01 编译的全过程都悄悄做了哪些事情?

你好，我是宫文学。

正如我在开篇词中所说的，这一季课程的设计，是要带你去考察实际编译器的代码，把你带到编译技术的第一现场，让你以最直观、最接地气的方式理解编译器是怎么做出来的。

但是，毕竟编译领域还是有很多基本概念的。对于编译原理基础不太扎实的同学来说，在跟随我出发探险之前，最好还是做一点准备工作，磨刀不误砍柴工嘛。所以，在正式开始本课程之前，我会先花8讲的时间，用通俗的语言，帮你把编译原理的知识体系梳理一遍。

当然，对于已经学过编译原理的同学来说，这几讲可以帮助你复习以前学过的知识，把相关的知识点从遥远的记忆里再调出来，重温一下，以便更好地进入状态。

今天这一讲，我首先带你从宏观上理解一下整个编译过程。后面几讲中，我再针对编译过程中的每个阶段做细化讲解。

好了，让我们开始吧。

编译，其实就是把源代码变成目标代码的过程。如果源代码编译后要在操作系统上运行，那目标代码就是汇编代码，我们再通过汇编和链接的过程形成可执行文件，然后通过加载器加载到操作系统里执行。如果编译后是在解释器里执行，那目标代码就可以不是汇编代码，而是一种解释器可以理解的中间形式的代码即可。

我举一个很简单的例子。这里有一段C语言的程序，我们一起来看看它的编译过程。

```
int foo(int a){
  int b = a + 3;
  return b;
}
```

这段源代码，如果把它编译成汇编代码，大致是下面这个样子：

```
          .section        __TEXT,__text,regular,pure_instructions
          .globl        _foo                    ## -- Begin function foo
_foo:                                   ## @foo
          pushq   %rbp
          movq    %rsp, %rbp
          movl    %edi, -4(%rbp)
          movl    -4(%rbp), %eax
```

```
addl    $3, %eax
movl    %eax, -8(%rbp)
movl    -8(%rbp), %eax
popq    %rbp
retq
```

你可以看出，源代码和目标代码之间的差异还是很大的。那么，我们怎么实现这个翻译呢？

其实，编译和把英语翻译成汉语的大逻辑是一样的。前提是你要懂这两门语言，这样你看到一篇英语文章，在脑子里理解以后，就可以把它翻译成汉语。编译器也是一样，你首先需要让编译器理解源代码的意思，然后再把它翻译成另一种语言。

表面上看，好像从英语到汉语，一下子就能翻译过去。但实际上，大脑一瞬间做了很多个步骤的处理，包括识别一个个单词，理解语法结构，然后弄明白它的意思。同样，编译器翻译源代码，也需要经过多个处理步骤，如下图所示。

