序员的上级甚至老板表扬自己工作成果的时候，这种被认同的感觉让人有一种归宿感。甚至用户对自己的认可都会让程序员倍感高兴。

3. 团队氛围

程序员在成长中，一定会和其他程序员以及项目经理打交道。每个程序员和每个项目经理由于个性、能力、经历的不同而与之交往的方式和结果也会不同。随着时间的推移，程序员在这种不断交往的过程中，增强了

团队的意识，增加了软件开发中团队的凝聚力。程序员在团队中一方面能够获得团队成员的帮助和支持，另一方面作为团队的一分子，也在为团队整体作出贡献。每当一个项目在千辛万苦的工作之后完工时，那种团队集体相拥的开心是难以言表的，有的男女甚至因此而结缘。也有个别程序员不能处理好与其他同事的关系，那工作起来就会感到很别扭。

4. 技能熟练

在编程初期，程序员编起程序起来可以用“一步一个跟头”来形容，编程速度慢得不可想象。随着大量积累编程经验，程序员逐步找到编程的工作流程和窍门，编程速度会大大加快。到后来他们几乎到了“兵来将挡，水来土掩”的境界。原来要好几天才能编好的程序，现在只要几分钟就摆平了。有时这种熟练程度连自己都会不敢相信。

5. 学生变老师

程序员开始工作的时候绝对是一个学生，在工作中慢慢由学生变成了老师，而在其后进入职场的则当起了学生。当学生们问起自己曾经问过上一任老师的问题的时候，那种老师的优越感不由你不产生，不由你不认真去解答。有的甚至有主动教学的冲动。

6. 扩大朋友圈

编程多了，项目自然就多了；项目多了，接触的人也就多了；接触人多了，程序员交友的机会就多了。程序员在这个过程中，无论是和程序员同行、软件设计师、项目经理、上级主管、公司老板、用户、合作伙伴甚至是网友都会有所接触。许多程序员因工作需要经常在用户单位进行开发和维护，这样与用户打交道的机会很多，因此，有可能会结交用户朋友。在 IT 人员稀缺的年代，有时用户会对看中的程序员挖墙脚，在项目验收后，程序员由乙方变成了甲方。

当然编程也会带来烦恼，而且这个烦恼因人而异，各不相同。

1. 遇到问题

程序员最大的烦恼就是会遇到问题。编程遇到的问题可以说是千奇百怪。常见的问题就是不会编、编不好、调不通、运行错、查不出错、效率慢等。尤其遇到那些无从下手，查不出问题，同时又找不到懂行的人来帮助的时候，最为心急和苦恼。这些问题困扰越深，一旦问题解决后程序员就越兴奋。

2. 加班劳累

加班是一件很劳累的事，尤其是连续加班更是如此。许多时候，我们都以不愉快的心情在加班，无论是从效率还是从最终结果来看，都不是一件太好的事情。程序员更应该劳逸结合，累的时候，休息一下，保持自己头脑的清醒，这样才能编出好的程序来。

3. 编程厌倦感

编程如同开车，刚开始时感到新鲜，有冲动、过瘾，有一种如痴如狂的感觉。过了一段时间后，累了、疲了，竟会有一种不想开的感觉。编程多了也是如此，可能有一段时间你见到程序就觉得疲倦。那真的是编写程序过多了。程序员到了这个阶段，就要特别小心，因为这种厌倦感加上周围发生的其他事，会产生一种合力，让你有一种强烈地要求离开这个职业的愿望。

4. 技术水平提高慢

虽然程序编了很多，编程速度也很快，但是，很多人在时间压力下，往往只关心能否编出来，而没有时间关心编写的程序的质量。复制粘贴、复制粘贴，你都没有时间和心情去品味程序的好坏，去思索程序的优劣。很多程序员尽管编了好几年的程序，你若去问他，编过什么让自己或是别人叫好的程序，他一定不会给出直接肯定的回答。时间有序地增长，而程序员的技术水平却不见得在提高。能提高技术水平的程序员一定是有心要提高并追求提高的程序员。那些把编程序纯粹当做混饭吃的程序员，100

个中有 100 个的技术水平不高。

5. 收入和劳动不成正比

每一个善良的劳动者都希望自己的劳动和收入成一种正比的关系。但是在现实中，你付出的和你得到的很难成正比。这种情况在程序员工作的初期特别常见。因此，很多程序员都心生怨言。也有少数水平高的程序员拿着相对高的工资，但是，我想这些程序员过去也是当过几年“媳妇”的，否则，他们也熬不成这个“婆婆”。

面对金钱，我只能说一句话，抓紧时间把自己的能力提高，扩大自己的就业机会，是金子总会闪光的。任何企业都不会拒绝能为他们创造财富的员工。

6. 跟不上新技术发展的步伐

从职业生涯的角度来看，程序员最担心的是自己跟不上技术发展的步伐。程序员这个职业的特点就是技术更新快，无论是编程环境（计算机、网络、操作系统、数据库等）还是程序设计语言都在不断地升级和更新。一些环境和语言在不知不觉中就淡出了人们的视野，一些不为人知的名词和概念不时闪出，令人眼花缭乱，目不暇接。而新的东西往往代表着未来，所以，程序员担心自己掌握不了新技术，担心自己的未来，而且这种担心贯穿于整个程序员的职业生涯。程序员编程序多了，没有时间和精力去了解新的技术，不去了解新的技术，只能用原有语言进行编程，如此形成一个非良性循环。每循环一次，程序员的心事都会加重一点儿。有的因为这种担心而放弃了程序员这个职业。

其实，大量编程会给程序员带来大量的快乐和苦恼，我们很难全部列举出这些快乐和苦恼。这个不是你自己愿不愿意的事，而是工作的需要所致。任何一个程序员从入门到成长都必须经历这个阶段，这个阶段将会为今后的成熟阶段和优秀阶段打下一个重要的职业基础。我们面对编程要怀有一颗快乐的心，无论多么烦恼，我们都要坦然面对。只有这样，我们才能真实享受编程中的种种快乐。否则，当烦恼超过快乐的时候，我们所有

的快乐将会化为乌有，我们所有以前积累的价值将回归于零。

2.3 需求总是变化，程序总在修改

当程序员学会编程序之后，开始慢慢地进入项目开发的状况。随着项目开发越来越多，编程也变得熟练起来，程序员渐渐地把学习和工作的重心转移到客户的需求上来。通过与用户进行交流，对用户需求有了由浅入深的感觉，并且越来越感觉到了用户需求和编程之间的密切关系后，如果对需求理解比较深而且理解比较正确，程序编起来就感到特别顺畅，反之，编起来总是担心程序编的对不对。程序员慢慢地就形成了由用户来验证程序正确性的习惯了。即使自己编写的程序很正确，心里还是没有什么信心。

这些仅仅是一个开始，让程序员感到头痛的是，明明上午说好了这样做的需求，到了下午又要改成那样做了。这样编好的程序又要修改或是重写了。这种情况在项目的开发阶段程序员还可以忍受，很多情况下项目到了测试阶段甚至是上线阶段的时候，用户还会提出功能变更的要求。另外，项目投产运行后，用户还会有变更维护的需求。

面对用户需求的不断变更，程序员倍感头痛但也无可奈何，抱怨归抱怨，还是只能按照用户的要求去做。因为用户需求始终掌握着程序员程序的生杀大权。很多软件公司、项目经理、程序员面对不断变化的用户需求，想尽各种办法，希望需求能够稳定下来，需求能够不变。他们想通过用户需求书，通过用户对需求书的确认，通过增加需求变更手续环节和增加需求变更的费用来约束用户对需求的变更。但是，实际收效甚微，很难有不变的用户需求书。用户需求不断改变不是什么个案，而是一个普遍现象。

1. 用户需求不断变化是一种客观规律

很多程序员想当然地认为，用户需求是不应该变的。如果这种观念正确的话，那么在现实中，程序员看到过有多少不变的用户需求？可以说，项目越大，用户需求变化的概率就越大。至于很小的程序、不复杂的程

序，用户需求不变是有可能的。但是，用户需求变化是很正常的事情。

用户需求不断变更应该是一种规律，它是和企业经营不断发展而导致企业信息化发展相一致的。由于市场竞争和企业发展的需要，企业经营必须随时进行调整和变革，以适应这种变化。而企业信息化则是这些调整和变革的集中反映。例如，2003 年中国人民银行提出反洗钱的要求，作为金融机构的银行必须接受央行的监管，就必须开展反洗钱工作，从而导致银行业各种反洗钱系统的开发。如果一个企业的经营方式没有太大的变化，这表明企业发展已经很正常、很健康了，而这种现状一定是原先信息化带来的结果，而当初的信息化一定是在一种不断修正中成熟的，由此也可以想象其用户需求变化之多、之快了；而到目前阶段，企业经营走向正常那就可能不用再进行软件开发了。如果一个企业的经营方式发生了变化，那就必须采用信息化的形式来实现这个变化，而这个变化是一个过程，信息化一定不会等到这个变化完成之后才开始。比如，当初银行有网上银行的设想时，银行不可能一下就能考虑到这种方式的方方面面，它需要走一步看一步，通过前进来修正前进中的问题。所以银行业的网上系统是一个不断完善的过程。我们可以回头看看，当初网上银行是什么样，然后看看现在网上银行是什么样，那就会知道，网上银行业一定是一个不断完善的过程。因此，在企业信息化过程中就会面对用户需求的不断变更。

2. 程序员要调整心态面对这些变化

既然用户需求不断变化是一种客观规律，那么程序员在心态上就要进行调整，将用户需求不能变化调整为用户需求一定会变化，树立起“变正常，不变不正常”的观点。有了这样的心态，我们就不会去抱怨用户需求了，就会对用户需求的变更有更多的理解。有时候，我们希望用户需求变更，如果现在不变更将来还是要变更的，而往后变更的代价要比现在变更的代价大得多。举例来说，如果一个用户在系统开发期内提出一个需求变更，程序员就可以在开发期内解决；如果用户在系统上线后提出同样的需求变更，那这个变更可能对其他程序产生重大影响，甚至对现有系统架构产生影响。更有可能发生的是，当初编这段程序的程序员已经离开了，现在又要有一个新人来实现这个需求。可想而知，后一种变更的代价的确太

大了。所以，用户提出需求变更对程序员来说不是一件坏事，而是一件好事。

3. 程序员要积极主动地面对需求变化

（1）理解用户需求

程序员编写程序时，理解用户的需求很重要。有的程序员凭需求书或自己的想象去理解用户需求，然后就开始编写程序了，编写完之后，经用户检查才发现需求理解错误，程序只好重新修改。这种情况不在少数，推翻原来程序重新编写的事情常有发生。因此，程序员要能正确理解客户需求，少走弯路。我的经验是增加一个验证需求的环节，以避免这种现象的出现。比如，拿到需求书后，可以向项目组的其他程序员讲述自己的理解，听听别人的看法，如果别人不能认同，就有可能说明自己的理解有问题。又比如，我们可以把自己的理解和用户当面进行交流，得到确认后，再去编程。

（2）正确对待需求书

很多人对用户需求书比较迷信，认为那就是需求的最终版本，不可更改，一切要按需求书来编程。这种一是一、二是二的看法，想法太理想、太天真，可以说这样的程序员还处于“学生气”状态。需求书固然是用户需求的书面表示，是开发项目程序编写的依据，但是，要知道需求书也是人写的，只要是人写的，这个需求书就会反映出作者对业务的理解能力和业务水平以及表达能力。而这些往往和最终理想的用户需求是有偏差的。另外，需求书的编写也是需要时间的，有的需求书写得匆忙，论证不够，不妥之处必定存在。在现实中，用户只是提出一个大概的业务功能要求，具体用户需求书都是由程序员来完成，最后由用户确认。这说明需求书并不一个绝对的东西。在现实中，最终的系统功能与最初的需求书中的功能往往相差很大，最终的系统功能要大大多于需求书的功能。我的意思是，程序员不可迷信需求书，要根据项目的设计目标来分析需求书中的逻辑性。

（3）不要急于编程

程序员一般性格比较急，用户一提出修改，往往马上会按用户的要求去改程序。虽然一般情况下用户要求的时间比较紧，但程序员还是不要急于编程为好。因为用户提出的修改意见不一定全是想得比较周全的。在现实中，程序员刚刚按用户需求改好了，这个时候用户又提出新的要求了，这种现象并不少见。因此，我的建议是，用户提出需求变更以后，要先和其沟通，看看是否真的需要变更，如果真的需要变更，则要和用户交流一下变更后可能出现的各种情况，让用户认可。此时可以建议用户再想一想，再看一看，还有什么要改动的地方。等到用户实在提不出新要求了，再考虑编程。还有一种情况，那就是将用户变更的需求放到一边，等到若干个变更需求都确定的时候，再去编程。这样做的好处在于避免单个需求编程和急促编程带来的重新编程的后果，可以从整体上考虑编程的效果。

（4）用技术应付需求变化

很多情况下，程序员抱怨用户需求变化太快，自己跟不上他们的变化，而且前面的编程工作白费，令人沮丧。但是，程序员是否考虑过用户需求变更一定要更改程序吗？如果程序员回答是，那说明程序员的技术水平还有待提高。如果项目一开始就对用户需求分析比较充分，软件架构设计比较合理，功能模块设计比较清晰，那么到了用户提出需求变更时，这种变更也是小范围的和有限度的，大部分可以通过参数化的方式来解决，也可以通过外挂功能的方式来解决。比如，原来用户提出打印两张报表，现在又提出打印3张报表。如果程序员对打印报表有足够经验的话，在开始设计的时候，就已经考虑到报表可能不止两张，已经考虑报表产生可以通过外挂方式来实现主程序的不更改。如果程序员只是一张一张地处理报表，整个程序没有架构，那么增加到的3张表将会使程序很长很乱，如果用户增加到30张、300张那又如何处理呢？所以，我们说，不好的技术可以解决一个需求，而好的技术可以解决一类需求。随着程序员开发经验的增加和技术水平的提高，用户需求变更将不再是一个令人头痛的问题。

（5）向用户提出建议

很多程序员也看到，很多时候用户提出的需求都是粗线条的，考虑并不周全，这就需要程序员根据自己的经验提出合理的建议，让需求完善起来，否则，最终用户还会回到这个建议上。例如，用户要求查询数据，并把查询结果给显示出来。其实有关数据查询和结果处理有很多种方案可供选择。但是，用户当时只会考到到查询和显示这些主要的功能。如果程序员能够和用户进行沟通，建议查询时要有权限控制，显示时可以增加导出为 Excel 文件等，我想用户是会很欢迎这些建议的。如果程序员不这样做，到后来，用户肯定会提出增加查询权限、增加 Excel 文件导出功能的。到这个时候再修改程序，肯定不如开始设计的时候就有这些功能省事。为用户着想，也是为自己着想。很多时候，用户是不完美的，程序员也是不完美的，但是，若两者结合，就有可能完美起来。

很多情况下，你不喜欢的东西就可能包含着对你有用的东西。用户需求不断变化，程序不断修改，程序员会感到厌烦和无奈。但是，当程序员静下心来，真正去分析和解决这些问题的时候，程序员可能会发现用户需求背后无限的学习空间；就可能发现用户需求背后的软件价值和市场价值所在；就可能发现自己技术的薄弱之处和改进之处；就可能感觉到技术对解决需求的巨大魅力；就可能感觉到自己的能力和价值所在。**程序员通过编写程序走向技术成熟，程序员通过用户需求感受价值获取的市场**。当我们不可能要求别人改变的时候，我们只能改变自己以适应别人。

2.4 为什么程序员不愿写文档

一提到文档，肯定会有人向你大谈外国公司、大公司、正规公司是怎么怎么重视文档的，什么 2/3 时间用于写文档，1/3 时间才用来编程序；写文档要按照什么 ISO、什么 CMM、什么标准；不按这些标准写出来的就不是文档，就不是好文档；不重视文档就是不正规等。当问及他自己写过什么文档，编写过什么文档标准的时候，这种人就哑口无言了。

不管怎么说，文档绝对是程序员最大的软肋。一些称为高手的程序员

往往可能是文档方面的低能儿。不管这个程序员是在大公司、小公司，不管程序员是写文档的、还是不写文档的，大部分程序员在内心深处都是不愿意写文档的。

程序员一般不愿意写文档，但是程序员却喜欢看别人的文档。即使写文档，程序员一般不会把所有功能都写入文档，却抱怨别人的文档中有的功能没有说明。即使愿意写某段文档，程序员一般不会把文档写得很详细，却抱怨别人写的文档不够详细。文档绝对是摆在程序员面前的一个矛盾，如果让程序员选择是喜欢写文档，还是喜欢看文档，我估计大多数程序员都会选择后者。

那么程序员为什么不愿意写文档呢？其中的原因很多，我自己归纳了几点：

1. 怕烦

程序员从入门之日起，就在心里埋藏了一颗编程的种子，认为程序员就是编程序的，就是和计算机打交道的，程序就是程序员的全部。无论是在编程之前写文档，还是在编程之后写文档。他们都认为写文档很烦人。

1）文档种类太多。越正规越多。一想到要写那么多的文档，程序员的头都要大了。

2）文档写作要求不低。有格式要求、内容要求，还需要画各种流程图、示意图、关系图、界面图和填写各种表格说明，以及要收集各种资料。虽然没有技术含量，但是要花的时间比编程序多，而且也不一定能写好。

3）在正规的开发公司中一般一个变更就要编写一系列的变更文档。当不断变更时，只有最后的变更文档是最重要的。当我们不知道当前是不是最后一个文档时，我们就不愿意写可能被后面的文档替代的文档。

4）当写了文档之后，就会不断有使用文档的人来询问细节，这些询问往往会让人崩溃。这是因为很难写出让每个人都提不出问题的文档。

2. 没空

在很多情况下，程序员都处于一种“时间紧、任务重”状态。在急于得出编程结果的驱使下，程序员一心扑在编程上，恨不得一分钟一个变化，哪有心思和时间先把文档写好再去编程呢？即使他们拥有先把程序编出来，然后再补写文档的想法，但是一旦他们完成一段程序之后，就会立即扑向第二段程序。如此下来，编写文档只能放在项目开发的后期了。你要是真的写文档，那写文档的时间早已将编程时间给挤占了，你的编程工作就完不成了。

3. 没有用

文档的重要性其实对不同对象是不一样的。如果你说文档没有用，立即会有人用唾沫把你淹死，他们立即会搬出哪个哪个说文档是重要的，文档是不可缺的。唯独不敢说“我自己认为文档是重要的”。这也反过来说明文档的有用性程度对不同人是不一样的。对于程序员来说，只要能把程序编出来就行了。很多程序员不写文档照样编出程序来，在他们的观念中文档不写也罢。如果按文档编程序，那就要确保文档的正确性、不可更改性。而实际上，文档不如编程快，编程不如变化快，不断变化的需求和代码让文档如同一张过时的废纸一样。

在现实中，有的文档变成了项目开发后的总结，对开发本身并不起作用，只是保留一个存在的形式，以应付各种各样规范的需要。在这种情况下，文档没有任何实用价值，所以即使程序员写了，也没有什么作用。尤其是项目投产后，几经升级，最初的文档早已和现实的情况对不上号了，文档更新和系统的一致性更是难于控制。

4. 没好处

既然认为编程序才是正道，那么程序员写文档就是一种额外和辅助的工作，做了就做了，对于程序员来说没有任何好处。

当然，文档的好处更多地体现在软件公司、单位、软件用户、后续程

序员身上，它是一种“前人栽树，后人乘凉”的好事。所以程序员不愿意写文档。

文档有多么重要呢？有人把它上升到“程序员头脑的拷贝”的高度，有的人甚至说，程序员走了后，只要有了文档，软件公司可以再招新人，公司照样运转起来。如此说来，天性聪明的程序员会把文档写得清清楚楚吗？

5. 不会写

从写作本身来看，写好一篇文档不是一件容易的事。文档有文档的格式和写作要求。现在的人只喜欢动嘴说事，没有多少人在平时会动笔写字，更不用说写文章了。所以，写文档从理论上来说也是需要学习和训练的，需要在平时进行写作积累。当要求一些程序员写文档的时候，他往往会回答你：“怎么写呀？我不会。”你不会写可以不怪你，你不去学，反而理直气壮地说不会，则是你的不对了。难道那些会写文档的人都是“呆子”吗？难道他们不懂干活辛苦的道理吗？

当一个男子向一个自己喜欢的女子示爱，需要用情书来表白的时候，难道他不会写情书吗？即使他不会写，他也一定会克服千难万险把情书写出来的，这是他的主观使然。如果程序员主观上想写文档，怎么会怕烦？怎么会怕没空？怎么会怕文档没有用？怎么会怕不会写？这些怕其实都是借口。真正优秀的程序员应该不单是编程的高手，同时也应该是写文档的高手。

程序员真正不喜欢写文档的原因是：文档是给别人看的，不是给自己看的。如果要使程序员喜欢写文档，那就要提高程序员的意识和境界，或者要给写文档的程序员以奖励。而靠制度、管理让程序员去写文档只能是一种职业上的弊端。当有一天写文档也成为一个专业化的岗位，程序员和文档人员分开时，程序员就会一心一意看文档写代码，文档员就会以此为职业，一心一意写文档拿工资。我们将不再为此进行讨论。这种分工一定是未来的趋势。一些大的公司或管理规范的公司都已经有这样的分工了。只是很多中小软件公司或企事业单位里的 IT 部门还没有专业化到如此程

度。程序员不喜欢写文档也许说明他们不愿意承担太多的角色吧。

不管怎么说，作为一个程序员（尤其想成为优秀程序员）一定要学会写文档，一定要学会欣赏文档。无论你是否喜欢，你都应该可以在文档的各种问题面前可进可退。可以这样说，那些既不会写文档，又不会欣赏文档的程序员是没有资格说“我就不喜欢写文档”这种话的。

2.5 为什么编程者总是高估自己低估别人

不知道大家感觉到了没有，一般程序员都很自信。这种自信成了这个职业的一个亮丽的特点。放眼望去，哪个程序员不自信满满？哪个程序员不胸怀比尔·盖茨之志？哪个程序员会对别的程序员发自内心的佩服？所以程序员几乎都是单兵作战，即使在同一个软件公司，同一个开发小组，他们之间也缺乏交流，很难建立那种理想的团队关系。

我认为程序员自信的建立来自于以下几个方面：

1. 社会评价

整个社会对 IT 有着很高的评价，凡是 IT 从业人员都沾了光。尽管这些年对 IT 的评价逐年降低，但是普通人对 IT 还是抱有神秘感的。这种社会目光投射到程序员身上，让我们程序员经常有得意的感觉。

2. 单位同事的评价

一般的企业或机关部门（除了软件公司），IT 人员只占到 10% 以下。人们天生会对自己不懂的东西产生崇拜心理（不管这个东西是不是值得崇拜），大多数人对 IT 行业只是知道些皮毛，而对 IT 程序员这样的专业人士，不流露出羡慕的目光是不可能的。在这种目光下，程序员能不自信吗？

3. 同行的评价

由于程序是由个人编写的，程序员之间必然存在竞争。尽管他们会合作共同编写一个大的项目，但是，这种合作是通过接口进行的，每个人还

是个体编程者，只要会编程的人都是他的替代人选。但是这种竞争不是显式竞争，而是用一种不言自明水平高低的方式来保持这种平衡。虽然同行认为自己编程水平不行，而且自己的水平确实很差，但是若别人不说，那么自己的自信又重新建立起了。

4. 编程的职业特点和程序员的业务水平

编程的职业特点之一是劳动工具昂贵，20 世纪 80 年代一个普通的 PC 机要 4 万～5 万元，即使是现在虽然 PC 机和笔记本便宜了许多，但是，PC 服务器和小型机甚至大型机都是非常昂贵的。这种劳动工具的昂贵也为程序员的自信增添了几分砝码。那些劳动工具低廉的职业是不可能使从业者有自信的。

编程职业的另一特点是程序员的个人劳动、个人聪明和能干能直接反映到编程结果中，而且不受他人的影响。所以，当有成果的时候，程序员无须把成果分享给其他同行。这种独占成果的特点是程序员自信的一个很重要的原因。至于程序编得好坏，由于程序最终都能够通过测试运行，而且即使出现程序上的问题，最终都能消灭，若不消灭则程序运行就不正常，但是没有人去评价编程的水平和质量。这种无人评判、自我欣赏的结果能使程序员不自信吗？

这 4 点是外部环境对程序员的职业评价，职业评价高造成了程序员自信心也高了起来。而最令程序员自信的是其业务水平。程序员有两大类，一类是会说的，一类是会做的。会说的往往精于对计算机发展的新技术、新产品进行跟踪，说起来头头是道，仿佛无所不知、无所不晓，给人以紧跟潮流、水平高超之感。尽管他们所说的对新技术和新产品推广有益，但是这些程序员从本质上来说就是国外新技术和新产品简介的朗读者，他们以别人高水平的光环照亮自己，好像自己也很光亮一样。这些程序员往往说得多，做得少，水平一般。会做的往往给自己的自信心以有力的支撑，他们靠自己的程序证明自己。大凡做过项目的，无论大小，随着时间延长，这种自信心越发得强，而且这种自信心根植于内心，不容改变。往往工作年限长的会看不起年限短的，项目做得多的会看不起做的少的。虽然

很少有人把这种感觉表现出来，但是内心还是客观存在的。

两者的自信心的膨胀都使程序员高估了自己，一个忘记了“光说不练假把式”的谚语，一个忘记了“山外有山”这句至理名言。

程序员要放下身段，脚踏实地，不断地提高编程水平，用自己的编程成果——产品说话，以证明自己的技术价值和社会价值。

2.6 我？还是我们

放眼当今社会，“我”字当头，相当普遍、相当流行。我想、我能、我做、我行，凡事以我为先的状况处处可见。人们有了自我，有了自我实现，突显了与众不同，提升了人的人生价值，这是一件好事。同样，程序员也是“我”字当头的职业，很多程序员以“我”为中心，自信满满。对于普通程序员来说，这是一件好事；但是对于追求优秀的程序员来说，就可能是一种约束。

程序员是一个自信心很强的职业，这主要是因为程序员编写程序是一种个人劳动，而且这种劳动往往能很快产生劳动成果。一段程序编写完成后，立即运行这个程序就得到了预期的结果。这样自己操纵了计算机，成就感就产生了。这个过程一般只要 1 ~ 2 个小时，有的只要几分钟就完成了。如此短暂、频繁的成就感刺激着程序员，想不自信也不行呀。而那些作家写一部小说往往需要几个月甚至几年，这就意味着成就感刺激的频率是很低的，自信远远不如程序员来得迅速和强烈。当然，自信程度还与外界的认可有关，外部认可，自信就可能强一些；外界不认可，自信就可能低一些。

这种成就感造就了很多程序员很自信，当自己比别的程序员编的程序质量更高，用的时间更少的时候；当别人编不出来，自己能够编出来的时候；当自己知道，别人不知道的时候，这种成就感就会油然而生。

程序员这种自信自然而然地会产生很强烈的自我意识，“我想怎么样就怎么样，我认为怎么样就怎么样，我是高手”。加上程序员很少关注外

界对自己评价，外界也很少对自己评价，久而久之，不加注意就会形成以“我”为主的意识和工作习惯。

说句实在话，这种自我对满足于为编程而编程的程序员是没有什么问题的，而且这种自我从某种程度上说是程序员各种痛苦的安慰剂，是程序员的心灵鸡汤。我们看到这种自信就会产生一种对程序员的尊敬。

但是，对于一个致力于成为优秀程序员的程序员来说，这种以“我”为中心的思维习惯就会影响他的发展。程序员在成长阶段要特别意识到个人在程序员这个职业中的局限性。要有意识地摆脱个人意识，从“我”过渡到“我们”。当然这个过程很痛苦，需要有战胜自我的坚强意志。

第一，这个过程首先要求我们把自己的兴奋程度降低，而这是一般人很难做到的。这就意味在自己产生成就感的那一瞬间就要告诫自己，高兴一会儿就可以了，还有更多的事情等待自己去做。这样做可以杜绝自信虚高的情况。

第二，编写程序时，要时刻想到程序是给别人看的，不要认为只要自己能看懂就行了。

第三，在编程序前最好将自己的编程思路与其他程序员进行交流和沟通。沟通的目的不仅仅是要获取别人好的建议和思路，更重要的是养成一种做事的方法。即使你已经确定自己的是最好的了，这个思路是不可改变的，但是也要和其他人进行交流。

第四，当别的程序员提出他们自己的想法的时候，应主动积极地对他们的想法进行分析，并提出看法。千万不要认为“别人的想法跟我没有关系，我没有时间去考虑别人的问题。”要把别人的想法看成自己的想法，以此来扩大自己的技术和业务范围，从而获取更多的有用信息。“三人行，必有我师。”

第五，无论自己在项目中处于什么角色，都要关心项目的整体进展。有时候，关心项目的整体进展要比关心自己的进展更重要。团队意识的强

弱是程序员素质高低的一个重要标志。

第六，要强迫自己写好文档，通过写文档可以把自己的劳动成果记录下来，更重要的是这些文档会给后来者或使用者以更多的帮助和参考。

第七，不要轻易挑战别人，不要轻易地 PK 这，PK 那。即使 PK 获胜，也说明不了什么，那只是个人的胜利，而不是“我们”的胜利。在程序员这个职业中，无论一个人怎么成功都是微不足道的。因为程序员的成功最终要反映到软件上，反映到软件产品上，反映到软件产品的市场价值上，反映到软件产品的市场占有率上，反映到软件产品对企业经营的影响上。要产生这些反映必须通过“我们”才能实现。

第八，一定要把用户当做“我们”中的一员，耐心倾听用户的意见和建议。不要把用户看做自己的对立面，一遇到用户提出新的功能、修改需求就心怀不满。要从用户的意见中看到自己设计上和编程上的不足，从而改进程序质量。同时也要把一些技术上的限制用通俗的语言向用户解释，以获得用户的理解。

上面的“我们”只是一个团队、一个部门的“小我们”，当我们把“我们”扩大到我们的用户、同行、外部的各种企业、外部的各种人才等的时候，我们视野才会扩大，这种扩大对于程序员的素质和程序员的技术水平提高都是非常非常有益的。

目前，在中国程序员已经度过了个人英雄主义的阶段，那个阶段的程序规模比较小，一个人就能完成的程序比较多。而现在一方面程序本身的规模越来越大，需要越来越多的程序员分工协作；另一方面，程序之间的关联性越来越多，也从客观上需要程序员之间加强协作。程序员要想获得更大的发展空间以及更多的收入，那就更需要和外界发生联系，去寻找和发现更具有市场价值的软件，以这个软件来寻找更多的“我们”。

总之，程序员在成长阶段，一定要意识到“我”是束缚自己成长的最大紧箍咒。一定要舍小我而求大我，养成以“我们”的角度思考的良好习

惯。这个习惯可以很好地支撑着程序员走向成熟阶段和优秀阶段。

2.7 为什么程序员的社会地位在下降

针对这个话题我已经想了很长时间，而且每次想到这个问题心情就特别沉重。今天我从广播中听到一则新闻，北京大学生就业报告出炉，失业或离职者最多的5大专业中，计算机科学与技术、信息管理与信息系统两个热门专业名列其中。这也印证了程序员的社会地位在下降这一说法。

大约在5~6年前，我就感觉到软件人员的地位和收入过了鼎盛期，开始下降了。我首先感觉到的是，相同规模的项目总价在下降，随后听说软件公司不赚钱了，再后来听说软件人员的收入大幅度下降了，去年听说新招入的大学本科生的月薪只有1500元。我今天询问了一般的装潢工人，他们的日收入一般在100元以上，好一点的更是200~300元。是什么原因造成程序员的地位和收入逐步走低呢？以下情况引起了我的深思。

1）**程序员人数不断在增加。**由于每年大学都要热招计算机专业的学生，以致程序员的存量在不断增加。物以稀为贵，人多了，供需出现了逆转，其价格必然下降。

2）**编程技术进步使得编程门槛降低。**由于程序设计语言的快速发展，许多复杂的功能都变成控件和库，原来很复杂的界面设计，只要拖拉拽就能实现，原来不懂的要自己去钻研，现在只要会网络搜索，下载调用就行了。所以，现在进行编程，只要会拖拉拽，只要会Ctrl+C、Ctrl+V，只要会上网搜索，基本上就OK了。这样对编程者的技术要求就会降低，程序员价格必然下降。

3）**企业竞争十分激烈。**企业信息化成了竞争中的利器。一个企业中的每个部门和科室都会提出信息化的具体需求，而且需求必须在极短的时间内实现。这样软件人员根本没有时间去提高自己的编程技能，能把功能实现就是上上策了，根本不管代码重复，质量不高的问题。

4）**一般软件企业就专注于一个行业。**有的甚至只在一个企业、一个系统中进行开发，这样程序员的业务知识和程序范围就很受局限，程序很难有新意，大部分就是复制了事，在这种情况下，技能有局限性的程序员不可能有太高的价格。

5）**企业信息化还在发展初期。**软件数量多规模小，而且竞争残酷，价格低，反映在程序员身上的价值自然就很低。例如，一个项目总价为10万元，核算为5个人月。实际上至少要10个人月（竞争中必须降低核算人月数，否则无法获得该项目）。程序员在这种情况下实际价格就会比核算价格低一半。

6）**开发方式专业化。**程序员脱离了系统设计和项目管理工作内容，专心编程，真正成为了编程工具。一些创造性的工作变成周而复始的机械工作，而且开发只注重结果不注重过程和质量，导致技术高、质量好的程序员得不到额外的鼓励，客观上也未能鼓励优秀的程序员出现。这同样会导致程序员价格下跌。

程序员的社会地位下降，这是整个社会需要反思的问题。很多事我们不可为，但很多事我们必须面对。我们真的要认真思考这个现象，提高程序员工作的含金量，提高程序员的技能水平，还程序员内在价值的本来面目。

2.8 加薪的问题

只要工作就会有薪水。这个问题贯穿一个人的职业生涯，而且是一个基本问题、首要问题。但是薪水高低的决定权并不取决于企业的员工，而是取决于企业的管理者。这就形成了员工与管理者之间的博弈，最激烈的结果是员工跳槽而去。

新手刚参加工作，自然在薪水问题上没什么话语权，更由于求职心切，薪水要求自然就会低了一些。随着时间推移，程序员逐渐进入成长阶段，程序员经过了大量编程实践的历练，自信心有了很大的提升。这时注意力可能并不完全集中于编程，开始分散到工作环境的好坏、周围员工和

领导的评价、自己的收入与公司内其他人或公司外其他人的比较等 。这种注意力分散说明程序员步入了一个新的阶段，这些注意力都会有利于程序员成熟起来。

对程序员来说，这些注意力中最纠结的就是加薪问题。其主要原因是：第一，薪水是程序员工作应得的报酬，是必须考虑的问题。第二，由于工作了一段时间后，自己的工作能力有了提高，自我价值增大，需要对这种变化获取价值上的补偿，否则，自己工作再努力、水平再提高，薪水还是没有增加，人可能就会失去工作的积极性。第三，不好意思开口。很多人在内心想要加薪，但是却不表现出来。很多人从表面上表现出对金钱的远离，以示自己不是唯利是图的小人，在想要加薪时也是欲言还休。第四，不知道如何表达。当程序员下定决心要求加薪的时候，却不知道如何正确表达。最怕说了薪水还是没有增加，却给领导留下一个不好的印象，影响今后工作的开展。

作者认为加薪问题是一个相对复杂的问题，每个企业、领导、员工的情况都不一样，这都会给加薪问题带来不同的结果。很难用一个通用的表述来解决。但是，我们也总结了一些要注意的问题，希望可以为程序员提供一些参考。

1. 加薪的时机

加薪并不是任何时候都可以提出的，主要还是看时机。一般在年初、年终或年终奖发放后，项目终了，程序员进企业的整数年时间为最好。这很大程度上要看企业本身加薪时间的一般规律。提出加薪的时机不对，会让企业领导们太“重视”你的加薪要求，这样加薪的可能性不大，还不如把握时机，顺势而为。

2. 加薪的理由

加薪的理由特别重要，因此程序员要找出足够充分的理由。如果企业是按照程序员工作年限或者资历来确定是否加薪的话，一般情况下，除非自己有特别的理由，否则只能等到满足加薪条件再说。

程序员加薪最重要的理由是为企业创造了巨大的效益，尤其是直接效益更佳。这个效益（扣除了企业成本之后）要远远大于程序员的收入才行。第一，在一般情况下，由于存在竞争，如果软件的利润很低、企业的效益很差，加薪的理由就显得不怎么充分；如果企业效益好，并和自己有很大的关系，这个时候提出加薪理由就很充分了。第二，软件项目开发完成和销售成功主要是自己的原因促成的。自己是企业或团队的骨干，比其他同伴更加努力、出色。当然，是否是骨干以及是否出色，这个问题程序员不要自己给自己评价，要听听其他人的意见，这样可能更客观一些。第三，程序员在不同的项目中，而不是一个项目中表现出色，这也是加薪的充分理由。这表明程序员可以在未来为企业创造更多的价值。

3. 加薪的表达

程序员有了加薪的理由之后，如何表达加薪的要求也很重要。

第一，程序员在内心要树立加薪不仅是为了自己，也是为了企业的思想。不要犹豫不决，要坚定加薪的信念，这样就不会在这个问题上给领导以欲言又止的印象。这种犹豫会影响到你的加薪理由的充分性、必要性。

第二，要找到表达的对象。不同的企业决定员工是否加薪的领导是不一样的。有的是主管起关键作用，有的是部门经理，有的是老板本人。因此，你要去了解你的前辈们加薪时找的是什么人，找错了对象，不仅可能加不了薪，还可能带来负面影响。

第三，表达的方式，可以是书面形式，也可以是面谈，或者是两者结合。根据一般的经验，当面表达可能更好一些，因为面谈能够看到提出加薪请求后领导者的反应。

第四，表达的时候，一定要表明自己非常注重企业的发展，表明是企业发展给自己带来了进步。这样会给领导一个良好的心理暗示。然后很客观地表达自己加薪的请求，请领导考虑。表达时不要过分说明自己的成果，因为这些成果领导应该是知道的。另外更不要夸大自己的成果，把别人的功劳算在自己的身上，在这点上领导也是心知肚明的。

第五，千万不要用威胁的语气提出加薪要求。即使这个企业离开你立即就会倒闭也不要这样去做。有的老板可能会被你威胁成功一次，但是，绝不可能会被你威胁成功第二次的。因为老板可以忍气吞声地答应加薪，但随后必然会找人替代你。

第六，倾听领导者的意见和建议。一般领导者在听到加薪请求之后会表示给予考虑的，但是很多情况下会讲出企业面临各种资金困难的问题。这时候程序员要认真倾听，表示理解企业的困难。如果不是真的很困难，那就再一次表明加薪的请求，让领导者了解你加薪的决心。如果领导者当面拒绝，也不要当面争吵，最好说希望领导能考虑和了解自己的请求。

第七，加薪的请求一次足矣，不要多次找领导交涉。2 次、3 次、4 次，多少次的结果都是一样的。领导定下来的事一般是很难改变的。所以，只要一次，不要多次。多次不仅无用，反而会给领导留下不好的印象。

4. 加薪的结果

提出加薪的结果一种是成功了，另一种则是失败了。成功了皆大欢喜，自己要更加努力工作，不待细说。而失败了，千万不要耿耿于怀，要从客观上找找自己的原因，找找企业的原因。把原因找到后，也许自己会对自己有一个全新的认识。

程序员面对失败不要气馁，你只需对自己说，“我已经努力过了”，“加不加是企业的事”，“明年再争取”。这样程序员就可以从加薪的苦恼中走出来，把注意力重新放在工作和技术提高上，为下一次加薪做好充分准备。

说到底，加薪是要靠自己做出成绩，要靠自己争取的。其中做出成绩是正道，争取方式是技巧。两者有机结合就可能达到为自己加薪的目的。如果一切做得很好，但是依然不能加薪，那可能就需要考虑换个环境了。

2.9 门门通还是精通一门

我一直希望计算机只有一种程序设计语言，哪怕是高级程序语言只有一种也行，这样我们就不必为学这种或那种语言而烦恼了；或者我们学习语言不费事，有一种学一种也行，也没有学不全语言的烦恼了。但是，两者都只是一种希望，不知道未来能否实现。

程序员进入了成长期，必然和程序语言打交道，这个时候学语言一定是和具体项目、应用、客户相关。学习语言已经不是一种个人爱好的选择，而是一种工作的选择。就如同扫雪时不仅需要扫帚，而且需要铁锹一样。

有的程序员在单位只负责一个系统的维护和升级工作，这可能只要掌握一种语言就足够了。有的程序员所在的软件公司会接来各种各样的项目，而各个项目应客户要求采用不同的语言开发，有的项目需要用 C#开发，有的项目需要用 Java 开发，有的项目需要用 PHP 开发。当公司人手不够时，程序员必须学习多种语言才能满足工作的需要。因此，程序员因工作需要学习语言的数量是不一样的。

程序员不但因为工作需要要学习语言，还会为未来职业的发展而考虑学习语言。未来职业发展需要何种语言，需要掌握多少语言，这一切都是一个未知数。所以，大家潜意识里有一种多多益善的想法，恨不得有一门学一门，一个都不漏过。因此，程序员心理负担极为沉重，无论是在工作时，还是休息时，每每都在想“门门通还是精通一门”这个话题。

当然有一些程序员还没有改变新手时期对语言的认识，还是以个人的兴趣去选择语言和学习语言。就这一点而言，站在实用主义角度，我自己反对因个人兴趣而学习语言，除非个人兴趣和工作需要以及未来职业规划相一致，这种个人兴趣才是值得提倡的。反之，为学语言而学语言且该语言与现有工作以及未来职业无关，仅仅是因为这种语言自己没有学过，或语言很流行就去学，那是没有必要的。因为中国不缺那种不实用的语言，只缺实用的软件。

从我自己的成长经历来看，一个程序员一般至少要精通二门语言。这两门语言一个应该是面向过程的语言，一个是面向对象的语言。至于了解和掌握多少其他的语言，那就要根据工作需要和个人职业规划来定了。一般3～5门也就足够了。这也就是说，一个程序员一般要掌握五六种程序设计语言，其中两门必须精通，其他3～4门只要求一般掌握，可以利用它们进行编程就行了。

我想对成长期的程序员在语言选择方面、学习方面、实用方面谈一谈自己的看法，以供参考。

1. 精通一门到什么程度

虽然很多程序员也知道要精通一门语言，但是他们并不清楚精通一门语言到底有什么标志。有的人会按照程序员掌握语言的时间来确认对语言的掌握程度。例如，如果一个人使用某种语言编了3年的程序，他自己或其他人就会认为他对这门语言很精通了。有的人会根据自己对这门语言的驾驭程度来说明自己对这种语言的掌握程度。例如，有的人认为自己编起程序来很顺手，不用查手册，说写就写，很熟练，几乎什么程序都能编出来，以此认为自己对这种语言很精通了。

编程时间的长短和编程的熟练程度仅仅是精通语言表现的一个方面。更重要的是程序员要掌握这门语言的适用范围、整体架构、语法规则、功能分类等基础理论方面的知识，并能利用这方面的知识，以最科学的方法解决现实中各种项目的各个问题。

通俗地说，如果你能当这个语言的老师（反映出对语言的理论和语言架构的掌握程度），又能熟练地用这门语言解决各种问题（反映出对语言的使用能力），那你就可以说你精通这门语言了。

如果有人让你介绍一下某种语言，你不会讲、讲不全、讲不透，说明你对这种语言的理论方面和架构方面的知识掌握得远远不够。有人问你一些这种语言的一些常见问题、一些常用的技巧、一些常见的错误，而你都无法解决和解释，说明你对这种语言的使用能力还不够充分，掌握还很肤

浅。这些都说明你没有精通这门语言。

2. 最好是精通两门

我感觉精通一门语言还是不够的。程序设计语言一般分两大类，一类是面向过程的语言，一类是面向对象的语言。如果要很扎实地在程序员这个职业中耕耘，最好精通两门语言，一门是面向过程的，另一门是面向对象的。如果怕时间和精力不够，我建议一定要精通一门面向对象的语言，因为在面向对象的语言中也包含了面向过程的编程内容。

3. 其他 3 ~ 4 门要怎么才算掌握

现实中我们常常会找一本书去学习一种语言，也会用这种语言编写几段小程序，那这样究竟算不算对这种语言已经掌握了呢？怎样才算已经掌握了呢？我认为只要能用这种语言开发一个项目，开发中没有太多问题，那就可以算掌握了。当然，开发的时候，断断续续，走一步、查一步、问一步，问题如同连环绊脚石，这样则不能算掌握了。

4. 不要太急于赶潮流

一些程序员往往对技术潮流关注过度，明明自己工作上用的是 C 语言，自己做的是后台维护的工作，但是，听说现在流行 C#、Java，就老想有时间把 C#、Java 学学，不学可能就落伍了。

说实在话，如果不急于应用，最好等这门语言成熟以后，再决定学习也不迟。从各种语言的发展过程来看，有些语言的生命周期也只有短短几年时间而已。

5. 语言是互通的

其实各种语言在本质上是相同的，它们有太多的共性，虽然有些个性，但是在实际中个性功能很少用到，用到时现学也不迟。因此，我们可以在精通一门语言的基础上，通过这门语言的结构去学习另外一门语言的结构，对这门的示例采用另外一门语言来编写，这样有对照、有比较地学进步应该会很快的。找出相同点，这是学习多种语言的技巧。

6. 语言的无知与有知

现实中经常会有人问你，你知道什么什么语言吗？如果你不知道，千万不要觉得自己无知而羞愧。**因为人的价值不在于你知不知道，而在于你知道后做了些什么**。常言道“光说不练假把式”，为知道而知道，只会浪费实用的工作时间，影响自己实战能力的提高。在很多情况下，有知和无知可能会发生逆转。有意的无知要比有意的有知表现得更加聪明。

当然，我们可以关注一些语言方面发展的新闻，就如同我们每天关注国内新闻、国际新闻一样。能给自己留下印象的那就成功了，不能留下印象的那就算没看，传播新闻那可是媒体的事，我们权当休闲就行了。

特别需要提醒的是，对程序员来说，实用主义是比较好的，把现有的实用的语言掌握好，研究深入一些比什么都重要。人们可能都会有“这山望着那山高”的想法，但是只有站在这山上时，才能望到那山。

2.10 程序是给自己看的还是给别人看的

不知道各位程序员有没有用自己姓名的拼音字母作为文件目录名或程序名的经历。从我对此的观察来看，很多人都是这样做的。他们为什么这样做呢？我发现这样做除了可以醒目地标记目录和文件之外，更重要的是反映了程序员的“私有”心理，他强调了程序的私有性。我们常常会发现有一个有趣的现象：程序员很少主动把自己的程序交给其他程序员阅读（除非上级要求、工作移交），程序员也不想看别人的程序，如果看的话也许会抱怨看不懂！程序员读不到别人的程序，程序员也不想将程序给别人看→程序员就得不到别人对其程序提出的意见→程序员就不会改进程序的可阅读性→程序可读性低→其他程序员就看不懂→看不懂就不愿意看→看的程序就少→阅读水平降低。这样恶性循环导致程序员只能读懂自己的程序，阅读他人程序的水平普遍很低。

一旦程序员将程序变成了私有的，程序只给计算机阅读，我们这些外人就无法从程序本身看出程序员的水平了。我们只能根据他们开发的项

目、承担的职能等外在的表现来判断他们的水平，而这些往往不如程序来得直接。我认为存在这个现象有以下几个原因。

1. 面子

程序员是很有自信心的人群，在前面的内容中我已分析了自信心的来源。这种自信反映在编程上就会使程序员认为自己编写的程序是最好的，即使知道自己编写的水平很差，也不会告诉别人。这是程序员的面子在作怪。他们不想进行比较和竞争，各自抱着自己的程序在那里自娱自乐，而且业界也容忍了这个现象。这导致程序员理所当然地认为程序是自己编的那就是自己的，想给别人看就给别人看，想不给就不给。尤其是新手，那些自我感觉水平不行的程序员更加不愿亮出程序，怕丢面子！

