SOFTART DEVELOPMENT GROUP

实用编译器构建指南

以 SoftArt Shader Language 为例

空明流转（Ye Wu）

SoftArt Development Group

2009/12/10

本文利用现有的编译器前端或后端的技术和库，以可控制和渐增的方式，从无到有，从小到大，从简单到复杂，从低效到高效的实现一个编译器。在本文里，编译器的构建过程被分解成多个迭代的阶段。其中的大部分阶段，你都能够在理解它之后，在一个小时到一天不等的时间内达到预计的目标。

实用编译器构建指南

Ye Wu，SoftArt Development Group

wuye9036@gmail.com

1. 前言

在你看到这篇文章的时候，你一定在想，编译器啊，真是个麻烦的东西。呃，没有错，在撰写此文的时候，我也一直有着这样的念头。本文不能将你从编译器的苦海里救出来，只是希望它能够让你少游一会儿，以尽可能的少喝水。

与一般的文章不同，本文不会涉及到有关于编译原理的理论细节，NFA，DFA，递归下降，文法，语法制导翻译等等概念，都不会深入的追究。但是希望你能有这方面的概念，这样在讨论到一些算法及其实现的时候，不至于理解的太过费力。

本文重点在于利用现有各式各样的编译器前端或后端技术和库，以可控制和渐增的方式，将我们的编译器从无到有，从小到大，从简单到复杂，从低效到高效的实现出来。本文的写作目标是，我们将编写编译器的任务，分解成多个迭代的阶段，其中的大部分阶段，你都能够在理解它之后，在一个小时到一天不等的时间内达到预计的目标。这样我们才有动力进行下去，不是吗？

唔，自然，这样的文章需要一个完整的实例贯穿前后。我们决定采用 SoftArt 项目内最复杂的一个子项目 SoftArt Shader Language 作为一个导引，你所看到的行文过程，就是 SoftArt Shader Language 的开发过程。SoftArt 开发到哪里，这篇博文就写到哪里，我们对 SVN 发誓，我们所经历的，你都能看得到。

SoftArt 的项目主页 http://code.google.com/p/softart

SoftArt 项目介绍
http://www.cppblog.com/lingjinggiu/archive/2009/12/07/102698.html

换句话说，我们所走的弯路，你一样得走（哈哈，被我们坑的），我们所路过的捷径，一样也会成为你的捷径。祝大家好运。

2. 本文的使用说明

在阅读本文的时候，我们希望您能有以下的基础：

- 熟悉至少一门高级语言
- 能阅读带模板的 C++（我们的示例代码用此所写）
- 会使用正则表达式
- 了解计算机的工作原理，能阅读汇编
- 对 LL 和 LR 文法有一定的了解
- 掌握基本的数据结构和算法

如果您具备以下条件，那么恭喜您，本文就是为您而写的：

- 需要快速开发编译器
- 对编译原理了解不深
- 对编译速度和编译后的程序在性能上要求不甚苛刻

如果您具有以下特征之一，那么本文可能会给您带来不适，请谅解我们的工作：

- 天然喷
- 紫禁之巅的高手
- 车轮党
- 反模板联盟

3. 一个迷你的虚拟机

哈，我想你一定会问，不是说编译器么，为什么要先说虚拟机呢？唔……用一句话来回答就是，因为它必要而简单。还记得编译原理中描述的一个语言是如何被分析、解释并执行的么？

在词法分析->低级代码生成四个阶段里，每个阶段的输出都作为下一个阶段

的输入。这些输出通常都是规范化的，而且都以计算机容易识别的形式保存。依据这些阶段的输出来判断程序运作的正常与否，是需要很多经验的。这显然不是我们所希望的。

那么最后一个阶段，低级代码的执行，便成为了最容易实现，也最容易检验的阶段。在这一阶段，我们只需要设计一个虚拟机便可以达到目的。

唔，虚拟机。是个很可怕的名词，不是么？没关系，对我们的实现来说，虚拟机最大的好处，就在于它可大可小。我们可以实现一个 x86 一样指令集的庞然大物，也可以实现一个只有几条到几十条我们所需要的指令构成的小小玩具。

