链滴

LLVM 概述二：LLVM 设计精髓及其优势

作者：Hanseltu
原文链接：https://ld246.com/article/1554474216576
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

本文是 LVM 概述一：编译器背景及 LLVM 初探的下半部分，本部分介绍 LLVM 主要的设计思想及优。
<h2 id="LLVM的三阶段实施">LLVM 的三阶段实施</h2>
在基于 LLVM 的编译器中，依然保持原始编译器的三阶段设计，LLVM 前端负责解析，验证和诊输入代码中的错误，然后将解析的代码转换为 LLVM IR（通常但不总是通过构建 AST 然后将 AST 转为 LLVM IR）。 该 IR 可选地通过一系列改进代码的分析和优化过程提供，然后被发送到代码生成器生成本机机器代码，如图 3 所示。 这是三阶段设计的非常简单的实现，但是这个简单的描述巧妙地现了 LLVM 架构从 LLVM IR 派生的一些功能和灵活性。

图3 LLVM三阶段实施
<h3 id="LLVM-IR是完整的代码表示">LLVM IR 是完整的代码表示</h3>
需要特别注意的是，LLVM IR 既是明确指定的，也是优化器的唯一接口。这个属性意味着为 LLV 编写前端所需要知道的就是 LLVM IR 是什么，它是如何工作的，以及它的变量分别代表什么。由于 LVM IR 具有统一的文本形式，因此构建将 LLVM IR 作为文本输出的前端既可能也是合理的，然后使用 Unix 管道通过你选择的优化器序列和代码生成器发送它。
这听起来可能是令人惊讶，但这实际上是 LLVM 的一个非常新颖的属性，也是其在各种不同应中取得成功的主要原因之一。即使是广泛成功且构思相对较好的 GCC 编译器也没有这个属性：它的 GMPLE 中级表示不是一个独立的表示。举个简单的例子，当 GCC 代码生成器发出 DWARF 调试信息，它会返回并遍历源级“树”形式。 GIMPLE 本身对代码中的操作使用“元组”表示，但（至少从 GC 4.5 开始）仍然将操作数表示为返回源级树形式的引用。
这意味着前端作者需要知道并生成 GCC 的树数据结构以及 GIMPLE 来编写 GCC 前端。 GCC 端有类似的问题，所以他们也需要了解 RTL 后端的工作原理。最后，GCC 没有办法转储“代表我代的所有内容”，或者以文本形式读取和编写 GIMPLE（以及构成代码表示的相关数据结构）的方法。果是，用 GCC 进行实验相对较难，因此它的前端相对较少。
<h3 id="LLVM是一个库集合">LLVM 是一个库集合</h3>
在 LLVM IR 的设计之后，LLVM 的还有一个最重要的特点是它被设计为一组库，而不是像 GCC 样的单片命令行编译器或像 JVM 或.NET 虚拟机那样的不透明虚拟机。 LLVM 是一种基础结构，是组有用的编译器技术设施，它可以解决特定问题（例如构建 C 编译器或特殊效果管道中的优化器）。然它是其最强大的功能之一，但它也是其最难理解的设计点之一。
让我们看看优化器的设计作为一个例子：它读取 LLVM IR，稍微分析处理它，然后发出 LLVM I，希望能更快地执行。在 LLVM 中（与许多其他编译器一样），优化器被组织为不同优化传递的管道每个优化传递在输入上运行并且有机会做某事。传递的常见示例是内联器（将函数体替换为调用站点，表达式重新关联，循环不变代码运动等。根据优化级别，运行不同的传递：例如在-O0（无优化） Cang 编译器不运行任何传递，在-O3 它在其优化器中运行一系列 67 次传递（从 LLVM 2.8 开始）。
每个 LLVM 传递都是 C ++ 类编写的，它从 <code>Pass</code> 类派生（间接）。大多数传都是用单个 <code>.cpp</code> 文件编写的，而 <code>Pass</code> 类的子类是在匿名命名间中定义的（这使得它对定义文件完全私有）。为了使传递有用，文件外部的代码必须能够获取它，此从文件中导出单个函数（创建传递）。这是一个稍微简化的例子，用于使任务具体化 [1]。
<pre><code class="highlight-chroma">namespace {
  class Hello : publ c FunctionPass {
  public:
    // Print out the ames of functions in the LLVM IR being optimized.
    virtual bool ru OnFunction(Function &F) {
      cerr << " ello: " << F.getName() << "\n";

