目 录

[1. 概述 1](#_Toc558372170)

[1.1 术语表 1](#_Toc2078982007)

[2. 使用 CMake 构建 LLVM 2](#_Toc1234081895)

[2.1 简介 2](#_Toc59218524)

[2.2 快速入门 2](#_Toc1350339604)

[2.3 CMake 基本用法 3](#_Toc2041855296)

[2.4 选项和变量 4](#_Toc322411198)

[2.5 常用的 CMake 变量 5](#_Toc1995373960)

[2.6 常用的 LLVM 相关变量 6](#_Toc260596573)

[2.7 很少使用的 CMake 变量 6](#_Toc752459470)

[2.8 LLVM 相关变量 7](#_Toc1694346121)

[2.9 高级变量 18](#_Toc290233137)

[2.10 CMake 缓存 18](#_Toc1213396968)

[2.11 执行测试 19](#_Toc159677438)

[2.12 交叉编译 19](#_Toc1776565599)

[2.13 在你的项目中嵌入 LLVM  19](#_Toc1446163908)

[2.14 Developing LLVM passes out of source 22](#_Toc1567714131)

[3. 编写 LLVM Pass 23](#_Toc1402634167)

[3.1 简介——什么是Pass？ 23](#_Toc888948240)

[3.2 快速入门——编写 hello world  23](#_Toc1915582584)

[3.2.1 设置构建环境 24](#_Toc1524779816)

[3.2.2 所需的基本代码 24](#_Toc628368795)

[3.2.3 使用opt来运行一个pass 26](#_Toc778681949)

[3.3 pass 相关类和要求 28](#_Toc1506559540)

[3.3.1 ImmutablePass 类 28](#_Toc1856942336)

[3.3.2 ModulePass 类 29](#_Toc2133908907)

[3.3.2.1 runOnModule 方法 29](#_Toc394417723)

[3.3.3 CallGraphSCCPass 类 29](#_Toc601469046)

[3.3.3.1 doInitialization(CallGraph &)方法 30](#_Toc1849190187)

[3.3.3.2 runOnSCC方法 30](#_Toc1755866931)

[3.3.3.3 doFinalization(CallGraph &) 方法 30](#_Toc1579334658)

[3.3.4 FunctionPass 类 30](#_Toc260078710)

[3.3.4.1 doInitialization(Module &)方法 31](#_Toc1687365290)

[3.3.4.2 runOnFunction方法 31](#_Toc665932905)

[3.3.4.3 doFinalization(Module &)方法 31](#_Toc319297234)

[3.3.5 LoopPass 类 31](#_Toc890221246)

[3.3.5.1 doInitialization(Loop \*, LPPassManager &)方法 32](#_Toc560304554)

[3.3.5.2 runOnLoop方法 32](#_Toc641708432)

[3.3.5.3 doFinalization()方法 32](#_Toc738111558)

[3.3.6 RegionPass 类 32](#_Toc820901127)

[3.3.6.1 doInitialization(Region \*, RGPassManager &)方法 33](#_Toc1394167902)

[3.3.6.2 runOnRegion方法 33](#_Toc284974031)

[3.3.6.3 doFinalization()方法 33](#_Toc1111134264)

[3.3.7 MachineFunctionPass 类 33](#_Toc460081223)

[3.3.7.1 runOnMachineFunction(MachineFunction &MF)方法 34](#_Toc444651469)

[3.3.8 Pass registration (pass 注册) 34](#_Toc740216215)

[3.3.8.1 print方法 34](#_Toc1906245131)

[3.3.9 指定 passes 之间的交互 35](#_Toc2012365601)

[3.3.9.1 getAnalysisUsage方法 35](#_Toc2142850383)

[3.3.9.2 AnalysisUsage::addRequired<>和AnalysisUsage::addRequiredTransitive<>方法 35](#_Toc647709723)

[3.3.9.3 AnalysisUsage::addPreserved<>方法 36](#_Toc1780464537)

[3.3.9.4 getAnalysis<>和getAnalysisIfAvailable<>方法 36](#_Toc1520146551)

[3.3.10 实现分析组(Analysis Groups) 37](#_Toc1276078519)

[3.3.10.1 分析组概念 37](#_Toc411662838)

[3.3.10.2 使用RegisterAnalysisGroup 38](#_Toc879222443)

[3.4 pass统计 39](#_Toc985537207)

[3.4.1 PassManager 做什么 39](#_Toc398088097)

[3.4.1.1 releaseMemory方法 42](#_Toc1273640166)

[3.5 注册动态加载的passes 42](#_Toc1587006253)

[3.5.1 使用现有的注册表 43](#_Toc99794636)

[3.5.2 创建新的注册表 44](#_Toc882023449)

[3.5.3 将 GDB 与动态加载的pass一起使用 44](#_Toc1018857263)

[3.5.3.1 在你的传递中设置一个断点 44](#_Toc359873346)

[3.5.3.2 杂项问题 45](#_Toc421905092)

[3.5.4 计划未来的扩展 46](#_Toc1684790169)

[3.5.4.1 多线程 LLVM  46](#_Toc679170581)

[4. LLVM 的分析和转换过程 46](#_Toc1312126338)

[4.1 简介 46](#_Toc97611075)

[4.2 分析passes(Analysis Passes) 46](#_Toc1320879013)

[4.3 Transform Passes 50](#_Toc2050237897)

[4.4 Utility Passes 58](#_Toc918512202)

[5. 堆栈安全分析 59](#_Toc567563268)

[5.1 简介 59](#_Toc187728280)

[5.2 它是如何工作 59](#_Toc2029646467)

[5.3 测试 60](#_Toc1027644491)

[6. MergeFunctions pass，它是如何工作的 60](#_Toc632379750)

[6.1 简介 60](#_Toc622379034)

[6.1.1 我应该知道什么才能跟上这份文件？ 60](#_Toc786405974)

[6.1.2 本文结构 61](#_Toc497261703)

[6.2 基础知识 61](#_Toc617745769)

[6.2.1 可能的解决方案 61](#_Toc1434115698)

[6.2.1.1 随机存取 61](#_Toc130242592)

[6.2.1.2 对数搜索 62](#_Toc2137892321)

[6.2.1.3 当前状态 62](#_Toc562710569)

[6.2.2 MergeFunctions、主要字段和 runOnModule  62](#_Toc541905430)

[6.2.2.1 runOnModule 62](#_Toc869631116)

[6.2.2.2 比较和对数搜索 63](#_Toc1548247776)

[6.3 函数比较 63](#_Toc939993527)

[6.3.1 FunctionComparator::compare(void) 63](#_Toc2143271283)

[6.3.2 FunctionComparator::cmpType  64](#_Toc987770382)

[6.3.3 cmpValues(const Value\*, const Value\*)  65](#_Toc1039788163)

[6.3.3.1 我们在 cmpValues 中关联什么？ 65](#_Toc877811084)

[6.3.3.2 如何实现cmpValues？ 66](#_Toc2006627645)

[6.3.4 cmpConstants 67](#_Toc1399661510)

[6.3.5 compare(const BasicBlock\*, const BasicBlock\*) 67](#_Toc1299716176)

[6.3.6 cmpGEP  68](#_Toc1543934166)

[6.3.7 cmpOperation 68](#_Toc2078832091)

[6.3.8 O(log(N))  68](#_Toc464358867)

[6.4 合并过程，mergeTwoFunctions  68](#_Toc1641545241)

[6.4.1 HasGlobalAliases, removeUsers 69](#_Toc1252227456)

[6.4.2 没有全局别名，replaceDirectCallers  70](#_Toc367113116)

[6.4.2.1 If “F” could not be overridden, fix it！ 70](#_Toc412573796)

[7. LLVM 别名分析基础架构 70](#_Toc1819790724)

[7.1 简介 70](#_Toc554841396)

[7.2 AliasAnalysis类概述 71](#_Toc294736615)

[7.2.1 指针的表示 71](#_Toc699951567)

[7.2.2 alias方法 72](#_Toc1187221146)

[7.2.2.1 Must, May, and No Alias Responses 72](#_Toc917115649)

[7.2.3 getModRefInfo方法 72](#_Toc1486357542)

[7.2.4 其他有用的AliasAnalysis方法 72](#_Toc1684482849)

[7.2.4.1 getModRefInfoMask方法 72](#_Toc1534861419)

[7.2.4.2 doesNotAccessMemory和 onlyReadsMemory方法 73](#_Toc772989592)

[7.3 编写一个新的AliasAnalysis实现 73](#_Toc1814725441)

[7.3.1 不同的 Pass 样式 73](#_Toc1525270092)

[7.3.2 所需的初始化调用 73](#_Toc1335700161)

[7.3.3 override所需的方法 74](#_Toc209147223)

[7.3.4 可以指定的接口 74](#_Toc247417560)

[7.3.5 AliasAnalysis链接行为 74](#_Toc736464289)

[7.3.6 更新转换分析结果 74](#_Toc1149140750)

[7.3.6.1 deleteValue方法 75](#_Toc243205195)

[7.3.6.2 copyValue方法 75](#_Toc1724234671)

[7.3.6.3 replaceWithNewValue方法 75](#_Toc41445266)

[7.3.6.4 addEscapingUse方法 75](#_Toc1121016280)

[7.3.7 效率问题 75](#_Toc1583378669)

[7.3.8 限制 75](#_Toc1441106776)

[7.4 使用别名分析结果 76](#_Toc273248808)

[7.4.1 使用MemoryDependenceAnalysis pass 76](#_Toc979829187)

[7.4.2 使用AliasSetTracker类 76](#_Toc1372455219)

[7.4.2.1 AliasSetTracker 实现 77](#_Toc737607675)

[7.4.3 直接使用AliasAnalysis接口 77](#_Toc473890781)

[7.5 现有的别名分析实现和client 端 77](#_Toc477199027)

[7.5.1 可用的AliasAnalysis实现 77](#_Toc1104720791)

[7.5.1.1 -basic-aa pass 77](#_Toc886464577)

[7.5.1.2 -globalsmodref-aa pass 77](#_Toc149506104)

[7.5.1.3 -steens-aa pass 78](#_Toc1659562188)

[7.5.1.4 -ds-aa pass 78](#_Toc1181201192)

[7.5.1.5 -scev-aa pass 78](#_Toc849457671)

[7.5.2 别名分析驱动的转换 78](#_Toc699299686)

[7.5.2.1 -adce pass 78](#_Toc2098316841)

[7.5.2.2 -licm pass 78](#_Toc188331565)

[7.5.2.3 -argpromotion pass 79](#_Toc236298888)

[7.5.2.4  -gvn,-memcpyopt和-dse passes 79](#_Toc1485694612)

[7.5.3 用于调试和评估实现的client 端 79](#_Toc961321157)

[7.5.3.1 -aa-eval pass 79](#_Toc2051024329)

[7.6 内存依赖分析 79](#_Toc863481056)

[8. memory SSA 79](#_Toc149537670)

[8.1 简介 79](#_Toc112687905)

[8.2 内存SSA结构 80](#_Toc1110898617)

[8.3 MemorySSA 的设计 82](#_Toc886001960)

[8.3.1 walker 82](#_Toc1261828655)

[8.3.1.1 默认 walker API  82](#_Toc1354103812)

[8.3.1.2 自己定位 clobbers  83](#_Toc462752983)

[8.3.2 使用和定义优化(Use and Def optimization) 83](#_Toc1303273921)

[8.3.3 失效和更新 84](#_Toc327636444)

[8.3.3.1 phi 放置(Phi placement) 84](#_Toc2046131652)

[8.3.4 非目标(Non-Goals) 85](#_Toc596897049)

[8.3.5 设计权衡(Design tradeoffs) 85](#_Toc600885253)

[8.3.5.1 精度 85](#_Toc878477192)

[8.3.5.2 实践中的精确度 86](#_Toc1969352268)

[8.4 LLVM 开发者会议演示 86](#_Toc1338492928)

[9. LLVM 循环术语（和规范形式） 86](#_Toc1352367973)

[9.1 循环定义 86](#_Toc299067648)

[9.1.1 术语 86](#_Toc295730072)

[9.1.2 重要说明 87](#_Toc91348902)

[9.2 循环信息LoopInfo 91](#_Toc448573752)

[9.3 循环简化形式Loop Simplify Form 92](#_Toc1955292260)

[9.4 闭环 SSA (LCSSA) Loop Closed SSA 92](#_Toc1272550094)

[9.5 “更规范的”循环“More Canonical” Loops 94](#_Toc1298031423)

[9.5.1 旋转循环Rotated Loops 94](#_Toc507108298)

[10. LLVM 中的自动矢量化 99](#_Toc1223383287)

[10.1 Loop Vectorizer 99](#_Toc1486362989)

[10.1.1 用法 99](#_Toc743407186)

[10.1.1.1 命令行标志 99](#_Toc561594252)

[10.1.1.2 Pragma 循环提示指令 100](#_Toc300200498)

[10.1.2 诊断 100](#_Toc646947868)

[10.1.3 特点 101](#_Toc1425075308)

[10.1.3.1 行程次数(tripcount)未知的循环 101](#_Toc449738169)

[10.1.3.2 指针的运行时检查 102](#_Toc759635773)

[10.1.3.3 减少Reductions 102](#_Toc388490277)

[10.1.3.4 归纳Inductions 102](#_Toc1335740129)

[10.1.3.5 如果转换 103](#_Toc2021464428)

[10.1.3.6 指针归纳变量 103](#_Toc1742594090)

[10.1.3.7 反向迭代器 103](#_Toc1798493112)

[10.1.3.8 分散/聚集Scatter / Gather 104](#_Toc1177254702)

[10.1.3.9 混合类型的向量化 104](#_Toc2070230534)

[10.1.3.10 全局结构别名分析 104](#_Toc1697141117)

[10.1.3.11 函数调用的向量化 105](#_Toc1774151751)

[10.1.3.12 矢量化过程中的部分展开 105](#_Toc523632139)

[10.1.3.13 结语矢量化Epilogue Vectorization 106](#_Toc428134661)

[10.1.4 性能 107](#_Toc1596020372)

[10.1.5 持续发展方向 108](#_Toc1862125068)

[10.2 SLP 向量化 108](#_Toc1780502634)

[10.2.1 详情 108](#_Toc1895088020)

[10.2.2 用法 109](#_Toc10371492)

[11. LLVM 链接时间优化：设计与实现 109](#_Toc1871851536)

[11.1 描述 109](#_Toc196178124)

[11.2 设计理念 109](#_Toc1965663752)

[11.2.1 链接时间优化示例 109](#_Toc996917982)

[11.2.2 替代方法 110](#_Toc1494209547)

[11.3 libLTO和链接器之间的多阶段通信 111](#_Toc325288402)

[11.3.1 阶段 1：读取 LLVM 位码文件 111](#_Toc72817621)

[11.3.2 第 2 阶段：符号解析 111](#_Toc833088888)

[11.3.3 第 3 阶段：优化 Bitcode 文件 111](#_Toc1068695589)

[11.3.4 阶段 4：优化后的符号解析 112](#_Toc634411873)

[11.4 libLTO 112](#_Toc1133289387)

[11.4.1 lto\_module\_t 112](#_Toc1715643457)

[11.4.2 lto\_code\_gen\_t 112](#_Toc2059487182)

[12. LLVM 黄金插件 113](#_Toc1583027556)

[12.1 简介 113](#_Toc327795582)

[12.2 如何构建它 113](#_Toc300493811)

[12.3 用法 114](#_Toc771284037)

[12.4 链接时间优化示例 114](#_Toc201776362)

[12.5 在自动工具项目中使用 LTO 的快速入门 115](#_Toc2043087901)

[12.6 许可 115](#_Toc422293501)

[13. 使用 LLVM 进行源代码级调试 115](#_Toc1379031064)

[13.1 简介 115](#_Toc1965834788)

[13.1.1 LLVM 调试信息背后的哲学 115](#_Toc2119434618)

[13.1.2 调试信息消费者Debug information consumers 116](#_Toc1005699168)

[13.2 调试信息和优化 116](#_Toc341983279)

[13.3 调试信息格式 117](#_Toc400085631)

[13.3.1 调试器内部函数 117](#_Toc454235892)

[13.3.1.1 llvm.dbg.addr 117](#_Toc56624699)

[13.3.1.2 llvm.dbg.declare 118](#_Toc33104617)

[13.3.1.3 llvm.dbg.value 118](#_Toc201840264)

[13.3.1.4 llvm.dbg.assign 118](#_Toc66996191)

[13.4 对象生命周期和范围 119](#_Toc1904956153)

[13.5 优化代码中的对象生命周期 120](#_Toc398018388)

[13.6 变量位置元数据在 CodeGen 期间如何转换 123](#_Toc2032659943)

[13.6.1 指令选择和 MIR 中的变量位置 124](#_Toc754390487)

[13.6.2 指令调度 127](#_Toc1892227935)

[13.6.3 寄存器分配期间的变量位置 128](#_Toc210464698)

[13.6.4 变量位置的 LiveDebugValues 扩展 128](#_Toc827208109)

[13.7 C/C++ 前端特定调试信息 130](#_Toc577833176)

[13.7.1 C/C++ 源文件信息 130](#_Toc1279160287)

[13.7.2 C/C++ 全局变量信息 131](#_Toc1461619982)

[13.7.3 C/C++ 函数信息 132](#_Toc1711122563)

[13.8 C++ 特定的调试信息 133](#_Toc847320096)

[13.8.1 C++ 特殊成员函数信息 133](#_Toc1373623516)

[14. 调试信息的指令参考 134](#_Toc1146666471)

[14.1 问题陈述 134](#_Toc1175115678)

[14.2 解决方案：指令引用 134](#_Toc1674117328)

[14.3 寄存器分配器注意事项 134](#_Toc1917950508)

[14.4 LiveDebugValues 135](#_Toc1376892040)

[14.5 所需的目标支持和转换指南 135](#_Toc1569721581)

[14.5.1 目标钩子 135](#_Toc192760361)

[14.5.2 针对特定目标的优化工具 135](#_Toc608439457)

[15. 编写 LLVM 后端---待学习 136](#_Toc1388072721)

[15.1 简介 136](#_Toc164711332)

[15.1.1 先决条件阅读 136](#_Toc1614138625)

[15.1.2 基本步骤 137](#_Toc1730056001)

[15.1.3 预备 138](#_Toc564796963)

[15.2 目标机器 138](#_Toc2068374517)

[15.3 目标注册 140](#_Toc1786680700)

[15.4 寄存器集和寄存器类 140](#_Toc597901581)

[15.4.1 定义一个寄存器 141](#_Toc122731133)

[15.4.2 定义寄存器类 142](#_Toc1853676892)

[15.4.3 实现一个TargetRegisterInfo的子类 145](#_Toc355374086)

[15.5 指令集 145](#_Toc520749521)

[15.5.1 指令操作数映射 147](#_Toc1738853187)

[15.5.1.1 指令操作数名称映射 149](#_Toc1109764574)

[15.5.1.2 指令操作数类型 149](#_Toc265493808)

[15.5.2 指令调度 150](#_Toc1949317885)

[15.5.3 指令关系映射 150](#_Toc1936972683)

[15.5.4 实现一个TargetInstrInfo的子类 150](#_Toc843326984)

[15.5.5 分支折叠和 If 转换 151](#_Toc1080994524)

[15.6 指令选择器 152](#_Toc1251109017)

[15.6.1 SelectionDAG 合法化阶段 154](#_Toc406965899)

[15.6.1.1 推广 154](#_Toc1928314620)

[15.6.1.2 展开 155](#_Toc477248886)

[15.6.1.3 自定义 155](#_Toc1553632370)

[15.6.1.4 Legal  155](#_Toc955946650)

[15.6.2 调用约定 156](#_Toc3882566)

[15.7 汇编printer 157](#_Toc1324099230)

[15.8 子目标支持 158](#_Toc185355043)

[15.9 JIT Support(即时支持) 160](#_Toc1573604147)

[15.9.1 机器代码发射器 160](#_Toc1516859592)

[15.9.1.1 目标 JIT 信息 162](#_Toc793794500)

[16. LLVM 目标独立代码生成器 --- 待学习 162](#_Toc814193221)

[16.1 简介 162](#_Toc1681570924)

[16.1.1 代码生成器中的必需组件 163](#_Toc260449477)

[16.1.2 代码生成器的高层设计 163](#_Toc396765574)

[16.1.3 使用 TableGen 进行目标描述 164](#_Toc98884239)

[16.2 目标描述类 164](#_Toc181340346)

[16.2.1 TargetMachine 类 164](#_Toc35962626)

[16.2.2 DataLayout class 165](#_Toc696785820)

[16.2.3 TargetLowering class 165](#_Toc304071479)

[16.2.4 TargetRegisterInfo class 165](#_Toc1889639518)

[16.2.5 TargetInstrInfo class 165](#_Toc1052159907)

[16.2.6 TargetFrameLowering class 165](#_Toc824821000)

[16.2.7 TargetSubtarget class 166](#_Toc1481009058)

[16.3 机器码描述类 166](#_Toc14440833)

[16.3.1 MachineInstr class 166](#_Toc1090314808)

[16.3.1.1 使用MachineInstrBuilder.h的函数 166](#_Toc1282843295)

[16.3.1.2 固定（预分配）寄存器 167](#_Toc1951413516)

[16.3.1.3 Call-clobbered registers 168](#_Toc1933641793)

[16.3.1.4 SSA 形式的机器码 168](#_Toc216354172)

[16.3.2 MachineBasicBlock class 168](#_Toc1055038885)

[16.3.3 MachineFunction class 168](#_Toc193124044)

[16.3.4 MachineInstr Bundles 168](#_Toc2144668792)

[16.4 “MC”层 169](#_Toc1532287771)

[16.4.1 MCStreamerAPI  170](#_Toc1746756415)

[16.4.2 MCContext class 170](#_Toc953131795)

[16.4.3 MCSymbol class 170](#_Toc1536170337)

[16.4.4 MCSection class 170](#_Toc923371997)

[16.4.5 MCInst class 171](#_Toc1138486838)

[16.4.6 目标文件格式 171](#_Toc962290837)

[16.5 与目标无关的代码生成算法 171](#_Toc292747941)

[16.5.1 指令选择 171](#_Toc1932281338)

[16.5.1.1 SelectionDAG 简介 172](#_Toc1776484058)

[16.5.1.2 SelectionDAG指令选择过程 172](#_Toc1974318865)

[16.5.1.3 构建初始 DAG 173](#_Toc45247167)

[16.5.1.4 SelectionDAG 合法化类型阶段 173](#_Toc25765984)

[16.5.1.5 SelectionDAG 合法化阶段 174](#_Toc2073203105)

[16.5.1.6 SelectionDAG 优化阶段：DAG 组合器 174](#_Toc226587513)

[16.5.1.7 SelectionDAG 选择阶段 175](#_Toc61728610)

[16.5.1.8 SelectionDAG Scheduling and Formation Phase 177](#_Toc622505277)

[16.5.1.9 SelectionDAG 的未来方向 177](#_Toc530658992)

[16.5.2 基于 SSA 的机器代码优化 177](#_Toc1951368129)

[16.5.3 Live Intervals 177](#_Toc1674665184)

[16.5.3.1 Live Variable Analysis实时变量分析 177](#_Toc1355479992)

[16.5.3.2 Live Intervals Analysis实时区间分析 178](#_Toc1284893539)

[16.5.4 注册分配 178](#_Toc1689106017)

[16.5.4.1 寄存器在 LLVM 中是如何表示的 178](#_Toc298311152)

[16.5.4.2 将虚拟寄存器映射到物理寄存器 179](#_Toc420253186)

[16.5.4.3 处理两条地址指令 180](#_Toc1493035885)

[16.5.4.4 SSA析构阶段 180](#_Toc84469297)

[16.5.4.5 指令折叠 180](#_Toc636607358)

[16.5.4.6 内置寄存器分配器 181](#_Toc400591123)

[16.5.5 Prolog/Epilog 代码插入 181](#_Toc277593342)

[16.5.6 Compact Unwind压缩展开 181](#_Toc633792503)

[16.5.7 后期机器代码优化 182](#_Toc1932878894)

[16.5.8 代码发射 182](#_Toc2024349757)

[16.5.8.1 发射函数堆栈大小信息 183](#_Toc1586924298)

[16.5.9 VLIW Packetizer 183](#_Toc1321565584)

[16.5.9.1 Mapping from instructions to functional units从指令到功能单元的映射 183](#_Toc800238106)

[16.5.9.2 如何生成和使用打包表How the packetization tables are generated and used 184](#_Toc577927488)

[16.6 Implementing a Native Assembler实现本机汇编器 184](#_Toc136372773)

[16.6.1 指令解析 184](#_Toc1092986048)

[16.6.2 指令别名处理 184](#_Toc362725178)

[16.6.2.1 Mnemonic Aliases助记符别名 184](#_Toc1912856831)

[16.6.2.2 指令别名 185](#_Toc919821265)

[16.6.3 指令匹配 185](#_Toc407972345)

[16.7 针对特定目标的实现说明 185](#_Toc1938622815)

[16.7.1 Tail call optimization尾调用优化 186](#_Toc845540722)

[16.7.2 Sibling call optimization兄弟调用优化 186](#_Toc634559858)

[16.7.3 X86 后端 187](#_Toc2000351425)

[16.7.3.1 支持 X86 目标三元组 187](#_Toc1468046000)

[16.7.3.2 支持 X86 调用约定 187](#_Toc1165218850)

[16.7.3.3 在 MachineInstrs 中表示 X86 寻址模式 187](#_Toc1804235906)

[16.7.3.4 支持 X86 地址空间 188](#_Toc995227536)

[16.7.3.5 指令命名 188](#_Toc373215194)

[16.7.4 AMDGPU 后端 188](#_Toc941645797)

[17. TableGen 188](#_Toc536849906)

[17.1 简介 188](#_Toc671526346)

[17.2 TableGen 程序 189](#_Toc1361898984)

[17.2.1 运行TableGen 189](#_Toc2029885791)

[17.2.2 例子 190](#_Toc755995644)

[17.3 语法 191](#_Toc1998506342)

[17.3.1 基本概念 191](#_Toc282993266)

[17.4 TableGen 后端 192](#_Toc1033588986)

[17.5 TableGen 缺陷 192](#_Toc484815197)

[18. MCJIT 设计与实现 192](#_Toc68388513)

[18.1 简介 192](#_Toc910455095)

[18.2 引擎创建 193](#_Toc2071739495)

[18.3 代码生成 193](#_Toc1389954097)

[18.4 对象加载 194](#_Toc1710693201)

[18.5 地址重映射 196](#_Toc502183335)

[18.6 最后的准备 196](#_Toc1526326870)

[19. ORC 设计与实现 197](#_Toc656195601)

[19.1 简介 197](#_Toc864908513)

[19.2 用例 197](#_Toc1291700053)

[19.3 特点 197](#_Toc1576016867)

[19.4 LLJIT 和 LLLazyJIT  198](#_Toc1272880858)

[19.5 设计概述 199](#_Toc1082839220)

[19.6 Top Level APIs 200](#_Toc274073941)

[19.7 Absolute Symbols, Aliases, and Reexports 201](#_Toc1907440716)

[19.7.1 绝对符号 201](#_Toc935706997)

[19.7.2 Aliases and Reexports 201](#_Toc1742119941)

[19.8 Laziness懒惰 202](#_Toc925175918)

[19.9 支持自定义编译器 202](#_Toc592459256)

[19.10 从 ORCv1 过渡到 ORCv2  202](#_Toc589863830)

[19.11 操作方法 203](#_Toc1298391112)

[19.11.1 如何管理符号字符串 203](#_Toc1534105053)

[19.11.2 如何创建 JITDylib 并设置链接关系 204](#_Toc1126713736)

[19.11.3 如何删除代码 204](#_Toc1969917459)

[19.11.4 如何添加对自定义程序表示的支持 204](#_Toc748520389)

[19.11.5 如何使用 ThreadSafeModule 和 ThreadSafeContext  206](#_Toc1009115879)

[19.12 如何将进程和库符号添加到 JITDylibs  207](#_Toc578429455)

[19.13 路线图 208](#_Toc599543084)

[19.13.1 当前工作 208](#_Toc1292109146)

[19.13.2 近期工作 208](#_Toc1612018441)

[19.13.3 进一步的未来工作 209](#_Toc1084358281)

[20. JITLink 和 ORC 的 ObjectLinkingLayer 209](#_Toc1360497659)

[20.1 简介 209](#_Toc374989888)

[20.2 JITLink 和 ObjectLinkingLayer  210](#_Toc1008614129)

[20.2.1 ObjectLinkingLayer 插件 210](#_Toc602968108)

[20.3 LinkGraph 212](#_Toc2085683089)

[20.4 通用链接算法 214](#_Toc1510797464)

[20.4.1 passes 216](#_Toc2129294978)

[20.4.2 使用 JITLinkMemoryManager 进行内存管理 217](#_Toc594395043)

[20.4.3 JITLinkMemoryManager 和安全性 218](#_Toc228222330)

[20.4.4 错误处理 219](#_Toc1273511383)

[20.5 Connection to the ORC Runtime 219](#_Toc22928262)

[20.6 构建 LinkGraphs  219](#_Toc1501103188)

[20.7 JIT Linking 219](#_Toc208866955)

[20.7.1 RuntimeDyld  220](#_Toc297002203)

[20.8 llvm-jitlink 工具 220](#_Toc1261060257)

[20.8.1 基本用法 220](#_Toc1144573952)

[20.8.2 llvm-jitlink 作为回归测试工具 221](#_Toc2039122145)

[20.8.3 通过 llvm-jitlink-executor 远程执行 221](#_Toc38752527)

[20.8.4 Harness mode线束模式 221](#_Toc1737033208)

[20.8.5 JITLink 后端开发人员的技巧 222](#_Toc481502327)

[20.9 路线图 223](#_Toc1337143640)

[20.10 实现对 COFF 的支持。 223](#_Toc1123654614)

[21. 调试 JIT 代码 223](#_Toc1608216063)

[21.1.1 背景 223](#_Toc1159577451)

[21.2 GDB 版本 224](#_Toc1872175003)

[21.3 LLDB 版本 224](#_Toc469848294)

[21.4 调试 MCJIT 代码 224](#_Toc1738006906)

[22. 命令行 2.0 库手册 226](#_Toc324234439)

[23. 扩展 LLVM：添加指令、内在函数、类型等 244](#_Toc1761957440)

[24. 附录 251](#_Toc1202541699)

# 概述

<https://llvm.org/docs/UserGuides.html>

## 术语表

|  |  |
| --- | --- |
| 术语 | 解释 |
|  |  |
| ABI | （Application Binary Interface）  ABI 是编译器和链接器遵守的一组规则，以让编译后的程序可以正常工作。ABI里包含很多方面的内容：   * ABI 最大和最重要的部分是规定函数的调用顺序，也称为“调用约定”。调用约定标准化了如何将“函数”转换为汇编代码。 * ABI 还规定了库中公开函数的name（如printf）应该如何表示，以便在链接后可以正确的调用这些库函数并接收参数。 * ABI 还规定可以使用什么类型的数据类型、它们必须如何对齐以及其他低级细节。 * 此外，ABI还涉及操作系统的内容，如可执行文件的格式，虚拟地址空间布局，还有Program Loading and Dynamic Linking等细节。 |  |
| ADCE | Aggressive Dead Code Elimination |
| CFG | control-flow graph |
| DAG | Directed Acyclic Graph (DAG),有向无环图 |
| DCE | Dead Code Elimination |
| DSE | DeadStoreElimination |
| DSL | Domain Specific Language，中文一般译为「领域特定语言」，在《领域特定语言》这本书中它有了一个定义：一种为特定领域设计的，具有受限表达性的编程语言。 |
| DWARF | Debugging with Attribute Record Formats ，是一种调试信息的存放格式。  DWARF 第一版发布于 1992 年，主要是为 UNIX 下的调试器提供必要的调试信息，例如内存地址对应的文件名以及代码行号等信息，通常用于源码级别调试使用。另外通过 DWARF，还能还原运行时的地址成为可读的源码符号（及行号）。 |
| FDE | Frame Description Entry (FDE) |
| GAS | GNU Assembler format (GAS) |
|  |  |
| LCSSA | 闭环 SSA ，Loop Closed SSA |
| LICM | Loop Invariant Code Motion |
| llc | LLVM静态编译器（LLVM Static Compiler，llc）是一个将DAG内容可视化的优秀工具。使用llc的命令行选项，在特定的执行阶段之前或之后显示SelectionDAG。 |
|  |  |
| LTO | Link Time Optimization (LTO) |
| ORC | ORC(On Request Compilation)是LLVM提供的新一代JIT引擎。JIT(Just In Time)也是一个程序，它在运行时，创建并执行了一些新的代码，而这些新代码并不是属于JIT程序本身的。以下是原英文解释：  Whenever a program, while running, creates and runs some new executable code which was not part of the program when it was stored on disk, it’s a JIT. |
| PBQP | Partitioned Boolean Quadratic Programming (PBQP) |
| SCC | strongly connected component |
| SCCP | Sparse Conditional Constant Propagation |
| SEME | Single-Entry-Multiple-Exits |
| SLP vectorization | The goal of SLP vectorization (a.k.a. superword-level parallelism) is to combine similar independent instructions into vector instructions. Memory accesses, arithmetic operations, comparison operations, PHI-nodes, can all be vectorized using this technique. |
| SSA | static single assignment 的缩写，也就是静态单赋值形式。顾名思义，就是每个变量只有唯一的赋值。 |
| TBAA | Type-Based Alias Analysis |
| VLIW | Very Long Instruction Word (VLIW) |

# 使用 CMake 构建 LLVM

## 简介

CMake是一个跨平台的构建生成器工具。CMake 不会构建项目，它会生成构建工具（GNU make、Visual Studio 等）构建 LLVM 所需的文件。

如果您是新贡献者，请从[LLVM 系统入门](https://llvm.org/docs/GettingStarted.html) 页面开始。此页面适用于从旧版配置/制作系统迁移的现有贡献者。

如果您真的很想获得一个功能性的 LLVM 构建，请转到[快速入门](https://llvm.org/docs/CMake.html" \l "quick-start)部分。如果您是 CMake 新手，请从[基本 CMake 用法](https://llvm.org/docs/CMake.html" \l "basic-cmake-usage)开始，然后在您了解自己在做什么 后返回[快速入门](https://llvm.org/docs/CMake.html" \l "quick-start)部分。[选项和变量](https://llvm.org/docs/CMake.html" \l "options-and-variables)部分是你自定义构建的 参考。如果您已经有使用 CMake 的经验，那么这是推荐的起点。

此页面面向 LLVM CMake 构建的用户。如果您正在寻找有关修改 LLVM CMake 构建系统的信息，您可能希望查看 [CMake 入门](https://llvm.org/docs/CMakePrimer.html)页面。它具有 CMake 语言的基本概述。

## 快速入门

我们在这里使用命令行、非交互式 CMake 界面。

1. 下载并安装 CMake。版本 3.13.4 是最低要求。
2. 打开一个shell。必须可以通过 PATH 环境变量， shell 访问您的开发工具。
3. 创建构建目录。不支持在源目录中构建 LLVM。cd 到这个目录：

|  |
| --- |
| $ mkdir mybuilddir$ cd mybuilddir |

1. 在 shell 中执行下面的命令，将path/to/llvm/source/root替换为 LLVM 源代码树根目录的路径：

|  |
| --- |
| $ cmake path/to/llvm/source/root |

CMake 将检测您的开发环境，执行一系列测试，并生成构建 LLVM 所需的文件。CMake 将为所有构建参数使用默认值。有关可以修改的构建参数列表，请参阅[选项和变量](https://llvm.org/docs/CMake.html" \l "options-and-variables)部分。

如果 CMake 无法检测到您的工具集，或者它认为环境不够健全，这可能会失败。在这种情况下，请确保您打算使用的工具集是唯一可从 shell 访问的工具集，并且 shell 本身是适合您的开发环境的工具集。例如，如果您有一个可通过 PATH 环境变量访问的 POSIX shell，CMake 将拒绝构建 MinGW makefile。您可以强制 CMake 使用给定的构建工具；有关说明，请参阅下面的[使用](https://llvm.org/docs/CMake.html" \l "usage)部分。您可能还希望控制 LLVM 启用哪些目标，或者构建哪些 LLVM 组件；请参阅下面的[常用 LLVM 相关变量](https://llvm.org/docs/CMake.html" \l "frequently-used-llvm-related-variables)。

1. CMake 运行完成后，继续使用 IDE 项目文件，或者从构建目录开始构建：

|  |
| --- |
| $ cmake --build . |

该--build选项指示cmake调用底层构建工具（make、ninja、xcodebuild、msbuild等）。当然，可以直接调用底层构建工具，但该--build选项是可移植的。

1. LLVM 完成构建后，从构建目录安装它：

|  |
| --- |
| $ cmake --build . --target install |

除了带有install参数的--target选项之外，--build选项告诉cmake构建install目标。

通过调用构建目录中生成的cmake\_install.cmake脚本，可以在安装时设置不同的安装前缀：

|  |
| --- |
| $ cmake -DCMAKE\_INSTALL\_PREFIX=/tmp/llvm -P cmake\_install.cmake |

## CMake 基本用法

本节介绍您在日常使用中可能需要的 CMake 的基本方面。

CMake 附带大量文档，格式为 html 文件，以及可通过cmake可执行文件本身访问的在线帮助。执行“cmake --help”进一步的帮助选项。

CMake 允许您指定构建工具（例如，GNU make、Visual Studio 或 Xcode）。如果未在命令行中指定，CMake 会根据您的环境尝试猜测要使用的构建工具。一旦确定了您的构建工具，CMake 就会使用相应的生成器为您的构建工具创建文件（例如，Makefiles 或 Visual Studio 或 Xcode 项目文件）。您可以使用命令行选项-G "Name of the generator" 显式指定生成器。要查看系统上可用生成器的列表，请执行

|  |
| --- |
| $ cmake --help |

这将在帮助文本的末尾列出生成器名称。

生成器的名称区分大小写，并且可以包含空格。出于这个原因，您应该完全按照cmake --help输出中列出的方式输入它们 ，用引号括起来。例如，要生成专门针对 Visual Studio 12 的项目文件，可以执行：

|  |
| --- |
| $ cmake -G "Visual Studio 12" path/to/llvm/source/root |

对于给定的开发平台，可以有不止一个足够的生成器。如果您使用 Visual Studio，“NMake Makefiles”是一个生成器，您可以使用它来使用 NMake 进行构建。默认情况下，CMake 选择您的开发环境支持的最具体的生成器。如果你想要一个替代的生成器，你必须用 -G选项告诉 CMake 。

## 选项和变量

变量自定义构建的生成方式。选项是布尔变量，可能的值为 ON/OFF。选项和变量在 CMake 命令行上定义举例如下：

|  |
| --- |
| $ cmake -DVARIABLE=value path/to/llvm/source |

您可以在初始 CMake 调用后设置变量以更改其值。您还可以取消定义变量：

|  |
| --- |
| $ cmake -UVARIABLE path/to/llvm/source |

变量存储在 CMake 缓存中。这是一个由cmake生成的名为CMakeCache.txt 的文件，存储在您的构建目录的根目录中。不建议自己编辑。

变量列在 CMake 缓存，本文档后面的内容中，变量名称和类型用冒号分隔。您还可以在 CMake 命令行上指定变量和类型：

|  |
| --- |
| $ cmake -DVARIABLE:TYPE=value path/to/llvm/source |

## 常用的 CMake 变量

下面是一些经常使用的 CMake 变量，并附有简要说明。有关完整文档，请参阅 CMake 手册，或执行 cmake --help-variable VARIABLE\_NAME。有关控制 LLVM 和启用子项目功能的常用变量的信息，请参阅下面的[常用 LLVM 相关变量](https://llvm.org/docs/CMake.html" \l "frequently-used-llvm-related-variables)。

**CMAKE\_BUILD\_TYPE:STRING**

这配置make或ninja构建的优化级别。默认CMAKE\_BUILD\_TYPE设置为Debug，但您应该仔细阅读下面的列表以找出最适合您的用例的配置。

可能的值：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Build Type | Optimizations | Debug Info | Assertions | Best suited for |
| Release | For Speed | No | No | Users of LLVM and Clang |
| Debug | None | Yes | Yes | Developers of LLVM |
| RelWithDebInfo | For Speed | Yes | No | Users that also need Debug |
| MinSizeRel | For Size | No | No | When disk space matters |

* 优化使 LLVM/Clang 运行得更快，但可能成为逐步调试的障碍。
* 带有调试信息的构建会占用大量 RAM 和磁盘空间，并且通常运行速度较慢。您可以使用 lld 提高 RAM 使用率，请参阅[LLVM\_USE\_LINKER](https://llvm.org/docs/CMake.html" \l "llvm-use-linker)选项。
* 断言是帮助您发现错误的内部检查。它们在启用时通常会减慢 LLVM 和 Clang 的速度，但在开发过程中很有用。您可以手动设置[LLVM\_ENABLE\_ASSERTIONS](https://llvm.org/docs/CMake.html" \l "llvm-enable-assertions)以覆盖 CMAKE\_BUILD\_TYPE 的默认值。

如果您使用的是 Visual Studio 或 Xcode 等 IDE，则应使用 IDE 设置来设置构建类型。

* CMAKE\_INSTALL\_PREFIX:PATH

构建“安装”目标时将安装 LLVM 的路径。

* CMAKE\_{C,CXX}\_FLAGS :STRING

分别编译 C 和 C++ 源文件时要使用的额外标志。

* CMAKE\_{C,CXX}\_COMPILER :STRING

指定要使用的 C 和 C++ 编译器。如果您安装了多个编译器，CMake 可能不会默认为您希望使用的编译器。

## 常用的 LLVM 相关变量

默认配置可能不符合您的要求。以下是经常用于控制它的 LLVM 变量。完整的描述在下面的[LLVM 相关变量](https://llvm.org/docs/CMake.html" \l "llvm-related-variables)中。

* LLVM\_ENABLE\_PROJECTS:STRING

控制启用哪些项目。例如，您可能希望通过指定 -DLLVM\_ENABLE\_PROJECTS="clang;lldb"，工作在clang 或 lldb上。

* LLVM\_ENABLE\_RUNTIMES:STRING

控制启用哪些运行时。例如，您可能希望通过指定 -DLLVM\_ENABLE\_RUNTIMES="libcxx;libcxxabi"，工作在 libc++ 或 libc++abi上。

* LLVM\_LIBDIR\_SUFFIX:STRING

要附加到要安装库的目录的额外后缀。在 64 位架构上，可以使用-DLLVM\_LIBDIR\_SUFFIX=64 将库安装到/usr/lib64.

* LLVM\_PARALLEL\_{COMPILE,LINK}\_JOBS :STRING

构建 llvm 工具链会占用大量资源，尤其是链接。当您使用 Ninja 生成器时，这些选项允许您限制并行度。例如，为避免 OOMs 或进入swap，在 32GB 机器上每 15GB 可用 RAM 只允许一个link job，指定 -G Ninja -DLLVM\_PARALLEL\_LINK\_JOBS=2。

* LLVM\_TARGETS\_TO\_BUILD:STRING

控制使能哪些目标。例如，您可能只需要使用本地目标，例如-DLLVM\_TARGETS\_TO\_BUILD=X86.

* LLVM\_USE\_LINKER:STRING

覆盖系统的默认链接器。例如，使用lld，就使用 -DLLVM\_USE\_LINKER=lld。

## 很少使用的 CMake 变量

下面是一些很少使用的 CMake 变量，以及简要说明和 LLVM 相关注释。有关完整文档，请参阅 CMake 手册，或执行 cmake --help-variable VARIABLE\_NAME。

* CMAKE\_CXX\_STANDARD:STRING

设置 C++ 标准以在构建 LLVM 时符合。可能的值为 17 和 20。LLVM 需要 C++ 17 或更高版本。默认为 17。

* CMAKE\_INSTALL\_BINDIR:PATH

安装可执行文件的路径，相对于CMAKE\_INSTALL\_PREFIX。默认为“bin”。

* CMAKE\_INSTALL\_INCLUDEDIR:PATH

安装头文件的路径，相对于CMAKE\_INSTALL\_PREFIX。默认为“include”。

* CMAKE\_INSTALL\_DOCDIR:PATH

安装文档的路径，相对于CMAKE\_INSTALL\_PREFIX。默认为“share/doc”。

* CMAKE\_INSTALL\_MANDIR:PATH

安装联机帮助页文件的路径，相对于CMAKE\_INSTALL\_PREFIX。默认为share/man”。

## LLVM 相关变量

这些变量可以很好地控制 LLVM 的构建和启用的子项目。几乎所有这些变量名都以 LLVM\_为前缀命名的。

* BUILD\_SHARED\_LIBS:BOOL

指示每个 LLVM 组件（例如支持）是作为共享库（打开）还是作为静态库（关闭）构建的标志。其默认值为 OFF。在 Windows 上，共享库可以在使用 MinGW 构建时使用，包括 mingw-w64，但在使用 Microsoft 工具链构建时则不能。

说明：BUILD\_SHARED\_LIBS 只推荐给 LLVM 开发者使用。如果你想将 LLVM 构建为共享库，你应该使用该 LLVM\_BUILD\_LLVM\_DYLIB选项。

* LLVM\_ABI\_BREAKING\_CHECKS:STRING

用于决定是否应该使用 ABI 中断检查来构建 LLVM。允许的值为WITH\_ASSERTS（默认）、FORCE\_ON和 FORCE\_OFF。 WITH\_ASSERTS在启用断言的构建中打开 ABI 中断检查。 FORCE\_ON ( FORCE\_OFF ) 打开（关闭）它们，无论是否启用正常（NDEBUG-based）断言。使用 ABI 中断检查构建的 LLVM 版本与不使用它构建的版本不兼容。

* LLVM\_UNREACHABLE\_OPTIMIZE:BOOL

此标志控制发布版本中llvm\_unreachable()的行为（通常禁用断言时）。当 ON（默认）时， llvm\_unreachable()被认为是“未定义的行为”并因此进行了优化。当关闭时，它被替换为有保证的“陷阱(trap)”。

* LLVM\_APPEND\_VC\_REV:BOOL

嵌入版本控制修订信息（Git revision ID）。版本信息由 llvm/include/llvm/Support/VCSRevision.h 中的LLVM\_REVISION宏提供 。使用不需要修订信息的 git 的开发人员可以禁用此选项以避免在分支切换后重新链接大多数二进制文件。默认为ON。

* LLVM\_BUILD\_32\_BITS:BOOL

在 64 位系统上构建 32 位可执行文件和库。此选项仅在某些 64 位 Unix 系统上可用。默认为OFF。

* LLVM\_BUILD\_BENCHMARKS:BOOL

将基准添加到默认目标列表。默认为OFF。

* LLVM\_BUILD\_DOCS:BOOL

添加所有启用的文档目标（即 Doxgyen 和 Sphinx 目标）作为默认构建目标的依赖项。这导致所有（启用的）文档目标成为正常构建的一部分。如果install 运行目标，那么这也可以安装所有构建的文档目标。默认为OFF。要启用特定文档目标，请参阅 LLVM\_ENABLE\_SPHINX 和 LLVM\_ENABLE\_DOXYGEN。

* LLVM\_BUILD\_EXAMPLES:BOOL

构建 LLVM 示例。默认为OFF。在任何情况下都会生成用于构建每个示例的目标。有关详细信息，请参阅上面的LLVM\_BUILD\_TOOLS文档。

* LLVM\_BUILD\_INSTRUMENTED\_COVERAGE:BOOL

如果启用，则在构建 llvm 时启用基于源代码的代码覆盖检测。如果 CMake 可以找到代码覆盖脚本以及与您的编译器配对的 llvm-cov 和 llvm-profdata 工具，则构建还将生成generate-coverage-report目标以生成 LLVM 的代码覆盖报告，以及clear-profile -data实用程序目标删除捕获的配置文件数据。有关配置代码覆盖率报告的更多信息，请参阅 LLVM\_CODE\_COVERAGE\_TARGETS和LLVM\_COVERAGE\_SOURCE\_DIRS的文档。

* LLVM\_CODE\_COVERAGE\_TARGETS:STRING

如果设置为以分号分隔的目标列表，这些目标将用于驱动代码覆盖率报告。如果未设置，目标列表将使用 LLVM 构建的 CMake 导出列表构建。

* LLVM\_COVERAGE\_SOURCE\_DIRS:STRING

如果设置为以分号分隔的目录列表，覆盖率报告会将代码覆盖率摘要限制为仅列出的目录。如果未设置，覆盖率报告将包括工具识别的所有来源。

* LLVM\_BUILD\_LLVM\_DYLIB:BOOL

如果启用，将添加构建 libLLVM 共享库的目标。该库在单个共享库中包含所有 LLVM 组件。默认为OFF。这不能与 BUILD\_SHARED\_LIBS 结合使用。如果 LLVM\_LINK\_LLVM\_DYLIB 也处于ON状态，工具将仅链接到 libLLVM 共享库。可以通过将 LLVM\_DYLIB\_COMPONENTS 设置为所需组件的列表来自定义库中的组件。此选项在 Windows 上不可用。

* LLVM\_BUILD\_TESTS:BOOL

在“所有”构建目标中包含 LLVM 单元测试。默认为OFF。在任何情况下都会生成用于构建每个单元测试的目标。您可以使用unittests下定义的目标构建特定的单元测试，例如 ADTTests、IRTests、SupportTests 等。（在unittestsadd\_llvm\_unittest的子目录中搜索以获得完整的单元测试列表。）可以构建所有单元测试目标UnitTests。

* LLVM\_BUILD\_TOOLS:BOOL

构建 LLVM 工具。默认为ON。在任何情况下都会生成构建每个工具的目标。您可以通过调用其目标来单独构建工具。例如，您可以通过在构建目录的根目录下执行make llvm-as来使用基于 Makefile 的系统构建llvm-as 。

* LLVM\_CCACHE\_BUILD:BOOL

如果启用并且该ccache程序可用，则将构建 LLVMccache以加速 LLVM 及其组件的重建。默认为OFF。由ccache 维护的缓存的大小和位置可以通过 LLVM\_CCACHE\_MAXSIZE 和 LLVM\_CCACHE\_DIR 选项进行调整，这些选项分别传递给 CCACHE\_MAXSIZE 和 CCACHE\_DIR 环境变量。

* LLVM\_CREATE\_XCODE\_TOOLCHAIN:BOOL

仅限 macOS：如果启用，CMake 将生成一个名为“install-xcode-toolchain”的目标。该目标将在 $CMAKE\_INSTALL\_PREFIX/Toolchains 中创建一个目录，其中包含一个 xctoolchain 目录，可用于覆盖默认系统工具。

* LLVM\_DEFAULT\_TARGET\_TRIPLE:STRING

未明确指定目标时用于代码生成的 LLVM 目标。它默认为“host”，这意味着它将选择正在构建 LLVM 的机器的体系结构。如果您正在构建交叉编译器，请将其设置为所需体系结构的目标三元组。

* LLVM\_DOXYGEN\_QCH\_FILENAME:STRING

 当-DLLVM\_ENABLE\_DOXYGEN=ON 和 -DLLVM\_ENABLE\_DOXYGEN\_QT\_HELP=ON给定时，将生成的 Qt 压缩帮助文件的文件 。默认为 org.llvm.qch. 此选项仅在与  -DLLVM\_ENABLE\_DOXYGEN\_QT\_HELP=ON 结合使用时才有用；否则无效。

* LLVM\_DOXYGEN\_QHELPGENERATOR\_PATH:STRING

qhelpgenerator可执行文件的路径。默认为 CMake 的 find\_program()可以找到的任何内容。此选项仅在与  -DLLVM\_ENABLE\_DOXYGEN\_QT\_HELP=ON 结合使用时才有用；否则无效。

* LLVM\_DOXYGEN\_QHP\_CUST\_FILTER\_NAME:STRING

有关详细信息，请参阅Qt 帮助项目。默认为 CMake 变量${PACKAGE\_STRING}，它是包名称和版本字符串的组合。然后可以在 Qt Creator 中使用此过滤器，以便在浏览您可能已加载的所有帮助文件时仅选择来自 LLVM 的文档。此选项仅在与 -DLLVM\_ENABLE\_DOXYGEN\_QT\_HELP=ON 结合使用时才有用；否则无效。

* LLVM\_DOXYGEN\_QHP\_NAMESPACE:STRING

中间 Qt 帮助项目文件所在的命名空间。有关详细信息，请参阅Qt 帮助项目 。默认为“org.llvm”。此选项仅在与 -DLLVM\_ENABLE\_DOXYGEN\_QT\_HELP=ON 结合使用时才有用；否则无效。

* LLVM\_DOXYGEN\_SVG:BOOL

对于 Doxygen 输出中的图表，使用 .svg 文件而不是 .png 文件。默认为OFF。

* LLVM\_ENABLE\_ASSERTIONS:BOOL

启用代码断言。当且仅当CMAKE\_BUILD\_TYPE 是Debug时才默认为 ON 。

* LLVM\_ENABLE\_BINDINGS:BOOL

如果禁用，请不要尝试构建 OCaml 绑定。

* LLVM\_ENABLE\_DIA\_SDK:BOOL

使用 MSVC DIA SDK 启用构建以支持 PDB 调试。仅适用于 MSVC。默认为ON。

* LLVM\_ENABLE\_DOXYGEN:BOOL

启用使用 doxygen 生成可浏览的 HTML 文档。默认为OFF。

* LLVM\_ENABLE\_DOXYGEN\_QT\_HELP:BOOL

启用 Qt 压缩帮助文件的生成。默认为关闭。这会影响 make 目标doxygen-llvm。启用后，除了由 doxygen 生成的正常 HTML 输出外，这将生成一个名为org.llvm.qch. 然后您可以将此文件加载到 Qt Creator 中。此选项仅在与 -DLLVM\_ENABLE\_DOXYGEN=ON 结合使用时才有用；否则这没有效果。

* LLVM\_ENABLE\_EH:BOOL

构建具有异常处理支持的 LLVM。如果您希望链接 LLVM 库并在您自己的代码中使用需要通过 LLVM 代码传播的 C++ 异常，则这是必要的。默认为OFF。

* LLVM\_ENABLE\_EXPENSIVE\_CHECKS:BOOL

启用额外的时间/内存昂贵的检查。默认为OFF。

* LLVM\_ENABLE\_FFI :BOOL

指示 LLVM 解释器是否将与外部函数接口库 (libffi) 链接以启用调用外部函数。如果库或其头文件安装在自定义位置，您还可以将变量 FFI\_INCLUDE\_DIR 和 FFI\_LIBRARY\_DIR 分别设置为可以找到 ffi.h 和 libffi.so 的目录。默认为OFF。

* LLVM\_ENABLE\_IDE :BOOL

告诉构建系统正在使用 IDE。这反过来又会禁用某些方便构建系统目标的创建，例如 various install-\*和check-\*目标，因为 IDE 并不总是能很好地处理大量目标。这通常是自动检测的，但可以手动配置以明确控制这些目标的生成。

* LLVM\_ENABLE\_LIBCXX :BOOL

如果主机编译器和链接器支持 stdlib 标志，-stdlib=libc++ 将传递给两者的调用，以便使用 libc++ 而不是 stdlibc++ 构建项目。默认为OFF。

* LLVM\_ENABLE\_LIBPFM :BOOL

使用 libpfm 启用构建以支持 LLVM 工具中的硬件计数器测量。默认为ON。

* LLVM\_ENABLE\_LLD :BOOL

此选项等效于-DLLVM\_USE\_LINKER=lld，但在 2 阶段构建期间除外，其中从第一阶段向第二阶段添加依赖项以确保在 stage2 开始之前构建 lld。

* LLVM\_ENABLE\_LTO :STRING

向编译和链接命令行添加-flto或 -flto= flags，启用链接时优化。可能的-flto=值为Off、 On、Thin和Full。默认为OFF。

* LLVM\_ENABLE\_MODULES :BOOL

使用[Clang 标头模块](https://clang.llvm.org/docs/Modules.html)编译。

* LLVM\_ENABLE\_PEDANTIC :BOOL

启用pedantic模式。如果可能，这会禁用特定于编译器的扩展。默认为ON。

* LLVM\_ENABLE\_PIC :BOOL

如果编译器支持此标志，则将 -fPIC 标志添加到编译器命令行。某些系统，如 Windows，不需要此标志。默认为ON。

* LLVM\_ENABLE\_PROJECTS :STRING

以分号分隔的要构建的项目列表，或用于构建所有（clang、lldb、lld、polly 等）项目的所有项目。此标志假定项目是并排检出而不是嵌套的，即 clang 需要与 llvm 并行而不是嵌套在llvm/tools中。此功能允许使用相同的源签出为 LLVM 构建一个版本，为 clang+llvm 构建另一个版本。完整列表是： clang;clang-tools-extra;cross-project-tests;libc;libclc;lld;lldb;openmp;polly;pstl。

* LLVM\_ENABLE\_RUNTIMES :STRING

使用刚刚构建的编译器构建 libc++、libc++abi、libunwind 或 compiler-rt。这是在组合工具链时构建运行时的正确方法。它将构建与其他运行时分开的内置函数，以保持正确的依赖顺序。如果您想使用系统编译器构建运行时，请参阅libc++ 文档。注意：该列表不应与LLVM\_ENABLE\_PROJECTS重复。完整列表是： compiler-rt;libc;libcxx;libcxxabi;libunwind;openmp 要启用所有这些，请使用： LLVM\_ENABLE\_RUNTIMES=all

* LLVM\_ENABLE\_RTTI :BOOL

使用运行时类型信息构建 LLVM。默认为OFF。

* LLVM\_ENABLE\_SPHINX :BOOL

如果指定，CMake 将搜索sphinx-build可执行文件并使SPHINX\_OUTPUT\_HTML和SPHINX\_OUTPUT\_MANCMake 选项可用。默认为OFF。

* LLVM\_ENABLE\_THREADS :BOOL

构建线程支持（如果可用）。默认为ON。

* LLVM\_ENABLE\_UNWIND\_TABLES :BOOL

在二进制文件中启用展开表。禁用展开表可以减小库的大小。默认为ON。

* LLVM\_ENABLE\_WARNINGS :BOOL

启用所有编译器警告。默认为ON。

* LLVM\_ENABLE\_WERROR :BOOL

如果触发编译器警告，则停止并使构建失败。默认为关闭。

* LLVM\_ENABLE\_Z3\_SOLVER :BOOL

如果启用，则会为 Clang 静态分析器激活 Z3 约束求解器。z3 库的最新版本需要在系统上可用。

* LLVM\_ENABLE\_ZLIB:BOOL

启用使用 zlib 构建以支持 LLVM 工具中的压缩/解压缩。默认为ON。

* LLVM\_EXPERIMENTAL\_TARGETS\_TO\_BUILD :STRING

要构建并链接到 llvm 的以分号分隔的实验目标列表。这将构建实验目标，而无需将其添加到 LLVM 的主 CMakeLists.txt 中所有可用目标的列表中。

* LLVM\_EXTERNAL\_{CLANG,LLD,POLLY}\_SOURCE\_DIR : PATH

这些变量分别指定外部 LLVM 项目 Clang、lld 和 Polly 相对于顶级源目录的源目录路径。如果存在外部项目的树内子目录（例如，Clang 的 llvm/tools/clang），则不会使用相应的变量。如果外部项目的变量未指向有效路径，则不会构建该项目。

* LLVM\_EXTERNAL\_PROJECTS :STRING

作为 llvm 的一部分构建的附加外部项目的分号分隔列表。对于每个项目，必须使用LLVM\_EXTERNAL\_<NAME>\_SOURCE\_DIR 进行项目源代码的路径指定 。例子：-DLLVM\_EXTERNAL\_PROJECTS="Foo;Bar" -DLLVM\_EXTERNAL\_FOO\_SOURCE\_DIR=/src/foo -DLLVM\_EXTERNAL\_BAR\_SOURCE\_DIR=/src/bar

* LLVM\_EXTERNALIZE\_DEBUGINFO :BOOL

生成 dSYM 文件并剥离可执行文件和库（仅限 Darwin）。默认为OFF。

* LLVM\_FORCE\_USE\_OLD\_TOOLCHAIN :BOOL

如果启用，则不会检查编译器和标准库版本。LLVM 可能根本无法编译，或者可能由于这些工具链中的已知错误而在运行时失败。

* LLVM\_INCLUDE\_BENCHMARKS :BOOL

为 LLVM 基准生成构建目标。默认为ON。

* LLVM\_INCLUDE\_EXAMPLES :BOOL

为 LLVM 示例生成构建目标。默认为ON。您可以使用此选项来禁用为 LLVM 示例生成构建目标。

* LLVM\_INCLUDE\_TESTS : BOOL

为 LLVM 单元测试生成构建目标。默认为ON。您可以使用此选项来禁用为 LLVM 单元测试生成构建目标。

* LLVM\_INCLUDE\_TOOLS : BOOL

为 LLVM 工具生成构建目标。默认为开。您可以使用此选项来禁用为 LLVM 工具生成构建目标。

* LLVM\_INSTALL\_BINUTILS\_SYMLINKS :BOOL

从 binutils 工具名称安装符号链接到相应的 LLVM 工具。例如，ar 将被符号链接到 llvm-ar。

* LLVM\_INSTALL\_CCTOOLS\_SYMLINKS :BOOL

将来自 cctools 工具名称的符号安装到相应的 LLVM 工具。例如，lipo 将被符号链接到 llvm-lipo。

* LLVM\_INSTALL\_OCAMLDOC\_HTML\_DIR :STRING

安装 OCamldoc 生成的 HTML 文档的路径。此路径可以是绝对路径，也可以是相对于 CMAKE\_INSTALL\_PREFIX 的路径。默认为 ${CMAKE\_INSTALL\_DOCDIR}/llvm/ocaml-html.

* LLVM\_INSTALL\_SPHINX\_HTML\_DIR:STRING

将 Sphinx 生成的 HTML 文档安装到的路径。此路径可以是绝对路径，也可以是相对于 CMAKE\_INSTALL\_PREFIX 的路径。默认为 ${CMAKE\_INSTALL\_DOCDIR}/llvm/html.

* LLVM\_INSTALL\_UTILS :BOOL

如果启用，实用程序二进制文件FileCheck将not安装到 CMAKE\_INSTALL\_PREFIX。

* LLVM\_INTEGRATED\_CRT\_ALLOC : PATH

在 Windows 上，允许将不同的 C 运行时分配器嵌入到 LLVM 工具和库中。使用如下所列的无锁分配器可以将 ThinLTO 链接时间大大减少大约一个数量级。它还将 Clang 构建时间缩短了大约 5-10%。目前，支持 rpmalloc、snmalloc 和 mimalloc。使用git clone的路径选择相应的分配器，例如：

$ D:\git> git clone https://github.com/mjansson/rpmalloc$ D:\llvm-project> cmake ... -DLLVM\_INTEGRATED\_CRT\_ALLOC=D:\git\rpmalloc

此标志需要与静态 CRT 一起使用，即。如果构建 Release 目标，请添加 -DLLVM\_USE\_CRT\_RELEASE=MT。

* LLVM\_INSTALL\_DOXYGEN\_HTML\_DIR :STRING

安装 Doxygen 生成的 HTML 文档的路径。此路径可以是绝对路径，也可以是相对于CMAKE\_INSTALL\_PREFIX的路径。默认为 ${CMAKE\_INSTALL\_DOCDIR}/llvm/doxygen-html.

* LLVM\_LINK\_LLVM\_DYLIB :BOOL

如果启用，工具将与 libLLVM 共享库链接。默认为关闭。将 LLVM\_LINK\_LLVM\_DYLIB 设置为 ON 也会将 LLVM\_BUILD\_LLVM\_DYLIB 设置为 ON。此选项在 Windows 上不可用。

* LLVM\_LIT\_ARGS : STRING

Arguments given to lit。 make check和make clang-test受到影响。默认情况下，'-sv --no-progress-bar'在 Visual C++ 和 Xcode上，'-sv'在其它上。

* LLVM\_LIT\_TOOLS\_DIR : PATH

用于测试的 GnuWin32 工具的路径。在 Windows 主机上有效。默认为空字符串，在这种情况下，lit 将在您的 %PATH% 中查找测试所需的工具（例如grep,sort等）。如果 GnuWin32 不在您的 %PATH% 中，那么您可以将此变量设置为 GnuWin32 目录，以便 lit 可以在该目录中找到测试所需的工具。

* LLVM\_OPTIMIZED\_TABLEGEN :BOOL

如果启用并构建调试或断言构建，CMake 构建系统将生成 Release 构建树以构建完全优化的 tablegen 以供在构建期间使用。启用此选项可以显着加快构建时间，尤其是在调试配置中构建 LLVM 时。

* LLVM\_PARALLEL\_COMPILE\_JOBS :STRING

定义并发编译作业的最大数量。

* LLVM\_PARALLEL\_LINK\_JOBS : STRING

定义并发链接作业的最大数量。

* LLVM\_PROFDATA\_FILE : PATH

要传递到 clang 的 -fprofile-instr-use 标志的 profdata 文件的路径。这只能在您使用 clang 构建时指定。

* LLVM\_REVERSE\_ITERATION : BOOL

如果启用，所有支持的无序 llvm 容器将以相反的顺序迭代。这对于揭示由无序容器的迭代引起的不确定性很有用。

* LLVM\_STATIC\_LINK\_CXX\_STDLIB : BOOL

如果可能，静态链接到 C++ 标准库。这使用标志“-static-libstdc++”，但如果与LLVM\_ENABLE\_LIBCXX标志一起使用，Clang 主机编译器将静态链接到 libc++。默认为OFF。

* LLVM\_TABLEGEN : STRING

本机 TableGen 可执行文件（通常名为llvm-tblgen）的完整路径。这用于交叉编译：如果用户设置此变量，则不会创建本机 TableGen。

* LLVM\_TARGET\_ARCH : STRING

用于本机代码生成的 LLVM 目标。这是 JIT 生成所必需的。它默认为“host”，这意味着它将选择正在构建 LLVM 的机器的体系结构。如果你正在交叉编译，将它设置为目标体系结构名称。

* LLVM\_TARGETS\_TO\_BUILD : STRING

以分号分隔的要构建的目标列表，或用于构建所有目标的全部。区分大小写。默认为所有。例子： -DLLVM\_TARGETS\_TO\_BUILD="X86;PowerPC"。截至 2022 年 10 月的完整列表是： AArch64;AMDGPU;ARM;AVR;BPF;Hexagon;Lanai;Mips;MSP430;NVPTX;PowerPC;RISCV;Sparc;SystemZ;VE;WebAssembly;X86;XCore

* LLVM\_TEMPORARILY\_ALLOW\_OLD\_TOOLCHAIN : BOOL

如果启用，编译器版本检查只会在使用即将被弃用的工具链时发出警告，而不是发出错误。

* LLVM\_UBSAN\_FLAGS : STRING

定义用于启用 UBSan 的编译标志集。仅在 LLVM\_USE\_SANITIZER包含Undefined. 这可用于覆盖默认的 UBSan 标志集。

* LLVM\_USE\_CRT\_{target} : STRING

在 Windows 上，指示应使用哪个版本的 C 运行时库 (CRT)。例如，-DLLVM\_USE\_CRT\_RELEASE=MT 会将 CRT 静态链接到 LLVM 工具和库中。

* LLVM\_USE\_INTEL\_JITEVENTS : BOOL

启用对 Intel JIT Events API 的构建支持。默认为OFF。

* LLVM\_USE\_LINKER : STRING

添加-fuse-ld={name}到链接调用。可能的值取决于您的编译器，对于 clang，该值可以是您的自定义链接器的绝对路径，否则 clang 将在名称前加上前缀ld.并应用其通常的搜索。例如，要将 LLVM 与 Gold 链接器链接起来，可以使用-DLLVM\_USE\_LINKER=gold.

* LLVM\_USE\_OPROFILE : BOOL

启用构建 OProfile JIT 支持。默认为OFF。

* LLVM\_USE\_PERF : BOOL

启用对 Perf（linux 分析工具）JIT 支持的构建支持。默认为OFF。

* LLVM\_USE\_RELATIVE\_PATHS\_IN\_FILES : BOOL

将源中的绝对源路径和调试信息重写为相对路径。可以通过 LLVM\_SOURCE\_PREFIX 变量调整源前缀。

* LLVM\_USE\_RELATIVE\_PATHS\_IN\_DEBUG\_INFO:BOOL

将调试信息中的绝对源路径重写为相对路径。可以通过 LLVM\_SOURCE\_PREFIX 变量调整源前缀。

* LLVM\_USE\_SANITIZER : STRING

定义用于构建 LLVM 二进制文件和测试的sanitizer。可能的值为Address、Memory、MemoryWithOrigins、Undefined、Thread、 DataFlow和Address;Undefined。默认为空字符串。

* LLVM\_USE\_SPLIT\_DWARF : BOOL

如果启用，CMake 将传递-gsplit-dwarf给编译器。此选项通过减少链接器需要解析的调试信息量来减少链接时内存使用。建议用于使用 ELF 对象格式的平台，例如链接器内存使用率过高的 Linux 系统。

* SPHINX\_EXECUTABLE : STRING

CMake 检测到的可执行文件sphinx-build的路径。有关安装说明，请参阅 https://www.sphinx-doc.org/en/master/usage/installation.html

* SPHINX\_OUTPUT\_HTML :BOOL

如果启用（并且LLVM\_ENABLE\_SPHINX已启用），则会添加将文档构建为 html 的目标（但默认情况下不会构建，除非 LLVM\_BUILD\_DOCS已启用）。源代码树中使用 sphinx 的每个项目都有一个目标（例如 docs-llvm-html,docs-clang-html 和docs-lld-html）。默认为ON。

* SPHINX\_OUTPUT\_MAN :BOOL

如果启用（并且LLVM\_ENABLE\_SPHINX已启用），则会添加用于构建手册页的目标（但默认情况下不会构建，除非LLVM\_BUILD\_DOCS 已启用）。目前唯一添加的目标是docs-llvm-man. 默认为开。

* SPHINX\_WARNINGS\_AS\_ERRORS :BOOL

如果启用，sphinx 文档警告将被视为错误。默认为ON。

## 高级变量

这些都是很少用到的，改变它们的默认值更有可能导致事情出错。它们在 LLVM 版本中也不稳定。

* LLVM\_TOOLS\_INSTALL\_DIR :STRING

安装主要 LLVM 工具的路径，相对于CMAKE\_INSTALL\_PREFIX。默认为CMAKE\_INSTALL\_BINDIR。

* LLVM\_UTILS\_INSTALL\_DIR :STRING

安装辅助 LLVM 实用程序的路径，相对于CMAKE\_INSTALL\_PREFIX。仅在启用LLVM\_INSTALL\_UTILS时才重要。默认为LLVM\_TOOLS\_INSTALL\_DIR。

* LLVM\_EXAMPLES\_INSTALL\_DIR :STRING

使用 LLVM 示例的路径，相对于CMAKE\_INSTALL\_PREFIX。仅在启用LLVM\_BUILD\_EXAMPLES时才重要。默认为“example”。

## CMake 缓存

最近 LLVM 和 Clang 添加了一些更复杂的构建系统功能。使用这些新功能通常涉及在命令行上传递的复杂的 CMake 变量链。Clang 提供了一组 CMake 缓存脚本，使这些功能更易于使用。

使用 CMake 的 -C 标志使用 CMake 缓存文件：

$ cmake -C <path to cache file> <path to sources>

CMake 缓存脚本在隔离范围内处理，只有缓存变量在主配置运行时保持设置。除非指定了 FORCE 选项，否则 CMake 缓存变量不会重置已设置的变量。

关于 CMake 缓存的一些注意事项：

* 命令行参数的顺序很重要
* -C 之前指定的 -D 参数在处理缓存之前设置，并且可以在缓存文件中读取
* -C 之后指定的 -D 参数在缓存处理后设置，并在缓存文件中取消设置
* 所有 -D 参数将覆盖缓存文件设置
* 在缓存文件和命令行参数之后评估 CMAKE\_TOOLCHAIN\_FILE
* 建议在-C之前指定所有 -D 选项

有关通过缓存文件支持的一些高级构建配置的更多信息，请参阅[高级构建配置](https://llvm.org/docs/AdvancedBuilds.html)。

## 执行测试

构建check-all目标时执行测试。例如，如果您使用的是 Makefile，请在构建目录的根目录中执行此命令：

$ make check-all

在 Visual Studio 上，您可以通过构建项目“check-all”来运行测试。有关测试的更多信息，请参阅[LLVM 测试基础架构指南](https://llvm.org/docs/TestingGuide.html)。

## 交叉编译

有关如何使用 CMake 进行交叉编译的一般说明，请参阅此 [wiki 页面](https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/CrossCompiling)。它进行了详细的解释，可能看起来令人生畏，但事实并非如此。在 wiki 页面上有几个示例，包括工具链文件。直接转到 Information how to set up various cross compiling toolchains

快速解决方案部分。

另请参阅[LLVM 相关变量](https://llvm.org/docs/CMake.html" \l "llvm-related-variables)部分了解交叉编译时使用的变量。

## 在你的项目中嵌入 LLVM

从 LLVM 3.5 开始，CMake 构建系统将 LLVM 库导出为可导入的 CMake 目标。这意味着 LLVM 的client 现在可以可靠地使用 CMake 针对已安装的 LLVM 版本开发他们自己的基于 LLVM 的项目，而不管它是如何构建的。

下面是一个导入 LLVM 库并使用它们构建简单应用程序的 CMakeLists.txt 文件的简单示例simple-tool。

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.13.4)  project(SimpleProject)  find\_package(LLVM REQUIRED CONFIG)  message(STATUS "Found LLVM ${LLVM\_PACKAGE\_VERSION}")  message(STATUS "Using LLVMConfig.cmake in: ${LLVM\_DIR}")  # Set your project compile flags.  # E.g. if using the C++ header files  # you will need to enable C++11 support  # for your compiler.  include\_directories(${LLVM\_INCLUDE\_DIRS})  separate\_arguments(LLVM\_DEFINITIONS\_LIST NATIVE\_COMMAND ${LLVM\_DEFINITIONS})  add\_definitions(${LLVM\_DEFINITIONS\_LIST})  # Now build our tools  add\_executable(simple-tool tool.cpp)  # Find the libraries that correspond to the LLVM components  # that we wish to use  llvm\_map\_components\_to\_libnames(llvm\_libs support core irreader)  # Link against LLVM libraries  target\_link\_libraries(simple-tool ${llvm\_libs}) |

在 CONFIG 模式下使用时， find\_package(...)指令（如上例所示）将在不同位置查找LLVMConfig.cmake文件（有关详细信息，请参阅 cmake 手册）。它创建一个LLVM\_DIR缓存条目来保存LLVMConfig.cmake找到的目录或允许用户指定目录（例如，通过传递-DLLVM\_DIR=/usr/lib/cmake/llvm给cmake命令或直接在ccmake或中设置cmake-gui）。

该文件位于两个不同的位置。

* <LLVM\_INSTALL\_PACKAGE\_DIR>/LLVMConfig.cmake<LLVM\_INSTALL\_PACKAGE\_DIR> 作为 LLVM CMake 模块已安装的LLVM 版本的一部分安装的位置在。 这通常 cmake/llvm/位于 lib 目录中。在 Linux 上，这通常是 /usr/lib/cmake/llvm/LLVMConfig.cmake.
* <LLVM\_BUILD\_ROOT>/lib/cmake/llvm/LLVMConfig.cmake<LLVM\_BUILD\_ROOT>LLVM 构建树的根在哪里 。注意：这仅在使用 CMake 构建 LLVM 时可用。

如果 LLVM 安装在您操作系统的正常安装前缀中（例如，在 Linux 上通常是/usr/），如果安装正确，find\_package(LLVM ...)将自动找到 LLVM。如果未安装 LLVM，或者您希望直接针对 LLVM 构建树进行构建，您可以 像前面提到的那样使用LLVM\_DIR。

该LLVMConfig.cmake文件设置了各种有用的变量。值得注意的变量包括

* LLVM\_CMAKE\_DIR

LLVM CMake 目录的路径（即包含 LLVMConfig.cmake 的目录）。

* LLVM\_DEFINITIONS

针对 LLVM 构建时应使用的预处理器定义列表。

* LLVM\_ENABLE\_ASSERTIONS

如果 LLVM 是使用断言构建的，则此项设置为 ON，否则设置为 OFF。

* LLVM\_ENABLE\_EH

如果 LLVM 是在启用异常处理 (EH) 的情况下构建的，则此项设置为 ON，否则设置为 OFF。

* LLVM\_ENABLE\_RTTI

如果 LLVM 是使用运行时类型信息 (RTTI) 构建的，则此项设置为 ON，否则设置为 OFF。

* LLVM\_INCLUDE\_DIRS

包含 LLVM 头文件的目录的包含路径列表。

* LLVM\_PACKAGE\_VERSION

LLVM 版本。该字符串可以与 CMake 条件一起使用，例如.if (${LLVM\_PACKAGE\_VERSION} VERSION\_LESS "3.5")

* LLVM\_TOOLS\_BINARY\_DIR

包含 LLVM 工具的目录路径（例如llvm-as）。

请注意，在上面的示例中，我们链接simple-tool 而不是几个 LLVM 库。库列表是使用 llvm\_map\_components\_to\_libnames() CMake 函数确定的。有关可用组件的列表，请查看llvm-config --components运行的输出。

请注意，对于 LLVM < 3.5 ，使用了llvm\_map\_components\_to\_libraries()，而不是llvm\_map\_components\_to\_libnames(). 现在已弃用，并将在 LLVM 的未来版本中删除。

## Developing LLVM passes out of source

可以从 LLVM 的源代码树中开发 LLVM passes（即针对已安装或构建的 LLVM）。下面提供了项目布局的示例。

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

<project dir>/CMakeLists.txt的内容：

|  |
| --- |
| find\_package(LLVM REQUIRED CONFIG)  separate\_arguments(LLVM\_DEFINITIONS\_LIST NATIVE\_COMMAND ${LLVM\_DEFINITIONS})  add\_definitions(${LLVM\_DEFINITIONS\_LIST})  include\_directories(${LLVM\_INCLUDE\_DIRS})  add\_subdirectory(<pass name>) |

<project dir>/<pass name>/CMakeLists.txt的内容：

|  |
| --- |
| add\_library(LLVMPassname MODULE Pass.cpp) |

add\_llvm\_library请注意，如果您打算在将来的某个时候将此传递合并到 LLVM 源代码树中，那么使用 LLVM 的内部函数和 MODULE 参数可能更有意义， 而不是通过……

将以下内容添加到<project dir>/CMakeLists.txt （ find\_package(LLVM ...)之后 ）

|  |
| --- |
| list(APPEND CMAKE\_MODULE\_PATH "${LLVM\_CMAKE\_DIR}")include(AddLLVM) |

然后修改<project dir>/<pass name>/CMakeLists.txt 为

|  |
| --- |
| add\_llvm\_library(LLVMPassname MODULE  Pass.cpp  ) |

完成开发过程后，您可能希望将其集成到 LLVM 源代码树中。您可以通过两个简单的步骤实现它：

1.将<pass name>文件夹复制到<LLVM root>/lib/Transform目录中。

2.将add\_subdirectory(<pass name>)行添加到 <LLVM root>/lib/Transform/CMakeLists.txt。

# 编写 LLVM Pass

## 简介——什么是Pass？

LLVM Pass Framework 是 LLVM 系统的重要组成部分，因为 LLVM pass 是编译器的最有趣的部分的所在。 Pass 执行构成编译器的转换和优化，它们构建这些转换使用的分析结果，最重要的是，它们是编译器代码的一种结构化技术。

所有 LLVM passes 都是 [Pass](https://llvm.org/doxygen/classllvm_1_1Pass.html) 类的子类，它们通过重写从Pass 继承的虚函数来实现相应的功能。  根据您的 Pass 的工作方式，您应该继承自[ModulePass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-modulepass)、[CallGraphSCCPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-callgraphsccpass)、[FunctionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)或[LoopPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-looppass)或[RegionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-regionpass)类，这为系统提供了有关您的 Pass 的更多信息，以及它如何与其他Pass 相结合。LLVM Pass Framework 的主要特性之一是，它根据您的 pass 满足的约束（由它们的父类指示）来调度 pass 以有效的方式运行。

我们首先向您展示如何构建一个 pass，包括从编写代码到编译、加载和执行它的所有内容。在了解了基础知识之后，将讨论更高级的功能。

**警告**

本文档涉及 legacy pass manager。LLVM 默认为优化 pipeline 使用新的 pass manager（codegen pipeline 仍在使用旧的pass manager），它有自己定义 pass 的方式。有关详细信息，请参阅[编写 LLVM pass](https://llvm.org/docs/WritingAnLLVMNewPMPass.html) 和 使用[新的 pass manager](https://llvm.org/docs/NewPassManager.html)。要将旧的 pass manager 与opt 一起使用，请将-enable-new-pm=0标志传递给所有opt调用。

## 快速入门——编写 hello world

这里我们描述如何编写 pass 的“hello world”。“Hello”pass 旨在简单地打印出正在编译的程序中存在的非外部函数的名称。它根本不修改程序，它只是检查它。此 pass 的源代码和文件可在lib/Transforms/Hello目录中的 LLVM 源代码树中找到。

### 设置构建环境

首先，配置和构建 LLVM。接下来，您需要在 LLVM 源库的某处创建一个新目录。对于此示例，我们假设您创建了lib/Transforms/Hello 目录. 最后，您必须设置一个 build script 来编译 new pass 的源代码。为此，将以下内容复制到CMakeLists.txt：

|  |
| --- |
| add\_llvm\_library( LLVMHello MODULE  Hello.cpp  PLUGIN\_TOOL  opt  ) |

和以下行lib/Transforms/CMakeLists.txt：

|  |
| --- |
| add\_subdirectory(Hello) |

（请注意，已经有一个以Hello的目录，该目录下是“Hello”pass示例；您可以使用它——在这种情况下您不需要修改任何 CMakeLists.txt文件——或者，如果您想从头开始创建所有内容，请使用另一个名称.)

