

# Rust改写Harmony LiteOS

RushToLight小组成员：舒佳豪 区家彬 吴宇翀 姬子琢



汇报人：区家彬



时间：2024. 04. 22

# 目录

C O N T E N T S

01 liteOS简介

02 改写模块选择

03 为什么选择Rust?

04 怎么使用Rust改写C

05 Rust、C相互调用

06 Linux下LiteOS的编译

07 未来计划



01

liteOS简介

*OpenHarmony LiteOS-M* 属于HarmonyOS开源项目的一个嵌入式系统。

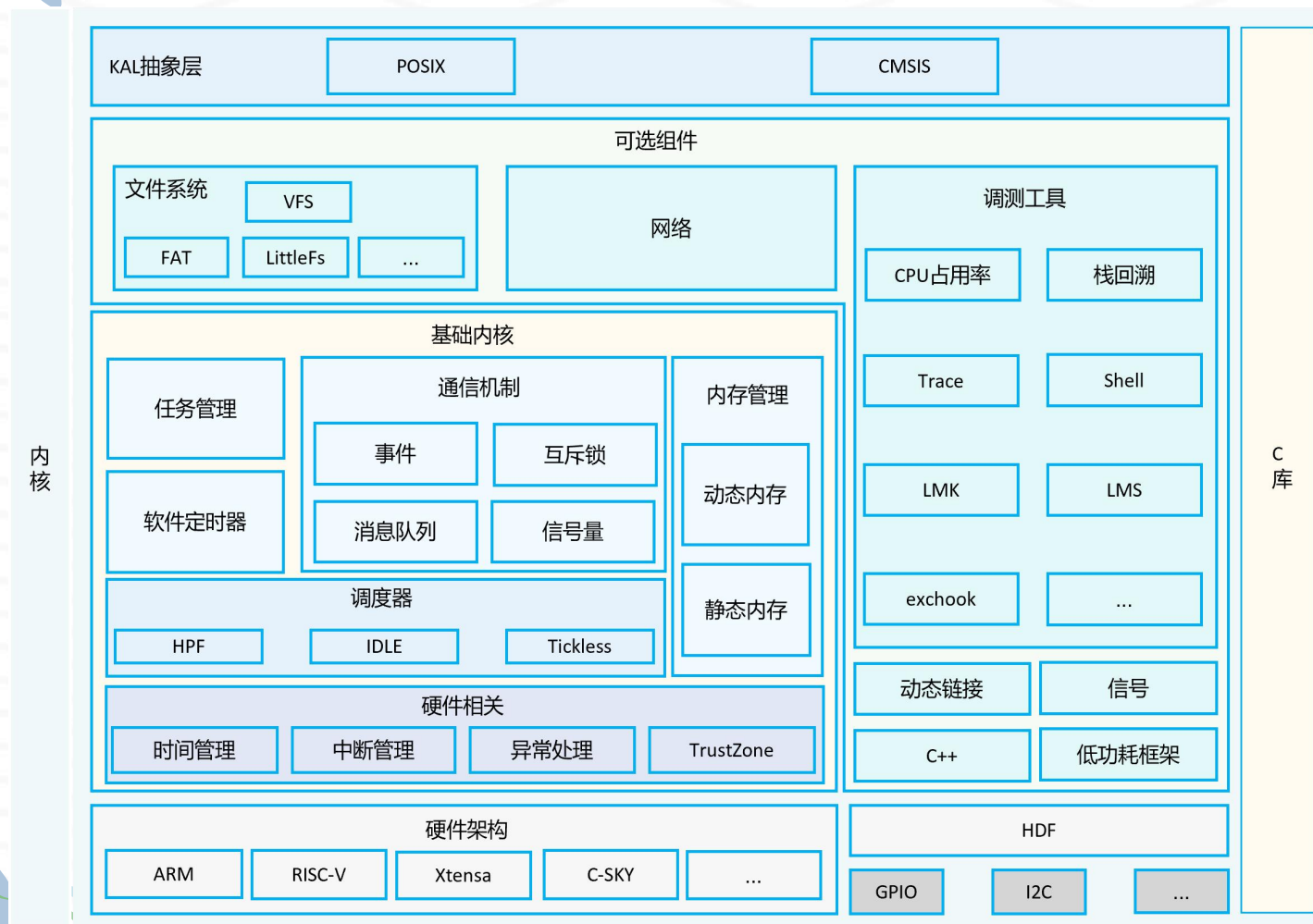
主要面向IoT领域构建的物联网操作系统

LiteOS-M适用于轻量级的芯片架构，面向的MCU一般是百K级内存，例如cortex-m、riscv32。

有趣的是，虽然我们和Rage\_of\_dUST小组都选择了lite-os，但是他们选择的是Huawei LiteOS，这是两个不同的开源项目，代码差异较大。







鸿蒙 liteOS 的内核框架如下，其中的基础内核文件树如下：

```
kernel
├── include
│   ├── los_config.h
│   ├── los_event.h
│   ├── los_membox.h
│   ├── los_memory.h
│   ├── los_mux.h
│   ├── los_queue.h
│   ├── los_sched.h
│   ├── los_sem.h
│   ├── los_sortlink.h
│   ├── los_swtmr.h
│   ├── los_task.h
│   └── los_tick.h
└── src
```



