基于 LLVM 框架的 Clang 编译器

一、LLVM 简介（可不看）

LLVM 是用 C++开发的一个开源编译器框架，它的研究目的在于提供一个基于 SSA（Static Single-Assignment，详见 http://blog.csdn.net/lm2302293/article/details/6791752）的编译器，支持任意语言的静态和动态编译。LLVM 项目集包括了：

* LLVM Core：LLVM 的核心库，提供了平台无关的优化器和代码生成器
* Clang：C/C++/Objective-C 编译器；Clang Static Analyzer，源代码检查工具，无需编译便可直接检查。
  dragonegg：GCC 后端插件，结合 GCC 前端使用，能编译 Ada,Fortran 以及其他 GCC 支持的语言。
  LLDB：调试器，官网介绍它比 GDB 执行速度更快、内存使用更高效。
  libc++：C++标准库，完全支持 C++11。
* compiler-rt：编译器运行时库，为编译产生中间代码提供支持。
  OpenMP subproject：在 Clang 编译器中实现 OpenMP（OpenMP：多线程程序的编译处理方案，#pragma 预处理指令自动将程序并行化）
  vmkit：基于 LLVM 的 Java 和.Net 虚拟机
  polly：优化 cache-locality，以及并行优化器
  libclc：旨在实现 OpenCL 标准库（OpenCL：面向异构系统的通用并行编程，CPU + GPU 并行运算）
  klee：尝试发现 bug 后产生测试用例，这功能听起来不错。
  SAFECode：编译 C/C++时检查内存安全问题（如内存溢出）。
  lld:旨在实现 Clang/LLVM 的内建链接器。目前 Clang/LLVM 还需使用操作系统的链接器。

（*为 clang 编译器必须组件）

二、LLVM 的特性

1. 模块化　　IR（Intermediate Representation）

SourceCode -> FrontEnd -> Optimizer -> BackEnd -> MachineCode
　　Clang　IR　　\ LLVM Core /

2. 优点

代码结构清晰，易跟 IDE 或文本编辑器集成，用作 coding 时的语法分析和代码提示，如 vim 的 clang_complete，https://github.com/Rip-Rip/clang_complete

错误提示友好，实测发现确实如此；

编译结果可用 gdb 调试。

三、Clang 与 GCC 比较（可不看）

1. 编译速度

2010 年的数据，跟 gcc 差别不大。

http://www.phoronix.com/scan.php?page=article&item=gcc_llvm_clang&num=1

http://www.phoronix.com/scan.php?page=article&item=llvm_gcc_dragonegg28&num=1

2. 目标文件

用 gcc 编译出来的 clang 大小 1.3GB，用 clang 编译出来的 clang 大小 1.1GB，二者文件数量相同都是 22403 个，说明 clang 编译出的目标文件小。

再分别编译一个我写的 SIMD 测试程序，运行目标文件发现二者速度差别不大，说明 clang

和 gcc 编译结果的优化程度相当。看了别人的测试结果亦如此：
http://www.linuxidc.com/Linux/2012-11/74062.htm

四、编译 LLVM

下载 gcc4.4.7 能编译的 LLVM 的最高版本 3.4.2，欲编译之，不知怎么开始。看了 Makefile 文件，太长看不完，转而看介绍文档 README.TXT 和 docs/GettingStarted.rst, 发现的编译相关操作（根据文档中的 svn 命令提炼出如下步骤）：

1. 三个组件的源码文件夹分别命名为 llvm, clang, compiler-rt
2. clang 放在 llvm/tools/目录下，compiler-rt 放在 llvm/projects 目录下。
3. 进入跟 llvm 同级的目录

```
$ mkdir build
$ cd build
$ ../llvm/configure --enable-optimized
$ make
$ make check-all
```

编译出来后在 llvm.org 查看此版本的 release note，得知它不支持 C++11, 而下一版本支持但需要更高版本的 gcc 来编译，没办法了只能用这个。

五、使用 Clang 及其工具集（常用的标记为蓝色）

1. Clang 使用方法（详见$ Clang -help）

首先说明一点，因为 clang 大部分编译选项兼容 gcc, 所以 Makefile 文件只需将 CC=gcc, CXX=g++改成 CC=Clang, CXX=Clang++

编译源文件生成目标文件

```
$ clang *.c  -o *.o 或 $ clang  -o *.o *.c
```

生成本机汇编代码

```
$ clang *.c -S -o *.s
```

gcc 方式，检查源文件语法错误

```
$ clang *.c -fsyntax-only
```

Clang 特有，静态分析源文件语法错误无需编译

```
$ clang --analyze *.c
```

生成中间代码，.bc 是二进制格式，.ll 是文本格式
-S 只执行编译，-emit-llvm 生成 LLVM 的中间代码

```
$ clang *.c -S -emit-llvm -o *.bc 或 $ clang -emit-llvm -c *.c -o *.bc
```

用.bc 生成.ll

```
$ llvm-dis *.bc
```

2. Clang 工具集（只列举手册介绍的）

bugpoint 针对.ll 或.bc 文件,在优化中间代码的时候用于 debug 以下三种情况:optimizer crash, optimizer mis-compilation, bad native code, 从而减少编译器后端遇到的问题。

```
$ bugpoint [options] [.ll or .bc file] [LLVM passes] [--args program arguments]
```

FileCheck　针对.ll 文件，从标准输入（shell 管道）读取一个.ll 文件用于检验另一个.ll 文件，看编译器是否达到预期的输出

```
$ llvm-as < *.ll | llc -march=x86-64 | FileCheck *.ll
```

llvm-lit　LLVM 自身测试工具，用在编译完成后测试各组件

```
$ llvm-lit ~/llvm/test/Integer/BitPacked.ll
```

llc　将 LLVM 中间代码编译成目标平台的汇编代码

```
$ llc *.ll -o *.s 或 $ llc *.bc -o *.s
```

lli　直接运行 llvm 中间代码

```
$ lli *.ll 或 $ lli *.bc
```

llvm-ar　同 Linux ar，将多个.o 文件打包成.a 静态库

```
$ llvm-ar rcs *.a *.o *.o
```

llvm-as　将.ll 文件转换成.bc 文件，二者都是 LLVM 中间代码，.ll 是可读文本，.bc 是二进制文件

```
$ llvm-as *.ll -o *.bc
```

llvm-bcanalyzer　分析.bc 文件生成一些统计信息

```
$ llvm *.bc
```

llvm-config　显示编译 LLVM 时的编译选项

```
$ llvm-config --cxxflags
```

llvm-dis　将.bc 文件转换成.ll 文件

```
$ llvm-dis *.bc -o *.ll
```

llvm-link　将多个.bc 文件链接成单个.bc 文件

```
$ llvm-link *.bc *.bc -o target.bc
```

llvm-nm　列出.bc 文件，.so 文件，.a 文件，或者可执行文件中的 symbol name

```
$ llvm-nm *.bc
```

llvm-stress　随机生成一个.ll 文件供 LLVM 组件测试用

```
$ llvm-stress > *.ll
```

opt　根据选项对.ll 或.bc 文件进行优化，生成.bc 文件

```
$ opt [options] *.ll
```