此 build script 指当前目录中的定Hello.cpp文件将被编译并链接到一个共享对象$(LEVEL)/lib/LLVMHello.so中，该共享对象可以由opt工具通过其-load 选项动态加载。如果您的操作系统使用的后缀不是.so（例如 Windows 或 macOS），则会使用相应的扩展名。

现在我们已经设置了 build script ，我们只需要为通道本身编写代码。

### 所需的基本代码

现在我们有了编译 new pass 的方法，我们只需要编写它。开始于：

|  |
| --- |
| #include "llvm/Pass.h"  #include "llvm/IR/Function.h"  #include "llvm/Support/raw\_ostream.h" |

这是必需的，因为我们正在编写一个 Pass，我们正在对 [Functions](https://llvm.org/doxygen/classllvm_1_1Function.html) 进行操作，并且我们将进行一些打印。

接下来我们有：

|  |
| --- |
| using namespace llvm; |

…这是必需的，因为 inlcude 文件中的函数位于 llvm 命名空间中。

接下来我们有：

|  |
| --- |
| namespace { |

…从一个匿名命名空间开始。匿名命名空间对于 C++ ，就像“ static”关键字对于 C（在全局范围内）一样。它使在匿名命名空间内声明的内容仅对当前文件可见。如果您不熟悉它们，请查阅一本不错的 C++ 书籍以获取更多信息。

接下来，我们声明我们的 pass 本身：

|  |
| --- |
| struct Hello : public FunctionPass { |

这声明了一个“ Hello”类，它是[FunctionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)的子类。稍后将详细描述不同的内置 pass 子类，但现在，知道FunctionPass一次对一个函数进行操作。

|  |
| --- |
| static char ID;  Hello() : FunctionPass(ID) {} |

这声明了 LLVM 用来标识 pass 的 pass 标识符。这允许 LLVM 避免使用昂贵的 C++ 运行时信息。

|  |
| --- |
| bool runOnFunction(Function &F) override {  errs() << "Hello: ";  errs().write\_escaped(F.getName()) << '\n';  return false;  }}; // end of struct Hello} // end of anonymous namespace |

我们声明了一个 [runOnFunction](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-runonfunction) 方法，它覆盖了一个从[FunctionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)继承的抽象虚方法。这是我们应该做我们的事情的地方，所以我们只用每个函数的名称打印出我们的消息。

|  |
| --- |
| char Hello::ID = 0; |

我们在这里初始化 pass ID。LLVM 使用 ID 的地址来标识一个 pass，因此，初始化值并不重要。

|  |
| --- |
| static RegisterPass<Hello> X("hello", "Hello World Pass",  false /\* Only looks at CFG \*/,  false /\* Analysis Pass \*/); |

最后，我们注册我们的类 Hello，给它一个命令行参数“ hello”，和一个名字“Hello World Pass”。最后两个参数描述了它的行为：如果一个pass在不修改它的情况下遍历 CFG，那么第三个参数设置为true；如果 pass 是一个 analysis pass，例如，dominator tree pass，则true作为第四个参数提供。

整个.cpp文件看起来像这样：

|  |
| --- |
| #include "llvm/IR/Function.h"  #include "llvm/Support/raw\_ostream.h"  #include "llvm/IR/LegacyPassManager.h"  #include "llvm/Transforms/IPO/PassManagerBuilder.h"  using namespace llvm;  namespace {  struct Hello : public FunctionPass {  static char ID;  Hello() : FunctionPass(ID) {}  bool runOnFunction(Function &F) override {  errs() << "Hello: ";  errs().write\_escaped(F.getName()) << '\n';  return false;  }  }; // end of struct Hello  } // end of anonymous namespace  char Hello::ID = 0;  static RegisterPass<Hello> X("hello", "Hello World Pass",  false /\* Only looks at CFG \*/,  false /\* Analysis Pass \*/); |

现在所有都整合在一起了，从你的构建目录的顶层用一个简单的“gmake”命令编译文件，你应该得到一个新文件“ lib/LLVMHello.so”。请注意，此文件中的所有内容都包含在一个匿名命名空间中——这反映了一个事实，即pass是自包含的单元，不需要外部接口（尽管它们可以有）就可以使用。

### 使用opt来运行一个pass

现在你有了一个全新的闪亮的共享对象文件，我们可以使用 opt 命令运行通过你的 pass 的LLVM 程序。因为您使用 RegisterPass 注册了您的 pass，加载后您将能够使用 opt工具访问它。

要测试它，请按照[LLVM 系统入门](https://llvm.org/docs/GettingStarted.html)中的示例将“Hello World”编译到 LLVM。我们现在可以像这样通过我们的转换运行程序的bitcode文件 (hello.bc)（当然，任何bitcode文件都可以）：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -hello < hello.bc > /dev/null  Hello: \_\_main  Hello: puts  Hello: main |

该-load选项指定opt应将您的 pass 加载为共享对象，这使“ -hello”成为有效的命令行参数（这是您需要[注册 pass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-registration) 的原因之一）。因为 Hello pass 没有以任何有趣的方式修改程序，所以我们只是丢弃了 opt的结果（将其发送到/dev/null）。

要查看您注册的其他字符串发生了什么，请尝试 使用 -help 选项运行opt ：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -help  OVERVIEW: llvm .bc -> .bc modular optimizer and analysis printer  USAGE: opt [subcommand] [options] <input bitcode file>  OPTIONS:  Optimizations available:  ...  -guard-widening - Widen guards  -gvn - Global Value Numbering  -gvn-hoist - Early GVN Hoisting of Expressions  -hello - Hello World Pass  -indvars - Induction Variable Simplification  -inferattrs - Infer set function attributes  ... |

pass 名称被添加为 pass 的信息字符串，为opt的用户提供一些文档。现在您已经有了工作通道，您可以继续让它进行您想要的很酷的转换。一旦您完成所有工作并经过测试，了解您的传球速度可能会很有用。[PassManager](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-passmanager)提供了一个很好的命令行选项 ([-time-passes](https://llvm.org/docs/CommandGuide/opt.html" \l "cmdoption-opt-time-passes))，它允许您获取有关您的pass以及您排队的其他pass的执行时间的信息。例如：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -hello -time-passes < hello.bc > /dev/null  Hello: \_\_main  Hello: puts  Hello: main  ===-------------------------------------------------------------------------===  ... Pass execution timing report ...  ===-------------------------------------------------------------------------===  Total Execution Time: 0.0007 seconds (0.0005 wall clock)  ---User Time--- --User+System-- ---Wall Time--- --- Name ---  0.0004 ( 55.3%) 0.0004 ( 55.3%) 0.0004 ( 75.7%) Bitcode Writer  0.0003 ( 44.7%) 0.0003 ( 44.7%) 0.0001 ( 13.6%) Hello World Pass  0.0000 ( 0.0%) 0.0000 ( 0.0%) 0.0001 ( 10.7%) Module Verifier  0.0007 (100.0%) 0.0007 (100.0%) 0.0005 (100.0%) Total |

如您所见，我们上面的实现非常快。列出的额外 pass 由opt工具自动插入，以验证您的pass发出的 LLVM 仍然是有效且格式正确的 LLVM，它没有以某种方式被破坏。

现在您已经了解了 pass 背后的基本机制，我们可以讨论更多有关 pass 工作原理和使用方法的细节。

## pass 相关类和要求

在设计一个新的 pass 时，您应该做的第一件事就是决定您应该为您的 pass 子类化哪个类。[Hello World](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-basiccode) 示例使用[FunctionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)类来实现，但我们没有讨论为什么或何时会发生这种情况。在这里，我们讨论可用的类，从最一般的到最具体的。

在为您的Pass选择superclass时，您应该尽可能选择最具体的类，同时仍然能够满足列出的要求。这为 LLVM Pass Infrastructure 提供了必要的信息来优化 pass 的运行方式，这样生成的编译器就不会不必要地变慢。

### ImmutablePass 类

最简单和乏味的 pass 类型是“ ImmutablePass ”类。此 pass 类型用于不必运行、不更改状态且永远不需要更新的pass。这不是转换或分析的正常类型，但可以提供有关当前编译器配置的信息。

虽然这个 pass 类很少使用，但它对于提供有关正在编译的当前目标机器的信息以及可能影响各种转换的其他静态信息很重要。

ImmutablePasses 永远不会使其他转换无效，永远不会无效，也永远不会“运行”。

### ModulePass 类

ModulePass类是您可以使用的所有 superclasses 中最通用的。派生自 ModulePass表示您的 pass 将整个程序作为一个单元，以不可预测的顺序引用函数体，或者添加和删除函数。因为对ModulePass 子类的行为一无所知，所以无法对其执行进行优化。

如果函数 pass 不需要任何模块或不可变passes， ModulePass 可以使用函数级 passes（例如支配者），该函数级 passes使用 getAnalysis接口getAnalysis<DominatorTree>(llvm::Function \*)来提供检索分析结果的函数。请注意，这只能对运行分析的函数进行，例如，在支配者的情况下，您应该只请求DominatorTree的函数定义，而不是声明。

要编写正确的ModulePass子类，请派生ModulePass并覆盖runOnModule具有以下签名的方法：

#### runOnModule 方法

|  |
| --- |
| virtual bool runOnModule(Module &M) = 0; |

该runOnModule方法执行 pass 的有趣工作。如果模块被转换修改， 它应该返回true;否则，返回false。

### CallGraphSCCPass 类

CallGraphSCCPass由需要在调用图上自下而上遍历程序（调用者之前的被调用者）的pass使用。派生自CallGraphSCCPass提供了一些构建和遍历 的机制CallGraph，但也允许系统优化CallGraphSCCPasses 的执行。如果您的pass 满足下面列出的要求，但不满足 FunctionPass的要求，您应该从 CallGraphSCCPass 类派生.

TODO: 简单解释一下SCC, Tarjan's algo, BU是什么意思。

明确地说，CallGraphSCCPass 子类是：

1....不允许检查或修改除当前 SCC 和 SCC 的直接调用者和直接被调用者之外的任何 Function。

2....需要保留当前CallGraph对象，更新它以反映对程序所做的任何更改。

3.…不允许在当前模块中添加或删除 SCC，尽管它们可能会更改 SCC 的内容。

4.…允许在当前模块中添加或删除全局变量。

5.…允许在调用[runOnSCC](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-runonscc)时保持状态（包括全局数据）。

在某些情况下，实现 CallGraphSCCPass 有点棘手，因为它必须处理其中包含多个节点的 SCC。如果它们修改了程序，下面描述的所有虚拟方法都应该返回true，或者 false如果它们没有。

#### doInitialization(CallGraph &)方法

virtual bool doInitialization(CallGraph &CG);

允许该doInitialization方法做大部分 CallGraphSCCPasses 不允许做的事情。他们可以添加和删除函数，获取函数指针等。该doInitialization方法旨在执行stuff的简单初始化类型， 该stuff不依赖于正在处理的 SCC 。该doInitialization方法调用不会与任何其他pass 的执行重叠（因此它应该非常快）。

#### runOnSCC方法

virtual bool runOnSCC(CallGraphSCC &SCC) = 0;

该runOnSCC方法执行 pass 的有趣工作，如果模块被转换修改，则应该返回true;否则，返回false 。

#### doFinalization(CallGraph &) 方法

virtual bool doFinalization(CallGraph &CG);

该方法是一个不常用的方法，当 pass 框架为正在编译的程序中的每个 SCCdoFinalization完成调用runOnSCC时调用。

### FunctionPass 类

与ModulePass子类相比，FunctionPass子类确实具有系统期望的可预测的本地行为。所有 FunctionPass在程序中的每个函数上执行，独立于程序中的所有其他函数。 FunctionPasses不要求按特定顺序执行，FunctionPasses不修改外部函数。

明确地说，FunctionPass子类不允许：

1.检查或修改Functions不同于当前正在处理的那个。

2.在当前Module 添加或删除Functions。

3.在当前Module 添加或删除全局变量。

4.在调用[runOnFunction](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-runonfunction)时维护状态（包括全局数据）。

实现 一个FunctionPass通常很简单（例如，参见Hello World pass）。 FunctionPasses 可以override三个虚拟方法来完成它们的工作。如果他们修改了程序，所有这些方法都应该返回true，或者 如果他们没有修改，返回false。

#### doInitialization(Module &)方法

virtual bool doInitialization(Module &M);

允许该doInitialization方法做大部分 FunctionPasses 不允许做的事情。他们可以添加和删除函数，获取函数指针等。该doInitialization方法旨在执行 stuff 的简单初始化类型， 该stuff 不依赖于正在处理的函数。该doInitialization方法调用不会调度与任何其他pass的执行overlap（因此它应该非常快）。

说明如何使用此方法的一个很好的示例是 [LowerAllocations](https://llvm.org/doxygen/LowerAllocations_8cpp-source.html) pass。该pass将malloc和free指令转换为平台相关 malloc()和free()函数调用。它使用该doInitialization 方法获取所需的malloc和free函数的引用，并在必要时向模块添加原型。

#### runOnFunction方法

virtual bool runOnFunction(Function &F) = 0;

该runOnFunction方法必须由您的子类实现，以完成您的pass的转换或分析工作。像往常一样，如果函数被修改，应该返回一个true值。

#### doFinalization(Module &)方法

virtual bool doFinalization(Module &M);

该doFinalization方法是一个不常用的方法，当 pass 框架完成为正在编译的程序中的每个函数调用runOnFunction时调用。

### LoopPass 类

所有LoopPass在函数中的每个循环上执行，独立于函数中的所有其他循环。 LoopPass 按循环嵌套顺序处理循环，以便最后处理最外层循环。

允许LoopPass 子类使用LPPassManager 接口更新循环嵌套。实现 loop pass 通常很简单。 LoopPasses 可以override三个虚拟方法来完成它们的工作。如果他们修改了程序，所有这些方法都应该返回true，或者如果他们没有修改返回false 。

LoopPass 子类作为main loop pass pipeline的一部分运行的，它需要保留其他loop pass 所需的所有相同功能分析。为了使这更容易，由LoopUtils.h提供了一个getLoopAnalysisUsage 函数。它可以在子类的getAnalysisUsage override中调用以获得一致的和正确的行为。同理，INITIALIZE\_PASS\_DEPENDENCY(LoopPass) 将初始化这组功能分析。

#### doInitialization(Loop \*, LPPassManager &)方法

virtual bool doInitialization(Loop \*, LPPassManager &LPM);

该doInitialization方法旨在执行stuff 的简单初始化类型，该stuff不依赖于正在处理的函数。该 doInitialization方法的不会调度与任何其他pass执行相重叠（因此它应该非常快）。 LPPassManager接口应该用于访问Function或Module级别分析信息。

#### runOnLoop方法

virtual bool runOnLoop(Loop \*, LPPassManager &LPM) = 0;

该runOnLoop方法必须由您的子类实现，以完成您的pass的转换或分析工作。像往常一样，如果函数被修改，应该返回一个true值。 LPPassManager接口应该用于更新循环嵌套。

#### doFinalization()方法

virtual bool doFinalization();

该doFinalization方法是一个不常用的方法，当 pass 框架完成为正在编译的程序中的每个循环调用runOnLoop时调用。

### RegionPass 类

RegionPass类似于LoopPass，但在函数中的每个单入口单出口区域执行。 RegionPass以嵌套顺序处理区域，以便最后处理最外面的区域。

允许RegionPass子类使用 RGPassManager接口更新区域树。您可以 override RegionPass的三个虚拟方法 来实现您自己的region pass。如果他们修改了程序，所有这些方法都应该返回true，或者如果他们没有修改返回false。

#### doInitialization(Region \*, RGPassManager &)方法

virtual bool doInitialization(Region \*, RGPassManager &RGM);

该doInitialization方法旨在执行stuff 的简单初始化类型，该stuff不依赖于正在处理的函数。该 doInitialization方法的调用不会调度与任何其他pass的执行相重叠（因此它应该非常快）。 RPPassManager接口应该用于访问Function或Module级别分析信息。

#### runOnRegion方法

virtual bool runOnRegion(Region \*, RGPassManager &RGM) = 0;

该runOnRegion方法必须由您的子类实现，以完成您的pass的转换或分析工作。像往常一样，如果区域被修改，应该返回一个真值。 RGPassManager接口应该用于更新区域树。

#### doFinalization()方法

virtual bool doFinalization();

该doFinalization方法是一个不常用的方法，当 pass 框架为正在编译的程序中的每个区域完成调用runOnRegion时调用。

### MachineFunctionPass 类

一个MachineFunctionPass是 LLVM 代码生成器的一部分，它在程序中每个 LLVM 函数的机器相关表示上执行。

Code generator passes 由 TargetMachine::addPassesToEmitFile和类似例程来专门注册和初始化，因此它们通常不能从opt或bugpoint命令运行。

一个MachineFunctionPass也是一个FunctionPass，因此适用于 FunctionPass的所有限制也适用于它。 MachineFunctionPasses 也有额外的限制。特别是，MachineFunctionPass不允许执行以下任何操作：

1.修改或创建任何 LLVM IR Instructions、BasicBlocks、 Arguments、Functions、GlobalVariables、 GlobalAliases 或Modules。

2.修改一个MachineFunction不同于当前正在处理的那个。

3.在调用runOnMachineFunction（包括全局数据）时维护状态。

#### runOnMachineFunction(MachineFunction &MF)方法

virtual bool runOnMachineFunction(MachineFunction &MF) = 0;

runOnMachineFunction可以被认为是 MachineFunctionPass 的主要入口点 ；也就是说，您应该override 此方法来完成您的MachineFunctionPass.

该runOnMachineFunction方法在 Module 中的 每个 MachineFunction 对象上调用，以便MachineFunctionPass 可以对函数的机器相关表示进行优化。如果您想为MachineFunction 获得您正在处理的 LLVM Function，请使用 MachineFunction 的 getFunction() 访问器方法——但请记住，您不能从MachineFunctionPass 修改 LLVM function 或它的内容。

### Pass registration (pass 注册)

在Hello World示例 pass 中，我们说明了 pass 注册的工作原理，并讨论了使用它的一些原因以及它的作用。在这里，我们讨论如何以及为何注册 pass。

正如我们在上面看到的， pass 是用RegisterPass模板注册的。模板参数是要在命令行上使用的 pass 的名称，以指定应将 pass 添加到程序中（例如，使用 opt或bugpoint）。第一个参数是 pass 的名称，它用于-help程序的输出，以及由–debug-pass选项生成的调试输出。

如果你想让你的 pass 很容易转储，你应该实现虚拟 print 方法：

#### print方法

virtual void print(llvm::raw\_ostream &O, const Module \*M) const;

该print方法必须通过“分析”来实现，以便打印分析结果的人类可读版本。这对于调试分析本身以及其他人了解分析的工作原理很有用。使用 opt-analyze参数调用此方法。

该llvm::raw\_ostream参数指定要写入结果的流，该Module参数给出一个指向已分析程序的顶层模块的指针。但是请注意，此指针在某些情况下（例如，从调试器调用 Pass::dump()）可能是NULL ，因此它应该只用于增强调试输出，不应该依赖它。

### 指定 passes 之间的交互

PassManager 的主要职责之一是确保 passes 彼此正确交互。因为PassManager要 [优化pass 的执行](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-passmanager)，所以它必须知道 pass 如何相互交互以及不同 pass 之间存在哪些依赖关系。为了跟踪这一点，每个 pass 都可以声明在当前 pass 之前需要执行的 pass 集，以及被当前 pass 无效的pass。

通常，此功能用于要求在运行 pass 之前计算分析结果。运行任意转换 passes 可以使计算的分析结果无效，这个计算结果是无效集指定的。如果一个 pass 没有实现[getAnalysisUsage](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-getanalysisusage)方法，它默认没有任何先决条件 passes，并使所有其他passes无效。

#### getAnalysisUsage方法

virtual void getAnalysisUsage(AnalysisUsage &Info) const;

通过实现该getAnalysisUsage方法，可以为您的转换指定所需的和无效的集合。实现应在AnalysisUsage对象中填充使用有关哪些 passes 是必需的且未失效的信息。为此，pass 可以在AnalysisUsage对象上调用以下任何方法：

#### AnalysisUsage::addRequired<>和AnalysisUsage::addRequiredTransitive<>方法

如果您的 pass 需要执行之前的pass（例如分析），它可以使用这些方法之一来安排它在您的pass之前运行。LLVM 有许多可能需要的不同类型的分析和pass，范围从DominatorSet 到 BreakCriticalEdges. 例如，要求 BreakCriticalEdges保证在运行 pass 时 CFG 中没有临界边。

一些分析链接到其他分析以完成它们的工作。例如， AliasAnalysis <AliasAnalysis>实现需要链接到其他别名分析过程。在分析链的情况下，应该使用addRequiredTransitive 方法而不是addRequired 方法。This informs the PassManager that the transitively required pass should be alive as long as the requiring pass is。

#### AnalysisUsage::addPreserved<>方法

PassManager其中一项工作是优化运行分析的方式和时间。特别是，除非需要，否则它会尝试避免重新计算数据。出于这个原因，如果可用，允许passes声明它们保留（即，它们不会使无效）现有分析。例如，一个简单的常量折叠pass不会修改 CFG，因此它不可能影响支配分析的结果。默认情况下，假定所有passes都使所有其他passes无效。

该类AnalysisUsage提供了几种在与 addPreserved 相关的特定情况下很有用的方法。特别是， setPreservesAll可以调用该方法来指示 pass 根本不修改 LLVM 程序（这对于分析是正确的），并且该 setPreservesCFG方法可以用于更改程序中的指令但不修改 CFG 或终止指令。

对于像 BreakCriticalEdges 这样的转换， addPreserved特别有用 。这个 pass 知道如何更新一小组循环和支配者相关分析（如果它们存在），因此它可以保留它们，尽管它破解了 CFG。

getAnalysisUsage 的实现举例

|  |
| --- |
| // This example modifies the program, but does not modify the CFG  void LICM::getAnalysisUsage(AnalysisUsage &AU) const {  AU.setPreservesCFG();  AU.addRequired<LoopInfoWrapperPass>();  } |

#### getAnalysis<>和getAnalysisIfAvailable<>方法

该Pass::getAnalysis<>方法由您的类自动继承，使您能够访问您声明的使用[getAnalysisUsage](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-getanalysisusage)方法所需的pass。它采用一个模板参数来指定您想要的 pass 类，并返回对该pass的引用。例如：

|  |
| --- |
| bool LICM::runOnFunction(Function &F) {  LoopInfo &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();  //...  } |

此方法调用返回对所需pass的引用。如果您尝试获取未在getAnalysisUsage实现中按要求声明的分析，您可能会遇到运行时断言失败。此方法可由您的run\*方法实现调用，或由您的能调用到run\* 的任何其他本地方法。

模块级 pass 可以通过此接口使用function级分析信息。例如：

|  |
| --- |
| bool ModuleLevelPass::runOnModule(Module &M) {  //...  DominatorTree &DT = getAnalysis<DominatorTree>(Func);  //...  } |

In above example, runOnFunction for DominatorTree is called by pass manager before returning a reference to the desired pass。

如果您的 pass 能够更新存在的分析（例如， 如上所述的 BreakCriticalEdges），您可以使用该 getAnalysisIfAvailable方法，如果分析处于活动状态，它会返回指向分析的指针。例如：

|  |
| --- |
| if (DominatorSet \*DS = getAnalysisIfAvailable<DominatorSet>()) {  // A DominatorSet is active. This code will update it.  } |

### 实现分析组(Analysis Groups)

现在我们已经了解了 pass 的定义方式、使用方式以及其他 pass 的要求等基础知识，是时候变得更有趣了。到目前为止我们看到的所有 pass 关系都非常简单：一个 pass 依赖于另一个特定的 pass 才能运行。对于许多应用程序来说，这很好，而对于其他应用程序，则需要更大的灵活性。

特别是，一些分析被定义为只有一个简单的分析结果界面，但有多种计算方法。例如考虑别名分析。最简单的别名分析会为任何别名查询返回“may alias”。最复杂的分析是流程敏感、上下文敏感的过程间分析，可能需要花费大量时间来执行（显然，这两个极端之间有很大的空间用于其他实现）。为了清楚地支持这种情况，LLVM Pass Infrastructure 支持分析组的概念。

#### 分析组概念

分析组是一个单一的简单接口，可以通过多个不同的 passes 来实现。分析组可以像pass一样被赋予人类可读的名称，但与pass不同的是，它们不需要从Pass 类派生。一个分析组可能有一个或多个实现，其中之一是“默认”实现。

client 端passes使用分析组，就像其他passes一样： AnalysisUsage::addRequired()和Pass::getAnalysis()方法。为了解决这个要求，[PassManager](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-passmanager)扫描可用的passes以查看分析组的任何实现是否可用。如果没有可用的，则创建默认实现以供传递使用。所有[pass之间相互作用](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-interaction)的标准规则 仍然适用。

尽管[Pass Registration](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-registration)对于普通pass 是可选的，但是所有分析组实现都必须注册，并且必须使用[INITIALIZE\_AG\_PASS](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-registeranalysisgroup)模板加入实现池。此外，接口的默认实现必须 注册到[RegisterAnalysisGroup](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-registeranalysisgroup)。

作为实际分析组的具体示例，请考虑 [AliasAnalysis](https://llvm.org/doxygen/classllvm_1_1AliasAnalysis.html)分析组。别名分析接口（ [basic-aa](https://llvm.org/doxygen/structBasicAliasAnalysis.html)pass）的默认实现只是做一些不需要大量分析来计算的简单检查（例如：两个不同的全局变量永远不能相互别名等）。使用[AliasAnalysis](https://llvm.org/doxygen/classllvm_1_1AliasAnalysis.html)接口的 pass（例如 [gvn](https://llvm.org/doxygen/classllvm_1_1GVN.html) pass），并不关心实际提供了哪个别名分析实现，它们只使用指定的接口。

从用户的角度来看，命令就像平常一样工作。发出命令“opt -gvn ...”将导致basic-aa类被实例化并添加到传递序列(pass sequence)中。发出命令“opt -somefancyaa -gvn ...”将导致 gvn pass 使用somefancyaa别名分析（实际上并不存在，这只是一个假设的例子）。

#### 使用RegisterAnalysisGroup

RegisterAnalysisGroup模板用于注册分析组本身，而INITIALIZE\_AG\_PASS用于向分析组添加pass实现。首先，应注册一个分析组，并为其提供一个人类可读的名称。与pass注册不同，没有为分析组接口本身指定的命令行参数，因为它是“抽象的”：

|  |
| --- |
| static RegisterAnalysisGroup<AliasAnalysis> A("Alias Analysis"); |

注册分析后，通过使用以下代码， passes可以声明它们是接口的有效实现：

|  |
| --- |
| namespace {  // Declare that we implement the AliasAnalysis interface  INITIALIZE\_AG\_PASS(FancyAA, AliasAnalysis , "somefancyaa",  "A more complex alias analysis implementation",  false, // Is CFG Only?  true, // Is Analysis?  false); // Is default Analysis Group implementation?  } |

这仅显示了一个类FancyAA，该类使用INITIALIZE\_AG\_PASS宏来注册和“加入” [AliasAnalysis](https://llvm.org/doxygen/classllvm_1_1AliasAnalysis.html)分析组。分析组的每个实现都应使用此宏加入。

|  |
| --- |
| namespace {  // Declare that we implement the AliasAnalysis interface  INITIALIZE\_AG\_PASS(BasicAA, AliasAnalysis, "basic-aa",  "Basic Alias Analysis (default AA impl)",  false, // Is CFG Only?  true, // Is Analysis?  true); // Is default Analysis Group implementation?  } |

这里我们展示了如何指定默认实现（使用INITIALIZE\_AG\_PASS模板的最终参数）。对于要使用的分析组，必须始终只有一个默认实现可用。只有默认实现可以派生自ImmutablePass. 这里我们声明[BasicAliasAnalysis](https://llvm.org/doxygen/structBasicAliasAnalysis.html)传递是接口的默认实现。

## pass统计

[Statistic](https://llvm.org/doxygen/Statistic_8h_source.html)类被设计成一种简单的方法来公开传球的各种成功指标。当在命令行上启用[-stats](https://llvm.org/docs/CommandGuide/opt.html" \l "cmdoption-opt-stats)命令行选项时，这些统计信息会在运行结束时打印出来。有关详细信息，请参阅程序员手册中的[统计部分](https://llvm.org/docs/ProgrammersManual.html" \l "statistic)。

### PassManager 做什么

PassManager类获取 passes 列表，确保正确设置其先决条件 ，然后安排 passes 以高效运行。所有运行passes的 LLVM 工具都使用 PassManager 来执行这些passes。

PassManager 做了两件主要的事情来尝试减少一系列传递(passes)的执行时间：

1.共享分析结果。PassManager 尽可能避免重新计算分析结果 的尝试。这意味着跟踪哪些分析已经可用，哪些分析无效，以及需要运行哪些分析才能通过。工作的一个重要部分是PassManager跟踪所有分析结果的确切生命周期，允许它在不再需要时立即释放分配给保存分析结果的[内存](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-releasememory)。

2.流水线程序的传递执行(Pipeline the execution of passes on the program)。 PassManager 尝试通过将传递流水线化在一起，从一系列传递中获得更好的缓存和内存使用行为。这意味着，给定一系列连续的[FunctionPass](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)，它将在第一个函数上执行所有[FunctionPasses](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-functionpass)，然后 在第二个函数上执行所有 FunctionPass，依此类推，直到整个程序运行完 passes。

这改进了编译器的缓存行为，因为它一次只接触单个函数的 LLVM 程序表示，而不是遍历整个程序。它减少了编译器的内存消耗;因为，例如，一次只需要计算一个[DominatorSet](https://llvm.org/doxygen/classllvm_1_1DominatorSet.html) 。这也使得将来可以实现一些[有趣的增强](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-smp)功能。

PassManager的有效性直接受到它所拥有的有关其正在调度的passes行为的信息量的影响。例如，面对未实现的[getAnalysisUsage](https://llvm.org/docs/WritingAnLLVMPass.html" \l "writing-an-llvm-pass-getanalysisusage)方法，“保留”集是有意保守的。在应该实现时不实现将导致不允许任何分析结果在您的传递执行过程中存在。

该类PassManager公开了一个--debug-pass命令行选项，可用于调试传递执行、查看工作方式以及诊断何时应保留比当前更多的分析。（要获取有关--debug-pass选项所有变体的信息，只需键入“ opt -help-hidden”）。

例如，通过使用 –debug-pass=Structure 选项，我们可以看到我们的 Hello World pass 如何与其他 pass 交互。让我们尝试使用 gvn 和 licm passes：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -gvn -licm --debug-pass=Structure < hello.bc > /dev/null  ModulePass Manager  FunctionPass Manager  Dominator Tree Construction  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Memory Dependence Analysis  Global Value Numbering  Natural Loop Information  Canonicalize natural loops  Loop-Closed SSA Form Pass  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Scalar Evolution Analysis  Loop Pass Manager  Loop Invariant Code Motion  Module Verifier  Bitcode Writer |

此输出向我们展示了何时构建pass。在这里我们看到 GVN 使用支配树信息来完成它的工作。LICM pass使用自然循环信息，它也使用支配树。

LICM pass 后，模块验证程序运行（由opt工具自动添加），它使用支配树检查生成的 LLVM 代码是否格式正确。请注意，支配树只计算一次，并由三个passes共享。

让我们看看当运行Hello World传递时，我们在两次传递之间这会发生什么变化：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -gvn -hello -licm --debug-pass=Structure < hello.bc > /dev/null  ModulePass Manager  FunctionPass Manager  Dominator Tree Construction  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Memory Dependence Analysis  Global Value Numbering  Hello World Pass  Dominator Tree Construction  Natural Loop Information  Canonicalize natural loops  Loop-Closed SSA Form Pass  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Scalar Evolution Analysis  Loop Pass Manager  Loop Invariant Code Motion  Module Verifier  Bitcode Writer  Hello: \_\_main  Hello: puts  Hello: main |

在这里我们看到Hello World pass 已经杀死了 Dominator Tree pass，即使它根本没有修改代码！要解决此问题，我们需要将以下getAnalysisUsage方法添加到我们的传递中：

|  |
| --- |
| // We don't modify the program, so we preserve all analyses  void getAnalysisUsage(AnalysisUsage &AU) const override {  AU.setPreservesAll();  } |

现在当我们运行我们的 pass 时，我们得到这个输出：

|  |
| --- |
| $ opt -load lib/LLVMHello.so -gvn -hello -licm --debug-pass=Structure < hello.bc > /dev/null  Pass Arguments: -gvn -hello -licm  ModulePass Manager  FunctionPass Manager  Dominator Tree Construction  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Memory Dependence Analysis  Global Value Numbering  Hello World Pass  Natural Loop Information  Canonicalize natural loops  Loop-Closed SSA Form Pass  Basic Alias Analysis (stateless AA impl)  Function Alias Analysis Results  Scalar Evolution Analysis  Loop Pass Manager  Loop Invariant Code Motion  Module Verifier  Bitcode Writer  Hello: \_\_main  Hello: puts  Hello: main |

这表明我们不会再不小心使支配者信息无效，因此不必计算两次。

#### releaseMemory方法

virtual void releaseMemory();

PassManager自动确定何时计算分析结果，以及将它们保留多长时间。因为传递对象本身的生命周期实际上是编译过程的整个持续时间，所以我们需要一些方法来释放不再有用的分析结果。releaseMemory虚拟方法是执行此操作的方法。

如果您正在编写一个分析或任何其他保留大量状态的传递（供另一个“需要”您的传递并使用getAnalysis方法的传递使用），您应该实现releaseMemory，好吧，释放分配给维护这个内存状态的内存。此方法在该类的run\*方法之后调用，在您的传递中下一次调用run\*方法之前调用。

## 注册动态加载的passes

在使用 LLVM 构建生产质量工具时，大小很重要，既用于分发目的，也用于在目标系统上运行时调节驻留代码大小。因此，有选择地使用一些passes，同时省略其他passes并保持以后更改配置的灵活性，变得可取且非常有必要的。您希望能够完成所有这些，并向用户提供反馈。这是passes注册发挥作用的地方。

passes注册的基本机制是 的 MachinePassRegistry类和子类MachinePassRegistryNode。

MachinePassRegistry的实例用于维护 MachinePassRegistryNode对象列表。此实例维护列表并将添加和删除传达给命令行界面。

子类MachinePassRegistryNode的实例用于维护提供的有关特定pass的信息。此信息包括命令行名称、命令帮助字符串和用于创建pass实例的函数的地址。这些实例之一的全局静态构造函数MachinePassRegistry进行相应的注册，静态析构函数注销。因此，工具中静态链接的pass将在启动时注册。动态加载的pass将在加载时注册并在卸载时注销。

### 使用现有的注册表

有预定义的注册表来跟踪指令调度 ( RegisterScheduler) 和寄存器分配 ( RegisterRegAlloc) 器pass。这里我们将描述如何注册寄存器分配器pass。

实施您的寄存器分配器pass。在您的寄存器分配器 .cpp文件中添加以下include：

|  |
| --- |
| #include "llvm/CodeGen/RegAllocRegistry.h" |

同样在您的寄存器分配器.cpp文件中，以以下形式定义creator函数：

|  |
| --- |
| FunctionPass \*createMyRegisterAllocator() {  return new MyRegisterAllocator();  } |

请注意，此函数的签名应与 RegisterRegAlloc::FunctionPassCtor 的type匹配. 在同一文件中添加“installing”声明，格式如下：

|  |
| --- |
| static RegisterRegAlloc myRegAlloc("myregalloc",  "my register allocator help string",  createMyRegisterAllocator); |

-help请注意，帮助字符串之前的两个空格会在查询中产生整洁的结果 。

|  |
| --- |
| $ llc -help  ...  -regalloc - Register allocator to use (default=linearscan)  =linearscan - linear scan register allocator  =local - local register allocator  =simple - simple register allocator  =myregalloc - my register allocator help string  ... |

就是这样。用户现在可以自由选择-regalloc=myregalloc作为选项使用。注册指令调度器是类似的，除了使用 RegisterScheduler类。请注意， RegisterRegAlloc::FunctionPassCtor与 RegisterScheduler::FunctionPassCtor明显不同 。

要强制将您的寄存器分配器加载/链接到 llc / lli工具中，请将您的creator函数的全局声明添加到Passes.h并添加一个“pseudo”调用行到 llvm/Codegen/LinkAllCodegenComponents.h。

### 创建新的注册表

最简单的入门方法是克隆一个现有的注册表；我们推荐llvm/CodeGen/RegAllocRegistry.h。要修改的关键是类名和FunctionPassCtor类型。

然后你需要声明注册表。示例：如果您的pass注册表是 RegisterMyPasses，然后定义：

|  |
| --- |
| MachinePassRegistry RegisterMyPasses::Registry; |

最后，为您的pass声明命令行选项。例子：

|  |
| --- |
| cl::opt<RegisterMyPasses::FunctionPassCtor, false,  RegisterPassParser<RegisterMyPasses> >  MyPassOpt("mypass",  cl::init(&createDefaultMyPass),  cl::desc("my pass option help")); |

这里的命令选项是“ mypass”，createDefaultMyPass作为默认创建者。

### 将 GDB 与动态加载的pass一起使用

不幸的是，使用带有动态加载pass的 GDB 并不像它应该的那样容易。首先，你不能在还没有加载的共享对象中设置断点，其次共享对象中的内联函数存在问题。这里有一些使用 GDB 调试 pass 的建议。

为了便于讨论，我将假设您正在调试由opt调用的转换，尽管此处描述的内容均不依赖于此。

#### 在你的传递中设置一个断点

您要做的第一件事是在 opt 进程上启动 gdb：

|  |
| --- |
| $ gdb opt  GNU gdb 5.0  Copyright 2000 Free Software Foundation, Inc.  GDB is free software, covered by the GNU General Public License, and you are  welcome to change it and/or distribute copies of it under certain conditions.  Type "show copying" to see the conditions.  There is absolutely no warranty for GDB. Type "show warranty" for details.  This GDB was configured as "sparc-sun-solaris2.6"...  (gdb) |

注意opt里面有很多调试信息，所以加载需要时间。耐心一点。由于我们还不能在我们的传递中设置断点（共享对象直到运行时才加载），我们必须执行该过程，并在它调用我们的传递之前让它停止，但在它加载共享对象之后。最简单的方法是设置一个断点， PassManager::run然后使用您想要的参数运行该过程：

|  |
| --- |
| $ (gdb) break llvm::PassManager::run  Breakpoint 1 at 0x2413bc: file Pass.cpp, line 70.  (gdb) run test.bc -load $(LLVMTOP)/llvm/Debug+Asserts/lib/[libname].so -[passoption]  Starting program: opt test.bc -load $(LLVMTOP)/llvm/Debug+Asserts/lib/[libname].so -[passoption]  Breakpoint 1, PassManager::run (this=0xffbef174, M=@0x70b298) at Pass.cpp:70  70 bool PassManager::run(Module &M) { return PM->run(M); }  (gdb) |

一旦opt在方法中停止，PassManager::run您现在可以自由地在您的传递中设置断点，以便您可以跟踪执行或执行其他标准调试操作。

#### 杂项问题

一旦掌握了基础知识，GDB 就会遇到一些问题，有些有解决方案，有些则没有。

* 内联函数具有伪造的堆栈信息。一般来说，GDB 在获取堆栈跟踪和单步执行内联函数方面做得非常好。但是，当动态加载pas时，它会以某种方式完全失去此功能。我知道的唯一解决方案是取消内联函数（将它从类的主体移动到.cpp文件）。
* 重新启动程序会中断断点。遵循以上信息后，您已经成功地在您的通行证中设置了一些断点。接下来你知道，你重新启动程序（即，你再次键入“run”），你开始收到关于断点不可设置的错误。我发现“修复”这个问题的唯一方法是删除你的 pass 中已经设置的断点，运行程序，一旦程序在PassManager::run中停止执行，重置所有断点。

希望这些技巧对常见的案例调试情况有所帮助。如果您想贡献自己的一些技巧，请联系Chris。

### 计划未来的扩展

尽管 LLVM Pass Infrastructure 就目前而言非常强大，并且做了一些漂亮的事情，但我们希望在未来添加一些东西。这是我们要去的地方：

#### 多线程 LLVM

多 CPU 机器变得越来越普遍，编译速度永远不够快：显然我们应该允许多线程编译器。由于上面为传递定义的语义（特别是它们不能在调用它们的run\*方法时保持状态），实现多线程编译器的一个很好的干净方法是为PassManager类创建每个传递对象的多个实例，并允许单独的实例同时对程序的不同部分进行hacking。

此实现将防止每个pass都必须实现多线程构造，只需要 LLVM 核心在几个地方锁定（用于全局资源）。虽然这是一个简单的扩展，但我们根本没有时间（或多处理器机器，这是一个原因）来实现它。尽管如此，我们已经让 LLVM 通过 SMP 准备就绪，您也应该这样做。

# LLVM 的分析和转换过程

## 简介

本文档作为 LLVM 提供的优化功能的高级摘要。优化被实现为遍历程序的某些部分以收集信息或转换程序的pass。下表将 LLVM 提供的pass分为三类。分析pass计算信息，其他pass可以使用或用于调试或程序可视化目的。转换passes可以使用（或使无效）分析passes。转换以某种方式传递所有改变的程序。Utility passes提供了一些Utility，但不符合分类。例如，将函数提取到位码或将模块写入位码的passes既不是分析过程也不是转换过程。

## 分析passes(Analysis Passes)

本节介绍 LLVM 分析 passes。

* -aa-eval: 详尽的别名分析精度评估

这是一个简单的 N^2 别名分析精度评估器。基本上，对于程序中的每个函数，它只是查询以查看别名分析实现如何回答函数中每对指针之间的别名查询。

这受到 Naveen Neelakantam、Francesco Spadini 和 Wojciech Stryjewski 代码的启发和改编。

* -basic-aa: 基本别名分析（无状态 AA 实现）

实现标识的基本别名分析传递（两个不同的全局变量不能别名，等），但不进行状态分析。

* -basiccg: 基本调用图构造

还未实现？

* -count-aa: 计数别名分析查询响应

可用于计算进行了多少别名查询以及正在使用的相应的别名分析如何实现的。

* -da: 依赖分析

依赖分析框架，用于检测内存访问中的依赖关系。

* -debug-aa: AA 使用调试器

这个简单的pass 会检查别名分析用户以确保，如果他们创建新值，他们不会在未通知 AA 该值的情况下查询 AA。它可以作为您想要的任何其他 AA pass的垫片(shim)。

是的，跟踪程序中的每个值是昂贵的，但这是一个调试过程。

* -domfrontier: Dominance Frontier Construction，支配边界构建

这个过程是一个简单的支配者构造算法，用于寻找前向支配者边界。

* -domtree: 支配树构建(Dominator Tree Construction)

这个过程是一个简单的支配者构造算法，用于寻找正向支配者。

* -dot-callgraph：将调用图打印到“点”文件

这个 pass，只在 opt 中可用，将调用图打印成一个.dot 图。然后可以使用“dot”工具处理此图，将其转换为 postscript 或其他一些合适的格式。

* -dot-cfg: 将函数的 CFG 打印到“dot”文件

此通道仅在 opt 中可用，将控制流图打印成 .dot图形。然后可以使用dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。此外，该-cfg-func-name=<substring>选项可用于过滤打印的函数。将打印包含指定子字符串的所有函数。

* -dot-cfg-only: 将函数的 CFG 打印到“dot”文件（没有函数体）

此通道仅在 opt 中可用，将控制流图打印成 .dot图形，省略函数体。然后可以使用dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。此外，该-cfg-func-name=<substring>选项可用于过滤打印的函数。将打印包含指定子字符串的所有函数。

* -dot-dom：将函数的支配树(dominator tree)打印到“dot”文件

此通道仅在 opt 中可用，将支配树打印到.dot 图形中。然后可以使用dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。

* -dot-dom-only：将函数的支配树(dominator tree)打印到“dot”文件（没有函数体）

此通道仅在 opt 中可用，将支配树打印成.dot 图形，省略函数体。然后可以使用 dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。

* -dot-postdom：将函数的后支配树(post dominator tree)打印到“dot”文件

此通道仅在 opt 中可用，将后支配树(post dominator tree)打印到 .dot图形中。然后可以使用dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。

* -dot-postdom-only：将函数的后支配树(post dominator tree)打印到“dot”文件（没有函数体）

此通道仅在 opt 中可用，将后支配树打印成 .dot图形，省略函数体。然后可以使用dot工具处理此图，将其转换为 postscript 或其他一些合适的格式。

* -globalsmodref-aa: 全局的简单 mod/ref 分析

这个简单的传递为没有使用其地址的全局值提供别名和 mod/ref 信息，并跟踪函数是读取还是写入内存（“pure”）。对于这种简单（但非常常见）的情况，我们可以提供相当准确和有用的信息。

* -instcount: 计算各种类型的Instructions

此遍收集所有指令的计数并报告它们。

* -intervals: Interval Partition Construction

此分析计算并表示函数的区间划分或预先存在的区间划分。

以这种方式，区间划分可用于将流图减少到其退化的单节点区间分区（除非它是不可约的）。

* -iv-users: 归纳变量用户

从归纳变量计算出的表达式的“interesting”用户的簿记。

* -lazy-value-info: 惰性数值信息分析

值约束信息的惰性计算接口。

* -libcall-aa: LibCall 别名分析

LibCall 别名分析。

* -lint: Statically lint-checks LLVM IR

此pass静态检查常见且易于识别的构造，这些构造会在 LLVM IR 中产生未定义或可能意外的行为。

它不是正确性的保证，有两种方式。首先，它不全面。有些检查可以静态完成，但尚未实施。其中一些由 TODO 注释指出，但这些也不全面。其次，许多条件无法静态检查。这个过程没有动态检测，所以它不能检查所有可能的问题。

另一个限制是它假定所有代码都将被执行。通过一个从未到达的基本块中的空指针进行存储是无害的，但无论如何都会对此发出警告。

优化过程可能会使此过程检查的条件或多或少变得明显。如果优化过程似乎引入了警告，则可能是优化过程只是暴露了代码中的现有条件。

此代码可能在[instcombine](https://llvm.org/docs/Passes.html" \l "passes-instcombine)之前运行。在许多情况下，instcombine 会检查相同类型的事物，并将具有未定义行为的指令变为无法访问（或等效）。因此，此pass会努力查看bitcasts等。

* -loops: 自然循环信息

该分析用于识别自然循环并确定 CFG 各个节点的循环深度。请注意，识别出的循环实际上可能是几个共享相同头节点的自然循环……而不仅仅是一个自然循环。

* -memdep: 内存依赖分析

一种分析，确定对于给定的内存操作，它依赖于哪些先前的内存操作。它建立在别名分析信息的基础上，并试图为一种常见的别名信息查询提供一个惰性的缓存接口。

* -module-debuginfo: 解码模块级调试信息

此通道解码模块中的调试信息元数据并以（充分准备好的）人类可读的形式打印。

例如，opt与-analyze选项一起运行此传递，它将打印到标准输出。

* -postdomfrontier: 后支配边界构造(Post-Dominance Frontier Construction)

这个pass是一个简单的后支配构造算法，用于寻找后支配边界。

* -postdomtree: 后支配树构造(Post-Dominator Tree Construction)

这个过程是一个简单的后支配构造算法，用于寻找后支配者。

* -print-alias-sets: 别名设置打印机

未实现。

* -print-callgraph: 打印调用图

此传递仅在 opt 中可用，以人类可读的形式将调用图打印到标准错误。

* -print-callgraph-sccs: 打印调用图的 SCC

此通道仅在 opt 中可用，它以人类可读的形式将调用图的 SCC 打印到标准错误。

* -print-cfg-sccs: 打印每个函数 CFG 的 SCC

此通道仅在 opt 中可用，将每个函数 CFG 的 SCC 打印为人类可读形式的标准错误。

* -print-externalfnconstants: Print external fn callsites passed constants

此传递仅在 opt 中可用，它打印出调用位置到使用常量参数调用的外部函数。这在寻找我们在别名分析中应该常量折叠或处理的标准库函数时很有用。

* -print-function: 打印函数到 stderr

该PrintFunctionPass类被设计为其它 FunctionPasses 进行流水线处理 ，并在处理时打印出模块的函数。

* -print-module: 打印模块到 stderr

此pass在执行时简单地打印出整个模块。

* -print-used-types: 查找使用过的类型

此pass用于查找程序使用的所有类型。请注意，此分析明确不包括仅由符号表使用的类型。

* -regions: 检测单入口单出口区域

RegionInfo pass 检测函数中的单入口单出口区域，其中区域定义为仅在两个点连接到剩余图的任何子图。此外，构建了分层区域树(hierarchical region tree)。

* -scalar-evolution: 标量演化分析(Scalar Evolution Analysis)

该ScalarEvolution分析可用于分析和分类循环中的标量表达式。它专门识别一般归纳变量，用抽象和不透明的SCEV类表示它们。鉴于此分析，可以获得循环的行程计数(trip count)和其他重要属性。

该分析主要用于归纳变量替换和强度缩减。

* -scev-aa: 基于 ScalarEvolution 的别名分析

根据ScalarEvolution查询实现的简单别名分析。

这不同于传统的循环依赖分析，因为它测试循环的单个迭代内的依赖性，而不是不同迭代之间的依赖性。

ScalarEvolution比临时(ad-hoc)分析的BasicAliasAnalysis集合对指针算法有更完整的理解。

* -stack-safety: 堆栈安全分析

该StackSafety分析可用于确定堆栈分配的变量是否可以被认为是安全的，不受内存访问错误的影响。

该分析的主要目的是供消毒剂(sanitizers)使用，以避免对安全变量进行不必要的检测。

* -targetdata: 目标数据布局(Target Data Layout)

为其他pass提供有关的各种数据类型的ABI 目标对所需的大小和对齐方式的信息的访问权限。

## Transform Passes

本节介绍 LLVM 转换passes。

* -adce：积极的死代码消除(Aggressive Dead Code Elimination)

ADCE 积极尝试消除代码。此过程与DCE类似，但它假设值是死的，直到证明不是这样。这类似于SCCP，除了应用于值的生命周期。

* -always-inline:always\_inline函数的内联器

一个自定义内联器，它只处理标记为“always inline”的函数。

* -argpromotion：将“by reference”参数提升为标量

此 pass 将“by reference”参数提升为“by value”参数。实际上，这意味着寻找具有指针参数的内部函数。如果它可以通过使用别名分析证明只加载了一个参数，那么它可以将值而不是值的地址传递给函数。这会导致代码的递归简化并导致 allocas 的消除（尤其是在像 STL 这样的 C++ 模板代码中）。

此过程还处理传递给函数的聚合参数，如果仅加载聚合的元素，则对它们进行标量化。请注意，它拒绝标量聚合，这需要将三个以上的操作数传递给函数，因为为大型数组或结构传递数千个操作数是无利可图的！

请注意，此转换也可以用于仅存储到（而不是返回值）但当前不存储的参数。当 LLVM 开始支持函数的多个返回值时，最好处理这种情况。

* -bb-vectorize: 基本块矢量化(Basic-Block Vectorization)
* 此pass 组合基本块内的指令以形成向量指令。它遍历每个基本块，尝试配对兼容的指令，重复此过程，直到没有可以进行矢量化的其它额外的配对。当某对兼容指令的输出被另一对兼容指令用作输入时，这些对是潜在矢量化链的一部分。指令对仅在它们属于比某个阈值长度更长的链的一部分时才会融合到向量指令中。此外，pass 尝试为每对兼容指令找到最佳链。这些启发式方法旨在防止向量化，因为它不会提高结果代码的性能。
* -block-placement: 配置文件引导的基本块放置

这个 pass 是一个非常简单的配置文件引导的基本块放置算法。这个想法是将频繁执行的块放在函数的开头，并希望增加失败条件分支的数量。如果没有特定函数的配置文件信息，则此 pass 基本上按深度优先顺序对块进行排序。

* -break-crit-edges: 打破 CFG 中的临界边(Break critical edges in CFG)

通过插入虚拟基本块来断开 CFG 中的所有关键边。无法处理关键边的 pass 可能“required”它。这种转换显然会使CFG失效，但可以更新前向支配者（set, immediate dominators, tree, and frontier）信息。

* -codegenprepare：优化代码生成

此pass 修改输入函数中的代码，以便更好地为基于 SelectionDAG 的代码生成做好准备。这解决了其 basic-block-at-a-time 方法的局限性。它最终应该被删除。

* -constmerge: 合并重复的全局常量

将重复的全局常量合并为一个共享的常量。这很有用，因为某些 passes（例如， TraceValues）会在程序中插入大量字符串常量，而不管现有字符串是否可用。

* -dce: 死代码消除(Dead Code Elimination)

死代码消除类似于[死指令消除](https://llvm.org/docs/Passes.html" \l "passes-die)，但它会重新检查已删除指令使用的指令，以查看它们是否是新死的。

* -deadargelim: 死参数消除(Dead Argument Elimination)

此pass 从内部函数中删除死参数。死参数消除删除直接死的参数，以及仅作为其他函数的死参数传递给函数调用的参数。此 pass 还以类似的方式删除死参数。

此 pass通常用作清理pass，以在激进的过程间passes 之后运行，这会添加可能无效的参数。(This pass is often useful as a cleanup pass to run after aggressive interprocedural passes, which add possibly-dead arguments.)

* -deadtypeelim: 死亡类型消除(Dead Type Elimination)

此pass 用于清理 GCC 的输出。它使用[find used types pass](https://llvm.org/docs/Passes.html" \l "passes-print-used-types)消除了整个翻译单元中未使用的类型的名称。

* -die: 死指令消除(Dead Instruction Elimination)

死指令消除对函数执行单次pass，删除明显死的指令。

* -dse: 死存储消除(Dead Store Elimination)

一个简单的死存储消除，只考虑基本块本地冗余存储。

* -function-attrs: 推导函数属性(Deduce function attributes)

一个简单的过程间pass遍历调用图，寻找不访问或只读取非本地内存的函数，并将它们标记为 readnone/ readonly。此外，如果对函数的调用没有创建任何比调用还长久的指针值副本，它会将函数参数（指针类型）标记为“nocapture”。这或多或少意味着指针只是被取消引用，而不是从函数返回或存储在全局中。此pass 以调用图的自下而上遍历来实现。

* -globaldce: 死亡全局变量消除(Dead Global Elimination)

此转换旨在从程序中消除无法访问的内部全局变量。它使用一种激进的算法，搜索已知存在的全局变量。在找到所有需要的全局变量后，它会删除剩下的任何东西。这允许它删除无法访问的程序的递归块。

* -globalopt: 全局变量优化器(Global Variable Optimizer)

此pass 转换从未使用其地址的简单全局变量。如果显然为真，它将读/写全局变量标记为常量，删除仅存储的变量等。

* -gvn: 全局值编号(Global Value Numbering)

此pass 执行全局值编号以消除完全和部分冗余指令。它还执行冗余负载消除。

* -indvars: 归纳变量规范化(Canonicalize Induction Variables)

这种转换分析归纳变量（以及从中得出的计算）并将其转换为适合后续分析和转换的更简单的形式。

此转换对具有可识别归纳变量的每个循环进行以下更改：

* 所有循环都被转换为具有单个规范归纳变量，该变量从零开始并逐步增加，步长为1。
* 规范归纳变量保证是循环头块中的第一个 PHI 节点。
* 任何指针算术递归要使用数组下标。(Any pointer arithmetic recurrences are raised to use array subscripts.)

如果循环的行程计数是可计算的，则此 pass 还会进行以下更改：

* 循环的退出条件被规范化以将归纳值与退出值进行比较。这会变成如下循环：

|  |
| --- |
| for (i = 7; i\*i < 1000; ++i)  变成了  for (i = 0; i != 25; ++i) |

* 从 indvar 派生的表达式在循环外的任何使用，都被更改为在循环外计算派生值，从而消除了对归纳变量的退出值的依赖。如果循环的唯一目的是计算某些派生表达式的退出值，则此转换将使循环死掉。

在执行了所有所需的循环转换后，应在该转换之后进行强度降低。此外，在有利可图的目标上，循环可以转换为倒计数到零（“do loop”优化）。

* -inline: 函数集成/内联

将函数自下而上内联到被调用者中。

* -instcombine: 合并冗余指令

组合指令以形成更少、更简单的指令。此 pass 不会修改 CFG。这个 pass 是代数简化发生的地方。

此 pass 结合了以下内容：

|  |
| --- |
| %Y = add i32 %X, 1  %Z = add i32 %Y, 1  变成了：  %Z = add i32 %X, 2 |

这是一个简单的工作列表驱动算法。

此传递保证对程序执行以下规范化：

* 1.如果二元运算符有常量操作数，则将其移至右侧。
* 2.具有常量操作数的位运算符总是分组，以便首先执行移位，然后ors，然后ands，然后xors。
* 3.如果可能，比较指令从<, >, ≤, or≥转换为 = or ≠。
* 4.所有基于布尔值的cmp指令都被替换为逻辑操作。
* 5.add X, X表示为mul X, 2 ⇒ shl X, 1
* 6.乘以一个恒定的二次幂参数被转换为移位。
* 7.… ETC。

此传递还可以简化对特定众所周知的函数调用（例如运行时库函数）的调用。例如，函数exit(3)内发生的main()调用可以简单地转换为 return 3 。 是否简化库调用由[-function-attrs pass](https://llvm.org/docs/Passes.html" \l "passes-function-attrs) 和 LLVM 对不同目标上的库调用的知识进行控制。

* -aggressive-instcombine: 组合表达模式

组合表达式模式以形成具有更少、简单指令的表达式。此pass 不会修改 CFG。

例如，此pass 在适用时将由 TruncInst 控制的表达式的宽度减小到更小的宽度。

它与 instcombine pass 的不同之处在于， 它包含需要比 O(1) 更高复杂度的模式优化，因此，它应该比 instcombine pass 运行更少的次数。

* -internalize: 内部化全局符号

此pass 遍历输入模块中的所有函数，寻找主函数。如果找到主函数，则所有其他函数和所有带有初始值设定项的全局变量都被标记为内部函数。

* -ipsccp: 过程间稀疏条件常数传播(Interprocedural Sparse Conditional Constant Propagation)

稀疏条件常数传播的过程间变体。

* -jump-threading: 跳转线程

跳转线程试图找到通过基本块运行的控制流的不同线程。此pass 查看具有多个前驱和多个后继的块。如果可以证明块的一个或多个前驱总是导致跳转到后继之一，我们通过复制该块的内容将边从前驱转发到后继。

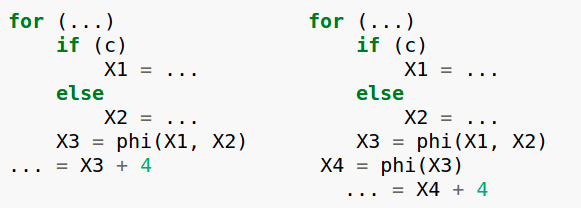
何时会发生这种情况的示例是这样的代码：

|  |
| --- |
| if () { ...  X = 4;  }  if (X < 3) { |

在这种情况下，第一个 if 末尾的无条件分支可以返回到第二个 if 的错误侧。

* -lcssa: 闭环 SSA 表单pass(Loop-Closed SSA Form Pass)

此pass 通过将 phi 节点放置在循环末尾，为跨循环边界的所有值转换循环。例如，它将左边的代码变成右边的代码：

这仍然这仍然是有效的 LLVM；额外的 phi 节点纯粹是多余的，将被InstCombine 消除掉。 这种转换的主要好处是它使许多其他循环优化（例如， LoopUnswitching）变得更简单。您可以 在 [LCSSA 表格的循环术语](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-lcssa)部分阅读更多内容。

* -licm: 循环不变代码移动(Loop Invariant Code Motion)

此 pass 执行循环不变代码移动，尝试从循环主体中删除尽可能多的代码。它通过将代码提升到 preheader 块，或者在安全的情况下将代码下沉到出口块来实现。此pass 还促进循环中必须别名的内存位置存在于寄存器中，从而提升和下沉“Invariant ”加载和存储。

循环外的提升操作是一种规范化转换。它启用并简化了中端的后续优化。提升指令的重新实现以减少寄存器压力，这是后端的责任，后端具有关于寄存器压力的更准确信息，并且还处理比 LICM 更多优化，例如，增加生命范围的其他优化。

此过程使用别名分析有两个目的：

* 1.移动循环不变加载和调用在循环之外。如果我们可以确定循环内的加载或调用永远不会对存储的任何内容进行别名，我们就可以像任何其他指令一样提升或下沉它。
* 2. 内存中的标量提升。如果循环内部有一条存储指令，我们会尝试将存储移动到循环之后而不是循环内部。这只有在满足以下几个条件时才会发生：
* 1.通过存储的指针是循环不变的。
* 2.循环中没有可能为指针起别名的存储或加载。循环中没有对指针进行修改/引用的调用。

如果这些条件成立，我们可以提升指针循环中的加载和存储以使用临时分配的变量。然后我们使用 mem2reg函数为变量构造为适当的 SSA 形式。

* -loop-deletion: 删除死循环

该文件实现了死循环删除 pass。此pass 负责消除具有非无限可计算行程计数的循环，这些循环没有副作用或易失性指令，并且不参与函数返回值的计算。

* -loop-extract: 将循环提取到新函数

围绕ExtractLoop()标量转换的pass包装器，用于将每个顶级循环提取到它自己的新函数中。如果循环是给定函数中的唯一循环，则不会被触及。这是通过错误点进行调试最有用的 pass 。

* -loop-extract-single: 将最多一个循环提取到一个新函数中

类似于[Extract loops into new functions](https://llvm.org/docs/Passes.html" \l "passes-loop-extract)，如果可以的话，这个过程从程序中提取一个自然循环到一个函数中。这是由bugpoint使用的。

* -loop-reduce: 循环强度降低

此 pass 对循环内的数组引用执行强度缩减，循环内的一个或多个组件是循环归纳变量。这是通过创建一个新值来保存第一次迭代的数组访问的初始值，然后在循环中创建一个新的 GEP 指令以将值递增适当的量来实现的。

* -loop-rotate: 旋转循环

一个简单的循环旋转变换。可以在[旋转循环的循环术语](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-loop-rotate)中找到它的摘要。

* -loop-simplify: 规范化自然循环(Canonicalize natural loops)

这个pass执行了几个转换，将自然循环转换成更简单的形式，这使得后续的分析和转换更简单、更有效。可以在[Loop Terminology, Loop Simplify Form](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-loop-simplify)中找到它的摘要。

循环头前插入保证从循环外部到循环头有一个单一的、非关键的入口边缘。这简化了一些分析和转换，例如LICM。

循环退出块插入保证了循环中的所有退出块（循环外部的块在循环内部有前驱）只有来自循环内部的前驱（因此由循环头控制）。这简化了 LICM 中内置的存储下沉等转换。

该pass还保证循环将恰好有一个 backedge。

请注意，[simplifycfg](https://llvm.org/docs/Passes.html" \l "passes-simplifycfg)pass 将清理被拆分但最终不再需要的块，因此使用此 pass 不应使生成悲观的代码。

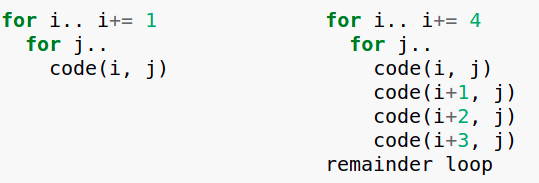
这个pass明显修改了CFG，但是更新了loop信息和dominator信息。

* -loop-unroll: 展开循环

这个过程实现了一个简单的循环展开器。当循环已被indvars pass 规范化时，它工作得最好，允许它轻松确定循环的行程计数。

* -loop-unroll-and-jam: 展开和堵塞循环

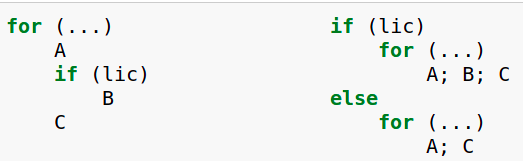
此通道实现了一个简单的展开和堵塞经典循环优化通道。它将循环的转换举例如下：



这可以看作是展开外环并将内环“干扰(jamming)”（融合）为一个。当可以在新的内部循环中共享变量或负载时，这可以显着提高性能。它使用 [依赖性分析](https://llvm.org/docs/Passes.html" \l "passes-da)来证明转换是安全的。

* -loop-unswitch: 取消循环切换

此过程将包含循环不变条件分支的循环转换为具有多个循环。例如，它将左边的代码变成右边的代码：



这会以指数方式增加代码的大小（每次循环未切换时都会加倍），因此我们仅在结果代码小于阈值时才取消切换。

此pass 期望LICM在它之前运行以将不变条件提升到循环之外，从而使取消切换的机会显而易见。

* -lower-global-dtors: 较低的全局析构函数(Lower global destructors)

此pass通过创建包装函数来 减少全局模块析构函数 (llvm.global\_dtors )，这些包装函数注册为llvm.global\_ctors 内的全局构造函数并且包含对注册其析构函数的\_\_cxa\_atexit调用。

* -loweratomic：将原子内在函数降低为非原子形式(Lower atomic intrinsics to non-atomic form)

此 pass 将原子内在函数降低为非原子形式，以便在已知的不可抢占环境中使用。

pass 不验证环境是不可抢占的（通常这需要了解程序的整个调用图，包括任何可能无法以bitcode形式提供的库）；它只是降低了每个atomic intrinsics。

* -lowerinvoke: Lower invokes to calls, for unwindless code generators

此转换旨在供尚不支持堆栈展开的代码生成器使用。此 pass 将invoke指令转换为 call指令，以便任何异常处理landingpad块都变成死代码（可以通过随后运行 -simplifycfg pass来删除）。

* -lowerswitch: Lower SwitchInsts to branches

用一系列分支重写 switch 指令，这允许目标在方便之前不执行 switch 指令。

* -mem2reg: Promote Memory to Register

该文件将内存引用提升为寄存器引用。它提倡只使用加载和存储的 alloca 指令。alloca通过使用支配边界来放置 phi 节点，然后按深度优先顺序遍历函数以适当地重写加载和存储来转换。这只是构建“修剪过的”SSA 形式的标准 SSA 构建算法。

* -memcpyopt: MemCpy 优化

此过程执行与消除memcpy 调用，或将mempys 转换为memsets 相关的各种转换。

* -mergefunc: 合并函数

此pass查找可合并的等效函数并将它们折叠。

在函数集合中引入了全序：我们定义比较来回答每两个函数中哪个更大。它允许将函数排列到二叉树中。

对于每个新函数，我们都会检查树中的等价物。

如果存在等效项，我们将折叠此类函数。如果两个函数都是可覆盖的，我们将函数移动到一个新的内部函数中，并为其留下两个可覆盖的 thunks。

如果没有等价物，那么我们将这个函数添加到树中。

查找例程的复杂度为 O(log(n))，而整个合并过程的复杂度为 O(n\*log(n))。

阅读 [本文](https://llvm.org/docs/MergeFunctions.html) 了解更多详情。

* -mergereturn: 统一函数出口节点

确保函数中最多有一条ret指令。此外，它还跟踪哪个节点是 CFG 的新出口节点。

* -partial-inliner: 部分内联

此 pass 执行部分内联，通常是内联围绕函数主体的if语句。

* -prune-eh：删除未使用的异常处理信息

该文件实现了一个简单的过程间 pass，它遍历调用图，当且仅当被调用者不能抛出异常时，将调用指令转换为调用指令。它通过调用图的自下而上遍历来实现。

* -reassociate: 重新关联表达式

此 pass 按照旨在促进更好的常量传播、GCSE、LICM、PRE 等的顺序重新关联交换表达式。

例如：4 + (x + 5) ⇒ x + (4 + 5)

在该算法的实现中，常量赋值rank = 0，函数参数赋值rank = 1，其他值赋值rank对应于当前函数的逆后序遍历（从2开始），有效地在深度循环中赋值比不在循环中的值更高的等级。

* -rel-lookup-table-converter: 相对查找表转换器(Relative lookup table converter)

此 pass 将查找表转换为 PIC 友好的相对查找表。

* -reg2mem：将所有值降级到堆栈槽

该文件将所有寄存器降级为内存引用。它旨在成为mem2reg的逆操作。通过转换为load 指令，跨基本块的值唯一有效期是phi 节点钱的alloca 指令和load 指令。这样做的目的是使 CFG 黑客攻击变得更加容易。为了使以后的黑客攻击更容易，入口块被分成两部分，这样所有引入的alloca 指令（没有其他指令）都在入口块中。

* -sroa: 聚合的标量替换(Scalar Replacement of Aggregates)

众所周知的聚合变换的标量替换。如果可能，此转换将alloca聚合类型（结构或数组）的指令分解为每个成员的单独alloca 指令。然后，如果可能的话，它将单个alloca指令转换为漂亮干净的标量 SSA 形式。

* -sccp: 稀疏条件常数传播(Sparse Conditional Constant Propagation)

稀疏条件常数传播和合并，可以概括为：

* 除非另有证明，否则假设值是恒定的
* 假设 BasicBlocks 是dead，除非另有证明
* 证明值是常数，并用常数替换它们
* 证明条件分支是无条件的

请注意，此pass 有使定义失效的习惯。运行此pass 之后的某个时间运行DCE pass是个好主意。

* -simplifycfg: 简化 CFG

执行死代码消除和基本块合并。具体来说：

* 删除没有前导的基本块。
* 如果只有一个基本块且前驱只有一个后继，则将基本块合并到其前驱中。
* 消除具有单个前身的基本块的 PHI 节点。
* 消除仅包含无条件分支的基本块。
* -sink: 代码下沉

如果可能，此遍将指令移动到后续块中，以便它们不会在不需要其结果的路径上执行。

* -strip: 去除模块中的所有符号

执行代码剥离。此转换可以删除：

* 虚拟寄存器的名称
* 内部全局变量和函数的符号
* 调试信息

请注意，此转换会大大降低代码的可读性，因此它只应在使用 strip 实用程序的情况下使用，例如，减少代码大小或使逆向工程代码更难。

* -strip-dead-debug-info: 去除未使用符号的调试信息

执行代码剥离。此转换可以删除：

* 虚拟寄存器的名称
* 内部全局变量和函数的符号
* 调试信息

请注意，这种转换会大大降低代码的可读性，因此它只应在使用 strip 实用程序的情况下使用，例如减少代码大小或使逆向工程代码更难。

* -strip-dead-prototypes: 剥离未使用的函数原型

此 pass 遍历输入模块中的所有函数，查找无效声明并将其删除。死声明是没有可用实现的函数声明（即未使用的库函数的声明）。

* -strip-debug-declare: 剥离所有llvm.dbg.declare内在函数

这个pass实现代码剥离。具体可以删除：

* 1.虚拟寄存器的名称
* 2.内部全局变量和函数的符号
* 3.调试信息

请注意，此转换会大大降低代码的可读性，因此它只能用于需要使用“strip”实用程序的情况，例如减少代码大小或使逆向工程代码更难。

* -strip-nondebug: 从模块中去除所有符号，除了 dbg 符号

这个pass实现代码剥离。具体可以删除：

* 1.虚拟寄存器的名称
* 2.内部全局变量和函数的符号
* 3.调试信息

请注意，此转换会大大降低代码的可读性，因此它只能用于需要使用“strip”实用程序的情况，例如减少代码大小或使逆向工程代码更难。

* -tailcallelim: 尾调用消除

该文件转换当前函数的调用（自递归），然后是带有分支的返回指令到函数的入口，从而创建一个循环。此通道还实现了对基本算法的以下扩展：

* 1.调用和返回之间的琐碎指令不会阻止转换的发生，但目前的分析无法支持移动任何真正有用的指令（只有死指令）。
* 2.此过程将通过关联表达式阻止尾递归的函数转换为使用累加器变量，从而将典型的朴素阶乘或 fib 实现编译为高效代码。
* 3.如果函数返回 void，或者如果返回的返回值是调用者返回的结果，或者如果在函数的所有退出时返回运行时常量，则会执行TRE。有可能，尽管不太可能，return 返回其他东西（比如常量 0），并且仍然可以被 TRE。如果函数中的所有其他返回指令返回完全相同的值，则它可以是 TRE 。
* 4.如果它可以证明被调用者没有访问他们的调用者堆栈帧，他们被标记为符合尾调用消除条件（由代码生成器）。

## Utility Passes

本节介绍 LLVM Utility Passes。

* -deadarghaX0r: Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT USE)

与死参数消除相同，但删除外部函数的参数。这仅供[bugpoint](https://llvm.org/docs/Bugpoint.html)使用。

* -extract-blocks：从模块中提取基本块（用于错误点）

bugpoint 使用此pass将模块中的所有块提取到它们自己的函数中。

* -instnamer: 为匿名指令分配名称

这是一个提供指令名称的小实用程序pass，这在比较优化效果时非常有用，因为删除未命名的指令会更改所有其他指令编号，从而使差异非常嘈杂。

* -verify: 模块验证器

验证 LLVM IR 代码。这对于在正在进行测试的优化之后运行很有用。请注意，llvm-as 在发出bitcode之前会验证其输入，而且格式错误的bitcode可能会导致 LLVM 崩溃。因此，鼓励所有语言前端在执行优化转换之前验证其输出。

* 1.二元运算符的两个参数属于同一类型。
* 2.验证内存访问指令的索引是否与其他操作数匹配。
* 3.验证算术和其他事情只在第一类类型上执行。验证移位和逻辑运算仅发生在整数上，等等。
* 4.switch 语句中的所有常量都是正确的类型。
* 5.该代码采用有效的 SSA 形式。
* 6.将标签放入任何其他类型（如结构）或返回标签是非法的。
* 7.只有 phi 节点可以自引用： %x = add i32 %x, %x is invalid.
* 8.PHI 节点必须每个前任都有一个条目，没有额外的。
* 9.PHI 节点必须是基本块中的第一件事，所有这些都组合在一起。
* 10.PHI 节点必须至少有一个条目。
* 11.所有基本块都应该只以终止符 ins 结尾，而不是包含它们。
* 12.函数的入口节点不能有前置节点。
* 13.所有指令都必须嵌入到一个基本块中。
* 14.函数不能采用 void 类型的参数。
* 15.验证函数的参数列表是否与其声明的类型一致。
* 16.为 void 值指定名称是非法的。
* 17.没有初始化器的内部全局值是非法的。
* 18. 返回值与函数返回值类型不一致的ret指令是非法的。
* 19.函数调用参数类型与函数原型匹配。
* 20.所有其他由断言测试的东西都散布在代码中。

请注意，这不提供完整的安全验证（如 Java），而只是试图确保代码格式正确。

* -view-cfg: 查看函数的CFG

使用 GraphViz 工具显示控制流图。此外，该-cfg-func-name=<substring>选项可用于过滤显示的函数。将显示包含指定子字符串的所有函数。

* -view-cfg-only: 查看函数的 CFG（没有函数体）

使用 GraphViz 工具显示控制流图，但省略函数体。此外，该-cfg-func-name=<substring>选项可用于过滤显示的函数。将显示包含指定子字符串的所有函数。

* -view-dom: 查看函数的支配树

使用 GraphViz 工具显示支配树。

* -view-dom-only: 查看函数的支配树（没有函数体）

使用 GraphViz 工具显示支配树，但省略函数体。

* -view-postdom: 查看函数的后支配树

使用 GraphViz 工具显示后支配树。

* -view-postdom-only: 查看函数的后支配树（没有函数体）

使用 GraphViz 工具显示后支配树，但省略函数体。

* -transform-warning: 报告遗漏的强制转换

发出有关尚未应用强制转换的警告（例如，来自#pragma omp simd）

# 堆栈安全分析

## 简介

Stack Safety Analysis 确定堆栈分配的变量是否可以被认为是“安全的”，不受内存访问错误的影响。

分析的主要目的是供消毒剂(sanitizers)使用，以避免对“安全”变量进行不必要的检测。SafeStack 将成为第一个用户。

“安全”变量可以定义为不能超出范围使用（例如返回后使用）或越界访问的变量。将来可以扩展它以跟踪其他变量属性。例如，我们计划通过检查来扩展实现，以确保在每次读取之前始终是已经初始化的变量，以优化未初始化内存检查的使用。

## 它是如何工作

分析分两个阶段进行：

* 过程内或“本地”阶段，在函数内部执行深度优先搜索，以收集每个 alloca 的所有使用，包括加载/存储和用作参数函数。在这个阶段之后，我们知道 alloca 的哪些部分被函数本身使用，但我们不知道在它作为参数传递给另一个函数后会发生什么。
* 过程间或“全局”阶段解决分配作为函数参数传递后发生的情况。此阶段对单个模块内的函数调用执行深度优先搜索，并通过函数调用传播 allocas 使用情况。

当与 ThinLTO 一起使用时，全局阶段对模块摘要索引执行整个程序分析。

## 测试

分析包含 lit 测试。

我们希望用户可以容忍将变量错误分类为“不安全”，而实际上它是“安全”的。这可能会导致代码效率低下。但是，我们不能接受错误的“安全”分类，这可能会导致消毒剂(sanitizers)错过检测代码中的实际错误。为避免这种情况，我们需要额外的验证工具。

AddressSanitizer 可能有助于此验证。我们可以像往常一样检测所有变量，但另外将堆栈安全信息存储在 ASanStackVariableDescription. 然后，如果 AddressSanitizer 检测到“安全”变量上的错误，我们可以生成额外的报告，让用户知道堆栈安全分析可能失败了，我们应该检查编译器中的错误。

# MergeFunctions pass，它是如何工作的

## 简介

有时代码包含相等的函数，或做完全相同的事情的函数，即使它们在 IR 级别上不相等（例如：乘法 2 和 'shl 1'）。这可能由于多种原因而发生：主要是模板和自动代码生成器的使用。虽然，有时用户自己可能会写同样的东西两次 :-)

这个pass的主要目的是识别这样的函数并合并它们。

本文档是pass comments的扩展，描述了pass 逻辑。它描述了用于比较函数的算法，并解释了我们如何正确组合相等的函数以保持模块有效。

材料以自上而下的形式提供，因此读者可以从高层级思想开始学习 pass，并以低层级算法细节结束，从而为阅读源代码做好准备。

主要目标是描述这里的算法和逻辑以及概念。如果你不想阅读源代码，但想了解pass算法，那么这篇文档对你很有帮助。作者尽量不重复源代码，只涵盖常见情况，以避免在任何微小的代码更改后需要更新此文档的情况。

### 我应该知道什么才能跟上这份文件？

读者应该熟悉常见的编译工程原则和 LLVM 代码基础知识。在本文中，我们假设读者熟悉 [Single Static Assignment](http://en.wikipedia.org/wiki/Static_single_assignment_form)概念并且了解 IR 结构。

我们将使用“[module](https://llvm.org/docs/LangRef.html" \l "high-level-structure)”, “[function](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-class)”, “[basic block](http://en.wikipedia.org/wiki/Basic_block)”, “[user](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-class)”, “[value](https://llvm.org/docs/ProgrammersManual.html" \l "the-value-class)”, “[instruction](https://llvm.org/docs/ProgrammersManual.html" \l "the-instruction-class)”等术语。

作为一个很好的起点，可以使用 Kaleidoscope 教程：

[LLVM 教程：目录](https://llvm.org/docs/tutorial/index.html)

理解教程的第 3 章尤为重要：

[万花筒教程](https://llvm.org/docs/tutorial/LangImpl03.html)

读者还应该知道通道在 LLVM 中是如何工作的。他们可以将本文用作参考和起点：

[编写 LLVM Pass](https://llvm.org/docs/WritingAnLLVMPass.html)

还有什么？好吧，也许读者也应该有一些 LLVM 通过调试和错误修复的经验。

### 本文结构

文章由三部分组成。第一部分解释了顶层的传递函数。第二部分描述了比较过程本身。第三部分描述合并过程。

在每一部分中，作者都试图将内容以自上而下的形式呈现。将首先描述顶层方法，然后在最后，在每个部分的尾部描述终端方法。如果读者看到对尚未描述的方法的引用，他们会在下面找到它的描述。

## 基础知识

怎么做？

我们需要合并函数吗？显而易见的答案是：是的，这是很有可能的情况。我们通常确实有重复项，最好摆脱它们。但是我们如何检测重复项呢？这个想法是：我们将功能拆分成更小的砖块或部分，然后比较“砖块”的数量。如果相等，我们比较“砖块”本身，然后对函数本身做出结论。

有什么区别？例如，在一台有 64 位指针的机器上（假设我们只有一个地址空间），一个函数存储一个 64 位整数，而另一个函数存储一个指针。如果目标是上面提到的机器，并且如果函数除了参数类型（我们可以认为它是函数类型的一部分）之外是相同的，那么我们可以将 uint64\_t 和 void\*视为相等。

这只是一个例子；下面描述了更多可能的细节。

再举个例子，读者可以想象出另外两个函数。第一个函数执行乘以 2，而第二个函数执行逻辑左移 1。

### 可能的解决方案

让我们简要考虑一下可能的选项，即我们必须如何实现以及必须实现什么才能创建功能齐全的函数合并，以及它对我们意味着什么。

等价函数检测显然假设要实现一个“detector”方法，后者应该回答“函数是否相等”的问题。这种“detector”方法由微小的“sub-detectors组成，每个sub-detectors都回答完全相同的问题，但针对的是功能部分。

作为第二步，我们应该合并相等的函数。所以它应该是一种“merger”的方法。“merger”接受两个函数F1和F2，并产生F1F2 函数，即合并的结果。

有了这样的例程，我们就可以处理整个模块，并合并所有相等的函数。

在这种情况下，我们必须将每个函数与另一个函数进行比较。读者可能会注意到，这种方式似乎相当昂贵。当然我们可以引入散列和其他帮助器(helpers)，但它仍然只是一种优化，因此复杂度为 O(N\*N) 级别。

我们能达到另一个层次吗？我们可以引入对数搜索或随机访问查找吗？答案是：“是的”。

#### 随机存取

这怎么可能呢？只需将每个函数转换为一个数字，然后将它们全部收集到一个特殊的哈希表中。具有相等哈希值的函数是相等的。好的散列意味着必须考虑每个功能部分。这意味着我们必须将每个函数部分转换为一些数字，然后将其添加到散列中。查找时间会很短，但由于散列例程，这种方法会增加一些延迟。

#### 对数搜索

我们可以在函数集之间引入总排序，一旦排序，我们就可以实现对数搜索。查找时间仍然取决于 N，但会增加一点延迟 ( log(N) )。

#### 当前状态

这两种方法（随机访问和对数）都已经实施和测试，并且都提供了很好的改进。最令人惊讶的是对数搜索速度更快；有时高达 15%。哈希方法需要一些额外的 CPU 时间，这是它工作速度较慢的主要原因；在大多数情况下，总的“散列”时间大于总的“对数搜索”时间。

因此，优先考虑“对数搜索”。

尽管在需要的情况下，对数搜索（读作“总排序”）可以用作我们实现随机访问的里程碑。

每次比较都基于数字或标志比较。在随机访问方法中，我们可以使用相同的比较算法。比较时，一旦发现差异就退出，但这里我们可能每次都要扫描整个函数体（注意，可能会更慢）。就像在“总排序”中一样，我们将跟踪每个数字和标志，但我们应该获取数字序列，然后创建哈希数字，而不是进行比较。因此，再一次，总排序可以被视为更快（理论上）的随机访问方法的里程碑。

### MergeFunctions、主要字段和 runOnModule

类中有两个主要的重要字段：

* FnTree – 所有唯一函数(function)的集合。它保留无法相互合并的项目。它被定义为：

std::set<FunctionNode> FnTree;

这FunctionNode是一个llvm::Function类的包装器，在函数集中实现了“<”运算符（下面我们将解释它的具体工作原理；这是快速函数比较的关键点）。

* Deferred– 合并过程会影响FnTree中已经存在的函数体。显然，应该再次重新检查此类函数。在这种情况下，我们将它们从 FnTree 中移除，并将它们标记为重新扫描，即放入Deferred列表中。

#### runOnModule

该算法非常简单：

* 1.将所有模块的函数放入worklist。
* 2.扫描worklist的函数两次：先枚举强函数，再枚举弱函数：
* 2.1. 循环体：从工作列表中取出一个函数 （称为FCur）并尝试将其插入FnTree：检查FCur是否等于 FnTree 中的函数之一。如果 FnTree 中有一个相等的函数 （称之为FExists）：将函数FCur与FExists合并。否则将该函数从工作列表添加到FnTree。
* 3.工作列表扫描和合并操作完成后，检查 Deferred 列表。如果不为空：用Deferred list重新填充worklist 的内容， 重做步骤2，如果Deferred list为空，则退出方法。

#### 比较和对数搜索

让我们回顾一下我们的任务：对于模块M中的每个函数F，我们必须在尽可能短的时间内找到相等的函数F`，并将它们合并为一个函数。

定义函数集之间的总排序允许我们将函数组织成一个二叉树。在这种情况下，查找过程的复杂度估计为 O(log(N))。但是我们如何定义全序(total-ordering)呢？

我们必须引入适用于每一对函数的单一规则，并遵循此规则，然后评估它们中的哪个更大。会是什么样的规则？让我们将其声明为“compare”方法，该方法返回 如下3 个可能值之一：

* -1，左边小于右边，
* 0，左右相等，
* 1、左大于右。

当然，这意味着我们必须维护 严格和非严格的顺序关系属性：

* 自反性 ( a <= a,a == a , a >= a),
* 反对称（if a <= b and b <= a then a == b），
* 传递性（a <= b and b <= c, then a <= c），
* 不对称性（if a < b, then a > b or a == b）。

如前所述，比较例程由“sub-comparison-routines”组成，每个比较例程又由“sub-comparison-routines”组成，依此类推。最后，它以原始比较结束。

下面，我们将使用以下操作：

* 1.cmpNumbers(number1, number2)是一种方法，如果左小于右，则返回 -1；如果左右相等，返回0；否则为 1。
* 2.cmpFlags(flag1, flag2)是比较两个标志的假设方法。逻辑与cmpNumbers 中的相同，其中true是 1， false是 0。

本文的其余部分基于MergeFunctions.cpp源代码（位于<llvm\_dir>/lib/Transforms/IPO/MergeFunctions.cpp 中）。我们希望读者保持打开此文件，以便我们可以将其作为进一步解释的参考。

现在，我们准备进入下一章，看看它是如何工作的。

## 函数比较

首先，让我们定义我们比较复杂对象的精确程度。

复杂对象比较（函数、基本块等）主要基于其子对象比较结果。它类似于下一个“tree”对象比较：

* 1.对于两棵树T1和T2，我们执行深度优先遍历，并有两个序列作为产品：“ T1Items ”和“ T2Items ”。
* 2.然后我们以 most-significant-item-first 顺序比较“ T1Items ”和“ T2Items ”链。items的比较结果将是T1和T2本身比较的结果。

### FunctionComparator::compare(void)

简单看一下源代码就会知道，比较是从“int FunctionComparator::compare(void) ”方法开始的。

* 1. 首先要比较的部分是函数的属性和一些在“attributes”术语之外的特性，但仍然可以在不改变函数体的情况下使函数有所不同。这部分比较通常在简单的cmpNumbers或cmpFlags操作中完成（例如，cmpFlags(F1->hasGC(), F2->hasGC()) ）。下面是在此阶段要比较的函数属性的完整列表：
* 属性（那些由Function::getAttributes() 方法返回）。
* GC，为了等价，RHS和LHS应该都没有 GC或有相同的 GC。
* Section，就像GC一样：RHS和LHS应该在同一部分中定义。
* 可变参数。LHS和RHS应该都带有或不带有var-args。
* 调用约定应该相同。
* 2.函数类型。通过 FunctionComparator::cmpType(Type\*, Type\*) 方法检查。它检查返回类型和参数类型；该方法本身将在后面描述。
* 3. 将函数形参相互关联起来。然后比较函数体，如果我们在 LHS 的函数体中看到 LHS 的第 i 个参数的用法，那么，我们希望在 RHS 的函数体中的相同位置看到RHS的第 i 个参数的用法，否则函数不同。在这个阶段，我们将优先权授予我们后来在函数体中遇到的那些（我们前面遇到的优先级的值会更小）。这是通过“ 。FunctionComparator::cmpValues(const Value\*, const Value\*)”方法完成的（将在稍后描述）
* 4.函数体比较。正如方法注释中所写：

“我们进行 CFG-ordered 遍历，因为链表中块的实际排序并不重要。我们的遍历从两个函数的入口块开始，然后按顺序从每个终止符中取出每个块。作为一个artifact，这也意味着无法到达的块将被忽略。”

所以，使用这个 walk 我们以相同的顺序从左和右得到 BBs ，并通过“ FunctionComparator::compare(const BasicBlock\*, const BasicBlock\*)”方法比较它们。

我们还将 BBs 相互关联，就像我们使用函数形式参数那样（参见下面cmpValues的方法）。

### FunctionComparator::cmpType

考虑类型比较的工作原理。

* 1. 将指针强制指向整数。如果左类型是指针，尝试将其强制转换为整数类型。如果其地址空间为 0，或者地址空间被完全忽略，则可以完成。对右类型做同样的事情。
* 2.如果左右类型相等，则返回0，否则我们需要优先选择其中一个。然后，继续下一步。
* 3. 如果类型属于不同类型（不同的类型 IDs）。返回类型 IDs 比较的结果，将它们视为数字来处理（使用cmpNumbers操作）。
* 4. 如果类型是向量或整数，则返回它们的指针比较结果，将它们作为数字进行比较。
* 5. 检查类型 ID 是否属于下一个组（称为equivalent-group）：
* Void
* Float
* Double
* X86\_FP80
* FP128
* PPC\_FP128
* Label
* Metadata.

