Cqq解释器的实现和优化

李嘉图

2019 年 1 月 18 日

目录

1 概述

编译器通过词法分析、语法分析、中间代码生成、代码优化和指令选择等多个步骤，将高级语言翻译成等价的机器语言。而解释器往往是“直译式”的，直接解释执行高级语言，或将其翻译成相对底层的中间代码再解释执行。与编译相比，解释执行的优势有：

1. 省去了编译时间。当进行测试或程序需要经常更新时，使用解释执行可以减少编译耗时，提升效率。

2. 更加灵活。解释器在较高的抽象层面执行代码，往往可以支持一些复杂的运行时语言特性。

3. 良好的可移植性。解释语言不依赖于目标机的体系结构，从而便于移植到各种平台上去。

作为拓展实验，Cqq解释器是一个C++语言的简单解释器，其语法是C++一个很小的子集（下称Cqq语言），用来学习编译器、解释器实现和优化中的技术细节。

2 语法和约定

Cqq语言是C++语言一个很小的子集，其基本语法来源于一道题目《未来程序-改》，其语言规范大致如下：

1. 支持整数变量和整数数组，支持高维数组，其大小由int常量给出；

2. 支持全局变量
3. 支持`for`, `while`, `if`三种控制语句;
4. 支持函数调用和递归，函数参数只能为`int`类型，返回值只能为`int`类型;
5. 支持整数的基本运算，包括`+`, `-`, `*`, `/`, `%`, `^`;
6. 使用整数表示`bool`类型，即`0`表示`false`，非`0`表示`true`;
7. 支持逻辑运算，包括`&&`, `||`, `!`, `<`, `<=`, `>`, `>=`, `!=`, `==`;

其完整的描述可以在http://uoj.ac/problem/98上找到，这里不再赘述。除了上述内容，`Cqq`语言增加了如下的特性：

1. 增加常量表达式，数组下标可以由常量表达式定义，且常量表达式在运行之前完成计算。
2. 增加变量的初始化表达式，即`int Id = Expr`。
3. 为翻译到Q字节码后端增加了`break`语句的支持。
4. `for`循环语句可以定义循环变量，其在`for`循环语句拥有的作用域中。
5. 增加`&&`, `||`运算符的短路计算。
6. 支持简单的报错，语法错误会在出现错误处停止分析，语义错误会尽可能多的分析出包含的错误。
7. 支持单行注释

3 解释器结构

解释器使用了AST和基于栈的“Q字节码”作为中间代码。通过词法分析、语法分析、类型检查、代码生成、解释执行等步骤实现代码的解释，其结构可以由下图给出：

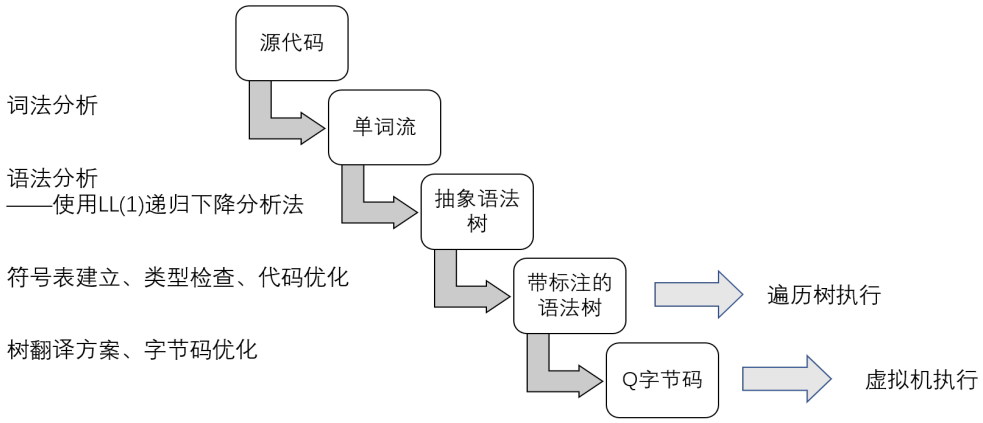


图 1: 解释器结构

源代码结构由下图给出：

图 2: 源代码结构

4 前端的实现

4.1 词法分析器

词法分析器是解释器的第一趟，它读入字符串形式的源代码，输出以$< type, value >$形式表示的单词流。实现词法分析器的一种经典方法是建立对应的有限状态自动机并利用自动机分析。`Cqq`解释器的词法分析器采

