LLVM中的pass及其管理机制

LLVM编译器框架的核心概念是任务调度和执行。编译器开发者将IR分解为不同的处理对象，并将其处理过程实现为单独的pass类型。在编译器初始化时，pass被实例化，并被添加到pass管理器中。pass管理器以流水线的方式将各个独立的pass衔接起来，然后以预定义顺序遍历每个pass，根据pass实例返回值启动、停止或重复运行不同pass。因此，LLVM pass管理机制的主要模块包括pass、pass管理器、pass注册及相关模块，如PassRegistry、AnalysisUsage、AnalysisResolver等。

1、LLVM Pass及常用子类
pass是一种编译器开发的结构化技术，用于完成编译对象（如IR）的转换、分析或优化等功能。pass的执行就是编译器对编译对象进行转换、分析和优化的过程，pass构建了这些过程所需要的分析结果。
LLVM Pass是LLVM系统的重要组成部分，定义在llvm\include\llvm\Pass.h中：

```
class Pass {
  AnalysisResolver *Resolver = nullptr;  // Used to resolve analysis
  const void *PassID;
  PassKind Kind;

public:
  explicit Pass(PassKind K, char &pid) : PassID(&pid), Kind(K) {}
  Pass(const Pass &) = delete;
  Pass &operator=(const Pass &) = delete;
  virtual ~Pass();
……
}
```

LLVM提供的pass分为三类:Analysis pass、Transform pass和Utility pass。Analysis pass计算相关IR单元的高层信息，但不对其进行修改这些信息可以被其他pass使用，或用于调试和程序可视化。简言之，Analysis pass提供其它pass需要查询的信息并提供查询接口。例如，basic-aa pass是基本别名分析（Basic Alias Analysis）pass，得到的别名分析结果可以用于后续的其它优化pass。Analysis pass不仅从IR中得到有用信息，还可以通过调用其它Analysis pass得到信息，并将这些信息结合起来，得到关于IR更有价值的信息。这些分析结果可以被缓存下来，直到分析的IR被修改，原有的分析结果当然也就失效了。
Transform pass可以使用Analysis pass。Transform pass会检视IR，查询Analysis pass得到IR的高层信息，然后以某种方式改变和优化IR，并保证改变后的IR仍然合法有效。例如adce pass是激进的死代码消除（Aggressive Dead Code Elimination）pass，会将死代码从原来的模块中删除。
Utility pass是一些功能性的实用程序，既不属于Analysis pass，也不属于Transform pass。例如，extract-blocks pass将basic block从模块中提取出来供bugpoint使用，这个utility pass既不属于Analysis pass，也不属于Transform pass。参考文献[1]中列出了LLVM提供的所有pass。当调用RegisterPass()注册自定义pass时，会要求指定是否为Analysis pass。通过RegisterPass()注册自定义pass后，就可以使用LLVM opt工具对IR调用自定义pass功能。
LLVM Pass的基础模块是Pass类，这是所有pass的基类。自定义的pass类都要从预定义子类中继承，并根据自定义pass的具体功能要求覆写虚函数或增加新的功能函数。预定义子类包括ModulePass、CallGraphSCCPass、FunctionPass、LoopPass和RegionPass类等等。不同的子类有不同的约束条件，这些约束条件在调度pass时会用到。设计自定义pass时的首要任务就是确定自定义pass的基类。在为pass选择基类时，应在满足要求的前提下，尽可能选择最相关的类。这些类会为LLVM Pass基础结构提供优化运行所必需的信息，避免生成的编译器因为选择的基类不合适而导致运行速度变慢。各种pass组合在一起，完成各种IR优化任务。Pass之间的组合可以分为两类：一、多个pass作用于同一个IR单元，function pass是一个典型例子。如下图A所示，Function pass作用于一个function IR，但也可以在某个function pass中运行其它几个function pass，将这几个function pass组合起来作用于同一个IR单元，获得更好的优化效果。二、将一个IR单元分解为

更小的单元，并用相应类型的pass处理。如下图B所示，module pass作用于module，但也可以在某个module pass中运行function pass，作用于module中的每一个function，这就将一个IR单元分解为粒度更细的单元来处理。在编译器开发时，可以混合使用两种方式，将各种pass组合为流水线，对IR做不同处理和优化。

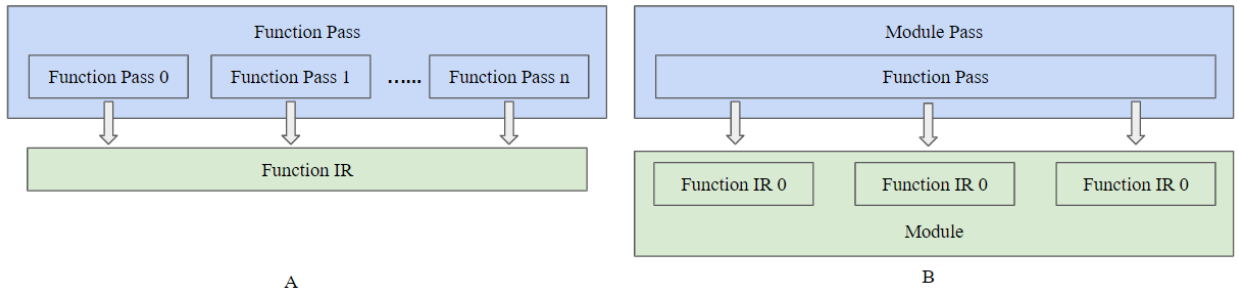

图 1. pass与IR单元的关系

LLVM Pass类及其子类如下图所示：

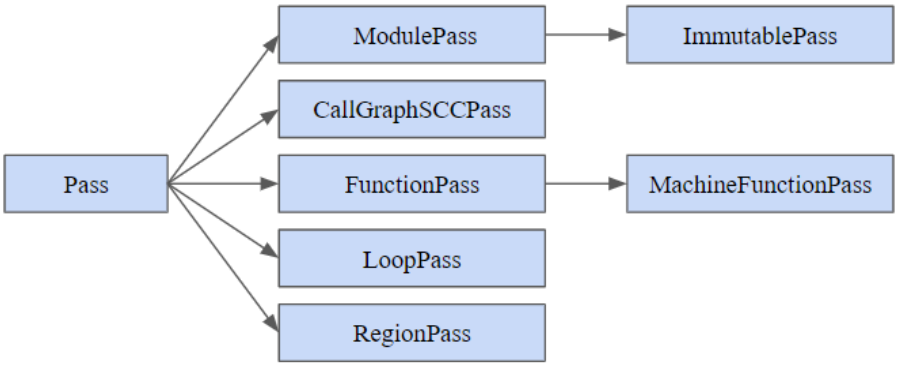

图 2. Pass类及其子类

1.1 ImmutablePass类

ImmutablePass类是ModulePass的派生类，类定义在llvm\include\llvm\Pass.h中：

```
class ImmutablePass : public ModulePass {
public:
  explicit ImmutablePass(char &pid) : ModulePass(pid) {}
  ~ImmutablePass() override;
  virtual void initializePass();
  ImmutablePass *getAsImmutablePass() override { return this; }

  bool runOnModule(Module &) override { return false; }
};
```