如果ID属于上面的组，则返回0。因为看到类型有相同TypeID就足够了。不需要其他信息。

* 6. 左右类型是指针。返回地址空间比较（数字比较）的结果。
* 7. 复杂类型（结构、数组等）。遵循复杂对象比较技术（参见本章第一段）。left和 right都将被展开，它们的元素类型将以相同的方式检查。如果我们在某个阶段得到 -1 或 1，则返回它。否则返回 0。
* 8. Steps 1-6 描述了所有可能的情况，如果我们通过了step 1-6 ，却没有得到任何结论，那么调用llvm\_unreachable，因为这是一个非常不可预料的情况。

### cmpValues(const Value\*, const Value\*)

比较本地值的方法。

这种方法给了我们一个非常奇怪的问题的答案：我们是否可以将局部值视为相等，否则哪个值更大。最好从示例开始：

考虑当我们在左函数“ FL ”和右函数“ FR ”中查看相同位置时的情况。左边的每个部分都等于右边的相应部分，并且（！）两个部分都使用 Value实例，例如：

|  |
| --- |
| instr0 i32 %LV ; left side, function FL  instr0 i32 %RV ; right side, function FR |

所以，现在我们的结论取决于Value实例比较。

该方法的主要目的是确定这些值之间的关系。

我们期望从等价函数中得到什么？在同一个地方，在函数“ FL ”和“ FR ”中，我们希望看到相等的值，或者在“ FL ”和“ FR ”的相同位置定义的值。

考虑一个小例子：

左边是

|  |
| --- |
| define void %f(i32 %pf0, i32 %pf1) {  instr0 i32 %pf0 instr1 i32 %pf1 instr2 i32 123  } |

右边是

|  |
| --- |
| define void %g(i32 %pg0, i32 %pg1) {  instr0 i32 %pg0 instr1 i32 %pg0 instr2 i32 123  } |

在这个例子中，pf0与pg0相关联，pf1与pg1相关联 ，我们还声明pf0 < pf1，因此，pg0 < pf1。

带有操作码“ instr0 ”的指令是相等的，因为它们的类型和操作码是相等的，并且值是关联的。

来自 f 的带有操作码“ instr1 ”的指令大于来自 g 的带有操作码“ instr1 ”的指令；这里我们有相同的类型和操作码，但是“ pf1 ”大于“ pg0 ”。

操作码为“ instr2 ”的指令是相等的，因为它们的操作码和类型是相等的，并且使用相同的常量作为值。

#### 我们在 cmpValues 中关联什么？

* 函数参数。来自左函数的 第i个参数与来自右函数的第i个参数相关联。
* BasicBlock 实例。在基本块枚举循环中，我们将左侧函数中的第i个 BasicBlock 与右侧函数中的第i个 BasicBlock 相关联。
* 指令。
* 指令操作数。请注意，我们可以在这里遇到以前从未见过的Value 。在这种情况下，它不是函数参数，也不是BasicBlock或 Instruction。这是一个全局值。它是一个常量，因为它是这里唯一能假定为global。该方法还比较： 相同类型的常量，如果右常量可以无损的bitcast为左常量，那么我们也比较它们。

#### 如何实现cmpValues？

Association is a case of equality for us。我们只是将这些值视为相等，但一般来说，我们需要实现反对称关系。如上所述，要了解什么是less，我们可以使用我们遇到的values的顺序。如果一个函数中的两个values具有相同的顺序（同时满足），那么我们将values视为关联的。否则——这取决于谁是第一个。

每次运行顶层比较方法时，我们都会初始化两个相同的映射（一个用于左侧，另一个用于右侧）：

|  |
| --- |
| map<Value, int> sn\_mapL, sn\_mapR; |

map的key是Value本身，value- 是它的顺序（称之为 序列号）。

要添加V，我们需要执行下一个过程：

|  |
| --- |
| sn\_map.insert(std::make\_pair(V, sn\_map.size())); |

对于第一个Value，map 将返回0，对于第二个Value map 将返回1，依此类推。

然后我们可以通过简单的比较来检查左右值是否同时满足：

|  |
| --- |
| cmpNumbers(sn\_mapL[Left], sn\_mapR[Right]); |

当然，我们可以结合插入和比较：

|  |
| --- |
| std::pair<iterator, bool>  LeftRes = sn\_mapL.insert(std::make\_pair(Left, sn\_mapL.size())), RightRes  = sn\_mapR.insert(std::make\_pair(Right, sn\_mapR.size()));  return cmpNumbers(LeftRes.first->second, RightRes.first->second); |

让我们看看整个方法是如何实现的。

* 1. 我们必须从坏消息开始。考虑函数自身和交叉引用的情况：

|  |
| --- |
| // self-reference  unsigned fact0(unsigned n) { return n > 1 ? n \* fact0(n-1) : 1; }  unsigned fact1(unsigned n) { return n > 1 ? n \* fact1(n-1) : 1; }  // cross-reference  unsigned ping(unsigned n) { return n!= 0 ? pong(n-1) : 0;}  unsigned pong(unsigned n) { return n!= 0 ? ping(n-1) : 0; } |

此比较已在初始MergeFunctions pass 版本中实现。但是，不幸的是，它不是可传递的。这是我们无法转换为 less-equal-greater 比较的唯一case。这是很少见的情况，10000 个函数有 4-5 个（已在测试套件中检查），我们希望读者能原谅我们为了获得 O(log(N)) pass 时间而做出的这种牺牲。

* 2. 如果 left/right Value是一个常量，我们必须比较它们。如果是相同的常量则返回 0，否则使用cmpConstants 方法。
* 3. 如果 left/right 是InlineAsm实例。返回值指针(Value pointers )比较的结果。
* 4. L（左值）和R （右值）的显式关联。我们需要找出值是否同时满足，从而相关联。或者我们需要制定规则：当我们处理L < R时。现在很简单：我们只返回数字比较的结果：

|  |
| --- |
| std::pair<iterator, bool>  LeftRes = sn\_mapL.insert(std::make\_pair(Left, sn\_mapL.size())),  RightRes = sn\_mapR.insert(std::make\_pair(Right, sn\_mapR.size()));  if (LeftRes.first->second == RightRes.first->second) return 0;  if (LeftRes.first->second < RightRes.first->second) return -1;  return 1; |

现在当cmpValues返回 0 时，我们可以继续进行比较过程。否则，如果我们得到（-1 或 1），我们需要将这个结果传递给顶层，并完成比较过程。

### cmpConstants

执行常量比较如下：

* 1. 使用cmpType方法比较常量类型。如果结果为 -1 或 1，则转到第 2 步，否则继续第 3 步。
* 2. 如果类型不同，我们仍然可以检查常量是否可以无损地相互转换。进一步的解释是 canLosslesslyBitCastTo方法的修改。
* 2.1 检查常量是否为第一类类型（isFirstClassTypecheck）：
* 2.1.1. 如果两个常量都不是第一类类型：返回结果cmpType。
* 2.1.2. 否则，如果左类型不是第一类，则返回 -1。如果正确的类型不是第一类，则返回 1。
* 2.1.3. 如果两种类型都是第一类类型，则继续执行下一步（2.1.3.1）。
* 2.1.3.1. 如果类型是向量，则使用 cmpNumbers比较它们的位宽。如果结果不为 0，则返回它。
* 2.1.3.2。不同的类型，但不是向量：
* 如果它们都是指针，对我们有好处，我们可以继续执行第 3 步。
* 如果其中一种类型是指针，则返回isPointer标志比较的结果（cmpFlags操作）。
* 否则我们没有方法来证明 是否 bitcastability ，因此返回类型比较的结果（-1 或 1）。

以下步骤适用于类型相等或常量可bitcast的情况：

* 3. 常量之一是“null”值。返回cmpFlags(L->isNullValue, R->isNullValue) 比较结果。
* 4.比较值IDs，不为0则返回结果：

|  |
| --- |
| if (int Res = cmpNumbers(L->getValueID(), R->getValueID()))  return Res; |

* 5.比较常量的内容。比较取决于常量的种类，但在这个阶段它只是字典序比较。看看“Functions comparison”段落开头是如何描述的。从数学上讲，它等同于下一种情况：我们对左常量和右常量进行编码（与bitcode-writer 的方式类似）。然后比较左码序列和右码序列。

### compare(const BasicBlock\*, const BasicBlock\*)

比较两个BasicBlock实例。

它枚举来自左BB和右BB的指令。

* 1. 为左右指令分配序号，使用 cmpValues方法。
* 2. 如果 left 或 right 之一是GEP ( GetElementPtr)，则将GEP视为大于其他指令。如果两条指令都是GEPs ，则使用cmpGEP 方法进行比较。如果结果为 -1 或 1，则将其传递给顶级比较（返回结果）。
* 3.
* 3.1. 比较操作。调用cmpOperation方法。如果结果为 -1 或 1，则返回它。
* 3.2. 比较操作数个数，结果为-1或1则返回。
* 3.3. 比较操作数本身，使用cmpValues方法。如果结果为 -1 或 1，则返回结果。
* 3.4. cmpType使用方法比较操作数的类型。如果结果为 -1 或 1，则返回结果。
* 3.5. 继续执行下一条指令。
* 4. 我们可以分3种情况完成指令枚举：
* 4.1. 我们到达了左右基本块的末端。我们没有在第 1-3 步退出，所以内容相等，返回 0。
* 4.2. 我们已经到达左侧基本块的末尾。返回-1。
* 4.3. Return 1（我们到达了右侧基本块的末尾）。

### cmpGEP

比较两个 GEP（getelementptr指令）。

它与常规操作比较的唯一不同之处在于：使用accumulateConstantOffset方法的可能性。

因此，如果我们得到左右GEP的常量偏移量，则将其作为数字进行比较，并返回比较结果。

否则将其视为常规操作（请参阅上一段）。

### cmpOperation

比较指令操作码和一些重要的操作属性。

* 1.比较操作码，如果不同则返回结果。
* 2.比较操作数的数量。如果不同——返回结果。
* 3.比较操作类型，使用cmpType。都一样——如果类型不同，则返回结果。
* 4.比较subclassOptionalData，用getRawSubclassOptionalData 方法获取，像数字一样比较。
* 5.比较操作数类型。
* 6. 对于一些特定的指令，检查一些重要属性的等价性（在我们的例子中是关系）。例如，我们必须比较 load指令的对齐方式。

### O(log(N))

上述方法实现顺序关系。后者可用于二叉树中的节点比较。所以我们可以将函数集组织到二叉树中，并将查找过程的成本从 O(N\*N) 降低到 O(log(N))。

## 合并过程，mergeTwoFunctions

一旦MergeFunctions检测到当前函数 ( G ) 等于之前分析过的函数（函数F），它就会调用mergeTwoFunctions(Function\*, Function\*)

操作以如下方式影响FnTree的内容： F 将留在 FnTree， G等于F将不会添加到FnTree。对G的调用 将被替换为其他内容。它改变了调用者的主体部分。因此，调用G的函数将被放入Deferred set 中， 并从 FnTree 中删除，然后再次分析。

接下来的方法是：

* 1.最希望的情况：当我们可以使用别名并且F和G都是weak时。我们使它们都具有第三个强函数H的别名。实际上H 是F。请看下面它是如何制作的（但最好直接查看源代码）。好吧，这是我们可以在任何地方用F替换G的情况 ，我们在这里使用replaceAllUsesWith 操作（RAUW）。
* 2. F 不能被覆盖，而G 可以。做下一步就好了： after merging the places where overridable function were used, still use overridable stub。So try to make G alias to F, or create overridable tail call wrapper around F and replace G with that call。
* 3. F和G都不能被overridden。我们不能使用RAUW。我们可以只更改调用者：调用F而不是G。就是 replaceDirectCallers这样。

下面是详细的主体描述。

**如果“F”可能被overridden**

mayBeOverridden的注释如下：“这个全局的定义是否可以在链接时被不等价的东西替换”。如果是这样，那没关系：我们可以使用F的别名替换G或更改调用指令本身。

### HasGlobalAliases, removeUsers

首先考虑当我们有一个函数名到另一个函数名的全局别名时的情况。我们的目的是使它们都具有第三强函数的别名。这样，如果我们让 F 保持活动状态并且不进行重大更改，我们可以将其保留在FnTree。尝试将这两个目标结合起来。

使用 F 的别名对 F 本身进行存根替换( stub replacement )。

* 1. 创建存根函数(stub function )H，具有与函数 F相同的名称和属性。它需要F和G的最大对齐。
* 2. 将所有函数F的使用替换为函数H的使用。它是两步处理完成。首先，我们必须考虑到，调用F的所有函数都将更改：因为我们更改了调用参数（从F到H）。如果是这样，我们必须在此过程后再次检查这些调用函数。FnTree我们从带有名称的方法removeUsers(F) 中 删除调用者（不要与 replaceAllUsesWith 混淆）：
* 2.1. Inside removeUsers(Value\* V)， 我们遍历所有使用值V（或我们上下文中的 F ）的值。如果 value 是指令，我们转到保存该指令的函数并将其标记为 to-be-analyzed-again （把它放入 Deferred set），我们还从FnTree中删除调用者。
* 2.2. 现在我们可以进行替换：调用F->replaceAllUsesWith(H)。
* 3.  H（现在“正式”扮演 F 的角色）被替换为F的别名。对G执行相同操作：将其替换为F的别名。所以最终在所有使用F 的地方，我们使用H并且它是F的别名，并且在所有使用G的地方我们也有F的别名。
* 4 .将F链接设置为私有。让它强大:-)

### 没有全局别名，replaceDirectCallers

如果不支持全局别名。我们调用 replaceDirectCallers。只需遍历G的所有调用并将其替换为F的调用。如果您查看该方法，您会发现它也会扫描G的所有使用，并且如果 use 是被调用方（如果用户是调用指令并且G用作要调用的内容），我们将其替换为F的使用。

#### If “F” could not be overridden, fix it！

我们调用 writeThunkOrAlias(Function \*F, Function \*G)。在这里，我们首先尝试将 G替换为F的别名。接下来的条件是必不可少的：

* 目标应该支持全局别名，
* G的地址本身 应该不重要，没有命名，也没有在任何地方被引用，
* 函数应该与外部、本地或弱链接一起出现。

否则我们写 thunk: some wrapper that has G's interface and calls F，所以G可以用这个包装器替换。

**writeAlias**

如下来自llvm参考：“别名充当别名值的第二个名称”。所以我们只想为F创建第二个名称并使用它代替G：

* 1. 自己创建全局别名（GA），
* 2.调整F的对齐方式，使其必须是当前和G 的对齐方式中的最大值；
* 3.替换G的使用：
* 3.1. 首先将G的所有调用者标记为要 to-be-analyzed-again，使用 removeUsers方法（参见上一章），
* 3.2. 调用G->replaceAllUsesWith(GA)。
* 4.Get rid of G.

**writeThunk**

正如方法注释中所写：“Replace G with a simple tail call to bitcast(F). Also replace direct uses of G with bitcast(F). Deletes G.”。

一般来说，当我们想要替换被调用者时，它与往常一样，除了第一点：

* 1. We generate tail call wrapper around F, but with interface that allows use it instead of G.
* 2. “As-usual”: removeUsers and replaceAllUsesWith then.
* 3. Get rid of G.

# LLVM 别名分析基础架构

## 简介

别名分析（又名指针分析）是一类技术，它试图确定两个指针是否可以指向内存中的同一个对象。有许多不同的别名分析算法和许多不同的分类方法：流敏感与流不敏感、上下文敏感与上下文不敏感、字段敏感与字段不敏感、基于统一与基于子集等(flow-sensitive vs. flow-insensitive, context-sensitive vs. context-insensitive, field-sensitive vs. field-insensitive, unification-based vs. subset-based, etc.)。传统上，别名分析以Must、May 或 No别名响应来响应查询，表明两个指针[始终](https://llvm.org/docs/AliasAnalysis.html" \l "must-may-or-no)指向同一个对象，可能指向同一个对象，或者已知永远不会指向同一个对象目的。

LLVM [AliasAnalysis](https://llvm.org/doxygen/classllvm_1_1AliasAnalysis.html)类是client端使用的主要接口和 LLVM 系统中别名分析的实现。此类是别名分析信息的client 端与提供它的实现之间的通用接口，旨在支持广泛的实现和client 端（但目前所有client 端都被假定为对流不敏感）。除了简单的别名分析信息之外，此类还公开了来自那些可以提供它的实现的 Mod/Ref 信息，从而允许强大的分析和转换能够很好地协同工作。

本文档包含成功实现此接口、使用它以及测试双方所需的信息。它还解释了关于结果的确切含义的一些细节。

## AliasAnalysis类概述

AliasAnalysis 类定义各种别名分析实现应支持的接口。此类导出两个重要的枚举：AliasResult和 ModRefResult分别表示别名查询或 mod/ref 查询的结果。

该AliasAnalysis接口公开有关内存的信息，以几种不同的方式表示。特别是，内存对象表示为起始地址和大小，函数调用表示为执行调用的实际 call或invoke指令。该 AliasAnalysis接口还公开了一些辅助方法，允许您获取任意指令的 mod/ref 信息。

所有AliasAnalysis接口都要求在涉及多个值的查询中，不是常量的值都定义在同一个函数中。

### 指针的表示

最重要的是，AliasAnalysis该类提供了几个方法，用于查询两个内存对象是否有别名，函数调用是否可以修改或读取内存对象等。对于所有这些查询，内存对象表示为一对它们的起始地址（一个符号 LLVM Value\*）和静态大小。

将内存对象表示为起始地址和大小对于正确的别名分析至关重要。例如，考虑这个（愚蠢但可能的）C 代码：

|  |
| --- |
| int i;  char C[2];  char A[10];  /\* ... \*/  for (i = 0; i != 10; ++i) {  C[0] = A[i]; /\* One byte store \*/  C[1] = A[9-i]; /\* One byte store \*/  } |

在这种情况下，basic-aa传递将消除对 C[0] 和  C[1] 的存储的歧义，因为它们是对相隔一个字节的两个不同位置的访问，并且每个访问都是一个字节。在这种情况下，循环不变代码移动 (LICM) pass 可以使用存储移动从循环中删除存储。相比之下，下面的代码：

|  |
| --- |
| int i;  char C[2];  char A[10];  /\* ... \*/  for (i = 0; i != 10; ++i) {  ((short\*)C)[0] = A[i]; /\* Two byte store! \*/  C[1] = A[9-i]; /\* One byte store \*/  } |

在这种情况下，C 的两个存储互为别名，因为对 &C[0]元素的访问是两个字节的访问。如果大小信息在查询中不可用，即使是第一种情况也必须保守地假设访问别名。

### alias方法

该alias方法是用于确定两个内存对象是否互为别名的主要接口。它采用两个内存对象作为输入，并根据需要返回 MustAlias、PartialAlias、MayAlias 或 NoAlias。

与所有AliasAnalysis接口一样，该alias方法要求在同一个函数中定义两个指针值，或者至少其中一个值是常量。

#### Must, May, and No Alias Responses

当基于一个指针的任何内存引用与 基于另一个指针的任何内存引用之间从不存在直接依赖性时，可以使用该NoAlias响应。最明显的例子是当两个指针指向不重叠的内存范围时。另一种情况是，这两个指针仅用于读取内存。另一种情况是，当内存在通过一个指针的访问和通过另一个指针的访问之间被释放和重新分配时——在这种情况下，存在依赖性，但它是由释放和重新分配调节的。

[noalias](https://llvm.org/docs/LangRef.html" \l "noalias)关键字是一个例外；“irrelevant”的依赖项将被忽略。

只要两个指针可能指向同一个对象，就会使用MayAlias响应。

当已知两个内存对象以某种方式重叠时使用PartialAlias响应，无论它们是否从相同地址开始。

仅当保证两个内存对象始终从完全相同的位置开始时，才可能返回MustAlias响应。MustAlias 响应并不意味着指针比较相等。

### getModRefInfo方法

这些getModRefInfo方法返回有关指令的执行是否可以读取或修改内存位置的信息。Mod/Ref 信息总是保守的：如果一条指令可能读取或写入一个位置， 则返回ModRef。

该类AliasAnalysis还提供了一种测试函数调用之间依赖关系的getModRefInfo方法。此方法采用两个调用站点 ( CS1 & CS2)，如果两个调用都没有写入由另一个读取或写入的内存。则返回NoModRef。如果CS1读取由CS2写入的内存，则返回Ref。如果CS1写入由 CS2 读取或写入的内存，则返回Mod。 CS1 可能读取或写入由 CS2 写入的内存，则返回ModRef。请注意，此关系不可交换。

### 其他有用的AliasAnalysis方法

其他一些信息通常由各种别名分析实现收集，可以被各种client 端很好地利用。

#### getModRefInfoMask方法

该getModRefInfoMask方法，基于关于指针是指向全局常量内存（它返回NoModRef）还是局部不变内存（它返回Ref）的知识，返回所提供指针的 Mod/Ref 信息的界限。全局常量内存包括函数、常量全局变量和空指针。局部不变内存是我们知道在其 SSA 值的生命周期内不变的内存，但不一定在程序的生命周期内不变：例如，readonly noalias参数指向的内存在相应函数调用期间是已知不变的. 给定内存位置 Loc 的Mod/Ref 信息 MRI，MRI可以使用类似MRI &= AA.getModRefInfoMask(Loc)的语句进行优化。另一个有用的idiom是  isModSet(AA.getModRefInfoMask(Loc)); 这将检查是否可以修改给定位置。为了方便，还有一个方法 pointsToConstantMemory(Loc)；这是同义词 isNoModRef(AA.getModRefInfoMask(Loc))。

#### doesNotAccessMemory和 onlyReadsMemory方法

这些方法用于为函数调用提供非常简单的 mod/ref 信息。如果分析可以证明函数从不读取或写入内存，或者函数仅从常量内存读取，则该doesNotAccessMemory方法为函数返回 true。具有此属性的函数没有副作用，仅依赖于它们的输入参数，如果它们形成公共子表达式或被提升到循环之外，则允许它们被消除。许多常见函数以这种方式运行（例如，sin和cos），但许多其他函数则不然（例如，acos修改errno变量的）。

如果分析可以证明（至多）该函数仅从非易失性存储器中读取，则该onlyReadsMemory方法为该函数返回 true。具有此属性的函数没有副作用，仅取决于它们的输入参数和调用时的内存状态。只要没有更改内存内容的存储指令，此属性就可以消除和移动对这些函数的调用。请注意，满足该doesNotAccessMemory方法的所有函数也满足onlyReadsMemory.

## 编写一个新的AliasAnalysis实现

为 LLVM 编写新的别名分析实现非常简单。已经有几个实现可以用作示例，以下信息应该有助于填写任何详细信息。例如，查看LLVM 中包含的[各种别名分析实现](https://llvm.org/docs/AliasAnalysis.html" \l "various-alias-analysis-implementations)。

### 不同的 Pass 样式

第一步是确定 您的别名分析需要使用哪种类型的[LLVM pass](https://llvm.org/docs/WritingAnLLVMPass.html) 。与大多数其他分析和转换的情况一样，从您要解决的问题类型来看，答案应该是相当明显的：

* 1.如果您需要过程间分析，它应该是一个Pass.
* 2.如果你是函数局部分析(If you are a function-local analysis)，子类FunctionPass。
* 3.如果你根本不需要看程序(If you don’t need to look at the program at all)，子类ImmutablePass。

除了你子类化的pass之外，你当然也应该继承 AliasAnalysis接口，并使用RegisterAnalysisGroup 模板注册为AliasAnalysis的实现。

### 所需的初始化调用

您的AliasAnalysis子类需要在 AliasAnalysis基类上调用两个方法：getAnalysisUsage和 InitializeAliasAnalysis。特别是， 除了声明和实现您的pass具有的任何pass依赖性之外，您的getAnalysisUsage实现还应该显式调用该 AliasAnalysis::getAnalysisUsage 方法。因此你应该有如下这样的东西：

|  |
| --- |
| void getAnalysisUsage(AnalysisUsage &AU) const {  AliasAnalysis::getAnalysisUsage(AU);  // declare your dependencies here.  } |

此外，您必须从分析运行方法（run for a Pass, runOnFunction for a FunctionPass, or InitializePass for an ImmutablePass）中调用该InitializeAliasAnalysis方法。例如（作为一个pass的一部分）：

|  |
| --- |
| bool run(Module &M) {  InitializeAliasAnalysis(this);  // Perform analysis here...  return false;  } |

### override所需的方法

您必须override AliasAnalysis的所有子类上的getAdjustedAnalysisPointer方法。此方法的示例实现如下所示：

|  |
| --- |
| void \*getAdjustedAnalysisPointer(const void\* ID) override {  if (ID == &AliasAnalysis::ID)  return (AliasAnalysis\*)this;  return this;  } |

### 可以指定的接口

所有[AliasAnalysis](https://llvm.org/doxygen/classllvm_1_1AliasAnalysis.html)虚拟方法默认提供到另一个别名分析实现的[链](https://llvm.org/docs/AliasAnalysis.html" \l "aliasanalysis-chaining)接，最终返回保守的正确的信息（分别为别名和 mod/ref 查询返回“May”Alias 和“Mod/Ref”）。根据您正在实现的分析功能，您只需重写需要改进的接口。

### AliasAnalysis链接行为

每个别名分析传递到另一个别名分析实现的链接（例如，用户可以指定“ -basic-aa -ds-aa -licm”以从两个别名分析中获得最大收益）。别名分析类会自动处理您未override的方法的大部分工作。对于您override的方法，在返回保守的 MayAlias 或 Mod/Ref 结果的代码路径中，只需返回超类(superclass)计算的任何内容。例如：

|  |
| --- |
| AliasResult alias(const Value \*V1, unsigned V1Size,  const Value \*V2, unsigned V2Size) {  if (...)  return NoAlias;  ...  // Couldn't determine a must or no-alias result.  return AliasAnalysis::alias(V1, V1Size, V2, V2Size);  } |

除了分析查询之外，您还必须确保无条件地将 LLVM 更新通知方法传递给超类（如果您覆盖它们），这允许更新更改中的所有别名分析。

### 更新转换分析结果

别名分析信息最初是为程序的静态快照(static snapshot )计算的，但client 端将使用此信息对代码进行转换。除了最普通的别名分析形式之外，所有其他形式的别名分析都需要更新其分析结果以反映这些转换所做的更改。

该AliasAnalysis接口公开了四种方法，用于将程序更改从client 端传达给分析实现。各种别名分析实现应该使用这些方法来确保它们的内部数据结构随着程序的变化（例如，当一条指令被删除时）保持最新，并且别名分析的client 端必须确保适当地调用这些接口.

#### deleteValue方法

deleteValue当转换从程序中删除指令或任何其他值（包括不使用指针的值）时，转换将调用该方法。通常，别名分析会保留包含程序中每个值的条目的数据结构。调用此方法时，他们应该删除指定值的任何条目（如果存在）。

#### copyValue方法

copyValue在程序中引入新值时使用该方法。没有办法向程序中引入一个以前不存在的值（这对于安全的编译器转换没有意义），所以这是引入新值的唯一方法。此方法指示新值与正在复制的值具有完全相同的属性。

#### replaceWithNewValue方法

此方法是一种简单的辅助方法，旨在使client 端更易于使用。它是通过将旧的分析信息复制到新值，然后删除旧值来实现的。此方法不能被别名分析实现override。

#### addEscapingUse方法

当指针值的使用以可能使预先计算的分析信息无效的方式发生变化时，将使用该addEscapingUse方法。实现可以使用此回调为自分析时间以来使用发生变化的点提供保守的响应，或者可以重新计算其部分或全部内部状态以继续提供准确的响应。

通常，指针值的任何新使用都被视为转义使用，并且必须通过此回调报告，但以下用途除外：

* A bitcast or getelementptr of the pointer
* A store through the pointer (but not a store of the pointer)
* A load through the pointer

### 效率问题

从 LLVM 的角度来看，要提供高效的别名分析，您唯一需要做的就是确保快速处理别名分析查询。别名分析结果的实际计算（“run”方法）只执行一次，但可能执行许多（可能重复）查询。因此，请尝试将尽可能多的计算移至 run 方法（在合理范围内）。

### 限制

AliasAnalysis 基础结构有几个限制，这使得编写新的AliasAnalysis实现变得困难。

无法override默认的别名分析。如果能够执行“opt -my-aa -O2-my-aa ”之类的操作并将其 用于所有需要 AliasAnalysis 的 pass，这将非常有用，但目前不支持它，除非更改源代码和重新编译。同样，也没有办法将分析链设置为默认值。

转换过程无法声明它们保留 AliasAnalysis实现。该AliasAnalysis接口包括 deleteValue和copyValue 的方法，旨在允许pass 保持 AliasAnalysis 一致，但是pass无法在其getAnalysisUsage 中声明它这样做。一些passes尝试使用AU.addPreserved<AliasAnalysis> ，但这实际上没有任何效果。

同样，该“opt -p”选项在每个pass 之间引入ModulePass，从而防止使用 FunctionPass 别名分析passes。

AliasAnalysis API 确实具有在删除或复制值时通知实现的AliasAnalysis功能，但这些还不够。还有许多其他方法可以修改 LLVM IR，这些方法可能与 AliasAnalysis无法表达的实现相关。

该AliasAnalysisDebugger 实用程序似乎表明，AliasAnalysis 实现可以期望在任何相关Value信息 出现在别名查询之前通知它们。但是，流行的client 端（例如）GVN 不支持此功能，并且已知在使用 AliasAnalysisDebugger 会引发错误。

该AliasSetTracker 类（由 LICM 所使用）进行了不确定数量的别名查询。这可能导致涉及在预定数量的查询后暂停执行的调试技术变得不可靠。

许多别名查询可以根据其他别名查询重新表述。当多个AliasAnalysis查询链接在一起时，从链的开头开始这些查询是有意义的，注意避免无限循环，但是目前想要这样做的实现只能从它自己开始这样的查询。

## 使用别名分析结果

有几种不同的方法可以使用别名分析结果。按照优先顺序，这些是：

### 使用MemoryDependenceAnalysis pass

该memdep pass 使用别名分析来提供有关内存使用指令的高级依赖信息。例如，这将告诉您哪个存储了要进行加载的负载(feed)。它使用缓存和其他技术来提高效率，并被 Dead Store Elimination、GVN 和 memcpy 优化使用。

### 使用AliasSetTracker类

许多转换需要有关在某些范围内处于活动状态的别名集的信息，而不是有关成对别名的信息。[AliasSetTracker](https://llvm.org/doxygen/classllvm_1_1AliasSetTracker.html)类用于根据AliasAnalysis 接口提供的成对别名分析信息有效地构建这些别名集 。

首先，您使用“ add”方法初始化 AliasSetTracker，以在您感兴趣的范围内添加有关各种潜在别名指令的信息。完成所有别名集后，您的pass 应该使用AliasSetTracker begin()/end()方法, 来简单地迭代构造的别名集。

由AliasSetTracker 形成的AliasSets 保证不相交，计算集合的 mod/ref 信息和波动性，并跟踪集合中的所有指针是否都是 Must 别名。AliasSetTracker 还确保集合根据调用指令正确折叠，并且可以提供每个集合中的指针列表。

作为此用户的示例，[循环不变代码移动](https://llvm.org/docs/doxygen/structLICM.html) pass 使用AliasSetTrackers 来计算每个循环嵌套的别名集。如果一个AliasSet在一个循环中没有被修改，那么来自那个集合的所有加载指令都可以被提升到循环之外。如果任何别名集被存储并且是必须的别名集，那么这些存储可能被下沉到循环之外，在循环嵌套期间将内存位置提升到寄存器。只有当指针参数是循环不变的时，这两种转换才适用。

#### AliasSetTracker 实现

AliasSetTracker 类的实现尽可能高效。当指针插入到别名多个集合的 AliasSetTracker 时，它使用联合查找算法(union-find algorithm)有效地合并 AliasSet。主要数据结构是一个哈希表，将指针映射到它们所在的 AliasSet。

AliasSetTracker 类必须维护每个 AliasSet 中的所有 LLVM Value\*的列表。由于哈希表已经有每个感兴趣的 LLVM Value\* 的条目，AliasesSets 通过这些哈希表节点将链表线程化，以避免不必要地分配内存，并使合并别名集非常有效（链表合并是常数时间） .

如果您只是 AliasSetTracker 的一个client 端，您应该不需要了解这些细节，但是如果您查看代码，希望这个简短的描述将有助于理解为什么事情是这样设计的。

### 直接使用AliasAnalysis接口

如果这些实用程序类都不是您的pass 所需要的，您应该直接使用该AliasAnalysis 类公开的接口。尽可能尝试使用更高级别的方法（例如，如果可能，使用 mod/ref 信息而不是直接使用 [别名](https://llvm.org/docs/AliasAnalysis.html" \l "alias)方法）以获得最佳精度和效率。

## 现有的别名分析实现和client 端

如果您打算使用 LLVM 别名分析基础架构，您应该知道哪些client 端和别名分析的实现是可用的。特别是，如果您正在实现别名分析，您应该了解可用于监视和评估不同实施的[client](https://llvm.org/docs/AliasAnalysis.html" \l "the-clients) 端。

### 可用的AliasAnalysis实现

本节列出了AliasAnalysis 接口的各种实现。所有这些都链接到其他别名分析实现。

#### -basic-aa pass

-basic-aa pass 是一种激进的局部分析，它知道许多重要的事实：

* Distinct globals, stack allocations, and heap allocations can never alias.
* 全局、堆栈分配和堆分配从不为空指针设置别名。
* 一个结构体的不同字段不会混用。
* 具有静态不同下标的数组索引不能别名。
* 许多常见的标准 C 库函数从不访问内存或只读取内存。
* 显然指向常量全局变量“ pointToConstantMemory”的指针。
* 如果函数调用永远不会从分配它们的函数中逃脱（自动数组的常见情况），则函数调用不能修改或引用堆栈分配。

#### -globalsmodref-aa pass

这个pass 为没有“have their address taken”的内部全局变量实现了一个简单的上下文相关的 mod/ref 和别名分析。如果一个全局的地址没有被占用，该pass 知道没有指针作为全局变量的别名。此pass 还跟踪知道从不访问内存或从不读取内存的函数。这允许某些优化（例如 GVN）完全消除调用指令。

这个pass 的真正强大之处在于它为调用指令提供上下文相关的 mod/ref 信息。这允许优化器知道对函数的调用不会破坏或读取全局值，从而可以消除加载和存储。

说明：

此pass的范围有限（仅支持非地址获取的全局变量），但分析速度非常快。

#### -steens-aa pass

该-steens-aa pass 实现了著名的“Steensgaard 的算法”的变体，用于过程间别名分析。Steensgaard 的算法是一种基于统一的、流不敏感、上下文不敏感和字段不敏感的别名分析，它也非常可扩展（有效的线性时间）。

LLVM -steens-aa pass 使用数据结构分析框架实现了 Steensgaard 算法的“speculatively field-sensitive”版本。这使其比标准算法具有更高的精度，同时保持出色的分析可扩展性。

说明：

-steens-aa在可选的“poolalloc”模块中可用。它不是 LLVM 核心的一部分。

#### -ds-aa pass

-ds-aa pass 实现了完整的数据结构分析算法。数据结构分析是一种基于模块化统一、流不敏感、上下文敏感和推测字段敏感的别名分析，它也具有相当可扩展性，通常在.O(n \* log(n))

该算法能够响应各种别名分析查询，并且还可以提供上下文敏感的 mod/ref 信息。迄今为止唯一未实现的主要功能是对 must-alias 信息的支持。

说明：

-ds-aa在可选的“poolalloc”模块中可用。它不是 LLVM 核心的一部分。

#### -scev-aa pass

该-scev-aa pass通过将 AliasAnalysis 查询转换为 ScalarEvolution 查询来实现它们。getelementptr和循环归纳变量与其他别名分析相比，有更全面的理解 。

### 别名分析驱动的转换

LLVM 包括几个别名分析驱动的转换，可以与上述任何实现一起使用。

#### -adce pass

-adce实现积极无用代码消除 的 pass ， 该pass 使用AliasAnalysis接口删除对没有副作用且未使用的函数的调用。

#### -licm pass

-licm pass 实现了各种循环不变代码移动相关的转换。它使用AliasAnalysis接口进行几种不同的转换：

* 如果循环中没有修改加载内存的指令，它使用 mod/ref 信息将加载指令提升或下沉到循环之外。
* 它使用 mod/ref 信息将函数调用提升到不写入内存且循环不变的循环之外。

它使用别名信息来提升在循环中加载和存储的内存对象存放在寄存器中。如果加载/存储的内存位置没有别名，它可以执行此操作。

#### -argpromotion pass

-argpromotion pass 促进按引用参数传递，而不是按值传递。特别是，如果仅从中加载指针参数，它会将加载的值而不是地址传递给函数。这个pass 使用别名信息来确保从参数指针加载的值在函数入口和指针的任何加载之间没有被修改。

#### -gvn,-memcpyopt和-dse passes

这些passes 使用 AliasAnalysis 信息来推断加载和存储。

### 用于调试和评估实现的client 端

这些通道对于评估各种别名分析实现很有用。您可以将它们与以下命令一起使用：

% opt -ds-aa -aa-eval foo.bc -disable-output -stats

-print-alias-sets pass

该-print-alias-sets pass 作为opt打印出由AliasSetTracker类形成的别名集的工具的一部分公开。如果您正在使用该AliasSetTracker 类，这将很有用。要使用它，请使用类似的东西：

% opt -ds-aa -print-alias-sets -disable-output

#### -aa-eval pass

-aa-eval pass 简单地遍历函数中的所有指针对，并询问别名分析指针是否为别名。这表明了别名分析的精度。打印的统计信息指示找到的 no/may/must 别名的百分比（更精确的算法将具有更少的 may 别名）。

## 内存依赖分析

说明：

我们目前正在将事物从 MemoryDependenceAnalysis迁移到MemorySSA。请尝试使用它。

如果您只是希望成为别名分析信息的client ，请考虑改用内存依赖性分析接口。MemDep 是别名分析之上的惰性缓存层，能够在块内或块间级别回答给定指令依赖于哪些先前内存操作的问题。由于其惰性和缓存策略，与直接访问别名分析相比，使用 MemDep 可以显着提高性能。

# memory SSA

## 简介

MemorySSA是一种分析，使我们能够廉价地推理各种内存操作之间的交互。它的目标是为大多数（如果不是全部）用例替换 MemoryDependenceAnalysis。这是因为，除非您非常小心，否则在 LLVM 中使用 MemoryDependenceAnalysis 很容易导致二次时间算法( quadratic-time algorithms)。此外，MemorySSA没有像 MemoryDependenceAnalysis 那样多的任意限制，因此您也应该获得更好的结果。MemorySSA 的一个常见用途是快速找出绝对不会发生的事情（例如，循环中的提升机不会发生的原因）。

在高层次上，MemorySSA目标之一是为内存提供基于 SSA 的形式，并配有 def-use 和 use-def 链，使用户能够快速找到内存操作的 may-def 和 may-uses。它也可以被认为是一种廉价地为内存的完整状态提供版本，并将内存操作与这些版本相关联的方法。

本文档介绍了MemorySSA它的结构，以及一些关于如何MemorySSA工作的基本直觉。

可以在[此处](http://www.airs.com/dnovillo/Papers/mem-ssa.pdf)找到有关 MemorySSA 的论文（附有关于它如何在 GCC 中实现的注释）。虽然，它相对过时了；该论文引用了多个内存分区，但 GCC 最终换成了只使用一个分区，就像我们现在在 LLVM 中使用的那样。与 GCC 一样，LLVM 的 MemorySSA 是过程内的。

## 内存SSA结构

MemorySSA 是一个虚拟 IR。构建完成后，MemorySSA将包含一个将Instructions 映射到MemoryAccesses 的结构， MemorySSA与 LLVM Instructions 是平行的。

每个 MemoryAccess可以是以下三种类型之一：

* MemoryDef
* MemoryPhi
* MemoryUse

MemoryDefs 是可以修改内存或引入某种排序约束的操作。MemoryDefs 的示例包括stores、函数调用、具有（或更高）顺序的loads、 volatile 操作、内存栅栏等。MemoryDef会引入新的版本，该版本会链接到一个MemoryDef/MemoryPhi，这是新版本所基于的旧版本。这意味着存在一条直接或间接连接所有 Defs的 singleDef chain。例如在：

|  |
| --- |
| b = MemoryDef(a)  c = MemoryDef(b)  d = MemoryDef(c) |

d与c 直接相连和 b 间接相连。这意味着d 可能会破坏（见下文）c 或 b /或两者。这反过来意味着，如果不使用[The walker](https://llvm.org/docs/MemorySSA.html" \l "the-walker)，最初每个MemoryDef都会破坏每个其它的MemoryDef。

MemoryPhis 是PhiNodes，但用于内存操作。如果在任何时候我们有两个（或更多）MemoryDef可以流入一个 BasicBlock， BasicBlock块的顶部MemoryAccess将是 一个 MemoryPhi。与 在LLVM IR 内一样，MemoryPhis 不对应于任何具体操作。因此，BasicBlocks 被映射到MemorySSA 内的MemoryPhis，而 Instructions 被映射到MemoryUses 和MemoryDefs.

另请注意，在 SSA 中，Phi 节点合并 must-reach 定义（即必须是新版本变量的定义）。在 MemorySSA 中，PHI 节点合并可能到达的定义（也就是说，在消除歧义之前，到达 phi 节点的版本可能会也可能不会破坏给定变量）。

MemoryUses 是使用但不修改内存的操作。 MemoryUse的一个示例是load, 或readonly函数调用。

存在的每个函数都有一个特殊的MemoryDef，称为liveOnEntry。它控制着正在运行MemoryAccess的函数中的每一个MemorySSA，并暗示我们已经到达了函数的顶部。在 LLVM IR 中，它是唯一映射到 no Instruction 的 MemoryDef。liveOnEntry的使用意味着正在使用的内存在函数开始之前是未定义的或已定义的。

下面是所有这些覆盖在 LLVM IR 上的示例（通过在 .ll文件上运行opt -passes='print<memoryssa>' -disable-output获得的）。查看此示例时，从 clobbers 的角度来查看它可能会有所帮助。一个给定的MemoryAccess 的操作数都是 said MemoryAccess 的（潜在的）clobber， 一个MemoryAccess 产生的值可以作为其他MemoryAccesses 的clobber。

如果 一个MemoryAccess是另一个的clobber，则意味着这两个 MemoryAccesses 可能访问同一内存。例如， x = MemoryDef(y)意味着x潜在地修改y修改/约束（或已修改/约束）的内存。以同样的方式，a = MemoryPhi({BB1,b},{BB2,c})意味着任何使用a 的人正在访问可能被b 和c 其中之一或（或两者）修改/约束的内存。最后，MemoryUse(x)意味着此使用访问x 已修改/约束的内存（例如，认为如果x = MemoryDef(...) 和MemoryUse(x)在同一个循环中，则不能单独将使用提升到外部）。

另一种有用的查看方式是根据内存版本。在那个方式中，给定MemoryAccess的操作数是操作前整个内存的版本，如果访问产生一个值（即MemoryDef/MemoryPhi），则该值是操作后内存的新版本。

|  |
| --- |
| define void @foo() {  entry:  %p1 = alloca i8  %p2 = alloca i8  %p3 = alloca i8  ; 1 = MemoryDef(liveOnEntry)  store i8 0, i8\* %p3  br label %while.cond  while.cond:  ; 6 = MemoryPhi({entry,1},{if.end,4})  br i1 undef, label %if.then, label %if.else  if.then:  ; 2 = MemoryDef(6)  store i8 0, i8\* %p1  br label %if.end  if.else:  ; 3 = MemoryDef(6)  store i8 1, i8\* %p2  br label %if.end  if.end:  ; 5 = MemoryPhi({if.then,2},{if.else,3})  ; MemoryUse(5)  %1 = load i8, i8\* %p1  ; 4 = MemoryDef(5)  store i8 2, i8\* %p2  ; MemoryUse(1)  %2 = load i8, i8\* %p3  br label %while.cond  } |

MemorySSA IR 指令之前的注释 显示在它们映射到的指令（如果存在这样的指令）。例如， 1 = MemoryDef(liveOnEntry)  是 一个  MemoryAccess（具体来说， 是一个  MemoryDef），它描述了 LLVM 指令 store i8 0, i8\* %p3 。MemorySSA 中的其他地方将此特定 MemoryDef 称为 1 （很像在 LLVM 中如何使用 %1 来 load i8, i8\* %p1）。同样， MemoryPhis 不对应于任何 LLVM 指令，因此 一个MemoryPhi 正下方的行并不特殊。

从上到下：

* 6 = MemoryPhi({entry,1},{if.end,4}) ，说明，当进入while.cond 时 ，它的到达定义(reaching definition )是1或4。这 MemoryPhi在文本 IR 中由编号引用6。
* 2 = MemoryDef(6)， 说明，store i8 0, i8\* %p1是一个定义，在它之前的到达定义是 6或while.cond之后的MemoryPhi。（请参阅下面的[Use 和 Def 优化](https://llvm.org/docs/MemorySSA.html" \l "use-and-def-optimization)以及[Precision](https://llvm.org/docs/MemorySSA.html" \l "precision)部分，了解为什么这个 MemoryDef 没有链接到单独的、明确的MemoryPhi）。
* 3 = MemoryDef(6)， 说明， store i8 0, i8\* %p2 是一个定义；其到达定义是 6。
* 5 = MemoryPhi({if.then,2},{if.else,3})， 说明，此块之前的 clobber 可以是2或3。
* MemoryUse(5) ， 说明， load i8, i8\* %p1 是对内存的使用，并且它被 5 破坏了。
* 4 = MemoryDef(5)， 说明， store i8 2, i8\* %p2是一个定义；它的到达定义是5。
* MemoryUse(1)， 说明，load i8, i8\* %p3只是对内存的使用，最后可能破坏此使用的是上面的while.cond（例如， 存储到%p3）。用内存版本控制的说法，它实际上只取决于内存版本 1，并且不受此后生成的新内存版本的影响。

顺便说一句，MemoryAccess 是Value，主要是为了方便；它并不意味着与 LLVM IR 交互。

## MemorySSA 的设计

MemorySSA是可以为任意函数构建的分析。当它被构建时，它会传递函数的 IR 以建立它的MemoryAccesses 映射。然后，您可以查询MemorySSA诸如MemoryAccesses 之间的支配关系之类的内容，并为任何给定的Instruction获取 MemoryAccess。

完成MemorySSA构建后，它还会为您提供一个MemorySSAWalker 可以使用的（见下文）。

### walker

有助于MemorySSA完成其工作的结构是MemorySSAWalker，或简称为walker。walker 的目标是为 MemoryAccesss 直接表示的内容之外的 clobber 查询提供答案。例如，给定：

|  |
| --- |
| define void @foo() {  %a = alloca i8  %b = alloca i8  ; 1 = MemoryDef(liveOnEntry)  store i8 0, i8\* %a  ; 2 = MemoryDef(1)  store i8 0, i8\* %b  } |

store to %a显然不是 store to %b 的clobber(clobber)。walker 的目标是弄清楚这一点，并在查询 MemoryAccess 2 的 clobber 时返回liveOnEntry。

默认情况下，MemorySSA提供一个 walker，它可以通过查询您碰巧使用的任何别名分析堆栈来优化MemoryDefs 和MemoryUses。不过，Walkers 是可灵活而构建的，因此创建更专业的 walkers 是完全合理的（并且是预期的）（例如专门查询GlobalsAA的 walkers，总是在MemoryPhi节点处停止的walkers， 等等）。

#### 默认 walker API

有两个主要的 API 用于使用 walker 检索破坏访问(clobbering access )：

* MemoryAccess \*getClobberingMemoryAccess(MemoryAccess \*MA);返回 MA 的破坏内存访问(clobbering memory access)，缓存所有中间结果，作为每个访问查询的一部分计算。
* MemoryAccess \*getClobberingMemoryAccess(MemoryAccess \*MA, const MemoryLocation &Loc); 返回访问破坏内存(clobbering memory access)位置Loc，从 MA 开始。因为此 API 不请求特定内存访问的破坏访问，所以没有可缓存的结果。

#### 自己定位 clobbers

If you choose to make your own walker, you can find the clobber for a MemoryAccess by walking every MemoryDef that dominates said MemoryAccess。The structure of MemoryDefs makes this relatively simple; they ultimately form a linked list of every clobber that dominates the MemoryAccess that you’re trying to optimize. In other words, the definingAccess of a MemoryDef is always the nearest dominating MemoryDef or MemoryPhi of said MemoryDef。

### 使用和定义优化(Use and Def optimization)

MemoryUses 保留一个操作数，是它们的定义或优化访问。传统上在构建时MemorySSA优化MemoryUses，达到给定的阈值。具体来说， MemoryUse 的每个操作数被优化为指向 said MemoryUse 的实际clobber。这可以在上面的例子中看到； if.end 内的第二个MemoryUse有一个操作数1，它 是 来自入口块的MemoryDef。这样做是为了使walking、value numbering等更快、更容易。在[本次修订](https://reviews.llvm.org/D121381)中，默认值已更改为在构建时不优化使用，以便在一个pass内不需要walking时提供减少编译时间的选项。如果以前没有这样做的话， 大多数用户调用新的 API ensureOptimizedUses()来保持以前的行为并做一次性优化MemoryUses 。新 pass 用户建议调用 ensureOptimizedUses()。

最初不可能以相同的方式优化 MemoryDefs，因为我们限制MemorySSA每次访问只能使用一个操作数。这已更改，MemoryDef现在保留两个操作数。第一个，定义访问，在同一个基本块中，总是前一个MemoryDef或MemoryPhi; 或者如果当前块没有任何其他写入内存的访问，则在dominating predecessor 的后一个。这是walking Def 链所必需的。第二个操作数是优化访问，如果之前有对 walker 的getClobberingMemoryAccess(MA)调用。此 API 将缓存信息作为MA的一部分。优化所有MemoryDefs 具有二次时间复杂度，默认情况下不会这样做。

对任何 MemoryDef 的使用一次walk，都可以找到针对它优化的访问。这个 walk的代码片段如下所示：

|  |
| --- |
| MemoryDef \*Def; // find who's optimized or defining for this MemoryDef  for (auto& U : Def->uses()) {  MemoryAccess \*MA = cast<MemoryAccess>(Use.getUser());  if (auto \*DefUser = cast\_of\_null<MemoryDef>MA)  if (DefUser->isOptimized() && DefUser->getOptimized() == Def) {  // User who is optimized to Def  } else {  // User who's defining access is Def; optimized to something else or not optimized.  }  } |

当MemoryUses 被优化时，对于给定的存储，您可以通过遍历存储的直接和传递使用来找到所有被该存储破坏的负载。

|  |
| --- |
| checkUses(MemoryAccess \*Def) { // Def can be a MemoryDef or a MemoryPhi.  for (auto& U : Def->uses()) {  MemoryAccess \*MA = cast<MemoryAccess>(Use.getUser());  if (auto \*MU = cast\_of\_null<MemoryUse>MA) {  // Process MemoryUse as needed.  }  else {  // Process MemoryDef or MemoryPhi as needed.  // As a user can come up twice, as an optimized access and defining  // access, keep a visited list.  // Check transitive uses as needed  checkUses (MA); // use a worklist for an iterative algorithm  }  }  } |

在 DeadStoreElimination pass中可以找到类似遍历的示例。

### 失效和更新

因为MemorySSA跟踪 LLVM IR，所以每当 IR 更新时它都需要更新。在这种情况下，“更新”包括添加、删除和移动 Instructions。更新 API 是根据需要进行的。如果您想要示例， GVNHoist和LICM 是 MemorySSAs update API 的用户。请注意，如果新访问触发许多MemoryPhies的多次插入和MemoryAccesses的重命名（优化失效） ，则添加新的 MemoryDefs（通过调用insertDef）可能是一个耗时的更新。

#### phi 放置(Phi placement)

MemorySSA只在真正需要它们的地方放置MemoryPhi。也就是说，它是一种经过修剪的 SSA 形式，就像 LLVM 的 SSA 形式一样。例如，考虑：

|  |
| --- |
| define void @foo() {  entry:  %p1 = alloca i8  %p2 = alloca i8  %p3 = alloca i8  ; 1 = MemoryDef(liveOnEntry)  store i8 0, i8\* %p3  br label %while.cond  while.cond:  ; 3 = MemoryPhi({%0,1},{if.end,2})  br i1 undef, label %if.then, label %if.else  if.then:  br label %if.end  if.else:  br label %if.end  if.end:  ; MemoryUse(1)  %1 = load i8, i8\* %p1  ; 2 = MemoryDef(3)  store i8 2, i8\* %p2  ; MemoryUse(1)  %2 = load i8, i8\* %p3  br label %while.cond  } |

因为我们从if.then和if.else中删除了store，所以if.end 内的 MemoryPhi 将毫无意义，所以我们不放置一个。因此，如果您需要在if.then或if.else内放置一个MemoryDef，您还需要为 if.end 创建一个MemoryPhi。

如果事实证明这是一个很大的负担，我们可以把MemoryPhis 放在各处。因为我们有Walkers能够优化上述 的 phis，所以这样做不应该禁止优化。

### 非目标(Non-Goals)

MemorySSA旨在推理内存操作之间的关系，并实现更快的查询。它并不是所有潜在的与内存相关的优化的唯一真实来源。Specifically, care must be taken when trying to use MemorySSA to reason about atomic or volatile operations; 例如：

|  |
| --- |
| define i8 @foo(i8\* %a) {  entry:  br i1 undef, label %if.then, label %if.end  if.then:  ; 1 = MemoryDef(liveOnEntry)  %0 = load volatile i8, i8\* %a  br label %if.end  if.end:  %av = phi i8 [0, %entry], [%0, %if.then]  ret i8 %av  } |

仅通过MemorySSA的分析，提升 “load to entry” 似乎是合法的。但是，因为它是一个 volatile load，所以不能这样做。

### 设计权衡(Design tradeoffs)

#### 精度

MemorySSA在 LLVM 中，故意以精度换取速度。让我们把内存变量看作是内存的不相交分区（也就是说，如果你有一个变量，如上所述，它代表整个内存，如果你有多个变量，每个变量代表内存的一些不相交的部分)

首先，因为alias分析结果相互冲突，而且每个结果可能是一个分析想要的结果（例如，TBAA可能会说no-alias，其他的可能会说must-alias），不可能按照每个优化想要的方式划分内存。其次，有些别名分析结果是不传递的（即A noalias B, and B noalias C, 并不代表A noalias C），所以如果没有变量来表示每一对，就不可能得出所有情况下的精确划分可能的别名。因此，精确划分可能需要引入至少 N^2 个新的虚拟变量、phi 节点等。

这些变量中的每一个都可能在多个 def sites 上被破坏。

To give an example, if you were to split up struct fields into individual variables, all aliasing operations that may-def multiple struct fields, will may-def more than one of them. This is pretty common (calls, copies, field stores, etc)。

在其他编译器中使用 SSA 形式的内存的经验表明，根本不可能精确地做到这一点，事实上，精确地做到这一点是不值得的，因为现在所有的优化都必须遍历大量的虚拟变量和 phi节点。

所以我们分区。在您进行分区时，经验再次告诉我们，分区到多个变量是没有意义的。它只会产生更多的 IR，并且优化仍然需要查询一些东西以进一步消除歧义。

结果，LLVM 划分为一个变量。

#### 实践中的精确度

实际上，LLVM 中的一些实现细节也会影响MemorySSA. 例如，AliasAnalysis 有各种上限，或对查看 phi 的限制，这会影响MemorySSA 可以推断的内容。不同passes所做的更改可能会使 MemorySSA“过度优化”（它可以提供比从头开始重新计算更准确的结果）或“优化不足”（如果重新计算它可以推断更多）。当结果依赖于由多个后续passes更新的MemorySSA获取的状态时，这可能会对单个隔离的pass重现结果带来挑战。使用和更新 MemorySSA的 passes 应通过 MemorySSAUpdater 提供的 API 或通过对 Walker 的调用来执行。直接优化到MemorySSA不允许。目前有一个范围狭窄的异常，其中 DSE (DeadStoreElimination) 在保证优化正确的遍历之后，更新存储的优化访问。这是完全允许的，因为遍历和推理超出了MemorySSA所做的范围，而且它们是“自由的”（即 DSE 无论如何都会这样做）。此异常设置在标志（“-dse-optimize-memoryssa”）下，可以禁用以帮助单独重现优化。

## LLVM 开发者会议演示

2016 年 LLVM 开发者会议：G. Burgess - [五分钟了解 MemorySSA](https://www.youtube.com/watch?v=bdxWmryoHak)。

2020 年 LLVM 开发者会议：S. Baziotis 和 S. Moll - [寻找绕过 LLVM 依赖分析族的方法](https://www.youtube.com/watch?v=1e5y6WDbXCQ)

# LLVM 循环术语（和规范形式）

## 循环定义

循环是代码优化器的一个重要概念。在 LLVM 中，控制流图中的循环检测由[LoopInfo](https://llvm.org/docs/LoopTerminology.html" \l "loopinfo)完成。它基于以下定义。

循环是来自控制流图（CFG；其中节点代表基本块）的节点子集，具有以下属性：

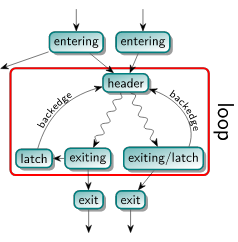
* 1.诱导子图(induced subgraph)（它是包含循环内 CFG 的所有边的子图）是强连接的（每个节点都可以从所有其他节点到达）。
* 2.从子集外部到子集的所有边都指向同一个节点，称为**header**。因此，头部支配循环中的所有节点（即，到任何循环节点的每条执行路径都必须通过**header**）。
* 3.循环是具有这些属性的最大子集。也就是说，不能添加来自 CFG 的额外节点，这样导出的子图仍将是强连接的，并且**header**将保持不变。

在计算机科学文献中，这通常被称为 natural loop。在 LLVM 中，更通用的定义称为 [循环](https://llvm.org/docs/CycleTerminology.html" \l "cycle-terminology)。

### 术语

循环的定义带有一些额外的术语：

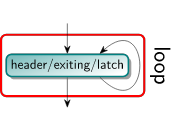
* 进入块( entering block)（或循环前导 loop predecessor）是一个非循环节点，它有一个边进入循环（必须是header）。如果只有一个entering block 进入 block，而且它的唯一边就是header，也称为loop's preheader。preheader 在循环中占主导地位，但它本身不是循环的一部分。
* latch 是一个循环节点，它有一个到header的边。
* backedge是从 latch 到 header 的边。
* 退出边(exiting edge)是从循环内部到循环外部节点的边。这样一条边的源称为exiting block，它的目标是exit block。



### 重要说明

这个循环定义有一些值得注意的结果：

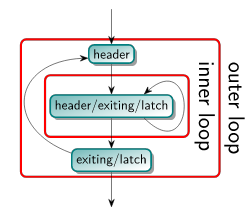
* 一个节点最多可以是一个循环的header。因此，循环可以通过其header来识别。由于header是进入循环的唯一入口，因此它可以称为单入口多出口 (SEME) 区域。
* 对于不能从函数入口到达的基本块，循环的概念是未定义的。这也源于未定义支配地位的概念。
* 最小的循环由一个分支到自身的单个基本块组成。在这种情况下，该块同时是header、latch （如果它有另一个边到另一个块，则是退出块）。没有分支到自身的单个块不被视为循环，即使它是微不足道的强连接也是如此。



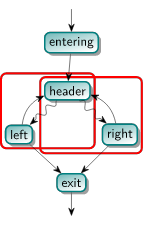
在这种情况下，header、exiting block 和 latch 的角色落在同一个节点上。LoopInfo将其报告为：

|  |
| --- |
| $ opt input.ll -passes='print<loops>'  Loop at depth 1 containing: %for.body<header><latch><exiting> |

* 循环可以相互嵌套。也就是说，一个循环的节点集可以是另一个具有不同循环头的循环的子集。函数中的循环层次结构形成了一个森林：每个顶级循环都是嵌套在其中的循环树的根。



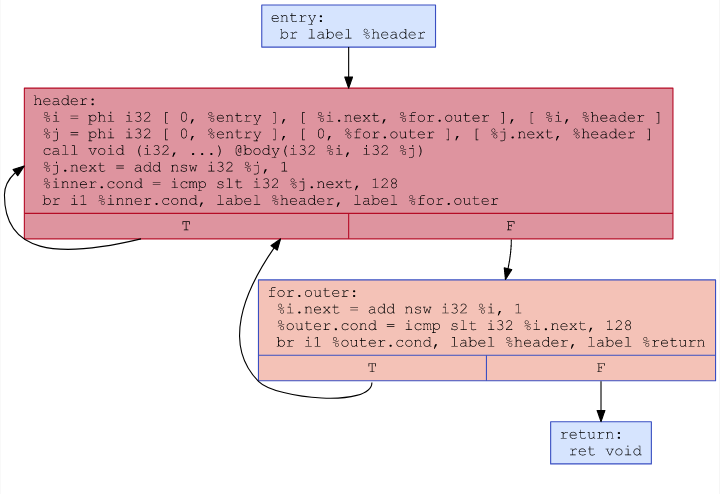
* 两个循环不可能只共享它们的几个节点。两个循环要么不相交，要么一个嵌套在另一个循环中。在下面的示例中，左右子集都违反了最大值条件。只有两个集合的合并才被认为是一个循环。



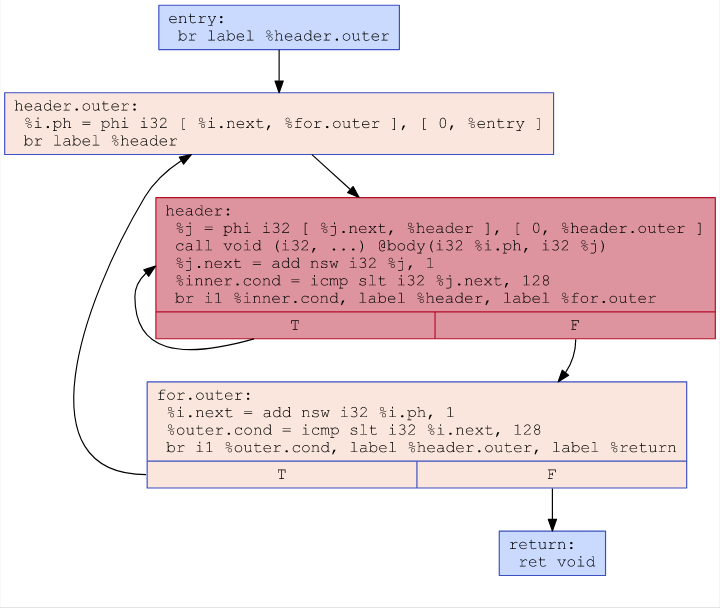
* 也有可能两个逻辑循环共享一个header，但被 LLVM 视为单个循环：

|  |
| --- |
| for (int i = 0; i < 128; ++i)  for (int j = 0; j < 128; ++j)  body(i,j); |

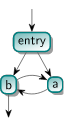
这可能在 LLVM-IR 中表示如下。请注意，只有一个标题，因此只有一个循环。



[LoopSimplify](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-loop-simplify)过程将检测循环并确保外循环和内循环的header分开。



* CFG 中的cycle并不意味着存在循环。下面的示例显示了这样一个 CFG，其中没有头节点支配循环中的所有其他节点。这称为irreducible control-flow。



术语可简化源于，通过用单个节点连续替换三个基本结构之一，将 CFG 折叠为单个节点的能力：基本块的顺序执行、非循环条件分支（或switches）和自身循环的基本块. [维基百科](https://en.wikipedia.org/wiki/Control-flow_graph" \l "Reducibility)有一个更正式的定义，基本上是说每个循环都有一个主导header。

* 不可简化的控制流(Irreducible control-flow)可以发生在循环嵌套的任何级别。也就是说，一个本身不包含任何循环的循环仍然可以在其主体中有循环控制流；没有嵌套在另一个循环中的循环仍然可以是外部cycle的一部分；并且在任何两个循环之间可以有额外的cycle，其中一个循环包含在另一个循环中。但是，LLVM循环涵盖了循环和不可简化的控制流。
* FixIrreducible pass 可以通过插入新的循环头将不可简化的控制流转换为循环。它没有包含在任何默认优化pass pipeline 中，但某些后端目标需要它。
* 退出边并不是跳出循环的唯一方法。其他可能性是无法访问的终止符、[[noreturn]] 函数、异常、信号和计算机的电源按钮。
* 循环“内部”的基本块没有返回循环的路径（即到锁 latch 或header ）不被视为循环的一部分。下面的代码说明了这一点。

|  |
| --- |
| for (unsigned i = 0; i <= n; ++i) {  if (c1) {  // When reaching this block, we will have exited the loop.  do\_something();  break;  }  if (c2) {  // abort(), never returns, so we have exited the loop.  abort();  }  if (c3) {  // The unreachable allows the compiler to assume that this will not rejoin the loop.  do\_something();  \_\_builtin\_unreachable();  }  if (c4) {  // This statically infinite loop is not nested because control-flow will not continue with the for-loop.  while(true) {  do\_something();  }  }  } |

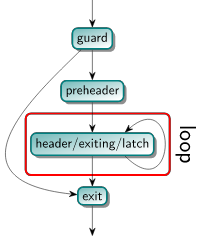
* 没有要求控制流最终离开循环，即循环可以是无限的。静态无限循环是没有退出边的循环。动态无限循环有退出边缘，但有可能永远不会被采用。这可能仅在某些情况下发生，例如当下面代码中的 n == UINT\_MAX 时。

|  |
| --- |
| for (unsigned i = 0; i <= n; ++i)  body(i); |

优化器有可能将动态无限循环转变为静态无限循环，例如当它可以证明退出条件始终为假时。因为永远不会采用退出边，所以优化器可以将条件分支更改为无条件分支。

如果 a 是一个循环， 用 [llvm.loop.mustprogress](https://llvm.org/docs/LangRef.html" \l "langref-llvm-loop-mustprogress) 元数据注释，则允许编译器假设它最终会终止，即使它无法证明这一点。例如，它可能会删除一个在其主体中没有任何副作用的 mustprogress-loop，即使程序可能永远停留在该循环中。C 和 C++等语言对某些循环具有这种向前推进的保证。另请参阅 [mustprogress](https://llvm.org/docs/LangRef.html" \l "langref-mustprogress)和 [willreturn](https://llvm.org/docs/LangRef.html" \l "langref-willreturn)函数属性，以及较旧的[llvm.sideeffect](https://llvm.org/docs/LangRef.html" \l "llvm-sideeffect)内在特性。

* 离开循环之前循环头的执行次数是循环行程次数(loop trip count )（或迭代次数）。如果根本不应该执行循环，则loop guard必须跳过整个循环：



* 由于循环头可能做的第一件事是检查是否有另一个执行，如果没有，立即退出而不做任何工作（另请参阅[旋转循环](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-loop-rotate)），循环行程次数(loop trip count )不是循环迭代次数的最佳度量。例如，对于非正数 n（在循环旋转之前），下面代码的header执行次数为 1，即使根本没有执行循环体。

|  |
| --- |
| for (int i = 0; i < n; ++i)  body(i); |

一个更好的衡量标准是 backedge-taken count，它是在循环之前任何 backedges 被采用的次数。它比进入header的执行的行程计数少一。

## 循环信息LoopInfo

LoopInfo是获取循环信息的核心分析。上面给出的定义几乎没有关键含义，这些含义对于成功使用此接口很重要。

* LoopInfo 不包含有关非循环周期的信息。因此，它不适用于任何需要完整循环检测以确保正确性的算法。
* LoopInfo 提供了一个接口，用于枚举所有顶级循环（例如，那些不包含在任何其他循环中的循环）。从那里，您可以遍历植根于该顶级循环的子循环树。
* 在优化期间变得静态不可访问的循环必须 从 LoopInfo 中删除。如果由于某种原因无法做到这一点，则需要进行优化以保持循环的静态可达性。

## 循环简化形式Loop Simplify Form

Loop Simplify Form 是一种规范形式，它使多个分析和转换更简单、更有效。它由 LoopSimplify ( -loop-simplify ) pass 确保，并在调度 LoopPass 时由 pass managers 自动添加。此pass 在 [LoopSimplify.h](https://llvm.org/doxygen/LoopSimplify_8h_source.html)中实现。成功时，循环具有：

* A preheader.
* A single backedge (which implies that there is a single latch).
* 专用出口。也就是说，循环的退出块都没有循环外的前置任务。这意味着所有退出块都由循环头控制。

## 闭环 SSA (LCSSA) Loop Closed SSA

如果程序处于 LCSSA 形式并且在循环中定义的所有值仅在该循环内使用，则该程序处于循环闭合 SSA 形式。

用 LLVM IR 编写的程序总是采用 SSA 形式，但不一定采用 LCSSA。为了实现后者，对于跨循环边界存在的每个值，将单入口 PHI 节点插入到每个出口块[[1]](https://llvm.org/docs/LoopTerminology.html" \l "lcssa-construction)，以便在循环内“close”这些值。特别是考虑以下循环：

|  |
| --- |
| c = ...;  for (...) {  if (c)  X1 = ...  else  X2 = ...  X3 = phi(X1, X2); // X3 defined  }  ... = X3 + 4; // X3 used, i.e. live  // outside the loop |

在内部循环中，X3 在循环内部定义，但在循环外部使用。在 Loop Closed SSA 形式中，这将表示如下：

|  |
| --- |
| c = ...;  for (...) {  if (c)  X1 = ...  else  X2 = ...  X3 = phi(X1, X2);  }  X4 = phi(X3);  ... = X4 + 4; |

这仍然是有效的 LLVM；额外的 phi 节点纯粹是多余的，但所有 LoopPass 都需要保留它们。这种形式由 LCSSA ([-lcssa](https://llvm.org/docs/Passes.html" \l "passes-lcssa) ) pass 确保，并在调度 LoopPass 时由 LoopPassManager 自动添加。循环优化完成后，这些额外的 phi 节点将被[-instcombine](https://llvm.org/docs/Passes.html" \l "passes-instcombine)删除。

请注意，退出块在循环之外，所以这样一个 phi 如何“close”循环内的值，因为它在循环外使用它？首先，对于 phi 节点，正如[LangRef](https://llvm.org/docs/LangRef.html" \l "id311) 中提到的：“每个传入值的使用被认为发生在从相应的前驱块到当前块的边上”。现在，到出口块的边被认为在循环之外，因为如果我们采用该边，它会明确地引导我们离开循环。

但是，边实际上不包含任何 IR，因此在源代码中，我们必须选择一个约定，即使用发生在当前块还是各自的前导块中。对于 LCSSA 的目的，我们认为使用发生在后者（以便考虑内部使用）[[2]](https://llvm.org/docs/LoopTerminology.html" \l "point-of-use-phis)。

LCSSA 的主要好处是它使许多其他循环优化变得更简单。

首先，一个简单的观察是，如果需要查看所有外部用户，他们可以迭代退出块中的所有（循环关闭）PHI 节点（另一种方法是扫描 def-use 链[[3 ]](https://llvm.org/docs/LoopTerminology.html" \l "def-use-chain)循环中的所有指令）。

然后，考虑例如 -loop-unswitch上面的循环。因为它是 LCSSA 形式，我们知道在循环内部定义的任何值将仅在循环内部或在循环关闭 PHI 节点中使用。在这种情况下，唯一的闭环 PHI 节点是 X4。这意味着我们可以只复制循环并相应地更改 X4，如下所示：

|  |
| --- |
| c = ...;  if (c) {  for (...) {  if (true)  X1 = ...  else  X2 = ...  X3 = phi(X1, X2);  }  } else {  for (...) {  if (false)  X1' = ...  else  X2' = ...  X3' = phi(X1', X2');  }  }  X4 = phi(X3, X3') |

现在，所有使用 X4 的都将得到更新后的值（一般来说，如果循环是 LCSSA 形式，在任何循环转换中，我们只需要更新循环关闭 PHI 节点，更改就会生效）。如果我们没有 Loop Closed SSA 形式，则意味着 X3 可能会在循环外使用。因此，我们必须引入 X4（即新的 X3）并用它替换 X3 的所有用途。然而，我们应该注意，因为 LLVM 为每个值保留了一个 def-use 链 [[3]](https://llvm.org/docs/LoopTerminology.html" \l "def-use-chain)，我们不需要执行数据流分析来查找和替换所有使用（有一个实用函数 replaceAllUsesWith() ，它通过迭代 def-use 链来执行此转换）。

另一个重要的优点是归纳变量的所有使用行为都是相同的。如果没有这个，您需要区分变量在其定义的循环之外使用的情况，例如：

|  |
| --- |
| for (i = 0; i < 100; i++) {  for (j = 0; j < 100; j++) {  k = i + j;  use(k); // use 1  }  use(k); // use 2  } |

从具有正常 SSA 形式的外部循环来看，k 的第一次使用表现不佳，而第二个是基数为 100 且步长为 1 的归纳变量。尽管在实践中和 LLVM 上下文中，这种情况SCEV 可以有效处理。标量演化（[scalar-evolution](https://llvm.org/docs/Passes.html" \l "passes-scalar-evolution)）或 SCEV(Scalar Evolution)，是一种（分析）pass，它分析和分类循环中标量表达式的演化。

通常，在 LCSSA 形式的循环中使用 SCEV 更容易。根据定义，SCEV 可以分析的标量（循环变量）表达式的演化与循环相关。表达式在 LLVM 中由 [llvm::Instruction](https://llvm.org/doxygen/classllvm_1_1Instruction.html)表示。如果表达式在两个（或更多）循环内（这只有在循环嵌套时才会发生，如上例所示）并且您想分析其演变（来自 SCEV），您还必须指定相对于你想要什么循环。具体来说，您必须使用[getSCEVAtScope()](https://llvm.org/doxygen/classllvm_1_1ScalarEvolution.html" \l "a21d6ee82eed29080d911dbb548a8bb68)。

但是，如果所有循环都是 LCSSA 形式，则每个表达式实际上由两个不同的 llvm::Instructions 表示。一个在循环内，一个在循环外，这是闭环 PHI 节点，表示最后一次迭代后表达式的值（实际上，我们将每个循环变量表达式分解为两个表达式，因此，每个表达式至多在一个循环内）。您现在可以只使用[getSCEV()](https://llvm.org/doxygen/classllvm_1_1ScalarEvolution.html" \l "a30bd18ac905eacf3601bc6a553a9ff49)。以及您传递给它的这两个 llvm::Instructions 中的哪一个消除了上下文/范围/相对循环的歧义。

脚注Footnotes

* [1] 要插入这些闭环 PHI 节点，必须（重新）计算优势边界(dominance frontiers)（如果循环有多个出口）。
* [2] 在各自的predecessor中考虑PHI 条目值的使用点，是整个 LLVM 的约定。原因主要是实际的；例如，它保留了 SSA 的优势属性。它也只是对实际使用次数的过度估计；传入的块可能会分支到另一个块，在这种情况下，该值实际上并未使用，但没有副作用（它可能会增加其生命周期，但这与 LCSSA 无关）。此外，如果我们考虑生命周期，我们可以获得一些直觉：PHI通常是插入到当前块，因为该值不能从此时起使用（即当前块是优势边界）。认为该值在当前块中使用（因为 PHI）是没有意义的，因为该值在 PHI 之前就不再有效。在某种意义上，PHI 定义只是“替换”了原始值定义，并没有实际使用它。需要强调的是，此类比仅作为示例，不做任何严格要求。例如，该值可能支配当前块，但我们仍然可以插入一个 PHI（就像我们对 LCSSA PHI 节点所做的那样）并 之后使用原始值（在这种情况下，两个生命范围重叠，尽管在 LCSSA 中（重点是那）我们从不那样做）。
* [3] ( 1 , 2 ) SSA 的一个属性是每个定义都存在一个 def-use 链，它是这个定义的所有使用的列表。LLVM 通过在内部数据结构中保留一个值的所有使用列表来实现此属性。

## “更规范的”循环“More Canonical” Loops

### 旋转循环Rotated Loops

循环由 LoopRotate ( [loop-rotate](https://llvm.org/docs/Passes.html" \l "passes-loop-rotate) ) pass 旋转，它将循环转换为 do/while 样式循环并在 [LoopRotation.h](https://llvm.org/doxygen/LoopRotation_8h_source.html)中实现。例子：

|  |
| --- |
| void test(int n) {  for (int i = 0; i < n; i += 1)  // Loop body  } |

