期末答疑与总结 再次审视学习编译原理的作用

你好，我是宫文学。到这里，咱们这门课程的主要内容就要结束了。有的同学在学习课程的过程中呢，提出了他感兴趣的一些话题，而我自己也会有一些想讲的话题，这个我也会在后面，以加餐等方式再做一些补充。接下来，我还会给你出一套期末测试题，帮你检测自己在整个学习过程中的所学所得。

那么，在今天这一讲，我们就来做个期末答疑与总结。在这里，我挑选了同学们提出的几个有代表性的问题，给你解答一下，帮助你更好地了解和掌握本课程的知识内容。

问题1：学习了编译原理，对于我学习算法有什么帮助？

> @无缘消受人间富贵：老师，想通过编译器学算法，单独学算法总是不知道有什么意义，每次都放弃，老师有什么建议吗？但是看到评论说用到的都是简单的数据结构，编译器用不到复杂的数据结构和算法？

针对这位同学提出的问题，我想谈一谈我对算法学习的感受。

前一阵，我在跟同事聊天时，提到了一个观点。我说，大部分的程序员，其实从来都没写过一个像样的算法。他们写的程序，都是把业务逻辑简单地翻译成代码。那么，如果一个公司写出来的软件全是这样的代码，就没有什么技术壁垒了，很容易被复制。

反之，一些优秀的软件，往往都是有几个核心的算法的。比如，对于项目管理软件，那么网络优化算法就很关键；对于字处理软件，那么字体渲染算法就很关键，当年方正的激光照排系统，就是以此为基础的；对于电子表格软件，公式功能和自动计算的算法很关键；对于视频会议系统，也必须掌握与音视频有关的核心算法。这样，因为有了算法的技术壁垒，很多软件就算是摆在你的面前，你也很难克隆它。

所以说，作为一名软件工程师，你就必须要有一定的算法素养，这样才能去挑战那些有难度的软件功能。而作为一个软件公司，其实要看看自己在算法上有多少积淀，这样才能构筑自己的技术壁垒。

那么，编译原理对于提升你的算法素养，能带来什么帮助呢？我给你梳理一下。

编译原理之所以硬核，也是因为它涉及了很多的算法。

在**编译器前端**，主要涉及到的算法有3个：

- **有限自动机构造算法**：这是在讲词法分析时提到的。这个算法可以根据正则文法，自动生成有限自动机。它是正则表达式工具的基础，也是像grep等强大的Linux命令能够对字符串进行模式识别的核心技术。
- **LL算法**：这是在讲自顶向下的语法分析时涉及的。根据上下文无关文法，LL算法能够自动生成自顶向下的语法分析器，中间还涉及对First和Follow集合的计算。
- **LR算法**：这是在讲自底向上的语法分析时涉及的。根据上下文无关文法，LR算法能自动生成自底向上的语法分析器，中间还涉及到有限自动机的构造算法。

总的来说，编译器前端的算法都是判断某个文本是否符合某个文法规则，它对于各种文本处理工作都很有效。有些同学也在留言区里分享，他在做全文检索系统时就会用到上述算法，使得搜索引擎能更容易地理解用户的搜索请求。

在**编译器后端**，主要涉及到的算法也有3个：

- **指令选择算法；**
- **寄存器分配算法；**
- **指令重排序（指令调度）算法。**

这三个算法也有共同点，它们都是寻找较优解或最优解，而且它们都是NP Complete（NP完全）的。简单地说，就是这类问题能够很容易验证一个解对不对（多项式时间内），但求解过程的效率却可能很低。对这类问题会采用各种方法求解。在讲解指令选择算法时，我介绍了**贪婪策略和动态规划**这两种不同的求解思路；而寄存器选择算法的图染色算法，则采用了一种**启发式算法**，这些都是求解NP完全问题的具体实践。

在日常工作中，我们其实也会有很多需要求较优解或最优解的需求。比如，在文本编辑软件中，需要把一个段落的文字分成多行。而如何分行，就需要用到一个这样的算法。再比如，当做一个报表软件，并且需要分页打印的时候，如何分页也是同类型的问题。

其他类似的需求还有很多。如果你没有求较优解或最优解的算法思路，对这样的问题就会束手无策。

而在**编译器的中端**部分，涉及的算法数量就更多了，但是由于这些算法都是对IR的各种分析和变换，所以IR采用不同的数据结构的时候，算法的实现也会不同。它不像前端和后端算法那样，在不同的编译器里都具有很高的一致性。

