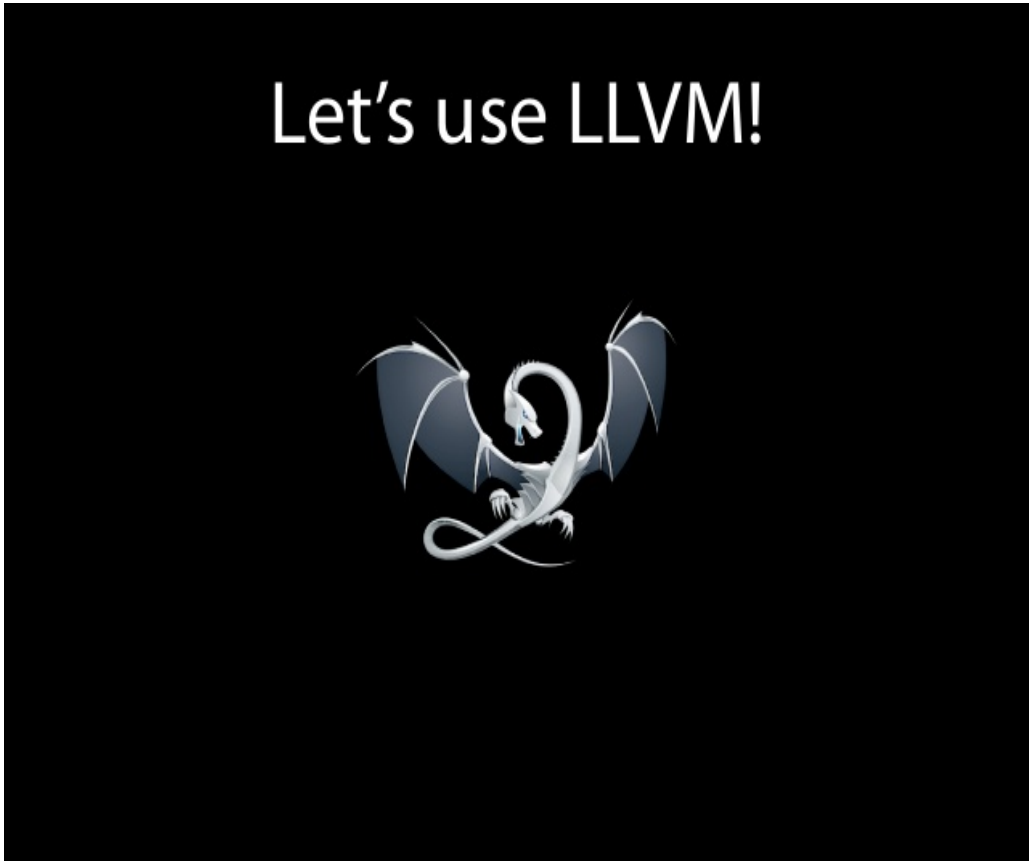
为什么人人都该懂点LLVM

LLVM (/lib/tag/LLVM) 2015-08-24 14:36:54 发布

只要你和程序打交道，了解编译器架构就会令你受益无穷——无论是分析程序效率，还是模拟新的处理器和操作系统。通过本文介绍，即使你对编译器原本一知半解，也能开始用LLVM，来完成有意思的工作。



LLVM是什么？

LLVM是一个好用、好玩，而且超前的系统语言（比如C和C++语言）编译器。

当然，因为LLVM实在太强大，你会听到许多其他特性（它可以是个JIT；支持了一大批非类C语言；还是App Store上的一种新的发布方式等等）。这些都是真的，不过就这篇文章而言，还是上面的定义更重要。

下面是一些让LLVM与众不同的原因：

- LLVM的“中间表示”（IR）是一项大创新。LLVM的程序表示方法真的“可读”（如果你会读汇编）。虽然看上去这没什么要紧，但要知道，其他编译器的中间表示大多是种内存中的复杂数据结构，以至于很难写出来，这让其他编译器既难懂又难以实现。
- 然而LLVM并非如此。其架构远比其他编译器要模块化得多。这种优点可能部分来自于它的最初实现者。

