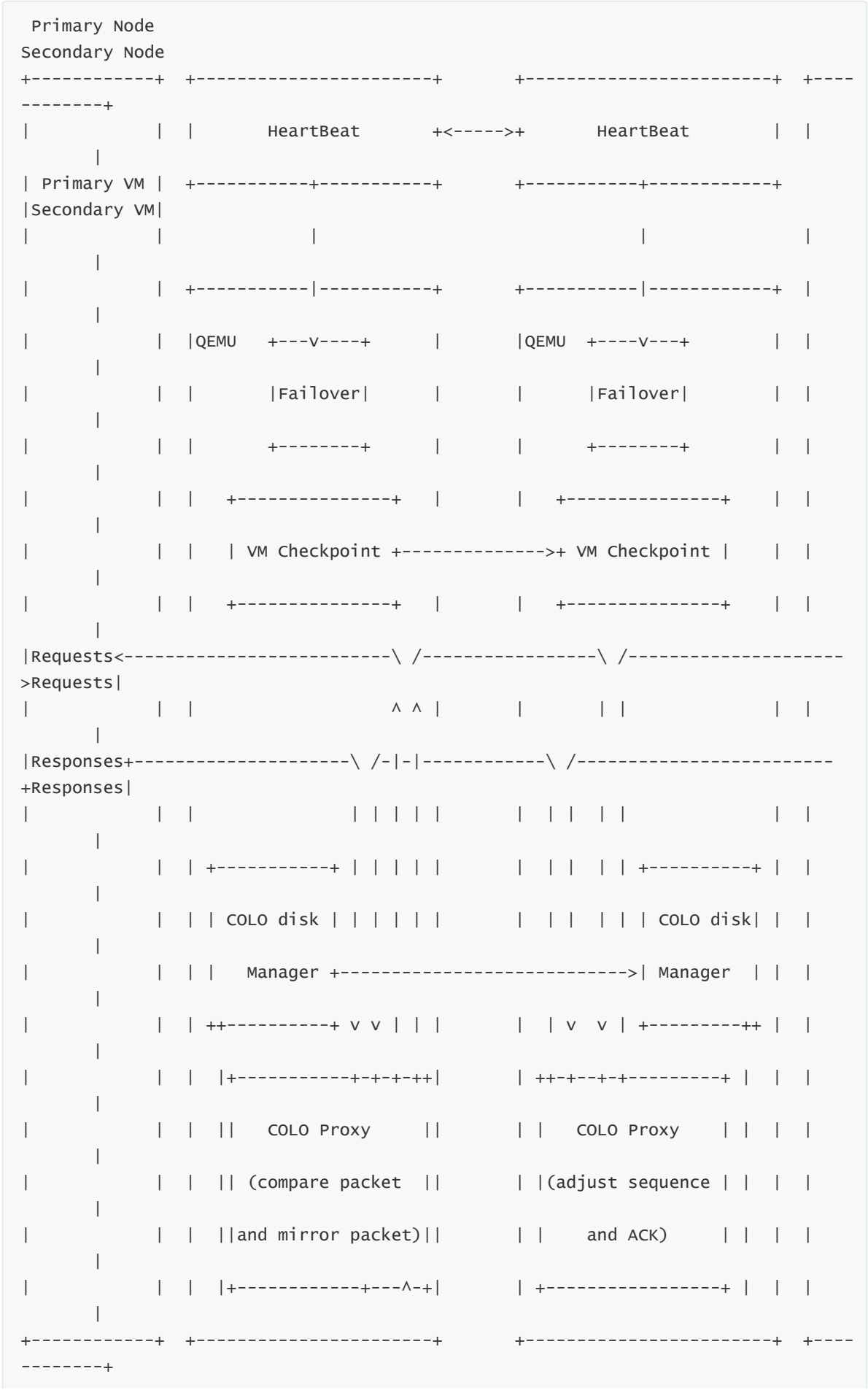
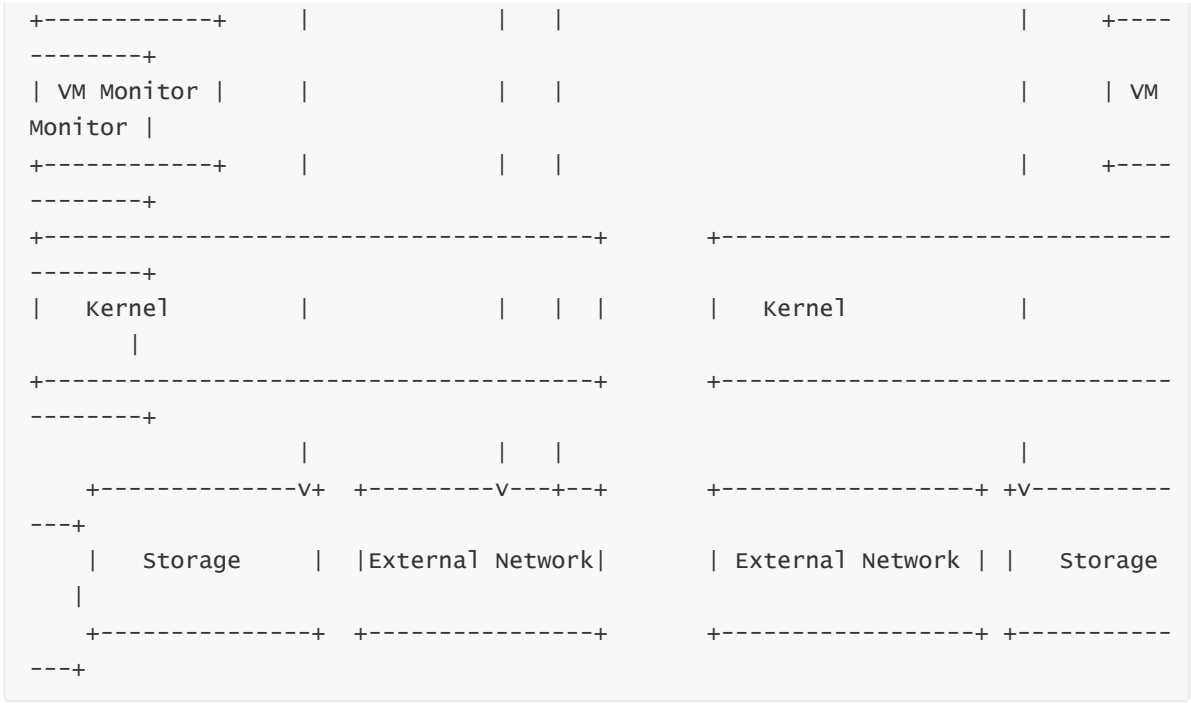
Migration and COLO





