zCore 内核对象及系统调用

zCore Summer of Code Day4

潘庆霖 王润基

2020.08.06 @ 鹏城实验室

提纲

- Fuchsia & Zircon 整体结构
- Zircon 内核对象
- Zircon 和 zCore 中内核对象的实现
- zCore 系统调用的实现
- zCore 用户线程生命周期

The Fuchsia layer cake

Topaz

implements interfaces defined by underlying layers & contains four major categories of software:
modules, agents, shells, and runners.

- modules include the calendar, email, and terminal modules;
- shells include the base shell and the user shell;
- agents include the email and chat content providers;
- runners include the Web, Dart, and Flutter runners.

Peridot

provides the services needed to create a cohesive, customizable, multi-device user experience assembled from modules, stories, agents, entities, and other components.

device, user, and story runners. and the ledger and resolver, as well as the context and suggestion engines

Garnet

provides device-level system services for software installation, administration, communication with remote systems, and product deployment.

network, media, and graphics services. and the package management and update system

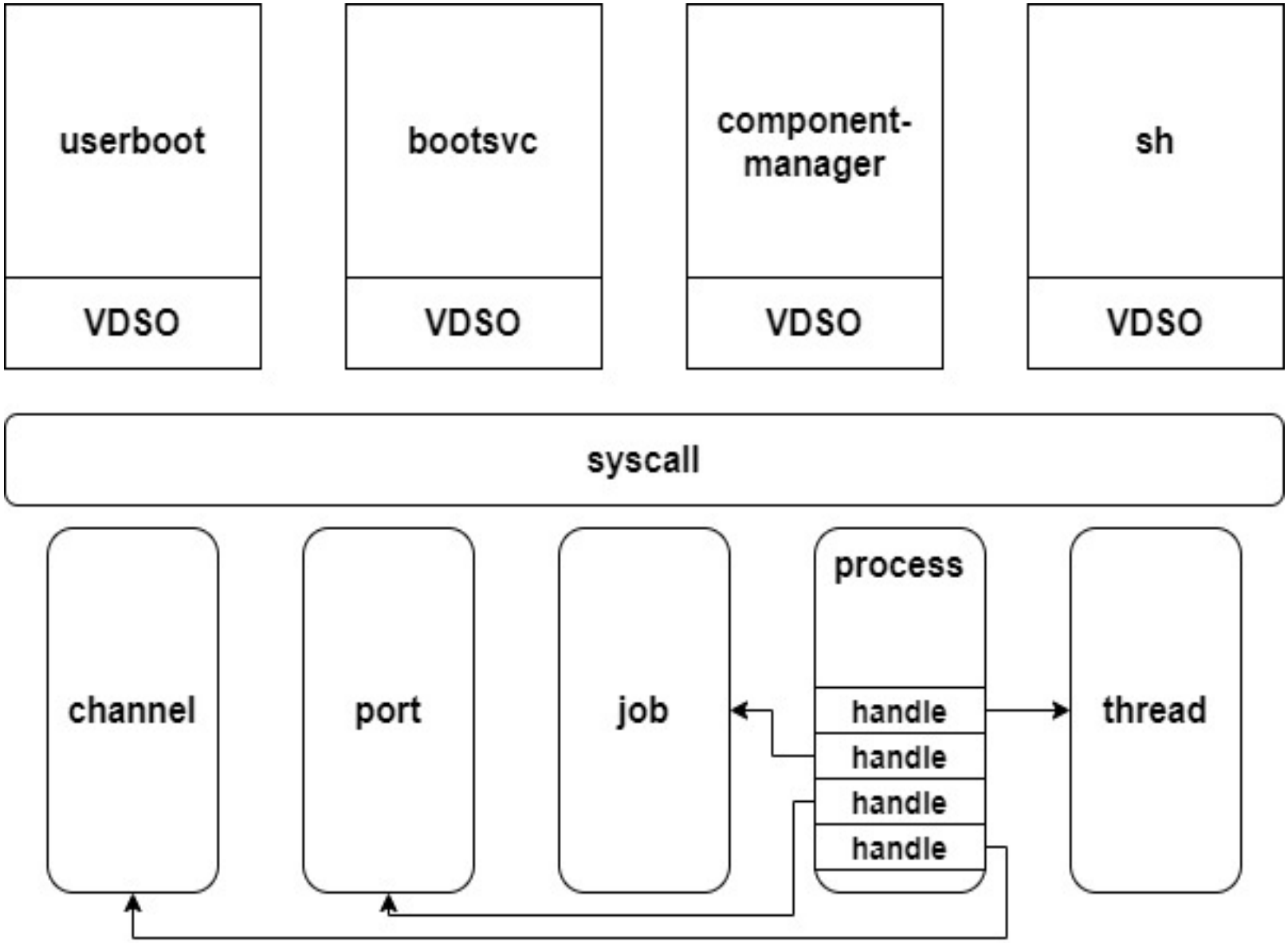
Zircon

mediates hardware access, implements essential software abstractions over shared resources, and provides a platform for low-level software development.

kernel, device manager, most core and first-party device drivers, and low-level system libraries, such as libc and fdio. and defines the Fuchsia IDL (FIDL)

Zircon 内核特点

- 实用主义微内核
- 使用 C++ 实现，支持 x86_64 和 ARM64
- 面向对象：将功能划分到内核对象
- 默认隔离：使用 Capability 进行权限管理
- 安全考量：强制地址随机化，使用 vDSO 隔离系统调用


userboot
bootsvc
component-
manager
sh
VDSO
VDSO
VDSO
VDSO
syscall
channel
port
job
process
handle
handle
handle
handle
thread

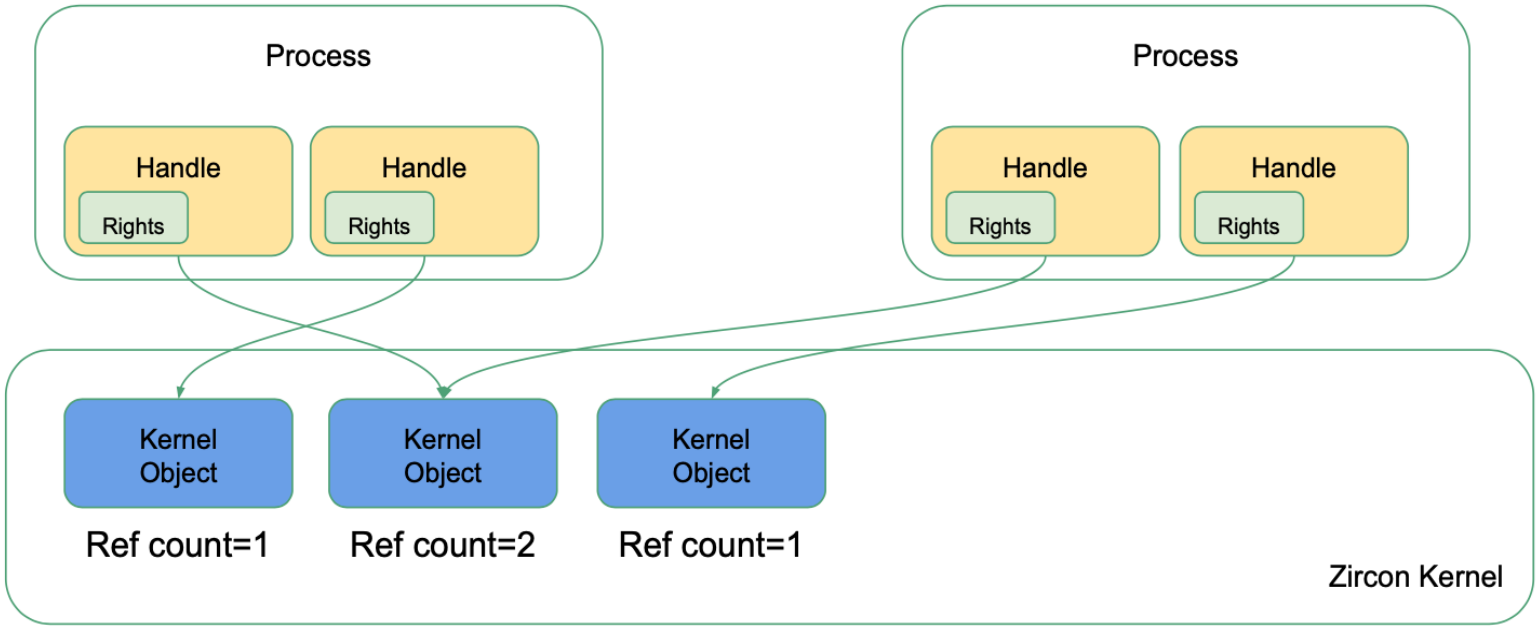
Zircon 内核对象

Everything can be KernelObject

- 任务：Job, Process, Thread, Exception
- 内存：VMAR, VMO, Pager, Stream
- IPC：Channel, FIFO, Socket
- 信号：Event, Timer, Port, Futex
- 驱动：Resource, Interrupt, PCI ...