- 尽管LLVM给我们这些狂热的学术黑客提供了一种研究工具的选择，它还是一款有大公司做后台的工业级编译器。这意味着你不需要去在“强大的编译器”和“可玩的编译器”之间做妥协——不像你在Java世界中必须在HotSpot和Jikes之间权衡那样。

为什么人人需要懂点儿LLVM？

是，LLVM是一款酷炫的编译器，但是如果不做编译器研究，还有什么理由要管它？

答：只要你和程序打交道，了解编译器架构就会令你受益，而且从我个人经验来看，非常有用。利用它，可以分析程序要多久一次来完成某项工作；改造程序，使其更适用于你的系统，或者模拟一个新的处理器架构或操作系统——只需稍加改动，而不需要自己烧个芯片，或者写个内核。对于计算机科学研究者来说，编译器远比他们想象中重要。建议你先试试LLVM，而不用hack下面这些工具（除非你真有重要的理由）：

- 架构模拟器；
- 动态二进制分析工具，比如Pin；
- 源代码变换（简单的比如sed，复杂一些的比如抽象语法树的分析和序列化）；
- 修改内核来干预系统调用；
- 任何和虚拟机管理程序相似的东西。
  就算一个编译器不能完美地适合你的任务，相比于从源码到源码的翻译工作，它可以节省你九成精力。

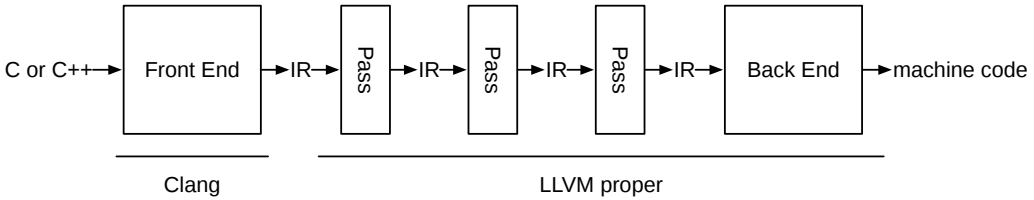
下面是一些巧妙利用了LLVM，而又不是在做编译器的研究项目：

- UIUC的Virtual Ghost，展示了你可以用编译器来保护挂掉的系统内核中的进程。
- UW的CoreDet利用LLVM实现了多线程程序的确定性。
- 在我们的近似计算工作中，我们使用LLVM流程来给程序注入错误信息，以模仿一些易出错的硬件。
  重要的话说三遍：LLVM不是只用来实现编译优化的！LLVM不是只用来实现编译优化的！LLVM不是只用来实现编译优化的！

组成部分

LLVM架构的主要组成部分如下（事实上也是所有现代编译器架构）：

前端，流程（Pass），后端



下面分别来解释：

- 前端获取你的源代码然后将它转变为某种中间表示。这种翻译简化了编译器其他部分的工作，这样它们就不需要面对比如C++源码的所有复杂性了。作为一个豪迈人，你很可能不想再做这部分工作；可以不加改动地使用Clang来完成。
- “流程”将程序在中间表示之间互相变换。一般情况下，流程也用来优化代码：流程输出的（中间表示）程序和它输入的（中间表示）程序相比在功能上完全相同，只是在性能上得到改进。这部分通常是给你发挥的地方。你的研究工具可以通过观察和修改编译过程流中的IR来完成任务。
- 后端部分可以生成实际运行的机器码。你几乎肯定不想动这部分了。
  虽然当今大多数编译器都使用了这种架构，但是LLVM有一点值得注意而与众不同：整个过程中，程序都使用了同一种中间表示。在其他编译器中，可能每一个流程产出的代码都有一种独特的格式。LLVM在这一点上对hackers大为有利。我们不需要担心我们的改动该插在哪个位置，只要放在前后端之间某个地方就足够了。

