使用 LLVM 框架创建一个工作编译器，第 1 部分

使用 LLVM 及其中间表示构建一个自定义编译器

Arpan Sen

2012 年 7 月 17 日发布

LLVM（之前称为低级虚拟机）是一种非常强大的编译器基础架构框架，专门为使用您喜爱的编程语言编写的程序的编译时、链接时和运行时优化而设计。LLVM 可运行于若干个不同的平台之上，它以能够生成快速运行的代码而著称。

LLVM 框架是围绕着代码编写良好的中间表示 (IR) 而构建的。本文（由两部分组成的系列文章的第一部分）将深入讲解 LLVM IR 的基础知识以及它的一些微妙之处。在这里，您将构建一个可以自动为您生成 LLVM IR 的代码生成器。拥有一个 LLVM IR 生成器意味着您所需要的是一个前端以供插入您所喜爱的编程语言，而且这还意味着您拥有一个完整的流程（前端解析器 + IR 生成器 + LLVM 后端）。创建一个自定义编译器会变得更加简单。

开始使用 LLVM

llc 和 lli

概览

因为 LLVM 是一个虚拟机，所以它可能应该拥有自己的中间字节代码表示，不是吗？最后，您需要将 LLVM 字节代码编译到特定于平台的汇编语言中。然后您才能通过本机汇编程序和链接器来运行汇编代码，从而生成可执行的共享库等。您可以使用 `llc` 将 LLVM 字节代码转换成特定于平台的汇编代码（请参阅 参考资料，获取关于此工具的更多信息的链接）。对于 LLVM 字节代码的直接执行部分，不要等到在本机执行代码崩溃后才发现您的程序中有一个或两个 bug。这正是 `lli` 的用武之地，因为它可以直接执行字节代码。`lli` 可以通过解释器或使用高级选项中的即时 (JIT) 编译器执行此工作。请参阅 参考资料，获取关于 `lli` 的更多信息的链接。

llvm-gcc

llvm-gcc 是 GNU Compiler Collection (gcc) 的修改版本，可以在使用 `-S -emit-llvm` 选项运行时会生成 LLVM 字节代码。然后您可以使用 `lli` 来执行这个已生成的字节代码（也称为 *LLVM 汇编语言*）。有关 llvm-gcc 的更多信息，请参阅 参考资料。如果您没有在自己的系统中预先安装 llvm-gcc，那么您应该能够从源代码构建它，请参阅 参考资料，获取分步指南的链接。

使用 LLVM 编写 Hello World

要更好地理解 LLVM，您必须了解 LLVM IR 及其微妙之处。这个过程类似于学习另一种编程语言。但是，如果您熟悉 `C` 语言和 `C++` 语言以及它们的一些语法怪现象，那么在了解 LLVM IR 方面您应该没有太大的障碍。清单 1 给出了您的第一个程序，该程序将在控制台输出中打印 "Hello World"。要编译此代码，您可以使用 llvm-gcc。

```
#include <stdio.h>
int main( )
{
  printf("Hello World!\n");
}
```

要编译此代码，请输入此命令：

```
Tintin.local# llvm-gcc helloworld.cpp -S -emit-llvm
```

完成编译后，llvm-gcc 会生成 helloworld.s 文件，您可以使用 `lli` 来执行该文件，将消息输出到控制台。`lli` 的用法如下：

```
Tintin.local# lli helloworld.s
Hello, World
```

现在，先看一下 LLVM 汇编语言。清单 2 给出了该代码。

清单 2. Hello World 程序的 LLVM 字节代码

```
@.str = private constant [13 x i8] c"Hello World!\00", align 1 ;

define i32 @main() ssp {
entry:
  %retval = alloca i32
  %0 = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @.str, i64 0, i64 0))
  store i32 0, i32* %0, align 4
  %2 = load i32* %0, align 4
  store i32 %2, i32* %retval, align 4
  br label %return
return:
```

```
17
18  declare i32 @puts(i8*)
```


