LLVM 的 JIT 辅助库 ORC JIT 介绍

樊其轩

September 9, 2023

ORC 是什么，不是什么

- 是辅助开发者实现 JIT 功能的通用库和运行时，提供一系列 JIT 用到的支持组件
- 可以将 LLVM IR 动态编译到机器码，但不局限于 LLVM IR
- 黑箱：交给你代码，查询某个符号，给我使这个符号可用并返回地址

- 不是一个完整的 JIT 编译器
- 不提供动态编译相关的代码分析和优化

ORC 的功能

- 即时编译，查找一个符号时立刻编译依赖代码
- 懒惰编译，延后到某函数符号被调用时再编译依赖代码
- 多线程编译和运行
- 远端运行，编译和运行分开在不同进程或网络上的不同机器
- 像普通程序一样的运行时环境，
  - 异常处理 (C++ 的 try catch)
  - 静态数据初始化（静态全局变量/类初始化）
  - dlopen
  - 线程局部存储 (TLS)
- 像普通链接器一样的功能，链接顺序，链接静态库和动态库，符号解析
- 符号注入，符号动态重定向与重命名，代码动态加载和卸载，缓存编译完成的代码
- 支持调试动态生成代码
- 提供相对稳定的 C ABI，但很多功能都没暴露出来
- 自动内存管理，丢弃不需要的中间产物

ORC 的当前局限

- 在多线程环境下提供 JIT 编译好的代码镜像与相关调试信息
  - 由于多线程编译以及代码加载与卸载，编译出的代码镜像会发生变动
  - JIT 专用链接器也会做死代码删除
- 造成的局限
  - JIT 编译的程序的调试支持是通过一个临时的 `notifyMaterializing` 接口实现的 [1]
    - 因为 GDB JIT 接口[2]需要编译好的镜像
  - 难以获取行号等等代码附带的信息，除非暂且用 `notifyMaterializing`，参考 Julia[3]
    - Linux Perf 的 JIT 接口需要
- 未来会研究出更好的机制，来暴露出这方面信息

[1] https://github.com/llvm/llvm-project/commit/ef2389235c5dec03be93f8c9585cd9416767ef4c
[2] https://sourceware.org/gdb/onlinedocs/gdb/JIT-Interface.html
[3] https://github.com/JuliaLang/julia/blob/master/src/jitlayers.cpp

ORC 的组件

依照抽象层级

- JITLink & RuntimeDyld: JIT 链接器，解析符号，链接 JIT 代码到当前进程
  - 与其他组件的接口：`ObjectLinkingLayer` 和 `JITLinkContext`
  - Section, Block, Edge, Symbol
- ORC 核心: 提供 ORC 库的基础抽象，各个功能特性都围绕于它
- LLJIT/LazyLLJIT: 方便用户的顶层 API，把 ORC 所有功能整合起来提供给用户
  - 层级设计
  - `IRTransformLayer -> IRCompileLayer -> ObjectTransformLayer -> ObjectLinkingLayer`

依照功能特性

- ExecutorProcessControl(EPC): 负责 JIT 生成的二进制的运行
- *Platform：用于支持不同平台加载动态共享对象 (DSO)，compiler-rt 下也有相关代码
- DynamicLibrarySearchGenerator: 加载当前进程或者某个动态库的各种符号
- SPS*: 简单序列化库，在 EPC 中用来支持与远端通信，调用远端代码，发送编译好的代码
- DebuggerSupportPlugin: JITLink 链接器的插件，用来支持 JIT 程序调试
- JITLinkMemoryManager: JITLink 的内存分配器
- 等等等

ORC 的组件

ORC 的核心抽象

- JITDylib
  - 就像动态库一样，可以连成一串链接 (DFS)
  - 灵活性，里面可以是任何内容，只要最终能给 JIT 链接器提供符号和地址
  - 懒惰性，里面的代码不需要立刻被编译，不需要立即可用
- ThreadSafeModule 和 ThreadSafeContext
  - LLVMModule 和 LLVMContext 的多线程安全封装
  - `WithModuleDo`
- MaterializationUnit
  - 记录可以一块被编译并解析好 (materialize) 的符号
  - `materialize()` 和 `discard()` 接口定义如何 materialize 这些符号
  - AbsoluteMaterializationUnit
- MaterializationResponsibility
  - 用于多线程环境下传递和追踪错误信息
  - `notifyResolved()`, `notifyEmitted()`, `failMaterialization()`
- ResourceTracker
  - 用来支持代码加载和卸载
  - 一般每个 JITDylib 会有一个，也可以定义多个
- ExecutionSession
  - 代表一个 JIT 运行实例，包含所有 JITDylib，符号名称字符串池，EPC 等等
  - 带锁，协调 JIT 全局各个组件，向相应 JITDylib 分发符号查询和编译请求

