使用 LLVM 框架创建有效的编译器，第 2 部分

使用 clang 预处理 C/C++ 代码

Arpan Sen

2012 年 7 月 31 日发布

使用 LLVM 框架创建一个工作编译器，第 1 部分 探讨了 LLVM 中间表示 (IR)。您手动创建了一个 “Hello World” 测试程序；了解了 LLVM 的一些细微差别（如类型转换）；并使用 LLVM 应用程序编程接口 (API) 创建了相同的程序。在这一过程中，您还了解到一些 LLVM 工具，如 `llc` 和 `lli`，并了解了如何使用 llvm-gcc 为您发出 LLVM IR。本文是系列文章的第二篇也是最后一篇，探讨了可以与 LLVM 结合使用的其他一些炫酷功能。具体而言，本文将介绍代码测试，即向生成的最终可执行的代码添加信息。本文还简单介绍了 clang，这是 LLVM 的前端，用于支持 `C`、`C++` 和 Objective-C。您可以使用 clang API 对 `C/C++` 代码进行预处理并生成一个抽象语法树 (AST)。

本系列的其他文章

查看 使用 LLVM 框架创建有效的编译器 系列中的更多文章。

LLVM 阶段

要注意的是 LLVM 为您提供了使用最少量的代码创建实用阶段 (utility pass) 的功能。例如，如果不希望使用 “hello” 作为函数名称的开头，那么可以使用一个实用阶段来实现这个目的。

了解 LLVM opt 工具

LLVM 阶段

从 opt 的手册页中可以看到，“opt 命令是模块化的 LLVM 优化器和分析器”。一旦您的代码支持定制阶段，您将使用 opt 把代码编译为一个共享库并对其进行加载。如果您的 LLVM 安装进展顺利，那么 opt 应该已经位于您的系统中。opt 命令接受 LLVM IR（扩展名为 .ll）和 LLVM 位码格式（扩展名为 .bc），可以生成 LLVM IR 或位码格式的输出。下面展示了如何使用 opt 加载您的定制共享库：

```
1 tintin# opt -load=mycustom_pass.so -help -S
```

还需注意，从命令行运行 opt -help 会生成一个 LLVM 将要执行的阶段的细目清单。对 help 使用 load 选项将生成一条帮助消息，其中包括有关定制阶段的信息。

创建定制的 LLVM 阶段

您需要在 Pass.h 文件中声明 LLVM 阶段，该文件在我的系统中被安装到 /usr/include/llvm 下。该文件将各个阶段的接口定义为 Pass 类的一部分。各个阶段的类型都从 Pass 中派生，也在该文件中进行了声明。阶段类型包括：

- BasicBlockPass 类。用于实现本地优化，优化通常每次针对一个基本块或指令运行

- FunctionPass 类。用于全局优化，每次执行一个功能

由于您打算创建一个阶段，该阶段把绝对以 "Hello" 开头的函数名，因此需要通过从 FunctionPass 派生来创建自己的阶段。从 Pass.h 中复制 清单 1 中的代码。

清单 1. 覆盖 FunctionPass 中的 runOnFunction 类

```
Class FunctionPass : public Pass {
  /// explicit FunctionPass(char &pid) : Pass(PT_Function, pid) {}
  /// runOnFunction - Virtual method overridden by subclasses to do the
  /// per-function processing of the pass.
  ///
  virtual bool runOnFunction(Function &F) = 0;
  /// …
};
```

同样，BasicBlockPass 类声明了一个 runOnBasicBlock，而 ModulePass 类声明了 runOnModule 纯虚拟方法。子类需要为虚拟方法提供一个定义。

返回到 清单 1 中的 runOnFunction 方法，您将看到输出为类型 Function 的对象。深入钻研 /usr/include/llvm/Function.h 文件，就会很容易发现 LLVM 使用 Function 类封装了一个 C/C++ 函数的功能。而 Function 派生自 Value.h 中定义的 Value 类，并支持 getName 方法。清单 2 显示了代码。

