# LLVM 程序员手册

* [介绍](https://llvm.org/docs/ProgrammersManual.html" \l "introduction)
* [一般信息](https://llvm.org/docs/ProgrammersManual.html" \l "general-information)
  + [C++ 标准模板库](https://llvm.org/docs/ProgrammersManual.html" \l "the-c-standard-template-library)
  + [其他有用的参考资料](https://llvm.org/docs/ProgrammersManual.html" \l "other-useful-references)
* [重要且有用的 LLVM API](https://llvm.org/docs/ProgrammersManual.html" \l "important-and-useful-llvm-apis)
  + [和模板\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-isa-cast-and-dyn-cast-templates)**[isa<>](https://llvm.org/docs/ProgrammersManual.html" \l "the-isa-cast-and-dyn-cast-templates)**[\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-isa-cast-and-dyn-cast-templates)**[cast<>dyn\_cast<>](https://llvm.org/docs/ProgrammersManual.html" \l "the-isa-cast-and-dyn-cast-templates)**
  + [传递字符串（the](https://llvm.org/docs/ProgrammersManual.html" \l "passing-strings-the-stringref-and-twine-classes)**[StringRef](https://llvm.org/docs/ProgrammersManual.html" \l "passing-strings-the-stringref-and-twine-classes)**[和](https://llvm.org/docs/ProgrammersManual.html" \l "passing-strings-the-stringref-and-twine-classes)**[Twine](https://llvm.org/docs/ProgrammersManual.html" \l "passing-strings-the-stringref-and-twine-classes)**[classes）](https://llvm.org/docs/ProgrammersManual.html" \l "passing-strings-the-stringref-and-twine-classes)
    - **[StringRef](https://llvm.org/docs/ProgrammersManual.html" \l "the-stringref-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-stringref-class)
    - **[Twine](https://llvm.org/docs/ProgrammersManual.html" \l "the-twine-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-twine-class)
  + [格式化字符串（](https://llvm.org/docs/ProgrammersManual.html" \l "formatting-strings-the-formatv-function)**[formatv](https://llvm.org/docs/ProgrammersManual.html" \l "formatting-strings-the-formatv-function)**[函数）](https://llvm.org/docs/ProgrammersManual.html" \l "formatting-strings-the-formatv-function)
    - [简单格式](https://llvm.org/docs/ProgrammersManual.html" \l "simple-formatting)
    - [自定义格式](https://llvm.org/docs/ProgrammersManual.html" \l "custom-formatting)
    - **[formatv](https://llvm.org/docs/ProgrammersManual.html" \l "formatv-examples)**[例子](https://llvm.org/docs/ProgrammersManual.html" \l "formatv-examples)
  + [错误处理](https://llvm.org/docs/ProgrammersManual.html" \l "error-handling)
    - [程序错误](https://llvm.org/docs/ProgrammersManual.html" \l "programmatic-errors)
    - [可恢复的错误](https://llvm.org/docs/ProgrammersManual.html" \l "recoverable-errors)
      * [字符串错误](https://llvm.org/docs/ProgrammersManual.html" \l "stringerror)
      * [与 std::error\_code 和 ErrorOr 的互操作性](https://llvm.org/docs/ProgrammersManual.html" \l "interoperability-with-std-error-code-and-erroror)
      * [从错误处理程序返回错误](https://llvm.org/docs/ProgrammersManual.html" \l "returning-errors-from-error-handlers)
      * [使用 ExitOnError 简化工具代码](https://llvm.org/docs/ProgrammersManual.html" \l "using-exitonerror-to-simplify-tool-code)
      * [使用 cantFail 简化安全调用站点](https://llvm.org/docs/ProgrammersManual.html" \l "using-cantfail-to-simplify-safe-callsites)
      * [易犯错误的构造函数](https://llvm.org/docs/ProgrammersManual.html" \l "fallible-constructors)
      * [基于类型传播和消费错误](https://llvm.org/docs/ProgrammersManual.html" \l "propagating-and-consuming-errors-based-on-types)
      * [使用 joinErrors 连接错误](https://llvm.org/docs/ProgrammersManual.html" \l "concatenating-errors-with-joinerrors)
      * [构建易出错的迭代器和迭代器范围](https://llvm.org/docs/ProgrammersManual.html" \l "building-fallible-iterators-and-iterator-ranges)
  + [传递函数和其他可调用对象](https://llvm.org/docs/ProgrammersManual.html" \l "passing-functions-and-other-callable-objects)
    - [函数模板](https://llvm.org/docs/ProgrammersManual.html" \l "function-template)
    - [类](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-ref-class-template)**[function\_ref](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-ref-class-template)**[模板](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-ref-class-template)
  + **[LLVM\_DEBUG()](https://llvm.org/docs/ProgrammersManual.html" \l "the-llvm-debug-macro-and-debug-option)**[宏和](https://llvm.org/docs/ProgrammersManual.html" \l "the-llvm-debug-macro-and-debug-option)**[-debug](https://llvm.org/docs/ProgrammersManual.html" \l "the-llvm-debug-macro-and-debug-option)**[选项](https://llvm.org/docs/ProgrammersManual.html" \l "the-llvm-debug-macro-and-debug-option)
    - [带有选项](https://llvm.org/docs/ProgrammersManual.html" \l "fine-grained-debug-info-with-debug-type-and-the-debug-only-option)**[DEBUG\_TYPE](https://llvm.org/docs/ProgrammersManual.html" \l "fine-grained-debug-info-with-debug-type-and-the-debug-only-option)**[的细粒度调试信息](https://llvm.org/docs/ProgrammersManual.html" \l "fine-grained-debug-info-with-debug-type-and-the-debug-only-option)**[-debug-only](https://llvm.org/docs/ProgrammersManual.html" \l "fine-grained-debug-info-with-debug-type-and-the-debug-only-option)**
  + **[Statistic](https://llvm.org/docs/ProgrammersManual.html" \l "the-statistic-class-stats-option)**[类别和](https://llvm.org/docs/ProgrammersManual.html" \l "the-statistic-class-stats-option)**[-stats](https://llvm.org/docs/ProgrammersManual.html" \l "the-statistic-class-stats-option)**[选项](https://llvm.org/docs/ProgrammersManual.html" \l "the-statistic-class-stats-option)
  + [添加调试计数器以帮助调试代码](https://llvm.org/docs/ProgrammersManual.html" \l "adding-debug-counters-to-aid-in-debugging-your-code)
  + [在调试代码时查看图表](https://llvm.org/docs/ProgrammersManual.html" \l "viewing-graphs-while-debugging-code)
* [为任务选择正确的数据结构](https://llvm.org/docs/ProgrammersManual.html" \l "picking-the-right-data-structure-for-a-task)
  + [顺序容器（std::vector、std::list 等）](https://llvm.org/docs/ProgrammersManual.html" \l "sequential-containers-std-vector-std-list-etc)
    - [llvm/ADT/ArrayRef.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-arrayref-h)
    - [固定大小的数组](https://llvm.org/docs/ProgrammersManual.html" \l "fixed-size-arrays)
    - [堆分配数组](https://llvm.org/docs/ProgrammersManual.html" \l "heap-allocated-arrays)
    - [llvm/ADT/TinyPtrVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-tinyptrvector-h)
    - [llvm/ADT/SmallVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-smallvector-h)
    - [<矢量>](https://llvm.org/docs/ProgrammersManual.html" \l "vector)
    - [<双端队列>](https://llvm.org/docs/ProgrammersManual.html" \l "deque)
    - [<列表>](https://llvm.org/docs/ProgrammersManual.html" \l "list)
    - [llvm/ADT/ilist.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-ilist-h)
    - [llvm/ADT/PackedVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-packedvector-h)
    - [ilist\_traits](https://llvm.org/docs/ProgrammersManual.html" \l "ilist-traits)
    - [llvm/ADT/ilist\_node.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-ilist-node-h)
    - [哨兵](https://llvm.org/docs/ProgrammersManual.html" \l "sentinels)
    - [其他顺序容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "other-sequential-container-options)
  + [绳状容​​器](https://llvm.org/docs/ProgrammersManual.html" \l "string-like-containers)
    - [llvm/ADT/StringRef.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-stringref-h)
    - [llvm/ADT/Twine.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-twine-h)
    - [llvm/ADT/SmallString.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-smallstring-h)
    - [标准::字符串](https://llvm.org/docs/ProgrammersManual.html" \l "std-string)
  + [类集合容器（std::set、SmallSet、SetVector 等）](https://llvm.org/docs/ProgrammersManual.html" \l "set-like-containers-std-set-smallset-setvector-etc)
    - [一个排序的“向量”](https://llvm.org/docs/ProgrammersManual.html" \l "a-sorted-vector)
    - [llvm/ADT/SmallSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-smallset-h)
    - [llvm/ADT/SmallPtrSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-smallptrset-h)
    - [llvm/ADT/StringSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-stringset-h)
    - [llvm/ADT/DenseSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-denseset-h)
    - [llvm/ADT/SparseSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-sparseset-h)
    - [llvm/ADT/SparseMultiSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-sparsemultiset-h)
    - [llvm/ADT/FoldingSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-foldingset-h)
    - [<设置>](https://llvm.org/docs/ProgrammersManual.html" \l "set)
    - [llvm/ADT/SetVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-setvector-h)
    - [llvm/ADT/UniqueVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-uniquevector-h)
    - [llvm/ADT/ImmutableSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-immutableset-h)
    - [其他类似集合的容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "other-set-like-container-options)
  + [类地图容器（std::map、DenseMap 等）](https://llvm.org/docs/ProgrammersManual.html" \l "map-like-containers-std-map-densemap-etc)
    - [一个排序的“向量”](https://llvm.org/docs/ProgrammersManual.html" \l "dss-sortedvectormap)
    - [llvm/ADT/StringMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-stringmap-h)
    - [llvm/ADT/IndexedMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-indexedmap-h)
    - [llvm/ADT/DenseMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-densemap-h)
    - [llvm/IR/ValueMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-ir-valuemap-h)
    - [llvm/ADT/IntervalMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-intervalmap-h)
    - [llvm/ADT/IntervalTree.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-intervaltree-h)
    - [<地图>](https://llvm.org/docs/ProgrammersManual.html" \l "map)
    - [llvm/ADT/MapVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-mapvector-h)
    - [llvm/ADT/IntEqClasses.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-inteqclasses-h)
    - [llvm/ADT/ImmutableMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "llvm-adt-immutablemap-h)
    - [其他类似地图的容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "other-map-like-container-options)
  + [位存储容器](https://llvm.org/docs/ProgrammersManual.html" \l "bit-storage-containers)
    - [位向量](https://llvm.org/docs/ProgrammersManual.html" \l "bitvector)
    - [小位向量](https://llvm.org/docs/ProgrammersManual.html" \l "smallbitvector)
    - [稀疏位向量](https://llvm.org/docs/ProgrammersManual.html" \l "sparsebitvector)
    - [合并位向量](https://llvm.org/docs/ProgrammersManual.html" \l "coalescingbitvector)
* [调试](https://llvm.org/docs/ProgrammersManual.html" \l "debugging)
* [常见操作的有用提示](https://llvm.org/docs/ProgrammersManual.html" \l "helpful-hints-for-common-operations)
  + [基本检查和遍历例程](https://llvm.org/docs/ProgrammersManual.html" \l "basic-inspection-and-traversal-routines)
    - [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-basicblock-in-a-function)**[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-basicblock-in-a-function)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-basicblock-in-a-function)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-basicblock-in-a-function)**
    - [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-basicblock)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-basicblock)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-basicblock)**[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-basicblock)**
    - [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-function)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-function)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-function)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-the-instruction-in-a-function)**
    - [将迭代器变成类指针（反之亦然）](https://llvm.org/docs/ProgrammersManual.html" \l "turning-an-iterator-into-a-class-pointer-and-vice-versa)
    - [查找呼叫站点：一个稍微复杂的示例](https://llvm.org/docs/ProgrammersManual.html" \l "finding-call-sites-a-slightly-more-complex-example)
    - [遍历 def-use 和 use-def 链](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-def-use-use-def-chains)
    - [迭代块的前驱和后继](https://llvm.org/docs/ProgrammersManual.html" \l "iterating-over-predecessors-successors-of-blocks)
  + [进行简单的更改](https://llvm.org/docs/ProgrammersManual.html" \l "making-simple-changes)
    - [创建和插入新](https://llvm.org/docs/ProgrammersManual.html" \l "creating-and-inserting-new-instructions)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "creating-and-inserting-new-instructions)**[的](https://llvm.org/docs/ProgrammersManual.html" \l "creating-and-inserting-new-instructions)
    - [删除指令](https://llvm.org/docs/ProgrammersManual.html" \l "deleting-instructions)
    - [用另一个值替换指令](https://llvm.org/docs/ProgrammersManual.html" \l "replacing-an-instruction-with-another-value)
      * [替换个别说明](https://llvm.org/docs/ProgrammersManual.html" \l "replacing-individual-instructions)
      * [删除指令](https://llvm.org/docs/ProgrammersManual.html" \l "schanges-deleting-sub)
      * [替换 Users 和 Values 的多次使用](https://llvm.org/docs/ProgrammersManual.html" \l "replacing-multiple-uses-of-users-and-values)
    - [删除全局变量](https://llvm.org/docs/ProgrammersManual.html" \l "deleting-globalvariables)
* [线程和 LLVM](https://llvm.org/docs/ProgrammersManual.html" \l "threads-and-llvm)
  + [结束执行](https://llvm.org/docs/ProgrammersManual.html" \l "ending-execution-with-llvm-shutdown)**[llvm\_shutdown()](https://llvm.org/docs/ProgrammersManual.html" \l "ending-execution-with-llvm-shutdown)**
  + [延迟初始化](https://llvm.org/docs/ProgrammersManual.html" \l "lazy-initialization-with-managedstatic)**[ManagedStatic](https://llvm.org/docs/ProgrammersManual.html" \l "lazy-initialization-with-managedstatic)**
  + [实现隔离](https://llvm.org/docs/ProgrammersManual.html" \l "achieving-isolation-with-llvmcontext)**[LLVMContext](https://llvm.org/docs/ProgrammersManual.html" \l "achieving-isolation-with-llvmcontext)**
  + [线程和 JIT](https://llvm.org/docs/ProgrammersManual.html" \l "threads-and-the-jit)
* [高级主题](https://llvm.org/docs/ProgrammersManual.html" \l "advanced-topics)
  + **[ValueSymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "the-valuesymboltable-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-valuesymboltable-class)
  + **[User](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-and-owned-use-classes-memory-layout)**[和拥有的](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-and-owned-use-classes-memory-layout)**[Use](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-and-owned-use-classes-memory-layout)**[类的内存布局](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-and-owned-use-classes-memory-layout)
    - **[User](https://llvm.org/docs/ProgrammersManual.html" \l "interaction-and-relationship-between-user-and-use-objects)**[和](https://llvm.org/docs/ProgrammersManual.html" \l "interaction-and-relationship-between-user-and-use-objects)**[Use](https://llvm.org/docs/ProgrammersManual.html" \l "interaction-and-relationship-between-user-and-use-objects)**[对象之间的交互和关系](https://llvm.org/docs/ProgrammersManual.html" \l "interaction-and-relationship-between-user-and-use-objects)
  + [设计类型层次结构和多态接口](https://llvm.org/docs/ProgrammersManual.html" \l "designing-type-hierarchies-and-polymorphic-interfaces)
  + [ABI 打破支票](https://llvm.org/docs/ProgrammersManual.html" \l "abi-breaking-checks)
* [核心 LLVM 类层次结构参考](https://llvm.org/docs/ProgrammersManual.html" \l "the-core-llvm-class-hierarchy-reference)
  + [类型类和派生类型](https://llvm.org/docs/ProgrammersManual.html" \l "the-type-class-and-derived-types)
    - [重要的公共方法](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-methods)
    - [重要派生类型](https://llvm.org/docs/ProgrammersManual.html" \l "important-derived-types)
  + **[Module](https://llvm.org/docs/ProgrammersManual.html" \l "the-module-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-module-class)
    - **[Module](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-module-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-module-class)
  + **[Value](https://llvm.org/docs/ProgrammersManual.html" \l "the-value-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-value-class)
    - **[Value](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-value-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-value-class)
  + **[User](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-user-class)
    - **[User](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-user-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-user-class)
  + **[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "the-instruction-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-instruction-class)
    - [该类的重要子](https://llvm.org/docs/ProgrammersManual.html" \l "important-subclasses-of-the-instruction-class)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "important-subclasses-of-the-instruction-class)**[类](https://llvm.org/docs/ProgrammersManual.html" \l "important-subclasses-of-the-instruction-class)
    - **[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-instruction-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-instruction-class)
  + [类](https://llvm.org/docs/ProgrammersManual.html" \l "the-constant-class-and-subclasses)**[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "the-constant-class-and-subclasses)**[和子类](https://llvm.org/docs/ProgrammersManual.html" \l "the-constant-class-and-subclasses)
    - [常量的重要子类](https://llvm.org/docs/ProgrammersManual.html" \l "important-subclasses-of-constant)
  + **[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "the-globalvalue-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-globalvalue-class)
    - **[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-globalvalue-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-globalvalue-class)
  + **[Function](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-function-class)
    - [重要公众成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-function)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-function)**
  + **[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "the-globalvariable-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-globalvariable-class)
    - **[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-globalvariable-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-globalvariable-class)
  + **[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "the-basicblock-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-basicblock-class)
    - **[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-basicblock-class)**[班级的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "important-public-members-of-the-basicblock-class)
  + **[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "the-argument-class)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "the-argument-class)

**警告**

这始终是一项正在进行的工作。

## [简介](https://llvm.org/docs/ProgrammersManual.html" \l "id8)

本文档旨在强调 LLVM 源代码库中可用的一些重要类和接口。本手册无意解释什么是 LLVM、它是如何工作的以及 LLVM 代码是什么样子的。它假定您了解 LLVM 的基础知识并且对编写转换或分析或操作代码感兴趣。

本文档应该让您有方向感，以便您可以在构成 LLVM 基础设施的不断增长的源代码中找到自己的方法。请注意，本手册不能替代阅读源代码，因此如果您认为这些类中的一个应该有一个方法来做某事，但没有列出，请检查源代码。[提供了指向doxygen](https://llvm.org/doxygen/)源的链接，以使其尽可能简单。

本文档的第一部分描述了在 LLVM 基础结构中工作时需要了解的一般信息，第二部分描述了核心 LLVM 类。将来，本手册将通过描述如何使用扩展库的信息进行扩展，例如支配者信息、CFG 遍历例程和有用的实用程序，如InstVisitor( [doxygen](https://llvm.org/doxygen/InstVisitor_8h_source.html) ) 模板。

## [一般信息](https://llvm.org/docs/ProgrammersManual.html" \l "id9)

本节包含一般信息，如果您在 LLVM 源代码库中工作，这些信息很有用，但并不特定于任何特定的 API。

### [C++ 标准模板库](https://llvm.org/docs/ProgrammersManual.html" \l "id10)

LLVM 大量使用 C++ 标准模板库 (STL)，可能比您习惯或以前看到的要多得多。正因为如此，您可能需要阅读一些背景知识，了解所使用的技术和库的功能。有很多讨论 STL 的好页面，您可以获得几本关于该主题的书籍，因此本文档不会讨论它。

以下是一些有用的链接：

1. [cppreference.com](https://en.cppreference.com/w/) - STL 和标准 C++ 库的其他部分的极好参考。
2. [cplusplus.com](https://cplusplus.com/reference/) - 另一个与上面类似的优秀参考资料。
3. [C++ In a Nutshell](http://www.tempest-sw.com/cpp/) - 这是一本 O'Reilly 正在制作的书。它有一个不错的标准库参考，可以与 Dinkumware 的相媲美，不幸的是，自该书出版以来，它不再免费。
4. [C++ 常见问题解答](https://www.parashift.com/c++-faq-lite/)。
5. [Bjarne Stroustrup 的 C++ 页面](https://www.stroustrup.com/C++.html)。
6. [Bruce Eckel 的《C++ 思考》，第 2 版。第 2 卷。（更好的是，拿到这本书）](https://archive.org/details/TICPP2ndEdVolTwo)。