ImmutablePass类不是通常意义的转换或分析类型。当使用其它类型pass有复杂性等问题时，可以考虑使用ImmutablePass。ImmutablePass可以不和任何IR单元关联。ImmutablePass结果是不变的（immutable），也不会失效(invalidated)。这种pass的一个重要作用是提供当前目标机器的编译器配置信息，以及可能影响其它各种转换的信息。例如，AMDGPU后端中的AMDGPUAAWrapperPass（llvm\lib\Target\AMDGPU\AMDGPUAliasAnalysis.h)就是派生自ImmutablePass类，这是一个提供AMDGPUAAResult对象的包装pass，通过TBAA metadata为其它pass返回别名查询结果：

```
class AMDGPUAAWrapperPass : public ImmutablePass {
  std::unique_ptr<AMDGPUAAResult> Result;

public:
  static char ID;

  AMDGPUAAWrapperPass() : ImmutablePass(ID) {
    initializeAMDGPUAAWrapperPassPass(*PassRegistry::getPassRegistry());
  }

  AMDGPUAAResult &getResult() { return *Result; }
  const AMDGPUAAResult &getResult() const { return *Result; }

  bool doInitialization(Module &M) override {
    Result.reset(new AMDGPUAAResult(M.getDataLayout(),
        Triple(M.getTargetTriple())));
    return false;
  }

  bool doFinalization(Module &M) override {
    Result.reset();
    return false;
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override;
};
```

1.2 ModulePass类

ModulePass类是Pass的派生类，类定义在llvm\include\llvm\Pass.h中：

```
class ModulePass : public Pass {
public:
  explicit ModulePass(char &pid) : Pass(PT_Module, pid) {}
  ~ModulePass() override;
  Pass *createPrinterPass(raw_ostream &OS,
                          const std::string &Banner) const override;

  virtual bool runOnModule(Module &M) = 0;
  void assignPassManager(PMStack &PMS, PassManagerType T) override;

  PassManagerType getPotentialPassManagerType() const override;

protected:
  bool skipModule(Module &M) const;
};
```

ModulePass类用于实现非结构化的过程间优化和分析，几乎可以对程序执行任何操作。因此，ModulePass类可能是所有Pass类中最常用的类。由ModulePass派生的自定义pass将整个程序作为一个处理单元，可以对其中任何函数体做删改。由于ModulePass派生类的行为不可知，因此无法对其执行过程进行优化。例如，AMDGPU后端的AMDGPULowerIntrinsics就是派生自ModulePass类：

```
class AMDGPULowerIntrinsics : public ModulePass {
private:
  bool makeLIDRangeMetadata(Function &F) const;

public:
  static char ID;

  AMDGPULowerIntrinsics() : ModulePass(ID) {}

  bool runOnModule(Module &M) override;
  bool expandMemIntrinsicUses(Function &F);
  StringRef getPassName() const override {
    return "AMDGPU Lower Intrinsics";
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<TargetTransformInfoWrapperPass>();
  }
};
```

派生的ModulePass子类要覆写runOnModule()方法，派生类的大部分功能都在这个方法中实现，例如AMDGPULowerIntrinsics类中intrinsic实现。如果修改了模块，则runOnModule()方法应返回true，否则返回false。

```
bool AMDGPULowerIntrinsics::runOnModule(Module &M) {
  bool Changed = false;

  for (Function &F : M) {
  …...
    switch (F.getIntrinsicID()) {
    case Intrinsic::memcpy:
    …...
      if (expandMemIntrinsicUses(F))
        Changed = true;
      break;

    case Intrinsic::amdgcn_workitem_id_x:
    …...
      Changed |= makeLIDRangeMetadata(F);
      break;

    default:
      break;
    }
  }
```

```
  return Changed;
}
```

1.3 CallGraphSCCPass类

CallGraphSCCPass类是Pass的派生类，类定义在llvm/include/llvm/Analysis/CallGraphSCCPass.h中：

```
class CallGraphSCCPass : public Pass {
public:
  explicit CallGraphSCCPass(char &pid) : Pass(PT_CallGraphSCC, pid) {}
  Pass *createPrinterPass(raw_ostream &OS,
                          const std::string &Banner) const override;

  using llvm::Pass::doInitialization;
  using llvm::Pass::doFinalization;

  virtual bool doInitialization(CallGraph &CG) {
    return false;
  }

  virtual bool runOnSCC(CallGraphSCC &SCC) = 0;
  virtual bool doFinalization(CallGraph &CG) {
    return false;
  }

  void assignPassManager(PMStack &PMS, PassManagerType PMT) override;
  PassManagerType getPotentialPassManagerType() const override {
    return PMT_CallGraphPassManager;
  }

  void getAnalysisUsage(AnalysisUsage &Info) const override;

protected:
  bool skipSCC(CallGraphSCC &SCC) const;
};
```

CallGraphSCCPass用于在调用图（call graph）上从下至上的遍历程序。为了理解CallGraphSCCPass，首先要了解两个概念：调用图和SCC。

调用图表示程序方法间的关系，其中的节点表示程序方法，其中的边表示从调用方法和被调方法的调用关系。调用图中是否有环，取决于代码中是否存在直接或间接递归调用。如果程序中有递归调用，那么定向调用图中也就包含环。实际上，LLVM中没有模块表示调用图，pass通过分析call指令来计算调用图，从IR中的call指令推断出调用图。

