目 录

[1. 概述 1](#_Toc1169904786)

[1.1 术语 1](#_Toc1958657606)

[2. 整个llvm相关的pass部分 1](#_Toc1125408200)

[2.1 概述 1](#_Toc1802089985)

[2.1.1 pass的分类概述 1](#_Toc272273151)

[2.1.2 Pass管理器 3](#_Toc1221879557)

[2.2 数据结构 4](#_Toc752248620)

[2.2.1 Pass相关数据结构 4](#_Toc1021802940)

[2.2.1.1 PassInfo 4](#_Toc463465967)

[2.2.1.2 Pass 4](#_Toc1877316994)

[2.2.1.3 ModulePass 5](#_Toc1966266852)

[2.2.1.4 ImmutablePass 5](#_Toc330439188)

[2.2.1.5 FunctionPass 5](#_Toc1008489066)

[2.2.1.6 PassRegistry 5](#_Toc2039108420)

[2.2.2 PassAnalysisSupport相关数据结构 6](#_Toc46814665)

[2.2.2.1 AnalysisUsage 6](#_Toc1729170714)

[2.2.2.2 AnalysisResolver 6](#_Toc574504555)

[2.2.3 PassSupport相关数据结构 8](#_Toc217766838)

[2.2.3.1 INITIALIZE\_PASS宏定义 8](#_Toc1736882135)

[2.2.3.2 INITIALIZE\_PASS\_BEGIN 8](#_Toc1488201970)

[2.2.3.3 INITIALIZE\_PASS\_END 8](#_Toc399270734)

[2.2.3.4 INITIALIZE\_PASS\_DEPENDENCY 8](#_Toc908494587)

[2.2.3.5 PassRegistrationListener 8](#_Toc574181123)

[2.3 全局变量 8](#_Toc44988221)

[2.3.1 Pass注册表 PassRegistryObj 8](#_Toc1708236402)

[2.3.2 标识该pass是否被注册过Initialize##passName##PassFlag 8](#_Toc537764824)

[2.4 相关的接口函数 9](#_Toc2085019923)

[2.4.1 callDefaultCtor 9](#_Toc1597459291)

[3. llvm command line解析llvm::cl workspace 9](#_Toc2092576022)

[3.1 相关的接口函数 9](#_Toc691717553)

[3.2 相关全局变量 9](#_Toc1064329798)

[3.2.1 GlobalParser 9](#_Toc1114997160)

[3.2.2 GeneralCategory 9](#_Toc502891511)

[3.2.3 TopLevelSubCommand 9](#_Toc42254350)

[3.2.4 AllSubCommands 9](#_Toc769603498)

[3.3 相关数据结构 9](#_Toc775164662)

[3.3.1 ExpansionContext 10](#_Toc1264133907)

[3.3.1.1 构造函数 10](#_Toc1521852118)

[3.3.2 CommandLineParser 10](#_Toc1796967603)

[3.3.2.1 registerSubCommand 注册subcommand 10](#_Toc1727599874)

[3.4 回调函数 10](#_Toc1251685465)

[3.4.1 TokenizeGNUCommandLine 10](#_Toc1615750807)

[4. ADT代码解析 10](#_Toc2058039063)

[4.1 数据结构 10](#_Toc112690883)

[4.1.1 int数值相关 10](#_Toc1507375579)

[4.1.2 浮点数值相关 11](#_Toc2104853728)

[4.1.2.1 IEEEFloat 11](#_Toc1841861597)

[4.1.2.2 APFloat 12](#_Toc2081880134)

[4.1.3 FoldingSet 相关 12](#_Toc175136918)

[4.1.3.1 FoldingSetNodeID 12](#_Toc1431260085)

[4.1.4 statistics相关 12](#_Toc1422598457)

[4.1.4.1 TrackingStatistic 12](#_Toc574407653)

[4.2 small vector代码解析 12](#_Toc192271024)

[4.3 Twine代码解析 12](#_Toc1996779580)

[4.3.1 概述 12](#_Toc619395874)

[4.3.2 构造函数 13](#_Toc1900507426)

[5. IR代码解析 14](#_Toc387060757)

[5.1 数据结构 14](#_Toc556932149)

[5.1.1 Type 14](#_Toc1350483069)

[5.1.2 FunctionType 14](#_Toc332153131)

[5.1.3 Use 14](#_Toc1248649702)

[5.1.4 DataLayout相关数据结构 14](#_Toc267329219)

[5.1.4.1 DataLayout 14](#_Toc1447150291)

[5.1.5 Value相关 15](#_Toc1751541213)

[5.1.5.1 Value 15](#_Toc309583569)

[5.1.5.2 User 15](#_Toc69270141)

[5.1.5.3 GlobalValue 15](#_Toc379222228)

[5.1.5.4 ValueHandleBase 16](#_Toc1573717477)

[5.1.5.5 GlobalObject 16](#_Toc1591122260)

[5.1.5.6 GlobalVariable 16](#_Toc28706183)

[5.1.6 OperandTraits相关类 16](#_Toc1153833703)

[5.1.6.1 FixedNumOperandTraits 16](#_Toc695324077)

[5.1.6.2 OptionalOperandTraits 17](#_Toc1644456990)

[5.1.6.3 VariadicOperandTraits 17](#_Toc1064389118)

[5.1.6.4 HungoffOperandTraits 17](#_Toc808014960)

[5.1.6.5 DECLARE\_TRANSPARENT\_OPERAND\_ACCESSORS 宏定义 17](#_Toc1004348922)

[5.1.6.6 DEFINE\_TRANSPARENT\_OPERAND\_ACCESSORS 宏定义 17](#_Toc1021759198)

[5.1.7 Constant相关数据结构 17](#_Toc502392909)

[5.1.7.1 Constant 17](#_Toc938745408)

[5.1.7.2 ConstantData 18](#_Toc1196896117)

[5.1.7.3 ConstantFP 18](#_Toc1933652994)

[5.1.7.4 ConstantExpr 18](#_Toc213860217)

[5.1.8 Comdat 18](#_Toc1771303770)

[5.1.9 BasicBlock 19](#_Toc2125924018)

[5.1.10 指令类相关 19](#_Toc63156150)

[5.1.10.1 Instruction 19](#_Toc243215996)

[5.1.10.2 ReturnInst 19](#_Toc1878947796)

[5.1.10.3 基于ReturnInst显示特化OperandTraits 19](#_Toc450216907)

[5.1.10.4 InstVisitor 类 19](#_Toc800148146)

[5.1.11 Function相关 20](#_Toc1081947218)

[5.1.11.1 Argment 20](#_Toc782370038)

[5.1.11.2 Function 20](#_Toc2048797848)

[5.1.12 LLVMContext相关 21](#_Toc1349276437)

[5.1.12.1 LLVMContextImpl 21](#_Toc82036681)

[5.1.12.2 LLVMContext 21](#_Toc1652855414)

[5.1.13 Module 21](#_Toc1658860007)

[5.1.14 IRBuilder相关 22](#_Toc151306823)

[5.1.14.1 IRBuilderFolder 22](#_Toc2032077642)

[5.1.14.2 ConstantFolder 23](#_Toc1085093836)

[5.1.14.3 IRBuilderBase 23](#_Toc1742429083)

[5.1.14.4 IRBuilder 23](#_Toc2060783825)

[5.1.15 Pass相关的类 23](#_Toc91443891)

[5.1.15.1 PassInstrumentationCallbacks 24](#_Toc290269512)

[5.1.15.2 PassInfoMixin 24](#_Toc1557757167)

[5.1.15.3 AnalysisInfoMixin 24](#_Toc1155833010)

[5.1.15.4 AnalysisManager 24](#_Toc1098284472)

[5.1.16 LegacyPassManager相关类 24](#_Toc414622441)

[5.1.16.1 PassManagerPrettyStackEntry 24](#_Toc30108560)

[5.1.16.2 PMStack 25](#_Toc1600677381)

[5.1.16.3 PMTopLevelManager 25](#_Toc1353367850)

[5.1.16.4 PMDataManager 28](#_Toc1227004677)

[5.1.16.5 FPPassManager 29](#_Toc1386846728)

[5.1.16.6 MPPassManager 31](#_Toc1567228067)

[5.1.16.7 legacy::PassManagerBase 31](#_Toc850824799)

[5.1.16.8 legacy::FunctionPassManager 31](#_Toc1365287098)

[5.1.16.9 FunctionPassManagerImpl 31](#_Toc1630384217)

[5.1.17 GVMaterializer 32](#_Toc1094040796)

[5.1.17.1 GVMaterializer 32](#_Toc1096751247)

[5.1.18 Diagnostic相关类 32](#_Toc2080601124)

[5.1.18.1 DiagnosticHandler 32](#_Toc1894188942)

[5.1.18.2 OptimizationRemarkAnalysis 32](#_Toc31214817)

[5.1.19 DominatorTree相关类 32](#_Toc715487514)

[5.1.19.1 DominatorTreeWrapperPass 32](#_Toc1795503142)

[5.1.20 Verifier相关类 32](#_Toc1380491254)

[5.1.20.1 DominatorTree 33](#_Toc797524196)

[5.1.21 AsmWriter 相关类 33](#_Toc1300874908)

[5.1.21.1 ModuleSlotTracker 33](#_Toc891867613)

[5.1.21.2 TypePrinting 33](#_Toc948831019)

[5.1.21.3 SlotTracker 33](#_Toc1185468902)

[5.2 静态变量或全局变量 33](#_Toc1976961449)

[5.2.1 Diagnostic相关 33](#_Toc543776454)

[5.2.1.1 PassRemarksPassedOptLoc 33](#_Toc1098769079)

[5.2.1.2 PassRemarksMissedOptLoc 33](#_Toc2068405341)

[5.2.1.3 PassRemarksAnalysisOptLoc 33](#_Toc834045966)

[5.3 接口 33](#_Toc509042599)

[5.3.1 ConstantFold相关接口 33](#_Toc1076754703)

[5.3.1.1 ConstantFoldBinaryInstruction 33](#_Toc1932330438)

[5.3.2 Verifier相关接口 34](#_Toc923665040)

[5.3.2.1 verifyFunction 34](#_Toc1106863263)

[5.4 相关处理流程 34](#_Toc1385524171)

[6. support 代码解析 34](#_Toc129549242)

[6.1 数据结构 34](#_Toc186384293)

[6.1.1 ModRef 相关 34](#_Toc624887251)

[6.1.1.1 ModRefInfo 34](#_Toc1696777310)

[6.1.1.2 MemoryEffects 34](#_Toc1037209092)

[6.1.2 TrailingObjects相关 34](#_Toc1990174350)

[6.1.3 TimeProfiler 相关 35](#_Toc1179677879)

[6.1.3.1 TimeTraceProfilerEntry 35](#_Toc2131249888)

[6.1.3.2 TimeTraceProfilerInstances 35](#_Toc939441949)

[6.1.3.3 TimeTraceProfiler 35](#_Toc1112795356)

[6.1.3.4 TimeTraceScope 35](#_Toc1877955182)

[6.1.4 PrettyStackTrace相关 35](#_Toc970656766)

[6.1.4.1 PrettyStackTraceEntry 35](#_Toc1828282870)

[6.2 全局变量和静态变量 35](#_Toc1525974677)

[6.2.1 TimeProfiler 相关 35](#_Toc203664372)

[6.2.1.1 Instances 35](#_Toc478323418)

[6.2.1.2 TimeTraceProfilerInstance 35](#_Toc679365937)

[6.3 接口 36](#_Toc1095531986)

[6.3.1 TargetSelect相关接口 36](#_Toc1427154437)

[6.3.1.1 InitializeNativeTarget 36](#_Toc1864834840)

[6.3.1.2 InitializeNativeTargetAsmPrinter 36](#_Toc925009787)

[6.3.1.3 InitializeNativeTargetAsmParser 36](#_Toc1970930891)

[7. Analysis相关代码解析 36](#_Toc816120271)

[7.1 数据结构 36](#_Toc845931480)

[7.1.1 AliasAnalysis相关 36](#_Toc657493209)

[7.1.1.1 BasicAAWrapperPass 37](#_Toc1325162870)

[7.1.1.2 AAResultsWrapperPass 37](#_Toc1922686183)

[7.1.2 AssumptionCache 相关 37](#_Toc442339999)

[7.1.3 BranchProbability相关 37](#_Toc101344263)

[7.1.3.1 LazyBranchProbabilityInfoPass 37](#_Toc882065799)

[7.1.4 BlockFrequency相关 37](#_Toc1827864171)

[7.1.4.1 LazyBlockFrequencyInfoPass 37](#_Toc230893505)

[7.1.5 LoopInfo相关 37](#_Toc1068450092)

[7.1.5.1 LoopInfoWrapperPass 37](#_Toc305267774)

[7.1.6 MemoryDependenceAnalysis相关 37](#_Toc1927670815)

[7.1.6.1 MemoryDependenceWrapperPass 37](#_Toc2105659184)

[7.1.7 OptimizationRemarkEmitter相关 37](#_Toc147958476)

[7.1.7.1 OptimizationRemarkEmitterWrapperPass 37](#_Toc959865047)

[7.1.8 PostDominatorTree 37](#_Toc2089425425)

[7.1.8.1 PostDominatorTreeWrapperPass 37](#_Toc1087400425)

[7.1.9 ScalarEvolution相关 38](#_Toc2072660403)

[7.1.9.1 ScalarEvolutionWrapperPass 38](#_Toc1819896959)

[7.1.10 TargetLibraryInfo相关 38](#_Toc2058057191)

[7.1.10.1 TargetLibraryInfoWrapperPass 38](#_Toc1753459625)

[8. Transforms相关代码解析 38](#_Toc1198387988)

[8.1 数据结构 38](#_Toc114237916)

[8.1.1 InstCombine相关 38](#_Toc84299396)

[8.1.1.1 InstructionCombiningPass 38](#_Toc1877753926)

[8.1.2 Scalar 相关 39](#_Toc1209769902)

[8.1.2.1 CFGSimplifyPass 39](#_Toc1511453833)

[8.1.2.2 GVNLegacyPass 39](#_Toc1595105118)

[8.1.2.3 LowerConstantIntrinsics 39](#_Toc2134779689)

[8.1.2.4 MergeICmpsLegacyPass 39](#_Toc1334901077)

[8.1.2.5 PartiallyInlineLibCallsLegacyPass 39](#_Toc263741741)

[8.1.2.6 ReassociateLegacyPass 39](#_Toc833227522)

[8.1.2.7 SimplifyCFGOptions 39](#_Toc1992394286)

[8.1.3 Utils 相关 39](#_Toc1588904612)

[8.1.3.1 InstructionWorklist 39](#_Toc608430057)

[8.1.3.2 LoopSimplifyPass 39](#_Toc287250638)

[8.2 全局变量和静态变量 40](#_Toc1690248875)

[8.2.1 InstCombine相关 40](#_Toc1490495856)

[8.3 接口 40](#_Toc2115114809)

[8.3.1 InstCombine相关接口 40](#_Toc1921142380)

[8.3.1.1 createInstructionCombiningPass 40](#_Toc411462300)

[8.3.1.2 initializeInstructionCombiningPassPassOnce 40](#_Toc272898935)

[8.3.2 Scalar 相关接口 40](#_Toc1701329548)

[8.3.2.1 createReassociatePass 40](#_Toc369637837)

[8.3.2.2 createGVNPass 41](#_Toc420857412)

[8.3.2.3 createCFGSimplificationPass 41](#_Toc513710947)

[9. ExecutionEngine 相关代码解析 41](#_Toc311579614)

[9.1 数据结构 41](#_Toc1508257837)

[9.1.1 orc ExecutorProcessControl相关结构体 41](#_Toc438887702)

[9.1.1.1 ExecutorProcessControl 41](#_Toc2131476573)

[9.1.2 Orc core相关结构体 41](#_Toc1418831381)

[9.1.2.1 ExecutionSession 41](#_Toc44863679)

[9.2 接口 41](#_Toc1182380914)

[10. CodeGen相关代码解析 41](#_Toc1533069297)

[10.1 数据结构 41](#_Toc129163075)