我们还鼓励您查看[LLVM 编码标准](https://llvm.org/docs/CodingStandards.html)指南，该指南侧重于如何编写可维护的代码，而不是将大括号放在哪里。

### [其他有用的参考资料](https://llvm.org/docs/ProgrammersManual.html" \l "id11)

1. [跨平台使用静态库和共享库](http://www.fortran-2000.com/ArnaudRecipes/sharedlib.html)

## [重要且有用的 LLVM API](https://llvm.org/docs/ProgrammersManual.html" \l "id12)

在这里，我们重点介绍一些 LLVM API，这些 API 在编写转换时通常很有用且值得了解。

### [和模板](https://llvm.org/docs/ProgrammersManual.html" \l "id13)**[isa<>cast<>dyn\_cast<>](https://llvm.org/docs/ProgrammersManual.html" \l "id13)**[\_ \_](https://llvm.org/docs/ProgrammersManual.html" \l "id13)

LLVM 源代码库广泛使用了自定义形式的 RTTI。这些模板与 C++ 运算符有很多相似之处dynamic\_cast<>，但它们没有一些缺点（主要是因为 dynamic\_cast<>它只适用于具有 v 表的类）。因为它们经常使用，所以您必须知道它们的作用以及它们的工作原理。所有这些模板都在llvm/Support/Casting.h( [doxygen](https://llvm.org/doxygen/Casting_8h_source.html) ) 文件中定义（请注意，您很少需要直接包含此文件）。

isa<>:

该isa<>运算符的工作方式与 Java 的“ instanceof”运算符完全相同。它根据引用或指针是否指向指定类的实例而返回 true 或 false。这对于各种约束检查非常有用（下面的示例）。

cast<>:

运算符是一个“cast<>检查转换”操作。它将指针或引用从基类转换为派生类，如果它实际上不是正确类型的实例，则会导致断言失败。如果您有一些信息使您相信某些东西属于正确的类型，则应该使用这种方法。isa<>和cast<> 模板的一个例子是：

**static** bool isLoopInvariant(**const** Value \*V, **const** Loop \*L) {

**if** (isa<Constant>(V) || isa<Argument>(V) || isa<GlobalValue>(V))

**return** true;

*// Otherwise, it must be an instruction...*

**return** !L->contains(cast<Instruction>(V)->getParent());}

请注意，您不应该**使用**testisa<>后跟 a cast<>，因为要使用dyn\_cast<>运算符。

dyn\_cast<>:

运算符是一个“dyn\_cast<>检查转换”操作。它检查操作数是否属于指定类型，如果是，则返回指向它的指针（此运算符不适用于引用）。如果操作数的类型不正确，则返回空指针。因此，它的工作方式与 C++ 中的运算符非常相似dynamic\_cast<>，并且应该在相同的情况下使用。通常，dyn\_cast<>运算符用在if 语句或其他一些流程控制语句中，如下所示：

**if** (**auto** \*AI = dyn\_cast<AllocationInst>(Val)) {

*// ...*}

这种形式的if语句有效地将一次调用 isa<>和一次调用组合cast<>成一条语句，非常方便。

请注意，dyn\_cast<>运算符，如 C++dynamic\_cast<>或 Java 的 instanceof运算符，可能会被滥用。特别是，您不应该使用大链if/then/else块来检查类的许多不同变体。如果您发现自己想要这样做，使用InstVisitor类直接分派指令类型会更简洁、更高效。

isa\_and\_nonnull<>:

该isa\_and\_nonnull<>运算符的工作方式与isa<>运算符类似，只是它允许将空指针作为参数（然后返回 false）。这有时很有用，允许您将多个空检查合并为一个。

cast\_or\_null<>:

运算符的cast\_or\_null<>工作方式与cast<>运算符一样，只是它允许将空指针作为参数（然后传播）。这有时很有用，允许您将多个空检查合并为一个。

dyn\_cast\_or\_null<>:

运算符的dyn\_cast\_or\_null<>工作方式与dyn\_cast<> 运算符一样，只是它允许将空指针作为参数（然后传播）。这有时很有用，允许您将多个空检查合并为一个。

这五个模板可用于任何类，无论它们是否具有 v 表。如果您想添加对这些模板的支持，请参阅文档 [如何为您的类层次结构设置 LLVM 样式的 RTTI](https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html)

### [传递字符串（](https://llvm.org/docs/ProgrammersManual.html" \l "id14)**[StringRef](https://llvm.org/docs/ProgrammersManual.html" \l "id14)**[和](https://llvm.org/docs/ProgrammersManual.html" \l "id14)**[Twine](https://llvm.org/docs/ProgrammersManual.html" \l "id14)**[类）](https://llvm.org/docs/ProgrammersManual.html" \l "id14)

虽然 LLVM 通常不会做太多的字符串操作，但我们确实有几个重要的 API 可以接受字符串。两个重要的例子是 Value 类——它有指令、函数等的名称——以及StringMap 在 LLVM 和 Clang 中广泛使用的类。

这些是通用类，它们需要能够接受可能嵌入空字符的字符串。因此，他们不能简单地获取 a ，并且获取 a需要客户端执行通常不需要的堆分配。相反，许多 LLVM API 使用 a 或 a来有效地传递字符串。const char \*const std::string&StringRefconst Twine&

#### **[StringRef](https://llvm.org/docs/ProgrammersManual.html" \l "id15)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id15)

该StringRef数据类型表示对常量字符串（字符数组和长度）的引用，并支持 上可用的常见操作 std::string，但不需要堆分配。

它可以使用 C 风格的以 null 结尾的字符串隐式构造 std::string，或者使用字符指针和长度显式构造。例如，StringMap查找函数声明为：

iterator find(StringRef Key);

客户可以使用以下任何一种方式调用它：

Map.find("foo"); *// Lookup "foo"*Map.find(std::string("bar")); *// Lookup "bar"*Map.find(StringRef("**\0**baz", 4)); *// Lookup "\0baz"*

同样，需要返回字符串的API可能会返回一个StringRef 实例，可以直接使用或std::string使用str成员函数转换为一个实例。有关详细信息，请参阅llvm/ADT/StringRef.h( [doxygen](https://llvm.org/doxygen/StringRef_8h_source.html) )。

您应该很少StringRef直接使用该类，因为它包含指向外部存储器的指针，因此存储该类的实例通常是不安全的（除非您知道外部存储器不会被释放）。 StringRef在 LLVM 中足够小且普遍，因此它应该始终按值传递。

#### **[Twine](https://llvm.org/docs/ProgrammersManual.html" \l "id16)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id16)

( [doxygen](https://llvm.org/doxygen/classllvm_1_1Twine.html)Twine )类是 API 接受串联字符串的有效方式。例如，一种常见的 LLVM 范式是根据另一条指令的名称加上后缀来命名一条指令，例如：

New = CmpInst::Create(..., SO->getName() + ".cmp");

该类Twine实际上是指向临时（堆栈分配）对象的轻量级[绳索。](http://en.wikipedia.org/wiki/Rope_(computer_science))Twine 可以隐式构造为将加号运算符应用于字符串（即，a C 字符串、an std::string或 a StringRef）的结果。twine 会延迟字符串的实际连接，直到实际需要为止，此时它可以有效地直接呈现为字符数组。这避免了在构建字符串连接的临时结果时涉及不必要的堆分配。有关更多信息，请参见llvm/ADT/Twine.h( [doxygen](https://llvm.org/doxygen/Twine_8h_source.html) ) 和[此处](https://llvm.org/docs/ProgrammersManual.html" \l "dss-twine) 。

与 a 一样StringRef，Twine对象指向外部存储器，几乎不应该直接存储或提及。它们仅用于定义一个应该能够有效地接受连接字符串的函数。

### [格式化字符串（](https://llvm.org/docs/ProgrammersManual.html" \l "id17)**[formatv](https://llvm.org/docs/ProgrammersManual.html" \l "id17)**[函数）](https://llvm.org/docs/ProgrammersManual.html" \l "id17)

虽然 LLVM 不一定会进行大量的字符串操作和解析，但它确实会进行大量的字符串格式化。从诊断消息到 llvm 工具输出（例如llvm-readobj打印详细的反汇编列表和 LLDB 运行时日志记录），字符串格式化的需求无处不在。

它formatv在本质上与 相似printf，但使用了不同的语法，该语法大量借鉴了 Python 和 C#。不像printf它在编译时推导要格式化的类型，所以它不需要格式说明符，例如 %d. 这减少了尝试构造可移植格式字符串的心理开销，尤其是对于特定于平台的类型（如size\_t或指针类型）。与printf和 Python 不同的是，如果 LLVM 不知道如何格式化类型，它也会无法编译。这两个属性确保该函数比传统的格式化方法（如printf函数族）更安全、更易于使用。

#### [简单格式](https://llvm.org/docs/ProgrammersManual.html" \l "id18)

对 的调用涉及由 0 个或多个 **替换序列**formatv组成的单一**格式字符串**，后跟可变长度的**替换值**列表。替换序列是 形式的字符串。{N[[,align]:style]}

N从替换值列表中引用参数的从 0 开始的索引。请注意，这意味着可以以任何顺序多次引用相同的参数，可能使用不同的样式和/或对齐选项。

align是一个可选字符串，指定要将值格式化为的字段的宽度，以及值在字段中的对齐方式。它被指定为可选的**对齐样式**，后跟正整数**字段宽度**。对齐方式可以是字符-（左对齐）、=（居中对齐）或+（右对齐）之一。默认是右对齐。

style是一个可选字符串，由控制值格式的特定类型组成。例如，要将浮点值格式化为百分比，您可以使用样式选项P。

#### [自定义格式](https://llvm.org/docs/ProgrammersManual.html" \l "id19)

有两种方法可以自定义类型的格式化行为。

1. 使用适当的静态格式方法llvm::format\_provider<T>为您的类型提供模板特化。T

**namespace** llvm {

**template**<>

**struct** **format\_provider**<MyFooBar> {

**static** void format(**const** MyFooBar &V, raw\_ostream &Stream, StringRef Style) {

*// Do whatever is necessary to format `V` into `Stream`*

}

};

void foo() {

MyFooBar X;

std::string S = formatv("{0}", X);

}}

这是一种有用的扩展机制，用于添加对使用您自己的自定义样式选项格式化您自己的自定义类型的支持。但是当您想扩展格式化库已经知道如何格式化的类型的机制时，它无济于事。为此，我们需要别的东西。

1. 提供**继承**自llvm::FormatAdapter<T>.

**namespace** anything {

**struct** **format\_int\_custom** : **public** llvm::FormatAdapter<int> {

**explicit** format\_int\_custom(int N) : llvm::FormatAdapter<int>(N) {}

void format(llvm::raw\_ostream &Stream, StringRef Style) **override** {

*// Do whatever is necessary to format ``this->Item`` into ``Stream``*

}

};}**namespace** llvm {

void foo() {

std::string S = formatv("{0}", anything::format\_int\_custom(42));

}}

如果检测到该类型是从 派生的FormatAdapter<T>，formatv 将调用 format以指定样式传递的参数上的方法。这允许提供任何类型的自定义格式，包括已经具有内置格式提供程序的格式。

#### **[formatv](https://llvm.org/docs/ProgrammersManual.html" \l "id20)**[例子](https://llvm.org/docs/ProgrammersManual.html" \l "id20)

下面旨在提供一组不完整的示例来演示formatv. 通过阅读 doxygen 文档或查看单元测试套件可以找到更多信息。

std::string S;*// Simple formatting of basic types and implicit string conversion.*S = formatv("{0} ({1:P})", 7, 0.35); *// S == "7 (35.00%)"*

*// Out-of-order referencing and multi-referencing*outs() << formatv("{0} {2} {1} {0}", 1, "test", 3); *// prints "1 3 test 1"*

*// Left, right, and center alignment*S = formatv("{0,7}", 'a'); *// S == " a";*S = formatv("{0,-7}", 'a'); *// S == "a ";*S = formatv("{0,=7}", 'a'); *// S == " a ";*S = formatv("{0,+7}", 'a'); *// S == " a";*

*// Custom styles*S = formatv("{0:N} - {0:x} - {1:E}", 12345, 123908342); *// S == "12,345 - 0x3039 - 1.24E8"*

*// Adapters*S = formatv("{0}", fmt\_align(42, AlignStyle::Center, 7)); *// S == " 42 "*S = formatv("{0}", fmt\_repeat("hi", 3)); *// S == "hihihi"*S = formatv("{0}", fmt\_pad("hi", 2, 6)); *// S == " hi "*

*// Ranges*std::vector<int> V = {8, 9, 10};S = formatv("{0}", make\_range(V.begin(), V.end())); *// S == "8, 9, 10"*S = formatv("{0:$[+]}", make\_range(V.begin(), V.end())); *// S == "8+9+10"*S = formatv("{0:$[ + ]@[x]}", make\_range(V.begin(), V.end())); *// S == "0x8 + 0x9 + 0xA"*

### [错误处理](https://llvm.org/docs/ProgrammersManual.html" \l "id21)

正确的错误处理有助于我们识别代码中的错误，并帮助最终用户了解其工具使用中的错误。错误分为两大类： 编程错误和可恢复错误，具有不同的处理和报告策略。

#### [编程错误](https://llvm.org/docs/ProgrammersManual.html" \l "id22)

编程错误是对程序不变量或 API 契约的违反，代表程序本身的错误。我们的目标是记录不变量，并在不变量在运行时被破坏时在故障点快速中止（提供一些基本诊断）。

处理编程错误的基本工具是断言和 llvm\_unreachable 函数。断言用于表达不变条件，并且应该包括描述不变条件的消息：

assert(isPhysReg(R) && "All virt regs should have been allocated already.");

如果程序不变量成立，llvm\_unreachable 函数可用于记录不应进入的控制流区域：

**enum** { Foo, Bar, Baz } X = foo();

**switch** (X) {

**case** **Foo**: */\* Handle Foo \*/*; **break**;

**case** **Bar**: */\* Handle Bar \*/*; **break**;

**default**:

llvm\_unreachable("X should be Foo or Bar here");}

#### [可恢复的错误](https://llvm.org/docs/ProgrammersManual.html" \l "id23)

可恢复错误表示程序环境中的错误，例如资源故障（文件丢失、网络连接断开等）或格式错误的输入。这些错误应该被检测到并传达给可以适当处理它们的程序级别。处理错误可能与向用户报告问题一样简单，也可能涉及尝试恢复。

**笔记**

虽然在整个 LLVM 中使用这种错误处理方案是理想的，但在某些地方应用它并不实际。在您绝对必须发出非编程错误并且Error模型不可用的情况下，您可以调用report\_fatal\_error，这将调用已安装的错误处理程序、打印消息并中止程序。不鼓励在这种情况下使用report\_fatal\_error 。

可恢复的错误是使用 LLVM 的Error方案建模的。此方案使用函数返回值表示错误，类似于经典的 C 整数错误代码或 C++ 的std::error\_code. 但是，Error该类实际上是用户定义的错误类型的轻量级包装器，允许附加任意信息来描述错误。这类似于 C++ 异常允许抛出用户定义类型的方式。

成功值是通过调用创建的Error::success()，例如：

Error foo() {

*// Do something.*

*// Return success.*

**return** Error::success();}

成功值的构造和返回成本非常低——它们对程序性能的影响极小。

失败值是使用构造的make\_error<T>，其中T是从 ErrorInfo 实用程序继承的任何类，例如：

**class** **BadFileFormat** : **public** ErrorInfo<BadFileFormat> {**public**:

**static** char ID;

std::string Path;

BadFileFormat(StringRef Path) : Path(Path.str()) {}

void log(raw\_ostream &OS) **const** **override** {

OS << Path << " is malformed";

}

std::error\_code convertToErrorCode() **const** **override** {

**return** make\_error\_code(object\_error::parse\_failed);

}};

char BadFileFormat::ID; *// This should be declared in the C++ file.*

Error printFormattedFile(StringRef Path) {

**if** (<check **for** valid format>)

**return** make\_error<BadFileFormat>(Path);

*// print file contents.*

**return** Error::success();}

错误值可以隐式转换为 bool：true 表示错误，false 表示成功，启用以下习惯用法：

Error mayFail();

Error foo() {

**if** (**auto** Err = mayFail())

**return** Err;

*// Success! We can proceed.*

...

对于可能会失败但需要返回值的函数，可以Expected<T> 使用实用程序。这种类型的值可以用 a T或 an构造Error。Expected<T> 值也可以隐式转换为布尔值，但约定相反Error：true 表示成功，false 表示错误。如果成功，则T可以通过取消引用运算符访问该值。如果失败，Error可以使用该 takeError()方法提取值。惯用用法看起来像：

Expected<FormattedFile> openFormattedFile(StringRef Path) {

*// If badly formatted, return an error.*

**if** (**auto** Err = checkFormat(Path))

**return** std::move(Err);

*// Otherwise return a FormattedFile instance.*

**return** FormattedFile(Path);}

Error processFormattedFile(StringRef Path) {

*// Try to open a formatted file*

**if** (**auto** FileOrErr = openFormattedFile(Path)) {

*// On success, grab a reference to the file and continue.*

**auto** &File = \*FileOrErr;

...

} **else**

*// On error, extract the Error value and return it.*

**return** FileOrErr.takeError();}

如果Expected<T>值处于成功模式，则该takeError()方法将返回成功值。使用这个事实，上面的函数可以重写为：

Error processFormattedFile(StringRef Path) {

*// Try to open a formatted file*

**auto** FileOrErr = openFormattedFile(Path);

**if** (**auto** Err = FileOrErr.takeError())

*// On error, extract the Error value and return it.*

**return** Err;

*// On success, grab a reference to the file and continue.*

**auto** &File = \*FileOrErr;

...}

对于涉及多个值的函数，第二种形式通常更具可读性， Expected<T>因为它限制了所需的缩进。

如果一个Expected<T>值将被移动到现有变量中，则该 moveInto()方法避免了命名额外变量的需要。这对于使operator->()值Expected<T>具有类似指针的语义很有用。例如：

Expected<std::unique\_ptr<MemoryBuffer>> openBuffer(StringRef Path);Error processBuffer(StringRef Buffer);

Error processBufferAtPath(StringRef Path) {

*// Try to open a buffer.*

std::unique\_ptr<MemoryBuffer> MB;

**if** (**auto** Err = openBuffer(Path).moveInto(MB))

*// On error, return the Error value.*

**return** Err;

*// On success, use MB.*

**return** processBuffer(MB->getBuffer());}

第三种形式适用于任何可以分配给 from 的类型T&&。如果Expected<T>需要将值存储为已声明的 Optional<T>. 例如：

Expected<StringRef> extractClassName(StringRef Definition);**struct** **ClassData** {

StringRef Definition;

Optional<StringRef> LazyName;

...

Error initialize() {

**if** (**auto** Err = extractClassName(Path).moveInto(LazyName))

*// On error, return the Error value.*

**return** Err;

*// On success, LazyName has been initialized.*

...

