程序员的素质

白杨
http://baiy.cn
2010-12-01

众所周知，软件和硬件工程师个体间存在巨大的生产力差异：经常有一个人一两天就能做好的事情另一个人花一两个月也没能做到同样好。

Apple 创始人史蒂夫·乔布斯曾在《In the Company of Giants》一书中接收采访时提到："一个最优秀的人完成工作的能力能抵 50 到 100 个一般水平的人"。而软件工程领域也经常提到最好和最差的程序员之间的生产力差异超过 1000 倍，因为如果一个程序员产出的代码中包含了很多 bug，需要其它程序员花费大量时间去修正，那么他的生产力是负值。

在世界名著《人月神话》中，作者也明确地提出了类似的观点："最优秀和一般的软件工程实践之间的差距是非常大的，可能比其他工程领域中的差距都要大……"。

除了团队组织和项目管理等行政方面的问题；以及语文、数学、逻辑思维和形象思维等个人基础素质以外，到底是哪些专业素养导致了程序员个体间的巨大生产力差异呢？本文尝试以由浅到深、由易到难的顺序来讨论程序员的个体素质。

1. 基本技术素养

基本素养包含了每个程序员都应当牢固掌握和充分理解的技能和知识。基本素养中又可以分为前景知识和背景知识两种。

* **前景知识**：今天的程序员主要通过使用编程语言书写代码来完成软件开发活动，前景知识则特指程序员对其使用语言的掌握程度。这里所说的"掌握"并不是仅仅能够写出合法的代码就行了，而是对语言中各种实现细节的深度掌握。

  如果一位程序员未能透彻掌握他正在使用的程序设计语言，那他写的程序就一定会或早或晚地出现一些至少在他看来有些"莫名其妙"的问题，比如：应用程序时常莫名其妙地崩溃、经常发生资源泄露、或是执行过某些操作时性能差的离谱等等。

  以 C++语言为例：正如大部分 C 程序员都能毫不费力地将任意 C 源码"肉眼翻译"成伪汇编码那样，一个合格的 C++程序员也应当可以做到将任意 C++源码"人脑编译"为对应的伪汇编码。但现实的情况是由于 C++新增了模板、虚函数、RTTI、虚基类、异常等大量高级特性，以至于能够真正透彻掌握 C++语言的程序员很少。

  除非完全不使用前文中列出的那些 C++高级特性，仅把它作为一个"稍好的 C"来使用。否则，对于任何一个职业程序员来说，贸然地使用一种尚不了解其底层（编译器或 VM）

实现细节的语言特性是一件很危险的事。

类似的例子同样适用于其它语言。不管是编译型的（如：C/C++）还是解释/虚拟机型的（如：PHP、Java 以及 JavaScript）程序设计语言，它们都有一些需要程序员事先透彻理解和掌握的实现细节。

* **背景知识**：透彻地理解和掌握至少一门编程语言对任何一位职业程序员来讲都是很重要的基础素质，但仅仅做到这点仍然是远远不够的。想要产出具备现实意义的产品，还需要具备足够的背景知识。以下，我们将背景知识分成“公共背景”和“领域背景”两类，分别进行讨论。

  - **公共背景**：公共背景包括了几乎每个程序员都需要掌握的知识。从操作系统原理、编译原理、数据结构、离散数学、计算机组成原理、网络原理等理论性知识到各种接口和实际环境的运行时 API（例如：对 C/C++来讲通常是操作系统 API，对 JavaScript 来讲通常是浏览器提供的 API）都是作为程序员必不可少的基础。

  - **领域背景**：领域背景中包含了与具体问题域相关的背景知识，这些知识并不是每个程序员都需要掌握的，但对于涉及该领域的开发人员来讲却必不可少。

    例如：对于要实现音频编辑器的程序员来说，至少具备起码的声学背景；MIDI 编曲软件的作者需要乐理和和声方面的知识；OCR 软件的作者则要学习模式识别和图像处理方面的理论等等。

2. 思维的条理性和连贯性

对于一个程序员来说，从产品的设计之初到其中组件的逐一实现完成乃至最后的联调。整个开发周期从始至终都需要保持清晰的条理和连贯的思维。以下准则有助于帮助程序员在开发过程中始终保持思维的条理性和连贯性。

* **3W 准则**：我想每个程序员都应该遵循 3W 准则，即：每次在开始编写代码之前，都应当先搞清楚 What、Why、How 三个问题。其中“What”指我们要做什么，也就是用户的需求。“Why”表示为什么我们要做这件事，这是对需求和架构的更深层理解。很多时候，只有充分了解了“为什么”以后，我们才能更好地完成设计（例如：使用更合适的算法和构架，以及在架构中预留恰当的接口等等）。最后“How”从设计层面指明了应当以何种架构、哪些模式和算法来最终构建出产品。