理解 LLVM IR

LLVM 提供了一个详细的汇编语言表示（参阅 参考资料 获取相关的链接）。在开始编写我们之前讨论的自己的 Hello World 程序版本之前，有几个需知事项：

- LLVM 汇编语言中的注解以分号 (;) 开始，并持续到行末。
- 全局标识符要以 @ 字符开始。所有的函数名和全局变量都必须以 @ 开始。
- LLVM 中的局部标识符以百分号 (%) 开始。标识符典型的正则表达式是 `[%@][a-zA-Z$._][a-zA-Z$._0-9]*`。
- LLVM 拥有一个强大的类型系统，这也是它的一大特性。LLVM 将整数类型定义为 `iN`，其中 *N* 是整数占用的字节数。您可以指定 1 到 223- 1 之间的任意位宽度。
- 您可以将矢量或阵列类型声明为 `[no. of elements X size of each element]`。对于字符串 "Hello World!"，可以使用类型 `[13 x i8]`，假设每个字符占用 1 个字节，再加上为 NULL 字符提供的 1 个额外字节。
- 您可以对 hello-world 字符串的全局字符串常量进行如下声明：`@hello = constant [13 x i8] c"Hello World!\00"`。使用关键字 `constant` 来声明后面紧跟类型和值的常量。我们已经讨论过类型，所以现在让我们来看一下值：您以 `c` 开始，后面紧跟放在双引号中的整个字符串（其中包括 `\0` 并以 `0` 结尾）。不幸的是，关于字符串的声明为什么需要使用 `c` 前缀，并在结尾处包含 NULL 字符和 0，LLVM 文档未提供任何解释。如果您有兴趣研究更多有关 LLVM 的语法怪现象，请参阅 参考资料，获取语法文件的链接。
- LLVM 允许您声明和定义函数。而不是仔细查看 LLVM 函数的整个特性列表，我只需将精力集中在基本要点上即可。以关键字 `define` 开始，后面紧跟返回类型，然后是函数名。返回 32 字节整数的 `main` 的简单定义类似于：`define i32 @main() { ;`

LLVM 等同物。该声明以关键字 declare 开始，后面紧跟着返回类型、函数名，以及该函数的可选参数列表。该声明必须是全内容局范围的。

- 每个函数均以返回语句结尾。有两种形式的返回语句：ret <type> <value> 或 ret void。对于您简单的主例程，使用 ret i32 0 就足够了。
- 使用 call <function return type> <function name> <optional function arguments> 来调用函数。注意，每个函数参数都必须放在其类型的前面。返回一个 6 位的整数并接受一个 36 位的整数的函数测试的语法如下：call i6 @test(i36 %arg1)。

这只是一个开始，您还需要定义一个主例程、一个存储字符串的常量，以及处理实际打印的 puts 方法的声明。清单 3 显示第一次尝试创建的程序。

清单 3. 第一次尝试创建手动编写的 Hello World 程序

```
1  declare  i32 @puts(i8*)
2  @global_str = constant [13 x i8] c"Hello World!\00"
3  define i32 @main {
4    call i32 @puts( [13 x i8] @global_str )
5    ret i32 0
6  }
```

这里提供了来自 lli 的日志：

```
1  lli: test.s:5:29: error: global variable reference must have pointer type
2    call i32 @puts( [13 x i8] @global_str )
3                               ^
```

将您引向了 LLVM 指令 getelementptr。请注意，您必须将 清单 3 中的 puts 调用修改为与 `call i32 @puts(i8* %t)` 类似，其中 %t 是类型 i8*，并且是 `[13 x i8] to i8*` 的类型转换结果。（请参阅 参考资料，获取 getelementptr 的详细描述的链接。）在进一步探讨之前，清单 4 提供了可行的代码。

内容
概览
开始使用 LLVM
相关主题
评论

清单 4. 使用 getelementptr 正确地将类型转换为指针

```
1  declare i32 @puts (i8*)
2  @global_str = constant [13 x i8] c"Hello World!\00"
3
4  define i32 @main() {
5    %temp = getelementptr [13 x i8]*  @global_str, i64 0, i64 0
6    call i32 @puts(i8* %temp)
7    ret i32 0
8  }
```

getelementptr 的第一个参数是全局字符串变量的指针。要单步执行全局变量的指针，则需要使用第一个索引，即 `i64 0`。因为 getelementptr 指令的第一个参数必须始终是 pointer 类型的值，所以第一个索引会单步调试该指针。0 值表示从该指针起偏移 0 元素偏移量。我的开发计算机运行的是 64 位 Linux®，所以该指针是 8 字节。第二个索引 (`i64 0`) 用于选择字符串的第 0 个元素，该元素是作为 puts 的参数来提供的。

创建一个自定义的 LLVM IR 代码生成器

了解 LLVM IR 是件好事，但是您需要一个自动化的代码生成系统，用它来转储 LLVM 汇编语言。谢天谢地，LLVM 提供了强大的应用程序编程接口 (API) 支持，让您可以查看整个过程（请参阅 参考资料，获取程序员手册的链接）。在您的开发计算机上查找 LLVMContext.h 文件；如果该文件缺失，那么可能是您安装 LLVM 的方式出错。

