使用 QMP 命令和结构体记录 checkpoint 过程相关信息

设置 QMP json对象

1. 在 ./qapi/migration.json 中增加结构体对象和命令

```
{ 'struct': 'checkpoint_recorder',
   'data': { 'tot_time': 'int64', 'tot_num': 'int64',
             'init_time': 'int64', 'wait_time': 'int64',
             'avg_time': 'int64', 'stat_time': 'int64',
             'dev_time': 'int64', 'ram_time': 'int64',
             'put_time': 'int64', 'recv_time': 'int64',
             'load_time': 'int64', 'resume_time': 'int64',
             'dev_size': 'int64', 'ram_size': 'int64',
             'info': 'MigrationInfo' } }
```

```
# { "execute": "info_checkpoint" }
# { "execute": "reset_checkpoint" }

{ 'command': 'reset_checkpoint' }
{ 'command': 'info_checkpoint', 'returns': 'checkpoint_recorder' }
```

2. **注意这里复用一部分 MigrationInfo 的代码，用来记录 COLO 过程迁移内存大小（后续可能会继续调整），而且目前应该先不会使用这部分代码，计划先把 get_clock() 记录时间调通过，再修改这部分。**

修改migration.c源码

1. 检查实际结构体声明和函数声明

```
checkpoint_recorder *qmp_info_checkpoint(Error **errp);
struct MigrationInfo;

//../qapi/qapi-types-migration.h
struct checkpoint_recorder {
    int64_t tot_time;
    int64_t tot_num;
    int64_t init_time;
    int64_t wait_time;
    int64_t avg_time;
    int64_t stat_time;
    int64_t dev_time;
    int64_t ram_time;
    int64_t put_time;
    int64_t load_time;
    int64_t resume_time;
    int64_t dev_size;
    int64_t ram_size;
```

```
    MigrationInfo *info;
};
void qapi_free_checkpoint_recorder(checkpoint_recorder *obj);
void qapi_free_MigrationInfo(MigrationInfo *obj);
```

2. 注意这里使用`MigrationInfo`结构体的指针
3. 声明全局指针

```
// ./migration.c

checkpoint_recorder *g_cr;
```

4. 选择初始化位置，我选择了在`migration_iteration_finish()`这个函数中初始化主节点`global checkpoint_recorder`，在`process_incoming_migration_co()`初始化从节点`global chekcpoint_recorder`
5. **释放是可以不用管，构造要单独构造。**

```
// 这里单独构造
info->xbzrle_cache = g_malloc0(sizeof(*info->xbzrle_cache));

MigrationInfo *qmp_query_migrate(Error **errp)
{
    MigrationInfo *info = g_malloc0(sizeof(*info));

    fill_destination_migration_info(info);
    fill_source_migration_info(info);

    // 这里没有析构
    return info;
}
```

这里我一开始想是先析构再构造，后来我觉得还是直接将结构体内部变量置`0`。因为我考虑对于全局变量，内存分区在全局变量区，是不是最好全生命周期只指向一个堆区内存比较好，频繁的构造析构也许可能会产生一些堆区碎片，而且最重要的是万一某变量拿到堆区指针，后面我构造又析构后，某变量可能会段错误。

```
/**
 * @Brief: init global checkpoint recorder point
 * @Author: mengsen
 * @Data: 2020-10-22 15:09:36
**/
 static void checkpoint_recorder_init() {
     if(g_cr != NULL) {
         memset(g_cr->info, 0, sizeof(MigrationInfo));
         memset(g_cr, 0, 8 * sizeof(int64_t));
       } else {
         g_cr = g_malloc0(sizeof(checkpoint_recorder));
     }
     return;
}
```

我这里仍然释放结构体

```
/**
 * @Brief: destroy global checkpoint_recorder
 * @Param: [void]
 * @Return [void]
 * @Author: mengsen
 * @Date: 2020-10-23 10:56:11
**/
static void checkpoint_recorder_destroy(void) {
    if(g_cr != NULL) {
        qapi_free_MigrationInfo(g_cr->info);
    }
    return;
}
```

