Kaleidoscope

代码解释(3)

万花筒语言 - LLVM 新手入门教程

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl04.html

PLCT - SSC

Ch3 vs Ch4

```
ready> def dou(x) (1+x)+(1+x);
ready> Read function definition:define double @dou(double %x) {
entry:
  %addtmp = fadd double 1.000000e+00, %x
  %addtmp1 = fadd double 1.000000e+00, %x
  %addtmp2 = fadd double %addtmp, %addtmp1
  ret double %addtmp2
}
```

```
ready> def dou(x) (1+x)+(1+x);
ready> Read function definition:define double @dou(double %x) {
entry:
  %addtmp = fadd double %x, 1.000000e+00
  %addtmp2 = fadd double %addtmp, %addtmp
  ret double %addtmp2
}
```

```
ready>          def             dou(x)          (1+x)+(1+x);
命令提示符      函数定义符      函数名(参数)    函数体

ready> Read function definition:

define          double                  @dou            (double %x              ) {
函数定义符      64位浮点数              全局变量                局部变量

函数体为Basic Blocks

entry:
符号表入口标签

        %addtmp         =       fadd            double          %x      ,       1.000000e+00
        局部变量                浮点运算加法    运算类型double                   浮点数

        %addtmp2        =       fadd            double          %addtmp ,       %addtmp

        ret             double          %addtmp2
        return          double类型      最后的结果

}
```

预处理、命名空间

```
#include "../include/KaleidoscopeJIT.h" // 为本教程编译了一个简单的JIT,后续Building a JIT in LLVM会涉及到
#include "llvm/ADT/APFloat.h"           // arbitrary precision float
#include "llvm/ADT/STLExtras.h"         // 标准库拓展
#include "llvm/IR/BasicBlock.h"         // Basic Block
#include "llvm/IR/Constants.h"          // 常量
#include "llvm/IR/DerivedTypes.h"       // derived types的实现
// arrays of x" or "structure of x, y, z" or "function returning x taking (y,z) as parameters"
#include "llvm/IR/Function.h"           // 函数
#include "llvm/IR/IRBuilder.h"          // IRBuilder
#include "llvm/IR/LLVMContext.h"        // LLVMContext负责global类型的管理
#include "llvm/IR/LegacyPassManager.h"  // 保存,维护,优化 Pass的执行
#include "llvm/IR/Module.h"             // Module
#include "llvm/IR/Type.h"               // 类型的声明
#include "llvm/IR/Verifier.h"           // 函数验证
#include "llvm/Support/TargetSelect.h"  // 用以确保特定类的目标被链接到主应用的执行程序,并正确初始化
#include "llvm/Target/TargetMachine.h"  // TargetMachine LLVMTargetMachine
#include "llvm/Transforms/InstCombine/InstCombine.h" // instcombine pass
#include "llvm/Transforms/Scalar.h"     // expose pass
#include "llvm/Transforms/Scalar/GVN.h" // Global Value Numbering pass
#include <algorithm>
#include <cassert>
#include <cctype>
#include <cstdint>
#include <cstdio>

using namespace llvm::orc;              // On Request Compilation
```

全局变量

```
static std::unique_ptr<legacy::FunctionPassManager> TheFPM;
// Pass 管理
static std::unique_ptr<KaleidoscopeJIT> TheJIT;
// JIT
static std::map<std::string, std::unique_ptr<PrototypeAST>> FunctionProtos;
// 函数声明映射
```

getFunction函数

```
Function *getFunction(std::string Name) {
  // 检查module里面是否有,也就是说是否已经生成过函数的代码.
  if (auto *F = TheModule->getFunction(Name))
    return F;

  // 然后检查是否有函数的第一次定义,继而去生成代码.
  auto FI = FunctionProtos.find(Name);
  if (FI != FunctionProtos.end())
    return FI->second->codegen();

  // 如果不存在声明,返回空指针
  return nullptr;
}
```

CallExprAST::codegen()

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i)
{
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp"
);
}
```

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i)
{
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp"
);
}
```

