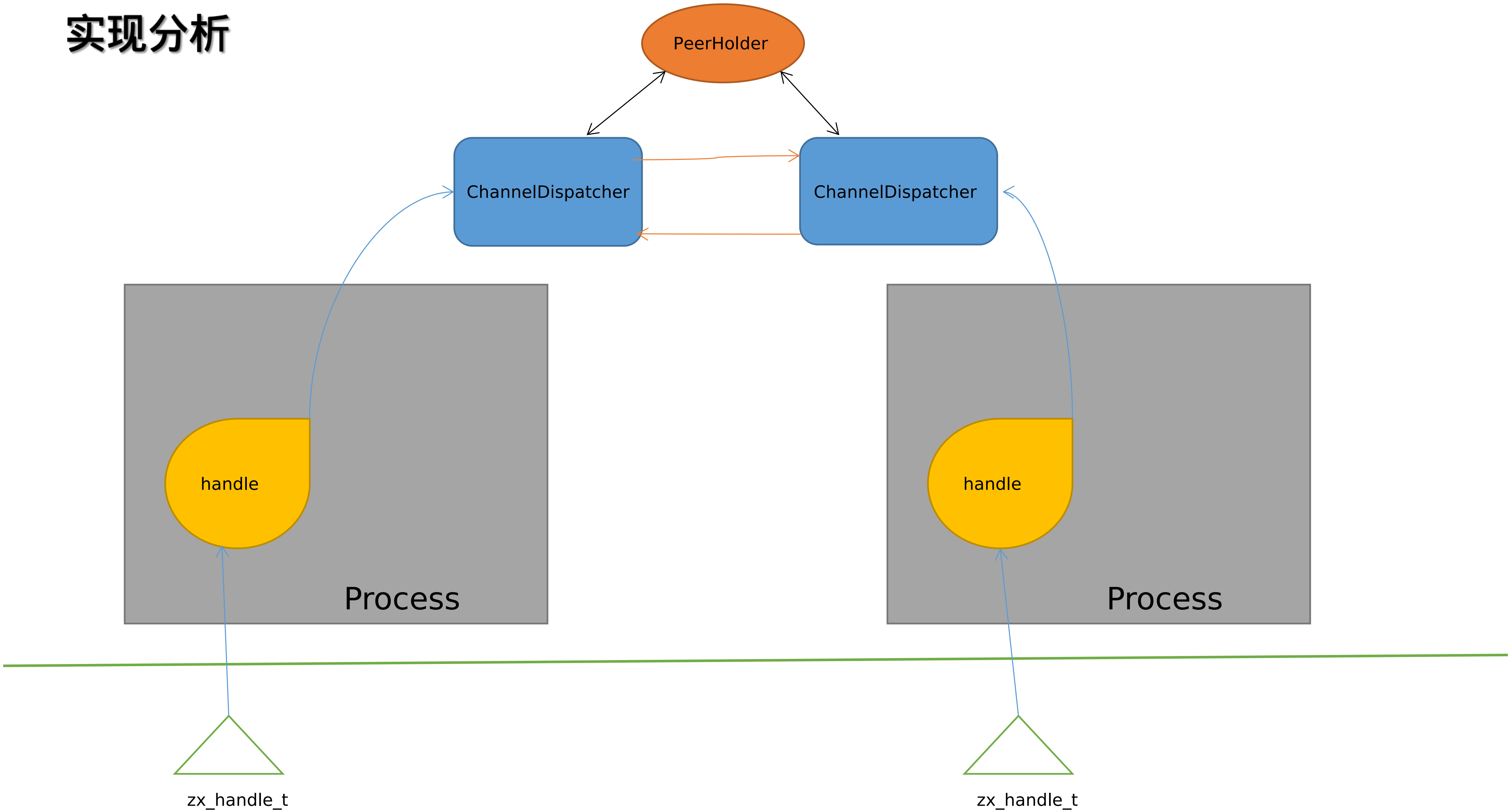
IPC: Channel

2019.11.22

主要用途

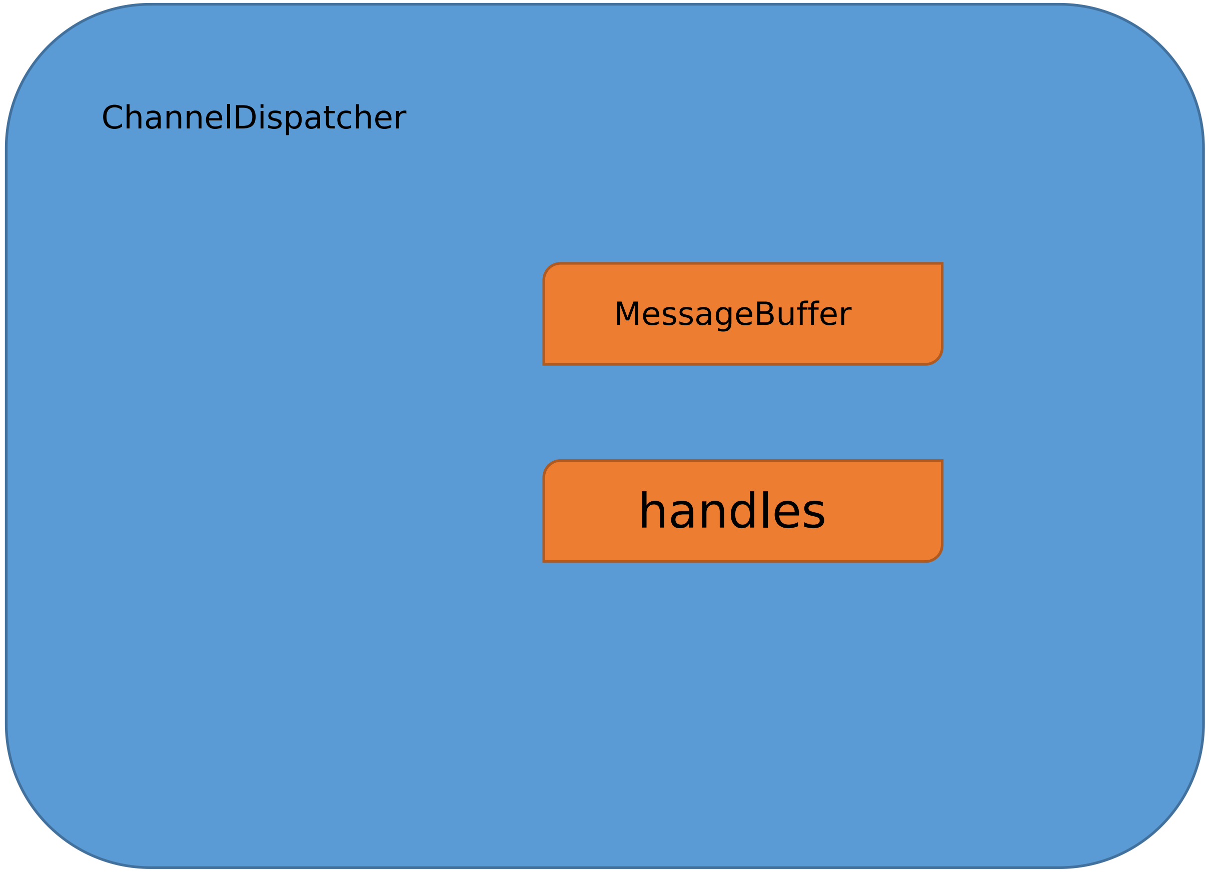