}};

所有Error实例，无论是成功还是失败，都必须在std::move销毁之前进行检查或移动（通过或返回）。意外丢弃未经检查的错误将导致程序在未经检查值的析构函数运行时中止，从而很容易识别和修复违反此规则的情况。

成功值一旦被测试（通过调用布尔转换运算符）就被视为已检查：

**if** (**auto** Err = mayFail(...))

**return** Err; *// Failure value - move error to caller.*

*// Safe to continue: Err was checked.*

相反，以下代码将始终导致中止，即使mayFail 返回成功值也是如此：

mayFail();*// Program will always abort here, even if mayFail() returns Success, since// the value is not checked.*

一旦错误类型的处理程序被激活，就认为失败值已被检查：

handleErrors(

processFormattedFile(...),

[](**const** BadFileFormat &BFF) {

report("Unable to process " + BFF.Path + ": bad format");

},

[](**const** FileNotFound &FNF) {

report("File not found " + FNF.Path);

});

该handleErrors函数将错误作为其第一个参数，后跟一个可变的“处理程序”列表，每个“处理程序”都必须是具有一个参数的可调用类型（函数、lambda 或带有调用运算符的类）。该 handleErrors函数将访问序列中的每个处理程序，并根据错误的动态类型检查其参数类型，运行第一个匹配的处理程序。这与用于决定为 C++ 异常运行哪个 catch 子句的决策过程相同。

由于传递给的处理程序列表handleErrors可能无法涵盖所有​​可能发生的错误类型，因此该handleErrors函数还会返回一个必须检查或传播的错误值。如果传递给的错误值 handleErrors与任何处理程序都不匹配，它将从 handleErrors 返回。thus的惯用用法handleErrors看起来像：

**if** (**auto** Err =

handleErrors(

processFormattedFile(...),

[](**const** BadFileFormat &BFF) {

report("Unable to process " + BFF.Path + ": bad format");

},

[](**const** FileNotFound &FNF) {

report("File not found " + FNF.Path);

}))

**return** Err;

如果您确实知道处理程序列表是详尽无遗的， handleAllErrors则可以改用该函数。这与 相同， handleErrors只是如果传入未处理的错误，它将终止程序，因此可以返回 void。通常应避免该handleAllErrors 功能：在程序的其他地方引入新的错误类型很容易将以前详尽的错误列表变成非详尽的列表，从而有意外终止程序的风险。在可能的情况下，使用 handleErrors 并将未知错误传播到堆栈中。

对于工具代码，可以通过打印错误消息然后使用错误代码退出来处理错误，[ExitOnError](https://llvm.org/docs/ProgrammersManual.html" \l "err-exitonerr)实用程序可能是比 handleErrors 更好的选择，因为它简化了调用易出错函数时的控制流。

在已知对易出错函数的特定调用总是会成功的情况下（例如，调用函数只能在已知输入是安全的输入的子集上失败）， [cantFail](https://llvm.org/docs/ProgrammersManual.html" \l "err-cantfail)函数可以是用于删除错误类型，简化控制流程。

##### [字符串错误](https://llvm.org/docs/ProgrammersManual.html" \l "id24)

许多类型的错误没有恢复策略，唯一可以采取的措施是将它们报告给用户，以便用户可以尝试修复环境。在这种情况下，将错误表示为字符串非常有意义。LLVMStringError为此提供了类。它有两个参数：一个字符串错误消息和一个等效std::error\_code的互操作性消息。它还提供了一个createStringError功能来简化此类的常见用法：

*// These two lines of code are equivalent:*make\_error<StringError>("Bad executable", errc::executable\_format\_error);createStringError(errc::executable\_format\_error, "Bad executable");

如果您确定您正在构建的错误永远不需要转换为std::error\_code您可以使用该inconvertibleErrorCode()函数：

createStringError(inconvertibleErrorCode(), "Bad executable");

只有在仔细考虑后才能这样做。如果尝试将此错误转换为错误，std::error\_code它将立即触发程序终止。除非您确定您的错误不需要互操作性，否则您应该寻找std::error\_code可以转换为的现有错误，甚至（尽管很痛苦）考虑引入一个新错误作为权宜之计。

createStringError可以采用printf样式格式说明符来提供格式化消息：

createStringError(errc::executable\_format\_error,

"Bad executable: %s", FileName);

##### [与 std::error\_code 和 ErrorOr 的互操作性](https://llvm.org/docs/ProgrammersManual.html" \l "id25)

许多现有的 LLVM API 使用std::error\_code和它的伙伴ErrorOr<T> （扮演与 相同的角色Expected<T>，但包装了一个std::error\_code 而不是一个Error）。错误类型的传染性意味着尝试将其中一个函数更改为返回Error或Expected<T> 取而代之通常会导致对调用者、调用者的调用者等的大量更改。（第一次这样的尝试，Error从 MachOObjectFile 的构造函数返回一个，在 diff 达到 3000 行后被放弃，影响了六个库，并且仍在增长）。

为了解决这个问题，引入了Error/std::error\_code互操作性要求。两对函数允许将任何Error值转换为 a std::error\_code，将任何值Expected<T>转换为 an ErrorOr<T>，反之亦然：

std::error\_code errorToErrorCode(Error Err);Error errorCodeToError(std::error\_code EC);

**template** <**typename** **T**> ErrorOr<T> expectedToErrorOr(Expected<T> TOrErr);**template** <**typename** **T**> Expected<T> errorOrToExpected(ErrorOr<T> TOrEC);

使用这些 API 可以很容易地制作手术补丁，将各个功能从 更新std::error\_code到Error，以及从更新ErrorOr<T>到 Expected<T>。

##### [从错误处理程序返回错误](https://llvm.org/docs/ProgrammersManual.html" \l "id26)

错误恢复尝试本身可能会失败。出于这个原因，handleErrors 实际上识别三种不同形式的处理程序签名：

*// Error must be handled, no new errors produced:*void(UserDefinedError &E);

*// Error must be handled, new errors can be produced:*Error(UserDefinedError &E);

*// Original error can be inspected, then re-wrapped and returned (or a new// error can be produced):*Error(std::unique\_ptr<UserDefinedError> E);

从处理程序返回的任何错误都将从函数返回，handleErrors 以便它可以自己处理，或向上传播堆栈。

##### [使用 ExitOnError 简化工具代码](https://llvm.org/docs/ProgrammersManual.html" \l "id27)

库代码永远不应该调用exit可恢复的错误，但是在工具代码（尤其是命令行工具）中，这可能是一种合理的方法。遇到错误时调用 exit可显着简化控制流，因为错误不再需要向上传播堆栈。这允许以直线方式编写代码，只要每个容易出错的调用都包含在检查和退出调用中。该类ExitOnError通过提供检查Error值的调用运算符支持此模式，在成功案例中剥离错误并stderr在失败案例中记录然后退出。

要使用此类，ExitOnError请在程序中声明一个全局变量：

ExitOnError ExitOnErr;

然后可以用对 的调用包装对易出错函数的调用ExitOnErr，将它们变成非失败调用：

Error mayFail();Expected<int> mayFail2();

void foo() {

ExitOnErr(mayFail());

int X = ExitOnErr(mayFail2());}

失败时，错误的日志消息将被写入stderr，可以选择在前面加上一个字符串“banner”，可以通过调用 setBanner 方法进行设置。也可以Error使用以下方法提供从值到退出代码 的映射setExitCodeMapper：

int main(int argc, char \*argv[]) {

ExitOnErr.setBanner(std::string(argv[0]) + " error:");

ExitOnErr.setExitCodeMapper(

[](**const** Error &Err) {

**if** (Err.isA<BadFileFormat>())

**return** 2;

**return** 1;

});

尽可能在您的工具代码中使用ExitOnError，因为它可以大大提高可读性。

##### [使用 cantFail 简化安全调用站点](https://llvm.org/docs/ProgrammersManual.html" \l "id28)

某些函数可能仅对其输入的一个子集失败，因此可以假定使用已知安全输入的调用会成功。

cantFail 函数通过封装其参数是成功值的断言来对此进行封装，并且在 Expected<T> 的情况下，解封装 T 值：

Error onlyFailsForSomeXValues(int X);Expected<int> onlyFailsForSomeXValues2(int X);

void foo() {

cantFail(onlyFailsForSomeXValues(KnownSafeValue));

int Y = cantFail(onlyFailsForSomeXValues2(KnownSafeValue));

...}

与 ExitOnError 实用程序一样，cantFail 简化了控制流。然而，它们对错误情况的处理非常不同：ExitOnError 保证在错误输入时终止程序，cantFail 只是断言结果是成功的。在调试版本中，如果遇到错误，这将导致断言失败。在发布版本中，未定义失败值的 cantFail 行为。因此，在使用 cantFail 时必须小心：客户必须确定 cantFail 包装调用确实不会因给定参数而失败。

cantFail 函数在库代码中的使用应该很少见，但它们可能更多地用于工具和单元测试代码，在这些代码中输入和/或模拟类或函数可能已知是安全的。

##### [容易出错的构造函数](https://llvm.org/docs/ProgrammersManual.html" \l "id29)

某些类需要资源获取或其他复杂的初始化，这些初始化可能会在构建过程中失败。不幸的是，构造函数不能返回错误，并且让客户在构造对象之后测试对象以确保它们有效是容易出错的，因为很容易忘记测试。要解决此问题，请使用命名构造函数惯用语并返回一个Expected<T>：

**class** **Foo** {**public**:

**static** Expected<Foo> Create(Resource R1, Resource R2) {

Error Err = Error::success();

Foo F(R1, R2, Err);

**if** (Err)

**return** std::move(Err);

**return** std::move(F);

}

**private**:

Foo(Resource R1, Resource R2, Error &Err) {

ErrorAsOutParameter EAO(&Err);

**if** (**auto** Err2 = R1.acquire()) {

Err = std::move(Err2);

**return**;

}

Err = R2.acquire();

}};

在这里，命名构造函数通过Error引用传递给实际构造函数，然后构造函数可以使用它来返回错误。该 ErrorAsOutParameter实用程序Error在构造函数的入口处设置值的检查标志，以便可以将错误分配给它，然后在退出时重置它以强制客户端（指定的构造函数）检查错误。

通过使用这个惯用语，试图构造 Foo 的客户会收到一个格式正确的 Foo 或一个错误，而不会收到一个处于无效状态的对象。

##### [基于类型传播和消费错误](https://llvm.org/docs/ProgrammersManual.html" \l "id30)

在某些情况下，已知某些类型的错误是良性的。例如，当遍历一个档案时，一些客户可能乐于跳过格式错误的目标文件，而不是立即终止遍历。跳过格式错误的对象可以使用精心设计的处理程序方法来实现，但 Error.h 头文件提供了两个实用程序，使这个习惯用法更清晰：类型检查方法，isA和consumeError函数：

Error walkArchive(Archive A) {

**for** (unsigned I = 0; I != A.numMembers(); ++I) {

**auto** ChildOrErr = A.getMember(I);

**if** (**auto** Err = ChildOrErr.takeError()) {

**if** (Err.isA<BadFileFormat>())

consumeError(std::move(Err))

**else**

**return** Err;

}

**auto** &Child = \*ChildOrErr;

*// Use Child*

...

}

**return** Error::success();}

##### [用 joinErrors 连接错误](https://llvm.org/docs/ProgrammersManual.html" \l "id31)

在上面的存档行走示例中，BadFileFormat错误被简单地消耗和忽略。如果客户希望在完成遍历存档后报告这些错误，他们可以使用该joinErrors实用程序：

Error walkArchive(Archive A) {

Error DeferredErrs = Error::success();

**for** (unsigned I = 0; I != A.numMembers(); ++I) {

**auto** ChildOrErr = A.getMember(I);

**if** (**auto** Err = ChildOrErr.takeError())

**if** (Err.isA<BadFileFormat>())

DeferredErrs = joinErrors(std::move(DeferredErrs), std::move(Err));

**else**

**return** Err;

**auto** &Child = \*ChildOrErr;

*// Use Child*

...

}

**return** DeferredErrs;}

该joinErrors例程构建了一个名为 的特殊错误类型ErrorList，其中包含用户定义的错误列表。例程识别此handleErrors类型并将尝试按顺序处理每个包含的错误。如果可以处理所有包含的错误，handleErrors将返回 Error::success()，否则handleErrors将连接剩余的错误并返回结果ErrorList。

##### [构建易错迭代器和迭代器范围](https://llvm.org/docs/ProgrammersManual.html" \l "id32)

上面的存档遍历示例通过索引检索存档成员，但是这需要相当多的样板来进行迭代和错误检查。我们可以通过使用“易错迭代器”模式来清理它，它支持以下易错容器（如 Archive）的自然迭代习惯用法：

Error Err = Error::success();**for** (**auto** &**Child** : Ar->children(Err)) {

*// Use Child - only enter the loop when it's valid*

*// Allow early exit from the loop body, since we know that Err is success*

*// when we're inside the loop.*

**if** (BailOutOn(Child))

**return**;

...}*// Check Err after the loop to ensure it didn't break due to an error.***if** (Err)

**return** Err;

为了实现这个习惯用法，易错容器上的迭代器以自然风格编写，它们的++and--运算符被易错的 and函数代替。例如：Error inc()Error dec()

**class** **FallibleChildIterator** {**public**:

FallibleChildIterator(Archive &A, unsigned ChildIdx);

Archive::Child &**operator**\*();

**friend** bool **operator**==(**const** ArchiveIterator &LHS,

**const** ArchiveIterator &RHS);

*// operator++/operator-- replaced with fallible increment / decrement:*

Error inc() {

**if** (!A.childValid(ChildIdx + 1))

**return** make\_error<BadArchiveMember>(...);

++ChildIdx;

**return** Error::success();

}

Error dec() { ... }};

这种易出错迭代器接口的实例然后被 fallible\_iterator 实用程序包装，该实用程序提供operator++和operator--，通过在构造时传递给包装器的引用返回任何错误。fallible\_iterator 包装器负责 (a) 跳转到错误范围的末尾，以及 (b) 每当将迭代器与迭代器进行比较end并发现不相等时，将错误标记为已检查（特别是：这将错误标记为已检查基于范围的 for 循环的主体），可以在没有冗余错误检查的情况下提前退出循环。

易错迭代器接口的实例（例如上面的 FallibleChildIterator）使用make\_fallible\_itr和make\_fallible\_end 函数包装。例如：

**class** **Archive** {**public**:

**using** child\_iterator = fallible\_iterator<FallibleChildIterator>;

child\_iterator child\_begin(Error &Err) {

**return** make\_fallible\_itr(FallibleChildIterator(\***this**, 0), Err);

}

child\_iterator child\_end() {

**return** make\_fallible\_end(FallibleChildIterator(\***this**, size()));

}

iterator\_range<child\_iterator> children(Error &Err) {

**return** make\_range(child\_begin(Err), child\_end());

}};

使用 fallible\_iterator 实用程序允许自然构造易出错的迭代器（使用失败inc和dec操作）和相对自然地使用 c++ 迭代器/循环习惯用法。

有关 Error 及其相关实用程序的更多信息，请参见 Error.h 头文件。

### [传递函数和其他可调用对象](https://llvm.org/docs/ProgrammersManual.html" \l "id33)

有时您可能希望将回调对象传递给函数。为了支持 lambda 表达式和其他函数对象，您不应使用采用函数指针和不透明 cookie 的传统 C 方法：

void takeCallback(bool (\*Callback)(Function \*, void \*), void \*Cookie);

相反，请使用以下方法之一：

#### [函数模板](https://llvm.org/docs/ProgrammersManual.html" \l "id34)

如果您不介意将函数的定义放入头文件中，请将其设为基于可调用类型模板化的函数模板。

**template**<**typename** **Callable**>void takeCallback(Callable Callback) {

Callback(1, 2, 3);}

#### [类](https://llvm.org/docs/ProgrammersManual.html" \l "id35)**[function\_ref](https://llvm.org/docs/ProgrammersManual.html" \l "id35)**[模板](https://llvm.org/docs/ProgrammersManual.html" \l "id35)

( [doxygen](https://llvm.org/doxygen/classllvm_1_1function__ref_3_01Ret_07Params_8_8_8_08_4.html)function\_ref ) 类模板表示对可调用对象的引用 ，模板化为可调用对象的类型。如果您不需要在函数返回后保留回调，那么这是将回调传递给函数的不错选择。这样， 就是这样。function\_refstd::functionStringRefstd::string

function\_ref<Ret(Param1, Param2, ...)>Param1可以从任何可以使用, , …类型的参数调用的可调用对象隐式构造Param2，并返回一个可以转换为 type 的值Ret。例如：

void visitBasicBlocks(Function \*F, function\_ref<bool (BasicBlock\*)> Callback) {

**for** (BasicBlock &**BB** : \*F)

**if** (Callback(&BB))

**return**;}

可以使用以下方式调用：

visitBasicBlocks(F, [&](BasicBlock \*BB) {

**if** (process(BB))

**return** isEmpty(BB);

**return** false;});

请注意，function\_ref对象包含指向外部存储器的指针，因此存储类的实例通常是不安全的（除非您知道外部存储器不会被释放）。如果您需要此功能，请考虑使用std::function. function\_ref足够小，它应该始终按值传递。

### **[LLVM\_DEBUG()](https://llvm.org/docs/ProgrammersManual.html" \l "id36)**[宏和](https://llvm.org/docs/ProgrammersManual.html" \l "id36)**[-debug](https://llvm.org/docs/ProgrammersManual.html" \l "id36)**[选项](https://llvm.org/docs/ProgrammersManual.html" \l "id36)

通常在处理你的 pass 时，你会把一堆调试打印输出和其他代码放入你的 pass 中。在你让它工作之后，你想删除它，但你可能在未来再次需要它（解决你遇到的新错误）。

自然地，正因为如此，你不想删除调试打印输出，但你也不希望它们总是嘈杂。一个标准的妥协是将它们注释掉，允许您在将来需要时启用它们。

( [doxygen](https://llvm.org/doxygen/Debug_8h_source.html) ) 文件提供了一个名为的宏 llvm/Support/Debug.h，它是解决这个问题的更好的方法。基本上，您可以将任意代码放入宏的参数中，并且仅当“ ”（或任何其他工具）使用“ ”命令行参数运行时才会执行：LLVM\_DEBUG()LLVM\_DEBUGopt-debug

LLVM\_DEBUG(dbgs() << "I am here!**\n**");

然后你可以像这样运行你的通行证：

$ opt < a.bc > /dev/null -mypass

<no output>

$ opt < a.bc > /dev/null -mypass -debug

I am here!

使用LLVM\_DEBUG()宏而不是自制的解决方案可以让您不必为您的传递的调试输出创建“又一个”命令行选项。请注意，LLVM\_DEBUG()对于非断言构建，宏是禁用的，因此它们根本不会造成性能影响（出于同样的原因，它们也不应该包含副作用！）。

关于宏的另一个好处LLVM\_DEBUG()是您可以直接在 gdb 中启用或禁用它。如果程序正在运行，只需使用 gdb 中的“ ”或“ ”。如果该程序尚未启动，您始终可以使用 运行它。set DebugFlag=0set DebugFlag=1-debug

#### [带有选项](https://llvm.org/docs/ProgrammersManual.html" \l "id37)**[DEBUG\_TYPE](https://llvm.org/docs/ProgrammersManual.html" \l "id37)**[的细粒度调试信息](https://llvm.org/docs/ProgrammersManual.html" \l "id37)**[-debug-only](https://llvm.org/docs/ProgrammersManual.html" \l "id37)**

有时您可能会发现自己处于启用-debug只会打开**太多**信息的情况（例如在代码生成器上工作时）。如果你想通过更细粒度的控制启用调试信息，你应该定义DEBUG\_TYPE宏并使用-debug-only如下选项：

#define DEBUG\_TYPE "foo"LLVM\_DEBUG(dbgs() << "'foo' debug type**\n**");#undef DEBUG\_TYPE#define DEBUG\_TYPE "bar"LLVM\_DEBUG(dbgs() << "'bar' debug type**\n**");#undef DEBUG\_TYPE

然后你可以像这样运行你的通行证：

$ opt < a.bc > /dev/null -mypass

<no output>

$ opt < a.bc > /dev/null -mypass -debug

'foo' debug type

'bar' debug type

$ opt < a.bc > /dev/null -mypass -debug-only=foo

'foo' debug type

$ opt < a.bc > /dev/null -mypass -debug-only=bar

'bar' debug type

$ opt < a.bc > /dev/null -mypass -debug-only=foo,bar

'foo' debug type

'bar' debug type

当然，在实践中，你应该只DEBUG\_TYPE在一个文件的顶部设置，来指定整个模块的调试类型。请注意，您只能在包含 Debug.h 之后执行此操作，而不是围绕任何 #include 标头。此外，您应该使用比“foo”和“bar”更有意义的名称，因为没有适当的系统来确保名称不冲突。如果两个不同的模块使用相同的字符串，则在指定名称时它们将全部打开。例如，这允许使用启用指令调度的所有调试信息-debug-only=InstrSched，即使源存在于多个文件中。该名称不得包含逗号 (,)，因为它用于分隔-debug-only选项的参数。

--enable-optimized出于性能原因，-debug-only 在LLVM的优化构建 () 中不可用。

该DEBUG\_WITH\_TYPE宏也可用于您想要设置的情况DEBUG\_TYPE，但仅限于一个特定的DEBUG语句。它需要一个额外的第一个参数，即要使用的类型。例如，前面的例子可以写成：

DEBUG\_WITH\_TYPE("foo", dbgs() << "'foo' debug type**\n**");DEBUG\_WITH\_TYPE("bar", dbgs() << "'bar' debug type**\n**");

### [类](https://llvm.org/docs/ProgrammersManual.html" \l "id38)**[Statistic](https://llvm.org/docs/ProgrammersManual.html" \l "id38)**[和](https://llvm.org/docs/ProgrammersManual.html" \l "id38)**[-stats](https://llvm.org/docs/ProgrammersManual.html" \l "id38)**[选项](https://llvm.org/docs/ProgrammersManual.html" \l "id38)

( [doxygen](https://llvm.org/doxygen/Statistic_8h_source.html) ) 文件提供了一个名为的类llvm/ADT/Statistic.h，用作一种统一的方式来跟踪 LLVM 编译器正在做什么以及各种优化的有效性。查看哪些优化有助于使特定程序运行得更快是很有用的。Statistic

通常你可能会在一些大程序上运行你的 pass，并且你有兴趣看看它进行了多少次特定的转换。虽然您可以通过手动检查或一些特别的方法来做到这一点，但这确实很痛苦，而且对大型程序来说不是很有用。使用Statistic该类可以很容易地跟踪此信息，并且计算出的信息以统一的方式呈现，其余的传递正在执行。

使用示例有很多Statistic，但基本的使用方法如下：

像这样定义您的统计信息：

#define DEBUG\_TYPE "mypassname" *// This goes after any #includes.*STATISTIC(NumXForms, "The # of times I did stuff");

该STATISTIC宏定义了一个静态变量，其名称由第一个参数指定。传递名称取自DEBUG\_TYPE宏，描述取自第二个参数。定义的变量（在本例中为“NumXForms”）就像一个无符号整数。

每当您进行转换时，请点击计数器：

++NumXForms; *// I did stuff!*

这就是你所要做的。要让“ opt”打印出收集到的统计信息，请使用“ -stats”选项：

$ opt -stats -mypassname < program.bc > /dev/null

... statistics output ...

