目 录

[1. 概述 1](#_Toc1316098713)

[1.1 术语 1](#_Toc238734556)

[2. llvm command line解析llvm::cl workspace 1](#_Toc1218818792)

[2.1 相关的接口函数 1](#_Toc384047640)

[2.2 相关全局变量 2](#_Toc160312817)

[2.2.1 GlobalParser 2](#_Toc394568436)

[2.2.2 GeneralCategory 2](#_Toc1967350879)

[2.2.3 TopLevelSubCommand 2](#_Toc1385566410)

[2.2.4 AllSubCommands 2](#_Toc1931759692)

[2.3 相关数据结构 2](#_Toc363363958)

[2.3.1 ExpansionContext 2](#_Toc1163255973)

[2.3.1.1 构造函数 2](#_Toc106175954)

[2.3.2 CommandLineParser 2](#_Toc1857294027)

[2.3.2.1 registerSubCommand 注册subcommand 2](#_Toc2014978542)

[2.4 回调函数 2](#_Toc327834563)

[2.4.1 TokenizeGNUCommandLine 2](#_Toc1248472684)

[3. ADT代码解析 3](#_Toc1126338384)

[3.1 small vector代码解析 3](#_Toc612341837)

[3.2 Twine代码解析 3](#_Toc1896237670)

[3.2.1 概述 3](#_Toc542478296)

[3.2.2 构造函数 3](#_Toc867420358)

[4. IR代码解析 4](#_Toc1189475679)

[4.1 数据结构 4](#_Toc1257193874)

[4.1.1 重要数据结构汇总及所在文件 4](#_Toc2097579587)

[4.1.2 LLVMContext 4](#_Toc207775163)

[4.1.3 Module 5](#_Toc180911393)

[4.1.4 IRBuilder 5](#_Toc1347568642)

[5. 参考文档 5](#_Toc1118310281)

# 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

## 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。

### 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

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。

# 参考文档