**第八章阅读笔记**

**---编译为目标代码**

## [8.2。选择目标](http://llvm.org/docs/tutorial/LangImpl08.html" \l "id2)

LLVM本身支持交叉编译。您可以编译到当前计算机的体系结构，也可以轻松编译其他体系结构。

要指定要定位的体系结构，我们使用称为“目标三元组”的字符串。这采用的形式 <arch><sub>-<vendor>-<sys>-<abi>（参见[交叉编译文档](http://clang.llvm.org/docs/CrossCompilation.html" \l "target-triple)）。

举个例子，我们可以看到clang认为我们当前的目标三元组：

$ clang --version | grep Target

Target: x86\_64-unknown-linux-gnu

运行此命令可能会在您的计算机上显示不同的内容，因为您可能正在使用不同的体系结构或操作系统。

幸运的是，我们不需要硬编码目标三元组来定位当前机器。LLVM提供sys::getDefaultTargetTriple，返回当前计算机的目标三元组。

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

LLVM不要求我们链接所有目标功能。例如，如果我们只是使用JIT，我们不需要组装打印机。同样，如果我们仅针对某些体系结构，我们只能链接这些体系结构的功能。

对于此示例，我们将初始化发出目标代码的所有目标。

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

我们现在可以使用我们的目标三元组来获得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;}

## [8.2。目标机器](http://llvm.org/docs/tutorial/LangImpl08.html" \l "id3)

我们还需要一个TargetMachine。此类提供了我们所针对的机器的完整机器描述。如果我们想要定位特定功能（例如SSE）或特定CPU（例如Intel的Sandylake），我们现在就这样做。

要查看LLVM知道哪些功能和CPU，我们可以使用 llc。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，我们将使用通用CPU，而无需任何其他功能，选项或重定位模型。

**auto** CPU = "generic";**auto** Features = "";

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

## [8.4。配置模块](http://llvm.org/docs/tutorial/LangImpl08.html" \l "id4)

我们现在准备配置我们的模块，以指定目标和数据布局。这不是绝对必要的，但[前端性能指南](http://llvm.org/docs/Frontend/PerformanceTips.html)建议这样做。通过了解目标和数据布局，优化得益。

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

## [8.5。发射对象代码](http://llvm.org/docs/tutorial/LangImpl08.html" \l "id5)

我们准备发出目标代码了！让我们定义我们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::F\_None);

**if** (EC) {

errs() << "Could not open file: " << EC.message();

**return** 1;}

最后，我们定义了一个发出目标代码的传递，然后我们运行该传递：

legacy::PassManager pass;**auto** FileType = TargetMachine::CGFT\_ObjectFile;

**if** (TargetMachine->addPassesToEmitFile(pass, dest, FileType)) {

errs() << "TargetMachine can't emit a file of this type";

**return** 1;}

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

## [8.6。全部放在一起](http://llvm.org/docs/tutorial/LangImpl08.html" \l "id6)

它有用吗？试一试吧。我们需要编译我们的代码，但请注意，参数llvm-config与前面的章节不同。

$ clang++ -g -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-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。这是源代码：

#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我们的预期：

$ clang++ main.cpp output.o -o main

$ ./main

average of 3.0 and 4.0: 3.5

组员：

阿依多斯 2016302580021

周彪 2016302580062