



# 课程实验报告

## 0x00 环境搭建与实验准备

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 3 月 6 日

# 目录

<b>1 Rust 编程实践 .....</b>	<b>3</b>
1.1 任务一：文件读取 .....	3
1.2 任务二：代码单元测试 .....	5
1.3 任务三：彩色文字终端 .....	6
1.4 任务四：枚举类型 .....	6
1.5 任务五：元组结构体 .....	7
<b>2 YSOS，启动! .....</b>	<b>8</b>
2.1 年轻人的第一个 UEFI 程序 .....	8
<b>3 思考题 .....</b>	<b>10</b>
3.1 现代操作系统的启动流程 .....	10
3.1.1 Windows 系统启动过程解析 .....	10
3.1.2 UEFI 与 Legacy BIOS 的技术差异 .....	10
3.2 Makefile 过程 .....	11
3.3 Cargo 包管理工具 .....	12
3.4 #[entry] 与 main .....	12
<b>4 附加题 .....</b>	<b>12</b>
4.1 彩色的日志 .....	12
4.2 Rust 实现简单 shell .....	14
4.3 线程模型 .....	17

## 图表

<b>图 1.1.1 任务一运行结果 .....</b>	<b>5</b>
<b>图 1.1.2 单元测试结果 .....</b>	<b>5</b>
<b>图 1.3.3 彩色文字终端 .....</b>	<b>6</b>
<b>图 2.1.4 UEFI 程序运行结果 .....</b>	<b>9</b>
<b>图 4.2.5 简单 shell 运行效果 .....</b>	<b>14</b>
<b>图 4.3.6 线程模型测试结果 .....</b>	<b>19</b>

# 1 Rust 编程实践

## 1.1 任务一：文件读取

### count\_down(seconds: u64)

该函数接收一个 u64 类型的参数，表示倒计时的秒数。函数应该每秒输出剩余的秒数，直到倒计时结束，然后输出 Countdown finished!。

利用简单的 for 循环和线程休眠实现倒计时功能，代码如下：

```
1 fn count_down(seconds: u64) {  
2     for i in (1..=seconds).rev() {  
3         println!("{}", seconds left...", i);  
4         std::io::stdout().flush().unwrap();  
5         std::thread::sleep(std::time::Duration::from_secs(1));  
6     }  
7     info!("Count down finished!");  
8 }
```

Rust

### read\_and\_print

该函数接收一个字符串参数，表示文件的路径。函数应该尝试读取并输出文件的内容。如果文件不存在，函数应该使用 expect 方法主动 panic，并输出 File not found!。

采用了 std::fs::File::open 方法打开文件，然后使用 std::io::BufReader 读取文件内容。用 error! 宏记录错误日志，并使用 ? 运算符将错误向上传播，代码如下：

```
1 fn read_and_print(file_path: &str) -> Result<(), std::io::Error> {  
2     let file = std::fs::File::open(file_path).map_err(|e| {  
3         error!("Failed to open file {}: {}", file_path, e);  
4         e  
5     })?;  
6     let reader = std::io::BufReader::new(file);  
7     for line in reader.lines() {
```

Rust

```
8         println!("{}", line?);
9     }
10    info!("File read successfully!");
11    Ok(())
12 }
```

### ☰ file\_size

该函数接收一个字符串参数，表示文件的路径，并返回一个 `Result`。函数应该尝试打开文件，并在 `Result` 中返回文件大小。如果文件不存在，函数应该返回一个包含 `File not found!` 字符串的 `Err`。

实现方式与 `read_and_print` 基本一致，代码如下：

```
1 fn file_size(file_path: &str) -> Result<u64, std::io::Error> { Rust
2     let file = std::fs::File::open(file_path).map_err(|e| {
3         error!("Failed to open file {}: {}", file_path, e);
4         std::io::Error::new(std::io::ErrorKind::NotFound, "File not
5         found!")
6     })?;
7     let mut reader = std::io::BufReader::new(file);
8     let mut buffer = Vec::new();
9     reader.read_to_end(&mut buffer)?;
10    Ok(buffer.len() as u64)
11 }
```

### ☰ Final

在 `main` 函数中，按照如下顺序调用上述函数：

1. 首先调用 `count_down(5)` 函数进行倒计时
2. 然后调用 `read_and_print("/etc/hosts")` 函数尝试读取并输出文件内容
3. 最后使用 `std::io` 获取几个用户输入的路径，并调用 `file_size` 函数尝试获取文件大小，并处理可能的错误。

