共享内存设计实现方式及使用说明

1. 文档说明

该文档主要描述 Commander 机器人上位机软件架构中的共享内存机制的设计实现方式及使用说明。

2. 共享内存的设计思路及实现方式

共享内存机制主要用于实现 Commander 上位机软件中大数据量的进程间交互，利用共享内存高效、低延时的特性，可以减轻 FastRTPS 的通信负担，同时提升整体软件工程的效率。

此实现主要基于 Boost::interprocess 库，所以需要在使用前安装 Boost 类库，安装方式见官网：https://www.boost.org/doc/libs/1_70_0/doc/html/quickbook/install.html

为了便于了解共享内存机制，可以阅读 boost 的官方文档中 Boost::interprocess 部分中的 shared memory 部分和 mutexes 部分。

https://www.boost.org/doc/libs/1_70_0/doc/html/interprocess.html

Boost 库提供的共享内存包括普通共享内存和托管共享内存(managed shared memory)，接下来的叙述以 managed shared memory 的方式实现。

为了实现多进程访问同一段共享内存，需要对进程间进行同步操作，Boost 提供了很多种进程间同步的方式，以下叙述采用 named_sharable_mutex 实现。

具体实现方式见代码，如下为 shmwrite.cpp 的示例代码，具体实现过程参见代码注释。该程序主要完成的工作包括：

在共享内存放置一个 int[1000]的数组，将其初始化，然后获取 mutex 锁，将 int[88]的数据加 1，解锁，睡眠 1 秒钟，如此循环。

此例程主要示例共享内存的写操作。

```cpp
#include <boost/interprocess/sync/sharable_lock.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <boost/interprocess/shared_memory_object.hpp>

int main()
{
    using namespace boost::interprocess;
    //删除共享内存和mutex，防止在初始化的时候已经存在了之前未删除的共享内存和mutex
    shared_memory_object::remove("hello");
    shared_memory_object::remove("sharedmutex");

    //构造一个共享内存，id叫做“hello”，大小是1024×1024字节
    managed_shared_memory managed_shm(open_or_create, "hello", 1024*1024);
    //构造一个named_sharable_mutex，名字叫做sharedmutex，由内核管理
    named_sharable_mutex named_shared_mtx(open_or_create, "sharedmutex");
    //在共享内存中构造一个int[1000]的对象，名字叫做“Integer”
    int *i = managed_shm.construct<int>("Integer")[1000](99);
    //p.first是“Integer”在共享内存的地址，p.second是1000
    std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
    //成功
    if (p.first)
    {
        //Interger数据初始化
        for(int i=0; i<1000; i++)
            *(p.first + i) = i;
    }
    else
        return 1;

    for(;;)
    {
        // sharable_lock<named_sharable_mutex> mylock(named_shared_mtx);
        //将"sharedmutex"上独占锁，当lock成功后，任何其他进程均不能获取"sharedmutex"
        scoped_lock<named_sharable_mutex> mylock(named_shared_mtx);
        //上锁成功
        if(mylock)
        {
            //在独占锁住"sharedmutex"的情况下，写“Integer”中的数据
            *(p.first + 88) += 1;
            std::cout << "write: " << *(p.first + 88) << std::endl;
            //手动解锁，让其他进程可以访问数据
            mylock.unlock();
        }
        sleep(1);
    }
    return 0;
}
```

下一段为 shmwrite2.cpp 的代码:

程序主要实现功能为：与 shmwrite 通过抢占的形式操作共享内存中的数据。

```cpp
#include <iostream>
#include <boost/interprocess/sync/named_sharable_mutex.hpp>
#include <boost/interprocess/sync/sharable_lock.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <boost/interprocess/shared_memory_object.hpp>

int main()
{
    using namespace boost::interprocess;
    //创建或打开名为hello的共享内存
    managed_shared_memory managed_shm(open_or_create, "hello", 1024*1024);
    //打开名为sharedmutex的mutex
    named_sharable_mutex named_shared_mtx(open_only, "sharedmutex");

    //找到名为Integer的int[1000]的对象，将其映射至本进程
    std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
    if (p.first)
    { //映射成功
        for(;;)
        {
            //将"sharedmutex"上独占锁，当lock成功后，任何其他进程均不能获取"sharedmutex"
            scoped_lock<named_sharable_mutex> mylock(named_shared_mtx);
            if(mylock)
            {//上锁成功
                //在独占锁住"sharedmutex"的情况下，写“Integer”中的数据
                *(p.first + 88) += 1;
                std::cout << "write: " << *(p.first + 88) << std::endl;
                //手动解锁，让其他进程可以访问数据
                mylock.unlock();
            }
            sleep(1);

        }
    }

    return 0;
}
```

下面代码是 shmread.cpp 中的代码，主要实现功能为，在以 sharable 的形式获得 mutex 锁的前提下，在共享内存中找到名为 Integer 的 int[1000]的数据，读取其中的 int[88]并将其打印出来。

```cpp
#include <iostream>
#include <boost/interprocess/sync/named_sharable_mutex.hpp>
#include <boost/interprocess/sync/sharable_lock.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <boost/interprocess/shared_memory_object.hpp>

int main()
{
    using namespace boost::interprocess;

    managed_shared_memory managed_shm(open_or_create, "hello", 1024*1024);
    named_sharable_mutex named_shared_mtx(open_only, "sharedmutex");

    std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
    if (p.first)
    {
        for(;;)
        {
            // named_shared_mtx.lock_sharable();
            //sharable_lock后的mutex不是独占式的，多个进程可以同时对mutex加sharable_lock
            //但是当有一个进程对mutex加了独占锁（scoped_lock或lock），则下面语句阻塞
            sharable_lock<named_sharable_mutex> mylock(named_shared_mtx);
            if(mylock)
            {
                std::cout << "read: " << *(p.first + 88) << std::endl;
                //手动解锁
                mylock.unlock();
            }
            sleep(1);
            // named_shared_mtx.unlock_sharable();
        }
    }

    return 0;
}
```

重点注意 sharable_lock 的使用!

3. 此外，共享内存的实现方式还可以参考 Boost 的官方文档:

https://www.boost.org/doc/libs/1_70_0/doc/html/interprocess/synchronization_mechanisms.html#interprocess.synchronization_mechanisms.mutexes

例程 2 的实现方式为普通共享内存+interprocess_mutex 的实现方式，最大的不同在于，此 interprocess_mutex 也位于构建的共享内存中。

两种实现方式均可。

4. 相关 tips

(1) 使用 boost 库的共享内存相关函数，在编译时需要加上-lrt 选项;

(2) 引用 boost 库的 cc 文件在使用 makefile 编译的时候，需要加入-lboost_system 等相关的链接库

(3) 查看 SystemV 共享内存用 shell 命令 ipcs，查看 POSIX 的共享内存，到/dev/shm 目录下查看，以文件的形式存在