这里复用了一部分`MigrationInfo`的代码

```
/**
 * @Brief: fill global checkpoint_recorder to temporary checkpoint_recorder,
 * and that temporary used lazy loading
 * @Param: cr [checkpoint_recorder *] temporary checkpoint_recorder pointer
 * @Return: [void]
 * @Author: mengsen
 * @Date: 2020-10-23 10:40:22
**/
static void fill_checkpoint_recorder(checkpoint_recorder *cr){
    cr->tot_time = g_cr->tot_time;
    cr->tot_num = g_cr->tot_num;
    cr->init_time = g_cr->init_time;
    cr->wait_time = g_cr->wait_time / cr->tot_num;
    cr->avg_time = cr->tot_time / cr->tot_num;
    cr->dev_time = g_cr->dev_time / cr->tot_num;
    cr->ram_time = g_cr->ram_time / cr->tot_num;
    cr->vmstate_time = g_cr->vmstate_time / cr->tot_num;
    cr->dev_size = g_cr->dev_size / cr->tot_num;
    cr->ram_size = g_cr->ram_size / cr->tot_num;
    fill_destination_migration_info(cr->info);
    fill_source_migration_info(cr->info);
    return;
}
```

`qmp`接口部分，这样每次除了`checkpoint_recorder`的内存需要我管理，其他的都不需要我管理了。

```
   /**
    * @Brief: qmp interface
    * @Param: errp [Error **] error handle
    * @Return: [checkpoint_recorder *]
    * @Author: mengsen
    * @Date: 2020-10-22 15:26:43
   **/
   checkpoint_recorder *qmp_info_checkpoint(Error **errp) {
       checkpoint_recorder *cr = g_malloc0(sizeof(checkpoint_recorder));
        cr->info = g_malloc0(sizeof(MigrationInfo));
        fill_checkpoint_recorder(cr);
       return cr;
```

```
}
```

修改`colo.c`源码

1. 声明全局变量

```
extern checkpoint *g_cr;
```

2. 一些添加位置

    1. 主节点流程图

```
migrate_start_colo_process(){
    g_cr->init_time = get_clock();  // init_time 开始点
    colo_process_checkpot();
        g_cr->init_time = g_cr->init_time - get_clock(); // init_time 结束点
        while() {
            temp_begin = get_clock();   // wait_time 开始点
            qemu_sem_wait();
            temp_end = get_clock();     // wait_time 结束点

            g_cr->wait_time += temp_end - temp_begin;

            temp_begin = get_clock();   // tot_time 开始点
            colo_do_checkpoint_transaction();
            temp_end = get_clock();     // tot_time 结束点

            g_cr->tot_time += temp_end - temp_begin;
            ++g_cr->tot_num;            // 总次数加1
        }
}
colo_do_checkpoint_transcaction(){
    temp_begin = get_clock(); // stat_time 开始点
    send_message(REQUEST);
    receive_message(REPLY);
    reset_buffer();
    change_state("run", "stop");
    nofity_compare_event();
    set_block_enable();
    maybe_replication_do_checkpoint();
    colo_send_message(VMSTATE_SEND);
    temp_end = get_clock();     // stat_time 结束点
    g_cr->stat_time += temp_end - temp_begin;

    temp_begin = get_clock();   // device_time 开始点
    qemu_save_device_state();
    temp_end = get_clock();     // device_time 结束点
    g_cr->dev_time += temp_end - temp_begin;

    temp_begin_time = get_clock();  // ram_time 开始点
    qemu_savevm_live_state(s->to_dst_file);
    qemu_fflush(fb);
    temp_end_time = get_clock();    // ram_time 结束点
    g_cr->ram_time += temp_end_time - temp_begin_time;
```

```
    temp_begin = get_clock();       // put_time 开始点
    colo_send_message_value(s->to_dst_file, COLO_MESSAGE_VMSTATE_SIZE,
                            bioc->usage, &local_err);
    qemu_put_buffer(s->to_dst_file, bioc->data, bioc->usage);
    qemu_fflush(s->to_dst_file);
    temp_end = get_clock(); // put_time 结束点
    g_cr->put_time += temp_end - temp_begin;

    temp_begin = get_clock();       // recv_time 开始点
    colo_receive_check_message(s->rp_state.from_dst_file,
                               COLO_MESSAGE_VMSTATE_RECEIVED,
&local_err);
    temp_end = get_clock(); // recv_time 结束点
    g_cr->put_time += temp_end - temp_begin;

    temp_begin = get_clock();       // load_time 开始点
    colo_receive_check_message(s->rp_state.from_dst_file,
                               COLO_MESSAGE_VMSTATE_LOADED, &local_err);
    temp_end = get_clock();         // load_time 结束点
    g_cr->load_time += temp_end - temp_begin;

    temp_begin = get_clock();       // resume_time 开始点
    qemu_mutex_lock_iothread();
    vm_start();
    qemu_mutex_unlock_iothread();
    trace_colo_vm_state_change("stop", "run");
    temp_end = get_clock();         // resume_time 结束点
    g_cr->resume_time += temp_end - temp_begin;
}
```