      return false;
    }
  };
}
FunctionPass *cre
teHelloPass() { return new Hello(); }
</code></pre>
如上所述，LLVM 优化器提供了许多不同的传递，每个传递都以类似的方式编写。这些传递被编成一个或多个 <code>.o</code> 文件，然后将这些文件构建到一系列存档库（Unix 系统上的 <cod
>.a</code> 文件）中。这些库提供了各种各样的分析和转换功能，并且通道尽可能松散耦合：如果们依赖于其他分析来完成它们的工作，它们应该独立存在，或者在其他通道中明确声明它们的依赖关。当给出一系列要运行的传递时，LLVM PassManager 使用显式依赖关系信息来满足这些依赖关系优化传递的执行。
库和抽象功能很强大，但它们实际上并不能解决问题。当有人想要构建一个可以从编译器技术中益的新工具时，有趣的是，可能是用于图像处理语言的 JIT 编译器。这个 JIT 编译器的实现者考虑了组约束：例如，图像处理语言可能对编译时延迟高度敏感，并且具有一些惯用语言属性，这些属性对出于性能原因进行优化非常重要。
LLVM 优化器的基于库的设计允许我们的实现者选择执行传递的顺序，以及哪些对图像处理域有义：如果所有内容都被定义为单个大函数，则不会感觉浪费时间内联。如果指针很少，别名分析和内优化就不值得烦恼了。然而，尽管我们尽最大努力，但 LLVM 并没有神奇地解决所有优化问题！由于递子系统是模块化的，并且 PassManager 本身对传递的内部不了解，因此实现者可以自由地实现他自己的语言特定传递，以弥补 LLVM 优化器中的缺陷或明确的语言特定优化机会。图 4 显示了我们设的 XYZ 图像处理系统的一个简单示例：

图4 LLVM处理下的XYZ系统
一旦选择了一组优化（并且对代码生成器做出了类似的决定），图像处理编译器就被构建到可执或动态库中。由于对 LLVM 优化传递的唯一引用是每个 <code>.o</code> 文件中定义的简单 create 函数，并且由于优化器位于 <code>.a</code> 归档库中，因此只有实际使用的优化传递链接到最应用程序，而不是整个 LLVM 优化器。在上面的示例中，由于存在对 PassA 和 PassB 的引用，它们被链接。由于 PassB 使用 PassD 进行一些分析，因此 PassD 被链接。但是，因为 PassC（以及许多他优化）未被使用，它的代码没有链接到图像处理应用程序。
这就是基于库的 LLVM 设计的强大功能。这种简单的设计方法允许 LLVM 提供大量功能，其中些功能可能仅对特定受众有用，而不会损害只想做简单事情的库的客户端。相比之下，传统的编译器化器被构建为紧密互连的大量代码，这对于子集，推理和加速来说要困难得多。使用 LLVM，您可以解各个优化器，而无需了解整个系统如何组合在一起的。
这种基于库的设计也是为什么很多人误解 LLVM 的原因：LLVM 库有很多功能，但它们实际上并有自己做任何事情。由库的客户端（例如，Clang C 编译器）的设计者决定如何最好地使用这些部件这种仔细的分层，分解和关注子集能力也是 LLVM 优化器可用于不同环境中如此广泛的不同应用的原。另外，仅仅因为 LLVM 提供 JIT 编译功能，并不意味着每个客户端都使用它。
<h2 id="可重定向LLVM代码生成器的设计">可重定向 LLVM 代码生成器的设计</h2>
LLVM 代码生成器负责将 LLVM IR 转换为目标特定的机器代码。一方面，代码生成器的工作就为任何给定的目标生成最好的机器代码。理想情况下，每个代码生成器应该是目标的完全自定义代码但另一方面，每个目标的代码生成器需要解决非常类似的问题。例如，每个目标需要为寄存器分配值尽管每个目标具有不同的寄存器文件，但应尽可能共享所使用的算法。
与优化器中的方法类似，LLVM 的代码生成器将代码生成问题分解为单独的传递 - 指令选择，寄器分配，调度，代码布局优化和汇编发射 - 并提供许多默认运行的内置传递。然后，开发人员有机会默认传递中进行选择，覆盖默认值并根据需要实现完全自定义的特定于目标的传递。例如，x86 后端用寄存器压力降低调度程序，因为它只有很少的寄存器，但 PowerPC 后端使用延迟优化调度程序，为它有很多寄存器。 x86 后端使用自定义传递来处理 x87 浮点堆栈，ARM 后端使用自定义传递将常池放置在需要的函数内。这种灵活性允许开发人员生成出色的代码，而无需从头开始为其目标编写整代码生成器。
<h3 id="LLVM目标描述文件">LLVM 目标描述文件</h3>
“混合和匹配”方法允许开发人员选择对其体系结构有意义的内容，并允许跨越不同目标重用大