清单 2. 创建一个定制 LLVM 阶段

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
class TestClass : public llvm::FunctionPass {
public:
virtual bool runOnFunction(llvm::Function &F)
  {
    if (F.getName().startswith("hello"))
```

```
 8       {
 9         std::cout << "Function name starts with hello\n";
10       }
```


内容

清单 2 中的代码遗漏了两个重要的细节：

概览

- FunctionPass 构造函数需要一个 char，用于在 LLVM 内部使用。LLVM 使用 char 的地址，因此您可以使用任何内容对它进行初始化。

LLVM 阶段

- 您需要通过某种方式让 LLVM 系统理解您所创建的类是一个新阶段。这正是 RegisterPass LLVM 模板发挥作用的地方。您在 PassSupport.h 头文件中声明了 RegisterPass 模板；该文件包含在 Pass.h 中，因此无需额外的标头。

clang 简介

预处理 C 文件

清单 3 展示了完整的代码。

创建一个解析树

清单 3. 注册 LLVM Function 阶段

```
 1  class TestClass : public llvm::FunctionPass
 2  {
 3  public:
 4    TestClass() : llvm::FunctionPass(TestClass::ID) { }
 5    virtual bool runOnFunction(llvm::Function &F) {
 6      if (F.getName().startswith("hello")) {
 7        std::cout << "Function name starts with hello\n";
 8      }
 9      return false;
10    }
11    static char ID; // could be a global too
12  };
13  char TestClass::ID = 'a';
14  static llvm::RegisterPass<TestClass> global_("test_llvm", "test llvm", false, false);
```

RegisterPass 模板中的参数 template 是将要在命令行中与 opt 一起使用的阶段的名称。也就是说，您现在所需做的就是在 清单 3 中的代码之外创建一个共享库，然后运行 opt 来加载该库，之后是使用 RegisterPass 注册的命令的名称（在本例中为

清单 4. 运行定制阶段

```
bash$ g++ -c pass.cpp -I/usr/local/include `llvm-config --cxxflags`
bash$ g++ -shared -o pass.so pass.o -L/usr/local/lib `llvm-config --ldflags -libs`
bash$ opt -load=./pass.so –test_llvm < test.bc
```

LLVM 阶段

现在让我们了解另一个工具（LLVM 后端的前端）：clang。

clang 简介

LLVM 拥有自己的前端：名为 *clang* 的一种工具（恰如其分）。Clang 是一种功能强大的 C/C++/Objective-C 编译器，其编译速度可以媲美甚至超过 GNU Compiler Collection (GCC) 工具（参见 参考资料 中的链接，获取更多信息）。更重要的是，clang 拥有一个可修改的代码基，可以轻松实现定制扩展。与在 使用 LLVM 框架创建一个工作编译器，第 1 部分 中对定制插件使用 LLVM 后端 API 的方式非常类似，本文将对 LLVM 前端使用该 API 并开发一些小的应用程序来实现预处理和解析功能。

开始之前的注意事项

Clang 目前正在开发阶段，和相同规模的任何项目一样，项目文档通常要在代码基完成之后才能就绪。因此，最好的方法是查看开发人员邮件列表（参见 参考资料 中的链接）。您可能希望构建并安装 clang 源，为此，您需要执行 clang 起步指南（参见 参考资料）中的说明。注意，要安装到默认的系统文件夹，您需要在构建完成后发出 make install 命令。本文后面的内容将假设 clang 标头和库分别位

常见的 clang 类

您需要熟悉一些最常见的 clang 类：

- CompilerInstance

- Preprocessor
- Fil..

于类似于 /usr/local/include 和 /usr/local/lib 的系统文件夹中。

- DiagnosticsEngine
- LangOptions
- TargetInfo
- ASTConsumer
- Sema

ParseAST 也许是最重要的 clang 方法。

稍后将详细介绍 ParseAST 方法。

要实现所有实用的用途，考虑使用适当的 CompilerInstance 编译器。它提供了接口，管理对 AST 的访问，对输入源进行预处理，而且维护目标信息。典型的应用程序需要创建 CompilerInstance 对象来完成有用的功能。清单 5 展示了 CompilerInstance.h 头文件的大致内容。

清单 5. CompilerInstance 类