ORC 使用

文档里的简单示例代码，省去错误处理

```
auto JIT = LLJITBuilder().create();
// Add the module.
JIT->addIRModule(TheadSafeModule(std::move(M), Ctx));

// Look up the JIT'd code entry point.
auto EntrySym = JIT->lookup("entry");
if (!EntrySym)
  return EntrySym.takeError();

// Cast the entry point address to a function pointer.
auto *Entry = EntrySym.getAddress().toPtr<void(*)()>();
// Call into JIT'd code.
Entry();
```

LLJIT 的更多例子 https://github.com/llvm/llvm-project/tree/main/llvm/examples/OrcV2Examples

ORC 与 MCJIT 对比

MCJIT

- 对应接口很繁琐，参看 ExecutionEngine 类[a]有多少 `get\*Address,getPointerTo\*` 这种函数
- 垒代码山垒出来的接口 (adhoc)，有设计问题
- 一个 MCJIT 实例单独处理一个 JIT 代码模块
- 复用 LLVM + JIT 专门的 RuntimeDyld 链接器
- 提供基础 JIT 功能，缺少一些功能如 small codemodel 支持，部分功能不完善不稳定比如 TLS 和 Windows COFF 格式
- 整个 JIT 流程简单固定，缺少对内部流程的可拓展性
- 没有的自动内存管理，内存浪费

ORC

- 提供方便的接口，来查找 JIT 出的函数/符号地址
- 一个 LLJIT 实例（单例模式）处理多个 JIT 代码模块
- 同时支持 RuntimeDyld 和新的灵活的 JITLink 链接器
- 支持更丰富更完整的功能
- 内部流程都可定制，灵活可拓展的接口，且各个组件相对独立可单独使用
- 提供自动内存管理，并可定制

[a] https://llvm.org/doxygen/classllvm_1_1ExecutionEngine.html

ORC 实际应用

- Julia 编译器[4] 用到了很多复杂功能
- CLASP 基于 LLVM 实现的 common lisp 编译器[5]

- PostgreSQL 重复且昂贵的查询指令 JIT 到二进制来加速
- Numba (通过 llvmlite) [6]
- LLVM IR 的解释器 lli

- LLVM BOLT 单独使用了底层的 JITLink 链接器[7]
- 把 C++ 当脚本语言写，Facebook 的用例[8]
- Clang-Repl 交互式 C++ 解释器[9]

[4] https://github.com/JuliaLang/julia/blob/master/src/jitlayers.cpp
[5] https://github.com/clasp-developers/clasp
[6] https://github.com/numba/llvmlite/pull/942
[7] https://github.com/llvm/llvm-project/commit/05634f7346a59f6dab89cde53f39b40d9a70b9c9
[8] https://www.youtube.com/watch?v=01WoFnyw6zE
[9] https://clang.llvm.org/docs/ClangRepl.html

相关资料推荐

LLVM 官方文档中的 Kaleidoscope 教程有实操用 ORC 给编译器添加 JIT 功能章节，很推荐

除去 ORC 官方文档[10]还有：

- Lang Hames 的 Youtube 上 ORC 介绍视频，一搜就有总共 4 个
- Sunhao 的 JITLINK 介绍和 windows 支持
  - https://www.youtube.com/watch?v=UwHgCqQ2DDA
- LLVM Discord 服务器上的 `#jit` 频道，开发者在此非常活跃
- 我之前的 LLVM ORC JIT 简介[11]
  - https://www.bilibili.com/video/BV13a41187NM

[10]注意，由于 ORC 发展挺快，官方文档有一定滞后

[11]有错误，欢迎指正。内容远不及看上面的材料和动手实验