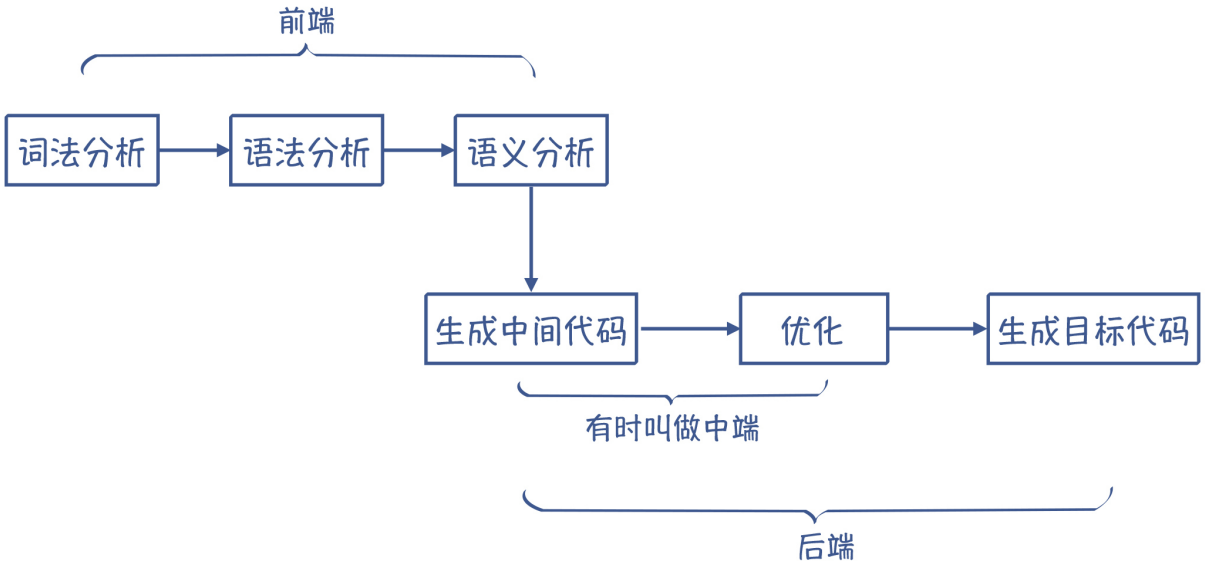


图1：编译的各个阶段

我来解释一下各个步骤。

词法分析（Lexical Analysis）

首先，编译器要读入源代码。

在编译之前，源代码只是一长串字符而已，这显然不利于编译器理解程序的含义。所以，编译的第一步，就是要像读文章一样，先把里面的单词和标点符号识别出来。程序里面的单词叫做Token，它可以分成关键字、标识符、字面量、操作符号等多个种类。**把字符串转换为Token的这个过程，就叫做词法分析。**

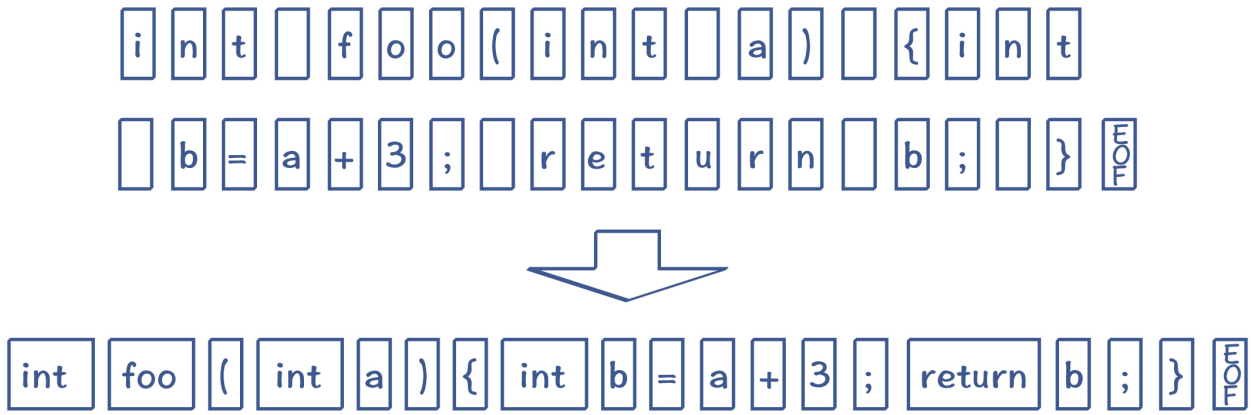


图2：把字符串转换为Token（注意：其中的空白字符，代表空格、tab、回车和换行符，EOF是文件结束符）

语法分析（Syntactic Analysis）

识别出Token以后，离编译器明白源代码的含义仍然有很长一段距离。下一步，**我们需要让编译器像理解自然语言一样，理解它的语法结构。**这就是第二步，**语法分析。**