[10.1.1 AssignmentTrackingAnalysis 41](#_Toc912651192)

[10.1.1.1 AssignmentTrackingAnalysis 41](#_Toc595355551)

[10.1.2 AtomicExpandPass 41](#_Toc1640616909)

[10.1.2.1 AtomicExpand 41](#_Toc360272662)

[10.1.3 BreakFalseDeps 42](#_Toc582651592)

[10.1.3.1 BreakFalseDeps 42](#_Toc828034338)

[10.1.4 BranchFolding 42](#_Toc624014403)

[10.1.4.1 BranchFolderPass 42](#_Toc1415879114)

[10.1.5 CFIInstrInserter 42](#_Toc672944976)

[10.1.5.1 CFIInstrInserter 42](#_Toc65435367)

[10.1.6 EdgeBundles 42](#_Toc2024309172)

[10.1.6.1 EdgeBundles 42](#_Toc960195614)

[10.1.7 ExpandLargeDivRem 42](#_Toc1755684242)

[10.1.7.1 ExpandLargeDivRemLegacyPass 42](#_Toc1367321380)

[10.1.8 ExpandLargeDivRemLegacyPass 42](#_Toc927826775)

[10.1.8.1 ExpandLargeDivRemLegacyPass 42](#_Toc1529342975)

[10.1.9 ExpandLargeFpConvert 42](#_Toc1778783681)

[10.1.9.1 ExpandLargeFpConvertLegacyPass 42](#_Toc1200725711)

[10.1.10 ExpandMemCmp 42](#_Toc1083188875)

[10.1.10.1 ExpandMemCmpPass 42](#_Toc937870)

[10.1.11 ExpandPostRAPseudos 42](#_Toc1621583123)

[10.1.11.1 ExpandPostRA 42](#_Toc1596899822)

[10.1.12 ExpandReductions 42](#_Toc312517484)

[10.1.12.1 ExpandReductions 42](#_Toc982357312)

[10.1.13 ExpandVectorPredication 42](#_Toc2035787524)

[10.1.13.1 ExpandVectorPredication 42](#_Toc296510409)

[10.1.14 FEntryInserter 43](#_Toc253705045)

[10.1.14.1 FEntryInserter 43](#_Toc2080651203)

[10.1.15 FinalizeISel 43](#_Toc1478891323)

[10.1.15.1 FinalizeISel 43](#_Toc1786774342)

[10.1.16 FuncletLayout 43](#_Toc62330631)

[10.1.16.1 FuncletLayout 43](#_Toc244058867)

[10.1.17 GCRootLowering 43](#_Toc234646245)

[10.1.17.1 GCMachineCodeAnalysis 43](#_Toc1702947540)

[10.1.18 IndirectBrExpandPass 43](#_Toc604331529)

[10.1.18.1 IndirectBrExpandPass 43](#_Toc817297838)

[10.1.19 LazyMachineBlockFrequencyInfo 43](#_Toc383498230)

[10.1.19.1 LazyMachineBlockFrequencyInfoPass 43](#_Toc1228345933)

[10.1.20 LiveDebugVariables 43](#_Toc85693304)

[10.1.20.1 LiveDebugVariables 43](#_Toc1056443206)

[10.1.21 LiveRegMatrix 43](#_Toc1293781300)

[10.1.21.1 LiveRegMatrix 43](#_Toc2110002476)

[10.1.22 LiveStacks 43](#_Toc2016638821)

[10.1.22.1 LiveStacks 43](#_Toc901981895)

[10.1.23 LiveVariables 43](#_Toc1329840209)

[10.1.23.1 LiveVariables 43](#_Toc796981948)

[10.1.24 LocalStackSlotAllocation 43](#_Toc283841222)

[10.1.24.1 LocalStackSlotPass 43](#_Toc961140242)

[10.1.25 MachineBlockPlacement 44](#_Toc1997707659)

[10.1.25.1 MachineBlockPlacement 44](#_Toc1367030097)

[10.1.26 MachineCopyPropagation 44](#_Toc962078112)

[10.1.26.1 MachineCopyPropagation 44](#_Toc1471807134)

[10.1.27 MachineCSE 44](#_Toc816446271)

[10.1.27.1 MachineCSE 44](#_Toc1274595596)

[10.1.28 MachineCycleAnalysis 44](#_Toc306680799)

[10.1.28.1 MachineCycleInfoWrapperPass 44](#_Toc704750147)

[10.1.29 MachineDominators 44](#_Toc1571106005)

[10.1.29.1 MachineDominatorTree 44](#_Toc560385844)

[10.1.30 MachineDominanceFrontier 44](#_Toc637917702)

[10.1.30.1 MachineDominanceFrontier 44](#_Toc902513681)

[10.1.31 MachineInstrBundle 44](#_Toc199676539)

[10.1.31.1 UnpackMachineBundles 44](#_Toc700248333)

[10.1.32 MachineLateInstrsCleanup 44](#_Toc1146572548)

[10.1.32.1 MachineLateInstrsCleanup 44](#_Toc434322784)

[10.1.33 MachineLICM 44](#_Toc255712225)

[10.1.33.1 EarlyMachineLICM 44](#_Toc1750904078)

[10.1.33.2 MachineLICM 44](#_Toc1251620622)

[10.1.34 MachineLoopInfo 44](#_Toc639210455)

[10.1.34.1 MachineLoopInfo 44](#_Toc831766363)

[10.1.35 MachineOptimizationRemarkEmitter 45](#_Toc1337313927)

[10.1.35.1 MachineOptimizationRemarkEmitterPass 45](#_Toc1695653662)

[10.1.35.2 MachineOptimizationRemarkAnalysis 45](#_Toc2125547663)

[10.1.36 MachinePostDominators 45](#_Toc1299832755)

[10.1.36.1 MachinePostDominatorTree 45](#_Toc1564808835)

[10.1.37 MachineTraceMetrics 45](#_Toc880045910)

[10.1.37.1 MachineTraceMetrics 45](#_Toc482189316)

[10.1.38 MachineSanitizerBinaryMetadata 45](#_Toc214307135)

[10.1.38.1 MachineSanitizerBinaryMetadata 45](#_Toc1163887132)

[10.1.39 MachineScheduler 45](#_Toc1443329558)

[10.1.39.1 MachineScheduler 45](#_Toc64531147)

[10.1.40 MachineSink 45](#_Toc383433581)

[10.1.40.1 MachineSinking 45](#_Toc257924022)

[10.1.41 OptimizePHIs 45](#_Toc1536338281)

[10.1.41.1 OptimizePHIs 45](#_Toc1199879852)

[10.1.42 PrologEpilogInserter 45](#_Toc1532519618)

[10.1.42.1 PEI 45](#_Toc1843019080)

[10.1.43 RegisterCoalescer 45](#_Toc1904629999)

[10.1.43.1 RegisterCoalescer 45](#_Toc956141976)

[10.1.44 RemoveRedundantDebugValues 45](#_Toc255921277)

[10.1.44.1 RemoveRedundantDebugValues 45](#_Toc395064054)

[10.1.45 ReplaceWithVeclib 46](#_Toc1858655657)

[10.1.45.1 ReplaceWithVeclibLegacy 46](#_Toc455597816)

[10.1.46 SafeStack 46](#_Toc1095312387)

[10.1.46.1 SafeStackLegacyPass 46](#_Toc857744557)

[10.1.47 ShadowStackGCLowering 46](#_Toc889920600)

[10.1.47.1 ShadowStackGCLowering 46](#_Toc1351024613)

[10.1.48 SlotIndexes 46](#_Toc461164987)

[10.1.48.1 SlotIndexes 46](#_Toc2141541223)

[10.1.49 SpillPlacement 46](#_Toc1990235068)

[10.1.49.1 SpillPlacement 46](#_Toc1292931350)

[10.1.50 StackColoring 46](#_Toc1331371502)

[10.1.50.1 StackColoring 46](#_Toc1538405082)

[10.1.51 StackMapLivenessAnalysis 46](#_Toc1270995366)

[10.1.51.1 StackMapLiveness 46](#_Toc483720609)

[10.1.52 StackProtector 46](#_Toc955730269)

[10.1.52.1 StackProtector 46](#_Toc3557628)

[10.1.53 TailDuplication 46](#_Toc965909926)

[10.1.53.1 EarlyTailDuplicate 46](#_Toc1170037405)

[10.1.54 UnreachableBlockElimLegacy 46](#_Toc1167444761)

[10.1.54.1 UnreachableBlockElimLegacyPass 46](#_Toc261755836)

[10.1.55 DeadMachineInstructionElim 46](#_Toc1234568552)

[10.1.55.1 DeadMachineInstructionElim 46](#_Toc1550878342)

[10.1.56 LiveIntervals 47](#_Toc519679859)

[10.1.56.1 LiveIntervals 47](#_Toc623423185)

[10.1.57 PatchableFunction 47](#_Toc603274547)

[10.1.57.1 PatchableFunction 47](#_Toc2052199477)

[10.1.58 PeepholeOptimizer 47](#_Toc318958618)

[10.1.58.1 PeepholeOptimizer 47](#_Toc360420898)

[10.1.59 PHIElimination 47](#_Toc860857805)

[10.1.59.1 PHIElimination 47](#_Toc574879895)

[10.1.60 PostRASchedulerList 47](#_Toc755484952)

[10.1.60.1 PostRAScheduler 47](#_Toc572029814)

[10.1.61 ProcessImplicitDefs 47](#_Toc1030477711)

[10.1.61.1 ProcessImplicitDefs 47](#_Toc1850797340)

[10.1.62 ReachingDefAnalysis 47](#_Toc1429774372)

[10.1.62.1 ReachingDefAnalysis 47](#_Toc1920398311)

[10.1.63 TailDuplication 47](#_Toc1054338305)

[10.1.63.1 TailDuplicate 47](#_Toc1890939359)

[10.1.64 TwoAddressInstructionPass 47](#_Toc1914455886)

[10.1.64.1 TwoAddressInstructionPass 47](#_Toc897089725)

[10.1.65 UnreachableBlockElim 47](#_Toc1036387062)

[10.1.65.1 UnreachableMachineBlockElim 47](#_Toc1098343740)

[10.1.66 VirtRegMap 47](#_Toc288011160)

[10.1.66.1 VirtRegMap 47](#_Toc159898780)

[10.1.67 XRayInstrumentation 48](#_Toc1582064350)

[10.1.67.1 XRayInstrumentation 48](#_Toc1243741429)

[10.2 接口 48](#_Toc163456408)

[11. Target相关代码解析 48](#_Toc400490628)

[11.1 数据结构 48](#_Toc266295186)

[11.1.1 X86LowerAMXType 48](#_Toc1330901169)

[11.1.1.1 X86LowerAMXTypeLegacyPass 48](#_Toc662246464)

[11.1.2 X86ISelDAGToDAG 48](#_Toc1500863738)

[11.1.2.1 X86DAGToDAGISel 48](#_Toc734295864)

[11.2 接口 48](#_Toc1181926323)

[11.2.1 X86相关 48](#_Toc2124286924)

[11.2.1.1 LLVMInitializeX86Target 48](#_Toc1337570411)

[11.2.1.2 LLVMInitializeX86AsmPrinter 48](#_Toc1086642153)

[12. 参考文档 48](#_Toc295761894)

# 概述

## 术语

|  |  |
| --- | --- |
| 术语 | 解释 |
|  |  |
| CFG | control flow graph，控制流图 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| POD | POD 是 Plain Old Data 的缩写，是 C++ 定义的一类数据结构概念，比如 int、float 等都是 POD 类型的。Plain 代表它是一个普通类型，Old 代表它是旧的，与几十年前的 C 语言兼容，那么就意味着可以使用 memcpy() 这种最原始的函数进行操作。两个系统进行交换数据，如果没有办法对数据进行语义检查和解释，那就只能以非常底层的数据形式进行交互，而拥有 POD 特征的类或者结构体通过二进制拷贝后依然能保持数据结构不变。也就是说，能用 C 的 memcpy() 等函数进行操作的类、结构体就是 POD 类型的数据。 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 整个llvm相关的pass部分

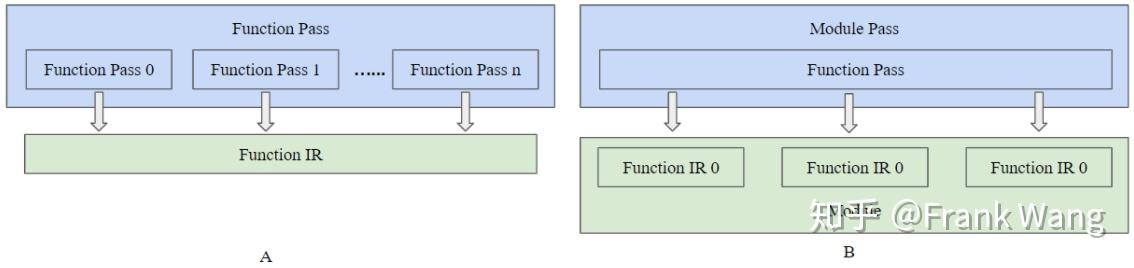
## 概述

### pass的分类概述

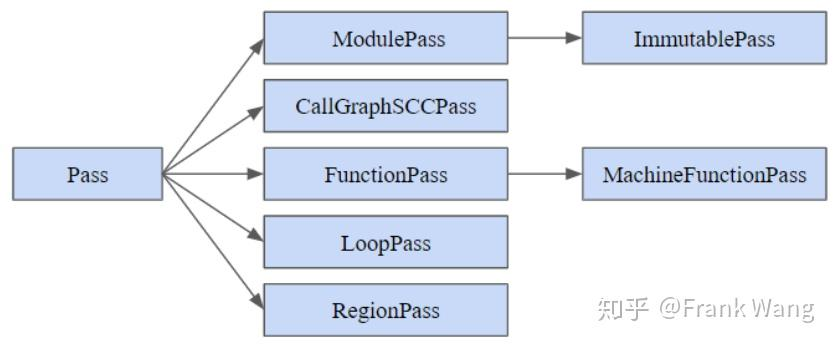
LLVM提供的pass分为三类:Analysis pass、Transform pass和Utility pass。

1. Analysis pass计算相关IR单元的高层信息，但不对其进行修改。这些信息可以被其他pass使用，或用于调试和程序可视化。简言之，Analysis pass提供其它pass需要查询的信息并提供查询接口。例如，basic-aa pass是基本别名分析（Basic Alias Analysis）pass，得到的别名分析结果可以用于后续的其它优化pass。Analysis pass不仅从IR中得到有用信息，还可以通过调用其它Analysis pass得到信息，并将这些信息结合起来，得到关于IR更有价值的信息。这些分析结果可以被缓存下来，直到分析的IR被修改，原有的分析结果当然也就失效了。
2. Transform pass可以使用Analysis pass。Transform pass会检视IR，查询Analysis pass得到IR的高层信息，然后以某种方式改变和优化IR，并保证改变后的IR仍然合法有效。例如，adce pass是激进的死代码消除（Aggressive Dead Code Elimination）pass，会将死代码从原来的模块中删除。
3. Utility pass是一些功能性的实用程序，既不属于Analysis pass，也不属于Transform pass。例如，extract-blocks pass将basic block从模块中提取出来供bugpoint使用，这个utility pass既不属于Analysis pass，也不属于Transform pass。参考文献[1]中列出了LLVM提供的所有pass。当调用RegisterPass()注册自定义pass时，会要求指定是否为Analysis pass。通过RegisterPass()注册自定义pass后，就可以使用LLVM opt工具对IR调用自定义pass功能。

