中山大學 SUN YAT-SEN UNIVERSITY

计算机学院 COMPUTER SCIENCE AND ENGINEERING

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

编译器构造实验

实验四 中间代码优化

Yat Compiler Construction with AI

yatcc-ai.com

中山大学 计算机学院

国家超级计算广州中心

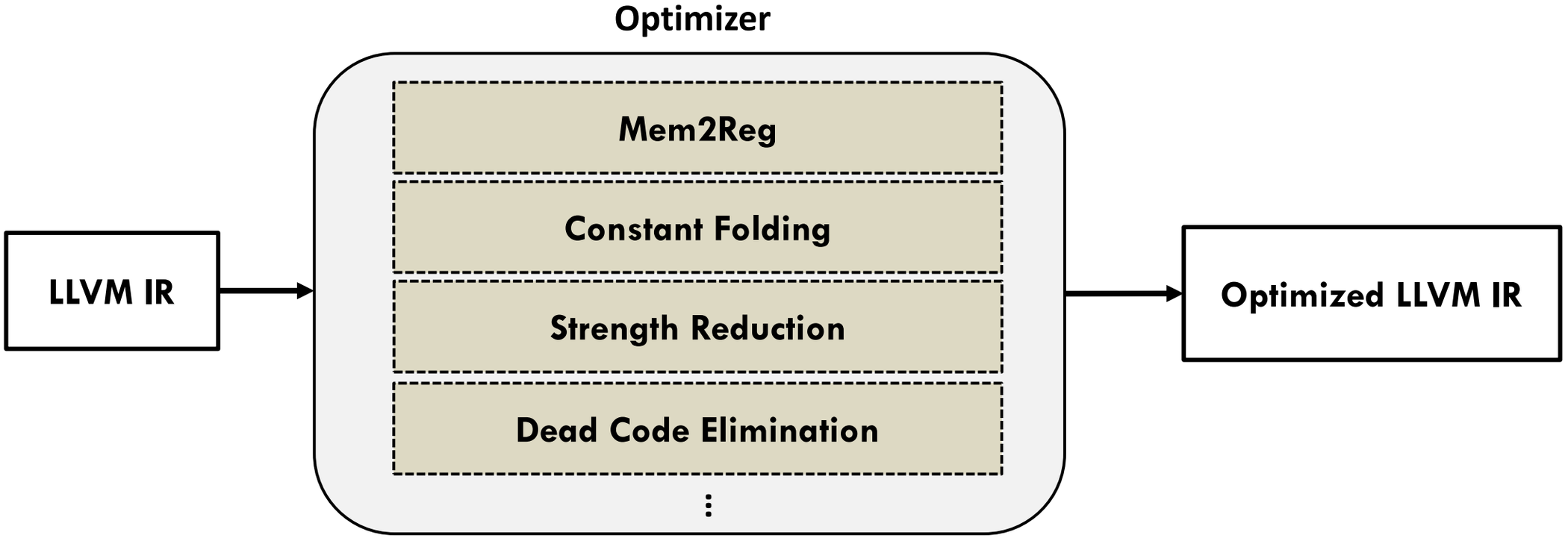
2025.5

OUTLINE

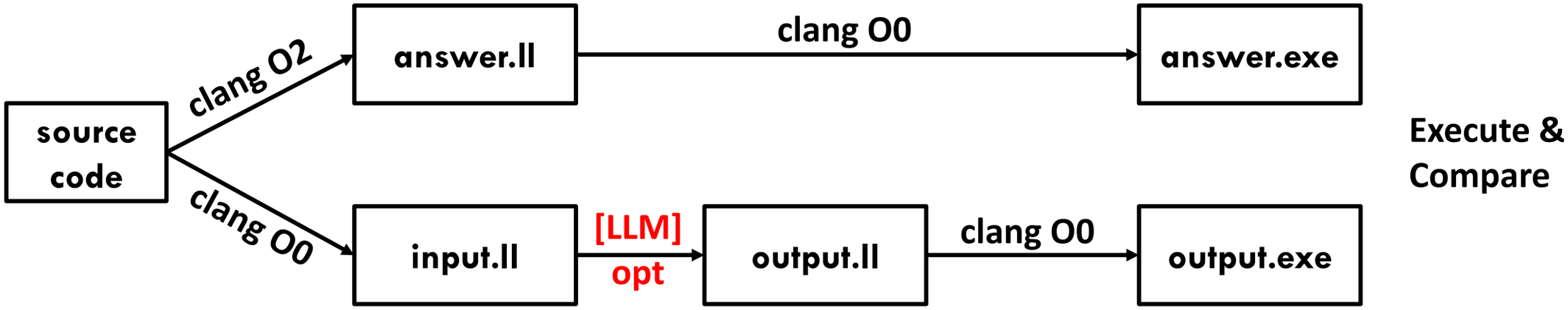
目　录

中山大学计算机学院
School of Computer Science & Engineering

- 总体任务
  - 编写优化器，对实验三输出的**LLVM IR**进行优化，得到**优化后的IR**
  - 根据 LLVM 优化工作流，编写一系列 LLVM Pass （LLVM 优化遍）
    - 实现编译优化中的传统经典算法（传统方法：**task4-classic**）
    - LLM 加持下的 LLVM Pass（大语言模型方法：**task-llm**）



