LLVM 3.9 介绍

宁宇 (sirning@mail.ustc.edu.cn)

张昱 (yuzhang@ustc.edu.cn)

December 18, 2016

目录

Part I

概要篇

撰写者：宁宇、郭兴、张昱

LLVM 3.9.0 于 2016 年 9 月 2 日发布。本篇概要介绍 LLVM 的安装与使用。

1 LLVM 的安装与使用

1.1 LLVM 的获取

LLVM 可从 http://llvm.org/releases/download.htm 下载, 其中包括 LLVM, Clang,Compiler RT, Polly(多面体优化).

1.2 安装 LLVM 3.9

Overview

- Core Components

  LLVM Suite: 包括各种工具、库、头文件, 包括一个汇编器、反汇编器、bitcode 分析器、bitcode 优化器, 同时包括测试 LLVM tools 和 Clang front end 的测试样例。

  Clang Front End: 将 C/C++ 和 Objective C/C++ 源程序编译为 LLVM bitcode, bitcode 可以进一步用 LLVM suite 中的各种工具来处理。

  Test Suite: 可选, 测试 LLVM 的功能和性能。

- External Projects
  - Clang-tools-extra repository 是一系列基于 Clang 的程序集, 其中包括

    Clang Modernizer 一个代码重构工具, 扫描 C++ 代码并将 old-style constructs 转换为与时俱进的符合更新标准的 constructs。

    Clang Tidy 一个 linter tool 检查代码是否违反了 LLVM 或者 Google 编程标准。

    Modularize 帮助确认哪些 C++ 头文件适合聚集在一起构成一个 module。

    PPTrace 一个用来跟踪 Clang C++ preprocessor 活动的工具。

    Clang-Include-Fixer 帮助管理 #include 预编译指令。
  - Compiler-RT(runtime) project 提供与机器相关的硬件不支持的而代码生成或者其他运行时组建需要的功能, 比如为一个 32 位目标机器生成代码时, 将一个 double 转换为一个 64 位无符号整数会被编译为对运行时函数 ___fixunsdfdi 的调用。此外还包括对 sanitizer runtimes、profile、BlocksRuntime 的支持。
  - DragonEgg project 为 GCC 提供一个插件用 LLVM 的优化器和代码生成器替换 GCC 的优化器和代码生成器, 要求 GCC 版本大于等于 4.5, 支持 x86-32/x86-64 和 ARM 体系结构, 当前完全支持 Ada、C/C++ 和 Fortran, 目标支持所有 GCC 前端支持的语言。
  - LLDB project 提供新一代高性能调试器, LLDB 由一系列高度复用已有的 LLVM 库的可重用的组件构成如 Clang expression parse 和 LLVM disassembler。LLDB 采用新的结构支持多线程调试、更有效地处理调试信息、支持插件形式的功能扩展。

Build with CMake 从 3.9 版本开始; 将不能使用 autoconf build system 编译 LLVM; 推荐使用 CMake 并且要求 CMake 版本至少为 3.4.3, 这里介绍如何使用 CMake 编译 LLVM。

Check/Install CMake CMake 的作用类似于 autoconf, 并不直接编译代码而是为指定的 build tool(如 GNU make) 生成 Makefile。首先使用CMake –version检查是否安装了 CMake 以及版本是否大于等于 3.4.3。如果满足要求, 可以按照下面的步骤安装 CMake-3.6.2

- 从源码编译 CMake-3.6.2。

```
1 wget http://cmake.org/files/v3.6/cmake-3.6.2.tar.gz
```

```
tar xzf cmake-3.6.2.tar.gz
cd cmake-3.6.2
./configure
make
sudo make install
```

configure 默认将 cmake 安装到/usr/local/bin，当然可以给 configure 添加各种选项以实现一次个性化的编译。

- 直接下载 prebuild-binary

```
wget http://cmake.org/files/v3.6/cmake-3.6.2-Linux-x86_64.tar.gz
tar xzf cmake-3.6.2-Linux-x86_64.tar.gz
mv cmake-3.6.2-Linux-x86_64 where-you-want-this-be
set your $PATH env-variable
```

Build LLVM 3.9 with CMake-3.6.2 下面是一些常用的 CMake 选项供用户实现一个自定义的安装。这些选项的使用方式类似: cmake -DOPTION=value llvm-3.9.0.src 或者 cmake -UOPTION llvm-3.9.0.src。其中 OPTION 用下面的选项名替换, -DOPTION 表示启用 OPTION 这个选项, -UOPTION 表示撤销前一次的 cmake -DOPTION，每一次对 cmake 的调用都会记录在 CMakeCache.txt 中。

- CMAKE_BUILD_TYPES: 一个字符串指示安装类型，可选值是"Release"、"Debug"、"RelWithDebInfo" 和"MinSizeRel"。
- CMAKE_INSTALL_PREFIX: 指定安装路径。
- CMAKE_C_FLAGS: 传递给 C 编译器的额外的编译选项。
- CMAKE_CXX_FLAGS: 传递给 C++ 编译器的额外的编译选项。

执行下面的命令行下载源代码并安装 LLVM Suite、Clang、lld、lldb 和 compiler-rt，将 Clang、lld 和 lldb 的代码放在 llvm-3.9.0.src/tools 下；将 compiler-rt 放在 llvm-3.9.0/ projects 下，然后调用 cmake 可以直接将这些附加的工具、库安装。事实上 llvm-3.9.0.src/tools 和 llvm-3.9.0.src/projects 下分别有一个 CMakeList.txt 的文件，从中可以看出安装过程中会搜索 tools 和 projects 文件夹下的哪些包含外部工具源代码的子文件夹并尝试安装这些外部工具。

```
mkdir llvm-src
cd llvm-src
#获取llvm-suite和clang
wget http://llvm.org/releases/3.9.0/llvm-3.9.0.src.tar.xz
wget http://llvm.org/releases/3.9.0/cfe-3.9.0.src.tar.xz
#获取lld、lldb和compiler-rt，lldb的编译可能会遇到一些无法解决的问题
#所以建议不要编译lldb
wget http://llvm.org/releases/3.9.0/lld-3.9.0.src.tar.xz
wget http://llvm.org/releases/3.9.0/compiler-rt-3.9.0.src.tar.xz
#wget http://llvm.org/releases/3.9.0/lldb-3.9.0.src.tar.xz
tar xf llvm-3.9.0.src.tar.xzf
tar cfe-3.9.0.src.tar.xz
tar xf lld-3.9.0.src.tar.xz
tar xf compiler-rt-3.9.0.src.tar.xz
#tar xf lldb-3.9.0.src.tar.xz
mv cfe-3.9.0.src llvm-3.9.0.src/tools/clang
mv lld-3.9.0.src llvm-3.9.0.src/tools/lld
mv compiler-rt-3.9.0 llvm-3.9.0.src/projects/compiler-rt
#mv lldb-3.9.0.src llvm-3.9.0.src/tools/lldb
```

```
20
21 #最好创建一个新的文件夹用于编译,
22 #否则LLVM的源代码会与cmake生成的文件混在一起
23 mkdir build
24 cd build
25 cmake -DCMAKE_INSTALL_PREFIX="WhereYouWantLLVMBe" ../llvm-3.9.0.src
26 cmake --build .
27 cmake --build . --target install
```

Build Errors Encountered

- Could NOT find PythonLibs:sudo apt-get install python2.7-dev
- Could NOT find CURSES:sudo apt-get install libncurses5-dev
- Could NOT find SWIT:sudo apt-get install swig
- histedit.h not found:sudo apt-get install libedit-dev

2 LLVM&Clang Release Notes

下面详细介绍 LLVM 和 Clang 各个版本之间的差异，读者也可以参考下面列出来的 release notes。

- 2015. 9. 2: 发布LLVM3.9.0
- 2016. 3. 8: 发布LLVM3.8.0
- 2015. 9. 1: 发布LLVM3.7.0
- 2015. 2.27: 发布LLVM3.6.0
- 2014. 9. 3: 发布LLVM3.5.0
- 2014. 1. 6: 发布LLVM3.4
- 2013. 6.17: 发布LLVM3.3
- 2012.12.20: 发布LLVM3.2

LLVM3.9. LLVM 3.9 改进了对 libstdc++ ABI 兼容性，支持 OpenMP 4.5 的所有 non-offloading 特性，添加了 clang-include-fixer，极大改进了 ELF 与 lld 的链接，继续改进优化过程，修复 bug。

下面是 general LLVM release notes 的更多细节

- LLVM 现在不能继续使用 autoconf build system 进行编译了，推荐使用 CMake。LLVM 3.9 要求 CMake 3.4.3 或者更高版本。
- 移除了一系列在旧版本中已经不推荐使用的 C API functions。
- 新增了 MemorySSA analysis，期望替代 MemoryDependenceAnalysis。前者在结果和算法效率上都要优于后者。
- llvm.masked.load、llvm.masked.store、llvm.masked.gather、llvm.masked.scatter 被添加到 IR 中以支持向量类型的内存访问。
- 对 ARM、MIPS、PowerPC、X86、AMDGPU 的支持有了改动。

下面是 Clang release notes 的更多细节

- 新增对 C++17 的部分特性的支持。
- 进一步改进 OpenMP 的 codegen。
- Static Analyzer 现在支持 C/C++ 代码中对 MPI API 的错误使用。
- clang-include-fixer 可以自动添加需要的 #include 编译指令。

- clang-tidy's check 中添加了大量新的检查并且修复了 3 个 bug，这算是一个重大改进，之前的版本对 clang-tidy's check 也有改动，但都被忽略不计了。

3.9 版本的 LLD 有了里程碑式的改进，是第一个可以用于实际大规模程序的版本，所以下面是 LLD release notes 的细节。

- Identical Code Folding:LLD 3.9 可以将同样的代码段合并以产生较小的输出文件。
- Symbol Versioning:LLD 3.9 可以连接、产生带有版本靠的符号。
- TLS Relocation Optimizations
- 新增大量链接选项。

LLVM3.8. 这个版本对核心的改动不多，一贯地继续改善优化过程，修复 bug。

下面是 general LLVM release notes 的更多细节

- Windows 用户至少需要再 Windows 7 的环境才能运行 LLVM。不再继续支持 Windows XP 和 Windows Vista。
- 不再推荐使用 autoconf build system，LLVM 3.9 版本将不能使用 autoconf build，现在推荐使用 CMake。
- 为 C API stability guarantees 添加文档。
- 部分 C API functions 有了新版本，不再推荐使用这些函数而是推荐使用新版本。一些已经不推荐使用的 C API functions 被彻底移除。
- 不再在目标文件中生成.data.rel.ro.local 和.data.rel 段。
- 对 ARM 的后端做了修改，对 MIPS、PowerPC、X86、Heaxgon、AVR 的支持有改动。

