35 案例总结与热点问题答疑：后端部分真的比前端部分难吗？

本节课，我会继续剖析一些，你们提出的，有代表性的问题（以后端问题为主），主要包括以下几个方面：

- 后端技术部分真的比前端技术部分难吗？
- 怎样更好地理解栈和栈桢（有几个同学提出的问题很好，有必要在这里探究一下）？这样，你对栈桢的理解会更加扎实。
- 有关数据流分析框架。数据流分析是后端技术的几个重点之一，需要再细化一下。
- 关于Java的两个知识点：泛型和反射。我会从编译技术的角度讲一讲。

接下来，进入第一个问题：后端技术真的难吗？正确的学习路径是什么？

后端技术真的难吗？该怎么学？

有同学觉得，一进到后端，难度马上加大了，你是不是也有这样的感觉？我承认，前端部分和后端部分确实不太相同。

前端部分偏纯逻辑，你只要把算法琢磨透就行了。**而后端部分，**开始用到计算机组成原理的知识，要考虑CPU、寄存器、内存和指令集，甚至还要深入到CPU内部，去看它的流水线结构，以便理解指令排序。当然，我们还要说清楚与操作系统的关系，操作系统是如何加载代码并运行的，如何帮你管理内存等等。另外，还涉及ABI和调用约定，NP完全的算法等等。**看上去复杂了很多。**

虽然比较复杂，但我认为，这并不意味着后端更难，只意味着知识点更多。可这些知识，往往你熟悉了就不难了。

比如，**@风**同学见到了汇编代码说：总算遇到了自己熟悉的内容了，不用天天看Java代码了。

我觉得，从算法的角度出发，后端部分的算法，至少没比前端的语法分析算法难。而且有些知识点，别的课程里应该讲过，如果你从以下三个方面多多积累，会更容易掌握后端内容：

- 计算机组成原理：CPU的运行原理、汇编指令等等。

- 数据结构和算法，特别是与树和图有关的算法：如果你之前了解过，与图有关的算法，了解旅行商问题，那么会发现，指令选择等算法似曾相识。自然会理解，我提到某些算法是NP完全的，是什么意思。
- 操作系统：大部分情况下，程序是在操作系统中运行的，所以，要搞清楚我们编译的程序是如何跟操作系统互动的。

@沉淀的梦想就对这些内容，发表过感触：感觉学编译原理，真的能够帮助我们贯通整个计算机科学，涉及到的东西好多。

确实如他所说，那么我也希望《编译原理之美》这门课，能促使你去学习另外几门基础课，把基础夯实。

后端技术的另一个特点，是它比较偏工程性，不像前端部分有很强的理论性，对于每个问题有清晰的答案。而后端技术部分，往往对同一个问题有多种解决思路和算法，不一定有统一的答案，甚至算法和术语的名称都不统一。

后端技术的工程性特点，还体现在它会涉及很多技术细节，这些细节信息往往在教科书上是找不到的，必须去查厂商（比如Intel）的手册，有时要到社区里问，有时要看论文，甚至有时候要看源代码。

总的来说，如何学好后端，**我的建议主要有三个方面：**

- 学习关联的基础课程，比如《数据结构与算法》，互相印证；
- 理解编译原理工程性的特点，接受术语、算法等信息的不一致，并从多渠道获得前沿信息，比如源代码、厂商的手册等等。
- 注重实操，亲自动手。比如，你在学优化算法时，即使没时间写算法，也要尽可能用LLVM的算法做做实验。

按照上面三条建议，你应该可以充分掌握后端技术了。当然，如果你只是想做一个概要的了解，那么阅读文稿也会有不错的收获，因为我已经把主线梳理出来了，能避免你摸不着头脑，不知如何入手。

接下来，我们进入第二个问题：再次审视一下栈桢。

再次认识栈桢

@刘强同学问：操作系统在栈的管理中到底起不起作用？

这是操作系统方面的知识点，但可以跟编译技术中栈的管理联系在一起看。

我们应用程序能够访问很大的地址空间，但操作系统不会慷慨地，一下子分配很多真实的物理内存。操作系统会把内存分成很多页，一页一页地按需分配给应用程序。**那么什么时候分配呢?**

当应用访问自己内存空间中的一个地址，但实际上没有对应的物理内存时，就会导致CPU产生一个PageFault（在Intel手册中可以查到），这是一种异常（Exception）。