请注意，为了使用 ' -stats' 选项，必须在启用断言的情况下编译 LLVM。

在 SPEC 基准测试套件中的 C 文件上运行opt时，它会给出如下所示的报告：

7646 bitcodewriter - Number of normal instructions

725 bitcodewriter - Number of oversized instructions

129996 bitcodewriter - Number of bitcode bytes written

2817 raise - Number of insts DCEd or constprop'd

3213 raise - Number of cast-of-self removed

5046 raise - Number of expression trees converted

75 raise - Number of other getelementptr's formed

138 raise - Number of load/store peepholes

42 deadtypeelim - Number of unused typenames removed from symtab

392 funcresolve - Number of varargs functions resolved

27 globaldce - Number of global variables removed

2 adce - Number of basic blocks removed

134 cee - Number of branches revectored

49 cee - Number of setcc instruction eliminated

532 gcse - Number of loads removed

2919 gcse - Number of instructions removed

86 indvars - Number of canonical indvars added

87 indvars - Number of aux indvars removed

25 instcombine - Number of dead inst eliminate

434 instcombine - Number of insts combined

248 licm - Number of load insts hoisted

1298 licm - Number of insts hoisted to a loop pre-header

3 licm - Number of insts hoisted to multiple loop preds (bad, no loop pre-header)

75 mem2reg - Number of alloca's promoted

1444 cfgsimplify - Number of blocks simplified

很显然，这么多的优化，这个东西有一个统一的框架是非常好的。使您的通行证很好地融入框架可以使其更易于维护和使用。

### [添加调试计数器以帮助调试代码](https://llvm.org/docs/ProgrammersManual.html" \l "id39)

有时，在编写新通行证或尝试追踪错误时，能够控制通行证中的某些事情是否发生是很有用的。例如，有时最小化工具只能轻松地为您提供大型测试用例。您希望使用二分法自动将错误缩小到发生或未发生的特定转换。这是调试计数器的帮助所在。它们提供了一个框架，使您的部分代码仅执行一定次数。

( [doxygen](https://llvm.org/doxygen/DebugCounter_8h_source.html) ) 文件提供了一个名为的类llvm/Support/DebugCounter.h，可用于创建控制部分代码执行的命令行计数器选项。DebugCounter

像这样定义您的 DebugCounter：

DEBUG\_COUNTER(DeleteAnInstruction, "passname-delete-instruction",

"Controls which instructions get delete");

该DEBUG\_COUNTER宏定义了一个静态变量，其名称由第一个参数指定。计数器的名称（在命令行中使用）由第二个参数指定，帮助中使用的描述由第三个参数指定。

无论您想要该控件的代码是什么，都可以使用DebugCounter::shouldExecute它来控制它。

**if** (DebugCounter::shouldExecute(DeleteAnInstruction))

I->eraseFromParent();

这就是你所要做的。现在，使用 opt，您可以使用“ --debug-counter”选项控制此代码何时触发。提供了两个计数器，skip和count。 skip是跳过代码路径执行的次数。 count是次数，一旦我们完成跳过，执行代码路径。

$ opt --debug-counter=passname-delete-instruction-skip=1,passname-delete-instruction-count=2 -passname

这将在我们第一次点击它时跳过上面的代码，然后执行两次，然后跳过其余的执行。

所以如果在下面的代码上执行：

%1 = **add** i32 %a, %b%2 = **add** i32 %a, %b%3 = **add** i32 %a, %b%4 = **add** i32 %a, %b

它会删除 number%2和%3。

在utils/bisect-skip-count中提供了一个实用程序来二进制搜索跳过和计数参数。它可用于自动最小化调试计数器变量的跳过和计数。

### [在调试代码时查看图表](https://llvm.org/docs/ProgrammersManual.html" \l "id40)

LLVM 中的几个重要数据结构是图：例如，由 LLVM [BasicBlocks 构成](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)的 CFG ，由 LLVM [MachineBasicBlocks](https://llvm.org/docs/CodeGenerator.html" \l "machinebasicblock)构成的 CFG ，以及[指令选择 DAG](https://llvm.org/docs/CodeGenerator.html" \l "selectiondag)。在许多情况下，在调试编译器的各个部分时，立即可视化这些图形是很好的。

LLVM 提供了几个可在调试构建中使用的回调来执行此操作。如果调用该Function::viewCFG()方法，例如，当前的 LLVM 工具将弹出一个窗口，其中包含函数的 CFG，其中每个基本块是图中的一个节点，每个节点包含块中的指令。同样，也存在Function::viewCFGOnly()（不包括指令）、MachineFunction::viewCFG()和 MachineFunction::viewCFGOnly()、以及SelectionDAG::viewGraph() 方法。例如，在 GDB 中，您通常可以使用诸如弹出窗口之类的东西。或者，您可以在代码中要调试的地方散布对这些函数的调用。call DAG.viewGraph()

让它工作需要少量的设置。在带有 X11 的 Unix 系统上，安装[graphviz](http://www.graphviz.org/)工具包，并确保“点”和“gv”在您的路径中。如果您在 macOS 上运行，请下载并安装 macOS [Graphviz 程序](http://www.pixelglow.com/graphviz/)并将其添加 /Applications/Graphviz.app/Contents/MacOS/（或安装在任何地方）到您的路径中。这些程序在配置、构建或运行 LLVM 时不需要存在，并且可以在活动调试会话期间在需要时简单地安装。

SelectionDAG已得到扩展，可以更轻松地 在大型复杂图中定位有趣的节点。从 gdb 中，如果你，那么下一个将突出显示指定颜色的节点（颜色的选择可以在[colors](http://www.graphviz.org/doc/info/colors.html)中找到。）可以提供更复杂的节点属性（选择可以在[Graph attributes](http://www.graphviz.org/doc/info/attrs.html)中找到。）如果你想重新启动并清除所有当前图形属性，然后就可以了 。call DAG.setGraphColor(node, "color")call DAG.viewGraph()call DAG.setGraphAttrs(node, "attributes")call DAG.clearGraphAttrs()

请注意，图形可视化功能是从发布版本中编译出来的，以减小文件大小。这意味着您需要 Debug+Asserts 或 Release+Asserts 构建才能使用这些功能。

## [为任务选择正确的数据结构](https://llvm.org/docs/ProgrammersManual.html" \l "id41)

LLVMllvm/ADT/目录中的数据结构非常多，我们常用的是STL数据结构。本节描述了您在选择一个时应该考虑的权衡。

第一步是选择你自己的冒险：你想要顺序容器、类集合容器还是类映射容器？选择容器时最重要的是您计划如何访问容器的算法属性。基于此，您应该使用：

* 一个[类似地图的](https://llvm.org/docs/ProgrammersManual.html" \l "ds-map)容器，如果你需要基于另一个值高效地查找一个值。Map-like 容器还支持高效的容器查询（无论键是否在地图中）。Map-like 容器通常不支持高效的反向映射（值到键）。如果需要，请使用两张地图。一些类似地图的容器还支持按排序顺序对键进行高效迭代。类似地图的容器是最昂贵的一种，只有在需要这些功能之一时才使用它们。
* 如果您需要将一堆东西放入自动消除重复项的容器中，则可以使用类似[集合的容器。](https://llvm.org/docs/ProgrammersManual.html" \l "ds-set)一些类似集合的容器支持按排序顺序对元素进行高效迭代。Set-like 容器比顺序容器更昂贵。
* [顺序](https://llvm.org/docs/ProgrammersManual.html" \l "ds-sequential)容器提供了最有效的方式来添加元素并跟踪它们被添加到集合中的顺序。它们允许重复并支持高效迭代，但不支持基于键的高效查找。
* [字符串](https://llvm.org/docs/ProgrammersManual.html" \l "ds-string)容器是用于字符或字节数组的专用顺序容器或引用结构。
* 位容器提供了一种有效的方法来存储和执行数字 id 集合上的集合操作，同时自动消除重复项[。](https://llvm.org/docs/ProgrammersManual.html" \l "ds-bit)对于要存储的每个标识符，位容器最多需要 1 位。

一旦确定了合适的容器类别，您就可以通过智能地选择该类别的成员来微调内存使用、常量因素和访问的缓存行为。请注意，常量因素和缓存行为可能很重要。例如，如果您有一个通常只包含几个元素（但也可能包含很多）的向量，那么使用 [SmallVector](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallvector)比使用[vector](https://llvm.org/docs/ProgrammersManual.html" \l "dss-vector)要好得多。这样做避免了（相对）昂贵的 malloc/free 调用，这使将元素添加到容器的成本相形见绌。

### [顺序容器（std::vector、std::list 等）](https://llvm.org/docs/ProgrammersManual.html" \l "id42)

根据您的需要，有多种顺序容器可供您使用。选择本节中的第一个将执行您想要的操作。

#### [llvm/ADT/ArrayRef.h](https://llvm.org/docs/ProgrammersManual.html" \l "id43)

该类llvm::ArrayRef是在接口中使用的首选类，该接口接受内存中元素的顺序列表并只从中读取。通过采用 an ArrayRef，可以向 API 传递固定大小的数组、 an std::vector、 anllvm::SmallVector以及内存中连续的任何其他内容。

#### [固定大小的数组](https://llvm.org/docs/ProgrammersManual.html" \l "id44)

固定大小的数组非常简单而且非常快。如果你确切地知道你有多少元素，或者你有多少元素的（低）上限，它们就很好。

#### [堆分配数组](https://llvm.org/docs/ProgrammersManual.html" \l "id45)

堆分配数组 ( new[]+ delete[]) 也很简单。如果元素的数量是可变的，如果您知道在分配数组之前需要多少元素，并且数组通常很大（如果不是，请考虑[SmallVector](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallvector)），它们就很好。堆分配数组的成本是新建/删除（也称为 malloc/free）的成本。另请注意，如果您使用构造函数分配类型的数组，则将为数组中的每个元素运行构造函数和析构函数（可调整大小的向量仅构造实际使用的那些元素）。

#### [llvm/ADT/TinyPtrVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id46)

TinyPtrVector<Type>是一个高度专业化的集合类，经过优化以避免在向量具有零个或一个元素的情况下进行分配。它有两个主要限制：1) 只能保存指针类型的值，2) 不能保存空指针。

由于此容器高度专业化，因此很少使用。

#### [llvm/ADT/SmallVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id47)

SmallVector<Type, N>是一个看起来和闻起来都像的简单类 vector<Type>：它支持高效迭代，按内存顺序排列元素（因此您可以在元素之间进行指针运算），支持高效的 p​​ush\_back/pop\_back 操作，支持对其元素的高效随机访问等。

**SmallVector 的主要优点是它为对象本身中的**一定数量的元素 (N) 分配空间。因此，如果 SmallVector 动态小于 N，则不会执行 malloc。在 malloc/free 调用比摆弄元素的代码昂贵得多的情况下，这可能是一个巨大的胜利。

这对“通常很小”的向量很有用（例如，一个块的前驱/后继的数量通常少于 8）。另一方面，这会使 SmallVector 本身的大小变大，因此您不想分配很多（这样做会浪费很多空间）。因此，SmallVectors 在堆栈上时最有用。

在内联元素的数量没有很好的选择时，N建议使用SmallVector<T>（即省略N）。这将选择适合在堆栈上分配的内联元素的默认数量（例如，尝试保持sizeof(SmallVector<T>)64 字节左右）。

SmallVector 还为 alloca.

SmallVector 比 std::vector 还具有一些其他的小优势， 因此比.SmallVector<Type, 0>std::vector<Type>

1. std::vector 是异常安全的，并且一些实现具有在 SmallVector 移动元素时复制元素的悲观化。
2. SmallVector 理解std::is\_trivially\_copyable<Type>并积极使用 realloc。
3. 许多 LLVM API 将 SmallVectorImpl 作为输出参数（参见下面的注释）。
4. N 等于 0 的 SmallVector 比 64 位平台上的 std::vector 小，因为它使用unsigned（而不是void\*）作为其大小和容量。

**笔记**

更喜欢使用ArrayRef<T>orSmallVectorImpl<T>作为参数类型。

它很少适合用作参数类型。如果 API 仅从向量中读取，则应使用[ArrayRef](https://llvm.org/docs/ProgrammersManual.html" \l "dss-arrayref)。即使 API 更新了矢量，“小尺寸”也不太可能相关；这样的 API 应该使用 类，它是“向量标头”（和方法），后面没有分配的元素。请注意，继承自 因此转换是隐式的并且不产生任何成本。例如SmallVector<T, N>SmallVectorImpl<T>SmallVector<T, N>SmallVectorImpl<T>

*// DISCOURAGED: Clients cannot pass e.g. raw arrays.*hardcodedContiguousStorage(**const** SmallVectorImpl<Foo> &In);*// ENCOURAGED: Clients can pass any contiguous storage of Foo.*allowsAnyContiguousStorage(ArrayRef<Foo> In);

void someFunc1() {

Foo Vec[] = { */\* ... \*/* };

hardcodedContiguousStorage(Vec); *// Error.*

allowsAnyContiguousStorage(Vec); *// Works.*}

*// DISCOURAGED: Clients cannot pass e.g. SmallVector<Foo, 8>.*hardcodedSmallSize(SmallVector<Foo, 2> &Out);*// ENCOURAGED: Clients can pass any SmallVector<Foo, N>.*allowsAnySmallSize(SmallVectorImpl<Foo> &Out);

void someFunc2() {

SmallVector<Foo, 8> Vec;

hardcodedSmallSize(Vec); *// Error.*

allowsAnySmallSize(Vec); *// Works.*}

尽管Impl名称中带有“ ”，但 SmallVectorImpl 被广泛使用并且不再是“私有实现”。像这样的名字 SmallVectorHeader可能更合适。

#### [<向量>](https://llvm.org/docs/ProgrammersManual.html" \l "id48)

std::vector<T>深受喜爱和尊重。但是， 由于上面列出的优点，通常是更好的选择。std::vector 在需要存储多个元素或与需要向量的代码交互时仍然有用 :)。SmallVector<T, 0>UINT32\_MAX

关于 std::vector 的一个值得注意的地方：避免这样的代码：

**for** ( ... ) {

std::vector<foo> V;

*// make use of V.*}

相反，将其写为：

std::vector<foo> V;**for** ( ... ) {

*// make use of V.*

V.clear();}

这样做将（至少）节省一个堆分配并在循环的每次迭代中释放。

#### [<双端队列>](https://llvm.org/docs/ProgrammersManual.html" \l "id49)

std::deque在某种意义上，是 . 的通用版本std::vector。与 一样std::vector，它提供了恒定时间随机访问和其他类似的属性，但它也提供了对列表前端的高效访问。它不保证内存中元素的连续性。

作为这种额外灵活性的交换，std::deque具有比 高得多的常数因子成本std::vector。如果可能的话，使用std::vector或更便宜的东西。

#### [<列表>](https://llvm.org/docs/ProgrammersManual.html" \l "id50)

std::list是一个极低效的类，很少有用。它为插入其中的每个元素执行堆分配，因此具有极高的常数因子，特别是对于小数据类型。 std::list同样只支持双向迭代，不支持随机访问迭代。

作为这种高成本的交换，std::list 支持对列表两端的高效访问（类似于std::deque，但不同于std::vectoror SmallVector）。此外，std::list 的迭代器失效特性比向量类强：在列表中插入或移除元素不会使迭代器或指向列表中其他元素的指针失效。

#### [llvm/ADT/ilist.h](https://llvm.org/docs/ProgrammersManual.html" \l "id51)

ilist<T>实现一个“侵入式”双向链表。它是侵入式的，因为它需要元素来存储和提供对列表的上一个/下一个指针的访问。

ilist具有与 相同的缺点std::list，另外还需要 ilist\_traits元素类型的实现，但它提供了一些新颖的特性。特别是，它可以有效地存储多态对象，当一个元素被插入或从列表中删除时，特征类会得到通知，并且ilists 保证支持恒定时间的拼接操作。

Anilist和 aniplist是彼此的using别名，后者目前仅出于历史目的而存在。

这些属性正是我们想要的Instructions 和基本块之类的东西，这就是为什么这些是用ilists 实现的。

以下小节解释了相关的兴趣类别：

* [ilist\_traits](https://llvm.org/docs/ProgrammersManual.html" \l "dss-ilist-traits)
* [llvm/ADT/ilist\_node.h](https://llvm.org/docs/ProgrammersManual.html" \l "dss-ilist-node)
* [哨兵](https://llvm.org/docs/ProgrammersManual.html" \l "dss-ilist-sentinel)

#### [llvm/ADT/PackedVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id52)

对于每个值仅使用少量位来存储值向量很有用。除了类向量容器的标准操作外，它还可以执行“或”集合操作。

例如：

**enum** **State** {

None = 0x0,

FirstCondition = 0x1,

SecondCondition = 0x2,

Both = 0x3};

State get() {

PackedVector<State, 2> Vec1;

Vec1.push\_back(FirstCondition);

PackedVector<State, 2> Vec2;

Vec2.push\_back(SecondCondition);

Vec1 |= Vec2;

**return** Vec1[0]; *// returns 'Both'.*}

#### [ilist\_traits](https://llvm.org/docs/ProgrammersManual.html" \l "id53)

ilist\_traits<T>是ilist<T>的定制机制。ilist<T> publicly 派生自这个 traits 类。

#### [llvm/ADT/ilist\_node.h](https://llvm.org/docs/ProgrammersManual.html" \l "id54)

ilist\_node<T>ilist<T>以默认方式实现（和类似容器）期望的前向和后向链接。

ilist\_node<T>s 意味着嵌入到节点类型T中，通常 T公开派生自ilist\_node<T>。

#### [哨兵](https://llvm.org/docs/ProgrammersManual.html" \l "id55)

ilist我们还有另一个必须考虑的专业。要想成为C++生态的好公民，就需要支持标准的容器操作，比如begin和end迭代器等。另外，在 非空的情况下，operator--必须在迭代器上正常工作。endilist

这个问题唯一明智的解决方案是分配一个所谓的哨兵 和侵入列表，它充当end迭代器，提供到最后一个元素的反向链接。然而，根据 C++ 约定，operator++超出哨兵是非法的，而且它也不能被取消引用。

这些约束允许在ilist如何分配和存储哨兵方面有一些实现自由。相应的政策由 规定 ilist\_traits<T>。默认情况下，T每当需要哨兵时，a 就会在堆上分配。

T虽然在大多数情况下默认策略就足够了，但如果不提供默认构造函数，它可能会崩溃 。此外，在 s 的许多实例的情况下，ilist关联哨兵的内存开销被浪费了。为了缓解 T-sentinel 数量众多的情况，有时会采用一种技巧，导致 ghostly sentinels。

幽灵哨兵是通过特制的方式获得的，将哨兵与内存中ilist\_traits<T>的实例叠加。ilist指针运算用于获取 sentinel，它是相对于ilist的 this指针。由一个额外的ilist指针扩充，该指针用作哨兵的反向链接。这是幽灵哨兵中唯一可以合法访问的字段。

#### [其他顺序容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "id56)

其他 STL 容器可用，例如std::string.