上语文课的时候，老师都会让你给一个句子划分语法结构。比如说：“我喜欢又聪明又勇敢的你”，它的语法结构可以表示成下面这样的树状结构。

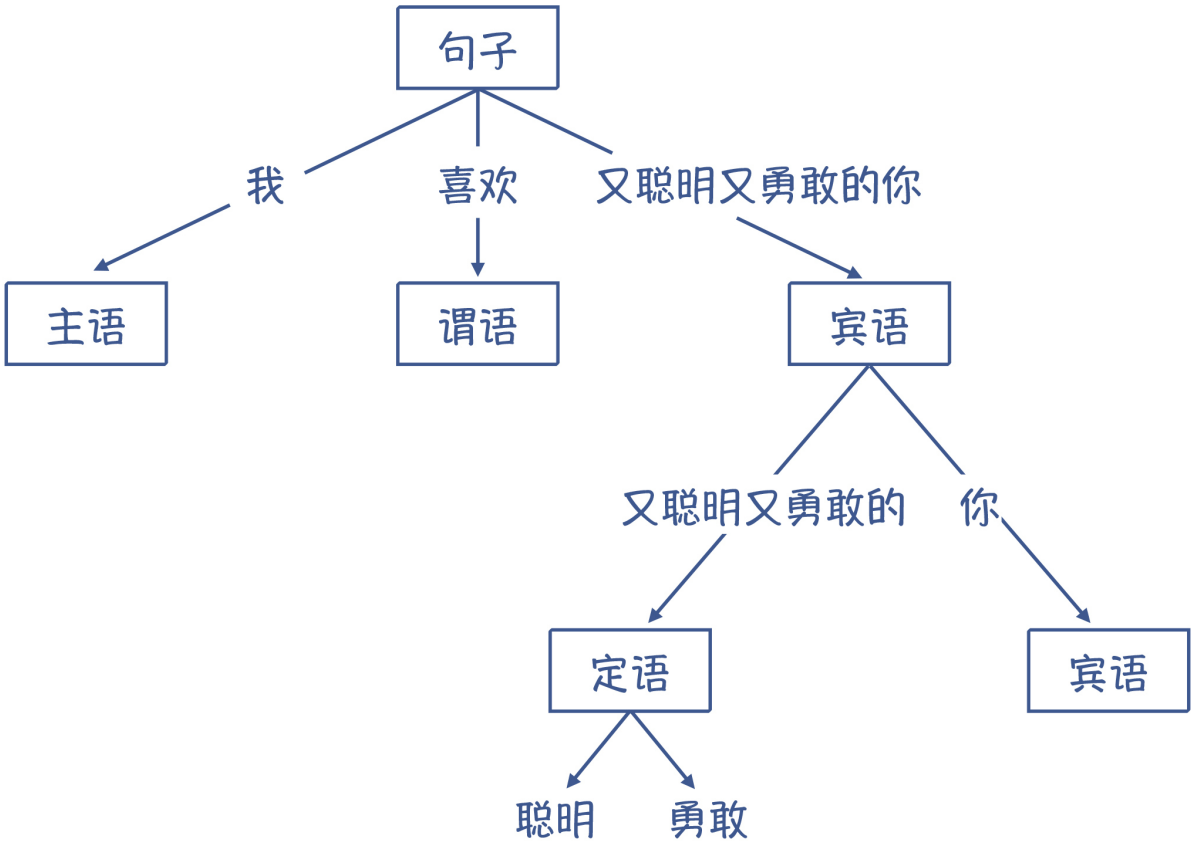


图3：把一个句子变成语法树

那么在编译器里，语法分析阶段也会把Token串，转换成一个**体现语法规则的、树状的数据结构**，这个数据结构叫做**抽象语法树**（AST，Abstract Syntax Tree）。我们前面的示例程序转换为AST以后，大概是下面这个样子：

