ORB-SLAM2系统的是如何初始化的?

orb slam

ORB-SLAM2系统初始化可以总结做了两件事:

1、把读取到的图片数据给到跟踪线程

2、把保存跟踪生成的轨迹到文件(可视化轨迹看前面的文章)

源码解读

```
/**
定义ORB-SLAM2系统的主线程结构,其他各模块都由这里开始被调用
*/
#include <string>
#include <opencv2/core/core.hpp>
#include "Tracking.h"
#include "FrameDrawer.h"
#include "MapDrawer.h"
#include "Map.h"
#include "LocalMapping.h"
#include "LoopClosing.h"
#include "KeyFrameDatabase.h"
#include "ORBVocabulary.h"
#include "Viewer.h"
#include "System.h"                 // 主进程的实现文件
#include "Converter.h"              // 常见转换

#include <thread>                   // 多线程
#include <pangolin/pangolin.h>      // 可视化
#include <iomanip>                  // cin,cout

namespace ORB_SLAM2
{
/**
 * @param strVocFile
 * @param strSettingsFile
 * @param sensor
 * @param bUseViewer
 * 传感器类型 eSensor mSensor
   枚举类型用于表示本系统所使用的传感器类型
   enum eSensor{
       MONOCULAR=0,
       STEREO=1,
       RGBD=2
    };
   传感器类型
   eSensor mSensor;
   类变量前缀 m
   指针类型变量前缀 p
   进程前缀 t

 * Initialize the SLAM system. It launches the Local Mapping, Loop Closing and Viewer th
   构造函数初始化整个系统
 * 创建可视化指针用Pangolin显示地图,相机位姿 Viewer* mpViewer
 * mbActivateLocalizationMode bool 记录模式的变量
 * mbDeactivateLocalizationMode bool 记录模式的变量
 */
System::System(const string &strVocFile, //指定ORB字典文件的路径
               const string &strSettingsFile, //指定配置文件的路径
               const eSensor sensor, //指定配置文件的路径
               const bool bUseViewer): //指定是否使用可视化界面
                     mSensor(sensor), //初始化传感器类型
                     mpViewer(static_cast<Viewer*>(NULL)), //可视化界面
                     mbReset(false), //复位标志
                     mbActivateLocalizationMode(false), //定位建图模式
                     mbDeactivateLocalizationMode(false) //定位模式
{
    // 输出当前传感器类型
    cout << "Input sensor was set to: ";

    if(mSensor==MONOCULAR)
        cout << "Monocular" << endl;
    else if(mSensor==STEREO)
        cout << "Stereo" << endl;
    else if(mSensor==RGBD)
        cout << "RGB-D" << endl;

    // Check settings file
    cv::FileStorage fsSettings(strSettingsFile.c_str(),cv::FileStorage::READ);
    // 如果打开失败, 输出调试信息
    if(!fsSettings.isOpened()){
       cerr << "Failed to open settings file at: " << strSettingsFile << endl;
       exit(-1);
    }

    // Load ORB Vocabulary
    cout << endl << "Loading ORB Vocabulary. This could take a while..." << endl;


    /**
     * ORB vocabulary used for place recognition and feature matching.
     * 定义存储ORB字典的指针
     * ORBVocabulary* mpVocabulary;
     * 建立一个新的ORB字典
     * typedef DBoW2::TemplatedVocabulary<DBoW2::FORB::TDescriptor, DBoW2::FORB> ORBVoca
     */
    mpVocabulary = new ORBVocabulary();

    // 获取字典加载状态
    bool bVocLoad = mpVocabulary->loadFromTextFile(strVocFile);
    // 如果加载失败 输出调试信息 然后退出
    if(!bVocLoad){
        cerr << "Wrong path to vocabulary. " << endl;
        cerr << "Falied to open at: " << strVocFile << endl;
        exit(-1);
    }
    // 加载成功
    cout << "Vocabulary loaded!" << endl << endl;

    //Create KeyFrame Database
    /**
     * KeyFrame database for place recognition (relocalization and loop detection).