用了一种利用一个“向前看标记”的简单的方法：跳过所有的“空白”和注释，利用向前看标记判断应该什么内容。其基本工作原理为：

Listing 1: 词法分析器

```
1   while(isBlank(lookahead)) lookahead = read()
2   switch(lookahead) {
3       case EOF: return EOF
4       case digit: return <number, getNumber()>
5       case alpha:
6           t = getWord()
7           if (t is keyword)
8               return <keyword, t>
9           else return <ident, t>
10      default: return getKeyword()
11  }
```

其中寻找关键字可以使用trie树高效的实现。由于标识符的长度并不很长，为了方便，Cqq中直接将标识符的所有前缀存放在哈希表中，每次直接查询对应的字符串是否是某个标识符的前缀。

另外，为了实现跳过注释，词法分析器将单行注释开始标记//当作一个标识符。一旦某次识别出了此标识符，跳过所有的内容，直到遇到一个换行符。

4.2 语法分析器

容易证明，Cqq语言的文法是LL(1)的，因此可以方便的实现递归下降的语法分析器。原题目《未来程序改》的题面中已经给定了文法，Cqq解释器也采用其给定的文法。Cqq解释器的语法分析部分使用L-翻译模式，将代码翻译成抽象文法树。

实现递归下降分析法的一个问题是处理左递归和左公因子。左公因子可以提出处理，例如产生式：

```
IfStmt :  IF '(' Expr ')' Stmt
IfStmt :  IF '(' Expr ')' Stmt ELSE Stmt
```

需要将左公因子IF '(' Expr ')' Stmt提出。另外，这里还会遇到悬空else问题。Cqq语言使用同C++一样的处理方法——每个else匹配最近

的if，即在分析到第一个Stmt之后，如果向前看标记是ELSE，那么使用第二条产生式，否则使用第一条产生式。

而对于左递归，形如：

```
Unit3 :  Unit3 '+' Unit2
```

常规的处理方式是将其改写成：

```
Unit3 :  Unit2 Rest
Rest :  '+' Unit2 Rest | ϵ
```

而改写后的文法完全可以用一个循环而非递归来处理。这样处理起来即自然又方便，不需要额外使用一个List记录所有的项并最后处理。

4.3 类型检查

由于解释器需要尽可能提高翻译效率，Cqq解释器的类型检查趟事实上完成了：变量表的建立，变量内存分配，常量表达式的计算，类型检查，树结构的替换等操作。

4.3.1 变量表和变量内存分配

文件scope.cpp实现了符号表和符号表栈。Cqq解释器使用多符号表组织，即维护一个符号表栈，全局作用域所对应的符号表在栈底，栈中维护当前扫描位置所有开作用域的表项。当一个新变量被定义时，首先将该变量标号，计算其在栈中的偏移量，然后在当前作用域内查找是否有同名变量，如果有则报错，否则将其放入栈顶的符号表中；当一个变量被引用时，从顶向下在符号表中查询，并记录其在栈中的偏移量和标号，以供解释运行时查找。

计算偏移量的简单方法是维护一个虚拟的“栈顶指针”。当一个新变量被定义时，为其分配栈顶的空间；当作用域关闭，即其中的所有变量“死亡”时，将栈顶指针回退，释放死亡变量的空间。最终一个函数所需栈空间的大小是分配过程中“栈顶指针”的最大偏移量。这样的分配方式既便于实现，又可以释放死亡变量的空间，节省空间并提高程序的空间局部性。