2. 从节点流程图

```
colo_process_incoming_thread(){
    g_cr->init_time = get_clock();
    g_cr->init_time = g_cr->init_time - get_clock();
    while(){
        colo_wait_handle_message()
    }
}
colo_wait_handle_message(){
    temp_begin = get_clock();   // wait_time 开始点
    receive_msg(REQUEST);
    temp_end = get_clock();     // wait_time 结束点
    g_cr->wait_time += temp_end - temp_begin;
    temp_begin = get_clock();   // tot_time 开始点
    colo_incoming_process_checkpoint();
    temp_end = get_clock();     // tot_time 结束点
    g_cr->tot_time += temp_end - temp_begin;
    ++g_cr->tot_num;    // 增加总次数
}

colo_incoming_process_checkpoint(){
    temp_begin = get_clock(); // stat_time 开始点
    send(REPLAY)
    receive(VMSTATE);
```

```
    temp_end = get_clock(); // stat_time 结束点
    g_cr->stat_time += temp_begin - temp_end;

    temp_begin_time = get_clock();  // ram_time 开始点
    qemu_mutex_lock_iothread();
    cpu_synchronize_all_pre_loadvm();
    ret = qemu_loadvm_state_main(mis->from_src_file, mis);
    qemu_mutex_unlock_iothread();
    temp_end_time = get_clock();    // ram_time 结束点
    g_cr->ram_time += temp_end_time - temp_begin_time;

    temp_begin_time = get_clock();  // put_time 开始点
    value = colo_receive_message_value(mis->from_src_file,
                             COLO_MESSAGE_VMSTATE_SIZE, &local_err);
    if (value > bioc->capacity) {
        bioc->capacity = value;
        bioc->data = g_realloc(bioc->data, bioc->capacity);
    }
    total_size = qemu_get_buffer(mis->from_src_file, bioc->data, value);
    temp_end_time = get_clock();    // put_time 结束点
    g_cr->put_time += temp_end_time - temp_begin_time;

    temp_begin_time = get_clock();  // recv_time 开始点
    colo_send_message(mis->to_src_file, COLO_MESSAGE_VMSTATE_RECEIVED,
                  &local_err);
    temp_end_time = get_clock();    // recv_time 结束点
    g_cr->recv_time += temp_end_time - temp_begin_time;

    temp_begin_time = get_clock();  // load_time 开始点
    qemu_mutex_lock_iothread();
    vmstate_loading = true;
    ret = qemu_load_device_state();
    vmstate_loading = false;
    vm_start();
    qemu_mutex_unlock_iothread();

    if (failover_get_state() == FAILOVER_STATUS_RELAUNCH) {
        failover_set_state(FAILOVER_STATUS_RELAUNCH,
                           FAILOVER_STATUS_NONE);
        failover_request_active(NULL);
        return;
    }

    colo_send_message(mis->to_src_file, COLO_MESSAGE_VMSTATE_LOADED,
                  &local_err);
    temp_end_time = get_clock();    // load_time 结束点
    g_cr->load_time += temp_end_time - temp_begin_time;

}
```