SCC全称strongly connected component，即强连接分量。要理解SCC，需要从理解什么是连通。如果从图中的节点v到节点w之间存在一条路径，则称v和w是连通的。如果无向图中任意两个节点均连通，则该图称为连通图。对于非连通图，其中可能仍然有部分子图属于连通图，这种连通子图称为连通分量。连通分量的顶点数达到极大，意味着新增加任何顶点都可能使原连通分量不再连通。另外，连通分量的边数达到极大，意味着与子图中所有顶点相连的边都应包含在连通分量中。例如，下图A中的图是非连通图，但仍可分解为图B所示的两个

连通分量。图C不能被称为连通分量，因为其中缺少顶点B和C之间的边。图D更不能被称为连通分量，因为其中缺少顶点A及其关联的边。

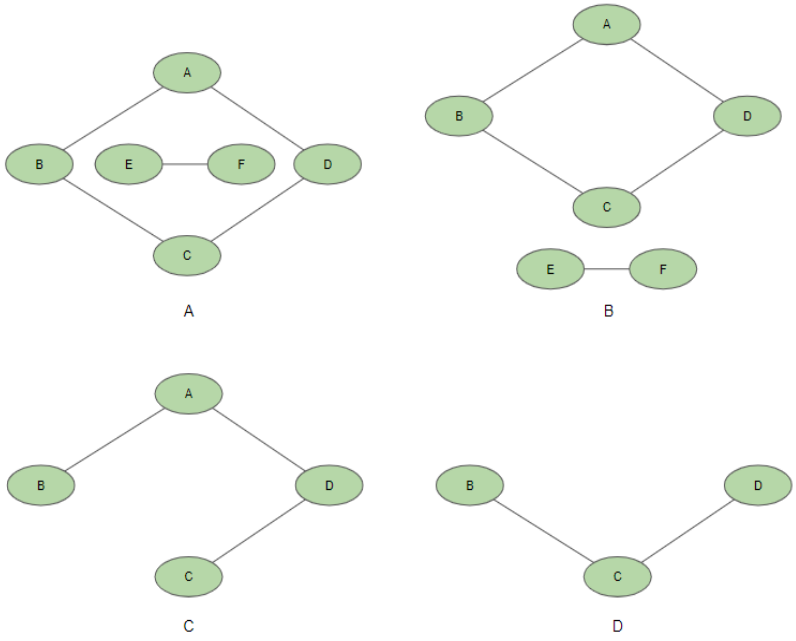

图 3. 连通图与连通分量

连通分量针对的是无向图，对于有向图则可分为强连通和弱连通两类。在有向图中，如果两个顶点间至少存在一条双向路径，则称两个顶点强连通（strongly connected）。如果有向图的任意两个顶点都强连通，称该图为强连通图。即，从图内任意一个顶点出发，存在通向图内任意一点的的一条路径。非强连通有向图的极大强连通子图，称为强连通分量。例如下图A是一个非强连通有向图，可以分解为图B所示的两个强连通分量。图B中的A、B、C组成一个强连通图，其中的任意两个顶点都存在一条双向路径。但如果加上顶点D，就不再是一个强连通图。顶点D是一个单独的强连通分量。将非强连通有向图分解为强连通分量的目的是由于强连通分量内部的节点性质相同，因此可以将一个强连通分量内的顶点简化为一个点，即消除了环。例如，图A中顶点A、B、C构成的强连通分量可简化为顶点E。如此一来，原图A就变成了图C所示的有向无环图(directed acyclic graph，DAG)。

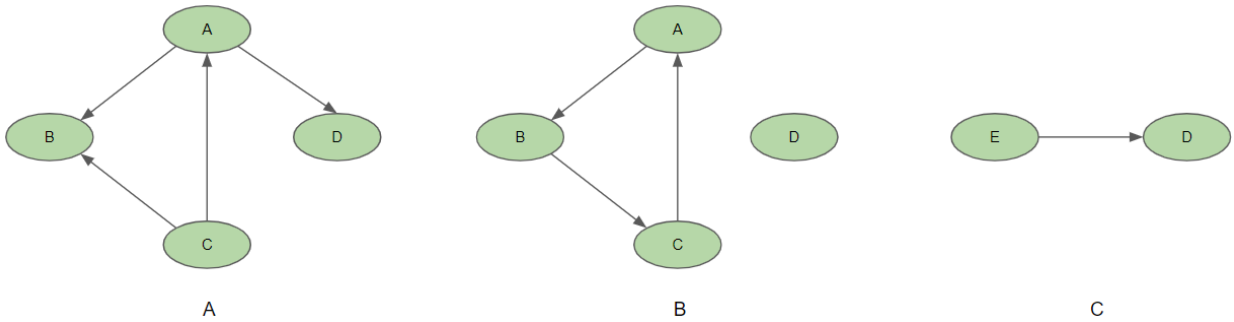

图 4. 强连通图

强连通分量调用图（SCC call graph）是调用图的强连通分量图，在LLVM中称为CallGraphSCC。强连通分量调用图的某个顶点可以仅包含一个方法，也可以包含调用图中的多个顶点。通过将调用图的有环子图分解为强连通分量，强连通分量调用图将变为无环图，也就是变为有向无环图。

CallGraphSCCPass构成了LLVM的过程间优化的重要组成部分。CallGraphSCCPass派生类在调用图的强连通分量上运行，并且提供了用于构建和遍历调用图的机制（即以后序方式遍历图），所以可以有效地对程序中的所有调用边进行成对的过程间优化，同时逐步细化并改善这些成对优化。
自定义SCC pass只能检查和修改当前SCC中的函数，以及直接调用当前SCC的函数和直接被当前SCC调用的函数。如下图所示，自定义SCC pass不能修改和检查顶点B中的函数，除此之外的函数都可以检查和修改。所以，自定义SCC pass并不只属于SCC，而是属于所有SCC可能调用的方法，以及这些方法可能调用的所有方法。自定义SCC pass还属于可能调用SCC中的方法的所有方法。自定义SCC pass还需要保留当前的CallGraph对象，如果程序有任何修改，自定义SCC pass都应负责对CallGraph对象进行更新。自定义SCC pass可能会修改当前SCC的内容，但不允许在当前模块中添加或删除SCC。