2. 保密

还有一些程序员则认为程序里包含了解决问题的算法和技巧，是自己辛勤劳动的成果，是自己的知识产权。如果给人看了，别人就会无偿地拿走，自己的劳动就会白费，所以他们不愿意公开自己的程序。

如果真的是这样，那我倒是非常理解的。但是，就我的经验来看，绝大部分程序员和绝大部分程序都是很难达到知识产权这个层面的。回忆一下，自己编的程序有哪些是别人不能编的呢？有哪些是别人不能超越的呢？即使有点小的技巧也不必藏而不露吧。好的程序员是不会背着这些包袱去获取新的进步的。

说到底还是程序员的程序私有化想法在作怪。

3. 环境

有时候我和程序员谈到这个问题时，他们认为不是自己不愿意给别人看，而是别人不想看，自己想看别人却不给看。在此确实有一个环境问题，没有人要求程序员这样做。这个编程环境并不鼓励程序公开，不鼓励程序的优化，不鼓励程序的共享，反而使那些编程水平差的得到了伪装，那些编程水平高的得到了淡化。由于没有制度上、管理上的明确要求，这

个问题就会长期存在。而我认为，程序员既然是个性张扬的群体，那为什么不能从自身做起，以给别人看的要求编写程序呢？不能以学习的态度请教程序写得好的同行呢？学习是一个获取知识的过程，也是将来给予的基础。只要有利于编程水平的提高，我们程序员都可以尝试。这样用不了多久我们的环境就会好起来，我们的程序交流和技术交流就会好起来，我们程序员的心胸就会开阔起来。

就我而言，我编写过无数的程序，但我从来没有保留过自己的程序，我的程序都可以给我的同行看。因为我认为程序就是给别人看的。虽然是自己写的，但是只有给别人看，那才能发挥其真正的价值，就如同一篇文章只有给读者阅读，才能评价出文章的好坏。唯有给别人看，才能促使自己提高程序的可读性，才能获得别人的批评和建议，才能改进程序的不足，才能提高自己的编程水平。

我建议程序员要养成“程序不是给自己看的，是给别人看的”的习惯，尤其是新手，从一开始就要放下面子，请别人看看自己的程序，请别人提提意见，这样自己的水平才能提高。同时，自己也要学着看别人的程序，学会向别人提出意见和建议，学会从别人的程序中学到好的编程风格和编程技巧。通过这两方面的努力，形成程序员之间互动的氛围，程序员的水平一定会有很大的提高。

2.11 程序越长水平越高吗

很多年前，我刚拿到驾驶执照不久，我到农村的叔叔家，请他把他的小车给我开。他问我会开吗？我说我有驾照。他又问我开了多少公里，我回答找了两次陪驾大概500公里吧。他对我说，那你还不会开车，你只有在开了5000公里后，才能说你会开。我当时很不服气，心里想为什么要开到5000公里才算会开车呀！难道开了1000公里、2000公里就不算开车了吗？直到多年后，我才切身体会到他讲的是对的，开了5000公里才能遇到各种各样的行车状况，驾驶员才能懂得如何去面对和处理这些状况。

回到编程这个话题，程序员成长也是这样。一般一个程序会有几千行

语句，有的甚至上万行，一个项目所有的程序加起来有几万、十几万行语句也很平常。对于程序员新手来说，要多编程序，尽量遇到编程中的各种问题，这样才能慢慢地成熟起来。编程慢慢地有了感觉，程序也有了积累，编程有了可以重复利用的语句，编程速度会大大加快。建议新手找一些较长的程序作为范例，而且这个程序应该比较完整，具有实用价值，最好这个程序正在投入运行！这样学起来看得见摸得着，容易进入状态。我的经验告诉我，程序员在初期要大量地编写程序，不要管编的质量好不好，先把编写语句的数量搞上去，水平自然而然就上去了。不编到上万行语句，程序员是很难找到编程感觉的。

当程序员度过了最初的编程阶段后，程序员就要转变思想，慢慢地把程序的精练放在重要的位置。我一直把最短的程序看做我编程的目标，一个几百行的程序哪怕可以节省一行都是我要追求的，我绝不允许自己的程序中有一句废话。

一次我的同事告诉我，他读了软件公司的一个应用程序，大约有3000～4000行语句。碰巧的是我也编了相同功能的程序，我只用了1000多行，他也读了。他说同样的功能为什么会相差这么大呢？是程序长的水平高？还是程度短的水平高？

我的回答是：相同的功能因人不同而程序不同是很正常的。区别主要是由于对功能的理解、功能的算法、编写习惯等方面不同而造成的。因此，程序可长可短。至于水平问题，那涉及的面就多了，包括程序的书写方式和习惯、程序的可读性、程序的逻辑结构、功能算法、编程技巧、运行效率、参数化程度、程序复用程度等多个方面。但是，有一点是可以肯定的，那就是相同功能编写语句少的，水平相对较高。当然这个少至少占参照语句的10%。如果对比的两个程序一个是100行，一个是104行，两者相差仅4行，这就很难在语句多少方面判别程序员之间的水平高低了。程序的精练除了反映在程序总的语句条数上，还反映在程序中的函数的长短上，尤其是在一个函数体内，其语句也是越短越好。我曾看到不少程序中一个函数能写到几百行，甚至上千行。这类程序员往往喜欢一气呵成，把一个功能从头到尾写在一个函数中。我认为这绝对不是一个好的编程习

惯和编程风格，理想中的一个函数大约有 20～30 条语句是最好的。

例如：我们常常会拼接一个文件名，这会用到 3 行语句（以 C 语言为例）：

```
Strcopy(file_name,f_name);
Strcat(file_name,".");
Strcat(file_name,ext_name)
```

其中 file_name 是最终的文件名，f_name 是文件名，ext_name 是文件的后缀名。

如果我们编写一个字符串拼接 strjion 函数。

```
Strjion (str, str1, str2, str3)
String str,
String str1, str2, str3;
{
    Strcpy (str, str1);
    Strcat (str, str2);
    Strcat (str, str3);
}
```

这个拼接只要一条语句就写成了：

```
Strjion (file_ name, f_ name," .", ext_ name);
```

如果程序中有 100 处拼接，原来需要写 300 条语句，通过一个函数 8 条语句，就只要 100 条语句了，节省了（200－8）条语句。

这仅仅是一个例子，我们也可以用 C 语言中的一个系统函数来实现上述功能，大家可以猜一猜这个函数是什么。

不少程序员热衷于编写越来越长的程序，放任自己编写不必要的语句，从不注意程序的精练性。导致时间越长，程序员的水平越低。现实中我们可以看到很多程序员在刚开始的时候，感觉其进步很快，但是几年之后，你依然看到他每天努力地编程序，依然看到他每天都没有进步。而有的程序员每隔半年或一年，他们就会发现自己以前编写的程序编得很差。这说明这个程序员始终在努力地提高自己的编程水平，而且具有判别编程水平的能力。一个好的程序员到了一定阶段其编程数量是呈下降趋势的，

但是其程序质量和水平却是呈上升趋势的。还是使用驾驶员的那个例子，当驾驶员开满了5000公里后，驾驶员会开车了，但是会开车并不意味驾驶员就能成为赛车手，成为赛车手需要其他的专业训练，只有那样才能又快又好地完成行驶路程。

综合各方面来看，程序员尤其是程序员新手编写的程序越长越好，越多越好。但是，在具有一定编程积累后，程序员要学会编写精练的程序，程序越短越好，程序的精练程度是标志程序员水平高低的重要因素。精练的语句永远是程序员追求的目标，程序员可以写长篇小说、短篇小说，但能写出诗来才是最好的。

2.12 动手能力强与技术水平低

很多搞计算机的人员是相当聪明的。我记得20多年前，那时使用的是DOS操作系统，我们的一个用户竟然把我们在他们单位机器上调试的源程序偷偷地拷贝下来了。要知道我们在每次编译过后都会把源程序删除的。他竟然在我们转身之际将文件的删除标志给去掉，把源程序还原出来了。到最后，我们自己的源程序没有了，他倒是保留了我们的源程序的各个版本。这些事都是若干年以后他告诉我们的，可见他有多么聪明了。他解释说，他只想看看我们写的程序，研究学习一下。这是好事呀。我们从此成了好朋友，交往至今。

我经常看到动手能力很强的程序员。他们做事迅速，整天只是坐在计算机前编程，一边编程，一边修改，一边编程，一边修改，如此往复。但是我看不到他们水平高超的地方。因为他们的程序编写了几天后便会大改一番，有的甚至要推倒重来。这并不是个别现象，我终于忍不住去问他们动手为什么这么快，他们的回答是项目或需求时间要求紧，没有办法只能立即编程，遇到问题再说解决的事。

于是我开始关注程序员的动手时间和动脑时间的比例关系，关注这个比例与程序员水平的关系。

我发现许多程序员的动手时间和动脑时间之比都在9:1以上，而且比

例越大动手能力越强。动手能力强除了因程序员性格的自然属性之外，主要是因为编程能刺激大脑高速运转，并刺激程序员的编程欲望。但是我也发现了动手能力强的主要原因：

1. 熟练

程序员对编制程序内容和方法十分熟悉，动起手来十分熟练，更有Ctrl + C、Ctrl + V 大法，编程速度飞快。

2. 调试

编得快必然会出现各种错误，出错后必然要调试，调试则是一个重复性和耗时性很强的工作，必然导致程序员动手能力提高。

3. 即想即编

由于编程前考虑不全面，在编程序的过程中，往往是只要冒出一个想法，就会按照这个想法编写程序，看看这个想法对不对，不行再另想办法。这是一种环环相扣的做法，许多程序员就是这样日夜不停地编写程序。如此高强度编程想不熟练也不行呀！

4. 时间要求

外部的时间要求给程序员造成了只有早动手才能及时完成任务的想法。程序员的上级也以程序员是否动手作为判别工作是否开始的标志。这两者结合迫使程序员养成了动手能力强的现状。

我认为程序员的水平反映出的是程序员的智慧。动手反映的是小智慧，而动脑反映的是大智慧、高水平。一个程序无论大小，都要经过大脑严密思考，在心中形成无形的程序，把整个流程都想清楚，再去动手编写程序！如果程序员足够聪明，动脑只需几秒钟便可完成这个过程。程序员要意识到动手能力强与技术水平低之间的关系，改变现有的编程方式，多花点时间在程序实现的思考上面，用更多的时间激发程序员的智慧，减少调试，减少反复，并最终反映到编写又快又好的程序上。

优秀程序员的动手时间和动脑时间之比应该在7:3以下。当其比值下降到5:5、3:7以下时，程序员也就完成了向系统设计人员转变的准备。

2.13 调试高手和编程高手

业界总是有高手之说，高手成了程序员心中的偶像，成了程序员成长的目标。但是，现实中人们往往很难认同自己所了解的程序员为高手；人们反而把那些自己不了解的，甚至是道听途说的人认作高手。高手，同武林高手一样，神秘之至。

许多程序员都有这样的经历：无论自己编写多短的程序，一般是很难一次就能通过编译的；而通过编译的其功能还不一定是正确的。我在很早以前就注意到这种现象了。刚开始的时候，我还不认为这是一个问题，还试着编写程序看看自己能否一次编译成功。我甚至还为这个事和别人打过赌呢！当然这都是年轻时做过的事了。后来我渐渐明白了，编程出错是编程的一个重要组成部分！出错是正常的，不出错才是不正常的。出错怎么办？当然是改错！不知道错在什么地方怎么办？当然是调试！于是调试和编程相依相伴了。程序员不但要会编写程序，还要会调试程序。

程序出错一般有：未定义、语法和运行出错这3个方面。

1. 未定义

未定义往往是程序员忽略定义或定义后名称与定义不一致造成的。

1）要避免忽略定义这个问题，程序员首先要养成严谨工作的习惯："先定义，后使用"。

2）解决定义名字与使用名字不一致问题，程序员在对函数名、变量名、文件名、数据库名、表名、字段名定义的时候，不要信手拈来，一定要定下心来仔细斟酌，这个名字不但自己要能懂能记得，别人一看也要能懂能记得。长此以往，在使用这些名字的时候程序员就会有行云流水般的感觉，而且很少出错。

2. 语法问题

1）新手编程走一步一个跟头，走两步两个跟头，主要是因为对语言的语法不熟悉，丢三落四，关键字拼错等。这种情况不可怕，多看看语法说明，多改改程序就会对语法越来越熟悉了。

2）老程序员语法问题出现较少。但是，很多情况下程序员在使用新语言编程序的时候会采用老语言的语法，反之亦然，从而导致出错。例如，某人先使用 C 语言，后使用 VFP。当用 VFP 编程的时候，往往习惯性地在语句后面加上“;”，这导致 VFP 语法出错。当熟悉 VFP 后，再编写 C 语言的时候，往往在语句后面忘记加上“;”，导致 C 语言语法出错。类似常见问题如：IF 语句后面表达式是否要加“()”等，for 语句、while 语句、case 语句在不同语言中的语法都有可能不同。另外，不同数据库中的 SELECT、INSERT 语法都有一些不同之处，这些不同之处往往让程序员不知所措，总是停下来思考，现在用的是什么语言，语法是什么。

3. 运行问题

定义、语法这些都是显式出错，相对好解决。但是运行中出现错误就五花八门了，我试着归纳如下：

1）运行中程序突然中断并退出。

往往是内存出错！可查变量、指针是否越界，指针是否为空，异常是否没有处理等。

2）环境问题。

例如，数据库密码错误导致数据库打不开，要打开的文件不存在，参数定义错，参数表为空，参数表没有缺省值等。

3）程序输出的不是自己的预期结果。

往往是程序的逻辑和算法存在问题。

4）程序不停地运行且没有停止的迹象。

检查循环的条件表达式是否永真，从而导致死循环。

5）程序无法退出。

程序乱了，导致内存出错，覆盖了退出指令！

6）程序运行了很长时间才出现错误。

累计误差、数据原因造成的错误等。

出错是不可避免的。如何找出错误，有些编写 C 语言的人只会用在可能出错的语言前后，通过显示语句 print 来判别出错的原因。他们不懂使用 debug、sdb、dbx 等调试工具调试程序，他们会编写不会调试。我们很难想象不会调试的程序员是如何编程的，调试对于程序员如此重要，以致可推出“不会调试的程序员就不是程序员”的结论。

现在的开发语言具有功能丰富的调试命令。无论用什么设计语言调试，你只要注意掌握以下几个命令或操作，就能入门调试了：

1）运行程序。
2）设置断点、取消断点。
3）运行到断点。
4）单步跟踪（step into）。
5）单步执行（step over）。
6）显示变量、对象、源程序。
7）退出调试。

你可以按此去寻找相应的命令和操作，并熟记于心！

通过调试，程序员可以学会如何运用断点，如何最快地找到可疑语句，如何很迅速地找到问题，从而改进程序。而不断调试使得程序员不停进行调试操作，变成操作高手。调试内容可以说千变万化，技巧也很多，关键看程序员如何积累。例如，循环语句调试的技巧、同步异步调试的技巧、中断调试技巧、外设调试技巧、数据库调试技巧、边界调试技巧、例

外调试技巧、屏蔽法调试技巧、替代法调试技巧、接口调试技巧等。

调试像开车一样，不会开的时候，程序员有恐惧感；会开了以后，程序员特别想调；开多了，程序员越发老练，调试操作越来越快，对问题发生的原因和纠正方法也熟记于心中，积累也会越来越多。

好的程序员不但调试自己的程序，而且会帮助其他程序员解决找不出错误的问题。程序员在帮助别人的时候，更能尽显调试才华，成为真正的调试高手。

调试总的目标是为了纠正程序的错误。但是，调试也可用于其他用途。例如，我经常会先把函数的调用语句写好，但是不写函数体，这样每次编译都会出现没有发现函数的提示。我不写函数体的目的就是让每次编译的时候提示我还有多少函数没有编写。让我对这个函数加深印象，让我有时间思考这个函数如何编写。当其他语句编写完成后，我再一个一个把没有编写的函数内容补上，这个时候编写就快得多了。有时候，我还喜欢看到几十条上百条的出错提示，然后，沉浸在一次改错就能消灭几十个错误的快感中，然后再改再调，直到无错。调试成了我编程娱乐的一部分。

通过调试你会发现：编程水平直接影响到调试。如果程序的模块性很强，那么调试起来就很快，几个或十几个单步执行就会到达程序的任何地方；如果程序本身逻辑性不强，变量命名不准确，一个函数内语句有数十条、数百条，那么几百个单步执行都到达不了你想要去的地方，这样的程序调试起来就会很麻烦，而且不容易定位出错的地方。所以调试高手往往会改进自己的编程，使编程水平日益提高，逐渐成为编程高手。

2.14 如何快速确定自身水平

有关确定程序员水平的问题是一个很敏感的话题。我们常常会听人说，某某编程水平很高，某某编程水平很低。似乎水平是靠听出来的。有时候，朋友单位招人也会请我看看应聘者的水平如何。说真的，看一个人的编程水平不是一件容易的事情。程序员有两类，一类是能吹的，一类是比较沉默的。能吹的人往往海天湖地，容易给人看出破绽；而不声不响的

人，你却很难判断其水平的高低。但是，程序员水平高低还是有一定规律的，水平高的人，必定是开发过许多大的项目，并且编程时间一定要在两年以上的。这个规律反过来却不正确，很多时候你会发现那些参与过大项目、编程时间超过两年的人水平一般。

程序员的水平通过“听其说，观其做”这两个方面就能大致确定了。听比较简单，只要听他讲从事编程时间有多长，参加过多少大的项目，就能基本判定程序员的水平了。编程时间在两年以下的程序员只能说是入门级的，没有做过大项目，也一定是水平一般，那些以兴趣编程者，水平更是一般。另外，听的过程中也可以判断出这个程序员头脑是否清晰，说话是否有条理，逻辑是否严密。这些也是衡量程序员水平的重要方面。

看要相对复杂一些。我要看一个人的水平如何，往往会请他把自己最得意的程序拿出来。因为自己满意的程序或自己认为很重要的程序，往往反映了他的编程水平。

我看别人程序的速度比较快，大约为 1 秒 2 行。如果程序有 300 行的话也就需要 3 分钟左右。我看程序主要看以下几个方面：

（1）程序名命名

如果是主程序，则要看命名是否能反映项目主要特质？程序名是否为中文？程序名是否为英文？程序名是否为英文缩写？程序名是否为拼音？程序名是否无意义？程序名是否包含程序员姓名？程序名是否包含版本信息？如果是非主程序，则要看程序名是否与项目、模块有关联？

（2）函数命名

函数名是否是英文命名？函数名是否是中文命名？函数名是否是拼音命名？函数名是否是英文拼音混合命名？如果是英文命名，英文是否准确？缩写是否正确？读者能否看懂。函数名能否表示函数所要提供的功能？所有函数命名的风格是否一致？

（3）变量命名

变量名是否是英文命名？变量名是否是中文命名？变量名是否是拼音命名？变量名是否是英文拼音混合命名？如果是英文命名，英文是否准确？缩写是否正确？读者能否看懂？变量名能否表示变量要代表的内容？通过查看变量名能否确定变量的数据类型？所有变量命名风格是否一致？

（4）函数中的参数

函数中是否有参数？参数的次序是否具有逻辑性？参数命名是否如同变量命名？所有函数中的参数命名是否具有相同的风格？

（5）程序中的全局变量

程序中有无全局变量？全局变量的个数有多少？全局变量是否以参数方式代入调用？

（6）函数中的常量、变量值

在包含调用功能的函数中是否有常量和变量值？被调用函数中是否有常量和变量值？一个函数中是否有超过 10 行的变量赋值语句？

（7）函数的行数

一个函数的程序行数大于 20 ~ 30 行吗？

（8）函数的格式

函数有统一的编写风格吗？函数头的格式一致吗？函数体的风格一致吗？函数尾的风格一致吗？语句开头对齐吗？语句的缩进空格一致吗？语句左右括号上下对齐吗？

（9）注释

程序中是否有注释？程序开头是否有注释？注释是否用英文？注释是否用中文？是否每个函数开头都有注释？是否程序重要之处都有注释？

（10）可读性

读者必须经过程序员亲自解释才能看懂程序的主要功能吗？读者在看不懂情况下需要程序员进行解释吗？读者是否不需要程序员亲自解释就能看懂程序的主要功能呢？

通过这些方面的考察，在心中给这个程序员打分，基本上可以断定这个程序员的水平是高、中，还是低。至于这个程序做什么，有什么特别的技巧在这里就不太重要了。当然还要结合前面听的结果进行综合的评判。

2.15 程序员应该掌握的实用编程技能

偶尔上一上 IT 网站，发现求新、求奇、求专、求偏的东西甚多！可谓大千世界无所不有！但是面对中国软件的现状，一方面我们发现有很多很多的实用软件正在编制过程之中，很多人都在加班加点，忙得不亦乐乎；还有很多很多的实用软件没有人开发，等待有人开发。另一方面，我们也发现存在大量闲置的程序员，不知道自己要开发什么！不知道要掌握什么编程技能，在网上学这学那，花了大量的时间，除了学了还是学了，就是没有做，美好时光就这样白白流失，令人心痛。很多程序员并不知道这些实用软件目前最缺乏的不是新的、奇的、专的、偏的东西，而是实用技术，是实用技术运用的质量和技巧。

由于编程是“条条大路通罗马”，程序员有很大的自主性，程序员可以采用任意方法实现功能，而这也导致了编程水平的良莠不齐。我认为程序员应该掌握一些实用编程技能，掌握这些技能就要掌握这些技能的本质，吃透这些本质，这样才能归纳其应用范围和注意点：而那些只会写程序却不知道为什么这样写的程序员则可能不会用最好的语句、最简的语句、最恰当的语句来编写程序，更不用说采用更高的技巧去组合这些语句，以达到高质量的程序。

在长期的编程实践中，我认为以下的编程技能是基础，是必须掌握的，其他更高层次的编程技能，则要用到才能学到。

1. 赋值

赋值是编程中不可缺少的基本语句，而且是最基本的语句。掌握这个

语句，看起来很简单，但是越是简单的东西越要关注。最简单的赋值是指将一个值赋给一个变量。例如 x = 1。

赋值的本质是事物之间的转移。而且赋值本身就是一个中间过程，反映到代码级意义就是将内存中的值或内存指针赋给一个内存指针。

赋值有两个方面含义：一个是，取什么值、变量或对象；一个是，赋值到什么变量或对象。程序员在编程的时候，往往在两个方面都有困惑：我取什么值、取什么变量、取什么对象，怎么找到这些值、这些变量、这些对象，我取到后，我赋给谁。这都是我们常常遇到的问题。

在赋值的时候，我们要注意：

1）等式两边的数据类型要相等。

2）赋值语句相比于其他语句是处理速度比较快的。这点在注重效率方面要注意。

赋值语句比函数调用要快，比循环语句要快。

例如：在一个巨大循环中编写了一个循环语句：

```
for(i=0;i<3;i++)
A[i]=i+1;
```

还不如将其变成赋值语句：

```
A[0]=1;
A[1]=2;
A[2]=3;
```

3）当有多条赋值语句的时候，赋值处理可能会有先后次序。

4）赋值语句多了（如 20 条以上时，阅读起来非常死板，没有技术含量），可能要考虑到循环赋值。

2. 条件处理

条件处理是仅次于赋值处理的编程内容，程序的变化基本上是由于条

件处理引起的。不同的条件成立将进行不同的处理。所以条件处理的本质是事物的变化带来相应的变化。

在编程实践中，我们往往困惑于：什么样的条件？做什么事？而且还要考虑什么时候开始考虑条件。

赋值处理是一个顺序处理。而条件则增加赋值处理的可能性，当条件满足时，A 赋值就可能执行，当条件不满足时，B 赋值就可能执行。

在条件处理的时候，我们要注意：

1）我们如何选择条件？即我们的条件表达式。

其实这个问题很复杂。

一般来说，我们会把最主要的条件作为第一个条件。但是，我们也会把满足条件的最大结果集的条件作为第一个条件。这样最后的条件往往是不能满足的，或是满足后也不需要太多处理的。

在条件表达式的设计中，我们可以用单个变量来表示条件，也可以多个变量运算来表示条件。在单个变量中，可以用数值型、字符型、逻辑型来表示，而这也是很有讲究的。

例如。`flag = =1;flag = ='1';flag = =True` 都可以让程序条件转移，但是如何选择则要考虑很多因素。

2）我们不要遗漏例外情况。

例如，我们考虑 `i =1` 和 `i =2` 的时候，就是没有考虑到 `i <1` 的时候和 `i >2` 的时候。

若遗漏条件，这往往说明我们的程序员缺少全局观念和缺少例外观念。很多程序质量不高的原因中就涉及这个问题。

3）条件间不能有交叉。

例如：

```
If(i>1 &&i<=5)
x=1;
If(i>4&&i<10)
x=2;
```

当 i=5 的时候，x 先等于了 1，后等于了 2。这个必须避免。很多程序出错都与此类问题有关。

4）要特别注意条件处理的覆盖问题。

例如：

```
if(flag==1)
X=1;
If(flag==2)
X=2;
X=5;
```

无论什么条件，x 始终等于 5。

5）要知道 if 和 case 的各自适用情况。懂得什么时候用 if，什么时候用 case。

3. 循环

循环是重复操作的简单表达，只要有重复操作，就可以采用循环语句。循环的本质是重复。

在循环处理的时候，我们要注意：

1）循环处理是影响效率的重要方面。

当程序出现效率问题的时候，要首先在循环语句中进行查找。

2）循环处理的前提条件。

一般来说，重复执行三次以上可以用循环语句。低于三次的，最好不要用循环语句。例如：

```
For(i=0;i<3;i++)
B[i]=i;
```

不如写成：

```
B[0]=0;
B[1]=1;
B[2]=2;
```

当然从可读性和可扩展性方面考虑，也可以用循环语句。

3）不同的循环条件采用不同循环语句。

程序员要懂得什么情况下用 for，什么情况下用 do while，什么时候用 foreach。尽管用以上语句都能达到相同目的，但是程序员还是要知道其应用范围，使得应用最恰当。

4）充分利用循环中的中断循环、继续循环、函数返回、程序退出等语句，使得循环更加丰富多彩。

4. 字符串操作

字符串是信息的重要表现形式。字符串操作是编程中最常用的操作之一。字符串操作的本质是信息的加工。由于许多信息没有标准，程序员对其操作可以符合自己的标准要求。

例如：有的字符串包含了多种信息，那就必须对字符串进行拆分；有的字符串缺失信息，那就要对字符串进行合并。

对字符串操作主要注意以下几个方面：

1）空串处理。

原始的字符串由于操作原因和系统原因，字符串的头尾会出现若干个空格，那么在字符串处理之前，必须将空格除去。

2）乱码处理。

有些字符串中存在各种乱码，导致字符串显示出现看不懂的字符。这些情况主要是字符串出现了控制字符的代码，汉字中出现字符不匹配。

3）分隔符处理。

分隔符往往会出现在一条记录之中或参数之中，起到分隔信息之用，通过分隔符来把信息取出来。实际当中会出现信息内容本身含有分隔符或乱码中产生分隔符，这些情况就需要改变分隔符或进行特殊处理。

4）字符与其他数据类型的转变。

在实际编程当中，为了运算对象的一致性，往往要进行字符串转向其他数据类型的操作，或将其他数据类型转为字符串的操作。一般来说，其他数据类型转为字符串比较容易，而字符串转换成其他数据类型，就要考虑转换前的字符串格式是否符合要求。

例如：将“1,000,000”转换成数值，则转换前要将“,”去掉。

5）子串处理。

子串处理在查询中经常使用。子串匹配有前、中、后三种。子串匹配往往花费的时间较多，子串越短、查询串越长则消耗的时间越长。在建立索引字段中进行查询时，只有前匹配才能利用索引，到达快速查询之目的，但是在中、后匹配中则索引无效，需要每个记录逐一匹配，时间最长。程序员要了解以上内容，因势利导，才能正确进行子串处理，以达到快速查询目的。

5. 算术运算

算术运算在编程之中是仅次于字符串操作的内容。其中加 1 操作很多，用途也很广。一般应用软件中加减乘除最为常用。算术运算本质是数值类信息的加工。算术运算一方面是实际应用的算法要求，另一方面则是编程算法的需要。

例如：应用系统中要计算长方形的面积，则会编写 S = L* D 语句。

假如要编写计算 100 个长方形面积，则需要一个指针，通过指针 +1

进行下一个长方形面积的计算。而指针加1，这个运算则是算法的需要了。

算术运算在应用中公式计算相对简单。但是，算术运算用于算法的技巧和实现就不那么简单了，其注意点是：定义一些中间变量，通过中间变量的加减，使之能变成循环操作。

6. 数组

数组是存放数据的一个集合，数组操作也是编程常常遇到的：数组的本质是事物的集合。但是要注意这个集合对象个数是有限的，是存放在内存中的，因此数组操作很快。数组的使用很大一部分是利用循环语句。数组和循环的结合使得程序的质量有很大的提高。

针对数组操作，我们应该注意：

1）数组的个数相关问题。
2）多维数组的表示方法以及存放形式。
3）数组越界问题。
4）空数组。
5）数组在循环语句中的运用。

7. 调用

调用在编程中是用得最多的语句之一，一般有系统调用、自编函数调用等。调用的本质是逻辑模块的处理。这表明调用是一个模块，是一个功能语句的集合，同时，这个功能是一个逻辑划分，一般不可能将一个有联系的语句分成两个函数来调用。

关于调用我们要注意以下几点：

1）如何进行自编函数的编写？
2）如何调用自编函数？
3）如何引用系统函数？

4）如何调用系统函数？

5）调用时要注意函数的参数类型和参数次序与定义相配。

6）调用时要注意函数的返回值：无值、有值、返回值的数据类型。

7）要特别注意自编函数中的指针参数的运用，不要指针溢出，从而导致函数出错。

8）掌握将函数变为参数的方法，使得函数更加灵活。

9）要了解松耦合函数的调用方法，尤其是掌握 WebService 的调用和动态调用。

8. 文件操作

文件操作在过去运用十分频繁，由于现在很多信息都存放在数据库中，所以文件操作的大部分工作都被数据库给取代了。但是有关数据的后台处理、参数处理、输出结果等方面还是会用到文件操作。文件操作的本质是对以文件形式存放的数据进行操作。文件既可以作为操作的数据源，也可以成为数据处理的结果。

关于文件操作我们要注意以下几点：

1）要知道文件的两个基本类型：二进制和 ASCII。两种类型打开的系统函数是不一样的，这两种处理体系也是不一样的！

2）要知道文件打开方式种类和应用范围：读、写、只读、只写等。

3）要知道文件操作基本功能：打开、关闭、读、写、偏移指针（定位）。

4）要知道流文件操作常用函数。

5）要特别注意文件操作失败的返回值：打开失败和写失败（文件只读、权限不够、空间满等）。

6）了解大文件（大于 4G）的处理方法。

7）由于 xls 文件的广泛应用，我们要特别掌握相关 xls 文件处理的读、

写等功能的调用方法。

8）掌握 FTP 的相关功能，并能引用和调用 FTP。

9）掌握如何解决大文件在传输没有结束时，就开始读此文件或重新传输产生的问题。

9. 逻辑运算

逻辑运算一般用于条件表达式之间的与或非等逻辑关系，程序中使用相对较少，但是在应用的算法中则常常出现，尤其是在查询条件中，与操作用得最为广泛。逻辑运算的本质是事物之间的逻辑关系。逻辑运算仅仅是某种处理的条件和前提。其一般不能单独存在。

关于逻辑操作我们一般要注意：

1）当出现多个条件的时候，最好将条件组合成两大条件，使得程序逻辑性更强，更加可读。

2）要了解进行与操作运算时，当只要有一个条件为假的时候，整个表达式就得到假的结果。而不会把所有条件都运算出来！

例如：表达式为 a>5&&b>6&&c>7&&d>8？如果 a=1，则表达式的结果为假，并且程序不会再去运算 b>6,c>7,d>8 了。

如果 a=6,b=7,c=8，则程序一定要去运算 d>8 的。掌握了整个道理，我们可以把最容易为假的条件放在表达式最前面，从而提高程序的效率。

同理，进行或操作运算时，当只要有一个条件为真的时候，整个表达式就为真，而不会把所有条件都运算出来。

3）学会利用逻辑运算符作为参数，动态生成查询的条件语句。

4）学会通过循环语句自动形成与条件的表达式。

例如：我们查找以姓名和性别为条件同时成立的结果。

一般我们可以写成 name = '王华' and sex = '1'。

我们可以将 name 和 sex 放在 fld_name 数组中，将王华和 1 放在 content 数组中。

通过对数组操作自动形成这个条件语句：

```
String exp = "";
For(int i =0;i <2;i + +)
{
    if(i = =0)
    exp = fld_name[i] + " = '" + content[i] + "'";
    else
    exp = exp + "&&" + fld_name[i] + " = '" + content[i] + "'";
}
```

当我们的查询字段很多的时候，我们的程序就会很简洁和通用。

10. 数据库访问

数据库是目前应用软件中使用最多的技术，没有数据库的软件几乎不可想象，大型软件更是如此。数据库的本质是事物的量化集合以及相互关系。程序员不但要掌握数据库本身的功能，如数据库建立、表建立、索引建立，数据操纵语言的 insert、update、select 使用方法等最基本要领，而且要掌握通过程序的方式访问数据库。

对访问数据库我们应该注意：

1）我们如何访问数据库，访问数据库的方式是什么，访问数据库需要什么参数。

2）如何提交数据库操作命令，如何执行存储过程。

3）如何获取数据库命令的返回值，了解返回值的意义。

4）如何获取数据库命令的结果集，以及结果集存放方式，结果集的操作方法。

5）学会用表名、字段名、字段个数、记录个数进行循环操作。

6）数据库操作出错处理。

7）数据集和数据库的更新同步。

8）数据库事务处理。

9）海量数据的处理方法（如何利用结果集与数据库之间的关系提高数据处理性能，其他处理方法不在此讨论）。

至于数据库本身的一些技能就不在此谈论了。

11. 控件

控件大都用在人机交互的界面中，当然还有很多不显示的功能控件。控件使用之广泛、频繁是程序员遇到最多的对象。通常用得最多的是标签、文本框、命令按钮、列表框、下拉框、表格等，人们通过拖、拉、拽进行操作。控件的本质是人机对话的媒介。当然还有可复用对象的使用功能。

对控件我们一般要注意以下几点：

1）要特别注意每个控件的特点和使用范围，学会准确使用控件。例如，文本框可以显示字符串，标签也可以显示字符串。但是，我们一般用标签控件显示常量而且是只读，用文本框显示变量而且可编辑。注意到这些区别之后，我们就可以正确选择控件了。如果我们用文本框去显现一个标题的话，那我们就可能不恰当地使用了控件。

2）除了要掌握拖、拉控件进行定位和控制大小之外，我们还要掌握对控件的属性进行设置，以及对控件的事件进行编程。特别要了解每个事件触发的时机。

3）要学会控件的数据绑定，尤其是和数据库的绑定和刷新机理。

4）要学会如何对控件输入值进行合法性检查，以确保输入数据的有效性。

5）要学会对控件的只读、可编辑、显示/不显示的设置。

6）要学会对系统控件的复用，尤其是下拉框控件。一般的下拉框很难满足显示汉字返回代码的功能，尤其是在显示框中输入代码和汉字可以将满足条件的结果显示出来，以供用户选择。例如，可以设计一个行政区域划分的下拉框，在框中输入 110000 或北京则返回 110000。而普通的下拉框只能在 3000 多个行政区域划分中按照行政区域划分代码次序来选择了。

7）可以尝试自己设计控件。

8）掌握对表格控件的数据绑定方法。尤其是大数据结果集的处理方法。

9）掌握加载各种控件的方法。

12. 类

类在编程中广泛运用，一些语言和程序都是类的表现形式。对程序员来说，一方面我们要学习面向对象的编程思想，另一方要学会定义类和使用类。类的本质是面向对象的抽象形式。有关类的内容很多，例如封装、继承、多态等特性，但是程序员不要被类的丰富功能所迷惑，而是要学会最基本的东西：一个是定义、编写、使用自己的类，另一个是学会引用使用系统的类和其他的类。

针对类，我们要注意：

1）定义一个类，最主要的是定义类的属性和事件。
2）对类的属性赋值。
3）事件的编程。
4）类的继承。
5）类的引用。
6）类的调用。
7）特别注意类中函数的入口参数和返回值的设计。

13. 参数化

很多人都在谈参数化，认为参数化可以使程序具有更多的灵活性、扩展性。但是很少有人知道什么是参数化，参数化的边界是什么，参数实现的方法是什么。

参数化本质上是指解决事物变化的方法。其重要的内容是：第一，如何将事物抽象为参数；第二，参数如何存放；第三，程序对参数的处理。

例如：打开一个数据库，为了适应数据库的变化，我们就要把数据库的用户名和密码抽象成参数。我们可以把用户名和密码放在 config 文件中。在程序中，当打开数据库时，我们要从 config 文件中获取用户名和密码形成数据库连接串。通过这个连接串打开数据库。

当数据库的用户名和密码发生变化时，我们只要改变 config 文件中的用户名和密码即可。

以上仅仅是参数化最简单的例子，其实参数化内容非常多，而这关键要看程序员的视野和水平了。

一般的程序员仅仅停留在函数参数的参数化，其实，函数、数据结构、功能模块、软件架构都能参数化的，而且参数存放形式也多种多样，参数处理更是因人而异。有些参数还要考虑安全性。

简单的参数化是基础技能，但是复杂的参数化则是编程中的高级技能了。

很多复杂的算法和架构大都是由以上基本技能组合而成的，程序员将基础打好后，可以站在这个基础上去架构更复杂的算法和写出更优秀的程序。

编程的基本技能内容很多，每个部分都可以谈出许多趣事和范例，每个部分都能独立成章节。但是，在这里我只能抛砖引玉，只是给程序员梳理了一条思路和一个方法。其目的就是希望程序员重视基础技能，重视实

用技能，要把大量时间用在这些技能上，把基础打好；希望程序员能改变只知道编程序，不问也不知道为何这样编程的现状。我希望程序员面对算法时，头脑首先想到的是一组基本技能，然后针对算法具体实现选择最合适的基本技能，最后再用这个技能去编程。这种方法可以提高程序员对理论的重视程度，养成以理论指导编程的思路，改变自发编程为自觉编程，这样我们程序员的编程水平才会有本质提高。

第三部分
Part 3

成　熟　篇

3.1　大项目或小项目都是程序员成熟之道

我们常听到同行说自己做过什么项目，说某某做过什么项目。一谈到项目就会眉飞色舞，兴高采烈。而不少新进单位的新大学生、一些编程新手往往不知道什么是项目，不知道项目与自己成长的关系，有的甚至声称编程好几年了，还都没有做过项目的经历。情况确实如此，只有参加过项目的程序员才是真正的程序员。那些没有做过项目的虽然自己编制了不少程序，虽然对自己的程序很得意，但是，毕竟和做过项目的程序员还是有很大的差别，这些差别主要在于：

1. 程序的价值

没有做过项目的程序员，编写程序的目的主要是学习，希望通过编程来提高自己的编程能力，编写什么以及怎么编写都由自己主观决定，自己能做什么和不能做什么都不是太清楚。至于程序能否被别人使用，程序能否卖出价钱，程序员并不太关心。做项目的程序员则不一样，他编写的程序不是用来学习的（尽管他是抱着学习的态度参加项目的），而是作为商品的一部分出售，编出的程序要投入日常运行的。他别无选择，必须完成程序功能。程序员的价值通过程序出售的价格以及程序的使用来体现。

2. 程序的时间要求

没有做过项目的程序员编写程序的时间长短由自己决定，自己高兴什

么时候编好就什么时候编好，遇到其他事情打搅，拖个十天半个月也无所谓！做项目的程序员则不一样，他必须在规定的时间内完成编程，只能提前不能延后，否则整个项目进度就会被它拖后腿，而由于项目延期不能按时交付给客户，其结果就有可能因延误被罚款，甚至取消项目。

3. 团队

没有做过项目的程序员基本上是单枪匹马地编写程序，程序功能相对简单，一个人多花点时间也能完成。做项目的程序员则成了项目组的一个成员，他只是负责整个项目的一个部分，或者说只编写其中的一段程序，而不是全部。因此，他的程序必须要和其他人编制的程序对接，他的程序必须读取别人的数据，他的数据也可能被别人读取。这里的每一个环节都不能出错，一个地方出错就会影响整个项目。所以他必须和团队的其他人很好协作，共同完成程序。

4. 学习氛围

没有做过项目的程序员靠自学，靠网上百度去学，学的内容随意性很大，学好学坏没有人监督。做项目的程序员不但靠自学和靠网上百度去学，还必须向项目负责人学，向项目组其他人学，向客户学。而且学的东西都有针对性。向项目负责人学习程序设计的详细方案，向项目组其他人学习程序接口、数据接口，向客户学习业务及需求等。程序的好坏要通过测试环节和用户使用加以验证。所以，通过参加项目，程序员可以克服自以为是的错误观念，树立为客户编程的思想，以软件销售价值来衡量自己的价值；树立团队意识，把自己融入团队中，以团队荣为荣，以团队耻为耻；在项目中学会从大局看待程序设计，学会评判程序难易之处，学习更加实用的程序方法和算法。

那么什么是项目？这里所指的项目可能和一般的项目定义侧重有所不同。这里的项目一般是指由客户提出需求，软件公司或企业内部项目小组按照需求进行设计、开发、投产、维护等工作的总和。它只包含软件相关的费用，其他硬件、网络、软件环境费用不在此考虑之列。项目有大有小，有的大项目以亿计，有的小项目以千计，千差万别。由于没有标准，

不同的人对项目的大小定义也不同。例如，有的企业把一百万以上的软件称为项目，把1千万元以上称为大项目。有的小企业把1万元以上的软件称为项目，把5万元以上称大项目。这些项目大小主要取决客户对资金管理的范围和等级，一般而言，项目越大，需要单位或企业越高的领导层批准。

而我今天说的项目大小是从软件项目本身来确定的，与客户对项目大小的定义没有什么太大的关系。我认为项目大小可以从以下几个维度考虑：资金、开发人月和项目复杂度。

1. 资金

我认为在当今物价状态下，5万以上50万以下为小项目，50万以上为大项目，500万以上为特大项目。

2. 开发人月

同理，2.5个人月到25个人月以下为小项目，25个人月以上为大项目。

3. 项目复杂度

软件项目的复杂度还可以用软件的用户使用人数、数据库中表的数量、表的记录数来衡量：

软件使用人数：10～1000人为小项目，1000人以上为大项目。

数据库表的数量：20～100张为小项目，100张以上为大项目。

表中的记录数：10万～1000万为小项目，1000万以上为大项目。

此外，项目运行能够给客户带来的收益大小、项目的业务逻辑的复杂度都可以成为项目大小考量的内容。如果项目都不能达到小项目的水平，我们这里就不把它看做项目了，因为低于小项目的项目很多是个人编程，这与项目众人参与的特点有点不符。

所以我对程序员的建议是：

1. 要主动参加项目

无论是大项目还是小项目，程序员都要努力参加，因为只有做了项目，自己的能力才能提高。不要静静待在那里，等待别人挑选，而是积极主动表示加入项目的愿望。在我负责过的项目过程中，我对主动要求加入项目的程序员往往给予更多的机会，因为这样的程序员具有主动性，工作更好开展。一个项目的出现就是一个机会的出现，把握项目就是把握机会。机不可失，失不再来。

2. 不要放过小项目

程序员不要以小而不为，只有做过若干个小项目后，程序员才能去做大项目。那些想一步就做大项目的程序员，往往会失去小项目锻炼的机会，往往参加到大项目后，感到力不从心。项目虽小也同样可以锻炼人，程序员可以有更多机会体验项目负责人的角色。学会从整体角度看待编程。

3. 要积极准备参加大项目

对于已参加过小项目的程序员，一定要把握机会，积极准备参加大项目，项目越大，越锻炼人。在大项目中要学会摆正自己的位置，虚心向团队其他成员学习。要在平常没有项目的时候多做些技术准备，多关注可能的大项目开发内容。在项目开发中则可以把重点放在体会不同功能模块之间的关系上，学会从关联的角度看待编程。

根据我的经验，我认为程序员要经过 5～6 个小项目的锻炼才能入门，而经历了 3 个以上的大项目的程序员才开始成熟。当然我们不能排除程序员的天才成分，有的程序员会在很短的时间内达到一个很高的水平。但是，绝大多数程序员成长是必须通过项目来催化的，尤其是大的项目催化更加重要。说白了，项目如同阳光，程序员如同禾苗，关系就是那么简单。

3.2 “顶梁柱”与“螺丝钉”的不同

程序员从新手开始，度过成长期，接着到了成熟阶段。程序员从“螺丝钉”发展到“顶梁柱”，这应该是一个很正常的轨迹，这个阶段有很多标志，其中一个重要的标志就是程序员是否是这个团队的技术骨干或是“顶梁柱”。程序员确定自己是否属于“顶梁柱”，一般可以从以下几个方面综合考虑：

1）工作年限是否在2～3年或以上；
2）是否已经担任项目经理职务或更高职务；
3）是否能独立承担一个项目或项目中主要的模块；
4）工资和奖金是否比同时进单位的同事高；
5）是否经常受到上级的表扬；
6）是否要接受比别人更多、更难的开发任务。

如果这几个方面都是肯定的回答，那这个程序员应该是一个技术骨干或是“顶梁柱”了。如果一个程序员虽然编程年限很长，工作也很努力，但是在单位或团队仍然是一个“螺丝钉”的作用。像这种程序员就不能说它是一个成熟的程序员了，他还需要重新认识自己，改变自己，提高自己，有些程序员甚至可以考虑是否要跳槽或转行，以便适应自己发展的要求。

但是，要真正成为一个“顶梁柱”，光靠满足以上外在的条件是远远不够的。在现实工作中我们常常看到一些“顶梁柱”干的是“螺丝钉”的活，而且乐此不疲。例如，我有一个程序员朋友，工作认真仔细，刻苦钻研，技术水平也不错，深得领导的欣赏，工作若干年之后，就被任命为开发组的领导。但是其工作方式在很长时间都没有改变，只知道自己干活，不知道管理，不知道做哪些才是“顶梁柱”的事。我还有一个朋友和他是同一时期的，也是在项目后被任命为项目经理的，从事管理工作，他则与前面的朋友做法完全相反，他的想法是自己的发展空间不是程序员，尽量脱离程序员的工作，转向管理工作。一个人的发展的因素有很多，决定发展因素也很难说得清楚，但是，就这两个朋友在同一阶段的表现，其

实都可以做得更好。他们共同的问题是在转向“顶梁柱”的时候，一个转向速度过慢，一个转向过快，这些都影响了他们有更好的发展。

程序员要成为真正的“顶梁柱”，需要注意以下几个方面问题。

1. “顶梁柱”意识

刚进入程序员这个职业的时候，程序员都有自己的理想和幻想，很多人都把比尔·盖茨作为自己追求的偶像，希望有一天能在软件业中创出一番天地。这是一个理想，而理想是靠自己一步一个脚印来实现的。从入门开始，程序员感觉理想离自己太远，这是由于自己的自信还没有建立起来，看不到理想可以实现的一天。到了成熟阶段，无论在技术方面还是工作能力方面，程序员都有了很大的提高，更重要的是程序员同时看到了软件制作销售全过程，看到了软件的价值所在，也看到自己的价值所在。所以，内心埋藏很久的理想促使程序员要挺身而出，挑起大梁，充当顶梁柱。程序员要具有“顶梁柱”意识，表明程序员不甘平庸，追求自我价值最大化，这样程序员才能向更高的方向前进。