还有各种 STL 适配器类，例如std::queue、 std::priority\_queue、std::stack等。它们提供了对底层容器的简化访问，但不影响容器本身的成本。

### [类似字符串的容器](https://llvm.org/docs/ProgrammersManual.html" \l "id57)

在 C 和 C++ 中有多种传递和使用字符串的方法，LLVM 添加了一些新选项以供选择。选择此列表中的第一个选项，可以满足您的需求，它们是根据相对成本排序的。

请注意，通常最好不要将字符串作为's 传递。它们有很多问题，包括它们不能表示嵌入的 nul (“0”) 字符，并且没有有效的可用长度。“ ”的一般替换是 StringRef。const char\*const char\*

有关为 API 选择字符串容器的更多信息，请参阅 [传递字符串](https://llvm.org/docs/ProgrammersManual.html" \l "string-apis)。

#### [llvm/ADT/StringRef.h](https://llvm.org/docs/ProgrammersManual.html" \l "id58)

StringRef 类是一个简单的值类，它包含指向字符和长度的指针，并且与[ArrayRef](https://llvm.org/docs/ProgrammersManual.html" \l "dss-arrayref)类非常相关（但专用于字符数组）。因为 StringRef 带有一个长度，它可以安全地处理其中嵌入了 nul 字符的字符串，获取长度不需要调用 strlen，它甚至有非常方便的 API 来切片和切分它所代表的字符范围。

StringRef 非常适合传递已知存在的简单字符串，因为它们是 C 字符串文字、std::string、C 数组或 SmallVector。这些情况中的每一个都有到 StringRef 的有效隐式转换，这不会导致执行动态 strlen。

StringRef 有几个主要的限制，这使得更强大的字符串容器变得有用：

1. 您不能将 StringRef 直接转换为“const char\*”，因为无法添加尾随 nul（与各种更强大的类上的 .c\_str() 方法不同）。
2. StringRef 不拥有或保留底层字符串字节。因此它很容易导致悬空指针，并且在大多数情况下不适合嵌入数据结构（相反，使用 std::string 或类似的东西）。
3. 出于同样的原因，如果方法“计算”结果字符串，则 StringRef 不能用作方法的返回值。相反，使用 std::string。
4. StringRef 不允许您改变指向的字符串字节，也不允许您在范围内插入或删除字节。对于这样的编辑操作，它与[Twine](https://llvm.org/docs/ProgrammersManual.html" \l "dss-twine) 类进行互操作。

由于其优点和局限性，函数采用 StringRef 而对象上的方法返回指向它拥有的某个字符串的 StringRef 是很常见的。

#### [llvm/ADT/Twine.h](https://llvm.org/docs/ProgrammersManual.html" \l "id59)

Twine 类用作 API 的中间数据类型，这些 API 想要采用可以通过一系列连接内联构造的字符串。Twine 的工作原理是在堆栈上形成 Twine 数据类型（一个简单的值对象）的递归实例作为临时对象，将它们链接在一起形成一棵树，然后在使用 Twine 时将其线性化。Twine 只能安全地用作函数的参数，并且应该始终是 const 引用，例如：

void foo(**const** Twine &T);...StringRef X = ...unsigned i = ...foo(X + "." + Twine(i));

此示例通过将值连接在一起形成类似“blarg.42”的字符串，而不形成包含“blarg”或“blarg.”的中间字符串。

因为 Twine 是用堆栈上的临时对象构建的，并且因为这些实例在当前语句结束时被销毁，所以它本质上是一个危险的 API。例如，这个简单的变体包含未定义的行为并且可能会崩溃：

void foo(**const** Twine &T);...StringRef X = ...unsigned i = ...**const** Twine &Tmp = X + "." + Twine(i);foo(Tmp);

…因为临时对象在调用之前就被销毁了。也就是说，Twine 比中间的 std::string 临时对象更有效，并且它们与 StringRef 一起工作得很好。请注意它们的局限性。

#### [llvm/ADT/SmallString.h](https://llvm.org/docs/ProgrammersManual.html" \l "id60)

SmallString 是 SmallVector 的子类，[它](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallvector)添加了一些方便的 API，例如 += 接受 StringRef 的 API。SmallString 避免在预分配空间足以容纳其数据的情况下分配内存，并在需要时回调一般堆分配。由于它拥有它的数据，所以使用起来非常安全，并且支持字符串的完全变异。

与 SmallVector 一样，SmallString 的最大缺点是它们的 sizeof。虽然它们针对小字符串进行了优化，但它们本身并不是特别小。这意味着它们非常适用于堆栈上的临时暂存缓冲区，但通常不应放入堆中：很少将 SmallString 视为频繁分配的堆数据结构的成员或按值返回。

#### [标准：：字符串](https://llvm.org/docs/ProgrammersManual.html" \l "id61)

标准 C++ std::string 类是一个非常通用的类（如 SmallString）拥有其基础数据。sizeof(std::string) 非常合理，因此它可以嵌入到堆数据结构中并按值返回。另一方面，std::string 对于内联编辑（例如将一堆东西连接在一起）效率非常低，并且因为它是由标准库提供的，所以它的性能特征在很大程度上取决于宿主标准库（例如 libc++ 和 MSVC）提供一个高度优化的字符串类，GCC 包含一个非常慢的实现）。

std::string 的主要缺点是几乎每个使它们变大的操作都可以分配内存，这很慢。因此，最好使用 SmallVector 或 Twine 作为暂存缓冲区，然后使用 std::string 来保存结果。

### [类集合容器（std::set、SmallSet、SetVector 等）](https://llvm.org/docs/ProgrammersManual.html" \l "id62)

当您需要将多个值规范化为单个表示时，类似集合的容器很有用。对于如何做到这一点，有几种不同的选择，提供各种权衡。

#### [一个排序的“向量”](https://llvm.org/docs/ProgrammersManual.html" \l "id63)

如果你打算插入很多元素，然后做很多查询，一个很好的方法是使用 std::vector（或其他顺序容器）和 std::sort+std::unique 来删除重复项。如果您的使用模式具有这两个不同的阶段（插入然后查询），并且可以与[顺序容器](https://llvm.org/docs/ProgrammersManual.html" \l "ds-sequential)的良好选择相结合，则此方法非常有效。

这种组合提供了几个不错的属性：结果数据在内存中是连续的（有利于缓存局部性），分配很少，易于寻址（最终向量中的迭代器只是索引或指针），并且可以使用标准二分搜索（例如 std::lower\_bound，如果你想要比较相等的元素的整个范围，使用std::equal\_range）。

#### [llvm/ADT/SmallSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id64)

如果您有一个类似集合的数据结构，该结构通常很小并且其元素也相当小，那么 a是一个不错的选择。这个集合有 N 个元素的空间（因此，如果集合动态小于 N，则不需要 malloc 流量）并通过简单的线性搜索访问它们。当集合增长超过 N 个元素时，它会分配一个更昂贵的表示来保证高效访问（对于大多数类型，它回退到[std::set](https://llvm.org/docs/ProgrammersManual.html" \l "dss-set)，但对于指针它使用更好的东西， [SmallPtrSet](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallptrset)。SmallSet<Type, N>

这个类的神奇之处在于它可以非常有效地处理小集合，但可以优雅地处理非常大的集合而不会降低效率。

#### [llvm/ADT/SmallPtrSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id65)

SmallPtrSet具有的所有优点SmallSet（并且SmallSet指针的 a 透明地用 a 实现SmallPtrSet）。如果执行多于 N 次插入，将分配单个二次探测哈希表并根据需要增长，提供极其高效的访问（恒定时间插入/删除/具有低常数因子的查询）并且对 malloc 流量非常吝啬。

请注意，与[std::set](https://llvm.org/docs/ProgrammersManual.html" \l "dss-set)SmallPtrSet不同， 只要发生插入， 的迭代器就会失效。此外，迭代器访问的值不会按排序顺序访问。

#### [llvm/ADT/StringSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id66)

StringSet[是StringMap<char>](https://llvm.org/docs/ProgrammersManual.html" \l "dss-stringmap)的薄包装器，它允许高效存储和检索唯一字符串。

功能上类似于SmallSet<StringRef>，StringSet也支持迭代。（迭代器取消对 a 的引用StringMapEntry<char>，因此您需要调用i->getKey()以访问 StringSet 的项目。）另一方面，StringSet不支持[SmallSet](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallset)和[SmallPtrSet](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallptrset)支持的范围插入和复制构造。

#### [llvm/ADT/DenseSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id67)

DenseSet 是一个简单的二次探查哈希表。它擅长支持小值：它使用单个分配来保存当前插入到集合中的所有对。DenseSet 是一种很好的方法来唯一化不是简单指针的小值（使用[SmallPtrSet](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallptrset)作为指针）。请注意，DenseSet 对值类型的要求与 DenseMap [相同](https://llvm.org/docs/ProgrammersManual.html" \l "dss-densemap)。

#### [llvm/ADT/SparseSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id68)

SparseSet 包含少量由中等大小的无符号键标识的对象。它使用大量内存，但提供的操作速度几乎与矢量一样快。典型的密钥是物理寄存器、虚拟寄存器或编号的基本块。

SparseSet 对于需要非常快速的清除/查找/插入/擦除和对小集合进行快速迭代的算法很有用。它不适用于构建复合数据结构。

#### [llvm/ADT/SparseMultiSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id69)

SparseMultiSet 向 SparseSet 添加多重集行为，同时保留 SparseSet 的理想属性。与 SparseSet 一样，它通常使用大量内存，但提供的操作速度几乎与向量一样快。典型的密钥是物理寄存器、虚拟寄存器或编号的基本块。

SparseMultiSet 对于需要非常快速地清除/查找/插入/擦除整个集合以及对共享密钥的元素集进行迭代的算法很有用。它通常是比使用复合数据结构（例如向量向量、向量图）更有效的选择。它不适用于构建复合数据结构。

#### [llvm/ADT/FoldingSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id70)

FoldingSet 是一个聚合类，非常擅长唯一化昂贵的创建对象或多态对象。它是链式哈希表与侵入式链接（唯一对象需要从 FoldingSetNode 继承）的组合，使用[SmallVector](https://llvm.org/docs/ProgrammersManual.html" \l "dss-smallvector)作为其 ID 过程的一部分。

考虑这样一种情况，您想要为复杂对象（例如，代码生成器中的节点）实现“getOrCreateFoo”方法。客户端有它想要生成的内容的描述**（**它知道操作码和所有操作数），但我们不想“新建”一个节点，然后尝试将它插入到一个集合中却发现它已经存在，此时我们将不得不删除它并返回已经存在的节点。

为了支持这种客户端风格，FoldingSet 使用 FoldingSetNodeID（包装 SmallVector）执行查询，该 ID 可用于描述我们要查询的元素。查询要么返回与 ID 匹配的元素，要么返回一个指示插入位置的不透明 ID。ID的构造通常不需要堆流量。

由于 FoldingSet 使用侵入式链接，它可以支持集合中的多态对象（例如，您可以将 SDNode 实例与 LoadSDNodes 混合）。因为元素是单独分配的，所以指向元素的指针是稳定的：插入或删除元素不会使指向其他元素的任何指针失效。

#### [<设置>](https://llvm.org/docs/ProgrammersManual.html" \l "id71)

std::set是一个合理的全能类，它在很多方面都不错，但一无所获。std::set 为每个插入的元素分配内存（因此它是非常密集的 malloc）并且通常为集合中的每个元素存储三个指针（因此增加了大量的每个元素空间开销）。它提供有保证的 log(n) 性能，从复杂性的角度来看并不是特别快（特别是如果集合的元素比较昂贵，如字符串），并且具有极高的查找、插入和删除常数因子。

std::set 的优点是它的迭代器是稳定的（从集合中删除或插入元素不会影响迭代器或指向其他元素的指针）并且保证对集合的迭代是按排序顺序进行的。如果集合中的元素很大，那么指针和 malloc 流量的相对开销就不是什么大问题，但如果集合中的元素很小，std::set 几乎从来都不是一个好的选择。

#### [llvm/ADT/SetVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id72)

LLVMSetVector<Type>是一个适配器类，它结合了您选择的类集合容器和[顺序容器](https://llvm.org/docs/ProgrammersManual.html" \l "ds-sequential)。它提供的重要属性是高效插入和唯一性（忽略重复元素）和迭代支持。它通过将元素插入类集合容器和顺序容器中来实现这一点，使用类集合容器进行唯一化，使用顺序容器进行迭代。

SetVector 与其他集合的区别在于迭代顺序保证与插入 SetVector 的顺序相匹配。此属性对于指针集之类的东西非常重要。因为指针值是不确定的（例如，在不同机器上运行的程序会有所不同），因此迭代集合中的指针将不会按明确定义的顺序进行。

SetVector 的缺点是它需要的空间是普通集合的两倍，并且具有来自类集合容器和它使用的顺序容器的常数因子之和。**仅**当您需要以确定的顺序遍历元素时才使用它。SetVector 在（线性时间）之外删除元素的成本也很高，除非您使用其更快的“pop\_back”方法。

SetVector是一个默认使用的适配器类，底层容器std::vector的大小SmallSet为 16，因此它非常昂贵。但是，"llvm/ADT/SetVector.h"还提供了一个SmallSetVector类，它默认使用指定大小的SmallVector和SmallSet。如果您使用它，并且您的集合动态小于N，您将节省大量堆流量。

#### [llvm/ADT/UniqueVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id73)

UniqueVector 类似于[SetVector](https://llvm.org/docs/ProgrammersManual.html" \l "dss-setvector)，但它为插入到集合中的每个元素保留唯一 ID。它内部包含一个映射和一个向量，并为插入到集合中的每个值分配一个唯一的 ID。

UniqueVector 非常昂贵：它的成本是维护 map 和 vector 的成本之和，它具有高复杂度，高常数因子，并且会产生大量的 malloc 流量。应该避免。

#### [llvm/ADT/ImmutableSet.h](https://llvm.org/docs/ProgrammersManual.html" \l "id74)

ImmutableSet 是基于 AVL 树的不可变（函数式）集合实现。添加或删除元素是通过 Factory 对象完成的，并导致创建新的 ImmutableSet 对象。如果具有给定内容的 ImmutableSet 已经存在，则返回现有的；相等性与 FoldingSetNodeID 进行比较。添加或删除操作的时间和空间复杂度与原始集合的大小成对数关系。

没有返回集合元素的方法，您只能检查成员资格。

#### [其他类似集合的容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "id75)

STL 提供了几个其他选项，例如 std::multiset 和 std::unordered\_set。我们从不使用像 unordered\_set 这样的容器，因为它们通常非常昂贵（每次插入都需要 malloc）。

如果您对消除重复项不感兴趣，std::multiset 很有用，但具有[std::set](https://llvm.org/docs/ProgrammersManual.html" \l "dss-set)的所有缺点。排序向量（您不删除重复的条目）或其他一些方法几乎总是更好。

### [类地图容器（std::map、DenseMap 等）](https://llvm.org/docs/ProgrammersManual.html" \l "id76)

当您想将数据与键相关联时，类似地图的容器很有用。像往常一样，有很多不同的方法可以做到这一点。:)

#### [一个排序的“向量”](https://llvm.org/docs/ProgrammersManual.html" \l "id77)

[如果您的使用模式遵循严格的先插入后查询方法，您可以简单地使用与排序向量](https://llvm.org/docs/ProgrammersManual.html" \l "dss-sortedvectorset)相同的方法来处理类似集合的容器。唯一的区别是您的查询函数（使用 std::lower\_bound 来获得高效的 log(n) 查找）应该只比较键，而不是键和值。这产生了与集合的排序向量相同的优势。

#### [llvm/ADT/StringMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id78)

字符串通常用作映射中的键，并且难以有效地支持它们：它们是可变长度的，当长时散列和比较效率低下，复制昂贵等。StringMap 是专门为解决这些问题而设计的容器。它支持将任意范围的字节映射到任意其他对象。

StringMap 实现使用二次探测哈希表，其中存储桶存储指向堆分配条目（和其他一些东西）的指针。映射中的条目必须是堆分配的，因为字符串是可变长度的。字符串数据（键）和元素对象（值）与紧跟在元素对象之后的字符串数据存储在同一个分配中。此容器保证“ (char\*)(&Value+1)”指向值的键字符串。

StringMap 之所以非常快，有几个原因：二次探测对于查找的缓存效率非常高，查找元素时不会重新计算存储桶中字符串的哈希值，StringMap 在查找值时很少需要触及不相关对象的内存（即使发生哈希冲突），哈希表增长也不会重新计算表中已有字符串的哈希值，并且映射中的每一对都存储在单个分配中（字符串数据存储在与 a 的值相同的分配中一对）。

StringMap 还提供采用字节范围的查询方法，因此它只会在将值插入表中时才复制字符串。

但是，不保证 StringMap 迭代顺序是确定性的，因此任何需要它的用途都应该改用 std::map。

#### [llvm/ADT/IndexedMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id79)

IndexedMap 是一个专门的容器，用于将小稠密整数（或可以映射到小稠密整数的值）映射到其他类型。它在内部实现为具有映射函数的向量，可将键映射到密集的整数范围。

这对于像 LLVM 代码生成器中的虚拟寄存器这样的情况很有用：它们有一个密集映射，由编译时常量（第一个虚拟寄存器 ID）偏移。

#### [llvm/ADT/DenseMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id80)

DenseMap 是一个简单的二次探查哈希表。它擅长支持小键和值：它使用单个分配来保存当前插入映射中的所有对。DenseMap 是将指针映射到指针或将其他小类型相互映射的好方法。

但是，您应该了解 DenseMap 的几个方面。与 map 不同，只要发生插入，DenseMap 中的迭代器就会失效。另外，由于DenseMap为大量键/值对分配空间（默认以64开头），如果你的键或值很大，它会浪费很多空间。最后，如果尚不支持，您必须为所需的键实现 DenseMapInfo 的部分特化。这需要告诉 DenseMap 它内部需要的两个特殊标记值（永远不能插入到地图中）。

DenseMap 的 find\_as() 方法支持使用备用键类型的查找操作。这在普通密钥类型的构造成本高但比较便宜的情况下很有用。DenseMapInfo 负责为使用的每个备用密钥类型定义适当的比较和散列方法。

#### [llvm/IR/ValueMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id81)

ValueMap 是[DenseMap](https://llvm.org/docs/ProgrammersManual.html" \l "dss-densemap)映射 Value\*（或子类）到另一种类型的包装器。当一个 Value 被删除或 RAUW'ed 时，ValueMap 将自我更新，以便新版本的键映射到相同的值，就像键是 WeakVH 一样。Config 您可以通过将参数传递给 ValueMap 模板来准确配置这是如何发生的，以及这两个事件还会发生什么。

#### [llvm/ADT/IntervalMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id82)

IntervalMap 是用于小键和值的紧凑映射。它映射键间隔而不是单个键，并且会自动合并相邻的间隔。当地图只包含几个区间时，它们存储在地图对象本身中以避免分配。

IntervalMap 迭代器非常大，因此不应将它们作为 STL 迭代器传递。重量级迭代器允许更小的数据结构。

#### [llvm/ADT/IntervalTree.h](https://llvm.org/docs/ProgrammersManual.html" \l "id83)

llvm::IntervalTree是一个保存间隔的灯树数据结构。它允许找到与任何给定点重叠的所有间隔。目前，它不支持任何删除或重新平衡操作。

IntervalTree 被设计为设置一次，然后在没有任何进一步添加的情况下进行查询。

#### [<地图>](https://llvm.org/docs/ProgrammersManual.html" \l "id84)

std::map 具有与[std::set](https://llvm.org/docs/ProgrammersManual.html" \l "dss-set)相似的特性：它使用每对插入到映射中的单个分配，它提供具有极大常数因子的 log(n) 查找，每对 3 个指针的空间惩罚地图等

当您的键或值非常大，如果您需要按排序顺序遍历集合，或者如果您需要稳定的迭代器进入映射（即它们不会在插入或删除时失效）时，std::map 最有用另一个元素发生）。

#### [llvm/ADT/MapVector.h](https://llvm.org/docs/ProgrammersManual.html" \l "id85)

MapVector<KeyT,ValueT>提供 DenseMap 接口的子集。主要区别在于迭代顺序保证为插入顺序，这使其成为指针映射上非确定性迭代的简单（但有些昂贵）解决方案。

它是通过从键映射到键值对向量中的索引来实现的。这提供了快速查找和迭代，但有两个主要缺点：密钥被存储两次并且删除元素需要线性时间。如果有必要删除元素，最好使用 remove\_if().