下面是 Clang release notes 的更多细节

- Clang disgnostics 没有实质的改进。
- 新的编译选项：-ggdb、-glldb、-gsce 可以分别为 gdb、lldb、Sony Computer Entertainment debugger 生成调试信息。为了与 GCC 兼容，这些选项后面可以跟上 0 到 3 的一个数字指示调试信息的级别。-g 会为平台特定的默认调试器产生调试信息。
- 改进了数据对齐。
- 全面支持 OpenMP 3.1 并且默认开启支持，-fopenmp 选项现在会使用 Clang OpenMP library 而不是 GCC OpenMP Library。进一步添加了新的对 OpenMP 4.0/4.5 的部分特性的支持，将继续努力完对 OpenMP 4.5 的支持。对 OpenMP 的 codegen 有了重大改进，可以生成更加稳定和高效的代码。
- Clang 现在有实验版本的对 end-to-end CUDA compilation 的支持
- 对 AST matcher 的函数名做了修改，大量 matcher 受到了影响。
- scan-build 和 scan-view 工具现在被集成到了 Clang 中，使用这些工具可以更方便地运行静态检查器。此外对 C++ lambda 表达式的静态分析有了重大改进，包括过程间的静态 lambda 表达式分析。

LLVM3.7. LLVM3.7 提供了对 OpenMP 3.1 的完全支持，增加了完整性的检查，为 Berkeley Packet Filter 提供了新的后端，支持 On Request Compilation JIT API，增加新的 Clang warnings。此外继续修复 bug，改善优化过程。

下面是 general LLVM release notes 的更多细节。

- 对 MPIS、PowerPC 和 SystemZ target 的支持都有大量改动。

- 新增了名为 On Request Compilation 的 C++ JIT API，3.6 版本的 release notes 中有提到。
- Polly: The Polyhedral Loop Optimizer (一个循环优化的框架，不属于 LLVM 核心) 有了多项修改。

下面是 Clang release notes 的更多细节。

- Clang's diagnostics 可以做一些新的诊断，通过下面-Woptions 的一些列编译选项控制 Clang 是否给出警告。
  - -Wrange-loop-analysis: 分析循环变量的类型和容器的类型，判断当循环访问一个容器时，容器的元素是否被拷贝并给出警告或者建议使用 const reference 以避免拷贝。
  - -Wredundant-move:
  - -Wpessimizing-move:
  - -Winfinite-recursion: 对可能的无限递归函数给出警告。
- 新增编译选项-fsized-deallocation 以支持 C++14 sized deallocation 特性。
- Clang 现在支持与 GCC 兼容的 Profile Guided Optimization 编译选项:-fprofile-generate=<dir>,-fprofile-use=<dir>, -fno-profile-generate 和-fno-profile-use，这些选项的语义与 GCC 是相同的。
- OpenMP 3.1 现在被全面支持，默认不启用支持，使用-fopenmp-libomp 选项启用 OpenMP。此外部分 OpenMP 4.0 版本的特性也被支持。
- Clang 3.7 是最后一个可以在 Windows XP 和 Windows Vista 上运行的 major release。后续的面向 Windows 的 major release 将会至少要求 Windows 7 的环境。

LLVM3.6. 这个版本修复了大量 bug，改善了优化过程，增加了更多对未来 C++17 标准提议的支持，改善了对 Windows 平台的兼容性。

下面是 general LLVM release notes 的更多细节。

- 不再支持 AuroraUX 操作系统。
- 新增对 native object file-based bitcode wrapper format 的支持，现在 LLVM IR 的 bitcode 可以封装在本地目标文件 (如 ELF,COFF) 中，bitcode 必须存在名字为.llvmbc 的段中，这些信息可以对于 LTO(Link Time Optimization) 是有用的。
- 移除了-std-compile-opts 优化选项。
- 移除了 leak detector，因为实践中 asan 和 valgrind 等工具可以发现更多的 bug。
- 增加了对 Win64 unwind informationd 的支持。
- 移除了 PreserveSource linker mode。
- 新增了一个实验性的机制描述可以用于垃圾收集的安全的程序点，然而该机制可能再 3.7 版本中才能被完善。
- 对 MIPS 架构的支持有重大进展，在这一点上 3.6 可称为一个里程碑式的版本。
- 现在可以编译 Linux Kernel，不过需要几个 kernel patches。
- 改进了对 PowerPC 的支持。
- 引入一系列对 Go 语言的 bindings。

下面是 Clang release notes 的更多细节。

- 增强了 Clang diagnostics 的诊断能力: Clang 现在尝试在更多的场合给出有用的建议，可以从更多的错误中恢复。
- -fpic 选项在 PowerPC 上使用 small pic。
- 现在支持在循环前添加 #pragma unroll 编译指令指示循环展开的优化。

- 改进了对 Windows 的支持。
  - 修复大量 bug。
  - 增加更多 MSVC compatibility hacks，使得使用 Active Template Library 或者 Windows Runtime Library 的程序能够正常编译。
  - 增加对 Visual C++ ___super 关键字的支持。
  - 增加对 MSVC ___vectorcall 调用约定的支持，该调用约定可以在 Visual Studio 2015 STL 中被使用。
  - 增加基本的对 COFF 文件中 DWARF 调试信息的支持。
- C/C++ 语言相关
  - Clang 对 C 的默认语言模式由 GNU 扩展的 C99 切换到 GNU 拓展的 C11, 使用-std=gnu99 设置语言模式为 GNU 扩展的 C99。
  - 增加对 C11 标准头文件 stdatomic.h 的支持。

LLVM3.5. 这一版本继续对 LLVM 优化和后端进行持续的性能改进和编译时改进，并支持一些新的目标平台且改进现有的目标平台。

- Clang 可以在 Linux/Sparc64 和 FreeBSD/Sparc64 自托管 (self-host).
- 循环向量化元数据的前缀由llvm.vectorizer 变更为 llvm.loop.vectorizer。此外,llvm.vectorizer.unroll 元数据被重命名为 llvm.loop.interleave.count.
- **后端**
  - 所有后端都使用 MC asm printer, 对非 MC 的支持已经被移除
  - LLVM 假设汇编器支持用于生成调试行号的.loc
  - 所有的内联汇编可以在集成的汇编器生效时被解析，以前这只用于目标文件输出的情况，现在也可以用于汇编输出。集成的汇编器可以使用选项-no-integrated-as 被禁用。
  - llvm-ar 目前像正规的目标文件那样处理 IR 文件, 特别地，会为 IR 文件中定义的符号创建正规的符号表，包括那些在文件内内联的汇编码中的符号。
  - LLVM 使用 cfi 制导来产生绝大多数的栈展开信息。
  - **对 ARM 后端的修改**: 集成汇编器 (IAS) 和 ARM 异常处理 (EHABI) 在 LLVM/Clang 上被缺省地启用。

    IAS 能接收许多 GNU 的扩展和制导指令，以及某些特定的 pre-UAL 指令。尽管 IAS 足以编译一些大项目 (包括绝大多数的 Linux Kernel)，但是仍然存在一些不能做的，这时可以在 Clang 中使用 -fno-integrated-as 选项。

    以前版本中启用 EHABI 所需的选项-arm-enable-ehabi 和 -arm-enable-ehabi-descriptors 被移除。所有的 ARM 代码将发射 EH 展开表、或 CFI 展开 (用于调试和 profiling)，或这两类。为避免运行时的不一致，C 代码也为其他架构 (如 x86_64) 将发射 EH 表。
  - **对 MIPS、AArch64 后端的修改**:

LLVM3.4. 和 LLVM 3.3 相比，改进的地方主要在向量化和后端，概述如下:

前端 Clang 现在支持所有的 C++11 标准 (包括草案), 对 static analyzer 和 clang-format 进行了改进。

优化遍 循环向量化现在在-O2 和 Os 级别就默认启用，而 SLP 向量化则从 O1 就开始默认启用。LLVM 的连接时优化不再是 O4 了，而是-O3 -flto

后端 R600 后端达到产品级别了，X86, SPARC, ARM32, Aarch64 和 SystemZ 做了很多改进。

3.4 是最后一版还可以使用仅实现了 C++'98 标准的编译器可以编译的版本，从 LLVM3.5 开始，需要支持 C++11 的编译器才能编译 LLVM，当然 LLVM 可以自编译。

注：编译 LLVM3.4 中的 lldb3.4 需要 gcc4.8.2(支持 c++11 特征) 以上版本。

LLVM3.3. 为 AArch64 和 AMD R600 GPU 架构增加了新的目标, 增加了对 IBM 的 z/Architecture S390 系统的支持, 对 PowerPC 的后端 (包括对 PowerPC 2.04/2.05/2.06 指令, 以及集成的汇编器的支持) 和 MIPS 目标进行了大调.

LLVM3.3 生成的代码的性能有实质性改进: 这一版本的 自动向量化在多数情况下能产生非常好的代码. 自动向量化在使用-O3 选项时被缺省地启用. 这一版本增加了一种新的超字级并行 (Superword Level Parallelism, SLP) 的向量化部件, 并做了许多改进. 通过性能评测,LLVM3.3 的性能在许多基准程序上优于 LLVM 3.2.

LLVM3.3 也是Clang前端的重要里程碑: 它现在完全支持 C++'11 的全部特性. 迄今为止,Clang 是唯一的支持完整 C++'11 标准的编译器, 包括像 std::regex 这些重要的 C++'11 库函数特征. Clang 支持 Unicode 字符的标识符. Clang 静态分析器支持几种新的检查器, 能跨越 C++ 构造器/析构器的边界执行过程间分析. Clang 还提供C++'11 Migrator工具帮助升级代码以使用 C++'11 的特征, 提供 Clang Format 插件工具插入到 vim 和 emacs 中来自动格式化代码.

