

2024.03.24-2024.03.31-work-log

工作进展

本阶段主要完成的任务有：研究阅读Rust编译器的标准库的实现，尝试基于 `libc` 编写适配 `aarch64-unknown-rtsmart` 平台的标准库

本阶段创建了一个新的项目 `rtsmart-std`，提供给新的编译平台类似于Rust标准库的功能。

原先计划是在Rust原有标准库上进行修改，使之能够适配 `aarch64-unknown-rtsmart` 平台。但是后来发现需要修改的内容过多，工作量较大，因此我们转变思路，为新平台重新编写一个适用的标准库。

资料收集

Rust标准库源代码：<https://github.com/rust-lang/rust/tree/master/library/std>

Rust标准库源代码解析：<https://github.com/Warrenren/inside-rust-std-library>

rtsmart-std代码思路

首先这是一个Rust的lib项目，在lib.rs中定义如下特性：

```
#![no_std]
#![feature(alloc_error_handler)]
#![feature(allow_internal_unstable)]
#![feature(linkage)]
#![feature(core_intrinsics)]
#![allow(dead_code)]
```

由于是标准库，因此是在no_std的情况下进行编译的

且需要自定义panic_handler，因此使用 `#![feature(alloc_error_handler)]`

`#![feature(allow_internal_unstable)]` 用于允许使用一些不稳定的功能例如 `core::intrinsics::unlikely`

`#![feature(linkage)]` 用于链接一些库函数

`#![feature(core_intrinsics)]` 用于启用core库中的一些函数

`#![allow(dead_code)]` 用于允许存在未使用的代码而不产生警告

global_allocator

需要定义一个全局内存分配器，创建 `malloc` 模块，定义一个内存分配器，为它实现 `GlobalAlloc` 这个 trait

通过阅读 `rtthread` 头文件可以观察到 `rt_malloc` 和 `rt_free` 两个控制内存分配与释放的函数，因此在 `libc` 中引入这两个函数

```
// rtthread.h
void *rt_malloc(rt_size_t size);
void rt_free(void *ptr);
```

```
// rtsmart/mod.rs
pub fn rt_malloc(size: rt_size_t) -> *mut ::c_void;
pub fn rt_free(ptr: *mut ::c_void);
```

然后在api模块中创建一个用于内存分配与释放的接口

```
//! For malloc.rs
use libc::c_void;
use libc::rt_free;
use libc::rt_malloc;

// Alloc memory
pub fn mem_alloc(bytes: usize) -> *mut c_void {
    unsafe { rt_malloc(bytes as _) }
}

// Free memory
pub fn mem_free(ptr: *mut c_void) {
    unsafe {
        rt_free(ptr);
    }
}
```

然后在malloc模块中调用这个接口实现全局内存分配器

```
use crate::api::*;
use crate::panic_on_atomic_context;
use core::alloc::{GlobalAlloc, Layout};
use libc::c_void;
use crate::api::mem::{mem_alloc, mem_free};

#[alloc_error_handler]
fn foo(_: core::alloc::Layout) -> ! {
    panic!("OOM!");
}

pub struct RttAlloc;

unsafe impl GlobalAlloc for RttAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        panic_on_atomic_context("malloc");
        mem_alloc(layout.size() as usize) as *mut u8
    }

    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        panic_on_atomic_context("dealloc");
        mem_free(ptr as *mut c_void)
    }
}
```

最后在lib.rs中声明这一全局内存分配器即可

```
#[global_allocator]
static GLOBAL: malloc::RttAlloc = malloc::RttAlloc;
```

panic_handler

`panic_handler` 属性定义的函数在发生panic时会被编译器调用。标准库提供了自己的panic处理函数，但那是在 `no_std` 环境中，我们需要自己定义它，引用了当前的库后也就拥有了panic_handler。因此在标准库中提供panic_handler是很必要的行为。未来还可以考虑专门写一个模块来处理panic

```
#[panic_handler]
#[inline(never)]
fn panic(info: &core::panic::PanicInfo) -> ! {
    print!("{:?}", info);
    __rust_panic()
}

#[linkage = "weak"]
#[no_mangle]
fn __rust_panic() -> ! {
    loop {}
}
```

panic函数和__rust_panic函数都属于发散函数，其中panic函数用于输出具体的错误信息，然后调用__rust_panic函数，__rust_panic函数永远不会将控制内容返回给调用者，从而使得线程崩溃，终止当前线程，达到类似于报错的效果。

prelude

用于引入一些在no_std模式下需要使用到的库，如 `core`、`alloc` 等

```
pub use core::cell::RefCell;
pub use core::cell::UnsafeCell;
pub use core::cmp::*;
pub use core::fmt;
pub use core::intrinsics::write_bytes;
pub use core::iter::Iterator;
pub use core::marker::PhantomData;
pub use core::mem;
pub use core::num::Wrapping;
pub use core::ops::Range;
pub use core::ops::{Deref, DerefMut};

pub use alloc::boxed::Box;
pub use alloc::rc::Rc;
pub use alloc::sync::{Arc, Weak};

pub use alloc::collections;
pub use alloc::string;
pub use alloc::vec;
```

总结

本周主要是决定了新建一个专门用于支持 `aarch64-unknown-rtsmart` 平台的标准库，并且成功创建了项目，搭建了代码框架，未来将进一步完善标准库的内容