



模块化内核进展报告

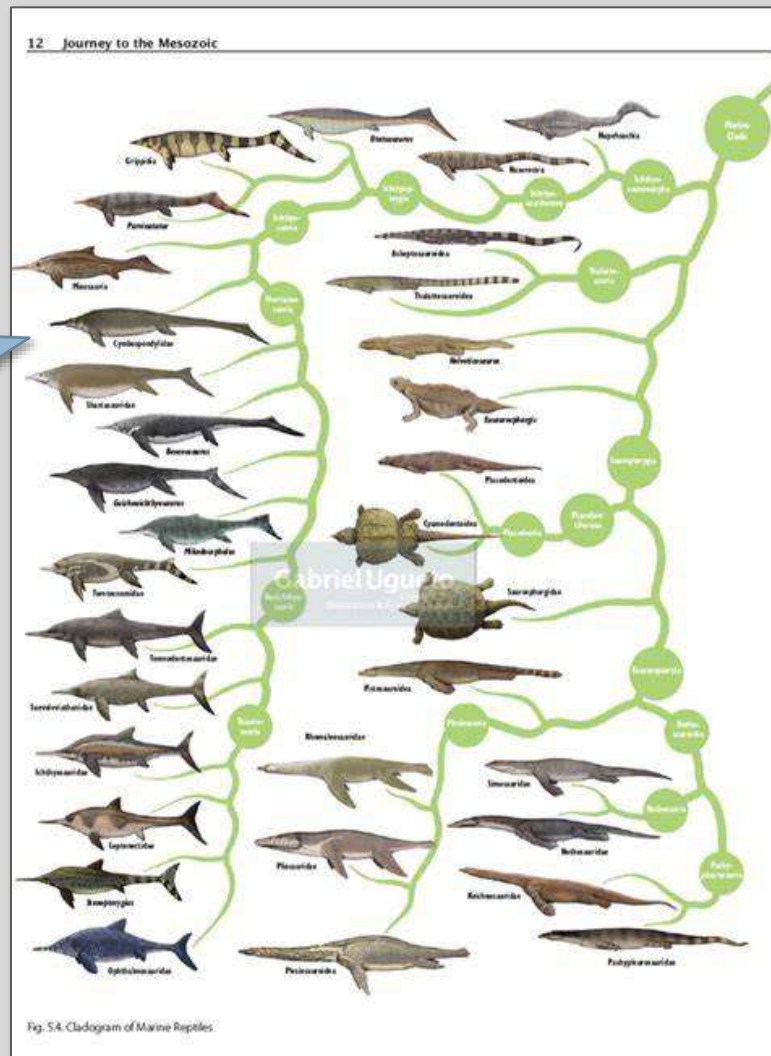
杨德睿·启元实验室

2022/08

模块化教程：项目目标

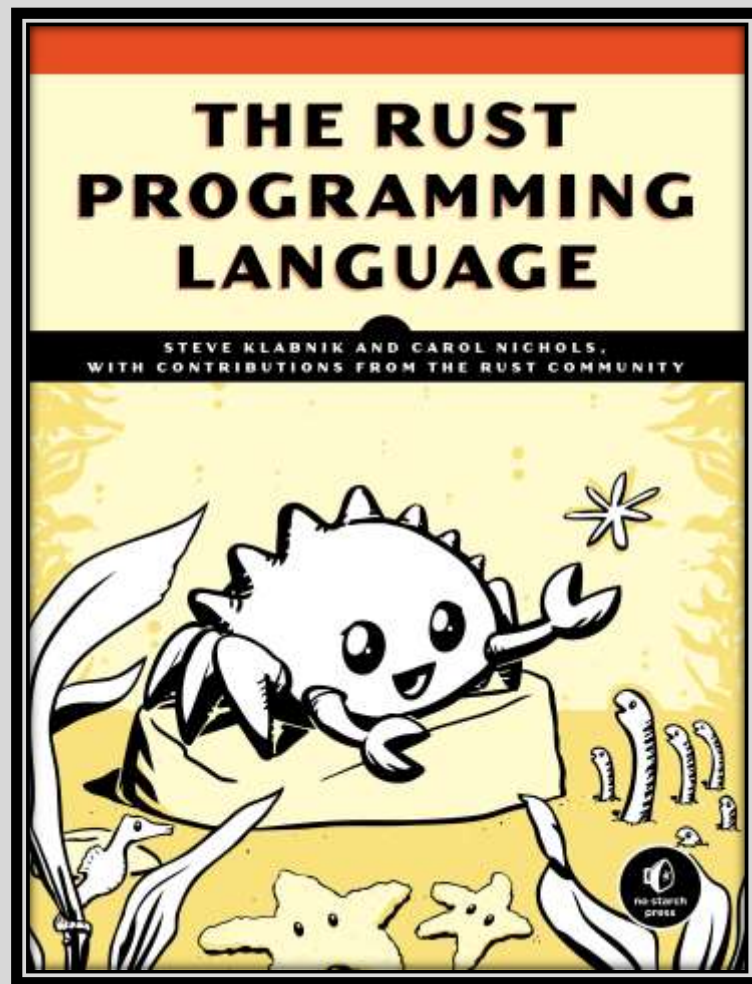
- 当前版本设计
 - 从发展史过渡到功能扩充，学习曲线平滑
 - 章节之间差异小但分散，容易造成困扰
 - 每章一个分支，迁移管理困难
- 优化设计
 - 保持各章功能不变
 - 章节之间差异集中化，共性分离为 lib crate
 - 每章一个 bin crate，在同一个分支里方便管理
- 补充设计
 - 利用 qemu 和 rust 的跨平台特性，支持 windows 学习
 - 支持细节优化，方便学生自主学习