开始

让我们开干吧。

获取LLVM

首先需要安装LLVM。Linux的诸发行版中一般已经装好了LLVM和Clang的包，你直接用便是。但你还是需要确认一下机子里的版本，是不是有所有你要用到的头文件。在OS X系统中，和XCode一起安装的LLVM就不是那么完整。还好，用CMake从源码构建LLVM也没有多难。通常你只需要构建LLVM本身，因为你的系统提供的Clang已经够用（只要版本是匹配的，如果不是，你也可以自己构建Clang）。

具体在OS X上，Brandon Holt有一个不错的指导文章。用Homebrew也可以安装LLVM。

去读手册

你需要对文档有所了解。我找到了一些值得一看的链接：

- 自动生成的Doxygen文档页 (http://llvm.org/doxygen/)非常重要。要想搞定LLVM，你必须要以这些API的文档维生。这些页面可能不太好找，所以我推荐你直接用Google搜索。只要你在搜索的函数或者类名后面加上“LLVM”，你一般就可以用Google找到正确的文档页面了。（如果

你够勤奋，你甚至可以“训练”你的Google，使得在不输入LLVM的情况下它也可以把LLVM的相关结果推到最前面）虽然听上去有点逗，不过你真的需要这样找LLVM的API文档——反正我没找到其他的好方法。

- 《语言参考手册》(http://llvm.org/docs/LangRef.html)也非常有用，如果你曾被LLVM IR dump里面的语法搞糊涂的话。
- 《开发者手册》(http://llvm.org/docs/ProgrammersManual.html)描述了一些LLVM特有的数据结构的工具，比如高效字符串，vector和map的替代品等等。它还描述了一些快速类型检查工具 isa、cast和dyn_cast），这些你不管在哪都要跑。
  ■如果你不知道你的流程可以做什么，读《编写LLVM流程》(http://llvm.org/docs/WritingAnLLVMPass.html)。不过因为你只是个研究人员而不是浸淫于编译器的大牛，本文的观点可能和这篇教程在一些细节上有所不同。（最紧急的是，别再用基于Makefile的构建系统了。直接开始用CMake构建你的程序吧，读读《“源代码外”指令》(http://llvm.org/docs/CMake.html#cmake-out-of-source-pass)）尽管上面这些是解决流程问题的官方材料，
- 不过在在线浏览LLVM代码时，这个GitHub镜像 (https://github.com/llvm-mirror/llvm)有时会更方便。

写一个流程

使用LLVM来完成高产研究通常意味着你要写一些自定义流程。这一节会指导你构建和运行一个简单的流程来变换你的程序。

框架

我已经准备好了模板仓库 (https://github.com/sampsyo/llvm-pass-skeleton)，里面有些没用的LLVM流程。我推荐先用这个模板。因为如果完全从头开始，配好构建的配置文件可是相当痛苦的事。

首先从GitHub上下载llvm-pass-skeleton仓库 (https://github.com/sampsyo/llvm-pass-skeleton)：

```
$ git clone git@github.com:sampsyo/llvm-pass-skeleton.git
```

主要的工作都是在skeleton/Skeleton.cpp中完成的。把它打开。这里是我们的业务逻辑：

```
virtual bool runOnFunction(Function &F) {
  errs() << "I saw a function called " << F.getName() << "!\n";
  return false;
}
```

LLVM流程有很多种，我们现在用的这一种叫函数流程（function pass）(http://llvm.org/docs/WritingAnLLVMPass.html#the-functionpass-class)（这是一个不错的入手点）。正如你所期望的，LLVM会在编译每个函数的时候先唤起这个方法。现在它所做的只是打印了一下函数名。

细节：

- errs()是一个LLVM提供的C++输出流，我们可以用它来输出到控制台。
- 函数返回false说明它没有改动函数F。之后，如果我们真的变换了程序，我们需要返回一个true。

构建

通过CMake来构建这个流程：

```
$ cd llvm-pass-skeleton
$ mkdir build
$ cd build
$ cmake ..  # Generate the Makefile.
$ make  # Actually build the pass.
```