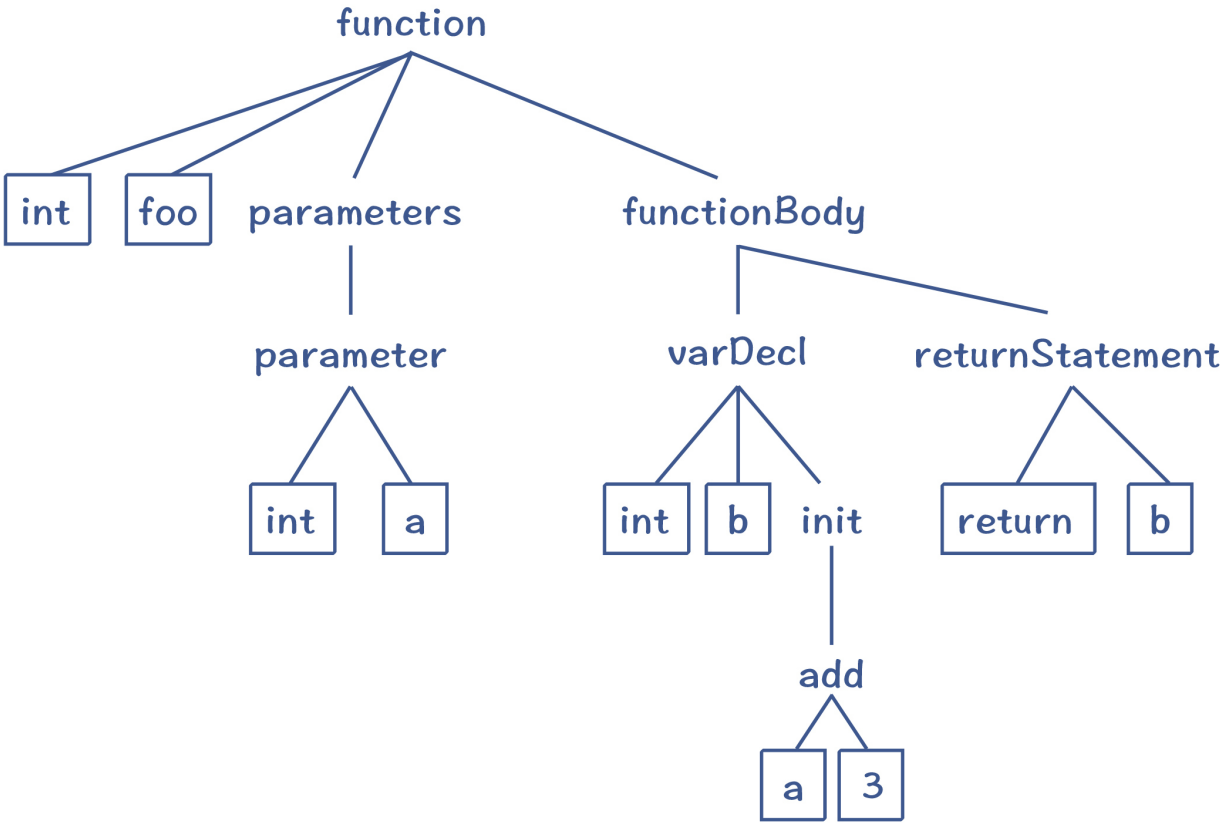


图4：foo函数对应的语法树

这样的一棵AST反映了示例程序的语法结构。比如说，我们知道一个函数的定义包括了返回值类型、函数名称、0到多个参数和函数体等。这棵抽象语法树的顶部就是一个函数节点，它包含了四个子节点，刚好反映了函数的语法。

再进一步，函数体里面还可以包含多个语句，如变量声明语句、返回语句，它们构成了函数体的子节点。然后，每个语句又可以进一步分解，直到叶子节点，就不可再分解了。而叶子节点，就是词法分析阶段生成的Token（图中带边框的节点）。对这棵AST做深度优先的遍历，你就能依次得到原来的Token。

语义分析（Semantic Analysis）

生成AST以后，程序的语法结构就很清晰了，编译工作往前迈进了一大步。但这棵树到底代表了什么意思，我们目前仍然不能完全确定。

比如说，表达式“a+3”在计算机程序里的完整含义是：“获取变量a的值，把它跟字面量3的值相加，得到最终结果。”但我们目前只得到了这么一棵树，完全没有上面这么丰富的含义。

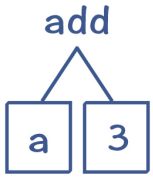


图5：a+3对应的AST

这就好比西方的儿童，很小的时候就能够给大人读报纸。因为他们懂得发音规则，能念出单词来（词法分析），也基本理解语法结构（他们不见得懂主谓宾这样的术语，但是凭经验已经知道句子有不同的组成部分），可以读得抑扬顿挫（语法分析），但是他们不懂报纸里说的是什么，也就是不懂语义。这就是编译器解读源代码的下一步工作，**语义分析**。

那么，怎样理解源代码的语义呢?

实际上，语言的设计者在定义类似“a+3”中加号这个操作符的时候，是给它规定了一些语义的，就是要把加号两边的数字相加。你在阅读某门语言的标准时，也会看到其中有很多篇幅是在做语义规定。在ECMAScript（也就是JavaScript）标准2020版中，Semantic这个词出现了657次。下图是其中加法操作的语义规则，它对于如何计算左节点、右节点的值，如何进行类型转换等，都有规定。

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumeric(*lprim*).
9. Let *rnum* be ? ToNumeric(*rprim*).
10. If Type(*lnum*) is different from Type(*rnum*), throw a **TypeError** exception.
11. Let *T* be Type(*lnum*).
12. Return *T*::add(*lnum*, *rnum*).

图6：ECMAScript标准中加法操作的语义规则

所以，我们可以在每个AST节点上附加一些语义规则，让它能反映语言设计者的本意。

- add节点：把两个子节点的值相加，作为自己的值；
- 变量节点（在等号右边的话）：取出变量的值；
- 数字字面量节点：返回这个字面量代表的值。

这样的话，如果你深度遍历AST，并执行每个节点附带的语义规则，就可以得到a+3的值。这意味着，我们正确地理解了这个表达式的含义。运用相同的方法，我们也就能够理解一个句子的含义、一个函数的含义，乃至整段源代码的含义。

这也就是说，AST加上这些语义规则，就能完整地反映源代码的含义。这个时候，你就可以做很多事情了。比如，你可以深度优先地遍历AST，并且一边遍历，一边执行语法规则。那么这个遍历过程，就是解释执行代码的过程。你相当于写了一个基于AST的解释器。

不过在此之前，编译器还要做点语义分析工作。**那么这里的语义分析是要解决什么问题呢？**

给你举个例子，如果我把示例程序稍微变换一下，加一个全局变量的声明，这个全局变量也叫a。那你觉得“a+3”中的变量a指的是哪个变量？

```
int a = 10;        //全局变量
int foo(int a){    //参数里有另一个变量a
  int b = a + 3;   //这里的a指的是哪一个？
  return b;
}
```

我们知道，编译程序要根据C语言在作用域方面的语义规则，识别出“a+3”中的a，所以这里指的其实是函数参数中的a，而不是全局变量的a。这样的话，我们在计算“a+3”的时候才能取到正确的值。