代码。 这带来了另一个挑战：每个共享组件都需要能够以通用方式推断目标特定属性。 例如，共享存器分配器需要知道每个目标的寄存器文件以及指令与其寄存器操作数之间存在的约束。 LLVM 的解方案是为每个目标提供由 tblgen 工具处理的声明性域特定语言（一组 <code>.td</code> 文件）的目标描述。 x86 目标的（简化）构建过程如图 5 所示。
<img src="https://ld246.com/images/img-loading.svg" alt="tu5.png" data-src="https://3logfile.com/file/2019/04/tu5-ff7064e1.png?imageView2/2/interlace/1/format/jpg">
图5 简化的X86目标机器定义
文件支持的不同子系统允许开发人员构建其目标的不同部分。 例如，x86 后端定义了一个寄存类，它包含所有名为“GR32”的 32 位寄存器（在 <code>.td</code> 文件中，目标特定定义都是写），如下所示：
<pre><code class="highlight-chroma">def GR32 : RegisterClass<[i32], 32,
  [EAX, ECX, EDX, SI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
</code></pre>
这个定义说明这个类中的寄存器可以保存 32 位整数值（“i32”），或者最好和 32 位对齐，具指定的 16 个寄存器（在 <code>.td</code> 文件的其他地方定义）并有更多信息指定首选分配顺和其他内容。 给定此定义，特定指令可以引用它，将其用作操作数。 例如，“补充 32 位寄存器”指定义为：
<pre><code class="highlight-chroma">let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
                (outs GR32:$dst), (ins GR32:$src),
                "not{l}\t$dst",
                [(set GR32:$dst, (not GR32:$src))]>;</code></pre>
这个定义说 NOT32r 是一个指令（它使用 <code>I</code> tblgen 类），指定编码信息（<code>0xF7</code>, <code>MRM2r</code>），指定它定义一个“输出”32 位寄存器 <code>$dst</code> 并且有一个 32 位寄存器“输入”命名 <code>$src</code>（上面定义的 <code>GR32</code> 寄存器类定义哪些寄存器对操作数有效），指定指令的汇编语法（使用{}语法处理 AT&T 和 Intel 语法），指定指令的效果并提供它的模式应该在最后一行匹配。第一行的“let”约束告诉寄存器配器必须将输入和输出寄存器分配给同一物理寄存器。
这个定义是对指令的非常详细的描述，并且通用的 LLVM 代码可以使用从它派生的信息（通过 tbgen 工具）做很多事情。这个定义足以使指令选择通过编译器的输入 IR 代码上的模式匹配来形成该令。它还告诉寄存器分配器如何处理它，足以对指令进行编码和解码以加工代码字节，并且足以以文形式解析和打印指令。这些功能允许 x86 目标支持生成独立的 x86 汇编程序（它是“gas”GNU 汇程序的直接替代品）和目标描述中的反汇编程序以及处理 JIT 指令的编码。
除了提供有用的功能之外，由相同的“真实”产生的多条信息由于其他原因是好的。这种方法使汇编器和反汇编器在汇编语法或二进制编码中彼此不同意几乎是不可行的。它还使目标描述易于测试指令编码可以进行单元测试，而不必涉及整个代码生成器。
虽然我们的目标是以一个很好的声明形式将尽可能多的目标信息放到 <code>.td</code> 文件，但我们仍然没有。相反，我们要求目标作者为各种支持例程编写一些 C ++ 代码，并实现他们可能要的任何目标特定传递（如 <code>X86FloatingPoint.cpp</code>，它处理 x87 浮点堆栈）。随着LLVM 继续增加新目标，增加可以在.td 文件中表达的目标数量变得越来越重要，并且我们继续增加达的目标性。<code>.td</code> 文件来处理这个。一个很大的好处是随着时间的推移，它可更容地在 LLVM 中编写目标代码。
<h2 id="模块化设计提供的有趣扩展功能">模块化设计提供的有趣扩展功能</h2>
除了通常优雅的设计外，模块化还为 LLVM 库的客户提供了一些有趣的功能。 这些功能源于 LL