特别的，对数组的分配基本上和对变量的分配相同。在Cqq语言中，数组按行放置在栈空间中。

在实现变量相关处理时遇到了一个有趣的Bug：Cqq解释器用类Var表示一个变量，记录了其标号、偏移量、（如果是数组）数组每一维大小对应

的表达式。其中，数组每一维大小对应的表达式是vector<Tree*>类型，在最初的实现中Var的析构函数包含了对数组大小表达式的析构。但是，每一次引用点都会从变量表中找出这个变量，因而数组每一维大小对应的表达式被多次析构了。解决方法也非常简单：借鉴Rust中“所有权”的思想，让拥有这个变量的人，而非这个变量去释放空间。

4.3.2 常量表达式的计算

一个表达式如果在编译期可以完全确定，则他应该在运行之前就被计算。首先引入关键字const用来表达一个常变量。如果一个表达式只包含常变量和常数，那么这个表达式是常量表达式，可以在类型检查期间完成计算并将树上的整个子树替换成一个常数节点。

Cqq解释器会尝试利用常数化简表达式，如果一个二元运算符的其中一个表达式是常数而足以简化整个表达式的值，那么就会进行化简。下面的表给出了Cqq解释器会试图做的代数化简：

	+	-	*	/	%	AND	OR
0 Op x	x	.	0	0	0	0	x
1 Op x	.	.	x	.	.	x	1
x Op 0	x	x	0	err	err	0	x
x Op 1	.	.	x	x	.	x	1

此外，常量表达式还用来进行控制流的简化。如果一个控制语句的条件表达式是常量，那么这个控制语句就可以被优化成不含控制的语句。例如当if语句的表达式为常量且为0，那么可以将整个子树替换成其false branch以达到简化控制流的目的。

4.3.3 类型检查

类型检查在Cqq中较为简单，只需要检查数组和变量有没有错误使用即可。例如对变量做数组操作，对数组做运算等错误。这些操作比较简单，这里不再赘述。

4.3.4 树结构替换

由于树遍历解释器需要遍历整棵树来进行解释，而树上节点（为了做

检查和优化）附带了大量的无用信息，这导致树的结构不够紧凑，也需要做许多多余的计算。Cqq解释器在类型检查趟同时进行树结构的替换。例如，对一个变量的引用被替换为了更紧凑的Load节点，变量初始化被替换为了更紧凑的Assigner节点，删去所有变量定义语句。通过这些替换，可以使得后续的实现更加简洁，也使得树遍历解释器运行更加高效。

4.4 Break语句的支持

判断Break是否在一个循环内非常简单，考虑如何进行代码翻译。由于采用树翻译方案，考虑使用类似拉链-代码回填的思路支持Break语句。使用一个全局的栈记录待回填的代码。在循环语句开始时记录当前栈中元素的个数id，在循环语句结束后，将大于id的代码回填，即可支持Break语句的生成。

5 后端的实现

5.1 树遍历解释器

树遍历解释器是一个简单的后端，它通过深度优先遍历树解释执行AST。在Cqq语言中，树遍历解释器有下图所示的结构：

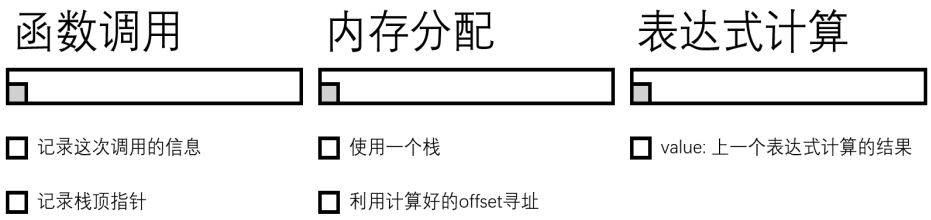


图 3: 树遍历解释器

解释器的抽象文法树框架使用Visitor模式设计，因此树遍历解释器对应于一个Visitor。树遍历解释器使用宿主语言的控制流和函数调用实现目标语言的控制流和函数调用，因此逻辑简单清晰，易于实现。但树遍历解释器也有很多不足：