- **评分标准：在线测评空间评分，本地评分仅作参考**
  - 以clang O2作为比较对象，进行优化正确性与性能测试：
    - 正确性：判断优化后IR (output.ll)的输出与返回值是否正确
    - 性能：比较answer.exe和output.exe的执行时间，两者的比值为该测例的得分，各测例得分取平均得到性能得分
  - 注意：task4-classic 和 task-llm 分别占实验总分的25分和5分。其中task4-classic达到性能得分的60分为满分，后者达到性能得分的75分为满分。



- **禁止行为（违者视为抄袭）：**
  - 面向测例编程，包括但不限于：识别文件名、输入、特定代码段等手段进行代码优化
  - 直接调用LLVM内置的Transform Pass，或者直接复制LLVM提供的Transform Pass代码
- **允许行为：**
  - 在理解LLVM优化源代码的基础上将其简化移植
  - 使用LLVM内置的Analysis Pass获取优化所需的信息
  - 调用 LLM 来获得期望的编译优化信息辅助优化决策以及进行优化

实验四作业提交时间：【6.19】

OUTLINE

目　录

中山大学计算机学院
School of Computer Science & Engineering

- 总体框架代码
  - 主目录下实现传统方法（task4-classic）
  - llm 目录下实现 LLM 方法（task4-llm）
  - 共用一个 main.cpp 入口文件
    - 以宏定义区分不同方法的逻辑
    - 分别编译产生两种构建目标
    - task4-llm 必须写在 TASK4_LLM 中
      - 否则可能产生编译错误

task4-llm

```
#ifdef TASK4_LLM
  /* 添加 LLM 辅助编译优化的 LLVM Pass */
#else
task4-classic
  /* 添加传统方法的 LLVM Pass */
#endif

  /* 其他公共代码逻辑 */
}
```

```
4
|-- CMakeLists.txt
|-- ConstantFolding.cpp
|-- ConstantFolding.hpp
|-- Mem2Reg.cpp
|-- Mem2Reg.hpp
|-- README.md
|-- StaticCallCounter.cpp
|-- StaticCallCounter.hpp
|-- StaticCallCounterPrinter.cpp
|-- StaticCallCounterPrinter.hpp
|-- llm
|   |-- LLMHelper.cpp
|   |-- LLMHelper.hpp
|   |-- PassSequencePredict.cpp
|   |-- PassSequencePredict.hpp
|   |-- __init__.py
|   `-- prompts
|       |-- PassSeqPredSysPrTpl.xml
|       |-- PassSeqPredUserPrTpl.xml
|       |-- PassSummarySysPrTpl.xml
|       `-- PassSummaryUserPrTpl.xml
`-- main.cpp
```

task4-classic

task4-llm

两种方法共用一个 入口文件

- **传统方法框架介绍：**
  - 定义和实现 Pass（*.hpp、*.cpp）
    - 建议每个Pass对应一个*.hpp/*.cpp
  - 注册 Pass（main.cpp）

- **两种Pass类别**
  - Analysis Pass：分析并返回信息（e.g. LoopAnalysis返回函数中的循环信息）
  - Transform Pass：实现代码变换（e.g. LICM根据LoopAnalysis的分析结果实现循环无关代码移动）

- 定义Pass
  - Transform Pass： 以 ConstantFolding.h 为例

```
class ConstantFolding : public llvm::PassInfoMixin<ConstantFolding>
{
public:
  explicit ConstantFolding(llvm::raw_ostream& out)
    : mOut(out)
  {
  }

  llvm::PreservedAnalyses run(llvm::Module& mod,
                              llvm::ModuleAnalysisManager& mam);

private:
  llvm::raw_ostream& mOut;
};
```

继承于llvm::PassInfoMixin对象

修改部分：

1. Pass名称
2. 继承对象
3. 构造函数

- 定义Pass
  - Analysis Pass： 以 StaticCallCounter.hpp 为例

```
class StaticCallCounter : public llvm::AnalysisInfoMixin<StaticCallCounter>
{
public:
  using Result = llvm::MapVector<const llvm::Function*, unsigned>;
  Result run(llvm::Module& mod, llvm::ModuleAnalysisManager&);

private:
  // A special type used by analysis passes to provide an address that
  // identifies that particular analysis pass type.
  static llvm::AnalysisKey Key;
  friend struct llvm::AnalysisInfoMixin<StaticCallCounter>;
};
```

继承于
llvm::AnalysisInfoMixin对象

修改部分：
1. Pass名称
2. 继承对象
3. run函数返回值定义
4. 私有静态变量Key
5. 友元类声明

- 实现Pass
  - Transform Pass：以 ConstantFolding.cpp 为例

```
PreservedAnalyses
ConstantFolding::run(Module& mod, ModuleAnalysisManager& mam)
{
  int constFoldTimes = 0;

  // 遍历所有函数
  for (auto& func : mod) {
    // 遍历每个函数的基本块
    for (auto& bb : func) {
      std::vector<Instruction*> instToErase;
      // 遍历每个基本块的指令
      for (auto& inst : bb) {
        // 对常量指令进行折叠
        // ...
      }
      // 统一删除被折叠为常量的指令
      for (auto& i : instToErase)
        i->eraseFromParent();
    }
  }

  mOut << "ConstantFolding running...\nTo eliminate " << constFoldTimes
       << " instructions\n";
  return PreservedAnalyses::all();
}
```

主体代码模板：

1. 遍历所有指令

2. 对目标指令进行分析与操作(增删改查)

