PA 寻宝

李晗

2025-11-04

南京大学 计算机学院

目录

目录

读项目的几种境界

- 读 API 文档，配置好参数能用就行。
- 读源码，知道 API 背后发生了什么。
- 读设计，知道为什么代码会这样写。

PA 的价值远比它表面上看上去大得多，希望大家多多参悟。

读代码的工具

~~放弃 vim，加入 VS Code~~

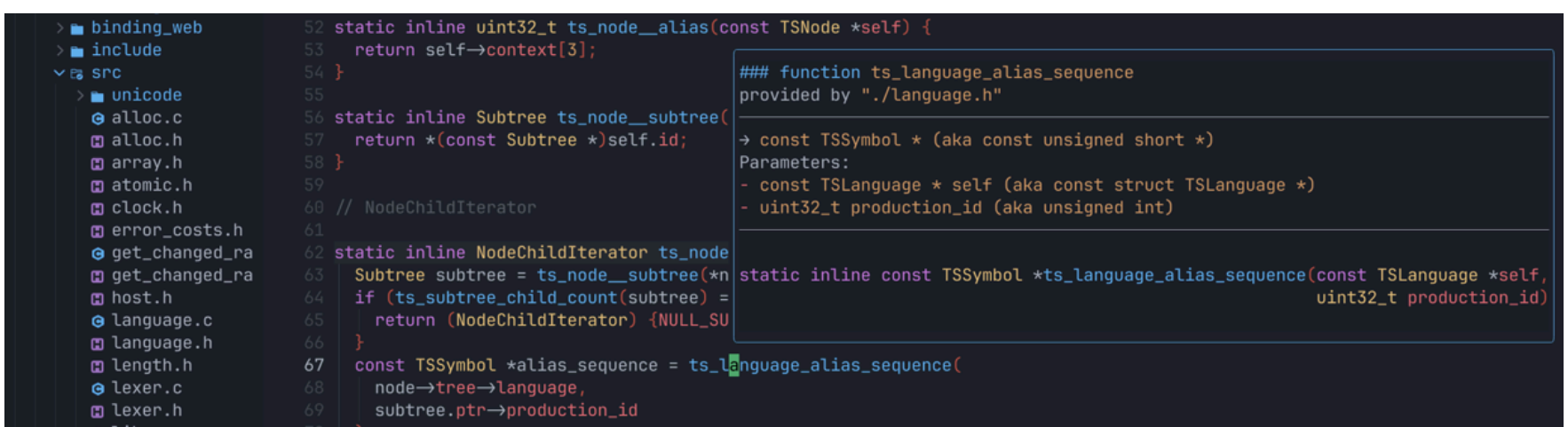
Vim	VS Code
• 几乎所有 Linux 系统自带	• SSH、WSL 插件可以无痛连接服务器
• 轻量级，启动速度快，支持多种语言	• 启动速度中等，占用较多资源
• 需要非常复杂的配置和插件，配置存储在服务器端	• 配置简单直观，配置存储在本地且可多设备同步
• 用于临时编辑服务器文件	• 用于长期开发、部署、维护

课程资料里 VS Code 插件已经教大家如何配置一个说的过去的开发环境。

读代码的工具

如果你还是坚持使用 vim，至少要做到以下几点：

- LSP 功能：语法高亮、代码补全、错误提示、符号跳转、重构……
- 光标移动：中文分词、搜索跳转、智能选择……
- 窗口管理：文件树、标签页、终端、分栏、浮动窗口……



阅读过程中要有产出

- 使用 devcontainer 声明开发环境，可以快速在其他服务器上复现完全相同的开发环境。
  - 其他常用开发环境管理工具：python - uv，java - sdkman，C - conan。
- 及时在笔记记录自己的学习过程和思考。归档中途遇到的问题和解决方案，形成一套完善的知识库。

这些不止是 PA 实验中的建议，平时的学习和以后工作中也同样适用。

读哪些文档

原则：一定要注意信息来源的可靠性！其他学生的博客、GPT 都有可能犯错。

RISC-V Manual (Unprivileged, Privileged, UnifiedDB, Documentation for RV64)