将上述代码合在 main 函数中分别调用即可，运行结果如图 1.1.1 所示<sup>1</sup>。

```

__allenge@allenge: /Volumes/Workspace/Code/RustLearning/word_hello
└─> cargo run
    Finished target/debug/word_hello
5 seconds left...
4 seconds left...
3 seconds left...
2 seconds left...
1 seconds left...
[INFO]: Count down finished!
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1 localhost
255.255.255.255 broadcasthost
::1 localhost
[INFO]: File read successfully!
Enter file paths (comma separated):
./test.txt, Cargo.toml, .gitignore, haha
[INFO]: File size of ./test.txt: 95 bytes
[INFO]: File size of Cargo.toml: 164 bytes
[INFO]: File size of .gitignore: 8 bytes
[ERROR]: Failed to open file haha: No such file or directory (os error 2)
===== [ERROR]: Failed to get file size for haha: File not found! =====

```

图 1.1.1: 任务一运行结果

```

__allenge@allenge: /Volumes/Workspace/Code/RustLearning/word_hello
└─> cargo test
    note: `[warn(non_snake_case)]` on by default
warning: `word_hello` (bin "word_hello" test) generated 6 warnings (run `cargo fix --bin "word_hello" --tests` to apply 1 suggestion)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.95s
    Running unittests src/main.rs (target/debug/deps/word_hello-2feaa02bab451c31)

running 4 tests
test tests::test_area ... ok
test tests::test_unique_id ... ok
test tests::test_humanized_size ... ok
[INFO]: Hello, world!
[WARNING]: I'm a teapot!
[ERROR]: KERNEL PANIC!!!
test tests::LogTest ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

/Volumes/W/Code/RustLearning/word_hello master 75  base 23:18:53

```

图 1.1.2: 单元测试结果

## 1.2 任务二：代码单元测试

### humanized\_size

1. 将字节数转换为人类可读的大小和单位。使用 1024 进制，并使用二进制前缀 (B, KiB, MiB, GiB) 作为单位
2. 补全格式化代码，使得你的实现能够通过如下测试：

```

1  #[test]
2  fn test_humanized_size() {
3      let byte_size = 1554056;
4      let (size, unit) = humanized_size(byte_size);
5      assert_eq!("Size : 1.4821 MiB", format!("{}", size, unit));
6  }

```

Rust

一个简单的单位转换，函数的具体实现如下：

```

1  fn humanized_size(size: u64) -> (f64, &'static str) {
2      let kb = 1024;
3      let mb = kb * 1024;
4      let gb = mb * 1024;
5      let tb = gb * 1024;
6      if size < kb {
7          (size as f64, "B")
8      } else if size < mb {

```

Rust

<sup>1</sup>haha 文件并不存在，因此触发了异常处理机制

```

9      ((size as f64) / kb as f64, "KiB")
10    } else if size < gb {
11      ((size as f64) / mb as f64, "MiB")
12    } else if size < tb {
13      ((size as f64) / gb as f64, "GiB")
14    } else {
15      ((size as f64) / tb as f64, "TiB")
16    }
17  }

```

测试代码补全如下：

```

1  #[test]
2  fn test_humanized_size() {
3      let byte_size = 1554056;
4      let (size, unit) = humanized_size(byte_size);
5      assert_eq!("Size : 1.4821 MiB", format!("Size : {} {}", size, unit));
6  }

```

Rust

测试通过，运行结果见图 1.1.2。

### 1.3 任务三：彩色文字终端

利用 Colored 库实现彩色文字终端。Colored 库通过终端输出 ANSI 控制字符实现彩色文字显示。效果如图 1.1.3 所示<sup>2</sup>。

```

[INFO]: Hello, world!
[WARNING]: I'm a teapot!
===== [ERROR]: KERNEL PANIC!!! =====

```

图 1.3.3: 彩色文字终端

### 1.4 任务四：枚举类型

#### ☞ 使用 enum 对类型实现同一化

实现一个名为 Shape 的枚举，并为它实现 pub fn area(&self) -> f64 方法，用于计算不同形状的面积。

<sup>2</sup>具体代码可见 4.1

- 你可能需要使用模式匹配来达到相应的功能
- 请实现 Rectangle 和 Circle 两种 Shape，并使得 area 函数能够正确计算它们的面积
- 使得你的实现能够通过如下测试：

```
1  #[test]
2  fn test_area() {
3      let rectangle = Shape::Rectangle {
4          width: 10.0,
5          height: 20.0,
6      };
7      let circle = Shape::Circle { radius: 10.0 };
8
9      assert_eq!(rectangle.area(), 200.0);
10     assert_eq!(circle.area(), 314.1592653589793);
11 }
```