#### [llvm/ADT/IntEqClasses.h](https://llvm.org/docs/ProgrammersManual.html" \l "id86)

IntEqClasses 提供小整数等价类的紧凑表示。最初，0..n-1 范围内的每个整数都有自己的等价类。可以通过将两个类代表传递给 join(a, b) 方法来连接类。当 findLeader() 返回相同的代表时，两个整数属于同一类。

一旦形成了所有等价类，就可以压缩映射，使每个整数 0..n-1 映射到 0..m-1 范围内的等价类编号，其中 m 是等价类的总数。地图必须先解压缩，然后才能再次编辑。

#### [llvm/ADT/ImmutableMap.h](https://llvm.org/docs/ProgrammersManual.html" \l "id87)

ImmutableMap 是基于 AVL 树的不可变（功能）映射实现。添加或删除元素是通过 Factory 对象完成的，并导致创建新的 ImmutableMap 对象。如果给定键集的 ImmutableMap 已经存在，则返回现有的；相等性与 FoldingSetNodeID 进行比较。添加或删除操作的时间和空间复杂度与原始映射的大小成对数关系。

#### [其他类似地图的容器选项](https://llvm.org/docs/ProgrammersManual.html" \l "id88)

STL 提供了几个其他选项，例如 std::multimap 和 std::unordered\_map。我们从不使用像 unordered\_map 这样的容器，因为它们通常非常昂贵（每次插入都需要 malloc）。

如果您想将一个键映射到多个值，std::multimap 很有用，但它具有 std::map 的所有缺点。排序向量或其他一些方法几乎总是更好。

### [位存储容器](https://llvm.org/docs/ProgrammersManual.html" \l "id89)

有几个位存储容器，选择何时使用每个容器相对简单。

另一种选择是std::vector<bool>：我们不鼓励使用它有两个原因 1) 许多常见编译器（例如 GCC 的常用版本）中的实现效率极低，以及 2) C++ 标准委员会可能会弃用此容器和/或对其进行重大更改不知何故。无论如何，请不要使用它。

#### [位向量](https://llvm.org/docs/ProgrammersManual.html" \l "id90)

BitVector 容器为操作提供了一个动态大小的位集。它支持单个位设置/测试，以及设置操作。设置操作需要时间 O（位向量的大小），但操作是一次一个字执行的，而不是一次一位执行的。与其他容器相比，这使得 BitVector 对于集合操作非常快。当您期望设置的位数很高（即密集集）时，请使用 BitVector。

#### [小位向量](https://llvm.org/docs/ProgrammersManual.html" \l "id91)

SmallBitVector 容器提供与 BitVector 相同的接口，但它针对只需要少量位（少于 25 位左右）的情况进行了优化。它还透明地支持更大的位计数，但效率略低于普通 BitVector，因此 SmallBitVector 应该只在大计数很少见的情况下使用。

此时，SmallBitVector 不支持集合运算（and、or、xor），其 operator[] 不提供可分配的左值。

#### [稀疏位向量](https://llvm.org/docs/ProgrammersManual.html" \l "id92)

SparseBitVector 容器与 BitVector 非常相似，但有一个主要区别：仅存储设置的位。当集合稀疏时，这使得 SparseBitVector 的空间效率比 BitVector 高得多，并且使集合操作 O（集合位数）而不是 O（宇宙大小）。SparseBitVector 的缺点是随机位的设置和测试是 O(N)，并且在大型 SparseBitVector 上，这可能比 BitVector 慢。在我们的实现中，按排序顺序（正向或反向）设置或测试位是 O(1) 最坏情况。测试和设置当前位的 128 位（取决于大小）内的位也是 O(1)。作为一般陈述，测试/设置 SparseBitVector 中的位是 O（与最后设置位的距离）。

#### [合并位向量](https://llvm.org/docs/ProgrammersManual.html" \l "id93)

CoalescingBitVector 容器在原理上类似于 SparseBitVector，但经过优化以紧凑地表示大的连续范围的设置位。它通过将连续的设置位范围合并为间隔来实现这一点。在 CoalescingBitVector 中搜索位的时间复杂度为 O(log(连续范围之间的间隙))。

当设置位范围之间的差距很大时，CoalescingBitVector 是比 BitVector 更好的选择。当 find() 操作必须具有快速、可预测的性能时，它是比 SparseBitVector 更好的选择。但是，它不是表示具有许多非常短范围的集合的好选择。例如，集合 {2\*x : x in [0, n)}将是一个病态输入。

## [调试](https://llvm.org/docs/ProgrammersManual.html" \l "id94)

为一些核心 LLVM 库提供了一些GDB[漂亮的打印机。](https://sourceware.org/gdb/onlinedocs/gdb/Pretty-Printing.html)要使用它们，请执行以下命令（或将其添加到您的~/.gdbinit）：

source /path/to/llvm/src/utils/gdb-scripts/prettyprinters.py

[启用打印漂亮](http://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_57.html)选项以避免将数据结构打印为大文本块也可能很方便。

## [常见操作的有用提示](https://llvm.org/docs/ProgrammersManual.html" \l "id95)

本节介绍如何对 LLVM 代码执行一些非常简单的转换。这是为了给出常用习语的示例，展示 LLVM 转换的实用方面。

因为这是一个“操作方法”部分，您还应该阅读您将使用的主要类。[核心 LLVM 类层次结构参考](https://llvm.org/docs/ProgrammersManual.html" \l "coreclasses)包含您应该了解的主要类的详细信息和描述。

### [基本检查和遍历例程](https://llvm.org/docs/ProgrammersManual.html" \l "id96)

LLVM 编译器基础设施有许多不同的数据结构可以被遍历。以C++标准模板库为例，遍历这些各种数据结构的技巧基本相同。对于可枚举的值序列，XXXbegin()函数（或方法）返回一个指向序列开头的迭代器，XXXend() 函数返回一个指向序列最后一个有效元素之后的迭代器，并且有一些XXXiterator数据类型是通用的这两个操作。

因为迭代模式在程序表示的许多不同方面是通用的，所以可以对它们使用标准模板库算法，并且更容易记住如何迭代。首先，我们展示一些需要遍历的数据结构的常见示例。其他数据结构的遍历方式非常相似。

#### [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "id97)**[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "id97)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "id97)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "id97)**

Function有一个您想要以某种方式进行转换的实例是很常见的；特别是，您想操纵它BasicBlock的 s。为此，您需要遍历BasicBlock构成Function. 以下是打印 a 的名称及其包含的 sBasicBlock数量的示例：Instruction

Function &Func = ...**for** (BasicBlock &**BB** : Func)

*// Print out the name of the basic block if it has one, and then the*

*// number of instructions that it contains*

errs() << "Basic block (name=" << BB.getName() << ") has "

<< BB.size() << " instructions.**\n**";

#### [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "id98)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id98)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "id98)**[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "id98)**

就像处理BasicBlocks in Functions 时一样，很容易遍历组成BasicBlocks 的各个指令。这是一个代码片段，它打印出 a 中的每条指令BasicBlock：

BasicBlock& BB = ...**for** (Instruction &**I** : BB)

*// The next statement works since operator<<(ostream&,...)*

*// is overloaded for Instruction&*

errs() << I << "**\n**";

然而，这并不是打印出 BasicBlock! 由于 ostream 运算符几乎对您关心的任何内容都超载，因此您可以只调用基本块本身的打印例程：。errs() << BB << "\n";

#### [遍历](https://llvm.org/docs/ProgrammersManual.html" \l "id99)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id99)**[a](https://llvm.org/docs/ProgrammersManual.html" \l "id99)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "id99)**

如果您发现您经常迭代 aFunction的 BasicBlocks 然后是 thatBasicBlock的Instructions， InstIterator应该改用它。您需要包含 llvm/IR/InstIterator.h( [doxygen](https://llvm.org/doxygen/InstIterator_8h.html) )，然后 InstIterator在您的代码中显式实例化 s。下面是一个小示例，展示了如何将函数中的所有指令转储到标准错误流：

#include *"llvm/IR/InstIterator.h"*

*// F is a pointer to a Function instance***for** (inst\_iterator I = inst\_begin(F), E = inst\_end(F); I != E; ++I)

errs() << \*I << "**\n**";

很简单，不是吗？您还可以使用InstIterators 以其初始内容填充工作列表。例如，如果你想初始化一个工作列表以包含FunctionF 中的所有指令，你需要做的就是：

std::set<Instruction\*> worklist;*// or better yet, SmallPtrSet<Instruction\*, 64> worklist;*

**for** (inst\_iterator I = inst\_begin(F), E = inst\_end(F); I != E; ++I)

worklist.insert(&\*I);

STL 集现在将包含 F 指向的worklist所有指令。Function

#### [将迭代器变成类指针（反之亦然）](https://llvm.org/docs/ProgrammersManual.html" \l "id100)

有时，当您手头只有一个迭代器时，获取对类实例的引用（或指针）会很有用。好吧，从迭代器中提取引用或指针非常简单。假设那i是一个 BasicBlock::iterator并且j是一个BasicBlock::const\_iterator：

Instruction& inst = \*i; *// Grab reference to instruction reference*Instruction\* pinst = &\*i; *// Grab pointer to instruction reference***const** Instruction& inst = \*j;

然而，您将在 LLVM 框架中使用的迭代器是特殊的：它们会在需要时自动转换为 ptr-to-instance 类型。您可以简单地将迭代器分配给正确的指针类型，而不是取消引用迭代器然后获取结果的地址，并且作为分配的结果获得取消引用和寻址操作（在幕后，这是一个结果重载铸造机制）。因此最后一个例子的第二行，

Instruction \*pinst = &\*i;

在语义上等同于

Instruction \*pinst = i;

也可以将一个类指针变成相应的迭代器，这是一个常量时间操作（非常高效）。以下代码片段说明了 LLVM 迭代器提供的转换构造函数的使用。通过使用这些，您可以显式获取某物的迭代器，而无需通过对某些结构的迭代实际获得它：

void printNextInstruction(Instruction\* inst) {

BasicBlock::iterator it(inst);

++it; *// After this line, it refers to the instruction after \*inst*

**if** (it != inst->getParent()->end()) errs() << \*it << "**\n**";}

不幸的是，这些隐式转换是有代价的；它们阻止这些迭代器符合标准迭代器约定，因此无法与标准算法和容器一起使用。例如，它们会阻止编译以下代码，其中Bis a BasicBlock：

llvm::SmallVector<llvm::Instruction \*, 16>(B->begin(), B->end());

因此，这些隐式转换可能有一天会被删除，并 operator\*更改为返回指针而不是引用。

#### [查找呼叫站点：一个稍微复杂的示例](https://llvm.org/docs/ProgrammersManual.html" \l "id101)

假设您正在编写一个 FunctionPass 并且想要计算整个模块中的所有位置（即跨越每个Function），其中某个函数（即 some ）已经在范围内。正如您稍后将了解到的，您可能希望使用以更直接的方式完成此操作，但此示例将使我们能够探索如果您身边没有，您将如何做到这一点。在伪代码中，这就是我们想要做的：Function \*InstVisitorInstVisitor

initialize callCounter to zero

for each Function f in the Module

for each BasicBlock b in f

for each Instruction i in b

if (i a Call and calls the given function)

increment callCounter

实际代码是（记住，因为我们正在编写一个FunctionPass，所以我们的 FunctionPass派生类只需重写该runOnFunction 方法）：

Function\* targetFunc = ...;

**class** **OurFunctionPass** : **public** FunctionPass {

**public**:

OurFunctionPass(): callCounter(0) { }

**virtual** runOnFunction(Function& F) {

**for** (BasicBlock &**B** : F) {

**for** (Instruction &**I**: B) {

**if** (**auto** \*CB = dyn\_cast<CallBase>(&I)) {

*// We know we've encountered some kind of call instruction (call,*

*// invoke, or callbr), so we need to determine if it's a call to*

*// the function pointed to by m\_func or not.*

**if** (CB->getCalledFunction() == targetFunc)

++callCounter;

}

}

}

}

**private**:

unsigned callCounter;};

#### [遍历 def-use 和 use-def 链](https://llvm.org/docs/ProgrammersManual.html" \l "id102)

通常，我们可能有一个Value类 ( [doxygen](https://llvm.org/doxygen/classllvm_1_1Value.html) ) 的实例，我们想确定哪些Users 使用Value. User特定 的所有 s 的列表Value称为def-use链。例如，假设我们有 一个特定函数的Function\*名字。查找所有使用的指令就像遍历def-use链 一样简单：Ffoo fooF

Function \*F = ...;

**for** (User \***U** : F->users()) {

**if** (Instruction \*Inst = dyn\_cast<Instruction>(U)) {

errs() << "F is used in instruction:**\n**";

errs() << \*Inst << "**\n**";

}

或者，通常有一个UserClass ( [doxygen](https://llvm.org/doxygen/classllvm_1_1User.html) ) 的实例并且需要知道 Value它使用了什么。a 使用的所有Values的列表User称为use-def链。class 的实例Instruction是 common Users，因此我们可能希望迭代特定指令使用的所有值（即特定 的操作数Instruction）：

Instruction \*pi = ...;

**for** (Use &**U** : pi->operands()) {

Value \*v = U.get();

*// ...*}

将对象声明为const是执行无突变算法（例如分析等）的重要工具。为此目的，上面的迭代器以不变的形式出现，如Value::const\_use\_iteratorand Value::const\_op\_iterator。use/op\_begin()它们分别在调用s 或s时自动出现 。取消引用后，他们返回s。否则上述模式保持不变。const Value\*const User\*const Use\*

#### [迭代块的前驱和后继](https://llvm.org/docs/ProgrammersManual.html" \l "id103)

使用定义在"llvm/IR/CFG.h". 只需使用这样的代码来遍历 BB 的所有前身：

#include *"llvm/IR/CFG.h"*BasicBlock \*BB = ...;

**for** (BasicBlock \***Pred** : predecessors(BB)) {

*// ...*}

同样，要迭代后继者，请使用successors.

### [做简单的改变](https://llvm.org/docs/ProgrammersManual.html" \l "id104)

LLVM 基础设施中存在一些值得了解的原始转换操作。在执行转换时，操作基本块的内容是相当常见的。本节描述了一些常用的方法并给出了示例代码。

#### [创建和插入新](https://llvm.org/docs/ProgrammersManual.html" \l "id105)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id105)**[的 s](https://llvm.org/docs/ProgrammersManual.html" \l "id105)

实例化指令

s 的创建Instruction非常简单：只需调用构造函数来实例化并提供必要的参数。例如， anAllocaInst只需要一个 (const-ptr-to) Type。因此：

**auto** \*ai = **new** AllocaInst(Type::Int32Ty);

将创建一个AllocaInst实例，表示在运行时在当前堆栈帧中分配一个整数。每个Instruction子类都可能有不同的默认参数，这些参数会改变指令的语义，因此请参阅您有兴趣实例化[的指令子类的 doxygen 文档。](https://llvm.org/doxygen/classllvm_1_1Instruction.html)

命名值

在可能的情况下命名指令的值非常有用，因为这有助于调试转换。如果您最终查看生成的 LLVM 机器代码，您肯定希望拥有与指令结果相关联的逻辑名称！Name通过为构造函数的（默认）参数提供一个值 Instruction，您可以将逻辑名称与指令在运行时的执行结果相关联。例如，假设我正在编写一个为堆栈上的整数动态分配空间的转换，并且该整数将被其他代码用作某种索引。为此，我将 anAllocaInst放在第一个BasicBlocksome的第一个点Function，并且我打算在同一个Function. 我可能会这样做：

**auto** \*pa = **new** AllocaInst(Type::Int32Ty, 0, "indexLoc");

whereindexLoc现在是指令执行值的逻辑名称，它是指向运行时堆栈上整数的指针。

插入说明

基本上有三种方法可以将 anInstruction插入现有的指令序列中，形成 a BasicBlock：

插入到指令列表中BasicBlock

给定一个，其中一个，以及我们希望在之前插入的新创建的指令，我们执行以下操作：BasicBlock\* pbInstruction\* piBasicBlock\*pi

BasicBlock \*pb = ...;Instruction \*pi = ...;**auto** \*newInst = **new** Instruction(...);

newInst->insertBefore(pi); *// Inserts newInst before pi*

附加到 a 的末尾BasicBlock是如此常见，以至于Instruction 类和Instruction派生类提供构造函数，这些构造函数采用指向BasicBlock要附加到的 a 的指针。例如看起来像这样的代码：

BasicBlock \*pb = ...;**auto** \*newInst = **new** Instruction(...);

newInst->insertInto(pb, pb->end()); *// Appends newInst to pb*

变成：

BasicBlock \*pb = ...;**auto** \*newInst = **new** Instruction(..., pb);

这更干净，特别是如果您正在创建长指令流。

使用实例插入IRBuilder

使用以前的方法插入几个Instructions 可能会非常费力。是一个方便的IRBuilder类，可用于将多条指令添加到 a 的末尾BasicBlock或特定的 Instruction. 它还支持常量折叠和重命名命名寄存器（参见IRBuilder的模板参数）。

下面的示例演示了一个非常简单的使用IRBuilderwhere 指令之前插入三个指令的方法pi。前两条指令是 Call 指令，第三条指令将两次调用的返回值相乘。

Instruction \*pi = ...;IRBuilder<> Builder(pi);CallInst\* callOne = Builder.CreateCall(...);CallInst\* callTwo = Builder.CreateCall(...);Value\* result = Builder.CreateMul(callOne, callTwo);

下面的示例与上面的示例类似，不同之处在于创建的 IRBuilderinserts 指令位于BasicBlock pb.