而无意识成为“顶梁柱”的程序员，即使能在一段时间内有所发展，但是毕竟这种“顶梁柱”缺乏内心的动因，发展时间过长，而且很难达到更高的境界。

2. 主动意识

“顶梁柱”程序员和“螺丝钉”程序员之间的主要差距在于，一个具有主动意识，另一个具有被动意识。主动意识反映出程序员在工作中的主动性，程序员主动关心项目进展，主动关心项目的设计，主动承担项目的模块，主动了解开发进度，主动解决遇到的问题等。被动意识反映出程序员在工作中的被动性，程序员所有的工作都是由上级安排的，程序员只关注自己那份工作，对项目或别人的工作从不关心过问，叫做什么就做什么。

程序员只有拥有主动意识时才能做好“顶梁柱”，当项目出现困难的时候，不用上级分派就能挺身而出，充当“顶梁柱”这个角色。从而使得

工作问题以最快的方式解决，使得工作向前迈进。

程序员主动性的基础往往是技术能力和工作能力，若没有足够的技术能力和工作能力，程序员是无法说出“让我来”这种豪言壮语的。但是也有一些程序员虽然有一定的技术水平，但对工作却无主动意识，一切听从安排。其结果就可能丧失了更多展示自己技术能力的机会，同时也给工作造成拖延和被动。这样的程序员就不可能称为成熟的程序员。

3. 做大事、做难事意识

程序员要树立“做大事、做难事”的意识。随着企业的发展，软件开发任务加重，各种开发越来越多，有的项目大，有的项目小，有的项目有技术难度，有的项目技术简单。因此，这里就有一个选择的问题。从理论上来说，合适的人做合适的事是最科学的，否则就会造成大材小用，小材无用的最坏情况。因此，成熟的程序员应该做大事、做难事，新手和成长期的程序员应该做辅助的事、简单的事和小事。这样整个工作才能整体向前推动。

有了这种意识，程序员就可能练就一双评判项目大小、难易的眼睛，就可能在大事和难事中得到更多的锻炼，从而让自己的技术和工作能力得到进一步展现和提高。

4. 放弃意识

“螺丝钉”工作作风是认真仔细，一丝不苟。开发任何一个程序都一视同仁，用时均匀，而且喜欢从头到尾的串行做法。这种做法在程序员能力还比较弱时，还不能驾驭程序的时候，是非常有用的。但是，在程序员到了成熟阶段，这种做法就不利于程序员的成长，不利于工作的整体开展。因此，程序员要学会放弃，这样才能得到更多。程序员放弃自我，可能得到别人的进步；程序员放弃非重要的编程，可能保证有时间解决重要的编程。程序员这种取舍意识，其根本在于程序员已经有很强烈的团队意识，他舍去的工作由团队中的其他人完成，而他将要做的恰恰是需要他来做的。

在现实中，我们常看到一些程序员，尤其是技术水平高的程序员，什么都要自己去做，尽管自己时间不够，有很多事要做，但是他宁愿自己加班加点，也不会把其中简单的工作交给其他人去做。他们的理由是自己做得快，而让别人去做，自己还要教别人，有这个时间还不如自己做了。这种想法在开发任务不紧张的情况下有一定的道理。但是，随着开发任务重、时间紧的情况出现，这种“不放弃”的意识和做法反而使自己成为影响到整个项目进展的主要因素。

5. 注意“顶梁柱”与“螺丝钉”的关系

我在上面列举的例子中，程序员并不能摆正“顶梁柱”和“螺丝钉”的关系。一个是将两者等同起来，不分彼此；一个将两者对立起来，有他无我。因此，程序员要正确处理两者的关系。首先程序员要意识到“顶梁柱”是“螺丝钉”发展的一个更高阶段，“顶梁柱”有“顶梁柱”的工作特点。当程序员发展到成熟阶段后，应该迅速地转变角色，转变成“顶梁柱”。同时，程序员也应该认识到“螺丝钉”的作用，一个项目没有“螺丝钉”同样也是完成不了的。因此，程序员一方面要发挥其他“螺丝钉”的作用，另一方面自己也应在必要时候，充当一下“螺丝钉”，否则，在没有其他任何一个“螺丝钉”的情况下，自己又不愿意当“螺丝钉”，又不会当“螺丝钉”时，那这个项目就会停滞下来。

6. 榜样的意识

“顶梁柱”应该是一种很高大的形象，这种形象也是一种榜样。成熟的程序员应该能够解决开发中遇到的各种问题，能够找出解决问题的方法和途径，在技术上成为新手和处在成长期的程序员的榜样，是他们心目中的技术“高手”；同时在工作上积极主动，勇挑重担，不怕困难，乐于助人，处处体现团队精神。程序员有了“榜样”意识之后，就会更加严格地要求自己，会更加注重自己的言行，会让自己在工作中发挥更大的作用。

一方面，从某种意义上来说，程序员的“顶梁柱”地位表明程序员进入一个成熟阶段，它会给所在单位部门创造更多的价值。从另一方面来看，程序员主动的“顶梁柱”意识，反映出自己最初理想从泯灭到复出的

希望，他看到了自己的价值所在，同时也看到了自己理想可实现的端倪。

3.3 如何对待新人

程序员到了成熟阶段后，从技术层面上来看，早就没有了编程恐惧心理，对编程的流程、编程的技术应用也掌握得非常好；从工作层面上来看，程序员已经适应了工作环境、工作氛围、人与人之间的关系等，无论是在技术层面还是在工作层面都形成了个人的一套处理方式，这个时候我们可以用“兵来将挡，水来土掩”来形容程序员的那种成熟程度。

在这个阶段，个人的成熟往往是和在单位的发展成正比的。如果在此过程中，所在单位程序员得不到发展，程序员往往会选择离开。所以个人成熟时候往往是单位招收新人的时候，正是单位发展的时候，也是团队扩张发挥团队作用的时候。只有发挥团队作用才能做更大的项目，程序员的价值才能有更大的提高。

到来的新人往往是程序员过去的一个影子，他注定要沿着程序员以前的成长轨迹成长。在这个成长道路上，程序员所遇到的痛苦和困惑，新手都将会面临。历史就是这样不断重复着。

有一些程序员面对新人时常抱着“他是他的，我是我的”这种事不关己的态度。和我们常说的，做了婆婆之后，忘了做媳妇的艰辛一样，很少主动关心新人。有的甚至怀有莫名的敌意，害怕新人成长后抢了自己的饭碗。有的则将自己等同于新手，和新手比高低，看不起新手的一言一行。

也有一些程序员则完全相反，对新手表现十分热情，只要新手有问题，他们有问必答。这种做法个中因素很多，其中有建立人与人之间关系的需要，但是大部分是出于自我的技术表现。在很多情况下，这些解答都是正确的。但是有些情况下，解答并不正确。不少程序员往往对此不太重视，这种不正确的解答却往往会误导新手的成长。

在我的工作经历中就遇到过上述这种情况。一名新大学生被分配到一个技术不是很出色的程序员手下工作。结果这名大学生深受这个程序员的

影响，其发展进度远远低于我们的预期。我们后来花了很大的精力才慢慢把这个程序员对其的影响减少到最低限度。这个例子让我感到新手工作上的第一个老师对其影响十分重大。安排新手的第一个老师十分重要。程序员感到自己技术能力、工作能力和工作态度还不够成熟的话，我建议不要带新手，否则真的误人子弟。如果自己各方面比其他程序员都强，则可以当这个新手的老师。这样不但对新手负责，也是对自己负责。

正确对待新手，其实就是正确对待自己的过去。一个成熟的程序员在面对团队中的新手时，无论是领导的安排，还是没有安排，心中都要装有帮扶新手的心。有了这个关心之后，当新手有问题的时候，则会耐心地给予回答，直到新手弄懂为止。当新手提不出问题的时候，要主动关心，指出他们将要面临的问题，并给出一些有益的建议。总之，让新手乐于找到自己，让自己的技术与其共享。

其实，面对新手的辅导并不完全是一种给予。在一些情况下，新手提出的一些问题，程序员未必就能回答上来，程序员给出的答案未必是正确答案。这个时候，程序员最好要反思自己在这个问题上的表现，为什么自己对这个问题没有深刻认识？为什么不能给出正确答案？这样通过新手的提问，程序员就可以找出自己技术上的薄弱环节。

当然，这是一种理想的状态，在现实中的很多情况下，无论是主观还是客观上的原因都会使程序员与新手关系大打折扣。但是，这并不影响成熟的程序员善待新手的那颗心。我认为成熟的程序员与新手相处的时候要注意以下几点：

1. 平等

作为程序员在和新手相处的时候，首先要树立平等之心，要主动把自己放在一个和程序员平等的地位。千万不要因为自己的技术能力、工作年限、等级职位而抬高自己看低别人。那种高人一等的目光只要稍稍落在新手身上一点点就会被新手的感觉无限放大，而铭记在心。对于新手来说，这种目光要么产生了恐惧，要么产生了忌恨。当有一天新手成长超过自己，千万不要怪新手目光中有高人一等的得意，因为这种目光是你教会

他的。

2. 耐心

新手要问的问题一定很多，有的听起来会让人感到幼稚可笑，有的新手有一种打破砂锅问到底的执著，越是不懂越是问。在这个时候，程序员要特别耐心，要一一解答。这样你才能拉近和新手的距离。否则，一不耐心，就会让新手远离自己，不敢再问你问题了。这样新手的进步步伐就会被你打断，你自己也会失去团队中的一个帮手。

3. 主动

新手虽然会主动提出各种问题，但是，很多都是万不得已的情况下才问的。这主要是新手刚刚入门，技术上的自信心还没有建立，害怕别人看到自己的弱点。所以，很多新手自力更生地学习，努力提高自己的技术技能，这种自学成材出来的程序员，就像温室里的花朵，经不起风吹雨打。所以，程序员要理解新手的心理，主动地和新手打成一片，常常主动询问新手还存在什么问题，让新手把埋藏在内心的问题说出来。

4. 关注

对新手程序员要给予更多的关注，毕竟团队中任何人的成长都有助于团队整体能力的提升。关注新手就是要做新手成长的有心人，注意观察新手入门和成长过程中出现的各种进步和问题。及时发现问题，及时解决问题。尤其发现新手正在走自己过去走过的弯路的时候，一定要提醒这个新手，让他步入正确的轨道。这样新手一定会感知你的关注，心中有了依靠，心中就少了许多恐惧。很多过来的程序员都感叹，要是自己在新手阶段有高手关照的话，自己要少走许多弯路，成长更快。

5. 榜样

程序员千万不要满足于热心回答新手的问题。其实，新手的模仿能力是很强的，程序员的一举一动都会在自觉和不自觉中对新手产生影响。“榜样的力量是无穷的”，程序员不但要在学习上做榜样，在工作上做榜

样，而且要在做人上做榜样。我们可以看到一个奇怪的现象，那就是一个成熟程序员手下的同伴同质化现象十分明显。如果这个程序员是一个内向性格，那么他的手下员工大部分也不太说话；若这个程序员会写工作日记，那么他手下的员工也会去写工作日记；这个程序员喜欢用什么工具，那么他的手下一般也用这个工具。

6. 激励

新手的自信心很大程度来自于别人对其工作的肯定。程序员要了解新手的心态，对新手的每一点进步最好给予鼓励和肯定。对新手应该布置其力所能及的工作以及花一定精力和时间才能做到的工作。这样在一个工作评估点上，程序员在对新手的工作成果方面能肯定的地方要给予肯定。对新手出现差错的时候，不要一味埋怨新手能力差，而是提出如何避免差错的方法，以免下次同样的问题再度发生。通过对新手进行激励可以大大增强新手的自信心，可以让新手感觉自己在团队中的价值，从而有更多的动力去把学习和工作做好。

程序员要把新手看做自己昨日的影子，要像爱护自己的过去一样爱护新手，要关注新手的成长，把新手看做团队一员，要把新手成长看做提升团队整体能力的一个重要方面。程序员要牢记对新手的付出就是对团队的付出，就是对自己未来的付出，这种真诚的付出一定会结出丰硕成果的。

3.4 有关程序员的性别、年龄、个性、编程方法的话题

我们关心一个人是不是适合从事某项工作，往往是从他自身的特点和从事的行业属性来看的。而程序员作为很多人眼中科技含量相对较高的一种职业，是不是会有一些特殊的地方，让一部分人更适合从事呢？

1. 性别

在长期的编程工作过程中，我遇到了很多男男女女的程序员，总的来说应该是男多女少。有时候一些女孩家长会问我女孩学计算机好不好这样的话题。我总是很迟疑地回答：还好吧！对程序员这项工作来说，男女应

该差别不大的。一般来说，男程序员能做的事，女程序员也能做到，因为这毕竟是一个脑力劳动，并不是拼体力的。在我遇到的程序员中，最好的女程序员要比最好的男程序员强得多。女性天性的好强、心细、有耐心都是优秀程序员的基础素质。但是，也有很多女程序员表现平平如同表现平平的男程序员一样，很多女程序员不能适应这项工作就很快离开了，导致沉淀下来的女程序员的人数越来越少。在实际工作中，许多情况下要加班加点和要出差，这些对于女程序员来说就不太合适了。总体来说，程序员这个职业无论是男是女都是适合的，关键是看个人的基础素质和发展潜力以及机遇。男性在大局观方面应该有优势的，女性在编程细节和坚持方面比男性更有特点。男性能出高水平的程序员相对比较少，而女性则不出则罢，一出一定一鸣惊人。在当今男性作主导的世界里，当你看到一群披着男人外衣的程序员默默无声地工作着，水平默默保持不提高的时候，亮点几年都看不见的时候，猛然发现几个思维敏捷、编程迅速、成效明显、水平极高的女程序员出现在你面前的时候，你就会感叹“中国男儿不如女”。所以作为中国的程序员无论男女真的要有男子气概，要有男人的胸襟，要做“比尔·盖茨”，要肩负中国软件业振兴的使命，从小做起，从实做起，让自己一步一步走向技术的高峰。

2. 年龄

程序员的年龄问题更是很多人津津乐道的话题，年轻人满怀自豪地占据了程序员这个青春饭的领地，仿佛越年轻水平越高似的。年过三十的程序员看到黑压压的后来者，看到他们嘴里喊着自己不懂的新技术的名词，感到自己落伍了，感觉自己不再青春了，于是要求脱离编程队伍了；而年过四十的程序员更是面对自身的年龄压力和外界对年龄的压力，外界普遍认为四十的程序员编不了程序，只是苦于出路难寻，苦中潜行罢了。现实真的如此吗？很多公司都不招收最年轻的大学生，而钟情于有工作经验的程序员。而二十多岁的程序员心里始终没有底气，不知道自己的水平是高是低。三十多岁的程序员往往是公司和单位的骨干，有的是软件设计师，有的甚至是项目经理。四十多岁的程序员虽然老骥伏枥，但是通过大浪淘沙，沉淀下来可能都是金子，在重要岗位发挥着重要作用。程序员工作有

两个方面的要求，一方面是智力方面的要求，另一方面是体能上的要求。年轻人在体能上可能会占优势，工作效率上会高出很多，但是在智力方面则需要慢慢积累才行；而中年人经过长期地经验积累，智力方面的优势则越来越明显，而且体能方面的工作可以减少些或交给年轻人去做，而专攻设计和主要或重要的模块。程序员绝不是拳击运动员，主要靠年龄打天下，而是像棋类运动员一样靠智力打天下。以年龄说事的程序员，说好听点是以程序员体能的小方面掩盖了智力的大方面，说的不好听是以此给自己无力竞争找个台阶下，或者以年龄来打击比自己水平高的同行罢了。我想寄语给程序员：不要太关注年龄问题，而是要关注能力问题、水平问题。年龄大小并不表示水平高低，关键要看一个人的努力和成功！

3. 个性

不知道大家关注不关注程序员的个性问题。有两种个性的程序员值得我们去关注和思考，一种是夸夸其谈、不知羞耻的个性，这类人几乎在所有行业都存在，但是在计算机行业却容易蒙蔽不少不明真相的人。另一种是沉默寡言的，如同沉默的羔羊，如同金陵牌蚊香“默默无蚊”，有一种“打死都不说”的固执。

对于前者，我每当看到这种人的时候，我就会帮他寻找羞耻两个字。关键是在你苦苦寻找的时候，而他永远若无其事，照样夸夸其谈，照样在听众的疑惑不解或惊讶或佩服中自我陶醉。这些人可能是互联网害了他们，搜索引擎害了他们，他们太轻易获取这些最新信息了，他们太不了解别人也和他们一样能同样获得这些信息。他们从没有做过什么大的项目，甚至是小的项目，更不用说做过项目负责人了；他们从没有写过精彩的程序让人能复制粘贴，哪怕是3行5行；他们从没有就程序员关注的技术问题哪怕一个细小的问题能由浅入深地、系统地、生动地讲解清楚。不客气地说，那些满嘴的英文缩写字母，哪天冷不丁地让他写出缩写字母的含义来，3个字母中有两字母的单词是会拼写错误的，对此我们并不吃惊。他们做了偷吃仙果的孙猴子，偷吃了不要紧，关键是偷吃后，不知吃了什么，还要炫耀，还以此为荣，以此为业，那就不对了。对这种披着程序员外衣的人，我总有恨铁不成钢的感觉。要知道程序员不仅要和计算机打交

道，而且还要和其他程序员进行交流，并且彼此之间要互相学习，还要向客户学习，还要学习应用的业务，还要了解程序使用的成效。以上哪一点不需要我们敞开心怀与外界交流呢？在实践中，我曾遇到很多这类的程序员，出现这个问题一方面是程序员封闭、不爱说话的个性使然，但是更多水平比较低的程序员爱面子，怕别人看不起，于是就什么都不说了。但是，你不说就能保证你有面子吗？就能说明你水平不低吗？这完全是掩耳盗铃！一个人的水平是客观存在的，并不是你说行就高，你说不行就低的。关键是要有增强自己能力的意识和紧迫感，千方百计地寻找机会和别人交流，既要主动听又要主动说，通过交流看到自己的不足，并且找到解决自己的问题的方法。长此以往，程序员的水平不提高也难。放眼望去，你们身边公认的高手哪个不是善于言语、善于交流的人呢？

4. 编程方法

编程方法是仁者见仁，智者见智的问题，过去的程序员喜欢独立思考，每条语句都是独立思考出来的，货真价实。现在的程序员更喜欢追逐别人，喜欢抄袭，复制粘贴。前者主要是由于社会工作节奏比较慢，有时间发挥个人才华，后者主要是由于社会的工作节奏加快，社会压力加大，导致无时间进行自我思考。两者都是社会环境造就的。我的建议是：第一，我们提倡个人独立思考，但不反对复制粘贴的编程方法。第二，我们提倡复制粘贴，但是我们不反对个人独立思考。对于成熟的、公认的解决方案，我们坚决复制粘贴，绝不浪费自己的宝贵时间去编写，但可以花时间去学习。对于没有答案的、没有公认的解决方案则要充分发挥自己的主观能动性，独立思考，尽量用自己的思路编写出来。切忌不管懂与不懂先复制粘贴再说。在实际工作中我真的发现有人为了一个小功能，能把一个模块都复制过来，而且不知道整体要做些什么，只用了其中的一个功能。程序员要懂得“等价交换”法则，不要老是复制粘贴，而是在复制粘贴的同时也要想到如何被复制粘贴。当你被别人复制粘贴的时候，表明你的水平已经相当高了。

对于性别、年龄、个性、编程方式这些话题我并没有展开深谈，我只想从一个侧面来看待程序员这个群体的生存方式，让程序员更加丰满起

来、鲜活起来、真实起来。这样我们才能更好地与程序员打成一片，融为一体。

3.5 程序员的上升空间在哪里

我见证过许多程序员的成长，他们很多人在进入成熟期之后，技术上相对较高，一般项目开发起来比较自信，没有什么太大的困难，有的职位上也有所提升，成了项目经理、设计师，有的甚至是到了管理者的位置。又经过很长时间，这些程序员却再也没有什么变化和发展，工作稳定、情绪稳定，好像一切都在按部就班地进行着。有的程序员满足于现在的收入和职位，安于现状；有的程序员却在安于现状的同时，苦苦思索，但却找不到自己的上升空间。

我想说的是，满足于现状的程序员并不能成为优秀的程序员，永远也不会实现程序员最初的那个梦想。而对于那些思索未来的程序员来说，探索自己的上升空间是一件非常重要的事，有上进心的人始终是这个社会追捧的对象。那么这个阶段程序员的上升空间在哪里呢？这是一个仁者见仁，智者见智的问题，同时也是和程序员个体情况相关的问题。因此，我们可以用尝试的心态，提出自己的看法和建议：

1. 技术上的反思和提高

无论程序员个体之间差异有多大，我认为程序员上升空间在于技术上的反思和提高。程序员到了成熟阶段，往往有种狂奔到终点后的感觉，身体疲乏无力，特别需要休整再战。要知道程序员在“奔跑”的时候，一般是在使用自己的体力，很少用到跑步的技术和技巧。尤其是到最后更是依靠体力来支撑。从技术提高的曲线来看，新手的技术提高速度最快，无论是从时间还是从急迫性这两个方面，对新手的压力都特别大。而到了成熟阶段，其编程时间也大大缩短了，学习技术的迫切性也降低了，所以技术提高速度就慢下来许多。在现实中，我们真的看到有些年轻的程序员还看不起那些老程序员的技术水平呢，就是这个道理。

因此，成熟的程序员在安顿之后，一定要坐下来反思自己，看看自己

的技术发展之路，哪些走得比较踏实；哪些走得比较急促；哪些是自己的技术特长；哪些是自己的技术薄弱环节。反思后一定要抽出时间抓紧学习，把自己的技术明显不足的地方给补上，这样程序员的技术水平加上丰富经验和工作能力会使得自己更加具有实力，可以应付各种挑战，为今后的发展铺平道路。

而那些顺其自然的思想使得程序员缺少技术提高的主动性，程序员有明显的技术上的问题，也不主动去解决，而是拖之再拖，从而使得程序员的成熟度大打折扣。

2. 走向软件设计师

其实很多程序员走到这个阶段已经在做软件设计师工作了。这是绝大多数程序员发展的必经之路。因为程序员在编程序的时候，一方面会实现设计的功能，另一方面也在对设计有一个理解、认识、熟知的过程。进而慢慢地从学习别人设计转向自己参与或独立设计。如果这个程序员真的非常喜欢程序员这个职业，则可以转向软件设计师。这个转变很大程度上是程序员意识的转变，就是说要把编程的时间逐步分摊到软件设计上，要把重点从编程实现转变到项目的功能设计上。我这里强调的是“逐步”这个过程，程序员要逐步减少编程的时间，增加设计的时间，要克服编程的强大吸引力，要主动地攀上设计这个台阶。一个设计师是很难完全不编程的，而且我认为，会编程的设计师才是真正的设计师。一般这种转变需要花上一到两年时间。

3. 走向项目经理

也有一些程序员工作机遇比较好，不但负责了项目的设计工作，而且负责了项目管理工作。对项目经理这个工作有了一定的尝试。因此，这类程序员可以审视自己是否适合这项工作。如果自己对这种工作比较满意，而且具备项目经理的要求，则可以向项目经理这个方向发展。项目经理承担着项目管理的职责，对项目负主要责任。它和程序员的作用也不相同，项目经理的重点已经从编程转移到对人、对技术、对进度、对项目的管理。由于软件的项目经理与软件项目的相关性太大，因此，他必须了解软

件开发的各个环节，了解开发的各种技术和运用，了解开发队伍人员的水平和特点。所以他依旧和程序员脱不了干系。从程序员到项目经理可以使得项目经理更好地理解程序员在项目中的地位和作用，了解软件开发的各种规律性的东西，从而保证项目正常完成。而且项目经理的收入水平在公司中也是比较高的。因此，我认为走向项目经理是程序员的另一个发展空间。我想提醒的是，程序员在担任项目经理之前最好把软件设计工作做好，这样在做项目经理的时候就会有很好的基础。

4. 走向管理者

很多程序员希望自己能成为公司中的主管、经理、老总、老板，这样无论收入还是在公司的地位都比较高一些。另外，一些程序员可能厌倦了年复一年的编程工作，对工作产生了抵触情绪，希望能摆脱这种步步紧逼的工作状态。走向管理者其实要求程序员应比一般人做更多的准备，要做更多的转型工作，并不是想当管理者就能成为管理者的。但是，如果程序员有这个志向和爱好，又积累了这方面的工作经验，自己也感觉在这个方面能够发展，更重要的是有这样的机会，走向管理者也是可能的。由于软件公司的管理者毕竟不同于一般公司的管理者，专业能力越强，管理起来就越得心应手，没有专业能力的管理者，其遇到的问题很多，也很难解决。因此，我建议程序员最好要把编程、项目设计、项目管理等基础打好，这样转型到管理者成功率就会高一些。

5. 走向软件营销

我一直说，程序员最大的缺点之一就是眼光只盯在技术上。若程序员在入门阶段和成长阶段，这种情况还属于正常情况。但是，到了成熟阶段这种缺点将严重影响到程序员的发展。程序员在工作中不免会接触到用户，不免会了解到用户的需求，从而了解到软件实际发挥的作用，进而了解自身的工作价值。有的程序员则故步自封，不愿意和外界打交道，从内心里坚守技术底线。有的程序员则喜欢和外人交流，喜欢把自己的软件推荐给用户，而且用户也乐意接受这种专业性的介绍，用户的接受是程序员价值的直接实现，程序员因此获得了工作成就感。这类程序员可以将自己

的专业知识和产品营销结合在一起，推动软件的销售和服务，转向软件营销职业。这个职业将是一个比程序员更有发展的职业，它可以直接反映程序员的工作能力和工作成果，程序员的收入与销售会直接挂钩。因此，可以让有才华的程序员获得更高的收入，接触到更大的软件市场，看到更广的个人发展空间。

6. 原地发展

当然也有一些程序员惰性比较大，满足于现状，享受着眼前的胜利果实，也无可非议。对待这些程序员，我们不要强求他们改变自己的想法，我建议是原地发展，把本职工作做得更好，把自己的技术水平往上再提高一步。在这个过程中他们的能力和自信心都会得到提高，他们的想法不会一成不变的。磨刀不误砍柴工，刀锋利了自然砍柴的时间就少了。

7. 跳槽走人

还有一种情况，程序员工作很努力，技术水平也有很大的提高，工作成绩也很突出，但是，公司的工作环境让他感觉不好。例如，收入和付出差距很大，而且短期内看不到加薪的可能；自己的技术水平已经超出公司开发所需，英雄无用武之地；公司的经营状况比较差，看不到公司的未来；公司的人际关系紧张，小人当道，自己得不到重用等。他可以选择跳槽走人。要说明的是，这个时候跳槽走人你是有本钱的，这个本钱就是你积累的工作业绩、技术水平和工作能力。

程序员在成熟期面临着上升空间的选择，是上还是不上往往决定了程序员今后的发展。水往低处流，人往高处走，只要程序员有向上的意识，不满足于现状，根据自身的特点和资源选择发展方向，我想程序员一定会有很好的发展前途。

3.6 跳槽还是留守

相对其他行业而言，IT 跳槽的频率还是比较大的。尤其是市面上不时传来跳槽就能涨工资的传说之后，很多人更加乐意考虑跳槽了。

跳槽是IT行业一个特点。其原因主要是这个行业的发展历史太短。任何行业发展初期，由于专业人才处于形成时期，人才非常缺乏，这就造成了人才稀缺，各家企业为了在发展中占得先机，就必须通过提供更高的薪水方法从别的企业挖走人才，招到自己的企业之中，只有这样企业才能在竞争中取得优势。这就使得人才有很大跳槽的空间，跳槽也就频繁发生了。随着行业的发展，这个行业的专业人才会越来越多，从而造成了行业中的人才过剩，跳槽的空间就会缩小，频率就会降低。跳槽遵循着人才从低端开始向高端发展，人数从多向少方向发展，频率从高向低发展的规律。从软件业发展的角度来看，当前正处于一个起步晚、发展初的阶段，因此，人才的稀缺程度虽然没有前十几年高，但是，中高端的人才依旧稀缺。

还有一个跳槽的原因是软件行业存在价格差异的特性。不同的软件公司研发相同软件的成本是不一样的，销售价格也不一样，而且可以存在很大的差距，这使得一些公司有超额利润支付其高额工资。这个就造成了能够获得超额利润的大公司可以提供更高的薪水岗位，更多的人才向大公司流入。也有一些小公司因为能获得超额利润，也同样能以更高的薪水招收到想要的人才。因此，软件行业的跳槽很普遍就不足为奇了。

程序员发展到成熟期，技术上应该相对成熟了，但是，在工资收入上能否满足自己的预期就很难说了。大部分程序员因为自己能力有了很大的提高，对自己的价值评估也比以前高了许多。绝大部分的程序员是很难对自己的收入满意的，因为无论他收入多高，他总能听到比他收入更高的传言，他总是将自己的收入和传说中的收入进行比较，差距越大，心理越是不平衡。尤其是听到曾经和自己共事过的，而且技术比不过自己的同事在别的公司拿到更高收入时，这种不平衡到达了顶峰。

是坚守还是跳槽？程序员切不要单单考虑收入高低因素，还要综合考虑其他因素。一个人的职业规划并非是一个钱字说得清楚的。跳槽有跳槽的好处，但也有坏处，坚守有坚守的坏处，但也有好处。得失权衡，切莫冲动行事，很多事要早考虑、早准备。

毕竟程序员到成熟期一般要三年左右的时间，在这期间应该积累了很多资源：技术、团队地位、职位、人际关系、领导印象、产品、客户资源等。这些资源虽然不像薪水是一个显式的财富，但一定是一个隐式的财富。这个财富是自己用时间和精力换来的，岂能轻易丢弃。

我认为跳槽还是坚守主要考虑以下因素：

1. 薪水

薪水是跳槽还是留守的重要因素，在考虑这个因素的时候，要做横向和纵向的比较，要做本公司和外部公司的比较，如果比较的结果相差比较大，并且这种差距自己不能够忍受，且加薪在短期之内无法实现，则要考虑跳槽，反之留守为好。

2. 领导赏识

领导赏识也是你考虑的重要因素之一，如果平时你的上级领导或是老板对你的工作非常肯定并对你信任和赏识有加，你则可以考虑留下。有的时候，领导的赏识并不一定给你带来薪水上的增加，但这种赏识会对你今后的发展是十分有利的。如果你在企业中、团队中默默无闻、可有可无，而且也得不到领导的重视，而且在未来的岁月里看不到任何变化的话，选择跳槽可能更有机会一些。

3. 企业成长性

如果从理论上来说，企业成长性是你应该考虑的最重要的因素，企业的未来和员工是息息相关的。企业成长性可以从它的发展历程和发展速度中看出端倪，也可以从企业的市场占有率、软件产品、企业开发力量和销售能力等多个方面看出来。如果企业成长性很好，就如同买了绩优股长期持有是没有错的。如果看不出企业有什么成长性，早点离开则是非常明智的。

4. 个人能力和地位

程序员的能力也是跳槽重要因素，因为新企业招收的一定是能力更强

的程序员，如果程序员没有真实的能力，即使侥幸地跳槽成功，在新企业也不会有很好的发展，我也听说过有的程序员水平太差，跳槽过去不久，在试用期中就被辞退了的故事。程序员有能力是一回事，能够充分发挥这个能力又是另外一回事了。如果现在的公司不能充分发挥自己的能力，自己的能力过剩，则跳槽非常有必要。有的程序员发展得很好，成了项目经理或部门主管，这些职务也体现了程序员在这个企业中的社会地位，如果能跳到比现在职位更高的岗位，跳槽是必需的。但是，如果是往地位低的职位跳，尽管薪水高一些，还是多多考虑为好。

5. 其他资源

其他因素也是去留要考虑的。例如团队的因素，有的程序员和团队相处很融洽，工作关系和人际关系相当好，害怕离开之后再也找不到相同的环境了。

跳槽还是坚守？程序员要把各种因素综合起来考虑，而且要慎重考虑，这是一个关系程序员前程的重要决定。说实在的，有的人跳槽跳得很成功，有的跳槽成瘾；有的坚守也很成功，有的坚守一直到底；也有跳槽不成功，坚守不成功的。这种成败也许并不在于跳槽和留守的选择，而是跳槽和留守都是一个机遇，关键是程序员内在的能力与把握这个机遇的结合。要提醒的是，程序员不要总是考虑跳槽还是留守这个问题，一旦这个选择决定之后，至少在半年或是一年之内就不要再考虑这个问题了。毕竟这些考虑会影响到程序员自身的工作心态，影响到与之有关的方方面面。

3.7 你能当老板吗

打工的是指被发工资的人，“老板”是指发工资的人。很多人有这样一个逻辑：老板是有钱人，当了老板就有钱了。于是乎当“老板”就成了许多人“脱贫致富”的一种理想和追求。

程序员经过了一个成长期发展之后，慢慢地成熟起来，原来那种单纯地把技术学好，把工作做好的想法也有了一些变化。尤其是那些发展较好的程序员（最好是一个人能独立开发软件的程序员），他们一方面在编程

技术上提高很快，另一方面在项目设计、项目管理上也开始了尝试。最重要的是他们看到了软件销售的金额和软件销售的市场。他们发现只要能把软件开发出来，然后把软件卖出去，自己也能开公司当老板。尤其是当自己的报酬和自己创造出的价值相差巨大的时候，这种当老板的念头更加强烈。

我记得“不想当将军的士兵不是好士兵”这句名言，对于程序员去当老板，我举双手赞成。无论程序员是在心中想当老板的，还是真正迈出当老板一步的，都是值得那些不想当老板的程序员尊敬的。毕竟我们更推崇那些有理想、有勇气的人。

很多人认为这个人当老板是不会成功的，但这个人却成功了；很多人认为这个人当老板一定会成功，但这个人却失败了。这些例子比比皆是。这说明当老板并没有一定的要求，每个人去当老板都可能有其当老板的理由；不是局中之人是没法体会的。所以对自己不了解的东西不要轻易给出自己的判断。市场之中，一切皆有可能。

在我身边的程序员也有去当老板的，有成功的，也有失败的。在这么多年的IT岁月里，我也特别留心程序员当老板的各种报道，总想从中悟出道理来。很多人就是从软件单价＊销售个数得出的巨大金额而动心去当老板的。例如，程序员预计这个软件单价为2万元，可能的用户有100个，2×100＝200万。程序员就会为这200万而下定决心去当老板了。

我认为程序员当老板应该注意以下几个问题：

1. 要有坚定的信念

要想当老板，不能只想不做，或者今天想做，明天又不想做了，反反复复，犹犹豫豫，这样当老板的机会就会丧失了。要做，那就要坚定信心，精心准备。准备越充分，信心就会越充足。

2. 要有独立开发能力

当软件公司的老板，并不需要太多的资金。开始的时候，一般只有一

个人，或是几个人就能开张了。正是因为如此，当老板的一定要具有独立的开发能力，有了开发能力，才能保证开发周期，才能保证按时向客户提供软件。否则，依靠创业伙伴或招收新员工都有很大可能在规定的时间内完不成开发任务，或保证不了软件质量。这样会造成创业失败。

3. 要有强烈的市场意识

当老板首要的条件就是要具备市场意识，具备市场意识意味着程序员要把工作重心转移到：软件用户在哪里，软件能卖多少钱，软件能否销售出去，成本是多少，利润是多少，是赚是亏，诸多方面。如果这种意识不强烈，脑子里还是考虑程序怎么编，程序要采用什么新技术，程序要有什么技术含量，这些技术问题上，那这种程序员真的不能当老板。因为当老板应该把绝大多数时间放在市场上，这样市场才有可能给你回报。就如同程序员把大部分时间放在程序上，程序员的编程技术才会提高。

4. 要有客户资源

很多创业者一般都会有一两个客户，他们会理所当然地认为，同类的客户都可能是自己的用户。其实这是一个很大的误区。潜在的客户并不能很容易地成为自己的用户。不少创业者做了一两个单子后，就因为缺少持续的用户而坚持不下去。所以当老板一定要首先考虑你的客户在那里，要营销多少客户公司才能维持下来，怎么样才能营销出这么多的客户。

5. 要有克服困难的准备

从现实的角度上来说，开公司当老板并非易事，我和软件公司的老板们交谈过他们的苦衷、苦水、苦头。

6. 要有凝聚力

凝聚力是非常重要的，在创业初期，除了要明确合作伙伴与公司之间的承诺，开发人员与公司之间的用工协议之外，还要把他们凝聚在一起，使大家为了一个目标而共同奋斗。这种凝聚力取决于领头者要具有那种给予同伴以信心，给予同伴以信任，给予同伴以支持，给予同伴以榜样的素

质和能力。

7. 要有管理能力

当老板要有管理意识和管理能力，开始的时候，程序员的角色还没有转变。以前是别人要自己干什么，自己处于一种被动和等待状态；现在则是要求别人干什么，自己处于一个主动和积极的状态。管理意识则要在创业前就要有所准备，管理能力是逐步提高的。随着企业的发展，管理越来越重要，不重视管理、不具备管理能力，企业就会很快垮掉。

8. 要有沟通能力

这里的沟通能力主要是指创业者的外部沟通能力。程序员由于长年和计算机打交道，客观上和主观上与外部人员接触比较少，因此，其沟通能力是比较弱的。但是，当老板必须要有沟通能力，这样才能了解到市场信息，才能把自己的产品介绍给客户，才能进行商务谈判定下合同。

总的来说，程序员在成熟之后，当老板未尝不可。但是，在此之前，要做些当老板的准备。准备越充分越好。即使失败了，也可以重新去找工作，也可以再一次等待当老板的机会。

3.8 动手与动脑的关系

不知道有没有人观察到这样一个现象：很多学习非常非常刻苦的人，往往学习成绩并不太好，甚至赶不上平时那些贪玩的学生。这种现象让很多人都迷惑不解。在常人看来所谓学习刻苦就是花在学习上的时间长，很多家长只要看到自己的孩子不去玩，不去看电视，不去上网，所有时间全部花在看书做作业上面，就感到安心了许多，就以为花在学习上的时间越长学习成绩就会越好。他们的逻辑是学生不花时间在学习上，其成绩一定很差，而花在学习上的时间越长，学生成绩就会越好。我同意这种逻辑的前半部分，不同意后半部分。这是因为，现实中有很多反例都不支持这种说法。其实，学生成绩好坏固然和学习时间长短有一定关系，但在相同时间内（如上课时间加上若干小时的做作业时间），不同的学生成绩是不同

的。这说明影响学生成绩的因素不在于学习时间。那学习成绩好坏和什么有关呢？当然是学习方法。那么哪种学习方法才能产生好的学习成绩呢？这是一个仁者见仁，智者见智的问题，很难用几句话说清楚。但是，方法很重要，方法决定成绩（反对天资决定成绩的观点）。

回到程序员这个话题。我们发现很多程序员存在同样的问题。我们经常看到一些程序员花了比其他人好几倍的时间，编出来的程序却很差。这些人往往是最可能影响整个项目进程的人，而项目经理往往对其毫无办法，这是因为他们非常认真地在编写不合格的程序。其实，程序员并不是编程时间越长就水平越高，并不是编程时间越长编写的程序就会越好。时间长短与熟练程度密切相关，但是，时间长短和水平之间的关系并不同步。

这些程序员坚持的观点是：程序员就是编程序的，就要天天在计算机上编程序，程序有问题是正常的，有问题只要改好就行了。他们表现出动手能力超过了动脑能力，而且还为动手能力强而沾沾自喜。问题是他们看不到自己这些观点存在什么问题。但是，他们能感受到的是不断地加班加点，感觉到编程的疲劳。并且认为这一切全是项目本身引起的，自己是项目的受害者，自己并没有任何问题。

程序员进入成熟阶段，早已过了培养动手能力的阶段。程序员一定要注意工作方式的转变，那就是由以前的动手转为动脑。实现动手与动脑的分离，逐渐把时间花在动脑上。这样才能避免编程中的无效劳动，提高编程的有效性和合理性，从整体上缩短开发周期。

1. 边想边做方式

程序员都是从边学边做，从边想边做走过来的。这种惯性非常大，少有人可以改变。若不信，你可以有意去了解身边的程序员，统计他们在计算机上所用的时间，这样就会验证这种惯性了。现实中绝大部分的开发项目远非一加一那么简单，而且项目将会越来越多，越来越复杂。这就从客观上产生了编写大量程序的需求。如果按照原有的开发方式，我们只有通过增加人力的方式去满足这些变化。但是，这种方式会加大成本支出，而

且也不能保证项目实效性的要求。所以，这就要求我们开动脑筋，改进原有的开发方式，通过开发方式的转变来适应不断增加的开发需求。所以说边想边做的方式不能适应软件业迅速发展的要求。因此，只有把动脑和动手分离开，加大动脑时间的比重，让编程时间不要间隔，一编到底，该思考的时候，不去动手，专心分析，专心想办法。这样我们整体的开发效率会有很大的提高。

2. 编程会出现反复和浪费

在现实中，很多程序员都有这种经历：编写好的程序很快就会发现问题，于是就会不停地修改，有的甚至会推倒重新编写。这种情况不在少数，而且占用了程序员大量时间。为什么出现这种现象？其中原因很多，但是，我认为，这都是和程序员对需求的一知半解，对程序的算法不能确定造成的，而最根本的原因是，程序员没有养成动脑的习惯。他们没有在这些方面花足够的时间去研究和比较各种方案、各种可行性和各种合理性。程序员想到哪里编到哪里，好了就留下来，不好就重来。这种编程模式造成了很大的浪费。

3. 编程会占用动脑时间

一个人一天的工作时间是有限的，花在这个方面时间多，必然花在另一方面时间少。动手时间长了，必然会造成动脑时间减少。而动脑时间少了，那就意味着寻找到解决问题的最佳方案的可能性也少了。这就意味着程序在各方面都不是最优的，而程序存在问题的可能性增大，程序开发时间变长。从某种意义上来说，到了某一阶段动手编程技术含量很少，它只是过去劳动的简单重复。而一个问题解决方法并非一个，它是无限的，它取决人们对其研究和思考，对其规律性的掌握。因此，程序员应该把更多的时间花在功能如何设计，编程细节如何最优上。

4. 动手与动脑的时间比例

程序员只要大概统计自己一天的动手时间和动脑时间，然后进行一个比较，就能知道自己在这个方面的问题了。如果动手时间占90%以上，那

就说明程序员还没有动脑的意识；如果动脑时间占30%左右，说明程序员已经有了动脑的意识；如果动手时间和动脑时间各占一半，说明程序员已经开始重视动脑了；如果动脑时间大于动手时间，说明程序员已经成熟起来，他已经懂得动手和动脑的专业化分工。动手的部分可以交给专门的编程人员，动脑的部分可以交给专门的设计人员。这对于他今后从事项目管理工作大有好处。

5. 改变工作方式

说起来容易做起来难，程序员边做边想的做事方式是一种职业习惯。如果让他们一下改变过来，那将是一件非常困难的事。但是，再难我们也要去做，否则，我们永远只能停留在普通程序员的水平上。我们可以用渐进的方式来逐步实现这种转变。比如，我们可以尝试每天有半个小时不碰计算机，用这个时间来思考自己要编写的程序，从结构到细节，都一一考虑一番，看看其中有什么问题，看看有没有更好的方法等。如果我们能坚持半个小时，那么接下来我们就可以坚持1个小时；如果我们能坚持1个小时，那么我们就能坚持2个小时。这样动脑的时间就会不断地增加，这样新的工作方式就产生了。一旦你习惯了这种工作方式，再让你回到边想边做的做法，你一定不愿意了。

所谓动脑筋就是让程序员能够静下心来想问题及其解决方法。专心才能致志。当然专心不是那种呆坐在办公桌前一个人想问题，我们可以独自地收集资料，可以上网查询，可以对资料进行分析，可以想出几个解决方案，可以比较方案的优劣，可以征求其他相关人员的意见和建议，可以把每个功能的实现算法考虑得很仔细等。做有其限，思无止境。

古人云：三思而后行。这个道理对于程序员来说尤为重要。光想不做的思想家容易成为空想家，光做不想的容易成为实干家，而成为实干家后，再去想再去做而成为思想家就要比一般的实干家和思想家更加强大。可以说动手是基础，动脑是动手的高级阶段。编程的天地是有限的，而思想的天空是无限的。当我们意识到将工作方式提升到动脑方式时，就真正把握了软件是人类智慧的结晶的本质，我们的价值不但体现在软件制作的

过程中，而且体现在软件本身具有的智慧中。我们可以让自己熟练起来，让自己充满更多智慧。

3.9 编程语言有高低之分吗

记得在很久以前就被教导过程序语言分为低级语言和高级语言。高低这两个词比较单纯容易记忆，而且从小就被灌输要高不要低的价值观。因此，那时候自己就有要学高级语言而不学低级语言的简单朴素的想法，仿佛学习高级语言自己就会高级起来一样。

没想到今天依旧存在这个话题。无论新手还是老程序员中都有许多人都在问："什么语言是最好的？什么语言是最高级的？"言下之意就是要学习最好的语言，只要掌握了最好的语言，自己就是最好的。我看到有人这样说：地球人都知道要选择 C#、Java。你能说说为什么吗？也看到了有人对自己原先选择 VFP 感到后悔，因为他认为 VFP 语言已经被淘汰了，同时他对自己没有选择 C 语言感到后悔，因为他认为使用 C 语言才能看出程序员的水平。归纳起来有以下几种看法：

1）什么语言是最好的？

2）我现在掌握的语言是否已经或将要被淘汰，我今后要学习什么样的语言？

3）C 语言是真正的高级语言。

这些问题从其本质上来说，它们都给程序设计语言定下了高低之分，并且它们想知道谁高谁低。那么程序设计语言是否有高低之分呢？根据我长期编程经历和经验，我认为：

1. 程序设计语言本质上没有高低之分

所谓程序设计语言是指让计算机能够听懂并能指令计算机进行操作的语言。每种语言的产生都有一定功能范畴，也就是说这个语言是为了解决某种问题而产生的。不同的问题就可能用不同的语言来解决。不同的问题

是很难用高低来区分的。例如，基于计算机底层的操作（如汇编语言、C语言）与应用于计算机应用层的操作（如C#、Java），很难说清楚哪个是高级的，哪个是低级的。又例如，基于单机程序的VFP和基于网络的ASP、C#也是很难说清楚的（虽然VFP也可以用于网络编程，ASP、C#也可以用于单机编程）。他们的侧重点是不同的，而这个侧重点正是这个语言的与其他语言区别的特点。因此，没有任何一种语言是最好的，哪怕是暂时的。C语言的确是一门高级语言，也能反映出程序员的技术水平，但是C语言有它的适用范围。我们不能以自己不懂的语言，或使用人数少的语言，或其稀缺就感到其高级。

2. 程序设计语言处在一个不断发展的过程中

没有一种语言是静止不变的，几乎所有语言都会随着技术进步而不断升级。同一种语言确实存在高低，版本高的语言因为改进其版本的问题，增加了新的功能，扩大了语言的应用范围。所以程序员要关注本语言的发展，不断跟踪语言发展的最新结果。有的语言由于其最初所设计的功能范畴满足不了现在流行的技术架构，就逐渐地被人淡忘和淘汰了。原来很流行的基于单机的、基于C/S架构的语言，如BASIC、VB、VFP、PB、DELPHI等语言渐渐地消失了。

认识到语言处在一个发展的过程中，跟上语言发展的步伐，这才是程序员必须牢记的。这就可以回答地球人都知道要选择C#、Java的理由了吧。不同语言的发展步伐不太一致，越是靠近计算机底层的语言发展得越慢，但是不等同于不发展。例如，C语言就增加了大文件打开的系统函数。

程序员确实要有危机意识，要主动关注语言发展的大趋势，如果你学的语言正好是主流的、流行的语言，那你可以减轻这方面的压力；如果你学的是非主流的，而且其编程工作呈下降趋势，则要考虑学习新的语言，否则真有一天会被淘汰。

3. 要牢记学以致用的原则

程序员选择何种语言从某种意义上来说并不取决于自己的喜好，而是

为了自身的生存，由外部来决定的，其总的原则是学以致用。例如，现在外部流行 B/S 架构，表明市场需要在这个架构下编程的程序员。程序员要想就业，增加就业的可能性，必须选择 C#、Java、PHP，否则学会了其他语言却无法使用，就业难度会增大。又如，单位需要后台和数据库方面的程序员，那程序员就必须学习 C 语言和 Pro C 语言。再如，软件公司使用 Java 编程，而你又想进入这家公司，那你就必须学习 Java。

其实，“编程语言是否有高低之分”是个伪命题，不管有或无，都无关大碍。关键是程序员要掌握一门用于生存和学以致用的语言。

学习语言有一个很漫长的过程：自然语言→程序设计语言→多种程序设计语言→程序设计语言存在高低之分→程序设计语言不存在高低之分→选择学以致用的程序设计语言→跟踪程序设计语言发展的趋势→领会程序设计语言的本质→回归自然语言。

每个程序员都可以静下心来，客观地了解自己处在这个过程的哪个阶段，只有这样才能用一个发展的眼光看到自己在学习语言方面的前途，迫使自己向更高的阶段迈进。

3.10 面向过程和面向对象的编程

这个话题其实很矛盾，我担心许多程序员对此话题不太感兴趣。很多人不太懂得面向过程和面向对象编程的巨大差距，以及对编程水平的影响。

在我的整个编程生涯中，有无数的转变和变化让我铭记在心。从汇编到 C 语言，从文件系统到数据库，从 C 语言到 Dbase，从 C/S 到 B/S，从程序语言到开发平台等。但是让我感觉最大的、最根本的转变是从 C 语言转学 C ++，可以说这是一个革命性的转变。在当时我对 C 语言的运用已经炉火纯青，得心应手，感到只要用 C 语言就能完成所有应用程序的开发。但是，学了 C ++ 之后，我才恍然大悟，程序还可以用另外一种思维编写。这个转变过程极其痛苦，每个概念的接受都需要和 C 语言进行对比，都要不停地问这是为什么？这整整花了我几乎半年时间，而且每分每

秒都让你在不解，似懂非懂中度过。我学习一种语言，主要是学习这个语言的架构和基本原理，也就是说要研究这个语言的由来和特点。又经过若干年后，我才真正意识到面向对象编程的重要意义。它绝对是程序员思维方式的变革和升华，它使得程序员可以从具体编程提升到抽象的结构。使得程序员可以编写更加通用的类，进而编写通用的软件产品。

人们从小就学习1、2、3、4，头脑里排满了次序！绝大多数人都是按照次序进行思考。这种次序就是一种过程。例如先做什么，后做什么。做得比较好的，表现做事有条理，做事很有逻辑性；做得不好的，则把过程做得歪歪扭扭，但是最后也能达到预期的结果。同样，绝大多数程序员都是按照面向过程思维方式编程。尽管他们用了面向对象的程序设计语言，学了对象、类，用了类，但是他们仍然用面向过程的方式编程。

1. 面向过程编程

面向过程编程体现了程序员的逻辑性，许多程序员的水平就体现在这里。如何设计语句的次序，如何组织函数，如何设计函数的参数，如何设计过程分支等都体现了程序员的逻辑能力。例如编写保存学生信息功能的过程：

（1）空值处理

我们首先要判断学生信息项是否有空值，如必输项为空，则给出提示。例如，学生姓名为空！

（2）合法性检查

然后，我们还要对各信息项的有效性进行判别，如果有无效信息则提示重新输入。例如，如学生的出生日期不符合日期格式！

（3）产生学生ID

可以给出自然数，也可以给出有意义的编码！但是ID必须是惟一的！如不惟一则进行相应的处理。

（4）打开数据库

建立数据库连接，获取数据库句柄。

（5）组织 insert 语句

将操作者输入的学生信息拼成一句 insert 语句（实现这个功能有很多种方法）。

（6）提交数据库运行

将 insert 语句提交给数据库。

（7）操作日志

将以上操作信息写入日志。

（8）提示写入成功

提示“保存成功!”向操作者提示操作成功!