3. 统一删除指令

4. 函数返回

- 实现Pass
  - Analysis Pass： 以 StaticCallCounter.cpp 为例

```
StaticCallCounter::Result
StaticCallCounter::run(Module& mod, ModuleAnalysisManager&)
{
  MapVector<const Function*, unsigned> result;

  for (auto& func : mod) {
    for (auto& bb : func) {
      for (auto& inst : bb) {
        // 寻找函数调用指令...
        Function *directInvoc;

        // 统计函数在源代码中被调用次数
        auto callCount = result.find(directInvoc);
        if (result.end() == callCount) {
          callCount = result.insert({ directInvoc, 0 }).first;
        }
        ++callCount->second;
      }
    }
  }

  return result;
}
AnalysisKey StaticCallCounter::Key;
```

1. 代码结构与Transform Pass基本相同

2. 返回自定义的信息(result)

3. 声明静态成员变量Key

- 注册Pass

```
void
opt(llvm::Module& mod)
{
  using namespace llvm;

  // 定义分析pass的管理器
  LoopAnalysisManager lam;
  FunctionAnalysisManager fam;
  CGSCCAnalysisManager cgam;
  ModuleAnalysisManager mam;
  ModulePassManager mpm;

  // 注册分析pass的管理器
  PassBuilder pb;
  pb.registerModuleAnalyses(mam);
  pb.registerCGSCCAnalyses(cgam);
  pb.registerFunctionAnalyses(fam);
  pb.registerLoopAnalyses(lam);
  pb.crossRegisterProxies(lam, fam, cgam, mam);

  // 添加分析pass到管理器中
  mam.registerPass([]() { return StaticCallCounter(); });

  // 添加优化pass到管理器中
  mpm.addPass(StaticCallCounterPrinter(llvm::errs()));
  mpm.addPass(ConstantFolding(llvm::errs()));

  // 运行优化pass
  mpm.run(mod, mam);
}
```

1. 只有被注册的Pass才能被调用
2. Transform Pass的添加顺序即执行优化的顺序

注册Analysis Pass

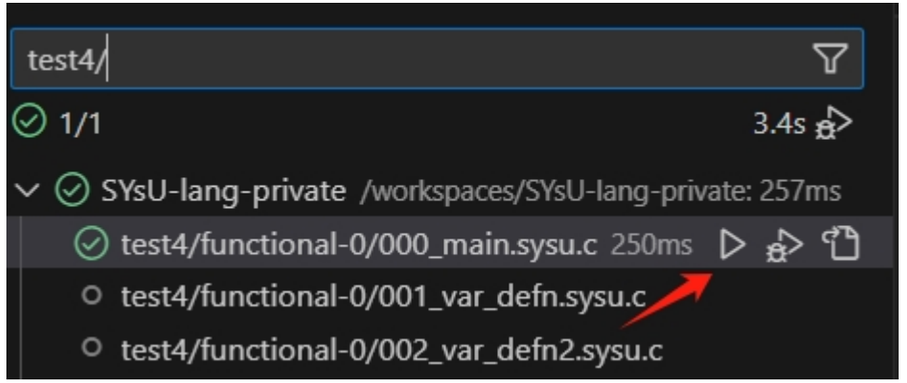
注册Transform Pass

• 使用Analysis Pass

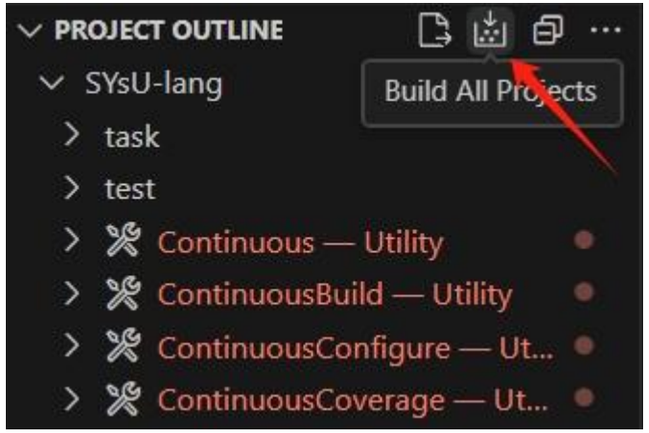
```
PreservedAnalyses
StaticCallCounterPrinter::run(Module& mod, ModuleAnalysisManager& mam)
{
  // 通过MAM执行StaticCallCounter并返回分析结果
  auto directCalls = mam.getResult<StaticCallCounter>(mod);

  mOut << "================================================\n";
  mOut << "     sysu-optimizer: static analysis results\n";
  mOut << "================================================\n";
  mOut << "       NAME              #N DIRECT CALLS\n";
  mOut << "------------------------------------------------\n";

  for (auto& callCount : directCalls) {
    std::string funcName = callCount.first->getName().str();
    funcName.resize(20, ' ');
    mOut << "       " << funcName << "    " << callCount.second << "\n";
  }

  mOut << "------------------------------------------------\n\n";
  return PreservedAnalyses::all();
}
```