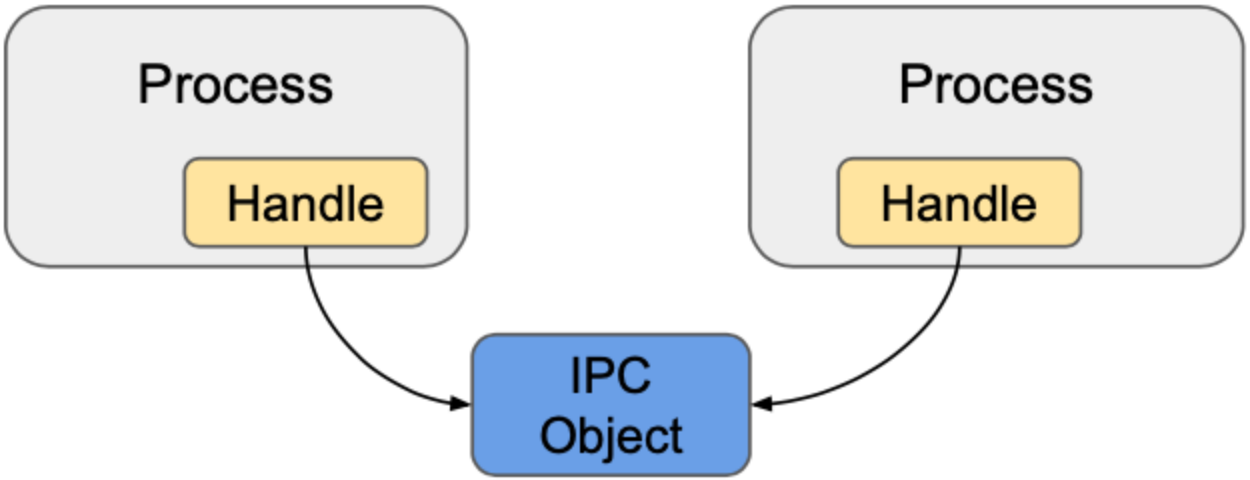
Object

- Object：内核对象
- Rights：对象访问权限
- Handle = Object + Rights：对象句柄（类似 fd）



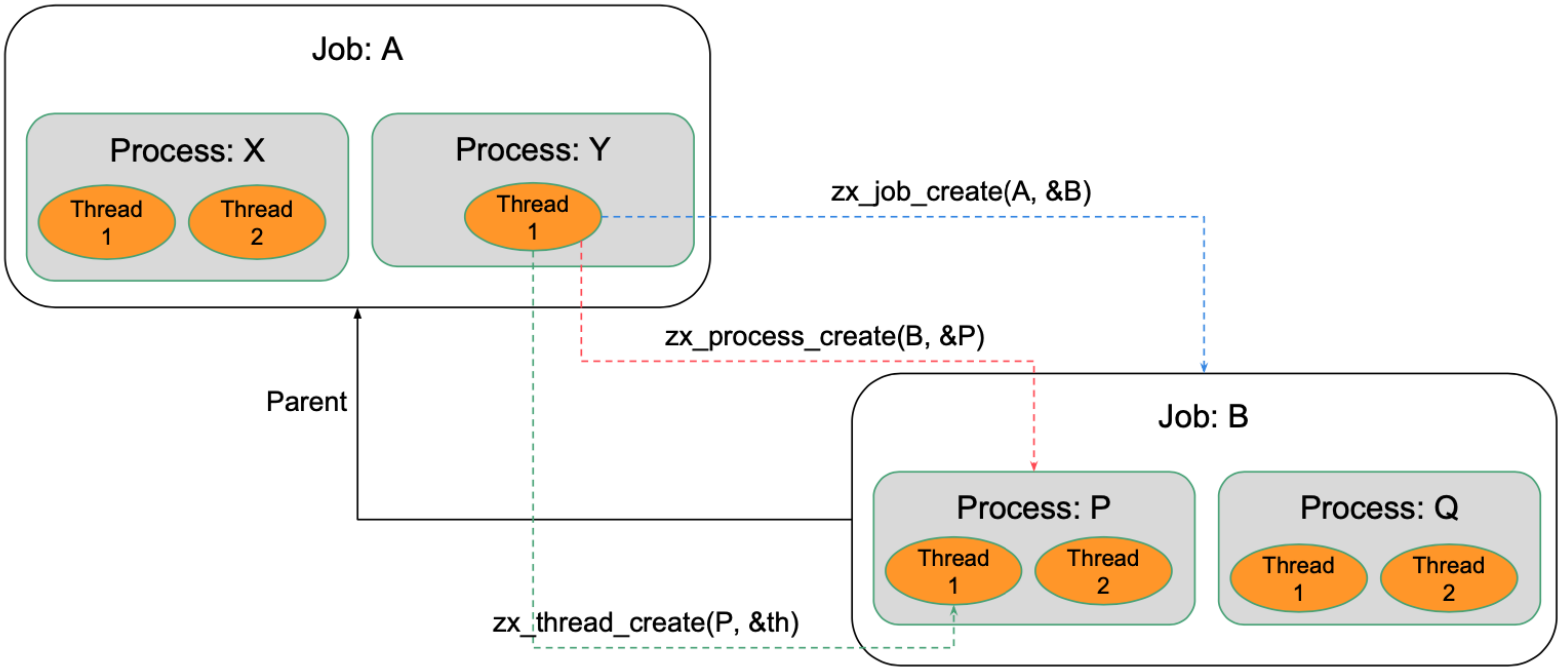
IPC

- Channel：进程间通信基础设施，可以传递数据和 handle
- FIFO：报文数据传输
- Socket：流数据传输



Tasks

- Job：作业，负责控制权限（类似容器）