转换为：

|  |
| --- |
| void test(int n) {  int i = 0;  do {  // Loop body  i += 1;  } while (i < n);  } |

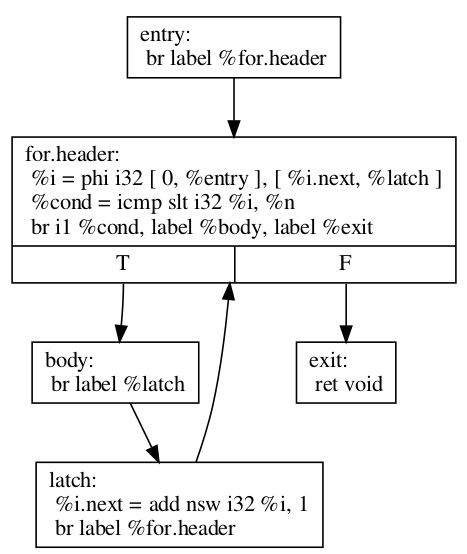
警告：只有当编译器可以证明循环体将至少执行一次时，此转换才有效。否则，它必须插入一个将在运行时对其进行测试的loop guard。在上面的示例中，这将是：

|  |
| --- |
| void test(int n) {  int i = 0;  if (n > 0) {  do {  // Loop body  i += 1;  } while (i < n);  }  } |

了解循环旋转在 LLVM IR 级别的影响很重要。我们遵循 LLVM IR 中的先前示例，同时还提供控制流图 (CFG) 的图形表示。您可以通过使用view-cfg pass获得相同的图形结果。

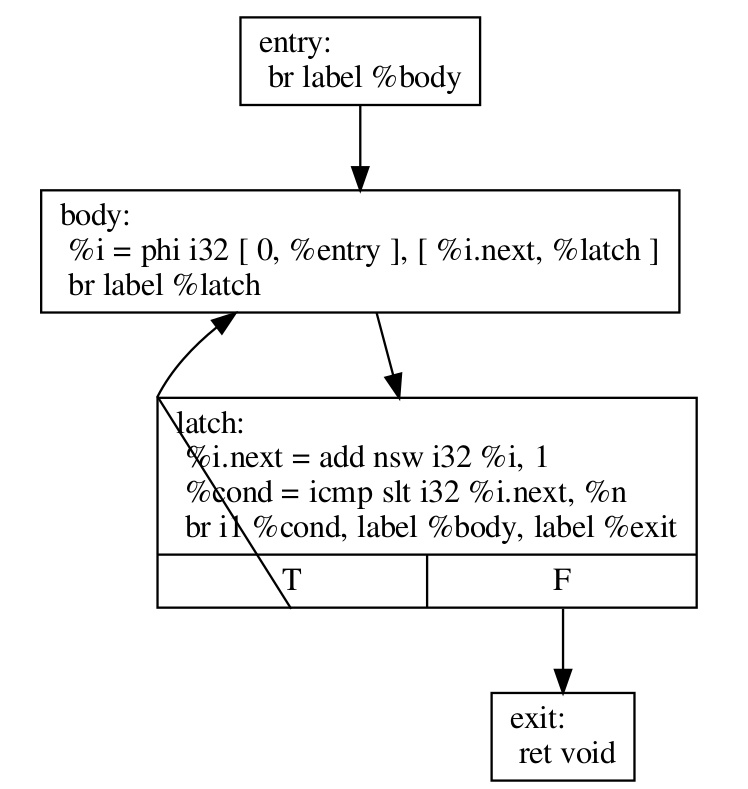
初始的 for循环可以翻译成：

|  |
| --- |
| define void @test(i32 %n) {  entry:  br label %for.header  for.header:  %i = phi i32 [ 0, %entry ], [ %i.next, %latch ]  %cond = icmp slt i32 %i, %n  br i1 %cond, label %body, label %exit  body:  ; Loop body  br label %latch  latch:  %i.next = add nsw i32 %i, 1  br label %for.header  exit:  ret void  } |



在我们解释 LoopRotate 将如何实际转换此循环之前，这里是我们如何（手动）将其转换为 do-while 样式循环的方法。

|  |
| --- |
| define void @test(i32 %n) {  entry:  br label %body  body:  %i = phi i32 [ 0, %entry ], [ %i.next, %latch ]  ; Loop body  br label %latch  latch:  %i.next = add nsw i32 %i, 1  %cond = icmp slt i32 %i.next, %n  br i1 %cond, label %body, label %exit  exit:  ret void  } |

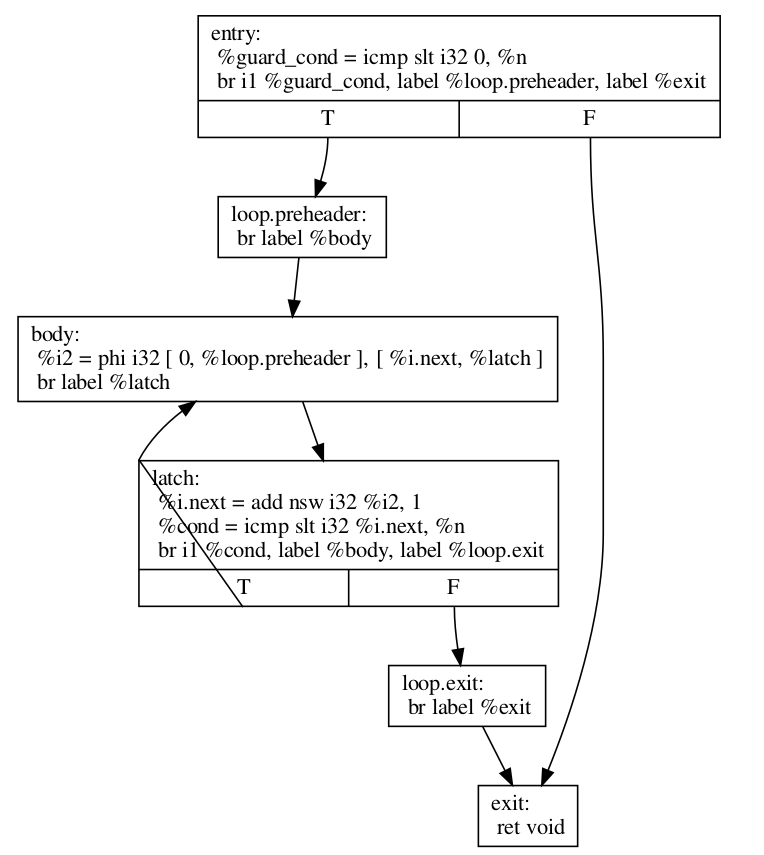


注意两点：

* 条件检查被移动到循环的“底部”，即latch。这是 LoopRotate 通过将循环的header复制到latch来完成的事情。
* 在这种情况下，编译器无法推断出循环肯定会至少执行一次，因此上述转换无效。如上所述，必须插入一个loopguard，这是 LoopRotate 会做的事情。

这就是 LoopRotate 转换此循环的方式：

|  |
| --- |
| entry:  %guard\_cond = icmp slt i32 0, %n  br i1 %guard\_cond, label %loop.preheader, label %exit  loop.preheader:  br label %body  body:  %i2 = phi i32 [ 0, %loop.preheader ], [ %i.next, %latch ]  br label %latch  latch:  %i.next = add nsw i32 %i2, 1  %cond = icmp slt i32 %i.next, %n  br i1 %cond, label %body, label %loop.exit  loop.exit:  br label %exit  exit:  ret void  } |



结果比我们预期的要复杂一点，因为 LoopRotate 确保循环 在旋转后处于[Loop Simplify Form](https://llvm.org/docs/LoopTerminology.html" \l "loop-terminology-loop-simplify)中。在这种情况下，它插入了 %loop.preheader 基本块，这样循环就有了一个预报头，它引入了 %loop.exit 基本块，这样循环就有了专用的出口（否则，%exit 会从 %latch 和%entry，但 %entry 不包含在循环中）。请注意，为了成功应用 LoopRotate，循环也必须事先处于 Loop Simplify Form 中。

这种形式的主要优点是它允许将不变指令（尤其是加载）提升到预报头中。这也可以在非旋转循环中完成，但有一些缺点。让我们用一个例子来说明它们：

|  |
| --- |
| for (int i = 0; i < n; ++i) {  auto v = \*p;  use(v);  } |

我们假设从 p 加载是不变的，并且 use(v) 是一些使用 v 的语句。如果我们只想执行一次加载，我们可以将它“移出”循环体，结果如下：

|  |
| --- |
| auto v = \*p;  for (int i = 0; i < n; ++i) {  use(v);  } |

但是，现在，在 n <= 0 的情况下，在初始形式中，循环体永远不会执行，因此，加载也永远不会执行。这是一个主要出于语义原因的问题。考虑 n <= 0 并且从 p 加载无效的情况。在最初的程序中不会有错误。然而，通过这种转换，我们将引入一个有效地打破初始语义的转换。

为了避免这两个问题，我们可以插入一个loop guard：

|  |
| --- |
| if (n > 0) { // loop guard  auto v = \*p;  for (int i = 0; i < n; ++i) {  use(v);  }  } |

这当然更好，但可以稍微改进。请注意，对 n 是否大于 0 的检查执行了两次（并且 n 在其间没有变化）。一次是在我们检查保护条件时，一次是在第一次执行循环时。为避免这种情况，我们可以进行无条件的首次执行并在最后插入循环条件。这实际上意味着将循环转换为 do-while 循环：

|  |
| --- |
| if (0 < n) {  auto v = \*p;  do {  use(v);  ++i;  } while (i < n);  } |

请注意，LoopRotate 通常不会进行此类提升。相反，它是为其他passes（如循环不变代码移动( [-licm](https://llvm.org/docs/Passes.html" \l "passes-licm) )）启用转换。

# LLVM 中的自动矢量化

LLVM 有两个矢量化器：在循环上运行的[Loop Vectorizer](https://llvm.org/docs/Vectorizers.html" \l "loop-vectorizer)和[SLP Vectorizer](https://llvm.org/docs/Vectorizers.html" \l "slp-vectorizer)。这些矢量器专注于不同的优化机会并使用不同的技术。SLP 矢量器将代码中找到的多个标量合并到矢量中，而循环矢量器扩展循环中的指令以对多个连续迭代进行操作。

Loop Vectorizer 和 SLP Vectorizer 都默认启用。

## [Loop Vectorizer](https://llvm.org/docs/Vectorizers.html" \l "loop-vectorizer)

### 用法

Loop Vectorizer 默认启用，但可以使用命令行标志通过 clang 禁用它：

|  |
| --- |
| $ clang ... -fno-vectorize file.c |

#### 命令行标志

循环矢量化器使用成本模型来决定最佳矢量化因子和展开因子。但是，矢量化器的用户可以强制矢量化器使用特定值。'clang' 和 'opt' 都支持下面的标志。

用户可以使用命令行标志“-force-vector-width”控制矢量化 SIMD 宽度。

|  |
| --- |
| $ clang -mllvm -force-vector-width=8 ...  $ opt -loop-vectorize -force-vector-width=8 ... |

用户可以使用命令行标志“-force-vector-interleave”控制展开因子

|  |
| --- |
| $ clang -mllvm -force-vector-interleave=2 ...  $ opt -loop-vectorize -force-vector-interleave=2 ... |

#### Pragma 循环提示指令

该指令允许为后续的 for、while、do-while 或基于 c++11 范围的 for 循环指定循环矢量化提示。该指令允许启用或禁用矢量化和交错。矢量宽度和交错计数也可以手动指定。以下示例明确启用矢量化和交错：

|  |
| --- |
| #pragma clang loop vectorize(enable) interleave(enable)  while(...) {  ...  } |

以下示例通过指定向量宽度和交错计数隐式启用向量化和交错：

|  |
| --- |
| #pragma clang loop vectorize\_width(2) interleave\_count(2)  for(...) {  ...  } |

有关详细信息，请参阅 [Clang 语言扩展](https://clang.llvm.org/docs/LanguageExtensions.html" \l "extensions-for-loop-hint-optimizations) 。

### 诊断

许多循环无法向量化，包括具有复杂控制流、不可向量化类型和不可向量化调用的循环。循环向量化器生成优化注释，可以使用命令行选项查询这些优化注释，以识别和诊断被循环向量化器跳过的循环。

优化备注启用使用：

* -Rpass=loop-vectorize标识已成功向量化的循环。
* -Rpass-missed=loop-vectorize标识矢量化失败的循环并指示是否指定了矢量化。
* -Rpass-analysis=loop-vectorize标识导致向量化失败的语句。如果另外提供-fsave-optimization-record，则可能会列出矢量化失败的多种原因（此行为将来可能会改变）。

考虑以下循环：

|  |
| --- |
| #pragma clang loop vectorize(enable)  for (int i = 0; i < Length; i++) {  switch(A[i]) {  case 0: A[i] = i\*2; break;  case 1: A[i] = i; break;  default: A[i] = 0;  }  } |

命令行-Rpass-missed=loop-vectorize打印备注：

|  |
| --- |
| no\_switch.cpp:4:5: remark: loop not vectorized: vectorization is explicitly enabled [-Rpass-missed=loop-vectorize] |

并且命令行-Rpass-analysis=loop-vectorize提示switch语句不能向量化。

|  |
| --- |
| no\_switch.cpp:4:5: remark: loop not vectorized: loop contains a switch statement [-Rpass-analysis=loop-vectorize]  switch(A[i]) {  ^ |

为确保生成行号和列号，请包括命令行选项 -gline-tables-only和-gcolumn-info. 有关详细信息，请参阅 Clang[用户手册](https://clang.llvm.org/docs/UsersManual.html" \l "options-to-emit-optimization-reports)

### 特点

LLVM Loop Vectorizer 具有许多功能，允许它对复杂循环进行矢量化。

#### 行程次数(tripcount)未知的循环

Loop Vectorizer 支持具有未知行程计数的循环。在下面的循环中，迭代start和finish点是未知的，并且 Loop Vectorizer 有一种机制来向量化不从零开始的循环。在此示例中，“n”可能不是矢量宽度的倍数，矢量化器必须将最后几次迭代作为标量代码执行。保留循环的标量副本会增加代码大小。

|  |
| --- |
| void bar(float \*A, float\* B, float K, int start, int end) {  for (int i = start; i < end; ++i)  A[i] \*= B[i] + K;  } |

#### 指针的运行时检查

在下面的示例中，如果指针 A 和 B 指向连续的地址，那么向量化代码是非法的，因为 A 的某些元素将在从数组 B 读取之前被写入。

一些程序员使用“restrict”关键字来通知编译器指针不相交，但在我们的示例中，Loop Vectorizer 无法知道指针 A 和 B 是唯一的。Loop Vectorizer 通过放置代码来处理此循环，该代码在运行时检查数组 A 和 B 是否指向脱节的内存位置。如果数组 A 和 B 重叠，则执行循环的标量版本。

|  |
| --- |
| void bar(float \*A, float\* B, float K, int n) {  for (int i = 0; i < n; ++i)  A[i] \*= B[i] + K;  } |

#### 减少Reductions

在此示例中，sum变量由循环的连续迭代使用。通常，这会阻止向量化，但向量化器可以检测到“sum”是一个Reduction variable。变量“sum”变成一个整数向量，在循环结束时将数组的元素相加以创建正确的结果。我们支持多种不同的Reduction 运算，例如加法、乘法、XOR、AND 和 OR。？？？，reduction可以翻译成降维吗？

|  |
| --- |
| int foo(int \*A, int n) {  unsigned sum = 0;  for (int i = 0; i < n; ++i)  sum += A[i] + 5;  return sum;  } |

当使用-ffast-math时，我们支持浮点Reduction 运算。

#### 归纳Inductions

在这个例子中，归纳变量 i 的值被保存到一个数组中。Loop Vectorizer 知道对归纳变量进行向量化。

|  |
| --- |
| void bar(float \*A, int n) {  for (int i = 0; i < n; ++i)  A[i] = i;  } |

#### 如果转换

Loop Vectorizer 能够“扁平化”代码中的 IF 语句并生成单个指令流。Loop Vectorizer 支持最内层循环中的任何控制流。最内层的循环可能包含 IF、ELSE 甚至 GOTO 的复杂嵌套。

|  |
| --- |
| int foo(int \*A, int \*B, int n) {  unsigned sum = 0;  for (int i = 0; i < n; ++i)  if (A[i] > B[i])  sum += A[i] + 5;  return sum;  } |

#### 指针归纳变量

此示例使用标准 c++ 库的“accumulate”函数。这个循环使用 C++ 迭代器，它是指针，而不是整数索引。Loop Vectorizer 检测指针归纳变量并可以对该循环进行矢量化。这个特性很重要，因为许多 C++ 程序都使用迭代器。

|  |
| --- |
| int baz(int \*A, int n) {  return std::accumulate(A, A + n, 0);  } |

#### 反向迭代器

Loop Vectorizer 可以矢量化从后向前计数的循环。

|  |
| --- |
| void foo(int \*A, int n) {  for (int i = n; i > 0; --i)  A[i] +=1;  } |

#### 分散/聚集Scatter / Gather

Loop Vectorizer 可以将代码矢量化，这些代码成为分散/聚集内存的标量指令序列。

|  |
| --- |
| void foo(int \* A, int \* B, int n) {  for (intptr\_t i = 0; i < n; ++i)  A[i] += B[i \* 4];  } |

在许多情况下，成本模型会通知 LLVM 这没有好处，并且 LLVM 只会在使用“-mllvm -force-vector-width=#”强制时矢量化此类代码。

#### 混合类型的向量化

Loop Vectorizer 可以矢量化具有混合类型的程序。Vectorizer 成本模型可以估计类型转换的成本并决定向量化是否有利可图。

|  |
| --- |
| void foo(int \*A, char \*B, int n) {  for (int i = 0; i < n; ++i)  A[i] += 4 \* B[i];  } |

#### 全局结构别名分析

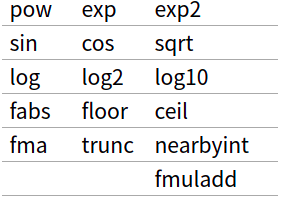
对全局结构的访问也可以被矢量化，使用别名分析来确保访问不会出现别名。还可以在对结构成员的指针访问上添加运行时检查。

支持许多变体，但一些依赖未定义行为的被忽略（如其他编译器所做的那样）仍然未被矢量化。

|  |
| --- |
| struct { int A[100], K, B[100]; } Foo;  void foo() {  for (int i = 0; i < 100; ++i)  Foo.A[i] = Foo.B[i] + 100;  } |

#### 函数调用的向量化

Loop Vectorizer 可以向量化内部数学函数。有关这些功能的列表，请参见下表。



请注意，如果库调用访问外部状态（例如“errno”），优化器可能无法向量化与这些内在函数相对应的数学库函数。为了更好地优化 C/C++ 数学库函数，请使用“-fno-math-errno”。

循环向量化器了解目标上的特殊指令，并将向量化包含映射到指令的函数调用的循环。例如，如果 SSE4.1 roundps 指令可用，下面的循环将在 Intel x86 上进行矢量化。

|  |
| --- |
| void foo(float \*f) {  for (int i = 0; i != 1024; ++i)  f[i] = floorf(f[i]);  } |

#### 矢量化过程中的部分展开

现代处理器具有多个执行单元，只有包含高度并行性的程序才能充分利用机器的整个宽度。Loop Vectorizer 通过执行循环的部分展开来提高指令级并行性 (ILP)。

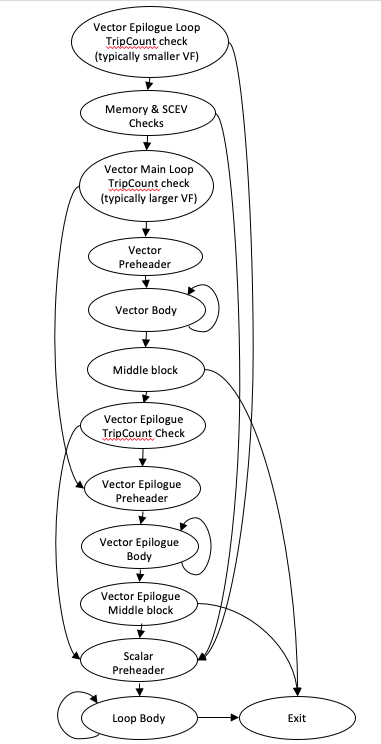
在下面的示例中，整个数组累积到变量“sum”中。这是低效的，因为处理器只能使用一个执行端口。通过展开代码，Loop Vectorizer 允许同时使用两个或多个执行端口。

|  |
| --- |
| int foo(int \*A, int n) {  unsigned sum = 0;  for (int i = 0; i < n; ++i)  sum += A[i];  return sum;  } |

Loop Vectorizer 使用成本模型来决定何时展开循环有利可图。展开循环的决定取决于寄存器压力和生成的代码大小。

#### 结语矢量化Epilogue Vectorization

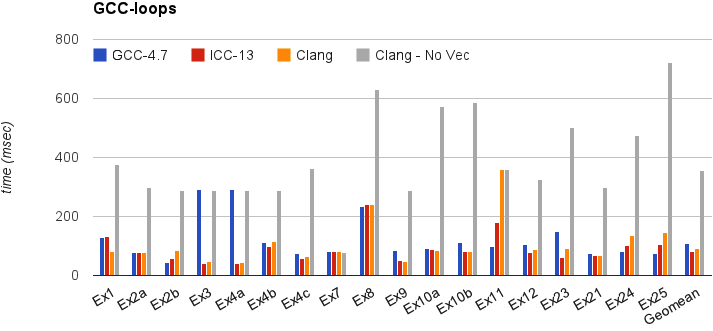
对循环进行矢量化时，如果循环行程计数未知或未平均划分矢量化和展开因子，则通常需要标量余数（尾声）循环来执行循环的尾部迭代。当向量化和展开因子很大时，具有较小行程计数的循环可能最终将大部分时间花在标量（而不是向量）代码中。为了解决这个问题，内部循环矢量化器增强了一项功能，允许它使用矢量化和展开因子组合来矢量化尾声循环，这使得小行程计数循环更有可能仍然在矢量化代码中执行。下图显示了带有运行时检查的典型结语矢量化循环的 CFG。



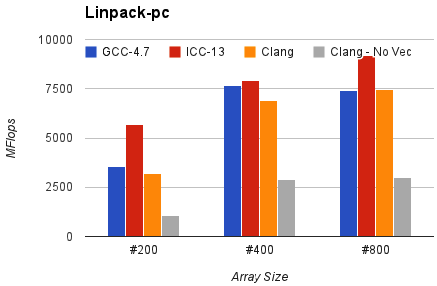
### 性能

本节展示了 Clang 在一个简单的基准测试中的执行时间： [gcc-loops](https://github.com/llvm/llvm-test-suite/tree/main/SingleSource/UnitTests/Vectorizer)。该基准测试是Dorit Nuzman的 GCC 自动矢量化[页面](http://gcc.gnu.org/projects/tree-ssa/vectorization.html)中的循环集合。

下图比较了 GCC-4.7、ICC-13 和 Clang-SVN，在 -O3 下使用和不使用循环矢量化，针对“corei7-avx”进行了调整，在 Sandybridge iMac 上运行。Y 轴显示以毫秒为单位的时间。越低越好。最后一列显示所有内核的几何平均数。



和linpack-pc配置一样。结果是 Mflops，越高越好。



### 持续发展方向

[矢量化计划](https://llvm.org/docs/Proposals/VectorizationPlan.html)

对过程建模并升级 LLVM 的循环矢量化器的基础结构。

## SLP 向量化

### 详情

SLP 向量化（又名超字级并行，superword-level parallelism）的目标是将相似的独立指令组合成向量指令。内存访问、算术运算、比较运算、PHI 节点都可以使用这种技术进行矢量化。

例如，以下函数对其输入 (a1, b1) 和 (a2, b2) 执行非常相似的操作。基本块向量化器可以将这些组合成向量操作。

|  |
| --- |
| void foo(int a1, int a2, int b1, int b2, int \*A) {  A[0] = a1\*(a1 + b1);  A[1] = a2\*(a2 + b2);  A[2] = a1\*(a1 + b1);  A[3] = a2\*(a2 + b2);  } |

SLP 向量化器自下而上地跨基本块处理代码，以搜索要组合的标量。

### 用法

默认情况下启用 SLP Vectorizer，但可以使用命令行标志通过 clang 将其禁用：

|  |
| --- |
| $ clang -fno-slp-vectorize file.c |

# LLVM 链接时间优化：设计与实现

## 描述

LLVM 具有强大的模块间优化功能，可在链接时使用。链接时间优化 (LTO) 是在链接阶段执行的模块间优化的另一个名称。本文档描述了 LTO 优化器和链接器之间的接口和设计。

## 设计理念

LLVM 链接时间优化器在编译器工具链中提供完全透明，同时进行模块间优化。它的主要目标是让开发人员利用模块间优化，而无需对开发人员的 makefile 或构建系统进行任何重大更改。这是通过与链接器的紧密集成来实现的。在此模型中，链接器将 LLVM 位码文件视为本机目标文件，并允许在它们之间进行混合和匹配。链接器使用共享对象[libLTO](https://llvm.org/docs/LinkTimeOptimization.html" \l "liblto) 来处理 LLVM 位码文件。链接器和 LLVM 优化器之间的这种紧密集成有助于进行其他模型无法实现的优化。链接器输入允许优化器避免依赖保守的逃逸分析。

### 链接时间优化示例

以下示例说明了 LTO 集成方法和简洁界面的优势。此示例需要一个系统链接器，它通过本文档中描述的接口支持 LTO。在这里，clang 透明地调用系统链接器。

* 输入源文件a.c被编译成 LLVM 位码形式。
* 输入源文件main.c被编译成本机目标代码。

|  |
| --- |
| --- a.h ---  extern int foo1(void);  extern void foo2(void);  extern void foo4(void);  --- a.c ---  #include "a.h"  static signed int i = 0;  void foo2(void) {  i = -1;  }  static int foo3() {  foo4();  return 10;  }  int foo1(void) {  int data = 0;  if (i < 0)  data = foo3();  data = data + 42;  return data;  }  --- main.c ---  #include <stdio.h>  #include "a.h"  void foo4(void) {  printf("Hi\n");  }  int main() {  return foo1();  } |

要编译，请运行：

|  |
| --- |
| % clang -flto -c a.c -o a.o # <-- a.o is LLVM bitcode file  % clang -c main.c -o main.o # <-- main.o is native object file  % clang -flto a.o main.o -o main # <-- standard link command with -flto |

说明：

* 在此示例中，链接器将foo2()识别为 LLVM 位码文件中定义的外部可见符号。链接器完成其通常的符号解析过程并发现它没有在任何地方使用foo2()。此信息由 LLVM 优化器使用并删除foo2().
* 一旦foo2()删除，优化器就会识别出条件.i < 0始终为假，这意味着foo3()永远不会使用。因此，优化器也会删除foo3()。
* 而这反过来又使链接器能够删除foo4().

此示例说明了与链接器紧密集成的优势。在这里，优化器无法在没有链接器输入的情况下删除foo3()。

### 替代方法

* 编译器驱动程序单独调用链接时间优化器。

在此模型中，链接时间优化器无法利用在链接器的正常符号解析阶段收集的信息。在上面的示例中，优化器无法在没有链接器输入的情况下删除foo2()，因为它是外部可见的。这反过来又禁止优化器删除foo3().

* 使用单独的工具从所有目标文件中收集符号信息。

在这个模型中，一个新的、独立的工具或库复制了链接器收集信息以优化链接时间的能力。这种代码重复不仅难以证明其合理性，而且还有其他几个缺点。例如，链接语义和链接器在各种平台上提供的功能并不是唯一的。这意味着，这个新工具需要在一个超级工具中支持所有这些功能和平台，或者每个平台需要一个单独的工具。这显着增加了链接时间优化器的维护成本，这是不必要的。这种方法还需要与各种平台上的链接器开发保持同步，这不是链接时间优化器的主要关注点。最后，这种方法增加了最终用户的构建时间，因为这种方法所做的工作重复。

## libLTO和链接器之间的多阶段通信

链接器收集有关符号定义和在各种链接对象中使用的信息，这些信息比其他工具在典型构建周期中收集的任何信息都更准确。链接器通过查看本机 .o 文件中符号的定义和使用以及使用符号可见性信息来收集此信息。链接器还使用用户提供的信息，例如导出符号列表。LLVM 优化器收集控制流信息、数据流信息，并从优化器的角度了解更多有关程序结构的信息。我们的目标是通过在各个链接阶段共享此信息来利用链接器和优化器之间的紧密集成。

### 阶段 1：读取 LLVM 位码文件

链接器首先按自然顺序读取所有目标文件并收集符号信息。这包括本机对象文件以及 LLVM 位码文件。为了在所有 .o 文件都是本机对象文件的情况下最小化链接器的成本，链接器仅在发现提供的对象文件不是本机对象文件时调用lto\_module\_create()。如果 lto\_module\_create() 返回该文件是一个 LLVM 位码文件，则链接器然后使用lto\_module\_get\_symbol\_name()和lto\_module\_get\_symbol\_attribute() 遍历模块以获取所有已定义和引用的符号。该信息被添加到链接器的全局符号表中。

lto\* 函数全部在共享对象 libLTO 中实现。这允许独立于链接器工具更新 LLVM LTO 代码。在支持它的平台上，共享对象被延迟加载。

### 第 2 阶段：符号解析

在此阶段，链接器使用全局符号表解析符号。它可能会报告未定义的符号错误、读取存档成员、替换弱符号等。链接器能够无缝地执行此操作，即使它不知道输入 LLVM 位码文件的确切内容。如果启用死代码剥离，则链接器会收集活动符号列表。

### 第 3 阶段：优化 Bitcode 文件

符号解析后，链接器告诉 LTO 共享目标文件本机目标文件需要哪些符号。在上面的示例中，通过使用 lto\_codegen\_add\_must\_preserve\_symbol()，链接器报告仅foo1()被本机目标文件使用。接下来，链接器通过使用 lto\_codegen\_compile()，调用 LLVM 优化器和代码生成器，它返回一个本地对象文件，该文件通过合并 LLVM 位码文件和应用各种优化passes创建。

### 阶段 4：优化后的符号解析

在此阶段，链接器读取优化的本机目标文件并更新内部全局符号表以反映任何更改。链接器还收集有关 LLVM 位码文件使用外部符号的任何更改的信息。在上面的示例中，链接器指出foo4()不再使用。如果启用死代码剥离，则链接器会适当地刷新活动符号信息并执行死代码剥离。

在此阶段之后，链接器继续链接，就好像它从未见过 LLVM 位码文件一样。

## libLTO

libLTO是一个共享对象，它是 LLVM 工具的一部分，旨在供链接器使用。libLTO提供了一个抽象的 C 接口来使用 LLVM 过程间优化器，而无需暴露 LLVM 内部细节。目的是保持接口尽可能稳定，即使 LLVM 优化器继续发展。完全不同的编译技术甚至应该有可能提供不同的 libLTO 来处理它们的目标文件和标准链接器工具。

### lto\_module\_t

非本机对象文件通过lto\_module\_t 来处理。 以下函数允许链接器检查文件（在磁盘上或内存缓冲区中）是否是 libLTO 可以处理的文件：

|  |
| --- |
| lto\_module\_is\_object\_file(const char\*)  lto\_module\_is\_object\_file\_for\_target(const char\*, const char\*)  lto\_module\_is\_object\_file\_in\_memory(const void\*, size\_t)  lto\_module\_is\_object\_file\_in\_memory\_for\_target(const void\*, size\_t, const char\*) |

如果目标文件可以由libLTO 处理，则链接器使用以下之一创建一个 lto\_module\_t：

|  |
| --- |
| lto\_module\_create(const char\*)  lto\_module\_create\_from\_memory(const void\*, size\_t) |

完成后，句柄通过如下接口释放

|  |
| --- |
| lto\_module\_dispose(lto\_module\_t) |

链接器可以通过获取符号的数量并通过以下方式获取每个符号的名称和属性来内省(introspect)非本机目标文件：

|  |
| --- |
| lto\_module\_get\_num\_symbols(lto\_module\_t)  lto\_module\_get\_symbol\_name(lto\_module\_t, unsigned int)  lto\_module\_get\_symbol\_attribute(lto\_module\_t, unsigned int) |

符号的属性包括对齐、可见性和种类。

在 Darwin 上处理目标文件的工具（例如 lipo）可能需要知道 CPU 类型等属性：

|  |
| --- |
| lto\_module\_get\_macho\_cputype(lto\_module\_t mod, unsigned int \*out\_cputype, unsigned int \*out\_cpusubtype) |

### lto\_code\_gen\_t

一旦链接器将每个非本机目标文件加载到一个 lto\_module\_t中，它就可以请求libLTO处理所有这些文件并生成一个本机目标文件。这是通过几个步骤完成的。

首先，创建一个代码生成器：

|  |
| --- |
| lto\_codegen\_create() |

然后，将每个非本地目标文件添加到代码生成器中：

|  |
| --- |
| lto\_codegen\_add\_module(lto\_code\_gen\_t, lto\_module\_t) |

然后链接器可以选择设置一些代码生成选项。是否生成 DWARF 调试信息设置为：

|  |
| --- |
| lto\_codegen\_set\_debug\_model(lto\_code\_gen\_t) |

设置哪种位置独立性：

|  |
| --- |
| lto\_codegen\_set\_pic\_model(lto\_code\_gen\_t) |

每个由本机目标文件引用的符号或不得优化掉的每个符号都设置为：

|  |
| --- |
| lto\_codegen\_add\_must\_preserve\_symbol(lto\_code\_gen\_t, const char\*) |

完成所有这些设置后，链接器请求使用以下设置从模块创建本机对象文件：

|  |
| --- |
| lto\_codegen\_compile(lto\_code\_gen\_t, size\*) |

它返回指向包含生成的本机目标文件的缓冲区的指针。然后链接器对其进行解析并将其与其余的本机对象文件链接。

# LLVM 黄金插件

## 简介

使用链接时间优化进行构建需要系统链接器的合作。Linux 系统上的 LTO 支持可通过[gold 链接器](http://sourceware.org/binutils)获得，该链接器通过插件支持 LTO。这与[GCC LTO](http://gcc.gnu.org/wiki/LinkTimeOptimization)项目使用的机制相同。

LLVM gold 插件在[libLTO](https://llvm.org/docs/LinkTimeOptimization.html" \l "liblto)之上实现 gold 插件接口 。其他工具也可以使用相同的插件，例如 ar和nm。请注意，binutils 版本 2.21.51.0.2 及更高版本中的 ld.bfd 也通过插件支持 LTO。但是，未测试 LLVM gold 插件与 ld.bfd 的结合使用，因此未得到官方支持或推荐。

从 LLVM 15 开始，gold 插件将忽略 ELF 目标文件内部的.llvmbc section的位码。但是，仍然支持带有位码文件的 LTO。

## 如何构建它

您需要拥有插件支持的gold 并构建 LLVMgold 插件。gold 链接器安装为 ld.gold。要查看 gold 是否是您系统上的默认值，请运行“/usr/bin/ld -v”.。 如果不是，它将报告“GNU gold”或“GNU ld”。如果 gold 已经安装在 /usr/bin/ld.gold，一种选择是通过备份您现有的/usr/bin/ld，并创建一个符号链接ln -s /usr/bin/ld.gold /usr/bin/ld来简单地将其设置为默认值。或者，您可以使用 clang -fuse-ld=gold 构建或添加 -fuse-ld=gold/usr/bin/ld.gold 到 LDFLAGS，这将导致 clang 驱动程序直接调用。

如果您安装了 gold，请通过运行/usr/bin/ld.gold -plugin 检查插件支持 。如果它抱怨“missing argument”，那么你有插件支持。如果没有，并且您会收到诸如“unknown option”之类的错误，那么您将需要构建 gold 或安装具有插件支持的版本。

* 使用插件支持下载、配置和构建 gold：

|  |
| --- |
| $ git clone --depth 1 git://sourceware.org/git/binutils-gdb.git binutils  $ mkdir build  $ cd build  $ ../binutils/configure --enable-gold --enable-plugins --disable-werror  $ make all-gold |

那应该留给您build/gold/ld-new ，它支持该-plugin选项。运行make将另外构建 build/binutils/ar和nm-new 二进制支持插件文件。

一旦您准备好切换到使用 gold，请备份您现有的/usr/bin/ld， 然后将其替换为ld-new. 或者，如前所述安装/usr/bin/ld.gold和使用-fuse-ld=gold。

可选地，添加--enable-gold=default到上面的配置调用以自动make install安装新构建的 gold。

* 构建 LLVMgold 插件。运行CMake 带有 -DLLVM\_BINUTILS\_INCDIR=/path/to/binutils/include。正确的包含路径将包含文件plugin-api.h.



## 用法

clang您应该使用选项 -flto 生成位码文件。此标志还将导致clang在其前缀下的lib目录中查找 gold 插件并将lib-plugin选项传递给 ld。. 它不会寻找没有 -fuse-ld=gold 的备用链接器，这就是为什么您需要 gold 作为路径中已安装的系统链接器的原因。

ar并且nm也接受该-plugin选项，并且可以安装LLVMgold.so到/usr/lib/bfd-plugins以进行无缝设置。如果您构建了自己的 gold ，请务必安装ar 和nm-new到您构建的/usr/bin.

## 链接时间优化示例

以下示例显示了 gold 插件混合 LLVM 位码和本机代码的工作示例。

代码

|  |
| --- |
| --- a.c ---  #include <stdio.h>  extern void foo1(void);  extern void foo4(void);  void foo2(void) {  printf("Foo2\n");  }  void foo3(void) {  foo4();  }  int main(void) {  foo1();  }  --- b.c ---  #include <stdio.h>  extern void foo2(void);  void foo1(void) {  foo2();  }  void foo4(void) {  printf("Foo4");  } |

命令

|  |
| --- |
| --- command lines ---  $ clang -flto a.c -c -o a.o # <-- a.o is LLVM bitcode file  $ ar q a.a a.o # <-- a.a is an archive with LLVM bitcode  $ clang b.c -c -o b.o # <-- b.o is native object file  $ clang -flto a.a b.o -o main # <-- link with LLVMgold plugin |

Gold 通知插件 foo3 永远不会在 IR 之外引用，导致 LLVM 删除该函数。然而，与libLTO 示例中不同的是， gold 当前并未消除 foo4。

## 在自动工具项目中使用 LTO 的快速入门

一旦您的系统ld、ar和nm都支持 LLVM 位码，一切就绪，可以轻松使用自动工具项目的 LTO 构建：

* 按照有关如何构建 [LLVMgold.so](https://llvm.org/docs/GoldPlugin.html" \l "lto-how-to-build)的说明进行操作。
* 安装新构建的 binutils 到$PREFIX
* 复制Release/lib/LLVMgold.so到$PREFIX/lib/bfd-plugins/
* 设置环境变量（$PREFIX是你安装 clang 和 binutils 的地方）：

|  |
| --- |
| export CC="$PREFIX/bin/clang -flto"  export CXX="$PREFIX/bin/clang++ -flto"  export AR="$PREFIX/bin/ar"  export NM="$PREFIX/bin/nm"  export RANLIB=/bin/true #ranlib is not needed, and doesn't support .bc files in .a |

* 或者你可以只设置你的路径：

|  |
| --- |
| export PATH="$PREFIX/bin:$PATH"  export CC="clang -flto"  export CXX="clang++ -flto"  export RANLIB=/bin/true |

* 像往常一样配置和构建项目：

|  |
| --- |
| % ./configure && make && make check |

环境变量设置也适用于非自动工具项目，但您可能还需要设置LD环境变量。

## 许可

黄金是根据 GPLv3 获得许可的。LLVMgold 使用 plugin-api.hgold 的接口文件，这意味着生成的LLVMgold.so 二进制文件也是 GPLv3。这仍然可以用于链接非 GPLv3 程序，就像没有插件的 gold 一样。

# 使用 LLVM 进行源代码级调试

## 简介

本文档是与 LLVM 中的调试信息相关的所有信息的中央存储库。它描述了[LLVM 调试信息采用的实际格式](https://llvm.org/docs/SourceLevelDebugging.html" \l "format)，这对那些有兴趣创建前端或直接处理信息的人很有用。此外，本文档还提供了 C/C++ 调试信息的具体示例。

### LLVM 调试信息背后的哲学

LLVM 调试信息的想法是捕获源语言的抽象语法树的重要部分如何映射到 LLVM 代码。几个设计方面塑造了此处出现的解决方案。重要的是：

* 调试信息应该对编译器的其余部分影响很小。调试信息不能修改任何转换、分析或代码生成器。
* LLVM 优化应该以[定义明确且易于描述的方式](https://llvm.org/docs/SourceLevelDebugging.html" \l "intro-debugopt)与调试信息进行交互。
* 因为 LLVM 旨在支持任意编程语言，所以 LLVM 到 LLVM 工具不需要知道任何关于源级语言语义的信息。
* 源代码级语言通常彼此之间存在很大差异。LLVM 不应该对源语言的风格施加任何限制，调试信息应该适用于任何语言。
* 有了代码生成器支持，应该可以使用 LLVM 编译器将程序编译为本机机器代码和标准调试格式。这允许与传统的机器代码级调试器（如 GDB 或 DBX）兼容。

LLVM 实现使用的方法是使用一小组 [内部函数](https://llvm.org/docs/SourceLevelDebugging.html" \l "format-common-intrinsics)来定义 LLVM 程序对象和源代码级对象之间的映射。源代码级程序的描述以 [实现定义的格式](https://llvm.org/docs/SourceLevelDebugging.html" \l "ccxx-frontend)保存在 LLVM 元数据中（C/C++ 前端目前使用DWARF 3 标准的工作草案 7 ）。

调试程序时，调试器与用户交互并将存储的调试信息转换为源语言特定信息。因此，调试器必须知道源语言，因此与特定语言或语言家族相关联。

### 调试信息消费者Debug information consumers

调试信息的作用是提供通常在编译过程中剥离的元信息。此元信息为 LLVM 用户提供了生成代码与原始程序源代码之间的关系。

目前，调试信息有两个后端消费者：DwarfDebug 和 CodeViewDebug。DwarfDebug 生成适用于 GDB、LLDB 和其他基于 DWARF 的调试器的 DWARF。CodeViewDebug生成 CodeView，即 Microsoft 调试信息格式，可用于 Visual Studio 和 WinDBG 等 Microsoft 调试器。LLVM 的调试信息格式大部分源自 DWARF 并受其启发，但转换为其他目标调试信息格式（如 STAABS）是可行的。

使用调试信息来提供用于分析生成的代码的分析工具，或者用于从生成的代码重建原始源代码的工具也是合理的。

## 调试信息和优化

LLVM 调试信息的一个极高优先级是使其与优化和分析良好交互。特别是，LLVM 调试信息提供以下保证：

* LLVM 调试信息始终提供信息以准确读取程序的源代码级状态，无论运行了哪些 LLVM 优化。[How to Update Debug Info: A Guide for LLVM Pass Authors](https://llvm.org/docs/HowToUpdateDebugInfo.html)指定了如何在各种代码转换中更新调试信息以避免破坏此保证，以及如何尽可能多地保留有用的调试信息。请注意，某些优化可能会影响使用调试器修改程序当前状态的能力，例如设置程序变量或调用已删除的函数。
* 根据需要，可以升级 LLVM 优化以了解调试信息，从而允许它们在执行积极优化时更新调试信息。这意味着，通过努力，LLVM 优化器可以像优化非调试代码一样优化调试代码。
* LLVM 调试信息不会阻止优化的发生（例如内联、基本块重新排序/合并/清理、尾部复制等）。
* LLVM 调试信息与程序的其余部分一起使用现有工具自动优化。例如，链接器会自动合并重复信息，自动删除未使用的信息。

基本上，调试信息允许您使用“ -O0 -g”编译程序并获得完整的调试信息，允许您在程序从调试器执行时任意修改它。使用“ -O3 -g”编译程序会为您提供始终可用且准确的完整调试信息（例如，尽管尾调用消除和内联，您仍可获得准确的堆栈跟踪），但您可能会失去修改程序和调用函数的能力（函数可能已经从程序中优化掉了），或完全内联。

[LLVM 测试套件](https://llvm.org/docs/TestSuiteMakefileGuide.html)提供了一个框架来测试优化器对调试信息的处理。它可以像这样运行：

|  |
| --- |
| % cd llvm/projects/test-suite/MultiSource/Benchmarks # or some other level  % make TEST=dbgopt |

这将测试调试信息对优化过程的影响。如果调试信息影响优化通过，那么它将被报告为失败。有关 LLVM 测试基础架构以及如何运行各种测试的更多信息，请参阅LLVM 测试基础架构指南。

## 调试信息格式

LLVM 调试信息经过精心设计，使优化器无需了解调试信息就可以优化程序和调试信息。特别是，元数据的使用从一开始就避免了重复的调试信息，如果决定删除函数，全局死代码消除pass 会自动删除该函数的调试信息。

为此，大多数调试信息（类型、变量、函数、源文件等的描述符）由语言前端以 LLVM 元数据的形式插入。

调试信息旨在与目标调试器和调试信息表示（例如 DWARF/Stabs/等）无关。它使用通用 pass来解码表示变量、类型、函数、命名空间等的信息：这允许使用任意源语言语义和类型系统，只要有为目标调试器编写的模块即可解释信息。

为了提供基本功能，LLVM 调试器确实必须对正在调试的源代码级语言做出一些假设，尽管它会将这些假设保持在最低限度。LLVM 调试器假定存在的唯一共同特征是[源文件](https://llvm.org/docs/LangRef.html" \l "difile)和[程序对象](https://llvm.org/docs/LangRef.html" \l "diglobalvariable)。调试器使用这些抽象对象来形成堆栈跟踪、显示有关局部变量的信息等。

文档的这一部分首先描述了任何源语言共有的表示方面。 [C/C++ 前端特定调试信息](https://llvm.org/docs/SourceLevelDebugging.html" \l "ccxx-frontend)描述了 C 和 C++ 前端使用的数据布局约定。

调试信息描述符是[专门的元数据节点](https://llvm.org/docs/LangRef.html" \l "specialized-metadata)，是Metadata 的子类.

### 调试器内部函数

LLVM 使用几个内部函数（名称以“ llvm.dbg”为前缀）通过优化和代码生成来跟踪源局部变量。

#### llvm.dbg.addr

|  |
| --- |
| void @llvm.dbg.addr(metadata, metadata, metadata) |

该内在函数提供有关局部元素（例如，变量）的信息。第一个参数是保存变量地址的元数据，通常是函数入口块中的静态分配。第二个参数是 包含变量描述的[局部变量](https://llvm.org/docs/LangRef.html" \l "dilocalvariable)。第三个参数是一个[复杂的表达式](https://llvm.org/docs/LangRef.html" \l "diexpression)。llvm.dbg.addr 内在描述 源变量的地址。

|  |
| --- |
| %i.addr = alloca i32, align 4  call void @llvm.dbg.addr(metadata i32\* %i.addr, metadata !1,  metadata !DIExpression()), !dbg !2  !1 = !DILocalVariable(name: "i", ...) ; int i  !2 = !DILocation(...)  ...  %buffer = alloca [256 x i8], align 8  ; The address of i is buffer+64.  call void @llvm.dbg.addr(metadata [256 x i8]\* %buffer, metadata !3,  metadata !DIExpression(DW\_OP\_plus, 64)), !dbg !4  !3 = !DILocalVariable(name: "i", ...) ; int i  !4 = !DILocation(...) |

前端应该在声明源变量时生成一次调用llvm.dbg.addr 。将变量从内存完全提升为 SSA 值的优化passes， 将可能用对llvm.dbg.value 的多次调用替换此调用。删除存储的passes 实际上是部分提升，它们将插入对llvm.dbg.value 和llvm.dbg.addr 的混合调用来跟踪源变量值（当源变量值可用时）。优化后，可能会多次调用来llvm.dbg.addr 来描述变量在内存中所在的程序点。对同一个具体源变量的所有调用必须就内存位置达成一致。

#### llvm.dbg.declare

|  |
| --- |
| void @llvm.dbg.declare(metadata, metadata, metadata) |

此内在函数与llvm.dbg.addr 相同，只是对于给定的具体局部变量只能调用一次 llvm.dbg.declare。它不依赖于控制，这意味着如果存在对llvm.dbg.declare的调用并且具有有效的位置参数，则该地址将被视为变量在其整个生命周期中的真正位置。这使得优化很难在 llvm.dbg.declare 存在的情况下保留准确的调试信息，因此我们正在远离它，我们计划在未来的 LLVM 版本中弃用它。

#### llvm.dbg.value

void @llvm.dbg.value(metadata, metadata, metadata)

当用户源变量设置为新值时，此内在函数提供信息。第一个参数是新值（包装为元数据）。第二个参数是包含变量描述的[局部变量](https://llvm.org/docs/LangRef.html" \l "dilocalvariable)。第三个参数是一个复杂的表达式。

llvm.dbg.value内在函数直接描述了源变量的值，而不是它的地址。请注意，此内部函数的值操作数可能是间接的（即，指向源变量的指针），前提是解释复杂表达式会导出直接值。

#### llvm.dbg.assign

|  |
| --- |
| void @llvm.dbg.assign(Value \*Value,  DIExpression \*ValueExpression,  DILocalVariable \*Variable,  DIAssignID \*ID,  Value \*Address,  DIExpression \*AddressExpression) |

此内在函数标记 IR 中发生源分配的位置。它对变量的值进行编码。它引用执行分配的存储（如果有）和目标地址。

前三个参数与llvm.dbg.value中的相同。 第四个参数DIAssignID用于引用一个存储。第五个是存储的目的地（包装为元数据），第六个是修改它的[复杂表达式](https://llvm.org/docs/LangRef.html" \l "diexpression)。

正式的 LLVM-IR 签名是：

|  |
| --- |
| void @llvm.dbg.assign(metadata, metadata, metadata, metadata, metadata, metadata) |

有关详细信息，请参阅[调试信息分配跟踪](https://llvm.org/docs/AssignmentTracking.html)。

## 对象生命周期和范围

在许多语言中，函数中的局部变量可以将它们的生命周期或作用域限制在函数的一个子集中。例如，在 C 语言家族中，变量仅在定义它们的源代码块中有效（可读和可写）。在函数式语言中，值只有在定义后才可读。尽管这是一个非常明显的概念，但在 LLVM 中进行建模并非易事，因为它没有这种意义上的作用域概念，并且不想受限于语言的作用域规则。

为了处理这个问题，LLVM 调试格式使用附加到 llvm 指令的元数据来编码行号和范围信息。例如，考虑以下 C 片段：

|  |
| --- |
| 1. void foo() {  2. int X = 21;  3. int Y = 22;  4. {  5. int Z = 23;  6. Z = X;  7. }  8. X = Y;  9. } |

编译为 LLVM，此函数将表示如下：

|  |
| --- |
| ; Function Attrs: nounwind ssp uwtable  define void @foo() #0 !dbg !4 {  entry:  %X = alloca i32, align 4  %Y = alloca i32, align 4  %Z = alloca i32, align 4  call void @llvm.dbg.declare(metadata i32\* %X, metadata !11, metadata !13), !dbg !14  store i32 21, i32\* %X, align 4, !dbg !14  call void @llvm.dbg.declare(metadata i32\* %Y, metadata !15, metadata !13), !dbg !16  store i32 22, i32\* %Y, align 4, !dbg !16  call void @llvm.dbg.declare(metadata i32\* %Z, metadata !17, metadata !13), !dbg !19  store i32 23, i32\* %Z, align 4, !dbg !19  %0 = load i32, i32\* %X, align 4, !dbg !20  store i32 %0, i32\* %Z, align 4, !dbg !21  %1 = load i32, i32\* %Y, align 4, !dbg !22  store i32 %1, i32\* %X, align 4, !dbg !23  ret void, !dbg !24  }  ; Function Attrs: nounwind readnone  declare void @llvm.dbg.declare(metadata, metadata, metadata) #1  attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "frame-pointer"="all" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  attributes #1 = { nounwind readnone }  !llvm.dbg.cu = !{!0}  !llvm.module.flags = !{!7, !8, !9}  !llvm.ident = !{!10}  !0 = !DICompileUnit(language: DW\_LANG\_C99, file: !1, producer: "clang version 3.7.0 (trunk 231150) (llvm/trunk 231154)", isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug, enums: !2, retainedTypes: !2, subprograms: !3, globals: !2, imports: !2)  !1 = !DIFile(filename: "/dev/stdin", directory: "/Users/dexonsmith/data/llvm/debug-info")  !2 = !{}  !3 = !{!4}  !4 = distinct !DISubprogram(name: "foo", scope: !1, file: !1, line: 1, type: !5, isLocal: false, isDefinition: true, scopeLine: 1, isOptimized: false, retainedNodes: !2)  !5 = !DISubroutineType(types: !6)  !6 = !{null}  !7 = !{i32 2, !"Dwarf Version", i32 2}  !8 = !{i32 2, !"Debug Info Version", i32 3}  !9 = !{i32 1, !"PIC Level", i32 2}  !10 = !{!"clang version 3.7.0 (trunk 231150) (llvm/trunk 231154)"}  !11 = !DILocalVariable(name: "X", scope: !4, file: !1, line: 2, type: !12)  !12 = !DIBasicType(name: "int", size: 32, align: 32, encoding: DW\_ATE\_signed)  !13 = !DIExpression()  !14 = !DILocation(line: 2, column: 9, scope: !4)  !15 = !DILocalVariable(name: "Y", scope: !4, file: !1, line: 3, type: !12)  !16 = !DILocation(line: 3, column: 9, scope: !4)  !17 = !DILocalVariable(name: "Z", scope: !18, file: !1, line: 5, type: !12)  !18 = distinct !DILexicalBlock(scope: !4, file: !1, line: 4, column: 5)  !19 = !DILocation(line: 5, column: 11, scope: !18)  !20 = !DILocation(line: 6, column: 11, scope: !18)  !21 = !DILocation(line: 6, column: 9, scope: !18)  !22 = !DILocation(line: 8, column: 9, scope: !4)  !23 = !DILocation(line: 8, column: 7, scope: !4)  !24 = !DILocation(line: 9, column: 3, scope: !4) |

这个例子说明了一些关于 LLVM 调试信息的重要细节。特别是，它展示了llvm.dbg.declare附加到指令的内在信息和位置信息如何一起应用，以允许调试器分析语句、变量定义和用于实现函数的代码之间的关系。

|  |
| --- |
| call void @llvm.dbg.declare(metadata i32\* %X, metadata !11, metadata !13), !dbg !14  ; [debug line = 2:7] [debug variable = X] |

第一个内部%llvm.dbg.declare编码变量X的调试信息。附加到内在函数的元数据!dbg !14提供变量的范围信息。

|  |
| --- |
| !14 = !DILocation(line: 2, column: 9, scope: !4)  !4 = distinct !DISubprogram(name: "foo", scope: !1, file: !1, line: 1, type: !5,  isLocal: false, isDefinition: true, scopeLine: 1,  isOptimized: false, retainedNodes: !2) |

这 !14 是提供[位置信息](https://llvm.org/docs/LangRef.html" \l "dilocation)的元数据。 在此示例中，作用域由 !4 (子程序描述符)编码。这样，附加到内在函数的位置信息表明,该变量X是在foo函数内函数级作用域的第 2 行声明的。

现在让我们再举一个例子。

|  |
| --- |
| call void @llvm.dbg.declare(metadata i32\* %Z, metadata !17, metadata !13), !dbg !19  ; [debug line = 5:9] [debug variable = Z] |

第三个内部 %llvm.dbg.declare 编码变量Z的调试信息。附加到内在函数的元数据!dbg !19 提供变量Z的范围信息。

|  |
| --- |
| !18 = distinct !DILexicalBlock(scope: !4, file: !1, line: 4, column: 5)  !19 = !DILocation(line: 5, column: 11, scope: !18) |

这里， !19表示Z在词法范围!18 内的第 5 行和第 11 列处声明。词法作用域本身位于上述子程序 !4 的内部。

每条指令附带的范围信息提供了一种简单的方法来查找范围涵盖的指令。

## 优化代码中的对象生命周期

在上面的例子中，每个变量赋值都唯一对应于一个内存存储到变量在堆栈上的位置。然而，在高度优化的代码中，LLVM 将大多数变量提升为 SSA 值，这些值最终可以放置在物理寄存器或内存位置。为了通过编译跟踪 SSA 值，当对象被提升为 SSA 值时， 为每个赋值创建一个llvm.dbg.value内在，记录变量的新位置。与llvm.dbg.declare内在相比：

* dbg.value 终止任何前面的 dbg.values 对指定变量（的任何重叠片段）的影响。
* dbg.value 在 IR 中的位置定义了变量值在指令流中更改的位置。
* 操作数可以是常量，表示变量被赋予一个常量值。

当优化 passes 更改或移动指令和块时，必须注意更新llvm.dbg.value内在——开发人员可以在调试程序时观察到反映在变量值中的此类变化。对于优化程序的任何执行，给定相同的输入，调试器呈现给开发人员的变量值集不应显示在未优化程序的执行中永远不会存在的状态。这样做可能会通过报告不存在的状态来误导开发人员，损害他们对优化程序的理解并破坏他们对调试器的信任。

有时完美保留变量位置是不可能的，通常是在优化冗余计算时。在这种情况下，应使用带有undef 操作数的 llvm.dbg.value 来终止较早的变量位置并让调试器呈现“optimized out”给开发人员。向开发人员隐瞒这些可能过时的变量值会减少可用调试信息的数量，但会增加剩余信息的可靠性。

为了说明一些潜在的问题，请考虑以下示例：

|  |
| --- |
| define i32 @foo(i32 %bar, i1 %cond) {  entry:  call @llvm.dbg.value(metadata i32 0, metadata !1, metadata !2)  br i1 %cond, label %truebr, label %falsebr  truebr:  %tval = add i32 %bar, 1  call @llvm.dbg.value(metadata i32 %tval, metadata !1, metadata !2)  %g1 = call i32 @gazonk()  br label %exit  falsebr:  %fval = add i32 %bar, 2  call @llvm.dbg.value(metadata i32 %fval, metadata !1, metadata !2)  %g2 = call i32 @gazonk()  br label %exit  exit:  %merge = phi [ %tval, %truebr ], [ %fval, %falsebr ]  %g = phi [ %g1, %truebr ], [ %g2, %falsebr ]  call @llvm.dbg.value(metadata i32 %merge, metadata !1, metadata !2)  call @llvm.dbg.value(metadata i32 %g, metadata !3, metadata !2)  %plusten = add i32 %merge, 10  %toret = add i32 %plusten, %g  call @llvm.dbg.value(metadata i32 %toret, metadata !1, metadata !2)  ret i32 %toret  } |

在!1 和!3 中包含两个源级变量。该函数也许可以优化为以下代码：

|  |
| --- |
| define i32 @foo(i32 %bar, i1 %cond) {  entry:  %g = call i32 @gazonk()  %addoper = select i1 %cond, i32 11, i32 12  %plusten = add i32 %bar, %addoper  %toret = add i32 %plusten, %g  ret i32 %toret  } |

应该放置哪些llvm.dbg.value内在来表示此代码中的原始变量位置？不幸的是，源函数中 !1 的第二个、第三个和第四个 dbg.values 已经优化了它们的操作数（%tval、%fval、%merge）。假设我们无法恢复它们，我们可能会考虑放置 dbg.values：

|  |
| --- |
| define i32 @foo(i32 %bar, i1 %cond) {  entry:  call @llvm.dbg.value(metadata i32 0, metadata !1, metadata !2)  %g = call i32 @gazonk()  call @llvm.dbg.value(metadata i32 %g, metadata !3, metadata !2)  %addoper = select i1 %cond, i32 11, i32 12  %plusten = add i32 %bar, %addoper  %toret = add i32 %plusten, %g  call @llvm.dbg.value(metadata i32 %toret, metadata !1, metadata !2)  ret i32 %toret  } |

然而，这将导致!3(存储的@gazonk()的返回值)与!1 存储的常量值零同时出现——一对未优化程序中从未发生过的赋值。为避免这种情况，我们必须通过为!3 在 dbg.value 之前插入一个 undef dbg.value来终止具有常量值分配的范围 !1：

|  |
| --- |
| define i32 @foo(i32 %bar, i1 %cond) {  entry:  call @llvm.dbg.value(metadata i32 0, metadata !1, metadata !2)  %g = call i32 @gazonk()  call @llvm.dbg.value(metadata i32 undef, metadata !1, metadata !2)  call @llvm.dbg.value(metadata i32 %g, metadata !3, metadata !2)  %addoper = select i1 %cond, i32 11, i32 12  %plusten = add i32 %bar, %addoper  %toret = add i32 %plusten, %g  call @llvm.dbg.value(metadata i32 %toret, metadata !1, metadata !2)  ret i32 %toret  } |

一般而言，如果任何 dbg.value 的操作数已优化且无法恢复，则需要使用 undef dbg.value 来终止较早的变量位置。当调试器可以观察到分配的重新排序时，可能需要额外的 undef dbg.values。

## 变量位置元数据在 CodeGen 期间如何转换

LLVM 在整个中级和后端 passes 中保留调试信息，最终生成源级信息和指令范围之间的映射。这对于行号信息来说相对简单，因为将指令映射到行号是一个简单的关联。然而，对于变量位置，情况要复杂得多。由于每个llvm.dbg.value内在代表一个源代码级别的值到源变量的赋值，变量位置内在有效地将一个小的命令式程序嵌入到 LLVM IR 中。到 CodeGen 结束时，这变成了从每个变量到它们在指令范围内的机器位置的映射。从 IR 到 “object emission”，影响变量位置保真度的主要转换是：

* 1.指令选择
* 2.寄存器分配
* 3.区块布局

每一个都在下面讨论。此外，指令调度可以显著改变程序的顺序，并出现在许多不同的passes中。

某些变量位置在 CodeGen 期间不会转换。由llvm.dbg.declare 指定的堆栈位置在函数的整个持续时间内有效且不变，并记录在一个简单的 MachineFunction 表中。函数序言和结尾中的位置更改也将被忽略：帧设置和销毁可能需要几条指令，需要输出二进制文件中不成比例的调试信息来描述，并且无论如何都应该由调试器跳过。

### 指令选择和 MIR 中的变量位置

指令选择从一个 IR 函数创建一个 MIR 函数，就像它把intermediate 指令转换成机器指令一样，所以必须 intermediate变量位置成为机器变量位置。在 IR 中，变量位置总是由一个值标识，但在 MIR 中可以有不同类型的变量位置。此外，某些 IR 位置变得不可用，例如，如果将多个 IR 指令的操作合并为一个机器指令（例如乘法和累加），则中间值将丢失。为了通过指令选择跟踪变量位置，首先将它们分为不依赖于代码生成的位置（常量、堆栈位置、分配的虚拟寄存器）和依赖于代码生成的位置。对于那些这样做的人，调试元数据附加到 SelectionDAG 中的 SDNodes。在发生指令选择并创建 MIR 函数后，如果与调试元数据关联的 SDNode 分配了一个虚拟寄存器，该虚拟寄存器用作变量位置。如果 SDNode 被折叠成机器指令或以其他方式转换为非寄存器，则变量位置变得不可用。

不可用的位置被视为已被优化掉：在 IR 中，该位置将由调试内在函数分配 undef，而在 MIR 中，使用等效位置。

在将 MIR 位置分配给每个变量后，对应于每个llvm.dbg.value和llvm.dbg.addr 内在 的机器伪指令被插入。这种类型的指令有两种形式。

* 第一种形式 ，DBG\_VALUE，如下：

|  |
| --- |
| DBG\_VALUE %1, $noreg, !123, !DIExpression() |

并具有以下操作数：

* 第一个操作数可以将变量位置记录为寄存器、帧索引、立即数或基地址寄存器（如果原始调试内在引用内存）。$noreg表示变量位置未定义，相当于一个undef dbg.value 操作数。
* 第二个操作数的类型表示变量位置是由DBG\_VALUE 直接引用的，还是间接引用的。$noreg 寄存器表示前者， 立即操作数 (0) 表示后者。
* 操作数 3 是原始调试内在变量的字段。
* 操作数 4 是原始调试内在函数的表达式字段。
* 第二种形式，DBG\_VALUE\_LIST，如下：

|  |
| --- |
| DBG\_VALUE\_LIST !123, !DIExpression(DW\_OP\_LLVM\_arg, 0, DW\_OP\_LLVM\_arg, 1, DW\_OP\_plus), %1, %2 |

并具有以下操作数：

* 第一个操作数是原始调试内部函数的变量字段。
* 第二个操作数是原始调试内在函数的表达式字段。
* 任意数量的操作数，从第 3 个开始，记录一系列变量位置操作数，这些操作数可以取与上述DBG\_VALUE指令的第一个操作数相同的任何值。这些变量位置操作数被插入到最终的 DWARF 表达式中，位置由DIExpression <LangRef.html#diexpression>中的 DW\_OP\_LLVM\_arg 运算符指示。

插入 DBG\_VALUE 的位置应与其匹配的llvm.dbg.value内在在 IR 块中的位置相对应。与优化一样，LLVM 旨在保留变量分配在源程序中发生的顺序。然而，SelectionDAG 执行一些指令调度，可以重新排序分配（下面讨论）。函数参数位置移动到函数的开头，以确保它们在函数入口时立即可用。

要在指令选择期间演示变量位置，请考虑以下示例：

|  |
| --- |
| define i32 @foo(i32\* %addr) {  entry:  call void @llvm.dbg.value(metadata i32 0, metadata !3, metadata !DIExpression()), !dbg !5  br label %bb1, !dbg !5  bb1: ; preds = %bb1, %entry  %bar.0 = phi i32 [ 0, %entry ], [ %add, %bb1 ]  call void @llvm.dbg.value(metadata i32 %bar.0, metadata !3, metadata !DIExpression()), !dbg !5  %addr1 = getelementptr i32, i32 \*%addr, i32 1, !dbg !5  call void @llvm.dbg.value(metadata i32 \*%addr1, metadata !3, metadata !DIExpression()), !dbg !5  %loaded1 = load i32, i32\* %addr1, !dbg !5  %addr2 = getelementptr i32, i32 \*%addr, i32 %bar.0, !dbg !5  call void @llvm.dbg.value(metadata i32 \*%addr2, metadata !3, metadata !DIExpression()), !dbg !5  %loaded2 = load i32, i32\* %addr2, !dbg !5  %add = add i32 %bar.0, 1, !dbg !5  call void @llvm.dbg.value(metadata i32 %add, metadata !3, metadata !DIExpression()), !dbg !5  %added = add i32 %loaded1, %loaded2  %cond = icmp ult i32 %added, %bar.0, !dbg !5  br i1 %cond, label %bb1, label %bb2, !dbg !5  bb2: ; preds = %bb1  ret i32 0, !dbg !5  } |

如果使用 llc -o - -start-after=codegen-prepare -stop-after=expand-isel-pseudos -mtriple=x86\_64--编译此 IR ，则会生成以下 MIR：

|  |
| --- |
| bb.0.entry:  successors: %bb.1(0x80000000)  liveins: $rdi  %2:gr64 = COPY $rdi  %3:gr32 = MOV32r0 implicit-def dead $eflags  DBG\_VALUE 0, $noreg, !3, !DIExpression(), debug-location !5  bb.1.bb1:  successors: %bb.1(0x7c000000), %bb.2(0x04000000)  %0:gr32 = PHI %3, %bb.0, %1, %bb.1  DBG\_VALUE %0, $noreg, !3, !DIExpression(), debug-location !5  DBG\_VALUE %2, $noreg, !3, !DIExpression(DW\_OP\_plus\_uconst, 4, DW\_OP\_stack\_value), debug-location !5  %4:gr32 = MOV32rm %2, 1, $noreg, 4, $noreg, debug-location !5 :: (load 4 from %ir.addr1)  %5:gr64\_nosp = MOVSX64rr32 %0, debug-location !5  DBG\_VALUE $noreg, $noreg, !3, !DIExpression(), debug-location !5  %1:gr32 = INC32r %0, implicit-def dead $eflags, debug-location !5  DBG\_VALUE %1, $noreg, !3, !DIExpression(), debug-location !5  %6:gr32 = ADD32rm %4, %2, 4, killed %5, 0, $noreg, implicit-def dead $eflags :: (load 4 from %ir.addr2)  %7:gr32 = SUB32rr %6, %0, implicit-def $eflags, debug-location !5  JB\_1 %bb.1, implicit $eflags, debug-location !5  JMP\_1 %bb.2, debug-location !5  bb.2.bb2:  %8:gr32 = MOV32r0 implicit-def dead $eflags  $eax = COPY %8, debug-location !5  RET 0, $eax, debug-location !5 |

首先观察llvm.dbg.value 源 IR 中的每个内在都有一个 DBG\_VALUE 指令，确保没有源代码级分配丢失。然后考虑记录变量位置的不同方式：

* 对于第一个 dbg.value，立即操作数用于记录零值。
* PHI 指令的 dbg.value 导致一个虚拟寄存器 %0的 DBG\_VALUE。
* 第一个 GEP 将其效果折叠到第一个加载指令中（作为 4 字节偏移量），但是通过将 GEP 效果折叠到 DIExpression 中来挽救变量位置。
* 第二个GEP也折叠到相应的负载中。但是，它不够简单，无法挽救，并作为$noreg DBG\_VALUE 发出，表明该变量位于未定义的位置。
* 最终的 dbg.value 将其 Value 放置在虚拟寄存器%1中。

### 指令调度

许多passes 可以重新安排指令，特别是指令选择和 pre-and-post RA 机器调度程序。指令调度可以显著改变程序的性质——在（极不可能）最坏的情况下，指令序列可能会完全颠倒。在这种情况下，LLVM 遵循适用于优化的原则，调试器最好不要显示任何状态，而不是误导性状态。因此，每当指令按执行顺序前进时，任何相应的 DBG\_VALUE 都将保留在其原始位置，如果指令被延迟，则在延迟期间为变量提供未定义的位置。为了说明这一点，请考虑这个伪 MIR：

|  |
| --- |
| %1:gr32 = MOV32rm %0, 1, $noreg, 4, $noreg, debug-location !5 :: (load 4 from %ir.addr1)  DBG\_VALUE %1, $noreg, !1, !2  %4:gr32 = ADD32rr %3, %2, implicit-def dead $eflags  DBG\_VALUE %4, $noreg, !3, !4  %7:gr32 = SUB32rr %6, %5, implicit-def dead $eflags  DBG\_VALUE %7, $noreg, !5, !6 |

想象一下，SUB32rr 向前移动，为我们提供了以下 MIR：

|  |
| --- |
| %7:gr32 = SUB32rr %6, %5, implicit-def dead $eflags  %1:gr32 = MOV32rm %0, 1, $noreg, 4, $noreg, debug-location !5 :: (load 4 from %ir.addr1)  DBG\_VALUE %1, $noreg, !1, !2  %4:gr32 = ADD32rr %3, %2, implicit-def dead $eflags  DBG\_VALUE %4, $noreg, !3, !4  DBG\_VALUE %7, $noreg, !5, !6 |

在这种情况下，LLVM 将离开 MIR，如上所示。如果我们使用 SUB32rr 向上移动虚拟寄存器 %7 的 DBG\_VALUE，我们将重新排序分配并引入程序的新状态。而使用上面的解决方案，调试器将看到更少的变量值组合，因为 !3和!5将同时更改值。这比歪曲原始程序更可取。

相比之下，如果将 MOV32rm 下沉，LLVM 将产生以下内容：

|  |
| --- |
| DBG\_VALUE $noreg, $noreg, !1, !2  %4:gr32 = ADD32rr %3, %2, implicit-def dead $eflags  DBG\_VALUE %4, $noreg, !3, !4  %7:gr32 = SUB32rr %6, %5, implicit-def dead $eflags  DBG\_VALUE %7, $noreg, !5, !6  %1:gr32 = MOV32rm %0, 1, $noreg, 4, $noreg, debug-location !5 :: (load 4 from %ir.addr1)  DBG\_VALUE %1, $noreg, !1, !2 |

在这里，为了避免出现第一个分配给!1 的状态消失，块顶部的 DBG\_VALUE 将变量分配给未定义的位置，直到它的值在块末尾可用，这里添加了额外 DBG\_VALUE 。Were any other DBG\_VALUE for !1 to occur in the instructions that the MOV32rm was sunk past, the DBG\_VALUE for %1 would be dropped and the debugger would never observe it in the variable.。这准确地反映了在原始程序的相应部分的值不可用。

### 寄存器分配期间的变量位置

为了避免调试指令干扰寄存器分配器， LiveDebugVariables pass 从 MIR 函数中提取变量位置并删除相应的 DBG\_VALUE 指令。一些本地化的复制传播是在块内执行的。寄存器分配后，VirtRegRewriter pass 将 DBG\_VALUE 指令重新插入到它们的原始位置，将虚拟寄存器引用转换为它们的物理机器位置。为避免对不正确的变量位置进行编码，在此过程中，任何不活动的虚拟寄存器的 DBG\_VALUE 都将替换为未定义的位置。由于虚拟寄存器重写，LiveDebugVariables 可能会插入冗余的 DBG\_VALUE。这些将随后被 RemoveRedundantDebugValues pass 删除。

### 变量位置的 LiveDebugValues 扩展

在所有优化运行之后和发射前不久，运行 LiveDebugValues pass 以实现两个目标：

* 通过复制和寄存器溢出来传播变量的位置，
* 对于每个块，记录该块中的每个有效变量位置。

在这一步之后，DBG\_VALUE 指令改变了含义：它不再对应于变量可能更改值的源代码级赋值，而是断言块中变量的位置，并在块外失去作用。通过副本和溢出传播变量位置非常简单：确定每个基本块中的变量位置需要考虑控制流。考虑以下 IR，它存在几个困难：

|  |
| --- |
| define dso\_local i32 @foo(i1 %cond, i32 %input) !dbg !12 {  entry:  br i1 %cond, label %truebr, label %falsebr  bb1:  %value = phi i32 [ %value1, %truebr ], [ %value2, %falsebr ]  br label %exit, !dbg !26  truebr:  call void @llvm.dbg.value(metadata i32 %input, metadata !30, metadata !DIExpression()), !dbg !24  call void @llvm.dbg.value(metadata i32 1, metadata !23, metadata !DIExpression()), !dbg !24  %value1 = add i32 %input, 1  br label %bb1  falsebr:  call void @llvm.dbg.value(metadata i32 %input, metadata !30, metadata !DIExpression()), !dbg !24  call void @llvm.dbg.value(metadata i32 2, metadata !23, metadata !DIExpression()), !dbg !24  %value = add i32 %input, 2  br label %bb1  exit:  ret i32 %value, !dbg !30  } |

这里的困难是：

* 控制流程与基本块顺序大致相反
* !23变量的值合并到%bb1，但是没有PHI节点

如上所述，llvm.dbg.value内在函数本质上形成了一个嵌入在 IR 中的命令式程序，每个内在函数定义了一个变量位置。这可以通过 mem2reg 转换为 SSA 形式，就像它使用 use-def 链来识别控制流合并并为 IR 值插入 phi 节点一样。然而，由于调试变量位置是为每条机器指令定义的，实际上每条 IR 指令都使用每个变量位置，这将导致生成大量调试内在函数。

检查上面的示例，通过函数在两个条件路径上分配变量!30 为%input，同时在任一路径上分配!23不同的常量值。在控制流合并的地方%bb1，我们希望!30保留其位置 (%input )，但!23变为未定义，因为我们无法在运行时确定在不插入 PHI 节点的情况下它在 %bb1 中应该具有什么值。mem2reg 不插入 PHI 节点以避免在启用调试时更改代码生成器，并且不插入其他 dbg.values 以避免添加大量内在函数。

相反，LiveDebugValues 在控制流合并时确定变量位置。数据流分析用于在块之间传播位置：当控制流合并时，如果一个变量在所有前导中具有相同的位置，那么该位置将传播到后继。如果前任位置不一致，则该位置将变为未定义。

LiveDebugValues 运行后，每个块都应具有块内 DBG\_VALUE 指令描述的所有有效变量位置。支持类（例如 DbgEntityHistoryCalculator）只需要很少的努力就可以构建每条指令到每个有效变量位置的映射，而无需考虑控制流。从上面的示例中，很难确定变量!30的位置应该“向上”流入块%bb1，但变量!23的位置不应该“向下”流入%exit块。

## C/C++ 前端特定调试信息

C 和 C++ 前端以一种在信息内容方面与DWARF实际上相同的格式表示有关程序 的信息。这允许代码生成器通过生成标准的 dwarf 信息来简单地支持本机调试器，并包含足够的信息供非 dwarf 目标根据需要进行翻译。

本节描述用于表示 C 和 C++ 程序的形式。其他语言可以在这之后自我模式化（它本身被调整为以与 DWARF 相同的方式表示程序），或者如果它们不适合 DWARF 模型，它们可以选择提供完全不同的形式。由于对调试信息的支持被添加到各种 LLVM 源语言前端，因此应在此处记录所使用的信息。

以下部分提供了一些 C/C++ 构造的示例以及最能描述这些构造的调试信息。规范引用是在  include/llvm/IR/DebugInfoMetadata.h 中定义的DINode类和辅助函数的实现lib/IR/DIBuilder.cpp。

### C/C++ 源文件信息

llvm::Instruction提供对指令附带的元数据的轻松访问。可以使用Instruction::getDebugLoc()和 DILocation::getLine() 提取在 LLVM IR 中编码的行号信息。

|  |
| --- |
| if (DILocation \*Loc = I->getDebugLoc()) { // Here I is an LLVM instruction  unsigned Line = Loc->getLine();  StringRef File = Loc->getFilename();  StringRef Dir = Loc->getDirectory();  bool ImplicitCode = Loc->isImplicitCode();  } |

当标志 ImplicitCode 为 true 时，表示 Instruction 已由前端添加，但与用户编写的源代码不对应。例如

|  |
| --- |
| if (MyBoolean) {  MyObject MO;  ...  } |

在作用域的末尾，调用了 MyObject 的析构函数，但并未明确编写。此信息有助于避免在进行代码覆盖时在括号中使用计数器。

### C/C++ 全局变量信息

给定一个整型全局变量声明如下：

|  |
| --- |
| \_Alignas(8) int MyGlobal = 100; |

C/C++ 前端会生成以下描述符：

|  |
| --- |
| ;;  ;; Define the global itself.  ;;  @MyGlobal = global i32 100, align 8, !dbg !0  ;;  ;; List of debug info of globals  ;;  !llvm.dbg.cu = !{!1}  ;; Some unrelated metadata.  !llvm.module.flags = !{!6, !7}  !llvm.ident = !{!8}  ;; Define the global variable itself  !0 = distinct !DIGlobalVariable(name: "MyGlobal", scope: !1, file: !2, line: 1, type: !5, isLocal: false, isDefinition: true, align: 64)  ;; Define the compile unit.  !1 = distinct !DICompileUnit(language: DW\_LANG\_C99, file: !2,  producer: "clang version 4.0.0",  isOptimized: false, runtimeVersion: 0, emissionKind: FullDebug,  enums: !3, globals: !4)  ;;  ;; Define the file  ;;  !2 = !DIFile(filename: "/dev/stdin",  directory: "/Users/dexonsmith/data/llvm/debug-info")  ;; An empty array.  !3 = !{}  ;; The Array of Global Variables  !4 = !{!0}  ;;  ;; Define the type  ;;  !5 = !DIBasicType(name: "int", size: 32, encoding: DW\_ATE\_signed)  ;; Dwarf version to output.  !6 = !{i32 2, !"Dwarf Version", i32 4}  ;; Debug info schema version.  !7 = !{i32 2, !"Debug Info Version", i32 3}  ;; Compiler identification  !8 = !{!"clang version 4.0.0"} |

DIGlobalVariable 描述中的对齐值指定变量对齐，它被 C11 \_Alignas()、C++11 alignas() 关键字或编译器属性 \_\_attribute\_\_((aligned ())) 强制。在其他情况下（当此字段缺失时）对齐被认为是默认的。这在为 DW\_AT\_alignment 值生成 DWARF 输出时使用。

### C/C++ 函数信息

给定一个声明如下的函数：

|  |
| --- |
| int main(int argc, char \*argv[]) {  return 0;} |

C/C++ 前端会生成以下描述符：

|  |
| --- |
| ;;  ;; Define the anchor for subprograms.  ;;  !4 = !DISubprogram(name: "main", scope: !1, file: !1, line: 1, type: !5,  isLocal: false, isDefinition: true, scopeLine: 1,  flags: DIFlagPrototyped, isOptimized: false,  retainedNodes: !2)  ;;  ;; Define the subprogram itself.  ;;  define i32 @main(i32 %argc, i8\*\* %argv) !dbg !4 {  ...  } |

## C++ 特定的调试信息

### C++ 特殊成员函数信息

DWARF v5 引入了定义的属性以增强 C++ 程序的调试信息。LLVM 可以生成（或省略）这些适当的 DWARF 属性。在 C++ 中，一个特殊的成员函数 Ctors、Dtors、Copy/Move Ctors、assignment operators 可以用 C++11 关键字声明删除。这在 LLVM 中使用 spFlags 值 DISPFlagDeleted 表示。

给定一个声明为已删除的复制构造函数的类声明：

|  |
| --- |
| class foo {  public:  foo(const foo&) = deleted;  }; |

C++ 前端将生成以下内容：

|  |
| --- |
| !17 = !DISubprogram(name: "foo", scope: !11, file: !1, line: 5, type: !18, scopeLine: 5, flags: DIFlagPublic | DIFlagPrototyped, spFlags: DISPFlagDeleted) |

这将产生一个额外的 DWARF 属性：

|  |
| --- |
| DW\_TAG\_subprogram [7] \*  DW\_AT\_name [DW\_FORM\_strx1] (indexed (00000006) string = "foo")  DW\_AT\_decl\_line [DW\_FORM\_data1] (5)  ...  DW\_AT\_deleted [DW\_FORM\_flag\_present] (true) |

# 调试信息的指令参考

本文档解释了 LLVM 如何使用值跟踪或指令引用来确定编译代码生成阶段调试信息的变量位置。此内容针对那些从事代码生成目标和优化过程的人员。任何对低级调试信息处理感兴趣的人也可能会对它感兴趣。

## 问题陈述

在编译结束时，LLVM 必须为该变量的词法范围内的每条指令生成一个 DWARF 位置列表（或类似列表），描述该变量可以在哪个寄存器或堆栈位置找到。我们可以通过编译跟踪变量所在的虚拟寄存器，但是这很容易受到 regalloc 和指令移动期间的寄存器优化的影响。