这些是我很随意给出的保存学生信息的过程。当然，不同的程序员可能会写出不同的过程，但是最终的结果是，学生信息保存到数据库中。处理的内容越多，不同程序员处理的逻辑越不同，可谓五花八门，而正是这些不同，体现了程序员的水平。但是，这种面向过程编写的程序只能用于具体的系统，却不能用于其他系统。例如，以上的程序只能用于学生信息处理程序。

2. 面向对象编程

面向对象编程体现了程序员的抽象能力，其本质是把一个事物或多个事物看做一个抽象的对象。还是举保存学生信息功能的过程：

1）我们可以把保存学生信息功能看成一个对象（这里仅是举例，不一定合理），注意我们这里先不要管面向过程中的那些处理流程，仅仅把它当做一个对象、一个空集；

2）我们为这个对象设计属性。例如输入信息、数据库连接串、数据

表名、日记表名等；

3）我们为这个对象设计事件；

4）我们编写这个类；

5）我们在保存功能的命令按钮里调用这个类。

面向对象编程思想与面向过程编程思想是不同的，前者把学生信息抽象成输入信息，把学生信息表抽象成数据表。基本上与我们所要编写的具体学生信息无关。这种抽象使得编写的类有可能用于学生信息处理、客户信息处理、员工信息处理、人口信息处理中的信息保存功能。

我并不是说面向过程不好，因为绝大多数程序是面向过程的，许多程序员面向过程的逻辑性能力还有待提高，因此，程序员还需要从基础做起，提高面向过程编程的水平。根据我的经验和体会，能够把面向过程的程序编好也不是一件很容易的事！新手首先要学会面向过程编程，有经验的程序员要提高面向过程的编程水平。而系统的结构、模块的结构、功能的结构、函数的结构、控件的结构都应该用面向对象的思维进行设计。但是总体上来说，从面向过程编程到面向对象编程是一个思维方式提升的过程，作为程序员应该对此有一个清醒的认识，而这一点正是优秀程序员必须具备的条件之一。当一个程序员能从具体的程序抽象出对象，并能把这个对象用于其他程序，这个程序员则完成了技术水平的蜕变，跻身优秀程序员行列了。

3.11 功能和界面哪个更重要

程序员编程序离不了功能和界面。有的涉及界面多一些，有些涉及少一些，也有一些后台维护的工作甚至不涉及界面。当前一些软件公司专业化程度较高，不但设计和编程分开了，甚至界面制作和功能制作也分开了，设立了专业的美工岗位。但是大多数单位和软件公司还没有出现这种界面和功能的分离。有关功能和界面哪个更重要的话题，时常在很多程序员之间不经意的交谈中流露出来。在工作中我也常常需要面对界面和功能

开发安排进行选择。有的人强调功能重要，有的人强调界面重要；我自己有时会强调功能重要，有时也会强调界面重要。那么到底哪个重要？在这里我想谈谈我自己对两者认识的过程。我想通过介绍这个过程，希望对程序员进一步认识功能和界面的关系有所参考。

第一阶段：两者不分

在自己编程的初期，真的没有什么界面和功能之分的概念，界面是都靠程序写出来的，总是感觉功能和界面是同一回事。那个时候的界面大都采用行字符形式，没有控件和图形，只有简单的线条和字符（包括汉字）。例如 C 语言主要是用 move、print 来实现界面。加上那时候用户对界面没有什么太高的要求，所以我们根本不把界面当做一回事，只是把界面当做程序的一部分，而且当做一个简单的部分。

第二阶段：功能重于界面

后来，程序的功能慢慢多了起来，系统也越来越大了，用户的界面也多了。但是，当时的重点还是放在功能设计和功能实现上。项目组的主要精力和人员调配还是放在功能设计和功能实现上，当然界面也开始有人做了，但是没有人专门去做。人们普遍的看法是编程是最有技术的，编界面是很一般的。当时的客户最关心的还是功能实现，主要是看程序能不能实现他的功能要求。

第三阶段：功能与界面并重

再后来，随着 Windows 操作系统兴起，各种图形界面运孕而生，客户对界面的要求不仅仅是实现功能，而且有美观、友好、易操作、提示、帮助等方面的要求。人机对话开始形成，界面开始形成了风格并流行。这个时候进行编程，你就会不自觉地把界面当做一个很重要的工作，你不仅要考虑功能能否实现，而且要考虑你的界面能否被客户接受，你的界面是否友好。这就迫使你要注重界面设计，迫使你学会欣赏和评价用户界面的好坏。你要在界面上花费很大精力。而由于程序员那时专注于功能设计，对

于转变至界面往往感到力不从心，很多人都会借口编程，而不敢碰及界面。但是，界面的重要性从此凸显出来了。

第四阶段：界面重于功能

再再后来，我发现界面很重要，有的甚至比功能都重要。当参加项目介绍、项目评选的时候，用户往往会选择界面美观的、友好的项目，尽管另外一个系统功能方面更优。用户往往首先从外表选定自己所喜欢的。有的甚至对 PPT 的做得好的给予更多的关注。

这就如同人们吃饱了，要娱乐一样。吃可能基本相同，但是娱乐就各不相同了。这个时候娱乐可能就比吃更重要了，而吃，因为有钱了，有了吃的保证就不显得重要了。以前人们是因吃而吃，现在是因娱乐而吃了。功能和界面就如同吃和娱乐一样。

人们流行这样一种观念：功能大家都是能实现的，但是界面就很难说了。界面反过来决定了功能。界面的这种重要地位和作用，很快让界面设计变成了专业，于是有了专业的美工，有了人机交互的概念和行业。

第五阶段：发现界面的本质

我经常反思自己对功能和界面的认识过程，为什么有这样转变？这些转变说明了什么问题？不记得是哪一天了，我突然发现这种转变是程序主导者发生了变化，程序由过去的程序员主导已经变成了客户主导了。这就是说，原来我们编写的程序，编写的界面都是由程序员主观进行认可的，程序员是主体；而现在竞争加剧，导致整个社会开始面向公众，企业面向客户的转变，反映在软件行业其中一个方面就是界面越来越重要。界面不仅仅是用户和计算机之间的接口和桥梁，更是反映了软件制作者对客户的一种服务态度。界面的本质是客户。只有面向市场、面向客户，树立“以客户为中心”的理念，获得客户对界面、功能的青睐，我们的软件才能销售出去，我们的软件才能被用户接受和使用，我们的功能和界面才有意义，我们的程序员才能够生存下去。

我对功能和界面的态度不是很硬性的，并不是说哪一个更重要。在项目开发初期，我强调的是功能，功能为先，程序员应该先把功能设计出来，界面不要花太多精力，能表意就行了。当功能基本上开发完成后，我就会把主要精力放在界面上，关心界面的风格、界面的色彩、界面色彩的协调性、界面在美学上的简明和平衡、界面风格的一致性、界面中用户的可操作性、用户操作后的容忍性、对意外出错的提示、界面向客户提供更多的联机帮助、界面向客户提供操作流程的易用性、界面使用通用流行的图标、界面是否不通过培训就能让用户正确使用等。

这些方面有些是和功能无关的，但是有些方面是和功能有关的，好的界面将会使功能更加丰富和合理。

我想寄语程序员的是：程序员要重视界面设计，不要把自己封闭在功能编码的小天地内，要通过界面设计，拓宽自己的审美概念，增强自己的客户意识，换位重新审视自己的功能设计，让更多编程以外的知识来丰富自己的编程思想，扩展自己的编程视野，使得自己编程水平得到根本性的提高。

3.12 你考虑过程序的复用问题吗

许多时候，程序员埋头编写程序，有可能会发现需求和功能是以前做过的，于是会找出以前编过的程序，复制粘贴，然后修改就搞定了。编完后，思绪又被下一个功能给吸引过去了，直到又遇到相同的情况，再次进行这些重复的步骤。程序员就这样周而复始地工作着。

程序员在入门和成长阶段，主要是把程序编出来，虽然也遇到很多相同的需求、相同的功能，都是将其编出来就行了。但是，到了成熟阶段，程序员应该重视程序的复用问题，应该把它放在一个很重要的位置来考虑。

第一，这是因为随着项目开发越来越多，相同功能出现的频率也越来越大，相同功能开发占整个功能开发的比重逐渐上升。一般相同功能占整个项目的比重会达到20% ~40%，有的项目甚至超过50%。

第二，复用的功能程序将影响到整个项目的质量。由于复用的功能编写主要是延续性编写，很多程序连程序员都没有仔细地看过，他们拿来就用，最多是在要修改的地方修改一下。因此，由于拿来的程序的编写年限很长，其功能很难和现有的需求完全吻合，加之拿来的程序经过多次复用，遍布漏洞，很难保证质量。许多问题只有在项目上线后才暴露出来。如果不采用复用方法，重新编写功能程序则质量更加难以保证，因为程序的质量是需要通过大量测试，使用后才能证明其是高水平的。

第三，相同功能程序编写将影响到项目开发进程。如果采用复用方式，则开发进度大大加快，如果采用新编方式，则进度将被放慢。

理论上程序复用是最经济最有效率的一种方法，但是在编程实际中也会存在一些问题。

1. 没有复用意识

有不少程序员认为，程序员天生就是编程序的，不编程序就不是程序员了，他们在意识上就是所有程序都要自己编，这样才能证明自己的价值。其实，这种观点是值得商榷的。信息化进程发展到今天这个程度，很多程序其实早就有人编过了，我们现在很多的工作是在做前人做过的工作，当然后人也会做我们的工作。如果我们树立复用的意识，我们就可以减少重复性劳动，将精力放在那些创造性的劳动上，这样我们的技术水平才会真正提高。

2. 只复用自己的程序

有些程序员知道复用的好处，也想复用，但是只是复用自己的程序，因为自己对自己的程序比较了解，复用起来比较方便。但是，由于自己编写的程序相对较少，可复用的程序也比较少，仍然不能充分利用别人的程序。所以要跳出复用范围的限制，花点时间向同事或在网上寻找复用的程序。

3. 怕烦

也有一些程序员认为向别人要程序、看别人程序、改别人程序这些复

用的过程很烦，所以宁愿自己重新编写也不愿意复用别人的程序。而且他们认为，这样时间可能更短。我认为这是一个工作习惯问题，在一些小功能和小程序上也许这个道理成立，但是，从整体上来说，复用程序绝对是比自己编写程序要快得多，好得多。程序员养成了程序复用习惯后，就会更加体会到程序复用的好处了。

4. 不愿意仔细阅读复用程序

有的程序员喜欢复用别人的程序，能拿则拿，的确要比自己编写强许多。但是，其中一些程序员在复用别人程序的时候，不喜欢对程序进行深入研究，只是把自己要用的那一部分参数和返回结果搞清楚就行了。至于这个程序的总体设计思想、功能情况、背景情况就没有兴趣去了解。这样往往会给程序质量带来隐患。很多欠考虑的地方就会在程序运行中出错。所以程序员在复用别人的程序的时候，一定要对这个程序有一个全面的了解，这样才能正确地使用这个程序。

程序复用本质上是企业经营活动中的规律性和重复性在企业信息化的应用系统中的反映。只要企业经营活动中存在规律，存在重复性的活动，程序复用就有存在的前提。随着企业经营不断向前发展，无论是生产技术还是管理水平都会向更高方向迈进，这种更高的方向表现为生产更加科学、管理更加专业、成本更低、效率更高、利润更多、市场更大。这些活动从另一个方面来说就是增加有效的重复活动，只有有效的重复活动才能减少活动的种类，减少活动中的人力和财力的成本，才能利用科技进步支持重复活动，从而达到企业经营的最终目的。所以企业经营活动中的规律性和重复性呈现越来越强的趋势，因此，关注相同功能，寻找相同功能，为项目功能制定标准，使用相同功能，加大程序复用是成熟程序员必须具备的一个重要意识。

在程序复用方面我们要注意：

1）在项目开发中，我们要跳开项目开发的本身，站在各种项目开发的历史中去寻找程序复用。这就意味着我们在编制程序的时候，首先要考虑这个功能在这个项目中是否已经开发过了，这个功能在其他项目中开发

过了没有，如果已经开发过了，则不要再重新开发了，拿来后做一些修改就可以了。养成这样的习惯后，我们则可以大大提高程序复用率。

2）程序复用是一个递减的过程，程序越小，复用的可能性就越大；程序越大，复用性的可能性就越小。越是与项目无关的公共类的程序复用程度越大，而与应用项目紧密相关的程序复用程度就很小。因此，程序员在分析功能和编写程序的时候，尽可能将功能细分，尽可能将程序细分，让函数的功能最简化，这样函数可复用性就可增强。所以，很多程序员要改变一气呵成的编程习惯，改变大而全的函数编写习惯。

3）很多拿到手的程序并不是一字不改就可以使用的，在大多数情况下，还是需要按照自身的需求进行修改。例如，在一个交易系统中，新增一个特殊的交易，那就可以把原有的交易处理作为复用的母本，在那基础上进行修改。

4）程序复用是有层次和等级的，最低层次是那些公共类的函数和控件等，上面还有大小功能，还有与应用有无关系等不同层级的程序。越是高层复用，复用效果越好，程序员的水平也就越高，因为这样程序员才能够真正意识到程序复用的价值和作用。

5）程序员除了关注程序复用之外，还要把更多精力放在数据以及数据结构方面的复用。在很多情况下，只有数据结构一致了，功能才能一致。所以程序员应尽可能先让数据结构复用，然后再让程序复用。

程序复用从某种意义上来说，它是功能标准化中的一个过程，程序复用的最终目的就是让被复用的程序变成标准程序，让程序在更大范围内复用。标准的程序由于其质量高、参数化程度高、适用范围广必然给程序编写带来时间上的节约和程序质量上的保证。因此，成熟的程序员要在主观意识上重视程序复用，要在自己的编程工作中尽量采用复用程序，尽可能编写高质量的可复用程序。“复用别人程序，被别人所复用”，做到这两点程序员才能算是真正成熟。

3.13 谈谈程序的参数化

对于成熟的程序员来说，程序的参数化是一种程序员水平的体现。他们认为参数化程度越高，系统的灵活性和可扩展性就越高，系统的内在价值就越大，同时他们认为参数化程度越高，对编写程序的要求就越高，编写程序的难度就越高。实际情况也正是如此。我想在此对参数化问题作一个初步的总结和归纳，让程序员对参数化有一个系统认识。

程序的参数化有两个方面的内容：一个是函数的参数化，另一个是功能的参数化。

1. 函数的参数化

函数的参数化主要关注函数要设计哪些参数才能满足函数的独立调用。

（1）函数参数化的由来

从程序员成长的客观过程来看，程序员在最初阶段往往不会注重程序的参数化问题，而是将很多精力放在函数的功能实现上，对功能的分类、分层、参数的运用并不会太注重，所以表现在程序上给人一种一气呵成的感觉。一个函数的语句条数很多，上百条、上千条都有。这样一来，函数结构性体现不出来，可共享调用的函数可能就大大减少。随着编程时间的推移，很多程序员更加乐意在一个功能函数中编写很短小的函数。函数多了，必然就会考虑函数共享和函数调用的问题，考虑这些问题，必然要考虑到函数的参数化问题。

举一个最简单的例子，假如在一个功能中，一个人的数学分数80分，语文分数90分，计算两门功课的分数之和。程序可以有各种编程方法：

第一种，不分函数

在这个功能中直接写下：

```
int sum=80+90;
```

这样就直接明了地完成了设计要求。

第二种，分函数

在这个功能中直接调取这个人的数学和语文分数之和的函数。

```
//-------------------------------------
int sum=get_sum(80,90)
//-------------------------------------
get_sum()
{
    int sum=80+90;
    return sum;
}
```

这种方法使得功能函数结构清晰、阅读性强（大家可以把 90 + 80 想象成一个处理方法，有几十条甚至上百条语句，这样就可能好理解一些），同时产生了一个函数 get_sum。但是这种函数没有采用参数，只能用于指定的人。

第三种，分函数，参数调用

```
//-------------------------------------
int sum=get_sum(80,90)
//-------------------------------------
get_sum(int x,int y)
{
    int sum=x+y;
    return sum;
}
```

这种编法采用了 x,y 这样的参数，这样无论是张三的数学语文成绩，还是李四的数学语文成绩，都可以调用 get_sum 这个函数完成分数相加操作了。函数可以被张三李四共享了。

由此可见一个问题可以有不同的编写方法。从结果来看，每种方法都是对的。但是，从技术层面上来看，各种方法将反映出程序员的眼界和技术水平。我们看到参数化是由于功能中结构化的要求产生了函数，又因为函数共享产生了参数。所以，程序员要牢记这一点，只要让函数进行共享就必须参数化。

（2）参数化和函数的关系

单从程序的结果来看，不同的程序员采用不同的方法都能达到相同的结果。但是达到这个结果并不意味着各种方法都是科学的。科学的方法评判标准之一就是这个程序能否被复用，能否参数化。若函数不能参数化，就不可能形成一个独立且与项目无关的函数，如果一个函数与项目有关，且不能被其他模块或其他项目所调用，就不能实现函数共享。程序员在设计函数时，一方面要考虑如何完成函数的功能，另一方面要考虑将功能中的变量、常量等作为函数参数抽取出来，并对参数进行处理，这个过程就是参数化过程。由此可见参数化与函数关系非常密切。

从某种意义上来说，参数化程度越高，函数难度就越大。所以一般程序员往往会编写参数化很弱的函数，这就是为了避免函数的难度。函数参数化的难度一般体现在以下几个方面：

功能中函数的划分

我们知道函数参数化是和函数有关的，函数又是和功能有关的，在一个功能中可以有一个函数，也可以有多个函数，那么函数在功能中的划分就显得十分重要。函数在划分的时候，一个很重要的原则就是合乎参数化原则，即在划分一个函数时，要确保这个函数的参数能够完成这个函数的功能。在上面的例子中，get_sum 中的 x，y 两个参数能够保证 x + y 的功能要求。如果这个函数中只有一个参数，函数则不能完成独立的相加功能。必然要有 x + 90 这样的语句，而 90 这个常数就破坏了 get_sum 的参数化原则。第二个原则是最小化原则，也就是说函数不可再进行拆分了。如果函数的功能还可以再拆分，则说明参数化不彻底，需要进一步进行拆分。

函数中的参数的抽取

在确定函数之后，程序员要尽可能地把函数中各种常量和变量抽取出来变成参数，这样函数才能完成参数化。例如 get_sum 中一共有两个常量 80 和 90。我们把这两个常量抽取为参数后，get_sum 则成了参数化

的函数。

参数的抽取和设计是一件需要技术的工作。以上例子中是以 x,y 两个参数来设计的。我们也可以用一个参数来设计，例如用 z 这个参数来设计。当调用 get_sum 函数时 z ="80,90"。这种参数的好处是，我们不但可以处理两个成绩之和，还可以处理多个成绩之和。例如，我们可以用 z ="80,90,85,95"参数来表示要将 80、90、85、95 这四门功课的成绩汇总。

函数中对参数的处理

相比对常量和变量的处理，对参数的处理要求更高。常量和变量的处理比较直观，容易编写，如果将其转化为参数处理，则需要程序员有更多的抽象能力。以上述 get_sum 中的 x,y 参数为例，这个函数将数学成绩、语文成绩抽象成两个数值再相加。因此，参数化的函数不如非参数化函数好理解。另外，参数处理还要考虑参数的范围，比如数值 x，我们就要考虑到 -100 到 200 这个范围（假定成绩都为 100 分制或 120 分制）。这个范围内的各种情况都要考虑，例如，0 分、负分等。假定函数中用到 x 分数去除某个被除数时，就要增加对 x 是否为 0 的判断。否则就会导致程序产生除 0 出错的中断，而常量和变量处理则无须过多判断。

参数化是无止境的

函数参数化工作是无止境的，这关系到程序员的抽象能力和模型算法能力。我们几乎可以对任何一个已经参数化的函数进行优化。尤其是功能复杂的函数更是如此。一般来说，功能越简单的函数，参数化越简单，功能越复杂的函数，参数化越复杂。

2. 功能的参数化

功能的参数化是指为了实现一个系统、一个模块、一个功能需要的那些参数，以满足客户需要的变化，而不修改程序。功能的参数化可能要比函数参数化更高一层。它关心的这个功能要有那些参数，这些参数对功能有什么影响。功能参数化更接近用户需求，而函数参数化更接近技术实现。比如，产品处理功能，其功能参数化考虑的是能否把产品代

码作为一个参数，如果产品代码变了，则相应的处理就会改变。而函数参数化也可能是功能参数一个部分，但是，它更多的是关注实现产品处理的过程。

（1）功能的参数化分类

功能的概念可以大到项目、小到一个事件触发的处理。因此，参数种类也是各不相同的。

按操作对象来分：

用户参数、功能参数

面向用户功能变化的参数往往是由用户根据需求变化而进行设置的参数，这些参数称为用户参数或功能参数，它是参数化的主要部分，关系到整个系统的灵活性和扩展性。例如，系统上线之初功能列表中可以处理3张报表，随着用户需求的变化，用户可以通过报表参数的设置，可以增加处理报表的个数到8张。

要特别指出，用户参数和功能参数往往和用户的业务做法有关，因此，在设计用户参数的时候，要能适应用户的业务要求。这必然要求我们的程序员对业务有所了解。在现实中，我们看到有些参数很“技术”，参数只能被程序员读懂，离业务较远。因此，参数化要“用户化”，设计出用户能懂、能改的参数才是参数化的一个重要原则。

系统参数

面向系统管理人员操作的参数往往是项目的技术环境中的那些参数，这些参数称为系统参数。例如，数据库名称、数据存放目录等，这些参数基本上和用户功能需求变化无关。

按内容来分：

标志类参数

标志类参数是指参数的内容作为一种标志来使用，这类参数是最

多的，它起到一个转移标志的功能，例如，参数为 A，则进行 A 处理，参数为 B，则进行 B 处理。功能函数根据参数内容转到相应的处理程序中。

操作类参数

操作类参数是指参数的内容作为处理操作中的一个操作数。例如，`get_sum` 例子中的 `x,y` 的值就是作为求和中的操作数。这类参数也在系统中广泛存在，尤其是在用户参数中广泛存在。

函数类参数

函数类参数是指参数的内容是为调用函数类而设定的值。例如，一个函数的指针、名称，WebService 的地址、名称等。通过这个值可以调用相应的函数和函数类的功能。它是实现函数参数化的一种高级形式。

常量类参数

常量类参数是指参数的内容通常为常量，主要是为系统环境而设立，一般不随着客户需求变化而变化，例如，系统名称、用户名称、数据库名、文件目录、菜单标题、窗口标题等。

（2）参数和功能之间的关系

功能参数设计是和功能本身密切相关的，而且与功能的最大应用范围和未来可能出现的情况有关。如果我们在功能设计方面能够考虑更加全面，更加适应未来的变化，那么采用参数形式来适应这些变化是一个必然的做法。这个时候，我们可以把功能看做一个模型和参数的结合体。模型通过输入参数而启动。作为程序员而言，我们则要在编程的时候考虑两个问题：我们的模型是什么？我们的参数是什么？而且，不同的参数，导致模型的处理方法不同。其实两个问题都是很难的，很多解决问题的方法都来自积累的经验。

（3）参数的形式

功能参数一般有 4 种形式：

1）文本文件

2）数据库

3）XML

4）交互输入

每一种形式都有其优点和适用范围，而且不同的应用系统采用参数形式也不惟一。一般而言，后台处理程序（C 语言）多采用文本文件形式，Web 类程序多采用 XML，各种应用系统多采用数据库，小程序多采用交互输入方式。从发展趋势的角度来看，XML 和数据库两种参数形式为更好的参数选择。

函数的参数化关注的重点是函数的共享，系统的参数化关注的重点是满足用户需求的变化。函数的参数化是系统参数化的技术实现的基础，系统参数化是系统灵活性和扩展性的保证。成熟的程序员要具有参数化意识和参数化技术水平，这种意识和技术水平为其成为优秀程序员打下了良好的基础。

3.14 漫谈程序的效率和水平

程序员常常将程序的效率和水平挂在嘴边。他们推崇高效的程序，并且把运行快的程序看成水平高的程序。但是很多程序员并不清楚什么是高效的程序，以及如何才能编制高效的程序。他们把编制高效程序看做一种奢望、一种追求。

程序运行的快和慢是相对而言的，其前提是相比的程序必须完成相同的功能，而且程序运行的硬件环境和软件环境必须一样。不同的人因其程序的不同，程序运行的时间也不同。因此，程序就有了差距。即使相同的人，因对程序进行了完善和变更，也同样会导致程序的不同，进而导致程序运行时间的不同。若没有这些前提，谈论程序的效率是没有意义的。

我们不能一概而论地认为，程序越快越好。不同程序对效率的要求是不相同的。现在许多程序员只知道程序越快越好；不知道程序效率改进是一个无止境的过程；不知道程序的快慢是有一个度的；不知道人们对效率的追求是需要成本的；不知道程序快慢与用户感受相关。而这些正是我最为担心的。我认为无论什么程序，只要有用户使用才有价值，用户的感受才是程序效率的目标。只有树立这个目标之后，我们改进程序效率才会有动力，我们改进程序效率才有一个尽头。我们可以把程序归为三种情况：

1. 批处理

指系统业务功能终止后或其他事物终了后，对其终了前的数据进行加工的过程，这个过程可能涉及多个程序。有的批处理时间很长，可以达到几小时甚至十几小时都有，有的批处理需要几分钟到几十分钟。批处理时间主要涉及数据量的大小、处理结果的大小、处理逻辑和处理逻辑实现。其中处理逻辑的实现就是批处理程序。一般而言，数据量大、处理结果内容多、处理逻辑复杂程序运行时间就长。需要特别指出的是，有的批处理时间会超过 24 小时，影响企业第二天的业务系统的运行，这样批处理就会对企业业务产生很重大的影响。

不同的批处理程序的编写能使批处理时间延长和缩短。例如，A 程序员编写这个批处理程序需要 10 个小时，B 程序员编写这个程序只要 5 个小时，C 程序员可能要 20 个小时。可见程序的效率是多么的重要，也难怪许多人认为程序效率高就是水平高呢。

2. 联机功能

联机功能是指操作人员在一个联机的应用系统中运行一个业务功能。例如，信息计算、保存信息、查询信息、打印信息等。联机功能运行时间应该是有限的，应该是以秒和分钟为单位的，不像批处理是以小时为单位。因为操作人员始终处于等待状态，他们是无法容忍长时间等待的。正因为如此，对于程序员来说，虽然缩短时间要求不像批处理那样长，但是总体时间要在限定的几分钟内，这样程序员压力就更大了。

3. 单个程序

单个程序是指为了某种数学运算或数据处理或其他处理编制的一个程序。这种程序的特点是一个人操作，运行到结束就是一个周期。例如：显示 hello World 程序、积分程序、从串口读出数据程序等。单个程序的运行时间就很难说了，可以用 0 - 无限来表示。很多单个程序都是由程序员一人来完成的，因此，其运行时间长短只能由其自己的程序的改进来感知。无论在什么情况下的程序，我都认为对程序的效率评判权应该交给使用程序的操作人员，而不是程序员。很多程序员站在自己的角度上认为自己的程序已经编得很好了，没有办法再缩短时间了，这种想法是极其错误的。程序员应该提高自己的编程能力，使得程序能够在操作人员可容忍的时间内完成。从理论上来说，只有运行时间无限延长的程序，没有运行时间不可缩短的程序。

例如，我们在编写批处理的时候，客户的容忍度是 4 个小时，你编的程序已经达到了 1 个小时，那你就不必再花时间修改你的程序达到半个小时。又例如，我们在编制联机处理的时候，客户容忍度是 2 秒钟，但是你现在的程序要运行 1 分钟，那你就有必要去修改你的程序，尽量达到客户的要求。又例如，我们在编制单个程序时候，我们就是为了练习编写程序，重点在程序的功能实现上，而不太关心程序的效率。那我们就没有必要去花时间修改自己的程序，自己会编程序后再来考虑程序效率的问题。

以上就是要帮程序员树立不同程序有不同容忍度的观念。有了这个容忍度我们再来看看如何在这个容忍度之中提高自己的程序效率。前面我们主要谈了提高程序效率要考虑满足用户的容忍度的问题，现在我们言归正传谈谈如何提高程序的效率问题。提高程序的效率涉及计算机基础知识、编程经验、程序硬件环境和软件环境、资金成本和时间等各个方面。依我的经验，提高程序效率要从以下几个方面入手：

1. 程序要简短

程序的效率本质是执行可执行代码（汇编指令）的次数。这一点是关

键的关键，程序员一定要牢记心中。程序越简短，其可执行代码就越少，就越有效率（如果程序是串行操作的话）。因此，我们在编写过程化程序的时候，要尽量改进我们的算法，让语句最少，源程序语句减少可以使可执行代码减少。

1）不要编写一些不起作用的“废话”，例如定义了不必要的变量；
2）不要编写不起作用的语句（有可能忘记注释掉的）；
3）不要编写不可能出现各种例外的判断语句；
4）不要编写一条语句能实现的功能非要用 3 条或 4 条语句来实现。
5）改进程序算法，改进算法其中最重要的一条就是减少语句。

但是并不总是程序短，时间就短。例如，我们可以编制 3 条导致死循环的语句。程序很短，但是时间却很长。其根本原因就是虽然可执行代码行数很短，但是执行可执行代码的次数是无穷的。在过去，有时候会把各自的程序（相同功能）拿出来比一比，看谁的程序最短。哪怕有一条语句的减少都被认为很了不起。所以“惜墨如金”程序员必须时刻牢记。长此过往，程序员就会养成一个良好的习惯。

我发现现在许多程序员对资源的意识很淡漠，仿佛程序设计语言中所有资源都可以任意使用，不需付代价，不需要理由。这种挥霍性的编程是一个不好的习惯，它会严重影响程序的效率。

2. 减少循环内操作

循环语句是影响程序效率最重要的原因之一，这里的循环语句一个是指程序员自己编写的，另一个是指程序调用的函数中使用的循环语句。

1）程序员编写的循环语句，第一，要尽可能减少循环次数，这样可以减少程序运行时间；第二，尽量减少循环内无用的操作，能在循环外执行的语句，就在循环外执行；第三，尽量减少嵌套循环，因为循环中的循环是两循环次数相乘的关系。若有可能可以变嵌套循环为顺序循环，顺序循环是两循环次数相加。

2）程序员要知道自己调用的各种函数中可能出现的循环操作，通过

参数，尽量避免循环次数的增加。例如查找子串函数，一定是包含循环操作的。程序员在使用这个函数的时候，一定要尽量减少被查找的字符串的长度，增加查找的字符串的长度。这样循环查找的次数就会减少。

3. 充分利用内存

利用内存是提高程序效率非常有效的方法，通过使用内存，我们可以缓存许多需要再加工的数据，无须进行 I/O 交换，这样就可以提高程序的效率了。另外，我们可以利用内存建立数据的索引，加快数据查询的速度，从而提高程序效率。

4. 减少 I/O 操作

很多情况下 I/O 操作是影响程序效率的最重要的原因，尤其是数据库大数据量操作更是明显。我们可以通过建立表分区、表索引的方式来提高数据操作效率，这些方法本质都是通过减少 I/O 操作来获取操作效率的。减少 I/O 操作有很多技巧和方法，但是无论哪种方法都是要把 I/O 操作减少到最低。例如，我们要输出 1000 万条记录到一个文本文件。一般的做法是形成一条记录写一次文件。如果，我们能形成 1000 条记录后，写一次文件，那么我们只要写 1 万次就可以了（其实上写文件还是很复杂的，它跟系统写缓冲区大小和硬盘本身结构有关）。这样程序效率就会有很大的提高。又例如，我们可以用低级文件打开方式代替流文件打开方式，这样效率会更高些。因为流处理最终还是要调用低级打开的。

5. 提高调用效率

在程序中，我们大量地调用系统函数、自己写的函数和各种引用的函数。这些函数的运行时间我们是无法改变的。因此我们要提高程序效率的方式有：

1）可以考虑使用功能相同但是运行时间较短的函数。

2）我们还可以自己编写一些函数替代系统函数或引用的函数，因为自己编制的函数所考虑因素比较少，功能比较直接，不需要考虑很多例外情

况，程序代码要比其他函数要少得多。

6. 使用全程变量

全程变量的使用可以减少传递参数所花费的时间。参数传递在可执行代码中需要花上 PUSH（压栈）和 POP（出栈）时间。但是，我不提倡使用全程变量，如果使用全程变量不能大幅度提高效率的话，还不如不用。但是，如果能够大幅度提高效率的话，使用全程变量也不失是一种好方法。

当以上的努力还不能使得程序效率达到用户容忍度的要求时，还可以从更高层面提高程序的效率，如提高硬件环境，选择更接近底层的程序设计语言，采用多机分布并行处理等方法。这个方法往往会导致更高的企业成本，关键是企业要评估其性价比。

一般人认为程序效率高，程序的水平就高。但是，我们也看到为了提高效率，我们可能就会更多地使用全程变量，就会申请更多的内存，就有可能破坏程序的结构，这样程序的水平反而会降低。一个好的程序不但外在要有好的效率，而且内在要有好的结构，要有最佳的内存效率比。高水平的程序一定是高效率的，但高效率程序并不总是高水平的。

很多人会问还有没有其他提高程序效率的方法，我的回答是肯定的。例如我们可以在可执行程序中嵌入汇编程序，以提高程序的效率。但是，这不是常态的做法，程序员只要了解就行了。

大部分程序员只会使用语言编写程序，但不知道这个源程序究竟是如何变成可执行代码和变成什么可执行代码，以及这些可执行代码的执行时间长短。因此，要从根本上提高程序的效率，需要程序员深刻理解从源程序到可执行程序的过程，深刻理解编译原理、汇编语言以及汇编中的系统调用。对数据库而言，程序员要深刻理解数据库原理、数据存放方式和数据查询方式，以及数据库操作与 I/O 操作与内存的关系。而这些理解需要长时间编程积累才行。这就是我为什么说程序效率的提高是无止境的。

3.15 好的程序像一首诗

从事编程工作这么多年，经常会有人问我什么样的程序是好程序，有的甚至把程序给我看，让我给程序提意见。而我从编程开始就踏上寻觅好程序之路。“路漫漫其修远兮，吾将上下而求索。”这好像正是我心路历程的写照。可以说追求编写好程序是我不变的目标。

好的程序？好的程序？什么样的程序是好程序？为什么这样的程序就是好的？我们从小就接受区分好与坏的教育，面对一件事物，总是喜欢评出好坏。评出结果，这一过程无非是想让复杂事物简单化，让人在第一时间能判别这件事好不好，这件事能不能做，好的就给予支持和赞美，好的就想去做，就想做好。反之，则被人们所摒弃。但是，好坏两个字好写，但是事物好坏的理由和标准却绝对的复杂，有时甚至绝对的对立。

“白日依山尽，黄河入海流。欲穷千里目，更上一层楼。”我们从小到大，经历过无数次教育，读过不知道多少本的书，看过不知道多少篇文章，能够记忆出来的，能够背诵出来的，能够打动自己心扉的，那就是诗了。诗成了生活的浓缩、提炼和升华。而诗人则成为历代百姓文人追捧的对象。虽然现代诗处于动荡和衰败过程中，但是经过历史长河沉淀下来的名诗依然脍炙人口，让人永生难忘，几乎找不到一个连一句诗都不会说的正常人，可见诗的魅力之大。

编写程序如同写文章，有的人会写成报告，有的人会写成小说，有的人会写成杂文，文体不限，各显神通。但是我以为，好的程序如同一首诗，读起来行云流水，韵味十足。诗中或对仗工整，或错落有致，或排比拟人，或比喻夸张，这些在程序中也能表现出来。

程序和诗歌都有内容和形式两个部分。内容是本质，反映了程序的功能，反映了程序的设计水平。形式是表现，反映了内容表述。好的程序应该像一首诗，有好的内容，也有好的表现形式。我发现好程序和好诗歌一样都有一些要素，这些两者要素有相当的共同点，如下所示：

第一要素：要有逻辑

程序的逻辑从大的方面来说是反映了程序的结构，结构清晰明了，才能引人入胜！例如，有诗的格律，如五言、七律等，诗歌只能按照格式填写，诗的格律就是诗的逻辑架构。程序的逻辑结构往往体现在时序上，例如：初始化处理、功能处理、结束处理。如此设计将使得程序更显逻辑。从小的方面来说是反映程序架构下的流程，流程次序井然、前接后续，才能使得程序执行顺畅。

第二要素：用词

程序的用词一方面是指各种变量的命名方式，另一方面是函数命名的方式。程序的用词应该用英语，命名变量和命名函数所用的英文单词无论是用缩写还是全写都要符合大众读者的习惯，避免用生词、偏词、自造词。我们看到古今中外的名诗大都是用通俗易懂的词汇进行艺术组合而形成的，很少出现非常用字。编程中要特别注意函数名中的主谓宾搭配，要让读者读起来更加上口。

第三要素：短小

编程如同编写文章，好的文章如歌如诗。其特点就是短小，一般的五言、七绝只有四行就能包含一个很深刻的主题。编写程序也是相同的道理，要尽量让程序的行数减少。一方面要考虑总的程序行数减少，另一方面要增加函数，使得函数里的语句行数减少。

第四要素：精炼

一首好诗能在几句话中就激发读者的情绪，鼓舞读者的斗志，展示大自然的美景，揭示各种人生哲理。其中主要原因是诗人能用精炼的词汇来表达诗的内容。对于编程来说，精炼就是多用函数，多用系统函数，让各种功能都浓缩在函数之中。

第五要素：一致

一首好诗往往能读到排比、对仗这样的修辞手法。说到底这都是诗歌

的一致性的要求。编写好的程序同样要注意一致性的要求，例如，我们尽可能采用相同的语句结构、相同的语句的缩进、相同的命名规则、相同的动词、相同长度的语句、相同计数变量、相同缓冲区变量、相同的处理函数、相同的处理流程等。这些相同使得程序各个层次和各个方面看上去非常规范和一致。

当我们能按写诗的要求去编写程序，能用诗朗诵的感觉去阅读程序时，我们的程序员将不再机械地编写那些毫无生命的代码，我们的程序员将充满激情谱写出一曲动人的乐章，感动着自己，也感动着读者。每一个代码都充满活力，是跳跃的艺术音符，无数个代码组合在一起就像一首流淌的诗，一段一段，一行一行，给人以诗的享受。我们可能永远不能写出传世之诗，但却能够要求自己编写如诗的程序，享受那种写诗的过程和情趣。不断地追求诗的境界将会让我们的程序更加精炼，编程工作更加美好。

3.16 如何计算程序员自身的价值

“价值”这个词，很正式，俗一点就是一个“钱”字。可以说这个字是当今社会的一个核心。无论经济活动还是社会活动，哪个能离开这个字呀！“人为财死，鸟为食亡”可能是人生的一种写照。

回到现实生活中，我们常常听到程序员抱怨自己的工资太低，抱怨自己的付出和自己的报酬严重背离，有的甚至不考虑自己的工作能力和工作贡献，以别人的工资或社会岗位工资来确定自己的报酬。同时我们也常常听到公司的老板抱怨软件公司不赚钱，最大的成本都花在员工的工资上了，自己在给员工打工。那么我们如何计算程序员自身的价值呢？我们怎么看待程序员价值的值与不值呢？我认为程序员自身价值有两个方面：一个方面是程序员的收入价值，另一方面是程序员的市场价值。程序员收入价值比较简单，就是程序员的年收入。例如一个程序员的年收入是5万，则这个程序员收入价值就是5万。程序员市场价值是指程序员编制的软件产品销售价值和预期的销售价值。例如，有5个程序员1年之中为1个客户编制了1个软件，软件售价5万元，假定忽略

销售成本和维护成本等，5 个程序员水平大致相同，则一个程序员的市场价值就是 1 万。

程序员的收入和程序员市场价值往往是不相同的：

1）当大学生刚刚参加工作的时候，由于能力限制，很少能编好程序，大部分时间都花在学习上，但是，他还是获得了工资报酬。但是他的市场价值为 0。这个时候，程序员是正收入。

2）当程序员具备编程能力，投入编程工作中，程序员不但获得工资报酬，其劳动成果也变成了公司的收入。假定公司能收支平衡，而且分配比较合理，那么程序员收入价值与程序员的市场价值相当。

3）当程序员具备较强的编程能力，投入编程工作中，程序员不但获得了工资报酬，其劳动成果也变成公司的收入。假定公司产生利润，而且分配比较合理，那么程序员的收入价值就会低于程序员市场价值，其差价就是程序员给公司的利润贡献。

4）当公司出现亏损的时候，程序员收入价值就会大于程序员的市场价值。通过以上分析我们可以看出，程序员的市场价值是程序员收入价值的基础，当程序员收入价值远远低于其市场价值的时候，程序员就会感到收入太少，分配不公，不愿意在公司长期呆下去。当程序员收入价值高于或等于其市场价值时候，公司老板只能维持或靠吃老本来维系公司的运作，若维持不下去公司只好倒闭。另外，程序员的市场价值要小于公司的软件销售收入。就目前的中国软件企业运行状况而言，中国软件业早已告别了暴利时代，残酷的竞争使得软件公司的利润趋于下降。有的软件公司在竞争中倒闭，大部分在维持中。究其原因很多，但是大部分软件公司都是靠开发项目来维持生计。而项目都是靠用户需求来驱动的，因此，开发出来的软件往往是定制的，不可复用。一些好的软件公司，往往在一个行业中积累了大量的行业知识和软件开发的程序。通过程序积累系统形成了行业系统的开发工具和通用软件。使得程序的复用性大大增强，逐步形成了行业软件的优势，这样的公司可以发展得很好。

无论是从程序员个人的收入价值或市场价值，还是从软件公司的运行状况来看，其核心是软件本身的市场价值。如果一个软件市场价值很高，则程序员的市场价值就会很高，公司的利润就会很高。一个软件市场价值往往取决于这个软件的复用性或商品化程度。假定一个软件销售价格为1万，预期的销售个数达1000个时，其市场价值就会达1000万，假定这个软件是由5个人开发的，相关销售、维护、管理费用不计，则一个程序员的市场价值就会达200万，尽管这个程序员年收入只有5万。但是程序员的市场价值是很高的。有了高的市场价值，就有了增加工资、奖金报酬的基础，就会有增加收入的希望。一般开明的公司都会根据程序员的市场价值来定义程序员报酬。只有这样公司才能留住人才，才能获得更大的利润。

另外一种情况，当一个程序员的市场价值达到一定数值的时候，如50万、100万的时候，而收入价值处在较低的水平，例如，5万、6万，程序员往往会产生单干的念头，以期获取更大的收入价值。但是程序员往往会忘记市场价值的实现并不仅仅在于软件本身，还涉及资金、资质、市场、销售、安装、维护、服务等各个环节。所以个人开软件公司的人数很多的，但是成功的并不多。人们通常说好的程序员往往不是好的老板，就是这个道理。

现实当中，大学生程序员年收入在2万元比比皆是，普通程序员年收入在4万~5万者居多，高级程序员在10万以上也不在少数，超过20万年收入的程序员也时有耳闻。如果我们取程序员的平均年收入，估计在5万左右。假定公司的利润率为50%，全部由程序员贡献，那么程序员的市场价值也就在7万~8万。这就意味着程序员一生（按35年计）市场价值也就280万。这同时也意味着中国程序员市场价值有很大的增长空间。