Newlib Document　　OS Standards　　POSIX.1-2024

最经典的问题：

- PA2 中的 `mulh`，`mulhu`，`mulhsu` 是有符号还是无符号，`div`，`divu`，`rem`，`remu` 除数是 0 的行为是什么？
- PA3 中 `ecall` 抛出的异常编号到底是多少？Abstract Machine 里的 `yield` 和 `Event` 应该怎么实现？
- PA3 中 libndl 的像素格式是什么？newlib 对 `open` 等系统调用的约定是什么？

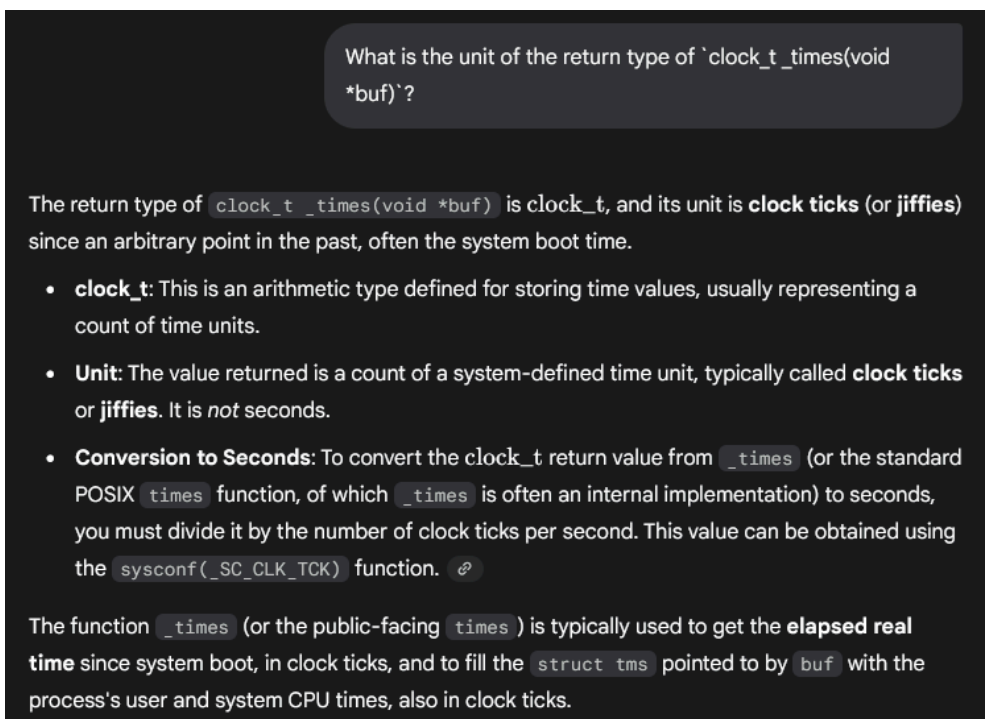
例子

navy-apps/libos 中有一个接口函数 `clock_t _times(void *buf)`。这个函数的参数和返回值分别是什么？

1. 根据 PA 手册或者“Go to References”发现这个函数只被 newlib 调用，大概率是 newlib 的一部分。
2. 前往 Newlib Document，确认这个函数属于 `newlib` 定义的操作系统接口。在 System Calls 一节中发现 “complies with the POSIX.1 standard (also known as IEEE 1003.1)”，得知 `_times` 应该符合 POSIX 标准。
3. 前往 POSIX 相关页面，最终找到 `clock_t times(struct tms *buffer)`。这个页面说明了 `tms` 结构体的定义和字段含义，但仍未明确给出返回值的单位。
4. 继续查阅文档，得知返回值的单位是“clock ticks”，而每秒有多少个 clock ticks 由 `CLOCKS_PER_SEC` 定义，通常为 1,000,000。

例子

~~另一种方法~~


What is the unit of the return type of `clock_t _times(void *buf)`?
The return type of clock_t _times(void *buf) is clock_t, and its unit is clock ticks (or jiffies) since an arbitrary point in the past, often the system boot time.
clock_t: This is an arithmetic type defined for storing time values, usually representing a count of time units.
Unit: The value returned is a count of a system-defined time unit, typically called clock ticks or jiffies. It is not seconds.
Conversion to Seconds: To convert the clock_t return value from _times (or the standard POSIX times function, of which _times is often an internal implementation) to seconds, you must divide it by the number of clock ticks per second. This value can be obtained using the sysconf(_SC_CLK_TCK) function.
The function _times (or the public-facing times) is typically used to get the elapsed real time since system boot, in clock ticks, and to fill the struct tms pointed to by buf with the process's user and system CPU times, also in clock ticks.