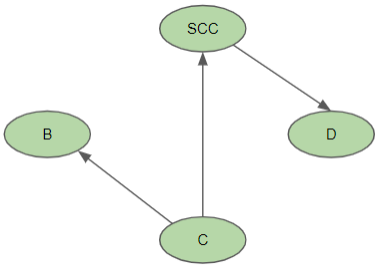

图 5. SCC pass

AMDGPU后端中的AMDGPUPerfHintAnalysis就是一个自定义SCC pass：

```
struct AMDGPUPerfHintAnalysis : public CallGraphSCCPass {
...
  bool runOnSCC(CallGraphSCC &SCC) override;

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesAll();
  }

  bool isMemoryBound(const Function *F) const;

  bool needsWaveLimiter(const Function *F) const;

  struct FuncInfo {
    unsigned MemInstCount;
    unsigned InstCount;
    unsigned IAMInstCount; // Indirect access memory instruction count
    unsigned LSMInstCount; // Large stride memory instruction count
    FuncInfo() : MemInstCount(0), InstCount(0), IAMInstCount(0),
                 LSMInstCount(0) {}
  };

  typedef ValueMap<const Function*, FuncInfo> FuncInfoMap;

private:

  FuncInfoMap FIM;
};
```

CallGraphSCCPass派生类的大部分功能在runOnSCC中实现。如果修改了模块，则runOnSCC()方法应返回true，否则返回false。

```
bool AMDGPUPerfHintAnalysis::runOnSCC(CallGraphSCC &SCC) {
  auto *TPC = getAnalysisIfAvailable<TargetPassConfig>();
  if (!TPC)
    return false;

  const TargetMachine &TM = TPC->getTM<TargetMachine>();

  bool Changed = false;
  for (CallGraphNode *I : SCC) {
    Function *F = I->getFunction();
    if (!F || F->isDeclaration())
      continue;

    const TargetSubtargetInfo *ST = TM.getSubtargetImpl(*F);
    AMDGPUPerfHint Analyzer(FIM, ST->getTargetLowering());

    if (Analyzer.runOnFunction(*F))
      Changed = true;
  }

  return Changed;
}
```

1.4 FunctionPass类

FunctionPass 类是Pass的派生类，类定义在llvm\include\llvm\Pass.h中：

```
class FunctionPass : public Pass {
public:
  explicit FunctionPass(char &pid) : Pass(PT_Function, pid) {}
  Pass *createPrinterPass(raw_ostream &OS,
                          const std::string &Banner) const override;

  virtual bool runOnFunction(Function &F) = 0;

  void assignPassManager(PMStack &PMS, PassManagerType T) override;
  PassManagerType getPotentialPassManagerType() const override;

protected:
  bool skipFunction(const Function &F) const;
};
```

与ModulePass子类相反，FunctionPass子类具有系统可以预期的局部行为。所有FunctionPass在程序中的每个方法上执行，独立于程序中的所有其它方法。 FunctionPass子类不需要以特定顺序执行，并且不会修改外部方法。

明确地说，FunctionPass子类不允许检查或修改当前正在处理的方法以外的其它方法，也不允许添加或删除当前模块的方法和全局变量。AMDGPU后端中的AMDGPUPromoteAllocaToVector pass就是一个function pass。这个pass通过将Alloc指令转换为向量消除Alloc指令：

```
class AMDGPUPromoteAllocaToVector : public FunctionPass {
……
  bool runOnFunction(Function &F) override;
……
  bool handleAlloca(AllocaInst &I);

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesCFG();
    FunctionPass::getAnalysisUsage(AU);
  }
};
```

FunctionPass派生类的大部分功能在runOnFunction()中实现。如果修改了模块，则runOnFunction()方法应返回true，否则返回false。以下是AMDGPUPromoteAllocaToVector pass的runOnFunction()：

```
bool AMDGPUPromoteAlloca::runOnFunction(Function &F) {
……
  bool Changed = false;
  BasicBlock &EntryBB = *F.begin();

  SmallVector<AllocaInst *, 16> Allocas;
  for (Instruction &I : EntryBB) {
    if (AllocaInst *AI = dyn_cast<AllocaInst>(&I))
      Allocas.push_back(AI);
  }

  for (AllocaInst *AI : Allocas) {
    if (handleAlloca(*AI))
      Changed = true;
  }

  return Changed;
}
```

1.5 LoopPass类

所有LoopPass在函数中的每个循环上执行，与函数中的所有其它循环无关。 LoopPass以循环嵌套顺序处理循环，最外层循环最后处理。LoopPass不仅仅是作用在IR中的某个loop结构，而是有可能（有时是必须）修改包含loop结构的外层结构（如包含该loop的函数），或将loop内部的指令移到loop外。类似的修改可能影响相邻的其它模块，从这个意义上说，LoopPass有点像FunctionPass 。

LoopPass子类可以使用LPPassManager接口更新循环嵌套。LoopPass子类需要重写三个虚函数来完成其工作。如果这些方法修改了程序，则应返回true；否则，应返回false。

作为主循环pass一部分运行的LoopPass子类，需要保存其它loop pass在pipeline中所需的所有函数分析pass（function analyses）。为了简化操作，LoopUtils.h提供了getLoopAnalysisUsage()函数。可以在LoopPass子类重写的getAnalysisUsage()函数中调用getLoopAnalysisUsage()函数，以获取正确的分析结果。getAnalysisUsage()函数的用法将在后文中介绍。

1.5 LoopPass类

LoopPass类是Pass的派生类，类定义在llvm/include/llvm/Analysis/LoopPass.h中：

```
class LoopPass : public Pass {
public:
  explicit LoopPass(char &pid) : Pass(PT_Loop, pid) {}
  Pass *createPrinterPass(raw_ostream &O,
                          const std::string &Banner) const override;
  virtual bool runOnLoop(Loop *L, LPPassManager &LPM) = 0;

  using llvm::Pass::doInitialization;
  using llvm::Pass::doFinalization;

  virtual bool doInitialization(Loop *L, LPPassManager &LPM) {
    return false;
  }

  virtual bool doFinalization() { return false; }
  void preparePassManager(PMStack &PMS) override;
  void assignPassManager(PMStack &PMS, PassManagerType PMT) override;
  PassManagerType getPotentialPassManagerType() const override {
    return PMT_LoopPassManager;
  }

protected:
  bool skipLoop(const Loop *L) const;
};
```

