第14章 进阶之路

14.1 程序的效率

每个人都希望自己的程序有效率，这无可厚非。就像我们每个人都希望自己能够漂亮一点儿一样。如果你是一个演员，漂亮对你来说至关重要。不过如果你是一个在荒岛上的程序员，那么很明显，能上网要比漂亮更有意义。希望程序有效率，这没有错，但是别忽视了应用环境。首先要问自己一个问题：“我的程序是否是时间敏感的？”

如果我用非常简单明了、直接的程序可以解决相应的问题，是否就不满足效率的要求了？千万别忘了，计算机干别的真不行，但是计算起来真的很快。

举例来说，一个包含一万个词的词典，满足用户基本的增加、删除和查找功能，你用最低效的数组和最高效的红黑树，对于最终的用户来说，根本感觉不到差别。

唯一的差别就是，对程序员来说，用红黑树实现，代码更多，开发周期更长，开发成本更高，代码更易出错。所以，在关注程序的效率问题之前，你首先要从面临的问题出发，问一问自己，效率是不是这个程序的瓶颈所在。完全不考虑对应的应用，只是在堆砌自己所学的知识，这个在《重构:改善既有代码的设计》[13]一书中有一个非常专业的称呼，叫“过设计”。

如果处理的问题数量级很大，或者是工业控制类时间敏感性程序，那么就不能简单地忽略效率的问题了。程序的效率归根结底在算法上，一个计算复杂度是 $O(n^2)$的算法，无论多么精巧地来实现，无论应用多少提高效率的小窍门，效率也不会高过一个复杂度为 n 的算法。这个问题出在根本上，就像你是老鼠王国里的冠军，也毕竟是个老鼠，也会被一只哪怕最瘦弱的狮子杀死。所以如果我们探讨程序的效率，第一个问题就是要选择合适的算法。

这又回到了算法的问题上了，有些同学一看算法就头疼，这我能理解，我现在看算法也头痛。但是没办法，任何学科、任何工作，抛去风光的外衣，都有一个坚硬的核，看你能不能咬穿它。算法就是计算机科学中最坚硬的壳，你必须咬穿它才能吃到里面的果实，最好像我一样，每天都去咬一咬。

好了，假设你已经咬穿它了，针对任何问题，都已经能找到非常合适的算法，下面就是在编码层次上进行效率的提升，这个时候，需要注意一点。哲学上有一个著名的原则，叫做 8-2 原则，也叫冰山原则。冰山原则的名字来源于这样一个事实，一座浮在水面的冰山，水面上通常只露出 20%，水面下还隐藏着 80%。由此推广开来，一个团队中 20%的人会完成整个团队的 80%的工作，就像西游团队的孙悟空，80%的妖精都是他一个人收拾的。我们再进一步推广到程序，一个程序的 80%的时间都是在运行 20%的代码。试图优化一个程序的时候，我们应该首先优化那 20%的代码，这样才是最有效率的。这 20%的代码段被称为“Hot Spots”。目前有很多检测工具可以帮助定位那 20%的代码，例如 VS2005 以后自带的性能分析工具 Performance Profiler 以及第三方的工具 dotTrace 等。

假设你已经通过工具发现了“Hot Spots”，终于可以利用我们以前学过的源码级的优化技巧来对源代码进行优化了。且慢，我们大家都知道 `i++;` 的效率要比 `i=i+1;` 高，`i>>1;` 效率要比 `i/2;` 高等。不过坦白地说，目前主流的编译器都会自动地把 `i/2;` 转换成 `i>>1;` 的，所以如果查看它们生成的汇编语言代码，你会发现这两段

代码生成的汇编语言代码都是一样的，但是对程序员来说，`i/2;` 能更好地表达出开发者的本意，而 `i>>1;` 这一段代码却会让一个新程序员一头雾水。这么来说，这种优化就没什么搞头了。不仅如此，一个好的编译器会做很多方面的优化，涵盖了很多内容，所以选择一个好的编译器并打开对应的优化开关进行编译，比自己手动优化更有效率。