Part II

Clang C Preprocessor 简介

3 LLVM C Preprocessor Interface

这一小节主要介绍 LLVM 中与 C 预处理相关的接口。

3.1 构造 Preprocessor

可以通过两种方式构造一个 Preprocessor: 第一种是直接调用 Preprocessor 的构造函数，稍后会看到 Preprocessor 的构造函数要求传入大量参数，因此这样会比较麻烦，但是了解这些繁杂的细节会帮助增进对 LLVM/CLANG 源代码的一些了解；第二种方法是通过 CompilerInstance 这个 LLVM 提供的统一控制编译过程及对象管理的接口来构造 Preprocessor。无论使用哪种方法，构造一个 Preprocessor 的过程都要涉及多个 LLVM/CLANG 类，这里先统一列出下面要涉及到的类以及他们的声明、定义所在的源文件的连接: Preprocessor(.h .cpp)；DiagnosticsEngine、DiagnosticConsumer(.h .cpp)；DiagnosticIDs(.h .cpp)；DiagnosticOptions(.h .cpp)；LangOptions(.h .cpp)；TargetInfo(.h .cpp)；TargetOptions(.h .cpp)；SourceManager(.h .cpp)；FileManager(.h .cpp)；HeaderSearch(.h .cpp)；HeaderSearchOptions(.h .cpp)；HeaderSearchOptions(.h .cpp)。由于有些类是声明、定义在同一个源文件中的，所以上述列举中会出现多个类名由顿号隔开后统一跟上源文件链接的情形，有兴趣的读者可以进一步阅读源文件加深了解，下面将从 Preprocessor 的构造函数出发，介绍上述各个类的作用。

下面给出 Preprocessor 构造函数的签名，并一一介绍其形参列表类类型的作用。

```
Preprocessor(IntrusiveRefCntPtr<PreprocessorOptions> PPOpts,
        DiagnosticsEngine &diags, LangOptions &opts,
        SourceManager &SM, HeaderSearch &Headers,
        ModuleLoader &TheModuleLoader,
        IdentifierInfoLookup *IILookup = nullptr,
        bool OwnsHeaderSearch = false,
        TranslationUnitKind TUKind = TU_Complete);
```

第一个参数类型 IntrusiveRefCntPtr<PreprocessorOptions> 是一个指向 PreprocessorOptions 的智能指针，其中 IntrusiveRefCntPtr 是一个智能指针类，如果我们忽略这个智能指针，那么只剩下一个所有成员都具有 public 访问属性的 PreprocessorOptions，从字面意思上看就能猜到 PreprocessorOptions 包含了与预处理器相关的配置，事实也是这样的，比如其成员

```
unsigned UsePredefines : 1;
//指示了是否使用一些与机器相关的predefines，比如在linux操作系统上可能会有
//#define ___linux，在windows上会有#define ___WIN32等，此外llvm也提供一些
//predefines如#define ___llvm___
```

此外，PreprocessorOptions 还包含了大量 public 成员共同指示 Preprocessor 的工作环境。

第二个参数类型 DiagnosticsEngines 是一个比较复杂的类，负责生成、报告描述编译过程中遇到的错误的消息。这个类的介绍只给出这么一句苍白的概括，因为它牵涉的东西比较多，我还没能理解以至简练、详细地概括它。

LangOptions 的部分源代码 (LangOptions.h) 是用宏定义生成的，LangOptions.h 中定义了生成代码用的宏定义而且其所在目录下有一个 LangOptions.def 文件，LangOptions.def 文件中调用了上述宏定义，然后在 LangOptions.h 的合适位置 #include “LangOptions.def” 就实现了生成代码的目的，这算是比较高端的编程技巧吧，总之最终 class LangOptions 中有一些成员如

```
unsigned C99 : 1; //指示支持C99
unsigned C11 : 1; //指示支持C11
unsigned MSVCCompat : 1;  //指示支持Microsoft VC++ full compatibility mode
 ......
unsigned Bool : 1;  //指示将bool、true、false当作关键字
unsigned WChar : 1; //关键字wchar_t
 ......
unsigned ArrowDepth : 32; //->运算符最多连续使用32次
unsigned InstantiationDepth : 32; //模板类最大实例化嵌套深度是32
unsigned BracketDepth : 32; //括号最多嵌套32次
```

等等与语言相关的选项。

SourceManager 也是一个臃肿的类，按照代码中注释的介绍，这个类负责将源文件加载到内存、或许还提供一些缓存机制以优化读写？与编译过程关系紧密的实用性功能是将 SourceLocation 类型的对象转换为 spelling 或者 expansion location。与之呼应的是 SourceLocation 的注释提到 SourceLocation 对象编码了代码中的一个位置，SourceManager 可以将编码了的位置解码以获得 full include stack , line 和 column 信息。

HeaderSearch 负责提供 include search path 的相关信息，其成员

```
private:
  std::vector<DirectoryLookup> SearchDirs;
  //保存了include搜索路径
public:
  void AddSearchPath(const DirectoryLookup&,bool isAngled);
  //可以用来添加include搜索路径
```

HeaderSearch 还通过其构造函数与一个 HeaderSearchOptions 对象关联，HeaderSearchOptions 对象提供更多的 include search 信息，比如其成员

```
public:
  std::string Sysroot;
  //非空时，作为所有使用相对路径指示的头文件的虚拟根路径
  std::string ResourceDir;
  //存储compiler resource files比如builtin includes
```

相关的类型就介绍到这里，一个通过构造函数构造 Preprocessor 的例子在 这里，一个通过 CompilerInstance 结构构造 Preprocessor 的例子在 这里。或者也可以通过

```
  git clone http://git.ustclug.org:SirNing/LabWork.git

```

获取上述示例代码。

3.2 通过 class PPCallbacks 跟踪预处理过程以及 pp-trace 的介绍

pp-trace 是 clang-tools-extra 中的一个程序，给 pp-trace 输入一个源文件可以得到对该源文件进行预处理时发生的事件的 dump 信息，比如有 test.c 如下所示

```
#include "test.h"
#define TEST
```

执行

pp-trace test.c

会得到如下的结果，当然这只是结果的一小部分。

- Callback: InclusionDirective
  IncludeTok: include
  FileName: ”test.h”
  IsAngled: false
  FilenameRange: ”test.h”
  File: ”/home/sirning/Desktop/test/test.h”
  SearchPath: ”/home/sirning/Desktop/test”
  RelativePath: ”test.h”
  Imported: (null)
  //表示预处理遇到了一个#include指令，这个指令包括了头文件test.h
  //IsAngled为false表示test.h出现在引号而不是尖括号中
  //随后有该文件的绝对路径以及搜索路径等

- Callback: MacroDefined
  MacroNameTok: TEST
  MacroDirective: MD_Define
  //表示预处理遇到了一个宏定义
  //宏定义产生了一个符号TEST

那么 pp-trace 是如何产生这些 dump 信息呢，答案是使用 PPCallbacks(.h .cpp)。class PPCallbacks 定义了一系列回调函数，通过调用 Preprocessor::addCallbacks 将某个 PPCallbacks(的子类实例) 与 Preprocessor 绑定，于是在预处理过程中某个事件发生后 Preprocessor 就会调用与其绑定的相应的回调函数。比如 PPCallbacks 中有成员

```
1  void FileChanged(clang::SourceLocation Loc,
2           clang::PPCallbacks::FileChangeReason Reason,
3           clang:SrcMgr::CharacteristicKind FileType,
4           clang::FileID PrevFID = clang::FileID());
5  //当打开了一个源文件并将要处理或者处理完一个源文件并关闭了该文件时会调用这个函数
6  //Loc表示新的要处理的代码位置；枚举值Reason指示这个函数被调用的原因(如关闭了文件或者
7  //打开了新文件)；枚举值FileType表示文件的类型(用户源文件、系统源文件等)/
8  //；如果Reason指示关闭了文件，那么PrevFID表示被关闭的文件
9
10 void InclusionDirective(clang::SourceLocation HashLoc,
11          const clang::Token &IncludeTok,
12          llvm::StringRef FileName, bool IsAngled,
13          clang::CharSourceRange FileameRange,
14          const clang::FileEntry *File,
15          llvm::StringRef SearchPath,
16          llvm::StringRef RelativePath,
17          const clang::Module *Imported);
18 //当处理一个inclusion directive后(如#include、#import)时，无论这个预编译
19 //指令是否真的会导致一个文件被包含进来(如上述因为优化、找不到而跳过文件)，这个函数会被调用
20 //HashLoc是该预编译指令的位置；IncludeTok指示预编译指令是#include还是#import;
21 //FileName是预编译指令指示要包含的文件名；IsAngled为true指示使用尖括号扩住了文件名、否则
```

```
//是用引号扩住了文件名；FilenameRange指示文件名的范围在代码中的位置，CharSourceRange
//结构体主要包含了2个SourceLocation对象分别指示SourceRange的开头和结尾
//File表示预编译指令指示要包含的文件，应该指出FileEntry结构体包含了跟多和文件相关的信息，
//或者说更加类似于一个系统调用中的文件描述符，而FileName只是一个文件名，StringRef类型
//只是和一个字符串绑定
```

下面是对一个类似 pp-trace 的程序（源代码在这里）的跟踪调试的记录，观察其调用栈以确切发现这些回调函数在什么场景下被谁调用，同时也算是描绘了 llvm preprocessing 过程的轮廓。LLVM/CLANG 中定义了类似于下面的大量的 predefines，就把它们当作要处理的源程序了。

```
# 1 "<built-in>" 3
#define ___llvm___ 1
#define ___clang___1
#define ___clang__major___ 3
#define ___clang__minor___ 9
#define ___clang__patchlevel___ 0
//后续还有各种数据类型的最大值等宏定义
 ......
```

处理第一行时的调用栈如下所示，

```
main
clang::Preprocessor::Lex
//Preprocessor的Lex函数判断当前应该使用哪个Lexer对象，可选的有Lexer、PTHLexer、
//TokenLexer、CachingLexer，在不同的状态下调用对应的Lexer
clang::Lexer::Lex
//Lexer的Lex会初始化一个新的Token，设置标记位标记Token是否在一行的开始、是否
//HasLeadingSpace、是否HasLeadingEmptyMacro等
//随后将该初始化的Token传入LexTokenInternal中
//Lex发现读入了一个#，并且出去空白符号这个#是在一行的开头，于是认定遇到了一个预编译指令
//这个#可以称为内部符号，于是调用LexTokenInternal
clang::Lexer::LexTokenInternal
//LexTokenInternal是一个C词法分析器，在这里真正开始进行词法分析了，首先从buffer里读入
//一个字符，并将buffer的指针向后移动一位，发现读入的字符是#，这时lookahead发现下一个字符是
//空格，而且#在一行的开头，于是判定parse了一个#，调用HandleDirective
clang::Preprocessor::HandleDirective
//HandleDirective会检测正在被parse的预处理指令是不是在一个宏定义调用里、增加宏定义个数的计
//数器等，最终分析处当前parse的预处理指令的类型；当前只读了#，上面的lookahead并不影响
//buffer指针的位置，于是HandleDirective会调用LexUnexpandedToken继续parse#后面
//的内容，随后就能确定当前directive指令的类型并调用相应的handle*函数
clang::Preprocessor::HandleDigitDirective
//HandleDigitDirective会调用回调函数FileChanged，由于这个例子中并没有重载FileChanged
//于是调用继承自基类的FileChanged，基类的FileChanged什么都没有干
```

随后处理第二行 #define ___llvm___ 1 的调用栈如下所示

```
main
clang::Preprocessor::Lex
clang::Lexer::Lex
//前3层函数做的事情类似上述处理第一行时的分析
clang::Lexer::LexTokenInternal
//LexTokenInternal中lookahead发现#的下一个字符是d，于是认为遇到了一个预处理指令，调用
//HandleDirective
//这里lookahead是想识别#@等特殊情形，不过我从来没有遇见过代码中出现#@
clang::Preprocessor::HandleDirective
```

```
//仍然类似于第一行的处理方式，调用LexUnexpandedToken分析#后面的符号，确定当前预编译指令的
//类型，当然这里是一个#define，并调用对应的HandleDefineDirective
clang::Preprocessor::HandleDefineDirective
//这个函数做了一大堆事情，最终调用了MacroDefined回调函数
DumpPPCallbacks::MacroDefined
```