所有LoopPass在函数中的每个循环上执行，与函数中的所有其它循环无关。 LoopPass以循环嵌套顺序处理循环，最外层循环最后处理。LoopPass不仅仅是作用在IR中的某个loop结构，而是有可能（有时是必须）修改包含loop结构的外层结构（如包含该loop的函数），或将loop内部的指令移到loop外。类似的修改可能影响相邻的其它模块，从这个意义上说，LoopPass有点像unctionPass 。
LoopPass子类可以使用LPPassManager接口更新循环嵌套。LoopPass子类需要重写三个虚函数来完成其工作。如果这些方法修改了程序，则应返回true；否则，应返回false。
作为主循环pass一部分运行的LoopPass子类，需要保存其它loop pass在pipeline中所需的所有函数分析pass（function analyses）。为了简化操作，LoopUtils.h提供了getLoopAnalysisUsage()函数。可以在LoopPass子类重写的getAnalysisUsage()函数中调用getLoopAnalysisUsage()函数，以获取正确的分析结果。
INITIALIZE_PASS_DEPENDENCY(LoopPass)将初始化这组函数分析pass（function analyses）。

1.6 RegionPass类

RegionPass类是Pass的派生类，类定义在llvm/include/llvm/Analysis/RegionPass.h中：

```
class RegionPass : public Pass {
public:
  explicit RegionPass(char &pid) : Pass(PT_Region, pid) {}
  virtual bool runOnRegion(Region *R, RGPassManager &RGM) = 0;
  Pass *createPrinterPass(raw_ostream &O,
                          const std::string &Banner) const override;

  using llvm::Pass::doInitialization;
```

```
  using llvm::Pass::doFinalization;

  virtual bool doInitialization(Region *R, RGPassManager &RGM) { return false; }
  virtual bool doFinalization() { return false; }
  void preparePassManager(PMStack &PMS) override;
  void assignPassManager(PMStack &PMS,
                         PassManagerType PMT = PMT_RegionPassManager) override;

  PassManagerType getPotentialPassManagerType() const override {
    return PMT_RegionPassManager;
  }

protected:
  bool skipRegion(Region &R) const;
};
```

RegionPass较少使用，其用法与LoopPass有相似之处，不过是在函数中的每个单入口单出口region执行。RegionPass由RGPassManager管理，以嵌套顺序处理region，最外层region放在最后处理。RegionPass子类可以通过使用RGPassManager接口更新region树。编译器开发者可以重写RegionPass的三个虚函数来实现自定义RegionPass。如果方法修改了程序，则返回true；否则，返回false。

1.7 MachineFunctionPass类

MachineFunctionPass类是FunctionPass的派生类，类定义在llvm/include/llvm/CodeGen/MachineFunctionPass.h中：

```
class MachineFunctionPass : public FunctionPass {
public:
  bool doInitialization(Module&) override {
    RequiredProperties = getRequiredProperties();
    SetProperties = getSetProperties();
    ClearedProperties = getClearedProperties();
    return false;
  }
protected:
  explicit MachineFunctionPass(char &ID) : FunctionPass(ID) {}
  virtual bool runOnMachineFunction(MachineFunction &MF) = 0;
  void getAnalysisUsage(AnalysisUsage &AU) const override;
  virtual MachineFunctionProperties getRequiredProperties() const {
    return MachineFunctionProperties();
  }
  virtual MachineFunctionProperties getSetProperties() const {
    return MachineFunctionProperties();
  }
  virtual MachineFunctionProperties getClearedProperties() const {
    return MachineFunctionProperties();
```

```
  }

private:
  MachineFunctionProperties RequiredProperties;
  MachineFunctionProperties SetProperties;
  MachineFunctionProperties ClearedProperties;

  /// createPrinterPass - Get a machine function printer pass.
  Pass *createPrinterPass(raw_ostream &O,
                          const std::string &Banner) const override;

  bool runOnFunction(Function &F) override;
};
```

MachineFunctionPass是LLVM代码生成器的一部分，在程序中与机器相关的LLVM function IR上执行。代码生成器pass由TargetMachine :: addPassesToEmitFile和类似例程专门注册和初始化，因此通常不能从opt或bugpoint命令调用代码生成器pass。MachineFunctionPass也是FunctionPass，因此适用于FunctionPass的所有限制也都适用于MachineFunctionPass。MachineFunctionPasses还有其他限制。例如，MachineFunctionPasses不允许修改或生成任何LLVM IR Instruction、BasicBlock、Argument、Functions、GlobalVariables、GlobalAlias或者Module，也不能修改当前正在处理的MachineFunction 以外的MachineFunction 。

2、Pass管理器（Pass Manager）

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

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

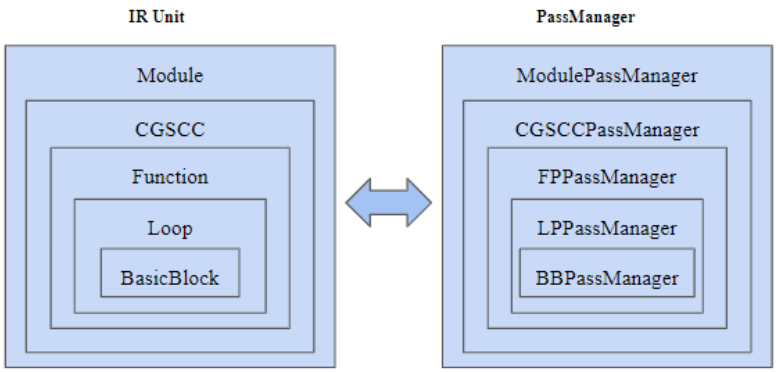

图 6. IR单元和PassManager类型

LLVM中有两类Pass管理器：Legacy Pass Manager和New Pass Manager。Legacy Pass Manager包含两个层次的类。llvm::Pass和 llvm::legacy::PassManagerBase。Pass类的功能上一节中已经提及，此处不再赘述。

PassManager类高效调度和运行pass。所有运行pass的LLVM工具都使用PassManager来执行这些pass。PassManager的责任是确保正确完成pass之间的交互。而当PassManager要以更优化的方式执行pass时，PassManager就必须维护有关pass之间如何交互以及pass之间依赖关系的信息。