LLVM 提供了一款出色的工具，叫做 `llvm-config`（参阅 参考资料）。运行 `llvm-config –cxxflags`，获取需要传递至 g++ 的编译标志、链接器选项的 `llvm-config –ldflags` 以及 `llvm-config –ldflags`，以便针对正确的 LLVM 库进行链接。在 清单 5 的样例中，所有的选项均需要传递至 g++。

清单 5. 通过 LLVM API 使用 llvm-config 构建代码

```
tintin# llvm-config --cxxflags --ldflags --libs \
-I/usr/include  -DNDEBUG -D_GNU_SOURCE \
-D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS \
-D__STDC_LIMIT_MACROS -O3  -fno-exceptions -fno-rtti -fno-common \
-Woverloaded-virtual -Wcast-qual \
-L/usr/lib  -lpthread -lm \
-lLLVMXCoreCodeGen -lLLVMTableGen -lLLVMSystemZCodeGen \
-lLLVMSparcCodeGen -lLLVMPTXCodeGen \
-lLLVMPowerPCCodeGen -lLLVMMSP430CodeGen -lLLVMMipsCodeGen \
-lLLVMMCJIT -lLLVMRuntimeDyld \
-lLLVMObject -lLLVMMCDisassembler -lLLVMXCoreDesc -lLLVMXCoreInfo \
-lLLVMSystemZDesc -lLLVMSystemZInfo \
-lLLVMSparcDesc -lLLVMSparcInfo -lLLVMPowerPCDesc -lLLVMPowerPCInfo \
-lLLVMPowerPCAsmPrinter \
-lLLVMPTXDesc -lLLVMPTXInfo -lLLVMPTXAsmPrinter -lLLVMMipsDesc \
-lLLVMMipsInfo -lLLVMMipsAsmPrinter \
-lLLVMMSP430Desc -lLLVMMSP430Info -lLLVMMSP430AsmPrinter \
-lLLVMMBlazeDisassembler -lLLVMMBlazeAsmParser \
-lLLVMMBlazeCodeGen -lLLVMMBlazeDesc -lLLVMMBlazeAsmPrinter \
-lLLVMMBlazeInfo -lLLVMLinker -lLLVMipo \
-lLLVMInterpreter -lLLVMInstrumentation -lLLVMJIT -lLLVMExecutionEngine \
-lLLVMDebugInfo -lLLVMCppBackend \
-lLLVMCppBackendInfo -lLLVMCellSPUCodeGen -lLLVMCellSPUDesc \
-lLLVMCellSPUInfo -lLLVMCBackend \
-lLLVMCBackendInfo -lLLVMBlackfinCodeGen -lLLVMBlackfinDesc \
-lLLVMBlackfinInfo -lLLVMBitWriter \
```

```
30  -lLLVMARMCodeGen -lLLVMARMDesc \
31  -lLLVMARMAsmPrinter -lLLVMARMInfo -lLLVMArchive -lLLVMBitReader \
32  -lLLVMAlphaCodeGen -lLLVMSelectionDAG \
33  -lLLVMAsmPrinter -lLLVMMCParser -lLLVMCodeGen -lLLVMScalarOpts \
34  -lLLVMInstCombine -lLLVMTransformUtils \
35  -lLLVMipa -lLLVMAnalysis -lLLVMTarget -lLLVMCore -lLLVMAlphaDesc \
36  -lLLVMAlphaInfo -lLLVMMC -lLLVMSupport
```

使用 LLVM 编写 Hello World

LLVM 模块和上下文环境等

理解 LLVM IR

创建一个自定义的 LLVM 代码生成器

LLVM 模块类是其他所有 LLVM IR 对象的顶级容器。LLVM 模块类能够包含全局变量、函数、该模块所依赖的其他模块和符号表等对象的列表。这里将提供了 LLVM 模块的构造函数：

结束语

```
1  explicit Module(StringRef ModuleID, LLVMContext& C);
```

评论

要构建您的程序，必须从创建 LLVM 模块开始。第一个参数是该模块的名称，可以是任何虚拟的字符串。第二个参数称为 LLVMContext。LLVMContext 类有些晦涩，但用户足以了解它提供了一个用来创建变量等对象的上下文环境。该类在多线程的上下文环境中变得非常重要，您可能想为每个线程创建一个本地上下文环境，并且想让每个线程完全独立于其他上下文环境运行。目前，使用这个默认的全局上下文来处理 LLVM 所提供的代码。这里给出了创建模块的代码：

```
1  llvm::LLVMContext& context = llvm::getGlobalContext();
2
3  llvm::Module* module = new llvm::Module("top", context);
```