```
class CompilerInstance : public ModuleLoader {
  /// The options used in this compiler instance.
  llvm::IntrusiveRefCntPtr<CompilerInvocation> Invocation;
  /// The diagnostics engine instance.
  llvm::IntrusiveRefCntPtr<DiagnosticsEngine> Diagnostics;
  /// The target being compiled for.
  llvm::IntrusiveRefCntPtr<TargetInfo> Target;
  /// The file manager.
  llvm::IntrusiveRefCntPtr<FileManager> FileMgr;
  /// The source manager.
  llvm::IntrusiveRefCntPtr<SourceManager> SourceMgr;
  /// The preprocessor.
```

```
13    llvm::IntrusiveRefCntPtr<Preprocessor> PP;
14    /// The AST context.
15    llvm::IntrusiveRefCntPtr<ASTContext> Context;
```



```
19    OwningPtr<Sema> TheSema;
20   //… the list continues
21  };
```

概览

预处理 C 文件

LLVM 阶段

clang 简介

在 clang 中，至少可以使用两种方法创建一个预处理器对象：

预处理 C 文件

- 直接实例化一个 Preprocessor 对象

创建一个解析树

- 使用 CompilerInstance 类创建一个 Preprocessor 对象

结束语

让我们首先使用后一种方法。

相关主题

评论

使用 Helper 和实用工具类实现预处理功能

单独使用 Preprocessor 不会有太大的帮助：您需要 FileManager 和 SourceManager 类来读取文件并跟踪源位置，实现故障诊断。FileManager 类支持文件系统查找、文件系统缓存和目录搜索。查看 FileEntry 类，它为一个源文件定义了 clang 抽象。清单 6 提供了 FileManager.h 头文件的一个摘要。

清单 6. clang FileManager 类

```
1  class FileManager : public llvm::RefCountedBase<FileManager> {
2    FileSystemOptions FileSystemOpts;
```

```
 /// \brief The virtual directories that we have allocated.  For each
/// virtual file (e.g. foo/bar/baz.cpp), we add all of its parent
/// directories (foo/ and foo/bar/) here.
```



```
/// NextFileUID - Each FileEntry we create is assigned a unique ID #.
 unsigned NextFileUID;
 // Statistics.
 unsigned NumDirLookups, NumFileLookups;
 unsigned NumDirCacheMisses, NumFileCacheMisses;
// …
 // Caching.
 OwningPtr<FileSystemStatCache> StatCache;
```

clang 简介

SourceManager 类通常用来查询 SourceLocation 对象。在 SourceManager.h 头文件中，清单 7 提供了有关 SourceLocation 对象的信息。

创建一个解析树

清单 7. 理解 SourceLocation

```
/// There are three different types of locations in a file: a spelling
/// location, an expansion location, and a presumed location.
///
/// Given an example of:
/// #define min(x, y) x < y ? x : y
///
/// and then later on a use of min:
/// #line 17
/// return min(a, b);
///
/// The expansion location is the line in the source code where the macro
/// was expanded (the return statement), the spelling location is the
/// location in the source where the macro was originally defined,
/// and the presumed location is where the line directive states that
/// the line is 17, or any other line.
```

很明显，SourceManager 取决于底层的 FileManager；事实上，SourceManager 类构造函数接受一个 FileManager 类作为输入参数。最后，您需要跟踪处理源代码时可能出现的错误并进行报告。您可以使用 DiagnosticsEngine 类完成这项工作。和

- 独立创建所有必需的对象

内容使用 CompilerInstance 完成所有工作

概览

使我们使用后一种方法。清单 8 显示了 Preprocessor 的代码；其他任何事情之前已经解释过了。

LLVM 阶段

清单 8. 使用 clang API 创建一个预处理器

clang 简介

