NEMU 框架选讲 (2): 代码导读

蒋炎岩

南京大学　　计算机科学与技术系　　计算机软件研究所

本讲概述

就算理解了构建的过程，NEMU 代码依然很难读?

- NEMU 代码导读
  - 浏览源代码
  - 启动代码选讲
  - 编辑器配置

RTFSC

拿到源代码，先做什么？

> NEMU 对大部分同学来说是一个“前所未有大”的 project。

先大致了解一下

- 项目总体组织
  - `tree` 要翻好几个屏幕
  - `find . -name "*.c" -o -name "*.h"` (110+ 个文件)
- 项目规模
  - `find ... | xargs cat | wc -l`
  - 5,000+ 行 (其实很小了)

尝试阅读代码：从 main 开始

C 语言代码，都是从 `main()` 开始运行的。那么哪里才有 `main` 呢？

- 浏览代码：发现 `main.c`，估计在里面
- 使用 IDE (vscode: Edit → Find in files)

The UNIX Way (无须启动任何程序，直接查看)

```
grep -n main $(find . -name "*.c") # RTFM: -n
find . | xargs grep --color -nse '\<main\>'
```

Vim 当然也支持

```
:vimgrep /\<main\>/ **/*.c
```

- 浏览 : cn, : cp, ...

main()

比想象中短很多……

```
int main(int argc, char *argv[]) {
  init_monitor(argc, argv);
  engine_start();
  return is_exit_status_bad();
}
```

Comments

- 把 argc, argv 传递给另一个函数是 C 的 idiomatic use
- `init_monitor` 代码在哪里？
  - 每次都 grep 效率太低
  - 需要更先进的工具 (稍候介绍)

parse_args()

这个函数的名字起的很好，看了就知道要做什么

- 满足好代码不言自明的特性
  - 的确是用来解析命令行参数的，`-b`, `-l`, ...
  - 使用了 `getopt` ➞ RTFM!

失败的尝试：`man getopt` ➞ getopt (1)

成功的尝试

- 捷径版：STFW “C getopt” ➞ 网页/博客/...
- 专业版：`man -k getopt` ➞ `man 3 getopt`

意外之喜：man 还送了个例子！跟 `parse_args` 的用法一样耶

NEMU: 一个命令行工具

The friendly source code

- 命令行可以控制 NEMU 的行为
- 我们甚至看到了 `--help` 帮助信息

如何让我们的 NEMU 打印它?

- 问题等同于: `make run` 到底做了什么
  - 方法 1: 阅读 Makefile
  - 方法 2: 借助 GNU Make 的 `-n` 选项

开始痛苦的代码阅读之旅: 坚持!

代码选讲

static inline

```
static inline void parse_args(int argc, char *argv[]) { ... }
```

`parse_args` 函数是 static, inline 的，这是什么意思？

- inline (C99 6.7.4 #5): Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined. (inline更多有趣的行为请大家RTFM)
- static (C99 6.2.2 #3): If the declaration of a file scope identifier for an object or a function contains the storage- class specifier **static**, the identifier has internal linkage.

联合使用

- 告诉编译器符号不要泄露到文件 (*translation unit*) 之外。

更多关于 static inline (1)

我们都知道，如果在两个文件里定义了重名的函数，能够分别编译，但链接会出错：

```c
/* a.c */ int f() { return 0; }
/* b.c */ int f() { return 1; }
```

b.c:(.text+0x0): multiple definition of *f*; a.c:(.text+0xb): first defined here

这也是为什么不在头文件里定义函数的原因

- 两个 translation unit 同时引用，就导致 multiple definition
- 思考题：为什么 C++ 能把 class 都定义到头文件里？？？像 `vector` 的实现就是直接粘贴进去的

更多关于 static inline (2)

如果你的程序较短且性能攸关，则可以使用 static inline 函数定义在头文件中。例子 (**/x86/**/reg.h)：

```c
static inline int check_reg_index(int index) {
  assert(index >= 0 && index < 8);
  return index;
}
```

check_reg_index完全可以单独放在一个 C 文件里，头文件中只保留声明：

```c
int check_reg_index(int index);
```

- 但这样会导致在编译时，编译出一条额外的 call 指令 (假设没有 **LTO**)
- 使用 inline 可以在调用 check_reg_index(0) 编译优化成零开销

新的问题：啥是 assert？

```
#define assert(cond) if (!(cond)) panic(...);
```

注意特殊情况：

```
if (...) assert(0); // 上面的assert对么？
else ...
```

```
#define assert(cond) \
  do { \
    if (!(cond)) { \
      fprintf(stderr, "Fail @ %s:%d", __FILE__, __LINE__); \
      exit(1); \
    } \
  } while (0)

#define assert(cond) ({ ... })
```

千辛万苦.....

