热点问题答疑 如何吃透7种真实的编译器?

你好，我是宫文学。

到这里，咱们就已经解析完7个编译器了。在这个过程中，你可能也积累了不少问题。所以今天这一讲，我就把其中有代表性的问题，给你具体分析一下。这样，能帮助你更好地掌握本课程的学习思路。

问题1：如何真正吃透课程中讲到的7种编译器?

在课程中，我们是从解析实际编译器入手的。而每一个真实的编译器里面都包含了大量的实战技术和知识点，所以你在学习的时候，很容易在某个点被卡住。那第一个问题，我想先给你解答一下，“真实编译器解析篇”这个模块的学习方法。

我们知道，学习知识最好能找到一个比较缓的坡，让自己可以慢慢爬上去，而不是一下子面对一面高墙。**那么对于研究真实编译器，这个缓坡是什么呢?**

我的建议是，你可以把掌握一个具体的编译器的目标，分解成四个级别的任务，逐步提高难度，直到最后吃透。

第一个级别，就是听一听文稿，看一看文稿中给出的示例程序和源代码的链接就可以了。

这个级别最重要的目标是什么？是掌握我给你梳理出来的这个编译器的技术主线，掌握一张地图，这样你就能有一个宏观且直观的把握，并且能增强你对编译原理的核心基础知识点的认知，就可以了。

小提示：关于编译器的技术主线和知识地图，你可以期待一下在期中复习周中，即将发布的“划重点：7种编译器的核心概念和算法”内容。

在这个基础上，如果你还想再进一步，那么就可以挑战第二级的任务。

第二个级别，是要动手做实验。

你可以运行一下我给出的那些使用编译器的命令，打印输出调试信息，或者使用一下课程中提到的图形化工具。

比如，在Graal和V8编译器，你可以通过修改命令行的参数，观察生成的IR是什么样子。这样你就可以了解到，什么情况下才会触发即时编译、什么时候才会触发内联优化、生成的汇编代码是什么样子的，等等。

这样，通过动手做练习，你对这些编译器的认识就会更加具体，并且会有一种自己可以驾驭的感觉，赢得信心。

第三个级别，是看源代码，并跟踪源代码的运行过程，从而进入到编译器的内部，去解析一个编译器的真相。

完成这一级的任务，对你动手能力的要求更高。你最容易遇到的问题，是搭建一个调试环境。比如，调试Graal编译器要采用远程调试的模式，跟你调试一个普通应用还是不大一样的。而采用GDB、LLDB这样的工具，对很多同学来说可能也是一个挑战。

而且，你在编译源代码和调试的过程中也会遇到很多与配置有关的问题。比如，我用GDB来调试Julia和MySQL的时候，就发现最好是使用一个Linux虚拟机，因为macOS对GDB的支持不够好。

不过，上述困难都不是说真的有多难，而是需要你的耐心。遇到问题就解决问题，最终搭建出一个你能驾驭的环境，这个过程也会大大提升你的动手实践能力。

环境搭建好了，在跟踪程序执行的过程中，一样要需要耐心。你可能要跟踪执行很多步，才能梳理出程序的执行脉络和实现思路。我在课程中建议的那些断点的位置和梳理出来程序的入口，可以给你提供一些帮助。

可以说，只要你能做好第三级的工作，终归是能吃透编译器的运行机制的。这个时候，你其实已经差不多进入了高手的行列。比如，在实际编程工作中，当遇到一个特别棘手的问题的时候，你可以跟踪到编译器、虚拟机的内部实现机制上去定位和解决问题。

而我前面说了，掌握一个具体的编译器的目标，是有四个级别的任务。那你可能要问，都能剖析源代码了，还要进一步挑战什么呢？

这第四个级别呢，就是把代码跟编译原理和算法结合起来，实现认识的升华。

在第三级，当你阅读和跟踪程序执行的时候，会遇到一个认知上的挑战。对于某些程序，你每行代码都能看懂，但为什么这么写，你其实不明白。

像编译器这样的软件，在解决每一个关键问题的时候，肯定都是有理论和算法支撑的。这跟我们平常写一些应用程序不大一样，这些应用程序很少会涉及到比较深入的原理和算法。

我举个例子，在讲Java编译器中的语法分析器的时候，我提到几点。第一，它是用递归下降算法的；第二，它在避免左递归时，采用了经典的文法改写的方法；第三，在处理二元表达式

时，采用了运算符优先级算法，它是一种简单的LR算法。

我提到的这三点中的每一点，都是一个编译原理的知识点或算法。如果对这些理论没有具体的了解，那你看代码的时候就看不出门道来。类似的例子还有很多。

所以，如果你其实在编译原理的基础理论和算法上都有不错的素养的话，你会直接带着自己的假设去代码里进行印证，这样你就会发现每段程序，其实都是有一个算法去对应的，这样你就真的做到融会贯通了。

那如何才能达到第四级的境界，如何才能理论和实践兼修且互相联系呢?