您要了解的下一个重要类是能实际提供 API 来创建 LLVM 指令并将这些指令插入基础块的类：IRBuilder 类。IRBuilder 提供了许多华而不实的方法，但是我选择了最简单的可行方法来构建一个 LLVM 指令，即使用以下代码来传递全局上下文：

```
3 llvm::Module* module = new llvm::Module("top", context);
4
5 llvm::IRBuilder<> builder(context);
```

概览

准备好 LLVM 对象模型后，就可以调用模块的 dump 方法来转储其内容。清单 6 给出了该代码。

开始使用 LLVM

清单 6. 创建一个转储模块

使用 LLVM 编写 Hello World

```
1  #include "llvm/LLVMContext.h"
2  #include "llvm/Module.h"
3  #include "llvm/Support/IRBuilder.h"
4
5  int main()
6  {
7    llvm::LLVMContext& context = llvm::getGlobalContext();
8    llvm::Module* module = new llvm::Module("top", context);
9    llvm::IRBuilder<> builder(context);
10
11   module->dump( );
12 }
```

运行 清单 6 中的代码之后，控制台的输出如下：

```
1 ; ModuleID = 'top'
```

然后，您需要创建 main 方法。LLVM 提供了 llvm::Function 类来创建一个函数，并提供了 llvm::FunctionType 将该函数与某个返回类型相关联。此外，请记住，main 方法必须是该模块的一部分。清单 7 给出了该代码。

清单 7. 将 main 方法添加至顶部模块

```
4
5  int main()
6  {
7    llvm::LLVMContext& context = llvm::getGlobalContext();
8    llvm::Module *module = new llvm::Module("top", context);
9    llvm::IRBuilder<> builder(context);
10
11   llvm::FunctionType *funcType =
12       llvm::FunctionType::get(builder.getInt32Ty(), false);
13   llvm::Function *mainFunc =
14       llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);
15
16   module->dump( );
17 }
```

创建一个自定义的 LLVM IR 代码生成器

请注意，您需要让 main 返回 void，这就是您调用 builder.getVoidTy() 的原因；如果 main 返回 i32，那么该调用会是 builder.getInt32Ty()。在编译并运行 清单 7 中的代码后，出现的结果如下：

结束语
相关主题

```
1  ; ModuleID = 'top'
2  declare void @main()
```

您还尚未定义 main 要执行的指令集。为此，您必须定义一个基础块并将其与 main 方法关联。*基础块* 是 LLVM IR 中的一个指令集合，拥有将标签（类似于 C 标签）定义为其构造函数的一部分的选项。builder.setInsertPoint 会告知 LLVM 引擎接下来将指令插入何处。清单 8 给出了该代码。

清单 8. 向 main 添加一个基础块

```
1  #include "llvm/LLVMContext.h"
2  #include "llvm/Module.h"
3  #include "llvm/Support/IRBuilder.h"
4
5  int main()
```

```
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType =
      llvm::FunctionType::get(builder.getInt32Ty(), false);
  llvm::Function *mainFunc =
      llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);

  llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
  builder.SetInsertPoint(entry);

  module->dump( );
}
```

这里提供了清单 8 的输出。请注意，由于现在已经定义了 main 的基础块，所以 LLVM 转储将 main 看作为是一个方法定义，而不是一个声明。非常酷！

结束语

```
; ModuleID = 'top'
define void @main() {
entrypoint:
}
```

现在，向代码添加全局 hello-world 字符串。清单 9 给出了该代码。

清单 9. 向 LLVM 模块添加全局字符串

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Support/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);
```

```
13    llvm::Function *mainFunc =
14        llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);
15
16    llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
17    builder.SetInsertPoint(entry);
18
19    llvm::Value *helloWorld = builder.CreateGlobalStringPtr("hello world!\n");
20
21    module->dump( );
22  }
```

理解 LLVM IR

在 清单 9 的输出中，注意 LLVM 引擎是如何转储字符串的：

创建一个自定义的 LLVM IR 代码生成器