Rust

枚举类型是 Rust 中的一种复合类型，可以包含多种不同的值。在这里，我们定义了一个 Shape 枚举，包含了 Rectangle 和 Circle 两种形状。利用模式匹配，我们可以很方便地实现 area 方法，代码如下，单元测试通过<sup>3</sup>。

```
1  enum Shape {
2      Rectangle { width: f64, height: f64 },
3      Circle { radius: f64 },
4  }
5  impl Shape {
6      pub fn area(&self) -> f64 {
7          match self {
8              Shape::Rectangle { width, height } => width * height,
9              Shape::Circle { radius } => std::f64::consts::PI * radius *
10                 radius,
11          }
12      }
13 }
```

Rust

## 1.5 任务五：元组结构体

<sup>3</sup>测试结果见图 1.1.2

### 现一个元组结构体 `UniqueId(u16)`

使得每次调用 `UniqueId::new()` 时总会得到一个新的不重复的 `UniqueId`。

- 你可以在函数体中定义 `static` 变量来存储一些全局状态
- 你可以尝试使用 `std::sync::atomic::AtomicU16` 来确保多线程下的正确性（无需进行验证，相关原理将在 Lab 5 介绍，此处不做要求）
- 使得你的实现能够通过如下测试：

```
1 #[test]
2 fn test_unique_id() {
3     let id1 = UniqueId::new();
4     let id2 = UniqueId::new();
5     assert_ne!(id1, id2);
6 }
```

**Rust**

注意到 `test` 中使用了 `assert_ne!` 宏，用于判断两个值是否不相等。因此我们需要有 `PartialEq` trait 的实现，可以用 `#[derive(PartialEq)]` 来自动生成。代码如下，单元测试通过<sup>4</sup>。

```
1 #[derive(Debug, Eq, PartialEq)]
2 struct UniqueId(u16);
3 impl UniqueId {
4     fn new() -> Self {
5         static COUNTER: AtomicU16 = AtomicU16::new(0);
6         UniqueId(COUNTER.fetch_add(1, Ordering::SeqCst))
7     }
8 }
```

**Rust**

## 2 YSOS，启动！

### 2.1 年轻人的第一个 UEFI 程序

使用 QEMU 启动 UEFI Shell 后，我们可以使用 Rust Toolchain 编译 UEFI 程序。下面是一个简单的 Hello World 程序，运行结果如图 2.1.4 所示。

```
1 #![no_std]
```