- Process：进程，负责管理资源
- Thread：线程，负责调度执行



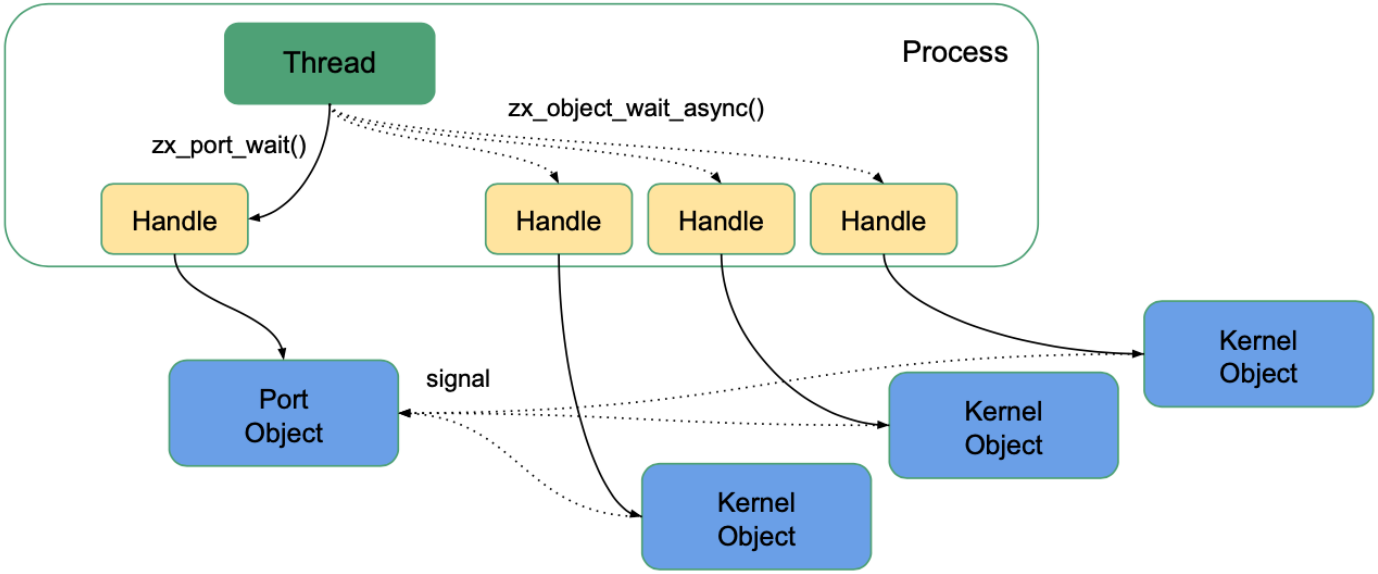
Memory and address space

- VMO: Virtual Memory Object
  - Paged：分页物理内存，支持写时复制
  - Physical：连续物理内存
- VMAR: Virtual Memory Address Region
  - 代表一个进程的虚拟地址空间
  - 树状结构
- Pager：用户态分页机制