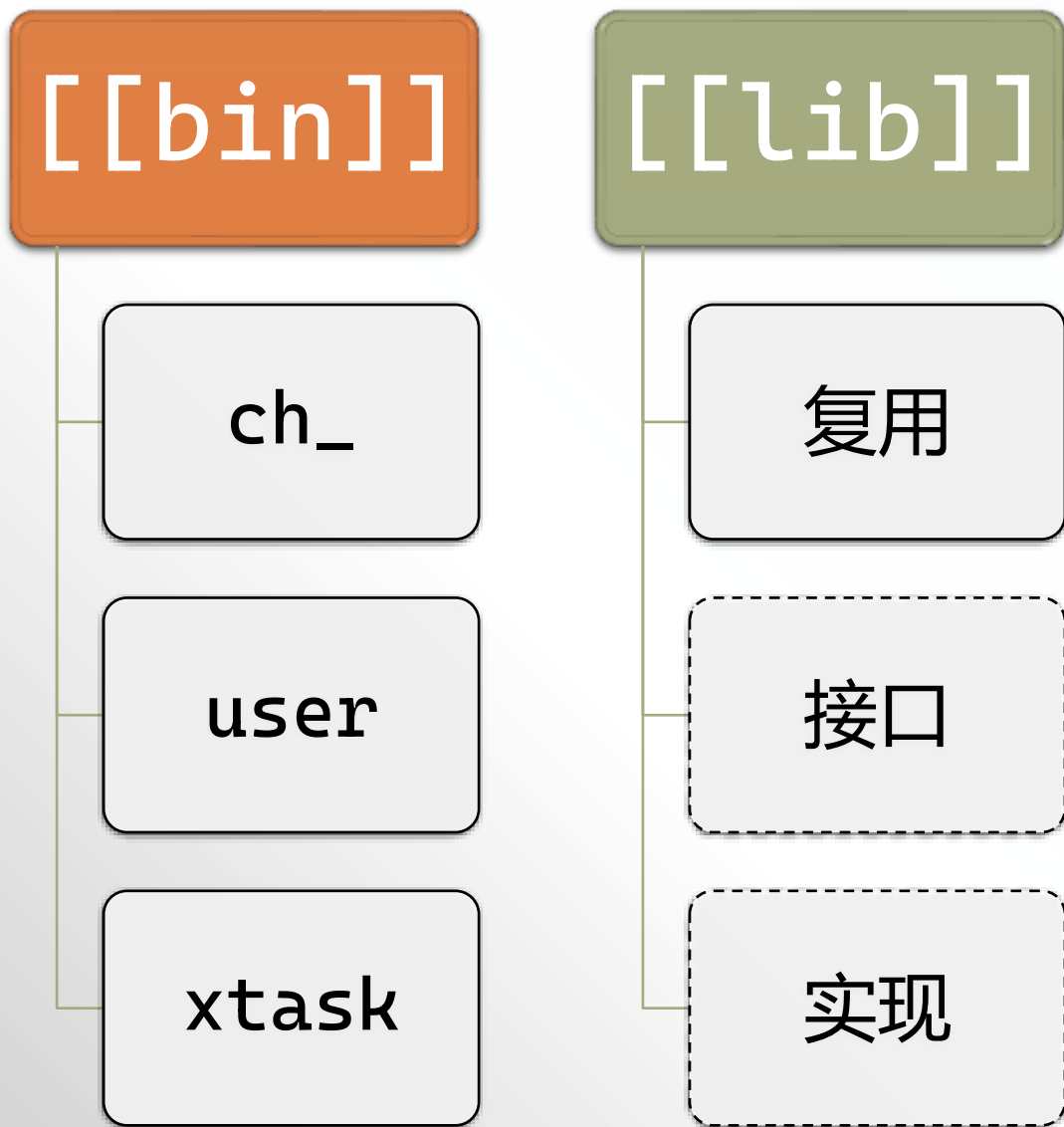
中生代
海爬
演化树



大纲·TOC

- 项目目标
- 模块设计
- 章节实现介绍
 - 第一章：裸机应用程序
 - 第一章实验：依赖注入
 - 第二章：单道批处理系统
 - 控制流切换设计
 - 系统调用分发
 - 第三章：多道批处理系统
 - 中断响应
 - 第四章：地址空间
 - 内存分配：可扩展伙伴分配器
 - 内核堆：分配器级联
 - 内核地址空间：游标式页表访问