1. 初始化

`qemu_init()`
1. `migration_object_init()`

- 初始化迁移传入对象，无论是否使用。

2. `blk_mig_init()`

- 初始化块设备迁移，包括块设备状态的初始化，块设备迁移列表初始化，和锁的初始化

3. `ram_mig_init()`

- `XBZRLE` 算法锁的初始化

4. `dirty_bitmap_mig_init()`

- 脏内存页面标记初始化

2. Migrate调用栈

1. 主节点

`hmp_migrate()` 迁移部分

- `qmp_migrate()` ./migration/migration.c
  - 获取初始的 `MigrationState` 对象
- `migrate_get_current()` ./migration/migration.c，确定 migrate 对象状态，并返回此对象
- `migrate_prepare()` ./migration/migration.c，做迁移前的准备工作，返回 `true` 证明准备完成继续迁移，返回 `false` 直接`return``
- `socket_start_outgoing_migration()` ./migration/socket.c，用 `TCP` 通讯，这里解析了socket参数，并进行函数调用
- `socket_start_outgoing_migration_internal()` ./migration/socket.c，这是内部函数，主要创建了 `qio_channel` 对象，并进行连接
  - `qio_channel_socket_connect_async()`

    - `exec_start_outgoing_migration()` ./migration/exec.c，用`shell`通讯，最终会用`qio_channel`维护通讯
    - `migration_channel_connect()` ./migration/channel.c
    - `fd_start_outgoing_migration()` ./migration/fd.c，用`fd`通讯，最终会用`qio_channel`维护通讯
    - `migration_channel_connect()` ./migration/channel.c
    - `error_set() migrate_set_state() block_cleanup_parameters()` 出错处理