     * 定义存储关键帧数据库的指针, 用于重定位和回环检测
     * KeyFrameDatabase* mpKeyFrameDatabase;
     * 构造函数
     * KeyFrameDatabase::KeyFrameDatabase (const ORBVocabulary &voc):
     */
    mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);

    // Create the Map
    /**
     * Map structure that stores the pointers to all KeyFrames and MapPoints.
     * 定义存储地图的指针, 地图包括关键帧和地图点
     * Map* mpMap;
     * 地图构造函数
     * Map::Map():mnMaxKFid(0)
     */
    mpMap = new Map();

    // Create Drawers. These are used by the Viewer
```

```
    // 这里的帧绘制器和地图绘制器将会被可视化的Viewer所使用
    /**
     * 创建帧显示指针
     * FrameDrawer* mpFrameDrawer;
     * FrameDrawer构造函数
     * FrameDrawer::FrameDrawer(Map* pMap):mpMap(pMap)
     */
    mpFrameDrawer = new FrameDrawer(mpMap);

    /**
     * 创建地图显示指针
     * MapDrawer* mpMapDrawer;
     * MapDrawer构造函数
     * MapDrawer::MapDrawer(Map* pMap, const string &strSettingPath):mpMap(pMap)
     */
    mpMapDrawer = new MapDrawer(mpMap, strSettingsFile);


    /**
     * Tracker. It receives a frame and computes the associated camera pose.
     * It also decides when to insert a new keyframe, create some new MapPoints and
     * performs relocalization if tracking fails.
     * 跟踪接收帧和帧的位姿跟踪，负责创建关键帧,新地图点,跟踪失败后的重定位.
     * Tracking* mpTracker;
     * 在主线程中初始化跟踪线程
     * Initialize the Tracking thread
     * (it will live in the main thread of execution, the one that called this construct
     */
    mpTracker = new Tracking(this,
                             mpVocabulary,              // 字典
                             mpFrameDrawer,             // 帧可视化
                             mpMapDrawer,               // 地图可视化
                             mpMap,                     // 地图
                             mpKeyFrameDatabase,        // 关键帧地图
                             strSettingsFile,           // 设置文件路径
                             mSensor);                  // 传感器类型


    /**
     * Local Mapper. It manages the local map and performs local bundle adjustment.
     * 创建局部建图指针器管理局部地图并且进行局部BA
     * LocalMapping* mpLocalMapper;
     * 初始化局部建图线程并运行
     * Initialize the Local Mapping thread and launch
     * LocalMapping构造函数
     * LocalMapping::LocalMapping(Map *pMap, const float bMonocular):
     */
    mpLocalMapper = new LocalMapping(mpMap,mSensor==MONOCULAR);
    /**
     * 局部建图线程
     * System threads: Local Mapping, Loop Closing, Viewer.
     * The Tracking thread "lives" in the main execution thread that creates the System
     * 系统在运行主进程跟踪的同时创建局部建图,回环检测,可视化线程
     * std::thread* mptLocalMapping;
     * std::thread* mptLoopClosing;
     * std::thread* mptViewer;
     */
    mptLocalMapping = new thread(&ORB_SLAM2::LocalMapping::Run,mpLocalMapper);

    /**
     * Loop Closer. It searches loops with every new keyframe. If there is a loop it per
     * a pose graph optimization and full bundle adjustment (in a new thread) afterwards
     * 创建回环检测指针. 搜索回环执行位姿图优化并且开启新线程进行全局BA
     * LoopClosing* mpLoopCloser;
     * Initialize the Loop Closing thread and launchiomanip
     * LoopClosing 构造函数
     * LoopClosing::LoopClosing(Map *pMap, KeyFrameDatabase *pDB, ORBVocabulary *pVoc, c
     */
    mpLoopCloser = new LoopClosing(mpMap,mpKeyFrameDatabase,mpVocabulary,mSensor!=MONOCU

    /**
     * System threads: Local Mapping, Loop Closing, Viewer.