模块设计

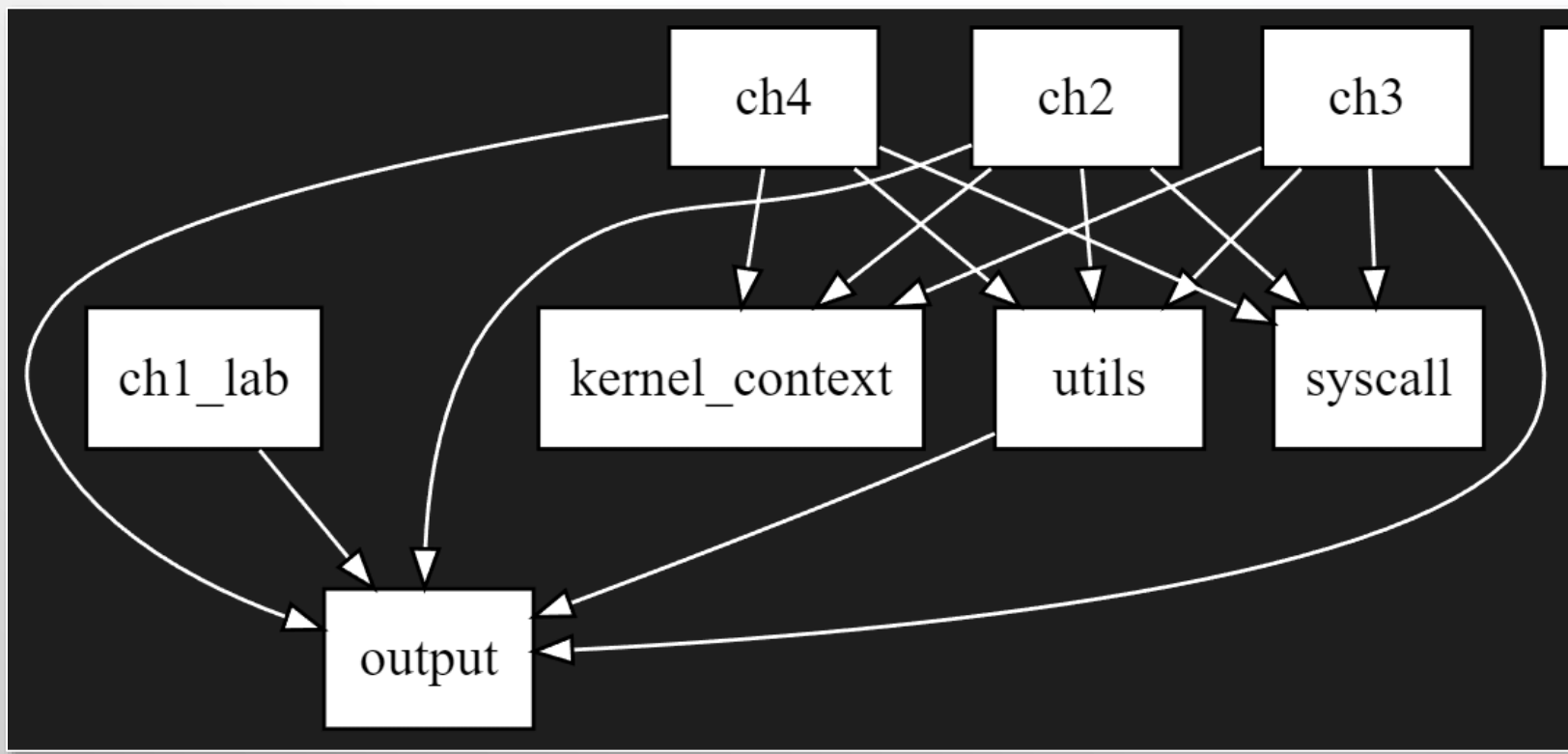
微内核 ≠ 模块化

动态加载 ≠ 模块化

它们是正交的

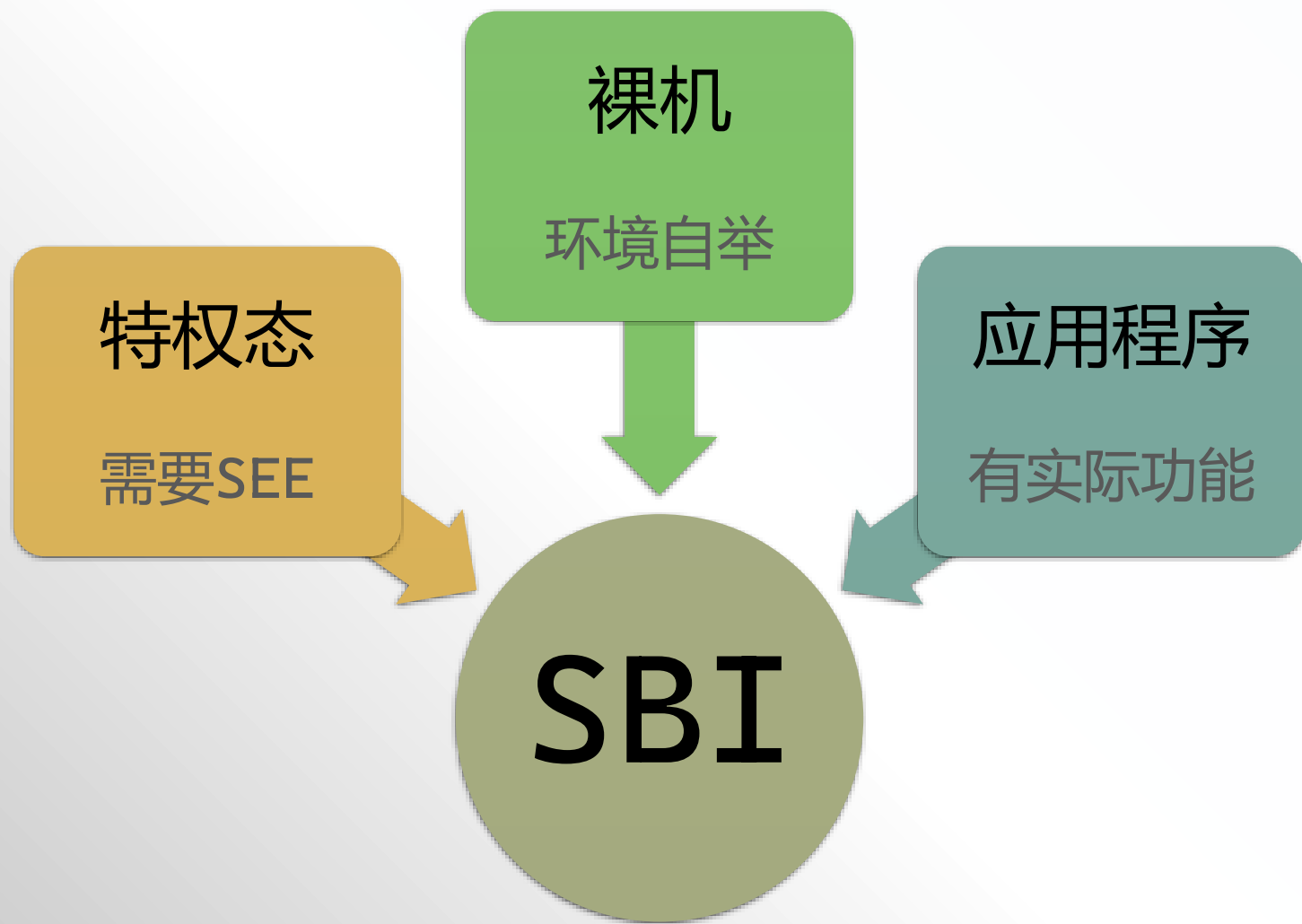
lib crates

lib	来源			描述
		接口	实现	
<i>output</i>	内部模块	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	提供print!/println!/log!
<i>syscall</i>		<input checked="" type="checkbox"/>		定义并分发系统调用
<i>kernel-context</i>				上下文切换
<i>utils</i>				静态加载和其他杂项
<i>sbi-rt</i>	依赖库			提供特权二进制接口调用
<i>page-table</i>		<input checked="" type="checkbox"/>		页式内存管理软硬协同
<i>buddy-allocator</i>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	伙伴分配器（重写）
<i>spin</i>	第三方库			自旋锁
<i>log</i>		<input checked="" type="checkbox"/>		日志