```
1  using namespace clang;
2  int main()
3  {
4      CompilerInstance ci;
5      ci.createDiagnostics(0,NULL); // create DiagnosticsEngine
6      ci.createFileManager();  // create FileManager
7      ci.createSourceManager(ci.getFileManager()); // create SourceManager
8      ci.createPreprocessor();  // create Preprocessor
9      const FileEntry *pFile = ci.getFileManager().getFile("hello.c");
10     ci.getSourceManager().createMainFileID(pFile);
11     ci.getPreprocessor().EnterMainSourceFile();
12     ci.getDiagnosticClient().BeginSourceFile(ci.getLangOpts(), &ci.getPreprocessor());
13     Token tok;
14     do {
15         ci.getPreprocessor().Lex(tok);
16         if( ci.getDiagnostics().hasErrorOccurred())
17             break;
18         ci.getPreprocessor().DumpToken(tok);
19         std::cerr << std::endl;
20     } while ( tok.isNot(clang::tok::eof));
21     ci.getDiagnosticClient().EndSourceFile();
22 }
```

清单 8 使用 CompilerInstance 类依次创建 DiagnosticsEngine（ci.createDiagnostics 方法调用）和 FileManager（ci.createFileManager 和 ci.CreateSourceManager）。使用 FileEntry 完成文件关联后，继续处理源文件中

要编译并运行 清单 8 中的代码，使用 清单 9 中的 makefile（针对您的 clang 和 LLVM 安装文件夹进行了相应调整）。主要想法是使用 llvm-config 工具提供任何必需的 LLVM（包含路径和库）：您永远不应尝试将这些链接传递到 g++ 命令行。

内容
概览

清单 9. 用于构建预处理器代码的 Makefile

```
CXX := g++
RTTIFLAG := -fno-rtti
CXXFLAGS := $(shell llvm-config --cxxflags) $(RTTIFLAG)
LLVMLDFLAGS := $(shell llvm-config --ldflags --libs)
DDD := $(shell echo $(LLVMLDFLAGS))
SOURCES = main.cpp
OBJECTS = $(SOURCES:.cpp=.o)
EXES = $(OBJECTS:.o=)
CLANGLIBS = \
    -L /usr/local/lib \
    -lclangFrontend \
    -lclangParse \
    -lclangSema \
    -lclangAnalysis \
    -lclangAST \
    -lclangLex \
    -lclangBasic \
    -lclangDriver \
    -lclangSerialization \
    -lLLVMMC \
    -lLLVMSupport \
all: $(OBJECTS) $(EXES)
%: %.o
        $(CXX) -o $@ $< $(CLANGLIBS) $(LLVMLDFLAGS)
```

编译并运行以上代码后，您应当获得 清单 10 中的输出。

清单 10. 运行清单 7 中的代码时发生崩溃

```
1 | Assertion failed: (Target && "Compiler instance has no target!")
5 |    line 294.
6 | Abort trap: 6
```


在这里，您遗漏了 CompilerInstance 设置的最后一部分：即编译代码所针对的目标平台。这里是 TargetInfo 和 TargetOptions 类发挥作用的地方。根据 clang 标头 TargetInfo.h，TargetInfo 类存储有关代码生成的目标系统的所需信息，并且必须在编译或预处理之前创建。和预期的一样，TargetInfo 包含有关整数和浮动宽度、对齐等信息。清单 11 提供了 TargetInfo.h 头文件的摘要。

清单 11. Clang TargetInfo 类

```
1  class TargetInfo : public llvm::RefCountedBase<TargetInfo> {
2    llvm::Triple Triple;
3  protected:
4    bool BigEndian;
5    unsigned char PointerWidth, PointerAlign;
6    unsigned char IntWidth, IntAlign;
7    unsigned char HalfWidth, HalfAlign;
8    unsigned char FloatWidth, FloatAlign;
9    unsigned char DoubleWidth, DoubleAlign;
10   unsigned char LongDoubleWidth, LongDoubleAlign;
11   // …
```

TargetInfo 类使用两个参数实现初始化：DiagnosticsEngine 和 TargetOptions。在这两个参数中，对于当前平台，后者必须将 Triple 字符串设置为相应的值。LLVM 此时将发挥作用。清单 12 显示了对 清单 9 所附加的可以使预处理器工作的内容。

清单 12. 为编译器设置目标选项