LLVM Pass的基础模块是Pass类，这是所有pass的基类。自定义的pass类都要从预定义子类中继承，并根据自定义pass的具体功能要求覆写虚函数或增加新的功能函数。预定义子类包括ModulePass、CallGraphSCCPass、FunctionPass、LoopPass和RegionPass类等等。不同的子类有不同的约束条件，这些约束条件在调度pass时会用到。设计自定义pass时的首要任务就是确定自定义pass的基类。在为pass选择基类时，应在满足要求的前提下，尽可能选择最相关的类。这些类会为LLVM Pass基础结构提供优化运行所必需的信息，避免生成的编译器因为选择的基类不合适而导致运行速度变慢。各种pass组合在一起，完成各种IR优化任务。Pass之间的组合可以分为两类：一、多个pass作用于同一个IR单元，function pass是一个典型例子。如下图A所示，Function pass作用于一个function IR，但也可以在某个function pass中运行其它几个function pass，将这几个function pass组合起来作用于同一个IR单元，获得更好的优化效果。二、将一个IR单元分解为更小的单元，并用相应类型的pass处理。如下图B所示，module pass作用于module，但也可以在某个module pass中运行function pass，作用于module中的每一个function，这就将一个IR单元分解为粒度更细的单元来处理。在编译器开发时，可以混合使用两种方式，将各种pass组合为流水线，对IR做不同处理和优化。



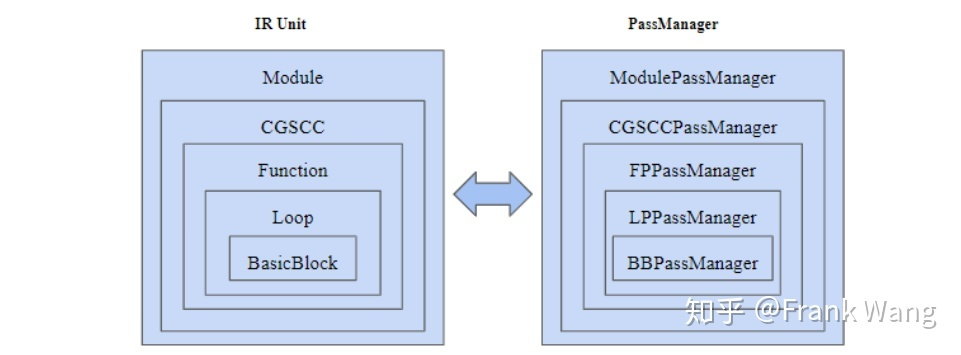
LLVM Pass类及其子类如下图所示：



### Pass管理器

Pass管理器用于注册、调度pass，并维护pass之间的依赖关系。Pass管理器会维护一个pass序列，Pass管理器负责维护和优化这些pass的执行，保证先行（prerequisite）pass正确设置。pass序列中的每一个pass在特定IR单元上依次运行。当前pass可以指定自己对其它pass的依赖性，也就是说，被依赖pass需要在当前pass之前运行。另外，当前pass可以指定将由于执行当前pass而失效的pass。 LLVM中的PassManager类可确保在运行pass之前获得所需的分析结果，并确保在编译过程结束并销毁PassManager时一并销毁pass。因此，PassManager类是pass流水线结构的最主要和最基本的构建模块。

PassManager类的pass流水线嵌套结构和IR单元的嵌套结构对应。IR中的Module IR有ModulePassManager与之对应，CGSCC(Call Graph Strongly Connected Component) IR有SCCPassManager，Function IR有FunctionPassManager等等。各种类型的PassManager通过其内部更小的流水线结构遍历对应的IR单元，如此这般才确定了所有pass的执行顺序。PassManager内部实际上是通过依赖图组织pass，开发者不需要了解依赖图如何实现，因为LLVM API使开发者可以在编译过程的不同阶段注册和添加任何pass，比如，只需要通过PassManager的add()接口就可以向PassManager添加pass。IR单元和PassManager类型的对应关系如下图所示：



## 数据结构

### Pass相关数据结构

文件：include/llvm/Pass.h include/llvm/PassInfo.h

说明：该文件定义了一个基类，指示指定的类是一个transformation pass的实现。

passes以这种方式设计，因此可以在缓存中运行passes并以最优的顺序组织而不必在前端指定它。这允许将任意passes串在一起并尽可能高效地执行它们。

passes应扩展以下类别之一，这取决于它可以保证在运行时将修改的内容。例如，大多数全局优化应该派生自 FunctionPass，因为它们不添加或删除函数，它们操作函数的内部结构。

请注意，此文件 #includes PassSupport.h 和 PassAnalysisSupport.h（位于底部），因此这些文件公开的 API 也自动可供所有用户使用这个文件。

#### **PassInfo**

文件：PassInfo.h

说明：PassInfo 类 - 此类的实例存在于系统已知的每个pass中，并且可以通过调用其 getPassInfo() 方法从live pass中获取。这些对象由 RegisterPass<> 模板创建。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| PassName | StringRef | Nice name for Pass |
| PassArgument | StringRef | Command Line argument to run this pass |
| PassID | const void \* |  |
| IsCFGOnlyPass | const bool | Pass only looks at the CFG. |
| IsAnalysis | const bool | True if an analysis pass |
| IsAnalysisGroup | const bool | True if an analysis group |
| ItfImpl | std::vector<const PassInfo \*> | Interfaces implemented by this pass |
| NormalCtor | NormalCtor\_t |  |

操作：

* Pass \*createPass() const

使用这个方法创建一个pass的实例。默认情况下，会调用callDefaultCtor， 见INITIALIZE\_PASS宏。

#### Pass

文件：Pass.h

说明：Pass接口--所有passes都需要的实现。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| Resolver | AnalysisResolver \* | Used to resolve analysis |
| PassID | const void \* |  |
| Kind | PassKind |  |

操作：

* virtual bool doInitialization(Module &)

doInitialization - 由子类覆盖的虚拟方法，在运行任何传递之前进行任何必要的初始化。

* virtual void \*getAdjustedAnalysisPointer(AnalysisID ID)

当一个pass通过多重继承实现一个解析接口时使用该方法。如果需要，它应该重写它以根据需要为指定的 pass info 调整该指针。

* template<typename AnalysisType> AnalysisType &getAnalysis() const;

getAnalysis<AnalysisType>()---子类使用此函数来获取分析信息，这些分析信息由 override getAnalysisUsage 函数来使用。

* virtual PassManagerType getPotentialPassManagerType() const

Return what kind of Pass Manager can manage this pass

* virtual void preparePassManager(PMStack &)

检查对于当前的pass是否有匹配的可用的pass managers。默认实现是，do nothing。

* void setResolver(AnalysisResolver \*AR)

设置Access AnalysisResolver.

#### ModulePass

文件：Pass.h

说明：ModulePass 类———此类用于实现非结构化过程间优化和分析。 ModulePasses 可以对程序做任何他们想做的事情。

#### ImmutablePass

文件：Pass.h

说明：ImmutablePass 类——该类用于提供不需要运行的信息。这对于诸如目标信息和 AnalysisGroups 的“基本”版本之类的东西很有用。

#### FunctionPass

文件：Pass.h

说明：FunctionPass 类——此类用于实现大多数全局优化。如果满足以下约束，优化应该将此类子类化：

1. 优化是全局组织的，即一次一个函数
2. 优化一个功能不会导致模块中任何功能的增加或删除

操作：

#### PassRegistry

文件：./include/llvm/PassSupport.h

类操作：

* static PassRegistry \*getPassRegistry()

getPassRegistry - 访问全局注册表对象，该对象在应用程序启动时自动初始化并由 llvm\_shutdown 销毁。

* void registerPass(const PassInfo &PI, bool ShouldFree = false)

registerPass - 向pass注册表中（通过其 PassInfo）注册一个pass。PassManager 要想使用该pass，这个注册操作是必须的。该PassInfo对象会存放在PassInfoMap和PassInfoStringMap中。并通知所有感兴趣的listeners。

### PassAnalysisSupport相关数据结构

文件：PassAnalysisSupport.h

说明：This file defines stuff that is used to define and "use" Analysis Passes. This file is automatically #included by Pass.h

#### AnalysisUsage

说明：表示一个pass的分析使用信息。此跟踪分析 pass REQUIRES（必须在pass运行时可用）、REQUIRES TRANSITIVE（必须在pass的整个生命周期内可用), 并分析pass PRESERVES (pass不会使这些分析的结果无效)。此信息通过 getAnalysisUsage 虚函数传递给 pass infrastructure。

属性

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| Required | SmallVector<AnalysisID, 8> |  |
| RequiredTransitive | SmallVector<AnalysisID, 2> |  |
| Preserved | SmallVector<AnalysisID, 2> |  |
| Used | SmallVector<AnalysisID, 0> |  |
| PreservesAll | bool |  |

操作：

* AnalysisUsage &addRequiredID(const void \*ID) AnalysisUsage &addRequiredID(char &ID)

添加一个需要的pass的ID到Required属性中

* template<class PassClass> AnalysisUsage &addRequired()

添加一个需要的pass的PassClass::ID到Required属性中

* AnalysisUsage &addRequiredTransitiveID(char &ID)

添加一个需要的Transitive pass的ID到RequiredTransitive属性中

* template<class PassClass> AnalysisUsage &addRequiredTransitive()

添加一个需要的Transitive pass的PassClass::ID到RequiredTransitive属性中

* const VectorType &getRequiredSet()

返回Required属性

#### AnalysisResolver

说明：AnalysisResolver - Pass 对象使用的简单接口，用于从负责管理 pass 的 pass manager中提取所有分析信息。

属性

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| AnalysisImpls | vector<std::pair<AnalysisID, Pass \*>> | 它会跟踪哪些passes实现了当前pass所需的接口（以实现 getAnalysis()）。 |
| PM | PMDataManager & | PassManager that is used to resolve analysis info |
|  |  |  |
|  |  |  |
|  |  |  |

操作：

* Pass \*findImplPass(AnalysisID PI)

Find pass that is implementing PI

### PassSupport相关数据结构

#### INITIALIZE\_PASS宏定义

说明：

1. 定义static void \*initialize##passName##PassOnce(PassRegistry &Registry)函数

该函数用于创建一个PassInfo对象，并注册到Pass注册表 PassRegistryObj中。

1. static llvm::once\_flag Initialize##passName##PassFlag 静态变量。

1. void llvm::initialize##passName##Pass(PassRegistry &Registry)函数

使用举例：

Transforms/Scalar/Reassociate.cpp 文件中

|  |
| --- |
| INITIALIZE\_PASS(ReassociateLegacyPass, "reassociate",  "Reassociate expressions", false, false) |

在ReassociateLegacyPass class的构造函数中调用：

|  |
| --- |
| ReassociateLegacyPass() : FunctionPass(ID) {  initializeReassociateLegacyPassPass(\*PassRegistry::getPassRegistry());  } |

#### INITIALIZE\_PASS\_BEGIN

#### INITIALIZE\_PASS\_END

#### INITIALIZE\_PASS\_DEPENDENCY

#### PassRegistrationListener

说明：PassRegistrationListener class - 此类旨在由对那些在运行时注册和取消注册的passes 感兴趣的客户端派生（这可能是因为 RegisterPass 构造函数在程序启动时运行，或者可能是因为刚刚加载了一个共享对象）。

操作：

* virtual void passRegistered(const PassInfo \*)

Callback functions - These functions are invoked whenever a pass is loaded or removed from the current executable.

## 全局变量

### Pass注册表 PassRegistryObj

static PassRegistry PassRegistryObj;

文件：PassRegistry.cpp

### 标识该pass是否被注册过Initialize##passName##PassFlag

static llvm::once\_flag Initialize##passName##PassFlag

文件：include/llvm/PassRegistry.h

具体使用在每个INITIALIZE\_PASS宏使用的地方，见INITIALIZE\_PASS宏定义说明。

## 相关的接口函数

### callDefaultCtor

说明：创建一个 Pass对象。

声明：template <class PassName, std::enable\_if\_t<std::is\_default\_constructible<PassName>{}, bool>= true> Pass \*callDefaultCtor()

template <class PassName, std::enable\_if\_t<std::is\_default\_constructible<PassName>{}, bool>= true> Pass \*callDefaultCtor()

# llvm command line解析llvm::cl workspace

## 相关的接口函数

|  |  |
| --- | --- |
| 函数名 | 解释 |
| ParseCommandLineOptions | 解析命令行，并保存信息到全局变量中。 |
| getGeneralCategory | general option category |
|  |  |
| getFileOrSTDIN | 读取输入文件或stdio，返回读取内容到MemoryBuffer。 |
|  |  |

## 相关全局变量

### GlobalParser

static ManagedStatic<CommandLineParser> GlobalParser;

### GeneralCategory

static OptionCategory GeneralCategory{"General options"};

### TopLevelSubCommand

ManagedStatic<SubCommand> llvm::cl::TopLevelSubCommand

### AllSubCommands

ManagedStatic<SubCommand> llvm::cl::AllSubCommands

## 相关数据结构

### ExpansionContext

#### 构造函数

基于BumpPtrAllocator和TokenizerCallback。

### CommandLineParser

#### registerSubCommand 注册subcommand

初始化全局变量TopLevelSubCommand和AllSubCommands。

## 回调函数

### TokenizeGNUCommandLine

# ADT代码解析

## 数据结构

### int数值相关

文件：APInt.h

说明：任意精度整数的类。

APInt 是常见大小写无符号整数类型（如“unsigned”、“unsigned long”或“uint64\_t”）的功能替代，但也允许non-byte-width的整数大小和超大的整数值类型，例如 3-bits、15-bits或超过 64 位的精度。 APInt 提供了多种算术运算符和方法来操作任何位宽的整数值。它同时支持典型的整数算术和比较运算以及按位操作。

该类有几个值得注意的不变量：

* 所有位、字节和字位置都是从零开始的。
* 一旦位宽设置后，它不会更改，除非通过 Truncate、SignExtend 或 ZeroExtend 操作。
* 所有二元运算符必须在具有相同位宽的 APInt 实例上。尝试在实例上使用不同的位宽的这些运算符将产生断言。
* 该值被规范地存储为无符号值。对于操作有所不同，操作有有符号和无符号变体。例如，sdiv 和 udiv。但是，由于位宽必须相同，因此无论值是否被解释为有符号，诸如 Mul 和 Add 之类的操作都会产生相同的结果。
* 一般来说，该类尝试遵循 LLVM 在其 IR 中使用的计算风格。这简化了它在 LLVM 中的使用。
* APInt 支持零位宽值，但需要位的操作未在其上定义（例如，你不能要求零位整数的符号）。这意味着定义了零扩展和逻辑移位等操作，但没有定义符号扩展和 ashr。零位值比较和散列相等自己，countLeadingZeros 返回 0。

类操作：

* static WordType tcAdd(WordType \*, const WordType \*, WordType carry, unsigned)

DST += RHS + CARRY where CARRY is zero or one. Returns the carry flag.

### 浮点数值相关

文件：APFloat.h

说明：独立于主机和目标的任意精度浮点软件实现。

APFloat 使用 APInt 类中的静态函数提供的 bignum 整数算术。lib使用 bignum 整数，其部分是至少 16 位宽的任何无符号类型，但建议使用 64 位。

为清晰而不是速度而写，特别是为了交叉编译器在前端使用，以便目标算法可以在主机上正确执行。尽管如此，性能应该是合理的，特别是对于其预期用途。它在开发更快的目标特定库期间，作为一个运行时库的基础实现可能很有用的。

IEEE-754R 草案中的所有 5 种舍入模式都针对所有已实现的操作进行了正确处理。当前实现的操作是加法， 减法、乘法、除法、融合乘法加法、转换为浮点数、转换为整数和从整数转换。可以使用三或四行代码添加新的舍入模式（例如，远离零）。

内置四种格式：IEEE 单精度、双精度、四倍精度和 x87 80 位扩展双精度（以完全扩展精度运行时）。添加符合 IEEE 语义的新格式只需要添加两行代码：格式的声明和定义。

所有操作都以exception bit-mask的形式返回该操作的状态，因此可以连续执行多个操作与他们的结果或运算在一起。返回的状态可用于编译器诊断；例如，不精确、下溢和溢出可以很容易地在常量折叠上诊断出来，编译器优化器可以确定折叠操作会引发哪些异常并相应地优化或不优化。

目前，下溢微小是在四舍五入后检测的；应该直接添加支持舍入前的情况。

该库按照 C99 读取十六进制浮点数，并在必要时根据指定的舍入模式正确舍入。语法必须被调用者验证。它还根据 C99 %a 和 %A 转换将浮点数转换为十六进制文本。 输出精度（或者自然最小精度）可以被指定；如果请求的精度小于自然精度，则针对指定的舍入模式正确舍入输出。

它还读取十进制浮点数并根据指定的舍入模式正确舍入。

当前未实现到十进制文本的转换。

非零有限数在内部表示为一个符号位、一个 16 位有符号指数和一个有效数字整数部分数组。数精度 P 归一化后，指数在格式范围内，如果数不是非正规数，则尾数的第 P 位设置为显式整数。对于非正规数，最高有效位向右移动，以便指数保持在格式的最小值，因此最小的非正规数只有有效数的最低有效位。零和无穷大的符号很重要；此类数字的指数和尾数未存储，但具有已知的隐式（确定性）值：有效数为 0，零指数为 0，所有指数位为1 用于无穷大指数。对于 NaN，符号和有效数是确定性的，尽管没有真正意义，并且在非转换操作中保留。指数是隐含的全 1 位。

除了默认异常处理之外，APFloat不提供任何异常处理。We represent Signaling NaNs via IEEE-754R 2008 6.2.1 should clause by encoding Signaling NaNs with the first bit of its trailing significand as 0.