**Rust**

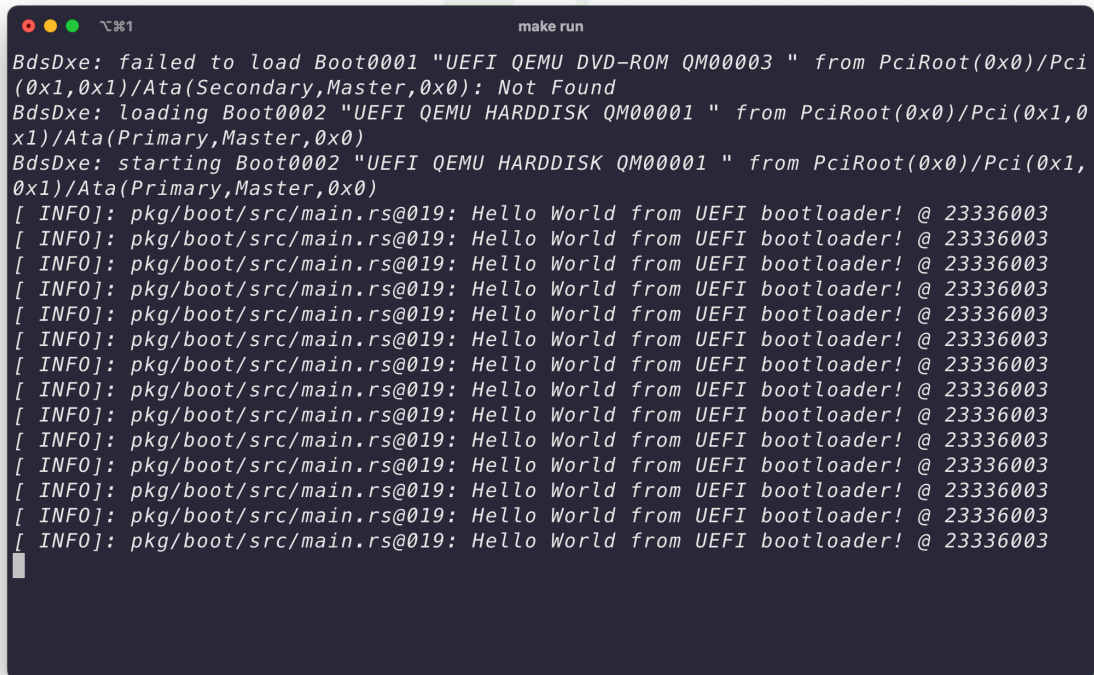
<sup>4</sup>见图 1.1.2



```

2  #![no_main]
3
4  #[macro_use]
5  extern crate log;
6  extern crate alloc;
7  use core::arch::asm;
8  use uefi::{Status, entry};
9
10 #[entry]
11 fn efi_main() -> Status {
12     uefi::helpers::init().expect("Failed to initialize utilities");
13     log::set_max_level(log::LevelFilter::Info);
14     let std_num = "23336003";
15     loop {
16         info!("Hello World from UEFI bootloader! @ {}", std_num);
17         for _ in 0..0x100000000 {
18             unsafe {
19                 asm!("nop");
20             }
21         }
22     }
23 }

```



```

make run
BdsDxe: failed to load Boot0001 "UEFI QEMU DVD-ROM QM00003 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Secondary,Master,0x0): Not Found
BdsDxe: loading Boot0002 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
BdsDxe: starting Boot0002 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003
[ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @ 23336003

```

图 2.1.4: UEFI 程序运行结果

## 3 思考题

### 3.1 现代操作系统的启动流程

#### 3.1.1 Windows 系统启动过程解析

计算机的启动过程是一个精密的多阶段协作机制。当按下电源键后，系统经历以下关键阶段：

##### 1. 电源通电自检 (POST)

电源管理系统完成供电后，固件立即执行 POST 流程。该过程会检测关键硬件组件（CPU、内存、存储控制器）的状态，并通过蜂鸣器代码或 LED 指示灯反馈检测结果。与传统 BIOS 的 16 位实模式检测不同，UEFI 直接在 32/64 位保护模式下运行，显著提升硬件初始化效率。

##### 2. 固件初始化阶段

固件在此阶段建立基本硬件抽象层：

- 枚举 PCIe 设备并分配资源
- 初始化 USB/SATA 控制器
- 构建 ACPI 表描述电源管理方案
- 创建内存映射表（UEFI 使用 GUID 分区表记录存储布局）

##### 3. 引导加载程序阶段

固件根据 NVRAM 中的引导条目定位引导分区。UEFI 引入的 ESP 分区（EFI System Partition）采用 FAT32 文件系统，可直接访问引导文件（如 \EFI\Boot\bootx64.efi），而传统 BIOS 需要从 MBR 中读取 512 字节的引导扇区代码。

##### 4. 操作系统加载阶段 接由固件加载，而 BIOS 需要通过 initrd 映像传递驱动。

##### 5. 系统初始化阶段

内核启动会话管理器（smss.exe），依次加载注册表配置、系统服务（services.exe）和用户态环境（winlogon.exe）。此时 UEFI 的安全启动（Secure Boot）机制会验证所有引导组件的数字签名，防止 rootkit 注入。

#### 3.1.2 UEFI 与 Legacy BIOS 的技术差异

架构设计方面，UEFI 采用模块化设计，支持驱动程序的动态加载。其服务分为引导服务（Boot Services）和运行时服务（Runtime Services），前者在操作系统加载后停

止，后者持续提供硬件抽象。相比之下，BIOS 工作在 16 位实模式，通过中断向量表 (IVT) 提供基本 IO 服务。

存储支持上，UEFI 的 GPT 分区方案突破 MBR 的 2TB 容量限制，支持 128 个主分区，并通过冗余分区表提供数据可靠性。GPT 头部的保护性 MBR 可防止传统磁盘工具误操作。