## 解决方案：指令引用

LLVM 不是识别变量值所在的虚拟寄存器，而是在指令引用模式中，而是引用定义该值的机器指令和操作数位置。考虑引用指令值的 LLVM IR 方式：

%2 = add i32 %0, %1call void @llvm.dbg.value(metadata i32 %2,

在 LLVM IR 中，IR Value 与计算值的指令同义，在某种程度上，在内存中，Value 是指向计算指令的指针。在指令选择之后，指令引用在 LLVM 的代码生成后端中实现了这种关系。考虑下面的 X86 程序集和引用调试信息的指令，对应于早期的 LLVM IR：

%2:gr32 = ADD32rr %0, %1, implicit-def $eflags, debug-instr-number 1

DBG\_INSTR\_REF 1, 0, !123, !456, debug-location !789

当函数保持 SSA 形式时，虚拟寄存器%2足以识别指令计算的值——但是函数最终离开 SSA 形式，寄存器优化将掩盖所需值所在的寄存器。取而代之的是一种更一致的识别方式指令的值是指MachineOperand定义值的位置：独立于那个定义的寄存器MachineOperand。在上面的代码中，DBG\_INSTR\_REF指令引用一号指令，操作数零，而ADD32rr有一个debug-instr-number属性表明它是一号指令。

将变量位置与寄存器解耦避免了涉及寄存器分配和优化的困难，但在优化指令时需要额外的检测。用计算相同值的优化版本替换指令的优化必须保留指令编号，或者记录从旧指令/操作数编号对到新指令/操作数对的替换 - 请参阅 MachineFunction::substituteDebugValuesForInst。如果不执行调试信息维护，或者一条指令作为死代码被删除，变量位置将被安全地删除并标记为“优化掉”。异常是变异而不是替换的指令，它们总是需要调试信息维护。

## 寄存器分配器注意事项

当寄存器分配器运行时，调试指令不直接引用任何虚拟寄存器，因此在 regalloc 期间不需要昂贵的位置维护（即LiveDebugVariables）。调试指令与函数断开链接，然后在寄存器分配完成后链接回来。

指令是个例外PHI：一旦 regalloc 完成，它们就成为控制流合并时的隐式定义，并且附加到 PHI指令的任何调试号都将丢失。PHI为了避免这种情况，在寄存器分配 ( ) 开始时记录 s 的调试号phi-node-elimination，然后 DBG\_PHI在 regalloc 完成后插入指令。这需要在 regalloc 期间对变量位于哪个寄存器进行一些维护，但在单个位置（块入口点）而不是指令范围。

一个例子，在 regalloc 之前：

bb.2:

%2 = PHI %1, %bb.0, %2, %bb.1, debug-instr-number 1

后：

bb.2:

DBG\_PHI $rax, 1

## LiveDebugValues

优化和代码布局完成后，有关变量值的信息必须转换为变量位置，即寄存器和堆栈槽。这是在 [LiveDebugValues pass][ LiveDebugValues] 中执行的，其中调试指令和机器代码被分成两个独立的函数：

一种为变量名赋值，

一种将值分配给机器寄存器和堆栈槽的方法。

LLVM 现有的 SSA 工具用于PHI为每个函数放置 s，在变量值和机器位置中包含的值之间，值传播消除任何不必要PHI的 s。然后可以将两者结合起来，为函数中的每条指令将变量映射到值，然后将值映射到位置。

此过程的关键是能够识别值在寄存器和堆栈位置之间的移动，以便值的位置可以在它们驻留在机器中的整个时间内得到保留。

## 所需的目标支持和转换指南

指令引用适用于任何目标，但可能覆盖率很低。支持指令引用井需要：

要实现的目标挂钩允许LiveDebugValues通过机器跟踪值，

要检测的特定于目标的优化，以保留指令编号。

### 目标钩子

TargetInstrInfo::isCopyInstrImpl必须实现以识别任何类似复制的指令——LiveDebugValues使用它来识别值何时在寄存器之间移动。

TargetInstrInfo::isLoadFromStackSlotPostFE并且 TargetInstrInfo::isStoreToStackSlotPostFE需要识别溢出和恢复指令。每个应该分别返回目标或源寄存器。LiveDebugValues将跟踪值从/到堆栈槽的移动。此外，任何写入堆栈溢出的指令都应该有一个MachineMemoryOperand附加的，这样LiveDebugValues才能识别出一个槽已经被破坏了。

### 针对特定目标的优化工具

优化有两种形式：一种是改变 aMachineInstr使其做一些不同的事情，另一种是创建新指令来替换旧指令的操作。

前者必须被检测——相关的问题是任何操作数中的任何寄存器定义是否会由于突变而产生不同的值。如果答案是肯定的，那么DBG\_INSTR\_REF 引用该操作数的指令可能会最终将不同的值分配给变量，从而为调试开发人员提供意外的变量值。在这种情况下，调用MachineInstr::dropDebugNumber()变异指令以擦除其指令编号。任何DBG\_INSTR\_REF 引用它都会产生一个空的变量位置，在调试器中显示为“已优化”。

对于后一种优化方式，为了增加覆盖率，您应该记录指令号替换：从旧指令号/操作数对到新指令号/操作数对的映射。考虑一下我们是否用双地址加法替换三地址加法指令：

%2:gr32 = ADD32rr %0, %1, debug-instr-number 1

成为

%2:gr32 = ADD32rr %0(tied-def 0), %1, debug-instr-number 2

将“指令编号 1 操作数 0”替换为“指令编号 2 操作数 0”，记录在MachineFunction. 在LiveDebugValues中， DBG\_INSTR\_REFs 将通过替换表映射，找到它所引用的值的最近的指令号/操作数号。

用于MachineFunction::substituteDebugValuesForInst自动生成旧指令和新指令之间的替换。它假定任何在旧指令中为 def 的操作数在新指令中的相同操作数位置都是 def。这在大多数情况下都有效，例如在上面的示例中。

如果新旧指令的操作数编号不一致，则使用 MachineInstr::getDebugInstrNum获取新指令的指令编号，MachineFunction::makeDebugValueSubstitution记录新旧指令中寄存器定义的映射关系。如果旧指令计算的某些值不再由新指令计算，则记录无替换——LiveDebugValues将安全地删除现在不可用的变量值。

如果您的目标克隆指令与TailDuplicator 优化过程非常相似，请不要尝试保留指令编号或记录任何替换。MachineFunction::CloneMachineInstr应该删除任何克隆指令的指令编号，以避免出现重复编号LiveDebugValues。处理重复指令是对当前未实现的指令引用的自然扩展。

# 编写 LLVM 后端---待学习

## 简介

本文档描述了编写编译器后端的技术，这些后端将 LLVM 中间表示 (IR) 转换为指定机器或其他语言的代码。用于特定机器的代码可以采用汇编代码或二进制代码的形式（可用于 JIT 编译器）。

LLVM 的后端具有一个独立于目标的代码生成器，它可以为多种类型的目标 CPU 创建输出——包括 X86、PowerPC、ARM 和 SPARC。后端还可用于生成针对 Cell 处理器或 GPU 的 SPU 的代码，以支持计算内核的执行。

该文档着重于 llvm/lib/Target在下载的 LLVM 版本的子目录中找到的现有示例。特别是，本文档着重于为 SPARC 目标创建静态编译器（生成文本程序集）的示例，因为 SPARC 具有相当标准的特征，例如 RISC 指令集和直接的调用约定。

本文档的读者是需要编写 LLVM 后端来为特定硬件或软件目标生成代码的任何人。

### 先决条件阅读

在阅读本文档之前必须先阅读这些基本文档：

[LLVM 语言参考手册](https://llvm.org/docs/LangRef.html)— LLVM 汇编语言的参考手册。

[The LLVM Target-Independent Code Generator](https://llvm.org/docs/CodeGenerator.html)——组件（类和代码生成算法）的指南，用于将 LLVM 内部表示转换为指定目标的机器代码。特别注意代码生成阶段的描述：指令选择、调度和形成、基于 SSA 的优化、寄存器分配、Prolog/Epilog 代码插入、后期机器代码优化和代码发射。

[TableGen 概述](https://llvm.org/docs/TableGen/index.html)— 描述 TableGen (tblgen) 应用程序的文档，该应用程序管理特定于域的信息以支持 LLVM 代码生成。TableGen 处理来自目标描述文件（.td后缀）的输入并生成可用于代码生成的 C++ 代码。

[Writing an LLVM Pass](https://llvm.org/docs/WritingAnLLVMPass.html)— 汇编printer是一个FunctionPass，几个SelectionDAG处理步骤也是如此。

要遵循本文档中的 SPARC 示例，请准备一份[The SPARC Architecture Manual, Version 8](http://www.sparc.org/standards/V8.pdf)以供参考。有关 ARM 指令集的详细信息，请参阅[ARM Architecture Reference Manual](http://infocenter.arm.com/)。有关 GNU 汇编程序格式 ( GAS) 的更多信息，请参阅[using As](http://sourceware.org/binutils/docs/as/index.html)，尤其是汇编printer。“Using As”包含目标机器相关功能的列表。

### 基本步骤

要为 LLVM 编写编译器后端，将 LLVM IR 转换为指定目标（机器或其他语言）的代码，请执行以下步骤：

* 创建描述目标机器特性的TargetMachine类的子类。复制特定 TargetMachine类和头文件的现有示例；例如，以 SparcTargetMachine.cpp和SparcTargetMachine.h 开始，但为你的目标更改文件名。同样，更改引用“ Sparc”的代码以引用您的目标。
* 描述目标的寄存器集。使用 TableGen 从特定于目标的RegisterInfo.td输入文件生成代码，该代码用于寄存器定义、寄存器别名和寄存器类。您还应该为该TargetRegisterInfo 类的子类编写额外的代码，该子类表示用于寄存器分配的类寄存器文件数据，并且还描述了寄存器之间的交互。
* 描述目标的指令集。使用 TableGen 从 TargetInstrFormats.td和TargetInstrInfo.td的目标特定版本的生成目标特定指令的代码。您应该为TargetInstrInfo 类的子类编写额外的代码来表示目标机器支持的机器指令。
* 描述 LLVM IR 从指令的有向无环图 (DAG) 表示到本地目标特定指令的选择和转换。根据TargetInstrInfo.td 的目标特定版本，使用 TableGen 生成匹配模式的代码。为XXXISelDAGToDAG.cpp 编写代码，其中XXX标识具体目标，执行模式匹配和 DAG-to-DAG 指令选择。还要在XXXISelLowering.cpp 中写入代码以替换或删除 SelectionDAG 中本机不支持的操作和数据类型。
* 为 assembly printer编写代码， assembly printer 将 LLVM IR 转换为目标机器的 GAS 格式。您应该将汇编字符串添加到TargetInstrInfo.td 的目标特定版本。您还应该为AsmPrinter 的子类编写代码，该子类执行 LLVM-to-assembly 转换和TargetAsmInfo的普通子类。
* 可选的，添加对子目标的支持（即具有不同功能的变体）。您还应该为该TargetSubtarget 类的子类编写代码 ，这样您就可以使用-mcpu=和 -mattr= command-line 选项。
* 可选地，添加 JIT 支持并创建一个机器代码发射器（TargetJITInfo的子类 ），用于将二进制代码直接发射到内存中。

在.cpp和.h 文件中，最初存根这些方法，然后再实现它们。最初，您可能不知道该类需要哪些私有成员以及哪些组件需要子类化。

### 预备

要实际创建编译器后端，您需要创建和修改一些文件。这里讨论绝对最小值。但是要实际使用 LLVM 目标无关代码生成器，您必须执行 [LLVM 目标无关代码生成器](https://llvm.org/docs/CodeGenerator.html)文档中描述的 步骤。

* 首先，你应该在下面创建一个子目录lib/Target来保存与你的目标相关的所有文件。如果您的目标名为“Dummy”，请创建目录lib/Target/Dummy.
* 在这个新目录中，创建一个CMakeLists.txt，复制另一个目标的 CMakeLists.txt 并修改它是最简单的。它至少应该包含LLVM\_TARGET\_DEFINITIONS变量。该库可以命名LLVMDummy （例如，参见 MIPS 目标）。或者，您可以将库拆分为LLVMDummyCodeGen和LLVMDummyAsmPrinter，后者应在lib/Target/Dummy下面的子目录中实现（例如，请参阅 PowerPC 目标）。

请注意，这两个命名方案被硬编码到llvm-config。使用任何其他命名方案会混淆llvm-config，并在链接llc时产生大量的链接错误。

* 为了让你的目标真正做一些事情，你需要实现一个TargetMachine的子类 。这个实现通常应该在文件 lib/Target/DummyTargetMachine.cpp中，但是lib/Target 目录中的任何文件都将被构建并且应该可以工作。要使用 LLVM 的目标独立代码生成器，您应该做所有当前机器后端所做的事情：创建LLVMTargetMachine的子类（要从头开始创建目标，请创建TargetMachine 的子类。）
* 要让 LLVM 实际构建和链接您的目标，您需要cmake 使用-DLLVM\_EXPERIMENTAL\_TARGETS\_TO\_BUILD=Dummy. 这将构建您的目标，而无需将其添加到所有目标的列表中。
* 目标稳定后，您可以将其添加到LLVM\_ALL\_TARGETS变量，该变量位于 main CMakeLists.txt。

## 目标机器

LLVMTargetMachine被设计为使用 LLVM 目标独立代码生成器实现的目标的基类。该类LLVMTargetMachine应该由实现各种虚拟方法的具体目标类特化。 LLVMTargetMachine被定义为 TargetMachine的子类，在include/llvm/Target/TargetMachine.h中。类 TargetMachine实现 ( TargetMachine.cpp) 还处理许多命令行选项。

要创建 LLVMTargetMachine 的具体目标特定子类，请先复制现有TargetMachine类和header。您应该命名您创建的文件以反映您的特定目标。例如，对于 SPARC 目标，将文件命名SparcTargetMachine.h为 SparcTargetMachine.cpp.

对于目标机器XXX， XXXTargetMachine 的实现必须有访问方法来获取表示目标组件的对象。这些方法被命名为get\*Info，旨在获取指令集 ( getInstrInfo) 、寄存器集 ( getRegisterInfo) 、栈帧布局 ( getFrameInfo) 和类似信息。 XXXTargetMachine还必须实现getDataLayout方法，该方法访问具有目标特定数据特征（例如数据类型大小和对齐要求）的对象。

例如，对于 SPARC 目标，头文件SparcTargetMachine.h 声明了几个简单的返回类成员的get\*Info和getDataLayout方法的原型。

|  |
| --- |
| namespace llvm {  class Module;  class SparcTargetMachine : public LLVMTargetMachine {  const DataLayout DataLayout; // Calculates type size & alignment  SparcSubtarget Subtarget;  SparcInstrInfo InstrInfo;  TargetFrameInfo FrameInfo;  protected:  virtual const TargetAsmInfo \*createTargetAsmInfo() const;  public:  SparcTargetMachine(const Module &M, const std::string &FS);  virtual const SparcInstrInfo \*getInstrInfo() const {return &InstrInfo; }  virtual const TargetFrameInfo \*getFrameInfo() const {return &FrameInfo; }  virtual const TargetSubtarget \*getSubtargetImpl() const{return &Subtarget; }  virtual const TargetRegisterInfo \*getRegisterInfo() const {  return &InstrInfo.getRegisterInfo();  }  virtual const DataLayout \*getDataLayout() const { return &DataLayout; }  static unsigned getModuleMatchQuality(const Module &M);  // Pass Pipeline Configuration  virtual bool addInstSelector(PassManagerBase &PM, bool Fast);  virtual bool addPreEmitPass(PassManagerBase &PM, bool Fast);  };  } // end namespace llvm |

* getInstrInfo()
* getRegisterInfo()
* getFrameInfo()
* getDataLayout()
* getSubtargetImpl()

对于某些目标，您还需要支持以下方法：

* getTargetLowering()
* getJITInfo()

某些架构（例如 GPU）不支持跳转到任意程序位置，不支持使用掩码执行实现分支，不支持使用围绕循环体的特殊指令进行循环。为了避免 CFG 修改引入此类硬件无法处理的不可简化的控制流，目标必须在初始化时调用setRequiresStructuredCFG(true) 。

此外，XXXTargetMachine构造函数应指定一个 TargetDescription字符串，用于确定目标机器的数据布局，包括指针大小、对齐方式和字节顺序等特征。例如，构造函数SparcTargetMachine包含以下内容：

|  |
| --- |
| SparcTargetMachine::SparcTargetMachine(const Module &M, const std::string &FS)  : DataLayout("E-p:32:32-f128:128:128"),  Subtarget(M, FS), InstrInfo(Subtarget),  FrameInfo(TargetFrameInfo::StackGrowsDown, 8, 0) {  } |

连字符分隔TargetDescription字符串的各个部分。

* 字符串中的大写“ E”表示大端目标数据模型。小写的“ e”表示小端。
* “ p:”后面是指针信息：大小、ABI 对齐方式和首选对齐方式。如果“p: ”后面只有两个数字，那么第一个值是指针大小，第二个值既是ABI又是首选对齐。
* 然后是数字类型对齐的字母：“ i”、“ f”、“ v”或“ a”（对应整数、浮点数、向量或聚合）。“ i”、“ v”或“ a”后接 ABI 对齐和首选对齐。“ f”后面跟着三个值：第一个表示long double的大小，然后是ABI对齐方式，然后是ABI首选对齐方式。

## 目标注册

您还必须向 TargetRegistry 注册您的目标，这是其他 LLVM 工具用来在运行时查找和使用您的目标的工具。可以直接使用 TargetRegistry，但对于大多数目标，都有帮助程序模板可以为您处理工作。

所有目标都应声明一个全局Target对象，用于在注册期间表示目标。然后，在目标的TargetInfo 库中，目标应该定义该对象并使用RegisterTarget 模板来注册目标。例如，Sparc 注册码如下所示：

|  |
| --- |
| Target llvm::getTheSparcTarget();  extern "C" void LLVMInitializeSparcTargetInfo() {  RegisterTarget<Triple::sparc, /\*HasJIT=\*/false>  X(getTheSparcTarget(), "sparc", "Sparc");  } |

这允许TargetRegistry按名称或目标三元组查找目标。此外，大多数目标还将注册其他功能，这些功能可在单独的库中使用。这些注册步骤是分开的，因为一些客户端可能希望只链接目标的某些部分——例如，JIT 代码生成器不需要使用汇编printer。下面是一个注册 Sparc 汇编printer的例子：

|  |
| --- |
| extern "C" void LLVMInitializeSparcAsmPrinter() {  RegisterAsmPrinter<SparcAsmPrinter> X(getTheSparcTarget());  } |

有关详细信息，请参阅“ [llvm/Target/TargetRegistry.h](https://llvm.org/doxygen/TargetRegistry_8h-source.html) ”。

## 寄存器集和寄存器类

您应该描述一个具体的特定于目标的类，它表示目标机器的寄存器文件。此类称之为XXXRegisterInfo （其中XXX标识目标）并表示用于寄存器分配的类寄存器文件数据。它还描述了寄存器之间的交互。

您还需要定义寄存器类以对相关寄存器进行分类。应该为某些指令以相同方式处理的寄存器组添加一个寄存器类。典型的例子是整数、浮点数或矢量寄存器的寄存器类。寄存器分配器允许指令使用指定寄存器类中的任何寄存器以类似方式执行指令。寄存器类将虚拟寄存器分配给来自这些集合的指令，并且寄存器类让独立于目标的寄存器分配器自动选择实际寄存器。

寄存器的大部分代码，包括寄存器定义、寄存器别名和寄存器类，都是由 TableGen 从输入XXXRegisterInfo.td文件生成， 并放置在XXXGenRegisterInfo.h.inc和XXXGenRegisterInfo.inc输出文件中。XXXRegisterInfo实现中的一些代码需要手工编码。

### 定义一个寄存器

该XXXRegisterInfo.td文件通常以目标机器的寄存器定义开头。Register类（在 Target.td中指定）用于为每个寄存器定义一个对象。指定的字符串n成为寄存器的名称。基本Register对象没有任何子寄存器，也没有指定任何别名。

|  |
| --- |
| class Register<string n> {  string Namespace = "";  string AsmName = n;  string Name = n;  int SpillSize = 0;  int SpillAlignment = 0;  list<Register> Aliases = [];  list<Register> SubRegs = [];  list<int> DwarfNumbers = [];  } |

例如，在X86RegisterInfo.td文件中，有利用该Register类的寄存器定义，如：

|  |
| --- |
| def AL : Register<"AL">, DwarfRegNum<[0, 0, 0]>; |

这定义了寄存器AL并为其分配值（使用DwarfRegNum），这些值由gcc、gdb或调试信息编写器使用，来标识寄存器。对于 register AL，DwarfRegNum采用代表 3 种不同模式的 3 个值的数组：第一个元素用于 X86-64，第二个用于 X86-32 上的异常处理 (EH)，第三个是通用的。-1 是一个特殊的 Dwarf 编号，表示 gcc 编号未定义，-2 表示寄存器编号对此模式无效。

根据X86RegisterInfo.td文件中先前描述的行，TableGen 在文件中生成以下代码

|  |
| --- |
| static const unsigned GR8[] = { X86::AL, ... };  const unsigned AL\_AliasSet[] = { X86::AX, X86::EAX, X86::RAX, 0 };  const TargetRegisterDesc RegisterDescriptors[] = {  ...  { "AL", "AL", AL\_AliasSet, Empty\_SubRegsSet, Empty\_SubRegsSet, AL\_SuperRegsSet }, ... |

从寄存器信息文件中，TableGenTargetRegisterDesc为每个寄存器生成一个对象。 TargetRegisterDesc定义在 include/llvm/Target/TargetRegisterInfo.h，有以下字段中：

|  |
| --- |
| struct TargetRegisterDesc {  const char \*AsmName; // Assembly language name for the register  const char \*Name; // Printable name for the reg (for debugging)  const unsigned \*AliasSet; // Register Alias Set  const unsigned \*SubRegs; // Sub-register set  const unsigned \*ImmSubRegs; // Immediate sub-register set  const unsigned \*SuperRegs; // Super-register set  }; |

TableGen 使用整个目标描述文件 ( .td) 来确定寄存器的文本名称（在TargetRegisterDesc 的AsmName和Name字段中 ）以及其他寄存器与定义寄存器的关系（在其他TargetRegisterDesc字段中）。在此示例中，其他定义将寄存器“AX”、“ EAX”和“ RAX”建立为彼此的别名，因此， TableGen为该寄存器别名集生成一个以 null 结尾的数组 ( AL\_AliasSet)。

该Register类通常用作更复杂类的基类。在Target.td中，该Register类是RegisterWithSubRegs类的基类，该RegisterWithSubRegs类用于定义需要在SubRegs列表中指定子寄存器的寄存器，如下所示：

|  |
| --- |
| class RegisterWithSubRegs<string n, list<Register> subregs> : Register<n> {  let SubRegs = subregs;  } |

在SparcRegisterInfo.td中，为 SPARC 定义了额外的寄存器类：一个Register子类，SparcReg， 和更多子类：Ri、Rf和Rd。SPARC 寄存器由 5-bit ID 号标识，这是这些子类的共同特征。请注意使用“ let”表达式来覆盖最初在超类中定义的值（例如，Rd类中SubRegs 的字段）。

|  |
| --- |
| class SparcReg<string n> : Register<n> {  field bits<5> Num;  let Namespace = "SP";  }  // Ri - 32-bit integer registers  class Ri<bits<5> num, string n> :  SparcReg<n> {  let Num = num;  }  // Rf - 32-bit floating-point registers  class Rf<bits<5> num, string n> :  SparcReg<n> {  let Num = num;  }  // Rd - Slots in the FP register file for 64-bit floating-point values.  class Rd<bits<5> num, string n, list<Register> subregs> : SparcReg<n> {  let Num = num;  let SubRegs = subregs;  } |

在该SparcRegisterInfo.td文件中，有使用 Register类的这些子类的寄存器定义，例如：

|  |
| --- |
| def G0 : Ri< 0, "G0">, DwarfRegNum<[0]>;  def G1 : Ri< 1, "G1">, DwarfRegNum<[1]>;  ...  def F0 : Rf< 0, "F0">, DwarfRegNum<[32]>;  def F1 : Rf< 1, "F1">, DwarfRegNum<[33]>;  ...  def D0 : Rd< 0, "F0", [F0, F1]>, DwarfRegNum<[32]>;  def D1 : Rd< 2, "F2", [F2, F3]>, DwarfRegNum<[34]>; |

上面显示的最后两个寄存器 (D0和D1) 是双精度浮点寄存器，它们是成对的单精度浮点子寄存器的别名。除了别名之外，定义的寄存器的子寄存器和超级寄存器关系在寄存器的TargetRegisterDesc的属性中。

### 定义寄存器类

RegisterClass类（在Target.td 中指定）用于定义表示一组相关寄存器的对象，还定义了寄存器的默认分配顺序。目标描述文件 XXXRegisterInfo.td使用Target.td，可以使用以下类构造RegisterClass：

|  |
| --- |
| class RegisterClass<string namespace,  list<ValueType> regTypes, int alignment, dag regList> {  string Namespace = namespace;  list<ValueType> RegTypes = regTypes;  int Size = 0; // spill size, in bits; zero lets tblgen pick the size  int Alignment = alignment;  // CopyCost is the cost of copying a value between two registers  // default value 1 means a single instruction  // A negative value means copying is extremely expensive or impossible  int CopyCost = 1;  dag MemberList = regList;  // for register classes that are subregisters of this class  list<RegisterClass> SubRegClassList = [];  code MethodProtos = [{}]; // to insert arbitrary code  code MethodBodies = [{}];  } |

要定义一个 RegisterClass，请使用以下 4 个参数：

* 定义的第一个参数是命名空间的名称。
* 第二个参数是在include/llvm/CodeGen/ValueTypes.td中定义的ValueType寄存器类型值的列表。定义的值包括整数类型（如i16、i32和i1对于布尔值）、浮点类型（f32、f64）和向量类型（例如， v8i16对于向量8 x i16）。一个RegisterClass 中的所有寄存器 必须具有相同的ValueType，但某些寄存器可能以不同的配置存储向量数据。例如，可以处理 128 位向量的寄存器可能能够处理 16 个 8 位整数元素、8 个 16 位整数、4 个 32 位整数等。
* RegisterClass定义的第三个参数指定寄存器在存储或加载到内存时所需的对齐方式。
* 最后一个参数regList，指定了哪些寄存器属于此类。如果未指定替代分配顺序方法，则regList 还定义寄存器分配器使用的分配顺序。除了使用 (add R0, R1, ...)简单地列出寄存器外，还可以使用更高级的集合运算符。有关详细信息，请参阅include/llvm/Target/Target.td。

在SparcRegisterInfo.td中，定义了三个RegisterClass对象： FPRegs、DFPRegs和IntRegs。对于所有三个寄存器类，第一个参数使用字符串“ SP”定义命名空间。 FPRegs 定义一组 32 个单精度浮点寄存器 ( F0to F31)；DFPRegs定义一组 16 个双精度寄存器 ( D0-D15)。

|  |
| --- |
| // F0, F1, F2, ..., F31  def FPRegs : RegisterClass<"SP", [f32], 32, (sequence "F%u", 0, 31)>;  def DFPRegs : RegisterClass<"SP", [f64], 64,  (add D0, D1, D2, D3, D4, D5, D6, D7, D8,  D9, D10, D11, D12, D13, D14, D15)>;  def IntRegs : RegisterClass<"SP", [i32], 32,  (add L0, L1, L2, L3, L4, L5, L6, L7,  I0, I1, I2, I3, I4, I5,  O0, O1, O2, O3, O4, O5, O7,  G1,  // Non-allocatable regs:  G2, G3, G4,  O6, // stack ptr  I6, // frame ptr  I7, // return address  G0, // constant zero  G5, G6, G7 // reserved for kernel  )>; |

与 TableGen 一起使用SparcRegisterInfo.td会生成多个输出文件，这些文件旨在包含在您编写的其他源代码中。 SparcRegisterInfo.td生成SparcGenRegisterInfo.h.inc，它应该包含在用于实现您编写的 SPARC 寄存器实现的头文件中 ( SparcRegisterInfo.h)。在 SparcGenRegisterInfo.h.inc中，定义了一个称为SparcGenRegisterInfo的新的结构，该结构使用TargetRegisterInfo作为它的基类。它还根据上面定义的寄存器类指定类型：DFPRegsClass、 FPRegsClass和IntRegsClass。

SparcRegisterInfo.td还生成SparcGenRegisterInfo.inc，它包含在在SPARC 寄存器实现SparcRegisterInfo.cpp 的底部。下面的代码仅显示生成的整数寄存器和关联的寄存器类。寄存器IntRegs的顺序反映了IntRegs目标描述文件中定义的顺序。

|  |
| --- |
| // IntRegs Register Class...  static const unsigned IntRegs[] = {  SP::L0, SP::L1, SP::L2, SP::L3, SP::L4, SP::L5,  SP::L6, SP::L7, SP::I0, SP::I1, SP::I2, SP::I3,  SP::I4, SP::I5, SP::O0, SP::O1, SP::O2, SP::O3,  SP::O4, SP::O5, SP::O7, SP::G1, SP::G2, SP::G3,  SP::G4, SP::O6, SP::I6, SP::I7, SP::G0, SP::G5,  SP::G6, SP::G7,  };  // IntRegsVTs Register Class Value Types...  static const MVT::ValueType IntRegsVTs[] = {  MVT::i32, MVT::Other  };  namespace SP { // Register class instances  DFPRegsClass DFPRegsRegClass;  FPRegsClass FPRegsRegClass;  IntRegsClass IntRegsRegClass;  ...  // IntRegs Sub-register Classes...  static const TargetRegisterClass\* const IntRegsSubRegClasses [] = {  NULL  };  ...  // IntRegs Super-register Classes..  static const TargetRegisterClass\* const IntRegsSuperRegClasses [] = {  NULL  };  ...  // IntRegs Register Class sub-classes...  static const TargetRegisterClass\* const IntRegsSubclasses [] = {  NULL  };  ...  // IntRegs Register Class super-classes...  static const TargetRegisterClass\* const IntRegsSuperclasses [] = {  NULL  };  IntRegsClass::IntRegsClass() : TargetRegisterClass(IntRegsRegClassID,  IntRegsVTs, IntRegsSubclasses, IntRegsSuperclasses, IntRegsSubRegClasses,  IntRegsSuperRegClasses, 4, 4, 1, IntRegs, IntRegs + 32) {}  } |

寄存器分配器将避免使用保留寄存器，并且在使用所有易失性寄存器之前不会使用被调用者保存的寄存器。这通常就足够了，但在某些情况下，可能需要提供自定义分配顺序。

### 实现一个TargetRegisterInfo的子类

最后一步是处理 XXXRegisterInfo 的代码部分，它实现了TargetRegisterInfo.h（请参阅[TargetRegisterInfo 类](https://llvm.org/docs/CodeGenerator.html" \l "targetregisterinfo)）中描述的接口。这些函数返回0、NULL或 false，除非被覆盖。以下是SparcRegisterInfo.cpp为 SPARC 实现覆盖的函数列表：

* getCalleeSavedRegs— 按照所需的被调用者保存堆栈帧偏移量的顺序返回被调用者保存的寄存器列表。
* getReservedRegs— 返回由物理寄存器编号索引的位集，指示特定寄存器是否不可用。
* hasFP— 返回一个布尔值，指示一个函数是否应该有一个专用的帧指针寄存器。
* eliminateCallFramePseudoInstr— 如果使用了调用帧设置或销毁伪指令，则可以调用它来消除它们。
* eliminateFrameIndex— 从可能使用它们的指令中消除抽象帧索引。
* emitPrologue— 将序言代码插入到函数中。
* emitEpilogue— 将结语代码插入函数中。

## 指令集

在代码生成的早期阶段，LLVM IR 代码被转换为一个带有节点的SelectionDAG，这些节点是包含目标指令的SDNode 类的实例。一个SDNode具有操作码、操作数、类型要求和操作属性。例如， 是一个可交换运算，从内存中加载一个运算。include/llvm/CodeGen/SelectionDAGNodes.h文件中描述了各种操作节点类型（ISD命名空间中NodeType枚举的值）。

TableGen 使用以下目标描述 ( .td) 输入文件来生成指令定义的大部分代码：

* Target.td—定义Instruction、Operand、InstrInfo和其他基本类的位置。
* TargetSelectionDAG.td— 由SelectionDAG指令选择生成器使用，包含SDTC\*类（选择 DAG 类型约束）、SelectionDAG节点的定义（例如imm, cond, bb, add, fadd, sub）和模式支持（Pattern, Pat, PatFrag, PatLeaf, ComplexPattern）
* XXXInstrFormats.td— 目标特定指令定义的模式。
* XXXInstrInfo.td— 指令集的指令模板、条件代码和指令的目标特定定义。对于架构修改，可以使用不同的文件名。例如，对于带有 SSE 指令的奔腾，该文件为X86InstrSSE.td，对于带有 MMX 的奔腾，该文件为X86InstrMMX.td。

还有一个特定于目标的XXX.td文件，其中XXX是目标的名称。该XXX.td文件包括其他.td输入文件，但其内容仅对子目标直接重要。

您应该描述一个具体的特定于目标的类XXXInstrInfo，它表示目标机器支持的机器指令。 XXXInstrInfo包含一组XXXInstrDescriptor对象，每个对象描述一条指令。一个指令描述符定义：

* 操作码助记符
* 操作数个数
* 隐式寄存器定义和使用列表
* 与目标无关的属性（例如内存访问，是可交换的）
* 特定目标的标志

Instruction 类（在Target.td 中定义）主要用作更复杂的指令类的基础。

|  |
| --- |
| class Instruction {  string Namespace = "";  dag OutOperandList; // A dag containing the MI def operand list.  dag InOperandList; // A dag containing the MI use operand list.  string AsmString = ""; // The .s format to print the instruction with.  list<dag> Pattern; // Set to the DAG pattern for this instruction.  list<Register> Uses = [];  list<Register> Defs = [];  list<Predicate> Predicates = []; // predicates turned into isel match code  ... remainder not shown for space ...  } |

SelectionDAG节点 ( SDNode) 应该包含一个对象，该对象表示在 XXXInstrInfo.td 中定义的特定于目标的指令。指令对象应表示来自目标机器体系结构手册的指令（例如 SPARC 目标的 SPARC 体系结构手册）。

体系结构手册中的单个指令通常被建模为多个目标指令，具体取决于其操作数。例如，手册可能会描述一条采用寄存器或立即操作数的加法指令。LLVM 目标可以使用名为ADDri和ADDrr 的两条指令对此进行建模。

您应该为每个指令类别定义一个类，并将每个操作码定义为该类别的子类，并带有适当的参数，例如，操作码的固定二进制编码和扩展操作码。您应该将寄存器位映射到它们被编码的指令的位（对于 JIT）。您还应该指定在使用自动汇编printer时如何打印指令。

如 SPARC Architecture Manual, Version 8 中所述，存在三种主要的 32 位指令格式。格式 1 仅用于CALL 指令。格式 2 用于条件代码和SETHI（设置寄存器的高位）指令的分支。格式 3 用于其他指令。

这些格式中的每一种在SparcInstrFormat.td中都有对应的类。 InstSP是其他指令类的基类。为更精确的格式指定了额外的基类：例如， 在 SparcInstrFormat.td 中，F2\_1 用于 SETHI，F2\_2 用于分支 。还有其他三个基类：F3\_1用于寄存器/寄存器操作、F3\_2用于寄存器/立即操作和F3\_3用于浮点操作。 SparcInstrInfo.td还添加了 合成 SPARC 指令的基类Pseudo。

SparcInstrInfo.td主要由 SPARC 目标的操作数和指令定义组成。在SparcInstrInfo.td中，以下目标描述文件条目，LDrr，定义了一个字从内存地址到寄存器（LD SPARC 操作码）的加载整数指令。第一个参数，值 3 ( 112 )，是此类操作的操作值。第二个参数（0000002）是LD/Load Word的具体操作值。第三个参数是输出目标，它是一个寄存器操作数，在Register 目标描述文件（IntRegs）中定义。

|  |
| --- |
| def LDrr : F3\_1 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMrr $rs1, $rs2):$addr),  "ld [$addr], $dst",  [(set i32:$dst, (load ADDRrr:$addr))]>; |

第四个参数是输入源，它使用 前面定义的地址操作数MEMrr，该操作数在SparcInstrInfo.td中靠前的位置定义：

|  |
| --- |
| def MEMrr : Operand<i32> {  let PrintMethod = "printMemOperand";  let MIOperandInfo = (ops IntRegs, IntRegs);  } |

第五个参数是汇编printer使用的字符串，在实现汇编printer接口之前可以保留为空字符串。第六个也是最后一个参数是在[LLVM 目标独立代码生成器](https://llvm.org/docs/CodeGenerator.html)中描述的 SelectionDAG 选择阶段用于匹配指令的模式。该参数在下一节[指令选择器](https://llvm.org/docs/WritingAnLLVMBackend.html" \l "instruction-selector)中有详细说明。

指令类定义不会为不同的操作数类型重载，因此寄存器、内存或立即值操作数需要不同版本的指令。例如，要对从立即操作数到寄存器的字执行加载整数指令，定义了以下指令类：

|  |
| --- |
| def LDri : F3\_2 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMri $rs1, $simm13):$addr),  "ld [$addr], $dst",  [(set i32:$rd, (load ADDRri:$addr))]>; |

为这么多类似的指令编写这些定义可能涉及大量的剪切和粘贴。在.td文件中，该multiclass指令允许创建模板以一次定义多个指令类（使用该 defm指令）。例如在 SparcInstrInfo.td中，multiclass 模式F3\_12被定义为每次 F3\_12调用时创建 2 个指令类：

|  |
| --- |
| multiclass F3\_12 <string OpcStr, bits<6> Op3Val, SDNode OpNode> {  def rr : F3\_1 <2, Op3Val,  (outs IntRegs:$rd), (ins IntRegs:$rs1, IntRegs:$rs1),  !strconcat(OpcStr, " $rs1, $rs2, $rd"),  [(set i32:$rd, (OpNode i32:$rs1, i32:$rs2))]>;  def ri : F3\_2 <2, Op3Val,  (outs IntRegs:$rd), (ins IntRegs:$rs1, i32imm:$simm13),  !strconcat(OpcStr, " $rs1, $simm13, $rd"),  [(set i32:$rd, (OpNode i32:$rs1, simm13:$simm13))]>;  } |

因此，当该defm指令用于XOR 和 ADD 指令时，如下所示，它会创建四个指令对象：XORrr、 XORri、ADDrr和ADDri。

|  |
| --- |
| defm XOR : F3\_12<"xor", 0b000011, xor>;  defm ADD : F3\_12<"add", 0b000000, add>; |

SparcInstrInfo.td还包括分支指令引用的条件代码的定义。以下SparcInstrInfo.td中的定义 指示 SPARC 条件代码的位位置。例如，第 10位表示整数的“大于”条件，第 22位表示浮点数的“大于”条件。

|  |
| --- |
| def ICC\_NE : ICC\_VAL< 9>; // Not Equal  def ICC\_E : ICC\_VAL< 1>; // Equal  def ICC\_G : ICC\_VAL<10>; // Greater  ...  def FCC\_U : FCC\_VAL<23>; // Unordered  def FCC\_G : FCC\_VAL<22>; // Greater  def FCC\_UG : FCC\_VAL<21>; // Unordered or Greater  ... |

（请注意，Sparc.h还定义了对应于相同 SPARC 条件代码的枚举。必须注意确保Sparc.h中的值 对应于SparcInstrInfo.td中的值。即SPCC::ICC\_NE = 9，SPCC::FCC\_U = 23

，等等） 。

### 指令操作数映射

代码生成器后端将指令操作数映射到指令中的字段。每当指令编码Inst 中的一个位被分配给没有具体值的字段时，outs 或 ins列表中的操作数应该具有匹配的名称。然后这个操作数填充那个未定义的字段。例如，Sparc 目标将XNORrr指令定义为 F3\_1， 该F3\_1具有三个操作数的格式指令：输出$rd、输入$rs1和$rs2。

|  |
| --- |
| def XNORrr : F3\_1<2, 0b000111,  (outs IntRegs:$rd), (ins IntRegs:$rs1, IntRegs:$rs2),  "xnor $rs1, $rs2, $rd",  [(set i32:$rd, (not (xor i32:$rs1, i32:$rs2)))]>; |

isSparcInstrFormats.td中的指令模板显示了F3\_1的基类InstSP 。

|  |
| --- |
| class InstSP<dag outs, dag ins, string asmstr, list<dag> pattern> : Instruction {  field bits<32> Inst;  let Namespace = "SP";  bits<2> op;  let Inst{31-30} = op;  dag OutOperandList = outs; dag InOperandList = ins;  let AsmString = asmstr;  let Pattern = pattern;  } |

InstSP定义op字段，并使用它来定义指令的位 30 和 31，但不为其赋值。

|  |
| --- |
| class F3<dag outs, dag ins, string asmstr, list<dag> pattern>  : InstSP<outs, ins, asmstr, pattern> {  bits<5> rd;  bits<6> op3;  bits<5> rs1;  let op{1} = 1; // Op = 2 or 3  let Inst{29-25} = rd;  let Inst{24-19} = op3;  let Inst{18-14} = rs1;  } |

F3定义rd、op3和rs1字段，并在指令中使用它们，并且再次不赋值。

|  |
| --- |
| class F3\_1<bits<2> opVal, bits<6> op3val, dag outs, dag ins,  string asmstr, list<dag> pattern> : F3<outs, ins, asmstr, pattern> {  bits<8> asi = 0; // asi not currently used  bits<5> rs2;  let op = opVal;  let op3 = op3val;  let Inst{13} = 0; // i field = 0  let Inst{12-5} = asi; // address space identifier  let Inst{4-0} = rs2;  } |

F3\_1为op和op3字段赋值，并定义rs2 字段。因此，F3\_1格式指令需要rd、rs1和rs2的定义，以便完全指定指令编码。

然后，XNORrr指令在其 OutOperandList 和 InOperandList 中提供这三个操作数，它们绑定到相应的字段，从而完成指令编码。

对于某些指令，单个操作数可能包含子操作数。如前所示，该指令LDrr使用 MEMrr类型的输入操作数。此操作数类型包含两个寄存器子操作数，由 MIOperandInfo值定义为(ops IntRegs, IntRegs) 。

|  |
| --- |
| def LDrr : F3\_1 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMrr $rs1, $rs2):$addr),  "ld [$addr], $dst",  [(set i32:$dst, (load ADDRrr:$addr))]>; |

由于此指令也是F3\_1格式，因此它也需要名为 rd、rs1和rs2的操作数。为了实现这一点，一个复杂的操作数可以选择为其每个子操作数命名。在此示例中，MEMrr的第一个子操作数被命名$rs1，第二个子操作数$rs2，和整个操作数也被命名为 $addr。

当特定指令不使用指令格式定义的所有操作数时，常量值可能会绑定到一个或所有操作数。例如，该RDASR指令只接受一个寄存器操作数，因此我们将常量零分配给rs2：

|  |
| --- |
| let rs2 = 0 in  def RDASR : F3\_1<2, 0b101000,  (outs IntRegs:$rd), (ins ASRRegs:$rs1),  "rd $rs1, $rd", []>; |

#### 指令操作数名称映射

TableGen 还将生成一个名为 getNamedOperandIdx() 的函数，该函数可用于根据其 TableGen 名称在 MachineInstr 中查找操作数的索引。在指令的 TableGen 定义中设置 UseNamedOperandTable 位，会将其所有操作数添加到 llvm::XXX:OpName 命名空间中的枚举中，并为其添加一个条目到 OperandMap 表中，可以使用 getNamedOperandIdx() 查询

|  |
| --- |
| int DstIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::dst); // => 0  int BIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::b); // => 1  int CIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::c); // => 2  int DIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::d); // => -1  ... |

OpName 枚举中的条目是逐字从 TableGen 定义中获取的，因此具有小写名称的操作数将在枚举中具有小写条目。

要在后端中包含 getNamedOperandIdx() 函数，您需要在 XXXInstrInfo.cpp 和 XXXInstrInfo.h 中定义一些预处理器宏。例如：

XXXInstrInfo.cpp:

|  |
| --- |
| #define GET\_INSTRINFO\_NAMED\_OPS // For getNamedOperandIdx() function  #include "XXXGenInstrInfo.inc" |

XXXInstrInfo.h:

|  |
| --- |
| #define GET\_INSTRINFO\_OPERAND\_ENUM // For OpName enum  #include "XXXGenInstrInfo.inc"  namespace XXX {  int16\_t getNamedOperandIdx(uint16\_t Opcode, uint16\_t NamedIndex);  } // End namespace XXX |

#### 指令操作数类型

TableGen 还将在 llvm::XXX::OpTypes 命名空间中生成一个枚举，该枚举由后端定义的所有命名操作数类型组成。include/llvm/Target/Target.td为所有目标定义了一些常见的立即操作数类型（例如， i8、i32、i64、f32、f64），并且在每个目标的 OpTypes 枚举中可用。此外，枚举中只出现命名的操作数类型：忽略匿名类型。例如，X86 后端定义了 TableGen Operand类的两个实例，brtarget 和brtarget8，它们代表分支目标操作数：

|  |
| --- |
| def brtarget : Operand<OtherVT>;  def brtarget8 : Operand<OtherVT>; |

这导致：

|  |
| --- |
| namespace X86 {namespace OpTypes {enum OperandType {  ...  brtarget,  brtarget8,  ...  i32imm,  i64imm,  ...  OPERAND\_TYPE\_LIST\_END} // End namespace OpTypes  } // End namespace X86 |

在典型的 TableGen 方式中，要使用枚举，您需要定义一个预处理器宏：

|  |
| --- |
| #define GET\_INSTRINFO\_OPERAND\_TYPES\_ENUM // For OpTypes enum  #include "XXXGenInstrInfo.inc" |

### 指令调度

可以使用 MCDesc::getSchedClass() 查询指令行程。该值可以由 TableGen 在 XXXGenInstrInfo.inc 中生成的 llvm::XXX::Sched 命名空间中的枚举命名。调度类的名称与 XXXSchedule.td 中提供的相同，加上默认的 NoItinerary 类。

调度模型由 TableGen 和 SubtargetEmitter 使用CodeGenSchedModels类生成。这与指定机器资源使用的行程方法不同。工具utils/schedcover.py 可用于确定调度模型描述涵盖了哪些指令，哪些没有。第一步是使用以下指令创建输出文件。然后在输出文件上运行schedcover.py：

|  |
| --- |
| $ <src>/utils/schedcover.py <build>/lib/Target/AArch64/tblGenSubtarget.with  instruction, default, CortexA53Model, CortexA57Model, CycloneModel, ExynosM3Model, FalkorModel, KryoModel, ThunderX2T99Model, ThunderXT8XModel  ABSv16i8, WriteV, , , CyWriteV3, M3WriteNMISC1, FalkorWr\_2VXVY\_2cyc, KryoWrite\_2cyc\_XY\_XY\_150ln, ,  ABSv1i64, WriteV, , , CyWriteV3, M3WriteNMISC1, FalkorWr\_1VXVY\_2cyc, KryoWrite\_2cyc\_XY\_noRSV\_67ln, ,  ... |

要捕获生成调度模型的调试输出，请切换到适当的目标目录并使用以下命令：带有subtarget-emitter调试选项的命令：

|  |
| --- |
| $ <build>/bin/llvm-tblgen -debug-only=subtarget-emitter -gen-subtarget \  -I <src>/lib/Target/<target> -I <src>/include \  -I <src>/lib/Target <src>/lib/Target/<target>/<target>.td \  -o <build>/lib/Target/<target>/<target>GenSubtargetInfo.inc.tmp \  > tblGenSubtarget.dbg 2>&1 |

其中<build>是构建目录，src是源目录，<target>是目标的名称。要仔细检查上面的命令是否需要，可以使用以下命令从构建中捕获确切的 TableGen 命令：

|  |
| --- |
| $ VERBOSE=1 make ... |

并在输出中搜索llvm-tblgen命令。

### 指令关系映射

此 TableGen 功能用于将指令相互关联。当您有多种指令格式并且需要在选择指令后在它们之间切换时，它特别有用。整个功能由关系模型驱动，关系模型可以根据特定于目标的指令集在XXXInstrInfo.td文件中定义。关系模型是使用InstrMapping类作为基类定义的。TableGen 解析所有模型并使用指定信息生成指令关系图。关系映射作为XXXGenInstrInfo.inc文件中的表与查询它们的函数一起发出。有关如何使用此功能的详细信息，请参阅[如何使用指令映射](https://llvm.org/docs/HowToUseInstrMappings.html)。

### 实现一个TargetInstrInfo的子类

最后一步是提交 XXXInstrInfo 的代码部分，它实现了TargetInstrInfo.h（请参阅[TargetInstrInfo](https://llvm.org/docs/CodeGenerator.html" \l "targetinstrinfo) 类）中描述的接口。除非被覆盖，否则这些函数返回0或布尔值或它们断言。以下是 SparcInstrInfo.cpp中为 SPARC 实现覆盖的函数列表：

* isLoadFromStackSlot— 如果指定的机器指令是直接从堆栈槽加载，则返回目标寄存器号和堆栈槽的 FrameIndex。
* isStoreToStackSlot— 如果指定的机器指令是直接存储到堆栈槽，则返回目标寄存器号和堆栈槽的 FrameIndex。
* copyPhysReg— 在一对物理寄存器之间复制值。
* storeRegToStackSlot— 将寄存器值存储到堆栈槽。
* loadRegFromStackSlot— 从堆栈槽加载寄存器值。
* storeRegToAddr— 将寄存器值存储到内存中。
* loadRegFromAddr— 从内存中加载一个寄存器值。
* foldMemoryOperand— 尝试组合指定操作数的任何加载或存储指令的指令。

### 分支折叠和 If 转换

可以通过组合指令或消除从未达到的指令来提高性能。可以实现XXXInstrInfo中的analyzeBranch方法来检查条件指令并移除不必要的指令。analyzeBranch查看机器基本块 (MBB) 的末尾以寻找改进机会，例如分支折叠和 if 转换。BranchFolder和IfConverter 机器函数通过（参见源文件BranchFolding.cpp和lib/CodeGen目录中的IfConversion.cpp）调用analyzeBranch 来改进表示指令的控制流图。

可以将analyzeBranch的（针对 ARM、Alpha 和 X86）几种实现作为您自己的analyzeBranch实现的模型进行检查。由于 SPARC 没有实现有用的 analyzeBranch， ARM 目标实现如下所示。

analyzeBranch返回一个布尔值并接受四个参数：

* MachineBasicBlock &MBB— 要检查的传入块。
* MachineBasicBlock \*&TBB— 返回的目标块。对于计算结果为 true 的条件分支，TBB是目的地。
* MachineBasicBlock \*&FBB— 对于计算结果为 false 的条件分支，FBB作为目标返回。
* std::vector<MachineOperand> &Cond— 用于评估条件分支条件的操作数列表。

在最简单的情况下，如果一个块在没有分支的情况下结束，那么它就会落到后继块。TBB 或FBB都没有指定目标块，因此两个参数都返回NULL。analyzeBranch的开头 （请参阅下面的 ARM 目标代码）显示函数参数和最简单情况的代码。

|  |
| --- |
| bool ARMInstrInfo::analyzeBranch(MachineBasicBlock &MBB,  MachineBasicBlock \*&TBB,  MachineBasicBlock \*&FBB,  std::vector<MachineOperand> &Cond) const{  MachineBasicBlock::iterator I = MBB.end();  if (I == MBB.begin() || !isUnpredicatedTerminator(--I))  return false; |

如果块以单个无条件分支指令结束，则 analyzeBranch（如下所示）应在TBB参数中返回该分支的目的地。

|  |
| --- |
| if (LastOpc == ARM::B || LastOpc == ARM::tB) {  TBB = LastInst->getOperand(0).getMBB();  return false;  } |

如果一个块以两个无条件分支结束，则永远不会到达第二个分支。在那种情况下，如下所示，删除最后一个分支指令并返回TBB参数中的倒数第二个分支。

|  |
| --- |
| if ((SecondLastOpc == ARM::B || SecondLastOpc == ARM::tB) &&  (LastOpc == ARM::B || LastOpc == ARM::tB)) {  TBB = SecondLastInst->getOperand(0).getMBB();  I = LastInst;  I->eraseFromParent();  return false;  } |

如果条件评估为假，则块可能以单个条件分支指令结束，该指令会落入后续块。在这种情况下， analyzeBranch（如下所示）应该在TBB参数中返回该条件分支的目的地和Cond参数中的操作数列表以评估条件。

|  |
| --- |
| if (LastOpc == ARM::Bcc || LastOpc == ARM::tBcc) {  // Block ends with fall-through condbranch.  TBB = LastInst->getOperand(0).getMBB();  Cond.push\_back(LastInst->getOperand(1));  Cond.push\_back(LastInst->getOperand(2));  return false;  } |

如果一个块以条件分支和随后的无条件分支结束，那么analyzeBranch（如下所示）应该在TBB参数中返回条件分支目的地（假设它对应于“true”的条件评估）和 在FBB参数中无条件分支目的地（相应的条件评估“ false”）。评估条件的操作数列表应在Cond 参数中返回。

|  |
| --- |
| unsigned SecondLastOpc = SecondLastInst->getOpcode();  if ((SecondLastOpc == ARM::Bcc && LastOpc == ARM::B) ||  (SecondLastOpc == ARM::tBcc && LastOpc == ARM::tB)) {  TBB = SecondLastInst->getOperand(0).getMBB();  Cond.push\_back(SecondLastInst->getOperand(1));  Cond.push\_back(SecondLastInst->getOperand(2));  FBB = LastInst->getOperand(0).getMBB();  return false;  } |

对于后两种情况（以单个条件分支结束或以一个条件和一个无条件分支结束）， Cond参数中返回的操作数可以传递给其他指令的方法以创建新分支或执行其他操作。analyzeBranch 的实现需要辅助方法removeBranch和insertBranch来管理后续操作。

analyzeBranch在大多数情况下应该返回 false 表示成功。 analyzeBranch应该只在该方法对要做什么感到困惑时才返回 true，例如，如果一个块具有三个终止分支。 analyzeBranch如果遇到无法处理的终止符（例如间接分支），则可能会返回 true。

## 指令选择器

LLVM 使用一个 SelectionDAG来表示 LLVM IR 指令，而 SelectionDAG 的节点理想地表示本机目标指令。在代码生成期间，执行指令选择过程以将非本机 DAG 指令转换为本机目标特定指令。XXXISelDAGToDAG.cpp中描述的传递用于匹配模式和执行 DAG 到 DAG 指令选择。可选地，可以定义一个pass（在 XXXBranchSelector.cpp 中）以对分支指令执行类似的 DAG 到 DAG 操作。后来，XXXISelLowering.cpp中的代码替换或删除了 SelectionDAG中本机不支持的操作和数据类型，进行合法化。

TableGen 使用以下目标描述输入文件生成指令选择代码：

* XXXInstrInfo.td— 包含目标特定指令集中的指令定义，生成XXXGenDAGISel.inc，由XXXISelDAGToDAG.cpp include。
* XXXCallingConv.td— 包含目标体系结构的调用和返回值约定，并生成XXXGenCallingConv.inc，包含在XXXISelLowering.cpp。

指令选择传递的实现必须包含一个标头，该标头声明 FunctionPass类或FunctionPass的子类。在 XXXTargetMachine.cpp中，传递管理器 (PM) 应该将每个指令选择传递添加到要运行的传递队列中。

LLVM 静态编译器 ( llc) 是可视化 DAG 内容的出色工具。要显示SelectionDAG的特定处理阶段之前或之后，请使用命令行选项llc，如 [SelectionDAG Instruction Selection Process](https://llvm.org/docs/CodeGenerator.html" \l "selectiondag-process)中所述。

为了描述指令选择器的行为，您应该将用于降低 LLVM 代码的模式添加到 SelectionDAG作为XXXInstrInfo.td中指令定义的最后一个参数。例如，在 SparcInstrInfo.td中，这个条目定义了一个寄存器存储操作，最后一个参数描述了一个带有存储 DAG 运算符的模式。

|  |
| --- |
| def STrr : F3\_1< 3, 0b000100, (outs), (ins MEMrr:$addr, IntRegs:$src),  "st $src, [$addr]", [(store i32:$src, ADDRrr:$addr)]>; |

ADDRrr是一种内存模式，也在 SparcInstrInfo.td 中定义：

|  |
| --- |
| def ADDRrr : ComplexPattern<i32, 2, "SelectADDRrr", [], []>; |

ADDRrr的定义指向了SelectADDRrr，它是在 Instructor Selector 的实现中定义的函数（如 SparcISelDAGToDAG.cpp）。

在lib/Target/TargetSelectionDAG.td中，store 的 DAG 算子定义如下：

|  |
| --- |
| def store : PatFrag<(ops node:$val, node:$ptr),  (unindexedstore node:$val, node:$ptr)> {  let IsStore = true;  let IsTruncStore = false;  } |

XXXInstrInfo.td还生成（在XXXGenDAGISel.inc中） SelectCode 方法，该方法用于为指令调用适当处理方法。在此示例中，SelectCode为 ISD::STORE操作码调用Select\_ISD\_STORE。

|  |
| --- |
| SDNode \*SelectCode(SDValue N) {  ...  MVT::ValueType NVT = N.getNode()->getValueType(0);  switch (N.getOpcode()) {  case ISD::STORE: {  switch (NVT) {  default:  return Select\_ISD\_STORE(N);  break;  }  break;  }  ... |

跟STrr 的模式匹配，因此在XXXGenDAGISel.inc 的其他地方，为 Select\_ISD\_STORE 创建的代码就是STrr的代码。该Emit\_22方法也被生成在XXXGenDAGISel.inc中，以完成对这条指令的处理。

|  |
| --- |
| SDNode \*Select\_ISD\_STORE(const SDValue &N) {  SDValue Chain = N.getOperand(0);  if (Predicate\_store(N.getNode())) {  SDValue N1 = N.getOperand(1);  SDValue N2 = N.getOperand(2);  SDValue CPTmp0;  SDValue CPTmp1;  // Pattern: (st:void i32:i32:$src,  // ADDRrr:i32:$addr)<<P:Predicate\_store>>  // Emits: (STrr:void ADDRrr:i32:$addr, IntRegs:i32:$src)  // Pattern complexity = 13 cost = 1 size = 0  if (SelectADDRrr(N, N2, CPTmp0, CPTmp1) &&  N1.getNode()->getValueType(0) == MVT::i32 &&  N2.getNode()->getValueType(0) == MVT::i32) {  return Emit\_22(N, SP::STrr, CPTmp0, CPTmp1);  }  ... |

### SelectionDAG 合法化阶段

合法化阶段将 DAG 转换为使用目标本机支持的类型和操作。对于本机不支持的类型和操作，您需要将代码添加到特定于目标的XXXTargetLowering实现，以将不支持的类型和操作转换为支持的类型和操作。

在XXXTargetLowering类的构造函数中，首先使用 addRegisterClass方法指定支持哪些类型以及与哪些寄存器类相关联。寄存器类的代码由 TableGen 从中生成XXXRegisterInfo.td并放置在 XXXGenRegisterInfo.h.inc. 例如，SparcTargetLowering 类（in SparcISelLowering.cpp）的构造函数的实现从以下代码开始：

|  |
| --- |
| addRegisterClass(MVT::i32, SP::IntRegsRegisterClass);  addRegisterClass(MVT::f32, SP::FPRegsRegisterClass);  addRegisterClass(MVT::f64, SP::DFPRegsRegisterClass); |

您应该检查ISD命名空间 (include/llvm/CodeGen/SelectionDAGNodes.h )中的节点类型并确定目标本机支持哪些操作。对于本机没有支持的操作，向XXXTargetLowering类的构造函数添加回调，以便指令选择过程知道要做什么。TargetLowering 类回调方法（在llvm/Target/TargetLowering.h 中声明）是：

* setOperationAction— 一般的操作。
* setLoadExtAction— 带扩展的加载。
* setTruncStoreAction— 截断存储。
* setIndexedLoadAction— 索引负载。
* setIndexedStoreAction— 索引存储。
* setConvertAction— 类型转换。
* setCondCodeAction— 支持给定的条件代码。

注意：在旧版本中，使用 setLoadXAction代替 setLoadExtAction. 此外，在旧版本上，setCondCodeAction可能不受支持。检查您的版本以查看具体支持哪些方法。

这些回调用于确定操作是否适用于指定类型（或多种类型）。在所有情况下，第三个参数都是 LegalAction枚举类型值：Promote、Expand、Custom或 Legal。 SparcISelLowering.cpp包含所有四个 LegalAction值的示例。

#### 推广

对于给定类型没有本机支持的操作，指定的类型可能会提升为受支持的更大类型。例如，SPARC 不支持布尔值 ( i1 type) 的符号扩展加载，因此在 SparcISelLowering.cpp下面的第三个参数，Promote，在加载之前将 i1 type 值更为大类型。

|  |
| --- |
| setLoadExtAction(ISD::SEXTLOAD, MVT::i1, Promote); |

#### 展开

对于没有本机支持的类型，可能需要进一步分解值，而不是提升值。对于没有本机支持的操作，可以使用其他操作的组合来达到类似的效果。在 SPARC 中，浮点正弦和余弦三角运算通过扩展到其他运算来支持，如第三个参数所示，Expand, 到 setOperationAction：

|  |
| --- |
| setOperationAction(ISD::FSIN, MVT::f32, Expand);  setOperationAction(ISD::FCOS, MVT::f32, Expand); |

#### 自定义

对于某些操作，简单的类型提升或操作扩展可能是不够的。在某些情况下，必须实现特殊的内部功能。

例如，常量值可能需要特殊处理，或者操作可能需要溢出和恢复堆栈中的寄存器，并使用寄存器分配器。

如下面的SparcISelLowering.cpp代码所示，要执行从浮点值到有符号整数的类型转换。

* 首先， 应该将调用setOperationAction， 其中，Custom为第三个参数：

|  |
| --- |
| setOperationAction(ISD::FP\_TO\_SINT, MVT::i32, Custom); |

* 在LowerOperation方法中，对于每一个Custom操作，都应该添加一个case语句来指明调用什么函数。在以下代码中，FP\_TO\_SINT操作码将调用该LowerFP\_TO\_SINT方法：

|  |
| --- |
| SDValue SparcTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) {  switch (Op.getOpcode()) {  case ISD::FP\_TO\_SINT: return LowerFP\_TO\_SINT(Op, DAG);  ...  }  } |

* 最后LowerFP\_TO\_SINT实现方法，使用浮点寄存器将浮点值转换为整数。

|  |
| --- |
| static SDValue LowerFP\_TO\_SINT(SDValue Op, SelectionDAG &DAG) {  assert(Op.getValueType() == MVT::i32);  Op = DAG.getNode(SPISD::FTOI, MVT::f32, Op.getOperand(0));  return DAG.getNode(ISD::BITCAST, MVT::i32, Op);  } |

#### Legal

Legal LegalizeAction枚举值仅表示本机支持某个操作。 Legal表示默认条件，所以很少使用。在SparcISelLowering.cpp中，仅 SPARC v9 原生支持CTPOP操作（对整数中的位集进行计数的操作）。以下代码为 non-v9 SPARC 实现启用Expand转换技术。

|  |
| --- |
| setOperationAction(ISD::CTPOP, MVT::i32, Expand);  ...  if (TM.getSubtarget<SparcSubtarget>().isV9())  setOperationAction(ISD::CTPOP, MVT::i32, Legal); |

### 调用约定

为了支持特定目标的调用约定，XXXGenCallingConv.td使用在lib/Target/TargetCallingConv.td中定义的接口(例如，CCIfType 和 CCAssignToReg)。TableGen 可以获取目标描述符文件XXXGenCallingConv.td并生成头文件 XXXGenCallingConv.inc，该XXXGenCallingConv.inc文件通常包含在 XXXISelLowering.cpp，您可以使用TargetCallingConv.td中的接口来指定：

* 参数分配的顺序。
* 放置参数和返回值的位置（即在堆栈上或在寄存器中）。
* 可以使用哪些寄存器。
* 调用者或被调用者是否展开堆栈。

以下示例演示了CCIfType和 CCAssignToReg接口的使用。如果CCIfType谓词为真（即，如果当前参数的类型为f32 或 f64），则执行操作。在这种情况下，CCAssignToReg操作将参数值分配给第一个可用的寄存器：要么R0要么R1。

|  |
| --- |
| CCIfType<[f32,f64], CCAssignToReg<[R0, R1]>> |

SparcCallingConv.td包含特定于目标的返回值调用约定 ( RetCC\_Sparc32) 和基本 32-bit C 调用约定 ( CC\_Sparc32) 的定义。（如下所示）RetCC\_Sparc32的定义指示哪些寄存器用于指定的标量返回类型。单精度浮点数返回到寄存器F0，双精度浮点数返回到寄存器D0。在寄存器I0或I1中返回一个 32 位整数。

|  |
| --- |
| def RetCC\_Sparc32 : CallingConv<[  CCIfType<[i32], CCAssignToReg<[I0, I1]>>,  CCIfType<[f32], CCAssignToReg<[F0]>>,  CCIfType<[f64], CCAssignToReg<[D0]>>  ]>; |

SparcCallingConv.td中的CC\_Sparc32的定义引入了 CCAssignToStack，它将值分配给具有指定大小和对齐方式的栈槽。在下面的示例中，第一个参数， 4，指示插槽的大小，第二个参数也是 4，指示沿 4 字节单元的堆栈对齐。（特殊情况：如果大小为零，则使用 ABI 大小；如果对齐为零，则使用 ABI 对齐。）

|  |
| --- |
| def CC\_Sparc32 : CallingConv<[  // All arguments get passed in integer registers if there is space.  CCIfType<[i32, f32, f64], CCAssignToReg<[I0, I1, I2, I3, I4, I5]>>,  CCAssignToStack<4, 4>  ]>; |

CCDelegateTo是另一个常用的接口，它试图找到指定的子调用约定，如果找到匹配项，则调用它。在下面的示例中（在 X86CallingConv.td中），  RetCC\_X86\_32\_C的定义以CCDelegateTo结尾。在将当前值分配给寄存器ST0或ST1后，将调用 RetCC\_X86Common。

|  |
| --- |
| def RetCC\_X86\_32\_C : CallingConv<[  CCIfType<[f32], CCAssignToReg<[ST0, ST1]>>,  CCIfType<[f64], CCAssignToReg<[ST0, ST1]>>,  CCDelegateTo<RetCC\_X86Common>  ]>; |

CCIfCC是一个尝试将给定名称与当前调用约定相匹配的接口。如果名称标识当前调用约定，则调用指定的操作。在以下示例中（在 X86CallingConv.td中 ），如果正在使用Fast调用约定，则调用RetCC\_X86\_32\_Fast。如果正在使用SSECall调用约定，则调用RetCC\_X86\_32\_SSE。

|  |
| --- |
| def RetCC\_X86\_32 : CallingConv<[  CCIfCC<"CallingConv::Fast", CCDelegateTo<RetCC\_X86\_32\_Fast>>,  CCIfCC<"CallingConv::X86\_SSECall", CCDelegateTo<RetCC\_X86\_32\_SSE>>,  CCDelegateTo<RetCC\_X86\_32\_C>  ]>; |

CCAssignToRegAndStack与 CCAssignToReg 相同，但在使用某些寄存器时也会分配一个栈槽。基本上，它的工作方式如下 ：

|  |
| --- |
| CCIf<CCAssignToReg<regList>, CCAssignToStack<size, align>>  class CCAssignToRegAndStack<list<Register> regList, int size, int align>  : CCAssignToReg<regList> {  int Size = size;  int Align = align;  } |

其他调用约定接口包括：

* CCIf <predicate, action>— 如果谓词匹配，则应用操作。
* CCIfInReg <action>— 如果参数标有“ inreg”属性，则应用操作。
* CCIfNest <action>— 如果参数标有“ nest”属性，则应用操作。
* CCIfNotVarArg <action>— 如果当前函数不采用可变数量的参数，则应用该操作。
* CCAssignToRegWithShadow <registerList, shadowList>— 类似于 CCAssignToReg，但带有一个影子寄存器列表。
* CCPassByVal <size, align>— 将值分配给具有最小指定大小和对齐方式的堆栈槽。
* CCPromoteToType <type>— 将当前值提升为指定类型。
* CallingConv <[actions]>— 定义支持的每个调用约定。

## 汇编printer

在代码发射阶段，代码生成器可以利用 LLVM pass 来生成汇编输出。为此，您需要使用以下步骤为目标机器实现将 LLVM IR 转换为 GAS 格式汇编语言的打印机代码：

* 为目标定义所有汇编字符串，将它们添加到XXXInstrInfo.td文件中定义的指令中。（请参阅 [指令集](https://llvm.org/docs/WritingAnLLVMBackend.html" \l "instruction-set)。）TableGen 将生成一个输出文件 ( XXXGenAsmWriter.inc)，其中包含该XXXAsmPrinter类的 printInstruction 方法的实现。
* 编写 XXXTargetAsmInfo.h，其中包含XXXTargetAsmInfo类（TargetAsmInfo 的子类）的基本声明。
* 编写XXXTargetAsmInfo.cpp，其中包含特定于目标的 TargetAsmInfo属性值，有时还包含方法的新实现。
* 编写 XXXAsmPrinter.cpp，它实现执行 LLVM-to-assembly 转换的 AsmPrinter 类。

XXXTargetAsmInfo.h 中的代码通常是在XXXTargetAsmInfo.cpp 中使用的XXXTargetAsmInfo类的简单声明。同样， XXXTargetAsmInfo.cpp通常有一些XXXTargetAsmInfo 替代值的声明来覆盖TargetAsmInfo.cpp 的默认值。 例如， 在SparcTargetAsmInfo.cpp：

|  |
| --- |
| SparcTargetAsmInfo::SparcTargetAsmInfo(const SparcTargetMachine &TM) {  Data16bitsDirective = "\t.half\t";  Data32bitsDirective = "\t.word\t";  Data64bitsDirective = 0; // .xword is only supported by V9.  ZeroDirective = "\t.skip\t";  CommentString = "!";  ConstantPoolSection = "\t.section \".rodata\",#alloc\n";  } |

X86 汇编printer实现 ( X86TargetAsmInfo) 是目标特定TargetAsmInfo类使用重写方法的示例： ExpandInlineAsm。

AsmPrinter的目标特定实现写在XXXAsmPrinter.cpp 中 ，它实现了将 LLVM-to-assembly 转换的AsmPrinter类。实现必须包括以下具有AsmPrinter和 MachineFunctionPass类声明的标头。MachineFunctionPass是FunctionPass 的子 类。

|  |
| --- |
| #include "llvm/CodeGen/AsmPrinter.h"  #include "llvm/CodeGen/MachineFunctionPass.h" |

作为一个FunctionPass，AsmPrinter首先调用doInitialization设置AsmPrinter。在SparcAsmPrinter中，Mangler对象被实例化以处理变量名称。

在XXXAsmPrinter.cpp中，必须为 XXXAsmPrinter实现 runOnMachineFunction方法（在MachineFunctionPass 中声明 ）。在 MachineFunctionPass中，runOnFunction方法调用 runOnMachineFunction。不同的目标特定实现 runOnMachineFunction，但通常执行以下操作来处理每个MachineFunction：

* 调用SetupMachineFunction以执行初始化。
* 调用EmitConstantPool以打印出（到输出流）已溢出到内存的常量。
* 调用EmitJumpTableInfo以打印出当前函数使用的跳转表。
* 打印出当前函数的标签。
* 打印出函数的代码，包括基本块标签和指令的汇编（使用printInstruction）

XXXAsmPrinter实现还必须包括由TableGen 生成的代码，该代码是XXXGenAsmWriter.inc文件中输出的。 XXXGenAsmWriter.inc中的代码 包含可以调用这些方法的printInstruction 方法的实现：

* printOperand
* printMemOperand
* printCCOperand（对于条件语句）
* printDataDirective
* printDeclare
* printImplicitDef
* printInlineAsm

printDeclare、printImplicitDef、 printInlineAsm和printLabelin的实现在AsmPrinter.cpp文件中，通常适用于打印组件，不需要重写。

该printOperand方法通过操作数类型的长switch/case 语句实现：寄存器、立即数、基本块、外部符号、全局地址、常量池索引或跳转表索引。对于带有内存地址操作数的指令，应该实现printMemOperand方法以生成正确的输出。同样， printCCOperand应该用于打印条件操作数。

应该在 XXXAsmPrinter中覆盖doFinalization，并且应该调用doFinalization来关闭汇编printer。在 doFinalization期间，全局变量和常量被打印到输出。

## 子目标支持

子目标支持用于通知给定芯片组的指令集变化的代码生成过程。例如，提供的 LLVM SPARC 实现涵盖了 SPARC 微处理器架构的三个主要版本：Version 8（V8，32 位架构）、Version 9（V9，64 位架构）和 UltraSPARC 架构。V8 有 16 个双精度浮点寄存器，也可用作 32 个单精度或 8 个四精度寄存器。V8 也是纯粹的大端。V9有32个双精度浮点寄存器，也可以作为16个四精度寄存器使用，但不能作为单精度寄存器使用。UltraSPARC 架构结合了 V9 和 UltraSPARC 可视化指令集扩展。

如果需要子目标支持，您应该为您的体系结构实现特定于目标的XXXSubtarget类。此类应处理命令行选项-mcpu=和-mattr=。

TableGen 使用Target.td和Sparc.td文件中的定义生成SparcGenSubtarget.inc中的代码。 在Target.td中，如下所示， 定义了SubtargetFeature接口。SubtargetFeature接口的前4个字符串参数分别是特征名称、特征设置的属性、属性的值、特征的描述。（第五个参数是隐含存在的特征列表，其默认值为空数组。）

|  |
| --- |
| class SubtargetFeature<string n, string a, string v, string d,  list<SubtargetFeature> i = []> {  string Name = n;  string Attribute = a;  string Value = v;  string Desc = d;  list<SubtargetFeature> Implies = i;  } |

在该Sparc.td文件中，SubtargetFeature用于定义以下功能。

|  |
| --- |
| def FeatureV9 : SubtargetFeature<"v9", "IsV9", "true",  "Enable SPARC-V9 instructions">;  def FeatureV8Deprecated : SubtargetFeature<"deprecated-v8",  "V8DeprecatedInsts", "true",  "Enable deprecated V8 instructions in V9 mode">;  def FeatureVIS : SubtargetFeature<"vis", "IsVIS", "true",  "Enable UltraSPARC Visual Instruction Set extensions">; |

在Sparc.td 中的其他地方，Proc类被定义，然后用于定义可能具有前面描述的特性的特定 SPARC 处理器子类型。

|  |
| --- |
| class Proc<string Name, list<SubtargetFeature> Features>  : Processor<Name, NoItineraries, Features>;  def : Proc<"generic", []>;  def : Proc<"v8", []>;  def : Proc<"supersparc", []>;  def : Proc<"sparclite", []>;  def : Proc<"f934", []>;  def : Proc<"hypersparc", []>;  def : Proc<"sparclite86x", []>;  def : Proc<"sparclet", []>;  def : Proc<"tsc701", []>;  def : Proc<"v9", [FeatureV9]>;  def : Proc<"ultrasparc", [FeatureV9, FeatureV8Deprecated]>;  def : Proc<"ultrasparc3", [FeatureV9, FeatureV8Deprecated]>;  def : Proc<"ultrasparc3-vis", [FeatureV9, FeatureV8Deprecated, FeatureVIS]>; |

从Target.td和Sparc.td文件中，生成的结果 SparcGenSubtarget.inc指定枚举值以标识功能、常量数组以表示 CPU 功能和 CPU 子类型，以及解析设置指定子目标选项的功能字符串的ParseSubtargetFeatures方法。生成的SparcGenSubtarget.inc文件应包含在SparcSubtarget.cpp。该XXXSubtarget方法的目标特定实现应遵循以下伪代码：

|  |
| --- |
| XXXSubtarget::XXXSubtarget(const Module &M, const std::string &FS) {  // Set the default features  // Determine default and user specified characteristics of the CPU  // Call ParseSubtargetFeatures(FS, CPU) to parse the features string  // Perform any additional operations} |

## JIT Support(即时支持)

目标机器的实现可选地包括一个即时 (JIT) 代码生成器，它发出机器代码和辅助结构作为二进制输出，可以直接写入内存。为此，通过执行以下步骤来实现 JIT 代码生成：

* 编写一个XXXCodeEmitter.cpp文件，其中包含将目标机器指令转换为可重定位机器代码的机器功能通道。
* 编写一个XXXJITInfo.cpp文件，为特定于目标的代码生成活动（例如发出机器代码和存根）实现 JIT 接口。
* 修改XXXTargetMachine以便它通过其getJITInfo方法提供一个TargetJITInfo对象。

有几种不同的方法来编写 JIT 支持代码。例如，TableGen 和目标描述符文件可用于创建 JIT 代码生成器，但不是强制性的。对于 Alpha 和 PowerPC 目标机器，TableGen 用于生成XXXGenCodeEmitter.inc，其中包含机器指令的二进制编码和 getBinaryCodeForInstr访问这些代码的方法。其他 JIT 实现没有。

XXXJITInfo.cpp和都XXXCodeEmitter.cpp必须包含 llvm/CodeGen/MachineCodeEmitter.h头文件，该头文件定义了MachineCodeEmitter 类， 该类包含用于将数据（字节、字、字符串等）写入输出流的多个回调函数的代码。

### 机器代码发射器

在XXXCodeEmitter.cpp中，特定于目标的Emitter类被实现为函数传递（MachineFunctionPass的子类）。runOnMachineFunction的特定目标的实现（由MachineFunctionPass 中的 runOnFunctionin调用）遍历 MachineBasicBlock，调用emitInstruction以处理每条指令并发出二进制代码。 emitInstruction主要是通过 XXXInstrInfo.h中定义的指令类型的case语句来实现的。例如，在X86CodeEmitter.cpp 中，该emitInstruction方法是围绕以下switch/case语句构建的：

|  |
| --- |
| switch (Desc->TSFlags & X86::FormMask) {  case X86II::Pseudo: // for not yet implemented instructions  ... // or pseudo-instructions  break;  case X86II::RawFrm: // for instructions with a fixed opcode value  ...  break;  case X86II::AddRegFrm: // for instructions that have one register operand  ... // added to their opcode  break;  case X86II::MRMDestReg:// for instructions that use the Mod/RM byte  ... // to specify a destination (register)  break;  case X86II::MRMDestMem:// for instructions that use the Mod/RM byte  ... // to specify a destination (memory)  break;  case X86II::MRMSrcReg: // for instructions that use the Mod/RM byte  ... // to specify a source (register)  break;  case X86II::MRMSrcMem: // for instructions that use the Mod/RM byte  ... // to specify a source (memory)  break;  case X86II::MRM0r: case X86II::MRM1r: // for instructions that operate on  case X86II::MRM2r: case X86II::MRM3r: // a REGISTER r/m operand and  case X86II::MRM4r: case X86II::MRM5r: // use the Mod/RM byte and a field  case X86II::MRM6r: case X86II::MRM7r: // to hold extended opcode data  ...  break;  case X86II::MRM0m: case X86II::MRM1m: // for instructions that operate on  case X86II::MRM2m: case X86II::MRM3m: // a MEMORY r/m operand and  case X86II::MRM4m: case X86II::MRM5m: // use the Mod/RM byte and a field  case X86II::MRM6m: case X86II::MRM7m: // to hold extended opcode data  ...  break;  case X86II::MRMInitReg: // for instructions whose source and  ... // destination are the same register  break;  } |

这些 case 语句的实现通常首先发出操作码，然后获取操作数。然后根据操作数，可以调用辅助方法来处理操作数。例如，在X86CodeEmitter.cpp 中，对于X86II::AddRegFrm这种情况，发出的第一个数据（由emitByte）是添加到寄存器操作数的操作码。然后提取表示机器操作数的对象MO1。isImmediate、isGlobalAddress、isExternalSymbol、 isConstantPoolIndex和isJumpTableIndex等辅助方法确定操作数类型。（X86CodeEmitter.cpp也有私有方法，例如emitConstant, emitGlobalAddress, emitExternalSymbolAddress, emitConstPoolAddress，emitJumpTableAddress，并将数据发送到输出流中。）

|  |
| --- |
| case X86II::AddRegFrm:  MCE.emitByte(BaseOpcode + getX86RegNum(MI.getOperand(CurOp++).getReg()));  if (CurOp != NumOps) {  const MachineOperand &MO1 = MI.getOperand(CurOp++);  unsigned Size = X86InstrInfo::sizeOfImm(Desc);  if (MO1.isImmediate())  emitConstant(MO1.getImm(), Size);  else {  unsigned rt = Is64BitMode ? X86::reloc\_pcrel\_word  : (IsPIC ? X86::reloc\_picrel\_word : X86::reloc\_absolute\_word);  if (Opcode == X86::MOV64ri)  rt = X86::reloc\_absolute\_dword; // FIXME: add X86II flag?  if (MO1.isGlobalAddress()) {  bool NeedStub = isa<Function>(MO1.getGlobal());  bool isLazy = gvNeedsLazyPtr(MO1.getGlobal());  emitGlobalAddress(MO1.getGlobal(), rt, MO1.getOffset(), 0,  NeedStub, isLazy);  } else if (MO1.isExternalSymbol())  emitExternalSymbolAddress(MO1.getSymbolName(), rt);  else if (MO1.isConstantPoolIndex())  emitConstPoolAddress(MO1.getIndex(), rt);  else if (MO1.isJumpTableIndex())  emitJumpTableAddress(MO1.getIndex(), rt);  }  }  break; |

在前面的示例中，XXXCodeEmitter.cpp使用变量rt，它是一个RelocationType可用于重定位地址的枚举（例如，具有 PIC基址偏移量的全局地址）。该目标的RelocationType枚举在简短的特定于目标的XXXRelocations.h 文件中定义。 XXXJITInfo.cpp中定义的relocate方法使用RelocationType重写引用的全局符号的地址。

例如，X86Relocations.h为 X86 地址指定以下重定位类型。在所有四种情况下，重定位的值都会添加到内存中已有的值中。对于reloc\_pcrel\_word和reloc\_picrel\_word，还有一个额外的初始调整。

|  |
| --- |
| enum RelocationType {  reloc\_pcrel\_word = 0, // add reloc value after adjusting for the PC loc  reloc\_picrel\_word = 1, // add reloc value after adjusting for the PIC base  reloc\_absolute\_word = 2, // absolute relocation; no additional adjustment  reloc\_absolute\_dword = 3 // absolute relocation; no additional adjustment  }; |