1. 调用前在mam注册
2. 在其他Pass中通过mam.getResult<…>(mod)调用，获取分析结果

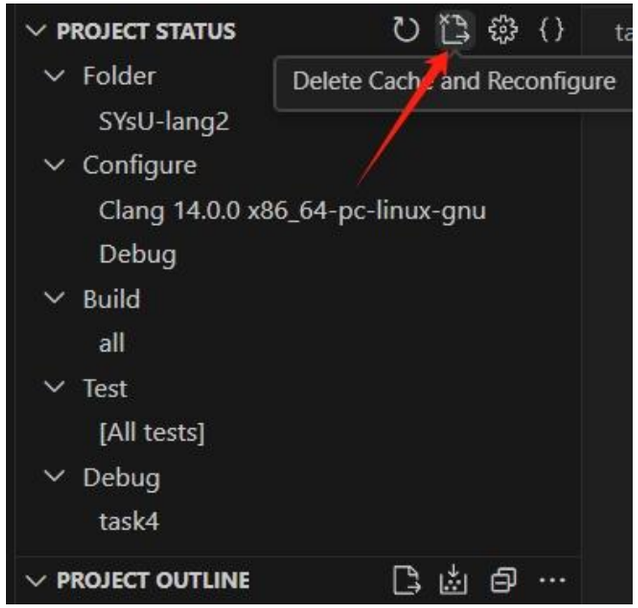
- 输出调试
  - 在VS Code中选择test4中的对应测例运行
  - 结果保存在/workspaces/YatCC/build/test/task4/Testing/Temporary/LastTest.log中

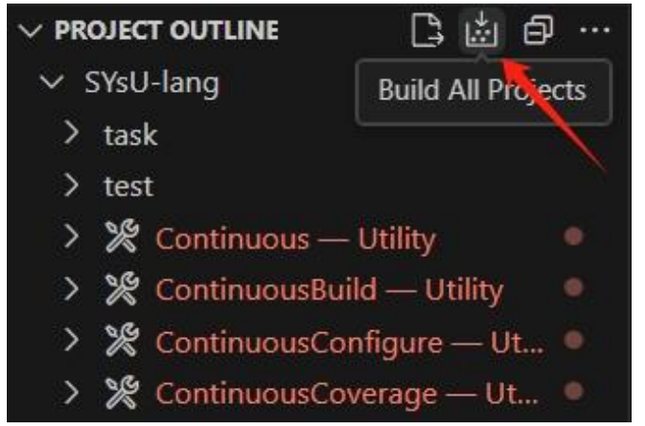


- 测例修改
  - 在performance文件夹增加、修改测例
  - 删除build文件夹后重新编译task4、生成标准答案
  - 也可以使用文档中提供的命令行调试方法



- 常见问题：
  - 添加新Pass后编译报错：undefined reference
    - 添加新文件后需要更新CMake配置文件
  - 使用命令行调试方法时更新代码没有效果
    - 请检查是否重新编译task4可执行文件





- 实例分析
  - 下面以强度削弱（Strength Reduction）为例，为大家提供查找信息的几个途径

强度削弱

代表测例：`bitset*.sysu.c` 、 `crypto-*.sysu,c`

难度：★☆☆☆☆

强度削弱（Strength Reduction）将一条高计算复杂度的指令，转化为一条或多条低复杂度的指令。强度削弱是一个很简单但非常有效的优化方式，因为这样的优化机会广泛存在于我们编写的程序之中。在算法比赛中我们常常在编程时使用到这些技巧，但编译器使得我们无需为了性能小心翼翼地编程而获得高性能的程序。下面是一些直观的例子：

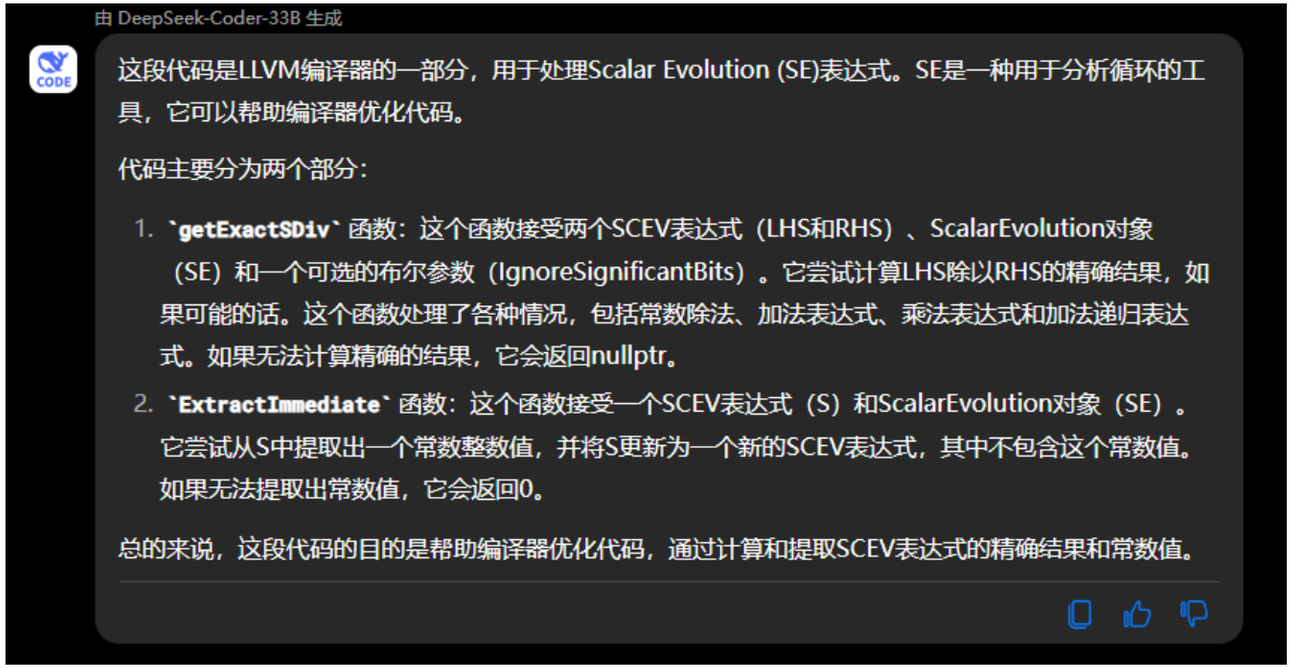
```C++
// 优化前
int a = x * 8, b = x % 32, c = x * 3;

// 优化后
int a = x << 3, b = x - (x / 32) << 5, c = (x << 1) + x;
```