安全机制层面，UEFI 2.4 引入的安全启动要求所有 EFI 可执行文件必须具有可信证书签名，有效抵御引导型恶意软件。配合 TPM 芯片还可实现全盘加密密钥的安全存储。

启动性能优化方面，UEFI 通过并行硬件初始化和延迟驱动加载，可将启动时间缩短至 10 秒内。其内置的快速启动技术 (Fast Boot) 会跳过不必要的设备检测，直接复用上次启动的硬件配置快照。

发展趋势上，现代 UEFI 固件已集成网络堆栈和图形化配置界面，支持远程操作系统部署和故障诊断。而传统 BIOS 由于架构限制，正逐步退出历史舞台。

## 3.2 Makefile 过程

```
1 [+] Building: bootloader...
  [?] Executing: /Users/__allenge/.cargo/bin/cargo build --release in /
2 Volumes/WorkSpace/Code/My_OS/pkg/boot //命令行中实际调用了 Cargo build --
  release.
  [?] Would copy: /Volumes/WorkSpace/Code/My_OS/target/x86_64-unknown-uefi/
3 release/ysos_boot.efi -> /Volumes/WorkSpace/Code/My_OS/esp/EFI/B00T/
  B00TX64.EFI
  [?] Would copy: /Volumes/WorkSpace/Code/My_OS/pkg/kernel/config/boot.conf
4 -> /Volumes/WorkSpace/Code/My_OS/esp/EFI/B00T/boot.conf
5 [+] Building: kernel...
  [?] Executing: /Users/__allenge/.cargo/bin/cargo build --release in /
6 Volumes/WorkSpace/Code/My_OS/pkg/kernel
  [?] Would copy: /Volumes/WorkSpace/Code/My_OS/target/x86_64-unknown-none/
7 release/ysos_kernel -> /Volumes/WorkSpace/Code/My_OS/esp/KERNEL.ELF
  [?] Executing: /opt/homebrew/bin/qemu-system-x86_64 -bios assets/OVMF.fd -
8 net none -nographic -m 96M -drive format=raw,file=fat:esp -snapshot
```

1. line1 表示 Makefile 开始在 pkg/boot 目录下执行 Cargo build
2. line2 命令行中调用了 Cargo build --release
3. line3 表明生成的启动程序会被复制到 ESP 目录
4. line4 表示 Makefile 开始在 pkg/kernel 目录下执行 Cargo build

5. line5 内核构建命令，调用了 Cargo build --release
6. line6 行表示内核 ELF 文件会被复制到 ESP 目录
7. line7 行表示 QEMU 启动命令，加载 OVMF.fd BIOS，关闭网络，使用 96M 内存，加载 ESP 目录作为硬盘，使用快照模式，启动模拟器加载 ESP 中的文件运行系统

### 3.3 Cargo 包管理工具

Rust 第三方库 (crate) 的源代码主要托管在 GitHub、GitLab 或其他开源代码托管平台上，并通过 crates.io 发布。当在 Cargo.toml 中声明依赖后：

#### 1. 下载源代码

Cargo 会自动下载依赖的源代码，并将其存放在本地缓存目录（通常是 `~/.cargo/registry/src`）。

#### 2. 编译

当执行 `cargo build` 时，Cargo 会根据依赖关系将这些库在项目编译过程中一并编译。也就是说，它们会在项目构建时从源代码编译成库文件（通常是 `rlib` 或动态链接库）。

### 3.4 #[entry] 与 main

在无需标准库 (`no_std`) 的环境下，即 UEFI 程序中，我们不能依赖默认的 Rust 运行时来处理入口点。`entry` 宏允许开发者定义一个自定义的入口函数（例如 `efi_main`），以匹配 UEFI 的启动规范，同时避免了使用标准库的运行时初始化工作。这保证了代码能在特定固件环境下正确运行，并与 UEFI 提供的 API 配合。

## 4 附加题

### 4.1 彩色的日志

可以通过第三方库 `colored` 实现彩色的日志输出。`colored` 库利用 ANSI 转义序列来为终端输出添加颜色和样式。原理如下：

#### • 扩展 Trait

通过为 `String` 和 `&str` 实现了扩展 trait `Colorize`，库为文本添加了方法，如 `red()`、`green()`、`bold()` 等，用于设置文本颜色和样式。这些方法返回封装了原始文本和对应 ANSI 控制序列的包装类型。