测试结果（不要启动就开始测试）

1核2G 编译内核

```
{"return": {"tot_time": 4295509854, "wait_time": 17651743242, "init_time":
140781582, "put_time": 26804, "stat_time": 31396893, "resume_time": 251641,
"load_time": 597324359, "dev_size": 16919, "dev_time": 595551, "ram_size": 0,
"avg_time": 715918309, "recv_time": 1947, "tot_num": 6, "ram_time": 86318844,
"info": {"expected-downtime": 1277, "status": "colo", "setup-time": 42, "total-
time": 334026, "ram": {"total": 2165121024, "postcopy-requests": 0, "dirty-sync-
count": 36, "multifd-bytes": 0, "pages-per-second": 1760, "page-size": 4096,
"remaining": 0, "mbps": 53.822902, "transferred": 1677952185, "duplicate":
553349, "dirty-pages-rate": 208, "skipped": 0, "normal-bytes": 1669709824,
"normal": 407644}}}}
```


2核2G 编译内核


```
{"return": {"tot_time": 4851209013, "wait_time": 16078759529, "init_time":
139707848, "put_time": 28805, "stat_time": 45556566, "resume_time": 296554,
"load_time": 607641014, "dev_size": 19130, "dev_time": 704995, "ram_size": 0,
"avg_time": 808534835, "recv_time": 2245, "tot_num": 6, "ram_time": 154303144,
"info": {"expected-downtime": 160, "status": "colo", "setup-time": 41, "total-
time": 257755, "ram": {"total": 2165121024, "postcopy-requests": 0, "dirty-sync-
count": 35, "multifd-bytes": 0, "pages-per-second": 290100, "page-size": 4096,
"remaining": 0, "mbps": 9172.72136, "transferred": 1458167517, "duplicate":
554359, "dirty-pages-rate": 2218, "skipped": 0, "normal-bytes": 1450344448,
"normal": 354088}}}}
```


4核2G 编译内核


```
{"return": {"tot_time": 5088784800, "wait_time": 18981378149, "init_time":
153395374, "put_time": 27636, "stat_time": 25471275, "resume_time": 513525,
"load_time": 678927630, "dev_size": 23552, "dev_time": 759800, "ram_size": 0,
"avg_time": 1017756960, "recv_time": 2538, "tot_num": 5, "ram_time": 312054565,
"info": {"expected-downtime": 316, "status": "colo", "setup-time": 41, "total-
time": 562359, "ram": {"total": 2165121024, "postcopy-requests": 0, "dirty-sync-
count": 48, "multifd-bytes": 0, "pages-per-second": 291670, "page-size": 4096,
"remaining": 0, "mbps": 9351.04824, "transferred": 3892962593, "duplicate":
580622, "dirty-pages-rate": 4528, "skipped": 0, "normal-bytes": 3880157184,
"normal": 947304}}}}
```


8核2G 编译内核


```
{"return": {"tot_time": 15926515359, "wait_time": 18735688737, "init_time":
137946849, "put_time": 40874, "stat_time": 70852667, "resume_time": 945973,
"load_time": 774318843, "dev_size": 32396, "dev_time": 1293836, "ram_size": 0,
"avg_time": 1327209613, "recv_time": 2315, "tot_num": 12, "ram_time": 479752230,
"info": {"expected-downtime": 652, "status": "colo", "setup-time": 41, "total-
time": 332218, "ram": {"total": 2165121024, "postcopy-requests": 0, "dirty-sync-
count": 29, "multifd-bytes": 0, "pages-per-second": 265740, "page-size": 4096,
"remaining": 0, "mbps": 8485.30008, "transferred": 6897523672, "duplicate":
606899, "dirty-pages-rate": 8314, "skipped": 0, "normal-bytes": 6878625792,
"normal": 1679352}}}}
```


16核2G 编译内核

{"return": {"tot_time": 16775957311, "wait_time": 18293712245, "init_time": 122877963, "put_time": 35059, "stat_time": 101646924, "resume_time": 1654671, "load_time": 961243842, "dev_size": 50084, "dev_time": 2267404, "ram_size": 0, "avg_time": 1863995256, "recv_time": 2007, "tot_num": 9, "ram_time": 797144114, "info": {"expected-downtime": 957, "status": "colo", "setup-time": 41, "total-time": 407745, "ram": {"total": 2165121024, "postcopy-requests": 0, "dirty-sync-count": 26, "multifd-bytes": 0, "pages-per-second": 287670, "page-size": 4096, "remaining": 0, "mbps": 9130.61304, "transferred": 10294216076, "duplicate": 651182, "dirty-pages-rate": 13499, "skipped": 0, "normal-bytes": 10268299264, "normal": 2506909}}}}
{"timestamp"