3.3 访问预处理结果

class PreprocessingRecord(.h, .cpp) 记录了词法分析 (或者预处理阶段) 的结果。这个类继承了 class PPCallbacks 重载了 PPCallbacks 中的部分回调函数，如介绍 PPCallbacks 的部分中提到的，这些回调函数会在预处理过程中被调用，PreprocessingRecord 正是通过这些回调函数记录预处理的信息。其私有成员变量

```
private:
  std::vector<PreprocessedEntity *> PreprocessedEntities;
  std::vector<PreprocessedEntity *> LoadedPreprocessedEntities;
```

分别记录了在被处理的文件里遇到的宏定义和从其他文件中加载的宏定义相关的类实例的指针 (PreprocessedEntity *，稍后介绍这个类型)。此外 class PreprocessingRecord 提供了 public iterator

```
public:
  iterator  begin();
  iterator  end();
  iterator  local_begin();
  iterator  local_end();
```

访问上述 PreprocessedEntity * 的列表，begin 和 end 用来访问所有的 (当前文件包含的和从其他文件中加载的)PreprocessedEntity * 的实例，local_begin 和 local_end 只用来访问当前文件包含的 PreprocessedEntity * 的实例。

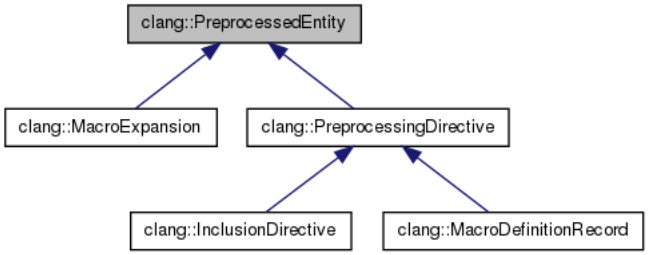
下面介绍一下 class PreprocessedEntity(.h, .cpp)，从上面的叙述中已经了解到这个类型的实例保存了和宏定义相关的信息，事实上 PreprocessedEntity 是具体表示不同预处理指令的类型的公共基类，下图 (图1) 给出了其继承关系，图中出现的所有子类都与 PreprocessedEntity 声明、定义在相同的文件中，就不一一给出他们源文件的连接了。PreprocessedEntity 中定义了枚举型 EntityKind 和私有成员 EntityKind Kind

```
enum EntityKind
{
  InvalidKind,
  MacroExpansionKind,
  MacroDefinitionKind,
  InclusionDirectiveKind,
  FirstPreprocessingDirective=MacroDefinitionKind,
  LastPreprocessingDirective=InclusionDirectiveKind,
};
```

来标记通过上述 iterator 访问到的一个 PreprocessedEntity 究竟是一个 MacroDefinitionRecord、InclusionDirective 还是 MacroExpansion(这三个子类的构造函数会先调用基类构造函数，调用基类构造函数时将其对应的 EntityKind 传递给基类构造函数)，于是在通过 iterator 访问到一个 PreprocessedEntity 实例时，可以先用 PreprocessedEntity::getKind 方法获取该实例类型，判断后将

其类型 cast 为对应的子类型，随后再通过其他接口获取该实例保存的预处理指令相关的信息。

图 1: PreprocessedEntity Inheritance



想要激活 Preprocessor 的 PreprocessingRecord 功能只需要在构造了 Preprocessor 之后调用 Preprocessor::createPreprocessingRecord,createPreprocessingRecord 的实现中会自动调用 addPPCallbacks, addPPCallbacks 的作用是将需要的回调函数与 Preprocessor 绑定，从而他们在预处理的过程中会被调用 (addPPCallbacks 的详细介绍见对 PPCallbacks 的介绍部分)。一个代码例子在

git clone https://git.ustclug.org:SirNing/LabWork.git
LabWork/clang-preprocessor-example/preprocessing-record

Part III

Clang AST 及其应用

4 AST Traverse Tutorial

LibTooling和 LibClang 都提供了操作 AST 的接口。LibTooling 提供了为开发基于 clang 的 standalone 工具的支持，然而 LLVM 和 Clang 都是发展很快的项目，有时新版本发布时，某些 API 可能会被修改。当用户需要一个稳定的接口时，LibClang(Clang's C Interface) 会是更好的选择。下面先简单介绍如何借助 LibTooling 进行 clang AST 的遍历。

首先需要了解 FrontendAction(.h, .cpp)

ASTConsumer (.h .cpp)

和 RecursiveASTVisitor (.h .cpp) 三个类，他们是 LibTooling 为遍历 AST 提供的接口。

4.1 FrontendAction

是大多数基于 LibTooling 的 stadalone 工具的入口。这是一个表示编译器前端可以执行的动作的接口，用户实现自己的 FrontendAction 以使自定义行为成为编译的一部分。这个类中有大量成员，它们的作用大致分为以下几个方面:

- set/get 编译器实例
- 提供当前被编译的文件、AST 的信息
- 指示该类定义的 Action 的模式，比如该 Action 是否只使用与处理器、该 Action 否支持使用 PCH、该 Action 是否支持使用 IR 等。
- Execute 成员用来激活 Action 的执行，BeginSourceFile、EndSourceFile 是 2 个回调函数，分别在 Action 执行前后被调用。
- 最后是一系列用户应该重载的回调函数，他们会在编译过程中某个事件发生时被调用，也正是通过重载这些函数实现自定义的 Action。在下面的例子中，暂时只会用到 CreateASTConsumer，这个函数作为上述 BeginSourceFile 的一部分被调用，CreateASTConsumer 为该 Action 创建 ASTConsumer 对象。

4.2 ASTConsumer

ASTConsumer 定义了面对不同类型 AST 节点应该执行的动作，此外使用 ASTConsumer 时不需要考虑 AST 是怎么生成的，不需要考虑 AST producer。这个类的成员相对有规律，除去少数几个成员函数外，其余的成员函数名都是 HandleASTNodeName 的格式，表示遍历遇到 ASTNodeName 类型的节点时应该执行的动作。比如

- HandleTopLevelDecl: 处理 AST 跟节点。
- HandleTranslationUnit: 在整个 translation unit 被 parse 过后被调用。
- HandleTagDeclDefintion: 当一个 TagDecl(struct、union、enum、class) 被 parse 时被调用。

4.3 RecursiveASTVisitor

RecursiveASTVisitor 可以对整个 AST 进行先序或者后续的深度优先的遍历。RecursiveASTVisitor 主要负责 3 个不同的任务

- 遍历 AST 访问每一个节点:成员函数 TraverseDecl(Decl *x) 完成这个任务;假设 x 是指向 AST 根节点的指针,那么 TraverseDecl 会根据 x 的动态类型将 x 分派到 TraverseDynamicType(DynamicType *x) 函数中去,DynamicType 指 x 的运行时类型。TraverseDynamicType 则会调用 WalkUpFromDynamicType

来完成下一个任务，随后 TraverseDecl 递归地处理 x 的子节点。成员函数 TraverseStmt 和 TraverseType 的工作方式与此类似。

- 给定一个 AST 节点，依次访问该节点的祖先节点直到遇到一个 Stmt、Decl 或者 Type（这三个是 AST 节点类型的基类）节点。上一个任务中提到的 WalkUpFromDynamicType(DynamicType *x) 完成这个任务,WalkUpFromDynamicType 首先会调用 WalkUpFromParentDynamicType(x), ParentDynamicType 是 x 的父节点的运行时类型，随后调用 VisitDynamicType(x) 来完成第三个任务。
- 给定一个节点，调用一个用户可重载的函数去真正访问这个节点。第二个任务中提到的 VisitDynamicType 函数来完成这个任务，这一系列 Visit 函数有 VisitStmt(Stmt *S)、VisitType(Type *T)、VisitDecl(Decl *D) 等。这些函数都返回一个 bool 值，true 指示继续进行 AST 的遍历，false 指示停止 AST 的遍历。

4.4 AST Traverse Example

下面是一个遍历 AST 并 dump 所有记录类型 (record type) 信息的例子，这段程序编译时需要链接库 libclangTooling。

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/Tooling.h"
using namespace clang;

class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  explicit FindNamedClassVisitor(ASTContext *Context)
    : Context(Context){}
  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
  //每遇到一个记录类型的声明就将该类型信息dump出来。
    Declaration->dump();
    return true;
  }
private:
  ASTContext *Context;
};

class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  explicit FindNamedClassConsumer(ASTContext *Context)
    : Visitor(Context){}
  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
  //当遇到一个translation unit，就从TranslationUnitDecl也即AST跟节点开始遍历。
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  FindNamedClassVisitor Visitor;
};

class FindNamedClassAction : public clang::ASTFrontendAction {
```

```
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNamedClassConsumer(&Compiler.getASTContext()));
  }
};

int main(int argc, char **argv) {
  if (argc > 1) {
    clang::tooling::runToolOnCode(new FindNamedClassAction, argv[1]);
  }
}
```

4.5 Build the AST Traverse Example Above

子小节4.4中只是一句话带过了上述例子需要在编译过程中链接 libclangTooling。这句话并不是针对直接调用编译器 (如 gcc) 加上-lclangTooling，下面详细介绍如何编译上述例子，读者也可以参考 Tutorial for Building tools using LibTooling and ASTMatchers。

总体而言，所要做的事情是将 AST Traverse Example 作为 clang-tools-extra 的一部分来编译。

Step 1下面的命令行首先获取 LLVM、Clang 和 clang-tools-extra 的代码，并将 clang 代码根目录重命名为 clang，放在 llvm 代码根目录的 tools 目录下，将 clang-tools-extra 代码根目录重命名为 extra，放在 clang 代码根目录的 tools/extra 目录下

```
mkdir project
cd project
#为了使行文清楚，这里定义一个环境变量供下文引用
PROJECT=$(pwd)
#获取并解压LLVM、Clang和clang-tools-extra版本3.9.0的代码
wget http://llvm.org/releases/3.9.0/llvm-3.9.0.src.tar.xz
wget http://llvm.org/releases/3.9.0/cfe-3.9.0.src.tar.xz
wget http://llvm.org/releases/3.9.0/clang-tools-extra-3.9.0.src.tar.xz
tar xf llvm-3.9.0.src.tar.xz
tar xf cfe-3.9.0.src.tar.xz
tar xf clang-tools-extra-3.9.0.src.tar.xz
#将clang的代码移动到llvm-3.9.0.src/tools并将clang代码的根目录重命名为clang
#将clang-tools-extra的代码移动到llvm-3.9.0.src/tools/clang/tools并重命名为extra
#重命名是必须的，这是由llvm-3.9.0.src/tools/CMakeLists.txt和
#llvm-3.9.0.src/tools/clang/tools/CMakeLists.txt决定的
mv cfe-3.9.0.src llvm-3.9.0.src/tools/clang
mv clang-tools-extra-3.9.0.src llvm-3.9.0.src/tools/clang/tools/extra
```

Step 2下面开始创建 AST Traverse Example 的项目文件 (源文件、CMakeLists 文件)，并向 clang-tools-extra 的 CMakeLists.txt 追加命令对例子进行编译。