PassManager通过两种方式减少pass序列的执行时间：第一，pass之间共享分析结果。PassManager的主要任务之一避免是重复计算分析结果，这就需要PassManager跟踪维护哪些分析可用、哪些分析失效以及哪些分析是必需的。PassManager跟踪分析结果的生命周期，并在不再需要某些分析结果时，释放这部分分析结果占用的内存，从而实现最优内存使用。第二，PassManager将pass连接起来，以pipeline的方式执行，可以获得更好的内存和缓存结果，从而改善了编译器的缓存行为。例如，当给出一系列连续的FunctionPass时，PassManager将在第一个函数上执行所有FunctionPass，然后在第二个函数上执行所有FunctionPass，依此类推。这种处理方式可改善缓存行为，因为这样一次只会处理LLVM IR的一个函数，而不是处理整个程序，减少了编译器的内存消耗。

LLVM中添加pass的方式非常灵活。AMDGPU后端是基于llvm::TargetPassConfig类派生AMDGPUPassConfig类，并通过重写其中的虚函数，如addIRPasses()、addCodeGenPrepare()、addInstSelector()等，在代码生成过程中的不同阶段（如如在寄存器分配之前、之后或在汇编代码生成之前）向AMDGPU后端目标添加pass，并应用自定义优化。以下是AMDGPU后端实现addIRPasses()的示例代码：

```
void AMDGPUPassConfig::addIRPasses() {
……
  addPass(createAMDGPUPrintfRuntimeBinding());

  addPass(createAMDGPUFixFunctionBitcastsPass());

  addPass(createAMDGPUPropagateAttributesEarlyPass(&TM));

  addPass(createAtomicExpandPass());
…..
 if (TM.getTargetTriple().getArch() == Triple::amdgcn) {
   addPass(createAMDGPUCodeGenPreparePass());
 }
……
```

在addIRPasses()实现中，只有amdgcn架构才添加CodeGenPrepare pass，即后端可以根据特定目标信息决定是否添加特定pass。

从2012年开始，LLVM开始开发New Pass Manager。Legacy Pass Manager下的pass是通过继承来定义pass接口，而New Pass Manager下的pass依赖于基于概念的多态（concept-based polymorphism）定义pass接口，也就是说，New Pass Manager下的pass没有显式接口或虚拟成员函数定义，也没有Legacy Pass Manager那样的Pass层次。所有pass都继承自CRTP（Curiously Recurring Template Pattern）mix-in PassInfoMixin<PassT>，如下所示：

```
class MyPass : public PassInfoMixin<MyPass > {
public:
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);
  static bool isRequired() { return true; }
……
};
```

如果自定义pass中定义了静态isRequired()方法并返回true，那么这个pass是一个必需（required ）pass。在执行时，pass管理器不能跳过必需pass，因为必需pass会计算需要保留的分析结果，后续的pass可能会用到这些分析结果。pass管理器本身是一个必需pass，因为pass管理器可能包含其它必需pass。

New Pass Manager下的pass都应有run()方法。不同于Lagacy Pass Manager下的runXXX()方法只返回一个布尔类型结果，New Pass Manager下的run()方法以IR单元以及分析管理器为参数，并返回PreservedAnalyses。例如，function pass的run()方法定义为：

PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM)

如果当前pass没有修改任何fucntion IR，其run()方法应返回PreservedAnalyses::all()，表示在当前pass执行之后，所有分析结果仍然有效。PreservedAnalyses是New Pass Manager提高效率的方式之一。

New Pass Manager下的pass注册在<root>/llvm/lib/Passes/PassRegistry.def文件中实现。

PassRegistry.def文件是LLVM核心库的pass注册表，其中描述了Analysis pass和Transform pass。编译器启动时，根据PassRegistry.def文件完成pass子系统的注册和初始化，并由PassRegistry类协助PassManager解决pass依赖关系。PassRegistry.def文件内容如下所示：

```
MODULE_ANALYSIS("callgraph", CallGraphAnalysis())
……
MODULE_PASS("always-inline", AlwaysInlinerPass())
……
CGSCC_ANALYSIS("no-op-cgscc", NoOpCGSCCAnalysis())
……
CGSCC_PASS("argpromotion", ArgumentPromotionPass())
……
FUNCTION_ANALYSIS("aa", AAManager())
……
FUNCTION_PASS("aa-eval", AAEvaluator())
……
```

New Pass Manager提供了模板化的 llvm::PassManager和llvm::AnalysisManager。模板化PassManager类定义如下：

```
class PassManager : public PassInfoMixin<
                PassManager<IRUnitT, AnalysisManagerT, ExtraArgTs...>> {
…..
```

}
当实例化New Pass Manager时，须以AnalysisManager<IRUnitT>作为参数。pass管理器会将该AnalysisManager传递给pass管理器运行的每个pass，并以pass的PreservedAnalyses返回结果为参数，调用AnalysisManager的失效例程(invalidation routine)。如果需要获取之前的分析结果，也要通过AnalysisManager。
LLVM中关于New Pass Manager的文档较少，本文涉及Pass管理器时仍以Legacy Pass Manager为主。
PassManager的主要职责之一是确保pass之间能够正确交互。因为PassManager试图优化pass的执行，所以必须知道pass之间如何交互以及各个pass之间存在什么样的依赖关系。为了记录这种依赖关系，每个pass可以声明哪些pass应该在当前pass之前执行，哪些pass可以使当前pass失效（因为运行任何Transform pass都可以使之前计算的分析结果失效）。通过在当前pass中实现getAnalysisUsage方法，可以为当前pass指定必需和失效pass集合。如果某个pass未实现getAnalysisUsage()方法，则默认为这个pass不需要提前运行任何pass。
3、AnalysisUsage和AnalysisResolver
AnalysisUsage表示pass的分析使用(analysis usage)情况信息。LLVM通过AnalysisUsage对象来处理pass依赖性。AnalysisUsage类定义在llvm\include\llvm\PassAnalysisSupport.h中。AnalysisUsage的类结构图如下所示:

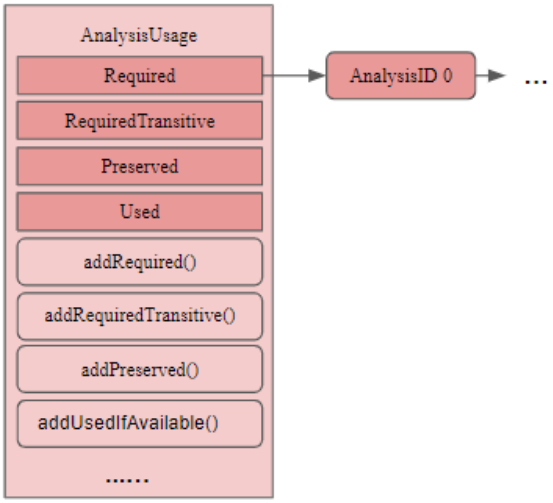

图 6. AnalysisUsage类

其中，向量Required、RequiredTransitive、Preserved、Used中分别保存当前pass需要使用的分析pass和需要保留结果的pass集合。成员函数addRequired()、addRequiredTransitive()、addPreserved()、addUsedIfAvailable()负责操作上述成员变量。
3.1 getAnalysisUsage()函数
编译器开发者在实现pass时，可以定义pass之间的依赖性。在LLVM中，通常通过实现getAnalysisUsage(AnalysisUsage&AU)方法来访问另一pass的数据结构。getAnalysisUsage()函数原型如下:

```
virtual void getAnalysisUsage(AnalysisUsage &Info) const;
```

前已提及，通过实现getAnalysisUsage方法，可以为pass指定必需和失效集。在实现getAnalysisUsage方法时，应在AnalysisUsage对象中指定需要哪些pass和使哪些pass失效。为此，pass可以调用AnalysisUsage对象的AnalysisUsage::addRequired<>方法，调度某个或某些pass在当前pass前执行。
例如，在AMDGPU后端的AMDGPULowerIntrinsics pass中需要用到TargetTransformInfoWrapperPass的分析结果，因此有：

```
  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<TargetTransformInfoWrapperPass>();
  }
```

TargetTransformInfoWrapperPass将codegen信息暴露给IR级别的pass。每个使用codegen信息的转换都分为三个部分：IR层次的analysis pass、提供信息的IR转换接口和使用特定目标hook的Codegen实现。TargetTransformInfoWrapperPass实现了其中的IR转换接口，这是IR级转换用来查询codegen的接口。通过调用addRequired()函数，表明AMDGPULowerIntrinsics pass需要用到TargetTransformInfoWrapperPass中的分析结果。当AMDGPULowerIntrinsics::runOnModule()方法调用expandMemIntrinsicUses()时，可通过getAnalysis()获得TargetTransformInfo引用：

```
bool AMDGPULowerIntrinsics::expandMemIntrinsicUses(Function &F) {
...
       const TargetTransformInfo &TTI =
          getAnalysis<TargetTransformInfoWrapperPass>().getTTI(*ParentFunc);
…
```

getAnalysis()函数定义在llvm\include\llvm\PassAnalysisSupport.h中：

```
template<typename AnalysisType>
AnalysisType &Pass::getAnalysis() const {
  assert(Resolver && "Pass has not been inserted into a PassManager object!");
  return getAnalysisID<AnalysisType>(&AnalysisType::ID);
}
```

getAnalysis()方法通过将模板参数指定为getAnalysisUsage()方法中声明的pass，可以得到pass的引用。这里的模板参数是TargetTransformInfoWrapperPass，然后通过TargetTransformInfoWrapperPass引用调用getTTI()函数得到TargetTransformInfo引用。

有些情况下，某些分析需要链接到其它分析才能完成。例如，某个别名分析pass实现需要链接到其它别名分析pass。这时，应调用addRequiredTransitive()方法而不是addRequired()方法。addRequiredTransitive()方法会通知PassManager，只要调用addRequiredTransitive()方法的pass（即提出请求的pass）还未销毁，被请求的pass就不能被销毁。addRequiredTransitive()方法较少使用。

PassManager的有效性直接受到其所调度的pass行为的影响。如果pass没有实现getAnalysisUsage()方法，pass执行期间得到分析结果不会被保留。如果后续pass要用到之前的分析结果，PassManager要重新启动之前已经运行过的pass，计算分析结果，这无疑增加了编译器的运行时间和内存开销。

3.2 AnalysisUsage::addPreserved<>函数

PassManager的任务之一是优化分析pass的执行，特别是避免重复计算数据。因此，LLVM允许pass声明保留现有分析结果（即不会使现有分析结果失效）。例如，简单的常量折叠pass不会修改CFG，因此不可能影响必经节点（dominator）分析的结果。默认情况下，应假定所有pass都会使其它pass失效。

AnalysisUsage类提供了几种addPreserved()方法。第一种方法是setPreservesAll()方法，用来表示pass（一般是analysis pass）没有修改程序，应保留所有之前的分析结果。比如AMDGPU后端中的别名分析pass：

```
void AMDGPUAAWrapperPass::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
}
```

第二种方法是setPreservesCFG()方法，用来表示pass更改了程序中的指令，但没有修改程序的CFG或终止（terminator）指令，例如AMDGPU backend中的SIInsertWaitcnts pass：

```
  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesCFG();
    AU.addRequired<MachinePostDominatorTree>();
    MachineFunctionPass::getAnalysisUsage(AU);
  }
```

3.3 AnalysisResolver

AnalysisResolver是一个简单的接口，Pass对象通过此接口从pass管理器中获得所有分析信息
AnalysisResolver类定义在llvm\include\llvm\PassAnalysisSupport.h中。AnalysisResolver的类结构图如下所示：

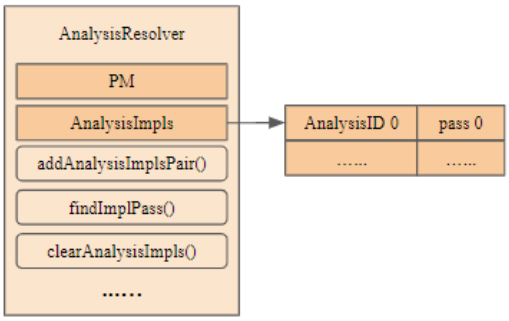

图 7. AnalysisResolver类

其中，AnalysisResolver类的成员变量PM是用于解析分析信息的PassManager，成员变量AnalysisImpls向量中保存了AnalysisID和pass的对应关系，可用于跟踪哪些pass实现了当前pass所需的接口：
std::vector<std::pair<AnalysisID, Pass *>> AnalysisImpls;
AnalysisResolver的成员函数findImplPass()和addAnalysisImplsPair()的代码实现都主要是围绕AnalysisImpls操作。为了建立不同pass之间的依赖关系，PassManager在调用run()方法遍历运行pass之前，会在PassManagerImpl::add()方法中调用findAnalysisPass()，得到当前pass依赖的passs对应的AnalysisID，并通过当前pass的getResolver()接口获得AnalysisResolver指针AR，然后再通过指针AR调用AnalysisResolver::addAnalysisImplsPair()，将AnalysisID及其对应的pass对象加入AnalysisResolver::AnalysisImpls，从而建立AnalysisID与依赖pass的映射关系。调用流程如图所示：