#### IEEEFloat

#### APFloat

操作：

* opStatus add(const APFloat &RHS, roundingMode RM)

根据语义，调用IEEE.add或Double.add操作。

### FoldingSet 相关

文件：FoldingSet.h

#### FoldingSetNodeID

说明：FoldingSetNodeID - 此类用于收集节点的所有唯一数据位。当收集到所有位时，此类用于为节点生成哈希值。

### statistics相关

文件：ADT/Statistic.h

说明：该文件定义了“Statistic”类，该类旨在作为一种从 pass 中公开各种指标的简便方法。当在命令行上传递 -stats 命令行选项时，这些统计信息将在运行结束时打印（来自 llvm\_shutdown）。

这对于报告信息非常有用，例如通，过各种转换简化、优化或删除的指令数量，如下所示：

|  |
| --- |
| static Statistic NumInstsKilled("gcse", "Number of instructions killed");  Later, in the code: ++NumInstsKilled; |

注意：Statistics \*must\*声明为全局变量。

#### TrackingStatistic

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| DebugType | const char \*const |  |
| Name | const char \*const |  |
| Desc | const char \*const |  |
| Value | std::atomic<uint64\_t> |  |
| Initialized | std::atomic<bool> |  |

## small vector代码解析

## Twine代码解析

### 概述

Twine-一种轻量级数据结构，用于有效地将临时值串接为字符串。

Twine是一种绳子，它使用二叉树表示连接的字符串，其中字符串是节点的前序。由于使用结果时可以将Twine有效地渲染到缓冲区中，因此避免了为中间字符串结果生成临时值的成本——特别是在从不需要Twine结果的情况下。通过显式跟踪叶节点的类型，我们还可以避免为转换操作创建临时字符串（例如，将整数附加到字符串）。  
Twine不打算直接使用，也不应该存储，它的实现依赖于存储指向临时堆栈对象的指针的能力，这些对象会在在语句末尾取消分配。当API希望接受可能连接的字符串时，只能将Twins用作参数中的常量引用。

Twine支持特殊的“null”值，它总是可以连接到表单本身，并呈现为空字符串。这This can be returned from APIs to effectively nullify any concatenations performed on the result。

**实现**

由于Twine的性质，Twine的串联方法不可能构造内部节点；结果必须在返回值内表示。因此，一个Twine对象实际上包含两个值，即串联的左侧和右侧。我们还有空的Twine对象，它们实际上是表示空字符串的 sentinel values 。

因此，Twine可以有效地有零个、一个或两个孩子。\see isNullary（）、\see isUnary（）和\see isBinary（）谓词用于测试孩子的数量。

我们在Twine对象上维护了许多不变量（FIXME:为什么）：

-Nullary twines总是在左侧表示其种类，而在右侧表示空的种类。

-Unary twines线始终用左侧的值表示，右边是空的。

-如果一个Twine有另一个Twine作为child，则该child应始终是二进制的（否则它可能已折叠到parent中）。

这些不变量请参见isValid（）。

**效率考虑**

twines的设计目的是为常见情况生成高效且小的代码。因此，concat（）方法被内联，以便连接叶节点的操作可以被优化为直接存储到单个堆栈分配的对象中。

实际上，并非所有的编译器都可以完全优化concat（），因此我们提供了两种额外的方法：（以及伴随的运算符+重载），以确保特别重要的情况（cstring加StringRef）按需生成代码。

### 构造函数

|  |  |
| --- | --- |
| 函数声明 | 解释 |
| Twine() | Construct from an empty string |
| Twine(const Twine &) | 默认复制构造函数 |
| Twine(const char \*Str) | 通过输入字符串构造Twine;如果字符串是’\0’,LHSKind是EmptyKind;否则，LHSKind是CStringKind，且LHS.cString为Str。 |
| Twine(std::nullptr\_t) | 禁用 |
| Twine(const std::string &Str) | 通过std::string构造Twine;LHSKind是StdStringKind，LHS.stdString是指向Str的指针。 |
| Twine(const std::string\_view &Str) | 通过std::string\_view构造Twine，并且把它转换成一个指针和一个length。LHS.ptrAndLength.ptr指向std::string\_view的data()，LHS.ptrAndLength.length等于std::string\_view的length() |
| Twine(const StringRef &Str) | 通过StringRef构造Twine。LHS.ptrAndLength.ptr指向StringRef.的data()，LHS.ptrAndLength.length等于StringRef.的size() |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# support 代码解析

## 数据结构

### ModRef 相关

ModRef.h

说明：ModRefInfo和MemoryEffects的定义，用于描述指令的memory effects。

#### ModRefInfo

#### MemoryEffects

### TrailingObjects相关

说明：此文件定义对实现类的支持，这些类附加了一些trailing object（或对象数组）。主要目的是明确这个idiom的使用位置，并用法更惯用，更难出错。

trailing object模板抽象出用于分配和访问的reinterpret\_cast、指针算法和大小计算的附加对象数组，并注意它们都按所需的对齐方式分配。此外，它还确保基类型是final - 派生自一个类，该类期望在通常不安全之后立即追加数据。

用户应从此模板派生，并为除最后一个类型之外的每种trailing 类型提供 numTrailingObjects 实现，例如，像这个实例：

|  |
| --- |
| /// \code  class VarLengthObj : private TrailingObjects<VarLengthObj, int, double> {  friend TrailingObjects;  unsigned NumInts, NumDoubles;  size\_t numTrailingObjects(OverloadToken<int>) const { return NumInts; }  };  /// \endcode |

您可以通过“getTrailingObjects”访问附加的数组，并通过“additionalSizeToAlloc”和“totalSizeToAlloc”确定分配所需的大小。

此类实现的所有方法都旨在供类的实现使用，而不是作为其接口的一部分（因此，建议私有继承）。

### TimeProfiler 相关

#### TimeTraceProfilerEntry

文件：TimeProfiler.cpp

说明：

#### TimeTraceProfilerInstances

文件：TimeProfiler.cpp

说明：

#### TimeTraceProfiler

文件：TimeProfiler.cpp

说明：

#### TimeTraceScope

文件：TimeProfiler.h

说明：TimeTraceScope 是一个辅助类，用于函数调用的开始和结束的时间跟踪分析器。构造对象时，它开始该部分；当它被摧毁时，它就会停止。如果未初始化时间分析器，则开销为单个分支。

### PrettyStackTrace相关

文件：Support/PrettyStackTrace.h

说明：该文件定义了PrettyStackTraceEntry 类，该类在程序崩溃时，提供更多的上下文信息。

#### PrettyStackTraceEntry

说明：此类用于表示程序崩溃时转储的“pretty”堆栈跟踪的帧。您可以定义它的子类并在程序堆栈上声明它们：当它们被构造和析构时，它们会将它们的符号框架添加到虚拟堆栈跟踪中。如果程序崩溃，这将被dumpout。

属性：

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| NextEntry | PrettyStackTraceEntry \* |  |

### GenericDomTree相关

文件：GenericDomTree.h

说明：该文件定义了一组模板，可以在通用图上有效地计算支配树。这通常在 LLVM 中用于对 CFG 进行快速支配查询，但它对于underlying graph typesgraph types 是完全通用的w.r.t。

与 ADT/\* 图算法不同，通用支配树对图的 NodeRef 有更多的要求。 NodeRef 应该是一个指针，或者 NodeRef->getParent() 必须返回 parente node, 该node 也是指针或 DomTreeNodeTraits 需要专门化。

FIXME：也许 GenericDomTree 需要一个 TreeTraits，而不是 GraphTraits。

#### DomTreeNodeBase

## 全局变量和静态变量

### TimeProfiler 相关

#### Instances

static TimeTraceProfilerInstances Instances;

#### TimeTraceProfilerInstance

static LLVM\_THREAD\_LOCAL TimeTraceProfiler \*TimeTraceProfilerInstance = nullptr;

## 接口

### TargetSelect相关接口

#### InitializeNativeTarget

声明：bool InitializeNativeTarget()

说明：InitializeNativeTarget - 主程序应调用此函数来初始化与主机对应的本机目标。 这对于 JIT 应用程序很有用，可确保目标以正确的方式链接。

客户端多次调用此函数是合法的。

#### InitializeNativeTargetAsmPrinter

声明：bool InitializeNativeTargetAsmPrinter()

说明：主程序应调用此函数来初始化本机目标 asm printer。

#### InitializeNativeTargetAsmParser

声明：bool InitializeNativeTargetAsmParser()

说明：主程序应调用此函数来初始化本机目标 asm parser。

# IR代码解析

## 数据结构

### Type

文件：Type.h

说明：Type 类的实例是不可变的：它们一旦被创建，就永远不会改变。另请注意，只会创建特定类型的一个实例。因此，查看两种类型是否相等是进行简单的指针比较的问题。为了强制不创建两个相等的实例，只能通过类 Type 和派生类中的静态工厂方法创建 Type 实例。一旦分配，类型永远不会被释放。

### FunctionType

文件：DerivedTypes.h

说明：标识funtion type，继承于Type。

### Use

文件：Use.h

说明：Use 表示值定义与其用户之间的一条边。

这在名义上是一个二维链表。它支持遍历特定值定义的所有用途。它也支持，当我们从用户的操作数到达时，端口直接跳转到已使用的值;当我们从值的使用中到达时，端口直接跳转到用户。

### DataLayout相关数据结构

文件：DataLayout.h

说明：此文件定义与数据类型大小/偏移量/对齐方式信息相关的布局属性。 它使用惰性批注来缓存有关如何布局和使用结构类型的信息。

此结构应该创建一次，如果默认值不正确，则填写，然后由 const&传递。 任何成员函数都不需要修改对象。

#### DataLayout

说明：目标数据布局字符串的解析版本以及用于查询它的方法。 目标数据布局字符串由target\*指定。 需要生成 LLVM IR 的前端才能生成被codegen的目标的正确的目标数据。

### Value相关

文件：Value.h，GlobalValue.h，ValueHandle.h

#### Value

文件：Value.h

说明：LLVM 值表示。这是一个非常重要的 LLVM 类。它是程序计算的所有值的基类，可用作其他值的操作数。Value是其他重要类的超类，例如，Instruction和Function。所有值都有一个类型。类型不是Value的子类。某些值可以具有名称，并且它们属于某个Module。 在值上设置名称会自动更新模块的符号表。

每个值都有一个“使用列表”，用于跟踪哪些其他值正在使用此值。 一个值还可以具有任意数量的 ValueHandle 对象，这些对象watch并侦听 RAUW 和销毁事件。 有关详细信息，请参阅 llvm/IR/ValueHandle.h。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
|  |  |  |
| NumUserOperands :27 | unsigned | The number of operands in the subclass |
| IsUsedByMD : 1 | unsigned |  |
| HasName : 1 | unsigned |  |
| HasMetadata : 1 | unsigned |  |
| HasHungOffUses : 1 | unsigned |  |
| HasDescriptor : 1 | unsigned |  |

#### User

文件：User.h

说明：此类定义使用 Value 的人必须实现的接口。Value 类的实例跟踪用户对其处理的内容。

Instructions 是最大的Users class。Constants可能是其他Constants的Users（想想数组之类的）

操作：

* unsigned getNumOperands() const

获取Value的NumUserOperands属性。

#### GlobalValue

文件：GlobalValue.h

说明：该类是所有全局可定义对象的公共基类。例如，它由 GlobalVariable、GlobalAlias 和 Function 子类化。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
|  |  |  |
| Parent | Module \* | The containing module |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

操作：

* Error materialize()

Make sure this GlobalValue is fully read.在Globals.cpp 文件中实现。

* Module \*getParent()

Get the module that this global value is contained inside of...

* bool isDeclaration() const

如果此global value 的主定义在当前 translation unit 之外，则返回 true。

* 如果是GlobalVariable对象，判断Value中的NumUserOperands属性是否为0，如果是0，则为declare。
* 如果是Function对象，则判断BasicBlocks是否为空且Function的isMaterializable操作返回为false。

#### ValueHandleBase

#### GlobalObject

文件：GlobalObject.h

说明：它标识一个独立的对象。它可以是一个function或一个全局变量，但是，不能是一个别名。

#### GlobalVariable

文件：GlobalObject.h

说明：Global variables是常量指针，指向由 VM 或静态编译器中的链接器分配的大块空间。全局变量可能有一个初始值，它被复制到可执行的 .data 区域。全局常量需要有initializers。

### OperandTraits相关类

文件：User.h，OperandTraits.h

说明：该文件定义了特征类，这些特征类有助于强制执行各种用户子类的正确布局。它还提供了以最有效的方式访问操作数的方法。

特别说明：OperandTraits 主模板类是在User.h上声明的(template <class> struct OperandTraits;)。主模板类只声明没有定义，如果用到主模板类，会是不完整类型，在编译时会报“xxx has incomplete type and cannot be defined”，见https://en.cppreference.com/w/cpp/language/type#Incomplete\_type。

#### FixedNumOperandTraits

声明：template <typename SubClass, unsigned ARITY> struct FixedNumOperandTraits；

说明：FixedNumOperandTraits - 当 Use 数组是 User 对象的前缀时，确定 Use 数组的分配机制，并且 Use 对象的数量在编译时已知。

#### OptionalOperandTraits

声明：template <typename SubClass, unsigned ARITY> struct FixedNumOperandTraits；

说明：OptionalOperandTraits - 当操作数的数量可能在运行时发生变化时。本质上，它可能只会减少，因为分配可能不会改变。

#### VariadicOperandTraits

声明：template <typename SubClass, unsigned MINARITY = 0> struct VariadicOperandTraits；

说明：VariadicOperandTraits - 当 Use 数组是 User 对象的前缀时，确定 Use 数组的分配机制，并且 Use 对象的数量仅在分配时已知。

#### HungoffOperandTraits

声明：template <unsigned MINARITY = 1> struct HungoffOperandTraits；

说明：HungoffOperandTraits - 当 Use 数组不是 User 对象的前缀，但分配在不相关的堆地址时，确定 Use 数组的分配机制。

这是 Use 数组必须可调整大小时所需的特征类。

#### DECLARE\_TRANSPARENT\_OPERAND\_ACCESSORS 宏定义

说明：在特定class内声明OperandTraits相关的操作；

#### DEFINE\_TRANSPARENT\_OPERAND\_ACCESSORS 宏定义

说明：在特定class内实现DECLARE\_TRANSPARENT\_OPERAND\_ACCESSORS 宏定义声明的OperandTraits相关的操作。

### Constant相关数据结构

文件：Constant.h Constants.h

#### Constant

说明：这是 LLVM 中的一个重要基类。它提供了 LLVM 程序中所有常量值的通用设施。常量是在运行时不可变的值。函数是常量，因为 EIR 地址是不可变的。与全局变量相同。

所有常量共享此类中提供的功能。所有常量都可以具有空值。他们可以有一个操作数列表。常量可以是简单的（整数和浮点值）、复杂的（数组和结构）或基于表达式（仅由某些运算符和其他常量值组成的常量值的计算产生）。

说明：常量是不可变的（一旦创建，它们永远不会改变），并且完全由结构等价共享。 这意味着两个结构上等效的常量将始终具有相同的地址。常量是根据需要按需创建的，永远不会被删除：因此客户端不必担心对象的生存期。

类操作：

* static Constant \*getBinOpIdentity(unsigned Opcode, Type \*Ty,bool AllowRHSConstant = false,bool NSZ = false)

返回二进制操作码的identity常量。当二元运算可交换时， 对于每个 X，identity常数 C 定义为 X op C = X 和 C op X = X。如果 binop 不可交换，调用者通过将 AllowRHSConstant 设置为 true 获取操作数 1 identity常量。例如，对于操作数 1，任何移位都有一个零identity常量：X 移位 0 = X。

如果这是一个 fadd/fsub 操作，我们不关心带符号的零，然后将 NSZ 设置为 true 返回标识 +0.0 而不是 -0.0。如果运算符没有identity常量，则返回 nullptr。

#### ConstantData

#### ConstantFP

说明：浮点值，float或double。

类操作：

* static ConstantFP \*get(LLVMContext &Context, const APFloat &V)

从Context.pImpl->FPConstants中获取指定的ConstantFP 对象，如果没有，则创建一个。

* static Constant \*get(Type \*Ty, double V)

对于指定类型的指定值，这将返回一个 ConstantFP 或包含一个 ConstantFP的splat 的向量。这应该只用于已知合法的简单常量值，如 2.0/1.0 等，既作为主机 double 又作为目标格式。

* static Constant \*get(Type \*Ty, const APFloat &V)

如果 Ty 是向量类型，则返回具有给定值的 splat 的常量。否则返回给定值的 ConstantFP对象。

#### ConstantExpr

说明：使用其他常量值的表达式初始化的常量值。

此类使用标准指令操作码来定义各种常量表达式。 ConstantExpr 类的 Opcode 字段在 Value::SubclassData 字段中维护。

类操作：

* static Constant \*get(unsigned Opcode, Constant \*C1, Constant \*C2, unsigned Flags = 0, Type \*OnlyIfReducedTy = nullptr);

### Comdat

略，https://maskray.me/blog/2021-07-25-comdat-and-section-group。

What is a Comdat object?