## • 生成 ANSI 转义码

每种颜色或样式对应一个特定的 ANSI 转义码。例如，当调用 `.red()` 方法时，库会在文本前后添加控制码 `\x1b[31m` 和 `\x1b[0m`，分别表示红色文本和重置样式，以实现红色的字符串输出效果。

## • Display 实现

封装后的类型实现了 `std::fmt::Display` trait，使得可以直接在 `println!` 宏中使用。例如，`println!("{}", "Hello".red())` 会输出红色的“Hello”。

一下是具体的代码（实现效果如图 1.1.3 所示）：

```
1  fn init_logging() {
2      env_logger::Builder::new()
3          .filter_level(log::LevelFilter::Info)
4          .format(|buf, record| {
5              let prefix_msg = record.args().to_string();
6              let formatted = match record.level() {
7                  Level::Info => {
8                      // INFO : 是绿色，后面的内容是白色
9                      format!(
10                         "{} {}",
11                         "[INFO]:".green(),
12                         prefix_msg.white()
13                     )
14                 }
15                 Level::Warn => {
16                     // "WARNING"是黄色加粗并加下划线，后面的内容是黄色加粗
17                     format!(
18                         "{} {}",
19                         "[WARNING]:".yellow().bold().underline(),
20                         prefix_msg.yellow().bold()
21                     )
22                 }
23                 Level::Error => {
24                     // "ERROR"及其后面内容是红色加粗，并居中
25                     let line = format!("[ERROR]: {}",
26                                         prefix_msg).red().bold().to_string();
27                     if let Some((Width(width), _)) = terminal_size() {
28                         let lth = line.len() as u16;
29                         let spaces = (width.saturating_sub(lth)) / 2;
```

Rust

```

29         format!("{}", "{}{}", "=", ".repeat(spaces as
           format!("{}", "{}{}", "=", ".repeat(spaces as
           format!("{}", "{}", line)
30     } else {
31         format!("{}", line)
32     }
33 }
34 _ => prefix_msg,
35 };
36 writeln!(buf, "{}", formatted)
37 })
38 .init();
39 }

```

## 4.2 Rust 实现简单 shell

通过 Rust 实现了一个包含了 `ls`, `cd`, `cat` 命令的简单 shell。实现效果以及代码如下所示。

```

cargo run --release
test.txt          95      file    2025-03-06 12:52:02
.git             288      dir     2025-03-05 23:20:25
src              96      dir     2025-03-05 23:20:25
shell> cat .git
===[ERROR]: cat: Failed to read file .git: Is a directory (os error 21)===
shell> cat test.txt
Never gonna give you up...
Never gonna let you down...
Never gonna run around and desert you...
shell> ls
Listing directory: "/Volumes/WorkSpace/Code/RustLearning/word_hello"
Name      Size   Type    CreatedTime
Cargo.lock 12896  file    2025-03-05 23:21:38
target    192    dir     2025-03-06 12:46:56
Cargo.toml 164    file    2025-03-05 23:20:25
.gitignore 8       file    2025-03-05 23:20:25
test.txt   95     file    2025-03-06 12:52:02
.git       288    dir     2025-03-05 23:20:25
src        96     dir     2025-03-05 23:20:25
shell> cd ..
shell> ls
Listing directory: "/Volumes/WorkSpace/Code/RustLearning"
Name      Size   Type    CreatedTime
word_hello 288    dir     2025-03-05 23:20:25
shell>

```

图 4.2.5: 简单 shell 运行效果

```

1  /// 实现一个简单的 shell, 支持 cd、ls 和 cat 命令
2  fn run_shell() -> Result<(), Box<dyn std::error::Error>> {
3      let mut input = String::new();

```

Rust

```

4      loop {
5          // 提示符
6          print!("shell> ");
7          io::stdout().flush()?;
8          input.clear();
9          io::stdin().read_line(&mut input)?;
10         let input = input.trim();
11         if input.is_empty() {
12             continue;
13         }
14         // 退出命令
15         if input == "exit" || input == "quit" {
16             break;
17         }
18         // 分解命令和参数
19         let mut parts = input.split_whitespace();
20         let cmd = parts.next().unwrap();
21         match cmd {
22             "cd" => {
23                 if let Some(path) = parts.next() {
24                     // 切换当前工作目录（这里不检查目标是否存在）
25                     if let Err(e) = env::set_current_dir(path) {
26                         error!("cd: Failed to change directory to {}:
27                             {}", path, e);
28                     }
29                 } else {
30                     println!("Usage: cd <directory>");
31                 }
32             }
33             "ls" => {
34                 let current_dir = env::current_dir()?;
35                 println!("Listing directory: {:?}", current_dir);
36                 println!("{:<20}\t{}\t{}\t{}\t{}", "Name", "Size", "Type",
37                     "CreatedTime");
38                 for entry in fs::read_dir(current_dir)? {
39                     let entry = entry?;
40                     let path = entry.path();
41                     let metadata = entry.metadata()?;
42                     let size = metadata.len();
43                     let created = match metadata.created() {