`migration_channel_connect()` IO 部分，./migration/channel.c 绑定`QIOChannel`和`MigrationState`

- `migrate_fd_connect()` ./migration/migration.c 设置一些参数，状态和通知函数，针对`PreCopy`和`PostCopy`操作有不同
- `qemu_thread_create()` ./migration/migration.c 创建新线程处理迁移的IO操作

`migration_thread()` 迁移部分专用的线程 ./migration/migration.c

- `qemu_clock_get_ms()` ./include/qemu/timer.h 拿到开始时间
- `rcu_register_thread()` ./util/rcu.c 注册这个线程到链表
- `object_ref()` .qom/object.c，增加这个迁移对象的引用次数
- `update_iteration_initial_status()` ./migration/migration.c，更新传入的`MigrationState`状态，防止计算错误
- `qemu_savevm_state_header()` ./migration/savevm.c，储存`QEMUFile`头状态，挂载一些头信息
- `qemu_savevm_start_setup()` ./migration/savevm.c，存储`QEMUFile`头状态和项目状态
- `migrate_set_state()` ./migration/migration.c，设置迁移状态
    - 由`MIGRATION_STATUS_SETUP`到`MIGRATION_STATUS_ACTIVE`
- `migrate_is_active()` ./migration/migration.c，判断迁移对象是否还在存活，是迁移过程while循环的一部分
    - 判断`MIGRATION_STATUS_ACTIVE`状态
- `migration_iteration_run()` ./migration/migration.c，主要迭代工作的控制函数
    - `qemu_savevm_state_pending()` ./migration/savevm.c，循环遍历每一个发送条目，调用不同回调发送
        - `save_live_pending()` ./include/migration/register.h，实际回调发送函数
    - `migration_completion()` ./migration/migration.c，剩余数据已经足够一次发送完毕
        - `migration_maybe_pause()` ./migration/migration.c，申请停止源虚拟机
        - `qemu_savevm_state_complete_percopy()` ./migration/savevm.c 调用回调和错误处理
            - `sace_live_complete_percopy()` ./include/migration/register.h，实际回调函数
        - `migrate_colo_enable()` ./migration/migration.c
            - 这里会检查若未开启colo则设置状态`MIGRATION_STATUS_COMPLETED`
- `migration_iteration_finish()` ./migration/migration.c，做一些清理工作
    - 检测若即`MIGRATION_STATUS_ACTIVE`状态还不可以`migrate_colo_enable`会报错，因为上一步若`migrate_colo_enable() False`已经设置为`MIGRATION_STATUS_COMPLETED`
- `object_unref(OBJECT(s))` .qom/object.c，减少这个迁移对象的引用次数
- `rcu_unregister_thread()` ./util/rcu.c，解除注册这个线程到链表

2. 从节点g

`hmp_migrate_incoming`

- `qmp_migrate_incoming` ./migration/migration.c
  - `qemu_start_incoming_migration` ./migration/migration/c 设置从节点状态 `MIGRATION_STATUS_SETUP`
  - `xx_steart_incoming_migration` ./migration/socket.c 设置状态和初始化监听程序
    - `qio_net_listener_set_name` ./io/net-listener.c 设置监听状态，设置回调函数
    - `migration_channel_process_incoming` ./migration/channel.c 创建新 `migration incoming` 对象，设置回调函数

`migration_ioc_process_incoming` 主工作函数

- `migration_incoming_setup` ./migration/migration.c 设置 `incoming` 状态
- `migration_incoming_process` ./migration/migration.c 创建 `Coroutine` 对象，注册 `process_incoming_migration_c` 函数
  - `process_incoming_migration_co` ./migration/migration.c 协程回调函数，设置状态为 `MIGRATION_STATUS_ACTIVE` ,注册 `colo_process_incoming_thread` 线程
    - `trace_process_incoming_migration_co_postcopy_end_main` **这个协程在哪没找到**
    - `qemu_loadvm_state` ./migration/savevm.c 加载 `vm` 状态
    - `migration_incoming_colo_enabled` ./migration/migration.c 判断是否需要 `COLO`
    - `qemu_bh_schedule` ./util/async.c 添加回调到调度器