就算是编译器没有做优化，比如把一个函数调用变成直接的一段调用，虽然避免了函数调用的栈操作代价，但也明显违反了模块化的原则。另外有些优化手段也是不可移植的，所以说优化问题是一个双刃剑，有利就有弊。

说到这里，如果你依然对优化和效率非常执着，那我不得不佩服你了。反正我一般编程序的时候主要就是考虑思路和正确性，很少考虑优化的问题。

14.2 C 语言的使用原则

对于刚刚接触编程的学生，有的老师建议不应该首先教授 C 语言，而是应该教授 Python 等脚本语言，如果就是为了教会同学们编写程序，C 语言确实有点难了，简单明了的脚本语言可能更适合。我不反对这种观点，但是我有点儿不同的意见。C 语言之所以难，是因为它要解决更难的问题，所以必须更底层、更灵活，也因此它必须引入指针、位运算等。如果问题本身很简单，那么使用 C 语言来完成也可以像使用 Python 语言来完成一样简单，这取决于使用方法和使用策略。

所以，使用 C 语言的一个最基本的原则就是尽量使用“简单的 C 语言”，不要过分炫耀和使用一些不常用的特性和技巧，这些特性和技巧也许根本不适合解决你的问题，而且还可能引入潜在的、难以发现的 bug，造成“C 语言很难”这种错误认识。

例如，如果想通过定义一个宏来完成交换两个变量的功能，貌似简单，但这绝对不是一个简单任务。你也可以用 C 语言支持的位操作中的异或操作交换两个变量，这

种晦涩的写法确实可以让人印象深刻，但是印象深刻并不总是褒义词。用一个临时变量通过三次赋值来完成交换，这种简单的方法虽然不酷，但是一般不会出错。

所以，我建议你把精力放到那些需要 C 语言解决的问题上，而不是放在语言本身。“简单的 C 语言”只是一个原则，同样的概念是“Keep simple，keep stupid”。这个原则说白了就是用最简单的方法达到自己的目的。C 语言中，实施这一原则有很多具体的建议，例如不写长表达式，如果表达式很长就用临时变量；多用括号确定运算优先级；一个函数只做一件事；定义指针变量时就初始化。最关键的是，要养成用简单办法解决问题的习惯，这样可以规避很多 C 语言程序的错误。

同时，你最好能保留一些经典的代码片段（snippet）。在本书中，4.7 节中的位操作宏，以及 8.5 节中的模拟扑克的洗牌程序，都是比较有价值的 snippet。学习这种 snippet 可以提高水平，开阔思路；保留这种经典的 snippet 可以方便今后进行复用。

14.3　加深对 C 语言的理解

如果你对本书的所有知识都已经掌握，而且书中的所有源代码你都亲自编写并实验了一遍，那么我相信，你可以比较流畅地使用 C 语言了。

但是别高兴得太早。当我完成这本书的时候，我并没有“荡胸生层云”的成就感，而是有点小恐惧。这不是故作姿态的谦虚，而是真实的感受。伴随着不断地查资料，不断地总结，不断地看到各种奇思妙想，我越来越感到 C 语言的博大精深，越来越感到自己的无知。学习就像一个不断扩大的圈，随着圈子的不断扩大，才会知道圈外的未知世界原来这么大！

一直到目前为止，我们都是在使用 C 语言，但是并不理解 C 语言。我告诉大家要这样用，不要那样用，但是很少解释为什么要这样用或者为什么不那样用。林语堂曾经说过，“只用一样东西，不明白他的道理，实在不高明”，说的就是我们这种状态了。

那如何才能变得“高明”呢？别忘了，C 语言要经过编译器的编译，才会变成可执行程序。所以有的时候，你是如何理解 C 语言的并不重要，编译器是如何理解 C 语言的才是终极的答案。