  值得指出的是，3W 准则不但适用与对系统进行整体分析和设计，同样也应当利用在模块或组件等粒度更细的层面上——在开始着手编写一个组件前，应当事先搞清楚这个组件的功能（What）；需要在何处使用它、为什么需要用到它、它的典型用例和工作上下文（Why）；最后是使用何种架构、模式和算法来实现它（How）。

  本质上讲，需求分析和总体设计的意义就是在系统的层面上回答以上三个问题。总体设计使开发人员能够对待开发的产品在整体上有一个比较清晰的了解，知道当前正在实现

的组件处于系统中的什么位置。

在着手实现一个组件前，如果程序员能先在组件的层面上搞清楚这三个问题，那么他就可以在一个目标、动机和实现方式都很清晰的状态下开始工作。进一步说，在开始工作之前，清楚的知道自己要做什么以及如何完成手上的工作对于任何行业的从业人员来说都是非常重要的。遗憾的是，相当多的程序员好像并不这么认为。

* **时刻保持清晰的逻辑**：3W 准则让我们在动手前就清楚自己要做什么以及如何做。但在具体实践的时候我们还需要随时清楚自己正在做什么？之前几步分别做了哪些事？以及之后几步还要干点啥？只有时刻了解自己“刚才”、“现在”、以及“将来”要做的事情，才能随时保持思维连贯和条理清晰。

* **以恰当的层次进行思考**：太高的层次过于抽象，而过低的层次容易让我们揪住某些细节问题不放。在合适的层次思考才能够最有效地想出恰当的解决方案。什么层次才算“恰当”没有简单的定论，这取决于产品的规模和正在解决的问题。但是在碰到棘手问题的时候换个层面来重新考量往往能够事倍功半。

3. 高效的产品维护

软件产品的日常维护主要包含排错、Workaround、功能变更、架构重构等活动。对于新手来说，花个几天时间找“臭虫”实属常见（有句顺口溜说的好：“锄禾日当午，不如 coding 苦，对着 C++，一调一下午……”）。提高产品维护的效率就在很大程度上提高了程序员的生产力（因为有更多的时间编写新代码）。

多年的经验显示，以下要点对于提高产品维护效率、以及提高产品质量都有着不可忽视的作用：

* **编码规范**：编码规范对于软件品质的影响怎么强调都不过分。程序代码作为一种文档，首先是供人类阅读的。好的编码规范能够有效地降低错误率、提高可读性、以及极大地曾强代码的可维护性。换个角度来看，代码只写一次，但却需要断地进行阅读、理解、修改等维护工作。没人愿意接手维护一大堆完全看不懂的代码，在书写时始终遵循一套完善的编码规范则可以在很大程度上缓解这些问题。

  编码规范对程序品质和可维护性方面的影响就像鞋对人类的影响一样：都属于效果巨大，但跟没体会过的人很难描述清楚的事情——真是谁用谁知道。这可能也是为什么那么多公司都在强调它，同时那么多程序员却又不重视它的原因所在——光脚的不怕穿鞋的！

* **错误处理和日志**：优秀的错误处理和日志记录方式能节省大量排错时间，同时揭示产品改进的重要线索。在另一本世界名著《Code Complete》中，作者凭借其多年的软件行业经验以及 NASA、IBM 等各大组织的长期研究报告得出结论：“在绝大多数项目中，最大规模的活动就是调试以及修改那些不能正常工作的代码……消除软件缺陷实际上是最昂贵且最耗时的软件工作”。

这方面牵扯到的技术和技巧太多太杂，以至于没有办法给出一个总结性的描述。因此，这里只能以日常维护场景为例，举一个简单的例子：

试想一下，你有一个可以向系统 syslog、网络 syslog server、磁盘文件、Event Log Service 等等各种日志目的实时发送日志消息的记录器，它能够工作在调用线程或独立的线程/进程中，就算进程崩溃它也能够捕捉到足够详细的错误信息，即使整个系统崩溃了也不会丢失重要的日志消息。

与此同时，你所有组件中的任何操作发生错误时，都会准确地将该操作的错误信息、操作过程中产生错误的模块信息、该模块调用的底层平台 API 以及这个 API 返回的错误信息等等的各级出错详细信息都按照层次结构如实地记录到日志中。在这样的环境中，无论是排错还是调优是不是都会点单很多呢？

正如前文所述，日志记录和错误处理是个很大的话题，足以著书立说。这个例子也只是揭示其冰山一角，从一个小小的侧面来反映一套优秀的日志记录和错误处理机制对排错和性能分析之类的活动有多大帮助。

* **环境和工具**：熟练掌握开发编辑、分析统计、调试跟踪等工具。优秀的开发、分析、和调试环境可以节省大量时间。尽量避免在不熟悉或者较“艰苦”的环境下分析和调试程序。

