pass的三种编译方式

参考的文献

1. 编写第一个llvm pass
2. 在 LLVM 代码树外编译 LLVM Pass (使用 OLLVM 示范)
3. LLVM从安装到手写一个入门Pass
4. LLVM Pass入门导引
5. LeadroyaL's website LLVM系列

LLVM pass 可以在源码目录编译，也可以在任意其他目录用命令行编译，官方推荐的是使用cmake编译，或者称之为树外编译。

为了醒目易识别，本文中出现的注释均使用 //， 在CMakeLists.txt中实际运行会报错，可以考虑删除或者替换成 #

0x00 安装LLVM

无论哪种方式，都需要电脑中安装LLVM。对于LLVM的安装，网上有很多的教程，可以结合 LLVM从安装到手写一个入门Pass 和 llvm学习（一）：简介与安装 进行相应的安装配置。同时LLVM 10.0.0 于2020年3月24号开始提供下载，不过鉴于当前网络上的相关教程基于 9.0 以及版本间的相同函数的接口和依赖函数可能发生变化，建议先安装LLVM 9.0 熟悉基本的操作后，再转到 10.0，本文相关的样例及搭建的环境是基于LLVM 9.0，未在LLVM10.0上进行测试。同时在编写pass时，建议使用Clion 等IDE工具，进行代码的自动提示，以快速了解相应函数的使用。

0x01 源码目录的编译

如果在安装LLVM 时，选择了源代码编译安装，除源码文件夹llvm，编译时应当是在同级目录新建了文件夹build，并在build内进行了程序的编译。因为我们当前关心的是pass的编写及运行，这里要留意两个路径：

1. 已安装的llvm的路径/lib/Transforms
2. 已安装的llvm的路径/include/llvm/Transforms

第一个路径下，放置了pass的源代码，包括cpp文件和cmake文件；第二个路径下放置了pass所需的头文件。

如果我们要添加自己的pass，可以在第一个文件路径下进行添加。这里，我们添加一个简单的OpcodeCounter，作用是统计程序中操作符使用的次数。

具体的流程

1. 在Transformation下新建文件夹OpcodeCounter，并添加相对应的文件OpcodeCounter.cpp 和CMakeLists.txt
2. 在Transformation 目录下的CMakeLists.txt中进行pass的注册
3. 编译pass
4. 编写测试用例，检测pass的执行效果

具体的样例

首先，我们进入到已安装的llvm的路径/lib/Transforms目录内，新建一个文件夹OpcodeCounter，并在内部新建一个OpcodeCounter.cpp 文件，内容如下：

```
#include "llvm/Pass.h"
```

```
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;
namespace{
    struct OpcodeCounter : public FunctionPass{
        static char ID;
        OpcodeCounter() : FunctionPass(ID){}

        bool runOnFunction(Function &F) override {
            std::map<std::string, int> opcodeCounter;
            errs() << "Function name: " << F.getName() << '\n';// 输出函数名
            for(Function::iterator fun = F.begin(); fun != F.end(); fun++){ //对
函数进行遍历
                for(BasicBlock::iterator bb = fun->begin(); bb != fun->end();
bb++){// 对基本块进行遍历
                    if(opcodeCounter.find(bb->getOpcodeName()) ==
opcodeCounter.end()){// 未出现过的操作符
                        opcodeCounter[bb->getOpcodeName()] = 1;
                    } else {
                        opcodeCounter[bb->getOpcodeName()] += 1;
                    }
                }
            }

            std::map<std::string, int>::iterator b = opcodeCounter.begin();
            std::map<std::string, int>::iterator e = opcodeCounter.end();
            while(b != e){
                llvm::outs() << b->first << " : " << b->second << '\n';
                b++;
            }
            llvm::outs() << '\n';
            opcodeCounter.clear(); // map资源释放
            return false;
        }

    };
}

char OpcodeCounter::ID = 0;
statidc RegisterPass<OpcodeCounter> X("oc", "Opcode Counter", false, false);
```

最后一行代码在pass加载时向pass管理器进行注册，其中参数的意义是：

- 第一个参数，表示pass名称，可以作为参数被opt命令识别并执行
- 第二个参数，表示pass的拓展名，对pass进行概要说明
- 第三个参数，表示当前pass是否改变了程序的控制流图（CFG），可选
- 第四个参数，表示当前pass是否实现了一个分析pass，可选

然后，编写文件CMakeLists.txt

```
add_llvm_library(LLVMOpcodeCounter MODULE
    OpcodeCounter.cpp

    PLUGIN_TOOL
    opt
)
```

接下来，返回到Transformation 目录下，修改该目录下的CMakeLists.txt

```
...
add_subdirectory(OpcodeCounter) // 在最后添加这一句
```

之后，在build目录下进行编译，这一步会编译项目所有的pass，有一定的时间开销

```
make
```

成功运行后，会在build/lib/内得到文件 LLVMOpcodeCounter.so

最后，回到build的同级目录，建立test目录，在内部放置测试用例，这里新建 testOpcodeCounter.c

```
#include <stdio.h>

int func(int a,int b){ // a*b
    int sum = 0;
        for(int i = 0; i < a; i++) {
            sum += b;
        }
        return sum;
}

int main(){
    int a = 6;
    int b = 7;
    printf("%d * %d = %d\n", a, b, func(a, b));

    return 0;
}
```