```



```

44         };
45         let file_name =
46         path.file_name().unwrap().to_string_lossy();
47         let colored_name = if metadata.is_dir() {
48             file_name.blue().bold()
49         } else if metadata.is_file() {
50             file_name.green()
51         } else {
52             file_name.red()
53         };
54         let file_type = if metadata.is_dir() { "dir" } else
55         { "file" };
56         let created_display = match metadata.created() {
57             Ok(time) => {
58                 let datetime: chrono::DateTime<chrono::Local>
59                 = time.into();
60                 datetime.format("%Y-%m-%d %H:%M:
61                 %S").to_string()
62             }
63             Err(_) => "Unknown".to_string(),
64         };
65         println!(
66             "{:<20}\t{}\t{}\t{}",
67             colored_name,
68             size,
69             file_type,
70             created_display
71         );
72     }
73     "cat" => {
74         if let Some(path) = parts.next() {
75             match fs::read_to_string(path) {
76                 Ok(content) => println!("{}", content),
77                 Err(e) => error!("cat: Failed to read file {}:
78                 {}", path, e),
79             }
80         } else {
81             println!("Usage: cat <filename>");
82         }
83     }
84     _ => {

```



```
81         println!("Unknown command: {}", cmd);
82         println!("Supported commands: cd, ls, cat, exit");
83     }
84 }
85 }
86 Ok(())
87 }
```

### 4.3 线程模型

下面给出一个示例代码，分别用 `static mut`（不安全版本）和 `AtomicU16`（安全版本）来生成 `UniqueId`，并在多线程下测试它们是否能保证每次生成的 ID 都唯一。

```
1  use std::collections::HashSet;
2  use std::sync::atomic::{AtomicU16, Ordering};
3  use std::thread;
4
5  #[derive(Debug, Eq, PartialEq)]
6  struct UniqueIdUnsafe(u16);
7
8  impl UniqueIdUnsafe {
9      // 使用 unsafe 的 static mut 实现，不保证线程安全
10     fn new() -> Self {
11         unsafe {
12             static mut COUNTER: u16 = 0;
13             let id = COUNTER;
14             // 模拟并发时可能产生竞态条件
15             COUNTER += 1;
16             UniqueIdUnsafe(id)
17         }
18     }
19 }
20
21 #[derive(Debug, Eq, PartialEq)]
22 struct UniqueIdSafe(u16);
23
24 impl UniqueIdSafe {
25     // 使用 AtomicU16 保证线程安全
26     fn new() -> Self {
27         static COUNTER: AtomicU16 = AtomicU16::new(0);
28         UniqueIdSafe(COUNTER.fetch_add(1, Ordering::SeqCst))
29     }
}
```