FunctionAST::codegen()

```
Function *FunctionAST::codegen() {


  Function *TheFunction = TheModule-
>getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;

    verifyFunction(*TheFunction);



    return TheFunction;
  }

  return nullptr;
}
```

```
Function *FunctionAST::codegen() {
  auto &P = *Proto;
  FunctionProtos[Proto->getName()] = std::move(Proto);
  Function *TheFunction = getFunction(P.getName());
  // 使用getFunction来进行判断,是否是第一次定义



  if (!TheFunction)
    return nullptr;

    verifyFunction(*TheFunction);

    TheFPM->run(*TheFunction); // 对代码进行优化

    return TheFunction;
  }

  return nullptr;
}
```

InitializeModuleAndPassManager函数

```
static void InitializeModuleAndPassManager() {
  // 新建一个module,并于jit绑定
  TheModule = std::make_unique<Module>("my cool jit", TheContext);
  TheModule->setDataLayout(TheJIT->getTargetMachine().createDataLayout());

  // 与pass manager绑定
  TheFPM = std::make_unique<legacy::FunctionPassManager>(TheModule.get());

  // “peephole” optimizations and bit-twiddling optimizations. 窥孔优化 与 位运算优化
  TheFPM->add(createInstructionCombiningPass());
  // Reassociate expressions.重新关联表达式
  TheFPM->add(createReassociatePass());
  // Eliminate Common SubExpressions. 子公共表达式消除
  TheFPM->add(createGVNPass());
  // Simplify the control flow graph 简化控制流程图
  TheFPM->add(createCFGSimplificationPass());

  TheFPM->doInitialization(); // 执行优化
}
```

处理定义

```
static void HandleDefinition() {
  if (auto FnAST = ParseDefinition()) {
    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Read function definition:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
      TheJIT->addModule(std::move(TheModule));
      InitializeModuleAndPassManager();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

处理extern

```
static void HandleExtern() {
  if (auto ProtoAST = ParseExtern()) {
    if (auto *FnIR = ProtoAST->codegen()) {
      fprintf(stderr, "Read extern: ");
      FnIR->print(errs());
      fprintf(stderr, "\n");
      FunctionProtos[ProtoAST->getName()] = std::move(ProtoAST);
    }
  } else {
    getNextToken();
  }
}
```

顶层表达式处理

```
static void HandleTopLevelExpression() {
  if (auto FnAST = ParseTopLevelExpr()) {

    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Read top-level expression:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
    }



  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

```
static void HandleTopLevelExpression() {
  if (auto FnAST = ParseTopLevelExpr()) {

    if (FnAST->codegen()) {
      auto H = TheJIT->addModule(std::move(TheModule));
      InitializeModuleAndPassManager();

      // 利用jit直接解析匿名函数
      auto ExprSymbol = TheJIT->findSymbol("__anon_expr");
      assert(ExprSymbol && "Function not found");
      double (*FP)() = (double (*)())(intptr_t)cantFail(ExprSymbol
.getAddress());
      fprintf(stderr, "Evaluated to %f\n", FP());
      TheJIT->removeModule(H);
    }


  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

拓展 库函数

```
// 与外部接口,用以使用库函数

#ifdef _WIN32
#define DLLEXPORT __declspec(dllexport) //省掉在DEF文件中手工定义导出哪些函数
#else
#define DLLEXPORT
#endif

// 读取一个double类型数字
extern “C” DLLEXPORT double putchard(double X) {
// extern “C” 指明使用C的编译和链接

  fputc((char)X, stderr);
  return 0;
}

// 输出一个double类型数字
extern "C" DLLEXPORT double printd(double X) {
  fprintf(stderr, "%f\n", X);
  return 0;
}
```

main函数

```
int main() {



  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40; // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  TheModule = std::make_unique<Module>("my cool jit", TheC
ontext);




  MainLoop();

  TheModule->print(errs(), nullptr);
  return 0;
}
```

```
int main() {
  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();
  InitializeNativeTargetAsmParser();
// 获取目标机器信息


  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40; // highest.