M 提供功能，但让客户决定如何使用它的大多数策略。
<h3 id="选择每个阶段运行的时间和地点">选择每个阶段运行的时间和地点</h3>
如前所述，LLVM IR 可以有效地（反）序列化为/称为 LLVM bitcode 的二进制格式。 由于 LLV IR 是自包含的，并且序列化是一个无损过程，我们可以进行部分编译，将进度保存到磁盘，然后在来的某个时间点继续工作。 此功能提供了许多有趣的功能，包括对链接时和安装时优化的支持，这两功能都会从“编译时”延迟代码生成。
链接时优化（LTO）解决了编译器传统上一次只能看到一个转换单元（例如，带有其所有头的 <c de>.c</code> 文件）的问题，因此不能跨文件边界进行优化（如内联）。像 Clang 这样的 LLVM 译器使用 <code>-flto</code> 或 <code>-O4</code> 命令行选项来支持它。 此选项指示编译向 <code>.ofile</code> 发出 LLVM bitcode，而不是写出本机对象文件，并将代码生成延迟到链时间，如图 6 所示。
<img src="https://ld246.com/images/img-loading.svg" alt="tu6.png" data-src="https://3logfile.com/file/2019/04/tu6-a41f601e.png?imageView2/2/interlace/1/format/jpg">
图6 链接时优化过程
细节因您所使用的操作系统而异，但重要的是链接器检测到 <code>.o</code> 文件中的 LLVM bitcode 而不是本机对象文件。当它看到这一点时，它会将所有 bitcode 文件读入内存，将它们链接一起，然后在聚合上运行 LLVM 优化器。由于优化器现在可以看到代码的更大部分，因此它可以内联传播常量，执行更积极的死代码消除，以及更多跨文件边界。虽然许多现代编译器支持 LTO，但大多（例如，GCC，Open64，英特尔编译器等）都是通过昂贵且缓慢的序列化过程来实现的。在 LLVM ，LTO 自然地脱离了系统的设计，并且适用于不同的源语言（与许多其他编译器不同），因为 IR 是正的源语言中立。
安装时优化是延迟代码生成的想法，甚至晚于链接时间，一直到安装时间，如图 7 所示。安装时是一个非常有趣的时间（如果软件在一个盒子中发送，下载，上传到移动设备等），因为这是当你找你所针对的设备的细节时。例如，在 x86 系列中，存在各种各样的芯片和特性。通过延迟指令选择，度和代码生成的其他方面，您可以为应用程序最终运行的特定硬件选择最佳答案。
<img src="https://ld246.com/images/img-loading.svg" alt="tu7.png" data-src="https://3logfile.com/file/2019/04/tu7-c4bfe9f6.png?imageView2/2/interlace/1/format/jpg">
图7 安装时优化过程
<h3 id="优化器的单元测试">优化器的单元测试</h3>
编译器非常复杂，质量很重要，因此测试至关重要。例如，在修复导致优化器崩溃的错误之后，添加回归测试以确保它不会再次发生。测试这种方法的传统方法是编写一个通过编译器运行的 <code> c</code> 文件，并拥有一个测试工具来验证编译器是否崩溃。例如，这是 GCC 测试套件使用的方 。
这种方法的问题在于编译器由许多不同的子系统组成，甚至包括优化器中的许多不同的传递，所这些传递都有机会在输入代码到达之前有问题的代码时改变输入代码的样子。如果前端或早期优化器生了某些变化，测试用例很容易无法测试它应该测试的内容。
通过使用 LLVM IR 的文本形式和模块化优化器，LLVM 测试套件具有高度集中的回归测试，可从磁盘加载 LLVM IR，通过一次优化传递运行它，并验证预期的行为。除了崩溃之外，更复杂的行为试想要验证实际执行了优化。这是一个简单的测试用例，它检查常量传播过程是否与添加指令一起使 ：
<pre><code class="highlight-chroma">; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test()