Signaling and Waiting

每个 Object 有 32 个信号位，用户程序可以阻塞等待。

- Event (Pair)：事件源/对
- Timer：计时器
- Futex：用户态同步互斥机制
- Port：事件分发机制（类似 epoll）

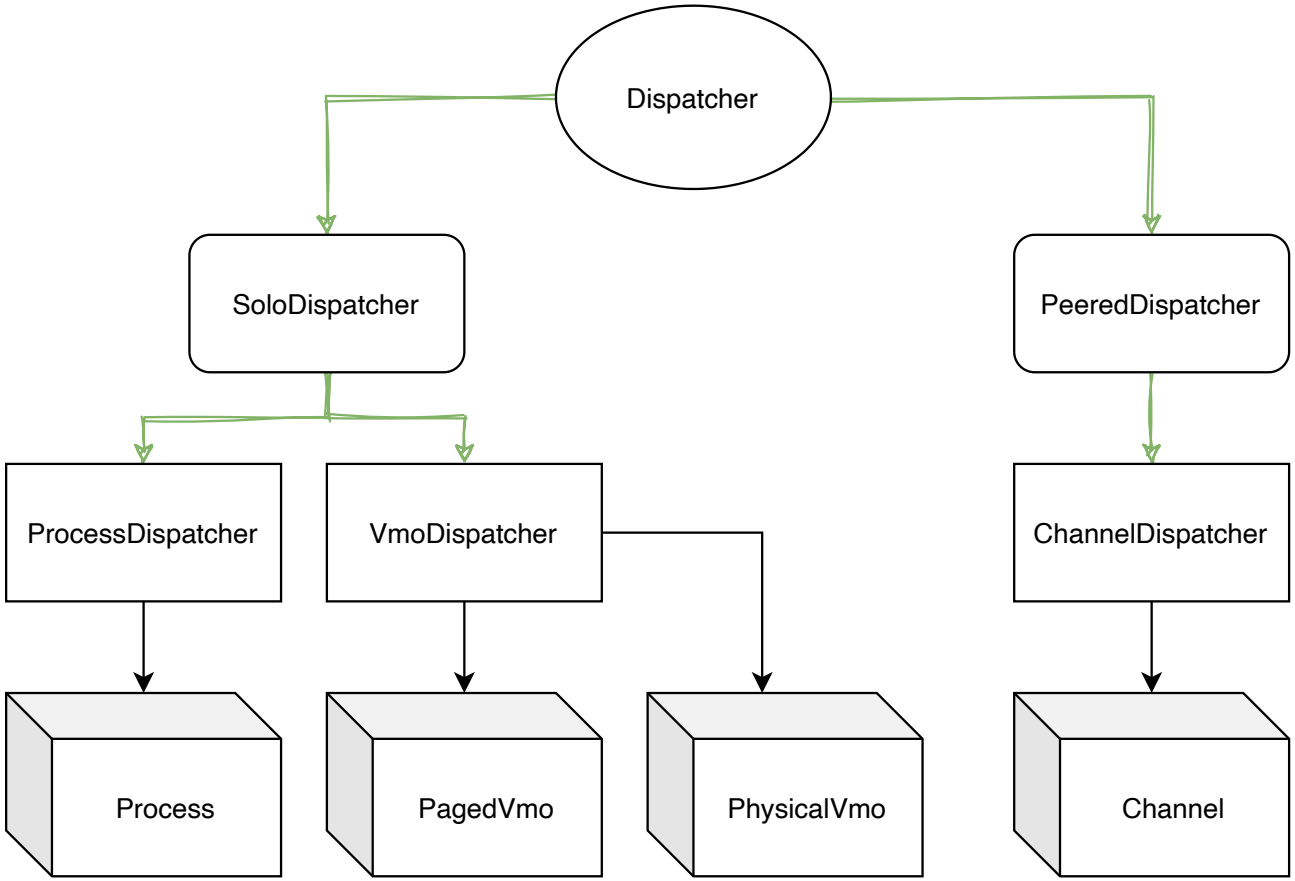


内核对象的实现

- Zircon（C++）
- zCore（Rust）

Zircon 中的 C++ 实现

面向对象：各内核对象继承公共基类，各自进行函数重载



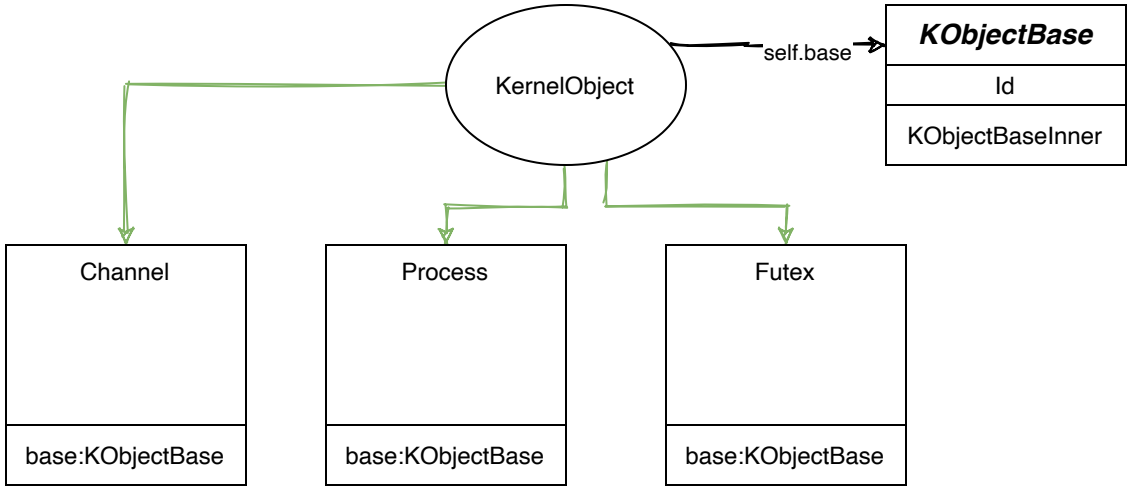
zCore 中的 Rust 实现：Handle

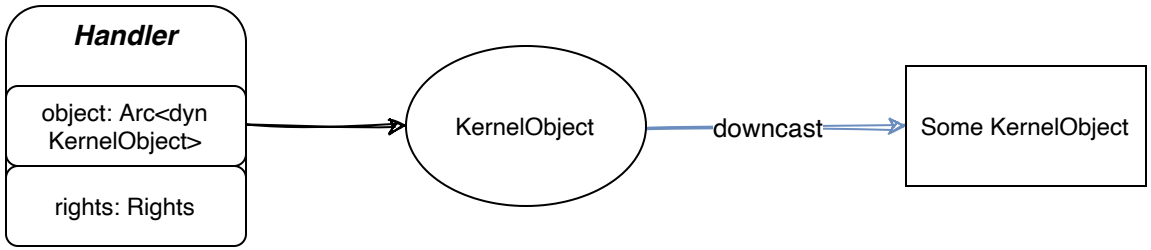
- Arc 引用计数指针
- 对 Object 进行操作的权限维护

```rust
pub struct Handle {
    /// The object referred to by the handle.
    pub object: Arc<dyn KernelObject>,
    /// The handle's associated rights.
    pub rights: Rights,
}
```

