共享内存通信框架设计文档

1. 设计目标

本框架旨在实现高效的进程间数据传输，支持：

- C++程序之间的数据传输（生产者-消费者模式）
- 支持图像和点云数据的实时共享
- 高性能的零拷贝数据传输
- 可靠的数据一致性保证

2. 架构设计

2.1 核心组件

1. **共享内存管理器**
   - `ShareMemoryManager`类（支持生产者和消费者模式）
   - 同一个类可以作为生产者或消费者使用
   - 支持异步回调方式接收数据
2. **内存布局**

```
+------------------+
|  Header (固定大小) |
+------------------+
|  Data (可变大小)   |
+------------------+
```

3. **同步机制**
   - 互斥锁：确保独占访问
   - 状态标志：Empty、Writing、Ready、Error

2.2 数据结构

共享内存头部 (SharedMemoryHeader)

```
struct SharedMemoryHeader {
    uint32_t Magic;          // 魔数 (0x12345678)
    uint32_t Status;         // 状态
    uint32_t DataSize;       // 数据大小
    uint32_t Checksum;       // 校验和
    uint32_t FrameId;        // 帧ID
    uint32_t DataType;       // 数据类型
    uint32_t Width;          // 宽度/点数
    uint32_t Height;         // 高度/点大小
```

```
    uint32_t Reserved;       // 通道数/维度
    char ErrorMsg[128];      // 错误信息
};
```

2.3 数据类型支持

1. **图像数据 (DataType = 0)**

```
width    = 图像宽度（像素）
height   = 图像高度（像素）
channels = 图像通道数（1=灰度图, 3=RGB, 4=RGBA）
数据大小  = width * height * channels
```

2. **点云数据 (DataType = 1)**

```
width      = 点的数量
height     = 每个分量的字节大小（通常为sizeof(float)）
dimensions = 点的维度（3=XYZ, 6=XYZRGB等）
数据大小    = width * height * dimensions
```

3. 工作流程

3.1 写入流程（生产者）

1. 获取互斥锁
2. 检查内存状态（必须为Empty）
3. 设置状态为Writing
4. 写入数据和元信息
5. 计算校验和
6. 设置状态为Ready
7. 释放互斥锁

3.2 读取流程（消费者）

1. 获取互斥锁
2. 检查内存状态（必须为Ready）
3. 验证魔数
4. 读取数据
5. 验证校验和
6. 设置状态为Empty
7. 释放互斥锁

4. 使用方法

4.1 生产者模式

```
// 创建生产者
SharedMemory::ShareMemoryManager producer("TestSharedMemory", 1024 * 1024 * 10);
if (!producer.Initialize()) {
    // 处理错误
}

// 写入图像数据
uint32_t width = 640;
uint32_t height = 480;
uint32_t channels = 3; // RGB图像
std::vector<uint8_t> imageData = GenerateImage(width, height, channels);
if (!producer.WriteData(imageData.data(), 0, width, height, channels)) {
    // 处理错误
}

// 写入点云数据
uint32_t pointCount = 1000;
uint32_t dimensions = 3; // XYZ点云
std::vector<uint8_t> pointCloudData = GeneratePointCloud(pointCount);
if (!producer.WriteData(pointCloudData.data(), 1, pointCount, sizeof(float), 0,
dimensions)) {
    // 处理错误
}
```

4.2 消费者模式

```
// 创建消费者
SharedMemory::ShareMemoryManager consumer("TestSharedMemory", 1024 * 1024 * 10);
if (!consumer.Initialize()) {
    // 处理错误
}

// 设置数据接收回调
consumer.SetDataReceivedCallback([](const uint8_t* data, size_t size,
    uint32_t dataType, uint32_t width, uint32_t height) {
    if (dataType == 0) {
        // 处理图像数据
        ProcessImage(data, width, height);
    } else {
        // 处理点云数据
        ProcessPointCloud(data, width);
    }
});

// 启动监听线程
consumer.StartMonitoring();

// ... 主程序逻辑 ...
```

```
// 停止监听
consumer.StopMonitoring();
```

5. 错误处理

5.1 主要错误类型

1. 初始化错误
    - 互斥锁创建失败
    - 共享内存创建失败
    - 内存映射失败
2. 运行时错误
    - 内存未初始化
    - 数据大小超限
    - 校验和不匹配
    - 状态不一致

5.2 错误恢复

1. 使用`ClearMemory()`方法重置共享内存状态
2. 重新初始化共享内存
3. 记录详细日志用于问题诊断

6. 性能优化

1. 零拷贝传输：直接在共享内存中读写数据
2. 校验和计算优化：使用简单快速的算法
3. 状态检查优化：避免不必要的日志记录
4. 互斥锁超时设置：防止死锁

7. 注意事项

1. 确保生产者和消费者使用相同的共享内存名称
2. 正确处理程序异常退出的情况
3. 定期检查和清理共享内存状态
4. 合理设置缓冲区大小，避免内存浪费
5. 注意32位和64位程序的兼容性

8. 调试方法

1. 使用日志文件
    - producer_log.txt：生产者日志
    - consumer_log.txt：消费者日志
2. 状态监控

- 使用`LogStatus()`方法查看当前状态
- 检查帧ID连续性
- 监控数据大小变化

3. 常见问题排查

- 检查魔数是否匹配
- 验证数据大小是否合理
- 确认互斥锁是否正常释放
- 查看校验和计算是否正确

9. 示例程序

9.1 生产者-消费者测试程序

```
int main()
{
    // 创建生产者和消费者
    ShareMemoryManager producer("TestSharedMemory", 1024 * 1024 * 10);
    ShareMemoryManager consumer("TestSharedMemory", 1024 * 1024 * 10);

    // 初始化
    producer.Initialize();
    consumer.Initialize();

    // 设置数据接收回调
    consumer.SetDataReceivedCallback([](const uint8_t* data, size_t size,
        uint32_t dataType, uint32_t width, uint32_t height) {
        std::cout << "\n[Consumer] Received data:"
                  << "\n - Type: " << (dataType == 0 ? "Image" : "PointCloud")
                  << "\n - Size: " << size << " bytes"
                  << "\n - Width: " << width
                  << "\n - Height: " << height
                  << "\n - First byte: 0x" << std::hex << (int)data[0]
                  << std::dec << std::endl;
    });

    // 启动消费者监听
    consumer.StartMonitoring();

    // 交替发送图像和点云数据
    bool isImage = true;
    for (int i = 0; i < 10; ++i) {
        if (isImage) {
            // 发送RGB图像
            auto imageData = GenerateTestImage(640, 480, 3);
            producer.WriteData(imageData.data(), 0, 640, 480, 3);
        } else {
            // 发送XYZ点云
            auto pointCloud = GenerateTestPointCloud(1000);
            producer.WriteData(pointCloud.data(), 1, 1000, sizeof(float), 0, 3);
        }
```

```
            isImage = !isImage;
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
        }

        // 停止监听
        consumer.StopMonitoring();
    }
```

9.2 性能测试结果

数据类型	大小	传输延迟	CPU占用
RGB图像 (640x480)	921.6KB	<1ms	<5%
点云 (1000点)	12KB	<1ms	❤️%