目录

宏观视角下的 PA

课程上 NEMU 框架代码导读已经带大家以微观视角出发，讲解 NEMU 是如何从 `main` 函数一步步执行到 `parse_args`、 `init`、并最终到关键的 `isa_exec`。

宏观视角下 NEMU 的架构设计其实也有很多值得花时间去研究的地方。

如果忽略 NEMU 现在已有的实现，你会怎么设计实现一个同时支持多种架构的全系统模拟器?

宏观视角下的 PA

一个从零开始的 NEMU：

- CPU 模块：抽象出不同架构的 CPU 特性，设计统一的接口供上层使用。
  - 需要考虑指令集架构的差异，包括寄存器数量、指令格式、内存映射……
- 设备模块：抽象出不同设备特性，设计统一的接口接入总线。
  - 在这种抽象下，内存也是在总线上的一种“平凡”的设备。
  - 多数设备拥有自己的寄存器和内存映射，读写寄存器会产生比内存复杂得多的行为。
- 监视器模块：管理 NEMU 自身的行为，例如参数处理、系统初始化、用户界面交互等。

宏观视角下的 PA

对比 NEMU / QEMU 的设计，有哪些不一样的地方，这些差异分别适用于哪些场景？

```
nemu/
├─ cpu/
│   └─ riscv/
│       ├─ instr/
│       ├─ csr/
│       └─ mmu/
├─ device/
│   ├─ uart/
│   ├─ virtio/
│   └─ bus/
└─ monitor/
```

宏观视角下的 PA

对比 NEMU / QEMU 的设计，有哪些不一样的地方，这些差异分别适用于哪些场景？

```
nemu/
├─ cpu/
│   └─ riscv/
│       ├─ instr/
│       ├─ csr/
│       └─ mmu/
├─ device/
│   ├─ uart/
│   ├─ virtio/
│   └─ bus/
└─ monitor/
```

```
qemu/
├─ target/
│   └─ riscv/
│       ├─ insn_trans/
│       ├─ csr.c
│       └─ cpu-helper.c
├─ hw/
│   ├─ uart/
│   └─ virtio/
└─ system/
    └─ physmem.c
```

宏观视角下的 PA

```
qemu/
├─ target/
│  └─ riscv/
│     ├─ insn_trans/
│     ├─ csr.c
│     └─ cpu-helper.c
├─ hw/
│  ├─ uart/
│  └─ virtio/
└─ system/
   └─ physmem.c
```

Where is the phycical address to device (hw) dispatch function?

40 references

The physical-address → device dispatch logic is implemented in QEMU's physical memory / memory-region subsystem. The core dispatch/mapping code is in:

system/physmem.c

```
/* system/physmem.c — builds AddressSpaceDispatch/map.sections and contains
   the code that walks the physical address map (MemoryRegionSection array).
   The print helper mtree_print_dispatch is here, and the dispatch/lookup
   implementation lives in this file where MemoryRegionSection entries are
   managed and looked up. */
```

You can also see related interfaces used by other subsystems and plugins:

include/qemu/qemu-plugin.h

```
/* qemu_plugin_hwaddr_phys_addr() and related helpers — expose the physical
   address / device information to plugins. This header shows the public
   query surface for obtaining a physical address and device name for a
   memory operation. */
```

宏观视角下的 PA

其实还有很多可以扩展的地方：

- AM $\longleftrightarrow$ OpenSBI / Zephyr / RT-Thread / Linux
  - `am/src/` $\longleftrightarrow$ `arch` / `driver` / `board`
  - `klib/` $\longleftrightarrow$ `libc`
- NanOS $\longleftrightarrow$ Zephyr / RT-Thread / Linux
- Navy $\longleftrightarrow$ Newlib / Compiler-RT / SDL2