A Comdat section is a section in the object file, in which objects are placed, which can be duplicated in other object files. Each object has information for the linker, indicating what it must do when duplicates are detected. The options can be: Any — do anything, ExactMatch — duplicates must completely match, otherwise an error occurs, Largest — take the object with the largest value, NoDublicates — there should not be a duplicate, SameSize — duplicates must have the same size, otherwise an error occurs.

COMDAT段（Section）是对8086目标文件原始集合的扩展。 它是在Microsoft C 7.0添加的。

### BasicBlock

文件：BasicBlock.h

说明：该类表示 LLVM 中的单个基本块。基本块只是顺序执行的指令的容器。基本块是Values，因为它们被指令引用，例如， branches和switchtables。 BasicBlock 的类型是“Type::LabelTy”，因为基本块表示分支可以跳转到的标签。

well-formed 的基本块由non-terminating指令后跟一个terminator 指令的列表组成。terminator 指令不能出现在基本块的中间，必须终止块。 BasicBlock 类允许出现格式错误的基本块，因为它可能在构建或修改程序的中间阶段很有用。然而，验证者将确保基本块是“well formed”。

### 指令类相关

文件：Instruction.h， Instructions.h

#### Instruction

说明：该类是所有LLVM指令的基类。

#### ReturnInst

说明： Return a value (possibly void), from a function. Execution does not continue in this function any longer.

#### 基于ReturnInst显示特化OperandTraits

#### InstVisitor 类

文件：InstVisitor.h

说明：指令访问者的基类。

当您想要对不同类型的指令执行不同的操作而不必使用大量强制转换和大的 switch 语句（在你的代码里，就是这样）时，可以使用指令访问者。要定义您自己的访问者，请继承此类，为“SubClass”模板参数指定您的新类型，并“override”您类中的 visitXXX 函数。我说“override”是因为使用此类是根据静态解析重载而不是虚函数定义的。

例如，这里是一个计算处理的 malloc 指令数的访问者：

|  |
| --- |
| /// Declare the class. Note that we derive from InstVisitor instantiated  /// with \_our new subclasses\_ type.  ///  struct CountAllocaVisitor : public InstVisitor<CountAllocaVisitor> {  unsigned Count;  CountAllocaVisitor() : Count(0) {}  void visitAllocaInst(AllocaInst &AI) { ++Count; }  }; |

该类可以按如下方式

|  |
| --- |
| CountAllocaVisitor CAV;  CAV.visit(function);  NumAllocas = CAV.Count; |

定义的类具有对指令的“visit”方法，也适用于 BasicBlock、Function 和 Module，它们递归地处理所有包含的指令。

请注意，如果您不为某些指令类型实现 visitXXX，指令 super 的 visitXXX 方法

类将被调用。因此，如果将来添加指令，如果您处理其中一个超类，它们将自动得到支持。

可选的第二个模板参数指定类型，构造访问函数应该返回该该类型。如果你指定这个类型，你\*必须\*提供 visitInstruction 的实现！。

注意这个类是专门设计成模板来避免虚函数调用开销。定义和使用 InstVisitor 与在指令操作码上使用自己的 switch 语句一样高效。

### Function相关

#### Argment

文件：Argment.h

说明：此类表示函数的传入formal 参数。formal参数，因为它是“formal”，不包含实际值，而是表示特定函数参数的类型、参数编号和属性。当在所述函数的主体中使用时，参数当然代表调用函数的实际参数的值。

#### Function

文件：Function.h

说明：表示 LLVM 中的单个函数/过程。

一个函数基本上由基本块列表、参数列表和符号表组成。

属性及解释：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 解释 |
| BasicBlocks | SymbolTableList<BasicBlock> | 基本块列表 |
| Arguments | mutable Argument \* | 参数 |
| NumArgs | size\_t | 参数个数 |
| SymTab | std::unique\_ptr<ValueSymbolTable> | 参数/指令的symbol 表 |
| AttributeSets | AttributeList | 参数属性。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

类操作及解释：

* static Function \*Create(FunctionType \*Ty, LinkageTypes Linkage,unsigned AddrSpace, const Twine &N = "", Module \*M = nullptr)

static Function \*Create(FunctionType \*Ty, LinkageTypes Linkage,const Twine &N, Module &M)

新建一个function，把function放置在module的数据layout指定的程序地址空间中。

对象操作：

* unsigned getInstructionCount() const;

返回该Function内 non-debug IR 指令的个数。它等价于该Function内各个basic block的size之和。

* void print(raw\_ostream &OS, AssemblyAnnotationWriter \*AAW,bool ShouldPreserveUseListOrder = false,bool IsForDebug = false) const;

文件：Function.h，AsmWriter.cpp

说明：Utility functions for printing and dumping Module objects

### LLVMContext相关

#### LLVMContextImpl

文件：lib/IR/LLVMContextImpl.h

说明：the opaque implementation of LLVMContext.

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 解释 |
|  |  |  |
| DiagHandler | std::unique\_ptr<DiagnosticHandler> |  |
|  |  |  |

#### LLVMContext

文件：LLVMContext.h

说明：这是在线程上下文中使用 LLVM 的重要类。 它（不透明地）拥有并管理LLVM核心基础设施的核心“global”数据，包括类型和常量单表。

LLVMContext 本身不提供锁定保证，因此您应该注意每个线程都有一个上下文。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 解释 |
| pImpl | LLVMContextImpl \*const |  |
|  |  |  |
|  |  |  |

操作：

* const DiagnosticHandler \*getDiagHandlerPtr()

获取LLVMContextImpl类型pImpl 属性中的DiagnosticHandler类型的DiagHandler属性的普通指针。

### Module

文件: Module.h

说明：模块实例用于存储与 LLVM 模块相关的所有信息。模块是所有其他 LLVM 中间表示 （IR） 对象的顶级容器。每个模块直接包含全局变量列表、函数列表、此模块所依赖的库（或其他模块）列表、符号表以及有关目标特征的各种数据。

模块维护一个 GlobalList 对象，该对象用于保存对模块中全局变量的所有常量引用。 销毁全局变量时，全局列表中不应包含任何条目。

LLVM 中间表示形式的主容器类。

属性及解释：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 解释 |
|  |  |  |
|  |  |  |
|  |  |  |
| ValSymTab | std::unique\_ptr<ValueSymbolTable> | 全局变量和函数标识的symbol 表 |
| Materializer | std::unique\_ptr<GVMaterializer> | Used to materialize GlobalValues。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

操作及解释：

* GlobalValue \*Module::getNamedValue(StringRef Name)

在model symbol表中返回指定名称的第一个全局值，可以是任何的类型。如果该名称没有找到对应的全局值，则返回null。

* LLVMContext &getContext()

Get the global data context.returns LLVMContext - a container for LLVM's global information。返回当前Module对象的Context属性。

* Function \*Module::getFunction(StringRef Name)

在module symbol 表中查找特定的函数，如果不存在，则返回null。

* const ValueSymbolTable &getValueSymbolTable() const { return \*ValSymTab; }

获取全局变量和函数标识的symbol 表。

* llvm::Error materialize(GlobalValue \*GV)

Make sure the GlobalValue is fully read.如果Materializer不是nullptr，则调用它的materialize方法对global value进行materialize。

* bool shouldEmitInstrCountChangedRemark()

Return true if size-info optimization remark is enabled, false otherwise.

### IRBuilder相关

文件：IRBuilderFolder.h， IRBuilder.h

#### IRBuilderFolder

说明：IRBuilder 中常量折叠的接口。

#### ConstantFolder

说明：ConstantFolder - 创建具有最小、目标独立、折叠的常量。

操作：

* Value \*FoldBinOp(Instruction::BinaryOps Opc, Value \*LHS,

Value \*RHS) const override

* Value \*FoldBinOpFMF(Instruction::BinaryOps Opc, Value \*LHS, Value \*RHS,

FastMathFlags FMF) const override

#### IRBuilderBase

说明：各种IRBuilders的通用基类。

操作：

* Value \*CreateFAdd(Value \*L, Value \*R, const Twine &Name = "",

MDNode \*FPMD = nullptr)

* CallInst \*CreateConstrainedFPBinOp(

Intrinsic::ID ID, Value \*L, Value \*R, Instruction \*FMFSource = nullptr,

const Twine &Name = "", MDNode \*FPMathTag = nullptr,

std::optional<RoundingMode> Rounding = std::nullopt,

std::optional<fp::ExceptionBehavior> Except = std::nullopt)

#### IRBuilder

文件：IRBuilder.h

说明：这提供了一个统一的 API，用于创建指令并将其插入基本块：在 BasicBlock 的末尾，或在块中的特定迭代器位置。

请注意，builder不会公开 LLVM 指令的完整通用性。 要访问额外的指令属性，请在创建指令后在指令上使用mutators（例如 setVolatile）。存在方便状态来指定fast-math标志和 fp-math 标记。

第一个模板参数指定用于创建常量的类。这默认为创建最小折叠的常量。 第二个模板参数允许客户端指定在每次新创建的插入时调用的自定义插入挂钩。

操作：

* Value \*CreateFAdd(Value \*L, Value \*R, const Twine &Name = "",MDNode \*FPMD = nullptr)
* ReturnInst \*CreateRet(Value \*V)

创建一个'ret <val>' 指令。

### Pass相关的类

文件：PassManager.h PassInstrumentation.h

说明：

* PassManager.h，此文件定义了用于 LLVM 中pass management的各种接口。LLVM本身没有“pass”接口。相反，任何类的实例支持在 IR 单元上“run”它的一个方法都可以用作一个pass。pass management通常是一种工具，用于收集在特定 IR 构造上运行的传递序列(a sequence of passes)，并在包含IR构造的每个此类构造上按顺序运行每个传递。由于模块没有包含 IR 构造，因此用于模块传递的管理器形成了基本情况，该基本情况在提供的单个模块上按顺序运行其管理的传递。

core IR lib提供了用于运行模块和函数传递的管理器。

* \* FunctionPassManager可以在一个模块上运行，在函数上运行每个pass。
* \* ModulePassManager必须直接运行，运行模块上的每个pass。

请注意，pass management的实现使用基于概念的多态性，如“Value Semantics and Concept-based”讨论中所述（or its abbreviated sibling "Inheritance Is The Base Class of Evil"）：

* <http://github.com/sean-parent/sean-parent.github.com/wiki/Papers-and-Presentations>
* http://www.youtube.com/watch?v=\_BpMYeUFXv8
* http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil

#### PassInstrumentationCallbacks

文件：IR/PassInstrumentation.h

#### PassInfoMixin

文件：PassManager.h

#### AnalysisInfoMixin

文件：PassManager.h

#### AnalysisManager

文件：PassManager.h

### LegacyPassManager相关类

文件：LegacyPassManager.h

说明：该文件定义了Legacy PassManager class。此类用于保存、维护和优化passes的执行。 PassManager class确保分析结果在一个pass运行之前可用，并且当 PassManager 被销毁时，Pass 也被销毁。

文件：include/llvm/IR/LegacyPassManagers.h

说明：。

#### PassManagerPrettyStackEntry

说明：这用于在/如果生成堆栈跟踪时打印有关正在运行的过程的信息。

属性：

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| P | Pass \* |  |
| V | Value\* |  |
| M | Module\* |  |

#### PMStack

文件：include/llvm/IR/LegacyPassManagers.h

说明：PMStack - 此类实现了 PMDataManager 指针的堆栈数据结构。

Top level pass managers（请参阅 PassManager.cpp）使用 PMStack 维护 active Pass Managers。每个 Pass 实现 assignPassManager() 以将自身与适当的管理器连接起来。 assignPassManager() 遍历 PMStack 来找到合适的manager。

属性：

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| S | std::vector<PMDataManager \*> | active Pass Managers |

操作：

#### PMTopLevelManager

文件：include/llvm/IR/LegacyPassManagers.h

说明：PMTopLevelManager 管理 LastUser 信息并收集 top level pass managers 使用的通用 API。

成员：

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| PassManagers | SmallVector<PMDataManager \*, 8> | Collection of pass managers |
| IndirectPassManagers | SmallVector<PMDataManager \*, 8> | ollection of pass managers that are not directly maintained by this pass manager |
| ImmutablePassMap | SmallDenseMap<AnalysisID, ImmutablePass \*, 8> | Map from ID to immutable passes. |
| AnUsageMap | DenseMap<Pass \*, AnalysisUsage\*> | Maps from a pass to it's associated entry in UniqueAnalysisUsages.Does not own the storage associated with either key or value.. |
| UniqueAnalysisUsages | FoldingSet<AUFoldingSetNode> | 包含 AnalysisUsage 的所有 unique combinations。当我们有多个相同pass 的实例时，这很有用，因为它们通常具有相同的 analysis usage 并且可以共享存储。 |
| activeStack | PMStack | Active Pass Managers |

构造函数：

* PMTopLevelManager::PMTopLevelManager(PMDataManager \*PMDM)

操作：

* void addImmutablePass(ImmutablePass \*P)

Add immutable pass and initialize it。调用P的initializePass()方法(当前该方法，do nothing)。把当前P 存放到ImmutablePasses 和 ImmutablePassMap属性中。调用findAnalysisPassInfo 方法，获取到PassInfo。

* void addPassManager(PMDataManager \*Manager)
* const PassInfo \*findAnalysisPassInfo(AnalysisID AID) const;

Retrieve the PassInfo for an analysis。基于AID获取PassInfo。

1. 从AnalysisPassInfos属性中获取该AID对应的PassInfo，如果为nullptr，则执行下一步；如果成功则执行第三步；
2. 从Pass注册表 PassRegistryObj中获取该AID对应的PassInfo，进入第四步。
3. 从Pass注册表 PassRegistryObj中获取该AID对应的PassInfo与第一步获取的PassInfo进行比对，看是否一致，如果不一致，assert挂起，如果一致进入第四步。
4. 返回获取的PassInfo。

* Pass \*findAnalysisPass(AnalysisID AID)

查找实现了Analysis AID的pass。搜索immutable passes和所有的pass managers。如果没有找到对应的Pass，则返回NULL。