- LLVM官网
  - LLVM官网有各种Pass的详细源码实现
  - 同学们可以参照这些Pass来实现自己的优化

```
747   // Distribute the sdiv over add operands, if the add doesn't overflow.
748   if (const SCEVAddExpr *Add = dyn_cast<SCEVAddExpr>(LHS)) {
749     if (IgnoreSignificantBits || isAddSExtable(Add, SE)) {
750       SmallVector<const SCEV *, 8> Ops;
751       for (const SCEV *S : Add->operands()) {
752         const SCEV *Op = getExactSDiv(S, RHS, SE, IgnoreSignificantBits);
753         if (!Op) return nullptr;
754         Ops.push_back(Op);
755       }
756       return SE.getAddExpr(Ops);
757     }
758     return nullptr;
759   }
760
761   // Check for a multiply operand that we can pull RHS out of.
762   if (const SCEVMulExpr *Mul = dyn_cast<SCEVMulExpr>(LHS)) {
763     if (IgnoreSignificantBits || isMulSExtable(Mul, SE)) {
764       // Handle special case C1*X*Y /s C2*X*Y.
765       if (const SCEVMulExpr *MulRHS = dyn_cast<SCEVMulExpr>(RHS)) {
766         if (IgnoreSignificantBits || isMulSExtable(MulRHS, SE)) {
767           const SCEVConstant *LC = dyn_cast<SCEVConstant>(Mul->getOperand(0));
768           const SCEVConstant *RC =
769               dyn_cast<SCEVConstant>(MulRHS->getOperand(0));
770           if (LC && RC) {
771             SmallVector<const SCEV *, 4> LOps(drop_begin(Mul->operands()));
772             SmallVector<const SCEV *, 4> ROps(drop_begin(MulRHS->operands()));
773             if (LOps == ROps)
774               return getExactSDiv(LC, RC, SE, IgnoreSignificantBits);
775           }
776         }
777       }
778
779       SmallVector<const SCEV *, 4> Ops;
780       bool Found = false;
781       for (const SCEV *S : Mul->operands()) {
782         if (!Found)
783           if (const SCEV *Q = getExactSDiv(S, RHS, SE,
784                                            IgnoreSignificantBits)) {
785             S = Q;
786             Found = true;
787           }
788         Ops.push_back(S);
789       }
790       return Found ? SE.getMulExpr(Ops) : nullptr;
791     }
792     return nullptr;
793   }
794
795   // Otherwise we don't know.
796   return nullptr;
797 }
```

- 大语言模型
  - 如果大家觉得阅读代码有问题，或者看不懂函数的API，请务必寻求大模型的帮助（LLM for LLVM）

输入
这段LLVM代码什么意思： [LLVM Code]



- 编译比赛参考
  - YatCC编译器中同样也实现了一系列的Pass，比LLVM的Pass可读性高
  - 数据结构不完全是LLVM的数据结构，同学们可以参考其实现思路

```
void strengthReduction(BlockPtr block) {
    int regCount = 0;
    for (auto &bb : block->basicBlocks)
    {
        unordered_set<InstructionPtr> toDelete;
        vector<InstructionPtr> toSave;
        for (int i = 0; i < bb->instructions.size(); i ++)
        {
            auto &ins = bb->instructions[i];
            if (ins->type == InsID::Binary)
            {
                auto binary = dynamic_cast<BinaryInstruction *>(ins.get());
                auto lhs = binary->a;
                auto rhs = binary->b;
                if (binary->op == '*')
                {
                    // dst = 2^shamt * rhs
                    if (lhs->isConst && rhs->type->ID == IntID)
                    {
                        auto const_lhs = dynamic_cast<Const *>(lhs.get());
                        if (const_lhs->intVal > 0 && isExp(const_lhs->intVal))
                        {
                            auto shamt = Const::getConst(Type::getInt(),int(log2(const_lhs->intVal)), "strengthReduction%" + to_string(regCount++));
                            // op ',' indicates arithmetic left shift('<<')
                            auto new_binary = new BinaryInstruction(rhs, shamt, ',', ins->basicblock);
                            replaceVarByVar(binary->reg, new_binary->reg);
                            bb->instructions[i] = shared_ptr<Instruction>(new_binary);
                        }
                    }
                    // dst = lhs * constant
                    else if (rhs->isConst && lhs->type->ID == IntID)
                    {
                        auto const_rhs = dynamic_cast<Const *>(rhs.get());
                        // dst = lhs * constant / constant
                        if(i + 1 < bb->instructions.size() && bb->instructions[i + 1]->type == Binary) {
                            auto nextBinary = dynamic_cast<BinaryInstruction *>(bb->instructions[i + 1].get());
                            if(nextBinary->op == '/' && nextBinary->b == rhs) {
                                replaceVarByVar(nextBinary->reg, binary->a);
                                toDelete.insert(bb->instructions[i]);
                                toDelete.insert(bb->instructions[i + 1]);
                                i ++;
                            }
                        }
```