  %A = add i32 4,

  ret i32 %A
  ; CHECK: @test()
  ; CHECK: ret i32 9
}
</code></pre>
RUN 行指定要执行的命令：在本例中为 <code>opt</code> 和 <code>FileCheck</code> 令行工具。 <code>opt</code> 程序是 LLVM 传递管理器的简单包装器，它连接所有标准传递（ 且可以动态加载包含其他传递的插件）并将它们公开到命令行。<code> FileCheck</code> 工具验

其标准输入是否与一系列 <code>CHECK</code> 指令匹配。在这种情况下，这个简单的测试验证了 <code>constprop</code> 传递是将 4 和 5 的 <code>add</code> 和成 9。
虽然这看起来像是一个非常简单的例子，但是通过编写 <code>.c</code> 文件来测试很难：端经常在解析时进行常量折叠，因此编写使其向下游变为常量的代码非常困难和脆弱折叠优化通过。为我们可以将 LLVM IR 作为文本加载并通过我们感兴趣的特定优化传递发送它，然后将结果转储为一个文本文件，对于回归和功能测试来说，确切地测试我们想要的内容真的很简单。
<h3 id="使用BugPoint自动减少测试用例">使用 BugPoint 自动减少测试用例</h3>
当在 LLVM 库的编译器或其他客户端中发现错误时，修复它的第一步是获得一个再现问题的测用例。一旦有了测试用例，最好将其最小化为最小的再现问题的示例，并将其缩小到发生问题的 LLVM 部分，例如故障时的优化传递。虽然您最终会学习如何执行此操作，但是对于编译器生成错误代码但会崩溃的情况，此过程非常繁琐，手动调试特别痛苦。
LLVM BugPoint 工具 [2] 使用 LLVM 的 IR 序列化和模块化设计来自动执此过程。例如，给定输入 <code>.ll</code> 或 <code>.bc</code> 文件以及导致优化器崩溃的化传递列表，BugPoint 会将输入减少到一个小测试用例并确定哪个优化器出错。然后它输出简化的试用例和用于重现失败的 <code>opt</code> 命令。它通过使用类似于“delta debugging”的技来减少输入和优化器传递列表。因为它知道 LLVM IR 的结构，所以与标准的“delta”命令行工具不，BugPoint 不会浪费时间生成无效的 IR 来输入优化器。
在更复杂的错误编译情况下，您可以指定输入代码生成器信息，传递给可执行文件的命令行以及考输出。 BugPoint 将首先确定问题是由优化器还是代码生成器引起的，然后将测试用例重复分为两分：一部分被发送到“已知良好”组件，另一部分被发送到“已知错误” “零件。通过迭代地将越越多的代码移出已发送到已知错误代码生成器的分区，它的特点是减少了测试用例。
BugPoint 是一个非常简单的工具，在 LLVM 的整个生命周期中节省了许多测试时间。没有其他源编译器具有类似功能强大的工具，因为它依赖于明确定义的中间表示。也就是说，BugPoint 并不美，并且会从重写中受益。它可以追溯到 2002 年，并且通常只有当某人有一个非常棘手的错误来追现有工具处理得不好时才会改进。它随着时间的推移而不断增长，在没有一致设计或所有者的情况下累新功能（例如 JIT 调试）。