1. 控制流不够灵活。例如实现goto，break的难度比较大。

2. 效率较低。因为树结构存储不够紧凑，且遍历树的过程中有许多调用、递归，运行效率不够高。

3. 无法做窥孔优化。因此相邻指令间有许多无用的存、取，这也影响程序的时空性能。

为了解决这些问题，不妨选择使用一种更加底层的中间表示：基于栈的字节码。这里用“Q字节码”来称呼Cqq解释器所使用的字节码。

5.2 Q字节码

Q字节码是一种基于栈的字节码，其运行在Q虚拟机上。Q虚拟机基础的指令集很小，包括运算指令（将栈顶的两个数取出做运算后放回栈顶）、存取指令（对栈的读写）、控制指令（跳转和条件跳转）、函数调用指令（申请栈空间和释放栈空间）和输入输出指令。

Q虚拟机没有寄存器和运算数栈，所有的操作都基于程序栈，其栈顶指针称为top。为了实现的方便，函数调用使用一个额外的调用栈，其栈顶指针称为sp。另外，Q虚拟机还有程序计数器pc和局部寻址指针fp。使用基于栈的字节码是一种折衷方案。它兼顾了翻译效率、运行效率和可移植性，并提升了对于寄存器和高速缓存的利用率。

5.3 运行时存储组织

由于Cqq语言并不支持堆分配，其所有的内存分配都在栈上。Q字节码使用了如下图所示的运行时存储组织：

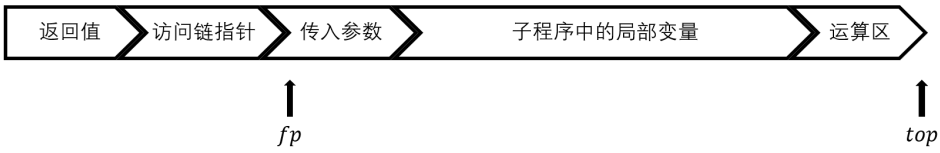


图 4: 运行时存储组织

当一个函数被调用时，会经历以下操作：

1. 在访问链处记录当前fp的位置

2. 将fp和top指向访问链指针的位置，依次计算每个参数

3. 跳转到函数的入口语句

这样实现函数调用方式统一了参数和局部变量的存储，即将参数视为定义在开头的一些变量，实现起来比较方便，同时节省了传参数的时间。

6 AST层面的优化

基于流图的数据流分析足够强大，但由于流图计算的复杂度稍高，有时往往并不能满足解释执行的需求。我们想知道在AST层面能做到什么程度的优化。使用AST层面优化的一个重要条件是：程序仅仅包含标准的控制流，即流图是可规约的。如果程序中包含跳转指令，其控制流无法被AST很好的描述，也就不能做更好的优化。

Cqq基于AST层面的循环优化的基本原则是：仅仅展开一层。即每个循环只处理直接包含于其中的代码。如果遵循这一点，我们可以在足够高的效率内对代码做一定程度的优化.

6.1 直接循环不变表达式

如果一个变量在循环中没有被定值，那么称其是循环不变的。循环不变的变量构成的表达式称为循环不变表达式。一个循环不变的表达式应当在循环的开头而不是循环中进行计算。特别的，如果一个数组在循环中未被定值，且这个数组的某个引用的所有下标也未被定值，那么这个引用也应当在开头被计算；否则，应当尽可能将所有循环不变的下标预先计算。