```
1 | int main()
```

```
2  {
3      CompilerInstance ci;
4      ci.createDiagnostics(0,NULL);
```



```
8      // create TargetInfo
9      TargetInfo *pti = TargetInfo::CreateTargetInfo(ci.getDiagnostics(), to);
10     ci.setTarget(pti);
11     // rest of the code same as in Listing 9…
12     ci.createFileManager();
13     // …
```

LLVM 阶段

就这么简单。运行代码并观察简单的 hello.c 测试的输出：

clang 简介

```
1  #include <stdio.h>
2  int main() {  printf("hello world!\n"); }
```

创建一个解析器

清单 13 展示了部分预处理器输出。

结束语

清单 13. 预处理器输出（部分）

```
1  typedef 'typedef'
2  struct 'struct'
3  identifier '__va_list_tag'
4  l_brace '{'
5  unsigned 'unsigned'
6  identifier 'gp_offset'
7  semi ';'
8  unsigned 'unsigned'
9  identifier 'fp_offset'
10 semi ';'
11 void 'void'
12 star '*'
13 identifier 'overflow_arg_area'
14 semi ';'
15 void 'void'
```

```
star '*'
identifier 'reg_save_area'
semi ';'
```



```
identifier '__va_list_tag'
identifier '__builtin_va_list'
l_square '['
numeric_constant '1'
r_square ']'
semi ';'
```


手动创建一个 Preprocessor 对象

清单 14. 构造一个 Preprocessor 对象

```
Preprocessor(DiagnosticsEngine &diags, LangOptions &opts,
             const TargetInfo *target,
             SourceManager &SM, HeaderSearch &Headers,
             ModuleLoader &TheModuleLoader,
             IdentifierInfoLookup *IILookup = 0,
             bool OwnsHeaderSearch = false,
             bool DelayInitialization = false);
```

查看该构造函数，显然，要想让这个预处理器工作，您还需要创建 6 个不同的对象。您已经了解了 DiagnosticsEngine、TargetInfo 和 SourceManager。CompilerInstance 派生自 ModuleLoader。因此您必须创建两个新的对象，一个用于 LangOptions，另一个用于 HeaderSearch。LangOptions 类使您编译一组 C/C++ 方言，包括 C99、C11 和 C++0x。

参考 LangOptions.h 和 LangOptions.def 标头，获取更多信息。最后，HeaderSearch 类存储目录的 std::vector，用于在其他对象中搜索功能。清单 15 显示了 Preprocessor 的代码。


```
1  using namespace clang;
2  int main()  {
3      DiagnosticOptions diagnosticOptions;
4      TextDiagnosticPrinter *printer =
5        new TextDiagnosticPrinter(llvm::outs(), diagnosticOptions);
6      llvm::IntrusiveRefCntPtr<clang::DiagnosticIDs> diagIDs;
7      DiagnosticsEngine diagnostics(diagIDs, printer);
8      LangOptions langOpts;
9      clang::TargetOptions to;
10     to.Triple = llvm::sys::getDefaultTargetTriple();
11     TargetInfo *pti = TargetInfo::CreateTargetInfo(diagnostics, to);
12     FileSystemOptions fsopts;
13     FileManager fileManager(fsopts);
14     SourceManager sourceManager(diagnostics, fileManager);
15     HeaderSearch headerSearch(fileManager, diagnostics, langOpts, pti);
16     CompilerInstance ci;
17     Preprocessor preprocessor(diagnostics, langOpts, pti,
18       sourceManager, headerSearch, ci);
19     const FileEntry *pFile = fileManager.getFile("test.c");
20     sourceManager.createMainFileID(pFile);
21     preprocessor.EnterMainSourceFile();
22     printer->BeginSourceFile(langOpts, &preprocessor);
23     // … similar to Listing 8 here on
24 }
```

对于 清单 15 中的代码，需要注意以下几点：

- 您没有初始化 HeaderSearch 并使它指向任何特定的目录。但是您应当这样做。
- clang API 要求在堆 (heap) 上分配 TextDiagnosticPrinter。在栈 (stack) 上分配会引起崩溃。

- 您还不能处理掉 CompilerInstance。总之是因为您正在使用 CompilerInstance，那么为什么还要费心去手动创建它而不是更舒适地使用 clang API 呢？

语言选择：C++

内容

您目前为止一直使用的是 C 测试代码：那么使用一些 C++ 代码如何？向 清单 15 中的代码添加 `langOpts.CPlusPlus = 1;`，然后尝试使用 清单 16 中的测试代码。

概览

LLVM 阶段

清单 16. 对预处理器使用 C++ 测试代码

clang 简介