```
#为了使行文清晰PROJECT在Step 1中有定义
#创建该例子的文件夹、源文件和CMakeLists文件
mkdir $PROJECT/llvm-3.9.0.src/tools/clang/tools/extra/ast-traverse-example
cd $PROJECT/llvm-3.9.0.src/tools/clang/tools/extra/ast-traverse-example
#创建源文件
#ExampleCode代表前一小结给出的示例代码
echo $ExampleCode > ASTTraverseExample.cpp
```

```
#创建CMakeLists.txt
#指定从源文件ASTTraverseExample.cpp生成ast-traverse-example
echo "add_clang_executalbe(ast-traverse-example ASTTraverseExample.cpp)" > CMakeLists.txt
#链接libClangTooling
echo "target_link_libraries(ast-traverse-example clangTooling)" >> CMakeLists.txt
#追加命令到clang-tools-extra的CMakeLists.txt对例子进行编译
cd $PROJECT/llvm-3.9.0.src/tools/clang/tools/extra
echo "add_subdirectory(ast-traverse-example" >> CMakeLists.txt
```

Step 3创建文件夹用于 CMake 的编译

```
#创建文件夹供cmake用来编译
#为了使行文清晰PROJECT在Step 1中有定义
cd $PROJECT
mkdir build
cd build
cmake ../llvm-3.9.0.src
#执行完make后便能发现在bin目录下面有AST Traverse Example生成的可执行文件
make
```

Part IV

Clang Static Analyzer

5 Clang Static Analyzer

clang 静态分析器 是用来找 C, C++ 和 Objective-C 源码中 bug 的源码分析程序, 可以作为一个独立程序运行, 也可以运行在 Xcode 中. 这个开源软件是 Clang 项目的一部分, 由 C++ 实现.

静态分析意味这个软件使用一系列算法和技术通过分析源码来自动寻找程序中的 bug. 类似于编译器的警告, 但是能力更强, 能检查出一般需要运行程序才能找到的错误.

Clang Static Analyzer 的目标是提供工业级质量的 C, C++ 和 Objective-C 程序的静态分析框架. 他基于 Clang 与 LLVM. 因为 Clang 是一系列可重用的类 C 语言源码相关的 C++ 库的集合. 所以也可以说 Clang Static Analyzer 是 Clang 的一部分.

需要注意的是:

- 这个项目正在开发中, 还有许多可以完善的地方.
- 运行耗时长于编译代码耗时. 一些算法的最坏时间复杂度为指数级.
- 检查不完美, 可能有假阳性, 几率取决于特定检查.
- 可能没有你需要的检查. 可以提 feature requests 或自己实现.

下面先介绍 Clang Static Analyzer 的安装使用, 提供的检查. 然后介绍如何实现自己的 Checker, 最后是对它的代码分析.

5.1 安装与使用

macOS 平台可以在官网上下载编译好的二进制. 其他平台需要从源码编译. CSA(Clang Static Analyzer) 是 Clang 的一部分, 安装参考 Clang 的安装方法.

使用 clang 调用 CSA

分析一个文件:

```
1 $ clang -cc1 -analyze -analyzer-checker=core.DivideZero test.c
```

列出所有可用的 ckeckers:

```
1 $ clang -cc1 -analyzer-checker-help
```

利用 scan-build 在命令行运行 CSA

scan-build 是一个让 CSA 分析一个代码库的命令行工具. 在一个代码库中, build 工程时, build system 会编译涉及到的源码. scan-build 通过替换掉原来的编译器, 使用一个伪编译器来执行静态分析.

scan-build 把环境变量 CC 暂时替换成 ccc-analyzer, ccc-analyzer 作为一个伪编译器, 把编译命令的参数传给原编译器去做编译, 传给 clang 做静态分析.

参考 官网上 scan-build 的文档.

使用时直接在编译命令前加上 scan-build 即可, 比如:

```
1 $ scan-build make
2 $ scan-build gcc -c t1.c t2.c
```

不加任何参数运行 scan-build 显示所有的选项.

scan-build 生成的报告是一组 HTML 文件, 每一个文件表示一种 bug 的报告. 可以直接点击 index.html 查看所有 bug 报告. -o 选项可以指定报告生成目录, -V 可以设置为在 scan-build 结束后立即打开报告.

建议将要分析的项目设为 debug 模式. debug 模式下, 程序中的断言 (assertion) 能够被 CSA 用来消除不可达的路径, 减少 false positives. 加上 –force-analyze-debug-code 可以使 scan-build 自动使断言生效.

5.2 可用的与可能实现的 Checkers

默认的 Checkers 被分为几大类:

- Core Checkers: 通用的检查, 比如除零, 空指针解引用, 使用未初始化的值
- C++ Checkers: C++ 语言特有的检查, 比如针对 new delete 的 double-free, use-after-free 检查
- Dead Code Checkers: 检查没有使用的代码
- OS X checkers: Objective-C 和 Apple SDK 的检查
- Security Checkers: 检查不安全 API 的使用, 例如 gets, strcpy
- Unix Checkers: 检查 Unix 和 POSIX 的 API 使用

没有指定时, CSA 运行默认的 Checkers. 此外并不还有一些实验性质的检查 (Alpha Checkers) 是所有的 CSA 支持的检查. 输入命令 clang -cc1 -analyzer-checker-help 可以查看所使用版本的 clang 所支持的 Checkers.

除了上面这些已有的 Checkers 以外, 官网上还在 Open Projects 这个页面上列出了能够提升分析器性能的开放项目. 在

List of potential checkers 页面上列出了一些可以去实现的 Checkers, 可以作为给 CSA 贡献代码的起点.

checkers 的源码在 clang/lib/StaticAnalyzer/Checkers

5.2.1 Default Checkers

Core Checkers

- core.CallAndMessage(C, C++, ObjC) , 检查函数调用时的逻辑错误. 如函数指针未初始化对象指针未初始化, 参数未初始化.
- core.DivideZero (C, C++, ObjC), 检查除零.
- core.NonNullParamChecker (C, C++, ObjC), 对于参数声明为 nonnull 的函数, 检查参数是否为空指针.
- core.NullDereference (C, C++, ObjC), 检查对空指针解引用.
- core.StackAddressEscape (C), 检查栈上内存是否逃出该函数.
- core.UndefinedBinaryOperatorResult (C), 检查二元操作符的计算结果是为否未定义值.
- core.VLASize (C), 可边长数组 (VLA) 的长度是否为 0 或未定义值.
- core.uninitialized.ArraySubscript (C), 数组下标是否未初始化.
- core.uninitialized.Assign (C), 赋值语句的值是否未初始化.
- core.uninitialized.Branch (C), 分支语句的条件测试的值是否未初始化.
- core.uninitialized.CapturedBlockVariable (C), 这个语法里的值是否未初始化.(从没见过)
- core.uninitialized.UndefReturn (C), 函数返回值是否未初始化.

C++ Checkers

- cplusplus.NewDelete (C++), 检查 double-free, use-after-free, 与 delete 相关的 bug.

Dead Code Checkers

- deadcode.DeadStores (C), 检查向不会再被使用到的变量赋值的语句.

OS X Checkers. OSX 操作系统的 API 与 Objective C 相关的一些检查.

Security Checkers

- security.FloatLoopCounter (C), 使用浮点数作为循环变量.
- security.insecureAPI.UncheckedReturn (C), 调用 setuid 等类似函数时应当检查其返回值.
- security.insecureAPI.getpw (C), 使用 gets 函数.
- security.insecureAPI.gets (C), 使用 gets 函数.
- security.insecureAPI.mkstemp (C), 使用 mkstemp 等函数时, 参数中格式字符串的'X' 少于 6 个. (用来生成临时文件的函数).
- security.insecureAPI.mktemp (C), 使用 mktemp 函数.
- security.insecureAPI.rand (C), 使用 random 等不被推荐使用的随机数生成器.
- security.insecureAPI.strcpy (C), 使用 strcpy 函数.
- security.insecureAPI.vfork (C), 使用 vfork 函数.

Unix Checkers

- unix.API (C)
- unix.Malloc (C)
- unix.MallocSizeof (C)
- unix.MismatchedDeallocator (C, C++, ObjC)
- unix.cstring.BadSizeArg (C)
- unix.cstring.NullArg (C)

5.2.2 Alpha Checkers

alpha checkers 并不是默认开启. 可能会有 false positives.

Core Alpha Checkers

- alpha.core.BoolAssignment (ObjC) , 将不是 0 或 1 的值赋值给布尔变量.
- alpha.core.CastSize (C), 将 malloc 返回的内存地址转换成某个类型, 但 malloc 内存大小字节数不是这个类型的整数倍. (须和 unix.Malloc 或 alpha.unix.MallocWithAnnotations 一同使用)
- alpha.core.CastToStruct (C, C++), 非结构体类型强制转换成结构体指针.
- alpha.core.FixedAddr (C), 将一个不变的地址赋值给指针.
- alpha.core.IdenticalExpr (C, C++), 可疑的相同表达式的使用. 比如两个分支条件相同.
- alpha.core.PointerArithm (C), 非数组的情况使用指针算数操作.
- alpha.core.PointerSub (C), 两个指向不同内存区域的指针相减.
- alpha.core.SizeofPtr (C), 对指针使用 sizeof, 可能应该对指针解引用.

C++ Alpha Checkers

- alpha.cplusplus.NewDeleteLeaks (C++), 检查 new delete 管理导致的内存泄露
- alpha.cplusplus.VirtualCall (C++) , 构造或析构中使用虚成员函数.

Variable Argument Alpha Checkers, 与可变长参数 stdarg.h 相关.

- alpha.valist.CopyToSelf (C), 宏 va_copy(x,y) , x,y 相同.
- alpha.valist.Uninitalized (C), 调用 va_arg , va_copy, 必须在 va_start 之后, va_end 之前.
- alpha.valist.Unterminated (C), va_start 必须有 va_end 匹配. 只能 va_end 一次

Dead Code Alpha Checkers

- alpha.deadcode.UnreachableCode (C, C++, ObjC), 不可达代码.

OS X Alpha Checkers, OS X 的 API 相关.

Security Alpha Checkers

- alpha.security.ArrayBound (C), 缓冲区溢出.
- alpha.security.ArrayBoundV2 (C), 缓冲区溢出.
- alpha.security.MallocOverflow, malloc 参数溢出.
- alpha.security.ReturnPtrRange(C), 函数返回值为越界指针.
- alpha.security.taint.TaintPropagation (C), 为其他 Checkers 提供 taint 信息.

Unix Alpha Checkers

- alpha.unix.Chroot (C), chroot 不当使用.
- alpha.unix.MallocWithAnnotations (C), 检查内存泄漏, double free, use-after-free. 假定所有释放资源的函数都有 annotation.
- alpha.unix.PthreadLock (C), 检查锁的使用.
- alpha.unix.SimpleStream (C), demo Checker, 检查 fopen fclose.
- alpha.unix.Stream (C), 检查流处理函数: fopen tmpfile fclose fread fwrite fseek ftell rewind fgetpos fsetpos clearerr feof ferror fileno
- alpha.unix.cstring.BufferOverlap (C), memcpy mempcpy 目的对于源缓冲区不能重叠.
- alpha.unix.cstring.NotNullTerminated (C), strlen 等标准库中字符串处理函数的参数是否是 null-terminated strings.
- alpha.unix.cstring.OutOfBounds (C), strncopy strncat 释放越界访问内存.

5.2.3 Potential Checkers

如果有兴趣对 CSA 的开发做贡献, 可以实现下面建议的 checkers.

memory

- memory.LeakEvalOrder (C,C++), 因参数执行顺序未定义导致的可能的内存泄漏.
- memory.DstBufferTooSmall (C, C++), 内存相关函数的目的缓冲区太小.
- memory.NegativeArraySize (C, C++), 数组大小负数.
- memory.ZeroAlloc (C, C++) malloc new 内存大小为 0.

constructors/destructors

- ctordtor.ExpInsideDtor (C++), 析构函数中抛出异常.
- ctordtor.PlacementSelfCopy (C++11), copy 或 move 的源对象和目的对象相同.

exceptions

- exceptions.ThrowSpecButNotThrow (C++) 函数接口有异常标识 throw(type), 但函数体没有抛出异常.
- exceptions.NoThrowSpecButThrows (C++) 函数借口有 throw() , 函数体中抛出了异常.
- exceptions.ThrownTypeDiffersSpec (C++) 函数借口与函数体抛出的异常类型不同.

smart pointers

- smartptr.SmartPtrInit (C++)
- C++03: auto_ptr
- C++11: 用 unique_ptr<type[]> 来保存 new[] 得到的指针.
-

dead code

- deadcode.UnmodifiedVariable (C, C++), 不是 const 或引用的变量从未被修改.
- deadcode.IdempotentOperations (C), idempotent 操作如:x = x, x /= x

POSIX

- posix.Errno (C)

undefined behavior , C/C++ 规范中的未定义行为

- undefbehavior.ExitInDtor (C++) , 在析构函数中调用 std::exit()
- undefbehavior.LocalStaticDestroyed (C++)
- undefbehavior.ZeroAllocDereference (C, C++)
- undefbehavior.DeadReferenced (C++)
- undefbehavior.ObjLocChanges (C++)
- undefbehavior.ExprEvalOrderUndef (C, C++03)
- undefbehavior.StaticInitReentered (C++)
- ...

different

- different.SuccessiveAssign (C), 连续对同一变量赋值.
- different.NullDerefStmtOrder (C), 可能发生空指针解引用的语句应该先测试指针是否为空.
- different.NullDerefCondOrder (C), 可能发生空指针解引用的条件应该先测试指针是否为空.
- different.MultipleAccessors (C++), accessor 的函数体相同.
- different.AccessorsForPublic (C++), 对 public 成员的 accessor. 该成员是否需要为 public ?
- different.LibFuncResultUnised (C, C++), 某些库函数的返回值没有被使用.
- different.WrongVarForStmt (C, C++), for 语句中, 条件判断所用的变量与递增变量不同, 可能错误. 比如 for(int i =0; i <3; j++)
- different.FloatingCompare (C), 浮点数比较, 比如 f == 0.5
- different.BitwiseOpBoolArg (C, C++), & | 的操作数为布尔类型. 应该为 && 或 || ?
- different.LabelInsideSwitch (C), switch 中的 label defalt: , 可能为 misprint
- different.IdenticalCondIfIf (C), 两个连续的 if 的条件拍的相同.
- different.LogicalOpUselessArg (C), 部分逻辑操作对表达式结果无影响 a < && a <10
- different.SameResLogicalExpr (C), 一个永真/永假的表达式, 如 i >0 && i <0
- different.OpPrecedenceAssignCmp (C, C++), 赋值优先级低于比较, 可能需要添加括号, 如 if (x = f() != y)
- different.OpPrecedenceIifShift (C, C++), ?: 优先级低于 << , 如 cout << a ? "a" : "b";
- different.ObjectUnused (C++), 创建的对象没有被使用.
- different.StaticArrayPtrCompare (C), 静态数组与 NULL 比较, 可能确实下标. int a[1][1]; if (a[0] == 0) {}
- different.ConversionToBool (C, C++), 不恰当的隐式转换成布尔值, 比如空字符串.
- different.ArrayBound (C++)

- different.StrcpyInputSize (C)
- different.IntegerOverflow (C)
- different.SignExtension (C)
- different.NumericTruncation (C)
- different.MissingCopyCtorAssignOp (C++)

WinAPI

optimization

- optimization.PassConstObjByValue (C, C++), 通过 const 引用传递 const 变量能避免复制, 如 void f(const struct A a)
- optimization.PostfixIncIter (C++), 前缀自增更快.
- optimization.MultipleCallsStrlen (C), 多次对同一字符串执行 strlen()
- optimization.StrLengthCalculation (C++), 使用 string::length() 计算 std::string 的长度更快. 避免 strlen(s.c_str())
- optimization.EmptyContainerDetect (C++), 使用 empty() 判断容器是否为空更快.

5.3 CSA 基本原理

在 clang/lib/StaticAnalyzer 有一个 README.txt. 对 CSA 有这样的介绍:

库的结构

这个静态分析库有两层, 底层的静态分析器 (GRExprEngine.cpp 和它的 friends) 和一些静态分析器 (*Checker.cpp). 后者通过 Checker 和 CheckerVisitor 的接口 (Checker.h CheckerVisitor.h) 建立在前者的基础上. 这样的设计将内部分析引擎与上层 Checkers 作者隔离开, Checker 作者只用接触简介的 Checker 的接口,

工作原理.

静态分析器受几篇基础研究论文 ([1] [2])

[1] Precise interprocedural dataflow analysis via graph reachability, T Reps, S Horwitz, and M Sagiv, POPL '95, http://portal.acm.org/citation.cfm?id=199462

[2] A memory model for static analysis of C programs, Z Xu, T Kremenek, and J Zhang, http://lcs.ios.ac.cn/ xzx/memmodel.pdf

Checker Developer Manual

探索程序中的所有路径, 执行 path-sensitive context-sensitive 的分析. 相比编译器能进行更深层次的分析, 找到更多 bug, 比如, use-after-free, 内存泄漏等.

CSA 核心在所给程序的控制流图上做符号执行 (symbolic execution). 符号执行类似于一般的程序执行, 但是会探索程序的所有可能执行路径, 变量的值用符号值来代替.

CSA 创建一个程序状态图. ExplodedGraph. 途中的节点为 ExplodedNode . 节点由 ProgramPoint 和 ProgramState 组成. ProgramPoint 表示在控制流图 CFG 上的位置. ProgramPoint 的 Kind 表示这个节点是在何时, 怎样加入图 (ExplodedGraph) 的. 比如 PreLoadKind, PostLoadKind. ProgramState 表示程序的抽象状态. 包括表达式到值的映射 Environment, 地址到值的映射 Store(符号内存模型), 符号值的约束 GenericDataMap.

在遇到 if 分支语句时, CSA 会跟踪两个分支的执行, 一个分支假定条件为真, 另一个分支假定条件为假. CSA 在每条执行路径上收集对这些符号值的约束, 用这些约束来判断符号值的可能取值以及一条路径是否可行. 当收集的约束无法满足时, 说明这一路径不可能到达. 便停止在这条路路上的搜索.

CSA 使用缓存节点的方法来尽量避免状态数指数级增长. 如果新生成的节点和已经存在的某节点的 ProgramStata 和 ProgramPoint 相同, 则复用这个节点. 所以 ExplodedNodes 基本上是不可变的对象, .

5.4 如何写 Checker

5.4.1 代码分析

代码使用的是 3.9, 超链接到 github 上的镜像, 可以在 Github 页面上方的搜索框输入filename: driver.cpp 或者点击 “Find file” 按钮来搜索文件. 虽然 doxygen 生成的文档更方便, 但是官网上的 3.9 与 4.0 的代码混着的. 有些关于类的继承关系还是链接到了 doxygen 网页上.

分析框架　Clang Static Analyzer(CSA) 的源码在 clang/include/clang/**StaticAnalyzer**/ 和 clang/lib/**StaticAn**. 分为三个部分.

- Checkers, 可加载到 CSA 上的独立的 checkers
- Core, 符号执行, 约束求解等核心部分
- Frontend, 与 clang 的前端的接口

frontend　Clang Static Analyzer 的分析实在 Clang 编译器前端的基础上实现, 前端生成源码的 AST 之后, CSA 作为一个ASTConsumer(doxygen) 在 AST 上进行静态分析. AnalysisConsumer.h (github) 中定义ASTConsumer 的子类AnalysisASTConsumer .

```
1 class AnalysisASTConsumer : public ASTConsumer {...};
```

上面这个只定义了一个 AddDiagnosticConsumer 的虚方法. 在 AnalysisConsumer.cpp(github) 中, AnalysisConsumer 继承AnalysisASTConsumer 与 RecursiveASTVisitor , 实现代码的分析.

```
1  class AnalysisConsumer : public AnalysisASTConsumer,
2                           public RecursiveASTVisitor<AnalysisConsumer> {
3   enum {
4     AM_None = 0,
5     AM_Syntax = 0x1,
6     AM_Path = 0x2
7    };
8    typedef unsigned AnalysisMode;
9
10   /// Mode of the analyzes while recursively visiting Decls.
11   AnalysisMode RecVisitorMode;
12   /// Bug Reporter to use while recursively visiting Decls.
13   BugReporter *RecVisitorBR;
14
15 public:
16   ASTContext *Ctx;
17   const Preprocessor &PP;
18   const std::string OutDir;
19   AnalyzerOptionsRef Opts;
20   ArrayRef<std::string> Plugins;
21   CodeInjector *Injector;
22
23   /// \brief Stores the declarations from the local translation unit.
```

```
  /// Note, we pre-compute the local declarations at parse time as an
  /// optimization to make sure we do not deserialize everything from disk.
  /// The local declaration to all declarations ratio might be very small when
  /// working with a PCH file.
  SetOfDecls LocalTUDecls;

  // Set of PathDiagnosticConsumers. Owned by AnalysisManager.
  PathDiagnosticConsumers PathConsumers;

  StoreManagerCreator CreateStoreMgr;
  ConstraintManagerCreator CreateConstraintMgr;

  std::unique_ptr<CheckerManager> checkerMgr;
  std::unique_ptr<AnalysisManager> Mgr;

  /// Time the analyzes time of each translation unit.
  static llvm::Timer* TUTotalTimer;

