LLVM IR

IR源代码位置：https://github.com/llvm/llvm-project/tree/release/10.x/llvm/include/llvm/IR

Context

LLVMContext类：llvm/include/llvm/IR/LLVMContext.h

- 包含了各种类型、常量表等
- 这个类里面主要实现了各种的API接口，底层实现是LLVMContextImpl类
- 成员中只有`LLVMContextImpl *const pImpl;`

LLVMContextImpl类：llvm/lib/IR/LLVMContextImpl.h

- LLVMContext类的具体实现
- 包含了IR创建过程的各种全局量，如
  - `DenseMap<const Value*, ValueName*> ValueNames;`
  - `ArrayConstantsTy ArrayConstants;`
  - `DenseMap<std::pair<const Function *, const BasicBlock *>, BlockAddress *> BlockAddresses;`
  - `Type VoidTy, LabelTy, HalfTy, FloatTy, DoubleTy, MetadataTy, TokenTy;`
    - Basic type instances.

Module

Module类：llvm/include/llvm/IR/Module.h

- Module是IR的最高级抽象
- 包含所有全局量以及函数

```
/// A Module instance is used to store all the information related to an
/// LLVM module. Modules are the top level container of all other LLVM
/// Intermediate Representation (IR) objects. Each module directly contains a
/// list of globals variables, a list of functions, a list of libraries (or
/// other modules) this module depends on, a symbol table, and various data
/// about the target's characteristics.
///
/// A module maintains a GlobalValRefMap object that is used to hold all
/// constant references to global variables in the module.  When a global
/// variable is destroyed, it should have no entries in the GlobalValueRefMap.
/// The main container class for the LLVM Intermediate Representation.
```

```
LLVMContext &Context;          ///< The LLVMContext from which types and
                                 ///< constants are allocated.
  GlobalListType GlobalList;     ///< The Global Variables in the module
  FunctionListType FunctionList; ///< The Functions in the module
  AliasListType AliasList;       ///< The Aliases in the module
  IFuncListType IFuncList;       ///< The IFuncs in the module
```

```
  NamedMDListType NamedMDList;     ///< The named metadata in the module
  std::string GlobalScopeAsm;      ///< Inline Asm at global scope.
  std::unique_ptr<ValueSymbolTable> ValSymTab; ///< Symbol table for values
  ComdatSymTabType ComdatSymTab;   ///< Symbol table for COMDATs
  std::unique_ptr<MemoryBuffer>
  OwnedMemoryBuffer;               ///< Memory buffer directly owned by this
                                   ///< module, for legacy clients only.
  std::unique_ptr<GVMaterializer>
  Materializer;                    ///< Used to materialize GlobalValues
  std::string ModuleID;            ///< Human readable identifier for the
module
  std::string SourceFileName;      ///< Original source file name for
module,
                                   ///< recorded in bitcode.
  std::string TargetTriple;        ///< Platform target triple Module
compiled on
                                   ///< Format: (arch)(sub)-(vendor)-(sys0-
(abi)
  NamedMDSymTabType NamedMDSymTab;  ///< NamedMDNode names.
  DataLayout DL;                   ///< DataLayout associated with the
module
```

Function

Function类：llvm/include/llvm/IR/Function.h

包含BasicBlocks和Arguments

```
  // Important things that make up a function!
  BasicBlockListType BasicBlocks;         ///< The basic blocks
  mutable Argument *Arguments = nullptr;  ///< The formal arguments
  size_t NumArgs;
  std::unique_ptr<ValueSymbolTable>
      SymTab;                             ///< Symbol table of
args/instructions
  AttributeList AttributeSets;            ///< Parameter attributes
```

BasicBlock

BasicBlock类：llvm/include/llvm/IR/BasicBlock.h

一系列指令的容器，以结束指令作为结尾

```
/// LLVM Basic Block Representation
///
/// This represents a single basic block in LLVM. A basic block is simply a
/// container of instructions that execute sequentially. Basic blocks are
Values
/// because they are referenced by instructions such as branches and switch
/// tables. The type of a BasicBlock is "Type::LabelTy" because the basic
block
/// represents a label to which a branch can jump.
///
/// A well formed basic block is formed of a list of non-terminating
/// instructions followed by a single terminator instruction. Terminator
```

```
/// instructions may not occur in the middle of basic blocks, and must terminate
/// the blocks. The BasicBlock class allows malformed basic blocks to occur
/// because it may be useful in the intermediate stage of constructing or
/// modifying a program. However, the verifier will ensure that basic blocks are
/// "well formed".
```

```
 InstListType InstList;
 Function *Parent;
```

Instruction

Instruction类：llvm/include/llvm/IR/Instruction.h

指令实现的基本类

```
 BasicBlock *Parent;
 DebugLoc DbgLoc;
```

Value

Value类：llvm/include/llvm/IR/Value.h

- 操作数、结果

```
/// LLVM Value Representation
///
/// This is a very important LLVM class. It is the base class of all values
/// computed by a program that may be used as operands to other values. Value is
/// the super class of other important classes such as Instruction and Function.
/// All Values have a Type. Type is not a subclass of Value. Some values can
/// have a name and they belong to some Module.  Setting the name on the Value
/// automatically updates the module's symbol table.
///
/// Every value has a "use list" that keeps track of which other Values are
/// using this Value.  A Value can also have an arbitrary number of ValueHandle
/// objects that watch it and listen to RAUW and Destroy events.  See
/// llvm/IR/ValueHandle.h for details.
```

```
 Type *VTy;
 Use *UseList; //使用这个Value的指令
```

IRBuilder

IRBuilder类：llvm/include/llvm/IR/IRBuilder.h

- helper类，辅助在BasicBlock中插入指令

```
```

```
/// This provides a uniform API for creating instructions and inserting
/// them into a basic block: either at the end of a BasicBlock, or at a specific
/// iterator location in a block.
///
/// Note that the builder does not expose the full generality of LLVM
/// instructions.  For access to extra instruction properties, use the mutators
/// (e.g. setVolatile) on the instructions after they have been
/// created. Convenience state exists to specify fast-math flags and fp-math
/// tags.
///
/// The first template argument specifies a class to use for creating constants.
/// This defaults to creating minimally folded constants.  The second template
/// argument allows clients to specify custom insertion hooks that are called on
/// every newly created insertion.
```

常量传播

- 似乎是在创建指令的时候就进行传播了

```
 Value *CreateICmp(CmpInst::Predicate P, Value *LHS, Value *RHS,
                   const Twine &Name = "") {
   if (auto *LC = dyn_cast<Constant>(LHS))
     if (auto *RC = dyn_cast<Constant>(RHS))
       return Insert(Folder.CreateICmp(P, LC, RC), Name);
   return Insert(new ICmpInst(P, LHS, RHS), Name);
 }
```