而把“a+3”中的a，跟正确的变量定义关联的过程，就叫做**引用消解**（Resolve）。这个时候，变量a的语义才算是清晰了。

变量有点像自然语言里的代词，比如说，“我喜欢又聪明又勇敢的你”中的“我”和“你”，指的是谁呢？如果这句话前面有两句话，“我是春娇，你是志明”，那这句话的意思就比较清楚了，是“春娇喜欢又聪明又勇敢的志明”。

引用消解需要在上下文中查找某个标识符的定义与引用的关系，所以我们现在可以回答前面的问题了，**语义分析的重要特点，就是做上下文相关的分析。**

在语义分析阶段，编译器还会识别出数据的类型。比如，在计算“a+3”的时候，我们必须知道a和3的类型是什么。因为**即使同样是加法运算，对于整型和浮点型数据，其计算方法也是不一样的。**

语义分析获得的一些信息（引用消解信息、类型信息等），会附加到AST上。这样的AST叫做**带有标注信息的AST**（Annotated AST/Decorated AST），用于更全面地反映源代码的含义。

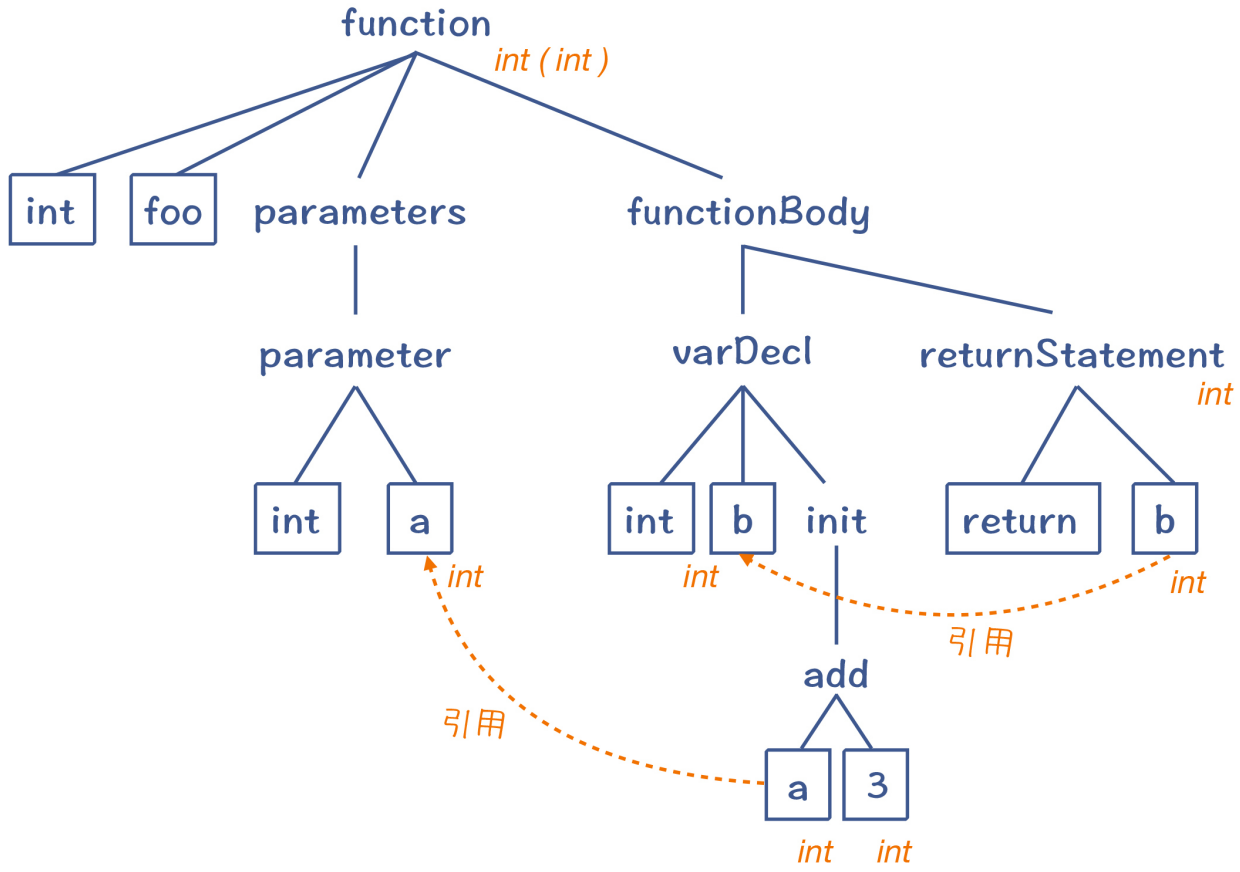


图7：带有标注信息的AST

好了，前面我所说的，都是如何让编译器更好地理解程序的语义。不过在语义分析阶段，编译器还要做很多语义方面的检查工作。

在自然语言里，我们可以很容易写出一个句子，它在语法上是正确的，但语义上是错误的。比如，“小猫喝水”这句话，它在语法和语义上都是对的；而“水喝小猫”这句话，语法是对的，语义上则是不对的。