3. COLO调用栈

1. 主节点

`migraion_iteration_finish()`

- `migrate_start_colo_process` ./migration/colo.c 调用 `colo` 的函数
  - `qemu_event_init` ./util/qemu-thread-posix.c 初始化 `colo_checkpoint_event` 事件
  - `time_new_ms` ./include/qemu/timer.h 设置 `colo_delay_timer` 和 `check_point` 回调函数 `colo_checkpoint_notify`
  - `migrate_set_state` ./migration/migration.c 设置从状态 `MIGRATION_STATUS_ACTIVE` 到 `MIGRATION_STATUS_COLO`
  - `colo_process_checkoutpoint` ./migration/colo.h `checkout_point` 工作函数

`colo_process_checkoutpoint()`

- `qemu_clock_get_ms` ./include/qemu/timer.h 获取当前时间
- `get_colo_mode` ./migration/colo.c 获得colo状态，包括 `COLO_MODE_PRIMARY` ，`COLO_MODE_SECONDART` 和 `COLO_MODE_NONE` ，这里提供一个信息，就是只有**主节点**才能进行 `checkout_point`
- `failover_init_state` ./migration/colo-failover.c 初始化 `FAILOVER_STATUS_NONE`
- `qemu_file_get_return_path` ./migration/qemu-file.c 得到输入参数文件
- `colo_compara_register_notifier` ./net/colo-compare.c 注册收包比较器
- `colo_receive_check_message` ./migration/colo.c 等待附属节点完成 `VM` 加载，并且进入 `COLO` 状态
- `ifdef CONFIG_REPLACTION` ./replication.c 这个宏必须定义，作用未知，这个函数目前也不知道作用

- `vm_start` ./softmmu/cpus.c 启动`VM`
- `qemu_event_wait` ./qemu-thread-posix.c 等待`colo_checkoutpoint_event`事件的发生，这里可能会有其他线程改变`MIGRATION_STATUS`
- `colo_do_checkpoint_transaction` 实际执行`checkpoint`的函数，这个函数在一个while循环里，在执行之前还要确认`MIGRATION_STATUS_CLOL`状态

`colo_do_checkpoint_transaction()`

- `colo_send_message` ./migration/colo.c 发送`COLO_MESSAGE_CHECKPOINT_REQUEST`状态
- `colo_receive_check_message` ./migration/colo.c 接受信息并且检查`COLO_MESSAGE_CHECKPOINT_REPLY`状态，至此为止，双方的状态应该都是等待`COLO`发生
- `qio_channel_io_seek` ./io/channel.c 清空`channel-buffer`
- `vm_stop_force_state` ./softmmu/cpus.c 强制转换状态为`RUN_STATE_COLO`
- `colo_send_message` ./migration/colo.c 发送`COLO_MESSAGE_VNSTATE_SEND`状态
- `qemu_save_device_state` ./migration/savevm.c 存储设备信息至buffer
- `qemu_save_vm_live_State` ./migration/savevm.c 存储活动信息，
  - **TODO:** 可能需要增加一个超时断开机制，防止阻塞
- `colo_send_message_value` ./migration/colo.c 得到`VMstate data size`在附属节点的大小
- `qemu_put_buffer` ./migration.qemu-file.c 将`QEMUFIle`转换为`char buffer`
- `colo_receive_check_message` ./migration/colo.c 接受并检查`COLO_MESSAGE_VMSTATE_RECEIVED`状态
- `colo_receive_check_message` ./mirgation/colo.c `COLO_MESSAGE_VMSTATE_LOADED` 附属节点load完成状态

2.从节点

`colo_process_incoming_thread`

- `colo_send_message` ./migration/colo.c 发送`COLO_MESSAGE_CHECKPOINT_READY`状态
- `colo_wait_handle_message` ./migration/colo.c 接受并处理信息
- `colo_incoming_process_checkpoint`./migration/colo.c 只有在接收到了`COLO_MESSAGE_CHECKPOINT_REQUEST`状态时进入，并发送`COLO_MESSAGE_CHECKPOINT_REPLY`状态
- `colo_receive_check_message` ./migration/colo.c 等待`COLO_MESSAGE_VMSTATE_SEND`状态
- `cpu_synchronize_all_state` ./siftmmu/cpus.c 同步cpu状态
- `qemu_load_vm_state_main` ./migration/savevm.c 加载全部状态
- `colo_recive_message_value` ./migration/colo.c 接收`COLO_MESSAGE_VMSTATE_SIZE`状态，并在后续发送实际值
- `colo_send_message` ./migration/colo.c 发送附属节点状态
- `qemu_load_device_state` ./migration/savevm.c 同步设备状态
- `colo_send_message` ./migration/colo.c 发送`COLO_MESSAGE_VMSTATUS_LOADED`状态