如果LLVM没有全局安装,你需要告诉CMake LLVM的位置.你可以把环境变量LLVM_DIR的值修改为通往share/llvm/cmake/的路径。比如这是一个使用Homebrew安装LLVM的例子：

```
$ LLVM_DIR=/usr/local/opt/llvm/share/llvm/cmake cmake ..
```

构建流程之后会产生一个库文件，你可以在build/skeleton/libSkeletonPass.so或者类似的地方找到它，具体取决于你的平台。下一步我们载入这个库来在真实的代码中运行这个流程。

运行

想要运行你的新流程，用clang编译你的C代码，同时加上一些奇怪的flag来指明你刚刚编译好的库文件：

```
$ clang -Xclang -load -Xclang build/skeleton/libSkeletonPass.* something
I saw a function called main!
```

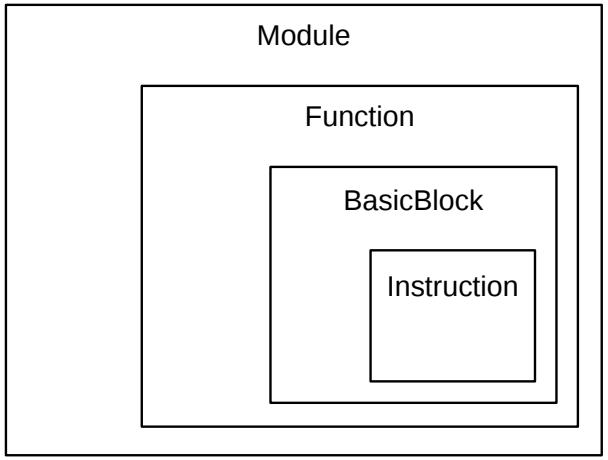
-Xclang -load -Xclang path/to/lib.so这是你在Clang中载入并激活你的流程所用的所有代码。所以当你处理较大的项目的时候，你可以直接把这些参数加到Makefile的CFLAGS里或者你构建系统的对应的地方。

（通过单独调用clang，你也可以每次只跑一个流程。这样需要用LLVM的opt命令。这是官方文档里的合法方式 (http://llvm.org/docs/WritingAnLLVMPass.html#running-a-pass-with-opt)，但在这里我就不赘述了。）

恭喜你，你成功hack了一个编译器！接下来，我们要扩展这个hello world水平的流程，来做一些好玩的事情。

理解LLVM的中间表示

想要使用LLVM里的程序，你需要知道一点中间表示的组织方法。



模块（Module），函数（Function），代码块（BasicBlock），指令（Instruction）
模块 (http://llvm.org/docs/doxygen/html/classllvm_1_1Module.html) 包含了函数 (http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html)，函数又包含了代码块 (http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html)，后者又是由指令 (http://www.llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html) 组成。除了模块以外，所有结构都是从值 (http://www.llvm.org/docs/doxygen/html/classllvm_1_1Value.html)产生而来的。

容器

首先了解一下LLVM程序中最重要的组件：

- 粗略地说，模块表示了一个源文件，或者学术一点讲叫翻译单元。其他所有东西都被包含在模块之中。
- 最值得注意的是，模块容纳了函数，顾名思义，后者就是一段段被命名的可执行代码。（在C++中，函数function和方法method都相应于LLVM中的函数。）
- 除了声明名字和参数之外，函数主要会做为代码块的容器。代码块和它在编译器中的概念差不多，不过目前我们把它看做是一段连续的指令。
- 而说到指令，就是一条单独的代码命令。这一种抽象基本上和RISC机器码是类似的：比如一个指令可能是一次整数加法，可能是一次浮点数除法，也可能是向内存写入。
  大部分LLVM中的内容——包括函数，代码块，指令——都是继承了一个名为值的基类的C++类。值是可以用于计算的任何类型的数据，比如数或者内存地址。全局变量和常数（或者说字面值，立即数，比如5）都是值。