```
1  ; ModuleID = 'top'
2  @0 = internal unnamed_addr constant [14 x i8] c"hello world!\0A\00"
3  define void @main() {
4  entrypoint:
5  }
```

评论

现在您需要做的就是声明 puts 方法，并且调用它。要声明 puts 方法，则必须创建合适的 FunctionType*。从您的 Hello World 源始代码中，您知道 puts 返回了 i32 并接受 i8* 作为输入参数。清单 10 给出了创建 puts 的正确类型的代码。

清单 10. 声明 puts 方法的代码

```
1  std::vector<llvm::Type *> putsArgs;
2  putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
3  llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);
4
5  llvm::FunctionType *putsType =
6    llvm::FunctionType::get(builder.getInt32Ty(), argsRef, false);
7  llvm::Constant *putsFunc = module->getOrInsertFunction("puts", putsType);
```

改变，输出显示将如 清单 11 所示。

内容
清单 11. 声明 puts 方法

概览
```
; ModuleID = 'top'
@0 = internal unnamed_addr constant [14 x i8] c"hello world!\0A\00"
define void @main() {
entrypoint:
}
declare i32 @puts(i8*)
```

剩下要做的是调用 LLVM 中的 puts 方法，并从 main 中返回。LLVM API 非常关注转换等操作：您需要做的是调用 puts 来调用 builder.CreateCall。最后，要创建返回语句，请调用 builder.CreateRetVoid。清单 12 提供了完整的运行代码。
结束语

清单 12. 输出 Hello World 的完整代码
相关主题

```
#include "llvm/ADT/ArrayRef.h"
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/Support/IRBuilder.h"
#include <vector>
#include <string>

int main()
{
  llvm::LLVMContext & context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("asdf", context);
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType = llvm::FunctionType::get(builder.getVoidTy(), false);
  llvm::Function *mainFunc =
    llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);
  llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
```

```
23
24    std::vector<llvm::Type *> putsArgs;
25    putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
26    llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);
27
28    llvm::FunctionType *putsType =
29      llvm::FunctionType::get(builder.getInt32Ty(), argsRef, false);
30    llvm::Constant *putsFunc = module->getOrInsertFunction("puts", putsType);
31
32    builder.CreateCall(putsFunc, helloWorld);
33    builder.CreateRetVoid();
34    module->dump();
35  }
```

创建一个自定义的 LLVM IR 代码生成器

结束语

结束语
相关主题
评论

在这篇初步了解 LLVM 的文章中，了解了诸如 `lli` 和 `llvm-config` 等 LLVM 工具，还深入研究了 LLVM 中间代码，并使用 LLVM API 来为您自己生成中间代码。本系列的第二部分（也是最后一部分）将探讨可以使用 LLVM 完成的另一项任务，即毫不费力地添加额外的编译传递。

相关主题

- 获取官方的 LLVM 教程，了解有关 LLVM 的精彩简介。
- 参见 *The Architecture of Open Source Applications* 中的 Chris Latner 的章节，获取有关 LLVM 开发的更多信息。
- 了解有关以下两种 LLVM 重要工具的更多信息：`llc` 和 `lli`。
- 查找有关 llvm-gcc 工具的更多信息，并了解如何通过分步指南从源代码构建此工具 Building llvm-gcc from Source。

- 查看 LLVM 的 语法文件（/llvm/trunk/utils/llvm.grm 的日志），获取有关 LLVM 中全局字符串常量的更多信息。
- 在 "The Often Misunderstood GEP Instruction" 文件中了解有关 `getelementptr` 指令 的更多信息。
- 研究 LLVM Programmer's Manual，这是有关 LLVM API 必不可少的资源。
- 阅读有关 `llvm-config` 工具 的资料，了解 LLVM 编译选项。
- 在 developerWorks 中国网站 Linux 技术专区中，查找数百篇 指南文章和教程，还有下载、论坛，以及针对 Linux 开发人员和管理员的丰富资源。
- developerWorks 中国网站 Web 开发专区 专门研究涵盖各种基于 Web 解决方案的文章。
- 以最适合您的方式 IBM 产品评估试用版软件：下载产品试用版，在线试用产品，在云环境下试用产品，或者在 IBM SOA 人员沙箱中花费几个小时来学习如何高效实现面向服务的架构。
- 随时关注 developerWorks 技术活动和网络广播。
- 访问 developerWorks Open source 专区获得丰富的 how-to 信息、工具和项目更新以及最受欢迎的文章和教程，帮助您用开放源码技术进行开发，并将它们与 IBM 产品结合使用。

评论

添加或订阅评论，请先登录或注册。

有新评论时提醒我