计算机程序也会存在很多类似的语义错误的情况。比如说，对于“int b = a+3”的这个语句，语义规则要求，等号右边的表达式必须返回一个整型的数据（或者能够自动转换成整型的数据），否则就跟变量b的类型不兼容。如果右边的表达式“a+3”的计算结果是浮点型的，就违背了语义规则，就要报错。

总结起来，在语义分析阶段，编译器会做**语义理解和语义检查**这两方面的工作。词法分析、语法分析和语义分析，统称编译器的**前端**，它完成的是对源代码的理解工作。

做完语义分析以后，接下来编译器要做什么呢？

本质上，编译器这时可以直接生成目标代码，因为编译器已经完全理解了程序的含义，并把它表示成了带有语义信息的AST、符号表等数据结构。

生成目标代码的工作，叫做后端工作。做这项工作有一个前提，就是编译器需要懂得目标语言，也就是懂得目标语言的词法、语法和语义，这样才能保证翻译的准确性。这是显而易见的，只懂英语，不懂汉语，是不可能做英译汉的。通常来说，目标代码指的是汇编代码，它是汇编器（Assembler）所能理解的语言，跟机器码有直接的对应关系。汇编器能够将汇编代码转换成**机器码**。

熟练掌握汇编代码对于初学者来说会有一定的难度。但更麻烦的是，对于不同架构的CPU，还需要生成不同的汇编代码，这使得我们的工作量更大。所以，我们通常要在这个时候增加一个环节：先翻译成中间代码（Intermediate Representation，IR）。

中间代码（Intermediate Representation）

中间代码（IR），是处于源代码和目标代码之间的一种表示形式。

我们倾向于使用IR有两个原因。

第一个原因，是很多解释型的语言，可以直接执行IR，比如Python和Java。这样的话，编译器生成IR以后就完成任务了，没有必要生成最终的汇编代码。

第二个原因更加重要。我们生成代码的时候，需要做大量的优化工作。而很多优化工作没有必要基于汇编代码来做，而是可以基于IR，用统一的算法来完成。

优化（Optimization）

那为什么需要做优化工作呢？这里又有两大类的原因。

第一个原因，是源语言和目标语言有差异。源语言的设计目的是方便人类表达和理解，而目标语言是为了让机器理解。在源语言里很复杂的一件事情，到了目标语言里，有可能很简单地就表达出来了。

比如“I want to hold your hand and with you I will grow old.” 这句话挺长的吧？用了13个单词，但它实际上是诗经里的“执子之手，与子偕老”对应的英文。这样看来，还是中国文言文承载信息的效率更高。

同样的情况在编程语言里也有。以Java为例，我们经常为某个类定义属性，然后再定义获取或修改这些属性的方法：

```
Class Person{
  private String name;
  public String getName(){
    return name;
```

```
  }
  public void setName(String newName){
    this.name = newName
  }
}
```

如果你在程序里用“**person.getName()**”来获取Person的name字段，会是一个开销很大的操作，因为它涉及函数调用。在汇编代码里，实现一次函数调用会做下面这一大堆事情：

```
#调用者的代码
保存寄存器1    #保存现有寄存器的值到内存
保存寄存器2
...
保存寄存器n

把返回地址入栈
把person对象的地址写入寄存器，作为参数
跳转到getName函数的入口

#_getName 程序
在person对象的地址基础上，添加一个偏移量，得到name字段的地址
从该地址获取值，放到一个用于保存返回值的寄存器
跳转到返回地
```

你看了这段伪代码，就会发现，简单的一个**getName()方法**，开销真的很大。保存和恢复寄存器的值、保存和读取返回地址，等等，这些操作会涉及好几次读写内存的操作，要花费大量的时钟周期。但这个逻辑其实是可以简化的。

怎样简化呢？就是**跳过方法的调用**。我们直接根据对象的地址计算出name属性的地址，然后直接从内存取值就行。这样优化之后，性能会提高好多倍。

这种优化方法就叫做**内联**（inlining），也就是把原来程序中的函数调用去掉，把函数内的逻辑直接嵌入函数调用者的代码中。在Java语言里，这种属性读写的代码非常多。所以，Java的JIT编译器（把字节码编译成本地代码）很重要的工作就是实现内联优化，这会让整体系统的性能提高很大的一个百分比！