不过，既然叫虚拟机，那麻雀虽小，五脏俱全。咱们这只小麻雀，看中了谁的五脏呢？一个大家都学过的典范，8086。还是觉得复杂了，是吗？没关系，我们先来看看，8086 的哪些部分是最必要的，我们只要实现了这一部分，就能让虚拟机跑起来。

嗯……我们来看看，8086 里面，要做一个 1+1=2 的运算，需要哪些最基础的设施呢？**通用寄存器，栈，指令寄存器，指令槽**，当然还有最重要的**执行引擎**。好吧，其余的什么段啊，分页啊，中断啊，我们一概不需要。

好，那么我们便用最直截了当的方式，把这些我们需要的设施拼凑成我们的虚拟机。

```
class vm{
    vector<instruction> code_buffer;
    vector<byte> stack;
    int r[16];
    intptr_t eip;
    bool execute(const instruction& ins);
};
```

唔，有了这些基本的设备之后，我们得让它运转起来。我们先设计一个最简单的驱动代码，这个代码依次读入 code_buffer 里 eip 位置上的指令，然后执行这条指令，执行完后，移向下一条，知道指令槽空为止。

```
class vm{
    bool call(){
        eip = 0;
        while ( eip < code_buffer.size() ){
            if( execute(code_buffer[eip]) ){
                ++eip;
            } else {
                break;
            }
        }
    }
};
```

知道这个虚拟机怎么去执行指令后，还得要定义有哪些指令可以执行，每条指令执行后会产生什么效果。这些能够执行的指令的集合，就称为虚拟机的**指令集**。而后者，需要在我们的执行引擎里，编写实际的执行代码来达到每个指令的效果。

什么？你说要实现 x86 的指令集？天哪，那些指令仅仅是读就够你读上半天的。一开始，我们只需要实现最基本的几条指令，告诉我们，这个虚拟机可以 works，这个就 OK 了。我想大家的数学都是从 1+1=2 开始的，这里，我们也用加法运算做为开始。由于咱们这个虚拟机是仿照 x86 的，所以所有的运算都要在寄存器里完成。所以我们只需要两条指令便可以完成运算：读入数据到寄存器的指令和在寄存器里执行加法的指令，咱们的数据类型，只有万年的 32 位有符号整数，简称 int。

明确了我们需要什么指令，就要设计指令的格式了。本着抄袭……啊，不，是借鉴 x86 的精神，我们也将指令设计成三元组的形式：指令码，操作数 1，操作数 2

够用吗？这个你去问 x86，它会告诉你，历史证明，三元组基本够用。当然，其实我觉得四元组更好使……也许我会把它改成 4 元组的。唔，咱们不 YY，先来看三元组的数据结构，简单明了。

```
struct instruction{
    op_code op;
    int arg0;
    int arg1;
};
```

真没啥想头的，不是嘛？嗯，有了指令的数据结构，咱们就可以设计指令了。

在 instruction 里，op_code 是一个指令码的枚举

```
enum op_code{
    op_add,
    op_loadrc
};
```

op_add 执行的是寄存器之间的加法，arg0 是目标寄存器号（实际上就是寄存器数组 r 的下标），arg1 是源寄存器号；该指令将 arg1 中的值加到 arg0 中。

op_loadrc 将一个数值读入到寄存器中。Arg0 是寄存器号，arg1 是数值。

然后在 execute 里面实现这两条指令

```
bool execute (op_code op, int arg0, int arg1){
    switch (op) {
    case op_add:
        r[arg0] += r[arg1]; break;
    case op_loadrc: {
            int& reg( r[arg0] );
            int val = arg1;
            reg = val;
            break;
        }
    }
    return true;
}
```

最后，再添上一些辅助性的代码，这个虚拟机就能够正常的工作了。不信？咱们可以试试：

```
int _tmain(int argc, _TCHAR* argv[])
{
    code_generator cg;
    cg
        //读取两个常量
        .op( op_loadrc, r0, 10 )
        .op( op_loadrc, r1, 67 )
        .op( op_add, r0, r1) //执行加法
        ;

    vm machine;
    int result = machine.raw_call( cg.codes() );
    std::cout << result << endl;
    system("pause");
    return 0;
}
```

呵呵，Run 起来吧~