例如，程序 14-1 中有两个函数 f1 和 f2。f1 函数中，printf 并没有打印出数组 a 的长度，而是只打印出一个 4。这是因为编译器会把传入这个函数的 a 数组变量转变为一个指针变量。所以 sizeof(a)只会打印出一个指针变量的尺寸，那就是 4。理解了这一点，就可以理解第二个函数 f2 中的代码。如果不是定义数组时的初始化，C 语言不允许对一个数组变量直接赋值，但是程序 14-1 中第 7 行却可以，这也是因为这个时候 str 已经是一个指针变量了。

程序 14-1　编译器对程序的理解

```
1  f1(char a[10]){
2      int i = sizeof(a);
3      printf("%d\n", i);
4  }

5  int f2(char str[]) {
6      if(str[0] == '\0')
7      str = "none";
8      ...
9  }
```

对编译原理的学习和理解，可以帮助我们真正地从核心理解 C 语言，而不再被语言本身的外衣所蒙蔽。

本书在第 10 章浓墨重彩地介绍了指针，貌似写的天花乱坠，其实我内心真的很发虚。因为我根本不知道编译器看到 int *p 这个语句后，是如何理解它并转换成对应的汇编语言的。这部分知识就像是一个被红盖头盖住了的新娘，让我充满了好奇和冲动。本来想从编译器的角度来介绍一下指针，但是发现我的这一部分知识不足以把这个事情阐述清楚，可以这么说，这是本书的一个遗憾。如果本书有机会出第 2 版，我一定要把这一部分的知识加上去。

14.4　C、C++以及 C#（java）

江山代有才人出　各领风骚数百年，在计算机编程语言方面，短短的不到 50 年的时间里，主流的开发语言已经进化了三代，它们分别是 C 语言为代表的面向过程的语言，以及 C++为代表的面向对象的语言，以及以 Java 和 C#为代表的基于虚拟机的语言。

每一次的进化，都是对上一代语言的优点的继承，同时对其缺点又加以了改造和改进。C++语言继承了几乎所有 C 语言的语法和库函数，同时为了提高语言的封装，继承和多态，C++引入了类的概念。同时，也引进了第 13 章中我们介绍过的异常处理。在 C#和 java 中，完全继承了 C++的面向对象的设计思想，同时放弃了 C++和 C 中备受指责的指针，而采用虚拟机的方式来自动地管理和回收内存。在一些细节方面，C#语言更是针对 C 语言的缺点进行了革新，例如，C#语言会自动检测数组越界，同时 C#语言引入 decimal 用于需要更多有效位的浮点数应用，C#还取消头文件以避免 C 语言的重复包含的问题等。

有趣的是，虽然有针对性地对上一代语言的缺点加以改造，但是目前这三代语言并没有完全取代的意思，而是形成了一种相安无事，优势互补的一种局面。我们在第 2 章也介绍过，这三种语言所占的份额也基本持平。这是因为所谓的缺点，在另外一种情景下，可能就是优点。就像指针，虽然会造成很多麻烦，但是处理一些数据结构的时候，还是最方便和最有效率的工具。虚拟机虽然帮助你自动管理内存，但是它使得程序的运行速度下降，虽然对虚拟机进行了各种优化和改进，但是 C#语言至少要比 C++慢一半，有的时候甚至更多。所以对这三代语言，正确的态度就是充分了解每种语言的特性，然后在正确的场景下，使用正确的语言。一般情况下，C#或 java 主要用于编写直接面向用户的各种（GUI）应用程序，C++多用于开发各种后台使用的算法

和逻辑库。而 C 语言则更底层，主要用于开发更核心的算法和靠近硬件的各种驱动程序和控制程序等。

这虽然是一本写 C 语言的书，但我必须承认，C#和 Java 目前使用得比 C 和 C++语言广泛，而且比起 C++也容易学习和使用。但是别忘了，好学就意味着 C#和 Java 的程序员要比 C++的程序员多。所以有一天你发现公司招聘 C/C++程序员，你的第一个问题就是，他为什么不去招一个人数更多的 Java 程序员？这个问题的答案有两个，第一，他要更快；第二，他要更底层。这两点 C#和 Java 都不胜任。所以说，作为一个 C/C++程序员，你完全有资格要一个高价钱。