寻找循环中未被定值的变量，利用树上的数据结构，最好可以做到线性于代码长度的复杂度。由于通常的循环嵌套层数不大，且上面的方法在Visitor模式中实现难度和常数较大，Cqq解释器只实现了复杂度为$O(\sum k|Code|)$的算法，k为这段代码嵌套循环的深度。其基本思路是：维护一个全局的向量记录所有的定值点，在一个循环语句开始时记录向量中的元素个数，在循环结束后扫描向量以找到所有新加入的所有定值点。

6.2 全局变量的分析

如果一个函数可以直接调用另外一个函数，在他们之间连一条有向边，建出的图称为函数的调用图。在一个函数的执行时，可能访问这个函数和

其所有能到达的函数中所有的定值点。使用Tarjan算法将图的强连通分量缩为一个点，在获得的DAG上做状态压缩的动态规划，可以容易的知道每个变量是否可能被一个函数及其所有调用的函数定值。

那么一个全局变量是循环不变的，如果其不在循环中调用的函数的可能定值的变量集合之并。

6.3 寻找深层的循环不变量

下面描述了一个基于AST的更加强大的寻找循环不变量算法：

1. 寻找所有直接的循环不变表达式
2. 跳过所有控制语句(for, if, while)，扫描AST
   (a) 遇到引用点，加入引用点集合中
   (b) 遇到定值点，如果定值变量未被引用，且其定值表达式是循环不变的，将这个变量标记
3. 如果一个被标记的变量在整个循环中仅有一次定值，将其设为循环不变的。
4. 回到1，直到没有新变量被标记为了循环不变的。

由于实现起来比较困难，`Cqq`解释器中并没有实现这个算法。`Cqq`解释器实现了类似的寻找循环变量的算法。我们称一个变量是循环变量，如果其在循环中以形如$x \mapsto kx + b$的形式增长。

1. 跳过所有控制语句(for, if, while)，扫描AST
   (a) 遇到定值点，如果其定值方式形如$x = f(x)$，且$f(x)$具有$kx+b$的形式，将其标记
2. 如果一个标记的变量在整个循环中只被定值了一次，将其设为循环变量

很明显，这个算法也可以经过修改，利用迭代找到更深层次的循环变量，甚至找出所有的循环变量表达式。这里找出的循环变量可以为生成代码时做优化带来方便。如果实现深层次循环变量的寻找，由于对数组地址的访问往往具有这种形式，可以大大增加数组寻址的速度。

7 Q字节码层面的优化

朴素实现的Q字节码虚拟机并不能得到更好的性能，甚至比树遍历解释器还要慢。这也是符合直观的：树遍历解释器的控制流直接基于宿主语言，而Q字节码实现了对控制流的解释，其控制流效率比较差。但经过以下各种优化，可以大大提高执行效率。

7.1 控制流化简

当一个跳转语句跳转到一条无条件跳转语句，可以直接将前一个跳转语句的目标位置设为后一个跳转的目标。通过简单的迭代算法或建图后利用拓扑排序处理，可以很容易的做到这一点。以下伪代码提供了使用迭代算法做控制流化简的基本方法：

Listing 2: 控制流化简算法

```
1  do {
2      change = false;
3      for each branch instruction i
4          if (target(i) == goto) {
5              target(i) = target(target(i))
6              change = true
7          }
8  } while (change)
```

7.2 无用指令删除

直接生成的代码往往拥有许多无用的指令。为了方便，我们规定两种特殊的指令：

- 原地指令：如果一个指令不会修改除栈顶元素的任何位置，且不会移动栈指针。例如逻辑非指令和读取内存位置指令。
- 纯入栈指令：如果一个指令只会将一个元素入栈，而不会修改其他位置，称其为纯入栈指令。

当一个原地指令或纯入栈指令紧接着一个出栈指令，那么这两个指令就是无用的，可以直接删除。之所以大量生成了这样的代码，一个原因是赋值语句作为表达式，并没有任何人需要其返回值，需要将其从栈中删除。