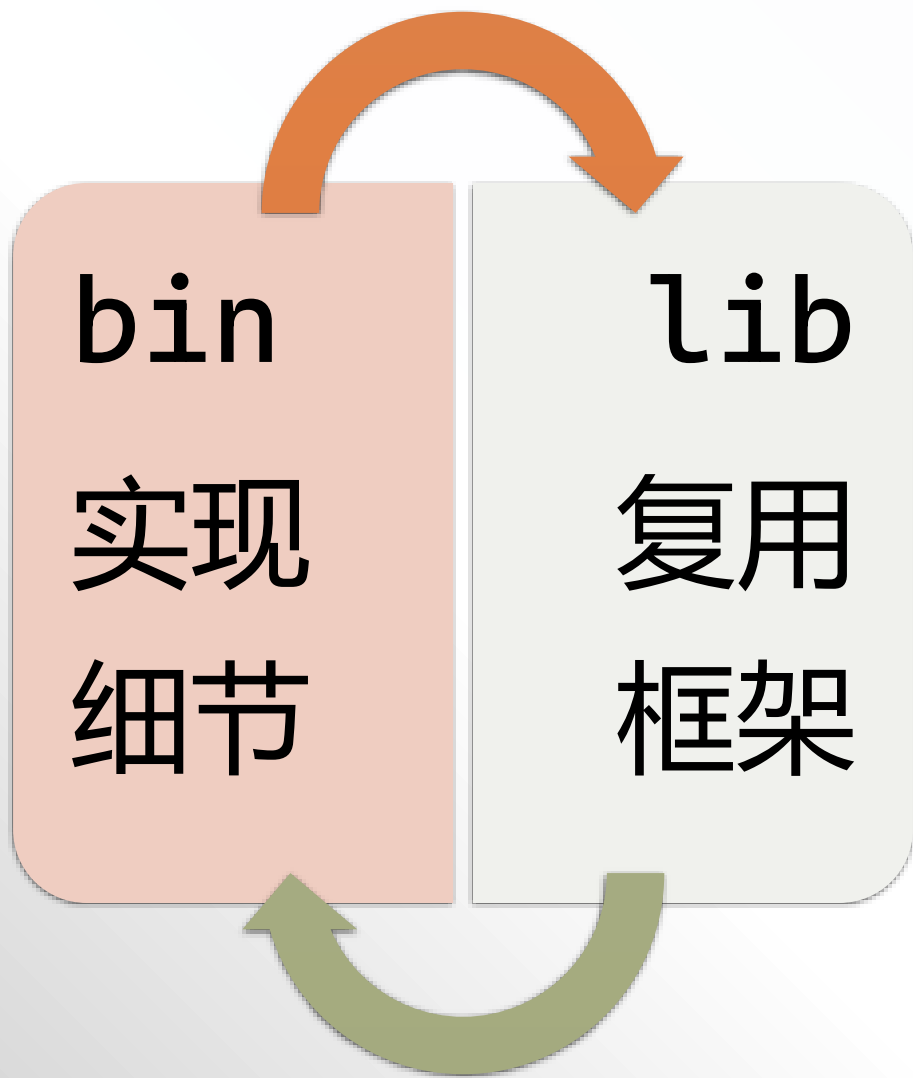
模块依赖关系

使用 rust-analyzer 绘制



第一章

特权态裸机应用程序
保持简单性



第一章实验

依赖注入

output crate

提供	要求
print! println! Log::Log 的一种实现和初始化 Console trait 传入用户实现的函数	实现 Console 并传入静态引用

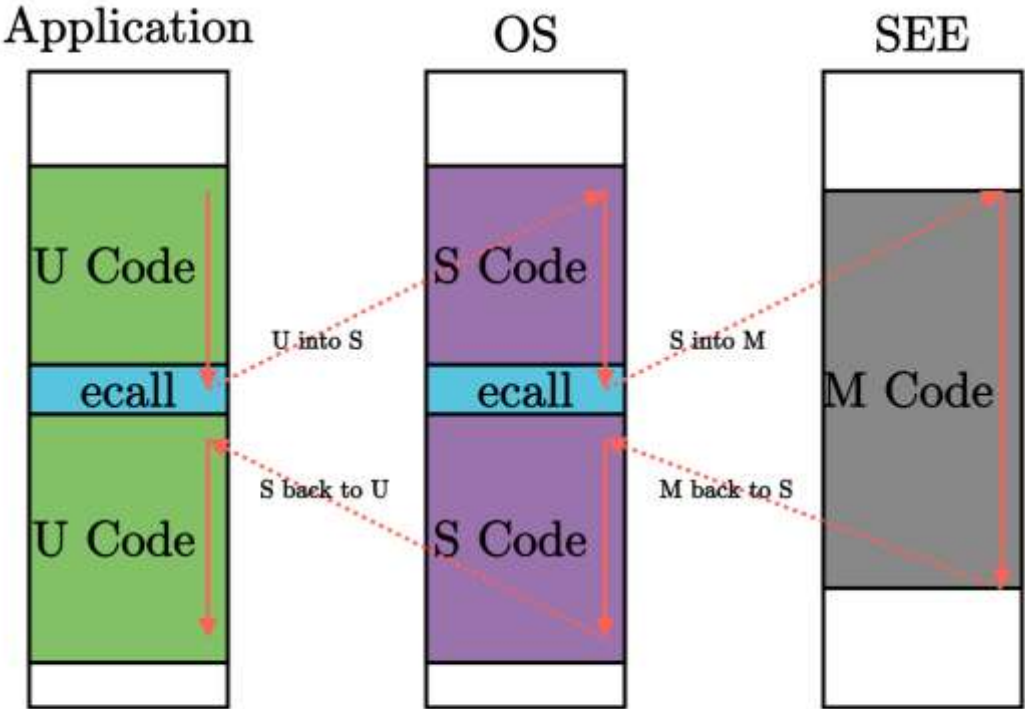
```
/// 将传给 `output` 的控制台对象。
///
/// 这是一个 Unit struct, 它不需要空间。否则需要传一个 static 对象。
1 implementation | 2 references
struct Console;

/// 为 `Console` 实现 `output::Console` trait。
impl output::Console for Console {
    6 references
    ... fn put_char(&self, c: u8) {
    ...     #[allow(deprecated)]
    ...     legacy::console_putchar(c as _);
    ... }
}
```

```
// 初始化 console      You, 3周前 • docs: 补充
init_console(&Console);
// 设置总的日志级别
log::set_max_level(log::LevelFilter::Trace);

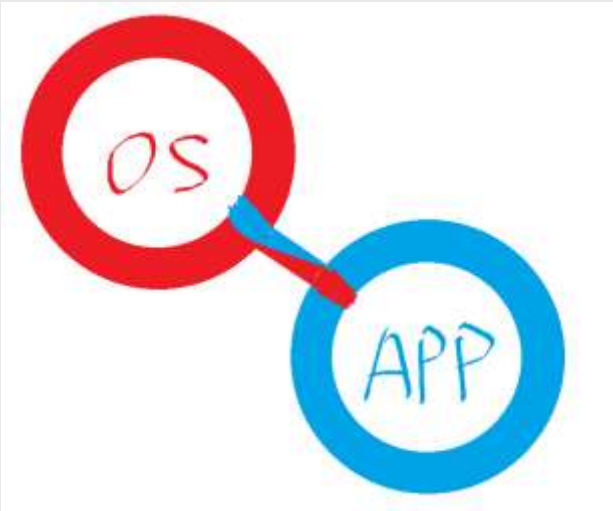
println!("[PRINT] Hello, world!");
log::trace!("Hello, world!");
log::debug!("Hello, world!");
log::info!("Hello, world!");
log::warn!("Hello, world!");
log::error!("Hello, world!");
```