作为一个合格的程序员，我认为应该很好的掌握这三代语言才行。大家也不用感到害怕和沮丧。如果让你学三门外语，你可能觉得很困难，不同的外语之间差别太大了。但是这三代计算机语言，差别并不是你想的那么大。尤其是一些有关语言的基本的概念，其实根本没有变。一旦你学会了一门语言，那么其他的另外两门语言，很快也就融会贯通了。例如，在 C 语言中，我们介绍了函数指针的概念，如果你把这个概念理解透彻了，你会发现 C#语言中的代理（delegate），其实就是一个函数指针。反之，如果你根本没学过函数指针，C#语言中的代理（delegate）就会让你感到非常的难以理解了。所以说如果你把一门语言学通，其他的语言一个礼拜搞定，也并不是什么难事。从这个角度来说，C 语言确实是一门很好的入门语言，虽然比较难，但是如果学会了，就等于一下子学了三门语言，而且还可以通过这门语言把很多基本的计算机概念搞明白，实在是性价比超高啊！

当然，你也别高兴的太早。C++中的面向对象的设计思想和设计模式，以及 C#后面的.Net 平台，都是恢弘巨大，高深莫测的内容。都需要扎实的了解和掌握才能成为 C++和 C#领域的专家。当你面向社会的时候，你会发现，工业界其实并不太需要会这些语言的人，他们真正需要的是这些语言的专家。工业界面临的问题，要远远复杂于你的考试题。所以请记住，路漫漫其修远兮，吾将上下而求索。

14.5　我们现在在哪里

这本书你能看到这里，说明多少你还比较喜欢这本书的。如果你能明白本书的大部分内容，那么恭喜你，你现在已经是一只程序猿，并且已经爬上了 C/C++的这棵大树了。这个大树枝干挺拔，树冠茂盛。你可能不知道你在这棵大树的什么位置，只是抬头一望，看到的都是其他程序猿的红屁股，这多少让你感到不太舒服，于是你决定继续往上爬，但是向哪个方向爬呢？