     * The Tracking thread "lives" in the main execution thread that creates the System
     * 系统在运行主进程跟踪的同时创建局部建图,回环检测,可视化线程
     * std::thread* mptLocalMapping;
     * std::thread* mptLoopClosing;
     * std::thread* mptViewer;
     * 创建回环检测线程
     */
    mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run,mpLoopCloser);

    // Initialize the Viewer thread and launch
    if(bUseViewer){
        mpViewer = new Viewer(this,mpFrameDrawer,mpMapDrawer,mpTracker,strSettingsFile);
        // 新建viewer线程
        mptViewer = new thread(&Viewer::Run, mpViewer);
        // 给动跟设置对应的可视化
        mpTracker->SetViewer(mpViewer);
    }

    // Set pointers between threads
    // 设置进程间的指针
    mpTracker->SetLocalMapper(mpLocalMapper);
    mpTracker->SetLoopClosing(mpLoopCloser);

    mpLocalMapper->SetTracker(mpTracker);
    mpLocalMapper->SetLoopCloser(mpLoopCloser);

    mpLoopCloser->SetTracker(mpTracker);
    mpLoopCloser->SetLocalMapper(mpLocalMapper);
}

/**
 * @brief 双目图像跟踪
 * @param imLeft
 * @param imRight
 * @param timestamp
 * @return
 *
 * 针对三种不同类型的传感器设计三种运动跟踪接口
 * 彩色图像为CV_8UC3类型，并且都将会被转换成为灰度图像
 * 跟踪接口返回估计的相机位姿，如果跟踪失败则返回NULL
 * Proccess the given stereo frame. Images must be synchronized(同步) and rectified(校正)
 * Input images: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to grayscale.
 * Returns the camera pose (empty if tracking fails).
 *
 * const cv::Mat &imLeft,          //左目图像
 * const cv::Mat &imRight,         //右目图像
 * const double &timestamp);       //时间戳
 */
cv::Mat System::TrackStereo(const cv::Mat &imLeft, const cv::Mat &imRight, const double
    if(mSensor!=STEREO){
        cerr << "ERROR: you called TrackStereo but input sensor was not set to STEREO."
        exit(-1);
    }

    // Check mode change
    {
        unique_lock<mutex> lock(mMutexMode);
        // 只开启定位模式
        if(mbActivateLocalizationMode){
            mpLocalMapper->RequestStop();// 局部建图停止
            // Wait until Local Mapping has effectively stopped
            while(!mpLocalMapper->isStopped()){
                usleep(1000);
            }
            mpTracker->InformOnlyTracking(true);// 只跟踪
            mbActivateLocalizationMode = false;// 设置定位模式状态
        }
        // 定位模式和建图模式同时开启
        if(mbDeactivateLocalizationMode){
            mpTracker->InformOnlyTracking(false);   //开启地图构建和跟踪模式
            mpLocalMapper->Release();// 局部建图工作
```

```
            mbDeactivateLocalizationMode = false;// 设置定位模式状态
        }
    }

    // Check reset, 检查是否有复位的操作
    // Reset flag 复位标志
    // std::mutex mMutexReset;
    // bool mbReset;
    {
        unique_lock<mutex> lock(mMutexReset);//上锁
        //是否有复位请求
        if(mbReset){
            mpTracker->Reset(); //有, 跟踪复位
            mbReset = false; // 清除标志
        }
    }

    /**
     * Tracker. It receives a frame and computes the associated camera pose.
     * It also decides when to insert a new keyframe, create some new MapPoints and
     * performs relocalization if tracking fails.
     * 跟踪接收帧和帧的位姿跟踪, 负责创建关键帧,新地图点,跟踪失败后的重定位.