不过，IR基本上就是三种数据结构：树结构、图结构和基于CFG的指令列表。所以，这些算法会训练你处理树和图的能力，比如你可以在树和图中发现一些模式，以此对树和图进行变换，等等。这在你日常的很多编程工作中也是非常重要的，因为这两种数据结构是编程中最常使用的数据结构。

那么，总结起来，认真学好编译原理，一定会给你的算法素养带来不小的提升。

问题2：现代编程语言这么多，我们真的需要一门新语言吗？

> @蓝士钦：前不久看到所谓的国产编程语言“木兰”被扒皮后，发现是Python套层壳，真的是很气愤。想要掌握编译原理设计一门自己的语言，但同时又有点迷茫，现代编程语言这么多，真的需要再多一门新语言吗？从人机交互的角度来看，任何语言都是语法糖。

关于是否需要一门新语言的话题，我也想跟你聊聊我自己的看法，主要有三个方面。当然，你也可以在此过程中思考一下，看看有没有什么跟我不同的见解，欢迎与我交流讨论。

第一，编程语言其实比我们日常看到的要多，很多的细分领域都需要自己的语言。

我们平常了解的都是一些广泛流行的通用编程语言，而进入到每个细分领域，其实都需要各自领域的语言。比如SaaS的鼻祖Salesforce，就设计了自己的Apex语言，用于开发商业应用。华为的实验室在研发方舟编译器之前，也曾经研发了一门语言Cm，服务于DSP芯片的研发。

第二，中国技术生态的健康发展，都需要有自己的语言。

每当出现一个新的技术生态的时候，总是有一门语言会成为这个技术生态的“脚本”，服务于这个技术生态。比如，C语言就是Unix系统的脚本语言；JavaScript、Java、PHP等等，本质上都是Web的脚本语言；而Objective-C和Swift显然是苹果设备的脚本语言；Android虽然一开始用了Java，但最近也在转成Kotlin，这样Google更容易掌控。

那么，从这个角度看，当中国逐步发展起自己的技术生态的时候，也一定会孕育出自己的语言。以移动计算生态而言，我们有全球最大的移动互联网用户群和最丰富的应用，手机的制造量也是全球最高的。而位于应用和硬件之间的应用开发平台，我们却没有话语权，这会使中国的移动互联网技术生态受到很大的掣肘。

我在第40讲，也已经分析过了，Android系统经过了很多年的演化，但技术上仍然有明显的短板，使得Android平台的使用体验始终赶不上苹果系统。为了弥补这些短板，各个互联网公司都付出了很大的成本，比如一些头部应用的核心功能采用了C/C++开发。

并且，Android系统的编译器，在支持新的硬件上也颇为保守和封闭，让中国厂商难以参与。这也是华为之所以要做方舟编译器的另一个原因。因为华为现在自研的芯片越来越多，要想充分发挥这些芯片的能力，就必须要对编译器有更大的话语权。方舟编译器的问世，也证明了我们其实是有技术能力的，可以比国外的厂商做得更好。既然如此，我们为什么要受别人的制约？华为方舟编译器团队其实也很渴望，在方舟编译器之后推出自己的语言。至于华为内部是否已经立项，这就不太清楚了，但我觉得这是顺理成章的事情。

另外，除了在移动端的开发上会受到很多掣肘，在云端其实也一样。比如说，Java是被大量后端开发的工程师们所掌握的语言，但现在Java是被Oracle掌控的。你现在使用Java的时候，可能已经多多少少感受到了一种不愉快。先不说Java8之后的收费政策，就说我们渴望的特性（如协程、泛型中支持基础数据类型等），一直没有被满足，就会感觉不爽。

我在讲到协程的时候，就指出Java语言目前支持协程其实是很别扭的一种状态，它都是一些第三方的实现，并没有官方的支持。而如果Java的技术生态是由我们主导，可能就不是这样了。因为我国互联网的并发用户数如此之多，我们对更好的并发特性其实是更关切的。到目前为止，像微信团队解决高并发的问题，是用C++加上自己开发的协程库才实现的。而对于很多没有如此强大的技术能力的公司来说，就只能凑合了。

第三，实现一款优秀的软件，一定会用到编译技术。

每一款软件，当发展到极致的时候，都会变得像一个开发平台。这也是《黑客与画家》的作者保罗·格雷厄姆（Paul Graham）表达的思维。他原来的意思是，每个软件写到最后，都会包含一个Lisp的变种。实际上，他所要表达的意思就跟我说的一样。

