Kaleidoscope 系列第八章：编译为目标代码

作者：Hanseltu
原文链接：https://ld246.com/article/1570096476155
来源网站：链滴

原文链接：Kaleidoscope 系列第八章：编译为目标代码

本文是使用 LLVM 开发新语言 Kaleidoscope 教程系列第八章，将生成的代码编译为目标机器代码。

第八章简介

欢迎来到“使用 LLVM 开发新语言 Kaleidoscope 教程”教程的第八章。本章介绍如何将我们的中间码编译为目标代码。

选择一个目标

LLVM 支持本地交叉编译。我们可以将代码编译为当前计算机的体系结构，也可以像针对其他体系结一样轻松地进行编译。在本教程中，我们主要针对当前计算机。

为了指定要定位的体系结构，我们使用一个称为 target Triple 的字符串。这采用表格形式 <arch><sb>-<vendor>-<sys>-<abi>（请参阅交叉编译文档）。

例如，通过 clang 我们可以看到我们当前的 target Triple：

```
$ clang --version | grep Target
Target: x86_64-unknown-linux-gnu
```

运行此命令可能会显示与您使用的体系结构或操作系统不同的计算机上的某些内容。

幸运的是，我们不需要对目标三元组进行硬编码就可以将当前机器作为目标。LLVM 提供了 sys::getDefaultTargetTriple，它返回当前计算机的目标三元组。

```
auto TargetTriple = sys::getDefaultTargetTriple();
```

LLVM 不需要我们链接所有目标功能。例如，如果我们仅使用 JIT，则不需要组装打印机。同样，如我们仅针对某些架构，则只能链接这些架构的功能。

在此示例中，我们将初始化所有目标以发出目标代码。

```
InitializeAllTargetInfos();
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();
```

现在，我们就可以使用目标三元组获得 Target：

```
std::string Error;
auto Target = TargetRegistry::lookupTarget(TargetTriple, Error);

// Print an error and exit if we couldn't find the requested target.
// This generally occurs if we've forgotten to initialise the
// TargetRegistry or we have a bogus target triple.
if (!Target) {
  errs() << Error;
  return 1;
}
```

目标机器

我们还将需要一个 TargetMachine 类。此类提供了我们要定位的机器的完整机器描述。如果我们要对特定功能（例如 SSE）或特定 CPU（例如英特尔的 Sandylake）。

要查看 LLVM 知道哪些功能和 CPU，我们可以使用 llc 命令。例如，让我们看一下 x86：

```
$ llvm-as < /dev/null | llc -march=x86 -mattr=help
Available CPUs for this target:

  amdfam10      - Select the amdfam10 processor.
  athlon        - Select the athlon processor.
  athlon-4      - Select the athlon-4 processor.
  ...

Available features for this target:

  16bit-mode            - 16-bit mode (i8086).
  32bit-mode            - 32-bit mode (80386).
  3dnow                 - Enable 3DNow! instructions.
  3dnowa                - Enable 3DNow! Athlon instructions.
  ...
```

对于我们的示例，我们将使用不带任何其他功能，选项或重定位模型的通用 CPU。

```
auto CPU = "generic";
auto Features = "";

TargetOptions opt;
auto RM = Optional<Reloc::Model>();
auto TargetMachine = Target->createTargetMachine(TargetTriple, CPU, Features, opt, RM);
```

配置 Module

现在，我们可以配置模块，以指定目标和数据布局。这不是严格必要的，但是前端性能指南建议这样。优化可以从了解目标和数据布局中受益。

```
TheModule->setDataLayout(TargetMachine->createDataLayout());
TheModule->setTargetTriple(TargetTriple);
```

生成目标代码

我们准备生成目标代码！我们先定义要将文件写入的位置：

```
auto Filename = "output.o";
std::error_code EC;
raw_fd_ostream dest(Filename, EC, sys::fs::OF_None);

if (EC) {
  errs() << "Could not open file: " << EC.message();
  return 1;
}
```

最后，我们定义一个发出目标代码的过程，然后运行该 pass：

```
legacy::PassManager pass;
auto FileType = TargetMachine::CGFT_ObjectFile;

if (TargetMachine->addPassesToEmitFile(pass, dest, nullptr, FileType)) {
  errs() << "TargetMachine can't emit a file of this type";
  return 1;
}

pass.run(*TheModule);
dest.flush();
```

组合起来

它行得通吗？试一试吧。我们需要编译代码，但是请注意，to 的参数 llvm-config 与前面的章节不同。

```
$ clang++ -g -O3 chapter8-Compiling-to-Object-Code.cpp `llvm-config --cxxflags --ldflags --
ystem-libs --libs all` -o toy
```

让我们运行它，并定义一个简单的 average 函数。完成后按 Ctrl-D。

```
$ ./toy
ready> def average(x y) (x + y) * 0.5;
^D
Wrote output.o
```

我们有一个目标文件！为了测试它，让我们编写一个简单的程序并将其与我们的输出链接。这是源代
：

```
#include <iostream>

extern "C" {
    double average(double, double);
}

int main() {
    std::cout << "average of 3.0 and 4.0: " << average(3.0, 4.0) << std::endl;
}
```

我们将程序链接到 output.o，并检查结果是否符合预期：

```
$ clang++ main.cpp output.o -o main
$ ./main
average of 3.0 and 4.0: 3.5
```

完整代码集合

chapter8-Compiling-to-Object-Code.cpp

参考： Kaleidoscope: Compiling to Object Code