  /// The information about analyzed functions shared throughout the
  /// translation unit.
  FunctionSummariesTy FunctionSummaries;
   ...
};
```

其中 RecursiveASTVisitor(doxygen) 是一个在 Clang AST 上做前序和后序深度优先遍历的类.
这个类完成三种不同的任务:

1. 遍历 AST
2. at a given node, walk up the class hierarchy, starting from the node's dynamic type, until the top-most class (e.g. Stmt, Decl, or Type) is reached.

   为什么是从 dynamic type 开始?

3. given a (node, class) combination, where 'class' is some base class of the dynamic type of 'node', call a user-overridable function to actually visit the node.

These tasks are done by three groups of methods, respectively:

1. TraverseDecl(Decl *x) does task #1. It is the entry point for traversing an AST rooted at x. This method simply dispatches (i.e. forwards) to TraverseFoo(Foo *x) where Foo is the dynamic type of* x, which calls WalkUpFromFoo(x) and then recursively visits the child nodes of x. TraverseStmt(Stmt *x) and TraverseType(QualType x) work similarly.
2. WalkUpFromFoo(Foo *x) does task #2. It does not try to visit any child node of x. Instead, it first calls WalkUpFromBar(x) where Bar is the direct parent class of Foo (unless Foo has no parent), and then calls VisitFoo(x) (see the next list item).
3. VisitFoo(Foo *x) does task #3.

These three method groups are tiered (Traverse* > WalkUpFrom* > Visit*). A method (e.g. Traverse*) may call methods from the same tier (e.g. other Traverse*) or one tier lower (e.g. WalkUpFrom*). It may not call methods from a higher tier.

Note that since WalkUpFromFoo() calls WalkUpFromBar() (where Bar is Foo's super class) before calling VisitFoo(), the result is that the Visit*() methods for a given node are called in the top-down

order (e.g. for a node of type NamespaceDecl, the order will be VisitDecl(), VisitNamedDecl(), and then VisitNamespaceDecl()).

This scheme guarantees that all Visit*() calls for the same AST node are grouped together. In other words, Visit*() methods for different nodes are never interleaved.

Clients of this visitor should subclass the visitor (providing themselves as the template argument, using the curiously recurring template pattern) and override any of the Traverse*, WalkUpFrom*, and Visit* methods for declarations, types, statements, expressions, or other AST nodes where the visitor should customize behavior. Most users only need to override Visit*. Advanced users may override Traverse* and WalkUpFrom* to implement custom traversal strategies. Returning false from one of these overridden functions will abort the entire traversal.

CRTP

By default, this visitor tries to visit every part of the explicit source code exactly once. The default policy towards templates is to descend into the 'pattern' class or function body, not any explicit or implicit instantiations. Explicit specializations are still visited, and the patterns of partial specializations are visited separately. This behavior can be changed by overriding shouldVisitTemplateInstantiation in the derived class to return true, in which case all known implicit and explicit instantiations will be visited at the same time as the pattern from which they were produced.

AnalysisConsumer.cpp(github) 中, AnalysisConsumer 类 override 了 ASTConsumer(doxygen) 的 Initialize HandleTopLevelDecl, HandleTopLevelDeclInObjCContainer(for Objective-C), HandleTranslationUnit .

```
void Initialize (ASTContext &Context) override {
  Ctx = &Context;
  checkerMgr = createCheckerManager(*Opts, PP.getLangOpts(), Plugins,
                                    PP.getDiagnostics());

  Mgr = llvm::make_unique<AnalysisManager>(
      *Ctx, PP.getDiagnostics(), PP.getLangOpts(), PathConsumers,
      CreateStoreMgr, CreateConstraintMgr, checkerMgr.get(), *Opts, Injector);
}
/// Store the top level decls in the set to be processed later on.
bool HandleTopLevelDecl(DeclGroupRef D) override;
void HandleTopLevelDeclInObjCContainer(DeclGroupRef D) override;
void HandleTranslationUnit(ASTContext &C) override;
```

Initialize 代码里对 AnalysisConsumer 的成员进行初始化. ASTContext Ctx 是保存 AST 的所有信息. 其他几个和静态分析相关的成员在 core 部分再介绍.

HandleTopLevelDecl 和 HandleTopLevelDeclInObjCContainer 在遍历到 Top level declaration 时调用 storeTopLevelDecls() 将 Local translation unit 中的 declarations 保存到AnalysisConsumer 的成员变量 LocalTUDecls 中.

HandleTranslantionUnit(github) 在一个 Translation Unit 的 AST 生成之后被调用, 为 AnalysisConsumer 最重要的方法.

```
void AnalysisConsumer::HandleTranslationUnit(ASTContext &C) {
  // Don't run the actions if an error has occurred with parsing the file.
  DiagnosticsEngine &Diags = PP.getDiagnostics();
  if (Diags.hasErrorOccurred() || Diags.hasFatalErrorOccurred())
```

```
5       return;
6
7     // Don't analyze if the user explicitly asked for no checks to be performed
8     // on this file.
9     if (Opts->DisableAllChecks)
10      return;
11
12    {
13      if (TUTotalTimer) TUTotalTimer->startTimer();
14
15      // Introduce a scope to destroy BR before Mgr.
16      BugReporter BR(*Mgr);
17      TranslationUnitDecl *TU = C.getTranslationUnitDecl();
18      checkerMgr->runCheckersOnASTDecl(TU, *Mgr, BR);
19
20      // Run the AST-only checks using the order in which functions are defined.
21      // If inlining is not turned on, use the simplest function order for path
22      // sensitive analyzes as well.
23      RecVisitorMode = AM_Syntax;
24      if (!Mgr->shouldInlineCall())
25        RecVisitorMode |= AM_Path;
26      RecVisitorBR = &BR;
27
28      // Process all the top level declarations.
29      const unsigned LocalTUDeclsSize = LocalTUDecls.size();
30      for (unsigned i = 0 ; i < LocalTUDeclsSize ; ++i) {
31        TraverseDecl(LocalTUDecls[i]);
32      }
33
34      if (Mgr->shouldInlineCall())
35        HandleDeclsCallGraph(LocalTUDeclsSize);
36
37      // After all decls handled, run checkers on the entire TranslationUnit.
38      checkerMgr->runCheckersOnEndOfTranslationUnit(TU, *Mgr, BR);
39
40      RecVisitorBR = nullptr;
41    }
42
43    // Explicitly destroy the PathDiagnosticConsumer.  This will flush its output.
44    Mgr.reset();
45
46    if (TUTotalTimer) TUTotalTimer->stopTimer();
47    ...
48  }
```

clang 的 AST 没有一个公共的父类，主要从类型Decl, Stmt, DeclContext or Stmt 继承. TranslationUnitDecl 是 Decl 和 DeclContext 的子类，是一个 translation unit 的顶层节点. 关于 AST 的所有信息都在 ASTContext 中，程序通过 getTranslationUnitDecl 得到顶层 AST 节点，然后执行 AnalysisManager::runCheckersOnASTDecl().

接下来设置访问模式，默认为 AM_Syntax语法层模式，如果分析选项中没有 *InlineCall* ，则设置为AM_Path 路径敏感模式. *InlineCall* 将在后面介绍.

for 循环中对所有的 declaration 调用 RecursiveASTVisitor<> 的TraverseDecl函数. 所有的 declarations 处理之后，执行 AnalsisManager::runCheckersOnEndOfTranslationUnit(). 最后如果是 *InlineCall* 的情况下，需

要先构建函数调用图, 然后按照图的拓扑排序顺序进行分析.

AnalysisConsumer 还实现了 VisitDecl, VisitFunctionDecl, VisitObjCMethodDecl, VisitBlockDecl . 这些回调函数在 TraverseDecl 遇到对应 AST 节点时会被调用.

core

执行流程 clang 的 main 函数在 driver.cpp (提到的 cl mode指 windows 下 Visual Studio 的编译器 cl.exe. cl namespace 是给处理 command line 选项的代码用的.)

如果第一个参数为 -cc1, 处理. 在 L298 的 ExecuteCC1Tool 跳到 cc1_main.cpp 的 cc1_main 函数.

在这个函数. 初始化 CompilerInstance 和 CompilerInvocation 等对象.

CompilerInstance, 管理编译需要的众多对象, 如负责报错的 DiagnosticsEngine, 预处理器Preprocessor, AST 等.

CompilerInvocation 保存调用 compiler 需要的一些数据, 各种编译选项.

L116 调用 ExecuteCompilerInvocation() 跳到 ExecuteCompilerInvocation.Cpp, 这个函数前部分处理一些简单的命令行选项后, 创建 FrontendAction 对象. 这个对象的 各种子类 用来表示前端的不同动作. 例如 AnalysiAction 表示静态分析. ASTDumpAction, CodeGenAction 字面意思. 这个对象在初始化时, 根据命令行的选项, 在初始化代码的 switch 语句中返回 AnalysisAction.

L241 执行前端动作. 跳到 CompilerInstance.cpp. L860 的 for 循环对每个文件, 执行前端动作. 跳到 FrontendAction.cpp.

L457 的 ExecuteAction() 是一个虚函数, 根据 FrontendAction的动态类型调用不同的实现. 我们的静态分析调用的是 ASTFrontendAction::ExecuteAction(), ASTFrontendAction是 AnalysisAction 的父类. L554 createSema会为这个 CompilerInstance 创建一个 Sema 对象. 它创建时使用的ASTConsumer 为 AnalysisASTConsumer.

Sema 里面放一些回调函数, 实现语义分析 (semantic analysis) 和 AST 的创建.

L556 调用 ParseAST() 跳到 ParseAST.cpp 这里, 分析所给文件, 在分析过程中调用ASTConsumer. 分析过程中得到的类型信息和声明保存在 ASTContext

L126 创建 Parser , L150 递归下降分析程序.

parse 后, 在 L167 我们的 Consumer 就登场了. 这里调用的是AnalysisASTConsumer 的方法, 程序跳转到 AnalysisConsumer.cpp, 执行 HandleTranslationUnit().

AnalysisConsumer 中包含指向 CheckerManager 的指针.

CheckerManager 是用来管理所有 checkers 的类. 包含注册 checker 的方法和在程序的不同地方回调 checker 的方法. 以 runCheckerOnASTDecl 为例, 当调用这个方法时. CheckerManager 注册的 checkers 中找到所有在 Decl 节点需要执行动作的 checkers, 并且缓存下来. 然后依次执行这些 checkers 的动作.

CheckerManager 是 AnalysisManager 的一部分

Part V

Some Other LLVM Subsystem

6 Writing an LLVM Pass(version 3.3)

3.9 版本的 Writing an LLVM Pass 文档同 3.3 版本。

What is a pass? LLVM Pass Framework 是 LLVM system 中很重要的部分，是编译器最有趣的部分。Passes 实现构成编译器功能的的程序变换、优化，Passes 构造收集变换需要的信息，同时 Passes 也体现了编译器结构化设计的思想。
所有 Pass 实例都是 class Pass 的子类，用户在这些子类中重载由基类继承的虚函数。根据 custom pass 的工作方式，用户可以考虑从

- ModulePass
- CallGraphSCCPass
- FunctionPass
- LoopPass
- RegionPass
- BasicBlockPass

等类继承以进一步让 LLVM 明白 custom pass 可能会做些什么、如何与其他 pass 耦合以及优化 pass 的执行。

Write a HelloWorld Pass llvm-3.3.src/lib/Transforms/Hello 目录下有一个 HelloWorld Pass 的例子，这个 pass 输出所有函数的函数名，代码如下所示。