4核4G 编译内核

{"return": {"tot_time": 8907004496, "wait_time": 18614704199, "init_time": 168837085, "put_time": 26383, "stat_time": 73947335, "resume_time": 710327, "load_time": 1138494420, "dev_size": 23552, "dev_time": 928426, "ram_size": 0, "avg_time": 1484500749, "recv_time": 1985, "tot_num": 6, "ram_time": 270390054, "info": {"expected-downtime": 345, "status": "colo", "setup-time": 73, "total-time": 417721, "ram": {"total": 4312604672, "postcopy-requests": 0, "dirty-sync-count": 29, "multifd-bytes": 0, "pages-per-second": 286570, "page-size": 4096, "remaining": 0, "mbps": 9175.74264, "transferred": 4411979030, "duplicate": 1119195, "dirty-pages-rate": 4820, "skipped": 0, "normal-bytes": 4393324544, "normal": 1072589}}}}

8核4G 编译内核

{"return": {"tot_time": 21011424810, "wait_time": 17466532986, "init_time": 96024858, "put_time": 37645, "stat_time": 83197156, "resume_time": 1169692, "load_time": 1294232898, "dev_size": 32396, "dev_time": 1368028, "ram_size": 0, "avg_time": 1910129528, "recv_time": 2864, "tot_num": 11, "ram_time": 530121020, "info": {"expected-downtime": 564, "status": "colo", "setup-time": 73, "total-time": 620221, "ram": {"total": 4312604672, "postcopy-requests": 0, "dirty-sync-count": 35, "multifd-bytes": 0, "pages-per-second": 294930, "page-size": 4096, "remaining": 0, "mbps": 9235.96776, "transferred": 14227831696, "duplicate": 1204424, "dirty-pages-rate": 7958, "skipped": 0, "normal-bytes": 14189277184, "normal": 3464179}}}}

16核4G 编译内核

{"return": {"tot_time": 20120123124, "wait_time": 16100320899, "init_time": 135084717, "put_time": 48167, "stat_time": 106906023, "resume_time": 1940474, "load_time": 1376798788, "dev_size": 50084, "dev_time": 2160751, "ram_size": 0, "avg_time": 2235569236, "recv_time": 0, "tot_num": 9, "ram_time": 747711836, "info": {"expected-downtime": 918, "status": "colo", "setup-time": 70, "total-time": 295369, "ram": {"total": 4312604672, "postcopy-requests": 0, "dirty-sync-count": 23, "multifd-bytes": 0, "pages-per-second": 294440, "page-size": 4096, "remaining": 0, "mbps": 9382.69728, "transferred": 7827294013, "duplicate": 1161056, "dirty-pages-rate": 13144, "skipped": 0, "normal-bytes": 7801606144, "normal": 1904689}}}}

{"return": {"tot_time": 16801194630, "wait_time": 17782170954, "init_time": 135084717, "put_time": 43762, "stat_time": 93411179, "resume_time": 3392813, "load_time": 1395134074, "dev_size": 50084, "dev_time": 1915789, "ram_size": 0, "avg_time": 2400170661, "recv_time": 3049, "tot_num": 7, "ram_time": 906270930, "info": {"expected-downtime": 1037, "status": "colo", "setup-time": 70, "total-time": 518508, "ram": {"total": 4312604672, "postcopy-requests": 0, "dirty-sync-count": 34, "multifd-bytes": 0, "pages-per-second": 293420, "page-size": 4096, "remaining": 0, "mbps": 9256.82544, "transferred": 15981245264, "duplicate": 1246396, "dirty-pages-rate": 14582, "skipped": 0, "normal-bytes": 15938895872, "normal": 3891332}}}}

```
# { "execute": "info_checkpoint" }
# { "execute": "reset_checkpoint" }
```