1. 从ImmutablePassMap中查找，如果找到，则返回该Pass。
2. 从PassManagers中查找，如果找到，则返回该Pass。
3. 从IndirectPassManagers中查找，如果找到则返回该Pass。
4. 返回nullptr

* AnalysisUsage \*findAnalysisUsage(Pass \*P)

为pass P 查找 analysis usage information。

1. 从AnUsageMap中查找P对应的<Pass \*, AnalysisUsage\*>表项；如果找到，从该表项中获取AnalysisUsage对象，并返回；
2. 如果没有找到，在调用该P的getAnalysisUsage 方法(注意，这个getAnalysisUsage 方法是虚函数，因此不同子类Pass会override该方法)来获取AnalysisUsage对象的信息。例如，当前P是InstructionCombiningPass类型，InstructionCombiningPass::getAnalysisUsage方法会获取到该Pass相关的Passes，有AAResultsWrapperPass，AssumptionCacheTracker等Passes添加到AnalysisUsage 的Required属性中，DominatorTreeWrapperPass、AAResultsWrapperPass等Passes添加到AnalysisUsage 的Preserved属性中，添加LazyBlockFrequencyInfoPass相关的Passes等等。
3. 从UniqueAnalysisUsages 查找该pass对应的“unique combinations of AnalysisUsage”；
4. 如果没有找到，就创建一个AUFoldingSetNode类型对象插入到UniqueAnalysisUsages ；
5. 把该AUFoldingSetNode类型对象的AU属性地址返回。

* inline SmallVectorImpl<ImmutablePass \*>& getImmutablePasses()

获取当前ImmutablePasses属性；

* unsigned getNumContainedManagers() const

获取当前PassManagers的个数；

* void initializeAllAnalysisInfo()

遍历PassManagers中的所有entry，调用PMDataManager的initializeAnalysisInfo操作；遍历IndirectPassManagers中的所有entry，调用PMDataManager的initializeAnalysisInfo操作；

* void schedulePass(Pass \*P)

调度 pass P 执行。确保在运行 P 之前运行 P 所需的passes。更新由管理器维护的分析信息。去除dead passes。这是一个递归函数。

1. 调用P的preparePassManager，给该pass一个机会来准备的阶段。默认情况下，do nothing。
2. 调用findAnalysisPassInfo函数，找到当前P的PassInfo类型对象PI。
3. 如果找到了PI，且当前PI是analysis，且调用findAnalysisPass找到了对应的Pass，则从AnUsageMap中删除该Pass，释放该Pass，返回。否则，进入下一步。
4. 调用findAnalysisUsage，为pass P 查找 analysis usage information，返回AnalysisUsage类型AnUsage。
5. 循环操作，while checkAnalysis 为True，执行如下操作：
6. 设置checkAnalysis 为True。
7. 调用AnUsage的getRequiredSet，获取required passes的vector。
8. 遍历该required passes的vector，执行如下操作：
9. 调用findAnalysisPassInfo，获取当前pass对应的passInfo对象PI。
10. 如果PI为null，这是异常情况。调用findAnalysisPass，搜索immutable passes和所有的pass managers，如果没有找到，则报错；如果找到，同样，报错。
11. 如果PI不为null，调用PI的createPass方法，默认是callDefaultCtor函数，创建一个Pass对象AnalysisPass。
12. 调用Pass的getPotentialPassManagerType()方法，获取管理该Pass的Pass Manager的类型和AnalysisPass的Pass Manager的类型。
13. 如果两个Pass的Pass manager 的类型一致，要调用递归的schedulePass 方法，输入是 AnalysisPass。
14. 如果Pass P的Pass Manager的类型 大于 AnalysisPass的Pass Manager的类型，则要调用递归的schedulePass 方法，输入是 AnalysisPass，且当前checkAnalysis设置为true。
15. 如果Pass P的Pass Manager的类型 小于 AnalysisPass的Pass Manager的类型，则删除该AnalysisPass。
16. 现在所有required passes都是可用的了。调用P的getAsImmutablePass 方法，如果P是ImmutablePass对象，则执行如下
17. 调用getAsPMDataManager 方法()，得到PMDataManager类型对象DM。
18. 基于DM创建AnalysisResolver类型对象AR。
19. 调用P的setResolver，设置resolver属性。
20. DM调用它的initializeAnalysisImpl，设置required passes的<AnalysisID, Pass\*>到当前P的AnalysisResolver的AnalysisImpls中。
21. 调用addImmutablePass方法，将ImmutablePass对象添加到ImmutablePasses 和 ImmutablePassMap属性中。
22. 调用DM的recordAvailableAnalysis方法，将P添加到DM的AvailableAnalysis属性中。
23. 返回。
24. 如果PI不是ImmutablePass对象，并且调用shouldPrintBeforePass()返回为true，则执行如下操作：
25. 调用P的createPrinterPass方法，创建一个新的Pass对象PP。
26. 调用PP的assignPassManager方法，Find appropriate Function Pass Manager or Call Graph Pass Manager in the PM Stack and add self into that manager。
27. 调用P的assignPassManager方法，Find appropriate Function Pass Manager or Call Graph Pass Manager in the PM Stack and add self into that manager。
28. 如果PI不是ImmutablePass对象，并且调用shouldPrintAfterPass()返回为true，则执行类似于调用shouldPrintBeforePass()返回为true的操作。
29. 返回。

#### PMDataManager

文件：include/llvm/IR/LegacyPassManagers.h

说明：PMDataManager provides the common place to manage the analysis data used by pass managers.。

成员：

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| TPM | PMTopLevelManager \* | top level manager |
| PassVector | SmallVector<Pass \*, 16> | 当前manager管理的pass的集合 |
| InheritedAnalysis[PMT\_Last] | DenseMap<AnalysisID, Pass \*> \* | 由Parent pass manager提供并由当前pass manager使用的分析集合。在一个时间不能有超过 PMT\_Last 的active pass mangers。 |
| AvailableAnalysis | DenseMap<AnalysisID, Pass\*> | 一组可用的分析。此信息在调度passes时使用。如果一个pass需要一个不可用的分析，那么所需的analysis pass 被调度在该pass被调度运行之前运行。 |
| HigherLevelAnalysis | SmallVector<Pass \*, 16> | Collection of higher level analysis used by the pass managed by this manager. |
| Depth | unsigned |  |
|  |  |  |

操作：

* void emitInstrCountChangedRemark(Pass \*P, Module &M, int64\_t Delta, unsigned CountBefore, StringMap<std::pair<unsigned, unsigned>> &FunctionToInstrCount,Function \*F = nullptr);
* DenseMap<AnalysisID, Pass\*> \*getAvailableAnalysis()

返回该PMDataManager对象的AvailableAnalysis属性的地址。

* unsigned getNumContainedPasses()

获取PassVector中entry的个数

* void initializeAnalysisInfo()

初始化可用的 analysis information。把AvailableAnalysis中的表项清空，把InheritedAnalysis中的指针数组中表项都设置为nullptr；

* void initializeAnalysisImpl(Pass \*P)

所有必需的分析都应该在pass运行时对pass可用！这里我们填充pass的AnalysisImpls成员，这样它就可以成功的使用getAnalysis()方法来获取它需要的实现。

* unsigned initSizeRemarkInfo( Module &M, StringMap<std::pair<unsigned, unsigned>> &FunctionToInstrCount)

如果user指定了他们想要标记大小，设置Module的初始大小。如果不需要标记，则返回0；

* void populateInheritedAnalysis(PMStack &PMS)

Collect AvailableAnalysis from all the active Pass Managers.把PMS中包含的所有PMDataManager对象的AvailableAnalysis存放到当前PMDataManager对象的InheritedAnalysis属性中。

* void recordAvailableAnalysis(Pass \*P);

Augment AvailableAnalysis by adding analysis made available by pass P.

* void removeDeadPasses(Pass \*P, StringRef Msg, enum PassDebuggingString);

Remove analysis passes that are not used any longer

* void removeNotPreservedAnalysis(Pass \*P);

Remove Analysis that is not preserved by the pass

* void verifyPreservedAnalysis(Pass \*P);

verifyPreservedAnalysis -- Verify analysis presreved by pass P.

#### FPPassManager

文件：include/llvm/IR/LegacyPassManagers.h

说明：FPPassManager manages BBPassManagers and FunctionPasses.

It batches all function passes and basic block pass managers together and sequence them to process one function at a time before processing next function.

操作：

* bool doInitialization(Module &M) override

doInitialization - 为function passes 运行所有的initializers .

* virtual PMDataManager \*getAsPMDataManager()

当前FPPassManager对象作为PMDataManager 对象返回。

* FunctionPass \*getContainedPass(unsigned N)

基于索引N，从PassVector中获取FunctionPass

* bool runOnFunction(Function &F)

通过调用 runOnFunction 方法执行调度执行的所有passes。跟踪是否有任何pass修改了函数，如果是，返回True。基本流程如下：

* 调用GlobalValue::isDeclaration函数，如果该函数返回为true，则直接返回false。因为只有声明的Function是不能优化的。
* 调用GlobalValue::getParent函数，获取该Function对象F对应的Module对象M。
* 调用PMDataManager::populateInheritedAnalysis，输入是TPM的activeStack。遍历activeStack中的 active pass managers(即它包含的所有PMDataManager对象)，调用PMDataManager::getAvailableAnalysis,获取该对象的AvailableAnalysis 属性，填充到InheritedAnalysis属性中。
* 调用M的Module::shouldEmitInstrCountChangedRemark操作，获取是否进行size-info的优化。默认是false。
* 如果需要进行size-info的优化，则调用PMDataManager::initSizeRemarkInfo函数，调用Function::getInstructionCount获取该Function的non-debug IR 指令的个数。
* 调用PMDataManager::getNumContainedPasses操作，获取PassVector中entry的个数。
* 遍历PassVector中entry的个数，进行如下的操作：

1. getContainedPass操作，输入是当前pass entry的索引，获取FunctionPass类型对象FP。
2. PMDataManager::initializeAnalysisImpl ，输入是FP，查看所有需要的Passes是否都已经实现了。
3. 默认EXPENSIVE\_CHECKS 没有定义
4. 在当前FP上调用runOnFunction 函数，输入为当前的Function。这里就是个递归调用的位置，这里会调用不同的FP。
5. 如果需要进行size-info的优化，调用Function::getInstructionCount获取该Function的non-debug IR 指令的个数。如果新的大小不等于之前的大小，则调用PMDataManager::emitInstrCountChangedRemark方法。
6. 调用PMDataManager::verifyPreservedAnalysis方法，Verify analysis presreved by pass P。
7. 如果没有change，则调用PMDataManager::removeNotPreservedAnalysis方法，Remove Analysis not preserved by Pass P。
8. PMDataManager::recordAvailableAnalysis，Augment AvailableAnalysis by adding analysis made available by pass P.
9. 调用PMDataManager::removeDeadPasses 方法，Remove analysis passes that are not used any longer

* 返回是否change。
* StringRef getPassName() const override

返回pass名称为"Function Pass Manager"。

#### MPPassManager

文件：LegacyPassManager.cpp

说明：MPPassManager manages ModulePasses and function pass managers.

It batches all Module passes and function pass managers together and sequences them to process one module.

操作：

* StringRef getPassName() const override

返回pass名称为"Module Pass Manager"。

#### legacy::PassManagerBase

操作：

* virtual void add(Pass \*P) = 0

将pass添加到要运行的pass队列中。这会将 Pass 的所有权传递给 PassManager。当PassManager被销毁时，pass也会被销毁，所以不需要删除pass。如果发现pass是多余的，甚至可能会立即销毁该pass。这意味着所有pass都必须通过”new”来分配。

#### legacy::FunctionPassManager

操作：

* void add(Pass \*P) override

调用FunctionPassManagerImpl的add操作

* bool doInitialization()

调用FunctionPassManagerImpl的doInitialization操作

* bool run(Function &F)

执行计划执行的所有passes。跟踪是否有任何passes修改了函数，如果是，则返回 true。调用Function的materialize操作(Function是GlobalValue的子类，实际上，调用的是GlobalValue的materialize操作)，然后调用FunctionPassManagerImpl 的run操作。

#### FunctionPassManagerImpl

操作：

* void add(Pass \*P)

调用PMTopLevelManager::schedulePass操作。

* bool doInitialization()

doInitialization - Run all of the initializers for the function passes

* FPPassManager \*getContainedManager(unsigned N)

基于索引N，获取PassManagers中对应的entry。

* bool run(Function &F)

执行该top level manager管理的所有passes。如果任何一个pass修改了函数，则返回 true。

* 调用PMTopLevelManager的initializeAllAnalysisInfo操作，初始化分析信息。
* 调用PMTopLevelManager的PMTopLevelManager::getNumContainedManagers操作，获取当前pass managers的个数。
* 遍历所有的pass managers，执行如下操作：

调用getContainedManager方法，

### GVMaterializer

文件：GVMaterializer.h

#### GVMaterializer

说明：此接口允许增量或随机访问从文件中加载的函数。这对于 JIT 编译器或过程间优化器等不需要同时将整个程序存储在内存中的应用程序很有用。

操作：

* virtual Error materialize(GlobalValue \*GV) = 0

当前为空函数。

### Diagnostic相关类

#### DiagnosticHandler

文件：DiagnosticHandler.h

说明：Base DiagnosticHandler class declaration. Derive from this class to provide custom diagnostic reporting.

操作：

* virtual bool isAnalysisRemarkEnabled(StringRef PassName) const

Return true if analysis remarks are enabled, override to provide different implementation.

#### OptimizationRemarkAnalysis

### DominatorTree相关类

文件：Support/GenericDomTree.h，Dominators.h

说明：该文件定义了一组模板，可以在通用图上有效地计算支配树。这通常在 LLVM 中用于对 CFG 进行快速支配查询，但它是完全通用的 w.r.t. 底层graph类型。

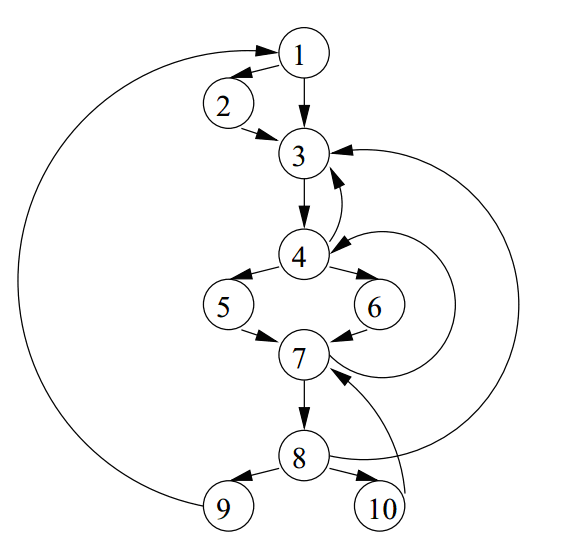
与 ADT/\* 图算法不同，通用支配树对图的 NodeRef 有更多的要求。 NodeRef 应该是一个指针，并且 NodeRef->getParent() 必须返回父节点，父节点也是一个指针或 DomTreeNodeTraits 需要专门化。

FIXME：也许 GenericDomTree 需要一个 TreeTraits，而不是 GraphTraits。

#### 支配树简介

<https://zhuanlan.zhihu.com/p/586372481>

domtree（支配树）是优化之基础。所谓“dominate（支配）”，即：在flowgraph（流图，这里指的是带root（根）的directed graph（有向图））中，若节点a支配节点b，则从入口节点至b则必经a，常记作a dom b。遵循拿来主义，copy龙书中的图（其实整段都是copy的（滑稽））：



假设1是入口节点，则1支配图中的所有节点（这是毋庸置疑的，因为去其他节点必从入口节点进入）。同时，也包括它自己，也就是每个节点都“支配它本身”。即1 dom 1。