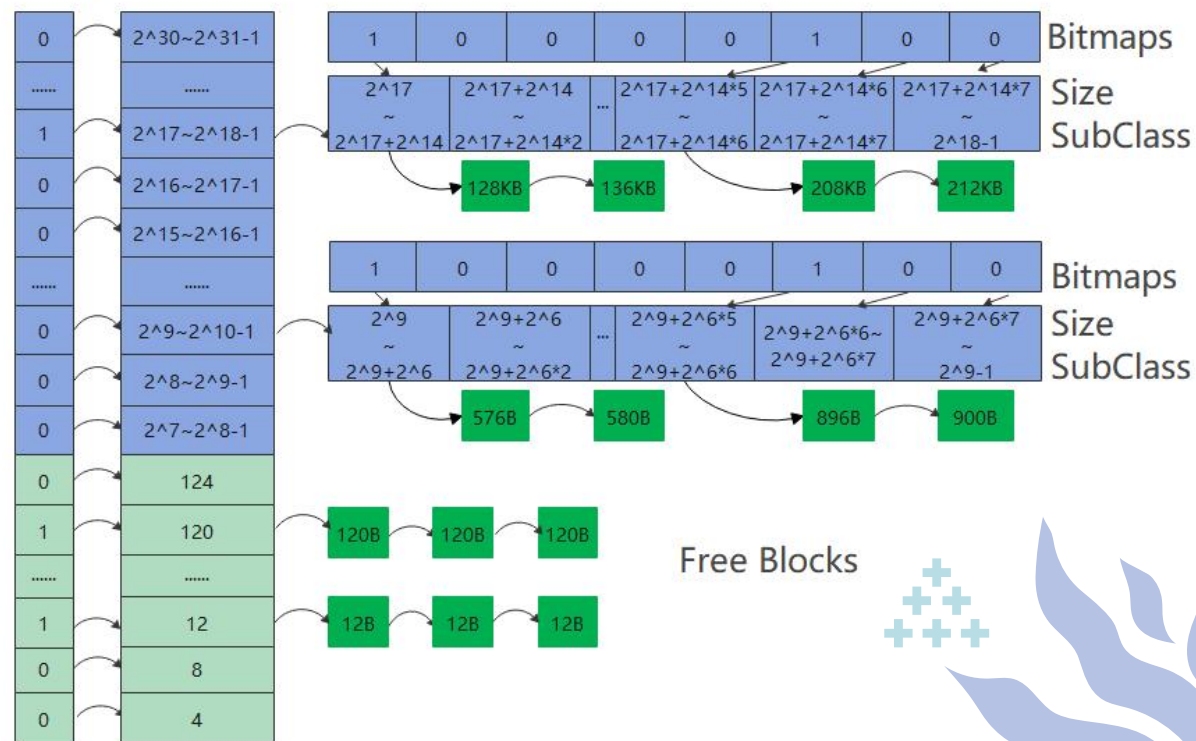
02

改写模块选择



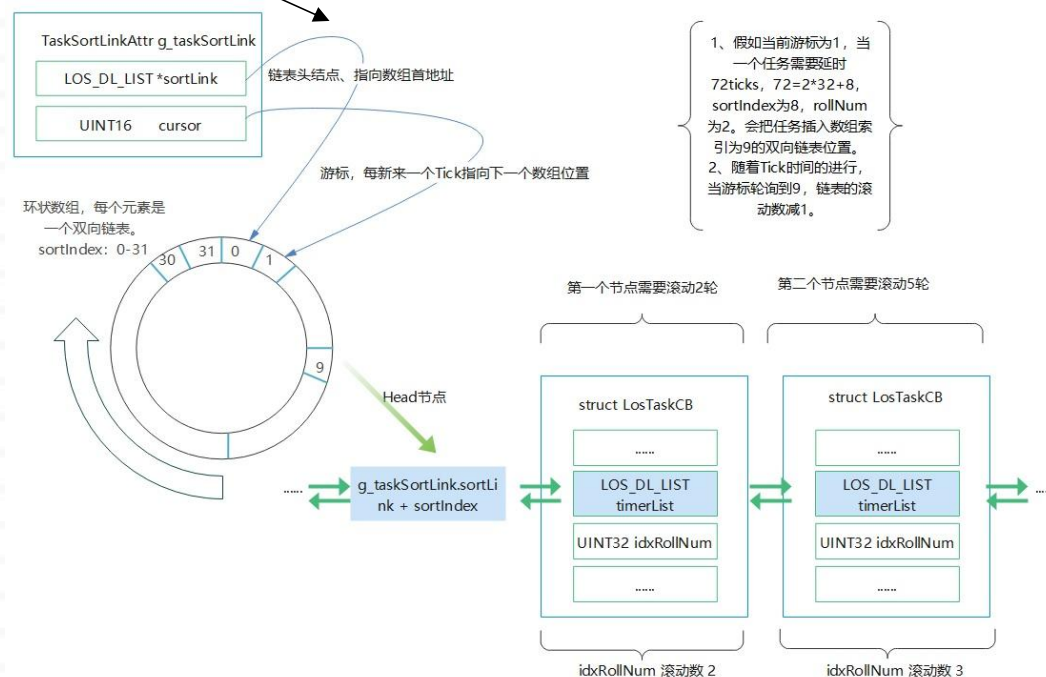
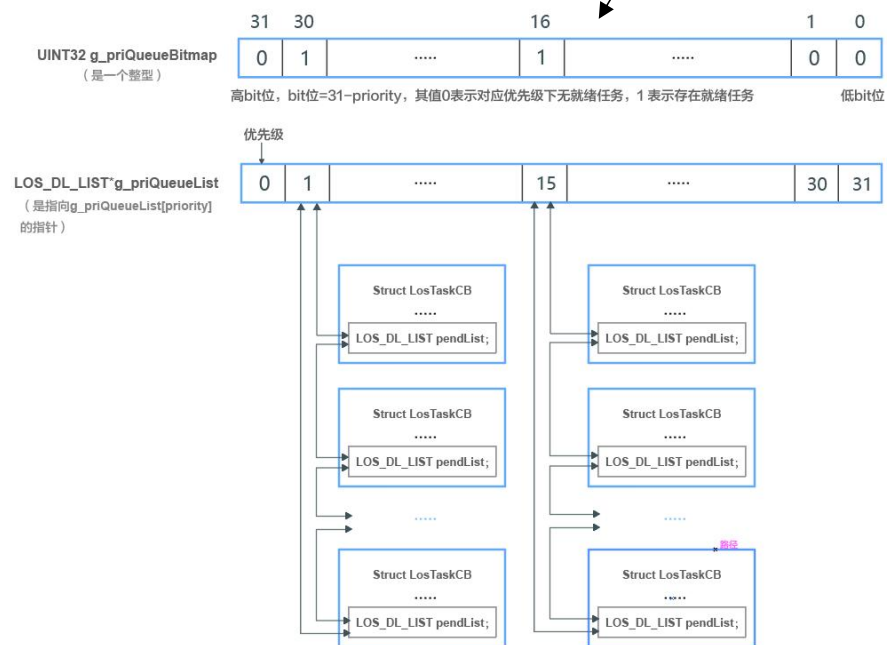
- ◆ memory是LiteOS-m负责提供动态内存资源分配、回收的相关的函数和宏的模块
- ◆ memory模块主要通过Best fit(最佳分配)算法对动态内存块进行分配
- ◆ 但与传统的遍历寻找方式不同的是：LiteOS-m维护了一个空闲内存块链表来链接起特定大小的内存块，以达到 $O(1)$ 的常数级分配速度