认识程序员自身价值的目的：

第一，希望程序员不要单看自己的收入价值，不要埋怨自己的收入低，而是要更多了解自己的市场价值到底有多高。

第二，市场价值就是编制满足用户需求的软件，如果这个软件销售不

出去，那么程序员再辛苦都白费，因为市场价值为0。程序员应该争做具有市场价值的软件，同时尽力帮助软件实现销售。

第三，市场价值最大化就是多编制可复用的程序，以提高单个软件开发效率，降低人力成本，提高利润率。

第四，市场价值最大化就是多编制商品化软件，在编制定制软件还是通用软件选择中，它可以成倍地提高程序员的市场价值。

第五，有了程序员市场价值的认识后，程序员和公司可以共同地为促进市场价值而作出贡献，并到达员工提高收入，公司提高利润的双赢局面。

中国的程序员市场价值相对较低，导致程序员收入不高，程序员技术水平降低，最终导致软件质量下降。我们要关注程序员市场价值的提高，技术水平的提高，使得软件行业中通用软件所占的比重大大增加，软件公司的收入大大增加，只有这样我们的程序员的收入才能大大增加，我们的软件才有更高的质量和更大的市场。

3.17 程序员成熟的标志

程序员在经历了若干年编程工作之后，很想知道自己的水平到底如何？自己是否已经成为成熟的程序员？虽然程序员会对自己有一个自我评价，但是，自己的评价和社会的评价、专业的评价会有差异，所以程序员自己并不能肯定这个评价。在现实中，除了各种证书之外，很少有人会专门给出一个程序员的成熟度的评价。人们往往是偶发性就事论事地对程序员的工作做出好与不好、行与不行的评论。因此，程序员对此感到很茫然，不知道要从那些方面去评价自己的能力。

一个程序员到底成熟不成熟，我想从以下几个方面谈谈自己的看法。

1. 技术标志

如果程序员不会编程序，那绝不是程序员，程序员至少要掌握一门程

序设计语言，要能够用这种语言编写程序去解决他想解决的问题。但是，成熟的程序员往往掌握不止一种程序语言，3～4 种语言的掌握是必需的，一种两种语言的精通也是必需的。

除了从掌握程序设计语言个数之外，我们还可以从其他几个方面去了解程序员在技术上的水平。例如，函数编写能力（命名、格式、大小、分类、参数、复用等）、面向过程的能力、面向对象的能力、数据库技术能力、效率处理能力、安全处理能力、网络处理能力、软件架构能力、人机交互能力、通用软件能力、软件文档能力等。尤其是面向对象技术的掌握和运用，以及面向服务的技术都是成熟程序员必须掌握的。

2. 时间标志

虽然程序员的天资、素质、基础知识各不相同，所经历的工作内容以及环境也不相同，但是，时间也是程序员成熟程度的标志之一。一般程序员需要经过 3～5 年的时间才能日趋成熟。其中入门需要一年，成长需要两年。这是我经过长期观察得到的平均数据。我并不认为成熟时间越短程序员就越聪明，就越了不起。享受每个阶段充分的时间，这会让自己成长更加充实、更加成熟。当然，也有超期却不成熟的情况，这也是很正常的。

3. 项目标志

程序员的社会性是程序员成熟的标志之一。没有参加过项目的程序员，程序编得再好，只能是纯程序类的程序员，是一个孤独的高手，是一种个人型的程序员，还远没有成熟。项目作为社会性活动体现了项目的社会价值，所以项目能力也是程序员成熟的重要标志之一。项目能力包括参加项目的个数、大小、在其中承担的角色等。就项目承担的角色而言，主持开发（项目经理）3 个以上项目是必需的，这是一个必要条件。一个程序员如果没有主持过开发，无论参加过多少项目的开发，无论是在程序编写或项目设计上发挥了多大的作用，都是很难被称之为成熟的，因为项目的组织、协调和管理是反映一个程序员成熟程度的又一个标志。就如同一个程序员能参与过 10 个以上大大小小的项目或能参加或能主持两个以上

大型项目的开发，其成熟程度是可以信赖的。若低于此数，则说明程序员离成熟还有相当的空间。“我们在项目中成长”，可见项目对程序员的意义是多么的巨大。

另外，一般程序员只是为一个企业客户进行开发一个或多个项目，或为同行业的企业开发项目，如果程序员开发过多个行业的项目，其成熟度要比一般人高一些。

4. 思维标志

幼稚和成熟在思维方式上具有很明显的区别。就程序员而言，不成熟的程序员逻辑性不强，程序编得没有条理，即使程序员自己进行了解释也没人能看懂。而成熟的程序员应该具有很强的逻辑性，程序编得井井有条，不用解释别人也能看得懂。这种逻辑性还体现在软件的架构设计、数据库设计、算法设计等多个方面。程序员通过全集子集概念、时间概念、顺序概念、重点和非重点概念等对各种事物进行逻辑分析。例如，以顺序概念为例，不成熟的程序员往往会采用自底向上的思维方式来开发程序。他们先考虑程序的具体实现，然后再考虑功能设计、最后考虑架构设计。而成熟的程序员则采用自顶向下的思维方式，先考虑架构设计，再考虑功能设计，最后才考虑编程的具体实现。前者思维方式主要是出于工作惯性，只适合入门阶段，而后者思维方式反映了后者的进步，适用于各种项目开发或大型项目的开发。

除了在思维内容上的逻辑性之外，程序员还应该处理好动脑和动手的关系。重视思维本身就是一种成熟的标志。成熟的程序员的思考时间要大于动手编程时间，想好之后只要一次就编程成功，而不成熟的程序员往往动手编程时间要远大于思考时间，而且是边做边想，通过反复来逼近最终目标。

另外，在思维范围上，成熟的程序员要比普通的程序员有更开放视野。他们更容易接受新的东西，更容易不受各种约束去考虑问题，更勇于去挑战自己和高手。

5. 与人交往

很多人认为程序员是和计算机打交道的行业，我认为这只是这个职业的特点。但是，只要是工作，就必然是一种社会劳动。而社会劳动则必须和人进行交流和沟通。尽管程序员的劳动工具是计算机，但并不意味着程序员只想着这个工具。从这个工具的下游来看，程序员还是要考虑用这个劳动工具生产出来的软件产品是否有人购买，是否有人使用，是否运行正常，从这个工具的上游来看，是谁让程序员了解设计方案的，是谁让程序员编程序的，是谁让程序员程序通过验收的等。因此程序员在软件制作各个环节都会与其他人打交道。只有和人进行有效的交流和沟通，我们的工作才能顺利做更好。

如果一个程序员还沉浸在个人劳动的意境中，对外界持冷漠、无奈、恐惧的心理，在内心里不愿意和外界打交道，无论自己感觉自己的技术水平有多高，那都还是一个不成熟的程序员。而成熟的程序员一定特别重视与人的交往，无论是上级领导、外部客户、项目经理、团队同伴这些与自身工作密切相关的人还是那些非同单位同行的朋友、网友等，他们都会认真听取别人的阐述、要求、意见、建议、反馈等，从而得到更多的工作上的、技术上的、生活上的好想法，以便自己参考和吸收。与此同时，与人交往也反映出你有好的想法和好的技术可以交流出去，而这些想法和技术水平也是成熟度的一种反映。那些没有想法和技术水平的程序员的确怕和别人交流。

与人交流时有两个基本能力要求：一个是理解能力，一个是表达能力，两者缺一不可。例如，有的程序员理解能力差，不能理解项目经理提出的要求，有的程序员表达能力差，无逻辑，无重点，啰里啰唆，让别人不知所云。这些都是不成熟的表现。

6. 别人的评价

别人的评价尤其是单位同事以及对自己工作情况比较了解的人对自己的评价是有参考价值的。一般而言，得到评价为差的程序员，其能力一定

是不行的，是不成熟的。评价好的要看情况而定，单位同事对人的评价会从两个方面考虑，一个是这个人的为人情况，另一个是这个人的工作能力。如果两者都不错，我们有理由认为这个程序员已经成熟。反之，无论是工作能力多么强，但为人不好，或者为人很好，工作能力不强，我看都不能算是一个成熟的程序员。

所以，程序员要注重别人对自己的评价，在提高自己技术水平的同时，学会做人，做好人，学会与他人分享，这样别人才会给自己更好的评价。

无视别人评价，其实这也是一种不成熟的表现。只有自己感觉好，大家感觉好，那才是真的好。

其实，别人的评价如果仅限于自己单位的话，恐怕这种评价的价值会打折扣，如果这个单位技术人员的人数很少，水平普遍低，即使你鹤立鸡群，大家对你的评价很好，但是，你和其他公司或单位的程序员相比，你真的不一定成熟。所以我说别人的评价仅仅是一个参考。

7. 收入标志

收入也是一个成熟程序员的参考标志。收入的多少往往是对程序员社会价值的认可度，表明程序员的劳动值这个价钱。一般而言，成熟的程序员能够挣得软件业平均收入的中上水平，或者在一个单位或部门中能够挣得比80%左右员工要高的收入，而刚参加工作不久的程序员其收入应该与其相差很大的。另外，从单位的项目奖金发放也可以看出程序员在项目中的地位和作用。

在现实中，我们知道程序员的收入与其付出是不成正比的，而且，越是能力强的、贡献大的程序员，可能不一定比那些其他能力不如他的程序员高出许多。这不是软件行业的通病，几乎所有行业都存在这种情况。通过分析我们认为程序员的成熟度应该是与其收入水平的高低挂钩的。如果我们的能力和贡献大大超出了收入，就有理由向上级领导提出提高自己收入的要求。

8. 心理素质

程序员常常面对各种各样的成功和失败，尤其是失败更是多于成功，这也是程序员这个职业的特点之一。以编程为例，几乎没有一个人一次就能将程序编好，他总是要遇到各种语法错误、各种遗漏，一个程序要反复多次修改调试才能完成。有的程序员因找不出来程序的 bug，束手无策，唉声叹气，心里极其不爽。以工作为例，有的程序员因工作进度和程序出错常常受到别人的批评和指责，心里极其不满，认为批评人不了解造成这个结果的客观原因，批评错了人，从而对人产生意见，甚至对工作造成了影响。面对失败和挫折，成熟的程序员会坦然面对：编程时出现问题不可怕，有什么问题就解决问题，解决不了的问题可以想其他方法进行解决，不在一棵树上吊死。面对别人的批评和指责，首先从自身查问题，是自己的问题，那就要主动承担责任，并尽快改正。不是自己的问题，应该换位思考，理解批评人的焦急心态，并找机会说明。拥有良好的心理素质的程序员在面对困难和挫折时，就会很坦然、很坚强、很自信。

程序员也会面对成功的。有些程序员因开发了某个项目，因编写了某个程序而感觉良好，在不自觉中表现出我最牛，我最好的样子，面对他人夸夸其谈，而对其他人的成就不屑一顾。更有甚者并无成果，表现平平，却依然会摆出一个高手的样子，有的仅仅参与了某个项目，而且不是项目主要开发者，却会贪天之功，归其所有，好像这个项目是他主持开发的。这些其实也是心理素质不成熟的一种表现。成熟的程序员面对成功并不会感觉高人一等，该是自己的功劳就是自己的功劳，该是别人的功劳就是别人的功劳，即使自己比别人水平高出许多，他想的是，还有更高的技术顶峰等待攀登，不可自傲，看到别人取得的成绩首先应该去祝贺，然后去学习，而不是心怀嫉妒，从中挑刺。

良好的心理素质使得程序员更加理性地处理好各种成功和失败带来的问题，更有利于程序员超越自我，以平常之心去迎接更大的挑战。

当然，一个程序员是否成熟是一个仁者见仁智者见智的话题。有的

人强调程序员的个人能力方面，有的人强调程序员的社会能力方面。我认为从以上8个方面综合地去评判一个程序员是否成熟应该能说明些问题的。我们标志成熟的一个目的是对程序员在成长过程给予一个肯定和鼓励，让程序员认清自己所处的阶段，为充满自信找出依据；另一个目的是对程序员未来的成长提出更高的要求。走向优秀是成熟程序员面临的更大挑战。

第四部分
Part 4

优 秀 篇

4.1 成熟到优秀的瓶颈问题

很多程序员到了成熟阶段之后就会处于一个停滞状态。技术上已经驾轻就熟，工作上已经按部就班，心里虽然感觉有些不甘，但是何去何从他们显得很茫然、很无助。我认为程序员到了成熟阶段后，如果还想要向优秀阶段发展，一定会遇到这个瓶颈，而穿过这个瓶颈就会走进另一片开阔的前景，穿不过则会停留在原地止步不前。

1. 技术瓶颈

技术上的瓶颈很明显，主要表现在：对学习缺乏热情，对技术缺乏钻研，对新技术发展缺乏了解等三个主要方面。其原因主要是：第一，成熟的程序员编程技术已经能够满足开发软件的要求，很多人认为只要能编出来就行了，至于编得更好，那就要看兴趣了，看时间允许了。第二，由于成熟程序员承担着比一般程序员更多的工作，比如软件设计和项目管理，比如与用户和领导打交道，很多时间和精力无法放在编程上面。第三，由于软件开发任务一般时间都比较紧，必须加班加点才能完成，因此，没有时间再做技术上的深入。第四，很多程序员处于一个自发的发展状态，自己的成长完全取决于自己的工作内容，工作内容强度和复杂程度决定了其技术水平的高低，因此，他自己根本不知道自己技术发展的方向是什么，技术上的差距是什么，也就无从谈起自己努力的方向。

因此，成熟的程序员应该有更上一层楼的意识，千万不要故步自封，夜郎自大。首先，要对自己在技术上有一个客观的分析，看看自己的长项在什么地方，弱项在什么地方。对自己的弱项就要想办法进行克服，千万不要留在那里，日后必然会影响自己的进步。比如，很多人弱项是面向对象的设计方法、人机界面交互、大数据量处理、处理效率等，如果有这些方面的问题就要想尽办法解决。其次，要注重编程基础技能的深入掌握，很多时候程序员只是拿来主义，知其然不知所以然，应该把基础缺失的那个部分补回来，为今后走向更高的技术境界打下基础，比如操作系统、网络知识、程序设计语言、数据库、系统架构、软件工程等基础知识，要重新审视，把其中有用的知识掌握好，并且要学会把各种基础知识串联起来，加深对基础知识的认识。最后，要逐步把编程技术从功能实现方面转向参数化设计、软件架构等高级阶段方面的研究，在共享、通用、标准化方面有所建树。

2. 工作瓶颈

程序员在工作上也存在向上的瓶颈。就一般情况而言，很多程序员在这个阶段工作上比较稳定：领导对自己比较了解和信任，同事对自己的工作能力和技术水平也比较认可，软件用户对自己的服务也比较满意。因此，在充满工作满足感的同时，工作日复一日，没有什么新意，疲倦感油然而生，而以往那种激情很少有地方释放了。所以程序员有时也会思考自己在工作上究竟要在什么地方进行突破，究竟怎样才能更上一层楼。

程序员在工作上的现状还会受到软件公司或上级部门安排的影响，尽管自己在具体工作上付出很多，编程上有很大的自主权，但是从总体上来看，程序员只是企业经营过程中的执行人员，是一个被动的角色。因此，程序员要改变工作上的现状，必须要有主动意识，尽可能主动地规划好自己的发展方向，主动地向上级反映自己的想法和打算，争取上级的理解和支持，在工作条件许可的情况下，把自己的时间和精力以及重点放在自己关心的方面。例如，程序员感觉自己编程方面已经满足工作要求了，但是自己与客户打交道的能力和市场资源的积累很差，他就可以主动向上级提

出自己想做售前工作，如果领导同意，则可以在这个岗位上提高自己这方面的能力。在岗位不调整的情况下，可以根据自己的工作范围，尽可能多与客户打交道，了解客户的需求，从而为自己开发的项目做好基础，同时通过与客户打交道和与客户交朋友，为自己积累市场资源。其实在工作层面上可以有很多值得改进的地方。

3. 收入瓶颈

说到底，程序员最大的瓶颈在于收入上的不均，虽然经过多年的努力奋斗，收入也有了一定提高，有的甚至达到了社会平均收入的中上水平，但是，面对中国的生活现实，买一套房需要花光几十年收入（即使按揭也还是要每月偿付本息的），而且工作很难稳定到几十年不变。因此，绝大部分程序员的收入是很难满足那种程序员向往的生活，尤其是过上中等生活的要求和过上稳定生活的要求。当然，例外总是有的，极少的程序员收入也是相当高的，生活过得很滋润。就普遍情况来看，我们付出的和得到的是不成比例的，这不能不说是社会的一种悲哀。从某个程序员的具体情况来看，程序员的收入一般比较稳定，有的一年动一次，有的几年动一次，这要看所在公司情况和所在单位部门情况而定。程序员和其他职业的员工一样很少在工资收入方面有自己的话语权。

程序员经历了成长过程的风风雨雨之后到了成熟阶段，收入应该比过去高了许多。但是程序员的预期和实际收入的反差是程序员内心最大的烦恼，因此，增加收入或大幅度增加收入是程序员无法突破的瓶颈。

面对收入瓶颈，程序员应该调整心态，光靠埋怨绝对无用。要解决这个问题，程序员可以从以下几个方面来考虑。第一，程序员可以在本公司内进行收入增加的挖潜，可以主动提出调整到收入高的岗位，可以参加高奖金的项目开发，凡是能够增加收入的各种工作都可以考虑去做。第二，在这个基础上，如果程序员感到自己的收入还是无法达到自己的预期，不妨可以考虑离开这个公司或部门，通过应聘高薪工作的方式来提高自己的收入，当然这种选择还是要慎重的，要考虑多方面的影响，很多情况下收入不是惟一的因素，未来收入和现有收入很小的差距更是忽略不计的因

素。第三，程序员在条件和精力许可的情况下，可以从事一些第二职业，由于程序员这种职业特性，程序员可以独立完成某个编程任务，也可以和其他人共同完成编程任务，通过从事第二职业，程序员可以增强市场意识，可以比本公司、本部门其他同事知道更多的软件价值，结识更多的软件同行，学习更多编程技术，并且获得相应的劳动报酬。更重要的是在此期间可能会遇到自己发展的机遇。

满足现状的程序员就不可能遇到这些瓶颈，他们会在一个自己的世界中得到满足，他们会在这种满足下持续自己的工作，求得一份平静的生活，而那些不满足现状的成熟程序员，那些追求优秀的程序员，因追求而困惑，因瓶颈而求突破。程序员只要不甘于现状，只要认定一切要靠自己，只要准备付出比过去多的精力，只要准备克服比过去多的困难，只要求新求变，最终都会破茧成蝶的。在那绚丽的天空下，那自由飘飞的彩蝶正是他们未来的身影。

4.2 梦想回归

很多人在进入 IT 这个行业时几乎都有自己的梦想，都有自己的偶像，都期望有一天能够像偶像一样实现自己的梦想。这个梦想可能是成为百万千万富翁、软件高手、软件公司的老板、这个行业的名人专家等。当真正踏进程序员这个岗位后，才会慢慢发现，自己离自己的梦想越来越遥远。久而久之梦想淡去，换来的只是平常之心，到最后，只要有工作有收入程序员心里就满足了。

但是，我们也发现这个行业不乏成功的企业以及成功的人士，有的可能是听说，有的还是身边熟悉的同行呢。在很多情况下，当听到这些信息时会暗自使自己沉沦下的梦想又搅动起来。不过时间一长，梦想又会很快消失了。

可以说程序员从入门、成长到成熟的各个阶段都是对自己梦想的一种现实考验。也就是说，这段时间会让程序员逐步认识到这个行业、这个企业和自己。看看这个行业是如何发展壮大的，看看企业是如何生存和发展

的，看看自己的能力到底有多大。结果是，程序员看到的是行业不断地爆炸性发展，自己的企业却在生存中挣扎，自己不停劳动的付出却看不到理想中的结果。这种行业的发展和自己处境反差加深了程序员对现实的失望，于是梦想丧失。

在这个世界上成功的人是少数，这是一个规律。这个规律昭示着成功的人是与众不同的。多数人在成功的道路上被挤掉队，留在成功道路上的人毕竟是少数，而且成功道路的前途更加曲折艰难，能走向尽头的少之又少。大浪淘沙，去芜存菁。当你能感觉到自己与众不同，当你能感觉到自己的能力超越你周围的同行，当你经历过很多很多的磨难还依然怀有过去的梦想时，你可能会成为一名优秀的程序员，而能留在优秀程序员队伍的程序员是很少的，并且只有这些人才有可能实现自己的梦想。

到了优秀阶段，程序员要比任何时候都接近自己的梦想，这个时候的梦想已经与入门时候的梦想不同了，那时的梦想主要是“梦”，还处于浮云天上飞的状态，而这个时候的梦想则是在于“想”，梦想已经回归到大地上，这个梦想已经和现实十分接近了，已经不是太遥远了，程序员主要在“想”如何实现这个梦想了。我们说优秀程序员可以实现自己的梦想的原因在于：

1. 积累了足够多的编程经验和编程技术

软件业核心就是软件，如果能够掌握编程技术，积累了足够多编程经验，以较高的水平编制各种软件，这就为程序员成功打下了良好的业务基础。有了这个基础就使制作软件产品得到有力保障。程序员以这个基础展示自己的业务才华，去编制有价值的软件。由于熟知编程的过程，程序员就可能成为软件业的项目管理者或是高一级的部门管理者，他的经验和能力可以确保软件制作按期完成。如果一个程序员连软件制作这一关都没有通过，他能成功只能靠想象了。但是，那些有商业头脑的人在 IT 浸过几天水之后，最后竟能成功，对此我们并不感到吃惊，我们认为这不是属于程序员的成功范畴，这只是商业头脑在 IT 成功的事例。

2. 目睹了软件业成功和失败的案例

走到优秀程序员这个阶段，程序员一定会看到自身企业或周边软件企业发展的成功和失败的案例，从中看到影响企业成败的各种因素，如：员工编程能力、项目管理、运营成本、软件价格、产品市场容量、用户服务、产品营销等。这些因素足以让优秀程序员思考怎么才能让这些因素由消极变成积极。通过思考，程序员就可能在主观上形成企业成败的判断标准，从而在自己的工作中选择最优的方式进行工作，克服了前人留下的弊端，让自己的发展少走弯路。

3. 找到了程序员成功的道路

程序员发现程序员成功之路并不只有当老板这一条道，即使当上老板也并不一定能成功。建立一个企业容易，让一个企业发展就不容易了，让一个企业持续发展那就更不容易了。程序员可以选择当老板，也可以当高级打工者，也可以成为专攻无人可及的技术高人。这些都是成功可选的途径，关键是程序员如何掂量自己，看看自己当前适合走什么道路，有老板之才可以当老板，无老板之才可以当高参。千万不能自不量力，否则当上了也要下来的。

4. 学会了设立分阶段目标并实现

程序员在编程序时，尤其是编写大型软件时，都是分步实施的。可以说一步一个脚印，最终完成整个程序。程序员这种职业习惯也会影响到程序员成功的方法，优秀的程序员一定会把自己的最终梦想采用自上而下的方法进行分解，把终极目标分解成一个一个有序的小目标，并努力实现。这样程序员的梦想就可能成为现实。比如，程序员的梦想是当老板，那他就要为此设立分阶段的目标：寻找潜在的客户、选择用户需要的软件、评价软件的市场价值、测算软件企业的经营成本和利润、招聘软件开发人员、进行项目管理、完成软件开发、软件的产品化、软件的销售、软件维护等。当他自己确定每个阶段目标的可行性之后，并能确保自己能够完成这样的目标后，就会一步一步地实现。

5. 梦想更加实际

每个人的梦想不会永远不变的，否则这个人就没有梦想。人的梦想是随着环境、条件甚至自己的心情来调整的。一般的程序员的梦想无法实现，其梦想不再。而优秀程序员的梦想则是通过程序员自己对现实的认知来进行调整的。比如，原来的梦想可能是成为亿万富翁，而面对现实就可能先调整成十万、百万的年收入，然后再做出更高收入的目标。原来想当老板，而面对自己只会编程没有管理能力的现实，那就先当高级程序员吧，等能够当项目经理、能够管理好一个部门之后，再去当老板也不迟。原来想当一名编程高手，而面对自己编程能力无法和高手过招，而自己擅长和用户沟通的现实，那就可以做客户经理或软件产品的售前，同样让自己的能力闪光。

6. 坚持梦想

梦想因为遥远才需要我们努力伸手去触及，这就要求我们坚持，只有这样才能到达梦想的彼岸。没有梦想的人不会经历那种困难和艰辛，因此，也不会坚持，他们无法享受那种成功的喜悦，他们永远是成功者的旁观者，他们永远也无法体会成功者与梦想相拥时刹那间那种喜极而泣的感觉。优秀的程序员在走进编程的那一刻起就养成了不编出程序不罢休的习惯，因此，面对自己的梦想，我想所有优秀程序员都是不会放弃的。梦想的道路有多苦，只有程序员自己心里清楚。从某种意义上来说，目标定得越高，必然困难就越大、可行性就越低。所以，在实现梦想的道路上我们一定会遇到各种各样的困难，一定会挑战我们的承受极限，我们只有坚持，梦想才有可能实现，没有坚持，梦想一定不能实现。

“谋事在人，成事在天”说的是人可以按自己的意愿做事，但是是否成功在于天命。这种话宿命味道十足。我想说的是，天道酬勤，由于程序员在成长的道路上会遇到各种各样的机遇和挑战，任何机遇和挑战都能使自己成功。所以，只要坚持自己的梦想，只要不断地提高自己的编程和与人与市场打交道的能力，找出适合自己的发展方向，积累发展的资本和资源，抓住机遇，程序员一定能够实现自己的梦想，即使在金钱或其他收入

上没有达到自己的预期，但是在实现梦想的过程中所得到的各种收获绝对是其一生取之不尽的财富，当程序员年迈回顾往事的时候，一定会为自己的经历而感到骄傲，一定会为自己没有虚度自己的年华而感到无悔。

4.3 激情！激情！激情

很少有像软件这样充满激情的行业，这个行业阳光灿烂，奋发向前，创造了许多像微软、Apple、IBM 等软件帝国，诞生了像比尔·盖茨那样的杰出人物，研发了许多像 Windows、Java 等各种软件产品，产生了许多像 C 语言、面向对象等各种软件技术，而且新技术不断涌现。这是一个智慧型的行业，人的智慧在这里得到了最充分的展现。一代又一代的 IT 人充满激情去挑战这个行业的制高点，在那里展示自己的智慧，获得比其他行业高得多的收益，从而实现自己的价值。很难想象，一个平庸且循规蹈矩的人，没有那种燃烧的激情去激发自己闪亮的智慧，怎么会产出优秀的软件，怎样管理好优秀的企业，怎么会变成一个优秀的人才呢？

当我们还没有进入这个行业的时候，我们就曾被这个行业的神话所鼓舞，当我们进入这个行业之后，我们心潮澎湃、充满朝气、激情四溢，面对着各种不断出现的技术困难，我们毫不惧怕，加班加点，一个又一个难点被我们克服了，我们编出来了一个个程序，这些初期的成功反过来刺激着我们的激情迸发，这几乎是每个程序员成长时经历的最初的激情。然而，激情并不能长久，当我们发现编程不仅仅是一个技术问题之后，我们还会发现我们的收入不够高，我们的技术水平提高并不快，我们的工作环境并不尽如人意，我们的激情开始递减了，我们的未来开始模糊了，于是我们开始怀疑我们的激情了，感觉充满激情是一件很傻的事情，反而趋于平常心了。加班加点由主动变成了被动，理想变成了现实，激情变得麻木。当然，偶尔的时候，我们仍然会有激情，但是，激情远不如以前那样热烈，无论是失败还是成功都没有那种最初发自内心的触动。

面对激情的话题，我想说：

1. 激情是程序员的职业特点

程序员职业充满激情是其职业的特殊性所决定的。由于打交道的对象是计算机，程序员又可以通过程序操纵计算机，而程序的编写并非那么简单，需要程序员绞尽脑汁想办法实现设计功能，在这个过程中充满挑战、智慧、坎坷，一旦跨过这道难关，心情想不激动都是不行的。这种激动就是那种控制欲望的满足，就是激情最终的释放。最终的目标绝大部分都是能够实现的，只是实现时间的长短、实现方法不同而已。因此，几乎所有程序员在初期都能够充满激情地编出程序。激情如火，燃烧着程序员的心，他们为了实现自己的目标，加班加点，废寝忘食，专注而不顾其他，让外人看来程序员光芒四射，不可思议，而程序员一旦完成了自己的程序，就会兴奋无比，不可言喻。

2. 激情大小取决于压力

激情有大有小，很多情况下它取决于程序员所承受的压力。一般而言，时间压力是最大的压力，很多项目开发时间被无限制地缩短在一个看上去不可能完成的时间内（用户总是希望越早越好），导致了程序员身负重担，不得不通过加班加点增加劳动时间来完成开发任务。除了时间紧之外，编写程序也是困难重重，特别是新手或是不太熟悉语言的程序员更是感到头痛不已。有时候程序编好了，但是运行起来有 bug，单就是找 bug 也不是一件轻而易举的事，有的时候什么方法都想尽了，但还是查不出问题所在，这给程序员造成了很大的压力，甚至程序员容易产生焦虑情绪。所有这些压力都是产生激情的直接动力，只有产生激情才能克服这些压力，才能完成编程任务。我们常常看到压力越大，我们的激情就越大，我们成功后的喜悦就越大。

3. 激情燃烧自己照亮别人

大多数情况下程序员是不自觉地就会充满激情去做自己想做的事，直到完成为止，其中之感慨只有自己才会有深刻的体会。一些年纪大的程序员在回忆自己当初那种编程的劲头时，那种疯狂程度连自己都感到不可思

议。但是，很多程序员就是看到其他程序员的激情工作而被感染的。尤其在一个团队，当其他程序员奋发工作的时候，自己是很难将激情置之度外的。这说明了这样一个道理，你的激情可能感染其他人，其他人的激情可能也会感染你。知道这一点之后，团队领导更要以身作则，身先士卒，一定会鼓励更多的同伴一起完成彼此共同的任务。

4. 激情不仅仅表现在编程

很多人认为程序员的激情仅仅表现在编写程序中的加班加点上，表现在编程的特别专注上，我认为这种想法是不全面的，不同程序员由于其在项目中的分工不同，其技术水平不同，性格不同，激情的表达方式也不尽相同。而且，由于软件行业是涉及面很广的行业，水平越高的程序员的激情可能越是表现在编程以外的项目设计、项目管理、产品设计、客户关系、新技术学习、同行交流等方面。

5. 激情也需要坚持

我们常说“坚持就是胜利”，但是，现实中能够坚持的人绝对是少数，很多人在面对各种困难和挫折之后，不再坚持，于是失败是必然的结果。就程序员而言，就激情而言，谁会每天保持着这种激情去从事程序员工作呢？再热烈的激情总有一天也会有消失的时候。从入门阶段到成长阶段，从成长阶段到成熟阶段，大多数程序员的激情由高变低，有的甚至连一点激情都没有了。但是，要想成为一名优秀的程序员，那就必须学会坚持，学会坚持自己的激情，无论遇到任何困难，无论自己能找出任何理由，都不能丢弃自己的激情。这是因为优秀程序员遇到的挑战要比别人更高，遇到的困难要比别人更多，遇到解决问题的难度要比别人更大，没有激情，你根本无法激发自己内在的潜能，发挥超常的智慧，根本不能获得比别人更多的成就。**坚持的过程是痛苦的、难忍的，但是坚持的结果是快乐的、值得的。**

从某种意义上来说，激情就是一种生活方式和工作态度，我们呼唤激情是我们情极所致，我们呼唤激情是我们不甘平庸，我们呼唤激情是我们敢于挑战，我们呼唤激情也是我们呼唤着美好的未来。**激情是程序员之**

本，是优秀程序员之命。

4.4 摆脱技术束缚，拓展业务视野

常言道：“磨刀不误砍柴工。”很多人都把磨刀看做是重中之重，生怕自己的刀不快，砍不下柴来。其实，这句话也可以从另一个方面来理解，那就是世上的柴很多，要砍得多，砍得快就要磨好刀。

在实践中，程序员面前的“柴”就是用户的业务需求。程序员手中的“刀”就是他掌握的编程技术。大部分程序员都是喜欢磨刀的，可以说是从入门开始就不停地磨刀，即使刀已经磨得很锋利了，仍然不知道自己的刀的厉害，还在继续磨刀。他们挥刀砍下的柴是别人放在自己面前的柴，不管这个柴是软是硬，是大是小，都会拼尽全力去砍。很少有人会去选择自己能砍得动的柴去砍，也不知道有哪些柴需要砍。

程序员是一个容易被技术束缚的行业，很多程序员在主观上就存在技术第一，以技术谋生存的思想，仿佛一旦离开了编程技术他就不是程序员一样。但是，程序员又是和用户的业务有着紧密联系的职业，程序员开发的程序都是用户的业务需求的实现，因此，又离不开对用户的业务需求的深入了解。程序员在编程技术和用户需求方面其实有两个方面的发展趋势：一个是编程技术学习和编程时间递减，另一个是用户需求学习和分析时间递增。通俗地说，程序员在开始阶段的时间主要花在编程技术的学习和编程上，随着掌握编程技术和水平提高，用在这方面的时间会越来越少。在用户需求方面，程序员在开始阶段的时间很少花在用户需求上，用户需求主要是靠同伴要求或客户的解释来理解，随着时间推移，程序员越来越要独立地面对用户的需求，需要自己进行用户需求分析。很多有经验的程序员都有这种体会，如果用户需求分析不透，程序编起来就会非常吃力而且容易反复，开发周期不断延长。

在编程技术和客户业务两个方面我们要注意以下几个方面的问题：

1. 克服技术第一

编程技术和客户业务哪个更重要？这要看程序员所处的阶段，在程序

员入门阶段绝对是编程技术最重要，程序员而且也很少接触到客户需求，客户需求都被项目经理分解成详细设计了。而到了成熟阶段和优秀阶段，客户业务绝对比编程技术重要，程序员会面对大量的客户需求，只有对客户业务的深入理解才能编出好的程序来。很多程序员即使在成熟阶段也从主观上排斥对业务需求的了解，认为自己的职责是编程序，其他都与自己无关。最好有人帮自己把用户需求变成具体的设计方案，自己只面对方案编程就行了。但是，实际情况往往并非如此，很多程序员必须面对用户需求，若不懂得需求，程序编起来一定有问题，因此，面对用户需求时躲都躲不掉。

2. 反对业务无用

对于一些程序员而言，他们认为学习客户业务是没有价值的工作，今天学习了这个客户业务，明天又会学习其他客户业务，因此，花时间和精力去学习客户业务，只用一次，使用机会不多，有这个时间还不如学习好编程技术。这种观念是错误的，主要是因为客户业务的学习和你的编程是密切相关的，不掌握好业务知识，程序就会编得很差，很差的程序对自己的技术也是没有提高的。另外，任何知识尤其是和开发相关的业务知识都是有价值的，且不谈掌握业务知识后对编程有立竿见影的效果，就是对在学其他用户业务时都是很有帮助作用的。由于各种用户业务知识尤其是行业内的用户业务知识都具有相关性，都有其内在的规律，因此，在弄懂一个业务知识之后再去学习另一个业务知识就容易多了，而且对不同业务知识的学习会加深对业务知识的深入了解。

3. 克服畏难情绪

程序员除了认为用户需求不是属于自己专业的想法之外，很多程序员认为用户业务很难学，很难搞懂。如果去学习用户需求，自己往往充当着学习者的角色，而自己原有的 IT 高手的形象就会受到影响，这会让自己丢面子。对新领域学习的畏难情绪在其他任何行业和岗位都有。但是，作为 IT 人员，学习其他知识应该是自己的天职，因为 IT 就是为其他行业信息化服务的，是边缘和综合的一个学科。加之，程序员要比其他人更具有

不怕困难和挑战的勇气，许多技术问题，不是仅仅看看书，编编程就能解决的。所以，程序员不应该有惧怕学习用户业务知识的恐惧。在长期工作实践中，我认为用户业务知识远比计算机相关技术要简单得多，关键是掌握学习用户业务知识的方法。

4. 认真学习业务

用户的业务知识往往是通过需求书以及与用户进行交流而获取的，实践中很少能找到完美的业务需求书，因此，寻找需求书之外的相关资料以及和用户面对面交流就非常重要，而且要非常认真，不要错过每一个细节。不少程序员学习业务不太认真，并且往往缺乏和用户打交道的经验和方法，开始的时候，习惯于自己看需求书去理解和学习，而不去寻找需求书以外的相关背景资料和文件，于是编程序的时候就会发现这样问题或那样的问题，这时就会希望和用户直接沟通。这样做就会增加和用户交流的次数，增加了用户的负担（用户因自身工作忙碌，无法及时满足你要交流的要求）。因此，程序员最好在编程之前，就要和用户进行深入地需求分析，把所有的需求分析完毕，在编程之中出现的问题应该积累起来向用户一次性地讨教。

5. 成为业务行家

程序员在用户业务知识和水平方面应该树立做一名业务行家的目标。我们看到那些高手和优秀的程序员除了编程技术高超之外都是某个行业或某个业务的行家。在用户眼里，这些人既懂得业务，又懂得编程，很了不起，在同行眼里，这些人既懂得编程，又懂得业务，也很了不起。在一些大型企业或公司里的 IT 人员中就会出现这些业务行家，他们在用户和软件公司之间搭建了业务和技术的桥梁。反之，仅仅满足于项目的开发，满足用户需求，是不能成为业务行家的。因为绝大部分用户需求有其部门局限性、业务的特殊性、功能的直接性和不完整性。只有对用户部门甚至整个公司的职能、业务流程、经营方式、管理制度、外部关系进行深入了解和研究，由大至小，由上至下，由宏观到微观，由过去到未来地研究这个业务的来龙去脉，并且经受住用户对其业务能力的考验，我们的程序员才

能成为业务行家。

现实中，无论是软件公司还是大型公司的软件部门最受欢迎的是那些既有编程技术又有行业背景的程序员，有的更把行业背景放在编程技术之前，这说明懂得用户业务知识是多么的重要。我们不要惧怕我们不能成为业务行家，很多人都说，从事计算机行业的人到业务部门一定能够胜任，而且比业务出身的人做得更好，反之则不然。这说明程序员是能够成为业务行家的，而且比业务人员更能成为业务行家。程序员一旦摆脱了技术束缚，就可以看技术以外的新天地，这个天地要比技术天地更广阔，更显无穷无尽。没有成为业务行家的时候，程序员只能按用户需求开发软件，成为业务行家之后，程序员可以指导用户完善用户需求，甚至可以自主开发软件销售给用户，由被动变主动。

4.5 预测趋势，让你的目光看得更远

我想每个程序员当面对 1、2、3 后面填写什么的时候，都会选择 4。1、2、3 让我们看到了一种规律，而 4 就成为这个规律后的一个结果。从时间维度来看，1、2、3 可以看做过去和现在发生的事，而 4 则可以看做将来所发生的事，因此，选择 4 是对前面事实的一种预测趋势，而且 4 是一种正确的选择，如果选择 5 或 6 则是错误的选择。我们可以说趋势是一种事物的规律在未来时间上的表现。而预测则是在掌握规律的基础上对未来规律结果的一个推断。

从哲学的角度看，万事皆有规律，万事皆有趋势。例如，人生趋势：从小到大，从大到老，从老到死；社会趋势：从野蛮到文明，从贫穷到富裕。有的规律被人们所认知和掌握，人们就会利用这些规律做合乎规律的事情。有的规律还没有被人们所认识，人们还在探索之中，做起事来可能成功也可能失败。而那些明知规律，却不按规律办事，必然会导致失败。

现实中有的人只有在谈及某个趋势的时候，人们才会给出肯定、否定或模棱两可的回答。在此之外，趋势跟他们无关。他们不想了解事物发展规律，不想预测事物的发展趋势，甚至连与自己工作和自己未来密切相关的规律都不想了解。他们把太多的精力放在眼前，他们认为未来多变化，

未来不可知。用他们的话说，了解趋势如何？不了解又如何？我们很难判断他们所说的对与否。因为不同的人因其能力不同，目标不同，对事物的看法和对自己的要求也不同。

但是，作为一名优秀的程序员，尤其是一名想成为优秀的程序员，则要养成预测趋势的意识。这不仅是普通优秀人才素质的要求，也是软件业这个职业的要求。软件业相比其他行业，新兴而富有变化，而这种变化之面广之迅速是其他很多行业不具有的。因此，要在软件业中有出色表现，就要了解这些变化，去适应这种变化，否则就很难在这个行业有大的作为，有的甚至被这个行业的剧烈变化所淘汰。

就程序员来说，他们可以关注周围很多事物的规律，并预测事物的未来趋势。这些趋势的预测对他们自身成长来说都是十分必要的。例如：

1. 预测自己成长状况

如果了解程序员的成长具有一般的规律性：从入门新手到成长阶段，从成长阶段到成熟阶段，从成熟阶段到优秀阶段。每个阶段都有每个阶段的特点，每个阶段都是前一个阶段的提升，每个阶段都不可跨越，每个阶段都有其时间要求。那么程序员就可以根据自己现在的状况来预测自己未来成长的过程。例如，现在这个程序员刚刚跨过入门阶段，处于成长阶段的初期，那么程序员就会根据程序员成长规律，大概预测一下自己的成长轨迹：2～3 年后自己将会步入成熟阶段，其后 3～5 年将会步入优秀阶段。也就是说这个程序员 5～8 年后才会步入优秀阶段。了解这个规律后，程序员就会更加严格要求自己，在每个成长阶段把对程序员的要求做好，绝不会此时因自己编制了一些程序而感到过于自信和自满，因为程序员的成功道路非常漫长，现在还远不是拍手称庆的时候。

很多程序员成长处于自发状态，其成长完全随着时间的推移和工作内容的推移进行着。很多人认为程序员水平和工作年限成正比，工作年限越长水平越高。这其实是一个误区，这个水平只能和程序员自己相比得出，如果和其他程序员相比结果就可能不正确。有的程序员可能花了 10 年时间还处于一个成长阶段，有的程序员永远也达不到优秀阶段，这就是我们

所能看到的客观现实。这很大程度上是由于程序员自己不了解程序员成长的规律，不知道如何要求自己造成的。

2. 预测自己的发展空间

程序员发展空间也是有规律可循的：程序员可以从新手开始，然后成为业务骨干，成为开发小组的负责人，成为软件设计师，成为项目经理，成为软件营销人员，成为企业的行政管理者或高管，甚至成为企业的股东。程序员也可以跳槽到其他企业，程序员也可以转行做其他工作。这些变化都有相应的外在和内在条件和环境因素。程序员了解到了这个发展空间后，就会根据自己的情况为自己设计未来的发展空间。例如，程序员准备花 2 ~ 3 年成为软件设计师，花 2 ~ 3 年成为项目经理，再花 2 ~ 3 年成为部门经理。这样的程序员目标明确，一步一步发展起来比较可行，程序员的目标就容易实现。

很多程序员并没有自己的目标，一心只想把程序编好，把自己的发展空间交给别人，希望有“伯乐”来识自己这匹“千里马”。一旦没有“伯乐”，就感到自己不是“千里马”了，就会抱怨世上没有“伯乐”让自己这匹“千里马”驰骋千里。

3. 预测未来语言发展

程序员最关心的是自己使用的语言是否会过时，以及未来会产生什么新的语言。通过程序设计语言的发展历史，我们可以看到：语言是从低级语言向高级语言发展的；一种语言是有一个产生、发展、消亡的过程的；低级语言发展到一定阶段相对稳定，而高级语言则发展迅速而且丰富多彩。高级语言趋同倾向明显，高级语言越来越容易掌握，功能越来越强大，并且有向自然语言靠拢的趋势。如果我们深入了解这些程序语言的发展规律，就可以预测程序语言的发展趋势。比如，某语言是一种新语言，我们就可以根据“一种语言都有一个产生、发展、消亡的过程”规律预测这个语言会有一个很好的发展，并有一个很大的应用前景（在这个语言设计应用范围内）。又比如，根据“高级语言趋同倾向”规律，程序员掌握好一二种语言就行了，不必个个都要学习。再比如，根据“高级语言越来

越容易掌握，功能越来越强大”规律，我们没必要恐惧未来语言不容易掌握。

4. 预测何种软件发展

不同种类的软件发展前景是不一样的，一般来说，单机软件将被网络软件所取代；企业内网软件将被互联网软件所取代；手机软件、电视软件、物流软件、管理类软件都会比现在有很大的发展。了解这些软件发展规律，我们可以根据这类软件现状预测未来几年内其发展趋势，从而根据自己的状况调整自己，以适应未来这款软件的发展。比如，我们预测手机类软件和管理类软件将会有很大的发展，我们可以在手机管理类软件抢先投入我们的精力，开发软件让企业各级管理者在手机上就可以看到各种企业经营类的报表和数据。而不是像现在只能在单位的计算机上才能看到这些管理信息。

5. 预测软件制作的发展方向

软件制作从个人方式向集体方式转变，从非专业到专业转变，从不规范到规范转变，从集中开发到分散开发转变，从完全开发到合作开发转变等。这些规律让我们对现在的软件制作有一个清醒的认识。我们不能理所当然地认为现在软件制作方式就是最好的、不变的。例如，我们可以根据这些摸索规律，可以预测今后有的软件会采用网络方式进行项目分解、模块功能开发的认领、软件组件的上传组合、软件网上测试方式，充分利用网络上无穷的开发人力资源和人力智慧，高效、低成本地完成软件的开发。

6. 预测所在企业发展

企业的发展与程序员发展有相当大的关系，企业发展了，程序员才有可能发展，否则，只能跳槽去寻找更好的发展空间。企业的发展，尤其是软件企业的发展也有其规律的，从小到大，从产生到消亡，而这些变化是和这个企业的软件、开发人员、市场、管理水平等密不可分的，尤其与软件类型密切相关。如果这个企业只能开发定制软件，无论定制软件数量有

多少，这个企业未来只能处于中等或中上等水平，不可能有翻天覆地的发展。如果这个企业目前和将来会开发通用软件，而且这个通用软件有很大的市场，那么这个企业未来就会处于上等水平，前途不可估量。知道这个规律后，程序员看看所在企业的软件类型就可以知道企业未来发展前景了。

放眼望去，软件世界中存在着各种各样的规律，我们只有了解和掌握这些规律，才能预测各种事物的发展趋势。而想了解和掌握这些规律，我们需要做这方面的有心人，通过学习，通过自己的经验和体会，慢慢学会发现和总结这些规律，并通过将过去发生的和正在发生的事件放入这个规律中进行验证，如果验证结果正确，说明规律很科学，如果验证结果不正确，我们就要修正我们所说的规律。在不断的发现和验证过程中，规律就浮出水面了。掌握了这些规律后，我们就可以比别人对事物有更科学的认知，我们就可以看到 1 年以后、3 年以后、5 年以后甚至更远的时候事物的状况，就可以别人看得更远，就可以比别人做得更早、更好。我们可以站在巨人的肩膀上看得更远，我们也可以站在自己的基石上看得更远，这个基石就是我们对趋势的预测。

4.6 有意识才会有行动——谈谈市场意识

我想套用一首歌曲中的歌词：“我的心中只有你没有他”，程序员心中只有程序没有其他。这的确是很多程序员内心的写照，很多程序员梦想成为编程高手，想把程序编得非常有技巧，非常有效率，非常有“艺术性”。别人编不出来他能编出来，别人编出来他能编得比别人更好、更快。至于其他任何事，程序员都懒得关注。总之，程序员很纯、很技术、很程序。

可以说程序员在入门阶段和成长阶段有这种想法是无可厚非的，这种想法可以让程序员集中精力，专注于编程，非常有益于快速进入角色。很多程序员看书或编程序，那可以说是废寝忘食、夜以继日。对程序上面的争论更是激情满怀，不惜面红耳赤。粗略统计了一下，程序员 90% 以上的工作时间都花在程序上，有的业余时间大部分也花在程序上。