指令

这是一个写成人类可读文本的LLVM中间表示的指令的例子。

```
%5 = add i32 %4, 2
```

这个指令将两个32位整数相加（可以通过类型i32推断出来）。它将4号寄存器（写作%4）中的数和字面值2（写作2）求和，然后放到5号寄存器中。这就是为什么我说LLVM IR读起来像是RISC机器码：我们甚至连术语都是一样的，比如寄存器，不过我们在LLVM里有无限多个寄存器。

在编译器内，这条指令被表示为指令(http://www.llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html)C++类的一个实例。这个对象有一个操作码表示这是一次加法，一个类型，以及一个操作数的列表，其中每个元素都指向另外一个值（Value）对象。在我们的例子中，它指向了一个代表整数2的常量(http://www.llvm.org/docs/doxygen/html/classllvm_1_1Constant.html)对象和一个代表5号寄存器的指令 (http://www.llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html)对象。（因为LLVM IR使用了静态单次分配格式(https://en.wikipedia.org/wiki/Static_single_assignment_form)，寄存器和指令事实上是一个而且是相同的，寄存器号是人为的字面表示。）

另外，如果你想看你自己程序的LLVM IR，你可以直接使用Clang：

```
$ clang -emit-llvm -S -o - something.c
```

查看流程中的IR

让我们回到我们正在做的LLVM流程。我们可以查看所有重要的IR对象，只需要用一个普适而方便的方法：dump()。它会打印出人可读的IR对象的表示。因为我们的流程是处理函数的，所以我们用它来迭代函数里所有的代码块，然后是每个代码块的指令集。

下面是代码。你可以通过在llvm-pass-skeleton代码库中切换到containers分支(https://github.com/sampsyo/llvm-pass-skeleton/tree/containers)来获得代码。

```
errs() << "Function body:\n";
F.dump();
for (auto& B : F) {
  errs() << "Basic block:\n";
  B.dump();
  for (auto& I : B) {
    errs() << "Instruction: ";
    I.dump();
  }
}
```

使用C++ 11里的auto类型和foreach语法可以方便地在LLVM IR的继承结构里探索。

如果你重新构建流程并通过它再跑程序，你可以看到很多IR被切分开输出，正如我们遍历它那样。

做些更有趣的事

当你在找寻程序中的一些模式，并有选择地修改它们时，LLVM的魔力真正展现了出来。这里是一个简单的例子：把函数里第一个二元操作符（比如+，-）改成乘号。听上去很有用对吧？

下面是代码。这个版本的代码，和一个可以试着跑的示例程序一起，放在了llvm-pass-skeleton仓库的 mutate分支 (https://github.com/sampsyo/llvm-pass-skeleton/tree/mutate)。

```
for (auto& B : F) {
  for (auto& I : B) {
    if (auto* op = dyn_cast<BinaryOperator>(&I)) {
      // Insert at the point where the instruction `op` appears.
      IRBuilder<> builder(op);

      // Make a multiply with the same operands as `op`.
      Value* lhs = op->getOperand(0);
      Value* rhs = op->getOperand(1);
      Value* mul = builder.CreateMul(lhs, rhs);

      // Everywhere the old instruction was used as an operand, use our
      // new multiply instruction instead.
      for (auto& U : op->uses()) {
        User* user = U.getUser();  // A User is anything with operands.
        user->setOperand(U.getOperandNo(), mul);
      }

      // We modified the code.
      return true;
    }
  }
}
```