#### 目标 JIT 信息

XXXJITInfo.cpp为特定于目标的代码生成活动实现 JIT 接口，例如发射机器代码和存根。至少，XXXJITInfo的特定于目标的版本实现了以下内容：

* getLazyResolverFunction— 初始化 JIT，为目标提供一个用于编译的函数。
* emitFunctionStub— 返回具有回调函数指定地址的本机函数。
* relocate— 根据重定位类型更改引用的全局变量的地址。
* 回调函数是函数存根的包装器，当实际目标最初未知时使用。

getLazyResolverFunction通常很容易实现。它将传入参数作为全局参数JITCompilerFunction并返回将使用函数包装器的回调函数。对于 Alpha 目标（在 AlphaJITInfo.cpp中），getLazyResolverFunction实现很简单：

|  |
| --- |
| TargetJITInfo::LazyResolverFn AlphaJITInfo::getLazyResolverFunction(  JITCompilerFn F) {  JITCompilerFunction = F;  return AlphaCompilationCallback;  } |

对于 X86 目标，getLazyResolverFunction实现稍微复杂一些，因为它为具有 SSE 指令和 XMM 寄存器的处理器返回不同的回调函数。

回调函数最初保存并稍后恢复被调用者寄存器值、传入参数以及帧和返回地址。回调函数需要对寄存器或堆栈进行低级访问，因此通常使用汇编程序实现。

# LLVM 目标独立代码生成器 --- 待学习

## 简介

LLVM 目标独立代码生成器是一个框架，它提供了一套可重用组件，用于将 LLVM 内部表示转换为指定目标的机器代码——汇编形式（适用于静态编译器）或二进制机器代码格式（可用于 JIT 编译器）。LLVM 目标独立代码生成器由六个主要组件组成：

* 1.[抽象目标描述接口](https://llvm.org/docs/CodeGenerator.html" \l "abstract-target-description)，它捕获关于机器各个方面的重要属性，独立于它们将如何使用。这些接口定义在include/llvm/Target/.
* 2.用于表示为[目标生成的代码](https://llvm.org/docs/CodeGenerator.html" \l "code-being-generated)的类。这些类旨在足够抽象以表示 任何目标机器的机器代码。这些类在 include/llvm/CodeGen/中定义。在此级别，明确公开了“常量池条目”和“跳转表”等概念。
* 3.用于表示目标文件级别代码的类和算法，即[MC 层](https://llvm.org/docs/CodeGenerator.html" \l "mc-layer)。这些类代表汇编级结构，如标签、部分和指令。在此级别，不存在“常量池条目”和“跳转表”等概念。
* 4.用于实现本机代码生成的各个阶段（寄存器分配、调度、堆栈帧表示等）的[目标独立的算法](https://llvm.org/docs/CodeGenerator.html" \l "target-independent-algorithms)。此代码位于lib/CodeGen/。
* 5.特定目标的[抽象目标描述接口的实现](https://llvm.org/docs/CodeGenerator.html" \l "implementations-of-the-abstract-target-description-interfaces)。这些机器描述利用了 LLVM 提供的组件，并且可以选择提供自定义的特定于目标的传递，为特定目标构建完整的代码生成器。目标描述位于lib/Target/。
* 6.与目标无关的 JIT 组件。LLVM JIT 完全独立于目标（它使用TargetJITInfo结构来处理特定于目标的问题。独立于目标的 JIT 的代码位于 lib/ExecutionEngine/JIT。

根据您对代码生成器的哪个部分感兴趣，其中的不同部分将对您有用。无论如何，您应该熟悉[目标描述](https://llvm.org/docs/CodeGenerator.html" \l "target-description)和[机器代码表示](https://llvm.org/docs/CodeGenerator.html" \l "machine-code-representation) 类。如果要为新目标添加后端，则需要为新目标 实现[目标描述类](https://llvm.org/docs/CodeGenerator.html" \l "implement-the-target-description)并了解[LLVM 代码表示](https://llvm.org/docs/LangRef.html)。如果你有兴趣实现一个新的[代码生成算法](https://llvm.org/docs/CodeGenerator.html" \l "code-generation-algorithm)，它应该只依赖于目标描述和机器代码表示类，确保它是可移植的。

### 代码生成器中的必需组件

LLVM 代码生成器的两个部分是代码生成器的高级接口和可用于构建目标特定后端的可重用组件集。两个最重要的接口（ [TargetMachine](https://llvm.org/docs/CodeGenerator.html" \l "targetmachine)和 [DataLayout](https://llvm.org/docs/CodeGenerator.html" \l "datalayout)) 是唯一需要为后端定义以适应 LLVM 系统的组件，但如果要使用可重用代码生成器组件，则必须定义其他组件。

这种设计有两个重要的意义。首先， LLVM 可以支持完全非传统的代码生成目标。例如，C 后端不需要寄存器分配、指令选择或系统提供的任何其他标准组件。因此，它只实现这两个接口，并做自己的事情。请注意，自 LLVM 3.1 发布以来，C 后端已从主干中删除。像这样的代码生成器的另一个示例是（纯假设的）后端，它将 LLVM 转换为 GCC RTL 形式并使用 GCC 为目标发出机器代码。

这种设计还意味着可以在 LLVM 系统中设计和实现完全不同的代码生成器，这些代码生成器不使用任何内置组件。根本不建议这样做，但对于不适合 LLVM 机器描述模型的截然不同的目标可能需要这样做：例如 FPGA。

### 代码生成器的高层设计

LLVM 目标独立代码生成器旨在支持基于标准寄存器的微处理器的高效和高质量代码生成。该模型中的代码生成分为以下几个阶段：

* 1.[指令选择](https://llvm.org/docs/CodeGenerator.html" \l "instruction-selection)——这个阶段决定了在目标指令集中表达输入 LLVM 代码的有效方式。该阶段为目标指令集中的程序生成初始代码，然后使用 SSA 形式的虚拟寄存器和表示由于目标约束或调用约定而需要的任何寄存器分配的物理寄存器。此步骤将 LLVM 代码转换为目标指令的 DAG。
* 2.[Scheduling and Formation](https://llvm.org/docs/CodeGenerator.html" \l "scheduling-and-formation)——这个阶段采用指令选择阶段产生的目标指令的 DAG，确定指令的顺序，然后发射指令作为使用那个顺序[的 MachineInstrs](https://llvm.org/docs/CodeGenerator.html" \l "machineinstr) 。请注意，我们在[指令选择部分](https://llvm.org/docs/CodeGenerator.html" \l "instruction-selection-section)对此进行了描述，因为它在[SelectionDAG](https://llvm.org/docs/CodeGenerator.html" \l "selectiondag)上运行。
* 3.[基于 SSA 的机器代码优化](https://llvm.org/docs/CodeGenerator.html" \l "ssa-based-machine-code-optimizations)——这个可选阶段由一系列机器代码优化组成，这些优化对指令选择器生成的 SSA 形式进行操作。模数调度(modulo-scheduling)或窥孔优化(peephole optimization)等优化在这里起作用。
* 4.[寄存器分配](https://llvm.org/docs/CodeGenerator.html" \l "register-allocation)——目标代码从 SSA 形式的无限虚拟寄存器文件转换为目标使用的具体寄存器文件。此阶段引入溢出代码并从程序中消除所有虚拟寄存器引用。
* 5.[Prolog/Epilog 代码插入](https://llvm.org/docs/CodeGenerator.html" \l "prolog-epilog-code-insertion)——一旦为函数生成了机器代码并且所需的堆栈空间量已知（used for LLVM alloca’s and spill slots），就可以为函数插入 prolog 和 epilog 代码以及“abstract stack location references”可以被消除。此阶段负责实施优化，如帧指针消除和堆栈打包。
* 6.[后期机器代码优化](https://llvm.org/docs/CodeGenerator.html" \l "late-machine-code-optimizations)——对“最终”机器代码进行操作的优化可以放在此处，例如，溢出代码调度和窥孔优化。
* 7.[Code Emission](https://llvm.org/docs/CodeGenerator.html" \l "code-emission)——最后阶段实际上以目标汇编格式或机器代码输出当前函数的代码。

代码生成器基于以下假设：指令选择器将使用最佳模式匹配选择器来创建高质量的本机指令序列。基于模式扩展和激进的迭代窥孔优化的替代代码生成器设计要慢得多。这种设计允许高效编译（对于 JIT 环境很重要）和积极优化（在离线生成代码时使用），方法是允许在任何编译步骤中使用不同复杂程度的组件。

除了这些阶段之外，目标实现还可以将任意特定于目标的传递插入到流程中。例如，X86 目标使用特殊通道来处理 80x87 浮点堆栈架构。可以根据需要使用自定义通道支持其他具有异常要求的目标。

### 使用 TableGen 进行目标描述

目标描述类需要目标体系结构的详细描述。这些目标描述往往有大量的共同信息（例如，一条add指令与一条sub 指令几乎相同）。为了允许提取出最大数量的共性，LLVM 代码生成器使用[TableGen Overview](https://llvm.org/docs/TableGen/index.html)工具来描述目标机器的大块，这允许使用特定于域和特定于目标的抽象来减少重复。

随着 LLVM 的不断发展和完善，我们计划将越来越多的目标描述移动到.td表单中。这样做给我们带来了很多好处。最重要的是它使移植 LLVM 变得更容易，因为它减少了必须编写的 C++ 代码量，以及在有人开始工作之前需要理解的代码生成器的知识面。其次，它使改变事物变得更容易。特别是，如果表和其他东西都由 tblgen 发出，我们只需要更改一个地方 ( tblgen) 即可将所有目标更新为新接口。

## 目标描述类

LLVM 目标描述类（位于include/llvm/Target 目录中）提供了独立于任何特定客户端的目标机器的抽象描述。这些类旨在捕获目标的抽象 属性（例如它具有的指令和寄存器），并且不包含任何特定的代码生成算法。

所有的目标描述类（除了[DataLayout class](https://llvm.org/docs/CodeGenerator.html" \l "datalayout)) 被设计为由具体的目标实现子类化，并实现了虚拟方法。为了获得这些实现，[TargetMachine](https://llvm.org/docs/CodeGenerator.html" \l "targetmachine)类提供应由目标实现的访问器。

### TargetMachine 类

该TargetMachine类提供虚方法，用于通过get\*Info方法（getInstrInfo、getRegisterInfo、 getFrameInfo等）访问各种目标描述类的目标特定实现。此类旨在实现各种虚拟方法的具体目标实现（例如，X86TargetMachine）的特例化。唯一需要的目标描述类是[DataLayout](DataLayout class) 类，但如果要使用代码生成器组件，则还应实现其他接口。

### DataLayout class

该类DataLayout是唯一需要的目标描述类，也是唯一不可扩展的类（不能从中派生新类）。 DataLayout指定有关目标如何为结构布置内存、各种数据类型的对齐要求、目标中指针的大小以及目标是小端还是大端的信息。

### TargetLowering class

基于SelectionDAG 的指令选择器使用该TargetLowering类主要是为了描述 LLVM 代码应如何降级为 SelectionDAG 操作。除其他事项外，此类指示：

* 用于各种ValueTypes 的初始寄存器类
* 目标机器本身支持哪些操作，
* setcc操作的返回类型，
* 用于shift数量的类型，以及
* 各种高级特征，例如将常量除法转换为乘法序列是否有利可图。

### TargetRegisterInfo class

该TargetRegisterInfo类用于描述目标的寄存器文件以及寄存器之间的任何交互。

寄存器在代码生成器中由无符号整数表示。物理寄存器（目标描述中实际存在的那些）是唯一的小数字，而虚拟寄存器一般都很大。请注意，寄存器#0保留为标志值。

处理器描述中的每个寄存器都有一个关联的 TargetRegisterDesc条目，它为寄存器提供了一个文本名称（用于汇编输出和调试转储）和一组别名（用于指示一个寄存器是否与另一个寄存器重叠）。

除了每个寄存器的描述之外，该TargetRegisterInfo类还公开了一组特定于处理器的寄存器类（TargetRegisterClass类的实例 ）。每个寄存器类都包含具有相同属性的寄存器组（例如，它们都是 32 位整数寄存器）。指令选择器创建的每个 SSA 虚拟寄存器都有一个关联的寄存器类。当寄存器分配器运行时，它用集合中的物理寄存器替换虚拟寄存器。

这些类的特定于目标的实现是从 寄存器文件的TableGen 概述描述中自动生成的。

### TargetInstrInfo class

该TargetInstrInfo类用于描述目标支持的机器指令。描述定义诸如操作码的助记符、操作数的数量、隐式寄存器使用列表和定义、指令是否具有某些与目标无关的属性（访问内存、可交换等），并持有任何特定于目标的标志。

### TargetFrameLowering class

该TargetFrameLowering类用于提供有关目标栈帧布局的信息。它保存堆栈增长的方向、每个函数入口处的已知堆栈对齐以及到局部区域的偏移量。到局部区域的偏移量是从函数入口处的堆栈指针到可以存储函数数据（局部变量、溢出位置）的第一个位置的偏移量。

### TargetSubtarget class

该TargetSubtarget类用于提供有关作为目标的特定芯片组的信息。子目标通知代码生成支持哪些指令、指令延迟和指令执行路线；例如，使用哪些处理单元、以什么顺序使用以及使用多长时间。

TargetJITInfo class

该TargetJITInfo类公开了一个抽象接口，即时代码生成器使用它来执行特定于目标的活动，例如发射存根。如果一个TargetMachine支持 JIT 代码生成，它应该通过getJITInfo方法提供这些对象之一。

## 机器码描述类

在高层，LLVM 代码被翻译成机器特定的表示形式，由[MachineFunction](https://llvm.org/docs/CodeGenerator.html" \l "machinefunction),  [MachineBasicBlock](https://llvm.org/docs/CodeGenerator.html" \l "machinebasicblock)， 和 [MachineInstr](https://llvm.org/docs/CodeGenerator.html" \l "machineinstr)实例（在 include/llvm/CodeGen 中定义）。这种表示完全与目标无关，以最抽象的形式表示指令：一个操作码和一系列操作数。此表示旨在支持机器代码的 SSA 表示，以及寄存器分配的非 SSA 形式。

### MachineInstr class

目标机器指令表示为MachineInstr 类的实例。此类是表示机器指令的一种极其抽象的方式。特别是，它只跟踪一个操作码编号和一组操作数。

操作码编号是一个简单的无符号整数，仅对特定后端有意义。目标的所有指令都应在 目标的\*InstrInfo.td文件中定义。操作码枚举值是根据此描述自动生成的。该MachineInstr类没有关于如何解释指令的任何信息（即指令的语义是什么）；为此你必须参考 [TargetInstrInfo](https://llvm.org/docs/CodeGenerator.html" \l "targetinstrinfo) 类。

机器指令的操作数可以有几种不同的类型：寄存器引用、常量整数、基本块引用等。此外，机器操作数应标记为 def 或值的使用（尽管只有寄存器允许为 defs）。

按照惯例，LLVM 代码生成器对指令操作数进行排序，以便所有寄存器定义都出现在寄存器使用之前，即使在通常以其他顺序打印的体系结构上也是如此。例如，SPARC 加法指令：“ add %i1, %i2, %i3”将“%i1”和“%i2”寄存器相加，并将结果存储到“%i3”寄存器中。在 LLVM 代码生成器中，操作数应存储为“%i3, %i1, %i2 ”：首先是目标。

将目标（定义）操作数放在操作数列表的开头有几个优点。特别地，调试printer将打印如下指令：

|  |
| --- |
| %r3 = add %i1, %i2 |

此外，如果第一个操作数是 def，则创建其唯一 def 是第一个操作数的[指令](https://llvm.org/docs/CodeGenerator.html" \l "create-instructions)会更容易。

#### 使用MachineInstrBuilder.h的函数

机器指令是使用文件BuildMI中的函数创建的include/llvm/CodeGen/MachineInstrBuilder.h。这些BuildMI 函数使构建任意机器指令变得容易。BuildMI函数的用法 如下所示：

|  |
| --- |
| // Create a 'DestReg = mov 42' (rendered in X86 assembly as 'mov DestReg, 42')  // instruction and insert it at the end of the given MachineBasicBlock.  const TargetInstrInfo &TII = ...  MachineBasicBlock &MBB = ...  DebugLoc DL;  MachineInstr \*MI = BuildMI(MBB, DL, TII.get(X86::MOV32ri), DestReg).addImm(42);  // Create the same instr, but insert it before a specified iterator point.  MachineBasicBlock::iterator MBBI = ...  BuildMI(MBB, MBBI, DL, TII.get(X86::MOV32ri), DestReg).addImm(42);  // Create a 'cmp Reg, 0' instruction, no destination reg.  MI = BuildMI(MBB, DL, TII.get(X86::CMP32ri8)).addReg(Reg).addImm(42);  // Create an 'sahf' instruction which takes no operands and stores nothing.  MI = BuildMI(MBB, DL, TII.get(X86::SAHF));  // Create a self looping branch instruction.  BuildMI(MBB, DL, TII.get(X86::JNE)).addMBB(&MBB); |

如果你需要添加一个定义操作数（除了可选的目标寄存器），你必须显式地标记它：

|  |
| --- |
| MI.addReg(Reg, RegState::Define); |

#### 固定（预分配）寄存器

代码生成器需要注意的一个重要问题是固定寄存器的存在。特别是，在指令流中经常有寄存器分配器必须将特定值安排在特定寄存器中的地方。这可能是由于指令集的限制（例如，X86 只能使用EAX/EDX 寄存器进行 32 位除法）或调用约定等外部因素造成的。在任何情况下，指令选择器都应该发射代码，在需要时将虚拟寄存器复制到物理寄存器中或从物理寄存器中复制出来。

例如，考虑这个简单的 LLVM 示例：

|  |
| --- |
| define i32 @test(i32 %X, i32 %Y) {  %Z = sdiv i32 %X, %Y  ret i32 %Z  } |

divX86 指令选择器可能会为and 生成此机器代码ret：

|  |
| --- |
| ;; Start of div  %EAX = mov %reg1024 ;; Copy X (in reg1024) into EAX  %reg1027 = sar %reg1024, 31  %EDX = mov %reg1027 ;; Sign extend X into EDX  idiv %reg1025 ;; Divide by Y (in reg1025)  %reg1026 = mov %EAX ;; Read the result (Z) out of EAX  ;; Start of ret  %EAX = mov %reg1026 ;; 32-bit return value goes in EAX  ret |

在代码生成结束时，寄存器分配器将合并寄存器并删除生成的标识移动，生成以下代码：

|  |
| --- |
| ;; X is in EAX, Y is in ECX  mov %EAX, %EDX  sar %EDX, 31  idiv %ECX  ret |

这种方法非常通用（如果它可以处理 X86 体系结构，它可以处理任何东西！）并且允许在指令选择器中隔离关于指令流的所有目标特定知识。请注意，为了生成良好的代码，物理寄存器的生命周期应该很短，并且所有物理寄存器在进入和退出基本块时（在寄存器分配之前）都被假定为死的。因此，如果您需要一个跨基本块边界的值，它必须存在于虚拟寄存器中。

#### Call-clobbered registers

一些机器指令，如调用，破坏了大量的物理寄存器。不是为所有这些添加<def,dead>操作数，而是可以使用一个MO\_RegisterMask操作数。寄存器掩码操作数包含保留寄存器的位掩码，其他所有内容都被认为被指令破坏了。

#### SSA 形式的机器码

MachineInstr最初以 SSA 形式选择，并以 SSA 形式维护，直到寄存器分配发生。在大多数情况下，这非常简单，因为 LLVM 已经采用 SSA 形式；LLVM PHI 节点成为机器代码 PHI 节点，虚拟寄存器只允许有单一定义。

寄存器分配后，机器代码不再是 SSA 形式，因为代码中没有虚拟寄存器。

### MachineBasicBlock class

该MachineBasicBlock类包含机器指令列表（[MachineInstr](https://llvm.org/docs/CodeGenerator.html" \l "machineinstr)实例）。它大致对应于指令选择器输入的LLVM代码，但可以存在一对多的映射（即一个LLVM基本块可以映射到多个机器基本块）。该MachineBasicBlock类有一个“ getBasicBlock”方法，它返回它来自的 LLVM 基本块。

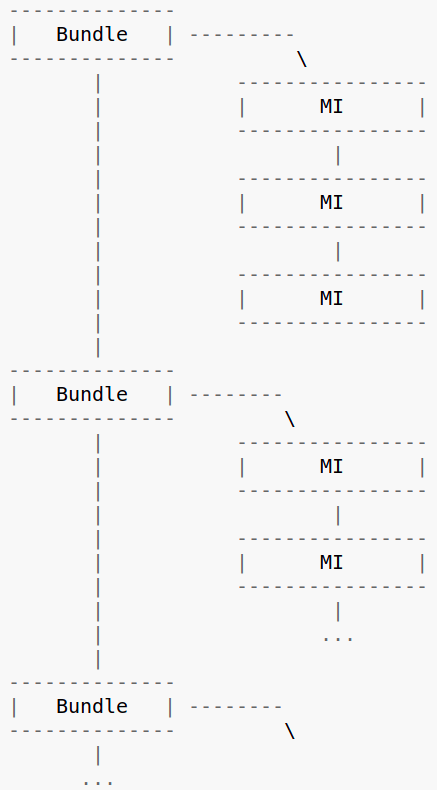
### MachineFunction class

该MachineFunction类包含机器基本块列表（ [MachineBasicBlock](https://llvm.org/docs/CodeGenerator.html" \l "machinebasicblock)实例）。它与指令选择器输入的LLVM函数一一对应。除了基本块列表之外，MachineFunction还包含一个 MachineConstantPool、 一个  MachineFrameInfo、 一个  MachineFunctionInfo和一个  MachineRegisterInfo。有关详细信息，请参阅include/llvm/CodeGen/MachineFunction.h。

### MachineInstr Bundles

LLVM 代码生成器可以将指令序列建模为 MachineInstr 包。MI bundle 可以模拟包含任意数量并行指令的 VLIW 组/包。它还可用于对不能合法分隔的顺序指令列表（可能具有数据依赖性）建模（例如 ，ARM Thumb2 IT 块）。

从概念上讲，MI 包是一个 MI，其中嵌套了许多其他 MI：



MI 包支持不会更改 MachineBasicBlock 和 MachineInstr 的物理表示。所有 MI（包括顶级和嵌套的 MI）都存储为 MI 的顺序列表。“捆绑”MI 标有“InsideBundle”标志。具有特殊 BUNDLE 操作码的顶级 MI 用于表示捆绑包的开始。将 BUNDLE MI 与不在 bundle 内也不代表 bundle 的单个 MI 混合是合法的。

MachineInstr pass 应该作为一个单元在 MI bundle 上运行。已被教导的成员方法正确处理捆绑包和捆绑包内的 MI。MachineBasicBlock 迭代器已被修改为跳过捆绑的 MI，以强制执行 bundle-as-a-single-unit的概念。另一个迭代器 instr\_iterator 已添加到 MachineBasicBlock 以允许passes 遍历 MachineBasicBlock 中的所有 MI，包括那些嵌套在 bundle 中的 MI。顶层 BUNDLE 指令必须具有正确的寄存器 MachineOperand 集，它们表示捆绑 MI 的累积输入和输出。

VLIW 架构的 MachineInstrs 的打包/捆绑通常应该作为寄存器分配超级通道的一部分来完成。更具体地说，确定哪些 MI 应该捆绑在一起的传递应该在代码生成器退出 SSA 形式之后完成（例如，在 two-address pass、PHI 消除和复制合并之后）。在将虚拟寄存器重写为物理寄存器后，应最终确定此类包（例如，添加 BUNDLE MI 和输入和输出寄存器 MachineOperands）。这消除了将虚拟寄存器操作数添加到 BUNDLE 指令的需要，这将有效地使虚拟寄存器 def 和 use 列表加倍。捆绑包可以使用虚拟寄存器并以 SSA 形式形成，但可能并不适合所有用例。

## “MC”层

MC 层用于在原始机器代码级别表示和处理代码，没有“常量池”、“跳转表”、“全局变量”等“高级”信息。在此级别，LLVM 处理诸如标签名称、机器指令和目标文件中的段之类的事情。该层中的代码用于许多重要目的：代码生成器的尾端使用它来编写 .s 或 .o 文件，它还被 llvm-mc 工具用于实现独立的机器代码汇编程序和反汇编程序。

本节介绍一些重要的类。还有一些重要的子系统在这一层进行交互，它们将在本手册的后面部分进行描述。

### MCStreamerAPI

最好将 MCStreamer 视为汇编程序 API。它是一个以不同方式实现的抽象 API （例如，输出 .s 文件、输出 ELF .o 文件等），但其 API 直接对应于您在 .s 文件中看到的内容。MCStreamer 每个指令都有一个方法，例如 EmitLabel、EmitSymbolAttribute、switchSection、emitValue（for .byte、.word）等，它们直接对应于汇编级指令。它还有一个 EmitInstruction 方法，用于将 MCInst 输出到流媒体。

这个 API 对于两个客户端最重要：llvm-mc 独立汇编器实际上是一个解析器，它解析一行，然后调用 MCStreamer 上的方法。在代码生成器中，代码生成器的[代码发射](https://llvm.org/docs/CodeGenerator.html" \l "code-emission)阶段将更高级别的 LLVM IR 和 Machine\* 构造降低到 MC 层，通过 MCStreamer 发出指令。

在MCStreamer的实现方面，主要有两种实现方式：一种是写出.s文件（MCAsmStreamer），一种是写出.o文件（MCObjectStreamer）。MCAsmStreamer 是一个简单的实现，它为每个方法（例如，EmitValue -> .byte）打印出一个指令，但 MCObjectStreamer 实现了一个完整的汇编程序。

对于特定于目标的指令，MCStreamer 有一个 MCTargetStreamer 实例。每个需要它的目标都定义了一个从它继承的类，并且很像 MCStreamer 本身：它每个指令都有一个方法和两个从它继承的类，一个目标对象streamer和一个目标 asm streamer。目标 asm streamer只是打印它 ( emitFnStart -> .fnstart)，对象streamer为它实现汇编器逻辑。

要使 llvm 使用这些类，目标初始化必须调用 TargetRegistry::RegisterAsmStreamer 和 TargetRegistry::RegisterMCObjectStreamer 传递分配相应目标streamer的回调并将其传递给 createAsmStreamer 或适当的对象streamer构造函数。

### MCContext class

MCContext 类是 MC 层各种唯一数据结构的所有者，包括符号、段等。因此，这是您与之交互以创建符号和段的类。这个类不能被子类化。

### MCSymbol class

MCSymbol 类表示程序集文件中的符号（又名标签）。有两种有趣的符号：汇编程序临时符号和普通符号。汇编器临时符号由汇编器使用和处理，但在生成目标文件时被丢弃。区别通常通过在标签上添加前缀来表示，例如“L”标签是 MachO 中的汇编器临时标签。

MCSymbols 由 MCContext 创建并在那里是唯一的。这意味着可以比较 MCSymbols 的指针等价性，以确定它们是否是相同的符号。请注意，指针不等式并不能保证标签最终会位于不同的地址。将类似这样的内容输出到 .s 文件是完全合法的：

|  |
| --- |
| foo:bar:  .byte 4 |

在这种情况下，foo 和 bar 符号将具有相同的地址。

### MCSection class

该MCSection类表示目标文件特定部分。它由目标文件特定实现（例如MCSectionMachO, MCSectionCOFF, MCSectionELF）子类化，这些由 MCContext 创建和唯一化。MCStreamer 具有当前段的概念，可以使用 SwitchToSection 方法更改（对应于 .s 文件中的“.section”指令）。

### MCInst class

MCInst类是指令的独立于目标的表示。它是一个简单的类（比MachineInstr简单得多），它包含一个特定于目标的操作码和一个 MCOperands 向量。MCOperand 又是三种情况的简单可区分联合：1) 简单立即数，2) 目标寄存器 ID，3)作为 MCExpr 的符号表达式（例如，“Lfoo-Lbar+42”）。

MCInst 是用于表示 MC 层机器指令的通用货币。它是指令编码器、指令打印机使用的类型，也是汇编解析器和反汇编器生成的类型。

### 目标文件格式

MC 层的对象编写器支持多种对象格式。由于对象格式的目标特定方面，每个目标仅支持 MC 层支持的格式的子集。大多数目标都支持发射 ELF 对象。其他特定于供应商的对象通常仅在该供应商支持的目标上受支持（即 MachO 仅在 Darwin 支持的目标上受支持，而 XCOFF 仅在支持 AIX 的目标上受支持）。此外，一些目标有自己的对象格式（即 DirectX、SPIR-V 和 WebAssembly）。

下表捕获了 LLVM 中目标文件支持的快照：

目标文件格式

|  |  |
| --- | --- |
| 格式 | 支持的目标 |
| COFF | AArch64、ARM、X86 |
| DXContainer | directX |
| ELF | AArch64、AMDGPU、ARM、AVR、BPF、CSKY、Hexagon、Lanai、LoongArch、M86k、MSP430、MIPS、PowerPC、RISCV、SPARC、SystemZ、VE、X86 |
| GCOFF | systemZ |
| MachO | AArch64、ARM、X86 |
| SPIR-V | SPIRV |
| WASM | WebAssembly |
| XCOFF | PowerPC |

## 与目标无关的代码生成算法

本节记录了[代码生成器的高级设计](https://llvm.org/docs/CodeGenerator.html" \l "high-level-design-of-the-code-generator)中描述的阶段。它解释了它们的工作原理以及设计背后的一些基本原理。

### 指令选择

指令选择是将提供给代码生成器的 LLVM 代码翻译成目标特定机器指令的过程。在文献中有几种众所周知的方法可以做到这一点。LLVM 使用基于 SelectionDAG 的指令选择器。

DAG 指令选择器的一部分是从目标描述 ( \*.td) 文件生成的。我们的目标是从这些 .td 文件中生成整个指令选择器，尽管目前仍然有一些东西需要自定义 C++ 代码。

[GlobalISel](https://llvm.org/docs/GlobalISel/index.html)是另一个指令选择框架。

#### SelectionDAG 简介

SelectionDAG 以一种适合使用自动技术（例如，基于动态规划的最佳模式匹配选择器）进行指令选择的方式为代码表示提供抽象。它也非常适合代码生成的其他阶段；特别是指令调度（SelectionDAG’s are very close to scheduling DAGs post-selection）。此外，SelectionDAG 提供了一个主机表示，可以在其中执行各种非常低级别（但与目标无关）的[优化](https://llvm.org/docs/CodeGenerator.html" \l "optimizations)；这些优化需要有关目标有效支持的指令的大量信息。

SelectionDAG 是一个有向无环图，其节点是 该类SDNode的实例。SDNode的主要有效载荷是它的操作代码（Opcode），它指示节点执行什么操作以及该操作的操作数。 include/llvm/CodeGen/ISDOpcodes.h 文件顶部描述了各种操作节点类型。

尽管大多数操作定义单个值，但图中的每个节点都可以定义多个值。例如，合并的 div/rem 操作将定义商和余数。许多其他情况也需要多个值。每个节点也有一定数量的操作数，它们是定义使用值的节点的边。因为节点可以定义多个值，所以边由SDValue类的实例表示，它是一个<SDNode, unsigned> 对，分别表示所使用的节点和结果值。由一个 SDNode 产生的每个值都有一个关联的MVT （machine value type）指示值的类型。

SelectionDAG 包含两种不同类型的值：代表数据流的值和代表控制流依赖性的值。数据值是具有整数或浮点值类型的简单边。控制边表示为MVT::Other类型的“链”的边。这些边提供具有副作用（例如，加载、存储、调用、返回等）的节点之间的排序。所有有副作用的节点都应该将令牌链作为输入并产生一个新的令牌链作为输出。按照惯例，令牌链输入始终是操作数 #0，而链结果始终是操作产生的最后一个值。然而，在指令选择之后，机器节点在指令的操作数之后有它们的链，并且可以跟随粘合节点。

SelectionDAG 指定了“Entry”和“Root”节点。Entry 节点始终是为ISD::EntryToken的 Opcode 的标记节点。根节点是令牌链中的最终副作用节点。例如，在单个基本块函数中，它将是返回节点。

SelectionDAG 的一个重要概念是“合法”与“非法”DAG 的概念。目标的合法 DAG 是仅使用支持的操作和支持的类型的 DAG。例如，在 32 位 PowerPC 上，具有 i1、i8、i16 或 i64 类型值的 DAG 将是非法的，使用 SREM 或 UREM 操作的 DAG 也是非法的。合法化类型和合法化操作阶段负责将非法 DAG 转变为合法 DAG 。

#### SelectionDAG指令选择过程

基于 SelectionDAG 的指令选择包括以下步骤：

* 1.[构建初始 DAG](https://llvm.org/docs/CodeGenerator.html" \l "build-initial-dag)——此阶段执行从输入 LLVM 代码到非法 SelectionDAG 的简单转换。
* 2.[优化 SelectionDAG](https://llvm.org/docs/CodeGenerator.html" \l "optimize-selectiondag)——此阶段对 SelectionDAG 执行简单优化以简化它，并为支持这些元操作的目标识别元指令（如旋转和div/rem对）。这使得生成的代码更加高效，并且DAG阶段（如下）中的选择指令更加简单。
* 3.[使SelectionDAG 类型合法化](https://llvm.org/docs/CodeGenerator.html" \l "legalize-selectiondag-types)——此阶段转换 SelectionDAG 节点以消除目标不支持的任何类型。
* 4.[优化 SelectionDAG](https://llvm.org/docs/CodeGenerator.html" \l "optimize-selectiondag)——运行 SelectionDAG 优化器以清除类型合法化暴露的冗余。
* 5.[使SelectionDAG 操作合法化](https://llvm.org/docs/CodeGenerator.html" \l "legalize-selectiondag-ops)——此阶段转换 SelectionDAG 节点以消除目标不支持的任何操作。
* 6.[优化 SelectionDAG](https://llvm.org/docs/CodeGenerator.html" \l "optimize-selectiondag)——运行 SelectionDAG 优化器以消除操作合法化带来的低效率。
* 7.[从 DAG 中选择指令](https://llvm.org/docs/CodeGenerator.html" \l "select-instructions-from-dag)——最后，目标指令选择器将 DAG 操作与目标指令相匹配。此过程将与目标无关的输入 DAG 转换为目标指令的另一个 DAG。
* 8. [SelectionDAG Scheduling and Formation](https://llvm.org/docs/CodeGenerator.html" \l "selectiondag-scheduling-and-formation) ——最后一个阶段将线性顺序分配给目标指令 DAG 中的指令，并将它们发送到正在编译的 MachineFunction 中。此步骤使用传统的预传递调度技术。

完成所有这些步骤后，将销毁 SelectionDAG 并运行其余的代码生成过程。

可视化这里发生的事情的一种好方法是利用一些 LLC 命令行选项。以下选项会在特定时间弹出一个显示 SelectionDAG 的窗口（如果您在使用它时仅将错误打印到控制台，您可能需要配置您的系统以添加对它的支持）。

* -view-dag-combine1-dags在构建之后、第一次优化之前显示 DAG。
* -view-legalize-dags显示合法化前的 DAG。
* -view-dag-combine2-dags在第二个优化阶段之前显示 DAG。
* -view-isel-dags在选择阶段之前显示 DAG。
* -view-sched-dags在调度之前显示 DAG。

-view-sunit-dags显示调度程序的依赖关系图。此图基于最终的 SelectionDAG，必须将必须一起调度的节点捆绑到单个调度单元节点中，并省略与调度无关的直接操作数和其他节点。

该-filter-view-dags选项允许选择您有兴趣可视化的基本块的名称，并过滤所有以前的 view-\*-dags选项。

#### 构建初始 DAG

初始 SelectionDAG 是由SelectionDAGBuilder类从LLVM输入扩展而来的简单的窥孔。此pass的目的是尽可能多地向 SelectionDAG 公开低级别的、特定于目标的详细信息。这个过程大部分是硬编码的（例如，一个 LLVM add变成一个 SDNode add ，同时，getelementptr 被扩展成明显的算术）。此 pass 需要特定于目标的挂钩来降低调用、返回、可变参数等。对于这些特性，会使用[TargetLowering](https://llvm.org/docs/CodeGenerator.html" \l "targetlowering)接口。

#### SelectionDAG 合法化类型阶段

SelectionDAG 合法化类型阶段负责将 DAG 转换为仅使用目标本机支持的类型。

将不受支持的标量类型的值转换为受支持类型的值的方法主要有两种：将小类型转换为大类型（“提升”(promoting)），以及将大整数类型分解为小整数类型（“扩展”expanding）。例如，目标可能要求将所有 f32 值提升为 f64，并将所有 i1/i8/i16 值提升为 i32。同一个目标可能需要将所有 i64 值扩展为成对的 i32 值。这些更改可以根据需要插入符号和零扩展，以确保最终代码具有与输入相同的行为。

将不受支持的向量类型的值转换为受支持的类型的值有两种主要方法：拆分向量类型，必要时多次拆分，直到找到合法类型，以及通过在末尾添加元素来扩展向量类型且将其四舍五入为合法类型（“扩大”widening）。如果一个向量一直被拆分成单元素部分而没有找到支持的向量类型，则元素将转换为标量（“标量化” scalarizing）。

目标实现通过在其TargetLowering构造函数中调用addRegisterClass方法来告诉合法化器支持哪些类型（以及为它们使用哪个注册类） 。

#### SelectionDAG 合法化阶段

SelectionDAG 合法化阶段负责将 DAG 转换为仅使用目标本机支持的操作。

目标通常有奇怪的约束，例如，不支持对每个支持的数据类型的每个操作（例如，X86 不支持字节条件移动，PowerPC 不支持从 16 位内存位置进行符号扩展加载）。Legalize 通过对另一个操作序列进行开放编码来模拟该操作（“扩展”expansion），通过将一种类型提升为支持该操作的更大类型（“提升”promotion），或通过使用特定于目标的挂钩来解决此问题实施合法化（“定制”custom）。

目标实现通过在其TargetLowering构造函数中调用setOperationAction方法来告诉合法化器不支持哪些操作（以及采取上述三种操作中的哪一种） 。

如果目标具有合法的向量类型，则期望使用这些类型为常见形式的 shufflevector IR 指令生成高效的机器代码。这可能需要对从 shufflevector IR 创建的 SelectionDAG 向量操作进行自定义合法化。应处理的 shufflevector 形式包括：

* 向量选择——向量的每个元素都是从 2 个输入向量的对应元素中选择的。此操作在目标汇编中也称为“blend”或“bitwise select”。这种类型的 shuffle 直接映射到shuffle\_vector SelectionDAG node。
* 插入子向量——一个向量被放入一个较长的向量类型中，从索引 0 开始。这种类型的shuffle 直接映射到index操作数设置为 0的insert\_subvector SelectionDAG 节点。
* 提取子向量——从索引 0 开始的较长向量类型中提取向量。这种类型的shuffle 直接映射到index 操作数设置为 0的extract\_subvector SelectionDAG 节点。
* Splat - 向量的所有元素都具有相同的标量元素。此操作在目标集合中也称为“broadcast”或“duplicate”。shufflevector IR 指令可能会改变向量长度，因此该操作可能会映射到多个 SelectionDAG 节点，包括shuffle\_vector、 concat\_vectors、insert\_subvector和extract\_subvector。

在 Legalize passes 存在之前，我们要求每个目标选择器都支持并处理每个运算符和类型，即使它们本身不受支持。Legalize 阶段的引入允许所有规范化模式在目标之间共享，并且使得优化规范化代码变得非常容易，因为它仍然是 DAG 的形式。

#### SelectionDAG 优化阶段：DAG 组合器

SelectionDAG 优化阶段运行多次以生成代码，在构建 DAG 后立即运行，在每次合法化后运行一次。该 pass 的第一次运行允许清理初始代码（例如，执行依赖于知道运算符具有受限类型输入的优化）。pass 的后续运行清理了 Legalize passes 生成的混乱代码，这使得 Legalize 变得非常简单（它可以专注于使代码合法，而不是专注于生成好的合法代码）。

执行的重要的一类优化是优化插入的符号和零扩展指令。我们目前使用临时技术，但将来可能会转向更严格的技术。这里有一些关于这个主题的好论文：

“[Widening integer arithmetic](http://www.eecs.harvard.edu/~nr/pubs/widen-abstract.html)”

Kevin Redwine and Norman Ramsey

International Conference on Compiler Construction (CC) 2004

“[Effective sign extension elimination](http://portal.acm.org/citation.cfm?doid=512529.512552)”

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani

Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation.

#### SelectionDAG 选择阶段

选择阶段是用于指令选择的大部分目标特定代码。该阶段以一个合法的 SelectionDAG 作为输入，将目标支持的指令 pattern 匹配到这个 DAG，并产生一个新的目标代码 DAG。例如，考虑以下 LLVM 片段：

|  |
| --- |
| %t1 = fadd float %W, %X  %t2 = fmul float %t1, %Y  %t3 = fadd float %t2, %Z |

此 LLVM 代码对应于基本上如下所示的 SelectionDAG：

|  |
| --- |
| (fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z) |

如果目标支持浮点乘加 (FMA) 运算，则其中一个加法可以与乘法合并。例如，在 PowerPC 上，指令选择器的输出可能看起来像这样的 DAG：

|  |
| --- |
| (FMADDS (FADDS W, X), Y, Z) |

该FMADDS指令是一个三元指令，它将前两个操作数相乘并加上第三个（作为单精度浮点数）。该FADDS指令是一个简单的二进制单精度加法指令。为执行此模式匹配，PowerPC 后端包含以下指令定义：

|  |
| --- |
| def FMADDS : AForm\_1<59, 29,  (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),  "fmadds $FRT, $FRA, $FRC, $FRB",  [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),  F4RC:$FRB))]>;  def FADDS : AForm\_2<59, 21,  (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRB),  "fadds $FRT, $FRA, $FRB",  [(set F4RC:$FRT, (fadd F4RC:$FRA, F4RC:$FRB))]>; |

指令定义的突出显示部分表示用于匹配指令的模式。DAG 运算符（如fmul/ fadd）在include/llvm/Target/TargetSelectionDAG.td文件中定义。“ F4RC”是输入值和结果值的寄存器类。

TableGen DAG 指令选择器生成器读取.td文件中的指令模式，并自动为您的目标构建部分模式匹配代码。它具有以下优势：

* 在编译器编译时，它会分析您的指令模式并告诉您您的模式是否有意义。
* 它可以处理模式匹配操作数的任意约束。特别是，可以直截了当地说“匹配任何 13 位符号扩展值的立即数”。有关示例，请参阅 PowerPC 后端中的immSExt16 和相关tblgen类。
* 它知道定义的模式的几个重要标识。例如，它知道加法是可交换的，所以它允许上面的FMADDS模式匹配“ (fadd X, (fmul Y, Z))”和“ (fadd (fmul X, Y), Z)”，而目标作者不必专门处理这种情况。





* 它有一个全功能的类型推断系统。特别是，您应该很少需要明确地告诉系统您的模式的类型部分是什么。在上面的FMADDS例子中，我们不必告诉tblgen模式中的所有节点都是“f32”类型。F4RC它能够从类型为“f32”的事实中推断和传播这些知识。
* 目标可以定义自己的（并依赖于内置的）“模式片段”。模式片段是在编译器编译期间内联到您的模式中的可重用模式块。例如，整数“ (not x)”操作实际上被定义为扩展为“ (xor x, -1)”的模式片段，因为 SelectionDAG 没有原生的“not ”操作。目标可以定义自己认为合适的速记片段。有关示例，请参阅 ' not' 和 ' ineg' 的定义。
* 除了指令之外，目标还可以使用“Pat”类指定映射到一条或多条指令的任意模式。例如，PowerPC 无法在一条指令中将任意整数立即数加载到寄存器中。为了告诉 tblgen 如何做到这一点，它定义了：

|  |
| --- |
| // Arbitrary immediate support. Implement in terms of LIS/ORI.  def : Pat<(i32 imm:$imm),  (ORI (LIS (HI16 imm:$imm)), (LO16 imm:$imm))>; |

如果没有任何用于将立即数加载到寄存器的单指令模式匹配，则将使用它。该规则表示“匹配任意 i32 立即数，将其转换为ORI（'或 16 位立即数'）和LIS （'加载 16 位立即数，其中立即数向左移动 16 位'）指令”。为实现此目的，使用LO16/HI16节点转换来操作输入立即数（在本例中，采用立即数的高 16 位或低 16 位）。

* 当使用“Pat”类将模式映射到具有一个或多个复杂操作数（例如，[X86 寻址模式](https://llvm.org/docs/CodeGenerator.html" \l "x86-addressing-mode)）的指令时，该模式可以使用ComplexPattern 将操作数指定为一个整体，或者它可以分别指定复杂的操作数组件。后者是由 PowerPC 后端完成的，例如用于预递增指令：

|  |
| --- |
| def STWU : DForm\_1<37, (outs ptr\_rc:$ea\_res), (ins GPRC:$rS, memri:$dst),  "stwu $rS, $dst", LdStStoreUpd, []>,  RegConstraint<"$dst.reg = $ea\_res">, NoEncode<"$ea\_res">;  def : Pat<(pre\_store GPRC:$rS, ptr\_rc:$ptrreg, iaddroff:$ptroff),  (STWU GPRC:$rS, iaddroff:$ptroff, ptr\_rc:$ptrreg)>; |

在这里，一对ptroff 和 ptrreg 操作数与STWU指令中memri类的复杂操作数dst相匹配。

* 虽然系统确实实现了很多自动化，但它仍然允许您编写自定义 C++ 代码来匹配特殊情况，如果有什么难以表达的话。

虽然它有很多优势，但该系统目前有一些局限性，主要是因为它是一项正在进行的工作，尚未完成：

* 总体而言，无法定义或匹配定义多个值（例如，SMUL\_LOHI、LOAD、CALL等）的 SelectionDAG 节点。这是目前您仍然需要为指令选择器编写自定义 C++ 代码的最大原因。
* 目前还没有很好的方法来支持匹配复杂的寻址模式。未来，我们将扩展模式片段以允许它们定义多个值（例如，X86 寻址模式的四个操作数，目前与自定义 C++ 代码匹配）。此外，我们将扩展片段，以便一个片段可以匹配多个不同的模式。
* 我们还不会自动推断像isStore/isLoad这样的标志。
* 我们还没有为Legalizer自动生成一组支持的寄存器和操作。
* 我们还没有绑定自定义合法节点的方法。

尽管有这些限制，指令选择器生成器对于典型指令集中的大多数二进制和逻辑运算仍然非常有用。如果您遇到任何问题或不知道如何做某事，请告诉克里斯！

#### SelectionDAG Scheduling and Formation Phase

调度阶段从选择阶段获取目标指令的 DAG 并分配一个顺序。调度程序可以根据机器的各种约束选择一个顺序（即最小寄存器压力的顺序或尝试覆盖指令延迟）。一旦建立顺序，DAG 将转换为[MachineInstrs](https://llvm.org/docs/CodeGenerator.html" \l "machineinstr) 表并且 SelectionDAG 被销毁。

请注意，此阶段在逻辑上与指令选择阶段是分开的，但在代码中与指令选择阶段紧密相关，因为它在 SelectionDAG 上运行。

#### SelectionDAG 的未来方向

1.Optional function-at-a-time selection。

2. Auto-generate entire selector from .td file。

### 基于 SSA 的机器代码优化

待写

### Live Intervals

Live Intervals 是变量处于活动状态的范围（间隔）。它们被某些[寄存器分配器](https://llvm.org/docs/CodeGenerator.html" \l "register-allocator) passes使用，以确定需要相同物理寄存器的两个或多个虚拟寄存器是否在程序中的同一点存在（即，它们冲突）。发生这种情况时，必须溢出一个虚拟寄存器。

#### Live Variable Analysis实时变量分析

确定变量存活间隔的第一步是计算指令后立即失效的寄存器组（即指令计算的输出值，但该值从未使用过）和指令使用的寄存器组, 但在指令之后永远不会使用（即，它们被杀死）。为每个虚拟寄存器和可分配物理寄存器，计算函数中实时变量信息。这是以非常有效的方式完成的，因为它使用 SSA 来稀疏地计算虚拟寄存器（采用 SSA 形式）的生命周期信息，并且只需要跟踪块内的物理寄存器。在寄存器分配之前，LLVM 可以假定物理寄存器仅存在于单个基本块中。这允许它进行单一的本地分析来解决每个基本块内的物理寄存器寿命。如果物理寄存器不可分配（例如，堆栈指针或条件代码），则不会对其进行跟踪。

物理寄存器可能存在于函数中或函数外。活在值中通常是寄存器中的参数。活Live out values are typically return values in registers。实时值被标记为这样，并在实时间隔分析期间给出虚拟的“defining”指令。如果函数的最后一个基本块是一个return，则它被标记为使用函数中的所有有效值。

PHI节点需要特殊处理，因为从函数的 CFG 的深度优先遍历计算活动变量信息不能保证PHI节点使用的虚拟寄存器在使用之前定义。当PHI遇到节点时，只处理定义，因为使用将在其他基本块中处理。

对于当前基本块的每个PHI节点，我们在当前基本块的末尾模拟一次赋值，并遍历后继基本块。如果后继基本块有一个PHI节点，并且该PHI节点的操作数之一来自当前基本块，则该变量 在当前基本块及其所有前导基本块中被标记为活动，直到具有定义指令的基本块遇到了。

#### Live Intervals Analysis实时区间分析

我们现在拥有可用于执行实时间隔分析并自行构建实时间隔的信息。我们从对基本块和机器指令进行编号开始。然后我们处理“live-in”值。它们在物理寄存器中，因此假设物理寄存器在基本块的末尾被杀死。虚拟寄存器的有效时间间隔是根据机器指令[1, N]的某些顺序计算的。一个活动区间是一个区间 [i, j)，其中1 >= i >= j > N，其中一个变量是活动的。

还有更多，后续补充……

### 注册分配

寄存器分配问题在于映射一个 程序Pv ，该Pv可以使用无限数量的虚拟寄存器，到一个程序Pp，该Pp包含有限（可能很少）数量的物理寄存器。每个目标体系结构都有不同数量的物理寄存器。如果物理寄存器的数量不足以容纳所有虚拟寄存器，则必须将其中一些映射到内存中。这些虚拟称为溢出虚拟。

#### 寄存器在 LLVM 中是如何表示的

在 LLVM 中，物理寄存器由通常介于 1 到 1023 之间的整数表示。要了解此编号是如何为特定架构定义的，您可以阅读该架构的GenRegisterNames.inc文件。例如，通过检查 lib/Target/X86/X86GenRegisterInfo.inc,我们看到32位寄存器 EAX表示为43，而MMX寄存器MM0映射为65。

一些架构包含共享相同物理位置的寄存器。一个值得注意的例子是 X86 平台。例如，在 X86 架构中，寄存器EAX,AX和AL共享前八位。这些物理寄存器在 LLVM中被标记为别名。给定一个特定的体系结构，您可以通过检查其RegisterInfo.td 文件来检查哪些寄存器被别名。此外，该MCRegAliasIterator类枚举了别名到一个寄存器的所有物理寄存器。

在 LLVM 中，物理寄存器在Register Classes中分组。同一寄存器类中的元素在功能上是等效的，可以互换使用。每个虚拟寄存器只能映射到特定类的物理寄存器。比如在X86架构中，有些virtual只能分配给8位寄存器。寄存器类由 TargetRegisterClass对象描述。要发现虚拟寄存器是否与给定的物理寄存器兼容，可以使用以下代码：

|  |
| --- |
| bool RegMapping\_Fer::compatible\_class(MachineFunction &mf,  unsigned v\_reg,  unsigned p\_reg) {  assert(TargetRegisterInfo::isPhysicalRegister(p\_reg) &&  "Target register must be physical");  const TargetRegisterClass \*trc = mf.getRegInfo().getRegClass(v\_reg);  return trc->contains(p\_reg);  } |

有时，主要是出于调试目的，更改目标体系结构中可用的物理寄存器的数量很有用。这必须在TargetRegisterInfo.td文件内静态完成。只是grep对于 RegisterClass，它的最后一个参数是一个寄存器列表。只是注释掉一些是避免它们被使用的一种简单方法。一种更礼貌的方式是明确地将某些寄存器排除在分配顺序之外。有关此示例，请参阅lib/Target/X86/X86RegisterInfo.td中的GR8寄存器类的定义。

虚拟寄存器也用整数表示。与物理寄存器相反，不同的虚拟寄存器从不共享相同的数字。物理寄存器是在TargetRegisterInfo.td文件中静态定义的，不能由应用程序开发人员创建，而虚拟寄存器则不是这样。为了创建新的虚拟寄存器，使用方法 MachineRegisterInfo::createVirtualRegister()。此方法将返回一个新的虚拟寄存器。使用一个IndexedMap<Foo, VirtReg2IndexFunctor>来保存每个虚拟寄存器的信息。如果您需要枚举所有虚拟寄存器，请使用TargetRegisterInfo::index2VirtReg()函数查找虚拟寄存器编号：

|  |
| --- |
| for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {  unsigned VirtReg = TargetRegisterInfo::index2VirtReg(i);  stuff(VirtReg);  } |

在寄存器分配之前，指令的操作数大部分是虚拟寄存器，尽管也可以使用物理寄存器。为了检查给定的机器操作数是否为寄存器，请使用布尔函数 MachineOperand::isRegister()。要获取寄存器的整数代码，请使用 MachineOperand::getReg()。指令可以定义或使用寄存器。例如，ADD reg:1026 := reg:1025 reg:1024 定义寄存器 1024，并使用寄存器 1025 和 1026。给定一个寄存器操作数，该MachineOperand::isUse()方法会通知指令是否正在使用该寄存器。该MachineOperand::isDef()方法通知是否正在定义该寄存器。

在寄存器分配预着色寄存器之前，我们将调用 LLVM 位码中存在的物理寄存器。预着色寄存器用于许多不同的情况，例如，传递函数调用的参数，以及存储特定指令的结果。有两种类型的预着色寄存器：隐式定义的和显式 定义的。显式定义的寄存器是普通操作数，可以使用MachineInstr::getOperand(int)::getReg()访问。为了检查指令隐式定义了哪些寄存器，请使用 TargetInstrInfo::get(opcode)::ImplicitDefs, 目标指令的操作码是在哪里的操作码。显式和隐式物理寄存器之间的一个重要区别是后者是为每条指令静态定义的，而前者可能会根据正在编译的程序而有所不同。例如，表示函数调用的指令将始终隐式定义或使用同一组物理寄存器。要读取指令隐式使用的寄存器，请使用 TargetInstrInfo::get(opcode)::ImplicitUses。 预着色寄存器对任何寄存器分配算法施加约束。寄存器分配器必须确保它们在仍然存在时不会被虚拟寄存器的值覆盖。

#### 将虚拟寄存器映射到物理寄存器

有两种方法可以将虚拟寄存器映射到物理寄存器（或内存槽）。第一种方式，我们称之为直接映射，是基于TargetRegisterInfo和MachineOperand类方法的使用。第二种方式，我们将称之为间接映射，依赖VirtRegMap 类来插入加载和存储，在内存中发送和获取值。

直接映射为寄存器分配器的开发者提供了更大的灵活性；但是，它更容易出错，并且需要更多的实施工作。基本上，程序员必须指定加载和存储指令应该插入到正在编译的目标函数中的什么位置，以便在内存中获取和存储值。要将物理寄存器分配给给定操作数中存在的虚拟寄存器，请使用MachineOperand::setReg(p\_reg)。 要插入存储指令，请使用TargetInstrInfo::storeRegToStackSlot(...)，要插入加载指令，请使用TargetInstrInfo::loadRegFromStackSlot。

间接映射使应用程序开发人员免于插入加载和存储指令的复杂性。为了将虚拟寄存器映射到物理寄存器，请使用VirtRegMap::assignVirt2Phys(vreg, preg)。 为了将某个虚拟寄存器映射到内存，使用VirtRegMap::assignVirt2StackSlot(vreg)。 此方法将返回vreg的值所在的堆栈槽。如果需要将另一个虚拟寄存器映射到同一个堆栈槽，请使用 VirtRegMap::assignVirt2StackSlot(vreg, stack\_location)。 使用间接映射时要考虑的重要一点是，即使将虚拟寄存器映射到内存，它仍然需要映射到物理寄存器。该物理寄存器是在存储之前或重新加载之后应该找到虚拟寄存器的位置。

如果使用间接策略，当所有的虚拟寄存器都映射到物理寄存器或栈槽后，就需要使用spiller对象在代码中放置load和store指令。每一个被映射到栈槽的虚拟寄存器，在定义后都会被存入内存，在使用前会被加载。spiller 的实现尝试回收加载/存储指令，避免不必要的指令。有关如何调用溢出器的示例，请参见 lib/CodeGen/RegAllocLinearScan.cpp 文件中的RegAllocLinearScan::runOnMachineFunction。

#### 处理两条地址指令

除了非常罕见的例外（例如，函数调用），LLVM 机器代码指令是三个地址指令。也就是说，每条指令最多定义一个寄存器，最多使用两个寄存器。但是，某些体系结构使用两个地址指令。在这种情况下，定义的寄存器也是使用的寄存器之一。例如，X86 中的ADD %EAX, %EBX这样的指令实际上等同于%EAX = %EAX + %EBX。

为了生成正确的代码，LLVM 必须将代表两个地址指令的三个地址指令转换为真正的两个地址指令。LLVM为这个特定目的提供了pass TwoAddressInstructionPass。它必须在寄存器分配发生之前运行。执行后，生成的代码可能不再是 SSA 形式。例如，在将一条指令%a = ADD %b %c转换为两条指令的情况下，例如：

|  |
| --- |
| %a = MOVE %b  %a = ADD %a %c |

请注意，在内部，第二条指令表示为ADD %a[def/use] %c，即，寄存器操作数%a同时由指令使用和定义。

#### SSA析构阶段

在寄存器分配期间发生的一个重要转换称为SSA 析构阶段。SSA 形式简化了对程序的控制流图执行的许多分析。然而，传统的指令集并不执行 PHI 指令。因此，为了生成可执行代码，编译器必须用保留其语义的其他指令替换 PHI 指令。

有许多方法可以安全地从目标代码中删除 PHI 指令。最传统的PHI析构算法是用拷贝指令代替PHI指令。这就是 LLVM 采用的策略。SSA解构算法在 lib/CodeGen/PHIElimination.cpp。为了调用此传递，标识符 PHIEliminationID必须在寄存器分配器的代码中按要求标记。

#### 指令折叠

指令折叠是在寄存器分配期间执行的优化，可删除不必要的复制指令。例如，一系列指令，例如：

|  |
| --- |
| %EBX = LOAD %mem\_address  %EAX = COPY %EBX |

可以安全地用一条指令代替：

|  |
| --- |
| %EAX = LOAD %mem\_address |

指令可以由 TargetRegisterInfo::foldMemoryOperand(...)方法折叠的。折叠指令时必须小心；折叠指令可能与原始指令完全不同。有关其使用的示例，请参见lib/CodeGen/LiveIntervalAnalysis.cpp的 LiveIntervals::addIntervalsForSpills。

#### 内置寄存器分配器

LLVM 基础架构为应用程序开发人员提供了三种不同的寄存器分配器：

* 快速——这个寄存器分配器是调试构建的默认值。它在基本块级别上分配寄存器，尝试将值保存在寄存器中并在适当时重用寄存器。
* 基本——这是一种递增的寄存器分配方法。活动范围按启发式驱动的顺序一次一个地分配给寄存器。由于可以在分配期间即时重写代码，因此该框架允许作为扩展进行有趣的分配器开发。它本身不是寄存器分配器生产者，而是一种潜在有用的独立模式，可用于对错误进行分类并作为性能基准。
* 贪心——默认分配器。这是Basic分配器的一个高度优化的实现，它结合了全局活动范围分割。这个分配器努力工作以最小化溢出代码的成本。
* PBQP — 基于分区布尔二次规划 (PBQP) 的寄存器分配器。该分配器的工作原理是构造一个表示正在考虑的寄存器分配问题的 PBQP 问题，使用 PBQP 求解器解决这个问题，并将解决方案映射回寄存器分配。

可以使用命令行选项-regalloc=... 选择在llc中使用的寄存器分配器类型：

|  |
| --- |
| $ llc -regalloc=linearscan file.bc -o ln.s  $ llc -regalloc=fast file.bc -o fa.s  $ llc -regalloc=pbqp file.bc -o pbqp.s |

### Prolog/Epilog 代码插入

待写

### Compact Unwind压缩展开

抛出异常需要退出一个函数。有关如何展开给定函数的信息传统上在 DWARF 展开（又名帧）信息中表示。但该格式最初是为调试器开发的，用于回溯，每个帧描述条目 (FDE) 每个函数需要约 20-30 个字节。在运行时从函数中的地址映射到相应的 FDE 也会产生成本。另一种展开编码称为压缩展开，每个函数只需要 4 个字节。

压缩展开编码是一个 32 位值，以特定于体系结构的方式进行编码。它指定要恢复哪些寄存器，从哪里恢复，以及如何从函数中展开。当链接器创建最终链接镜像时，它会创建一个\_\_TEXT,\_\_unwind\_info 段。此部分是运行时访问任何给定函数的展开信息的一种小而快速的方法。如果我们为该函数发出压缩的展开信息，那么该压缩的展开信息将在该\_\_TEXT,\_\_unwind\_info 段中进行编码。如果我们发射 DWARF 展开信息，该\_\_TEXT,\_\_unwind\_info 段将包含最终链接镜像的\_\_TEXT,\_\_eh\_frame 段中的FDE的偏移量。

对于 X86，压缩展开编码有以下三种模式：

* 带有帧指针的函数（``EBP`` 或 ``RBP``）

EBP/RBP-based frame，在返回地址之后EBP/RBP立即被压入堆栈，然后ESP/RSP被移动到 EBP/RBP. 因此要展开，用当前 EBP/RBP值恢复ESP/RSP，然后通过弹出堆栈恢复EBP/RBP，返回是通过再次弹出堆栈到 PC 中完成的。所有需要恢复的非易失性寄存器必须已经保存从EBP-4开始到EBP-1020（RBP-8到 RBP-1020）的堆栈上的一个小范围内。偏移量（在 32 位模式下除以 4，在 64 位模式下除以 8）在位 16-23（掩码：0x00FF0000）中编码。保存的寄存器在位 0-14（掩码：0x00007FFF）中编码，下表中的五个 3 位条目：

|  |  |  |
| --- | --- | --- |
| 压缩数 | i386 寄存器 | x86-64 寄存器 |
| 1 | EBX | RBX |
| 2 | ECX | R12 |
| 3 | EDX | R13 |
| 4 | EDI | R14 |
| 5 | ESI | R15 |
| 6 | EBP | RBP |

* Frameless with a Small Constant Stack Size (``EBP`` or ``RBP`` is not used as a frame pointer)

返回时，将常量（以压缩展开编码方式编码）添加到 ESP/RSP. 然后通过将堆栈弹出到 PC 来完成返回。所有需要恢复的非易失性寄存器必须在返回地址之后立即保存在堆栈中。堆栈大小（在 32 位模式下除以 4，在 64 位模式下除以 8）在位 16-23（掩码： 0x00FF0000）中编码。32 位模式下的最大堆栈大小为 1024 字节，64 位模式下为 2048 字节。保存的寄存器数量在位 9-12（掩码：0x00001C00）中编码。位 0-9（掩码：0x000003FF）包含保存了哪些寄存器及其顺序。（有关 编码算法，请参阅lib/Target/X86FrameLowering.cpp中的encodeCompactUnwindRegistersWithoutFrame()函数 。）

* Frameless with a Large Constant Stack Size (``EBP`` or ``RBP`` is not used as a frame pointer)

这种情况类似于“Frameless with a Small Constant Stack Size”的情况，但堆栈大小太大而无法在压缩展开编码中进行编码。相反，它要求函数在其序言中包含“subl $nnnnnn, %esp”。紧凑编码在第 9-12 位（掩码：0x00001C00）中包含函数中$nnnnnn值的偏移量。

### 后期机器代码优化

待写

### 代码发射

代码生成的代码发射步骤负责从代码生成器抽象（如[MachineFunction](https://llvm.org/docs/CodeGenerator.html" \l "machinefunction)、[MachineInstr](https://llvm.org/docs/CodeGenerator.html" \l "machineinstr)等）下降到 MC 层使用的抽象（[MCInst](https://llvm.org/docs/CodeGenerator.html" \l "mcinst)、[MCStreamer](https://llvm.org/docs/CodeGenerator.html" \l "mcinst)等）。这是通过几个不同类的组合完成的：（错误命名的）与目标无关的 AsmPrinter 类、AsmPrinter 的特定于目标的子类（例如，SparcAsmPrinter）和 TargetLoweringObjectFile 类。

由于 MC 层在目标文件的抽象级别工作，它没有函数、全局变量等概念。相反，它考虑labels, directives, and instructions。此时使用的一个关键类是 MCStreamer 类。这是一个以不同方式实现的抽象 API（例如，输出一个 .s 文件，输出一个 ELF .o 文件等），实际上是一个“汇编程序 API”。MCStreamer 每个指令一个方法，如 EmitLabel、EmitSymbolAttribute、switchSection 等，直接对应汇编级指令。

如果您有兴趣为目标实现代码生成器，则必须为您的目标实现三件重要的事情：

* 1.首先，您的目标需要一个 AsmPrinter 的子类。此类实现将 MachineFunction 转换为 MC 标签构造的一般降低过程。AsmPrinter 基类提供了许多有用的方法和例程，还允许您以一些重要的方式覆盖下降过程。如果您要实现 ELF、COFF 或 MachO 目标，您应该免费获得大部分降低，因为 TargetLoweringObjectFile 类实现了很多通用逻辑。
* 2.其次，您需要为您的目标实现指令printer。指令printer采用MCInst并将其作为文本呈现给 raw\_ostream。其中大部分是从 .td 文件自动生成的（当您在指令中指定类似“add $dst, $src1, $src2”的内容时），但您需要实施例程来打印操作数。
* 3.第三，您需要实现将MachineInstr降低为 MCInst 的代码，通常在“<target>MCInstLower.cpp”中实现。这个降低过程通常是特定于目标的，并负责将跳转表条目、常量池索引、全局变量地址等适当地转换为 MCLabels。该翻译层还负责将代码生成器使用的伪操作扩展为它们对应的实际机器指令。由此生成的 MCInsts 被送入指令printer或编码器。
* 最后，根据您的选择，您还可以实现 MCCodeEmitter 的子类，它将 MCInst 降低为机器代码字节和重定位。如果您想支持直接 .o 文件发射，或者想为您的目标实现一个汇编器，这很重要。

#### 发射函数堆栈大小信息

当 TargetLoweringObjectFile::StackSizesSection不为 null 并 TargetOptions::EmitStackSizeSection设置 (-stack-size-section) 时，将发射包含函数堆栈大小元数据的段。该段将包含一组函数符号值（指针大小）和堆栈大小（无符号 LEB128）对。堆栈大小值仅包括函数序言中分配的空间。不包括具有动态堆栈分配的函数。

### VLIW Packetizer

在超长指令字 (VLIW) 架构中，编译器负责将指令映射到架构上可用的功能单元。为此，编译器创建了一组称为packets或 bundles的指令。LLVM 中的 VLIW Packetizer 是一种独立于目标的机制，用于实现机器指令的Packetization 。

#### Mapping from instructions to functional units从指令到功能单元的映射

VLIW 目标中的指令通常可以映射到多个功能单元。在打包的过程中，编译器必须能够推理出是否可以将指令添加到数据包中。这个决定可能很复杂，因为编译器必须检查所有可能的指令到功能单元的映射。因此，为了减轻编译时的复杂性，VLIW 分包器解析目标的指令类并在编译器构建时生成表。然后可以通过提供的独立于机器的 API 查询这些表，以确定是否可以将指令容纳在数据包中。

#### 如何生成和使用打包表How the packetization tables are generated and used

Packetizer 从目标的行程中读取指令类，并创建一个确定性有限自动机 (DFA) 来表示数据包的状态。DFA 由三个主要元素组成：输入、状态和转换。生成的 DFA 的输入集表示添加到数据包中的指令。状态表示数据包中的指令可能消耗的功能单元。在 DFA 中，从一种状态到另一种状态的转换发生在向现有数据包添加指令时。如果存在功能单元到指令的合法映射，则 DFA 包含相应的转换。没有转换表明不存在合法映射并且无法将指令添加到数据包中。

要为 VLIW 目标生成表，请将Target GenDFAPacketizer.inc 作为目标添加到目标目录中的 Makefile。导出的 API 提供了三个函数：DFAPacketizer::clearResources()、 DFAPacketizer::reserveResources(MachineInstr \*MI)和 DFAPacketizer::canReserveResources(MachineInstr \*MI)。这些函数允许目标分包器向现有数据包添加指令，并检查是否可以将指令添加到数据包。有关 详细信息，请参阅llvm/CodeGen/DFAPacketizer.h。

## Implementing a Native Assembler实现本机汇编器

尽管您阅读本文可能是因为您想编写或维护编译器后端，但 LLVM 也完全支持构建本机汇编程序。我们努力尝试从 .td 文件（特别是指令语法和编码）自动生成汇编程序，这意味着可以分解大部分手动和重复的数据输入并与编译器共享。

### 指令解析

待写

### 指令别名处理

一旦指令被解析，它就进入 MatchInstructionImpl 函数。MatchInstructionImpl 函数执行别名处理，然后进行实际匹配。

别名处理是将相同指令的不同词汇形式规范化为一种表示的阶段。有几种不同类型的别名可以实现，它们按处理顺序列在下面（从最简单/最弱到最复杂/强大的顺序）。通常您希望使用第一个满足您的指令需要的别名机制，因为它将允许更简洁的描述。

#### Mnemonic Aliases助记符别名

别名处理的第一阶段是对允许使用两种不同助记符的指令类进行简单的指令助记符重新映射。这个阶段是一个简单且无条件的从一个输入助记符到一个输出助记符的重新映射。这种形式的别名根本不可能查看操作数，因此重新映射必须适用于给定助记符的所有形式。助记符别名定义简单，例如 X86 有：

|  |
| --- |
| def : MnemonicAlias<"cbw", "cbtw">;  def : MnemonicAlias<"smovq", "movsq">;  def : MnemonicAlias<"fldcww", "fldcw">;  def : MnemonicAlias<"fucompi", "fucomip">;  def : MnemonicAlias<"ud2a", "ud2">; |

……还有很多其他的。使用 MnemonicAlias 定义，可以简单直接地重新映射助记符。尽管 MnemonicAlias 无法查看指令的任何方面（例如，操作数），但它们可以通过 Requires 子句依赖于全局模式（匹配器支持的相同模式）：

|  |
| --- |
| def : MnemonicAlias<"pushf", "pushfq">, Requires<[In64BitMode]>;  def : MnemonicAlias<"pushf", "pushfl">, Requires<[In32BitMode]>; |

在此示例中，助记符根据当前指令集被映射到不同的助记符。

#### 指令别名

别名处理的最一般阶段发生在匹配发生时：它为匹配器提供新形式以匹配以及要生成的特定指令。指令别名有两部分：要匹配的字符串和要生成的指令。例如：

|  |
| --- |
| def : InstAlias<"movsx $src, $dst", (MOVSX16rr8W GR16:$dst, GR8 :$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX16rm8W GR16:$dst, i8mem:$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX32rr8 GR32:$dst, GR8 :$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX32rr16 GR32:$dst, GR16 :$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX64rr8 GR64:$dst, GR8 :$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX64rr16 GR64:$dst, GR16 :$src)>;  def : InstAlias<"movsx $src, $dst", (MOVSX64rr32 GR64:$dst, GR32 :$src)>; |

这显示了指令别名的一个强大示例，根据汇编中存在的操作数以多种不同方式匹配相同的助记符。指令别名的结果可以包括与目标指令顺序不同的操作数，并且可以多次使用输入，例如：

|  |
| --- |
| def : InstAlias<"clrb $reg", (XOR8rr GR8 :$reg, GR8 :$reg)>;  def : InstAlias<"clrw $reg", (XOR16rr GR16:$reg, GR16:$reg)>;  def : InstAlias<"clrl $reg", (XOR32rr GR32:$reg, GR32:$reg)>;  def : InstAlias<"clrq $reg", (XOR64rr GR64:$reg, GR64:$reg)>; |

此示例还表明绑定操作数仅列出一次。在 X86 后端，XOR8rr 有两个输入 GR8 和一个输出 GR8（输入与输出相连）。InstAliases 采用扁平操作数列表，没有重复的绑定操作数。指令别名的结果也可以使用立即数和固定物理寄存器，它们作为简单的立即数操作数添加到结果中，例如：

|  |
| --- |
| // Fixed Immediate operand.  def : InstAlias<"aad", (AAD8i8 10)>;  // Fixed register operand.  def : InstAlias<"fcomi", (COM\_FIr ST1)>;  // Simple alias.  def : InstAlias<"fcomi $reg", (COM\_FIr RST:$reg)>; |

指令别名也可以有一个 Requires 子句来使它们特定于子目标。

如果后端支持它，指令printer可以自动发射别名而不是什么被别名。它通常会产生更好、更易读的代码。如果打印出别名的内容更好，则将“0”作为第三个参数传递给 InstAlias 定义。

### 指令匹配

待写

## 针对特定目标的实现说明

文档的这一部分解释特定于特定目标的代码生成器的特性或设计决策。

### Tail call optimization尾调用优化

尾调用优化，被调用者重用调用者的堆栈，目前在 x86/x86-64、PowerPC、AArch64 和 WebAssembly 上得到支持。它在 x86/x86-64、PowerPC 和 AArch64 上执行，如果：

* 调用者和被调用者具有调用约定fastcc,cc 10 (GHC 调用约定), cc 11(HiPE 调用约定), tailcc或swifttailcc。
* 该调用是尾调用 - 在尾部位置（ret 紧跟在 call 之后并且 ret 使用 call 的值或者为 void）。
* 选项-tailcallopt已启用或调用约定为tailcc.
* 满足特定于平台的约束。

x86/x86-64 约束：

* 不使用可变参数列表。
* 在 x86-64 上，生成 GOT/PIC 代码时仅支持模块本地调用（可见性 = 隐藏或受保护）。

AArch64 约束：

* 不使用可变参数列表。

例子：

调用为 llc -tailcallopt test.ll

|  |
| --- |
| declare fastcc i32 @tailcallee(i32 inreg %a1, i32 inreg %a2, i32 %a3, i32 %a4)  define fastcc i32 @tailcaller(i32 %in1, i32 %in2) {  %l1 = add i32 %in1, %in2  %tmp = tail call fastcc i32 @tailcallee(i32 inreg %in1, i32 inreg %in2, i32 %in1, i32 %l1)  ret i32 %tmp  } |

-tailcallopt的影响：

为了在被调用者的参数多于调用者的情况下支持尾调用优化，使用了“被调用者弹出参数”约定。目前，这会导致每个fastcc未优化尾调用（因为未满足上述一个或多个约束）的调用之后重新调整堆栈。所以在这种情况下性能可能会更差。

### Sibling call optimization兄弟调用优化

兄弟调用优化是尾调用优化的一种限制形式。与上一节中描述的尾调用优化不同，它可以在-tailcallopt未指定选项时自动对任何尾调用执行。

当满足以下约束时，当前在 x86/x86-64 上执行同级调用优化：

* 调用者和被调用者具有相同的调用约定。它可以是c或 fastcc。
* 该调用是尾调用 - 在尾部位置（ret 紧跟在 call 之后并且 ret 使用 call 的值或者为 void）。
* 调用方和被调用方具有匹配的返回类型或未使用被调用方结果。
* 如果在堆栈中传递任何被调用方参数，则它们必须在调用方自己的传入参数堆栈中可用，并且帧偏移量必须相同。

例子：

|  |
| --- |
| declare i32 @bar(i32, i32)  define i32 @foo(i32 %a, i32 %b, i32 %c) {  entry:  %0 = tail call i32 @bar(i32 %a, i32 %b)  ret i32 %0  } |

### X86 后端

X86 代码生成器位于该lib/Target/X86目录中。此代码生成器能够针对各种 x86-32 和 x86-64 处理器，并包括对 ISA 扩展（例如 MMX 和 SSE）的支持。

#### 支持 X86 目标三元组

以下是 X86 后端支持的已知目标三元组。这不是一个详尽的列表，添加那些人们测试过的内容会很有用。

i686-pc-linux-gnu — Linux

i386-unknown-freebsd5.3 — FreeBSD 5.3

i686-pc-cygwin — Win32 上的 Cygwin

i686-pc-mingw32 — Win32 上的 MingW

i386-pc-mingw32msvc — Linux 上的 MingW 交叉编译器

i686-apple-darwin\* — X86 上的 Apple Darwin

x86\_64-unknown-linux-gnu — Linux

#### 支持 X86 调用约定

后端已知以下特定于目标的调用约定：

x86\_StdCall — 在 Microsoft Windows 平台上看到的 stdcall 调用约定 (CC ID = 64)。

x86\_FastCall — 在 Microsoft Windows 平台上看到的 fastcall 调用约定 (CC ID = 65)。

x86\_ThisCall — 类似于 X86\_StdCall。在 ECX 中传递第一个参数，其他参数通过堆栈传递。Callee 负责堆栈清理。MSVC 默认将此约定用于其 ABI (CC ID = 70) 中的方法。

#### 在 MachineInstrs 中表示 X86 寻址模式

x86 具有非常灵活的内存访问方式。它能够在整数指令（使用 ModR/M 寻址）中直接形成以下表达式的内存地址：

|  |
| --- |
| SegmentReg: Base + [1,2,4,8] \* IndexReg + Disp32 |

为了表示这一点，LLVM 为这种形式的每个内存操作数跟踪不少于 5 个操作数。这意味着 ' mov' 的“加载”形式按以下MachineOperand顺序包含：

Index: 0 | 1 2 3 4 5

Meaning: DestReg, | BaseReg, Scale, IndexReg, Displacement Segment

OperandTy: VirtReg, | VirtReg, UnsImm, VirtReg, SignExtImm PhysReg

存储和所有其他指令以相同的方式和相同的顺序处理四个内存操作数。如果未指定段寄存器 (regno = 0)，则不会生成段覆盖。“Lea”操作没有指定段寄存器，因此它们只有 4 个操作数用于内存引用。

#### 支持 X86 地址空间

x86 有一个特性，它提供了通过 x86 段寄存器执行加载和存储到不同地址空间的能力。指令上的段覆盖前缀字节导致指令的内存访问转到指定的段。LLVM 地址空间 0 是默认地址空间，其中包括堆栈和程序中任何不合格的内存访问。地址空间 1-255 当前保留给用户定义的代码。GS 段由地址空间 256 表示，FS 段由地址空间 257 表示，SS 段由地址空间 258 表示。其他 x86 段尚未分配地址空间编号。

虽然这些地址空间可能通过关键字看起来类似于 TLS thread\_local ，并且通常使用相同的底层硬件，但还是存在一些根本差异。

thread\_local关键字适用于全局变量并指定它们将在线程本地内存中分配。不涉及类型限定符，这些变量可以用普通指针指向并用普通加载和存储访问。该thread\_local关键字在 LLVM IR 级别与目标无关（尽管 LLVM 还没有针对某些配置实现它）

相反，特殊地址空间适用于静态类型。每个加载和存储在其地址操作数类型中都有一个特定的地址空间，这决定了访问哪个地址空间。LLVM 忽略全局变量上的这些特殊地址空间限定符，并且不提供直接在其中分配存储空间的方法。在 LLVM IR 级别，这些特殊地址空间的行为部分取决于底层操作系统或运行时环境，并且它们特定于 x86（并且 LLVM 在某些情况下尚未正确处理它们）。

一些操作系统和运行时环境使用（或将来可能使用）FS/GS 段寄存器用于各种低级目的，因此在考虑它们时应小心。

#### 指令命名

指令名称由基本名称、默认操作数大小和每个操作数的字符组成，具有可选的特殊大小。例如：

|  |
| --- |
| ADD8rr -> add, 8-bit register, 8-bit register  IMUL16rmi -> imul, 16-bit register, 16-bit memory, 16-bit immediate  IMUL16rmi8 -> imul, 16-bit register, 16-bit memory, 8-bit immediate  MOVSX32rm16 -> movsx, 32-bit register, 16-bit memory |

### AMDGPU 后端

AMDGPU 代码生成器位于该lib/Target/AMDGPU 目录中。此代码生成器能够针对各种 AMD GPU 处理器。有关详细信息，请参阅AMDGPU 后端用户指南。

# TableGen

## 简介

TableGen 的目的是帮助人类开发和维护特定领域信息的记录。因为这些记录的数量可能很多，所以专门设计来允许编写灵活的描述并提取这些记录的共同特征。这减少了描述中的重复量，减少了出错的机会，并且更容易构建特定领域的信息。

TableGen 前端解析文件，实例化声明，并将结果交给特定领域的[后端](https://llvm.org/docs/TableGen/index.html" \l "backend)进行处理。有关 TableGen的深入描述，请参阅[TableGen 程序员参考](https://llvm.org/docs/TableGen/ProgRef.html)。有关运行各种 TableGen的\*-tblgen命令的详细信息，请参见[tblgen - C++ 代码说明](https://llvm.org/docs/CommandGuide/tblgen.html)。

目前 TableGen 的主要用户是 [LLVM Target-Independent Code Generator](https://llvm.org/docs/CodeGenerator.html)和 [Clang diagnostics and attributes](https://clang.llvm.org/docs/UsersManual.html" \l "controlling-errors-and-warnings)。

请注意，如果您经常使用 TableGen 并使用 emacs 或 vim，您可以在 LLVM 发行版的llvm/utils/emacs和llvm/utils/vim目录中分别找到一个 emacs“TableGen mode”和一个 vim language file。

## TableGen 程序

TableGen 文件由 TableGen 程序解释：llvm-tblgen在您的构建目录下的bin下可用。它没有安装在系统中（或者你的 sysroot 设置的位置），因为它在 LLVM 的构建过程之外没有用处。