BasicBlock \*pb = ...;IRBuilder<> Builder(pb);CallInst\* callOne = Builder.CreateCall(...);CallInst\* callTwo = Builder.CreateCall(...);Value\* result = Builder.CreateMul(callOne, callTwo);

有关. 的实际使用，请参阅[Kaleidoscope 教程](https://llvm.org/docs/tutorial/LangImpl03.html)IRBuilder。

#### [删除指令](https://llvm.org/docs/ProgrammersManual.html" \l "id106)

[从构成BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)的现有指令序列中删除指令 非常简单：只需调用指令的 eraseFromParent()方法即可。例如：

Instruction \*I = .. ;I->eraseFromParent();

这会取消指令与其包含的基本块的链接并将其删除。如果您只想取消指令与其包含的基本块的链接而不是删除它，则可以使用该removeFromParent()方法。

#### [用另一个值替换指令](https://llvm.org/docs/ProgrammersManual.html" \l "id107)

##### [替换单个指令](https://llvm.org/docs/ProgrammersManual.html" \l "id108)

包含“ [llvm/Transforms/Utils/BasicBlockUtils.h](https://llvm.org/doxygen/BasicBlockUtils_8h_source.html) ”允许使用两个非常有用的替换函数：ReplaceInstWithValue和 ReplaceInstWithInst。

##### [删除指令](https://llvm.org/docs/ProgrammersManual.html" \l "id109)

ReplaceInstWithValue

此函数用一个值替换给定指令的所有使用，然后删除原始指令。以下示例说明了AllocaInst用指向整数的空指针为单个整数分配内存的特定结果的替换。

AllocaInst\* instToReplace = ...;BasicBlock::iterator ii(instToReplace);

ReplaceInstWithValue(instToReplace->getParent(), ii,

Constant::getNullValue(PointerType::getUnqual(Type::Int32Ty)));

ReplaceInstWithInst

此函数用另一条指令替换特定指令，将新指令插入到旧指令所在位置的基本块中，并用新指令替换旧指令的任何使用。以下示例说明了一个替换为 AllocaInst另一个。

AllocaInst\* instToReplace = ...;BasicBlock::iterator ii(instToReplace);

ReplaceInstWithInst(instToReplace->getParent(), ii,

**new** AllocaInst(Type::Int32Ty, 0, "ptrToReplacedInt"));

##### [替换 Users 和 Values 的多次使用](https://llvm.org/docs/ProgrammersManual.html" \l "id110)

您一次可以使用Value::replaceAllUsesWith和User::replaceUsesOfWith更改多个用途。有关详细信息，请分别参阅 [Value Class](https://llvm.org/doxygen/classllvm_1_1Value.html)和[User Class的 doxygen 文档。](https://llvm.org/doxygen/classllvm_1_1User.html)

#### [删除全局变量](https://llvm.org/docs/ProgrammersManual.html" \l "id111)

从模块中删除全局变量与删除指令一样简单。首先，您必须有一个指向要删除的全局变量的指针。您使用此指针将其从其父模块（模块）中删除。例如：

GlobalVariable \*GV = .. ;

GV->eraseFromParent();

## [线程和 LLVM](https://llvm.org/docs/ProgrammersManual.html" \l "id112)

本节描述 LLVM API 与多线程的交互，包括客户端应用程序和托管应用程序中的 JIT。

请注意，LLVM 对多线程的支持还比较年轻。从 2.5 版开始，支持线程化托管应用程序的执行，但不支持线程化客户端对 API 的访问。虽然现在支持此用例，但客户端必须遵守下面指定的准则以确保在多线程模式下正常运行。

请注意，在类 Unix 平台上，LLVM 需要存在 GCC 的原子内在函数才能支持线程操作。如果您需要在没有合适的现代系统编译器的平台上使用支持多线程的 LLVM，请考虑在单线程模式下编译 LLVM 和 LLVM-GCC，并使用生成的编译器构建具有多线程支持的 LLVM 副本。

### **[llvm\_shutdown()](https://llvm.org/docs/ProgrammersManual.html" \l "id113)**[结束执行](https://llvm.org/docs/ProgrammersManual.html" \l "id113)

当您使用完 LLVM API 后，您应该调用llvm\_shutdown()以释放用于内部结构的内存。

### [惰性初始化](https://llvm.org/docs/ProgrammersManual.html" \l "id114)**[ManagedStatic](https://llvm.org/docs/ProgrammersManual.html" \l "id114)**

ManagedStatic是LLVM中的一个实用类，用于实现静态资源的静态初始化，比如全局类型表。在单线程环境中，它实现了一个简单的惰性初始化方案。然而，当 LLVM 编译为支持多线程时，它使用双重检查锁定来实现线程安全的惰性初始化。

### [实现隔离](https://llvm.org/docs/ProgrammersManual.html" \l "id115)**[LLVMContext](https://llvm.org/docs/ProgrammersManual.html" \l "id115)**

LLVMContext是 LLVM API 中的一个不透明类，客户端可以使用它在同一地址空间内同时操作多个独立的 LLVM 实例。例如，在假设的编译服务器中，单个翻译单元的编译在概念上独立于所有其他翻译单元，并且希望能够在独立的服务器线程上同时编译传入的翻译单元。幸运的是，LLVMContext 存在就是为了实现这种场景！

从概念上讲，LLVMContext提供隔离。LLVM 内存 IR 中的每个 LLVM 实体（Modules、Values、Types、Constants 等）都属于一个LLVMContext. 不同上下文中的实体 不能相互交互：Module不同上下文中的 s 不能链接在一起，Functions 不能添加到Module不同上下文中的 s 等。这意味着同时在多个线程上编译是安全的，只要没有两个线程对同一上下文中的实体进行操作。

LLVMContext实际上，除了Type创建/查找 API之外，API 中很少有地方需要明确指定 a 。因为 every 都 Type带有对其所属上下文的引用，所以大多数其他实体可以通过查看自己的 来确定它们属于哪个上下文Type。如果您要向 LLVM IR 添加新实体，请尽量保持此接口设计。

### [线程和 JIT](https://llvm.org/docs/ProgrammersManual.html" \l "id116)

LLVM 的“eager”JIT 编译器可以安全地用于线程程序。多个线程可以并发调用ExecutionEngine::getPointerToFunction()or ExecutionEngine::runFunction()，多个线程可以并发运行JIT输出的代码。用户仍然必须确保只有一个线程访问给定的 IR，LLVMContext而另一个线程可能正在修改它。一种方法是在 JIT 外部访问 IR 时始终持有 JIT 锁（JIT通过添加s修改IR）。CallbackVH另一种方法是仅从getPointerToFunction()的 LLVMContext线程调用。

当 JIT 配置为延迟编译（使用 ExecutionEngine::DisableLazyCompilation(false)）时，当前在函数延迟编译后更新调用站点时存在[竞争条件。](https://bugs.llvm.org/show_bug.cgi?id=5184)如果您确保一次只有一个线程可以调用任何特定的惰性存根并且 JIT 锁保护任何 IR 访问，那么仍然可以在线程程序中使用惰性 JIT，但我们建议在线程程序中仅使用热切 JIT。

## [高级主题](https://llvm.org/docs/ProgrammersManual.html" \l "id117)

本节介绍大多数客户不需要了解的一些高级或晦涩的 API。这些 API 倾向于管理 LLVM 系统的内部工作，并且只需要在异常情况下访问。

### **[ValueSymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "id118)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id118)

ValueSymbolTable( [doxygen](https://llvm.org/doxygen/classllvm_1_1ValueSymbolTable.html) ) 类提供了一个符号表，[函数](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)和[模块](https://llvm.org/docs/ProgrammersManual.html" \l "module)类使用该符号表来命名值定义。[符号表可以为任何Value](https://llvm.org/docs/ProgrammersManual.html" \l "value)提供名称。

请注意，SymbolTable大多数客户端不应直接访问该类。仅当需要对符号表名称本身进行迭代时才应使用它，这是非常特殊的用途。请注意，并非所有 LLVM [Value](https://llvm.org/docs/ProgrammersManual.html" \l "value)都有名称，没有名称的（即它们具有空名称）在符号表中不存在。

符号表支持迭代符号表中的值 with begin/end/iterator并支持查询以查看特定名称是否在符号表中（with lookup）。该类ValueSymbolTable不公开任何公共修改器方法，而是简单地调用setName一个值，该值会将其自动插入到适当的符号表中。

### **[User](https://llvm.org/docs/ProgrammersManual.html" \l "id119)**[和拥有的](https://llvm.org/docs/ProgrammersManual.html" \l "id119)**[Use](https://llvm.org/docs/ProgrammersManual.html" \l "id119)**[类的内存布局](https://llvm.org/docs/ProgrammersManual.html" \l "id119)

( [doxygen](https://llvm.org/doxygen/classllvm_1_1User.html)User )类提供了表达对其他 [Value 实例](https://llvm.org/doxygen/classllvm_1_1Value.html)的所有权的基础。( [doxygen](https://llvm.org/doxygen/classllvm_1_1Use.html) ) 辅助类用于记账并促进O(1)添加和删除。UserUse

#### **[User](https://llvm.org/docs/ProgrammersManual.html" \l "id120)**[和](https://llvm.org/docs/ProgrammersManual.html" \l "id120)**[Use](https://llvm.org/docs/ProgrammersManual.html" \l "id120)**[对象之间的交互和关系](https://llvm.org/docs/ProgrammersManual.html" \l "id120)

的子类User可以选择合并其Use对象或通过指针离线引用它们。混合变体（一些s inline 其他 hung off）是不切实际的，并且打破了属于同一对象Use 的不变量 形成一个连续的数组。UseUser

我们在User（子）类中有 2 种不同的布局：

布局一）

对象位于Use对象内部（分别位于固定偏移处），User 并且它们的数量是固定的。

布局 b)

Use对象由指向对象数组的指针引用， 并且User它们的数量可能是可变的。

从 v2.4 开始，每个布局仍然拥有一个指向Uses 数组开头的直接指针。虽然布局 a) 不是强制性的，但为了简单起见，我们坚持这种冗余。该User对象还存储Use它拥有的对象的数量。（理论上，这些信息也可以根据下面给出的方案计算得出。）

特殊形式的分配运算符 ( ) 强制执行以下内存布局：operator new

User布局 a) 是通过在对象前加上数组来建模的Use[] 。

...---.---.---.---.-------...

| P | P | P | P | User

'''---'---'---'---'-------'''

布局 b) 是通过指向Use[]数组来建模的。

.-------...

| User

'-------'''

|

v

.---.---.---.---...

| P | P | P | P |

'---'---'---'---'''

（上图中' P'代表 Use\*\* 存储在 Use 成员中每个对象中的 Use::Prev ）

### [设计类型层次结构和多态接口](https://llvm.org/docs/ProgrammersManual.html" \l "id121)

有两种不同的设计模式往往会导致在 C++ 程序的类型层次结构中对方法使用虚拟分派。第一个是真正的类型层次结构，层次结构中的不同类型对功能和语义的特定子集建模，并且这些类型严格相互嵌套。Value 在或Type类型层次结构中可以看到这方面的很好的例子。

第二个是跨多态接口实现集合动态分派的愿望。后一种用例可以通过定义一个抽象接口基类来使用虚拟分派和继承进行建模，所有实现都从该基类派生和覆盖。然而，这种实施策略强制存在实际上没有意义的**“is-a”关系。**通常没有一些有用的概括的嵌套层次结构，代码可能与之交互并上下移动。相反，有一个跨一系列实现分派的单一接口。

第二个用例的首选实现策略是泛型编程（有时称为“编译时鸭子类型”或“静态多态性”）。例如，可以跨符合接口或概念T的任何特定实现实例化某个类型参数的模板。这里的一个很好的例子是在有向图中对节点建模的任何类型的高度通用属性。LLVM 主要通过模板和通用编程对这些进行建模。此类模板包括 和。当这种类型的多态性确实需要**动态**分配时，您可以使用一种称为基于概念的多态性的技术来概括它LoopInfoBaseDominatorTreeBase. 此模式使用非常有限的虚拟分派形式模拟模板的接口和行为，以在其实现中进行类型擦除。您可以在系统中找到该技术的示例，PassManager.hSean Parent 在他的几篇演讲和论文中对此有更详细的介绍：

1. [Inheritance Is The Base Class of Evil](http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil) - 描述这种技术的 GoingNative 2013 演讲，可能是最好的起点。
2. [值语义和基于概念的多态性](http://www.youtube.com/watch?v=_BpMYeUFXv8)- C++Now！2012 年的演讲更详细地描述了这项技术。
3. [Sean Parent's Papers and Presentations](http://github.com/sean-parent/sean-parent.github.com/wiki/Papers-and-Presentations) - 一个 GitHub 项目，其中包含幻灯片、视频和有时代码的链接。

在决定创建类型层次结构（使用标记或虚拟分派）和使用模板或基于概念的多态性之间时，请考虑是否对抽象基类进行了一些改进，它是接口边界上具有语义意义的类型。如果比根抽象接口更精致的东西作为语义模型的部分扩展来讨论是没有意义的，那么你的用例可能更适合多态性，你应该避免使用虚拟分派。然而，可能有一些紧急情况需要使用一种或另一种技术。

如果您确实需要引入类型层次结构，我们更愿意使用带有手动标记分派和/或 RTTI 的显式封闭类型层次结构，而不是 C++ 代码中更常见的开放继承模型和虚拟分派。这是因为 LLVM 很少鼓励库消费者扩展其核心类型，并利用其层次结构的封闭和标记分派性质来生成更高效的代码。我们还发现，我们对类型层次结构的大量使用更适合基于标签的模式匹配，而不是跨公共接口的动态调度。在 LLVM 中，我们构建了自定义助手来促进这种设计。请参阅本文档关于[isa 和 dyn\_cast](https://llvm.org/docs/ProgrammersManual.html" \l "isa)的部分以及我们的[详细文档](https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html)其中描述了如何实现此模式以与 LLVM 助手一起使用。

### [ABI 破坏性检查](https://llvm.org/docs/ProgrammersManual.html" \l "id122)

更改 LLVM C++ ABI 的检查和断言基于预处理器符号LLVM\_ENABLE\_ABI\_BREAKING\_CHECKS – 使用LLVM\_ENABLE\_ABI\_BREAKING\_CHECKS构建的 LLVM 库不是在未定义它的情况下构建的 ABI 兼容 LLVM 库。默认情况下，打开断言也会打开LLVM\_ENABLE\_ABI\_BREAKING\_CHECKS ，因此默认的 +Asserts 构建与默认的 -Asserts 构建不兼容。希望 +Asserts 和 -Asserts 构建之间的 ABI 兼容性的客户端应该使用 CMake 构建系统来独立于LLVM\_ENABLE\_ASSERTIONS设置LLVM\_ENABLE\_ABI\_BREAKING\_CHECKS。

## [核心 LLVM 类层次结构参考](https://llvm.org/docs/ProgrammersManual.html" \l "id123)

#include "llvm/IR/Type.h"

标头来源：[Type.h](https://llvm.org/doxygen/Type_8h_source.html)

doxygen 信息：[类型类](https://llvm.org/doxygen/classllvm_1_1Type.html)

核心 LLVM 类是表示正在检查或转换的程序的主要方式。核心 LLVM 类在目录中的头文件中定义include/llvm/IR，并在lib/IR 目录中实现。值得注意的是，由于历史原因，这个库的名字libLLVMCore.so并不是libLLVMIR.so大家想象的那样。

### [类型类和派生类型](https://llvm.org/docs/ProgrammersManual.html" \l "id124)

Type是所有类型类的超类。每个Value都有一个Type。 Type不能直接实例化，只能通过其子类实例化。某些原始类型（VoidType、LabelType和）具有隐藏的子类FloatType。 DoubleType它们是隐藏的，因为Type除了将它们自己与Type.

所有其他类型都是DerivedType. 可以命名类型，但这不是必需的。在任何时候都只存在一个给定形状的实例。这允许使用类型实例的地址相等来执行类型相等。也就是说，给定两个Type\*值，如果指针相同，则类型相同。

#### [重要的公共方法](https://llvm.org/docs/ProgrammersManual.html" \l "id125)

* bool isIntegerTy() const：对任何整数类型返回 true。
* bool isFloatingPointTy()：如果这是五种浮点类型之一，则返回真。
* bool isSized()：如果类型的大小已知，则返回 true。没有大小的东西是抽象类型、标签和 void。

#### [重要派生类型](https://llvm.org/docs/ProgrammersManual.html" \l "id126)

IntegerType

DerivedType 的子类，表示任意位宽的整数类型。可以表示IntegerType::MIN\_INT\_BITS(1) 和 (~800 万)之间的任何位宽。IntegerType::MAX\_INT\_BITS

* static const IntegerType\* get(unsigned NumBits)：获取特定位宽的整数类型。
* unsigned getBitWidth() const：获取整数类型的位宽。

SequentialType

这是 ArrayType 和 VectorType 的子类。

* const Type \* getElementType() const：返回顺序类型中每个元素的类型。
* uint64\_t getNumElements() const：返回顺序类型的元素个数。

ArrayType

这是 SequentialType 的子类，定义了数组类型的接口。

PointerType

指针类型的 Type 的子类。

VectorType

向量类型的 SequentialType 的子类。矢量类型类似于 ArrayType，但有所区别，因为它是第一类类型，而 ArrayType 不是。向量类型用于向量运算，通常是整数或浮点数类型的小向量。

StructType

结构类型的 DerivedTypes 的子类。

FunctionType

函数类型的 DerivedTypes 的子类。

* bool isVarArg() const：如果它是可变参数函数，则返回 true。
* const Type \* getReturnType() const：返回函数的返回类型。
* const Type \* getParamType (unsigned i)：返回第i个参数的类型。
* const unsigned getNumParams() const：返回形式参数的数量。

### **[Module](https://llvm.org/docs/ProgrammersManual.html" \l "id127)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id127)

#include "llvm/IR/Module.h"

标头来源：[Module.h](https://llvm.org/doxygen/Module_8h_source.html)

doxygen 信息：[模块类](https://llvm.org/doxygen/classllvm_1_1Module.html)

该类Module代表 LLVM 程序中存在的顶级结构。LLVM 模块实际上是原始程序的翻译单元，或者是链接器合并的几个翻译单元的组合。该类Module跟踪[Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)列表、[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "globalvariable)列表和[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)。此外，它还包含一些有用的成员函数，旨在简化常见操作。

#### **[Module](https://llvm.org/docs/ProgrammersManual.html" \l "id128)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id128)

Module::Module(std::string name = "")

构建[模块](https://llvm.org/docs/ProgrammersManual.html" \l "module)很容易。您可以选择为其提供名称（可能基于翻译单元的名称）。

Module::iterator- 函数列表迭代器的 Typedef

Module::const\_iterator- const\_iterator 的类型定义。

begin(), end(), size(),empty()

这些转发方法使访问 Module对象的[函数](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)列表的内容变得容易。

Module::FunctionListType &getFunctionList()

返回[Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)的列表。当您需要更新列表或执行没有转发方法的复杂操作时，这是必要的。

Module::global\_iterator- 全局变量列表迭代器的 Typedef

Module::const\_global\_iterator- const\_iterator 的类型定义。

global\_begin(), global\_end(), global\_size(),global\_empty()

这些转发方法使访问 Module对象的[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "globalvariable)列表的内容变得容易。

Module::GlobalListType &getGlobalList()

返回[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "globalvariable)的列表。当您需要更新列表或执行没有转发方法的复杂操作时，这是必要的。

SymbolTable \*getSymbolTable()

为此返回对[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)Module的引用。

Function \*getFunction(StringRef Name) const

Module [在SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)中查找指定的函数。如果不存在，则返回null。

FunctionCallee getOrInsertFunction(const std::string &Name, const FunctionType \*T)

Module [在SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)中查找指定的函数。如果不存在，则为该函数添加一个外部声明并返回。请注意，已存在的函数签名可能与请求的签名不匹配。因此，为了实现将结果直接传递给 EmitCall 的常见用法，返回类型是一个 struct of ，而不是简单的带有潜在意外签名的 the 。{FunctionType \*T, Constant \*FunctionPtr}Function\*

std::string getTypeName(const Type \*Ty)

如果指定[Type在](https://llvm.org/docs/ProgrammersManual.html" \l "type)[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)中至少有一个条目，则返回它。否则返回空字符串。

bool addTypeName(const std::string &Name, const Type \*Ty)

在映射到的[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)中插入一个条目。如果已经存在该名称的条目，则返回 true 并且不修改[SymbolTable 。](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)NameTy

### **[Value](https://llvm.org/docs/ProgrammersManual.html" \l "id129)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id129)

#include "llvm/IR/Value.h"

标头来源：[Value.h](https://llvm.org/doxygen/Value_8h_source.html)

doxygen 信息：[值类](https://llvm.org/doxygen/classllvm_1_1Value.html)

该类Value是 LLVM 源代码库中最重要的类。它代表一个类型化的值，可以（除其他外）用作指令的操作数。有许多不同类型的Values，例如 [Constant](https://llvm.org/docs/ProgrammersManual.html" \l "constant) s、[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument) s。甚至[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "instruction)和[Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)也是Values。

在一个程序的 LLVM 表示中，一个特定的Value可能会被多次使用。例如，函数的传入参数（用[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument)类的实例表示）被函数中引用该参数的每条指令“使用”。为了跟踪这种关系，Value 该类保留了所有User正在使用它的 s 的列表（[User](https://llvm.org/docs/ProgrammersManual.html" \l "user)类是 LLVM 图中所有可以引用Values 的节点的基类）。这个使用列表是 LLVM 在程序中表示 def-use 信息的方式，并且可以通过use\_\*如下所示的方法访问。