总结起来，我们在把源代码翻译成目标代码的过程中，没有必要“直译”，而是可以“意译”。这样我们完成相同的工作，对资源的消耗会更少。

第二个需要优化工作的原因，是程序员写的代码不是最优的，而编译器会帮你做纠正。比如下面这段代码中的**bar()函数**，里面就有多个地方可以优化。甚至，整个对bar()函数的调用，也可以省略，因为bar()的值一定是101。这些优化工作都可以在编译期间完成。

```
int bar(){
    int a = 10*10;  //这里在编译时可以直接计算出100这个值，这叫做“常数折叠”
    int b = 20;     //这个变量没有用到，可以在代码中删除，这叫做“死代码删除”
```

```
    if (a>0){        //因为a一定大于0，所以判断条件和else语句都可以去掉
        return a+1; //这里可以在编译器就计算出是101
    }
    else{
        return a-1;
    }
}
int a = bar();      //这里可以直接换成 a=101
```

综上所述，在生成目标代码之前，需要做的优化工作可以有很多，这通常也是编译器在运行时，花费时间最长的一个部分。

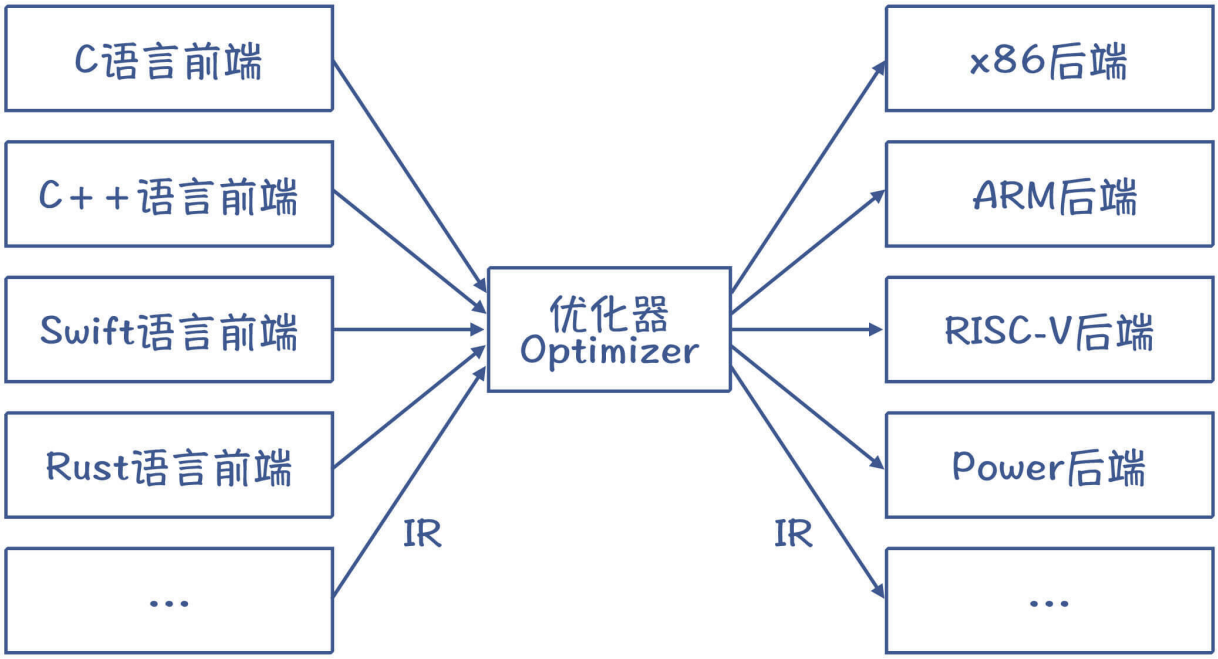


图8：多个前端和多个后端，可以采用统一的IR

而采用中间代码来编写优化算法的好处，是可以把大部分的优化算法，写成与具体CPU架构无关的形式，从而大大降低编译器适配不同CPU的工作量。并且，如果采用像LLVM这样的工具，我们还可以让多种语言的前端生成相同的中间代码，这样就可以复用中端和后端的程序了。