图 14-1 标出了你现在的位置。下面我简单地对这个图进行一下解释。

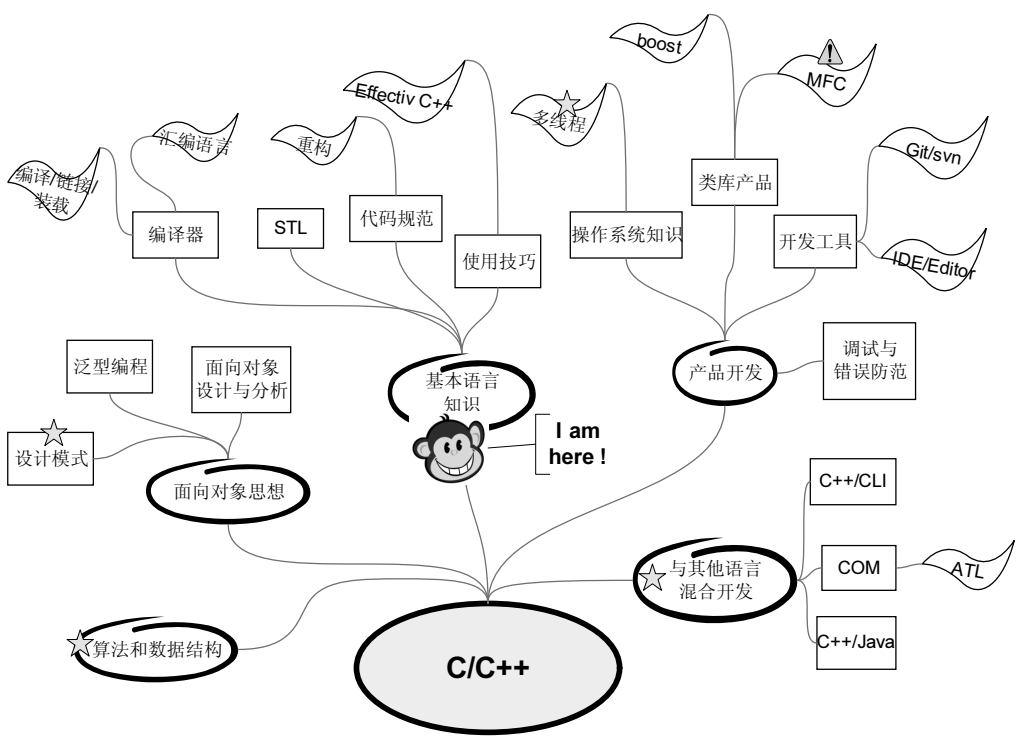

图 14-1　程序猿现在的位置

- 这个图没有任何权威性，它只是一个参考。事实上，它完全基于我个人的理解，所以一定有遗漏和不准确的地方。
- 不要太乐观，以为学完本书，你就很厉害了。你也看到了，这其实是颗大树。每一片叶子相关的书摞起来都比你高。作为一只程序猿，我曾经不太安分地每一个树枝都上去看过，但也都是一知半解，比你懂更多不敢说，屁股比你看得多是一定的！
- 也不要太悲观。这棵树虽然很大，但是事实上，这个树上的知识都是相关的。如果你能把本书理解得很好，你会发现学习 C++会比较容易。一旦学精了设计模式，你会发现 MFC 和 STL 也并不是太难。所以，踏踏实实地一点一点学习，慢慢地融会贯通。
- C/C++是更快、更底层的语言，想高效使用 C/C++语言还需要更多的算法和数据结构的知识，需要更多的操作系统和多线程的知识。尤其是操作系统的知识，因为程序最终要运行在操作系统上。例如，如何读取一个文件的时间属性，如何读取键盘的特定键，如何建立一个目录等，这些都与程序运行的平台（操作系统）有紧密的关系。对这些知识了解得越多，就越能编写出高效、简洁的 C/C++语言程序。如果要编写一个大规模的程序，还需要设计模式的知识。这些我都在图中用星号进行了标注。
- 上节提到过，一般大中型的程序都是混合利用多种语言来完成的。这样就可以充分发挥各种语言的优点。例如，为了得到更好的界面，一般都会采用 C#和 Java 进行开发，而核心的算法和靠近硬件的部分都需要采用 C/C++语言编写。所以你也要知道一些与其他语言混合开发的知识，例如 C++/CLI，或者是 COM 组件的知识。这一部分我也用星号进行了标注。值得一提的是 MFC，它目前的地位比较尴尬，用来编内核有点太复杂，用来编界面有点太简单。我个人并不太看好它的未来。

14.6　计算机领域的继续学习

首先恭喜你，经过了一段时间的学习，你终于到达了这里，本书的最后一节。如果你是从头读到这里的，而不是顺手翻到这里的话，说明你还喜欢本书，我很开心，谢谢你！

下面的文字来源于我的一个学生在人人网上发表的一个博客，我的那位学生很聪明并且有些天分，别人的毕业设计都是网站什么的，他的毕业设计却是发明一种全新的语言，并且开发了这种语言的编译器。他曾经利用人人网的一个漏洞成功攻击过人人网，并最终被人人网封号。如果你立志把程序员当成终身职业，你喜欢这个职业，并且愿意在这个职业上不断进步并提高，那么下面的观点对你来说有一定的参考意义。

现在的编程语言，以至编程世界，被诸君有意无意地神化了。

我只会 C++，虽然我写过一些脚本语言和本机语言的编译器，但基本上我会的就只有 C++，用的也只有 C++。我并不觉得很乏味，因为会一门编程语言就够了，无论它是什么。我不会 Python，但是你要我用 Python 的时候，我可能在十几分钟内看看语法，查看 API 就可以写出相关的程序；我也不会 PHP，但要写网页，我还是看看语法，查看 API 就可以写出来，十几分钟的事。如果我觉得有爱，我还会实现这些语言的编译器，这对我是一件很容易的事情（毕竟写了好多个了）。