### 运行TableGen

TableGen 就像任何其他 LLVM 工具一样运行。第一个（可选）参数指定要读取的文件。如果未指定文件名，llvm-tblgen 则从标准输入读取。

为了有用，必须使用[后端](https://llvm.org/docs/TableGen/index.html" \l "backends)之一。这些后端可在命令行上选择（键入“llvm-tblgen -help ”以获得列表）。例如，要获取子类化特定类型的所有定义的列表（这对于构建这些记录的枚举列表很有用），请使用-print-enums 选项：

|  |
| --- |
| $ llvm-tblgen X86.td -print-enums -class=Register  AH, AL, AX, BH, BL, BP, BPL, BX, CH, CL, CX, DH, DI, DIL, DL, DX, EAX, EBP, EBX,  ECX, EDI, EDX, EFLAGS, EIP, ESI, ESP, FP0, FP1, FP2, FP3, FP4, FP5, FP6, IP,  MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7, R10, R10B, R10D, R10W, R11, R11B, R11D,  R11W, R12, R12B, R12D, R12W, R13, R13B, R13D, R13W, R14, R14B, R14D, R14W, R15,  R15B, R15D, R15W, R8, R8B, R8D, R8W, R9, R9B, R9D, R9W, RAX, RBP, RBX, RCX, RDI,  RDX, RIP, RSI, RSP, SI, SIL, SP, SPL, ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7,  XMM0, XMM1, XMM10, XMM11, XMM12, XMM13, XMM14, XMM15, XMM2, XMM3, XMM4, XMM5,  XMM6, XMM7, XMM8, XMM9,  $ llvm-tblgen X86.td -print-enums -class=Instruction  ABS\_F, ABS\_Fp32, ABS\_Fp64, ABS\_Fp80, ADC32mi, ADC32mi8, ADC32mr, ADC32ri,  ADC32ri8, ADC32rm, ADC32rr, ADC64mi32, ADC64mi8, ADC64mr, ADC64ri32, ADC64ri8,  ADC64rm, ADC64rr, ADD16mi, ADD16mi8, ADD16mr, ADD16ri, ADD16ri8, ADD16rm,  ADD16rr, ADD32mi, ADD32mi8, ADD32mr, ADD32ri, ADD32ri8, ADD32rm, ADD32rr,  ADD64mi32, ADD64mi8, ADD64mr, ADD64ri32, ... |

默认后端打印出所有记录。还有一个通用后端，它将所有记录输出为 JSON 数据结构，使用-dump-json选项启用。

如果您计划使用 TableGen，您很可能必须编写一个[后端](https://llvm.org/docs/TableGen/index.html" \l "backend) 来提取特定于您需要的信息并以适当的方式对其进行格式化。您可以通过在 C++ 中扩展 TableGen 本身，或者通过使用任何语言编写脚本，该脚本可以使用 JSON 输出。

### 例子

在没有其他参数的情况下，llvm-tblgen解析指定的文件并打印出所有类，然后是所有定义。这是查看各种定义完全扩展到什么的好方法。在X86.td文件上运行它会打印出这个（在撰写本文时）：

|  |
| --- |
| ...  def ADD32rr { // Instruction X86Inst I  string Namespace = "X86";  dag OutOperandList = (outs GR32:$dst);  dag InOperandList = (ins GR32:$src1, GR32:$src2);  string AsmString = "add{l}\t{$src2, $dst|$dst, $src2}";  list<dag> Pattern = [(set GR32:$dst, (add GR32:$src1, GR32:$src2))];  list<Register> Uses = [];  list<Register> Defs = [EFLAGS];  list<Predicate> Predicates = [];  int CodeSize = 3;  int AddedComplexity = 0;  bit isReturn = 0;  bit isBranch = 0;  bit isIndirectBranch = 0;  bit isBarrier = 0;  bit isCall = 0;  bit canFoldAsLoad = 0;  bit mayLoad = 0;  bit mayStore = 0;  bit isImplicitDef = 0;  bit isConvertibleToThreeAddress = 1;  bit isCommutable = 1;  bit isTerminator = 0;  bit isReMaterializable = 0;  bit isPredicable = 0;  bit hasDelaySlot = 0;  bit usesCustomInserter = 0;  bit hasCtrlDep = 0;  bit isNotDuplicable = 0;  bit hasSideEffects = 0;  InstrItinClass Itinerary = NoItinerary;  string Constraints = "";  string DisableEncoding = "";  bits<8> Opcode = { 0, 0, 0, 0, 0, 0, 0, 1 };  Format Form = MRMDestReg;  bits<6> FormBits = { 0, 0, 0, 0, 1, 1 };  ImmType ImmT = NoImm;  bits<3> ImmTypeBits = { 0, 0, 0 };  bit hasOpSizePrefix = 0;  bit hasAdSizePrefix = 0;  bits<4> Prefix = { 0, 0, 0, 0 };  bit hasREX\_WPrefix = 0;  FPFormat FPForm = ?;  bits<3> FPFormBits = { 0, 0, 0 };  }  ... |

这个定义对应于x86架构的32位寄存器-寄存器add指令。 “def ADD32rr”定义一个名为 ADD32rr 的记录 ，行尾的注释表示定义的超类。记录的主体包含 TableGen 为记录组装的所有数据，表明该指令是“X86”命名空间的一部分，该模式表明代码生成器如何选择指令，它是一个双地址指令，具有特定的编码等。记录中信息的内容和语义是特定于X86后端的需要的，仅作为示例显示。

如您所见，代码生成器支持的每条指令都需要大量信息，并且全部手动指定将无法维护，容易出现错误，而且一开始就很累。因为我们使用的是 TableGen，所以所有信息都来自以下定义：

|  |
| --- |
| let Defs = [EFLAGS],  isCommutable = 1, // X = ADD Y,Z --> X = ADD Z,Y  isConvertibleToThreeAddress = 1 in // Can transform into LEA.  def ADD32rr : I<0x01, MRMDestReg, (outs GR32:$dst),  (ins GR32:$src1, GR32:$src2),  "add{l}\t{$src2, $dst|$dst, $src2}",  [(set GR32:$dst, (add GR32:$src1, GR32:$src2))]>; |

该定义使用在 X86 特定的 TableGen 文件中定义的自定义类I（从自定义类X86Inst扩展）来提取其类指令共享的共同特征。TableGen 的一个关键特性是它允许最终用户定义他们在描述信息时喜欢使用的抽象。

## 语法

TableGen 的语法松散地基于 C++ 模板，具有内置类型和规范。此外，TableGen 的语法引入了一些自动化概念，如 multiclass、foreach、let 等。

### 基本概念

TableGen 文件由两个关键部分组成：“classes”和“definitions”，两者都被视为“records”。

TableGen 记录有一个唯一的名称、一个值列表和一个超类列表。值列表是TableGen为每条记录构建的主要数据；正是它保存了应用程序的域特定信息。此数据的解释留给特定的后端，但结构和格式规则由 TableGen 处理和修复。

TableGen definitions 是“records”的具体形式。这些通常没有任何未定义的值，并用 ' def' 关键字标记。

|  |
| --- |
| def FeatureFPARMv8 : SubtargetFeature<"fp-armv8", "HasFPARMv8", "true",  "Enable ARMv8 FP">; |

在此示例中，FeatureFPARMv8 是使用一些值进行初始化的SubtargetFeature 记录。类的名称是通过关键字class在同一文件或其他文件中定义的。大多数目标 TableGen 文件包括include/llvm/Target 内的通用的文件。

TableGen classes 是用于构建和描述其他记录的抽象记录。这些类允许最终用户为他们的目标域（例如 LLVM 代码生成器中的“Register”、“RegisterClass”和“Instruction”）或实现者构建抽象，以帮助分解出记录的共同属性（比如“FPInst”，在X86后端用来表示浮点指令）。TableGen 跟踪所有用于构建定义的类，因此后端可以找到特定类的所有定义，例如“instruction”。

|  |
| --- |
| class ProcNoItin<string Name, list<SubtargetFeature> Features>  : Processor<Name, NoItineraries, Features>; |

在这里，ProcNoItin class 接收字符串类型的参数Name和目标特征列表，它通过向下传递参数以及硬编码 NoItineraries 来专门化 Processor 类。

TableGen multiclasses是一组一次性实例化的抽象记录。每个实例化都可能导致多个 TableGen 定义。如果一个 multiclass 继承自另一个 multiclass ，则 sub-multiclass 中的定义将成为当前multiclass 的一部分，就好像它们是在当前 multiclass 中声明的一样。

|  |
| --- |
| multiclass ro\_signed\_pats<string T, string Rm, dag Base, dag Offset, dag Extend,  dag address, ValueType sty> {  def : Pat<(i32 (!cast<SDNode>("sextload" # sty) address)),  (!cast<Instruction>("LDRS" # T # "w\_" # Rm # "\_RegOffset")  Base, Offset, Extend)>;  def : Pat<(i64 (!cast<SDNode>("sextload" # sty) address)),  (!cast<Instruction>("LDRS" # T # "x\_" # Rm # "\_RegOffset")  Base, Offset, Extend)>;  }  defm : ro\_signed\_pats<"B", Rm, Base, Offset, Extend,  !foreach(decls.pattern, address,  !subst(SHIFT, imm\_eq0, decls.pattern)),  i8>; |

有关 TableGen的深入描述，请参阅TableGen 程序员参考。

## TableGen 后端

没有后端，TableGen 文件就没有真正的意义。运行\*-tblgen时的默认操作是以文本格式打印信息，但这仅对调试 TableGen 文件本身有用。然而，TableGen 的强大之处在于可以将源文件解释为可以生成任何你想要的内容的内部表示。

Current usage of TableGen is to create huge include files with tables that you can either include directly (if the output is in the language you’re coding), or be used in pre-processing via macros surrounding the include of the file。

如果后端已经打印了 C 格式的表，或者输出只是一个字符串列表（用于错误和警告消息），则可以使用直接输出。如果需要在不同的上下文中使用相同的信息（如指令名称），则应使用预处理输出，因此您的后端应打印一个元信息列表，该列表可以被塑造成不同的编译时格式。

有关可用后端的列表，请参阅[TableGen BackEnds](https://llvm.org/docs/TableGen/BackEnds.html) ， 有关如何编写和调试新后端的信息，请参阅[TableGen 后端开发人员指南](https://llvm.org/docs/TableGen/BackGuide.html)。

## TableGen 缺陷

尽管非常通用，但 TableGen 有一些缺陷已被多次指出。共同的主题是，虽然 TableGen 允许您构建领域特定语言，但您创建的最终语言缺乏其他 DSL 的功能，这反过来又大大增加了 TableGen 文件的大小和复杂性。

同时，TableGen 允许您通过定制的后端来创建几乎任何含义的基本概念，这会扭曲原始设计并使新手很难理解邪恶的 TableGen 文件。

有些人支持进一步扩展语义，但要确保后端遵守严格的规则。其他人建议我们应该转向为特定目的设计的更少、更强大的 DSL，甚至重用现有的 DSL。

# MCJIT 设计与实现

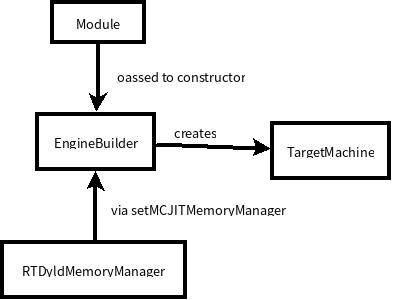
## 简介

本文档描述了 MCJIT 执行引擎和 RuntimeDyld 组件的内部工作原理。它旨在作为实现的高级概述，显示整个代码生成和动态加载过程中对象的流程和交互。

## 引擎创建

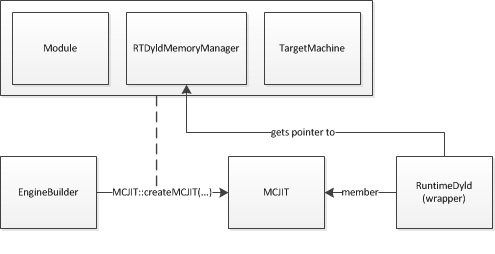
在大多数情况下，EngineBuilder 对象用于创建 MCJIT 执行引擎的实例。EngineBuilder 将 llvm::Module 对象作为其构造函数的参数。然后客户端可以设置我们控制的各种选项，这些选项稍后会传递给 MCJIT 引擎，包括选择 MCJIT 作为要创建的引擎类型。特别感兴趣的是 EngineBuilder::setMCJITMemoryManager 函数。如果此时客户端没有显式创建内存管理器，则在实例化 MCJIT 引擎时会创建一个默认的内存管理器（具体为 SectionMemoryManager）。

设置选项后，客户端调用 EngineBuilder::create 来创建 MCJIT 引擎的实例。如果客户端不使用将 TargetMachine 作为参数的函数的形式，则将基于与用于创建 EngineBuilder 的模块关联的目标三元组创建一个新的 TargetMachine。



EngineBuilder::create 将调用静态 MCJIT::createJIT 函数，将其指针传递给模块、内存管理器和目标机器对象，所有这些随后都将由 MCJIT 对象拥有。

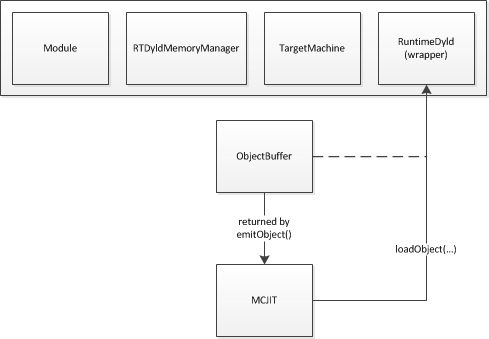
MCJIT 类有一个成员变量 Dyld，它包含 RuntimeDyld 包装类的一个实例。该成员将用于 MCJIT 与加载对象时创建的实际 RuntimeDyldImpl 对象之间的通信。



创建时，MCJIT 持有一个指向它从 EngineBuilder 接收到的模块对象的指针，但它不会立即为该模块生成代码。代码生成被推迟，直到显式调用 MCJIT::finalizeObject 方法或调用 MCJIT::getPointerToFunction 等需要生成代码的函数。

## 代码生成

当触发代码生成时，如上所述，MCJIT 将首先尝试从其 ObjectCache 成员中检索对象镜像（如果已设置）。如果无法检索缓存的对象镜像，MCJIT 将调用其 emitObject 方法。MCJIT::emitObject 使用本地 PassManager 实例并创建一个新的 ObjectBufferStream 实例，在调用创建它的模块上的 PassManager::run 之前将这两个实例传递给 TargetMachine::addPassesToEmitMC。

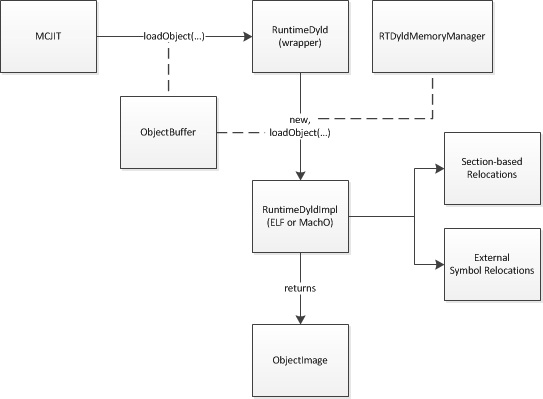


PassManager::run 调用导致 MC 代码生成机制将完整的可重定位二进制对象镜像（ELF 或 MachO 格式，具体取决于目标）发射到 ObjectBufferStream 对象中，该对象被刷新以完成该过程。如果正在使用 ObjectCache，则镜像将传递到此处的 ObjectCache。

此时，ObjectBufferStream 包含原始对象镜像。在执行代码之前，必须将此映像中的代码和数据部分加载到合适的内存中，必须应用重定位，并且必须完成内存许可和代码缓存失效（如果需要）。

## 对象加载

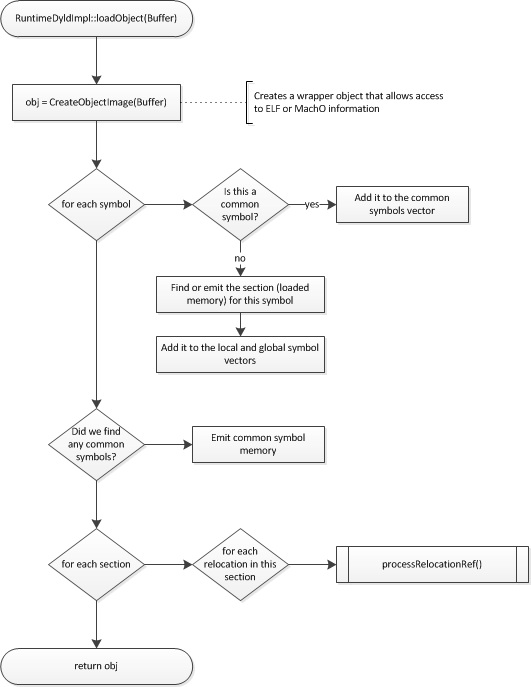
一旦通过代码生成或从 ObjectCache 检索到对象镜像，它就会被传递给 RuntimeDyld 进行加载。RuntimeDyld 包装类检查对象以确定其文件格式并创建 RuntimeDyldELF 或 RuntimeDyldMachO 的实例（两者都派生自 RuntimeDyldImpl 基类）并调用 RuntimeDyldImpl::loadObject 方法来执行实际加载。



RuntimeDyldImpl::loadObject 首先从它收到的 ObjectBuffer 创建一个 ObjectImage 实例。ObjectImage 包装了 ObjectFile 类，是一个帮助程序类，它解析二进制对象镜像并提供对格式特定标头中包含的信息的访问，包括段、符号和重定位信息。

RuntimeDyldImpl::loadObject 然后遍历镜像中的符号。收集有关常用符号的信息以备后用。对于每个函数或数据符号，关联的部分被加载到内存中，符号存储在符号表映射数据结构中。迭代完成后，会为公共符号发射一个段。

接下来，RuntimeDyldImpl::loadObject 遍历对象图像中的段，并针对每个段遍历该段的重定位。对于每个重定位，它调用特定格式的 processRelocationRef 方法，该方法将检查重定位并将其存储在两个数据结构之一中，一个是基于段的重定位列表映射，另一个是外部符号重定位映射。



当 RuntimeDyldImpl::loadObject 返回时，该对象的所有代码和数据段都已加载到内存管理器分配的内存中，并且重定位信息已准备就绪，但重定位尚未应用，生成的代码仍在还没有准备好被执行。

[目前（截至 2013 年 8 月）MCJIT 引擎将在 loadObject 完成时立即应用重定位。然而，这不应该发生。因为代码可能是为远程目标生成的，所以在应用重定位之前，应该给客户端重新映射节地址的机会。可以多次应用重定位，但在要重新映射地址的情况下，第一次应用是浪费精力。]

## 地址重映射

在生成初始代码之后和调用 finalizeObject 之前的任何时候，客户端都可以重新映射对象中各个段的地址。通常这样做是因为代码是为外部进程生成的，并且正在映射到该进程的地址空间。客户端通过调用 MCJIT::mapSectionAddress 重新映射段地址。这应该在段内存被复制到它的新位置之前发生。

当调用 MCJIT::mapSectionAddress 时，MCJIT 将调用传递给 RuntimeDyldImpl（通过其 Dyld 成员）。RuntimeDyldImpl 将新地址存储在内部数据结构中，但此时不更新代码，因为其他部分可能会更改。

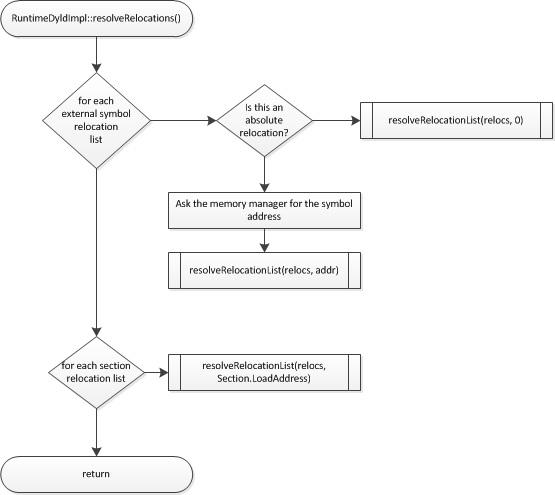
当客户端完成重映射段地址时，它将调用 MCJIT::finalizeObject 来完成重映射过程。

## 最后的准备

当调用 MCJIT::finalizeObject 时，MCJIT 调用 RuntimeDyld::resolveRelocations。此函数将尝试定位任何外部符号，然后为对象应用所有重定位。

外部符号通过调用内存管理器的 getPointerToNamedFunction 方法来解析。内存管理器将返回目标地址空间中请求符号的地址。（注意，这可能不是主机进程中的有效指针。）然后 RuntimeDyld 将遍历它存储的与此符号关联的重定位列表，并调用 resolveRelocation 方法，该方法通过特定于格式的实现将应用重定位到加载的段内存。

接下来，RuntimeDyld::resolveRelocations 遍历段列表，并为每个段遍历已保存的重定位列表，这些重定位引用该符号并为此列表中的每个条目调用 resolveRelocation。这里的重定位列表是重定位的列表，重定位关联的符号位于与列表关联的段中。这些位置中的每一个都将有一个目标位置，将在该位置应用重定位，该位置可能位于不同的段。



如上所述应用重定位后，MCJIT 调用 RuntimeDyld::getEHFrameSection，如果返回非零结果，则将段数据传递给内存管理器的 registerEHFrames 方法。这允许内存管理器调用任何所需的特定于目标的函数，例如使用调试器注册 EH 帧信息。

最后，MCJIT 调用内存管理器的 finalizeMemory 方法。在此方法中，内存管理器将在必要时使目标代码缓存无效，并将最终权限应用于它为代码和数据内存分配的内存页面。

# ORC 设计与实现

## 简介

本文档旨在提供 ORC JIT API 的设计和实现的高级概述。除非另有说明，否则所有讨论均指的是现代 ORCv2 API（自 LLVM 7 起可用）。希望从 OrcV1 过渡的客户应该看到从 [ORCv1 过渡到 ORCv2](https://llvm.org/docs/ORCv2.html" \l "transitioning-orcv1-to-orcv2) 部分。

## 用例

ORC 提供了用于构建 JIT 编译器的模块化 API。这种 API 有很多用例。例如：

* 1. LLVM 教程使用一个简单的基于 ORC 的 JIT 类来执行从玩具语言编译的表达式：Kaleidoscope。
* 2. LLVM 调试器 LLDB 使用交叉编译 JIT 进行表达式计算。在此用例中，交叉编译允许在调试器进程中编译的表达式在调试目标进程上执行，该进程可能位于不同的设备/架构上。
* 3. 在想要在现有 JIT 基础设施中使用 LLVM 优化的高性能 JIT（例如 JVM、Julia）中。
* 4.在解释器和 REPL 中，例如 Cling (C++) 和 Swift 解释器。

通过采用模块化的、基于库的设计，我们的目标是让 ORC 在尽可能多的这些环境中发挥作用。

## 特点

ORC 提供以下功能：

* JIT链接

ORC 提供 API 以在运行时将可重定位目标文件（COFF、ELF、MachO）[[1]](https://llvm.org/docs/ORCv2.html" \l "id5)链接到目标进程。目标进程可能是包含 JIT 会话对象和 jit-linker 的同一个进程，也可能是通过 RPC 与 JIT 通信的另一个进程（甚至运行在不同机器或架构上的进程）。

* LLVM IR 编译

ORC 提供现成的组件（IRCompileLayer、SimpleCompiler、ConcurrentIRCompiler），可以轻松地将 LLVM IR 添加到 JIT 进程中。

* Eager and lazy compilation，急切和懒惰的编译

默认情况下，一旦在 JIT 会话对象 ( ExecutionSession) 中查找符号，ORC 就会编译它们。默认情况下急切编译使得使用 ORC 作为现有 JIT 的内存编译器变得容易（类似于 MCJIT 的常用方式）。然而，ORC 还通过 lazy-reexports 为延迟编译提供内置支持（请参阅[Laziness](https://llvm.org/docs/ORCv2.html" \l "laziness)）。

* 支持自定义编译器和程序表示

客户可以为他们在 JIT 会话中定义的每个符号提供自定义编译器。当需要定义符号时，ORC 将运行用户提供的编译器。ORC 实际上是完全与语言无关的：LLVM IR 没有被特殊对待，并且通过 MaterializationUnit用于自定义编译器的相同包装器机制（类）得到支持。

* 并发 JIT 代码和并发编译

JIT 代码可以在多个线程中执行，可以生成新线程，并且可以从多个线程并发地重新进入 ORC（例如，请求惰性编译）。启动我的 ORC 的编译器可以并发运行（前提是客户端设置了适当的调度程序）。内置的依赖性跟踪确保 ORC 不会释放指向 JIT 代码或数据的指针，直到所有依赖性也已被 JIT 并且它们可以安全地调用或使用。

* 可移动代码

JIT 程序表示的资源

* 正交性和可组合性

以上每个功能都可以独立使用。可以将 ORC 组件放在一起以形成非惰性、进程内、单线程 JIT 或惰性、进程外、并发 JIT，或介于两者之间的任何东西。

## LLJIT 和 LLLazyJIT

ORC 提供了两个现成的基本 JIT 类。它们既可用作如何组装 ORC 组件以制作 JIT 的示例，又可用作早期 LLVM JIT API（例如 MCJIT）的替代品。

LLJIT 类使用 IRCompileLayer 和 RTDyldObjectLinkingLayer 来支持 LLVM IR 的编译和可重定位目标文件的链接。所有操作都在符号查找时急切地执行（即，一旦您尝试查找其地址，就会编译符号的定义）。在大多数情况下，LLJIT 是 MCJIT 的合适替代品（注意：尚不支持一些更高级的功能，例如 JITEventListeners）。

LLLazyJIT 扩展了 LLJIT 并添加了一个 CompileOnDemandLayer 以启用 LLVM IR 的惰性编译。当通过 addLazyIRModule 方法添加 LLVM IR 模块时，该模块中的函数体在首次调用之前不会被编译。LLLazyJIT 旨在提供 LLVM 原始（pre-MCJIT）JIT API 的替代品。

LLJIT 和 LLLazyJIT 实例可以使用它们各自的构建器类创建：LLJITBuilder 和 LLazyJITBuilder。例如，假设您在 一个ThreadSafeContext Ctx 上加载了一个模块M：

|  |
| --- |
| // Try to detect the host arch and construct an LLJIT instance.  auto JIT = LLJITBuilder().create();  // If we could not construct an instance, return an error.  if (!JIT)  return JIT.takeError();  // Add the module.  if (auto Err = JIT->addIRModule(TheadSafeModule(std::move(M), Ctx)))  return Err;  // Look up the JIT'd code entry point.  auto EntrySym = JIT->lookup("entry");  if (!EntrySym)  return EntrySym.takeError();  // Cast the entry point address to a function pointer.  auto \*Entry = (void(\*)())EntrySym.getAddress();  // Call into JIT'd code.  Entry(); |

构建器类提供了许多可以在构造 JIT 实例之前指定的配置选项。例如：

|  |
| --- |
| // Build an LLLazyJIT instance that uses four worker threads for compilation,  // and jumps to a specific error handler (rather than null) on lazy compile  // failures.  void handleLazyCompileFailure() {  // JIT'd code will jump here if lazy compilation fails, giving us an  // opportunity to exit or throw an exception into JIT'd code.  throw JITFailed();  }  auto JIT = LLLazyJITBuilder()  .setNumCompileThreads(4)  .setLazyCompileFailureAddr(  toJITTargetAddress(&handleLazyCompileFailure))  .create();  // ... |

对于想要开始使用 LLJIT 的用户，可以在llvm/examples/HowToUseLLJIT 找到一个最小的示例程序。

## 设计概述

ORC 的 JIT 程序模型旨在模拟静态和动态链接器使用的链接和符号解析规则。这允许 ORC 对任意 LLVM IR 进行 JIT，包括由使用符号链接和可见性等构造的普通静态编译器（例如 clang）生成的 IR，以及weak [[3]](https://llvm.org/docs/ORCv2.html" \l "id7)和通用符号定义。

要了解这是如何工作的，请想象一个foo链接到一对动态库的程序：libA和libB。在命令行上，构建此程序可能如下所示：

|  |
| --- |
| $ clang++ -shared -o libA.dylib a1.cpp a2.cpp  $ clang++ -shared -o libB.dylib b1.cpp b2.cpp  $ clang++ -o myapp myapp.cpp -L. -lA -lB  $ ./myapp |

在 ORC 中，这将转化为对假设的 CXXCompilingLayer 的 API 调用（为简洁起见省略了错误检查），如下所示：

|  |
| --- |
| ExecutionSession ES;  RTDyldObjectLinkingLayer ObjLinkingLayer(  ES, []() { return std::make\_unique<SectionMemoryManager>(); });  CXXCompileLayer CXXLayer(ES, ObjLinkingLayer);  // Create JITDylib "A" and add code to it using the CXX layer.  auto &LibA = ES.createJITDylib("A");  CXXLayer.add(LibA, MemoryBuffer::getFile("a1.cpp"));  CXXLayer.add(LibA, MemoryBuffer::getFile("a2.cpp"));  // Create JITDylib "B" and add code to it using the CXX layer.  auto &LibB = ES.createJITDylib("B");  CXXLayer.add(LibB, MemoryBuffer::getFile("b1.cpp"));  CXXLayer.add(LibB, MemoryBuffer::getFile("b2.cpp"));  // Create and specify the search order for the main JITDylib. This is  // equivalent to a "links against" relationship in a command-line link.  auto &MainJD = ES.createJITDylib("main");  MainJD.addToLinkOrder(&LibA);  MainJD.addToLinkOrder(&LibB);  CXXLayer.add(MainJD, MemoryBuffer::getFile("main.cpp"));  // Look up the JIT'd main, cast it to a function pointer, then call it.  auto MainSym = ExitOnErr(ES.lookup({&MainJD}, "main"));  auto \*Main = (int(\*)(int, char\*[]))MainSym.getAddress();  int Result = Main(...); |

这个例子没有告诉我们编译将如何或何时发生。这将取决于假设的 CXXCompilingLayer 的实现。但是，无论该实现如何，都将应用相同的基于链接器的符号解析规则。例如，如果 a1.cpp 和 a2.cpp 都定义了一个函数“foo”，那么 ORCv2 将产生重复定义错误。另一方面，如果 a1.cpp 和 b1.cpp 都定义了“foo”，则没有错误（不同的动态库可能定义相同的符号）。如果 main.cpp 引用“foo”，它应该绑定到 LibA 中的定义而不是 LibB 中的定义，因为 main.cpp 是“main”dylib 的一部分，并且 main dylib 在 LibB 之前链接到 LibA。

许多 JIT 客户端将不需要严格遵守通常的提前链接规则，并且应该能够通过将所有代码放在一个 JITDylib 中来顺利完成。然而，想要为传统上依赖提前链接（例如 C++）的语言/项目编写 JIT 代码的客户会发现此功能使生活变得更加轻松。

除了为符号提供地址外，ORC 中的符号查找还有两个其他重要功能：(1) 它触发对搜索到的符号的编译（如果它们尚未编译），以及 (2) 它提供同步机制并发编译。查找过程的伪代码是：

|  |
| --- |
| construct a query object from a query set and query handler  lock the session  lodge query against requested symbols, collect required materializers (if any)  unlock the session  dispatch materializers (if any) |

在这种情况下，materializer是根据请求提供符号的工作定义的东西。通常materializer只是编译器的包装器，但它们也可以直接包装一个 jit-linker（如果支持定义的程序表示是一个目标文件），或者甚至可以是一个将bits直接写入内存的类（例如，如果定义是存根）。Materialization是生成可安全调用或访问的符号定义所需的任何操作（编译、链接、展开位、向运行时注册等）的总称。

当每个materializer完成其工作时，它会通知 JITDylib，JITDylib 又会通知任何正在等待新Materialization定义的查询对象。每个查询对象都维护它仍在等待的符号数量的计数，一旦该计数达到零，查询对象就会调用查询处理程序，并使用描述结果的SymbolMap（符号名称到地址的映射）。如果任何符号未能实现查询，则立即调用查询处理程序并返回错误。

收集到的Materialization单元被发送到ExecutionSession进行调度，调度行为可以由客户端设置。默认情况下，每个materializer都在调用线程上运行。客户端可以自由创建新线程来运行materializer，或者将工作发送到线程池的工作队列（这就是 LLJIT/LLLazyJIT 所做的）。

## Top Level APIs

ORC 的许多top level API 在上面的示例中可见：

* ExecutionSession表示 JIT 程序并为 JIT 提供上下文：它包含 JITDylib、错误报告机制并分派materializer。
* JITDylibs提供符号表。
* 层（ObjLinkingLayer 和 CXXLayer）是编译器的包装器，允许客户端将这些编译器支持的未编译程序表示添加到 JITDylib。
* ResourceTrackers允许您删除代码。

明确使用了其他几个重要的 API。JIT 客户端不需要知道它们，但 Layer 作者将使用它们：

* MaterializationUnit - 当 XXXLayer::add 被调用时，它将给定的程序表示（在本例中为 C++ 源）包装在 MaterializationUnit 中，然后存储在 JITDylib 中。MaterializationUnits 负责描述它们提供的定义，并负责展开程序表示并在需要编译时将其传回层（这种所有权洗牌使得编写线程安全层更容易，因为程序表示的所有权将被传回在堆栈上，而不是必须从层成员中取出，这需要同步）。
* MaterializationResponsibility - 当 MaterializationUnit 将程序表示返回给层时，它会附带一个关联的 MaterializationResponsibility 对象。该对象跟踪必须具体化的定义，并提供一种在成功具体化或发生故障时通知 JITDylib 的方法。

## Absolute Symbols, Aliases, and Reexports

ORC 可以很容易地定义具有绝对地址的符号，或者只是其他符号的别名的符号：

### 绝对符号

绝对符号是直接映射到地址而不需要进一步具体化的符号，例如：“foo”= 0x1234。绝对符号的一个用例是允许解析进程符号。例如，

通过此映射添加到 JIT 的已建立代码可以象征性地引用 printf，而不需要“baked in”printf 的地址。这又允许在 JIT 会话中重复使用 JIT 代码的缓存版本（例如编译对象），因为 JIT 代码不再更改，只有绝对符号定义会更改。

对于进程和库符号，DynamicLibrarySearchGenerator 实用程序（请参阅[如何将进程和库符号添加到 JITDylibs](https://llvm.org/docs/ORCv2.html" \l "processandlibrarysymbols)）可用于自动为您构建绝对符号映射。但是，absoluteSymbols 函数对于使 JIT 中的非全局对象对 JIT 代码可见仍然很有用。例如，假设您的 JIT 标准库需要访问您的 JIT 对象以进行一些调用。我们可以将您的对象的地址“baked in”到库中，但随后需要为每个会话重新编译它：

我们可以把它变成 JIT 标准库中的符号引用：

然后在 JIT 启动时使用绝对符号定义使我们的 JIT 对象对 JIT 标准库可见：

### Aliases and Reexports

别名和重新导出允许您定义映射到现有符号的新符号。这对于更改跨会话的符号之间的链接关系非常有用，而无需重新编译代码。例如，假设 JIT 代码可以访问日志函数void log(const char\*)，在 JIT 标准库中有两个实现：log\_fast和 log\_detailed 。 通过在 JIT 启动时设置别名，您的 JIT 可以选择在引用 log符号时使用这些定义中的哪一个：

该symbolAliases函数允许您在单个 JITDylib 中定义别名。该reexports函数提供相同的功能，但跨 JITDylib 边界运行。

reexports 实用程序可以方便地通过从其他几个 JITDylib 重新导出符号来组成单个 JITDylib 接口。

## Laziness懒惰

ORC 中的惰性是由一个名为“lazy reexports”的实用程序提供的。惰性重新导出类似于常规重新导出或别名：它为现有符号提供新名称。然而，与常规再导出不同的是，惰性再导出的查找不会立即触发再导出符号的具体化。相反，它们只会触发函数存根的具体化。此函数存根被初始化为指向延迟调用，它提供重新进入 JIT。如果在运行时调用存根，则惰性调用将查找重新导出的符号（必要时为其触发实体化），更新存根（以在后续调用中直接调用重新导出的符号），然后通过重新导出的符号返回符号。通过重新使用现有的符号查找机制，延迟重新导出继承相同的并发保证：可以从多个线程并发调用延迟重新导出，重新导出的符号可以是任何编译状态（未编译，已经在编译过程中） ，或已经编译）并且调用将成功。这允许惰性与远程编译、并发编译、并发 JIT 代码和推测编译等功能安全地混合在一起。

一些客户必须注意的常规再导出和惰性再导出之间的另一个关键区别：惰性再导出的地址将 不同于再导出符号的地址（而常规再导出保证与再导出符号）。关心指针相等性的客户通常希望使用重新导出的地址作为重新导出符号的规范地址。这将允许在不强制实现再导出的情况下获取地址。

使用示例：

如果 JITDylibJD包含符号foo\_body和bar\_body 的定义，我们可以通过调用在 JITDylib JD2 中创建惰性入口点Foo和Bar：

|  |
| --- |
| auto ReexportFlags = JITSymbolFlags::Exported | JITSymbolFlags::Callable;  JD2.define(  lazyReexports(CallThroughMgr, StubsMgr, JD,  SymbolAliasMap({  { Mangle("foo"), { Mangle("foo\_body"), ReexportedFlags } },  { Mangle("bar"), { Mangle("bar\_body"), ReexportedFlags } }  })); |

有关如何将 lazyReexports 与 LLJIT 类一起使用的完整示例，请访问 llvm/examples/OrcV2Examples/LLJITWithLazyReexports。

## 支持自定义编译器

待定。

## 从 ORCv1 过渡到 ORCv2

自 LLVM 7.0 以来，新的 ORC 开发工作的重点是添加对并发 JIT 编译的支持。支持并发的新 API（包括新的层接口和实现，以及新的实用程序）统称为 ORCv2，而原来的非并发层和实用程序现在称为 ORCv1。

大多数 ORCv1 层和实用程序在 LLVM 8.0 中使用“Legacy”前缀重命名，并在 LLVM 9.0 中附有弃用警告。在 LLVM 12.0 中，ORCv1 将被完全删除。

对于大多数客户来说，从 ORCv1 过渡到 ORCv2 应该很容易。大多数 ORCv1 层和实用程序都有可以直接替换的 ORCv2 对应项[2] 。但是，需要注意 ORCv1 和 ORCv2 之间的一些设计差异：

* 1. ORCv2 完全采用了从 MCJIT 开始的 JIT-as-linker 模型。模块（和其他程序表示，例如目标文件）不再直接添加到 JIT 类或层。相反，它们是逐层添加到JITDylib 实例中的。JITDylib层决定了定义所在的位置，层决定了定义的编译方式。JITDylibs之间的链接关系决定了模块间引用如何解析，不再使用符号解析器。有关详细信息，请参阅[设计概述](https://llvm.org/docs/ORCv2.html" \l "design-overview)部分。

除非需要多个 JITDylib 来建模链接关系，否则 ORCv1 客户端应将所有代码放在一个 JITDylib 中。MCJIT 客户端应该使用 LLJIT（请参阅[LLJIT 和 LLLazyJIT](https://llvm.org/docs/ORCv2.html" \l "lljit-and-lllazyjit)），并且可以将代码放在 LLJIT 默认创建的主 JITDylib 中（请参阅  LLJIT::getMainJITDylib()）。

* 2.所有 JIT 堆栈现在都需要一个ExecutionSession实例。ExecutionSession 管理字符串池、错误报告、同步和符号查找。
* 3.ORCv2 使用唯一字符串（SymbolStringPtr实例）而不是字符串值，以减少内存开销并提高查找性能。请参阅[如何管理符号字符串小节](https://llvm.org/docs/ORCv2.html" \l "how-to-manage-symbol-strings)。
* 4.IR 层需要 ThreadSafeModule 实例，而不是 std::unique\_ptr<Module>s。ThreadSafeModule 是一个包装器，可确保不会同时访问使用相同 LLVMContext 的模块。请参阅[如何使用 ThreadSafeModule 和 ThreadSafeContext](https://llvm.org/docs/ORCv2.html" \l "how-to-use-threadsafemodule-and-threadsafecontext)。
* 5.符号查找不再由layers处理。相反， lookupJITDylib 上有一种方法可以获取要扫描的 JITDylib 列表。

|  |
| --- |
| ExecutionSession ES;  JITDylib &JD1 = ...;  JITDylib &JD2 = ...;  auto Sym = ES.lookup({&JD1, &JD2}, ES.intern("\_main")); |

* 6.removeModule/removeObject 方法被替换为 ResourceTracker::remove. 请参阅[如何删除代码小节](https://llvm.org/docs/ORCv2.html" \l "how-to-remove-code)。

有关如何使用 ORCv2 API 的代码示例和建议，请参阅[操作方法部分](https://llvm.org/docs/ORCv2.html" \l "how-tos)。

## 操作方法

### 如何管理符号字符串

ORC 中的符号字符串是唯一的，可以提高查找性能，减少内存开销，并允许符号名称作为有效的键。要获取SymbolStringPtr字符串值的唯一性，请调用以下 ExecutionSession::intern方法：

|  |
| --- |
| ExecutionSession ES;  /// ...  auto MainSymbolName = ES.intern("main"); |

如果您希望使用符号的 C/IR 名称执行查找，您还需要在驻留字符串之前应用平台链接器处理。在 Linux 上，这种重整(mangling)是空操作，但在其他平台上，它通常涉及向字符串添加前缀（例如 Darwin 上的 '\_'）。mangling 方案基于目标的 DataLayout。给定一个 DataLayout 和一个 ExecutionSession，您可以创建一个 MangleAndInterner 函数对象来为您执行这两项工作：

|  |
| --- |
| ExecutionSession ES;  const DataLayout &DL = ...;  MangleAndInterner Mangle(ES, DL);  // ...  // Portable IR-symbol-name lookup:  auto Sym = ES.lookup({&MainJD}, Mangle("main")); |

### 如何创建 JITDylib 并设置链接关系

在 ORC 中，所有符号定义都驻留在 JITDylibs 中。JITDylib 是通过调用ExecutionSession::createJITDylib具有唯一名称的方法创建的：

|  |
| --- |
| ExecutionSession ES;auto &JD = ES.createJITDylib("libFoo.dylib"); |

JITDylib 由ExecutionEngine实例拥有，在销毁时将被释放。

### 如何删除代码

要从 JITDylib 中删除单个模块，必须首先使用显式ResourceTracker. 然后可以通过调用删除该模块 ResourceTracker::remove：

|  |
| --- |
| auto &JD = ... ;  auto M = ... ;  auto RT = JD.createResourceTracker();  Layer.add(RT, std::move(M)); // Add M to JD, tracking resources with RT  RT.remove(); // Remove M from JD. |

直接添加到 JITDylib 的模块将由该 JITDylib 的默认资源跟踪器跟踪。

可以通过调用从 JITDylib 中删除所有代码JITDylib::clear。这使已清除的 JITDylib 处于空但可用的状态。

可以通过调用删除 JITDylibs ExecutionSession::removeJITDylib。这会清除 JITDylib，然后将其置于失效状态。不能对 JITDylib 执行进一步的操作，一旦释放最后一个句柄，它就会被销毁。

有关如何使用资源管理 API 的示例，请参见 llvm/examples/OrcV2Examples/LLJITRemovableCode。

### 如何添加对自定义程序表示的支持

为了添加对自定义程序表示的支持，需要程序表示的自定义MaterializationUnit 和自定义Layer。Layer 将有两个操作：add和emit。该add操作采用您的程序表示的一个实例，构建您的自定义之一MaterializationUnits来保存它，然后将它添加到一个JITDylib。 emit 操作获取一个MaterializationResponsibility对象和一个程序表示的实例并将其具体化，通常是通过编译它并将生成的对象传递给ObjectLinkingLayer.

您的自定义MaterializationUnit将有两个操作：materialize和discard。当查找该单元提供的任何符号时，该materialize 函数将被调用，它应该只调用layer上的emit函数，传入给定 MaterializationResponsibility的和包装的程序表示。如果不需要您的单元提供的某些弱符号（因为 JIT 找到了覆盖定义），将调用该discard函数。您可以使用它来尽早删除您的定义，或者只是忽略它并让链接器稍后删除定义。

下面是一个 ASTLayer 的例子：

|  |
| --- |
| // ... In you JIT class  AstLayer astLayer;  // ...  class AstMaterializationUnit : public orc::MaterializationUnit {  public:  AstMaterializationUnit(AstLayer &l, Ast &ast)  : llvm::orc::MaterializationUnit(l.getInterface(ast)), astLayer(l),  ast(ast) {};  llvm::StringRef getName() const override {  return "AstMaterializationUnit";  }  void materialize(std::unique\_ptr<orc::MaterializationResponsibility> r) override {  astLayer.emit(std::move(r), ast);  };  private:  void discard(const llvm::orc::JITDylib &jd, const llvm::orc::SymbolStringPtr &sym) override {  llvm\_unreachable("functions are not overridable");  }  AstLayer &astLayer;  Ast &ast;  };  class AstLayer {  llvhm::orc::IRLayer &baseLayer;  llvhm::orc::MangleAndInterner &mangler;  public:  AstLayer(llvm::orc::IRLayer &baseLayer, llvm::orc::MangleAndInterner &mangler)  : baseLayer(baseLayer), mangler(mangler){};  llvm::Error add(llvm::orc::ResourceTrackerSP &rt, Ast &ast) {  return rt->getJITDylib().define(std::make\_unique<AstMaterializationUnit>(\*this, ast), rt);  }  void emit(std::unique\_ptr<orc::MaterializationResponsibility> mr, Ast &ast) {  // compileAst is just function that compiles the given AST and returns  // a `llvm::orc::ThreadSafeModule`  baseLayer.emit(std::move(mr), compileAst(ast));  }  llvm::orc::MaterializationUnit::Interface getInterface(Ast &ast) {  SymbolFlagsMap Symbols;  // Find all the symbols in the AST and for each of them  // add it to the Symbols map.  Symbols[mangler(someNameFromAST)] =  JITSymbolFlags(JITSymbolFlags::Exported | JITSymbolFlags::Callable);  return MaterializationUnit::Interface(std::move(Symbols), nullptr);  }  }; |

查看[Building A JIT 第 4 章](https://llvm.org/docs/tutorial/BuildingAJIT4.html)的源代码以获得完整示例。

### 如何使用 ThreadSafeModule 和 ThreadSafeContext

ThreadSafeModule 和 ThreadSafeContext 分别是 Modules 和 LLVMContexts 的包装器。ThreadSafeModule 是一对 std::unique\_ptr<Module> 和（可能共享的）ThreadSafeContext 值。ThreadSafeContext 是一对 std::unique\_ptr<LLVMContext> 和一个锁。这种设计有两个目的：为 LLVMContext 提供锁定方案和生命周期管理。可以锁定 ThreadSafeContext 以防止使用相同 LLVMContext 的两个模块意外并发访问。一旦指向它的所有 ThreadSafeContext 值都被销毁，底层的 LLVMContext 就会被释放，从而允许在引用它的模块被销毁后立即回收上下文内存。

ThreadSafeContext 可以从 std::unique\_ptr<LLVMContext> 显式构造：

|  |
| --- |
| ThreadSafeContext TSCtx(std::make\_unique<LLVMContext>()); |

ThreadSafeModules 可以由一对 std::unique\_ptr<Module> 和 ThreadSafeContext 值构造。ThreadSafeContext 值可以在多个 ThreadSafeModules 之间共享：

|  |
| --- |
| ThreadSafeModule TSM1(  std::make\_unique<Module>("M1", \*TSCtx.getContext()), TSCtx);  ThreadSafeModule TSM2(  std::make\_unique<Module>("M2", \*TSCtx.getContext()), TSCtx); |

在使用 ThreadSafeContext 之前，客户端应确保上下文只能在当前线程上访问，或者上下文已锁定。TSM1在上面的示例中（其中上下文从不锁定）我们依赖于和TSM2以及 TSCtx 都是在一个线程上创建的这一事实 。如果上下文要在线程之间共享，则必须在访问或创建附加到它的任何模块之前将其锁定。例如

|  |
| --- |
| ThreadSafeContext TSCtx(std::make\_unique<LLVMContext>());  ThreadPool TP(NumThreads);JITStack J;  for (auto &ModulePath : ModulePaths) {  TP.async(  [&]() {  auto Lock = TSCtx.getLock();  auto M = loadModuleOnContext(ModulePath, TSCtx.getContext());  J.addModule(ThreadSafeModule(std::move(M), TSCtx));  });  }  TP.wait(); |

为了使对模块的独占访问更易于管理，ThreadSafeModule 类提供了一个方便的函数，withModuleDo它隐式地 (1) 锁定关联的上下文，(2) 运行给定的函数对象，(3) 解锁上下文，以及 (3) 返回函数对象生成的结果。例如

|  |
| --- |
| ThreadSafeModule TSM = getModule(...);  // Dump the module:size\_t NumFunctionsInModule =  TSM.withModuleDo(  [](Module &M) { // <- Context locked before entering lambda.  return M.size();  } // <- Context unlocked after leaving.  ); |

希望最大化并发编译可能性的客户将希望在新的 ThreadSafeContext 上创建每个新的 ThreadSafeModule。为此，提供了一个方便的 ThreadSafeModule 构造函数，它从 std::unique\_ptr<LLVMContext> 隐式构造一个新的 ThreadSafeContext 值：

|  |
| --- |
| // Maximize concurrency opportunities by loading every module on a// separate context.for (const auto &IRPath : IRPaths) {  auto Ctx = std::make\_unique<LLVMContext>();  auto M = std::make\_unique<LLVMContext>("M", \*Ctx);  CompileLayer.add(MainJD, ThreadSafeModule(std::move(M), std::move(Ctx)));  } |

计划运行单线程的客户端可以选择通过在同一上下文中加载所有模块来节省内存：

|  |
| --- |
| // Save memory by using one context for all Modules:ThreadSafeContext TSCtx(std::make\_unique<LLVMContext>());for (const auto &IRPath : IRPaths) {  ThreadSafeModule TSM(parsePath(IRPath, \*TSCtx.getContext()), TSCtx);  CompileLayer.add(MainJD, ThreadSafeModule(std::move(TSM));  } |

## 如何将进程和库符号添加到 JITDylibs

JIT 代码可能需要访问宿主程序或支持库中的符号。启用此功能的最佳方法是将这些符号反映到您的 JITDylib 中，以便它们看起来与执行会话中定义的任何其他符号相同（即它们可以通过ExecutionSession::lookup找到，因此在链接期间对 JIT 链接器可见） .

反映外部符号的一种方法是使用 absoluteSymbols 函数手动添加它们：

|  |
| --- |
| const DataLayout &DL = getDataLayout();MangleAndInterner Mangle(ES, DL);  auto &JD = ES.createJITDylib("main");  JD.define(  absoluteSymbols({  { Mangle("puts"), pointerToJITTargetAddress(&puts)},  { Mangle("gets"), pointerToJITTargetAddress(&getS)}  })); |

如果要反映的符号集很小且固定，则使用 absoluteSymbols 是合理的。另一方面，如果符号集很大或可变，则由 定义生成器按需为您添加定义可能更有意义。定义生成器是一个可以附加到 JITDylib 的对象，接收回调每当该 JITDylib 中的查找未能找到一个或多个符号时。在查找继续之前，定义生成器有机会生成缺失符号的定义。

ORC 为您提供了DynamicLibrarySearchGenerator从进程（或特定动态库）反映符号的实用程序。例如，要反映运行时库的整个接口：

|  |
| --- |
| const DataLayout &DL = getDataLayout();  auto &JD = ES.createJITDylib("main");  if (auto DLSGOrErr =  DynamicLibrarySearchGenerator::Load("/path/to/lib"  DL.getGlobalPrefix()))  JD.addGenerator(std::move(\*DLSGOrErr);  else  return DLSGOrErr.takeError();  // IR added to JD can now link against all symbols exported by the library  // at '/path/to/lib'.  CompileLayer.add(JD, loadModule(...)); |

该DynamicLibrarySearchGenerator实用程序还可以构建一个过滤功能，以限制可能反映的符号集。例如，要从主进程公开一组允许的符号：

|  |
| --- |
| const DataLayout &DL = getDataLayout();  MangleAndInterner Mangle(ES, DL);  auto &JD = ES.createJITDylib("main");  DenseSet<SymbolStringPtr> AllowList({  Mangle("puts"),  Mangle("gets")  });  // Use GetForCurrentProcess with a predicate function that checks the  // allowed list.  JD.addGenerator(cantFail(DynamicLibrarySearchGenerator::GetForCurrentProcess(  DL.getGlobalPrefix(),  [&](const SymbolStringPtr &S) { return AllowList.count(S); })));  // IR added to JD can now link against any symbols exported by the process  // and contained in the list.  CompileLayer.add(JD, loadModule(...)); |

对进程或库符号的引用也可以使用符号的原始地址硬编码到您的 IR 或目标文件中，但是应该首选使用 JIT 符号表的符号解析：它使 IR 和对象在后续的 JIT 会话中可读和可重用。硬编码地址难以阅读，通常只适用于一个会话。

## 路线图

ORC 仍在积极开发中。下面列出了一些当前和未来的工作。

### 当前工作

* 1.TargetProcessControl：改进了对进程外执行的树内支持

TargetProcessControlAPI 在 JIT 目标进程（将执行 JIT 代码的进程）上提供各种操作，包括内存分配、内存写入、函数执行和进程查询（例如，针对目标三元组）。通过以这个 API 为目标，可以开发新的组件，它们同样适用于进程内和进程外的 JITing。

* 2.基于 ORC RPC 的 TargetProcessControl 实现

目前正在开发基于 ORC RPC 的TargetProcessControlAPI 实现，以通过文件描述符/套接字实现简单的out-of-process JITing 。

* 3.核心状态机清理

核心 ORC 状态机目前在 JITDylib 和 ExecutionSession 之间实现。方法正在慢慢移至ExecutionSession。这将整理代码库，并允许我们支持异步删除 JITDylib（实际上删除 ExecutionSession 中的关联状态对象并使 JITDylib 实例处于失效状态，直到释放对它的所有引用）。

### 近期工作

* 1.ORC JIT 运行时库

我们需要一个用于 JIT 代码的运行时库。这将包括 TLS 注册、重入函数、语言运行时（例如 Objective C 和 Swift）的注册代码以及其他 JIT 特定的运行时代码。这应该以类似于 compiler-rt 的方式构建（甚至可能作为它的一部分）。

* 2.远程 jit\_dlopen / jit\_dlclose

为了更全面地模拟静态程序运行的环境，我们希望 JIT 代码能够“dlopen”和“dlclose”JITDylib，在当前线程上运行它们的所有初始化程序/取消初始化程序。这将需要上述运行时库的支持。

* 3.调试支持

当使用 RuntimeDyld 作为底层 JIT 链接器时，ORC 当前支持 GDBRegistrationListener API。我们需要为基于 JITLink 的平台提供新的解决方案。

### 进一步的未来工作

* 1.Speculative Compilation

ORC 对并发编译的支持使我们能够轻松启用 推测性JIT 编译：编译目前不需要但我们有理由相信将来会需要的代码。这可用于隐藏编译延迟并提高 JIT 吞吐量。已经开发了一个使用 ORC 进行推测编译的概念验证示例（参见 参考资料 llvm/examples/SpeculativeJIT）。未来的工作可能会集中在重用和改进现有的分析支持（目前由 PGO 使用）来提供推测决策，以及内置工具来简化推测编译的使用。

[1] 格式/架构在支持的功能方面有所不同。MachO 和 ELF 往往比 COFF 有更好的支持。非常欢迎补丁！

2.

3.

[2] LazyEmittingLayer、RemoteObjectClientLayer 和RemoteObjectServerLayer在新系统中没有对应项。在这种情况下，它根本不再需要LazyEmittingLayer：在 ORCv2 中，默认情况下推迟编译直到查找符号。删除RemoteObjectClientLayer和RemoteObjectServerLayer意味着 JIT 堆栈不能再跨进程拆分，但是似乎没有使用此功能。

[3] 弱定义目前在动态库中得到了正确处理，但是如果多个动态库提供了一个符号的弱定义，那么每个动态库最终都会有自己的定义（类似于 Windows DLL 中处理弱定义的方式）。这将在未来得到解决。

# JITLink 和 ORC 的 ObjectLinkingLayer

## 简介

本文档旨在提供 JITLink 库的设计和 API 的高级概述。它假定您熟悉链接和可重定位目标文件，但不需要深厚的专业知识。如果您知道什么是段、符号和重定位，您应该可以找到该文档。如果不是，请提交补丁（[Contributing to LLVM](https://llvm.org/docs/Contributing.html)）或提交错误（如何提交 [LLVM 错误报告](https://llvm.org/docs/HowToSubmitABug.html)）。

JITLink 是一个用于JIT 链接的库. 它是为支持 ORC JIT API 而构建的，最常通过 ORC 的 ObjectLinkingLayer API 访问。JITLink 的开发旨在支持每种对象格式提供的全套功能；包括静态初始值设定项、异常处理、线程局部变量和语言运行时注册。支持这些功能使 ORC 能够执行从依赖这些功能的源语言生成的代码（例如，C++ 需要对静态初始化器的对象格式支持以支持静态构造函数、异常的 eh-frame 注册以及对线程本地的 TLV 支持；Swift 和 Objective- C 需要为许多功能注册语言运行时）。对于某些对象格式功能的支持完全在 JITLink 中提供，而对于其他格式功能则与（原型）ORC 运行时合作提供。

JITLink 旨在支持以下功能，其中一些功能仍在开发中：

* 1.将单个可重定位对象跨进程和跨体系结构链接到目标执行程序进程。
* 2.支持所有对象格式功能。
* 3.打开链接器数据结构 ( LinkGraph) 并传递系统。

## JITLink 和 ObjectLinkingLayer

ObjectLinkingLayer是 JITLink 的 ORC 包装器。它是一个 ORC 层，允许将对象添加到JITDylib，或从某些更高级别的程序表示中发射。当发射对象时，ObjectLinkingLayer使用 JITLink 构造一个LinkGraph（请参阅构造 LinkGraphs）并调用 JITLink 的link函数将image链接到执行程序进程中。

该ObjectLinkingLayer类提供了一个插件 API， ObjectLinkingLayer::Plugin用户可以将其子类化，以便LinkGraph在链接时检查和修改实例，并对重要的 JIT 事件（例如将对象发射到目标内存中）做出反应。这启用了许多在 MCJIT 或 RuntimeDyld 下不可能实现的功能和优化。

### ObjectLinkingLayer 插件

该类ObjectLinkingLayer::Plugin提供以下方法：

* modifyPassConfig每次 LinkGraph 将被链接时调用。可以覆盖安装 JITLink Passes以在链接过程中运行。

|  |
| --- |
| void modifyPassConfig(MaterializationResponsibility &MR,  const Triple &TT,  jitlink::PassConfiguration &Config) |

* notifyLoadedMaterializationResponsibility在链接开始之前被调用，并且可以被覆盖以在需要时为给定设置任何初始状态。

|  |
| --- |
| void notifyLoaded(MaterializationResponsibility &MR) |

* notifyEmitted在链接完成并且代码已发送到执行程序进程后调用。如果需要，可以覆盖它以为MaterializationResponsibility完成状态。

|  |
| --- |
| Error notifyEmitted(MaterializationResponsibility &MR) |

* notifyFailed，如果链接在任何时候失败，则调用。它可以被覆盖以对失败做出反应（例如，取消分配任何已经分配的资源）。

|  |
| --- |
| Error notifyFailed(MaterializationResponsibility &MR) |

* notifyRemovingResources当请求为MaterializationResponsibility 删除与ResourceKey K关联的任何资源时被调用。

|  |
| --- |
| Error notifyRemovingResources(ResourceKey K) |

* notifyTransferringResources如果/当发出请求以将与ResourceKey SrcKey关联的任何资源的跟踪转移到DstKey时，将被调用。

|  |
| --- |
| void notifyTransferringResources(ResourceKey DstKey,  ResourceKey SrcKey) |

插件作者需要实现notifyFailed, notifyRemovingResources, 和notifyTransferringResources方法，以便在资源移除或转移或链接失败的情况下安全地管理资源。如果插件没有管理任何资源，那么这些方法可以实现为无操作返回Error::success()。

ObjectLinkingLayer通过调用addPlugin方法[1]将插件实例添加到。例如

|  |
| --- |
| // Plugin class to print the set of defined symbols in an object when that  // object is linked.  class MyPlugin : public ObjectLinkingLayer::Plugin {  public:  // Add passes to print the set of defined symbols after dead-stripping.  void modifyPassConfig(MaterializationResponsibility &MR,  const Triple &TT,  jitlink::PassConfiguration &Config) override {  Config.PostPrunePasses.push\_back([this](jitlink::LinkGraph &G) {  return printAllSymbols(G);  });  }  // Implement mandatory overrides:  Error notifyFailed(MaterializationResponsibility &MR) override {  return Error::success();  }  Error notifyRemovingResources(ResourceKey K) override {  return Error::success();  }  void notifyTransferringResources(ResourceKey DstKey,  ResourceKey SrcKey) override {}  // JITLink pass to print all defined symbols in G.  Error printAllSymbols(LinkGraph &G) {  for (auto \*Sym : G.defined\_symbols())  if (Sym->hasName())  dbgs() << Sym->getName() << "\n";  return Error::success();  }  };  // Create our LLJIT instance using a custom object linking layer setup.  // This gives us a chance to install our plugin.  auto J = ExitOnErr(LLJITBuilder()  .setObjectLinkingLayerCreator(  [](ExecutionSession &ES, const Triple &T) {  // Manually set up the ObjectLinkingLayer for our LLJIT  // instance.  auto OLL = std::make\_unique<ObjectLinkingLayer>(  ES, std::make\_unique<jitlink::InProcessMemoryManager>());  // Install our plugin:  OLL->addPlugin(std::make\_unique<MyPlugin>());  return OLL;  })  .create());  // Add an object to the JIT. Nothing happens here: linking isn't triggered  // until we look up some symbol in our object.  ExitOnErr(J->addObject(loadFromDisk("main.o")));  // Plugin triggers here when our lookup of main triggers linking of main.o  auto MainSym = J->lookup("main"); |

## LinkGraph

JITLink 将所有可重定位对象格式映射到一个通用LinkGraph类型，该类型旨在使链接变得快速和简单（LinkGraph也可以手动创建实例。请参阅构造 [LinkGraphs](https://llvm.org/docs/JITLink.html" \l "constructing-linkgraphs)）。

可重定位对象格式（例如 COFF、ELF、MachO）在细节上有所不同，但有一个共同的目标：用注释表示机器级代码和数据，允许它们在虚拟地址空间中重定位。为此，它们通常包含文件内部或外部定义的内容的名称（符号），必须作为一个单元移动的内容块（部分或子部分，取决于格式），以及描述如何根据某些目标符号/部分（重定位）的最终地址。

在高层次上，LinkGraph类型将这些概念表示为 decorated graph。图中的节点代表符号和内容，边代表重定位。此处列出了图中的每个元素：

* Addressable– 链接图中的一个节点，可以在执行程序进程的虚拟地址空间中分配一个地址。

绝对符号和外部符号使用普通Addressable 实例表示。目标文件中定义的内容使用 Block子类表示。

* Block–具有Content（或标记为零填充）、父级Section、a Size、Alignment（和 AlignmentOffset）以及Edge实例列表的Addressable节点。

块为二进制内容提供了一个容器，它必须在目标地址空间（布局单元）中保持连续。许多有趣的LinkGraph实例低级操作涉及检查或改变块内容或边缘。

* Content表示为llvm::StringRef，并可通过getContent方法访问。内容仅适用于内容块，而不适用于零填充块（用于isZeroFill检查，并且更喜欢 getSize只需要块大小的情况，因为它适用于零填充块和内容块）。
* Section表示为Section&引用，可通过getSection方法访问。下面Section将更详细地描述该类。
* Size表示为size\_t，并且可以通过 getSize内容块和零填充块的方法访问。
* Alignment表示为uint64\_t，并可通过 getAlignment方法获得。它表示块开头的最小对齐要求（以字节为单位）。
* AlignmentOffset表示为uint64\_t，并可通过 getAlignmentOffset方法访问。它表示块开始所需的对齐偏移量。这是支持其最小对齐要求来自块内某个非零偏移量的数据的块所必需的。例如，如果一个块由一个字节（字节对齐）后跟一个 uint64\_t（8 字节对齐）组成，那么该块将具有 8 字节对齐，对齐偏移量为 7。
* Edge实例列表。此列表的迭代器范围由该edges方法返回。下面Edge将更详细地描述该类。
* Symbol– 从 an Addressable（通常是 a Block）得到的的偏移量，带有可选Name的 、a Linkage、a Scope、Callable标志和 Live标志。

符号可以命名内容（块和可寻址是匿名的），或使用Edge.

* Name表示为一个 llvm::StringRef（如果符号没有名称则等于 llvm::StringRef()），并且可以通过该getName方法访问。
* Linkage是Strong或Weak之一，可以通过 getLinkage方法访问。可以使用JITLinkContext此标志来确定是否应保留或删除此符号定义。
* Scope是Default、Hidden或Local之一，并且可以通过该getScope方法访问。JITLinkContext可以使用它来确定谁应该能够看到该符号。具有默认范围的符号应该是全局可见的。具有隐藏范围的符号应该对同一模拟 dylib（例如 ORC JITDylib）或可执行文件中的其他定义可见，但对其他地方不可见。具有局部作用域的符号应该只在当前LinkGraph。
* Callable是一个布尔值，如果可以调用此符号，则设置为 true，并且可以通过该isCallable方法访问。这可用于自动引入调用存根以进行惰性编译。
* Live是一个布尔值，可以设置为将此符号标记为根以用于死剥离目的（请参阅[通用链接算法](https://llvm.org/docs/JITLink.html" \l "generic-link-algorithm)）。JITLink 的 dead-stripping 算法将在删除任何未标记为活动的符号（和块）之前通过graph将活动标志传播到所有可到达的符号。
* Edge– 一个四边形Offset（隐含地从包含 Block的开始）、一个Kind（描述重定位类型）、一个 Target和一个Addend。

边表示块和符号之间的重定位，偶尔也表示其他关系。

* Offset，可通过 访问getOffset，是从 Block包含Edge.
* Kind, accessible viagetKind是一种重定位类型——它描述了应该 Offset根据Target.
* Target，可通过 getTarget访问，是指向 a Symbol 的指针，表示其地址与边的 Kind指定的修正计算相关。
* Addend，可通过 getAddend访问，是一个常量，其解释由边的 Kind决定。
* Section– 一组Symbol实例，加上一组Block 实例，有一个Name，一组ProtectionFlags，和一个Ordinal。

段使迭代与源对象文件中特定节相关联的符号或块变得容易。

* blocks()返回节中定义的块集的迭代器（作为Block\*指针）。
* symbols()返回节中定义的符号集的迭代器（作为Symbol\*指针）。
* Name表示为llvm::StringRef, 并且可以通过 getName方法访问。
* ProtectionFlags表示为 sys::Memory::ProtectionFlags 枚举，并可通过该getProtectionFlags方法访问。这些标志描述了该部分是可读的、可写的、可执行的还是这些的某种组合。最常见的组合是RW-可写数据、 R--常量数据和R-X代码。
* SectionOrdinal，可通过访问getOrdinal，是一个数字，用于相对于其他部分对部分进行排序。它通常用于在布置内存时保留段（具有相同内存保护的一组段）内的段顺序。

对于图论学家：这LinkGraph是二分的，有一组 Symbol节点和一组Addressable节点。每个Symbol节点都有一条到其目标的（隐式）边Addressable。每个Block都有一组边（可能是空的，表示为Edge实例）返回到集合的元素Symbol。为了通用算法的方便和性能，符号和块进一步分组为Sections.

它LinkGraph本身提供了用于构造、删除和迭代节、符号和块的操作。它还提供与链接过程相关的元数据和实用程序：

* 图元素操作
* sections返回图中所有段的迭代器。
* findSectionByNameSection\*如果存在，则返回指向具有给定名称（作为 a nullptr）的段的指针，否则返回 。
* blocks返回图中所有块的迭代器（跨所有部分）。
* defined\_symbols返回图中所有已定义符号的迭代器（跨所有部分）。
* external\_symbols返回图中所有外部符号的迭代器。
* absolute\_symbols返回图中所有绝对符号的迭代器。
* reateSection创建具有给定名称和保护标志的段。
* createContentBlock使用给定的初始内容、父段、地址、对齐方式和对齐偏移量创建一个块。
* createZeroFillBlock创建具有给定大小、父段、地址、对齐方式和对齐偏移量的零填充块。
* addExternalSymbol使用给定的名称、大小和链接创建一个新的可寻址和符号。
* addAbsoluteSymbol使用给定的名称、地址、大小、链接、范围和活性创建一个新的可寻址和符号。
* addCommonSymbol用于创建具有给定名称、范围、部分、初始地址、大小、对齐方式和活性的零填充块和弱符号的便利函数。
* addAnonymousSymbol为给定的块、偏移量、大小、可调用性和活跃度创建一个新的匿名符号。
* addDefinedSymbol使用名称、偏移量、大小、链接、范围、可调用性和活跃度为给定块创建一个新符号。
* makeExternal通过创建一个新的可寻址并将符号指向它，将以前定义的符号转换为外部符号。现有块不会被删除，但可以通过调用手动删除（如果未引用）removeBlock。符号的所有边仍然有效，但现在必须在 this 之外定义符号LinkGraph。
* removeExternalSymbol删除外部符号及其目标可寻址。目标可寻址不能被任何其他符号引用。
* removeAbsoluteSymbol删除绝对符号及其目标可寻址。目标可寻址不能被任何其他符号引用。
* removeDefinedSymbol删除定义的符号，但不删除其目标块。
* removeBlock删除给定的块。
* splitBlock在给定索引处将给定块一分为二（在已知块包含可分解记录的情况下很有用，例如 eh-frame 部分中的 CFI 记录）。
* 图实用程序操作
* getName返回此图的名称，通常基于输入目标文件的名称。
* getTargetTriple为执行进程返回一个llvm::Triple 。
* getPointerSize返回执行程序进程中指针的大小（以字节为单位）。
* getEndinaness返回执行程序进程的字节顺序。
* allocateString将给定的数据复制llvm::Twine到链接图的内部分配器中。这可用于确保在传递内创建的内容比该传递的执行更有效。

## 通用链接算法

JITLink 提供了一个通用的链接算法，可以通过引入 JITLink Pa[https://llvm.org/docs/JITLink.html#passes](https://llvm.org/docs/JITLink.html" \l "passes)sses在某些点进行扩展/修改：

* 1.阶段1

一旦初始配置（包括传递管道设置）完成，link函数就会立即调用此阶段。

* 1).运行 pre-prune passes。

在graph被修剪之前，这些passes在graph上被调用。在这个阶段 LinkGraph节点仍然有它们原来的 vmaddr。mark-live pass（由JITLinkContext 提供）将在此序列的末尾运行以标记初始的实时符号集。

值得注意的用例：将节点标记为活动、访问/复制将被修剪的graph数据（例如，对 JIT 很重要但链接过程不需要的元数据）。

* 2).修剪（dead-strip）LinkGraph。

从初始的实时符号集中删除所有无法访问的符号和块。

这允许 JITLink 删除无法访问的符号/内容，包括覆盖的弱和冗余 ODR 定义。

* 3).运行post-prune passes。

这些passes在 dead-stripping 之后，但在分配内存或节点分配其最终目标 vmaddrs 之前在graph上运行。

在此阶段运行的passes受益于修剪，因为死函数和数据已从graph中剥离。然而，新的内容仍然可以添加到graph中，因为目标和工作内存尚未分配。

值得注意的用例：构建全局偏移表 (GOT)、过程链接表 (PLT) 和线程局部变量 (TLV) 条目。

* 4).将块分类为段。

按顺序对所有块进行排序，然后按地址对所有块进行排序。将具有匹配权限的部分收集到段中，并计算这些段的大小以进行内存分配。

* 5).分配段内存，更新节点地址。

调用为graph分配工作内存JITLinkContext和JITLinkMemoryManager目标内存，然后将所有节点地址更新为其分配的目标地址。

注意：此步骤仅更新此graph中定义的节点地址。外部符号仍将具有空地址。

* 6).运行post-prune passes

在分配工作内存和目标内存之后，但在JITLinkContext通知graph中符号的最终地址之前，这些passes在graph上运行。这使这些passes有机会在任何 JITLink 客户端（尤其是符号解析的 ORC 查询）尝试访问它们之前设置与目标地址关联的数据结构。

值得注意的用例：设置目标地址和 JIT 数据结构之间的映射，例如 \_\_dso\_handle和JITDylib\* 之间的映射 。

* 7).通知JITLinkContext分配的符号地址。

调用JITLinkContext::notifyResolved链接graph，允许客户端对为此graph分配的符号地址做出反应。在 ORC 中，这用于通知任何对已解析符号的挂起查询，包括来自并发运行的 JITLink 实例的挂起查询，这些实例已到达下一步并正在等待此grpah中符号的地址以继续其链接。

* 8).识别外部符号并异步解析它们的地址。

调用JITLinkContext来解析图中任何外部符号的目标地址。此步骤是异步的——JITLink 会将链接状态打包到一个延续中，以便在解析符号后运行。

这是第 1 阶段的最后一步。

* 2. 阶段2

这个阶段由在上面的外部符号解析步骤结束时构造的延续调用。

* 1.)应用外部符号解析结果。

这会更新所有外部符号的地址。此时图中的所有节点都有它们的最终目标地址，但是节点内容仍然指向目标文件中的原始数据。

* 2.) 运行pre-fixup passes

在为所有节点分配了最终目标地址之后，但在将节点内容复制到工作内存并修复之前，将在grpah上调用这些传递。在此阶段运行的通道可以根据地址布局对graph和内容进行后期优化。

值得注意的用例：GOT 和 PLT 放宽，其中绕过 GOT 和 PLT 访问以获取可在分配的内存布局下直接访问的修复目标。

* 3.)将块内容复制到工作内存并应用修正

将所有块内容复制到分配的工作内存中（遵循目标布局）并应用修正。graph块被更新以指向固定的内容。

* 4.)运行post-fixup passes。

在应用修复并更新块以指向修复的内容后，将在graph上调用这些passes。

修复后psses可以检查块内容以查看将被复制到指定目标地址的确切字节。

* 5).异步完成内存.

调用JITLinkMemoryManager将工作内存复制到执行程序进程并应用请求的权限。此步骤是异步的——JITLink 会将链接状态打包到一个延续中，以便在内存被复制和保护后运行。

这是第 2 阶段的最后一步。

* 3.第 3 阶段。

这个阶段由在上面的内存终结步骤结束时构造的延续调用。

* 通知上下文graph已发出。

为该图的内存分配调用JITLinkContext::notifyFinalized并移交JITLinkMemoryManager::Allocation 对象。这允许上下文跟踪/保持内存分配并对新发出的定义做出反应。在 ORC 中，这用于更新ExecutionSession实例的依赖关系图，如果它们的所有依赖关系也已发出，这可能会导致这些符号（可能还有其他符号）变为Ready 。

### passes

JITLink passes 是std::function<Error(LinkGraph&)>实例。他们可以自由地检查和修改给定的LinkGraph主题，以遵守他们正在运行的任何阶段的约束（请参阅[通用链接算法](https://llvm.org/docs/JITLink.html" \l "generic-link-algorithm)）。如果passes返回Error::success()，则链接继续。如果passes 返回失败值，则链接停止并通知JITLinkContext链接失败。

JITLink 后端（例如 MachO/x86-64 将 GOT 和 PLT 构造实现为一个pass）和外部客户端（如 ObjectLinkingLayer::Plugin）都会使用passes。

结合开放的LinkGraphAPI，JITLink passes可以实现强大的新功能。例如：

* 松弛优化(Relaxation optimizations)——pre-fixup pass 可以检查 GOT 访问和 PLT 调用，并识别入口目标地址和访问地址足够接近以直接访问的情况。在这种情况下，pass 可以重写包含块的指令流并更新 fixup edges 以使直接访问。

这个代码看起来像：

|  |
| --- |
| Error relaxGOTEdges(LinkGraph &G) {  for (auto \*B : G.blocks())  for (auto &E : B->edges())  if (E.getKind() == x86\_64::GOTLoad) {  auto &GOTTarget = getGOTEntryTarget(E.getTarget());  if (isInRange(B.getFixupAddress(E), GOTTarget)) {  // Rewrite B.getContent() at fixup address from  // MOVQ to LEAQ  // Update edge target and kind.  E.setTarget(GOTTarget);  E.setKind(x86\_64::PCRel32);  }  }  return Error::success();  } |

* 元数据注册——Post allocation passes可用于记录目标中各部分的地址范围。一旦内存完成，这可用于在目标中注册元数据（例如异常处理框架、语言元数据）。

|  |
| --- |
| Error registerEHFrameSection(LinkGraph &G) {  if (auto \*Sec = G.findSectionByName("\_\_eh\_frame")) {  SectionRange SR(\*Sec);  registerEHFrameSection(SR.getStart(), SR.getEnd());  }  return Error::success();  } |



* Record call sites for later mutation – post-allocation pass 可以记录对特定函数的所有调用的调用点，允许这些调用点稍后在运行时更新（例如用于检测，或使函数能够被延迟编译但仍然编译后直接调用）。

|  |
| --- |
| StringRef FunctionName = "foo";std::vector<JITTargetAddress> CallSitesForFunction;  auto RecordCallSites =  [&](LinkGraph &G) -> Error {  for (auto \*B : G.blocks())  for (auto &E : B.edges())  if (E.getKind() == CallEdgeKind &&  E.getTarget().hasName() &&  E.getTraget().getName() == FunctionName)  CallSitesForFunction.push\_back(B.getFixupAddress(E));  return Error::success();  }; |

### 使用 JITLinkMemoryManager 进行内存管理

JIT 链接需要分配两种内存：JIT 进程中的工作内存和执行进程中的目标内存（这些进程和内存分配可能是相同的，具体取决于用户希望如何构建他们的 JIT）。它还要求这些分配符合目标进程中请求的代码模型（例如 MachO/x86-64 的 Small 代码模型要求模拟 dylib 的所有代码和数据分配在 4Gb 以内）。最后，很自然地让内存管理器负责将内存传输到目标地址空间并应用内存保护，因为内存管理器必须知道如何与执行程序通信。

为了满足这些要求，JITLinkMemoryManager采用了以下设计：内存管理器本身只有一个虚方法，该虚方法返回一个JITLinkMemoryManager::Allocation：

|  |
| --- |
| virtual Expected<std::unique\_ptr<Allocation>>allocate(const JITLinkDylib \*JD, const SegmentsRequestMap &Request) = 0; |

此方法采用JITLinkDylib\*表示目标模拟的 dylib，以及必须为此对象分配的完整的段的集合。 JITLinkMemoryManager实现可以（可选）使用该JD 参数来管理每个模拟的 dylib 内存池（因为代码模型约束通常强加在每个 dylib 的基础上，而不是跨 dylib）[2]。该Request参数通过预先描述当前对象中的所有部分，允许实现者将这些段分配为单个 slab，在预先分配的 per-jitdylib 池中或直接从系统内存中分配。

后续所有操作均由 JITLinkMemoryManager::Allocation接口提供：

* virtual MutableArrayRef<char> getWorkingMemory(ProtectionFlags Seg)

应该被覆盖以返回具有给定保护标志的段在工作内存中的地址。

* virtual JITTargetAddress getTargetMemory(ProtectionFlags Seg)

应该被覆盖以返回具有给定保护标志的段的执行程序地址空间中的地址。

* virtual void finalizeAsync(FinalizeContinuation OnFinalize)

应该被覆盖以将工作内存的内容复制到目标地址空间并对所有段应用内存保护。在工作内存和目标内存分开的情况下，此方法应释放工作内存。

* virtual Error deallocate()

应该被覆盖以释放目标地址空间中的内存。

JITLink 提供了此接口的简单进程内实现： InProcessMemoryManager。 它分配一次页面并将它们重新用作工作内存和目标内存。

ORC 提供了一个基于orc::TargetProcessControl API: OrcRPCTPCJITLinkMemoryManager的基于 ORC-RPC 实现的cross-process JITLinkMemoryManager。此 API 使用 TargetProcessControl API 调用来分配和管理远程进程中的内存。底层通信通道由 ORC-RPC passes类型决定。常见选项包括 unix 套接字或 TCP。

### JITLinkMemoryManager 和安全性

JITLink 为单独的执行程序进程链接 JIT 代码的能力可用于提高 JIT 系统的安全性：执行程序进程可以在沙箱中运行，在 VM 中运行，甚至在完全独立的机器上运行。

JITLink 的内存管理器接口非常灵活，可以在性能和安全性之间进行一系列权衡。例如，在必须对代码页进行签名（防止代码被更新）的系统上，内存管理器可以在链接到运行 JITLink 的进程中的空闲内存后释放工作内存页。或者，在允许 RWX 页面的系统上，内存管理器可以通过将相同的页面标记为 RWX 来为工作内存和目标内存使用相同的页面，从而允许在没有额外开销的情况下就地修改代码。最后，如果不允许 RWX 页面但允许物理内存页面的双虚拟映射，那么内存管理器可以将物理页面双重映射为 RW- in the JITLink process和R-X in the executor process。

### 错误处理

JITLink 广泛使用了该类型（有关详细信息，请参阅[LLVM 程序员手册](https://llvm.org/docs/ProgrammersManual.html)llvm::Error的错误处理部分）。链接进程本身、所有passes、内存管理器接口和对 JITLinkContext的操作 都允许失败。鼓励链接graph构造实用程序（尤其是对象格式的解析器）在应用之前验证输入和验证修正（例如，使用范围检查）。

任何错误都将停止链接过程并通知上下文失败。在 ORC 中，报告的故障会传播到等待失败链接提供的定义的查询，并且还会通过依赖graph的边缘传播到等待依赖符号的任何查询。

## Connection to the ORC Runtime

ORC 运行时（目前正在开发中）旨在为高级 JIT 功能提供运行时支持，包括需要在执行器中执行重要操作的对象格式功能（例如运行初始化器、管理线程本地存储、注册语言运行时等）。

ORC Runtime 对对象格式特性的支持通常需要运行时（在执行程序进程中执行）和 JITLink（在 JIT 进程中运行并且可以检查 LinkGraphs 以确定必须在执行程序中执行哪些操作）之间的协作。例如：ORC 运行时中 MachO 静态初始化器的执行由jit\_dlopen函数执行，该函数回调 JIT 进程以请求要遍历的 \_\_mod\_init 段的地址范围列表。此列表由 MachOPlatformPlugin整理 ，它安装一个pass来记录每个对象的此信息，因为它链接到目标中。

## 构建 LinkGraphs

客户端通常访问和操作LinkGraph实例为它们创建的ObjectLinkingLayer实例，但它们可以手动创建：

* 1.通过直接构建和填充LinkGraph实例。
* 2.通过使用createLinkGraph函数族 LinkGraph从包含目标文件的内存缓冲区创建一个。这 ObjectLinkingLayer通常是创建LinkGraphs.
* 1.) createLinkGraph\_<Object-Format>\_<Architecture>可以在对象格式和体系结构都是提前知道的时候使用。
* 2).createLinkGraph\_<Object-Format>可以在对象格式提前已知但体系结构未知时使用。在这种情况下，体系结构将通过检查对象标头来确定。
* 3.)createLinkGraph可以在事先不知道对象格式和体系结构的情况下使用。在这种情况下，将检查对象标头以确定格式和体系结构。

## JIT Linking

JIT 链接器概念是在 LLVM 的早期 JIT API MCJIT 中引入的。在 MCJIT 中，RuntimeDyld组件通过将内存中链接步骤添加到常用编译器管道的末尾，实现了将 LLVM 重新用作内存中编译器。MCJIT 不是像编译器通常那样将可重定位对象转储到磁盘，而是将它们传递给 RuntimeDyld 以链接到目标进程。

这种链接方法不同于标准的静态或动态链接：

* 静态链接器将一个或多个可重定位目标文件作为输入，并将它们链接到磁盘上的可执行文件或动态库中。
* 动态链接器将重定位应用于已加载到内存中的可执行文件和动态库。
* JIT 链接器一次获取一个可重定位目标文件并将其链接到目标进程，通常使用上下文对象以允许链接代码解析目标中的符号。

### RuntimeDyld

为了保持 RuntimeDyld 的实现简单，MCJIT 对编译代码施加了一些限制：

* 1.它必须使用大型代码模型，并且经常限制可用的重定位模型以限制必须支持的重定位类型。
* 2.它需要所有符号的强链接和默认可见性——其他链接/可见性的行为没有明确定义。
* 3.它限制和/或禁止使用需要运行时支持的功能，例如静态初始化程序或线程本地存储。

由于这些限制，并非 LLVM 支持的所有语言功能都在 MCJIT 下工作，并且必须编译要在 JIT 下加载的对象以针对它（排除在 JIT 下使用来自其他来源的预编译代码）。

RuntimeDyld 还提供了对链接过程本身非常有限的可见性：客户端可以访问节大小的保守估计（RuntimeDyld 将存根大小和填充估计捆绑到节大小值中）和最终重定位的字节，但无法访问 RuntimeDyld 的内部对象表示。

消除这些约束和限制是开发 JITLink 的主要动机之一。

## llvm-jitlink 工具

该llvm-jitlink工具是 JITLink 库的命令行包装器。它加载一组可重定位目标文件，然后使用 JITLink 链接它们。根据使用的选项，它将执行它们，或验证链接的内存。

该llvm-jitlink工具最初旨在通过提供简单的测试环境来帮助 JITLink 开发。

### 基本用法

默认情况下，llvm-jitlink将链接命令行上传递的对象集，然后搜索“main”函数并执行它：

|  |
| --- |
| % cat hello-world.c  #include <stdio.h>  int main(int argc, char \*argv[]) {  printf("hello, world!\n");  return 0;  }  % clang -c -o hello-world.o hello-world.c  % llvm-jitlink hello-world.o  Hello, World! |

可以指定多个对象，并且可以使用 -args 选项将参数提供给 JIT 的主函数：

|  |
| --- |
| #include <stdio.h>  void print\_args(int argc, char \*argv[]) {  for (int i = 0; i != argc; ++i)  printf("arg %i is \"%s\"\n", i, argv[i]);  }  % cat print-args-main.c  void print\_args(int argc, char \*argv[]);  int main(int argc, char \*argv[]) {  print\_args(argc, argv);  return 0;  }  % clang -c -o print-args.o print-args.c  % clang -c -o print-args-main.o print-args-main.c  % llvm-jitlink print-args.o print-args-main.o -args a b c  arg 0 is "a"  arg 1 is "b"  arg 2 is "c" |

可以使用该选项指定替代入口点。-entry <entry point name>

可以通过调用找到其他选项。llvm-jitlink -help

### llvm-jitlink 作为回归测试工具

llvm-jitlink的主要目标之一是为 JITLink 启用可读的回归测试。为此，它支持两个选项：

* 该-noexec选项告诉 llvm-jitlink 在查找入口点之后并在尝试执行它之前停止。由于未执行链接代码，即使您无权访问被链接的目标，这也可用于链接其他目标（在这种情况下，-define-abs或-phony-externals 选项可用于提供任何缺少的定义）。
* 该-check <check-file>选项可用于 针对工作内存运行一组jitlink-check表达式。它通常与 -noexec结合使用 ，因为目的是验证 JIT 内存而不是运行代码，并-noexec-check允许我们从当前进程链接任何支持的目标架构。在-check模式下，llvm-jitlink将扫描给定的检查文件以查找# jitlink-check: <expr>格式. 请参阅llvm/test/ExecutionEngine/JITLink 中的此用法示例。

### 通过 llvm-jitlink-executor 远程执行

默认情况下llvm-jitlink会将给定的对象链接到它自己的进程中，但这可以被两个选项覆盖：

* 该-oop-executor[=/path/to/executor]选项指示llvm-jitlink执行给定的执行程序（默认为llvm-jitlink-executor）并通过文件描述符与其进行通信，带有filedescs=<in-fd>,<out-fd>格式的文件描述符作为第一个参数传递给执行程序。
* 该-oop-executor-connect=<host>:<port>选项指示llvm-jitlink通过给定主机和端口上的 TCP 连接到已经运行的执行程序。要使用此选项，您需要手动启动llvm-jitlink-executor并将 listen=<host>:<port>作为第一个参数。

### Harness mode线束模式

该-harness选项允许将一组输入对象指定为测试工具，并将常规对象文件隐式视为要测试的对象。线束集中符号的定义覆盖测试集中的定义，线束的外部引用导致测试集中本地符号的自动作用域提升（这些对通常的链接器规则的修改是通过在看到llvm-jitlink-harness选项时安装由ObjectLinkingLayer::Plugin来完成的).