第一人称视角的设计		
机器态	特权态	用户态
☑主要控制流		
内核是函数	调度是函数	系统调用是函数
	SBI 调用是函数	



第二章

线程和特权级切换
提供系统调用



内核设计：线程切换

目标	设计
高内聚（形成模块）	线程切换封装为函数
低耦合（调用简洁）	
零开销	
可扩展	支持用户态/内核态控制流
	支持中断开启/屏蔽

地址空间切换	
内核固定（双页表）	集成为执行器
内核可变（单页表）	分散

```
/// 线程上下文。
#[repr(C)]
1 implementation | 14 references
pub struct LocalContext {
    sctx: usize,
    x: [usize; 31],
    sepc: usize,
    /// 是否以特权态切换。
    pub supervisor: bool,
    /// 线程中断是否开启。
    pub interrupt: bool,
}
```

```
pub unsafe fn execute(&mut self) → usize {
    let sstatus: usize;
    core::arch::asm!("csrc {}, sstatus", out(reg) sstatus);
    let mut sstatus: usize = build_sstatus(sstatus);
    core::arch::asm!(
        "csrc sscratch, {sscratch}
         csrc sepc, {sepc}
         csrc sstatus, {sstatus}
         addi sp, sp, -8
         sd ra, (sp)
         call {execute_naked}
         ld ra, (sp)
         addi sp, sp, 8
         csrc {sepc}, {sepc}
         csrc {sstatus}, {sstatus}
        ",
        sscratch = in(reg) self,
        sepc = inlateout(reg) self.sepc,
        sstatus = inlateout(reg) sstatus,
        execute_naked = sym execute_naked,
    );
    sstatus
}
```

```
unsafe extern "C" fn execute_naked() {
    core::arch::asm!(
        "r" .altmacro...,
        // 位置无关加载
        .option push,
        .option pic,
        // 保存调度上下文
        addi sp, sp, -32*8,
        SAVE_ALL,
        // 设置返回入口
        la t0, if,
        csrc svec, t0,
        // 保存调度上下文地址并切换上下文
        csrc t0, sscratch,
        sd sp, (t0),
        mv sp, t0,
        // 恢复线程上下文
        LOAD_ALL,
        ld sp, 2*8(sp),
        // 执行线程
        sret,
        // 陷入
        .align 2,
        // 切换上下文
        *1: csrcw sp, sscratch, sp,
        // 保存线程上下文
        SAVE_ALL,
        csrcw t0, sscratch, sp,
        sd t0, 2*8(sp),
        // 切换上下文
        ld sp, (sp),
        // 恢复调度上下文
        LOAD_ALL,
        addi sp, sp, 32*8,
        // 返回调度
        ret,
        .option pop,
        options(noreturn)
    );
}
```

异界传送门

```
/// 异界传送门。
///
/// 必须位于公共地址空间中。
#[repr(C)]
0 implementations | 2 references
pub struct ForeignPortal {
    a0: usize,           // (a0) 目标控制流 a0
    ra: usize,           // 1*8(a0) 目标控制流 ra
    satp: usize,         // 2*8(a0) 目标控制流 satp
    sstatus: usize,      // 3*8(a0) 目标控制流 sstatus
    sepc: usize,         // 4*8(a0) 目标控制流 sepc
    stvec: usize,        // 5*8(a0) 当前控制流 stvec (寄存, 不用初始化)
    sscratch: usize,     // 6*8(a0) 当前控制流 sscratch (寄存, 不用初始化)
    execute: [usize; 1024], // 7*8(a0) 执行代码
}
```

只用于内核固定式内核，封装了隔离页表任务的切换执行

用一段汇编实现了：切换地址空间
→ 执行任务 → 回到原来的地址空间

不需要两次完整的通用寄存器切换，而是一次通用寄存器切换和一次跨地址空间执行（即“异界传送”）。只有传送门对象需要在地址空间公共位置，暴露更少

跨地址空间任务也不需要内核栈，共享代码（一段可重定位汇编）和数据都在传送门对象内部。只需要正常的任务上下文加描述地址空间的页表

```
#[cfg(not(feature = "coop"))]  
sbi_rt::set_timer(stime_value: time::read64() + 12500);  
unsafe { tcb.execute() };  
  
use scause::*;  
let finish: bool = match scause::read().cause() {
```

第三章

多道加载

任务状态扩充

系统调用扩充

虚地址空间		
应用程序地址空间	应用程序动态加载	ELF 解析
硬件机制	动态内存分配	页表管理
内核软件	控制流/地址空间同时切换	
	内核堆	

内存分配

可扩展伙伴分配器

分配器级联

页表管理

页式内存管理

游标式访问

第四章

地址空间

内核设计：内存分配

目标	设计
内核堆动态分配	分配器级联（内存池） 伙伴分配器（弃用位图分配器，分离伙伴行和寡头行 trait）
连续页/大页分配	
提高性能	

```
/// 伙伴分配器的一个行。
3 implementations | 8 references
pub trait BuddyLine {
    /// 空集合。用于静态初始化。
    const EMPTY: Self;

    /// 侵入式元数据的大小。
    const INTRUSIVE_META_SIZE: usize = 0;

    /// 伙伴分配器可能需要集合知道自己的阶数和基序号。
    #[inline]
    5 references
    fn init(&mut self, _order: usize, _base: usize) {}

    /// 提取指定位置的元素，返回是否提取到。
    3 references
    #[inline]
    fn take(&mut self, _idx: usize) -> bool {
        unimplemented!()
    }
}
```

```
/// 寡头集合。伙伴分配器的顶层，不再合并。
2 implementations |
pub trait Oligo {
    /// 提取任意一个满足 'align_order' 的内存块。
    ///
    /// 返回提取到的元素。若集合为空则无法提取，返回 ['None']。
    4 references
    fn take_any(&mut self, align_order: usize) -> Option<usize>;

    /// 放入一个元素 'idx'。
    4 references
    fn put(&mut self, idx: usize) -> Option<usize>;
}
```

```
/// 伙伴集合。一组同阶的伙伴。
3 implementations | 9 references
pub trait BuddyCollection: BuddyLine {
    /// 提取任意一个满足 'align_order' 的内存块。
    ///
    /// 返回提取到的元素。若集合为空则无法提取，返回 ['None']。
    4 references
    fn take_any(&mut self, align_order: usize) -> Option<usize>;

    /// 放入一个元素 'idx'。
    ///
    /// 如果 'idx' 的伙伴元素存在，则两个元素都被提取并返回他们在上一层的序号。
    /// 否则 'idx' 被放入集合。
    5 references
    fn put(&mut self, idx: usize) -> Option<usize>;
}
```