另外，Q字节码中有通过相对基址位置获取绝对位置的指令`fptopool`。如果一个取绝对位置指令和一个从绝对位置中读取或向绝对位置存放的指令，可以将这两个指令删除，用一个从相对基址位置寻址的指令代替之。

7.3 寄存器和栈顶缓存

基于栈的虚拟机如果不能利用目标机寄存器，其操作成本会大大上升。我们希望尽可能利用寄存器对虚拟机进行优化。首先将指针`top`，`fp`，`pc`绑定到寄存器（实现中使用了GCC提供的内联汇编指令）。另外，可以使用所谓“栈顶缓存”的技术进行优化。

“栈顶缓存”的思想是，由于计算都发生在栈顶，如果将栈顶的若干个元素绑定到寄存器，那么就可以大大优化计算性能。Q虚拟机只将栈顶一个元素绑定到寄存器变量`ax`：即每次栈顶改变时存取内存放入寄存器，而计算时可以利用`ax`直接做运算，省去了存取内存的时间。当计算比较简单，特别是有大量原地指令时，这种策略可以带来很大的性能提升。

7.4 指令合并

由于基础的指令大多是零地址或一地址的，在做寻址和计算时往往需要多次入栈和出栈。这对于虚拟机的性能是致命的——虚拟机分发指令的成本很高，执行多条指令将大量的时间用在了函数调用上。一个简单的想法是扩充虚拟机的指令级，使得其支持更复杂的运算，以减小指令分发和出入栈的成本。具体而言，`Cqq`解释器和Q虚拟机做了如下的指令合并：

1. 基础指令集中只含有`save`指令，即将次栈顶元素存放到栈顶元素所指的位置中，并将栈顶元素出栈。而大量的`save`是用于存放到立即数提供位置的内存位置中的。可以新增存放到对应位置的一地址指令以合并指令。`load`指令同理。

2. 一个运算指令有时用于计算一个立即数和栈顶元素的值，这带来了一定的成本。考虑新增将栈顶元素和立即数操作并修改栈顶的“原地指令”来合并指令。

3. 使用save命令后往往伴有pop指令，可以新建savepop指令将其合并。

4. 数组寻址中往往使用形如：

$$c_0 + c_1x_1 + c_2x_2 + \ldots + c_{k-1}x_{k-1}$$

的表达式，其中c_i是常数。考虑增加一个特殊的一地址指令leap，表示将栈顶元素乘一个常数，加到次栈顶，以减小数组寻址需要的指令个数。

5. 可以根据前面对循环变量的寻找做一个简单的优化，对于探查出的循环变量的赋值，用一个特殊的指令affine（表示一个映射$x \mapsto kx + b$）处理。

7.5 表达式代码生成

基于寄存器生成表达式树的最佳代码可以使用Ershov数和基于动态规划的算法生成。基于栈的虚拟机也可以使用类似的方法，下面的例子指出，交换一个可交换的二元运算的求值顺序，可能带来很大的性能提升。