从此之后不再惧怕任何大型项目了，无非就是多花点时间对应上我们现在假想的设计。

宏观视角下的 PA

可以对比的也不只是宏观架构，还可以是项目配置方式：

	Makefile	Kconfig	Configure	CMake	SCons
NEMU	✓	✓			
Linux	✓	✓			
QEMU		✓	✓		
RT-Thread		✓			✓
Zephyr				✓	

类似的工具还有 Ninja、Meson、Bazel 等。

宏观视角下的 PA

思考题：为什么只有 C 语言项目才会涉及这么繁多的配置方式？Java 语言要配置的时候是怎样的？

宏观视角下的 PA

思考题：为什么只有 C 语言项目才会涉及这么繁多的配置方式？Java 语言要配置的时候是怎样的？

- C 语言的配置工具的场景大多是构建时配置，用于裁剪需要编译的代码和资源。
  - 使用宏来控制运行时行为也有，但是会让代码非常难看，我个人不推荐。

```
#ifdef CONFIG_RVH
static inline bool check_permission(PTE *pte, bool ok, vaddr_t vaddr, int type, int virt, int mode) {
bool ifetch = (type == MEM_TYPE_IFETCH);
#else
static inline bool check_permission(PTE *pte, bool ok, vaddr_t vaddr, int type) {
  bool ifetch = (type == MEM_TYPE_IFETCH);
  uint32_t mode = (get_mprv() && !ifetch ? mstatus->mpp : cpu.mode);
#endif
  assert(mode == MODE_U || mode == MODE_S);
  ok = ok && pte->v;
  ok = ok && !(mode == MODE_U && !pte->u);
  ok = ok && (!pte->n || (pte->ppn & SVNAPOTMASK) == 0b1000);
#ifdef CONFIG_RVH
  ok = ok && !(pte->u && ((mode == MODE_S) && (!(virt? vsstatus->sum: mstatus->sum) || ifetch)));
  Logtr("ok: %i, mode == U: %i, pte->u: %i, ppn: %lx, virt: %d", ok, mode == MODE_U, pte->u, (uint64_t)pte->ppn << 12, virt);
#else
  ok = ok && !(pte->u && ((mode == MODE_S) && (!mstatus->sum || ifetch)));
  Logtr("ok: %i, mode: %s, pte->u: %i, a: %i d: %i, ppn: %lx ", ok,
        mode == MODE_U ? "U" : MODE_S ? "S" : MODE_M ? "M" : "NOTYPE",
        pte->u, pte->a, pte->d, (uint64_t)pte->ppn << 12);
#endif
  if (ifetch) {
    Logtr("Translate for instr reading");
#ifdef CONFIG_SHARE
//  update a/d by exception
    bool update_ad = !pte->a;
    if (update_ad && ok && pte->x)
      Logtr("raise exception to update ad for ifecth");
#else
    bool update_ad = false;
#endif
```

宏观视角下的 PA

思考题：为什么只有 C 语言项目才会涉及这么繁多的配置方式？Java 语言要配置的时候是怎样的？

- C 语言的配置工具的场景大多是构建时配置，用于裁剪需要编译的代码和资源。
  - 使用宏来控制运行时行为也有，但是会让代码非常难看，我个人不推荐。
- 同为系统级编程语言的 Rust 和 Zig 则使用 feature 标记提供类似的构建时配置。
- Java 语言的配置工具则更多地是运行时配置，如 Spring Boot 依赖于反射和注解等特性来实现动态配置。