4. 函数细节

`migrate_prepare()` ./migration/migration.c

- 传入过程中会设置`bool resume` ，在设置过`True` 的情况下会导致特殊情况

- 这里好像是说`Postcopy recovery`不能很好的和`release-ram`一起工作，因为这可能会导致网络的发送缓冲区页面丢失，但是幸运的事`release-ram`被设计源主机和目的主机在同一物理服务器上工作，所以应该是没问题的。
- 当调用`migrate_release_ram()`返回`True`时，此函数返回`True`，否则返回`False`
- 传入过程中会设置`bool resume`，在设置过`False`的情况下\
- 会检查`migrate`对象状态，并且初始化此对象，初始化`ram_counters`返回`True`
  `migrate_fd_connect()`./migration/migration.c

`migrate_thread()`./migration/migration.c

`migration_iteration_run()`./migration/migration.c 迭代的关键逻辑

5. 结构体细节

1. 内存相关

```
/**
 * AddressSpace: describes a mapping of addresses to #MemoryRegion objects
 * 描述MMIO和 #MemoryRegion 之间的映射
 */
struct AddressSpace {
    /* private: */
    struct rcu_head rcu;
    char *name;
    MemoryRegion *root; /* MemoryRegion 无向图的根节点 */

    /* Accessed via RCU.  */
    struct FlatView *current_map;

    int ioeventfd_nb;
    struct MemoryRegionIoeventfd *ioeventfds;
    QTAILQ_HEAD(, MemoryListener) listeners;
    QTAILQ_ENTRY(AddressSpace) address_spaces_link;
};
```

- 表示虚拟机能够访问的所有地址
- 和`MemoryRegion *root`对应

```
/** MemoryRegion:
 *
 * A struct representing a memory region.
 */
struct MemoryRegion {
    Object parent_obj;

    /* private: */

    /* The following fields should fit in a cache line */
    bool romd_mode;
    bool ram;
    bool subpage;
    bool readonly; /* For RAM regions */
    bool nonvolatile;
    bool rom_device;
    bool flush_coalesced_mmio;
```

```
    bool global_locking;
    uint8_t dirty_log_mask;
    bool is_iommu;
    RAMBlock *ram_block; /* 实际分配的物理内存 */
    Object *owner;

    const MemoryRegionOps *ops; /* 封装了一组回调函数，在对此结构体进行操作的时候会用到
*/
    void *opaque;
    MemoryRegion *container; /* 此节点的上一级 */
    Int128 size;
    hwaddr addr; /* 虚拟机中的物理地址 */
    void (*destructor)(MemoryRegion *mr);
    uint64_t align; /* 对齐作用 */
    bool terminates; /* 是否是叶子节点 */
    bool ram_device;
    bool enabled;
    bool warning_printed; /* For reservations */
    uint8_t vga_logging_count;
    MemoryRegion *alias;
    hwaddr alias_offset;
    int32_t priority; /* 优先级，在多个对象地址重复时，谁的大谁就会被虚拟机可见 */
    QTAILQ_HEAD(, MemoryRegion) subregions; /* 子节点 */
    QTAILQ_ENTRY(MemoryRegion) subregions_link; /* 兄弟节点 */
    QTAILQ_HEAD(, CoalescedMemoryRange) coalesced;
    const char *name;
    unsigned ioeventfd_nb;
    MemoryRegionIoeventfd *ioeventfds;
    };
```

- 整个内存的模拟通过`MemoryRegion`构成的无环图完成，图的**叶子**结点是**实际**分配给**虚拟机**的**物理内存**或者**MMIO**，**中间**节点表示**内存总线**，其他不关联`AddressSpace`的`MemoryRegion`是**内存控制器**