     * Tracking* mpTracker;
     *
     * 输入左右目图像, 可以为RGB、BGR、RGBA、GRAY
     * 1、将图像转为mImGray和imGrayRight并初始化mCurrentFrame
     * 2、进行tracking过程
     *  输出世界坐标系到该帧相机坐标系的变换矩阵
     *  cv::Mat Tracking::GrabImageStereo(
     *  const cv::Mat &imRectLeft,      //左侧图像
     *  const cv::Mat &imRectRight,     //右侧图像
     *  const double &timestamp)        //时间戳
     *
     *  用矩阵Tcw来保存估计的相机位姿, GrabImageStereo运动估计函数
     */
    cv::Mat Tcw = mpTracker->GrabImageStereo(imLeft,imRight,timestamp);

    // 运动跟踪状态上锁
    unique_lock<mutex> lock2(mMutexState);

    /**
     * 获取运动跟踪状态
     * Tracker. It receives a frame and computes the associated camera pose.
     * It also decides when to insert a new keyframe, create some new MapPoints and
     * performs relocalization if tracking fails.
     * 跟踪接收帧和帧的位姿跟踪, 负责创建关键帧,新地图点,跟踪失败后的重定位.
     * Tracking* mpTracker;
     */
    mTrackingState = mpTracker->mState;
    // 获取当前帧跟踪到的地图点向量指针
    mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
    // 获取当前帧跟踪到的关键帧特征点向量的指针
    mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;
    // 返回获得的相机运动估计
    return Tcw;
}

/**
 * @brief RGBD图像跟踪
 * @param im
 * @param depthmap
 * @param timestamp
 * @return
 *
 * Process the given rgbd frame. Depthmap must be registered(配准|匹配)to the RGB frame.
 * Input image: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to grayscale.
 * Input depthmap: Float (CV_32F).
 * Returns the camera pose (empty if tracking fails).
 *
 * const cv::Mat &im,                //彩色图像
 * const cv::Mat &depthmap,          //深度图像
 * const double &timestamp);         //时间戳
 */
cv::Mat System::TrackRGBD(const cv::Mat &im, const cv::Mat &depthmap, const double &time
    if(mSensor!=RGBD){
        cerr << "ERROR: you called TrackRGBD but input sensor was not set to RGBD." << e
        exit(-1);
    }

    // Check mode change
    // Change mode flags 记录模式的变量
    // std::mutex mMutexMode;
    // bool mbActivateLocalizationMode;
    // bool mbDeactivateLocalizationMode;
    {
        unique_lock<mutex> lock(mMutexMode);
        if(mbActivateLocalizationMode){
            // Local Mapper. It manages the local map and performs local bundle adjustme
            // 局部建图和局部BA(bundle adjustment)
            mpLocalMapper->RequestStop();

            // Wait until Local Mapping has effectively stopped
            while(!mpLocalMapper->isStopped()){
                usleep(1000);
            }

            mpTracker->InformOnlyTracking(true);
            mbActivateLocalizationMode = false;
        }
        if(mbDeactivateLocalizationMode){
            mpTracker->InformOnlyTracking(false);
            mpLocalMapper->Release();
            mbDeactivateLocalizationMode = false;
        }
    }

    // Check reset 检查复位
    {
    unique_lock<mutex> lock(mMutexReset);
    if(mbReset){
        mpTracker->Reset();
        mbReset = false;
    }
    }


    /**
     * 获得相机位姿的估计
     * cv::Mat Tracking::GrabImageRGBD(
     * const cv::Mat &imRGB,            //彩色图像
     * const cv::Mat &imD,              //深度图像
     * const double &timestamp)         //时间戳
     * 输入左目RGB或RGBA图像和深度图
     * 将图像转为mImGray和imDepth初始化mCurrentFrame
     * 进行tracking过程
     * 输出世界坐标系到该帧相机坐标系的变换矩阵
     */
    cv::Mat Tcw = mpTracker->GrabImageRGBD(im,depthmap,timestamp);

    unique_lock<mutex> lock2(mMutexState);

    // 跟踪状态 eTrackingState mState;
    // Tracking state 记录跟踪状态
    // int mTrackingState;
    mTrackingState = mpTracker->mState;
    /**
     * MapPoints associated to keypoints, NULL pointer if no association.