- 最基本的IPC模型
- 可以通过buffer传递消息/handles

实现分析



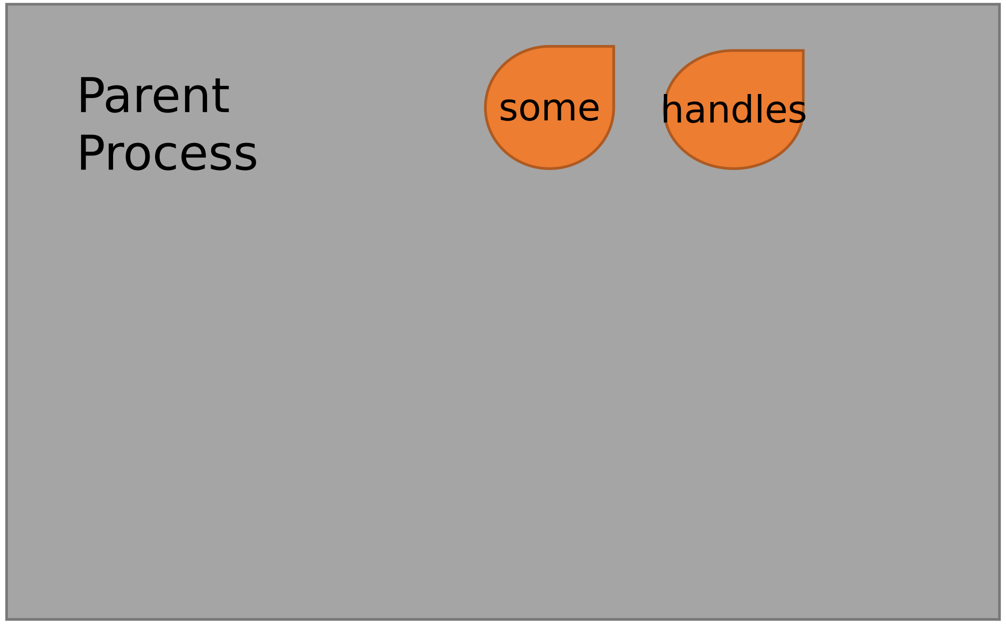
PeerHolder
ChannelDispatcher
ChannelDispatcher
handle
Process
handle
Process
zx_handle_t
zx_handle_t