```
1  template <typename T, int n>
2  struct s {
3    T array[n];
4  };
5  int main() {
6    s<int, 20> var;
7  }
```

相关主题

清单 17 展示了程序的部分输出。

评论

清单 17. 清单 16 中代码的部分预处理器输出

```
 1  identifier 'template'
 2  less '<'
 3  identifier 'typename'
 4  identifier 'T'
 5  comma ','
 6  int 'int'
 7  identifier 'n'
 8  greater '>'
 9  struct 'struct'
10  identifier 's'
11  l_brace '{'
```

```
12 identifier 'T'
13 identifier 'array'
14 l_square '['
```



```
18 r_brace '}'
19 semi ';'
20 int 'int'
21 identifier 'main'
22 l_paren '('
23 r_paren ')'
```

LLVM 阶段

创建一个解析树

clang 阶段

预处理 C 文件

clang/Parse/ParseAST.h 中定义的 ParseAST 方法是 clang 提供的重要方法之一。以下是从 ParseAST.h 复制的一个例程声明：

创建一个解析树

```
1 void ParseAST(Preprocessor &pp, ASTConsumer *C,
2       ASTContext &Ctx, bool PrintStats = false,
3       TranslationUnitKind TUKind = TU_Complete,
4       CodeCompleteConsumer *CompletionConsumer = 0);
```

评论

ASTConsumer 为您提供了一个抽象接口，可以从该接口进行派生。这样做非常合适，因为不同的客户端很可能通过不同的方式转储或处理 AST。您的客户端代码将派生自 ASTConsumer。ASTContext 类存储有关类型声明的信息和其他信息。最简单的尝试就是使用 clang ASTConsumer API 在您的代码中输出一个全局变量列表。许多技术公司就全局变量在 C++ 代码中的使用有非常严格的要求，这应当作为创建定制 lint 工具的出发点。清单 18 中提供了定制 consumer 的代码。

清单 18. 定制 AST consumer 类

```
1 class CustomASTConsumer : public ASTConsumer {
2 public:
3  CustomASTConsumer () :  ASTConsumer() { }
```

```
    virtual ~ CustomASTConsumer () { }
    virtual bool HandleTopLevelDecl(DeclGroupRef decls)
    {
```



```
            clang::VarDecl *vd = llvm::dyn_cast<clang::VarDecl>(*it);
            if(vd)
               std::cout << vd->getDeclName().getAsString() << std::endl;;
        }
        return true;
    }
};
```

您将使用自己的版本覆盖 HandleTopLevelDecl 方法（最初在 ASTConsumer 中提供）。Clang 将全局变量列表传递给您；您对该列表进行迭代并输出变量名称。清单 19 摘录自 ASTConsumer.h，显示了客户端 consumer 代码可以覆盖的一些其他方法。

清单 19. 其他一些可以在客户端代码中覆盖的方法

```
/// HandleInterestingDecl - Handle the specified interesting declaration. This
/// is called by the AST reader when deserializing things that might interest
/// the consumer. The default implementation forwards to HandleTopLevelDecl.
virtual void HandleInterestingDecl(DeclGroupRef D);

/// HandleTranslationUnit - This method is called when the ASTs for entire
/// translation unit have been parsed.
virtual void HandleTranslationUnit(ASTContext &Ctx) {}