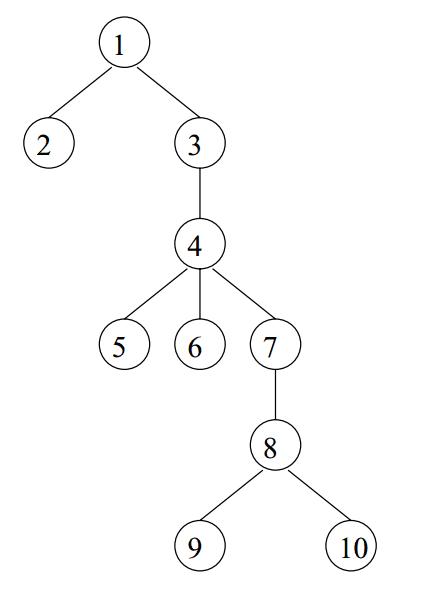
节点2：除了自己，只有1直接流向它，即路径1->2，因此1/2 dom 2。

节点3：除了自己，有1->3，1->2->3两条路径。这两条路径都有节点1、3，说明必经这两节点，所以，1/3 dom 3。

节点4：除了自己，有1->2->3->4与1->3->4两条路径。两条路径中都有节点1、3、4，说明必经这3个节点，所以1/3/4 dom 4。

不难得出，节点a的dominators，即为从入口节点至a的每条路径中节点的交集。

以此类推，将每个节点的支配关系连接起来就有了下图（依然出自龙书）：



即dominator tree（支配树）！

转换到编译原理的语境中，若给出的是个control-flow graph（控制流图，简作CFG），则dominator tree描述的是basic block之间的支配关系。root即函数入口的basic block，方向则指明执行调用关系。

（在某种意义上来说，难道basic block不是一种解耦的手段吗？分割basic block的条件wikipedia总结的很好了，在clang的codegen中搜索所有的createBasicBlock(...)函数以及所创basic block的插入点，配合wikipedia食用即可基本摸清。）

immediate dominator：简记作idom。immediate有“离……最近”的意思。若a idom b，则a是b的dominators中，离b最近且非b自身的那一个dominator。如上支配树中，8 idom 9/10、7 idom 8等等。

不难看出，在dominator tree中，parent（父节点）是children（子节点）的immediate dominator。

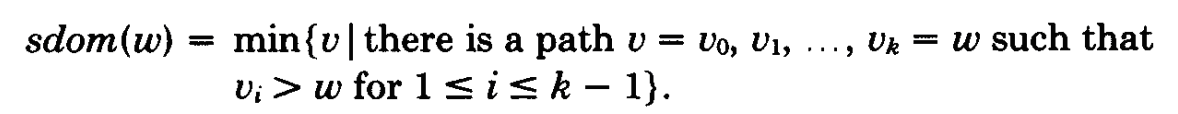
dominator tree的实现方法有很多。在LLVM中，D34258补丁之前采用的是Simple Lengauer-Tarjan算法，此后为Semi-NCA算法（该算法是LT算法的改良，区别在于如何利用semidominator构造dominator tree）。作者给出的解释是：后者相对前者有轻微的性能提升（理论上SLT复杂度低，但实践经验是大多情况下Semi-NCA稍快），且便于拓展出对已构建的支配树进行增量式更新（incremental update）。

其中，Semi即semidominator的缩写，NCA则为nearest common ancestors（最近公共祖先）的缩写。

欲得semidominator，需要先以DFS（depth-first-search，深度优先遍历）的方式过一遍原flowgraph，构造一棵对应的spanning tree（生成树），遍历过程中按序对节点（有文献中也称vertex顶点）进行编号（也作DFS number）。直观上讲，也就是每次都在同一方向上一条道走到黑，走到最深处撞南墙再往回返。

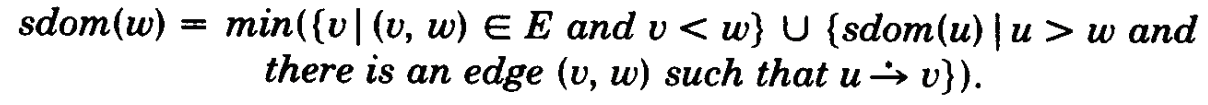
文献中描述的算法采用preorder（先序）遍历顺序，即先遍历当前vertex，再沿着child一直走，遇到遍历过的vertex或叶节点，再回溯改走其他child。待到全图遍历完，DFS算法终止。

经DFS后，对于flowgraph G中的vertex w，若w != r（即非root），则我们称sdom(w)的值为w的semidominator，其定义为（出自[LT79]）：



即，若存在一条由vertex v0, v1, ...,vk构成的路径（亦称semidominator path），其中v = v0为起始vertex，vk = w为终止vertex。且除了起始vertex v = v0与终止vertex vk = w之外，v1, ..., vk-1的DFS number都大于w的DFS number，则sdom(w)为v0, ..., vk中DFS number的最小值（其实就是v0本人罢了）。

[LT79]中还给出了定理4及其证明：对于任意vertex w != r（即非根vertex），有



即，sdom(w)取如下两个集合并集中的最小值：

1. 1） 集合1中的vertex v满足：v与w存在v->w（原图中的）这条边，且v的dfs number < w的dfs number。（也就是说，v是w在dfs spanning tree中的parent）
2. 2）若集合2中的vertex u，其dfs number > w的dfs number，且存在一条边v -> w（虽未标明是tree edge还是原graph edge，但是不难推测出指的是原graph中的edge），满足u是v在dfs spanning tree中的ancestor（祖先，这里指的是proper ancestor（非真祖先），也就是可能u = v。若dfs spanning tree中存在两vertex v与w，存在路径v可达w，则称v是w的ancestor），则集合中取sdom(u)。（概括之，即若存在w在原graph上的parent v。从v回溯tree edge（包括v本身），途中满足dfs number > w的dfs number的vertex u，以sdom(u)作为集合2中的值。回溯至u的dfs number <= w的dfs number时，这条path的回溯终止）

很明显，其实理论4即semidominator定义的实现手段，便于实践。

依理论4分别计算出根r之外其他vertex的semidominator。

接下来便利用semidominator求取immediate dominator，从而构成dominator tree。这里就是Semi-NCA与LT的差别。

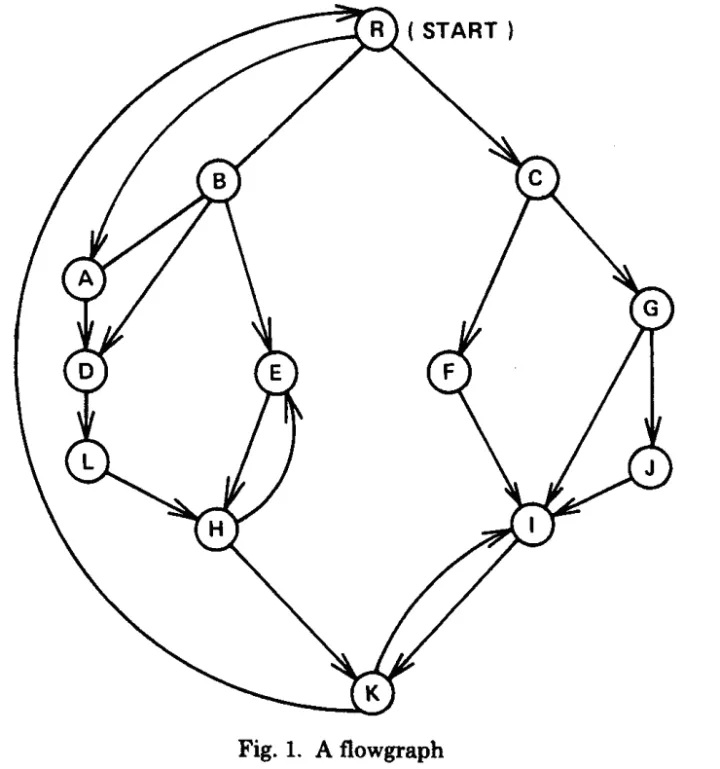
为了便于表述，这里引用llvm中Semi-NCA算法求idom的公式：

IDom[i] = NCA(SDom[i], SpanningTreeParent(i))

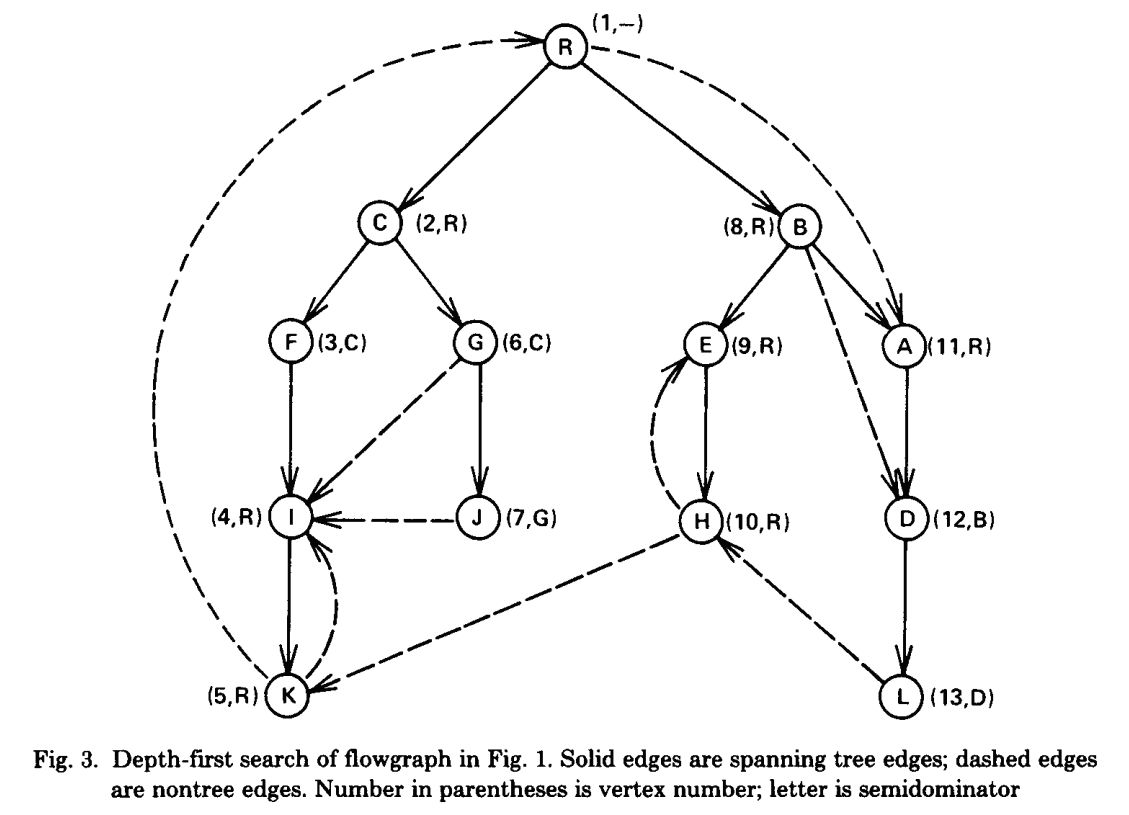
即，若要得到vertex i（1为root的dfs number，以此类推）的idom，方法是求取SDom[i]与i在dfs spanning tree上parent的NCA。NCA在上文提到过——最近公共祖先。顾名思义，若dfs spanning tree上存在两vertex v与w，则NCA(v, w)为v、w离根r最远的共有ancestor。

每个vertex以各自的idom为链条，构成了一棵完整的dominator tree。

现在用[TL79]给出的示例进行演示。

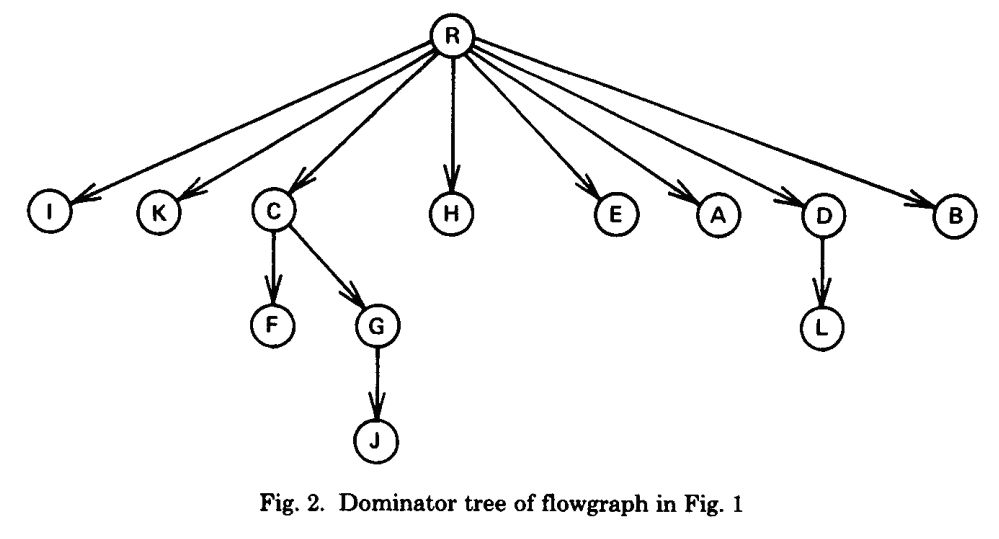


原flowgraph



原flowgraph经dfs遍历所得。实线edge为dfs spanning tree edge；虚线edge为原flowgraph edge中的非tree edge。括弧中的数字为该vertex的dfs number；括号中的字母为其semidominator。

* 首先进行dfs的preorder遍历。
* R为起点，首先遍历R，将R的dfs number（后简记作dfsnum）标记为1。
* 遍历R的child C，将其dfsnum标记为2。
* 遍历C的child F，将其dfsnum标记为3。
* 遍历F的child I，将其dfsnum标记为4。
* 遍历I的child K，将其dfsnum标记为5。
* K的child为R与I，都已遍历过。
* 回溯到I，唯一的child K已经遍历过。
* 同理，继续回溯，直到C，其child G还未遍历，因而将G的dfsnum标记为6。
* 遍历G的child J，将其dfsnum标记为7。
* J的child为I，已经遍历过，回溯。
* 同理，回溯直到R，其child B未遍历，将其标记为8。（图中有几处箭头未标，但可从下图推断粗来）
* 遍历B的child E，将其dfsnum标记为9。
* 遍历E的child H，将其dfsnum标记为10。
* ……
* 直至回溯到原点R发现无点可遍历，算法终止。
* 步骤二，计算root外各vertex的sdom。
* 计算C的sdom
* 指向C的只有root R，且R dfsnum < C dfsnum（1 < 2），所以集合1中仅有一个值1。
* 考虑其集合2。R指向C，但是R的dfsnum 1 < C的dfsnum 2。因此，R不在集合2中。在dfs spanning tree上，R未有parent，因为虚线为原graph edge，而非tree edge，因此集合2为空。
* 综上所述，有sdom(C) = 1（即R）。
* 同理，sdom(F) = 2（即C），sdom(G) = 2（即C），sdom(J) = 6（即G），sdom(B) = 1（即R），sdom(L) = 12（即D）。
* 对于A，集合1中有B与R，集合2为空。
* 为何A的集合2为空？指向A有两条edge，B->A与R->A。接下来看B与R在dfs spanning tree上的precessor。R没有这种precessor，原因很简单，实线是tree edge，虚线是原graph edge，R只有K在原图中有条edge指向它。考虑到集合里提到的是ancestor，所以存在R即是其ancestor的可能。但是R的dfsnum为1，小于A的dfsnum 11，所以R不是集合2中所述的u。B的tree edge为R，但是其dfsnum为1，小于B的dfsnum 8，所以不作数。考虑到集合里提到的是ancestor，所以存在B即是其ancestor的可能。但是B的dfsnum为8，小于A的dfsnum 11，所以B也不是集合2中所述的u。因而集合2为空。
* 这样，在集合1中取最小值，即sdom(A) = 1（即R）。
* D与A的情况相似，有B->D，A->D。又B与A的回溯路径vertex均不满足集合2的条件，因而集合2为空。集合1取最小值，B的dfsnum 8小于A的dfsnum 11，所以sdom(D) = 8（即B）。
* 对于H，集合1中仅有E。
* 对于H的集合2，有E->H与L->H。但是E回溯方向上的vertex dfsnum均小于E的dfsnum，所以不考虑此条路径。再来看L回溯方向，tree edge经过的节点依次为L、D、A。由于B的dfsnum 8小于H的dfsnum 10，所以集合2中的u只能为L、D与A，这几个vertex的dfsnum均大于H的dfsnum，且皆为H的ancestor。因而集合2中为L、D、A的semidominator，即{D, B, R}，换算成值为{12, 8, 1}。
* 取集合1与集合2并集的最小值，即sdom(H) = 1。
* 以此类推求得除root外，各vertex的semidominator。
* 接下来利用Semi-NCA法计算除root外，各vertex的immediate dominator。
* 考虑C。sdom(C) = R，SpanningTreeParent(C) = R（即在dfs spanning tree上C的parent）。则NCA(C)只能是R。
* 考虑J。sdom(J) = G，SpanningTreeParent(J) = G，因而NCA(J) = G。
* 以此类推，求取除root外，各vertex的immediate dominator。从而构成一棵美丽的dominator tree，见下图。