实现分析



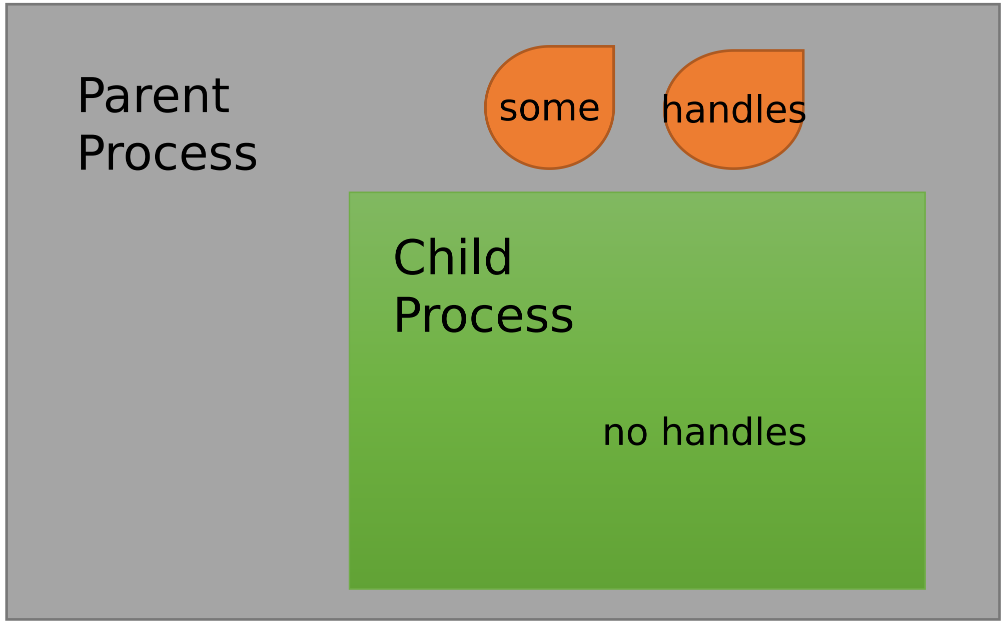
典型用法

- 此时，一个父进程经过：



典型用法

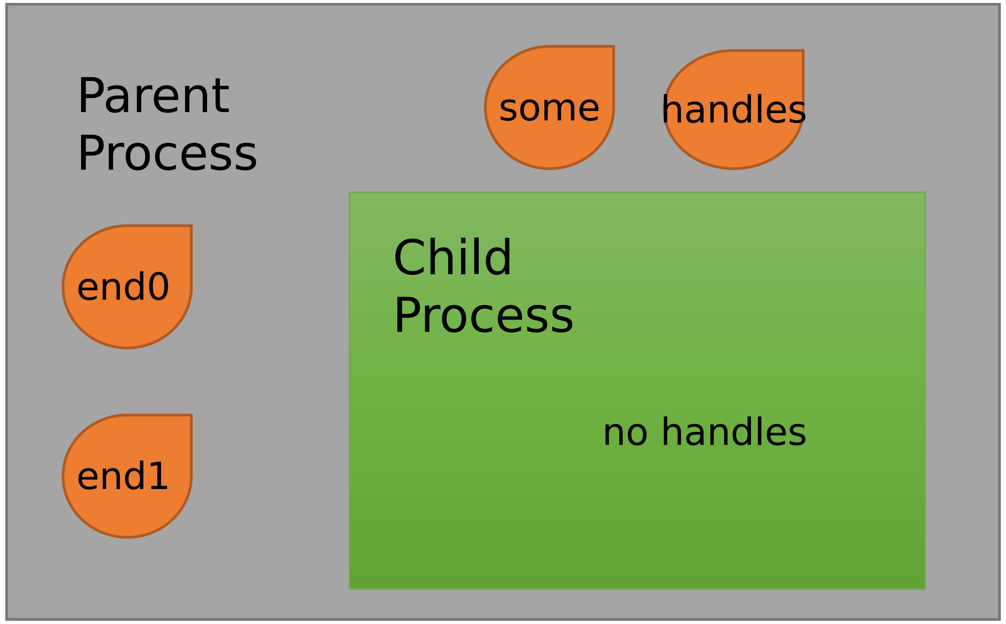
- 准备生一个子进程……：



孩子如何继承家里的
kobj？？？

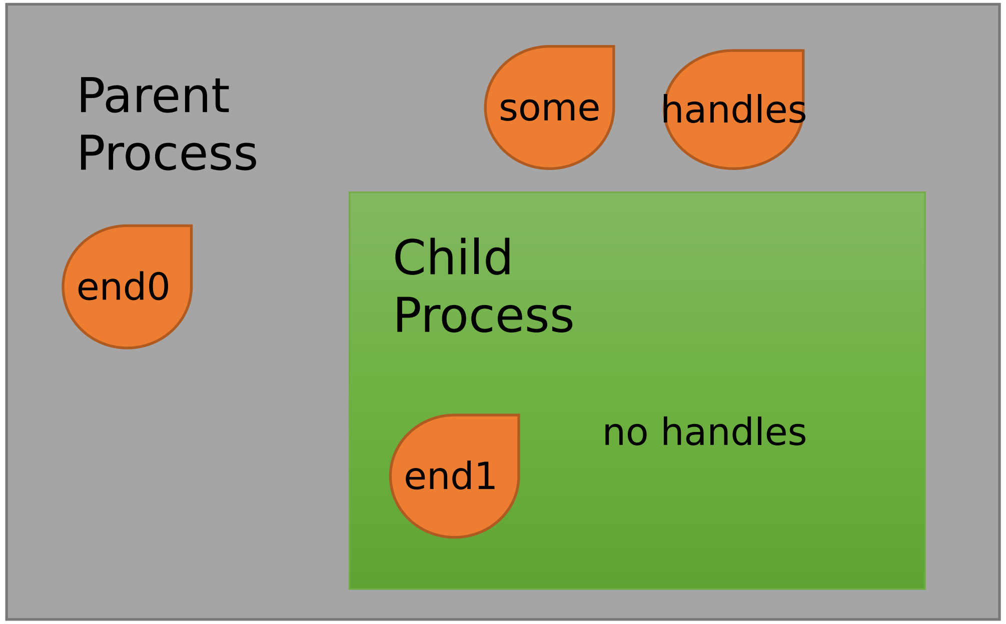
典型用法

- 创建一个channel：



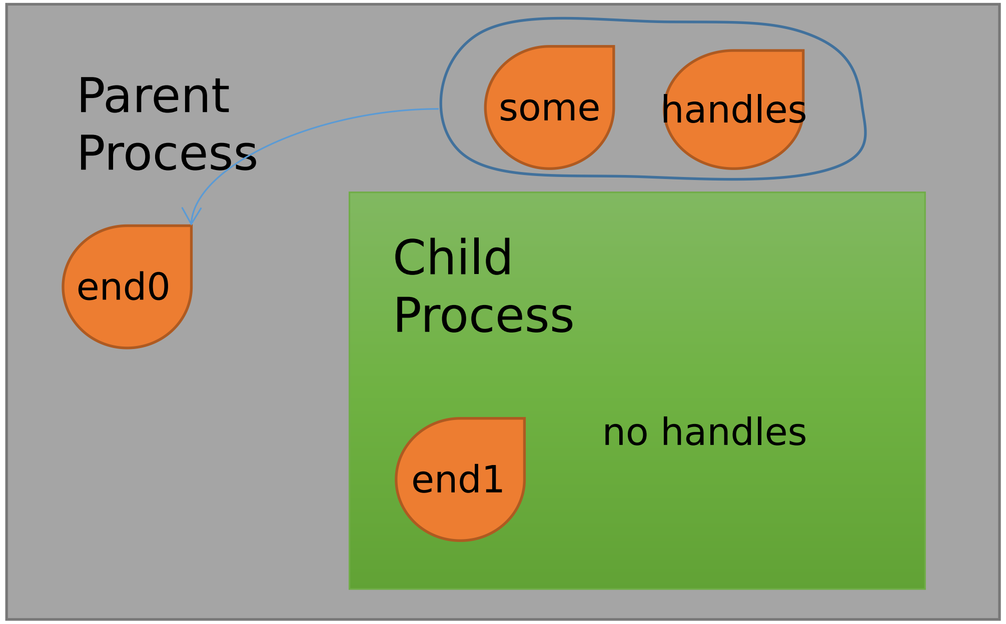
典型用法

- 只给一个handle：



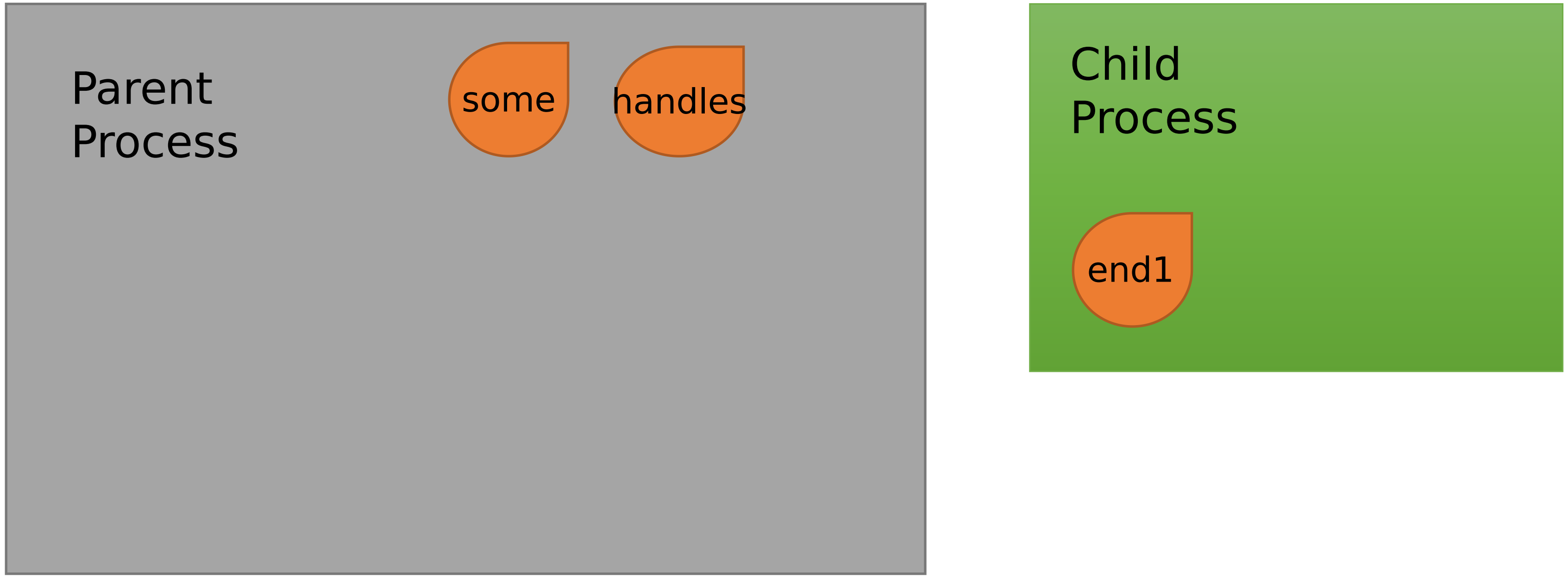
典型用法

- 剩下的交给channel来解决：



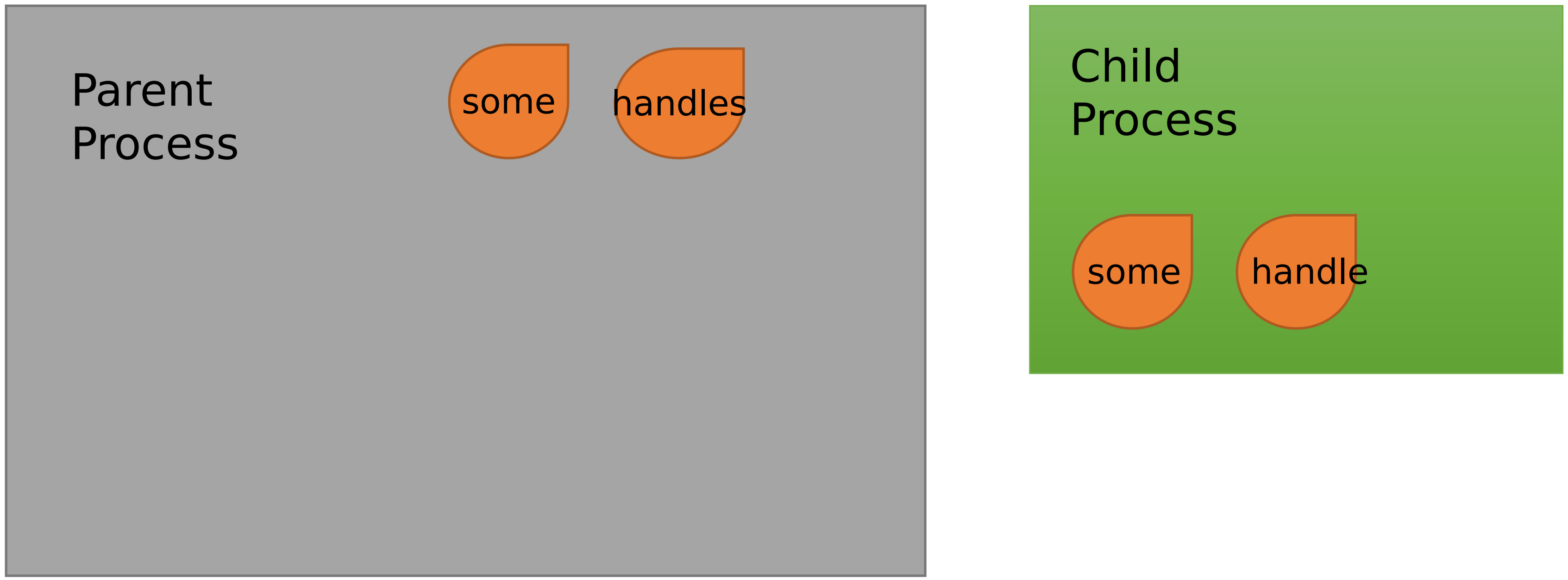
典型用法

- 丢弃end0（孩子长大了）：



典型用法

- 丢弃end0（孩子长大了）：



相关系统调用

- channel_write():
  - 将信息/handle写入到channel中

相关系统调用

- channel_write():
  - 将信息/handle写入到channel中
- channel_read() :
  - 从channel中读取信息

相关系统调用

- channel_write():
  - 将信息/handle写入到channel中
