## 1. 禁用标准库

### 默认情况下，std会链接到每一个Rust的封装对象，它依赖于操作系统 通过#![no\_std]将其禁用，此时可以利用核心库，他有较少的依赖

## 2.错误项

### `#[panic\_handler]` function required, but not found

### 原因：这个的意思是需要一个panic\_handler函数，在程序发生panic的时候进行调用

### 解决方法：定义一个panic\_handler函数

### language item required, but not found:`eh\_personality`

### 原因：这是一个错误相关语义项，它是一个标记某函数用来实现堆栈展开处理功能的语义项。这个语义项也与 panic 有关

### 解决方法：修改Cargo.toml 设置为不堆栈展开

[profile.dev]

panic = "abort"

[profile.release]

panic = "abort"

### profile里是自定义rustc编译配置

### requires `start` lang\_item

### 原因：一个典型的链接了标准库的 Rust 程序会首先跳转到 C 语言运行时环境中的 crt0，进入 C 语言运行时环境 设置 C 程序运行所需要的环境 然后 C 语言运行时环境会跳转到 Rust 运行时环境的入口点 进入 Rust 运行时入口函数继续设置 Rust 运行环境，而这个 Rust 的运行时入口点就是被 start 语义项标记的。Rust 运行时环境的入口点结束之后才会调用 main 函数进入主程序。

### 解决方法：重写整个入口点 覆盖start语义项不能解决问题，覆盖 crt0 中的 \_start 函数

pub extern "C" fn \_start()

### 表示这个函数是一个C函数

### 链接错误

### 原因：链接器的默认配置假定程序依赖于 C 语言的运行时环境，但我们的程序并不依赖于它

### 解决方法：提供特定的链接器参数（Linker Argument），也可以选择编译为裸机目标

rustup target add riscv64imac-unknown-none-elf

### 此时可设置 ./.cargo/config

### 

## 3.调整内存

### 入口地址不匹配

### 文档上写的是11000但是我生成的入口地址是11120 而不是11000 反汇编得到的结果也不一样 这是为什么?

### 反汇编的结果：

### 11120: 09 a0 j 2

### 11122: 01 a0 j 0

### 比文档中少了四行

### 11000: 41 11 addi sp, sp, -16 //sp=sp-16

### 11002: 06 e4 sd ra, 8(sp) //

### 11004: 22 e0 sd s0, 0(sp) //

### 11006: 00 08 addi s0, sp, 16

这个问题保留 等实验做完再探究

## 4. 接口封装和代码整理

### Sbi文档中:a7 (or t0 on RV32E-based systems) encodes the SBI extension ID

### 其中 a7对应x17(在riscv-spec)

### 通过指定a7的值来对应函数

### SBI functions must return a pair of values in a0 and a1, with a0 returning an error code. This is analogous to returning the C structure

### a0是x10 从而内联汇编中

llvm\_asm!("ecall"

: "={x10}" (ret)

: "{x10}" (arg0), "{x11}" (arg1), "{x12}" (arg2), "{x17}" (which)

: "memory" // 如果汇编可能改变内存，则需要加入 memory 选项

: "volatile"); // 防止编译器做激进的优化（如调换指令顺序等破坏 SBI 调用行为的优化）

### 这里使用a0 a1 a2应该是只能用到这三个参数

### 返回a0是指错误信息

### 后面调用函数就不用解释了

## 5. 实现格式化输出

### 实现

fn write\_str(&mut self, s: &str) -> Result

fn write\_fmt(mut self: &mut Self, args: Arguments<'\_>) -> Result

console\_putchar=>write\_str=>write\_fmt

//调用关系 其中console\_putchar已经实现了

fn write\_str(&mut self, s: &str) -> fmt::Result {

let mut buffer = [0u8; 4];

for c in s.chars() {

for code\_point in c.encode\_utf8(&mut buffer).as\_bytes().iter() {

console\_putchar(\*code\_point as usize);

}

}

Ok(())

}

### 假设 s=”abc”

### 则c=‘a’ ‘b’ ‘c’

### c.encode\_utf8(&mut buffer) 将c转换为了str 且将对应的ascii码存入buffer

### 

### 最后得到的code\_point是对应字符的ascii码

### code\_point是&u8类型 需要解引用

pub fn print(args: fmt::Arguments) {

Stdout.write\_fmt(args).unwrap();

}

### 这个会调用我们已经写好的write\_str

#[macro\_export]

macro\_rules! print {

($fmt: literal $(, $($arg: tt)+)?) => {

$crate::console::print(format\_args!($fmt $(, $($arg)+)?));

}

}

/// 实现类似于标准库中的 `println!` 宏

///

/// 使用实现了 [`core::fmt::Write`] trait 的 [`console::Stdout`]

#[macro\_export]

macro\_rules! println {

($fmt: literal $(, $($arg: tt)+)?) => {

$crate::console::print(format\_args!(concat!($fmt, "\n") $(, $($arg)+)?));

}

}

### 定义print宏 这里应该是固定格式

### 将得到的参数原封不动的传回了

### 

### $(...)+ 表示一次或者多次匹配 tt是词法树 ,是指前面有没有带, literal是迭代

### 指多个变量 目前猜测是这样