- # 优化算法整体介绍

  根据各种算例的情况以及难易程度，推荐的Pass实现顺序如下：

  - 常量传播/常量折叠
  - 强度削弱
  - Mem2reg（已经实现）
  - 公共子表达式消除
  - 死代码/死存储消除
  - 循环无关变量移动
  - 函数内联
  - 循环展开

- 公共子表达式消除
  - CSE的基本原理就是复用之前计算的结果。可通过搜索算法实现

公共子表达式消除

代表测例：`hoist-*.sysu.c`

难度：★★☆☆☆

公共子表达式消除（Common Subexpression Elimination，CSE）是一个非常经典的优化算法，如果一个表达式E在计算得到后没有变化，且同时作为多条指令的操作数被使用，那么E能够被称为公共子表达式：

```cpp
// (a + b)是d和e的公共子表达式
int d = a + b - c;
int e = a + b + c;

// CSE优化后
int tmp = a + b;
int d = tmp - c;
int e = tmp + c;
```

- 死代码/死存储消除
  - 两种消除都是删除对输出结果无影响的代码，可通过检查每条指令的use-def链来实现

死代码消除

代表测例：

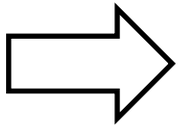
难度：★★★☆☆

死代码消除（Dead Code Elimination，DCE）是一种将对全局变量、输出和返回结果无影响（无副作用）的指令删除的优化。对于一个程序而言，满足以上条件的指令我们认为它在进行一些无意义的计算，其计算结果对于外界来说是不可感知的，因此即使删除也不会对程序的结果产生影响。下面是一个简单的例子：

```C++
// 优化前
int main() {
  int sum = 0;
  int j = 0;
  for(int i = 0; i < 10; i ++) sum += i;
  ...
  print(j);
  return 0;
}

// 优化后
int main() {
  int j = 0;
  print(j);
  return 0;
}
```

- 循环无关变量移动
  - 通过分析可以发现在循环中有运算的值不受循环的影响，那么就可以把它提升到循环的外面，是循环展开的前置条件。

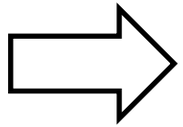
```
for(int i = 0; i < n;
i++)
{
  a[i] += x*x;
}
```



```
int x_2 = x*x;
for(int i = 0; i < n; i++)
{
  a[i] += x_2;
}
```

- 函数内联
  - 函数内联是一种常见的编译优化，可以消除函数调用的开销，同时提供更多的优化机会。函数内联即将函数调用语句替换为调用函数本身所包含的指令。从IR的角度来看，即在函数调用语句对应的地方，插入对应函数的基本块。

```
int mul(int a, int b)
{
  return a*b;
}

int c = mul(a,b);
```



```
int c = a*b;
```

- 循环展开
  - 此处仅介绍循环变量为常数的循环展开。循环展开就是将原本的循环删去，变成多条重复指令，结合后续优化可以显著提升代码性能

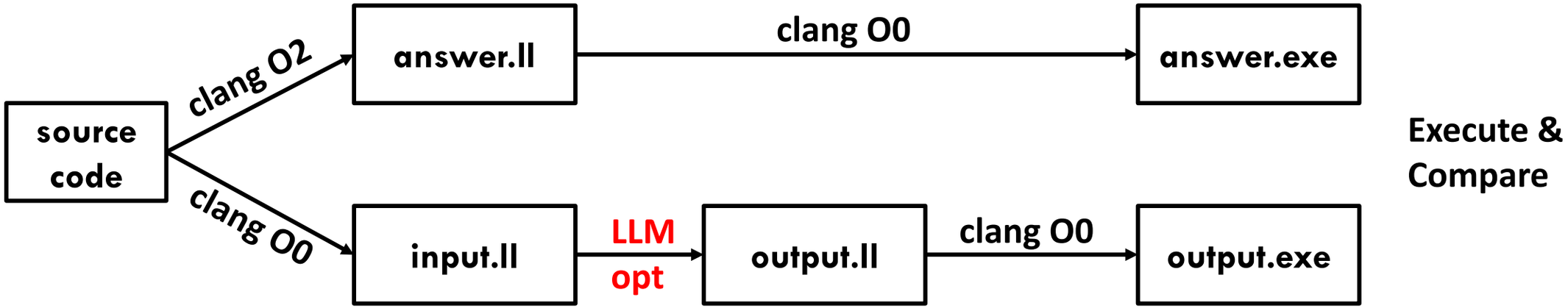
```
//  循环展开前
while(j<60)
{
  ans = ans + 20;
}

// 循环展开后
ans = ans + 20;
ans = ans + 20;
...
ans = ans + 20;

// 结合后续优化
ans = ans + 20 * 60;
```

OUTLINE

目　录

中山大学计算机学院
School of Computer Science & Engineering

- 实验目标
  - 在实验四中添加了LLM for Compiler部分，这部分将会占据实验总分的5分，不会给同学们带来很大压力
  - 评分方式与实验四相同，与clang O2进行对比，但必须使用LLM
  - 在实验四传统方法的基础上，进一步添加大模型辅助编译
  - 允许同学们基于现有框架自由探索