- channel_read() :
  - 从channel中读取信息
- channel_wait_one() :
  - 让channel对象等待某个signal长达deadline的时间

相关系统调用

- channel_write():
  - 将信息/handle写入到channel中
- channel_read() :
  - 从channel中读取信息
- channel_wait_one() :
  - 让channel对象等待某个signal长达deadline的时间
- channel_call() :
  - write() + wait_one() + read()

个人进展

- 外部kernel.elf：
  - kernel.elf -> kernel.bin -> (KASLR)支持的image.bin -> zircon.zbi
  - 修改gn脚本，让我们能够指定kernel.elf的位置
  - 尝试用rust代码解析zbi格式的文件，初步成功
  - 遇到困难：原有kernel.elf的start.S有点复杂，不容易重用
    - 各类call语句，后续调用链的逻辑关系未完全清楚
    - 各地址未完全清楚

- 分析IPC：
  - 目前只分析了Channel的相关系统调用

start.S

```
    // move_fixups_and_zero_bss copied the fixup code to _end.
    // It expects %rdi to contain the actual runtime address of __code_start.
    lea __code_start(%rip), %rdi
    call _end   // 这里跳转到fixup
    // The fixup code won't be used again, so the memory can be reused now.

    /* reload the gdtr after relocations as it relies on relocated VAs */
    lgdt _temp_gdtr(%rip)

    // Set %gs.base to &bp_percpu.  It's statically initialized
    // with kernel_unsafe_sp set, so after this it's safe to call
    // into C code that might use safe-stack and/or stack-protector.
    lea bp_percpu(%rip), %rax
    mov %rax, %rdx
    shr $32, %rdx
    mov $X86_MSR_IA32_GS_BASE, %ecx
    wrmsr

    /* set up the idt */
    // 设置中断向量表
    lea _idt_startup(%rip), %rdi
    call idt_setup
    call load_startup_idt

    /* assign this core CPU# 0 and initialize its per cpu state */
    xor %edi, %edi
    call x86_init_percpu

    // Fill the stack canary with a random value as early as possible.
    // This isn't done in x86_init_percpu because the hw_rng_get_entropy
    // call would make it eligible for stack-guard checking itself.  But
    // %gs is not set up yet in the prologue of the function, so it would
    // crash if it tried to use the stack-guard.
    call choose_stack_guard
```