

# 让编程语言爱上 RISC-V

## 以 Crystal 为例

Coelacanthus

PLCT Arch RISC-V 小队

2022 年 4 月 6 日

# 移植工作的主要组成部分



编译器的支持 主要包括代码生成、一些常量，以及一些架构相关的特殊语法

# 移植工作的主要组成部分



**编译器的支持** 主要包括代码生成、一些常量，以及一些架构相关的特殊语法

**标准库的支持** 主要需要处理的是系统调用和内嵌汇编，以及某些在 RISC-V 上不再满足的假设

# 移植工作的主要组成部分



**编译器的支持** 主要包括代码生成、一些常量，以及一些架构相关的特殊语法

**标准库的支持** 主要需要处理的是系统调用和内嵌汇编，以及某些在 RISC-V 上不再满足的假设

**一些周围设施** 比如某些被广泛使用的语言库（可选）

# 两种主要的情况



- 使用 LLVM 作为代码生成后端
  很多现代新设计的语言采用了这种方案
- 使用自己编写的代码生成后端

# 两种主要的情况



- 使用 LLVM 作为代码生成后端
  很多现代新设计的语言采用了这种方案
- 使用自己编写的代码生成后端
  比较困难，主要问题在于：
  - 你需要自行编写代码生成后端
  - 直到你完成编译器后端的移植，你都无法通过测试编译来排查需要移植的地方

# 一些共性的问题



- 这些语言大多没有移植文档，我们需要自行确定有什么东西需要移植
- 大多数语言的编译器和标准库都有测试，他们是帮助我们测试移植正确性的有效工具
- 如果有集成测试，这将是我们确定代码哪部分出现问题的有效工具

# 使用 LLVM 作为代码生成后端



可以分为以下几个步骤

1. 首先添加相关常量和代码路径，使之能产生 RISC-V 代码
2. 然后，尝试编译，观察编译器和标准库在编译时有什么问题和缺少的东西
3. 尝试移植缺少的部分和补充修改错误的假设
4. 继续尝试第二步，如此迭代，直到没有编译问题
5. 尝试运行单元测试和集成测试，修复问题，直到测试完全通过

# 使用 LLVM 作为代码生成后端



—些其他注意事项：

- 写代码时记得关注支持的 LLVM 版本，不要写出不支持的语法
- LLVM 从 9 开始才支持 RISC-V，如果版本不满足记得做适当处理

# 使用自己编写的代码生成后端



与 LLVM 基本相同，不同的是第一步变为添加 RISC-V 代码生成后端

# 寻找需要的信息



- 编译器代码在哪里
- 标准库代码在哪里
- 编译器依赖标准库的哪些部分
- 集成测试和单元测试在哪里

# 编译器部分



首先我们根据报错信息，我们找到这段代码，可见我们需要在这里添加 `riscv64` 的分支，并且实现从 `LLVM.init_riscv64` 开始的一系列调用链。

```
def to_target_machine(cpu = "", features = "", release = false,
  code_model = LLVM::CodeModel::Default) : LLVM::TargetMachine
  case @architecture
  when "i386", "x86_64"
    LLVM.init_x86
  when "aarch64"
    LLVM.init_aarch64
  when "arm"
    LLVM.init_arm
  else
    raise Target::Error.new("Unsupported architecture for target triple: #{self}")
  end

...
end
```

这部分比较简单，主要是照猫画虎地添加 RISC-V branch。

# 标准库部分



然后我们可以进入标准库部分，参考报错信息，和架构名关键词，我们可以知道至少有以下几个部分需要处理

| | |
|---|---|
| ABI | src/llvm/abi@crystal-lang/crystal |
| syscall | src/syscall@crystal-lang/crystal |
| fiber | src/fiber@crystal-lang/crystal |
| libc | src/lib_c@crystal-lang/crystal |
| ELF | src/crystal/elf.cr@crystal-lang/crystal |

# syscall



其中，`syscall` 部分最为简单，由于在很多架构上，Linux 使用的系统调用号是一致的[1][2]，比如 AArch64 和 RISC-V，因此我们只需要参照系统调用传参表修改寄存器和系统调用指令即可。该表可于 `syscall(2)`[3] 找到

[1] https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/unistd.h

[2] https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html

[3] https://man7.org/linux/man-pages/man2/syscall.2.html

# ABI



ABI 主要参考 Calling Conventions[4] 和 ELF[5]，可以额外参考张四维- RISC-V psABI 简介及其在 LLVM 中的实现。

注意这里实现了一部分判断函数和 enum，这部分可以参考 Rust 的实现[6]和 LLVM 的实现[7]。

---

[4] https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc

[5] https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc

[6] https://github.com/rust-lang/rust/blob/master/compiler/rustc_target/src/abi/call/riscv.rs

[7] https://github.com/llvm/llvm-project/blob/8e780252a7284be45cf1ba224cabd884847e8e92/clang/lib/CodeGen/TargetInfo.cpp#L9311-L9773

# libc



libc 部分难度较低，但是工作量比较大，主要任务是筛除掉 RISC-V 上没有支持的功能。
基本就是对着 glibc 改就好。
注意其中的 signal 部分，以常见架构为基础修改较好，其他架构可能编号并不一致。具体区别可以参考 `signal(7)`[8]

[8] https://man7.org/linux/man-pages/man7/signal.7.html

# fiber



fiber 部分主要难度取决于你对 RISC-V 的寄存器结构和上下文切换的了解，主要组成部分是一个 `makecontext` 和一个 `swapcontext` 函数。可以参考上下文切换的资料和 张四维-RISC-V psABI 简介及其在 LLVM 中的实现

# ELF



参考 riscv64 上的 `/usr/include/elf.h` 的内容添加相关常量。

# 周围设施部分



写完代码之后，不要忘记添加 CI 和文档。

# 小插曲



在我写完这篇文章之后不久，我发现有人给 Crystal PR 了 WASM 支持，所以通过别人的 PR 寻找线索也是个好主意。



# 谢谢大家