  fprintf(stderr, "ready> ");
  getNextToken();


  TheJIT = std::make_unique<KaleidoscopeJIT>();
// 声明JIT
  InitializeModuleAndPassManager();
// 初始化module和pass manager

 MainLoop();

  return 0;
}
```

def dou(x) (1+x)+(1+x);

```
main()
        InitializeNativeTarget(); 获取目标机器信息
        InitializeNativeTargetAsmPrinter();
        InitializeNativeTargetAsmParser();
        getNextToken(); 获取第一个token
        TheJIT = std::make_unique<KaleidoscopeJIT>(); 构建KaleidoscopeJIT类型的TheJIT
        InitializeModuleAndPassManager(); 初始化module和pass manager
                TheModule = std::make_unique<Module> 构建Module类型的TheModule
                TheModule->setDataLayout(TheJIT->getTargetMachine().createDataLayout()); 与jit绑定
                TheFPM = std::make_unique<legacy::FunctionPassManager>(TheModule.get()); 与pass manager绑定
                TheFPM->add(createInstructionCombiningPass()); 窥孔优化 位运算优化
                TheFPM->add(createReassociatePass()); 重新关联表达式
                TheFPM->add(createGVNPass()); 公共子表达式消除
                TheFPM->add(createCFGSimplificationPass()); 简化控制流图
                TheFPM->doInitialization(); pass 初始化
        MainLoop();
                HandleDefinition() 处理函数
                        FnAST = ParseDefinition() 解析函数
                                ParsePrototype() 解析函数声明
                                ParseExpression() 解析函数体
                        FnAST->codegen() 函数代码生成
                                auto &P = *Proto;
                                FunctionProtos[Proto->getName()] = std::move(Proto); 映射函数名到FunctionProtos
                                Function *TheFunction = getFunction(P.getName()) getFunction获取函数声明代码
                                        TheModule->getFunction(Name) module查询函数名称,如果已经生成过
                                        auto FI = FunctionProtos.find(Name); FunctionProtos如果有映射过函数
                                        return FI->second->codegen() 第一次就生成代码
                                BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
                                函数体代码生成
```

```
TheJIT = std::make_unique<KaleidoscopeJIT>(); 构建KaleidoscopeJIT类型的TheJIT
InitializeModuleAndPassManager(); 初始化module和pass manager
        TheModule = std::make_unique<Module> 构建Module类型的TheModule
        TheModule->setDataLayout(TheJIT->getTargetMachine().createDataLayout()); 与jit绑定
        TheFPM = std::make_unique<legacy::FunctionPassManager>(TheModule.get()); 与pass manager绑定
        TheFPM->add(createInstructionCombiningPass()); 窥孔优化 位运算优化
        TheFPM->add(createReassociatePass()); 重新关联表达式
        TheFPM->add(createGVNPass()); 公共子表达式消除
        TheFPM->add(createCFGSimplificationPass()); 简化控制流图
        TheFPM->doInitialization(); pass 初始化
MainLoop();
        HandleDefinition() 处理函数
                FnAST = ParseDefinition() 解析函数
                        ParsePrototype() 解析函数声明
                        ParseExpression() 解析函数体
                FnAST->codegen() 函数代码生成
                        auto &P = *Proto;
                        FunctionProtos[Proto->getName()] = std::move(Proto); 映射函数名到FunctionProtos
                        Function *TheFunction = getFunction(P.getName()) getFunction获取函数声明代码
                                TheModule->getFunction(Name) module查询函数名称,如果已经生成过
                                auto FI = FunctionProtos.find(Name); FunctionProtos如果有映射过函数
                                return FI->second->codegen() 第一次就生成代码
                        BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
                        函数体代码生成
                        Value *RetVal = Body->codegen() 返回值生成与接收
                        TheFPM->run(*TheFunction); 进行优化
                        return TheFunction; 返回优化过的函数
                FnIR->print(errs()); 打印结果
                TheJIT->addModule(std::move(TheModule)); 函数、变量定义只有先添加进来，后面才能执行
                InitializeModuleAndPassManager();
```