我前一段时间，在北京跟某公司的老总探讨一个优秀的行业应用软件。这个软件在上世纪90年代就被开发出来了，也被我国广泛采用。一方面它是一个应用软件，另一方面它本身也是一个开发平台。所以它可以经过定制，满足不同行业的需求。

但是，我们国内的软件行业的情况是，在去客户那里实施的时候，几乎总是要修改源代码，否则就不能满足用户的个性化需求。

很多软件公司想去克隆一下我刚才说的那套软件，结果都放弃了。除了有对领域模型理解的困难以外，缺少把一个应用软件做成软件开发平台的能力，是其中很大的一个障碍。

实际上，目前在很多领域都是这样。国外的软件就是摆在那里，但中国的工程师就是做不出自己的来。而对于编译技术的掌握和运用，就是能够提升国内软件水平的重要途径。

我在开头跟同事交流的时候，也提出了软件工程师技术水平修养提升的几个境界。其中一个境界，就是要能够利用编译技术，做出在更大范围内具有通用性的软件。如果你能达到这个境界，那么也一定有更大的发展空间。

问题3：如何判断某门语言是否适合利用LLVM作为后端？

> @ヾ(◍°∇°◍)ﾉﾞ：老师，很多语言都声称使用LLVM提升性能，但是在Lua领域好像一直是LuaJIT无法超越？

这个问题涉及到了如何利用后端工具的问题，比较有代表性。

LLVM是一个通用的后端工具。在它诞生之初，首先是用于支持C/C++语言的。所以一门语言，在运行机制上越接近C/C++语言，用LLVM来做后端就越合适。

比如Rust用LLVM就很成功，因为Rust语言跟C/C++一样，它们的目标都是编写系统级的程序，支持各种丰富的基础数据类型，并且也都不需要有垃圾收集机制。

那么，如果换成Python呢？你应该记得，Python不会对基础数据类型进行细粒度的控制，不需要把整型区分成8位、16位、32位和64位的，它的整型计算可以支持任意长度。这种语义就跟C/C++的相差比较远，所以采用LLVM的收益相对就会小一些。

而对于JavaScript语言来说，浏览器的应用场景要求了编译速度要尽量地快，但在这方面LLVM并没有优势。像我们讲过的隐藏类（Shapes）和内联缓存（Inline Caching）这样的对JavaScript很重要的机制，LLVM也帮不上忙。所以，如果在项目时间比较紧张的情况下，你可以暂时拿LLVM顶一顶，Safari浏览器中的JavaScript引擎之前就这么干过。但是，要想达到最好的效果，你还是编写自己的后端更好一些。

那对于Lua语言，其实你也可以用这个思路来分析一下，是采用LLVM，还是自己写后端会更好一些。不过，由于Lua语言比较简单，所以实现后端的工作量应该也相对较小。

小结

这一讲，我主要回答了几个比较宏观的问题，它们都涉及到了编译原理这门课的作用。

第一个问题，我是从提升算法素养的角度来展开介绍的。编译原理知识里面涉及了大量的算法，我把它总结成了三大类，每类都有自己的特点，希望能对你宏观把握它们有所帮助。

第二个问题，其实是这门课程的一条暗线。我并没有在课程里去情绪化地鼓吹，一定要有自己的编译器、自己的语言。我的方式其实是想做一点具体的事情，所以在第二个模块中，我带着你一起探究了现有语言的编译器都是怎么实现的，破除你对编译器的神秘感、距离感；在第三个模块，我们又一起探讨了一下实现一门语言中的那些关键技术点，比如垃圾收集、并行等，它们都是如何实现的。

在课程最后呢，我又带你了解了一下具有中国血统的方舟编译器。我想说的是，其实我们不但能做出编译器和语言来，而且可能会做得更好。虽然我们对方舟编译器的分析还没有做完，但通过分析它的技术思路，你应该或多或少地感受到了它的优秀。所以，针对“我们真的需要一门新语言吗”这个问题，我的回答是确定的。并且，即使你不去参与实现一门通用的语言，在实现自己领域的语言，以及把自己的软件做得更具通用性这点上，编译原理仍然能发挥巨大的作用，对你的职业生涯也会有切实的帮助。

好，请你继续给我留言吧，我们一起交流讨论。同时我也希望你能多多地分享，做一个知识的传播者。感谢你的阅读，我们下一讲再见。