伙伴行 (原理)

layer	order	接口	内存块															
4	N+4	寡头	0															
3	N+3	伙伴	0								1							
2	N+2	伙伴	0				1				2				3			
1	N+1	伙伴	0		1		2		3		4		5		6		7	
0	N	伙伴	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- 伙伴分配器是内存块的二叉森林
- 每个层表示的内存范围是一样的
- 层和地址的关系：每层表示一个位
- 层高对性能影响很大，尤其是大块分配
- 根的数量是无所谓的
- 根和枝操作不同

- 改进：最高层使用不同的接口（寡头行）

- 伙伴行的操作
 1. 取出一个块
 2. 放回一个块
 - 如果伙伴存在则两个一起取出
 - 否则放回这个块
- 寡头行的操作
 1. 取出连续的多个块
 2. 放回块（不需要找伙伴）

伙伴行（位图）

```
You, 4天前 | 1 author (You)
4  /// 用一个 usize 作为位图保存占用情况的伙伴行。
5  ///
6  /// - 非侵入式
7  /// - 静态分配，容量有限
8  5 implementations |
9  pub struct UsizedBuddy {
10     bits: usize,
11     base: usize,
12 }
13 > impl UsizedBuddy { ...
28
29 > impl BuddyLine for UsizedBuddy { ...
44
45 > impl OligarchyCollection for UsizedBuddy { ...
82
83 > impl BuddyCollection for UsizedBuddy { ...
121
122 > impl fmt::Debug for UsizedBuddy { ...
```

- 节点数静态有限
 - 出于性能考虑最好一个 `usize`，最多一个 `u128`
 - 如果限制在一个整数里则性能极高
- 非侵入式，不需要寻址到内存块内部
- 实现了所有接口但非常适合寡头

- 注意：通用位图实际上是 N 层块表示 2^K 个 $N-1$ 层块的扩展的伙伴分配器，其操作逻辑是一致的。但实际上只有位图才能实现比较有效率的发现多个连续伙伴块的存在，所以把伙伴分配器扩展出来是没意义的。而且一旦每层不能放在一个整数里，性能也高不到哪去，要找的连续块越多就会产生越多查找失败，性能就越差。想减少分配失败只能减少层间块分裂的数量，其极限就是基本的伙伴分配器。

伙伴行（链表）

- 每行块数量是无限的
- 侵入式，需要修改所管理内存块的内容
- 最小的块是一个链表结点的大小（一个指针）
- 提取一个任意的块可以直接第一个，很快
- 由于总是按顺序取，产生的碎片更少
- 释放块时需要找伙伴，等价于插入排序，需要遍历链表

- 优化方案：
 - 针对一个伙伴行的优化不影响伙伴分配器本体
 - 平衡二叉树 → 最稳定的存取开销
 - 小顶堆 → 取很快，存也不慢
 - 其他更复杂、针对应用的优化

```
/// 侵入式链表伙伴行。
4 implementations | 9 references
pub struct LinkedListBuddy {
    ... free_list: Node,
    ... order: Order,
}

impl BuddyLine for LinkedListBuddy { ...

impl OligarchyCollection for LinkedListBuddy { ...

impl BuddyCollection for LinkedListBuddy { ...

impl fmt::Debug for LinkedListBuddy { ...

You, 上周 | 1 author (You)
#[repr(transparent)]
1 implementation | 9 references
struct Node {
    ... next: Option<NonNull<Node>>,
}

impl Node { ...
```


伙伴分配器

方法

类型

```
67 pub struct BuddyAllocator<
68     const N: usize,
69     O: OligarchyCollection,
70     B: BuddyCollection
71 > {
72     /// 寡头集合。
73     oligarchy: O,
74
75     /// 'N' 阶 'B' 型伙伴集合。
76     buddies: [B; N],
77
78     /// 最小阶数。
79     ///
80     /// 'buddy[0]' 伙伴行分配的内存块的阶数。
81     min_order: usize,
82     You, 4天前 • feat: 完成链表分配器
83
84     /// 空闲容量。
85     free: usize,
86
87     /// 总容量。
88     capacity: usize,
89 }
```

```
impl<const N: usize, O: OligarchyCollection, B: BuddyCollection> BuddyAllocator<N, O, B> {
    /// 最大层数。
    const MAX_LAYER: usize = N;
    /// 寡头支持的最小阶数。
    const O_MIN_ORDER: usize = O::INTRUSIVE_META_SIZE.next_power_of_two().trailing_zeros();
    /// 伙伴支持的最小阶数。
    const B_MIN_ORDER: usize = B::INTRUSIVE_META_SIZE.next_power_of_two().trailing_zeros();

    /// 构造分配器。
    #[inline]
    2 references
    pub const fn new() → Self { ... }

    /// 返回分配器管理的总容量。
    #[inline]
    3 references
    pub fn capacity(&self) → usize { ... }

    /// 返回分配器剩余的空间容量。
    #[inline]
    3 references
    pub fn free(&self) → usize { ... }

    /// 最大阶数。寡头块的阶数。
    #[inline]
    4 references
    const fn max_order(&self) → usize { ... }

    /// 运行时初始化。...
    #[inline]
    2 references
    pub fn init<T>(&mut self, min_order: usize, base: NonNull<T>) { ... }

    /// 将一个 'ptr' 指向的长度为 'usize' 的内存块转移给分配器。...
    #[inline]
    2 references
    pub unsafe fn transfer<T>(&mut self, ptr: NonNull<T>, size: usize) { ... }

    /// 分配可容纳 'T' 对象的内存块。
    5 references
    pub fn allocate_type<T>(&mut self) → Result<(NonNull<T>, usize), BuddyError> { ... }

    /// 分配。...
    1 reference
    pub fn allocate<T>(&mut self) → Result<(NonNull<T>, usize), BuddyError> { ... }

    /// 回收。
    6 references
    pub fn deallocate<T>(&mut self, ptr: NonNull<T>, size: usize) { ... }
} impl BuddyAllocator<N, O, B>
```

```
let mut layer: usize = layer0;
let mut idx: usize = loop {
    /// 从寡头借
    if layer == Self::MAX_LAYER {
        match self.oligarchy.take_any(allocation) {
            Some(idx: usize) ⇒ break idx,
            None ⇒ Err(BuddyError)?,
        }
    }
    /// 从伙伴借
    match self.buddies[layer].take_any(allocation) {
        Some(idx: usize) ⇒ break idx,
        None ⇒ layer += 1,
    }
};
```