- LLM for Compiler： Pass序列预测
  - 可以使用LLM来动态调整Pass顺序
  - Pass顺序对于程序的性能有着显著的影响

```
// 添加 LLM 加持的 Pass 到优化管理器中
mpm.addPass(PassSequencePredict(
  "<api_key>",
  "<base_url>",
  {
    { "StaticCallCounterPrinter",
      TASK4_DIR "/StaticCallCounterPrinter.hpp",
      TASK4_DIR "/StaticCallCounterPrinter.cpp",
      "StaticCallCounterPrinter.xml",
      [](llvm::ModulePassManager& mpm) {
        mpm.addPass(StaticCallCounterPrinter(llvm::errs()));
      } },
    { "Mem2Reg",
      TASK4_DIR "/Mem2Reg.hpp",
      TASK4_DIR "/Mem2Reg.cpp",
      "Mem2Reg.xml",
      [](llvm::ModulePassManager& mpm) { mpm.addPass(Mem2Reg()); } },
    { "ConstantFolding",
      TASK4_DIR "/ConstantFolding.hpp",
      TASK4_DIR "/ConstantFolding.cpp",
      "ConstantFolding.xml",
      [](llvm::ModulePassManager& mpm) {
        mpm.addPass(ConstantFolding(llvm::errs()));
      } },
  }));
```

Large Language Models for Compiler Optimization

Chris Cummins[†*], Volker Seeker[†], Dejan Grubisic[†],
Mostafa Elhoushi, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle,
Kim Hazelwood, Gabriel Synnaeve, Hugh Leather[†]
Meta AI

Youwei Liang
UC San Diego

- 如何将LLM与编译器结合（以Pass序列预测为例）
  - PassSequencePredict()的两步工作流
    - 调用 LLM总结传入的每个Pass
      - 发送 Pass 名字、头文件和实现文件内容
      - 获得 Pass 名字、功能和对 IR 所起的效果
      - 将分析结果缓存，避免重复分析
      - 实现在 PassSequencePredict::pass_summary() 函数中
    - 根据Pass总结与待优化的IR，调用LLM输出最优的Pass序列
      - LLM 按照指定格式返回 Pass 序列（提示词限制格式）
      - 遍历列表，根据 Pass 名字，使用 mpm.addPass() 添加 Pass
      - 使用 mpm.run() 应用 Pass
      - 实现在 PassSequencePredict::run() 函数中
    - 文档中有更为详细的工作流解释

- 将LLM与编译器结合(以Pass序列预测为例)

  第一步：修改main.cpp

  - 填入API key与Base URL
  - 填入被预测顺序的Pass信息
  - 右图共有3个Pass被预测并参与执行

```
// 添加 LLM 加持的 Pass 到优化管理器中
mpm.addPass(PassSequencePredict(
  "<api_key>",
  "<base_url>",
  {
    { "StaticCallCounterPrinter",
      TASK4_DIR "/StaticCallCounterPrinter.hpp",
      TASK4_DIR "/StaticCallCounterPrinter.cpp",
      "StaticCallCounterPrinter.xml",
      [](llvm::ModulePassManager& mpm) {
        mpm.addPass(StaticCallCounterPrinter(llvm::errs()));
      } },
    { "Mem2Reg",
      TASK4_DIR "/Mem2Reg.hpp",
      TASK4_DIR "/Mem2Reg.cpp",
      "Mem2Reg.xml",
      [](llvm::ModulePassManager& mpm) { mpm.addPass(Mem2Reg()); } },
    { "ConstantFolding",
      TASK4_DIR "/ConstantFolding.hpp",
      TASK4_DIR "/ConstantFolding.cpp",
      "ConstantFolding.xml",
      [](llvm::ModulePassManager& mpm) {
        mpm.addPass(ConstantFolding(llvm::errs()));
      } },
  }));
```

感兴趣的同学可阅读并修改PassSequencePredict()类的内容

- 将LLM与编译器结合(以Pass序列预测为例)

  第二步：修改llm/prompts/*.xml

  - PassSummary*.xml：总结每个Pass的功能，用于Pass序列预测
  - PassSeqPred*.xml：预测Pass顺序
  - 修改xml内容，帮助LLM输出更准确的Pass总结和更高效的Pass序列（xml内容的修改没有任何约束）

```
<task>
  你是一位熟悉 LLVM 和 C++17 的编译优化专家，精通 LLVM IR、LLVM Pass 和相关优化技术。
  <instructions>
    1. 分析用户提供的 LLVM Pass 类名，以及对应的头文件和实现文件；
    2. 总结这个 Pass 的功能，实现了什么优化，它如何影响 LLVM IR；
    3. 提供有价值的见解，以帮助他人更好地理解代码；
    4. 遵循以下 xml 格式进行输出：
    <pass>
      <name>
        {{ LLVM Pass 类名，即用户指定需要分析的 LLVM Pass }}
      </name>
      <description>
        {{ 优化方法概述 }}
      </description>
      <effect>
        {{ 对 LLVM IR 起到的效果 }}
      </effect>
    </pass>
    5. 不要输出任何额外的信息，务必按照上述要求进行回复。
  </instructions>
</task>
```

注：

- 双花括号{{}}表示内容待LLM填充
- 单花括号{}用于Python格式化输出

- **完成实验四大语言模型部分的提示**
  - **Pass序列预测的修改方向：**
    - 改变 PassSequencePredict() 的工作流逻辑
    - 撰写更高质量的提示词
  - **在传统方法中，推荐优先实现并传入：**
    - 强度削弱（StrengthReduction）
    - 公共子表达式消除（Common Subexpression Elimination，CSE）
    - 循环无关变量移动（Loop-invariant code motion，LICM）
    - 推荐将以上优化加入到 PassSequencePredict 中
  - **同学们可尽情尝试其他将 LLM 融入编译优化的方法**
    - IR 变换，优化参数预测，辅助决策……