如果有人跟程序员谈及市场问题的时候，很多人都会感到不耐烦，认

为这不是他们关心的事，是老板的事，是领导的事，和自己无关。还有一些人认为只要有过硬的编程技术，就不愁没有高收入。这些把自己的收入决定权完全交给别人，而别人可能以最低的工资获取最大的劳动效能原则来支付你的收入，你的收入达到想要的程度几乎不可能。

程序员这么专注程序其实是一个误区。程序员说到底就是一种职业，是因为谋生而从事的职业。就如同当今这个世界还没有发展到物质极大丰富阶段，人们依然通过从事社会劳动而生存着，谋生是生存的第一前提。在商品经济社会，人们首先通过社会劳动获取劳动报酬以维持生存。即使有程序员说自己是因为兴趣而选择了软件这个行业。但是，我们要问，其兴趣何来？很大程度上是因为这个行业可以给人以挑战，可以给人以成功。而且那些成功的诸如比尔·盖茨、微软的个人和企业更加令人心往神至。获取价值应该是当前人们追求的目标。有了这个目标后，人们会选择各种职业以获取价值的最大化。获取更高的收入是目的，而从事程序员职业则是一种手段。简单地说：目的是一种意识，行动则是一种手段的体现。

在市场经济社会，要想获得最大价值，就必须有市场意识。因为市场关乎企业的发展，市场关乎你的收入，市场关乎你开发的软件，市场关乎你自身的价值。在这个社会上很少有绝对脱离市场的事物，何况职业是市场中一个要素。

有人会问，有了市场意识之后，程序员就会怎么样？我以为程序员有了市场意识之后会在下列几个方面有所改变。

1. 收入

收入是程序员的工作的起点也是终点。有了市场意识之后，程序员就会关注软件行业的平均收入，关注所在城市软件行业的平均收入，关注行业同类型软件行业的平均收入，关注单位同事的平均收入，将这些收入和自己的收入进行比较，看看差距是否合理。如果感觉自己的收入还行，则可以继续在这个单位进行努力，如果感觉自己的收入太低，并且看不到未来改变的预期，则可以考虑换一个岗位和企业。

2. 技术

其实世上本无纯技术，表面上来看，技术是为了生产满足人们需要产品的智慧和手段。其本质依然是人们获取最大利润时的手段。新技术的产生、某种技术的流行、某种技术的衰败都是与市场密切相关的。有了市场意识，程序员就要审视自己的技术现状，对流行的技术一定要掌握，对新技术一定要关注，对将要淘汰的技术一定要放弃。程序员不能抱着现有的技术，不管其是否流行，能用就行的态度，否则会面临“书到用时方恨少”的局面，原来技术的弃用和对新技术不掌握会导致程序员英雄无用武之地，影响未来工作。

3. 产品

软件产品是程序员劳动成果，软件产品只有出售给用户才能获得其价值，从狭隘的角度上来看，软件产品价格越高或者软件产品销售的数量越高，则程序员凝聚在这个产品中的价值就越大，程序员身价就会越高。有了市场意识之后，程序员应该积极主动去了解自己参与开发的软件产品的销售价格，了解非自己参与开发的软件产品的销售价格，了解软件产品的用户数和这个软件当前销售金额的总额，以及未来这个软件产品可能的销售市场。这样程序员就会慢慢地养成对软件价值进行评估的习惯，在可以选择的情况下，程序员会主动选择参与到那些市场价值大的软件产品开发。一般而言，项目越大，用户数越多的软件对程序员的要求也越高。参与到这些项目之中可以大大提高自己的技术水平，并且为自己获取更高的收入找到一个有力的支点。

4. 环境

程序员的环境因素很多，其中有单位内部的，如：工作场所、计算机设备、团队氛围、人际关系、与领导关系、部门间协作等。也有单位外部的环境，如：与外部合作伙伴、客户、用户之间的关系。这些环境对程序员的成长和未来是至关重要的。比如在团队氛围问题上，程序员一方面要寻求整体团队氛围最佳状态，这样团队才能发挥团队作用，最终影响到软

件开发效率和开发质量；另一方面要在团队中多和高手进行交流，从中学习他们的技术和价值观，成为最好的朋友。又比如，用户关系，程序员一定要注重和用户的关系，一方面通过建立用户关系可以更好地了解软件的使用状况和存在的问题，另一方面可以实地了解软件在用户单位的地位和作用，也可以了解自己的真正价值，更重要的是要把自己的能力展示给用户并与其做朋友。我们常常发现自己的“同事”突然变成自己的“用户”了。这说明人家的市场意识强，在你不经意间完成了工作的转换。所以，我们要有市场意识，更多地以市场意识来看待环境因素，有的放矢地建立各种社会关系和收集社会资源，为自己今后更好的发展做好充分准备。

5. 价值

说到底市场意识就是价值意识，我们所思所想所作所为，都是围绕价值展开的，一个优秀的程序员必须具备市场意识，在市场中看到价值，在市场中体现自己的价值，让自己的价值最大化。市场越大，你的价值就越大。因此，优秀的程序员要尽最大可能跳开自己编程的小圈子，从你的周围、单位、用户、单位外部发现现有价值和未来价值，让自己融入市场之中，这样自己才会看到自己的价值，自己的价值才会逐步地变大，自己的价值追求才会越来越高。通过价值来调整自己的行动是最有效的成功之路。

我们说意识决定行动，当你没有市场意识或市场意识很薄弱的时候，你就会在市场经济的大潮中随波逐流，风平浪静的时候，你会感到安逸；当浪涛汹涌的时候，你就会感到由颠簸产生的恶心，甚至被大海淹没。当你有了市场意识之后，你就会不断去寻找大海中的生命之岛，时刻准备着救生衣，准备着木筏，锻炼好自己的体质和能力，无论是风平浪静，还是浪涛汹涌，都会向生命之岛划去。

4.7 制作有价值的软件才是程序员内在的目标

很多程序员不关心自己编制程序的价值，一些新手沉迷于“学习”，编制一些“学习”程序，以达到练兵学习的目的；有的程序员天天忙于公

司布置的工作，也不清楚这个程序用在什么地方，这个程序能卖多少钱。他们认为这些都是公司的事。他们一点市场意识都没有，认为在市场方面，他们毫无话语权。也有一些程序员虽然市场意识很强，通过各种关系去找项目，但是项目也仅仅是杯水车薪，远远达不到挣大钱的目标，而且是饥一顿饱一顿，永远处于项目的“有”和“无”之间。那到底什么是有价值的软件呢？可以从以下几个角度来看：

1. 软件层次

我们可以从软件的应用层次上来看，软件是分层的，从低到高：操作系统软件、各种驱动软件、工具类软件、办公软件、开发工具、数据库、BI（商务智能）、应用软件等。这些软件的价值体现在它们的使用价值和销售价格上。像操作系统、开发工具、数据库、BI等低层的核心软件基本由国外公司所垄断，虽然其价值很高，但是中国公司很少有能力与其竞争。因此，在中国这种急功近利的现状下，鲜有程序员去研发这些重要的基础软件。但是，不排除未来我们会补习这些重要功课。另外，我们看到应用软件是中国软件公司和程序员最能发挥才华的地方，但是应用软件的价值有高有低，不可一概而论。

2. 软件数量

我们也可以从软件的数量上来看，有的软件只此一套，称为定制软件。有的软件可以拷贝无数，称为通用软件或商品化软件。当然通用软件套数是受到这个软件的可能用户数所限制的。例如一个杀毒软件，它最高不可能超过计算机用户数。一个财务软件可售套数它最高不可能超过企业总数。一般情况下，软件可销售的套数越多，软件的价值就越大。当然也有一套吃天下的定制软件。

3. 软件售价

我们也可以从软件的售价上来看，有的软件（含项目）价格极高，有的甚至达到亿计，有的软件价格很低，甚至只有几十元。我们一方面可以从价格上来给这个软件作个价值判断，比如这个软件在100万上就感觉价

值很高了。另一方面我们要通过软件数量×软件单价来计算这个软件的价值。例如，软件A的单价是1万元，软件预计销售套数为5，则软件A的价值约为5万元；软件B的单价是100万，软件预计销售套数为1，则软件B的价值约为100万；软件C的单价是1万，软件预计销售套数为1000，则软件C的价值为1000万；就软件A、B、C来说，软件C价值要大于软件B，软件B价值大于软件A。

4. 销售形式

我们也可以从软件销售形式上来看，有的软件通过产品方式提供给客户，软件公司收入软件的产品销售价。有的软件是通过收取服务费方式提供给客户使用。通过产品方式的价值比较容易计算，有一个算一个，而通过服务费方式计算，尤其是通过网上服务方式，则相对比较复杂一些。服务方式的价值往往取决于用户数和用户使用的年限。另外，服务方式的各种折扣也相对多些，例如10用户会收取90%的服务费，1000用户可能会收取70%的服务费。从发展的角度上看，软件服务方式比重将逐步增加，其价值也在逐步增大。

5. 是否收费

我们也可以从软件是否收费上来看，绝大部分软件是收费的，但是也有一些软件称为绿色软件，可以供用户免费使用。关于免费软件的价值情形比较复杂，有的免费软件是软件公司的赠品，有的免费软件是程序员自身能力的表现，有的免费软件是其他原因所致。但是，本文所谈的重点都是收费软件。

软件的价值无论大小都体现了软件有用性的大小，软件的有用性则又受制于软件用户数和软件的功能。中国的程序员无论在单位还是在家，都要注重自己所编制软件的价值，只有编制有价值的软件，程序员自身的收入才能提高，程序员社会地位才会提高，程序员自我实现度才能提高。有价值的软件的增加说明软件在国民经济和居民生活中的应用程度得到了大大提高。

我们可能听到一些程序员说出这样的话：

1）我们编什么程序我们做不了主，公司让我们编什么我们就编什么。我们才不管软件有没有价值。

2）我们想知道自己编程的价值，但是我们无法获得自己编制软件的销售价格和套数。

3）我们每天特别忙，加班加点，知道价值怎么样，不知道价值又怎么样。

4）我们很想编制有价值的软件，但我们不知道什么软件是有价值的。

程序员命运是否掌握在自己手中，虽然不能完全取决于自己，但是要争取取决于自己，程序员可以通过编制软件的价值来判断公司成长性的高低，进而选择自己的未来。程序员职业特点说明程序员有很多的自有空间，这个空间可以做自己想做的事，而选择有价值的软件制作则是中国程序员内在的和可行的目标。

4.8 从程序制作到架构制作

从技术层面上来看，优秀程序员和一般的程序员相比，其中一个重要区别就是重心从程序制作转变到架构制作。绝大部分程序员关心的是如何将各种各样的需求通过计算机语言编程实现。因此他们有三个方面的重点：需求、编程、需求到编程的那个过程。他们的主要精力在用于需求是什么？需求怎么变成功能设计？如何实现功能的程序编写？他们的出发点是需求，终点是程序。

随着编写程序的数量增多，慢慢就会发现有两条主线同时在运动，一条是用户需求，一条是程序编写。就用户需求而言，我们发现不同用户的项目之间的需求比较类似，做的越多看到相同的需求或类似的需求就越多。在程序编写方面，我们面对相同需求或类似需求的时候，就会自然用原来的程序或修改原来的程序来完成。同理，程序编写的越多，我们用到原来程序的次数就越高。这种现象和需求趋同的规律是一致的。在项目开

发过程中，我们就会自然而然地将相同的需求用相同的程序来完成。于是产生了需求和程序之间的对应关系。例如，我们想要在 A 项目中检查一下用户的权限，我们自然就会想起以前在 B 项目编写过的 check_usr_right 这个函数。于是我们就会把这个函数拷贝过来用在 A 项目中。虽然拷贝的函数并不能完全满足项目 A 检查用户权限的要求，但是我们可以对 check_usr_right 函数进行改造，这样就成了新的 check_usr_right 函数了。如此一来，我们就会积累出很多相同功能的程序，包括公用函数、控件、类库等。这种编程方法和最初一心实现需求，采用原始的连续性编程方式相比，已经开始考虑软件架构问题了。

那么什么是软件架构？软件架构就是站在软件自身角度上（而不是站在需求角度上）编制软件的技术方法。通俗地说是软件之上的软件。如果我们把一扇窗户看做一个项目，那么窗框就可以看做架构，而窗框里的小框和玻璃以及玻璃上的贴画和贴膜，则可以看做小架构和架构下的各种功能。我们可以看到，窗户的形状（方形、长方形、半圆形、异形等）是由窗框决定的，而窗框中的小框和玻璃等包含在窗框范围之内。因此，我们可以这样认为，软件架构决定了这个软件的大范围和架构，而其中的功能软件则是在这个架构之下的功能实现。

既然我们知道软件架构是软件之上的软件，那么软件架构和我们一般程序有什么不同呢？我认为其不同之处在以下几个方面。

1. 抽象和具体之间的差别

一般程序主要是解决具体的需求，从而形成一个具体的功能。比如，要计算 1 +2 等于几，非常之具体，程序员容易理解。而软件架构并不针对一个具体的需求，比如，要进行算术表达式处理。当然这个算术表达式，也可以处理 1 +2 等于几的需求，但是，这个算术表达式是抽象的，并没有具体的需求，但是能处理具体需求。

2. 大和小之间的差别

一般程序主要是解决具体的需求，程序很小，功能点也比较直接，并

不复杂。而软件架构则是软件之上的软件，需要考虑问题比较多。例如，需要考虑软件的一个整体架构，架构中各种模块的功能，架构中模块之间的关系和接口，架构怎么与具体程序相联系等各种问题。因此，架构程序量是很大的，功能也相对复杂。

3. 高和低之间的差别

一般程序在解决需求时并不会给软件分层，最多是按 B/S、C/S 或者是按表示层、逻辑层、数据层三层架构来进行。另外，一般程序只要能满足具体需求就 OK 了，其技术上的并没有硬性的要求。但是，软件架构就不同了，它将软件进行了分层处理，最上面是架构，最下面才是具体功能。因此，软件架构对技术要求相对比较高，比如程序的结构、程序的效率、程序的通用性和扩展性、程序的并发处理、程序的例外处理等。这样有了软件架构概念之后，软件就分为架构软件和具体软件两个部分了。架构软件处于上层，可以应用于各个项目，而具体软件或称具体程序则处于下层，这个具体程序和我们现在所说的一般程序很接近，这两者结合就形成了具体的项目或系统。

4. 全面和局部之间的差别

一般程序主要是解决一个项目的某个具体需求，因此，它体现出程序的局部性，所以一个项目可以给不同的程序员完成。软件架构在设计时或者在制作时，则是站在全局的高度来审视整个抽象的项目，一方面要考虑架构本身的功能，另一方面要考虑到如何向具体程序提供合适的接口。这两个方面都需要对项目或软件有一个整体认知，而不仅仅局限于某个局部。

当然以上的差别是非常宏观的，并不很具体。但是，有一点是可以肯定的，那就是制作软件架构是和编写一般程序不同的。这一点需要程序员细心体会。如果程序员没有软件架构的意识，只是满足于停留在一般程序编写的优劣之中，那是很难进入优秀程序员行列之中的。那么什么是软件架构的主要内容呢，程序员应该关注那些问题呢？

1. 架构意识

意识——我仍然强调意识的重要性。没有对某个事物重要性的意识，就不会关注和研究这个事物，就不会主动掌握这个事物，就不会利用这个事物去创造更大的价值。同理，如果我们的程序员没有架构意识，就会在普通程序员中求得那种优越感，时间一长就感到自己在软件中没有可以发展的地方。而有了架构意识，我们的程序员就可以在原有的编程基础上，向更高层面发展，那里有更大的空间需要我们去学习和展示自己的才华，可以说架构领域中的学识和技术是没有止境的。

2. 架构制作

光有架构意识还不够，还要让架构从设计走向程序，这就需要进行架构制作，也就是说我们要编制架构程序。而编制架构程序就涉及具体的程序设计语言，涉及具体程序语言的特性、编程技术和技巧。这些编程需要程序员拥有很扎实的编程基本功、很高的程序设计技巧以及程序编制的丰富经验。架构制作绝大部分都是面向对象、面向抽象功能、面向通用、面向参数化的编程。

3. 架构应用

我们不能为架构而架构，我们编制架构的目的就是可以更快、更好、更节约地开发出我们的应用系统。如果我们编制的架构不能运用到具体项目开发中，或者编制的架构反而比普通的编程效率更低，质量更差，达不到我们预期的目标，那么我们的架构只能是失败的架构，失败的架构比没有架构更加糟糕。因此，我们要编制那些有用的架构，并且通过具体项目的开发来验证架构的正确性和有效性。可以这样说，开发的项目越多，积累的架构也就越多。只有这样我们才能让架构更具有实用性和实用价值，在开发和运用中发挥越来越大的作用。

4. 架构研究

架构一方面是一个实用的程序，另一方面也是一门学问，它需要理论

和实践不断地结合和完善以体现其价值。例如，如何划分架构程序与具体程序之间的界限；面对各种应用系统时是采用统一架构方式，还是采用多架构方式完成；在应用架构时如何设计和应用参数，并使得参数运用更加简便合理；采用不同程序语言制作架构时应该注意什么问题；如何处理架构与具体需求之间的关系；怎么利用具体程序与架构之间的接口实现动态程序的挂接等话题都是我们可能要面对和解决的。因此，我们要静下心来，学习架构的各种知识，大量积累项目开发经验，区分哪些属于架构方面的功能，哪些属于具体程序方面的功能，研究采用什么样的编程技术才能达到架构的最佳效果。理论和实践的结合才会让我们拥有“腾飞的翅膀”。

采用架构的方法开发程序可以让你的程序技术上个档次，让你的项目结构更加清晰，让你的项目开发更快、更好，让你的项目更具有可维护性和具备可扩展性。

没有架构的约束，我们可能更加自由，但是，有了架构的约束，我们可以让自由更加有价值。追求价值才是我们最终的目标。

4.9 从定制软件到通用软件

程序员对于通用软件并不陌生，从 office 办公软件到开发工具，从数据库到操作系统都是一睁眼就能看到的通用软件。通用软件的知名度以及通用软件的市场价值和技术含量都在程序员的心中烙下了深刻印象。很多程序员把能够参与制作通用软件看做自己的技术水平和价值水平的体现。目前通用软件仅限于系统类和工具类的软件，而绝大部分的应用软件都还处于一种定制软件开发阶段（应用软件中常常含有一些通用软件作为工具和控件）。

定制软件之所以成为主导，主要是因为我国企业信息化起步相对较晚，很多企业经营模式还处于转变和探索中，这种企业经营的不成熟导致企业经营的各个方面（如产品设计、产品生产、产品销售、客户服务、财务、管理等）个性化、非标准化现象非常普遍。即使是在同一个行业内部，不同企业的信息化也是不同的，每个企业都会按照自己的经营水平和

意识来开展信息化，而其他企业信息化很少去借鉴和参考。这种个性化的要求就必然导致定制软件的产生。

因此我们看到绝大部分程序员整天忙于定制软件的开发。但是，我们又看到了一个很普遍的现象，那就是程序员再怎么加班加点，再怎么技术提高，企业项目再怎么增加，软件企业的收入和员工的收入依然提高缓慢，有的甚至不提高，有的则长期处于停滞的状态。程序员收入太少的这种现象越来越多，值得我们深思。从某种角度来看这个问题，我们发现软件的个数在增加，但是软件的质量却没有提高，甚至下降。而软件的个数增加是通过大量增加软件制作者即程序员来完成的。这又从另一个方面说明，定制软件对程序员要求比较低，程序员很容易上手编制软件，加之软件企业之间不断竞争，使得软件价格日趋下降，最终影响到程序员的收入和软件企业的快速发展。所以我们有理由说，定制软件制约了我国软件的快速发展，也影响到了程序员的收入水平和技术水平。

由于工作关系，我接触到一些软件公司，它们十年如一日做着定制软件开发工作，公司每每希望有很大发展，每每希望落空，始终处于维持吃饭这种状态。而在一些著名的软件公司发展历程中，我们也可以发现这些公司正是凭借着通用软件而使得企业获得了飞速发展。由此可见，开发和销售通用软件是软件公司飞跃发展的必由之路。

很多人都会问，既然通用软件价值如此之大，定制软件弊端如此明显，为什么软件公司还去开发定制软件而不去开发通用软件呢？问题是，很多事道理大家都很清楚，但是做起来并非那么容易。通用软件也是如此。要开发通用软件至少会遇到两个问题，一是开发什么样的通用软件，二是有没有开发通用软件的人才。前者是市场问题，后者是技术问题。这两个问题都不是轻易能解决的。而定制软件市场虽小，但是没有什么技术问题，因此许多公司乐于这样做下去，毕竟不是所有软件公司都肯冒开发通用软件失败的风险。

虽然程序员开发着定制软件，但是并不意味着通用软件与己无关，尤其是梦想成为一个优秀程序员的程序员更应该对通用软件进行关注，充分

意识到通用软件对自己成长起的作用。

1. 通用软件关系到软件公司的发展

开发什么样的软件绝对和公司今后的发展有关，程序员必须对此有一个清楚的认识，并且始终将此作为对公司评判的重要内容之一。从理论上来说，虽然定制软件和通用软件都是程序员编制出来的软件，都是用来满足用户需求的，但是，定制软件只有一个用户，而通用软件有一类用户，并且这类用户都有相同的需求。因此，定制软件是不共享的。程序员所有的劳动都凝聚在这一个软件之中，所以，定制软件的人力成本太大，并且无法平摊到其他项目之中，形成了软件利润直接和人力成本挂钩的状况。而通用软件的用户则是一类用户，用户数远远大于1，软件是共享的，是由所有这类用户共享的。所以，通用软件的人力成本平摊到每个用户的时候，其成本是递减的。这就形成了软件利润直接和用户数挂钩的状况，也就是说，软件的制作成本往往可以忽略不计，而软件利润主要考虑用户个数，通用软件价值实现主要在软件营销上，而不是在软件制作上。通过以上分析我们可以看到，通用软件可以有效减少软件制作中的人力成本，可以大大增加软件的销售收入，从而提高软件公司的利润，促进软件公司的快速发展。

优秀程序员在单位应该有一定的话语权，应该深知通用软件在公司发展中的巨大作用，在公司进行项目选择和项目制作之中，应尽可能促进公司向开发通用软件方面发展。只有这样公司才能逐步摆脱定制软件的束缚，向通用软件靠近。

2. 通用软件关系到程序员自身收入

程序员要想增加其收入，大的前提条件是企业要有收入和赢利，一般来说，企业发展了，程序员收入才有可能提高。制作通用软件正是促使公司发展的一种有效方式，所以，程序员要想提高自己的收入，就必须积极推进和参与通用软件的制作。

3. 通用软件关系到程序员技术水平

很多公司不去开发通用软件一个很重要的原因就是没有这方面的人才。绝大部分的程序员长期养成了开发定制软件的习惯，习惯于需求是什么就编写什么，习惯于什么时候需求变化就什么时候更改程序，习惯于用户发现程序问题再去解决问题，习惯于不断地维护程序。而通用软件更多地要求程序功能明确（意味着需求的确定），程序通过参数化来适应需求的变化。因此，它不允许程序出错，不允许有太多的维护工作。否则，销售出去的软件越多，更新维护的成本就越大，软件的声誉就越差，最终导致这个通用软件的失败。要做到软件的通用，首先，需要对这类用户需求的抽象，使得软件需求能满足这类用户的需求，而不是针对某个用户的需求。其次，在实现这些需求的时候，软件的模块和功能要更加具有逻辑性，便于软件结构的清晰，便于制作，便于程序质量的提高（不能出错），便于程序的可用性。再次，通用软件不但要能满足这类用户的通用功能，而且要能满足这类用户的个性化要求，这些要求可以通过参数配置、接口调用等方式实现。最后，通用软件的编写要能实现参数化，实现软件的可扩展性。要做到这些，程序员墨守成规，不去做大的改变是无法完成的。当你习惯用编写通用程序的方法去编写定制软件的时候，你就会感到莫名的快感。而当你用编写定制程序的方法去编写通用软件的时候，你就会感到寸步难行。也许这就是两者之间的差距。

现实中，很多软件公司拥有一定的用户群，开发的软件也大同小异。有的公司则逐步积累了一些行业中的软件模块，形成了通用软件的雏形，它将可以复用很多程序，减少重复开发的成本，从而获得更高利润；而有的公司则没有这方面的意识，每个软件都是独立开发，一个用户一个软件，而且一个版本一个软件，始终处于一个软件开发初级阶段，开发成本巨大，利润减少。无论所在的公司或单位开发通用软件与否，程序员都要有强烈的通用软件意识，始终有将定制软件转变成通用软件的愿望，并且在技术上用制作通用软件的要求约束自己，不断提高自己的技术水平。程序员也许不能开发出完整的通用软件，但是可以在编写定制软件中，按照通用软件的要求，编制一些通用的函数、通用的功能、通用的模块，使得

这些通用的小程序能够复用和共享，为今后有机会编制完整的通用软件打下良好的技术基础，只有这样程序员的最大价值才能够有在通用软件上体现。

虽然当前定制软件是主流，但是未来一定是通用软件成为主流。我们正处于这两种软件形式的过渡期间，我们不管这个过渡期有多长，只要我们能够看到这个趋势，并且迎合这个趋势，我们就能先人一步，成为软件潮流的领跑者。定制软件只是我们现在的经历，而通用软件才是我们未来的理想。

4.10 何为 EOM

企业经营模型（Enterprise Operating Model，EOM）是从企业经营的全局出发，创新思维，通过模型理论和软件新架构技术而建立的一种企业信息化解决方案。

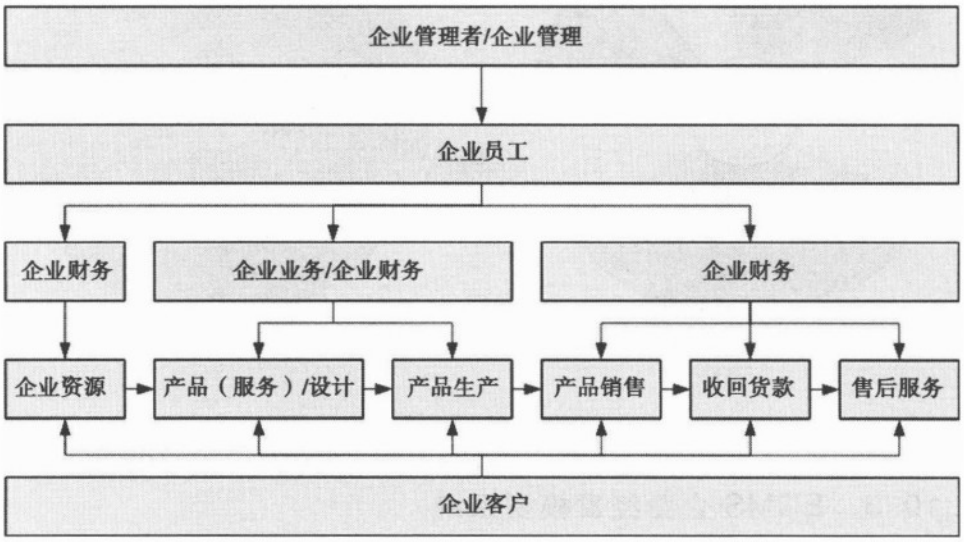
4.10.1 企业信息化现状

当前企业信息化发展喜忧参半。采用“需求驱动”、整体解决方案（例如 ERP、CRM）、流行技术等方式从局部上暂时解决了企业信息化中的具体问题，促进了企业信息化的向前发展。但是，以上方式都是从企业经营的某个角度而不是全局看待企业信息化。因此，它们就必然造成企业信息化的无序发展，造成企业内部系统建设的重复和信息共享困难。信息化发展到一定程度后，必然会对企业经营发展产生负面的影响。反映到企业、行业之间，表现为各自为政、不通有无、资源浪费。许多企业经营者、软件公司、软件开发队伍对企业信息化究竟何去何从，充满疑虑和困惑，甚至不清楚企业信息化应该“做什么”和“怎么做”。

4.10.2 EOM 及其作用

促进企业经营发展是企业信息化的出发点和最终目标。EOM 正是抓住了企业经营这个核心，对日常经济活动中大量的企业经营行为进行分析、整理、归纳，并抽象为企业资源、客户、员工、产品或服务、业务、财

务、管理七大要素，通过这七大要素之间的相互关系，最终建立企业经营模型。这个模型客观、系统、科学地涵盖了企业经营的整个过程，反映了企业经营本质特征，如下图所示。



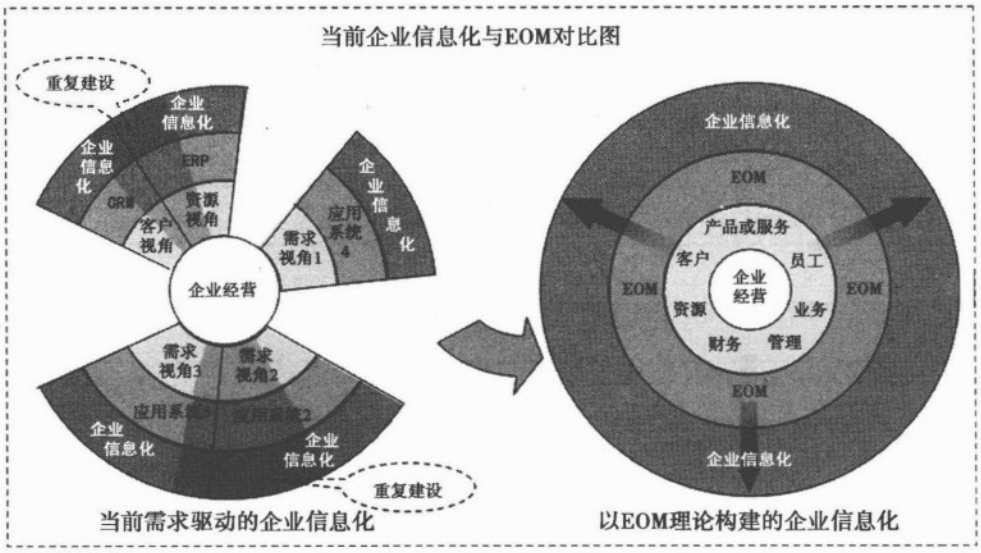
利用企业经营模型来构建企业信息化模型，使得各个企业有可能结束“需求驱动”方式开展信息化的局面。使得企业信息化从无理论转为有理论，无序转为有序，无规划转为有规划，不全面转为全面，不通用转为通用，浪费转为节约，不共享到共享。业务发展水平和管理水平不一的企业有可能通过 EOM 的应用系统享受到最好的业务和最好的管理，真正享受到企业信息化对企业经营发展的促进作用，如下图所示。

图示表明：

1）EOM 包含整个企业经营，而现有的只包含企业经营局部。

2）EOM 从整体来架构企业信息化，不产生重复建设，而现有的会产生重复建设。

3）EOM 面对的是企业经营模型，具有通用性，而现有的面对的是具体企业，只具有个性。

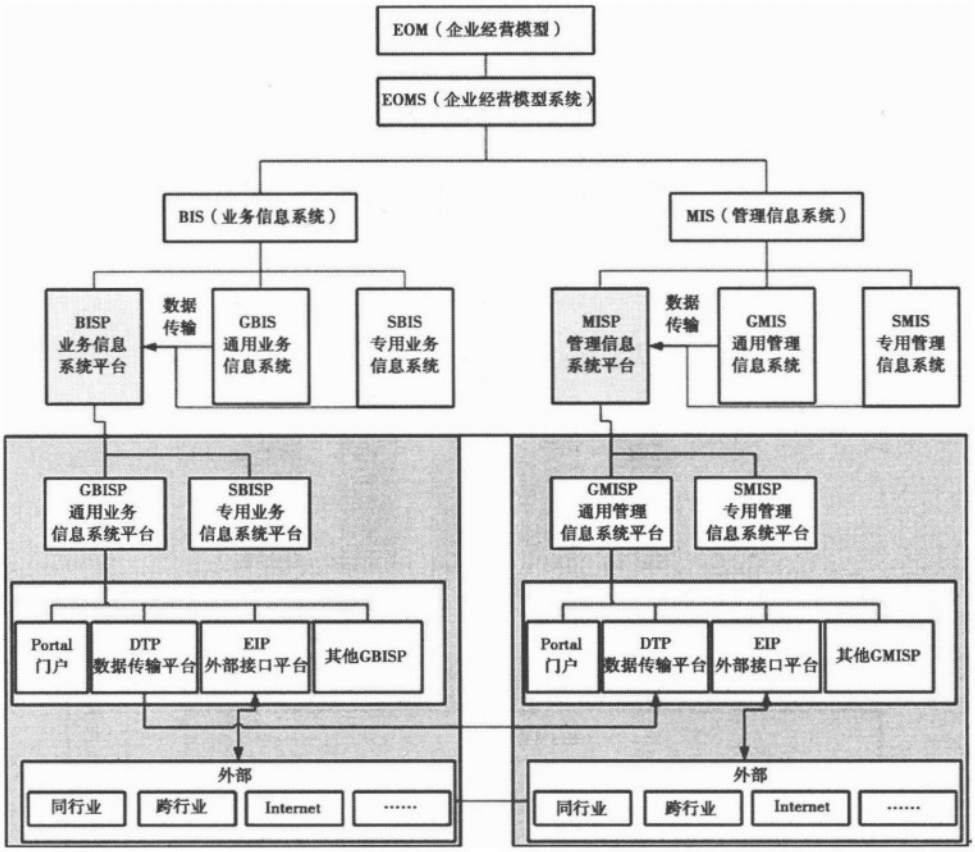


4.10.3　EOMS 企业经营模型系统

EOM 把企业信息化作为一个系统工程，即建立企业经营模型系统（Enterprise Operating Model System，EOMS）作为信息化的整体架构，并根据资源、客户、员工、产品（或服务）、业务、财务、管理这七个基本要素之间的相互作用和联系，将 EOMS 分成业务信息系统（BIS）和管理信息系统（MIS）两个基本架构，实现了业务和管理分离。由于资源、客户、员工、产品（或服务）、业务、财务、管理是 EOM 从大量的企业经营活动中抽象出来的基本要素，由此建立的 EOMS 可以涵盖企业经营的方方面面。例如，EOMS 中管理信息系统（MIS）就分为通用管理信息系统平台和专用管理信息系统平台。而通用管理信息系统平台上的人事管理、财务管理、绩效考核、客户管理等多种通用产品就是针对客户、员工、财务、管理等要素而设计的。它完全可以适用于各行各业，从而实现跨行业、跨企业、跨部门，如下图所示。

EOMS 按照 EOM 的 4 个要素以及相互关系，采用了平台和通用系统（见下图）以及专用系统相结合的架构形式。它解决了企业信息化中的共性和个性问题，解决了企业信息化中“做什么”这个最重要的问题。

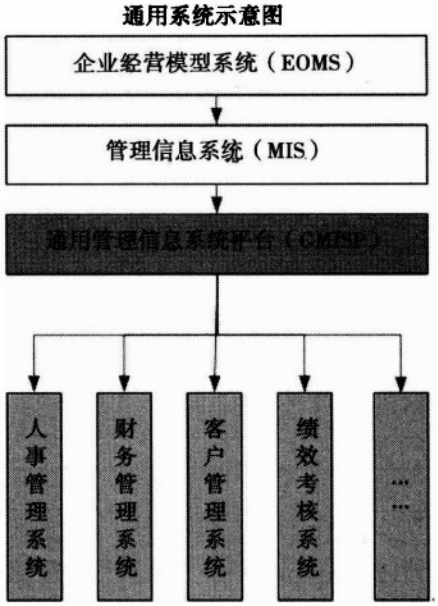
EOM企业信息化架构图

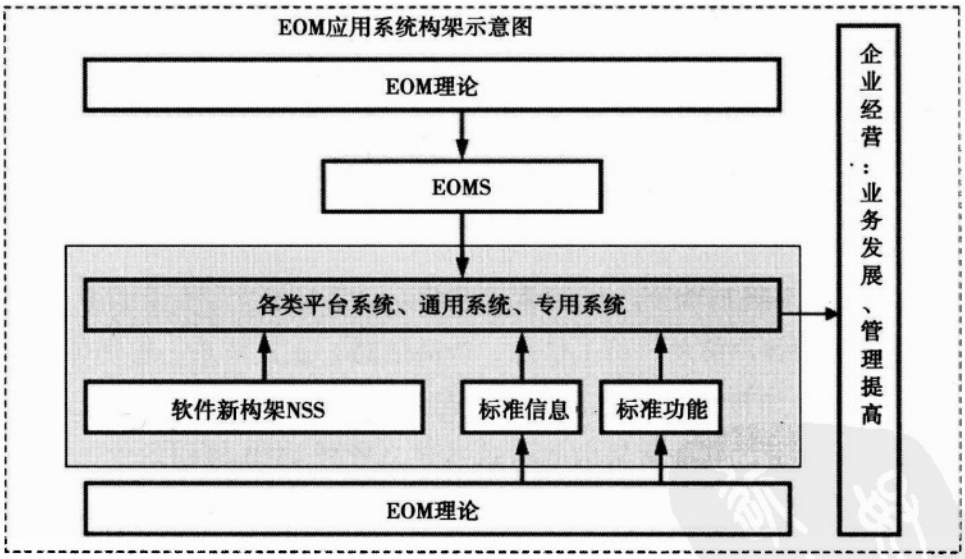


EOMS 还按照 EOM 的七个要素以及相互关系，建立了企业信息化中的标准信息（如客户信息、员工信息、产品信息、业务信息、管理信息等）和标准功能（如接口、参数、权限、查询、打印、数据转换、信息交换等）概念，并通过软件新架构 NSS 技术建立各种 EOM 应用系统，如下图所示。因此这些系统具有了通用系统的特性。EOM 解决了企业信息化中另外一个重要的问题："怎么做"的问题。

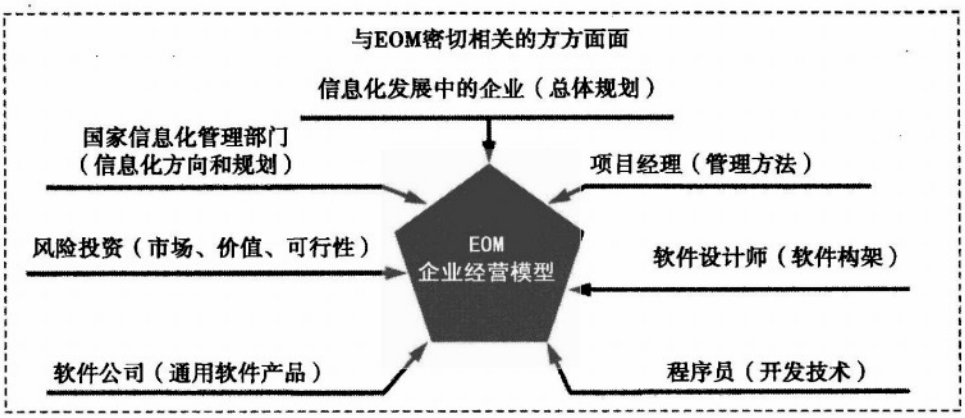
4.10.4 与 EOM 相关的单位和人员

EOM 解决了企业信息化中"做什么"和"怎么做"的两大难题，必然引起企业信息化产业链中各个方面的关注。国家信息化管理部门、信息化中的企业、风险投资公司、软件公司、软件项目经理、软件设计师、程





序员都可以和EOM发生某种联系，而这些联系可以促进相关人员重新思考企业信息化的现状，重新找到各自在企业信息化中的位置，如下图所示。



4.10.5 EOM 的特点和未来

与流行的 ERP、CRM 等企业信息化解决方案不同，EOM 是对中国企业信息化发展长期跟踪、分析、总结、归纳、抽象的结果，它根植于中国软件这片沃土，它带有强烈的中国烙印。随着今后对 EOM 的深入研究以及 EOM 产品的面世，将会有越来越多的人认识到其绝非是一种单纯的理念，认识到其巨大的实用价值，以及其在中国软件业发展中的重要作用。

EOM 特有的“中国创造”、模型理论、架构规划、通用软件、标准信息、标准功能、软件新架构技术内容，将会引起越来越多人们的关注。有关更详细的 EOM 介绍请看 EOM 系列文章，以及相关的介绍。敬请关注 EOM 的发展。

4.11 用 EOM 的眼光评判“我要做全国最好的标准权限组件和通用权限管理软件”1

光靠理论上阐述 EOM 显然不太容易理解，现在举例加以说明。我在博客园中看到了《我要做全国最好的标准权限组件和通用权限管理软件》系列随笔。作者根据自己的亲身经历，提出要做全国最好的标准权限组件和通用权限管理软件。当然“全国”、“最好”、“标准组件”、“通用软件”这些词确实很吸引眼球。我们不能望文生义，较真于“最好”，毕竟作者作为程序员还是有想法的、有激情的、有勇气的、有市场意识的，想

按照自己的想法去做，从而实现自我价值。就这些而言，我认为有很多程序员过于平淡、过于安于现状，应该从他的身上发现自己的不足，学学其中的闪光点。

权限管理几乎每个应用系统都会涉及，也是重复开发的重灾区，但凡做过项目的人都应该涉猎到！许多人开发一个系统就编制一个权限管理程序，建立自己的用户标准、功能标准、权限标准，也有不少人开发一个系统就复制一个权限管理程序，并在此基础上进行修改，形成新的权限管理程序。当然也有一些人开始考虑编制通用的权限管理程序，希望这个程序能够在更大范围内的通用。所有这些努力都是在功能级上考虑程序的复用，以减轻开发量，实现权限管理的标准化。他们特点就是**从实践中来到实践中去，为程序而程序**，慢慢地逼近科学。

但是，我们怎么来评判这个权限管理呢？也就是说我们评价依据是什么呢？通常我们对软件评价只是停留在主观判断层面上。比如，功能完整不完整、功能实用不使用、程序复用不复用、程序用户数多不多、程序技术含量高不高。所有的评判标准由评判者来定。这样评判者既是运动员又是裁判员，容易做出有利于自己的评判。另外评判者的水平高低也决定了这些评判的水平高低。而该论断的作者正是依据主观判断得出了自己是“最好的”结论。

我在这里要换一个角度来对此进行评判。那就是按照 EOM 理论来评判权限管理。

1. 权限管理由来

首先为什么要有权限管理？很多程序员认为此问题可笑：一个应用系统怎么可能没有权限管理呢？权限管理是必需的。

那为什么不能没有权限管理？为什么要有应用系统？为什么要有企业信息化？为什么要有企业经营？如果我们提完这一连串问题之后，就可以这样思考问题了：

企业要生存必须进行企业经营，企业经营需要企业信息化，企业信息

化需要建立各种应用系统，各种应用系统需要用户来使用，不同用户使用的功能可能会是不同的，相同功能因其职位和岗位不同也有内容方面的限制。而这些不同和限制就是权限的业务含义。

于是我们对权限理解和企业经营建立了联系，把权限上升到企业经营的高度，使得权限管理有了最原始的依据。

2. 权限管理和企业经营

程序员在编制权限管理程序时，根本不会考虑权限管理与企业经营的关系，所有权限管理的需求来自企业的需求方，他们默认需求是正确的。而实际上企业需求反映了社会需求的一般水平，也就是说需求不会是最好的，这样就会导致程序功能并不是最好的。也许程序员会说这个责任不在于程序员，而是企业的需求者。问题是我们不是来追究责任的，我们是来寻求权限管理最好的解决方案的。

权限管理和企业经营挂钩，表明不同的企业经营方式会导致不同的权限管理的需求。差的企业经营方式会导致权限管理的需求质量很差，这会加深企业经营得差，好的企业经营方式会导致权限管理需求质量好。这就充分说明了权限管理和企业经营挂钩后，我们可以依据权限管理是否促进企业经营向前发展作为评判权限管理好坏的标准。

因此，我们可以看出难的不是实现企业的需求，难的是对企业需求的评判，以找到一种最优的需求。这种需求的最终目标是促进企业经营更好的发展。

3. EOM 方法

如果我们用 EOM 来看待权限管理，我们就会根据 EOM 推导出 EOMS，根据 EOMS 推导出 BISP 和 MISP，从 BISP 和 MISP 推导出系统管理，从系统管理中推导出权限管理。

这样我们的权限管理就从具体的企业需求变成了 EOM 中一个通用功能了。权限管理和 EOM 建立了联系。

建立联系的目的是，我们要寻找科学的权限管理的需求，这种需求不但能满足具体企业的需求，而且要向具体企业提供更好的权限管理的模式，以提高具体企业经营管理水平。例如，某企业提出了管理系统虚拟用户（非实名）加权限的需求，这种需求明显不利于加强企业管理，因为实名用户可以使得用户操作更加严谨，容易防范操作风险，便于对用户加强管理和事后问题认定。那么我们的权限管理应采用实名模式。又例如，企业提出客户系统采用实名（非虚名）加权限管理的模式，这种需求显然不利于客户便利性操作和客户的操作习惯，那么我们的权限管理应采用虚名模式。

4. 抽象的方法

那么我们如何实现 EOM 中的权限管理呢？我反对那种从实践中来到实践中去的做法，那个仅限于新手入门，因为靠实践无法穷尽所有可能的问题，所以我们要依据 EOM 理论，对权限管理本身进行抽象，从理论上定义什么是权限，什么是权限管理，权限管理的要素是什么，权限管理有什么模式，如何实现权限管理，以及实现后的权限管理对企业经营会产生何种影响。通过理论抽象，我们再把抽象出的权限管理拿到现实中去验证，如此反复，使得权限管理逐步趋于科学和最优。这个过程就是理论到实践，实践到理论反复的过程。

5. 整体观、大局观

但是，权限管理毕竟是单一的功能，如果我们的眼光只放在单一功能上，那么只能使这个功能孤立化，功能关联性和可扩展性就得不到实现。所以我们要站在更高的角度看待权限管理，例如我们要建立与权限相关的用户管理，那么我们就要考虑用户管理和权限管理的关系。只有考虑到用户管理以及相关的其他功能后再来考虑权限管理，这样权限管理就容易与其他功能实现对接。实现了企业信息化的一体化或整体化。

综合以上对权限管理的分析，我们再来看看“我要做全国最好的标准权限组件和通用权限管理软件”，此时就会发现作者做权限管理是根据实际中的需求不断改进而得来的，其权限管理的总体目标并没有联系到企业

经营这个根本的出发点和归宿点，而且没有对权限管理的各种要素进行科学的定义和抽象，也没有就权限管理和相关功能进行联系分析和考虑，更没有说明各种权限管理对企业经营的影响，没有提供最优的权限管理以改进企业经营管理的意识。从本质上来说，它还是一种“需求驱动”的程序，没有理论的支持，仅仅以满足用户需求（或自己需求）就可以了，所以就权限管理的总体架构和总体目标而言，作者还有很长的一段路要走。

仅凭主观评判一个程序就会导致仁者见仁、智者见智、众口难调的局面，往往导致差的程序得到好评，好的程序得到恶评的结果。所以我们不能以主观评判来评价一个程序，我们对程序的评判需要有一个理论的支持，这样评判才能客观和公正。而这个理论一定是问题的出发点和归宿点。而 EOM 站在企业经营角度，通过抽象方式对企业信息化推演，涵盖了企业信息化的方方面面，几乎所有软件功能都能在其中找到自己的位置，并以能否促进企业经营向前发展作为最终的评判标准。对一种权限管理进行分析评判，正是基于这个理论。