/// HandleTagDeclDefinition - This callback is invoked each time a TagDecl
/// (e.g. struct, union, enum, class) is completed.  This allows the client to
/// hack on the type, which can occur at any point in the file (because these
/// can be defined in declspecs).
virtual void HandleTagDeclDefinition(TagDecl *D) {}

/// Note that at this point it does not have a body, its body is
  /// instantiated at the end of the translation unit and passed to
  /// HandleTopLevelDecl.
  virtual void HandleCXXImplicitFunctionInstantiation(FunctionDecl *D) {}
```

最后，清单 20 显示了您开发的定制 AST consumer 类的实际客户端代码。


```
    CompilerInstance ci;
    ci.createDiagnostics(0,NULL);
    TargetOptions to;
    to.Triple = llvm::sys::getDefaultTargetTriple();
    TargetInfo *tin = TargetInfo::CreateTargetInfo(ci.getDiagnostics(), to);
    ci.setTarget(tin);
    ci.createFileManager();
    ci.createSourceManager(ci.getFileManager());
    ci.createPreprocessor();
    ci.createASTContext();
    CustomASTConsumer *astConsumer = new CustomASTConsumer ();
    ci.setASTConsumer(astConsumer);
    const FileEntry *file = ci.getFileManager().getFile("hello.c");
    ci.getSourceManager().createMainFileID(file);
    ci.getDiagnosticClient().BeginSourceFile(
       ci.getLangOpts(), &ci.getPreprocessor());
    clang::ParseAST(ci.getPreprocessor(), astConsumer, ci.getASTContext());
    ci.getDiagnosticClient().EndSourceFile();
    return 0;
}
```

评论

结束语

这篇两部分的系列文章涵盖了大量内容：它探讨了 LLVM IR，提供了通过手动创建和 LLVM API 生成 IR 的方法，展示了如何为 LLVM 后端创建一个定制插件，以及解释了 LLVM 前端及其丰富的标头集。您还了解了如何使用该前端进行预处理和使用 AST。在计算史上，创建一个编译器并进行扩展，特别是针对 C++ 等复杂的语言，看上去是个非常复杂的过程，但是有了 LLVM，一切都变得非常简单。文档工作是 LLVM 和 clang 需要继续加强的部分，但是在此之前，我建议尝试 VIM/doxygen 来浏览这些标头。祝您使用愉快！

本系列的其他文章

查看 使用 LLVM 框架创建有效的编译器 系列中的其他文章。

- 在 使用 LLVM 框架创建一个工作编译器，第 1 部分（Arpan Sen，developerWorks，2012 年 6 月）中了解 LLVM 的基础知识。使用功能强大的 LLVM 编译器基础架构优化用任何语言编写的应用程序。构建一个定制编译器现在变得非常简单！
- 了解有关 LLVM 阶段 的更多信息。
- 获取 clang 开发人员邮件列表。
- 阅读 Getting Started: Building and Running Clang，获取有关构建和安装 clang 的详细信息。
- 参加 官方 LLVM 教程，获取有关 LLVM 的出色介绍。
- 深入钻研 LLVM Programmer's Manual，这是了解 LLVM API 的必不可少的资源。
- 访问 LLVM 项目站点 并下载 最新版本。
- 从 LLVM 站点查找有关 clang 的详细信息。
- 在 developerWorks 中国网站 Linux 技术专区，查找数百篇 how-to 文章和教程，以及下载、讨论论坛和面向 Linux 开发人员和管理员的其他丰富资源。
- developerWorks 中国网站 Web 开发专区 提供了涵盖各种基于 Web 的解决方案的文章。
- IBM 产品评估试用版软件：下载产品试用版，在线试用产品，在云环境中使用产品，或在 IBM SOA 人员沙箱 中花几个小时了解如何高效实现面向服务的架构。
- 随时关注 developerWorks 技术活动和网络广播。
- 访问 developerWorks Open source 专区获得丰富的 how-to 信息、工具和项目更新以及最受欢迎的文章和教程，帮助您用开放源码技术进行开发，并将它们与 IBM 产品结合使用。

评论

添加或订阅评论，请先登录或注册。

内容