PA 作为思维模型

我们除了将 PA 视为一个大型项目学习框架设计，还可以将 PA 作为一种思维模型，帮助我们更好地理解和分析其他系统的设计。

比方说 7 月底出现了这样一条新闻：
腾讯重拳打击 DMA 外挂！《三角洲行动》宣布 CPU 虚拟化正式接入 AMD 平台

我们马上就能从中捕捉到很多关键词并生成问题：

- DMA 是什么？
- DMA 为什么能够用于外挂？
- CPU 虚拟化是什么？
- CPU 虚拟化为什么能够防止或检测外挂？

AI 的回答

PA 作为思维模型

但这种解释还不够！DMA 到底是怎么发起的，IOMMU 又是什么？反外挂检测程序如何设置硬件状态以检测外挂的 DMA 访问？

经历了 PA 之后，你心里应该一直装着一个 PA 框架。有想要验证的问题时应该很快能从 PA 的架构出发，找到相关的设计文档和真实实现：

1. 找标准：RISCV IOMMU Specification 规定了 DMA 流程和 IOMMU 硬件接口。
2. 硬件设计：NEMU 没有，看 QEMU 也是一样的。QEMU DMA 实现、IOMMU 实现
3. 软件协同：NanOS 没有，看 Linux 也是一样的。Linux IOMMU 子系统

至此，终于可以很有把握地说我已经掌握了 DMA 和 IOMMU 的相关知识，~~随时可以手搓一个外挂程序甚至反反外挂程序了~~。

发散！发散！

PA 里面还有很多支线任务，每完成一个都能提供巨额的经验值，一定不要错过。

- 60+ 支线任务，涵盖计算机系统的方方面面。~~学透之后终身免疫 C++ 八股~~。
- Lab1、Lab2、期末编程测试都取自这些支线任务。

发散！发散！

PA2 中提到 `INSTPAT("??????? ????? ????? ??? ????? 00101 11", auipc, U, ...);` 宏会被展开成下面这样。看上去 `pattern_decode` 存在字符串解析会很慢？

```
do {
  uint64_t key, mask, shift;
  pattern_decode("??????? ????? ????? ??? ????? 00101 11", 38, &key, &mask, &shift);
  if ((((uint64_t)s->isa.inst >> shift) & mask) == key) {
    decode_operand(s, &rd, &src1, &src2, &imm, TYPE_U);
    // ...
    goto *(__instpat_end);
  }
} while (0);
```

