目 录

[1. 概述 1](#_Toc743953495)

[1.1 术语 1](#_Toc1851771196)

[2. llvm command line解析llvm::cl workspace 1](#_Toc1413475537)

[2.1 相关的接口函数 1](#_Toc88960026)

[2.2 相关全局变量 2](#_Toc868395403)

[2.2.1 GlobalParser 2](#_Toc406169374)

[2.2.2 GeneralCategory 2](#_Toc1644295762)

[2.2.3 TopLevelSubCommand 2](#_Toc1578155956)

[2.2.4 AllSubCommands 2](#_Toc2015150367)

[2.3 相关数据结构 2](#_Toc1899118716)

[2.3.1 ExpansionContext 2](#_Toc168096448)

[2.3.1.1 构造函数 2](#_Toc1601234328)

[2.3.2 CommandLineParser 2](#_Toc650000220)

[2.3.2.1 registerSubCommand 注册subcommand 2](#_Toc382606195)

[2.4 回调函数 2](#_Toc528236658)

[2.4.1 TokenizeGNUCommandLine 2](#_Toc1409748390)

[3. ADT代码解析 3](#_Toc110839382)

[3.1 small vector代码解析 3](#_Toc2102258058)

[3.2 Twine代码解析 3](#_Toc656205441)

[3.2.1 概述 3](#_Toc1646614484)

[3.2.2 构造函数 3](#_Toc1712267594)

[4. IR代码解析 4](#_Toc1159328335)

[4.1 数据结构 4](#_Toc425798016)

[4.1.1 Type 4](#_Toc904419461)

[4.1.2 FunctionType 4](#_Toc280076907)

[4.1.3 Use 4](#_Toc2101574106)

[4.1.4 Value 5](#_Toc1826746952)

[4.1.5 User 5](#_Toc423221271)

[4.1.6 Constant 5](#_Toc1060370229)

[4.1.7 Comdat 5](#_Toc1995176699)

[4.1.8 GlobalValue 5](#_Toc1426788668)

[4.1.9 GlobalObject 5](#_Toc1804323724)

[4.1.10 Function 6](#_Toc1699464248)

[4.1.11 LLVMContext 6](#_Toc692780558)

[4.1.12 Module 6](#_Toc1893283750)

[4.1.13 IRBuilder 7](#_Toc420376003)

[5. 参考文档 7](#_Toc1098949932)

# 概述

## 术语

|  |  |
| --- | --- |
| 术语 | 解释 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 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。 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 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。

## 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() |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 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。

### Value相关

#### 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。

#### 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。

### 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）

### 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。

### 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。

### 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。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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。

### 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。

### 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 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

操作及解释：

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

获取全局变量和函数标识的symbol 表。

* 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。

* 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。

### 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。

### 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。

### Pass相关的类

文件：PassManager.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

#### PassInfoMixin

文件：PassManager.h

#### AnalysisInfoMixin

文件：PassManager.h

### Verifier相关类

#### 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。

## 接口

### ConstantFold相关接口

#### ConstantFoldBinaryInstruction

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

### Verifier相关接口

#### verifyFunction

# 参考文档