4.12 用 EOM 的眼光评判“我要做全国最好的标准权限组件和通用权限管理软件”2

上一节主要是从权限管理的目标和总体架构上作了 EOM 的阐述。在我看来，很多程序员并不关心“要做什么”，而是只关心“怎么做”。对权限管理而言，他们并不关心权限管理在企业经营中的作用，只关心权限管理的程序是如何编制的，这是令人感到非常遗憾的事。如果权限管理不能满足企业经营管理的需要，不能促进企业经营向前发展，权限管理编好了那又有什么意义呢？就我而言，如果没有了目标，做任何事都是没有意义的。

上大学本身是获取知识的手段，获取知识才是上大学的目的，但是，当前社会只关心大学是否毕业，只关心考试得了多少分，但不关心知识获取了没有。我学到的东西基本是在上学的时候，毕业后，虽然开发程序很多，但学到的东西比较少，尤其是理论的东西更少，我基本上是依靠理论来推导出各种解决思路的。

回到权限管理问题的本身，从平常的角度来看，权限管理的编程有两部分，一个是权限设置，一个是权限比对。这两部分本身都相对比较简单，没有太多复杂性。权限设置主要是向系统管理者提供一个用户权限设置的功能。权限比对主要是在程序中提供一个函数，用于检查用户是否有权限进入下一个功能。除了这两个部分，还有就是相关的参数设置等。

其实权限管理的亮点不在于编程，而在于权限管理的设计，有了好的设计，编程应该没有什么问题的。我准备从需求设计、功能设计、数据库设计、软件架构设计四个方面来谈谈这个话题。

需求设计

1. 权限管理的定位

1）一般人编写权限管理，主要是在一个系统内建立用户和权限关系。其特点是用户和权限范围仅限于一个系统。

2）也有能力比较强的人，探索的是一个企业内多系统的权限管理。其特点是用户和权限范围仅限于这个企业或这个企业的多个系统。

3）EOM 的权限管理定位于所有企业，它是真正意义上的通用。通用包括这两个含义：一个企业多系统使用或调用，多个企业多系统使用或调用。这样就可以避免大量的权限管理程序的重复开发了。

2. 需求分析

有了定位之后，那我们就要进行需求分析，明晰权限管理的内容和边界。

1）一般程序员对权限管理的需求来自于具体系统开发需求，他们关心的是这个系统的菜单是什么？功能是什么，有哪些操作？有哪些特殊的权限控制？如何设置用户的权限？是否有角色？角色是如何设计的？根据这些需求，明晰数据库设计和程序的流程处理。他们所有的需求分析都是建立在具体的系统之上的。他们并不关心什么是权限，什么是用户，什么

是功能，什么是操作，什么是角色！他们只要把他们认为的具体的内容可以对应就行了。例如，某个菜单查询，他们就把它作为一个功能。

2）水平稍高的程序员由于想建立企业内多系统的权限管理，往往会对企业内的各种权限管理进行需求分析，力图找到一些共性的东西，然后通过共性的东西对各系统进行权限管理的整合。他们的重点是如何把系统的因素考虑进去，并能实现权限的集中管理。他们的需求本质上还是来自具体需求，只是其中增加了需求共性的分析，朝着通用的方向迈了一步。

3）EOM 对权限管理的需求分析并不是针对某个企业或某个系统，而是根据 EOM 来分析员工和客户（用户）在企业经营中的职责和权利以及限制，以及建立与企业经营管理模式相适应的权限管理模式。因此，EOM 要从理论上解决：什么是权限？什么是用户？什么是功能？什么是操作？什么是角色？什么是权限管理的要素，以及要素之间的关系是什么？什么是权限管理模式，这种管理模式与什么样的企业经营模式相适应？当我们把这些问题考虑清楚之后，我们才能说对权限管理有了一个初步的认识。在此之外，我们还要跳出权限管理的本身的框框，站在更高的层次去看待权限管理与其他功能之间的关系，例如，用户管理、参数管理等。

有了这些理论和抽象的需求之后，我们就可以找一些具体企业和具体的系统去验证这些需求的正确性。

当然很多程序员还不习惯这种从理论到实际的思维方式，还是满足于从实践中来到实践中去，往往会说你，怎么会知道别的系统权限管理需求呢？你怎么知道你的权限管理能适应所有企业权限管理需求呢？因为他不能穷尽所有的企业，不能知道所有企业权限管理的需求，所以他们认为通用的权限管理是不可能的。究其根本原因，是他们不能将实践的问题抽象到理论高度，然后用理论来解决实际问题。

3. 需求设计

我们可以根据需求分析进行需求设计，通过一系列的定义，重新规范需求的内容和边界。这样我们就可以依据这个设计，展开功能设计、数据

库设计、软件架构设计了。

1）权限的定义。

权限是指系统用户的职能与系统的功能和操作之间的关系。

这表明权限主体是系统的用户，权限的根据是用户职能，权限所涉及的是系统的功能和系统的操作。

注意：用户的概念和职能的概念都是 EOM 中员工、客户、管理三大要素的概念以及其内容的延伸。

2）系统的定义。

系统是指企业为企业信息化而建立的各种应用系统。应用系统具有相对独立的、相对完整的功能，有相应的用户群。一个企业往往会建立几十个甚至成百上千个应用系统来满足企业经营管理的需要。

3）系统用户的定义。

系统用户是指使用系统的员工和客户。其中狭义上的员工是指企业的员工，广义上的员工可能包含上级或下级企业的员工。

狭义的客户是指购买企业产品或服务的用户等，广义上的客户可能包含企业的管理部门、行业协会以及潜在的用户人群等。

无论是员工还是客户都可能是系统的用户。上级员工和外部客户都可能访问企业的系统而成为系统用户。

4）用户职能的定义。

用户职能是指使用系统的用户（员工）按照其岗位职能要求，以及企业管理的要求，从事业务或管理活动。同时也指系统用户（客户）按照企业经营管理的要求，享受企业提供的各种产品、服务。

要注意无论是员工职能还是客户，所享受的产品和服务都是有条件限制的。

5）系统功能的定义。

系统功能是指系统中相对独立的功能模块，这个模块常常具有用户界面，并可以经过操作完成界面中的各种操作。在后台处理中，功能模块可能并没有用户界面。

通常菜单、子菜单、具有菜单属性的命令按钮都是系统功能控件。权限管理是指用户触发这个控件后，程序对用户是否有权限运行其后的功能进行的检查和比对，若允许，则程序继续，若不允许，则程序提示返回。

6）系统操作的定义。

系统操作是指在系统功能中为完成某一特定功能而编制的程序。系统操作涉及面很广，例如在一个信息录入功能模块之中会出现新增、保存、删除、退出等操作控件，当一个用户进入功能模块后，很可能只有增加权限，而没有删除权限。又例如，在一个查询功能模块之中，用户只能查询到本单位数据，但不能查到其他单位的数据。

用户是否可以操作或操作程度如何也是权限管理的重要内容。

7）角色的定义。

角色是指具体若干功能或操作的权限用户的别称。用户通过拥有角色而拥有角色所定义的若干功能或操作的权限。设计角色主要是方便用户权限的设置。

需要特别指出的是，角色并不单指一个系统内若干功能和操作的集合。角色也可以指多系统的若干功能和操作的集合。

8）权限设置的定义。

权限设置是指建立用户与系统的功能和操作之间的关系。权限设置可以提供用户与功能和操作的直接关系，也可以提供建立用户与角色之间的对应关系，以获取角色中的各种功能和操作的权限。

9）权限比对的定义。

权限比对是指当前用户在当前功能或操作状态下，与预先定义的用户功能和操作权限的比对，返回值，有或无权限。若有权限，则程序继续，无则提示返回。

10）参数的定义。

参数定义是指为实现权限管理的功能，而适应权限管理变化的参数，以及相关库表（字典）的维护功能。

例如：系统代码表、功能代码表、操作代码表、数据库用户名、密码、权限管理模式参数等都属于参数定义范畴。而为了实现权限管理而与软件架构的有关参数也归属其中。

11）权限管理模式定义。

权限管理模式是指企业或系统采用权限管理的方式或方法。一般有系统管理模式、授权管理模式、集中管理模式。

系统管理模式。所有的用户权限由这个系统的系统管理员来设置。

授权管理模式。系统管理员可以设置用户权限，但是得到授权的用户，可以为其他用户设置权限。

集中管理模式。企业对所属各种应用系统的用户权限采取集中管理，各个系统的用户权限设置由一个部门集中统一完成。集中管理是系统管理或授权管理的高级阶段，它可以有效地规范企业用户的各种角色，方便用户权限的设置，实现单一用户和实名用户的管理，方便用户操作，降低和防范操作风险。

例如，一个用户要在系统 A、系统 B、系统 C 中增加相应的权限。

在系统管理模式下，系统 A、系统 B、系统 C 的系统管理员要分别为用户设置功能权限。

在集中管理模式下，只要进入一次权限管理功能，就能为用户设置系统 A、系统 B、系统 C 的权限了。如果用户的这些权限正好是一个多系统

的角色，那只要将此用户赋予这个角色就行了。操作非常方便。

通过以上我们可以看到，对权限管理的需求分析和设计是按照 EOM 理论，采用抽象的方式进行的。大家可以拿具体的系统、具体的用户权限来验证这个需求是否完整，如果这个需求不完整、不全面，我们可以完善上述的定义，以使之更加科学更加合理。这样我们不但可以开发出一个通用的权限管理软件，而且建立了与之相适应的理论，这些理论将成为 EOM 一部分。而 EOM 提倡的正是理论和实践的结合。光有理论不行，光有实践也不行，只有理论和实践的结合，才能发挥理论的价值和实践的价值。

4.13 用 EOM 的眼光评判“我要做全国最好的标准权限组件和通用权限管理软件”3

上一节主要介绍了需求的问题，本节将介绍功能设计和数据库设计的问题。

4.13.1 功能设计

有了需求分析之后，我们就可以进行功能设计了，我们可以看看一般的权限管理和 EOM 权限管理的相同之处和不同之处。在大的功能方面两者没有什么不同，但是在功能适用范围和细节上那就有不同之处了。

1. 一般的权限管理

一般的权限管理大同小异，主要有权限设置、权限比对、参数设置三大功能。权限设置的变化比较多一些，例如，有角色的或无角色的，用户名直接输入的或通过单位部门进行选择的等。单一系统的权限管理和多系统的权限管理之间的差别在于后者增加了系统代码。由于增加了系统代码，所以在权限设置中就有了系统代码的选项，权限比对中就有了系统代码的入口参数，参数设置中就有了系统代码的创建和维护。

而且这些功能因程序员而异，可以说十个系统的权限管理就有十个不同的功能，其功能细节上各不相同，但是大的功能方面还是一样的。

2. EOM 的权限管理

主要由于其定位于通用程序，无论是十个系统还是 1000 个系统，均采用相同的权限管理系统或模块，并且具有相同的功能。EOM 当然具有一般权限管理的功能，但是 EOM 站在企业经营管理大局之上，更注重权限管理的参数化管理、企业内部系统关系管理，以及企业与企业之间或企业内部和外部之间的用户权限的管理，以适应各种权限管理的要求。

（1）权限设置

1）用户功能权限设置。建立用户与系统功能代码之间的关系，注意，此间增加了系统代码，体现了可实现多系统权限管理的功能。

2）用户操作权限设置。建立用户与系统操作代码之间的关系，注意，此间增加了系统代码、功能代码，体现了多系统的功能。

3）用户角色设置。建立用户与角色之间的关系。

4）角色的创建。建立角色和功能之间、角色和操作之间的对应关系。

（2）权限比对

1）权限比对相比一般权限比对增加了操作代码、系统代码、企业代码等入口参数，使得权限比对的范围最大化，并更具有灵活性。

例如，当比对类型参数为 1 时，比对只比用户名和功能代码，当参数为 2 时，则比对系统代码、用户名和功能代码。当参数为 3 时，就要比对企业代码、系统代码、用户名、功能代码了。

2）当出现特殊权限需求的时候，权限比对则可以通过特殊权限类型来确定以何种方式调用权限管理的接口程序。

例如，有一种特殊的权限要求，如果用户年龄大于 50 岁，则不可以进行某项查询操作。像用户年龄大于 50 的权限条件是非常特殊的权限要求。EOM 不可能满足，但是它可以提供各种接口程序来满足这些要求。

假定检查用户年龄大于50岁的函数是以WebService方式实现。则权限比对首先会去查找接口表，看本操作是否存在特殊权限，如果有则查看其处理方式，当发现是WebService方式后，就会用WebService名字调用这个函数。

（3）参数设置

1）功能代码的创建和维护。
2）操作代码的创建和维护。
3）系统代码的创建和维护。
4）特殊权限定义的创建和维护。
5）权限参数维护

（4）外部用户权限管理

当企业有外部用户访问的时候，必须建立外部用户权限管理。

1）外部用户的角色管理。

2）外部企业代码创建和维护。

（5）日志管理

所有重要的用户访问，包括允许或不允许的访问和操作都要进行日志记录，以便对用户进行操作监管。

EOM不但能满足各种权限管理的需求，而且其通过EOM权限管理来规范各种应用系统的与权限相关的功能，解决了现有权限管理功能的无序设计，实现了标准权限管理功能的共享。

4.13.2 数据库设计

在有了权限管理的定位和功能设计之后，最重要的是建立满足功能的数据库设计，这个设计不但要反映权限管理定位、功能的要求，更重要的是反映权限管理的各种要素的抽象，以及要素之间的关系。在数据库设计方面一般的权限管理和EOM权限管理存在较大的差异。

1. 一般的权限管理

其数据库设计相对比较简单，根据现实中的系统功能权限状况来设计功能表、用户功能表、角色表、角色功能表等，它与权限管理和实际系统相关比较直接，比如，功能代码，如果这个系统只有 3 个功能，其代码可能就会用一个字符，也就是说，这个数据库设计是为了这个系统的权限管理而设计的。

2. EOM 权限管理

EOM 的数据库设计则是站在整个企业信息化的高度来建立企业信息化体系的，所有的信息设计都要考虑到企业信息化所有系统的共享和标准化，并且根据 EOM 的七大要素采用的是抽象的方式对所有信息进行合理的分类，使得所有信息在整个企业信息化的大框架下形成有机整体。因此，EOM 权限管理并不从具体的系统、具体的功能上进行数据库设计，而是在抽象权限管理的要素基础上，标准化权限管理的各种信息，并且尽可能地利用其他的公共信息。

权限管理的主要信息有：

1）用户信息
2）功能信息
3）操作信息
4）角色信息
5）用户和功能关系信息
6）角色和功能关系信息
7）用户和操作关系信息
8）角色和操作关系信息
9）应用系统信息
10）外部用户信息
11）外部角色信息
12）外部用户角色关系信息

13）外部企业信息

14）特殊权限定义信息

15）权限管理标准函数信息

与权限管理相关联的信息：

1）部门信息

2）员工职位信息

这些信息对员工的权限影响很大，例如，员工只能看本部门的数据，一般员工看不见企业的利润和奖金发放情况等，管理者可以看到普通员工看不到的报表等。

我想指出的是，这里的各种信息都是建立在抽象要素基础上的，与实际内容没有关系，但是能满足各种实际的要求，而且每个信息的设计都要得到 EOM 理论的支持。

3. 两者设计比较

以功能信息设计为例进行比较。

（1）一般权限管理

在功能设计上会根据这个系统有多少个功能来设计，例如，这个系统最多有 30 项功能（主菜单和子菜单），那么可以这样设计功能信息表。

1）功能代码 char（2）（代码方式表示功能，最多可以定义 100 个功能，假定代码是数字方式，同下）。

2）功能名称 char（50）（说明功能名称）。

3）功能界面程序名 char（100）（用于调用这个功能程序）。

4）功能标志 char（1）（此功能是否有效）。

这个功能表的确能满足这个系统的权限管理的要求。

但是这个功能表：

1）并不能满足任意应用系统的要求。例如，如果一个应用系统有150个功能，则功能代码定义的长度就不够了。

2）不能反映多系统的要求，只能是单一系统。

3）只考虑了功能的个数，没有考虑功能的层级关系，一般来说一个系统的功能是分层级的（有主菜单和子菜单），用户往往不会是一个或零散的功能，而是具有一类或多类的功能，因此，在功能表设计的时候，要考虑功能的分类，以及分类后的各种功能的具体实现。

4）有的程序员在设计功能代码时会采用数据位方式来体现功能的层级关系。

例如：系统共有五个主菜单，第一个主菜单下有3个子菜单。假定功能层级为2。他可能会定义10为第一个主菜单，20为第二个主菜单，30、40、50分别为第3、4、5主菜单。11、12、13为第一个主菜单下面的3个子菜单。

这种靠数据位来区分功能分类的方式比较原始（来自于汇编中的位操作），并不能适应功能分类的灵活变化。例如，原来的12表示功能将要放在第2个主菜单下面，这种情况处理起来就比较麻烦了。

（2）EOM权限管理

在设计功能表时将会考虑：

1）任意应用系统中最大的功能个数。

2）反应系统与功能之间的关系，就是说功能表中要有系统代码的字段。

3）功能之间的分类以及关系。

4）功能的调整。

5）仅考虑与功能有关的信息，不考虑功能相关的信息，相关信息可以放在另外一张表中如下表所示。

6）功能信息和其他信息的一致性。例如，操作信息、角色信息、部门信息等，如下图所示。

7）其他因素。

系统功能表设计说明：

系统功能表					
Sysfunction					
序号	名称	字段名	说明	类型	备注
1	应用系统	appsys	应用系统	Char（3）	
2	系统功能编码	Id	系统功能编码	Char（9）	
3	系统功能编码 1	Code1	系统功能编码 1	Char（3）	系统功能大类或第一层次
4	系统功能编码 1 名称	Name1	系统功能编码 1 名称	Char（50）	
5	系统功能编码 2	Code2	系统功能编码 2	Char（3）	系统功能中类或第二层次
6	系统功能编码 2 名称	Name2	系统功能编码 2 名称	Char（50）	
7	系统功能编码 3	Code3	系统功能编码 3	Char（3）	系统功能小类或第三层次
8	系统功能编码 3 名称	Name3	系统功能编码 3 名称	Char（50）	
9	系统功能名称	Name	系统功能名称	Char（80）	系统功能名称，总称或简称
10	上级系统功能编码	Up_code1	上级系统功能编码	Char（9）	当为大类时
11	上上级系统功能编码	Up_code2	上上级系统功能编码	Char（9）	当为中类时
12	编码等级	Codelevel	编码等级	Char（1）	用于说明编码等级 0 最高层 1 1 层 2 2 层 3 3 层
13	标志	Flag	有效标志	Char（1）	有效标志 EX：N 无效 缺省空值 有效
14	备注	Remark	备注	Char（50）	

NRP 系统功能的实例，901 表示 NRP：

appsys	id	code1	name1	code2	name2	code3	name3	up_code1	up_code2	flag	name	codelevel	remark
901	001002000	001	报表发布	002	发布固定报表	000	发布固定报表	001000000	000000000	NULL	发布固定报表	2	NULL
901	001003000	001	报表发布	003	修改发布	000	修改发布	001000000	000000000	NULL	修改发布	2	NULL
901	001004000	001	报表发布	004	撤销发布	000	撤销发布	001000000	000000000	NULL	撤销发布	2	NULL
901	001005000	001	报表发布	005	设置固定报表...	000	设置固定报表...	001000000	000000000	NULL	设置固定报...	2	NULL
901	001006000	001	报表发布	006	检查个人未发...	000	检查个人未发...	001000000	000000000	NULL	检查个人未...	2	NULL
901	001007000	001	报表发布	007	检查部门未发...	000	检查部门未发...	001000000	000000000	1	检查部门未...	2	NULL
901	001008000	001	报表发布	008	我的报表箱	000	我的报表箱	001000000	000000000	1	我的报表箱	2	NULL
901	002000000	002	报表浏览	000	报表浏览	000	报表浏览	000000000	000000000	1	报表浏览	1	NULL
901	002001000	002	报表浏览	001	浏览个人报表	000	浏览个人报表	002000000	000000000	1	浏览个人报表	2	NULL
901	002002000	002	报表浏览	002	浏览部门报表	000	浏览部门报表	002000000	000000000	1	浏览部门报表	2	NULL
901	002003000	002	报表浏览	003	浏览全部报报	000	浏览全部报报	002000000	000000000	1	浏览全部报报	2	NULL
901	002004000	002	报表浏览	004	浏览我的报表箱	000	浏览我的报表箱	002002000	000000000	1	浏览我的报...	2	NULL
901	002005000	002	报表浏览	005	浏览外部部门...	000	浏览外部部门...	002005000	002000000	1	浏览外部部...	2	NULL
901	003000000	003	报表查询	000	报表查询	000	报表查询	000000000	000000000	NULL	报表查询	1	NULL
901	003001000	003	报表查询	001	报表查询	000	报表查询	003000000	000000000	1	报表查询	2	NULL
901	003002000	003	报表查询	002	外部报表查询	000	外部报表查询	003000000	000000000	1	外部报表查询	2	NULL
901	003003000	003	报表查询	003	查询归档报表	000	查询归档报表	003000000	000000000	1	查询归档报表	2	NULL
901	004000000	004	报表下载	000	报表下载	000	报表下载	000000000	000000000	NULL	报表下载	1	NULL
901	004001000	004	报表下载	001	下载临时报表	000	下载临时报表	004000000	000000000	1	下载临时报表	2	NULL
901	004002000	004	报表下载	002	下载固定报表	000	下载固定报表	004000000	000000000	1	下载固定报表	2	NULL
901	005000000	005	报表清理	000	报表清理	000	报表清理	000000000	000000000	NULL	报表清理	1	NULL
901	005001000	005	报表清理	001	清理临时报表	000	清理临时报表	005000000	000000000	1	清理临时报表	2	NULL
901	005002000	005	报表清理	002	清理固定报表	000	清理固定报表	005000000	000000000	1	清理固定报表	2	NULL
901	005003000	005	报表整理	003	建立归档报表	000	建立归档报表	005000000	000000000	1	建立归档报表	2	NULL
901	006000000	006	统计数...	000	统计数据装入	000	统计数据装入	000000000	000000000	NULL	统计数据装入	1	NULL
901	006001000	006	统计数...	001	装入EXCEL	000	装入EXCEL	006000000	000000000	NULL	装入EXCEL	2	NULL

4.14 用 EOM 的眼光评判“我要做全国最好的标准权限组件和通用权限管理软件”4

本节主要从软件架构设计的角度来阐述。

软件架构设计

有了相同的功能设计、相同的数据库设计，不同的程序员依然能开发出不同的软件。而这些不同绝大多数是在程序上的不同，例如，命名不同、使用语句不同、算法不同、三层架构不同，可以说是相差无几，难怪程序员之间相互都认为别人跟自己一样呢。其实当程序员水平上升到一定程度就会考虑软件架构问题了。这个架构比程序本身的技术实现要重要得多。

1. 一般的权限管理

一般的权限管理并不关心程序的架构，无论是单一系统的权限管理还是多系统权限管理，他们根据功能设计和数据库设计将程序编出来就行

了。最多考虑一下采用 B/S 还是 C/S，或者考虑一下采用 C#还是 JAVA。稍好的程序员最多在程序内部考虑一下采用三层架构（界面层、逻辑层、数据层）。

他们不关心架构的主要原因包括：

1）不清楚什么是架构；

2）软件定位于单个系统或多个系统，不考虑架构也能实现软件功能；

3）自身编程水平还没有到达这个层面。

2. EOM 权限管理

EOM 则根据其定位，站在企业信息化整体的高度，所有应用系统都必须考虑软件的架构，否则无法实现其定位目标。就权限管理而言，由于权限管理定位于多企业、多系统，则必然考虑到多企业、多系统的现状（见图 4-1 ~ 图 4-6）：

1）权限管理有两个版本：B/S 和 C/S；

2）开发语言：C#、JAVA 或其他语言；

3）B/S 和 C/S 共用相同的公共核心库和权限管理库，这些公共核心库和权限管理库都是标准权限管理模块，可供上层进行调用；

4）EOM 提供两种权限管理方式，一个是系统方式，即完整的权限管理系统，另一个是标准模块方式，即可以向用户提供标准的权限管理功能，用户可以根据这些功能结合自己的界面形成自己的权限管理系统；

5）权限管理即可以在企业内网运行，也可以在互联网运行；

6）程序实现界面和处理的分离，所有处理都放在公共核心库和权限管理库之中，可以通过两种形式即 DLL 或 WebService 来调用；

7）程序采用参数多形式方式实现参数化：数据库、文本、XML 等；

8）通过参数化和接口方式实现特殊权限管理的需求。

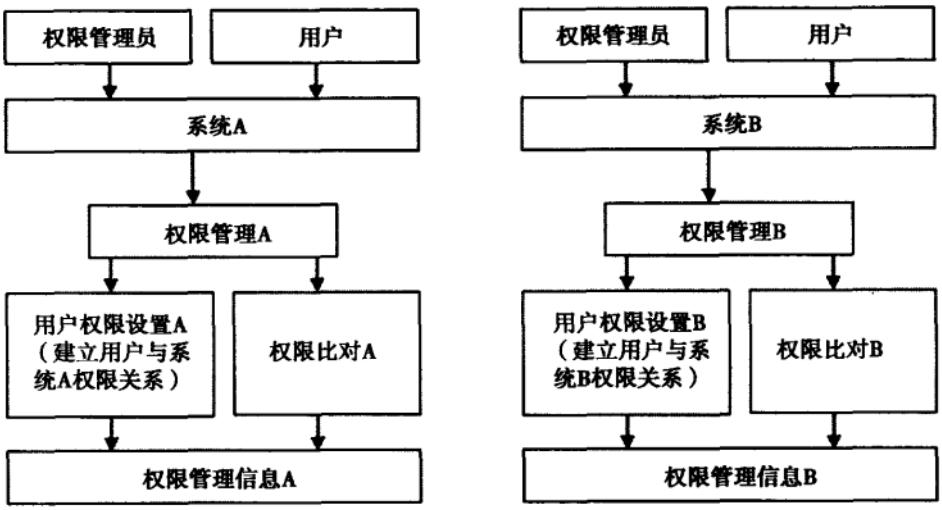


图 4-1 现行企业权限管理示意图

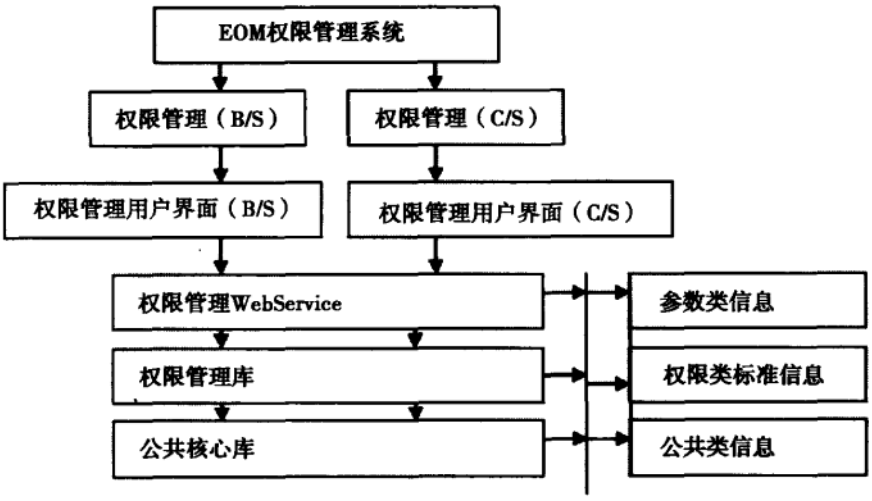


图 4-2 EOM 权限管理系统架构图

权限比对函数（部分程序）。此程序仅供参考，仅反映了软件的架构。用户可以在界面一开始的时候进行功能权限的比对，例如，在 901 系统中的 003001001 功能进行权限比对。

```
if(Check_authority("901","003001001","")==0) //0 表示无权限,1 表示有
权限;
return 0;
```

用户可以在界面中的某一操作时候进行比对，例如，在 901 系统中的 003001001 功能中进行 001001001 操作的比对。

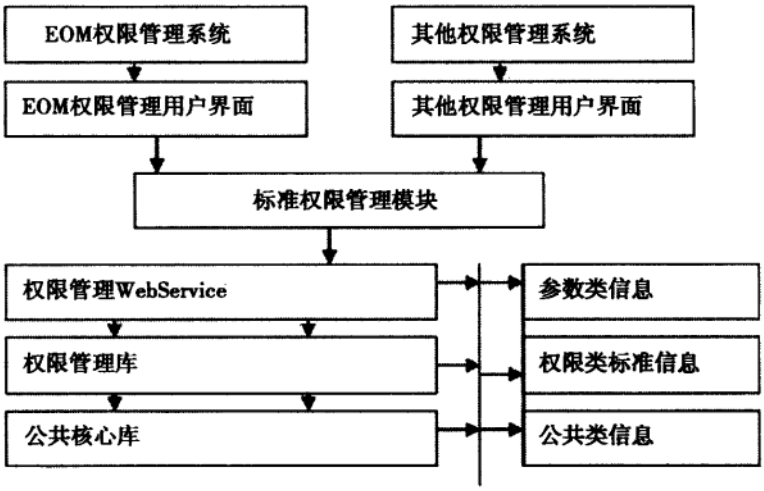


图 4-3 EOM 的权限管理系统和标准权限管理模块

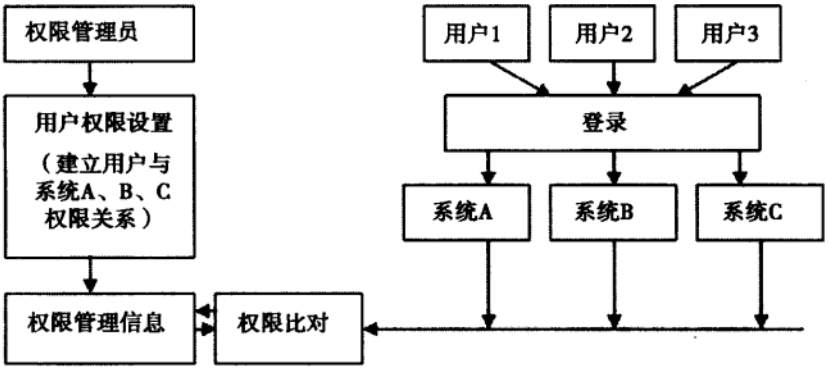


图 4-4 企业内部多系统状况下的 EOM 权限管理示意图

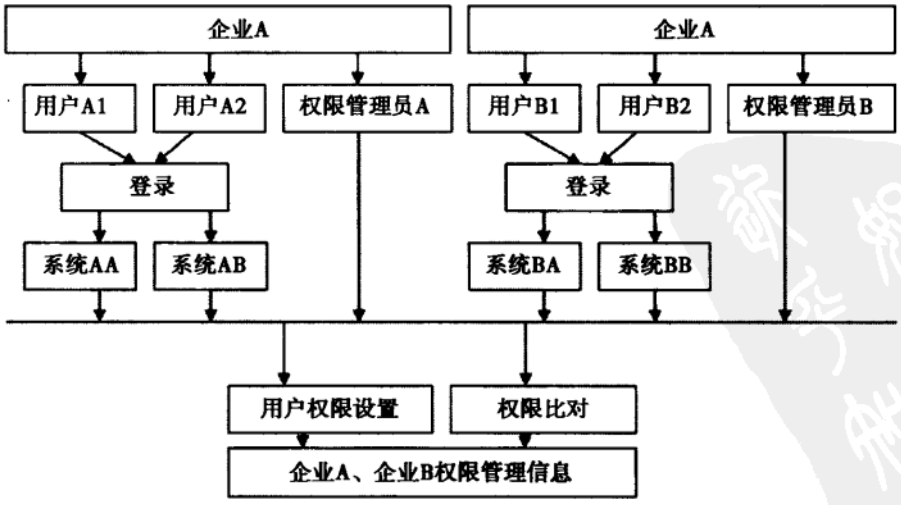


图 4-5 多企业 EOM 权限管理示意图

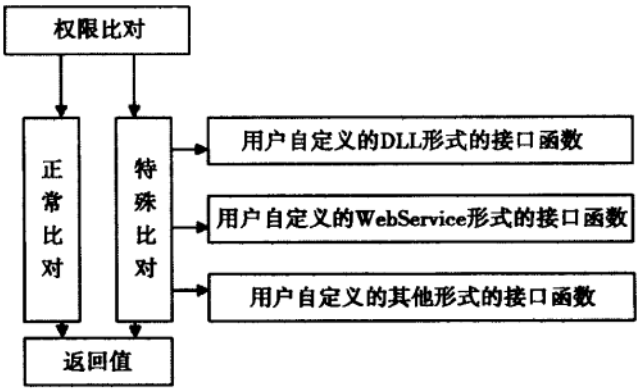


图 4-6 EOM 特殊的权限管理处理示意图

```
if(Check_authority("901","003001001","001001001") = =0) //0 表示无权限,1 表
示有权限;
return 0;
private void check_authority(String appsys,String fucntion_code,String op_
code)
{
    int s =0;
    string usr_name =get_usr_name();
    string authority_mode = eomlib.get_sysparameter("authority_mode");//取
系统参数
    string aurthority_dll = eomlib.get_sysparameter("authority_dll");//取
系统参数,参数中含 DLL 名和函数名,中间用逗号分开。
     string authority_url = eomlib.get_sysparameter("get_authority_
url");//取系统参数,参数中含 URL 名和函数名,中间用逗号分开。
     switch(authority_mode)
     {
         case "eom":
            s =eomlib.check_authority(usr_name,appsys_code,function_code,
op_code);
           break;
         case "url":
s =iasglib.call_webservice(authority_url,usr_name,appsys_code,function_
code,op_code);//调用外部 webservice
             break;
         case "dll":
s =iasglib.call_dll(authority_dll,usr_name,appsys_code,function_code,op_
code);//调用外部 DLL
             break;
         default:
             break;
     }
     return s;
}
private string get_usr_name()
{
return publicdata.usr_name;//用户名在用户登录时保存在公共变量 publicdata 中
```

```
}

//---------------此处为权限管理的 WebService 函数 --
[WebMethod(EnableSession = true)]
private int check_authority(string usr_name, string appsys_code, string
function_code,string op_code)
{
return eomlib.eom_check_authority(usr_name,appsys_code,fucntion_code,op_
code);
}

//--eomlib ---此处为 EOM 公共库函数,其包含了权限管理的功能 ----------------
private int eom_check_authority(string usr_name,string appsys_code,string
function_code,string op_code)
{
int s;
  if(op_code.Trim().Length = =0)//当操作代码为空的时候,表示只检查功能代码,反之
检查操作代码
    s = check_function_authority(usr_name,appsys_code,function_code);
  else
       s = check_opertion_authority(usr_name,appsys_code,op_code);
       return s;
}

private int check_function_authority(string usr_name,string appsys_code,
string function_code)
{
int s =0;
string role_mode = eom.get_sysparameter("role_mode");
    if(role_mode = = "Y")
    s = check_authority_with_role(usr_name,appsys_code,fucntion_code);
    else
s = check_authority_with_function(usr_name,appsys_code,function_code);
    return s;
}

private int check_authority_with_role(string usr_name,string appsys_code,
string fucntion_code)
{
int s =0;
  string role_name = get_role_name_with_usr_name(usr_name);//从用户角色表中
查询出角色
  string[] fucntion_code_tab = get_all_function_with_role_name(role_
name);//按照角色将所有的功能代码都取出来
    foreach(string function_code in fucntion_code_tab)
    {
    s = check_function_with_usr_name(usr_name,appsys_code,fucntion_code_
tab);//对用户功能表按照系统、功能、用户三个字段查询,查到且权限标志为 1 则返回 1,否则返
回 0
       if(s = =1)
       return s;
    }
```

```
        return s;
    }
```

综合所示，通过对权限管理的一个典型的程序剖析，大家可以看到采用 EOM 构建一个程序的基本流程。这个流程是：第一，关注权限管理在企业经营管理中的地位和作用；第二，给权限管理一个多企业、多系统的定位；第三，给出权限管理的功能设计；第四，给出权限管理的数据库设计，这个数据库设计是站在 EOM 高度对权限管理的各元素的抽象，并共享其他相关的标准信息；第五，为了实现权限管理的定位，对软件进行架构设计，这个架构可以实现多企业、多系统的权限管理的要求，也满足单企业、多系统，既满足互联网也可以满足局域网，既可以满足 B/S 也可以满足 C/S 的各种选择；第六，按照以上五个部分用程序实现就行了（程序实现并没有太多的难度）。

EOM 是一个全新的企业信息化的思路，它拓宽了程序员的视野，它所强调的企业经营理念、程序的定位、程序的抽象数据库设计、程序的软件架构都将给程序员带来更多、更深入的思考。

4.15 程序员的春天：EOM 与程序员

2009 年 12 月我开始撰写 EOM 与程序员的随笔。当我注意到程序员收入下降，程序员水平下降，程序员缺乏理想，程序员队伍涣散的时候，我感到非常心痛。要知道程序员是我一个向往并且从事了几十年的职业呀！当我写到什么是 EOM，写到用 EOM 眼光看待一个编程实例（权限管理）的时候，我开始感觉到那种暖意，就如同春天到了，春风吹了，原来枯萎的树枝冒出了新芽，油菜花开了，桃花开了，梨花开了，杜鹃花开了，一个五彩缤纷的世界来到了。那么 EOM 到底会给程序员带来什么样的思考呢？

1. 企业信息化的不同的解决方案

对于许多程序员来说，他们每天为各种企业开发不同的程序，例如，银行的核心业务系统、信贷系统、信用卡系统，医院的 HIS 系统，商业的物流系统、ERP，服务业的 CRM，通用软件的财务系统、报表制作系统、

门户软件，工具的文件传输、文件加密、文件压缩、图像处理、用户控件等。这些系统都应该归属于企业信息化范畴，因此程序员和企业信息化密切相关，没有企业信息化可能软件业也没有现在这么大的发展空间，也不会催生出如此众多的程序员。

EOM 作为企业经营模型，也是为企业信息化服务的。从 EOM 也可以推导出企业信息化的各种软件，这些软件涵盖了现有各种软件的功能。从这个意义上来说，程序员现在做的和 EOM 要做的都是相同的东西，只是 EOM 要做的东西和使用的方法与现在有差异，程序员通过了解 EOM 可以看到企业信息化的不同的解决方案，从而对现在从事的开发工作有一个重新的思考和认识。

2. 软件价值

对于大多数程序员从事的软件开发工作，绝大多数都是属于企业“需求驱动”、“用户定制”的软件，少有的通用软件也是针对性非常强的，大部分只针对企业信息化的某个方面，很少从整体上考虑这个通用软件和其他应用系统的关系。这种软件制作模式导致软件价值低、没有通用性、软件制作时间短、软件质量差、程序员收入低等恶性循环。程序员往往面对这种现状时，看不到未来的希望。

EOM 提倡的是科学的企业信息化，提倡是的 EOM 理论指导下的信息化，提倡的是制作符合企业经营发展的各种通用软件，通过通用软件来提升企业经营管理水平，而不是通过软件来迎合企业落后的经营管理方式。企业间的差异和变化可以通过参数化和参数接口方式来满足。因此，程序员需要有制作通用软件的理念，需要有制作通用软件的技术水平，需要有制作通用软件质量的意识。只有这样，软件的价值才能成百上千倍的提高，程序员的收入才能大幅度提高，程序员的技术水平才能大大提高。

3. 项目来源

许多软件企业和程序员被动等待着企业的开发需求，不断地寻求项

目，由于竞争激烈，软件价格越来越低，造成了“没有项目等死，有了项目找死”的无奈状况。其中，由于用户要求开发时间越来越短，程序员编程“只求有无，不求更好”，根本没有时间研究算法和技巧、研究新技术新工具、研究软件架构、研究软件共享。这导致了编程熟练程度越来越高，编程水平越来越低。

EOM 则与之相反，企业信息化中的各种项目都是事先按照 EOM 理论规划好的，并不是由企业提出，企业只是使用 EOM 中的各种软件。因此，相对于软件公司和程序员来说，他们不需要找关系托人情去拿项目了，他们可以主动地在 EOM 中寻找适合自己的开发项目。因此，EOM 解决了企业“要做什么”的问题，也解决了软件公司和程序员的“要做什么”的问题。由于 EOM 是企业信息化整体的解决方案，因此其中的各种平台、通用软件、专用软件需要最好的算法和技巧，需要最合适的新技术、新工具，需要研究软件架构、需要研究软件共享。所以程序员必须提高自身的技术水平和素质，向优秀程序员发展，这样才能制作出“全国最好”的软件。

4. 参与 EOM

现实中，许多程序员抱怨自己无法左右自己开发的项目，公司叫做什么就做什么，也不知道什么软件是好软件，是值得开发的软件，有的程序员因常年加班加点身心疲惫，也没有什么时间做自己想做的事。

EOM 的出现至少给很多有市场意识、有理想的程序员一个机会，他们可以利用业余时间进行研究、宣传、设计、开发、销售、维护 EOM 的各种产品。我们可以建立网上 EOM 团队，以团队方式吸引有思想、有才华的各类人才，开发这个产品，经营这个产品，实现自我的价值。

5. 可行性

也许很多人会问 EOM 是否只是一个理论？是一种空想？只是说说而已的“传说”？我的答案是否定的，因为 EOM 并不是心血来潮，一时冲动形成的。EOM 可以说是伴随着中国企业信息化成长的一个必然结果。

正是由于长期关注企业信息化发展，研究其中的成功之经验、失败之教训、发展之规律，才最终形成 EOM（前面的章节中曾提到这个过程），这是其一。EOM 产生是伴随着自己的软件技术不断进步的，是有大量应用系统制作支持的，这是其二。EOM 已经提出有二三年了，在此期间我们一直在研究其理论的可行性，研究其产品可行性，并且找到了 EOM 产品的软件新架构，而且正根据这个新架构制作 EOM 的产品。我们想通过这些产品来验证 EOM 的可行性及其市场价值，这是其三。EOM 项下估计大约要有数百个平台系统、通用系统和专用系统。这些系统将涵盖企业信息化的主要方面，通过宣传 EOM 和大家参与，必定有其中的 EOM 产品被制作，其市场价值和可行性也是一个说明，这是其四。随着 EOM 得到人们越来越多地关注，EOM 的研究和发展，其可行性就会越来越明显。

6. 关注 EOM

其实，你可以不去参与 EOM 的研究和开发。但是你可以关注 EOM 的发展，了解 EOM 是如何看待企业信息化现状的，是如何分析这个现状的；你可以了解 EOM 所制作的软件和现有的软件有什么根本性的不同；你可以了解 EOM 制作软件的流程和思路，看看和现有软件制作有什么不同；你可以了解一个 EOM 产品的市场价值和一个定制软件市场价值的差距；你可以感受有 EOM 为标准来评判软件的好坏和无标准以自己的标准来评判的异同；你可以了解 EOM 的抽象的方法和现在“眼见为实”、“需求驱动”的不同之处；你可以了解 EOM 提倡的团队精神、团队方式以及与自己目前项目小组之间的差异；你可以试着用 EOM 理念重新将自己手头开发的程序抽象成为标准功能和标准信息等。我想这些思考对每一个想进步的程序员是有百利而无一害的。

程序员与 EOM 有着密切的关系，随着 EOM 逐渐被人了解和认识，EOM 产品逐步推出，国家对这个民族的企业信息化的理念和产品的支持，我想越来越多的程序员将会加入到 EOM 这个队伍中来，随着 EOM 队伍的壮大，EOM 在促进我国企业信息化发展方面将起着越来越重要的作用。我敢说，对 EOM 关注和了解可以扩大程序员的视野，就如人们因

春天到来而踏青，在春天里感受着春风，感受着春花烂漫，感受着生命的多彩……

4.16 优秀程序员应该具备哪些素质

中国人常常喜欢用十来表现事物，例如“十大关系”、“十大明星”、“十大新闻”、“十大技术”等。这可能与十全十美有关，也可能便于记忆吧（整数容易记忆）。作为优秀程序员应该具备的素质，我也提出了一个十大素质。

1. 心怀理想

我认为优秀程序员首先具备的素质是心怀理想。我曾说过，许多程序员都是冲着“比尔·盖茨”而来的，是被他的巨大成功所感召的。可以说不想当比尔·盖茨的程序员不是好程序员。大家要记住，盖茨的成功是两个方面的成功，一个是他个人的成功，另一个是IT行业的成功。作为个人成功，只要个人能在众人中脱颖而出，应该说行行都可以出状元的。而行业的成功则表现出行业的生命力和朝阳性。而从事这个行业的人则能够伴随着这个行业的发展而成功。在IT软件行业中，一个好的软件通过无数复制使用能产生巨大的经济效益和社会效益，而好的软件往往是程序员所编制的，与其有很大的关联性，这就意味着程序员在这个行业存在着成功的必然可能。现实中除了比尔·盖茨，还有许许多多中外IT名人都是程序员出身的，他们的成功昭示着程序员成功的可能。

既然程序员有成功的可能，那优秀程序员一定心怀成功的理想，这种理想激励他们不断进取走向成功。我这里说心怀，是说程序员不必天天把理想挂在嘴边，而应该把理想放在心里，时刻牢记自己的目标是什么！目标是成功的一半，有了目标，优秀程序员可以不断地调整自己，以最优的方式去接近和实现自己的目标。现实中还有两类程序员，一类是也想成为比尔·盖茨，但始终是想，最终只是一种幻想；另一类知道比尔·盖茨，但是望而却步，从不敢有半点奢望，做到哪就算到哪，顺其自然。这两类程序员都不具备优秀程序员的基本素质。

2. 充满激情

优秀的程序员一定是一个充满激情的人，他们的理想和目标的远大注定了他们不能以常人的方式平淡地、机械地工作。我们很难想象一个优秀程序员不会争做一个大的项目；我们很难想象一个优秀程序员在大家遇到困难时，无助待援时，不挺身而出；我们很难想象一个优秀程序员历经千辛万苦编制出来的程序被用户认可那一瞬间，眼泪不夺眶而出；我们很难想象一个优秀程序员在自己遇到困难的时候，不大声叫喊永不放弃。对于优秀程序员来说付出和收获都是充满激情的。而他们的激情不但感染了自己，而且感染了团队，感染了用户。

3. 市场意识

市场意识是许多程序员最不具备的，有的程序员甚至反感市场意识，认为软件卖不卖钱是公司的事，自己只要把程序做好了就行了。借用一句“只拉车不看路”来形容这类程序员是最为贴切的。有些人从事编程的确是个人兴趣所致，他们喜欢在编程中找到自身的乐趣。但是，更多的人从事编程是为了谋生，是为了获取更多的收入。程序员树立市场意识，就是要做有市场价值的软件，做价值最大化的软件。虽然程序员在单位可能无法挑选开发的项目，但是程序员可以在有市场价值的软件开发上投入更大的精力，可以在业余时间开发有价值的软件。如果公司的软件市场没有前景，程序员应该未雨绸缪，做好最坏的打算，选择更有成长性的软件公司。否则，编程能力再强也是英雄无用武之地，达不到自己的预期。做市场价值最大的软件，让自己市场价值最大化应该是优秀程序员必备的素质之一。

4. 永不放弃

程序员可能和其他行业从业人员不太一样，一般遇到困难，其他人员可能进行一番战斗后，无法克服就放弃了。而程序员遇到困难就可能让程序无法运行下去，他们必须解决了问题程序才能完成。优秀的程序员不但敢于解决各种编程中的问题，而且还会善于解决各种问题。有的程序员遇

到问题，随便找一种方法就给解决了，他可能不问解决后会带来什么新的问题，不问是否有更好的解决方案。而优秀的程序员往往追求最优的解决方案，往往在这个过程中遇到千难万难，但是他们抱着永不放弃的信念，日复一日，月复一月，甚至年复一年去实现自己的目标，最终他们能收获其他程序员无法收获的胜利成果。