```
1  //需要的头文件和命名空间
2  #include "llvm/Pass.h"
3  #include "llvm/Function.h"
4  #include "llvm/Support/raw_ostream.h"
5  using namespace llvm;
6
7  namespace {
8    //FunctionPass一次作用在一个函数上
9    struct Hello:public FunctionPass
10   {
11     //成员ID供LLVM识别不同的pass
12     static char ID;
13     Hello():FunctionPass(ID){}
14     //重载继承的函数，在这个函数中实现
15     //用户自定义动作
16     virtual bool runOnFunction(Function &F)
17     {
18       errs()<<"Hello:";
19       errs().write_escaped(F.getName())<<'\n';
20       return false;
21     }
22   };
23 }
24 //初始化Hello::ID，由于LLVM根据ID的地址识
25 //别不同的pass，所以ID的初值是多少并不重要
26 char Hello::ID=0;
27 //注册Pass HelloWorld，在命令行中这个pass
28 //可以通过参数"hello"被调用，"Hello World
29 //Pass"是该pass的名字，如果一个pass遍历CFG
```

```
30 //并不修改CFG，那么第三个参数设置为true,
31 //如果一个pass是analysis pass那么第四个
32 //参数设置为true
33 static RegisterPass<Hello> X("hello","Hello World Pass",false,false);
```

如果使用 cmake 编译 LLVM 3.3，那么上面这个例子会自动被编译，生成 <OutputDir>/lib/LLVMHello.so。那么使用

```
1 opt -load <OutputDir>/lib/LLVMHello.so <input bitcode>
```

就可以在指定的文件上执行该 pass。

Pass Classes 下面简单介绍各种 base pass classes。

- ImmutablePass是不做任何事情、不是必须要被执行的、永远不需要升级的 pass，不是一个正常的 transformation 或 analysis，但是可以用来输出一些有关编译器配置的信息和可能影响各种 transformati on 的信息。
- ModulePass是所有 base pass classes 中最通用的 pass，继承 ModulePass 意味着 custom pass 作用在整个程序上，可能以任意的顺序访问函数、添加修改函数。由于 ModulePass 很通用，对其不能做出任何猜测，因此它的执行不能被优化。
- CallGraphSCCPass会以 botto m-up 的顺序遍历程序的 call graph。执行可以被优化。
- FunctionPass独立地每次作用在程序的每一个函数上，这意味着不能修改被处理函数外的函数、不能在编译模块中添加或删除函数和全局变量、不能在作用在一个函数上的同时维护 pass 的状态。
- LoopPass以从内层到外层的顺序、独立地作用在一个函数中的每一个循环上。可以通过 LPPassManager 接口更新循环嵌套。
- RegionClass以从内层到外层的顺序作用在仅有一个入口和一个出口的代码块上。可以通过 RGPassManager 更新代码块的嵌套。
- BasicBlockPass每次作用在一个 basic block 上。类似于 FunctionPass，BasicBlockPass 不能修改或访问当前正被作用的 basic block 外的 basic block、不能在作用在一个 basic block 上的同时维护 pass 的状态、不能修改控制流图、不能修改全局变量。
- MachineFunctionPass是 LLVM code generator 的一部分，作用在与机器相关的 LLVM 函数表示上。不再细述。

对于 LoopPass，进一步详细介绍用户可以重载的 3 个函数以实现自定义功能，其他 pass 有的类似于 LoopPass 如 FunctionPass、RegionPass 和 BasicBlockPass，用户也可以查询详细的手册。
这 3 个函数是

```
1 virtual bool doInitialization (Loop *,LPPassManager &LPM);
2
3 virtual bool runOnLoop(Loop *,LPPassManager &LPM)=0;
4
5 virtual bool doFinalization();
```

他们的返回值都是 bool，返回 true 指示函数修改了程序，返回 false 指示函数没有修改程序。
doInitialization 函数应该完成简单的与要处理的函数无关的初始化，参数 LPPassManager 用来访问 function or module level analysis information。
runOnLoop 函数是必须要被重载的，实现期望的 analysis 或者 transformation，LPPassManager 用

来更新 loop nest。
当所有的循环都被 runOnLoop 作用后，会调用 doFinalization 函数。

Specifying interactions between passes PassManager 负责不同 pass 之间的正确交互,PassManager 也会尝试优化 pass 的执行，因此它必须知道不同 pass 之间的依赖关系。那么这些依赖关系是由 pass 本身给出的，每一个 pass 都可以指定自己执行之前必须执行哪些 pass (required sets)、自己的执行会使哪些 pass 无效 (invalidated sets)。
可以通过重载函数

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

来指定上述 required sets 和 invalidated sets。getAnalysisUsage 函数应该用 pass 之间的依赖关系信息填充参数 AnalysisUsage 对象实例，具体通过调用下面的函数实现。

```
AnalysisUsage::addRequired<>
AnalysisUsage::addRequiredTransitive<>
AnalysisUsage::addPreserved<>
Pass::getAnalysis<>
Pass::getAnalysisIfAvailable<>
```

custom pass 用 addRequired 和 addRequiredTransitive 指定 custom pass 运行前必须要运行的 pass, Transitive 意味着 passes chain 在一起。
custom pass 用 addPreserved 指定 custom pass 不会使哪些已经存在的 pass 计算的结果无效 (complementary set of invalidated set)，利用这些信息 PassManager 可以优化 pass 的执行。
custom pass 用 getAnalysis<PassType> 可以获取 custom pass 用 addRequired 指定的 pass 的引用，用想要想要获取的 pass 的类型替换模板类型 PassType。如果获取一个没有用 addRequired 指定的 pass 会发生运行时错误。getAnalysisIfAvailable 不是返回一个引用而是返回一个指针，当要求的 pass 不存在时 (没有被激活? 不是很懂) 返回 NULL，存在 (激活) 时，custom pass 可以通过指针 update pass。

7 LLVM Alias Analysis Infrastructure (version 3.3)

Overview Alias Analysis(又名 Pointer Analysis) 是一系列分析 2 个指针是否指向相同的内存区域的技术。传统上 Alias Analysis 返回 Must、May or No alias response，分别对应 2 个指针一定指向相同内存区域、可能指向相同内存区域、不可能指向相同内存区域。
LLVM 的 class AliasAnalysis 是用户使用 alias analysis 的主要接口，除了可以提供 alias analysis 相关信息外 class AliasAnalysis 还可以提供 Mod/Ref 信息 (Mod for modification、Ref for reference，通过一个指针修改了另一个指针指向的内存、通过一个指针引用)
下面主要介绍如何通过 class AliasAnalysis 进行 alias analysis。

Class AliasAnalysis Overview class AliasAnalysis 包含 2 个枚举

```
enum AliasResult
  {
    MustAlias,
    PartialAlias,
    MayAlias,
```

```
    NoAlias
  };
enum ModRefResult
  {
    ModRef,
    Mod,
    Ref,
    NoModRef
  };
```

分别表示 alias 和 mod/ref 分析结果。

class AliasAnalysis 使用起始地址和大小描述内存对象，使用 call instruction 描述函数。其中起始地址使用

```
class Vaule;
```

的对象表示，结合静态类型计算大小。

class AliasAnalysis::alias 成员函数是用来进行分析的主要接口，这个函数接受 2 个指针值作为参数返回 enum AliasResult 的一个枚举值。

- MustAlias 表示 2 个指针指向的对象在内存中一定有相同的起始地址。
- PartialAlias 表示 2 个指针指向的对象在内存中以某种方式重叠但起始地址不同。
- MayAlias 表示 2 个指针指向的对象可能是同一个对象。
- NoAlias 表示通过 2 个指针对内存的操作是相互独立的。比如 2 个指针指向 2 块完全不重叠的区域、比如只用 2 个指针进行读操作、比如 2 个指针指向同一块内存区域但是使用一个指针前已经释放了另一个指针。

class AliasAnalysis::getModRefInfo 成员函数分析 2 次操作 (可以是一条指令也可以是一个函数) 对内存的访问是否互相有读写依赖，返回一个 enum ModRefResult 枚举值。

- ModRef 表示操作 (参数 1)读或写了操作 (参数 2)写了的区域。
- Mod 表示操作 (参数 1)写了操作 (参数 2)读或写了的区域。
- Ref 表示操作 (参数 1)读了操作 (参数 2)写了的区域。
- NoModRef 表示 2 个操作都不向对方所读取的内存区域写任何数据。

class AliasAnalysis::pointsToConstantMemory 成员函数返回 bool 值指示指针是否指向 unchanging memory location(如函数、全局常量和空指针)。

class AliasAnalysis::doesNotAccessMemory 和 onlyReadsMemory 成员函数可以分析函数对内存的访问情况。前者返回 true 指示被分析函数一定不会读写内存或只读取 constan memory，这类函数的返回值只依赖于参数，那么可以放心地用于公共子表达式、循环常量外移等优化 (比如 sin、cos)。后者返回 true 指示被分析函数只会从 non-volatile memory 读取数据，这类函数的返回值只依赖于参数和调用时内存状态，只要没有 store 指令改变内存状态，这类函数的调用位置就可以随意改变，这从优化的角度看也是很好的性质。应该注意到符合前者所表达性质的函数也符合后者表达的性质。

Write a new AliasAnalysis 下面介绍如何借助 class AliasAnalysis 实现一个 custom alias analysis(CAA)。

首先要确定 CAA 的作用域，并根据作用域选择不同的 pass 基类作为 CAA 的父类。比如

- class Pass for interprocedural analysis。
- class FunctionPass for function-local analysis。

- 请参考 LLVM Pass 部分获取关于更多 Pass 基类的介绍。

除了继承某个 Pass 基类还要继承 AliasAnalysis。CAA 必须要重载 Pass::getAnalysisUsage 并在这个重载函数中显示调用 AliasAnalysis::getAnalysisUsage,CAA 必须重载 Pass::run(FunctionPass::runOnFunction) 函数并在这个重载函数中调用 AliasAnalysis::InitializaAliasAnalysis。如果用代码更能说明问题的话，CAA 应该看起来像下面的代码

```
class CAA:public Pass,public AliasAnalysis
{
  void getAnalysisUsage(AnalysisUsage  &AU) const
  {
    AliasAnalysis::getAnalysisUsage(AU);
    //declare your dependencies
  }
  bool run(Module &M)
  {
    AliasAnalysis:: InitializeAliasAnalysis (this);
    //perform analysis
    return false;
  }
};
```

8 LLVM Alias Analysis Infrastructure (version 3.9)

Overview LLVM 3.9.0 对实现 custom alias analysis(CAA) 有了新的要求，除了上一小节 3.3 版本中提到的要求外现在 CAA 还必须重载 Pass::getAdjustedAnalysisPointer。一个实例如下述代码所示。

```
void *getAdjustedAnalysisPointer(const void *ID) override
{
  if( ID == &AliasAnalysis::ID)
    reutrn (AliasAnalysis*)this;
  return this;
}
```