OUTLINE

目　录

中山大学计算机学院
School of Computer Science & Engineering

- LLM for Compiler：超参数预测
  - 可以使用LLM来预测优化超参数或者辅助优化决策
  - 函数是否内联、循环展开因子
  - 因为当前优化较少，所以性能可能并不明显
  - 集成在Pass中为其他Pass服务

- LLM for Compiler：IR代码生成
  - 可以使用LLM来生成代码
  - 大模型天然适合做代码生成工作
  - 迭代生成代码（self-refine）
  - 如何保证生成代码的正确性呢
  - 单元测试与形式化验证（Alive2）

LLM-Vectorizer: LLM-Based Verified Loop Vectorizer

VecTrans: LLM Transformation Framework for Better Auto-vectorization on High-performance CPU

- LLM for Compiler：do whatever you want
  - 可以使用LLM来做你想做的一切优化，鼓励同学们积极探索
  - 寻找合适的切入点，实现能带来性能提升的优化
  - 可能后续可以发展为科研课题

如何在编译器中调用 LLM

- python3-openai → llm/__init__.py → pybind11 → llm/LLMHelper → task4
- 主要通过 LLMHelper 来进行与 LLM 的交互
  - 构造函数：指定 LLM，添加凭证
  - create_new_session：新建一场对话
  - delete_session：删除一场历史对话
  - add_content：以不同角色在对话中发送消息
  - chat：发送一场对话
    - model：指定模型名字，如 deepseek-r1
    - handlers：对 回复进行一系列处理
      - 元素为处理并返回字符串的函数
      - 去除 deepseek-r1 思维链（</think>）
    - params：请求参数，temperature、max_tokens
  - 创建对话 → 添加内容 → 发送对话 → 解析回复 → 删除对话

```
class LLMHelper
{
public:
  enum class Role : std::uint8_t
  {
    kUser,
    kSystem,
    kAssistant,
  };

  LLMHelper(llvm::StringRef apiKey, llvm::StringRef baseURL);

  std::string create_new_session();

  void delete_session(llvm::StringRef sessionID);

  void add_content(llvm::StringRef sessionID,
                   Role role,
                   llvm::StringRef content);

  std::string chat(llvm::StringRef sessionID,
                   llvm::StringRef model,
                   const pybind11::list& handlers,
                   const pybind11::dict& params);
};
```

- 如何在编译器中调用 LLM
  - 创建 LLMHelper 实例
  - 创建对话
  - 添加消息（提示词）
  - 发送消息
    - 指定如何处理回复
    - 指定请求参数
  - 获得回复
  - 删除对话

```cpp
// 初始化 LLMHelper，传入 api_key 和 base_url
auto helper = LLMHelper("<api_key>", "<base_url>");

// 新建一场对话
std::string sessionID helper.create_new_session();

// 加入消息（提示词）
// 系统级提示词
std::string systemPrompt = "你是一位得力助手，能很好的解决用户的问题";
helper.add_content(sessionID, LLMHelper::Role::kSystem, systemPrompt);
// 用户级提示词
std::string userPrompot = "你好！";
helper.add_content(sessionID, LLMHelper::Role::kUser, userPrompot);

// 创建 handlers
auto handlers = pybind11::list();
// 加入函数，去除 deepseek-r1 的思维链
// 导入 `llm/__init__.py`，去除思维链的函数在此文件中
pybind11::module_ llm = pybind11::module_::import("llm");
// 获得函数
auto remove_deepseek_r1_think = llm.attr("remove_deepseek_r1_think");
// 加入到 handlers 中
handlers.attr("append")(remove_deepseek_r1_think);

// 其他请求参数，使用用户定义字面量语法
auto params = pybind11::dict("max_tokens"_a = 8192, "stream"_a = false, "temperature"_a = 0);

// 发送对话，获得回复
std::string response = helper.chat(sessionID, "deepseek-reasoner", handlers, params);

// 删除对话
helper.delete_session(sessionID);
```

- _a：字面量语法，创建命名参数
  - 左边为参数名字
  - 右边为值
- 等价于 params[“max_tokens”] = 8192
- _a 还可用于函数的关键字传参等任何需要给参数（值）指定名字的地方

- 如何在编译器中调用 LLM
  - pybind11，用于在 C++ 和 Python 之间创建双向绑定
    - 将 C++ 文件编译成 Python 可 import 使用的格式
    - 在 C++ 中导入 Python 模块，使用 Python 解释器和基本语言特性
      - 必须先初始化 Python 解释器，仅可在解释器作用域内使用

```
// 初始化 Python 解释器，使得可以使用 Python 语言特性
// 仅能在 guard 所在的作用域内使用 Python 语言特性
pybind11::scoped_interpreter guard {};
// 导入 sys 包，添加实验四的路径到 sys.path 中，方便 import llm
pybind11::module_ sys = pybind11::module_::import("sys");
sys.attr("path").attr("append")("<task4_dir>");
```

      - 通过 .attr()，以取出对象的成员变量和方法
      - pybind11 中类型实例可进行动态类型转换 .cast<T>
      - 可在 llm/__init__.py 中编写 Python 脚本，在 C++ 中通过 sys 添加包搜索路径后，使用 import(“llm”) 和 .attr() 来使用
  - **pybind11 的引入是为了方便调用 LLM 以及简化复杂字符串处理，而非在 C++ 中使用 Python，此库并非实验四重点，无需过度深究！**

谢 谢!

yatcc-ai.com