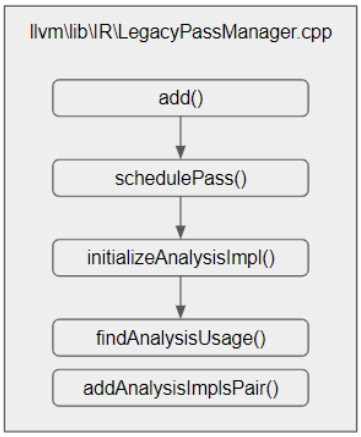

图 8. pass调用流程

其中的initializeAnalysisImpl()方法代码实现如下：

```
void PMDataManager::initializeAnalysisImpl(Pass *P) {
  AnalysisUsage *AnUsage = TPM->findAnalysisUsage(P);

  for (const AnalysisID ID : AnUsage->getRequiredSet()) {
    Pass *Impl = findAnalysisPass(ID, true);
    if (!Impl)
      // This may be analysis pass that is initialized on the fly.
```

```
    // If that is not the case then it will raise an assert when it is used.
    continue;
   AnalysisResolver *AR = P->getResolver();
   assert(AR && "Analysis Resolver is not set");
   AR->addAnalysisImplsPair(ID, Impl);
  }
}
```

AnalysisResolver的addAnalysisImplsPair()方法的代码实现如下：

```
 void addAnalysisImplsPair(AnalysisID PI, Pass *P) {
  if (findImplPass(PI) == P)
    return;
  std::pair<AnalysisID, Pass*> pir = std::make_pair(PI,P);
  AnalysisImpls.push_back(pir);
 }
```

综上所述，在编译器开发过程中，如果要建立不同pass之间的依赖关系，如pass A要使用pass B的分析结果，具体步骤如下：

1）pass A为了表明自己需要pass B，pass A需要在其getAnalysisUsage()方法中调用AU.addRequired<passBPass>()。

每个pass各自需要实现自己的getAnalysisUsage()函数，其中调用addRequiredID()或addRequired()或其它函数，将该pass所依赖的pass的AnalysisID加入对应变量，如AnalysisUsage::Required。例如，pass A依赖于pass B，passAPass::getAnalysisUsage()会将pass B的AnalysisID加入AnalysisUsage::Required。

```
void passAPass::getAnalysisUsage(AnalysisUsage& pUsage) const {
 pUsage.addRequired<passBPass>();
}
```

2）为了在编译器中正确建立pass之间的依赖关系，需要考虑两个方面。第一是不同pass的执行顺序，即某个pass一定要在另一个pass之前运行。第二是不同pass的输入输出之间的依赖，即某个pass要用到另一个pass的分析结果。不能保证pass执行顺序的正确，就不能保证输入输出结果的依赖。

当pass运行过程中需要用到其依赖pass的处理结果时，可以以所依赖pass的AnalysisID为参数，在当前pass的XXXPass::runOnModule()（或runOnFunction()、runOnBasicBlock()等）方法中调用Pass::getAnalysis()。Pass::getAnalysis()再调用Pass::getAnalysisID()，进而调用AnalysisResolver::findImplPass()，从AnalysisResolver::AnalysisImpls中找到AnalysisID的依赖pass。例如，pass A要使用pass B的分析结果，就应当在pass A的runOnModule()方法中通过调用getAnalysis<passBPass>()获得pass B的引用（或指针）：

```
bool passAPass::runOnModule(Module &pModule) {
…….
 passBPass *passB = getAnalysis<passBPass >();
…...
}
```

getAnalysis根据Analysis id找到对应的pass，子类可以使用此方法获取在getAnalysisUsage()中声称要使用的分析信息：

```
template<typename AnalysisType>
AnalysisType &Pass::getAnalysis() const {
 assert(Resolver && "Pass has not been inserted into a PassManager object!");
 return getAnalysisID<AnalysisType>(&AnalysisType::ID);
}
```

Pass::getAnalysis()再调用Pass::getAnalysisID()，Pass::getAnalysisID()代码实现如下：

```
template<typename AnalysisType>
AnalysisType &Pass::getAnalysisID(AnalysisID PI) const {
  assert(PI && "getAnalysis for unregistered pass!");
  assert(Resolver&&"Pass has not been inserted into a PassManager object!");

  Pass *ResultPass = Resolver->findImplPass(PI);
  assert(ResultPass &&
         "getAnalysis*() called on an analysis that was not "
         "'required' by pass!");

  return *(AnalysisType*)ResultPass->getAdjustedAnalysisPointer(PI);
}
```

AnalysisResolver::findImplPass()代码实现如下：

```
  Pass *findImplPass(AnalysisID PI) {
    Pass *ResultPass = nullptr;
    for (const auto &AnalysisImpl : AnalysisImpls) {
      if (AnalysisImpl.first == PI) {
        ResultPass = AnalysisImpl.second;
        break;
      }
    }
    return ResultPass;
  }
```

3) pass B要实现一个返回分析结果的方法，如getAnalysisResult()。pass A可以使用getAnalysis<passBPass>()获取对pass B对象的引用，然后通过pass B对象调用此方法以获取pass B的分析结果。

```
  passBPass *passB = getAnalysis<passBPass >();
  AR = passB->getAnalysisResult();
```

AMDGPU backend用类似方法通过getAnalysis()获得TargetTransformInfo引用：

```
bool AMDGPULowerIntrinsics::expandMemIntrinsicUses(Function &F) {
...
        const TargetTransformInfo &TTI =
           getAnalysis<TargetTransformInfoWrapperPass>().getTTI(*ParentFunc);
```

参考文献

[1] https://llvm.org/docs/Passes.html
[2] https://llvm.org/devmtg/2019-10/slides/Warzynski-WritingAnLLVMPass.pdf