对异常的处理跟中断的处理很相似，会调用注册好的一个操作系统的例程，在内核态运行，来处理这个异常。这时候，操作系统就会实际分配物理内存。之后，回到用户态，继续执行你的程序，比如，一个push指令等等。整个过程对应用程序是透明的，其实背后有CPU和操作系统的参与。

@风提出了关于栈桢的第二个问题：看到汇编代码里管理栈桢的时候，用了rbp和rsp两个寄存器。是不是有点儿浪费？一个寄存器就够了啊。

确实是这样，用这种写法是习惯形成的，其实可以省略。而我在34讲里，用到的那个foo函数，根本没有使用栈，仅仅用寄存器就完成了工作。这时，可以把下面三行指令全部省掉：

```
pushq %rbp
movq %rsp, %rbp
popq %rbp
```

从而让产生的机器码少5个字节。最重要的是，还省掉两次内存读写操作（相比对寄存器的操作，对内存的操作是很费时间的）。

实际上，如果你用GCC编译的话，可以使用-fomit-frame-pointer参数来优化，会产生同样的效果，也就是不再使用rbp。在访问栈中的地址时，会采用4(%rsp)、8(%rsp)的方式，在rsp的基础上加某个值，来访问内存。

@沉淀的梦想提出了第三个问题：栈顶（也就是rsp的值）为什么要16字节对齐?

这其实是一个调用约定。是在GCC发展的过程中，形成的一个事实上的标准。不过，它也有一些好处，比如内存对齐后，某些指令读取数据的速度会更快，这会让你产生一个清晰的印象，每次用到栈桢，至少要占16个字节，也就是4个32位的整数的空间。那么，如果把一些尾递归转化为循环来执行，确实会降低系统的开销，包括内存开销和保存前一个桢的bsp、返回地址、寄存器的运行时间开销。

而**@不的**问了第四个问题： 为什么要设计成区分调用者、被调用者保护的寄存器，统一由被调用者或者调用者保护，有什么问题么?

这个问题是关于保护寄存器的，我没有仔细去研究它的根源。**不过我想，这种策略是最经济的。**

如果全部都是调用者保护，那么你调用的对象不会破坏你的寄存器的话，你也要保护起来，那就增加了成本；如果全部都是被调用者保护，也是一样的逻辑。如果调用者用了很少几个寄存器，被调用者却要保护很多，也不划算。

所以最优的方法，其实是比较中庸主义的，两边各负责保护一部分，不过，我觉得这可以用概率的方法做比较严谨的证明。

关于栈桢，我最后再补充一点。有的教材用活动记录这个术语，有的教材叫做栈桢。你要知道这两个概念的联系和区别。活动记录是比较抽象的概念，它可以表现为多种实际的实现方式。在我们的课程中，栈桢加上函数调用中所使用的寄存器，就相当于一个活动记录。

讲完栈桢之后，再来说说与数据流分析框架有关的问题。

细化数据流分析框架

数据流分析本身，理解起来并不难，就算不引入半格这个数学工具，你也完全可以理解。

对于数据流分析方法，不同的文献也有不同的描述，有的说是3个要素，有的说是4个要素。而我在文稿里说的是5个要素：方向（D）、值（V）、转换函数（F）、相遇运算（meet operation, Λ）和初始值（I）。你只要把这几个问题弄清楚，就可以了。

引入半格理论，主要是进一步规范相遇运算，这也是近些年研究界的一个倾向。用数学做形式化地描述虽然简洁清晰，但会不小心提升学习门槛。如果你只是为了写算法，完全可以不理半格理论，但如果为了方便看这方面算法的论文，了解半格理论会更好。

首先，半格是一种偏序集。偏序集里，某些元素是可以比较大小的。但怎么比较大小呢？其实，有时是人为定的，比如，{a, b}和{a, b, c}的大小，就是人为定的。

那么，既然能比较大小，就有上界（Upper Bound）和下界（Lower Bound）的说法。给定偏序集P的一个子集A，如果A中的每个元素a，都小于等于一个值x（x属于P），那么x就是A的一个上界。反过来，你也知道什么是下界。

半格是偏序集中，一种特殊的类型，它要求偏序集中，每个非空有限的子集，要么有最小上界（并半格，join-semilattice），要么有最大下界（交半格，meet-semilattice）。

其实，如果你把一个偏序集排序的含义反过来，它就会从交半格转换成并半格，或者并半格转换成交半格。我们还定义了两个特殊值：Top、Bottom。在不同的文献里，Top和Bottom有时刚好是反着的，那是因为排序的方向是反着的。