Bitmaps Size Class



- Task是LiteOS-m负责创建任务、调度、管理任务、回收任务资源的模块
- 核心数据结构：任务优先级队列 + 延迟等待队列


LiteOS-m依靠**优先级队列**来调度任务，并且通过时间片的方法提高并发性  
对于未运行完的任务，会进入到**延迟等待队列**，延迟给定时间





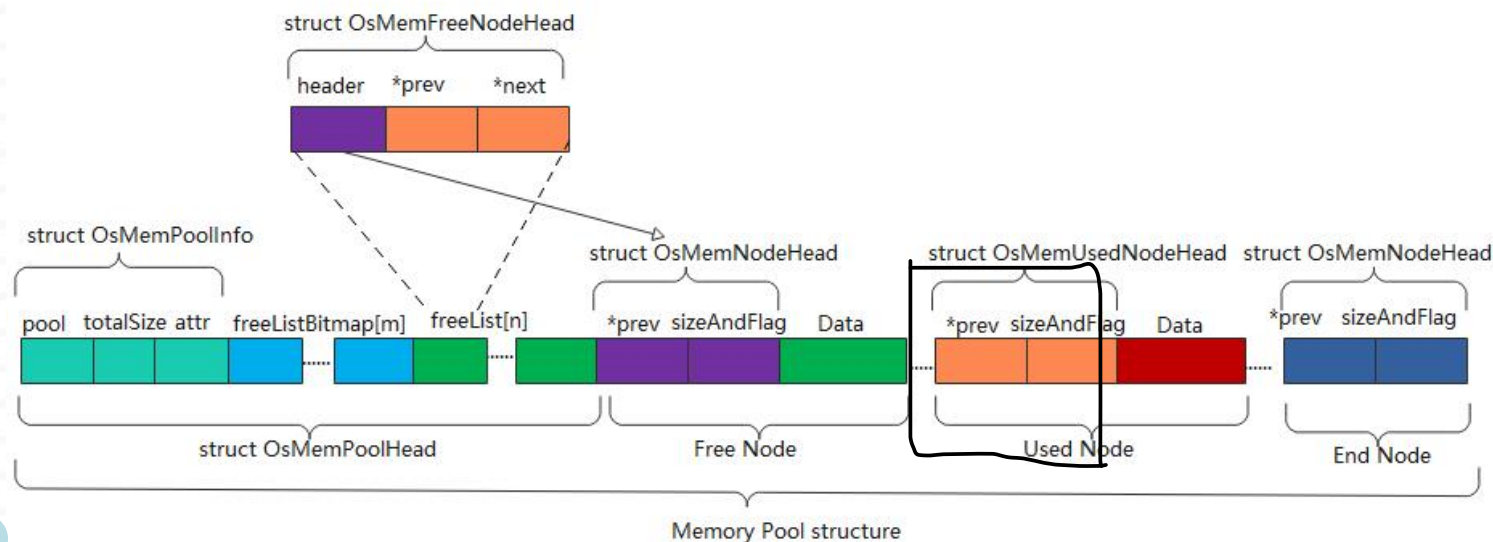
- **memory**模块的函数频繁的对指针进行传递操作,若使用不当可能会造成空指针解引用

- 右图是LiteOS-m对用户提供的内存分配API, 在内存池不够时会未加提示的返回NULL空指针, 编程人员如果未加检测, 可能会造成空指针解引用



```
VOID *LOS_MemAlloc(VOID *pool, UINT32 size)
{
    if ((pool == NULL) || (size == 0)) {
        return NULL;
    }
}
```

- memory模块存在较严重的缓冲区溢出问题
- 作为嵌入式操作系统，LiteOS-m的内存空间是连续分配的，且缺乏分页保护机制，这可能导致越界访问到相邻内存块的头节点，将其中关于内存块的前后节点、内存块长度等信息覆盖掉，引发严重危机



- Task中大量涉及任务间的切换、资源的竞争，以及对于内存块资源的分配使用、大量链表处理的操作
- 由于嵌入式系统对实时性和并发性要求较高，我们认为使用并发性较好的语言对Task的改写可以进一步提高它的运行效率，减少不必要的等待时间
- 任务对内存块的使用、链表的指针传递也可能存在内存安全，比如栈溢出

- 对于mutex.c模块，我们在初步阅读代码后认为改写难度过大(其中涉及较多的互斥，逻辑比较复杂)
- 而其他的例如init.c等模块,我们组认为可改进的地方较小，且考虑到任务量过大，故作罢







03

为什么选择Rust?

### 03 为什么选择Rust?

#### Rust特性



*An Rust Adage :fast, reliable, productive:  
pick three.*

fast:

- 兼具高性能和效率
- 对系统资源有底层的控制权

reliable:

- 具有强静态类型和所有权系统
- 防止了空指针或悬空指针解引用、缓冲区溢出和数据竞争

productive:

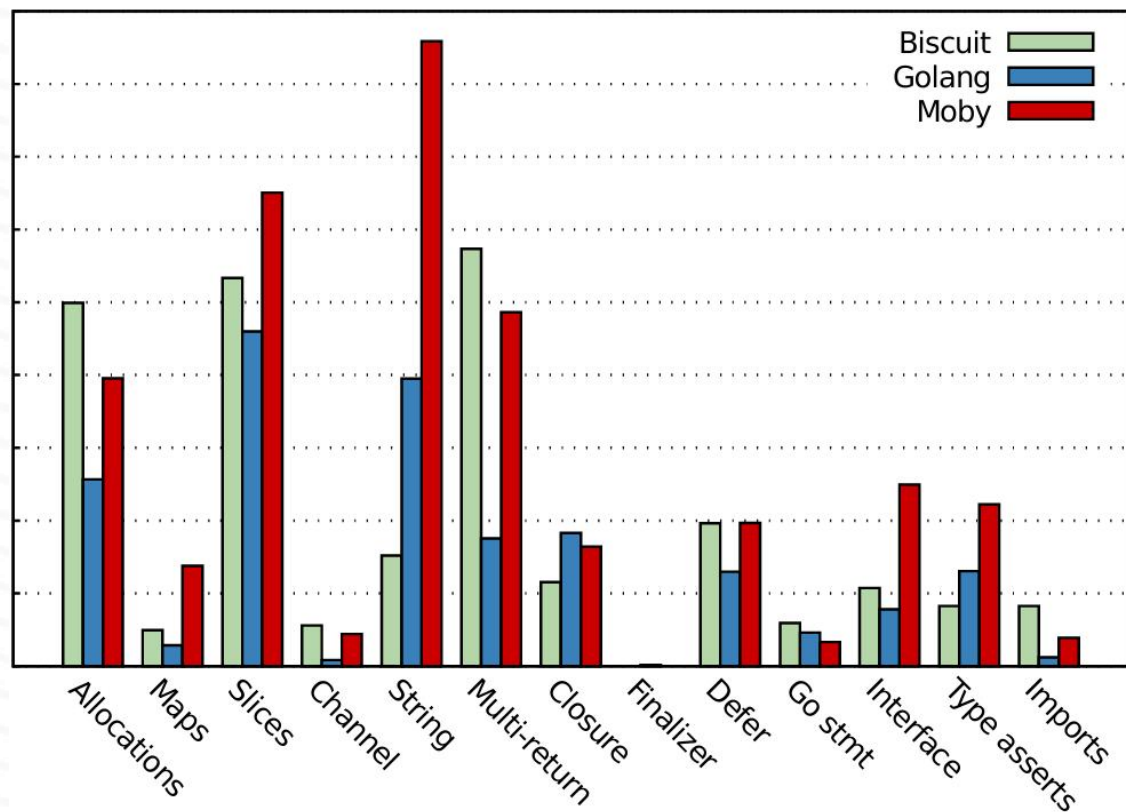
- 具有丰富的标准库
- 相关社区和生态正在完善中

但为什么用Rust而非其他语言改写操作系统，例如GO?

### 03 为什么选择Rust?

## GO VS C 性能比较

**实验：**使用Go编写Biscuit（POSIX内核），并与C编写的Biscuit进行性能比较。  
(OSDI18:The benefits and costs of writing a POSIX kernel in a high-level language)



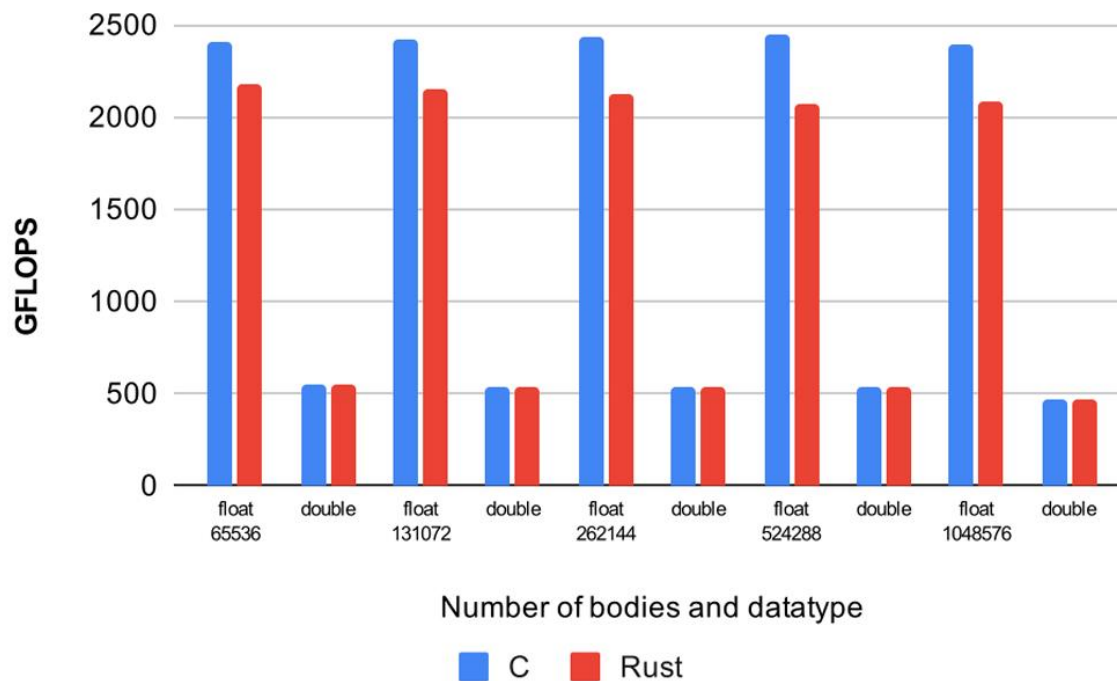
**结论：**client和server采用ping-pong测试模式时，C语言比Go性能高15%，专门针对Page fault做性能测试时，发现C比Go性能高5%。对Go语言来说，由于垃圾回收的存在，对性能影响也不可忽略，垃圾回收占CPU开销的1%~3%，致命的是它影响业务时延，造成业务单次请求的最大时延在574ms