  当然，有时也会有一些不可避免的情况出现。比如：程序的最终目标平台是基于MIPS架构的NetBSD，但开发人员最熟悉，各类工具最齐全的开发环境确实 x86 平台下的Windows系统。对此，我们的解决方案是：实现一套与跨平台的应用支撑框架，这个支持框架向上封装所有平台相关的功能，为上层应用提供统一的，平台无关的API。然后使用该支撑框架在Windows（x86）环境完成应用的开发、分析和调试工作，然后在NetBSD（MIPS）平台上重新编译并发布。

* **其它因素**：产品维护的成本还会受到很多其它因素的影响。比如在设计之初以 3W 原则精心衡量过的方案可以避免很多后期不必要的变更；比如重用已经经过验证的可靠组件不但节省了开发成本，而且也避免了重新实现产生的缺陷；再比如经验、设计思想和品味对设计产生的影响等等。

4. 标准化和重用

标准化本质上就是为重用做准备。想要拥有大量标准的可重用组件需要长时间的积累。当然，我们这里提到的“标准化”并不仅仅是指 ISO/IEC 之类的国际标准或者 GB 之类的国家标准。实际上，小到公司甚至是个人也可以有自己的内部标准（其实编码规范就属于这种内部标准之一）。C++之父 Bjarne Stroustrup 曾经说过：产品开发就是重用已有标准组件、实现新的标准组件、然后将它们粘接起来的过程。

想要自己实现一套完备的标准组件库当然需要长时间的积累，但如果利用其它人已经实现的

现成库呢？我们知道，当今的开源时代，能够免费取用的第三方功能库数不胜数。但是使用第三方库的成本也不低。具体表现在几方面：首先，要用好它，你需要阅读大量代码和文档；其次，掌握一个库不光是会使用即可，一旦把它用到自己的产品里，那么当它发生问题时你要修正、当它不满足要求时你要修改和重构、当它缺少功能时你要添加……也就是说，一旦在自己的产品中使用了一套第三方库，那么维护它就是你的义务，而要维护一大堆代码，前提是你需要先读懂这些代码。最后，第三方库的架构和品质不一定满足你的要求，而且这个问题可能到最后才会被发现。

例如：由于架构的限制，Windows IOCP 机制无法被在 Boost 库 asio 组件良好地支持。如果你的产品使用了 asio 组件，并且在开发的最后阶段才考虑需要提供针对 Windows 平台的支持，那么你将会面临艰难选择。再比如：你需要一套跨平台 C/C++框架，但 Boost、Mozilla NSPR、Apache APR、GNU Common C++ 等均无法满足你的需求（举个最简单的例子：他们都支持对线程设置 CPU 粘滞属性）这也使你面临两难的选择：是选择其中一个库，大规模的修改它以满足你的功能需求，并且忍受这个实现从架构到编码规范中所有你看不顺眼的地方，同时还要日复一日地将官方补丁合并到你自己的私人分支。还是索性从头开始建立自己的库？

当然，无论你如何选择，这些代价都是值得的。因为我们知道，程序员的生产力基本等同于他在单位时间内产出的有效代码行数与其中每行代码平均表达能力的乘积。即：

```
生产力＝有效代码行数×每行代码的表达能力
```

大量使用标准组件不光获得了由于组件被反复使用因而比较成熟的优点，而且也使得每行代码的平均表达能力大大提高（比如：标准组件通过成千上万行代码封装好的一个功能，使用一行代码即可调用），这就使得程序员的生产力能够以乘积的形式增长。

5. 经验和设计思想

经验和个人品味是永远也无法被替代的。Stroustrup 曾说：任何一个成功的产品中都必然充斥着其作者的味道。这话虽然有一定夸张的成分，但也不可否认，设计者的经验和品味会极大地影响产品的最终品质。而设计思想其实是来自于经验的总结和升华，当然其中也糅合了个人的品味与喜好。因此，设计思想只有成熟和幼稚的区别，而成熟的设计思想之间很难有好坏之分——成熟的思维各有各的老练之处，但所有人幼稚起来则是大同小异的。

6. 人品（态度）

这标题看起来有点怨天尤人的意思。但我们这里提到的人品既不是传统上的“忠孝礼义仁智信”，也不是现代网络用语中“运气”的意思。这里主要是指程序员的责任感和追求完美的态度，还有乐于与人交流的生活方式。窃以为缺乏这几样东西的人无论如何都不可能成为真正优秀的程序员。

总结

写完以后发现此小文中的各小节有一节更比一节短的趋势。本文开头说各小节是以由浅到深、由易到难的顺序来排列的。写完后再看又何尝不是从具体到抽象的排列呢？前面说的都是比较具体的技术技巧和知识，容易说出个 1234。而越到了后面哲学色彩就越浓厚，评判标准也越主观，自然也就越发难以表达清楚了。