- 第一，你要掌握“预备知识”模块中的编译原理核心基础知识和算法。
- 第二，你要阅读相关的论文和设计文档。有一些论文是一些经典的、奠基性的论文。比如，在讲Sea of Nodes类型的IR的时候，我介绍了三篇重要的论文，需要你去看。还有一些论文或设计文档是针对某个编译器的具体的技术点的，这些论文对于你掌握该编译器的设计思路也很有帮助。

达到第四级的境界，你其实已经可以参与编译器的开发，并能成为该领域的技术专家了。针对某个具体的技术点加以研究和钻研，你也可以写出很有见地的论文。

当然，我不会要求每个同学都成为一个编译器的专家，因为这真的要投入大量的精力和实践。你可以根据自己的技术领域和发展规划，设定自己的目标。

我的建议是:

1. 首先，每个同学肯定要完成第一级的目标。这一级目标的要求是能理解主线，有时候要多读几遍才行。
2. 对于第二级目标，我建议你针对2~3门你感兴趣的语言，上手做一做实验。
3. 对于第三级目标，我希望你能够针对1门语言，去做一下深入探索，找一找跟踪调试一个编译器、甚至修改编译器的源代码的感觉。
4. 对于第四级目标，我希望你能够针对那些常见的编译原理算法，比如前端的词法分析、语法分析，能够在编译器里找到并理解它们的实现。至于那些更加深入的算法，可以作为延伸任务。

总的来说呢，“真实编译器”这个模块的课程内容，为你的学习提供了开放式的各种可能性。

好，接下来，我就针对同学们的提问和课程的思考题，来做一下解析。

问题2：多重分派是泛型实现吗?

> @d：“多重分派能够根据方法参数的类型，确定其分派到哪个实现。它的优点是容易让同一个操作，扩展到支持不同的数据类型。”宫老师，多重分派是泛型实现吗？

由于大多数同学目前使用的语言，采用的都是面向对象的编程范式，所以会比较熟悉像这样的一种函数或方法派发的方式：

```
Mammal mammal = new Cow();  //Cow是Mammal的一个子类
mammal.speak();
```

这是调用了mammal的一个方法：speak。那这个speak方法具体指的是哪个实现呢？根据面向对象的继承规则，这个方法可以是在Cow上定义的。如果Cow本身没有定义，就去它的父类中去逐级查找。所以，**speak()具体采用哪个实现，是完全由mammal对象的类型来确定的。**这就是单一分派。

我们认为，mammal对象实际上是speak方法的第一个参数，虽然在语法上，它并没有出现在参数列表中。而Java的运行时机制，也确实是这么实现的。你可以通过查看编译生成的字节码或汇编代码来验证一下。你如果在方法中使用“this”对象，那么实际上访问的是方法的0号参数来获取对象的引用或地址。

在采用单一分派的情况下，对于二元（或者更多元）的运算的实现是比较别扭的，比如下面的整型和浮点型相加的方法，你需要在整型和浮点型的对象中，分别去实现整型加浮点型，以及浮点型加整型的计算：

```
Integer a = 2;
Float b = 3.1;
a.add(b);    //采用整型对象的add方法。
b.add(a);    //采用浮点型对象的add方法。
```

但如果再增加新的类型怎么办呢？那么所有原有的类都要去修改，以便支持新的加法运算吗？

多重分派的情况，就不是仅仅由第一个参数来确定函数的实现了，而是会依赖多个参数的组合。这就能很优雅地解决上述问题。在增加新的数据类型的时候，你只需要增加新的函数即可。

```
add(Integer a, Float b);
add(Float b, Integer a);
add(Integer a, MyType b);   //支持新的类型
```