因为交半格和并半格是可以相互转化的，所以有的研究者采用的框架，就只用交半格。交半格中，集合{x, y}的最大下界，就记做x Λ y。在做活跃性分析的时候，我们就规定{a, b} > {a, b, c}

就行了，这样就是个交半格。如果按照这个规矩，我在28讲中举的那个常数传播的例子，应该把大小反过来，也做成个交半格。文稿中的写法，实际是个并半格，不过也不影响写算法。

这样讲，你更容易理解了吧？现在你再看到不同文献里，关于数据流分析中的偏序集、半格的时候，应该可以明白是怎么回事了。

最后，我再讲讲关于Java的两个知识点：泛型和反射。这也是一些同学关注的问题。

Java的两个知识点：泛型和反射

泛型机制大大方便了我们编写某些程序，不用一次次做强制类型转换和检查了。比如，我们要用一个String类型的List，就声明为：

```
List<String> myList;
```

这样，你从myList中访问一个元素，获取的自然就是一个String对象，而不是基类Object对象。

而增加泛型这个机制其实很简单。它只是在编译期增加了类型检查的机制，运行期没有任何改变。List 和List 运行的字节码都是完全相同的。

那么反射机制呢？它使我们能够在运行期，通过字符串形式的类名和方法名，来创建类，并调用方法。这其实绕过了编译期的检查机制，而是在运行期操纵对象：

```
//获取Class
Class<?> clazz = Class.forName("MyClass");
//动态创建实例
Object obj = clazz.newInstance();
//获取add方法的引用
Method method = clazz.getMethod("add",int.class,int.class);
//调用add方法
Object result = method.invoke(obj,1,4);
```

这样能带来很多灵活性，方便你写一些框架，或者写IDE。

从编译技术的角度看，实现反射很容易。因为在32讲中，你已经了解了字节码的结构。当时，我比较侧重讲指令，其实你还会看到它前面的，完整的符号表（也就是记录了类名、方法名等信息）。正因为有这些信息，所以反编译工具能够从字节码重新生成Java的源文件。

所以，虽然在运行时，Java类已经编译成字节码了，但我们仍然可以列出它所有的方法，可以实例化它，可以执行它的方法（因为可以查到方法的入口地址）。**所以你看，**一旦你掌握了底层机制，理解上层的一些特性就很容易了。

课程小结

编译器的后端技术部分也告一段落了。我们用16讲的篇幅，涵盖了运行时机制、汇编语言基础知识、中间代码、优化算法、目标代码生成、垃圾收集、即时编译等知识点，还针对内存计算和Java的字节生码成做了两个练习，中间还一直穿插介绍LLVM这个工具。我之前就提到，实现一个编译器，后端的工作量会很大，现在你应该有所体会。

在这里，我也想强调，后端技术的工程性比较强，每本书所采用的术语和算法等信息，都不尽相同。在我们的课程中，我给你梳理了一条，比较清晰的脉络，你可以沿着这条脉络，逐步深化，不断获得自己的感悟，早日修炼成后端技术的高手！

在答疑篇的最后，我总结了一些案例，供你参考。

案例总结

第一批示例程序，与汇编代码有关，包括手写的汇编代码，以及从playscript生成汇编代码的程序。这部分内容，主要是打破你对汇编代码的畏惧心，知道它虽然细节很多，但并不难。在讲解后端技术部分时，我总是在提汇编代码，在34讲，我甚至写了一个黑客级的小程序，直接操作机器码。我希望经历了这些过程之后，你能对汇编代码亲切起来，产生可以掌握它的信心。

第二批示例程序，是基于LLVM工具生成IR的示例代码。掌握LLVM的IR，熟悉调用LLVM的API编程，能让你在写完前端以后，以最短的时间，拥有所有后端的功能。通过LLVM，你也会更加具体的体会，代码优化等功能。

第三批示例程序，是内存计算和字节码生成，这两个应用题目。通过这两个应用题目，你会体会到两点：

- 编译器后端技术对于从事一些基础软件的开发很有用；
- 虽然课程没有过多讲解Java技术，只通过一个应用篇去使用Java的字节码，但你会发现，我们对后端技术的基本知识，比如对中间代码的理解，都可以马上应用到Java语言上，得到举一反三的感觉。

一课一思

如果你在工作中真的接到了一个任务，要实现某编译器的后端，你觉得学过本课程以后，你敢接手这个任务吗？还有哪些地方是需要你再去补足的？你完成这个任务比较可靠的路径是什么？欢迎在留言区分享你的观点。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。