运行shell命令

```
clang -emit-llvm -S testOpcodeCounter.c -o test.ll // 生成汇编码文件
opt -load ../build/lib/LLVMOpcodeCounter.so -oc test.ll -o test.bc
```

这里的 -oc参数是我们在OpcodeCounter.cpp 中进行的设定。执行效果如下：

```
ollvm@vm:~/Desktop/LLVM/test$ opt -load ../build/lib/LLVMOpcodeCounter.so -oc test.ll -o test.bc
Function name: func
add : 2
alloca : 4
br : 4
icmp : 1
load : 6
ret : 1
store : 6

Function name: main
alloca : 3
call : 2
load : 4
ret : 1
store : 3

ollvm@vm:~/Desktop/LLVM/test$
```

0x02 命令行编译

这种方法不依赖于cmake文件，直接使用命令行编译，步骤相对简单

具体的流程：

1. 新建文件夹，然后将pass的相关文件放置在内部
2. 使用clang或者clang++ 生成相对应的 .o文件
3. 生成相对应的 .so文件
4. 编写测试用例，进行pass效果检测

具体的实例

这里我们使用方法一中提到的OpcodeCounter。

首先，新建目录pass，将OpcodeCounter.cpp 复制到文件夹内。

然后，使用shell命令

```
clang -emit-llvm -S testOpcodeCounter.c -o test.ll // 这里的testOpcodeCounter.c和方法一中的一致
clang++ -c OpcodeCounter.cpp `llvm-config --cxxflags` // 生成 .o文件
clang++ -shared OpcodeCounter.cpp -o OpcodeCounter.so `llvm-config --ldflags` // 生成 .so文件

// 也可以使用合并的命令
// clang -shared OpcodeCounter.cpp -o OpcodeCounter.so `llvm-config --cxxflags --ldflags`

opt -load ./OpcodeCounter.so test.ll -o test.bc
```

其中，llvm-config提供的 --cxxflags 和 --ldflags 参数方便查找LLVM的头文件和库文件。

运行结果同方法一。

0x03 树外编译

树外编译，指的是不在LLVM的源代码包下，且同时配置cmake文件，一方面尽可能的加快构建的时间，另一方面通过cmake等配置文件，方便解决依赖关系和命令的编写。对于文件架构图，可以参照官网给出的样例

```
<project dir>/
    |
    CMakeLists.txt
    <pass name>/
        |
        CMakeLists.txt
        Pass.cpp
        ...
```

这里，我们的文件树如下：

```
<ollvm>/
    |
    CMakeLists.txt
    <build>
    <test>
    <Obfuscation>/
        |
        <include>
        CMakeLists.txt
        BogusControlFlow.cpp
        Flattening.cpp
        OpcodeCounter.cpp
        ...
```

- 外层的CMakeLists.txt 中进行了固定的设置，一般不需要改动，内容如下：

```
cmake_minimum_required(VERSION 3.10) // 这里根据安装的cmake的版本进行配置，一般不要太低
find_package(LLVM REQUIRED CONFIG)
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
link_directories(${LLVM_LIBRARY_DIRS})

add_subdirectory(Obfuscation)
```

- build 文件夹，方便我们后续的pass编译，因为编译时的命令是固定的，写成shell脚本如下：

  ```
  cmake .. -DLLVM_DIR=/usr/local/lib/cmake/llvm/ //这里根据本地LLVMConfig.cmake的位置进行改变
  cmake --build . -- -j$(nproc)
  ```

- test文件夹内，放了测试用例
- Obfuscation文件夹内是pass相关的头文件和源文件，其中头文件放置于include子文件夹内

具体的流程

1. 在Obfuscation目录下添加pass文件，并修改同目录下的CMakeLists.txt
2. 在build目录下进行编译，这里可以直接运行shell脚本 run.bash，在build的子目录 Obfuscation/ 内生成 libLLVMObfuscation.so 文件
3. 在test目录下创建测试用例，并进行pass效果的检测

具体的实例

这里我们仍使用OpcodeCounter进行测试。

首先，我们将其放置于Obfuscation目录下。同时修改CMakeLists.txt文件如下：

```
cmake_minimum_required(VERSION 3.10)
include_directories(./include)
add_library(LLVMObfuscation MODULE
  BogusControlFlow.cpp
  Flattening.cpp
  OpcodeCounter.cpp // 在这里添加相应的文件
)
// LLVM is (typically) built with no C++ RTTI. We need to match that;
// otherwise, we'll get linker errors about missing RTTI data.
set_target_properties(LLVMObfuscation PROPERTIES
    COMPILE_FLAGS "-fno-rtti"
)
add_dependencies(LLVMObfuscation intrinsics_gen)
```

然后，到build目录下，运行run.bash脚本，生成.so文件

```
bash run.bash
```

最后，在test目录下创建文件testOpcodeCounter.c，内容同之前方法中提到的一样，之后运行shell命令

```
clang -emit-llvm -S testOpcodeCounter.c -o test.ll
opt -load ../build/Obfuscation/libLLVMObfuscation.so -oc test.ll -o test.bc
```

运行结果一致，但速度上明显快于第一种；命令参数上较第二种也更简单。