不过，这里又有一个问题出现了。如果对每种具体的类型，都去实现一个函数的话，那么实现的工作量也很大。这个时候，我们就可以用上泛型了，或者叫参数化类型。

通过泛型的机制，我们可以让相同的实现逻辑只书写一次。在第三个模块“现代语言设计篇”中，专门有一讲给你进一步展开**泛型的实现机制**，到时你可以去深入学习下。

问题3：安全点是怎么回事？为什么编译器生成的某些汇编代码我看不懂？

> @智昂张智恩震：请问老师，和JVM握手就是插入safepoint的过程吗？具体的握手是在做什么？

你在查看编译器生成的汇编代码的时候，经常会看到一些辅助性的代码逻辑。它们的作用不是要把你的代码翻译成汇编代码才生成的，而是要去实现一些运行时机制。

我举几个例子。

第一个例子，是做逆优化。比如V8中把函数编译成机器码，是基于对类型的推断。如果实际执行的时候，编译器发现类型跟推断不符，就要执行逆优化，跳转到解释器去执行。这个时候，你就会看到汇编代码里有一些指令，是用于做逆优化功能的。

第二个例子，是在并行中会遇到的抢占式调度问题。协程这种并发机制，是应用级的并发。一个线程上会有多个协程在运行。但是，如果其中一个协程的运行时间很长，就会占据太多的计算资源，让这个线程上的其他协程没有机会去运行。对于一些比较高级的协程调度器，比如Go语言的调度器，就能够把一个长时间运行的协程暂停下来，让其他协程来运行。怎么实现这种调度呢？那就要编译器在生成的代码里，去插入一些逻辑，配合调度器去做这种调度。

第三个例子，是垃圾收集。根据编译器所采用的垃圾收集算法，在进行垃圾收集时，可能会做内存的拷贝，把一个对象从一个地方拷贝到另一地方。这在某些情况下，会导致程序出错。比如，当你读一个Java对象的成员变量的值的时候，生成的汇编代码会根据对象的地址，加上一定的偏移量，得到该成员变量的地址。但这个时候，这个对象的地址被垃圾收集器改变了，那么程序的逻辑就错了。所以在做垃圾回收的时候，相关的线程一定要停在叫做“安全点（safepoint）”的地方，在这些地方去修改对象的地址，程序就不会出错。

@智昂张智恩震 同学提出的问题，就针对垃圾收集这种场景的。在Java生成的汇编代码里，程序要在安全点去跟运行时做一下互动（握手）。如果需要的话，当前线程就会被垃圾收集器停下，以便执行垃圾收集操作。

所以你看，只有了解了一门语言的运行时机制，才能懂得为什么要生成这样的代码。关于垃圾收集和并发机制，我也会在第三个模块中跟你去做进一步的探讨。

问题4：SSA只允许给变量赋一次值，循环中的变量是多次赋值的，不是矛盾了吗？

> @qinsi：关于思考题，SSA只允许给变量赋一次值，如果是循环的话就意味着要创建循环次数那么多的临时变量了？

@qinsi 同学问的这个问题其实挺深入，也很有意思。

是这样的。我们在做编译的时候，大部分时候是做静态的分析，也就是仅仅基于程序从词法角度（Lexically）的定义，不看它运行时的状态。**注意**，我这里说的词法，不是指词法分析的词法，而是指程序文本中体现的“使用和定义”（use-def）关系、控制流等。词法作用域（Lexical Scope）中的词法，也是同一个意思。

所以，SSA中说的赋值，实际上是对该变量（或称作值）做了一个定义，体现了变量之间的“使用和定义”（use-def）关系，也就是体现了变量之间的数据依赖，或者说是数据流，因此可以用来做数据流分析，从而实现各种优化算法。

小结

这一讲的答疑，我首先帮你梳理了学习真实世界编译器的方法。一个真实的编译器里涉及的技术和知识点确实比较多，但有挑战就有应对方法。我给你梳理了四级的学习阶梯，你探索内容的多少，也可以根据自己的需求和兴趣来把握。按照这个学习路径，你既可以去做一些宏观的了解，也可以在某个具体点上去做深入，这是一个有弹性的学习体系。

另外，我也挑了几个有意思的问题做了解答，在解答中也对涉及的知识点做了延伸和扩展。其中一些知识点，我还会在第三个模块中做进一步的介绍，比如垃圾收集机制、并发机制，以及泛型等。等你学完第三个模块，再回头看实际编译器的时候，你的认知会再次迭代。

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。

上一页 下一页