因为 LLVM 是类型化表示，所以每个 LLVM都是类型化的，并且可以通过方法获得Value此 [Type](https://llvm.org/docs/ProgrammersManual.html" \l "type)getType()。此外，所有 LLVM 值都可以命名。的“名称”Value是打印在 LLVM 代码中的符号字符串：

%foo = **add** i32 1, 2

该指令的名称是“foo”。**请注意**，任何值的名称都可能丢失（空字符串），因此名称**只能**用于调试（使源代码更易于阅读，调试打印输出），它们不应用于跟踪值或之间的映射他们。为此，请改用 std::map指向Value自身的指针。

LLVM 的一个重要方面是 SSA 变量和生成它的操作之间没有区别。正因为如此，任何对指令产生的值（或作为传入参数可用的值）的引用都表示为指向表示该值的类实例的直接指针。虽然这可能需要一些时间来适应，但它简化了表示并使其更易于操作。

#### **[Value](https://llvm.org/docs/ProgrammersManual.html" \l "id130)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id130)

Value::use\_iterator- 使用列表上的迭代器的 Typedef

Value::const\_use\_iterator- typedef for const\_iterator over the use-list

unsigned use\_size()- 返回值的用户数。

bool use\_empty()- 如果没有用户则返回真。

use\_iterator use\_begin()- 获取指向使用列表开头的迭代器。

use\_iterator use\_end()- 获取一个迭代器到使用列表的末尾。

User \*use\_back()- 返回列表中的最后一个元素。

这些方法是访问 LLVM 中的 def-use 信息的接口。与 LLVM 中的所有其他迭代器一样，命名约定遵循[STL](https://llvm.org/docs/ProgrammersManual.html" \l "stl)定义的约定。

Type \*getType() const 此方法返回值的类型。

bool hasName() const

std::string getName() const

void setName(const std::string &Name)

该系列方法用于访问 a 并为其分配名称Value，请[注意上述预防措施](https://llvm.org/docs/ProgrammersManual.html" \l "namewarning)。

void replaceAllUsesWith(Value \*V)

该方法遍历一个Value改变当前值的所有[User](https://llvm.org/docs/ProgrammersManual.html" \l "user)的使用列表，改为引用“ V”。例如，如果您检测到一条指令总是产生一个常量值（例如通过常量折叠），您可以用常量替换指令的所有使用，如下所示：

Inst->replaceAllUsesWith(ConstVal);

### **[User](https://llvm.org/docs/ProgrammersManual.html" \l "id131)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id131)

#include "llvm/IR/User.h"

标头来源：[User.h](https://llvm.org/doxygen/User_8h_source.html)

doxygen 信息：[用户类](https://llvm.org/doxygen/classllvm_1_1User.html)

超类：[值](https://llvm.org/docs/ProgrammersManual.html" \l "value)

该类User是可能引用 Values 的所有 LLVM 节点的公共基类。它公开了一个“操作数”列表，Value其中包含用户所指的所有 s。该类User本身是 的子类 Value。

一个点的操作数直接指向它引用User的 LLVM 。Value因为LLVM使用Static Single Assignment (SSA)形式，所以只能引用一个定义，允许这种直接连接。此连接提供 LLVM 中的 use-def 信息。

#### **[User](https://llvm.org/docs/ProgrammersManual.html" \l "id132)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id132)

该类User以两种方式公开操作数列表：通过索引访问接口和通过基于迭代器的接口。

Value \*getOperand(unsigned i)

unsigned getNumOperands()

这两个方法User以方便的形式公开了操作数以供直接访问。

User::op\_iterator- 操作数列表上迭代器的 Typedef

op\_iterator op\_begin()- 获取指向操作数列表开头的迭代器。

op\_iterator op\_end()- 获取一个指向操作数列表末尾的迭代器。

这些方法共同构成了一个基于迭代器的操作数接口User。

### **[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id133)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id133)

#include "llvm/IR/Instruction.h"

标头来源：[Instruction.h](https://llvm.org/doxygen/Instruction_8h_source.html)

doxygen 信息：[指令类](https://llvm.org/doxygen/classllvm_1_1Instruction.html)

超类：[用户](https://llvm.org/docs/ProgrammersManual.html" \l "user)，[价值](https://llvm.org/docs/ProgrammersManual.html" \l "value)

该类Instruction是所有 LLVM 指令的公共基类。它只提供了几个方法，但却是一个非常常用的类。Instruction类本身跟踪的主要数据是操作码（指令类型）和嵌入其中的父[BasicBlock 。](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)Instruction为了表示特定类型的指令，使用了 的许多子类之一 Instruction。

因为Instruction该类是[User](https://llvm.org/docs/ProgrammersManual.html" \l "user)类的子类，所以它的操作数可以用与其他Users 相同的方式访问（使用 getOperand()/getNumOperands()和op\_begin()/op\_end()方法）。Instruction该类的一个重要文件是llvm/Instruction.def 文件。该文件包含一些关于 LLVM 中各种不同类型指令的元数据。它描述了用作操作码的枚举值（例如Instruction::Add和Instruction::ICmp），以及Instruction实现指令的具体子类（例如[BinaryOperator](https://llvm.org/docs/ProgrammersManual.html" \l "binaryoperator)和[CmpInst](https://llvm.org/docs/ProgrammersManual.html" \l "cmpinst)）。不幸的是，在这个文件中使用宏会混淆 doxygen，因此这些枚举值不会在 [doxygen 输出](https://llvm.org/doxygen/classllvm_1_1Instruction.html)中正确显示。

#### [该类的重要子](https://llvm.org/docs/ProgrammersManual.html" \l "id134)**[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id134)**[类](https://llvm.org/docs/ProgrammersManual.html" \l "id134)

BinaryOperator

这个子类代表所有两个操作数指令，它们的操作数必须是同一类型，比较指令除外。

* CastInst 这个子类是 12 个铸造指令的父类。它提供了强制转换指令的常见操作。

CmpInst

该子类表示两个比较指令， [ICmpInst](https://llvm.org/docs/LangRef.html" \l "i_icmp)（整数操作数）和 [FCmpInst](https://llvm.org/docs/LangRef.html" \l "i_fcmp)（浮点操作数）。

#### **[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "id135)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id135)

BasicBlock \*getParent()

返回嵌入其中 的[BasicBlock 。](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)Instruction

bool mayWriteToMemory()

如果指令写入内存，则返回 true，即它是 a call、 free、invoke或store。

unsigned getOpcode()

返回 的操作码Instruction。

Instruction \*clone() const

返回指定指令的另一个实例，除了指令没有父项（即它没有嵌入到[BasicBlock 中](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)）之外，在所有方面都与原始指令相同，并且它没有名称。

### [类](https://llvm.org/docs/ProgrammersManual.html" \l "id136)**[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "id136)**[和子类](https://llvm.org/docs/ProgrammersManual.html" \l "id136)

Constant 表示不同类型常量的基类。它的子类有ConstantInt、ConstantArray等，用于表示各种类型的常量。 [GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)也是一个子类，代表一个全局变量或函数的地址。

#### [常量的重要子类](https://llvm.org/docs/ProgrammersManual.html" \l "id137)

* ConstantInt ：Constant 的这个子类表示任意宽度的整数常量。
  + const APInt& getValue() const：返回此常量的基础值，一个 APInt 值。
  + int64\_t getSExtValue() const：通过符号扩展将底层 APInt 值转换为 int64\_t。如果 APInt 的值（不是位宽）太大而不适合 int64\_t，则会产生断言。因此，不鼓励使用此方法。
  + uint64\_t getZExtValue() const：通过零扩展将底层 APInt 值转换为 uint64\_t。如果 APInt 的值（不是位宽）太大而不适合 uint64\_t，则会产生断言。因此，不鼓励使用此方法。
  + static ConstantInt\* get(const APInt& Val): 返回表示由 提供的值的 ConstantInt 对象Val。该类型隐含为对应于 的位宽的 IntegerType Val。
  + static ConstantInt\* get(const Type \*Ty, uint64\_t Val)Val：返回代表整数类型提供的值的ConstantInt对象Ty。
* ConstantFP ：此类表示浮点常量。
  + double getValue() const：返回此常量的基础值。
* ConstantArray ：这表示一个常量数组。
  + const std::vector<Use> &getValues() const：返回构成此数组的组件常量向量。
* ConstantStruct ：这表示常量结构。
  + const std::vector<Use> &getValues() const：返回构成此数组的组件常量向量。
* GlobalValue ：这表示全局变量或函数。在任何一种情况下，该值都是一个常量固定地址（链接后）。

### **[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "id138)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id138)

#include "llvm/IR/GlobalValue.h"

标头来源：[GlobalValue.h](https://llvm.org/doxygen/GlobalValue_8h_source.html)

doxygen 信息：[GlobalValue 类](https://llvm.org/doxygen/classllvm_1_1GlobalValue.html)

超类：[常量](https://llvm.org/docs/ProgrammersManual.html" \l "constant)、[用户](https://llvm.org/docs/ProgrammersManual.html" \l "user)、[值](https://llvm.org/docs/ProgrammersManual.html" \l "value)

全局值（[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "globalvariable)或[Function ）是唯一在所有Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)的主体中可见的 LLVM 值。因为它们在全局范围内可见，所以它们也受制于与不同翻译单元中定义的其他全局变量的链接。要控制链接过程，请GlobalValue了解它们的链接规则。具体来说，GlobalValues 知道它们是否具有LinkageTypes枚举所定义的内部或外部链接。

如果 aGlobalValue有内部链接（相当于static在 C 中），则它对当前翻译单元之外的代码不可见，并且不参与链接。如果它有外部链接，它对外部代码可见，并参与链接。除了链接信息之外， GlobalValues 还跟踪它们当前属于哪个[模块。](https://llvm.org/docs/ProgrammersManual.html" \l "module)

因为GlobalValues 是内存对象，所以它们总是由它们的**地址**引用。因此，全局[类型](https://llvm.org/docs/ProgrammersManual.html" \l "type)始终是指向其内容的指针。在使用该指令时记住这一点很重要，GetElementPtrInst 因为必须首先取消引用该指针。例如，如果你有一个GlobalVariable（它的一个子类GlobalValue)是一个 24 整数类型的数组，那么 the是一个指向那个数组的指针。虽然这个数组的第一个元素的地址和 the 的值是相同的，但是它们有不同的类型。 ' 的类型是。第一个元素的类型是 因此，访问全局值需要您取消引用指针[24 x i32]GlobalVariableGlobalVariableGlobalVariable[24 x i32]i32.GetElementPtrInst首先，然后可以访问它的元素。这在[LLVM Language Reference Manual](https://llvm.org/docs/LangRef.html" \l "globalvars)中有解释。

#### **[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "id139)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id139)

bool hasInternalLinkage() const

bool hasExternalLinkage() const

void setInternalLinkage(bool HasInternalLinkage)

这些方法操纵GlobalValue.

Module \*getParent()

这将返回GlobalValue 当前嵌入的[模块。](https://llvm.org/docs/ProgrammersManual.html" \l "module)

### **[Function](https://llvm.org/docs/ProgrammersManual.html" \l "id140)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id140)

#include "llvm/IR/Function.h"

标头来源：[Function.h](https://llvm.org/doxygen/Function_8h_source.html)

doxygen 信息：[函数类](https://llvm.org/doxygen/classllvm_1_1Function.html)

超类：[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)、[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "constant)、[User](https://llvm.org/docs/ProgrammersManual.html" \l "user)、[Value](https://llvm.org/docs/ProgrammersManual.html" \l "value)

该类Function表示 LLVM 中的单个过程。它实际上是 LLVM 层次结构中更复杂的类之一，因为它必须跟踪大量数据。该类Function跟踪 [BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)列表、正式[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument)列表和[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)。

[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)的列表是对象中最常用的部分Function 。该列表强加了函数中块的隐式排序，这表明代码将如何由后端进行布局。此外，第一个[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)是Function. 在 LLVM 中明确分支到这个初始块是不合法的。没有隐式出口节点，事实上，单个 Function. 如果[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)列表为空，这表明这 Function实际上是一个函数声明：函数的实际主体尚未链接。

除了[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)的列表之外，Function该类还跟踪函数接收的正式[Argument的列表。](https://llvm.org/docs/ProgrammersManual.html" \l "argument)该容器管理[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument)节点的生命周期，就像[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)列表对[BasicBlock 所做](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)的一样。

[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)是一个很少使用的 LLVM 特性，它仅在您必须按名称查找值时使用。除此之外，内部使用[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)来确保 函数体中[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "instruction) s、[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock) s 或[Argument s 的名称之间没有冲突。](https://llvm.org/docs/ProgrammersManual.html" \l "argument)

请注意，这Function是一个[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)，因此也是一个[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "constant)。函数的值是它的地址（链接后），它保证是常量。

#### [重要的公众成员](https://llvm.org/docs/ProgrammersManual.html" \l "id141)**[Function](https://llvm.org/docs/ProgrammersManual.html" \l "id141)**

Function(const FunctionType \*Ty, LinkageTypes Linkage, const std::string &N = "", Module\* Parent = 0)

需要新建Functions来添加程序时使用的构造函数。构造函数必须指定要创建的函数的类型以及函数应具有的链接类型。[FunctionType](https://llvm.org/docs/ProgrammersManual.html" \l "functiontype)参数指定函数的形式参数和返回值。相同的 [FunctionType](https://llvm.org/docs/ProgrammersManual.html" \l "functiontype)值可用于创建多个函数。Parent 参数指定定义函数的模块。如果提供此参数，该函数将自动插入到该模块的函数列表中。

bool isDeclaration()

返回 是否Function定义了正文。如果函数是“外部的”，则它没有主体，因此必须通过链接到在不同翻译单元中定义的函数来解析。

Function::iterator- 基本块列表迭代器的 Typedef

Function::const\_iterator- const\_iterator 的类型定义。

begin(),,,,,,, end()\_ size()\_ empty()\_ insert()\_ splice()\_erase()

这些转发方法使访问 Function对象的[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "basicblock)列表的内容变得容易。

Function::arg\_iterator- 参数列表迭代器的 Typedef

Function::const\_arg\_iterator- const\_iterator 的类型定义。

arg\_begin(), arg\_end(), arg\_size(),arg\_empty()

这些是转发方法，可以轻松访问 Function对象的[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument)列表的内容。

Function::ArgumentListType &getArgumentList()

返回[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "argument)列表。当您需要更新列表或执行没有转发方法的复杂操作时，这是必要的。

BasicBlock &getEntryBlock()

返回BasicBlock函数的入口。因为函数的入口块始终是第一个块，所以这将返回Function.

Type \*getReturnType()

FunctionType \*getFunctionType()

这个遍历了函数的[Type](https://llvm.org/docs/ProgrammersManual.html" \l "type)，Function返回了函数的返回类型，也就是实际函数的[FunctionType](https://llvm.org/docs/ProgrammersManual.html" \l "functiontype)。

SymbolTable \*getSymbolTable()

返回指向this的[SymbolTable](https://llvm.org/docs/ProgrammersManual.html" \l "symboltable)Function的指针。

### **[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "id142)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id142)

#include "llvm/IR/GlobalVariable.h"

标头来源：[GlobalVariable.h](https://llvm.org/doxygen/GlobalVariable_8h_source.html)

doxygen 信息：[GlobalVariable 类](https://llvm.org/doxygen/classllvm_1_1GlobalVariable.html)

超类：[GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)、[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "constant)、[User](https://llvm.org/docs/ProgrammersManual.html" \l "user)、[Value](https://llvm.org/docs/ProgrammersManual.html" \l "value)

全局变量用 (surprise surprise)GlobalVariable 类表示。与函数一样，GlobalVariables 也是 [GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)的子类，因此始终由它们的地址引用（全局值必须存在于内存中，因此它们的“名称”指的是它们的常量地址）。有关更多信息，请参见 [GlobalValue](https://llvm.org/docs/ProgrammersManual.html" \l "globalvalue)。全局变量可能有一个初始值（必须是一个[Constant](https://llvm.org/docs/ProgrammersManual.html" \l "constant)），如果它们有一个初始化器，它们本身可能被标记为“常量”（表明它们的内容在运行时永远不会改变）。

#### **[GlobalVariable](https://llvm.org/docs/ProgrammersManual.html" \l "id143)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id143)

GlobalVariable(const Type \*Ty, bool isConstant, LinkageTypes &Linkage, Constant \*Initializer = 0, const std::string &Name = "", Module\* Parent = 0)

创建指定类型的新全局变量。如果isConstant为真，则全局变量将被标记为程序不变。Linkage 参数指定变量的链接类型（内部、外部、弱、linkonce、附加）。如果链接是 InternalLinkage、WeakAnyLinkage、WeakODRLinkage、LinkOnceAnyLinkage 或 LinkOnceODRLinkage，则生成的全局变量将具有内部链接。AppendingLinkage 将变量的所有实例（在不同的翻译单元中）连接在一起成为一个变量，但仅适用于数组。有关链接类型的更多详细信息，请参阅 [LLVM 语言参考](https://llvm.org/docs/LangRef.html" \l "modulestructure)。可选地，也可以为全局变量指定初始值设定项、名称和将变量放入其中的模块。

bool isConstant() const

如果这是已知不会在运行时修改的全局变量，则返回 true。

bool hasInitializer()

如果它GlobalVariable有一个初始值设定项，则返回 true。

Constant \*getInitializer()

返回 a 的初始值GlobalVariable。如果没有初始化器，调用此方法是不合法的。

### **[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "id144)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id144)

#include "llvm/IR/BasicBlock.h"

标头来源：[BasicBlock.h](https://llvm.org/doxygen/BasicBlock_8h_source.html)

doxygen 信息：[BasicBlock 类](https://llvm.org/doxygen/classllvm_1_1BasicBlock.html)

超类：[值](https://llvm.org/docs/ProgrammersManual.html" \l "value)

此类表示代码的单入口单出口部分，通常被编译器社区称为基本块。该类BasicBlock维护一个[Instruction](https://llvm.org/docs/ProgrammersManual.html" \l "instruction)列表，它们构成了块的主体。与语言定义相匹配，此指令列表的最后一个元素始终是终止符指令。

除了跟踪构成块的指令列表之外， BasicBlock该类还跟踪它嵌入的[函数。](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)

请注意，BasicBlocks 本身是[Value](https://llvm.org/docs/ProgrammersManual.html" \l "value) s，因为它们被分支之类的指令引用，并且可以进入开关表。 BasicBlock有类型label。

#### **[BasicBlock](https://llvm.org/docs/ProgrammersManual.html" \l "id145)**[类的重要公共成员](https://llvm.org/docs/ProgrammersManual.html" \l "id145)

BasicBlock(const std::string &Name = "", Function \*Parent = 0)

构造BasicBlock函数用于创建新的基本块以插入到函数中。构造函数可选择为新块取一个名称，以及一个将其插入的[函数。](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)如果 Parent指定参数，则自动将新的插入到指定的[Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)BasicBlock的末尾，如果未指定，则必须手动将 BasicBlock 插入到[Function](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)中。

BasicBlock::iterator- 指令列表迭代器的 Typedef

BasicBlock::const\_iterator- const\_iterator 的类型定义。

begin(), end(), front(), back(), size(), empty(),splice() 用于访问指令列表的 STL 样式函数。

这些方法和 typedef 是转发函数，它们与同名的标准库方法具有相同的语义。这些方法以易于操作的方式公开基本块的底层指令列表。

Function \*getParent()

返回一个指向嵌入块的[函数的指针，如果它是无家可归的，则返回一个空指针。](https://llvm.org/docs/ProgrammersManual.html" \l "c-function)

Instruction \*getTerminator()

返回指向出现在 BasicBlock. 如果没有终止符指令，或者块中的最后一条指令不是终止符，则返回空指针。

### **[Argument](https://llvm.org/docs/ProgrammersManual.html" \l "id146)**[班级\_](https://llvm.org/docs/ProgrammersManual.html" \l "id146)

Value 的这个子类定义了函数传入形式参数的接口。函数维护其形式参数的列表。一个参数有一个指向父函数的指针。