     * 每个特征点keypoints对应地图点MapPoint.
     * 如果特征点没有对应的地图点,将存储一个空指针
     * std::vector<MapPoint*> mvpMapPoints;
     * Frame mCurrentFrame; 追踪线程的一个Current Frame(当前帧)
     * std::vector<MapPoint*> mTrackedMapPoints;
     * std::vector<cv::KeyPoint> mTrackedKeyPointsUn;
```

```
     */
    mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
    // std::vector<cv::KeyPoint> mvKeysUn; 校正mvKeys后的特征点
    mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;
    return Tcw;
}

/**
 * @brief 单目图像跟踪
 * @param im
 * @param timestamp
 * @return
 *
 * Proccess the given monocular frame
 * Input images: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to grayscale.
 * Returns the camera pose (empty if tracking fails).
 *
 * const cv::Mat &im,            //图像
 * const double &timestamp);    //时间戳
 */
cv::Mat System::TrackMonocular(const cv::Mat &im, const double &timestamp){
    if(mSensor!=MONOCULAR){
        cerr << "ERROR: you called TrackMonocular but input sensor was not set to Monocu
        exit(-1);
    }

    // Check mode change
    {
        // 独占锁，主要是为了mbActivateLocalizationMode和mbDeactivateLocalizationMode不会发生
        unique_lock<mutex> lock(mMutexMode);
        // mbActivateLocalizationMode为true会关闭局部地图线程
        if(mbActivateLocalizationMode){
            mpLocalMapper->RequestStop();
            // Wait until Local Mapping has effectively stopped
            while(!mpLocalMapper->isStopped()){
                usleep(1000);
            }

            // 局部地图关闭以后，只进行追踪的线程，只计算相机的位姿，没有对局部地图进行更新
            // 设置mbOnlyTracking为真
            mpTracker->InformOnlyTracking(true);
            // 关闭线程可以使得别的线程得到更多的资源
            mbActivateLocalizationMode = false;
        }
        // 如果mbDeactivateLocalizationMode是true，局部地图线程就被释放， 关键帧从局部地图中删除
        if(mbDeactivateLocalizationMode){
            mpTracker->InformOnlyTracking(false);
            mpLocalMapper->Release();
            mbDeactivateLocalizationMode = false;
        }
    }

    // Check reset
    {
    unique_lock<mutex> lock(mMutexReset);
    if(mbReset){
        mpTracker->Reset();
        mbReset = false;
    }
    }

    // 获取相机位姿的估计结果
    cv::Mat Tcw = mpTracker->GrabImageMonocular(im,timestamp);
    unique_lock<mutex> lock2(mMutexState);
    mTrackingState = mpTracker->mState;
    mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
    mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;

    return Tcw;
}

/**
 * @brief 按照TUM格式保存相机运行轨迹并保存到指定的文件
 * @param filename 用来存储轨迹的文件名字
 * Save camera trajectory in the TUM RGB-D dataset format.
 * NOTE Only for stereo and RGB-D. This method does not work for monocular.
 * Call first Shutdown()
 * See format details at: http://vision.in.tum.de/data/datasets/rgbd-dataset
 * 保存相机的运动轨迹(TUM数据集的格式，保存轨迹操作要在Shutdown函数之后调用
 */
    void System::SaveTrajectoryTUM(const string &filename){
        cout << endl << "Saving camera trajectory to " << filename << " ..." << endl;
        // 只有在传感器为双目或者RGBD时才可以工作
        if(mSensor==MONOCULAR){
            cerr << "ERROR: SaveTrajectoryTUM cannot be used for monocular." << endl;
            return;
        }

        // 从地图中获取所有的关键帧
        vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
        // 根据关键帧生成的先后顺序(lId)进行排序
        sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

        // Transform all keyframes so that the first keyframe is at the origin.