### 03 为什么选择Rust?

#### RUST VS C 性能比较

N 体问题 (N-Body)，是由二十世纪数学家希尔伯特提出的数学难题之一，有很多近似算法。2021年某篇论文使用c和rust解决N体问题，并进行性能比较。

*(Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body)*



结论：在单精度方面，C语言版本优于Rust，实现了1.18倍的改进，而在双精度方面，两种实现的性能几乎相同。





04

怎么使用Rust改写C

## 04 怎么使用Rust改写LiteOS



C中许多类型与Rust相对应，在liteOS中相应的数据类型可以直接转换。

C 类型	Rust 类型
long	isize
unsigned long	usize
__sz	usize
__nsec	core::time::Duration
代表错误信息的 int	Result<(),i32>
取 1 或 0 的 int	bool
T[N]	[T;N]
void __noreturn	!
void*	(泛型函数) *mut T
void*	(泛型函数) *const T
void*	*const u8
void*	*mut u8
const char*	&str
传递结果的 T*, 返回 int	Result<T,int> 或 Optional<T>
传递结果的 T*, 返回 void	T
T*+长度	&mut [T]
const T*+长度	&[T]

### 条件宏编译

Rust通过cfg属性，对条件编译提供了支持。cfg宏存在两种使用方式：

#### 1. 在代码块外使用

可以在整个模块或crate级别上使用#[cfg(...)]注解来决定整个模块或crate是否包含在编译中；



#### 2. 在代码块内使用

通过使用cfg!宏来根据条件选择性的编译代码；cfg!会返回一个布尔值；





05

Rust、C交叉编译



1

用命令 `cargo init -lib Linklist` 创建名为 Linklist 的 Rust 库，lib.rs 内容为（这个例子是双向链表的改写）：

```
1 #![crate_type = "staticlib"] //表示这是一个静态库 (static library) ，而非可  
   执行文件  
2 extern crate libc; //引入c标准库，用于与c语言接口交互  
3 use core::ffi::c_int; //使用core模块的ffi模块，其中的c_int类型代表c语  
   言中的整型  
4 #[repr(C)] // 使用c语言的内存布局  
5 pub struct Node{ //定义一个结构体Node，用于表示链表中的节点  
6     value: c_int,  
7     next: *mut Node,  
8     pre: *mut Node,  
9 }  
10 #[inline] //使用inline属性确保这个函数在编译时被内联  
11 #[no_mangle] //告诉编译器不要为函数生成特定的名字，以便于与c  
   语言接口兼容  
12 pub extern "C" fn LOS_ListInit(list: &mut Node, node_value: c_int)  
13 {  
14     list.next = list as *mut Node;  
15     list.pre = list as *mut Node;  
16     list.value = node_value;  
17     println!("List initialized. Head value: {:?}", list.value);  
18 }
```

Cargo.toml为:

```
1 [package]
2 name = "Linklist"           # 名称, 用于识别项目
3 version = "0.1.0"          # 版本号, 遵循 SemVer 标准
4 edition = "2021"           # Rust 语言的版本
5
6 [lib]
7 name = "rust_linklist"      # 库的名称, 与Cargo.toml文件中的其他模块区分
8 crate-type = ["staticlib"] # 指定生成的库类型, 这里是静态库 (staticlib)
9 path = "src/lib.rs"         # 指定源代码文件的位置, 这里是 src 目录下的 lib.rs
                                文件
0
1 [dependencies]
2 libc = "0.2"                # "libc" 是一个依赖项, 它是一个标准C库的Rust绑定, 版本号
                                本号为"0.2"
```

3

用编译命令 `cargo build --release` 编译静态库

4

在Linklist根目录下新建 `cbindgen.toml` 文件，内容为 `language= "C"`

5

执行命令 `cbindgen --config cbindgen.toml - crate Linklist --output rust_linklist.h`，生成头文件 `rust_list.h` 如右图（`crate` 后为库文件名称，`output` 后为目标头文件名称）：

```
1 #include <stdarg.h>
2 #include <stdbool.h>
3 #include <stdint.h>
4 #include <stdlib.h>
5
6 typedef struct Node {
7     int value;
8     struct Node *next;
9     struct Node *pre;
10 } Node;
11
12 void LOS_ListInit(struct Node *list, int node_value);
```

6

在Linklist目录下新建一个main.c文件用于测试rust\_linklist.h头文件。Main.c如右图：

```
1 #include <stdio.h>
2 #include <malloc.h>
3 #include "rust_linklist.h"
4 int main()
5 {
6     Node* p;
7     p = (Node *)malloc(sizeof(Node));
8     LOS_ListInit(p, 5);
9     printf("%d", (*p).value);
10    free(p);
11    return 0;
12 }
```