通过这些修改，我们可以通过模拟这些函数的被调用者来选择性地测试目标文件中的函数。例如，假设我们有一个目标文件 ，test\_code.o它是从以下 C 源代码（我们不需要访问它）编译而来的：

|  |
| --- |
| void irrelevant\_function() { irrelevant\_external(); }  int function\_to\_mock(int X) {  return /\* some function of X \*/;  }  static void function\_to\_test() {  ...  int Y = function\_to\_mock();  printf("Y is %i\n", Y);  } |

如果我们想知道当我们改变function\_to\_mock行为时，function\_to\_test行为如何，我们可以通过编写测试工具来测试它：

|  |
| --- |
| void function\_to\_test();  int function\_to\_mock(int X) {  printf("used mock utility function\n");  return 42;  }  int main(int argc, char \*argv[]) {  function\_to\_test():  return 0;  } |

一般情况下这些对象是不能链接在一起的： function\_to\_test是静态的，不能在外面解析 test\_code.o，这两个function\_to\_mock函数会导致重复定义的错误，irrelevant\_external而是未定义的。但是，使用-harness 和 -phony-externals我们可以运行此代码：

|  |
| --- |
| % clang -c -o test\_code\_harness.o test\_code\_harness.c  % llvm-jitlink -phony-externals test\_code.o -harness test\_code\_harness.o  used mock utility function  Y is 42 |

-harness想要对构建产品执行一些非常晚的测试以验证编译后的代码是否按预期运行的人可能会对这个选项感兴趣。在基本的 C 测试用例中，这是相对简单的。更复杂的语言（例如 C++）的模拟要棘手得多：任何涉及类的代码往往有很多非平凡的表面区域（例如 vtables），需要非常小心地模拟。

### JITLink 后端开发人员的技巧

1.自由使用 assert 和llvm::Error. 不要假设输入对象的格式正确：返回 libObject（或您自己的对象解析代码）产生的任何错误，并在构造时进行验证。llvm::Error仔细考虑合同（应使用断言和 llvm\_unreachable 进行验证）和环境错误（应生成实例）之间的区别 。

2.不要假设您正在链接过程中。在LinkGraph.

作为“最小可行”JITLink 包装器，该llvm-jitlink工具对于引入新 JITLink 后端的开发人员来说是宝贵的资源。一个标准的工作流程是首先在工具上抛出一个不受支持的对象，然后查看返回的错误，然后修复它（您通常可以根据其他格式或体系结构的现有代码合理猜测应该做什么）。

在 LLVM 的调试版本中，该-debug-only=jitlink选项会在链接过程中从 JITLink 库中转储日志。这些对于一目了然地发现一些错误很有用。该-debug-only=llvm\_jitlink选项从llvm-jitlink 工具中转储日志，这对于调试测试用例（通常比 更简洁 -debug-only=jitlink）和工具本身很有用。

和选项有助于测试跨进程-oop-executor和-oop-executor-connect跨架构用例的处理。

## 路线图

JITLink 正在积极开发中。到目前为止的工作主要集中在 MachO 的实现上。在 LLVM 12 中，对 x86-64 上的 ELF 的支持有限。

主要优秀项目包括：

重构架构支持以最大化跨格式共享。

所有格式都应该能够为每个受支持的体系结构共享大量体系结构特定代码（尤其是重定位）。

重构 ELF 链接图构造。

ELF 的链接图构造目前在ELF\_x86\_64.cpp 文件中实现，并绑定到 x86-64 重定位解析代码。大部分代码是通用的，应该按照与现有通用 MachOLinkGraphBuilder 相同的方式拆分为 ELFLinkGraphBuilder 基类。

为 arm64 实现 ELF 支持。

一旦架构支持代码被重构以支持共享并且 ELF 链接图构造被重构以允许重用，我们应该能够通过组合这些现有的部分来构建 ELF / arm64 JITLink 实现。

实施对新架构的支持。

## 实现对 COFF 的支持。

目前还没有 JITLink 的 COFF 实现。这样的实现应该遵循 MachO 和 ELF 路径：一个通用的 COFFLinkGraphBuilder 基类，可以专门用于每个体系结构。

设计并实现基于共享内存的 JITLinkMemoryManager。

一个预期常见的用例是针对同一台机器上的另一个进程的进程外链接。这允许 JIT 将 JIT 代码沙盒化。对于此用例，基于共享内存的 JITLinkMemoryManager 将提供最有效的分配形式。创建一个将需要为共享内存设计一个通用 API，因为 LLVM 目前没有。

Availability and Status

Architecture ELF COFF MachO

arm64 Partial (small code model, PIC relocation model only)

x86-64 Partial Full (except TLV and debugging)

[1] 有关llvm/examples/OrcV2Examples/LLJITWithObjectLinkingLayerPlugin完整的示例，请参见。

[2] 如果不是隐藏范围的符号，我们可以消除 JITLinkDylib\*参数JITLinkMemoryManager::allocate并将每个对象视为一个单独的模拟 dylib，用于内存布局。隐藏符号通过生成对外部符号的范围内访问来打破这一点，要求访问和符号在彼此的范围内分配。也就是说，为每个模拟的 dylib 提供一个预先保留的地址范围池保证松弛优化将启动所有 intra-dylib 引用，这对性能有好处（以通过保留地址范围引入的任何开销为代价）预先）。

# 调试 JIT 代码

### 背景

如果没有特殊的运行时支持，调试动态生成的代码会非常痛苦。调试器通常从磁盘上的目标文件中读取调试信息，但对于 JIT 代码，没有这样的文件可供查找。

为了移交必要的调试信息，GDB 建立了一个 用于向调试器注册 JITed 代码的接口。LLDB 在 JITLoaderGDB 插件中实现了它。在 JIT 方面，LLVM MCJIT 确实实现了 ELF 对象文件的接口。

在高层次上，每当 MCJIT 生成新的机器代码时，它都会在包含 DWARF 格式的调试信息的内存目标文件中执行。然后 MCJIT 将这个内存中的目标文件添加到动态生成的目标文件的全局列表中，并调用 \_\_jit\_debug\_register\_code调试器知道的特殊函数。当调试器附加到一个进程时，它会在此函数中放置一个断点并将一个特殊的处理程序与其相关联。一旦 MCJIT 调用了注册函数，调试器就会捕捉到断点信号，从下级内存中加载新的目标文件并恢复执行。这样它可以获得纯内存中目标文件的调试信息。

## GDB 版本

为了调试由 LLVM 进行的 JIT 代码，您需要 GDB 7.0 或更新版本，它在大多数现代 Linux 发行版上都可用。Apple 随 Xcode 一起发布的 GDB 版本暂时冻结在 6.3。

## LLDB 版本

由于 6.0 版本的回归，LLDB 暂时不支持 JITed 代码调试。该错误最近在主线中得到修复，因此从即将发布的 12.0 版本开始，应该可以再次调试 JITed ELF 对象。在 macOS 上，必须使用plugin.jit-loader.gdb.enable 设置明确启用该功能。

## 调试 MCJIT 代码

LLVM 的新兴 MCJIT 组件允许使用 GDB 对 JIT-ed 代码进行全面调试。这是由于 MCJIT 能够使用 MC 发射器向 GDB 提供完整的 DWARF 调试信息。

请注意，必须将 lli--jit-kind=mcjit标志传递给使用 MCJIT 而不是更新的 ORC JIT 对代码进行 JIT。

例子

考虑以下 C 代码（添加了行号以使示例更易于理解）：

|  |
| --- |
| 1 int compute\_factorial(int n)  2 {  3 if (n <= 1)  4 return 1;  5  6 int f = n;  7 while (--n > 1)  8 f \*= n;  9 return f;  10 }  11  12  13 int main(int argc, char\*\* argv)  14 {  15 if (argc < 2)  16 return -1;  17 char firstletter = argv[1][0];  18 int result = compute\_factorial(firstletter - '0');  19  20 // Returned result is clipped at 255...  21 return result;  22 } |

这是一个示例命令行会话，显示了如何通过lliLLDB 内部构建和运行此代码：

|  |
| --- |
| > export BINPATH=/workspaces/llvm-project/build/bin  > $BINPATH/clang -g -S -emit-llvm --target=x86\_64-unknown-unknown-elf showdebug.c  > lldb $BINPATH/lli  (lldb) target create "/workspaces/llvm-project/build/bin/lli"  Current executable set to '/workspaces/llvm-project/build/bin/lli' (x86\_64).  (lldb) settings set plugin.jit-loader.gdb.enable on  (lldb) b compute\_factorial  Breakpoint 1: no locations (pending).  WARNING: Unable to resolve breakpoint to any actual locations.  (lldb) run --jit-kind=mcjit showdebug.ll 5  1 location added to breakpoint 1  Process 21340 stopped  \* thread #1, name = 'lli', stop reason = breakpoint 1.1  frame #0: 0x00007ffff7fd0007 JIT(0x45c2cb0)`compute\_factorial(n=5) at showdebug.c:3:11  1 int compute\_factorial(int n)  2 {  -> 3 if (n <= 1)  4 return 1;  5 int f = n;  6 while (--n > 1)  7 f \*= n;  (lldb) p n  (int) $0 = 5  (lldb) b showdebug.c:9  Breakpoint 2: where = JIT(0x45c2cb0)`compute\_factorial + 60 at showdebug.c:9:1, address = 0x00007ffff7fd003c  (lldb) c  Process 21340 resuming  Process 21340 stopped  \* thread #1, name = 'lli', stop reason = breakpoint 2.1  frame #0: 0x00007ffff7fd003c JIT(0x45c2cb0)`compute\_factorial(n=1) at showdebug.c:9:1  6 while (--n > 1)  7 f \*= n;  8 return f;  -> 9 }  10  11 int main(int argc, char\*\* argv)  12 {  (lldb) p f  (int) $1 = 120  (lldb) bt  \* thread #1, name = 'lli', stop reason = breakpoint 2.1  \* frame #0: 0x00007ffff7fd003c JIT(0x45c2cb0)`compute\_factorial(n=1) at showdebug.c:9:1  frame #1: 0x00007ffff7fd0095 JIT(0x45c2cb0)`main(argc=2, argv=0x00000000046122f0) at showdebug.c:16:18  frame #2: 0x0000000002a8306e lli`llvm::MCJIT::runFunction(this=0x000000000458ed10, F=0x0000000004589ff8, ArgValues=ArrayRef<llvm::GenericValue> @ 0x00007fffffffc798) at MCJIT.cpp:554:31  frame #3: 0x00000000029bdb45 lli`llvm::ExecutionEngine::runFunctionAsMain(this=0x000000000458ed10, Fn=0x0000000004589ff8, argv=size=0, envp=0x00007fffffffe140) at ExecutionEngine.cpp:467:10  frame #4: 0x0000000001f2fc2f lli`main(argc=4, argv=0x00007fffffffe118, envp=0x00007fffffffe140) at lli.cpp:643:18  frame #5: 0x00007ffff788c09b libc.so.6`\_\_libc\_start\_main(main=(lli`main at lli.cpp:387), argc=4, argv=0x00007fffffffe118, init=<unavailable>, fini=<unavailable>, rtld\_fini=<unavailable>, stack\_end=0x00007fffffffe108) at libc-start.c:308:16  frame #6: 0x0000000001f2dc7a lli`\_start + 42  (lldb) finish  Process 21340 stopped  \* thread #1, name = 'lli', stop reason = step out  Return value: (int) $2 = 120  frame #0: 0x00007ffff7fd0095 JIT(0x45c2cb0)`main(argc=2, argv=0x00000000046122f0) at showdebug.c:16:9  13 if (argc < 2)  14 return -1;  15 char firstletter = argv[1][0];  -> 16 int result = compute\_factorial(firstletter - '0');  17  18 // Returned result is clipped at 255...  19 return result;  (lldb) p result  (int) $3 = 73670648  (lldb) n  Process 21340 stopped  \* thread #1, name = 'lli', stop reason = step over  frame #0: 0x00007ffff7fd0098 JIT(0x45c2cb0)`main(argc=2, argv=0x00000000046122f0) at showdebug.c:19:12  16 int result = compute\_factorial(firstletter - '0');  17  18 // Returned result is clipped at 255...  -> 19 return result;  20 }  (lldb) p result  (int) $4 = 120  (lldb) expr result=42  (int) $5 = 42  (lldb) p result  (int) $6 = 42  (lldb) c  Process 21340 resuming  Process 21340 exited with status = 42 (0x0000002a)  (lldb) exit |

# 命令行 2.0 库手册

简介

本文档描述了 CommandLine 参数处理库。它将向您展示如何使用它，以及它可以做什么。CommandLine 库使用声明性方法来指定程序采用的命令行选项。默认情况下，这些选项声明隐式保存为声明的选项解析的值（当然这可以更改）。

尽管有许多不同语言的命令行参数解析库，但没有一个能很好地满足我的需要。通过查看其他库的特性和问题，我将 CommandLine 库设计为具有以下特性：

1.速度：CommandLine 库非常快并且占用的资源很少。库的解析时间与解析的参数数量成正比，而不是识别的选项数量。此外，命令行参数值被透明地捕获到用户定义的全局变量中，可以像访问任何其他变量一样访问这些变量（并且具有相同的性能）。

2.类型安全：作为 CommandLine 的用户，您不必担心记住您想要的参数类型（它是一个 int？一个字符串？一个 bool？一个枚举？）并继续转换它。这不仅有助于防止容易出错的构造，还可以显着简化源代码。

3.不需要子类：要使用 CommandLine，您实例化对应于您想要捕获的参数的变量，而不是解析器的子类。这意味着您不必编写任何 样板代码。

4.全球可访问：图书馆可以指定命令行参数，这些参数在链接到图书馆的任何工具中自动启用。这是可能的，因为应用程序不必保留要传递给解析器的参数列表。这也使得支持动态加载的选项变得 微不足道。

5.更清洁：CommandLine 直接支持枚举和其他类型，这意味着库中内置了更少的错误和更高的安全性。您不必担心您的完整命令行参数是否意外地分配了一个对您的枚举类型无效的值。

6.强大：CommandLine 库支持许多不同类型的参数，从简单的布尔标志到标量参数（字符串、 整数、枚举、双精度），再到参数列表。这是可能的，因为 CommandLine 是……

7.可扩展：向 CommandLine 添加新的参数类型非常简单。只需在声明时通过命令行选项指定要使用的解析器。自定义解析器没问题。

8.节省劳动力：CommandLine 库减少了用户必须完成的繁重工作量。例如，它会自动提供一个 -help选项，显示您的工具可用的命令行选项。此外，它还为您完成大部分基本的正确性检查。

9.Capable：CommandLine 库可以处理许多在实际程序中常见的不同形式的选项。例如，位置参数、 ls样式分组选项（以允许自然地处理“”）、样式前缀选项（以解析“ ”）和解释器样式选项。ls -ladld-lmalloc -L/usr/lib

本文档有望让您快速轻松地开始在您的实用程序中使用 CommandLine。此外，它应该是一个简单的参考手册来弄清楚东西是如何工作的。

快速入门指南

手册的这一部分贯穿了一个基本编译器工具的简单命令行化。这是为了向您展示如何在您自己的程序中开始使用 CommandLine 库，并向您展示它可以做的一些很酷的事情。

首先，您需要将 CommandLine 头文件包含到您的程序中：

#include "llvm/Support/CommandLine.h"

此外，您需要将其添加为主程序的第一行：

int main(int argc, char \*\*argv) {

cl::ParseCommandLineOptions(argc, argv);

...}

…实际上解析参数并填写变量声明。

现在您已准备好支持命令行参数，我们需要告诉系统我们想要哪些参数，以及它们是什么类型的参数。CommandLine 库使用声明性语法对命令行参数进行建模，并使用捕获解析值的全局变量声明。这意味着对于您想要支持的每个命令行选项，都应该有一个全局变量声明来捕获结果。例如，在编译器中，我们希望支持 Unix 标准的 ' ' 选项来指定输出的位置。使用 CommandLine 库，它表示如下：-o <filename>

cl::opt<string> OutputFilename("o", cl::desc("Specify output filename"), cl::value\_desc("filename"));

这声明了一个全局变量“ OutputFilename”，用于捕获“ o”参数（第一个参数）的结果。cl::opt我们通过使用“ ”模板（而不是“ ”模板）指定这是一个简单的标量选项cl::list，并告诉 CommandLine 库我们正在解析的数据类型是字符串。

第二个和第三个参数（可选）用于指定“ -help”选项输出的内容。在这种情况下，我们得到一行如下所示：

USAGE: compiler [options]

OPTIONS:

-h - Alias for -help

-help - display available options (-help-hidden for more)

-o <filename> - Specify output filename

因为我们指定命令行选项应该使用 string数据类型进行解析，所以声明的变量在可以使用普通 C++ 字符串对象的所有上下文中自动用作真实字符串。例如：

...std::ofstream Output(OutputFilename.c\_str());if (Output.good()) ......

您可以使用许多不同的选项来自定义命令行选项处理库，但上面的示例显示了这些选项的通用界面。选项可以按任何顺序指定，并使用cl::desc(…)等辅助函数指定，因此无需记住位置依赖性。参考指南中详细讨论了可用的选项 。

继续这个例子，我们想让我们的编译器接受一个输入文件名和一个输出文件名，但我们不希望用连字符指定输入文件名（即，not -filename.c）。为了支持这种类型的参数，CommandLine 库允许为程序指定位置参数。这些位置参数由非选项形式的命令行参数填充。我们像这样使用此功能：

cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::init("-"));

此声明表明第一个位置参数应被视为输入文件名。这里我们使用cl::init选项为命令行选项指定一个初始值，如果未指定选项则使用该初始值（如果不为选项指定cl::init修饰符，则默认构造函数为数据类型用于初始化值）。命令行选项默认为可选，因此如果我们希望要求用户始终指定输入文件名，我们将添加cl::Required标志，并且我们可以消除cl::init修饰符，如下所示：

cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::Required);

同样，CommandLine 库不要求以任何特定顺序指定选项，因此上述声明等同于：

cl::opt<string> InputFilename(cl::Positional, cl::Required, cl::desc("<input file>"));

通过简单地添加cl::Required标志，如果未指定参数，CommandLine 库将自动发出错误，这会将所有命令行选项验证代码从您的应用程序转移到库中。这只是使用标志如何在每个选项的基础上改变库的默认行为的一个例子。通过添加上述声明之一，-help选项概要现在扩展为：

USAGE: compiler [options] <input file>

OPTIONS:

-h - Alias for -help

-help - display available options (-help-hidden for more)

-o <filename> - Specify output filename

…表示需要输入文件名。

布尔参数

除了输入和输出文件名之外，我们希望编译器示例支持三个布尔标志：“ -f”强制将二进制输出写入终端，“ --quiet”启用安静模式，“ -q”用于向后兼容我们的一些用户。我们可以通过像这样声明布尔类型的选项来支持这些：

cl::opt<bool> Force ("f", cl::desc("Enable binary output on terminals"));cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));cl::opt<bool> Quiet2("q", cl::desc("Don't print informational messages"), cl::Hidden);

这符合您的预期：它声明了三个布尔变量（“ Force”、“ Quiet”和“ Quiet2”）来识别这些选项。请注意，“ -q”选项是使用“ cl::Hidden ”标志指定的。这个修饰符防止它被标准的“ -help”输出显示（注意它仍然显示在“ -help-hidden”输出中）。

CommandLine 库对不同的数据类型使用不同的解析器。例如，在字符串情况下，传递给选项的参数被逐字复制到字符串变量的内容中……然而，在布尔情况下我们显然不能这样做，因此我们必须使用更智能的解析器。在布尔解析器的情况下，它不允许任何选项（在这种情况下它将 true 的值分配给变量），或者它允许指定值“ true”或“ false”，允许以下任何输入：

compiler -f # No value, 'Force' == truecompiler -f=true # Value specified, 'Force' == truecompiler -f=TRUE # Value specified, 'Force' == truecompiler -f=FALSE # Value specified, 'Force' == false

……你明白了。bool 解析器只是将字符串值转换为布尔值，并拒绝诸如 ' ' 之类的东西。同样， float、double和int解析器的工作方式与您预期的一样，使用 ' ' 和 ' ' C 库调用将字符串值解析为指定的数据类型。compiler -f=foostrtolstrtod

通过上面的声明，“ ”发出以下内容：compiler -help

USAGE: compiler [options] <input file>

OPTIONS:

-f - Enable binary output on terminals

-o - Override output filename

-quiet - Don't print informational messages

-help - display available options (-help-hidden for more)

和“ ”打印这个：compiler -help-hidden

USAGE: compiler [options] <input file>

OPTIONS:

-f - Enable binary output on terminals

-o - Override output filename

-q - Don't print informational messages

-quiet - Don't print informational messages

-help - display available options (-help-hidden for more)

这个简短的示例向您展示了如何使用“ cl::opt ”类来解析简单的标量命令行参数。除了简单的标量参数外，CommandLine 库还提供原语来支持 CommandLine option aliases和选项列表。

参数别名

到目前为止，这个例子运行良好，除了我们现在需要像这样检查安静条件：

...

if (!Quiet && !Quiet2) printInformationalMessage(...);...

……这是一个真正的痛苦！我们可以使用“ cl::alias ”类使“ ”选项成为“ -q”选项的 别名，而不是为同一条件定义两个值-quiet，而不是提供一个值本身：

cl::opt<bool> Force ("f", cl::desc("Overwrite output files"));cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));cl::alias QuietA("q", cl::desc("Alias for -quiet"), cl::aliasopt(Quiet));

第三行（这是我们从上面修改的唯一一行）定义了一个“ -q”别名，它在指定时更新“ Quiet”变量（由cl::aliasopt 修饰符指定）。因为别名不保存状态，所以程序唯一需要查询的就是Quiet变量 now。别名的另一个不错的特性是它们会自动从-help 输出中隐藏自己（尽管它们仍然在 中可见）。-help-hidden output

现在应用程序代码可以简单地使用：

...

if (!Quiet) printInformationalMessage(...);...

……这好多了！“ cl::alias ”可用于为任何变量类型指定替代名称，并且有很多用途。

从一组可能性中选择一个备选方案

到目前为止，我们已经了解了 CommandLine 库如何处理内置类型，例如 std::string,bool和int，但是它如何处理它不知道的东西，例如枚举或 ' int\*'s？

答案是它使用表驱动的通用解析器（除非您指定自己的解析器，如扩展指南中所述）。此解析器将文字字符串映射到所需的任何类型，并要求您告诉它此映射应该是什么。

假设我们想为我们的优化器添加四个优化级别，使用标准标志“ -g”、“ -O0”、“ -O1”和“ -O2”。我们可以像上面那样使用布尔选项轻松实现这一点，但是这种策略存在几个问题：

1.用户可以一次指定多个选项，例如“ ”。CommandLine 库将无法为我们捕获此错误输入。compiler -O3 -O2

2.我们将不得不测试 4 个不同的变量以查看设置了哪些变量。

3.这不会映射到我们想要的数字级别……因此我们无法轻松查看是否-O1启用了某个级别>=“”。

为了解决这些问题，我们可以使用一个枚举值，让 CommandLine 库直接用适当的级别填充它，它的使用方式如下：

enum OptLevel {

g, O1, O2, O3};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),

cl::values(

clEnumVal(g , "No optimizations, enable debugging"),

clEnumVal(O1, "Enable trivial optimizations"),

clEnumVal(O2, "Enable default optimizations"),

clEnumVal(O3, "Enable expensive optimizations")));

...

if (OptimizationLevel >= O2) doPartialRedundancyElimination(...);...

该声明定义了OptimizationLevel“ OptLevel”枚举类型的变量“ ”。可以为该变量分配声明中列出的任何值。CommandLine 库强制用户只能指定其中一个选项，并确保只能指定有效的枚举值。“ clEnumVal”宏确保命令行参数与枚举值相匹配。添加此选项后，我们的帮助输出现在是：

USAGE: compiler [options] <input file>

OPTIONS:

Choose optimization level:

-g - No optimizations, enable debugging

-O1 - Enable trivial optimizations

-O2 - Enable default optimizations

-O3 - Enable expensive optimizations

-f - Enable binary output on terminals

-help - display available options (-help-hidden for more)

-o <filename> - Specify output filename

-quiet - Don't print informational messages

在这种情况下，标志名称直接对应于枚举名称有点尴尬，因为我们可能不希望g在我们的程序中使用名为“ ”的枚举定义。因此，我们也可以这样写这个例子：

enum OptLevel {

Debug, O1, O2, O3};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),

cl::values(

clEnumValN(Debug, "g", "No optimizations, enable debugging"),

clEnumVal(O1 , "Enable trivial optimizations"),

clEnumVal(O2 , "Enable default optimizations"),

clEnumVal(O3 , "Enable expensive optimizations")));

...

if (OptimizationLevel == Debug) outputDebugInfo(...);...

通过使用“ clEnumValN”宏而不是“ clEnumVal”，我们可以直接指定标志应该得到的名称。一般来说，直接映射很好，但有时您不能或不想保留映射，而这正是您要使用它的时候。

命名替代品

另一种有用的论证形式是命名的替代风格。我们将在我们的编译器中使用这种风格来指定可以使用的不同调试级别。我们希望支持以下选项，而不是每个调试级别都有自己的开关，一次只能指定其中一个：“ --debug-level=none”、“ --debug-level=quick”、“ --debug-level=detailed”。为此，我们使用与优化级别标志完全相同的格式，但我们还指定了一个选项名称。对于这种情况，代码如下所示：

enum DebugLev {

nodebuginfo, quick, detailed};

// Enable Debug Options to be specified on the command linecl::opt<DebugLev> DebugLevel("debug\_level", cl::desc("Set the debugging level:"),

cl::values(

clEnumValN(nodebuginfo, "none", "disable debug information"),

clEnumVal(quick, "enable quick debug information"),

clEnumVal(detailed, "enable detailed debug information")));

该定义定义了一个类型为“ ”的枚举命令行变量，其工作方式与之前完全相同。这里的区别只是暴露给程序用户的界面和“ ”选项的帮助输出：enum DebugLev-help

USAGE: compiler [options] <input file>

OPTIONS:

Choose optimization level:

-g - No optimizations, enable debugging

-O1 - Enable trivial optimizations

-O2 - Enable default optimizations

-O3 - Enable expensive optimizations

-debug\_level - Set the debugging level:

=none - disable debug information

=quick - enable quick debug information

=detailed - enable detailed debug information

-f - Enable binary output on terminals

-help - display available options (-help-hidden for more)

-o <filename> - Specify output filename

-quiet - Don't print informational messages

同样，调试级别声明和优化级别声明之间的唯一结构差异是调试级别声明包含一个选项名称 ( "debug\_level")，它会自动更改库处理参数的方式。CommandLine 库支持这两种形式，因此您可以选择最适合您的应用程序的形式。

解析选项列表

现在我们有了标准的普通参数类型，让我们来点疯狂的。假设我们希望我们的优化器接受要执行的优化列表，允许重复。例如，我们可能想要运行：“ ”。在这种情况下，参数的顺序和出现的次数非常重要。这就是“ ”模板的用途。首先，首先定义您要执行的优化的枚举：compiler -dce -instsimplify -inline -dce -stripcl::list

enum Opts {

// 'inline' is a C++ keyword, so name it 'inlining'

dce, instsimplify, inlining, strip};

然后定义你的“ cl::list”变量：

cl::list<Opts> OptimizationList(cl::desc("Available Optimizations:"),

cl::values(

clEnumVal(dce , "Dead Code Elimination"),

clEnumVal(instsimplify , "Instruction Simplification"),

clEnumValN(inlining, "inline", "Procedure Integration"),

clEnumVal(strip , "Strip Symbols")));

这定义了一个概念上类型为“”的变量。因此，您可以使用标准向量方法访问它：std::vector<enum Opts>

for (unsigned i = 0; i != OptimizationList.size(); ++i)

switch (OptimizationList[i])

...

…遍历指定的选项列表。

请注意，“ cl::list”模板是完全通用的，可以与可与“ ”模板一起使用的任何数据类型或其他参数一起使用cl::opt。使用列表的一种特别有用的方法是将所有位置参数一起捕获（如果可能指定了多个位置参数）。例如，在链接器的情况下，链接器需要几个 ' .o' 文件，并且需要将它们捕获到一个列表中。这自然指定为：

...cl::list<std::string> InputFilenames(cl::Positional, cl::desc("<Input files>"), cl::OneOrMore);...

这个变量就像一个“ vector<string>”对象。因此，访问列表很简单，就像上面一样。在这个例子中，我们使用 cl::OneOrMore修饰符来通知 CommandLine 库如果用户没有.o在我们的命令行上指定任何文件，这是一个错误。同样，这只是减少了我们必须做的检查量。

收集选项作为一组标志

除了在列表中收集选项集，还可以在位向量中收集枚举值的信息。cl::bits类使用的表示 是一个unsigned整数。枚举值由枚举的序数值位位置中的 0/1 表示。1 表示指定了枚举，否则为 0。在解析每个指定值时，结果枚举的位设置在选项的位向量中：

bits |= 1 << (unsigned)enum;

多次指定的选项是多余的。第一个之后的任何实例都将被丢弃。

修改上面的列表示例，我们可以用cl::bits替换cl::list：

cl::bits<Opts> OptimizationBits(cl::desc("Available Optimizations:"),

cl::values(

clEnumVal(dce , "Dead Code Elimination"),

clEnumVal(instsimplify , "Instruction Simplification"),

clEnumValN(inlining, "inline", "Procedure Integration"),

clEnumVal(strip , "Strip Symbols")));

要测试是否instsimplify已指定，我们可以使用以下cl:bits::isSet 函数：

if (OptimizationBits.isSet(instsimplify)) {

...}

也可以使用以下cl::bits::getBits 函数获取原始位向量：

unsigned bits = OptimizationBits.getBits();

最后，如果使用外部存储，则指定的位置必须是 类型 unsigned。在所有其他方面，cl::bits选项等同于 cl::list选项。

添加自由格式文本以帮助输出

随着程序的发展和成熟，我们可能会决定将有关其功能的摘要信息放入帮助输出中。帮助输出的样式类似于 Unixman页面，提供有关程序的简明信息。然而， Unixman页面通常会描述程序的作用。要将其添加到您的 CommandLine 程序中，只需将第三个参数传递给 main 中的cl::ParseCommandLineOptions调用即可。这个额外的参数然后作为程序的概览信息被打印出来，允许你包含任何你想要的额外信息。例如：

int main(int argc, char \*\*argv) {

cl::ParseCommandLineOptions(argc, argv, " CommandLine compiler example\n\n"

" This program blah blah blah...\n");

...}

将产生帮助输出：

\*\*OVERVIEW: CommandLine compiler example

This program blah blah blah...\*\*

USAGE: compiler [options] <input file>

OPTIONS:

...

-help - display available options (-help-hidden for more)

-o <filename> - Specify output filename

将选项分组

如果我们的程序有大量选项，我们工具的用户可能很难浏览-help. 为了缓解这个问题，我们可以将我们的选择分类。这可以通过声明选项类别（cl::OptionCategory对象）然后使用cl::cat选项属性将我们的选项放入这些类别中来完成。例如：

cl::OptionCategory StageSelectionCat("Stage Selection Options",

"These control which stages are run.");

cl::opt<bool> Preprocessor("E",cl::desc("Run preprocessor stage."),

cl::cat(StageSelectionCat));

cl::opt<bool> NoLink("c",cl::desc("Run all stages except linking."),

cl::cat(StageSelectionCat));

-help如果声明了选项类别，则输出将被分类。输出看起来像

OVERVIEW: This is a small program to demo the LLVM CommandLine APIUSAGE: Sample [options]

OPTIONS:

General options:

-help - Display available options (-help-hidden for more)

-help-list - Display list of available options (-help-list-hidden for more)

Stage Selection Options:

These control which stages are run.

-E - Run preprocessor stage.

-c - Run all stages except linking.

除了-help声明选项类别时更改的行为外，命令行选项-help-list变得可见，这会将命令行选项打印为未分类列表。

请注意，未明确分类的选项将放在 cl::getGeneralCategory()类别中。

参考指南

现在您已经了解了如何使用 CommandLine 库的基础知识，本节将为您提供调整命令行选项工作方式所需的详细信息，以及有关更多“高级”命令行选项处理功能的信息。

位置参数

位置参数是那些未命名且未使用连字符指定的参数。仅由其位置指定选项时，应使用位置参数。例如，标准 Unixgrep工具接受一个正则表达式参数和一个可选的要搜索的文件名（如果未指定文件名，则默认为标准输入）。使用 CommandLine 库，这将被指定为：

cl::opt<string> Regex (cl::Positional, cl::desc("<regular expression>"), cl::Required);cl::opt<string> Filename(cl::Positional, cl::desc("<input file>"), cl::init("-"));

鉴于这两个选项声明，-help我们的 grep 替换的输出将如下所示：

USAGE: spiffygrep [options] <regular expression> <input file>

OPTIONS:

-help - display available options (-help-hidden for more)

……生成的程序可以像标准grep 工具一样使用。

位置参数按其构造顺序排序。这意味着命令行选项将根据它们在 .cpp 文件中的列出方式进行排序，但如果位置参数在多个 .cpp 文件中定义，则不会定义排序。解决此问题的方法很简单，就是在一个 .cpp 文件中定义所有位置参数。

用连字符指定位置选项

有时您可能希望为位置参数指定一个以连字符开头的值（例如，在-foo文件中搜索 ' '）。起初，这样做会遇到麻烦，因为它会尝试查找名为 ' -foo' 的参数，但会失败（单引号也救不了你）。注意系统grep有同样的问题：

$ spiffygrep '-foo' test.txt

Unknown command line argument '-foo'. Try: spiffygrep -help'

$ grep '-foo' test.txt

grep: illegal option -- f

grep: illegal option -- o

grep: illegal option -- o

Usage: grep -hblcnsviw pattern file . . .

此问题的解决方案对于您的工具和系统版本都是相同的：使用 ' --' 标记。当用户--在命令行上指定 ' ' 时，它告诉程序 ' --' 之后的所有选项都应被视为位置参数，而不是选项。因此，我们可以这样使用它：

$ spiffygrep -- -foo test.txt

...output...

使用 getPosition() 确定绝对位置

有时一个选项会影响或修改另一个选项的含义。例如，考虑gcc的选项。这告诉忽略后续位置参数的后缀，并强制将文件解释为好像它包含 language 中的源代码。为了正确处理这个问题，您需要知道每个参数的绝对位置，尤其是列表中的参数，以便正确应用它们的交互。这对于选项也很有用，例如which 实际上是一个以破折号开头的位置参数。-x LANGgccLANG-llibname

所以，一般来说，问题是你有两个cl::list以某种方式相互作用的变量。为确保正确交互，您可以使用该 cl::list::getPosition(optnum)方法。此方法返回optnum项目在 cl::list.

用法的成语是这样的：

static cl::list<std::string> Files(cl::Positional, cl::OneOrMore);static cl::list<std::string> Libraries("l");

int main(int argc, char\*\*argv) {

// ...

std::vector<std::string>::iterator fileIt = Files.begin();

std::vector<std::string>::iterator libIt = Libraries.begin();

unsigned libPos = 0, filePos = 0;

while ( 1 ) {

if ( libIt != Libraries.end() )

libPos = Libraries.getPosition( libIt - Libraries.begin() );

else

libPos = 0;

if ( fileIt != Files.end() )

filePos = Files.getPosition( fileIt - Files.begin() );

else

filePos = 0;

if ( filePos != 0 && (libPos == 0 || filePos < libPos) ) {

// Source File Is next

++fileIt;

}

else if ( libPos != 0 && (filePos == 0 || libPos < filePos) ) {

// Library is next

++libIt;

}

else

break; // we're done with the list

}}

请注意，出于兼容性原因，cl::opt还支持 将提供该选项的绝对位置的选项。您可以使用与上面相同的方法来使用一个和一个 选项，就像您可以使用两个列表一样。unsigned getPosition()cl::optcl::list

cl::ConsumeAfter修饰符\_

cl::ConsumeAfter 格式化选项用于构造使用“解释器样式”选项处理的程序。使用这种选项处理方式，在最后一个位置参数之后指定的所有参数都被视为特殊的解释器参数，不被命令行参数解释。

作为一个具体的例子，假设我们正在开发标准 Unix Bourne shell ( /bin/sh) 的替代品。要运行/bin/sh，首先要为 shell 本身指定选项（比如-x打开跟踪输出），然后指定要运行的脚本的名称，然后为脚本指定参数。脚本的这些参数由 Bourne shell 命令行选项处理器解析，但不会解释为 shell 本身的选项。使用 CommandLine 库，我们将其指定为：

cl::opt<string> Script(cl::Positional, cl::desc("<input script>"), cl::init("-"));cl::list<string> Argv(cl::ConsumeAfter, cl::desc("<program arguments>..."));cl::opt<bool> Trace("x", cl::desc("Enable trace output"));

它自动提供帮助输出：

USAGE: spiffysh [options] <input script> <program arguments>...

OPTIONS:

-help - display available options (-help-hidden for more)

-x - Enable trace output

在运行时，如果我们将新的 shell 替换运行为'，则变量将设置为 true，变量将设置为“ ”，并且列表将包含，因为它们是在最后一个位置参数之后指定的（这是脚本名称）。`spiffysh -x test.sh -a -x -y barTraceScripttest.shArgv["-a", "-x", "-y", "bar"]

cl::ConsumeAfter何时可以指定选项有几个限制。例如，cl::ConsumeAfter每个程序只能指定一个，必须至少指定一个位置参数，不能有任何cl::list位置参数，cl::ConsumeAfter选项应该是cl::list选项。

内部与外部存储

默认情况下，所有命令行选项都会自动保存它们从命令行解析的值。这在常见情况下非常方便，尤其是结合在使用它们的文件中定义命令行选项的能力时。这称为内部存储模型。

然而，有时将命令行选项处理代码与解析值的存储区分开会很好。例如，假设我们有一个 ' -debug' 选项，我们希望使用该选项在整个程序主体中启用调试信息。在这种情况下，控制调试代码的布尔值应该是全局可访问的（例如，在头文件中），但命令行选项处理代码不应暴露给所有这些客户端（需要大量的 .cpp 文件）。#include CommandLine.h

为此，请使用您的选项设置 .h 文件，例如：

// DebugFlag.h - Get access to the '-debug' command line option//

// DebugFlag - This boolean is set to true if the '-debug' command line option// is specified. This should probably not be referenced directly, instead, use// the DEBUG macro below.//extern bool DebugFlag;

// DEBUG macro - This macro should be used by code to emit debug information.// In the '-debug' option is specified on the command line, and if this is a// debug build, then the code specified as the option to the macro will be// executed. Otherwise it will not be.#ifdef NDEBUG#define LLVM\_DEBUG(X)#else#define LLVM\_DEBUG(X) do { if (DebugFlag) { X; } } while (0)#endif

这允许客户愉快地使用LLVM\_DEBUG()宏，或者 DebugFlag如果他们愿意的话可以显式地使用。现在我们只需要能够在设置DebugFlag选项时设置布尔值。为此，我们将一个附加参数传递给命令行参数处理器，并指定用cl::location属性填充的位置：

bool DebugFlag; // the actual valuestatic cl::opt<bool, true> // The parserDebug("debug", cl::desc("Enable debug output"), cl::Hidden, cl::location(DebugFlag));

在上面的示例中，我们将“ true”指定为 cl::opt模板的第二个参数，表示模板不应维护值本身的副本。除此之外，我们还指定了cl::location 属性，以便DebugFlag自动设置。

选项属性

本节介绍您可以在选项上指定的基本属性。



选项名称属性（所有选项都需要， 位置选项除外）指定选项名称是什么。此选项用简单的双引号指定：



cl::opt<bool> Quiet("quiet");



cl::desc属性指定要-help在程序输出中显示的选项的描述。此属性支持多行描述，每行以“n”分隔。

cl::value\_desc属性指定一个字符串，可用于微调-help命令行选项的输出。在这里看一个例子。



cl::init属性指定标量 选项的初始值。如果未指定此属性，则命令行选项值默认为该类型的默认构造函数创建的值。



警告



如果您同时为选项指定cl::init和cl::location ，则必须首先指定cl::location，以便当命令行解析器看到cl::init时，它知道将初始值放在哪里。（如果你没有按正确的顺序放置它们，你将在运行时得到一个错误。）



如果使用外部存储，cl::location属性用于存储已解析命令行选项的值。有关详细信息，请参阅内部与外部存储部分。

cl::aliasopt属性指定cl::alias选项是哪个选项的别名。



cl::values属性指定通用解析器使用的字符串到值的映射。它采用（选项、值、描述）三元组列表，这些三元组指定选项名称、映射到的值以及-help工具中显示的描述。因为通用解析器最常与枚举值一起使用，所以两个宏通常很有用：



1.clEnumVal宏用作为枚举指定三元组的一种非常简单的方法。此宏自动使选项名称与枚举名称相同。宏的第一个选项是枚举，第二个是命令行选项的描述。

2.clEnumValN宏用于指定选项名称不等于枚举名称的宏选项。对于这个宏，第一个参数是枚举值，第二个是标志名，第二个是描述。

如果您尝试将 cl::values 与不支持它的解析器一起使用，则会出现编译时错误。

cl::multi\_val属性指定此选项具有多个值（例如：）。该属性采用一个无符号参数——选项值的数量。此属性仅对选项有效（如果您尝试将其与其他选项类型一起使用，将会失败并出现编译错误）。允许在多值选项上使用所有常用修饰符（ 显然除了 ）。-sectalign segname sectname sectvaluecl::listcl::ValueDisallowed

cl::cat属性指定选项所属的选项类别。该类别应该是一个cl::OptionCategory对象。



cl::callback属性指定一个回调函数，当看到一个选项时调用该函数，并可用于设置其他选项，因为在选项 B 中暗示选项 A。如果选项是cl::list，并且cl::还指定了CommaSeparated，回调将为每个值触发一次。这可用于验证组合或有选择地设置其他选项。



cl::opt<bool> OptA("a", cl::desc("option a"));cl::opt<bool> OptB(

"b", cl::desc("option b -- This option turns on option a"),

cl::callback([&](const bool &) { OptA = true; }));cl::list<std::string, cl::list<std::string>> List(

"list",

cl::desc("option list -- This option turns on options a when "

"'foo' is included in list"),

cl::CommaSeparated,

cl::callback([&](const std::string &Str) {

if (Str == "foo")

OptA = true;

}));



选项修饰符

选项修饰符是您传递给cl::opt和cl::list的构造函数的标志和表达式。这些修饰符使您能够调整选项的解析方式以及-help输出的生成方式以很好地适应您的应用程序。

这些选项分为五个主要类别：

1.-help从输出中隐藏选项

2.控制要求和允许的出现次数

3.控制是否必须指定一个值

4.控制其他格式化选项

5.杂项选项修饰符

不可能将来自同一类别的两个选项（您将收到运行时错误）指定为单个选项，杂项类别中的选项除外。CommandLine 库为所有这些设置指定了默认值，这些设置在实践中最有用，也是最常见的，这意味着您通常不必担心这些。

-help从输出中隐藏一个选项

、cl::NotHidden和修饰符用于控制选项是否出现在编译程序的和 cl::Hidden输出中：cl::ReallyHidden-help-help-hidden

cl::NotHidden修饰符（它是cl::opt和 cl::list选项的默认值）表示该选项将出现在两个帮助列表中。

cl::Hidden修饰符（这是cl::alias选项的默认值）指示该选项不应出现在-help输出中，但应该出现在-help-hidden输出中。

cl::ReallyHidden修饰符指示该选项不应出现在任何帮助输出中。

控制要求和允许的出现次数

这组选项用于控制允许（或要求）在程序的命令行上指定选项的次数。为此设置指定一个值允许 CommandLine 库为您进行错误检查。

此选项组的允许值为：

cl::Optional修饰符（这是cl::opt和 cl::alias类的默认值）表示您的程序将允许指定选项的出现次数为零次或一次。

cl::ZeroOrMore修饰符（这是cl::list 类的默认值）表示您的程序将允许该选项被指定零次或多次。

cl::Required修饰符表示指定的选项必须精确指定一次。

cl::OneOrMore修饰符表示该选项必须至少指定一次。

位置参数部分描述了cl ::ConsumeAfter修饰符。

如果未指定选项，则选项的值等于cl::init属性指定的值。如果cl::init未指定属性，则使用数据类型的默认构造函数初始化选项值。

如果多次为cl::opt类的选项指定一个选项，则只会保留最后一个值。

控制是否必须指定一个值

这组选项用于控制选项是否允许存在值。在 CommandLine 库的情况下，值可以用等号（例如“ -index-depth=17”）指定，也可以指定为尾随字符串（例如“ ”）。-o a.out

此选项组的允许值为：

cl::ValueOptional修饰符（这是bool类型化选项的默认值）指定是否可以接受值。布尔参数可以通过出现在命令行上来启用，或者它可以有一个显式的 ' -foo=true'。如果使用此模式指定选项，则提供的值不带等号是非法的。因此 ' ' 是非法的。要获得此行为，您必须使用cl::ValueRequired修饰符。-foo true

cl::ValueRequired修饰符（这是所有其他类型的默认值，除了使用通用解析器的未命名替代项）指定必须提供一个值。此模式通知命令行库，如果选项未提供等号，则提供的下一个参数必须是值。这允许诸如“ ”之类的东西起作用。-o a.out

cl::ValueDisallowed修饰符（这是使用通用解析器的未命名替代项的默认值）指示用户指定值是运行时错误。可以提供此选项以禁止用户向布尔选项（如“ -foo=true”）提供选项。

通常，此选项组的默认值就像您希望的那样工作。如上所述，您可以将cl::ValueDisallowed 修饰符指定为布尔参数以限制您的命令行解析器。这些选项在扩展库时最有用。

控制其他格式化选项

格式化选项组用于指定命令行选项具有特殊能力，并且与其他命令行参数不同。像往常一样，您最多只能指定这些参数之一。

cl::NormalFormatting修饰符（默认所有选项）指定此选项为“正常”。

cl::Positional修饰符指定这是一个位置参数，没有与之关联的命令行选项。有关详细信息，请参阅 位置参数部分。

cl::ConsumeAfter修饰符指定此选项用于捕获“解释器样式”参数。有关详细信息，请参阅本节。

cl::Prefix修饰符指定此选项作为其值的前缀。对于“前缀”选项，等号不会将值与指定的选项名称分开。相反，该值是前缀之后的所有内容，包括任何等号（如果存在）。这对于处理奇数参数很有用，例如-lmalloc在-L/usr/lib链接器工具或 -DNAME=value编译器工具中。此处，' l'、' D' 和 ' L' 选项是普通字符串（或列表）选项，添加了cl::Prefix 修饰符以允许 CommandLine 库识别它们。请注意， cl::Prefix选项不得指定cl::ValueDisallowed修饰符。

控制选项分组

cl::Grouping修饰符可以与除cl::Positional之外的任何格式化类型组合。它用于实现 Unix 风格的工具（如ls），这些工具有很多单字母参数，但只需要一个破折号。例如，' ' 命令实际上启用了四个不同的选项，所有选项都是单个字母。ls -labF

请注意，cl::Grouping选项只有在单独使用或在组末尾使用时才能具有值。对于cl::ValueRequired，如果在组中的其他地方使用了这样的选项，则会出现运行时错误。

CommandLine 库不限制您如何使用cl::Prefix或 cl::Grouping修饰符，但可以指定不明确的参数设置。因此，可以有多个作为前缀或分组选项的字母选项，并且它们仍将按设计工作。

为此，CommandLine 库使用贪婪算法将输入选项解析为（可能多个）前缀和分组选项。该策略基本上是这样的：

parse(string OrigInput) {

1. string Input = OrigInput;

2. if (isOption(Input)) return getOption(Input).parse(); // Normal option

3. while (!Input.empty() && !isOption(Input)) Input.pop\_back(); // Remove the last letter

4. while (!Input.empty()) {

string MaybeValue = OrigInput.substr(Input.length())

if (getOption(Input).isPrefix())

return getOption(Input).parse(MaybeValue)

if (!MaybeValue.empty() && MaybeValue[0] == '=')

return getOption(Input).parse(MaybeValue.substr(1))

if (!getOption(Input).isGrouping())

return error()

getOption(Input).parse()

Input = OrigInput = MaybeValue

while (!Input.empty() && !isOption(Input)) Input.pop\_back();

if (!Input.empty() && !getOption(Input).isGrouping())

return error()

}

5. if (!OrigInput.empty()) error();

}

杂项选项修饰符

杂项选项修饰符是唯一可以从集合中指定多个标志的标志：它们不是相互排斥的。这些标志指定修改选项的布尔属性。

cl::CommaSeparated修饰符指示为选项值指定的任何逗号都应该用于将值拆分为选项的多个值。例如，这两个选项在 cl::CommaSeparated指定时是等价的：“ ”和“ ”。此选项仅在允许选项接受一个或多个值（即它是cl::list 选项）的情况下才有意义。-foo=a -foo=b -foo=c-foo=a,b,c

cl::DefaultOption修饰符用于指定该选项是默认值，可以被特定于应用程序的解析器覆盖。例如，-help别名-h以这种方式注册，因此它可以被需要将该-h选项用于其他目的的应用程序覆盖，无论是作为常规选项还是其他选项的别名。

cl::PositionalEatsArgs修饰符（仅适用于位置参数，并且仅对列表有意义）指示位置参数应该使用它之后的任何字符串（包括以“-”开头的字符串）直到另一个可识别的位置参数。例如，如果您有两个“eating”位置参数，“ pos1”和“ pos2”，字符串“ ”将导致“ ”字符串应用于“ ”选项，“ ”字符串应用于“ ”选项.-pos1 -foo -bar baz -pos2 -bork-foo -bar -baz-pos1-bork-pos2

cl::Sink修饰符用于处理未知选项。如果至少有一个选项指定了cl::Sink修饰符，解析器会将无法识别的选项字符串作为值传递给它，而不是发出错误信号。与 一样，此修饰符仅对cl::list 选项cl::CommaSeparated有意义。

响应文件

一些系统，例如 Microsoft Windows 的某些变体和一些较旧的 Unices 对命令行长度的限制相对较低。因此，习惯上使用所谓的“响应文件”来规避此限制。这些文件在命令行（使用“@file”）语法中提到。该程序读取这些文件并将内容插入到 argv 中，从而绕过命令行长度限制。

顶级类和函数

尽管有所有内置的灵活性，CommandLine 选项库实际上只包含一个函数cl::ParseCommandLineOptions和三个主要类：cl::opt、cl::list和cl::alias。本节详细介绍这三个类。

cl::getRegisteredOptions功能\_

该cl::getRegisteredOptions函数旨在让程序员能够访问已声明的非位置命令行选项，以便可以在调用cl::ParseCommandLineOptions-help之前修改它们的显示方式。请注意，不应在任何静态初始化期间调用此方法，因为不能保证所有选项都已初始化。因此它应该从.main

此函数可用于访问工具编写者可能无法直接访问的库中声明的选项。

该函数检索一个StringMap，它将选项字符串（例如-help）映射到一个Option\*.

以下是如何使用该函数的示例：

using namespace llvm;int main(int argc, char \*\*argv) {

cl::OptionCategory AnotherCategory("Some options");

StringMap<cl::Option\*> &Map = cl::getRegisteredOptions();

//Unhide useful option and put it in a different category

assert(Map.count("print-all-options") > 0);

Map["print-all-options"]->setHiddenFlag(cl::NotHidden);

Map["print-all-options"]->setCategory(AnotherCategory);

//Hide an option we don't want to see

assert(Map.count("enable-no-infs-fp-math") > 0);

Map["enable-no-infs-fp-math"]->setHiddenFlag(cl::Hidden);

//Change --version to --show-version

assert(Map.count("version") > 0);

Map["version"]->setArgStr("show-version");

//Change --help description

assert(Map.count("help") > 0);

Map["help"]->setDescription("Shows help");

cl::ParseCommandLineOptions(argc, argv, "This is a small program to demo the LLVM CommandLine API");

...}

cl::ParseCommandLineOptions功能\_

该cl::ParseCommandLineOptions函数设计为直接从 调用main，用于一次性填写所有命令行选项变量的值，argc并且argv可用。

该cl::ParseCommandLineOptions函数需要两个参数（argc 和argv），但也可以采用可选的第三个参数，该参数包含 调用该选项时要发出的额外文本。-help

cl::SetVersionPrinter功能\_

该cl::SetVersionPrinter函数旨在直接从 main和之前 cl::ParseCommandLineOptions调用。它的使用是可选的。它只是安排调用一个函数来响应--version 选项，而不是让CommandLine库打印出 LLVM 的常用版本字符串。这对于不属于 LLVM 但希望使用这些CommandLine工具的程序很有用。这样的程序应该只定义一个小函数，它不接受任何参数并返回void，并打印出适合该程序的任何版本信息。将该函数的地址传递给以安排在用户给出选项cl::SetVersionPrinter时调用它。--version

cl::opt班级\_

cl::opt类是用来表示标量命令行选项的类，也是最常使用的类。它是一个模板类，最多可以接受三个参数（除了第一个参数外，所有参数都有默认值）：

namespace cl {

template <class DataType, bool ExternalStorage = false,

class ParserClass = parser<DataType> >

class opt;}

第一个模板参数指定命令行参数的基础数据类型，并用于选择默认的解析器实现。第二个模板参数用于指定选项是否应包含选项的存储（默认值）或是否应使用外部存储来包含为选项解析的值（有关更多信息，请参阅内部与外部存储 ）。

第三个模板参数指定要使用的解析器。parser默认值根据选项的基础数据类型选择类的实例化。通常，此默认值适用于大多数应用程序，因此此选项仅在使用自定义解析器时使用。

cl::list班级\_

该类cl::list是用于表示命令行选项列表的类。它也是一个模板类，最多可以接受三个参数：

namespace cl {

template <class DataType, class Storage = bool,

class ParserClass = parser<DataType> >

class list;}

此类与cl::opt类的工作方式完全相同，除了第二个参数是外部存储的类型，而不是布尔值。对于此类，标记类型“ bool”用于指示应使用内部存储。

cl::bits班级\_

cl::bits类是用于以位向量的形式表示命令行选项列表的类。它也是一个模板类，最多可以接受三个参数：

namespace cl {

template <class DataType, class Storage = bool,

class ParserClass = parser<DataType> >

class bits;}

此类的工作方式与cl::list类完全相同，除了如果使用外部存储，第二个参数必须是类型。 unsigned

cl::alias班级\_

该类cl::alias是一个非模板类，用于为其他参数形成别名。

namespace cl {

class alias;}

cl::aliasopt属性应该用于指定这是哪个选项的别名。别名参数默认为cl::Hidden，并使用别名选项解析器进行从字符串到数据的转换。

cl::extrahelp班级\_

该类cl::extrahelp是一个非模板化类，它允许为该-help选项打印出额外的帮助文本。

namespace cl {

struct extrahelp;}

要使用 extrahelp，只需构造一个带有构造函数参数的帮助。传递给构造函数的文本将逐字打印在帮助消息的底部。请注意，可以使用多个，但不鼓励这种做法。如果您的工具需要打印额外的帮助信息，请将所有帮助信息放入一个 实例中。const char\*cl::extrahelp cl::extrahelp

例如：

cl::extrahelp("\nADDITIONAL HELP:\n\n This is the extra help\n");

cl::OptionCategory班级\_

该类cl::OptionCategory是一个用于声明选项类别的简单类。

namespace cl {

class OptionCategory;}

选项类别必须有一个名称和一个可选的描述，这些描述将作为 传递给构造函数。const char\*

请注意，在解析选项（例如静态）之前声明选项类别并将其与选项相关联会将输出-help从未分类更改为已分类。如果声明了选项类别但未与选项关联，则它将在 的输出中隐藏，-help 但会显示在 的输出中-help-hidden。

内置解析器

解析器控制如何将从命令行获取的字符串值转换为适合在 C++ 程序中使用的类型化值。默认情况下，parser<type>如果命令行选项指定它使用“ type”类型的值，则 CommandLine 库使用 的实例。因此，自定义选项处理是使用“ parser”类的特化指定的。

CommandLine 库提供了以下内置的解析器特化，这对大多数应用程序来说已经足够了。但是，它也可以扩展为使用新的数据类型和解释相同数据的新方法。有关此类库扩展的更多详细信息，请参阅 编写自定义解析器。

通过使用指定映射信息的cl::values属性，通用parser<t>解析器可用于将字符串值映射到任何数据类型。此解析器最常见的用途是解析枚举值，它允许您使用 CommandLine 库进行所有错误检查，以确保仅指定有效的枚举值（而不是接受任意字符串）。然而，尽管如此，通用解析器类可用于任何数据类型。

parser<bool> 特化用于将布尔字符串转换为布尔值。当前接受的字符串是“ true”、“ TRUE”、“ True”、“ 1”、“ false”、“ FALSE”、“ False”和“ 0”。

parser<boolOrDefault> 特化用于值为布尔值的情况，但我们还需要知道该选项是否已指定。boolOrDefault 是一个具有 3 个值的枚举，BOU\_UNSET、BOU\_TRUE 和 BOU\_FALSE。这个解析器接受与``parser<bool>``相同的字符串。

parser<string> 特化只是将解析后的字符串存储到指定的字符串值中。不执行数据的转换或修改。

parser<int> 特化使用 Cstrtol函数来解析字符串输入。因此，它将接受必须以非零数字开头的十进制数（带有可选的“+”或“-”前缀）。它接受八进制数（用“ ' 0”前缀数字标识）和十六进制数（用“ ' 0x”或“ 0X'”作为前缀）。

parser<double>和parser<float> 特化使用标准 Cstrtod函数将浮点字符串转换为浮点值。因此，支持范围广泛的字符串格式，包括指数表示法（例如：）1.7e15并正确支持语言环境。

扩展指南

尽管 CommandLine 库已经内置了许多功能（如前所述），但其真正的优势之一在于其可扩展性。本节讨论 CommandLine 库如何在幕后工作，并说明如何进行一些简单、常见的扩展。

编写自定义解析器

最简单和最常见的扩展之一是使用自定义解析器。如前所述，解析器是 CommandLine 库的一部分，它将用户的字符串输入转换为特定的已解析数据类型，并在此过程中验证输入。

有两种方法可以使用新的解析器：

1.

为您的自定义数据类型专门化cl::parser模板。

2.

这种方法的优点是，只要您的自定义数据类型的用户使用您的数据类型的值类型定义选项，他们就会自动使用您的自定义解析器。这种方法的缺点是，如果您的基本数据类型已经受支持，它就不起作用。

3.

4.

编写一个独立的类，从需要它的选项中明确使用它。

5.

这种方法在您需要使用特殊语法为不是很特殊的数据类型解析选项的情况下效果很好。这种方法的缺点是您的解析器的用户必须知道他们正在使用您的解析器而不是内置的解析器。

6.

为了指导讨论，我们将讨论一个接受文件大小的自定义解析器，在数字大小后使用可选单位指定。例如，我们想将“102kb”、“41M”、“1G”解析为合适的整数值。在这种情况下，我们要解析的基础数据类型是“ unsigned”。我们选择上面的方法 #2，因为我们不想将其设为所有 unsigned选项的默认值。

首先，我们声明我们的新FileSizeParser类：

struct FileSizeParser : public cl::parser<unsigned> {

// parse - Return true on error.

bool parse(cl::Option &O, StringRef ArgName, const std::string &ArgValue,

unsigned &Val);};

我们的新类继承自cl::parser模板类，为我们填充默认的样板代码。我们给它我们解析成的数据类型，parse方法的最后一个参数，这样我们自定义解析器的客户端就知道要传递给解析方法的对象类型。（这里我们声明我们解析成 ' unsigned' 变量。）

对于大多数用途，必须在自定义解析器中实现的唯一方法是parse方法。parse每当调用选项时都会调用该方法，传入选项本身、选项名称、要解析的字符串以及对返回值的引用。如果要解析的字符串格式不正确，解析器应输出错误消息并返回 true。否则它应该返回 false 并将 ' Val' 设置为解析后的值。在我们的示例中，我们实现parse为：

bool FileSizeParser::parse(cl::Option &O, StringRef ArgName,

const std::string &Arg, unsigned &Val) {

const char \*ArgStart = Arg.c\_str();

char \*End;

// Parse integer part, leaving 'End' pointing to the first non-integer char

Val = (unsigned)strtol(ArgStart, &End, 0);

while (1) {

switch (\*End++) {

case 0: return false; // No error

case 'i': // Ignore the 'i' in KiB if people use that

case 'b': case 'B': // Ignore B suffix

break;

case 'g': case 'G': Val \*= 1024\*1024\*1024; break;

case 'm': case 'M': Val \*= 1024\*1024; break;

case 'k': case 'K': Val \*= 1024; break;

default:

// Print an error message if unrecognized character!

return O.error("'" + Arg + "' value invalid for file size argument!");

}

}}

这个函数为我们感兴趣的字符串类型实现了一个非常简单的解析器。虽然它有一些漏洞（例如它允许“ 123KKK”），但对于这个例子来说已经足够好了。请注意，我们使用该选项本身来打印错误消息（该error方法始终返回 true），以便获得一个很好的错误消息（如下所示）。现在我们有了解析器类，我们可以像这样使用它：

static cl::opt<unsigned, false, FileSizeParser>MFS("max-file-size", cl::desc("Maximum file size to accept"),

cl::value\_desc("size"));

将其添加到我们程序的输出中：

OPTIONS:

-help - display available options (-help-hidden for more)

...

-max-file-size=<size> - Maximum file size to accept

我们现在可以测试我们的解析是否正常工作（测试程序只打印出 max-file-size 参数值）：

$ ./test

MFS: 0

$ ./test -max-file-size=123MB

MFS: 128974848

$ ./test -max-file-size=3G

MFS: 3221225472

$ ./test -max-file-size=dog

-max-file-size option: 'dog' value invalid for file size argument!

看起来行得通。我们收到的错误消息很好而且很有帮助，而且我们似乎可以接受合理的文件大小。“自定义解析器”教程到此结束。

利用外部存储

几个 LLVM 库定义了静态cl::opt实例，这些实例将自动包含在与该库链接的任何程序中。这是一个特点。然而，有时需要在库之外知道命令行选项的值。在这些情况下，图书馆确实或应该提供图书馆用户可以访问的外部存储位置。这方面的示例包括文件导出和文件llvm::DebugFlag导出 的标志。lib/Support/Debug.cppllvm::TimePassesIsEnabledlib/IR/PassManager.cpp

动态添加命令行选项

# 扩展 LLVM：添加指令、内在函数、类型等

简介和警告

在使用 LLVM 的过程中，您可能希望为您的研究项目或实验定制它。此时，您可能意识到需要向 LLVM 添加一些内容，无论是新的基本类型、新的内部函数还是全新的指令。

当你意识到这一点时，停下来想一想。你真的需要扩展 LLVM 吗？它是 LLVM 在其当前版本中不支持的新的基本功能，还是可以从已经存在的 LLVM 元素中合成？如果您不确定，请在LLVM 论坛上提问。原因是扩展 LLVM 会涉及到，因为你需要更新你打算在扩展中使用的所有不同的传递，并且有manyLLVM 分析和转换，所以它可能需要相当多的工作。

添加内在函数比添加指令容易得多，并且对优化过程是透明的。如果您添加的功能可以表示为函数调用，则内部函数是 LLVM 扩展的首选方法。

在您投入大量精力进行重要的扩展之前， 请在列表中询问您想要做的事情是否可以使用现有的基础设施来完成，或者其他人是否已经在处理它。这样做会为自己节省大量时间和精力。

添加一个新的内部函数

向 LLVM 添加新的内部函数比添加新指令要容易得多。几乎所有对 LLVM 的扩展都应该从一个内部函数开始，然后在必要时变成一条指令。

1.

llvm/docs/LangRef.html:

2.

记录内在的。确定它是否特定于代码生成器以及限制是什么。与其他人讨论这件事，以确保这是个好主意。

3.

4.

llvm/include/llvm/IR/Intrinsics\*.td:

5.

为你的内在添加一个条目。描述其内存访问特性以进行优化（这控制着它是否会被 DCE、CSE 等）。如果任何参数需要立即数，则必须使用 ImmArg 属性指示这些参数。请注意，任何使用其中一种llvm\_any\*\_ty类型作为参数或返回类型的内在函数都将被视为tblgen重载，并且内在函数的名称上需要相应的后缀。

6.

7.

llvm/lib/Analysis/ConstantFolding.cpp:

8.

如果可以不断折叠您的内在函数，请在 canConstantFoldCallTo和ConstantFoldCall函数中添加对它的支持。

9.

10.

llvm/test/\*:

11.

将测试用例的测试用例添加到测试套件

12.

将内在函数添加到系统后，您必须为其添加代码生成器支持。通常您必须执行以下步骤：

为您在 .td 中选择的目标添加对 .td 文件的支持 lib/Target/\*/\*.td。

这通常是向 .td 文件添加一个与内在匹配的模式的问题，尽管它可能显然还需要添加您想要生成的指令。PowerPC 和 X86 后端中有很多示例可供参考。

添加一个新的 SelectionDAG 节点

与内在函数一样，向 LLVM 添加新的 SelectionDAG 节点比添加新指令要容易得多。通常会添加新节点以帮助表示对许多目标通用的指令。这些节点通常映射到 LLVM 指令（添加、子）或内部（字节交换、人口计数）。在其他情况下，已添加新节点以允许许多目标执行共同任务（在浮点和整数表示之间转换）或在单个节点中捕获更复杂的行为（旋转）。

1.

include/llvm/CodeGen/ISDOpcodes.h:

2.

为新的 SelectionDAG 节点添加枚举值。

3.

4.

lib/CodeGen/SelectionDAG/SelectionDAG.cpp:

5.

添加代码以将节点打印到getOperationName. 如果你的新节点可以

6.

7.

在给定常量参数（例如常量与另一个常量相加）时在编译时求值，找到getNode采用适当数量参数的方法，并将节点的 case 添加到 switch 语句，该语句对节点执行常量折叠采用与新节点相同数量的参数。

8.

9.

10.

lib/CodeGen/SelectionDAG/LegalizeDAG.cpp:

11.

根据需要添加代码以合法化、提升和扩展节点。至少，您需要为您的节点添加一个 case 语句，在 LegalizeOp该语句中对节点的操作数调用 LegalizeOp，如果任何操作数因合法化而发生更改，则返回一个新节点。很可能并非 SelectionDAG 框架支持的所有目标都会原生支持新节点。在这种情况下，您还必须在节点的 case 语句中添加代码，以将节点LegalizeOp扩展为更简单、合法的操作。将ISD::UREM余数扩展为除法、乘法和减法的情况就是一个很好的例子。

12.

13.

lib/CodeGen/SelectionDAG/LegalizeDAG.cpp:

14.

如果目标可能支持仅以特定大小添加新节点，则您

15.

16.

还需要将代码添加到您的节点的 case 语句中，LegalizeOp 以将您的节点的操作数提升到更大的大小，并执行正确的操作。您还需要添加代码来PromoteOp执行此操作。举一个很好的例子，请参阅ISD::BSWAP，它将其操作数提升到更大的大小，执行字节交换，然后将正确的字节右移以在更宽的类型中模拟更窄的字节交换。

17.

18.

19.

lib/CodeGen/SelectionDAG/LegalizeDAG.cpp:

20.

为您的节点添加一个案例，ExpandOp以教导合法化器如何对已拆分为高低两半的值执行新节点表示的操作。这种情况将用于支持您的节点在 32 位目标上使用 64 位操作数。

21.

22.

lib/CodeGen/SelectionDAG/DAGCombiner.cpp:

23.

如果您的节点可以与自身或其他现有节点以类似窥视孔的方式组合，请为其添加一个访问函数，并从中调用该函数。您可以做几个简单组合的好例子； visitFABS并且visitSRL是很好的起点。

24.

25.

lib/Target/PowerPC/PPCISelLowering.cpp:

26.

每个目标都有一个TargetLowering类的实现，通常在它自己的文件中（尽管有些目标将它包含在与 DAGToDAGISel 相同的文件中）。目标的默认行为是假设您的新节点对于该目标合法的所有类型都是合法的。如果此目标本身不支持您的节点，则告诉目标升级它（如果它支持更大的类型）或扩展它。这将导致您在LegalizeOp上面编写的代码将您的新节点分解为该目标的其他合法节点。

27.

28.

include/llvm/Target/TargetSelectionDAG.td:

29.

LLVM 支持的大多数当前目标使用 DAGToDAG 方法生成代码，其中 SelectionDAG 节点与特定于目标的节点进行模式匹配，代表单个指令。为了使目标将指令与您的新节点相匹配，您必须将该节点的 def 添加到该文件的列表中，并具有适当的类型约束。查看 add、bswap和fadd例如。

30.

31.

lib/Target/PowerPC/PPCInstrInfo.td:

32.

每个目标都有一个描述目标指令集的 tablegen 文件。对于使用 DAGToDAG 指令选择框架的目标，为使用一个或多个目标节点的新节点添加模式。目前这方面的文档有点稀疏，但有几个不错的例子。查看rotlin的模式PPCInstrInfo.td。

33.

34.

TODO：记录复杂的模式。

35.

36.

llvm/test/CodeGen/\*:

37.

将新节点的测试用例添加到测试套件中。 llvm/test/CodeGen/X86/bswap.ll是一个很好的例子。

38.

添加新指令

警告

添加指令会改变位码格式，并且需要一些努力来保持与以前版本的兼容性。仅在绝对必要时才添加说明。

1.

llvm/include/llvm/IR/Instruction.def:

2.

为您的指令添加一个数字和一个枚举名称

3.

4.

llvm/include/llvm/IR/Instructions.h:

5.

为将代表您的指令的类添加定义

6.

7.

llvm/include/llvm/IR/InstVisitor.h:

8.

为访问者添加新指令类型的原型

9.

10.

llvm/lib/AsmParser/LLLexer.cpp:

11.

添加一个新令牌以从汇编文本文件中解析您的指令

12.

13.

llvm/lib/AsmParser/LLParser.cpp:

14.

添加有关如何阅读指令以及结果将构造什么的语法

15.

16.

llvm/lib/Bitcode/Reader/BitcodeReader.cpp:

17.

为您的指令添加一个案例以及如何从位码中解析它

18.

19.

llvm/lib/Bitcode/Writer/BitcodeWriter.cpp:

20.

为您的指令添加一个案例以及如何从位码中解析它

21.

22.

llvm/lib/IR/Instruction.cpp:

23.

添加一个案例，说明您的指令将如何打印到程序集

24.

25.

llvm/lib/IR/Instructions.cpp:

26.

实现你定义的类llvm/include/llvm/Instructions.h

27.

28.

测试你的指令

29.

30.

llvm/lib/Target/\*:

31.

添加对代码生成器指令的支持，或添加降低通道。

32.

33.

llvm/test/\*:

34.

将您的测试用例添加到测试套件中。

35.

此外，您需要实施（或修改）您想要理解此新指令的任何分析或传递。

添加新类型

警告

添加新类型会更改位码格式，并且会破坏与当前现有 LLVM 安装的兼容性。仅在绝对必要时才添加新类型。

添加基本​​类型

1.

llvm/include/llvm/IR/Type.h:

2.

为新类型添加枚举；Type\*为此类型添加静态

3.

4.

llvm/lib/IR/Type.cpp和llvm/lib/CodeGen/ValueTypes.cpp：

5.

TypeID从=>添加映射Type\*；初始化静态Type\*

6.

7.

llvm/include/llvm-c/Core.h和llvm/lib/IR/Core.cpp：

8.

添加枚举LLVMTypeKind并修改 新类型LLVMTypeKind LLVMGetTypeKind(LLVMTypeRef Ty)

9.

10.

llvm/lib/AsmParser/LLLexer.cpp:

11.

添加从文本程序集中解析类型的能力

12.

13.

llvm/lib/AsmParser/LLParser.cpp:

14.

为该类型添加一个标记

15.

16.

llvm/lib/Bitcode/Writer/BitcodeWriter.cpp:

17.

修改以序列化您的类型void ModuleBitcodeWriter::writeTypeTable()

18.

19.

llvm/lib/Bitcode/Reader/BitcodeReader.cpp:

20.

修改以读取您的数据类型Error BitcodeReader::parseTypeTableBody()

21.

22.

include/llvm/Bitcode/LLVMBitCodes.h:

23.

TypeCodes为新类型添加枚举

24.

添加派生类型

1.

llvm/include/llvm/IR/Type.h:

2.

为新类型添加枚举；还添加该类型的前向声明

3.

4.

llvm/include/llvm/IR/DerivedTypes.h:

5.

添加新类来表示层次结构中的新类；向 TypeMap 值类型添加前向声明

6.

7.

llvm/lib/IR/Type.cpp和llvm/lib/CodeGen/ValueTypes.cpp：

8.

添加对派生类型的支持，特别是enum TypeID和is , get方法。

9.

10.

llvm/include/llvm-c/Core.h和llvm/lib/IR/Core.cpp：

11.

添加枚举并为新类型LLVMTypeKind修改 LLVMTypeKind LLVMGetTypeKind(LLVMTypeRef Ty)

12.

13.

llvm/lib/AsmParser/LLLexer.cpp:

14.

修改以添加从文本程序集中解析类型的能力lltok::Kind LLLexer::LexIdentifier()

15.

16.

llvm/lib/Bitcode/Writer/BitcodeWriter.cpp:

17.

修改以序列化您的类型void ModuleBitcodeWriter::writeTypeTable()

18.

19.

llvm/lib/Bitcode/Reader/BitcodeReader.cpp:

20.

修改以读取您的数据类型Error BitcodeReader::parseTypeTableBody()

21.

22.

include/llvm/Bitcode/LLVMBitCodes.h:

23.

TypeCodes为新类型添加枚举

24.

25.

llvm/lib/IR/AsmWriter.cpp:

26.

修改 以输出新的派生类型void TypePrinting::print(Type \*Ty, raw\_ostream &OS)

27.

如何添加一个受约束的浮点内在

添加内在的

添加 SelectionDAG 节点类型

更新映射

更新 IR 组件

更新选择器组件

o构建 SelectionDAG

添加文档和测试

警告

这是一个正在进行的工作。

添加内在的

添加新的约束内在函数时需要更新多个文件。

将新的内在函数添加到内在函数表中：

include/llvm/IR/Intrinsics.td

添加 SelectionDAG 节点类型

将节点类型的新 STRICT 版本添加到 ISD::NodeType 枚举：

include/llvm/CodeGen/ISDOpcodes.h

严格版本名称必须是前缀STRICT\_和相应非严格节点名称的名称的串联。例如，节点 FADD 的严格版本必须是 STRICT\_FADD。

更新映射

将新记录添加到指令映射到受限内部节点和 DAG 节点：

include/llvm/IR/ConstrainedOps.def

按照此文件中提供的说明进行操作。

更新 IR 组件

更新 IR 验证器：

lib/IR/Verifier.cpp

更新选择器组件

构建 SelectionDAG

SelectionDAGBuilder::visitConstrainedFPIntrinsic 函数使用 ConstrainedOps.def 中指定的映射构建 DAG 节点。但是，如果此默认构建不足够，则可以修改构建，查看它是如何为 STRICT\_FP\_ROUND 实现的。新的 STRICT 节点最终将转换为匹配的非 STRICT 节点。出于这个原因，它应该具有与非 STRICT 版本相同的操作数和值，但也应该使用链。这使得后续共享 STRICT 和非 STRICT 代码路径的代码变得更加容易：

lib/CodeGen/SelectionDAG/SelectionDAGBuilder.cpp

大多数 STRICT 节点与匹配的非 STRICT 节点一样合法化。必须将具有此属性的新 STRICT 节点添加到 SelectionDAGLegalize::LegalizeOp() 中的开关。：

lib/CodeGen/SelectionDAG/LegalizeDAG.cpp

合法器的其他部分可能也需要更新。寻找非 STRICT 对应方合法化的地方，并根据需要进行更新。小心链，因为 STRICT 节点使用它，但它们的对应节点通常不使用。

将 STRICT 节点转换或突变为节点的非 STRICT 版本的代码发生在 SelectionDAG::mutateStrictFPToFP() 中。在大多数情况下，函数可以使用来自 ConstrainedOps.def 的信息进行转换。更新此函数时要小心，因为有些节点的返回类型与其输入操作数相同，但有些不同。这两种情况都必须妥善处理：

lib/CodeGen/SelectionDAG/SelectionDAG.cpp

是否会发生突变取决于新节点如何在 TargetLoweringBase::initActions() 中注册。默认情况下，所有严格节点都注册了 Expand 操作：

lib/CodeGen/TargetLoweringBase.cpp

为了使调试日志可读，更新 SelectionDAG 的调试记录器是有帮助的：

lib/CodeGen/SelectionDAG/SelectionDAGDumper.cpp

添加文档和测试

docs/LangRef.rst

# 附录