        // After a loop closure the first keyframe might not be at the origin.
        // 获取第一个关键帧的位姿的逆，并将第一帧作为世界坐标系
        cv::Mat Two = vpKFs[0]->GetPoseInverse();

        // 文件写入的准备工作
        ofstream f;
        f.open(filename.c_str());
        // 输出浮点数的时候不使用科学计数法
        f << fixed;


        /**
         * Frame pose is stored relative to its reference keyframe (which is optimized b
         * We need to get first the keyframe pose and then concatenate the relative tran
         * Frames not localized (tracking failure) are not saved.
         * 之前的帧位姿都是基于其参考关键帧的，现在我们把它恢复
         *
         * For each frame we have a reference keyframe (lRit), the timestamp (lT) and a
         * which is true when tracking failed (lbL).
         * 参考关键帧列表
         */
        list<ORB_SLAM2::KeyFrame*>::iterator lRit = mpTracker->mlpReferences.begin();
        // 所有帧对应的时间戳列表
        list<double>::iterator lT = mpTracker->mlFrameTimes.begin();
        // 每帧的追踪状态组成的列表
        list<bool>::iterator lbL = mpTracker->mlbLost.begin();
        // 对于每一个mlRelativeFramePoses中的帧lit
        for(list<cv::Mat>::iterator lit=mpTracker->mlRelativeFramePoses.begin(),
                    lend=mpTracker->mlRelativeFramePoses.end();
            lit!=lend;
            lit++, lRit++, lT++, lbL++)
        {
            // 如果该帧跟踪失败,进行下一个
            if(*lbL)
                continue;

            // 获取其对应的参考关键帧
            KeyFrame* pKF = *lRit;

            // 变换矩阵的初始化，初始化为一个单位阵
            cv::Mat Trw = cv::Mat::eye(4,4,CV_32F);

            // If the reference keyframe was culled(剔除), traverse the spanning tree to
            // 查看当前使用的参考关键帧是否为bad
            while(pKF->isBad()){
                // 更新关键帧变换矩阵的初始值.
                Trw = Trw*pKF->mTcp;
                // 并且更新到原关键帧的父关键帧
                pKF = pKF->GetParent();
```

```
            }
            // 最后一个Two是原点校正
            // 最终得到的是参考关键帧相对于世界坐标系的变换
            Trw = Trw*pKF->GetPose()*Two;

            // 在此基础上得到相机当前帧相对于世界坐标系的变换
            cv::Mat Tcw = (*lit)*Trw;
            // 然后分解出旋转矩阵
            cv::Mat Rwc = Tcw.rowRange(0,3).colRange(0,3).t();
            // 分解出平移向量
            cv::Mat twc = -Rwc*Tcw.rowRange(0,3).col(3);

            // 用四元数表示旋转
            vector<float> q = Converter::toQuaternion(Rwc);

            // 然后按照给定的格式输出到文件中
            f << setprecision(6) << *lT << " " <<  setprecision(9) << twc.at<float>(0) <
        }

        // 关闭文件
        f.close();
        cout << endl << "trajectory saved!" << endl;
    }

/**
 * 保存关键帧的轨迹
 * @param filename 用来存储轨迹的文件名字
 * Save keyframe poses in the TUM RGB-D dataset format.
 * NOTE This method works for all sensor input.
 * Call first Shutdown()
 * See format details at: http://vision.in.tum.de/data/datasets/rgbd-dataset
 * 保存相机的运动轨迹(TUM数据集的格式，保存轨迹操作要在Shutdown函数之后调用
 */
    void System::SaveKeyFrameTrajectoryTUM(const string &filename){
        cout << endl << "Saving keyframe trajectory to " << filename << " ..." << endl;

        //获取关键帧vector并按照生成时间对其进行排序
        vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
        sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

        // Transform all keyframes so that the first keyframe is at the origin.