zCore 中的 Rust 实现：内核对象

struct `KObjectBase` + trait `KernelObject`

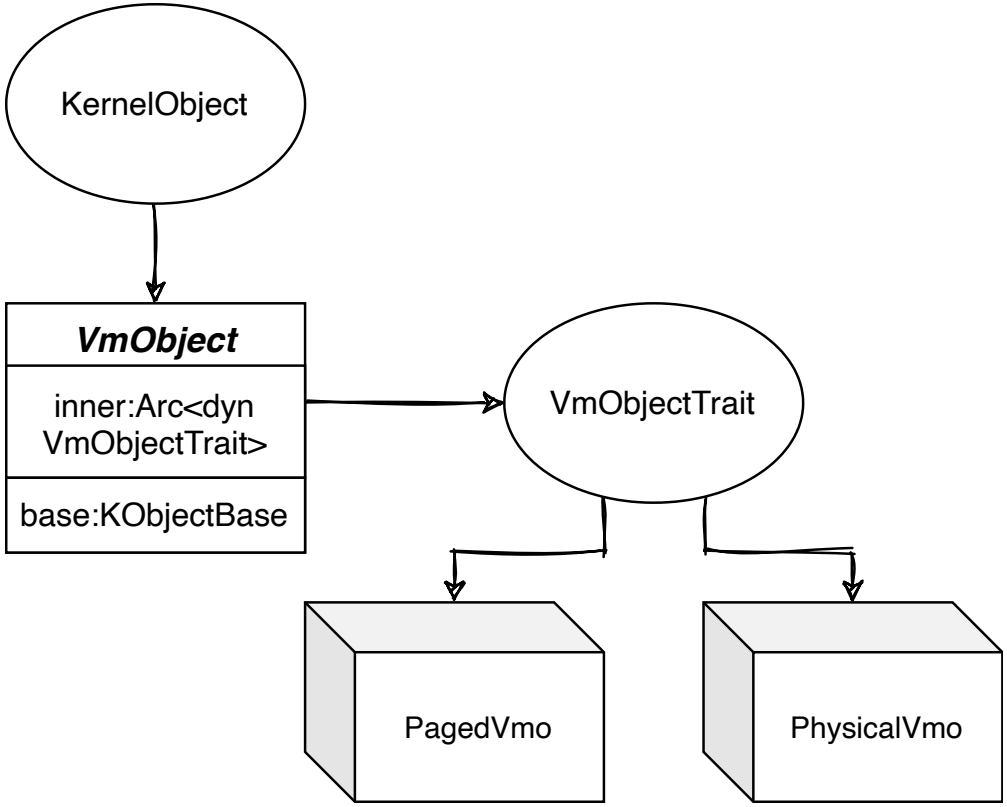




具体可参考：zCore Tutorial

zCore 中的 Rust 实现：VMO

用两层 trait 实现多级继承



zCore 系统调用的实现

- 构造 `Syscall` 结构体：保存上下文信息
- 调用 `syscall()` 函数：请求分发和结果处理
- `sys_object_function()`：具体处理系统调用

`Syscall` 结构体：保存上下文信息

目的：避免访问全局变量

```rust
// zircon-syscall/src/lib.rs

pub struct Syscall<'a> {
    /// 常用寄存器
    pub regs: &'a mut GeneralRegs,
    /// 指向当前线程实例的引用
    pub thread: Arc<Thread>,
    /// 可用于创建新线程的函数
    pub spawn_fn: fn(thread: Arc<Thread>),
    /// 当前线程在执行完该系统调用后是否退出
    pub exit: bool,
}
```

- 构建 `Syscall` 实例
- 调用 `syscall` 接口，进入系统调用处理例程

`syscall()` 函数：请求分发和结果处理

```
// zircon-syscall/src/lib.rs
impl Syscall {
    pub async fn syscall(
        &mut self, num: u32, args: [usize; 8]
    ) -> isize
    {
        let ret: ZxResult = match num {
            Sys::HANDLE_CLOSE =>
                self.sys_handle_close(...),
            Sys::OBJECT_WAIT_ONE =>
                self.sys_object_wait_one(...).await,
            ...
        };
        match ret {
            Ok(_) => 0,
            Err(err) => err as isize,
        }
    }
}
```

实现具体系统调用的套路

- 记录 log 信息：`info!()`
- 检查参数合法性，从用户指针读入数据
- 获取当前进程对象：`self.thread.proc()`
- 根据句柄从进程中获取对象，检查类型和权限
- 调用内核对象 API 执行具体功能
- 检查结果，向用户指针写入数据

注：中途每一步都可能返回错误：`?`

zCore 用户线程生命周期

创建

- 第一个线程 `userboot` 在 `zircon-loader` 中构造
- 后面的线程由系统调用 `zx_thread_create` 创建

启动

- 由 `thread.start(.., spawn_fn)` 开始运行线程
- `spawn_fn` 函数创建一个内核协程，用来执行此用户线程

运行

`zircon-loader/src/lib.rs: spawn`

最外层一个大 loop，重复执行以下过程：

- 等待线程就绪：`thread.wait_for_run().await`
- 获取用户态上下文：`let cx = ...`
- 进入用户态运行：`kernel_hal::context_run(&mut cx)`
- 检查返回原因，做不同处理：`match cx.trap_num {...}`
- 归还上下文：`thread.end_running(cx)`
- 检查是否退出：`if exit { break; }`

谢谢大家