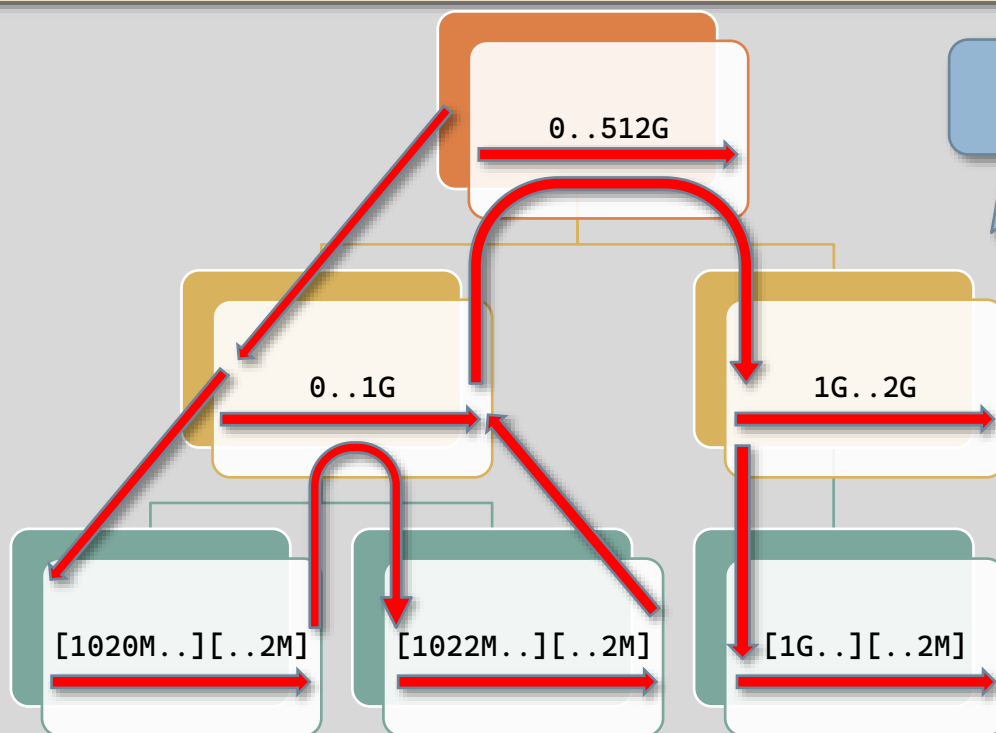
核心分配算法

核心释放算法

```
// 释放
for layer: usize in (order - self.min_order).. {
    /// 释放寡头
    if layer == Self::MAX_LAYER {
        self.oligarchy.put(idx);
        break;
    }
    /// 释放伙伴
    match self.buddies[layer].put(idx) {
        Some(parent: usize) ⇒ idx = parent,
        None ⇒ break,
    }
}
```

内核设计：页表访问

目标	设计
连续访问同一张页表上的连续的项	游标式遍历
尽量减少跨页时上溯的页表级别	



原理

应用：遍历打印

```
Running target\debug\examples\debug.exe
0x0000000000695913a
0x0000000000695913c  0x0000000012345678  -- 0x00000001c0000000  0x00000001c0200000  ( RV
0x0000000000695913d  0x00000000002339c  0x00000001c080c000  0x00000001c080d000  ( X V
0x0000000000695913e  0x00000001c080d000  0x00000001c080e000  0x00000001c080f000  ( X V
0x0000000000695913f  0x00000000002330e  0x00000001c080e000  0x00000001c080f000  ( X V
0x00000000006959140  0x00000000002330f  0x00000001c080f000  0x00000001c0810000  ( X V
0x00000000006959141  0x000000000023310  0x00000001c0810000  0x00000001c0811000  ( X V
0x00000000006959142  0x000000000023311  0x00000001c0811000  0x00000001c0812000  ( X V
0x00000000006959143  0x00000000002331f  0x00000001c081f000  0x00000001c0820000  ( RV
0x00000000006959144  0x000000000023320  0x00000001c0820000  0x00000001c0821000  ( RV
0x00000000006959145  0x000000000023321  0x00000001c0821000  0x00000001c0822000  ( RV
0x00000000006959146  0x000000000023322  0x00000001c0822000  0x00000001c0823000  ( RV
0x00000000006959147  0x000000000023323  0x00000001c0823000  0x00000001c0824000  ( RV
0x00000000006959148  0x000000000023324  0x00000001c0824000  0x00000001c0825000  ( RV
0x00000000006959149  0x000000000023325  0x00000001c0825000  0x00000001c0826000  ( RV
0x0000000000695914a  0x000000000023326  0x00000001c0826000  0x00000001c0827000  ( RV
0x0000000000695914b  0x000000000023327  0x00000001c0827000  0x00000001c0828000  ( RV
```