5. 团队精神

现代的程序员和以前的程序员已经完全不一样了，原来程序员是全能型程序员，现在程序员是专业的程序员。尽管还有很少的程序员集设计、开发、销售、维护为一身，但是专业化的潮流不可阻挡，这就表示一个软件或多个软件是靠多个人开发的，他们通过分工协作从事着整个制作工作。程序员编写的程序不仅要给自己看，而且要给团队其他人看。当发现程序问题的时候，团队成员可以一起帮助分析解决。如果没有团队精神，任何一个程序员也不可能有真正的成功，即使成功也是渺小的、暂时的。因此要获得事业上的成功，必须依靠团队的力量才能达到。优秀程序员的优秀之处可能并不只是关心自己的优秀，而是更关心他周围的程序员和其他人员是否优秀。让更多优秀的人在一个团队中形成一个优秀的团队，这样的程序员才是充满竞争力的，是战无不胜的。

6. 面向对象

很多人都是从面向过程思维方式开始学习编程的，随后也学了面向对象编程方法，也用了面向对象开发工具和语言，但是真正的用面向对象方法来编程、设计编程架构、进行功能分析、需求分析还是很少的。优秀的程序员一定具备面向对象的思维方式，从设计到编程都贯彻着面向对象的痕迹，他们把面向对象作为思维的日常化。具有面向对象思维方式的人可以对复杂混乱的事物进行抽象，具有很强的分类能力，化繁为简，程序更加模块化，程序共享度得到提高，具有大程序、超大程序开发能力。有关面向对象的问题，我在前面的文章中已经阐述很多了，这里就不再重复了。

7. 基础扎实

要想自己编程能力得到持续提高，我们就必须注重自己的基础建设，让自己提高的能力都建立在每一层的坚实基础之上，即使能力再高，也不会被风吹得摇晃，尽显强者的刚毅不摧。我们不要小看已经学过的操作系统原理、编译原理、程序设计原理、数据库设计、数据结构设计甚至计算机英语。这些基础是否扎实直接影响到以后的各种应用技术的掌握和掌握的程度。而大多数新的技术都是在原有技术的发展，因此掌握好原有的技术对最快掌握新技术是有很大帮助的。此外，扎实的基础也使程序员把精力放在创新和解决疑难问题上，这样优秀程序员的处理能力将会大大增强。

8. 条理清楚

普通的程序员则不太注重条理问题，他们对可并行处理的语句从来不进行次序的考虑；对语句的使用没有一致性，显得很随意；对处理流程往往一气呵成，一个函数能有上百上千条语句。他们最大的特点就是从来不多问一个为什么！而优秀程序员更注重程序条理，每个语句、语句的先后、语句的行数、函数命名等，他们都能说出为什么这样做。这种条理清楚的素质使得程序员能够把控程序的制作时间进度，也有利于对大程序的任务分隔。大凡程序员编写的程序能被其他人看懂的，基本上都是条理很清楚的。程序有条理，说明程序员的思维有逻辑性，优秀程序员是最讲究逻辑的。一个优秀的程序员能把一个庞大复杂的程序说的有条有理，并且在程序上也显得可读性极强。条理清楚的表现形式之一就是程序函数多，函数中的语句少。从这个意义上来说，优秀程序员也是优秀的设计师、项目经理。因为他们的逻辑是一脉相承的。

9. 全面细致

我这里说的全面有五个方面的含义：第一，程序员在编程过程中要全面细致，不要有遗漏，而这些遗漏往往造成编译错误，需要花费大量不必要的时间去查改。第二，程序员在设计过程中要全面细致，不要在功能设

计方面有遗漏，尤其不要忘记对例外情况的处理，许多功能上运行出错都是这种原因造成的。第三，程序员的编程技术要全面，尽可能多掌握各种编程技术，例如，程序员不但要掌握前台开发技术，也要掌握后台开发技术，既要会文件处理，也要会数据库处理，既要会编写程序也要会制作安装盘等。第四，程序员的业务知识要全面，程序员不但要掌握编程技术，而且要主动学习和掌握用户的业务知识，通过对业务知识的掌握，才能更好地理解功能的意义，有助于编写更好使用的程序。第五，程序员的知识面要广。优秀的程序员不但要掌握编程知识和用户业务知识，而且要有意识地拓宽自己的知识面，多关心新闻、时事、时尚、经济、技术、文艺、文学等各种知识领域，在知识的海洋中去畅游工作的轨迹，你就会有万事相同的感觉，这些知识会潜移默化地影响你编程的风格和效率。

10. 求新求精

IT 发展之迅速、技术更新之快捷，使得 IT 成为公认的追逐潮流的行业，因循守旧，固步自封，甚至年龄的增长都会导致程序员被这个行业所淘汰。有时候叹息行业无情也是情理之中。面对潮流的挑战，优秀程序员应该是无所畏惧的，他们应该始终怀有一种追求新技术的愿望，有一种追赶潮流、伴随潮流的勇气和胆识。因为这个职业就注定了这样的工作方式。当然，在新技术满天飞的情况下，我们不能也不可能有什么新技术就去学什么新技术，我们可以花很少的时间去浏览和关注各种技术的发展状况，但是对于自己所要掌握的新技术则要不惜代价，花大力气去学习和掌握它。掌握新技术的价值评判标准是用于软件的开发，用于更好地解决以往程序中存在不可解决的问题，用于所开发软件的销售价值。优秀的程序员会很好地选择新技术，并利用新技术立即去革新自己的程序，并使程序投向市场产生价值。那种以学习、知识储存的态度对待新技术都是不成熟的程序员。

我们常以新为炫耀，但是我们更以精为实力。优秀的程序员不但要求新，更要求精，使得自己在编程的某个方面成为专家和高手，让普通的程序员敬仰。例如，精通客户的业务知识、精通程序架构、精通某种语言、精通数据库、精通模块化设计、精通三层结构、精通界面设计、精通接口

程序、精通安装盘制作等。程序员可以精通的领域和内容非常多，关键是程序员对自己的要求。很多程序员满足于会，而不追求于精，使得自己变成什么都懂，什么都不精的，任何人都能取代的普通程序员。这样的程序员怎么可能获得更高的报酬，又怎么可能变成优秀呢？

当然优秀程序员还会有其他的良好的素质，在这里我很难穷尽，我也希望读者能在其后续上更多更好的素质。其目的只有一个：我们了解优秀，我们追求优秀，我们一定优秀。

第五部分
Part 5

附　录

5.1　创新模型简介

5.1.1　EOM

EOM（Enterprise Operating Model）即企业经营模型，它是指从企业经营的全局出发，将企业经营抽象为资源、客户、员工、产品（服务）、业务、财务、管理等七大要素，并通过七大要素之间的相互关系，最终建立企业经营模型。这个模型客观、系统、科学地涵盖了企业经营的整个过程，反映了企业经营的本质特征（见图5-1）。

该模型立足于企业经营这个企业信息化的起点和终点，对于企业信息化具有非常大的影响。当前企业信息化都是以具体需求为驱动的，而不是站在整个企业经营的层次上分析问题，这必然造成企业信息化的无序发展，必然造成企业内部系统建设的重复，必然造成信息共享困难。信息化发展到一定程度后，就必然会对企业经营发展产生负面的影响。利用企业经营模型（EOM）来构建企业经营模型系统（EOMS），使得各个企业有可能结束以“需求驱动”方式开展信息化的局面。使得企业信息化从无理论转为有理论，无序转为有序，无规划转为有规划，不全面转为全面，不通用转为通用，浪费转为不浪费，不共享到共享。业务发展水平和管理水平不一的企业就有可能通过EOM的应用系统享受到最好的业务和最好的管理，真正享受到企业信息化对企业经营发展的促进作用。

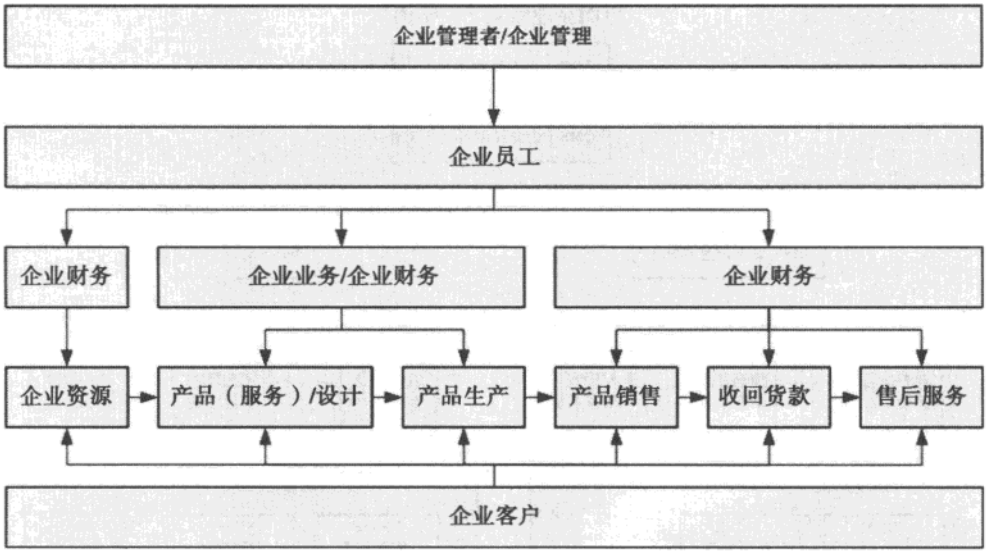


图 5-1 EOM 结构图

程序员可以通过 EOM 了解到企业信息化应该做什么，知道自己现在编写的程序用在什么地方。

EOM 是作者于 2008 年提出的。

5.1.2 EOMS

EOMS（Enterprise Operating Model System）即企业经营模型系统，是基于 EOM 的企业信息化的整体架构。根据资源、客户、员工、产品（或服务）、业务、财务、管理这七个基本要素之间的相互作用和联系，可以将 EOMS 进一步分成业务信息系统（BIS）和管理信息系统（MIS）两个基本架构。EOMS 按照 EOM 的七大要素以及相互关系，采用了平台和通用系统以及专用系统相结合的架构形式。这解决了企业信息化中的共性和个性问题，解决了企业信息化中“做什么”这个最重要的问题（见图 5-2）。

EOMS 是作者于 2008 年提出的。

5.1.3 NSS

NSS 是作者 2010 年初为实现 EOMS 而提出来的一种软件架构方案，它为 EOMS 的实现提供了技术上的保证。作者已经完成了 NSS 的一个应用

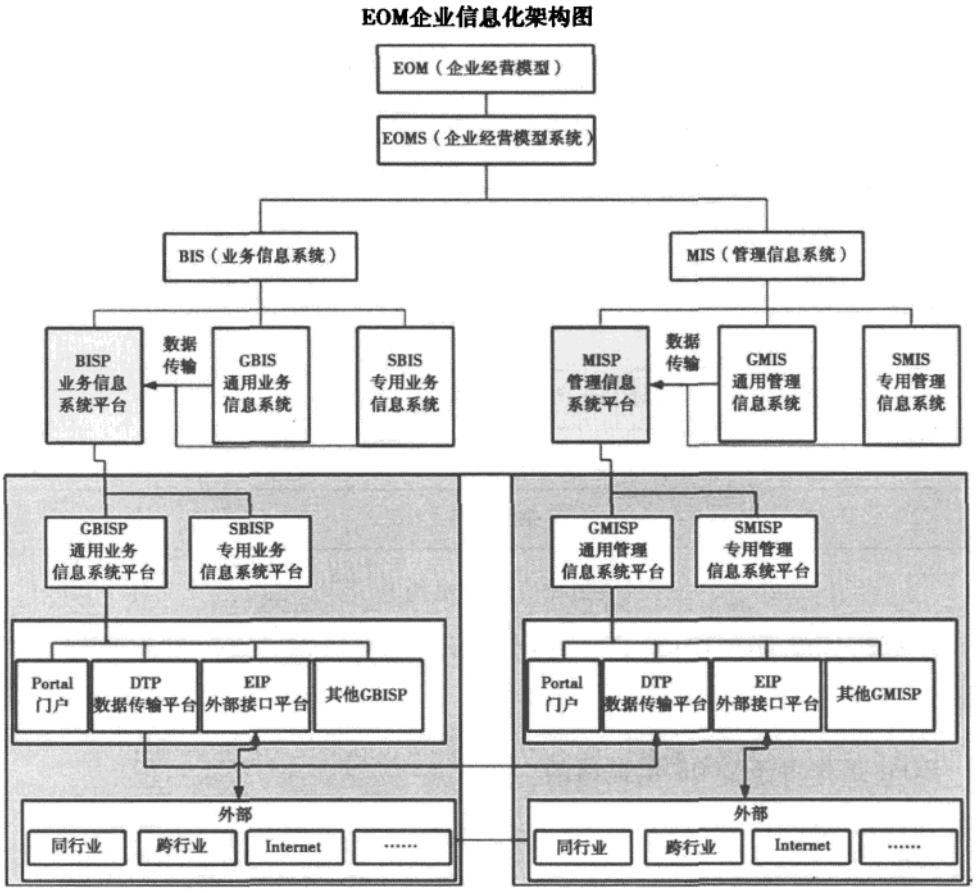


图 5-2 EOM 企业信息化架构图

实例。

NSS（New Software Structure）即软件新架构是按照分层和面向服务方式对软件进行架构的方案。它将一个应用系统分解成用户开发层和服务层，服务层包含应用小类层、应用大类层、EOM 类层、基础核心层。服务层中的每一层都是一种服务，供上一层服务调用。而用户开发层则包含了参数、用户界面以及人际交互的接口处理。

现有的各种软件有的是有架构的，有的是无架构的。有架构的基本上采用 C/S、B/S 等方式，在功能方面会采用三层架构（用户界面层、业务逻辑层、数据层），也有采用面向服务、面向分层的架构，这些架构基本上基于技术上的考虑，主要用于具体应用系统的建立，而且很少考虑到通

用系统的建立。

NSS 则基于 EOMS 的要求，体现了软件面向分层、面向服务的架构思想，可以在最大范围内实现功能的共享，为通用平台、通用系统的出现提供了技术上的保障。也为标准功能的建立和应用提供了可靠的技术保障。

采用 NSS 之后，一个应用系统分成两个部分，一个部分是程序员进行用户开发层的开发，他们只要把参数、用户界面、人机交互接口做好，具体的功能只要调用服务层各层的服务就行了；另一个部分就是服务层，它由专业的服务商提供（当然，程序员也可以按照服务层的分类方法编写服务函数）。服务层理论上向所有的应用系统提供服务，因此，服务层可以在最大范围内复用。

1. NSS 用于应用系统

NSS 可以用于一般应用系统的开发，如图 5-3 所示。

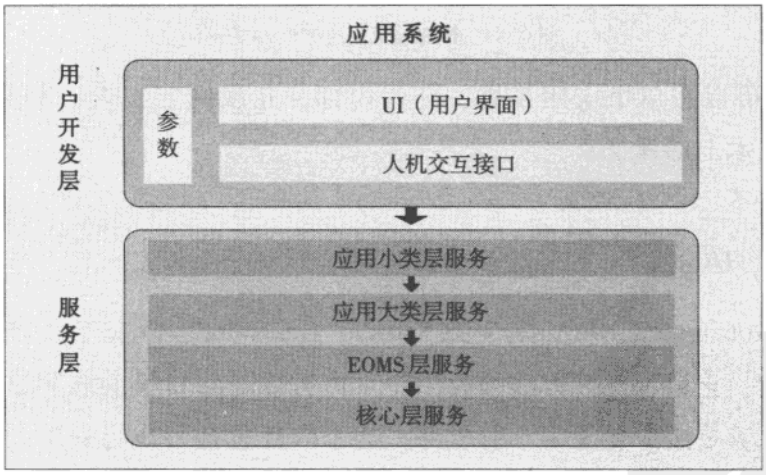


图 5-3 NSS 构架图

2. NSS 用于通用系统

NSS 可以用于企业内部各应用系统的建设以及用于各企业共同的通用系统，如图 5-4 所示。

3. NSS 用于 EOMS

NSS 可以用于各企业共同使用 EOMS，实现了企业信息化在最大范围的标准化、通用和共享（见图 5-4），结束了当前以具体需求驱动的应用系统开发模式，避免了重复开发，实现了各企业和各部门信息和功能最大的共享。

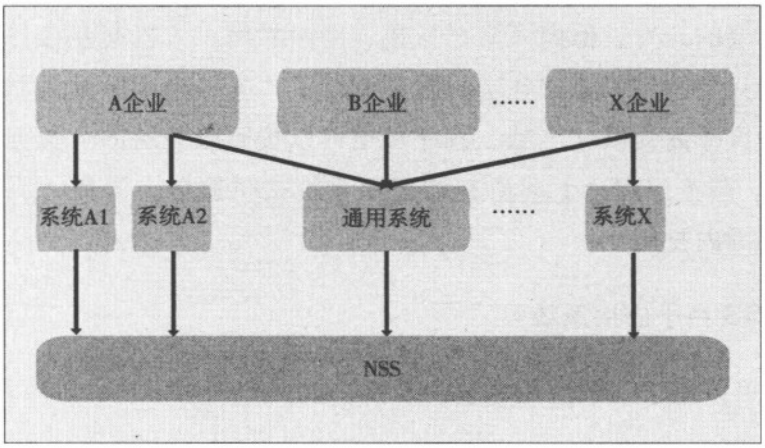


图 5-4 各企业共享 NSS 示意图

NSS 可以在单个应用系统、企业内部、互联网上实现。NSS 为 EOMS 提供了技术上的保证。

5.1.4 IASG

IASG（Interactive Automated Software Generator）即交互式软件自动生成器理论，它通过交互方式接收用户（或程序员）提供的各种信息，自动地产生能满足用户功能需求的软件，这个软件包括可执行程序和这个软件的文档。

该理论由作者在 1991 年提出。针对当时应用程序都是手工编写的现状，力图从技术层面解决软件没有架构，功能没有复用，模块没有标准，程序容易出错等突出问题，建立程序员或用户通过软件描述交互方式产生软件描述文件，再由 IASG（包含了应用系统中很多标准模块和功能程序）

根据软件描述文件来生成一个应用系统。这样方式可以大大提高程序编写的效率和质量，规范用户系统的架构。

作者当时认为绝大多数软件都有一些明显共性，仅仅是操作对象不同而已，如数据的建立（增、删、改）、检查、统计、运算；报表、图形生成；软件系统本身的维护（如数据的检查、备份、恢复、数据转换等）；通信、网络处理等。作者将这些共性抽象成标准操作；与此之外的一些软件也存在特殊需求，如开关机处理、与外设发生联系等，作者将这类需求抽象成特殊操作。由此可见软件的用户需求可分为两个部分：一是标准操作，二是特殊操作。标准操作满足了用户的大部分功能需求，特殊操作满足了用户的特殊功能要求。

基于以上的分析，作者产生了 IASG 思想，即从用户的自描述的数据结构出发，形成了对这些数据结构进行标准操作和特殊操作的可执行程序，并可以产生软件文档，所有这些过程均采用交互方式进行人机信息交流。IASG 主要功能可归纳为：接受软件描述，产生软件，运行程序和修改维护软件等；而 IASG 所产生的程序功能，可以归纳抽象为：显示功能菜单，提供标准操作和提供特殊操作。IASG 的实现步骤：

1）确定程序中的标准操作；
2）确定程序中的特殊操作；
3）编制标准操作程序；
4）编制 IASG 自身程序；
5）运行和调试 IASG 和程序。

在实际工作中，IASG 采用 C 语言实现，首先设计出软件描述文件的格式，然后根据这些格式文件作相应的模块化编程，把标准操作模块编译成 obj 文件，把用户的特殊功能描述文件也变成 obj 文件，然后链接两者，即可得到一个可执行程序。

该思想的提出极大提高了程序员的工作效率，作者当时利用 IASG 在短时间内生成了近 10 个实用软件，其中一次用了 4 个小时，就实现了公安自行车管理系统的需求分析、设计、编码、安装盘制作等所有开发流

程。这些实用软件不仅具有良好的一致性，而且软件质量很高，几乎没有 bug。

作者实现了 IASG 两个版本：一个是基于 UNIX 系统的 IASG，采用 C 语言编写；另一个是基于 Windows 的 IASG，采用 VFP 语言编写。

作者用 UNIX 系统的 IASG 开发了人口信息管理信息系统 5.0 以及公安派出所综合系统管理系统等；用 Windows 系统的 IASG 开发了公安户口审批系统、银行报修系统、公司设备管理系统等。

5.2 项目简介

5.2.1 公安常住人口信息管理系统

为了迅速准确打印居民身份证底卡，对居民身份证进行科学管理，为了实现公安人口信息管理现代化，特设计公安人口信息管理信息系统。自 1986 年起，它在江苏省各市县投入使用，为江苏全省颁发居民身份证和对人口信息进行现代化管理作出了重要贡献。该系统是一个不断完善和发展的系统，版本由 1.0、2.0、3.0、4.0 升级到 5.0。作者参与了 1.0、2.0 开发工作，主持了 3.0、4.0、5.0 开发工作。

系统完全按照公安业务的要求进行设计，能够满足实际的工作要求，其共有 6 个子系统：1）人口信息的建立和维护系统；2）人口信息检索统计系统；3）居民身份证管理系统；4）人口信息户籍管理系统；5）人口信息文件管理系统；6）人口信息管理系统说明。

系统主机采用 386 微机、内存 2M、硬盘 90M 2 只、汉卡一块、4 I/O 卡一块。采用 XENIX 多用户操作系统；一般可带 4 台终端（最多可带 64 台）。采用 C 语言为开发语言，在核心处理上采用了汇编语言。一个人口信息为 48 字节，采用压缩存放，信息按照文件系统进行处理。一台主机能支持 200 万以上人口信息的有效管理。

5.2.2 报表中心系统

这是2001年作者主持的开发项目。针对当时银行日常经营管理中报表多、报表负担重、报表手工制作成分大，这不仅仅加重了员工的负担，降低了劳动效率，而且对于报表数据的准确性也不能得到很好地保证，降低了管理的精细化程度等状况，于是作者提出了改变自下而上、层层汇总的传统报表模式，建立了一个统一的、不分业务的统一报表平台，集中各种业务系统数据采用BI工具制作报表，让业务部门用户自行制作报表等的系统设计目标。

该系统集中了零售、会计、信用卡等银行核心业务数据；涉及各类报表100余种（同时提供报表的下载、打印功能）；支持多种数据采集方式（包括手工数据）；提供面向用户的报表制作、处理工具（便于用户自己制表和维护）。系统上线以后极大提高了报表的产生速度和数据精度，减轻了一线人员的报表制作负担。同时系统还提供了查询类需求的二次开发接口、简单的查询类需求，通过界面对数据源、条件、结果集进行配置就可以完成。

该系统自2002年上线以来，应用面非常广，包括省行各业务、管理部门，所有的二级分行、支行及网点，使用人员包括各类经营管理人员、专业技术人员及操作人员；使用频率非常高，日均访问量近万，总访问量已超过千万；通过二次开发接口以动态配置查询数百个，这些查询在上线之前都是当成独立的小系统来开发；处理数据量非常大，数据处理量累计数据库空间和报表空间均达到TB级。每日的源文件增量、数据库增量、报表增量均达到GB级。无论是从数据规模、空间规模，还是从功能规模、用户规模都是目前该行最大的MIS系统。

系统在开发初期有数十个子系统，采用了多种技术，其既有前台也有后台，其中前台既有B/S架构，也有C/S架构，B/S架构中前台采用ASP、PHP等技术，C/S架构中的前台界面使用了PB、VB、VC等多种技术，数据库使用ORACLE、SQL Server、ACCESS、VFP等。系统到了2004年进行第一次升级时，将系统整合为单纯的B/S结构；前台采用.NET技

术，使用 C#作为界面开发语言；报表工具采用 BO；后台操作系统为 IBM AIX；数据库为 ORACLE 8，使用 SHELL、PRO C 作为开发语言。前台服务器采用 IBM 3650，后台服务器为 IBM RS6000。

5.2.3 银行反洗钱系统

“洗钱”（Money Laundering）是指将毒品犯罪、黑社会性质的组织犯罪、恐怖活动犯罪、走私犯罪或者其他犯罪的违法所得及其产生的收益，通过各种手段掩饰、隐瞒其来源和性质，使其在形式上合法化的行为。这不仅侵害了金融管理秩序而且也严重破坏了公平竞争规则，破坏了市场经济主体之间的自由竞争，从而对正常、稳定的经济秩序带来一定的负面影响。“洗钱”活动所必经的“入账”、“分账”、“融合”等 3 个阶段都必须借助商业银行网络来实现，因此世界各国在打击洗钱犯罪时都非常注重商业银行的作用。

我国的商业银行反洗钱工作主要通过上报人民币、外汇大额和可疑资金交易的方法来实现。2003 年，作者主持了所在商业银行（省级分行）的全辖统一的反洗钱系统的开发。在系统开发中首次提出了“标准金融交易”的概念，解决了银行各业务系统交易信息异构化的问题；并建立了一套完整的反洗钱算法模型，该模型可以动态地调整“额度”、“频度”等参数信息，从而较好地适应监管部门的要求。系统通过定时从各类业务系统中采集数据，生成标准的金融交易，再加以逐级汇总、逐级补录、逐级甄别，最终产生各类结果文件，并且还有报表分析功能，系统投产以后有效地减轻了一线人员的负担。后来总行在此基础上开发了全辖性的反洗钱系统，实现了与中国反洗钱检测中心的对接，获得了中国人民银行对此的通报表扬。

系统采用省分行、总行两种架构，其中省分行反洗钱系统由一个后台批处理软件和一个反洗钱前台组成，总行反洗钱系统由一个反洗钱网站组成。两个架构前台均采用 .NET 技术，后台均采用 IBM AIX 操作系统及 ORACLE 数据库，批处理采用 VFP。

5.3 作者和网友之间的精彩回帖节选

5.3.1 “功能和界面哪个更重要”的精彩回帖

卡通一下：

楼主对功能与界面的理解不太准确。对于娱乐类的软件，当然是界面越漂亮越好，因为要“娱人以乐”嘛！可是，对于办公类或管理类的软件，就恰恰相反，界面越简洁越好。楼主同意这个观点吗？

N216：

不同意！界面重要并不是指界面越漂亮越好，或者越丑越好！首先界面设计要满足客户的要求！其次，界面设计要符合界面设计内在的客观要求。你说的界面简洁就是一般界面要求中的一项内容。我在这里并没有对界面使用对象进行分类，我也没有强调界面设计的具体要求。我只是希望程序员能走出编程的小天地，更多地考虑客户感受，更加接近用户，这样程序员的眼界才能更加开阔！

生鱼片：

楼主最后提到程序员要重视界面设计，我想楼主没有真正认识功能和界面，首先，你无法说明功能和界面哪个重要，只是在不同的阶段、不同的场合注重不同，但是最终二者都是不可忽视的。其次，功能和界面应该分离的，由不同的人来完成，程序员由于在专业上并不擅长，程序员就是该注重程序的功能，界面是由专业的前端人员去完成，但是程序员应该和前端的界面人员做好协作和沟通。

N216：

理想状态下功能和界面应分离并由专业人员完成！但是在现实中，许多程序员身兼了功能和界面制作的两项工作。我的想法是程序员不要因为不擅长而畏惧界面制作，而且要通过界面设计去直接了解用户的需求和感受，这样无论是程序还是界面都会更有用户价值。

对于已经分离了的程序员，我依然希望他们能开阔眼界，能把界面设计当做程序员准入门槛。做一个心中装有色彩、装有美好的程序员。

N216：

我只是希望程序员能走出编程的小天地，更多地考虑客户感受，更加接近用户，这样程序员的眼界才能更加开阔！

卡通一下：

楼主能否举一个自己所经历的例子，以便谈谈改进客户的感受。

N216：

十多年前，我给客户做了一个审批系统，当时我做系统的时候，采用的是 IASG 技术，通俗地说，就是通过功能模板进行对象的配置，以达到功能的实现。因为我写的程序从来就不是为一个系统所写的。

记得当时的功能是打印审批表！是套打的那种！当时用户指定了打印机和打印格式，我就按照格式设计出打印格式，并且打印完全符合客户的要求，当时我很得意，客户也很满意。但是，过了一段时间我的客户提出了更多的要求：

1）打印机型号会变更；
2）审批表格式如果有变化怎么办；
3）每次打印可能不止一份；
4）打印时可能需要预览；
5）可能要打印其他格式的表格。

面对客户的需求，我认真进行了分析，我发现客户的要求是很合理的。于是，我专门设计了菜单让客户自己设计报表格式，专门设计了报表定义功能，让命令按钮和报表之间建立联系，在原来通用打印界面中增加了预览和打印份数以及报表选择的功能。

这样客户的需求当时的需求和未来的需求都满足了。这也使我的 IASG 功能得到了扩展。这件事给我一个启发：那就是要经常面向客户，向客户

了解需求，客户需求总是对的，要注重界面设计，尊重用户的操作习惯等。这样你的界面和功能才能满足用户的要求。光是自己在办公室想问题是想不出来的。不知道这个例子能不能满足你的要求！

卡通一下：

楼主，我觉得你举的例子很好。换成其他的打印机，首先面临的就是定位不准，例如上边界、左边界、裁纸线等的问题，这些都要重新进行调整。个别的还有字体、字号情况，如使用硬字库的。预览和打印几份，这不算什么事。让客户自己设计报表，我只是听说过，没有做过。界面的问题我们说了很多，我讲讲自己的看法。

界面是用来与人交互的，所以人的感觉是最直观的，也就是大家所说的客户体验。我对客户体验的认知是，不但要有视觉上的体验，更要有操作上的体验。娱乐类、门户网站类等，这类界面的操作是“随机”的或是“无序”的；而管理类软件，这类界面的操作是“循环”的“有序”的，所以各自有各自的客户体验。

管理类软件，人机的交互强度很高，那如何设计出让客户在操作上感到流畅、便捷、轻松，这不是技术上的问题，而是开发人员是否能站到操作人员的立场上，做到“贴心”地开发。我不太赞成将功能与界面相比较，因为常常就是添加了一个小小的功能，而使界面更加友好。至于如何做到这些，确实也不是一两句话就能说清楚的。

N216：

我对你回帖中的“硬字库”印象深刻！我想现在的程序员应该很少了解吧！大概在 20 世纪 80 年代，当时计算机内存容量很小，由于要汉化，出现了汉化操作系统，其中就是要支持打印汉字和汉字符号，由于当时国标 2 级汉字大概在 7000 个汉字左右，加上符号区 21 个左右，一个区大概是 94 个符号吧（记不清了）。很多是空区，估计总共要 1 万个汉字和符号。这些汉字和符号都是以点阵存放的，其中 8 个点为一个字节存放！一个汉字如果用 24 ×24 点阵存放就需要 3 ×24 个字节。1 万多个汉字符号，

可以放置在硬盘上，称为“软字库”。但是速度相对较慢，而且还占用硬盘空间。当时硬盘空间是相当宝贵的，1985 年的时候硬盘空间只有30M ~ 40M。为了不占用硬盘空间，提高系统速度，当时就把汉字点阵放到了汉卡中，而汉卡又称为硬件，所以叫硬字库。当系统调用要打印汉字时，将要打印汉字的国标码作为参数，调用读汉卡的接口程序，读出这个汉字的点阵，放到内存中的打印缓冲区，然后根据打印机的打印序列，将打印点阵序列输出到打印机上，这样汉字就打印出来了。如果要放大字体那就要计算点整大小，放大点阵，同理输出到打印机上，这样不同字号的汉字也就出来了。

往事历历在目！不过由于当时很清楚地了解了操作系统和打印机原理，所以到现在谈及此事都是不能忘怀的。

5.3.2 “优秀程序员应该具备哪些素质”的精彩回帖

WorkTimer：

要求太高了，感觉世上没有一个人能全部满足此条件了，即使能满足此人也不再做程序员了。

N216：

“世上无难事，只怕有心人。”一个人什么都可以怕，但是不可以怕要求。如果一个可以成为有优秀的程序员，他自然可以成为优秀的设计师、优秀的项目经理、优秀的企业管理者、优秀的……

因为人的进步是没有止境的！现在许多程序员对自己的要求太低了，低得不可想象！这种现象说明了什么？我想我们每个人都会给出自己的答案！

冰河魔法师：

或许是经历不够，反正至少现在是没办法领悟：“优秀程序员也是优秀的项目经理”，在我看来，项目经理是管理类人员不是设计类人员。

N216：

优秀程序员不是一日可达的，它确实需要日积月累！但是并不是时间长了就会编程优秀了。大部分程序员都是普通程序员，优秀的只是很少的一部分。在我看来，许多设计师来自于程序员，许多项目经理来自于设计师，这种情况现实中很多！因为只有这样，高层的人员才有好的基础，才会把程序和项目管理好！这两类高级人才都需要很强的逻辑能力。

金色海洋：

程序员是和代码打交道的，优秀的程序员就要更了解代码，更了解计算机。项目经理是和人打交道的，人和代码可是完全不同的呀。普通的程序员只要了解代码和计算机，他们的视野很低。而优秀的程序员则有开放目光，他们不但知道做什么，而且知道为什么要做，做了后有什么价值。有了内在的理想和动力，这样的程序员和普通的程序员具有本质上的不同。

不过你们可以限定优秀的范围定义在程序员对代码和计算机的了解程度上！也可算是次优秀吧！

OPQRST：

满足这些十大要求并不难，就看老板出的钱到不到位。

N216：

有的人活着是为了别人，有的人活着是为了自己。先有鸡还是先有蛋，在这个现实社会中，我们可能都会有自己的客观答案。优秀程序员自然可以找到合适的公司、合适的岗位和合适的报酬。暂时找不到只是个时间问题。实际情况是，老板出了钱却找不到合适的人，这种情况比较普遍。而程序员做出了超出实际报酬的工作的情况也时有发生。这种情况在于公司老板太注重自身的利润，不注重员工的贡献，比较短视。所以说优秀的老板还是很少的呀！

#24 楼：

优秀的程序员要有勇于“自我否定”的勇气！这不仅是素质，也是一种修养。

N216：

这是非常非常重要的素质和修养！所以说十全十美是不够的！这可以算成第十一条素质！“自我否定”是一种进步的标志。很多程序员在工作一段时间后，回顾前面的工作，就会感到自己原来是菜鸟，于是乎感觉到了进步。再过一段时间，又重复了这个感觉！这样反复多次，一次一次否定自己的过去，使得自己最终到达了优秀的顶峰！

反之，那些自以为是的程序员，编了一些程序后，尾巴翘上了天。过了一段时间，尾巴还在翘，再过段时间，尾巴始终没有掉下来的迹象。这种不知道“自我否定”的程序员是不可能优秀的！不过，程序员最要紧的素质还是“心怀理想”。这点我是很坚持的！逻辑推理固然重要，那仅仅在技术层面，真正让一个人优秀起来是这个人的理想和目标，这是他提高自己技术的内在动因。没有这个动因，再有什么逻辑推理都无用武之地！或有用武之地，但这个地很小很小。

JamesYY：

条件太严苛，收入不对称，十项俱全的人才肯定不会来做程序员。

N216：

条件高低，在于优秀人才的多少。如果放宽优秀的条件，普通天下程序员都是编程的优秀程序员，那优秀还有什么意义？如果条件太高，没有一个能成为优秀程序员，那优秀变成了虚幻的影子。我感到这个十大素质应该是适中的。更苛刻的条件还有很多，我也不想列举了。但是，有了这些条件，就会给人奋斗的方向。

收入问题是一个很复杂的问题，下面文章中我会涉及这个话题的。但是我要说的是，能力是程序员自己的，你可以把握，收入是老板给予的，

你很难改变。提高自己的能力是硬道理。到时候，你可以高喊：“此处不留爷，爷到他处去。”现在和今后的社会靠能力吃饭还是正道！十项俱全的人才肯定不会来做程序员，这个有一定的道理。主要是付出和收入差距太大。如果有一天程序员的付出和收入差距处在一个合理的范围之内，我想还是有人愿意当程序员的。我写这些文章的目的之一就是要让程序员“富”起来。

奎彤：

在我想法中的软件行业也不全然是个夕阳行业，但是在我概念中的软件行业范围可能大些，如果以博主所推广的 EOM 来说，我认为是属于企业管理软件行业的一部分，而我恰好认为企业管理软件行业是没有什么爆发力的行业。如果以我的角度来看，ECOM 等于 EOM = ERP + CRM + WF + EIP + DW。

这个行业的缺点之前提到了多次，例如重复开发、低技术、没有累积、需求五花八门。但即使这样，我认为想要利用 EOM 来做标准化还是过于理想化。因为我也身处这个行业中，做一些 ERP 相关的定制化开发，发觉如果想要做一套放诸四海皆可用的系统是不切实际的；目前采取的做法是“标准化开发方法”，减少开发人员重复学习的时间，需求要怎么五花八门也只能看客户想法，再做些折中吧，但还是客户为大。软件行业当中当然也有很多获利丰厚的市场，这造就了大型的跨国企业，例如 Oracle、微软、Apple、SAP 之类。不过他们占有的市场我们应该是很难与之分食的。

总之，诚如博主所言，我们的观念不同，我不认为能够大幅提升程序员技术水平及生活水平的行业是企业管理软件。据我个人所知，能够将软件搭配硬件的 3C 产品，它的软件开发人员待遇不错。不然就是大型互联网公司的软件开发人员吧。

N216：

提出 EOM 的原因是 ERP、CRM 等软件在中国举步艰难，而ORACLE、

Windows、SAP、BI 等软件大行其道。

前者举步艰难的原因是这些软件要适应具体企业各种生产管理的现状，而企业经营管理的现状是五花八门，即使是一个企业内部它也不停地在变化。让软件适应企业必然会有这种结果。而后者往往不关心具体企业的经营状况，只是作为信息化的工具，企业只能使用工具，而工具不必适应企业，这样必然有其市场，利润丰厚。

EOM 则是站在企业经营的高度上描绘出企业信息化整体架构，它通过架构来规范企业信息化的未来，同时建立适合现代企业经营发展的各种通用软件，企业使用 EOM 产品可以提升其经营管理水平，因此企业处于适应 EOM 的状态，这样 EOM 产品就是一个通用的产品，制作 EOM 产品就可以创造出甚至比 ORACLE、Windows、SAP、BI 更大的市场和利润。

相对企业来说，EOM 是成品而 ORACLE 等只是原材料。

至于不可能做出放诸四海皆可用的软件之说，可能缺乏对软件的分层研究造成的。这是一种以结果取代过程的思维方式造成的，是一种满足现有为标准的思维方式造成的。

1）软件当提升到整个企业信息化高度的时候，软件不仅仅一段功能的程序实现。软件出现了分层。首先企业信息化出现了架构，这个架构可以是每个企业都是一样的。至于架构的内容每个企业都不一样。就内容而言，也是可以分层的。例如可以分为通用的软件和专用的软件，通用的软件就可以解决共性的问题，而绝大多数的企业连共性的问题都没有解决。因此，通过软件的分层，我们可以看到有许多软件将会以架构形式、通用软件形式提供给各个不同的企业。

2）评价一个软件的好坏不是以其能否满足现有企业的经营管理要求为准，而是看它能否改善和提高企业经营管理的水平。从统计学上来看，目前绝大多数的企业经营管理的水平是相对比较低的。用先进的科学技术去满足水平低的经营管理不能不说是一种无奈。所以，我们应该建立一个

理论体系，根据这个体系构建科学和先进的企业信息化来提高其经营管理水平。而 EOM 正是这个体系。如果我们能建成 EOMS，我相信目前出现的各种问题，绝大多数都能得到最好的解决。

5.3.3 “程序员的春天：EOM 与程序员”的精彩回帖

卡通一下：

楼主从头至尾全都是“褒义”词的堆砌，根本就感受不出什么春天的气息。我想所有阅读过你这篇文章的人都会有这种感觉！也许 EOM 确实是你多年实践中的结晶，但最好用浅显的例子来讲解，不然我总是觉得像电视广告一样！

N216：

春天到了，但是你感觉不到，你是不是习惯于冬天的环境？你这样感觉，我也没有办法。我认为用权限管理的例子可以说明 EOM 的实际应用的流程了。

卡通一下：

楼主在文中说道：“EOM 的出现至少给很多有市场意识、有理想的程序员一个机会，他们可以利用业余时间进行研究、宣传、设计、开发、销售、维护 EOM 的各种产品。我们可以建立网上 EOM 团队，以团队方式吸引有思想、有才华的各类人才，开发这个产品，经营这个产品，实现自我价值。”

我始终强调企业开发不同于网站，它需要“盯现场”，需要了解企业的管理模式、基础数据、规章制度、各种台账、各种票据、硬件现状等。更重要的是，在大致架构出来之后，还要与客户逐项地进行沟通与确认，这能是通过网上来做的吗？说到定制开发与你推崇的 EOM 理念，使我想到 ERP 的处境，它的失败率远远高于成功率，为什么呢？

N216：

你的思维定势就是“需求驱动”。你认为企业目前的管理模式都是正确的，这是你的出发点和终点。而 EOM 则是抽象地看待所有企业经营，它能从理论上阐述什么是好的企业经营方式。用这个经营方式和现有的企业经营方式进行比较找出其不足，通过提供通用的软件来改进企业经营方式，使得企业经营获得更好发展。这就是和你的不同点。

ERP 的问题是 ERP 仅仅是从某个角度看待企业经营，比如从进、销、存等，而没有以整个企业经营为其落脚点。另外，ERP 系统在实践中往往会和现实的企业管理模式产生矛盾，解决这个矛盾的方法是 ERP 去适应企业现状，从而造成了 ERP 走样，产生了目前的两难境地。

EOM 则和 ERP 走的是完全不同的道路，它首先是建立一个理论模型，通过对模型的研究来规范企业经营的各个方面，可以说 EOM 是有理论为先的科学，只有理论为后的信息化才有客观的评判标准。如果你认同需求驱动、ERP、CRM，那你就会面临现在的各种困境，而且很难自拔。而 EOM 则是看到这些问题的根源才提出了解决企业信息化的科学途径。

卡通一下：

楼主写文章确实充满了一股激情，回帖也非常认真，应当给予肯定，只是这个系列给人的感觉都是讲宏观面上的东西，让大家不好认同。对于大家的质疑，楼主一般认为对方的立场、观点是浅层次的，达不到楼主的层次。我们一直按照客户需求来编写程序，即便是自己练习编程，也还是会自己定义一个场景，不是吗？

就楼主的 EOM 构想而言，我认为从本质上来说，也应当有你自己认可的客户模式，不可能是无形的东西，否则怎么设计呢？从你前面的一些示意图来看，应当是有相对固定的模式，只不过你是想用自己的 EOM 模式来套取天下所有的企业管理，行得通吗？如果你对企业老总说，他现在的管理方式不科学，需要用你的 EOM 来管理。我想这个老总一定会对你说：你以为你是谁？

N216：

对于你的问题看来还真得一一道来：

1. 宏观问题

程序员长期编程，长期工作于微观世界，结果如何？看看周围的程序员，看看有哪几个能超过你的，看看哪个不是存在各种各样的问题。你分析过其中的原因了吗？我告诉你，程序员的眼界非常重要，如果程序员只埋头于微观世界，这个程序员是不能成为大器的。程序员真的应该多多开阔视野，尤其是要加强对宏观问题的学习和研究，有的放矢地开展编程工作。也许在现实中，太多的大道理让人们调头就走，但是道理依然存在，关键是学会接纳宏观的看法，学会判别宏观中有用的东西，这样程序员才能真正懂得“看路拉车”的意义。

2. 质疑问题

对于任何的质疑，我都会认真对待。我始终对事不对人，对事我就要找出别人质疑的原因，从而指出其想法的根源。的确，有些质疑反映出程序员常见的问题，例如，经验不足、思维定势、对人不对事、逻辑不清晰、只拉车不看路、眼界狭小等，我很直率地指出就是想让人能意识到自己的问题，别无他意。因为在这里讨论问题并不是想显示自己的水平高低，我早已过了和别人比高低的阶段，我只是想把一些自己总结的道理说出来与大家讨论和共享。听之则幸，不闻则由己。

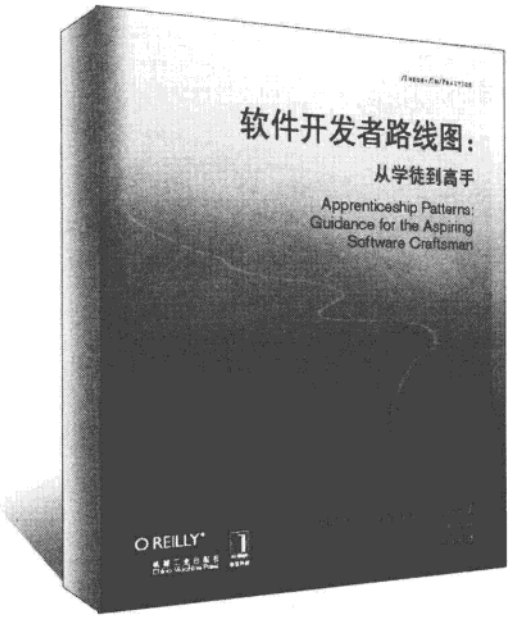
3. 关于 EOM

EOM 能否解决天下所有企业管理问题，我想首先要从理论上其能否说得通，如果在理论上行得通，则事成大半。理论可以了，还有一个实施、推广的问题，这个问题可能只有做了才知道，但是只要做了，则一定会有成效的。所以我关心的是理论和实施。世上都说不可能的事，往往是最有可能的。

4. 关于“你以为你是谁”

老总会说“你以为你是谁”吗？如果你现在当老总，你会说这样的话，但是正因为你有了这种想法，所示你当不了老总。企业高管存在的目的就是为了这个企业的经营发展，凡是对企业经营发展有利的事，他们都会听之、辩之、办之。否则他们的自大意识就决定其当不了老总。EOM 之所以能成为一个理论，自然有它的由来，如果它连企业经营管理各种问题都发现不了，找不出解决方案的话，那它如何能成为经济学、企业管理学、企业信息化的理论基石呢？

当然我们不否认有人对 EOM 的认识不同，那只是我们的宣传问题，有关别人的理解的问题，还是那句话：“懂吃懂的苦，不懂吃不懂的苦。”



软件开发者路线图：从学徒到高手

作者：Dave H. Hoover　Adewale Oshineye
ISBN: 978-7-111-31006-8
定价：35.00元

内容简介

作为一名软件开发者，你在奋力推进自己的职业生涯吗？面对今天日新月异和不断拓展的技术，取得成功需要的不仅仅是技术专长。为了增强专业性，你还需要一些软技能以及高效的学习技能。本书的全部内容都是关于如何修炼这些技能的。两位作者Dave H. Hoover和Adewale Oshineye给出了数十种行为模式，来帮你提高主要的技能。

本书中的模式凝结了多年的调查研究、无数次的访谈以及来自o'reilly在线论坛的反馈，可以解决程序员、管理员和设计者每天都会面对的困难情形。本书介绍的不只是经济方面的成功，学徒模式还把软件开发看成一种自我实现的途径。读一读这本书吧，它会帮你充分利用好自己的生命和职业生涯。

厌倦了自己的工作？去找一个玩具项目来帮你重拾解决问题的乐趣吧，这叫“培养激情”。

感觉要被新知识淹没了？做点以前做过的事情，重新探索一下自己熟悉的领域，然后通过“以退为进”再次前进。

学习停滞了？那就去寻找一支由富有经验和才能的开发者组成的团队，暂时呆在里面“只求最差”。

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容	☐很好	☐一般	☐较差	☐理由_____
文字质量	☐很好	☐一般	☐较差	☐理由_____
版式封面	☐很好	☐一般	☐较差	☐理由_____
印装质量	☐很好	☐一般	☐较差	☐理由_____
图书定价	☐太高	☐合适	☐较低	☐理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是_____　☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收
邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com