7

输入命令`gcc -o main main.c -l. -Ltarget/release -lrust_linklist -ldl -lpthread -Wl,-gc-section`对文件进行编译（其中-o后面依次为目标可执行文件名称，测试文件名称，-Ltarget/release -l后面为库名称），生成main文件。

8

.main执行，得结果：

```
List initialized. Head value: 5
```



## 05 Rust调用C

1

以斐波那契为例，用`cargo new c_to_rust`创建一个名为`c_to_rust`的rust工程，在`src`下创建`fibonacci.h`：

```
1 #ifndef FIBONACCI_H
2 #define FIBONACCI_H
3
4 #ifdef __cplusplus
5 extern "C" {
6 #endif
7
8 int fibonacci(int n); // 函数声明
9
10 #ifdef __cplusplus
11 }
12 #endif
13
14 #endif // FIBONACCI_H
```

## 05 Rust调用C

2

在src下创建fibonacci.c:

```
1 #include "fibonacci.h"
2
3 int fibonacci(int n) {
4     if (n <= 1) {
5         return n;
6     }
7     else {
8         return fibonacci(n - 1) + fibonacci(n - 2);
9     }
10 }
```

3

在c\_to\_rust目录下创建build.rs文件:

```
1 extern crate cc;    //Rust编译器插件(Cargo crate),允许从Rust项目中编译C和C++代  
   码  
2  
3 fn main() {  
4     cc::Build::new().file("src/fibonacci.c").compile("libfibonacci.a");  
5     //编译 src/fibonacci.c 文件, 并生成一个名为 libfibonacci.a 的静态库。  
6 }
```

4

Cargo.toml修改为:

```
1 [package]
2 name = "Linklist"           # 名称, 用于识别项目
3 version = "0.1.0"          # 版本号, 遵循 SemVer 标准
4 edition = "2021"           # Rust 语言的版本
5
6 [lib]
7 name = "rust_linklist"      # 库的名称, 与Cargo.toml文件中的其他模块区分
8 crate-type = ["staticlib"] # 指定生成的库类型, 这里是静态库 (staticlib)
9 path = "src/lib.rs"         # 指定源代码文件的位置, 这里是 src 目录下的 lib.rs
                                文件
0
1 [dependencies]
2 libc = "0.2"                # "libc" 是一个依赖项, 它是一个标准C库的Rust绑定, 版本号
                                本号为"0.2"
```



Main.rs内容为:

```
1 extern crate libc;
2 //引入libc库, 包含标准库函数与类型定义
3
4 extern {
5     fn fibonacci(n: libc::c_int) -> libc::c_int;
6 }
7 //外部声明: 引入名为fibonacci的c函数, 它接受一个c_int类型的参数并返回c_int类型
8
9 fn main()
10 {
11     for i in 1..10{
12         println!("fibonacci({:?}) = {:?}", i, unsafe{fibonacci(i)});
13     }
14     //由于c函数的类型和安全性以及Rust默认不支持直接调用c函数, 需要unsafe块来调用
15 }
```

Cargo run命令得执行结果：（或者cargo build再加./target/debug/c\_to\_rust\_test）

```
Finished dev [unoptimized + debuginfo] target(s) in 0.02s
Running `target/debug/c_to_rust_test`
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
```



06

Linux下LiteOS的编译

## 06 Linux下LiteOS的编译

1

准备工作：安装GNU Arm Embedded Toolchain编译器：

在ARM官方下载gcc-arm-none-eabi-10-2020-q4-major-x86\_64-linux.tar.bz2(建议这个版本)。

用tar -xvf gcc-arm-none-eabi-10-2020-q4-major-x86\_64-linux.tar.bz2解压。

！！添加编译器的执行路径到环境变量。

用sudo vim /etc/profile打开/etc/profile文件，在最后一行加上(vim操作)

Export PATH=\$PATH:··/gcc-arm-none-eabi-10-2020-q4-major/bin(··为你自己这个文件夹的路径)

保存退出 (: wq!)

Source /etc/profile使新设置的环境变量生效，然后便会出现shell，在其中输入Arm-none-eabi-gcc --version验证是否成功。