可见，编程语言只是工具罢了，纯粹的工具。学会一门语言并不像你想的那么难，看看它的简明教程和语法，再看看它的例子，我相信你可以学会这门语言。C++虽然是一门庞大的语言，但绝不是现在人们口中谈虎色变的东西，它是很靠谱的编程语言，无论是性能、库、还是 IDE，都是齐全的。

也许你会问，现在不都是说用 Vim、Notepad++、Emacs 什么的吗，Visual Studio 是不是俗气了些？那是初学者都不理会的东西吧？IDE 什么的别开玩笑了，我又不是大一的小孩。

也许你会问，Linux 和 Mac 才真正是酷的吧？现在只有初级用户才用 Windows 吧？也许你会问很多很多诸如此类、被误导的问题，原因是现在的编程语言和编程世界被神化了。本来朴质的工具，被渲染上神秘主义的面纱，让众人觉得那些无关痛痒的东西是我们需要的。

我可以负责任地告诉你，我的想法是：编程语言是什么都无所谓，编程工具是什么也都无所谓，编程所在系统是什么根本没关系；真正有用的是算法和设计模式。算法和设计模式才是编程的根本。只要明白这两个，其他都是浮云般的存在。

算法和设计模式是独立于上述的一切而存在的。无论用 C++还是 Python，或者用 JavaScript，算法该咋实现还咋实现，它是程序能力和效率的保证；而设计模式也是同样的道理，无论用 Linux，还是 GitHub，只要明白设计模式，都能设计出很规范的、相对很鲁棒、有利于后续开发的程序。

大学中说：“事有始终，物有本末，知其前后，则近道矣”。然而，现在人在给初学者意见、甚至是自己在学习的时候，不但不从根本的算法和设计模式入手，反而扯出一堆皮毛的东西，还形成了阵营，相互挖苦和嘲笑，这本身不是很奇怪的么？让那些本来应该得到重视的智慧被无视，让那些无关痛痒的技巧被学习，从古至今像这样而成功的人，我没有听说过。

如果你是初学者，现在迷茫于或者迷惑于这些建议的话，我劝你静下心来，不要被这个时代的喧嚣和浮躁所感染。你需要做下面这些事情。

1）把基础的计算机结构学好（计算机组成原理、体系结构）。

2）把数据结构学好，也要掌握一些比较高级的数据结构，每种数据结构自己都

动手去做一下，形成一个自己的数据结构小类库，以后对你绝对会有用。

3）把操作系统的基本知识学好，不是 Linux，也不是 Windows，是那些并发、调度、缓存机制、文件系统等算法性的东西。这些东西在以后绝对会用得上，并不是在你实现操作系统的时候，而是在你写一些稍底层的结构的时候。

4）算法这东西可以说是无穷无尽的。首先把基础算法弄明白，比如动态规划、贪婪、分支限界此类的经典算法，然后随着兴趣去学更多有意思有用的算法。如果喜欢智能、自然语言处理，可以去尝试看看机器学习的书，然后动手实现一个机器学习小类库。这个类库未必用，也未必能让别人用，写它的最重要的意义在于理解那些算法。

5）致力于设计模式。算法是超脱的，是理性的。要让计算机执行这个算法，必须化为程序，那就必然用到编程。无论是什么语言，如果不会设计模式，即便你对这门语言再熟悉，也不可能设计出优秀的程序。所以设计模式在程序设计的时候是必须的，也是很重要的。

6）蔑视那些沉浸在神秘主义编程论里的人吧！在明白了上面那些后，你自己就可以实现编程语言、编程工具甚至是编程用到的操作系统。然后告诉他们：“too young，too naïve”。

我学生的文章转载完毕，我希望你能明白我把这篇文章放到这里的最终目的。我并不是要求以后我们每个人都去学算法和设计模式，毕竟每个人的天赋并不一样。这里我真正想传达的是：**找到你感兴趣的领域，在这个领域不断地深入，并最终成为这个领域的专家**。至于这个领域是什么，可大可小，可方可圆。我曾经亲眼见过一个人用 Excel 软件设计出了令人目眩的界面。虽然他不会什么 C 语言和算法，但是我依然相信他是专家。正所谓“领域万变，但精神唯一!”。