2019-10-10 CPU占用率的优化

一、CPU占用率

1.1 原因

在while(true)中，线程sleep()时间过短，会导致线程上下文切换过于频繁，数据频繁压栈出栈，从而使CPU占用率急剧上升

1.2 优化方法

1. sleep() 时间加大，使上下文切换变缓
   治标不治本
2. 容器中的大数据，使用指针，将数据存在堆中而非栈中，提高数据访问效率（请参考本文的第二节）
3. 使用 不可重入互斥锁 与 条件变量，使线程挂起，而非一直在后台刷新（请参考本文的第三节）

二、容器数据

2.1 STL中的容器

- 如：vector*unordered_map*map/queue等等所有容器
- 容器只管理数据，不管理内存
- 只有耗尽一定程度内存，容器才会统一释放（与编译器有关）

2.2 优化方法

- 养成习惯，需要容器存取大数据的，必须用指针
  1. 因数据较大，将其存在堆中，提高存取效率
  2. 更好地方便内存管理
  3. 可以使用shared_ptr/unique_ptr，使其自动释放内存
     强烈建议自己写，提高自己使用内存的能力，但为了项目考虑，不要自己写，直接用智能指针
- 不使用指针
  1. 导致内存不可控
  2. 频繁复制
  3. 数据使用频繁时，隐藏的压栈/出栈操作过多

三、条件变量的使用

3.1 目的

- 主要目的 使当前线程挂起、恢复，而非while(true)一直在后台跑，能大大减轻CPU占用率
- 使外部线程随时自由控制该线程

3.2 头文件

```
#include <condition_variable>
```

3.3 示例一：类的线程安全与协作方法

3.3.1 声明

```
std::condition_variable m_CompleteCond;      // 监控任务的结束情况
std::mutex m_completeMutex;                  // 任务完成的互斥锁
```

3.3.2 用法

封装两个方法：

1. 将当前线程 挂起监听
2. 通知 当前挂起的线程 恢复，恢复当前线程继续处理

```
336 // ------------------------------------------------------------------------------------------
337 // 功能函数
338 // ------------------------------------------------------------------------------------------
339
340 // 等待信息，完成或者出现异常，以便继续
341 void CAutomatedGuidedVehicle::WaitSignalToContinue() {
342     src::severity_logger< severity_level > slg;
343
344     BOOST_LOG_SEV(slg, normal) << "agvNo: " << m_AgvNo << " - WaitSignalToContinue locked!!!";
345
346     std::unique_lock<std::mutex> locker(m_completeMutex);
347     m_CompleteCond.wait(locker, [this] {
348         return m_AgvCompleteTask;
349     });
350
351     BOOST_LOG_SEV(slg, normal) << "agvNo: " << m_AgvNo << " - WaitSignalToContinue unlocked!!!";
352 }
353
354 // 启动条件变量监听
355 void CAutomatedGuidedVehicle::CondNotifyOne() {
356     src::severity_logger< severity_level > slg;
357     m_CompleteCond.notify_one();
358     BOOST_LOG_SEV(slg, normal) << "agvNo: " << m_AgvNo << " - CondNotifyOne!!!";
359
360 }
```

3.4 示例二：监控数据队列 的方法

3.4.1 头文件中声明

```
std::condition_variable m_QueueCond;          // 队列条件变量
```

3.4.2 用法

1. 外部socket往回调函数压入数据时，使用条件变量通知挂起的线程

```
238 // 任务队列：总站->基站
239 void CAGVsPathPlanning::SetTartgetPos(const std::vector<std::string> targetPos) {
240     src::severity_logger< severity_level > slg;
241 
242     if (targetPos.size() == 0) {
243         BOOST_LOG_SEV(slg, error) << "SetTartgetPos: targetPos is null。";
244         return;
245     }
246 
247     std::lock_guard<std::mutex> locker(m_TaskQueueMutex);
248     for (auto e : targetPos) {
249         if (m_TargetPos.size() == 0) {              // 如果数据被取光，则释放内存
250             ClearQueue();
251         }
252         m_TargetPos.push(e);
253     }
254     m_QueueCond.notify_one();                       // 通知线程，有数据压入
255 
256     BOOST_LOG_SEV(slg, normal) << "SetTartgetPos: m_targetPos success...";
257 }
```

2. 数据处理线程等待数据队列，将当前线程挂起，减少CPU占用，防止后台一直刷数据

```
869     // 该线程负责获取任务
870     m_TaskThread = new std::thread([this] {
871         while (true)
872         {
873             src::severity_logger< severity_level > slg;
874 
875             std::string tmp;
876             std::unique_lock<std::mutex> locker(m_TaskQueueMutex);
877 
878             if (m_TargetPos.empty()) {
879                 ClearQueue();
880                 BOOST_LOG_SEV(slg, normal) << "MISSION COMPLETE";
881             }
882 
883             m_QueueCond.wait(locker, [this] {
884                 return !(m_TargetPos.empty()); });
885 
886             tmp = m_TargetPos.front();
887             m_TargetPos.pop();
888 
889             locker.unlock();
890 
891             BOOST_LOG_SEV(slg, normal) << "task: tartget position is " << tmp;
892 
893             // 负责任务分发
```

#读书笔记/知识点

#邓波/2019/10