细节如下：

- dyn_cast<T>(p)构造函数是LLVM类型检查工具(http://llvm.org/docs/ProgrammersManual.html#isa)的应用。使用了LLVM代码的一些惯例,使得动态类型检查更高效，因为编译器总要用它们。具体来说，如果I不是“二元操作符”，这个构造函数返回一个空指针，就可以完美应付很多特殊情况（比如这个）。
- IRBuilder用于构造代码。它有一百万种方法来创建任何你可能想要的指令。
- 为把新指令缝进代码里，我们需要找到所有它被使用的地方，然后当做一个参数换进我们的指令里。回忆一下，每个指令都是一个值：在这里，乘法指令被当做另一条指令里的操作数，意味着乘积会成为被传进来的参数。
- 我们其实应该移除旧的指令，不过简明起见我把它略去了。
  现在我们编译一个这样的程序（代码库中的example.c (https://github.com/sampsyo/llvm-pass-skeleton/blob/mutate/example.c)）：

```
#include <stdio.h>
int main(int argc, const char** argv) {
    int num;
    scanf("%i", &num);
    printf("%i\n", num + 2);
    return 0;
}
```

如果用普通的编译器，这个程序的行为和代码并没有什么差别；但我们的插件会让它将输入翻倍而不是加2。

```
$ cc example.c
$ ./a.out
10
12
$ clang -Xclang -load -Xclang build/skeleton/libSkeletonPass.so example.
$ ./a.out
10
20
```

很神奇吧！

链接动态库

如果你想调整代码做一些大动作，用IRBuilder (http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html)来生成LLVM指令可能就比较痛苦了。你可能需要写一个C语言的运行时行为，然后把它链接到你正在编译的程序上。这一节将会给你展示如何写一个运行时库，它可以将所有二元操作的结果记录下来，而不仅仅是闷声修改值。

这里是LLVM流程的代码，也可以在llvm-pass-skeleton代码库的rtlib分支 (https://github.com/sampsyo/llvm-pass-skeleton/tree/rtlib)找到它。

```
// Get the function to call from our runtime library.
LLVMContext& Ctx = F.getContext();
Constant* logFunc = F.getParent()->getOrInsertFunction(
  "logop", Type::getVoidTy(Ctx), Type::getInt32Ty(Ctx), NULL
);

for (auto& B : F) {
  for (auto& I : B) {
    if (auto* op = dyn_cast<BinaryOperator>(&I)) {
      // Insert *after* `op`.
      IRBuilder<> builder(op);
      builder.SetInsertPoint(&B, ++builder.GetInsertPoint());

      // Insert a call to our function.
      Value* args[] = {op};
      builder.CreateCall(logFunc, args);

      return true;
    }
  }
}
```

你需要的工具包括Module::getOrInsertFunction (http://llvm.org/docs/doxygen/html/classllvm_1_1Module.html#a66057011b4f824c8a8d04de9697c194a) 和IRBuilder::CreateCall (http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html#aa6912a2a8a62dbd8706ec00df02c4b8a)。前者给你的运行时函数logop增加了一个声明（类似于在C程序中声明void logop(int i);而不提供实现）。相应的函数体可以在定义了logop函数的运行时库（代码库中的rtlib.c (https://github.com/sampsyo/llvm-pass-skeleton/blob/rtlib/rtlib.c)）找到。

```
#include <stdio.h>
void logop(int i) {
  printf("computed: %i\n", i);
}
```

要运行这个程序，你需要链接你的运行时库：

```
$ cc -c rtlib.c
$ clang -Xclang -load -Xclang build/skeleton/libSkeletonPass.so -c examp
$ cc example.o rtlib.o
$ ./a.out
12
computed: 14
14
```