若成功，会出现：

```
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.2.1 20201103 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```



## 06 Linux下LiteOS的编译

准备工作：make构建器（已有可以跳过）。

在make官网下载make-4.3-tar.gz（不是这个版本也可以）

用tar -xf make-4.3-tar.gz解压为make-4.3,

进入make-4.3文件夹，./configure检查编译与安装make所需的依赖。

用sh build.sh

sudo make

sudo make

2

准备工作：安装图形化配置解析工具。

Python3.8:sudo apt-get install python3.8

3

对应的pip:sudo apt-get install python3-setuptools python3-pip -y与

sudo pip3 install -upgrade pip

Kconfiglib库: sudo pip install Kconfiglib

准备工作：从gitee下载完整的liteOS代码

4

## 06 Linux下LiteOS的编译

拷贝模拟器工程配置文件为根目录.config文件。（以realview-pbx-a9模拟器为例）

在LiteOS的根目录下用`cp tools/build/config/realview-pbx-a9.config .config`命令把QEMU的realview-pbx-a9模拟器工程的config文件拷贝到LiteOS根目录的.config（隐藏文件，`ctrl+h`可见）

5

配置系统。（可选，不做就是系统的默认配置）

配置想要执行的Demo。（可选，只是一个示例）（不做的话默认InspectEntry表示执行所有内核Demo）

6

在LiteOS目录下执行命令：`make menuconfig`

！如果出现报错：`***compiler arm-none-eabi is not in the ENV.Stop.`

只要再执行一次`source /etc/profile`再次进入shell执行命令即可。

在配置框中进入Demos->Kernel Demo, 依次使能Enable Kernel Demo, Kernel Demo Entry, Run Kernel Task Demo。

S保存，q退出。，如下页图。

## 06 Linux下LiteOS的编译

```
(Top) → Demos → Kernel Demo → Enable Kernel Demo → Kernel Demo Entry
( ) InspectEntry
(X) DemoEntry
[*] Run Kernel Task Demo
[ ] Run Kernel Dynamic Mem Demo
[ ] Run Kernel Static Mem Demo
[ ] Run Kernel Interrupt Demo
[ ] Run Kernel Queue Demo
[ ] Run Kernel Event Demo
[ ] Run Kernel Mutex Demo
[ ] Run Kernel Semaphore Demo
[ ] Run Kernel SysTick Demo
[ ] Run Kernel Software Timer Demo
[ ] Run Kernel List Demo
[ ] Run Kernel Test Task Demo
[ ] Run Kernel Test Queue Demo
[ ] Run Kernel Test Event Demo
[ ] Run Kernel Test Mutex Demo
[ ] Run Kernel Test Semaphore Demo
```

## 06 Linux下LiteOS的编译

7

清理工程：清理以前编译出的二进制文件。

在LiteOS根目录下make clean

！！如果出现permission deny的问题，用su root并输入密码升高权限，之后可能又会出现\*\*\*compiler arm-none-eabi is not in the ENV.Stop的问题，仍执行一次source /etc/profile即可。

8

编译工程：在LiteOS根目录下make，成功后如下：

```
alview-pbx-a9/Huawei_LiteOS.elf
#####
#####
##### LiteOS build successfully!
#####
#####
#####
```



## 06 LiteOS的运行

1

准备工作：安装QEMU模拟器。

```
Sudo apt-get install qemu
```

```
Sudo apt-get install qemu-system
```

2

准备工作：完成编译。

3

运行：

```
Qemu-system-arm -machine realview-pbx-a9 -smp 4 -m 512M -kernel  
out/realveiw-pbx-a9/Huawei_LiteOS.bin -nographic（如果结果不对把Huawei_LiteOS  
改为你的文件夹名称）
```

上述命令各参数含义如下，更多信息可以通过执行qemu-system-arm --help命令查看：

- -machine：设置QEMU要仿真的虚拟机类型。
- -smp：设置guest虚拟机的CPU的个数。
- -m：为此guest虚拟机预留的内存大小，如果不指定，默认为128M。
- -kernel：设置要运行的镜像文件（包含文件路径）。
- -nographic：以非图形界面启动虚拟机。

## 06 LiteOS的运行

运行结果:

```
*****Hello Huawei LiteOS*****

LiteOS Kernel Version : 5.1.0
Processor   : Cortex-A9 * 4
Run Mode    : SMP
GIC Rev     : GICv1
build time  : Apr 21 2024 00:33:11

*****

main core booting up...
OsAppInit
releasing 3 secondary cores
cpu 1 entering scheduler
cpu 0 entering scheduler
app init!
cpu 3 entering scheduler
cpu 2 entering scheduler

Huawei LiteOS #
Kernel task demo start to run.
Create high priority task successfully.
Create low priority task successfully.
Enter high priority task handler.
Enter low priority task handler.
High priority task LOS_TaskDelay successfully.
High priority task LOS_TaskSuspend successfully.
High priority task LOS_TaskResume successfully.
Kernel task demo finished.
```

07

未来计划

未来  
计划

1. 使用Rust改写liteOS中的一个独立模块，验证编译、改写可行性
2. Rust改写mem.c, task.c以及相关函数（如果有条件，继续改写mux.c等模块）
3. 验证：将Rust改写好的liteOS用qemu编译运行



可能的挑战

1. Rust对C的条件宏的引用

2. 改写后内核的编译

.....



# Question & Answer