页表游标：接口

用户设定游标目标，库移动游标

游标命中目标或无法移动时请求用户操作

支持只读访问或边访问边修改

```
/// `Meta` 方案中页表上的一个位置。
#[derive(Clone, Copy)]
4 implementations | 28 references
pub struct Pos<Meta: VmMeta> {
    /// 目标页表项包含的一个虚页号。
    pub vpn: VPN<Meta>
    /// 目标页表项的级别
    pub level: usize,
}

/// 遍历中断时的更新方案。
0 implementations | 4 references
pub enum Update<Meta: VmMeta> {
    /// 修改目标。
    Target(Pos<Meta>),
    /// 新建中间页表。
    Pte(Pte<Meta>, VPN<Meta>),
}
```

```
pub trait Visitor<Meta: VmMeta> {
    /// 出发时调用一次以设置第一个目标。
    ///
    /// `pos` 是页表上最高级别的第一个页的位置。
    3 references
    fn start(&mut self, pos: Pos<Meta>) -> Pos<Meta>;

    /// 到达 `target_hint` 节点。
    3 references
    fn arrive(&mut self, pte: Pte<Meta>, target_hint: Pos<Meta>) -> Pos<Meta>;

    /// 在访问 `target` 的过程中，经过一个包括
    ///
    /// 以下两种情况会调用这个方法：
    ///
    /// - 访问到包含目标虚页的大页节点；
    /// - 访问到包含目标虚页的无效节点；
    3 references
    fn meet(&mut self, level: usize, pte: Pte<Meta>, target_hint: Pos<Meta>) -> Pos<Meta>;
} trait Visitor
```

```
pub trait Decorator<Meta: VmMeta> {
    /// 出发时调用一次以设置第一个目标。
    ///
    /// `pos` 是页表上最高级别的第一个页的位置。
    1 reference
    fn start(&mut self, pos: Pos<Meta>) -> Pos<Meta>;

    /// 到达 `target_hint` 节点。
    1 reference
    fn arrive(&mut self, pte: &mut Pte<Meta>, target_hint: Pos<Meta>) -> Pos<Meta>;

    /// 在访问 `target` 的过程中，经过一个包括
    ///
    /// 以下两种情况会调用这个方法：
    ///
    /// - 访问到包含目标虚页的大页节点；
    /// - 访问到包含目标虚页的无效节点；
    1 reference
    fn meet(&mut self, level: usize, pte: Pte<Meta>, target_hint: Pos<Meta>) -> Update<Meta>;
} trait Decorator
```

页表游标：遍历打印页表

```
struct FmtVisitor<'f1, 'f2, Meta: VmMeta> {  
    f: &'f1 mut fmt::Formatter<'f2>,  
    max_level: usize,  
    new_line: bool,  
    _phantom: PhantomData<Meta>,  
}  
  
impl<'f1, 'f2, Meta: VmMeta> Visitor<Meta> for FmtVisitor<'f1, 'f2, Meta> {  
    #[inline]  
    2 references  
    fn start(&mut self, pos: Pos<Meta>) → Pos<Meta> {  
        self.max_level = pos.level;  
        // 总是从头开始  
        pos  
    }  
  
    2 references  
    fn arrive(&mut self, pte: Pte<Meta>, target_hint: Pos<Meta>) → Pos<Meta> {  
        2 references  
        #[inline]  
        fn meet(&mut self, _level: usize, _pte: Pte<Meta>, _target_hint: Pos<Meta>) {  
            // 不会跳着遍历  
            unreachable!()  
        }  
    }  
}  
impl Visitor for FmtVisitor<Meta>
```

```
impl<Meta: VmMeta, F: Fn(PPN<Meta>) → VPN<Meta>> fmt::Debug  
for PageTableShuttle<Meta, F> {  
    385 references  
    fn fmt(&self, f: &mut fmt::Formatter<'_>) → fmt::Result {  
        self.walk(visitor: FmtVisitor {  
            f,  
            max_level: 0,  
            new_line: true,  
            _phantom: PhantomData,  
        });  
        Ok(())  
    }  
}
```

```
println!(  
    "{:?}",  
    PageTableShuttle {  
        table: root,  
        f: |ppn| VPN::new(ppn.val())  
    }  
);  
You, 现在 · Uncommitted change
```

页表游标：内核虚存映射


```
pub struct KernelSpaceBuilder;

impl Decorator<Sv39> for KernelSpaceBuilder {
    #inline
    2 references
    fn start(&mut self, _: Pos<Sv39>) → Pos<Sv39> {
        Pos::new(vpn: VAddr::new(__text as usize).floor(), level: 0)
    }

    #inline
    2 references
    fn arrive(&mut self, pte: &mut Pte<Sv39>, target_hint: Pos<Sv39>) → Pos<Sv39> {
        let addr: usize = target_hint.vpn.base().val();
        let bits: usize = if addr < __transit as usize {
            0b1011 // X_RV ← .text
        } else if addr < __rodata as usize {
            0b1111 // X_RV ← .trampoline
        } else if addr < __data as usize {
            0b0011 // _RV ← .rodata
        } else if addr < __end as usize {
            0b0111 // _RV ← .data + .bss
        } else {
            return Pos::stop(); // end of kernel sections
        };
        *pte = unsafe { VmFlags::from_raw(bits) }.build_pte(ppn: PPN::new(target_hint.vpn.val()),
            target_hint.next())
    }
}
```

```
let mut shuttle: PageTableShuttle<Sv39>, |...| → ...> = PageTableShuttle {
    table,
    f: |ppn| VPN::new(ppn.val()),
};
shuttle.walk_mut(KernelSpaceBuilder);
println!("{shuttle:?}");
unsafe { satp::set(satp: Mode::Sv39, asid: 0, ppn: kernel_root.floor().val()) };

let ppn: PageNumber<Sv39>, physbase: PPN::new(vpn.val());
Update::Pte(unsafe { VmFlags::from_raw(1) }.build_pte(ppn, vpn)
}
} impl Decorator for KernelSpaceBuilder
```

 **page-table** 0.0.5

Follow

Encapsulation of the page table for common architectures.
[#mmu](#) [#pagetable](#)

[Readme](#) [5 Versions](#) [Dependencies](#) [Dependents](#) [Settings](#)

页表


[crates.io](#) [v0.0.5](#) [license](#) [MIT](#) [CI](#) [passing](#) [docs](#) [passing](#) [issues](#) [opened](#)


封装页表的构建和查询操作。


架构支持：

- ☒ RISC-V
- ☐ x86_64
- ☐ aarch64

Metadata

 20 minutes ago

 MIT


 11.4 kB

Install


Add the following line to your Cargo.toml file:

page-table = "0.0.5"


Documentation

 [docs.rs/page-table/0.0.5](#)

Repository

 [github.com/YdrMaster/page-L...](#)


Owners


 YdrMaster

Categories

- No standard library

Stats Overview

 **3,151**
Downloads all time

 **5**
Versions published

欢迎试用！

#![deny(missing_docs)]



欢迎提问!