        // After a loop closure the first keyframe might not be at the origin.
        // cv::Mat Two = vpKFs[0]->GetPoseInverse();
        ofstream f;
        f.open(filename.c_str());
        f << fixed;

        // 遍历关键帧
        for(size_t i=0; i<vpKFs.size(); i++)
        {
            // 获取该关键帧
            KeyFrame* pKF = vpKFs[i];

            // pKF->SetPose(pKF->GetPose()*Two);

            if(pKF->isBad())
                continue;

            // 抽取旋转部分和平移部分，前者使用四元数表示
            cv::Mat R = pKF->GetRotation().t();
            vector<float> q = Converter::toQuaternion(R);
            cv::Mat t = pKF->GetCameraCenter();
            //按照给定的格式输出到文件中
            f << setprecision(6) << pKF->mTimeStamp << setprecision(7) << " " << t.at<fl
              << " " << q[0] << " " << q[1] << " " << q[2] << " " << q[3] << endl;

        }
        f.close();
        cout << endl << "trajectory saved!" << endl;
    }

/**
 * @brief 按照KITTI数据集的格式将相机的运动轨迹保存到文件中
 * @param filename 用来存储轨迹的文件名字
 * Save camera trajectory in the KITTI dataset format.
 * NOTE Only for stereo and RGB-D. This method does not work for monocular.
 * Call first Shutdown()
 * See format details at: http://www.cvlibs.net/datasets/kitti/eval_odometry.php
 * 保存相机的运动轨迹(以KITTI数据集的格式，保存轨迹操作要在Shutdown函数之后调用
 */
    void System::SaveTrajectoryKITTI(const string &filename){
        cout << endl << "Saving camera trajectory to " << filename << " ..." << endl;
        if(mSensor==MONOCULAR){
            cerr << "ERROR: SaveTrajectoryKITTI cannot be used for monocular." << endl;
            return;
        }
        vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
        sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

        // Transform all keyframes so that the first keyframe is at the origin.
        // After a loop closure the first keyframe might not be at the origin.
        cv::Mat Two = vpKFs[0]->GetPoseInverse();

        ofstream f;
        f.open(filename.c_str());
        f << fixed;

        // Frame pose is stored relative to its reference keyframe (which is optimized b
        // We need to get first the keyframe pose and then concatenate the relative tran
        // Frames not localized (tracking failure) are not saved.

        // For each frame we have a reference keyframe (lRit), the timestamp (lT) and a
        // which is true when tracking failed (lbL).
        list<ORB_SLAM2::KeyFrame*>::iterator lRit = mpTracker->mlpReferences.begin();
        list<double>::iterator lT = mpTracker->mlFrameTimes.begin();
        for(list<cv::Mat>::iterator lit=mpTracker->mlRelativeFramePoses.begin(), lend=mp
        {
            ORB_SLAM2::KeyFrame* pKF = *lRit;

            cv::Mat Trw = cv::Mat::eye(4,4,CV_32F);

            while(pKF->isBad()){
                Trw = Trw*pKF->mTcp;
                pKF = pKF->GetParent();
            }

            Trw = Trw*pKF->GetPose()*Two;

            cv::Mat Tcw = (*lit)*Trw;
            cv::Mat Rwc = Tcw.rowRange(0,3).colRange(0,3).t();
            cv::Mat twc = -Rwc*Tcw.rowRange(0,3).col(3);

            f << setprecision(9) << Rwc.at<float>(0,0) << " " << Rwc.at<float>(0,1)  <<
              Rwc.at<float>(1,0) << " " << Rwc.at<float>(1,1)  << " " << Rwc.at<float>(1
              Rwc.at<float>(2,0) << " " << Rwc.at<float>(2,1)  << " " << Rwc.at<float>(2
        }
        f.close();
        cout << endl << "trajectory saved!" << endl;
    }




/**
 * This stops local mapping thread (map building) and performs only camera tracking.