如果你希望的话，你也可以在编译成机器码之前就缝合程序和运行时库。llvm-link工具——你可以把它简单看做IR层面的ld的等价工具，可以帮助你完成这项工作。

注记（Annotation）

大部分工程最终是要和开发者进行交互的。你会希望有一套*注记（annotations）*，来帮助你从程序里传递信息给LLVM流程。这里有一些构造注记系统的方法：

- 一个实用而取巧的方法是使用*魔法函数*。先在一个头文件里声明一些空函数，用一些奇怪的、基本是独特的名字命名。在源代码中引入这个头文件，然后调用这些什么都没有做的函数。然后，在你的流程里，查找唤起了函数的CallInst指令(http://llvm.org/docs/doxygen/html/classllvm_1_1CallInst.html)，然后利用它们去触发你真正要做的“魔法”。比如说，你可能想调用__enable_instrumentation()和__disable_instrumentation()，让程序将代码改写限制在某些具体的区域。
- 如果想让程序员给函数或者变量声明加记号，Clang的__attribute__((annotate("foo")))语法会发射一个元数据 (http://llvm.org/docs/LangRef.html#metadata)和任意字符串，可以在流程中处理它。Brandon Holt（又是他）有篇文章(http://homes.cs.washington.edu/~bholt/posts/llvm-quick-tricks.html)讲解了这个技术的背景。如果你想标记一些表达式，而非声明，一个没有文档，同时很不幸受限了的__builtin_annotation(e, "foo")内建方法 (https://github.com/llvm-mirror/clang/blob/master/test/Sema/annotate.c)可能会有用。
- 可以自由修改Clang使它可以翻译你的新语法。不过我不推荐这个。
- 如果你需要标记类型——我相信大家经常没意识到就这么做了——我开发了一个名为Quala (https://github.com/sampsyo/quala)的系统。它给Clang打了补丁，以支持自定义的类型检查和可插拔的类型系统，到Java的JSR-308 (http://types.cs.washington.edu/jsr308/)。如果你对这个项目感兴趣，并且想合作，请联系我。
  我希望能在以后的文章里展开讨论这些技术。

其他

LLVM非常庞大。下面是一些我没讲到的话题：

- 使用LLVM中的一大批古典编译器分析；
- 通过hack后端来生成任意的特殊机器指令（架构师们经常想这么干）；
- 利用debug info (http://llvm.org/docs/SourceLevelDebugging.html)连接源代码中的行和列到IR中的每一处；
- 开发[Clang前端插件]。(http://clang.llvm.org/docs/ClangPlugins.html (http://clang.llvm.org/docs/ClangPlugins.html))
  我希望我给你讲了足够的背景来支持你完成一个好项目了。探索构建去吧！如果这篇文章对你帮助，也请让我知道 (mailto:asampson@cornell.edu)。

感谢UW的架构与系统组，围观了我的这篇文章并且提了很多很赞的问题。

以及感谢以下的读者：

- Emery Berger (http://emeryberger.com/)指出了动态二进制分析工具，比如Pin，仍然是你在观察系统结构中具体内容（比如寄存器，内存继承和指令编码等）的好帮手；
- Brandon Holt (http://homes.cs.washington.edu/~bholt/)发了一篇《LLVM debug 技巧》(http://homes.cs.washington.edu/~bholt/posts/llvm-debugging.html)，包括如何用GraphViz绘制控制流图；
- John Regehr (http://www.cs.utah.edu/~regehr/)在评论中提到把软件搭在LLVM上的缺点：API不稳定性。LLVM内部几乎每版都要大换，所以你需要不断维护你的项目。Alex Bradbury (http://asbradbury.org/)的LLVM周报 (http://llvmweekly.org/)是个跟进LLVM生态圈的好资源。

原文：http://adriansampson.net/blog/llvm.html (http://adriansampson.net/blog/llvm.html) 作者： Adrian Sampson
译文：http://geek.csdn.net/news/detail/37785 (http://geek.csdn.net/news/detail/37785) 译者： 张洵恺

扩展阅读

LLVM 入门简单教程 (/lib/view/open1353806821527.html)
LLVM 的 Java 版：JLLVM (/lib/view/open1350951552758.html)
LLVM 的 Erlang 支持：ErLLVM (/lib/view/open1395907747209.html)
Objective-C在LLVM 3.1中的新特性 (/lib/view/open1339334784678.html)
谈Objective-C block的实现 (/lib/view/open1460869392348.html)

为您推荐

用python实现一个抓取腾讯电影的爬虫 (/lib/view/open1376730868131.html)
百度上传控件webuploader的基本用法 (/lib/view/open1421293847328.html)
velocity语法教程 (/lib/view/open1395753470025.html)