假设都先计算左子树。考虑表达式2+(3+(4+5))的抽象文法树，先计算左孩子或右孩子将改变计算的开销。

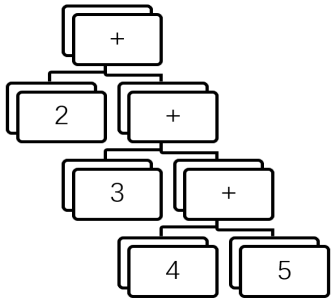


图 5: 表达式变换

Listing 3: 始终先计算右边

```
1 PUSHIMM 5
2 ADDON 4
3 ADDON 3
4 ADDON 2
```

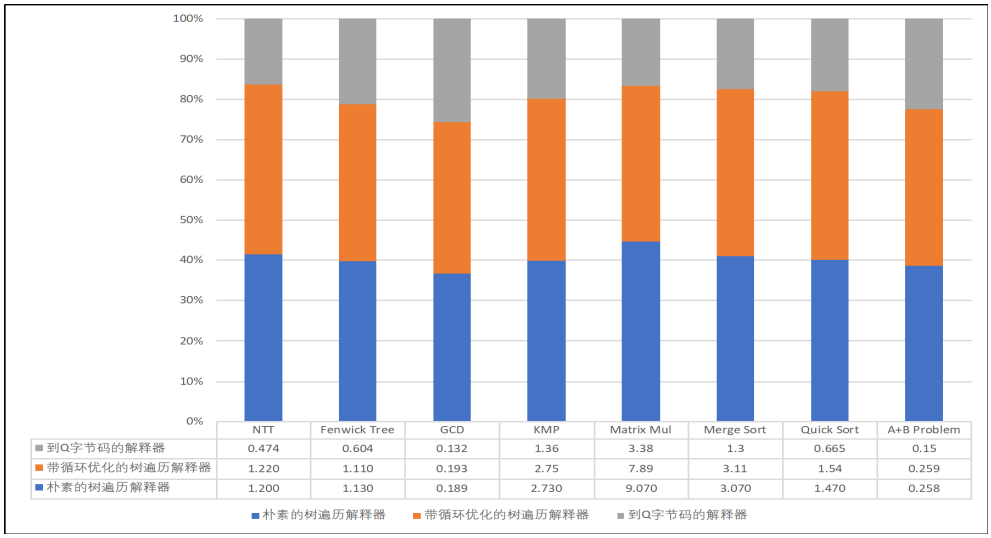
Listing 4: 始终先计算左边

```
1 PUSHIMM 2
2 PUSHIMM 3
3 PUSHIMM 4
4 ADDON 5
5 ADD
6 ADD
```

很明显，第一个方式充分利用了栈顶缓存，开销更小。

从直观上讲，先计算较“重”的子树会使得期望的运算次数更少。这里我们简化模型，让计算所用栈顶指针的平均高度最小，以达到更好的空间局部性并尽可能使用栈顶缓存。计算方法和Ershov数是相同的，在Cqq解释器中，将在AST层面优化中完成对其重量的计算。这样在翻译代码时，只需要优先计算更“重”的子树，即可最小化使用栈空间的平均值。

8 性能测试



	NTT	Fenwick Tree	GCD	KMP	Matrix Mul	Merge Sort	Quick Sort	A+B Problem
到Q字节码的解释器	0.474	0.604	0.132	1.36	3.38	1.3	0.665	0.15
带循环优化的树遍历解释器	1.220	1.110	0.193	2.75	7.89	3.11	1.54	0.259
朴素的树遍历解释器	1.200	1.130	0.189	2.730	9.070	3.070	1.470	0.258

图 6: 性能测试

测试机器是Microsoft Surface Laptop笔记本电脑，配置是Intel(R) Core(TM) i5-7300U，基准频率2.71GHz，最高睿频3.5GHz。测试系统是Windows

Ubuntu Subsystem（Ubuntu 16.04）。

分别测试没有优化的树遍历解释器、加入优化的树遍历解释器和作为最好结果的Q字节码解释器。选用的代码包括快速数论变换、树状数组、欧几里得算法、KMP算法、矩阵乘法和排序算法，最后的测试数据A+B Problem是综合测试，包含了排序算法、数据结构、网络流算法等多种算法的实现。

可以发现，对于树遍历解释器，循环优化只有矩阵乘法获得了比较理想的优化。这是因为`Cqq`实现的简单的基于AST层面的循环不变表达式分析受限于代码实现，如果代码比较复杂，或者循环不变表达式的计算并不是瓶颈，则无法获得理想的优化效果。这是AST层面分析粒度较粗的结果。

而基于Q字节码解释器无论从生成代码质量还是基于体系结构的优化都远远好于树遍历解释器，性能翻了一倍。其中快速排序的执行速度已经达到了G++编译生成代码性能的四分之一。