 * 激活定位模式 局部建图部分不工作载入已经有的地图 仅有跟踪部分工作
 */
void System::ActivateLocalizationMode(){
    unique_lock<mutex> lock(mMutexMode);// 上锁
    mbActivateLocalizationMode = true;    // 设置标志
    }

/**
```

```
 * This resumes local mapping thread and performs SLAM again.
 * 取消只定位模式 局部建图部分工作，跟踪部分工作，
 */
void System::DeactivateLocalizationMode(){
    unique_lock<mutex> lock(mMutexMode);
    mbDeactivateLocalizationMode = true;
}

/**
 * Returns true if there have been a big map change (loop closure, global BA)
 * since last call to this function
 * 记录地图是否改变的变量
 */
bool System::MapChanged(){
    static int n=0;
    //其实整个函数功能实现的重点还是在这个GetLastBigChangeIdx函数上
    int curn = mpMap->GetLastBigChangeIdx();
    if(n<curn){
        n=curn;
        return true;
    }
    else
        return false;
}

/**
 * Reset the system (clear map)
 * 复位SLAM系统就是清空地图
 */
void System::Reset(){
    unique_lock<mutex> lock(mMutexReset);
    mbReset = true;
}

/**
 * All threads will be requested to finish.
 * It waits until all threads have finished.
 * This function must be called before saving the trajectory.
 * 关闭SLAM系统的所有线程，然后保存轨迹到地图.
 */
void System::Shutdown(){
    // 对局部建图线程和回环检测线程发送终止请求
    mpLocalMapper->RequestFinish();
    mpLoopCloser->RequestFinish();
    // 使用可视化窗口
    if(mpViewer){
        // 向可视化窗口发送终止请求
        mpViewer->RequestFinish();
        while(!mpViewer->isFinished())
            usleep(5000);
    }
    // Wait until all thread have effectively stopped
    while(!mpLocalMapper->isFinished() ||
          !mpLoopCloser->isFinished()  ||
          mpLoopCloser->isRunningGBA())
    {
        usleep(5000);
    }

    if(mpViewer)
        // 可视化窗口
        pangolin::BindToContext("ORB-SLAM2: Map Viewer");
}

/**
 * 获取当前帧的跟踪状态
 * Information from most recent processed frame
 * You can call this right after TrackMonocular (or stereo or RGBD)
 * 记录最近的帧的跟踪状态 int GetTrackingState();
 *
 * std::mutex mMutexState;
 *
 * Tracking state 记录跟踪状态 int mTrackingState;
 */
int System::GetTrackingState(){
    unique_lock<mutex> lock(mMutexState);
    return mTrackingState;
}

/**
 * 获取跟踪到的地图点
 * std::mutex mMutexState;
 * std::vector<MapPoint*> mTrackedMapPoints;
 * std::vector<MapPoint*> GetTrackedMapPoints();
 */
vector<MapPoint*> System::GetTrackedMapPoints(){
    unique_lock<mutex> lock(mMutexState);
    return mTrackedMapPoints;
}

/**
 * 获取跟踪到的关键帧上的关键点
 * std::vector<cv::KeyPoint> GetTrackedKeyPointsUn();
 * std::vector<cv::KeyPoint> mTrackedKeyPointsUn;
 *
 * The keypoint is characterized by the 2D position,
 * a point feature found by one of many available keypoint detectors,
 * such as cv::FAST, cv::StarDetector, cv::SURF, cv::SIFT,
 */
vector<cv::KeyPoint> System::GetTrackedKeyPointsUn(){
    unique_lock<mutex> lock(mMutexState);
    return mTrackedKeyPointsUn;
}

} //namespace ORB_SLAM
```