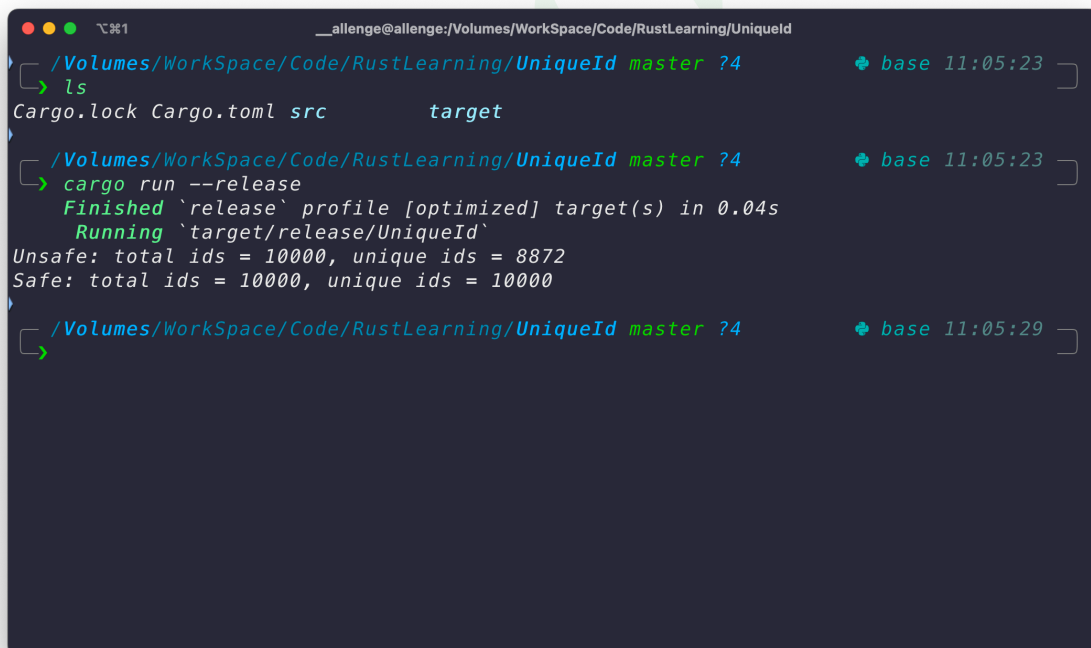
Rust

```
30 }
31
32 fn main() {
33     // 用 static mut 生成 UniqueIdUnsafe, 在多线程下测试是否有重复
34     let mut unsafe_handles = Vec::new();
35     for _ in 0..10 {
36         unsafe_handles.push(thread::spawn(|| {
37             let mut ids = Vec::new();
38             for _ in 0..1000 {
39                 ids.push(UniqueIdUnsafe::new().0);
40             }
41             ids
42         }));
43     }
44     let mut unsafe_ids = Vec::new();
45     for h in unsafe_handles {
46         unsafe_ids.extend(h.join().unwrap());
47     }
48     let unique_unsafe: HashSet<u16> =
49     unsafe_ids.iter().cloned().collect();
50     println!(
51         "Unsafe: total ids = {}, unique ids = {}",
52         unsafe_ids.len(),
53         unique_unsafe.len()
54     );
55     // 用 AtomicU16 生成 UniqueIdSafe, 在多线程下测试是否有重复
56     let mut safe_handles = Vec::new();
57     for _ in 0..10 {
58         safe_handles.push(thread::spawn(|| {
59             let mut ids = Vec::new();
60             for _ in 0..1000 {
61                 ids.push(UniqueIdSafe::new().0);
62             }
63             ids
64         }));
65     }
66     let mut safe_ids = Vec::new();
67     for h in safe_handles {
68         safe_ids.extend(h.join().unwrap());
69     }
70     let unique_safe: HashSet<u16> = safe_ids.iter().cloned().collect();
```

```
71     println!(  
72         "Safe: total ids = {}, unique ids = {}",  
73         safe_ids.len(),  
74         unique_safe.len()  
75     );  
76 }
```

1. 在 `UniqueIdUnsafe::new()` 中，我们使用了 `static mut` 来存储计数器，这是不安全的，因为缺乏同步机制，多个线程可能同时访问和修改计数器，导致竞态条件。
2. 在 `UniqueIdSafe::new()` 中，我们使用了 `AtomicU16` 来存储计数器，这是线程安全的，因为 `AtomicU16` 提供了原子操作，保证了多线程下的正确性。
3. 主函数中，我们分别在多线程下生成了 1000 个 ID，然后统计了生成的 ID 总数和唯一 ID 数量。从输出结果可以看出，使用 `AtomicU16` 生成的 ID 是唯一的，而使用 `static mut` 生成的 ID 中存在重复。

测试结果如下所示。



```
__allenge@allenge:/Volumes/WorkSpace/Code/RustLearning/UniqueId  
/Volumes/WorkSpace/Code/RustLearning/UniqueId master ?4 base 11:05:23  
ls  
Cargo.lock Cargo.toml src target  
/Volumes/WorkSpace/Code/RustLearning/UniqueId master ?4 base 11:05:23  
cargo run --release  
Finished `release` profile [optimized] target(s) in 0.04s  
Running `target/release/UniqueId`  
Unsafe: total ids = 10000, unique ids = 8872  
Safe: total ids = 10000, unique ids = 10000  
/Volumes/WorkSpace/Code/RustLearning/UniqueId master ?4 base 11:05:29
```

图 4.3.6: 线程模型测试结果