发散！发散！

pattern_decode 内部大量使用宏展开成 64 个 if 语句字符串每一位判断，最终会被编译器内联优化成赋值语句。（PA1+：编译优化是双刃剑，之后专门有一节课会讲编译优化）

```
int main() {
  uint64_t key, mask, shift;
  pattern_decode(
    "??????? ????? ????? ??? ????? 00101 11",
    38, &key, &mask, &shift);

  printf("key = %08x, mask = %08x, shift = %d\n", key,
mask, shift);
}
```

```
.LC0:
    .string "key = %08x, mask = %08x, shift = %d\n"
main:
    sub     rsp, 8
    xor     ecx, ecx
    mov     edx, 127
    mov     esi, 23
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

所以未来做 PA2+ 性能优化的时候就不要打 decode 的主意了，~~除非你能缓存 decode 结果~~

发散！发散！

等等，缓存 decode 结果？

很明显函数调用和循环等场景都会反复地执行一系列固定的指令序列，并且很多时候循环占了大部分时间。虽然 `pattern_decode` 已经被优化成了常数，但是每个 `INSTPAT` 仍然会展开成一个很复杂的 `if ((inst >> shift) & mask) == key)`，而经历了一系列判断之后，真正需要执行的可能只是一个 `Reg(rd) = Reg(rs1) + Reg(rs2)`。

你完全可以利用 RISC-V 指令最低两位为 `11` 和函数指针最低两位为 `00` 的特性，在第一次执行时把指令翻译成对应 exec 函数的函数指针，后续就能跳过 decode 直接执行了！

恭喜，你发明了最简单的 JIT（Just-In-Time Compilation）！

不要觉得这是一件很复杂的事情，也不用因为它简单就觉得没有用处。
Python 在 3.13 版本引入了类似的 JIT 方案，其实并没有比我们刚刚发明的方案复杂多少。

发散！发散！

宏其实是 C 语言中很重要的一个特性，它为 C 语言提供了唯一的“元编程能力”。

- 编程：输入数据、输出数据
- 元编程：代码也是一种数据，输入代码、输出代码

宏仅使用文本替换，提供了接近图灵完备的编程能力。

- 说接近图灵完备是因为 C 语言给宏加了一些限制：
  - 宏不能递归：避免了 `#define f(x) f(x)`
  - 已经完全展开的宏不会参与后续展开：例子
- C Macro Explainer

发散！发散！

元编程能力太重要了，但是 C 的宏又太难用了。

元编程强需求场景：序列化、ORM。在这些场景下 C/C++ 至今没有一个好用的库。

其他语言是怎么实现元编程的？

- C++ 提供了模板 template 和编译期计算 constexpr，如编译期正则
- Java 提供了注解处理器 Annotation Processor 和反射代理 Proxy，如 Lombok 的 `@Data`
- Python 提供了装饰器 Decorator 和魔法方法 Magic Method，如 pytorch `@torch.compile`
- Rust 提供了卫生宏 `macro_rules` 和过程宏 `#[proc_macro]`，如 `#[serde]`

发散！发散！

交叉编译（Cross Compiling）其实也比大家想的要复杂的多，~~都怪 x86 平台和 JVM 等虚拟机把大家惯的太好了~~。

- riscv64-linux-gnu 这个前缀是什么意思？为什么它能够编译 rv32im 的程序？为什么要在 PA 里让大家修改 `lp64` 相关的头文件？
  - PA 需要的工具链实际上是 `riscv32-unknown-elf --with-arch=rv32im_zicsr --with-abi=ilp32`，如果你自己编译了这个工具链就可以直接使用。
- 如果哪一天领导突然派一个任务，要求你开发的软件必须跑在国产 CPU（LoongArch）上，你应该怎么选择工具链？
  - 如果领导还进一步要求全链条自主可信（人话：不能用任何现成二进制，Linux kernel、Grub、GCC、Java 等所有软件包都要审查源码后自己编译），你能不能完成？

发散！发散！

所有上面这些发散出来的支线任务，其实都只是来源于 PA2.2 一个小节中的一部分思考题。因此二周目三周目的大有人在，并且真的能学到不少东西。除此之外还有 Linux、一生一芯等进阶大型副本等你挑战。

这门课的目的不止是尝试教会大家如何写代码，而是要让大家理解代码背后的原理，体会手册背后的设计。相信“CPU 是个状态机”，认可“机器永远是对的”，从此不再惧怕编程中的各种问题。

目录

PA1