生成目标代码

编译器最后一个阶段的工作，是生成高效率的目标代码，也就是汇编代码。这个阶段，编译器也有几个重要的工作。

第一，是要选择合适的指令，生成性能最高的代码。

第二，是要优化寄存器的分配，让频繁访问的变量（比如循环变量）放到寄存器里，因为访问寄存器要比访问内存快100倍左右。

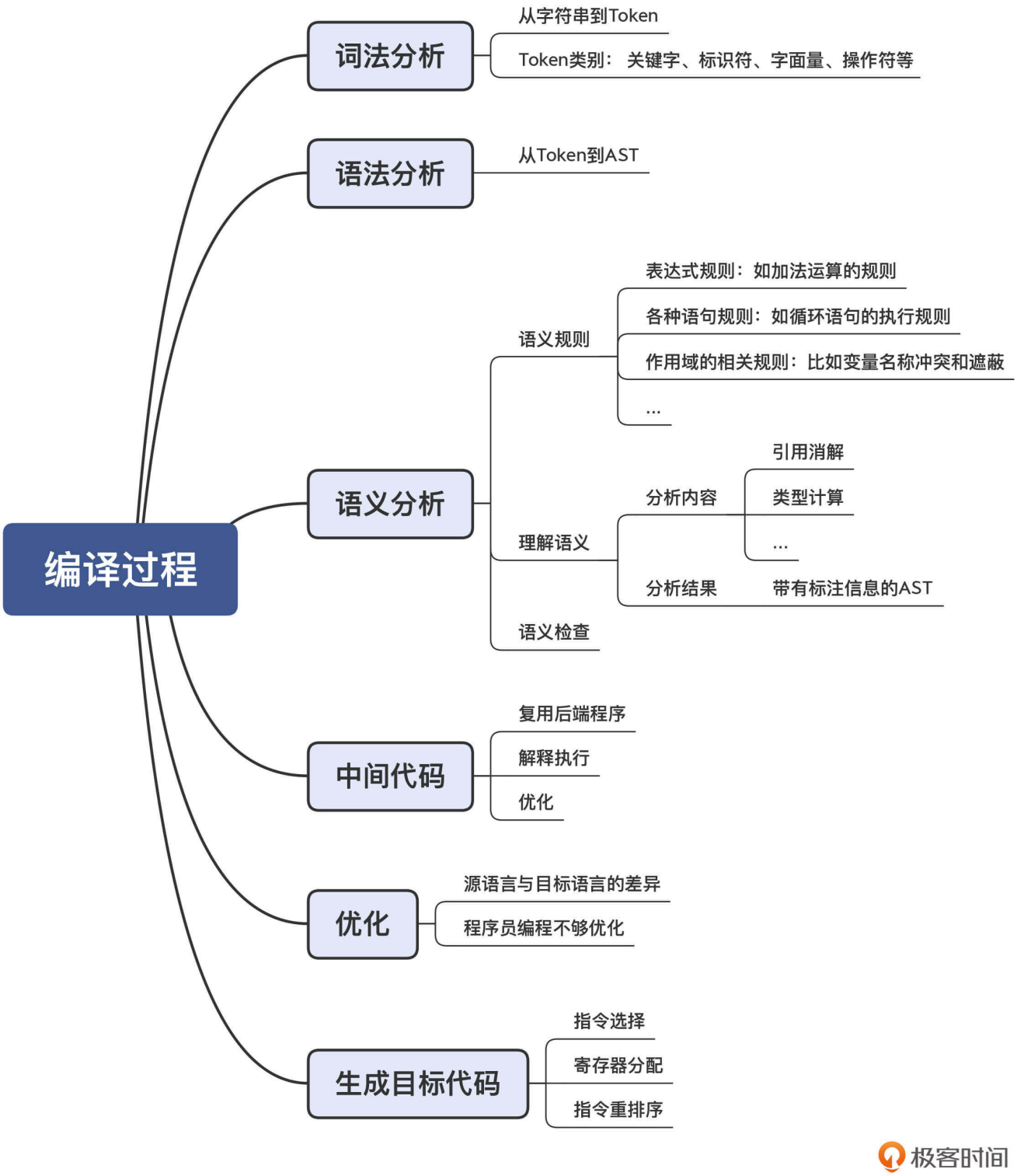
第三，是在不改变运行结果的情况下，对指令做重新排序，从而充分运用CPU内部的多个功能部件的并行计算能力。

目标代码生成以后，整个编译过程就完成了。

课程小结

本讲我从头到尾概要地讲解了编译的过程，希望你能了解每一个阶段存在的原因（Why），以及要完成的主要任务（What）。编译是一个比较复杂的过程，但如果我们能够分而治之，那么每一步的挑战就会降低很多。这样最后针对每个子任务，我们就都能找到解决的办法。

我希望这一讲能帮你在大脑里建立起一个概要的地图。在后面几讲中，我会对编译过程的各个环节展开讨论，让你有越来越清晰的理解。



一课一思

你觉得做计算机语言的编译和自然语言的翻译，有哪些地方是相同的，哪些地方是不同的？

欢迎在留言区分享你的见解，也欢迎你把今天的内容分享给更多的朋友。感谢阅读，我们下一讲再见。