之后的历程似乎就比较轻松了。有些东西不太明白(比如 `init_device()`)，但好像也不是很要紧，到了 `welcome()`:

```
static inline void welcome() {
  ...
  printf("Welcome to \33[1;41m\33[1;33m%s\33[0m-NEMU!\n",
    str(__ISA__)); // bad code! jyy doesn't like it.
}
```

哇，还能顺带打印出编译的时间日期，奇怪的知识又增加了！

- 初始化终于完成
- 啊.....根本没碰到核心代码

理解代码：更进一步

来自同学的反馈

> “我决定挑战自己，坚持在命令行中工作、使用 Vim 编辑代码。但在巨多的文件之间切换真是一言难尽。”

上手以后还在用 grep 找代码？

- 你刚拿到项目的时候，grep 的确不错
- 但如果你要一学期在这个项目上，效率就太低了
  - 曾经有无数的同学选择容忍这种低效率

Vim: 这都搞不定还引发什么编辑器圣战

Marks (文件内标记)

- `ma`, `'a`, `mA`, `'A`, ...

Tags (在位置之间跳转)

- `:jumps`, `C-]`, `C-i`, `C-o`, `:tjump`, ...

Tabs/Windows (管理多文件)

- `:tabnew`, `gt`, `gT`, ...

Folding (浏览大代码)

- `zc`, `zo`, `zR`, ...

更多的功能/插件

- (RTFM, STFW)

VSCode: 现代工具来一套?

刚拿到手，VSCode 的体验并不是非常好

- 满屏的红线/蓝线
  - 因为 Code 并知道 NEMU 是怎么编译的
  - IDE “编译运行” 背后没有魔法
- 另一方面，这些东西一定是可以配置的
  - 配置解析选项: `c_cpp_properties.json`
    - 解锁正确的代码解析
  - 配置构建选项: `tasks.json`
    - 解锁 make (可跟命令行参数)
  - 配置运行选项: `launch.json`
    - 解锁单步调试 (我们并不非常推荐单步调试)

插入福利：调试 Segmentation Fault

听说你的程序又 Segmentation Fault 了？

- ~~百度 Segmentation Fault 得到的首个回答的解释是完全错误的~~
- 正确的解释
  - 指令越权访问内存 (r/w/x)
    - 原因很多，数组越界、memory corruption, ...
  - 指令未被执行，进程收到 SIGSEGV 信号
    - 默认的信号处理程序会 core dump 退出

好的编辑器：也许不是万能的

exec.c 也太难读了吧 (~~元编程，害死人~~)

```
static inline def_EHelper(gp1) { // ???
  ...
  EMPTY(0)
    // EMPTY(idx)  =>  EX(idx, inv)
    // EX(idx, inv)  =>  EXW(idx, inv, 0)
    // !@%#&%^!#@&%!^@%#$%*^!#@*
}
```

产生“这是什么操作”的困惑：

- 办法 1: RTFM + RTFSC + 写小程序尝试
- 办法 2: 预编译以后的代码应该好理解！
  - 还记得我们对 Makefile 的导读吗？
    - (说的容易做得难。直接 gcc -E 不是编译错误吗……)

Don't Give Up Easy

我们既然知道 Makefile 里哪一行是 .o ➡ .c 的转换

- 我们添一个一模一样的 `gcc -E` 是不是就行了？

```
$(OBJ_DIR)/%.o: src/%.c
    @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
    @$(CC) $(CFLAGS) $(SO_CFLAGS) -E -MF /dev/null $< | \
      grep -ve '^#' | \
      clang-format - > $(basename $@).i
```

敲黑板：征服你畏惧的东西，就会有意想不到的收获。

总结

怎样读代码？

读代码 ≠ “读” 代码

- 用正确的工具，使自己感到舒适
- 但这个过程本身可能是不太舒适的 (走出你的舒适区)
  - 我们看到太多的同学，到最后都没有学会使用编辑器/IDE
    - 要相信一切不爽都有办法解决

信息来源

- 在 `/etc/hosts` 中屏蔽百度
- 去开源社区找 tutorials
  - 例子：**vim-galore**, **awesome-c**

End.