PA1 中我们让大家实现了一个简单的表达式求解器，~~其实是个 leetcode 中等~~。

而这道题在 LeetCode 上有着相当简洁的解法，即使加上 &&、*x 语法后也就 100 行。

编译原理的语法制导翻译在这个问题上更是大材小用（Lab1 提供的 calc 就是基于 bison/flex 写的）。

算法上没有什么好讲的，主要讲讲工程上的事情。

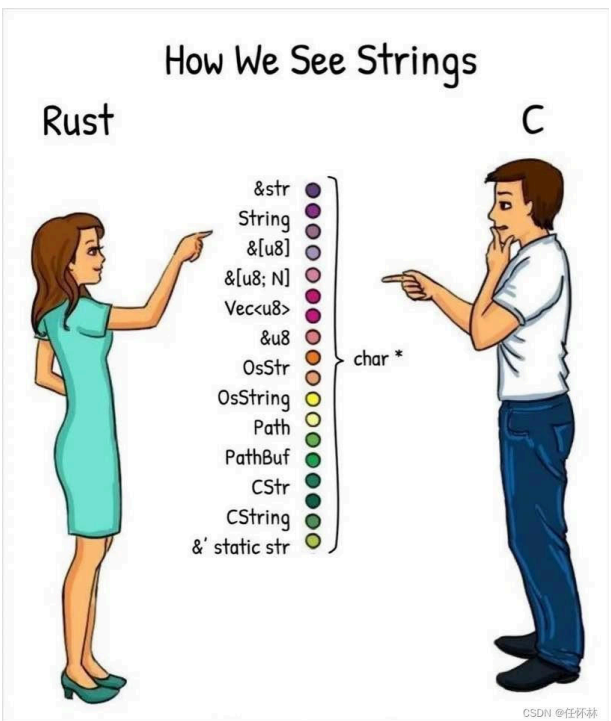
PA1

字符串

字符串的坑其实比大家想的要多得多

- C 语言使用 `\0` 表示字符串的结尾，那如何表示包含 `\0` 的字符串呢?
- C 语言字符串字面量 `"Hello"` 的类型是 `char *`，类型上竟然允许修改?
- 一个字符串的声明周期究竟有多长，需不需要手动释放?
- `strncpy`、`snprintf` 系列函数的 `n` 究竟应该取多大?

Java 这类带 gc 的语言可以合并成一个 `&'static str`，但是 OsStr、Path、CStr 仍然难以避免（看不到 ≠ 没有开销）。



PA1

Don’t repeat yourself

高内聚低耦合，让同一个功能的相关代码尽量聚集在同一处。

如果确实难以做到，尽量保证有办法利用类型系统或运行时检查在修改完一个地方的代码之后自动提醒修改另一个地方的代码。

- PA1 某份代码（不公开）

Lab1/PA1+

来自 21 年的一语成谶

UB, 编译优化和datalab

听闻大班的lab1(datalab)曾经因为使用了debian10的新版gcc而翻车. 后来了解到, 是因为datalab的参考代码中含有int整数溢出的UB, debian 10的gcc利用了该UB进行编译优化, 导致参考代码生成了错误的参考答案.

C语言标准规定, int整数溢出的行为是未定义的, 但大部分程序员并不知道这一约定, 甚至连市面上流行的C语言教科书都认为int整数溢出的结果是wrap around. datalab是CMU设计的实验, 但原作者也会编写出含有UB的代码, 说明原作者对UB的理解也并未到位. 在旧版本的编译器中, 这些UB均未被触发. 但UB毕竟是UB, 只能说明作者写代码的时候没有充分理解C语言标准.

这篇论文对整数溢出的分类和行为进行了梳理, 并且在实际应用中找出了大量整数溢出的例子进行分析, 推荐大家阅读. 论文中提到有一个被广泛应用(包括Office和Windows)的函数库SafeInt用于避免整数溢出, 但这个函数库自身的代码就被论文作者检测出整数溢出导致的UB, 可谓是 `SafeInt is not safe` .

这些例子给我们的启示是: 我们不仅需要编写通过测试的代码, 而且需要编写符合语言规范的well-defined的代码. 退一步讲, 人都会犯错误, 但我们至少要在出错的时候知道, 什么才是对的.

Lab1/PA1+

问题在于虽然 RISC-V 手册明确规定了除以零的行为，但是 GCC 不认可，认为是未定义行为。因为 GCC 需要适配各种架构，例如 x86 手册就明确规定了除以零需要抛出异常。

Undefined Behavior 是相当危险的，它的含义不止是在不同的平台下可能有不同的行为那么简单，而是字面意思上的任何事情都有可能发生。真实面经

其实除了 Undefined Behavior，还有 Implementation Defined Behavior 和 Unspecified Behavior。如果除以零是 Implementation Defined Behavior，就不会出锅了。

~~解决方案：不要用 GCC 作为 reference~~

Lab1 原本的考点是利用一个合适的 Reference Implementation 做 Differential Testing，不是如何解决 GCC UB。明年如果还有这道题，我们会提供 calc 程序作为 reference。

Lab1/PA1+

建议：

- 减少 copy-paste。Lab1 的某份代码（不公开）

Lab1/PA1+

建议：

- 减少 copy-paste。Lab1 的某份代码（不公开）
- 不要太过随机，稍微一点点硬编码能大幅提升生成的表达式的质量。
  - 已知 OJ 要求要生成长度为 10000 的表达式：
    - 调整递归生成概率让长度精准达到 10000 而不超相当困难。
    - 但是把剩余需要生成的长度当成参数穿进去，动态调整结束概率就容易多了。
  - 已知 OJ 要求有固定比例的表达式必须包含乘除法、除以 0 等
    - 我已经在代码里按照概率调用生成函数了，但是专门随机也没随机出来怎么办？
    - 丢掉不符合要求的表达式重新生成不就好了。

总结

多读书、多看手册、多看代码

- 完整的知识体系 ≫ 零散的片段，原始文档 ≫ 转载
- PA 不止表层的实验，还有大量架构设计和工程实践可以参悟。

选做支线任务

- PA 手册里全是宝藏：gdb script，santitizer，{i,f,s}trace……

可维护的项目

- 100k+ 的代码量，需要保持代码简洁易懂，让代码自己解释自己