<h2 id="回顾和未来的方向">回顾和未来的方向</h2>
LLVM 的模块化最初并非旨在直接实现此处描述的任何目标。这是一种自卫机制：很明显，我们会在第一次尝试时把一切都弄好。例如，存在模块化传递管道，以便更容易隔离传递，以便在被更好实现**[3]**替换后可以丢弃它们。
LLVM 保持灵活的另一个主要方面是我们愿意重新考虑先前的决策并对 API 进行广泛的更改而不担心向后兼容性。例如，对 LLVM IR 本身的侵入性更改需要更新所有优化过程并导致对 C ++ API 大量流失。我们已经多次这样做了，虽然它会给客户带来痛苦，但是保持快速前进的进展是正确的。了让外部客户端更轻松（并支持其他语言的绑定），我们为许多流行的 API（旨在非常稳定）提供 C 装器，新版本的 LLVM 旨在继续读取旧的 <code>.ll</code> 和 <code>.bc</code> 文件。
展望未来，我们希望继续使 LLVM 更加模块化，更易于子集化。例如，代码生成器仍然过于单：目前无法根据功能对 LLVM 进行子集化。例如，如果您想使用 JIT，但不需要内联汇编，异常处理调试信息生成，则应该可以构建代码生成器而无需链接以支持这些功能。我们还不断提高优化器和代生成器生成的代码质量，添加 IR 功能以更好地支持新语言和目标构造，并为在 LLVM 中执行高级语特定优化添加更好的支持。
LLVM 项目以多种方式不断发展和完善。看到 LLVM 在其他项目中使用的不同方式的数量以及它何在设计师从未想过的令人惊讶的新环境中不断出现，真是令人兴奋。新的 LLDB 调试器就是一个很的例子：它使用 Clang 的 C / C ++ / Objective-C 解析器来解析表达式，使用 LLVM JIT 将它们转为目标代码，使用 LLVM 反汇编程序，并使用 LLVM 目标来处理调用约定等等。能够重用这些现有码允许开发调试器的人专注于编写调试器逻辑，而不是重新实现另一个（边缘正确的）C ++ 解析器
尽管迄今为止 LLVM 取得了成功，但仍有许多工作要做。随着年龄的增长 LLVM 将变得不那么活和更加复杂化的风险将永远存在。虽然这个问题没有最终的答案，但我希望继续接触新问题领域，新评估先前决策的意愿，以及重新设计和丢弃代码将有所帮助。毕竟，目标不是完美，而是随着时间推移不断变好。
<hr>
上半部分：LLVM 概述一：编译器背景及 LLVM 初探

<hr>
<h2 id="Footnotes">Footnotes</h2>
<ol start="6">
For all the details, please see_Writing an LLVM Pass manual_at <code>http://llvm.org/docs/WritingAnLLVMPass.html</code>.
<code>http://llvm.org/docs/Bugpoint.html</code>
I often say that none of the subsystems in LLVM are really good until they have been rewritten at least once.