原flowgraph的对应dominator tree

参考文献：

[LT79] A Fast Algorithm for Finding Dominators in a Flowgraph. THOMAS LENGAUER and ROBERT ENDRE TARJAN

Linear-Time Algorithms for Dominators and Related Problems. Loukas Georgiadis

#### DominatorTreeWrapperPass

### Verifier相关类

文件：

#### DominatorTree

说明：用于计算normal dominator tree的 DominatorTreeBase 的具体子类。

定义：如果存在从函数入口到一个block的路径，则称该块是前向静态可达的。在优化过程中，静态可访问块可能会变得静态不可访问。

前向不可达块可能出现在支配树中，也可能不出现。如果是这样，支配查询将返回

结果好像所有可达的块都支配它。当请求与潜在无法访问的块对应的节点时，调用代码必须处理块无法访问的情况以及 getNode()的结果是 nullptr 的情况。、

通常，在构建支配树时，已知不可达的块不会在树中。在最初构造支配树后变得无法访问的块可能会一直存在于树中，即使树被正确更新。调用代码不应依赖于前面的语句；这只是为了帮助人类理解。

具体解释见《编译原理》中的机器无关代码优化部分。

### AsmWriter 相关类

文件：ModuleSlotTracker.h, AsmWriter.cpp

#### ModuleSlotTracker

文件：ModuleSlotTracker.h

说明：slot tracker 存储的抽象接口。

#### TypePrinting

#### SlotTracker

## 静态变量或全局变量

### Diagnostic相关

#### PassRemarksPassedOptLoc

static PassRemarksOpt PassRemarksPassedOptLoc;

#### PassRemarksMissedOptLoc

static PassRemarksOpt PassRemarksMissedOptLoc;

#### PassRemarksAnalysisOptLoc

static PassRemarksOpt PassRemarksAnalysisOptLoc;

## 接口

### ConstantFold相关接口

#### ConstantFoldBinaryInstruction

Constant \*llvm::ConstantFoldBinaryInstruction(unsigned Opcode, Constant \*C1,Constant \*C2)

### Verifier相关接口

#### verifyFunction

## 相关处理流程

# Analysis相关代码解析

Analysis pass计算相关IR单元的高层信息，但不对其进行修改。这些信息可以被其他pass使用，或用于调试和程序可视化。简言之，Analysis pass提供其它pass需要查询的信息并提供查询接口。例如，basic-aa pass是基本别名分析（Basic Alias Analysis）pass，得到的别名分析结果可以用于后续的其它优化pass。Analysis pass不仅从IR中得到有用信息，还可以通过调用其它Analysis pass得到信息，并将这些信息结合起来，得到关于IR更有价值的信息。这些分析结果可以被缓存下来，直到分析的IR被修改，原有的分析结果当然也就失效了。

## 数据结构

### AliasAnalysis相关

文件：AliasAnalysis.h

该文件定义了通用的AliasAnalysis接口，作为所有别名分析信息客户端使用的通用接口，由“所有别名分析实现”来实现。“Mod/Ref 信息”也被这个接口捕获。

此接口的实现必须实现各种虚拟方法，这些方法自动为整套客户端 APIs 提供功能。

这个API 使用 MemoryLocation 类标识内存区域。指针组件指定区域的内存地址的基址。 Size 指定内存区域的最大大小（in address units），或者如果大小未知的话，使用MemoryLocation::UnknownSize。TBAA 标签标识内存引用的“类型”；有关详细信息，请参阅 TypeBasedAliasAnalysis 类。

一些不明显的细节包括：

指向内存中两个完全不同的对象的指针永远不会别名，无论 Size 组件的值如何。

NoAlias 并不意味着不相等的指针。最明显的例子是指向“constant memory”的两个指针。即使它们相等，也永远不会存储到常量内存，因此永远不会有任何依赖关系。在这种和其他情况下，指针可能同时是 NoAlias 和 MustAlias。当前的 API 只能返回一个结果，尽管这在实践中很少成为问题。

#### BasicAAWrapperPass

#### AAResultsWrapperPass

### AssumptionCache 相关

文件：

说明：

### BranchProbability相关

#### LazyBranchProbabilityInfoPass

### BlockFrequency相关

#### LazyBlockFrequencyInfoPass

### LoopInfo相关

#### LoopInfoWrapperPass

### MemoryDependenceAnalysis相关

#### MemoryDependenceWrapperPass

### OptimizationRemarkEmitter相关

#### OptimizationRemarkEmitterWrapperPass

### PostDominatorTree

#### PostDominatorTreeWrapperPass

### ScalarEvolution相关

#### ScalarEvolutionWrapperPass

### TargetLibraryInfo相关

TargetLibraryInfo.h

说明：(个人理解)包含如下信息，

1. 当前使用的cpu-arch、cpu-sub-arch、cpu厂商、OS、Environmenttype(例如，gnu等)、目标文件类型。
2. 当前环境下支持的数据类型和标准库中支持的API。
3. 判断当前的Function是否是个标准库中的函数。

#### TargetLibraryInfoWrapperPass

操作：

* TargetLibraryInfo &getTLI(const Function &F)

调用TargetLibraryAnalysis对象的run操作，返回TargetLibraryInfo 对象。

# Transforms相关代码解析

Transform pass可以使用Analysis pass。Transform pass会检视IR，查询Analysis pass得到IR的高层信息，然后以某种方式改变和优化IR，并保证改变后的IR仍然合法有效。例如，adce pass是激进的死代码消除（Aggressive Dead Code Elimination）pass，会将死代码从原来的模块中删除。

## 数据结构

### InstCombine相关

文件：InstCombiner.h、InstCombine.h

说明：

* InstCombiner.h --- 该文件为 instcombine pass 实现提供了接口。该接口用于此文件夹中的通用转换和目标中的目标特定组合。visitor 实现在 InstCombineInternal.h 的 InstCombinerImpl 中。
* InstCombine.h ---该文件提供了 instcombine pass 的主要接口。此 pass 适合在新的 pass manager 中使用。对于与 legacy pass manager 一起使用的 pass，请使用 createInstructionCombiningPass()。

#### InstructionCombiningPass

文件：InstCombine.h

说明：This is a basic whole-function wrapper around the instcombine utility. It will try to combine all instructions in the function。

属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| Worklist | InstructionWorklist |  |
| MaxIterations | const unsigned |  |
| ID | static char | Pass identification, replacement for typeid |

操作：

* void getAnalysisUsage(AnalysisUsage &AU) const override;

该pass要用到的其它Pass 分析的信息。 把需要的Pass加入到AU的

该Pass 依赖的Pass有：

* AAResultsWrapperPass
* AssumptionCacheTracker
* TargetLibraryInfoWrapperPass
* TargetTransformInfoWrapperPass
* DominatorTreeWrapperPass
* GlobalsAAWrapperPass
* OptimizationRemarkEmitterWrapperPass
* LazyBlockFrequencyInfoPass
* ProfileSummaryInfoWrapperPass

### Scalar 相关

#### CFGSimplifyPass

文件：lib/Transforms/Scalar/SimplifyCFGPass.cpp，

#### GVNLegacyPass

文件：lib/Transforms/Scalar/GVN.cpp

#### LowerConstantIntrinsics

#### MergeICmpsLegacyPass

#### PartiallyInlineLibCallsLegacyPass

#### ReassociateLegacyPass

文件：lib/Transforms/Scalar/Reassociate.cpp

#### SimplifyCFGOptions

文件：lib/Transforms/Utils/SimplifyCFGOptions.h

说明：A set of parameters used to control the transforms in the SimplifyCFG pass.

### Utils 相关

#### InstructionWorklist

#### LoopSimplifyPass

## 全局变量和静态变量

### InstCombine相关

统计相关的全局变量：

* NumWorklistIterations ：Number of instruction combining iterations performed
* NumCombined ： Number of insts combined
* NumConstProp ： Number of constant folds
* NumDeadInst ： Number of dead inst eliminated
* NumSunkInst ： Number of instructions sunk
* NumExpand ： Number of expansions
* NumFactor ： Number of factorizations
* NumReassoc ： Number of reassociations
* VisitCounter ： instcombine-visit

循环和迭代次数：

* InstCombineDefaultMaxIterations
* InstCombineDefaultInfiniteLoopThreshold

## 接口

### InstCombine相关接口

#### createInstructionCombiningPass

原型：FunctionPass \*llvm::createInstructionCombiningPass()

FunctionPass \*llvm::createInstructionCombiningPass(unsigned MaxIterations)

说明：创建一个InstructionCombiningPass实例。

#### initializeInstructionCombiningPassPassOnce

原型：static void \*initializeInstructionCombiningPassPassOnce(PassRegistry &Registry)

说明：

1. 对它所依赖的其它Passes，调用对应的initialize##depName##Pass 函数进行初始化操作。
2. 创建PassInfo类型对象。

### Scalar 相关接口

#### createReassociatePass

原型：FunctionPass \*llvm::createReassociatePass()

说明：创建一个ReassociateLegacyPass实例。

#### createGVNPass

原型：FunctionPass \*llvm::createGVNPass(bool NoMemDepAnalysis)

说明：创建一个GVNLegacyPass实例。

#### createCFGSimplificationPass

原型：FunctionPass \*llvm::createCFGSimplificationPass(SimplifyCFGOptions Options,std::function<bool(const Function &)> Ftor)

说明：创建一个CFGSimplifyPass实例。

# ExecutionEngine 相关代码解析

## 数据结构

### orc ExecutorProcessControl相关结构体

文件：ExecutionEngine/Orc/ExecutorProcessControl.h

说明：支持与一个JIT 目标进程之间进行交互。

#### ExecutorProcessControl

说明：支持与一个JIT 目标进程之间进行交互。

### Orc core相关结构体

文件：llvm/include/llvm/ExecutionEngine/Orc/core.h

#### ExecutionSession

说明：一个ExecutionSession标识一个running JIT program。

## 接口

# CodeGen相关代码解析

## 数据结构

### AssignmentTrackingAnalysis

#### AssignmentTrackingAnalysis

### AtomicExpandPass

#### AtomicExpand

### BreakFalseDeps

#### BreakFalseDeps

### BranchFolding

#### BranchFolderPass

### CFIInstrInserter

#### CFIInstrInserter

### EdgeBundles

#### EdgeBundles

### ExpandLargeDivRem

#### ExpandLargeDivRemLegacyPass

### ExpandLargeDivRemLegacyPass

#### ExpandLargeDivRemLegacyPass

### ExpandLargeFpConvert

#### ExpandLargeFpConvertLegacyPass

### ExpandMemCmp

#### ExpandMemCmpPass

### ExpandPostRAPseudos

#### ExpandPostRA

### ExpandReductions

#### ExpandReductions

### ExpandVectorPredication

#### ExpandVectorPredication

### FEntryInserter

#### FEntryInserter

### FinalizeISel

#### FinalizeISel

### FuncletLayout

#### FuncletLayout

### GCRootLowering

#### GCMachineCodeAnalysis

### IndirectBrExpandPass

#### IndirectBrExpandPass

### LazyMachineBlockFrequencyInfo

#### LazyMachineBlockFrequencyInfoPass

### LiveDebugVariables

#### LiveDebugVariables

### LiveRegMatrix

#### LiveRegMatrix

### LiveStacks

#### LiveStacks

### LiveVariables

#### LiveVariables

### LocalStackSlotAllocation

#### LocalStackSlotPass

### MachineBlockPlacement

#### MachineBlockPlacement

### MachineCopyPropagation

#### MachineCopyPropagation

### MachineCSE

#### MachineCSE

### MachineCycleAnalysis

#### MachineCycleInfoWrapperPass

### MachineDominators

#### MachineDominatorTree

### MachineDominanceFrontier

#### MachineDominanceFrontier

### MachineInstrBundle

#### UnpackMachineBundles

### MachineLateInstrsCleanup

#### MachineLateInstrsCleanup

### MachineLICM

#### EarlyMachineLICM

#### MachineLICM

### MachineLoopInfo

#### MachineLoopInfo

### MachineOptimizationRemarkEmitter

#### MachineOptimizationRemarkEmitterPass

#### MachineOptimizationRemarkAnalysis

### MachinePostDominators

#### MachinePostDominatorTree

### MachineTraceMetrics

#### MachineTraceMetrics

### MachineSanitizerBinaryMetadata

#### MachineSanitizerBinaryMetadata

### MachineScheduler

#### MachineScheduler

### MachineSink

#### MachineSinking

### OptimizePHIs

#### OptimizePHIs

### PrologEpilogInserter

#### PEI

### RegisterCoalescer

#### RegisterCoalescer

### RemoveRedundantDebugValues

#### RemoveRedundantDebugValues

### ReplaceWithVeclib

#### ReplaceWithVeclibLegacy

### SafeStack

#### SafeStackLegacyPass

### ShadowStackGCLowering

#### ShadowStackGCLowering

### SlotIndexes

#### SlotIndexes

### SpillPlacement

#### SpillPlacement

### StackColoring

#### StackColoring

### StackMapLivenessAnalysis

#### StackMapLiveness

### StackProtector

#### StackProtector

### TailDuplication

#### EarlyTailDuplicate

### UnreachableBlockElimLegacy

#### UnreachableBlockElimLegacyPass

### DeadMachineInstructionElim

#### DeadMachineInstructionElim

### LiveIntervals

#### LiveIntervals

### PatchableFunction

#### PatchableFunction

### PeepholeOptimizer

#### PeepholeOptimizer

### PHIElimination

#### PHIElimination

### PostRASchedulerList

#### PostRAScheduler

### ProcessImplicitDefs

#### ProcessImplicitDefs

### ReachingDefAnalysis

#### ReachingDefAnalysis

### TailDuplication

#### TailDuplicate

### TwoAddressInstructionPass

#### TwoAddressInstructionPass

### UnreachableBlockElim

#### UnreachableMachineBlockElim

### VirtRegMap

#### VirtRegMap

### XRayInstrumentation

#### XRayInstrumentation

## 接口

# Target parse相关代码解析

文件：Triple.h

## 数据结构

### TargetParser相关的枚举类型

Triple::ArchType 枚举cpu arch类型

Triple::SubArchType 枚举cpu arch子类型

Triple::VendorType 枚举cpu厂商类型

Triple::OSType 枚举操作系统类型

Triple::EnvironmentType 枚举运行环境类型，例如，GNU/NetBSD有着不同的用户空间。它们以linux-gnu风格来呈现。

Triple::ObjectFormatType 枚举目标文件的类型

### Triple

属性

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 说明 |
| Data | string |  |
| Arch | ArchType | cpu arch 类型 |
| SubArch | SubArchType | cpu arch 子类型 |
| Vendor | VendorType | 厂商 |
| OS | OSType | 操作系统 |
| Environment | EnvironmentType | 运行环境类型， |
| ObjectFormat | ObjectFormatType | 目标文件类型。 |
|  |  |  |
|  |  |  |

# Target相关代码解析

## 数据结构

### X86LowerAMXType

#### X86LowerAMXTypeLegacyPass

### X86ISelDAGToDAG

#### X86DAGToDAGISel

## 接口

### X86相关

#### LLVMInitializeX86Target

#### LLVMInitializeX86AsmPrinter

# 参考文档